AVR单片机

C语言开发入门指导

沈 文 Eagle lee 詹卫前 编著

清华大学出版社

策划编辑：曾　刚
责任编辑：肖　丽
封面设计：秦　铭

AVR单片机

C语言开发入门指导

本书介绍了ICCAVR编译器使用C语言的有关知识，也穿插介绍ICCAVR与常用的其他C编译器使用C语言的一些异同点，并简单介绍ICCAVR的集成环境和ICCAVR 6.26C能支持的库函数。本书重点放在如何利用C语言来操作AVR单片机的硬件资源，以及如何编写一些实用的程序段，最后再通过一些简单的应用实例来说明如何使用C语言来开发AVR芯片。本书适合开发AVR单片机的工程技术人员，也适合大中专院校电子专业的学生学习使用。

ISBN 7-302-06530-6
9 787302 065302 >
定价：40.00元

AVR 单片机 C 语言开发入门指导

沈　文　Eagle lee　詹卫前　编著

清 华 大 学 出 版 社

北　京

内 容 简 介

本书介绍了 ICCAVR 编译器使用 C 语言的有关知识，也穿插介绍 ICCAVR 与常用的其他 C 编译器使用 C 语言的一些异同点，并简单介绍 ICCAVR 的集成环境和 ICCAVR 6.26C 能支持的库函数。本书重点放在如何利用 C 语言来操作 AVR 单片机的硬件资源，以及如何编写一些实用的程序段，最后再通过一些简单的应用实例来说明如何使用 C 语言来开发 AVR 芯片。本书适合开发 AVR 单片机的工程技术人员，也适合大中专院校电子专业的学生学习使用。

图书在版编目（CIP）数据

AVR 单片机 C 语言开发入门指导/沈文、Eagle Ice 詹卫前编著. —北京：清华大学出版社，2003

ISBN 7-302-06530-6

I.A... II. ①沈... ②Lee... ③詹... 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 026186 号

出 版 者：清华大学出版社（北京清华大学学研大厦，邮编 100084）

http://www.tup.tsinghua.edu.cn

责任编辑：肖 丽

印 刷 者：清华大学印刷厂

发 行 者：新华书店总店北京发行所

开 本：787×1092 1/16 **印张**：31 **字数**：710 千字

版 次：2003 年 5 月第 1 版 2003 年 5 月第 1 次印刷

书 号：ISBN 7-302-06530-6/TP・4897

印 数：0001～5000

定 价：40.00 元

前　言

ATMEL 公司是世界上著名的高性能、低功耗、非易失性存储器和数字集成电路的一流半导体制造公司，公司最引人注目的是 EEPROM 电可擦除技术、闪速存储器技术和高质量、高可靠的生产技术。在 CMOS 器件生产领域中，ATMEL 的先进设计水平、优秀的生产工艺及封装技术，一直处于世界领先地位，ATMEL 将这些技术应用于单片机的生产，使单片机也具有优秀的品质。

在 20 世纪 90 年代初，ATMEL 公司率先把 MCS-51 内核与其擅长的 Flash 技术相结合，推出轰动业界的 AT89C51/52、1051/2051 系列单片机，取代了 8751 系列单片机，至今还在大规模生产。1997 年，ATMEL 公司挪威设计中心的 A 先生与 V 先生出于市场需求考虑，推出全新配置的精简指令集(Reduced Instruction Set CPU)单片机，简称为 AVR。与其相比，采用复杂指令集 CISC(Complex Instruction Set CPU)的单片机(如 51 系列)，在效率、速度及指令格式上就显得比较复杂，更不适合于在嵌入式系统中使用。近几年来，随着 AVR 单片机不断改进并持续推出新的品种，现已形成系列产品，其中 ATtiny、AT90 与 ATmega 分别对应为低、中、高档产品，在国外已得到了广泛的应用。特别是近一两年，随着高档 ATmega 系列单片机的大幅降价，部分产品的价格甚至低于同类中档 AT90 系列单片机的价格，其性价比更高，在国内也有广阔的市场前景。

本书适合开发 AVR 单片机的工程技术人员，也适合大中专院校电子专业的学生学习使用。本书的编写以介绍用 C 语言开发 AVR 单片机入门为目的，假定读者已学习过 AVR 单片机的硬件结构及汇编指令、C 语言(如 Turbo c)的基础知识。本书以介绍 ICCAVR 编译器为主，希望通过结合应用实例的讲解，使读者能够熟练使用 ICCAVR 编译器来开发 AVR 单片机。本书还对国内市场上常见的 GCCAVR、CodeVision 和 IAR 编译器进行了简单的介绍，使读者能够举一反三，掌握其中一种或几种 C 编译器的使用。实际上，一般情况只要精通其中的一种 C 编译器就足够应用了。

本书先介绍 ICCAVR 编译器使用 C 语言的有关知识，也穿插介绍了 ICCAVR 与常用的其他 C 编译器使用 C 语言的一些异同点，并简单介绍了 ICCAVR 的集成环境和 ICCAVR 6.26C 支持的库函数。本书重点放在如何利用 C 语言来操作 AVR 单片机的硬件资源，以及如何编写一些实用的程序段，最后再通过一些简单的应用实例来说明如何使用 C 语言来开发 AVR 芯片。至于用 C 语言开发 AVR 的技巧与提高，以及目前正在单片机上开始流行的实时操作系统、USB 和 TCP/IP 等完整应用实例，则放到编者正在编写的《AVR 单片机 C 语言开发应用与提高》一书中讲述。

编者推荐读者使用 ICCAVR 编译器，因为 ICCAVR 提供 30 天的免费试用，在试用期

内等同于标准版(除了部分功能被限制，如代码压缩优化等)，能够生成 INTEL HEX 格式的烧写文件，而且价格低廉(相对于 keil、IAR 等软件而言)、升级速度快，在国内又有双龙电子提供技术支持，用ICCAVR编写的程序能够很快转到IAR中使用。当然，IAR和CodeVision以及GCCAVR相对ICCAVR而言还是各有所长的，特别值得一提的是free的GCCAVR，相对于ICCAVR还是很有竞争优势的，这几种C编译器目前还有市场，也说明了这一点。

由于 ICCAVR 升级较快，至本书截稿时，ICCAVR 的最新版本为 6.26C 版(2003 年 1 月 10 日推出)，读者需要新版的软件，可以到 imagecraft 公司网站(http://www.imagecraft.com/software)或其国内总代理双龙电子公司网站下载。

本书在出版前，ICCAVR 和 GCCAVR 均推出新版软件，所以再次做了修订，其中与 ICCAVR 有关的内容已全部按 6.26C 版修订，因此本书所讲述 ICCAVR 均针对 6.26C 版。本书也针对新版的 GCCAVR(20030115 版)进行了修订，特别安装了过程与函数库，尽管在应用实例中有些语句还是使用旧版(20020625 版)的 API 函数，而不是新版本建议使用的新语句，这一点请读者注意，但是这些用法并不会影响应用实例在新版中的编译结果。

本书在编写过程中，得到了双龙电子公司的大力支持；得到福建武夷通信设备厂厂长柯友德以及曾兴和张德群的指导和帮助；也得到建阳市教育局领导和建阳三中王庭霞的关心；耿德根和詹卫前对本书进行了全面的审阅，提出许多建议性的意见并修改了部分章节。本书在编写的过程中，也参考了 21icBBS (http://www.21ic.com) 和 c51BBS (http://www.c51bbs.com)上部分网友的帖子，向这些无私奉献的网友致谢，并且对关心和支持本书的所有同志表示衷心的感谢。

鉴于国内目前还没有专门讲述用 C 语言开发 AVR 单片机的书籍，因此本书的出版只是为了满足初学者的需要而抛砖引玉。由于编者经验不足、水平有限，书中肯定会有不少缺点及错误，请广大读者批评指正，以便再版时能够改正。读者在学习本书过程中如有任何疑问，可以到由双龙电子公司赞助的 21icBBS 中“AVR 单片机论坛”提出，本书编者及论坛中其他热心的网友会进行解答。

沈文

2003 年 2 月于福建建阳

目　　录

第 1 章　AVR 单片机与 C 语言

1.1　用 C 语言开发单片机的优势

随着市场竞争的日趋激烈，要求电子工程师能够在短时间内编写出执行效率高而又可靠的嵌入式系统的执行代码。同时，由于实际系统的日趋复杂，要求所写的代码规范化、模块化，便于多个工程师以软件工程的形式进行协同开发。汇编语言作为传统嵌入式系统的编程语言，具有执行效率高等优点，但其本身是一种低级语言，编程效率低下，且可移植性和可读性差，维护极不方便，从而导致整个系统的可靠性也较差。而 C 语言以其结构化和能产生高效代码等优势满足了电子工程师的需要，成为他们进行嵌入式系统编程的首选开发工具，得到了广泛支持。用 C 语言进行嵌入式系统的开发，具有汇编语言编程所不可比拟的优势：

1．可以大幅度加快开发进度，特别是开发一些复杂的系统，程序量越大，用 C 语言就越有优势。

2．无需精通单片机指令集和具体的硬件，也能够编出符合硬件实际专业水平的程序。

3．可以实现软件的结构化编程，它使得软件的逻辑结构变得清晰、有条理，便于开发小组计划项目、分工合作。源程序的可读性和可维护性都很好，基本上可以杜绝因开发人员变化而给项目进度或后期维护以及升级所带来的影响，从而保证整个系统的可靠性。

4．省去了人工分配单片机资源(包括寄存器、RAM 等)的工作。在汇编语言中要为每一个子程序分配单片机的资源，这是一个复杂、乏味而又容易出差错的工作。在使用 C 语言后，只要在代码中申明一下变量的类型，编译器就会自动分配相关资源，根本不需要人工干预，从而有效地避免了人工分配单片机资源的差错。

5．当写好了一个算法(在 C 中称为函数)后，需要移植到不同种类的 MCU 上时，在汇编中只有重新编写代码，因而用汇编的可移植性很差。而用 C 语言开发时，符合 ANSI C 标准的程序基本不必修改，只要将一些与硬件相关的代码作适度的修改，就可以方便地移植到其他种类的单片机上，甚至可以将代码从单片机移植到 DSP 或 ARM 中。

6．C 语言提供 auto、static、flash 等存储类型，针对单片机的程序存储空间、数据存储空间及 EEPROM 空间自动为变量合理地分配空间，而且 C 语言提供复杂的数据类型(如数组、结构体、指针等类型)，极大地增强了程序处理能力和灵活性。C 编译器能够自动实现中断服务程序的现场保护和恢复，并且提供常用的标准函数库，供用户使用，使用户节省了重复编写相同代码的时间，并且 C 编译器能够自动生成一些硬件的初始化代码。

7. 对于一些复杂系统的开发，可以通过移植(或 C 编译器提供)的实时操作系统来实现，如需实现 TCP/IP 协议及 http、ftp、ppp 等功能，而使用汇编是不可能的。

虽然使用 C 语言写出来的代码会比用汇编语言占用的空间大 5%~20%，但是由于半导体技术的发展，芯片的容量和速度有了大幅度的提高，占用空间大小的差异已经不很关键，相比之下，应该更注重软件是否具有长期稳定运行的能力，注重使用先进开发工具所带来的时间和成本的优势。

因此编者推荐使用 C 语言开发嵌入式系统。

1.2 AVR 单片机的特点

ATMEL 在设计 AVR 系列单片机时吸取 PIC 及 8051 单片机的优点，并作了重大改进，如下所列：

1．AVR 程序存储器由可擦写 1000 次的 Flash 构成，并具有较大容量可擦写 100000 次的 EEPROM，因此可低价快速完成产品商品化，且可多次更改程序(产品升级)而不必浪费单片机或电路板，大大提高产品质量及竞争力。

2．高速度和低功耗，具有 SLEEP(休眠)功能。每一指令执行速度可达 50ns(20MHZ)，而耗电则在 1~2.5mA 间(典型功耗，当 WDT 关闭时为 100nA)，AVR 运用 Harvard 结构概念(具有预取指令功能)，即对程序存储和数据存储带有不同的存储器和总线，当执行某一指令时，下一指令被预先从程序存储器中取出，这使得在每一个时钟周期内都可以执行一条指令。

3．高度保密(LOCK)。可多次烧写的 Flash 具有多重密码保护锁死(LOCK)功能，并且 Flash 单元深藏于芯片内部，不像 Mask ROM 那样可通过电子显微镜破解，因此目前国内还无法破解 AVR 单片机，这有利于保护设计成果，并且 AVR 可以通过 Self Programming 方式远程下载加密的更新代码。

4．工业级产品，具有 (吸入电流)10~20mA 或 40mA(单一输出)大电流，可直接驱动 SSR，内置看门狗定时器(WDT)，防止程序走飞，提高产品的抗干扰能力。

5．超功能精简指令。具有 32 个通用工作寄存器(相当于 8051 中的 32 个累加器，克服了单一累加器数据处理造成的瓶颈现象)及 128-4KB SRAM；部分型号的单片机内置硬件乘法器，一条乘法指令只需 2 个时钟周期。

6．程序写入器件可以并行写入(用万用编程器)，也可串行在线下载(ISP)擦写，也就是说不必将 IC 拆下拿到万用编程器上烧写，而可直接在电路板上进行程序修改烧写等操作，方便产品升级，尤其是 SMD 封装，更利于产品微型化。

7. 并行 I/O 口的输入输出特性与 PIC 的 HI/LOW 输出及三态高阻抗 HI-Z 输入相似外，也可设定类似 8051 系列内部上拉电阻作输入端的功能，以满足各种应用特性所需，AVR

是真正的 I/O 口，能正确反映 I/O 口的输入输出真实情况。

8．单片机内置模拟比较器，可组成廉价且具有较高精度的 A/D 转换器。

9．像 8051 一样，不同中断向量的入口地址不同，可快速响应，而不会像 PIC 一样，所有中断都在同一向量地址，需要以程序判别后才可响应。

10．同 PIC 一样可重设启动复位。AVR 系列单片机也有内置的 POR(上电复位)和 BOD(电源电压监测)，只要在复位端接一个上拉电阻就可以了，不必使用外部复位 IC。

特别说明：对于早期生产的 AT90 芯片，大多只有 POR 而没有 BOD，使用这类器件时强烈建议同时使用外部复位 IC。近几年生产的 Mega 系列(如 ATMega8 等)都有 BOD，但一定要使能 BOD 编程熔丝位。

11．最大具有 6 种睡眠模式。

12．AT90S1200 等部分 AVR 器件具有内部 RC 振荡器，使该类单片机无需外加元器件即可工作。

13．计数器/定时器：有 8 位和 16 位，可作比较器、计数器外部中断和 PWM(也可作 D/A)用于控制输出。

14．有串行异步通信 UART(不占用定时器)和 SPI 传输功能，因其高速，故晶振可以工作在一般标准整数频率，而波特率可达 115.2Kbps 和 576Kbps。

15．AT90S4414 和 AT90S8515 的引脚排列及功能与 8051 相似，可替代 8051 系列单片机(8751/8752)的应用系统，当然还在硬件、软件上带来很多优点(WDT 看门狗、模拟比较器作 A/D、PWM 作 D/A 等)。

16．工作电压范围宽 1.8~6.0V，电源抗干扰性能较强。

17．部分 AVR 单片机具有可扩展外部存储器和多通道 10 位 A/D 及实时时钟 RTC 等功能，特别是 ATmega128 单片机更有 128KB Flash、4KB EEPROM、4KB RAM、多达 48 个 I/O 端口、34 个不同的中源断，以及 ISP 下载及 JTAG 仿真等功能。

18．进入门槛低，可以通过自制下载线(最简单的并行下载线仅需 4 个电阻)，利用 ATMEL 提供的汇编和仿真软件即可以进行开发。

19．从高级语言 C 代码看各种单片机性能比较：

```
/*一个小C函数*/
/* Return the maximum value of a table of 16 integers */
int max(int *array)
{
char a;
int maximum=-32768;
for (a=0;a<16;a++)
if (array[a]>maximum)
maximum=array[a];
return (maximum);
}
```

各种单片机的性能比较如表 1.1 所示。

表 1.1 各种单片机性能比较

单片机种类	代码大小 (Bytes)	执行周期 (cycles)	函数执行时间 (μs)	消耗电流 (mA)	执行 /S/Mw
AT90S8515	46(1)	335	42(1)	11(1)	434(1)
80C51	112(2.4)	9384	391(9)	16(1.5)	32(1.07)
68HC11	57(1.2)	5244	437(10)	27(2.5)	17(0.04)
PIC16C74	87(1.9)	2492	125(3)	13.5(1.2)	119(0.27)

由上得结论：

1．8MHz AVR 等于 224MHz80C51(用 C 语言)；

2．68HC11：代码效率高，但处理能力只有 AVR 的 1/10，功耗却高 2.5 倍；

3．PIC 速度也比较快，但是在相同功耗下，AVR 性能比其高 3.5 倍。

注：以上比较可能不够全面，比较结果仅供参考。

1.3 从 Keil C51 向 ICCAVR 快速过渡

1.3.1 AVR 和 MCS-51 存储器配置的对比

说明：在对比中，C 语言均指 ICCAVR 编译器所使用的 C 语言。

1.3.1.1 存储器布置

MCS-51 的存储器从使用角度看，分三个地址空间，三个空间分别用 MOV、MOVX 和 MOVC 指令访问，而 AVR 的存储器在物理结构上可分为五个部分，以 AT90S8515 为例来说明：

1．程序空间(000H~FFFH)，访问时用 LPM 指令访问。

2．片内数据存储器(0060H~025FH)，访问时用 STS、LDS 和 ST、LD 指令访问。

3．片外数据存储器(0260H~FFFFH)，访问时用 STS、LDS 和 ST、LD 指令访问。

4．32 个通用寄存器 R0~R31，它们之间数据传送可使用 MOV 指令。

5．I/O 寄存器(00H~3FH)，使用 IN、OUT 指令访问。

有一部分数据存储器的地址(0000H~005FH)被映射为通用寄存器(R0~R31)和 I/O 寄存

器的数据空间地址，具体为 32 个通用寄存器直接映射到数据存储器的 0000H~001FH；64 个 I/O 寄存器直接映射到数据存储器空间的 0020H~005FH。这种映射关系大大增强了 AVR 指令的灵活性，对寄存器可以像 SRAM 一样地访问，也可以使用 X、Y 和 Z 寄存器组作为索引，从而大大提高了访问寄存器的灵活性。

1.3.1.2　堆栈工作方式

MCS-51 的堆栈是一个由堆栈指针寄存器 SP(单字节)控制的向上生长型堆栈，即将数据压入堆栈时 SP 增大。AVR 系列单片机的堆栈同样受 SP 寄存器控制，而堆栈的生长方向与 MCS-51 相反，为向下生长型，即数据压入堆栈时 SP 减小。另外要注意以下几点：

1．MCS-51 的堆栈空间只能放置在片内的 SRAM 中，而 AVR 的堆栈空间既可以放置在片内 SRAM 中，也可以放置在片外 SRAM 中。

2．AVR 单片机中，对不支持外部 SRAM 的单片机，SP 寄存器为一个字节宽度，对支持外部 SRAM 的单片机为两个字节宽度(SPL、SPH)。

3．为了提高速度，一般在初始化 SP 时将其定位于内部 SRAM 的顶部，对 8515 而言为 025FH。

4．AT90S1200 不支持软件堆栈(即由 SP 控制堆栈)，只包含了一个三级深度的硬件堆栈。

5．在对 AVR 编程时一定要对 SP 进行初始化，否则很可能出现：在 AVR Studio 中模拟调试正常，而程序下载到芯片后却不工作的现象。

1.3.1.3　外部 SRAM 的配置

在 MCS-51 中，外部 SRAM 是使用专用的“MOVX”指令访问的，而在 AVR 中访问片内或片外 SRAM 均使用相同的指令，当访问数据空间的地址超过片内 SRAM 范围时会自动选择片外的 SRAM 空间。但为了正常工作还必须对寄存器 MCUCR 的 SRE(D7)、SRW(D6)位进行设置。MCUCR 寄存器如表 1.2 所示。

表 1.2　MCUCR 寄存器

MCUCR	SRE	SRW	SE	SM	ISC11	ISC10	ISC01	ISC00

当 SRE=1 时，使能外部 SRAM；SRE=0 时，禁止外部 SRAM。当 SRW=1 时在访问外部 SRAM 中插入一个等待周期，当 SRW=0 时在访问外部 SRAM 中不插入等待周期。在 C 语言中可以直接用 MCUCR|=0xC0 或 MCUCR&=0x3F 来配置外部 SRAM。

1.3.1.4　程序空间的访问

MCS-51 的程序存储器是以字节为单位的，地址是按字节进行寻址的，使用 MOVC 指令访问程序存储器，这和指令寄存器访问程序 ROM 没有什么区别。在 AVR 中程序存储器的总线为 16 位，即指令寄存器访问程序 ROM 时是以字(双字节)为单位的，即一个程序地

址对应两个字节，而 AVR 的数据存储器的总线为 8 位，当用户使用 LPM 指令访问程序 ROM 时是以字节为单位读取的，此时 Z 寄存器中的一个地址只对应一个字节，因此要注意这两个地址的换算，否则很容易产生错误，具体的换算是 LPM 指令使用的 Z 寄存器中地址应该是程序地址的两倍。

如：

```
                                ;初始化 z 寄存器
ldi  ZH,high(F_TABLE*2)         ;Z 寄存器高位存放程序存储器地址的高位
ldi  ZL,low(F_TABLE*2)          ;Z 寄存器低位存放程序存储器地址的低位
                                ;用户程序
lpm                             ;将 Z 寄存器的低位数送 R0
st   Y+,   R0                   ;Y 变址，将 R0 数送 SRAM 后，Y=Y+1
;用户程序
F_TABLE:
.db 0,1,2,3,4,5                 ;数据表的起点
```

1.3.2 AVR 输入/输出端口的使用

MCS-51 单片机的 I/O 端口大部分是准双向口，在复位时全部输出高电平，对端口的输入和输出操作也是直接通过 I/O 端口的地址进行的。而 AVR 的 I/O 端口为标准双向口，在复位时所有端口处于没有上拉电阻的输入状态(高阻态，管脚电平完全由外部电路决定)，这在强调复位状态的场合是很有用的。AVR 的每一个端口对应三个地址，即 DDRX、PORTX 和 PINX(X 针对不同的单片机可从 A~F 中分别取不同的符号，注意只有 PINX 可取 F)。各端口功能配置如表 1.3 所示。

表 1.3 端口功能配置表

DDRXn	PORTXn	I/O	上 拉	备 注
0	0	输入	关闭	三态(高阻)
0	1	输入	打开	提供弱上拉，低电平必须由外电路拉低，PXn 脚输出电流
1	0	输出	关闭	推挽 0 输出
1	1	输出	关闭	推挽 1 输出

DDRX 为端口方向寄存器，当 DDRX 的某一位置 1 时，相应端口的引脚作为输出使用；当 DDRX 的某一位清 0 时，对应端口的引脚作为输入使用。PORTX 为端口数据寄存器，当引脚作为输出使用时，PORTX 的数据由相应引脚输出；当引脚作为输入使用时，POTRX 的数据决定相应端口的引脚是否打开上拉功能。PINX 为相应端口的输入引脚地址，如果希望读取相应引脚的逻辑电平值，一定要读取 PINX 而不能读取 PORTX，PORTX 为端口锁

存器的值，这与 MCS-51 是有区别的。

注意：在使用 AVR 单片机之前，一定要根据引脚功能定义对相应的端口初始化，否则，端口很可能在用作输出时不能正常工作(复位后默认为输入状态)。如设置端口 B 的高四位为输出，低四位为输入，则有：

汇编语言

```
ldi R16, $F0
out DDRB, R16
```

在 C 语言中

```
DDRB=0xF0;
```

1.3.3　AVR 和 MCS-51 定时器的对比

1.3.3.1　功能比较

在 MCS-51 中，定时/计数器有两种基本用法，即以晶振频率的十二分频信号为输入的定时器工作方式，或以外部引脚 INT0、INT1 上输入信号的计数器工作方式。在 AT90S8515 中有两个定时器 T0 和 T1，T0 的功能与 MCS-51 相似；而 T1 的功能更强，除了普通的定时/计数功能外，还有一些增强的功能，如：比较匹配 A、比较匹配 B、由 ICP 引脚或模拟比较器触发的捕捉功能、8~10 位的 PWM 调制器。AVR 的定时/计数器用作定时器时，其输入信号可选为晶振频率的某一个分频信号，分频比为 1、8、64、256、1024 五种，作为计数器使用时，既可上升沿触发也可下降沿触发。

1.3.3.2　T0 的使用

在 AT90S8515 中，T0 为八位长度，其由 TCCR0 寄存器控制。TCCR0 组成如表 1.4 所示，CS02、CS01、CS00 定义定时器 0 的预定比例源，见表 1.5 所示。

表 1.4　TCCR0 寄存器

TCCR0	×	×	×	×	×	CS02	CS01	CS00

表 1.5　时钟源预定比例选择

CS02	CS01	CS00	说　明	CS02	CS01	CS00	说　明
0	0	0	T0 停止工作	1	0	0	CK/256，定时器
0	0	1	CK，定时器	1	0	1	CK/1024，定时器
0	1	0	CK/8，定时器	1	1	0	计数器，下降沿触发
0	1	1	CK/64，定时器	1	1	1	计数器，上升沿触发

例：定时器 T0 用作定时器，晶振频率 4MHz，定时时间 10ms，可以这样对 T0 初始化。

汇编语言

```
LDI R16,$D9
OUT TCNT0,R16                ;定时常数到定时寄存器 TCNT0
LDI R16,$05
OUT TCRR0,R16                ;1024 分频比
```

C 语言

```
TCNT0=0xD9;                  //TCNT0 为定时计数器寄存器
TCRR0=0x05;
```

1.3.3.3 T1 的使用

在 AVR 中 T1 是 16 位的，其控制寄存器有 TCCR1A 和 TCCR1B 两个，TCCR1A 组成如表 1.6 所示。

表 1.6 TCCR1A 寄存器

TCCR1A	COM1A1	COM1B1	COM1B1	COM1B0	×	×	PWM11	PWM10

各功能如表 1.7 所示。

表 1.7 比较 1 模式和 PWM 模式选择

COM1X1	COM1X0	说 明	PWM11	PWM10	说 明
0	0	定时器/计数器 1 与输出脚 OC1X 不连接	0	0	禁止 PWM 操作
0	1	OC1X 电平翻转	0	1	8 位 PWM
1	0	OC1X 为低电平	1	0	9 位 PWM
1	1	OC1X 为高电平	1	1	10 位 PWM

TCCR1B 的组成如表 1.8 所示，其中 CS10~CS12 的用法同 T0。

表 1.8 TCCR1B 寄存器

TCCR1B	ICNC1	ICES1	×	×	CTC1	CS12	CS11	CS10

ICNC1：置 1 时使能输入捕捉噪声消除，清 0 时禁止输入捕捉噪声消除。

ICES1：置 1 时在 ICP 的上升沿发生定时器捕捉，清 0 时在 ICP 的下降沿发生定时器捕捉。

CTC1：置 1 时，当比较匹配 A 发生时将 TCNT1 清 0，清 0 时 TCNT1 继续计数，直至被停止、清除、溢出为止。注意：只有比较匹配 A 有效，另外在 PWM 方式下该位无效。

例：定时器 1 用比较匹配 A 方式，在 OC1A 引脚输出一个 50Hz 的方波，晶振频率为 4MHz，可以这样对 T1 初始化。

汇编语言

```
LDI R16,$20
OUT DDRD, R16              ;引脚 OC1A 输出
LDI R16,$00
OUT TCNT1H,R16
OUT TCNT1L, R16            ;TCNT1 清 0
LDI R16,$02
OUT OCR1AH,R16
LDI R16, $71
OUT OCR1AL,R16             ;送入对应 10ms 的比较常数
LDI R16, $40
OUT TCCR1A, R16            ;T1 和 OC1A 相连比较匹配时 OC1A 翻转
LDI R16, $0B
OUT TCCR1B,R16             ;T1 以定时器方式工作分频比为 64
                           ;比较匹配后自动清除 TCNT1
```

C 语言

```
DDRD=0x20;                 //引脚 OC1A 输出
TCNT1=0x00;                //TCNT1 清 0
OCR1A=0x271;               //送入对应 10ms 的比较常数
TCCR1A=0x40;               //T1 和 OC1A 相连比较匹配时 OC1A 翻转
TCCR1B=0x0B;               //T1 以定时器方式工作，分频比为 64，
                           //比较匹配后自动清除 TCNT1
```

注意：

1. 由于 T1 的 TCNT1、OCR1A、OCR1B 和 ICR1 均为 16 位的定时器，为了正确地写入和读出，在写入数据时应先写入高位字节，后写入低位字节，在读取数据时应先读取低位字节，后读取高位字节。

2. T1 的捕捉方式，可用于 ICP 引脚上频率或周期的测量，在使用时只需使能捕捉中断即可，对 T1 的设置可参考定时的用法。

1.3.4　AVR 和 MCS-51 中断系统的对比

MCS-51 有六个中断源(5 个中断入口地址)，分两个优先级，并且是通过 IE 寄存器控制中断的使能，通过 IP 控制中断的优先等级。而在 AVR 中根据不同的单片机有不同数量的中断源，典型的 AT90S8515 有 12 个中断源，这 12 个中断源各有自己的中断向量入口地址，

如表 1.9 所示。

表 1.9 复位和中断向量

向 量 号	程 序 地 址	中 断 源	中 断 定 义
1	$000	RESET	硬件引脚和看门狗复位
2	$001	INT0	外部中断请求 0
3	$002	INT1	外部中断请求 1
4	$003	TIMER1 CAPT	定时器/计数器 1 捕获事件
5	$004	TIMER1 COMPA	定时器/计数器 1 比较匹配 A
6	$005	TIMER1 COMPB	定时器/计数器 1 比较匹配 B
7	$006	TIMER1 OVF	定时器/计数器 1 溢出
8	$007	TIME0 OVF	定时器/计数器 0 溢出
9	$008	SPI，STC	串行传送完成
10	$009	UART，RX	UART，RX 完成
11	$00A	UATR，UDRE	UATR 数据寄存器空
12	$00B	UART，TX	UART，TX 完成
13	$00C	ANA_COMP	模拟比较器

AVR 通过寄存器 GIMSK、TIMSK 和 SREG 来控制中断使能，其中 SREG 的 D7 位 I 是全局中断使能标志，在 AVR 中只有全局中断控制位和某一特定中断控制位同时使能，中断才会起作用。GIMSK 和 TIMSK 的功能见表 1.10 所示。

表 1.10 GIMSK 和 TIMSK 寄存器

GIMSK	INT1	INT0	×	×	×	×	×	×
TIMSK	TOIE1	OCIE1A	OCIE1B	×	TICIE1	×	TOIE0	×

INT0、INT1 外部中断请求使能，置 1 时开放中断，清 0 时禁止中断。

外部中断的触发方式由 MCUCR 的 D0~D3 位控制，如表 1.11 所示。

TOIE0、TOIE1：定时器 0、定时器 1 的溢出中断使能。

OCIE1A、B：定时器 1 的比较匹配 A、B 中断使能。

TICIE1：定时器 1 的输入捕捉中断使能。

表 1.11 外部中断触发方式

ISCX1	ISCX0	说 明	ISCX1	ISCX0	说 明
0	0	低电平触发中断	1	0	下降沿触发中断
0	1	高电平触发中断	1	1	上升沿触发中断

在 AVR 中没有专门的中断优先级控制寄存器来区分中断的优先等级，用户可在中断服务程序中通过使能全局中断 I 来使系统响应高优先级的中断。具体的做法是当 AVR 单片机响应任何一个中断时，硬件会禁止全局中断 I，从而禁止系统响应其他中断，而当从中断服

务程序中退出时，硬件重新使能全局中断 I。而当在中断服务程序中用 SEI 指令打开全局中断使能时，系统在没有退出中断服务程序的情况下又恢复了对中断的响应能力，从而可以响应高优先级的中断。另外，在同一优先级中入口地址较低的中断优先级较高。

例：系统使能定时器 1 溢出中断和外部 INT0 中断，其中 INT0 的优先级较高，此时可以这样对 MCU 初始化。

汇编语言

```
LDI R16,$40
OUT GIMSK, R16              ;使能 INT0 中断
LDI R16,$80
OUT TIMSK,R16               ;使能 T1 溢出中断
SEI                         ;使能全局中断
timer1_ovf:                 ;T1 溢出中断服务程序
SEI                         ;在 T1 溢出中断服务程序中开放全局中断,
                            ;保证 INT0 的优先级
```

注意：在 AVR 的子程序中硬件不保护 SREG 状态寄存器，应根据实际情况由软件进行保护。

C 语言

```
#pragma interrupt_handler timer1:7          //声明 timer1()为中断处理函数
#pragma interrupt_handler int0:2            //声明_int0()为中断处理函数
void main (void)
    {
    GIMSK=0x40                              //使能 INT0 中断
    TIMSK=0x80                              //使能 T1 溢出中断
    _SEI()                                  //使能全局中断
    }
void timer1(void)                           //T1 溢出中断服务程序
    {
_SEI()                                      //在 T1 溢出中断服务程序中开放全局中断
                                            //保证 INT0 的优先级

    }
```

在 C 语言的中断服务程序(中断处理函数)中，会自动保护中断服务程序使用过的所有寄存器。

1.3.5　AVR 和 MCS-51 位操作的对比

MCS-51 和 AVR 有较强的位操作功能，在汇编语言写的 AVR 源程序中对端口的某一位置 1 可用 SBI 指令，清 0 可用 CBI 指令。在 C 语言程序中，可用位运算或在线汇编完成

上述功能，如置 PORTB 的 D2 位为 1，清 PORTB 的 D6 位为 0：

```
PORTB|=(1<<2);                    //D2 位置 1
PORTB&=~(1<<6)                    //D6 位清 0
```

或用在线汇编：

```
ASM("SBI 0x18,2" )                //D2 位置 1
ASM("CBI 0x18,6" )                //D6 位清 0
```

1.3.6 AVR 单片机内置 EEPROM 的使用

AVR 是通过三个寄存器来访问 MCU 内置的 EEPROM 的，一个寄存器是 EEAR，存放访问 EEPROM 的地址，根据片内 EEPROM 的多少可能有不同的长度；另一个是 8 位的 EEDR，用于存放访问 EEPROM 的数据；第三个是 EECR，用于控制对 EEPROM 的读写，EECR 的结构如表 1.12 所示。

表 1.12 EECR 寄存器

EECR	×	×	×	×		EEMWE	EEWE	EERE

EEMWE：EEPROM 主写使能。只有在其置 1 后的 4 个时钟周期内将 EEWE 置 1，才能完成 EEPROM 写入，否则写操作无效。EEMWE 置 1 后，在 4 个周期后由硬件自动清除。

EEWE：EEPROM 写入使能。

EERE：EEPROM 读取使能。

例：写数据到片内 EEPROM 中子程序。

汇编语言：

```
.def EEdwr=r16                    ;写入 EEPROM 的数据
.def EEawr=r17                    ;EEPROM 的地址低位
.def EEawrh=r18                   ;EEPROM 的地址高位
EEWrite:
sbic EECR,   EEWE
rjmpEEWrite                       ;等待 EEPROM 就绪
out EEARH, Eeawrh
out EEARL, Eeawr                  ;送入 EEPROM 地址
out EEDR, EEdwr                   ;送入写入 EEPROM 的地址
sbi EECR, EEMWE                   ;设置 EEPROM 主写使能
sbi EECR, EEWE                    ;设置 EEPROM 写使能
ret
```

C 语言：(注意，应包含头文件 eeprom.h)

```
int EEPROMwrite(int location,unsigned char);
int location: 片内 EEPROM 的地址
unsigned char: 写入 EEPROM 的数据
```

1.3.7　AVR 单片机内置看门狗电路(Watchdog)的使用

AVR 系列单片机内置看门狗电路，由寄存器 WDTCR 控制。WDTCR 的结构如表 1.13 所示。

表 1.13　WDTCR 寄存器

WDTCR	×	×	×	WDTOE	WDE	WDP2	WDP1	WDP0

WDTOE：看门狗关闭使能，只有在该位被置 1 后的 4 个时钟周期内将 WDE 清 0，才能关闭看门狗电路，否则看门狗电路不会被关闭。WDTOE 在被置 1 后，在 4 个周期后由硬件自动清 0。

WDE：置 1 时，使能看门狗电路，清 0 时关闭看门狗电路。注意，关闭看门狗电路应在对 WDTOE 置 1 后 4 个时钟周期内进行。

WDP0~WDP2：看门狗电路的分频系数(产生复位所需要的振荡周期数)，其影响看门狗电路复位的时间如表 1.14 所示。

表 1.14　看门狗电路的分频系数

WDP2	WDP1	WDP0	分 频 系 数	DC3V 时 产生复位所需时间/s	DC5V 时 产生复位所需时间/s
0	0	0	16K	0.047	0.015
0	0	1	32K	0.094	0.030
0	1	0	64K	0.19	0.060
0	1	1	128K	0.38	0.12
1	0	0	256K	0.75	0.24
1	0	1	512K	1.5	0.49
1	1	0	1024K	3.0	0.97
1	1	1	2048K	6.0	1.9

注意，看门狗电路的振荡器为内部 RC 振荡器，其振荡频率受电压影响，在 DC5V 时约为 1MHz。在 AVR 汇编中有一条指令 WDR 来清除看门狗定时器，在 C 语言中对应为 _WDR( )函数或 WDR( )函数。

1.3.8 AVR 和 MCS-51 中串口通信 UART 功能的对比

在 MCS-51 中，串口通信的波特率发生需要使用一个定时器，而且支持的波特率也较低。AVR 单片机可以有较高的波特率，最高波特率可达 115200bps，而且有专用的波特率发生器(注意：AT90S1200 没有 UART，只能用软件模拟串口通信)。在 AVR 中用于 UART 的寄存器主要有以下几个：接收和发送数据寄存器 UDR、状态寄存器 USR、控制寄存器 UCR 和波特率寄存器 UBRR。

UDR 寄存器由两个物理上分开的寄存器共享同一个地址，写入数据时是写到发送寄存器，读出数据时是读取接收寄存器。USR 如表 1.15 所示，此表反映了 UART 的状态。

表 1.15 UART 寄存器

UART	RXC	TXC	UDR	FE	OR	×	×	×

RXC：UART 接收完成。

TXC：UART 发送完成。

UDR：UART 数据寄存器空标志。

FE： 帧出错。

OR： 超越出错。

UCR 控制 UART 的工作，其组成如表 1.16 所示。

表 1.16 UCR 寄存器

UCR	RXCIE	TXCIE	UDRIE	RXEN	TXEN	CHR9	RXB8	TXB8

RXCIE：接收中断使能。

TXCIE：发送中断使能。

UDRIE UART：数据寄存器空中断使能。

RXEN：接收使能。

TXEN：发送使能。

CHR9：发送 9 位字符。

RXB8：接收到的第 8 位字符。

TXB8：发送的第 8 位字符。

UBRR 控制 UART 的波特率，其与波特率的计算公式为：

BAUD=FCK/[16(UBRR+1)]

其中 BAUD 表示波特率，FCK 表示晶振频率，UBRR 表示 UBRR 寄存器中的常数。注意，波特率的计算值与标准波特率相差不能超过 2%，否则会影响串行通信。

例 1：在 8MHz 晶振下以 19200bps 和 PC 机通信，可对 AVR 进行如下初始化。

汇编语言：

```
LDI R16,25
OUT UBRR,R16
LDI R16,$18
OUT UCR,R16
```

C 语言：

```
UBRR=25;
UCR=0x18;
```

例 2：在 8MHz 晶振下以 19200bps 与 PC 机通信，每接收到一个非 0 字节，发送一个“OK”。

```
#include "io8515.h"
#include "stdio.h"
void main(void)
    {
    unsigned char temp;
    UBRR=25;
    UCR=0x18;
    puts("Hello World!");
    putchar(0x0d);
    putchar(0x0a);
    while (1)
        {
        temp=getchar();
        if (temp!=0)
             {
             puts("OK!");
             putchar(0x0d);
             putchar(0x0a);
             temp=0;
             }
        }
    }
```

1.3.9 C51 的源代码向 ICCAVR 的快速转换

熟悉 Keil C51 的读者看了以上内容，完全可以很快地写出 AVR 的 C 源程序，下面将 C51 向 ICCAVR 的转换进行一次总结。

1．头文件。对 C51 中定义寄存器的头文件，如 reg51.h、at89x51.h 等，替换成相应的 AVR 头文件，如 io8515v.h、iom128v.h 等。

2．中断处理函数。在 C51 中以 interrupt 关键字来说明某一个函数为中断处理函数，在 ICCAVR 中可采用#pragma interrupt_handler 预处理命令在子程序前声明，具体用法如下：

#pragma interrupt_handler <中断处理函数名>:<中断向量号>

注意，原 C51 源程序中的 interrupt 和 using 关键字应当删除。

3．对 C51 中的 bit 和 sbit 数据类型的处理。在 ICCAVR 中不支持 bit 和 sbit 数据类型，对这两种类型可用 unsigned char 来代替；对有关位运算可用标准 C 的位运算功能进行处理，也可采取在线汇编处理。

4．对中断系统、定时器初始化。需重新根据相应控制寄存器的功能给其赋值，方法与 C51 相同。例如：对 MCS51 中 TMOD、TCON 的处理改为对 AVR 的 TCCR0、TCCR1A、TCCR1B、TIFR 的处理；对 MCS51 中 IE、IP 的处理改为对 AVR 中 GIMSK、TIMSK、MCUCR、SREG 的处理。

5．将原 C51 中有关对看门狗电路、外部 EEPROM 的处理改为对 AVR 芯片内部看门狗电路、内部 EEPROM 的处理。

6．对 MCS-51 的 UART 初始化改为对 UCR 和 UBRR 初始化。

7．如果使用片外 SRAM，应当对 MCUCR 初始化；如果有引脚作为输出引脚使用，应当对其方向寄存器进行初始化。

8．对 C51 中符合 ANSI 标准的 C 语言，一般可以不作修改。

第 2 章　ICCAVR 使用的 C 语言基础知识

C 语言功能丰富、表达能力强、使用灵活方便、应用面广、目标程序效率高、可移植性好，既有高级语言的优点，又具有低级语言的许多特点，非常适合于开发嵌入式操作系统。ICCAVR 是针对 AVR 单片机设计的 C 语言编译器，支持符合 ANSI 标准的 C 语言程序设计，同时针对 AVR 单片机的一些特点进行扩展。下面结合 ICCAVR 编译器介绍 C 语言的一些基础知识。

2.1　标识符、关键字和数据类型

2.1.1　标识符

C 语言的标识符是用来标识源程序中某个对象的名字，这些对象可以是函数、变量、常量、数组、数据类型、存储方式、语句等。合法的标识符由字母、数字和下划线组成，并且第一个字符必须为字母或下划线。C 语言的标识符区分大小写字母，这与我们常用的 Windows 操作系统是不同的，因此“i”与“I”在 C 语言中代表不同的标识符。

ICCAVR 中能够识别的标识符为 30 位英文字符。在使用标识符时应注意：标识符不能用中文，长度允许超过 30 位，但超过 30 位以后的标识符不能被识别，即两个标识符前 30 位相同，第 31 位不同，ICCAVR 认为这是同一个标识符。

