国外计算机科学经典教材

THOMSON

Embedded C Programming and the Atmel AVR

嵌入式C编程与Atmel AVR

Richard Barnett
(美) Larry O'Cull 著
Sarah Cox
周俊杰 等译

清华大学出版社

Embedded C Programming and the Atmel AVR

欢迎进入嵌入式编程和微控制器应用的世界！使用日益流行且价格适宜的Atmel AVR嵌入式控制器作为学习平台，本书是目前可以获得的最佳书籍之一。对于初学者，本书将会是极好的选择。随着对 Atmel AVR RISC 处理器的介绍，读者会立即进入嵌入式 C 语言教程之中。在本书中，读者将学习 C 语言的变量和常量、运算符和表达式、控制语句、指针和数组、存储器类型、预处理指令、实时方法等等。

本书特点

- 强调嵌入式系统编程，包括了大量的示例，本书通过这些示例来引导读者入门，进而学习高级嵌入式 C 编程技术。
- 本书第 4 章完整地介绍了 CodeVision AVR C Compiler，该章提供了关于 IDE 安装与操作以及 Code Wizard Code Generator 的使用的一个清晰的、循序渐进的指导。
- 本书还介绍了外设的使用，如键区、LCD 显示器以及其他常用的嵌入式微控制器相关设备。

关于作者

Richard H. Barnett 是 Purdue 大学的电子工程技术专业的教授。在教学方面，Barnett 博士获得了很多奖项，包括作为 Purdue 大学杰出的毕业生教师而获得了 Charles B.Murphy Award，同时因为教学出色而获得 Electrical Engineering Technology Award 以及 CTS Electrical Engineering Technology Award。

Sarah A. Cox 获得了 Purdue 大学的计算机和电子工程两个专业的学士学位。她目前担任 Progressive Resources LLC 的软件开发主任，并开发了一些软件项目，包括消费品和工业品及测试设备。

Larry D. O'Cull 获得了 Purdue 大学的电子工程技术专业的学士学位，目前是 Progressive Resources LLC 的高级操作员，他专攻于具有创新性的商业、工业和消费者产品开发。

ISBN 7-302-06955-7
9 787302 069553 >
定价：54.00 元

嵌入式 C 编程与 Atmel AVR

(美) Richard Barnett
Larry O'Cull
Sarah Cox
著

周俊杰
等译

清华大学出版社

北 京

Richard Barnett,Larry O'Cull,Sarah Cox

Embedded C Programming and the Atmel AVR

EISBN: 1-4018-1206-6

981-243-982-x

北京市版权局著作权合同登记号　图字：01-2003-2176

本书封面贴有清华大学出版社激光防伪标签，无标签者不得销售。

图书在版编目(CIP)数据

嵌入式C编程与Atmel AVR/(美)巴雷特，(美)古尔等著；周俊杰等译. -北京：清华大学出版社，2003

书名原文：Embedded C Programming and the Atmel AVR

ISBN 7-302-06955-7

Ⅰ.嵌… Ⅱ.①巴…②古…③周… Ⅲ.①单片微型计算机，AVR②C语言—程序设计 Ⅳ.①TP368.1②TP312

中国版本图书馆CIP数据核字(2003)第064032号

出 版 者：清华大学出版社　　**地　　址**：北京清华大学学研大厦

http://www.tup.com.cn　　**邮　　编**：100084

社 总 机：010-62770175　　**客户服务**：010 62776969

组稿编辑：曹康

文稿编辑：陈宗斌

封面设计：康博

版式设计：康博

印 刷 者：北京牛山世兴印刷厂

发 行 者：新华书店总店北京发行所

开　　本：185×260　**印张**：27　**字数**：684千字

版　　次：2003年9月第1版　2003年9月第1次印刷

书　　号：ISBN 7-302 06955-7/TP·5137

印　　数：1～4000

定　　价：54.00元

前　言

本书不仅讲授 C 语言嵌入式微控制器的编程方法，还提供了应用 Atmel 公司生产的 AVR RISC 系列微控制器的相关知识。

适用对象

本书是针对两类不同的读者而设计的。

- 第一类读者是电子计算机工程、电子工程、电气工程技术、电子工程技术及计算机工程技术专业的学生。本书很适合以下两类学生：
  - 没有学过 C 语言的学生：本书可以作为两学期或 4 个季度课程的课本。这样，学生就可以先学习 C 语言编程，再学习如何在嵌入式微控制器设计中应用 C 语言。按照这个学习顺序，他们就可以学习更复杂的嵌入式应用程序，这些程序应该都能运行在嵌入式微控制器中，而几乎不需要硬件知识。第 1 章(嵌入式 C 语言导论)将作为后续课程的参考资料。
  - 学过 C 语言的学生：本书可以作为一学期或两个季度课程的课本。在这种情况下，学生只需要学习第 1 章中有关嵌入式环境编程的内容，然后就可以迅速进入硬件知识的学习。第 1 章的内容为以后的学习提供必要的参考信息。
- 第二类读者是那些想在工作领域中增加微控制器应用知识的工程师和技术人员。根据不同的需要，第 1 章可以作为学习的内容，或者只是作为参考资料(这将由读者的编程水平而定)。而有关 Atmel AVR 微控制器硬件知识的章节，将引导读者去学习一个新的微控制器。同时它们也都可作为以后学习研究的参考资料。

必备知识

本书要求读者有数字系统和逻辑设计的知识。第 1 章中的相关知识已成功地在微控制器基础课程上使用了。本书是两个学期的基础数字逻辑课程的后续课程(相当于大学二年级水平，没有编程基础)。同时本书也是很好的高级微控制器选修课的教材。很多时候，选修了该课的学生都会保留这本书，以作为以后高级课程项目设计时的参考资料，有的甚至把它当作参考书带到工厂去了。

本书的组织

本书是根据内容的逻辑主题来划分各部分的，所以教师可以依照本书的组织顺序来讲授，从 C 语言开始，然后是 AVR 的硬件，再进入更高级的主题。也可以根据自己具体的需要选择相应的内容。每一个主题都是相对独立，自成一体的。每章后面的练习和上机实习题目都是针对相应主题的，这样可以使读者方便、容易地在具体应用中使用它们。

各章内容简介

第 1 章　嵌入式 C 语言导论

本章详细地讲述了 C 语言，并按部就班地将其应用到嵌入式微控制器的编程中。每个编程概念都有一到两个例子演示它们的用法。学习完本章后，学生就可以编写 C 语言程序来解决问题。

第 2 章　Atmel RISC 处理器

本章内容覆盖了微控制器的基本结构及其内部的每一个标准外设。同样利用示例程序来演示每个外设的常规用法。完成了前面两章的学习之后，学生应该可以利用 AVR RISC 微控制器来解决问题了。

第 3 章　标准 I/O 和预处理函数

本章向学生介绍 C 语言的内部函数及用法。同样，也是利用示例程序来演示这些内部函数的用法。完成第 3 章的学习后，就可以利用库函数加快编程速度并提高解决问题的能力。

第 4 章　CodeVisionAVR C 编译器和集成开发环境

本章是 CodeVisionAVR 编译器及其集成开发环境的使用手册。在本章中，可以学习如何利用 CodeVisionAVR 和 IDE 来有效地创建和调试 C 语言程序。

第 5 章　项目开发

本章集中讨论了应用微控制器的项目的开发过程。通过开发一个完整的无线室内/户外气象台系统来演示这一过程。在本章中，可以学习如何高效地开发一个项目，以取得最大的成效。

附录

附录 A 库函数参考　包含到本书出版时为止，所有可以获得的内部库函数的完整参考资料。

附录 B CodeVisionAVR 和 STK500 入门　本附录是利用 CodeVisonAVR 开发 Atmel STK500 芯片的快速使用指南。

附录 CAVR 微控制器编程　这是 AVR 设备的 FLASH 存储器区域编程的操作指南，通过该指南，可以让您明白编程的原理。

附录 D 安装和使用 CableAVR　该用户手册介绍了如何安装和使用 Cable AVR 软件及硬件。

附录 E MegaAVR-DEV 开发板　MegaAVR-DEV 开发板的说明书和简图。

附录 F ASCII 字符表。

附录 G AVR 指令集汇总　这是示例程序中所用到的汇编语言指令的简要总结。

附录 H 部分练习答案。

基本目的

围绕着微控制器的技术在不断地发展与推进，不停地为微控制器提供更多的功能和更快的速度。这使人们更普遍使用高级语言(例如 C 语言)来处理一些对时间要求很严格的任务，以前这种任务只能用汇编语言来完成。同时，微控制器的应用变得越来越容易了，这使它成为一种很好的教学设备。很多学校都采用微控制器作为他们课程的实验器材。另外，微控制器开发板的价钱已经降低到了一个可以接受的水平，很多学校都可以要求学生自己买开发板作为整个课程的配件，这样每个学生都可以拥有自己的开发板。有的课程需要有 C 语言的基础，有的课程则把 C 语言和嵌入式微控制器的应用综合起来。

本书满足了以上两种课程的要求，并可以作为学生学习后续课程的参考书。

使用的硬件

本书中大部分编程应用实例都是由 Progressive Resources, LLC 公司提供的 AVR 测试版开发板(详细说明请参看附录 E)开发的。这一开发板很适合教学使用，也是很好的通用开发板。Atmel AVR 微控制器非常容易使用。把它插在模型板上，外加一个晶体振荡器，几个电容；以及 4 根用于编程的数据线就可以让它运行得很好。两种方法都可以取得很好的效果。

本书的例子已经在微控制器 AT90S8535 或者 Atmega163 上测试过。AVR 系列微控制器的一大优点是，它们的结构以及设备的编程方式都很相似。因此，所提供的例子可以在任何 Atmel AVR 微控制器上运行，只要它上面有相应的外设和资源——没有必要为 AVR 系列其他的微控制器改变代码。因此，本书也适用于 AVR 系列的其他微控制器。

本书涵盖了最常用的外设，当遇到 AVR 系列其他成员的特殊外设时，本书的代码可以作为参考模板。

作者简介

本书是 3 位作者高度协作的成果。每部分都是先由其中一个人编写，写完以后再由其他两个人认真地审阅，如果有需要，他们会重写其中大部分的内容。所以很难指出那一个作者对书中那一个部分负责。

本书的作者是：

Richard H. Barnett
电子工程技术教授
珀杜大学

Barnett 博士在过去的 18 年里一直从事嵌入式微控制器领域的教学工作。从 Intel 公司的 8085 开始，到 8051 系列嵌入式微控制器，现在则利用 Atmel AVR 设备从事高级嵌入式微控制器的教育工作。在暑假和另外两个假期里，他广泛地从事多处理器嵌入式系统的研究工作，并将它们应用到各种面向控制的应用中。另外，他还积极地在上述领域从事咨询工作。在珀杜大学任职期间，他当了 10 年的航空电子工业工程师。

在教学方面，Barnett 博士获得了很多奖项，包括作为珀杜大学中最好的教师而获得了 Charles B. Murphy 奖。他同时被列入珀杜大学的知名教师手册中，这个知名教师手册上只列举了珀杜大学中最有影响的 225 位教师。本书是他写的第二本教材。

如果您有任何意见或者建议，可以通过珀杜大学的电话 765-494-7497 和他取得联系，也可以发送电子邮件到 rbarnett@purdue.edu

Larry D. O'Cull

资深操作员

Progressive Resources LLC 公司

O'Cull 先生从珀杜大学的电子工程技术学院获得理学学士学位。他开始是为 CNC(Computer Numeric Controlled，计算机数值控制)设备设计软件和控制系统的，后来转向电子工程的其他领域，并为各种各样的系统和设备开发软件，如视觉系统，激光机械设备，医疗诊断设备和各种工业品或消费品等。同时，他还是多项专利的拥有者或者共同拥有者。

O'Cull 在从事了几年的电子和软件工程工作以及工程管理工作后，于 1995 年成立了 Progressive Resources 公司。Progressive Resources LLC 公司(http://prllc.com)专门从事商业、工业和消费品的新产品开发工作。该公司还是 Atmel AVR 顾问团的成员。

如果有任何意见或者建议，可以发送电子邮件到 locull@prllc.com 和他取得联系。

Sarah A. Cox

公司

软件开发主任

Cox 女士在珀杜大学学习软件设计并获得了计算机和电子工程两个理学学士学位。

她曾在一个很大的顾问公司中工作过一段时间，为数据库管理系统提供咨询，之后她被微处理器设计的快速节奏和无限魅力吸引了。在成为 Progressive Resources LLC 公司的合作者之前，她曾独立完成了很多医疗测试设备的设计。

在 Progressive Resources 公司中，Cox 女士为各种项目开发软件，范围涉及最小的消费品到工业产品和测试设备。这些项目跨越了很多领域，包括自动化、医疗、娱乐、儿童发展、公共安全/教育、声音和图像压缩、建筑等。除此之外，她还是很多专利的共同拥有者。她也为系统编程编写软件，为 Atmel AVR 处理器编写开发工具。

如果有任何意见或者建议，请将电子邮件发送到 sac@prllc.com 和她取得联系。

引　言

嵌入式微控制器就是在同一块集成电路中集成了 CPU、主要外设和所需内存的微型计算机，实际上就是在一块芯片上的微型计算机。

嵌入式微控制器的使用历史已经超过 30 年了。Intel 公司的 8051 系列是最早把内存、I/O、算术逻辑部件(ALU)、程序 ROM，以及其他外设都封装到一个小巧的芯片中的微控制器之一。这些处理器现在还被用来设计新产品。跟在 Intel 公司之后进入嵌入式微控制器竞争市场的公司有 General Instruments、National Semiconductor、Motorola、Philips/Signetics、Zilog、AMD、Hitachi、Toshiba 和 Microchip 公司，以及其他公司。

最近几年，Amtel 成为 FLASH 内存技术开发的世界领头人。FLASH 是一种具有非易失性(nonvolatile)但可以重复编程的存储器，它通常用在数码像机、便携式音响设备和 PC 主板上。这种存储技术由于能提供嵌入式系统编程的解决方案，所以大大推进了 Atmel 在微控制器领域中的发展。这样，结合 AVR RISC(Reduced Instruction Set Computing，精简计算指令集)核心结构的发展，就能在对功耗要求很低的情况下提供令人惊讶的功能。

这个高科技领域的另一大发展就是专门针对这类微控制器的高级语言编译器的出现。这些编译器生成代码和优化代码的能力惊人。由于 C 编程语言它有自由的风格和结构，以及代码的可移植性，使它在这方面的应用上具有优势。这类语言最关键的益处在于它创建的智能属性池可以重用。这些底层的开发可以降低随后设计的开发周期，从而降低成本。

目前，最好的一个 C 语言开发工具就是 CodeVisionAVR。它是由 HP InfoTech S.R.L 的 Pavel Haiduc 编写的。这个完整的集成开发环境(IDE)可以允许在 PC 机的 Windows 应用程序中进行编辑、编译、分块编程和调试。

AVR 和其他 RISC 微控制器的流行、集成水平的不断提高(一个芯片中集成越来越多器件，而电路板上的器件却越来越少)，以及在利用这些技术开发产品时对“协调思想”的需求，促使我们产生了编写这本书的动机。您可能有过为 PC 编写 C 语言程序，或者为微控制器编写汇编语言程序的经历。但是在为嵌入式微控制器编写 C 语言代码时，要改变所用的方法，以达到期望的结果——短小、高效、可靠及可重用的代码。本书的目的之一就是为初学者提供一个比较好的起点，同时，本书对于那些在嵌入式微控制器设计上有丰富经验的人来说，也是一本有用的参考书。

目　　录

第 1 章　嵌入式 C 语言导论 …… 1
1.1　本章目标 …… 1
1.2　引言 …… 1
1.3　基本概念 …… 1
1.4　变量和常量 …… 4
1.4.1　变量类型 …… 4
1.4.2　变量的作用域 …… 4
1.4.3　常量 …… 5
1.4.4　枚举和定义 …… 7
1.4.5　存储类型 …… 8
1.4.6　类型转换 …… 8
1.5　输入/输出操作 …… 9
1.6　运算符和表达式 …… 10
1.6.1　赋值运算符和算术运算符 …… 10
1.6.2　逻辑运算符与关系运算符 …… 13
1.6.3　自增运算符、自减运算符和复合赋值运算符 …… 14
1.6.4　条件表达式 …… 15
1.6.5　运算符优先级 …… 16
1.7　控制语句 …… 17
1.7.1　while 循环 …… 17
1.7.2　do/while 循环 …… 18
1.7.3　for 循环 …… 19
1.7.4　if/else 语句 …… 20
1.7.5　switch/case 语句 …… 23
1.7.6　break、continue 和 goto 语句 …… 25
1.8　函数 …… 29
1.8.1　原型和函数组织 …… 30
1.8.2　函数返回值 …… 32
1.8.3　递归函数 …… 33
1.9　指针和数组 …… 37
1.9.1　指针 …… 37
1.9.2　数组 …… 40
1.9.3　多维数组 …… 42

1.9.4 指向函数的指针 ····· 44
1.10 结构与共用体 ····· 48
1.10.1 结构 ····· 48
1.10.2 结构数组 ····· 50
1.10.3 指向结构的指针 ····· 51
1.10.4 共用体 ····· 52
1.10.5 typedef 运算符 ····· 54
1.10.6 位和位域 ····· 55
1.10.7 sizeof 运算符 ····· 56
1.11 存储器类型 ····· 57
1.11.1 常量和变量 ····· 57
1.11.2 指针 ····· 59
1.11.3 寄存器变量 ····· 59
1.12 实时方法 ····· 62
1.12.1 使用中断 ····· 62
1.12.2 状态机 ····· 65
1.13 本章小结 ····· 71
1.14 练习 ····· 71
1.15 上机实习 ····· 73

第 2 章 Atmel Risc 处理器 ····· 75
2.1 本章目标 ····· 75
2.2 引言 ····· 75
2.3 体系结构概述 ····· 75
2.4 存储器 ····· 76
2.4.1 FLASH 代码存储器 ····· 77
2.4.2 数据存储器 ····· 77
2.4.3 EEPROM 存储器 ····· 81
2.5 复位和中断功能 ····· 82
2.5.1 中断 ····· 83
2.5.2 复位 ····· 86
2.6 并行 I/O 端口 ····· 88
2.7 计时器/计数器 ····· 93
2.7.1 计时器/计数器预定标器和输入选择器 ····· 93
2.7.2 Timer 0 ····· 94
2.7.3 Timer 1 ····· 97
2.7.4 Timer 2 ····· 108
2.8 使用 UART 进行串行通信 ····· 112

2.9 模拟接口 …… 118
2.9.1 模数转换背景知识 …… 118
2.9.2 模数转换器外设 …… 119
2.9.3 模拟比较器 …… 122
2.10 利用 SPI 进行串行通信 …… 127
2.11 AVR RISC 汇编语言指令集 …… 130
2.12 本章小结 …… 132
2.13 练习 …… 136
2.14 上机实习 …… 137
第 3 章 标准 I/O 和预处理函数 …… 139
3.1 本章目标 …… 139
3.2 引言 …… 139
3.3 字符输入/输出函数 getchar()和 putchar() …… 139
3.4 标准输出函数 …… 145
3.4.1 输出字符串函数 puts() …… 145
3.4.2 从 FLASH 输出字符串函数 PUTSF() …… 146
3.4.3 格式打印函数 printf() …… 147
3.4.4 字符串格式打印函数 sprintf() …… 149
3.5 标准输入函数 …… 150
3.5.1 获得字符串函数 gets() …… 150
3.5.2 格式扫描函数 scanf() …… 151
3.5.3 字符串格式扫描函数 sscanf() …… 152
3.6 预处理指令 …… 153
3.6.1 #include 指令 …… 153
3.6.2 #define 指令 …… 154
3.6.3 #ifdef，#ifndef，#else 和#endif 指令 …… 156
3.6.4 #pragma 指令 …… 162
3.6.5 其他宏和指令 …… 165
3.7 本章小结 …… 166
3.8 练习 …… 166
3.9 上机实习 …… 167
第 4 章 CodeVisionAVR C 编译器和集成开发环境 …… 168
4.1 本章目标 …… 168
4.2 引言 …… 168
4.3 集成开发(IDE)环境操作 …… 169
4.3.1 项目 …… 169
4.3.2 源文件 …… 171

4.3.3 编辑文件 ····· 175
4.3.4 打印文件 ····· 176
4.3.5 文件导航器 ····· 176
4.4 C 编译器选项 ····· 177
4.4.1 UART ····· 178
4.4.2 存储器模式 ····· 178
4.4.3 优化 ····· 179
4.4.4 程序类型 ····· 179
4.4.5 SRAM ····· 179
4.4.6 编译 ····· 179
4.5 编译和生成项目 ····· 180
4.5.1 编译项目 ····· 180
4.5.2 生成项目 ····· 182
4.6 对目标设备编程 ····· 183
4.6.1 芯片 ····· 183
4.6.2 FLASH 和 EEPROM ····· 183
4.6.3 FLASH 锁定位 ····· 185
4.6.4 保险位 ····· 185
4.6.5 Boot Lock Bit 0 和 Boot Lock Bit 1 ····· 185
4.6.6 签名 ····· 185
4.6.7 芯片擦除 ····· 186
4.6.8 编程速度 ····· 186
4.6.9 Program | All ····· 186
4.6.10 其他编程器 ····· 186
4.7 CodeWizardAVR 代码生成器 ····· 188
4.7.1 Chip 选项卡 ····· 189
4.7.2 Ports 选项卡 ····· 190
4.7.3 External IRQ 选项卡 ····· 191
4.7.4 Timers 选项卡 ····· 192
4.7.5 UART 选项卡 ····· 193
4.7.6 ADC 选项卡 ····· 194
4.7.7 Project Information 选项卡 ····· 195
4.7.8 生成源代码 ····· 196
4.8 终端工具 ····· 207
4.9 Atmel AVR Studio 调试器 ····· 208
4.9.1 为 AVR Studio 新建一个 COFF 文件 ····· 208
4.9.2 从 CodeVisionAVR 启动 AVR Studio ····· 209
4.9.3 打开文件进行调试 ····· 209

4.9.4 开始、中断和跟踪 ……209
4.9.5 设置和清除断点 ……210
4.9.6 查看和修改寄存器和变量的值 ……210
4.9.7 查看和修改机器状态 ……211
4.10 本章小结 ……211
4.11 练习 ……212
4.12 上机实习 ……213

第 5 章 项目开发 ……214
5.1 本章目标 ……214
5.2 引言 ……214
5.3 概念开发阶段 ……214
5.4 项目开发过程的步骤 ……214
5.4.1 定义阶段 ……214
5.4.2 设计阶段 ……216
5.4.3 测试定义阶段 ……217
5.4.4 建立和测试硬件原型阶段 ……217
5.4.5 系统集成和开发阶段 ……218
5.4.6 系统测试阶段 ……218
5.4.7 庆祝阶段 ……218
5.5 项目开发过程总结 ……218
5.6 示例项目：一个气象监测器 ……219
5.6.1 构思阶段 ……219
5.6.2 定义阶段 ……219
5.6.3 测量方法在设计方面的考虑 ……224
5.6.4 室外装置的硬件设计 ……235
5.6.5 室外装置的软件设计 ……237
5.6.6 室内装置的硬件设计 ……239
5.6.7 室内装置的软件设计 ……241
5.6.8 测试定义阶段 ……243
5.6.9 建立和测试原型硬件阶段 ……244
5.6.10 系统集成和软件开发阶段——室外装置 ……250
5.6.11 系统集成和软件开发阶段——室内装置 ……258
5.6.12 系统测试阶段 ……284
5.7 挑战 ……288
5.8 本章小结 ……288
5.9 练习 ……288
5.10 上机实习 ……289

附录 A　库函数参考 …… 290
附录 B　CodeVisionAVR 和 STK500 入门 …… 369
附录 C　AVR 微控制器编程 …… 381
附录 D　安装并使用 TheCableAVR …… 384
附录 E　MegAVR-DEV 开发板 …… 394
附录 F　ASCII 字符表 …… 397
附录 G　AVR 指令集汇总 …… 401
附录 H　部分练习答案 …… 409

第1章　嵌入式C语言导论

1.1　本章目标

通过本章的学习，您应该掌握以下内容：

- 定义、描述和识别变量类型与常量类型，以及这些类型的作用域和用法
- 为各种大小的数值数据和字符串构造变量常量声明
- 将枚举应用于变量声明
- 通过赋值运算符给变量和常量赋值
- 评估 C 语言中使用的所有运算符的效果
- 解释每种控制语句对程序流程产生的效果
- 创建包含有变量、运算符和控制语句的函数，以完成特定任务
- 使用指针、数组、结构和共用类型作为函数变量
- 用本章的概念创建 C 语言程序来完成任务

1.2　引言

本章介绍了将 C 语言应用于嵌入式微控制器程序时必须了解的基本知识，包括作为 CodeVisionAVR C 语言的一部分的 C 语言扩展。您将从基本概念的学习开始，然后编写完整的程序，同时通过可在微控制器上实现的例子来加强学习。

本章的介绍是根据程序员需要学习的内容安排的：

- 声明变量和常量
- 简单的输入输出(I/O)，让程序可以通过 I/O 来使用微控制器的并行端口
- 变量和常量赋值，对变量执行算术运算
- C 构造语句与控制语句，通过这些语句来形成完整的 C 程序

最后一节将介绍更高级的主题，例如指针、数组、结构和共用类型，以及它们在 C 程序中的用法。最后以实时程序设计和中断等高级概念来结束本章。

1.3　基本概念

编写 C 程序在某种意义上来说就好像是用砖建房子：首先打好地基，使用沙子和水泥造砖，把这些砖堆砌起来，然后建成房子。在嵌入式 C 语言程序中，指令集放在一块形成函数，函数

被当作更高一级的运算，由函数结合组成程序。

每个 C 语言程序至少有一个函数，即 main()函数。main()函数是 C 语言程序的基础，是程序代码执行的起点。所有的函数都是通过 main()函数直接或间接调用的。尽管函数可以是完整的和独立的，但变量和参数可以同函数结合在一起。

main()函数通常被认为是最低级的任务，因为它是启动该程序的系统所调用的第 1 个函数。在很多情况下，main()都只包含很少的语句，这些语句的作用仅仅是初始化和指导从一个函数到另一个函数的程序操作。

一个最简单的嵌入式 C 语言程序如下：

```
void main()
{
    while (1)   // do forever..
          ;
}
```

以上程序可以很好地进行编译和运行，不过您可能不会知道这一点，因为该程序在运行时没有给出任何界面内容。可以对该程序进行改进，以便您能看出它的存在和功能，并且可以开始学习这种语言的语法元素。

```
#include<stdio.h>

void main()
{
    printf("HELLO WORLD");/*the classic C test program..*/
    while(1)            //do forever..
        ;
}
```

以上程序将结果“HELLO WORLD”输出到标准输出设备上，该输出设备最有可能是一个串行端口。如果在微控制器中运行该程序，那么微控制器会一直等待下去，或者直到微控制器重新启动。这种情况显示了 PC 机程序与嵌入式微控制器程序之间的基本区别：嵌入式应用程序包含无限循环。PC 机有操作系统，一旦程序执行，它把控制权交给操作系统。而嵌入式微控制器没有操作系统，不允许随时任意结束程序，所以每个嵌入式微控制器程序都具有无限循环(这个无限循环位于该程序中的某个位置)，例如上面例子中的 while(1)。这样可以防止程序无事可做或者做不可预测的随机事件。While 结构将在后面的一节中介绍。

上例程序也提供了一个常见预处理编译器(preprocessor compiler)指令的实例。#include 告诉编译器包含 stdio.h 文件作为本程序的一部分。printf()函数由外部库提供，可以在该程序中使用，是因为该函数的定义位于 stdio.h 中文件中。在以后的章节中，这些概念很快就会放在一块讨论。

上述的例子中，有一些值得注意的元素，其含义见表 1-1：

表 1-1　程序元素

元　素	含　义
;	分号用于指示表达式的末尾。一个最简单的表达式就是一个分号本身
{}	花括号{}用于描述函数内容的开始与结束，还用于指示将一系列语句作为一个整块来对待
"text"	双引号用来标记文本字符串的开始与结束
//或/*...*/	双斜杠或斜杠加星号用作注释定界符

注释只是程序员加上的注解。注释对于程序的可读性十分重要，不管程序是让其他人阅读还是最初的原作者在以后阅读。本书中给出的注释用来解释示例代码中每一行的功能。注释总是解释程序中的代码行的作用，而不是重复在这一行代码中使用的特定指令。

传统的注释定界符使用斜杠-星号(/*)和星号-斜杠(*/)。斜杠-星号用于注释的开始。编译器一旦遇到斜杠-星号(/*)，就忽略后面的文本(即使是多行文本)，直到遇到星号-斜杠(*/)。对于前面的程序而言，main()函数的第 1 行就是此分界符的例子。

对于双斜杠分界符，编译器只忽略它所在那一行的文本。例如前面程序中 main()函数的第 2 行。

随着对细节介绍的深入，我们应记住一些语法规则和基本术语：

- 标识符是变量名或函数名，由字母或下划线后跟一连串的字母和/或数字，和/或下划线组成
- 标识符区分大小写
- 标识符可以为任意长度，但有些编译器只识别有限数量的字符，如前 32 个字符。因此应当小心
- 特定单词对于编译器具有特殊的含义，被认为是保留字。这些保留字应当以小写形式输入，并且从不应用作标识符。下面列出了这些保留字。

auto	defined	float	Long	static	while
break	do	for	Register	struct	
bit	double	funcused	return	switch	
case	eeprom	goto	short	typedef	
char	else	if	signed	union	
const	enum	inline	sizeof	unsigned	
continue	extern	int	sfrb	void	
default	flash	interrupt	sfrw	volatile	

- 因为 C 语言是一种自由形式的语言，除非有双引号引用，否则“空白”均被忽略。空白包括空格(间隔)、制表符和新行(回车符和/或换行符)。

1.4 变量和常量

下面考虑以变量和常量形式存储的数据。在代数中，变量是可被改变的值，而常量是固定的。变量和常量有多种形式和大小，它们在程序存储器中以各种形式存储，我们在讲述的过程中会分析存储形式。

1.4.1 变量类型

变量是通过保留字(用于指示变量的类型和大小)和跟在保留字后面的标识符来声明的：

```
unsigned char Peabody ;
int dogs , cats ;
long int total_dogs_and_cats;
```

变量和常量存储在微控制器的有限存储器中，编译器需要知道为每个变量预留多少存储器，而不浪费不必要的存储器空间。因此，程序员必须声明变量，同时指明变量的大小和类型。表 1-2 列出了变量类型及其大小。

表 1-2　变量的类型与大小

类　型	大小(位)	范　围
bit	1	0, 1
char	8	- 128～127
unsigned char	8	0～255
signed char	8	- 128～127
int	16	- 32 768～32 767
short int	16	- 32 768～32 767
unsigned int	16	0～65 535
signed int	16	- 32 768～32 767
long int	32	- 2 147 483 684～2 147 483 647
unsigned long int	32	0～4 294 967 295
signed long int	32	- 2 147 483 648～2 147 483 647
float	32	±1.175e - 38～±3.402e38
double	32	±1.175e - 38～±3.402e38

1.4.2 变量的作用域

如同前面已经讲过的，变量和常量在使用前需要先声明。变量的作用域是变量在程序中的可访问性。变量可被声明为局部作用域或全局作用域。

1. 局部变量

局部变量是在创建函数时由函数分配的存储器空间，典型情况是在程序栈或由编译器创建

的栈空间上。这些变量不能被其他的函数访问，这意味着这些变量的作用域只限于所声明的函数内。局部变量声明可以在多个函数中使用，而不会引起冲突，因为编译器会将这些变量视为每个函数的一部分。

2. 全局变量

全局变量(或外部变量)是由编译器分配的存储器空间，可被程序内所有的函数访问(即作用域不受限制)。全局变量能被任何函数修改，并且会保持全局变量的值，以便其他函数可以使用。

典型情况下，全局变量在 main()函数开始执行时进行了清除(置为零)。此操作最常由编译器产生的启动代码执行，对于程序员来说是不可见的。

下面的代码块用于演示变量的作用域：

```
unsigned char globey;              //a global variable

void function_z (void)             //this is a function called from main()
{
      unsigned int tween;          // a local variable

      tween=12;                    //OK because tween is local
      globey=47;                   //OK because globey is global
      main_loc=12;                 //This line will generate an error
                                   //because main_loc is local to main
}

void main()
{
      unsigned char main_loc ;     // a variable local to main()
      globey=34;                   //OK because globey is a global
      tween=12;                    //will cause an error—tween is local
                                   //to function_z

      while(1)                     //do forever.....
            ;

}
```

在一个函数内，如果局部变量和全局变量同名，那么，该函数只使用局部变量，而不会用到全局变量。

1.4.3　常量

如前所述，常量有固定值，在程序执行时不会改变。常量在很多情况下是经过编译之后的程序本身的一部分，位于只读存储器(ROM)中，不是被分配到可变随机存储器(RAM)中。在赋值运算

```
x=3+y;
```

中，数字 3 是常量，可以由编译器直接编码到加法操作中。常量同样可以是字符形式或字符串形式：

```
printf("hello world");
        //The text hello world is placed in program memory
        //and never changes.
x='B'   //The letter 'B' is permanently set in
        //program memory.
```

您同样可以通过保留字 const 来声明常量，并且定义其类型和大小。还需要标识符和指定的值来完成常量的声明：

```
const char c=57;
```

把变量指定为常量，可使该变量存储在程序代码空间中，而不是存储在 RAM 中的有限可变存储空间中，这样有助于节省有限的 RAM 空间。

1. 数值型常量

数值型常量可以通过指定基数来以多种形式声明，从而使得程序更具有可读性。整型或长整型常量可写成如下形式：

- 无前缀的十进制形式(如 1234)
- 前缀为 0b 的二进制形式(如 0b101001)
- 前缀为 0x 的十六进制形式(如 0xff)
- 前缀为 0 的八进制形式(如 0777)

还有一些修饰符可以更好地定义常量的预期大小和用法：

- 无符号整型常量可以有后缀 U(如 10000U)
- 长整型常量可以有后缀 L(如 99L)
- 无符号长整型常量可以有后缀 UL(如 99UL)
- 浮点型常量可以有后缀 F(如 1.234F)
- 字符型常量必须用单引号括起来，例如'a'或'A'

2. 字符型常量

字符型常量可以是可打印字符(比如 0～9，A～Z)，也可以是无法打印出来的字符(如换行符、回车符或者制表符)。可打印字符型常量可能由单引号引起来(例如'a')。反斜杠后跟一个八进制或十六进制值，并由单引号引起来，也可表示一个字符型常量：

```
't'可由'\164'表示(八进制)，也可以由'\x74'表示(十六进制)
```

表 1-3 列出了可以由 C 语言识别的不可打印字符。

表 1-3　不可打印字符

字　符	表　示	对应的十六进制值
BEL	'\a'	'\x07'
Backspace	'\b'	'\x08'
TAB	'\t'	'\x09'
LF(换行符)	'\n'	'\x0a'
VT	'\v'	'\x0b'
FF	'\f'	'\x0c'
CR	'\r'	'\x0d'

反斜杠(\)和单引号(')字符本身必须由一个前导的反斜杠来区分，以免编译器混淆。例如，'\'' 是一个单引号字符，'\\'是一个反斜杠。BEL 是一个响铃字符，当被计算机终端或终端仿真器接收时会发出声音。

1.4.4　枚举和定义

在 C 语言程序中，可读性是十分重要的。C 语言提供了枚举类型和定义类型，这使程序员能够用有意义的名字或其他更为有意义的短语来代替数字。

枚举是指列出的常量，保留字 enum 用于将连续的整型常量值赋给一列标识符：

```
int num_val;                    //declare an integer variable

//declare an enumeration
enum{zero_val , one_val , two_val , three_val };

num_val=two_val;                // the same as :num_val=2;
```

名字 zero_val 被赋予常量值 0，one_val 被赋予常量 1，two_val 被赋予常量 2，依此类推。还可以强加一个初始值，例如：

```
enum {start=10 ,next1 ,next2 ,end_val}
```

将 start 赋值为 10，相应的后续名字依次加 1。next1 为 11，next2 为 12，next3 为 13。枚举用单词或词语来替代程序员必须查找的纯数字，以描述数字的用法。

定义在某种程度上同枚举类似，因为定义允许用一个文本串代替另一个文本串，例如：

```
enum { red_led_on = 1 , green_led_on , both_leds_on};
#define leds PORTA
.
.
PORTA=0x1;   //means turn the red LED on
leds=red_led_on;   //means the same thing
```

语句#define leds PORTA 使编译器在遇到单词 leds 时用 PORTA 来代替 leds。注意#define

这一行没有以分号结束。枚举设置 red_led_on 的值为 1，green_led_on 的值为 2，both_leds_on 的值为 3。它可能在程序被用来控制红绿发光二极管(LED)，当输出结果 1 时，红灯亮；输出结果 2 时，绿灯亮等。其意义在于程序中的常量 leds＝red_led_on 比 PORTA＝0×1 更易懂。

#define 是预处理程序指令。预处理程序指令实际上不是 C 语言语法的一部分，但是它们同样得到了广泛的接受与使用。预处理程序是与实际的程序编译分开的一个阶段，预处理程序在编译开始之前就开始了。关于预处理程序的更多内容参见第 3 章。

1.4.5　存储类型

变量可声明为 3 种存储类型：auto、static 和 register。Auto(automatic)是默认的类型，所以保留字 auto 可以省略。

1. 自动变量

自动类型的局部变量在分配空间时没有进行初始化，所以在它使用前由程序员来确认它包含有效数据。当函数退出时，它所有占用的存储器空间释放掉了，这同时意味着当再次进入函数时，变量的值会丢失或无效。自动类型的变量声明方式如下：

```
auto int value_1;
```

或者

```
int value_1;                    //this is the common, default form
```

2. 静态变量

静态局部变量只有所定义的函数作用域(不能被其他的函数访问)，但在全局存储器空间中分配。当函数第一次调用时，它被初始化为零，当函数退出时，该变量保持它的值。因此，在每次重新调用该函数时，这个变量的值是最新的并且仍然有效。

```
static int value_2;
```

3. 寄存器变量

寄存器局部变量与自动变量类似，它没有被初始化，并且是临时的。不同之处在于：在微控制器中编译器会试图使用实际机器寄存器作为变量，以减少访问该变量所需的机器循环周期。在典型的计算机中，与它的总存储器相比，其寄存器的数量极少。因此这种类型的变量应当节省使用，只用于加快特定过程的速度。

```
register char value_3;
```

1.4.6　类型转换

程序员有时希望临时强制变量的类型和大小。类型转换允许在操作执行期间覆盖以前声明的类型。类型转换通过在表达式前加上括号和所需类型来实现。

例如，假设有如下的声明和赋值语句：

```
int x;          //   a signed,16-bit ,integer ( - 3278 to 32767)
char y;         //    a signed ,8-bit character( - 128 to 127)
x=12;           //     x is a integer,(but its value will fit into a character)
```

对于这些变量的类型转换操作为：

```
y=(char)x+3     // x is converted to a character and then 3 is added, and the value is then placed into y
x=(int)y;       //y is extended up to an integer, and assigned to x
```

当使用不同大小的变量来执行算术运算时，类型转换就显得特别重要。在很多情况下，运算的准确度由最小类型的变量决定。考虑如下的表达式：

```
int z;          //declare z
int x=150;      // declare and initialize x
char y=63;      //declare and initialize y

z=(y*10)+x;
```

当编译器处理等式右边的运算时，它将考虑 y 的大小，并假设(y*10)是一个字符(8 位)乘法运算。放在堆栈上的结果将超出存储位置的宽度—— 一个字节或值 255，这样会得到截断之后的结果 118(0x76)，而不是正确的值 630(0x276)。在求值的下一个阶段，编译器认为是 16 位的运算，118 将被扩展为整型，然后再和 x 相加。最后 z 将赋值为 268，这是严重错误！

应当用数据类型转换来对这种假设进行控制，如果将同样的算术表达式写成

```
z=((int)y*10)+x;
```

这条语句会告诉编译器 y 在这次运算中被作为 16 位的整型数。所以 16 位乘法运算中堆栈的值为整数 630。加上整型值 x 得到 780，于是 z 将赋值为 780。

C 语言的灵活性很强。您要求什么它将给您什么。编译器所要作出的假定就是程序员知道自己想要什么东西。在上面的例子中，如果整数 x 是 6 而不是 63，则不会发生错误。在写表达式时，应考虑到表达式的最大值和加法、乘法结果的最大值。

要遵循的一条好的规则是："有疑问时，进行类型转换"。除非能够确定不需要进行转换，否则要进行变量转换。

1.5　输入/输出操作

嵌入式微控制器必须直接同其他硬件交互。因此，它们的许多输入输出操作是通过其内置的并行端口实现的。

大多数 C 编译器提供了同并行端口进行交互的便利方法，这些编译器是通过一些库或头文件来实现这些交互方法的，这些库或头文件使用 sfrb 和 sfrw 编译器命令来给每个并行端口和其他 I/O 设备分配标签。我们将在 1.6 节中讨论这些命令，下面的代码只用于演示并行端口的用法：

```
#include <90s8535.h>          //register definition header file for an Atmel AT90s8535
```

```
unsigned char z;                    //declare z
void main(void)
{
    DDRB=0xff;                      //set all bits of port B for output

    while(1)
    {
        z=PINA;                     //read the binary value on the port A pins(i.e.  input from port A)
        PORTB=z+1;                  //write the binary value read from porta plus 1 to port B
    }
}
```

上例演示了读写并行端口的方法。z 声明为无符号字符型大小变量，因为端口是一个 8 位的端口，而无符号字符型变量正好容纳 8 位数据。注意到端口 A 和输出端口 B 的引脚标签均为大写，因为这些标签必须同头文件 90s8535.h 中定义标签的方式相匹配。

DDRx 寄存器用来决定端口 x(由处理器规定的 A、B 等)的哪些位用于输出。当重启时，所有 I/O 端口默认为输入，同时，DDRx 寄存器的所有位清零。接着，程序员根据将哪些位用于输出来设置 DDRx 位。例如：

```
DDRB＝0xc3   //set the upper 2 and lower 2 bits of port B for output
```

本示例将最高两位和最低两位配置成输出位。

1.6 运算符和表达式

表达式是一条语句，在该语句中，运算符将标识符连接起来。当对表达式求值时，结果可为真(true)、假(false)，也可以是数值。运算符用其周围的标识符指明编译器执行哪种类型的运算。在执行的运算中，有用于操作优先级或者顺序的规则。当一个表达式中结合了多个运算符时，必须应用优先级规则才能获得结果。

1.6.1 赋值运算符和算术运算符

一旦声明了变量，就可使用赋值运算符(＝)来对其进行运算。赋给变量的值可为常量、变量或表达式。在 C 语言中，表达式是操作数(标识符)与运算符的结合。典型的赋值运算如下：

```
dog=35;
val=dog;
dog=dog+0x35;
val=(2*(dog+0172))+6;
y=(m*x)+b
```

已编译好的程序在算术地处理表达式时，就好像进行手算一样。从左到右运算，有括号的话，先计算括号内的表达式。上例中 y=(m*x)+b，m 和 x 先相乘，然后将相乘的结果再加上 b，最后将结果赋值给 y。除括号外，运算符本身存在优先级。乘法和除法优先于加法和减法。因

此表达式：

y=m*x+b；同 y=(m*x)+b；是一样的。

括号可以增加代码的可读性。表 1-4 按照优先级顺序列举了典型的算术运算符。

表 1-4　算术运算符

名　称	符　号
乘法	*
除法	/
取余	%
加法	+
减法或取负	-

除了算术运算符和赋值运算符外，还存在其他许多类型的运算符，包括位运算符、逻辑运算符、关系运算符、递增运算符、递减运算符、复合赋值运算符和条件运算符。

位运算符

位运算符执行的功能影响处于位一级的操作数。这些运算符只对非浮点操作数(char、int 和 long 类型)起作用。表 1-5 按照优先级顺序列出了位运算符。

表 1-5　位运算符

名　称	符　号
取反	~
左移	<<
右移	>>
与	&
异或	^
或	|

各个位运算符的描述如下：

- 取反运算符用于对操作数按位取反，即将 0 变为 1，从 1 变为 0
- 左移运算符用来将操作数的各二进制位全部左移若干次，这个次数是通过右边的操作数来指定的。将已经为“空”的低位补零。左移 1 位相当于对操作数乘以 2
- 右移运算符用来将操作数的各二进制位全部右移若干次，这个次数是通过右边的操作数来指定的。右移 1 位相当于除以 2。在右移时，对于有符号变量和无符号变量的处理方式不同。在有符号整数中的符号位(左边的位或者最高有效位)会保持不变。符号扩充允许正数或负数在保持符号不变的同时进行右移。当无符号变量右移时，总是从左边移入 0
- 与运算符在两个操作数的位都为 1 时，其结果才为 1
- 异或运算符在两个操作数的位不同时(一个为 0，另一个为 1)，结果才为 1
- 或运算符在两个操作数的位只要有一个为 1，其结果就为 1

表 1-6 给出了一些例子。

假设无符号字符 y=0xC9。

表 1-6　位运算示例

运　算	结　果
x=~y	x=0x36
x=y<<3	x=0x48
x=y>>4	x=0x0C
x=y&0x3F	x=0x09
x=y^1	x=0xC8
x=y|0x10	x=0xD9

按位与和按位或运算符在处理并行端口时十分有用。考虑下面的例子：

```
#include <90s8535.h>          //register definition header file for an Atmel AT90s8535
unsigned char z;              //deciare z

void main(void)
{
    DDRB=0xff;                //set all bits of port B for output

    while(1)
    {
        z=PINA&0x6;           //read the binary value on the
                              //port A pins ANDed with 00000110
        PORTB=PORTB|0x60;     //write the binary value from port B
                              //ORed with 01100000 back to port B
        PORTB=PORTB&0xfe;     //write the binary value from port B
                              //ANDed with 01100000 back to port B
                              //plus 1 to port B
    }
}
```

上面的例子演示了屏蔽(masking)和位端口控制(bitwise port control)。屏蔽是用来确定二进制值特定位的值的技术。本例屏蔽的实现方法是：用 0 同不想要的位进行与操作(x 同 0 进行与操作得 0)，需要的位则同 1 进行与操作(x 同 1 进行与操作得 x)。这样，所有的位(最低四位字节的中间两位除外)都设置为 0 了。

位端口控制可以改变并行端口的一个或多个位，但不会影响这个端口的其他位。第 1 次出现 PORTB 的代码行使用或运算符来使这个端口的两位变高，而不影响该端口的其他位。第 2 次出现 PORTB 的代码行显示了如何通过用一个 0 进行与运算来使端口的一位变低。

1.6.2　逻辑运算符与关系运算符

逻辑运算符和关系运算符都是二元运算符，但产生的结果要么为真(True)，要么为假(False)；真由所有非零值表示，而假由零值表示。这些操作通常用在控制语句中，引导程序执行流程。

1. 逻辑运算符

表 1-7 按照优先级显示了逻辑运算符。

表 1-7　逻辑运算符

名　称	符　号
与(AND)	&&
或(OR)	||

逻辑运算符与位运算符差别很大，因为逻辑运算符以真假含义来处理操作数。只有当两个操作数都为真时逻辑与操作的结果才为真，否则结果为假。两个操作数中只要有一个为真，逻辑或操作的结果就为真；只有当两个操作数均为假时，其结果才为假。以下示范了它们之间的区别：

假设 x=5，y=2；

(x&&y)　　结果为真，因为两个操作数都是非零的。

(x&y)　　结果为假，因为模式 101b 和 010b 进行按位与操作得结果 0。

(x||y)　　结果为真，因为两个操作数中的任何一个都为非零。

(x|y)　　结果为真，因为模式 101b 和 010b 进行按位或操作得出结果 111b(非零)。

2. 关系运算符

关系运算符使用比较操作。和在逻辑运算符中一样，操作数也是从左至右进行求值，所得到结果可以为真，也可以为假。实际上，关系运算符通过询问两个表达式的关系得出一个真或假的答案。

“左边的操作数大于右边的操作数吗？”

“左边的操作数小于或等于右边的操作数吗？”

表 1-8 所示为关系运算符。表 1-9 为例子。

表 1-8　关系运算符

名　称	符　号
等于	==
不等于	!=
小于	<
小于或等于	<=
大于	>
大于或等于	>=

假定 x=3，y=5。

表 1-9　关系运算符示例

运　算	结　果
(x==y)	假
(x!=y)	真
(x<y)	真
(x<=y)	真
(x>y)	假
(x>=y)	假

1.6.3　自增运算符、自减运算符和复合赋值运算符

随着 C 语言的发展，我们努力使 C 语言程序变得简洁而明确。一些速记运算符相应引入到 C 语言中，以便在程序开发中简化表达式，减少按键次数。这些运算符包括自增运算符、自减运算符和复合赋值运算符。

1. 自增运算符

自增运算符可修改标识符，使用前加或者后加的方式，例如

```
x=x+1
```

和

```
++x;    //pre-increment operation
```

是一样的。同

```
x++;    //post-increment operation
```

也是一样的，x 值加 1。++x 是前加操作，而 x++是后加操作。这意味着，经过编译的代码对表达式进行求值时，值是通过前面求值或者后面求值来进行修改的。

例如，

```
i=1;
k=2*i++;        //at completion, k=2 and i=2

i=1,
k=2*++i;        //at completion k=4 and i=2
```

在第 1 种情况下，i 先和 2 相乘，再加 1。在第 2 种情况下，i 先加 1 再和 2 相乘。

2. 自减运算符

自减运算符与自增运算符类似，执行一次减 1 操作，有前减操作和后减操作：

```
j—;  //j=j—1
—j;  //j=j—1
```

3. 复合赋值运算符

复合赋值运算符是在程序构造的过程中减少语法数量的另一种方法。它由赋值运算符(=)同算术运算符或逻辑运算符合成。表达式从右至左处理，在语法结构上有点类似自增和自减运算符。例如：

```
a+=3;                    //a=a+3
b - =2;                  //b=b - 2
c*=5;                    //c=c*5
d/=a;                    //d=d/a
```

赋值运算符还可以和取模运算符、位运算符(%、>>、<<、&、|和^)结合使用，如下所示：

```
a|=3;                    //a=a OR3
b&=2;                    //b=b AND 2
c^=5;                    //c=c exclusively OR-ed with 5

PROTC &=3;               //write the current value on PORTC
                         //ANDed with 3 back to port C. Forcing
                         //all of the bits except the lower 2 to 0
                         //and leaving the lower 2 bits unaffected
```

1.6.4　条件表达式

条件表达式可能是最隐蔽和最不常用的运算符。它的发明很明显是为了节省打字时间，但通常对于初级程序员来说不容易掌握。因为它的物理构造，我们在本节介绍条件表达式。不过它与 1.7 节的控制语句联系更加紧密。如果有控制语句：

```
if (expression_A)
    expression_B;
else
    expression_C;
```

用条件表达式表示，则为：

```
expression_A ? expression_B:expression_C;
```

以上任何一种形式，都会首先计算逻辑表达式 expression_A：如果该表达式为真，则执行表达式 expression_B，否则执行表达式 expression_C。

在程序中，条件表达式可写成如下的形式：

```
(x<5) ? y—1:y=0;    //if x is less than 5, then y=1,else y=0
```

1.6.5　运算符优先级

在一个语句中有多个表达式时，运算符的优先级确定了编译器对表达式进行求值的顺序。在出现赋值和表达式的各种情况下，必须记住这些优先级。当有疑问时，可以加上括号来保证处理顺序，或者查找当前涉及到运算符的优先级。在前面列出的一些运算符具有相同的优先级。表 1-10 列出了在执行(从左至右或从右至左)时它们的优先级和次序。处理次序称为分组(Grouping)或关联(association)。

表 1-10　运算符优先级

类　型	级　别	运　算　符	组　次　序
基本运算符	1(最高级)	( ) . [ ] - >	从左至右
一元运算符	2	! ~ - - (type) * & ++ - - sizeof	从右至左
二元运算符	3	* / %	从左至右
算术运算符	4	+ -	从左至右
移位运算符	5	<< >>	从左至右
关系运算符	6	< <= > >=	从左至右
比较运算符	7	== !=	从左至右
位运算符	8	&	从左至右
位运算符	9	^	从左至右
位运算符	10	|	从左至右
逻辑运算符	11	&&	从左至右
逻辑运算符	12	||	从左至右
条件运算符	13	?:	从右至左
赋值运算符	14(最低级)	= + = - = /= *= %= << = >>= &= |=	从右至左

表中的一些运算符将在 1.9 节和 1.10 节中介绍，包括基本运算符，如点运算符(.)、方括号([])，还有间接运算符(->)，它们用来指示标识符的细节信息，例如数组元素或结构元素。还包括指针运算符和间接运算符等，如取内容(*)运算符和取地址(&)运算符。下面的示例将帮助您更好地理解以上内容：

```
y=3+2*4;          //would yield y=11 because the * is higher
                  //in precedence and would be done before the +.
y=(3+2)*4;        //would yield y=20 because the () are the
                  //highest priority and force the + to be done first.
y=4>>2*3;         //would yield y=4>>6 because the * is higher
                  //in priority than the shift right operation.
y=2*3>>4;         //would yield y=6>>4 due to the precedence of the * operator.
```

运算符优先级强制规则的使用与类型转换相似：如果有疑惑，则使用括号来确保数学计算按照所期望的顺序实现。多余的括号不会增加代码的大小，但可以增强其可读性，以及正确执行其任务的可能性。

1.7　控制语句

控制语句用来控制程序的执行流。if/else 语句用根据判断的结果执行给出的两种操作之一。while、do/while 和 for 语句用来控制指令块的重复。switch/case 语句用于允许用单一的判断以一种清晰和简洁的方式将程序流指引到多个可能的指令块之一。

1.7.1　while 循环

while 循环语句很早就出现在 C 语言编程的描述中。它是最基本的控制元素之一。while 语句的格式如下：

```
while(expression)                    或者          while(expression)
{                                                      statement;
    statement1;
    statement2;
    ...
}
```

当程序的执行进入 while 循环的的顶部时，对表达式求值，如果该表达式为真(非零)，则执行 while 循环内的语句。与 while 语句相关的“循环”是包含于 while 语句后面花括号({})内的所有代码行。如果“循环”只有一条语句，则不需要括号，该“循环”就是直接紧跟 while 语句的那条语句。当执行到循环底端时，马上返回到 while 循环的顶部，再次对表达式进行求值。如果值为真，则继续循环，否则完全绕过该循环，而继续执行执行紧跟在 while 循环之后的语句。例如：

```
#include <stdio.h>


void main(void)
{
    char    c;

    c=0;
    printf("Start of program   \n");
    while(c<100)                        //if c less than 100 then ...
    {
        printf("c=%d\n",(int)c);        //print c's value each time through the loop
        c++;                            //increment c
    }
    printf("End of program\n");         //indicate that the program is finished
    while(1)                            //since 1 is always TRUE,then just sit here...
    ;
}
```

本例中，变量c首先初始化为0，打印出字符串Start of program。然后执行while循环，c值从0至99，每次加1，并打印c的值。当c值增加到100时，由于不再小于100，循环结束。打印字符串End of program。此时在while(1)语句上程序无限等待。

while(1)语句的作用十分明显。因为1是需要求值的表达式，并且是一个常量(1总是非零的，因此认为是真)，它永远为真，所以虽然上例循环内没有指令，但是判断表达式一直为真，循环一直进行下去。在这个例子中，while(1)通过无限地执行循环来停止程序的执行，从而让处理器不执行该程序外不存在的代码。

同时注意到在while循环内printf()函数内把字符型变量c转换为整型。这一步是必须的，因为大部分嵌入式C语言编译器的printf()函数只能正确处理整型变量。

while循环还可以用来等待并行端口上事件的发生。

```
void main(void)
{
    while(PINA&0x02)                    //hangs here waiting while the second bit of port A is high
      ;

    while(1)                            //since 1 is always TRUE, then just sit here...
      ;
}
```

在这个例子中，while循环被用来等待端口A的一位下降。对表达式求值是PINA & 0x02，它只获取从端口A读出的数据的二进制第二位，其余位都屏蔽掉了。当这一位为1时，其结果非零(为真)，所以程序一直执行循环，直到第二位的值变为0。在后面，我们将清楚地知道，在实时编程中，这种构造是不合理的，但这是while语句的正确用法。

1.7.2 do/while循环

do/while循环与while循环非常类似。区别在于do/while语句是先执行循环后判断。即循环内的语句至少执行一次，然后再判断是否继续循环。而在while结构中，是在每次执行循环的指令前先进行判断。do/while的形式如下：

```
do                                  或          do
{                                               statement;
    statement1;                                 while(expression)
    statement2;
    .....
}while(expression);
```

当执行到do/while构造的末尾时，对表达式求值，如果求值结果为真(非零)，程序流程返回到do/while循环的顶部。每次执行到达该构造的末尾时，都会对该表达式进行判断。只要表达式为真，就会执行该循环；否则继续执行该循环结构之后的指令。对前面的例子使用do/while循环结构时，如下所示：

```
#include <stdio.h>
```

```
void main(void)
{
        char    c;

        c=0;
        printf("Start of program \n");

        do
        {
            printf("c=%d\n",(int)c);                    //print c's value each time through the loop
            c++;                                        //increment c
                }while(c<100);                          //if c is less than 100 repeat the operation
                printf("End of program\n");             //indicate that the program is finished
                 while(1)                               //since 1 is always TRUE,then just sit here...
                 ;
}
```

在本例中，变量 c 被初始化为 0，打印出字符串 Start of program，然后执行 do/while 循环，c 从 0 到 99，每次加 1，并打印出来。当增到 100 时，不再小于 100，此时循环结束。打印字符串 End of program，然后程序无限等待。

1.7.3　for 循环

典型情况下，for 循环构造用于对语句或语句块执行特定的次数。For 循环可以描述成初始化、判断和导致该条件满足的动作。for 循环语句的格式如下：

```
for(expr1;expr2;expr3)          或者          for(expr1;expr2;expr3)
{                                                       statement;
      statement1;
      statement2;
      .....
}
```

expr1 只在进入 for 循环时执行一次，典型情况下，expr1 是赋值语句，可以用于初始化条件 expr2。

expr2 是条件控制语句，用来决定是否继续循环。expr3 也是赋值语句，用来满足条件 expr2。

当程序的执行进入 for 循环的顶部时，执行 expr1。然后对条件 expr2 进行求值，如果结果为真，执行 for 循环内的语句—— 程序停留在循环内。当执行到达该结构的底部时，执行 expr3，程序流回到 for 循环的顶部，再一次判断 expr2，如果该 expr2 为真值，则执行循环，否则完全绕过该循环，执行其后的语句。For 循环的结构用 while 循环表示如下：

```
expr1;
while(expr2)
{
```

```
        statement1;
        statement2;
        .....

        expr3;
    }
```

下面是一个例子：

```
#include <stdio.h>

void main(void)
{
        char    c;

        c=0;
        printf("Start of program \n");
        for(c=0;c<100;c++)                      //if c less than 100 then ...

        {
                printf("c=%d\n",(int)c);        //print c's value each time through the loop
        }                                       //c++ is executed before the loop returns to the top
        printf("End of program\n");             //indicate that the program is finished
        while(1)                                //since 1 is always TRUE,then just sit here...
                ;
}
```

在本例中，首先打印字符串 Start of program，接下来变量 c 初始化为 0，执行 for 循环，c 从 0 依次增加 1 递增到 99，并把它的值打印出来。当 c 累加到 100，不再小于 100 时，绕过 for 循环。打印字符串 End of program，程序通过 while(1)语句无限等待。

1.7.4　if/else 语句

if/else 语句根据条件语句的求值结果来选择执行路径。

1. if 语句

if 语句形式如下：

```
if(expression)                  或          if(expression)
{                                               statement;
        statement1;
        statement2;
        ....
}
```

如果 if 后的表达式为真(非零)，则执行随后的语句或语句块；如果结果为假，则忽略随后

的语句或语句块。

2. if/else 语句

if/else 语句有如下的形式：

```
if(expression)                    或          if(expression)
{                                                 statement1;
      statement1;                             else
      statement2;                                 statement2;
      ......
}
else
{
      statement3;
      statement4;
      ....
}
```

else 语句向程序流添加了特殊的特征，即只有当表达式的值为假时才会执行与 else 语句相关的语句或语句块。如果表达式的值为真则忽略这些语句块。

一个 else 语句必须有一个 if 语句与之配对。

一种常见的编程技巧就是级联 if/else 语句，形成选择树：

```
if(expr1)
      statement1;
else if (expr2)
      statement2;
else if (expr3)
      statement3;
else if (expr4)
      statement4;
```

这些 if/else 语句序列只选择和执行一条语句。如果第 1 个表达式 expr1 为真，则执行语句 statement1，其余部分忽略。如果 expr2 为真值，则执行语句 statement2，其余部分忽略，依此类推，如果 expr1，expr2，expr3 均为假，则执行 statement4。

下面是一个 if/else 操作的例子：

```
#include <stdio.h>

void main(void)
{
    char c;

    printf("Start of program   \n");
```

```
    for(c=0; c<100; c++)                    //while c less than 100 then ...
    {
        if(c<33)
            printf("0<c<33");               //use if/else to show the range of
        else if((c>32)&&(c<66))             //numbers that c is in.
            printf("33<c<66");
        else
            printf("66<c<100");
            printf("c=%d\n",(int)c);        //print c's value each time
                                            //through the loop
    }

    printf("End of program\n");             //indicate that the program is finished

    while(1)                                //since 1 is always TRUE, then just sit here...
      ;
}
```

在本例中，首先打印出文本字符串 Start of program，接着 c 在 for 循环构造内部被初始化为 0。然后执行 for 循环，在 for 循环结构内 c 从 0 依次增加 1 直到 99，并依次打印其值。

当 c 值小于 33 时，打印文本字符串 0<c<33；当 c 值在 34 和 65 之间时，打印文本字符串 33<c<66；如果 c 值不在上述范围内，打印文本字符串 66<c<100。

当 c 增加到 100 时，for 循环结束。打印文本字符串 End of program，程序进入无限等待状态。

通过目前所学的构造和技术，可以创建一个程序，用于有效地检测输入端口的每一位并打印消息来说明该位的状态。

```
#include <90s8535.h>            //register definition file for an Atmel AT90s8535
#define test_port PORTA

void main(void)
{

    unsigned char cnt,bit_mask;         //variables
    bit_mask= 1;                        //start with lowest bit

    for(cnt=0;cnt<8;cnt++)                  //for loop to test 8 bits
    {
                //the instructions below test port bits and print result
    if(test_prot  &  bit_mask)
        printf("Bit  %d  is high.\n",(int)cnt);
    else
        printf("Bit %d low .\n",(int)cnt);
```

```
        bit_mask << =1;                    // shift bit to be tested
    }

    while(1)                               // since 1 is always TRUE,then just sit here ....
        ;
}
```

本例使用了设置 8 次循环的 for 循环，每一位均被检测。变量 bit_mask 的值从 1 开始，用于屏蔽除了刚检测的位之外的其余位。用作屏蔽后，它使用复合赋值左移一位以检测下一位。在每一次循环期间，if/else 构造均用来打印每一位的正确状态。在 if 构造中的条件语句是一个按位与表达式，在判断中使用按位与运算屏蔽不想要的位。

3. 条件表达式

条件表达式是另一种形式的 if/else 控制语句，可以简化语法和节省程序员的时间。if/else 序列如下所示：

```
if (expression_A)
    statement1;
else
    statement2;
```

它可被简化为如下所示的条件表达式：

```
expression_A?statement1:statement2;
```

在以上两种形式中，都先对逻辑表达式 expression_A 求值，如果值为真，执行 stetement1，否则执行 statement2。

一个程序中，条件表达式可写为如下形式：

```
(x<5)?y=1 ： y=0;    // if x is less than 5,then y=1,else y=0
```

1.7.5　switch/case 语句

switch/case 语句用于根据表达式的值来选择执行一条语句或一组语句。形式如下：

```
switch(expression)
{
    case const1:
        statement1;
        statement2;
    case const2:
        statement3;
        .....
        statement4;
    case constx;
        statement5;
```

```
        statement6;
    default:
        statement7;
        statement8;
}
```

表达式 expression 的值与常量(const1、const2、...constx)进行比较，执行与表达式的值相匹配的常量之后的语句。常量必须是整数或字符值。上面的代码中与常量匹配的所有语句都将执行，直到 switch 结构结束。通常这并不是所需的操作，所以，break 语句放在每个语句块的末尾，可以防止执行与常量匹配的所有语句，并允许程序流在 switch 结构后继续。

default 语句是可选的，但是它接受在没有匹配常量时需要执行的语句。

```
#include <stdio.h>
#include <90s8535.h>            //register definition header file for an Atmel AT90s8535
void main(void)
{
    unsigned char c;
    while(1)
    {
        c=PINA & 0xf;           //read the lower nibble of port A
        switch(c)
        {
            case '0':
            case '1':                   //you can have mutiple casess
            case '2':                   //for a set of statements.
            case '3':
                printf("c is a number less than 4 \n");
                break;                  //break to skip out of the loop
            case '5':                   // of just one is ok.
                printf(" c is a 5 \n");
                break;
            default:
                printf("c is 4 or 5 is >5 \n");
        }
    }
}
```

这个程序先读取端口 A 上的值，屏蔽高四位。并将低四位与常量比较，如果字符是 0、1、2 或 3，那么文本字符串 c is a number less than 4 将输出到标准输出设备上；如果字符为 5，输出文本字符串 c is a 5；如果字符不是上述这些(是 4，或大于 5)，则执行 default 语句，输出文本字符串 c is 4 or is >5。一旦执行了适当的 case 语句，程序将返回到 while 循环的顶部，然后开始重复。

1.7.6　break、continue 和 goto 语句

break、continue 和 goto 语句用来修改 for、while、do/while 和 switch 语句的执行。

1. break 语句

break 语句用来从 for、while、do/while 和 switch 语句中退出。如果这些语句嵌套在其他语句内部，那么 break 语句会只从它当前的语句块中退出。

下面的程序将 c 的值打印到标准输出设备，c 值连续地从 0 递增到 100，然后重新初始化为 0。在内 while 循环中，c 递增到 101，然后执行 break 语句，退出内层 while 循环，继续执行外层 while 循环。在外层 while 循环中，c 被设为 0，又返回到内循环。程序将永远反复循环下去。

```
#include <stdio.h>

void main(void)
{
    int c;
    while(1)
    {
        while(1)
        {

            if(c>100)
                break;              //this will take us out of this while
                ++c;                //bolck ,clearing c...
            printf("c=%d\n",c);
        }
            c=0;                    //clear c and then things will begin again.
                                    //printing the values 0～100,0～100,etc..
    }
}
```

2. continue 语句

continue 语句允许程序重新开始进行下一次 while、do/while 或 for 循环。continue 语句在停止循环语句的执行方面与 break 有点类似。区别在于 continue 语句重新从顶部开始循环，而 break 语句完全退出循环。

```
#include <stdio.h>

void main(void)
{
    int c;

    while(1)
    {
```

```
        c=0;
        while(1)
        {

            if(c>100)
                continue;          //this will cause the printing to stop when c>100,because the
                                   //continue will cause the rest of this loop to be skipped!!
            ++c;
            printf("c=%d\n",c);    //no code after the continue will be executed
         }
    }
}
```

在本例中，c 值会显示出来，直到 100。此时，程序好像停止，但实际上仍在运行。只是它跳过了递增语句和 printf()语句。

3. goto 语句

goto 语句用于准确地将程序的执行直接跳到带有标签的语句，并执行该语句。goto 语句不利于结构化编程，但在嵌入式系统中可以很好地节省代码和存储器。标签可以在 goto 语句的前面和后面，形式如下：

```
goto identifier;    或者      identifier:
  …                             statement:
identifier:                       …
statement                     goto identifier;
```

标签是有效的 C 名称或标识符，后面跟冒号(:)。

```
#include <stdio.h>

void main(void)
{
    int c,d;
    while(1)
    {
    start_again;

        c=0;
        d=-1;
        while(1)
        {
            if(d==c)
                goto start_again;      //(stuck? bail out !)
            d=c;
            if(c>100)
                continue;              //this will reinitiate this while loop
```

```
                ++c;                    //d will be checked to see if c is stuck
                printf("c=%d\n",c);     //because c won't change(>100!!)
            }
        }
    }
```

在本例中，c 和 d 初始化为不同的值。if (d==c)语句检测两变量的值，只要不相等，马上执行语句 d=c。如果 c 小于等于 100，执行 c 递增 1 的操作，输出 c 值，重新进行 while 循环。c 和 d 的值一直相差 1 直到 c 值大于 100。当 c 等于 101 时，continue 语句会忽略++c 和 printf() 语句。如果 if (d==c)为真，两值相等执行 goto 语句，程序跳转到 start_again 标签。结果为一遍又一遍的重复显示出 c 的值，从 0 到 100。

第 1 章示例项目：A 部分

本部分为一个项目的第一部分，用来演示第 1 章所学的概念。此例基于一个简单的游戏：假设您的老师要求您建立一个便携式投币自动售货机，作为期中测试项目。具体说明如下：

- 按钮的按下和松开用于“拉吃角子老虎机的臂”(译者注：拉吃角子老虎机有一臂状杆，操纵使币落入槽口)
- 按下和松开期间用于产生一个伪随机数来建立三列图形
- 闪光灯用来指示机器在“移动”
- 一列中可以显示 4 种图形：酒吧(Bar)、铃声(Bell)、柠檬(Lemon)和樱桃(Cherry)
- 机器提供的报酬方式有如下规则：
  - 3 个同一种类的给予 5 美分
  - 一个樱桃给予 10 美分铸币
  - 3 个樱桃获得头奖
  - 其他东西则不会获得报酬

首先开发一简短的程序，涉及到一些学过的概念。其中，main()函数包含多种类型的控制语句来实现预期的操作。首先是按下和松开算出一个伪随机数，使用标准输入输出函数以数字形式显示出结果。示例代码中示范了 while、do/while、for 循环以及 if/else 控制语句的使用。

```
//
//"Slot Machine" ---The Mini-Exercise
//
//Phase 1....

#include <90s8515.h>          /*processor specific information*/

#define xtal 4000000L         /*quartz crystal frequency [Hz]*/
#define baud 9600             /*Baud rate*/

#include <stdio.h>            /* this gets the printf() definition */
// Global Variable Declarations..
char    first,second,third;   // Columns in the slot machine.
```

```
char    seed;           // Number to form a seed for random #s.
char    count;          // General purpose counter.
int     delay;          // A variable for delay time.

void main(void)
{
    PORTA = 0xFF;       // Initialize the I/O ports
    DDRA = 0x00;        // port A all inputs
    DDRB = 0x00;        // port B.1 is an output (light)
    DDRB = 0x02;

//  Initialize the UART control register
//  RX & TX enabled, 8 data bits
    UCR=0x18;
//  initialize the UART's baud rate
    UBRR=xtal/16/baud-1;

    while(1)            // do forever..
    {
        while(PINA.0)   // Wait for a button and
            seed++;     // let the counter roll over and
                        // over to form a seed.

        first = second = third = seed; // preload the columns

        do                          // mix up the numbers
        {                           // while waiting for button release.
            first ^= seed>>1;       // Exclusive ORing in the moving seed
            second^= seed>>2;       // can really stir up the numbers.
            third ^= seed>>3;
            seed++;                 // keep rolling over the seed pattern
        } while(PINA.0 == 0); // while the button is pressed

        for(count=0;count<5;count++)    //flash light when done..
        {
            for(delay=0;delay<10000;delay++)
                ;                       //just count up and wait
            PORTB.1=0;                  //turn the LED   on.
            for(delay=0;delay<10000;delay++)
                ;
            PORTB.1=1;                  //turn the LED   off..
        }
            first &=3;                  //limit number to values from 0 to 3
            second &=3;
            third &=3;
```

```
                                        //show values..
                printf("->%d,%d,%d,<-\n",first ,second,third);

                                        //determine a payout..
                if((first==3) && (second==3) && (third==3))
                    printf("Paid out:JACKPOT!!\n");          //Three "Cherries"
                elseif((first==3) || (second==3) || (third==3))
                    printf("Paid out:One Dime\n");           //One "Cherry"
                elseif((first==second) && (second ==third))
                    printf("Paid out:One Nickle\n");         //Three of a kind
                else
                    printf("Paid out:ZERO\n");           //Loser...
        }
    }
```

1.8　函数

函数是语句块的一种封装，可以在程序中多次使用。有些语言把函数定义为子例程或过程。函数用来将一个程序的操作元素分成多个基本的部分。这样使程序员可以调试每个元素，然后反复使用。

函数的一个主要优点就是可以作为库的扩展，设计函数的目的是为了执行特定的任务，可对其加以保存，以便以后供其他应用程序或其他程序员使用。这样可节省时间，维持稳定性，因为所开发的函数可以重用与测试。

函数可以在没有任何参数的情况下执行一项独立的任务；也可以接受参数，以便在执行其事先设计好的任务时提供指导。函数不仅可以接受参数，还可以返回值；它可能接收多个参数，但返回值只有一个。一些例子如下：

```
sleep() ;                   //this function perform a 1 second delay
                            //with no parameters, in or out

printf("this is a parameter %d ",x);
                            //printf will print its parameters

c=getchar();
                            //getehar will return a value from the standard input
```

函数的标准形式是：

```
type function_name(type param1,type param2,...)
{
    statement1;
    statement2;
    ...
```

```
    statementx;
}
```

函数的标准形式是类型加函数名，再加基本运算符()。括号(即函数运算符)向编译器指明这个名字是作为一个函数执行的。

函数的类型和参数可为有效的可变类型，如整型、字符型或浮点型，也可为空或 void 类型。void 类型是一种有效的类型，用于指明参数或返回值为零大小(zero size)。函数的默认类型为整型，所以一个典型的函数声明如下：

```
unsigned char getchar(void)
{
    while((UCSRA & 0x80)==0)
        ;                    //wait for a character to arrive
    return UDR;              //return its unsigned char value to the caller
}
```

在本例中，getchar()函数不需要任何参数，在执行结束时返回一无符号字符值。getchar()函数是 C 编译器提供的库函数。这些函数对程序员很有用，将在后面更进一步讨论这些函数。

1.8.1　原型和函数组织

正如使用常量和变量一样，函数类型和函数的参数类型须先声明后使用。这可通过两种方法实现：一种是函数的声明顺序，另一种是函数原型的使用。

排列函数的顺序是一种很好的方法。它使编译器在使用前获得函数的所有信息。也就意味着程序有如下的格式：

```
//declaration of global variable would be first
int var1,var2,var3;

//declaration of functions would come next
int function1(int x,int y)
{
}

void function2(void)
{
}

//main would be built last
void main(void)
{
    var3=function1(var1,var2);
    function2();
}
```

这种次序很好，但有时不太可能。在很多情况下，函数通过调用另一个函数来完成任务，

而不全都以这种自顶向下的顺序来声明这些函数。

函数原型用于让编译器在函数实际声明之前知道函数的类型和需求，减少了自顶向下顺序的需要。

上例可按如下方式进行组织：

```
int var1,var2,var3;            //declaration of global variable

void main(void)                //main
{
      //these functions are not yet known to the compiler
      var3=function1(var1,var2);

      function2();
}

//declaration of functions here now generate a
//"Function Redefined Error"
int function1(int x,int y)
{
}

void function2(void)
{
}
```

典型情况下，这种组织方式会导致编译器发出错误消息。编译器没有 main()函数中调用的函数或者其格式的足够信息，从而不能生成正确的代码。函数原型可按如下方式加在代码的顶部：

```
//the prototype of a function tells the compiler what to expect
int function1(int,int);
void function2(void);

int var1,var2,var3;            //declaration of global variable
void main(void)                //main
{
      var3=function1(var1,var2);
      function2();
}
//the declaration of functions here is perfectly OK,since
//the format of the function are presented in the prototypes
int function1(int x,int y)
{
}
```

```
void function2(void)
{
}
```

此时，编译器在处理函数名时有每个函数所有必要的信息。只要实际声明的函数与原型匹配，程序就将很好地执行。

C 编译器提供了许多库函数，通过 include<filename.h>语句，可以在程序中使用这些库函数。其中 filename.h 是包含了库函数的函数原型的文件。前面的例子中，使用了 printf()函数，该函数本身在别处声明，但它的原型在头文件 stdio.h 中。这样可让程序员可以在程序的开头包括头文件，然后开始编写程序代码。每个库函数在附录 A 中进行了定义。

1.8.2 函数返回值

在很多情况下，设计函数是为了完成特定的任务并从任务中返回值或状态。控制字 return 用于获得返回值并指明函数的出口点。如果为非 void 类型函数，调用程序将获得返回值。

在返回值为类型 void 的函数

```
int z;              //global variable z

void sun(int x,int y)
{
    z=x+y;                          //z is modified by the function ,sum()
}
```

中，return 隐含在函数的末尾。如果将 return 放在 void 函数中，例如：

```
void sum(int x, int y)
{
   return ;              //this would return nothing..
   z=x+y;                //and this would be skipped
}
```

return 语句将使程序的执行退出到调用程序，return 语句后的部分不会执行。

如果函数的类型不是 void，return 语句也会使程序退出本函数返回到调用者。另外 return 将表达式的值置于 return 的右边，以在函数声明中指定类型的形式放在堆栈上。下面的函数类型为 float，所以，当 return 执行时，float 放到了堆栈上。

```
float  cube  (float v )
{
   return (v*v*v);          //this return a type float value
}
```

函数有返回值可使它作为表达式的一部分，例如在下面的赋值程序中。

```
void main(void)
{
```

```
    float a;

    b=3.14159;                          //put PI into b

    a=cube(b);                          //pass b to the cubed function ,and
                                        //assign its return value to a
    printf("a=%f,b=%f \n", a, b);       //print the result

    while(1)                            //done
        ;
}
```

变量 a 将赋值为函数 cube(b)的返回值。结果如下：

```
a=31.006198,b=3.14159
```

1.8.3　递归函数

递归函数是调用该函数本身的函数。生成递归代码的能力是 C 语言强大的一面。当调用一个函数时，被调用函数的局部变量置于堆栈或堆空间中，与调用者的地址在一起，所以它知道如何返回。每次函数被调用，都会重新进行这些分配。这样使函数可重新进入，因为以前的调用仍留在以前的分配地址，而没有发生改变。

一个最常见的递归操作是求阶乘。例如：

```
5!=5*4*3*2*1=120。
```

在代数学上的表示为：

n!=n*(n-1)*(n-2)..*2*1 或 n!=n*(n-1)!，所以演示该操作的程序如下：

```
#include <stdio.h>

int fact(int n)
{
    if(n==0)
        return 1;                       //if n is zero ,return a one，by
                                        //definition of factorial
    else
        return(n*fact(n - 1));          //otherwise ,call myself with n - 1 until n=0, then
                                        //return n*the result to the call to myself
}

void main(void)
{
    int n;

    n=fact(5);                          //compute 5! ,recursively
```

```
    printf("5!=%d\n",n);          //print the result
    while(1)                      //done.
        ;
}
```

当函数 fact()被参数 n 调用时，它调用本身 n - 1 次。每调用其本身一次，n 的值就减 1。当 n 等于 0 时，函数不再调用本身，而是返回值 1。这样将引起一个返回的多米诺效应或链锁反应，从而导致将返回到 main()中执行的调用，在 main()函数中打印返回结果。

递归在阶乘运算、快速排序和链表搜索中都有强大的应用。它同时也是一个十分消耗存储器的操作。每次函数调用其本身，都需要重新分配函数的局部变量、返回值、返回地址和调用过程中用到的参数。在前例中包括：

int n (passed parameter)	2 bytes
return value	2 bytes
return address	2 bytes
Total	6 bytes, per recursion ,minimum

如果是求 5!，则至少需要 30 字节的存储器空间来在求阶乘操作中进行分配和根据其返回进行分配。这样会使操作变得十分危险，在小的微控制器中是不可能实现的。如果由于递归而进行分配的深度太大，堆栈或堆将会溢出，程序的操作将变得难以预测。

第 1 章示例项目：B 部分

对于便携式自动售货机的开发，将设计两个函数，将主循环分解为小的代码块并使代码更具可读性。发光部分和计算报酬方式部分被移入到函数中实现，缩短了 main()函数并使其更易懂。switch/case 语句应用于付款显示。

```
//
// "Slot Machine"-The Mini-Exercise
//
// Phase 2..

#include <90s8515.h>          /* processor specific information */

#define xtal 4000000L         /* quartz crystal frequency [Hz] */
#define baud 9600             /* Baud rate */

#include <stdio.h>            /* this gets the printf() definition */

// Global Variable Declarations..
char    first,second,third;       // Columns in the slot machine.
char    seed;                     // Number to form a seed for random #s.
char    count;                    // General purpose counter.
int     delay;                    // A variable for delay time.
```

```
#define JACKPOT     3          /* this defines give different payouts */
#define DIME        2          /* names in order to make the code more */
#define NICKEL      1          /* readable to humans */
#define ZERO        0

void flash_lights(char n)     // flash the lights a number of times
{
        for(count = 0; count < n; count++)   // flash light when done..
        {
            for(delay = 0; delay < 10000; delay++)
                ;                          // just count up and wait
            PORTB.1 = 0;                   // turn the LED on..
            for(delay = 0; delay < 10000; delay++)
                ;
            PORTB.1 = 1;                   // turn the LED off..
        }
}

int get_payout(void)
{
    if((first == 3) && (second == 3) && (third == 3))
        return JACKPOT;     // if all "cherries"..
    else if((first == 3) || (second == 3) || (third == 3))
        return DIME;    // if any are "cherries"..
    else if((first == second) && (second == third))
        return NICKEL;    // if three are alike, of any kind..
    else
        return ZERO;    // otherwise - you lose..
}

void main(void)
{
     int i;                    // declare a local variable for temporary use..

     PORTA = 0xFF;             // Initialize the I/O ports
     DDRA = 0x00;              // port A all inputs
     DDRB = 0x00;              // port B.1 is an output (light)
     DDRB = 0x02;

  //  Initialize the UART control register
  //  RX & TX enabled, 8 data bits
     UCR=0x18;
  //  initialize the UART's baud rate
     UBRR=xtal/16/baud-1;
```

```
 while(1)          // do forever..
{
     while(PINA.0)          // Wait for a button and
          seed++;           // let the counter roll over and
                            // over to form a seed.

     first = second = third = seed; // preload the columns

     do                     // mix up the numbers
     {                      // while waiting for button release.
          first ^= seed>>1;      // Exclusive ORing in the moving seed
          second^= seed>>2;      // can really stir up the numbers.
          third ^= seed>>3;
          seed++;                // keep rolling over the seed pattern
   } while(PINA.0 == 0); // while the button is pressed

   flash_lights(5);         // flash the lights 5 times..

   first  &= 3;             // limit number to values from 0 to 3
   second &= 3;
   third  &= 3;
                            // show the values..

   printf(" - > %d, %d ,%d < - \n", firest, second, third);

                            // determine a payout..
     i = get_payout();

     switch(i)          // now print the payout results
     {
               case      ZERO:
                    printf("Paid out: ZERO\n");
                    break;

               case      NICKEL:
                    printf("Paid out: One Nickle\n");
                    break;

               case      DIME:
                    printf("Paid out: One Dime\n");
                    break;

               case      JACKPOT:
                    printf("Paid out: JACKPOT!!\n");
                    break;
```

```
        }
    }
}
```

1.9　指针和数组

指针和数组在 C 语言中有着广泛的应用，因为它们允许程序执行更一般化的和更有效率的操作。要求聚集数据的操作可以通过指针或数组来更方便地访问和操纵数据，而不需移动存储器中的数据。指针和数目同样也适用于关联变量的分组，例如通信缓冲区和字符串。

1.9.1　指针

指针是包含变量、常量、函数和数据对象的地址与存储单元的变量。通过间接访问操作符*，变量被声明为指针变量：

```
char   *p;   // p is a pointer to a character
int    *fp;  // fp is a pointer to an integer
```

指针数据类型在内存中分配足够大的空间，以容纳变量的机器地址。例如一个典型的微控制器的存储器分配地址被描述为 16 位。所以在典型的微控制器中，指向字符的指针也为 16 位的值，尽管字符本身只有 8 位。

一旦声明了指针，即可处理指针所指向的变量地址，而不是变量的值本身。您需分清地址和地址的内容。取地址运算符(&)获取变量地址，该地址可能被设计为指针，是指针的值。间接引用运算符(*)获得指针指向的地址的内容。例如：

```
char *p ;    //p is a pointer to a character
char a, b ;  // a and b are characters

p=&a ;       // p is now pointing to a
```

在本例中，p 赋值为变量 a 的地址，所以说 p 指向 a。为了取得指针变量 p 所指向的值，使用间接运算符(*)。当执行时，间接运算符将导致 p 的值(一个地址)被用来查找存储器中的某个位置。根据包含间接操作符的表达式，会读出或写入该位置的值。于是，在如下的代码中，*p 会读取包含于 p 中的地址处的值，然后赋值给变量 b。

```
b=*p;      //b equals the contents pointed to by p
```

因此，上面两例的代码产生同样的结果：b＝a。间接运算符同样也可在赋值运算符的右边，例如：

```
char *p;                    //p is a pointer to a charater
char a,b;                   //a and b are charaters

p=&a;
*p=b;                  //the location pointed to by p ,is assigned the value of b
```

在内存中，保存在 p 中的地址被赋值为变量 b 的值，这会产生相同的效果 a=b。当您理解这些操作时，应当样去理解：b 被赋值为 p 所指向的值(内容)，p 被赋值为 a 的地址。这样有助于避免犯最常见的错误：

```
b=p ;          // b will be assigned a value of p, an address , not what p points to.

p=a ;          // p will assigned the value of a, not its address.
```

上面的两条赋值语句在语法上是对的，但不是我们想要的结果。强大的能力和简单性可能会引起巨大的和简单的错误。指针操作可能会破坏程序的正常工作。但如果小心一点并多了解语法，可大大减少不小心改变存储器的危险。

指针还是一种访问系统中的外围设备(比如输入输出端口)的优秀方法。例如，我们有一个 8 位的并行端口输出位于存储器中的 0x1010，则可以通过间接引用运算符来访问该端口：

```
unsigned char *out_port      //declare out_port as a pointer

out_port=0x1010;             //assign out_port with the address value

*out_port=0xaa;              //now assign out_port's address a value
```

在本段代码中，由 out_port 指向的地址被赋值为 0xAA。同样也可描述为：任何赋给*out_port 的值都被写入存储器地址 0x1010 中。

在 C 语言结构中，还有指向指针的指针。实际上，对于这种类型的间接深度并无限制，只要不会引起混乱即可。例如：

```
int *p1;                     //p1 is a pointer to an integer
int **p2;                    //p2 is a pointer ,to a pointer to an integer
int ***p3;                   //p3 is a pointer ,to a pointer , to a pointer
                             //to an integer
int i,j;
p1=&i;                       //p1 is assigned that address of i
p2=&p1;                      //p2 is now pointing to the pointer to i
p3=&p2;                      //p3 is pointing to the pointer that is pointingtoi

                             //therefor,
j=***p3;                     //j is assigned the value pointed to by the value
                             //pointed to by the value pointed to by p3
```

其结果为

```
j=i;      //any question??
```

由于指针是高效的寻址方式，它可以通过很少的指令来移动、复制和改变存储器。当进行地址运算时，C 编译器确认正确的地址是基于变量的类型或被寻址的常量来进行计算的。例如：

```
int *ptr;
```

```
long *lptr;

ptr=ptr+1;          //move the pointer to the next integer
                    //(2 bytes away)

lptr=lptr+1;        //move the pointer to the next long integer
                    //(4 bytes away)
```

ptr 和 lptr 都自增一个位置，实际上 ptr 移动 2 字节，而 lptr 移动 4 字节，这是因为它们后来的类型不同。自减操作同样如此。

```
ptr+ +;             //move the pointer to the next integer location
                    //(2 bytes away)

- lptr;             //move the pointer to the preceding long integer location
                    //(-4 bytes)
```

因为间接引用运算符(*)和取地址运算符(&)都是一元运算符，具有最高优先级，它们在表达式中总是优先操作。自增运算符和自减运算符也是一元运算符，具有同样的优先级，包含这些运算符的表达式从左至右求值。例如，下面的自增操作(包括前自增和后自增操作)是赋值操作的一部分。请注意到每行中的注释，这些注释是关于相同的优先级怎样影响操作结果的。

```
char c;
char *p;

c=*p++;             //assign c the value pointed to by p ,and then
                    //increment the address p
c=*++p;             //increment the address p ,then assign c the
                    //value pointed to by p

c=++*p              // increment the value pointed to by p ,then assign it to c ,leaving the
                    // value of p untouched

c=(*p)++;           //assign c the value pointed to by p ,and then
                    //increment the value pointed to by p leaving
                    //the value of p untouched
```

指针可用来扩充函数返回的信息数目。使用 return 控制语句函数本身只能返回一个值，如果将参数作为指针传递给函数，则允许函数返回附加的值。考虑下面的 swap2()函数：

```
void swap2(int *a, int *b)
{
    int temp;

    temp=*b;        //place value pointed to by b into temp
    *b=*a;          //move the value pointed by a into location
```

```
                    // pointed to by b
        *a=temp;    //now set the value of location a to the
                    //value of temp
}
```

本例中的函数交换 a 和 b 的值。调用者提供要换位的变量的指针，如下：

```
int v1, v2 ;

swap2(&v1, &v2) ;          //pass the addresses of v1 and v2
```

因 swap2()函数使用了传递给它的地址，直接交换变量 V1 和 V2 的值。传递指针的过程经常使用，在标准类库函数(如 scanf())中也是这样。在 stdio.h 中定义的 scanf()函数允许从标准输入中以格式化的方式接收多个参数，存储在存储器中指定的位置。对 scanf()函数的典型调用如下：

```
int x, y ,z;

scanf("%d %d %d",&x ,&y ,&z );
```

这一 scanf()函数将从标准输入中接收 3 个十进制整数值，并放入变量 x、y 和 z 中。scanf()函数的详细介绍请参见第 3 章或附录 A。

1.9.2　数组

数组是另一种间接引用系统。数组是已声明类型的有序数据的集合。数组声明同常量或变量声明一样，只不过需要指定数组元素的个数：

```
int digits[10] ;  //this declares an array of 10 integers
char str[20] ;    //this declares an array of 20 characters
```

数组元素的引用是通过索引或下标实现的。索引的范围从 0 到数组长度减 1。

str[0],　str[1],　str[2], …… str[19]。

定义声明可以包括初始化。在变量数组中，初始化值将放在程序的存储器区域中，在 main()函数执行前复制到实际的数组中。常量数组则不同，其值置于程序存储器中，以节省 RAM 内存空间，而 RAM 存储器在微控制器中容量都比较小。一个典型的初始化数组会如下所示：

```
int array[5]={12, 15, 27, 56, 94};
```

此时 array[0]=12、array[1]=15、array[2]=27、array[3]=56、array[4]=94。

C 语言不提供数组的边界检测，如果索引值超过了数组边界，存储器将会以不可预测的方式改变而导致不可预测的结果。例如

```
char digits[10]={0,1,2,3,4,5,6,7,8,9}; //an array of characters

numb=digits[12];   //this read outside the array
```

数组存储在存储器中的连续存储单元中。读 digits[5]操作会使处理程序进入数组第 1 个索引的存储单元，然后读取索引位置 6 上的数据。因此第 2 行的代码将会让处理器读取索引位置 13 上的数据。在这个存储单元的数据会赋值给 numb，但会出现奇怪的结果。所以，正如其他编程领域一样，必须小心和事先考虑。

使用数组和指针最大的区别在于：数组在存储器中分配了实际存在的区域，而指针只分配了地址索引(地址引用存储单元)，由程序员声明和定义实际的存储器访问区域(变量)。

最常见的数组类型是字符数组，又称为串或字符串。字符串变量被定义为字符数组，而常量字符串是通过用引号包括一行文本来进行声明的。在常量字符串中，C 编译器自动使用空的终结符，或者将零添加到字符串结尾。当您声明字符串、常量或变量时，声明的数组大小至少需比内容的大小多 1，以便允许使用空终结符。

```
char str[12];                          //variable string

printf("Hello World!");                //constant string in program memory

const cstr[16] ="Constant String";     //constant string in program memory
```

本例中，cstr 设定容纳 16 个值，因为字符串本身有 15 个字符，另一个为终结符。

数组名后的索引可引用任何类型的数组元素。它同样可只用一个名字来引用一个数组的第 1 个元素。没有索引时，数组名作为数组的第一个元素的地址。考虑下面的声明：

```
char stng[20];
char *p;
```

赋值语句

```
p=stng;
// p is pointing to stng[0]
```

等同于

```
p=&stng[0];  //p is pointing to stng[0]
```

字符串通常需要按照逐个字符的方式进行处理，将消息发送到串行设备或液晶显示器(Liquid crystal display，LCD)时就需要这样。下面的例子将演示数组索引和指针间接引用的可互换性。在这个例子中，使用库函数 putchar()来将一个字符发送到很可能是一个串行行端口的标准输出设备：

```
#include <stdio.h>

const char s [15]={" This is a test"};

char i;
char *p;
```

```
void main(void)
{
    for(i=0;i<15;i++)              //print each character of the array
        putchar(s[i]);             //by using i as an index
    p=s;                           //point to string as a whole
    for(i=0;i<15;i++)              //point each character of the array
        putchar(*p++)              //and move to the next element
                                   //by incrementing the pointer p
    while(1)
        ;
}
```

程序的第一部分使用 for 循环来单个地输出数组的每个字符。使用 for 循环计数器作为索引来检索该数组的每一个字符，以便它可以被传递到 putchar()函数。第二部分使用指针来访问数组元素。p=s；让指针来访问该数组第一个元素的存储单元。然后 for 循环使用指针(每一次后加)来获得数组元素，并将其传递给 putchar()。

1.9.3 多维数组

C 语言支持多维数组。在多维数组声明后，应将其视为数组的数组。多维数组可构造为二维、三维、四维或更多维。相邻的存储器存储单元总是由声明的最右边的索引所引用。

典型的二维整型数组声明如下：

```
int two_d[5][10];
```

在存储器中，数组元素将被存储为如下的序列：

```
two_d[0][0], two_d[0][1], two_d[0][2], ... two_d[0][9],
two_d[1][0], two_d[1][1], two_d[1][2], .... two_d[1][9],
two_d[2][0], two_d[2][1], two_d[2][2], .... two_d[2][9],
two_d[3][0], two_d[3][1], two_d[3][2], .... two_d[3][9],
two_d[4][0], two_d[4][1], two_d[4][2], .... two_d[4][9]
```

当二维数组被初始化时，其布局也是一样的，序列如下：

```
int matrix[3][4]={0, 1,  2, 3,
                  4, 5,  6, 7,
                  8, 9, 10, 11 };
```

多维数组对于矩阵运算、过滤器和查找表(look-up-table，LUT)十分有用。

例如我们假定有一个电话键区，当键按下时，产生一行和一列读数或扫描码。二维数组可用作查找表格，将扫描码转变为按键的实际 ASCII 码字符。我们将在本例中假定例程 getkeycode()对键进行扫描，并设置 row 和 col 的值，以指示按键位置。执行这种转换操作的代码可能如下所示：

```
#include <stdio.h>
```

```
#define     TRUE 1;
#define     FALSE 0;
char getkeycode(char *row ,char *col)
      //the getkeycode routine gets the key press and
      //return TRUE if a key is pressed ,FALSE if not

/*look up table for ASCII values*/
const char keys[4][3]={   '1', '2', '3',
                          '4', '5', '6',
                          '7', '8', '9',
                          '*', '0', '#'};
void main(void)
{
      char row,col;
      char i;

      while(1)              //while forever....
      {
          i=getkeycode(&row ,&col);
              //TRUE,if there was a key pressed
              //and row and col contain which key
          if(i==TRUE)
              //only print the key value that is pressed
              putchar(keys[row][col]);
       }
}
```

另一种更为常见的二维数组形式是字符串数组。该例声明了一个串数组，这个串数组被初始化为一周的每一天：

```
char day_of_the_week[7][10]={
                "Sunday",
                "Monday",
                "Tuesday",
                "Wednesday",
                "Thursday",
                "Friday",
                "Saturday"};
```

在这个数组中，字符串具有不同的长度。编译器不管字符串长度如何，均在字符串最后一个字符后置一空终止符。存储器中多余的存储单元未被初始化。printf()等函数遇到空终止符时才停止打印字符串，因此可以正确打印出各种长度的字符串。

为了访问一周中第四天并打印相关联的字符串，使用 printf()函数。printf()函数需要字符串第 1 个字符的位置：

```
printf("%s",&day_of the_week[3][0]);   //prints Wednesday
```

字符串的名字被认为是字符串中第 1 个字符的地址。因为规定只有第一维才可以有效地引用作为第二维的整个字符串，所以，以上的数组可这样来进行访问：

```
printf("%s",day_of _the_week[3]);    //prints Wednesday
```

1.9.4　指向函数的指针

指针和间接引用一个更加强大的功能在于它们可以应用于函数。使用指向函数的指针允许函数从查找表操作结果中进行调用。指向函数的指针也允许函数按引用方式传递给其他函数。这可以被用来创建动态的执行流，有时也称为“自修改”代码。

下面是一个从函数指针表中调用函数的例子，使用 scanf()函数从标准输入中得到输入值。该值受到检查，以便确保它处在 1～6 范围内。如果输入值是正确的，则 func_number 作为函数指针数组的索引。在 func_number 索引处的数组被赋值为 fp，fp 是指向返回类型为 void 的函数的一个指针。

1.8 节中提到的函数标准形式为：

```
type function_name (type param1 ,type param2, ...)
{
    statement1;
    statement2;
        ...
}
```

首先是类型，然后是函数名与基本运算符()。括号(即函数运算符)告诉编译器该标识符应当作为函数。在这种情况下，fp 包含函数的地址。添加间接运算符，以获得指针 fp 指向的函数的地址。现在函数可通过添加函数运算符这一简单的方式进行调用：

```
(*fp)();    //execute the function pointed to by fp
```

以下是完整的程序：

```
#include <stdio.h>

void do_start_task(void)
{
   printf("start selected\n");
}

void do_stop_task(void)
{
   printf("stop selected\n");
}

void do_up_task(void)
```

```
{
   printf("up selected\n");
}

void do_down_task(void)
{
   printf("down selected\n");
}

void do_left_task(void)
{
   printf("left selected\n");
}

void do_right_task(void)
{
   printf("right selected\n");
}

void (*stack_list[6])(void)={
  do_start_task;
  do_stop_task;                          //*an array of pointers to functions */
  do_up_task;
  do_down_task;
  do_left_task;
  do_right_task;
};

void main(void)
{
      int func_number;
      void(*fp)(void);              /*fp is a pointer to a function*/

      while(1)
      {
            printf("\n select a function 1-6:");
            scanf("%d",&func_number);

            if((func_number>0)&& (func_number<7))
            {
                  fp=task_list[func_number - 1];
                  (*fp)();
                  /* assign the function address to fp*/
            }                    /*and call the selected function*/
      }
}
```

第 1 章示例项目：C 部分

在便携式自动售货机的开发的这一阶段，我们使用枚举类型和多维数组来润色项目的外观。我们不是打印表示列值的数字，也不应用报酬计算中的数字，而是为这些值命名。

```
//
//   "Slot Machine" -- The Mini-Exercise
//
//     Phase 3..

#include <90s8515.h>          /* processor specific information */

#define xtal 4000000L         /*  quartz crystal  frequency  [Hz]  */
#define baud 9600             /* Baud rate  */

#include <stdio.h>            /*  this gets  the printf()  definition */

// Global Variable Declarations..
char        first,second,third;      // Columns in the slot machine.
char        seed;                    // Number to form a seed for random #s
char        count;                   // General purpose counter.
int         delay;                   // A variable for delay time.

#define JACKPOT     3          /* this defines give different payouts */
#define DIME        2          /* names in order to make the code more */
#define NICKEL      1          /* readable to humans */
#define ZERO        0

enum { BAR,   BELL,   LEMON,   CHERRY };      // the values for each kind


char    kind[4] [7]   =   {                   //   for kinds of names..
        "BAR",
        "BELL",
        "LEMON",
        "CHERRY"
};

void flash_lights(char n)        //   flash the lights a number of times
{
            for(count = 0;   count < n;   count++)   //   flash light when done..
            {
                for(delay = 0;   delay < 10000;   delay++)
                    ;                           //   just count up and wait
                PORTB.1 = 0;                    // turn the LED on..
                for(delay = 0;   delay < 10000;   delay++)
                    ;
                PORTB.1 = 1;                    // turn the LED off..
        }
```

```
}

int get_payout (void)
{
    if((first == CHERRY)  &&  (second == CHERRY)  &&  third == CHERRY))
        return JACKPOT;        // if all  "cherries"..
    else if((first == CHERRY)  ||  (second == CHERRY  || (third == CHERRY))
        return DIME;    //  if any are  "cherries"..
    else if((first == second)  &&  (second == third))
        return NICKEL;    // if three are alike,  of any kind..
    else
        return ZERO;    // otherwise - you lose..
}

void main(void)
{
        int i;                          // declare a local variable for
                                        //  temporary use..

        PORTA = 0xFF;                   //  Initialize the I/O ports
        DDRA = 0x00;                    // port A all inputs
        DDRB = 0x00;                    // port B.1 is an output  (light)
        DDRB = 0x02;

      // Initialize the UART control register
      // RX & TX enabled,  8 data bits
        UCR=0xl8;
      // initialize the UART's baud rate
        UBRR=xtal/16/baud-1;

        while(l)                    // do forever..
        {
            while(PINA.0)           // Wait for a button and
              seed++;               // let the counter roll over and
                                    // over to form a seed.
            first = second = third = seed;  // preload the columns

            do                              // mix up the numbers
            {                               // while waiting for button release.
              first ^= seed>>l;  // Exclusive ORing in the moving seed
              second^= seed>>2;  // can really stir up the numbers.
              third ^= seed>>3;
              seed++;                   // keep rolling over the seed pattern
            } while(PINA.0 == 0);  // while the button is pressed

            flash lights(5);  //  flash the lights 5 times..

            first  &= 3;            //  limit number to values from 0 to 3
            second &= 3;
```

```
        third   &= 3;
                                // show the values..   BY NAME!!
                                //   simply change the %d to %s
                                // and pass the pointer to the string from
                                //   the 2D array kind[]   to printf()

        printf("-> %s, %s, %s <-\n",
                        kind[first],   kind[second],   kind[third]);

                                // determine a payout..
        i = get_payout();

      switch(i)                 // now print the payout results
      {

                case        ZERO:
                    printf("Paid out:   ZERO\n");
                    break;

                case        NICKEL:
                    printf("Paid out:   One Nickel\n");
                    break;

                case        DIME:
                    printf("Paid out:   One Dime\n");
                    break;

                case        JACKPOT:
                    printf("Paid out:   JACKPOT!!\n");
                    break;
            }
        }
    }
```

1.10 结构与共用体

结构和共用体用来在一个标题(或名称)下对变量进行分组。因为在 C 语言程序中。“对象”通常是指一组数据成员，或数据成员的共用，结构和共用体是面向对象程序设计中的基本类型。面向对象程序设计涉及到方法，程序使用这些方法来基于关系来处理数据。结构或共用体可以视为对象。结构或共用体的成员被认为是该对象的属性(变量)。在程序中，对象名提供了一种(在整个程序中)标识对象属性共用的方法。

1.10.1 结构

结构是从一个或多个变量创建一个单一数据的方法。结构中的变量称为成员。这种方法允

许通过一个名字引用成员集合。有些高级语言称这种对象为记录或基本变量。结构与数组不同，包含在结构里面的变量可为不同类型。

一个结构声明如下：

```
struct structure_tag_name {
        type member_1;
        type member_2;
        ...
        type member_x;
};
```

或

```
struct structure_tag_name {
        type member_1;
        type member_2;

        type member_x;
}structure_var_name;
```

在程序中，一旦结构的模板定义好了之后，structure_tag_name 作为一个公共的描述符，用来声明这种类型的结构。下面的示例声明 var1 和 var2 两个结构，声明 var3 为结构数组。

```
struct struct_tag_name var1,var2,var3[5];
```

结构模板可包含各种类型的变量，包括其他结构、指向函数的指针及指向结构的指针。应该注意：当模板定义好后，并未分配存储器空间，只有实际声明结构变量后才分配空间。

使用成员运算符(.)访问结构内的成员，它把结构和成员名连接起来：

```
structure_var_name.member_1   structure_var_name.member_x
```

和数组一样，结构的初始化可在结构名后跟一个用方括号括起来的初始化值列表来实现：

```
struct DATE   {
        int month;                  //declare a template for a
        int day;                    //date structure
        int year;
}

//declare a structure variable and initialize it.....

struct DATE date_of_birth={2,21,1961};
```

产生与以下赋值语句相同的效果：

```
date_of_birth.month=2;
date_of_birth.day=21;
date_of_birth.year=1961;
```

因为结构是一种有效的类型，所以对于结构内部成员的嵌套没有限制。例如一组结构声明如下：

```
struct LOCATION {
        int x;
        int y;              //this is the location coordinates x and y
};
struct PART{
```

```
    char part_name[20];              // a string for the part name
    long int sku;                    //a sku number for the part
    struct LOCACTION bin;            //its location in the warehouse
}widget;
```

要访问 widget 变量的存储单元，需提供如下的引用：

```
x_location=widget.bin.x;   // the x coordinate of the bin of the widget
y_location=widget.bin.y;   //the y coordinate of the bin of the widget
```

要设置 widget 的存储单元，使用相同的规则：

```
widget.bin.x=10;   //the x coordinate of the bin of the widget
widget.bin.y=23;   //the y coordinate of the bin of the widget
```

结构可作为参数传递给函数，也可从函数中返回。例如：

```
struct PART new_location(int x, int y)
{
    struct PART temp;

    temp.part_name=" ";          //initialized the name to NULL
    temp.sku=0;                  //and zero the sku number
    temp.bin.x=x;                //set the location to the passed x an y
    temp.bin.y=y;
    return temp;                 //and then return the structure to the caller
}
```

它将返回结构 PART，其中的 bin.x 和 bin.x 的存储单元成员赋值给参数并传递给函数。成员 sku、part_name 在这个结构返回之前清空。于是，上面的函数可按照如下的形式赋值：

```
widget=new_location(10,10);
```

结果是，part_name 和 sku 会被清空，widget.bin.x 和 widget.bin.y 的值设为 10。

1.10.2　结构数组

与其他类型的变量一样，还可以声明结构数组。结构数组的声明如下：

```
struct PART{
    char part_name[20];              // a string for the part name
    long int sku;                    //a sku number for the part
    struct LOCACTION bin;            //its location in the warehouse
} widget[100];
```

访问成员的方法仍然是一样的，唯一的不同之处在于结构变量本身的索引。因此，要访问一个特定 widget 的存储单元，可以使用类似于下面的引用：

```
x_location=widget[12].bin.x;
```

```
                    // the x coordinate of the bin of the widget 12
y_location=widget[12].bin.y;
                    //the y coordinate of the bin of the widget 12
```

本例，存在字符串 part_name，可以当作普通的字符串访问：

```
widget[12].part_name;                    //the name of widget 12
```

或

```
widget[12].part_name[0];
                //the first character in the name of widget 12
```

结构数组的初始化的方法是：在结构数组的名字后用大括号把初始化值括起来，仅仅需要对每个数组元素中的每个结构元素进行初始化。

```
struct DATE      {
        int month;
        int day;
        int year;
}
struct DATE birthdates[3]={      2, 21, 1961,
                                 8, 8, 1974,
                                 7, 11, 1997 };
```

1.10.3　指向结构的指针

有时需要使用一般化的方式来描述结构的成员。其方法之一就是使用引用结构的指针。例如向函数传递指向结构的指针而不是传递整个结构。指向结构的指针可声明如下：

```
struct   structure_tag_name *structure_var_name;
```