C 语言的标识符可以分为三类。

2.1.1.1　关键字

关键字是一类具有固定名称和特定含义的特殊标识符，有时又称为保留字。在编写 C 语言源程序时，对标识符的命名不能与关键字相同。表 2.1 列出 ANSI C 的 32 个关键字。

表 2.1　ANSI C 标准的关键字

关　键　字	用　　途	说　　明
auto	存储种类声明	用以声明局部变量，默认值为此
break	程序语句	退出最内层循环体
case	程序语句	switch 语句中的选择项

续表

关 键 字	用 途	说 明
char	数据类型声明	单字节整型数或字符型数据
const	存储类型声明	在程序执行过程中不可修改的变量值
continue	程序语句	退出本次循环，转向下一次循环
defaut	程序语句	switch 语句中的失败选择项
do	程序语句	构成 do...while 循环结构
double	数据类型声明	双精度浮点数
else	程序语句	构成 if...else 选择结构
enum	数据类型声明	枚举
extern	存储类型声明	在其他程序模块中声明了的全局变量
float	数据类型声明	单精度浮点数
for	程序语句	构成 for 循环结构
goto	程序语句	构成 goto 循环结构
if	程序语句	构成 if...else 选择结构
int	类据类型声明	基本整型数
long	数据类型声明	长整型数
register	存储类型声明	使用 CPU 内部的寄存器的变量
return	程序语句	函数返回
short	数据类型声明	短整型数
signed	数据类型声明	有符号数，二进制的最高位为符号位
sizeof	运算符	计算表达式或数据类型的字节数
static	存储类型声明	静态变量
struct	数据类型声明	结构类型数据
switch	程序语句	构成 switch 选择结构
typedef	数据类型声明	重新进行数据类型定义
union	数据类型声明	联合类型数据
unsigned	数据类型声明	无符号数据
void	数据类型声明	无类型数据
volatile	数据类型声明	声明该变量在程序执行中可被隐含改变
while	程序语句	构成 while 和 do...while 循环语句

ICCAVR 编译器除了支持 ANSI C 标准的关键字以外，还根据 AVR 单片机的自身特点扩展了如表 2.2 所示的关键字及汇编伪指令。

表 2.2 ICCAVR 扩展的关键字及汇编伪指令

关 键 字	用 途	说 明
pragma	编译附注声明	编译附注
//	注释	可以使用 C++中的//类型注释
const	存储类型说明	对 ANSI C 中的 const 的功能进行了扩充

续表

关 键 字	用　途	说　明
tasks	函数类型说明	pragma ctask 合用，说明生成的函数不必保存和恢复寄存器
interrupt	函数类型说明	pragma interrupt_handler，说明函数为中断类型函数
vector	中断向量说明	在汇编语句中说明中断向量
eeprom	存储类型声明	AVR 单片机中的 EEPROM
data	存储类型声明	AVR 单片机中的 SRAM
text	存储类型声明	AVR 单片机中的 FLASH
globl	数据类型说明	定义一个全局符号
asm	程序类型说明	汇编类型
area	区域说明	汇编中说明不同的区域，伪指令
abs	代码定位方式说明	汇编中绝对定位区域，伪指令
rel	代码定位方式说明	汇编中重定位区域，伪指令
con	代码定位方式说明	汇编中连接定位，伪指令
ovr	代码定位方式说明	汇编中覆盖定位，伪指令
byte	定义常数	汇编中表示字节常数，伪指令
word	定义常数	汇编中表示字常数，伪指令
long	定义常数	汇编中表示双字常数，伪指令
blkb	定义常数	汇编中表示保留字节空间而不赋值，伪指令
blkw	定义常数	汇编中表示保留字空间而不赋值，伪指令
blkl	定义常数	汇编中表示保留双字空间而不赋值，伪指令

说明：在 ICCAVR 中，可以使用 ANSI C 语言的“/*……*/”的注释，当选择了编译扩充(Project->Options->Compiler)后，还可以使用 C++类的“//”注释。应注意这两种注释的区别，当使用“/*……*/”的注释时，在两个注释符号中可以插入多行的注释内容，而选用“//”注释时，只能插入一行的注释内容，如果需插入多行注释，必须每行注释最前面插入“//”符号。在 ICCAVR 的在线汇编中，注释只能用“;”号和“//”符号来引导。

2.1.1.2　预定义标识符

这些标识符在 C 语言中都有特定的含义，如 C 语言提供的库函数的名字(如 printf)和预编译处理命令(如 define)等，C 语言语法允许用户把这类标识符另作他用，但这将使这些标识符失去系统规定的原意。因此为了避免误解，建议读者不要把这些预定义标识符另作他用。

2.1.1.3　用户标识符

由用户根据需要定义的标识符称为用户标识符。程序中使用的用户标识符除了要遵循标识符的命名规则以外，一般不要用代数符号(如 a、b、x1、y1)作为变量名，应选取具有相关含义的英文单词(或缩写)或汉语拼音作为标识符，以增加程序的可读性，如：count、number1、red、work 等，这是结构化程序的一个特征。本书在一些简单的例子中，为了简洁，仍用单字符的变量名(如 a、b、c)，但请读者注意尽量不要在其他程序中这样使用。

虽然 ICCAVR 允许用户自定义的标识符的第一位使用下划线“_”，但是在 ICCAVR 编译系统所使用的标识符也常在变量和函数名前加下划线(如在线汇编中)。如果用户自定标识也用下划线作前导，容易与系统中所使用的标识符混淆，建议用户自定义的标识符以英文字母开头。

2.1.2 数据类型

C 语言提供的数据结构是以数据类型的形式出现的，共有以下几种数据类型：

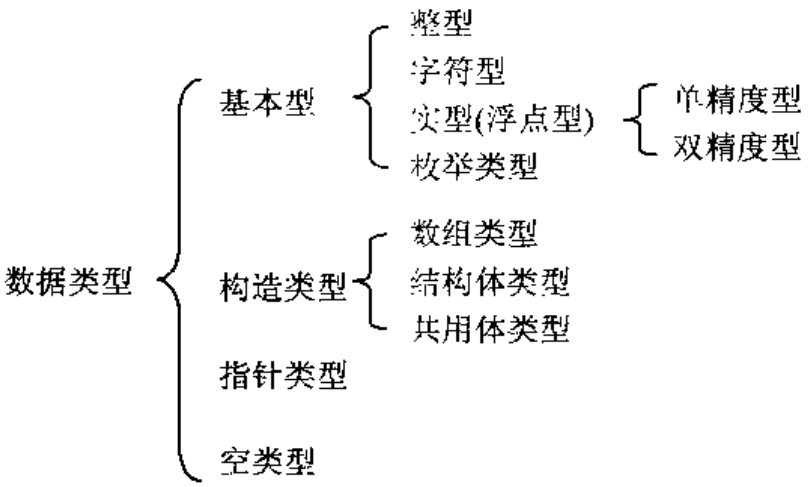


表 2.3 为 ICCAVR 中对基本型的数据类型分配的存储空间。

表 2.3 ICCAVR 的数据类型

类　型	长度(字节)	范　围
无符号字符型(unsigned char)	1	0~255
有符号字符型(signed char)	1	-128~127
字符型(char (*))	1	0~255
无符号短整型(unsigned short)	2	0-65535
(有符号)短整型((signed) short)	2	-32768~32767
无符号整型(unsigned int)	2	0~65535
(有符号)整型((signed) int)	2	-32768~32767
无符号长整型(unsigned long)	4	0~4294967295
(有符号)长整型((signed) long)	4	-2147483648~2147483647
(单精度)浮点型(float)	4	+/-1.175e-38~3.40e+38
双精度(浮点型)(double)	4	+/-1.175e-38~3.40e+38

注意：

1. 在 ICCAVR 中，“char”默认为“unsigned char”，这与“short”、“int”和“long int”的默认为“signed”类型是不同的，而且与 ANSI C 和大部分的编译器(如 keil、CVAVR 等)中“char”默认为“signed char”也是不同的。在 IAR 中虽然也默认为“unsigned char”，但可以使用“--char_is_signed”参数将“char”类型默认改为有符号类型(signed char)，而在 ICCAVR 中则不能。

2. 在 GCCAVR 和 IAR 中，均可以指定 64 位的双精度浮点型数据，而 ICCAVR、Keil 和 CVAVR 则不能。

2.1.2.1　常量与变量

在程序运行过程中，值不能改变的量称为常量，常量又有整型常量、实型常量、字符型常量和字符串常量之分；整型常量又分为短整型、长整型……。

在程序运行过程中，值可以改变的量称为变量，程序中用到的每一个变量都应有一个名字作为标识符，该名字也称为变量名，属于用户标识符类。变量在内存中占据一定的存储单元。在该存储单元中存放变量的值，用户对变量名操作就是对该存储单元进行操作。C 语言规定，程序中所用到的变量应该先定义后使用。与常量一样，变量也有类型的区分，如整型变量、实型变量、字符型变量等。

2.1.2.2　整型数据的分类

前面已介绍过，整型数据可分为整型常量和整型变量。通常称“int”类型为基本整型，除此以外，ANSI C 中还包括其他三种整数类型，它们是短整型(short int)、长整型(long int)、无符号型(unsigned int)，若不指定为无符号型，即默认类型为有符号型(signed)。在 ICCAVR 中，短整型(short int)和基本整型(int)的存储空间和值的范围是一致的(2 个字节)，使用时不加区分。

ICCAVR 中整型常量可以用以下四种形式表示：

1．十进制：如 0、314、 123。

2．八进制：以 0 开头的数是八进制，如 0314。

3．十六进制：以 0x 或 0X 开头的数是十六进制数，如 0x314。

4．如果选择了编译扩充(Project->Options->Compiler)，可以使用二进制，用 0b 开头，如 0b11100011。

注意：在一个整型常量后面加一个字母“L”或“l”，则认为是“long int”型常量，例如 123l、0L 等。一般情况下，整型常量可以赋值给一个长整型变量，但在一些特殊情况下，如函数的形参为“long int”型时，要求实参也必须为“long int”型，这时就不能用 123 作实参，而要用 123l。

整型变量：

整型变量可分为基本型(int)、短整型(short int)、长整型(long int)、无符号型(unsigned int)四种。不同类型的整型变量之间可以进行算术运算。

2.1.2.3　实型数据的分类

实型数据可分为实型常量和实型变量，实型常量又称为实数或浮点数。在 ICCAVR 中，实型常量用 4 个字节存储，符合 IEEE-754 标准，“float”和“double”存储空间和值的范围是一致的(4 个字节)，共有两种表示形式：

1．十进制数形式。它由数字和小数点组成(注意必须有小数点)。如 0.123、.123、123.等。

2．指数形式。字母 e(或 E)之前必须有数字，e(或 E)之后必须为整数。如 1.23e123 等。

实型变量：

实型变量分为单精度(float)和双精度(double)两类。在 ICCAVR 中，单精度和双精度的实型变量均用 4 个字节存储，使用时不加区分。

2.1.2.4 字符型数据的分类

C 语言中字符型数据可分为字符常量、字符串常量、字符变量三种，其中再分为有符号“signed”和无符号“unsigned”二种情况。注意：在 ICCAVR 中，若不指定字符型数据为有符号型，隐含的类型为无符号型“unsigned”，这与 ANSI C 及 keil C51 中不指定为无符号型则默认为有符号数是不同的。为了保证程序的可移植性，建议对于字符型数据，应直接明确标示出是有符号型还是无符号数。

字符常量是用单引号括起来的一个字符。如'a'、'1'、'?'，在字符常量中，C 语言还允许用一种特殊形式的字符常量，就是以一个“\”开头的转义字符，常用的以“\”开头的特殊字符如表 2.4 所示。

表 2.4 C 语言中常用的转义字符表

字 符 形 式	功 能
\n	换行
\t	横向跳格(即跳到下一个输出区)
\v	竖向跳格
\b	退格
\r	回车
\f	走纸换页
\\	反斜杠字符“\”
\'	单引号字符
\ddd	1 到 3 位 8 进制数所代表的字符
\xhh	1 到 2 位 16 进制数所代表的字符

说明：在表 2.4 的最后两行是用 ASCII 码来表示一个字符，如'\101'代表'A'，'\x41'也代表'A'。

字符串常量：

字符串常量是用一对双引号括起来的字符序列，如"010"、"a"、"ICCAVR"等。

字符常量与字符串常量是不同的，下例将简单说明两者的不同。

[例 2.1]

```
void main (void)
   {
   char a='a';                    //前面一个 a 代表变量名，'a'为字符常量
```

```
char *b="b";                    //*b 代表指向字符型变量的指针，"b"代表字符串常量
printf ("a=%c",a);              //输出 a
printf ("b=%s",*b);             //输出 b
}
```

'a'在内存中占用了 1 个字节的存储单元。C 语言规定：在每一个字符串的结尾加一个“\0”作为结束标志，因此字符串"b"在内存中占用 2 个字节的存储单元(第一个字节存 b 的 ASCII 码，第二个字节存“\0”)；两者的输出方式也不相同，其中字符常量用“%c”格式说明符，字符串常量使用“%s”格式说明符。

字符型变量：

字符型变量用来存放字符型常量（注意，一个字符型变量只能存放一个字符常量）。将一个字符常量放到一个字符变量中，实际上并不是把该字符本身放到内存单元中去，而是将该字符相应的 ASCII 代码放到内存单元中。例如'A'的 ASCII 代码为 65，则在内存中的值为 65，以二进制的形式存放。由于字符数据以 ASCII 码存储，存储形式与整数的存储形式类似，因此在 C 语言中，字符型数据与整型数据之间可以通用，既可以以字符形式输出，也可以以整数形式输出。

[例 2.2]

```
void main(void)
   {
   char c1,c2;                          //指定 c1、c2 为字符型变量
   c1=97;c2=98;                         //将整数 97、98 赋值给 c1 和 c2
   printf ("c1=%c  c2=%c",c1,c2);       //输出 c1 和 c2 的值
   }
```

输出结果

```
c1=a  c2=b
```

在上例中，将 97 和 98 赋值给 c1 和 c2，该行语句也相当于：

```
c1='a';c2='b';
```

2.1.2.5　各类数值型数据间的混合运算

整型、实型(包括单精度和双精度)、字符型数据间可以混合运算。在 ICCAVR 中，不同类型的数据按以下规则先转换成同一类型，然后进行运算。

char→(short) int→unsigned int→long→float(double)

如下：

a=3+'a';

系统自动将'a'转换为整数 97，再与 3 相加，得结果为 100。

2.1.2.6 算术运算和算术表达式

C 语言把除了控制语句和输入输出以外的几乎所有基本操作都作为运算符处理，有以下几类：

1. 算术运算符	+ - * / %
2. 关系运算符	> < == >= <= !=
3. 逻辑运算符	& | !
4. 位运算符	>> << ~ | ^ &
5. 赋值运算符	=及扩展赋值运算符
6. 条件运算符	? :
7. 逗号运算符	,
8. 指针运算符	* &
9. 求字节运算符	sizeof
10. 强制类型转换运算符	( )
11. 分量运算符	. →
12. 下标运算符	[ ]
13. 其他	函数调用运算符( )

要注意：

(1) 两个整数相除结果为整数(舍去小数部分)，如 2/4 的结果为 0，4/3 的结果为 1。但如果两个数中有一个为实数，则结果为 double 型，如 2/4.0 的结果为 0.5。

(2) C 语言规定在表达式求值时，先按运算符的优先级别高低次序执行，如果在一个运算符对象两侧运算符的优先级别相同，则按“结合方向”处理。C 语言规定了各种运算符的结合性，算术运算符的结合方向为“自左至右”，即先左后右，也称为“左结合性”；有些运算符的结合方向为“自右向左”，具有“右结合性”，如：!、~、++、--、-(负号)、*、&、sizeof、( )(类型转换运算符)。

(3) 可以利用强制类型转换运算符将一个表达式转换成所需类型。如：

```
(int) (x+y)                    ;将 x+y 的值转换成整型
(float) (2/4)                  ;将 2/4 的值转换成 float 型
```

要注意，表达式的值应该用括号括起来，如果写成以下的形式就错了。

```
(int) x+y                      ;将变量 x 转换成整型后再与变量 y 相加
```

如果有以下的赋值语句：

```
float x                        ;x 为 float 型
int y                          ;y 为 int 型
```

```
y=(int) x                    ;将 x 转换为 int 型后赋值给 y
```

注意，完成上述的赋值语句后，x 的值并没有发生变化，还是 float 型。

(4) 自增(++)、自减(--)运算符

自增、自减运算符只能用于变量，不能用于常量或表达式，结合方向为“自右向左”，如果有表达式-i++，由于“-”与“++”的优先级相同，而结合方向为自右向左，该表达式相当于-(i++)。

++i 和 i++在表达式中是不同的，++i 是将 i 的值加 1 后再进行运算，而 i++是将 i 的值进行运算后再加 1。例如：

```
int a,b,c,i=4;
a=i++;
b=++i;
c=i++;
```

运算结果：a=4，b=6，c=6，i=7。

2.1.2.7　赋值运算符和赋值表达式

赋值符号“=”就是赋值运算符，赋值号左边必须是代表某存储单元的变量名或代表某存储单元的表达式，赋值号的右边必须是 C 语言中合法的表达式。如果赋值号两侧的类型不一致，系统会自动将右侧表达式求得的数值按赋值号左边的变量类型进行转换，但这种转换仅限于数值型数据之间(如地址值就不能赋给一般变量)。转换规则如下。

1．将实型数赋值给整型数时，舍弃实数的小数部分。如：

```
int i;
i=2.86;
```

则 i 的值为 2。注意：是舍去小数部分，而不是四舍五入。

2．将整型数赋值给实型数时，数值不变，以浮点数的形式存储到变量中。如：

```
float a;
a=3;
```

则 a 的值为 3.0，而不是 3(整型)。

3．字符型数据赋给整型变量时：当字符型数据为无符号数据时，将字符数据放到整型数据的低 8 位，高 8 位均补 0；当字符型数据为有符号数据时，将字符数据放到整型数据的低 8 位，高 8 位全补上字符数据的符号位。如：

```
signed char a=0x86;              //a=0b10000110
signed int b;
b=a;                             //b=0b1111111110000110
```

4. 将整型数据赋值给长整型数据时：当整型数据为无符号数据时，将整型数据放到长整型数据的低 16 位，高 16 位均补 0；当整型数据为有符号数据时，将整型数据放到长整型数据的低 16 位，高 16 位全补上整型数据的符号位。

5. 将有符号数赋给长度相同的无符号数据时，连符号一起作为数值传递。要注意：当有符号数为正数时，赋值后的值不变，当有符号数为负值时，赋值后的值就会发生变化。如：

```
unsigned int a;                  //a 为无符号整数
int b=-1;                        //b 为-1
a=b;                             //将 b 赋值给 a
```

运算结果：a=65535

6. 将无符号数赋给长度相同的有符号数据时，赋值机制同上。如果无符号数的最高位为 1，赋值后按负数处理。如：

```
unsigned char a=0x81;            //a=129
signed char b;
b=a                              //b=-1
```

复合赋值表达式：

C 语言规定可以使用以下 10 种复合赋值表达式：

+=、-=、*=、/=、%=、<<=、>>=、&=、^=、|=

如：

```
i+=1;                            //等价于 i=i+1
x*=y+8                           //等价于 x=x*(y+8)
```

采用这些复合运算符，有利于简化程序书写，并且可以提高编译效率，产生质量较高的目标代码。

2.1.2.8 逗号运算符和逗号表达式

C 语言提供一种特殊的运算符——逗号运算符，用它将两个表达式连接起来。C 语言规定：逗号运算符的结合为从左到右，因此逗号表达式将从左到右进行运算。在所有运算符中，逗号表达式的优先级别最低。

2.2　数据的输入/输出

C 语言本身并不提供输入输出语句，输入输出操作是通过调用输入输出函数来实现的。在 ANSI C 中提供了一些输入输出函数，如 printf 函数和 scanf 函数等，其中输入一般指键盘，输出一般指显示器，因此 ANSI C 提供的 stdio.h 库中的大多数函数并不适用于单片机。在 ICCAVR 中的 stdio.h 库中仅保留了部分 ANSI C 的输入输出函数，并且针对 AVR 单片机的特点进行了改造，随着 ICCAVR 版本的提升，将逐渐增加对 ANSI C 输入输出函数的支持。ICCAVR 的 stdio.h 库中输入输出是针对 AVR 单片机的 UART 进行操作，在使用前应包含“#include "stdio.h"”预处理，并且需要初始化 UART 端口。在实际应用中，大部分情况下输入输出的实现均需自己重新编写相应的函数。

由于数据的输入输出是 C 语言中一个重要的组成部分，因此本节结合 ICCAVR 的输入输出函数特点讲述数据的输入输出及其格式。

2.2.1　数据输入

1．getchar 字符输入函数

在 ANSI C 中此函数的作用是从终端(或系统隐含的输入设备，如键盘)输入一个字符。getchar 没有参数。一般格式为：

```
getchar ()
```

getchar 函数得到的字符可以赋给一个字符变量或整型变量，也可以不赋给任何变量，作为表达式的一部分：

```
unsigned char c;
c=getchar();
printf("%c",c);
```

最后两句也可以用以下语句代替：

```
printf("%c",getchar());
```

在 ICCAVR 中，getchar 函数使用查询方式从 UART 返回一个字符，一般格式为：

```
int getchar();
```

ICCAVR 规定：

(1) 如果函数返回比整型参数小的参数类型(如 char)，都将其扩展成整型(int)数据返回。

(2) 如果函数的实参为比整型参数小的参数类型(如 char)，都将其扩展成整型(int)数据。

所以在 ICCAVR 中，getchar( )的返回类型是 int，而不是 ANSI C 中的 char 类型，下面介绍 scanf 函数的返回值也是 int 类型。对于 ANSI C 中的字符类型函数，如 isalnum 函数，其形参和返回值都是 char 型，而在 ICCAVR 中均为 int 型，也是这个原因。

2．scanf 格式输入函数

在 ICCAVR 中，本函数是通过调用 getchar( )实现读入功能的，scanf 函数可以用来输入任何类型的多个数据。一般格式为：

```
scanf (格式控制,地址列表)
```

ICCAVR 中的 scanf 格式控制字符如表 2.5 所示。

表 2.5 ICCAVR 中 scanf 格式控制字符

格 式 字 符	说 明
d	输入十进制整数
x	输入以 0x 开头无符号的十六进制整数
X	输入以 0X 开头无符号的十六进制整数
u	输入无符号的十进制整数
o	输入无符号的十进制整数，与 u 相同
c	输入一个字符型数据
l(字母 l)	插在%号之后，d、x、X、u、o 前，说明为长整数类型

从上表中可以看到，在 scnaf 中，ANSI C 中的有些格式控制符在 ICCAVR 中并不完全通用，主要有以下区别：

(1) 有些格式控制符相同，但控制内容不同，如%o，在 ANSI C 中用来说明输入八进制整数，而在 ICCAVR 中，用来说明输入无符号的十进制整数，与%u 相同。

(2) ANSI C 中的有些格式控制符，在 ICCAVR 中并不支持，如%s、%f、%e 等。

(3) ANSI C 中一般不使用%u 说明符，对于 unsigned 型数据，通常以%d、%o 或%x 格式输入。

说明：上表仅针对 ICCAVR 6.26C 版本，随着版本的增加，支持的格式控制字符会增加。

在应用格式控制符时应注意：

(1) 如果在"格式控制"字符串中除了格式说明以外还有其他字符，则在输入数据时应输入与这些字符相同的字符。

(2) 在格式控制符后面应该是变量的地址，而不能是变量名。scanf ("%d,%d",&a,&b); //&a、&b 代表变量 a、b 的地址。

(3) scanf 格式输入函数。这是 ICCAVR 新增的输入函数，除了输入对象为数据缓冲区外，其他的均与 scanf 函数相同。

2.2.2　数据输出

1．putchar 字符输出函数

在 ANSI C 中，putchar 函数的作用是向终端(显示器)输出一个字符。

在 ICCAVR 中，putchar 函数向 UART 输出一个字符。要注意，如果输出到 Windows 的超级终端时，应在每一个输出的字符后面再输出“\n”，才能得到正确的显示结果。

2．puts 字符串输出函数

```
puts (char *s)
```

输出指针 s 指向的一个字符串。

3．printf 格式输出函数

在 ICCAVR 中，printf 函数可以用来输出多种类型的多个数据。一般格式为：

```
printf (格式控制，地址列表)
```

ICCAVR 中的 printf 格式控制字符如表 2.6 所示。

表 2.6　ICCAVR 中 printf 格式控制字符

格 式 字 符	说　　明
d	输出有符号十进制整数
o	输出无符号八进制整数
x	输出无符号十六进制整数，字母使用'a'~'f'
X	输出无符号十六进制整数，字母使用'A'~'F'
u	输出无符号十进制整数
s	输出一个以空字符('\0'或 NULL)为结束符的字符串
c	以 ASCII 码输出，只输出一个字符
f	以小数的格式(aaa.bbbb)输出一个浮点数
S	输出在 FLASH 存储器中的字符串变量
g 或 G	以十进制小数或科学计数法输出一个浮点数 (自动选择其占用输出宽度较小的一种格式)

在 ICCAVR 中，可以精确地控制输出的格式，具体如下：

%[标记]*[宽度][.精度][l]<需转化字符>

标记：

(1) 如果在“%”和“x”或“X”之间有一个“#”标记，那么会自动在输出数据的前面分别加上 0 或 0x 的引导。

(2) “-”(负号)，则输出数据左对齐。

(3) “+”(正号)，则输出正整数时在前面加上一个“+”号。

(4) “0”(零)，用“0”代替输出的空格。

宽度：

只能是一个大于零的正整数，用于指定输出的最小宽度，当指定的最小宽度小于或等于输出的实际宽度时，本指定宽度格式无效。如果指定宽度大于输出实际宽度时，则按照标记指定的格式插入相应的空格或零。

精度：

只能是一个大于零的正整数，当后面跟着“s”格式说明符时，用于指定输出字符的个数，当后面跟着其他格式说明符时，用于指定输出小数点以后数据的位数。

l(L 的小写字母)：

如果在“%”和一个整数格式字符之间有一个“l”字符，则输出长整型整数。

在 ICCAVR 的设置中有一个 PRINTF Version 选项(Project->Options->Target)，有三类输出格式：

短整型(或基本型)(small)，只支持%c、%d、%x(%X)、%u、%s 几种输出格式

长整型(long)，除了支持短整型类，还增加%ld、%lx(%lX)、%lu

浮点型(float)，除了支持长整型类，还增加%f

采用何种输出类型，取决于实际中输出精度的需要，只要选取适当的输出类型即可。从短整型(small)→长整型(long)→浮点型(float)，程序代码增加很多，运行速度降低。

4．sprintf 格式输出函数

除了输出对象为数据缓冲区外，其他的均与 printf 函数相同。

2.3 逻辑运算和判断选取控制

2.3.1 关系表达式和逻辑表达式

1．关系运算

关系运算是逻辑运算中比较简单的一种，所谓的“关系运算”实际上就是“比较运算”，将两个值进行比较，判断比较的结果是否符合给定的条件。

C 语言提供以下 6 种关系运算符：

< <= > >= == !=

(1) 其中前 4 种关系运算符的优先级别相同，后 2 种也相同，而前 4 种高于后 2 种。

(2) 关系运算符是双目运算符，具有自左至右的结合性。

(3) 关系运算符的优先级低于算术运算符，高于赋值运算符。

例：

```
c>a+b;                    //等效于 c>(a+b)
```

```
c>b!=c;                          //等效于(a>b)!=c
a==b<c;                          //等效于 a==(a<c)
```

2．关系表达式

用关系运算符将两个表达式连接起来的表达式，称关系表达式。关系运算符两边的运算对象可以是 C 语言中任意合法的表达式。C 语言不像其他高级语言一样有专门的“逻辑值”，因此规定用 0 代表“假”，用非 0 代表“真”，关系表达式的值是一整数：0 或 1。当表达式成立时，表达式值为整数 1，当表达式不成立时，值为整数 0。例：

```
int a=4;
float b=3.0;
a>b;                             //表达式成立，值为 1
'a'<'b';                         //'a'为 97，'b'为 98，表达式不成立，值为 0
(a=4)>(b>4);                     //(a=4)的值为 1，(b>4)的值为 0，
                                 //1>0 成立，表达式的值为 1
```

关系运算符两边值类型不一致时将自动转换。若一边为整型，一边为实型，系统自动把整型转换为实型，然后进行比较。当 a、b 为实型时，应当避免使用 a==b 这样的关系表达式，因为存放在内存中的实型数是有误差的，可以比较大小，但不可能精确相等，这将导致关系表达式 a==b 的值始终为 0。

3．逻辑运算符

C 语言提供以下三种逻辑运算符：

&&　||　!

其中“&&”和“||”是双目运算符，它要求有两个操作数，而“!”是单目运算符，只要求有一个操作数，逻辑运算符具有自左向右的结合性，其优先级次序是：!>&&>||。

注意：“&”和“|”是位运算符，不要将逻辑运算符与位运算符混用。

4．逻辑表达式

用逻辑运算符将关系表达式或逻辑量连接起来就是逻辑表达式，逻辑运算对象可以是 C 语言中任意合法的表达式。C 语言编译系统在给出逻辑运算结果时，其结果不是 0 就是 1，不可能是其他数值，而在逻辑表达式中作为参加逻辑运算的运算对象(操作数)可以是任何数值(包括整型、实型、字符型、指针型数据)，将 0 判断为“假”，非 0 判断为“真”。例：

```
unsigned char a=1,b=2,c=0
a&&b;                            //a、b 的值均非 0，表达式的值为 1
b&&c;                            //b 的值非 0，c 的值为 0，表达式的值为 0
b||c;                            //b 的值非 0，表达式的值为 1
!c;                              //C 的值为 0，表达式的值为 1
```

要注意：

(1) 在数学上 0<a<10 表示 a 的值在大于 0 小于 10 的范围之内，但是 C 语言中用 0<a<10 表达式时，无论 a 为任何值，表达式的值总是 1，只有用 0<a&a<10 逻辑表达式才能正确表示上述关系。

(2) 在逻辑表达式的求解中，并不是所有的逻辑运算符都被执行，只是在必须执行下一逻辑运算符才能求出表达式的解时，才执行该运算符。如：++a&b++。若++a 的值为 0，系统完全可以确定逻辑表达式的结果总是为 0，因此将跳过 b++而不再对它求值，在这种情况下，a 的值将自增 1，而 b 的值不变。只有++a 的值不为 0，系统才会对 b++进行求值。表达式 a++|b++也相似。

熟练掌握 C 语言的关系运算符和逻辑运算符后，可以巧妙地用一个逻辑表达式来表示一个复杂的条件。例如，判别某一年 year 是否是闰年，必须符合下面的条件之一：

(1) 能被 4 整除，但不能被 100 整除。

(2) 能被 4 整除，又能被 400 整除。

可以用一个逻辑表达式来表示：

```
(year%4==0&&year%100!=0)||year%400==0
```

当上述表达式的值为 1，则 year 为闰年，否则不是闰年。

2.3.2 if 语句

if 语句是用来判定所给定的条件是否成立，根据判定的结果(真或假)决定执行给出的两种操作之一。C 语言提供了三种形式的 if 语句：

1. if(表达式) 语句;

```
例: if (x>y) a=1;                    //如果 x 大于 y，则把 1 赋值给 a
```

2. if(表达式) 语句 1;
else 语句 2;

```
例: if (x>y) a=1;                    //如果 x 大于 y，则把 1 赋值给 a
    else a=0;                        //如果 x 小于等于 y，则把 0 赋值给 a
```

3. if(表达式) 语句 1;

```
else if (表达式) 语句 2;
else if (表达式) 语句 3;
┆
else if (表达式) 语句 n;
```

说明：

(1) 表达式的类型不限于逻辑表达式，可以是任意合法的 C 语言表达式(如：逻辑表达式、关系表达式、算术表达式、赋值表达式等)，也可以是任意类型的数据(如：整型、实型、字符型、指针型等)，下面的语句是合法的：

```
if (3) a=1;                          //条件表达式的值为 3，非 0，将 1 赋值给变量 a
if ('a') a=1;                        //'a'的 ASCII 码为 97，非 0，将 1 赋值给变量 a
```

表达式也可以是复杂的表达式，如：

```
if (a+b<c&&b+c>a&&c+a>b)
      {
      a=10;
      b=20;
      c=30;
      }
```

(2) 注意 else 子句不能作为单独的语句使用，它必须是 if 语句的一部分，与 if 配对使用。

(3) 不能多加或忘掉“;”号。例如：

```
if (a>0)                             //后面不能加分号
      a=1;                           //后面不能忘记加分号
else                                 //后面不能加分号
      a=0;                           //后面不能忘记加分号
```

如果多加或漏加“;”号，会出现语法错误或者与设计者的意图不符。

4. 在 if 和 else 的后面只能含一个操作语句，如果需要多个操作语句，必须用大括号“{}”括起来作为一个复合语句。

5. 在 if 语句中又包含一个或多个 if 语句，在使用时必须注意 if 与 else 的配对关系，从最内层开始，else 总是与它上面最近未曾配对的 if 配对，如果 if 与 else 的数目不一样时，可以加大括号来确定配对关系。

C 语言具有比较自由的书写格式，但是过于“自由”的程序书写格式，往往使人们很难读懂，因此要求读者在编写程序时应参照下例，采用阶梯形按层缩进的书写格式来书写：

```
if (表达式 1)
    语句 1
else
    if (表达式 2)
        语句 2
    else
        if (表达式 3)
```

```
        语句 3
    else
        语句 n
```

[例 2.3] 根据输入的学生成绩，给出相应的等级。90 分及 90 分以上的等级为 A，80 分及 80 分以上的等级为 B，70 分及 70 分以上的等级为 C，60 分及 60 分以上的等级为 D，60 分以下的等级为 E。

```
void main(void)
    {
    int g;
    printf("Enter g:  ");
    scanf("%d",&g);
    printf("g=%d:",g);
    if    (g>=90)   printf("A\n");
    else  if(g>=80)   printf("B\n");
    else  if(g>=70)   printf("C\n");
    else  if(g>=60)   printf("D\n");
    else            printf("E\n");
    }
```

当执行以上程序时，首先输入学生的成绩，然后在 if 语句中依次对学生成绩进行判断，输出相应的语句。

2.3.3 条件运算符

条件运算符要求有三个操作对象，称三目运算符，它是 C 语言中惟一的三目运算符，其表达式的一般形式为：

```
表达式 1 ? 表达式 2 : 表达式 3;
```

说明：

(1) 条件运算符的执行顺序。先求解表达式 1，若值为真(非 0)，则求解表达式 2，此时表达式 2 的值就作为整个条件表达式的值。若表达式 1 的值为假(0)，则求解表达式 3 的值，此时表达式 3 的值就作为整个条件表达式的值。如下例：

```
max=(a>b)?a:b;
```

由于条件运算符优先于赋值运算符，低于关系运算符和算术运算，因此上面赋值表达式的求解过程是先求解条件表达式的值，再将值赋给变量 max。

(2) 条件运算符的结合方向为“自右向左”，如果有以下表达式：

```
a>b?a:c>d?c:d,
```

则相当于

```
a>b?a:(c>d?c:d)
```

(3) 条件表达式中，表达式 1 的类型可以与表达式 2 和表达式 3 的类型不同。如下例：

```
char x, y;
┆            //给变量 x 和 y 赋值
x>y ?1.5:1
```

表达式 1 中 x 和 y 均为 char 型，表达式 2 中，1.5 为 float 型，表达式 3 中，1 为 int 型。

当 x>y 时，表达式 1 成立，则整个条件表达式的值为 1.5(float 型)；当 x<=y，表达式 1 不成立，整个条件表达式的值本应为 1，但是由于表达式 2(float)的类型比表达式 3(int)的高，因此自动将表达式 3 转换为 float 型，因此最后整个条件表达式的值为 1.0。

2.3.4　switch 语句

switch 语句是多分支选择语句，它的一般形式如下：

```
switch (表达式)
     {
     case 常量表达式 1: 语句 1; bread;
     case 常量表达式 2: 语句 2; bread;
          ┆
     case 常量表达式 n: 语句 n; bread;
     default            语句 6
     }
```

说明：

(1) 新 ANSI C 标准允许 switch 后面括号内的表达式和 case 常量表达式可以是任何类型的数据。

(2) 当表达式的值与 case 后面的常量表达式的值相等时，就执行此 case 后面的语句，若所有 case 中的常量表达式的值都没有与表达式的值匹配，就执行 default 后面的语句，如果没有 default 语句，就跳过 switch 语句体，什么也不做。

(3) 每一个 case 的常量表达式必须互不相同，但各 case 出现的次序不影响执行结果。case 语句标号后面语句可以省略不写，在关键字 case 和常量表达式之间一定要有空格。

(4) 在 ICCAVR 6.26C 中，case 的常量表达式的数目不能超过 255 个。

使 switch 语句改写[例 2.3]，改后如下：

[例 2.4]

```
void main(void)
     {
     int g;
     printf("Enter g:  ");
     scanf("%d",&g);
     printf("g=%d:",g);
     switch (g/10)
           {
           case 10 :
           case 9 :   printf("A\n");break;
           case 8 :   printf("B\n");break;
           case 7 :   printf("C\n");break;
           case 6 :   printf("D\n");break;
           default:   printf("E\n");
           }
     }
```

2.4 循 环 控 制

在 C 语言中可以用以下语句来实现循环：

1. 用 goto 语句
2. 用 while 语句
3. 用 do...while 语句
4. 用 for 语句

下面将分别给予介绍。

2.4.1 goto 语句以及用 goto 语句构成循环

goto 语句为无条件转向语句，它的一般形式为：

```
goto 语句标号
```

语句标号不必特殊加以定义，可以是任意合法的标识符，在标识符后面加一个冒号，如“warn:”，该标识符就成为一个语句标号。在 ANSI C 中，标号可以与变量名同名。ICCAVR 6.26C 中支持标号与变量名同名(完全相同，即字母的大小写都可以相同)，编译器

还是可以将同名的标号与变量名识别为两个不同的标识符，但有些 C 编译器就不支持标号与变量名同名(如 CVAVR，即使标号与变量的大小写不同，编译器也会认为是同一标识符而出错)。实际应用中，为了防止使用上的混乱，建议读者在编写程序时，标号与变量名不同。

结构化程序设计方法主张限制使用 goto 语句，因为滥用 goto 语句将使程序流程无规律，可读性差。但也不是绝对禁止使用 goto 语句。一般来说，使用 goto 语句可以有以下两种用途：

1．与 if 语句一起构成循环结构

[例 2.5] 求 1 到 100 的整数累加和。

```
void main(void)
     {
     int i,sum=0;
     i=1;
     loop: if (i<=100)
           {
           sum=sum+i;
           i++;
           goto loop;
           }
     printf("%d",sum);
     }
```

2．从循环体中跳转到循环体外

在 C 语言中可以用 break 语句和 continue 语句跳出本层循环和结束本次循环，goto 语句的使用机会已大大减少，只是需从多层循环的内层跳到多层循环体外时才用到 goto 语句。但是这种用法不符合结构化原则，不宜采用，只有在特殊情况(如需要大大提高生成代码的效率)时才使用。

2.4.2　while 语句

while 用来实现“当型”循环结构，其一般形式如下：

```
while (表达式)  语句
```

while 语句的特点是先判断，再执行语句。当表达式为非 0 值时，执行 while 语句中的内嵌语句，当表达式值为 0 时，不执行内嵌语句。

[例 2.6] 将[例 2.5]求 1 到 100 的整数累加和改为使用 while 语句的形式。

```
void main (void)
     {
     int i, sum=0;
```

```
i=1;
while (i<=100)
      {
      sum=sum+i;
      i++;
      }
printf ("%d",sum);
}
```

在使用时需要注意：

(1) 不要把 while 语句构成的循环结构与 if 语句构成的选择结构弄混了，while 的条件表达式为非 0 时，其后的循环体将被重复执行，而 if 的条件表达式的值为非 0 时，其后的 if 子句只执行一次。

(2) 循环体如果包含一个以上的语句，应该用大括号括起来，以复合语句的形式出现。如果不加大括号，则 while 语句的范围只到 while 后面第一分号处。

(3) 在循环体中应有使循环趋向于结束的语句。如果无此语句，则进入死循环。

2.4.3 do...while 语句

do...while 语句用来实现“直到型”循环结构，其一般形式为：

```
do   语句;
while (表达式);
```

说明：

(1) do...while 语句的特点是先执行语句，再判断表达式是否为真。当表达式为非 0 值时继续执行 do 与 while 之间的语句；当表达式值为 0 时，不再执行 do 与 while 之间的语句，转向 while 后面的语句。

(2) do...while 语句由 do 开始，用 while 结束，必须注意的是，在 while (表达式)后面的分号不可丢，它表示 do...while 语句的结束。

下面将[例 2.5]求 1 到 100 的整数累加和的程序改为使用 do...while 的形式。

[例 2.7]

```
void main (void)
   {
   int i, sum=0;
   i=1;
   do
        {
```

```
        sum=sum+i;
        i++;
        }
    while (i<=100);
    printf ("%d",sum);
    }
```

可以看到，对于同一个问题，可以用 while 语句处理，也可以用 do...while 语句处理。在一般情况下，用 while 语句和用 do...while 语句处理同一问题时，若二者的循环体部分是一样的，它们的结果也是一样。但是在 while 后面的表达式一开始就为假(0)时，while 一次也不执行循环体，而对 do...while 循环来说，则会执行一次循环体。

2.4.4　for 语句

C 语言中，for 语句的使用最灵活，不仅可以用于循环次数已确定的情况，而且可以用于循环次数不确定而只给出循环结束条件的情况，它完全可以代替 while 语句。for 语句的一般形式为：

```
for (表达式 1;表达式 2;表达式 3)  语句;
```

for 中的三个表达式可以是任意形式的表达式，通常用于 for 循环控制，紧跟在 for 之后的语句，在语法上要求是一条语句。若需要多条语句，应该用大括号括起来组成复合语句。

它的执行过程如下：

(1) 先求解表达式 1。

(2) 求解表达式 2，若其值为真(非 0)，则执行 for 语句中的内嵌语句，然后执行下一步(3)，若为假(0)，则结束循环，转到第 5 步。

(3) 若表达式为真，在执行指定的语句后，求解表达式 3。

(4) 转回上面第 2 步骤继续执行。

(5) for 语句执行结束，执行后面的下一个语句。

以下将[例 2.5]求 1 到 100 的整数累加和的程序改为使用 for 的形式。

[例 2.8]

```
void main (void)
    {
    int i, sum=0;
    for (i=1;i<=100;i++)
        sum=sum+i;
    printf ("%d",sum);
    }
```

显然，用 for 语句简单、方便。下面对 for 语句说明如下：

(1) for 语句一般形式的“表达式 1”可以省略，此时应该在 for 语句之前给循环变量赋初值。注意省略表达式 1 时，其后面的分号不能省略。

(2) 如果表达式 2 省略，即不判断循环条件，循环无终止地进行下去，也就是认为表达式 2 始终为真。

(3) 表达式 3 也可以省略，但此时应该另外设法保证循环能正常结束。

(4) 可以省略表达式 1 和表达式 3，只有表达式 2，即只给出循环条件，如：

```
for ( ;i<=100;)                          while (i<=100)
  {sum=sum+i;          相当于：              {sum=sum+i;
   i++                                      i++
   }                                        }
```

在这种情况下，完全等同于 while 语句，可见 for 语句比 while 语句功能强，除了可以给出循环条件外，还可以赋初值，使循环变量自动增加等功能。

(5) 三个表达式都可以省略，但两个“;”不能省略，如：

```
for ( ; ; )  {语句;}        相当于:        while (1)  {语句;}
```

即不设初值，不判断条件，循环变量不增值，循环将会无限制地执行，形成死循环。通常单片机在执行完初始化程序后，就利用上述两种循环语句，无终止地执行循环体。

(6) 表达式 1 和表达式 3 可以是设置循环变量初值的赋值表达式，也可以是与循环变量无关的其他表达式，既可以是一个简单的表达式，也可以是逗号表达式(即包含一个以上的简单表达式，中间用逗号间隔)，可以将[例 2.8]改写为以下的形式：

[例 2.9]

```
void main (void)
   {
   int i, sum;
   for (i=1,sum=0;i<=100;sum=sum+i,i++);
   printf ("%d",sum);
   }
```

在 for 后面的圆括号中出现各种形式的、与循环控制无关的表达式，虽然在语法上是合法的，但这会降低程序的可读性。建议初学者在编程时，在 for 后面的一对括号中仅包含对循环进行控制的表达式，其他操作尽量放到循环体内去完成。

2.4.5　几种循环的比较

(1) 以上四种循环体都可以用来处理同一问题，一般情况下，它们可以互相代替，但不提倡用 goto 型循环。

(2) while 和 do...while 循环，只在 while 后面指定循环条件，在循环体中包含应反复执行的操作语句，包括使循环趋于结束的语句(如 i++或 i=i+1 等)。

for 循环可以在表达式 3 中包含使循环趋于结束的操作，甚至可以将循环体中的操作全部放到表达式 3 中。因此 for 语句的功能最强，凡是用 while 循环能完成的，用 for 循环都能实现。

(3) 用 while 和 do...while 循环时，循环变量初始化的操作应在 while 和 do...while 语句之前完成，而 for 语句可以放在表达式 1 中实现循环变量的初始化。

(4) while 循环、for 循环是先判断表达式，后执行语句；而 do...while 循环是先执行该语句，再判断表达式。

(5) 对 while 循环、do...while 循环和 for 循环，可以用 break 语句跳出本层循环，用 continue 语句结束本次循环。而对用 goto 语句和 if 语句构成的循环，不能用 break 语句和 continue 语句进行控制。其中 break 语句可以使用流程跳出循环体，提前结束循环，接着执行循环下面的语句；而 continue 语句的作用是结束本次循环，即跳过循环体中下面尚未执行的语句，接着进行下一次是否执行循环的判断。

(6) 一个循环体内又包含另一个完整的循环结构，我们称之为循环嵌套，内嵌的循环中还可以再嵌套循环，这就是多层循环嵌套。

2.5　数　　组

数组是 C 语言提供的一种最简单的构造类型。每个数组包含一组具有同一类型的变量，这些变量在内存中占有连续的存储单元；在程序中，这些变量具有相同的名字，但具有不同的下标。在用 C 语言的程序设计中，数组是一种十分有用的数据结构，有许多问题，不用数组几乎难以解决。

2.5.1　一维数组的定义和引用

1. 一维数组的一般定义方式

类型说明符　数组名[常量表达式];

例如：int a[10];

它表示定义一个一维数组，数组名为 a，此数组有 10 个元素。

说明：

(1) 数组名定义规则和变量名相同，遵循标识符定义规则。

(2) 数组名后面是用方括号括起来的常量表达式，不能用其他符号。

(3) 常量表达式表示元素的个数，即数组长度。例如 a[10]表示 a 数组有 10 个元素，下标从 0 开始，这 10 个元素是：a[0]、a[1]、a[2]、a[3]、a[4]、a[5]、a[6]、a[7]、a[8]、a[9]。要注意，a[10]这个数组中并没有 a[10]这个数组元素。

(4) 常量表达式中可以包括常量和符号常量，不能包含变量。

下面的数组定义是允许的：

```
#define n 10
int a[n];
```

而下面的数组定义是错误的，因为 n 是变量。

```
int n=10;
int a[n];
```

2. 一维数组的引用

C 语言规定：数组必须先定义后使用，而且只能逐个引用数组元素，而不能一次引用整个数组。

```
a[0]=a[5]+a[7]-a[3*4-1];
```

一个数组元素实质上就是一个变量名，代表内存中的一个存储单元，一个数组占有一串连续的存储单元。在引用数组元素时，数组元素下标表达式的值必须是整数，下标表达式的下限为 0。C 语言在运行过程中，系统并不自动检验数组元素的下标是否越界，因此数组两端都因为可能越界而破坏了其他存储单元的数据，在编写程序时保证数组下标不越界是十分重要的。

3. 一维数组元素的初始化

可以用赋值语句或输入语句使数组元素初始化。

```
int a[3];
a[0]=0;
a[1]=1;
a[2]=2;
```

这种赋初值方式占用运行时间，也可以使数组在程序运行之前初始化，即在编译阶段使之得到初值。初始化可以用以下方式实现：

```
int a[3]={0,1,2,};
```

经过以上的定义和初始化后，a[0]=0，a[1]=1，a[2]=2。

注意：

(1) 可以只给一部分元素赋值。例如：

```
int a[10]={0,1,2,3};
```

定义数组有 10 个元素，但只提供 4 个初值。编译时给前面 4 个元素赋指定的初值，后面 6 个元素系统自动赋初值为 0。

(2) 如果想使一个数组中全部元素值为 0，可以写成：

```
int a[10]={0,0,0,0,0,0,0,0,0,0};
```

或者写成：

```
int a[10]={0};
```

注意，不能写成：

```
int a[10]={0*10};
```

因为不能给数组整体赋初值，也不能不写任何数值，如：

```
int a[10]={ };
```

对于静态存储类型的数组(static)也可以不指定初值，程序启动时相当于对所有静态类型数组元素赋初值 0。如：

```
static int a[10];
```

在程序启动时会自动将 a[0]~a[9]赋初值 0。因为 ANSI C 规定在启动时必须对所用到的内存清零，因此 ICCAVR 在启动时，会将所有使用到的内存全部清 0，也相当于将数组 a[10]中的 10 个元素赋初值 0。

(3) 在对全部数组元素赋初值时，可以不指定数组长度。例如：

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

它相当于：

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

[例 2.10] 用冒泡法对 10 个数按从小到大的顺序进行排序。

```
void main(void)
     {
     int a[11];
     int i,j,t;
     printf("input 10 numbers: \n");
```

```
for (i=1;i<=10;i++)
      scanf("%d",&a[i]);
printf("\n");
for (j=1;j<=9;j++)
      for (i=1;i<=10-j;i++)
            if (a[i]>a[i+1])
                  {
                  t=a[i];a[i]=a[i+1];a[i+1]=t;}
printf("the sorted numbers:\n");
for (i=1;i<=10;i++)
      printf("%d  ",a[i]);
}
```

2.5.2 二维数组的定义和引用

1．二维数组的定义

二维数组定义的一般形式为：

类型说明符 数组名[常量表达式1][常量表达式2]

例如：int a[3][4]

它表示定义二维数组，数组名为a，有3行4列共12个元素。

说明：

(1) 二维数组可以看作是一种特殊的一维数组，如上例中，a[3][4]可以看作一个一维数组，它有3个元素：a[0]、a[1]、a[2]，每个元素又是一个包含4个元素的一维数组。

(2) 二维数组中元素在内存中排列顺序是：按行存放，即在内存中先顺序存放第一行的元素，再存放第二行的元素，一直这样排放下去。

2．二维数组的引用

二维数组元素用以下形式表示：

数组名[下标1][下标2]

其中下标可以是整数，也可以是整型表达式。如 a[2-1][6*2/3-1]，在使用数组时，应该注意下标值应在已定义的数组大小的范围之内。

请读者注意区分在定义数组时用a[3][4]和引用元素时的a[3][4]的区别。前者用来定义数组的维数和各维的大小，后者中的3和4是下标值，a[3][4]代表一个元素。当然，定义一个a[3][4]的数组，并没有a[3][4]这个数组元素。

3．二维数组的初始化

可以用以下方式给二维数组初始化：

(1) 分行给二维数组赋初值。

```
int a[3][4]={{1,2,3,4},{5,6,7,8},{9,10,11,12}};
```

(2) 也可以将所有数据写在一个大括号内，按数组排列的顺序对各元素赋初值。如：

```
int a[3][4]={1,2,3,4,5,6,7,8,9,10,11,12};
```

第二种赋初值方式效果与方式 1 相同，但是用方式 1 赋初值的方法比较好，一行对一行，界限清楚。而用方式 2，如果数据多，写成一大片，容易遗漏，也不容易检查。

(3) 可以只对部分元素赋初值。

```
int a[3][4]= {{1,2},{ },{9}};
```

(4) 如果对全部元素都赋初值，则定义数组时对第一维的长度可不指定，但第二维的长度不能省略。如：

```
int a[][4]={{1,2,3,4},{5,6,7,8},{9,10,11,12}};
```

或：

```
int a[][4]={1,2,3,4,5,6,7,8,9,10,11,12};
```

2.5.3　字符数组

2.5.3.1　字符型数组

用来存放字符型数据的数组是字符数组，字符数组中一个元素存放一个字符，注意：'a'、'A'是不同的字符常量。' '(两个单引号中有一个空格符)也是一个字符常量(空格字符常量)，但是在表示空格字符常量时，不能写成两个连续的"号。

1．字符数组的定义

字符数组的定义与普通的一维数组的定义类似，如：

```
char a[10]={'I','C','C','A','V','R'};
```

注意：如果括号中提供的初值个数(即字符个数)大于数组长度，C 语言作为语法错误处理(对于 Keil、ICCAVR、GCCAVR 等几乎全部的 C 编译器均提示语法错误)，如果初值个数小于数组长度，则只将这些字符赋给数组中前面那些元素，其余的元素自动定义为空字符(即'\0')。则上面字符数组的状态如下所示：

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
I	C	C	A	V	R	\0	\0	\0	\0

如果提供的初值个数与预定的数组长度相同，在定义时可以省略数组长度，系统会根据初值个数确定数组长度。如：

```
char a[]={'I','C','C','A','V','R'};
```

相当于：

```
char a[6]={'I','C','C','A','V','R'};
```

2．字符数组的引用

可以引用字符数组中的任一个元素，得到一个字符。

[例 2.11]

```
void main (void)
     {
     char a[]={'I','C','C','A','V','R'};
     int i;
     for (i=0;i<6;i++)
     printf ("%c",a[i]);
     }
```

运行结果：ICCAVR

2.5.3.2 字符串和字符串结束标志

字符串常量是由双引号括起来的一串字符。在 C 语言中，将字符串常量作为字符数组来处理。如在上例中就是用一个一维字符型数组来存放一个字符串常量“ICCAVR”，这个字符串的实际长度与数组的长度相等。有时，我们关心的是有效字符串的长度而不是字符数组的长度，例如，在上例中定义一个字符数组 a 的长度为 10 个字节，而实际的有效字符只有 6 个字节。为了便于测定字符串的实际长度，C 语言规定以字符“\0”作为字符串结束标志。也就是说，在遇到第一个字符“\0”时，表示字符串结束，由它前面的字符组成字符串。系统对字符串常量也自动加放一个“\0”作为结束符，例如字符串“Atmel”共有 5 个字符，但系统将该字符串作为一维数组来存放，在字符串的结尾自动加放一个“\0”符号，因此整个字符串共占用 6 个字节的存储空间。

有了结束标志“\0”后，字符数组的长度就显得不那么重要了。在程序中，往往依靠检测“\0”来判定字符串是否结束，而不是依据数组长度来决定字符串是否结束。当然，在定义字符数组时应该估计实际字符串的长度，使数组长度始终至少大于字符串实际长度 1 个字节，以防止出错。

字符数组并不要求它的最后一个字符为“\0”，可以不包含“\0”字符，但是我们为了使字符型数组和字符串常量的处理方法一致，一般情况下在字符数组中也加上“\0”，这样便于测定字符数组中字符实际长度。

需要说明，“\0”代表 ASCII 码为 0 的字符，该字符不是一个可以显示的字符，是一个“空操作符”。用它来作为字符串结束标志只是起一个供辨别的标志，不会产生附加的操作或增加有效字符。

在了解了 C 语言处理字符串的方法后，我们再对字符串数组初始化做补充说明。

对字符型数组可以用以下方式进行初始化：

```
char a[]={'I','C','C','A','V','R'};
```

也可以改为用字符串常量来使字符数组初始化：

```
char a[]="ICCAVR";
```

显然，使用字符串常量来初始化数组，更符合我们的习惯，但是应注意，该字符串数组的长度不是 6 个字节，而是 7 个字节，因为系统自动在 ICCAVR 后面加上“\0”作为结束标志。

如果写成以下形式就错了：

```
char a[6]="ICCAVR";
```

由于字符串数组实际上将占用 7 个字节的内存空间，而 C 编译器仅为整个数组分配 6 个字节的内存空间，但 C 编译器也会为自动增加的“\0”分配一个字节空间。该空间已不属于数组 a，而是数组 a 之后的下一个字节空间，这样编译器在保存和处理“\0”时可能会破坏其他数据，引起不可预测的错误。更麻烦的是，对于 Keil、ICCAVR、GCCAVR 等几乎全部的 C 编译器，这条语句都能够编译通过，均不会提示错误。这一点在设计时应引起足够的重视。如果错误的定义为以下形式：

```
char a[5]="ICCAVR";
```

则所有的 C 编译器均会提示语法错误。

字符串在初始化时可以赋值给一个一维数组，但不能在程序运行中赋值给字符数组，下面的赋值方式是错误的：

```
char a[10];
a="ICCAVR";
```

应该改为以下形式：

```
char a[10]="ICCAVR";          //初始化时赋值
```

或者在运行中对数组的元素逐个赋值。

2.5.3.3　字符数组的输入输出

字符数组的输入输出有两种方法：

1．逐个输入或输出，用格式符“%c”输入或输出一个字符。

2．将整个字符串一次输入或输出，用“%s”格式符。

在使用“%s”时，应注意：

(1) 输出字符不包括结束符“\0”。

(2) 在 printf 函数中的输入项是字符数组名，而不是数组元素名。应写成如下形式：

```
printf ("%s",a);
```

而不能写成这样：

```
printf ("%s",a[0]);
```

如果仅仅是需要输出 a[0]，而不是整个数组，应用以下形式：

```
printf ("%c",a[0]);
```

(3) 如果数组长度大于字符串实际长度，遇到第一个“\0”时就结束输出。

2.5.3.4 ICCAVR 中对字符串的处理

ANSI C 语言是针对冯诺依曼结构的计算机系统而设计的，即所有的程序都必须调入内存后才能运行，在运行时，程序存储器和数据存储器是相同的(均在内存中)。而 AVR 单片机是哈佛结构的 MCU，它的程序存储器和数据存储器是分开的。标准的 C 语言只能指向数据存储器或程序存储器之一，在运行时不能随意更改指向，而在 AVR 单片机中，要求指针能够指向程序存储器和数据存储器的任意空间，因此 ICCAVR 提出两种解决的方法：

方法一，在启动时将程序中的所有用到的字符串都先复制到数据存储器中，这也是 ICCAVR 及大多数编译器默认的字符串分配方式。

ICCAVR 在启动时将存储于程序存储器的“lit”区的数据(包括字符串)复制到数据存储器，数据存储器中有程序运行中所需的字符串，因此就与 ANSI C 的存储情况相同，可直接使用 C 库函数中的字符串操作函数。

方法二，为了节省数据存储器空间，字符串不复制到数据存储器，只存在于程序存储器中。

由于 C 编译器将字符串转换为 char 型指针来处理，因此如果字符串是只分配到程序存储器中，那么对于这类字符串操作的库函数必须改成不同于指针的操作。为了实现以上目的，各种 C 编译器采用以下不同的方法：

在 keil 中引入"code"关键字，可以通过以下语句实现将表格或字符串仅保存在程序存储器中：

```
unsigned char code table[]={1,2,3,4,5};
unsigned char code a[]="Keil";
```

在 IAR 中引入"flash"关键字，可以通过以下语句实现将表格或字符串仅保存在程序存储器中：

```
flash unsigned char table[]={1,2,3,4,5};
flash unsigned char a[]="IAR";
```

在 ICCAVR 中扩展了"const"限定词，可以通过以下语句实现将表格或字符串仅保存在程序存储器中：

```
const unsigned char table[]={1,2,3,4,5};
const unsigned char a[]="ICCAVR";
```

ICCAVR 扩展了 const 限定词，这与标准的 ANSI C 有一定的冲突，会使程序在移植到其他的编译器时需要修改相关内容，导致程序的移植性降低。如在 ANSI C 的标准库中有一个函数：strcpy(char *dst, const char *src)，在该函数中，const 限定词的原意是该函数不能修改参数，而在 ICCAVR 编译器中却解释为该参数指向程序存储器。因此，ANSC C 标准库中与之类似的函数必须经过相应的改造后才能应用于 ICCAVR 编译器。由于不同的编译器对字符串的处理方式不同，也使源程序在不同编译器之间移植时需要进行相应的修改。

下面为 ICCAVR 中对字符串和常数表格分配可能出现的五种情况：

```
const int table[]={1,2,3};          //table 表格只分配进程序存储器中
const char string[]="ICCAVR";       //字符串数组只分配进程序存储器中
const char *ptr1                    //指针 ptr1 位于数据存储器空间指向程序存储器空间
                                      的字符型数据
char *const ptr2                    //指针 ptr2 位于程序存储器空间指向数据存储器空间
                                      的字符型数据
const char *const ptr3              //指针 ptr3 位于程序存储器空间指向程序存储器空间
                                      的字符型数据
```

在实际使用中，“table[ ]”、“string[ ]”和“*ptr1”三种情况比较常用，ICCAVR 均生成 LPM 指令来访问程序存储器。

注意：在 ICCAVR 中，使用 const 限定词表格及字符串不能在函数体中定义，必须在使用该语句的函数之前定义。一般情况下 const 限定词语句紧跟在头文件或全局变量之后定义，或者在一个单独的文件中定义，并将该文件加入工程中。

在下例中，将 LED 的字形表存于程序存储器中，并访问字形表。

[例 2.12]

```
unsigned char a[10];
#pragma data:code
const unsigned char led[]={0x3f,0x06,0x5b,0x4f,0x66,0x6d,0x7d,0x07,0x7f,0x6f,
                           0x77,0x7c,0x39,0x5e,0x79,0x71};    //LED 字形表
const unsigned char sl[]="http://www.sl.com.cn";
#pragma data:data
void main (void)
     {
     //用户程序
     DDRB=0xff;                     //设置 PB 端口为输出
     PORTB=led[5];                  //在数码管上显示 5
     //用户程序
```

```
    a[1]=sl[5];
    }
```

从上例中可以看出，定位于程序存储器的数据表和字符串，在作为查表使用时还是很方便的(与标准 C 使用上是一致的)，但该类数据表或字符串在作为函数的参数时就要注意。下例说明如何输出一个存储于程序存储器的字符串。

[例 2.13]

```
const char string[]="ICCAVR";
const char *sl="http://www.sl.com.cn";
void main (void)
    {
    printf("%S",string);
    printf("%S",sl);
    }
```

ICCAVR 中对 ANSI C 标准函数库中的 printf 函数进行扩展，增加了%S 参数(注意：是大写字母“S”)，可以输出仅保存于程序存储器中的字符串。

在 ICCAVR 中有一个 Strings in FLASH only 复选取框(Project->Options->Target)，如果该复选框被选中，则源程序中无论是否使用 const 限定词，ICCAVR 编译器将源程序中的全部字符串都只保存在程序存储器中。使用这个选项，有利于节省数据存储器的空间，但是在编写源程序时调用与字符串有关的库函数应十分小心，防止调用错误；如果该复选框没有被选中，则源程序中只有使用了 const 限定词的字符串，编译器才将该字符串仅保存到程序存储器，对于没有使用 const 限定词的字符串，还是按默认的字符串分配方式在启动时将字符串复制到数据存储器中。

当 Strings in FLASH only 复选框被选中时，[例 2.13]必须改为以下的形式：

```
const char string[]="ICCAVR";
const char *sl="http://www.sl.com.cn";
void main (void)
    {
    sprintf("%S",string);
    sprintf("%S",sl);
    }
```

说明：

(1) Strings in FLASH only 这个选项仅对字符串有效，对数据表格无效。

(2) 当 Strings in FLASH only 复选取框没有被选中时，只能用 printf 函数来输出字符串，而不能使用 sprintf 和 csprintf 函数，应使用“%S”参数。

(3) 当 Strings in FLASH only 复选取框被选中后，只能用 sprintf 和 csprintf 函数来输出字符串，而不能使用 printf 函数，且在使用 sprintf 和 csprintf 函数只能使用“%S”参数。

(4) 在实际使用中，一般不选中 Strings in FLASH only 复选框，而是由编程人员根据情况需要决定字符串的分配方式。

2.6　函　　数

一个较大的程序一般都应分为若干个程序模块，每一个模块用来实现一个特定的功能，在程序设计中模块的功能是用子程序来实现。在 C 语言中，子程序的作用是由函数来完成的。一个 C 程序总是由主函数 main ( )开始，main ( )函数是一个控制程序流程的特殊函数，是程序的起点。主函数可以调用任何函数，函数之间也可以互相调用(不能调用主函数)，同一个函数可以被一个或多个函数调用任意多次，同一工程中的函数也可以分放在不同文件中一起编译。C 语言提供了丰富的运行库函数，对于一些需要经常使用的子程序，还可以设计成一个专门的函数库，供反复调用，有利于提高编程效率和质量。

2.6.1　库函数

在调用库函数时要用 include 命令包含相应的头文件，如调用数学库时，在源程序中应包含以下命令：

```
#include "math.h"
```

注意，文件包含并不是 C 语言中的语句，因此不能加“;”号，有关 include 头文件的用法，请参照“2.11.2 文件包含处理”一节。

函数的调用常用以下形式：

函数名 (参数表)

如：

```
printf ("a=%d",a);
```

在 printf 函数语句之后加了一个分号，就构成一条独立的函数调用语句。

ICCAVR 编译器提供了丰富的库函数，这些函数包括了常用的数学函数、对字符和字符串处理函数和各种输入输出函数等。详见“3.6 ICCAVR 6.26C 支持的库函数介绍”一节，读者只要根据需要，选用合适的库函数，正确地进行调用，就可以完成指定的操作或得到相应的结果。

2.6.2 函数的定义和返回值

1．函数定义

函数定义的一般形式：

```
函数返回值的类型名 函数名(类型名 形式参数 1,类型名 形式参数 2,…);
```

或者使用原版 C 语言的函数定义：

```
函数返回值的类型名 函数名(形式参数 1,形式参数 2,…)
```

形式参数类型说明;

注意：

(1) 目前常用的 C 编译器均支持以上两种函数定义形式，ICCAVR 6.26C 版也支持这两种形式。

(2) 函数名和形式参数都是用户命名的标识符，在同一工程中，函数名必须惟一；形式参数名只要在同一函数中惟一即可，可以与其他函数名中的变量同名。

(3) C 语言规定，不能在一个函数内部再定义函数。

(4) 在定义函数时应指明函数返回值的类型，若在函数的首部省略了函数返回的类型名，则默认函数返回值的类型为 int 型。

(5) 所定义的函数可以没有形参，没有语句，但是函数名后面的一对圆括号不能省略，如：

```
sum () {};
```

上例中定义了一个空函数，没有形参，没有需要执行的语句，该函数什么也不做，这是符合 C 语言的语法的。要说明：虽然该函数什么也不做，但是 C 程序调用该函数也还是有保护和恢复寄存器的操作，如果想在调用该函数时不再插入保护和恢复寄存器的代码，则应将该函数定义为 C 任务(ctask)函数。

[例 2.14] 编写求两个浮点数之和的函数.

```
float add(float a,float b)
      {
      float s;
      s=a+b;
      return s;
      }
```

2．函数的返回值

函数通过 return 语句返回，形式如下：

```
return (表达式);
```

通常，希望通过函数调用使主调函数能得到一个确定的值，这个值就是函数的返回值。对函数的返回值说明如下：

(1) 函数的返回值是通过函数中的 return 语句获得的。return 语句将被调用函数中的一个确定值带回主调函数中去。

(2) 如果不需要从被调用函数带回函数值时可以不要 return 语句，一个函数中也可以有一个以上的 return 语句，执行到哪一个 return 语句，哪一个语句起作用。

(3) 返回值的类型必须与函数首部所说明的类型一致，若类型不一致，则以函数值的类型为准，由系统自动进行转换。

(4) 为了明确表示“不带回值”，可以用“void”定义“空类型”，这样就可以保证函数不带回任何值。为了减少出错，保证函数的正确调用，凡是不要求有返回值的函数，都应该将其定义成 void 类型。由于主函数(main)一定不会有返回值，通常可以将主函数定义为以下形式：

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

2.6.3　函数的参数

在调用函数时，大多数情况下，主调函数和被调函数之间有数值传递关系。在定义函数时，函数名后面括号内的变量名称为“形式参数”，简称“形参”，在调用函数时，函数名后面括号中的表达式称为“实际参数”，简称“实参”，下面是关于形参与实参的说明。

1. 在定义函数时指定的形参变量，在定义时并没有在内存中开辟存储单元，只在函数被调用时才开辟存储单元，当退出函数时，这些临时开辟的存储单元全部释放掉。这些变量只在函数体内部起作用，与其他函数体中的变量不相关，因此它们可以和其他函数中的变量同名。

2. 实参可以是常量、变量或表达式，但是要求它们有确定的值。在调用时通常将实参的值赋给形参，但如果实参是数组名时，则传递的是数组的首地址而不是变量的值。用数组名作函数参数时，应该注意以下几点：

(1) 在主调函数和被调函数应分别定义数组，不能只在一个函数中定义而在另一个函数中不定义。

(2) 实参数组与形参数组类型应该一致，如不一致，运行结果将出错(编译器并不一定会报错)。

(3) 实参数组和形参数组大小可以一致，也可以不一致，C 编译器对形参数组大小不作检查，只是将实参数组的首地址传给形参数组，如果要求形参数组得到实参数组的全部元素值，则应该指定形参数组与实参数组的大小一致或者形参数组大于实参数组。

3．在被定义的函数中，必须指定形参的类型，如果没有形参，可以不用指定，但通常用“void”定义为“空类型”。

4．实参的个数与形参个数必须一致，在类型上按位置与形参一一对应匹配。如果类型不匹配，C 编译程序按赋值兼容的规则进行转换，如果实参和形参的类型不符合赋值兼容的原则，通常并不给出出错信息，而且程序一般能够执行，只是得到不正确的结果，因此应该特别注意实参和形参的类型匹配。

5．C 语言规定，实参变量对形参变量的数据传递是“值传递”，即单向传递，只能由实参传给形参，而不能由形参传回给实参。因此，在调用一个函数时，形参的值如果发生改变，并不会影响主调函数实参的值。原因是在内存中，形参和实参是不同的存储单元，在调用函数时，给形参分配存储单元，并将实参对应的值传递给形参，调用结束后，形参单元被释放，而实参单元仍保留原值。

6．实参允许是表达式。

请仔细观察下面程序的运行结果，确认函数实参与形参的传递方向。

[例 2.15]

```
void main (void)
     {
     int x=2,y=3,z=0;
     printf("(1)x=%d  y=%d  z=%d\n",x,y,z);
     try (x,y,z);                    //调用 rty 函数，由于没有返回值，所以没有特别说明
     printf("(4)x=%d  y=%d  z=%d\n",x,y,z);
     }
try(int x,int y,int z)
     {
     printf("(2)x=%d  y=%d  z=%d\n",x,y,z);
     z=x+y;
     x=x*x;
     y=y*y;
     printf("(3)x=%d  y=%d  z=%d\n",x,y,z);
     }
```

程序运行结果：

```
(1) x=2  y=3  z=0
(2) x=2  y=3  z=0
(3) x=4  y=9  z=5
(4) x=2  y=3  z=0
```

在上例中，实参的值传递给形参，但形参值的变化并没有影响对应的实参，可以说明

实参与形参之间是单向传递。

2.6.4　函数的调用

2.6.4.1　函数调用的方式

函数调用的一般形式为：

函数名（实参列表）

如果是调用无参数函数，则实参列表可以没有，但括号不能省略。

按函数在程序中出现的位置来分，可以有以下三种函数调用方式：

1．函数语句。把函数当作一个语句。如：

```
void lcd_init (void);                    //LCD 初始化函数
```

2．函数表达式。函数出现在一个表达式中，这时要求函数带回一个确定的值以参加表达式的运算。如：

```
c=2*max(a,b);
```

3．函数调用另一个函数作为实参。如：

```
c=max(a,max(b,c))
```

2.6.4.2　对被调用函数的说明

函数说明可以是一条独立的语句，如：

```
float add (float a,float b);
```

也可以与普通的变量一起出现在同一定义语句中。如：

```
float x,y, add (float,float);
```

一个函数调用另一个函数，必须符合以下条件：

1．被调用的函数是已存在的函数，如编译器的库函数或用户自己定义的函数。

2．如果使用函数是库函数，一般还应在文件开头用#include 命令将调用有关库函数时所需用到的信息包含到文件中。

3．如果使用用户自己定义的函数，即使该函数与调用它的函数(即主调函数)在同一文件中，一般在调用之前，还应该对被调函数的返回值的类型作说明。C 语言规定，在以下二种情况下，不在调用函数前对被调用函数作类型说明：

(1) 如果函数的返回值是整型或字符型，可以不必说明，系统对它们自动按整型返回值处理。

(2) 如果被调用函数的定义出现在主调函数之前，可以不必加以说明，因为编译系统

已经知道了已定义函数的类型，会自动处理的。

2.6.4.3 调用函数和被调用函数之间的数据传递

C 语言中，调用函数和被调用函数之间的数据传递可以通过以下三种方式：

1．通过全局变量。但这不是一种好的方式，不建议使用。

2．实际参数和形式参数之间进行数据传递。在 C 语言中，数据只能从实参单向传递给形参，对于不同类型的实参，有三种不同的参数传递方式：

(1) 当函数的参数是基本类型的变量时，主调函数将实际参数的值传递给被调函数中的形式参数，这种方式称为值传递，这种传递方式是单向传递。

(2) 数组类型的实际参数传递。当函数的参数是数组类型的变量时，主调函数将实际参数(数组)的起始地址传递到被调函数中形式参数的临时存储单元，这种传递方式称为地址传递，是一种双向传递。

(3) 指针类型的实际参数传递。当函数的参数是指针类型的变量时，主调函数将实际参数的地址传递给被调函数中形式参数的临时存储单元，因此已属于地址传递，也是一种双向传递。

3．通过 return 语句把函数值返回调用函数。

2.6.4.4 函数的嵌套调用和递归调用

C 语言的函数定义都是平行的、独立的，也就是说在定义函数时，在一个函数内不能定义另一个函数，但是可以在一个函数内调用另一个函数，也就是嵌套调用。

理论上，只要有足够的内存，函数的嵌套调用可以是不受限制的。ICCAVR 中，函数间的调用及数据保存与恢复是通过硬件堆栈和软件堆栈实现的。当没有使用外部数据存储器(External SRAM)，硬件堆栈和软件堆栈均在内部数据存储器中，当有外部数据存储器时，硬件堆栈在内部数据存储器中，软件堆栈则在外部的数据存储器中。在 ICCAVR 中，嵌套层数只受到硬件堆栈和软件堆栈的限制，如果嵌套层次太深，有可能导致硬件堆栈或软件堆栈溢出，其他的 C 编译器也是采用相类似的处理方式。

在调用一个函数的过程中又出现直接或间接地调用该函数本身，称为函数的递归调用，C 语言允许函数递归调用。

AVR 单片机适用的 ICCAVR、GCCAVR、IAR 等编译器均能够自动处理函数递归调用的问题，在递归调用时不必做任何的声明，调用深度仅受到堆栈大小的限制。Keil 编译器使用了特殊的内存覆盖技术，普通函数如果直接或间接递归调用，会导致以前的数据被覆盖而丢失，因此在 Keil 中，只有把某一函数定义为再入函数(函数名后加入 reentrant 关键字)，在调用时编译器将数据保存到模拟堆栈中，才能够正确地递归调用。

2.6.5　中断服务函数

keil 引入了一个扩充关键字 interrupt，用来定义一个函数为中断服务函数，定义形式为：

```
void 函数名() [interrupt n1] [using n2];
```

其中 n1 为中断向量号，n2 为选择寄存器组。

IAR 和 CodeVisionAVR 都扩充了“interrupt”关键字，用来定义一个函数为中断处理函数，定义形式为：

```
在 IAR 中：interrupt [TIMER1_OVF1_vect] void timer1_overflow(void)
在 CodeVisionAVR 中：interrupt [TIM1_OVF] void timer1_overflow(void)
```

在 interrupt 关键词后面方括号中的内容为中断向量号，只不过 IAR 和 CodeVisionAVR 在有关头文件中用不同的符号对同一个中断号进行了宏定义，实际上它们对应于同一个中断向量。

ICCAVR 中使用预处理命令#pragma interrupt_handler 来定义一个函数为中断处理函数，定义中断服务程序的形式为：

```
#pragma interrupt_handler 函数名 1:中断号 1 函数名 2:中断号 2
```

例如：

```
#pragma interrupt_handler timer1_ovf:7
```

在 ICCAVR 中定义中断服务时应注意以下几点：

1．中断向量号是从 1(复位向量)开始的，不同芯片的中断向量号是不同的，具体应用时应查相应芯片的使用手册。以 AT90S8515 为例，AT90S8515 有 12 个中断源，这 12 个中断源的中断向量入口地址如表 1.9 所示。

2．如果想多个中断入口使用同一中断服务程序，只要用不同的向量号多次声明就可以了。如计划定时器 1 和定时器 0 溢出共用 time_ovf 中断函数，可以定义如下：

```
#pragma interrupt_handler time_ovf:7 time_ovf:8
```

3．中断服务函数不能进行参数传递，如果中断函数中包含任何参数声明，都将导致编译出错。

4．中断函数没有返回值，如果企图定义一个返回值，大部分的编译程序均可以通过编译而不报错，但得不到正确的结果。

5．在任何情况下，都不能直接调用中断函数，否则会产生编译错误。因为普通函数是由 RET 指令退出，而中断函数是由 RETI 指令退出。如果没有实际中断操作请求的情况下调用中断函数，则编译器生成 RETI 指令退出函数会产生不可预料的错误。

在某些特殊情况下，要调用中断函数，应该通过设置(或改变)中断条件，引起中断而进入中断服务程序。

ICCAVR 支持用汇编语言编写中断操作函数。在汇编中，必须自行保存和恢复汇编中用到的全部寄存器，如果用汇编编写的中断函数中调用 C 函数，更要小心，要注意保存和恢复全部的可变寄存器(Volatile Registers，具体寄存器详见“2.12 在线汇编”一节)。在汇编中，必须使用“abs”属性来描述绝对区域，用“.org”来声明 rjmp 或 jmp 指令的正确地址，要注意“.org”声明使用的是字节地址。对全部非 ATmega 的 AVR 单片机，每个中断入口地址使用一个字(2 个字节)，中断声明如下：

```
.area vectors(abs)                    //中断向量
.org 0x6
rjmp _time                            //time 为 C 中断函数名，在汇编中名称要加上“_”前缀
```

对 ATmega 类的 AVR 单片机，每个中断入口地址使用二个字(4 个字节)，中断声明如下：

```
.area vectors(abs)                    //中断向量
.org 0xC
jmp _time                             //time 为 C 中断函数名，在汇编中名称要加上“_”前缀
```

在 GCCAVR 中，以使用 8515 芯片为例，中断向量定义在头文件 io8515.h 中，涉及的头文件还有 signal.h 和 intrrupt.h。中断函数的声明有两种方式，以外部中断 INT0 为例介绍如下：

```
#define SIG_INTERRUPT0 _VECTOR(1)     //定义中断向量
SIGNAL(SIG_INTERRUPT0)                //定义 INT0 中断服务程序，不允许中断嵌套
  {
  //中断服务程序
  }
INTERRUPT(SIG_INTERRUPT0)             //定义 INT0 中断服务程序，允许中断嵌套
  {
  //中断服务程序
  }
```

2.7 指　　针

指针是 C 语言中的一个重要的概念，也是 C 语言的一个重要特色，可以说不掌握指针，就掌握不了 C 语言的精华。正确而灵活地运用它，可以有效地表示和解决复杂的数据结构问题：能动态分配内存，能方便地使用字符串，能有效而方便地使用数组，能够用同一个

函数调用语句，而实现不同情况下分别调用不同的函数，还能直接处理内存地址等，而且使用指针也可以使得程序代码更简洁、效率更高。但是，若对指针使用不当，将使指针指向意料不到的地方，致使程序失控，严重的将导致系统崩溃。因此，正确掌握指针的概念、正确使用指针是十分重要的。

2.7.1　变量的指针和指向变量的指针变量

变量的指针指变量的地址，一个变量名实质上是代表“内存中的某个存储单元”。在程序中定义了一个变量，C 编译器就会根据定义的变量的类型，为其分配一定的存储空间(字符型 1 个字节，整型 2 个字节、长整型和浮点型 4 个字节)，变量的内存地址也就确定了。在一般情况下，我们只需指出变量名就可以对变量进行操作，实质上就是对某个地址的存储单元进行操作。这种操作无需指出变量在内存中的具体地址，变量名与具体地址的联系是由 C 编译器来完成的，这种按变量的地址存取变量的方式称为“直接存取”方式。

在 C 语言中，还可以定义一种特殊的变量，这种变量是用来存放内存地址的。假设我们定义了一个变量 a 和 p，a 在数据存储器中的地址为 0x100，p 在数据存储器中的地址为 0x70，若将变量 a 的地址存放到变量 p 中。这样，要访问变量 a，可以访问变量 a 的地址 0x100，或者先访问 p 的地址 0x70，从中取出 a 的地址 0x100，然后访问变量 a。后一种方式称为“间接存取”。在这种方式中，我们称变量 p 指向了变量 a，变量 a 是变量 p 所指向的对象。我们将一个变量的“地址”称为该变量的“指针”，那么用来存放指针(即地址)的变量就称作“指针变量”。所以“变量 p 指向了变量 a”的含义就是指针变量 p 中存放了变量 a 的地址。

2.7.2　指针变量的定义和指针变量的基类型

指针变量定义的一般形式如下：

```
类型名 *指针变量名 1,*指针变量名 2,…;
```

例：`int *p1,*p2,a,b;`

在上例中，定义两个指针变量 p1、p2 和两个整型变量 a、b，又定义 p1、p2 是指向整型变量的指针变量。在 p1 和 p2 的变量名前有一个星号(*)，表示该变量为指针变量，但指针变量名是 p1、p2，而不是*p1 和*p2，这与前面介绍的变量定义是不同的。在定义指针变量时应注意，变量名前的星号(*)不可省略，若省略了星号，就变成了把 p1、p2 定义为整型变量而不是指针型变量。类型名为 int(也称 int 是指针变量 p1 和 p2 的“基类型”)，说明 p1、p2 是两个指向整型变量的指针，也就是说变量 p1、p2 中只能存放 int 类型变量的地址。

一个指针变量中存放的是一个存储单元的地址值，这里的“一个存储单元”中的“一”所代表的字节数是不同的，当基类型为字符型，代表一个字节，当基类型为整型，代表二个字节，当基类型为长整型或浮点型则代表四个字节。当指针移动时(也就是要对地址进行

增减运算)，指针移动最小单位是一个存储单元(注意：不是一个字节)，对于不同基类型的指针变量，其指针(地址值)增减一个存储单元所“跨越”的字节数是不同的，因此基类型不同的指针变量不能混合使用。

在 Keil 中，使用以下方式定义指针变量：

数据类型 [存储器类型 1] * [存储器类型 2] 标识符;

数据类型：指该指针变量所指向的变量类型。

存储器类型 1：指针变量定义为基于存储器类型的指针，如果无此选项，则定义为通用指针。在 Keil 中，指向不同存储器种类的指针，存储该指针所需的空间是不同的，编译后生成的代码也是不同的。其中通用指针需要 3 个字节，编译后生成的代码量最大，运行速度最慢；指向 xdata 和 code 的指针占 2 个字节；指向 idata、data 和 pdata 的指针占 1 个字节，编译后生成的代码量最小，运算速度最快。但后两类指针需要指向一个确定的存储空间，缺乏灵活性和通用性。

存储器类型 2：用于指定指针变量本身的存储空间。

标识符：用户定义的指针变量名。

例如：

```
int xdata * srt;                    //指向 xdata 空间整型数据的指针
int * xdata srt;                    //位于 xdata 空间指向整型数据的通用指针
char data * xdata srt;              //位于 xdata 空间指向 data 空间字符型数据的指针
```

在 ICCAVR 中，根据指针所在的存储器类型和指针的目标所在的存储器类型的不同，可以分为以下四种，各种类型指针占用的存储空间是相同的。

char *ptr1：位于数据存储器空间指向数据存储器空间字符型数据的通用指针

const char *ptr1：指针 ptr1 位于数据存储器空间指向程序存储器空间的字符型数据

char *const ptr2：指针 ptr2 位于程序存储器空间指向数据存储器空间的字符型数据

const char *const ptr3：指针 ptr3 位于程序存储器空间指向程序存储器空间的字符型数据

2.7.3 对指针变量的操作

2.7.3.1 对指针变量赋值

在对指针变量操作前，应先给指针变量赋值。只有对指针变量赋值后，使指针变量获得一个确定的地址值，从而指向一个具体的对象，才能对指针变量进行操作。指针变量只能存放地址(指针)，不要将一个整型量(或任何其他非地址类型的数据)赋给一个指针变量。要给指针变量赋值通常用以下几种方式：

1．通过求地址运算(&)获得地址值

单目运算符“&”用来求出运算对象的地址，利用求地址运算可以把一个变量的地址赋给指针变量，如：

```
int k=1,*p1,*p2;
p1=&k;
```

赋值语句把 k 的地址赋予 p1，就可以说 p1 指向了变量 k。求地址运算符只能应用于变量，而且运算对象的类型必须与指针变量的基类型相同。

2．通过指针变量获得地址

可以通过赋值运算，把一个指针变量中的地址赋给另一个指针变量，从而使这两个指针变量指向同一地址。例如，若有上例中的定义，则语句：

```
p2=p1;
```

使指针 p2 也存放了 k 的地址，也就是说指针变量 p1 和 p2 都指向了变量 k。注意，在进行赋值运算时，赋值号两边的指针变量的基类型必须相同，而且不能写成*p2=*p1。

3．通过标准函数获得地址值

可以通过调用库函数 malloc 和 calloc 在内存中开辟动态存储单元，并把所开辟的动态存储单元的地址赋给指针变量。

```
p3=(struct student *)malloc(LEN);
```