运算符(*)表明 struct_var_name 是一个指向 structure_tag_name 类型结构的指针。和其他类型一样，指针需赋值为一个确实存在的对象，像已声明的变量。变量声明保证了存储器的分配。

以下代码用来声明结构变量 widget 和指向结构变量的指针 this_widget，最后一行将 widget 的地址赋值给指针 this_widget。

```
struct LOCATION {
        int x;
        int y;              //this is the location coordinates x and y
};

struct PART{
        char part_name[20];                  // a string for the part name
        long int sku;                        //a sku number for the part
        struct LOCACTION bin;                //its location in the warehouse
};
```

```
struct PART ,*this_widget;
        //declare a structure and a pointer to a structure

...
this_widget=&widget;
        //assign the pointer the address of a structure
```

当一个指针用于引用结构之后，可使用结构指针运算符(->)来间接访问结构的成员：

```
this_widget->sku=1234;
```

同样也可用间接运算符来指向结构的第 1 个位置，并使用成员运算符(.)来访问成员 sku：

```
(*this_widget).sku=1234;
```

因为 this_widget 是 widget 的指针，以上显示的两种赋值方法都是合理的。this_widget 外的括号是必须的，因为成员运算符比间接运算符的优先级更高。如果省略括号，则表达式会被误解为：

```
*(this_widget.partname[0]),
```

而这实际上为 widget 的地址。

结构可包含指向其他结构的指针，还可以包含指向同一类型的结构的指针。结构不能包含自己作为成员，因为会导致递归声明，编译器缺少分析声明的信息。无论指针指向什么，都具有相同的大小。因此，通过指向相同类型的结构，结构可以成为“自引用的”。基本示例如下：

```
struct LIST_ITEM{
    char *string;              //a text string
    int position ;             //its position in a list
    struct LIST_ITEM *next_item;        // a pointer to another
                                        //structure of the same type
}item,item2;

item.next_item=&item2;
        //assign the pointer with the address of the item2
```

现在

```
item.next_item->position
```

为 next_item 指向的 position 结构成员。等价于

```
item2.position
```

典型情况下，自引用结构用于链表和快速排序等数据操作中。

1.10.4　共用体

共用体的声明和访问与结构类似。共用体的声明有如下形式：

```
union union_tag_name {
        type member_1;
        type member_2;
          ...
        type member_x;
};
```

或

```
union union _tag_name {
        type member_1;
        type member_2;
             ...
        type member_x;
}union_var_name;
```

共用体和结构最大的区别在于存储器的分配方式。共用体的成员实际上共享共用体中一个最大成员的存储器：

```
union SOME_TYPES{
      char character;
      int integer;
      long int long_one;
} my_space;
```

在本例中，my_space 分配的最大存储器数量等于长整型变量 long_one(4 字节)的大小。如果长整型变量的赋值为：

```
my_space.long_one=0x12345678L;
```

则 my_space.character 和 my_space_integer 的值也将被修改。本例中，它们的值为：

```
my_space.character=0x12;
my_space.integer=0x1234;
```

共用体有时被用作节省内存空间的方法。如果许多变量临时性使用，且不会同时使用，则共用体是一种定义存储器“暂存笺”区域的方法。共用体还常用来从更大的数据对象中抽取数据的一小部分。在前面的例子中，实际数据的位置取决于所使用的数据类型和特定编译器如何处理大于整型长度(8 位)的数据。编译器不同，存储的数据也不同。数据可为按字节次序交换也可为按字次序交换，或者两者都行。本例用来测试编译器怎样组织存储器中的数据。

共用体的声明可以节省将数据从一种组织转换到另一种组织所需的编码步骤。在下面的两个例子中，两个位输入端口结合在一个 16 位的值中。第 1 种方法使用移位和结合，第 2 种方法使用共用体。

```
#include <delay.h>
#include <90s8515.h>  // register definition header file for a Atmel AT9058535

void main(void)
  {
     unsigned int port_w;

     while(1)
     {
          port_w=PINA;              //get port A into the 16 bit value
          port_w <<=8;              //shift it up...
          port_w |=PINB;            //now combine in port B
```

```
                printf("16-bits=%d04x\n",port_w);
        }                                       //put the combined value out to
    }                                           //the standard output
```

使用共用体声明的方法可以获得同样的结果。

```
#include <delay.h>
#include <90s8515.h>        //register definition header file for an Atmel AT90s8535

void main(void)
{
            //declare the two types of data in
            //a union to occupy the same space...
        union {
            unsigned char port_b[2];
            unsigned int port_w;
        }value;

            while(1)
            {
            value.portb[0]=PINA;        //get port A
            value.portb[0]=PINB;        // get port B
                        //the union eliminates the need
                        // to combine the data manually
                printf("16-bits=%d04x\n",port_w);
        }                                       //put the combined value out to
}                                               //the standard output
```

1.10.5　typedef 运算符

C 语言支持创建新类型名的操作。typedef 运算符允许将某个名称声明为与某个现有类型同义。例如：

```
typedef unsigned char byte;
typedef unsigned int word;
```

现在别名 byte 和 word 可以用来声明实际上为 unsigned char 类型和 unsigned int 类型的变量。

```
byte var1;   //this is the same as an unsigned char
word var2;   //this is the same as an unsigned int
```

使用别名的方法对于结构和共用同样适用：

```
typedef struct
{
    char name[20];
    char age;
    int home_room_number;
}student;
```

```
student   Bob;          //these allocate memory in the form of
student   Sally;        // a structure of type student
```

在编译预处理程序中，#define 语句有时用于通过文本替代执行别名操作。typedef 由编译器直接求值，可以用于声明、类型转换和应用等超出预处理能力的操作。

1.10.6　位和位域

存储器空间溢出时，通常会使用位和位域。一些编译器支持 bit 类型，由编译器自动分配存储器，并且作为其自己的变量引用。例如：

```
bit running ;           //the compiler allocate a single bit of
                        //storage for this flag..

running=1;              //the only two possibilities for value are 1 and 0
running=0;

if(running)
 ;                              //the value can be tested as well as assigned
```

与位不同的是，位域更适应于更大的更一般的系统。但不是所有的嵌入式编译器都支持位域。位域通常与结构相关联，因为它的声明形式如下：

```
struct structure_tag_name{
      unsigned int bit_1:1;
      unsigned int bit_2:1;
           …
      unsigned int bit_15:1;
};
```

或

```
struct structure_tag_name{
      unsigned int bit_1:1;
      unsigned int bit_2:1;
           …
      unsigned int bit_15:1;
}struct_var_name;
```

指定位域的方法是，首先给出成员名(只为 unsigned int 类型)，紧跟一个冒号(:)以及值所需要的位的大小。成员的宽度大小从 1 到 unsigned int 类型的大小(16 位)。这样允许一个结构内的一些位域通过一个单独的无符号整型存储器存储单元表示。

```
struct{
      unsigned int   running:1;
      unsigned int   stoped:1;
      unsigned int   counter:4;
}machine;
```

如同访问结构中的成员一样，也可以通过成员名访问位域：

```
machine.stopped=1;      //these are single bits..so only 1 and 0 are allowed

machine.running=0;

machine.counter++;      //this is 4-bit value,so 0-15 are allowed
```

有时，在嵌入式系统中位域用来描述 I/O 端口引脚：

```
typedef struct
{
        bit_0:1;
        bit_1:1;
        bit_2:1;
        bit_3:1;
        bit_4:1;
        bit_5:1;
        bit_6:1;
        bit_7:1;
}bits;
```

接下来的#define 用来在 0x38 存储单元标注存储器的内容(#define 被预处理程序当作文本替换指令对待)。这允许程序员在整个程序中通过 PORTB 来访问存储器的存储单元 0x38，就像它是个变量一样：

```
//PORTB is the contents of address 0x38 , a bitfield "bits"
#define PORTB(*(bits *)0x38)
```

bits 位域允许独立访问 I/O 端口 PORTB 的单个位，有时称为位范围。

```
//bitfields can be used in assignments
PORTB.bit_3=1;
PORTB.bit_5=0;

//bitfields can be used in conditional expressions
if(PORTB.bit_2|| PORTB.bit_1)
        PORTB.bit_4=1;
```

1.10.7　sizeof 运算符

C 语言支持一种称为 sizeof 的一元运算符。这个运算符是一个编译时特征，产生一个与数据对象或类型相关联的常量值。sizeof 运算符的形式如下：

```
sizeof (type_name)   //type_name could be keyword int, char, long.etc

sizeof(object)       //object could be a variable ,array, structure ,or union variable name
```

这些操作产生一个显示括号中的类型或数据对象大小的整型常量。例如，假定有如下的声明：

```
int value,x;
long int array[2][3];

struct record
{
        char name[24];
```

```
    int    id_number;
}students[100];
```

以下是对上面的变量进行 sizeof 操作的可能结果：

```
x=sizeof(int);          //this would set x=2,since an int is 2 bytes
x=sizeof(value);        //this would set x=2,since value ia an int
x=sizeof(long);         //this would set x=4,since an int is 4bytes

x=sizeof(array);        // x= x=sizeof(long)*array_width*array_length=4*3*2=24
x=sizeof(record);       //x=24+2 for the character string plus the integer
x=sizeof(students);     //x=100 Elements*(24 characters +sizeof(int))
                        //x=100*(24+2)=2600!!
```

1.11　存储器类型

微处理器的结构可能要求变量和常量分别存储在不同类型的存储器中。不会发生改变的数据应当存储在同一种类型的存储器中，而必须在程序中反复读取和写入的数据应当存储在另一种类型的存储器中。还存在第 3 种类型的存储器，用于存储可变数据，这些数据甚至在系统断电时也必须保留下来。当访问特殊的存储器类型(如指针和寄存器变量)时，还必须考虑附加因素。

1.11.1　常量和变量

AVR 微控制器使用 Harvard 结构设计，为数据(SRAM)、程序(FLASH)和 EEPROM 存储器使用分开的地址空间。这种结构将在第 2 章中详细介绍。CodeVisionAVR 和其他编译器实现 3 种存储器描述符，以允许更易访问这些不同类型的存储器。

变量的默认或自动分配(不使用存储器描述符关键字)是在 SRAM 中进行的。常量可以使用 flash 或 const 关键字置于 FLASH 存储器(程序空间)中。对于放置在 EEPROM 中的变量，要使用关键字 eeprom。变量声明以后，flash 和 eepron 关键字的位置就十分重要。如果 const、flash 或 eeprom 首先出现，这告诉编译器实存的存储器分配或数据存储单元是在存储器区域中。如果类型声明紧跟在 flash 或 eeprom 关键字之后，这表示该变量是引用 FLASH 或 EEPROM 的变量，但变量本身在物理上位于 SRAM 中。这种情况在声明 FLASH 或 EEPROM 中的指针时常常用到。

下面的声明将把物理数据直接置于程序存储器(FLASH)中。这些数据的值都为常量，在程序执行时不能改变。

```
flash int    integer_constant=1234+5;
flash char char_constant='a';
flash long long_int_constant1=99L;
flash long long_int_constant2=0x10000000;
flash int    integer_array1[]={1,2,3};

//The first two elements will be 1 and 2 ,the rest will be 0.
flash int    integer_array2[]={1,2};
```

```
flash int   multidim_array[2,3]={{1,2,3},{4,5,6}};
flash char string_constant1[]="This is a string constant";
const char string_constant2[]="This is also a string constant";

flash struct {
        int a;
        char b[3],c[3];
}sf={{0x000a},{0xb1,0xb2,0xb3},{0xc1,0xc2,0xc3}};
```

EEPROM 空间是非易失的存储器变量区域。变量可通过简单的声明而置于 EEPROM 存储器中：

```
eeprom int   cycly_count;          //allocate an integer space in EEPROM
eeprom char ee_string[20];         //allocate a bytes area in EEPROM

eeprom struct{
    char a;
    int b;
    char c[15];
}se;                                    //allocate 18 byte structure "se" in EEPROM
```

这些永久(FLASH)和半永久(EEPROM)存储器区域在嵌入式系统中有许多特定的用途。FLASH 空间是保存不变数据的优秀区域。程序代码本身就放在这个地方。在此区域声明对象，如文本字符串和算术查找表，可以节省 SRAM 空间。

如果一个字符串使用如下所示的初始化方式声明：

```
char mystring[30]=" This string is placed in SRAM";
```

则该字符串将在 SRAM 中分配 30 个字节。字符串 This string is placed in SRAM 物理上位于 FLASH 存储器中。FLASH 中驻留的数据在启动时复制到 SRAM 中，并且无论何时要访问 mystring，程序都会在 SRAM 中运行。如果程序在运行期间并没有对该字符串进行修改，则会浪费 30 字节的 SRAM 空间。为了防止 SRAM 空间的浪费，字符串可通过直接声明存储在 FLASH 存储器中：

```
flash char mystring[30]=" This string is placed in SRAM";
```

EEPROM 区域称为非易失的，这意味着当微控制器断电时，数据仍完好无损。但是程序的半永久性可以改变位于区域的数据。EEPROM 也有一个生命期，即在进行电修改失败之前可以执行写循环的最大次数。这取决于 EEPROM 存储器本身的构造及其电化学特性。在大多数情况下，这个存储器区域最多能进行 10000 次写操作。随着新技术的发展，写操作的数目在不断增长，可达到数十万次，甚至上百万次。对于数据的读出次数则没有限制。

当使用 EEPROM 设计软件时，理解这种物理约束非常重要。要保存并且不经常改变的数据可以存储在该区域中。对于访问速度要求较慢的数据日志、校准表、运行时刻仪表、软件安装和配置值等，该区域非常有用。

1.11.2　指针

程序运行期间，对这些特殊存储器区域的指针进行处理的方法是不同的。尽管指针可指向 EEPROM 和 FLASH 存储器区域，但实际上指针本身也存储在 SRAM 中。在这种情况下，分配多为普通的 char 和 int 等，但引用的存储器类型必须描述清楚，以使得编译器能够生成正确的代码来访问所要求的区域。关键字 flash 和 eeprom 用来详述名字，例如：

```
    /*pointer to a string that is located in SRAM */
char *ptr_to_ram="This string is placed in SRAM";

    /*pointer to a string that is located in FLASH */
char *ptr_to_flash="This string is placed in FLASH";

    /*pointer to a string that is located in EEPROM */
char *ptr_to_eeprom="This string is placed in EEPROM";
```

1.11.3　寄存器变量

AVR 微控制器的 SRAM 区域包括一个称为寄存器文件(Register File)的区域。这个区域包括 I/O 端口、计时器和其他外围设备，以及工作区或“暂存笺”区。如果要告诉编译器将变量分配给一个或几个寄存器，使用存储类 register 修饰符。

```
register int abc;
```

即使未使用 register 修饰符，编译器也可选择自动将变量分配到一个或几个寄存器中。要防止变量分配到寄存器中，需要使用 volatile 修饰符。这警告编译器该变量可能在赋值时被外部修改。

```
volatile int abc;
```

当微控制器处于休眠状态时(休眠状态是一种停止的和低耗电模式，典型情况下在电池应用程序中使用)，变量存储在 SRAM 中的应用程序经常使用 volatile 修饰符。这样，每次在微控制器被唤醒后返回到正常操作时，SRAM 中的变量有效。

还没有分配给寄存器的全局变量存储在 SRAM 的全局区域或全局变量区域中。还没有分配到寄存器中的局部变量存储在 SRAM 区域的数据栈或堆空间中动态分配的空间中。

sfrb 和 sfrw

I/O 端口和外围设备位于寄存器文件中。因此，特殊指令用来告诉编译器 SRAM 位置、I/O 端口或寄存器文件中另一个外围设备之间的差别。

sfrb 和 sfrw 关键字告诉编译器 IN 和 OUT 汇编指令是用来访问 AVR 微控制器的 I/O 寄存器的。

```
/*Define the Special Function Registers(SFR)*/

sfrb   PINA=0x19;                    //8-bit access to the SFR input port A
```

```
sfrb   TCNT1L=0x2c;                    //8-bit access to the lower part of timer;
sfrb   TCNT1H=0x2d;                    //8-bit access to the upper part of timer;
sfrw   TCNT1=0x2c;          //16-bit access to the timer(unsigned int)

void main(void)
{
      unsigned char a;

      a=PINA;                          //Read PORTA input pins
      TCNT1=0x1111;                    //Write to TCNT1L & TCNT1H registers

      while(1)
            ;
}
```

通过将位选择器附加在 I/O 寄存器的名字之后，可以允许对 I/O 寄存器进行位一级的访问。因为对寄存器的位一级访问 I/O 是通过使用 CBI、SBI、SBIC 和 SBIS 汇编语言指令来实现的，所以在使用 sfrb 声明时，寄存器地址的范围须在 0x00～0x1F 之间，在使用 sfrw 声明时，寄存器地址必须在 0x00～0x1E 之间。例如：

```
sfrb   PORTA=0x1b;
sfrb   DDRA=0x18;
sfrb   PINA=0x19;

void main(void)
{
      DDRA.0=1;                        //set bit 0 of Port A as output
      DDRA.1=0;                        //set bit 1 of Port A as input
      PORTA.0=1;                       //set bit 0 of Port A to a 1

      while(1)
   {
         if(PINA.1)                    //test bit 1 input of Port A
            PORTA.0=0;
   }
}
```

通常，sfrb 和 sfrw 关键字位于程序开头部分使用#include 预处理指令的头文件中。典型情况下，这些头文件与特定的处理程序相关联。这些头文件提供预先确定的 I/O 名称，并且提供其他位于(在特定应用程序中使用的)微控制器中的有用寄存器。

以下是一个由编译器提供的典型包含文件(90s8515.h)：

```
//I/O registers definitions for the AT90S8515
sfrb ACSR=8;
sfrb UBRR=9;
sfrb UCR=0xa;
```

```
sfrb USR=0xb;
sfrb UDR=0xc;
sfrb SPCR=0xd;
sfrb SPSR=0xe;
sfrb SPDR=0xf;
sfrb PIND=0x10;
sfrb DDRD=0x11;
sfrb PORTD=0x12;
sfrb PINC=0x13;
sfrb DDRC=0x14;
sfrb PORTC=0x15;
sfrb PINB=0x16;
sfrb DDRB=0x17;
sfrb PORTB=0x18
sfrb PINA=0x19;
sfrb DDRA=0x1a;
sfrb PORTA=0x1b;
sfrb EECR=0x1c;
sfrb EEDR=0x1d;
sfrb EEARL=0x1e;
sfrb EEARH=0x1f;
sfrb EEAR=0x1e;                    //16 bit access
sfrb WDTCR=0x21;
sfrb ICR1L=0x24;
sfrb ICR1H=0x25;
sfrb ICR1=0x24;                    //16 bit access
sfrb OCR1BL=0x28;
sfrb OCR1BH=0x29;
sfrb OCR1B=0x28;                   //16 bit access
sfrb OCR1AL=0x2a;
sfrb OCR1AH=0x2b;
sfrb OCR1A=0x2a;                   //16 bit access
sfrb TCNT1L=0x2c;
sfrb TCNT1H=0x2d;
sfrb TCNT1=0x2c;                   //16 bit access
sfrb TCCR1B=0x2e;
sfrb TCCR1A=0x2f;
sfrb TCNT0=0x32;
sfrb TCNR0=0x33;
```

```
sfrb MCUSR=0x34;
sfrb MCUCR=0x35;
sfrb TIFR=0x38;
sfrb TIMSK=0x39;
sfrb GIFR=0x3a;
sfrb FIMSK=0x3b;
sfrb SPL=0x3d;
sfrb SPH=0x3e;
sfrb  SREG=0x3f;

//Interrupt vectors definitions

#define EXT_INT0 2
#define EXT_INT1 3
#define TIM1_CAPT 4
#define TIM1_COMPA 5
#define TM1_COMPB 6
#define TIM1_OVF 7
#define TIM0_OVF 8
#define SPI_STC 9
#define UART_RXC 10
#define UART_DRE 11
#define UART_TXC 12
#define UART_COMP 13
```

1.12 实时方法

实时程序设计有时候被误认为是复杂而神秘的处理，只运行在安装了 Linux 或 UNIX 等操作系统的大型机器上。实际上并非如此，在很多情况下，嵌入式系统更多地需要执行实时处理。

一个简单的程序可能重复运行。它也许能响应所在的硬件环境，但会用自己的时间去做。术语“实时”用来指示程序的功能，它在分配的时间内通过批量的方式执行它所有的功能。它也表明程序有能力对外部刺激(如硬件输入)立即作出响应。

拥有丰富外围设备集的 AVR 不仅可以响应硬件计时器，还可以对输入变化作出响应。程序对实时变化作出响应的能力称为中断或异常处理。

1.12.1 使用中断

中断为由内部或外部硬件引起的程序操作中的异常、流改变或打断。实际上中断是一个硬件产生的函数调用。结果中断会导致在中断函数执行时暂停执行流，该中断函数称为中断服务例程(interrupt service routine，ISR)。一旦 ISR 完成，程序流重新开始，继续从中断的位置执行。

在 AVR 中，中断使状态寄存器和程序计数器被置于堆栈中，并且基于中断源，程序计数器会被分配一个来自地址表中的值。这些地址被称为向量。

一旦程序被中断的向量重定向，就可以通过机器指令 RETI(RETurn from Interrupt)返回到正常操作状态。RETI 指令将状态寄存器恢复储存为中断之前的值，并将程序计数器中设置成紧跟中断指令之后的下一条指令。

AVR 微控制器中有许多可用的中断源。AVR 越大，可用的中断源就越多。下面所列出的是一些可能性。这些定义通常建立于程序开始部分的头文件中，并且对给定的微控制器而言是特殊的。这些定义放在 ATMega128 的头文件 Mega128.h 中。

```
//Interrupt vectors definitions

#define EXT_INT0 2
#define EXT_INT1 3
#define EXT_INT2 4
#define EXT_INT3 5
#define EXT_INT4 6
#define EXT_INT5 7
#define EXT_INT6 8
#define EXT_INT7 9
#define TIM2_COMP 10
#define TIM2_OVF 11
#define TIM1_CAPT 12
#define TIM1_COMPA 13
#define TIM1_COMPB 14
#define TIM1_OVF 15
#define TIM0_COMP 16
#define TIM0_OVF 17
#define SPI_STC 18
#define USART0_RXC 19
#define USART0_DRE 20
#define USART0_TXC 21
#define ADC_INT 22
#define EE_RDY 23
#define ANA_COMP 24
#define TIM1_COMPC 25
#define TIM3_CAPT 26
#define TIM3_COMPA 27
#define TIM3_COMPB 28
#define TIM3_COMPC 29
#define TIM3_OVF 30
```

```
#define USART1_RXC 31
#define USART1_DRE 32
#define USART1_TXC 33
#define TWI 34
#define SPM_RDY 35
```

这个列表包含与描述中断源的名字相关联的一系列向量索引。要创建由中断系统调用的函数 ISR，可以通过将保留字 interrupt 作为函数类型修饰符来进行声明。

```
interrupt [EXT_IINT0] void external_int0(void)
{
    /*Called automatically on external interrupt 0*/
}
```

或

```
interrupt [EXT_OVF] void timer0_overflow(void)
{
    /*Called automatically on TIMER0 overflow*/
}
```

关键字 interrupt 紧跟在索引后，索引是中断源的向量位置。向量数字从[1]开始，但因为第 1 个向量是重置向量，所以程序员可以使用的实际中断向量是从[2]开始的。

一旦中断源得到初始化，并且支持全局中断，ISR 就可以在任何时候执行。ISR 不能返回任何值，因为在技术上不存在“调用者”，它总是被声明为 void 类型。基于同样的原因，不能向 ISR 传递任何参数。

在嵌入式环境中，中断能够真正地实现实时执行。在有很多外围设备的系统中，经常可以看到在 main()函数中有空 while(1)循环。在这些情况下，main()函数只是对硬件进行初始化，而中断则要在中断发生时执行全部必需的任务。

```
//I/O register definitions for the AT90S8515
#include <90s8515.h>

//quartz crystal frequency [Hz]
#define xtal 400000

//LED blink frequency[Hz]
#define Blink_Rate 2

//TIMER1 overflow interrupt   service   routine
//occurs every 0.5 seconds

interrupt [TIM1_OVF] void timer1_overflow(void)
{
    //preset again TIMER1
```

```
    TCNT1=0x10000 - (XTAL/1024/Blink_Rate);

    PORTB.0^=1;              //toggle the LSB of  PORTB
}

void main(void)
{
    //set the I/O ports and initialize TIMER1

    DDRB=0x01;               //PORTB.0 pin is an output

    TCCR1A=0;                //TIMER1 is disconnected from pin 0C1, no PWM
    TCCR1B=5;                //TIMER1 clock is xtal/1024
    TCNT1=0x10000 - (XTAL/1024/Blink_Rate);     //preset TIMER1
    TIFR=0;                  //clear any TIMER1 interrupt flags

    TIMSK=0x80;              //enable TIMER1 overflow interrupt

    GIMSK=0;                 //all other interrupt sources are disabled

    #asm("sei");             //global enable interrupts

    while(1)
    ;
    //the rest is done by the"timer1_overflow"function
}
```

本例中，main()函数用来初始化系统，支持 TIMER1 溢出中断。中断函数 timer1_overflow 每半秒使 LED 闪烁一下。值得注意的是：当 LED 一直在闪烁时，main()一直停在无限循环 while(1)中。

1.12.2　状态机

在设计程序中，为防止该程序闲置等待输入，通常使用状态机。状态机经常在 switch/case 语句中编码，标志用来指示进程何时从当前状态转移到下一个状态。状态机也可以提供一个更好的机会来改变函数和程序流，而无需重写代码，这是因为状态可在不影响周围状态的情况下进行增加、修改和移动操作。

状态机允许程序的主要逻辑操作在后台进行。因为处理每个实际状态需要很少的 CPU 时间，所以 CPU 的大量有用时间留给时间急迫的任务，如收集模拟信息，处理串行通信和执行复杂的算数运算等。额外的 CPU 时间一般用于与用户的交流：用户界面、显示、键盘服务、数据加密、报警和参数修改等。

图 1-1 所示为控制虚拟交通灯原理图。

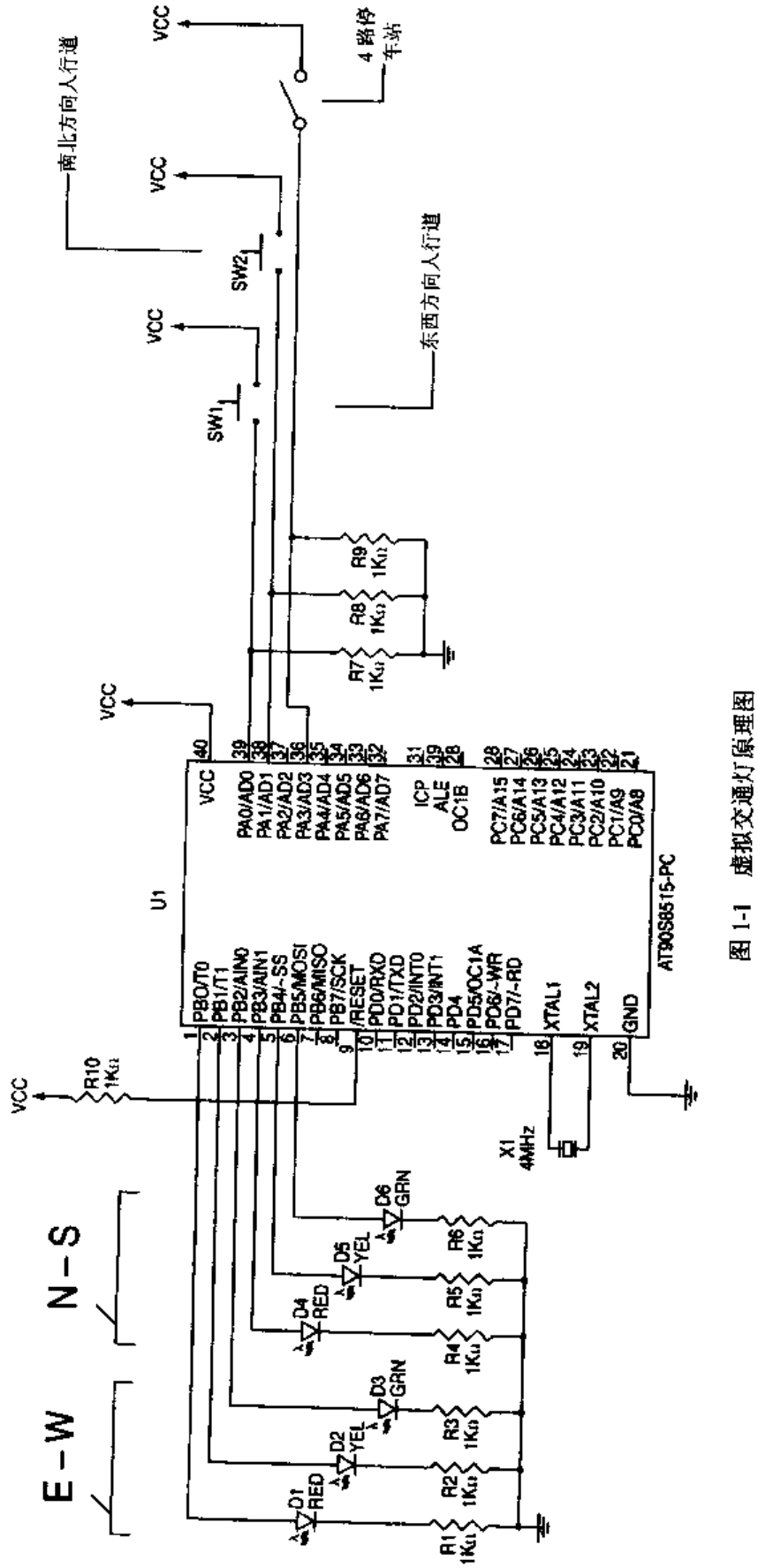


图 1-1　虚拟交通灯原理图

下面的示例用于控制图 1-1 所示的虚拟交通灯。

```
#include <90s8515.h>
#include <delay.h>

#define     EW_RED_LITE PORTB.0        /* definitions   to actual outputs   */
#define     EW_YEL_LITE PORTB.1        /* used to control the lights   */
#define     EW_GRN_LITE PORTB.2
#define     NS_RED_LITE PORTB.3
#define     NS_YEL_LITE PORTB.4
#define     NS_GRN_LITE PORTB.5

#define     PED_XING_EW PINA.0    /* pedestrian crossing push button */
#define     PED_XING_NS  PINA.1    /* pedestrian crossing push button */
#define     FOUR_WAY_STOP PINA.3        /* switch input   for 4-Way Stop */

char time_left;              // time in seconds spent in each state
int current_state;           // current state of the lights
char flash_toggle;           // toggle used for FLASHER state

// This enumeration creates a simple way to add states to the machine
// by name.   Enumerations generate an integer value for each name
// automatically,   making the code easier to maintain.

enum { EW_MOVING,   EW_WARNING,   NS_MOVING,   NS_WARNING,   FLASHER };

// The actual state machine is here..
void Do_States(void)
{
      switch(current_state)
      {
          case    EW_MOVING:       // east-west has the green!!
                  EW_GRN_LITE = 1;
                  NS_GRN_LITE = 0;
                  NS_RED_LITE = 1;    // north-south has the red!!
                  EW_RED_LITE = 0;
                  EW_YEL_LITE = 0;
                  NS_YEL_LITE = 0;

                  if(PED_XING_EW || FOUR_WAY_STOP)
                  {          // pedestrian wishes to cross,or a 4-way stop is required
                             if(time_left > 10)
                               time_left = 10;   //   shorten the time
                  }
                  if(time_left   != 0)              // count down the time
```

```
        {
                --time_left;
                return;                    // return to main
        }                                  // time expired,  so..
        time_left = 5;          // give 5 seconds to WARNING
        current_state = EW_WARNING;
                                   // time expired,  move
        break;                     // to the next state

case    EW WARNING:
        EW_GRN_LITE = 0;
        NS_GRN_LITE = 0;
        NS_RED_LITE = 1;    // north-south has the red..
        EW_RED_LITE = 0;
        EW_YEL_LITE = 1;    // and east-west has the yellow
        NS_YEL_LITE = 0;

        if(time_left   != 0)       // count down the time
        {
              --time_left;
              return;              // return to main
        }                          //   time expired,  so..
        if(FOUR_WAY_STOP)   //   if 4-way requested then start
              current_state = FLASHER;   // the flasher
        else
                                   // otherwise..
               time_left = 30;     // give 30 seconds to MOVING
               current_state = NS_MOVING;
        }                          //   time expired,  move
        break;                     //   to the next state

case    NS_MOVING:
        EW_GRN_LITE = 0;
        NS_GRN_LITE = 1;
        NS_RED_LITE = 0;    // north-south has the green!!
        EW_RED_LITE = 1;    // east-west has the red!!
        EW_YEL_LITE = 0;
        NS_YEL_LITE = 0;

        if(PED_XING_NS || FOUR_WAY_STOP)
        {          //   if a pedestrian wishes to cross,or a 4-way stop is required..
              if(time left > 10)
              time_left = 10;      // shorten the time
        }
        if(time left   != 0)       // count down the time
```

```
        }
            --time_left;
            return;                 // return to main
        }                           // time expired,  so..
        time_left = 5;              // give 5 seconds to WARNING
        current_state = NS_WARNING;     //  time expired,  move
        break;                          // to the next state
case    NS_WARNING:
        EW_GRN_LITE = 0;
        NS_GRN_LITE = 0;
        NS RED_LITE = 0;        // north-south has the yellow..
        EW_RED_LITE = 1;
        EW_YEL_LITE = 0;        // and east-west has the red..
        NS_YEL_LITE = 1;

        if(time_left   != 0)        // count down the time
        {
            --time_left;
            return;                     // return to main
        }                               // time expired,  so..
        if(FOUR_WAY_STOP)               // if 4-way requested then start
            current_state = FLASHER;    // the flasher
        else
        {                               // otherwise..
            time_left = 30;             // give 30 seconds to MOVING
            current state = EW_MOVING;
        }                               // time expired,  move
        break;                          // to the next state

case    FLASHER:
        EW_GRN_LITE = 0;    // all yellow and
        NS_GRN_LITE = 0;    // green lites off
        EW_YEL_LITE = 0;
        NS_YEL_LITE = 0;

        flash_toggle ^= 1;          // toggle LSB..
        if(flash_toggle & 1)
        {
            NS_RED_LITE = 1;    // blink red lights
            EW_RED_LITE = 0;
        }
        else
        {
            NS_RED_LITE = 0;    // alternately
            EW_RED_LITE = 1;
```

```
                }
                if(!FOUR_WAY_STOP)          //  if no longer a 4-way stop
                    current_state = EW_WARNING;
                break;                                  // then return to normal

            default:
                current_state = NS_WARNING;
                break;          // set any unknown state to a good one!!
            }
    }

    void main(void)
    {
        DDRB = 0xFF;          // portb all out
        DDRA = 0x00;          // porta all in

        current_state = NS_WARNING;     // initialize to a good starting
                                        // state (as safe as possible)
        while(1)
        {
            delay ms(1000);     // 1 second delay.,  this time could
                                // be used for other needed processes
            Do_States();        // call the state machine,  it knows
                                // where it is and what  to do next
        }
    }
```

本例中，状态机使用 PORTB 来驱动南北方向和东西方向的红绿黄灯。注意：main()中的 delay_ms()函数用来控制时间，这样可使例子简单。在实际生活中，这个时间可用来控制无数种任务和功能，灯的计时操作可由来自硬件计时器的中断进行控制。

状态机 Do_States()每秒调用一次，但它在返回到 main()函数之前只执行很少的指令。全局变量 current_state 通过 Do_States()来控制程序流。PED_XING_EW、PED_XING_NS 和 FOUR_WAY_STOP 输入通过改变时间选择或机器的路径(或者两者)来产生正常机器流的异常。

这个示例系统起到正常的停车灯的功能。当东西方向为红灯时，南北方向为绿灯。在每个方向马路通行时间 32 秒，从绿灯变为红灯有 5 秒的黄灯警告。如果一行人想穿过马路，按下 PED_XING_EW 或 PED_XING_NS 按钮将导致在改变方向前通行时间少于 10 秒。如果打开 FOUR_WAY_STOP 开关，那么所有的灯将变为红灯。这种改变只会发生在黄灯改变时，以保证交通安全。

用户界面、显示、键盘服务、数据入口、报警和参数修改的执行同样可以使用状态机。使用标志和状态来将程序“切得越细”，实时性就越强。更多的事件可以得到持续处理，而不会陷入等待状态发生变化的情况。

1.13　本章小结

本章是开始进行 C 语言编程的基础。

基本的概念演示了 C 程序的基本结构。变量、常量、枚举类型及其作用域和构造，以及数组、结构和共用类型等对于定义存储器的分配十分有用。

表达式和运算符(包括 I/O 操作)提供了执行算术操作和判断逻辑条件的基础。这些表达式和运算符还同控制结构，如 while、do/while、for 循环、switch/case 和 if/else 语句一起使用，来形成函数，指示程序的执行流。

本章还介绍了使用中断和状态机的实时编程高级概念，这些概念用于演示如何把程序执行流制成流线型，改善其可读性，以及如何在 C 语言项目中对进程提供计时控制。

1.14　练习

1. 定义变量和常量(1.4 节)。

*2. 根据以下要求创建正确的声明(1.4 节)：

A. 常量 x，值为 789

B. 变量 fred，取值范围 3～456

C. 变量 sensor_out，能表示 - 10～+45

D. 有 10 个元素的变量数组，每个元素的取值范围为 - 23～345

E. 一个值为 Press here to end 的字符串常量

F. 指针 arry-ptr，指向由 3～567 的数字组成的数组

G. 使用枚举类型将 uno、dos、tres 的值分别设为 21、22、23

3. 对下面给出的条件求值(1.6 节)

A. unsigned char t;　t=0x23;　//t=?

B. unsigned int t;　t=0x78/34; //t=?

C. unsigned char x; x=678;　//x=?

D. char d; d=456;　//d=?

E. enum{start=11,off ,on ,gone}; //gone=?

F. x=0xc2; y=0x2; z=x^y;　//z=?

G. e=0xffed; e=e>>4;　e=?

H. e=27;e+=3;　//e=?

I. f=0x90|7;//f=?

J. x=12;y=x+2; //y=?　x=?

*4. 像在条件语句中一样，判断下面的值是真还是假，其中 x＝0x45；y＝0xc6(1.6 节)：

A. (x==0x45)

B. (x|y)

C. (x>y)

D. (y - 0x06==0xc)

5. 对下面程序代码段运行后的变量求值(1.7 节)：

```
unsigned char loop_count;
unsigned int value=0;
for(loop_count=123;loop_count<133;loop_count++)
value+=10;
//value=??
```

6. 对程序代码段运行后的变量求值(1.7 节)：

```
unsigned char cntr=10;
unsigned int value=10;
do
{
      value++;
}while (cntr<10);
//value=??  Cntr=??
```

7. 对程序代码段运行后的变量求值(1.7 节)

```
unsigned char cntr=10;
unsigned int value=10;
while(cntr<10)
{
   value++;
}
//value=?? Cntr=??
```

8. 假定有：unsigned char num_array[]={1,2,3,4,5,6,7,8,9,10,11,12}; (1.9 节)
求 num_array[3]=?

9. 假定有：unsigned int*ptr_array; (1.9 节)

```
unsigned int num_array[]={10,20,30,40,50,60,70};
unsigned int x,y;
ptr_array=num_array;
x=*ptr_array++;
y=*ptr_array++;
```

求 x=?? y=?? ptr_array=??

10. 编写一段 C 语言代码，声明一个合适数组，并将 2 的 21－26 次幂放到该数组中

前面带有*的习题表示完整的答案或部分解答在附录 H 中可以找到。

1.15　上机实习

1. 创建、编译和测试一个程序，在标准输出设备上打印字符串 C Rules and Assembler Drools!!。

修改程序，使用 for 循环显示这个相同的短语 4 次。

使用 while 循环进行重复，打印这个短语 3 次。

使用 do/while 循环来打印这个短语 5 次。

2. 设计一程序，计算粪池的容积。高、宽、长从端口 A、B、C(单位为英尺)中读出，将结果打印到标准输出设备上。程序可处理的最大尺寸为 50 英尺宽，50 英尺长，25 英尺深(那是最大的池了)。在要求正确计算容量的地方使用正确的变量大小和类型转换，并以如下方式打印结果(x、y、z 和 qq 应为实际数值)：

池的容积 x 英尺长，y 英尺高，z 英尺宽，qq 立方英尺

程序运行一次，打印结果后停止。

3. 修改第 2 题的程序，如果 3 个输入中的任何一个超出允许范围，则程序打印字符串 The input exceeds the program range。

4. 设计一程序显示 2 的 20～212 次幂。

5. 根据表 1-11 设计一程序，使用 switch/case 语句连续从端口 A 读数据并输出到端口 B：

表 1-11　switch/case 练习

输　入	输　出
0x00	0b00000000
0x01	0b00001111
0x02	0b11110000
0x03	0b01010101
0x04	0b10101010
0x05	0b11000011
0x06	0b00111100
0x07	0b10000001
Other	0b11111111

6. 根据 1.10 节中有关共用的内容，创建一个程序来确定长变量的存储顺序。

7. 创建一个程序，该程序能够在结构数组中搜索特定的值。声明一个包含成员 month、day 和 year 的结构数组。将数组初始化为一个只有您自己生日的实例。您生日的年月日需在数组中其他地方显示出来，而不是全都在同一位置。程序必须对数组进行搜索来发现您的生日，然后打印一条消息，显示生日信息驻留在数组中的哪个索引。

8. 修改第 7 题的程序，使用指针来搜索结构数组。

9. 创建一个程序，使用标准输入函数 getchar()来获取一个字符，然后将这个字符的值同置

于端口 A 的值进行比较。如果匹配，使用 printf()函数打印消息到标准输出设备。如果端口 A 的值大于接收到的字符的值，则将 0x01 输出到端口 B，否则将 0x80 输出到端口 B。

10. 每当中断发生时，使用外部硬件中断将一条消息打印到标准输出设备。同时打印发生的中断次数。您可以不必使用全局变量来计算中断次数。

第2章 Atmel Risc处理器

2.1 本章目标

通过本章的学习，您应该掌握以下内容：

- 定义与描述 RISC 微控制器的用法，并标识它的存储空间和寄存器
- 将微控制器的资源分配给某一项目
- 在 C 语言中使用中断
- 编写 C 程序以完成简单的 I/O 任务
- 应用 AVR 微控制器的主要外设来解决电子设备的接口问题
- 能编写 C 程序来完成串行通信
- 阅读 AVR 汇编语言程序，并确定其结果

2.2 引言

对于需要成功地应用 Atmel AVR RISC 微控制器来解决问题的项目师而言，本章提供了大多数硬件背景知识。而成功地应用的前提是：设计者必须非常了解和精通硬件知识，就同他们非常精通编程语言一样。

本章首先描述了处理器的总体结构：体系结构和功能。紧接着详细介绍了它的存储器空间，以便标识和了解存储器中变量的存放位置，以及为什么这样存储等。接着，我们结合其内置的常用外设来讨论中断问题。最后，综合以上几方面的知识，将其运用到项目的处理器资源分配过程中。

本章讨论 Atmel 微控制器的各种外设，包括了新的 Mega 系列部件。这些设备对于很多 Atmel 微控制器来说都是常用的。许多 Atmel 的部件都有另外的外设或者能利用外设来扩展功能。当您要了解某个特定微控制器的某个部件的全部外设功能时，请参考相应的说明书。在本书中，当在一个项目中应用某个部件的时候，同时也给出了它的用法。最后本章对汇编语言也做了介绍，因为这和理解微控制器的功能是密切相关的。

2.3 体系结构概述

本书所提到的微控制器是 Atmel AVR RISC 微控制器。本节介绍其体系结构。

首先，它们是 RISC 设备。RISC 代表 Reduced Instruction Set Computing(精简计算指令集，

或简称精简指令集)。也就是说，这些设备可以利用精简了的机器级(Machine-level)指令，而使运行速度更快。这些精简了的指令之所以能够提高运行速度，是因为每个指令的机器指令数目变少了，很多指令可以在一个处理器时钟周期内完成。如果使用 MIPS(Millions of Instructions per Second，每秒所执行的百万条指令数)来衡量计算速度，一个有 8MHz 时钟频率的 AVR 处理器可以在一秒钟内执行 8 百万条左右的指令，也就是将近 8 个 MIPS 的速度。

其次，这些设备是微控制器。微控制器是为了优化硬件控制而将整个处理器、存储器和所有 I/O 外设封装到了单个硅片中的完整的计算机系统。由于是集成在一块硅片中，使速度得到了进一步提高，因为内嵌 I/O 设备的读、写所需要的时间将比外部设备要少。

优化硬件控制意味着在机器级指令集中提供了方便简捷的指令，使控制 I/O 设备变得更简单、容易。例如，提供设置、清零、查询并行 I/O 端口或寄存器中某一位的独立指令。设置和清零某一位功能的典型应用就是用于控制某个硬件设备的开关。在一个没有对硬件控制进行优化的处理器中，设置、清零或者读入某一个 I/O 端口(或者其他任何寄存器)的指定位时，要用额外的 AND、OR、XOR 或者其他指令去改变该端口的位。

以下的代码段是把 Port A 中第三位的值设置为高电平：

```
PORTA = PORTA | 0x4;  // set bit 2 high
```

虽然这行代码在标准微控制器和微控制器上都是正确的，但是二者的结果却是截然不同的。

以下的代码段显示了在执行上面的语句时，标准微处理器和微控制器所产生的不同的汇编代码：

标准微控制器：

```
IN    R30, 0x1B
MOV   R26, R30
LDI   R30, LOW(4)
OR    R30, R26
OUT   Rx1B, R30
```

微控制器：

```
SBI   0X1B, 2
```

上面的例子中，标准微控制器要用 5 个指令来把某一位设置为高电平，而微控制器则只要一个指令。这意味着对于端口控制微控制器的位操作速度相当于标准微控制器的 5 倍，而代码量只有其 1/5。微控制器通过提供直接控制硬件的指令来提高执行速度和减少代码。

总的说来，Atmel AVR 设备的结构就是一个微控制器的结构。它分为三部分：CPU(Central Processing Unit)、存储器和 I/O。CPU 部分是我们看不见的部分，而存储器和 I/O 部分是极其清楚的，需要应用设计人员了解。我们将会在本章后面的内容中分别对这三个部分进行讨论。

2.4　存储器

Atmel RISC AVR 处理器的存储器单元是以 Harvard 模型为基础的，在该模型中，存储器的

不同部分是分离的，以提高访问的速度和系统性能。CPU 拥有独立的 FLASH 代码存储器(FLASH Code Memory)和数据存储器(Data Memory)的接口，如果带有 EEPROM，还包括 EEPROM 存储器的接口。

实际的存储器和中型微控制器的一样。有关某个特定微控制器的存储器的实际情况，请参考相应微控制器的说明书。

2.4.1　FLASH 代码存储器

FLASH 代码存储器单元是一个首地址为 0x0000 的 FLASH(闪存)存储器，而它的容量由具体的 Atmel AVR 微控制器决定。FLASH 代码存储器是非易失性(non-volatile)存储器，即当电源被切断以后，保存在里面的数据也不会消失。所以它被用来存储可执行代码和常量，因为这些数据在电源被切断后还必须保存在存储器中。代码存储器中的每个存储单元是 16 位的，每个存储单元可以容纳一个 16 位长的机器级指令。图 2-1 给出了 FLASH 代码存储器单元的存储器映射。

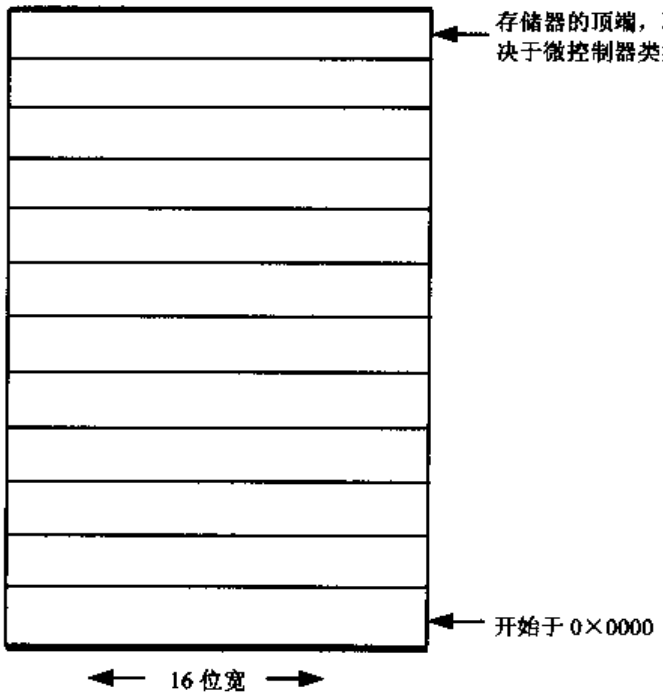


图 2-1　FLASH 代码存储器映射

虽然 FLASH 代码存储器里面的执行代码可以重复编写，但是不能用可执行程序来对 FLASH 编程，必须利用其他的方法编程。因此，从程序员的角度来看，它只是一个只读的存储器而已。事实上，它也只是用来存储可执行代码和常量。常量在保存到 FLASH 的时候，由于 FLASH 代码存储器单元长度的缘故，会自动地转换成整型(int)大小的变量。

2.4.2　数据存储器

Atmel AVR 处理器的数据存储器一般包括 3 个相互独立的读/写存储区。最低的区域是 32 个通用工作寄存器，接着是 64 个 I/O 寄存器，之后就是内部 SRAM。

通用工作存储器一般用于存储程序运行过程中的局部变量和其他暂时性的数据，甚至还可

以用于存储全局变量。64 个 I/O 寄存器则用作微控制器上 I/O 设备和外设的接口。而内部 SRAM 则作为通用的变量存储空间，同时也是处理器的堆栈。

数据存储器的映射图如图 2-2 示。

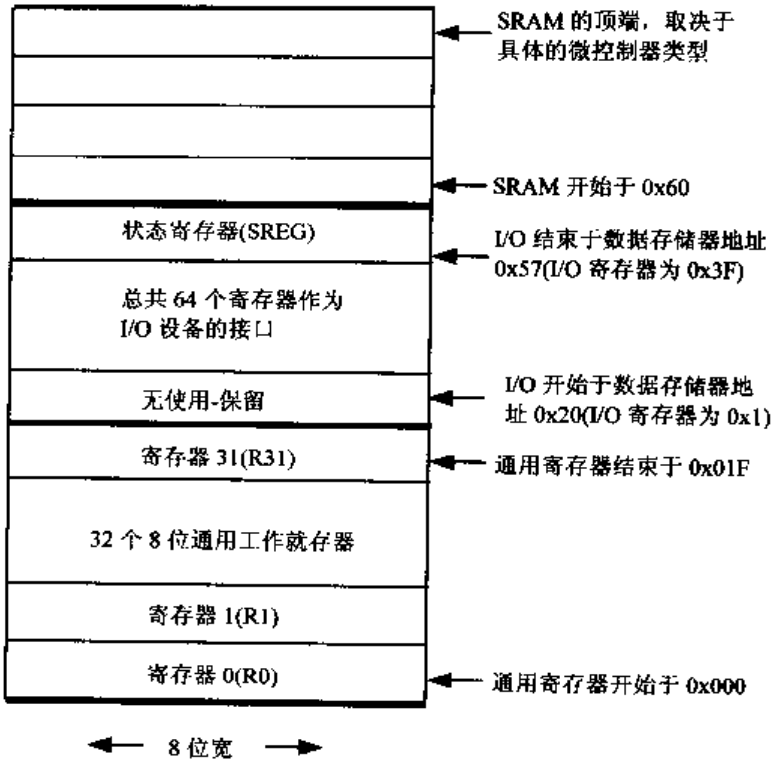


图 2-2　数据存储器映射图

1. 寄存器

通用工作寄存器占据了数据存储器最低的 32 个单元。这些寄存器就和计数器里面的存储单元一样，只是在里面保存暂时或者中间的计算结果。它们有时用于存储局部变量，有时用于存储全局变量，有时用于存储指向处理器要用的存储器单元的指针。简而言之，处理器在运行程序时才用到这 32 个工作寄存器。这些寄存器的使用一般由 C 编译器来控制，而不是由程序员来控制的，除非他使用汇编语言。

2. I/O 寄存器

I/O 寄存器则占据了通用寄存器后面的最高 64 个字节的数据存储器空间(参考图 2-2)。其中的每一个寄存器都提供对微控制器内部的 I/O 外设的控制寄存器或数据寄存器的访问。程序员可以大量使用 I/O 寄存器来作为微控制器的 I/O 外设的接口。

在第 1 章中，我们已经接触过利用 I/O 寄存器的例子 PORTA 和 DDRA。在那些例子中的 I/O 寄存器是关于在一个并行端口写入和读取数据的。表 2-1 所示为和并行端口相关联最多的几个 I/O 寄存器列表。

表 2-1　常用 I/O 寄存器

I/O 寄存器名	I/O 寄存器地址	SRAM 地址	说　明
DDRA	0x1A	0x3A	Port A 的数据定向寄存器
DDRB	0x17	0x37	Port B 的数据定向寄存器
PORTA	0x1B	0x3B	Port A 的输出寄存器
PINB	0x16	0x36	Port B 输入引脚
SREG	0x1F	0x5F	处理器状态寄存器

每个 I/O 寄存器都有一个名字、一个 I/O 地址和一个 SARM 地址。C 语言程序员通常会用到 I/O 寄存器名字。在编写汇编程序的时候，两个不同的数值地址是非常重要的，因为有些指令是与 SRAM 地址相关的，而有些指令却是与 I/O 地址相关的。如果要了解有关 AVR 指令集或者汇编语言的详细内容，参见 2.11 节。

相对于 C 语言程序员而言，I/O 寄存器的名字用起来比地址都方便。但是，C 语言编译器并不是天生就知道这些名字的含义以及它和地址的关联关系的。每个程序都要包含一个#include 头文件，该头文件为编译器定义这些名字以及与之关联的地址。头文件的内容在第 1 章中已经详细地进行说明了。下面的 C 语言程序演示了上面的概念，使用了寄存器名来初始化端口。

```
# include <90s8535.h>
main()
{
    DDRA = 0xFF;        // all bits of Port A are output
    DDRB = 0x00         // all bits are inut
    while (1)
    {
        PORTA = PINB;       //read port B and write to A
    }
}
```

上面的程序利用 PINB I/O 寄存器来读入引脚 PORT B 的数据，用 PORTA 寄存器来把结果输出到 Port A 的锁存器中。C 语言编译器根据程序第 1 行的#include 头文件中的定义使用寄存器名字来获得对应的寄存器地址，本例中是 90s8535.h 文件。

总的说来，C 语言编译器利用 I/O 寄存器作为微控制器内部 I/O 硬件设备的接口。下面将会说明微控制器中每个 I/O 外设以及和它们相关的 I/O 寄存器的用法和功能。

3. SRAM

SRAM 存储器区域是用来存储那些不适合存放在寄存器中的变量，并用于存储处理器堆栈。SRAM 存储器区域如图 2-3 所示。

如图 2-3 所示，在 SRAM 中是没有特别的存储器区间或内存分块。数据一般从 SRAM 的底部开始存储，而处理器的堆栈则是从存储器顶部开始存储的。也就是说数据存储是由底而上利用存储器。

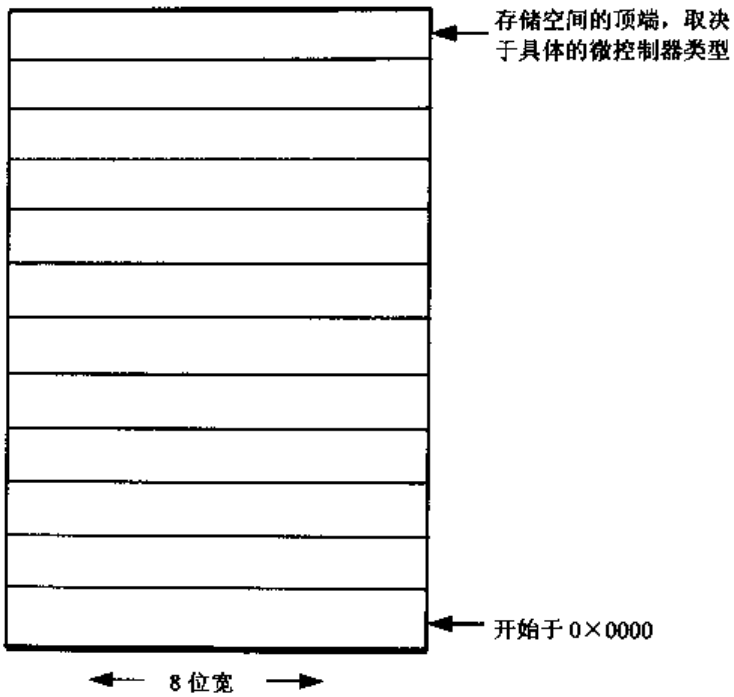


图 2-3　SRAM 存储器映射图

处理器用堆栈的存储器空间来暂时保存函数的返回地址、计算的中间结果和其他短期的、暂时性的数据。实际上，CodeVision AVR C 语言编译器实现两个堆栈：系统堆栈(System Stack)和数据堆栈(data stack)。系统堆栈从 SRAM 区域的顶部开始，用于存储存函数的返回地址和处理器的控制信息；数据堆栈就在系统堆栈的下面，以相似于系统堆栈的工作方式自顶而下工作，用于存储暂时数据，如函数中的局部变量。

SRAM 数据存储的用法是相当简单的，但是当它被用于堆栈的时候就不同了。堆栈可以看作是为暂时存储而预留出来的存储器空间，就像在桌子上预留出叠放纸张的空间一样。它以 LIFO(Last in First out，后进先出)的工作方式工作，新的数据(如同一张新的纸)被压进堆栈；而里面原有的数据(相当于桌子上最上面的那张纸)可以从堆栈中弹出来。堆栈指针(Stack Pointer，SP)指向当前可用的存储器单元地址。

堆栈指针总是指向下一个可以用于存放堆栈数据的存储单元。它的工作方式工作如下：

(1) 当有新的数据要放到堆栈中时，处理器就利用 SP 来找出存放数据的存储器地址。例如，如果当前的 SP 里面的值是 0x200，则数据会存放在 SRAM 中的 0x200 内存地址中。

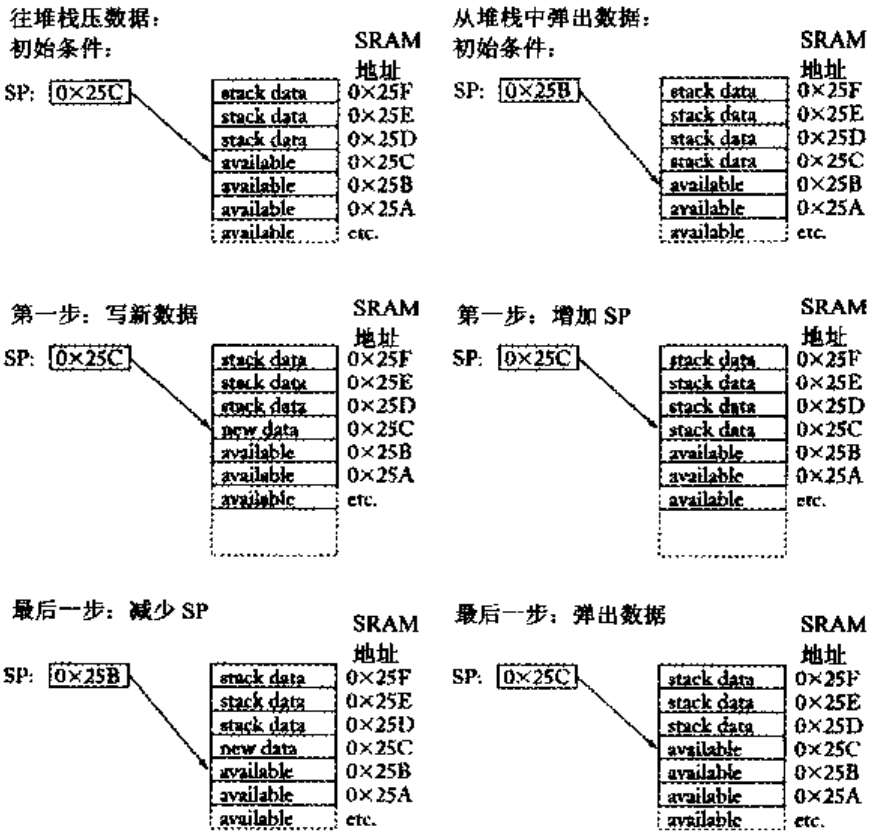
(2) 这时候 SP 会减少到 0x1FF，同时处理器开始执行下一个指令。在这种方式下，SP 总是存放着下一个可用于存放堆栈数据的存储器地址。

在以后的操作中，处理器可能要恢复最后一个压进堆栈中的数据。这个恢复的工作过程如下：

(1) 当处理器需要恢复数据或者从堆栈中弹出数据时，它读取 SP 的值并立即对 SP 加 1。因为 SP 总是指向下一个可用于存放堆栈数据的存储器地址(空着的)，对堆栈指针加 1 之后，就意味着当前由 SP 指向的就是上一次用于存放数据的存储单元。

(2) 处理器用存放在 SP 中的地址从堆栈中弹出数据或恢复数据。由于该存储单元中的数据已经从这个位置恢复，其地址就会留在 SP 中，以供堆栈中的下一个可用的存储单元使用。

堆栈的操作小结如图 2-4 所示。



注意：所有标有 available 的单元实际上都存放有数据，而不管是加电的时候就存在的数据，还是以前的堆栈操作留下的数据。但是这个数据已经不再需要，所以这些单元可以允许写入新的数据。

图 2-4　堆栈操作

和堆栈相关的主要概念就是，随着数据压进堆栈，堆栈在 SRAM 中占用的存储器就会慢慢变大，从 SRAM 的顶部开始向下延伸；随着数据从堆栈中弹出来，原先被堆栈利用的存储器就会被释放出来，同时当前可用于堆栈的存储单元向上移动。

在被堆栈占用的同时，SRAM 也被用于存储变量，不过是从 SRAM 存储器的底部开始，向上延伸。每个微控制器的 SRAM 大小是有限的，所以在应用的时候要确保堆栈不会延伸得太向下，或者数据不会移动到太向上，以防止它们相互重叠和干扰，这是非常重要的。数据重写了堆栈，或者堆栈重写了数据，都会使程序产生不可预测的结果。

2.4.3　EEPROM 存储器

存储器中的 EEPROM 区域是一块可以读/写非易失性存储空间的区域。它常被用来存放那些在掉电以后不能丢失的和微控制器要反复运用的数据。EEPROM 的首地址是 0x000，最大的地址值则取决于所用的具体微控制器。

虽然 EEPROM 存储器是可读写的，但很少用它来存放一般的变量。这是因为 EEPROM 的写速度非常慢，它要用一个毫秒才能完成一个字节数据的写操作。大量地用这类存储器来存放变量会很大程度上降低处理器的速度。同时，在第 1 章已经提到过了，EEPROM 只能经受有限次数的写周期。所以，EEPROM 通常只是为那些在掉电的情况下必须要保存的数据预留的。

第 2 章示例项目：A 部分

这个示例项目用来进一步解释在第 2 章中出现的概念，这是它的第一部分。这个示例项目

将会侧重于赛车的数据采集系统，其场景如下：

假设您的老板将一个基于 Atmel Mega163 微控制器的数据采集系统交给您。同时他解释道，这个系统是设计用来采集一个 15 秒的短程加速汽车赛的数据的。但是他要求您把这个系统改成采集他的改装赛车的数据，要求每秒一次，采集整个 2 分钟长赛时的数据，然后把数据上传到 PC 机中进行分析。在该系统原来的设计中，是根据火花塞的脉冲来记录引擎的 rpm(Revolutions Per Minute，每分钟转数)，根据从一个磁感应器中读出粘附在传动轴上的磁体的数据来记录传动轴的 rpm，根据一个粘附在汽缸上面的 RTD(电阻式温度探测器)热电偶来记录引擎的温度。老板还说，以前他曾经请别人改装过这个系统，但是很不幸的是，那个笨拙的程序员只有在擦除整个程序的那一步是成功的。但这对您来说却是个好消息，因为这意味着所有关于引擎信号检测的电路都是现成的，您所需要做的就是编写一个程序去采集数据并把它们送到 PC 机中去。

您的首要任务就是检验一下处理器是否拥有足够的资源去完成这个工作。首先，我们先检验一下该系统的内存是否足够大，能不能够容纳所有的数据和处理器变量。所要存储的数据就是在两分钟内，每隔一秒采集一组的数据的总和，所以它所需要的数据组的数量计算如下：

2 分×60 秒/分×1 组/秒＝120 (组)

所以您还必须知道每个数据组占用的字节数。从老板那里获知，引擎的 rpm 范围为 2 000～10 000，轴承的 rpm 范围为 1 000～2 000，而引擎的温度范围是 100°F～250°F。老板要求 rpm 要精确到 1 rpm，而温度要精确到 1°F。

这样，在一组数据中，要存储两个 rpm 数据，需要两个整型大小的数值(每个整型数据为两个字节，总共 4 个字节)；而为了存储温度数据，您需要一个字节大小的数值。所以总共需要的存储空间为：

120 组×5 字节/组＝600 字节

除了数据存储以外，您还需要大约 25 个字节的系统堆栈和大约 40 字节的数据堆栈(随着编程能力的提升，您就会学会如何判断所需要堆栈的大小。一般说来，应用越多的数学计算或函数，就需要越多的堆栈)。查阅 Mega163 的说明书，会发现该微控制器有 1 024 字节的 SRAM 存储器，所以它的存储器资源是足够大的。通常在这个时候，还会检查一下外设资源是否也是足够的。本例中您不了解外设，不过不要紧，既然这个系统曾经用在短程加速赛上，那它的外设资源应该还是在的。

2.5　复位和中断功能

中断在第 1 章中已经讨论过了，它是必不可少的硬件触发功能。本节的目的就是说明中断和复位(是一种特殊的中断)是如何工作的。

中断，顾名思义，就是中断处理器当前处理的程序流，并引发它去处理一个中断服务程序(Inerrupts Service Routine，ISP)，该服务程序是相应的中断发生时要执行的任务。中断在需要处

理器做出立即响应和处理器要浪费很多时间去轮询某个事件发生时都是很有用的。需要作出快速响应的例子包括利用中断去跟踪时间(中断可以为时钟提供信号的基准)，或者在紧急情况出现时用中断控制紧急停止按钮，以停止相应的机器运行。

其他中断应用程序的例子包括键盘或其他的输入设备。和微控制器处理指令的能力相比，人类的速度是非常非常慢的。如果要处理器去轮询我们是否单击一个键，将会是很大的浪费。在这种情况下，按键事件可以产生一个中断，微控制器就会暂停并去查看所单击的按钮是否要求它去做别的事。如果该按键动作只是产生某个动作中的第 1 个行为，处理器会回到原来的工作中，一直到单击足够产生某个动作的键。中断的利用，将处理器从不停地以某个大大高于我们按键频率的频率来查询键盘中解放出来。

中断也包括复位，它们基本上都是以同样的方式工作的。每个中断都有一个指向程序存储器地址的向量。编译器把每个中断服务程序的首地址和相应的转跳指令放在相应的中断向量中。当中断发生时，程序在完成了当前正在执行的指令后，就转向与该中断相关联的中断向量。然后，程序执行相应的转跳指令，跳转到中断服务程序代码处并开始执行。

中断发生时，返回地址(执行完中断后执行的下一条指令的地址)会储存到系统堆栈里面。中断服务程序的最后一条语句是 RETI 汇编语言指令。这条指令导致返回地址(return address)从系统堆栈中弹出，这样程序从中断的地方开始继续执行。

在 AVR 系列处理器中，所有中断的优先级都是一样的。所以是不允许任何中断去打断正在执行的中断程序，即一条中断都不可能优先于其他中断。但是有时候可能有两个中断同时发生，这时候就需要一个判断模式，以决定先执行哪个中断。有时候，这个判断模式称为优先级。当两个中断同时发生时，拥有较小编号向量的中断会优先执行。详细资料请参考 2.5.1 小节的中断向量表或具体的微控制器的说明书。

2.5.1　中断

表 2-2 给出的是在 AT90S8535 中提供的中断源中前 6 个。中断源及其向量的选择要根据具体的处理器而定。

中断必须先初始化才能工作，或者说才能可用。初始化中断是一个两步的过程：第一步是把将要工作的中断的屏蔽去掉；第二步是把所有没有被屏蔽的中断打开。

去掉中断的屏蔽就是把中断要用到的控制寄存器的控制位设置为 1。通用中断屏蔽(General Interrupt Mask，GIMSK)寄存器如图 2-5 所示。

表 2-2　简单的中断向量

首　地　址	向 量 编 号	资　　源	说　　明
0x0000	1	复位中断	外部复位，上电，Watchdog 超时
0x0001	2	外部中断 0	INT0 引脚的硬件信号
0x0002	3	外部中断 1	INT1 引脚的硬件信号
0x0003	4	2 号计时器比较	2 号计时器匹配
0x0004	5	2 号计时器溢出	2 号计时器溢出

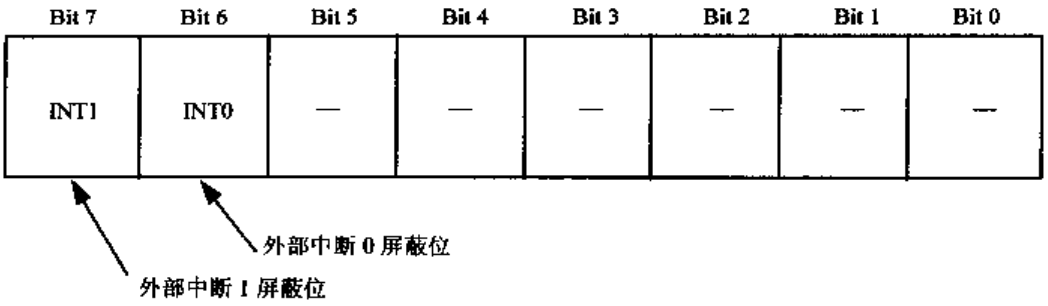


图 2-5　GIMSK 通用中断屏蔽寄存器

GIMSK 是用来支持外部中断的。设置 INT0 位会支持外部中断 0(一个外部硬件信号，运用于名称也为 INT0 的引脚)，同样设置 INT1 位则支持外部中断 1。如果两个控制位都设置了，则支持两个中断。屏蔽位就像是挡板一样使用。

当有适当的硬件信号出现在 INT1 时，例如该信号是和 GIMSK 中的 INT1 位进行逻辑与的结果。如果结果是 1，这个中断就会被触发。但是如果 INT1 位设置为 0，与该位的逻辑与的值永远都是 0，即相应的中断永远不会被触发。所以为了能够使中断能正常工作，必须设置中断屏蔽位。

支持中断的第二步就是设置位于处理器状态寄存器(Status Register，SREG)中的全局中断支持位。这一步是如下完成的：

```
#asm("sei")
```

这一行代码应该插在 C 语言程序中要打开全局中断支持位的地方。这一行代码利用了编译指令 #asm，指示编译器在这里插入汇编语言指令 SEI，该汇编语言指令的作用是设置全局中断支持位为 1。

中断发生时，中断请求信号会先和相应的中断控制寄存器中的中断屏蔽位进行逻辑与运算，得到的结果再和全局中断支持位进行逻辑与运算。如果最后的结果是逻辑 1 的话，则该中断请求得到批准，此时处理器会在中断向量表中找到适当的代码地址。下面的示例程序演示了外部中断 0 的用法。

```
#include <90s8535.h>
interrupt [EXT_INT0] void ext_int0_isr(void)
{
  PORTA = PORTA ^ 0x1;   //toggle the LSB of port A
}

void main(void)
{
     DDRA=0x01;      //port A LSB set for output

     GIMSK = 0x40;   //set the INT0 bit to enable external interrupt 0
     MCUCR= 0x02;    //set the ISC01 to activate on a falling edge
```

```
    #asm("sei")     //enable the global interrupt enable bit

    while (1)
        ;               //do nothing – the interrupt does it all
}
```

示例程序利用外部中断 0 来触发连接在 Port A 上 0 位的 LED 灯。每当连接在 INT0 引脚的按钮被单击时，Part A 上 0 位的 LED 就会亮起来。图 2-6 给出了该程序相应的硬件连接图。

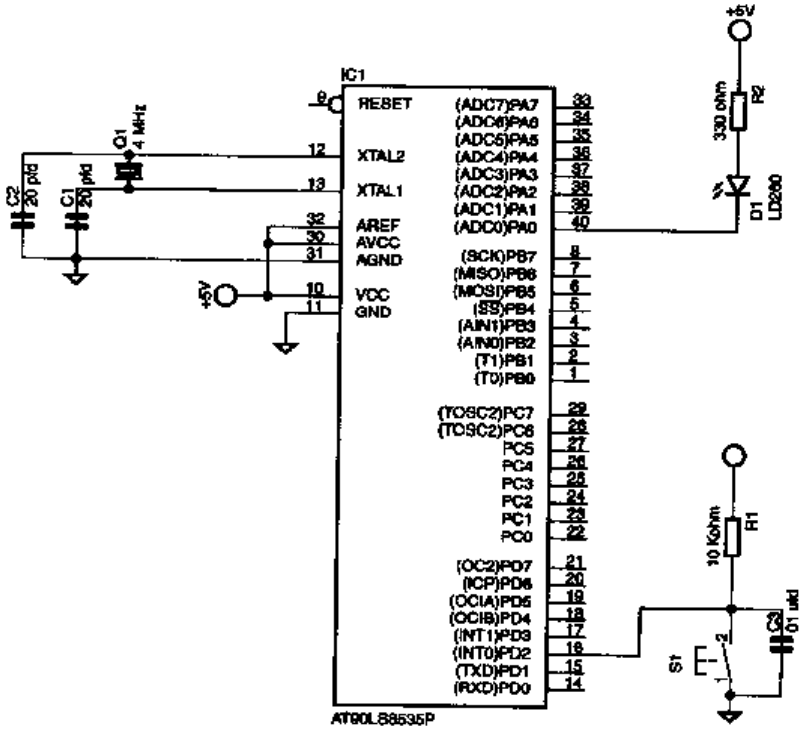


图 2-6　中断程序硬件连接图

审视这个程序时，发现该程序的第 1 行是 # include 语句，用于定义 AT90S8535 微控制器的寄存器和中断向量。跟着的 4 行程序就是外部中断 0 的中断服务程序(ISR)。ISR 程序的第 1 行是以保留字 interrupt 开始，这是告知编译器紧跟着的函数是一个中断服务程序。编译器通过这个信息，就知道寄存器的内容必须由这个函数来保存和恢复(在该函数开始的时候把数据压入堆栈，结束时再把相应的数据弹出来)，同时这个函数必须以一个 RETI 指令结束，而不是以 RET 指令结束。

[EXT_INT0]是由#include 文件定义的，并且设置数值为 2，因为外部中断 0 的向量编号是 2。编译器利用这个信息正确地放置中断向量表中的 ISR 函数的跳转地址。这个函数声明的其余部分和其他普通函数声明一样。

该 ISR 程序的第 3 行是一条有效的执行语句。每当它被执行时，就用 OR 语句去触发 Port A 的最低位。

在 main()函数的第 1 行，Port A 的最低位被设置作为输出。接着的一行代码是设置 GIMSK 中的屏蔽位，使在全局中断支持位为 1 时外部中断 0 能够正常工作。再下一行代码是设置 MCU 控制寄存器(Control Register，MCUCR)中的中断感应位(Interrupt Sense Bit)，使外部中断 0 能被

引脚 INT0 上的下降沿脉冲触发。中断感应位的定义以及和两个外部中断的关系如图 2-7 和表 2-3 所示。

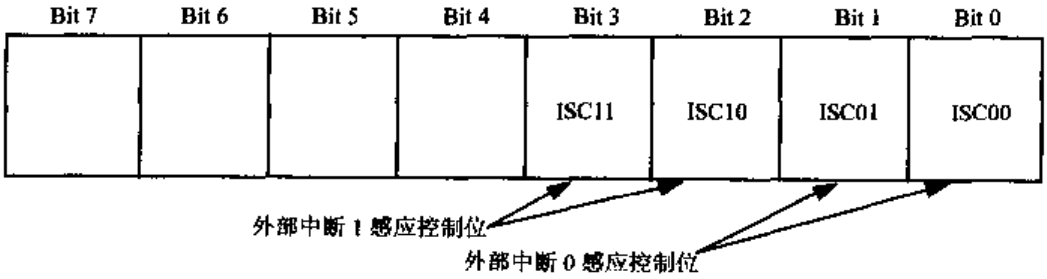


图 2-7　MCUCR 中断感应控制位

表 2-3　MCVCR 中断感应控制位

ISCx1	ISCx0	中断工作方式
0	0	中断由低电平触发
0	1	保留，没有使用
1	0	中断由下降沿触发
1	1	中断由上升沿触发

代码#asm ("sei")是一个汇编语言指令(全称是 Set Enable Interrupt，即设置中断为支持)该指令设置全局中断支持位，以便引脚 INT0 上的下降沿信号可以产生中断并触发 LED。而在 mian() 中的代码"while (1)"则没有任何特别的作用，只是使处理器处于忙碌状态，直到中断发生。

2.5.2　复位

复位(Reset)是编号最低的中断。同时它也是一个特殊的中断，因为它优先于任何中断和正在运行的代码。有 3 个原因可以引起复位：外部复位引脚上超过 50ns 的逻辑低信号；微控制器的上电复位操作；watchdog 计时器的超时信号。复位用来将微控制器预先置为一个已知状态，使它能重新从代码存储器的 0x000 存储单元开始执行程序。

复位以后，微控制器状态的改变基本上都是一样的，不同的处理器之间的差别是很小的。主要的改变如下：

- 所有的外设(包括 Watchdog 计时器)都设置为禁止
- 所有的并行端口都设置为输入模式
- 所有中断都被设置为禁用

因为禁用了所有的外设和中断，微控制器就可以开始执行程序，而不会受到中断或外设的影响，引起不可预测的跳转操作而导致异常或其他不正常的行为。把并行端口设置为输入模式是为了确保接口和外设不会被反向驱动，以防止接口引脚被损坏。

1. Watchdog 计时器和复位

Watchdog 计时器(WDT)是一个安全装置。它被设计用来复位处理器。当处理器处于迷失、混乱或执行其他一切非预设操作时，watchdog 计时器就会引起处理器复位。

watchdog 计时器是这样一个计时器：如果处理器允许它计数到超时状态，它就会引发系统复位。程序正常执行时，程序会计时重置 Watchdog 的计时器，以防止它超时。只要程序正常运行，并且重置 Watchdog 的计时器，那么处理器的复位是永远也不会发生的。如果程序一旦迷失了，或被困在某个地方(例如程序死循环)，就会产生超时信号，处理器就会被复位。复位操作的原理就是它可以把程序带回到正常操作。

图 2-8 和表 2-4 给出了 watchdog 计时器控制寄存器(Watch Dog Timer Control Register，WDTCR)的位信息。

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
			WDTOE	WDE	WDP2	WDP1	WDP0

图 2-8　Watchdog 计时器控制寄存器，WDTCR

表 2-4　WDTCR 的有关说明

位	名　字	说　明
WDTOE	支持关闭 Watchdog(Watchdog Timer Off Enable)	允许禁用 WDT
WDE	支持打开 Watchdog(Watchdog Timer Enable)	打开 Watchdog 计时器
WPx	Watchdog 预定标器(Watchdog Prescaler Bit x)	设置 WDT 的超时间隔

watchdog 的时钟触发信号的振荡器是独立于系统时钟的。它的频率取决于加在处理器上的工作电压。当 Vcc 引脚接 5V 电压时，它的频率近似为 1MHz，而接 3V 电压时，它的频率大约为 350kHz。这样，就算 Watchdog 预定标器中设置了超时，但是它实际的超时时间会随着处理器 Vcc 引脚的电压不同而不同。表 2-5 给出了在两种不同电压下，Watchdog 预定标器中各种预设值对应的实际超时时间。

表 2-5　Watchdog 超时时间选择

WDP2	WDP1	WDP0	5V Vcc 电压的超时时间	3V Vcc 电压的超时时间
0	0	0	15ms	47ms
0	0	1	30ms	94ms
0	1	0	60ms	190ms
0	1	1	120ms	380ms
1	0	0	240ms	750ms
1	0	1	490ms	1 500ms
1	1	0	970ms	3 000ms
1	1	1	1 900ms	6 000ms

表 2-5 中给出的时间都是近似值，因为振荡器的频率本来就是近似的，它也是会随 Vcc 的电压而变化。而下面的程序框架则演示了 Watchdog 计时器的用法。

```
void main(void)
{
    WDTCR = 0xB;          //enable WDT and set to 120 ms timeout
    while (1)
    {
        #asm ("wdr")     //reset the watchdog timer
        if (expression)          //if true disable the WDT
        {
          WDTCR = 0x18;       //set both WDE and WDTOE
          WDTCR = 0x00;        //clear WDE
            .
            .
              /* other code here */
        }
    }
}
```

程序框架中的第 3 行代码是支持 Watchdog 的，并把超时时间设置为 120 毫秒。这是通过往 WCTCR 写数据的的一条语句来设置 WDE 位和 watchdog 预定标器位的。当每次执行 while(1)循环中的#asm("wdr")指令时，watchdog 的计时器就会被重置。这是一条 C 语言中没有的汇编语言指令(WDR，Watch Dog Reset)，并且这个语句必须在 watchdog 计时器超时之前执行。

代码中的 if (expression)是用来判断什么时候禁用 Watchdog 计时器。当 expression 为真时，WDT 就会被禁用。如上面的程序所示，禁用 WDT 是一个两步的过程。首先用一步设置 watchdog 关闭支持位(Watchdog Turn Off Enable Bit，WDTOE)和 watchdog 支持位(Watchdog Enable Bit，WDE)；之后马上清除 WDE 位。禁用 watchdog 计时器是一个故意的复合操作，主要是为了防止 watchdog 计时器被意外地停止或者被失常的程序操作禁用。

一般来说，watchdog 计时器一旦打开了，就永远不会被禁用。因为打开它的原因就是为了保护处理器正常处理程序，而不受错误和失常的程序影响。

2.6　并行 I/O 端口

并行 I/O 端口是最通用的 I/O 设备。每个并行 I/O 端口都有 3 个相关的 I/O 寄存器：数据定向寄存器(data direction register)DDRx(这里 x 是 A、B、C 等，取决于具体的处理器和所使用的并行端口)；端口驱动寄存器(port driver register)，典型情况下称为 PORTx；以及端口引脚寄存器(port pins register)PINx。

数据定向寄存器的作用是确定端口中哪些位用来输入，哪些位用来输出。根据程序员的不同需求，输入输出位是可以混用的。当把数据定向寄存器的所有位都复位为逻辑 0 时，该端口的所有位都是输入模式；如果把数据定向寄存器的任一位设置为逻辑 1，则该端口相应的位就处于输出模式。例如，把 DDRA 最低的两个位设为 1，而其他设为 0 的时候，就是把端口 A 的最低的两个位设置为输出模式，而其他的位则作为输入模式。

向端口 A 的输出位输出数据的方式如下：

```
PORTA  = 0x02;      //sets the second bit of port A
                    //and clears the other seven bits
```

从端口 A 的输入位读取数据的方式如下：

```
x  =  PINA; //reads all 8 pins of port A
```

在上面第 2 个例子中，x 会包含所有端口 A 的位值，不管是输出还是输入。因为 PIN 寄存器反映的是端口中所有位的值。

输入引脚都是悬空的，即不必为每个相关的端口引脚加入上拉电阻。如果真的有需要，处理器可以通过设置相应的端口驱动寄存器的相应位为 1，达到上拉电阻的功能。如下所示：

```
DDRA  =  0xC0;      // upper 2 bits as output, lower 6 as input
PORTA  = 0x3;       // enable internal pull-ups on lowest 2 bits
```

尽管不同的处理器可能有些不同，但端口引脚通常都可以接受 20mA 的电流，但它们能提供的驱动电流就相对少多了。这意味着，如果端口引脚如图 2-6 所示接受电流，引脚可以直接驱动 LED。

下面的示例程序用端口 C 的 4 个低位作为输入，而 4 个高位作为输出。该端口输入引脚中的 2 个低位的上拉电阻有效。

另外一些有关 I/O 端口的信息和示例程序可以参见 1.5 节。

```
#include <90S8535.h>        //provide the I/O definitions

mai()
{
    DDRC = 0xf0;        //upper 4 bits as output
    PORTC = 0X3;        //upper 4 bits are cleared and lower
                        //two bits have pull-ups enabled

    while(1)
    {
        PORTC = PINC << 4;          //read the lower 4 bits, shift them 4 bits left and write
                                    //the result to the upper 4 bits of port C
     }
}
```

第 2 章示例项目：B 部分

这是为赛车手建立一个赛车数据采集系统示例项目的第二部分。第一部分中已经检测了系统是否拥有足够的存储器容量去处理任务。这一部分将会建立整个程序代码的框架和用户界面。

确定每个测量、输入和输出所使用的外设资源是很重要的。如果正在初步设计数据采集系统，您需要决定要使用什么资源。分配这些资源时，您需要确认处理器上有足够的资源。本例中需要确定每一次测量使用的是哪个外设，这样您才能相应地编写程序。

通过调查和一些线路检测，可以获得的系统连接如下：

- Start 按钮和 Mega163 的 INT1 引脚连接。该按钮单击时，INT1 上是低电平
- Upload 按钮和端口 A 的 0 位相连接。该按钮单击时，会把 Port A 的 0 位电平拉低
- 引擎的 rpm 脉冲连接到 ICP(Input Capture Pin)引脚
- 传动轴的 rpm 脉冲连接到 INT0 引脚
- 引擎的温度信号连接到 ADC3(Ananlog-to-Digital Converter Input #3，3 号模数转换输入)引脚

现在，您所需要考虑的就是 Start 和 Upload 按钮，其他的信号会在示例项目的第三部分中处理。

首先，您应该设计整个程序的总体结构。回想一下 1.12 节，将那里介绍的方法应用到这个项目中会收到不错的效果。总的说来，这个程序的设计功能是在规定的时间间隔内采集数据，最后把数据上传到 PC 中进行分析。采用实时编程方法可以使我们能利用中断实时地进行数据测量，并且只是在一秒钟间隔到来的那一刻才去记录相应的数据。换句话说，这个测量方法能最大程度地实时跟踪数据，而不管它们是否被记录了。到了记录数据的时间时，记录函数只是简单地提取最新的数据并记录下来。

所以需要中断服务程序来处理两个 rpm 的测量和一个温度的测量。另外，还必须处理 Start 和 Upload 按钮。处理这个任务的一个简单方案是让程序记录 120 个单元数据后就停下来；单击 Start 按钮则只是简单地清除数据单元计数器，并在需要时记录下 120 个单元的数据。这种方案的优点是，当有人错误地开始记录数据的时候，只要单击 Start 按钮就可以重新开始记录。如果 120 个单元的数据记录完成，只要单击 Upload 按钮就会把数据发送出去。

以下的程序框架看起来是满足要求的，并且会随着示例项目的进行而不断充实和完善：

```
#include <mega163.h>
#include <stdio.h>

unsigned char data_set_cntr;   // number of data sets recorded
unsigned int e_rpm[120], s_rpm[120];   //arrays to hold 120 sets of data
unsigned char temp[120];
unsigned int current_e_rpm, current_s_rpm;     //variables to hold current values
unsigned char current_temp;

// External Interrupt 0 service routine
interrupt [EXT_INT0] void ext_int0_isr(void)
{
//place code here to handle drive shaft RPM measurement
}

// External Interrupt 1 service routine
```

```
interrupt [EXT_INT1] void ext_int1_isr(void)
{
data_set_cntr = 0;       //clear counter to start counting
}

// Timer 0 overflow interrupt service routine
interrupt [TIM0_OVF] void timer0_ovf_isr(void)
{
//place code here to produce 1 second intervals
if (data_set_cntr < 120)   //place code here to record a set of data here
                           //if less than 120 sets have been recorded
}

// Timer 1 input capture interrupt service routine
interrupt [TIM1_CAPT] void timer1_capt_isr(void)
{
// place code here to handle engine RPM measurement
}

#define ADC_VREF_TYPE 0x00
//ADC interrupt service routine
interrupt [ADC_INT] void adc_isr(void)
{
//place code here to measure engine temperature
}

void main(void)
{
// Input/Output Ports initialization
PORTA=0x01;          //enable pull up on bit 0 for upload switch
DDRA=0x00;           //all input
PORTB=0x00;
DDRB=0x00;           //all input
PORTC=0x00;
DDRC=0x00;           //all input
PORTD=0x4C;          //enable pull ups on Port D, Bits 2,3, & 6
DDRD=0x00;           //all input

// Timer/Counter 0 initialization
// initialization to be added later

// Timer/Counter 1 initialization
```

```
// initialization to be added later

// External Interrupt(s) initialization
GIMSK=0xC0;    // both external interrupts enabled
MCUCR=0x0A;    // set both for falling edge triggered
GIFR=0xC0;     // clear the interrupt flags in case they are errantly set

// Timer(s)/Counter(s) Interrupt(s) initialization
TIMSK=0x21;    // timer interrupts enabled for later use

// UART initialization
// initialization to be added later

// Analog Comparator disabled as it is not used
ACSR=0x80;
SFIOR=0x00;

// ADC initialization
// initialization to be added later

// Global enable interrupts
#asm("sei")

while (1)
    {
    if (!PINA.0) send_data();   //send data to PC if switch pressed.
    };
}
```

以上代码为整个系统建立了操作模式。INT1 被连接到 Start 按钮，单击按钮时，数据单元计数器会被复位为 0。Timer 0 被用来产生一个间隔为 1 秒的中断(具体细节将会在下一部分的示例项目中讨论)。在 Timer 0 中断服务程序中，如果数据单元计数器的值小于 120，该服务程序会以 1 秒的间隔记录当前数据。所以该模式为：单击 Start 按钮，把数据单元计数器置 0，开始记录数据，直到记录了 120 组数据为止。同时，通过不同的中断以实时方式取得所有数据保持为最新数据。最后一步就是把数据上传到 PC 中去，是由 main()函数中的 while(1)循环来完成的。一旦单击 Upload 按钮，数据就会被连续地发送出去。

下面，示例项目中的程序会加进更多的模块。

2.7　计时器/计数器

计时器/计数器(Timer/counter)可能是微控制器中最常用的复杂外设。它们的用处非常广泛，能够用来测量时间、检测脉冲宽度、测量速度、测量频率，以及提供输出信号等。应用程序实例包括：测量赛车引擎的 rpm；精确计算某一时段的时间(例如计算子弹速度的时间)；产生音调以制作音乐或驱动汽车的点火系统；或者提供一个脉冲宽度或变频脉冲驱动器去控制电机的速度。在本节中，先在总体上讨论计时器/计数器，然后再介绍 AVR 微控制器中的典型计时器/计数器。

虽然计时器/计数器被用于两个完全不同的工作模式——计时和计数，其实它们只是简单的二进制计数器。用于计时状态时，二进制计数器计算加在输入端上的时间段；而在计数状态时，它们计算事件、脉冲或者其他类似的东西。例如，如果一个二进制计数器用 1 毫秒脉冲作为输入，在事件发生时开始计数，并在事件结束的时候停止计数，那么计数器内的最终数值就是整个事件过程的毫秒数。

当一个计时器/计数器用作计数器时，将要计数的事件连接到二进制计数器的输入，这样该事件发生的次数将会被记录。例如，计数器可以用来计算从生产线上出来的豌豆罐头的数目，在这里，每个豌豆罐头都会向计数器的输入发出一个脉冲。计数器上的数据可以随时读取，以检测目前生产线上一共生产了多少豌豆罐头。

AVR 微控制器带有 8 位和 16 位的计时器/计数器。在两种情况下，程序的关键问题都是要知道什么时候计数器达到最大值并进行翻转。当 8 位计数器达到了 255 时，再来一个脉冲，它就会翻转到 0；而 16 位计数器相应的计数值是 65 535。计时器/计数器的翻转事件是极其重要的，它直接影响程序能否从计时器/计数器取得正确的计数结果。事实上，计时器/计数器的翻转是如此重要，以致每次溢出翻转时都会产生一个中断信号。

虽然不同的 AVR 微控制器可能会有不同的配置，但通常都会有两个 8 位计时器(Timer 0 和 Timer 2)和一个 16 位计时器(Timer 1)。下面介绍计时器的预定标器(Prescaler)和输入选择器(Seledtor)，这些都是所有计时器共有的部件，然后再依次讲解每个常用计时器。虽然有些计时器/计数器有很多功能，但是这里只是讨论它们最常用的功能。对于具体的处理器，请参阅相关的说明书，以获取计时器/计数器所提供的所有功能信息。

2.7.1　计时器/计数器预定标器和输入选择器

计时器/计数器的输入可以是从系统时钟获得的不同频率的信号，也可以从外部的引脚获得。计时器/计数器选择哪个引脚上的信号作为输入脉冲，是由相关联的计时器/计数器控制寄存器(Timer/counter Control Register，TCCRx)上的计数器选择位(Counter Select Bit，CSx2，CSx1，CSx0)决定的。图 2-9 给出了大部分 AVR 微控制器的计时器/计数器控制寄存器上的预定标器和输入选择器的设置。

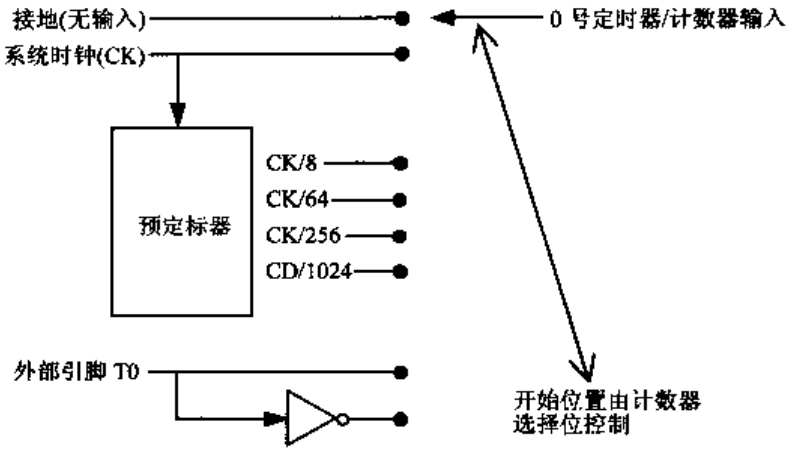


图 2-9　Timer 0 预定标器/多路复用器

以下的代码演示了如何把 Timer 0 初始化为使用系统的 8 分频信号作为输入信号(TCCR0 的 3 个最低位是计数输入选择位)：

```
TCCR0 = 0x02 ; // Timer 0 uses clock/8
```

2.7.2　Timer 0

Timer 0 是一个典型的 8 位计时器，但是这可能也会随具体处理器类型不同而不同。虽然它拥有普通计时器/计数器的所有功能，但是一般只为程序提供时基(Time Base)或作为程序的“钟摆”(Tick)。0 号计时器/计数器控制寄存器(TCCR0)通过选择作用于 Timer 0 的时钟输入来控制 Timer 0 工作。图 2-10 和表 2-6 给出了 TCCR0 的各位的定义。

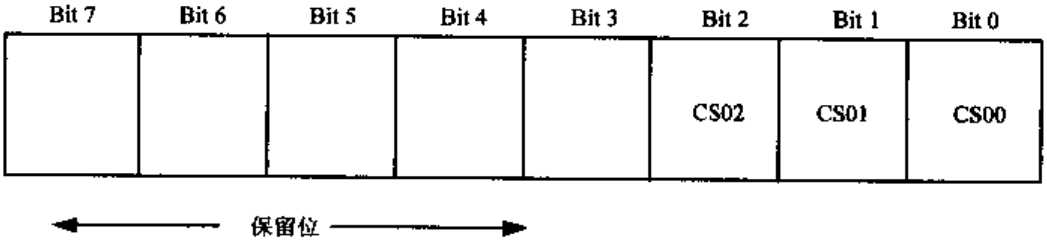


图 2-10　TCCR0

表 2-6　TCCR0 各位的定义

CS2	CS01	CS00	中断工作方式
0	0	0	停止，0 号计时器停止
0	0	1	系统时钟，CK
0	1	0	1/8 系统时钟，CK/8
0	1	1	1/64 系统时钟，CK/64
1	0	0	1/256 系统时钟，CK/256
1	0	1	1/1 024 系统时钟，CK/1024
1	1	0	外部引脚 T0，对下降沿信号计数
1	1	1	外部引脚 T0，对上升沿信号计数

程序的“钟摆”，如同钟表的钟摆一样，提供非常精确的计时事件。整个模式的总体过程是选中一个数值，然后把它加载到计时器中；然后计时器以此为基础向上计数，直到到达 255 溢出翻转，并产生一个中断信号；中断服务程序先把相同的数值重新加载到计时器中，执行任何请求的时间评审行为然后返回到程序中。计时器重复计数到 255，溢出翻转，产生中断，复位的循环。中断是计时产生的。加载到计时器中的数值决定了间隔的长短，数值越小，到达 255 所需要的时间就越长，翻转“钟摆”的间隔就越长。

例如，假设某个程序每 0.5 秒就要触发一次 LED。LED 如图 2-11 所示连接到 Port A 的最高位。

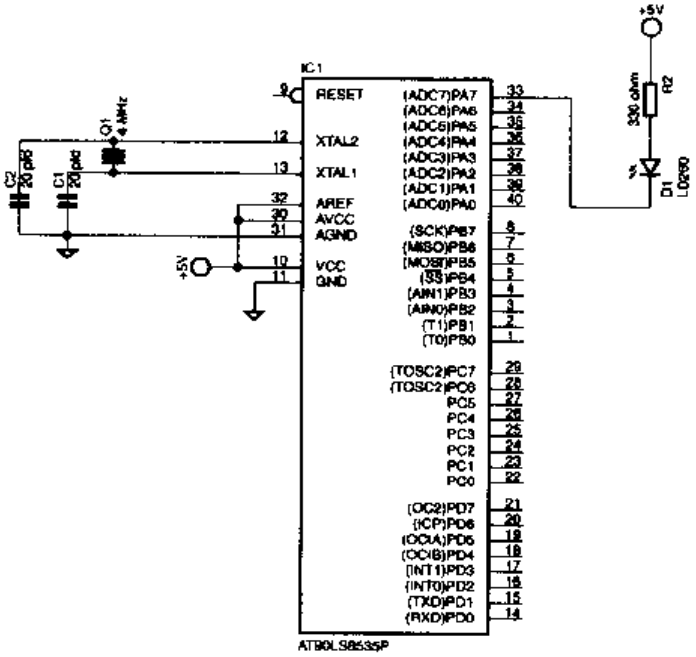


图 2-11　LED 触发硬件连线图

因为计时器被用作“钟摆”，首先必须确定在每次中断发生时加载到计时器的数值。本例中，每隔 0.5 秒触发一次 LED，很显然是需要一个 0.5 秒的计时中断。如果采用 Timer 0，它最慢的预定标器是系统时钟的 1/1024。

4MHz/1024＝3.906KHz

这个频率的时钟信号的间隔是：

1/3.906kHz＝256 微秒

这表明每隔 256 微秒 Timer 0 就会获得一个计数脉冲。Timer 0 是一个 8 位的计时器/计数器，所以它最多只能计算 256 个脉冲，在这里它的最大计时范围是：

256×256 微秒＝65.535 毫秒

对于 500 毫秒的事件来说，65 毫秒显然是不够的。为了计算较长的时段，必须在中断服务程序中放置一个全局计数器。本例中，每 50 毫秒产生一次计时中断，则计数器累计中断次数到

10 才触发 LED(10×50 毫秒＝500 毫秒)。

重新加载到计时器的数值是由程序员决定的。一般都希望该数值能产生一个易于处理的延时，如 1 毫秒或 10 毫秒等。在 4MHz 的系统时钟的情况下，如果采用系统时钟的 8 分频信号，就可以每 2 微秒向计时器发送一个时钟脉冲。假设这个 2 微秒的时钟脉冲连接到 8 位的计数器，那么这个计数器的最大计时范围是：

2 微秒×256＝512 微秒

512 微秒并不是很适合的工作间隔，但是如果只是计数到 250，就可以产生 500 微秒的时间间隔。那么计时器的重新加载数值就是：

256–250 ＝6

翻转是在计数器计数到了 256 时才发生的，而我们需要计数器在计算了 250 次就产生翻转，所以从 6 开始计算。这意味着中断服务程序每隔 500 微秒就执行一次。(2 微秒/时钟周期×250 时钟周期/中断周期＝500 微秒/中断周期)。那么全局的中断计数变量要计数到 1000 才能产生 500 毫秒的间隔(500 微秒×1000＝500 毫秒)。整个程序如下所示。

```
#include <90s8535.h>
unsigned int timecount = 0; //global time counter

//Timer 0 overflow ISR
interrupt [TIM0_OVF] void timer0_ovf_isr(void)
{
    TCNT = 6;                   //reload for 500 us tricks
    if (++timecount == 1000)
    {
        PORTA = PORTA ^ 0X80;   //toggle MSB of port A
        timecount = 0;          //clear for next 500 ms
    }
}

void main(void)
{
   DDRA=0X80;      //Port A MSB as output

   TCCR0=0x02;     //set for clock/8 as counter input
   TCNT0=0x00;   //start timer with 0 in timer

   //Timer 0 interrupt initialization
   TIMSK=0x01;   //unmask Timer 0 overflow interrupt

   //Global enable interrupts
   #asm("sei")
```

```
    while (1)
      ;    //do nothing
}
```

以上程序演示了上面提到的所有概念，每次执行 ISR 的时候，计时器/计数器寄存器 TCCNTO 就会重新设置为 6，再次计算 250 个脉冲到达 256，然后溢出翻转，产生中断。

全局变量 timecount 用于计算中断次数，每次执行中断服务程序时，它就会增加 1。timecount 在 if 语句表达式中被检测，同时加 1。当变量 timecount 达到 1000 的时候，端口 A 的最高位就会被触发，同时为了能进行下一个 500 毫秒的计算，timecount 被重新设置为 0。也可以以这种方式处理其他需要 500 毫秒间隔的事件。

2.7.3　Timer 1

典型的 16 位计时器是 Timer 1，与 8 位的 Timer 0 相比，它显得更加通用和复杂了。除了普通计时器具有的寄存器以外，Timer 1 还有一个 16 位的输入捕捉寄存器(Input Capture Regster)和两个 16 位的输出比较寄存器。输入捕捉寄存器用来检测输入脉冲的宽度和捕捉时间。输出比较寄存器用来产生输出到微控制器的一个输出引脚的频率或脉冲信号。虽然我们是分别讨论这两种模式，但是它们可能(而且在很多情况下)会在同一个程序中混合使用。

Timer 1 在概念上和 Timer 0 有很大的差别。在正常的使用中，Timer 0 的操作通常是停止、开始和重置等；而 Timer 1 则相反，通常是一直运行着的。这使 Timer 1 在用法上存在很大的差别。下面将会讨论这些不同之处，包括 Timer1 的特殊用法。

1. Timer 1 预定标器和选择器

尽管 Timer 1 有许多特别的功能，但是它也是一个二进制计数器。它的计算速度或计时间隔和 Timer 0 一样，都是由输入的时钟信号确定的。和微控制器的其他外设一样，Timer 1 也是通过一个控制寄存器来控制的。

1 号计时器/计数器控制寄存器(Timer/counter ControlRegister 1，TCCR1)，实际上是由两个寄存器组成的：TCCR1A 和 TCCR1B。TCCR1A 控制 Timer 1 的比较模式和脉冲宽度调制模式，这些会在接下来的内容中讨论。TCCR1B 控制 Timer 1 的预定标器、输入多路复用器以及输入捕捉模式。图 2-12 和表 2-7 给出了 TCCR1B 上各位的定义。

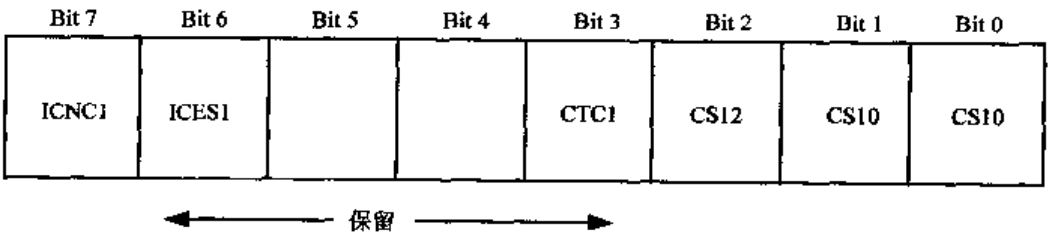


图 2-12　TCCR1B 的位

表 2-7　TCCR1B 的位定义

位	功　能
ICNC1	输入捕捉噪声消除器(1＝有效)
ICES1	输入捕捉边沿选择(1＝上升沿，0＝下降沿)
CTC1	匹配时清除计时器/计数器(1＝有效)
CS12	计数器输入选择位 同 Timer 0 上的定义一样
CS11	
CS10	

TCCR1B 的计数器选择位(Counter Select Bits)控制 Timer 1 的输入选择，它的工作方式和 Timer 0 的一样。事实上，这 3 个选择位为 Timer 1 提供的输入信号和 Timer 0 上的是一样的。

2. Timer 1 输入捕捉模式

利用 Timer 0 来检测时间间隔时，是在事件发生时启动计数器，并在事件结束时停止计数器，最后从计时器的计数器寄存器中读取该事件所用的时间。Timer 1 的处理方法就不同了，因为它总是在运行状态中。要检测某个事件用的时间，Timer 1 捕捉事件发生和结束瞬间的时间数值并保存起来，通过这两个数值相减就可以获得事件经过的时间。所做的工作就像要计算您从历史教室到书店要用多长时间一样。在离开历史教室的时候您记录当时的时间，当到达书店的时候您也记录那时的时间，然后这两个时间相减，就得到要用的时间了。在 Timer 1 中，这些任务是由输入捕捉寄存器(Input Capture Register，ICR1)来管理的。

IRC1 是一个 16 位的寄存器，当微控制器接受某一特定信号时，它就会捕捉 Timer1 当前的读数。引起捕捉的信号可以是微控制器输入捕捉引脚(Input Capture Pin，ICP)上的上升沿或下降沿信号。由图 2-12 可知，引发捕捉的信号是上升沿还是下降沿，是由输入捕捉边沿选择位(Input Capture Edge Select Bit，ICES1)来决定的。设置 ICES1 是使 ICR1 在一个上升沿信号下捕获 Timer 1 的时间；清除 ICES1 则是使 ICR1 在下降沿信号下工作。

很明显，Timer 1 中只是提供了一个捕捉寄存器。所以捕捉到的内容必须在捕捉到以后立即读取，以防止下一次的捕捉操作重写并破坏上一次读取的数据。为此，每当 ICR1 上捕捉到了新的数据时就会产生一个中断请求。每次捕捉中断产生时，程序必须确定该信号是计时的开始还是结束，以使它能恰当地处理 ICR1 中的数据。

Timer 1 还提供了输入噪声消除的功能，以防止混杂在 ICP 上的尖峰信号在错误的时间引起数据捕捉。当噪声消除功能打开时，ICP 上的电平高度必须在有效高度(上升沿信号对应高电平，下降沿信号对应低电平)上维持 4 个连续脉冲周期，微控制器才会认为它是合法的中断信号并捕捉数据。这是为了防止尖峰噪音触发捕捉寄存器。设置 TCCR1B 的输入捕捉噪声消除位(Input Capture Noise Canceller Bit，ICNC1)可以支持输入噪声消除功能(参见图 2-12)。

图 2-13 所示的硬件和下面的示例程序演示了如何使用输入捕捉寄存器。该示例的目的是测

量连接到微控制器 ICP 上的正向脉冲的宽度，并以毫秒形式将结果输出到端口 C。

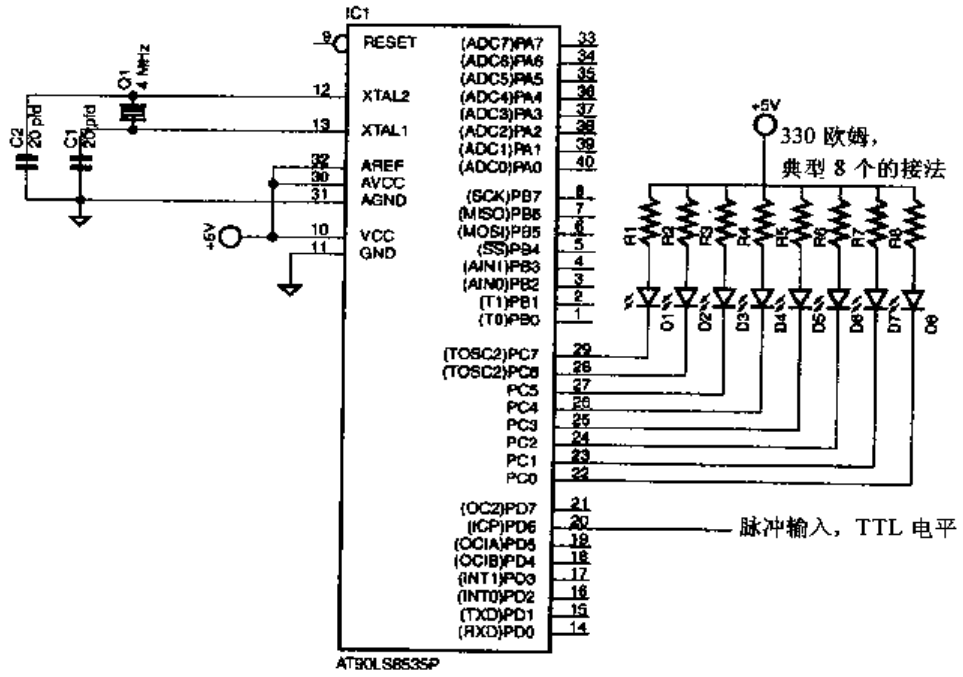


图 2-13　脉冲宽度检测硬件连线图

示例程序中有几个值得注意的地方。#define 语句用于连接输出端口和具可读性的名字，同时用于定义 ICP 引脚，使该引脚可以在程序中被检测到。

在检测脉冲过程中产生溢出时，Timer 1 溢出的 ISR 只是使溢出计数器加 1。溢出的数目用来计算脉冲的宽度。

```
#include <90s8535.h>

//define Port C as output for pulse width
#define pulse_out PORTC

//define ICP so we can read the pin
#define ICP PIND.6

unsigned char ov_counter;       //counter for timer 1 overflow
unsigned int rising_edge,   falling_edge;    //storage for times
unsigned long pulse_clocks;        //storage for actual clock counts in the pulse

interrupt   [TIMi_OVF]   void timerl_ovf_isr(void)   // Timer 1 overflow ISR
{
    ov counter++;       //increment counter when overflow occurs
}
interrupt   [TIMi_CAPT]   void timerl_capt_isr(void)   // Timer 1 input capture ISR
{
    //Check for rising or falling edge by checking the
    //level on ICP. If high, the interrupt must have
```

```
    //triggered by a rising edge and if not,  the trigger
    //must have been a falling edge.
    if (ICP)
    {
        rising_edge = ICR1;    //save start time for pulse
        TCCR1B = TCCR1B & 0xBF;    //set to trigger on falling edge next
        ov_counter = 0;    //clear overflow counter for this measurement
    }
    else
    {
        falling_edge = ICR1;    //save falling edge time
        TCCR1B = TCCR1B | 0x40;    //set for rising edge trigger next
        pulse_clocks =  (unsigned long)falling edge  -
              (unsigned long)  risingedge
              +(unsigned long)ov_counter * 0x10000;  //calculation
        pulse_out = pulse_clocks / 500;    //output milliseconds to Port C
     }
}

void main (void)
{
     DDRC=0xFF;    //set Port C for output
     TCCR1B=0xC2;//Timer 1 input to clock/8,  enable input
                 //capture on rising edge and noise canceller
     TIMSK=0x24;  //unmask timer 1 overflow and capture interrupts

     #asm("sei")  //enable global interrupt bit

     while  (1)
       ;    //do nothing - interrupts do it all
}
```

在 Timer 1 输入捕捉事件的 ISR 检查 ICP 的实际状态，以确定是出现了上升沿还是下降沿信号。如果中断是由上升沿信号触发的(ICP 是高电平)，程序必须重新开始测量一个新的脉冲，同时 ISR 记录这个上升沿出现的时间，把输入捕捉寄存器的触发方式改为下降沿触发(以捕捉脉冲的末端)，并把溢出计数器清零。如果相反，即 ICP 处于低电平，该中断是由下降沿信号触发，已经到达脉冲的末端了。当脉冲的长度计算出来以后，就把结果输出到端口 C，同时把输入捕捉寄存器的触发方式改为由上升沿信号触发，以开始测量下一个脉冲。请注意，CodeVisionAVR 编译器允许用同一个名称 ICR1 访问 16 位的输入捕捉寄存器 ICR1。这些是在 90S8535.h 文件中定义的。但并不是所有的编译器都为 16 位寄存器提供 16 位的访问方式，CodeVisionAVR 也不提供全部 16 位寄存器的 16 位访问方式。详细信息请参见具体处理器的相关头文件，或者 1.11.4 小节中的 sfrb 和 sfrw。

脉冲实际的宽度(毫秒格式)是利用连接到计数器的时钟脉冲速率来计算的。上面例子中的

Timer 1 的时钟是系统时钟(4MHz)的 8 分频，即 500kHz。这是通过在 main()函数的开始处初始化 TCCR1B 设置的。500kHz 的时钟间隔是 2 微秒。在 2 微秒的时钟脉冲下，需要 500 个脉冲才够 1 毫秒，所以该程序中的结果除以 500，就可以得到以毫秒为单位的正确结果了。

3. Timer 1 输出比较模式

输出比较模式是微控制器用来产生输出信号的。输出可以是对称或不对称的方波，并且可以调节它们频率或波形占空比。例如，当要用一个微控制器来播放学校的校歌时，您就可以利用输出比较模式了。在这种情况下，输出比较模式用于产生组成歌曲的音符。

输出比较模式和输入捕捉模式在一定程度上是对立的。在输入捕捉模式中，由外部的信号来触发输入捕捉寄存器去捕捉或保存当前的时间。在输出比较模式中，程序加载一个输出比较寄存器(Output Compare Register)。输出比较寄存器中的数值将会和计时器/计数器中的值比较，两个值相匹配时，就会产生一个中断。这个中断行为就像闹钟一样，促使处理器在刚好需要时去执行和该信号相关联的功能。

除了能够产生中断外，输出比较模式还能自动设置、清除或触发指定的输出引脚。就 Timer1 而言，输出比较模式是由计时器/计数器控制寄存器 1A(Timer/counter Control Register 1 A，TCCR1A)来控制的。图 2-14 和表 2-8 给出了 TCCR1A 各位的定义。

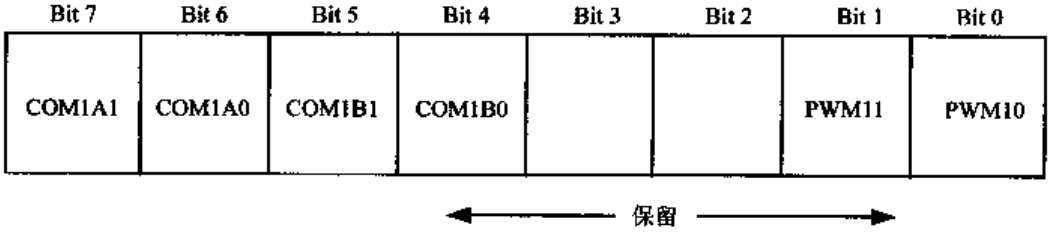


图 2-14　COM1A0 与 COM1A1 控制比较寄存器、COM1B0&COM1B1 控制比较寄存器的比较模式

表 2-8　TCCR1A 各位的定义

COM1x1	COM1x0	功能(根据情况，x 是 A 或 B)
0	0	没有输入
0	1	匹配时触发 OC1x
1	0	匹配时把 OC1x 置 0
1	1	匹配时把 OC1x 置 1

比较控制位决定了当比较寄存器和计时寄存器的数值相等时要执行的动作。相关联的输出引脚可能未受影响、被触发、被设置，也有可能被清零。这个匹配同时产生一个中断。该中断的服务程序的目的是为下一次比较复位或重新加载比较寄存器。

例如，假设微控制器要产生一个 20kHz 的方波信号，则该模式的工作步骤如下：

(1) 当第一个匹配出现时。输出位被触发，同时产生中断

(2) 在中断服务程序中，程序会计算下一个匹配出现的时间，并把相应的数值加载到比较寄存器中。本例中，20kHz 的波形应该有 50 微秒的长度，每半个波形有 25 个微秒。所以，从

一个触发到下一个触发的时间应该是 25 微秒。程序利用连接到计时器的时钟频率，就可以计算出 25 微秒内所需要的时钟数。

(3) 这个数值会和比较寄存器中的数值相加，再把结果重新加载到比较寄存器，以引发下一次的触发，并重复进行计算和重新加载循环。

图 2-15 和下面的程序给出一个具有代表性的例子，在硬件和软件上演示了以上的概念。请注意，因为所有的信号都是由微控制器内部产生提供的，所以不需要任何外部硬件。

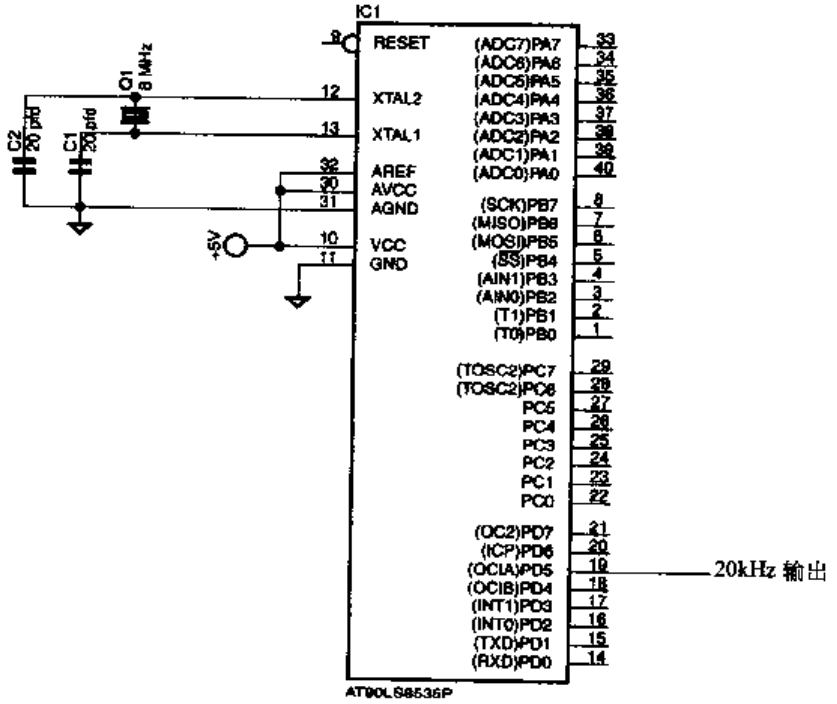


图 2-15　20kHz 例子的硬件图

```
#include <90s8535.h>

// Timer 1 output compare A interrupt service routine
interrupt  [TIMi_COMPA]  void timerl_compa_isr(void)
{
      OCR1A = OCR1A + 25;        //set to next match  (toggle)  point
}

void main (void)
{
      DDRD=0x20;          //set OC1A bit for output

      TCCR1A=0x40;      //set prescaler to Clock/8   (1 us clocks)
      TCCR1B=0x02;      //enable output compare mode to toggle OC1A pin on match

      TIMSK=0xl0;       //unmask output compare match interrupt for
      register A
```

```
    #asm("sei")        //set global interrupt bit

    while  (1)
        ;          //do nothing
}
```

在上面的程序代码中，最先的初始化是在 main()函数中进行的。寄存器 DDRD 设置为 0x20，以便输出比较位(OC1A)就和输出比较寄存器 A(OCR1A)相关联，同时该位还被设置为输出模式，使输出信号能够出现在该位上。TCCR1A 和 TCCR1B 把预定标器设为时钟的 1/8(该例中，时钟＝8MHz，时钟的 1/8＝1MHz)，并把输出比较寄存器 A 设为输出比较模式，这样当匹配时，比较寄存器 A 就会触发输出位 OC1A。最后通过在计时器中断屏蔽寄存器(Timer Interrupt MaskRegister，TIMSK)设置 OCIE1A 来取消比较中断的屏蔽。

图 2-16 所示的信息用于计算添加到比较寄存器中的数值，以决定下一次的触发时间。在这里，半个波形的长度是 25 微秒。由于计数器的时钟频率是 1MHz，那么两个匹配点(波形触发点)之间的时钟周期计算公式如下：

两个触发点之间的时间长度/时钟周期＝间隔数目

在以上例子的情况下：

25 微秒 / 1 微秒每时钟周期＝每触发点 25 个时钟周期

因此，在这个例子里每次发生匹配中断时，比较寄存器的值就会增加 25，以确定下一次匹配的时间，从而在输出端输出相应的波形。

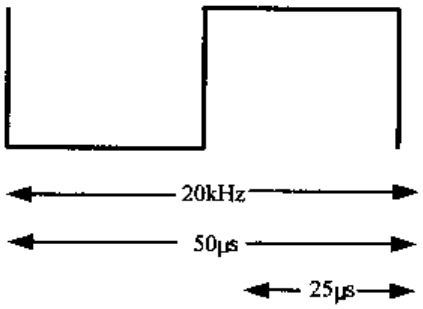


图 2-16　20kHz 的波形图

另外，存在一个关于比较寄存器的非常重要的问题，特别是和计算下一个匹配数值密切相关，这个问题就是该数值和比较寄存器的数值相加以后的结果大于 16 位可以表示的范围。例如，输出比较寄存器中的数值是 65000，而下一个的间隔是 1000，那么

65000＋1000＝66000(一个大于 65535 的数)

只要这个计算采用的是无符号的整型数，那么那些超出 16 位的位就会被截断，真正的结果会是：

65000＋1000＝464 (把 66000 的第 17 位截掉，得到 464)

这样就可以正确地工作了，因为输出比较寄存器和计时器/计数器都是 16 位的。计时器会从 65000 一直计数到 65535(共计了 536 次)，然后再计数 464 次就会达到匹配值了。536 次加上

464 次，刚好是我们所需要的 1000 次。换句话说，只要在相关的数学运算中应用了无符号的整型数，就算计时器/计数器和比较寄存器同时发生溢出，也不会有问题。

4. Timer 1 脉宽调制器模式

脉宽调制(Pulse Width Modulation，PWM)模式是许多提供数模转换的方法中的一种。PWM 模式就是：微控制器的方波输出循环通过过滤实际的输出波形就可以得到一个可调节的平均直流输出信号，从而提供不同的直流输出信号。图 2-17 演示了这个原理。

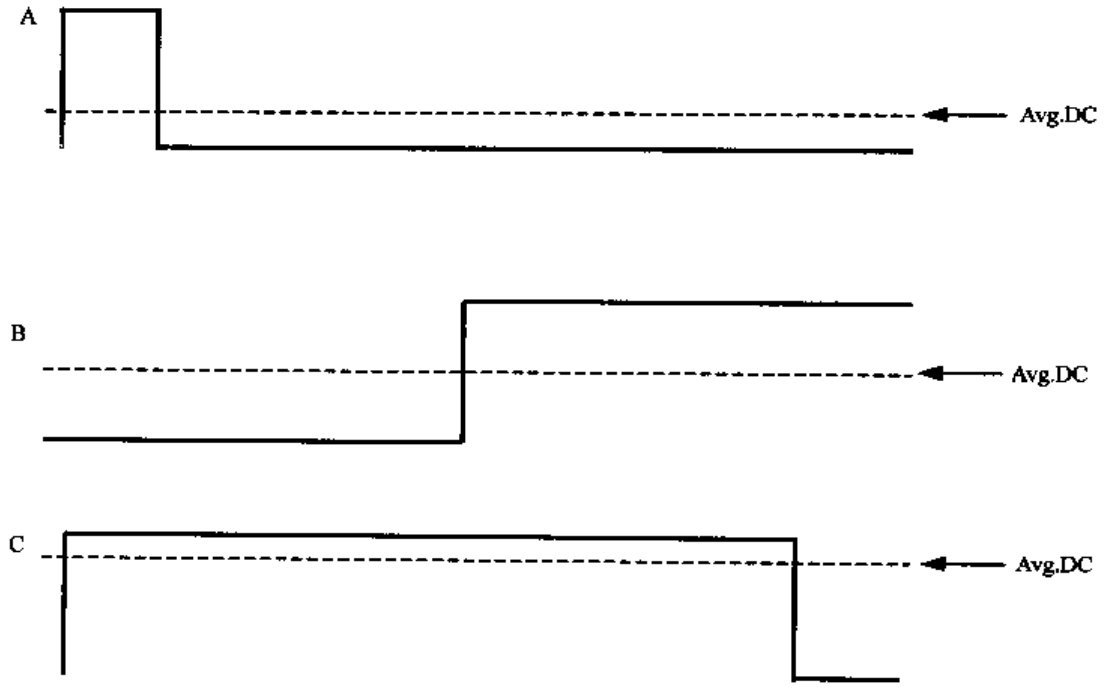


图 2-17　PWM 波形图

如图 2-17 所示，改变波形的占空比或高电平在周期中的比例都会改变它的平均直流电压。把波形过滤掉以后就可以用来控制模拟设备，这样就做成了一个数模转换器(Digital-to-Analog Converter，DAC)。图 2-18 给出了一些 PWM 控制模式的示例。

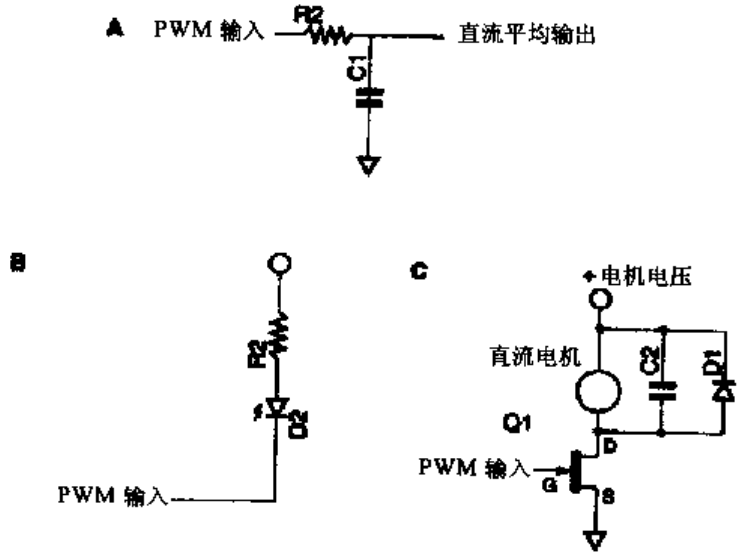


图 2-18　PWM 的示例

图 2-18 中的 A 示例中，RC 电路提供过滤功能。RC 电路的时间常数(振荡周期)必须明显大于 PWM 波形的周期。B 示例中 LED 的亮度是由 PWM 波形控制的。本例中逻辑 0 是点亮 LED 的，所以它的亮度应该和 PWM 成反比。过滤功能是人眼提供的，因为人的眼睛是不能分辨高于 42Hz 的频率，也就是我们所说的闪烁频率。所以在这种情况下，PWM 波形的频率必须高于 42Hz，否则将会看到 LED 在闪烁。

图 2-18 中的 C 示例是用 PWM 来控制直流电机的。该电路的过滤功能大部分是由直流电机的机械惯性和线圈自感系数提供的，这使电机速度的改变无法跟上波形的改变。当然电容也提供了额外的过滤。二极管的存在是很重要的，它是用来吸收感应电机内电流通断引起的尖峰电压。

利用 Timer 1 制作 PWM 的一种方法是利用输出比较寄存器，每次发生匹配时，改变被重新加载的递增值，以产生 PWM 波形。但是 Timer1 向提供 PWM 内嵌了的方法，而不需要程序不断地改变比较寄存器来制作脉宽调制器。

为了提供 PWM，Timer 1 要改变它的工作模式。在 PWM 模式工作时，由于 Timer1 时而向上计数，时而向下计数，所以使其他使用 PWM 模式的 Timer1 模式难以使用。在 PWM 操作中，Timer 1 从 0 计数，一直到达峰值(Top Value)，然后转向下计数，直至 0。峰值是由分辨率确定的。PWM 提供有 8 位、9 位和 10 位不同的分辨率，它们由 PWM 的选择位(Select Bit)TCCR1A(参照图 2-14)确定。表 2-9 所示为这些选择位的作用。

表 2-9　PWM 的选择位

PWM 选择位		PWM 分辨率	Timer 峰值
PWM11	PWM11		
0	0	PWM 关闭	
0	1	8 位	255(0xff)
1	0	9 位	511(0x1ff)
1	1	10 位	1023(0x3ff)

表 2-9 显示分辨率的选取会确定计数器来回计数的峰值，同时也会影响 PWM 的振动频率。例如，选择 9 位的分辨率可以得到 511 的计数峰值，而 PWM 的振动频率计算如下(假设系统的时钟是 8MHz 的)：

$$f_{PWM} = f_{系统时钟} / (预定标器值 \times 2 \times 峰值) = 8MHz / (8 \times 2 \times 511) = 978.5Hz$$

分辨率决定了 PWM 控制的优劣或精确度。在 8 位分辨率的模式中，PWM 控制精度可以到达 1/256；在 98 位的模式中，PWM 控制精度可以到达 1/512；在 10 位的模式中，PWM 控制精度可以到达 1/1024。分辨率必须和频率相互协调，以取得最优的组合。

PWM 模式输出波形的实际占空比是由加载到计时器/计数器的输出比较寄存器的数值决定的。在正常的 PWM 模式下，计数器向下计数时遇到匹配，就把输出位设置为高电平；如果是在向上计数时遇到匹配，就将输出位的值清除，即设置为低电平。在这些方式下，如果加载到

输出比较寄存器的数值等于峰值的 20%，那么就可以产生 20%的占空比的波形。同样可以为应用程序提供反向的 PWM 波形，以控制直接连接到 PWM 输出引脚，以吸收电流方式工作的 LED 的亮度——当加载输出比较寄存器的数值是峰值的 80%时，输出的方波波形比例中就有 80%的波形是低电平。

图 2-19 和表 2-10 给出了一个具有代表性的关于 PWM 应用程序示例的硬件和软件。这个示例可以根据端口 C 上不同的逻辑值提供 4 种不同占空比(当时钟频率接近 2kHz 的时候分别为 10%、20%、30%和 40%)。

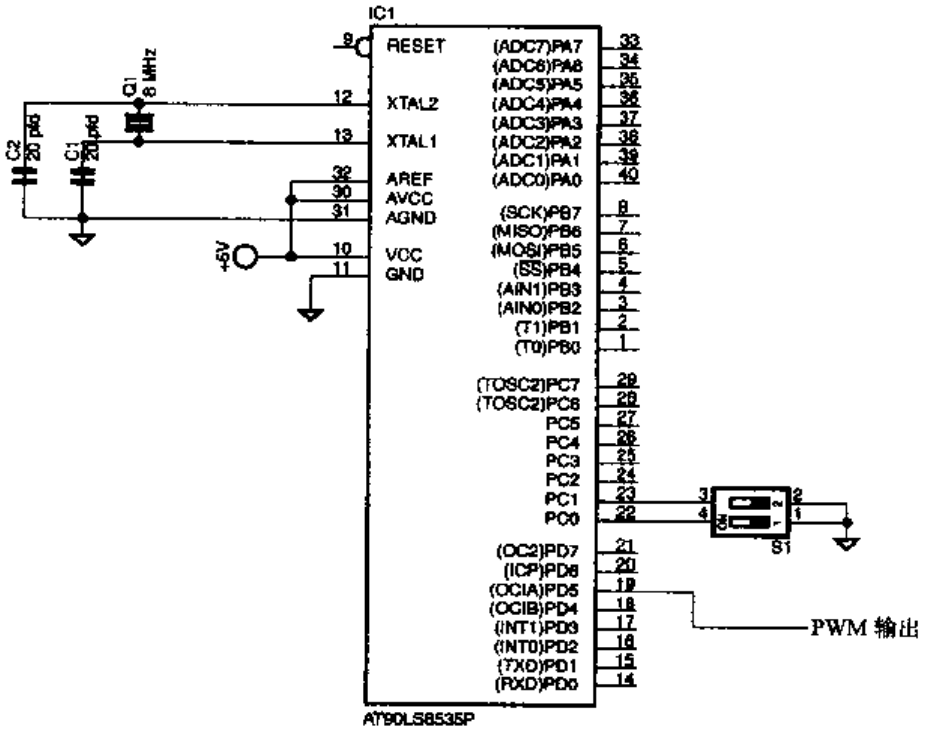


图 2-19　PWM 硬件线路图

要设置准确的输出频率，需要反复设置计时器预定标器(时钟分频)和分辨率，以取得尽可能和期望频率一样的输出。表 2-10 所示为在系统时钟是 8MHz 的情况下，所有分频率和分辨率的组合下的输出频率(利用上面提到的 f_{PWM} 的计算公式计算)。从该表中可以选取和期望频率最接近的组合方式。如果遇到需要一个特殊分辨率的 PWM，由于表中的选项被实际的分辨率限制了，所以得到的频率可能就没有那么接近要求了。

表 2-10　频率选择表

分频比例因子	PWM 的模式		
	8 位(Top=255)	9 位(Top=511)	10 位(Top=1023)
系统时钟	f_{PWM}=15.76k Hz	f_{PWM}=7.86k Hz	f_{PWM}=3.916k Hz
系统时钟/8	f_{PWM}=1.96k Hz	f_{PWM}=978 Hz	f_{PWM}=489 Hz
系统时钟/64	f_{PWM}=245 Hz	f_{PWM}=122 Hz	f_{PWM}=61 Hz
系统时钟/256	f_{PWM}=61 Hz	f_{PWM}=31 Hz	f_{PWM}= 15 Hz
系统时钟/1024	f_{PWM}=15 Hz	f_{PWM}=8 Hz	f_{PWM}=4Hz

$f_{PWM}=f_{系统时钟}$ /(比例因子×2×峰值)

例如，表 2-10 中给出的 8 位分辨率和 8 分频系统时钟的组合可以产生 1.96kHz 的频率，这个和期望的频率 2kHz 是很接近的。同时表 2-10 也给出了其他组合的频率，但是 PWM 的分辨率是相当有限的。如果项目需要一个特殊的 PWM 频率，很可能就要改变系统时钟的晶体振荡器，以产生期望的精度。

下面的程序利用一个 switch/case 语句来选择输出的 PWM 波形。#define 语句用于屏蔽端口 C，只允许它的最低两位作为输入。把端口 C 的值设为 0x3，这是为相同的最低两位支持内部的上拉功能，这样就可以不使用外部的上拉电阻。

TCCR1A 和 TCCR1B 则是被初始化来选择 PWM 模式和预定标器的，以提供期望的频率输出。

在许多微控制器应用中，处理器时间都是很珍贵的。在这个程序中，用 if 语句来判断变量 oldtogs，这样只有在开关改变了的时候才向脉宽调制器写新数据。这样就避免了连续向输出比较寄存器写入数据。

```
#include <90s8535.h>

//use a 'define'   to set the two lowest bits of port C as the control input.
#define PWM_select   (PINC & 3)

void main(void)
{
   unsigned int oldtogs;      //storage for past value of input

   PORTC=0x03;      //enable the internal pull-ups for in the input bits

   TCCR1A=0x91;     //compare A for non-inverted PWM and 8 bit resolution
   TCCR1B=0x02;     //clock / 8 prescaler

   while  (1)
   {
        if  (PWM_select   != oldtogs)
        {
            oldtogs = PWM_select;            //save toggle switch value
            switch   (PWM select)
            {
                case 0:
                    OCR1A = 25;     //10 % duty cycle desired
                    break;
                case 1:
                    OCR1A = 51;     //20% duty cycle desired
                    break;
                case 2:
                    OCR1A = 76;     //30% duty cycle desired
                    break;
```

```
                case 3:
                    OCR1A = 102;   //40% duty cycle desired
                }
            }
        }
    }
```

2.7.4 Timer 2

通常 Timer 2 是一个功能和 Timer 1 相似的 8 位计时器/计数器，也有输出比较和 PWM 功能(当然这可能会随不同的微控制器而不同)。

和 Timer1 最重要的差别在于，Timer 2 可以使用一个独立于系统时钟的晶体振荡器作为时钟脉冲。外部振荡时钟的选择是通过设置异步状态寄存器(Asynchronous Status Register，ASSR)中的 AS2 位来完成的。ASSR 各位的定义如图 2-20 所示。

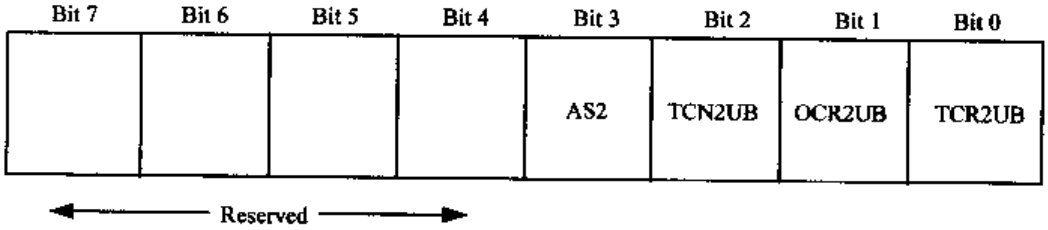


图 2-20　ASSR 的位定义

设置 AS2 位允许 Timer2 使用外部的振荡器作为时钟脉冲。这就意味着 Timer 2 的时钟脉冲和系统的时钟是异步的。ASSR 寄存器中其他三位是被程序员用来确保程序不会在硬件更新 Timer 2 寄存器的同时往里面写数据。这是非常有必要的，因为 Timer 2 的振荡器和系统的不同步，如果在 Timer 2 正更新寄存器数据时向寄存器写入数据，很可能会破坏里面的数据。

Timer 2 的操作是由一个单独的控制寄存器 TCCR2 来控制的。TCCR2 的位定义如图 2-21 和表 2-11 所示。

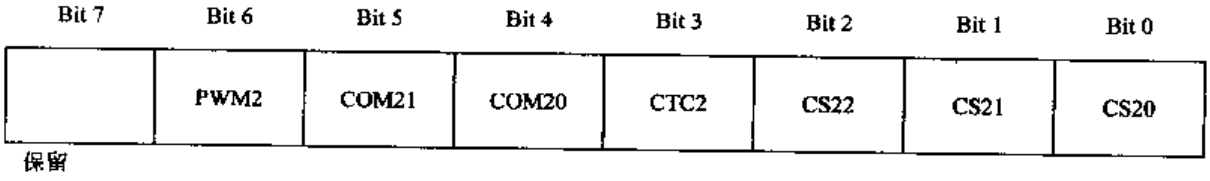


图 2-21　TCCR2 的位定义

例如，使用 32.768kHz 的晶体振荡器允许把 Timer 2 作为实时时钟的时基来用。需要时，这样的 Timer 2 可以使微控制器保持精确的时间。

因为 Timer 2 的功能和 Timer 1 的功能非常相像，所以在这里就不再赘述了。

表 2-11　TCCR2 的位定义

位	说　明
PWM2	设置该位支持 Timer 2 的 PWM 功能
COM21	这两位设置输出比较模式的功能，和 Timer1 的 COM1x1 和 COM1x0 一样
COM20	
CTC2	设置匹配时支持清除计数器
CS22	Timer 2 计数器预定标器选择位，详情参看 AVR 的说明书。
CS21	
CS20	

第 2 章示例项目：C 部分

这是示例项目 2 赛车数据采集系统的第二部分。这部分将会涉及到用计时器来设置一秒钟间隔的测量周期，并利用它们来测量两个 rpm 数据。

1. 用 Timer 0 来设置一秒钟的测量周期

前面已经讨论过如何用计时器来做程序的“钟摆”或者产生延时时间间隔。这次，您需要产生一个相当长的(相对于微秒而言)时间间隔，同时还有别的基于中断的进程在运行。所以最好在这个“钟摆”中使用尽可能慢的时钟，以便把花费在该“钟摆”程序中的时间最小化。

进一步检查要使用的 Mega163 系统，会发现它的系统时钟是 8MHz。而 Timer 0 预定标器的分频比例因子可以是 1、8、64、256 或 1024。选择 64 的分频值可以使接到 Timer 0 上的时钟频率为 125kHz。

现在就能计算出一秒钟内该时钟脉冲出现的个数。这个非常简单的，125kHz 的脉冲在一秒内有 125 000 个脉冲。Timer 0 是一个 8 位计数器，它最多只能计数到 255，所有您选择了让它在每个“钟摆”内计数到 250。125 000/250＝500，即该中断连续出现 500 次才代表 1 秒钟。

以下的代码是 main()函数中初始化 Timer 0 的部分：

```
//Timer/Counter 0 initialization
TCCR0 = 0x03 ;     // clock set to clk/64
TCNT0 = 0x00;      // start timer0 at 0
```

以下的代码是放置 main()函数之前的，作为 Timer 0 溢出的服务程序：

```
int time_cntr = 0;     //global variable for number of Timer 0 overflows

// Timer 0 overflow interrupt service routine
interrupt [TIM0_OVF] void timer0_ovf_isr(void)
{
    TCNT0 = 6;              //reload timer 0 for next period
    if (time_cntr++ == 500) //check for one second, increment counter
    {
            if (data_set_cntr < 120)   //record data if less than 120 sets
```

```
            {
                e_rpm[data_set_cntr] = current_e_rpm;     //record engine rpm
                s_rpm[data_set_cntr] = current_s_rpm;     //resocrd shaft rpm
                temp[data_set_cntr++] = current_temp;     //record engine temp
            }
            time_cntr = 0;        //reset counter for next 1 second interval
        }
    }
```

请注意，变量 data_set_cntr 是作为保存当前温度数据的一部分采用后加方式的。

2. 用 Timer 1 测量引擎的 rpm

输入捕捉功能用来捕捉引擎 rpm 脉冲间的时间间隔。可以利用该时间间隔来计算 rpm(Revolutions Per Minute，每分钟转数)。请注意，这辆车是一个 4 冲程的引擎，它每转两圈就发出一个脉冲。

系统时钟是 8MHz，所以采用 8 分频就可以得到一个 1MHz 的脉冲，也就是一个方便的 1 微秒的时间间隔。用该时钟脉冲驱动 Timer 1，当 Mega163 的 ICP 出现下降沿信号时，输入捕捉寄存器就会被用来捕捉 Timer 1 上的读数，与此同时还会产生一个中断。在中断服务程序中，当前捕捉到的数据与前一次捕捉到的数据进行比较，以确定两个脉冲之间的时间间隔，并利用该时间间隔来计算引擎的 rpm。

以下的代码在 main()函数中用于把 Timer 1 初始化为以下降沿信号：

```
// Timer(s)/Counter(s) Interrupt(s) initialization
TIMSK=0x21;    // timer0 overflow, ICP interrupts enabled

// Timer/Counter 1 initialization
TCCR1A=0x00;
TCCR1B=0x02;          //prescaler = 8, capture on falling edge
TCNT1H=0x00;          //start at 0
TCNT1L=0x00;
```

以下代码是捕捉时间的 ISR 程序：

```
unsigned int previous_shaft_capture_time;   //saved time from previous capture

// Timer 1 input capture interrupt service routine
interrupt [TIM1_CAPT] void timer1_capt_isr(void)
{
    unsigned int current_capture_time, period;    //current time and period

    current_capture_time = (256* ICR1H) + ICR1L;      //get captured time
    if (current_capture_time > previous_capture_time)    //check rollover
        period = current_capture_time - previous_capture_time;
    else
        period = 0xFFFF - current_capture_time + previous_capture_time;
```

```
        current_e_rpm = (unsigned long)120E6 / (unsigned long)period;
        previous_capture_time = current_capture_time;   //save for next calculation
    }
```

在上面的代码中，全局变量 previous_capture_time 被初始化，以便用来保存前一次捕捉到的数据。ISR 函数读取当前捕捉到的数据，然后结合上一次捕捉到的数据来计算引擎的 rpm。请注意，代码中的 if 语句是用来检测计数器是否翻转了。由于 16 位的计时器/计数器在检测过程中会由 0xFFFF 翻转到 0x0000，所以这个判断是很有必要的。ISR 程序最后的语句是保存 current_capture_time 的值，以作为下一次脉冲的 previous_capture_time。

转速 rpm 计算如下：

RPM＝1E6 微秒/秒×1 脉冲/时间间隔(微秒表示)×2 圈/脉冲×60 秒/分钟

请注意，强制转换用于确保大量数值的准确性。同时，常量的合并也缩短了计算公式的长度。

3. 利用 Timer 1 测量传动轴的 rpm

这个测量相对会有一点复杂，因为 Mega163 上再也没有第 2 个捕捉寄存器了，并且传动轴的信号是连接到 INT0 上的。在中断发生时，您可以通过在 INT0 中读取 Timer1 的数值来建立自己的捕捉寄存器。而其余的功能就和上面测量引擎转速 rpm 差不多：

```
unsigned int previous_shaft_capture_time;   //saved time from previous capture

// External Interrupt 0 service routine
interrupt [EXT_INT0] void ext_int0_isr(void)
{
    unsigned int current_capture_time, period;     //current time and period

    current_capture_time = TCNT1;   //get Timer 1 time
    //check for roll-over
    if (current_capture_time > previous_shaft_capture_time)
            period = current_capture_time -
            previous_shaft_capture_time;
    else
        period = 0xFFFF - current_capture_time +
            previous_shaft_capture_time;
    current_s_rpm = (unsigned long)60E6 / (unsigned long)period;
    previous_shaft_capture_time = current_capture_time;   //save for next calculation
}
```

除了每个脉冲对应一圈，而不是两圈以外，计算方法和引擎转速的计算基本相同。连一些局部变量的名字都是一样的，由于它们是局部的，所以也是允许的。不过在以后调试程序，追踪这些同名变量的值时，可能会遇上调试困难。

2.8 使用 UART 进行串行通信

串行通信(Serial Communication)就是利用一根线来传送多位数据的过程。它是源于电报的一个分支，只是电报的位信息是由点和短横线组成的摩西码(Morse Code)。串行字节的每一位都会以时间来分开，所以接收设备可以确定每一位的逻辑电平。

微控制器利用 UART 来和其他设备进行通信。这些设备可以是 CodeVision AVR 的终端工具(用来进行故障诊断和程序调试)，其他需要与控制系统通信的微控制器，或者是要和微控制器通信以一起完成某一任务的 PC。

串行通信的常用方式，也是本节将要讨论的内容，是异步(Asynchronous)串行通信。说它是异步的原因是，发送器和接收器并不需要一个同步的时钟来同步检测数据。异步串行通信中使用了一个开始位和一个结束位添加到数据字节中，以便接收器可以确定每个位的时间。

图 2-22 显示了一个标准异步串行通信字节中的元素。同时也显示了串行字的波形和每一位的定义。串行传输线上空载的时候是逻辑 1，当它跳转到逻辑 0 时，就预示着开始位的开始。开始位占用一个位的宽度，之后紧跟着 8 个数据位。数据位在串行线上是以反序排列的，最低位出现在最前面，而最高位排在最后面。最高位数据后面的是结束位，结束位是逻辑 1，和空载状态时是相同的。

State	Idle	Idle	Active	Active	Active	Active	Active	Active	Active	Active	Active	Active	Idle
Logic Level	1	1	0	1	0	0	0	0	0	1	0	1	1
Bit Type			Start	Data	Data	Data	Data	Data	Data	Data	Data	Stop	
Data Bit Number				0	1	2	3	4	5	6	7		

图 2-22　串行通信字节格式

开始位的下降沿信号会触发串行接收器开始计时接收。从开始位的下降沿处开始，接收器等待 1.5 个位的时间，然后对接收线取样以获得第一位数据的值。之后，接收器每隔一个位的时间就采样一次，这样每次采样的值都是该位的中点，可以获得最大的可靠性。

幸运的是，所有和串行字节格式相关的计时、串行位的采样、额外的开始结束位都会由通用异步接收/发送器(Universal Asynchronous Receiver-transmitter，UART)自动处理，而不需要程序员的直接干涉。

唯一困扰程序员的是要确保串行通信中发送器和接收器参数相匹配，包括设置正确的数据位数(通常是 8 位)，决定是否包括奇偶校验位(通常是不包括)，设置波特率等。波特率是串行通信的速度，也决定了每一位的长度。波特率的定义为每位所占用的时间的倒数(波特率＝1/每一位所占用时间)。

上面讨论到的所有关于串行通信的波形和参数都涉及要被传送的信息。串行通信的信息是独立于携带它们的媒介的。传输媒介可以是一根电线、一束红外线、无线电通信线路或其他方式。

最常用的传输协议被定义为 RS-232。它是为了能用电缆来提供可靠的长距离串行通信而开发的。RS-232 是一个反向定义模式：逻辑 1 由远低于 0 伏的－3 伏的电压表示；逻辑 0 则是由确定为 3 伏的电压表示。用非零的电压来表示逻辑电平，是为了能够检测一些硬件错误，因为当通信意外中断时，接收器获得的电压信号是 0 伏的。大部分微控制器都是采用 RS-232 与 PC 进行通信的。

像大部分 C 语言编译器一样，CodeVision AVR 提供了处理常见串行通信任务的内置库函数。这些任务通常都会和某种终端设备通信，例如在 PC 上执行一个终端程序把在键盘上输入的字符连续地传送出去，并接受从微控制器连续的字符，然后把它们显示在窗口上。CodeVision AVR 在它的开发环境中提供了一个内嵌的终端模拟器，这对于程序员来说是非常方便的。标准的库函数都包含在头文件 stdio.h 中，只要在 C 程序中包含它就可以了。

利用带有库函数的头文件可以使串行通信变得非常简单。下面的示例程序演示了在一个终端上打印 Yabba Dadda Do。

```
#include <90s8535.h>
#include <stdio.h>     // Standard Input/Output functions

void main(void)
{
    UCR=0x18;                 //serial port initialized
    UBRR=0x33;                //set baud rate

    printf ("\n\rYabba Dabba Do");      //print phrase

    while (1)
        ;                     //do nothing else
}
```

就像上面的简单示例所显示的那样，除了该程序中 main()函数的头两行用来初始化串行端口的代码以外，利用库函数的串行通信真的是极其简单。有关 printf()函数的详细信息参见第 3 章。

UART 的接口包括 4 个寄存器：UART 控制寄存器(UCR)、UART 波特率寄存器(UBRR)、UART 状态寄存器(USR)和 UART 数据寄存器(UDR)。UDR 实际上是两个共享同一个 I/O 地址的寄存器：一个只读寄存器和一个只写寄存器。前者用于放置接收到的数据，后者用于放置将要发送出去的串行字节。所以，当程序读 UDR 的时候，其实是读取接收 UDR 来得到以串行方式接收到的数据；当程序把数据写到传送 UDR 的时候，它就是写将要以串行方式进行传送的数据。

图 2-23 和表 2-12 给出了 UART 控制寄存器(UCR)上各位的定义。

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
RXCIE	TXCIE	UDRIE	RXEN	TXEN	CHR9	RXB8	TXB8

图 2-23　UART 控制寄存器

表 2-12　UART 控制寄存器的位

位	说　明
RXCIE	支持接收中断屏蔽位，设置非屏蔽则允许该中断
TXCIE	支持发送中断屏蔽位，设置非屏蔽则允许该中断
UDRIE	UART 数据寄存器为空(UART Data Register Empty)中断屏蔽位，设置非屏蔽则允许该中断
RXEN	设置使串行接收器有效
TXEN	设置使串行发送器有效
CHR9	设置允许 9 位的串行字符
RXB8	以 9 位的模式接收 9 位的数据
TXB8	以 9 位的模式发送 9 位的数据

UART 控制寄存器是用来初始化 UART 和设置 UART 的功能。它的最高三位是和 UART 相关联的 3 个中断的屏蔽位。第 1 个中断发生时是 UART 接收到了一个字符；第 2 个中断发生是 UART 成功地完成了一个字符的传送；第 3 个中断发生是传送 UDR 已经为传送下一个字符做好了准备。

UART 的当前状态会反映在 UART 状态寄存器中(USR)。USR 上的位信息可以指示什么时候接收到了一个字符，什么时候发送了一个字符，什么时候 UDR 是空的，或者其他任何可能发生的错误。图 2-24 和表 2-13 所示为 USR 上各位的定义。

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
RXC	TXC	UDRE	FE	OR			
					← Reserved →		

图 2-24　USR 的位定义

表 2-13　USR 的位定义说明

位	说　明
RCX	设置以指示接收到一个串行字符
TCX	设置以指示已经发送了一个串行字符
UDRE	设置以指示发送 UDR 已经为空
FE	设置以指示有一个成帧错误
OR	设置以指示有一个 overrun 错误

```
void sendmsg (char *s)
{
  qcntr = 0;              //preset indices
  sndcntr = 1;            //set to one because first character already
  sent
  queue[qcntr++] = 0x0d;    //put CRLF into the queue first
  queue[qcntr++] = 0x0a;
  while (*s) queue[qcntr++] = *s++;       //put characters into queue
  UDR = queue[0];       //send first character to start process
}

void main(void)
{
  UCR=0x58;                  //enable receiver, transmitter and transmit interrupt
  UBRR=0x33;                 //baud rate = 9600

  #asm("sei")                //global interrupt enable

  while (1)
  {
    if (USR & 0x80)          //check for character received
    {
        ch = UDR;          //get character sent from pc
        switch (ch)
        {
          case 'a':
            sendmsg(msg1); //send first message
            break;
          case 'b':
            sendmsg(msg2); //send second message
            break;
          default:
            sendmsg(msg3);   //send default message
        }
      }
    }
}
```

Main()函数的开头两行初始化 UART。UCR 被设置为 0x58，使接收器、发送器及发送中断可用。UBRR 被设置为 0x33，即波特率为 9600。一旦 UART 初始化和中断设置完成后，程序就进入 while(1)循环，并用一个 if 语句不断地检查是否接收到字符。If 语句里的表达式在接收到一个字符后其值就会变为 TRUE。因为接收到一个字符的时候，USR 中的 RXC 位就会被设置，而这一位正是 if 语句检测的值。这是一个用手工方法不断询问串行端口状态以确定何时接收到字符的例子。

当 if 语句确定接收到了一个字符时，程序马上从 UDR 中读取该字符并用于 switch 语句，以确定该发送哪一条消息。要发送的消息都是由多个字符组成的(例如“That was an a.”)。每条

消息在发送的时候都要附上回车(Carriage Return，CR)和换行(Line Feed，LF)符，这样每条消息都会从新的一行开始输出。最长的一条消息由 25 个字符组成，加上 CR 和 LF 字符就是 27 个字符。在 9600 的波特率(每 1.04 毫秒发送一个串行字节)下发送这个消息要用 27 毫秒。为了发送这些消息而要微控制器等待 27 微秒是不合适的，因为在这期间它可以执行大约 216 000 条指令(8 条指令/微秒×27 毫秒×1 000 微秒/毫秒＝216 000 条指令)。为了把微控制器从等待消息发送中解放出来，在实际发送消息时会用一个 FIFO(First In First Out，先进先出)的队列结合发送中断一起使用。

队列是一个以先进先出的原理工作，用于暂时存储数据的设备。这里该队列是由一个数组变量 queue 来实现的。索引 qcntr 用于指示要放到队列的新数据的位置。当一个新的字节加入队列时，索引就会增加 1，这样就可以依次加入下一个字节，直到把所有字节都加入到队列中。

另一个索引 sndcntr 是用来回收队列中的数据的。当从队列中取回一个字节时，该索引也会增加 1。当两个索引相等时，程序就知道队列已经为空了。

实际上，CodeVision AVR C 语言编译器可以利用它的代码生成器功能来提供发送队列和接收队列中的一个或者全部。这个示例演示队列的工作原理，只是为了教学目的。

现在回到示例程序的代码中，在 switch 语句中调用的 sendmsg()函数把消息装进队列中，并开始传送工作。switch 语句在调用该函数的时候向正确消息传递一个指针(作为地址)。该函数首先把 CR 和 LF 放入队列，然后再利用指针把消息传送到队列里面。最后，它把队列中的第一位字节写到 UDR 中，以开始传送过程。该字节传送完成，就会发生 TXC 中断，然后 ISR 把队列中下一个字符加载到 UDR 中。这样不停地循环，直到两个索引相等为止，即队列为空了。有关队列功能的详细的阐述请参考第 3 章。

2.9　模拟接口

尽管数字设备很流行，但是世界的本质本来就是模拟的。微控制器要先把模拟数据转换成为数字形式才能对它们进行处理。AVR 微控制器同时拥有一个模数转换器(Analog to Digital Converter，ADC)和一个模拟比较器。这些模拟接口都会在本节中介绍，包括模数转换的简单背景知识。

微控制器利用模数转换器转换温度和电压等模拟量(例如一个低电监视器)，把音频信号转成为数字形式，作为一个有额外功能的主机。

2.9.1　模数转换背景知识

模数转换(或数模转换)基本上是一个比例上的问题。也就是说，由 ADC 产生的数字值是和输入电压与转换器量程的比值相关的。例如，如果 2V 的电压输入到一个满量程为 5V 的 ADC，则数字输出结果应该是 ADC 输出的最大数字量的 40%(2V/5V＝0.4)。

可用的 ADC 可以提供各种和输出数字范围。输出的数字范围通常以“位”来表示，如 8 位，10 位等。输出的位数决定了可以从转换器输出端读取的数值范围。一个 8 位转换器可以提供 0～255(2^8-1)数值范围的输出，而 10 位转换器可以提供 0～1023($2^{10}-1$)数值范围的输出。

前面的例子——2V 的电压输入到一个量程为 5V 的转换器——中，8 位的转换器会输出 255

的 40%，即 102。比例转换的总结如以下公式：

$$\frac{V_{in}}{V_{fullscale}} = \frac{X}{2^n - 1}$$

在上面的公式中，X 是数字输出，n 是数字输出的位数。利用上面的公式，解出 X，您就可以知道计算机获得一个给定输入电压时所读取的数字值。同样，如果知道了 X 的值，也可以用上面的公式在程序中计算出所供给的电压值。这在需要把实际电压显示在 LED 显示器上的时候会很有用。

隐藏在这个公式中的一个问题是精确度。测量的精确度，或者说能测量到的最小增量计算如下：

$$V_{resolution} = \frac{V_{fullscale}}{2^n - 1}$$

对一个 8 位的，输入量程为 5V 的转换器来说，它的精度计算如下：

$V_{resolution} = 5V / (2^8 - 1) = 5V / 255 = 20mV$(近似)

那么，能够测量得到的最小的电压增量将会是 20mV。如果用这个转换器来测量小于 20mV 的值(例如 5mV)是不恰当的。

2.9.2　模数转换器外设

AVR 微控制器的 ADC 外设精度可达到 10 位，转换速度可达到 15kSPS(Kilo-Samples per Second，千采样每秒)。它能从微控制器上 8 个不同的输入引脚上读取电压信号，因此它能够读取 8 个不同的模拟信号。

控制 ADC 的寄存器有两个：ADC 控制与状态寄存器(ADC Control and Status Register，ADCSR)控制 ADC 的运行；ADC 多路复用选择器(ADC Multiple Select Register，ADMUX)，控制 8 个可能测量的输入。图 2-26 和表 2-14 给出了 ADC 控制与状态寄存器上各位的定义。

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
ADEN	ADSC	ADFR	ADIF	ADIE	ADPS2	ADPS1	ADPS0

图 2-26　ADCSR 的位定义

当 ADC 以最大精度操作时，要求使用 50Hz～200Hz 之间的时钟频率。当然在更高的频率下也能工作，但会降低精度。ADC 的时钟是从系统时钟中通过分频得到的，与计时器的方式相似。ADCSR 的最低三位控制分频的比例因子。这三位是必须要设置的，这样系统时钟才能够通过分频比例因子的分频为 ADC 提供 50Hz～200kHz 之间的时钟信号。表 2-15 所示为选择位和分频比例因子的关系。

表 2-14　ADCSR 的位定义

位	说　明
ADEN	ADC 支持位，设置该位为 1 则允许 ADC
ADSC	ADC 开始转换位，设置该位为 1 则开始一次转换
ADFR	ADC 自由模式选择位，设置该位为 1 则允许自由模式
ADIF	ADC 中断标志位，这由硬件在转换结束时设置
ADIE	ADC 中断屏蔽位，设置该位为 1 则允许在每个转换结束时产生中断
ADPS2	ADC 比例因子选择位
ADPS1	
ADPS0	

表 2-15　ADC 的分频比例因子

ADPS2	ADPS1	ADPS0	分 频 系 数
0	0	0	2
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

在确定分频比例因子时，虽然可以通过不断实验和修正，但是最直接的方法是用系统时钟除以 200kHZ，然后选择紧接着的稍大一点的比例因子。这样就能确保 ADC 的时钟频率仅次于 200kHz。

和串行 UART 一样，ADC 的速度比处理器慢。如果处理器在 ADC 转换模拟数据时处于等待状态，会浪费宝贵的时间。所以，ADC 通常采用中断驱动模式。

虽然下面讨论的是最常用的中断驱动模式，但是 ADC 也可以在自由转换模式下操作，在该模式下，ADC 可以以最快的速度持续地进行转换。当在自由转换模式下读取 ADC 的输出时，必须先关掉中断和停止自由转换，然后读取结果，再打开中断和自由转换模式。这是为了确保读数的准确，避免了程序在处理器更新 ADC 结果寄存器的时候读取数据。

ADC 的初始化步骤通常如下：

(1) 设置 ADCSR 的最低三位，确定分频比例因子

(2) 设置 ADIE 为高电平，打开中断模式

(3) 设置 ADEN 为高电平，使 ADC 有效

(4) 设置 ADSC，以立即开始转换

如果分频比例因子为 8，下面的代码演示了如何初始化 ADC 以读取引脚 ADC2 上的模拟电

压信号：

```
ADMUX＝2 ；          // read analog voltage on ADC2
ADCSR＝0xcb ；       // ADC on ,interrupt mode , /8 , & started
```

上面的初始化设置并启动 ADC，然后立即开始第一次转换。这是非常有用的，因为在 ADC 启动时设置 ADC 需要较长的时间，从而导致第 1 个转换周期要用很长一段时间。在这一个周期里，主程序会进行其余的初始化，ADC 中断会在刚设置好全局中断支持位时立即发生。注意，ADMUX 会被读取的 ADC 信道加载。

图 2-27 所示为一个具有代表性的限制检测系统硬件图，它是基于 ADC 的 3 号通道的模拟输入电压的。简要的说一下该系统的功能：当输入电压超过 3V 的时候，系统点亮红色的 LED；当输入电压低于 2V 的时候，点亮黄色的 LED；当输入在 2V～3V 之间时，点亮绿色的 LED。

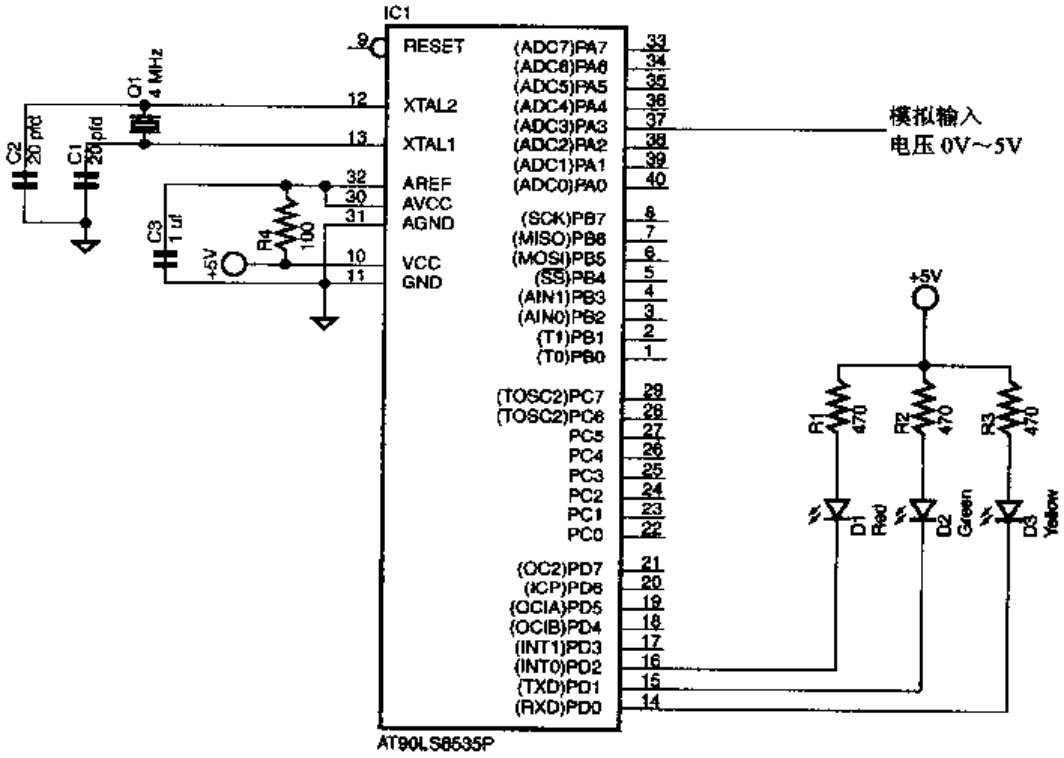


图 2-27　ADC 例子的硬件图

下面的限制检测系统程序是一个典型的 ADC 应用(与图 2-27 对应)。在 main()中通过把 ADCSR 设置为 0xCE 来初始化和启动 ADC。通过把 ADMUX 设为 3 而选择 3 号通道。这样就在第 1 个转换周期结束时，产生 ADC 中断。在 ADC 中断服务程序中检查 ADC 的输出，以确定应该点亮那种颜色的 LED，并点亮相应的 LED。

ADC 10 位的输出是从 ADCW 中读取的。ADCW 是 Codevision AVR 编译器为了能够一次从两个 ADC 结果寄存器(ADCL、ADCH)中读取数据而提供的一个特殊寄存器。其他编译器可能要求程序员在程序中分别读出两个寄存器的内容(以正确的顺序)，再把它们组合成为一个 10 位的结果。

同时请注意 ISR 中使用了模数转换公式，编译器会计算它们的结果，并创建一个常量来代替它们真正用于程序中。您会发现这种技术可以应用在很多场合，例如设置 UART 的 UBRR。

AVR 微控制器的 ADC 外设会根据所使用的具体微控制器而有所不同。所有 ADC 都对 Vcc 连接中的引脚电压有噪声抑制要求(参见图 2-27 或图 2-29)。有些甚至拥有内置的噪声消除器，有些则拥有在内部控制 Vref 的能力。使用 ADC 时，需要查阅相关微控制器的说明书。

```
#include <90s8535.h>

//Define output port and light types
#define LEDs PORTD
#define red 0b011
#define green 0b101
#define yellow 0b011

interrupt [ADC_INT] void adc_isr(void)     //ADC ISR
{
  unsigned int adc_data;        //variable for ADC results

  adc_data = ADCW;     //read all 10 bits into variable

  if (adc_data > (3*1023)/5)
          LEDs = red;                 //too high (>3v)
  else if (adc_data < (2*1023)/5)
          LEDs =yellow;                 //too low (<2v)
  else
    LEDs = green;     //must be just right – like Goldilock's porridge

  ADCSR = ADCSR | 0x40;               //start the next conversion
}

void main(void)
{
   DDRD =0x07;         //least significant 3 bits for output

   ADMUX = 0x3;     //select to read only channel 3
   ADCSR=0xCE;     //ADC on,   /64, interrupt unmasked, and started

   #asm("sei")        //global interrupt enable bit

   while (1)
      ;                //do nothing but wait on ADC interrupt
}
```

2.9.3　模拟比较器

模拟比较器是用来比较两个模拟输入的设备。AIN0 是模拟比较器正输入端；AIN1 是模拟比较器负输入端。如果 AIN0>AIN1，则会设置模拟比较器输出位(Analog Comparator Output Bit，

ACO)，即将该位设置为 1。当 ACO 状态发生改变(正方向、负方向改变，或者两者同时改变)，如果输入比较中断没有被屏蔽，则会产生中断，否则 Timer 1 计时器/计数器产生一次输入捕捉。

图 2-28、表 2-16 和表 2-17 显示了模拟比较器的控制与状态寄存器(ACSR)，它用来控制模拟比较器。

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
ACD	—	ACO	ACI	ACIE	ACIC	ACIS1	ACIS0

图 2-28　ACSR 的位

表 2-16　ACSR 的位定义

位	说　明
ACD	模拟比较禁用位，设置为 1 则禁用模拟比较器
ACO	模拟比较器输出位
ACI	模拟比较器中断标志位
ACIE	模拟比较器中断屏蔽位
ACIC	模拟比较器输入捕捉位 设置为 1 则允许在比较器状态变化时支持输入捕捉
ACIS1 ACIS0	模拟转换比较器模式选择位 (参看表 2-17 的定义)

表 2-17　ACSR 位的中断模式

ACIS1	ACIS0	中断模式
0	0	由 ACO 触发的比较器中断
0	1	保留，没有使用
1	0	ACO 下降沿信号触发的比较器中断 (AIN1 变得比 AIN0 高)
1	1	ACO 上升沿信号触发的比较器中断 (AIN0 变得比 AIN1 高)

图 2-29 所示为一个模拟比较器功能的应用例子。该系统以电池供电，所以知道何时电池的电压过低是很重要的。

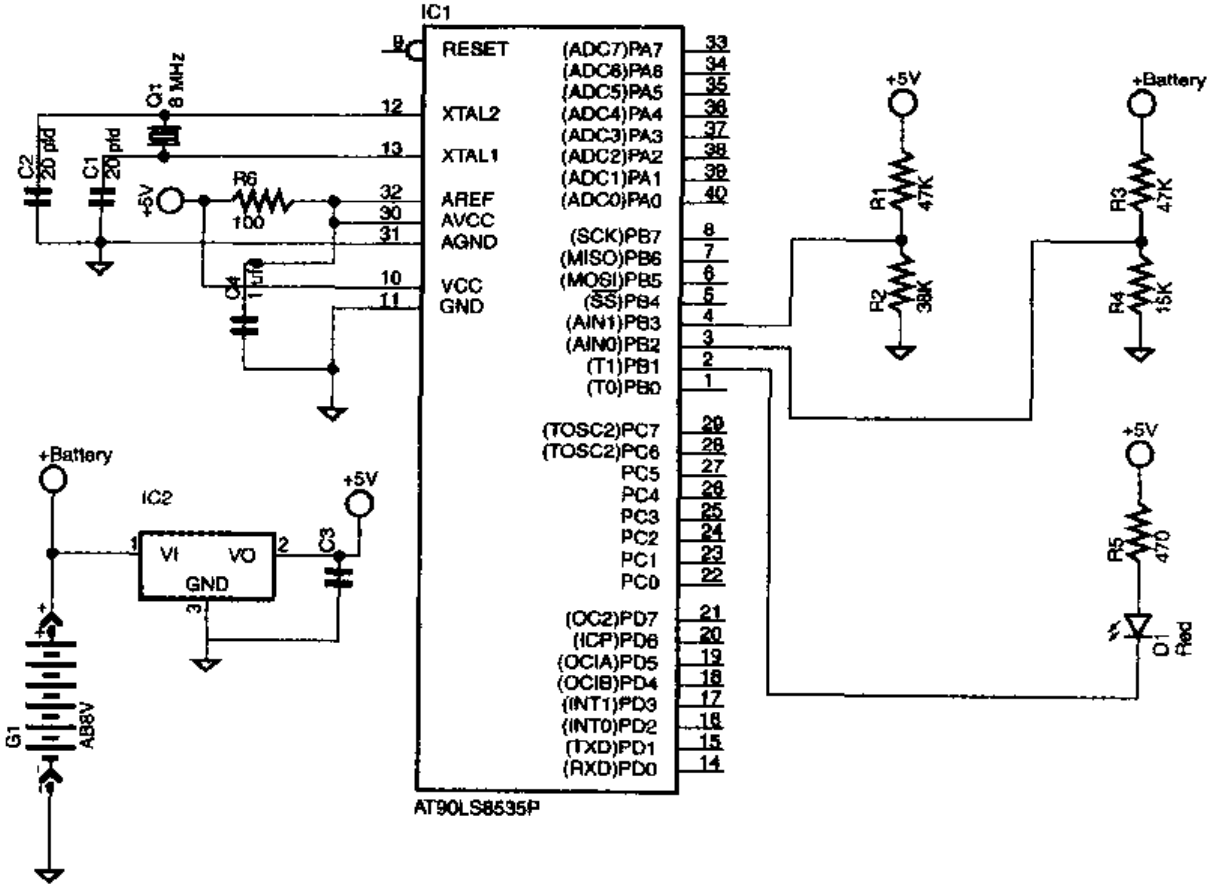


图 2-29　模拟比较器的硬件图

图 2-29 中的系统不断地监视着电池电压，而不需要任何的处理器时间，因为它完全由模拟比较器控制。当电池的电压过低时，LED 就会发亮。

注意：

这个电路有设计上的缺陷。当电池电压变低时，点亮 LED，并让它持续亮着，会加快电池的损耗，所以 LED 应该由额外的电流供电。虽然这样，它还是演示了模拟比较器的用法。

两个模拟转换器的输入端分别连接到两个分压器上：一个是由稳定的 5V 电压供电，作为参考电压；另一个直接由系统电池供电。由 5V 电压供电的分压器提供的电压在中心点约为 2.2V；而由电池直接供电的分压器在电池电压降到 6V 时，提供的电压也是 2.2V。模拟比较器利用由参考电压提供的2.2V 电压就可以检测由电池供电的分压器上的电压何时低于既定电压值。

下面的程序使用模拟比较器是相对简单的。在该示例中，ACSR 被 0x0A 加载，这样就使模拟比较器及其中断有效，同时还把该中断的触发方式设置为下降沿触发(这时 AIN0 下降到 AIN1 之下，即 AIN1 大于 AIN0)。

```
#include <90s8535.h>

// Analog Comparator interrupt service routine
interrupt [ANA_COMP] void ana_comp_isr(void)
{
```

```
    PORTB.1 = 0; //light the LED
}

void main(void)
{

  PORTB=0x02;    //start with LED off
  DDRB=0x02;     //set bit 1 for output

  ACSR=0x0A;     //enable analog comp, AC interrupt, falling edge

  #asm("sei")   //set global interrupt enable bit

  while (1)
      ;      //do nothing
}
```

第 2 章示例项目：D 部分

这是赛车数据采集系统示例项目的第四部分。这部分将会涉及到引擎温度的检测和如何向 PC 中传送收集的数据。

1. 利用模数转换器(ADC)测量引擎温度

以前的调查显示温度信号是连接到 ADC3——Port A(引脚 3)上的 ADC 输入。在查阅了 RTD 热电偶的说明书和调试现有的热电偶电路后，发现该热电偶的测量范围正是老板想要的 100°F～250°F。利用 ADC 10 位的测量模式会使测量结果的值表示如下：

$100°F = 0x000 = 0_{10}$

$250°F = 0x3FF = 1023_{10}$

这使转换的计算公式为：

Temp＝(150°F × ADC 读数)/ 1023＋100°F

在这个测量中，ADC 在自由运行模式下会比较合适。在这种方式下，温度值会尽可能快地更新。这样每次保存的数据都是最新的。在自由运行模式下，每次转换结束时，都会产生一个 ADC 中断，这个中断可以用来更新当前的温度。

初始化 ADC 的一个步骤是通过选择适当的分频比例因子而为 ADC 时钟提供一个合适的时钟频率。ADC 时钟必须是在 50kHz～200kHz 之间。由于系统的时钟是 8MHz，选用 64 的比例因子，则可以得到一个 125kHz 的 ADC 时钟。

ADC 的初始化过程包括选择要测量的通道，打开自由运行模式，开始第一次转换。这样当第一次转换结束以后，就会产生中断，可以保证程序的执行。

```
// ADC initialization
ADMUX=0x3;          // select channel 3 and AREF pin as the reference voltage input
ADCSR =0xE9;        //enable ADC , free run ,  started , clock prescaler of 64
```

ADC 的 ISR 的任务是读取当前的转换结果，并把它转换成温度值。

```
//ADC interrupt service routine
interrupt [ADC_INT] void adc_isr(void)
{
    unsigned int ADC_DATA;
    ADC_DATA = ADCW;      //get data from ADC result register
    current_temp = ((long)150 * (long)ADC_DATA)/ (long) 1023 + (long)100;
```

2. 将采集到的数据发送到 PC 中

采集的数据已经转变为 rpm 或者°F 的形式了，下面就是要把数据发送到 PC 上。这涉及到初始化 UART 和使用内嵌功能去格式化及发送数据。

下面的代码把 UART 初始化为 9600 波特率，8 位，无奇偶检验：

```
// UART initialization
UCSRA=0x00;
UCSRB=0x18;      // enable transmitter and receiver
UBRR=0x33;       //select 9600 baud
UBRRHI=0x00;
```

发送到 PC 的数据将会用 Microsoft Excl 电子表格处理程序来分析。调查显示，数据可以直接用逗号分隔的格式输入。因此，在电子表格的每一行中用逗号来分隔数据，而用回车符来分开每一行。换句话说，数据将按以下的格式发送：

```
data1，data2，data3
data4，data5，data6
```

它们在电子表格中显示出来的格式与上面相似，是两个单元格高，3 个单元格宽的表格。本例中，您决定用表格的第一列放置引擎的 rpm 数据，第二列放置传动轴的 rpm 数据，第三列放置引擎的温度数据。电子表格的每一行将会容纳一组数据，所以每一行的数据都相隔 1 秒。以下的一段代码是 mian()函数中的 while(1)循环：

```
while (1)
{
    if (!PINA.0)    //note: switch must be released before data is all sent
    {
        unsigned char x;          //temporar counter variable

        //print column titiles into the spreadsheet in the first row
        printf ("%s, %s, %s,   \n", "Engine RPM", "Shaft RPM", "Temperature");
        for (x = 0; x < 120; x++)
```

```
            {
              // print one set of data into one line on the spreadsheet
              printf ("%d, %d, %d \n", e_rpm[x], s_rpm[x], temp[x]);
            }
        }
    };
```

这段程序利用内嵌函数 printf()发送 120 组数据(和一行标题)。每组数据都包含了换行符('\n')，使下一组数据会在电子表格的下一行中显示。

2.10　利用 SPI 进行串行通信

AVR 微控制器中另外一种可用的串行通信的形式是串行外设接口(Serial Peripheral Interface，SPI 读 spy 的音)。它是一个同步的串行通信总线，因此在 SPI 通信中的发送器和接收器必须使用相同的时钟，使接收器能同步检测数据位。通常 SPI 总线(Bus)是用于同一块电路板上的外设或其他微控制器之间的短距离通信，至少它们也应该在同一个硬件模块中。这和 UART 不同，UART 用于长距离通信，例如不同的设备之间，或者微控制器和 PC 之间。SPI 总线是为了在使用最少的微控制器引脚的前提下，提供相对高速的短程通信而开发的。

SPI 通信包括一个主端(Master)和一个从端(Slave)。虽然主端和从端都同时接收或发送数据，但是主端负责提供数据传送的同步时钟。这样主端就可以控制数据传送的速度，也就控制了数据的传送。

图 2-30 所示为 SPI 通信中主端和从端的连接。主端提供时钟和从 MOSI(Master-Out-Slave-In)引脚处移出 8 位数据。这 8 位数据将会由它的 MOSI 线移入从端，每个时钟脉冲一位。当 8 位数据从主端移出并移入从端时，也有 8 位数据从从端的 MISO(Master-In-Slave-Out)引脚移出，并进入主端的 MISO 引脚。实际上，SPI 通信就是一个环形的数据传输，8 位数据由主端流向从端，同时也有另一组 8 位数据由从端流向主端。这样，主端和从端可以在一次通信中交换数据。

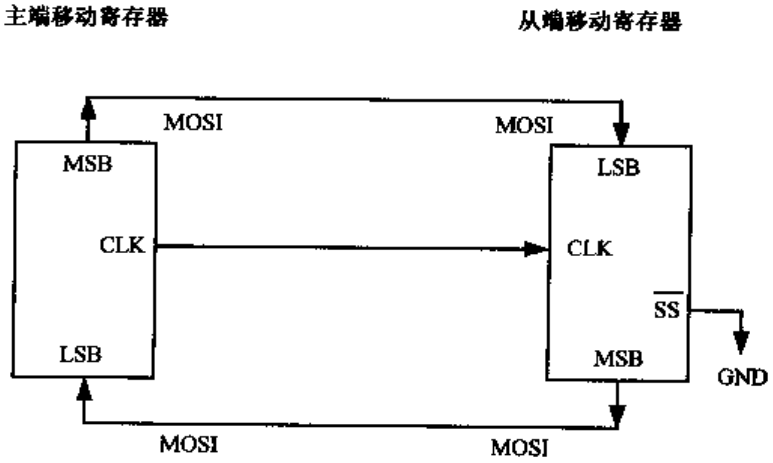


图 2-30　SPI 通信连接方式

利用 SPI 总线把多个不同的设备连接起来是完全有可能的，因为所有的 MOSI 和 MISO 引

脚都可以连接在一起的。网络上的任何一个设备只要想要发送数据都可以成为主端。在网络上的设备的从端选择(Slave Select，SS)引脚接地以后，它就会变成一个从端。通常从端的 SS 引脚都是连接到主端的并行接口，或者用于确定哪个设备将作为从端的解码器上。

SPI 控制寄存器(SPI Control Register，SPCR)控制 SPI 接口的操作。图 2-31、表 2-18 和表 2-19 所示为 SPCR 的位定义。SPCR 和以前学习过的控制寄存器相似，有些位用来使 SPI 接口及其相关的中断有效，有些位用来控制通信的速度。还有一位是在该设备做主端时才设置的，这一位就是 MSTR 位。

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0

图 2-31　SPCR 的位

表 2-18　SPCR 的位定义

位	说　明
SPIE	SPI 中断屏蔽位
SPE	SPI 允许位，设置位 1 则允许 SPI 操作
DORD	数据顺序位。清除则使 MSB 的数据首先发送
MSTR	主/从选择位，设置为 1 则该设备为主端
CPOL	时钟极性位
CPHA	时钟相位位
SPR1	SPI 时钟频率位
SPR0	(参看表 2-19)

表 2-19　SPCR 位的时钟频率

SPR1	SPR0	时钟频率
0	0	系统时钟/4
0	1	系统时钟/16
1	0	系统时钟/64
1	1	系统时钟/128

另外，还有些位用于控制数据顺序，是 MSB 与 LSB 哪个在先；也有用于选择时钟极性和相位的位。这些位是在和其他设备进行 SPI 通信匹配时使用的。

图 2-32 所示为一个具有代表性的简单的 SPI 演示系统的硬件。这个系统从端口 D 读入数据，然后通过 SPI 总线发送出去。数据发送时，SPI 接收数据并显示在端口 C 上。这个 SPI 演示系统仅仅是一个演示。由于 MOSI 和 MISO 引脚是相互连接的，所以无论发送什么，收到的也是一样的。这没有什么实用意义。但是两个这样的系统可以相互交换数据，只是其中一个要改成

从端，其他程序几乎是一样的。这将会组成一个简单的高速网。

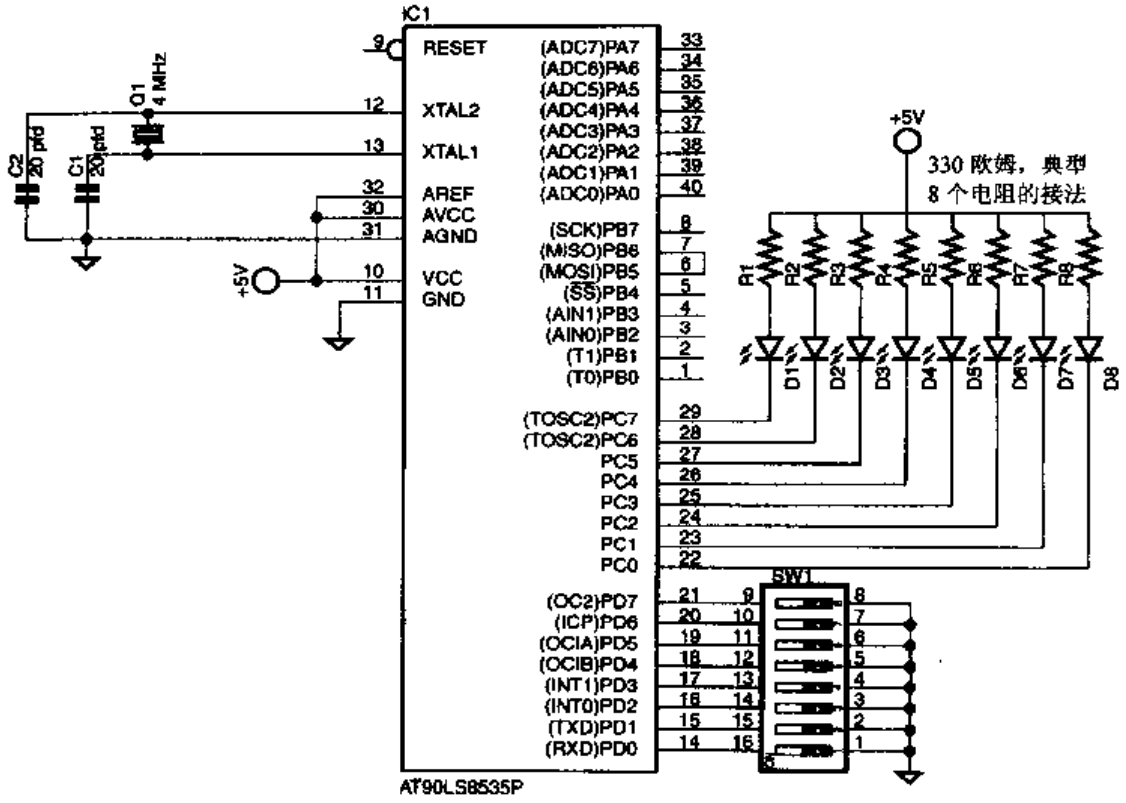


图 2-32　SPI 演示系统的硬件图

下面的程序代码是图 2-32 所示系统的相应软件。这个 SPI 演示软件相当简单。初始化端口 B，把 MOSI、SCLK 和 SS 引脚作为输出，同时 MISO 作为输入引脚，并打开了上拉电阻。端口 C 和端口 D 分别作为相应的输出和输入。这样该 SPI 系统及其相关联的中断就有效了，并被设置为主端。

紧跟在初始化的后面是由 CodeVision AVR 自动插入的一段汇编代码，以保证 SPI 中断是空闲的。这是非常重要的一步，以确保 SPI 中断服务程序在您想要的时候才被执行。最后，把 0x00 写到 SPDR 中，启动 SPI 通信程序。

一旦启动了，整个过程就由 SPI 的 ISR 来处理了。ISR 读取接收到的数据，然后将它们写到输出端，即 LED 处。然后它再读取输入端口，即 DIP 开关，并通过把数据从输入端写入 SPDR 来开始新的通信循环。这样的循环不停地重复着。

SPI 总线通常用来在同一个设备中建立高速网络。连接方式可以是并行的，把所有 MOSI 引脚连在一起，同样所有 MISO 引脚也连在一起；或者是一个大的串行环，所有的数据在环中的设备游走，就相当于一个很大的移位寄存器。并行方式要求有一个选择从端设备的方法，并且只有两个设备可以共享数据流；而在大的串行环中，所有的数据都流经全部设备。在任何一种方式下，基于 SPI 的网络上都可以有多个 CPU 和从端共享数据。

```
#include <90s8535.h>

#define SPI_output_data PORTC
#define SPI_input_data PIND
```

```
// SPI interrupt service routine
interrupt [SPI_STC] void spi_isr(void)
{
  SPI_output_data = SPDR; //read out new data received
  SPDR = SPI_input_data;   //load new data to start SPI transmission
}

void main(void)
{
   PORTB=0x40;    //pull up on MISO
   DDRB=0xB0;     //sclk, MOST,  SS = output
   DDRC=0xFF;     //all output
   PORTD=0xFF;    //pull ups for dip switch

   SPCR=0xD0;     //enable SPI, and its interrupt

   // Provided by CodeVisionAVR to clear the SPI interrput flag
   #asm
    in   r30, spsr
    in   r30, spdr
   #endasm

   #asm("sei") // Global enable interrupts

   SPDR = 0x00; //start SPI communication
   while (1)
      ;
}
```

2.11　AVR RISC 汇编语言指令集

C 语言是一种高级语言。高级语言就是易于人类阅读和能够用一行代码实现复杂操作的编程语言。另一方面，微控制器实际上只能执行机器代码。机器语言只是由数字组成，并且是一种非常低级的语言。

汇编语言是非常接近机器语言的，它可以非常直接地翻译为机器代码。由于汇编语言很接近机器语言，所以它也被认为是一种低级语言。

C 语言编译器首先把 C 语言代码编译成汇编代码，然后再把汇编代码汇编成机器代码。编译器通过使用以汇编函数实现 C 语言语句的函数库来实现第一步。编译器对照着 C 语言语句，在库中选择实现这些语句的库函数，再把它编译到汇编语言程序中。最后，C 语言编译器利用一个汇编器把汇编代码转换成机器代码。

实际的编译和汇编过程还隐藏了两个步骤：连接(Linking)和定位(Locating)。连接过程就是

把库函数和其他代码融合到正在编译的程序中。定位过程就是为每一个可执行指令分配实际的代码存储器地址。

CodeVision AVR 还增加了一步，把可执行的机器代码转换成 Intel 格式(Intel-formatted)的文本文件，以便可以被芯片编程器下载到 AVR 微控制器中。

下面的代码利用 2.9.2 小节的 ADC 示例中的一行代码演示了以上的过程。除了两点以外，该程序已经把整个过程勾勒得比较清晰了。第一点是，程序的存储器空间是 16 位宽的，在 step 3 中的每个字地址对应一个字指令(请注意每个给出的指令都是放在一个字中，这很适合 RISC 处理器)。但是这些字程序在 Intel 十六进制代码中却分开成一个一个的字节，同时地址值也变为了原来的两倍(2×0x52＝0xA4)。为了帮助理解这行代码，图 2-33 给出了 Intel 十六进制代码的格式。

第二点和 step2、step3 不同的是这些字节的出现顺序不一样。在 step3 中的字节是高位在左边，和平时书写数字的习惯一样，但是在 Intel 十六进制文件中却是高位在右边，这是因为在存储器中高位被存于低位之上。

Step1：C 语言代码：

```
if (adc_data >(3*1023)/5)   LEDs = red;   //too high (>3V)
```

Step2：编译：

编译器利用自带的库把 C 语言代码编译成为以下的汇编代码(注释是作者为了清晰另外加上去的)：

```
LDI   R30, LOW(614)          ;put the low byte of contant into R30
LDI   R31, HIGH(614)         ;put the high byte of the constant into R31
CP    R30, R16               ;compare the low byte with adc_data low
                             ;byte (must be stored in R16)
CPC   R31, R17               ;now compare the high bytes (with carry)
BRSH _0x2                    ;branch if untrue to the 'else' statement
LDI   R30, LOW(3)            ;if true, load the value to light the red
                             ;LED into R30
OUT   0x15, R30              ;and output the value to the port d I/O
                             ; register
```

Step3：汇编

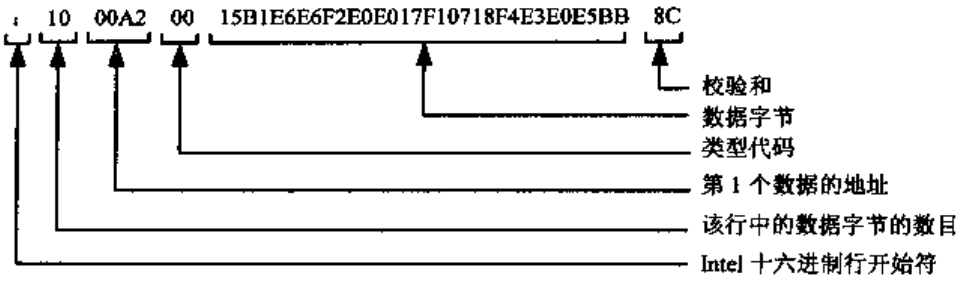
中间的那一列就是汇编器生成的机器代码。左手边的那一列是放置机器代码的代码存储器地址，右手边的汇编指令只是为了使程序更清楚，而不是汇编器产生的。

```
000052 e6e6        LDI    R30, LOW(614)
000053 e0f2        LDI    R31, HIGH(614)
000054 17e0        CP     R30, R16
000055 07f1        CPC    R31, R17
000056 f418        BRSH   _0x2
000057 e0e3        LDI    R30, LOW(3)
000058 bbe5        OUT    0x15, R30
```

Step4：为下载而转换成 16 进制的文件：

这行 Intel 十六进制代码包含了上面的机器代码(加下划线的部分)：

：1000A20015B1E6E6F2E0E017F10718F4E3E0E5BB8C



注意：在图中加上空格是为了使图更清楚

图 2-33　Intel 十六进制代码格式

Intel 十六进制格式要求每行代码都以:开始，用以标志一行代码的开始。紧跟着的一个字节指示该行代码包含的字节数(记住这个数是一个 16 进制的数字，所以 10 实际上就是表示该行有 16 个字节)。下两个字节是第 1 个数据字节的地址，后面的数据地址依次放置。再下一位就是类型代码(Type Code)，在很多较老较小的处理器中用于指示代码的性质。在 AVR 系列控制器中，类型代码是用来指示一个 64KB 的存储器分页，0 表示第 0 页，1 表示第 1 页，依此类推。在程序的内存空间要求超过 16 位时，类型代码就必不可少了。

最后是适当数目的数据字节，包括一个用于检验错误的校验和。

就上面提到的情况而言，让人觉得汇编语言是比较难用的，特别是和 C 语言那样的高级语言比起来，显得既晦涩又抽象。您可能觉得很奇怪，为什么还有人会使用汇编语言。在程序员需要直接操作微处理器的寄存器或者要求严格控制程序执行速度时，汇编语言就变得非常有用了。前者的例子如编写实时操作系统等高级程序，或者其他需要人工直接操作微控制器存储器的程序。同样，它也是一个有用的调试工具。在分析汇编代码以确定处理器的实际动作时，C 语言的错误通常都会暴露出来。

使用汇编语言的第 2 个原因就是利用它来紧密地控制执行时间。C 语言编译器使用一个函数库来编译 C 语言。根据编译器完整性的不同，有些只是尝试利用有限的函数来涵盖所有的情况，有的则拥有很多函数，可以覆盖所有的情况，而且编译后的代码都非常有效率。如果只有几个覆盖所有情况的函数，为了能覆盖所有可能的情况，这些函数就会有更多的机会使用更多的执行时间。这一点对于优秀的 C 语言编译器(如 CodeVision AVR)而言，它们有大型的函数库，足以有效地覆盖最多的情况。在这种情况下，利用汇编语言来编程的需要就降低了。

是不是说用汇编语言来编程的需要就会增加了呢？一种折中的方法是：先用 C 语言编写函数，然后利用编译器生成的汇编文件分析编译器产生的汇编代码。您就可以通过检查函数中的每个指令来分析汇编代码(参看附录 G 中的指令)，并确认是否有些指令是多余的。根据经验显示，即使是优化过的代码，也可以通过认真分析其汇编代码来提速。

2.12　本章小结

本章为选择和运用 Atmel 生产的 AVR RISC 系列微控制器提供了必须的基本硬件知识。

具体地说，本章介绍了微控制器的结构，包括它们的存储器系统设计和内部工作原理；也

讨论了每种常用的外设，包括它们的用法，以及演示如何使用它们的示例。即使遇到在本章中没有专门讨论过的设备，您也应该知道如何使用它们，因为它们的工作方式都是非常相似的。

到现在为止，您应该掌握本章所列出来的每一个目标了。

第 2 章示例项目：E 部分

这是第 2 章示例项目的总结，它的各部分贯穿了本章，以下是完整的程序代码：

```
#include <mega163.h>
#include <stdio.h>

unsigned char data_set_cntr;   //counter to track number of data sets recorded
unsigned int e_rpm[120], s_rpm[120];   //arrays to hold 120 sets of data
unsigned char temp[120];
unsigned int current_e_rpm, current_s_rpm;     //variables to hold current values
unsigned char current_temp;

unsigned int previous_shaft_capture_time;   //saved time from previous capture

// External Interrupt 0 service routine
interrupt [EXT_INT0] void ext_int0_isr(void)
{
     unsigned int current_capture_time, period;     //current time and period

   current_capture_time = TCNT1;   //get Timer1 time
   //check for roll-over
   if (current_capture_time > previous_shaft_capture_time)
         period = current_capture_time - previous_shaft_capture_time;
   else
         period = 0xFFFF - current_capture_time + previous_shaft_capture_time;
   current_s_rpm = (unsigned long)60E6 / (unsigned long)period;
   previous_shaft_capture_time = current_capture_time;   //save for next calculation

}

// External Interrupt 1 service routine
interrupt [EXT_INT1] void ext_int1_isr(void)
{
   data_set_cntr = 0;       //clear counter to start counting
}

int time_cntr = 0;     //global variable for number of Timer 0 overflows

// Timer 0 overflow interrupt service routine
interrupt [TIM0_OVF] void timer0_ovf_isr(void)
{
```

```
    TCNT0 = 6;      //reload timer 0 for next period
    if  (time_cntr++ == 500) //check for one second, increment counter
    {
        if (data_set_cntr < 120)   //record data if less than 120 sets
        {
            e_rpm[data_set_cntr] = current_e_rpm;     //record engine rpm
            s_rpm[data_set_cntr] = current_s_rpm;     //resocrd shaft rpm
            temp[data_set_cntr++] = current_temp;     //record engine temp
        }
        time_cntr = 0;        //reset counter for next 1 second interval
    }
}

unsigned int previous_capture_time;   //saved time from previous capture

// Timer 1 input capture interrupt service routine
interrupt [TIM1_CAPT] void timer1_capt_isr(void)
{
    unsigned int current_capture_time, period;     //current time and period

    current_capture_time = (256* ICR1H) + ICR1L;   //get captured time
    //check for roll-over
    if (current_capture_time > previous_capture_time)
        period = current_capture_time - previous_capture_time;
    else
        period = 0xFFFF - current_capture_time + previous_capture_time;
        current_e_rpm = (unsigned long)120E6 / (unsigned long)period;
        previous_capture_time = current_capture_time;   //save for next calculation
}

// ADC interrupt service routine
interrupt [ADC_INT] void adc_isr(void)
{
    unsigned int ADC_DATA;

    ADC_DATA = ADCW;     //get data from ADC result register
    current_temp = ((long)150 * (long)ADC_DATA)/ (long) 1023 + (long)100;
}

void main(void)
{
    // Input/Output Ports initialization
    PORTA=0x01;         //enable pull up on bit 0 for upload switch
    DDRA=0x00;          //all input
    PORTB=0x00;
```

```
DDRB=0x00;          //all input
PORTC=0x00;
DDRC=0x00;          //all input
PORTD=0x4C;         //enable pull ups on Port D, Bits 2,3, & 6
DDRD=0x00;          //all input

// Timer/Counter 0 initialization
TCCR0=0x03;         //clock set to clk/64
TCNT0=0x00;         //start timer0 at 0

// Timer/Counter 1 initialization
TCCR1A=0x00;
TCCR1B=0x02;          //prescaler = 8, capture on falling edge
TCNT1H=0x00;          //start at 0
TCNT1L=0x00;

// External Interrupt(s) initialization
GIMSK=0xC0;    // both external interrupts enabled
MCUCR=0x0A;    // set both for falling edge triggered
GIFR=0xC0;     // clear the interrupt flags in case they are errantly set

// Timer(s)/Counter(s) Interrupt(s) initialization
TIMSK=0x21;    // timer0 overflow, ICP interrupts enabled

// UART initialization
UCSRA=0x00;
UCSRB=0x18;      // enable transmitter and receiver
UBRR=0x33;       //select 9600 baud
UBRRHI=0x00;


// Analog Comparator disabled
ACSR=0x80;
SFIOR=0x00;

// ADC initialization
ADMUX=0x3;   //select channel 3 and AREF pin as the reference voltage input
ADCSR=0xE9; //enable ADC, free run, started, clock prescaler of 64

// Global enable interrupts
#asm("sei")

while (1)
{
        if (!PINA.0)   //note: switch must be released before data is all sent
```

```
        {
            unsigned char x;   //temporary counter variable

            //print column titles in the first spreadsheet row
            printf ("%s , %s , %s \n","Engine RPM", "Shaft RPM", "Temperature");
            for (x = 0; x < 120; x++)
            {
                // print one set of data into one line on the spreadsheet
                printf ("%d , %d , %d \n",e_rpm[x], s_rpm[x], temp[x]);
            }
        }
    }
}
```

总之，这个程序只是该任务的一种解决方案，可能还有其他或者更好的方案。同时，该系统还有很多可以提高系统性能的地方。以下列出一些最有可能进行改善或提高的地方：

- 保存 rpm 和温度的原始数据，而不转换为 rpm 或温度。这样，在测量每个参数时就可以节省时间。原始数据可以在向 PC 发送数据时才进行转换。
- 测量 rpm 时，实际上是通过计算 1 秒内的脉冲数，然后乘以 60 得到的。这样，那个间隔为一秒的程序在记录数据以后，应该清零，以方便下一次的计数。
- 该系统没有提供任何防范措施来防止在记录数据后和向 PC 传送数据前触发启动开关。这样是会导致数据丢失的。其中一个方案是：在向 PC 传送数据前，不允许记录数据。这个方案的问题是，如果驾驶员不经意地单击了 Start 按钮，并开始记录数据，就不可能恢复和传送正确的数据。另一个方案是，可以提供一个“数据锁”触发器开关。当这个开关拨向某一边时，无论何时单击 Start 按钮，都可以记录数据；当该开关拨向另一边时，禁用进一步的数据记录，直到把数据保存到 PC 中。

还有别的需要改善和提高的地方，完成了本章的学习后，您自己应该能够增加几处了。

2.13 练习

1. 解释微控制器和微型计算机的区别(2.3 节)。

*2. 介绍下列类型的存储器，并描述它们的用法(2.4 节)：

A. FLASH 代码存储器(FLASH Code Memory)

B. 数据存储器

C. EEPROM

3. 解释为什么有些数据存储器被称为 I/O(2.4 节)。

4. 解释处理器堆栈的功能和目的(2.4 节)。

5. 解释当中断发生时，处理器如何找到相应的中断服务程序并执行它(2.5 节)。

*6. 编写一段 C 语言程序初始化外部中断 1，使外部中断引脚出现下降沿信号时触发中断(2.5 节)。

7. 什么是 Watchdog 计时器，如何使用(2.5 节)?