除了可以给指针变量赋地址值外，还可以给指针变量赋空值(NULL)，如：

```
p1=NULL;
```

NULL 是在 ICCAVR 编译器中定义的预定义符(常用的编译器，如 GCCAVR 等，都有 NULL 预定义符，而含义和用法均相同)，由于 NULL 的为 0，当执行了以上赋值语句后，称 p1 为空指针。当然也还可以直接给一个指针赋整数 0，与给指针赋 NULL 等效，如：

```
p1=0;
```

或

```
p1='\0';
```

注意：

(1) p 的值为 0 与未对 p 赋值是两个不同的概念。前者是有值的(值为 0)，不指向任何变量，而未对 p 赋值并不等于 p 无值，只是它的值不确定，可能指向某一个事先并未确定的地址。

(2) 赋空值后，指针 p1 并不是指向地址为 0 的存储单元，而是值为“空”，企图通过一个空指针访问一个存储单元，将会得到一个出错信息。

2.7.3.2 通过指针来引用一个存储单元

C 语言中提供了一个称作“间接访问运算符”(也称指针运算符)的单目运算符“*”。当指针变量中存放了一个确定的地址值时，就可以用“间接访问运算符”来引用相对应的存储单元。

假定有以下语句：

```
int *p,i=10,j;
p=&i;                          //把变量 i 的地址赋予 p
j=*p;                          //把 p 所指的单元 i 的内容赋予变量 j
```

注意：这里*p 代表 p 所指的向的变量 i，等价于 j=i。此处的“*”不是乘号，也不是说明语句中用来说明指针变量的说明符，而是指针运算符。指针运算符必须出现在运算对象的左边，其运算对象应该是地址或者是存放地址的指针变量。

前面已经说过，在赋值语句 x=x 中，x 出现在赋值号左边和右边所代表的含义是不同的。当利用指针来引用一个存储单元时，也有同样的情况。在以下例句中：

```
int *p,k=0;
p=&k;                          //把变量 k 的地址赋予 p
*p=100;                        //把 100 赋予变量 k
*p=*p+1                        //取指针变量 p 所指向的存储单元的值，
                               //加 1 后再放入 p 所指向的存储单元
```

在上例中，*p 出现在赋值号的左边，代表的是指针所指的存储单元，*p 出现在赋值号右边时，代表的是指针所指向存储单元的内容。

*p=*p+1 也可以改为复合赋值表达式：

```
*p+=1;    或++*p;    或(*P)++;
```

[例 2.16] 用指针指向两个变量，通过指针运算选出值更小的那个数。

```
void main (void)
     {
     int a,b,min;
     int *pa,*pb,*pmin;
     pa=&a;pb=&b;pmin=&min;
     printf("a=%d  b=%d\n",a,b);
     scanf("%d%d",pa,pb);
     *pmin=*pa;
     if (*pa>*pb) *pmin=*pb;
```

```
    printf("min=%d\n",min);
    }
```

2.7.3.3　移动指针

所谓的移动指针变量是对指针变量加上或减去一个整数，或者通过赋值运算，使指针变量指向相邻的存储单元。因此，只有在指针指向一串连续的存储单元时，指针移动才有意义。

当指针指向一串连续的存储单元时，可以对指针变量加上或减去一个整数的运算，也可以对指向同一串连续的存储单元的两个指针进行相减运算；除此以外，不可以对指针进行任何其他的算术运算。

```
int *p,*q,a[5]={0,1,2,3,4,5};
p=a;                          //数组 a 的首地址赋给指针变量 p
q=p;                          //q 也指向数组 a 的首地址
                              //赋值完成后，就可以对指针进行移动
p=q+2;                        //使指针变量 p 指向 a[2]
q++;                          //向高地址移动指针，使 q 指向 a[1]
p++;                          //向高地址移动指针，使 p 指向 a[3]
q--;                          //向低地址移动指针，使指针变量 q 指向 a[0]
p--;                          //向低地址移动指针，使指针变量 p 指向 a[3]
```

注意：在对指针进行加、减运算中，数字“1”不再代表十进制整数“1”，而是指向 1 个存储单元的长度。至于 1 个长度占多少存储空间，则视指针的基类型而定。

在编程时只要注意让指针的基类型和变量的类型保持一致就可以了，在程序中需要移动指针时，无论指针的基类型是什么，只需简单加、减一个整数，而不必去管它们移动的具体长度，系统会根据指针的基类型自动确定移动的字节数。

2.7.3.4　指针比较

在关系表达式中可以对两个指针进行比较。如下例是符合 C 语言的语法的：

```
int *p,*q,a[5]={0,1,2,3,4,5};
p=a;                          //数组 a 的首地址赋给指针变量 p
q=p;                          //q 也指向数组 a 的首地址
                              //用户程序，在程序中移动指针 p 和 q
if (q>p) a[0]=100;
if (q<=p) a[1]=100;
```

通常两个或多个指针指向同一目标(如数组或一串连续的存储单元)，比较才有意义。

2.7.4 数组的指针和指向数组的指针变量

一个数组包含若干个元素，每个数组元素都在内存中占有一个或一个以上的存储单元，因此每一个元素都有相应的地址。数组的指针就是指数组的起始地址，把数组起始地址(或某一元素的地址)放到一个指针变量中，使指针变量指向该数组(或数组元素)，就称该指针为指向数组的指针变量。

引用数组元素可以用下标法(如 a[2])，也可以使用指针法，即通过指向数组元素的指针找到所需的元素。使用指针法能使生成的目标程序质量更高(占内存少，运行速度快)。

2.7.4.1 指向数组的指针变量的定义与赋值

这种指针变量的定义与前面介绍指向变量的指针变量相同。例如：

```
int *p,a[10];              //定义 a 为包含 10 个整型数据的数组，p 为指向整型变量的指针
p=&a[0];                   //把 a[0]元素的地址赋给指针变量 p
```

C 语言规定数组名代表数组的首地址，也就是第一个元素的地址，因此也可以采用下面的赋值语句：

```
p=a;
```

要注意，上述语句的用处是将数组 a 的首地址(即 a[0]的地址)赋给指针变量 p(注意，不能写成*p)，而不是把整个数组赋值给指针变量。

在定义指针变量时也可以赋初值，例如：

```
int a[10];
int *p=&a[0];
```

或者为：

```
int a[10];
int *p=a;
```

2.7.4.2 通过指针引用数组元素

1．通过指针引用数组元素

如果指针 p 指向一个一维数组 a[10]，并且 p 已赋初值&a[0]，可以使用以下三种方式引用数组元素：

(1) 下标法

p+i 和 a+i，就是 a[i]，或者说它们都是指向 a 数组的第 i 个元素。

[例 2.17] 用下标法输出数组元素。

```
void main (void)
     {
     int a[10];
     int i;
     printf("input 10 integer");
     for (i=0;i<10;i++)
          scanf ("%d",&a[i]);
     printf("\n");
     for (i=0;i<10;i++)
          printf("%d",a[i]);
     }
```

(2) 地址法

(p+i)和(a+i)也就是 a[i]。实际上，编译器对数组元素 a[i]就是处理成*(a+i)，即按数组的首地址加上相对位移量得到要找元素的地址，然后找出该单元中的内容。

[例 2.18] 用地址法输出数组元素。

```
void main (void)
     {
     int a[10];
     int i;
     printf("input 10 integer");
     for (i=0;i<10;i++)
          scanf ("%d",&a[i]);
     printf("\n");
     for (i=0;i<10;i++)
          printf("%d",*(a+i));
          }
```

(3) 指针法

指向数组的指针变量也可以带下标，如 p[i]，与*(p+i)等价。

[例 2.19] 用指针法输出数组元素。

```
void main (void)
     {
     int a[10];
     int *p,i;
     printf("input 10 integer");
     for (i=0;i<10;i++)
          scanf ("%d",&a[i]);
```

```
    printf("\n");
    for (p=a;p<(a+10);p++)
        printf("%d",*p);
    }
```

以上三种方式均可引用数组元素，有如下特点：

(1) 下标法和地址法执行效率相同。C 编译系统是将 a[i]转换为*(a+i)处理的，即先计算元素的地址。因此用第 1 种和第 2 种方法查找数组元素费时较多。

(2) 指针法比前两种方法快。因为用指针变量直接指向元素，不必每次都重新计算地址，在 C 中，像 p++这样的自加操作是比较快的，这种有规律地改变地址值(p++)的方法能大大提高执行效率。

(3) 用下标法比较直观，能直接知道是第几个元素，如 a[5]就是数组中第 6 个元素。用地址法和指针法不直观，不容易很快判断出当前正在处理的是哪一个元素。

2．在使用指针变量时，要注意以下几个问题：

(1) 指针变量可以使本身的值改变，如 p++使 p 值不断改变，分别指向数组中的不同元素。而使用 a++是不行的，因为 a 是数组名，它是数组的首地址，a 的值在程序运行期间是固定不变的，是常量，不能用自加或自减号。

(2) 指针变量指向一个数组，在使用指针变量时可能会指到数组之外，如定义一个数组 a[10]，在使用指针变量时，误使指针为(p+10)，也就是指向 a[10]这个并不存在的数组元素，但编译器并不会报错，在程序运行时可能会出现无法预料的错误。

(3) 在使用指针变量的自加(++)和自减(--)运算时要小心，很容易弄错，在编程时应该引起足够的注意。如果先使 p 指向数组 a(即 p=a)，则：

- p++(或 p+=1)，使 p 指向下一元素，即 a[1]，接着执行*p，就可以取出下一个元素 a[1]的值。
- *p++，由于++和*同优先级，是自右而左的结合方向，因此它等价于*(p++)。作用是先得到 p 指向的变量的值(即*p)，然后再使 p=p+1。
- *(p++)与*(++p)作用不同，前者是先取*p 的值，后使 p 加 1。后者是先使 p 加 1，再取*p 的值。若 p 初值为 a(即&a[0])，使用*(p++)表达式时，得 a[0]的值，而使用*(++p)表达式时，则得到 a[1]的值。
- (*p)++表示 p 所指向的元素加 1，即(a[0]++，如果 a[0]=3，则(*p)++后，a[0]=4。注意：是元素加 1，而不是指针值加 1。

2.7.5 字符串指针和指向字符串的指针变量

在 C 程序中，可以用两种方法实现字符串运算，一种是使用字符数组来实现，另一种可用字符串指针实现，分别如下：

```
char string[]= {'I','C','C','A','V','R','\0'};
char *string="ICCAVR";
```

由于 C 语言对字符串常量是按字符数组处理的，实际在使用字符串指针时，C 编译器也在内存中开辟了一个字符数组用来存放字符串常量。我们在上面定义了一个字符指针变量 string，并把字符串首地址赋给指针变量，即存放 ICCAVR 字符串的字符数组的首地址赋给 string 指针变量。对于字符串指针，我们不能理解成 string 是一个字符串变量，并且把字符串"ICCAVR"赋给该字符串变量 string。

对于一个数值型数组，是不能用数组名输出它的全部元素的，只能逐个元素的输入输出。对于字符数组和字符串数组，既可以逐个元素的输入输出，也可以对整个字符数组或字符串进行整体输入输出，如上面两个定义均可以用以下语句输出整个数组中的全部元素：

```
printf ("%s",string);
```

虽然字符指针变量与字符数组都能够实现字符串的存储和运算，但它们之间是有区别的：

1．字符数组由若干个元素组成，每个元素中放一个字符，而字符指针变量中存放的是地址(字符串的首地址)，决不是将字符串放到字符指针变量中。

2．赋初值的方式不同。如：

```
char *p="ICCAVR";
```

等价于：

```
char *p;
p="ICCAVR";
```

而对数组初始化时：

```
char str[10]={"ICCAVR"};
```

就不能等价于：

```
char str[10];
str[]="ICCAVR";
```

即数组可以在变量定义时整体赋初值，但决不能在赋值语句中整体赋值。

3．赋值方式不同。对字符数组只能对各个元素赋值，而对字符指针变量，可以整体赋值。

4．在定义一个数组时，编译时即已分配内存单元，有确定的地址。而定义一个字符指针变量时，给指针变量分配内存单元，该指针变量在使用过程中可以指向任何一个字符型数据，但在编译时并没有对它赋一个地址值，该指针变量在编译时并没有具体指向哪一个字符数据。也就是说，一个指针变量在编译时并不能够明确的知道它所指向的字符串的具体位置，需在运行过程中才能确定。

5. 指针变量的值在程序运行过程中是可以改变的，而数组的值在程序的运行过程中是一个常数，不能改变。

2.7.6 函数的指针和指向函数的指针变量

一个函数在编译时分配了一个入口地址，这个入口地址就称为函数的指针，可以用指针变量指向这个函数，然后通过该指针变量调用此函数。

用以下形式说明指向函数的指针变量：

```
int (*p) ();                    //一个指向函数的指针变量，此函数带回整型的返回值
```

由于*p 两侧有括号，p 先与*结合，是指针变量，然后再与后面的( )结合，表示此指针变量是指向函数，这个函数的返回值是整型。如果在书写时省略了*p 两侧的括号，写为如下形式：

```
int *p ();
```

则变成说明这是一个函数，这个函数的返回值是指向整型变量的指针。

上面的语句只是说明在本文件中定义了这样一个类型的变量，它是专门用来存放函数的入口地址变量，在定义后该变量还没有指向任何一个函数，在程序中把哪一个函数的入口地址赋给它，它就指向哪一个函数。在一个程序中，一个指针变量可以先后指向不同的函数。如果使用以下赋值语句：

```
p=max;                          //max 为一个函数的函数名
```

在 C 语言中，函数名和数组名的表示方法是一样，对于函数也是用函数名代表该函数的入口地址。在上面的赋值语句中，将函数 max 的入口地址赋给指针变量 p，这时，p 就是指向函数 max 的指针变量，也就是 p 和 max 都是指向该函数的起始位置，在程序中出现*p 就是调用函数 max。

注意：p 是指向函数的指针变量，它只能指向函数的入口地址，而不可能指向函数中的某一条指令，因此函数的指针和数组不一样，不能用*(p+1)来表示函数的下一条指令。

[例 2.20] 设一个函数 process，在调用它的时候，每次要实现不同的功能。输入 a 和 b 两个数，第一次调用 process 时，要找出 a 和 b 中大者，第二次找出其中小者，第三次求 a 与 b 之和。

```
void main (void)
     {
     int max(),min(),add(),a,b;
     printf("enter a and b:");
     scanf ("%d , %d",&a,&b);
     printf("max=");
```

```
    process(a,b,max);
    printf("min=");
    process(a,b,min);
    printf("sum=");
    process(a,b,add);
    }
max(int x,int y)
    {return (x>y?x:y);}
min(int x,int y)
    {return (x<y?x:y);}
add(int x,int y)
    {return (x+y);}
process(int x,int y,int (*fun)())
    {printf("%d\n",(*fun)(x,y));}
```

在上例中，max、min 和 add 是三个已定义的函数，分别用来实现求大数、求小数和求和的功能。在 main 函数中第一次调用 process 函数时，除了将 a 和 b 作为实参传给 process 的形参 x、y 外，还将函数 max 作为实参将其入口地址传给 process 函数中的形参(指向函数的指针变量 fun)，这时 process 函数中的(*fun)(x,y)就相当于 max(x,y)，执行 process 可以输出 a 和 b 中大的数；改用函数名 min 作实参，此时 process 函数的形参 fun 指向函数 min，这时 process 函数中的(*fun)(x,y)就相当于 min(x,y)；在第三次调用 process 函数时也相同。

从上例中，不论执行 max、min 和 add，函数 process 一点都没有改动，只是在调用 process 函数时将实参中的函数名改变而已，这就增加了函数使用的灵活性，可以编写一个通用函数来实现各种专用功能。

2.7.7　指针数组和指向指针的指针

一个数组，其元素均为指针类型数据，称为指针数组，也就是说，指针数组中的每一个元素都是指针变量。指针数组的定义形式为：

类型标识符　*数组名[数组长度说明]

例如：

```
int *p[10]
```

由于[ ]的优先级比*高，因此 p 先与[10]结合，形成 p[10]形式，这显然是数组形式，它有 10 个元素，然后再与 p 前面的“*”结合，表示此数组是指针类型的，数组的每个元素(指针变量)都可以指向一个整型变量。注意：不能写成 int (*p)[10]，这是指向一维数组的指针变量。使用指针数组比较适合于用来指向若干个字符串，使字符串处理更方便灵活。

[例 2.21] 定义一个若干字符串并输出。

```
void main (void)
     {
     int i;
     char *name[]={"http://www.sl.com.cn","http://www.imagecraft.com","
                   http://avr.51.net"};
     for (i=0;i<3;i++)
          printf("%s\n",name[i]);
     }
```

在这个简单的例子中，由于三个字符串的长度不相同，如果使用普通的一个数组来保存，会造成存储空间的浪费，特别是在字符串长短相差较大且字符串数量很大时，浪费的存储空间和数据空间更是惊人。而使用指针数组，使数组中的各指针指向长短不同的字符串，这样长短不同的字符串既可以存储在一片连续的空间中，也可存储在不连续的空间中。特别是用指针数组在处理字符串组的排序时，具有很大的优势，因为移动指针变量的值(地址)要比移动字符串所用的时间少得多。

在 ICCAVR、GCCAVR 以及其他常用的编译器中，指针数组中元素的数量只受到可以使用内存的限制，而在 CVAVR 中，指针数组至多只能有 8 个元素。

一个指针变量，不但可以指向变量，而且也可以指向一个指向变量的指针变量。一般的定义形式如下：

```
char **p;
```

在 p 前面有两个星号(*)，由于*号运算符的结合性是从右到左，因此**p 相当于*(*p)，显然*p 是指针变量的定义形式，在括号内的*p 就是定义了一个指向字符型数据的指针变量，在它的前面又加一个“*”号，表示指针变量是指向一个字符型指针变量(即指向字符型数据的指针变量)。

2.7.8 有关指针的数据类型和指针运算的小结

前面已经讨论了有关指针的数据类型和指针的运算，为使读者有一个系统而完整的概念，下面作一个小结。

1．有关指针的数据类型

表 2.7 是有关指针的数据类型小结，为了便于比较，我们把一些其他的类型定义也列在一起。

表 2.7 常用数据类型定义

定　义	含　义
int i;	定义整型变量 i
int *p;	p 为指向整型数据的指针变量

续表

定　义	含　义
int a[n];	定义整型数组 a，它有 n 个元素
int *[p];	定义指针数组 p，它由 n 个指向整型数据的指针元素组成
int (*p)[n];	p 为指向含 n 个元素的一维数组的指针变量
int f( );	f 为带回整型函数值的函数
int *p( );	p 为带回一个指针的函数，该指针指向整型数据
int (*p)( );	p 为指向函数的指针，该函数返回一个整型值
int **p;	p 是一个指针变量，它指向一个指向整型数据的指针变量

2．指针运算小结

前面已经用过一些指针运算(如 p++，p+i 等)，现把全部的指针运算整理如下。

(1) 可以给指针变量加(减)一个整数，如：

```
p++、p--、p+i、p-i、p+=i、p-=i 等。
```

(2) 指针变量赋值

将一个变量地址赋给一个指针变量，如：

```
p=&a;                    //将变量 a 的地址赋给 p
p=array;                 //将数组 array 首地址赋给 p
p=&array[i];             //将数组 array 第 i 个元素的地址赋给 p
p=max;                   //max 为已定义的函数，将 max 的入口地址赋给 p
p1=p2                    //p1 和 p2 都是指针变量，将 p2 的值赋给 p1
```

注意，不能把一个整数赋给指针变量，也不能把指针变量的值(地址)赋给一个整型变量。

(3) 指针变量可以有空值，即该指针变量不指向任何变量，可以这样表示：

```
p=NULL;
```

它等效于：

```
p=0;
```

或

```
p='\0';
```

上述语句均使 p 指向地址为 0 的单元。

(4) 在引用指针变量之前应先对它赋值。

(5) 两个指针变量可以比较或进行相减运算

如果两个指针变量指向同一个数组元素，则可以进行比较或相减运算，如果两个指针变量不指向同一数组，则比较和相减运算无意义。

3．抽象类型指针类型

抽象指针类型(void *)是 ANSI C 新标准增加的，即在定义一个指针变量，可以不明确指定它是指向哪一种类型数据。ANSI C 新标准规定调用动态存储分配函数时应返回 void 指针，它可以用来指向一个抽象的类型数据，可以将它的值直接赋给另一个抽象类型的指针，如要将一个抽象类型的指针的值赋给另一种类型的指针变量时，要进行强制类型转换，使之适合于被赋值的变量的类型。如：

```
char *p1;
void *p2;
//用户程序，省略
p1=(char *)p2;
```

在 ICCAVR 6.26C 版的 C 编译器也支持新增的“void *”指针类型。

2.8 结构体与共用体

在 C 语言中，用户可以构造以下的四种数据类型：

1．结构体，把具有相互关系不同类型的数据组成一个有机的整体。

2．共用体，又称联合体，使几种不同类型的变量共用一段存储空间。

3．枚举类型，将变量的值一一列举出来，变量的值只限于列举出来的范围。

4．用户定义类型，对已有的类型，另说明一个新的类型标识符。

下面我们将分别讲述。

2.8.1 定义结构体类型变量的方法

结构体是一种比较复杂但却非常灵活的构造型数据类型。一个结构体类型可以由若干个称为成员(或称为域)的成分组成，不同的结构体类型可根据需要由不同的成员组成。对某个具体的结构体类型，成员必须固定，这一点与数组相同，但结构中各个成员的类型可以不同，这是结构体与数组的重要区别。因此，当需要把一些相关信息组合在一起时，采用结构体这种类型就很方便。

一般用以下形式定义一个结构体：

```
struct 结构体名
{成员列表};
```

其中 struct 是关键字，不能省略，大括号内是该结构体中各成员列表，成员名的命名规则与变量名相同，对列表中的各成员都应进行类型说明，具体说明方式如下：

类型标识符 成员名

例如，定义一个 student 类型的结构体：

```
struct student
       {
       int num;
       char name[20];
       char sex;
       int age;
       float score;
       char addr[30];
       };
```

要注意，上面仅仅是定义一个结构体类型！要定义一个结构体类型的变量，可以采用以下三种方法。

1．先定义结构体类型再定义变量名

如上面已定义了一个结构体类型 struct student，可以用它来定义变量，如：

```
struct student student1,student2;
```

上述语句定义 student1 和 student2 为 struct student 类型的变量，即它们具有 struct student 类型的结构。

说明：将一个变量定义为基本数据类型与定义为结构体类型的不同之处是，结构体不仅要求指定变量为结构体类型，而且要求指定为某一特定的结构体类型(如 struct student)，不能只指定为“struct”而不指定结构体名。而在定义变量为整型时，只需指定为 int 类型即可。

为了使用方便，通常用一个符号常量代表一个结构体类型，如在程序开头：

```
#define STUDENT  struct student
```

这样在程序中，STUDENT 与 struct student 完全等效，可以这样定义结构体：

```
STUDENT
    {
    int num;
    char name[20];
    char sex;
    int age;
    float score;
    char addr[30];
    };
```

定义之后就可以直接用 STUDENT 定义变量。如：

```
STUDENT student1,student2;
```

2．在定义类型的同时定义变量，定义形式如下：

```
struct 结构体名
{成员列表}
变量名列表;
```

例如：

```
struct student
       {
       int num;
       char name[20];
       char sex;
       int age;
       float score;
       char addr[30];
       } student1,studen2;
```

它的作用与前面定义的相同，即定义了两个 struct student 类型的 student1、studen2。

3．直接定义结构体类型变量

```
struct
{成员列表}
变量名列表;
```

即不出现结构体名，如下例：

```
struct
      {
      int num;
      char name[20];
      char sex;
      int age;
      float score;
      char addr[30];
      } student1,studen2;
```

关于结构体类型，有几点要说明：

(1) 结构体类型与结构体变量是不同的概念，不要混同。在定义一个结构体类型时，只是给出该结构的组织形式，并没有给出具体的组织成员，因此结构名不占用任何存储空间，也不能对结构名进行赋值、存取和运算。而结构体变量则是一个结构中的具体组织成员，编译器会给该结构体变量分配确定的存储空间，因此可以对结构体变量名进行赋值、存取和运算。

(2) 对结构体的成员(即“域”)可以单独使用，它的作用与地位相当于普通变量。

(3) 一个结构中的结构元素还可以是另一个结构类型的变量，即可以形成结构的嵌套。如：

```
struct date
    {
    int month;
    int day;
    int year;
    };
struct student
    {
    int num;
    char name[20];
    char sex;
    int age;
    struct date  birthday;
    char addr[30];
    } student1,studen2;
```

先定义一个 struct date 类型，它代表“日期”，包括三个成员：month、day、year。然后，在定义 struct student 类型时，成员 birthday 的类型定义为 struct date 类型。ANSI C 标准规定结构体至少允许嵌套 15 层，并允许内嵌结构体成员的名字与外层成员的名字相同，ICCAVR 6.26 版完全符合 ANSI C 的规定。

(4) 结构体中的成员名可以和程序中其他变量名同名，不同结构体中的成员名也可以同名。如：在程序中还可以再定义变量 num、age 等，它与 struct student 中的 num、age 是两个不同的变量，互不相干。

2.8.2　结构体变量的初始化

结构体变量在定义时就可以初始化，如下：

```
struct prople
    {
```

```
    char name[20];
    char sex;
    char mobile[13];
    char tel[13];
    char. www[30]
    char mail[30];
    } people1={"Shen Wen",'M',"013905998618","05995886602","http://avr.51.net",
              "fjmcu@public.npptt.fj.cn"};      //后面这个分号不要忘了
```

在上面的结构体中，由于手机号(mobile)和电话(tel)号是特殊的数字序列，一般不参与数学运算，因此都将这样的数字序列定义为字符型数组。在赋初值时要注意，由于定义了 char sex，因此对应的赋值为'M'(用单引号，代表字符型)，只能用“%c”格式符输出；定义数组 char name[20]，则对应的赋值为"Shen Wen"(用双引号，代表字符串)，输出格式符为“%s”；不要弄错了。

对结构体变量赋初值时，C 编译程序按每个成员在结构体中的顺序一一对应赋初值；不允许跳过前边的成员给后面的成员赋初值，但可以只给前面的若干成员赋初值，对于后面未赋初值的成员，如果为数值型和字符型，系统自动赋初值零。

2.8.3 结构体类型变量的引用

在定义了结构体变量以后，可以引用这个变量，但应遵循以下规则：

1．不能将一个结构体变量作为一个整体进行输入和输出，只能对结构体变量名中的各成员分别输出，引用方式为：

结构体变量名.成员名

“.”是成员(分量)运算符，它在所有的运算符中优先级最高，因此在下例中：

```
prople.sex='M';
```

将字符“M”赋值给 prople 结构体变量中的成员 sex。

结构体变量可以直接赋值给另一个具有相同结构的结构体变量。

```
student2=studen1;
```

2．如果需访问结构体变量中各内嵌结构体成员时，必须逐层使用成员名定位，只能对最低级的成员进行赋值或存取以及运算。如上例中定义的结构体变量 student1，可以这样访问各成员：

```
student1.num
student1.name
student1.birthday.month
```

```
student1.birthday.day
student1.birthday.year
```

注意：不能用 student1.birthday 来访问 student1 变量中的成员 birthday，因为 birthday 本身也是一个结构体变量。对于多层嵌套的结构体，引用方式与此类似，即按照从最外层到最内层的顺序逐层引用，每层之间用点号隔开。

3．若结构体中的成员是作为字符串使用的字符型数组，可以将其看作“字符串变量”来使用。如：

```
printf("%s", student1.name);
```

4．对结构体的成员变量可以像普通变量一样进行各种运算(根据其类型决定可以进行的运算)。如：

```
student2.scor=student1.scor;
sun=student1.scor+student2.scor;
student1.age++
```

5．可以引用成员的地址，也可以引用结构体变量的地址(结构体名就是结构体变量的地址)，主要用作函数参数，传递结构体的地址。

2.8.4　定义一个结构体数组

一个结构体变量中可以存放一组相关的数据，如果有多组数据(多个结构体)要参加运算，显然就应该用数组，这就是结构体数组。结构体数组与前面介绍过的数值型数组不同之处在于数组的每一个元素都是一个结构体类型的数据，它们又分别包括各个成员(分量)项。

结构体数组的定义与定义结构体变量的方法相类似，只需说明其为数组即可，如：

```
struct student
       {
       int num;
       char name[20];
       char sex;
       int age;
       float score;
       char addr[30];
       };
struct student stu[3];
```

以上定义了一个数组 stu，其元素为 struct student 类型数据，该数组共有 3 个元素。也

可以直接定义一个结构体数组，如：

```
struct student
       {
       int num;
       char name[20];
       char sex;
       int age;
       float score;
       char addr[30];
       } stu[3];
```

结构体数组中各元素在内存中连续存放。

2.8.5 指向结构体类型数据的指针

一个结构体变量的指针就是该变量所占据内存段的起始地址。可以设计一个指针变量，用来指向一个结构体变量，此时该指针变量的值是结构体变量的起始地址，这样用指针变量也可以指向结构体数组中各成员。

2.8.5.1 指向结构体变量的指针

```
struct student
      {
      int num;
      char name[20];
      char sex;
      };
struct student stu;
struct student *p;
p=&stu;                              //将结构体 stu 的首地址赋给指针变量 p
```

在上例中定义了一个 struct student 类型的结构体，然后又定义了一个 struct student 类型的变量 stu，还定义了一个指向 struct student 类型结构体的指针变量 p，在执行过程中又将 stu 的首地址赋给指针变量，也就是使 p 指向 stu。这样要引用结构体中的成员 num 可以用 stu.num 或(*p).num。注意：*p 两侧的括号不可省略，因为成员运算符“.”优先于“*”运算符，如果没有括号就等价于*(p.num)了。在 C 语言中，为了使用方便和直观，可以把(*p).num 改用 p->num 来表示(“->”称为指向运算符)。也就是说，以下三种形式等价：

1. stu.num

2．(*p).num

3．p->num

2.8.5.2　指向结构体数组的指针

```
struct student
      {
      int num;
      char name[20];
      char sex;
      };
struct student stu[3];
struct student *p;
p=&stu;
```

在上例中定义了一个 struct student 类型的结构体，然后又定义了一个 struct student 类型的数组 stu[]，它有 3 个元素，还定义了一个指针变量 p，指向一个 struct student 类型的结构体。在执行过程中又将 stu 的首地址赋给指针变量，也就是使 p 指向数组 stu。这样执行 p++，使 p 自加 1，p+1 意味着增加的地址值为结构体类型数组 stu 中的一个元素所占的字节数，执行后使 p 指向 stu[1]的起始地址。因此，p+1 后并不是指向 stu[0]中的某个成员，而是指向数组中的下一个元素。要注意下面两种表达方法的区别：

```
(++p)->num          //先使 p 自加 1，然后得到它指向的元素中的 num 成员值
(p++)->num          //先得到 p->num 的值，然后使 p 自加 1，指向 stu[1]
```

2.8.5.3　用指向结构体的指针作函数参数

有时想将一个结构体变量的值传递给另一个函数，有三种方法：

1．用结构体变量的成员作参数。例如用 stu[1].num 作实参，将实参值传给形参，用法和普通变量作实参是一样的，属“值传递”方式。

2．用结构体作为函数的参数。在这种方式中，必须保证实参与形参的类型相同，也是属于“值传递”。把一个完整的结构体变量作参数传递，并一一对应传递各成员的数据，为了在函数返回后取用，系统同时要保存结构体参数中所有成员的当前值而进行一系列的内部操作(在单片机中，这些操作是通过压栈和出栈来实现的)，这些操作将增加系统的处理时间，会影响程序的执行效率，还需要较大的数据存储空间(堆栈)。

结构体变量作实参时，传递给函数对应形参的是它的值，函数体内对形参结构体变量中任何成员的操作，都不会影响对应实参中成员的值。从而保证调用函数中数据的安全，但这也限制了将运算结果返回给调用函数。

3．用指向结构体变量(或结构体数组)的指针作实参，将结构体变量(或结构体数组)的

地址传给形参。系统只需为形参指针开辟一个存储单元存放实参结构体变量的地址值，而不必另行建立一个结构体变量。这样既可以减少系统操作所需的时间，提高程序的执行效率，又可以通过函数调用，有效地修改结构体中成员的值。

2.8.6 用指针处理链表

2.8.6.1 链表概述

利用数组来存储“批量”的数据时，定义数组必须指明(显式或隐含)元素的个数，从而也就限定了能够在一个数组中存放数据的量。在实际应用中，一个程序每次运行时要处理数据的量通常并不确定，如果数组定义得小了，就没有足够的空间存放数据，定义得大了，又会浪费存储空间。对于这种情况，如果能够在程序执行过程中，根据需要随时开辟存储单元，不需要时随时释放，就能比较合理地利用存储空间。C 语言的动态存储分配提供了这种可能。但每次动态分配的存储单元，其地址不一定是连续的，而所需处理的批量数据往往是一个整体，各数据之间存着连接关系，如果利用链表这样的存储结构就完全可以反映出数据之间的相互联系。在这种链表的每个结点中，除了要存放数据本身的数据域外，至少还需要有一个指针域，用它来存放下一个结点元素的地址，以便通过这些指针把各结点连接起来，从而形成链表。由于链表中的每个存储单元都由动态存储分配获得，故称这样的链表为“动态链表”(简称链表)。

在链表的结构中，有一个“头指针”变量来指向链表的开始，存放链表中第一个结点的地址。链表中每一个元素都称为“结点”，每个结点都应该包括两部分。一部分为用户需要的实际数据，另一部分为下一个结点的地址；最后一个元素称为“表尾”，不再指向其他单元，它的地址部分放一个"NULL"符号，表示“空地址”，链表至此结束。从链表的结构中可以看出，链表中各元素在内存中可以不是连续存放的。在这种动态链表中，每个结点元素并没有自己的名字，只能靠指针维系结点元素之间的接续关系，要找某一元素，必须先找到上一个元素，根据它提供的下一元素地址才能找到下一元素，如果不提供“头指针”或链表中的某一元素指向下一元素的指针(地址)丢失，则后面的元素将无法访问。这种链表只能从当前结点找到后继结点，故称为“单向链表”。

在下例中：

```
struct student
       { int num;
       float score;
       struct student *next;                    //next 是成员名，它是指针类型
                                                //指向 struct student 类型数据的指针
};
```

每一个结点都属于 struct student 类型，它的成员 next 存放下一结点的地址，程序设计人员可以不必具体知道地址值，只要保证将下一个结点的地址放到前一结点的成员 next 中即可。

键表结构是动态分配存储单元的，只在需要时才开辟存储单元，请注意：上面只是定义一个 struct student 类型，并未实际分配存储空间。在 C 语言编译系统的库函数中提供以下有关函数：

1．malloc(size)。在内存的动态存储区中分配一个长度为 size 的连续空间。在此函数的返回值是一个指针，它的值是分配存储区域的起始地址。如果此函数执行不成功，则返回空指针(也就是整数 0)。由调用 malloc 函数所分配的存储单元，系统将各单元置初值 0。

2．calloc(n,size)。在内存的动态区存储中分配 n 个长度为 size 的连续空间。函数返回分配域的起始地址。如果函数执行不成功，则返回空指针(也就是整数 0)。由调用 calloc 函数所分配的存储单元，系统将各单元置初值 0

3．free(*prt)。释放由 prt 指向的内存区。

在使用上面三个函数时，必须在文件开头包含头文件 stdlib.h。参数 n 和 size 的类型为 unsigned int，在旧的 ANSI C 中，参数 prt 为字符型指针，新的 ANSI C 标准将 prt 改为 void 类型的指针(空指针)，函数 malloc 和 calloc 返回值也为空指针。ICCAVR 6.26C 版提供的 malloc、calloc 和 free 三个函数完全符合 ANSI C 新标准。

2.8.6.2　建立链表

所谓建立链表是指从无到有地建起一个链表，即一个一个地输入各结点数据，并建立起前后相链接的关系。下面通过一个例子来说明如何建立一个链表。

[例 2.22] 建立键表函数，如果 num=0，则表示建立链表完成，该结点不连接到链表中。

```
#define NULL 0
#define LEN sizeof (struct student)          //求字节运算符 sizeof 计算出
                                               struct student 的字节数

struct student
       {
       long num;
       float score;
       struct student *next;
       };
int n;                                       //n 为结点的个数
struct student *creat ()                     //此函数带回一个指向链表头的指针
       {
       struct student *head;
       struct student *p1,*p2;
       n=0;
```

```
        p1=p2=(struct student *)malloc(LEN); //开辟一个新单元
        scanf("%ld,%f",&p1->num,&p1->score);
        head=NULL;
        while (p1->num!=0)
        {
        n=n+1;
        if (n==1)
             head=p1;
        else
             p2->next=p1;
        p2=p1;
        p1=( struct student *)malloc(LEN);
        scanf("%ld,%f",&p1->num,&p1->score);
        }
p2->next=NULL;
return(head);                          //此函数返回的是head的值，也就是链表头的地址
}
```

先设三个指针变量：head、p1、p2，它们都指向结构体类型数据。先用 malloc 函数开辟一个结点，并使 p1、p2 指向它。然后从键盘读入一个学生的数据并赋给 p1 所指的结点。先使 head 的值为 NULL，这是链表为“空”的情况，即 head 不指向任何结点，链表中无结点，以后增加一个结点就使 head 指向该结点。

如果输入的 p1->num 不等于 0，而且输入的是第一个结点数据(n=1)时，则令 head=p1，即把 p1 的值赋给 head，也就是使 head 也指向新开辟的结点，p1 所指向的结点就成为链表中第一个结点。然后再开辟另一个结点并使 p1 指向它，接着读入该结点的数据。如果输入的 p1->num 不等于 0，则应链入第 2 个结点(n=2)，由于 n 不等于 1，则将 p1 的值赋给 p2->next，也就是使第一个结点的 next 成员指向第二个结点。接着使 p2=p1，也就是使 p2 指向刚建立的结点，再开辟一个结点并使 p1 指向它，并读入该结点的数据。在第三次循环中，由于 n=3，又将 p1 的值赋给 p2->next，也就是将第 3 个结点连接到第 2 个结点之后，并使 p2=p1，使 p2 指向最后一个结点。

再开辟一个新结点，并使 p1 指向它，输入该结点的数据。由于 p1->num 的值为 0，不再执行循环，此新结点不应该再连接到链表中。此时将 NULL 赋给 p2->next。建立链表结束，p1 最后所指的结点未链入链表中，最后一个结点的 next 成员的值为 NULL，它不指向任何结点，虽然 p1 指向新开辟的结点，但从链表中无法找到该结点。

2.8.6.3 输出链表

链表与结构体一样，也不能整体输入和输出，只能将链表中各结点的数据依次输出。要输出链表，首先要知道链表头元素的地址，也就是要知道 head 的值，然后设一个指针变

量 p，先指向第一个结点，输出 p 所指的结点，然后使 p 后移一个结点再输出，直到链表的最后一个结点。下例为一个输出链表的函数 print。

```
void print(struct student *head)
     {
     struct student *p;
     printf ("\nNow,These %d records are:\n",n);
     p=head;
     if (head!=NULL)
     do
          {
          printf ("%ld %5.1f\n",p->num,p->score);
          p=p->next;                              //p 指向下一个结点
          }
     while (p!=NULL);
     }
```

2.8.6.4　对链表的插入和删除操作

要将一个结点插到一个已有的链表中，将要插入的链表中位置的前一个元素的结点指向要插入的结点位置，再将要插入的结点指向要插入位置的后一个元素，就可以通过这个链表来访问这个结点了。要删除链表中某个结点，方法类似，只要改变链表的链接关系，使链表中要删除的结点的前一个结点指向要删除的结点的后一个结点，这样虽然要删除的结点还在内存中，但已不能通过链表来访问。

指针链表并不仅仅是我们所讨论的这一种单向链表，还有双向链表、环形链表等。用类似的结构还可以形成队列、树、栈、图等数据结构，有关这些问题及算法应该去学习“数据结构”。

2.8.7　共用体

2.8.7.1　共用体的概念

共用体的类型说明和变量定义的方式和结构体相同，不同的是结构体变量中每个成员分别占有自己的内存单元，所占内存长度是各成员所占的内存长度之和，而共用体变量每个成员占用相同的内存单元，所占的内存长度等于最长的成员长度。我们将这种几个不同的变量共占用相同的存储空间的结构，称为“共用体”类型结构，使用“共用体”可以提高内存的利用效率。一般定义形式为：

```
union 共用体名
{成员列表
}变量列表;
```

例如：

```
union data
    {
    int i;
    char ch;
    float f;
    }a,b,c;
```

上例中定义的共用体变量共占 4 个字节数据存储器(因为最大的 float 占 4 个字节)，如果将上例中的共用体改用结构体定义，则共需占 7 个字节数据存储器。

[例 2.23] 利用共用体，分别取出 int 变量中高字节和低字节两个数，并以 ASCII 码输出字符。

```
union change
      {
      char c[2];
      int a;
      }un;
void main(void)
      {
      un.a=16961;
      printf ("%c\n",un.c[0]);
      printf ("%c\n",un.c[1]);
      }
```

在上例中，共用体 un 中包含两个成员：字符数组 c 和整型变量 a，它们共同占有两个字节的单元。给成员 un.a 赋 16961 后，系统按 int 整型把数存放到存储空间中，其中低字节存放整数 65，高字节存放整数 66，当用 printf 输出 un.c[0]时，在输出低字节中存放整数 65 相应的 ASCII 码“A”，printf 输出 un.c[1]时，在输出高字节中存放整数 66 相应的 ASCII 码“B”。

2.8.7.2 共用体变量的引用方式

与结构体类似，只有先定义了共用体变量才能引用，而且也不能引用整个共用体变量，只能引用共用体变量中的成员。例如在上例中定义了 a、b、c 为共用体变量，可以用以下

方式引用：

```
a.i                    //引用共用体变量 a 中的整型变量 i
a.ch                   //引用共用体变量 a 中的字符变量 ch
b.f                    //引用共用体变量 b 中的实型变量 f
```

2.8.7.3　共用体类型数据的特点

在使用共用体数据时要注意以下几点：

1. 同一个内存段可以用来存放几种不同类型的成员，但在每一时刻只能存放其中一种，而不能同时存放几种，也就是说任一时刻只能有一个成员起作用。

2. 共用体变量中起作用的成员是最后一次存放的成员，在存入一个新的成员后原有的成员就失去作用。如在上面定义了 a、b、c 为共用体，有以下赋值语句：

```
a.i=1;
a.ch='a';
a.f=15.0;
```

在完成以上三个赋值运算以后，只有 a.f 是有效的，a.i 和 a.ch 已无意义了。此时用 printf("%f",a.f)是可以的，而用 printf("%d",a.i)是不行的，得不到设计者预期的结果。因此，在引用共用体变量时应该十分注意，当前存放在共用体变量中的究竟是哪个成员。

3. 共用体变量的地址和它的各成员的地址都是同一地址。例如：&a、&a.i、&a.ch、&a.f 都是指同一地址值。

4. 不能对共用体变量名赋值，也不能企图引用变量名来得到成员值，也不能在定义共用体时对它初始化。

5. 不能把共用体变量作为函数参数，也不能使函数带回共用体变量，但可以使用指向共用体变量的指针。

6. 共用体类型可以出现在结构体类型定义中，也可以定义共用体数组。反之，结构体也可以出现在共用体类型中，数组也可以作为共用体的成员。

2.8.8　枚举类型

如果一个变量只有几种可能的值，可以定义为枚举类型。所谓“枚举”是指将变量的值一一列举出来，变量的值只限于列举出来的范围。

枚举类型是 ANSI C 新标准增加的，ICCAVR 6.26C 并不支持枚举类型，但以后随着版本的增加，也会提供对枚举类型的支持，在这里只对枚举类型作简单的介绍。枚举类型定义的一般形式：

```
enum 枚举体标识符 {枚举列表};
```