*8. 编写一段 C 语言代码初始化端口 C 引脚，使它的高四位作为输入，而低四位在作为输出(2.6 节)。

9. 编写一段 C 语言代码初始化 Port A 和 Port B，使 Port A 的所有位和 Port B 的最低位作为输入，而 Port B 的其余位作为输出(2.6 节)。

10. 至少列出计时器/计数器的两个不同功能，并说明(2.7 节)。

*11. 当 VART 以 9600 波特的速率发送字符 H 时，画出其输出端的波形图(2.8 节)。除了波形外，该图还应该显示正确的电压和每位的宽度。

12. 画出在 1200 波特率的速度下发送“F”时 UART 的输出波形(2.8 节)。除了波形外，该图还应该显示正确的电压和位宽度。

*13. 计算以下模数转换表(表 2-20)中缺少的数据(2.9 节)：

表 2-20　模数转换表

V_{in}	$V_{fullscale}$	Digital Out	#of bits
4.2V	10V	8	
1.6V	5V	10	
5V	123	10	
10V	223	8	

14. 描述 UART 和 SPI 串行通信之间的区别(2.8 节和 2.10 节)。

15. 详细说明编译过程的每一个步骤(2.11 节)。

题号前标有*的题目，在附录中给出了该题的全部或者部分答案。

2.14　上机实习

1. 创建一个控制 LED 亮灭的程序，当外部中断 0 出现下降沿信号时点亮 LED，当外部中断 1 出现上升沿信号时熄灭 LED。

2. 创建一个程序，演示当程序陷入无限循环时，Watchdog 计时器如何复位处理器。

3. 创建一个程序，读取 Port A 的所有 8 位的数据，然后将高四位的数据同低四位的数据进行交换，再输出到 Port B。

4. 创建一个模拟引擎的监视器，当发动机速度(是一个 TTL 电平的方波)低于 2000Hz 时，点亮黄色 LED；当速度超出 4000Hz 时，点亮红色 LED；当速度在两者之间的时候，点亮绿色 LED。

5. 创建一个程序，用 PWM 控制一个小电机的速度(或者 LED 的亮度)，全程 16 步。通过 Port A 的低四位信号来控制速度(或亮度)。

6. 创建一个程序，利用 UART 在 9600 波特的速度下每 50 毫秒输出一个 ASCII 码 G。利用示波器观测微控制器 TXD 线上的 TTL 电平信号和总线发送/接收器上的 RS-232 信号。对每个波形，都要辨别传输信号中的开始位、结束位、发送信号的数据位及波形信号的电压电平。

7. 修改实习 6，使 SPI 总线同时发送字符 G，利用示波器显示传送的 SPI 信号和 SPI 时钟，然后辨别每个数据位。

8. 利用模数转换器制作一个简单的电压表，用于检测 0V～5V 的电压，要求精度为 0.1V。把结果显示在终端或一个端口上，如果是后者，要求该端口的高四位显示电压值的个位，低四位显示电压值的十位。

第3章　标准I/O和预处理函数

3.1　本章目标

在完成本章的学习之后，您应该掌握以下内容：

- 使用标准 I/O 函数从程序中产生数据或提供调试信息
- 重定义基本 I/O 操作，以便标准 I/O 函数能够对通用异步收发器(UART)之外的其他外设进行操作
- 使用标准输入函数将用户数据读入程序
- 会使用标准输出格式来简化程序代码，并获得专业化的界面效果
- 使用#define 声明语句来声明常量和重定义函数
- 使用#include 声明语句将信息从外部文件引入用户程序
- 使用#pragma 声明语句在编译过程中优化或定制用户程序

3.2　引言

标准 C 语言 I/O(输入/输出)函数提供了一种在程序执行过程中发送信息和获取信息的方法。这些函数建立在一些简单的字符输入输出函数的基础之上，这些字符输入输出函数能很容易地满足系统硬件的要求。标准 C 语言中的一些高级 I/O 函数可以格式化数字和文本输出，并能处理和存储输入的数字或文本数据。

这里给出的标准 C 语言 I/O 函数都经过了改写，以便在资源有限的嵌入式微控制器中使用。这些函数的原型可以在 stdio.h 文件中找到。这个文件必须在函数使用前通过#include 预处理指令加入到 C 源代码中。

预处理指令可以用来增加程序的可读性和扩展程序的功能。这些指令包括#define、#include 和#pragma，也包括其他条件性的指令，如#ifdef、#else 和#endif。这些指令可以引导编译器，在程序被实际编译前提供关于程序的信息。预处理指令还被用来使程序更易于移植(能移植到另外一个硬件平台上)。预处理程序允许程序员更加详细地定义和构造程序。

3.3　字符输入/输出函数 getchar()和 putchar()

最低级的输入/输出函数是 getchar()和 putchar()。这两个函数提供从程序中接收单个字符和发送单个字符的方法。它们通常与 UART 或微控制器的串行通信端口相关联。高级的标准 C 语

言 I/O 函数使用这些基本函数作为操作的基础。这使程序员只需改变这两个函数就能十分容易地改变整个程序的数据的输入输出源。这两个函数的标准形式如下：

```
char getchar (void);   //returns a character received
                       //by the UART, using polling.
void putchar(char c);   //transmits the character c using
                        //the UART, using polling.
```

一般在嵌入式系统中使用这些库函数之前，用户必须先完成如下工作：

- 初始化 UART 的波特率
- 使 UART 发送器有效
- 使 UART 接收器有效

例如，在下面的代码中，程序初始化 UART，然后运行一个循环。在循环中程序调用 getchar() 等待接收一个字符。一旦接收到了字符，程序通过调用 putchar()函数回显：

```
#include <90s8515.h>
                /* include processor specific informaiton */
#include <stdio.h>
                /* include the standard C I/O function definitions */

#define xtal 4000000L          /* quartz crystal frequency [Hz] */
#define baud 9600              /* Baud rate */

vod main(void)
{
      char k;

      /* initialize the UART'S baud rate */
      UBRR=xtal/16/baud - 1;

      /*   Initialize the UART control register
           RX & TX enabled, no interrupts,
           8 data bits */
      UCR=0x18;

      while (1)
      {
            k=getchar();       /*receive the character */
            putchar(k);        /* and echo it back */
      };
}
```

所有标准输入/输出函数都使用 getchar()和 putchar()。如果您希望使用其他外设来代替 UART 进行标准输入/输出，就必须修改 getchar()和 putchar()函数。这些函数的源代码一般位于 stdio.h 文件中。

以下是 getchar()和 putchar()的典型版本，用于 AVR 微控制器，这些代码可以在 stdio.h 文件中找到：

```
char getchar (void)
{
    while ((USR & RXC)==0)      // wait for character to enter UART
          ;
    return UDR;                 //return the character to the caller
}

void putchar (char c)
{
      while((USR &UDRE)==0   //wait for transmit register
           ;                 //to be empty..
       UDR = c;              //put c out to the UART
}
```

getchar()和 putchar()函数可以被自定义函数所代替，这些自定义函数使用不同的 UART、串行外设接口(Serial Peripheral Interface，SPI)或并行端口。

在前面的例子中调用 getchar()函数时，会使微控制器一直等待，直到接收到一个字符后程序才会继续运行。在一些应用程序中，往往不希望程序挂起直到一个字符到达。这时可以利用中断在后台处理串行通信，而主程序继续运行。在后台，中断驱动的串行通信允许字符在到来时放置于缓存或队列中。然后，当主程序准备接收字符时，再从缓存中读取字符并处理。

发送字符也是一样。使用中断驱动的传送可以避免等待字符从串行端口发出。当 UART 的发送寄存器为空时，将产生一个中断，允许下一字符从发送缓存中发送。只要缓存中存在字符，这样的过就会不断重复进行。

在下面的例子中，getchar()和 putchar()分别从 Rx_Buffer 和 Tx_Buffer 字符数组中输入和输出字符。其中 getchar()和 putchar()函数及其使用方法如下：

```
#include <90s8515.h>

#define xtal 4000000L      /* quartz crystal frequency [Hz] */
#define baud 9600          /*Baud rate */

//UART Receiver buffer
#define                    RX_BUFFER_SIZE          24
char Rx_Buffer[Rx_BUFFER_SIZE+1];   //character array (buffer)
char RX_Wr_Index; //index of next char to be put into the buffer
char RX_Rd_Index; //index of next char to be fetched from the buffer
char RX_Counter;   //a total count of characters in the buffer
bit   RX_Buffer_Overflow;       //This flag is set on UART Receiver buffer overflow
//UART Transmit buffer
#define                    TX_BUFFER_SIZE       24
char TX_Buffer[Rx_BUFFER_SIZE+1];   //character arry (buffer)
```

```
char TX_Rd_Index; //index of next char to be put into the buffer
char TX _Wr_Index; //index of next char to be fetched from the buffer
char TX _Counter;   //a total count of characters in the buffer
bit   fPrimedIt;       //This flag is used to start the transmit
                       // interrupts, when the buffer is no longer empty

// UART Receiver interrupt service routine
interrupt [UART_RXC] void uart_rx_isr(void)
{
     char c;

   c = UDR;

   Rx_Buffer[RX_Wr_Index] = c;     /* put received char in buffer */

   if(++RX_Wr_Index > RX_BUFFER_SIZE)      /* wrap the pointer */
    RX_Wr_Index = 0;
    if(++RX_Counter > RX_BUFFER_SIZE)      /* keep a character count */
    {                                      /* overflow check..   */
          RX_Counter = RX_BUFFER_SIZE;    /* if too many chars came */
          RX_Buffer_Overflow = 1;  /* in before they could be used */
     }                                    /* that could cause an error!! */
}

//Get a character from the UART Receiver buffer
char getchar(void)
{
  char c;

  while(RX_Counter == 0)                    /* wait for a character... */
             ;

  c = Rx_Buffer [RX_Rd_Index];              /* get one from the buffer..*/

    if (++RX_Rd_Index > RX_BUFFER_SIZE)   /* wrap the pointer */
       Rx_Rd_Index = 0;

  if(RX_Counter)
         RX_Counter - ;                  /* keep a count (buffer size)   */

  return c;
}

//UART Transmitter interrupt service routine
interrupt [UART_TXC] void uart_tx_isr(void)
```

```
{
  if (TX_Counter != 0)
   {
       if (fPrimedIt == 1)
       {                    /* only send a char if one in buffer */
          fPrimedIt = 0;    /* transmision, then don't send the test and wrap the pointer */

          if(++TX_Rd_Index > TX_BUFFER_SIZE)
                TX_Rd_Index = 0;
          TX_Counter - ; /* keep track of the counter */
        }

        if (TX_Counter != 0)
        {
          UDR = TX_Buffer[TX_Rd_Index];
                          /* otherwise, send char out port */

                          /* test and wrap the pointer */
           if (++TX_Rd_Index > TX_BUFFER_SIZE)
             TX_Rd_Index = 0;

         TX_Counter - ;   /* keep track of the counter    */
        }
      }
      UCR |= 0x40;              /* clear TX interrupt flag */
}

//Write a character to the UART Transmitter buffer
void putchar (char c)
{
   char stuffit = 0;

   while (TX_Counter >(TX_BUFFER_SIZE - 1))
                  ;                    /* WAIT!! Buffer is getting full!! */
   IF (TX_Counter == 0) /* if buffer empty, setup for interrupt */
       stuffit = 1;

   TX_Buffer[Tx_Wr_Index++] =c;   /* jam the char in the buffer.. */

   if (TX_Wr_Index > TX_BUFFER_SIZE)          /* wrap the pointer */
       TX_Wr_Index = 0;
                                  /* keep track of buffered chars */

  TX_Counter++;
```

```
    if (stuffit == 1)
    {                           /* do we have to "Prime the pump"? */
            fPrimedIt   = 1 )
            UDR = c;    /* this char starts the TX interrupts.. */
    }
}

//These defines tell the compiler to replace the stadio.h
//version of getchar() and putchar() with ours..
//That way, all the other stdio.h functions can use them!!
#define             _ALTERNATE_GETCHAR_
#define             _ALTERNATE_PUTCHAR_

// now, we include the library and it will understand our replacements

#include   <stdio.h>

void main()
{
   char k;

   // initialize the UART's baud rate
   UBRR = xtal/16/baud - 1
   //          Initialize the UART control register:
   //          RX & TX enabled,
   //          RX & TX interrupts enabled,
   //          8 data bits
   UCR=0xD8;

   // Global interrupt enable
   #asm("sei")

   whiles(1)
   {
            if (RX_Counter)   //are there any received characters??
            {
                    k = getchar();      // get the character
                    putchar (k);        // and echo it back
            }

           // since there is no waiting on getchar or putchar..
           // other things can be done!
    }
}
```

本例中，变量 RX_Counter 表示中断程序已经接收的字符数，这些字符被保存在接收器字符数组 RX_Buffer 中。这可以避免主程序在调用 getchar()函数时空闲等待新字符的到来。在 while 的每一遍循环中都对 RX_Counter 进行检测。如果不为零，则字符存在于 RX_Buffer 字符数组中，调用 getchar()函数就能获得字符而不须等待。

由于 putchar()同样是中断驱动的，它几乎不给程序增加额外的执行时间；putchar()把一个字符放置在发送缓存中，然后启动发送中断。从那一刻开始，每次 UART 发送寄存器为空时，中断程序就从发送缓存中取一个字符移到 UART，直到发送缓存为空。

_ALTERNATE_GETCHAR_和_ALTERNATE_PUTCHAR_用来告诉编译器程序使用重新声明的 getchar()和 putchar()函数(而不是预置在 stdio.h 头文件中的)。

3.4　标准输出函数

标准库的输出函数包括字符串输出函数 puts()；格式打印函数 printf()；字符串格式打印函数 sprintf()等。AVR 微控制器带有有 RAM 和 FLASH 存储器区域，它提供了额外的 FLASH 字符串的函数或函数 putsf()来打印位于 FLASH 存储器区域中的字符串常量。

3.4.1　输出字符串函数 puts()

puts()的标准形式是：

```
void puts(char *str);
```

该函数使用 putchar()输出以空字符结束的字符串 str(该字符串位于 SRAM 中.)，并在后面自动加上一个换行符('\n')。当程序调用 puts()函数时，必须将指向字符串的指针传给函数。下面的例子调用 puts()来输出字符串 Hello，并以一个换行符结尾。

```
#include <90s8515.h>
        /* include processor specific information */
#include <stdio.h>
        /* include the standard C I/O function definitions */


#define xtal 4000000L   /* quartz crystal frequency [Hz] */
#define baud 9600       /* Baud rate */

char s[6]= "Hello";     /* declare a RAM based string and initialize it */

void main (void)
{
        /* initialize the UART's baud rate */
        UBRR= xtal /16/baud - 1;
        /*      Initialize the UART control register
          RX & TX enabled, no interrupts,
```

```
        8 data bits */
      UCR=0x18;

      puts(s);          // prints Hello follow by a newline character
      while (1)
       ;
}
```

在这个程序中，s 是数组 s[]的地址，它作为数组的指针传给了 puts()函数。

3.4.2 从 FLASH 输出字符串函数 PUTSF()

从 FLASH 输出字符串常量函数的标准形式如下：

```
void putsf(char flash *str);
```

该函数使用 putchar()输出以空字符结束的字符串 str(该字符串位于 SRAM 中.)，并在后面自动加上一个换行符('\n')。下面的例子调用 putsf()来输出位于 FLASH 存储器中的字符串 Hello, 并以一个换行符结尾。

```
#include <90s8515.h>
        /* include processor specific information */
#include <stdio.h>
        /* include the standard C I/O function definitions */

#define xtal 4000000L /* quartz crystal frequency [Hz] */
#define baud 9600     /* Baud rate */

flash char s[6]= "Hello";     /* declare a FLASH based string and initialize it */

void main (void)
{
      /* initialize the UART's baud rate */
      UBRR= xtal /16/baud - 1;
       /*    Initialize the UART control register
             RX & TX enabled, no interrupts,
             8 data bits */
       UCR=0x18;

      putsf(s);          // prints Hello follow by a line-feed

      putsf("Hello");    // prints Hello follow by a line-feed

      while (1)
          ;
}
```

3.4.3　格式打印函数 printf()

格式打印函数的标准形式为：

```
void printf(char flash *fmtstr [ , arg1, arg2, ...]);
```

该函数根据字符串常量 fmtstr 中的格式规范，使用 putchar()函数输出格式化文本。格式规范字符串 fmtstr 是一个常量字符串，并且必须位于 FLASH 存储器中。printf()函数将对 fmtstr 进行处理。fmtstr 可以由一些字符常量组成并直接输出，也可以由特殊的格式命令或说明组成。当函数 printf()处理 fmtstr 时，它输出其中的字符，并根据包含在字符串中的格式规范展开每个参数。百分号(%)被用于指示格式规范的开头。每项格式规范联系到一个参数，依次是 arg1, arg2 等。每项格式规范必须对应一个参数，反之亦然。

printf()的格式规范的实现只是标准 C 语言函数的简化版本。它能满足嵌入式系统开发的最低要求。全部实现 ANSI-C 规范需要大量的内存空间，而在嵌入式应用程序中，这些函数在大多数情况下都是无法使用的。

表 3-1 列出了一些在 CodeVisionAVR 库中可以使用的格式规范。

表 3-1　printf()格式规范

规　范	参 数 格 式
%c	将下一项参数以 ASCII 码字符输出
%d	将下一项参数以十进制整数输出
%i	将下一项参数以十进制整数输出
%u	将下一项参数以无符号十进制整数输出
%x	将下一项参数以无符号十六进制整数输出，使用小写字母
%X	将下一项参数以无符号十六进制整数输出，使用大写字母
%s	将下一项参数以空字符结束的字符串输出，位于 SRAM
%%	输出百分号%

使用格式规范时，需要注意一些规则和修饰符：

- 所有的数值输出形式均为右对齐，左边位数不够时加空格
- 如果在%和字母 d，i，u，x 或 X 之间插入字符 0，则在输出数值的左边补 0
- 如果在%和字母 d，i，u，x 或 X 之间插入减号(-)，所有的数值输出形式均为左对齐
- 在%和字母 d，i，u，x 或 X 之间插入数字 1~9 来限定所显示数字的最小位数
- 字符串格式规范用来告诉 printf()函数需要处理的参数个数

下面有一些关于格式规范影响输出的例子。管道标志符(|)通过指明文本的开头结尾及空格的位置来帮助阐明格式规范的影响。它们并不是 printf()函数一定要求的：

```
int i = 1234;
char p[10] = "Hello";

printf("|%d|", i);                    // would print:          |1234|
```

```
printf("|%5d|", i);              // would print:      |  1234|
printf("|% - 5d|", i);           // would print:      |1234  |
printf("|%05d|", i);             // would print:      |01234|
printf("|%x|", i);               // would print:      |4d2|
printf("|%04X|", i);             // would print:      |04D2|
printf("|%s: %04X|", p, i);      // would print:      |Hello:04D2|
```

下面的程序在同样的语句下用 printf()函数打印输出一个字符串和一个整数值。

```
#include <90s8515.h>
        /* include processor specific information */
#include <stdio.h>
        /* include the standard C I/O function definitions */


#define xtal 4000000L   /* quartz crystal frequency [Hz] */
#define baud 9600       /* Baud rate */

void main (void)
{
        int I;

        /* initialize the UART's baud rate */
        UBRR= xtal /16/baud - 1;
         /*    Initialize the UART control register
              RX & TX enabled, no interrupts,
              8 data bits */
         UCR=0x18;

        for (I=0; i<100; I++)
        {     // print a formatted string and tell about I
              printf ("printf example, I = %d\n" , I);
        }
        while (1)
           ;
}
```

运行该程序的结果是文本字符串“printf example, I=”被打印输出了 100 次。每次打印时，数字 0~99 也输出在该次的字符串后面。值得注意的是，fmtstr 包括了所有的文本，%d 表示数字 I 应该显示的位置，同时也有行末结束符'\n'。一个语句就完成了许多工作！

用户也必须意识到使用 printf()所付出的代价。Printf()函数能处理许多工作，因此也需要大量的代码空间来完成。使用 printf()函数能提高程序的灵活性、健壮性，并实现格式化输出，而不需要输入太多的代码。然而，标准库的 printf()函数本身含有几百行的汇编代码，这些代码将自动加入程序中来支持 printf()函数。一旦函数被添加了这些代码，每一次调用函数就只需要增加少量的汇编代码了。

3.4.4　字符串格式打印函数 sprintf()

字符串格式打印的标准形式如下：

```
void sprintf (char *str, char flash *fmtstr [ , arg1, arg2, ...]);
```

该函数的操作和 printf()基本一样，只是格式化文本放置在了以空字符结束的字符串 str 中而不是调用 putchar()函数。举例说明如下：

```
#include <90s8515.h>
        /* include processor specific information */
#include <stdio.h>
        /* include the standard C I/O function definitions */

#define xtal 4000000L /* quartz crystal frequency [Hz] */
#define baud 9600 /* Baud rate */

char s[28]; /* declare a FLASH based string and initialize it */

void main (void)
{
        int I;

        /* initialize the UART's baud rate */
        UBRR= xtal /16/baud - 1;
         /*    Initialize the UART control register
               RX & TX enabled, no interrupts,
               8 data bits */
         UCR=0x18;

        for (I=0; i<100; I++)
        {     // load the string s with the formatted data
              sprintf (s, "sprintf example, I = %dn" , I);
              puts(s);          // then 's'!!
        }
        while (1)
           ;
 }
```

运行该程序的结果是字符串“sprintf example, I=”被打印输出了 100 次。序号 0~99 也输出在该字符串后面。本例中，sprintf()函数格式化输出并将它置于字符数组 s 中，而不是使用 putchar()将它从 UART 中发送出去。Puts()函数使用 putchar()函数将字符串 s 从 UART 中发送出去，并且在字符串的末尾加上换行符('\n')。

3.5 标准输入函数

标准库中的输入函数包括字符串接收函数 gets()；格式扫描函数 scanf()和字符串格式扫描函数 sscanf()函数等。就像最基本的输出函数是 putchar()一样，最基本的输入函数是 getchar()。在任何情况下都是调用 getchar()来获取必要的数据。因此通过改变 getchar()的操作就能很容易地改变数据源。

3.5.1 获得字符串函数 gets()

gets()的标准形式为：

```
char *gets(char *str, unsigned char len);
```

该函数使用 getchar()将字符输入以换行符结束的字符串 str。gets()函数将用空白('\0')替换换行符。字符串的最大长度用 len 表示，如果在 len 个字符的读取过程中没有遇到换行符，则函数将用空字符作为字符串的结尾并结束函数。gets()返回一个指向 str 的指针。

举例说明如下：

```
#include <90s8515.h>
        /* include processor specific information */
#include <stdio.h>
        /* include the standard C I/O function definitions */

#define xtal 4000000L  /* quartz crystal frequency [Hz] */
#define baud 9600      /* Baud rate */

char s[12]; /* declare a RAM based string */

void main (void)
{
        /* initialize the UART's baud rate */
        UBRR= xtal /16/baud - 1;
        /*      Initialize the UART control register
                RX & TX enabled, no interrupts,
                8 data bits */
    UCR=0x18;

     while (1)
     {   gets(s, 10);          //get a string from the standard input

     puts(s);    // prints string 's' follwed by a line feed
     }
}
```

在上面的程序中，字符将从标准输入设备读取，每次 10 个字符，并加上一个换行符后回显至标准输出设备。

3.5.2　格式扫描函数 scanf()

格式扫描函数的标准形式如下：

```
signed char scanf(char flash *fmtstr [ , arg1 address, arg2 address, ...]);
```

该函数根据包含在 fmtstr 字符串中的格式规范，使用 getchar()执行格式化的文本输入。格式规范字符串 fmtstr 是一个常量，必须位于 FLASH 存储器中。

和 printf()函数一样，这里 fmtstr 由 scanf()函数处理；fmtstr 既可以由输入流中的字符常量组成，也可以由特殊的格式命令或说明组成。当函数 scanf ()处理 fmtstr 时，它使用 getchar()函数输入数据，然后将这些数据与 fmtstr 中的字符进行匹配，或者根据字符串 fmtstr 中内嵌的格式规范为每个参数转换字符。百分号(%)用于指示格式规范的开头。每项格式规范被关联到一个参数，依次是 arg1, arg2 等。每项格式规范必须对应一个参数，反之亦然。

参数 arg1，arg2 等是变量的地址或指针(在函数中不使用此类地址或指针常常会导致非常糟糕的结果！)函数将返回成功的输入数，或者在出错时返回 - 1。

scanf()格式规范的实现只是标准 C 语言函数的简化版本。它能满足嵌入式系统开发的最低要求。全部实现 ANSI-C 规范需要大量的内存空间，在嵌入式应用程序中，这些函数在大多数情况下都是无法使用的。

表 3-2 列出了可用的格式规范。

表 3-2　scanf()格式规范

规　范	参 数 格 式
%c	将下一项参数以 ASCII 码字符输入
%d	将下一项参数以十进制整数输入
%I	将下一项参数以十进制整数输入
%u	将下一项参数以无符号十进制整数输入
%x	将下一项参数以无符号十六进制整数输入
%s	将下一项参数以空字符结束的字符串输入

使用扫描格式规范时，需要注意一些规则：

- 空格字符及制表符 tab 都将被忽略
- 普通字符(除了%)必须匹配下一个输入的非空字符

下面的例子提醒用户输入(用逗号分隔)，然后调用 scanf()等待输入并将输入存入变量中，最后，调用 printf()函数格式化输出数字到用户终端：

```
#include <90s8515.h>
        /* include processor specific information */
#include <stdio.h>
```

```
            /* include the standard C I/O.function definitions */

#define xtal 4000000L   /* quartz crystal frequency [Hz] */
#define baud 9600       /* Baud rate */

void main (void)
{           int i,j
            /* initialize the UART's baud rate */
            UBRR= xtal /16/baud - 1;
           /*       Initialize the UART control register
                     RX & TX enabled, no interrupts,
                     8 data bits */
             UCR=0x18;

            putsf ("Enter two integers 'i, j':");
            scanf ("%d, %\n", &i, &j);
                                // get two integers, one decimal
                                // and one hexadecimal with a comma
                                // separating them

            printf ("%d, %0xX", i, j);   // print the values and stop..


            while (1)
                  ;
}
```

3.5.3 字符串格式扫描函数 sscanf()

字符串格式扫描的标准形式为：

```
signed char sscanf(char *str, char flash *fmtstr [ , arg1 address, arg2 address, ...]);
```

该函数的操作和 scanf()基本一样，只是格式化文本是从空字符结束的字符串 str 中读取，str 位于 SRAM 中。

修改前面的例子，使用 sscanf()来从字符串中而不是从标准输入中获取数据(UART 使用 getchar())。修改后程序如下：

```
#include <90s8515.h>
            /* include processor specific information */
#include <stdio.h>
            /* include the standard C I/O function definitions */

#define xtal 4000000L   /* quartz crystal frequency [Hz] */
#define baud 9600       /* Baud rate */
```

```
/* declare a RAM based string and initialize it */
char s[14] = {"1234,abcd\n"};

void main (void)
{
        int i, j

        /* initialize the UART's baud rate */
        UBRR= xtal /16/baud - 1;
        /*      Initialize the UART control register
           RX & TX enabled, no interrupts,
           8 data bits */
        UCR=0x18;

                    //get the values from the RAM string 's'
        sscanf (s, "%d, %x\n", &i, &j);
                    // get two integers, one decimal
                    // and one hexadecimal with a comma
                    // separating them

        printf ("%d, %0xX", i, j);   // print the values and stop..

        while (1)
              ;
}
```

3.6　预处理指令

预处理指令事实上并不是 C 语言语法的一部分，但是由于它的用途和较为人熟悉，因此仍然被使用。预处理是实际编译程序中独立的一步，它在编译开始前进行。最普遍的指令是#define 和#include。预处理指令允许用户进行以下操作：

- 包括其他文件中的文本，如包含库和用户函数原型的头文件
- 定义宏，减少编程工作量，提高源代码的可读性
- 设置条件编译以进行调试，并提高程序可移植性
- 发出明确编译指令来生成和优化编译

3.6.1　#include 指令

#include 指令可以用来将其他文件包含在自己的源代码中。只要需要且编译器允许，可以包含任意数目的文件。include 还允许嵌套(即在包含的文件中还可以包含其他非递归的#include)。通常，嵌套有深度的限制，这种限制根据编译器的不同而不同。在 CodeVisionAVR C 编译器中，最多允许 16 层文件的嵌套。语句的标准形式如下：

```
#include <file_name>
```

或

```
#include "file_name"
```

小于(<)和大于(>)分隔符表明括弧内的文件是标准库的一部分或库文件集。编译器通常会在\inc 目录中查找文件。当文件名被引号分隔时，编译器将首先查找被编译的 C 文件的当前目录。如果找不到文件，编译器再在默认的库文件目录(\inc)下查找。

#include 语句可以位于代码的任何位置，但它通常设置在程序模块的顶部，以提高程序的可读性。

3.6.2 #define 指令

#define 指令将被看作一种“文本替换”，把它称作宏。#define 的标准形式如下：

```
#define     NAME    Replacement_Text
```

通常#define 指令被用来声明常量，但标记 NAME 和 Replacement_Text 也可以带参数。在编译过程中，只要预处理程序检测到 NAME 标记，就会用 Replacement_Text 来代替(展开)它。如果 NAME 含有参数，程序中的实际参数将代替标记在 Replacement_Text 文本里的形式参数。

有几个简单规则应用于宏定义和#define 指令中：

- 解释#define 行时应常常使用注释符号(/*...*/)
- 记住句子行末就是#define 的末尾，左边的文本将取代右边所有的文本

例如下列的宏定义：

```
#define ALPHA 0xff      /* mask lower bits */
#define SUM(a, b) a+b
```

下面的表达式：

```
int x = 0x3def & ALPHA;
```

将被以下的表达式取代：

```
int x = 0x3def & 0xff   /* mask lower bits */;
```

下面的表达式：

```
int i = SUM(2,3);
```

将被以下的表达式取代：

```
int i = 2+3 ;
```

#define 指令的多样性和能力在某种程度上依赖于编译器预处理程序的复杂性。大多数现代的编译器能够重定义函数和操作，也可以解析出它们的参数(如上面的例子)。CodeVisionAVR 编译器在这方面功能很强大。较小的编译器可能只允许最简单的操作，例如定义常量和字符串。

使用#define 指令为函数取别名或为函数创建备用调用都是十分强大而且方便的用法。例如，库中的一个现有函数可以显示如下：

```
int _write_SCII(int c)   // this function sends a char to
{                                // an alternate comm..port
    ...
   return c;
 }
```

给这个函数取一个别名：

```
#define putchar2(c) _write_SCII(c)
```

这样就可以用两种方式来引用这个函数：

```
putchar2(' b');
```

或

```
_write_SCII(' b');
```

这种方法对于从其他程序中重用源代码，或者在一个大型开发中给合多个库都十分有用。不需要对每一次的函数调用或参考进行定位和重命名，只需要简单地指定函数的别名，就能用多种方式来使用这个函数。

#define 的这种功能也可以用来对函数进行“二次声明”以加强程序的可读性。例如，假定一个名为 set_row_col()的函数用 row(行)和 column(列)两个参数在一个 4 行每行 20 字符的液晶显示器(以下称 LCD)中设置光标：

```
void set_row_col(char row, char colum)
{
 ...
}
```

当使用宏进行二次声明时，LCD 将以 x 和 y 坐标显示位置，而不是以行和列显示：

```
#define   goto_xy(x, y)    set_row_col(y, x)
```

另外一种可能是二次声明可以使用字符位置来设置光标：

```
#define   set_char_pos(p)    set_row_col((p/20), (p%20))
```

用光标的位置除以列数，就能计算出行数；而位置对宽度取摸所得余数，就是列数。这将被看作一种“奇特的文本替换”函数运行。预处理过程在程序运行前生效，因此编译器只看到#define 右边的内容。这些参数的改变使编者和读者都能更加容易读懂程序。

这种替换处理有时能使控制更精细。定义宏时，可以使用#操作符将宏参数转换为字符串。例如：

```
#define print_MESSAGE(t)   printf(#t)
```

将用

```
printf("Hello")
```

代替表达式

```
printf_MESSAGE(Hello);
```

也可以使用##操作符来连接两个参数。下面的例子中，形式参数 a 和 b 将被连接。在替换中左边的 ALPHA(a,b) 将被右边的 ab 代替。程序使用了实参数 x 和 y。宏定义如下：

```
#define alpha(a, b)   a ## b
char alpha(x, y) =1;
```

结果为：

```
char xy = 1;
```

通常，替换文本使用标记名定义在一行之内。也可以通过在一行的结尾使用反斜线符号(\)另起一行扩展定义：

```
#define MESSAGE "This is a very \
long text ..."
```

反斜线字符不会在表达式中出现，换行符也会被清除。作为结果的“This is a very long text..”将被视为一个单独的字符串，不会被割断。

宏也可以使用#undef 指令解除定义。#undef 主要在标记名称用作标志的条件编译中使用：

```
#undef ALPHA
```

#undef 指令允许重新定义已经定义过的宏。例如，如果一个宏被定义成

```
#define A_CHAR 'A'
```

可以重新将其定义为小写的 a：

```
#undef A_CHAR
#define A_CHAR 'a'
```

3.6.3　#ifdef，#ifndef，#else 和#endif 指令

#ifdef，#ifndef，#else 和#endif 指令可以用于条件编译中。语法是：

```
#ifdef macro_name
    [set of statements 1]
 #else
    [set of statements 2]
#endif
```

如果 macro_name 是定义的宏名字，那么#ifdef 表达式如果为真则编译 set_of_statements_1，

否则编译 set_of_statements_2。else 和 set_of_statements_2 是可选的。另外一种形式是：

```
#ifndef   macro_name
        [set of statements]
#endif
```

如果 macro_name 没有定义，则#ifdef 表达式为真。其他语法规则和#ifdef 相同。#if，#elif，#else 和#endif 可以用于条件编译中。

```
#if expression1
            [set of statements 1]
#elif expresson2
            [set of statements 2]
#else
            [set of statements 3]
#endif
```

如果 expression1 为真，则编译 set_of_statements_1。

如果 expression2 为真，则编译 set_of_statements_2。

否则编译 set_of_statements_3。#else 和 set_of_statements_3 是可选的。

在创建需要许多配置运行的程序时，条件编译非常有用。同一源代码可以应用于许多操作设置中，这使维护程序(或运行程序的产品)十分容易。

另一方面，条件编译更普遍的用法是用于调试。条件编译和标准 I/O 库函数的结合能帮助您改善程序，并能在不需要在线仿真器和示波器等设备的情况下，加速程序的运行。

第 1 章示例中的状态机用来控制“虚拟的”交通灯，可以修改这个示例来增加一些调试功能。使用条件编译，调试功能可以按需要随时打开和关闭，从而使用户能在提供信息和增加性能这两方面进行选择。代码如下所示。

```
/* Warnings are disabled */
#pragma warn -

/* Write some code here */

/* Warnings are enabled */
#pragma warn+

#include <90s8515.h>
#include <delay.h>

//#define    DEBUGGING_ON

// Removing the comment will cause the DEBUGGING_ON "flag" to
// be enabled. This will cause the compiler to include the
// additional library and lines of code required allowing the
// programmer to watch the state machine run from the UART.
```

```
#ifdef  DEBUGGING_ON
        // include the standard C I/O function definitions
    #include <stdio.h>

    #define xtal 4000000L  /* quartz crystal frequency [Hz] */
    #define baud 9600          /* Baud rate */
#endif

#define    EW_RED_LITE PORTB.0  /* definitions to actual outputs */
#define    EW_YEL_LITE PORTB.1  /* used to control the lights */
#define    EW_GRN_LITE PORTB.2
#define    NS_RED_LITE  PORTB.3
#define    NS_YEL_LITE  PORTB.4
#define    NS_GRN_LITE  PORTB.5

#define    PED_XING_EW  PINA.0   /* pedestrian crossing push button */
#define    PED_XING_NS PINA.1   /* pedestrian crossing push button */
#define    FOUR_WAY_STOP PINA.3      /* switch input for 4-Way Stop */

char time_left;          // time in seconds spent in each state
int current_state;       // current state of the lights
char flash_toggle;       // toggle used for FLASHER state

// This enumeration creates a simple way to add states to the machine by name.
// Enumerations generate an integer value for each name automatically,
// making the code easier to maintain.

enum { EW_MOVING , EW_WARNING , NS_MOVING , NS_WARNING , FLASHER };

// The actual state machine is here..
void Do_States(void)
{
    switch(current_state)
    {
      case  EW_MOVING:        // east-west has the green!!
            EW_GRN_LITE = 1;
            NS_GRN_LITE = 0;
            NS_RED_LITE = 1;   // north-south has the red!!
            EW_RED_LITE = 0;
            EW_YEL_LITE = 0;
            NS_YEL_LITE = 0;

            if(PED_XING_EW || FOUR_WAY_STOP)
            {     // pedestrian wishes to cross, or
```

```
            // a 4-way stop is required
            if(time_left > 10)
                time_left = 10;   // shorten the time
        }
        if(time_left != 0)        // count down the time
        {
            - time_left;
            return;                    // return to main
        }                   // time expired, so..
        time_left = 5;     // give 5 seconds to WARNING
        current_state = EW_WARNING;
                            // time expired, move
        break;              // to the next state

    case EW_WARNING:
        EW_GRN_LITE = 0;
        NS_GRN_LITE = 0;
        NS_RED_LITE = 1;   // north-south has the red..
        EW_RED_LITE = 0;
        EW_YEL_LITE = 1;   // and east-west has the yellow
        NS_YEL_LITE = 0;

        if(time_left != 0)        // count down the time
        {
            - time_left;
            return;                    // return to main
        }                   // time expired, so..
        if(FOUR_WAY_STOP)      // if 4-way requested then start
            current_state = FLASHER; // the flasher
        else
        {                           // otherwise..
            time_left = 30;   // give 30 seconds to MOVING
            current_state = NS_MOVING;
        }                           // time expired, move
        break;                      // to the next state

    case NS_MOVING:
        EW_GRN_LITE = 0;
        NS_GRN_LITE = 1;
        NS_RED_LITE = 0;   // north-south has the green!!
        EW_RED_LITE = 1;   // east-west has the red!!
        EW_YEL_LITE = 0;
        NS_YEL_LITE = 0;
        if(PED_XING_NS || FOUR_WAY_STOP)
        {     // if a pedestrian wishes to cross, or
```

```
            // a 4-way stop is required..
            if(time_left > 10)
                time_left = 10;   // shorten the time
        }
        if(time_left != 0)        // count down the time
        {
            - time_left;
            return;                    // return to main
        }                  // time expired, so..
        time_left = 5;     // give 5 seconds to WARNING
        current_state = NS_WARNING;  // time expired, move
        break;                       // to the next state

    case NS_WARNING:
        EW_GRN_LITE = 0;
        NS_GRN_LITE = 0;
        NS_RED_LITE = 0;   // north-south has the yellow..
        EW_RED_LITE = 1;
        EW_YEL_LITE = 0;   // and east-west has the red..
        NS_YEL_LITE = 1;

        if(time_left != 0)        // count down the time
        {
            - time_left;
            return;                    // return to main
        }                  // time expired, so..
        if(FOUR_WAY_STOP)      // if 4-way requested then start
            current_state = FLASHER; // the flasher
        else
        {                          // otherwise..
            time_left = 30;   // give 30 seconds to MOVING
            current_state = EW_MOVING;
        }                          // time expired, move
        break;                     // to the next state

    case FLASHER:
        EW_GRN_LITE = 0;  // all yellow and
        NS_GRN_LITE = 0;  // green lites off
        EW_YEL_LITE = 0;
        NS_YEL_LITE = 0;

        flash_toggle ^= 1;       // toggle LSB..
        if(flash_toggle & 1)
        {
            NS_RED_LITE = 1;   // blink red lights
```

```
                EW_RED_LITE= 0;
            }
            else
            {
                NS_RED_LITE = 0;   // alternately
                EW_RED_LITE= 1;
            }
            if(!FOUR_WAY_STOP)     // if no longer a 4-way stop
                current_state = EW_WARNING;
            break;                 // then return to normal

        default:
            current_state = NS_WARNING;
            break;       // set any unknown state to a good one!!

    }
}

void main(void)
{
    DDRB = 0xFF;  // portb all out
    DDRA = 0x00;  // porta all in

    #ifdef DEBUGGING_ON
        //    Initialize the UART control register
        //    RX & TX enabled, no interrupts,
        //    8 data bits
        UCR=0x18;
        // initialize the UART's baud rate
        UBRR=xtal/16/baud - 1;
    #endif

    current_state = NS_WARNING;  // initialize to a good starting
                                 // state (as safe as possible)
    while(1)
    {
        delay_ms(1000); // 1 second delay.. this time could
                        // be used for other needed processes

        Do_States();     // call the state machine, it knows
                         // where it is and what to do next

    #ifdef DEBUGGING_ON
            // send state and time data to serial port
        printf("Current State = %d :", current_state);
```

```
            printf("Time Left = %d\r", time_left);
        #endif
        }
    }
```

3.6.4 #pragma 指令

#pragma 指令允许特定编译器指令或开关；#pragma 语句依赖于特定的编译器，因此，当使用这些控制时，应当参考编译器的用户手册。下面对此类控制命令的描述是针对 CodeVisionAVR 编译器的，但其大多数特征在其他编译器中也可找到。

1. #pragma warn

#pragma warn 指令用来启用或禁止编译器警告。对于那些不喜欢警告的程序员可以使用它来关闭警告功能。警告的内容可以包括 variable declared but never referenced 或者 possible loss of precision 等。不应该永久关闭警告，这将导致失去一些有用的警告信息，使用户不知道为什么程序老是产生错误的结果。最好的方法是声明、编码、强制转换直到所有的警告都已经消除。

2. #pragma opt

#pragma opt 指令用来启用或关闭编译器代码优化程序。

优化程序可以改善编译器产生的汇编语言输出。这可以通过减小汇编输出的大小，调整代码使其更快运行，或者两者同时使用来完成。#pragma opt 指令必须放置于源文件的开端，并且默认设置优化功能已启用：

```
/*Turn optimization off, for testing purposes */
#pragma opt -
```

或者

```
/*Turn optimization on */
#pragma opt+
```

3. #pragma optsize

优化程序的代码大小将会导致程序运行稍慢。这是因为普通代码都转换成了子程序，子程序调用代替通代码来运行程序。而更多地使用了内嵌函数(in-line)的程序由于没有从子程序调用和返回的额外开销，运行将更快一些。

如果启用了代码优化，可以通过使用#pragma optsize 指令在大小和运行速度方面优化全部或部分程序。

```
/* The program will be optimized for minimum size */
 #pragma optsize+

/* Place your program functions here */
```

或者

```
/* Now the program will be optimized for maximum execution speed */
#pragma optisize -

 /* Place your program functions here */
```

默认的状态由 CodeVisionAVR 环境中的 Project | Configure | C Compiler | Compilation | Optimization 命令设置。

4. #pragma savereg

#pragma savereg 指令可以启用或关闭在中断寄存器的保存和恢复功能。这些寄存器有 R0，R1，R22，R23，R24，R25，R26，R27，R30，R31 和 SREG。该控制命令用来手动地减少在中断服务程序中由于保存和恢复机器状态而增加的额外负载。这类命令对于初学者比较危险，因此，当需要使用此类“专家级”的控制命令时，必须先仔细研究编译器如何产生代码：

```
/* Turn registers saving off */
#pragma savereg -

/* interrupt handler */
interrupt [1] void my_irq(void) {
/* now save only the registers that are affected by the routines in the handler,
for example R30, R31 and SREG */
#asm
    push r30
    push r31
    in    r30, SREG
    push r30
#endasm

/* place the C code here */

/* now restore SREG, R31 and R30 */
#ASM
    pop r30
    out SREG, R30
    pop r31
    pop r30
#endasm
}
/* re-enable register saving for the other interrupts */
#pragma savereg+
```

默认状态是 savereg+，在中断过程中自动保存寄存器内容。这类设置没有菜单选项。当不再需要关闭此功能时，确保程序的设置是 savereg+，以防止不稳定的操作使得寄存器程序得不

到保存。

5. #pragma regalloc

CodeVisionAVR 编译器将使用寄存器变量来尽可能提高运行性能。例如，在使用在线仿真器(In-Circuit-Emulator，ICE)进行调试时，寄存器变量将很难跟踪，因此可能需要指示编译器停止寄存器自动分配功能。

使用#pragma regalloc 指令可以启用或者关闭全局变量的寄存器自动分配功能：

```
/* the follwing global variable will be automatically allocated to a register */
#pragma regallloc+
unsigned char alpha;

/* the follwing global variable will not be automatically
allocated to a register and will be placed in normal SRAM */
#pragma regalloc -
unsigned char beta;
```

默认状态由 CodeVisionAVR 环境中的 Project | Configure | C Compiler | Compilation | Automatic Register Allocation 命令设置。

6. #pragma promotechar

在 C 语言 ANSI 标准中，字符被看作一个 16 位的值。这在一个小型的嵌入式系统中是不必要(或不想要)的。CodeVisionAVR 编译器的默认设置是把一个字符看作 8 位字符型的值，但如果应用程序要求将字符看作整数，使用 promotechar 就能实施 ANSI 标准。

使用#pragma promotechar 指令可以启用或关闭 ANSI 字符到整数操作数的提升(或转换)功能：

```
/* turn on the ANSI char to int promotion */
#pragma promotechar+
```

或者

```
/* turn off the ANSI char to int promotion */
#pragma promotechar -
```

这个选项可以在 CodeVisionAVR 环境中的 Project | Configure | C Compiler | Promote char to int 菜单中设置。

7. #pragma uchar

当声明字符型变量时，默认的格式如下：

```
char var ; // declares a signed character variable
unsigned char var2 ; // declares an unsigned character variable
```

#pragma uchar 指令允许改变默认的字符格式。如果启用了 uchar 指令，将产生下面的效果：

```
signed char var ;     // this is now an signed character variable
char var2 ;           // and this is an UNSIGNED character variable
```

使用#pragma uchar 指令可以启用或者关闭 uchar 功能：

```
/* char will be unsigned by default */
#pragma uchar+
```

或者

```
/* char will be signed by default */
#pragma uchar -
```

该选项可以由 CodeVisionAVR 环境中的 Project | Configure | C Compiler | char is unsigned 命令设置。

8. #pragma library

您可以创建自己的函数库，并像使用标准 I/O 库一样使用它。一旦您的函数库创建完成，使用#pragma library 指令就能在编译时将它包含到程序中：

```
#pragma library mylib.lib
```

库的创建是就特定的编译器而言的，可以参考编译器的用户手册说明按要求的方法进行创建。

3.6.5　其他宏和指令

表 3-3 列出了一些预先定义的宏。这些标记是特定于 CodeVisionAVR 的，但也可以在其他编译器中找到。它们可用于从已编译的程序中获取信息，如编译的日期和时间。它们也可以用于基于内存模型或优化设置的条件编译中。

表 3-3　CodeVisionAVR 预定义宏

宏	描　述
LINE	编译文件的当前行数
FILE	当前的编译文件
TIME	以 hh:mm:ss 格式显示的当前时间
DATE	以 mm dd yyyy 格式显示的当前日期
_MODEL_TINY_	设置是否编译使用微存储模式
_MODEL_SMALL_	设置是否编译使用小存储模式
_OPTIMIZE_SIZE_	设置是否用优化程序大小编译
_OPTIMIZE_SPEED_	设置是否用优化运行速度编译
_UNSIGNED_CHAR_	设置是否打开编译选项或使用#pragma uchar+
_GLOBAL_DEFINES_	设置是否全局#define 编译选项打开

#error 指令可以用来终止编译过程并且显示错误消息。

语法为：

```
#error error_message
```

例如：

```
#error This is an error
```

3.7 本章小结

本章主要提供了关于使用标准库函数输出信息、获取用户输入及调试程序的必要知识。您也可以学到如何修改基本函数 getchar()和 putchar()，以便标准库能够用于各种硬件配置。

您已经学习了如何使用编译指令#include 和#define，以及如何使用它们来提高程序的可重用性和可读性。也学习了使用#ifdef、#ifndef、#else 和#endif 进行条件编辑。本章还提供了将单一源代码应用于不同系统配置中和通过增加临时代码来进行调试的方法。

本章还介绍了#pragma 语句，它可以使代码在编译时变得通用和优化，并使程序获得最佳重用性和性能。

3.8 练习

*1. 用#define 指令编写一个宏，为函数 putchar()生成一个别名为 send_the_char()的函数(3.6 节)。

2. 编写一个函数，提示用户输入两个整数值，将它们相加，用 The values A + B = C 的格式输出它们的值和相加的和，其中 A、B 和 C 是十进制整数(3.4 节)。

3. 使用 scanf()编写一个函数，以 MM/DD/YYYY 格式接收数据，然后使用 printf()函数以 DD-MM-YY 格式输出(3.5 节)。

*4. 编写一个函数，打印输出它的编译日期和时间。使用编译器内部标记名获得日期和时间值(3.6 节)。

5. 编写一个函数，打印输出它的编译内存模式和优化程序设置。使用编译器内部标记名获得日期和时间值(3.6 节)

*6. 编写一个函数，输入 16 位的十六进制数，然后打印输出相应的二进制数。注意：没有标准的输出函数打印输出二进制数。

7. 使用#define 声明一个 8 位的并行 I/O 端口 PORTX，端口地址为 0x1200。声明必须使该端口能够通过 PORTX = 34 或者 i = PORTX 进行访问。

在练习序号前的'*'表示该问题已经在附录 H 中有全部或部分答案了。

3.9　上机实习

1. 编写一个程序，每 50 毫秒通过 UART 输出一个 ASCII 字符 G。使用示波器观察在微控制器 TXD 线路上的 TTL 电平信号和媒体驱动器输出的 RS-232 信号。在每一个波形中，辨别出传输信号的起始位、终止位和数据位。

2. 修改实习 1 的程序，使 G 也可以从 SPI 总线中发送。使用示波器显示发送信号和时钟，并指出数据位。

3. 编写一个程序，使用 getchar()获得 一个串行字符(从 PC 中)并且使用 putchar()返回一个字符，使得每个返回的字符在字母表中的顺序都比接收的要大 2。

4. 修改实习 1 的程序，改变 putchar()函数，使返回的 ASCII 字符代码

第4章 CodeVisionAVR C编译器和集成开发环境

4.1 本章目标

通过本章的学习，您应该掌握以下内容：

- 操作 CodeVisionAVR C 编译器与集成开发环境
- 正确使用编译器选项
- 应用 CodeVisionAVR 环境对一个目标设备编程
- 应用 CodeVisionAVR 代码向导自动生成代码外壳
- 应用 CodeVisionAVR 终端工具发送和接收 RS-232 通信
- 使用 Atmel 的 AVR Studio 作为软件仿真器完成基本的调试操作

4.2 引言

本章的目的是使您熟悉 CodeVisionAVR C 编译器。

CodeVisionAVR C 编译器只是众多可用于 Atmel AVR 微控制器的编译器之一。由于它的外部开发环境比较好，编译器性能比较高，因此我们决定在本书中使用它。不像一般的编译器是为 AVR 指令集而修改的，CodeVisionAVR 是专门为 AVR 微控制器设计的。因此说它产生的代码非常严密，使用了 AVR 微控制器的很多特性而没有浪费。与其他编译器相比，它为 Atmel AVR 微控制器产生的代码量很小、效率很高。CodeVisionAVR 还有一个 CodeWizardAVR 代码产生器，这是一个在开发大多数项目时都很有用的工具。

本章提供 CodeVisionAVR 开发环境的一个概览及其很多特性。您会学到如何创建一个项目，如何为项目设置具体的编译器选项，如何将一个项目生成一个输出文件并将输出文件加载到目标设备上。另外，本章还将介绍 CodeWizardAVR 代码产生器和终端工具。最后，还将介绍作为软件仿真器的 AVR Studio 的一些基本特点。

目前 CodeVisionAVR 可在 Windows 98、Windows 2000 及 Windows NT 平台上运行。

4.3　集成开发(IDE)环境操作

CodeVisionAVR C 编译器通过其集成开发环境(Integrated Development Environment，IDE)代码来使用。IDE 允许用户新建项目、向项目中添加源代码文件、为项目设置编译器选项等；另外还可将项目编译生成可执行程序文件。这些可执行文件然后被加载到目标微处理器上。本章的目标是使您熟悉 CodeVisionAVR IDE 的一般用法。而 IDE 的很多高级特性不在本书的讨论范围之内。

在讨论 CodeVisionAVR IDE 时，将以下面的方式表示命令：主菜单后跟一个竖线（|）和子菜单名。例如 File 菜单的 Save As 命令表示为 File | Save As。CodeVisionAVR IDE 的主菜单栏位于应用程序标题栏的下面。

4.3.1　项目

一个项目是一组文件和编译器设置的集合，可以用它来建立一个具体的程序。项目文件的扩展名为.prj。

图 4-1 是一个已经打开的项目。屏幕的左下方是文件导航器，这里列出了属于该项目的文件以及不属于该项目但是该项目用到的其他文件。只有在项目名下面列出的文件才会作为该项目的一部分被编译。在图 4-1 中，项目名为 IOM，项目的源文件为 IOM.c、cp.c、LCD.c 及 keypad.c。在 other files 中列出的文件(ilcd.h、cp.h)不作为项目的一部分进行编译。源代码编辑器位于屏幕的右侧。编辑器一次可以打开很多源文件，根据用户的需要，可以将这些文件平铺、重叠或最小化。在屏幕的顶部是菜单和工具栏，这些东西在本章中都会用到。最后，在屏幕的底部是 Messages 选项卡，编译过程中出现的编译错误和警告都会列在这里。

1. 打开已有项目

选择 File | Open 命令或单击工具栏的 Open File 按钮可以打开一个已有项目。进行以上任意一项操作后会打开一个 Open File 对话框。浏览到适当的目录选择要打开的项目文件。项目文件打开以后会保留最后一次使用时的状态。图 4-1 所示是一个典型的打开的项目文件导航器中会列出项目名和可用的文件。

2. 新建项目

选择 File | New 命令或在工具栏单击 Create New File 按钮可以新建一个项目。使用这两种方法都会打开一个如图 4-2 所示的 Create New File 对话框，要求指定要打开的文件的类型。选择 Project 单选按钮，然后单击 OK 按钮。会弹出如图 4-3 所示对话框，要求您确认是否使用 CodeWizardAVR 新建项目。代码向导可以为项目自动产生部分源代码，在本章后面的内容中会深入地讨论该工具。不使用代码向导创建项目，则在提示是否使用代码向导时单击 No 按钮即可。最后一个对话框要求指定项目的文件名和位置。

在这些步骤都完成之后，CodeVisionAVR 就会新建一个项目并打开一个用于等项目的 Configure Project 对话框。该对话框有几个选项卡，可以用来为具体的应用程序定制项目，如图 4-4 所示。

图 4-1　CodeVisionAVR 中一个已经打开的项目

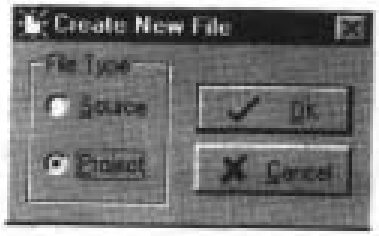


图 4-2　Create New File 对话框

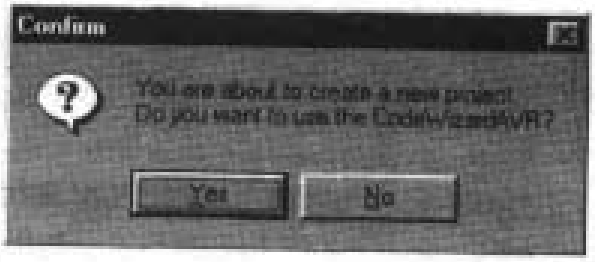


图 4-3　CodeVisionAVR 的 Confirm 对话框

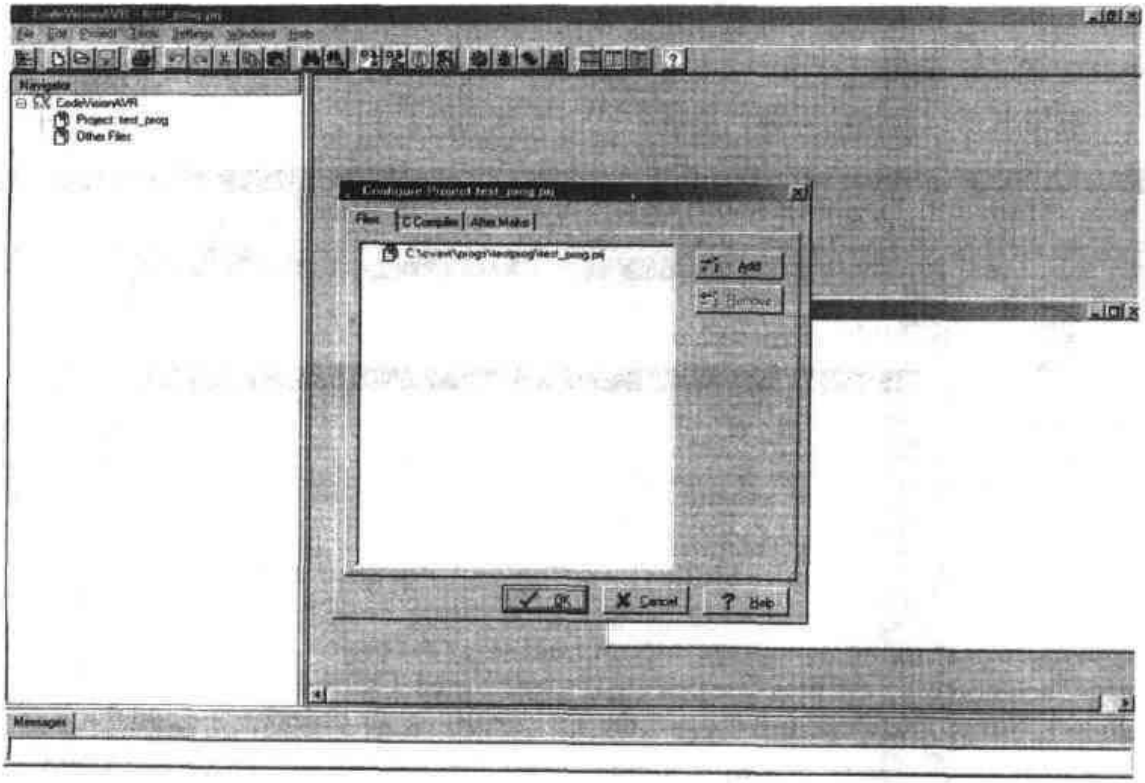


图 4-4　新建一个项目后的 CodeVisionAVR

3. 配置项目

选择 Project | Configure 命令或在工具栏单击 Project Configure 按钮都可以配置项目。使用这两种方法都会打开 Configure Project 对话框。这个对话框中有 3 个选项卡：Files、C Complier 和 After Make 选项卡。在 Files 选项卡中可以向项目中添加源文件或者从项目中删除源文件。在 C Complier 选项卡中可以为项目设置与目标设备及可执行程序文件有关的属性。最后，在 After Make 选项卡可以选择生成过程完成之后要运行的具体程序(如 chip programmer)。在本章后面的部分会更详细地讨论 Files 和 C Complier 选项卡。

4. 关闭项目

选择 File | Close Project 命令可以退出当前的项目。如果项目文件被修改过但是还没有保存，IDE 就会提示您保存修改过的文件。在保存文件时，IDE 会创建一个后缀名为.pr~的备份文件。

4.3.2　源文件

源文件包含程序源代码的文件。这些文件都是您辛勤劳动的结果，它们使微处理器能按照您的要求工作。IDE 允许您向项目中添加源文件或者从项目中删除源文件。IDE 还有一个功能强大的编辑器用于编辑代码。该 IDE 编辑器的一些特性非常适合进行代码编辑，在开发和调试代码时都非常有用。

1. 打开一个已有源文件

选择 File | Open 命令或者在工具栏单击 Open File 按钮可以打开一个已有源文件。弹出的 Open File 对话框可以用来浏览目录，以定位要打开的文件。选择目标文件后单击 OK 按钮就可

以打开这个文件。另外，在文件导航器中单击某个文件名也可以打开相应的文件。打开的文件会出现在 IDE 的一个编辑器窗口中。虽然在图 4-5 中有很多文件被打开了，但 cp.c 是当前被选中的文件。编辑器窗口顶部的标题栏会列出选中文件的路径和文件名。

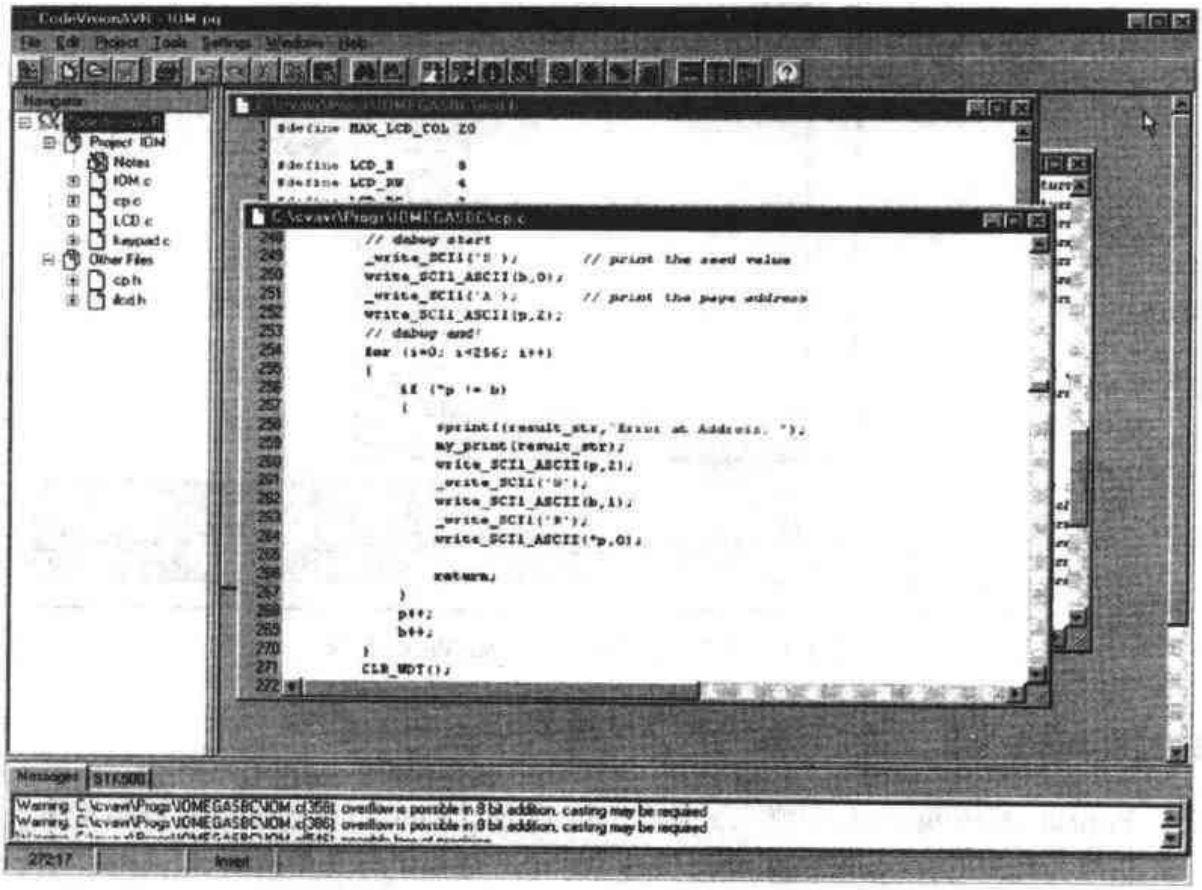


图 4-5　源文件 cp.c 被选中

2. 新建一个源文件

选择 File | New 命令或在工具栏单击 Create New File 按钮可以新建源文件。同样弹出的 Create New File 对话框会提示您选择一种文件类型。选择 Source 单选按钮并单击 OK 按钮，IDE 就会为新建的文件打开一个新的编辑器窗口，如图 4-6 所示。标题栏显示文件名为 untitled.c，该文件列在文件导航器的 Other Files 目录下。

选择 File | Save As 命令可以用一个新名字保存这个文件。在图 4-7 可以看到图 4-6 中新建的文件被保存为 mynewfile.c。在标题栏和文件导航器都可以看到这个变化。

3. 向项目中添加一个已有文件

简单地打开一个文件或新建一个文件并不能把它们自动地添加到当前的项目中。要想将文件添加到一个项目中，必须打开项目，然后选择 Project | Configure 命令或者在工具栏单击 Project Configure 按钮打开 Configure Project 对话框，切换到 Files 选项卡，该选项卡中列出了项目名和源文件名。在对话框右边单击 Add 按钮可以向项目中添加文件，单击 Remove 按钮可以从项目中删除文件。

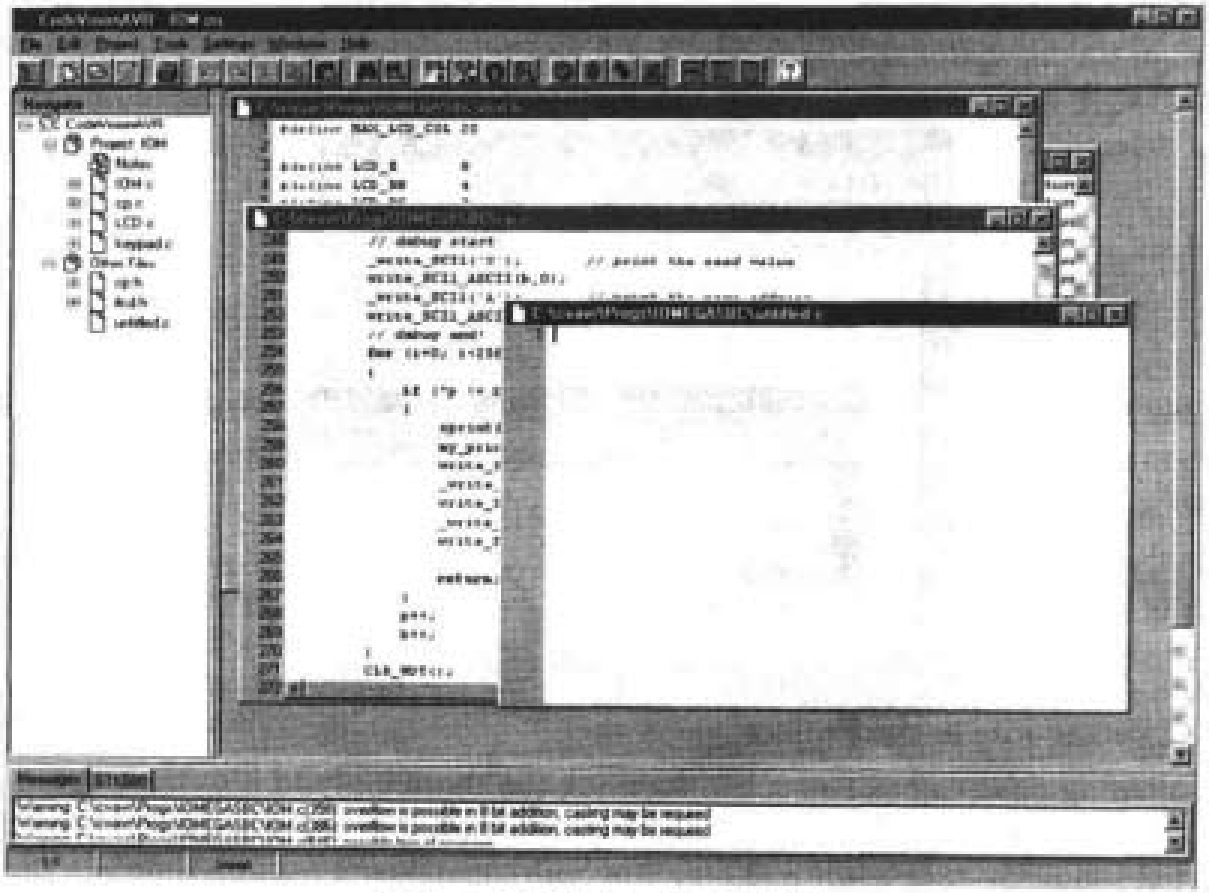


图 4-6　新的源文件 untitled.c 被创建

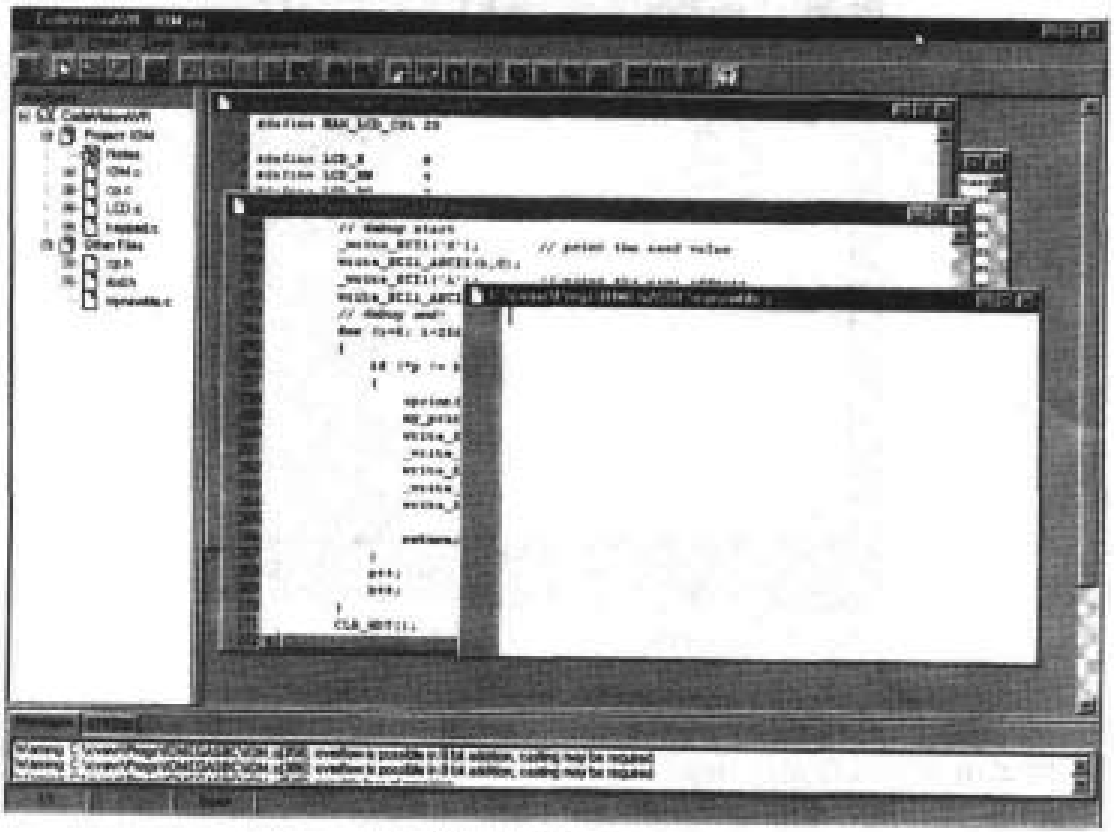


图 4-7　创建新的源文件被重命名为 mynewfile.c

图 4-8 是一个 Project Configure 对话框，其中列出了属于项目的 iom.c、LCD.c、keypad.c 和 cp.c 等源文件。

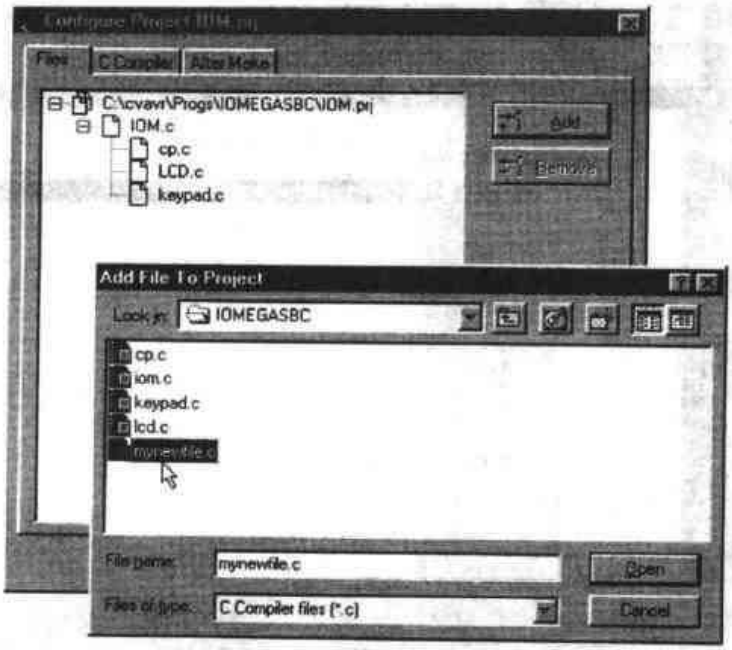


图 4-8　Configure Project 对话框，单击 Add 按钮

单击 Add 按钮，然后选择新建的文件 mynewfile.c，将其作为一个源文件添加到项目中。新的源文件列表如图 4-9 所示。

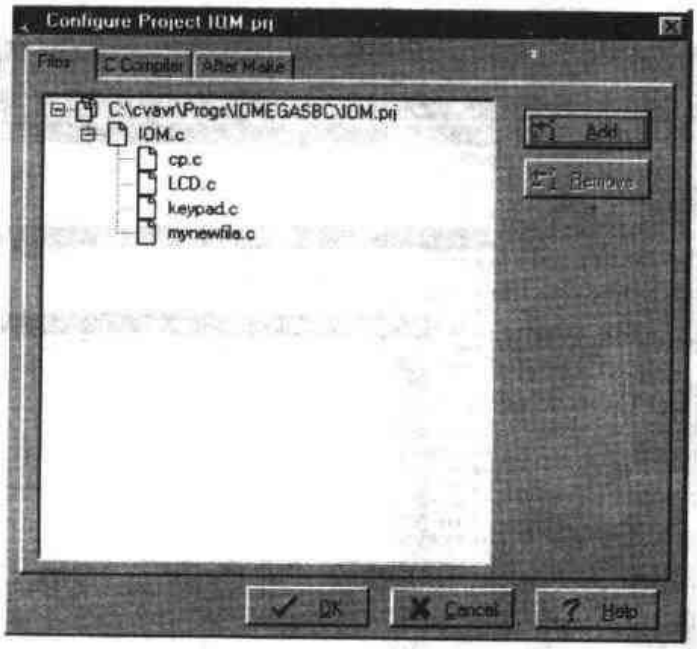


图 4-9　mynewfile.c 作为一个源文件被添加到项目中

在关闭 Configure Project 对话框后，文件导航器中的内容显示 mynewfile.c 已经是项目的一部分了。在图 4-10 中可以看到在 Other Files 目录中已经没有这个文件了。

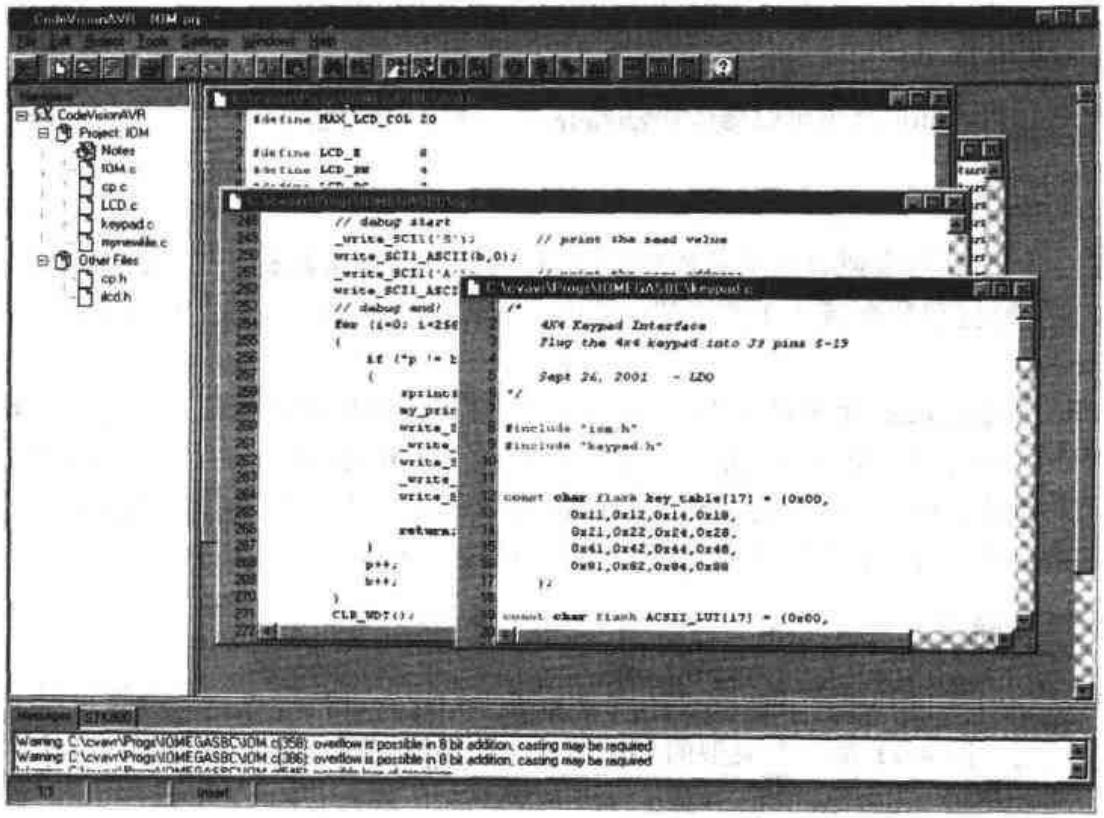


图 4-10　文件导航器目录中已经将 mynewfile.c 作为一个源文件

4.3.3　编辑文件

内嵌于 CodeVisionAVR IDE 的编辑器支持标准编辑器功能。可以使用 HOME、END 键及鼠标指针移动光标。单击并拖拽鼠标可以选择部分文本进行整体操作。使用标准的快捷键、Edit 菜单或鼠标右键可以对选定的文本进行剪切、复制及粘贴等操作。同时它还支持查找、替换以及撤销、恢复等编辑功能。由于编辑器是专为编辑代码设计的，所以具有很多优秀的特性。某些特性可以使用户很容易地在文件中移动，Alt+G 快捷键或者 Edit | Goto Line 命令可以将光标移动到指定的行。使用 Edit | Toggle Bookmark 命令或者 Shift+Ctrl+0 到 9 快捷键可以在当前的光标位置插入和取消书签。Edit | Jump to Bookmark 命令或 Ctrl+0 到 9 快捷键可以将光标跳转到上次设置的书签处。书签可以使我们在程序内不用翻页或记住代码的行号而移动到想要找的代码。

编辑器有两个特性用于简化格式化的代码从而提高程序可读性。第一，编辑器可以缩进文本块和撤消缩进文本块。单击并拖拽鼠标可以高亮显示文本，然后按 Ctrl+I 或 Ctrl+U 快捷键就可以将文本块缩进或者撤销缩进一个跳格键的空间。这在 if 语句或 while 语句里会很有用。第二，编辑器支持自动缩进特性。在按回车键时，光标会直接移动到上一行代码的开始处的下方，而不是移动到下一行的最左端。这一特性可以通过 Settings | Editor 命令控制。

编辑器还支持括号匹配特性。如果将光标定位到一个左括号或右括号处，选择 Edit | Match Brace 命令或者按 Ctrl+M 快捷键可以使位于与之匹配的右括号或左括号之间的文本部分高亮显示。程序员可以使用这一功能检查 if 语句，while 语句等的括号是否成对使用。按任意键或者单击可以取消文本的高亮显示。

最后，编辑器还有语法高亮显示特性，这使代码更易于读写。语法高亮显示特性使用特殊的颜色显示关键字、字符串、常量和注释，以便它们可以在您书写代码或者复查代码时突出显示。使用 Settings | Editor 命令可以修改语法高亮显示使用的颜色。

4.3.4　打印文件

选择 File | Print 命令或者在工具栏单击 Print 按钮可以使用默认的 Windows 打印机打印当前的活动文件。要打印某个活动文件的某一部分，可以先选中要打印的文本部分，然后选择 Edit | Print Selection 命令。

选择 File | Page Setup 命令可以修改当前的打印设置。选择该命令打开 Page Setup 对话框，可以选择是否打印行号、页头和语法高亮显示。Page Setup 对话框也可以指定打印的页边距及页边距使用的单位。Page Setup 对话框的 Printer 按钮用于设置打印机，单击该按钮会打开第二个对话框，用于修改打印机设置或者选择一个不同的打印机。

4.3.5　文件导航器

文件导航器用于显示和打开源文件。在导航器窗口单击文件名可以在编辑窗口中最大化或者打开一个文件。图 4-11 是一个典型的导航器窗口。项目名位于窗口最顶端，项目的源文件直接列在项目名的下面。已打开的但不属于该项目的源文件列在 Other Files 目录下，位于窗口的底部。

图 4-11　文件导航器窗口

如果在文件名的左侧有一个+或者 - 按钮，说明该文件包含更多的信息。单击+或 - 可以展开或折叠这些信息。这些信息可能包含一些在每个被编译的 C 源文件中声明的全局变量和函数(它们只有在文件被编译之后才会出现)。单击变量名或函数名可以在相应的 C 源文件高亮显示相应的变量或函数声明。在导航器窗口也显示了编译过程中产生的错误和警告。图 4-11 所示是一个展开了一些文件分支的示例。

文件导航器还显示 Edit 菜单下 Find in Files 命令的结果。当运行这个命令时，它根据指定的内容搜索导航器列出的所有文件。包含查找内容的文件会作为结果列在文件导航器窗口中。单击 Found 行的+按钮可以查看查找结果。图 4-12 是使用 Edit | Find in Files 命令查找 sprintf 的结果。在 CodeVisionAVR 窗口的底部，在 Messages 选项卡旁边会出现一个 Find in Files 选项卡窗口，其中列出了查找结果。

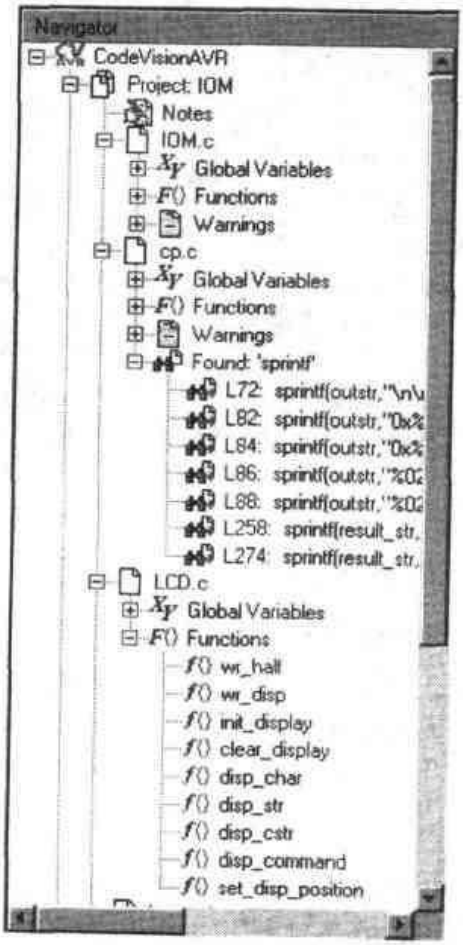


图 4-12　Find in Files 命令执行后的文件导航器窗口

4.4　C 编译器选项

为了产生适当的可执行程序文件，编译器必须知道一些目标设备的关键信息。这些信息包括芯片类型、设备数据堆栈的大小，以及应用程序的内部 SRAM 和外部 SRAM 的大小。编译器还必须知道如何在应用程序的源文件中解释和优化 C 代码。

选择 Project | Configure 命令可为当前打开的项目设置 C 编译器选项。选择该命令会打开 Configure Project 对话框，其中包含的几个选项卡用于不同的组的设置。打开 C Complier 选项卡如图 4-13 所示。

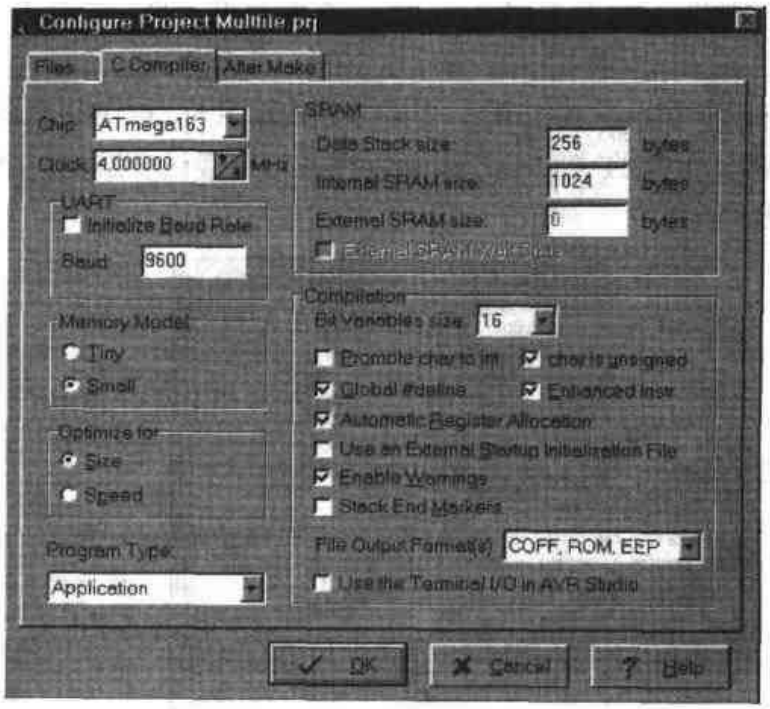


图 4-13　Configure Project 对话框的 C Compiler 选项卡

两个最常用的设置是 Clip(芯片型号)和 Clock(时钟频率)。Chip 下拉列表框用于选择目标 AVR 微控制器的芯片(注意 CodeVisionAVR 的测试版和精简版并不支持所有的 Atmel AVR 处理器。http://www.hpinfotech.ro 给出了每个版本支持的芯片类型)。Clock 微调框用于设置系统时钟，是必须指定的。可以直接输入频率值或使用上下箭头调整得到合适的值。

4.4.1　UART

在 UART 选项组内，选中 Initialize Baud Rate 复选框可使编译器自动产生启动代码去初始化 UART 波特率。要求的波特率可以在复选框下面的文本框输入，也可以根据 Clock 微调框中输入的系统时钟值计算得到。启动代码不会添加到源文件中，但是会添加到编译器产生的汇编文件输出程序中。其启动代码的汇编程序可以通过打开汇编程序创建的列表(.lst)文件来查看。这与 CodeWizardAVR 将串行通信例程添加到应用程序的方式不同，CodeWizardAVR 添加的是串行通信例程，而 UART 代码则被设置到源文件中，在源文件中可以阅读和编辑它。您也可以选择在启动代码中既不用 CodeWizardAVR 也不用 UART 初始化程序，而自己手工编写 UART 初始化程序。

4.4.2　存储器模式

需要的存储器模式可在 Memory Model 选项组中选择。为了提高代码效率，CodeVisionAVR 实现了两种存储器模式。Tiny 单选按钮使用 8 个位来存储指向 SRAM 中变量的指针。在这种存储器模式下，程序只能访问 SRAM 中最开始的 256 字节。Small 单选按钮使用 16 个位来存储指向 SRAM 中变量的指针。这种模式下可以访问的 SRAM 的容量为 65536 字节。为了提高程序速度，缩短程序长度，应该尽可能地使用 Tiny 存储器模式。指向 FLASH 和 EEPROM 存储器区域的指针一般都使用 16 位，因此存储器模式的选择对这些区域已经没有影响。

4.4.3　优化

选择 Optimize For 选项组的 Size 或 Speed 单选按钮可以将编译的程序优化为最小长度或者最大执行速度。通常会要求优化软件的大小，因为微控制器的存储器容量是有限的。当选择优化大小时，编译器会十分小心地抽出在几个地方重复出现的代码并为其创建一个子例程。每次需要这段代码时就用子例程调用来代替。如果在编译之后查看.lst 文件，就会发现编译器为了节省空间创建了很多子例程。然而，每次子例程调用都会消耗执行时间。而某些应用程序对时间的要求比较严格，所以需要顺序执行重复出现的代码。在这种情况下，就要选择优化速度，不选择优化大小。

4.4.4　程序类型

对于允许自编程的设备而言，在 Program Type(程序类型)下拉列表框可以选择 Application(应用程序)或 Boot Loader(引导加载程序)选项。引导加载应用程序可以在运行时从一个更高的存储器位置重新对主程序存储器编程。这就允许微控制器的软件可以使用连到处理器的串行通信直接更新。如果选择使用引导加载程序编程，微控制器不需要特殊的编程电缆或重设微控制器就可以对微控制器重复编程。

4.4.5　SRAM

CodeVisionAVR 会根据选定的具体微控制器选择数据栈大小和内部 SRAM 大小的默认值。Data Stack Size(数据堆栈大小)是为变量预留的内存大小。数据量大的应用程序要求的变量空间比一般的应用程序大些，这时就要增加数据堆栈大小。这会影响到程序堆栈可用的存储器空间。所以只有在数据容量有特别要求时才增加数据堆栈大小。

4.4.6　编译

Compilation 选项组包含的大部分复选框可以开启或关闭特定的选项。它还包含一个下拉列表框，用于确定使用的输出文件的格式。

选中 Promote char to int 复选框，将 ANSI 字符型操作数转换为整型。对于像 AVR 这样的 8 位芯片微控制器来说，将字符型转换为整型会使代码长度增加，运行速度降低。

如果 char is unsigned 复选框被选中，编译器会默认地将字符型数据当作 8 位无符号数来对待，其范围是 0～255。如果该复选框没有被选中，字符数据类型则默认为 8 位有符号变量，其范围是 - 128～127。将字符数据类型设为无符号类型可以缩短代码长度，提高运行速度。

警告：

有一些判断语句不会对无符号字符数据类型进行预测操作。大于和小于条件语句就是这样的两个语句。如果一个变量用在这类语句中，那么最好使用有符号数据类型。如果变量的值超过了有符号字符数据类型表示的范围，可以将其转换为有符号整数。这看起来似乎有些繁琐，但是可以节省调试时间。

选中 Global#define 复选框将使主源文件的#define 语句在所有的项目文件中可见。一般情况下不选中该复选框，而且#define 语句要放在 include 文件中，所有需要这些定义(definition)的文

件只需简单地包含到能访问它们的适当的文件。如果 Global#define 复选框被选中，那么主源文件中的定义对于其他源文件可用，但是其余的源文件中的定义对于任何其他源文件和主源文件都不可用(主源文件是文件导航器中的第 1 个文件。)

一些 AVR 微控制器支持增强指令集。要使用增强指令集，就需要选中 Enhanced Instruction 复选框。这个复选框允许开启或关闭某些微控制器的增强核心(Enhanced Core)指令，如 Atmega128、Atmega32 或 Atmega8 等微制器。这个复选框只有在控制器支持增强指令时才可用。

编译时是否产生警告信息可以通过 Enable Warning 复选框来设置。调试时暂时禁止警告信息的产生可能比较有用，这样可以防止出现 unreachable code or variables being declared or never used 等信息的出现。在程序的最新版本中工作时，最好开启所有的警告而不要简单地关闭它们。

为了进行调试，Stack End Markers 复选框将会使编译器把字符串 DSTACKEND 和 HSTACKEND 放在数据堆栈和硬件堆栈的末端。当您使用仿真程序或者类似 Atmel 的 AVR Studio 的仿真器调试程序时，可以看到这些字符串是否被重写并相应地修改了数据堆栈的大小。在程序能正确运行后，为了缩短代码长度，可以清除这些字符串。

File Output Format(s)下拉列表框可以选择以下由编译器产生的文件格式：

- Intel HEX　用于大多数可嵌入系统的编程器(programmer)
- COFF, ROM, EEP　Atmel AVR Studio 调试程序使用 COFF，其他一些可嵌入到系统的编程器使用 ROM 和 EEP
- OBJ, ROM, EEP　用于某些可嵌入到系统的编程器。

4.5　编译和生成项目

要得到一个可执行文件，需要执行以下两步工作：

(1) 将项目的 C 源文件编译成汇编程序源文件。

(2) 汇编汇编程序源文件。

编译文件只需步骤(1)，产生可执行文件需要执行步骤(1)和步骤(2)。

4.5.1　编译项目

CodeVisionAVR C 编译器可由 Project | Compile File 命令、F9 键或工具栏的 Compile 按钮调用。执行 CodeVisionAVR C 编译器产生一个扩展名为.asm 的汇编程序源文件。在编译过程之后，会打开一个 Information 对话框显示编译结果，如图 4-14 所示。Information 对话框的顶部列出了编译器的选项，下面是被编译的代码行数和警告、错误的个数。对话框的其余部分列出了存储器的使用量和用途。

如果在编译过程中有错误或警告产生，它们就会出现在编译器窗口下面的 Messages 选项卡以及导航器窗口中。双击错误或警告信息可以高亮显示出问题的代码行。在图 4-15 中，第 1 个警告被双击，相应的代码行被高亮显示。

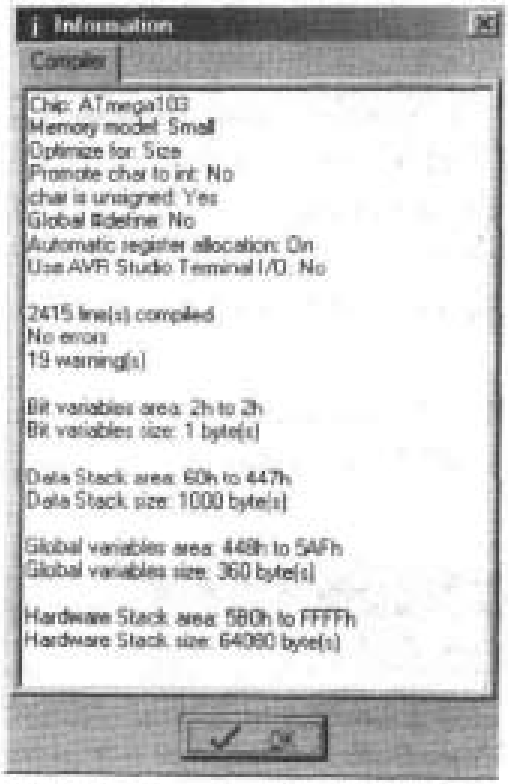


图 4-14　Information 窗口显示编译结果

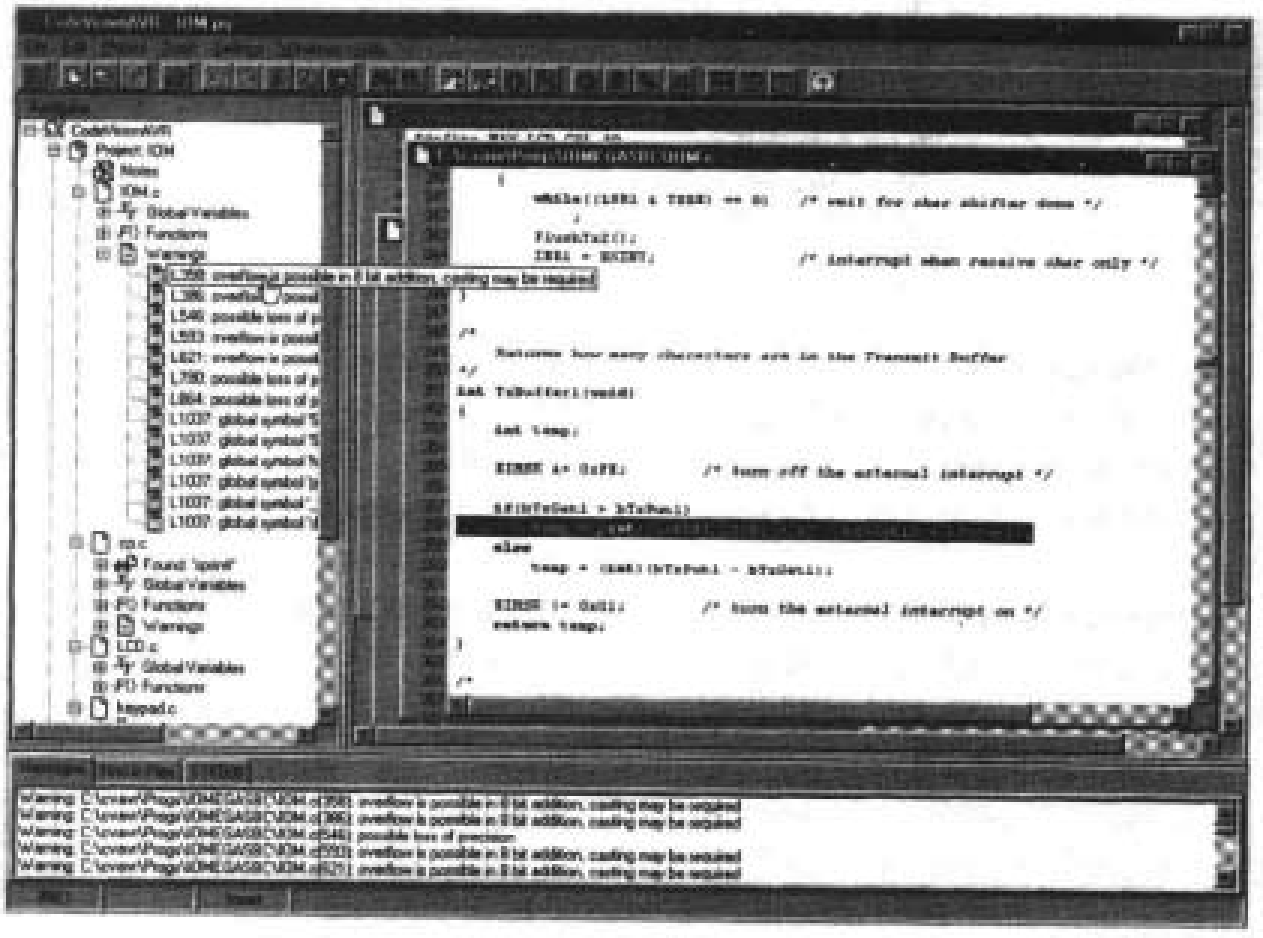


图 4-15　在 Messages 选项卡和导航器中显示的错误

4.5.2 生成项目

生成项目会创建一个可执行文件。选择 Project | Make 命令或按 Shift+F9 键，或者在工具栏单击 Make 按钮都可以生成一个项目。CodeVisionAVR C 编译器的执行会产生一个扩展名为.asm 的汇编程序源文件。如果编译器没有产生错误，Atmel AVR 汇编程序 AVRASM32 就会被自动调用，并运行新生成的汇编程序源文件。汇编程序会创建一个可执行程序文件，其格式是 Configure Project 对话框的 C Compiler 选项卡中指定的格式(选择 Project | Configure | C Compiler 命令)。

在生成过程完成之后，会弹出 Information 对话框显示编译结果。在这个对话框有两个选项卡：Complier 和 Assembler。从这两个选项卡可以查看编译和汇编的结果。图 4-14 是一个编译结果。图 4-16 所示为汇编结果。

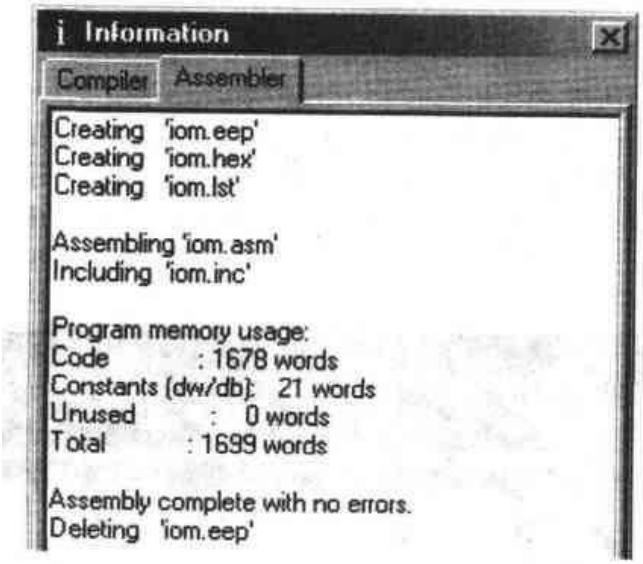


图 4-16　汇编结果

Assembler 选项卡顶部显示了编译的步骤。编译器为 EEPROM 初始化过程创建的文件放在 iom.eep 中。第二，编译器为汇编程序创建一个.asm 文件用于存放 iom.hex 中的汇编代码。最后，编译器创建扩展名为.asm 的文件供汇编程序使用。如果编译器没有检测到错误，汇编过程开始，同时创建可执行文件和.lst 文件。对话框的下半部分是汇编程序的反馈结果，说明了程序使用的存储器量。

总存储器使用量的下面有一行非常重要的信息：Assembly complete with no errors。偶尔会发生编译正常而汇编出错的情况。如果发生这种情况，会打开一个对话框并显示汇编过程中发生错误的消息(Information 对话框的 Assembler 选项卡的)这一行显示了汇编过程中发生的错误的个数。在.lst 文件中查找 Error 一词可以找出错误所在。

Assembler 选项卡的底部显示 iom.eep 文件已经删除了。这个文件为 EEPROM 初始化过程创建，但在这个程序中不会出现，所以不需要它。

4.6　对目标设备编程

CodeVisionAVR IDE 有一个内嵌的可嵌入系统的 AVR 芯片编程器(AVR Chip Programmer)，可以很容易地将编译好的程序发送到微控制器用于测试。编程器是为 Atmel STK500, Kanda Systems STK200+/300, Dontronics DT006, Vogel Elektronik VTEC-ISP, MicroTronics ATCPU 和 Mega2000 开发板设计的，这些开发板都可以通过打印机并行端口与计算机连接。选择 Setting | Programmer 命令会打开一个窗口用于选择使用的编程器的类型。

选择 Tools | Chip 命令或在工具栏单击 Chip Programmer 按钮可以打开编程器，图 4-17 所示为选择 Kanda System Programmer 时打开的芯片编程器。

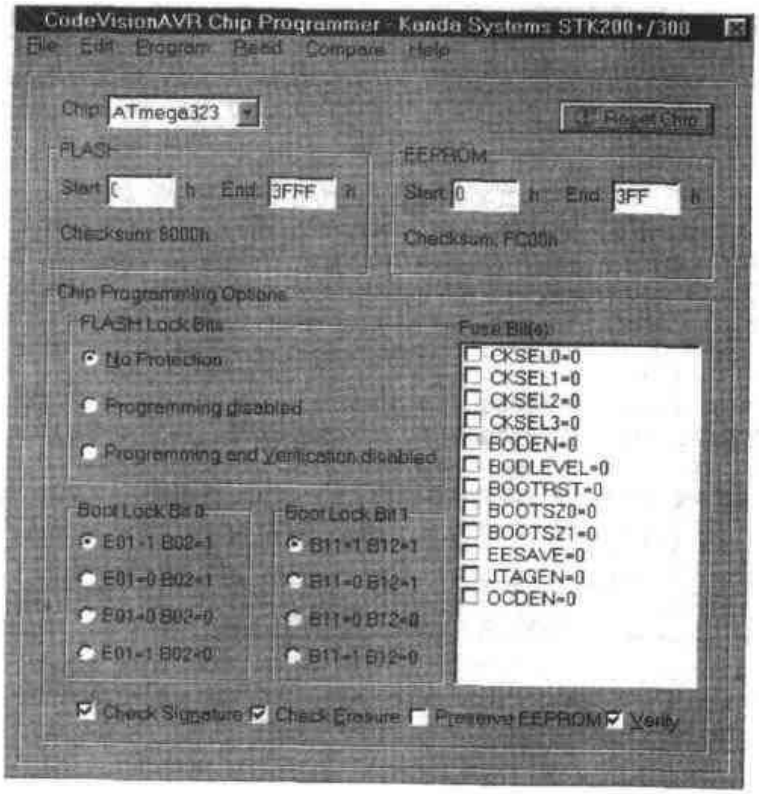


图 4-17　CodeVisionAVR 芯片编程器

4.6.1　芯片

Chip 下拉列表框中的选项决定了 Chip Programmer 对话框中其他可用的选项，在该下拉列表框可以选择需要编程的芯片的类型，选择的芯片类型决定了 FLASH 和 EEPROM 缓存的大小，还决定了显示的内容：Fuse Bit(s)、Boot Lock Bit 0、Boot Lock Bit 1 选项组，或者是否根本就不显示这些选项组。

4.6.2　FLASH 和 EEPROM

编程器有两个存储器缓冲区：FLASH 存储器缓冲区和 EEPROM 存储器缓冲区。缓冲区的内容可以使用 File | Load 和 File | Save 命令加载和保存。支持的文件格式有：

- Atmel .rom 和.eep

- Intel HEX
- 二进制 .bin

在一个文件被加载到相应的缓冲区后，其 Start(起始)地址和 End(结束)地址会相应更新。Checksum 的值也会随加载文件校验和的改变而改变。

FLASH 和 EEPROM 缓冲区的内容是可以查看和编辑的，选择 Edit | FLASH 和 Edit | EEPROM 命令可以显示和编辑相应缓冲区的内容。根据选择的命令，会打开一个 Edit 窗口显示相应缓冲区的内容，如图 4-18 所示。

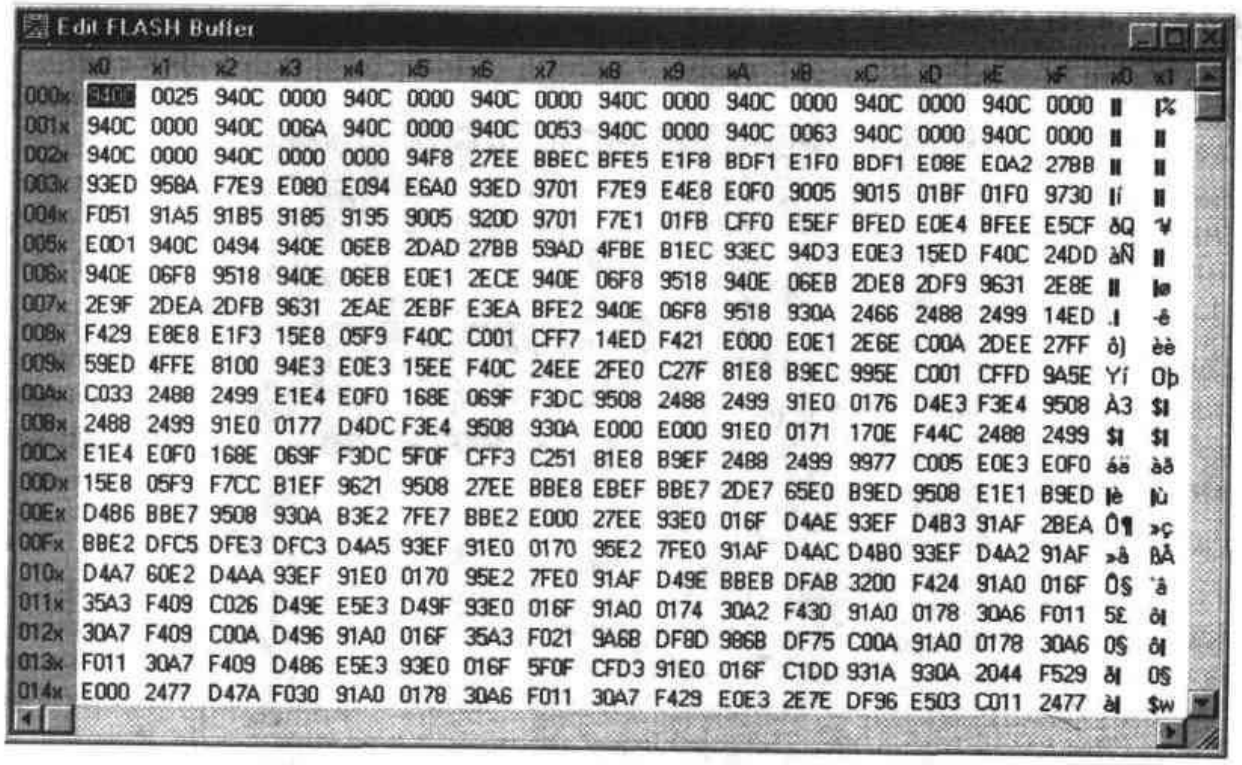


图 4-18　CodeVisionAVR 的 Edit FLASH Buffer 窗口

被高亮显示部分的缓冲区内容可以通过输入一个新值来修改。高亮显示的地址可以按箭头键、Tab 键、Shift+Tab 键、PageUp 键和 PageDown 键来修改。

有时用一个给定值填充一块 FLASH 或 EEPROM 存储区可能很有用，在 Edit 窗口右击可以打开 Fill Memory Block 对话框，如图 4-19 所示。在这个对话框可以指定被填充区域的起始地址(Start Address)、结束地址(End Address)和填充值(Fill Value)。

图 4-19　CodeVisionAVR Chip Programmer Fill Memory Block 对话框

分别使用 Program、Read 和 Compare 命令可以对 FLASH 和 EEPROM 缓存编程使其成为芯片，并从芯片读取数据及比较芯片数据。在对 FLASH 缓冲区编程前一定要先擦除芯片内容。如果使用 Program | All 命令，芯片擦除将自动完成。如果芯片被锁定，对芯片上的 FLASH 和 EEPROM 的读取和比较操作将不可用。

4.6.3　FLASH 锁定位

Chip Programmer 对话框的 FLASH Lock Bits 选项组有几个选项可以选择芯片编程使用的安全级别。安全级别包括以下几种：No Protection(允许所有的读取和编程操作)，Programming Disabled(对 FLASH 和 EEPROM 存储区只能读，不能写),Programming and Verification disabled(不能对芯片内容进行任何读写操作)。在完成对芯片编程之后，锁定位(Lock Bit)的状态将不再有效。FLASH 在内容被擦除之前不能再次进行读写操作。

在选择 Program | All 命令之后，锁定位将自动地写到芯片上。选择 Program | Lock Bit 命令可以单独写入锁定位。

使用 Read | Lock Bit 命令可以读取锁定位当前的值。芯片擦除操作会将所有的锁定位设为 No Protection。

4.6.4　保险位

如果您选择使用的芯片具有可编程的保险位(Fuse Bit)，就会出现一个辅助的 Fuse Bit(s)选项组。在这个选项组中有各种复选框，其是每个对应所选芯片上的一个保险位。保险其使用在 Atmel 的数据表中有详细描述。

选择 Program | Fuse Bits 命令可以对芯片的保险位编程，选择 Read | Fuse Bits 命令可以从芯片读取保险位。选择 Program | All 命令进行自动编程的过程中也会自动地对保险位编程。

4.6.5　Boot Lock Bit 0 和 Boot Lock Bit 1

支持引导加载程序的芯片有更多的安全位，可以锁定存储器中的引导加载区。Boot Lock Bit 0 和 Boot Lock Bit 1 选项组和选项支持这些附加的安全位。如果想知道有关这些位的具体用法，可以参阅 Atmel 的数据表。

4.6.6　签名

Atmel AVR 系列中不同型号的芯片有其惟一的签名。选择 Read | Chip Signature 菜单可以读取芯片的签名，以确定连接到编程器的芯片类型。有些编程器在执行任何操作前，都会用签名确认所选芯片与实际硬件的连接。不同类型芯片有不同的编程参数，所以如果实际芯片的类型与选择的芯片类型不匹配就有可能会发生不可预知的结果。

然而有时需要将一种芯片作为另外一种类型来编程和读取数据，例如 Atmega128 刚刚推出时，很多编程器还只支持 Atmega103 模式，所以有必要忽略 Atmega128 的签名。正是由于这个原因，才提供一个可以开启和关闭读取签名的复选框。如果签名不正确而且 Check Signature 又被选中，编程将不会继续编程。如果未选中 Check Signature 复选中，将不管签名而继续编程。

4.6.7　芯片擦除

在使用 Program | All 命令时，芯片擦除会自动完成。如要在不影响编程操作的情况下擦除芯片，可以使用 Program | Erase Chip 命令。Program | Blank Check 命令检查芯片的 FLASH 和 EEPROM 存储器区域是否已被擦除。如果 Check Erasure 复选框被选中，FLASH 和 EEPROM 是否被擦除的检查也可以由 Program | All 功能自动完成。

Preserve EEPROM 复选框通过芯片擦除循环来保存 EEPROM 的数据。它首先将 EEPROM 中的内容读入一个缓冲区，然后擦除芯片，最后将缓冲区的内容写回到 EEPROM 中。如果芯片被锁定为禁止读取数据，则 EEPROM 的内容将不被保留，因为数据根本无法检索。

4.6.8　编程速度

取消选中 Check Erasure 复选框可以提高编程速度。这种情况下，就不会检查芯片擦除的正确性。还可以取消选中 Verify 复选框，这样就不会检查 FLASH 和 EEPROM 编程的正确性。这在快速调试代码以及测试目标设备时比较有用。在编制芯片代码的最终版本时，最好选中 Check Erasure 和 Verify 复选框。

4.6.9　Program | All

Program | All 命令会自动化编程过程。这个命令可能是在编程器中使用最多的命令。其步骤如下：

(1) 擦除芯片。

(2) FLASH 和 EEPROM 空白检查(如果 Check Erasure 复选框被选中)。

(3) 编程、检查 FLASH。

(4) 编程、检查 EEPROM。

(5) 编程保险位和锁定位。

4.6.10　其他编程器

CodeVisionAVR 不直接支持的可嵌入系统的编程器如果支持命令行工具，就可以被添加到工具列表中。我们的例子显示可嵌入系统的编程器 TheCableAVR 作为一个优秀的编程器，可以通过命令行被 CodeVisionAVR 调用。选择 Tools | Configure 命令可以添加一个新工具。在 Configure Tools 对话框打开之后，单击 Add 按钮会打开一个浏览器，用于选择可执行文件。图 4-20 所示是 TheCableAVR 在 Configure Tools 对话框被添加到 Tools 菜单的过程。

一旦工具被添加到 CodeVisionAVR 以后，单击 Settings 按钮可以打开 Tool Settings 对话框对工具进行设置：设置工具在 Tools 菜单显示的名字、可执行文件的路径和文件名、工具的命令行参数，以及工具的工作目录等，图 4-21 所示是 TheCableAVR 的设置。

TheCableAVR 的命令行参数很简单，就是 TheCableAVR 的项目名(iom.isp)。TheCableAVR 可以为 FLASH 和 EEPROM 存储器区域创建一个包含程序文件的项目，还可以设置保险位和锁定位。其他几项设置也是项目的一部分，但是都取决于具体的目标设备。

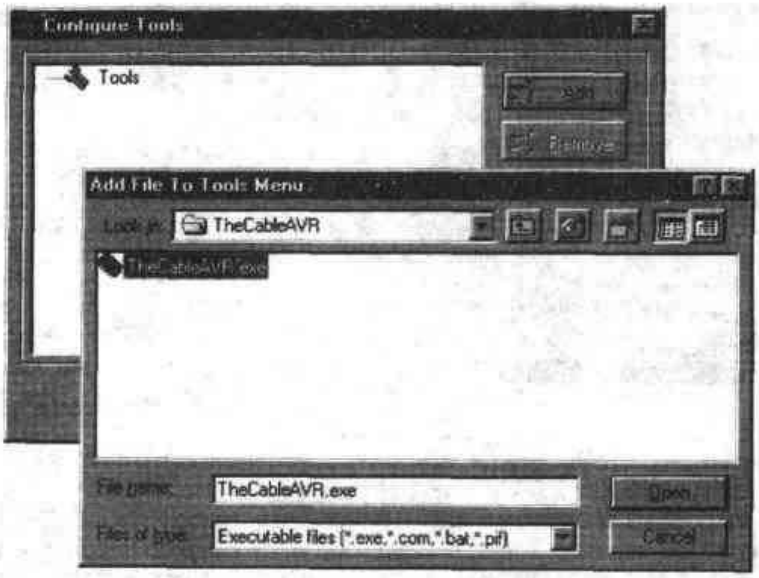


图 4-20　Configure Tools 对话框，单击 Add 按钮

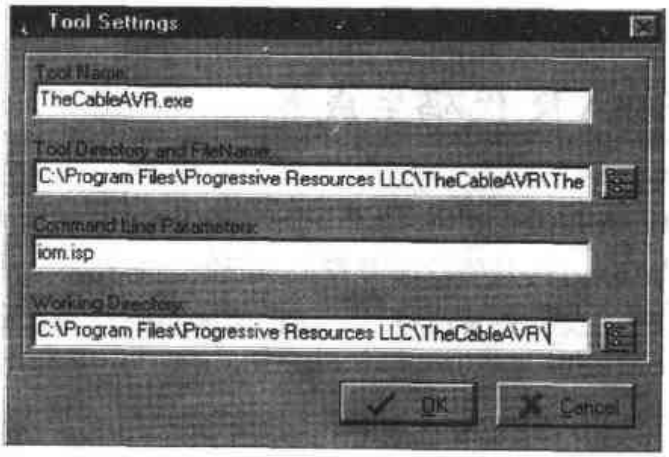


图 4-21　Tool Settings 对话框

工具被添加、设置好以后就会出现在 Tools 菜单中。图 4-22 所示为 TheCableAVR.exe 列在了 Tools 菜单中。只要简单地选择这个命令就可以调用 TheCableAVR 对芯片编程了。TheCableAVR 对话框打开，加载项目文件，然后进入自动编程循环。如果在编程循环中发生错误，TheCableAVR 对话框会停留在打开的状态，方便用户查看错误(见图 4-23)。如果编程正确完成，TheCableAVR 对话框会自动关闭。

图 4-22　CodeVisionAVR 的 Tools 菜单

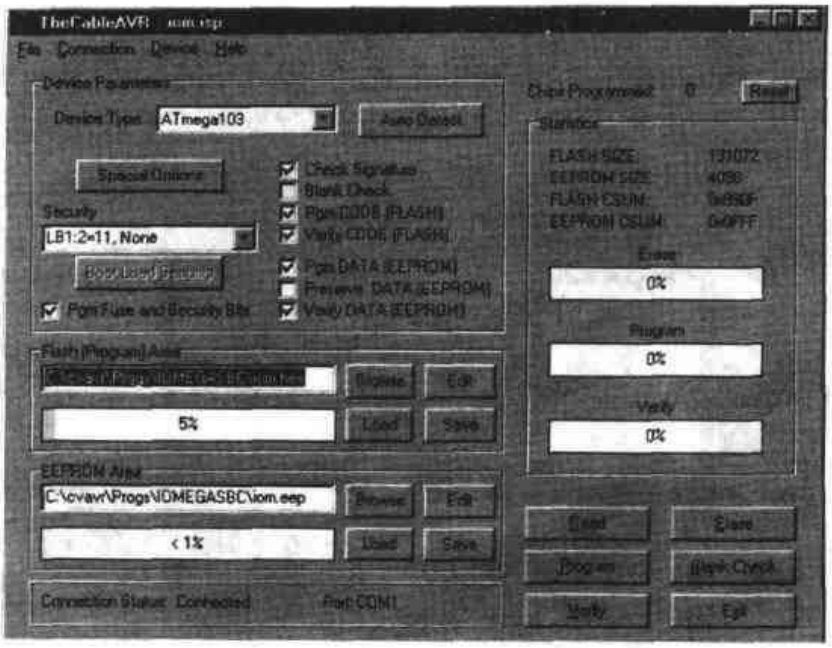


图 4-23　TheCableAVR 软件

4.7　CodeWizardAVR 代码生成器

在新建项目时，您可以选择是否使用 CodeWizardAVR 为项目自动生成一个代码外壳。使用 CodeWizardAVR 可以在项目的开始阶段节省很多时间，它能自动生成代码，设置时钟、通信端口、I/O 端口、中断源以及很多其他特性。这样您就不必查阅大量的有关控制寄存器及它们要求的设置的数据。