如：

```
enum weekday {sun,mon,tue,wed,thu,fir.sat};
enum weekday workday,week_end;
```

在使用枚举类型时应注意：

(1) C 编译器均将枚举元素按常量处理，故称枚举常量，因此枚举元素不是变量，不能对它们赋值。

(2) 枚举元素作为常量，它们是有值的，C 语言编译按定义时的顺序使它们的值为 0、1、2、…，也可以在定义时由程序员指定枚举元素的值。如：

```
enum weekday {sun=7,mon=1,tue,wed,thu,fir.set};
```

(3) 枚举值可以用来作判断比较。如：

```
if (workday==mon) ……
if (workday>sun) ……
```

枚举值的比较规则是：按其在定义时的顺序号比较。如果定义时未人为指定，则第一个枚举元素的值为 0，后面顺序增加。

(4) 一个整数不能直接赋给一个枚举变量。如：

```
workday=2;
```

是不对的，它们属于不同的类型，应先强制转换后才能赋值。如：

```
workday=(enum weekday)2;          //相当于 workday=tue;
```

表达式经过强制转换后也可以赋值给枚举型变量。如：

```
workday=(enum weekday)(6-4);          //相当于 workday=tue;
```

2.8.9 用 typedef 定义类型

除了可以直接使用 C 提供的标准类型名(如 int、char、float、double、long 等)和自定义的结构体、共用体、指针、枚举元素外，还可以用 typedef 定义新的类型名来代替已有的类型名。说明新类型名的语句的一般形式为：

```
typedef 类型名 标识符;
```

其中“类型名”必须是此语句之前已有定义的类型标识符，“标识符”是一个用户定义标识符，用作新的类型名。

例：

```
typedef unsigned char UNCHAR;
```

在上例定义过后，

```
unsigned char a,b;
```

可以定义为：

```
UNCHAR a,b;
```

两句是等效的。

用 typedef 还可以定义一个结构体，如：

```
typedef struct
     {
     int month;
     int day;
     int year;
     }DATA;
```

上例定义了一个新的类型名 DATE，它代表上面定义的一个结构体类型，以后就可以用 DATE 定义变量：

```
DATE birthday;                    //不要写成 struct DATE birthday;
DATE *p;                          //定义 一个指向此结构体类型数据的指针
```

还可以进一步用 typedef 定义：

```
1. typedef int NUM[100];          //定义 NUM 为整型数组
   NUM a,b;                       //定义 a、b 为整型数组变量
2. typedef CHAR *STRING;          //定义 STRING 为字符指针变量
   STRING p,s[10];                //定义 p 为字符指针变量，s 为指针数组
```

在使用 typedef 时，应注意以下几点：

(1) 用 typedef 可以定义各种类型名，但不能用来定义变量。

(2) 用 typedef 语句只是对已经存在的类型增加一个变量名(原有的类型名仍然有效)，而没有创造新的类型。

(3) 要注意 typedef 与#define 的不同之处，如：

```
typedef int COUNT
#define COUNT int
```

这两个语句的作用都是用 COUNT 代表 int，但事实上，它们二者是不同的。#define 是在预编译时处理的，它只能作简单的字符串替换，而 typedef 是在编译时处理的。

(4) 当不同源文件中用到同一类型数据(特别是像数组、指针、结构体、共用体等类型数据)时，常用 typedef 定义一些数据类型，把它们单独放在一个文件中，然后在需要用到它们的文件中用#include 命令把它们包含进来。

2.9 位 运 算

C 语言是为描述系统而设计的，因此它具有汇编语言所能完成的一些功能，有较好的位操作指令。

在控制领域，有时经常需要控制某一个二进制位，为了编程的方便，在 Keil 中扩充了两个数据类型 bit 和 sbit，其中前者只可定义 MCS-51 的位寻址区，后者只能定义为可位寻址的特殊功能寄存器 SFR(含 I/O 寄存器)中的某一位。

CodeVisionAVR 也定义了 bit 类型数据，在访问 I/O 寄存器时，可以直接访问 I/O 寄存器的某一位，如需将 DDRB 的 D3 位置 1，可以这样：

```
DDRB3=1;
```

而在 ICCAVR、GCCAVR、IAR 中都没有引进相类似的扩充关键字，当它们需要访问 I/O 寄存器的某一位时，只能使用 ANSI C 语言的位运算功能。如要将 DDRB 的 D3 位置 1，在 ICCAVR 中可以这样：

```
DDRB|=(1<<3)
```

或使用 macros.h 头文件中定义的 BIT(i)宏，如下：

```
DDRB|=BIT(3)
```

在使用 BIT(i)宏时，能够再加上 io*v.h 头文件定义的端口宏名称，可改为如下方式：

```
DDRB|=BIT(DDB3)
```

说明：各种 C 编译器对位定义及处理的方式不同，如果使用位变量，会使程序的移植性降低。

2.9.1 位运算符

ANSI C 语言提供表 2.8 所示的位运算符：

表 2.8　ANSI C 语言提供的位运算符

运　算　符	含　义
&	按位与
|	按位或
^	按位异或
~	取反
>>	左移
<<	右移

说明：

(1) 位运算符中，除~以外，均为二目运算符，即要求两侧各有一个运算量。

(2) 运算量只能是整型或字符型的数据，不能为实型数据。

(3) 位运算符可以与赋值运算符组成扩展运算符，如：

&=、|=、^=、>>=、<<=。

在对单片机的位操作中，最典型的就是对 I/O 端口进行操作，下面介绍在 ICCAVR 的位操作时，就以 ANSI C 提供的位运算符对 AVR 单片机的端口进行相应的操作为例。如果需对内存或寄存器进行位操作，与对端口的操作类似。

1．按位与运算符(&)

参加运算的两个运算量，如果两个相应的位都为 1，则该位的结果值为 1，否则为 0。按位与常有以下用途：

(1) 清零某一端口，其他位保持不变。如果想将 PA7 清零，可以用：

```
PORTA&=0x7f;
```

- 当选择编译扩充(Project->Options->Compiler)后，可以使用二进制常数，会更直观，如将上例中的 16 进制常量改为二进制常量：

```
PORTA&=0b01111111;
```

- 在本节的例子中仅是以操作某一位为例，实际上可以同时对同一地址(端口)的某几位或全部位进行操作，与操作其中的一个位类似。如要清零 PA7、PA4、PA0，可以用：

```
PORTA&=0x6e;
```

(2) 取某一位状态。假设 PA7 通过一个上拉电阻与一个按钮开关相连，按钮开关的另一端接地，当按钮断开时，PA7 端口为高电平，按钮按下时，PA7 端口为低电平，程序中要根据按钮的状态决定程序执行的方向，可以用：

```
if ((PIND&0x80)==0) 程序语句;
```

也可以使用以下语句实现：

```
if ((PIND&(1<<7))==0) 程序语句;
```

在本例中按钮按下(PA7 为低电平)则程序转移，如果程序要按钮断开(PA7 高电平)程序转移，可以使用以下语句实现：

```
if ((PIND&0x80)==0x80) 程序语句;
```

或：

```
if ((PIND&(1<<7))==(1<<7)) 程序语句;
```

实际上，AVR 单片机的 I/O 口在作为输入时可以打开自带的上拉电阻，因此在实际应用时可以省略上拉电阻，将 I/O 口通过按钮与地直接连接。

(3) 保留某位的状态，其余位均清零。假设需要保留 PA7 的状态，可以用：

```
PORTA&=0x80;
```

2．按位或运算符(|)

两个相应位中只要有一个为 1，该位的结果值为 1。常用来置位某一端口，其他位保持不变。如果想将 PA7 置位，可以用：

```
PORTA|=0x80;                    //0x80=0b10000000
```

3．异或运算符(^)

参中运算的两个相应位相同，结果为 0；相异，结果为 1。常有以下用途：

(1) 使特定的位翻转，其他位不变。假设需要翻转 PA7，可以用：

```
PORTA^=0x80;
```

(2) 不用临时变量、交换两个变量的值。已定义 a、b 为 char，要交换 a、b 的数据，如下：

```
a^=b;
b^=a;
a^=b;
```

4．取反运算符(~)

~是一个单目运算符，用来对一个二进制数按位取反，常与其他位运算符一起使用。设已将 0x80 赋值给 a(类型为 unsiged char)，则~a 后，a 的值为 0x7f。

5．左移运算符(<<)

用来将一个数据二进制位全部左移若干位，右端补 0，高位左移后溢出，舍弃不用。

左移 1 位相当于该数据乘以 2，左移 n 位等于该数据乘以 2^n。当然，这个结论只能适用于数据左移时，被舍弃的高位中不包含 1 的情况。

在不支持硬件乘法的单片机中，如果需要乘以 2^n 运算，用左移 n 位的方法比调用乘法子程序的方法快得多。实际上在 ICCAVR 6.26C 中就是用左移的方法实现乘以 2^n 运算。

6．右移运算符(>>)

用来将一个数据二进制位全部右移若干位，在右移时，需要注意符号位的问题：对于无符号数，右移时左边高位移入 0，右边低位舍弃不用；对于有符号数，右移时左边高位移入符号位，右边低位舍弃不用。

右移 1 位相当于该数据除以 2，左移 n 位等于该数据除以 2^n。

7．不同长度的数据进行位运算

如果两个数据长度不同(如 long 型和 int 型)进行位运算时，系统会将二者按右端对齐，如果字节的数据为有符号数且为正数，系统则在左端高位补 0，若字节少的数据为有符号数且为负数，系统则在左端高位补 1。如果字节少的数据为无符号数据，系统则在左端高位补 0。

2.9.2　位域

在 ICCAVR 6.26C 中，不能直接定义位变量，但是可以使用 C 语言中的结构体定义位变量。如下例：

```
struct data
       {
       unsigned bit0:1;
       unsigned bit1:1;
       unsigned bit2:1;
       unsigned bit3:1;
       unsigned bit4:1;
       unsigned bit5:1;
       unsigned bit6:1;
       unsigned bit7:1;
       }a,b;
```

在上例中，成员列表的位域的类型必须用 unsigned 或 signed 定义关键字。a、b 均占用一个字节的存储空间。定义后就可以直接使用位变量，如：

```
a.bit1=0;
a.bit7=1;
if (a.bit5) 程序语句;
```

由于 ICCAVR6.26C 版中，编译器没有充分利用 AVR 单片机对于寄存器所具有的位寻址功能，在使用位变量结构体时，编译器将位变量结构体定位于数据存储器而不是寄存器中，因此生成的代码量大，执行效率低，这也是 ICCAVR 目前的一个不足之处，在以后的

版本中会逐渐改进。

在 macros.h 头文件中，定义了一个“#define BIT(x) (1 << (x))”宏，用该宏来操作位变量，编译后生成的代码质量更高。如下例：

```
#define PORTA0 0
#define PORTA1 1
#define PORTA2 2
#define PORTA3 3
#define PORTA4 4
#define PORTA5 5
#define PORTA6 6
#define PORTA7 7
```

在使用前，应在文件头加入“#include "macros.h"”定义。如要将 PORTA7 清零，可以使用以下语句：

```
PORTA&=~BIT(PORTA7);
```

如需要将 PORTA7 置位，可以使用以下语句：

```
PORTA|=BIT(PORTA7);
```

这样对 I/O 端口进行位操作，比较直观，符合一般人的思维习惯，且生成的代码能够充分利用 AVR 单片机对 I/O 端口操作的位指令，生成高质量的代码。

2.10 标识符的作用域和存储类型

2.10.1 局部变量和全局变量

由用户命名的标识符都有一个有效的作用域。所谓标识符的作用域，就是指在程序中的某一部分中，该标识符是有定义的，可以被 C 编译和连接程序所识别。例如在函数内部定义的变量，就不能在其他函数中引用，显然，变量的作用域与其定义语句在程序中出现位置有直接的关系，据此可以划分出局部变量和全局变量。

在函数内部或复合语句内部定义的变量，称为局部变量，函数的形参也属于局部变量。在函数外部定义的变量，称为全局变量。有时，把局部变量也称为内部变量，全局变量也称为外部变量。

在 C 语言中，有两种存储类别：自动类和静态类。局部变量既可以说明成自动类，也可以说明成静态类，而全局变量只能是静态类。有四个与存储类别有关的说明符，它们为：auto(自动)、register(寄存器)、static(静态)、extern(外部)。这些说明符通常与类型名一起出

现，它们可以放在类型名的左边，也可以放在类型名的右边。如：

```
static int a,b;
```

也可以写成：

```
int static a,b;
```

存储类别确定了所说明对象在内存中的存储位置，从而也确定了所说明对象的作用域和生存期。

2.10.2　局部变量及其作用域和生存期

2.10.2.1　auto 变量

当在函数内部或复合语句内定义变量时，如果没有指定存储类型或使用了 auto 说明符，系统就认为所定义的变量具有自动类别功能。

auto 类变量的存储单元被分配在内存的动态存储区。每当进入函数体(或复合语句)时，系统就自动为 auto 变量分配存储单元，退出时自动释放这些存储单元并分配给其他函数使用，当再次进入函数体(或复合语句)时，系统将为它们再次分配存储单元。函数的每次调用，动态存储区内为某变量分配的存储单元位置也不固定，为随机分配，这类局部变量的作用域是从定义位置起到函数(或复合语句)结束为止，这就是 auto 类局部变量的“生存期”。

自动变量赋初值是在程序运行过程中进行的，每次进入函数体(或复合语句)，就赋一次指定的初值。使用这类局部变量最突出的优点是：可在各函数之间造成信息隔离，不同函数中使用了同名变量也不会相互影响，从而可避免因为不慎赋值而影响到其他函数。

要注意，没有赋初值的 auto 类变量并不是没有值或值为零，而是一个随机的数值，称为“无定义”。

2.10.2.2　register 变量

寄存器变量也是自动类变量，只有局部自动变量和形式参数可以作为寄存器变量，其他(如全局变量)则不行。寄存器变量与 auto 类变量的区别仅在于：用 register 说明的变量是建议编译器将变量的值存放在寄存器中，用 auto 说明的变量是要求编译器将变量的值存放在数据存储单元中。由于编程人员用 register 说明变量类型仅仅只是建议而非强制，因此，最终由 C 编译器决定是否将该变量放入寄存器中，如果编译程序认为指定的变量不适合放在寄存器中，或者它认为将其他变量放在寄存器中生成的代码更优化，就会把说明为 register 类的变量按 auto 类变量来处理，并将其置于数据存储器(SRAM)中；或者由于单片机内部的寄存器有限，如果用户在函数中定义了太多的寄存器变量，就可能没有足够的寄存器可用，C 编译器也会将 register 类变量按 auto 类变量来处理。

对于全部用 C 语言编写的源文件，如果 C 编译器将编程人员定义的 register 变量转为

auto 类来处理，编译器会自动生成读写 SRAM 的指令来访问用户原先定义的寄存器变量，但是，如果用户将某一变量定义为 register 类型，并且使用了在线汇编语句对该变量按寄存器类型进行访问(使用访问寄存器的指令)，C 编译器将这一变量从 register 类型改为 auto 类型并放置于 SRAM 后，C 编译器并不会自动转换源程序中用户用汇编指令访问该变量的语句，因此在编译后生成的目标文件中就会出现用访问寄存器的指令去访问 SRAM 中的数据，导致程序运行出错，对于这样的错误，C 编译器并不会发出警告。

在 ICCAVR、GCCAVR、IAR、Keil 中，不建议用户指定寄存器变量，因为 C 编译器在优化时，都能够将最常用的自动类变量定义为寄存器类型变量。

由于在冯诺依曼结构的 PC 机中寄存器没有地址，因此 ANSI C 中规定不能对它进行求地址运算(&)。在哈佛结构的 AVR 中，寄存器也是有地址的(在 AT90S8515 芯片中，寄存器的地址为 0x00~0x5f)，但 ICCAVR、GCCAVR、IAR、Keil 中为了与 ANSI C 兼容，也不允许对 register 变量进行求地址运算。

2.10.2.3 静态存储类的局部变量

当在函数体(或复合语句)内部，用 static 来说明一个变量时，可以称该变量为静态局部变量。静态局部变量的作用域仍与 auto、register 类的变量一样，但它与前两者有本质上的区别：

1. 在整个程序运行期间，静态局部变量在内存的静态存储区中占据着永久性的存储单元。即使退出函数后，下次再进入该函数时，静态局部变量仍使用原来的存储单元，这些单元的值得以保留，可以继续使用原来的值。静态局部变量的生存期将一直延续到整个程序运行结束。

2. 静态局部变量的初值是在编译时赋予的，在程序执行期间不再赋初值，对未赋初值的静态局部变量，C 编译程序自动给它赋初值 0。

静态局部变量的上述特点，对于编写那些在函数调用之间必须保留局部变量值的独立函数是非常有用的。

2.10.3 全局变量及其作用域和生存期

全局变量只有静态一种类别，对全局变量可以使用 extern 和 static 两种说明符。

2.10.3.1 全局变量的作用域和生存周期

全局变量是在函数外部任意位置上定义的变量，它的作用域是从变量定义的位置开始，到整个源文件结束为止，生存期为整个程序运行期间。若全局变量和某一函数内部的局部变量同名，则在该函数中，此全局变量被屏蔽，在函数内通过该变量名访问的是局部变量而不是全局变量。

使用全局变量有以下特点：

1．不论是否需要，全局变量在整个程序运行期间都占用内存空间。

2．全局变量必须在函数体以外定义，降低了函数的通用性，影响了函数的独立性。

3．使用全局变量，容易因使用不当导致全局变量的值意外改变，产生难以查找的错误。

2.10.3.2　用 extern 来说明全局变量

一个 C 程序总是由许多函数组成，这些函数可以分别存放在不同的源文件中，每个源文件可以单独进行编译和进行语法检查，若无错误都可以生成各自的目标文件(.obj)，然后可用系统提供的链接程序把多个目标文件连接。通常，我们把每个可进行单独编译的源文件称为“编译单位”。

当一个程序由许多编译单位组成，并且在每个文件中均需要引用同一个全局变量，这时，若在每个文件中都定义了一个需要的同名全局变量，则在“连接”时将会产生“重复定义”错误。解决的办法通常是在其中一个文件中定义所需全局变量，而在其他用到这些全局变量的文件中用 extern 对这些全局变量进行说明，声明这些变量已在其他编译单位中定义，通知在本文件的编译过程中，编译程序不必再为它们开辟存储单元。

还有一种情况，在同一个编译单位中，当全局变量定义在后，而引用它的函数在前时，应该在引用它的函数中用 extern 对此全局变量进行说明，以便通知编译程序该变量是一个已在外部定义了的全局变量，已分配了存储单元，在本函数内部不必为它另外开辟存储单元，这时的作用域从 extern 说明开始，一直延伸到该函数的结局。

注意，全局变量的说明与全局变量的定义不同：变量的定义(开辟存储单元)只能出现一次，当定义全局变量时，不可以使用 extern 说明符；而对全局变量的说明，就必须用 extern 进行说明。全局变量只能定义一次，却可以多次说明。

2.10.3.3　用 static 来说明全局变量

当用 static 说明符说明全局变量，此变量可以称作“静态”全局变量，静态全局变量只限于在本编译单位使用，不能被其他编译单位所引用，使用 static 说明限制了全局变量作用域的扩展，达到了信息隐蔽的目的。这对于编写一个具有众多编译单位的大型程序是十分有益的，程序员不必担心因全局变量重名而引起混乱。

2.10.3.4　函数的存储分类

所有函数在本质上都是外部的，因为 C 语言不允许在函数内部再定义另一个函数。但当定义函数时，可以使用 extern 和 static 说明符。

在定义一个函数时，在函数的返回值的类型前面加上说明符 extern 时，则称此函数为“外部”函数。extern 说明可以省略，默认为 extern 状态，具有 extern 说明符的函数，可

以被其他编译单位中的函数调用。通常，当函数调用语句与被调用函数不在同一编译单位，且函数的返回值为非整型时，应该在调用语句所在函数的说明部分用 extern 对所调用的函数进行说明。函数定义与函数说明是不同的，要注意区别。

当定义一个函数时，在函数的返回值的类型前面加上说明符 static 时，则称此函数为“静态”函数。具有 static 说明符的函数，只限于本编译单元的其他函数调用它，而不允许其他编译单位中的函数调用，因此静态函数又称作“内部”函数，即只限于内部文件的函数。

使用静态函数，可以避免不同编译单位因函数同名而引起混乱。若强行调用其他编译单位的静态函数，会产生出错信息。

2.11 编译预处理

编译预处理是C编译系统的一个组成部分，这是C语言与其他高级语言的一个重要区别。C语言允许在程序中使用几种特殊的命令，在C编译系统对程序进行通常的编译(包括词法和语法分析、代码生成、优化等)前，先对程序中这些特殊的命令进行“预处理”，然后将预处理的结果和源程序一起再进行通常的编译，得到目标代码。

C语言中，预处理命令组成的预处理命令行必须在一行的开头以“#”号开始，每行的末尾不得加“;”，以区别于C语言的定义和说明语句。这些命令行的语法与C语言中其他部分的语法无关，它们可以根据需要出现在程序的任何一行的开始部位，其作用一直持续到源文件的末尾。

C语言提供的预处理主要有以下三种：

1. 宏定义
2. 文件包含
3. 条件编译

2.11.1 宏定义

2.11.1.1 不带参数的宏定义

用一个指定的标识符(即名字)来代表一个字符串，它的一般形式为：

```
#define 标识符 字符串
```

如：

```
#define PI 3.1415926
```

上句的作用是用指定标识符PI来代替“3.1415926”这个字符串，在编译预处理时，将

源文件中在该命令以后的所有 PI 都用“3.1415926”代替。这种方法使用户能以一个简单的名字代替一个长的字符串，因此把这个标识符(名字)称为“宏名”，在预编译时将宏名替换成字符串的过程称为“宏展开”，#define 是宏定义命令。

在进行宏定义时应注意：

1．宏定义不是 C 语句，不必加分号，如果误加分号，则会连分号一起置换。如将上例错误的定义为：

```
#define PI 3.1415926;
```

假设源文件中有以下语句：

```
area=PI*r*r;
```

则宏展开后，该语句变为：

```
area=3.1415926;*r*r;
```

显然出现语法错误。

2．当宏定义在一行中写不下，需要在下一行继续写时，应该在最后一个字符后紧跟着加一个反斜线“\”，新的一行应从第一列开始书写，不能插入空格。

```
#define LEAP_YEAR year%4==0\
&year%100!=0|year%400==0          //注意，新起的一行应从第一列开始书写
```

如果在“\”前或在下一行的开头留有空格，那么在宏替换时也将加入这些空格。

3．宏名一般习惯用大写字母表示，以区别于变量名。但这并非规定，也允许用小写字母。

4．使用宏名代替一个字符串，可以减少程序中重复书写某些字符串的工作量，易记且不易出错，当需要改变某一个常量时，只要改变#define 命令行就可以了。

5．宏定义是用宏名代替一个字符串，也就是作简单的置换，不作语法检查。如果语法有错，只有在编译已被宏展开后的源程序时才报错。

6．#define 命令通常出现在函数的外面，宏名的有效范围为宏定义之后到宏所在的源文件结束。通常#define 命令写在文件开头，作为文件的一部分，在此文件范围之内都有效。

7．可以用#undef 命令中止宏定义的作用域。

8．在宏定义时，可以引用已定义的宏名，还可以层层置换。如：

```
#define R 3
#define H 10
#define PI 3.1415926
#define AREA PI*R*R
#define BULK AREA*H
```

在进行宏定义时，ICCAVR、GCC、IAR 和 Keil 均支持不限层数的置换，但有些早期版本的 C 编译器有置换层数限制(如 CVAVR)。

9. ICCAVR 6.26C 编译器对程序中用双引号括起来的字符串内的字符、标号和标识符，即使与宏名相同，也不进行置换，但有些 C 编译器可能会对标号或标识符也进行置换操作，但不影响编译结果(必须在文件头进行宏定义)。

2.11.1.2 带参数的宏定义

1. 带参数的宏定义

带参数的宏在预编译时不但进行简单的字符串替换，还要进行参数替换。其定义的一般形式为：

```
#define 宏名(参数表) 字符串
```

字符串中包含在括号中所指定的参数。如：

```
#define S(a,b) a*b
┆
area=S(3,2);
```

此宏展开后为：

```
area=3*2;
```

对带参的宏定义是这样展开置换的：在程序中如果有带实参的宏，则按#define 命令行中指定的字符串从左到右进行置换。如果字符串中包含宏中的形参，则将程序语句中相应的参数(可以是常量、变量或表达式)代替形参，如果宏定义字符串中的字符不是参数字符，则保留。

2. 对带参的宏定义的说明

(1) 对带参数的宏的展开，只是将语句中的宏名后面括号内的实参字符串代替#define 命令行中的形参。在下例中：

```
#define PI 3.1414926
#define S(r) PI*r*r
area=S(a+b);
```

这时把实参 a+b 代替 PI*r*r 中的形参 r，成为：

```
area=PI*a+b*a+b;
```

这与设计原意不符，原意是希望得到：

```
area=PI*(a+b)*(a+b);
```

为了得到这个结果，应当在定义时在字符串中的形参外面加一个括号，即：

```
#define S(r) PI*(r)*(r)
```

(2) 在宏定义时，在宏名与带参数的括号之间不应加空格，否则将空格以后的字符都作为替代字符串中的一部分。

3．带参数的宏与函数的区别：

(1) 函数调用时，先求出表达式的值，然后代入形参。而使用带参的宏只是进行简单的字符替换，在宏展开时并不求解表达式的值。

(2) 函数调用是在程序运行时处理的，而宏展开则是在编译时进行的，在展开时不分配内存单元，不进行值的传递处理，也没有“返回值”的概念。

(3) 对函数中的实参和形参都要定义类型，二者的类型要求一致，如不一致，编译器会对之进行类型转换。而宏不存在类型问题，宏名无类型，它的参数也无类型，只是一个符号代表，展开时代入指定的字符即可。因此在定义宏时，字符串可以是任何类型的数据。

(4) 调用函数只可得到一个返回值，而用宏却可以设法得到几个结果。

(5) 使用宏定义时，宏展开后源程序变长，而函数调用不使源程序变长。

(6) 宏替换只占用编译时间，不占用运行时间，而函数调用则占用运行时间(分配单元、保留现场、值传递、返回)。

2.11.2　“文件包含”处理

所谓“文件包含”处理是指一个源文件可以将另外一个源文件的全部内容包含进来，即将其他的文件包含到本文件之中。C 语言提供了#include 命令来实现“文件包含”的操作。其一般形式为：

```
#include "文件名"
```

例如：

```
#include "stdio.h"
```

在使用文件包含时应注意：

(1) #include 命令行应书写在所用文件的开头，故有时也把包含文件称作“头文件”。头文件名可以由用户指定，其后缀不一定用“.h”。

(2) 文件包含在编译时并不是作为两个文件链接，而是合为一个文件进行编译，只得到一个目标文件，因此被包含的文件也应该是源文件而不能是目标文件。

(3) 一个 include 命令只能指定一个被包含文件，如果要包含 n 个文件，要用 n 个#include 命令。

(4) 如果文件 1 包含文件 2，而文件 2 要用到文件 3 的内容，则可以在文件 1 中用两个#include 命令分别包含文件 2 和文件 3，而且文件 3 应该出现在文件 2 之前，即要在 file1.1

中定义：

```
#include "file3.h"
#include "file2.h"
```

但如果在文件 2 中已用#include "file3.h"包含了文件 3，在文件 1 中就不一定非用#include "file3.h"定义。

(5) 在一个被包含的文件中又可以包含另一个被包含的文件，即文件包含是可以嵌套的。

(6) 在#include 命令中，文件名可以用双引号或尖括号括起来，如：

```
#include "file2.h"
```

也可以定义成：

```
#include <file3.h>
```

以上二者都是合法的，区别为：用双引号的形式，系统先在引用被包含文件的源文件所在的目录中寻找要包含的文件，若找不到，再按系统隐含的标准方式检索其他目录。而用尖括号的形式时，不检查源文件所在的文件目录而直接按系统标准方式检索文件目录。

(7) 当包含文件修改后，对包含该文件的源程序必须重新进行编译。

(8) 被包含文件中有全局静态变量，它在包含文件中也有效，不必用 extern 说明。

2.11.3 条件编译

一般情况下，源程序中所有的行都参加编译，但是有时希望满足某些条件时对一组语句进行编译，当条件不满足时则编译另一组语句，这就是“条件编译”。条件编译功能是 C 语言特有的，有利于增加程序的灵活性。通常有以下三种形式：

1. 一种形式为：

```
#ifdef 标识符
    程序段 1;
#else
    程序段 2;
#endif
```

它的作用是：当标识符已经被定义过(一般是用#define 命令定义)，则对程序段 1 进行编译，否则编译程序段 2，其中#else 部分可以省略。这里的程序段可以是语句组，也可以是命令行。

例如，在调试时，常常希望输出一些所需的信息，而在调试完成后不再输出这些信息。可以在源程序中插入以下的条件编译段：

```
#ifdef DEVUG
printf("x=%d,y=%d,z=%d\n",x,y,z);
#endif
```

如果在它前面有以下命令行：

```
#define DEBUG
```

则在程序运行时输出 x、y、z 的值，以便调试时分析程序运行的情况。调试完成后只需将这个 define 命令行删去即可。

2．另一种形式为：

```
ifndef 标识符
    程序段 1
#else
    程序段 2
#endif
```

它的作用是：若标识符未被定义则编译程序段 1，否则编译程序段 2，与第一种形式的作用相反。

3．第三种形式：

```
#if 表达式
    程序段 1
#else
    程序段 2
#endif
```

它的作用是：当指定的表达式值为真(非零)时就编译程序段 1，否则编译程序段 2。可以事先给出一定的条件，使编译器在不同的条件下编译不同的语句。

在程序中，有时不使用条件编译而改用 if 语句也能达到要求，但是使用 if 语句会使目标程序变长，因为所有的语句都参加编译，而采用条件编译后，可以减少被编译的语句，从而减少目标程序的长度，当条件编译段比较多时，目标程序长度可以大大减少。

在 ICCAVR 6.26C 中支持以上三种条件编译，但在使用条件编译前必须选择编译扩充选项(Project->Options->Compiler)，且最多只能有 10 层的条件编译嵌套。

2.11.4　编译附注和扩充

由于 C 语言是为冯诺依曼结构的 PC 机而开发的，MCS-51 和 AVR 均为哈佛结构，数据存储器与程序存储器是分开的，因此几乎所有针对单片机的 C 编译器都进行了不同的语

法扩充，以适应这种结构的变化。

1．对寄存器变量的访问

IAR 和 CodeVisionAVR 都定义了新的数据类型“sfrb”和“sfrw”，使 C 语言可以直接访问 MCU 的有关寄存器。如：sfrb DDRD=0x11。

而 ICCAVR 没有定义 sfrb 和 sfrw 数据类型，而是采用强制类型转换和指针的概念来实现访问 MCU 的寄存器，如：

```
#define DDRD (*(volatile unsigned char *)0x31)
```

则在 ICCAVR 程序中可以使用如下方式访问：

```
DDRD=0xff;
```

在 GCCAVR 中，同 ICCAVR 一样采用强制类型转换。在笔者现在使用的版本中(2002-06-25)，以 AT90S8515 为例，定义寄存器 DDRD(在头文件 io8515.h 中)：

```
#define DDRD _SFR_IO8(0x11)
```

0x11 是寄存器 DDRD 的 IO 地址，_SFR_IO8 是 GCCAVR 在头文件 sfr_defs.h 中定义的宏：

```
#define _SFR_IO8(io_addr) _MMIO_BYTE((io_addr) + 0x20)
```

有意思的是，这里的宏 _MMIO_BYTE 也是 GCCAVR 在头文件 sfr_defs.h 中定义的宏：

```
#define _MMIO_BYTE(mem_addr) (*(volatile unsigned char *)(mem_addr))
```

经过宏替换后，最后的定义格式是和 ICCAVR 一样的。

对 I/O 寄存器的操作，GCCAVR 的 2002-06-25 版本只支持使用 GCCAVR 编译系统提供的 API 函数：

```
outp(0xFF, DDRD);                    //向 DDRD 寄存器写 0xFF
temp=inp((DDRD);                     //读取 DDRD 寄存器到变量 temp 中
```

新版中(2003-01-15)建议使用直接赋值的方法操作寄存器，如：

```
DDRD=0xFF;                           //向 DDRD 寄存器写 0xFF
temp=DDRD;                           //读取 DDRD 寄存器到变量 temp 中
```

与 ICCAVR 中访问方式一致！

2．使用 AVR 片内不同的存储空间

由于 AVR 单片机内部有三种类型的存储空间(RAM、EEPROM 和 FLASH)，为了能有效地访问这三种存储器，IAR、CodeVisionAVR、ICCAVR 和 GCCAVR 编译器分别进行了不同的语法扩充：

(1) IAR 中扩充了一个关键词 flash，由于 AVR 的内部 RAM 数量有限，使用 flash 关键词可以将使用 const 类型定义的常量分配进 FLASH 存贮器，以节省 RAM 的使用。在 IAR 中对片内 EEPROM 的访问只能通过函数_EEPUT 和_EEGET 进行。

(2) 在 CodeVisionAVR 中扩充了 flash 和 eeprom 两个关键词，flash 的用法同 IAR 一样，而 eeprom 关键词限定的变量则被分配进片内 EEPROM 中，在 C 语言中访问 EEPROM 中变量的方法与访问 RAM 中的变量的方法完全相同，包括指针形式的访问。

(3) 在 ICCAVR 中对 const 类型进行了扩充，编译器自动将 const 类型数据分配进 FLASH 存储器中，对片内 EEPROM 存贮器的访问，ICCAVR 只能通过调用头文件 eeprom.h 中的函数 EEPROM 进行。ICCAVR 同时也扩充了一个新的 eeprom 存贮区域，可以在 eeprom 区域中定义变量，然后再通过“&”运算符获取变量的地址对其进行访问。

(4) 在 GCCAVR 中，如果要把数据存放在 FLASH 中，使用 prog_XXX 数据类型进行声明，使用 FLASH 数据时，需要包括头文件 progmem.h。

字符型(8 位) prog_char

整型(16 位) prog_int

长整型(32 位)prog_long

双精度(64 位) prog_long_long

举例说明：

```
pro_char LINE={1};                                    //声明字符型常数
pro_char LINE110]={0,1,2,3,4,5,6,7,8,9};              //声明字符型常数数组
char * LINE2=PSTR("The first line of my test.");      //声明字符串指针
char  LINE3[] __attribute__((progmem)) = "The first line of my test.";
                                                      //声明字符串指针
```

读取 FLASH 操作时使用 GCCAVR 提供的 API 函数 PRG_RDB，假定声明 res, res1, res2, res3 为 char 类型变量。

```
res=PRG_RDB(&LINE);                                   //读取操作，下同
res1=PRG_RDB(&LINE1[5]);
res2=PRG_RDB(LINE2+3);
res3=PRG_RDB(&LINE2[3]);
```

对 EEPROM 的操作，需要包括头文件 eeprom.h，访问 eeprom 数据也是通过 API 函数实现：

```
unsigned char eeprom_rb(unsigned int addr);           //从地址 addr 读取一
                                                        个字节
unsigned int eeprom_rw(unsigned int addr);            //从地址 addr 读取
                                                        个整形数据
```

```
void eeprom_wb(unsigned int addr, unsigned char val);  //向地址 addr 写字节
                                                          数据 val
//从 addr 开始的地址读取 n 个字节到 buf 所指的空间
void eeprom_read_block(void *buf, unsigned int addr, size_t n);
```

可以采用下面这种方式定义不需指定特殊地址的 eeprom 变量：

```
static uint8_t val1 __attribute__ ((section(".eeprom")));//声明 val1 为 eeprom
                                                          空间的 static 变量
eeprom_wb((uint8_t)&val1, 0xAA);                        //写 0xAA 到 eeprom
                                                          中的变量 val1
eeprom_wb(0x00, 0xAA);                                  //写 0xAA 到 eeprom
                                                          中的 0x00 地址
```

上述四种 C 语言编译器对 FLASH 中的代码和常数均可以生成数据存储器用的 INTEL HEX 格式文件，而 ICCAVR、GCCAVR 和 CodeVisionAVR 还可以对 EEPROM 的初始化数据生成 INTEL HEX 格式的烧写文件，IAR 则没有这个功能。

3．ICCAVR 的编译附注

(1) 中断函数附注：

```
#pragme interrupt_handle <fun1>:<vector number> <fun2>:<vector number>
```

说明函数 func1 和 func2 为中断操作函数，详见“2.6.5 中断服务函数”一节。

该附注必须在函数前定义。例：

```
#pragma interrupt_handler time1:7
void time1(void)
{函数体;}
```

(2) C 任务：

```
#pragma ctask <fun1><fun2>…
```

这个附注指定了函数中不插入保存和恢复可变寄存器(Volatile Registers)中数据的代码，它的典型应用是在 RTOS 实时操作系统中让 RTOS 核直接管理寄存器。

在 ICCAVR 中，将以下寄存器称为可变寄存器：

R0/R1/R2/R3/R4/R5/R6/R7/R8/R9/R24/R25/R26/R27/R30/R31

如在 UCOS/II 中，可以使用以下语句说明 drive_motor 和 emit_siren 为 C 任务函数：

```
#pragma ctask drive_motor emit_siren
void drive_motor() {程序语句;}
void emit_siren() {程序语句;}
```

(3) 改变代码段名称：

```
#pragma text:<name>
```

改变代码段名称，使其与命令行选项相适应。
例如：

```
#pragma text:myarea
void boot()                    //函数说明
#pragma text:text
```

在上例中定义了一个函数，将该函数放在用户自定义的 bootloader 区中，在源文件中加入了以上改变代码段名称的编译附注后，还必须在“Other OPtions(Project->Options->Target)”选项中填入： bmyarea:0x1FC00.0x20000。在使用时应注意：

- 用户可以管理在“bootloader”区中自己定义的中断向量。
- 使用该定义时，在选中代码优化(Optimizstions->Enable Code Compression)时可能会有问题。
- 该定义必须紧跟着头文件定义之后进行定义(至少必须在定义函数之前定义)，或者单独使用一个文件定义，然后通过#include 命令将该文件包含进来。

(4) 改变数据段名称：

```
#pragma data:<name>
```

改变数据段名称，使其与命令行选项相适应。例：

```
#pragma data:code                          //设置数据区为程序存储器
const unsigned char tabe[]={0x3f,0x06,0x5b,0x4f,0x66,0x6d,0x7d,0x07,0x7f,
                            0x6f,0x77,0x7c,0x39,0x5e,0x79,0x71};
#pragma data:eeprom                        //设置数据区至 eeprom 存储器
int a=1,b=2,c=3;
#pragma data:data                          //设置数据区回到数据存储器
```

说明：该定义必须紧跟着头文件定义之后进行定义(至少必须在定义函数之前定义)，或者单独使用一个文件定义，然后通过#include 命令将该文件包含进来。

(5) 程序存储器绝对定位：

```
#pragma abs_address:<address>
```

函数与全局数据不使用浮动定位，而是从<address>开始分配绝对地址。

```
#pragma end_abs_address
```

结束绝对定位，使目标程序恢复正常的浮动定位。

说明：使用程序存储器的绝对定位指令在访问中断向量或其他硬件项目时特别有用，该定义必须紧跟着头文件定义之后进行定义(至少必须在定义函数之前定义)，或者单独使用一个文件定义，然后通过#include 命令将该文件包含进来。

例：我们需将液晶模块中使用的汉字库存放于 0x800 开始的地址，可以用以下方式。

```
#pragma abs_address:0x800
const unsigned char s001[]={0x20,0x40,0x10,0x40,0x10,0x40,0x87,0xFE,0x4C,0x44,0x54,
0x48,0x10,0xC0,0x10,0xC0,0x20,0xC0,0x20,0xC0,0xE1,0x40,0x21,0x42,0x22,0x42,0x24,
0x42,0x38,0x3E,0x00,0x00};                          //沈，16*16 点阵汉字库
const unsigned char s002[]={0x02,0x00,0x01,0x00,0x01,0x00,0xFF,0xFE,0x08,0x20,0x08,
0x20,0x08,0x20,0x04,0x40,0x04,0x40,0x02,0x80,0x01,0x00,0x02,0x80,0x04,0x60,0x18,
0x1E,0xE0,0x08,0x00,0x00};                          //文，16*16 点阵汉字库
#pragma end_abs_address
```

4．ICCAVR 的编译扩充

ICCAVR 默认的设定为接受编译扩充“Accept Extensions(C++ comments,binary constants)”(在 Project->Options->Compiler 中)．接受编译扩充后，能够使用以下功能：

(1) C++注释

可以在源代码中使用 C++的“//”类型的注释。

(2) 二进制常数

可以使用 0b<1|0>来指定二进制常数，例如，0b00010101 等于十进制数 21。

(3) 在线汇编

可以使用 asm("string")函数来指定在线汇编代码，读者可参考“2.12 在线汇编”的内容。

(4) 条件编译

条件编译是 ANSI C 中的内容，但只在选择了编译扩充功能之后，ICCAVR 6.26C 才支持条件编译。

2.12 在 线 汇 编

IAR 不支持在线汇编，ICCAVR、GCCAVR 和 CodeVisionAVR 均支持在线汇编，即可在 C 语言程序中直接嵌入汇编语言程序，甚至可以将汇编语言放在所有的 C 函数体之外或者使用单独的汇编文件。

在 CodeVisionAVR 中，在线汇编有两种格式，一种是使用#asm 和#endasm 预处理命令来说明它们之间的代码为汇编语言，程序如需访问 DDRB 的 D3 位，可以使用下述语句：

```
#asm
sbi 0x17, 3
nop
cbi 0x17 , 3
#endasm
```

另外一种方式和 ICCAVR 有点类似，使用#asm("string")的形式，如上述程序我们可以改写为：

```
#asm("sbi0x17,3\n nop\n cbi 0x17,3")
```

其中符号“\n”表示汇编指令换行。要注意这种方式与 ICCAVR 的区别，在 CVAVR 中，这种方式的行尾不能加“;”，而 ICCAVR 和 GCCAVR 的在线汇编则必须以“;”号结尾。

我们以从 PORTD 读入数据为例子来简单介绍 GCCAVR 的在线汇编：

```
asm("in %0,%1":"=r"(value):"I"(PORTD):)
```

由上可以看出 GCCAVR 在线汇编由以下 4 个部分组成：

1．汇编指令本身，以字符串"in %0, %1"表示；

2．由逗号分隔的输出操作数，本例为"=r"(value)

3．由逗号分隔的输入操作数，本例为"I"(PORTD)