要有效地使用 CodeWizardAVR，必须知道微控制器在项目中是如何使用的，还必须对处理器的硬件知识有基本的了解。本书的第 2 章已讲过了很多 Atmel AVR 微控制器的体系结构特性。第 5 章将详细给出一个好的项目建议及其辅助工作，以及编写软件的准备工作。现在要仔细考虑的是哪个 I/O 引脚用作输入、哪个引脚用作输出，通信端口需要什么样的波特率，需要什么样的计时器和中断源。

CodeWizardAVR 直接将产生的代码放入您指定的文件中。这是程序的开始点。因为以后可以编辑这个文件，所以任何设置都可以修改，但是必须查阅数据表中的新设置的资料。

CodeWizardAVR 提供的特性取决于所选择的具体微控制器和 CodeVisionAVR 的版本(注意测试版和精简版 CodeVisionAVR 并不支持所有 Atmel AVR 处理器，它所支持的特性与标准版的 CodeVisionAVR 支持的特性也不完全相同，访问http://www.hpinfotech.ro可了解每个版本的详细信息)。图 4-24 所示为标准版 CodeVisionAVR 中 ATTiny22 可用的选项卡，图 4-25 所示为 Atmega163 的选项卡，可以看到 Atmega163 可用的选项卡多一些。

本节将使用 Atmega163 作为目标设备，内容会覆盖标准版 CodeVisionAVR 可用的一些基本选项。本节中提供的示例代码片断都是针对 Atmega163 设备的。CodeWizardAVR 产生的初始化代码放在 main()函数的顶部，中断例程放在 main()函数的上面。

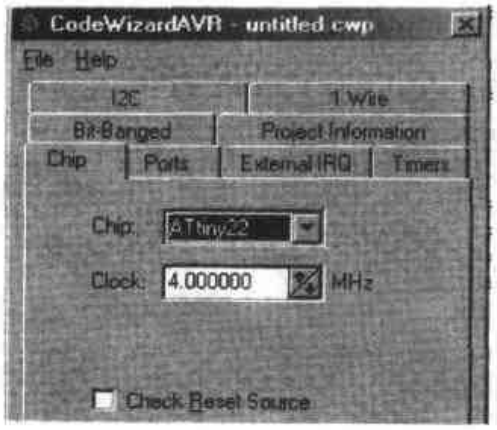


图 4-24　CodeVisionAVR ATTiny22 的选项

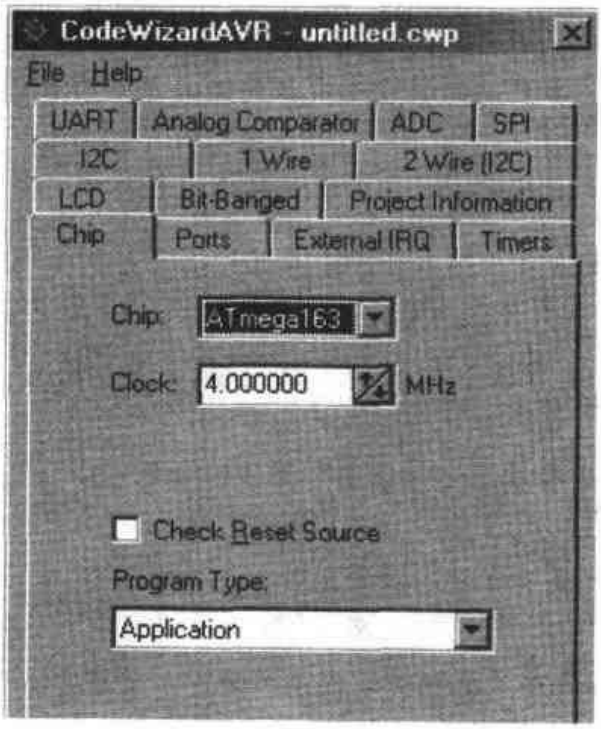


图 4-25　CodeVisionAVR Atmega163 的选项

注意：

CodeWizardAVR 在不断地更新和扩展，本节的目的不是完全解释它所有的特性和可能情况，而只是给出这个工具当前功能的概况。

4.7.1　Chip 选项卡

Chip 选项卡是 CodeWizardAVR 对话框打开时第一个被选择的选项卡。Chip 下拉列表框用于设置目标设备。在这个下拉列表框改变选择的芯片会更新对话框的其余部分。Clock 微调框的时钟频率以 MHz(兆赫)为单位，它的值可以使用上下箭头按钮调整，也可以在编辑区直接输入数字。选择正确的时钟频率对于正确选择其他对时间敏感的设置是非常重要的。

在 Chip 选项卡的底部，选中 Check Reset Source 复选框使代码可以检查复位源。如果选中该复选框，后面的代码就会自动产生。这段代码允许程序能够根据最后的复位源完成某些动作。这段代码放在 main()函数的最开始处。

```
//Reset Source checking
if (MCUSR & 1)
{
   //Power-on Reset
   MCUSR=0;
   //Place your code here
}
else if (MCUSR &2)
{
   //External Reset
   MCUSR=0;
   //Place you code here
}
else if (MCUSR & 4)
{
   //Brown-out Reset
   MCUSR=0;
   //Place your code here
}
else
{
   //Watchdog Reset
   MCUSR=0;
   //Place your code here
}
```

在 Check Reset Source 复选框的下面是 Program Type 下拉列表框。在这里可以选择生成的程序类型是引导加载程序还是一般应用程序。本例选择的是一般应用程序。

4.7.2 Ports 选项卡

Atmega163 有 4 个 I/O 端口：A、B、C、D。图 4-26 中端口 A 被选中。对于端口 A 的每一个引脚，有两个选项可用，数据方向(Data Direction)和上拉/输出值(Pullup/Output Value)。这些设置完成两样事情：第一，数据方向的设置决定 DDRx 寄存器(端口 A 的为 DDRA)的值，用于控制每个引脚是用于输入还是用于输出。第二，上拉/输出值的设置用来建立端口的初始值。如果引脚用于输入，上拉/输出值可以开启或关闭内部上拉电阻。如果引脚用于输出，则这个设置决定端口输出的初始值。

例如，如果使用端口 A 的上半部分用作输入，开启内部上拉电阻器；端口 A 的下半部分用作输出，初始值是 0x0A，Ports 选项卡的设置如图 4-26 所示。

为端口 A 产生的初始化代码如下：

```
// Input / Output Ports initialization
// Port A
 PORTA=0xFA;
 DDRA=0x0F;
```

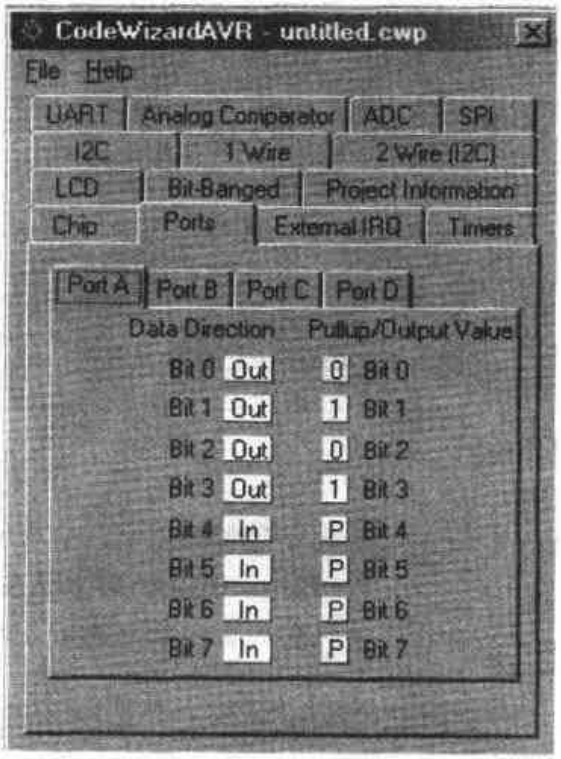


图 4-26　CodeWizardAVR Atmega163 的 Ports 选项卡

4.7.3　External IRQ 选项卡

External IRQ 选项卡有两个复选框可以用于开启或关闭外部中断 INT0 和 INT1。如果中断被开启，就会出现 Mode 下拉列表框，用于决定什么条件引起中断。例如，在图 4-27 中 INT0 被开启并被设为在信号的下降沿产生中断。

下面的中断函数被添加到源文件中：

```
// External Interrupt 0 service routine
interrupt [EXT_INT0] void ext_int0_isr(void)
{
    // Place your code here
}
```

产生的代码使 INT0 在信号下降沿产生中断，该段代码被添加到 main()函数中，内容如下：

```
// INT0: On
// INT0 Mode: Falling Edge
// INT1: Off
GIMSK=0x40;
MCUCR=0x20;
GIFR=0x40;
```

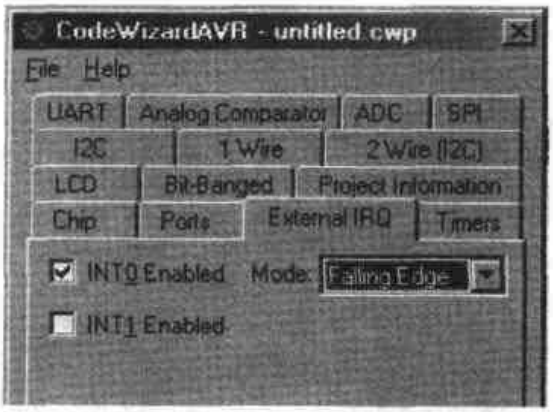


图 4-27　External IRQ 选项卡

4.7.4　Timers 选项卡

图 4-28 所示的 Timers 选项卡是一个用途非常多的工具。它里面包括从 Timer 0 到 Timer 2 以及 Watchdog 共 4 个计时器。在本例中，Timer 0 设置为使用系统时钟作为时钟源，频率是 50kHz。计时器翻转时就生成一个 IRQ。代码向导输出中断例程和用于初始化 Timer 0 的代码。

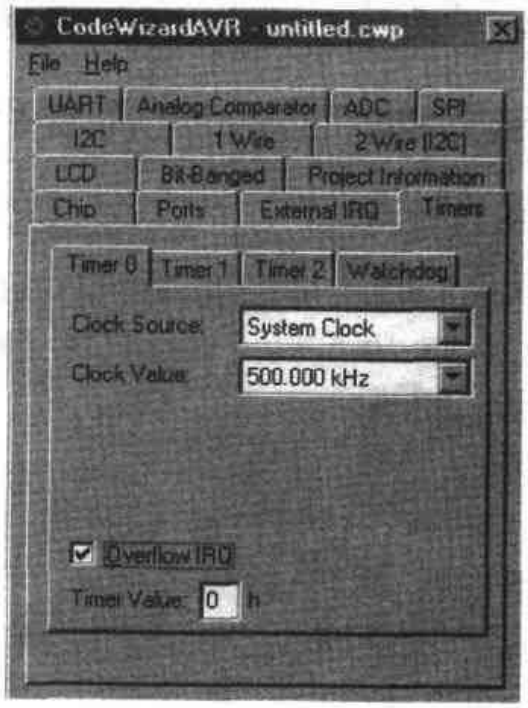


图 4-28　Timers 选项卡

生成的初始化代码如下：

```
// Timer / Counter 0 initialization
// Clock source: System Clock
// Clock value: 500.000 kHz
// Mode: Output Compare
// OC0 output: Disconnected
TCCR0=0x02;
TCNT0=0x00;
```

在计时器溢出时被调用的中断例程的外壳放在 main()函数的上面：

```
// Timer 0 overflow interrupt service routine
interrupt [TIM0_OVF] void time0_ovf_isr(void)
{
// Place your code here
}
```

4.7.5　UART 选项卡

如果 Receiver 或 Transmitter 复选框被选中，则 UART 选项卡会有很多选项。对于每一个复选框，都有一个选项用于驱动接收器或发送器中断，还有一个地方用于设置相关数据缓冲区的长度。当 CodeWizardAVR 生成中断驱动的 UART 例程时，标准 I/O 库函数 putchar()和 getchar()就会被源文件中生成的代码替换。这使 putchar()和 getchar()函数可以由中断驱动。相应地，又会使所有使用 putchar()和 getchar()的函数，如 printf()和 scanf()函数都由中断驱动。

Transmitter 设置域的下面是 UART Baud rate 下拉列表框，这里有很多常用的波特率。实际可用的波特率由驱动设备的时钟频率决定。CodeWizardAVR 会显示出现与所选择的波特率和系统时钟有关的错误的百分率。在本例中，选用的波特率是 9600，波特率错误率是 0.2%。这个错误百分率已经相当小，不会影响到与其他设备的通信。

UART 选项卡的最下方是 Communication Parameters 下拉列表框。选择需要的设置后，CodeWizardAVR 就会为 UART 生成所需的初始化代码以获得这些设置。在图 4-29 中的设置为数据位 8 位，停止位 1 位，没有校验位。

现在，为 UART 生成的中断驱动的代码是汇编语言的形式，位于源文件的顶部。如果使用标准 I/O 库函数例程，就不能修改这些代码(在这里没有给出代码，因为我们希望代码向导产生的汇编语言例程很快会被用 C 写成的例程代替。)

图 4-29 的 UART 设置的初始化代码如下，它放在 main()函数中：

```
// UART initialization
// Communication Parameters: 8 Data, 1 Stop No Parity
// UART Receiver: On
// UART Transmitter: On
// UART Baud rate: 9600
```

```
UCSRA=0x00;
UCSRB=0xD8;
UBRR=0x19;
UBRRHI=0x00;
```

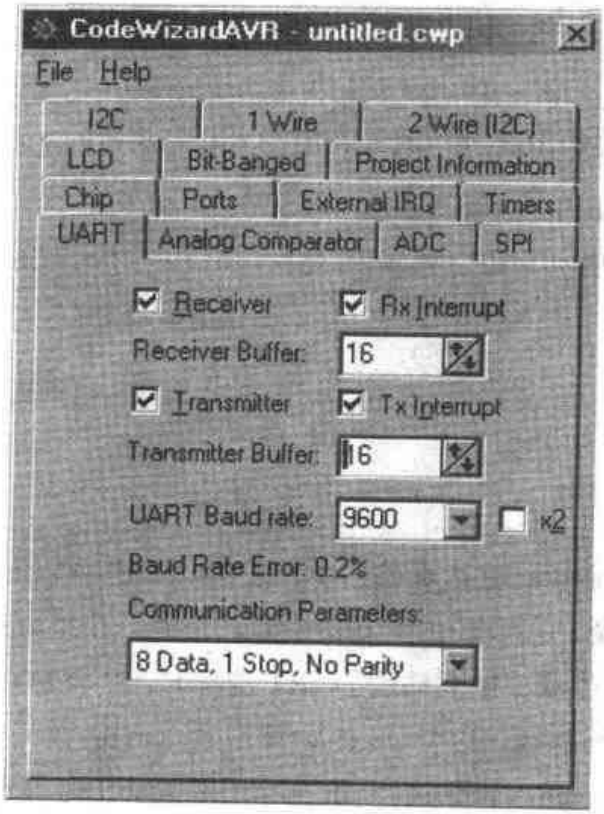


图 4-29　UART 选项卡

4.7.6　ADC 选项卡

ADC(模数转换器)选项卡可以用来生成初始化 ADC 的代码。如果 ADC Interrupt 复选框被选中，就会产生一个在转换完成时要调用的中断例程的外壳。Volt. Ref.下拉列表框用于选择转换器的参考电压。ADC Clock 下拉列表框可以为转换器设置预定标器。设置预定标器是为了分割系统时钟以达到选择的频率。图 4-30 是一个简单的 ADC 设置。

代码向导为图 4-30 中的设置生成的初始化代码如下：

```
// ADC initialization
// ADC Clock frequency: 250.000 kHz
// ADC Voltage Reference: AVCC pin
ADMUX=ADC_CREF_TYPE;
ADCSR=0x84;
```

图 4-30　ADC 选项卡

4.7.7　Project Information 选项卡

Project Information 选项卡的作用是输入项目信息，然后将它们与项目的设置信息一起格式化后写入生成的源文件中。图 4-31 所示为示例项目输入的项目信息。

下面的例子是 Project Information 选项卡产生的添加到源文件中的信息：

```
/*****************************************************************
This program was produced by the
CodeWizardAVR V.0.2.1 Stardard
Automatic Program Generator
© Copyright 1998-2001
Pavel Haiduc, HP InfoTech S.R.L.
http://infotech.ir.ro
e-mail:dhptechn@ir.ro, hpinfotech@xnet.ro

Project    :Chapter 4 Example Project
Version    :1.0
Date       : 1/7/2002
Author     :Sally
Company :Sally's Source Code
Comments:
Atmegal163 Example Project
Chip type              :ATmegal163
Program type           :Application
Clock frequency        :4.000000 MHz
Memory model           :Small
Internal SRAM size     :1024
External SRAM size     :0
Data Stack size        :256
*****************************************************************/
```

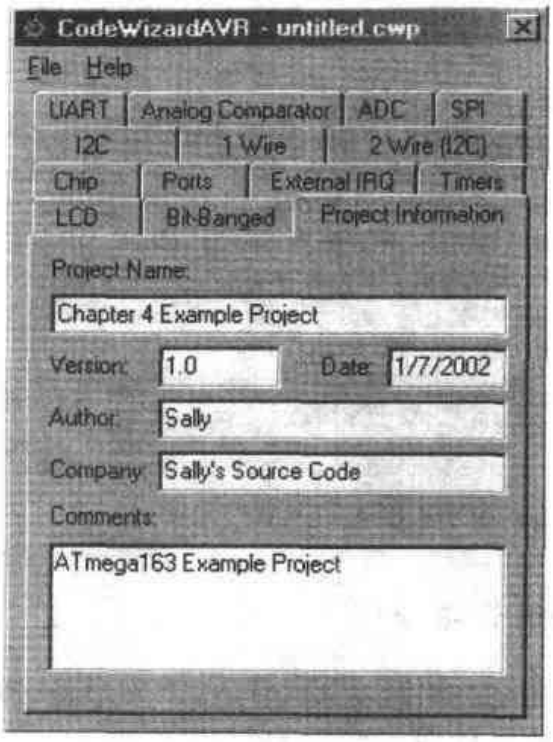


图 4-31　Project Information 选项卡

4.7.8　生成源代码

一旦项目的 CodeWizardAVR 选项设置好后，CodeWizardAVR 就准备生成源文件了。选择 File | Generate, Save and Exit 命令(见图 4-32)。

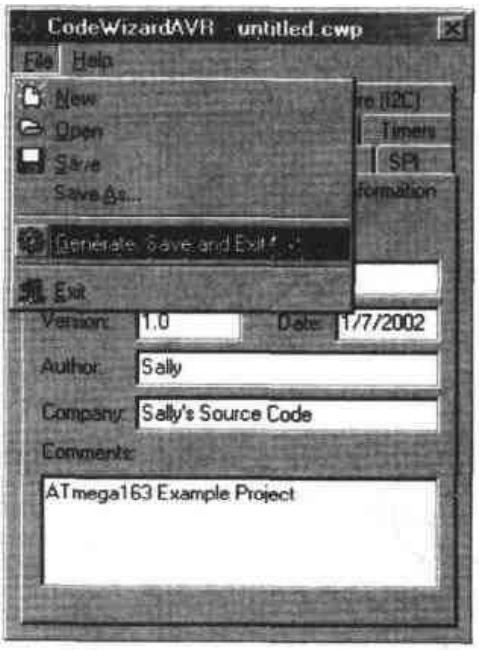


图 4-32　选择 File | Generate, Save and Exit 命令

这样会打开几个浏览窗口用于选择源文件名、项目文件名和代码向导文件名。输入了正确的文件名后，项目就会被创建并打开。CodeVisionAVR 的界面如图 4-33 所示。下面的代码列出了根据前面几节的设置所创建的源代码文件。

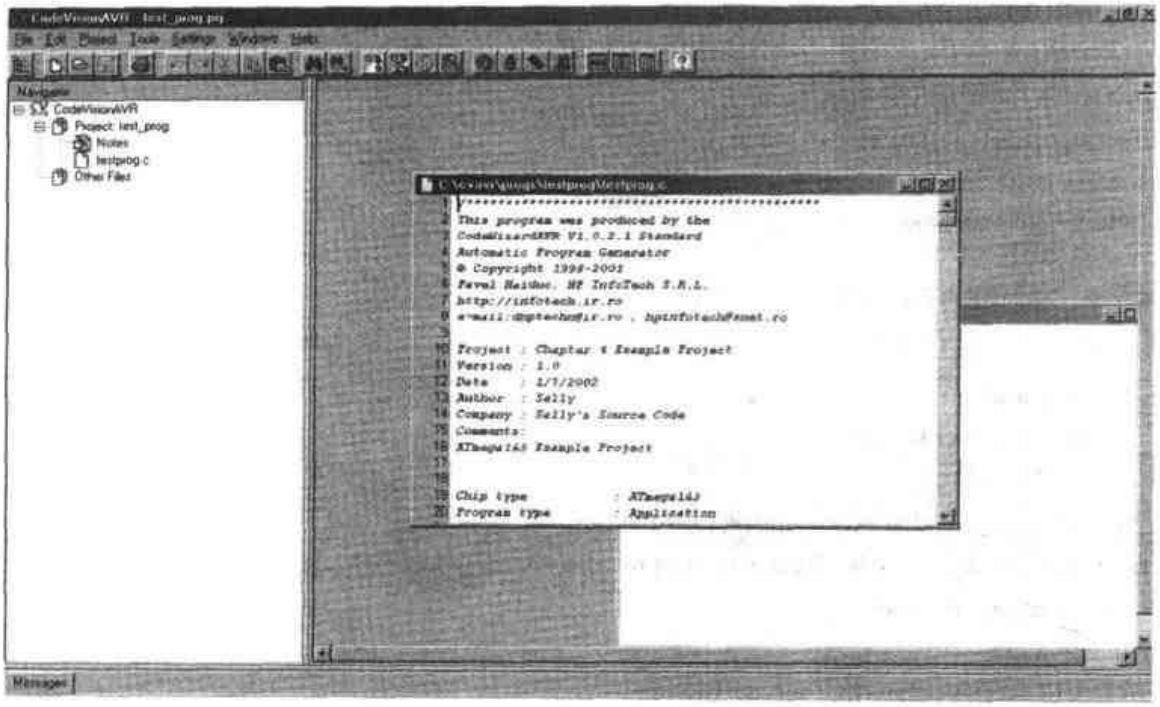


图 4-33　由 CodeVisionAVR 新建的项目

```
/*****************************************************************
This program was produced by the
CodeWizardAVR V1.0.2.1 Stardard
Automatic Program Generator
©Copyright 1998-2001
Pavel Haiduc, HP InfoTech S.R.L.
http://infotech.ir.ro
e-mail:dhptechn@ir.ro, hpinfotech@xnet.ro

Project    :Chapter 4 Example Project
Version    : 1.0
Date       : 1/7/2002
Author     :Sally
Company    :Sally's Source Code
Comments   :
Atmega1163 Example Project

Chip type           :ATmega1163
Program type        :Application
Clock frequency     :4.000000 MHz
Memory model        :Small
Internal SRAM size  :1024
External SRAM size  :0
Data Stack size     :256
*****************************************************************/
```

```
#include <mega1163.h>

//External Interrupt 0 service routine
interrupt [EXT_INT0] void ext_int0_isr(void)
{
//Place you code here

}

 #pragma used+
 //UART Receiver buffer
 char rx_buffer[16];
 volatile unsigned char rx_wr_index, rx_rd_index ,rx_conter;
 //This flag is set on UART Receive buffer overflow
 bit rx_buffer__overflow
 #pragma used -
 // UART Receiver Interrupt service routine
 #pragma savereg -
interrupt [UART_RXC] void uart_rx_isr(void)
{
 #asm
        .equ_rx_buffer_size=16

        push r26
        in    r26,sreg
        push r26
        push r27
        push r30
        push r31
#endasm
#ifdef _MODEL_TINY
#asm
     ldi     r26 ,_rx_buffer
     lds     r30,_rx_wr_index
     add     r26,r30
     clr     r27
#endasm
#endif
#ifdef _MODEL_SMALL_
#asm
     ldi     r26 ,low(_rx_buffer)
     ldi     r27,high(_rx_buffer)
     lds     r30,_rx_wr_index
     clr     r31
     add     r26,r30
```

```
    adc    r27 ,r31
#endasm
#endif
#asm
    in     r30 ,udr
    st     x,r30
    lds    r30 ,_rx_wr_index
    inc    r30
    cpi    r30,_rx_buffer_size
    brlo   uart_rx_isr0
    clr    r30
_uart_rx_isr0
    sts    _rx_index ,r30
    lds    r30, _rx_counter
    inc    r30
    sts    _rx_counter ,r30
    cpi    r30, _rx  buffer_size+1
    brlo   _uart_rx_isr1
#endasm
rx_buffer_overflow=1
#asm
 _uart_rx_isr1
    pop r31
    pop r30
    pop r27
    pop r26
    out  sreg ,r26
    pop r26
#endasm
}
#pragma savereg+

#ifdef _DEBUG_TERMINAL_IO_
//get a character from the UART Recerver buffer
#define _ALTERNATE_GETCHAR_
#pragma warn-
#pragma used+
char getchar (void)
{
#asm
    lds   r30,_rx_counter
    tst   r30
    breg _getchar
#endasm
#ifdef _MODEL_TINY_
```

```
#asm
    ldi   r26 ,_rx_buffer
    lds   r30,_rx_rd_index
    add   r26,r30
    clr   r27
#endasm
#endif
#ifdef   _MODEL_SMALL
#asm
    ldi    r26 ,low(_rx_buffer)
    ldi    r27,high(_rx_buffer)
    lds    r30,_rx_rd_index
    clr    r31
    add    r26,r30
    adc    r27 ,r31
#endasm
#endif
#asm
    inc    r30
    cpi    r30,_rx_buffer_size
    brlo   _getchar0;
    clr    r30
_getchar0:
    sts    _rx_rd_index ,r30
    ld     r30,x
    cli
    lds    r26 ,_rx_counter
    dec    r26
    sts    _rx_counter ,r26
    sei
#endasm
}
#pragma used-
#ifdef _WARNINGS_ON_
#pragma warn+
#endif
#endif

#pragma used+
//UART Transmitter buffer
char tx_buffer[16]
volatile unsigned char tx_wr_index ,tx_rd_index,tx_counter;
#pragma used -
//UART Transmitter interrupt service routine
#pragma savereg-
```

```
interrupt [UART_TXC] void uart_tx_isr(void)
{
#asm
    . equ_ rx_buffer_size=16

      push r26
      in    r26,sreg
      push r26
      lds   r26,_tx_counter
      tst   r26
      breq  _uart_tx_isr1
      dec   r26
      sts   _tx_counter,r26
      push r27
      push  r30
      push  r31
#endasm
#ifdef _MODEL_TINY_
#asm
    ldi     r26, _tx_buffer
    lds     r30, _tx_wr_index
    add     r26,r30
    clr     r27
#endasm
#endif
#ifdef _MODEL_SMALL_
#asm
    ldi r26 ,low(_tx_buffer)
    ldi r27,high(_tx_buffer)
    lds r30,_tx_wr_index
    clr r31
    add r26,r30
    adc r27 ,r31
#endasm
#endif
#asm
    inc     r30
    cpi     r30,_tx_buffer_size
    brlo    uart_tx_isr0
    clr     r30
uart_tx_isr0:
    sts     _tx_index ,r30
    ld      r30,x
    out udr,r30
```

```
        pop     r31
        pop     r30
        pop     r27
uart_tx_isr1:
        pop     r26
        out     sreg ,r26
        pop     r26
#endasm
}
#pragma savereg+

#ifdef _DEBUG_TERMINAL_TO_
//Write a character to the UART Transmitter buffer
#define _ALTERNATE_PUTCHR_
#pragma warn-
#pragma used+
void putchar (char c)
{
#asm
        lds     r30 ,_tx_counter
        cpi     r30,_tx_buffer_size
        bush    _putchar
        tst     r30
        brne    _putchar00
        sbis    ucsra ,udre
        rjmp    _putchar00
        ld      r30,y
        out     udr,r30
        rjmp    _putchar03
_putchar00:
#endasm
#ifdef _MODEL_TINY_
#asm
        ldi     r26,_tx_buffer
        lds     r30,_tx_td_index]
        add     r26,r30
        clr     r27
#endasm
#endif
#ifdef_MODEL_SMALL_
#asm
        ldi r26 ,low(_tx_buffer)
        ldi r27,high(_tx_buffer)
```

```
    lds r30,_tx_rd_index
    clr r31
    add r26,r30
    adc r27 ,r31
#endasm
#endif
#asm
    inc     r30
    cpi     r30,_tx_buffer_size
    brlo    _putchar01
    clr     r30
_putchar01:
    sts     _tx_rd_index ,r30
    ld      r30,y
    st      x,r30
    cli
    lds     r30 ,_tx_counter
    inc     r30
    sts     _tx_counter ,r30
    sei
_putchar03:
#endasm
}
#pragma used–
#ifdef _WARNINGS_ON_
#pragma warn+
#endif
#endif

//Standard Input/Output functions
#include <stdio.h>

//Timer 0 overflow interrupt service routine
interrupt [TIM0_OVF] void timer0_ovf_isr(void)
{
//Place you code here

}

#define ADC_VREF_TYPE 0x40
//Read the ADC conversion result
unsigned int read_adc(unsigned char adc_input)
{
ADMUX=adc_input|ADC_VREF_TYPE;
```

```
ADCSR|=0x40;
while ((ADCSR&0x10)==0);
ADCSR|=0x10;
return ADCW;
}

//Declare you global variable here
void main(void)
{
//Declare you local variable here

//Reset Source checking
if (MCUSR&1)
  {
  //Power-on Reset
  MCUSR=0;
  //Place your code here

  }
else if (MCUSR&2)
   {
   //External Reset
   MCUSR=0;
   //Place your code here

   }
else if (MCUSR&4)
    {
    //Brown-Out Reset
    MCUSR=0;
    //Place your code here

    }
else
    {
    //Watch Reset
    MCUSR=0;
    //Place your code here

    }
//Input/Output Ports initialization
//Port A
```

```
PORTA=0xFA;
DDRA=0x0F;

//Port B
PORTB=0x00;
DDRB=0x00;

//Port C
PORTC=0x00;
DDRC=0x00;

//Port D
PORTD=0x00;
DDRD=0x00;

//Timer/Counter 0 initialization
//Clock source:System Clock
//Clock value : 500.00kHz
//Mode : Output Compare
//0C0 output : Disconnected
TRCR0=0x02;
TCCNT0=0x00;

//Timer/Counter 1 initialization
//Clock source:System Clock
//Clock value : Timer 1 Stopped
//Mode : Output Compare
//0C1A output : Disconn:
//0C1B output : Disconn:
//Noise Canceler : Off
//Input Capture on Failing Edge
TRCR1A=0x00;
TCCR1B=0x00;
TCNT1H=0x00;
TCNT1L=0x00;
OCR1AH=0x00;
OCR1AL=0x00;
OCR1BH=0x00;
OCR1BL=0x00;

//Timer/Counter 2 initialization
//Clock source:System Clock
```

```
//Clock value : Timer 2 Stopped
//Mode : Output Compare
//OC2 output : Disconnected
TCCR2=0x00;
ASSR=0x00;
TCNT2=0x00;
OCR2=0x00;

//External Interrupt(s) Initialization
//INT0 : On
//INT0 Mode : Falling Edge
//INT1 : Off
GIMSK=0x40;
MCUCR=0x02;
GIFR=0x40;

//Timer(s)/Counter(s) Interrupt(s) initialization
TIMSK=0x01;

//UART initialization
//Communication Parameters : 8 Data ,1 Stop , No Parity
//UART   Receiver:On
//UART   Transmitter : On
//UART   Baud rate : 9600
UCSRA=0x00;
USCRB=0xD8;
UBRR=0x19;

UBRRHI=0x00;

//Analog Comparator initialization
//Analog Comparator : Off
//Analog Comparator Input Capture by Timer/Counter 1 : Off
ACSR=0x80;
SFIOR=0x00;

//ADC initialization
//ADC Clock frequency : 250.000kHz
//ADC Voltage Reference : AVCC pin
ADMUX=ADC_VREF_TYPE;
ADCSR=0x84;

//Global enable interrupts
```

```
#asm("sei")

while(1)
      {
      //Place you code here
      };
}
```

4.8　终端工具

终端工具(Terminal Tool)用于调试嵌入式系统，它使用串行通信(RS-232、RS-422 和 RS-485)。选择 Tools | Terminal 命令或者在工具栏单击 Terminal 按钮可以打开 Terminal 窗口，如图 4-35 所示。

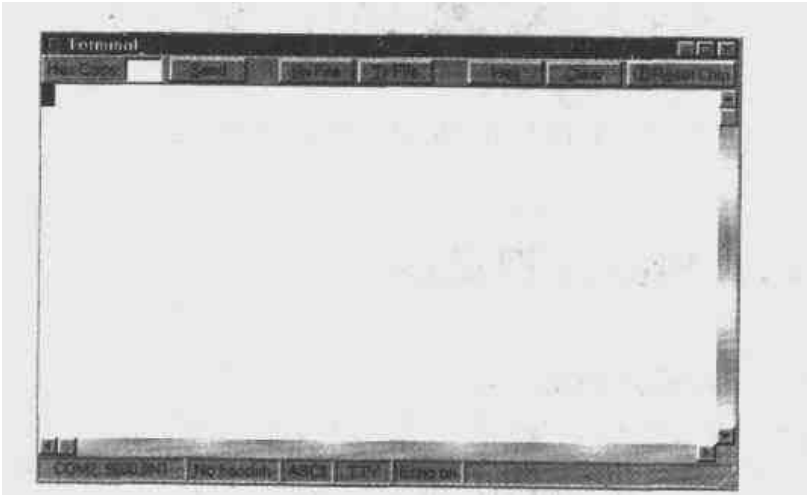


图 4-35　CodeVisionAVR 的 Terminal 窗口

接收到的字符以 ASCII 码或十六进制格式显示在 Terminal 窗口的主体中。字符的显示模式可通过单击 Hex/ASCII 按钮选择。单击 Rx File 按钮可将接收到的字符存入一个文件。

在 Terminal 窗口输入的任何字符都会通过 PC 串行端口发送。输入的字符可以使用退格键删除。单击 Send 按钮，Terminal 就会发送一个字符，其十六进制 ASCII 码值在 Hex Code 文本框指定。单击 Tx File 按钮并选择适当的文件可以发送一个文件。

单击 Reset Chip 按钮可以重新启动 STK200+/300、VTEC-ISP、DT006、ATCPU、Mega200 或者其他相似开发板上的 AVR 芯片。

在 Terminal 窗口的底部是一个状态栏，显示以下内容：

- 选择的 PC 通信端口
- 通信参数
- 握手模式
- 接收字符的显示模式
- 模拟终端的类型

- 发送的字符回显设置的状态

选择 Settings | Terminal 命令可以修改 Terminal 窗口的设置。图 4-36 所示为 Terminal Settings 对话框。在修改设置之前，必须关闭 Terminal 窗口。当前的设置显示在 Terminal 窗口底部的状态栏中。

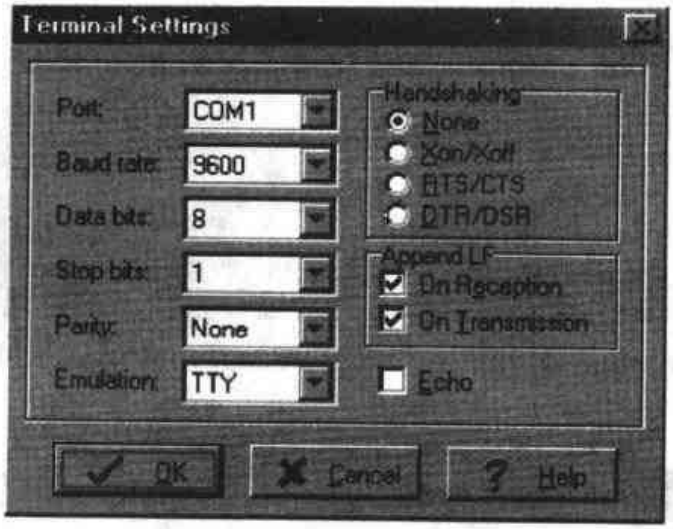


图 4-36　CodeVisionAVR 的 Terminal Settings 对话框

4.9　Atmel AVR Studio 调试器

软件仿真器和可嵌入线路的仿真器对于调试软件很有用，所以有时候被称作调试器(debugger)。大部分调试器都包含一些开始、停止和跟踪(step)方法。多数也都允许用户将寄存器和变量值设置为最大，甚至直接修改它们的值。它们对于找出程序为什么不按照希望的方式执行很有帮助。

CodeVisionAVR 可以与一些特定的调试器联合工作，如 Atmel AVR Studio debugger 3.5 或其以后版本。AVR Studio 是一个 Atmel AVR 系列的软件仿真器。它也支持可嵌入线路的仿真器，如 Atmel 的 ICE 200、ICE 50 和 JTAG ICE。AVR Studio 可以从 Atmel 的 Web 站点(http://www.atmel.com)免费得到。

虽然 AVR Studio 的详细操作不是本书讨论的内容，但是对每个程序员来说，懂一些基本操作是很有用的，例如，加载一个 C 文件用于调试、启动、停止或跟踪一个程序；设置和清除断点；查看、修改变量和机器状态等。如果需要高级操作，可以使用 AVR Studio 的帮助文件。

调试器通常也需要一些特殊格式的文件。这些文件包含 C 文件的格式、变量名、函数名及其他类似信息。AVR Studio 使用的文件的扩展名为.cof。CodeVisionAVR 本身可以创建这种类型的文件，可以从工具栏按钮启动 AVR Studio。

4.9.1　为 AVR Studio 新建一个 COFF 文件

在 AVR Studio 中进行 C 源代码级的调试需要 COFF 文件。选择 Project | Configure 命令打开 Configure Project 对话框，可以在 CodeVisionAVR 中创建这种类型的文件。切换到 C Compiler

选项卡，在右下角的 File Output Format(s)下拉列表框中选择 COFF、ROM、EEP 选项。单击 OK 关闭对话框，保存设置。

4.9.2　从 CodeVisionAVR 启动 AVR Studio

CodeVisionAVR 有两种启动调试器应用程序的方法，一是使用命令，一是使用工具栏按钮。不论使用哪一种方法，在启动调试器之前都需要指定调试器的路径。指定路径可以选择 Settings | Debugger 命令，在弹出的对话框单击 Browse 按钮，找到调试器并选择它即可。AVR Studio 的默认路径是：

C:\Program Files\Atmel\AVR Tools\AvrStudio4\AVRStudio.exe

然而，在您的系统中它可能放在另外一个地方。如果有必要，可以查找 AVRStudio.exe 定位该文件。选定文件以后，从 CodeVision AVR 启动 AVR Studio 就很简单了，可以选择 Tools | Debugger 命令，或者在工具栏中单击一个虫子图标的命令按钮，或者按 Shift+F3 快捷键。

4.9.3　打开文件进行调试

AVR Studio 调试的源文件称为目标文件(Object files)。选择 File | Open Objectfile 命令可以打开并浏览 CodeVisionAVR 创建的用于调试的.cof 文件。选择.cof 文件并单击 OK 按钮会打开 Select Device and Debug Platform 对话框。在左边可以选择用于调试的平台(本例中使用仿真器)，在右边可以选择模拟(或仿真)的设备(本例使用 AT90S8515)。单击 Finish 按钮关闭对话框并打开调试器窗口。

程序在打开时在调试器中以汇编程序显示。要以 C 源代码格式查看程序，必须运行程序然后停止它。然后，C 源代码就会出现在汇编程序窗口的后面。Run 按钮在工具栏的右上部，样子好像一页文字在右边有个箭头坚穿下来。Break 按钮位于 Run 按钮的右边，看上去像一个暂停键(两个垂直的条)。Debug 按钮也可以用来运行及中断程序。单击源代码窗口的标题栏使其出现在汇编程序窗口的前方。当程序中断时，在代码窗口的左边空白处会出现一个黄色的箭头，表示单击 Break 按钮时程序执行到的地方。

(听上去打开一个源代码文件像是一个笨拙的方法，确实是。编写本书时，AVR Studio 正在升级中，希望升级后的 AVR Studio 使用起来更加直观。)

由于已经找出并改正了错误的地方，所以在源文件改变时需要重新加载调试器文件。通常，AVR Studio 会检测到这一事件并提示您重新加载文件。照其提示做，或者简单地使用 File 菜单重新打开文件即可。

4.9.4　开始、中断和跟踪

在进入调试器这一非常重要的内容之前，需要回忆一下 AVR Studio 的一点内容：它有编辑和调试两种模式。在 Debug 菜单下有 Start Debugging 和 Stop Debugging 两个命令，说明现在使用的模式。同一时刻这两个命令只能有一个可用。如果 Start Debugging 可用(没有禁用变灰)，选择它可以进入调试模式。如果 Start Debugging 不可用，说明您已经在调试模式了。如果您不在调试模式，那么所有的调试命令都不可用。

启动（运行）、停止和单步执行调试器的按钮都位于 AVR Studio 工具栏的右上角。把鼠标指针放在其上停留片刻就可以显示每个按钮的功能。在 Debug 菜单也有相应的命令，可用的命令有 Run、Break、Reset、Show Next Statement、Step Into、Step Over、Step Out Of、Run to Cursor 和 AtuoStep。虽然它们很多都是很直观的，但是我们还是应该回忆一下它们的功能。表 4-1 给出了它们的用法。

表 4-1　调试器命令按钮

命　令	描　述	功 能 键
Run	开始或恢复程序的执行。程序一直运行，直到用户停止它或者遇到一个断点	F5
Break	停止或中断程序的执行。当执行被挂起时，所有窗口中的所有信息都会更新	Ctrl+F5
Reset	重设目标设备。如果正在执行，程序将会挂起。如果执行处于源代码模式，程序会一直运行，到遇到源代码语句(启动代码被执行)	Shift+F5
Show Next Statement	在实际的程序计数器的位置设置一个黄色标记，使窗口获得焦点	没有
Step Into	如果是在反汇编模式下，执行一行汇编程序；如果是在源代码模式，执行一行源代码	F11
Step Over	执行一条指令。如果指令是一个函数调用，函数跟挂起前一样执行	F10
Step Out Of	执行指令直至当前的函数执行完或遇到断点	Shift+F11
Run to Cursor0	执行指令直至程序遇到源文件中光标所在行的指令	Ctrl+F10
AutoStep	一次执行指令直至遇到一个断点或用户发出一个中断命令。指令间的延时由 Debug Options 菜单决定。在每条指令执行时，窗口中的所以信息都会更新	没有

4.9.5　设置和清除断点

多数调试器都有一个非常有用的工具用于设置和清除断点。所谓断点就是在程序中您希望程序暂停执行的地方。一旦暂停，就可以查看变量或寄存器的值；或者使用单步跟踪，以确定程序正在做什么。

在 AVR Studio 中设置和清除断点的方法如下：在想设置断点的代码行右击，会弹出一个含有 Insert Breakpoint 和 Remove Breakpoint 命令的快捷菜单；选择适当的任务，一个大红点就会出现在代码窗口的左边或者从代码窗口左边消失。

4.9.6　查看和修改寄存器和变量的值

在 View 菜单有 Watch、Memory Window 和 Register 3 个命令。使用这些命令可以弹出一个

窗口，该窗口显示变量、存储器的各部分以及相应的寄存器。程序运行期间显示在窗口中的这些信息不会更新。一旦程序被挂起，无论是使用断点还是用户发出一个中断命令，窗口中的所有信息都会更新。

Watch 窗口默认地显示在右下角。要添加或者删除查看的项目，可以在窗口上右击使用快捷菜单；或者在一个空 Name 域双击，然后输入需要监视的变量名。要改变变量的值，双击 Value 列然后输入新值即可。

Memory 窗口位于其他活动窗口的前端，用于查看各种存储器空间，包括 EEPROM、数据、I/O、程序和寄存器。还可以在这里直接查看正在被修改的存储器。双击某个值，可以输入一个新值对它。在顶部的 Address 文本框可以指定想查看的存储器空间的地址。

Registers0 到 Registers 30 有很多方法可以查看。使用 View | Registers 命令可以将它们一次全部显示出来。双击某个寄存器可以修改它的值。

4.9.7　查看和修改机器状态

有时候查看微处理器的确切状态时非常有用的。计数器正在做什么？堆栈指针指向哪里？I/O 端口的情况如何？AVR Studio 屏幕的左端是一个 Workspace 窗格。窗格的底部有 3 个选项卡：Project、I/O 和 Info。在 I/O 选项卡可以看到机器状态并对其进行修改。

有几项的左边有一个加号(+)。单击+展开列表。展开的 Processor 项有程序计数器，堆栈指针，循环计数器，X、Y、Z 寄存器，以及其他有用的小片信息。而展开的 I/O 项包括一系列微处理器的特性，如 I/O 端口、UART、SPI 及 Timer 的状态等；对于每一个状态同时还会显示与其相关的控制寄存器的状态。现在您就可以真正看到机器内部正在发生什么了。虽然已经将端口 C 的引脚 2 调得很高，但是它依然还是很低？您是否忘了将该引脚设置为输出？您的计时器没有在产生中断—— 它还在运行吗？

像其他查看处理器和存储器状态的窗口一样，这些窗口在调试器执行程序时不会更新。一旦调试器挂起，这些窗口中的信息就会更新。

4.10　本章小结

本章的内容涵盖了很多 CodeVisionAVR C 编译器和 IDE 的特性。IDE 控制整个项目，并且有一个功能强大的代码编辑器。IDE 可以访问编译器、汇编程序以及与它们相关的选项。它还提供其他一些工具，如芯片编程器、CodeWizardAVR 和终端工具等。

现在您应该已经能够在 CodeVisionAVR 新建项目、向项目中添加源文件，以及将其生成可执行文件。如果有一个合适的编程数据线，就能够使用内嵌的芯片编程器对目标设备编程。您还应该知道 CodeVisionAVR 提供的 CodeWizardAVR 代码生成器和终端工具的基本操作。

最后提到了 AVR Studio 调试器的一些基本用法，但是只是简单触及了这个工具功能的皮毛。您现在应该熟悉如何在 AVR Studio 中加载、启动、停止及跟踪程序了。还应该熟悉在程序中设置和清除断点，查看和修改变量及寄存器的值。建议您从 Atmel 的站点下载这个免费的程序，使用它提供的帮助文件自己亲身体验一下。

4.11　练习

1. 图 4-37 所示项目的名称是什么？列出属于该项目的源文件名(4.3 节)。

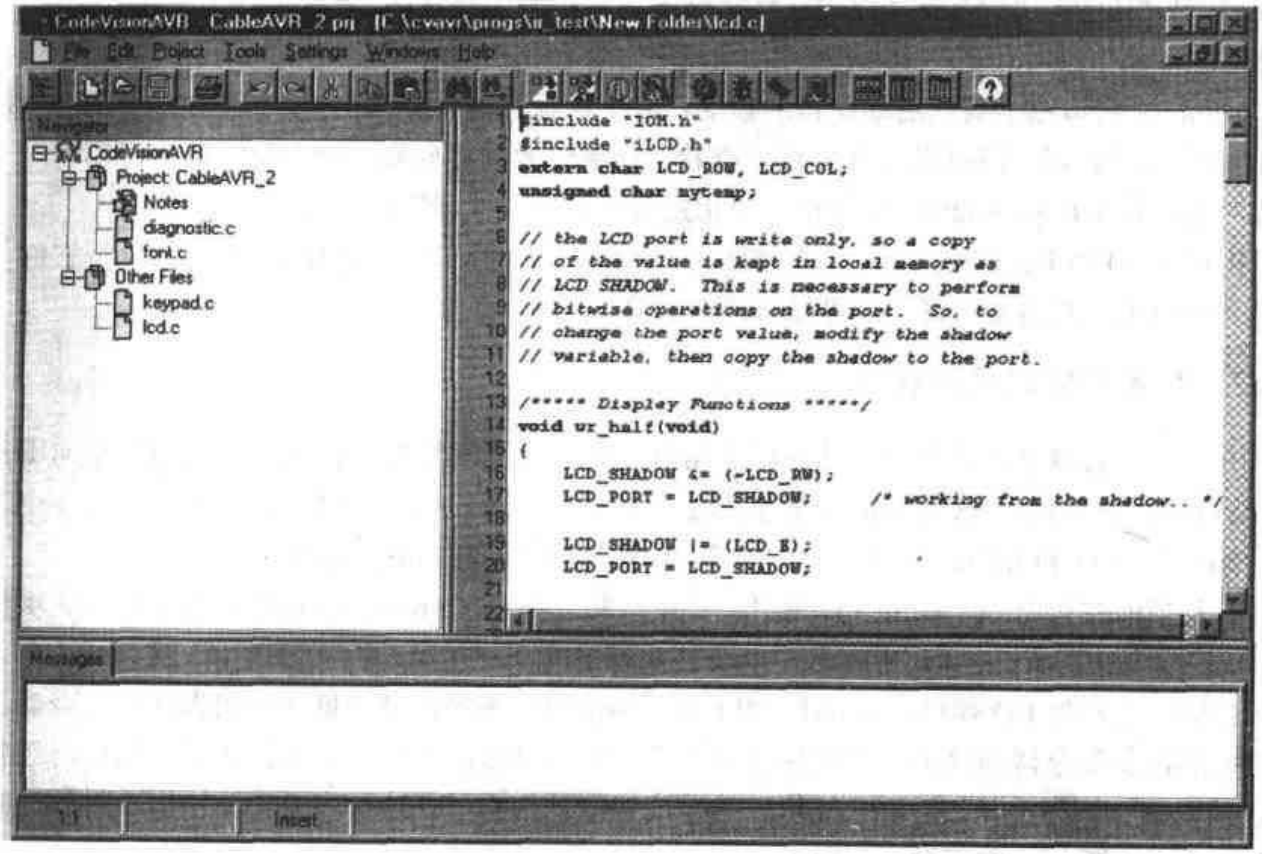


图 4-37　项目名

2. 在编译器选项中，要取得优化的代码速度和长度应该选用哪种存储器模式？其他存储器模式什么时候使用(4.4 节)？

3. 说明在编译之后两个显示错误和警告消息的地方(4.5 节)。

4. 描述编译一个项目和生成一个项目的区别(4.5 节)。

5. 将下面这段由图 4-38 中 CodeWizardAVR 的 Ports 选项卡为端口 B 生成的初始化代码补充完整(4.7 节)。

```
// Input / Output Ports Initialization
// PORT B
PORTB=0x__
_____=0x3E
```

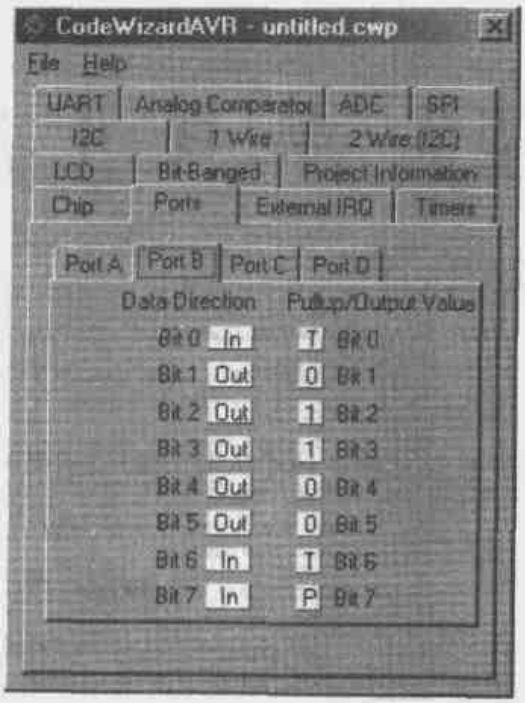


图 4-38　CodewizardAVR 的 Ports 选项卡

4.12　上机实习

1. 使用 CodeWizardAVR 新建一个项目以生成源代码外壳。项目的配置如下：

- UART：9600 波特，8 个数据位，一个停止位，没有校验位；支持发送和接收功能
- 端口 A：所有引脚用于输入
- 端口 B：所以引脚用于输出
- Timer 1：每 5ms 引发一次中断

2. 修改实习 1 中生成的项目，将端口 A 的值每秒钟向 UART 发送一次。不从中断例程发送值。

3. 使用 CodeVisionAVR 提供的终端工具以十六进制方式在程序中读取实习 2 中端口 A 的数据。

4. 以十六进制方式从终端工具向目标设备发送数据。将目标设备上 UART 接收的值送到端口 B。检查出现在端口 B 的值是否正确。

5. 使用 CodeWizardAVR 生成一个代码外壳，使其外部中断 0 能够在时钟下降沿打开 LED，外部中断 1 能够在时钟上升沿关闭 LED。完成之后测试程序。

第5章 项 目 开 发

5.1 本章目标

本章的惟一目的就是让您能够开发商业或个人使用的包括微控制器的电子项目。

5.2 引言

通过使用一种按序推进的方法，可以最有效地开发使用微控制器的电子产品。这种思想来自于许多成功的经验。

在本章中将描述这一过程，并通过一个实例项目来进一步演示这一过程是如何实现的。

5.3 概念开发阶段

每一个项目都基于一种来自各方面需要的思想或概念。有的是为了填补产品市场的空白，有的是为了改善生产过程，有的是为了满足教学要求，还有的仅仅是为了开发一种前所未有的东西。因为启动一个项目常常是为了满足某种需要，所以项目的初始描述有时也称为问题陈述或需求说明。

5.4 项目开发过程的步骤

开发一个项目需遵循如下步骤：

(1) 定义阶段
(2) 设计阶段
(3) 测试定义阶段
(4) 建立并测试原型硬件阶段
(5) 系统集成及软件开发阶段
(6) 系统测试阶段
(7) 庆祝阶段

5.4.1 定义阶段

项目定义的目的是要清楚地描述项目需要完成什么。这一步包括规定设备要做什么，通过

调查研究以保证项目确实可行，开发一系列完整描述项目功能的规格说明。在商业环境下，还需提交一份关于项目进展的正式建议书。这一步有时也称为可行性分析。

在定义阶段进行调查研究的目的是为了保证项目确实能够实现。调查的早期阶段可以得出一个大概的或宏观的框图，如图 5-1 所示。用某种通用术语来说，这一框图显示了组成最终项目的电路。如果调查研究中发现项目某一部分使用了尖端的技术或是对已有技术的一种全新的应用，就要模拟或建立并测试一些电路，以证明该电路能用于本项目。调查研究的目的就是让设计师有充分理由确信该项目确实能够运行。

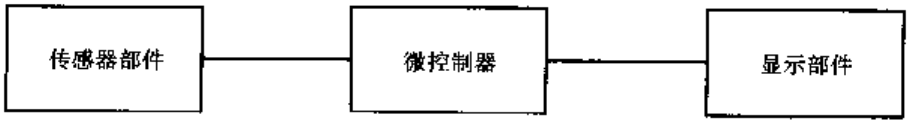


图 5-1 基本框图

定义阶段的最后，要得出一套完整的项目开发规格说明书，包括电气方面的规范，还要详细描述项目操作及人机界面的使用说明。尽管项目师们不愿在项目的人机交互方面花费时间，但是确保完成的项目不但能够运行而且便于操作也是十分重要的。在很多系统中，制作使用说明比起制作电气方面或功能方面的说明更加费力。

在商业环境下，项目建议用于对项目定义阶段进行总结。另外，还用于总结调查研究和可行性测试，以及提供书面形式的项目规格列表。该建议还应包括项目预算和完成项目的进度表。项目建议是为了让客户(或高层管理人员)相信：对这个项目的投资将会带来成功。图 5-2 所示为一个项目建议书的轮廓。

项目的标题

A. 项目描述

这一部分对项目作全面的描述，包括项目的总体功能，完成项目的原因和理论基础，以及项目完成时期望得到的成果。这一部分的目的是让读者理解项目的实质及其重要性。

B. 调研总结

总结在项目定义阶段完成的项目调研，包括对市场和(或)竞争产品的调研，对相似或相关项目或产品的调研，以及所有概念证明试验(proof-of-concept testing)完成时得出的结果等。这一部分的目的是为了表明已经对项目作了完整的调研，澄清该项目与其他已有项目的关系。

C. 框图

这一部分介绍了项目的总体宏观框图。紧接着讨论了关系到项目总体功能的各个模块的分功能。这一部分的目的是让读者相信项目能够成功完成。

D. 项目预算

这一部分详细地介绍了项目的预算，列出了所有的预期花费，包括与项目相关的人力和物力等。

E. 项目进度表

项目进度表显示了每个模块开始和完成的转折点，开发过程中每个阶段的开始和结束，项目中的每个回顾或报告点(review or reporting points)，以及项目最后的完成和提交。这一部分的目的是为了实际地表明项目完成所花的时间，让读者相信项目是经过完全规划的。

F. 附录

附录包括调研时涉及的参考书目，以及任何其他的因为太大而不能放入正文的信息。

图 5-2 建议书概要

定义阶段相对来说比较短，但却是一个十分重要的步骤。这一步骤约占整个项目开发时间的 10%～15%。这一步对确保完成的项目能按设计运行至关重要。

5.4.2 设计阶段

项目设计阶段的主要目的是用实际的电路图填充定义阶段得出的宏观框图，并用流程图的形式对项目软件进行规划。由于项目的硬件和软件是紧密相连的，硬件设计和软件设计必须并行完成。这一步的整个过程将占项目总投资时间的 40%～50%。不幸的是，下面的描述反映了许多项目师在项目定义阶段所使用的方法：

- 不管是从功能还是从有效开发来说，设计阶段比其他任何阶段都更能决定项目的成功与否(说到效率，请参考“按时和按预算”)
- 设计阶段最容易被一些项目师忽视，或者不喜欢，甚至憎恨。他们总是喜欢将硬件或程序编写与作研究和书面工作相提并论

项目设计包括对在项目定义阶段形成的项目如何工作的模糊想法的实现，用所有的方法来实现图表和软件流程图。项目设计阶段并不涉及任何硬件原型化和软件编写工作。项目设计过程的描述有硬件和软件之分。

硬件开发步骤：

(1) 从项目定义阶段形成的基本框图开始，这种框图很可能存在一些包含不止一种功能的图框。例如，如果要开发一种记录汽车参数(如速度、耗油量以及簧片组的偏差等)的设备，您可能会需要将一个名为传感器的模块直接连到微控制器。或者每个传感器有一个单独的模块用一条线连接到微控制器。也可能会有不止一个模块用于显示，如图 5-1 所示。这一步的目的是为了从模块的形式出发，接下来如果用正确的电路代替模块，就能得到所需的成果。

(2) 对相关组件和电路进行深入研究，以决定每个模块使用哪种电路和组件进行填充。当您决定将使用的电路和组件时，就自然将项目分成更小的模块，其目的是将模块分成您打算分别测试的最小单位。这些完成时，每个模块应该会列出其输入、输出电平和(或)信号。这些信号定义将用于对各个模块的测试。图 5-3 显示的仅是速度传感器(图 5-1 中所示的 3 个模块中的一个)被分成了合理的，可测试的模块。

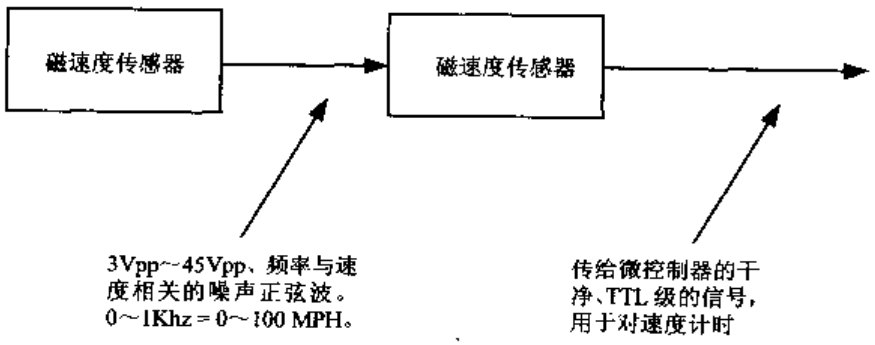


图 5-3　扩展传感器框图

(3) 以您的研究为基础，为每个包含电路的模块开发实际电路图，使用 CadencePSpice 或 MultiSIM 等电路仿真器来模拟您的电路操作，并按要求修改电路，直到仿真器正确地实现了模

块的目标。

(4) 用独立的电路模块创建项目的总电路图。使用剪切和粘贴技术或替换技术合成电路图。这样就可以避免将模块的电路重画在一起时引入错误。

软件开发步骤：

(1) 列出软件要完成的任务

(2) 定义任务的优先级，确定哪些是紧迫的要用中断技术来处理，哪些又不那么重要。用我们的例子来说，对速度脉冲计时可能被认为是紧迫的，需要一次中断，而刷新显示可能被认为是低优先级的，可以在处理器不处理紧急任务时来完成。

(3) 为每个中断函数创建软件流程图或略图。

(4) 为完成项目完整任务的程序代码创建流程图或略图。

当完整设计项目后，您将得到一张详细的框图(显示所有的图框和记号)，一套完整的电路图(虽然还没有在硬件上试验)，以及一套软件流程图或略图。

5.4.3 测试定义阶段

项目测试阶段的目的是通过项目测试说明的开发，确保项目满足所有规格和实现项目定义阶段提出的目标。该测试说明分成两部分：中间测试和最终测试(或系统测试)。

中间测试的测试说明是一系列的对项目的每个模块或模块组要实现的功能的测试。设计这些测试是为了确保项目被测试部分能够实现其功能。例如，测试一个滤波器，将需要指定频率范围，数据点数量，以及模块测试的预期结果。

最终测试说明是一个包含所有测试过程的文档。这些测试过程将用于证明项目原型能满足所有预期的规格说明。在商业环境中，最终测试说明在完成时很可能要经受客户和测试结果的考验。

5.4.4 建立和测试硬件原型阶段

这里要解释一下建立并测试硬件的过程。在这一步中要构造并完整测试项目原型的硬件、在每个可能的地方使用实际的输入输出设备。如果使用实际的硬件不大可行(例如在一种交通工具中的传感器)，那么就有必要使用电子测试设备和其他的一些电路，以便能够提供来自每个传感器的模拟输入来测试中期测试说明中定义的项目模块或部分。使设计好的信号仿真器随手可得，保证它们在系统集成和软件开发中可用。

这一步完成时，您应该确信来自每个传感器的准确输入信号被用到微控制器中，而来自微控制器的准确的输出信号按预定方式控制输出电路。这一步的目的是排除当程序没有与硬件正确交互时所发生的困境。通常在这种困境下开发者不能确定问题出在电路上还是软件上，但是因为在这一步中测试了电路，开发者就比较确信问题是出在软件上。

有时会发现测试微控制器本身所驱动的硬件部分比较容易。这样，在开发输出设备驱动功能时，就可以将精力放到下一步，也就是系统集成和软件开发阶段(比如编写和测试软件)。就不必把时间花在以后毫无用处的测试代码的开发上，而是致力于开发有用的函数。

5.4.5　系统集成和开发阶段

这一步是对应上述硬件步骤的软件步骤。编写和测试软件意味着开发和分别测试设计项目时画了流程图或略图的每个函数。使用简单的 main()函数可以分别测试每个单独的输入设备函数和输出设备函数。像前面提到的硬件步骤一样，这样就可以确信每个独立的输入输出函数正常工作。这一步同样是为了排除以后可能发生的与预期不符的不确定因素。

当各个函数正常工作时，以逐步推进的方式编写总体软件——为代码添加附加函数，调试代码，然后再添加更多的附加函数——直到使用实际或模拟输入时整个项目在功能上无误。

通常事先编写能运行输出设备的代码是很有用的，这样在测试输入设备时就可以用显示器等输出设备显示结果。作为调试和测试软件的一种方式，使用 CodeVisionAVR 的串行终端程序显示结果也是很有用的。

这一步的目标是拥有一个能使用实际或模拟的输入输出进行工作的完整项目。

5.4.6　系统测试阶段

系统测试阶段将项目真正投入使用。在此阶段，将根据以前得出的最终测试说明对项目进行测试，以确保项目能满足在项目定义阶段规定的规格。

对于一个商业项目，系统测试阶段还应包括对客户进行演示，与项目有关的知识产权的移交，IP 等。IP 包括与项目有关的完整的文档和记录，归客户所有。

5.4.7　庆祝阶段

成功的完成了一个项目总是值得庆祝的，不过这超出了本文的范围，在此不作讨论。

5.5　项目开发过程总结

表 5-1 总结了项目开发阶段。该表显示了每一步，预期结果，以及最终结果或每一步的成果。

表 5-1　项目开发过程总结

过 程 步 骤	期中预期/结果	成　　果
定义阶段	调查研究 基本框图	完整的项目规格说明 进度表 预算 提议
设计阶段	更多的调查研究 局部选择 电路模拟结果	最终的详细框图 电路框图 每个中断函数的流程图 项目测试流程图
测试定义阶段		每块或部分的测试说明 最终测试说明

(续表)

过程步骤	期中预期/结果	成果
建立和测试原型硬件阶段	各模块测试结果	能工作的硬件
系统集成及软件开发阶段	各个函数的测试结果	能工作的软件
系统测试阶段		完整的、功能完备的项目 项目文档
庆祝阶段		

5.6 示例项目：一个气象监测器

5.6.1 构思阶段

举个例子，假设我们是R-U风向标(Wind Vine)公司的项目师，公司最近正在寻求扩大产品供应渠道。某天，同我们WVRU公司的员工共进午餐，有人抱怨在这样冷的天气里穿得太少。这引发了一场有关当地天气预报的准确性和怎样通过自己观察天气决定穿多少衣服的讨论。

进一步的讨论引起大家对怎样的天气参数对估计天气有用的思考，最后用餐巾纸(很多项目通常都开始于一张鸡尾酒的餐巾纸和一只笔)拟出一个系统应有的特性。

设计一个气象监测器时，列出了如下特点：

- 便于安装
- 收集尽可能多的天气数据
- 费用低
- 结构简单

除了这些特点外，还考虑到气象监测器应由两部分组成：一个室内装置，一个室外装置。室外装置应收集尽可能少的参数，尽可能保持简单性。温度、风、降雨量和湿度等必须在室外测量，因为这些室外才有。而大气压在室内和室外基本上一样，因此在室内测量。

室内温度和相对湿度当然应该在室内测量。观测天气还有时间因素，我们可能想知道某个时候，甚至是更长时间的天气情况，例如什么时候有大风或低温，所以室内装置应该包括一个准确的时钟。

5.6.2 定义阶段

我们可以将构思阶段得出的需求进一步细化，修饰或定义在纸上列出的每一个构想。请注意，在此还没有可行性分析。也请记住，项目在本阶段的目标是开发系统的基本框图和一套完整的项目规格说明。

从扩展所列的构想开始：

(1) 便于安装

- 无线——900MHz，单向
- 室外装置无需外部电源：用太阳能/电池供电
- 室外装置电池能维持一个星期的下雨时间
- 安装简单(像在院子里安装信筒一样简单)
- 室内装置的液晶显示器(图像效果好，但不能太贵)
- 室内装置为用户提供的按钮，用于查看不同的参数及设置时间
- 现有的室内装置的电源以及时钟的备用电池

(2) 收集尽可能多的不同类型的天气数据

- 室内、室外温度
- 风速
- 风寒(Wind chill)，可以根据此参数决定如何穿衣
- 风向
- 降雨量
- 大气压
- 室内、室外的相对湿度(干净，但不能太贵)
- 露点(是否需要将折篷汽车停在室内)
- 室内装置的 RS-232 端口，用个人计算机监测天气

(3) 费用低

- 估计每种测量方法的花费
- 保持总体开销较低(使用尽可能少的部件)

(4) 结构简单

- 部件能方便地买到
- 简单的反馈系统，不要太花哨
- 风和降雨量的测量装置的结构要简单或用现有的

这张扩展的列表提供了我们工作范围的思维蓝图。扩展这张表时，形成了需求，指出了复杂度和花费情况。

在这种情况下，该过程同您根据个人爱好设计某样东西或者在公司的项目/市场会议上讨论下一代产品的开发问题是一回事。

1. 电气规范

现在有了需求列表，可以扩展定义来建立一个性能规范。在这里要对对应于每个功能或特性的参数赋值，指明性能和公差的范围。并不总是需要创建性能规范，在很多情况下，通过已有的相似的或有竞争性的产品也可以找到性能规范。表 5-2 所示为一个气象监测器的初步功能规范。

表 5-2 天气测量的初步规范

参 数	范 围	公 差
温度	−40° ~140° F	+/−1° F
湿度	10~90% RH	+/− 5% RH
大气压	28 ~ 32 inHg	+/−0.5 inHg
降雨量	0.00 ~99.9 英寸每天	+/−4%
月/年 降雨量	0.00 ~199.99 英寸	+/−4%
风速	2~120 MPH	+/− 2MPH
风向	0~360° (精确到 1°)	+/−7°
露点	−6° ~117° F	+/−2° F
风寒	−102° ~ 48° F	+/−2° F
时间	24 小时(精确到秒)	+/−1 分钟/月
日期	MM−DD−YYYY	N/A

2. 操作说明书

由于在气象站要用到人机接口，所以要制定操作说明书。

最好的方法就是画出草图，标明显示器会显示什么，以及写出一张表(甚至是一本操作手册)，说明如何选择按钮来操作设备。在很多情况下，早点做这一步可以大大缩短软件开发时间，因为有一张这样的表对心理有很好的影响。当表中的各项完成后，核对一下。气象监测器的操作说明书可以如下所示。

室外装置操作说明：

- 保持装置正常，模数转换器和计时器计数正常
- 测量温度
- 测量湿度
- 测量风速
- 测量风向
- 测量降雨量
- 测量电池/太阳能板电压
- 将信息传送到室内装置，大约每秒钟一次
- 只允许在传送时才打开射频 (radio frequency，RF) 组件的电源，以节省能量

室内装置操作说明：

- 测量温度
- 测量湿度
- 测量大气压
- 测电池/墙壁电源的电压
- 校对时钟
- 室外装置传来信息包时，收集信息包

- 使用 4×20 的 LCD 显示器，每屏显示尽可能多的信息，指示室内或室外及其他尽可能多的相关部件
- 提供 Units 按钮，允许用户改变显示数据的单位，如°F 到°C，mph 到 kph 等
- 提供 Select 按钮，允许用户选择每小时、每天、每月、每年的降雨量
- 提供 Set 按钮，允许用户设置时间和日期
- 仅当可以使用墙壁电源时允许液晶显示器使用背光
- 低压(low battery)指示 LED
  a. 室内装置电池电压低时，LED 闪烁
  b. 室外装置电池电压低时，LED 不变
- 提供射频通信活动 LED，用于指示室外装置传送信息

3. 基本框图

用当前已有的数据——需求表，扩展的定义，初始性能和操作说明——我们就可以画出一些系统的基本框图。图 5-4 和图 5-5 所示的框图显示了硬件结构的基本模块。包括供电电源，要测量的参数，将处理过的数据输出到何处，用什么方法，通过射频装置还是液晶显示器。

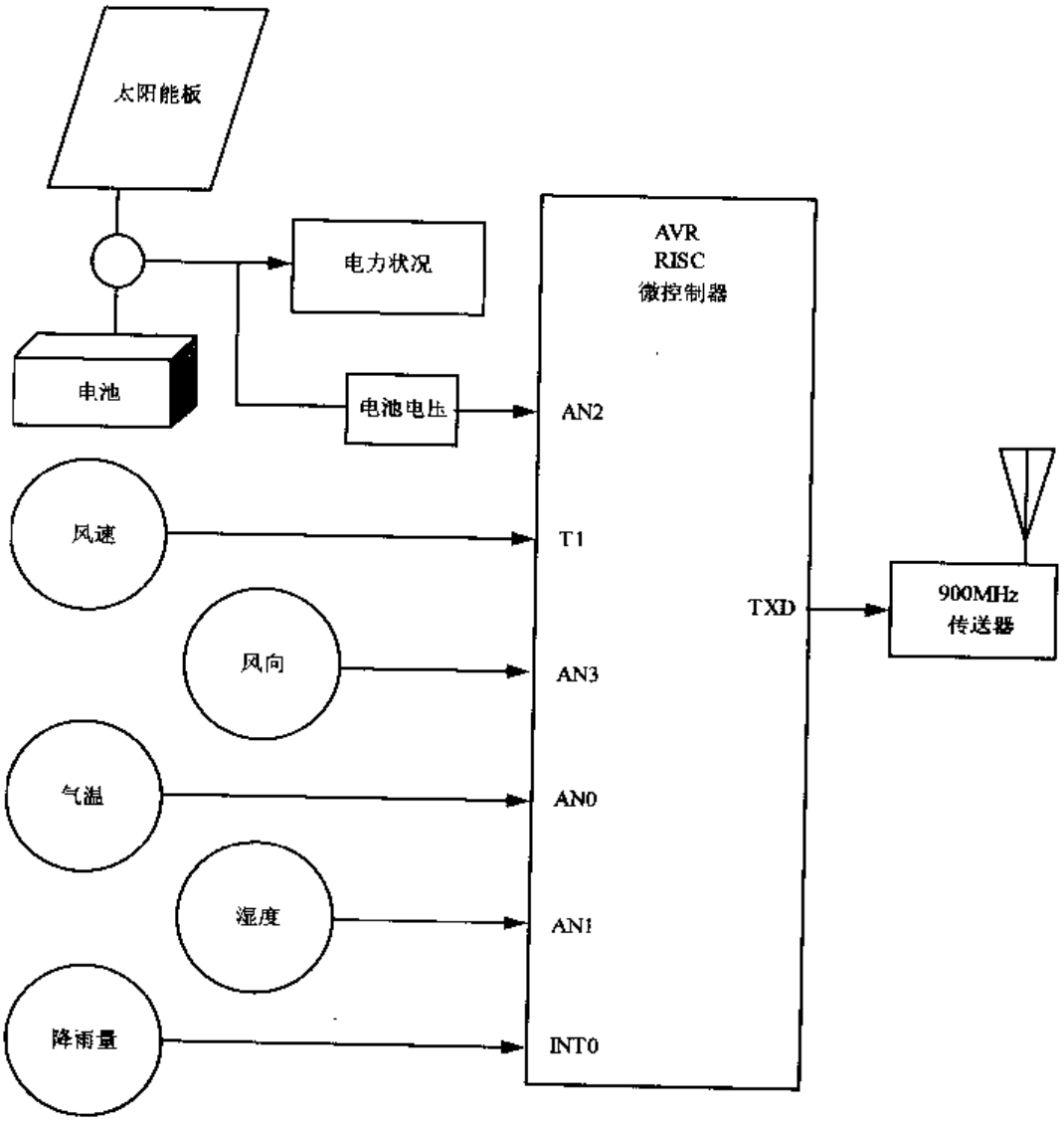


图 5-4　室外装置框图

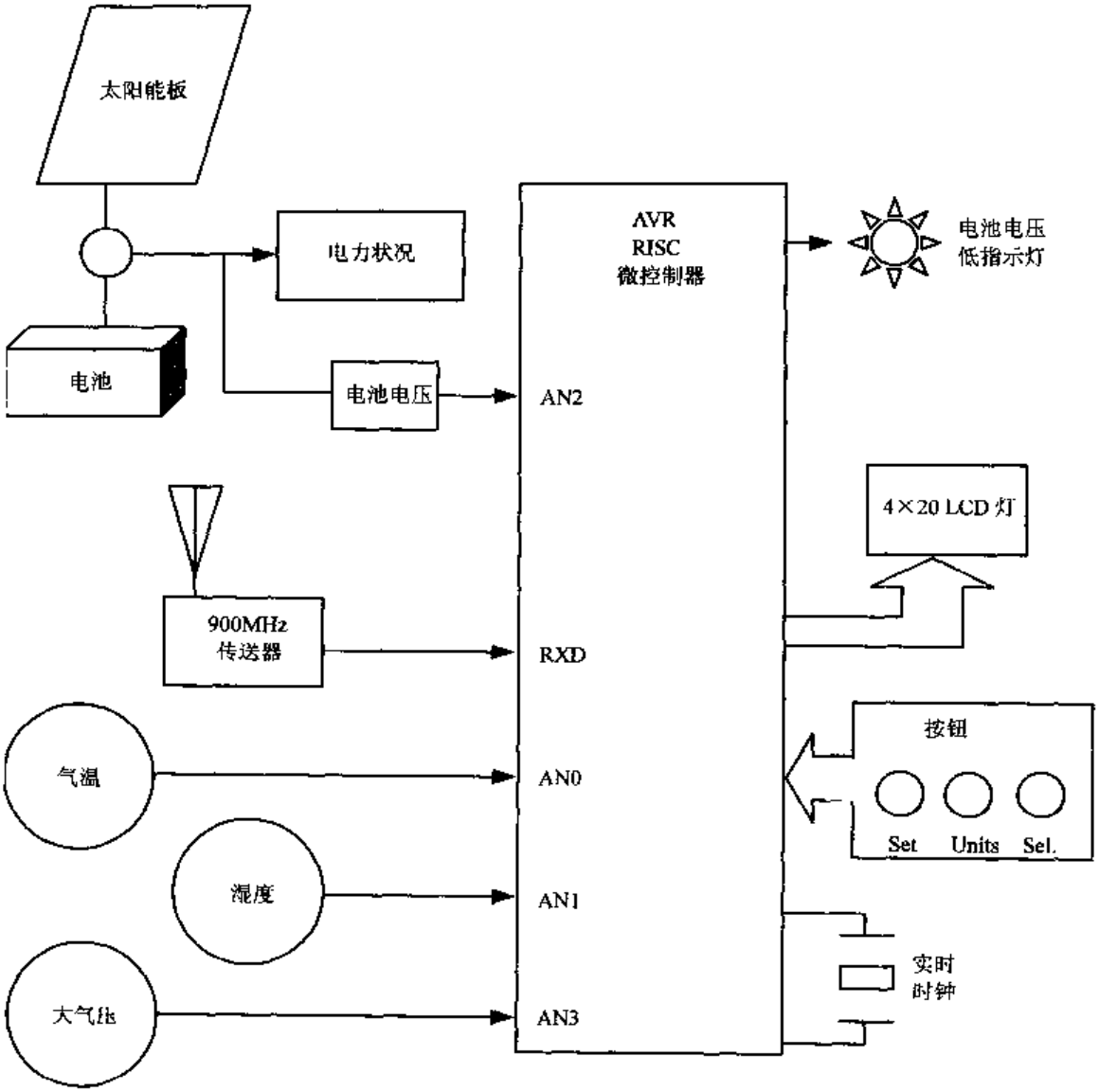


图 5-5 室内装置框图

这样的框图显示了在定义阶段可行性分析中所定义的细节。在框图中，通往微控制器的输入以该输入的硬件功能命名。下一节将描述如何每个测量和要求有哪些数据要输入到微处理器。在项目的这一阶段中，这样做很重要，因为专门输入(如模拟输入或计时器/计数器输入)的数字要受到微控制器的限制。

以框图为指导，就可以选择有足够 I/O 和特定性能的适当的微控制器。在接下来的设计过程中，就可以检验所选的微控制器是否符合项目需要，以确保选择了恰当的微控制器。

在原型开发中选择了 ATMega163。Mega163 与 AT90S8535 模式的内存是引脚对引脚兼容的，同时还与更大的内存 ATMega323 兼容。这使设计阶段有更大的自由空间。我们需要增加存储器模式的选项，因为我们不能确切知道每个部件中需要多大的代码空间。在项目结束和某些部件变得很慢时，如果代码可以存放在更小的存储空间或更廉价的组件中，那么这样的选择也是有用的。

另一种可能有更多明显选择性的是模数转换器 UART 和 32kHz 的实时时钟振荡驱动器。

5.6.3　测量方法在设计方面的考虑

定义阶段的最后一步是系统的可行性分析和健壮性检验。现在已明确规定了想要系统做些什么，那么，我们能实现这些期望吗，或者这些期望能否根据物理或电子方面限制作些修改？本小节描述了在进入设计阶段前确保项目可行的步骤。

然而，要将定义阶段的最后一步和设计阶段完全分离是比较困难。在很多情况下，有必要对设计阶段制定专门细致的描述，以确保能够检验我们的规格说明。所以，在谈到测量气象对物理和电子方面的要求时，本节在很多地方将跨越到设计阶段。

审查基本框图时，您可能会注意到有很多共同的元素。室内和室外都要求测量温度和湿度，他们的测量方法基本一样，因为都是用模拟电压表示。风速，风向和降雨量只在室外测量。室内室外的大气压基本一样，因此在条件更好的室内测量。

系统中的每个参数可以用不同的方式确定它们的值。某些情况下，一个参数可能有几种测量方法，考虑到性能、费用来选择适当的方法变得越来越重要。接下来将讨论设计方面对每个被测参数的考虑。

一旦选定了一种测量方法，就要在总体范围和各种分辨率下检验它。当您处理电压和 ADC 时，ADC 的分辨率是步因子(pacing factor)。当脉冲被送入一个计数器时，计数器的大小和计数比率取决于读数器的范围和分辨率。

在这个气象监测器项目中，温度、湿度、电池电压和大气压都是数模转换器要处理的电压。在设计阶段，这些数据被计算出来以证明一切都没有超出范围并满足规格说明。不至于风速和降雨量呈直线形式。

1. 温度

通过使用一个 PTC(正温度系数)或 NTC(负温度系数)电阻装置或热敏电阻就可以测量温度。这些设备可以通过适当的调节产生一些大范围的值，而不需要专门放大或调节。

还有很多其他的温度传感器，包括双金属热电耦，温度补偿二极管，以及电阻温度设备(RTD)。还有集成电路，如 National Semiconductor(国家半导体)LM35，这是已标定的固定的测量设备，可以在每∞C 传送特定数量的电压。例如，LM35 输出 10mV/∞C。然而，这些设备费用较高或需要额外的电路，使它用起来费用更高。

为我们的设计选择一个玻璃管 NTC 热敏电阻来作温度检测。图 5-6 显示了一个典型的 NTC 热敏电阻的温度和电阻的关系。

由于热敏电阻在每摄氏度的温度变化中有一定比率的电阻变化，使用一个电阻分压器可以对温度进行合理的线性测量。分压器的输出电压范围可以限定成与 ADC 的电压范围匹配，为预期的测量范围提供最好的解决方案。这就是 NTC 设备比 LM35 等集成电路优越的地方。要取得 NTC 一样的效果，LM35 还需要添加电路、脉冲等额外开销。虽然 NTC/电阻器分压器的输出电压可能不完全呈线性，但可以用智能软件和查找表来修改不满意的数据。

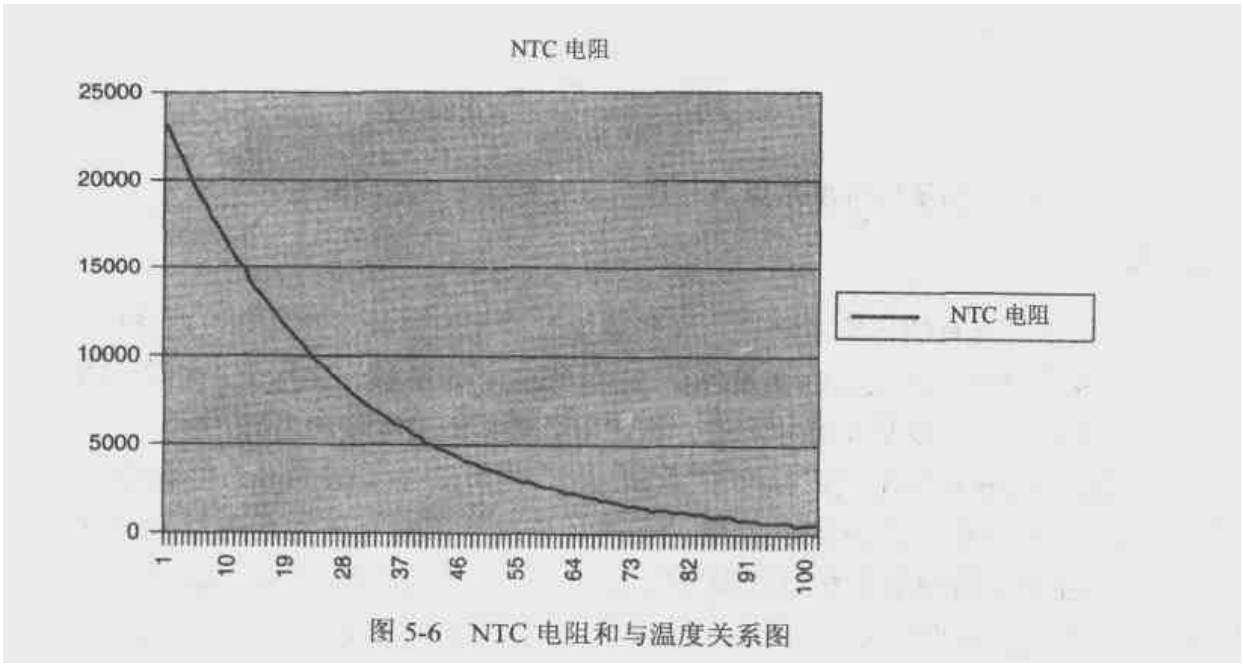


图 5-6 NTC 电阻和与温度关系图

作为整个 AVR 系列中的一员，Mega163 处理器更倾向于使用一个带有相当低的输出阻抗(约 5 kohm 或更少)的电路作为 ADC 的输入。我们用一个 4.7K 的电阻来做一个带 2K 热敏电阻(在 25°C 或 77°F 下的大致电阻)的分压器。温度每变化 1°C，选定的热敏电阻的值就变化 3.83%(热敏电阻说明书的参考数据)。这为 ADC 提供了从－40°F 下的 0.84V 到 140°F 下的 4.5V 的大致范围。由于随着温度的变化热敏电阻会有负阻抗，它被放到分压器的最上面。这就允许 ADC 的读数就随着温度的升高而增加。图 5-7 显示了阻抗分压器的输出电压和热敏电阻装置的温度之间的关系。

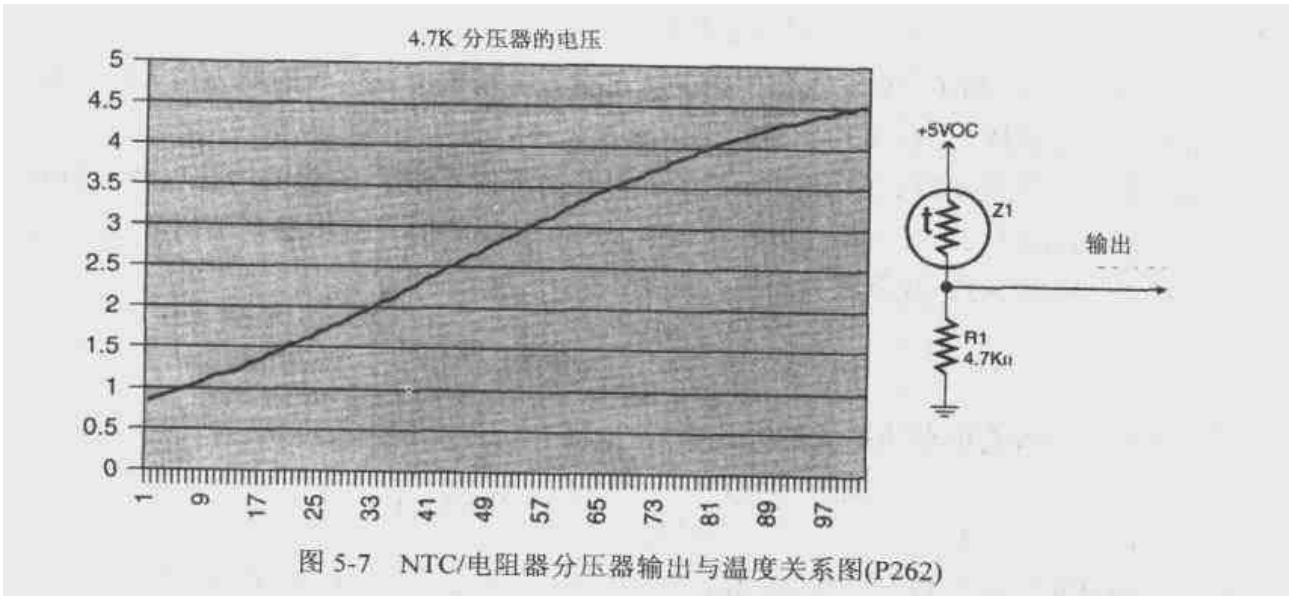


图 5-7 NTC/电阻器分压器输出与温度关系图(P262)

ADC 的分辨率是 10 位二进制数(或十进制的 1024)，这就意味着每 1° F 对应着 ADC 上的十进制数 4.16。通过算式

$$\frac{4.5V - 0.84V}{5V} \times 1024 = 749.56$$

得到这个数，这是 - 40° F~140° F 的范围。这意味着在 ADC 的每个计数点有大约 0.240°F

的分辨率。算式如下：

$$\frac{140°-F(-40°F)}{749.56}=0.240°F$$

这个分辨率说明测量精确度足以满足项目规定的±1°F 温度偏差。

2. 大气压

气压可以使用已有的传感器测量。这些装置有很大的值域，并且可以选择性调整，无需专门的放大器或调整器。天气压的测量以英寸贡柱(inHg)为单位。实际上，相似气象站的研究表明，比较典型的完整范围是 4 英寸贡柱。这就需要测量的灵敏度相当高。

气压通常用 3 种转换形式：标准压强(gauge)，相对压强(differential)和绝对压强(absolute)。标准压强(psig)和相对压强(psid)转换器，在两种情况下都是有区别地测量压强(两种压强的差别)。标准压强换能器测量是相对周围空气而言的——尤其是要设计成确保没有大气压算入读数。相对压强转换器测量对一个参考压强的相对值，这样两端输入的气压就可以相互抵消。绝对压强测量对真空的相对压强(0 psia)，这样被读到的气压就包括了大气压的作用，以适合于项目。

SemSym ASCX30AN 是一个 0~30 psia(磅/每平方英寸，单位)的压强转换器。在气象监测器中，只需将启动传感器并将其放在空气中，这就意味着只测大气压。大气压的范围非常小，通常介于 28～32 英寸贡柱之间，换算成压强是 13.75～15.72psia。说明书表明，ASCX30AN 的灵敏度为 0.15V/psi。这意味着从 13.75～15.72psia 的实际输出范围是直流电压的 2.06～2.36V。

这可不大出色。因此，在应用到微控制器的 ADC 之前，要对转换器的信号进行放大和直流抵消。将传感器的电压范围放大 16 倍到可用的水平：

$$(2.36V-2.06V)\times 16=(0.03\times 16)=4.8V$$

4.8V 的范围用了 ADC 5V 输入范围内的大部分，将精确测量信号的能力最大化。然而这样做时，必须在用于 ADC 之前将传感器的电压减去 2.36V 的直流偏移电压。

这里使用了一个差动放大器(differential amplifier)将信号作了 16 倍的增益以及－2.06V 的偏移。这使 ADC 相对于 28～32 英寸贡柱获得了从 0.0～4.8V 的直流电压范围的信号。根据测量范围，ADC 在 28.00～32.00 范围内的总范围是：

$$\frac{4.8V-0.0V}{5V}\times 1024=983.04$$

这意味着我们拥有的分辨率大约为：

$$\frac{32inHg-28inHg}{983.04}=0.0041inHg$$

0.0041 英寸贡柱对应每个 ADC 计数，这就更加符合规定的±0.5 inHg 精确度。

3. 湿度

湿度同样可以用现有的转换器测量。为了精确，这些设备有热补偿，并且很贵。更省钱的方法是使用一个湿度传感电容，但这要花很多时间来补偿和处理设备的输入，以得到精确的湿度测量值。

在这个气象监测器中，使用一个现有的温度补偿恒湿器(humidistat)来测湿度，这样做是为

了简化工具。这不是最便宜的解决方案，但便于集成，而且很准确。设计中采用的设备是一个Honeywell(宏开关)HIH-3602-L。该设备输出处理过的对应于0~100%湿度的0.8V～4V的直流电压。该输出直接作为Mega163处理器的ADC输入。ADC输出的总的湿度范围是

$$\frac{4.0V-0.8V}{5V}\times 1024 = 655.36$$

使用这一数据，确定了湿度测量的最小单位是：

$$\frac{100\%-0\%}{655.36} = 0.152\%$$

精确到0.152%的湿度的测量值绰绰有余地符合了电子规格RH±5%。

4. 风速

测风速的典型方法是用一套旋转杯或一种名为风速表(anemometer)的扇叶片。基本上，当起风时，风速表就以一个与风速直接相关的速度旋转。用于测量某物转速的设备叫做转速表(tachometer)。测量风速时可能要用到好几种类型的转速表。

常用的一种转速表看上去像一个小马达或发电机。转动时，它就生成一个与其转速相关的直流电压，本例中，使用微控制器的ADC可以检测到输出电压，并从中得出风速。

另一种的转速表是数字转速表。一个典型的数字转速表通常生成一串与转速有关的脉冲，该脉冲的频率将被检测到并转换为rpm。很多情况下，一个数字转速表比起模拟转速表来是个更便宜的选择。它可以轻易的由一个分时磁盘和一个光电断续器(photo-interrupt)或光学耦合器(opto-coupler)构成。用光编码器的形式也是一种现存的解决方案。由于电枢的重量，没有了磁力，惯性也更小了，数字转速表比模拟转速表有更大的优越性，因为通常它对系统产生的阻力更小。

对于我们的气象监测器而言，选择购买一种转速表。这一选择是仅仅是基于机械学方面考虑的(参考图5-8)。如果是一个特别方便的个人，那么您可以做一个具有同样效果而花费更少的装备。

这种特殊的部件使用磁干簧继电器(magnetic reed-switch)来对速度编码。一般情况下，一个小的金属开关被装在基板上，一块磁铁被装在转速表上。当转速表转动时，磁铁随之将开关闭合断开。虽然从机械上来说，这是一种简便的方法，但是它失去了一些电子品质。因为开关在本质上不是电子的，开关存在一个很大的反弹势能。开关反弹是：当它们挤在一起时，就会像橡皮球一样在触点有反弹，产生一些过渡信号。如果这个信号不经电路防反弹或过滤处理，就直接送入到微控制器的计时器输入端，就会得出错误的结果，因为计时器将会收到开关实际的转换数目的3～10倍的数据。因此，需要一些防反弹电路来提供准确的风速测量，如图5-10(室外装置图)所示。

该设备的规格说明显示它会以0.9mph的速度测量风速，与真实的风速相比可以精确到1.1%，在我们确定的最高风速下(120mph)，可以精确到120的1.1%，即1.32mph，符合要求的±2mph。

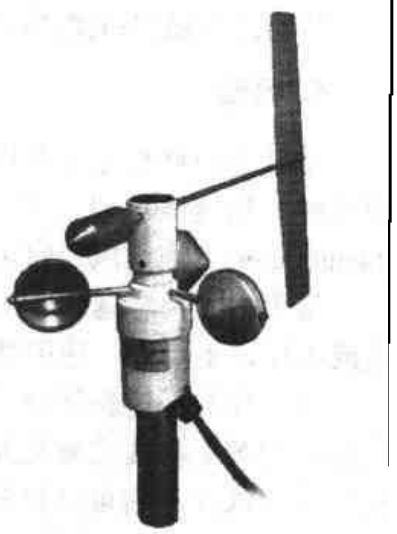
在本项目中，风速输入是一个脉冲串。为计算风速我们设想从计时器的一个输入端计算脉冲数，得出每秒的总脉冲数。为了检验我们的测量能力，我们需要知道对应一定的风速，转速表的扇叶或转杯会转多快。现在，我们可以假定对应1英寸直线空气流转速表转动0.15转，这就意味着1英尺/秒转化成1.8转/秒，也就是108转/分钟(rpm)。每小时1英里的风速相当于1.46

英尺/秒，对应的转速表的转速为 158 rpm。对于 100 英里/小时的风速，转速表以 15 768rpm 的速度转动。

WSD-PB 双重风速/风向传感器

034 型号的风传感器将风速和风向的测量组合成一种单个的遥感部件。034A 风传感器只要几分钟就可以安装完备，可以在恶劣的环境下长期连续地进行准确的气象检测。

- 用一个单一的传感器测量风速和风向
- 耐用的铝质和钢质的无变型结构
- 低起始点
- 低电耗



操作：

034A 使用的温度范围是－30°C～＋70°C，最高风速 155mph。风速使用一种封装的干簧继电器(reed switch)测得，这种干簧继电器根据风速产生相应频率的脉冲串。铝杯被双重复用。风向通过使用一个直接连到精确分压器上的铝风向标测得。电压随风向的变化而变化，这样一来该装置就成了既是数字的又是模拟的测量系统。

结构：

高质量的铝和无形变的钢材料的组件，高亮度白色粉末涂层保证了长时间的使用寿命，而且不需要很多的维护。电缆可以长达数百英寸，而且不会影响测量效果。

WSD-PB 规格说明

风速

范围：0~155 mph(0－69 m/s)

启动点：0.9 mph (0.4m/s)

精确度 < 22.7 mph: 0.25mph(0.1m/s)

精确度 > 22.7 mph: 真实值的 ± 1.1%

风向

范围

机械： 0~360°

电子：0~356°

启动点： 0.9 mph(0.4 m/s)

精确度： ± 4°

阻尼率：0.25 标准(.4~.6 可选)

分辨率：0.5°

输出信号

风速：触点关闭脉冲

风向：电位计输出(0~10 kohms)

温度范围： －30°～＋70°

传感器重量：1 lb 12.5 oz(0.81kg)

图 5-8　WSD-PB 复用风速/风向传感器

如果在一个转速表每转一次就发送 4 个脉冲的系统中放置一个编码器，那么，对于 1 英里/小时的风每秒钟将产生 10.5 个脉冲(PPS 或 Hertz)。100 英里/小时的风将产生 1051.2 PPS 的脉冲串。如果以每秒一次的频率对信号抽样，用一个 8 位计数器最大可以测量 255PPS 的风速。对此进行逆运算，就可以发现转速表在最大风速下的转速如下：

$$\frac{255\text{脉冲}}{4\text{脉冲}/\text{转}} = 63.75\text{转}/\text{秒}$$

如果转速表每转 0.15 转对应于空气 1 英寸的流速，那么就可以从转速表的转速换算出空气的流速：

$$\frac{63.75\text{转}/\text{秒}}{0.15\text{转}/\text{英寸}} \times \frac{1\text{英尺}}{12\text{英寸}/\text{英尺}} = 35.41\text{英尺}/\text{秒}$$

$$\frac{35.41\text{英尺}/\text{秒}}{5280\text{英尺}/\text{英里}} = 0.0060707\text{英里}/\text{秒}$$

$$0.0060707\text{英里}/\text{秒} \times 3600\text{秒}/\text{小时} = 24.15\text{英里}/\text{小时}$$

我们最初规格的最大测量值是 120mph，这是极强的风了。复核显示，要避免计数器在测量过程中翻转，必须每秒抽样 5 次(或每 200 毫秒抽样一次)。同时还可以看出，在风速和 rpm 之间转速表有很大的变化量，因而在设计方面仍是安全可靠的。

如果考虑装置的全部因素会发现，在 120mph 的风速下每 200 毫秒抽样一次还不够，而需要以更快的频率抽样，或许是每 150 毫秒一次。相反，如果发现风速和 rpm 之间的关系更小些，就可以以更低的频率抽样，这样就减少了系统开销。

另一种提高测量能力的简单方法是使用更大的计数器。Timer1 是一个 16 位的计数器，比原先大了 256 倍。这意味着我们以 1/50(或每 10 秒一次)的频率抽样就够了。如果抽样频率是每秒一次，那么在 Timer1 得到的数字是用 8 位计数器得到的 5 倍。

5. 风向

风向通常由风向标或气象风向标测定，在此专门使用 R－U 风向标。气象风向标通常是一根细轴，有点像一支箭，在水平面上转动。风向标可以自由转动。标尾被风带动，形成阻力，风向标的标头就指明了风的初始方向。判断风向就是对标头所指的地方进行转换，从 0°～360°，0°表示北方，90°表示东方，以此类推，这样就可以以电子的方式读取。

这可以用一个光电转换器实现。光电转换器是一种可连续转动的电位器，或一种简单机制的开关编码器。光学和开关编码器通常用作相对测量。这意味着先确立一个起始点或零点，而编码器则指出到起始点的距离。另外还有绝对型的编码器，如 Gray Scale 编码器；这些编码器为一个单周内的转动位置提供不同的编码，其优点是可以限制分辨率。一个 4-信道的 Gray Scale 编码器以一个 4 位字表示 360°以内的转动位置。这就将可分辨的位置数限定在 16，或者每两个可分辨位置之间有 22.5°的转角。

使用一个可连续转动的电位器可以得到更高的分辨率。电位器(如众所周知的变阻器或可变电阻)可以有效地提供绝对位置，而且可以无限划分。于是 ADC 或其他用于读值的设备的最小刻度就成为了对位置编码的限制因素。电位器通常不是可连续转动的设备，但一般是单转或 10 转的设备。如果您想要做一个自己的风向传感器，就有必要考虑安装一个电位器。

图 5-9 显示的森林技术系统公司(The Forest Tehnology Systems.Inc)的 WSD-PB 复用风速/风向传感器使用了一个 10kohm 的可连续转动电位计。注意，在规格说明里的测量范围是 360°，而可用的范围是 356°。这是由于在一个单转电位计中有些不能到达到小死点。在安装气象监测器时，这些死点可以用于确定北方或南方的参考点。

该设备的输出可以直接用作 Mega163 处理器的 ADC 的输入。ADC 是二进制的 10 位的(或十进制的 1024)，传感器的输出电压是 0V～5V 的直流。这就意味着每 1°反应在 ADC 内就是 2.876(十进制)。可以通过以下算式得到：

$$\frac{1024}{365^{\circ}} = 2.876/\text{度}$$

总范围是 0° ～356° 。也就是说，对于 ADC 每个计数点就有约 0.347°的偏转。效果远远好于规定的精确度±7°。

6. 降雨量

降雨量用立方英寸度量。典型的降雨量测量器由一个漏斗和一个圆柱形量筒组成。根据美国农场和家庭年鉴的 Ref031.02A，降雨量的 1 英寸定义如下：

每英亩的土地包含 43 560 平方英寸，这样的话在一英亩的土地上一英尺的降雨量等于 6 272 640 立方英寸的水，相当于 3 630 立方英尺。一立方英尺的水大约重 62.4 磅(因密度而异)；在每亩土地上均匀地铺上 1 英寸高的水，则水重大约是 226 512 磅(约 113 短吨)。一 U.S.加仑的纯水约重 8.345 磅，因而一英亩土地上高为英寸的降水就是 27 143 加仑的降水。

测量降雨量的最困难的部分是长时间地收集并读取大量的雨水。这里有一些常用的方法。

一种方法是使用一个开有尖嘴的漏斗，它可以形成均匀的水滴。当以一个随机速率将雨水充入漏斗时，尖嘴就产生均匀水滴，水滴的数量可以长时间计数。这样就可以转换计算出体积，体积可以转换成降雨量的英寸数。这种方法的缺点是填充漏斗的速率难于决定。如果漏斗的体积在水滴出并被计数之前不足以盛下所有的水，就可能产生不正确的读数。

另外一种方法是使用浆轮(paddle wheel)。轮子上有一些已知容量的杯子，雨水经一个漏斗流入杯中。当杯子满时，轮子就翻转。轮子转动的周数指示了降雨率、降雨量以及轮子翻转时抽样的清空频率。

浆轮的一种简化版本是一种跷跷板形。两者的构想基本一致，只是后者只有两个杯子，当其中一个杯子满时，它就将雨水卸空，并带动另外一个杯子到它原先的位置，以便在漏斗下收集抽样的雨水。当第 2 个杯子满时，它也将雨水卸空，将第 1 个杯子带到原来位置。这种方法在电子化雨水测量中大概是最常见的，因为它用最简单的机械装置就可以收集到所有必需的信息。每次当一个杯子从一个地方翻转到另一个地方，卸空的雨水就被计数，并以此产生电子信号。

所有这些方法中，都要用到一个光学或机械形式的编码器来获取信息，以供分析。信息量取决于抽样的大小和符合最少抽样的抽样频率。这里为了简化机械设计，我们又一次要使用从森林技术系统公司购买的传感器，如图 5-9 所示。就像在介绍风速传感器时所说的一样，如果更倾向于机械化，那么就可以找到一个更省钱的办法。

像风速表一样，本装置也使用了一个磁干簧继电器来对降雨量编码。基本上都是一个小金

属开关装在底座，一块磁铁装在跷跷板上。当跷跷板来回翻转时，干簧继电器的触点就随之关上或打开。由于干簧继电器是机械装置，在风速表中同样存在开关反弹现象。因此需要一些额外电路来防止出现错误的读数。

FTS RG-T 雨水测量器

RG-T 雨水测量器的特点是使用一个翻转桶，为了保持平衡和清晰的刻度，翻转桶要经过化学磨光。翻转桶可在一个轴承上低摩擦地、灵敏地翻转，这样就可以精确地测量雨水量。

为保证一个开关每当翻转桶翻转时都会闭合，还需要对精确校正。翻转桶、轴承和磁干簧继电器固定在结实的铝片上，再以坚固的铝板为底座。刻度校正螺丝被保护起来，以防使用时刻度混乱。

显示器放在漏斗中的一盘水上，盘子出口被封闭，以防蜘蛛和昆虫爬入。连到圆形防水插座连接器上的电缆被不锈钢保护起来以防腐蚀。内置水平仪与精确校正水平仪连在一起，加上底座，这使 RG-T 安装起来又快又简单。

FTS 雨水测量器受到不少室外专家的青睐，包括那些拥有丰富野外经历的专家们。他们认为它是世界上最好的翻转桶式雨水测量器之一。因为它结构合理，精确度高，且刻度不易磨损。

重新设计这种品质的产品，需要经历一次精确的浇铸过程，使我们精确地浇铸出器件的内部结构，且必须是一个完整的整体，而不是几个分散的部分。产品应该更加精确，更能抗震，易于复制。

RG-T 说明

分辨率：0.01 in per tip(0.245 mm)

精确度：±2% 在 2/小时(50mm)

圆柱体规格：10.5 in×8 (直径 25.7 cm×20.3 cm)

底盘直径：12 in×12 in (30.5 cm×30.5 cm)

电缆长度：20 ft (6.1 m)

重量：10 lb (4.5 kg)

图 5-9 RG-T 雨水测量器

不管您是购买这个测量系统，还是自己动手做，不管您多么小心地做漏斗，抽样杯，轮子或跷跷板系统，在到进水之前您无法知道测量系统会出什么结果。在构想阶段，这样的建立一测试过程是十分宝贵的，尤其是在要用到复杂机器和物理学应用中的转向仪的时候。重力、摩擦和惯性等因素可以搞得项目师们头晕脑涨。

我们可以设想，在这个气象监测器项目中，采用一个以一个直径为 4 英寸的漏斗作为喂水装置的跷跷板模型，而不是我们所选的很好但很贵的买来的传感器。构想是这样的：水进入漏斗再流入跷跷板一边的杯子里。当杯子的重量超过了系统的杠杆力和摩擦力，杯子就会翻转。这样不但倒空了杯子中的水，而且使跷跷板另一端杯子被移到了漏斗底下，以收集更多的水。当这个杯子满时，又重复上述过程。收集雨水的速度相对于观察抽样的速度的微处理器来说非常缓慢。如果杯子比跷跷板的套筒更小些，那么就可以加快抽样频率，从而得到更高的分辨率。

然而，如果杯子太小，在下大雨的时候，系统将被搞垮。而咖啡杯那么大的杯子却有正好相反的问题。想让一个杯子装满水，得要好几个星期。这使抽样频率太慢，测量的分辨率也就大大降低。

为了检测系统到底如何，假设每个杯子盛了 1 立方英寸的水。用这样的系统，可以准确地得出在一个简单的可控制模型中需要多大的抽样频率。一加仑的水重 8.345 磅。1 立方英尺的水重 62.4 磅。这样的话，一加仑的水是：

$$1\text{加仑} = \frac{8.345\text{磅/加仑}}{62.4\text{磅立方/英尺}} = 0.1337\text{立方英尺}$$

$$1\text{加仑} = \frac{0.1337\text{立方英尺}}{1728\text{立方英寸/立方英尺}} = 231.036\text{立方英寸}$$