4．Clobber 寄存器

在线汇编的通用格式为：

```
asm(代码 : 输出操作数列表 : 输入操作数列表 : clobber 列表);
```

实例为：

```
asm(code:output operand list:input operand list:clobber list);
```

下面详细介绍 ICCAVR 中在线汇编的格式和使用方法，具体的汇编指令及作用，请参考 AVR 单片机的手册及相应的书籍。

2.12.1　汇编界面

ICCAVR 中在线汇编的语法为：

```
asm("<string>");
```

其中“<string>”可以被用来指定多个汇编声明、多个不同的声明可以被“\n”符号分隔成新的一行。为了在汇编中使用(访问)一个 C 的变量，可使用%<变量名>格式：

```
unsigned char uc;
asm("mov %uc,R0\n"
    "sleep\n");
```

说明：

1．关于引用全局变量

要引用一个全局变量，可以在变量名前加一个“%”号，用这种方法引用工程中的全局变量时要注意：如果在汇编中引用的全局变量为本文件定义的全局变量或者在本文件中已用 extern 说明过，则可直接编译通过，如果引用工程中其他源文件定义的全局变量且在本文件中没有用 extern 说明过，则 C 编译器将直接使用该符号(全局变量)，该符号的值由链接器决定，在编译时可能会产生一个警告(ASM 中的符号没有定义)，如果程序设计人员能够肯定该符号已在其他文件中定义过，可以不必理会该警告。

2．关于引用局部变量

只有汇编语句所在函数内部定义的自动变量，才能通过该方法引用，其他函数中定义的局部变量，不能通过该方法引用，因为编译器在编译汇编语句时，其他函数定义的局部变量可能是不存在或者不可读的。因此要在汇编中引用一个局部变量，必须在该变量的有效作用域内才能引用。

C 语言中的名称在汇编文件中是以下划线为前缀的，如函数 main( )在汇编中是以 _main ( )被引用的。在上例中，使用变量 uc 也可以改为以下形式：

```
asm("mov _uc,R0\n"
    "sleep\n");
```

用这种方式引用全局变量或局部变量的有效范围与用“%”方式相同。

汇编文件中的名称(标识符)有效长度为 30 个字符，一般符号(用一个冒号定义的符号)只能在本汇编模块中被引用，如果需要在汇编模块之外的 C 程序中使用，必须定义成全局符号(用两个冒号定义)，并且在 C 中必须先声明再使用。例如：

```
_foo::.blkw                          //定义一个全局变量
```

因为 ICCAVR 编译器不知道在汇编中定义的全局变量，因此在 C 文件中声明如下：

```
extern int foo;
```

ICCAVR 中提供的 WDR( )、SEI( )、CLI( )、NOP( )之类无操作数的函数(准确的说应称之为宏)，就是以在线汇编的形式实现的，在预编译时直接将这些宏用相应的在线汇编指令代替，macros.h 头文件中定义了以下的宏：

```
#define WDR() asm("wdr")
#define SEI() asm("sei")
#define CLI() asm("cli")
#define NOP() asm("nop")
#define _WDR() asm("wdr")
```

```
#define _SEI() asm("sei")
#define _CLI() asm("cli")
#define _NOP() asm("nop")
```

2.12.2　在线汇编中函数调用规则

如果是使用 C 语言来编写函数，函数的调用是由 C 按规则自动完成的，如果使用汇编语言编写函数，就必须注意函数调用及返回的规则，并且在汇编中按照该规则使用相应的寄存器进行参数传递：

1．传递参数和返回值所使用的寄存器

第一个参数若是整型则通过 R16/R17 传递，第二个参数则通过 R18/R19 传递，剩下的参数通过软件堆栈传递。如果参数是长整型或浮点数则通过 R16/R17/R18/R19 传递；其余参数通过软件堆栈传递。比整型参数小的参数(如 char 型)扩展成整型(int)长度传递。如果 R16/R17 已传递了第一个参数，而第二个参数是长整型或浮点数，则第二个参数的低半部通过 R18/R19 传递，而高半部通过软件堆栈传递。

整型返回值是通过 R16/R17 返回，而长整型或浮点数返回则通过 R16/R17/R18/R19 返回，返回值比整型小的参数(如 char 型)扩展成整型(int)长度返回。

2．保护寄存器(Preserved Registers)

C 中的函数调用汇编函数，被调用的汇编函数必须保证以下寄存的内容不被改变，如果汇编函数需要这些寄存器，就必须在汇编中插入保护和恢复这些寄存器的指令；在汇编中调用了 C 函数就不必在汇编中插入保护和恢复下列寄存器的指令，这些寄存器的保护和恢复由被调用的 C 函数负责。

R28/R29(Y 指针)，作为结构指针使用。

R10/R11/R12/R13/R14/R15/R20/R21/R22/R23。

以上寄存器的内容在被汇编语言函数调用后必须保持不变。

3．可变寄存器(Volatile Registers)

C 中的函数调用汇编函数，被调用的汇编函数不必插入保护和恢复以下可变寄存器的指令，只需按照相应的规则进行参数的传递和返回即可，但在汇编中调用了 C 函数，就必须在汇编中插入保护和恢复下列寄存器的指令，因为被调用的 C 函数不进行这些保护和恢复的操作：

R0/R1/R2/R3/R4/R5/R6/R7/R8/R9/R24/R25/R26/R27/R30/R31 和 SREG。

4．中断服务函数

在中断处理函数中，必须保护在中断中使用的全部寄存器(包括 SREG)，并且在退出中断服务函数前恢复这些寄存器。

2.12.3 汇编语法

1．汇编符号

在 ICCAVR 中汇编符号与 C 语言中标识符的命名规则相同，只能由字母、数字和下划线组成，并且第一个字符必须为字母或下划线，ICCAVR 能够识别的名称最长为 30 位英文字符(允许超过 30 个字符，但超过的部分不予识别)。在汇编中使用 Atmel 标准文档中的汇编指令和汇编伪指令可以不必区分大小写，ICCAVR 增补的汇编伪指令也可以不区分大小写，但在使用 C 中定义的标识符和汇编中定义的符号，都必须区分大小写。

2．数的表示方法

在汇编中，一个数的表示方法与 C 中整型常量的表示方法相同，可以有以下四种形式表示：

(1) 十进制：如：0、314、-123。

(2) 八进制：以 0 开头的数是八进制，如：0314。

(3) 十六进制：以 0x 或 0X 开头的数是 16 进制数，如 0xb14f。

(4) 二进制：以 0b 开头的数是二进制，如 0b10010110。

3．汇编文件格式

ICCAVR 中的汇编文件必须是一个 ASCII 纯文本文件，扩展名必须使用“.s”，而且该汇编文件必须加入工程中，与工程中的其他文件一起编译。

汇编文件的每一行应该是如下的格式：

```
[label: [:]] [command] [operands] [;comments]
```

label：表示标号，一个冒号表示局部符号，两个冒号表示全局符号。标号的值是程序中某一点 PC 计数器的值。

command：表示操作码(指令码)。操作码应该是 AVR 的指令、伪指令或宏。

operands：表示操作数。操作数是指令所需要的参数。

comments：表示注释。注意在 C 文件中注释可以用“//”或“/*….*/”两种注释方式，而在汇编文件中只能使用“;”或“//”两种注释方式。

在上式中每项之间必须用一个或多个空格分开，“[”和“]”框起来的部分表示为可选项，编译系统对注释部分不进行处理。

在 ICCAVR 中使用的汇编指令除了 xcall 和 xjmp 外均与标准 Atmel 文档中的指令相同。其中 xcall 指令只应用于 Mega 类芯片，ICCAVR 根据目标标号的地址把它编译为 rcall 或 call 中的一个；对于 xjmp 也是应用于 Mega 类芯片，ICCAVR 根据目标标号地址把它编译为 rjmp 或 jmp 中的一个。

4．表达式

在指令后的操作数可以是表达式，直接寻址模式是一个简单的表达式，表示如下：

```
lds R10,0x0200
```

在 ICCAVR 的在线汇编中，允许使用表达式、变量名称或标号名作操作数，如 a 和 b 均为 C 中的变量名，则下面的表达式是合法的：

```
lds R10 %a
```

或：

```
lds R10 %a+0x10
```

在表达式只能出现一个重定位符号(如变量名)，下面的表达式是非法的：

```
lds R10 %a+%b
```

因为同时使用了 a 和 b 两个重定位符号(变量名)。

注意：在表达式中只有“+”(加号)可以用于重定位符号的计算，如前面的例子，除了加号以外的其他运算符只能用于常数或文件中定义的符号的运算，如：

```
lds R10 (0x10+NULL)/2
```

一个复杂的表达式可以表述为：

```
expr:term |
     (expr)
     unop expr
     expr binop expr
     term: . ;                    圆点代表当前程序计数器(PC)的值
               name |
               #name
```

5．运算符

汇编语言中运算符的优先级与 C 中相同，表 2.9 列出了各种运算符和它们的优先级。

表 2.9　运算符及优先级

运　算　符	功　　能	类　　型	优　先　级
*	乘法	二进制	10
/	除法	二进制	10
%	取模	二进制	10
<<	左移	二进制	5
>>	右移	二进制	5
^	按位异(或 XOR)	二进制	4
&	按位与(或 AND)	二进制	4
|	按位或(或 OR)	二进制	4
-	负号	单目运算符	11

续表

运 算 符	功 能	类 型	优 先 级
~	取补运算	单目运算符	11
<	取低字节	单目运算符	11
>	取高字节	单目运算符	11

6. 圆点(.)

如果圆点出现在表达式中，那么用当前程序计数器的值来代替圆点(dot)。

2.12.4 ICCAVR 增补的汇编伪指令

1. .area <name> [(attributes)]

定义代码或数据装入的内存区域，链接器将所有使用同一名称的区域集合在一起，并且根据它们的属性进行连接或覆盖。

属性有两类，一类为 abs(绝对定位区域)或 rel(重定位区域)；另一类为 con(连接定位)或 ovr(覆盖定位)。

绝对定位区域的起始点地址是在汇编文件中由用户指定的，而重定位区域的起始地址是由链接器的一个命令选项来指定的。对带连接定位属性的区域，链接器连接这个区域到另一个同名区域后面；对带有覆盖定位属性的区域，对于每一个文件，链接器都是从同一地址开始安排区域。

下面举例说明它们的区别：

```
file1.o:
.area text      ;10 bytes，调用 text_1
.area data      ;10 bytes
.area text      ;20 bytes，调用 text_2
file2.o:
.area data      ;20 bytes
.area text      ;40 bytes，调用 text_3
```

text_1 和 text_2 在这个例子中是我们所起的名称，实际上它们在编译后是不会获得单独名称的。假设 text 区域的起始地址设置为 0。如果这个 text 区域有“con”属性，那么 text_1 将从 0 开始，text_2 从地址 10 开始，而 text_3 从 30 地址开始，如果这个 text 区域有“ovr”属性，那么 text_1 和 text_2 将分别地从 0 和 10 开始，但对 text_3，由于它在另外一个文件中定义，所以它也从 0 地址开始。

所有同名的区域必须有相同的属性，即使它们用在不同的模块中。下面是具有合适属性的部分例子：

```
.area foo(abs)
.area foo(abs,con)
.area foo(abs,ovr)
.area foo(rel)
.area foo(rel,con)
.area foo(rel,ovr)
```

2．ASCII 字符串和 ASCIZ 字符串

ASCII 字符串：用这个伪指令用于定义 ASCII 类型的字符串，字符串必须包含在一对分隔符之中。ICCAVR 允许用户设计定"("和")"、"["和"]"、"{"和"}"、"<"和">"作为字符串的分隔符，只要前后两个分隔符匹配就可以了，在两个分隔符之内可以是任意可打印的 ASCII 字符，以及下列以“\”开始的 C 中的转义字符。

```
\e：ESC
\b：退格
\f：换页
\n：换行
\r：回车
\t：TAB
\<最多三个八进制数>：字符的值等于此八进制数的值
```

ASCIZ 字符串：用这个伪指令用于定义 ASCIZ 类型的字符串，使用“”作为字符串的分隔符，编译器自动在字符串的结尾插入一个 NULL("\0")，其余的与 ASCII 字符串相同。如：

```
.asciz "Hello World\n"
asciz "23\n456"
```

3．下面的伪指令定义常数，分别表示字节常数(1 个字节)、字常数(2 个字节)和双字长数(4 个字节)。

```
.byte <expr> [,<expr>]*
.word <expr> [,<expr>]*
.long <expr> [,<expr>]*
```

注意：

(1) 双字常数(.long)只能用作操作数，字节常数和字常数可以用于重定位表达式。

(2) 在 AVR 单片机中，字常数和双字常数是以高低字节倒置的格式输出的。

例如：

```
.byte 1,2,3
.word label,foo
```

4. 下面的伪指令是保留空间而没有给它们赋值，指令后面的数分别是指保留的字节、字或双字的数量。

```
.blkb <value>
.blkw <value>
.blkl <value>
```

例：.blkb 100　　保留 100 个字节的空间

5. 文本替代符

```
.define <symbol> <value>
```

上面的伪指令定义了一个文件替代符，使用“symbol”时，无论在表达式中或在标号中，都可以被“value”所替代。

例如：

```
.define quot R15
mov quot,R16
```

相当于以下指令：

```
mov R15,R16
```

6. 条件汇编

```
.if <symbol name>
.else
.endif
```

上述三个伪指令定义了一个条件汇编语句，如果条件<symbol name>为真(非 0)，则执行.if 和.else 之间的指令，如果条件<symbol name>为假(0)，则执行.else 和.endif 之间的指令。如：

```
.if cond
lds R10,a
.else
lds R10,b
.endif
```

在上例中，如果 cond 不等于 0，则将 a 装入 R10，如果 cond 等于 0，则将 b 装入 R10。

注意：条件语句最多只可以嵌套 10 层，而且.else 及其后面的语句可以省略。

7. 宏定义

```
.macro <macroname>;                    定义一个宏
.endmacro;                             取消一个宏定义
<macro> [<arg0> [,<args>]*];           调用宏
```

除了另外一个宏语句外，任意汇编语句可以是宏的组成部分。在宏表达式内部@digit(digit 由 0~9 的数字代替)是宏被调用时相应的宏参数，可以指定多个参数。调用宏是在操作码的位置放置宏名，后面跟上相应的参数。汇编器使用组成宏的语句来替换宏，同时用相应的宏参数来扩展。注意，定义的宏名称不能与汇编指令及汇编伪指令名称相同。

例：

```
.macro foo;                            定义宏名“foo”
lds @0,a
mov @1,@0
.endmacro
```

调用宏 foo 需要两参数，如：

```
foo R10,R11
```

等同于下面的语句：

```
lds R10,a
mov R11,R10
```

8. 定义一个符号等于常数值

```
<symbol>=<value>
```

如：

```
foo=5;                                 定义 foo 代表常量 5
```

9. 文件包含

```
.include "<filename>"
```

如果指定的目录中该文件不存在，汇编器将到系统默认的路径中寻找该文件。

如：

```
.include "math.h"
```

10. 设定起始位置

```
.org <value>
```

设定程序计数器 PC 的值为“value”，这个伪指令中在带有“abs”属性的区域内有效，注意“value”是字节地址。

例如：

```
.area interrupt_vectors(abs)
org 0xFFD0
dc.w reset
```

11. 定义一个符号为全局变量

```
.globl <symbol> [, <symbol>]*
```

将一个符号定义为全局符号，这和跟着两个冒号的标号效果相同。

2.13 C 源程序常见错误分析

1. C 源程序出错有以下两种情况：

(1) 语法错误。指编程时违背了 C 语言语法的规定，对这类错误，编译程序一般都能够给出“出错信息”，并且告知哪一行出错及出错的类型，只要仔细检查，是可以很快发现错误并排除的。

(2) 逻辑错误。程序并无违背语法规则，但程序执行结果与设计不符。这是由于程序设计人员输入的源程序与设计人员的本意不相同，即出现了逻辑上的混乱。

例如：

```
unsigned char i=1;
unsigned int sum=0;
while (i<=100)
    sum=sum+i;
    i++;
```

在上例中，设计者本意是想求从 1 到 100 的整数和，但是由于循环语句中漏掉了大括号，使循环变为死循环而不是求累加。对于这种错误，C 编译通常都不会有出错信息(因为符合 C 语法，但有部分编译系统会提示有一个死循环)，要查找这类的逻辑错误，要求程序设计者有丰富的设计经验(不会有类似的错误)和有丰富的排错经验(通过仿真能够很快发现问题)，这需要经过较长时间的学习和实践才能够掌握。

2. 初学者在编写C源程序时的常见错误及分析

(1) 忘记定义变量就使用

例如：

```
void main (void)
    {
    x=3;
    y=x;
    }
```

在上式中看似正确，实际上却没有定义变量x和y的类型。C语言规定，所有的变量必须先定义，后使用。因此在函数开头必须有定义变量x和y的语句，应改为：

```
void main (void)
    {
    int x,y;
    x=3;
    y=x;
    }
```

(2) 变量没有赋初值就直接使用

例如：

```
unsigned int addition (unsigned int n)
    {
    unsigned int i;
    unsigned int sum;
    for (i=0;i<n;i++)
    sum+=i;
    return (sum);
    }
```

上例中本意是计算1到n之间整数的累加和，但是由于sum没有赋初值，sum中的值是不确定的，因此得不到正确的结果。应改为如下形式：

```
unsigned int addition (unsigned int n)
    {
    unsigned int i;
    unsigned int sum=0;
    for (i=0;i<n;i++)
    sum+=i;
```

```
    return (sum);
    }
```

或者将 sum 定义为全局变量(全局变量在初始化时自动赋值，为 0)。

```
unsigned int sum;
unsigned int addition (unsigned int n)
    {
    unsigned int i;
    for (i=0;i<n;i++)
    sum+=i;
    return (sum);
    }
```

(3) 输入输出的数据类型与所用格式说明符不一致

例如：

```
void main (void)
    {
    int a=3,b=4.5;
    printf("%f %d\n",a,b);
    }
```

在上例中，a 与 b 变量错位，但编译时并不给出出错信息，输出结果也必定不正确。当输入输出的数据类型与所用格式说明符不一致时，C 编译器并不会按赋值的规则进行转换(如把 3 转换成 3.0，把 4.5 转换成 4)，而是将内存单元中的数据按格式符要求的宽度直接输出，如 b 占 4 个字节却用“%d”说明，则只有最后两个字节中的数据当成一个整数输出，a 占 2 个字节却用“%f”说明，则将 a 地址前两个字节(并不属于 a)与变量 a 的两个字节合并成一个 4 个字节的浮点数输出。

(4) 没有注意数值的有效范围

8 位单片机适用的 C 编译器，对字符型变量均分配一个字节的存储单元，因此有符号字符型变量的数值范围为-128~127，无符号字符型变量的数值范围为 0~255。其他类型的变量均有一个有效的范围，这里就不再一一列举，请读者参见相应编译器的使用手册。

例如：

```
void main (void)
    {
    char x;
    x=300;
    }
```

在上例中，有很多读者会认为 x 的值就是 300，实际上却是错误的。

300 的二进制为 0b100101100，赋值给 x 时，只将最后的 8 位赋值给 x，高位截去，因此 x 的值实际上为 0b101100(即整数 44)。

如果将 500 赋给一个有符号的字符型变量时，变量的值还会变成负数，由读者自行分析原因。

(5) 输入变量时忘记使用地址符号

例如：

```
void main (void)
    {
    int a,b;
    scanf ("%d %d",a,b);
    }
```

应改为：

```
void main (void)
    {
    int a,b;
    scanf ("%d%d",&a,&b);
    }
```

(6) 输入时数组的组织方式与要求不符

例如以下语句：

```
scanf ("%d %d",a,b);
```

如果输入数据格式为：

```
3,4
```

则是错误的，两个数据之间应用空格来分隔，应为：

```
3 4
```

(7) 误把“=”作为关系运算符“等于”

在数学和其他高级语言中，都是把“=”作为关系运算符“等于”，因此编写 C 源程序时容易将程序误写为：

```
if (a=b)
     c=0;
else
     c=1;
```

在上例中，本意是如果a等于b，则c=0，否则c=1。但C编译系统却认为程序员要将b赋值给a，并且如果a不等于0，则c=0，如果a等于0，则c=1，这与原来的意图完全不同。应将条件表达式更改为：

```
a==b
```

(8) 语句后面漏加分号

C语言规定语句末尾必须有分号，分号是C语句不可缺少的一部分。

例如：

```
void main (void)
    {
    unsigned int i,sum;
    sum=0;
    for (i=0;i<10;i++)
        {sum+=i}
    }
```

很多初学者认为用大括号括起来就不必加分号，这是错误的，即使该语句用大括号括起来，也必须加入分号，特别是在复合语句中，初学者往往容易漏写最后一个分号。并且当漏写分号而导致编译出错时，几乎所有的C编译器均将光标停留在漏写分号的下一行。

上例应改为如下形式：

```
void main (void)
    {
    unsigned int i,sum;
    sum=0;
    for (i=0;i<10;i++)
    {sum+=i;}
    }
```

(9) 在不该加分号的地方加了分号，在该加分号的地方忘了加分号

例如，在定义头文件时：

```
#include "io8515v.h";
```

由于伪指令不是C程序语句，因此后面不能加分号。初学者也常在判断语句的条件表达式后面误加入分号，例如：

```
void main (void)
    {
    unsigned int i,sum;
```

```
    sum=0;
    for (i=0;i<10;i++);
    sum+=i;
    }
```

在上例中，在 for 的表达式后面中加入分号，则 C 编译器认为循环体是一个空操作，这与设计者的本意不符。

(10) 对应该加括号的复合语句，忘记加括号

例如：

```
unsigned char i=1;
unsigned int sum=0;
while (i<=100)
   sum=sum+i;
   i++;
```

我们在前面举过这个例子，应改为：

```
unsigned char i=1;
unsigned int sum=0;
while (i<=100)
{
sum=sum+i;
i++;
}
```

(11) 括号不配对

当一个复合语句中使用多层括号时，常会出现括号不配对的错误。

例如：

```
while ((c=getchar ()!='a')
   putchar(c);
```

(12) 没有注意 C 语言区分大小写字母

例如：

```
void main (void)
    {
    int Exemple;
    exemple=0;
    }
```

在上例中，C 编译系统会提示变量 exemple 没有定义。应改为：

```
void main (void)
    {
    int Exemple;
    Exemple=0;
    }
```

(13) 引用数组元素时误用圆括号

```
void main (void)
    {
    int i,a[10];
    for (i=0;i<10;i++)
         scanf ("%d",a(i));
    }
```

C 编译器在通常情况下都能够指出这个错误，如果源程序中恰好定义了一个 a( )函数，则可以通过编译，但结果不正确。

(14) 引用数组元素超界

例如：

```
void main (void)
    {
    int i,a[5]={1,2,3,4,5};
    for (i=1;i<=5;i++)
         printf ("%d",a[i]);
    }
```

上例中，本意是想输出数组 a 的全部元素，实际上，定义的数组 a[5]中，只有 a[0]~a[4]共 5 个元素，并不存在 a[5]元素。应改为如下形式：

```
void main (void)
    {
    int i;
    int a[5]={1,2,3,4,5};
    for (i=0;i<5;i++)
         printf ("%d",a[i]);
    }
```

(15) 对二维或多维数组定义和引用的方式不对

例如：

```
void main (void)
    {
    int a[5,4],
    /*  其他程序  */
    }
```

在 C 语言中，对二维数组和多维数组在定义和引用时必须将每一维数组中的数据分别用方括号括起来，因此定义一个二维数组，应改为：

```
int a[5][4];
```

(16) 误以为数组名代表整个数组

例如：

```
void main (void)
    {
    int a[5]={1,2,3,4,5};
    printf ("%d,%d,%d,%d,%d",a);
    }
```

在上例中，本意是输出数组 a 中的全部元素，但是数组名 a 却只是代表数组的首地址，并不能代表数组中的所有元素，因此并不能得到所需的结果，应改为：

```
void main (void)
    {
    int a[5]={1,2,3,4,5};
    printf ("%d,%d,%d,%d,%d",a[0],a[1],a[2],a[3],a[4]);
    }
```

(17) 混淆字符数组与字符指针的区别

例如：

```
void main (void)
    {
    char str[10];
    str="ICCAVR";
    printf ("%s\n",str);
    }
```

在上例中，编译出错。因为 str[10]是一个数组，str 代表数组名，是一个常量，不能被

赋值，可将 str 改为指针变量，将字符串"ICCAVR"的首地址赋值给指针变量 str，然后在 Printf 函数中输出字符串。如下：

```
void main (void)
    {
    char *str;
    str="ICCAVR";
    printf ("%s\n",str);
    }
```

在上例中，如果坚持要使用数组，一种方式为初始化时赋值，另一种只能在程序中一个一个元素进行赋值。分别如下：

```
void main (void)
    {
    char str[10]="ICCAVR";
    printf ("%s\n",str);
    }
```

或

```
void main (void)
    {
    char str[10];
    str[0]='I';str[1]='C';str[2]='C';str[3]='A';str[4]='V';
    str[5]='R';str[6]='\0';str[7]='\0';str[8]='\0';str[9]='\0';
    printf ("%s\n",str);
    }
```

(18) 在引用指针变量之前没有对它赋值

```
void main (void)
    {
    char *p;
    scanf ("%s",p);
    }
```

没有给指针变量赋值就使用它，由于指针变量 p 的值不确定，因此有可能误指向有用的存储空间，导致程序运行出错。应当改为：

```
void main (void)
```

```
    {
    char *p,str[20];
    p=str;
    scanf ("%s",p);
    }
```

(19) switch 语句的各分支漏写了 break 语句

例如：

```
switch (time)
    {
    case 0 : a=0;
    case 1 : a=1;
    case 2 : a=2;
    defult : a=3;
    }
```

上例中，本意是根据 time 的值来决定 a 的值，但是上例中无论 time 为何值，最后程序执行的结果都一样(a=3)，因为漏写了 break 语句，程序将从相应的 case 开始顺序执行，并且连续执行至最后一条语句结束，应改为：

```
switch (time)
    {
    case 0 : a=0; break;
    case 1 : a=0; break;
    case 2 : a=2; break;
    defult : a=3;
    }
```

(20) 混淆了字符和字符串的表示形式

例如：

```
char sex;
sex="M";
```

由于 sex 是字符变量，只能存放一个字符，用单引号括起来的是字符常量，才能赋值给一个字符型变量，而用双引号括起来的是字符串常量，它包括 “M” 和 “\0” 两个字符常量，无法存放到字符变量中。应改为：

```
char sex;
```

```
sex='M';
```

(21) 使用自加(++)和自减(--)时出错

例如，想利用下述语句输出 a[1]的值。

```
void main (void)
    {
    int *p,a[5]={0,1,2,3,4};
    p=a
    printf ("%d",*p++);
    }
```

在上例中，“*p++”本意是指 p 加 1，即指向第 1 个元素 a[1]外，然后输出第一个元素 a[1]的值 1。但实际上是先执行 p++，先使用 p 的原值，使用后再加 1，因此实际执行结果是输出 a[0]的值。应改为：

```
void main (void)
    {
    int *p,a[5]={0,1,2,3,4};
    p=a;
    printf ("%d",*(++p));
    }
```

说明：要注意 p++、*p++、*(++p)和(*p)++的区别，详见“2.7.4 数组的指针和指向数组的指针变量”一节的介绍。

(22) 所调用的函数在调用语句之后定义，但在调用之前没有说明

例如：

```
void main (void)
    {
    float x=2.0,y=6.0,z;
    z=max(x,y);
    printf ("%f",z);
    }
float max (float x,float y)
    {
    return (x>y ? x:y);
    }
```

在上例的程序中，max 函数在 main 函数之后定义，在调用之前又没有说明，因此出错。

应在调用前对函数进行说明：

```
void main (void)
    {
    float max (float x,float y);
    float x=2.0,y=6.0,z;
    z=max(x,y);
    printf ("%f",z);
    }
float max (float x,float y)
    {
    return (x>y ? x:y);
    }
```

也可以将函数 max( )在函数 main( )之前定义。

(23) 误认为形参值的改变会影响实参的值

例如：

```
void main (void)
    {
    int x=3,y=4;
    swap(x,y);
    printf ("%d,%d",x,y);
    }
int swap (int x,int y)
    {
    int z;
    z=x;x=y;y=z;
    }
```

在上例中，设计者的意图本是利用 swap 函数交换 x 和 y 的值，但是由于实参和形参之间单向传递，在函数 swap 中改变了 x 和 y 值，main 函数中的 x 和 y 是不会改变的，可以改为使用指针的形式：

```
void main (void)
    {
    int x=3,y=4;
    int *p1,*p2;
    p1=&x ; p2=&y;
    swap(p1,p2);
    printf ("%d,%d",x,y);
```

```
    }
int swap (int *p1,int *p2)
    {
    int z;
    z=*p1;*p1=*p2;*p2=z;
    }
```

说明：虽然函数 swap 在调用函数之后定义，而且在函数 main 调用之前又没有说明，但是由于 swap 返回值为整型，C 语法规定返回值为整型的函数在调用之前可以不必说明，因此本例是符合 C 语法规定的。

(24) 函数的实参和形参类型不一致。

如：

```
void main (void)
    {
    int x=3,y=4;
    int *p1,*p2;
    p1=&x ; p2=&y;
    swap(p1,p2);
    printf ("%d,%d",x,y);
    }
int swap (int p1,int p2)
    {
    int z;
    z=p1;p1=p2;p2=z;
    }
```

C 语言要求实参与形参的类型一致，一个为指向整型变量的指针，另一个为整型变量，类型不同，因此编译出错，修改以后请参见前例。

(25) 不同类型的指针混用

例如：

```
void main (void)
    {
    char x=3,*p1;
    unsigned char *p2;
    p1=&x;
    p2=p1;
    printf ("%d,%d",*p1,*p2);
    }
```

在上例中，设计者本意是想将指针 p1 的值赋给指针 p2，但是由于 p1 与 p2 所指向的类型不同，会导致编译出错，在赋值时必须进行强制类型转换。如：

```
void main (void)
    {
    char x=3,*p1;
    unsigned char *p2;
    p1=&x;
    p2=(unsigned char *)p1;
    printf ("%d,%d",*p1,*p2);
    }
```

要注意，在上例中，将指向有符号字符型变量的指针 P1 强制转换为指向无符号变量的指针后赋给 p2，如果 x 的值是正数，则输出结果中，*p1 和*p2 的值是相同的，如果 x 的值是负数，则输出结果中，*p1 和*p2 的值是不同的，但 x 的值及其类型并没有改变(本例中 x=3，还是 char 类型)。

指向不同类型的指针变量进行强制转换后赋值，在 C 中是常用的，例如，在 ICCAVR 中，用 malloc 函数开辟数据存储单元，函数的返回值是一个空指针(void)，需要强制转换成指向一个结构体的指针类型，常用来组成一个链表：

```
struct str
    {
    int a;
    struct str *next;
    }*p1;
/*   用户程序  */
p1=(struct str *)malloc (size_t,size)
```

(26) 混淆数组与指针变量的区别

例如：

```
void main (void)
    {
    int i,a[5]={1,2,3,4,5};
    for (i=0;i<5;i++)
         printf ("%d",a++);
    }
```

在上例中，设计者意图通过 a++的操作，输出数组 a 的不同元素。由于 C 规定数组名代表数组的首地址，它的值是一个常量，因此用 a++是错误的，应改为用指针变量来实现，如：

```
void main (void)
    {
    int i,a[5]={1,2,3,4,5};
    int *p;
    p=a;
    for (i=0;i<5;i++)
         printf ("%d",p++);
    }
```

或：

```
void main (void)
    {
    int a[5]={1,2,3,4,5};
    int *p;
    for (p=a;p<a+5;p++)
         printf ("%d",p);
    }
```

(27) 混淆结构体类型与结构体变量的区别

例如：

```
struct worker
    {
    unsigned char num;
    char name[20];
    };
worker.num=1;
```

在上例中，只是说明了一种 struct worker 的结构，但是 C 编译器并没有为这种类型的结构体变量开辟存储空间，因此不能对结构体中的元素赋值。只有使用该结构体类型定义了一个该类型的变量后，才能对这个变量的元素赋值，上例应改为：

```
struct worker
    {
    unsigned char num;
    char name[20];
    };
struct worker work;
work.num=1;
```

2.14　C 源程序调试

所谓源程序调试是指对程序的查错和排错，一般应经过以下几个步骤：

1．进行静态检查。

在写好一个程序后，不要匆忙用编译器编译，应对写好的源程序进行人工检查，这一步是十分重要的，它能发现程序设计人员由于疏忽而造成的大多数错误。多数人往往对这一步并不重视，总希望把查错的工作推给编译系统去做，但是编译器并不是万能的，并不一定能指出源程序中全部的语法错误。作为一个优秀的设计人员，应当养成严谨的作风，每一步都要严格把关，不能把问题留给后面的工序。

为了更有效地进行人工检查，所编写的程序中应力求做到以下几点：

(1) 应当采用结构化程序方法编程，以增加可读性。

(2) 尽可能多加注释，以帮助理解每段程序的作用。

(3) 在编写复杂的程序时，不要将全部的语句都写在 main 函数中，而要多利用函数，用一个函数来实现单独的功能，既易于阅读也便于调试。各函数之间除用参数传递数据这一渠道外，能够不用其他的渠道就尽量不用，数据间应尽量减少耦合的关系，以便于分别检查和处理。

(4) 对于每一个函数模块，建议加上如下所示的详细说明：

```
/*************************************
**工程名称：这是一个函数说明示例
**编译器类型及版本：ICCAVR 6.26C
**日期：2002.10.15
**创建人：沈文
**所属单位：福建武夷通信设备厂

**芯片类型：ATmega128L
**时钟频率：14.7456MHz
**硬件接口说明：
  Vcc=+5V，使用 PC 机键盘输入，一个 320×240LCD，一块 MAX232 集成电路

**函数功能：在 LCD 上画一个圆
**函数输入参数及说明：unsigned char x，unsigned char y，unsigned char r
                    x 和 y 为圆心的座标，r 为圆的半径
**函数返回值：无
**在本函数以外定义的变量：无
**调用的函数说明：调用了一个画点函数 pset()
```

```
**说明:

**修改人:
**修改日期:
**修改说明:
*************************************/
```

在本书中，为了节约篇幅，没有使用以上的函数说明，但建议读者在编写程序时能够使用比较全面的函数说明，这样便于以后的维护和升级。当然，上面给出的说明仅仅是个示例，读者不一定要完全照搬上面的说明。

2．在静态检查无误后，可进行上机调试．通常称上机调试为动态检查。

根据编译器提示的语法错误，提出编译器提示的全部错误(error)并一一改正，直到通过编译，生成下载文件或调试文件，还应该仔细检查编译器的警告(warning)信息，确认所有的警告信息并不会影响编译结果的正确性。有时，编译器的错误提示并非正确，而且出错的情况繁多且各种错误相互关联，因此要善于分析，找出真正的错误，而不能死抱提示的出错信息不放，钻牛角尖。

3．对于一些简单的小程序，可以直接下载到实际电路或实验器材进行调试，也可以使用硬件仿真器或进行软件仿真，对于 MCS-51 单片机，通常都是在 Keil uVision2 中进行硬件仿真或软件仿真，对于 AVR 单片机，通常都是在 AVR Studio 环境中进行硬件仿真或软件仿真。

通过初步仿真后应对完成的整个硬件进行测试，测试的目的是为了测试软硬件能否在各处复杂的情况下正常工作，在测试时应当尽可能地将程序中遇到的各种情况均测试到，力争将程序流程中的各分支都测试一次(这比较难)，至少应将可能的各种极限情况都进行测试。

对于一些复杂的程序，通常都应将其分解为若干个子函数模块并分别进行测试。对于初学者，如果有一个硬件仿真器，对于了解程序的运行过程和解决一些奇怪的错误是非常有帮助的。对于 ATtiny 和 AT90 系列的单片机可以选用 ICE200 型仿真器，对于 ATmega 系列的单片机，可以选用 Jtag 仿真器。由于仿真器都非常昂贵，而且仿真器能够仿真的芯片种类比较少，对于 ATmega 系列单片机，可以将代码写到 boot 区，利用串口直接进行仿真。对于初学者，也可以暂时不买硬件仿真器，先进行软件仿真，但软件仿真只能观察芯片的内部运行效果，与实际硬件运行情况还是有一定区别的。

程序的运行结果不对，大多属于逻辑错误，对这类错误往往要仔细检查和分析才能发现。在使用仿真排除的同时，还应将源程序与流程图(或伪代码)仔细对照，如果流程图正确，程序写错了，是很容易发现的。如果这样找不到错误，仿真又没有发现问题，就应检查流程图有无错误，程序中使用的算法有无问题。

总之，程序的调试是一项细致深入的工作，需要下功夫、动脑子、积累经验。在程序的调试过程中，往往能反映一个人的水平、经验和工作态度，希望读者能够予以重视。

第 3 章　ICCAVR 集成环境

3.1　ICCAVR 编译器的安装与注册

3.1.1　ICCAVR 编译器的安装

可以直接运行光盘上 setup.exe 安装程序进行安装，有以下几种方法：

方法 1：

(1) 打开“我的电脑”；

(2) 打开光盘驱动器所对应的盘符；

(3) 双击光盘中文件“setup.exe”的图标；

(4) 按照屏幕提示选定一个安装路径后进行安装。默认的安装路径为“c:\icc”，一般情况选默认的安装位置。

方法 2：

(1) 在开始菜单中选择“运行”命令；

(2) 在运行对话框中输入“<drive>:\setup.exe”；

注意，“<drive>”对应机器中的光盘驱动器盘符，如果光盘为 F 盘，则填入“f:\setup.exe”；

(3) 按“确定”键开始安装；

(4) 按照屏幕提示，选定一个安装路径后进行安装。

方法 3：

由于 ICCAVR 升级较快，光盘中的安装文件并不一定是最新的，可以到 ImageCraft 公司的网站上(http://www.imagecraft.com/software)或国内总代理双龙电子公司网站(http://www.sl.com.cn)下载最新版，将安装文件下载到电脑内一个子目录中再安装。

(1) 打开“我的电脑”；

(2) 打开存放下载升级版的文件夹；

(3) 双击文件夹中的“iccavrdem.exe”的图标；

(4) 按照屏幕提示选定一个安装路径后进行安装。

按上述方法安装后，在开始菜单中生成如图 3.1 的“ImageCraft Development Tools”一栏。安装后得到的只是一个演示版(标题栏如图 3.2 所示)，演示版只能使用 30 天，在 30 天之内没有使用限制(等同于标准版)，超过 30 天后，要注册才能使用，正版用户还要进行注册才可以得到一个无使用时间限制的正式版。ICCAVR 分为标准版和专业版，在专业版中

的文件配置检查在标准版中不能用，标准版的代码压缩功能有 30 次的使用限制，而专业版没有。

图 3.1 开始菜单中的 ICCAVR

图 3.2 ICCAVR 演示版的标题栏

注意：

1．旧版 ICCAVR 有安装路径的限制(即安装路径中不能有空格或中文，否则不能正常编译)，在 ICCAVR6.26 版及更新的版本中，对安装的路径则没有任何限制，可以安装在任意文件夹中。

2．如果硬盘上原先已安装过 ICCAVR 演示版且尚未到期(30 天免费试用期)，此时重新安装 ICCAVR，则软件会自动终止免费试用，因此在演示版 30 天的有效期内不能重装 ICCAVR。如果演示版到期后还需要继续使用演示版软件，只要格式化硬盘后重装 Windows 和 ICCAVR，又可以再使用 30 天。建议读者在电脑中安装 2 种以上的 Windows 版本(如：Windows 98、Windows 2000 和 Windows XP)，一个操作系统安装经常使用的软件，另一个操作系统安装 ICCAVR，这样在超过有效期后便于重装 Windows 和 ICCAVR，也有利于 ICCAVR 的稳定运行。当然，读者也可以利用其他方法在超过有效期后继续使用 ICCAVR。

3.1.2 ICCAVR 编译器的注册

对 ICCAVR 编译器的注册，可以分为以下两种情况。

1．对首次安装并且使用期没有超过 30 天的用户，按以下方式注册：

(1) 将正式版中附带的一张名称为“Unlock Disk”的软盘插入机器的软盘驱动器中；

(2) 启动 ICCAVR 编译器的集成环境 IDE；

(3) 在 IDE 的“Help”菜单中寻找标题为“Import License from Floppy”的一项并且单击，如图 3.3 所示；

(4) ICCAVR 提示关闭正在运行中的反病毒软件，如果要进行注册，请单击“OK”；如果不想注册，请单击“CANCEL”；

(5) ICCAVR 软件自动进行注册。当注册完成后会提示用户注册文件已从软盘移走。当确定并重新启动 ICCAVR 后，可以发现软件已经完成注册，标题栏提示为标准版(STANDARD)，如图 3.4 所示。

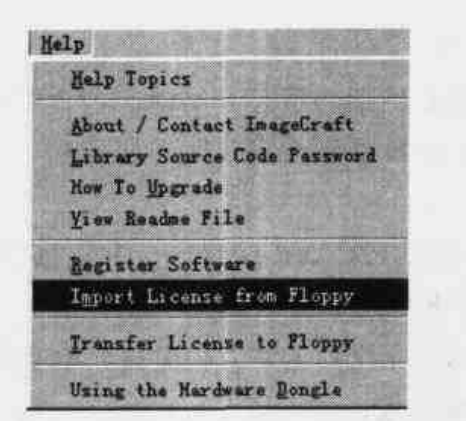


图 3.3　注册命令菜单

图 3.4　已完成注册的标题栏

2．对不是首次安装或使用时间已超过 30 天的用户，按以下方式注册：

(1) 将正式版中附带的一张名称为 Unlock Disk 的软盘插入机器的软盘驱动器中；

(2) 用户在程序启动时已不能进入 IDE 环境，而是出现一个提示注册的对话框，应该选择“YES”按钮；

(3) 这时会出现一个注册对话框，如图 3.5 所示，对话框上“Insert Floppy in Drive A: and Click:”边上有一个“Import License”按钮，单击该按钮；

(4) ICCAVR 提示关闭正在运行中的反病毒软件，如果要进行注册，请单击“OK”，如果退出注册，请单击“CANCEL”；

(5) ICCAVR 软件自动进行注册。当注册完成后会提示注册文件已从软盘移走。当确定并重新启动 ICCAVR 后，可以发现软件已经完成注册，标题栏提示为标准版(STANDARD)，如图 3.4 所示。