漏斗的直径是 4 英寸，它里面的水面面积是 12.56 平方英寸。所以一加仑水相对应的降雨量为：

$$\frac{231.036\text{立方英寸}}{12.56\text{平方英寸}} = 18.89\text{英寸}$$

如果每个杯子大约装 1 立方英寸的水，那么在 1 加仑水倒入漏斗的过程中，跷跷板将翻转 231 次。也就是说，每次翻转代表的降雨量为

$$\frac{18.39\text{英寸的降雨量}}{231\text{次翻转}} = 0.0796\text{英寸}$$

建立好这个系统，倒一些水，就可以发现它是如何工作了。倒入最后 1 加仑的水后，将得出一个比我们预期的大些或小些的数字。这没关系，只要知道这是什么就可以了。

最初的规格说明是最大可测降雨量每天 99.9 英寸。也就是说，每小时有大约 4.16 英寸的降雨量(如果真有这么大的降雨，您就得考虑造船了，而不是做什么气象监测器)。在这样大的降雨情况下，每小时将有的翻转次数为

$$\frac{99.9\text{英寸}}{24\text{小时}} \times \frac{1\text{次翻转}}{0.0796\text{英寸}} = 52.25\text{次}$$

这虽然比不上在倾盆大雨的时候每分钟跷跷板翻转 1 次，但还是可以提供每翻转 1 次 0.0796 英寸降雨量的分辨率。比起降雨量监测器规定的 ±4%的允许误差已经足够了。

7. 露点

露点是从对空气温度和湿度的计算中得来的，能够表明什么时候水分会凝结在比它更冷的物体表面上，即在什么时候水才会收集在早晨的玻璃上或者是在什么时候下水会汇集在您的敞篷汽车里——即使汽车盖已打开。

露点的计算非常复杂。因此，为了省事，我们建立了一张表，表 5-3 所示提供了从温度和湿度到露点的换算。

表 5-3 露点计算指南

	温度(单位：°F)										
	20°	30°	40°	50°	60°	70°	80°	90°	100°	110°	120°
%RH											
30	-6	4	13	20	28	36	44	52	61	69	77
35	-2	8	16	23	31	40	48	57	69	74	83
40	1	11	18	26	35	43	52	61	69	78	87
45	4	13	21	29	37	47	56	64	73	82	91
50	6	15	23	31	40	50	59	67	77	86	94
55	9	17	25	34	43	53	61	70	80	89	98
60	11	19	27	36	45	55	64	73	83	95	101
65	12	20	29	38	47	57	66	76	85	93	103
70	13	22	31	40	50	60	68	78	88	96	105
75	15	24	33	42	52	62	71	80	91	100	108
80	16	25	34	44	54	63	73	82	93	102	110
85	17	26	36	45	55	65	75	84	95	104	113
90	18	28	37	47	57	67	77	87	97	107	117

在表 5-3 中，在表的上方一行中找到空气的温度，左边一列中找到空气的湿度(百分比)，列和行交叉处单元格里的值就是露点。如果温度和湿度值在表中显示的值之间(不是整的数)，可以用插入法计算露点。例如，对于 65°F 的空气温度和 45%RH 的湿度，露点温度的值就等于在气温为 60°F 和 70°F 时各自露点的一半，即

$$37+(47-(37\div 2))=37+5=42°F$$

在气温为 65°F，相对湿度为 45%的情况下，表面温度为 42°F 或更低的物体将可以收集凝结的水分。

在我们所设计的气象站中，将这张表包含在软件中，用于根据气温和湿度查找露点。

8. 风寒

风寒温度是指由于冷空气和风的作用造成的人体相对不舒适指数。它是一个确切的温度，由于从裸露皮肤里排出的水蒸气，人体在强风下比在无风的冷空气里更“觉得”冷。该参数由 Paul A.Siple 和 Charles F.Passel 在 1941 年提出，当时经过了对各种环境温度和风速组合情况下失热率的生理研究。风寒温度等于风速为 4mph 或更小时的空气温度。在更强的风下，风寒温度比空气温度要低，会有更冷的感觉，人体不舒适指数随风速而变。

2001 年 11 月 1 日，国家气象服务中心(NWS)颁发了新的风寒温度检索表。新表的算式如下所示：

$$\text{风寒}\ (°F)=35.74+0.6215T-35.75(V^{0.16})+0.4275T(V^{0.16})$$

其中 T 代表空气温度，单位是°F，V 代表风速，单位为英里每小时。NWS 所作的变化是对从 1945 就开始使用的 Siple 和 Passel 数据的一个改进。表 5-4 将包含在软件中，用于决定风寒

温度。

等式本身的特性(它的各个参数)使其在快速、内置式计算中并不流行。在露点特征时提到的一样，这张表的数据也包括在软件中，用于根据气温和湿度查找风寒的值。

表 5-4　NWS-2001 Siple 和 Passel 风寒计算指南

	风速(单位：米每秒)							
	5	10	15	20	25	30	35	40
温度(°F)								
40	36	34	32	30	29	28	28	27
30	25	21	19	17	16	15	14	13
20	13	9	6	4	3	1	0	-1
10	1	-4	-7	-9	-11	-12	-14	-15
0	-11	-16	-19	-22	-24	-26	-27	-29
-10	-22	-28	-32	-35	-37	-39	-41	-43
-20	-35	-41	-45	-48	-51	-53	-55	-57
-30	-46	-53	-58	-61	-64	-67	-69	-71
-40	-57	-66	-71	-74	-78	-80	-82	-84

9. 电池状态

电池保护的特点完全因选择而异。它可以像测量电压那样简单，也可以像检测当前电流并确定电池内部阻抗变化那么复杂。很多情况下，一个嵌入式系统要求比较简单："有没有足够的电压使系统正常运行"。大多数情况下，这可以通过一个电阻分压器和 ADC 做到，分压器可以标出被测电压。或者，如果想用简单的 GO/No-GO 读数器，也可以在 AVR 设备外围使用模拟比较仪。

在这样的测量中分辨率并不是很重要，只需确立一个初始点来标出问题即可。只要我们小心地确立了初始点，分辨率就无所谓了。

10. 实时

实时是与我们生活同步的。在一个嵌入式系统中，实时可以像周期性的事件或时钟和日历一样简单。当我们进行测量，尤其是对测量值取对数时，时钟和日历有助于使数据更加有用。如果数据看上去有些失序，那么，实时就有助于知道该事件是在什么时候发生的。这使我们可以将数据和特定的事件或外部环境联系起来。

制造集成电路的实时时钟还需要一些伴随部件。有些完全是独立的，如 Maxim DS1302 或 DS1307，在没有微处理器的情况下维持时间，使用一个小小的锂电池就可以供几年的电，通常用 3-线 SPI 或 2-线 I^2C 与微处理器连接。大多数这样的集成电路需要一个 32.767kHz 的表面玻璃，以进行适当的操作。

有些微控制器，如 Atmel AVR，为了在没有外部集成电路时钟的微控制器内维持实时，被设计成支持精确到秒的石英输入源，例如一个 32.767kHz 的石英表。只要处理器有电，时钟就保持工作状态。还有一个特性就是支持处理器“休眠”，这种操作方式下耗电很低，基本上不耗电。为了更新实际时间，可以用计时器将处理器唤醒，事后处理器又进入休眠状态。在这个气象监测器中，室内装置一直由一个外部的插入式电源供电，另外还有用于备份的电池。显示器一直处于激活状态，所以没必要为了电池寿命而使微处理器休眠。而这种省电模式对室外装置却十分有用。

5.6.4 室外装置的硬件设计

现在已确信我们的系统是可行的，可以开始设计最后的硬件了。通过使用定义阶段得出的框图、规格说明书以及可行性研究，可以很快地将各个电路图组合成硬件的电路图。首先，开发出室外装置的每个模块图，接着将它们组成一个总图。

室外装置被设计成完全无线的模式。一个 900MHz 的传送器大约每秒钟传送一次收集到的信息至室内装置。室外装置由电池和太阳能供电，其设计思想是白天由太阳能供电，同时还对电池充电，到了晚上或阴天，则完全由电池供电。

低耗电装置用于测量温度、湿度、风速、风向和降雨量等参数，为了省电，费电的 900MHz 的传送器仅用于传送信息。微处理器监测电池/太阳能板的电压，这样就可以将供电系统的稳定情况报告给室内装置。室外装置的电路图如图 5-10 所示。某些特定部分将在下面讨论。

1. 风速输入

像前面提到的那样，风速检测器是一个磁驱动的干簧继电器模型。每次风速表转动，磁铁被带到开关的位置，磁铁在此存在期间，开关闭合。机械开关有个开关反弹的问题。由于来自风速表的输入要被计入计数器，那么就很有可能会得到多于风速表分辨率的计数。

室外装置包含的无源电路将有助于减少这样的额外计数。用一个电阻器将干簧继电器打到 Vcc(5V 直流)，这样就为微处理器提供了一个 TTL 电平输入，还用一系列的电容器将信号连到微处理器。微处理器输入端的降低电阻和 AVR 中的内置保护二极管一起组成了一个脉冲产生系统。当干簧继电器合上时，在 Timer1 的输入端就产生一个窄脉冲，这个脉冲足以让 Timer1 计数一次脉冲。由降低电阻和一系列电容组成的 RC 时间可以使信号慢到不会产生开关反弹，又足够快，以使在强风时信号不会丢失。

2. 降雨量输入

降雨量输入也要经过稍微的过滤。由于我们最关心微处理器的输入电压(开或关)，在测量降雨量时使用过滤器的主要目的就是消除噪音和过滤来自干簧继电器的信号。

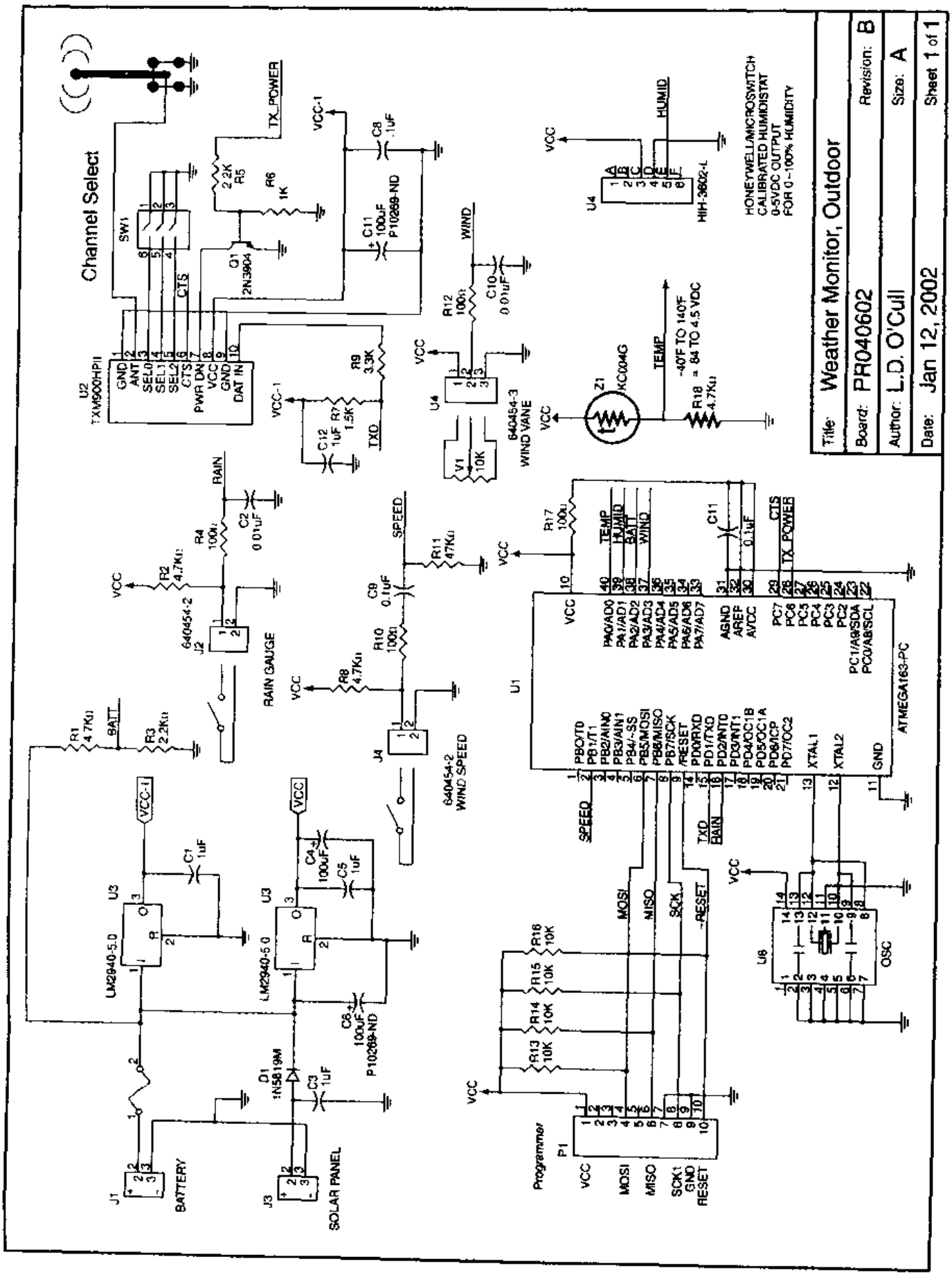


图 5-10　气象监测器室外装置电路图

3. 900MHz 传送器

900MHz 传送器是一种现有的 Linx 技术公司的 FM 模式传送器。TXM900HPII 模型支持便携式电话带宽中的 8 个不同信道，使用一个 DIP 开关来选择要用的信道，允许将发射频率与一个当地便携式电话或干扰信号的频率隔开。由于室外装置使用电池和太阳能供电，为了省电，传送器支持休眠功能。

该组件的输入是直接来自 Mega163 的 UART TXD 信号。为了不至于引起传送器“过度调制”，使用了一个 3.9K 的电阻器用于防止对传送器的输入进行过度调制。最大调制电压每设备规格说明不是 TTL，它只有大约 1V 直流。组件的输入阻抗很低。一些试验证明 3.9K 的阻抗可以给系统带来最好的性能。过度调制会导致传送器的输出超出适当的 FM 调制范围，这将引起接收端不能“听到”被传送的信号，还可能会导致传送器产生不必要的 RF 干扰信号。

4. 电力供应

电力供应相当关键。注意到这里有两组 5V 的电压供应，一组用于信号发射，一组用于其他方面。这样做是为了减少从处理器到传送器的数字噪音，同时还可以降低从传送器传回到各处的 RF。电池是一种 12V，2200 mAh 的凝胶电池型号，非常耐用，而且可充电。它们被认为是零维修型的电池。

太阳能板用二极管连入系统中，在晴天它不但可以完全为室外装置供电，而且还能重复充电。您可能还注意到在太阳能板和系统之间没有电流限制。由于太阳能板有很高的内部阻抗，它本身就有能力限制电流。增加更多的电阻只会降低电池的充电和太阳能板对系统的供电的效率。

5.6.5 室外装置的软件设计

设计软件实际上只是构想好软件如何实现功能。当软件设计完成时，就得到了一系列的任务和它们的优先级，以及一套用于描述任务之间关系的流程图。

根据描述，室外装置的软件可以认为是有些简单的。其基本思想就是收集数据并将其送入室内装置。虽然并非那么简单，但是比起室内装置来它在很多方面都是很基本的，而且从这里开始会比较好。

按优先级从高到低排列的任务列表如下：

(1) 从 Timer 1 读取风速计数

(2) 更新降雨量测量器状态

(3) 处理 ADC，开始转换和存储结果

(4) 格式化并处理要送到传送器中的数据

这 4 条任务都使用中断技术进行处理。软件的主循环执行的惟一任务就是计算何时将数据发送给传送器以及将信息存入为传送器准备的缓冲区中。

图 5-11 所示为室外装置软件的流程图。注意这里有 1 个主过程和 4 个中断或异常过程：降雨量测量中断(INT0)，对温度、湿度、风向和电压读数的抽样(ADC)，为 UART 的传送(UART_TXT)，对风速计时器 Timer 1 的风速抽样以及启动传送的抽样(TMR0)。

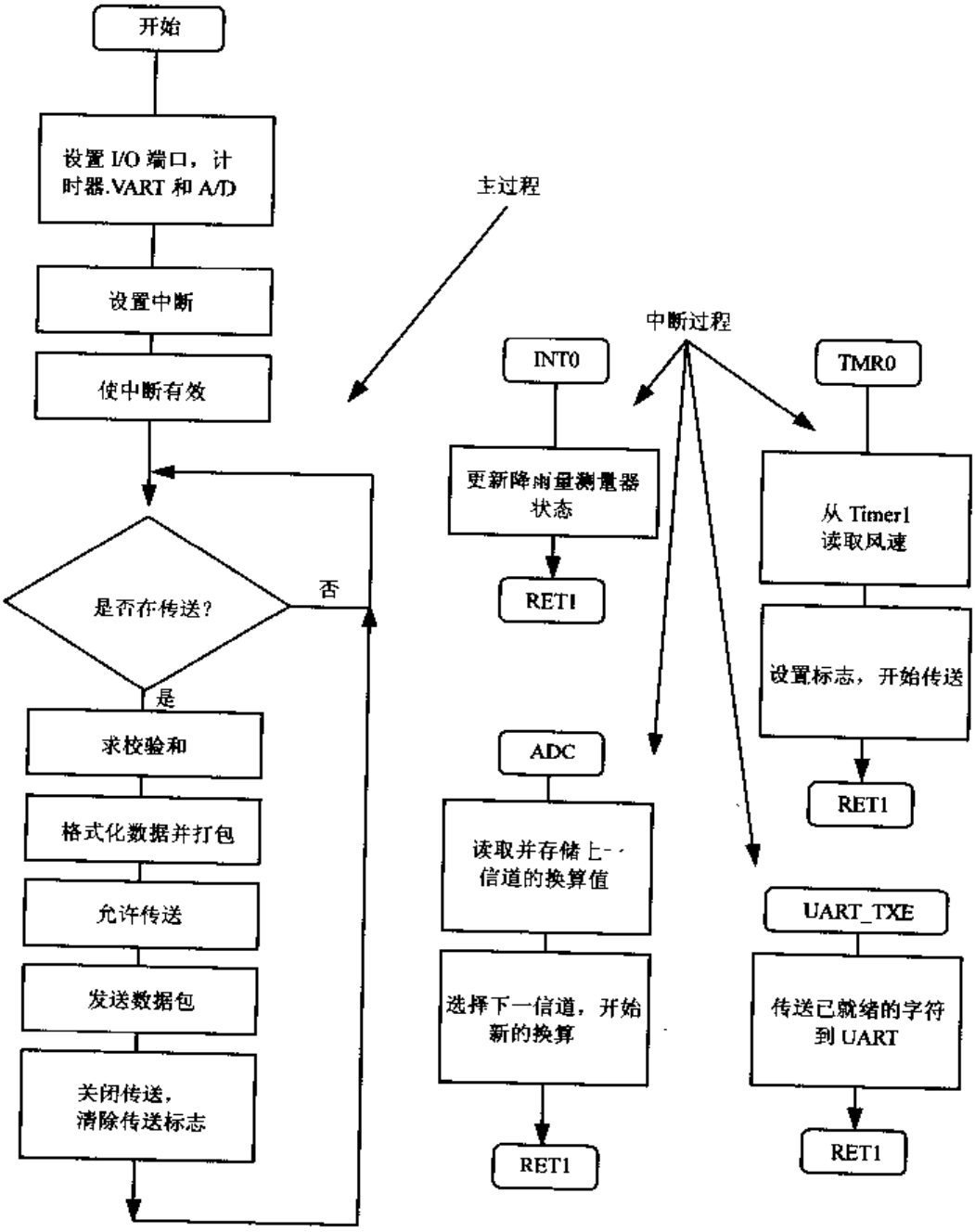


图 5-11　室外装置基本软件流程图

基本的实际测量过程在讲硬件时就已经讲过了。安装在 AVR 中的 ADC 测量温度、湿度和电池状况。Timer 1 对风速进行抽样。INT0 用于检测降雨量。通过使用 AVR 的 UART 可以保持无线电遥测的简单性，这样就可以用标准 I/O 线路实现“暗箱直接输出”。

5.6.6 室内装置的硬件设计

现在来重复对室内装置的设计过程。硬件设计完成时，就可以得到室内装置的完整电路图了，如图 5-12 所示。

室内装置中有一个 900MHz 的接收组件，用于接收室外装置收集的信息。像室外装置一样，室内装置使用基本的供电系统、微处理器、温度传感器及湿度传感器。而额外的部件包括大气压传感器，按钮，LED，一个声信号装置，一个为实时时钟而设的 32.767kHz 的石英钟，一个 LCD 显示器，这些部件使它在功能上与室外装置有很大不同。

室内装置中没有电池充电系统，900MHz 的接收器始终保持激活状态，以便能收到来自室外装置的报告。输入电源/电池的电压处在监测下，以保证室内装置能报告电压偏低的情况。

1. 900MHz 接收器

900MHz 接收器是一种现有的 Linx 技术公司生产的 FM 组件。RXM900HPII 组件支持“便携式电话”带宽中的 8 个不同信道。使用一个 DIP 开关就可以选择要用的信道，允许将发射频率与当地便携式电话频率和干扰信号频率隔离开来。该组件的输出被连接到 Mega163 的 UART RXD 信号上。使用一个 1K 的电阻，通过减少可能从微处理器流回到接收器的 RF 电流量就可以将少量的 RF 与微处理器相隔离。微处理器流回到接收器的 RF 电流是由微处理器所有输入端处的电容产生的，电阻和这些电容一起形成一个简单的过滤系统。

接收器的 RSSI 输出提供了信号的强度指示。通过对该输出的处理和让 LED 亮起来就可以指示正在接收来自室外装置的数据(LED 可能还会指示有干扰信号，而且可以帮助选择信道)。经过处理的 RSSI 信号还可以被用于在 UART 处对输入的数据进行质量把关——指示数据是适用的，而不是噪声。寄生噪声或大气噪声可能会出现在数据中，分辨哪些是信号哪些是噪声的工作留给软件来做。通过创建一个“RF 检测”信号 RF.DET，可以使用一个中断和门信号来使 UART 有效，以及降低软件用于对来自室外装置的信号进行编码的开销。

2. 电力供应

像室外装置一样，这里提供两组 5V 电压供应，一组用于发射，另一组用在其他地方。这样做是为了减少从处理器到接收器的噪声。电池是 6V 的，由 4 节五号电池串联组成。现有的电力供应是由二极管耦合到系统中的，允许在电力不足时快速切换到后备电池。

LCD 背光由墙壁电源直接驱动，由于背光很耗电，需要足够的电流产生大量的热量，所以选择使用墙壁电源直接驱动背光灯，从而绕过了这些环节。当没有背光时，装置继续操作(用电池电源)，但是没有背光，显示的内容就不易看清 ——如果我们想看清显示的内容，可以使用闪光。

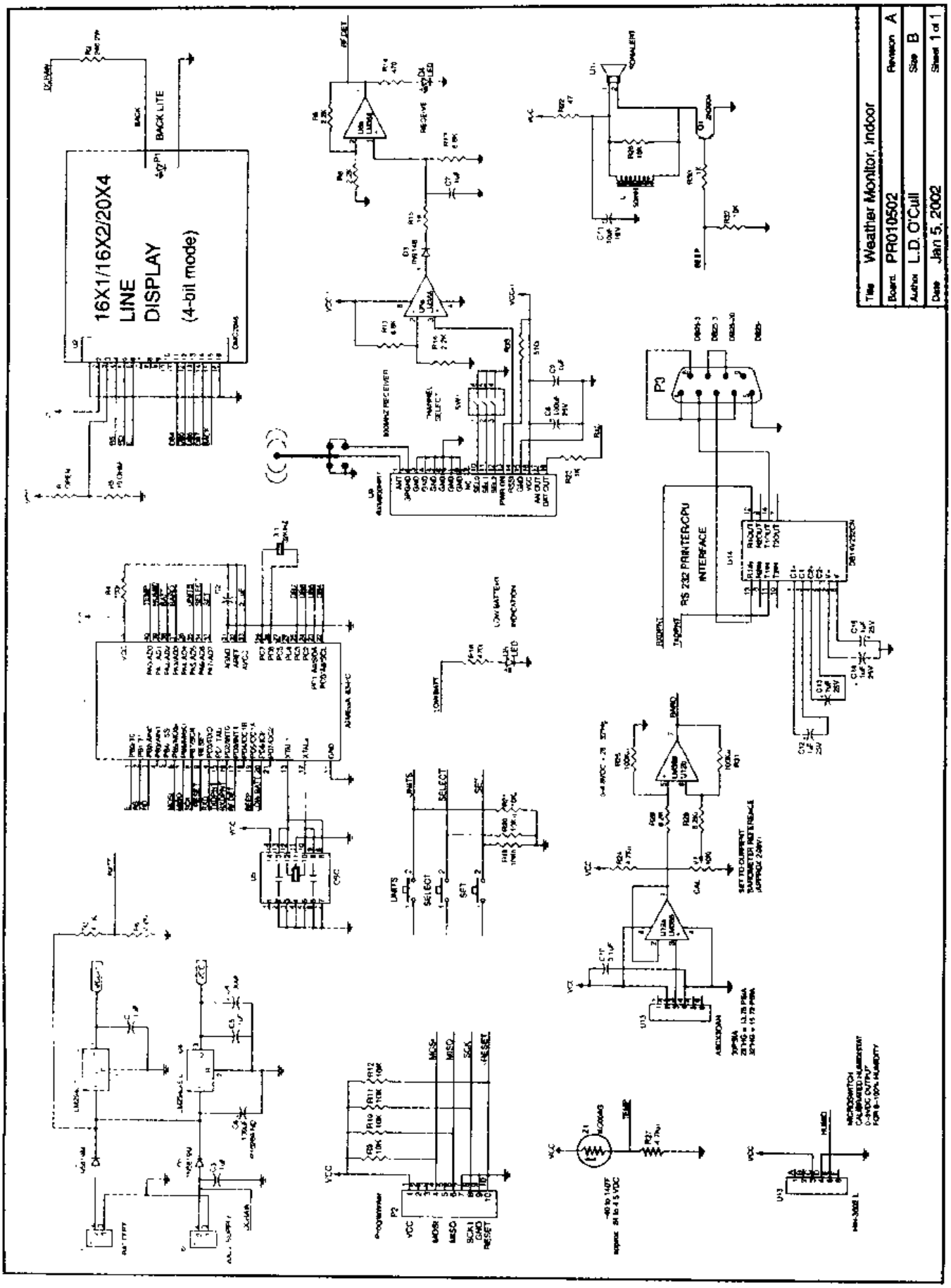


图 5-12　气象监测器室内装置电路图

5.6.7 室内装置的软件设计

室内装置有两个基本功能：收集和转换数据以及人机界面。收集过程相对简单，就是室内装置测量室内温度、湿度、大气压及其自身的 ADC 电池状况，这些与室外装置的情况差不多。室外的温度、湿度、降雨量、风速、风向和电池状况等参数，由与室内装置相同的装置收集(ADC 计数，计数器计数等)。

室内软件最困难的地方就是定义人机界面(请参考 5.6.2 节)。这个气象监测器使用 4 行 20 列 LCD 和 3 个按钮。一个 4×20 的 LCD 看上去似乎可以显示很多信息，但很快就会发现其实是很有限的。在此系统中，所有的 4 行都用于显示参数。参数分为 3 组，如表 5-5 所示。

表 5-5 所显示的参数组

参 数	参 数	参 数
室内温度	室外温度	风寒(室外)
大气压	风速	风向
室内湿度	室外湿度	露点(室外)
降雨量	时间(极短)	日期

图 5-13 给出了一个室内装置的 LCD、LED 和按钮的分布。

I	-	0	6	8	F		O	-	0	4	3	F		W	C	0	3	3	F
B	3	1	.	5	▲		O	1	5	M	P	H		3	4	5	N	N	E
I	-	5	5	%			O	-	5	5	%			D	P	1	2	3	F
0	3	.	5	"	D		1	4	:	5	9	*			0	1	/	0	8

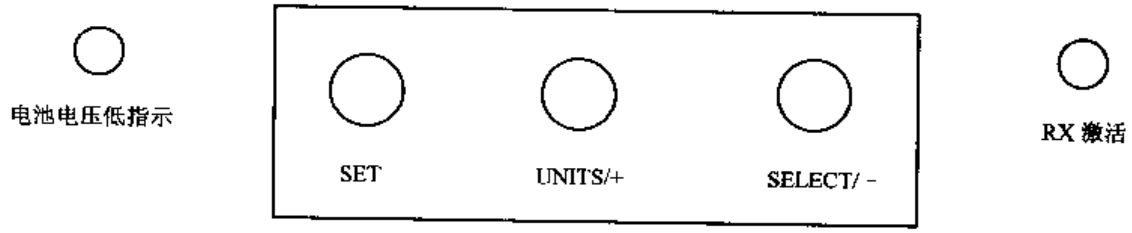


图 5-13 室内显示装置 LCD 及按钮布局示例

按钮的功能分别是 SET、UINTS/+和 SELECT/-。SET 按钮允许用户设置时间。UNITS/+和 SELECT/-按钮用于增多或减少时间或日期部件，相应地，SET 按钮跳到设置下一个部件直到所有部件都被设置。每次当用户按下 UNITS/+按钮时，当前的装置就会由英制变为米制或者 Vice versa。这样温度就可以以°C 或°F 显示，而风速也可以以 mph 或 kph 显示。SELECT/-按钮可以让用户选择以英寸/小时、英寸/天、英寸/月或英寸/年来显示降雨量。

这样的气象监测器可以侦测到很多的警告或警报情况。温度过高或过低，风和雨过大，室内温度过高可能损坏家具或设备，大气压的大幅升降可能预示着暴风雨的来临。表 5-6 显示了一些可能的警报情形。

表 5-6　可能的警报/警告信息

参数/警告信息
室内温度偏高
室内温度偏低
室外温度偏高
室外温度偏低
气压偏高警报
气压偏低警报
大风警报
降雨量偏大警报
室外装置电池电压偏低
室内装置电池电压偏低
无来自室外的通信

出于本书的目的，这些警告已经超越了我们要讲述的范围，但如果您选择在气象监测器中实施它们，则这些消息很有用。对于我们的气象监测器，可以键入两个基本的系统错误，以合理地阻止监测器工作：

- 室内或室外装置电压不足
- 没有来自室外装置的数据传送

显示器的附近有一个 LED 指示器，用于指示室内装置的电池偏少或电压偏低情形。只要电压偏低的情形出现，软件就会使 LED 闪烁，直到按下了 SET 按钮。这样用户在用电过程中就可以发现电压偏低，该改变这种情况了。用户可以换上新电池，然后按 SET 按钮清除错误。可以在任何时候检查电池状况，只要将装置断电。

这个低电压检测也可以用来智能地关闭系统，无论是室内还是室外装置。如果检测到电压低于某个正常值，累计降雨量等数据可以保存到 EEPROM 中，然后装置就可以进入下一个循环，等待电源关闭。以这种需要时才写入的方式处理 EEPROM 存储可以延长其寿命，因为连续地往 EEPROM 中写数据最终会导致存储失败。像第 1 章中介绍的，这种失败可能是由于 EEPROM 存储器本身的构造(即电线学功能)引起的。在很多情况下，这个存储器最多可允许 10 000 或 100 000 次写操作。

室外装置的电压过低最终会导致它不能报告数据。在这种情况下，室内装置的电压偏低 LED 指示灯会一直亮着，直到外部电压恢复到正常值，LED 灯才会熄灭。

如果在 15 秒内没有接收到外部装置的合法数据，就会出现一个“No Communication”错误。这可能是由于室外装置停电或长时间的 RF 干扰所致。如果存在这种情况，所有与室外有关的数据都由问号代替，例如，风速可能表示为：

```
???MPH
```

5.6.8 测试定义阶段

项目的测试定义阶段指我们打算怎样测试单个模块和整个项目。有些地方很容易测试，而有些则需要更多的工作和创造性，有些可能还需要为微控制器编写软件。

项目各部分的测试定义如下所示：

1. 风向

输入：来自一个 10 kohm 电位计的 0V～5V 直流电

预期结果：0V～5V 的直流电输入到微控制器

方法：通过一个电位计产生不同的输入电压用于模拟一个风向标，用电压计测量输入到微控制器中的电压。

2. 风速

输入：30 mph 的风——需要一个稳定的好天气和以方便可靠的方式传输

预期结果：75 Hz 的信号输入到微处理器

方法：以一个已知的频率将风速表放在风中，用一个频率计数器、示波器或者微控制器自己来测出传送到微控制器中的脉冲串。如果想用便携式的方法对脉冲计数，您可能还要编写脉冲计数的软件，将计数结果传送到室内装置并显示出来。本项目选用的风速表出厂前就经过校准，每翻转一次产生一个脉冲。输出大约是 2.515 Hz/mph。现在只需要小幅的校正就可以了。

3. 降雨量测量仪

输入：一杯(8 盎斯)水

预期结果：有代表意义的翻转次数作为微控制器的输入

方法：将水缓缓倒入降雨量测量仪中，用一个计数器或微控制器本身来计数翻转次数。RG-T 对应每 0.01 英寸的降雨量翻转一次，因此计数器要指示 722 次的上下翻转，或者 1444 次的状态变化。可能还要编写软件来计数翻转次数，将其传给室内装置并显示出来。这里选用的降雨量测量仪在出厂前就经过校对，无需再做调节。

4. 空气温度

输入：冰水，室温

预期结果：有代表意义的电压作为微控制器的输入

方法：将一个 NTC 装置接在一个约 24 英寸的导线上。用一个塑料袋将设备包起来(尽量排出袋子里面的空气)。将装置放入 32°F(0°C)的冰水中，过了大约 1 分钟后，微控制器就会收到一个 2.39V DC 的输入。对于室温，70°F(21°C)对应着 3.3V 的电压。

5. 大气压

输入：空气

预期结果：有代表意义的电压作为微控制器的输入

方法：使用另外一个大气压值作为参考点(或是当地气象站不断通过电视发送的值)，用电压计测出输入到微控制器的电压：0V DC 代表 28 inHg，而 5V DC 代表 32 inHg。

6. 相对湿度

输入：空气

预期结果：有代表意义的电压作为微控制器的输入

方法：使用另外一个湿度值作为参考点，用电压计测出输入到微控制器的电压并作比较：0 V DC代表0% RH，5V DC代表100% RH。本项目选用的恒湿器出厂前经过校准，无需再做调整。

7. 整个项目的系统测试

系统测试定义的扩展随项目最终用户的不同而不同。如果气象站项目是为家庭设计的，测试就只是将测量结果与当地气象站的预报结果相比较，看是否一致。如果是基于商业目的或者是为客户设计的，那就需要更多的测试。在这种情况下，要在各种天气条件下使用温度和湿度控制箱来运行室内和室外装置，并得出结果。例如，在测量温度时可能每隔5°F记录下结果比较合适，这样可以保证装置能够满足温度范围的要求。每个天气参数都要以同样的方式测试，并将结果制成表，交给客户。

无论对系统作怎样的扩展测试，目的只有一个：让项目的最终用户相信装置能满足定义阶段的规格说明。

5.6.9 建立和测试原型硬件阶段

项目硬件设计已经完成了，硬件部分的预期结果也指定了，现在是建立硬件的时候了。硬件组装完毕后，将执行前一阶段的测试。本小节简单讨论一些可用于进行测试并得出测量结果的实际方法，以及用于执行测试所需的软件。

电压计和示波器可以最有效地完成硬件的基本测试。当一个AVR微控制器被完全清除后，所有引脚都用于输入，这对周围电路没有影响。这样就可以进行一些硬件功能的基本检测。无论多好的软件，都不能弥补硬件不能工作的缺陷。下面列出的是对室外装置和室内装置的硬件的测试步骤。这种测试方法适用于很多类型的设计。如果能确定信号和所预期的一样，那么就可以大大地缩短软件开发和集成的时间，还可以减少出现事情迫在眉睫的那种紧张情况。

1. 室外装置校正

(1) 在断开电池或太阳能板电源的情况下，将实验电源(lab supply)设置成12V DC，电流限制在大约1A。

(2) 将实验电源与电池或连在室外装置上的太阳能板相连。

(3) 保证Vcc和Vcc-1是5V DC，如果电源供应不正确，系统将不能正常工作。

(4) 使用电压计检测确认BATT信号(Mega163的38号引脚)大约为3.8V DC。

(5) 使用电压计检测确认恒湿器的输出值与室内湿度大致相同，办公室或教室的典型湿度值为20%~30% RH。不管是多少，都应当在HUMID信号(Mega163的39号引脚)测量其对5V的百分比值。

(6) 使用电压计检测确认温度传感器的输出与室内温度大致相同，办公室和教室的平均温度是70°F(21°C)，这样TEMP信号(Mega163的40号引脚)大概应该为3.3V。如果用手指捏一

下热敏电阻，会发现电压升高了，因为您的体温使传感器温度升高了。

(7) 将风速表和风向标接到适当的输入端。

(8) 使用一个示波器确认 SPEED 信号(Mega163 的 2 号引脚)有足够强的脉冲信号(TTL 级，高于 3.3V 为“高”，低于 1.5V 为“低”)。这个信号只在转动风速表时才会出现，所以首先要让它转起来。

(9) 使用一个电压计，检测确认 WIND 输入(Mega163 的 37 号引脚)。当让风速表缓慢旋转时，该值从 0V DC 渐变到 5V DC；当在南北方向移动时，电压从 0V 跳到 5V，或者从 5V 跳到 0V。

(10) 将降雨量测量仪连接到适当的输入端。

(11) 使用电压计或示波器，检测确认 RAIN 信号(Mega163 的 16 号引脚)，如果用手拍打跷跷板，或通过测量仪用水冲跷跷板，就可以发现 RAIN 信号从 0V 变化到 5V，然后又变回来。

(12) 使用下面的测量程序，根据以下的保险位(fuse bit)设置对 Mega163 编程，这些设置保证示波器和用于重置微控制器的警报灯侦测器(brownout detector)操作无误：

- CKSEL3:0=1100,Crystal Oscillator,BOD enabled
- BODEN=ON
- BODLVL=ON

```
/*********************************************
      Outdoor unit test
*********************************************/

#include <mega163.h>
#include <stdio.h>
#include <delay.h>

void main(void)
{
    PORTC=0x00;          // setup port C and turn on TX_POWER..
    DDRC=0x40;

    //VART Transmitter: On
    // UART Baud rate: 9600, 8 Data, 1 Stop, No Parity
    UCSRA=0x00;
    UCSRB=0x08;
    UBRR=0x19;
    UBRRHI=0x00;

    while (1)
    {
     delay_ms(100);  // allow power up time
     putsf("UUU$test*QQQ\r");
     delay_ms(500);  // transmit message
     PORTC.6 = 1;    // RF power off
     delay_ms(500);   // wait..
```

```
            PORTC.6 = 0;    // RF power on
          }
        }
```

(13) 使用一个 RF 信号强度测量仪，一个频谱分析仪或一个 Linx Technologies TXM900HPII 仿真工具以及一台 PC，监控传送的消息。注意 RF 能量强度，应该在传送过后的 500 毫秒内消失。传送每隔 1 秒进行一次。

2. 室内装置测试

(1) 在断开电池或太阳能板电源的情况下，将实验电源设置成 9V DC，电流限制在大约 1A。

(2) 将实验电源与室内装置的墙壁电源相连。

(3) 保证 Vcc 和 Vcc-1 是 5V DC，如果电源供应不正确，系统将不能正常工作。

(4) 确信 LCD 的背光是亮着的。

(5) 使用电压计检测确认 BATT 信号(Mega163 的 38 号引脚)大约为 2.8V DC。

(6) 使用电压计检测确认恒湿器的输出值与室内湿度大致相同，办公室或教室的典型湿度值为 20%~30% RH。不管是多少，都应当在 HUMID 信号(Mega163 的 39 号引脚)测量其对 5V 的百分比值。

(7) 使用电压计检测确认温度传感器的输出与室内温度大致相同，办公室和教室的平均温度是 70°F(21°C)，这样，TEMP 信号(Mega163 的 40 号引脚)大概应该为 3.3V。如果用手指捏一下热敏电阻，会发现电压升高了，因为您的体温使传感器温度升高了。

(8) 使用一个电压计测量 CAL 电位计的滑动片电压(potentiometer wiper)，将其调至 2.06V DC。紧接着，测量 RARO 信号(Mega163 的 37 号引脚)的电压，测量确保它正好代表了当前的大气压。记住，BARO 信号的 0V DC 代表 28 inHg，5V DC 代表 31 inHg。平常，读数大约在中间的位置。这是标注气压计刻度的好时候。得到一个本地当前的气压值，算出在该气压下 BARO 信号对应的电压。校准 CAL 电位计，直到从 BARO 读到了这个电压值。

(9) 当室外装置正在运行其能产生 RF 信号的测试程序时，通过监测 RX 的处于激活状态的 LED 灯，检查 RF.DET 信号。RX 激活 LED 灯应该持续亮 600 毫秒，然后熄灭 500 毫秒，如此循环往复。利用示波器观测 RXD 输入端(Mega163 的 14 号引脚)，检测确信数据正在输入到处理器。波形看上去应该有点像室外装置的 TXD 输出(15 号引脚)。

(10) 使用下面的测试程序，根据以下的保险位设置对 Mega163 编程，这些设置保证示波器和用于重置微控制器的部分警报灯侦测器操作无误：

- CKSEL3:0=1100,Crystal Oscillator,BOD enabled
- BODEN=ON
- BODLVL=ON

```
/*************************************************
    Indoor Unit Test Code
*************************************************/

#include <mega163.h>
#include <delay.h>
```

```
#define LCD_E       PORTB.0      /* LCD Control Lines */
#define LCD_RS      PORTB.1
#define LCD_RW      PORTB.2
#define LCD_PORT    PORTC        /* LCD DATA PORT */
unsigned  char  LCD_ADDR;

/***** Display Functions *****/
void wr_half(unsigned char data)
{
     LCD_RW = 0;          // set WR active

     LCD_E = 1;       // raise E
     LCD_PORT = (data & 0x0F); // put data on port
     LCD_E = 0;       // drop E

     LCD_RW = 1;          // disable WR

    delay_ms(3);          // allow display time to think
}

void wr_disp(unsigned char data)
{
    LCD_RW = 0;     // enable write..

    LCD_E = 1;
    LCD_PORT= (data >> 4);
    LCD_E = 0;       // strobe upper half of data

    LCD_E = 1;       // out to the display
    LCD_PORT = (data & 0x0F);
    LCD_E = 0;       // now, strobe the lower half

    LCD_RW = 1;      // disable write

    delay_ms(3); // allow time for LCD to react
}

void disp_char(unsigned char c)
{
    LCD_RS = 1;
    wr_disp(c);
    LCD_ADDR++;
}
```

```
void disp_cstr(unsigned char flash *sa)   /* display string from ROM */
{
    while(*sa != 0)
      disp_char(*sa++);
}

void init_display(void)
{
    char i;

    LCD_RW = 1;
    LCD_E = 0;          // preset interface signals..
    LCD_RS = 0;             // command mode..
    delay_ms(50);

    wr_half(0x33);      // This sequence enables the display
    wr_half(0x33);      // for 4-bit interface mode
    wr_half(0x33);      // these commands can be found
    wr_half(0x22);      // in the manufacturer's data sheets

    wr_disp(0x28);      // Enable the internal 5x7 font
    wr_disp(0x01);
    wr_disp(0x10);      // set cursor to move (instead of display shift)
    wr_disp(0x06);      // set the cursor to move right, and not shift the display
    wr_disp(0x0c);      // turns display on, cursor off, and cursor blinking off
    // init character font ram to "00"
    for(i=0x40; i<0x5F; i++)
    {
        delay_ms(10);
          LCD_RS = 0;
        wr_disp(i);
        delay_ms(10);
        disp_char(0);
    }
    LCD_RS = 1;                 // data mode
}

void clear_display(void)
{
    LCD_RS = 0;
    wr_disp(0x01);          // clear display and home cursor
    delay_ms(10);
        LCD_RS = 1;
}
```

```
void main(void)
{
    // Port B
    PORTB=0x00;
    DDRB=0x07;              //outputs for LCD signals

    // Port C
    PORTC=0x00;
    DDRC=0x0F;              //outputs for LCD data

    // Port D
    PORTD=0x00;
    DDRD=0x60;              //beeper and LED outputs

    // Enable 32K Clock source: TOSC1 pin
    ASSR=0x08;


    delay_ms(100);

    init_display();     // initialize LCD
    delay_ms(100);

    clear_display();   // clear display
    delay_ms(100);

    disp_cstr("Indoor Test"); // print test message..

    while (1)
    {
        if(PINA & 0xE0)          // if any button pressed...
        {
            PORTD.6 = 1;  // LOW_BATT   LED   ON
            PORTD.5 = 1;
            delay_ms(1);
            PORTD.5 = 0;  // beep the beeper
            delay_ms(1);
        }
        else
        {
            PORTD.6 = 0;  // LOW_BATT LED OFF
            PORTD.5 = 0;  // beeper OFF
        }
    }
}
```

(11) 检验确定显示在 LCD 上的 Indoor Test 信息。

(12) 分别按下 UNITS，SELECT 和 SET 按钮。报警器应该会发声，频率大约为 500Hz，并且只要按钮是按下的，它就一直响着。

(13) 利用一个示波器，通过检测 Mega163 引脚 28 和 29 上的波形，检验确保运行过程中 32kHz 振荡器的工作状态。

5.6.10 系统集成和软件开发阶段——室外装置

设计室外装置的软件的第一步是定义装置的操作。本例中，必须做出一个关于在哪里处理真实数据的系统决定，室外还是室内。如果是在室外处理数据，那么每类数据都必须进行标准化，并且一些常用的数据类型，例如温度到底是用°F 还是°C，也需要进行定义，以方便传送。处理降雨量可能会有点麻烦，因为它基本上是“永恒”不变的(相对采样的频率来说)，但是您要让室外装置知道到底“永恒”是多长一段时间。基于以上的考虑，我们尽可能保留所收集数据的原始性，留到最后让室内装置进行转换。另外，室内装置有时钟，还有人机交互的接口。

因此，这个系统的最基本的数据是 ADC 的计数，Timer 1 的数值，雨量计的状态和风向标的角度。这些数据可以定期打包再发送出去。包传送的周期并不严格。要有足够的数据才能记录到风速的峰值和准确表示降雨量。这些不是变化特别快的参数，但是人们喜欢它们被高频地更新。例如，一个典型的数字电压计提供一秒的采样率。这里对采样要求严格的参数只有一个，就是风速。两次读数之间的时间对于计算风速是很重要的。虽然它被以很快的速度采样，但是这些读数的平均值就足以代表它的速度了。

另一个值得注意的问题是，在数据传输中电池的耗费。在往室内装置传送数据时，是系统对电流量要求最大的时候。占空比，或者说活动时间和非活动时间的比值，基本上决定了电源要提供的平均电流，无论它是太阳能板、电池、还是两者共同供电。

编写软件的下一步就是要定义将要用到的微控制器的输入和输出。这是一个开始的好办法，因为在系统健全性分析中已经画好了系统的简图，一旦定义了，回头查看简图的需要将会大大减少。室外装置 I/O 的定义如下所示。注意#define 语句后面的解释要有“/*”和“*/”作为分隔符。

```
#define TX_CTS                  PINC.7      /* transmitter clear to send */
#define TX_POWER                PORTC.6     /* transmitter power enable */
#define WIND_SPEED_INPUT        PINB.1      /* anemometer input(Timer1) */
#define RAIN_INPUT              PIND.2      /* rain gauge input */

#define FIRST_ADC_INPUT 0                   /* A/D converter parameters */
#define LAST_ADC_INPUT 3
#define ADC_VREF_TYPE 0x40
unsigned int adc_data[LAST_ADC_INPUT-FIRST_ADC_INPUT+1];

#define TEMPERATURE     adc_data[0] /* analog chan 0, int */
#define HUMIDITY        adc_data[1] /* analog chan 1, int */
#define BATTERY         adc_data[2] /* analog chan 2, int */
#define BAROMETER       adc_data[3] /* analog chan 3, int */
```

一旦指定了引脚，就要设置相应的输入、输出和外围设备。这些通常都是在 main()开始时进行处理或者调用。在初始化例程 init_AVR 中(参看下面的代码)，请注意 Timer 0 被设置为创建一个溢出中断。基于一个 4.0MHz 的振荡器，这个中断将会每 4.096 毫秒发生一次。这个实时“钟摆”将会用来对风速计采样，以及确定何时往室内装置发送数据。Timer 1 是被设置来捕捉风速计上开关的转移，以用来计算风速。INT 的中断产生条件被设置为 any change，这将会捕捉到雨量计翻斗的每一次翻转。UART 的设置是 9600 波特，8 位和一个无线电发射的结束位。ADC 设置为以最慢的速度进行转换，通常这样的效果最好，并在每次转换结束时产生一个中断。

```
#define FIRST_ADC_INPVT 0 /*A/D converter parameters */
#define LAST_ADC_INPVT 3
#define ADC_VREF_TYPE 0x40
void init_AVR(void)
{
      // Input/Output Ports initialization

      // Port C
      PORTC=0x00;
      DDRC=0x40;   //PORTC.6 = output, all others are input

      // Port D
      PORTD=0x02;
      DDRD=0x02;    //PORTD.1 = output, all others are input

      // Timer/Counter 0 initialization
      // Clock source: System Clock
      // Clock value: 62.500 kHz
      // Mode: Output Compare
      // OC0 output: Disconnected
      TCCR0=0x03;
      TCNT0=0x00;

      // Timer/Counter 1 initialization
      // Clock source: T1 pin Rising Edge
      TCCR1A=0x00;
      TCCR1B=0x07;

      // External Interrupt(s) initialization
      // INT0: On
      // INT0 Mode: Any change
      // INT1: Off
      GIMSK=0x40;
      MCUCR=0x01;
      GIFR=0x40;
```

```
    // Timer(s)/Counter(s) Interrupt(s) initialization
    TIMSK=0x01;

    // UART initialization
    // Communication Parameters: 8 Data, 1 Stop, No Parity
    // UART Receiver: Off
    // UART Transmitter: On
    // UART Baud rate: 9600
    UCSRA=0x00;
    UCSRB=0x48;
    UBRR=0x19;

    UBRRHI=0x00;

    // ADC initialization
    // ADC Clock frequency: 31.250 kHz
    // ADC Voltage Reference: AVCC pin
    ADMUX=FIRST_ADC_INPUT | ADC_VREF_TYPE;
    ADCSR=0xCF;

    // Watchdog Timer initialization
    // Watchdog Timer Prescaler: OSC/2048
    WDTCR=0x0F;

    // Global enable interrupts
    #asm("sei")
}
```

1. 温度、湿度、风向和电池电量

温度，湿度，风向和电池电量都可以利用 ADC 中断服务程序在后台处理。数组 adc_data 的值会被自动加载。TEMPERATURE、HUMIDITY、WIND_DIRECTION 和 BATTERY 的定义是帮助我们记住哪个信号代表哪个信道的别名。通用 ADC 中断的子例程的代码如下所示。

```
#define FIRST_ADC_INPUT 0       /* A/D converter parameters */
#define LAST_ADC_INPUT 3
#define ADC_VREF_TYPE 0x40
unsigned int adc_data[LAST_ADC_INPUT-FIRST_ADC_INPUT+1];
char  input_index=0;

#define TEMPERATURE         adc_data[0] /* analog chan 0, int */
#define HUMIDITY            adc_data[1] /* analog chan 1, int */
#define BATTERY             adc_data[2] /* analog chan 2, int */
#define WIND_DIRECTION      adc_data[3] /* analog chan 3, int */
```

```
// ADC interrupt service routine
// with auto input scanning
interrupt [ADC_INT] void adc_isr(void)
{
    // Read the ADC conversion result
    adc_data[input_index]=ADCW;

    // Select next ADC input
    if (++input_index>(LAST_ADC_INPUT - FIRST_ADC_INPUT))
        input_index=0;

    ADMUX=input_index+FIRST_ADC_INPUT|ADC_VREF_TYPE;
    // Start the next ADC conversion

    ADCSR|=0x40;
}
```

2. 降雨量

降雨量只是通过简单地跟踪 I/O 引脚的状态来处理。在这里使用一个中断，可能有点小题大做了，但是现实中雨量计的操作还有些问题，留一些余地总是个好办法。PIND.2 的 INT0 功能的生成方式被设置为 any change。这意味着 PIND.2 引脚上面的信号一旦发生改变，从高转向低或从低转向高，中断服务程序 ext_int0_isr()就会被调用。在这个程序中，字符变量 rain_state 是设置来反映雨量计输入 RAIN_INPUT 的，也就是引脚本身。前面曾经讨论过开关跳动的可能性。就雨量计的情况而言，翻斗活动的频率非常高，而向室内设置报告的是实际输入引脚的状态，所以实际上能产生读数错误的几率是可以忽略的。该中断服务程序的代码如下所示。

```
#define RAIN_INPUT PIND.2     /* rain gauge input   */
char rain_state;              /* current state of rain gauge as a char */

// External Interrupt 0 service routine
interrupt [EXT_INT0] void ext_int0_isr(void)
{
    if(RAIN_INPUT)
        rain_state = 1;    // keep change around in a variable
    else                   // for later transmission to indoor
        rain_state = 0; // unit...
}
```

3. 风速

在前面的 sanity check 中，我们决定用 Timer 1 计算风速的脉冲。对 Timer 1 计数值的采样是在 Timer0 中断程序中处理的。前面已经提过，Timer0 中断每 4.096 毫秒执行一次。如果把采样率降低到 1 秒，这是能从风力计上获取正确数据的最低频率，那么从 Timer1 收集到的数据将

会是原来的 5 倍，效率得到大大提高。为了取得大约位 1 秒的采样率，需要在 Timer 0 中断每出现 244 次的时候读取一次 Timer 1 的数据。

$$(1秒 \div 4.096微秒/次)=244.1$$

这个 1 秒的时间间隔同样可以用来决定什么时候开始无线电遥测。一旦采样频率确定，标志 RF_TX_Time 就会被设置(如下面的代码所示)，并被 main()选取以开始一次传输。

```
#define WIND_SPEED_INPUT PINB.1 /* anemometer input(Timer1)   */
unsigned int wind_speed;                    /* average counts.. */
int wind_test_pass;      /* pass count before wind speed sample */

// Timer 0 overflow interrupt service routine
interrupt [TIM0_OVF] void timer0_ovf_isr(void)
{
    if(++wind_test_pass > 243)        // passes before wind speed sample
    {
       wind_speed += TCNT1;          //read (accumulate) the timer value
       TCNT1 = 0;                    // reset counter for next sample..
       wind_test_pass = 0;     // reset pass counter

       wind_speed /= 2;          // simple average of two..
       RF_TX_Time = 1;                // time to transmit another packet
    }
}
```

4. 无线电遥测

无线电遥测(RF Telemetry)是由 main()函数处理的。标志 RF_TX_Time 是用来指示何时把数据发送给室内装置。数据都是预先打包了的。

数据包为数据提供一个结构，使接收方能方便地译码。当数据以这种方式组织时，接收方就可以预测是什么类型的数据，数据量是多少等。数据的格式如下：

```
UUU$ttt.hhh.vvv.ddd.ssss.r.xxxx*QQQ
```

其中 ttt 是温度 ADC 的读数，hhh 是湿度 ADC 的读数，vvv 是电池上的 ADC 的读数，ddd 是风向。所有 ADC 中的读数都以 3 个半字节的值发送，因为 ADC 值的范围是 0x000～0xFFF。这样就可以通过减少发送的字符总量来节省发送时间。ssss 的值是 Timer 1 的读数，r 是雨量计的状态，而 xxxx 则是所有数据的校验和。所有数据都是十六进制的。

开头的 UUU 用来提供一个对称的字节合模式。字母 U 的 ASCII 码的十六进制的值是 55(即 01010101)，这有助于发送器特征的调节和室内装置的检测。美元符号$是用来指示实际数据包的开始。点号(.)是各个值之间的分隔符，这样可以为室内装置中 sscanf()函数译码提供方便。星号(*)用来指示数据的结束。QQQ 附在后面，使数据包显得对称，Q 的十六进制 ASCII 码是 51(即 01010001)。

校验和用于检验室内装置接收的信息的有效性。校验和可以有很多种形式，例如数据和的

一个补码，两个补码，或者 CRCs(循环冗余检验码)等。通常来说，校验和就是消息中包括检验位在内的所有数据的和，接收方可以利用它可以检验数据的有效性。当校验和与接收数据的和不匹配时，就丢弃该数据。在无线系统中，无线电波干扰和大气干扰引起的数据改变时很罕有的。最重要的是，我们不会利用不正确的数据来计算天气情况。

所以来用以下格式的数据包：

```
UUU$ttt.hhh.vvv.ddd.ssss.r.xxxx*QQQ
```

如果采用上述的格式来读取 70°F 的温度，50%的湿度，11.5V 的电压值，西向 5mph 的风速，没有降雨量，则该数据包的 ASCII 码如下：

```
UUU$276.800.2EE.2FF.0032.0.1095*QQQ
```

每个测量参数的实际数据是根据变换的因子计算出来的。由于例子中的温度是 70°F，那它被传送的值计算如下：

$$\frac{\text{使用的量}}{\text{整个范围}}\times 750+172(\text{因为有0.82V的偏置量})=\text{ADC输出}$$

实际数据如下：

$$\frac{70-(-40)}{140-(-40)}\times 750+172=630.310->276_{16}$$

这样该温度是在数据包中数值就是 0x276，如下示：

```
UUU$276.800.2EE.2FF.0032.0.1095*QQQ
```

就像本书中的其他例子，将会利用中断来把字符从 UART 中发送出去。如下面的代码所示，函数 putchar()被重新定义了，以和中断服务程序一起工作，而 define_ALTERNATE_PUTCHAR_ 用来告诉标准库原来的内置函数被 putchar()代替了。

```
#define TX_BUFFER_SIZE          48
char   TX_Buffer [TX_BUFFER_SIZE+1];    // UART Transmitter Buffer
char   TX_Counter;          // interrupt service parameters
char   TX_Rd_Index;
char   TX_Wr_Index;         //UART Transmitter interrupt service routine

interrupt [UART_TXC] void uart_tx_isr(void)
{
    if (TX_Counter != 0)
    {
        if (fPrimedIt == 1)          // only send a char if one in buffer
        {
            fPrimedIt = 0;   // transmision, then don't send the char

            if(++TX_Rd_Index > TX_BUFFER_SIZE)     // test and wrap the pointer
                TX_Rd_Index = 0;
```

```
            TX_Counter--;   // keep track of the counter
        }

        if(TX_Counter != 0)
        {
            UDR = TX_Buffer[TX_Rd_Index]; // otherwise, send char out port

            if(++TX_Rd_Index > TX_BUFFER_SIZE)     // test and wrap the pointer
                    TX_Rd_Index = 0;

                TX_Counter--;                  // keep track of the counter
        }
    }
    UCSRA |= 0x40;              /* clear TX interrupt flag */
}

// Write a character to the UART Transmitter buffer
void putchar(char c)
{
    char stuffit = 0;

    while(TX_Counter > (TX_BUFFER_SIZE-1)) // WAIT!! Buffer is getting full!!
     ;

    if(TX_Counter == 0)          // if buffer empty, setup for interrupt
     stuffit = 1;

    TX_Buffer[TX_Wr_Index++] = c;          // jam the char in the buffer..

    if(TX_Wr_Index > TX_BUFFER_SIZE)               // wrap the pointer
        TX_Wr_Index = 0;

    TX_Counter++;   // keep track of buffered chars

    if(stuffit == 1)
    {               // do we have to "Prime the pump"?
        fPrimedIt = 1;
        UDR = c;   // this char starts the interrupt..
    }
}

#define             _ALTERNATE_PUTCHAR_

// Standard Input/Output functions
#include <stdio.h>
```

mian()函数列于下面的程序中。main()检测 RF_TX_Time 标志，并执行遥测操作。该操作包括计算校验和，格式化数据并打包，打开无线电发射器，发送数据包，等待传输的结束，以及关掉无线电发射器。示例中演示了这些既定的操作。在打开和关闭无线电发射器时，可能要考虑如何选择适当的时机。注意到 Watchdog 计时器在 main()例程中被定时复位。Watchdog 可以使室外装置能在低电压或其他不稳定的电压或操作条件下持续正确运作。在前面已经提到了，TEMPERATURE、HUMIDITY、WIND_DIRECTION 和 BATTERY 的定义被赋予别名，帮助我们记住每个信道对应的信号。在下面的代码中，可以看到它们如何被用作变量名。

```
#define TX_CTS        PINC.7      /* transmitter clear to send */
#define TX_POWER   PORTC.6     /* transmitter power enable */

unsigned char packet[TX_BUFFER_SIZE];    /* RF data packet */
bit RF_TX_Time;        /* time to transmit another packet */
unsigned int checksum;

#define TEMPERATURE          adc_data[0] /* analog chan 0, int */
#define HUMIDITY             adc_data[1] /* analog chan 1, int */
#define BATTERY              adc_data[2] /* analog chan 2, int */
#define WIND_DIRECTION       adc_data[3] /* analog chan 3, int */

void main(void)
{
    init_AVR();

    while (1)
    {
        #asm("wdr") //watchdog timer reset
        if(RF_TX_Time)
        {
            // disable interrupts to prevent data from changing
            // during the formatting of the packet
            #asm("cli")

            checksum = TEMPERATURE + HUMIDITY + BATTERY;
            checksum += WIND_DIRECTION + wind_speed;
            checksum +=rain_state;

          sprintf(packet,"UUU$%03X.%03X.%03X.%03X.%04X.%u.%04X*QQQ",TEMPERATURE,
                HUMIDITY,BATTERY,WIND_DIRECTION,wind_speed,
                    (int)rain_state,checksum);
          // Note: dots were added to make sscanf work on indoor unit..

            #asm("sei")
        // reenable interrupts to allow sending the packet and continue to gather data..
```

```
                #asm("wdr")
                TX_POWER = 0;              // turn on transmitter
                delay_ms(20);          // allow 20ms for power up..
                while(TX_CTS == 0)
                                // wait until transmitter is ready..
                {
                    #asm("wdr")
                }
                puts(packet);     // send packet out
                while(TX_Counter)
                {             // wait until data is all gone
                #asm("wdr")
                }
                delay_ms(20);
                // hold carrier for 20ms after end of tx..
                RF_TX_Time = 0; // reset the flag until next time
                TX_POWER = 1;       // power down until next time..
            }
        }
    }
```

5.6.11　系统集成和软件开发阶段——室内装置

室内装置的软件和室外装置的有一些共同之处，它们都用 ADC 来测量湿度、温度、大气压力和电池电量。事实上，在软件中使用相同的别名和自动扫描方法，可为后来的计算提供参考。当然和室外的比较起来还是有一些主要区别的，特别是出发点的不同。在室内装置，我们的精力主要集中在软件的设计和集成描述方面。

在室内装置中，有额外的例程来保持时间的一致，点亮 LED，以及接收来自室外装置的数据。大部分室内装置的程序都是用来显示数据和进行公式计算。这在需要和人类交互的系统中是很常见的。在开发关与此相关的软件时，就显得有点难以应付了。因为人们能接纳的天气监测的通信形式只是 LCD，一些 LED，一些按钮和报警器。

1. 时间的一致性

所使用的时间是由 Mega163 的 Timer 2 提供的。这个微控制器有一个很特别的地方，就是允许使用一个连接在 PINC.6 和 PINC.7 引脚上 32.767kHz 的石英表来驱动内部的振荡器。Timer 2 被初始化为 T1OSC/128 的预计值。这就允许 Timer 2 在 256Hz 内完成一次计数，中断会在翻转时产生，也就是每秒产生一次。

```
// Clock source: TOSC1   pin
// Clock value : PCK2/128
TCCR2=0x05 ;
```

```
ASSR=0x08 ;
```

把所有关于时间的变量统一到一个结构中，作为一个整体。这是把一组变量捆绑在一起的好方法。

```
struct TIME_DATE {
 int    hour ;
 int    minute ;
 int    second ;
 int    month ;
 int    day;
} time ;
```

Timer2 翻转时，生成一个中断，同时时间得到更新。注意，这里没有年份，同时每个月的日期也只是在这个月完成时通过查询表获取，这就意味着，这个监测系统要求用户自己调整闰年的日期。下面的代码给出了这个中断服务程序。

```
struct TIME_DATE {
      int hour;
      int minute;
      int second;
      int month;
      int day;
} time;

bit editing;              // time being edited flag

/* seconds without valid communications */
#define OUTDOOR_TIMEOUT   15
char Outdoor_Okay;    // outdoor unit is talking..

/* return max day for a given month */
const char flash MAX_DAY[13] = {
          0,31,28,31,30,31,30,31,31,30,31,30,31
};

bit    lowV_in_error;                //low battery flag..

int rain_this_hour;             //collected rainfall data
int rain_this_day;
int rain_this_month;
int rain_this_year;
void backup_rainfall(void); //prototype of EEPROM backup routine

// Timer 2 overflow interrupt service routine
// This is used to keep "Real-Time" and happens every 1 second
```

```
interrupt [TIM2_OVF] void timer2_ovf_isr(void)
{
    if(editing)                 // time is being editcd.. so don't update it!!
        return;

    if(Outdoor_Okay)            // time down for valid communnications
        Outdoor_Okay--;             // from outside unit

    if(++time.second > 59)      // count up seconds
    {                           // enough for a minute?
        time.second = 0;
        if(++time.minute > 59)  //enough minutes for an hour?
        {                       // if running on low battery
            if(lowV_in_error == 1)
                backup_rainfall();
                // backup rainfall data every hour..
            rain_this_hour = 0;
                // reset rain for the hour..
            time.minute = 0;
                // enough hours a day?
            if(++time.hour > 23)
            {
                backup_rainfall();
                    // backup data every day to eeprom
                rain_this_day = 0;
                    // reset rain for the day..
                time.hour = 0;      // enough hours for a day?
                if(++time.day > MAX_DAY[time.year])
                {   // reset rain for the month..
                    rain_this_month = 0;
                    time.day = 1;
                    // enough days for this month?
                    if(++time.month > 12)
                    {   // reset rain for the year..
                        rain_this_year = 0;
                        time.month = 1;
                        // another year gone by..
                    }
                }
            }
        }
    }
}
```

这个中断程序的实时性也为检查室外装置的通信提供了一个方法。每当室外装置的遥测消

息被成功译码时，变量 Outdoor_Okey 被设置为 15。每秒，或者在该中断程序中，该变量都会被减一。只要其值不为 0，就认为现在的天气数据是有效的。

同时要注意，在用户编辑时间和日期期间，标志 editing 是用来延长时间数据的更新。

2. 电压过低指示

这里利用一个计时器中断来监测和控制低压指示灯 LED 和低压条件。这样就允许检测和处理 LED 是一个依赖于时间的过程。这些代码可以放在 main()例程中，但是刷新 LED 的工作应该由其他过程或异常完成，例如显示更新了的时间，或者一个用户按下了一个按钮。下面列出了处理这个任务的代码。

```
#define BATTERY       adc_data[2] /* analog chan 2, int */

#define LOW_INDOOR_V        306 /* A/D counts that relate to 4.7V */
#define LOW_OUTDOOR_V     306 /* A/D counts that relate to 4.7V */
bit     lowV_out_error, lowV_in_error;

#define LOW_BATT_LED   PORTD.6 /* low battery indicator */
char rtc;
                                        //function prototype
void backup_rainfall(void)              //function prototype
// Timer 0 overflow interrupt service routine
// this happens about every 4.096ms
interrupt [TIM0_OVF] void timer0_ovf_isr(void)
{
      rtc++;       // keep count for blinky light..
                   //Detect errors and "latch" them
      if(outdoor_battery < LOW_OUTDOOR_V)
            lowV_out_error = 1;

                        // the errors are cleared elsewhere
      if(BATTERY < LOW_INDOOR_V)
      {                  // backup rain gauge data.. in case
            if(lowV_in_error == 0)       // of power failure..
                  backup_rainfall();
            lowV_in_error = 1;   //but, only per detection
      }

                  // display errors on LED
      if(lowV_out_error)
            LOW_BATT_LED = 1;             // low outdoor.. on solid..
      else if(lowV_in_error)
            {
                if(rtc & 0x40)                        // low indoor.. blink..
                   LOW_BATT_LED = 1;
```

```
            else
                LOW_BATT_LED = 0;
        }

    #asm("wdr")          /* pet the dog... */
}
```

使用计时器中断的另一个好处是，电池电压读数是由中断例程更新的，这样就为那些可能在中断测试过程中会改变的值提供了保护。通常在进行这类计算时，都要关掉中断，但是由于在一个过程中已有一个中断时，是不可能执行另外一个中断的，所以就没有必要把它关掉了。

3. 按钮和报警器

按钮只是简单的开关，由输入引脚连接，并由软件直接监测。按钮的定义类似下面的代码：

```
/*  definitions for buttons  */
#define    UNITS_BUTTON     PINA.5
#define    SELECT_BUTTON    PINA.6
#define    SET_BUTTON       PINA.7
```

报警器的管理也只是简单地利用工作在 PWM 模式下的 Timer1 来实现。这允许我们只为想要的输出频率设置计时器即可，然后把占空比的数值放到输出比较寄存器(OCR1AL)中，我们就得到一个和这个占空比相应的音调。PWM 的初始化如下示：

```
// Timer/Counter 1 initialization
// Clock source: System Clock
// Clock value: 500 kHz
// Mode: 8 bit Pulse Width Modulation
// OC1A output: Non-Inv.
TCCR1A=0x61;
TCCR1B=0x03;
```

然后使用了一些宏来控制报警器：

```
#define BEEP_ON()   {TCCR1A=0x81 ;  TCCR1B =0x0A ; OCR1AL = 0x40 ;}
#define BEEP_OFF()  {TCCR1A=0x00 ;  TCCR1B =0x00 ; OCR1AL = 0x00 ;}
```

这样就可以使代码更易于阅读。编译时，BEEP_ON()会被{TCCR1A=0x81 ； TCCR1B =0x0A ; OCR1AL = 0x40 ;}所代替。下面给出了按钮/报警器的处理程序。

```
void check_buttons(void)
{
    if(UNITS_BUTTON)          // toggle units: Imperial <-> Metric
    {
        if(last_units == 0)
        {
            units ^= 1;       // toggle units..
            last_units = 1;   // remember that button is down..
```

```
                BEEP_ON();   // we only want one beep per press,
                delay_ms(25);   // even if the button is held down.
            }
        }
        else
            last_units = 0;          // finger off of button?

        if(SELECT_BUTTON)            //toggle rainfall period: H, D, M, Y
        {
            if(last_select == 0)
            {
                which_rain++;          // toggle rainfall period..
                which_rain &= 3;       // 0-3 tells which perid..
                last_select = 1;       // remember that button is down..
                BEEP_ON();
                delay_ms(25);
            }
        }
        else
            last_select = 0;          // finger off of button?

        if(SET_BUTTON)            // Set time and date??
        {
            if(lowV_out_error || lowV_in_error)
            {
                lowV_out_error = 0; // clear LED errors and return
                lowV_in_error = 0;
                BEEP_ON();
                delay_ms(25);
            }
            else
                if(last_set == 0)  // otherwise, edit time
                {
                    last_set = 1;       // remember that button is down..
                    BEEP_ON();
                    delay_ms(25);   //beep, then on to the editing
                    set_time_date();
                }
        }
        else
            last_set = 0;             // finger off of button?

        BEEP_OFF();
}
```

很多系统只是通过按钮的触摸感觉或者单调的“哔叭”声来指示按钮已经被按下了，而报警器则提高了系统的“感觉”。报警器通过软件发出声音，确认一个按钮信号已经被接收了。将来，可以把该气象监测器的报警器功能扩展：提供天气的声音警报，甚至闹铃报警功能。这只要对软件稍微进行润色。

4. 无线电遥测的译码

室外装置遥测的处理和室外装置对它们进行结构化的处理方法非常相像。RE.DET 输入用作门控信号，以指出无线电信号的频率吻合，也就是说，它很可能使室外设备发送过来的数据包。INT1 中断是设置为工作在上升沿信号触发。这样，当检测到无线电波时，就会产生中断。INT1 的中断例程只是打开 UART 的接收器和中断：

```
// External Interrupt 1 service routine
// This occurs on RF reception..
interrupt [EXT_INT1] void ext_int1_isr(void)
{
    UCSRB=0x98;                 // turn on RX enable and IRQ
    RX_Wr_Index = UDR;   // clear any chars from input of UART

    RX_Wr_Index = 0;       //reset index of next char to be put
    RX_Rd_Index = 0;       //reset index of next char to be fetched
    RX_Counter = 0;        //reset the total count of characters
    RX_Buffer_Overflow = 0; // ..and any errors
}
```

滤波方法可以大大降低把干扰信号当数据的几率。UART 接收功能由中断控制，和室外装置发送功能的控制方式差不多。由于字节流被自动接收和缓冲了，可以利用标准 I/O 函数 getchar()和 scanf()去分析来自室外装置的信息包。下面给出了该中断服务程序。

```
// UART Receiver interrupt service routine
interrupt [UART_RXC] void uart_rx_isr(void)
{
    char c;

    c = UDR;

    Rx_Buffer[RX_Wr_Index] = c;   /* put received char in buffer */

    if(++RX_Wr_Index > RX_BUFFER_SIZE) /* wrap the pointer */
        RX_Wr_Index = 0;

    if(++RX_Counter > RX_BUFFER_SIZE) /* keep a character count */
    {                                          /* overflow check.. */
      RX_Counter = RX_BUFFER_SIZE; /* if too many chars came */
      RX_Buffer_Overflow = 1; /* in before they could be used */
```

```
    }                          /* that could cause an error!! */
}

// Get a character from the UART Receiver buffer
char getchar(void)
{
    char c;
    int i;

      i = 0;
      while(RX_Counter == 0)          /* if empty, wait for a character... */
          if(i++ > 2000)
              return - 1;          /*   timed out.. reply with - 1 */


    c = Rx_Buffer[RX_Rd_Index];       /* get one from the buffer..*/

    if(++RX_Rd_Index > RX_BUFFER_SIZE) /* wrap the pointer */
        RX_Rd_Index = 0;

    if(RX_Counter)
            RX_Counter--;             /* keep a count (buffer size) */

    return c;
}

// This define tells the compiler to replace the stdio.h
// version of getchar() with ours..
// That way, all the other stdio.h functions can use them!!
#define   _ALTERNATE_GETCHAR_

// now, we include the library and it will understand our replacements
#include <stdio.h>
```

在 mian()函数的主循环中监视变量 RX_Counter，这个变量指示接收缓冲区中的字符数。当缓冲区中有字符时，调用函数 getchar()，然后检查返回的字符，看它是不是消息的开头。

就如在室外装置软件中定义的一样，遥测包应该是以下面格式传送的：

```
UUU$ttt.hhh.vvv.ddd.ssss.r.xxxx*QQQ
```

其中 ttt 是温度 ADC 的读数，hhh 是湿度 ADC 的读数，vvv 是 ADC 上电池电压的读数，ddd 是风向。所有 ADC 中的读数都以 3 个半位的数据发送，因为 ADC 能表示的范围是 0x000～0xFFF。这样就可以通过减少发送字符的总量来节省发送时间的。ssss 的值是风速(Timer1)的读数，r 是雨量计的状态，而 xxxx 则是所有数据的校验和。所有数据都是十六进制的。

美元符$是用来指示数据包的开始，星号(*)用于指示数据包的结尾。UUU 和 QQQ 分别是

报文头和报文尾数据，用于稳定无线电传送的消息，可以被接收器忽略。一旦检测到信息的开头$，就会调用程序 get_outdoor_info()去获取并译码数据。一个名为 packet 的临时缓冲区会被用来加载$和*之间的数据。一旦接收到*，UART 的接收器就会被禁止，以防止伪输入，直到下一个来自室外装置的传输到来。标准的库函数 sscanf()被用来分析数据，并把相应的值存放到适当的变量中。遥测数据分析程序代码如下所示。

```
#define OUTDOOR_TIMEOUT   15    /* seconds without valid communications */
char Outdoor_Okay;    // outdoor unit is talking..

int out_t,w_speed,out_batt;
int w_dir,out_h,rain,checksum; // temporary variables
int last_rain,rainfall,which_rain;
bit dp_valid, wc_valid; // values are valid flag..

char packet[48];  // buffer for incoming outdoor data

void get_outdoor_info(void)     // UUU$276.800.2EE.2FF.0032.0.1095*QQQ
{
      char *p;
      char c;
      int chk;

      p = packet;

      chk = 0;
      c = getchar();
      while(c != '*')    // gather data until end of message..
      {
            *p++=c;
            if(++chk > 46)   // too much garbage?
            {
                  c = - 1;
                  break;
            }
            c = getchar();
            if(c == - 1)        // not enough characters?
                  break;
      }
      *p = 0;                   // null terminate the string..

   UCSRB=0x08;       // disable receiver until next time..

     if(c == - 1)        // packet was junk.. toss it..
          return;
                        // parse out the parameters into variables
```

```
        c = (char)sscanf(packet,"%x.%x.%x.%x.%x.%u.%x",&out_t,
              &out_h,&out_batt,&w_dir,&w_speed,
                  &rain,&checksum);
        // c now contains the count of assigned parameters..

        chk = out_t + out_h + out_batt + w_dir;
        chk += w_speed + rain;

    // test the number of parameters and the checksum for a valid message
        if((chk == checksum) && (c == 7))
        {
              Outdoor_Okay = OUTDOOR_TIMEOUT;    // reset comm timeout..
              convert_outdoor_data();      // update data for display
        }
}
```

当 sscanf()把文本流从缓冲区 packet 取出时，先把所有的值相加，与 checksum 的值比较，以确认数据没有错误。(求和的方法可以效仿室外装置软件)。如果数据是完整的，变量 Outdoor_Okey 就会被设置为 15 秒，以重新设置“无室外数据”的条件。把数据放进临时变量中，是因为这些数据还要在下一步中分析，以提取真正的变量值。

5. 采集和保护降雨量数据

降雨量是通过计数一段时间内雨量计翻斗翻转的次数而获得。在我们的天气监测系统中，采用的是 RG-T 雨量计。雨量计中的翻斗每翻转一次，就是 0.01 英寸的降雨量。变量 rain 和 last_rain 的值是用来确定雨量计的翻斗何时翻转，只有在发生翻转时的数据才允许采集。变量分别被设置以时、日、月和年来累计。翻斗每翻转一次，就往这些变量上加 0.01 英寸。实时时钟(Timer2)的中断服务程序在这些时间发生变化的时候，重置相应的累计变量(例如每个小时开始时重置以时累计的变量，每日开始时重置以日累计的变量，依次类推)。

降雨量的数学运算和其他测量值的处理方法一样，是用固定点数来完成的。这就意味着，可以利用最基础的整数运算法则进行计算，只是要在结果上加上小数点。CodeVision AVR 编译器和其他很多编译器一样，都支持浮点运算。但通常浮点数都很大，运算速度又慢，没有必要在简单的数据转换中使用。就降雨量来说，漏斗每翻转一次，就增加 0.01 英寸，但是在下面的程序中，变量每次增加的是 1，这里的小数点向右移动了两位，即每英寸的降雨量由数字 100 代表。当我们要显示 1/10 英寸的降雨量时，只要简单地把变量的值除以 10，再在显示时正确地放置小数点即可。

```
int last_rain,rain,which_rain;
int rain_this_hour;
int rain_this_day;
int rain_this_month;
int rain_this_year;

void get_rainfall(void)
```

```
{
    if(rain != last_rain)     // bucket (seesaw) has transitioned
    {
    rain_this_hour++;       // Each tip of the bucket = 0.01" of rain.
    rain_this_day++;        // These values are all integers and
    rain_this_month++;    // are treated as if the value is a fraction
    rain_this_year++;      // with the decimal at the 100s place.

    switch((int)which_rain) // convert selected value for display
    {
        case  0:
            rainfall = rain_this_hour/10; // we only dislay to 0.1"
            break;                        // so we scale the number down..
        case  1:
            rainfall = rain  this_day/10;
            break;
        case  2:
            rainfall = rain_this_month/10;
            break;
        case  3:
            rainfall = rain_this_year/10;
            break;
    }
    // develop fixed decimal point, 1/10ths of a inch
    rain_mantissa = rainfall / 10;     // now the value is "mant.frac"
    rain_frac = rainfall - (rain_mantissa * 10);
    last_rain = rain;
  }
}
```

降雨量的增加很缓慢。电量不足时引起断电可能会引起数据的丢失，而这些数据可能是以年累计的。我们可以通过简单地声明降雨量数据要保存在 EEPROM 中来解决这个问题，但是这样也有 EEPROM 由于过度擦写而损坏的问题。在这个天气监测系统中，通过把雨量数据“归档”到 EEPROM 中来保护数据。几种智能的方法每日负责数据的“归档”。

这些方法在实时时钟(Timer2)的中断服务程序中。一般来说，每日结束时，所有的累计数据就会被保存到 EEPROM 中。说是智能的，是因为具有 smart save 功能。监测到电量不足时(或者在 4 微秒之内)，当前的累计数据就会被保存起来。如果装置在低电的情况下还继续运行，则实时中断服务程序会每小时保存一次数据，直到系统断电而停止。

这样“归档”数据，大大地降低了 EEPROM 的磨损，增加了它的寿命。这里还有需要考虑的地方。当系统是第一次运行或刚从掉电中重新启动，在继续累计之前，要先把 EEPROM 中的数据调出来。另一个问题是数据有效性的问题。以下的代码给出了在这个天气监测器中是如何的：

```
eeprom unsigned char tagbyte;
 if(tagbyte != 0x55)     // if eeprom uninitialized.. then
 {                       // clear it out before using it..
       backup_rainfall();
       tagbyte = 0x55;
 }

// get values from eeprom..
 rain_this_hour = rain_hour_save;
 rain_this_day = rain_day_save;
 rain_this_month = rain_month_save;
 rain_this_year = rain_year_save;
```

变量 tagbyte 设置在 EEPROM 中，以确定 EEPROM 是否在已知状态下。监测器第一次运行时，EEPROM 的值默认(或者擦除了的)都是 0xFF，所以 tagbyte 的读数是 0xFF。用该值和一个已知的值比较，例如 0x55。如果它不等于已知值，EEPROM 则会被初始化，同时 tagbyte 被设置为 0x55。

6. 把计数转换为实际值

在 5.6.2 小节中已经规定了，所有有关天气的参数都以最基本的形式采集，例如 ADC 和计时器的读数。这使实际的采集过程既简单又清晰。但是现在再次回到人机接口的问题上。问题是许多人不能利用心算的能力来把 ADC 的读数转换为实际的温度值（° F)。事实上，很多人把 °F 格式的温度转到° C 都要费一番周折。作为开发者，这些转换将会是由我们来完成，并把信息以一个友好的形式表现出来。

下面给出了在内部转换温度、湿度和大气压的代码。就和在降雨量测量过程中一样，都是用固定点数来进行运算。大部分的转换都包括两个部分：增量部分和偏置量部分。这取决于所作的转换要用到哪一部分。

```
#define IMPERIAL    0
#define METRIC      1
char units;             // current display units

const int flash I_K_it = 12;
    // (* 0.12) scale indoor temperature, Imperial
const int flash I_Offset_it = 172;
    // (-.84V) offset for indoor temperature, Imperial

const int flash K_b = 26;
    // (1/26) scale for barometric pressure
const int flash Offset_b = 280;
    // (28.0"Hg)offset for barometric pressure

const int flash K_oh = 10;
    // (1/10) scale, indoor humidity
```

```
const int flash Offset_ih = 0;
    // offset, outdoor humidity

int indoor_temp;
int barom_mantissa,barom_frac;
int indoor_humidity;

void convert_indoor_data(void)
{
    int bar;

    // disable interrupts to prevent data from changing
    // while we are performing the calculations
    #asm("cli")

    indoor_temp = ((TEMPERATURE - I_Offset_it)*I_K_it)/100;

    if(units == METRIC)
        indoor_temp = ((indoor_temp - 32)*5)/9;   // scale to degrees C

    bar = BAROMETER / K_b + Offset_b;
    barom_mantissa = bar / 10;
    barom_frac = bar - (barom_mantissa * 10);

    indoor_humidity = HUMIDITY / K_oh + Offset_oh;

    // reenable interrupts..
    #asm("sei")
}
```

就温度而言，它有一个 0.84V 的电压偏置量(相当于 ADC 计数中的 172)，在比例缩放该值的时候，要先把偏置量减掉。删除偏置量以后，温度读数就会乘以 0.24，得到°F 格式的值。例子中的程序是通过 ADC 的计数乘以 24 来完成的，它把结果的小数点右移了 2 位(也就是，应该是 61.44°F)，然后再通过除以 100 把它转回°F 格式。一旦温度被转变成°F，如果需要，转变成公有制也是很简单。

大气压和湿度的处理就有些不同了，是在偏移前进行比例缩放。这是因为 ADC 的值是满量程使用的(0V～5V 的直流电)，同时偏移量取代 ADC 的计数直接作用在装置中。例如，就大气压而言，当压力是 28.0inHg 时，ADC 的值是 0。ADC 的读数除以一个常量(K_b)才得到一个正确表示水银高度的值(英寸)。而且这个值还是被偏置 280(或者 28.0inHg)，以获得真正的结果(介于 28.0 和 32.0inHg 的数值)。

同样的过程也适用于室外装置传送的温度、湿度和风速的测量。不过风寒和风露点需要特别处理，他们通过查表得到。在下面的程序中，显示了如何转换温度、湿度和风速，然后将这些值作为索引，从表中返回风寒和风露点的值，该值是°F 格式的。就和前面的一样，如果系统

以公制形式显示，这些值也可以转换成公制的形式。

```
#define IMPERIAL    0
#define METRIC      1
char units;             // current display units

int outdoor_temp, wind_chill;
int wind_speed, wind_degrees;
int outdoor_humidity, dew_point;
int outdoor_battery;
int out_t, w_speed, out_batt;
int w_dir,out_h,rain,checksum; // temporary variables
bit dp_valid, wc_valid; // values are valid flag..

// Dew Point Look-Up-Table    - Imperial
const char flash DEW_LUT[13][11] = {
 -6,  4,   13,  20,  28,  36,  44,  52,  61,  69,  77,
 -2,  8,   16,  23,  31,  40,  48,  57,  69,  74,  83,
 1,   11,  18,  26,  35,  43,  52,  61,  69,  78,  87,
 4,   13,  21,  29,  37,  47,  56,  64,  73,  82,  91,
 6,   15,  23,  31,  40,  50,  59,  67,  77,  86,  94,
 9,   17,  25,  34,  43,  53,  61,  70,  80,  89,  98,
 11,  19,  27,  36,  45,  55,  64,  73,  83,  95,  101,
 12,  20,  29,  38,  47,  57,  66,  76,  85,  93,  103,
 13,  22,  31,  40,  50,  60,  68,  78,  88,  96,  105,
 15,  24,  33,  42,  52,  62,  71,  80,  91,  100,  108,
 16,  25,  34,  44,  54,  63,  73,  82,  93,  102,  110,
 17,  26,  36,  45,  55,  65,  75,  84,  95,  104,  113,
 18,  28,  37,  47,  57,  67,  77,  87,  97,  107,  117
};

// Wind Chill Look-Up-Table   - Imperial
const char flash WC_LUT[9][8] =   {
     36,    34,   32,   30,  29,   28,    28,   27,
     25,    21,   19,   17,  16,   15,    14,   13,
     13,     9,    6,    4,   3,    1,     0,   -1,
     1,     -4,   -7,   -9,  -11,  -12,  -14,  -15,
    -11,    -16,  -19,  -22,  -24,  -26,  -27,  -29,
    -22,    -28,  -32,  -35,  -37,  -39,  -41,  -43,
    -34,    -41,  -45,  -48,  -51,  -53,  -55,  -57,
    -46,    -53,  -58,  -61,  -64,  -67,  -69,  -71,
    -57,    -66,  -71,  -74,  -78,  -80,  -82,  -84
};

const int flash I_K_ot = 12;
```

```
        // (* 0.12) scale outdoor temperature, Imperial
const int flash I_Offset_ot = 171;
        // ( - .84V) offset for outdoor temperature, Imperial

const int flash I_K_ws = 25;          // scale wind speed, Imperial
const int flash I_Offset_ws = 0;      // offset wind speed, Imperial

const int flash K_wd = 35;                  // (* 0.35) scale wind direction
const int flash Offset_wd = 0;              // offset wind direction

const int flash K_oh = 10;                  // (1/10) scale, outdoor humidity
const int flash Offset_oh = 0;              // offset, outdoor humidity

void convert_outdoor_data(void)
{
    int dt,dh,dw;
    long temp;

    outdoor_temp = ((out_t - I_Offset_ot)*I_K_ot)/100; // degrees F

    wind_speed = ((w_speed * 10)/ I_K_ws) + I_Offset_ws;     // MPH

    temp = (long)w_dir;                     // use long so we don't overflow!!
    temp *= K_wd;
    temp /= 100;                            // after this, int is safe..
    wind_degrees = temp + Offset_wd; // degrees from North

    outdoor_humidity = out_h / K_ih + Offset_ih;  // % RH

    outdoor_battery = out_batt;             // just counts

    if((outdoor_temp >= 20) && (outdoor_temp <= 120) &&
          (outdoor_humidity >= 30) && (outdoor_humidity <= 90))
    {
        dt = (outdoor_temp - 20) / 10;   // scale for table index
        dh = (outdoor_humidity - 30) / 5;
        dew_point = DEW_LUT[dh][dt];
        dp_valid = 1;     // interpolation could be added here!!!
    }
    else
    {
        dew_point = 0;
        dp_valid = 0;
    }
```

```
    if((outdoor_temp >= - 40) && (outdoor_temp <= 40) &&
        (wind_speed >= 5) && (wind_speed <= 40))
    {
        dt = (outdoor_temp - 40) / 10; // scale for table index
        dw = (wind_speed - 5) / 5;
        wind_chill = WC_LUT[dt][dw];
        wc_valid = 1;     // interpolation could be added here too!!!
    }
    else
    {
        wind_chill = 0;
        wc_valid = 1;
    }


    if(units == METRIC)
    {
        wind_chill =   ((wind_chill - 32)*5)/9;      // scale to degrees C
        dew_point   =   ((dew_point - 32)*5)/9;   // scale to degrees C
        outdoor_temp =   ((outdoor_temp - 32)*5)/9;     // scale to degrees C
        wind_speed =      ((wind_speed)*88)/55;          // scale to KPH
    }
}
```

7. 控制 LCD 的例程

下面给出了控制 4X20LCD 的例程。这是一个 Optrex 兼容的 4 位显示器。这意味着到达该设备的所有数据都是通过 4 条数据线来传输。这样的接口方法可以节省微控制器上珍贵的 I/O 引脚资源，同时还允许显示数据快速更新。显示器上有 3 个控制信号 E，RE(或者 R/W)和 RS。E 用于选通并确认送到显示器的数据(DB4-7)。RD 是用于指示对显示器进行读操作(RD=1)或写操作(RD=0)。RS 用于指示传送到显示器的数据是影响显示器工作模式的命令(RS=0)，还是要显示的数据(RS=1)。

函数 init_display()为天气监测器设置 LCD 的适当工作模式。使用内部的字符集，关掉了光标，清除了图形字符。LCD 显示器上的制造商铭牌应该描述了显示命令和寄存器的设置，以及对时间的要求。有些命令需要一定的时间才能显示执行。这是就需要插入延时，使当前命令能在下一条命令发出之前获得显示时间并完成操作。函数 wr_half()和 wr_disp()是分别用来发送一个字节的低四位数据或者全部数据。

剩下的命令执行文本显示操作，会使格式化和显示天气信息更方便。函数 set_LCD_cur()把不可见的光标放到指定的行和列上。函数 line()把不可见的光标放到指定行的开始处。函数 clear_display()把显示器完全擦干净，并把不可见的光标放置在左上角(0，0)。

注意，函数 disp_char()相当于 LCD 的 putchar()函数。它的作用就是把一个字符显示在显示器上。不可见的光标会在显示过程中自动移到下一个显示位置上。必须执行一些测试来检查每

一行的长度。光标通常不会自动卷到下一行，所以要检查光标的位置，然后在地址大小的基础上，正确地调用 line()函数。

函数 disp_str()和 diap_cstr()调用 disp_char()，以便显示格式化的 C 字符串。disp_str()是用来显示放在 SRAM 中的字符串；而 diap_cstr()则是显示放在 FLASH 中的字符串。

```
#define LCD_E      PORTB.0        /* LCD Control Lines */
#define LCD_RS     PORTB.1
#define LCD_RW     PORTB.2
#define LCD_PORT PORTC            /* LCD DATA PORT (lowest 4 bits)*/

// this Look-Up-Table translates LCD line/cursor positions
// into the displays' internal memory addresses
const unsigned char flash addLUT[4] = {0x80,0xC0,0x94,0xD4};

unsigned char LCD_ADDR,LCD_LINE;

/***** Display Functions *****/
void wr_half(unsigned char data)
{
      LCD_RW = 0;          // set WR active

      LCD_E = 1;           // raise E
      LCD_PORT = (data & 0x0F); // put data on port
      LCD_E = 0;           // drop E

      LCD_RW = 1;          // disable WR

    delay_ms(3);       // allow display time to think
}

void wr_disp(unsigned char data)
{
    LCD_RW = 0;     // enable write..

    LCD_E = 1;
    LCD_PORT= (data >> 4);
    LCD_E = 0;        // strobe upper half of data

    LCD_E = 1;        // out to the display
    LCD_PORT = (data & 0x0F);
    LCD_E = 0;        // now, strobe the lower half

    LCD_RW = 1;        // disable write

    delay_ms(3);       // allow time for LCD to react
```

```
}

void init_display(void)
{
    char i;

    LCD_RW = 1;
    LCD_E = 0;          // preset interface signals..
    LCD_RS = 0;             // command mode..
    delay_ms(50);

    wr_half(0x33);      // This sequence enables the display
    wr_half(0x33);      // for 4-bit interface mode
    wr_half(0x33);      // these commands can be found
    wr_half(0x22);      // in the manufacturer's data sheets

    wr_disp(0x28);      // Enable the internal 5x7 font
    wr_disp(0x01);
    wr_disp(0x10);      // set cursor to move (not shift display)
    wr_disp(0x06);      // set the cursor to move right, and not shift the display
    wr_disp(0x0c);      // turns display on, cursor off, and cursor blinking off

    for(i=0x40; i<0x5F; i++) // init character font ram to "00"
    {
        delay_ms(10);
        LCD_RS = 0;
        wr_disp(i);
        delay_ms(10);
        disp_char(0);
    }
    LCD_RS = 1;             // data mode
}

void line(char which_line)
{
    LCD_RS = 0;

    LCD_ADDR = addLUT[which_line - 1];
    wr_disp(LCD_ADDR); // move to which_line line

    LCD_RS = 1;
    LCD_ADDR = 0;
    LCD_LINE = which_line;
}
```

```
void clear_display(void)
{
    LCD_RS = 0;
    wr_disp(0x01);              // clear display and home cursor
    delay_ms(10);
    LCD_RS = 1;
    line(1);
}

void set_LCD_cur(char LCDrow, char LCDcol)
{
    LCD_RS = 0;

    LCD_ADDR = addLUT[LCDrow] + LCDcol;

    wr_disp(LCD_ADDR); // move to row and column

    LCD_RS = 1;
    LCD_LINE = LCDrow;
}

void disp_char(unsigned char c)
{
    LCD_RS = 1;
    wr_disp(c);
    LCD_ADDR++;                 // automatically wrap text as required
    if(LCD_ADDR == 20)
        line(2);
    if(LCD_ADDR == 40)
        line(3);
    if(LCD_ADDR == 60)
        line(4);
    if (LCD_ADDR == 80)
        line(1);
}

void disp_str(unsigned char *sa)    // display string from RAM
{
    while(*sa != 0)
        disp_char(*sa++);
}

void disp_cstr(unsigned char flash *sa)   /* display string from ROM */
{
    while(*sa != 0)
```

```
        disp_char(*sa++);
    }
```

8. 保证显示的更新

显示是一个相对低级的任务。这意味着，需要时 mian()函数进行显示或者调用与显示相关的函数。要显示的数据都有相同的形式，所以更新显示的一个简单方法是以相同的方法来处理每一笔数据。下面的结构用来指示信息的所有属性：

```
struct DISP_ITEM {
    char flash row;          // row position of text
    char flash col;          // col position of text
    char flash *fmtstr;      // sprintf format
    int *value;              // pointer to variable to display
    char flash *units_I;     // units mark imperial
    char flash *units_M;     // units mark metric
    char flash source;       // inside or outside measurement
};
```

每一个显示信息都有一个位置(row 和 col)，一个格式，一个值，一个数据指针以及一个用于指示数据来源的指示器(是室外的还是室内的)。通过创建一个这些类型数据的数组，就能完整描述显示的内容：

```
struct DISP_ITEM main_display [MAX_ITEM] =
{
    0,0,"I%3d",&indoor_temp,"F ","C ",INSIDE,
    0,7,"O%3d",&outdoor_temp,"F ","C ",OUTSIDE,
    0,14,"WC%2d",&wind_chill,"F ","C ",OUTSIDE,
    1,0,"B%02u",&barom_mantissa,".",".",INSIDE,
    1,4,"%1u",&barom_frac,"","",INSIDE,
    1,7,"%3u",&wind_speed,"MPH","KPH",OUTSIDE,
    1,14,"%03u",&wind_degrees,"","",OUTSIDE,
    2,0,"I%3u",&indoor_humidity,"%","%",INSIDE,
    2,7,"O%3u",&outdoor_humidity,"%","%",OUTSIDE,
    2,14,"DP%3d",&dew_point,"F","C",OUTSIDE,
    3,0,"%02u",&rain_mantissa,".",".",OUTSIDE,
    3,3,"%1u",&rain_frac,"\"","\"",OUTSIDE,
    3,7,"%02u",&time.hour,":",":",INSIDE,
    3,10,"%02u",&time.minute,"","",INSIDE,
    3,15,"%02u",&time.month,"/","/",INSIDE,
    3,18,"%02u",&time.day,"","",INSIDE
};
```

下面程序中的函数 update_display_item()格式化和显示要求的信息。先定位光标，然后利用 sprintf()把要显示的数值格式化并显示为字符串，再用 disp_str()函数把它们显示到显示器上。如

果数据是室外测量的结果，利用 main_display[item].source 可以知道来源，并且该数据在过去的 15 秒内都没有被刷新(Outdoor_Okay)，然后该信息会被问号代替。同时，如果风寒和露点值超出范围，会被短横线代替。最后，根据用户要求的数据，传送相应的文本信息 main_display [item].unit_I 或者 main_display[item].unit_M 并显示出来。

```
#define IMPERIAL    0
#define METRIC      1
char units;             // current display units
char text_buffer[24];

char Outdoor_Okay;   // outdoor unit is talking..

//-----------------------------------------------------
// display item table description for the main display
//-----------------------------------------------------
#define INSIDE      'i'
#define OUTSIDE     'o'
#define MAX_ITEM    16

char which_item;
struct DISP_ITEM {
    char flash row;              // row position of text
    char flash col;              // col position of text
    char flash *fmtstr;      // sprintf format
    int *value;                  // pointer to variable to display
    char flash *units_I;     // units mark imperial
    char flash *units_M;     // units mark metric
    char flash source;       // inside or outside measurement
};

struct DISP_ITEM main_display [MAX_ITEM] =
{
    0,0,"I%3d",&indoor_temp,"F ","C ",INSIDE,
    0,7,"O%3d",&outdoor_temp,"F ","C ",OUTSIDE,
    0,14,"WC%2d",&wind_chill,"F ","C ",OUTSIDE,
    1,0,"B%02u",&barom_mantissa,".",".",INSIDE,
    1,4,"%1u",&barom_frac,"","",INSIDE,
    1,7,"%3u",&wind_speed,"MPH","KPH",OUTSIDE,
    1,14,"%03u",&wind_degrees,"","",OUTSIDE,
    2,0,"I%3u",&indoor_humidity,"%","%",INSIDE,
    2,7,"O%3u",&outdoor_humidity,"%","%",OUTSIDE,
    2,14,"DP%3d",&dew_point,"F","C",OUTSIDE,
    3,0,"%02u",&rain_mantissa,".",".",OUTSIDE,
    3,3,"%1u",&rain_frac,"\"","\"",OUTSIDE,
    3,7,"%02u",&time.hour,":",":",INSIDE,
```

```
    3,10,"%02u",&time.minute,"","",INSIDE,
    3,15,"%02u",&time.month,"/","/",INSIDE,
    3,18,"%02u",&time.day,"","",INSIDE
};

void update_display_item(char item)
{
    char *s;

    // first, set the LCD cursor to the item's position
    set_LCD_cur(main_display[item].row,main_display[item].col);

    // place the variable into its proper format as a string
    sprintf(text_buffer,main_display[item].fmtstr,*main_display[item].value);

    // if inside data.. then always display
    if(main_display[item].source != INSIDE)
    {   // outside data may not be up to date.. if it is, great!
        if(Outdoor_Okay > 0)
        {                   // if wind chill or dew point out of range...
            if(((item == WIND_CHILL_ITEM) && (wc_valid == 0)) ||
                    ((item == DEW_POINT_ITEM) && (dp_valid == 0)))
            {
                s = text_buffer;  // point to the formatted text..
                while(*s)
                {                   // replace numbers with dashes...
                    if(isdigit(*s))
                        *s = '-';
                    s++;
                }
            }           // otherwise all is well, just print it..
            disp_str(text_buffer);
        }
        else
        {       // the outside info is not current so, ..
            s = text_buffer;  // point to the formatted text..
            while(*s)
            {   // replace numbers with question marks..
                if(isdigit(*s))
                    *s = '?';
                s++;
            }
        }
}                       // always place units string last..
```

```
    disp_str(text_buffer);

    if(units == IMPERIAL)
        disp_cstr(main_display[item].units_I);
    else
        disp_cstr(main_display[item].units_M);
}
```

整个显示的更新是由mian()函数在一个for循环中简单调用update_diaplay_item()来实现的：

```
// update entire display
for(which_item=0; which_item< MAX_ITEM; which_item++)
    update_display_item(which_item);

set_LCD_cur(3,5);          // indicate which rainfall period
disp_char(RAIN_TYPE[which_rain]);

set_LCD_cur(1,17);         // indicate wind heading
if(Outdoor_Okay)
    disp_cstr(WIND_HEADING[wind_degrees/22]);
else
    disp_cstr("---");  //..if possible..

set_LCD_cur(3,12);         // flash star with seconds..
if(time.second & 1)     // to indicate life!!
    disp_char('*');
else
    disp_char(' ');
```

降雨期，风向和一个不停闪烁的用以指示一秒间隔的星号等额外信息，都是在更新的结尾处加上去的。查询表为显示提供信息正确的字符串，估计一下要调用 disp_cstr()的 if-else 语句的数目：

```
const char flash RAIN_TYPE[4] = {'H', 'D', 'M', 'Y'}

char flash *WIND_HEADING[17] = {
    "N  ",
    "NNE",
    "NE ",
    "ENE",
    "E  ",
    "ESE",
    "SE ",
    "SSE",
    "S  ",
    "SSW",
```

```
        "SW ",
        "WSW",
        "W  ",
        "WNW",
        "NW ",
        "NNW",
        "N  "
};
```

9. 编辑时间和日期

时间和日期的编辑也是通过表格查询来处理的。这就允许用一个简单的过程用相同的方法来处理所有的数据，甚至是用相同的代码。设计一个编辑结构体，让它包含是由必需的元素：条目在 LCD 上的位置，条目的标题或名字(告诉编辑者，他正在编辑什么)，一个指向该值且可以更改的指针，以及一些防止用户“越界”的限制。

```
struct EDIT_ITEM {
    char flash row;          // row position of text
    char flash col;          // col position of text
    char flash title[10];    // item title
    int *value;              // pointer to variable to display
    int flash MinValue;      // Minimum Value
    int flash MaxValue;      // Maximum Value
};

struct EDIT_ITEM edit_display[4] = {
    0,5,"Month:  ",&time.month,1,12,
    1,5,"Day:    ",&time.day,1,31,
    2,5,"Hour:   ",&time.hour,0,23,
    3,5,"Minute: ",&time.minute,0,59
};
```

显示器必须清除并重建，以形成编辑屏幕，代码如下所示。用大于号(>)来指示当前选择的信息。整个过程是直接通过按钮来完成的：改变选择的信息，选择现一个要编辑的信息，或者退出。标志 editing 是用来通知实时中断服务程序(Timer2)，数据正在编辑中，在用户进行更改时不要更新其值。

```
    while(SET_BUTTON)        // wait for finger off the button..
    {  //pet the dog...
        #asm("wdr")
    }

    cur_item = 0;
    while(1)                 // now work each item.. +/-, and SET to move..
    {
```

```
set_LCD_cur(edit_display[cur_item].row,edit_display[cur_item].col - 1);
disp_char('>');                    // point to the item..

while(SET_BUTTON)// finger off?
{   //pet the dog...

    #asm("wdr")
}

delay_ms(25);

while((!SET_BUTTON) && (!UNITS_BUTTON) && (!SELECT_BUTTON))
{                    // wait for a button press..
    #asm("wdr")
}                    //pet the dog...

if(UNITS_BUTTON)                    // increment the current item..
{                                   // and do limit checks..
     BEEP_ON();
     time.second      = 0;
                // reset the seconds to 0 when time edited

*edit_display[cur_item].value += 1;
if(*edit_display[cur_item].value > edit_display[cur_item].MaxValue)
   *edit_display[cur_item].value = edit_display[cur_item].MinValue;

}

if(SELECT_BUTTON)                   // decrement the current item..
{                                   // and do limit checks..
     BEEP_ON();
     time.second      = 0;  // reset the seconds to 0 when time edited

     *edit_display[cur_item].value - = 1;
     if(*edit_display[cur_item].value < edit_display[cur_item].MinValue)
      *edit_display[cur_item].value = edit_display[cur_item].MaxValue;
}
                                    // update the item on the screen..
set_LCD_cur(edit_display[cur_item].row,
                                edit_display[cur_item].col);
sprintf(text_buffer,"%s %02u",edit_display[cur_item].title,
      *edit_display[cur_item].value);

delay_ms(25);          // a little time for switch settling..
BEEP_OFF();
```

```
            while((UNITS_BUTTON) || (SELECT_BUTTON))
            {   //pet the dog...
                #asm("wdr")
            }

            if(SET_BUTTON)
            {
                BEEP_ON();
                delay_ms(25);           // a little time for switch settling..
                BEEP_OFF();
                set_LCD_cur(edit_display[cur_item].row,
                                        edit_display[cur_item].col-1);
                disp_char(' ');         // erase the current cursor...
                cur_item++;
                if(cur_item > 3)
                    break;
            }
        }

        while(SET_BUTTON)  // finger off?
        {   //pet the dog...
            #asm("wdr")
        }

        editing = 0;
        clear_display();    // back to business as usual..
    }
```