说明：ICCAVR 还允许使用 License 号注册或使用软件狗(如图 3.5 所示)，但是在国内，目前只能提供软盘注册这一种方式。

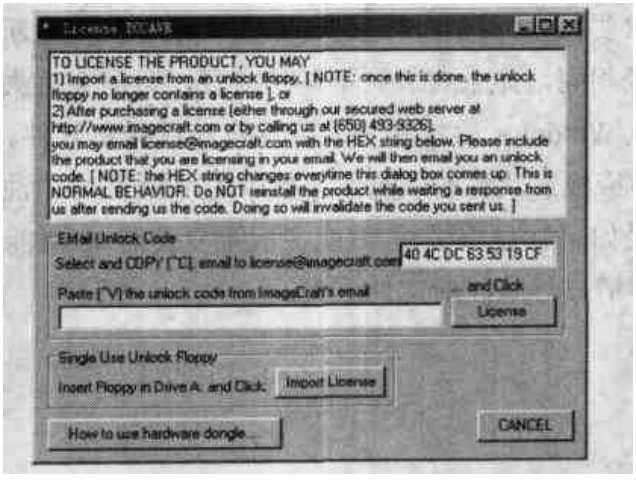


图 3.5　注册对话框

在使用软盘注册时应注意：

(1) “Unlock Disk”软盘在注册时应打开写保护，否则无法完成注册；

(2) 完成注册后，“Unlock Disk”软盘成为一张空盘，不可以在另一台机器上安装和注册。

当需要在不同的电脑中使用 ICCAVR 或在同一台电脑中重装 ICCAVR，并且重装在与原来不同的位置(文件夹)时，应该按以下步骤进行操作：

(1) 在软盘驱动器中插入一张空白的软盘，并打开软盘的写保护；

(2) 启动 ICCAVR 的集成环境；

(3) 在 IDE 的“Help”菜单中寻找标题为“Transfer License to Floppy”的一项，并且单击该项，如图 3.6 所示；

(4) ICCAVR 提示关闭正在运行中的反病毒软件，如要导出 License，请单击“OK”；如果不需要导出，请单击“CANCEL”；

(5) ICCAVR 将注册文件传送到软驱中的软盘上，已不能再进入 ICCAVR 的集成环境中，可以利用该软盘根据前面介绍的方法重新进行安装和注册。

图 3.6 导出 License 命令

如果仅仅是升级 ICCAVR，对于正版用户，只要将新版的 ICCAVR 覆盖安装原来的 ICCAVR 目录即可，只要原先为已注册过的标准版，则不需要重新注册，但一定要与原来安装的 ICCAVR 的路径相同，如果安装到不同的路径，可能会丢失注册文件。对于演示版用户，则只有重新安装 Windows 后才能安装新版的 ICCAVR 软件。

在导出注册文件时应注意软盘的质量，虽然 ICCAVR 在导出注册文件后会检查注册文件是否成功导出，但是如果软盘质量较差(或多次重复使用)，可能会出现成功导出注册文件却无法重新注册的事情，导致注册文件丢失。

3.2　ICCAVR 编译器的特点

3.2.1　ICCAVR 编译器简介

ICCAVR 是一种使用 ANSI 标准 C 语言来开发微控制器(MCU)程序的一个工具，它有以下几个主要特点：

1．ICCAVR 是一个综合了编辑器和工程管理器的集成工作环境(IDE)，是一个纯 32 位的程序，可在 Windows 95/Windows 98/Windows ME/windows NT/Windows 2000/Windows XP 下运行。

2．源文件全部被组织到工程之中，文件的编辑和工程的构筑也在 IDE 的环境中完成。编译错误在状态窗口中显示，用鼠标单击编译错误时，光标会自动跳转到编辑窗口中引起错误的那一行。这个工程管理器还能直接产生 INTEL HEX 格式文件的烧写文件和符合 AVR Studio 的调试文件(COFF 格式)。

3．ICCAVR 是一个 32 位的程序，支持长文件名。

4．ICCAVR 提供了全部的库源代码及一些简单的应用实例供初学者参考，特别是提供库源代码，对于用户理解库函数的参数及返回值等是非常有益的，并且用户能够根据库源代码对 ICCAVR 提供的库函数进行剪裁和扩充。

3.2.2　ICCAVR 中的文件类型及扩展名

文件类型是由它们的扩展名决定的，ICCAVR 的 IDE 和编译器可以使用以下几种类型的文件：

1．输入文件类型

.c 扩展名，表示是 C 语言源文件；

.s 扩展名，表示是汇编语言源文件；

.h 扩展名，表示是 C 语言的头文件；

.prj 扩展名，表示是工程文件，这个文件保存由 IDE 所创建和修改与整个工程的有关信息；

.a 扩展名，表示是库文件，它可以由几个库封装在一起，也可以创建或修改自定义的库。

2．输出文件类型

.s 扩展名，对应每个 C 语言源文件，由编译器在编译时产生的同名汇编输出文件；

.o 扩展名，汇编产生的同名目标文件，多个目标文件可以链接成一个可执行文件；

.hex 扩展名，INTEL HEX 格式文件，其中包含了程序的全部可执行代码；

.eep 扩展名，INTEL HEX 格式文件，包含了 EEPROM 的初始化数据；

.cof 扩展名，COFF 格式输出文件，用于在 ATMEL 的 AVR Studio 环境下进行程序调试；

.lis 扩展名，列表文件，列举出了源文件中的全部语句对应的汇编代码，但变量和代码没完成绝对定位。说明：如果使用文件编译方式，则生成该文件的.lis 文件，如果对整个工程进行编译，则每一个源文件均生成一个与该源文件同名的.lis 文件。

.lst 扩展名，列表文件，列举了含启动文件一起编译生成的全部汇编代码，是整个工程绝对定位后的完整列表文件。说明，如果使用文件编译方式，则生成该文件的.lst 文件，如果对整个工程进行编译，则只生成一个与工程同名的.lst 文件。

.mp 扩展名，内存映像文件，它包含了程序中有关符号及其所占内存大小的信息；

.cmd 扩展名，NoICE 2.xx 调试命令文件；

.noi 扩展名，NoICE 3.xx 调试命令文件；

.dbg 扩展名，ImageCraft 调试命令文件。

3.2.3 AVR 存储器的使用

ICCAVR 共包含三种不同类型的存储器空间，下面分别对其进行介绍。

1．程序存储器(Flash)

程序存储器是用于保存程序代码以及常数表和数据的初始值等的空间。ICCAVR 编译器可以生成一个对应程序存储器映像的输出文件(INTEX HEX 文件)，大部分编程器均支持用这个格式的文件对芯片编程。

由于 AVR 中的 X、Y、Z 指针为 16 位，有效的访问空间为 64KB，因此在 ICCAVR 编译器中，使用普通的 C 程序不能直接访问大于 64KB 的程序存储器，为了访问 64KB 以上的存储器(如在 Mega128 中)，应选中“Use RAMPZ/ELPM”(Project->Option->Target)前的复选框，并且在设定 RAMPZ 寄存器后直接调用 ELPM 指令来访问 Flash 存储器。

编译器生成代码分配到不同的区域“areas”，区域按照程序存储器地址增高的顺序被使用，共分为以下区域：

interrupt vectors：这个区域包含中断向量

func_lit：函数表区。这个区的每个字保存了函数入口的地址，为了与代码压缩完全兼容，所有间接的函数索引必须进行额外对准。如果在 C 中通过函数指针调用函数，编译器可自动完成对准的操作。在汇编中，这个对准必须在用户程序中进行，举例如下：

```
.area func_lit                    ;假设_foo 是函数的名称
PL_foo::.word _foo                ;创建函数表入口
.area text;
ldi R30,<PL_foo;
```

```
ldi R31,>PL_foo;
rcall xicall;
```

编程人员应把函数表入口地址先送入 R30、R31 寄存器后，才可以使用库函数 xicall 间接调用这个函数。

lit：这个区域包括了整型数和浮点数常量。

idata：全局变量和字符串的初始值保存在这个区域，在启动时，由程序复制到数据存储器的 data 区。

text：这个区域包括程序代码。

2．内部数据存储器(SRAM)

这个数据存储器是用于保存变量、堆栈和动态内存分配的堆，通常它们不出现在输出文件中，但在程序运行时被使用。

一个没有使用外部扩展数据存储器的程序使用数据内存如图 3.7 所示，在该图中，从地址 0 开始的 96 个字节(0x60)是 CPU 寄存器和 I/O 寄存器，编译器从 96 往上放置全局变量和字符串，在变量区域的顶部是用户可以利用的分配动态内存，在高端地址，硬件堆栈开始于 SRAM 的最高字节，在它的下面是软件堆栈，硬件堆栈和软件堆栈均为向下生长型。要求程序员在设计程序时要确保硬件堆栈不生长进软件堆栈，而软件堆栈不生长进已分配数据的动态分配区中，否则将会导致意外的结果。

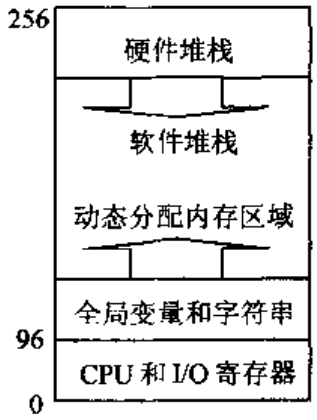


图 3.7　数据内存的使用

编译器生成数据分配到不同的区域“areas”，区域按照数据存储器的地址增高的顺序使用，共分为以下区域：

data：是包含全局变量、静态变量和字符串的数据区域。全局变量和字符串的初始值保存在程序存储器的“idata”区域内，在启动时被复制进 data 数据区的。

bss：这个区域包含未初始化的 C 全局变量，按 ANSI C 标准这些变量在启动时将初始化为 0。

3．外部数据存储器(SRAM)

如果选择带有外部 SRAM(32KB 或 64KB)的目标器件，那么堆栈是放置在内部 SRAM

的顶部并且朝低端内存地址生长，数据内存(即图 3.7 中的动态分配内存区域)是开始于硬件堆栈的顶部(即外部 SRAM 的底部)并且向上生长，这样分配的原因是在多数场合访问内部 SRAM 比访问外部 SRAM 的速度要快，分配堆栈到较快的内存有很多好处。

4．EEPROM 存储器

EEPROM：这个区域包含 EEPROM 数据。ICCAVR 编译器将需要置于 EEPROM 数据写进扩展名为.eep 的输出文件中，在对 AVR 芯片编程时将该文件写入芯片的 EEPROM 存储器中。

3.2.4 启动文件

在一些特殊的应用中，如用户需判断单片机是上电复位还是由其他原因引起复位(如看门狗等)，并且针对不同的复位情况采取不同的对策，这时用户可能需要使用自己的启动文件，在 IAR 中需要修改相应的 XCL 文件才能实现改变启动文件的目的，ICCAVR 和 CodeVisionAVR 在工程属性窗口中可以直接指定使用外部的启动文件。

根据目标 MCU 的种类和功能，ICCAVR 编译器将从下面默认的启动文件中选择一个。

crtavr.o：非 ATmega 类芯片默认的启动文件。

crtatmega.o：ATmega 类芯片默认的启动文件。

ICCAVR 也提供了下面四种常用的启动文件，用户可以在工程选项对话框中(Project->Options->Target->Non Default Startup)指定一个启动文件(或用户自定义的启动文件)，应注意必须指定启动文件的绝对路径，如果没有指出启动文件的路径，则默认启动文件位于工程选项库路径所指定的目录中(默认为 c:\icc\lib)。

crtavrram.o：与 crtavr.o 相比，增加了初始化外部 SRAM。

crtatmegaram.o：与 crtatmega.o 相比，增加了初始化外部 SRAM。

crtboot.o：与 crtavr.o 相比，增加了 bootloader 的内容，只有 ATmega 类的芯片才能选用。

crtboothi.o：与 crtboot.o 相比，增加了使用 ELPM/RAMPZ 内容，一般在需要读取大于 64KB 字节存储器中的常数表格或字符串时才用。

上面的非 ATmega 类芯片的每一个中断入口地址使用一个字(2 个字节)，而 ATmega 类的每个中断入口地址使用 2 个字(4 个字节)，因此两种芯片的启动文件不能混用。

启动文件的功能有：

1．初始化硬件和软件堆栈指针。

2．从 idata 区复制初始化数据到直接寻址数据区 data 区。

3．将 bss 区全部初始化为零。

4．启动文件定义了一个全局符号“_start”，它是程序的起点。

5．调用用户主程序。

6．定义一个退出点，即定义为一个无限循环。如果主函数 main( )一旦退出，它将进入这个退出点进行无限循环。

下面简单介绍如何修改和创建一个新的启动文件：

使用 ICCAVR 的 IDE 或 UlterEdit32 打开需要修改的启动文件(crtavr.s、crtatmega.s 或其他文件)，对启动文件进行相应的修改并保存，如果用 UlterEdit32 软件编辑，应退出 UlterEdit32 软件并用 IDE 调用修改后的启动文件，在 IDE 中选择“File”菜单的“Compile File To->Startup File To Object”，生成相应的目标文件(crtavr.o、crtatmega.o 或其他的文件名)。要注意，在默认设置的情况下，应在 libsrc.avr 文件夹中生成目标文件，再将修改后的目标文件复制到 c:\icc\lib 文件夹中，并且应在工程选项对话框中指定修改后的启动文件。

3.3　ICCAVR 菜单解释

本章的菜单是按 ICCAVR6.26C 版本来介绍的，其他版本的菜单可能会与本章介绍的略有不同。在介绍菜单前，读者应先了解以下关于文件的几种说法：

活动的文件：是指打开的且当前正在编辑的文件；

打开的文件：是指已调入 IDE 环境中的文件。可以是活动的文件(正在编辑)，也可以是放在后台的文件；

没有打开的文件：是指该文件保存在硬盘上，没有调入 IDE 环境。

1．鼠标右键弹出菜单，如图 3.8 和 3.9 所示。

在 ICCAVR 环境中单击鼠标右键，那么 ICCAVR 会根据实际情况弹出相应的工具菜单，图 3.8 为在编辑窗口中(图 3.25 中的①)单击鼠标右键弹出的工具菜单，图 3.9 为在工程窗口中(图 3.25 中的②)单击鼠标右键弹出的工具菜单。

图 3.8　编辑窗口右键菜单

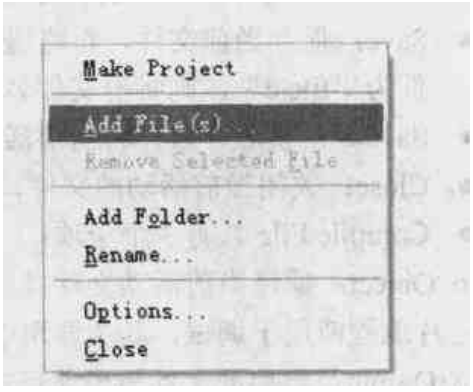


图 3.9　工程窗口右键菜单

2．File Menu 文件菜单，如图 3.10 所示。

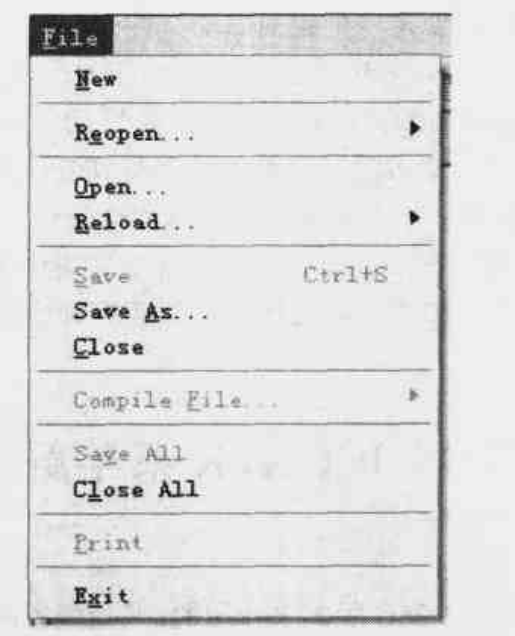


图 3.10 文件菜单

- New：新建一个文件，可在编辑窗口中输入文本或代码。
- Reopen：重新打开历史文件，有关历史文件显示在右边的子菜单中。
- Open：打开一个已经保存在磁盘上的文件用于编辑，用浏览窗口选择需要打开的文件。
- Reload 中有两个子项。

from File：放弃全部未保存的修改，从磁盘中重装当前文件，假设正在编辑的源文件为“file.c”，则重装的是“file.c”文件。

from Backup：放弃全部未保存的修改，从磁盘中重装当前文件的备份文件，假设正在编辑的源文件为“file.c”，则重装的是“file._c”文件。

- Save：保存当前文件，再将磁盘上保存前的文件以<file>._<ext>形式备份。如源文件为“file.c”，则备份文件名为“file._c”。
- Save as：换名存盘，将当前活动文件用另外一个名称来保存。
- Close：关闭当前活动的文件，如果文件修改而没有保存过，则系统会提示保存。
- Compile File 共有 3 个子项：

to Object：编译当前活动文件并生成目标文件。注意，生成的目标文件不可以直接用于对芯片编程或用于调试，其主要用于语法检查、创建新的库文件。

to Output：将启动文件与当前活动文件一起合并编译，生成输出文件，产生的输出文件可用于编程器和调试器。

Startup file to Object：创建新的启动文件。

- Save All：保存所有打开的文件。
- Close All：关闭所有打开的文件，系统会提示保存已经修改而没有保存的文件。
- Print：用 Windows 默认的打印机打印当前文件。

- Exit：关闭所有打开的文件并退出 ICCAVR 的 IDE 环境，系统会提示保存已经修改而没有保存的文件。

3. Edit Menu 编辑菜单，如图 3.11 所示。

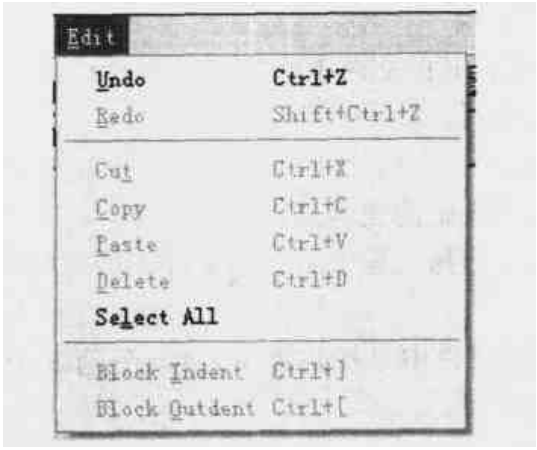


图 3.11　编辑菜单

- Undo：撤销最后一次的修改。
- Redo：撤销最后一次的 Undo。
- Cut：剪切选择的内容到剪贴板。
- Copy：复制选择的内容到剪贴板。
- Paste：将剪贴板内容粘贴在当前光标的位置。
- Delete：删除选择的内容。
- Select All：选择当前活动文件的全部内容。
- Block Indent：对选择的整块内容右移一个制表位。
- Block Outdent：对选择的整块内容左移一个制表位。

4. Search menu 查找菜单，如图 3.12 所示。

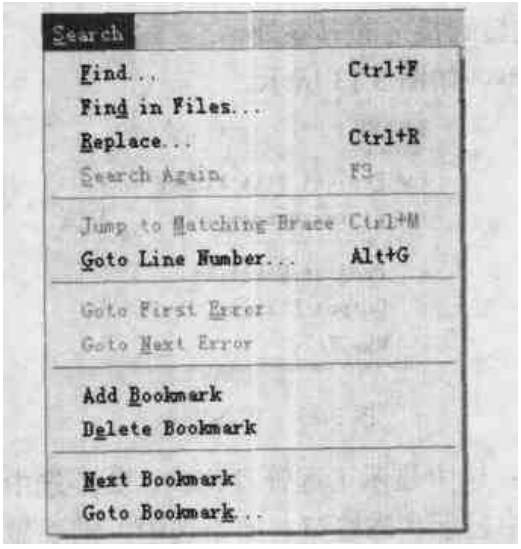


图 3.12　查找菜单

- Find：在当前活动文件中查找一段文本。它有 3 个选项，可设定为中文显示。

Match Case：区分大小写。

Whole Word：全字匹配。

Up/Down：往上或往下寻找。

- Find in Files：在当前打开的文件中查找一段文本。

Case Sensitive：区分大小写。

Whole Word：全字匹配。

Regular Expression：查找规则的表达式。

Where：查找位置。可以选择工程中的全部文件、所有打开的文件、当前目录中全部文件。

- Replace：在当前活动文件中寻找并替换文本。它有 3 个选项，可设定为中文显示。

Case Sensitive：区分大小写。

Whole Word：全字匹配。

Find Again：寻找下一个。

- Jump to Matching brace：跳转到与光标后面符号相匹配的位置。ICCAVR 能够自动寻找以下符号之间的匹配符号，"("和")"、"["和"]"、"{"和"}"、"<"和">"，在使用该选项前，应先将光标移动到需要寻找匹配符号的前面位置，再选中本菜单，则光标就自动跳到匹配符号的前面。
- Goto Line Number：光标跳转到指定行号。
- Goto First Error：光标跳转到当前活动文件中的第一个错误之处。
- Goto Next Error：光标跳转到当前活动文件中的下一个错误之处。
- Add Bookmark：添加书签。
- Delete Bookmark：删除书签。
- Next Bookmark：跳转到下一个书签处。
- Goto Bookmark：跳转到指定的书签处。

5. View Menu 视图菜单，如图 3.13 所示。

图 3.13 视图菜单

- Project File Window：选中显示工程管理窗口，如不选中，则不显示工程管理窗口。
- Status Window：选中显示状态窗口，如不选中，则不显示状态窗口。
- Project makefile：生成 Makefile 文件，Makefile 文件包含整个工程中的全部编译

信息。

- Output Listing：输出列表文件，列表文件包含了源文件中的全部语句对应的汇编代码，供调试程序和了解汇编生成的代码使用。
- Map File：输出内存映像文件，包含了程序中有关符号及其所占内存大小的信息。

6. Project Menu 工程菜单，如图 3.14 所示。

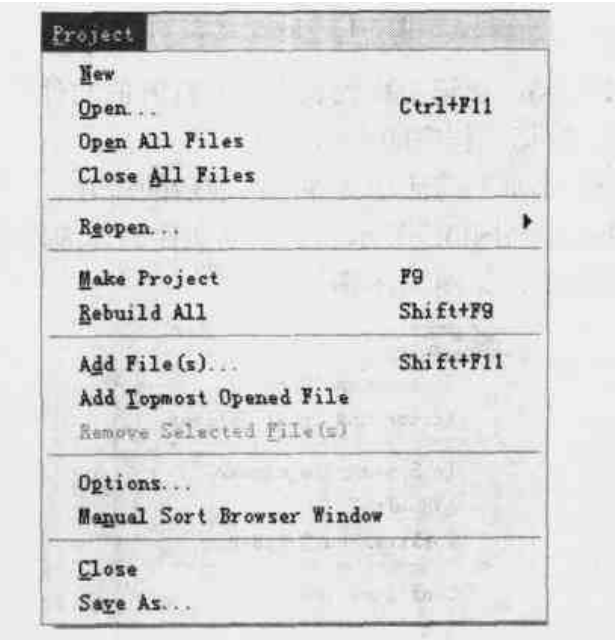


图 3.14　工程菜单

- New：创建一个新的工程文件。
- Open：打开一个已经存在的工程文件。
- Open All Files：打开工程的全部源文件。
- Close All Files：关闭工程全部打开的文件。
- Reopen：重新打开历史文件，有关的历史文件显示在子菜单中。
- Make Project：解释和编译已经修改的文件，并且生成与工程同名的输出文件。
- Rebuild All：重新构筑全部文件，并且生成与工程同名的输出文件。
- Add File(s)：添加一个文件到工程中，这个文件不一定打开，也可以不是源文件。
- Add Topmost Opened File：将当前的活动文件添加到工程中。
- Remove Selected Files：从工程中删除选择的文件。
- Option：打开编译设置对话框，详见编译设置说明。
- Manual Sort Browser Window：工程管理窗口的显示方式。
- Close：关闭打开的工程。
- Save As：将工程换名存盘。

7. RCS 菜单，如图 3.15 所示。

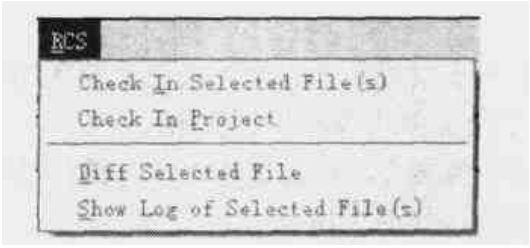


图 3.15 RCS 菜单

- Check In Select File(s)：登记工程列表中所有打开的文件。
- Check In Project：登记工程中的全部文件。
- Diff Selected File：显示当前活动文件修改前后的差异。
- Show Log of Selected File(s)：显示当前活动文件的详细修改过程(记录)。

8. Tools Menu 工具菜单，如图 3.16 所示。

图 3.16 Tools 菜单

- Environment Options：打开系统设置对话框，详见系统设置说明。
- Editor and Print Options：打开编辑和打印设置对话框。
- In System Programmer：打开 IDE 内置的下载功能设置对话框。
- AVR Calc：打开 AVR 定时器/计数器设置计算器。
- Application Builder：打开应用构筑向导程序，用于生成硬件的初始化代码。
- Configure Tools：允许添加自己的内容到工具菜单。
- Run：以命令行方式运行一个程序。

9. Terminal 终端仿真程序，如图 3.17 所示。

图 3.17 终端仿真菜单

- Show Terminal Window：打开 IDE 内置的终端仿真窗口。
- Clear Terminal：关闭 IDE 内置的终端仿真窗口，回到编辑状态。

- Capture：捕获。

10．Help 帮助菜单，如图 3.18 所示。

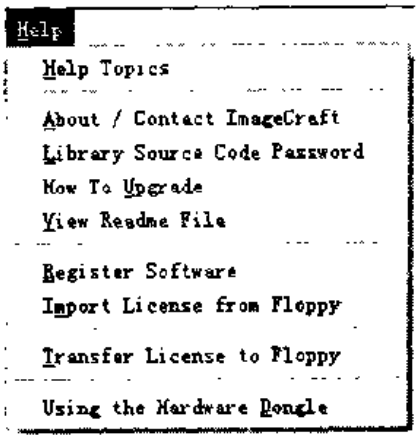


图 3.18　帮助菜单

- Help Topics：帮助主题，从这里可以查阅到 ICCAVR 的全部帮助。
- About/Contact ImageCraft：与 ImageCraft 联系。
- Library Source Code Password：库源代码口令。如果是正式版用户，可以得到库源代码(libsrc.zip)的口令。
- How To Upgrade：升级方式。
- View Readme File：查看自述文件，记录下 ICCAVR 的升级过程。
- Register Software：输入软件使用许可。
- Import License from Floppy：从软盘输入注册文件。
- Transfer License to Floppy：将注册文件导出到软盘。
- Using the Hardware Dongle：使用软件狗。

11．快捷菜单，如图 3.19 所示。

图 3.19　快捷工具栏菜单

工具栏菜单从左至右分别为：

- 新建一个文件，相当于执行“File->New”命令。
- 打开一个文件，相当于执行“File->Open”命令。
- 保存当前活动文件，相当于执行“File->Save”命令。
- 打印当前活动文件，相当于执行“File->Print”命令。
- 剪切选择的内容到剪贴板，相当于执行“Edit->Cut”命令。
- 复制选择的内容到剪贴板，相当于执行“Edit->Copy”命令。
- 将剪贴板内容粘贴在当前光标的位置，相当于执行“Edit->Paste”命令。
- 在当前活动文件中查找一段文本，相当于执行“Search->Find”命令。

- 解释和编译工程文件，相当于执行“Project->Make Project”命令。
- 停止最后的动作，例如停止当前正在进行的编译工作等。
- 打开编译设置对话框，相当于执行“Project->Option”命令。
- 光标跳转到第一个有错误的行，相当于执行“Search->Goto First Error”命令。
- 光标跳转到下一个有错误的行，相当于执行“Search->Goto Next Error”命令。
- 打开应用构筑向导程序，相当于执行“Tools->Application Builder”命令。
- 打开下载设置对话框，相当于执行“Tools->In System Programmer”命令。

12．Compiler Options 编译设置。

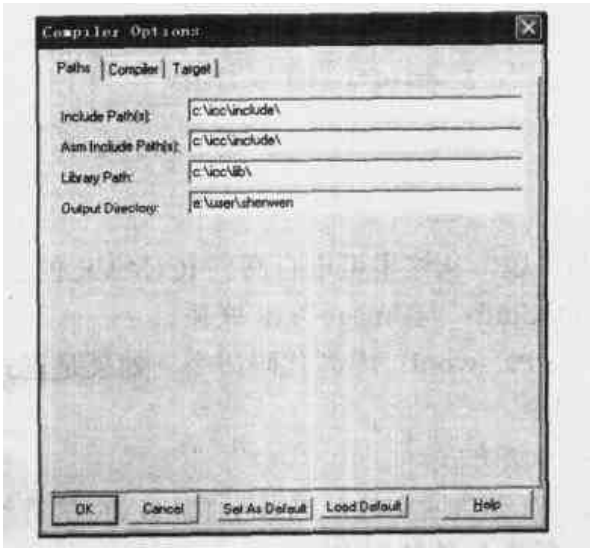


图 3.20 Paths 页面

编译设置(Compiler Options)的最下方有 5 个按钮，分别介绍如下：

- OK：同意所做的更改。
- Cancel：撤销所做的更改。
- Set As Default：将当前各页面的设置保存为默认值。
- Load Default：读出文件中的各页面的默认设置值，并覆盖当前的设置。
- Help：帮助。

编译设置总共有三个页面(Paths、Compiler 和 Target)，下面分别介绍：

路径(Paths)页面如图 3.20 所示，共有以下几项设置：

- Include Path(s)：可以指定包含文件的路径，ICCAVR 6.26C 可以根据用户选定的安装路径自动设置本选项的内容，但在 6.25 以前的版本，需手工设置。
- Asm Include Path(s)：指定包含汇编文件的路径。
- Library Path：链接器所使用库文件的路径，ICCAVR6.26C 可以根据用户选定的安装路径自动设置本选项的内容，但在 6.25 以前的版本，需手工设置。
- Output Directory：输出文件的目录，如不指定，则输出至源文件所在的目录。

编译器(Compiler)页面如图 3.21 所示，共有以下几项设置：

- Strict ANSI C Checking：严格的 ANSI C 语法检查。

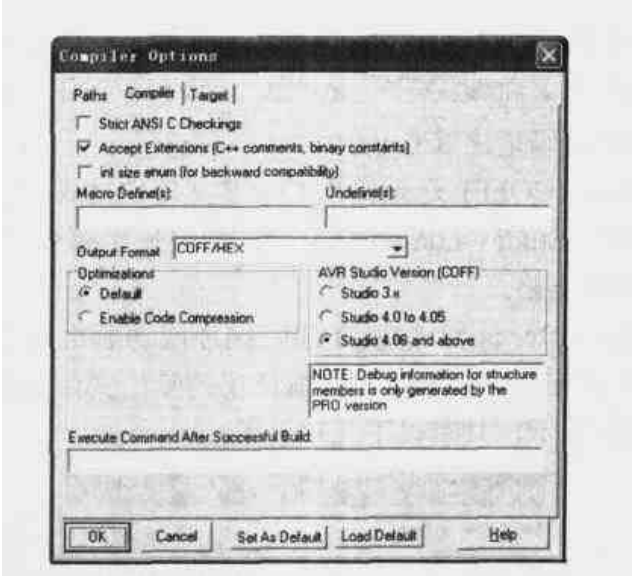


图 3.21　Compiler 页面

- Accept Extensions(C++ comments,binary constants)：接受 C++类型语法扩充。
- int size enum(for backward compatibility)：接受整数扩充(为了与其他 C 编译器兼容)。
- Macro Define(s)：定义宏，宏之间用空格或分号分开，宏定义形式如下：

```
name[:value] 或 name[=value]
```

例如：

```
DEBUG:1;PRINT=printf
```

等价于在 C 源程序中的以下指令：

```
#define DEBUG 1
#define PRINT printf
```

- Macro Undefine(s)：取消宏定义，用法同上。
- Output File Format：选择输出文件格式，有以下 COFF/HEX、COFF、Intel HEX 三种格式可以选择，一般选取 COFF/HEX 选项，即同时生成“.cof”和“.hex”文件。
- Optimizations：代码优化。有以下两个选项：

Default：基本优化。只优化寄存器分配、共用相同的子例程等。

Maximize Code Size Reduction：代码压缩优化，演示版无此功能，标准版有使用 30 次的限制，专业版没有使用次数限制。

注意：在使用代码压缩优化时，即使编译出错，也算使用一次，因此应该先用默认的基本优化进行编译，等编译通过后再用压缩优化重新编译一次。如果代码压缩优化的 30 次使用限制已用完，对于正版用户，可以将 ICCAVR 重新安装到一个新的目录中，又可以再使用 30 次代码压缩优化(安装前别忘了导出注册文件)，还可以重新安装 Windows 和 ICCAVR，效果相同。还有，在 ICCAVR 中使用代码压缩优化可能会引起一些奇怪的问题，因此，除

非万不得已，尽量少用代码压缩优化功能。不过这个问题也并非 ICCAVR 独有，就算是 Keil 在选用 9 级优化时也会有很多问题，只有 GCCAVR 在代码压缩方面比较稳定，基本不会有以上的问题，但是生成的代码量比 ICCAVR 大。

- AVR Studio Version(COFF)：选择输出 COFF 文件的格式，共有以下选项：Studio 3.x、Studio 4.0 to 4.05、Studio 4.06 and above，如果使用哪个版本的 Studio 进行仿真，就应选用相应的输出格式。
- Execute Command After Successful Build：编译成功后执行的命令，可以让 ICCAVR 在编译完成后执行相应的指令，如将编译出的输出文件下载至芯片等操作。

Target 页面如图 3.22 所示，共有以下几项设置：

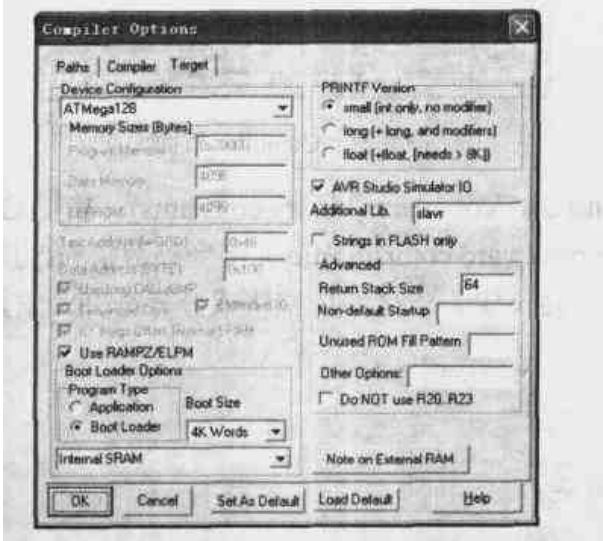


图 3.22 Target 页面

- Device Configuration：选择目标 MCU 类型，当选择 Custom 选项时，必须指定以下选项中的内容，如果选择了相应的器件(如：AT90S8515、ATmega8 等)，这些选项由 ICCAVR 自动设置。
- Memory Sizes(bytes)：选择“Custom”时必须指定，共有三个选项，分别设置 ROM、SRAM 和 EEPROM 的大小。
- Text Address(Word)：代码开始的位置，选择“Custom”时必须指定，通常开始于中断向量区域的后面。
- Data Address(byte)：数据存储器起始地址，选择“Custom”时必须指定，通常以 0x60 开始。
- Enhanced Core：硬件是否支持增强核指令，选择“Custom”时必须指定。
- Extended IO：用于指定 MCU 是否有扩展的 I/O 端口，选择"Custom"时必须指定。
- IO Registers Offset Internal SRAM：指定内部 SRAM 的偏移量，选择“Custom”时必须指定。例如 8515 的 SRAM 起始于 0x60，在 I/O 寄存器空间后面共延伸 512 个字节；而 Mega 的 I/O 寄存器覆盖在 SRAM 空间中，因此 Mega 系列 SRAM 是从 0 开始的。

- Use Long JMP/CALL：所用芯片是否支持长跳转和长调用指令，选择“Custom”时必须指定。
- Boot Loader Options：共有 2 个内容选项，只有指定为 ATmega 系列单片机才会出现本选项。
- Internal SRAM 或 External SRAM：指定目标系统的数据 SRAM 类型。
- PRINTF Version：选择 PRINTF 的版本，有以下选项。
  Small：选择本项时支持%c、%d、%x、%X、%u 和%s 的格式。
  Long：选择本项时除支持 Small 类型外，还支持%ld、%lu、%lx 和%lX 的格式。
  Floating point：选本项时除支持 Long 类型外，还支持%f 格式(注意这个选项需要很大的内存)。
- AVR Studio Simulator IO：如果选中，输出文件可以在 AVR Studio 的终端模拟仿真。
- Additional Libraries：使用标准库以外的附加库。
- Strings in FLASH：字符串只保存在 FLASH 存储器中。
- Advance：高级选项，有以下内容：
  Return Stack Size：指定编译器使用的硬件堆栈的大小。如果子程序调用嵌套不深(不超过 4 层)，那么使用默认的 16 个字节就足够了，如果使用了浮点函数，则至少应设定为 30 个字节。在一般情况下，除了层次很深的递归调用及使用了%f 格式说明符外，设定为 40 个字节就足够了。
  Non Default Startup：允许指定一个启动文件而不使用默认的启动文件(系统默认的启动文件在“Paths”页面中设定)，这样 IDE 可以使用多个启动文件。
  Unused ROM Fill Pattern：用一串十六进制数填充空余的 ROM 空间。
  Other Options：自定义代码区在 Flash 中的起始地址，详见“2.11.4 编译附注”中改变代码段名称。
  Do Not use R20..R23：不使用 R20、R21、R22、R23 寄存器，使用本选项有利于降低堆栈的大小，节约数据存储器。

13．Environment Options 系统设置，如图 3.23 所示。

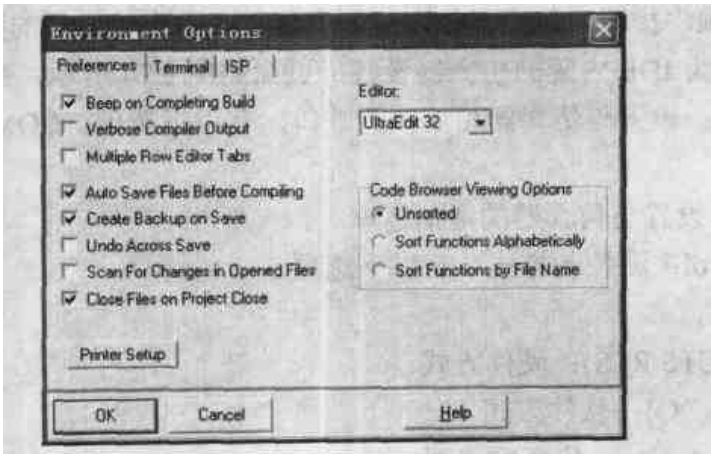


图 3.23　Preferences 页面

系统设置选项中总共有三个页面 Preferences、Terminal 和 ISP，其中 Preferences 页面为参数设置，页面如图 3.23 所示，共有以下几项设置。

- Beep on Completing Build：编译完成后发出提示音。
- Verbose Compiler Output：在状态窗口中显示详细的编译过程。
- Mulitiple Row Editor Tabs：多重跳格(制表位置)设定。
- Auto Save Files Before Compiling：在编译前自动保存文件。
- Create Backup on Save：在保存文件时，将原文件保存为后备文件(扩展名前加一个下划线)。
- Undo Across Save：撤销改变，即使曾经保存过也撤销。
- Scan for changes in Opened File：检查其他编辑器打开的文件。在选用其他软件编辑源文件时，使用这个功能可以自动检查源文件是否被修改，如果被修改就提示存盘。
- Close Files on Project Close：当关闭工程文件时，自动关闭全部与此工程相关的已打开的文件。
- Printer Setup：设置打印机。
- Editor：编辑器选择。有以下选项：IDE Builtin、UltraEdit 32、TextPad、WinEdit。

当选用其他编辑器时，用 ICCAVR 的 IDE 打开一个文件时，IDE 会自动使用指定的编辑器打开该文件(不是用 ICCAVR 的 IDE 打开)。由于 ICCAVR 6.26C 对中文的支持不够完善，在删除汉字时存在半个汉字现象，并且粘贴带有中文注释的源文件时偶尔会出错。因此编者通常都在选择 UltraEdit 32 选项时使用 UltraEdit 软件编辑源文件。

- Code Browser Viewing Options：工程管理窗口中的代码浏览器显示方式选择，有以下三个选项：

  Unsorted：不排序。如果有几个源文件，各源文件分开显示，每个源文件按变量及函数出现的顺序排列。

  Sort Functions Alphabetically：按字母顺序排列。如果有几个源文件，将各源文件的变量及函数合并后再按字母的顺序排列。

  Sort Functions by File Name：按源文件字母顺序排列。如果有几个源文件，各源文件按字母顺序排列，各源文件内部的变量及函数也按字母的顺序排列。

Terminal 页面为 IDE 内置的仿真器设置，页面如图 3.24 所示，共有以下几项设置。

- COM Port：设置与仿真器相连接的端口，有以下选项：COM1、COM2、COM3、COM4。
- Baudrate：设置与仿真器的通信速率。
- Flow Control：流控制方式，有以下选项。

  None：没有。

  Hardware(CTS/RTS)：硬件方式。

  Software(^s/^Q)：软件方式。
- Keep DTR Active：仿真器控制。
- ASCII Transfer Protocol：ASCII 码传送协议。

其中 ISP 页面为系统设置，该下载器默认为 STK500，如果使用其他下载器，请根据下载器作相应的设置。

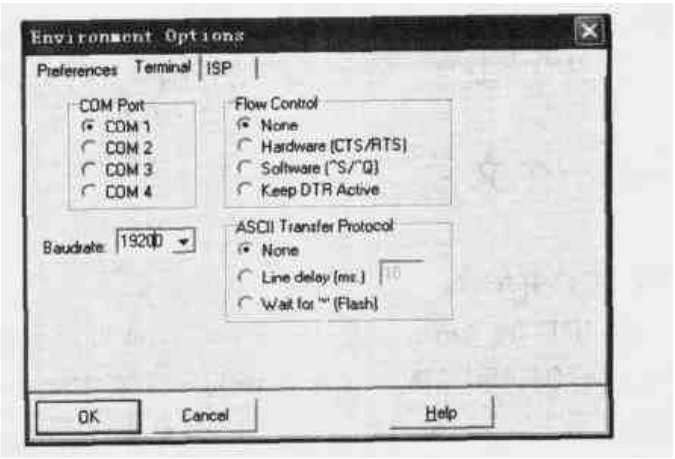


图 3.24　Terminal 页面

3.4　ICCAVR 编译器的 IDE 环境

3.4.1　工程管理

启动 ICCAVR 后，就进入 IDE 环境。在 IDE 环境中，由编辑窗口、工程管理窗口、状态窗口和文件切换组成，如图 3.25 所示。

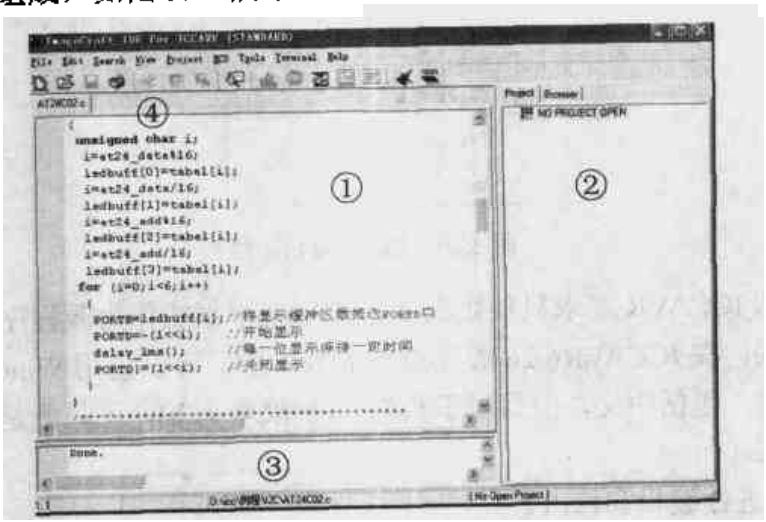


图 3.25　ICCAVR 的 IDE 窗口

编辑窗口，如图 3.25 中①的区域，这是用户与 IDE 交流信息的主要区域，在这个窗口中可以输入并修改相应的文件，一般源程序的输入及修改都在这个窗口内完成。

工程管理窗口，如图 3.25 中②的区域，由工程栏“Project”和代码浏览器“Browser”