图 5-14 显示了在编辑过程中 LCD 的画面。

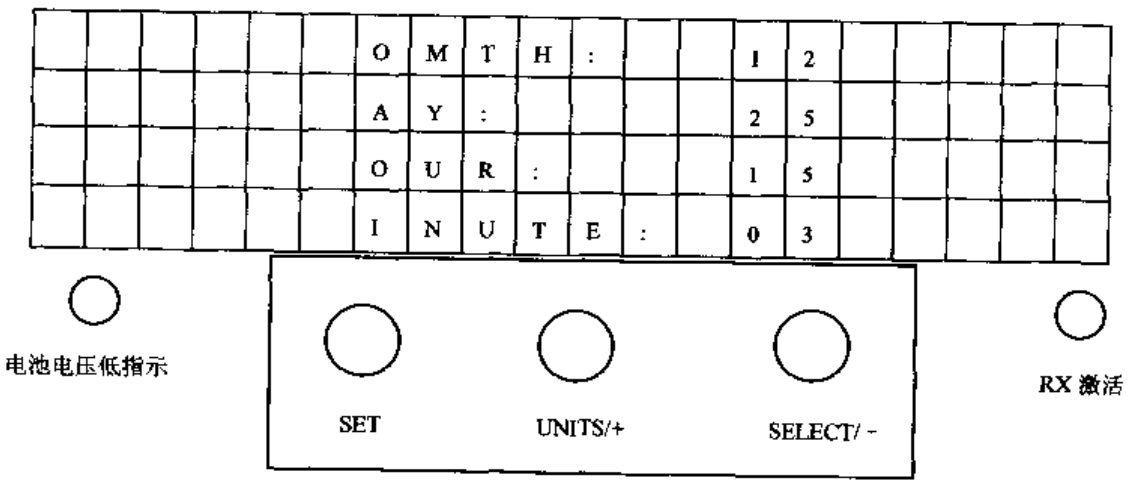


图 5-14 时间/日期编辑窗口

函数 set_time_date()是被 check_buttons()过程调用的，而后者又是被 main()调用的。前面已

经提过，显示维护和用户接口都是在最低级的层面上进行的。耗时的数据转换和显示操作都不会干扰设备的正常操作，但是在编辑过程中，实时时钟并不进行更新，它将忽略当时的数据采集过程，直到退出编辑模式。

5.6.12　系统测试阶段

这时候，就要执行在项目的测试定义阶段指定的系统测试，以使用户相信系统与说明一致。前面已经提过，测试可能是广泛的，因为要使用户信服；也可能是很简单的，只是和当地的气象台比较一下。

就本书而言，广泛的测试已经超出了要求的范围，而且这对扩展项目开发知识没有什么大的帮助。

但是，在这里必须提到的一个问题是，如果项目没有达到规格说明的要求，即系统测试失败，该怎么做。在系统测试中温度的结果没有校准，湿度的线性度太差，降雨量测量上的错误等问题都会在系统测试中成为缺陷。最重要的是怎么处理它，通常都会有两种选择：修正它或改变说明规定。改变说明是一个非常差的方法，如果前面的定义分析等都是正确的，是不应该采取这种方法的。所以实际上只有修正它一个选择了。

例如，就风速指示器来说，它计划中的校验是基于一个得自生产厂家的经验数据(2.515 Hertz/mph)，而且这个设计考虑到了结果中的一些变化，考虑了由于摩擦力，不同测量方法等引起的损失，以使风速能被正确地校准。

例如，假设在本地一个大学的风洞中测量了风速计的输出，发现有 4%的误差。设定的风速为 100mph，我们就是无法从系统中得到足够脉冲。所以我们画出的输出频率(Hz)与风速(mph)的关系就如图 5-15 中的“测量 A”线所示。

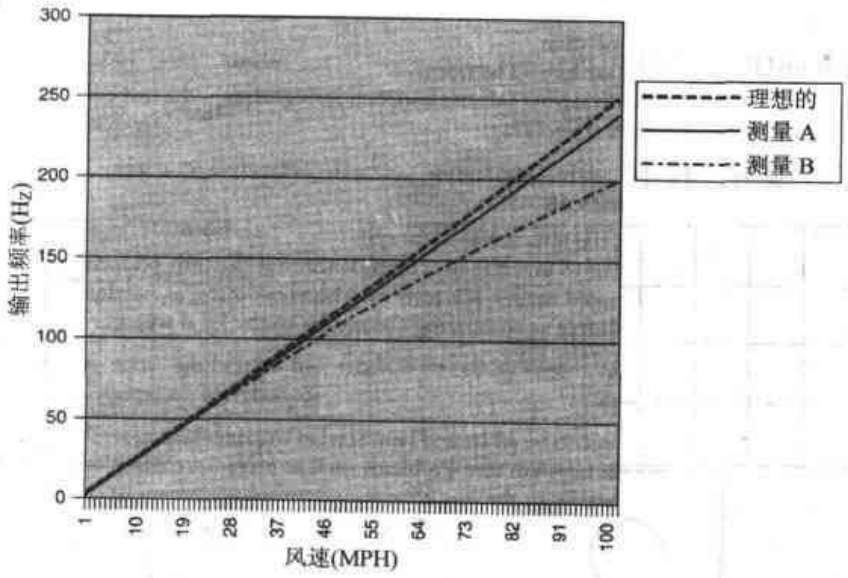


图 5-15　风速计测量输出

图中虽然有些误差，但是结果还是线性的。所以简单调整缩放比例就可以把它校正过来。这可能通过改变软件中的缩放比例常量来完成，从：

```
const int flash I_K+ws=25 ;  // scale wind speed ,Imperial
```

变为

```
const int flash I_K+ws=24 ;  // scale wind speed ,Imperial
```

这个简单的调整显示了把转换常量和修正因子组合成为一个常量存储到FLASH中的方法。这样就容易调整了，不用在程序中找出这些常量和比例因子。如果数据被用在多个地方时，可能会漏掉一两个。检验程序显示他们是被插入到程序中的，并且使用他们也没有增加或者减少代码的大小，所以他们是不需要任何开销。

如果结果是非线性的，如图5-15中的“测量B”线所示。可能需要一个更大的调整才能把它校正，可能需要一个复杂的代数表达式，或者类似查表的方法和插值法。这时候方法的选择很大程度上取决于要求的精度、结果的准确性、以及允许的计算时间。总的说来，查表(look-up-table，LUT)是一个不错的选择，比一些奇怪的数学运算好多了。在实际的系统中(就像我们正在讨论的)，在设备采集上来的数据中遇到一些形状奇怪的曲线一点都不奇怪。只要地球是圆的，地心吸引力还存在，太阳还是每天升起，并引起温度持续的变化，那么采集到的数据就会有形状，并且很可能它不是一条直线。

所以可以利用LUT和现行插值的方法，把“测量B”的数据转换为mph。LUT的一个最好的地方就是能利用实际上采集的数据。表5-7给出了测量的频率。

表5-7 “风洞”测试的测量频率

风速(mph)	输出频率(Hz)
1	2.5
10	24.6
20	48.3
30	68.7
40	92.6
50	113.3
60	132.9
70	151.5
80	169.2
90	185.8
100	201.5

如果数据比曲线或者S线更直，则表中的点数就可以少些。因为可以利用线性插值的方法补充两点之间的值，如果有需要。利用在风洞中测试频率的10倍，可以创建以下的数组来生成一个LUT。

```
const int flash Wfreqs[11] =     /* freq   X   10 , i.e. 24.6 = 246   */
{25,246,483,687,926,1133,1329,1515,1692,1858,2015 } ;
```

下面给出的函数，显示了如何把计数器检测到并放到变量w_speed中的频率与LUT中的值比较。如果速度低于LUT的下限，即小于1mph，就不用做任何计算。如果速度高于LUT的上

限，也不用做任何计算，并显示一个假的风速“888MPH”来显示有错误。

一旦从表中选出了两个数据，一个仅高于 w_speed 假设为 t，另一个仅低于 w-speed 的假设为 b，就计算两个值之间的增量(t－b)，以确定两者之间的一短线段，该线段是在一条长曲线中的一段有效直线段。从检测值 w_speed 到 b 的增量，是用来计算测量值会落在短线段的什么位置(百分比)。基本的 mph 是通过索引 x 来计算的，再加上由增量计算出来的那一部分，就可以得到实际的 mph。

```
void calc_wind_speed(void)   // compute global var wind_speed
{
        int t, b ,x;
        long v1, v2, v3;

        if(w_speed <= Wfreqs[0])
        {
              wind_speed = 0; //too slow to measure, < 1MPH
              return;
        }

        if(w_speed) > Wfreqs[10])
        {
              wind_speed = 888;   //too high to measure, > 100MPH
              return;             // "888" is an error message!!
        }

        for(x=0; x <9; x++)
        {
              if(w_speed > Wfreqs[x])   // find where the counts fall
              {                         // in the table
                    t = Wfreqs[x];
                    b = Wfreqs[x - 1];   //top and bottom values
                    break;               //x will be our base MPH/10
              }
        }
        v1 = (long)w_speed – (long)b;        // (note the 'casts')..
        v1 *= 100L;       // now calculate the percentage between
        v2 = (long)t – (long)b;              //the two values
        v3 = (v1/v2);         // percentage of MPH DIFF *10

        v1 = (long)x * 100L; // make x into MPH * 100 (fraction!)
        v1 += v3;// now add in percentage of difference

        wind_speed = (int) (v1 / 10L); //now scale to whole MPH
}
```

这样，在这个例子中，如果 w_speed 测量的结果是 1200。则在 for 循环的结果，索引 x 和

短线段两端的值 t 和 b，将会是：

```
x − 5
t = Wfreqs[5+1] =1329
b = Wfreqs[5] =1133
```

…步步深入程序，会发现跟着的计算会是：

```
v1 =(long)w_speed − (long)b ;    // (note the 'casts' )
```

结果：v1 = 1200–1133 = 67

```
v1 * =100L ;    // now calculate the percentage between
```

结果：v1 =67 ×100=6700

```
v2 =(long)t − (long)b ;    // (note the 'casts' )
```

结果：v2 = 1329 –1133 = 196

```
v3 = (v1/v2);  // percentage of MPH diff *10
```

结果：v3 =(6700/196)=34，也就是 10mph 的 34%，即 3.4mph

```
v1 * =(long)x*100L ;    // make x into MPH *100 (fraction!)
```

结果：v1 =5×100=500，就是 50.mph 的固定点

```
v1 +=v3 ;    //now add in percentage of difference
```

结果：v1 = 500 + 34 =534，就是 53.4mph 的固定点

```
wind_speed = (int)(v1/10L);   //now scale to whole MPH
```

最后：

```
wind_speed = 534 /10 =53 mph
```

您可能会想起来，在室内设备的规格说明中，只有整数部分的 mph 才能显示。另外，可以很容易看出利用这种方法能达到的精度水平。在上面的例子中，只是简单地丢掉了小数部分。在这种情况下，一个简单的 LUT 和线性插值法能非常有效地用来线性化一个非线性的结果，使项目能达到它的要求。

正如上面例子所示，系统测试的目的是为了确认系统在要求的程度上是符合规格说明的。在系统测试完成，并做了必要的调整以后，该项目就能够被使用了，并且结果具有很高的可信度。

5.7 挑战

向软件中加进新的特点或者改进原来的功能，都可以提高该天气监测器的可操作性和性能。以下列出了一些特点：

- 增加年和闰年的记录
- 增加在 5.6.7 小节中讨论过的天气报警功能。
- 求取风速和降雨量的平均值，使它们更稳定，更具有可读性。
- 在风寒和风露点的计算中采用线性插值法。
- 降雨量的追踪：不是以今天、这个月、今年的形式，而是使系统能跟踪过去 24 小时，过去 30 日和过去一年的数据。
- 跟踪和显示大气压的变化：上升，下降还是保持不变。
- 在软件中增加一个串行通信功能，把显示的信息通过串行端口和连接器 P3 发送到 PC 中。
- 使软件能利用 UART 接收从 PC 发往 INT0 引脚的命令。
- 增加记录功能，以便在峰值出现时跟踪风速、降雨量、温度和大气压的峰值。并允许用户在显示器上观察它们或者把它们下载到 PC 中。
- 使监测器的测量功能更独立些，以便时间/日期的编辑或者报警点的修改等操作不会阻止正常的监测操作。

5.8 本章小结

在本章中，把项目开发作为一个过程来讨论，事实上，顺着有序的步骤开发项目，通常会把您领向成功。本章在一个基于 Atmel Mega163 微控制器的天气监测器的开发过程中演示了开发的步骤。

5.9 练习

1. 列出项目开发的每一个步骤，并给出在这些步骤中可能出现的活动(小节 5.4)。
2. 在项目开发中的哪一步中，以下列出来的情况都发生了(小节 5.4)?
- 传感器的选型及其工作电路的设计
- 模拟电路的操作
- 为项目建立详细的规范说明
- 画出项目的草图
- 单独检验每个软件模块的功能
- 编写软件的程序流程图
- 根据规范说明来检测完成了的项目
- 在一个可疑的电路上做“构思验证”测试

3. 利用表 5-8 的数据，编写一个程序，进行查表和线性插值计算，然后返回一个值，作为

ADC 温度读数的补偿值(5.6.12 小节)。

表 5-8 线性插值练习

A/D 值，10 位(0～1023)	温度(°C)
123	-20
344	0
578	20
765	40
892	60
1002	80

5.10 上机实习

惟一适合本章的上机实习就是通过开发一个项目来演示项目开发的步骤。以下是一些项目建议。下面的任何一个主意，都可以通过使用显示器、额外的传感器或者输入设备来进行修改或者扩充。

- 一个能够在白色地板上沿着黑带走的机器老鼠。这个项目要求在能感应黑带的同时，能还控制“老鼠”的速度和驾驶操作
- 一个由微控制器和摄影机组成的设备，它可以监测简单的形状，例如在白色表面上由黑纸条组成的方形、矩形或者圆
- 一个利用电机和传感器跟踪热源的设备，它利用一个纸板箭头来指向一个热源。如果热源在房子里移动，箭头能一直跟踪着热源。这个设备需要很好地利用步进电机
- 一个炉子的温度调节装置
- 一个汽车的安全系统，利用传感器监测车门是否被打开了(包括油箱盖、车尾行李箱和引擎盖)并监测汽车何时被开动

附录A　库函数参考

C 语言强大的编程能力和应用的方便性在很大程度上要归功于它的内嵌(Built-in)函数，或称为库(Library)函数。这些函数可以为 C 程序员完成许多常见的任务。

对于所有的 C 函数，您可以在自己的代码中通过函数名来调用 C 函数库中的函数，并向函数传递参数或接收函数的返回值。要真正用好这些库函数，程序员需要了解应向函数传递什么参数和函数返回值所代表的含义。这些信息通常在函数原型(Prototype)中声明清楚。函数原型则被包含在一系列头文件(Head File)中，需要程序员在代码中将它们“包含(Include)”进来。

大多数 C 语言的编译器都提供了许多库函数，这些函数通常以函数组的形式出现。这种分组可以避免在每个程序中都包含一大堆无用的函数原型，而只需包含要使用的函数所在的函数组即可。例如，与标准输入输出例程有关的函数原型通常被放在一个名为 stdio.h 的头文件中。如果程序员想使用这些函数，将下面的语句放在程序代码的开头即可。

```
# include <stdio.h>
```

通过包含这个头文件，程序员就可以使用与标准输入/输出有关的函数了，如 printf 函数。

本参考将介绍 CodeVisionAVR 编译器自带的库函数。以下是一张按头文件名分组的库函数清单，以及按函数名字母排序的详细的函数描述和示例程序。

基于库文件名函数清单

bcd.h
 unsigned char bcd2bin(unsigned char n)
 unsigned char bin2bcd (unsigned char n)
ctype.h
 unsigned char isalnum(char c)
 unsigned char isalpha(char c)
 unsigned char isascii(char c)
 unsigned char iscntrl(char c)
 unsigned char isdigit(char c)
 unsigned char islower(char c)
 unsigned char isprint(char c)
 unsigned char ispunct(char c)
 unsigned char isspace(char c)
 unsigned char isupper(char c)
 unsigned char isxdigital(char c)
 unsigned char toint(char c)

```
 char tolower(char c)
 char toupper(char c)
 char toascii(char c)
delay.h
 void delay_us(unsigned int n)
 void delay_ms(unsigned int n)
gray.h
 unsigned char gray2binc(unsigned char n)
 unsigned int gray2bin (unsigned int n)
 unsigned long gray2binl(unsigned long n)
 unsigned char bin2grayc(unsigned char n)
 unsigned int bin2gray (unsigned int n)
 unsigned long bin2gray (unsigned long n)
math.h
 unsigned char cabs(signed char x)
 unsigned int abs(int x)
 unsigned long labs(long x)
 float fabs(float x)
 signed char cmax(signed char a, signed char b)
 int max(int a,int b)
 long lmax(long a,long b)
 float fmax(float a,float b)
 signed char cmin(signed char a, signed char b)
 int min(int a,int b)
 long lmin(long a,long b)
 float fmin(float a,float b)
 signed char csign(signed char x)
 signed char sign(int x)
 signed char lsign(long x)
 signed char fsign(float x)
 unsigned char isqrt(unsigned int x)
 unsigned int lsqrt(unsigned long x)
 float sqrt(float x)
 float floor(float x)
 float ceil(float x)
 float fmod(float x,float y)
 float modf(float x,float *ipart)
 float ldexp(float x,int expon)
 float frexp(float x,int *expon)
 float exp(float x)
 float log(float x)
 float log10(float x)
 float pow(float x,float y)
 float sin(float x)
 float cos(float x)
```

```
  float tan(float x)
  float sinh(float x)
  float cosh(float x)
  float tanh(float x)
  float asin(float x)
  float acos(float x)
  float atan(float x)
  float atan2(float y,float x)
 mem.h
  void pokeb(unsigned int addr,unsigned char data)
  void pokew(unsigned int addr,unsigned int data)
  unsigned char peekb(unsigned int addr)
  unsigned int peekw(unsigned int addr)
 sleep.h
  void sleep_enable(void)
  void sleep_disable(void)
  void idle(void)
  void powerdown(void)
  void powersave(void)
  void standby(void)
  void extended_standby(void)
 spi.h
  unsigned char spi(unsigned char data)
 stdio.h
  char getchar(void)
  void putchar(char c)
  void puts(char *str)
  void putsf(char flash *str)
  char *gets(char *str,unsigned int len)
  void printf(char flash *fmtstr)
  void sprintf(char *str,char flash *fmtstr)
  signed char scanf(char flash *fmtstr)
  signed char sscanf(char *str,char flash *fmtstr)
 stdlib.h
  int atoi(char *str)
  long atol(char *str)
  float atof(char *str)
  void itoa(int n,char *str)
  void ltoa(long n,char *str)
  void ftoa(float n,unsigned char decimals,char *str)
  void ftoe(float n,unsigned char decimals,char *str)
  void srand (int seed)
  int rand(void)
 string.h
  char *strcat(char *str1,char *str2)
```

```
char *strcatf(char *str1,char flash *str2)
char *strchr(char *str,char c)
signed char strcmp(char *str1,char *str2)
signed char strcmpf(char *str1,char flash *str2)
char *strcpy(char *dest,char *src)
char *strcpyf(char *dest,char flash *src)
unsigned char strcspn(char *str,char *set)
unsigned char strcspn(char *str,char flash*set)
unsigned iint strlenf(char flash*set)
char *strncat(char *str1, char *str2,unsigned char n)
signed char *strncmp(char *str1, char *str2,unsigned char n)
signed char *strncmpf(char *str1, char flash *str2,unsigned char n)
char *strncpy(char *dst,char *src,unsigned char n)
char *strncpyf(char *dst,char flash *src,unsigned char n)
char *strpbrk(char *str, char *set)
char *strpbrkf(char *str, char flash *set)
char strpos(char *str,char c)
char *strrchr(char *str,char c)
char *strrpbrk(char *str, char *set)
char *strrpbrkf(char *str, char flash *set)
signed char strrops(char *str,char c)
char *strstr(char *str1,char str2)
char *strstrf(char *str1,char flash str2)
unsigned char strspn(char *str,char *set)
unsigned char strspnf(char *str,char flash *set)
char *strtok(char *str1,char flash str2)
unsigned int strlen(char *str)
void memccpy(void *dst,void *src,char c,unsigned n)
void memchr(void *buf,unsigned char c,unsigned n)
signed char memcmp(void *buf1,void *buf2,unsigned n)
signed char memcmpf(void *buf1,void flash *buf2,unsigned n)
void *memcpy(void *dst,void *src,unsigned n)
void *memcpyf(void *dst,void flash *src,unsigned n)
void *memmove(void *dst,void *src,unsigned n)
void *memset(void *buf,unsigned char c,unsigned n)
```

abs

```
#include<math.h>
unsigned int abs(int x)
unsigned char cabs(signed char x)
unsigned long labs(long x)
float fabs(float x)
```

abs 函数返回整数 x 的绝对值，cabs，labs 和 fabs 函数分别返回有符号字符型，长整型和浮点型变量 x 的绝对值，返回类型分别为无符号字符型，无符号长整型和浮点型。

返回值：x 的绝对值，其表示数的范围由调用的函数的类型决定

```
#include <math.h>
void main()
{
        unsigned int int_pos_val;
        unsigned long long_pos_val;

        int_pos_val=abs(-19574);          //get absolute value of an integer
        int_pos_val=labs(-12500);         //get absolute value of a long

        while(1)
        {
        }
}
```

结果：　int_pos_val = 19574
　　long_pos_val = 125000

acos

```
#include<math.h>
float acos(float x)
```

acos 函数计算浮点型数 x 的反余弦函数值。返回值的结果在-π/2～π/2 之间，变量 x 的取值在-1～1 之间。

返回值：当 x 的值在-1～1 之间时，acos(x)的返回值范围在-π/2～π/2

```
#include <math.h>
void main()
{
        float new_val;
        new_val=acos(0.875);
        while(1)
        {
        }
}
```

结果：new_val = 0.505

asin

```
#include<math.h>
float asin(float x)
```

asin 函数计算浮点型数 x 的反正弦函数值。返回值范围在-π/2～π/2 之间，变量 x 的取值

必须在–1～1 之间。

返回值：当 x 的值在–1～1 之间时，asin(x)的返回值范围在–π/2～π/2 之间

```
#include <math.h>
void main()
{
        float new_val;
        new_val=asin(0.875);
        while(1)
        {
        }
}
```

结果：new_val = 1.065

atan

```
#include<math.h>
float atan(float x)
```

atan 函数计算浮点型数 x 的反正切函数值。返回值范围在–π/2～π/2 之间。

返回值：atan(x)的返回值范围在–π/2～π/2 之间

```
#include <math.h>
void main()
{
        float new_val;
        new_val=atan(1.145);
        while(1)
        {
        }
}
```

结果：new_val = 0.825

atan2

```
#include<math.h>
float atan2(float y, float x)
```

atan2 函数计算浮点型数 y/x 的反正切函数值。返回值范围在–π～π之间。

返回值：atan(y/x)的返回值范围在–π～π之间

```
#include <math.h>
void main()
{
        float new_val;
```

```
        new_val=atan2(1.145);
        while(1)
        {
        }
}
```

结果：new_val = 0.428

atof，atoi，atol

```
#include<stdlib.h>
float atof(char *str)
int atoi(char *str)
long atol(char *str)
```

atof 函数将 str 指向的 ASCII 码字符串转换为相应的浮点型数，而 atoi 函数和 atol 函数则分别将字符串转换为整型和长整型数。这 3 个函数在转换时都不会考虑字符串前端的空格字符(White Space Characters)的影响。待转换的字符串可以以+、–号开头，合法的数字字符为数字 0～9。对于 Atof 函数，小数点也是有效的数字字符之一。

当函数发现一个非空格字符，便开始把它转化为对应的数值，当第一次发现非数字字符时，转换便停止下来。如果没有发现数字字符，函数将返回 0。3 个函数都返回有符号的数值。

返回值：

- 当没有转换任何字符，atof，atoi 和 atol 函数将返回 0
- atof – str 指向的 ASCII 码字符串所对应的浮点数
- atoi – str 指向的 ASCII 码字符串所对应的有符号整数
- atol – str 指向的 ASCII 码字符串所对应的有符号长整型整数

```
#include <90s8515.h>
#include <stdio.h>              //this to include putchar and printf!
#include <stdlib.h>

/*quartz crystal frequency [hz]*/
#define xtal 7372000L

/*baud rate*/
#define baud 9600

#define MAX_ENTRY_LENGTH10
void main(void)
{
      char mystr[MAX_ENTRY_LENGTH+1];
      int myint;
      char c;
```

```
/*initialize the UART's baut rate*/
UBBR=xtal/16/baud - 1;

/*initialize the UART control register
RX&TX enable,no interrupts,8 data bits*/
UCR=0x18;

While(1)
{
        c=0
        putsf("Enter a signed integer number followed by !\n\r");
        while (c < MAX_ENTRY_LENGTH)
        {
        mystr [c++] = getchar ();     // wait for a character
        // if it is our terminating character, then quit!
        if (mystr[c-1] == '!')
                break;
        }
        mystr [c] = '\0';     // null terminate the string!

        myint = atoi (mystr);                    // convert

        printf("Your integer value is: %d\n\r", myint);
}
}
```

结果：UART 传送以下信息，

```
Enter a signed integer number followed by !
```

当收到“!”后，该字符串被转换并把结果传送给 UART。

bcd2bin

```
#include <bcd.h>
unsigned char bcd2bin(unsigned char n)
```

bcd2bin 函数将一个用二-十进制(bcd)表示的值转换为二进制值。按二-十进制码的编码规则，一个字节的前四位表示一个数的十位数值，后四位表示该数的个位数值。因此，参数 n 的值应在 0d～99d 之间。

返回值：n 的二进制数值

```
#include <bcd.h>
void main()
{
        unsigned char bcd_value, bin_value;
```

```
        bcd_val = 0x15;                    /*bcd representation of decimal 15*/
        bin_value = bcd2bin(bcd_value);
        while(1)
        {
        }
}
```

结果：bin_value = 15(= 0x0F)

bin2bcd

```
#include <bcd.h>
unsigned char bin2bcd(unsigned char n)
```

bin2bcd 函数将一个二进制值转换为用二-十进制码表示的值(bcd)。按二-十进制码的编码规则，一个字节的前四位表示一个数的十位数值，后四位表示该数的个位数值。因此，n 的值应在 0d～99d 之间。

返回值：n 的二-十进制编码数值

```
#include <bcd.h>
void main()
{
        unsigned char bcd_value,bin_value;
        bin_value=0x0F;                    /* 15 decimal */
        bcd_value=bin2bcd (bcd_value);
        while(1)
        {
        }
}
```

结果：bcd_value = 0x15

bin2grayc, bin2gray, bin2grayl

```
#include <gray.h>
unsigned char bin2grayc(unsigned char n)
unsigned char bin2gray(unsigned int n)
unsigned char bin2grayl(unsigned long n)
```

bin2gray 系列函数将一个用二-十进制码表示的值 n 转换为格雷码(Gray Coded，也称二进 进循环码)数值。当系统进行模数转换时，采用格雷码可以有效抑制系统的噪声。格雷码较之二进制编码的好处在于相邻的数据之间编码只有一位不同。bin2gray 系列函数，bin2grayc，bin2gray 和 bin2grayl 分别用来处理无符号字符型，无符号整型和无符号长整型数据。

表 A-1 列举了数字 0～15 的格雷码和对应的二-十进制码。

表 A-1　格雷码对应的十进制和二进制数值

十进制数字	二进制值 B_3 B_2 B_1 B_0	格雷码值 B_3 B_2 B_1 B_0
0	0 0 0 0	0 0 0 0
1	0 0 0 1	0 0 0 1
2	0 0 1 0	0 0 1 1
3	0 0 1 1	0 0 1 0
4	0 1 0 0	0 1 1 0
5	0 1 0 1	0 1 1 1
6	0 1 1 0	0 1 0 1
7	0 1 1 1	0 1 0 0
8	1 0 0 0	1 1 0 0
9	1 0 0 1	1 1 0 1
10	1 0 1 0	1 1 1 1
11	1 0 1 1	1 1 1 0
12	1 1 0 0	1 0 1 0
13	1 1 0 1	1 0 1 1
14	1 1 1 0	1 0 0 1
15	1 1 1 1	1 0 0 0

返回值：n 的二进制数值

```
#include <gray.h>
void main()
{
    unsigned char gray_char,bin_char;

    bin_char=0x07;
    gray_char=bin2gray(bin_char);
   while(1)
   {
   }
}
```

结果：gray_char = 4

cabs

```
#include <math.h>
unsigned char cabs(signed char x)
```

返回值：x 的绝对值

参见 abs

ceil

```
#include <math.h>
float ceil(float x)
```

ceil 函数返回不小于浮点型数 x 的最小整数值。换句话说，ceil 函数对 x 只入不舍，返回下一个整数值。

返回值：不小于浮点型数 x 的最小整数

```
#include <math.h>
void main()
{
    float new_val;

    new_val = ceil(2.531);

    while(1)
    {
    }
}
```

结果：new_val=3

cmax

```
#include <math.h>
signed char cmax(signed char a, signed char b)
```

返回值：a 和 b 之中的较大的一个数

参见 max

cmin

```
#include <math.h>
signed char cmin(signed char a, signed char b)
```

返回值：a 和 b 之中的较小的一个数

参见 min

cos

```
#include <math.h>
float cos(float x)
```

cos 函数计算浮点型数 x 的余弦函数值，角度 x 应该用弧度表示。

返回值：cos(x)

```
#include <math.h>
void main()
{
    float new_val;
    new_val = cos(5.121);
    while(1)
    {
    }
}
```

结果：new_val = 0.397

cosh

```
#include <math.h>
float cosh(float x)
```

cosh 函数计算浮点型数 x 的双曲余弦函数值，角度 x 应该用弧度表示。
返回值：cosh(x)

```
#include <math.h>
void main()
{
    float new_val;
    new_val = cosh(5.121);
    while(1)
    {
    }
}
```

结果：new_val = 83.754

csign

```
#include <math.h>
signed char csign(signed char x)
```

返回值：当 x 为负数、零或正数时，分别返回 -1、0 或 1
参见 sign

delay_ms

```
#include <delay.h>
void delay_ms(unsigned int n)
```

delay_ms 函数在返回之前产生一个 n 毫秒的延迟。在调用 delay_ms 函数时其他中断必须关闭，否则延迟时间将比预计的要长得多。而实际的延迟时间取决于右英钟频率，因此，选定恰当的时钟频率非常重要。时钟频率通过 Project|Configure|C Compiler 命令来指定；也可以在语

句#define xtal xL 中指定，其中 xL 是以赫兹为单位的时钟频率。

返回值：无

```
#include <delay.h>
#include <90s8515.h>
#include <stdio.h>

/*quartz crystal frequency [Hz]*/
#define xtal 7372000L

/*baud rate*/
#define baud 9600

void main(void)
{
      unsigned int pause_time;

      /*initialize the UART's baud rate*/
      UBRR=xtal/16/baud-1;

      /*initialize the UART control register
      RX & TX enabled,no interrupts,8 data bits*/
      UCR=0x18;
      while(1)
      {
      putsf("How many milliseconds should Ipause?(ENTER)\n\r");
      if(scanf("%u\n",&pause-time)!= - 1)
      {
            printf("Thanks!I'll pause for %u milliseconds.\n\r",pause_time);
            //disable interrupts
            #asm("cli");
            delay_ms(pause _time);
            //enable interrupts
            #asm("sei");
      }
      }
}
```

结果：微处理器传送以下信息：

```
How many milliseconds should I pause?(ENTER)
```

然后将等待，直到用户在下一行输入一个数字。一旦输入了(比如 2000)，它将显示：

```
Thanks! I'll pause for 2000 milliseconds.
```

于是经过一个适当的停顿(在例子中是 2 秒)，第一行提示又将被传送。

delay_us

```
#include <delay.h>
void delay_us(unsigned int n)
```

delay_us 函数在返回之前产生一个 n 微秒的延迟。n 必须是一个常量表达式。在调用 delay_us 函数时其他中断必须关闭，否则延迟时间将比预计的要长得多。而实际的延迟时间取决于右英钟频率，因此，指定恰当的时钟频率非常重要。这可以通过 Project|Configure|C Compiler 命令来指定；也可以在语句#define xtal xL 中指定，其中 xL 是以赫兹为单位的时钟频率。

返回值：无

```
#include <delay.h>
#include <90s8515.h>
#include <stdio.h>

/*quartz crystal frequency [Hz]*/
#define xtal 7372000L

/*baud rate*/
#define baud 9600

void main(void)
{
      unsigned int pause_time;

      /*initialize the UART's baud rate*/
      UBRR=xtal/16/baud-1;

     /*initialize the UART control register
     RX & TX enabled,no interrupts,8 data bits*/
      UCR=0x18;
      while(1)
      {
      putsf("How many thousands of useconds should I pause?\r");
      putsf("Enter a mumber 1 and 5.press ENTER.\n\r");
      if(scanf("%u\n",&pause-time)!= - 1)
      {
            if((pause_time>0)&&(pause_tme<6))
            {
                  printf("I'll pause %lu000 microseconds.\n\r",pause_time);
            }
            else
                  printf("Huh?\n\r");
            //disable interrupts
            #asm("cli");
```

```
                switch(pause_ time)
                {
                    case 1:
                        delay_us(1000);
                        break;
                    case 2:
                        delay_us(2000);
                        break;
                    case 3:
                        delay_us(3000);
                        break;
                    case 4:
                        delay_us(4000);
                        break;
                    case 5:
                        delay_us(5000);
                        break;
                    default:
                        break;
                    }
                    //enable interrupts
                    #asm("sei");
                }
            }
        }
```

结果：微处理器传送以下信息：

```
How many thousands of uSeconds should I pause?
Enter a number between 1 and 5.Press ENTER.
```

然后将等待，直到用户在下一行输入一个数字。一旦用户输入(比如 2)，它将显示：

```
I'll pause 2000 microseconds.
```

于是经过一个适当的停顿(在例子中是 2 毫秒)，第一行提示又将被传送。

exp

```
#include <math.h>
float exp(float x)
```

exp 函数计算变量 x 的自然指数值(e 为底数)。
返回值：e^x

```
#include <math.h>
void main()
{
```

```
        float new_val;
        new_val = exp(5);
        while(1)
        {
        }
    }
```

结果：new_val = e^5 = 148.413

extended_standby

```
#include <sleep.h>
void extended_standby(void)
```

extended_standby 函数使 AVR 微控制器进入扩展备用模式(extended standby mode)。这是一种与 powerdown 函数作用类似的休眠模式。

想了解函数的用法请参见本附录的 powerdown 函数说明。

这种休眠模式被应用于一种特殊的 AVR 设备，想了解这种模式的详细信息请参考 Atmel 数据表(datasheet)。应注意并不是所有的 AVR 设备都有扩展的备用休眠模式。

fabs

```
#include <math.h>
float fabs(float x)
```

返回值：x 的绝对值

参见 abs

floor

```
#include <math.h>
float floor(float x)
```

floor 函数返回浮点型数 x 的整数部分。即小于或等于 x 的最大整数。

返回值：x 的整数部分

```
#include <math.h>
void main()
{
    float new_val;
    new_val = floor(2.531);
    while(1)
    {
    }
}
```

结果：new_val = 2

fmax

```
#include <math.h>
float fmax(float a，float b)
```

返回值：a 和 b 之中的较大一个数
参见 max

fmin

```
#include <math.h>
float fmin(float a，float b)
```

返回值：a 和 b 之中的较小一个数
参见 min

fmod

```
#include <math.h>
float fmod(float x，float y)
```

fmod 函数返回 x 除以 y 的余数。这是一个专门为浮点型变量设计的模函数，模运算符%用于对整型变量进行取模运算。

返回值：x 除以 y 的余数

```
#include <math.h>
void main()
{
    float remains;
    remains = fmod(25.6，  8 );
    while(1)
    {
    }
}
```

结果：remains = 1.6

frexp

```
#include <math.h>
float frexp(float x，int *expon)
```

frexp 函数返回浮点型数值 x 的尾数值(mantissa)，如果 x 等于 0 则返回 0 。x 的二阶幂指数被存在指针 expon 所指的存储器地址中。如果 x 等于 0 ，则 expon 所指向地址的值也是 0 。

换句话说，当用如下的语句调用函数时：

```
y = frexp(x，&expon);
```

expon，x 和返回值 y 的关系可表示为：

$$x = y*2^{expon}$$

返回值：当 x 等于 0 则返回 0，当 x 在 0.5 和 1.0 之间则返回 x 的尾数

```
#include <90s8515.h>
#include <math.h>

void main(void)
{
      float x,z;
      int y;
      float a,c;
      int b;
      float d,f;
      int e;

      x=3.14159;
      z=frexp(x,&y);

      a=3.14159;
      z=frexp(a,&b);

      d=.707000;
      f=frexp(d,&e);

      while(1)
      {
      }
}
```

结果：

```
x = 3.141590
y = 2
z = 0.785398
a = 0.141590
b = –2
c = 0.566360
d = 0.707000
e = 0
f = 0.707000
```

fsign

```
#include <math.h>
```

```
signed char fsign(float x)
```

返回值：当 x 为负数、零和正数时，分别返回 - 1， 0 和 1

参见 sign

ftoa

```
#include <stdlib.h>
void ftoa(float n，unsigned char decimals，char *str)
```

ftoa 函数将浮点型数值 n 转换为对应的 ASCII 码字符串。该值是以指定位数的十进制表示的。转换后的字符串储存在*str 所指定的地址中。注意确保为 str 分配足够的空间，以便能容纳整个字符串加上空的结尾字符。

返回值：无

```
#include <90s8515.h>
#include <stdio.h>          //this to include putchar and printf!
#include <stdlib.h>

/*quartz crystal frequency [Hz]*/
#define xtal 7372000L

/*baud rate*/
#define baud 9600

void main(void)
{
      char mystr[11];

      /*initialize the UART's baut rate*/
      UBBR=xtal/16/baud-1;
      /*initialize the UART control register
      RX&TX enable,no interrupts,8 data bits*/
      UCR=0x18;
      ftoa(12470.547031,2,mystr);
      putsf("the floating point value is:");
      puts(mystr);

      while(1)
      {
      }
}
```

结果：UART 以 9600 波特率的速率传输，

```
The floating point value is:
                    12470.55
```

ftoe

```
#include <stdlib.h>
void ftoe(float n，unsigned char decimals，char *str)
```

ftoe 函数将浮点型数 n 转换为对应的 ASCII 码字符串。该值是以科学记数法表示的，即指定小数位数的尾数与 10 的整数次幂相乘的形式。转换后的字符串被存放在*str 所指定的地址中。注意为 str 分配足够的空间，以确保能容纳整个字符串加上空的结尾字符。

返回值：无

```
#include <90s8515.h>
#include <stdio.h>              // this to include putchar and printf!
#include <stdlib.h>

/*quartz crystal frequency [Hz]*/
#define xtal 7372000L

/*baud rate*/
#define baud 9600

void main(void)
{
      char mystr[11];
      /*initialize the UART's baut rate*/
      UBBR=xtal/16/baud-1;
      /*initialize the UART control register
      RX&TX enable,no interrupts,8 data bits*/
      UCR=0x18;
      ftoe(0.001247,2,mystr);
      putsf("the floating point value is:");
      puts(mystr);

      while(1)
      {
      }
}
```

结果：UART 以 9600 波特率的速率传输：

```
The floating point value is:
                        1.25e-3
```

getchar

```
#include <stdio.h>
char getchar(void)
```

getchar 函数是标准 C 语言的输入/输出函数，但对其做了修改使其能在系统资源有限的嵌入式微控制器中使用。getchar 函数返回 UART 接收的一个字符。该函数采用轮询的方式获取 UARL 接收的字符，因此该函数在返回前可能需要为接收的字符等待一段不确定的时间。

在函数使用之前，应保证 UART 已初始化，且其接收器处于可用的状态。

如果想使用另一外设来接收字符，只需对 getchar 函数做相应修改。该函数的源代码被放在 stdio.h 头文件中。

如果您想定义自己的 getchar 函数，替代标准的 getchar 库函数，需要做 3 件事。第一，必须在包含标准库之前将新的 getchar 函数放在 c 文件中。第二，必须在新的 getchar 函数后用一个定义语句，告诉编译器忽略随后发现的其他 getchar 函数。最后，将标准库包含进来。总之，代码应是如下的形式：

```
char getchar(void)
{
      //your getchar routine statements here
}

#define ALTERNATEGETCHAR_
#include <stdio.h>

//the rest of the source file!.
```

标准 getchar()函数的返回值：UART 接收的字符

```
#include <90s8515.h>
#include <stdio.h.

/*quartz crystal frequency [Hz] */
#define xtal 7372000L

/*Buad rate */
#define buad 9600

void main(void)
{
     char   k;
     /*initialize the UART's baud rate */
     UBBR=xtal/16/buad-1;

     /*initialize the UART control register
     RX & TX enable ,nointerrupts,8 data bits */
     UCR=0x18;

     while(1)
     {
```

```
            /* receive the character */
            k=getchar();
            /* and echo it back */
            putchar(k);
        }
    }
```

结果：UART 接收的字符从 UART 回显，直到处理器断电为止。

*gets

```
#include <stdio.h>
char *gets(char *str，unsigned int len)
```

*gets 函数使用 getchar 函数从 UART 中读取字符直到遇到换行符为止，并将它们放到 str 所指向的字符串中。换行符被 0 代替，同时返回指向 str 的指针。如果在遇到换行符之前已读取 len 个字符，函数将以 0 作为字符串的结尾字符并将其返回。

在函数被使用之前，UART 必须已被初始化，且 UART 接收器处于可用的状态。

返回值：指向字符串 str 的指针

```
#include <90s8515.h>
#include <stdio.h>

/* quartz crystal frequency [Hz] */
#define xtal 7372000L

/*Buad rate */
#define buad 9600

void main(void)
{
    char   your_name[11];          //room for 10 chars plus termination

    /* initialize the UART's baud rate */
    UBBR=xtal/16/buad-1;

    /* initialize the UART control register
    RX & TX enable ,nointerrupts,8 data bits */
    UCR=0x18;

    putsf("Please enter your name and press return.\r");
    putsf("(only 10 characters are allowed)\r");
    gets(your_name,10);    // up to 10 chars!
    printf("Hi %s!\n\r",your_name);
    while(1)
    {
```

```
        }
    }
```

结果：UART 发送一个输入名字的提示：

```
Please enter your name and press return.
(Only 10 characters are allowed.)
```

然后微处理器将进入等待状态，直到 UART 接收到 10 个字符或一个换行符。假定收到的是字符串 Jane Doe 和一个换行符，则会出现以下结果：

```
Hi Jane Doe!
```

gray2binc,gray2bin,gray2binl

```
#include <gray.h>
unsigned char gray2binc(unsigned char n)
unsigned char gray2bin(unsigned int n)
unsigned char gray2binl(unsigned long n)
```

gray2bin 系列函数将一个格雷码十进制值 n 转换为二进制值。当系统进行模数字转换时，采用格雷码可以有效抑制系统噪声。格雷码较之二进制编码的好处在于相邻的数据之间编码只有一位不同。gray2bin 系列函数，gray2binc，gray2bin 和 gray2binl 分别用来处理无符号字符型，无符号整型和无符号长整型数据。

参见表 A-1，该表列举了数字 0～15 的格雷码和对应的二-十进制码。

返回值：n 的二进制值

```
#include <gray.h>
void main()
{
    unsigned char gray_char,bin_charl

    gray_char=0x04; /* gray reresentation of decimal 7 */
    bin_char=gray2bin(gray_char);
    while(1)
    {
    }
}
```

结果：bin_char = 7

idle

```
#include <sleep.h>
void idle(void)
```

idle 函数使 AVR 微控制器进入空闲模式。在此函数被调用之前，必须先调用 sleep_enable

函数。在空闲模式下，CPU 停止运行，但计时器/计数器、watchdog 计时器和中断系统仍然继续运行中。这样，当出现内部或外部中断时，微控制器很容易被唤醒。当被中断唤醒后，微控制器(MCU)将执行中断，并执行(由 idle 函数发出的)休眠命令以后的指令。

返回值：无

参见 sleep_enable 函数说明的示例程序

isalnum

```
#include <ctype.h>
unsigned char isalnum(char c)
```

isalnum 函数测试参数 c，看它是否是数字字符。

返回值：若 c 是数字字符则返回 1

```
#include <ctype.h>
void main()
{
    unsigned char c_alnum_flag,d_alnum_flag;
    c_alnum_flag=isalnum('1');          //test the ASCII value of 10x31
    d_alnum_flag=isalnum(1);            //test the value 1
    while(1)
    {
    }
}
```

结果：

c_alnum_flag = 1

d_alnum_flag = 0

isalpha

```
#include <ctype.h>
unsigned char isalpha(char c)
```

isalpha 函数测试参数 c，看它是否是字母字符。

返回值：若 c 是字母字符则返回 1

```
#include <ctype.h>
void main()
{
    unsigned char c_alpha_flag,d_alpha_flag;
    c_alpha_flag=isalpha ('a');         //test the ASCII character of 'a'
    d_alpha_flag=isalpha ('1');         // test the ASCII character of '1'
    while(1)
    {
    }
```

```
}
```

结果：

c_alpha_flag = 1

d_alpha_flag = 0

isascii

```
#include <ctype.h>
unsigned char isascii(char c)
```

isascii 函数测试参数 c，看它是否是 ASCII 码字符。ASCII 码字符的范围是 0d～127d。返回值：若 c 是 ASCII 码字符则返回 1

```
#include <ctype.h>
void main()
{
   unsigned char c_ascii_flag,d_ascii_flag;
   c_alpha_flag=is ascii ('a');              //test the ASCII character 'a'
      d_alpha_flag=is ascii (153);              // test the the value '153'
   while(1)
   {
   }
}
```

结果：

c_ascii_flag = 1

d_ascii_flag = 0

iscntrl

```
#include <ctype.h>
unsigned char iscntrl(char c)
```

iscntrl 函数测试参数 c，看它是否是控制字符。控制字符的范围是 0d～31d 以及 127d。返回值：若 c 是控制字符则返回 1

```
#include <ctype.h>
void main()
{
      unsigned char c_cntrl_flag,d_cntrl_flag;
      c_cntrl_flag=iscntrl('\t');              //test the control character,horizontal tab
      d_cntrl_flag=iscntrl('a');               // test the ASCII character 'a'
      while(1)
      {
      }
}
```

结果：

c_iscntrl_flag = 1

d_iscntrl_flag = 0

isdigit

```
#include <ctype.h>
unsigned char isdigit(char c)
```

isdigit 函数测试参数 c，看它是否是以 ASCII 码表示的十进制数字。

返回值：若 c 是十进制数字则返回 1

```
#include <ctype.h>
void main()
{
    unsigned char c_isdigital_flag,d_isdigital_flag;
    c_isdigital_flag=isdigital('1');          // test the ASCII character 1
    d_isdigital_flag=isdigital('a');          // test the ASCII character 'a'
    while(1)
    {
    }
}
```

结果：

c_isdigit_flag = 1

d_isdigit_flag = 0

islower

```
#include <ctype.h>
unsigned char islower(char c)
```

islower 函数测试参数 c，看它是否是小写字母。

返回值：若 c 是小写字母则返回 1

```
#include <ctype.h>
void main()
{
    unsigned char c_islower_flag,d_islower_flag;
    c_islower_flag=islower('A');          // test the ASCII character A
    d_islower_flag=islower('a');          // test the ASCII character a
    while(1)
    {
    }
}
```

结果：

c_islower_flag = 1

d_islower_flag = 0

isprint

```
#include <ctype.h>
unsigned char isprint(char c)
```

isprint 函数测试参数 c，看它是否是可打印字符。可打印字符的范围是 32d～127d。

返回值：若 c 是可打印字符则返回 1

```
#include <ctype.h>
void main()
{
    unsigned char c_isprint_flag,d_isprint_flag;
    c_isprint_flag=isprint('A');            //test the ASCII character A
    d_isprint_flag=isprint('0x03');         // test the control character backspace
    while(1)
    {
    }
}
```

结果：

c_isprint_flag = 1

d_isprint_flag = 0

ispunct

```
#include <ctype.h>
unsigned char ispunct(char c)
```

ispunct 函数测试参数 c，看它是否是标点字符。所有的非控制字符和非字母数字字符都被视为标点字符。

返回值：若 c 是标点字符则返回 1

```
#include <ctype.h>
void main()
{
    unsigned char c_ispunct_flag,d_ispunct_flag;
    c_ispunct_flag=ispunct(',');            //test the ASCII character comma
    d_ispunct_flag=ispunct('\t');           // test the horizontal tab character
    while(1)
    {
    }
}
```

结果：

c_isprint_flag = 1

d_isprint_flag = 0

isqrt

```
#include <math.h>
unsigned char isqrt(unsigned int x)
```

返回值：无符号整型变量 x 的平方根

参见 sqrt

isspace

```
#include <ctype.h>
unsigned char isspace(char c)
```

isspace 函数测试参数 c，看它是否是空白空格字符。空格、换行符和横向制表符均被认为是空白空格字符。

返回值：若 c 是空白空格字符则返回 1

```
#include <ctype.h>
void main()
{
    unsigned char c_isspace_flag,d_isspace_flag;
    c_isspace_flag=isspace('A');            //test the ASCII character A
    d_isspace_flag=isspace('\t');           // test the horizontal tab character
    while(1)
    {
    }
}
```

结果：

c_isspace_flag = 0

d_isspace_flag = 1

isupper

```
#include <ctype.h>
unsigned char isupper(char c)
```

isupper 函数测试参数 c，看它是否是大写字母。

返回值：若 c 是大写字母则返回 1

```
#include <ctype.h>
void main()
{
    unsigned char c_isupper_flag,d_isupper_flag;
```

```
    c_isupper_flag=isupper('A');          // test the ASCII character A
    d_isupper_flag=isupper('a');          // test the ASCII character a
    while(1)
    {
    }
}
```

结果：

c_isupper_flag = 1

d_isupper_flag = 0

isxdigit

```
#include <ctype.h>
unsigned char isxdigit(char c)
```

isxdigit 函数测试参数 c，看它是否是以 ASCII 码表示的十六进制数。

返回值：若 c 是十六进制数字则返回 1

```
#include <ctype.h>
void main()
{
    unsigned char c_isxdigital_flag,d_isxdigital_flag;
    c_isxdigital_flag=isxdigital('a');          // test the ASCII character a
    d_isxdigital_flag=isxdigital('z');          // test the ASCII the character z
    while(1)
    {
    }
}
```

结果：

c_isxdigit_flag = 1

d_isxdigit_flag = 0

itoa

```
#include <stdlib.h>
void itoa(int n，char *str)
```

itoa 函数将有符号整数 n 转换成为对应的 ASCII 码字符串。转换后的字符串储存在*str 所指定的地址中。注意为 str 分配足够大的空间，以确保能容纳整个字符串加上空的结尾字符。

返回值：无

```
#include <90s8515.h>
#include <stdio.h>          //this to include putchar and printf
#include <stdlib.h>
```

```
/* quartz crystal frequency [Hz] */
#define xtal 7372000L

/* Buad rate */
#define buad 9600

void main(void)
{
      /* initialize the UART's baud rate */
      UBBR=xtal/16/buad-1;
      /* initialize the UART control register
      RX & TX enable ,nointerrupts,8 data bits */
      UCR=0x18;

       itoa(-1231,mystr);
       putsf("the value is:\r");
       puts(mystr);
      while(1)
      {
      }
}
```

结果：UART 以 9600 波特率的速率传输：

```
The value is:
-1231
```

labs

```
#include <math.h>
unsigned long labs(long x)
```

返回值：x 的绝对值
参见 abs

ldexp

```
#include <math.h>
float ldexp(float x，int expon)
```

ldexp 函数计算 x 与 2 的 expon 次幂的乘积。
返回值：$x*2^{expon}$

```
#include <math.h>
void main()
{
      float new_val;
```

```
        new_val = ldexp(5，3)；

        while(1)
        {
        }
    }
```

结果：new_val = $5*2^3 = 40$

lmax

```
#include <math.h>
long lmax(long a，long b)
```

返回值：a 和 b 之中较大的一个数
参见 max

lmin

```
#include <math.h>
long lmin(long a，long b)
```

返回值：a 和 b 之中较小的一个数
参见 min

log

```
#include <math.h>
float log(float x)
```

log 函数计算浮点型数 x 以 e 为底的对数(或称为自然对数)。x 必须是非零正数。
返回值：log(x)

```
#include <math.h>
void main()
{
        float new_val；

        new_val = log(5)；

        while(1)
        {
        }
    }
```

结果：new_val = 1.609

log10

```
#include <math.h>
float log10(float x)
```

log10 函数计算浮点型数 x 以 10 为底的对数。x 必须是非零正数。
返回值：log10(x)

```
#include <math.h>
void main()
{
    float new_val;
    new_val = log10(5);
    while(1)
    {
    }
}
```

结果：new_val = 0.699

lsign

```
#include <math.h>
signed char lsign(long x)
```

返回值：当 x 是负数、零或正数时，分别返回–1，0 和 1
参见 sign

lsqrt

```
#include <math.h>
unsigned int lsqrt(unsigned long x)
```

返回值：无符号长整型变量 x 的平方根
参见 sqrt

ltoa

```
#include <stdlib.h>
void ltoa(long n, char *str)
```

ltoa 函数将有符号长整型整数 n 转换成为对应的 ASCII 码字符串。转换后的字符串储存在*str 所指定的地址中。注意为 str 分配足够大的空间，以确保能容纳整个字符串加上空的结尾字符。

返回值：无

```
#include <90s8515.h>
#include <stdio.h>          //this to include putchar and printf!
#include <stdlib.h>
```

```
/* quartz crystal frequency [Hz] */
#define xtal 7372000L

/* Buad rate */
#define buad 9600

void main(void)
{
      char mystr[11];

      /* initialize the UART's baud rate */
      UBBR=xtal/16/buad-1;
      /* initialize the UART control register
      RX & TX enabled,no interrupts,8 data bits */
      UCR=0x18;

       ltoa(120031,mystr);
       putsf("The long signed integer value is:\r");
       puts(mystr);
      while(1)
      {
      }
}
```

结果：UART 以 9600 波特率的速率传输：

```
The long signed integer value is:
120031
```

max

```
#include <math.h>
int max(int a，int b)
signed char cmax(signed char a，signed char b)
long lmax(long a，long b)
float fmax(float a，float b)
```

max 函数返回一个整型数，其值为整数 a 和 b 之中的较大者。cmax，lmax 和 fmax 函数分别返回两个有符号字符型，长整型和浮点型数之中的较大者。

返回值：a 和 b 之中的较大者，返回值的取值范围由调用的函数决定

```
#include <math.h>
voi main()
{
     int big_int;
     signed char a_char,b_char,big_char;
```

```
    big_int=max(1200,3210);          //get the maximum of the values
    a_char=23;
    b_char=0x7A;
    big_char=cmax(a_char,b_char);
    while(1)
    {
    }
}
```

结果：

big_int = 3210

big_char = 0x7A

*memccpy

```
#include <string.h>
```

对于微型存储器(TINY Memory)模式：

```
void *memccpy(void *dst，void *src，char c，unsigned char n)
```

对于小型存储器(SMALL Memory)模式：

```
void *memccpy(void *dst，void *src，char c，unsigned int n)
```

memccpy 函数将数据从指针 src 所指向的存储器地址复制到指针 dst 所指的地址中，一次性复制的最大字节数为 n，且当复制完字符 c 后，复制过程将被终止。因此，dst 和 src 所指的存储器块不能有重叠。若函数复制的最后一个字符是 c，则函数返回一个空指针；如果 c 没有出现于所复制的 n 个字节的数据中，则函数将返回指向 dst 所指地址的下一个地址的指针(计算方法是 dst+n+1)。

返回值：如果 c 被复制到 dst 所指的地址中，则返回空指针；否则返回 dst+n+1

```
#include <string.h>

char inputstr[]="$11.2#";
char outputstr1[6];
char outputstr2[6];
void*a,*b;

void main(void)
{
    a=memccpy(outputstr1,inputstr,'.'3);
    b=memccpy(outputstr2,inputstr,'.'4);
    while(1)
    {
    }
}
```

结果：

outputstr1 =“$11”

a = outputstr1+3+1 = &(outputstr1[4])

outputstr2 =“$11.”

b = NULL

*memchr

```
#include <string.h>
```

对于微型存储器模式：

```
void *memchr(void *buf，char c，unsigned char n)
```

对于小型存储器模式：

```
void *memccpy(void *buf，char c，unsigned int n)
```

memchr 函数将从 buf 所指的存储器地址开始搜索，直到找到字符 c 或者搜索完 n 个字节的存储器空间。如果 buf 之后的 n 个字节之内能找到 c，则函数返回一个指向 c 的指针；如果没有找到 c，则返回一个空指针。

返回值：如果能找到 c，则返回指向 buf 中字符 c 出现位置的指针；否则返回 NULL

```
#include <string.h>

char inputstr[]="$11.2*";
char *a;
char *b;

void main(void)
{
    a=memchr(inputstr,'.',3);
    b= memchr(inputstr, '.',5);
    while(1)
    {
    }
}
```

结果：

a = NULL

b = &(inputstr[3])=".2#"

memcmp,memcmpf

```
#include <string.h>
```

对于微型存储器模式：

```
signed char memcmp(void *buf1，void *buf2，unsigned char n)
signed char memcmpf(void *buf1，void　flash *buf2，unsigned char n)
```

对于小型存储器模式：

```
signed char memcmp(void *buf1，void *buf2，unsigned int n)
signed char memcmpf(void *buf1，void flash *buf2，unsigned int n)
```

memcmp 和 memcmpf 函数比较指针 buf1 与 buf2 所指地址中的内容是否相同，一次比较一个字节。当发现某两个字节的内容不相同或已比较完 n 个字节，函数将返回。

返回值：

- 当*buf1<*buf2 时，　有符号字符型<0
- 当*buf1=*buf2 时，　有符号字符型=0
- 当*buf1>*buf2 时，　有符号字符型>0

```
#include<string.h>

char inputstr1[]="name a";
char inputstr2[]="name b";

void main(void)
{
    signed char a;
    signed char b;
    signed char c;

    a=memcmp(inputstr1,inputstr2,5);
    b=memcmp(inputstr1,inputstr2,6);
    c=memcmp(inputstr1,"name 1",6);
    while(1)
    {
    }
}
```

结果：

a = 0

b = oxFF

c = 0x30

*memcpy，*memcpyf

```
#include <string.h>
```

对于微型存储器模式：

```
void *memcpy(void *dst，void *src，unsigned char n)
```

对于小型存储器模式：

```
void *memcpy(void *dst，void *src，unsigned int n)
```

若 src 指向 FLASH 存储器(FLASH)地址，对两种模式都有：

```
void *memcpyf(void *dst，void *src，unsigned int n)
```

memcpy 和 memcpyf 函数将 n 个字节的数据从 src 所指的存储器地址复制到 dst 所指的地址中。对于 memcpy 函数，dst 和 src 所指的存储器块不能有重叠。而使用 memcpyf 函数则不需要考虑这个问题，因为 memcpyf 函数中 src 所指的地址一定在 FLASH 存储器中。如果存储器块确实有重叠，可以考虑使用 memmove 函数代替 memcpy 函数。

返回值：指向 dest 的指针

```
#include <string.h>

char inputstr[]="$11.2*";
char outputstr[6];
void *a;
char outputstrf[6];
void *b;

void main(void)
{
    a=memcpy(outputstr,inputstr+1,4);
    outputstr[4]='\0';                    //null terminate our new string
    b= memcpy(foutputstrf, "Hello World!",5);
    outputstrf[5]='\0';                   //null terminate our new string!
    while(1)
    {
    }
}
```

结果：

outputstr = “11.2”

a = “11.2”

outputstrf = “Hello”

b = “Helllo”

*memmove

```
#include <string.h>
```

对于微型存储器模式：

```
void *memmove(void *dest，void *src，unsigned char n)
```

对于小型存储器模式：

```
void *memmove(void *dest，void *src，unsigned int n)
```

memmove 函数将 n 个字节的数据从指针 src 所指的存储器地址复制到指针 dst 所指的地址中。与 memcpy 函数不同的是，调用 memmove 函数时，dst 和 src 所指的存储器块可以重叠。

返回值：指针 dest

```
#include <string.h.>

char inputstr1[]="abc1";
char*a

void main(void)
{
    //move the string one place to the right
    a=memmove (&(inputstr[1]),inputstr1,3);

    while(1)
    {
    }
}
```

结果：

a = “abc”

inputstr1 = “aabc”

*memset

```
#include <string.h>
```

对于微型存储器模式：

```
void *memset(void *buf，unsigned char c，unsigned char n)
```

对于小型存储器模式：

```
void *memcpy(void *buf，unsigned char c，unsigned int n)
```

memset 函数将 buf 所指的存储器地址后的 n 个字节都设置为字符 c。

返回值：指针 buf

```
#include <string.h.>

char inputstr1[]="abc1";
char *a;

void main(void)
```

```
{
    //starting after a,fill in with some 2's
    a=memset (&(inputstr[1]),'2',3);
    while(1)
    {
    }
}
```

结果：

a = "222"

inputstr1 = "a222"

min

```
#include <math.h>
int min(int a，int b)
signed char cmin(signed char a，signed char b)
long lmin(long a，long b)
float fmin(float a，float b)
```

min 函数返回一个整型数，其值为整数 a 和 b 之中的较小者。cmin，lmin 和 fmin 函数分别返回两个有符号字符型，长整型和浮点型数之中的较小者。

返回值：a 和 b 之中的较小者，返回值的取值范围由调用的函数决定

```
#include <math.h>
void main()
{
    int little_int;
    signed char a_char,b_char,little_char;

    little_int=min(1200,3210);          //get the minimun of the value
    a_char=23;
    b_char=0x7A;
    little_char=cmin(a_char,b_char);
    while(1)
    {
    }
}
```

结果：

little_int = 1200

little_char = 23

modf

```
#include <math.h>
```

```
float modf(float x，float *ipart)
```

modf 函数将浮点型数 x 拆分为整数和小数两个部分。x 的小数部分作为有符号浮点型数返回，而整数部分作为浮点型数被存放在指针 ipart 所指的存储器地址中。应注意是用变量的地址来存放整数部分，而不是把变量本身传递给 modf 函数。拆分后整数部分和小数部分均与 x 的符号相同。

返回值：

- 浮点型变量 x 的小数部分作为一个有符号浮点型数返回
- x 的整数部分被存放在*ipart 所指向的地址中

```
#include <math.h>

void main()
{
    float integer_portion，fract_portion;

    fract_portion = modf(–45.7，&integer_portion);

    while(1)
    {
    }
}
```

结果：

```
fract_portion = –.7
integer_portion = –45
```

peekb，peekw

```
#include <mem.h>
unsigned char peekb(unsigned int addr)
unsigned int peekw(unsigned int addr)
```

peekb 函数根据 addr 提供的地址，从静态随机存储器(SRAM)中读数据，而 peekw 函数根据 addr 提供的地址，从 SRAM 中读无符号整型数。peekw 函数先从 addr 所指的地址读出 data 的最低有效位(LSB)，然后再从 add+1 所指的地址读出 data 的最高有效位(MSB)。使用 pokeb 和 pokew 函数可将数据直接写入 SRAM 中。

返回值：无

参见 pokeb，pokew 的程序实例

pokeb，pokew

```
#include <mem.h>
void pokeb(unsigned int addr，unsigned char data)
void pokew(unsigned int addr，unsigned int data)
```

pokeb 函数根据 addr 提供的地址，把 datar 的值写入 SRAM 中，而 pokew 函数将整型数 data 写入 addr 地址指向的 SRAM 中。peekw 函数先把 data 的最低有效位写入 addr 所指的地址，然后再将 data 的最高有效位写入 add+1 所指的地址。使用 peekb 和 peekw 函数可从 SRAM 中读取数据。

返回值：无

```
#include <90s8515.h>
#include <mem.h>

/*the strucyure "alpha" is stored in SRAM at address 260h */
struct x {
        unsigned char b;
        unsigned int w;
}alpha @0x260;

void main()
{
        unsigned int read_b;
        unsigned int read_w;
        unsigned int read_b2;
        unsigned int read_w2;

        MCUCR=0xC0;          //enable external SRAM with 1 wait state

        alpha.b=0x11;        //initialize the value at 0x260
        alpha.w=0x2233;      //initialize the value at 0x261

        read_b=(unsigned int) peekb(0x260);
        read_w=peekw(0x261);

        pokeb(0x260,0xAA);          //place 0xAA at address 0x260
        pokew(0x261,0xBBCC);        //place 0xCC at address 0x261 and
                                    //0xBB at address 0x2662
        read_b2=(unsigned int) alfa.b;
        read_w2=alfa.w;

        while(1)
        {
        }
}
```

结果：以下是从 UART 中传送过来的运行结果。

```
read_b = 0x0011
read_w = 0x2233
```

```
read_b2 = 0x00AA
read_w2 = 0xBBCC
```

pow

```
#include <math.h>
float pow(float x，float y)
```

pow 函数计算 x 的 y 次幂

返回值：x^y

```
#include <math.h>

void main()
{
    float new_val;

    new_val = pow(2，5);
    while(1)
    {
    }
}
```

结果：new_val = 31.9

powerdown

```
#include <sleep.h>
void powerdown(void)
```

powerdown 函数使 AVR 微控制器进入断电状态。函数使用前，必须先调用 sleep_eable 函数。在断电状态下，外部振荡器停止工作，而外部中断和 watchdog(如果已被设为可用状态)仍然可以工作。只有外部的复位动作，如 watchdog(如果是可用的)的复位或外部中断，才能唤醒微控制器(MCU)。当微控制器被中断唤醒后，便开始处理该中断并执行(powerdown 函数发出的)休眠命令之后的指令。

返回值：无

```
#include <90s8515.h>
#include <stdio.h>
#include <sleep.h>
#include <delay.h>

/* quartz crystal frequency [Hz] */
#define xtal 6000000L

/* Baud rate */
#define baud 9600
```

```
interrupt [EXT_INT1] void int1_isr(void)
{
      putsf("I was interrupted!\r");
}

void main(void)
{
      PORTD=0xFF;              //turn on internal pull-up on pin3
      DDRD=0x00;               //make INT (PORTD pin 3) an input

      MCUCR=0x00;              //low level interrupt on INT1
      GIMSK=0x80;              //turn on INT1 interrupts
      /*initialize the UART's buad rate */
      UBRR=xtal/16/baud-1;

      /*initialize the UART control register
      RX & TX enabled,no interrupts,8 data bits */
      UCR=0x18;

      #asm("sei");             //enable interrupts
      sleep_enable();          //enable us to go to sleep when we are ready!

      putsf("reset Occurred\r");
      while(1)
      {
      putsf("Good Night!\r");
      putsf("I am going to sleep until you bug me!\r");
      delay_ms(100);                        //wait for string to be transmitted!
      Powerdown();             //enter powerdown until INT1 wakes us up
      }
}
```

结果：

微处理器先发送如下提示：

```
Reset Occurred
Good Night!
I am going to sleep until you bug me!
```

然后就进入断电休眠模式。当 PORTD 的引脚 3 由于低电压产生 INT1 中断，或者有一个外部复位时，微处理器将被唤醒。当它被唤醒后，便继续执行断电前尚未执行的指令，如果此时 PORTD 引脚 3 仍是低电压，它将发出如下提示：

```
I was interrupted!
```

接着微处理器将执行 while 循环，直到断电为止。

pwersave

```
#include <sleep.h>
void powersave(void)
```

powersave 函数使 AVR 微控制器进入省电模式。这是一种休眠模式，与 powerdown 函数类似。想了解函数的用法，参见本附录对 powerdown 函数的说明。

这种休眠模式只能被应用于一种特殊的 AVR 设备，想了解其详细内容请参考 Atmel 数据表(datasheet)。并不是所有的 AVR 设备都有省电休眠模式。

printf

```
#include <stdio.h>
void printf(char flash *fmtstr [, arg1, arg2, ...])
```

printf 函数根据 fmtstr 字符串中的格式说明符传送式化了的文本。传送是用 putchar 函数实现的。标准 putchar 函数默认用 UART 传送，不过也可以使用自定义的 putchar 函数更改数据传送的地址。具体方法请参阅 putchar 函数说明。

格式说明符 fmtstr 是一个必须存放在 FLASH 存储器中的常量。

printf 函数是标准 C 函数的简化版本。这是特别为嵌入式系统做的必要精简，因为如果函数完全按标准过程来运行，需要较大的存储器空间。

下面是 printf 函数可用的一些格式说明符：

%c	下一个参数是以 ASCII 码字符的形式输出的
%d	下一个参数是以十进制整数的形式输出的
%i	下一个参数是以十进制整数的形式输出的
%u	下一个参数是以无符号十进制整数的形式输出的
%x	下一个参数是以小写无符号十六进制整数的形式输出的
%X	下一个参数是以大写无符号十六进制整数的形式输出的
%s	下一个参数是以空字符为结尾的字符串的形式输出的，该字符串存放在 SRAM 中
%%	输出%字符

格式化规则：

- 所有的数值输出形式均为右对齐格式，左边位数不够加空格
- 如果在%和字母 d，i，u，x 或 X 之间插入字符“0”，则在输出数值的左边补 0
- 如果在%和字母 d，i，u，x 或 X 之间插入字符“-”，所有的数值输出形式均为左对齐格式
- 在%和字母 d，i，u，x 或 X 之间插入数字 1～9 作为宽度说明符，以限定输出数据的最小位数。数据的显示默认是右对齐的，如果在字符宽度说明符之前加一个“-”，则数值的输出形式将改为左对齐

返回值：无

```
#include <90s8515.h>
#include <stdio.h>
/* quartz crystal frequency [Hz] */
#define xtal 7372000L

/* Baud rate */
#define baud 9600

void main(void)
{
      unsigned int j;
      char c;

       /*initialize the UART's buad rate */
      UBRR=xtal/16/baud-1;

      /*initialize the UART control register
      RX & TX enabled,no interrupts,8 data bits */
      UCR=0x18;

      For(j=0;j<=500;j+=250)
      {
            //print the current value of j
            printf("Decimal:%u\tHexadecimal:%x\n\r",j,j);
            printf("Zero padded Decimal:%0u\n\r",j);
            printf("Four Digit Lower Case Hexadecimal:%04x\r\n\n",j);
      }

      while(1)
      {
            /* receive the character */
            c=getchar();
            /*and echo it back*/
            printf("The received character was %c\n\r", c);
      }
}
```

结果：

在系统启动时，微处理器向 UART 传递并输出以下信息：

```
Decimal: 0        Hexidecimal: 0
Zero Padded Decimal: 00000
Four Digit Lower Case Hexadecimal: 0000

Decimal: 250          Hexidecimal: FA
Zero Padded Decimal: 00250
```

```
Four Digit Lower Case Hexadecimal: 00fa

Decimal: 500          Hexidecimal: 1F4
Zero Padded Decimal: 00500
Four Digit Lower Case Hexadecimal: 01f4
```

此后，UART 接收到的任何字符都会按格式说明符的要求格式化后再传送回去。例如，假设接收到字符“z”，则如下的信息将被发送给传送器。

```
The received character was z.
```

putchar

```
#include <stdio.h>
void putchar(char)
```

putchar 函数是一个标准 C 语言输入/输出函数，但对其做了修改使之能在系统资源有限的嵌入式微控制器中使用。putchar 函数通过 UART 传送一个字符。该字符使用轮询的方式通过 UART 传送字符，因此该函数在返回值之前需要为发送的字符等待一段不确定的时间。

在函数被使用之前，应保证 UART 已被初始化，且其传送器应处于可用的状态。

如果想使用另一外设来传送字符，对 putchar 函数做相应修改即可。该函数的源代码被放在 stdio.h 头文件中。

如果您想定义自己的 putchar 函数，替代标准的 putchar 库函数，需要做 3 件事：第一，在包含标准库之前必须将一个新的 putchar 函数放在.c 文件中；第二，必须在新的 putchar 函数后用一个定义语句，告诉编译器忽略随后发现的其他 putchar 函数；最后，将标准库包含进来。总之，代码应是如下的形式：

```
void putchar(void)
{
    //your putchar routine statements here
}

#define _ALTERNATE_PUTCHAR_
#include <stdio.h>

// the rest of the source file!
```

标准的 putchar()函数返回值：无

```
#include <90s8515.h>
#include <stdio.h>

/*quartz crystal frequency [Hz] */
#define xtal 7372000L

/*Baud rate*/
```

```
#define baud 9600

void main(void)
{
    char k;

    /*initialize the UART's buad rate */
    UBRR=xtal/16/baud-1;

    /*initialize the UART control register
    RX & TX enabled,no interrupts,8 data bits */
    UCR=0x18;

    while(1)
    {
        //receive the character
        k=getchar();
        //and echo it back
        putchar(k);
    }
}
```

结果：UART 不断将接收到的字符回显回来，直到处理器断电。

puts

```
#include <stdio.h>
void puts(char *str)
```

puts 函数使用标准 putchar 函数或自定义的(如果定义了的话)putchar 函数来传送字符串 str。该字符串必须以空字符为结尾。当字符串传送全部传送完毕后，会被自动在结尾加上一个换行符。最后，字符串必须是在 SRAM 中(putsf 函数要求字符串被存放在 FLASH 存储器中)。默认状态下，putchar 函数用 UART 传送数据。

返回值：无

```
#include <90s8515.h>
#include <stdio.h>

/*quartz crystal frequency [Hz]*/
#define xtal 7372000L

/*Baud rate*/
#define baud 9600

char hello_str[]="Hello Word";
```

```
void main(void)
{
    /*initialize the UART's buad rate */
    UBRR=xtal/16/baud-1;

    /*initialize the UART control register
    RX & TX enabled,no interrupts,8 data bits */
    UCR=0x18;

    puts(hello_str);

    while(1)
    {
    }
}
```

结果：字符串“Hello World”通过 UART 传送，光标换了一行但没有回车。

putsf

```
#include <stdio.h>
void putsf(char *str)
```

putsf函数使用标准putchar函数或自定义的(如果定义了的话)putchar函数来传送字符串 str。该字符串必须以空字符为结尾，当字符串传送全部传送完毕后，会被自动在结尾加上一个换行符。最后，字符串必须被存储在 FLASH 存储器中(对于字符串被存放在 RAM 的情况请参见 puts 函数说明)。putchar 函数默认用 UART 传送数据。

返回值：无

```
#include <90s8515.h>
#include <stdio.h>

/*quartz crystal frequency [Hz]*/
#define xtal 7372000L

/*Baud rate*/
#define baud 9600

char flash hello_str[ ]="Hello World";
void main(void)
{
    /*initialize the UART's buad rate */
    UBRR=xtal/16/baud-1;

    /*initialize the UART control register
    RX & TX enabled,no interrupts,8 data bits */
```

```
        UCR=0x18;

        putsf("hello_str");

        while(1)
        {
        }
}
```

结果：UART 传送字符串“Hello World”，光标换了一行但没有回车。

rand

```
#include <stdlib.h>
int rand(void)
```

rand 函数返回一个 0～32 767 之间的伪随机数。
返回值：无

```
#include <90s8515.h>
#include <stdio.h>                  //this to include putchar and printf!
#include <stdlib.h>

/*quartz crystal frequency [Hz]*/
#define xtal 7372000L

/*Baud rate*/
#define baud 9600

void main(void)
{
      int seed;
      int rand_num;

       /*initialize the UART's buad rate */
      UBRR=xtal/16/baud-1;

      /*initialize the UART control register
      RX & TX enabled,no interrupts,8 data bits */
      UCR=0x18;

      putsf("Enter a seed value followed by ENTER.");
      scanf("%d\n", &seed);
      srand(seed);                  //seed the generator

      putsf("\n\rSend anything to a pseudorandom number!\n\r");
```

```
        while(1)
        {
            getchar();              //this will return when SOMETHING is received
            rand_num=rand();
            printf("%d\n\r", rand_num);
        }
    }
```

结果：UART 以 9600 波特率的速率传送以下信息：

```
Enter a seed value followed by ENTER.
```

一旦用户输入了一个数值，则打印以下信息：

```
Send anything to get a pseudorandom number!
```

当 UART 接收到另一个字符，又有一个随机数被传送出去。注意，如果以一个相同的“种子数”开始，那么您每次都会得到相同的随机数序列。

scanf

```
#include <stdio.h>
signed char scanf(char flash *fmtstr)
```

scanf 函数根据字符串 fmtstr 指定的格式来输入一个格式化文本，输入值的存放地址也由 fmtstr 指定。字符串 fmtstr 的格式说明部分用引号括起来，并通过下列格式化说明符告诉 scanf 函数读入数据的格式。fmtstr 的第二部分指定了读入数据存放的地址。

请注意！总是用指向变量的指针来接收 scanf 函数中 fmtstr 字符串的值是非常重要的，千万不要用变量本身来接收该字符串的值。如果不是将指针传递给 scanf 函数将导致不确定的后果，因为这会使由函数得到的值被存放在被传递的变量的值所指定的存储器地址中，而不是存放在变量本身的地址中。

接收数据是通过使用 getchar 函数来实现的。getchar 函数默认用 UART 接收，不过也可以自己定义 getchar 函数，以便从其他源来接收数据。具体细节请参阅 getchar 函数。

格式说明符 fmtstr 是一个必须存放在 FLASH 存储器中的常量。

scanf 函数是对应的标准 C 函数的简化版本。这是特别为嵌入式系统而做的必要精简，因为如果函数完全按标准过程来运行，需要较大的存储器空间。

下面是 scanf 函数可用的一些格式说明符：

%c　　下一个参数是以 ASCII 码字符的形式输入的
%d　　下一个参数是以十进制整数的形式输入的
%i　　下一个参数是以十进制整数的形式输入的
%u　　下一个参数是以无符号十进制整数的形式输入的
%x　　下一个参数是以小写无符号十六进制整数的形式输入的
%s　　下一个参数是以空字符为结尾的字符串的形式输入的

返回值：当输入过程中有错误发生时返回- 1，否则返回成功接收数据的入口地址

```
#include <90s8515.h>
#include <stdio.h>

/*quartz crystal frequency [Hz]*/
#define xtal 7372000L

/*Baud rate*/
#define baud 9600

void main(void)
{
      char your_initial;
      int your_number;

       /*initialize the UART's buad rate */
      UBRR=xtal/16/baud-1;

      /*initialize the UART control register
      RX & TX enabled,no interrupts,8 data bits */
      UCR=0x18;

      while(1)
      {
            putsf("\n\r");
            putsf("Please enter your first initial followed by \r");
            putsf("a comma then your favorite number.\r");
            putsf("Press ENTER to finish.\r");
            //tell scanf what to look for
            //NOTICE the ADDRESS of the variables are sent to scanf!
            If(scanf("%c,%d\n",&your_initial,&your_number)==2)
            {
                  printf("%c,congrates! You got a %d%% on your exam!\n\r", your_initial, your_number);
            }
            else
            {
                  putsf("I didn't understand.Please try again.\n\r");

            }

      }
}
```

结果：UART 发出需要首字母和数字的提示：

```
Please enter your first initial followed by
a comma then your favorite number.
Press ENTER to finish.
```

于是 scanf 函数等待接收首字母，逗号，数字和换行符。如果用户输入

```
S，32
```

则显示以下信息：

```
S，congrats！ You got a 32% on your exam！
```

如果用户输入：

```
S，AC
```

则发生输入错误，于是会出现提示：

```
I didn't understand. Please try again.
```

sign

```
#include <math.h>
signed char sign(int x)
signed char csign(signed char x)
signed char lsign(long x)
signed char fsign(float x)
```

sign 函数返回整数 x 的符号标记。当 x 的符号是负的，返回–1；当 x 等于零时，返回 0；当 x 是正数时，返回 1。csign，lsign 和 fsign 函数分别返回有符号字符型，长整型和浮点型变量 x 的符号标记。

返回值：

- 当 x 是负数时返回–1
- 当 x 等于零时返回 0
- 当 x 是正数时返回 1

```
#include<math.h>
void main()
{
    signed char pos_sign;
    signed char neg_sign;
    signed char zero_sign;

    neg_sign=sign(-19574);    //get the sign of an integer
    pos_sign=lsign(12500);    //get the sign of a long
    zero_sign=csign(0);       //get the sign of a char

    while(1)
```

```
        {
        }
    }
```

结果：

pos_sign = 1

neg_sign = –1

zero_sign = 0

sin

```
#include <math.h>
float sin(float x)
```

sin 函数计算浮点型数 x 的正弦函数值，角度 x 应该用弧度表示。

返回值：sin(x)

```
#include <math.h>
void main()
{
    float new_val;
    new_val = sin(5.121);
    while(1)
    {
    }
}
```

结果：new_val = –0.918

sinh

```
#include <math.h>
float sinh(float x)
```

sinh 函数计算浮点型数 x 的双曲正弦函数值，角度 x 应该用弧度表示。

返回值：sinh(x)

```
#include <math.h>
void main()
{
    float new_val;
    new_val = sinh(5.121);
    while(1)
    {
    }
}
```

结果：new_val = 83.748

sleep_disable

```
#include <sleep.h>
void sleep_disable(void)
```

sleep_disable 函数将 MCUCR 寄存器的 SE 位清零。这样可以防止微控制器因为偶然原因进入休眠模式。

返回值：无

```
#include <90s8535.h>
#include<stdio.h>
#include<sleep.h>
#include<delay.h >

/*quartz crystal frequemcy [Hz]*/
#define xtal 6000000L

/*Baud rate*/
#define Baud 9600

interrupt [EXT_INT1] void intl_isr(void)
{
        putsf("I was interrupted!\r ");
}

void main(void)
{
      PORTD=0XFF;        //turn on internal pull-up on pin3
      DDRD=0x00;         //make INT1(PORTD pin 3)an input

      MCUCR=0X00;        //low level interrupt on INT1
      GIMSK=0X80;        //turn on INT1 interrupts

      /*initialize fhe UART's baud rate*/
      UBRR=xtal/16/baud-1;

      /*initialize fhe UART control register
      RX&TX enable ,no interrupts 8 date bits */
      UCR=0x18;

      #asm("sei");       //enable interrupts
      sleep_enable();    //enable us to go to sleep when we are ready!

      Putsf("Reset Occurred\r");
```

```
    While(1)
    {
        Putsf ("Good Night!\r")
        Putsf("I am going to sleep until yor bug me!\r");
        delay_ms(100);   //wait for string to be tramsmitted!
        idle();          //enter idle until INT1wakes us up
        sleep_disable();  //stop future sleeps
    }

}
```

结果：微处理器首先传送，

```
Reset Occurred
Good Night!
I am going to sleep until you bug me!
```

然后进入空闲休眠模式。当 PORTD 引脚 3 获得一个低电压产生 INT1 中断，或者有一个外部复位，微处理器将被唤醒。在它被唤醒后，便继续执行进入空闲休眠模式前尚未执行的指令，如果此时 PORTD 引脚 3 仍是低电压，它将发出如下提示：

```
I was interrupted!
```

由于下一行程序使微处理器无法进入休眠模式。因此，尽管微处理器一直调用 idle 函数试图休眠，却毫无效果。这样，微处理器将持续显示上面的提示直到断电为止。

sleep_enable

```
#include <sleep.h>
void sleep_enable(void)
```

sleep_enble 函数将 MCUCR 寄存器的 SE 位。执行 SLEEP 指令时，该位必须设置为允许微控制器进入休眠模式。在休眠模式下，微控制器的一些功能会被停止，以减少电能消耗。根据休眠方式的不同，微控制器可以被内部或外部的中断唤醒。一些特定的微控制器还可以采用一些特殊的休眠方式。想了解休眠方式及其适用的微控制器的详细信息，请参考 Atmel AVR 数据表。

返回值：无

```
#include <90s8515.h>
#include <stdio.h>
#include<sleep.h>
#include<delay.h>

/*quartz crystal frequency[Hz] */
#include xtal 7372000L
```

```
/*Baud rate */
#define baud 9600

void main(void)
{
    PORTD=0x04;          //turn on internal pull-up on pin2
    DDRD=0XFB;           //make INTO(PORTD pin 2)an input

    MCUCR=0X00;          //low level interrupt on INT0
    IFT=0X40;            //turn on INT0 interrupts

    /*initialize the UART's baud rate*/
    UBRR=xtal/16/baud-1;

    /*initialize the UART control register
    RX&Txenabled,no interrupts,8 data bits */
    UCR=0x18;

    #asm("sei");         //enable interrupts
    sleep_enable();      //enable us to go to sleep when we are ready!


    Putsf("Reset occurred");
    while(1)
    {
        putsf("Good Night!");
        putsf("I am going go sleep until you bug me!");
        delay_ms(100);
        idle();   //emter idle until INTO wakes us up
        putsf("I was interrupted!");
        sleep_disable();     //stop future sleeps

    }
}
```

结果：微处理器首先发出如下提示，

```
Reset Occurred
Good Night!
I am going to sleep until you bug me!
```

然后进入空闲休眠模式，当 PORTD 引脚 2 获得一个低电压产生 INT0 中断，或者有一个外部复位，微处理器将被唤醒。当它被唤醒后，便继续执行进入空闲休眠模式前尚未执行的指令，并且发出如下提示：

```
I was interrupted!
```

随后，微处理器将接着运行 while 循环，直到断电为止。

spi

```
#include <spi.h>
unsigned char spi(unsigned char data)
```

串行外设接口(SPI)允许在设备间同步高速传输数据。许多 Atmel AVR 设备都有硬件 SPI 端口。SPI 端口通过以下 3 种连接就能实现同时发送和接收数据：时钟、主入从出(MISO)和主出从入(MOSI)。主设备先产生一个时钟信号，然后将数据放在 MOSI 引脚与该时钟信号同步送出。从设备也与时钟同步，将数据放在 MISO 引脚。这样，就实现了在主设备和从设备同时发送和接收数据。

假如微控制器是主设备，一旦 SPI 端口被初始化，便通过将数据放入 SPI 数据寄存器中开始数据传输过程；否则，微控制器必须等待主设备发出时钟信号再开始传送数据。当数据传输结束后，SPI 数据寄存器中便存放从其他设备中读取的数据。关于如何初始化 SPI 端口请参阅 Atem AVR 数据表。

SPI 系列函数的目标是在 C 程序和使用 SPI 总线的不同外部设备之间提供一个友好的接口。spi 函数先将字节数据加载到 SPI 数据寄存器中，然后等待硬件设备完成同步数据输出并从外部设备中读取数据。当数据传输完成后，spi 函数将返回 SPI 数据寄存器中的值。由于 spi 函数以轮询的方式来判定数据传输是否完成，因此在调用该函数时不需要激活 SPI 中断。

返回值：当数据传输完成后，返回从 SPI 数据寄存器中读出的字符

```
#include <90s8515.h>
#include <delay.h>
#include<spi.h>
#include<stdio.h>

//This example reads the status from am Armel AT45D081 Flash
//memory chip
/over SPI
#define xtal 7372000L

int stats;

void main(void)
{
        char junk;
        PORTA=0XFF;
        PORTB=0x00;
        PORTC=0x00;          //clear port registers
        PORTD=0x10;

        DDRA=0X03;          //all inputs ,except the lights!
        DDRB=0XBF;          //all outputs ,except MISO pin!
```

```
DDRC=0X03;        //all inputs ,except for bit 0 and 1
DDRD=0XFF;        //all outputs

/*Serial Peripheral interface setup*/
SPCR=0x5E;

SREG=0x00;          //Disable interrupts
GIMSK=0x00;         //no external interrupts

/*UART*/
UBRR=7;               //576kb 0% error with 7.37228 MHz clock
UCR=0X98;             //UART TX ,RX,8bit+SS,
Junk=UDR;             //force a clear of this register and
                       //associated interrupt flags
delay_ms(2000);       //wait for things to get stable

PORTD.4=0;          pull the chip select line low
Delay_ms(5);
//to get status send command 0x57   0x00
//status is reruned while the second byte of the command is being sent
spi(0x57);                    //byte 1
stats=(int) spi (0x00);       //byte2

PORTD.4=1;          //release the clip select line
Printf("Status: %x\n\r",Stats);

while(1)
{
}

}
```

结果：微处理器在启动时以 57 600 波特的速率传输：

```
Status:A0
```

sqrt

```
#include <math.h>
float sqrt(float x)
unsigned char isqrt(unsigned int x)
unsigned int lsqrt(unsigned long x)
```

sqrt 函数返回一个浮点型变量，该数为正的浮点型变量 x 的平方根。isqrt 和 lsqrt 函数则分别返回无符号整型和无符号长整型变量 x 的平方根。应注意 isqrt 和 lsqrt 函数的返回值分别是字符型和整型的，相对于原来是整型和长整型的变量 x 来说，可表示数据的范围缩小了。

返回值：

sqrt — 正的浮点型变量 x 的平方根

isqrt — 无符号整型变量 x 的平方根，该值是无符号字符型数

lsqrt — 无符号长整型变量 x 的平方根，该值是无符号整型数

```
#include<math.h>
void main()
{
   unsigned char my_i_sgrt
   unsigned int my_i_sqrt;
   float my_f_sqrt;

   my_f_ sqrt= sqrt(6.4);              //get the square root of a float
   my_f_ sqrt=lsqrt(250000);           // get the square root of a long value
   my_f_ sqrt= sqrt(81);               // get the square root of an int value

   while(1)
   {
   }
}
```

结果：

my_f_sqrt = 2.503

my_l_sqrt = 500

my_i_sqrt = 9

srand

```
#include <stdlib.h>
void srand(int seed)
```

srand 函数为为伪随机数生成器 rand 设置随机种子数。

返回值：无

参见 rand 函数说明中的示例程序

sscanf

```
#include <stdio.h>
signed char sscanf(char * str，char flash *fmtstr)
```

sscanf 函数根据字符串 fmtstr 指定的格式，从 SRAM 中的文本字符串 str 接收输入，输入值的存放地址也由 fmtstr 指定。格式说明符字符串 fmtstr 的格式说明部分用引号括起来，并通过下列格式化说明符告诉 sscanf 函数读入数据的格式。fmtstr 的第二部分指定了读入数据存放的地址。

请注意！总是用指向变量的指针来接收 sscanf 函数中 fmtstr 字符串的值是非常重要的，千

万不要用变量本身来接收该字符串的值。如果不是将指针传递给 scanf 函数将导致不确定的后果，因为这会使由函数得到的值被存放在被传递的变量的值所指定的存储器地址中，而不是存放在变量本身的地址中。

格式说明符 fmtstr 是一个常量且必须存放在 FLASH 存储器中。

sscanf 函数是对应的标准 C 函数的简化版本。这是特别为嵌入式系统中使用而做的必要精简，因为如果函数完全按标准过程来运行，需要较大的存储器空间。

下面是 sscanf 函数可用的一些格式说明符：

%c	下一个参数是以 ASCII 码字符的形式输入的
%d	下一个参数是以十进制整数的形式输入的
%i	下一个参数是以十进制整数的形式输入的
%u	下一个参数是以无符号十进制整数的形式输入的
%x	下一个参数是以小写无符号十六进制整数的形式输入的
%s	下一个参数是以空字符为结尾的字符串的形式输入的

返回值：当输入过程中有错误发生时返回–1，否则返回成功接收数据的入口地址

```
#include <90s8515.h>
#include<stdio.h>

/*quartz crystal frepuency [Hz]*/
# define xtal 7372000L
/*baud rate*/
#define baud 9600

char SOFTWAER_VERSION[ ] = "3.5b";

void main(void)
{
      char version_letter;
      int major_version;
      int minor_version;
      char results;

      /*initialize the UART's baud rate */
      UBRR=xtal/16/baud–1;

      /*initialize the UART control register
      RX &TX enabled, no interrupts,8 data bits*/
      UCR=0x18;

      //tell sscanf what fo look for
      //NOTICE the ADDRESSES of the variables are sent to sscanf!
      Results=sscanf(SOFTWARE_VERSION , "&d,&d&c",&major_version,
            &minor_version, &version_letter);
```

```
    if(results!=-1)
    {
        printf
        ("major Version:&d,Minor Version %d,Lerrer &c.\n\r",
        major version,minorversion,version_letter);
    }
    else
        putsf("An error occurred. Something is not right!\r");

    while(1)
    {
    }
}
```

结果：UART 发送以下信息：

Major Version：3，Minor Version 5，Letter b.

standby

```
#include <sleep.h>
void standby(void)
```

standby 函数使 AVR 微控制器进入备用模式。这是一种休眠模式，与 powerdown 函数类似。想了解函数的用法请参见本附录的 powerdown 函数。

这种休眠模式被应用于一种特殊的 AVR 设备，想了解其详细内容请参考 Atmel 数据表。并不是所有的 AVR 设备都有备用休眠模式。

*strcat，*strcatf

```
#include <string.h>
char *strcat(char *str1，char *str2)
char *strcatf(char *str1，char flash *str2)
```

strcat 和 strcatf 函数将字符串 str2 连接在字符串 str1 的末尾。因此必须为 str1 分配足够大的存储器，以便能容纳连接后的新的、更长的字符串加上空的结尾字符，否则将会出现难以预料的后果。在 strcatf 函数中，str2 所指的字符串必须存放在 FLASH 中。函数的返回值是指向 str1 的指针。

返回值：*str1(指向字符串 str1 和 str2 连接后的字符串的指针，该字符串以空字符结尾)

```
#include <90s8515.h>
#include <stdio.h>
#include <string.h>

/*quartz crystal frepuency [Hz]*/
# define xtal 7372000L
```

```
/*baud rate*/
#define baud 9600

void main(void)
{
    char stra[]="abc";
    char strb[10];
    /*initialize the UART's baud rate */
    UBRR=xtal/16/baud-1;

    /*intialize the UART control register
    RX &TX enabled, no interrupts,8 data bits*/
    UCR=0x18;

    strcpy(strb,"xyz");   //initialize the strb
    strcat(strb,stra);    //add stra to strb
    strcatf(strb,"123");  //add a flash string too !
    puts(strb);           //put it out to the UART!
    while(1)
    {
    }
}
```

结果：UART 以 9600 波特率的速率发送以下信息，

```
xyzabc123
```

*strcpy，*strcpyf

```
#include <string.h>
char *strcpy(char *dest，char *src)
char *strcpyf(char *dest，char flash *src)
```

strcpy 和 strcpyf 函数将 src 所指向的字符串复制到 dest 所指的地址中，并以空字符结尾。因此，必须为 dest 分配足够大的存储器，以便能容纳整个 src 字符串加上空的结尾字符。

返回值：dest 的指针

```
#include <90s8515.h>
#include <stdio.h>
#include <string.h>

/*quartz crystal frepuency [Hz]*/
# define xtal 7372000L

/*baud rate*/
#define baud 9600
```

```
void main(void)
{
    char stra[]="hello";
    char strb[6];
    char strc[6];

    /*initialize the UART's baud rate */
    UBRR=xtal/16/baud-1;

    /*initialize the UART control register
    RX &TX enabled, no interrupts,8 data bits*/
    UCR=0x18;

    strcpy(strb, stra);        //copy stra to strb
    strcpy(strc,"world");      //copy "world" to strc

    puts(strb);                //transmit strb
    puts(strc);                //transmit strc
    while(1)
    {
    }
}
```

结果：UART 以 9600 波特率的速率发送以下信息，

```
Hello
    World
```

两行之间的回车符是调用 puts 函数产生的。注意 puts 函数在将一个字符串传送给 UART 时会在它的结尾加上一个回车符。

strcspn，strcspnf

```
#include <string.h>
unsigned char strcspn(char *str，char *set)
unsigned char strcspnf(char *str，char flash *set)
```

strcspn 和 strcspnf 函数返回字符串 str 中第 1 个与字符串 set 中的某个字符匹配的字符的索引。如果 str 中的字符都不在 set 中，则返回字符串 str 的长度(字符个数)；如果 str 中的第 1 个字符就在 set 中，则返回 0。对于 strspnf 函数，字符串 set 必须放在 FLASH 存储器中。

返回值：字符串 str 中第 1 个与字符串 set 中的某个字符匹配的字符的索引

```
#include <<string.h>

void main(void)
{
    char set[]=" 1234567890-()";
```

```
    char stra[]="1.800.555.1212";
    char index_1;
    char index_2

    index_1=strcspn(stra,set);
    index_1=strcspn(stra,".-()");

    while(1)
    {
    }
}
```

结果：

index_1 = 0

index_2 = 1

strlen，strlenf

```
#include <string.h>
```

对于微型存储器模式：

```
unsigned char strlen(char *str)
```

对于小型存储器模式：

```
unsigned int strlen(char *str)
```

对两种模式都有：

```
unsigned int strlenf(char flash *str)
```

strlen 函数和 strlenf 函数返回字符串 str 的长度，但不包括空的结尾字符。对于 strlen 函数，若在微型存储器模式下使用，能计算的字符串的长度在 0～255 之间。若在小型存储器模式下调用该函数，返回的字符串长度可以是0～65 535。而对于strlenf函数来说，字符串必须放在FLASH存储器中，且不管在哪种模式下调用，能计算的范围都在 0～65 535 之间。

返回值：字符串 str 的长度

```
#include <string.h>
void main(void)
{
    char stra[]=" 1234567890";
    unsigned char len1;
    unsigned int len2;

    len1=strlen(stra);
    len2=strlenf("abcdefghijklmnopqrstuvwxyz");
```

```
        while(1)
        {
        }
}
```

结果：

len1 = 10

len2 = 26

*strncat，*strncatf

```
#include <string.h>
char *strncat(char *str1，char *str2，unsigned char n)
char *strncatf(char *str1，char flash *str2，unsigned char n)
```

strncat 和 strncatf 函数将字符串 str2 的前 n 个字符连接在字符串 str1 后。因此为字符串 str1 分配的存储器必须足够大，以便能容纳连接后的新的、更长的字符串加上空的结尾字符，否则将会出现难以预料的后果。在 strcatf 函数中，str2 所指的字符串必须存放在 FLASH 存储器中。函数的返回值是指向 str1 的指针。

返回值：*str1(指向字符串 str1 和 str2 连接后的字符串的指针，该字符串以空字符结尾)

```
#include <90s8515.h>
#include <stdio.h>
#include <string.h>

/*quartz crystal frepuency [Hz]*/
# define xtal 7372000L

/*baud rate*/
#define baud 9600

void main(void)
{
      char stra[]="abc";
      char strb[8];

      /* initialize the UART's baud rate */
      UBRR=xtal/16/baud-1;

      /*initialize the UART control register
      RX &TX enabled, no interrupts,8 data bits*/
      UCR=0x18;

      strcpyf(strc,"xyz");   //initialize strb!
```

```
    strncat(strb,stra,2);   //add stra to strb
    strncatf(strb,"123",1);   //add a flash string too!

    puts(strb);          //put it out to the UART!
    while(1)
    {
    }
}
```

结果：UART 以 9600 波特率的速率发送以下信息，

xyzab1

strncmp，strncmpf

```
#include <string.h>
signed char strncmp(char *str1，char *str2，unsigned char n)
signed char strncmpf(char *str1，char flash *str2，unsigned char n)
```

strncmp 和 strncmpf 函数至多比较字符串 str1 和字符串 str2 的前 n 个字符。对于 strncmpf 函数，str2 必须指向位于 FLASH 存储器中的字符串。函数从两个字符串的第 1 个字符开始比较，当 str1 的某个字符与 str2 的对应字符不匹配时，这两个字符之间的差别将决定函数的返回值。函数的返回值为 str1 与 str2 对应的不匹配字符的 ASCII 码值之差。不过，第 n 个字符以后的差异将不会影响函数的返回值。

返回值：

- 当 str1 < str2 时，返回负数
- 当 str1 = str2 时，返回零
- 当 str1 > str2 时，返回正数

```
#include <string.h>

void main(void)
{
    char stra[]= " george";
    char strb[]= " georgie";
    unsigned char result;
    unsigned char resultf;

    result=strcmp(stra,strb,5);
    resultf=strlencmpf(stra,"george",6);

    while(1)
    {
    }
}
```

结果：

result = 0

resultf = 0

*strncpy，*strncpyf

```
#include <string.h>
char *strncpy(char *dest，char *src，unsigned n)
char *strncpyf(char *dest，char flash *src，unsigned n)
```

strncpy 和 strncpyf 函数将 src 所指的字符串中的前 n 个字符复制到 dest 指向的地址中。如果字符串 src 小于 n 个字符，则在其末尾补空字符作为结尾符，以保证复制到 dest 所指字符串中的字符个数为 n。如果字符串 src 的长度大于等于 n，则不会有结尾字符添加在字符串 dest 末尾。

返回值：指向 dest 的指针

```
#include <90s8515.h>
#include <stdio.h>
#include <string.h>

/*quartz crystal frepuency [Hz]*/
# define xtal 7372000L

/*baud rate*/
#define baud 9600

char stra[]="Hello";
char strb[]="HELLO";
char strc[6];

void main(void)
{
    char stra[]="abc";
    char strb[8];

    /*initialize the UART's baud rate */
    UBRR=xtal/16/baud-1;

    /*initialize the UART control register
    RX &TX enabled, no interrupts,8 data bits*/
    UCR=0x18;

    strncpy (strb,stra,3);          //copy stra to strb
    strncpyf (strc,"world",10);   //copy "world" to strc
    printf("%s %s\n\r",strb,strc);
```

```
    while(1)
    {
    }
}
```

结果：因为“3”小于字符串 stra 的总长度，所以只有前 3 个字符被复制到字符串 strb 中，但并没有复制作为结尾符的空字符。因此，字符串 strb 用原有的结尾符来作为新字符串的结尾符。这样，UART 将以 9600 波特率的速率发送以下信息，

```
Hello World
```

*strpbrk，*strpbrkf

```
#include <string.h>
char *strpbrk(char *str，char *set)
char *strpbrkf(char *str，char flash *set)
```

strpbrk 和 strpbrkf 函数在字符串 str 中查找是否有在字符串 set 中出现的字符。如果有，函数将返回指向 str 中第 1 个匹配字符的指针。如果没有匹配的字符，则返回一个空指针。对于 srpbrkf 函数，字符串 set 必须位于 FLASH 存储器中。

返回值：指向字符串 str 中第 1 个与字符串 set 中字符相匹配的字符的指针

```
#include <90s8515.h>
#include <stdio.h>
#include <string.h>

/*quartz crystal frepuency [Hz]*/
# define xtal 7372000L

/*baud rate*/
#define baud 9600

void main(void)
{
    char stra[]="11/25/00";
    char set[]="/.,!-";
    char strb[]="November 25, 2000";
    char *pos;
    char *fpos;

    /*initialize the UART's baud rate */
    UBRR=xtal/16/baud-1;

    /*initialize the UART control register
    RX &TX enabled, no interrupts,8 data bits*/
```

```
    UCR=0x18;

    pos=strpbrk(stra,set);   //find first occurrence of something

    fpos=strpbrkf(strb,",.-");
    printf("Initial Date:%s\n\r",stra);
    printf("string following match:%s\n\r",pos);
    printf("Just the year:%s\n\r",fpos+1);

    while(1)
    {
    }
}
```

结果：UART 以 9600 波特率的速率发送以下信息，

Initial Date：11/25/00
String following match：/25/00
Just the year：2000

strpos

```
#include <string.h>
char strpos(char *str，char c)
```

strpos 函数将查找字符 c 在字符串 str 中第一次出现的位置。函数的返回值便是 c 第一次出现位置的索引。如果字符 c 没有在字符串 str 中出现，则函数的返回值为-1。

返回值：如果字符 c 没有在 str 中出现，返回-1；否则返回 c 第一次出现的位置的索引

```
#include <90s8515.h>
#include <stdio.h>
#include <string.h>

/*quartz crystal frepuency [Hz]*/
# define xtal 7372000L

/*baud rate*/
#define baud 9600

void main(void)
{
    char stra[]="11/25/2000";
    char month_day;
    char day_year;

    /*initialize the UART's baud rate */
    UBRR=xtal/16/baud-1;
```

```
    /*initialize the UART control register
    RX &TX enabled, no interrupts,8 data bits*/
    UCR=0x18;

    month_day=strpos(stra,'/');   //find first slash character
    day_year=strrpos(stra,'/');   //find last slash character

    printf("Starting string:%s\n\s",stra);

    //replace slash character with dashes
    stra[month_day]='-';
    stra[day_year]='-';

    printf("Modified string:%s\n\r",stra);

    while(1)
    {
    }
}
```

结果：UART 以 9600 波特率的速率发送以下信息

Starting String：11/25/2000
Modified String：11-25-2000

*strrchr

```
#include <string.h>
char *strrchr(char *str，char c)
```

strtchr 函数查找字符 c 在字符串 str 中最后一次出现的位置。如果 c 没有在字符串 str 中出现，则返回一个空指针。

返回值：字符 c 最后一次在字符串 str 中出现的位置的指针。如果 c 没有在字符串 str 中出现，将返回一个空指针

```
#include <90s8515.h>
#include <stdio.h>
#include <string.h>

/*quartz crystal frepuency [Hz]*/
# define xtal 7372000L

/*baud rate*/
#define baud 9600
```

```
void main(void)
{
    char stra[]="123.45.789";
    char *strb;     //no need to allocate space ,pointing to stra-already allocated!

    /*initialize the UART's baud rate */
    UBRR=xtal/16/baud-1;

    /*initialize the UART control register
    RX &TX enabled, no interrupts,8 data bits*/
    UCR=0x18;

    strb=strrchr(stra,'.');
    printf("Full string:%s\n\rNew String :%s\n\r",stra,strb);
    *strb='6' ;              //replace the decimal point with a 6
    printf("Modified string:%s\n\r",stra);

    while(1)
    {
    }
}
```

返回值：UART 以 9600 波特率的速率发送以下信息，

```
Full String：123.45.789
New String：.789
Modified String：123.456789
```

*strrpbrk，*strrpbrkf

```
#include <string.h>
char *strrpbrk(char *str，char *set)
char *strrpbrkf(char *str，char flash *set)
```

strrpbrk 和 strrpbrkf 函数在字符串 str 中查找是否有在字符串 set 中出现的字符。如果有，函数将返回指向 str 中最后一个匹配字符的指针。如果没有匹配的字符，则返回一个空指针。对于 srpbrkf 函数，字符串 set 必须存放在 FLASH 存储器中。

返回值：指向字符串 str 中最后一个与字符串 set 中字符相匹配的字符的指针

```
#include <90s8515.h>
#include <stdio.h>
#include <string.h>

/*quartz crystal frepuency [Hz]*/
# define xtal 7372000L
```

```
/*baud rate*/
#define baud 9600

void main(void)
{
    char stra[]="11/25/00";
    char set[]="/.,!-";
    char strb[]="November 25 2000";
    char *pos;
    char *fpos;

    /*initialize the UART's baud rate */
    UBRR=xtal/16/baud-1;

    /*initialize the UART control register
    RX &TX enabled, no interrupts,8 data bits*/
    UCR=0x18;

    pos=strrpbrk(stra,set);   //find last occurrence of something!
    fpos= strrpbrkf(strb,",.-@$ ");
    printf("Year in 11/25/00 :%s\n\r",pos);
    printf("String following match:%s\n\r",fpos);

    while(1)
    {
    }
}
```

结果：UART 以 9600 波特率的速率发送以下信息，

```
Year in 11/25/00：/00
String following match：，2000
```

strrpos

```
#include <string.h>
char strpos(char *str，char c)
```

strpos 函数将查找字符 c 在字符串 str 中最后一次出现的位置。函数返回 c 最后一次出现位置的索引。如果字符 c 没有在字符串 str 中出现，则函数的返回值为-1。

返回值：如果 c 没有在 str 中出现，返回-1；否则返回 c 最后一次出现的位置的索引

```
#include <90s8515.h>
#include <stdio.h>
#include <string.h>

/*quartz crystal frepuency [Hz]*/
```

```
# define xtal 7372000L

/*baud rate*/
#define baud 9600

void main(void)
{
      char stra[]="11/25/2000";
      char month_day;
      char day_year;

      /*initialize the UART's baud rate */
      UBRR=xtal/16/baud-1;

      /*initialze the UART control register
      RX &TX enabled, no interrupts,8 data bits*/
      UCR=0x18;

      month_day=strpos(stra,'/');   //find first slash character
      day_year=strrpos(stra,'/');   //find last slash character

      printf("Starting string:%s\n\s",stra);

      //replace slash character with dashes
      stra[month_day]='-';
      stra[day_year]='-';

      printf("Modified string:%s\n\r",stra);

      while(1)
      {
      }
}
```

结果：UART 以 9600 波特率的速率发送以下信息：

Starting String：11/25/2000
Modified String：11-25-2000

strspn，strspnf

```
#include <string.h>
unsigned char strspn(char *str，char *set)
unsigned char strspnf(char *str，char flash *set)
```

strspn 和 strspnf 函数返回字符串 str 中第 1 个不在字符串 set 中出现的字符的索引。如果字

符串 str 中的所有字符都在字符串 set 中，则函数将返回 str 的长度。如果字符串 str 中没有字符在字符串 set 中，则函数将返回 0。对于 strspnf 函数，字符串 set 必须存放在 FLASH 存储器中。

返回值：字符串 str 中第 1 个不在字符串 set 中的字符的索引

```
#include <string.h>

void main(void)
{
    char set[]="1234567890－()";
    char stra[]="1.8000.555.1212";
    char index_1;
    char index_2;

    index_1=strspn(stra,set);
    index_2=strspnf(stra,".1234567890－()");

    while(1)
    {
    }
}
```

结果：

index_1 = 1

index_2 = 14

***strstr，*strstrf**

```
#include <string.h>
char *strstr(char *str1，char *str2)
char *strstrf*(char *str1，char flash *str2)
```

strstr 和 strstrf 函数查找在字符串 str1 中是否包含字符串 str2。如果字符串 str2 被包含在字符串 str1 中，函数将返回指向 str1 中与 str2 相匹配的子字符串中第 1 个字符的指针。如果 str2 不在 str1 中，则返回一个空指针。对于 strstrf 函数，字符串 str2 必须存放在 FLASH 存储器中。

返回值：如果 str2 不在 str1 中，则返回一个空指针；否则返回指向 str1 中与 str2 相匹配的子字符串中第 1 个字符的指针

```
#include <90s8515.h>
#include <stdio.h>
#include <string.h>

/*quartz crystal frepuency [Hz]*/
# define xtal 7372000L

/*baud rate*/
```

```
#define baud 9600

void main(void)
{
    char stra[]="Red,Green,Blue";
    char strb[]="Green";
    char *ptr;
    char *ptrf;

    /*initialize the UART's baud rate */
    UBRR=xtal/16/baud-1;

    /*initialize the UART control register
    RX &TX enabled, no interrupts,8 data bits*/
    UCR=0x18;

    //grab a pointer to where Green is
    ptr=strstr(stra,strb);

    //grab a pointer to where the first B is
    ptrf=strstrf(stra,"B");

    printf("Starting String:%s\n\r",stra);
    printf("Search String:%s\n\r",strb);
    printf("Results String 1:%s\n\r",ptr);
    printf("Results String 2:%s\n\r",ptrf);

    while(1)
    {
    }
}
```

结果：UART 以 9600 波特率的速率发送以下信息，

```
Starting String：Red Green Blue
Search String：Green
Results String 1：Green Blue
Results String 2：Blue
```

*strtok

```
#include <string.h>
char *srtok(char *str1，char flash *str2)
```

strtok 函数扫描字符串 str1 中第 1 个没有包含在 str2 中的字段(token)，其中字符串 str2 是被存放在 FLASH 存储器中的。函数期望字符串 str1 由一系列的文本字段组成，被字符串 str2 中

的一个或多个字符(标记分隔符，token separators)分隔开。通过重复地调用这个函数，就可以从 str1 中取出各个由 str2 中字符分隔的字段来。

以一个指向 str1 的非空指针作为参数第 1 次调用 strtok 函数，将返回指向字符串 str1 第 1 个字段的第 1 个字符的指针。函数会自动在找到的字段的末尾，在原来的记号分隔符(str2 中的字符)中第 1 个字符的位置放置一个空字符作为结尾符。接着，以空字符作为参数 str1 的值来调用 strtok 函数，返回字符串 str1 中按顺序的下一个字段，直到 str1 中再无字段。当字符串中再也找不出字段时，函数将返回一个空指针。

应注意的是调用 strtok 函数会在字符串 str1 的每个字段之后加上空字符作为结尾符，str1 的结构会有所改变，因此，要在调用 strtok 函数之前做好备份。

返回值：当字符串 str1 中不再有字段时，返回空指针；否则，返回指向下一个字段的指针

```
#include <90s8515.h>
#include <stdio.h>
#include <string.h>

/*quartz crystal frepuency [Hz]*/
# define xtal 7372000L

/*baud rate*/
#define baud 9600

char mytext[]="(888)777-2222";
char flash separators[]="()-";

void main(void)
{
      char area_code[4];
      char *prefix;
      char *postfix;
      char backup_copy[14];

      /*initialize the UART's baud rate */
      UBRR=xtal/16/baud-1;

      /*intialize the UART control register
      RX &TX enabled, no interrupts,8 data bits*/
      UCR=0x18;

      //we want to keep the original around too!
      strcpy(backup_copy,mytext) ;

      //grab a pointer to the area code
      strcpy(area_code,strtok(mytext,separators)) ;
```

```
        //grab a pointer to the prefix
        prefix=strtok(0,separators);

        //grab a pointer to the postfix
        postfix=strtok(0,separators);

        printf("Starting String:%s\n\r",backup_copy);
        printf("Area_code:%s\n\r",area_code);
        printf("Prefix:%s\n\r",prefix);
        printf("Postfix:%s\n\r",postfix);

        while(1)
        {
        }
    }
```

结果：UART 以 9600 波特率的速率发送以下信息

Starting String：(888)777-2222
Area Code：808
Prefix：777
Postfix：2222

tan

```
#include <math.h>
float tan(float x)
```

tan 函数计算浮点型数 x 的正切函数值，角度 x 应该用弧度表示。
返回值：tan(x)

```
#include <math.h>
void main()
{
    float new_val;

    new_val = tan(5.121);
    while(1)
    {
    }
}
```

结果：new_val = –2.309

tanh

```
#include <math.h>
float tanh(float x)
```

tanh 函数计算浮点型数 x 的双曲正切函数值，角度 x 应该用弧度表示。

返回值：tanh(x)

```
#include <math.h>
void main()
{
    float new_val;
    new_val = tanh(5.121);
    while(1)
    {
    }
}
```

结果：new_val = 0.999

toascii

```
#include <ctype.h>
unsigned char toascii(char c)
```

toascii 函数将字符 c 转换为 7 位 ASCII 码格式。这种转换通过以下定义实现：

```
#define toascii(c) (c)& 0x7f
```

返回值：c 的 ASCII 码值(0～127)

```
#include <ctpye.h>
void main()
{
    char ascii_value;
    ascii_value = toascii(0xB1);
    while(1)
    {
    }
}
```

结果：ascii_value = 0x31 = '1'

toint

```
#include <ctype.h>
unsigned char toint(char c)
```

toint 函数将 ASCII 码字符 c 看作一个十六进制数，返回 0～15 的无符号字符。如果字符 c 不能被看作一个合法的十六进制数，函数将返回 0。

返回值：ASCII 码字符 c 代表的 0～15 之间的无符号字符

```
#include <ctype.h>
void main()
{
    unsigned char hex_value;
```

```
        hex_value = toint('a');
        while(1)
        {
        }
}
```

结果：hex_value = 0x0a

tolower

```
#include <ctype.h>
char tolower(char c)
```

tolower 函数将 ASCII 码字符 c 从大写形式转换成为小写形式。如果 c 原来就是小写形式，返回值将不会有改变。

返回值：以小写形式出现的 ASCII 码字符 c

```
#include <ctype.h>
void main()
{
    char lower_case_c;
    lower_case_c = tolower('U');
    while(1)
    {
    }
}
```

结果：lower_case_c = 'u'

toupper

```
#include <ctype.h>
char toupper(char c)
```

toupper 函数将 ASCII 码字符 c 从小写形式转换为大写形式。如果 c 原来就是大写形式，返回值将不会有改变。

返回值：以大写形式出现的 ASCII 码字符 c

```
#include <ctype.h>
void main()
{
    char upper_case_c;
    upper_case_c = toupper('u');
    while(1)
    {
    }
}
```

结果：upper_case_c ='U'

附录B　CodeVisionAVR和STK500入门

概要

- 用 Atmel STK500 启动工具包(Start Kit)和 AVR Studio 调试器安装并配置好 CodeVisionAVR
- 用 CodeWizardAVR 自动程序生成器创建一个新的项目
- 编辑并编译 C 程序
- 利用 STK500 启动工具包将程序加载到目标微控制器并执行

引言

本附录的目的是指导读者如何用 CodeVisionAVR C 编译器准备编写 C 语言程序。作为本附录中心内容的示例程序，是利用 STK500 启动工具包为 Atmel AT90S8515 微控制器编写的一个简单程序。

准备工作

运行 setup.exe 文件，安装 CodeVisonAVR C 编译器。
假定上述程序被安装到默认的路径：C:\cvavr 下。
运行 setup.exe 文件，安装 Atmel AVR Studio 调试器。
假定 AVR Sttudio 程序被安装到默认的路径：C:\Program Files\Atmel\AVR Studio 下。
下面将要介绍的示例程序需要有 Atmel AT90S8515 微控制器和 STK500 启动工具包。
按 STK500 用户手册中的提示安装好该启动工具包。
确定断电后，将 AT90S8515 芯片插入标有 SCKT3000D3 标号的插槽中。
将 XTAL1 跳线接好，并将 OSCSEL 跳线接在 1 号和 2 号引脚之间。
用一根 10 引脚的带状电缆将 PORTB 和 LEDS 两个接头连接好。
这样可以显示 AT90S8515 芯片 PORTB 输出端的状态。
用一根 6 引脚的带状电缆将 ISP6PIN 和 SPROG3 两个接头连接好。
这样，当程序编译成功后，CodeVisionAVR 可以自动对 AVR 芯片进行编程。
为此，还需要在 CodeVisionAVR 软件中进行如下的设置：
打开 CodeVisionAVR IDE，选择 Setting | Programmer 命令，出现如图 B-1 所示对话框。

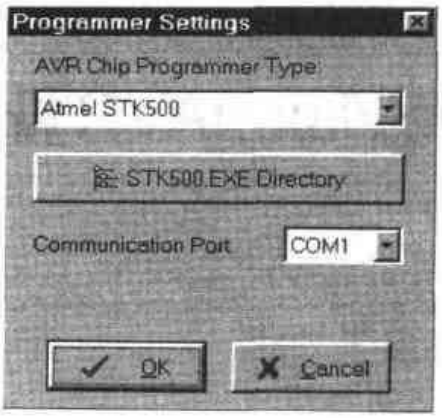


图 B-1　Programmer Settings 对话框

在 AVR Chip Programmer Type 下拉列表中选择 Atmel STK500 选项，并选择好 STK500 启动工具包使用的 Communication Port 。

然后单击 STK500.EXE Directory 按钮，确定 AVR Studio 提供的 stk500.exe 命令行工具的位置，出现如图 B-2 所示对话框。

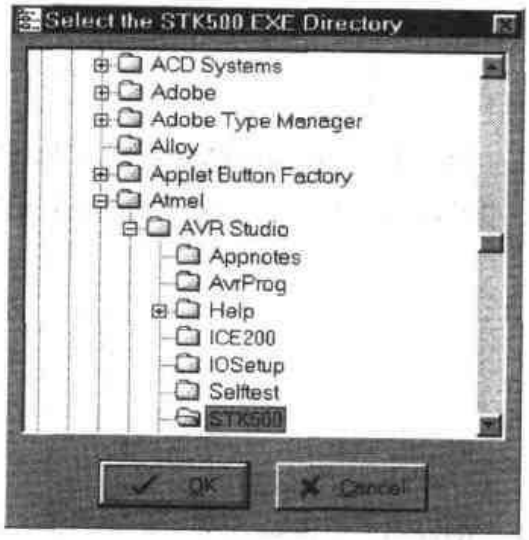


图 B-2　选择 STK500.EXE 目录

选择 C:\Program Files\Atmel\AVR Studio\STK500 目录，并单击 OK 按钮。

然后再次单击 OK 按钮来保存 Programmer Settings 设置。

为了激活 AVR Studio 提供的调试器/仿真器，最后还需要在 CodeVisionAVR IDE 中进行一项设置，即选择 Settings | Debuger 命令，会出现如图 B-3 所示对话框。

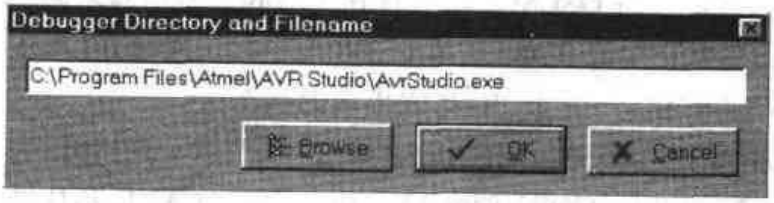


图 B-3　Debugger Directory and Filename 对话框

设置路径，输入“C:\Program Files\Atmel\AVR Studio\AvrStudio.exe”，然后单击 OK 按钮

即可。

创建一个新项目

选择 File | New 命令，或单击工具栏上的加按钮图标，按钮来新建一个项目，会出现如图 B-4 所示对话框。

选择 Project 单选按钮，然后单击 OK 按钮，出现如图 B-5 所示对话框。

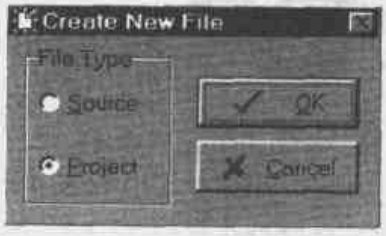


图 B-4　Create New File 对话框

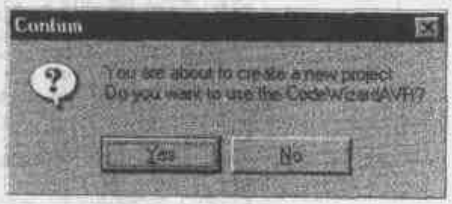


图 B-5　Confirm 提示对话框

单击 Yes 按钮，即可使用 CodeWizardAVR 自动程序生成器。

使用 CODEWIZARDAVR 自动程序生成器

CodeWizardAVR 可以针对不同的 AVR 微控制器简化编写程序起始代码的任务。

对本例来说，在图 B-6 所示对话框中，应该选择 AT90S8515 微控制器，并且根据 STK500 启动工具包的时钟，选择 3.68MHz 的时钟频率。

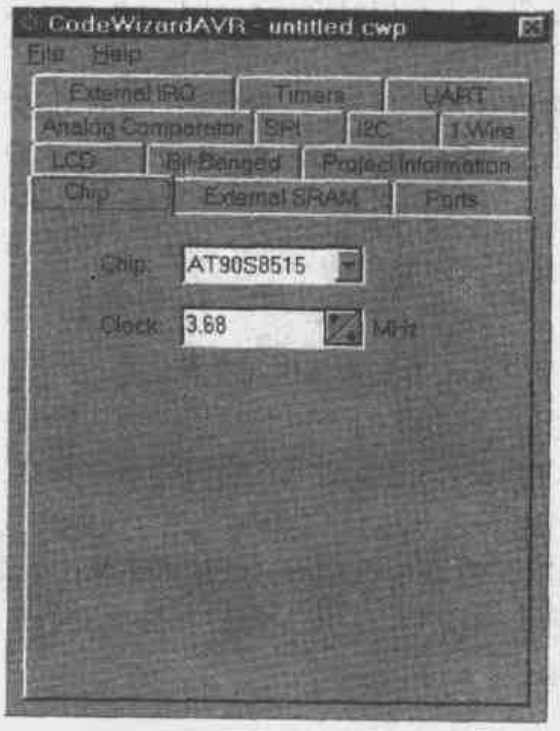


图 B-6　选择微调控制器与时钟频率

配置输入/输出端口

切换到 Ports 选项卡，设置各 I/O 端口在目标系统中的初始值，如图 B-7 所示。

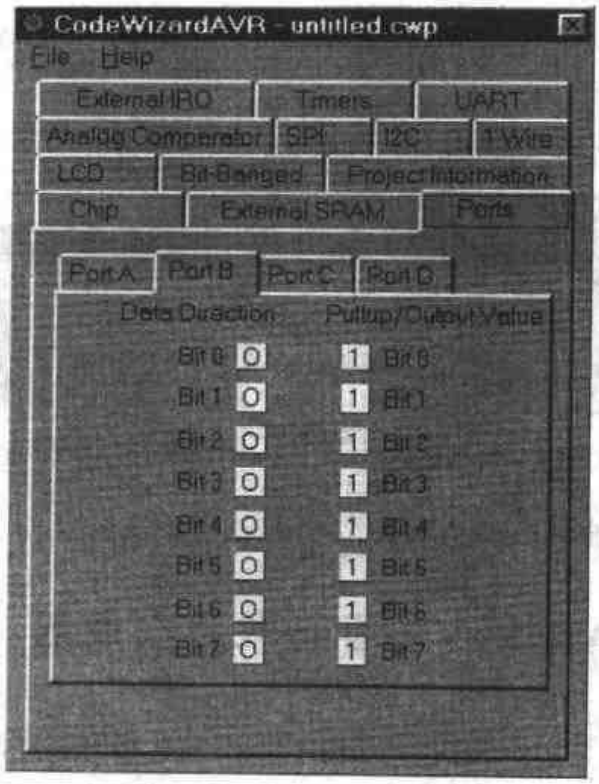


图 B-7　初始化 I/O 端口

在三态模式下(Tri-state)，默认设置所有的端口对于目标系统都是用作输入的(Data Direction 位均被设置为 I)。

在这个例子中，想要用端口 B 输出信息，只需打开 Port B 选项卡，设置所有的 Data Direction 位为 0(单击它们即可)。要使 STK500 中的 LED 断开，还要设置 Output Values 值全部为 1。

配置 Timer 1

在这个项目中，还要设置 Timer 1 来产生溢出中断。

为此，在图 B-8 所示对话框中选择 Timers 选项卡，并选择其中的 Timer 1 选项卡。

按图 B-8 所示来设置 Timer 1 的各选项。将时钟频率值选定为 3.594kHz(系统时钟频率 3.68MHz 除以 1024)。

这样，计时器被设置在默认的 Output Compare 模式下来产生溢出中断。

为了获得 LED 在每 2 秒之间的变化频率，需要在每次溢出中断后，重新设置 Timer 1 的值为 0x10000–(3594/2)=0xF8FB。

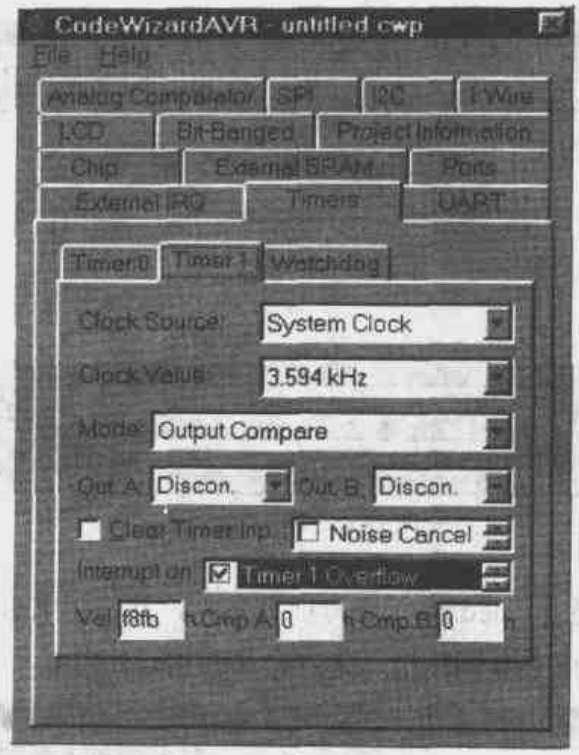


图 B-8　选择 Timer 1

编译项目

选择 File | Generate，Save and Exit 命令，Code Wizard 将生成 C 程序框架，同时也将正确设置端口 B 和 Timer 1 的溢出中断，接着会出现如图 B-9 所示对话框。

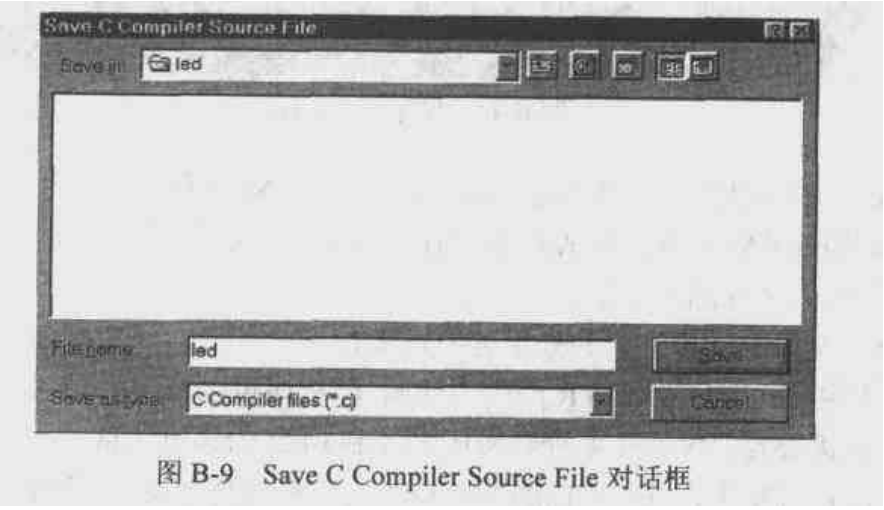


图 B-9　Save C Compiler Source File 对话框

单击按钮，建立一个新的目录：C:\cvavr\led。这个示例项目的所有文件都被保存在这个目录下。

然后在 File name 文本框中输 C 源文件的文件名“led.c”，并单击 save 按钮，出现如图 B-10 所示的对话框。

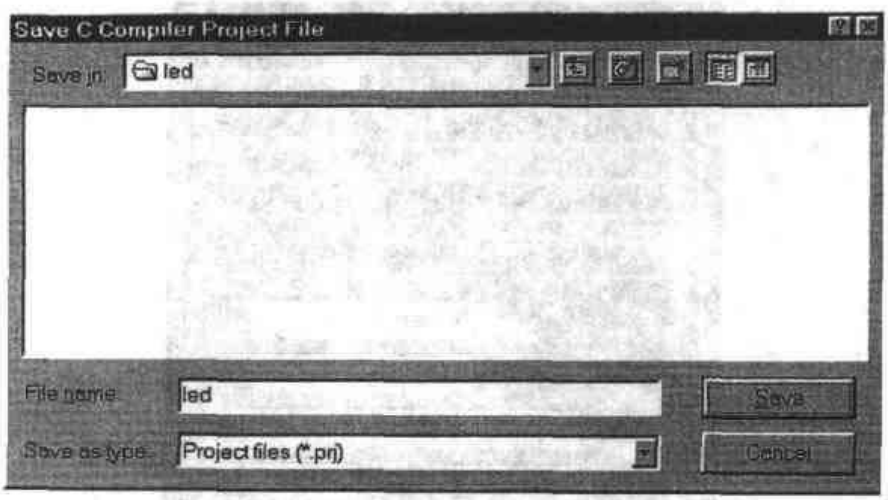


图 B-10　指定项目名

在 File name 文本框中输入“led.prj”作为项目名称，并将该文件保存到 C:\cvavr\led 目录下。最后得到提示要保存 CodeWizard 项目文件，如图 B-11 所示。

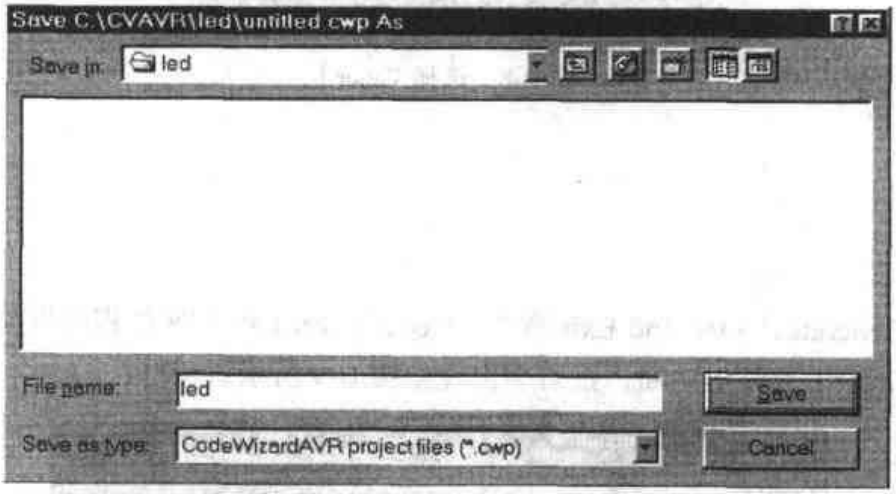


图 B-11　保存项目文件

在 File name 文本框中指定项目名为：led.cwp 并单击 Save 按钮。

由于将 CodeWizardAVR 外设配置信息保存在 led.cwp 项目文件中，下次需要建立新的项目时，便可重用其中的一些初始化代码。

现在，led.c 源文件自动打开，并处于可编辑状态。

用户可以编辑由 CodeWizardAVR 产生的代码。源程序可以参考附录 A 中的 C 库函数清单。

在本例中，只需要自己编写中断处理程序来控制 LED 的输出信息。

在 Code WizardAVR 中，用户自己添加的小块代码将被高亮显示，其余的则是系统提供的。

查看或修改项目配置

在任何时候，选择 Project | Configure 命令或单击工具栏上的按钮，都可以修改项目的配置。此时出现如图 B-12 所示的对话框。

要向项目中添加或删除文件，切换到 Files 选项卡，再单击 Add 按钮或 Remove 按钮即可。

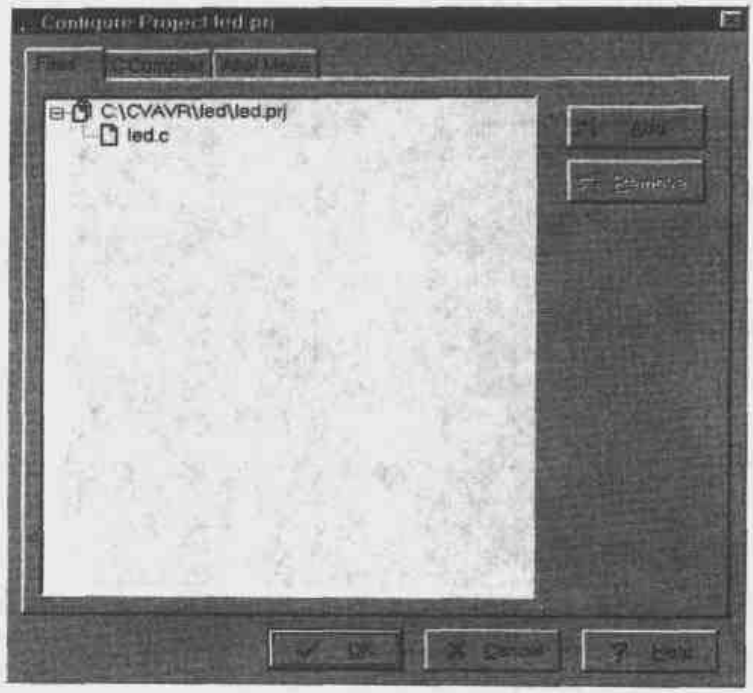


图 B-12　Configure Project 对话框

要改变目标微控制器类型、时钟频率或其他各种编译器的选项，切换到 C Compiler 选项卡。在图 B-13 所示的对话框中，可以改变编译器的各种设置。

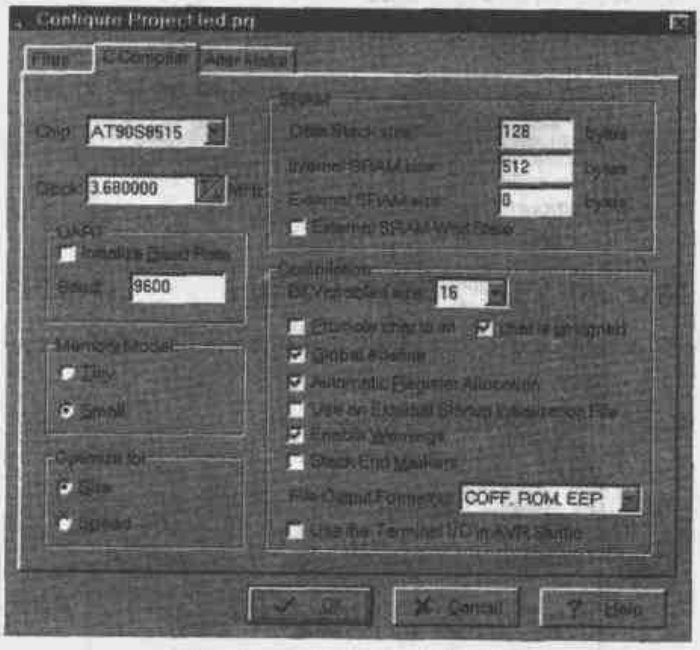


图 B-13　改变编译器的设置

还可以决定在构建程序后，是否需要自动对目标微处理器进行自动编程。这需要在 After Make 选项卡中指定，如图 B-14 所示。

按示例程序的要求，必须选中 Program the Chip 复选框。这将允许在生成(Make)完成之后，自动对 AVR 芯片进行编程。

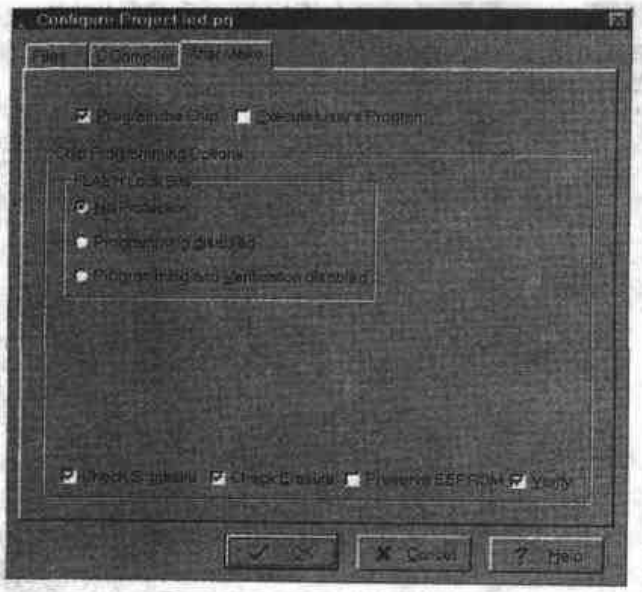


图 B-14　After Make 选项卡

构建项目

选择 Project | Make 命令或单击工具栏上的按钮，将构建项目程序。成功地编译和汇编后，将会弹出如图 B-15 所示的对话框。

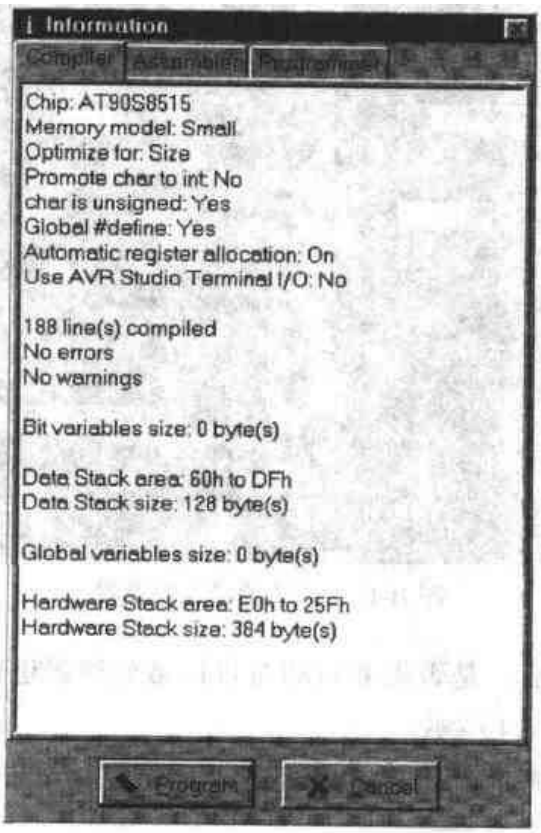


图 B-15　Information 对话框

该对话框显示了编译器是如何使用 RAM 存储器空间的。

选择 Assembler 选项卡，Assembler 选项卡将显示汇编代码的容量。

选择 Programmer 选项卡，Programmer 选项卡将会显示 Chip Programming Counter 的值，如图 B-16 所示。

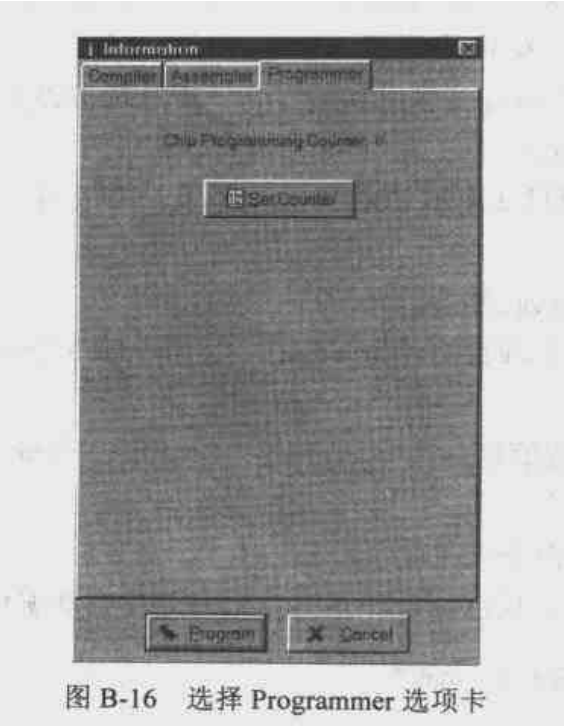


图 B-16　选择 Programmer 选项卡

单击 Set Counter 按钮可以重新初始化计数器的值。

如果整个构建过程成功，运行 STK500 启动工具包并单击 Program 按钮，将开始对芯片进行自动编程。

当编程过程完成后，程序将通过 STK500 启动工具包在目标微控制器上开始运行。

小结

准备工作：

(1) 安装 CodeVisionAVR C 编译器。

(2) 安装 Atmel AVR Studio 调试器。

(3) 安装 STK500 启动工具包。

(4) 通过选择 Setting | Programmer 命令，在 CodeVisionAVR IDE 中设置对 SEK500 程序的支持。

AVR 可编程芯片类型：STK500
指定 STK500.EXE 的安装目录：C:\Program Files\Atmel\AVR Studio\STK500

(5) 选择 Settings | Debugger 命令，在 CodeVisionAVR IDE 中设置好对 AVR Studio 的支持。输入：C:\Program Files\Atmel\AVR Studio

编程步骤：

(1) 选择 File | New | Select Project 命令，建立一个新项目。

(2) 指定使用 CodeWizardAVR 来生成 C 语言源文件和项目文件：对话框出现提示 Use the CodeWizard? 时，单击 Yes 按钮。

(3) 在 CodeWizardAVR 对话框中选择芯片类型为 AT90S8515，时钟频率为 3.86Hz。

(4) 配置输入/输出端口，在 Ports 选项卡中进行相应设置。

(5) 配置 Timer 1：在 Timers 选项卡中的 Timer 1 选项卡设置 Clock Value 为 3.594kHz，Interrupt on 为 Timer 1 Overflow，Val 为 0xF8FB。

(6) 生成 C 源文件，C 项目文件和 CodeWizardAVR 项目文件，选择 File | Generate，Save and Exit 命令。

建立一个新的目录：C:\cvavr\led

保存源文件为 led.c，保存项目文件为 led.prj，保存 Code Wizard AVR 项目文件为 led.cwp

(7) 编写 C 语言源程序。

(8) 查看并更改项目配置信息，选择 Project | Configure 命令，在 After Make 选项卡选中 Program the Chip 复选框。

(9) 选择 Project | Make 命令编译程序。

(10) 使用 STK500 启动工具包对 AT90S8515 芯片进行自动编程：

Apply power | Information | Program

源代码清单

```
/*****************************************************************************
This program was produced by the
CodeWizardAVR V1.0.1.8C Standard
Automatic Program Generator
© Copyright 1998-2001
Pavel Haiduc, HP InfoTech S.R.L.
http://infotech.ir.ro
e-mail: hpinfotech@xnet.ro, hpinfotech@xmail.ro

Project   :
Version   :
Date      :
Author    :
Company   :
Comments  :


Chip type           : AT90S8515
Clock frequency     : 3.680000 MHz
Memory model        : Small
Internal SRAM size  : 512
```

```
External SRAM size    : 0
Data Stack size       : 128
*****************************************/

#include <90s8515.h>

// the LED 0 on PORTB will be on
unsigned char led_status=0xFE;

// Timer 1 overflow interrupt service routine
interrupt [TIM1_OVF] void timer1_ovf_isr(void)
{
// Reinitialize Timer's 1 value
TCNT1H=0XF8;
TCNT1L=0XFB;
//place your code here
//move the LED
led_status <<=1;
led_status |=1;
if (led_status==0xFF) led_status=0xFE;
// turn on the LED
PORTB=led_status;
}

void main(void)
{
// Input/Output Ports initialization
//Port A
PORTA=0X00;
DDRA=0X00;

//Port B
PORTB=0XFF;
DDRB=0XFF;

//Port C
PORTC=0X00;
DDRD=0X00;

// Timer/Counter 0 initialization
// Clock source: System Clock
// Clock value: Timer 0 Stopped
// Mode: Output Compare
// OC0 output: Disconnected
TCCR0=0X00;
```

```
TCNT0=0X00;

// Timer/Counter 1 initialization
// Clock source: System Clock
// Clock value: 3.594 kHz
// Mode: Output Compare
// OC1A output: Discon.
// OC1B output: Discon.
// Noise Canceler: Off
//Input Capture on Falling Edge
TCCR1A=0x00;
TCCR1B=0x00;
TCNT1H=0xF8;
TCNT1L=0xFB;
OCR1AH=0x00;
OCR1AL=0x00;
OCR1BH=0x00;
OCR1BL =0x00;

// External Interrupt (s)    initialization
// INT0: Off
// INT1: Off
GIMSK=0x00;
MCUCR=0x00;

// Timer(s)/Counter(s) Interrupt (s) initialization
TIMSK =0x80;

// Analog Comparator initialization
// Analog Comparator: Off
// Analog Comparator Input Capture by Timer/Counter 1: Off
ACSR=0x80;

// Global enable interrupts
#asm("sei")

// the rest is done by TIMER1 overflow interrupts
while (1);
}
```

附录C　AVR微控制器编程

本附录的目的是介绍对 AVR 微控制器的各种可行的编程方法，并提供足够信息来解决编程中的问题。不过，本附录提供的信息还不足以让读者能自己来设计编程器件(Programming Device)，如果想设计自己的编程器请参阅微控制器规格说明。

对所有的微控制器和各种类型的微机来说，启动系统后，都需要让全部或部分的操作系统常驻在计算器中。微控制器和计算机不能“什么也不做”——它们必须一直执行。如果它们在执行空循环，可能看起来什么事也没做，但其实仍然在执行，空闲循环语句。

这意味着操作代码必须长期存放在不易丢失的内存区域中。对于 Atmel AVR 微控制器来说，这种存储区域由 FLASH 存储器构成。假如在您的系统中没有使用在线(in-circuit)仿真器或模拟器，那么所有准备运行的指令代码必须都是放在 FLASH 存储器中的。

Atmel 提供了两种机载(on-board)编程机制：SPI 端口和并行编程模式(scheme)，但后者并不常用。事实上，SPI 方法几乎被专门用来对设备编程。程序字节按一个特定的顺序应用在 SPI 端口，随后被存放在 FLASH 存储器中。在微控制器复位时，这些代码就会被再次执行。SPI 编程的一大优点在于所有操作都几乎可在线完成；设备不需要从它们的应用程序脱离，也不需要特定的编程设备。在商业应用中，采用这种方式可以很方便也升级。

SPI 端口编程

下面的例子中，对 AT90S8535 芯片进行编程。应注意的是，所有编程命令都要求按 4 个字节传输到待编程的微控制器。总是按 4 个字节传输是为了进行同步，即使其中可能有一些是哑元字(dummy word)。

(1) 微控制器通电并运行后，按下 RESET 键并使 SCLK 维持至少 20ms 的低电压。

(2) 通过 SPI 端口传送 Programming Enable 命令字给微控制器。在传送第 3 个字节的过程中，将值 0x53 回显给编程设置。最后，在传送第 4 个字节时送一个哑元表示命令结束。

(3) 传送 Write Program Memory 命令，该命令中包含了要写入字节的地址和字节本身的地址(参见表 C-1)。

(4) 使用 Read Program Memory 命令不断读取刚刚送出的字节。读回字节直到读出正确的内容。在该字节被写入 FLASH 存储器之前，将一直返回 0xff。

(5) 重复步骤(3)至(4)，直到所有的程序代码都被加载入 FLASH 存储器中。

(6) 向 RESET 端发送一个高电平，使微控制器恢复正常操作，并且开始执行刚刚加载入 FLASH 存储器的代码。

其他操作也可以通过类似方式在 AT90S8535 微控制器上执行，只需在表 C-1 中选择相应的命令字即可。

实际的 SPI 编程命令根据 Atmel AVR 微控制器的不同而有所区别，想了解具体信息请查阅相应设备的说明。

表 C-1　AT90S8535 SPI 编程命令字

命　令	命令字节					函　数
		字节 1	字节 2	字节 3	字节 4	
Programming Enable	Sand	10101100	01010011	xxxxxxxx	xxxxxxxx	Enable Programming Mode
	Rcv.	xxxxxxxx	xxxxxxxx	01010011	xxxxxxxx	
Chip Erase	Send	10101100	100xxxxx	xxxxxxxx	xxxxxxxx	Erase Flash and EEPROM
	Rcv.	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	
Read Program Memory High Byte	Sand	00101000	xxxxxxxx	aaaaaaaa	xxxxxxxx	Read High Byte from Program Memory word at aaaaaaaaaaaa
	Rcv.	xxxxxxxx	xxxxxxxx	xxxxxxxx	bbbbbbbb	
Read Program Memory Low Byte	Sand	00100000	xxxxaaaa	aaaaaaaa	xxxxxxxx	Read Low Byte from Program Memory word at aaaaaaaaaaaa
	Rcv.	xxxxxxxx	xxxxxxxx	xxxxxxxx	bbbbbbbb	
Write Program Memory-High Byte	Sand	00100000	xxxxaaaa	aaaaaaaa	bbbbbbbb	Write High Byte to Program Memory word at aaaaaaaaaaaa
	Rcv.	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	
Write Program Memory-Low Byte	Sand	01000000	xxxxaaaa	aaaaaaaa	bbbbbbbb	Write Low Byte to Program Memory word at aaaaaaaaaaaa
	Rcv.	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	
Read EEPROM Memory Byte	Sand	10100000	xxxxxxxa	aaaaaaaa	xxxxxxxx	Read EEPROM Memory Byte at aaaaaaaaa
	Rcv.	xxxxxxxx	xxxxxxxx	xxxxxxxx	bbbbbbbb	
Write EEPROM Memory Byte	Sand	11000000	xxxxxxxa	aaaaaaaa	bbbbbbbb	Write EEPROM Memory Byte at aaaaaaaaa
	Rcv.	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	

注意：1. a = address bit
b = data bit
x = don't care bit
2. 芯片也支持论询和向锁定位中写内容的命令，详细信息请参见规格说明部分。
3. 在使用 Chip Erase 命令成功擦除后，需要一个 RESET 及新的 Programming Enable 命令。

商业编程器

目前市场上已经有许多可以对保存在微控制器 FLASH 存储器中的代码进行编程的商业编程器。例如，Progressive Resource 和 LLC 开发的 CableAVR，采用串行通信与 PC 机相连接，可使用典型的 Intel Hex 文件作为代码的来源。这些装置可以使用精密的 PC 软件控制编程过程，并具有如自动擦除/写入/校验序列等优点。通过附加的电子设备，还能将串行信号转换为 SPI 通信兼容信号。

许多其他设备也能通过 PC 软件来对微控制器编程，但它们是利用 PC 机的并行端口来实现的。这些编程器通过对并行端口的字位进行操纵，来模拟 SPI 总线与微控制器的通讯。如 Kanda Systems(STK200+/300)和 Vogel Electronik(VTEK-ISP)的编程器，以及 Ateml STK500 在其开发板上自带的编程器都属于这种类型。一些最新的编程器可以直接被 Code VisionAVR IDE 支持，这样微控制器可以直接在 Code VisionAVR IDE 下被编程。不过这些编程器较之先前介绍的串行编程器来说，功能弱一些，适应性稍差一些，速度上也慢一些。

启动装载程序编程

一些 Atmel 设备(如 Atmega163)可以通过一个启动装载程序进行自编程。这类设备能够在板载的启动装载程序(boot loader)的控制下，向 FLASH 代码存储器写入数据。

启动装载程序必须被写入并存放到 FLASH 代码存储器的启动装载部分。这部分的存储器空间由于设置了相应的锁定位，在一次写入之后不允许被重写。

启动装载程序的好处在于它可以为应用程序量身定做。例如，希望通过串行端口更新 FLASH 代码存储器，便可以编写一个启动装载程序，以 Intel Hex 格式化文件等标准格式连续地接收数据，然后再将这些数据存放到 FLASH 存储器中。

为了使用启动装载程序，必须编写一个加载程序，并让它驻留在 FLASH 存储器中称为引导区的部分。然后设置控制位，这样当设备 RESET 后，启动装载程序便开始监控被选中的输入设备，如串行端口。程序指令之前的一个特定字符串，将告诉启动装载程序对 FLASH 存储器中的设备开始重新编程。如果接收的字符串不对，或者经过一定的时间仍然没有接收到控制字符串，启动装载程序将把控制转换到 FLASH 中的代码入口，来执行其中保存的程序。

启动装载程序的一大优点就是在选择好符合要求的输入设备后，可以对各种设备进行字段重新编程。例如，有些设备可以使用串行端口，并可以被 PDA 进行完全重新编程，如掌上电脑。

附录D　安装并使用TheCableAVR

TheCableAVR 是为了对 Atmel 系统自带的可编程微控制器进行连续编程而设计的一种工具。它由一个 Windows 应用程序和一根专用下载电缆组成。通过下载电缆中的特定硬件，PC 应用程序能对许多系统自带的 Atmel 可编程微处理器进行读，编程和校验。图 D-1 和 D-2 所示分别为 TheCableAVR 的硬件和软件的用户界面。

图 D-1　TheCableAVR 硬件

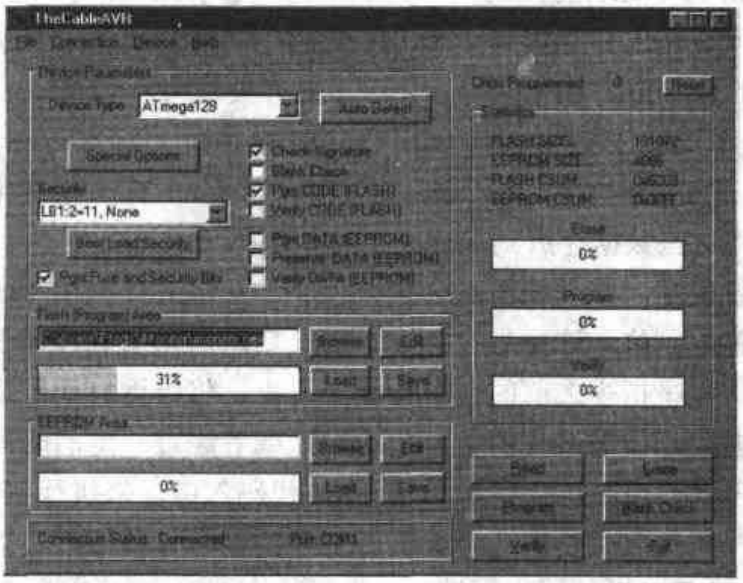


图 D-2　TheCableAVR 软件界面

以下是为确保编程器 PC 应用程序能进行正常操作的最低硬件和软件配置：

- 100%兼容 IBM 386DX 或更高级别的处理器(建议最低为 486)
- Windows 95 或更高操作系统
- 最少 4MB RAM
- 最少 1MB 空闲硬盘空间
- 可用的 PC 串行端口

软件安装

(1) 将标有 Installation Disktte 1 of 2 的软盘插入软盘驱动器中(A：\B：)
(2) 选择窗口中的 Start | Run 命令。
(3) 单击 Browse 按钮。
(4) 定位到软盘驱动器，单击 Open 按钮。
(5) 在 Run 对话框中单击 OK 按钮。随后出现程序安装向导。
(6) 按屏幕上的提示完成软件的安装。
安装结束后，安装程序会在一个名为 TheCable 的程序组中放置一个 TheCableAVR 图标。双击 TheCableAVR 图标即可运行程序。

硬件安装

(1) 将 TheCableAVR 编程器插入 PC 机的串行端口。
(2) 将 IDC 插头插入目标板上一个 10 引脚的 IDC 插座中。
(3) 为目标板接通电源；TheCableAVR 则通过目标板供电。
(4) 选择 TheCableAVR 软件中 Connection | Select Port 命令。
(5) 在出现的 Com Port Setup 对话框中选择好相应的 COM 端口。
(6) 单击 OK 按钮，保存所选的 COM 端口并关闭该对话框。
要检查 PC 机和编程电缆的连接，选择 Connection | Check Connection 命令。如果连接正常，将返回信息：TheCableAVR programming cable is correctly connected.
如果连接有问题，会出现通信错误。请先检查目标板是否接通了电源(编程器上的电源 LED 灯应该是亮着的)。随后，检查是否有其他软件或硬件正在使用所选的 COM 端口。

THECABLEAVR 软件

TheCableAVR 软件允许用户选择要编程的处理器类型、被编程的处理器区域和编程时的安全等级。该软件还能让用户打开并编辑位于被编程处理器的 FLASH 存储器和数据区域中的二进制或十六进制文件。
TheCableAVR 软件也能通过命令行调用。这允许其他软件直接打开并运行 TheCableAVR。

主界面

TheCableAVR 软件的主界面可以分为 6 个部分：菜单、Device Parameters 选项组，Flash(Program)Area 选项组、EEPROM Area 选项组、Statistics 选项组和控制按钮组。图 D-2 即为 TheCableAVR 软件的主界面。

菜单

共有 4 个主要菜单：File、Connection、Device 和 Help。

File 菜单

Open Project — 提示用户选择打开并加载一个以前保存的项目。

Save Project — 提示用户选择一个项目文件名，并在此保存当前的设置。

Load Buffers — 如果某个文件是为 FLASH 和/或 EEPROM 的缓冲区而选择的，将它加载到缓冲区中。

Export Project for Palm Pilot — 打开一个窗口，将项目下载至能在掌上电脑或类似设备上运行的 TheCableAVR 配套软件中。

Exit — 退出应用程序。

Connection 菜单

Select Port — 打开 Com Port Setup 对话框，让用户选择一个 COM 端口。

Check Connection — 尝试与 TheCableAVR 硬件通信。如果能正常通信，将返回信息：TheCableAVR programming cable is correctly connected.

Disconnect — 断开 COM 端口。如果下一个命令需要与硬件通信，则重新建立连接。

Device 菜单

Read — 将所选区域(FALSH 和/或 EEPROM)的数据读入相应的缓冲区中，并更新所有已读缓冲区的校验和(checksum)字段。将缓冲区文件名更新为 Flash_Buff 或 EEPROM_Buff，表示着缓冲区中的内容来自处理器而不是加载的文件。

Progam — 根据已经选项对处理器编程

Verify — 读并且校验处理器所选区域(FLASH 和/或 EEPROM)中的内容。如果发现差异，将通知用户，并将差异之处列表。

Erase — 在处理器上进行芯片内容擦除操作。

Blank Check — 读取芯片的 FLASH 和 EEPROM 区域。如果所有的存储器地址均包括 0xFF，则告知用户这是一块空白芯片；否则，通知用户芯片不是空白的。

Reset — 进行处理器的复位操作。

Read Security — 读当前处理器的安全等级并显示给用户。

Write Security — 将选定的安全等级写入处理器中。

Help 菜单

About TheCableAVR — 显示软件的版本和开发公司信息。

Get Cable Version — 向 TheCableAVR 编程电缆硬件询问版本号，并将该版本号告知用户。也许需要单击该菜单两次才能得到版本信息。有时通信不是按顺序的，因此版本信息的形式可能是 0.0。

Device Parameters 选项组

在 Device Parameters 选项组中，用户可以选择要操作的处理器类型以及对处理器的哪个部

分进行操作。用户还能指定在这些编程过程中如何处理处理器，包括在编程过程中处理器是否检查标识字节，是否进行空白检查等。该选项组中有许多区域：Device Type，Security，Check Signature，Blank Check，Pgm CODE(FLASH)，Verify CODE(FLASH)，Pgm DATA(EEPROM)，Preserve DATA(EEPROM)和 Verify CODE(EEPROM)，如图 D-3 所示。选项组中还有几个按钮：Auto Detect，Special Options 和 Boot Load Security。

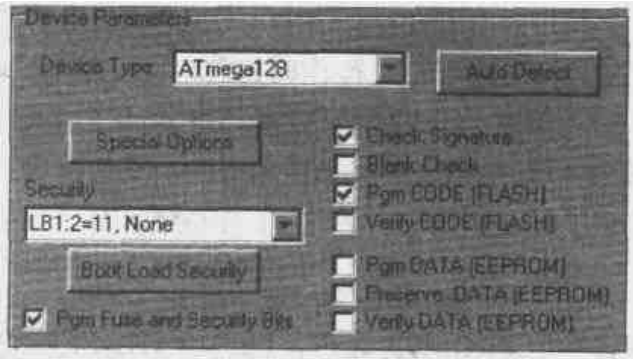


图 D-3　Device Parameters 选项组

Device Type — 通过该下拉列表框可选择目标处理器的类型。选择一个处理器后，程序将自动清空 FLASH 和 EEPROM 缓冲区并使 Statistics 字段复位。如果处理器还有一些特殊的选项可以设置，Special Options 按钮将是可用的。当处理器没有使用限制时，单击 Auto Detect 按钮将为用户自动选择处理器的类型。

Special Options — 该按钮仅当处理器存在可供编程的熔丝(fuse)位时才可用。单击它后，将出现一个 Special Options 对话框，可以设置某个特定处理器的熔丝位。

Auto Detect — 这个按钮将自动选择处理器的类型。单击 Auto Detect 按钮，TheCableAVR 软件读取处理器的标记字节并判定处理器的类型，随后在 Device Type 下拉列表框中相应的选项将被选中，Statistics 字段也会相应更新。

注意：

如果处理器是被读保护的(安全熔丝位为 1)，将返回一个无效的标记位，就无法确定处理器的类型。这时会返回一条信息：Device is Locked. Unable to determine type of device。Device Type 下拉列表框中的内容不会改变。

Security — 这个下拉列表框将允许用户选择 3 种与处理器的锁定位有关的安全等级。

Level 1 — 无锁定位被设置，处理器可以被任意读写。

Level 2 — LB1 被设置，LB2 没有被设置，对 FLASH 和 EEPROM 的编程将被禁止。

Level 3 — LB1 和 LB2 都被设置了，对 FLASH 和 EEPROM 的编程和读操作都将被禁止。

通过 Device 菜单的 Read Security 和 Write Security 子菜单，安全等级信息可以随时从处理器中读出或写入处理器中。

Boot Load Security — 这个按钮允许用户选择要保护的区域和保护级别，可以选择引导区、应用程序区，或者二者都选。

Check Signature — 选择复选框后，TheCableAVR 在对处理器进行编程，读取数据和或校验操作之前，将检查标记字节。如果标记字节与所选的处理器不匹配，用户会得到一个警告信

息：Device Signature Error，且随后的操作将会终止。

Blank Check — 选择该复选框后，TheCableAVR 在编程过程中对芯片进行擦除后，将自动进行空白检查。如果处理器没有被擦除干净，用户将收到一个警告信息：FLASH(or EEPROM) Area is Not Blank，随后的编程操作也将被终止。

Pgm CODE(FLASH)— 这个复选框决定是否对处理器的 FLASH 存储区域进行编程，读取数据和空白检查操作。上述操作只有在 Pgm CODE(FLASH)复选框被选中时才在 FLASH 区域中进行；如果未选中该复选框，则进行上述操作时将忽略 FLASH 区域。

Pgm DATA(EEPROM)— 该复选框决定是否对处理器的 EEPROM 区域进行编程，读取数据和空白检查操作。上述操作只有在 Pgm DATA(EEPROM)复选框时才在 EEPROM 区域中进行；如果未选中该复选框，则进行上述操作时将忽略 EEPROM 区域。

注意：

如果选中 Preserve DATA(EEPROM)复选框，Pgm DATA(EEPROM)复选框将被自动选中且不能去掉。因为 Preserve Data 的部分操作将在 EEPROM 区域中进行，因此需要选中 Pgm DATA(EEPROM)复选框。

Preserve DATA(EEPROM)— 为了对处理器的 FLASH 区域进行编程，首先要进行芯片擦除操作，以清空 FLASH 和 EEPROM 区域中的内容。但在一些情况下，需要保存当前处理器中 EEPROM 中的内容。当 Preserve DATA(EEPROM)复选框被选中后，在芯片擦除操作进行之前，EEPROM 中的内容将被读出并存放到(PC 应用程序的)EEPROM 缓冲区中。在 FLASH 区域执行完编程和校验操作之后，EEPROM 区域将通过 EEPROM 缓冲区进行编程。

注意：

如果 Pgm CODE(FLASH)复选框没有被选中，该复选框将不可选，因为在编程过程中并不进行芯片擦除操作。

Flash(Program) Area 选项组

Flash(Program)Area 选项组由两个主要部分组成：缓冲区信息框和命令按钮(参见图 D-4)。缓冲区信息框将告知用户哪个(如果有多个)文件被选入到 FLASH 缓冲区中，以及某个文件是否被加载到缓冲区中。按钮用于控制选择文件，从文件中将数据加载入缓冲区中，编辑缓冲区以及以文件的形式保存缓冲区中的数据。

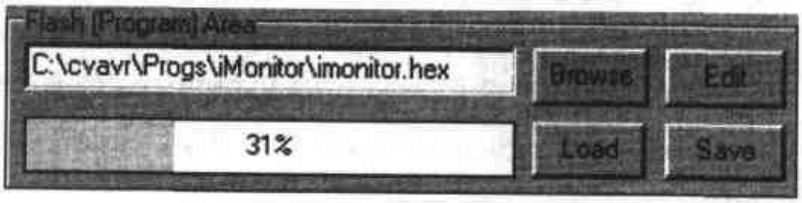


图 D-4　Flash(Program)Area 选项组

Browse — 提示用户选择一个要加载到缓冲区的文件。一旦文件被选中，文件名将被显示在 Browse 按钮左边的文本框中。

Load — 将选择的文件加载到缓冲区。文件名文本框下的进度条将显示有多少可用的内存

被缓冲区占用。

Edit — 打开缓冲区编辑器，允许用户编辑缓冲区中的内容。

Save — 提示用户输入一个文件名，用于保存当前缓冲区中的内容。

EEPROM Area 选项组

EEPROM Area 选项组由两个主要部分组成：缓冲区信息框和命令按钮(参见图 D-5)。缓冲区信息框将告知用户哪个(如果有多个)文件被选入到 EEPROM 缓冲区中，以及某个文件是否被加载到缓冲区中。按钮用来控制选择文件，从文件中将数据加载到缓冲区中，编辑缓冲区以及将缓冲区中的数据保存为文件。

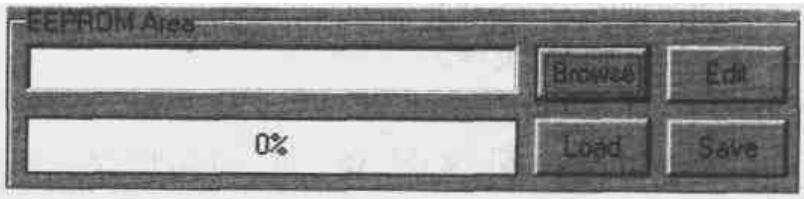


图 D-5　EEPROM Area 选项组

Browse — 提示用户选择一个要被加载到缓冲区中的文件。一旦文件被选中，文件名将被显示在 Browse 按钮左边的文本框中。

Load — 将选中的文件加载到缓冲区。文件名文本框下的进度条将显示有多少可用内存被缓冲区占用。

Edit — 打开缓冲区编辑器，允许用户编辑缓冲区中的内容。

Save — 提示用户输入一个文件名，用于保存当前缓冲区中的内容。

Statistics 选项组

Statistics 选项组显示所选处理器、FLASH 缓冲区和 EEPROM 的缓冲区的信息，以及所执行操作的进度。如图 D-6 所示。

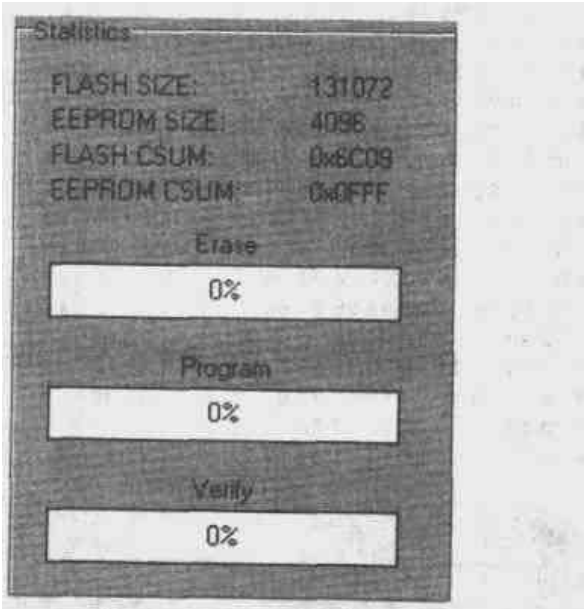


图 D-6　Statistics 选项组

FLASH SIZE	所选处理器的 FLASH 区域容量
EEPROM SIZE	所选处理器的 EEPROM 区域容量
FLASH CSUM	FLASH 缓冲区中内容的校验和
EEPROM CSUM	EEPROM 缓冲区中内容的校验和

进度条代表当前进行的操作完成的百分比，该操作的名称被标在进度条上方。

命令按钮

在软件的主界面上共有 6 个按钮：Read、Program、Verify、Erase、Blank 和 Exit。

Read — 将所选区域(FLASH 和/或 EEPROM)中的数据读入相应的缓冲区中，并更新所有已读缓冲区中的校验和，将缓冲区文件名更新为 Flash_Buff 或 EEPROM_Buff，表示缓冲区中的内容来自处理器而不是一个已加载的文件。

Progam — 根据已所选的选项对处理器进行编程

Verify — 读并且校验处理器的所选区域(FLASH 和/或 EEPROM)。如果发现差异，将通知用户，并将差异之处列表。

Erase — 在处理器上进行芯片擦除操作。

Blank Check — 读取芯片的 FLASH 和 EEPROM 区域。如果所有的内存地址中均包含 0xFF，则告知用户这是一块空白芯片；否则，通知用户芯片不是空白的。

Exit — 退出 TheCableAVR 应用程序。

Buffer Editor 对话框

选定一个 FLASH 或 EEPROM 区域，单击 Edit 按钮，将打开 Buffer Editor 对话框。相应缓冲区(名称显示在窗口的标题栏上)的内容将显示在窗口中，用户可以查看并编辑其中的内容，见图 D-7。

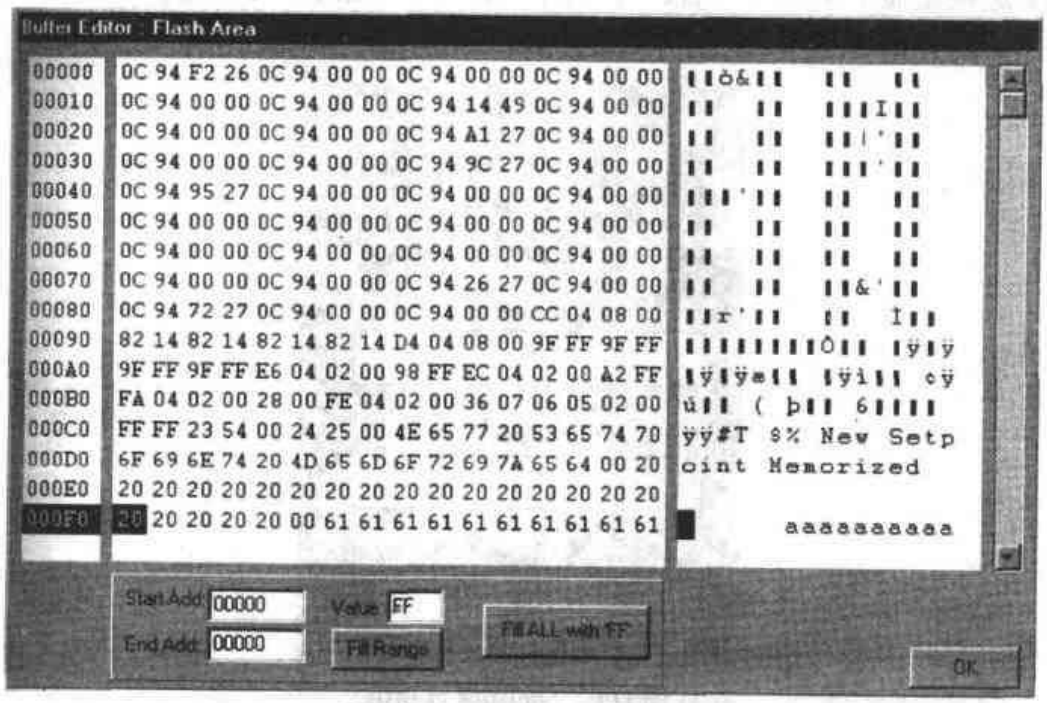


图 D-7　Buffer Editor 对话框

对话框左边的图文框显示了每一行的起始地址，中间的图文框以十六进制的形式显示了缓冲区中的数据，右边的图文框则以 ASCII 字符的形式显示数据。被选中的值以蓝色高亮显示或在其周围显示一个虚线框，虚线框指示其中的值是可以编辑的。单击选中该值，再输入新的值即可完成修改。所有缓冲区中的数据可以以十六进制形式(在中间图文框中编辑)或 ASCII 码字符形式修改(在右边的图文框中编辑)。如果缓冲区中的内容太多，一个屏幕显示不下，可以用最右边的滚动条翻页。

如果需要用个特定的值填充某一范围，将该区域的起始地址、结束地址和要填入的值分别输入对话框下方的 Start Add、End Add 和 Value fields 文本框中，再单击 Fill Range 按钮即可。这样，从 Start Add 到 End Add 地址的所有单元都将被填入一个值。如果想要将整个缓冲区的值都改为 0xFF，只需单击 Fill ALL with 'FF'按钮即可。应注意的是以上需要输入的值均是十六进制的。

单击 OK 按钮将关闭 Buffer Editor。

注意：

在该对话框中设置好要修改的缓冲区中的值之后，相应缓冲区中的数据将被修改。如果用户不能确定修改是否正确，在修改之前将缓冲区的内容以文件的形式备份。

命令行支持

调用应用程序时，命令行支持希望传递如下一些参数：

- 项目文件名(如果与程序不是放在同一目录下，还应指明路径)
- 用户在线标志
  - 当用户在线时，若有错误发生，将出现消息对话框及时告知用户。
  - 如果用户不在线，只能根据程序的返回值来确定程序是否运行成功。

用户在线时的实例为：

```
"C:\Promgram Files\Progressive Resources\TheCableAVR\TheCableAVR.exe"
    "C:\PROJ FILES\PROJ.jsp" "1"
```

用户不在线时的实例为：

```
"C:\Promgram Files\Progressive Resources\TheCableAVR\TheCableAVR.exe"
    "C:\PROJ FILES\PROJ.jsp" "0"
```

表 D-1 列出了程序结束后的返回值及其含义。

表 D-1　返回值

返　回　值	含　义
11	由于设备被锁定，不能保存 EEPROM 中的数据
10	由于设备被锁定，不能仅对 EEPROM 编程
9	不能对安全标志位编程
8	不能对熔丝位编程

(续表)

返 回 值	含　义
5	在芯片擦除操作后，EEPROM 没有被完全清空
4	在芯片擦除操作后，FLASH 没有被完全清空
3	所选设备与标记字段不匹配
2	对 FLASH 或 EEPROM 编程后，发生校验错误
1	不能成功地与 TheCableAVR 硬件正常通信
0	对设备编程成功结束
-1	不能成功地与 TheCableAVR 硬件正常通信
-2	用户取消对设备编程操作
-3	不能打开所选的项目文件
-4	不能打开 FLASH 或 EPPROM 文件
-5	对于所选设备而言，FLASH 文件太大
-6	对于所选设备而言，EEPROM 文件太大
-7	(只针对十六进制或 eep 文件)FLASH 或 EEPROM 文件校验和错误

TheCableAVR 硬件

TheCableAVR 的硬件设备是一根编程电缆，一端是一个 9 引脚的串行端口，另一端是一个 10 引脚的 IDC 插头。电缆上有两个 LED 指示灯，分别是 Power 及 Busy。绿色的 Power LED 灯亮起表示电缆已经(通过目标系统)获得供电。红色的 Busy LED 灯亮起表示电缆正在进行某些编程操作(读，编程，校验等)。

引脚分配

插在目标系统上的 10 引脚 IDC 插头的各引脚功能如表 D-2 所示。

表 D-2　10 引脚 IDC 插头引脚信号说明

引　脚	名　称	说　明	输入/输出
1	Vcc	编程器电源输入(+Vcc)	-
2	NC	无连接	-
3	NC	无连接	-
4	MOSI	SPI—主出从入	O
5	NC	无连接	-
6	MISO	SPI—主入从出	I
7	GND	编程器接地	-
8	SCK1	SPI—连续时钟信号	O
9	GND	编程器接地	-
10	RESET	接目标系统 RESET 控制引脚	O

图 D-8 是 IDC 插头的俯视图。所显示的是一个从插入目标板对应插座的视角看到的 10 引脚 IDC 插头。

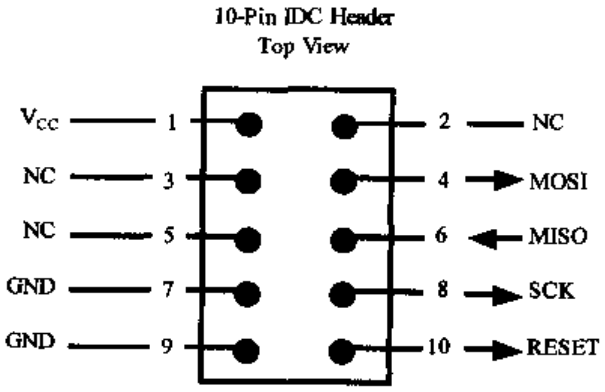


图 D-8　TheCableAVR 10 引脚 IDC 插头

设置目标系统

为了使 TheCableAVR 编程器正常工作，必须在目标系统中设置一些特殊条件。

SPI Enable 熔丝位

为使 TheCableAVR 编程器能正常工作，目标处理器的 SPI Eable 熔丝位(SPIEN)必须被设置为允许(ENABLED)。SPIEN 熔丝位仅在并行编程模式下能被读取和编程，其默认值被设为允许。

电路复位

将 RESET 引脚设置为相应的 RESET 状态，可以初始化 ATMEL AVR 设备的串行编程模式。因此，若想将目标处理器设置为串行编程模式，TheCableAVR 硬件必须能够从外部控制 RESET 引脚的状态。

电源

TheCableAVR 硬件是通过目标应用程序供电的。引脚 1 和引脚 9 分别接 Vcc(+5V 直流)和 GND。

目标系统振荡器

为了访问目标处理器，TheCableAVR 还需要一个目标系统振荡器。该信号的频率必须是目标处理器。可接受的频率可以通过微控制器数据表查到。

注意：

该振荡器可以由外部晶振/共振器或者内部 RC 振荡器产生。

附录E　MegAVR-DEV 开发板

MegAVR-DEV 开发板是由美国印第安娜州印第安娜市的 Progressive Resources LLC 公司设计制造的，供制作样机和实验室设计时使用。该开发板支持 40 引脚 DIP 封装的 AT90S8535，Mega16，Mega163，Mega32 和 Mega323 芯片。该开发板如图 E-1 所示。

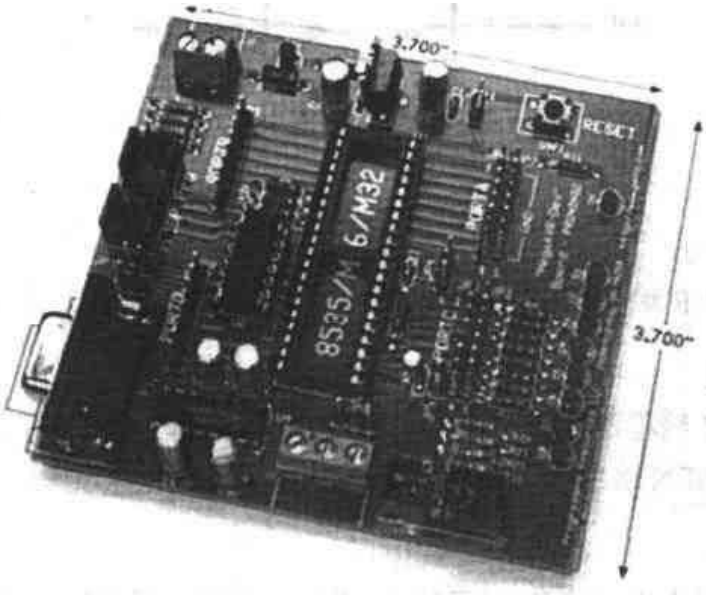


图 E-1　MeagAVR-Dev 开发板

该开发板的特性如下：

- 标准配置包括一个 ATMega163-8PC 处理器
- 预编程配置包括启动装载固件。不需要其他编程硬件。只需通过一个 RS-232 电缆将 MegaAVR-Dev 和 PC 机连接起来，再运行 Windows 应用程序 AVRBL Boot Loader 即可。(程序免费下载地址：*http://www.prllc.com/products/BootApp.zip*)
- RS-232 电缆连接一个 9 引脚的 D 型外壳、螺旋型终端和一个跳线接头
- 最大支持 32KB 可编程系统内建的 FLASH 存储器，1KB 的 EEPROM 和 1KB 的内部 RAM(由所选的处理器类型决定)
- 8 路 10 位模拟输入，使用内部或用户访问引用
- 9 盏由 I/O 端口控制的 LED 灯，其中 8 盏可通过跳线选择
- 可为板载实时操作提供 32kHz 的时钟晶振信号
- 一个通用时钟插座，适用于“固定振荡器”，以及各种晶振，陶瓷谐振器和无源终端
- 0.1”中央头(Centered Header)，能为处理器的特殊功能引脚和 I/O 端口之间提供简单的连接
- 为系统内编程和调试提供了 10 引脚角的 极化(Polarized)ISP 和 JTAG 连接。

MegaAVR-Devs 也能通过 RS-232 电缆，使用相应的启动装载程序进行编程

- 板载 8～38V 直流电输入调节器，有 LED 指示灯显示
- 为终端提供 5V 直流电压，250mA 的电流输出

想了解更详细的信息请访问 Progressive Resources 网站：http://www.prllc.com

连接线、插头和跳线如图 E-2 所示。

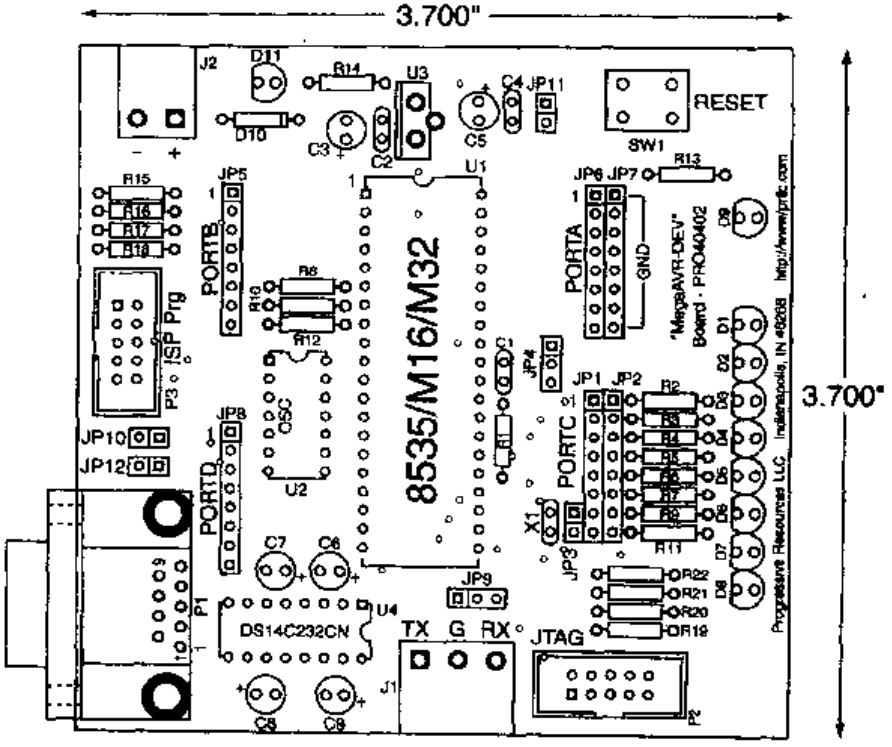


图 E-2　连接线、插头和跳线

规格说明

规格说明如表 E-1 所示。

表 E-1　MegaAVR-Dev 规格

规　格	范　围
电压范围	8V～38V DC
功率	250 mW(标称值)
尺寸	W3.7 英寸× H 3.7 英寸
底座	4 个橡皮座脚
重量	约 3 oz
工作温度	0° C ～ +60°C
保存温度	0° C ～ +85°C
湿度	+85°C 下 0%至 95%(不凝结)

原理图

原理图如图 E-3 所示。

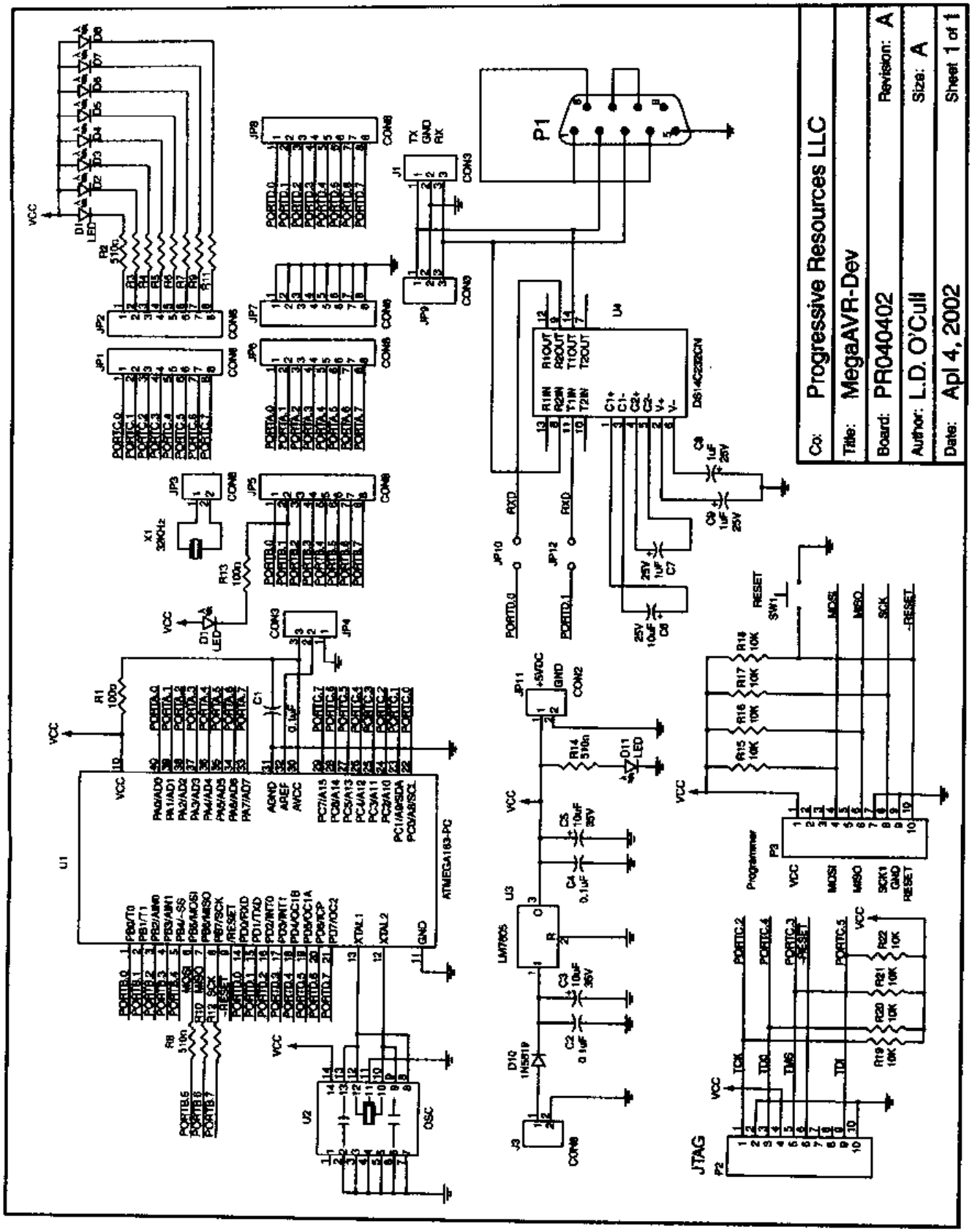


图 E-3　MegaAVR-Dev 示意图

附录F　ASCII字符表

美国信息交换标准码(American Standard Code for Information Interchange)

十进制值	八进制值	十六进制值	二进制值	字　符
000	000	000	00000000	NUL
001	001	001	00000001	SOH
002	002	002	00000010	STX
003	003	003	00000011	ETX
004	004	004	00000100	EOT
005	005	005	00000101	ENQ
006	006	006	00000110	ACK
007	007	007	00000111	BEL
008	010	008	00001000	BS
009	011	009	00001001	HT
010	012	00A	00001010	LF
011	013	00B	00001011	VT
012	014	00C	00001100	FF
013	015	00D	00001101	CR
014	016	00E	00001110	SO
015	017	00F	00001111	SI
016	020	010	00010000	DLE
017	021	011	00010001	DC1
018	022	012	00010010	DC2
019	023	013	00010011	DC3
020	024	014	00010100	DC4
021	025	015	00010101	NAK
022	026	016	00010110	SYN
023	027	017	00010111	ETB
024	030	018	00011000	CAN
025	031	019	00011001	EM
026	032	01A	00011010	SUB
027	033	01B	00011011	ESC
028	034	01C	00011100	FS

(续表)

十进制值	八进制值	十六进制值	二进制值	字　符
029	035	01D	00011101	GS
030	036	01E	00011110	RS
031	037	01F	00011111	US
032	040	020	00100000	SP(Space)
033	041	021	00100001	!
034	042	022	00100010	"
035	043	023	00100011	#
036	044	024	00100100	$
037	045	025	00100101	%
038	046	026	00100110	&
039	047	027	00100111	`
040	050	028	00101000	(
041	051	029	00101001	)
042	052	02A	00101010	*
043	053	02B	00101011	+
044	054	02C	00101100	,
045	055	02D	00101101	-
046	056	02E	00101110	.
047	057	02F	00101111	/
048	060	030	00110000	0
049	061	031	00110001	1
050	062	032	00110010	2
051	063	033	00110011	3
052	064	034	00110100	4
053	065	035	00110101	5
054	066	036	00110110	6
055	067	037	00110111	7
056	070	038	00111000	8
057	071	039	00111001	9
058	072	03A	00111010	:
059	073	03B	00111011	;
060	074	03C	00111100	<
061	075	03D	00111101	=
062	076	03E	00111110	>
063	077	03F	00111111	?

(续表)

十进制值	八进制值	十六进制值	二进制值	字　符
064	100	040	01000000	@
065	101	041	01000001	A
066	102	042	01000010	B
067	103	043	01000011	C
068	104	044	01000100	D
069	105	045	01000101	E
070	106	046	01000110	F
071	107	047	01000111	G
072	110	048	01001000	H
073	111	049	01001001	I
074	112	04A	01001010	J
075	113	04B	01001011	K
076	114	04C	01001100	L
077	115	04D	01001101	M
078	116	04E	01001110	N
079	117	04F	01001111	O
080	120	050	01010000	P
081	121	051	01010001	Q
082	122	052	01010010	R
083	123	053	01010011	S
084	124	054	01010100	T
085	125	055	01010101	U
086	126	056	01010110	V
087	127	057	01010111	W
088	130	058	01011000	X
089	131	059	01011001	Y
090	132	05A	01011010	Z
091	133	05B	01011011	[
092	134	05C	01011100	\
093	135	05D	01011101	]
094	136	05E	01011110	^
095	137	05F	01011111	—
096	140	060	01100000	`
097	141	061	01100001	a
098	142	062	01100010	b

(续表)

十 进 制 值	八 进 制 值	十六进制值	二 进 制 值	字　符
099	143	063	01100011	c
100	144	064	01100100	d
101	145	065	01100101	e
102	146	066	01100110	f
103	147	067	01100111	g
104	150	068	01101000	h
105	151	069	01101001	i
106	152	06A	01101010	j
107	153	06B	01101011	k
108	154	06C	01101100	l
109	155	06D	01101101	m
110	166	06E	01101110	n
111	167	06F	01101111	o
112	160	070	01110000	p
113	161	071	01110001	q
114	162	072	01110010	r
115	163	073	01110011	s
116	164	074	01110100	t
117	165	075	01110101	u
118	166	076	01110110	v
119	167	077	01110111	w
120	170	078	01111000	x
121	171	079	01111001	y
122	172	07A	01111010	z
123	173	07B	01111011	{
124	174	07C	01111100	|
125	175	07D	01111101	}
126	176	07E	01111110	~
127	177	07F	01111111	DEL

附录G　AVR指令集汇总

附录 G　AVR 指令集汇总

AVR 处理器上的完整指令可以在 Atmel 的 Web 站点 http://www.atmel.com 找到。

AVR 指令集标准命名：												
状态寄存器(SREG)	符号	寄存器与操作数：	I/O 寄存器									
C：进位(Carry)标志	+ 加法	Rd：目标(源)寄存器	RAMPX、RAMPY、RAMPZ：同X、Y和Z连接起形成间接地址的寄存器，大于64K									
Z：零(Zero)标志	& 与	Rr：源寄存器										
N：负(Negative)标志	<- 赋值	R：返回指令执行的结果										
V：两个完成的溢出指示器	= 等于	K：常量文字或数据字节(8 位)										
S：有符号测试的 N^V 标志	^ 异或	k：程序计数器的常量地址数据	栈									
H：半进位标志	* 乘法	b：寄存器文件或 I/O 寄存器中的位(3 位)										
T：由 BLD 和 BST 指令使用的传输(Transfer)位	| 或	s：状态寄存器(SREG)中的位(3 位)	STACK：用于返回地址和紧缺寄存器的栈									
I：全局中断有效/禁用标志	- 减法	X,Y,X：间接地址寄存器										
		P：I/O 端口地址										
		q：直接寻址位移(6 位)	SP：指向栈的栈指针									
						SREG						

(续表)

指　令	操作数	描　　述	操　　作	Z	C	N	V	H	S	T	I	#Clocks
数学与逻辑指令												
ADC	Rd,Rr	对两个寄存器进位执行加操作	Rd <- Rd + Rr + C	•	•	•	•	•				1
ADD	Rd,Rr	将两个寄存器相加	Rd <- Rd + Rr	•	•	•	•	•				1
ADIW	Rdl,K	直接向文字添加	Rdh:Rdl <-Rdh:Rdl + K	•	•	•	•		•			2
AND	Rd,Rr	逻辑"与"寄存器	Rd. <- Rd & Rr	•		•	•					1
ANDI	Rd,K	逻辑“与”寄存器和常量	Rd. <- Rd & K	•		•	•					1
CBR	Rd,K	寄存器中的清除位	Rd <- Rd &($FF -K)	•		•	•					1
CLR	Rd	清除寄存器	Rd <- Rd ^ Rd	•		•	•					1
COM	Rd	1 的补	Rd <- $$Rd - Rd	•	•	•	•					1
DEC	Rd	自减	Rd <- Rd - 1	•		•	•					1
EOR	Rd,Rr	异或寄存器	Rd <- Rd ^ Rr	•		•	•					1
FMUL	Rd,Rr	无符号小数乘法	R1:R0 <- (Rd X Rr)<< 1	•	•							2
FMULUS	Rd,Rr	有符号小数乘法	R1:R0 <- (Rd X Rr)<< 1	•	•							2
FMULUSU	Rd,Rr	有符号小数同无符号小数乘法	R1:R0 <- (Rd X Rr)<< 1	•	•							2
INC	Rd	自增	Rd <- Rd + 1	•		•	•					1
MUL	Rd,Rr	无符号乘法	R1:R0 <- Rd X Rr	•	•							2
MULS	Rd,Rr	有符号乘法	R1:R0 <- Rd X Rr	•	•							2
MULSU	Rd,Rr	有符号同无符号乘法	R1:R0 <- Rd X Rr	•	•							2
NEG	Rd	2 的补	Rd <- $00 - Rd	•	•	•	•	•				1
OR	Rd,Rr	逻辑“或”寄存器	Rd.<- Rd | Rr	•		•	•					1
ORI	Rd,K	逻辑“或”(直接)寄存器和常量	Rd.<- Rd | K	•		•	•					1

（续表）

指　令	操作数	描　　述	操　　作	Z	C	N	V	H	S	T	I	#Clocks
SBC	Rd,Rr	通过进位两个寄存器来执行减法	Rd <- Rd - Rr - C	•	•	•	•	•				1
SBCI	Rd,K	通过进位来自寄存器的常量来执行减法	Rd <- Rd - K - C	•	•	•	•	•				1
SBIW	Rdl,K	直接减字	Rdh:Rdl <-Rdh:Rdl - K	•	•	•	•		•			2
SBR	Rd,K	设置寄存器中的位	Rd <- Rd | K	•		•	•					1
SER	Rd	设置寄存器	Rd <- $FF									1
SUB	Rd,Rr	减两个寄存器	Rd <- Rd - Rr	•	•	•	•	•				1
SUBI	Rdl,K	(直接)减来自寄存器中的常量	Rd <- Rd - K	•	•	•	•	•				1
TST	Rd	对零或负数进行测试	Rd <- Rd & Rd	•		•	•					1
分支指令												
BRBC	s,k	状态标志清除时分支	if (SREG(s)=0) then PC<-PC+k+1									1/2
BRBS	s,k	状态标志设置时分支	if (SREG(s)=1) then PC<-PC+k+1									1/2
BRCC	k	进位清除时分支	if (C=0) then PC<-PC+k+1									1/2
BRCS	k	进位设置时分支	if (C=1) then PC<-PC+k+1									1/2
BREQ	k	相等时分支	if (Z=1) then PC<-PC+k+1									1/2
BRGE	k	大于或者等于时分支,有符号	if (N^V) then PC<-PC+k+1									1/2
BRHC	k	半进位标志清除时分支	if (H=0) then PC<-PC+k+1									1/2
BRHS	k	半进位标志设置时分支	if (H=1) then PC<-PC+k+1									1/2

(续表)

指　令	操作数	描　　述	操　　作	Z	C	N	V	H	S	T	I	#Clocks
BRID	k	中断禁用时分支	if (I=0) then PC<-PC+k+1									1/2
BRIE	k	中断有效时分支	if (I=1) then PC<-PC+k+1									1/2
BRLO	k	降低时分支	if (C=1) then PC<-PC+k+1									1/2
BRLT	k	小于零时分支，有符号	if (N^V=1) then PC<-PC+k+1									1/2
BRMI	k	相减时分支	if (N=1) then PC<-PC+k+1									1/2
BRNE	k	不相等时分支	if (Z=0) then PC<-PC+k+1									1/2
BRPL	k	为正时分支	if (N=0) then PC<-PC+k+1									1/2
BRSH	k	相同或者较高时分支	if (C=0) then PC<-PC+k+1									1/2
BRTC	k	标志清除时分支	if (T=0) then PC<-PC+k+1									1/2
BRTS	k	标志设置时分支	if (T=1) then PC<-PC+k+1									1/2
BRVC	k	溢出标志清除时分支	if (V=0) then PC<-PC+k+1									1/2
BRVS	k	溢出标志设置时分支	if (V=1) then PC<-PC+k+1									1/2
CALL	k	直接子例程调用	PC<-k									4
CP	Rd,Rr	比较	Rd-Rr	•	•	•	•	•				1
CPC	Rd,Rr	使用进位进行比较	Rd-Rr-C	•	•	•	•	•				1
CPI	Rd,K	直接比较寄存器	Rd-K	•	•	•	•	•				1
CPSE	Rd,Rr	比较，相等时跳过	if(Rd=Rr)PC <-PC+2or3									1/2/3
ICALL		对(Z)间接调用	PC<-Z									3
IJMP		间接跳到(Z)	PC<-Z									2

(续表)

指　令	操作数	描　　述	操　　作	Z	C	N	V	H	S	T	I	#Clocks
JMP	k	直接跳转	PC<-k									3
RCALL	k	相对子例程调用	PC<-PC+k+1									3
RET		子例程返回	PC<-STACK									4
RETI		中断返回	PC<-STACK									4
RJMP	k	相对跳转	PC<-PC+k+1									2
SBIC	P,b	I/O 寄存器中的位清除时跳过	if(P(b)=0) PC<-PC+2 or 3									1/2/3
SBIS	P,b	I/O 寄存器中的位设置时跳过	if(P(b)=1) PC<-PC+2 or 3									1/2/3
SBRC	Rr,b	寄存器中的位清除时跳过	if(Rr(b)=0) PC<-PC+2 or 3									1/2/3
SBRS	Rr,b	寄存器中的位设置时跳过	if(Rr(b)=1) PC<-PC+2 or 3									1/2/3
数据传输指令												
ELPM		扩展加载程序内存	R0<-(RAMPZ:Z)									3
ELPM	Rd,Z	扩展加载程序内存	Rd<-(RAMPZ:Z)									3
ELPM	Rd,Z+	扩展加载程序内存和 Post-Inc	Rd<-(RAMPZ:Z), RAMPZ:Z<-(RAMPZ:Z)+1									3
IN	Rd,P	端口内	Rd<-P									1
LD	Rd,X	间接加载	Rd<-(X)									2
LD	Rd,X+	间接加载和 Post-Inc.	Rd<-(X), X<-X+1									2
LD	Rd,-X	间接加载和 Pre-Inc.	X<-X-1,Rd<-(X)									2
LD	Rd,Y	间接加载	Rd<-(Y)									2
LD	Rd,Y+	间接加载 Post-Inc.	Rd<-(Y),Y<-Y+1									2
LD	Rd,-Y	间接加载 Pre-Dec.	Y<-Y+1, Rd<-(Y)									2

（续表）

指　令	操作数	描　　述	操　　作	Z	C	N	V	H	S	T	I	#Clocks
LD	Rd,Z	间接加载	Rd<-Z									2
LD	Rd,Z+	间接加载和 Post-Inc.	Rd<-Z, Z<-Z+1									2
LD	Rd,-Z	间接加载和 Pre-Dec.	Z<-Z-1, Rd<-(Z)									2
LDD	Rd,Y +q	使用位移进行间接加载	Rd<-(Y+q)									2
LDD	Rd,Z+ q	使用位移进行间接加载	Rd<-(Z+q)									2
LDI	Rd,K	直接加载	Rd<-K									1
LDS	Rd,k	从 SRAM 直接加载	Rd<-(k)									2
LPM		加载程序内存	R0<-(Z)									3
LPM	Rd,Z	加载程序内存	Rd<-(Z)									3
LPM	Rd,Z+	加载程序内存和 Post-Inc.	Rd<-(Z), Z<-Z+1									3
MOV	Rd,Rr	在寄存器之间移动	Rd<-Rr									1
MOVW	Rd,Rr	复制寄存器字	Rd+1:Rd<-Rr+1: Rr									1
OUT	P,Rr	输出端口	P<-Rr									1
POP	Rd	从栈中弹出寄存器	Rd<-STACK									2
PUSH	Rr	将寄存器压入栈中	STACK<-Rr									2
SPM		存储程序内存	(Z)<-R1:R0									-
ST	X,Rr	间接存储	(X)<-Rr									2
ST	X+,Rr	间接存储和 Post-Inc.	(X)<-Rr, X<-X+1									2
ST	-X,Rr	间接存储和 Pre-Dec.	X<-X-1,(X)<-Rr									2
ST	Y,Rr	间接存储	(Y)<-Rr									2
ST	Y+,Rr	间接存储和 Post-Inc.	(Y)<-Rr,Y<-Y+1									2
ST	-Y,Rr	间接存储和 Pre-Dec.	Y<-Y-1,(Y)<-Rr									2

（续表）

指　令	操作数	描　　述	操　　作	Z	C	N	V	H	S	T	I	#Clocks
ST	Z,Rr	间接存储	(Z)<-Rr									2
ST	Z+,Rr	间接存储和 Post-Inc.	(Z)<-Rr, Z<-Z+1									2
ST	-Z,Rr	间接存储和 Pre-Dec.	Z<-Z-1,(Z)<-Rr									2
STD	Y+q, Rr	使用位移进行间接存储	(Y+q)<-Rr									2
STD	Z + q,Rr	使用位移进行间接存储	(Z+q)<-Rr									2
STS	k, Rr	直接存储到 SRAM	(k)<-Rr									2
位和位测试指令												
ASR	Rd	算术右移	Rd(n)<-Rd(n+1), n=0..6	•	•	•	•					1
BCLR	s	标志清除	SREG(s)<- 0	SREG								1
BLD	Rd,b	从 T 到寄存器加载位	Rd(b)<- T							•		1
BSET	s	标志设置	SREG(s)<- 1	SREG								1
BST	Rr,b	从寄存器到 T 存储位	T<-Rr(b)							•		1
CBI	P,b	清楚 I/O 寄存器中的位	I/O(P,b)<- 0									2
CLC		清除进位	C<-0		•							1
CLH		清除 SREG 中的半进位标志	H<-0					•				1
CLI		禁用全局中断	I<-0								•	1
CLN		清除负标志	N<-0			•						1
CLS		清除有符号测试标志	S<-0						•			1
CLT		清除 SREG 中的 T	T<-0							•		1
CLV		清除 2 的位移溢出	V<-0				•					1
CLZ		清除零标志	Z<-0	•								1
LSL	Rd	逻辑左移	Rd(n+1)<-Rd(n),Rd(0)<-0	•	•	•	•					1

(续表)

指　令	操作数	描　　述	操　　作	Z	C	N	V	H	S	T	I	#Clocks
LSR	Rd	逻辑右移	Rd(n)<-Rd(n+1),Rd(7)<-0	•	•	•	•					1
ROL	Rd	通过进位向左旋转	Rd(0)<- C,Rd(n+1)<-Rd(n),C<-Rd(7)	•	•	•	•					1
ROR	Rd	通过进位向右旋转	Rd(7)<- C,Rd(n)<-Rd(n+1),C<-Rd(0)	•	•	•	•					1
SBI	P,b	设置 I/O 寄存器中的位	I/O(P,b)<- 1									2
SEC		设置进位	C<-1		•							1
SHE		设置 SREG 中的半进位标志	H<-1					•				1
SEI		全局中断有效	I<-1								•	1
SEN		设置负标志	N<-1			•						1
SES		设置有符号测试标志	S<-1						•			1
SET		设置 SREG 中的 T	T<-1							•		1
SEV		设置 2 的位移溢出	V<-1	•								1
SEZ		设置零标志	Z<-1									1
SWAP	Rd	少量交换	Rd(3..0)<--Rd(7..4)<-Rd(3..0)									1
MCU 控制指令												
NOP		无操作										1
SLEEP		休眠	引用 Sleep 函数描述的厂商数据									1
WDR		Watchdog 复位	引用 WDR/timer 函数描述的厂商数据									1
BREAK		中断	只适合用于"On-Chip-Debug",引用用于描述的厂商数据									N/A

附录H　部分练习答案

第1章

***2. 根据以下要求创建正确的声明：**

A. 常量‘x’，值为789

答案：

```
flash int x = 789;
    或
Const int x = 789;
    或
Const int flash x = 789;
```

提示：

flash和const指的是同样一种不易丢失数据的存储器类型。在本例中声明的是一个整型值，因为它是用16位的数值来表示的(+/－32767或0～65535)，范围比字符型变量(char或者unsigned char)大，后者是用8位的数值表示的(+/－128或0～255)。

B. 变量‘fred’，取值范围3～456

答案：

```
unsigned int fred;
    或
int fred;
```

提示：

由于要表示的是正数，将变量定义为int或unsigned int都可以，但要数值的范围限制了不能定义为char或unsigned char类型。

C.变量‘sensor_out’，能表示－10～+45之间的值

答案：

```
signed char sensor_out;
```

提示：

根据要表示数的范围应该将其定义为字符型(+/－127)。

D. 有 10 个元素的数组变量，每个元素的取值范围为－23～345

答案：

```
int array[10];
```

提示：

根据要表示数的范围，不应该将其定义为字符类型，因为它超过了 128(8 位)。

E. 一个值为 Press here to end 的字符串常量

答案：

```
flash char press_string[] = "Press here to end";
    或
const char flash press_string[] = "Press here to end";
    或
const char press_string[] = "Press here to end";
```

提示：

flash 和 const 指的是一种相同的不易丢失数据的存储器类型。以上 3 种声明可生成相同的代码

F. 指针 array_ptr，指向由 3～567 的数字组成的数组

答案：

```
unsigned int *array_ptr;
```

提示：

声明一种类型的指针时，它的大小不是指本身的大小，而是所指向对象的表示范围。在本例中，指针所指的值比 8 位数大，但可用 16 位整数表示。

G. 使用枚举类型将 uno，dos 和 tre 的值分别设为 21，22，23

答案：

```
enum { uno=21, dos, tres };
```

提示：

一个枚举类型默认的起始标号值为 0。因此必须将对第 1 个变量指定其起始值，其余变量会依次变化。

***4. 像在条件语句中一样，判断下面的值是真(TURE)还是假(FALSE)，其中 x = 0x45；y = 0xC6**

A. (x == 0x45)

答案：

TURE

B. (x | y)

答案：

TURE 因为 0x45 | 0xC6 = 0xC7,不等于 0

C. (x > y)

答案：

FALSE

D. (y-0x06 = 0x0C)

答案：

FALSE，因为 0xC6 - 0x06 = 0xC0，不等于 0x0C

***6. 对程序代码段运行后的变量求值**

```
unsigned char cntr = 10;
unsigned int value = 10;
do
{
    value++;
}while(cntr<10)
//value = ??      cntr = ??
```

答案：

value = 11　　cntr = 10

提示：

变量 *value* 的值增加了，而变量 *cntr* 的值没有改变。

*10. 编写一段 C 语言代码，声明一个合适的数组，并将 2 的 21～26 次幂放到该数组中

```
unsigned char twos [6];
//size array to hold the 6 figures..
char x, y;     //dedare a couple of indexes
y = 0;
for(x = 1; x!=0x80; x<<=1)
                //shift a one left until it is 2^7
    twos[y++] = x;
                //store value into array and increment index
```

第 2 章

*2. 介绍下列类型的存储器，并描述它们的用法

A. FLASH Code Memory

B. Data Memory

C. EEPROM Memory

部分答案：EEPROM 存储器是一种可记忆的存储器，当处理器断电后存放在其中的数据不会丢失。但它有写次数的限制，因此一般不用来存放经常变动的数据。

*6. 编写一段 C 语言程序初始化外部中断 1，使外部中断引脚出现下降沿信号时触发中断

答案：

```
GIMSK = 0b1x00000; //enable INT1
MCUCR = 0bxxxx10xx;
//set INT1 to react to a falling edge
```

提示：

x 不是一个无法接受的数字，但在此用来表示该位并不影响结果，只有被设置为 0 或 1 的位才对结果有影响。

***8. 编写一段 C 语言代码初始化端口 C 的引脚，使它的高四位作为输入，低四位作为输出**

答案：

```
DDRC = 0x0f;          // intialize Port C
```

***11. 当异步收发器以 9600 波特的速率传送字符 H 时，画出输出端的波形图。除了波形外，该图应该显示正确的电压和位宽度。**

答案：从 ASCII 表中，可知 H 的值为 0x48，每位的传输时间为 1/9600 = 104 微秒。在图 H-1 中，每一列代表一个位，且宽度为 104 微秒。

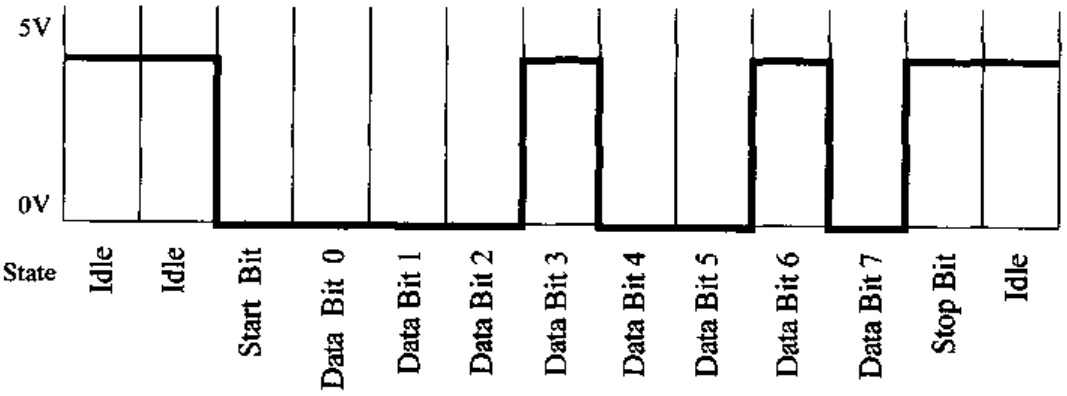


图 H-1　输出端的波形图

***13. H-1 列举了关于模数转换的数据，计算其中缺少的数据**

表 H-1　模数转换数据

V_{in}	$V_{fullscale}$	Digital Out	# of bits
4.2V	10V	107 或 0x6B	8
1.6V	5V		10
	5V	123	10
	10V	223	8

第一行答案(其余由读者自己完成):

$V_{in} / V_{fullscale} = X / 2^n - 1$

$4.2V / 10V = X / 2^8 - 1$

$4.2(2^8-1)/ 10V = X = 107_{10} = 0x6B$

提示：

由于定义的是整型数据，答案的小数部分被去除了。

第 3 章

*1. 用#define 指令编写一个宏，为 putchar()函数生成一个别名为 send_the_char()的函数

答案：

```
#define send_the_char(a) putchar(a)
```

*4. 编写一个函数打印输出它的编译日期和时间。使用编译器的内部标记名来得到日期和时间值

答案：

```
putsf("\n\rCompile Date :\r");
putsf(__DATE__);
putsf("\n\rCompile Time :\r");
putsf(__TIME__);
```

提示：

putsf()函数将一个存放在 FLASH 存储器中的字符串(常量字符串)打印输出到标准输出端。

*6. 编写一个函数，输入 16 位的十六进制值，然后打印输出相应的二进制值。

提示：

库函数中没有标准的二进制值输出函数——因此需要自己编写！

一个可行的解法：

```
void put_bin16(unsigned int value)
{
    int k;                          // working counter
    for(k=0, k<16, k++)      // do 16 digits
    {
        if(value &(1<<k))   // if 2^k then print a '1'
            puchar('1');
        else
            putchar('0');
    }                               // else print a '0'
}
```

提示：

本例的做法是将数 1 左移，以此来对 16 位整型数的每一位分别产生一个掩码(mask)，通过掩码来测试每一位。