栏组成。工程栏用于显示与工程相关的全部文件(文件均需由用户指定)，代码浏览器中显示该工程生成代码中的有关信息。

状态窗口，在图 3.25 中③的区域，显示 IDE 的工作状态及提示信息。

文件切换，在图 3.25 中④的区域，用于快速选择已打开的文件。

3.4.2 创建并编译一个文件

1．创建一个新的源文件并编译

(1) 创建新文件。从 IDE 的“File”菜单中选择“New”，创建一个新的文件，可以在该文件中输入源程序并进行编辑和修改，也可以作为一个文本文件，用来保存与工程有关的信息或作为其他用途。

(2) 存盘。新创建的文件必须存盘，在存盘时必须指定文件类型，如果是 C 语言源程序，必须用“.c”扩展名，如果是汇编语言源程序，必须用“.s”扩展名，其他文件(如工程说明等)一般使用“.txt”扩展名。如果没有存盘或者扩展名不是“.c”或“.s”，则“File->Compile File”一栏是灰色的，表示该功能目前不能使用，如图 3.26 所示。

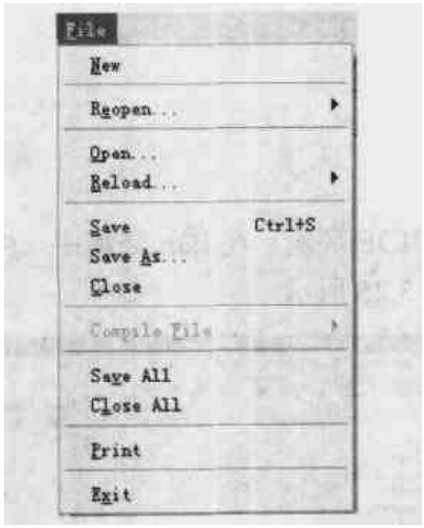


图 3.26 Compile File 栏

说明：旧版的 ICCAVR 要求只能用英文、数字或下划线作为源程序的文件名，否则可能会导致编译错误，在 ICCAVR6.26 版及更新的版本中，可以使用 Windows 允许的字符作为源程序的文件名，包括中文，但是对于头文件(扩展名为.h)，还只能是使用英文、数字或下划线作为文件名。

(3) 对源文件进行编辑和修改。

(4) 编译源文件，如果需要生成目标文件(用于语法检查、创建新的库文件)，应选用“File->Compile File->to Object”编译选项；如果需要生成的新的启动文件，应选用“File->Compile File->Startup File To Object”编译选项；如果需要生成的目标文件用于 AVR studio 调试或用于编程器下载，则应选用“File->Compile File->to Output”编译选项。

在编译源文件时，如果有语法错误，则将有关的错误显示在状态窗口中，用鼠标单击状态窗口中的错误信息时，编辑器会自动将光标定位于与错误相关的首行，修改所有的错误，直到生成所需的目标文件或输出文件。注意：对于 C 源文件中缺少“;”的错误，编译器定位于下面一行。

(5) 在 AVR studio 中进行软件仿真或硬件仿真调试，如果使用硬件仿真，应将扩展名为“.hex”和“.eep”的文件分别下载到 AVR 芯片的 Flash 存储器和 EEPROM 存储器中。在调试过程中发现错误，应先打开源文件修改错误，重新编译并调入 AVR studio 继续调试。

2．打开磁盘上的源文件并编译

(1) 打开文件。从 IDE 的“File”菜单中选择“Open”，通过 IDE 提供的“打开文件对话框”选择磁盘上所需打开的文件，或者通过“File”菜单中的“Reopen”命令的子菜单打开最近使用过的文件。

(2) 对源文件进行编辑和修改。

(3) 编译源文件，如果需要生成目标文件(用于语法检查、创建新的库文件)，应选用“File->Compile File->to Object”编译选项。如果需要生成新的启动文件，应选用“File->Compile File->Startup File To Object”编译选项。如果需要生成的目标文件用于 AVR studio 调试或用于编程器下载，则应选用“File->Compile File->to Output”编译选项。

(4) 在 AVR studio 中仿真调试。

3.4.3　创建并编译一个工程文件

ICCAVR 的 IDE 允许用户将多个文件组织到同一工程中，而且可以分别定义每个文件的编译选项，这个特性允许用户将工程分解成许多小的模块，单独对小的模块进行编译，最后再链接成一个完整的文件。为避免工程所在的文件夹(目录)混乱，用户也可以指定输出文件和中间文件到一个指定的文件夹中，通常这个文件夹是工程目录的一个子文件夹。

1．创建一个新的工程并编译

(1) 创建新工程。从 IDE 的菜单“Project”中选择“New”命令，IDE 会弹出一个对话框，在对话框中用户可以指定工程存放的文件夹以及工程的名称。要注意：工程编译以后输出的文件名与工程名相同，而不是与工程中源文件名相同。

在建立一个新工程后，在工程管理器的窗口会现三个子目录，Files、Headers、Documents，如图 3.27 所示。要注意，ICCAVR 并不是真的在工程存放的文件夹中再生成这三个文件夹，而是将工程文件夹中的源文件分别登记到这三个文件夹中，以方便管理。

(2) 由于在工程中并不能编辑源文件，因此必须新建或打开磁盘上已有的源文件，按照前面介绍的步骤，新建或打开一个源文件，一般情况下，应将新建或打开的源文件保存在与工程文件同一个文件夹中，或者工程文件所在文件夹的下一级文件夹中，如图 3.28 所示。

(3) 将源文件添加到工程中。要将当前活动的文件加入工程管理器的文件中，可以用“Project->Add Topmost Opened File”命令，也可以在编辑窗口中单击鼠标右键，选择弹出

菜单的“Add To Project”命令。如果要将磁盘上的文件加入工程管理器中，可以用“Project->Add File(s)”命令，也可以在工程管理器窗口单击鼠标右键，选择弹出菜单中的“Add To Project”命令，通过“选择文件对话框”将磁盘上的文件加入到文件夹中。

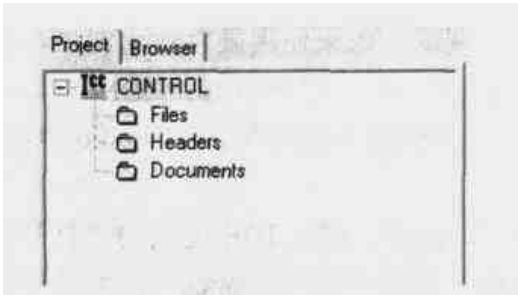


图 3.27 工程管理器

图 3.28 工程管理文档示意

说明：如果用户添加到工程中的是 C 语言源文件(扩展名为.c)或汇编语言源文件(扩展名为.s)，则 IDE 自动将该文件保存到“File”文件夹中，如果用户添加到工程中的是头文件(扩展名为.h)，IDE 自动将该文件保存到“Headers”文件夹中，如果用户保存的是以上三种扩展名以外的文件，则 IDE 自动将其保存到“Documents”文件夹。

单击工程管理器文件夹前的“+”号，就可以展开该文件夹中的内容，单击文件夹前的“-”号，就可以收起文件夹中的内容。如果需要打开某个文件夹中的文件，只要双击文件夹中的文件名，IDE 就会自动打开该文件。

如果需要从工程管理器中删去一个文件，先单击需要删除的文件，然后选择“Project->Remove Selected File(s)”命令就可以从工程管理器中删去该文件，也可以单击需要删除的文件，再单击鼠标右键，从弹出菜单中选中“Remove Selected File(s)”命令，也可以从工程管理器中删去该文件。要注意，从工程管理器中删除一个文件，只是工程管理器不再管理这个文件，并不是把该文件从磁盘上真正删除，该文件还是保存在磁盘上。

(4) 对源文件进行编辑和修改。

(5) 编译源文件。可以单独编译其中的一个源文件，或对其进行语法检查，使用“File->Compile File->to Object”或“File->Compile File->to Output”命令可以只编译当前的活动文件，也可以直接对工程中的全部的源文件进行编译，使用“Project->Make Project”或“Project->Rebuild All”命令。

要注意这几种编译命令的区别，“File->Compile File->to Object”或“File->Compile File->to Output”命令均只是针对当前活动的文件，编译生成的目标文件与源文件的名称相同(扩展名不同)。“Project->Make Project”或“Project->Rebuild All”命令是编译工程中全部的源文件，不管工程中的文件是否打开，也不管当前已打开的活动文件是什么，ICCAVR 只管编译工程管理器中管理的源文件(Files 文件夹中的文件)，编译生成的目标文件与工程的名称相同(扩展名不同)。至于“File->Compile File->Startup File To Object”编译选项一般只用于生成新的启动文件时使用。

在编译过程中，如果源文件存在语法错误(error)，应修改错误，直到通过编译，并且还

应该确认状态窗口的警告信息(warn)不会影响生成正确的代码。

(6) 在 AVR studio 中仿真调试。注意，在 AVR studio 软件中应打开与源文件同名的目标文件进行调试，而不能打开与工程文件同名的目标文件进行调试。如果在调试过程中发现错误，应先打开源文件修改错误，重新编译并调入 AVR studio 继续调试。

3.5　用应用构筑向导生成一个工程文件

可以点击快捷工具栏的“Build Project”图标或在菜单“Tools”栏中的“Application Builder”命令，就可以打开应用构筑向导对话框，在对话框中有以下页面 CPU、Memory、Ports、Timer0、Timer1、UART、SPI、Analog，下面分别介绍。

CPU 选项：如图 3.29 所示，可以设定芯片的种类、频率、是否使用看门狗及设定看门狗的预定比例、以及设定是否使用 INT0 和 INT1，设定 INT0 和 INT1 的触发方式。

Memory 选项：如图 3.30 所示，可以设定是否使用扩展 SRAM，是否插入等待周期等内存信息。

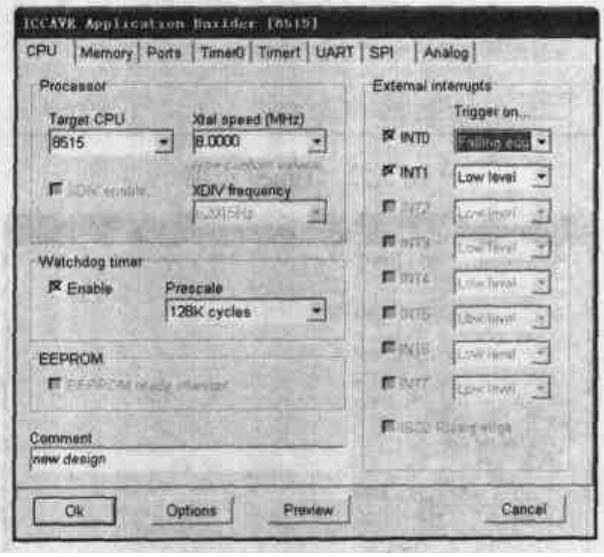


图 3.29　CPU 选项

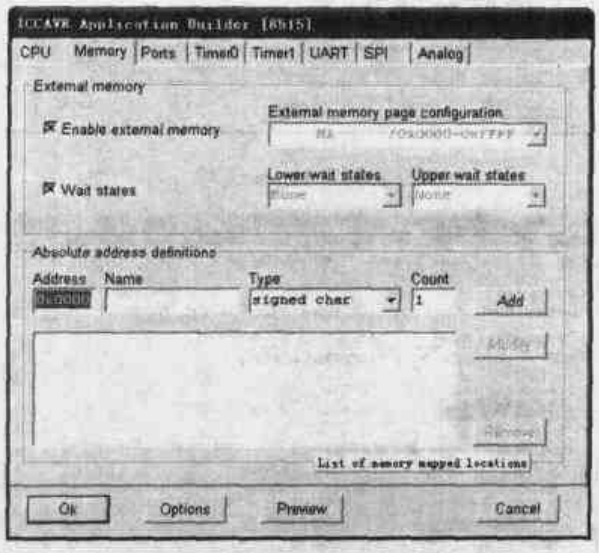


图 3.30　Memory 选项

Ports 选项：如图 3.31 所示，可以设定 I/O 口的特性。表 3.1 为 Ports 选项中符号与端口实际设定的关系。

表 3.1　Ports 选项中符号与端口实际设定

	Direction：“I” Value：“↑”	Direction：“I” Value：“ ”(空)	Direction：“0” Value：“1”	Direction：“0” Value：“0”
PORTA	0xff	0x00	0xff	0x00
DDRA	0x00	0x00	0xff	0xff

说明：在上表中，以 PA 端口为例，并假定 PA 端口的全部设定均相同。

Timer0 选项：如图 3.32 所示，可以设定是否使用 Timer0，是否打开 Time0 溢出中断，以及设定 Time0 的周期、并可以从“Actual Value”中了解实际值与设定值误差，TCNT0 的初值等信息。

Timer1 选项：如图 3.33 所示，除了具有与 Time0 相同的选项外，还可以设定比较寄存器 A 和比较寄存器 B 以及输入捕获等功能。

UART 选项：如图 3.34 所示，在这个选项中可以设定是否使用 UART、8 位或 9 位数据位、RT 和 TX 是否中断、通信波特率的设定与实际值的误差等。

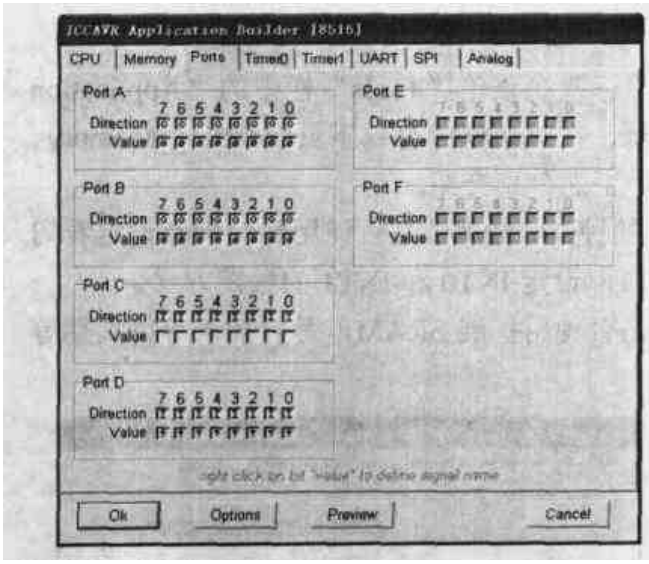


图 3.31 Ports 选项

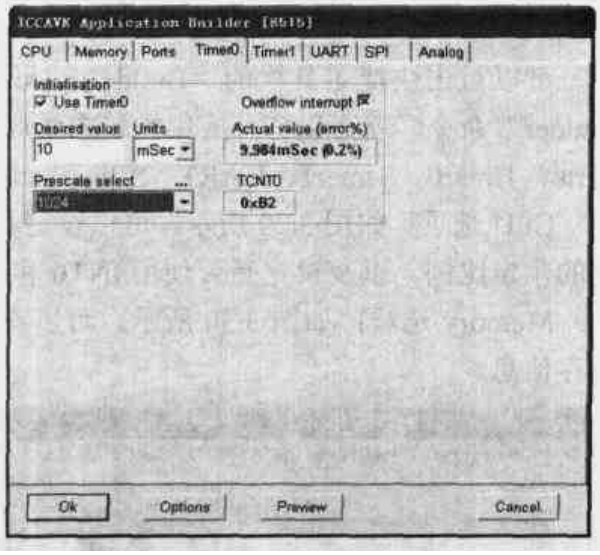


图 3.32 Time0 选项

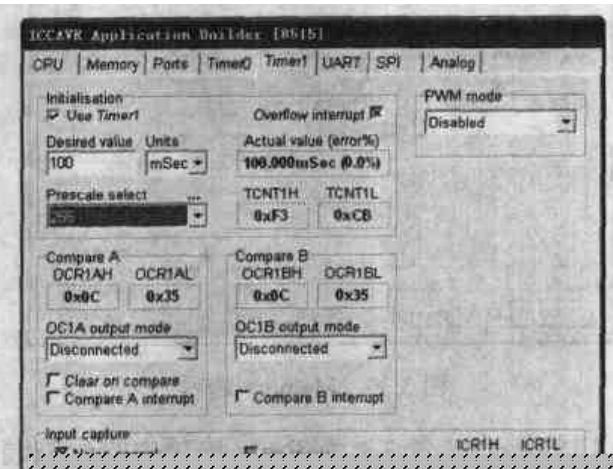


图 3.33 Time1 选项

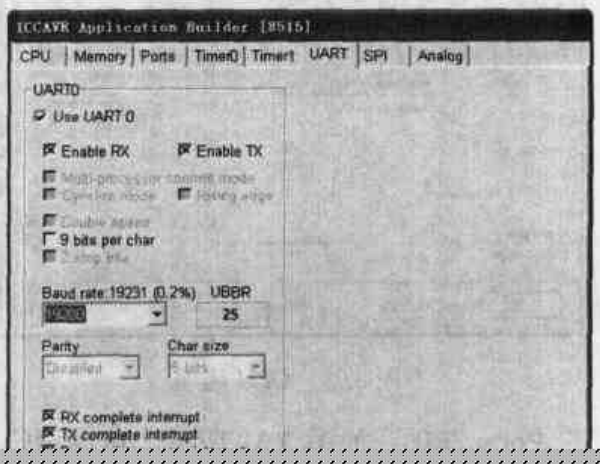


图 3.34 UART 选项

SPI 选项：如图 3.35 所示，可以设定是否使用 SPI 及 SPI 的模式等参数。

Analog 选项：如图 3.36 所示，可设定 MCU 内的模拟比较器的参数。

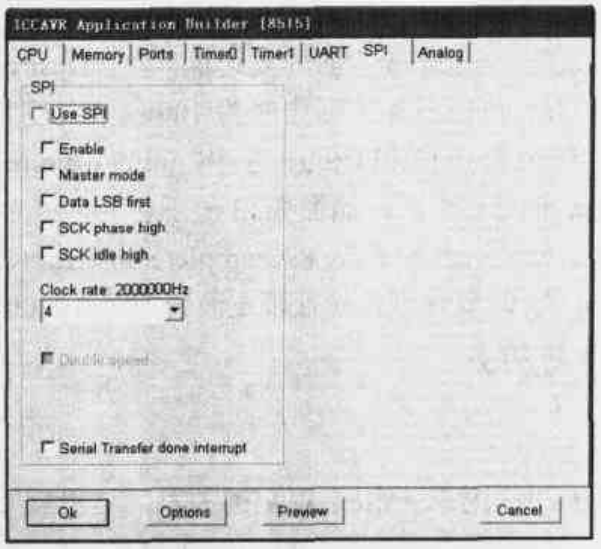


图 3.35　SPI 选项

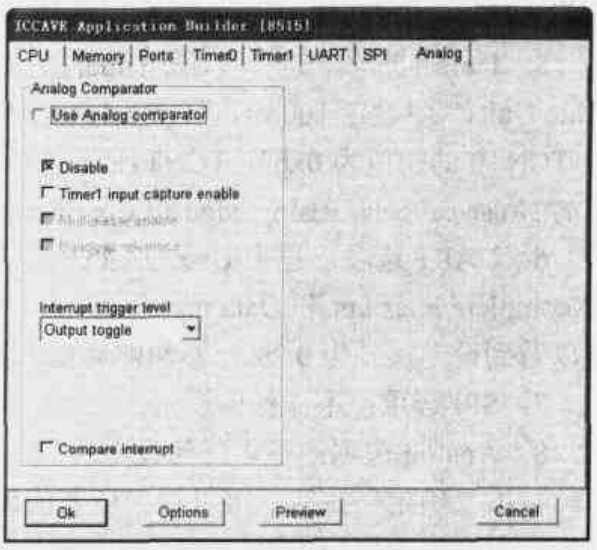


图 3.36　Analog 选项

下面详细说明如何用该向导生成一个所需的硬件初始化文件。假设使用 AT90S8515 芯片，晶振为 8MHz，并且需要使用以下功能：

1．使用看门狗定时器，且预定比例为 128ms(Vcc=+5V)。

2．INT0 定义为下降沿触发，INT1 定义为低电平触发。

3．带有 32KB 的外部 SRAM，访问时需插入一个等待周期。

4．PA、PB 端口全部输出低电平，PC、PD 为输入端口，其中 PC 为高阻输入，PD 打开端口内部的上拉电阻。

5．使用 Time0，定时周期为 10ms 中断。

6．使用外部 T1 引脚的上升沿触发，打开捕获触发中断，打开捕获噪音清除。T1 的引脚输入的外部信号周期不超过 80ms，计划 T1 溢出时间为 100ms，并打开溢出中断(如进入中断则按出错处理)。

7．使用 UART，9 个数据位(含校验位)，打开发送完成、接收完成中断。

8．不使用 SPI 和片内模拟比较器。

根据以上的要求，对“ICCAVR Application Builder”的对话框进行以下设置：

1. CPU 选项：“Target CPU”为 8515，“Xtal speed(MHz)”为 8.0000，在“Watchdog timer”中选中“Enable”前的复选框，“Prescale”为 128K cyclus。“INT0”选择 Falling edge，“INT1”选择 low level，如图 3.29 所示。

2．Memory 选项：选中“Enable external memory”和“Wait states”前的复选框，如图 3.30 所示。

3. Ports 选项：将 Prot A 和 Prot B 的 Direction 和 Value 均设定为“0”，Port C 的 Direction 设定为“I”，Value 设定为空，Port D 的 Direction 设定为“I”，Value 设定为“↑”，如图 3.31 所示。

4．Timer0 选项：选中“Use Timer0”和“Overflow interrupt”前的复选框。在“Desired value Units”中设定 10ms，“Prescale select”为 1024。可以看到实际定时为 9.984ms，与 10ms

误差 0.2%，TCNT0 的初值为 0xb2，如图 3.32 所示。

5．Timer1 选项：选中“Use Timer1”和“Overflow interrupt”前的复选框。在“Desired value Units”中设定 100ms，“Prescale select”为 256。可以看到实际定时为 100ms，误差为 0，TCNT0 的初值为 0xb2，TCNT1H 和 TCNT1L 的值为 0xf3 和 0xcb、选中“input capture”中的 Noise cancel、Rising edge 和 Capture interrupt 前的复选框，如图 3.33 所示。

6．UART 选项：选中 Use UART0、Enable RX、Enable TX、RXcomplete interrupt、TXcomplete interrupt 和 Data register empty interrupt 前的复选框，并且设定波特率为 19200，可以看到设定误差为 0.2%，不影响通信，如图 3.34 所示。

7．SPI 选项：不作任何设定。

8．Analog 选项：不作任何设定。

按下“确定”按钮后可以得到所需的硬件初始化程序段，限于篇幅的关系，这里就不再列出所产生的程序代码。

3.6 ICCAVR 6.26C 支持的库函数介绍

C 语言一般都提供很多库函数供用户调用，在 ICCAVR 的安装文件夹中有一个 LIB 文件夹，就提供了标准 C 库和 AVR 专用的库供用户调用，用户也可以创建或修改自定义的库。如果一个库中的函数被引用，链接器会将该相关代码链接到模块或生成的输出文件中。

下面对 ICCAVR 6.26C 所提供的库函数及双龙电子增补的库函数作简单的介绍。

3.6.1 头文件

ICCAVR 的头文件的形式为：

```
# include "io*v.h"
```

或

```
# include "iom*v.h"
```

其中如果使用的芯片为非 Mega 类，则应使用# include "io*v.h"定义，如果使用的芯片为 Mega 类，则应使用# include "iom*v.h"定义，其中“*”用相应芯片的型号代替，如芯片为 ATmega8，则头文件为：

```
# include "iom8v.h"
```

ICCAVR 提供的头文件定义了相应 MCU 的具体寄存器，这些定义是从 ATMEL 的标准文档经过修改后得到的，以匹配 ICCAVR 编译器的语法要求。

ICCAVR 自 6.26 版本开始，使用新的头文件，在旧的头文件的型号后加入“v”。如 ICCAVR6.25 版使用的头文件为“iom161.h”，在 6.26 及以后的版本中均应改为“iom161v.h”。但实际在 6.26C 版本中也可以兼容旧的头文件，使用旧的头文件也可以正常编译，但会产生一个指示性警告，可以不必理会，大部会情况也可以得到正确的编译结果。新版头文件修正了旧版头文件中的一些端口和中断向量的定义错误，如果恰好使用到有错误定义的头文件编译，会产生意想不到且难以查找的错误，建议读者在使用 6.26 版本以上的编译器时，还是应使用新的头文件。

注意：

在 ICCAVR 中规定：如果函数的参数传递和返回值为比整型参数小的类型(如 char)，编译器都将其扩展成整型(int)数据后再传递或返回，所以下面介绍的字符型函数的参数及返回值都是 int 型而不是通常的 char 型。但是通常使用时可以不必考虑这么多，这个规定一般情况下并不会影响编译结果的正确性。以 int isalnum(int c)函数为例对以下两种情况分别分析：

1. 在使用该函数时给函数实参一个 char 型变量，ICCAVR 在编译时会按赋值兼容的原则进行转换，将 char 扩展成 int 型后再对函数进行编译。无论是 signed char 型还是 unsigned char 型变量，在扩展成相应的 signed int 或者 unsigned int 型后，其值不变，因此并不影响函数执行结果的正确性。

2. 将 int isalnum(int c)函数返回的 int 型变量赋给相应的 char 型变量，ICCAVR 在赋值时只是将 int 型变量的低 8 位赋给 char 型变量，而将高 8 位丢弃，赋给 char 型变量的值恰好就是所需要的值，因此也不会影响生成结果的正确性。

当然也可以对函数的实参和返回值进行强制类型转换，但这会降低程序的可移植性。

3.6.2　库源代码

ICCAVR 提供了全部函数库的源代码及部分应用实例，建议读者能够将这些源代码和应用实例全部看一遍。看完这些源代码，对于学习 C 语言的编程是非常有益的，而且对于理解库函数形参的类型及作用，准确使用库函数也是有帮助的。

因为库源代码(libsrc.zip)是一个加密的 ZIP 压缩文件，在解压缩前，电脑中必须先安装 WINZIP 或 WINRAR 压缩/解压缩软件。库源代码在默认安装情况下的路径为 c:\icc\libsrc.avr\，(其中 c:\icc 视安装路径而不同)。对于正版用户，在注册后，可以从 Help->Library Source Code Password 菜单中得到库源代码的密码。对于非正版用户，只能对其进行解密。可以到互联网上下载任意一个能够解密 ZIP 文件的软件(可到 google 以“解密”为关键词进行搜索)，对它进行解密后即可以得到密码，以现在 PC 机的速度，不长的时间就可以解密成功。提示，密码为 6 位大写字母。

3.6.3 macros.h

ICCAVR 中的 macros.h 定义了一些常用的宏，如 bit(x)及看门狗复位、开/关全局中断等宏，在使用这些宏时，必须包含以下的预处理行：

```
#include "macros.h"
```

限于本书的篇幅，就不一一列举 macros.h 中的宏的内容，希望读者认真地看一看。

3.6.4 字符类型函数

下列函数支持 ASCII 字符运算，使用这些函数之前应当用#include "ctype.h"预处理。

int isalnum(int c)：如果 c 是字母或数字返回非零数值，否则返回零。

int isalpha(int c)：如果 c 是字母返回非零值，否则返回零。

int iscntrl(int c)：如果 c 是控制字符(如 FF、BELL、LF 等)返回非零值，否则返回零。

int isdigit(int c)：如果 c 是数字返回非零值，否则返回零。

int isgraph(int c)：如果 c 是一个可打印字符而非空格返回非零值，否则返回零。

int islower(int c)：如果 c 是小写字母返回非零值，否则返回零。

int isprint(int c)：如果 c 是一个可打印字符返回非零值，否则返回零。

int ispunct(int c)：如果 c 是一个可打印字符而不是空格、数字或字母返回非零值，否则返回零。

int isspace(int c)：如果 c 是空格、CR、FF、HT、NL 和 VT 中的一个返回非零值，否则返回零。

int isupper(int c)：如果 c 是大写字母返回非零值，否则返回零。

int isxdigit(int c)：如果 c 是十六进制数字返回非零值，否则返回零。

int tolower(int c)：如果 c 是大写字母返回 c 对应的小写字母，其他类型返回 c。

int toupper(int c)：如果 c 是小写字母则返回 c 对应的大写字母，其他类型仍然返回 c。

3.6.5 浮点类型函数

下列函数支持浮点数运算，使用这些函数之前必须用#include "math.h"预处理。

float asin(float x)：以弧度形式返回 x 的反正弦值。

float acos(float x)：以弧度形式返回 x 的反余弦值。

float atan(float x)：以弧度形式返回 x 的反正切值。

float atan2(float x,float y)：返回 y/x 的反正切，其范围在$-\pi \sim +\pi$之间。

float ceil(float x)：返回对应 x 的一个整型数，小数部分四舍五入。

float cos(float x)：返回以弧度形式表示 x 的余弦值。

float cosh(float x)：返回 x 的双曲余弦值。

float exp(float x)：返回以 e 为底 x 的幂，即 e^x。

float exp10(float x)：返回以 10 为底的 x 幂，即 10^x。

float fabs(float x)：返回 x 的绝对值。

float floor(float x)：返回不大于 x 的最大整数。

float fmod(float x,float y)：返回 x/y 的余数。

float frexp(float x,int *pexp)：把浮点数 x 分解成数字部分 y(尾数)和以 2 为底的指数 n 两个部分，即 $x=y*2^n$，y 的范围为 $0.5 \leqslant y < 1$，y 值被函数返回，而 n 值存放在 pexp 指向的变量中。

float fround(float x)：返回最接近 x 的整型数。

float ldexp(float x,int exp)：返回 $x*2^{exp}$。

float log(float x)：返回 x 的自然对数。

float log10(float x)：返回以 10 为底的对数。

float modf(float x,float *pint)：把浮点数分解成整数部分和小数部分，整数部分存放到 pint 指向的变量，小数部分应当大于或等于 0 而小于 1，并且作为函数返回值返回。

float pow(float x,float y)：返回 x^y 值。

float sqrt(float x)：返回 x 的平方根。

float sin(float x)：返回以弧度形式表示的 x 的正弦值。

float sinh(float x)：返回 x 的双曲正弦函数值。

float tan(float x)：返回以弧度形式表示的 x 的正切值。

float tanh(float x)：返回 x 的双曲正切函数值。

说明：ICCAVR 不断的对以前的旧版浮点函数库进行改进，使得改进后的库函数占用空间更小，运行速度更快，但是为了使浮点函数库保持相对的稳定，ICCAVR 将改进后的函数在原先函数名后增加一个“x”作为标记。ICCAVR6.26C 版中提供了以下新的浮点函数：

fabsx，改进前的原型为 fabs。

floorx，改进前的原型为 floor。

ceilx，改进前的原型为 ceil。

fmodx，改进前的原型为 fmod。

modfx，改进前的原型为 modf。

sinx，改进前的原型为 sin。

cosx，改进前的原型为 cos。

tanx，改进前的原型为 tan。

expx，改进前的原型为 exp。

exp10x，改进前的原型为 exp10。

sinhx，改进前的原型为 sinh。

coshx，改进前的原型为 cosh。

tanhx，改进前的原型为 tanh。

asinx，改进前的原型为 asin。

acosx，改进前的原型为 acos。

atanx，改进前的原型为 atan。

atan2x，改进前的原型为 atan2。

frexpx，改进前的原型为 frexp。

ldexpx，改进前的原型为 ldexp。

sqrtx，改进前的原型为 sqrt。

logx，改进前的原型为 log。

log10x，改进前的原型为 log10。

powx，改进前的原型为 pow。

这些新浮点函数的参数和返回值均与改进前的函数相同，读者可以直接调用新的浮点函数，也可以用 include "mathx.h"预处理命令代替 include "math.h"预处理，ICCAVR 编译器在预处理时会自动将源程序中旧版函数用相应的新版函数替代。

3.6.6 标准输入/输出函数

前面提到，ANSI C 中的输入输出函数大部分是不能真正移植入微控制器(MCU)的，在 ICCAVR 中，也只支持 ANSI C 中的一些 I/O 函数，ANSI C 中 stdio.h 的许多函数不可以使用。同样使用这些可以使用的函数之前应用#include "stdio.h"预处理，并且还需要初始化输出端口(默认为 UART)。最底层的 I/O 函数是单字符的输入(getchar)和输出(putchar)程序，如果需要针对不同的装置使用高层的 I/O 函数，例如用 printf 输出至 LCD，则需要全部重新定义最底层的函数。

为在 ATMEL 的 AVR Studio 模拟器终端的 I/O 窗口中使用标准 I/O 函数，应当选中“AVR Studio Simulator IO (Project->Potions->Target)”复选框，并且应在 Compiler 选项中的 AVR Studio Version(COFF)选择框中选择所使用相应 AVR Studio 的版本。

int getchar( )：使用查寻方式从 UART 返回一个字符。

int printf(char *fmt,...)：按照格式说明符输出格式化文本 fmt 字符串，格式说明符是标准格式的一个子集。

%d：输出有符号十进制整数。

%o：输出无符号八进制整数。

%x：输出无符号十六进制整数，字母使用'a'~'f'。

%X：输出无符号十六进制整数，字母使用'A'~'F'。

%u：输出无符号十进制整数。

%s：输出一个以空字符('\0'或 NULL)为结束符的字符串。

%c：以 ASCII 字符形式输出，只输出一个字符。

%f：以小数的格式(aaa.bbbb)输出一个浮点数。

%S：输出在 FLASH 存储器中的字符串变量。

%g 或%G：以十进制小数或科学计数法输出一个浮点数(自动选择占用输出宽度较小的一种格式)。

在 ICCAVR 中，可以精确的控制输出的格式，具体如下：

```
%[标记]*[宽度][.精度][l]<需转化字符>
```

标记：

1．如果在“%”和“x”或“X”之间有一个“#”标记，那么会自动在输出数据的前面分别加上 0(八进制)或 0x(十六进制)的引导。

2．“-”(负号)，则输出数据左对齐。

3．“+”(正号)，则输出正整数时在前面加上一个“+”号。

4．“0”(零)，用“0”代替输出的空格。

宽度：

格式说明符只能是一个大于零的正整数，用于指定输出的最小宽度，当指定的最小宽度小于或等于输出的实际宽度时，应指定宽带格式无效。如果指定宽度大于输出实际宽度时，则按照标记指定的格式插入相应的空格或零。

精度：

只能是一个大于零的正整数，当后面跟着“s”格式说明符时，用于指定输出的字符的个数，当后面跟着其他格式说明符时，用于指定输出小数点以后数据的位数。

l(L 的小写字母)：

如果在“%”和一个整数格式字符之间有一个“l”字符，则输出长整型整数。

在 ICCAVR 的设置中有一个 PRINTF Version 选项(Project->Options->Target)，有三种输出形式：

短整型(或基本型)(small)：只支持%c、%d、%x(%X)、%u、%s 五种输出格式。

长整型(long)：除了支持短整型类，还增加%ld、%lx(%lX)、%lu。

浮点型(float)：除了支持长整型类，还增加%f。

采用何种输出类型，取决于实际中输出精度的需要，只要选取适当的输出类型即可。从短整型(small)→长整型(long)→浮点型(float)，程序代码增加很多，运行速度降低。选用长整型和浮点型参数时，还应注意增加硬件堆栈的字节数，防止堆栈溢出。

int putchar(int c)：向 UART 输出一个字符。要注意，如果输出到 Windows 的超级终端时，应在每一个输出的字符后面再输出“\n”，才能得到正确的显示结果，具体请参见第 5 章的应用实例。

int puts(char *s)：输出指针 s 指向的一个字符串。

int sprintf(char *buf,char *fmt)：按照格式说明符输出格式化文本 frm 字符串到一个缓冲区，格式说明符同 printf ( )。

int scanf(char *fmt,...)：按照 fmt 格式说明符输入数据。

注意：

(1) 本函数是通过调用 getchar( )读入的。

(2) 如果在“格式控制”字符串中除了格式说明以外还有其他字符，则在输入数据时应输入与这些字符相同的字符。

(3) 在 ICCAVR 6.26C 中 scanf 函数只支持以下的格式控制字符：

d 输入十进制整数。

x 输入以 0x 开头无符号的十六进制整数。

X 输入以 0X 开头无符号的十六进制整数。

u 输入无符号的十进制整数。

o 输入无符号的十进制整数，与 u 相同。

c 输入一个字符。

l(字母 l) 插在%号之后，d、x、X、u、o 前，说明为长整数类型。

int sscanf(char *buf char *fmt, ...)：除了输入为缓冲区数据外，其他与 scanf 相同。

int cprintf(char *fmt,...)：“const char *”支持函数，除了操作对象只能是在 Flash 中常数字符串外，其余与 printf 函数相同。

int csprintf(char *buf,char *fmt)：“const char *”支持函数，除了操作对象只能是在 Flash 中常数字符串外，其余与 sprintf 函数相同。

3.6.7 读/写内置 EEPROM 函数

int EEPROMwrite(int location, unsigned char)：向 EERPOM 中写入一个字符，适用于大于 256 字节 EEPROM 的 AVR 单片机。

unsigned char EEPROMread(int)：从 EEPROM 中读出一个字符，适用于大于 256 字节 EEPROM 的 AVR 单片机。

int _256EEPROMwrite(int location, unsigned char)：向 EEPROM 中写入一个字符，适用于小于或等于 256 字节 EEPROM 的 AVR 单片机。

unsigned char _256EEPROMread(int)：从 EEPROM 中读出一个字符，适用于小于或等于 256 字节 EEPROM 的 AVR 单片机。

void EEPROMReadBytes(int addr, void *ptr, int size)：从 EEPROM 中读出 size 个字节，适用于全部的 AVR 单片机。

void EEPROMWriteBytes(int addr, void *ptr, int size)：向 EEPROM 中写入 size 个字符，适用于全部的 AVR 单片机。

3.6.8 标准库和内存分配函数

标准库头文件"stdlib.h"定义了宏 NULL、RAND_MAX 和新定义的类型 size_t，并且描述了下列函数，使用这些函数之前必须用#include "stdlib.h "预处理。注意在调用任意内存分配程序(如 calloc、malloc 和 realloc)之前，必须调用_NewHeap 函数来初始化堆(heap)。

int abs(int i)：返回 i 的绝对值。

int atoi(char *s)：转换字符串 s 为整型数并返回它，字符串 s 起始必须是整数形式字符，否则返回 0。

double atof(const char *s)：转换字符串 s 为双精度浮点数并返回它，字符串 s 起始必须是浮点数形式字符串，否则返回 0。

long atol(char *s)：转换字符串 s 为长整型，并返回它，字符串 s 起始必须是长整型字符串，否则返回 0。

void *calloc(size_t nelem,size_t soze)：分配 nelem 个数据项的内存连续空间，每个数据项的大小为 size 字节并且初始化为 0。如果成功返回分配内存单元的首地址，否则返回 0。

void exit(status)：终止程序运行，退出 main 函数，进入启动函数所定的退出点，并在该点死循环。

void free(void *ptr)：释放 ptr 所指向的内存区。

void ftoa(char *buf, float f)：把 ASCII 字符串转换成 xxxx.yyy 格式，注意*buf 指向的数组(或缓冲区)至少 11 个字节。

void itoa(char *buf, int value, int base)：转换整数为字符串。

void ltoa(char *buf, long value, int base)：转换长整数为字符串。

void *malloc(size_t size)：分配 size 字节的存储区，如果分配成功则返回内存区地址，如内存不够分配则返回 0。

void _NewHeap(void *start,void *end)：初始化内存分配程序的堆(空间)。典型的应用是将符号_bss_end+1 的地址用作“start”值，符号_bss_end 定义为编译器用来存放全局变量和字符串的数据内存的结束，加 1 的目的是堆栈检查函数使用_bss_end 字节存储为标志字节。例：

```
extern char _bss_end;
_NewHeap(&_bss_end+1,& _bss_end+201);          //初始化 200 字节大小的堆(空间)
```

int rand(void)：返回一个在 0 和 RAND_MAX 之间的随机数。

void *realloc(void *ptr,size_t size)：重新分配 ptr 所指向内存区的大小为 size 字节，size 可比原来大或小，返回指向给内存区的地址指针。

void srand(unsigned seed)：初始化随后调用的随机数发生器的种子数。

long strtol(char *s,char **endptr,int base)：按照“base.”的格式转换“s”中起始字符为长整型数。如果“endptr”不为空，*endptr 将设定“s”中转换结束的位置。

unsigned long strtoul(char *s,char **endptr,int base)：除了返回类型为无符号长整型数外，其余同“strtol”。

3.6.9 字符串函数

下列函数支持字符串的操作，使用这些函数之前必须用#include "string.h"预处理。在"string"中定义了 NULL、类型 size_t 和下列字符串及字符阵列函数。

void *memchr(void *s,int c,size_t n)：在字符串 s 中搜索 n 个字节长度以寻找与 c 相同的字符，如果成功，返回匹配字符的地址指针，否则返回 NULL。

int memcmp(void *s1,void *s2,size_t n)：对字符串 s1 和 s2 的前 n 个字符进行比较，如果相同返回 0，如果 s1 中字符大于 s2 中字符，则返回 1，如果 s1 中字符小于 s2 中字符，则返回-1。

void *memcpy(void *s1, void *s2, size_t n)：复制 s2 中 n 个字符至 s1。

void *memmove(void *s1,void *s2,size_t n)：复制 s2 中 n 个字符至 s1，其他与 memcpy 基本相同，但复制区可以重叠。

void *memset(void *s,int c,xize_t n)：在 s 中填充 n 个字节的 c，它返回 s1。

char *strcat(char *s1,char *s2)：复制 s2 到 s1 的结尾，返回 s1。

char *strchr(char *s1,int c)：在 s1 中搜索第一个出现的 c，包括结束 NULL 字符。如果成功，返回指向匹配字符的指针，如果没有找到匹配字符，返回空指针。

int strcmp(char *s1,char *s2)：比较两个字符串，如果相同返回 0，如果 s1>s2 则返回 1，如果 s1<s2 则返回-1。

char *strcpy(char *s1,char *s2)：复制字符串 s2 至字符串 s1，返回 s1。

size_t strcspn(char *s1,char *s2)：在字符串 s1 搜索与字符串 s2 匹配的第一个字符，包括结束 NULL 字符，其返回 s1 中找到的匹配字符的索引。

size_t strlen(char *s)：返回字符串 s 的长度，不包括结束 NULL 字符。

char *strncat(char *s1,char *s2,size_t n)：复制字符串 s2(不包含结束 NULL 字符)中 n 个字符到 s1，如果 s2 长度比 n 小，则复制完 s2 后就返回。

int strncmp(char *s1,char *s2,size_t n)：基本和 strcmp 函数相同，但其只比较前 n 个字符。

char *strncpy(char *s1,char *s2,size_t n)：基本和 strcpy 函数相同，但其只复制前 n 个字符。

char *strpbrk(char *s1,char *s2)：基本和 strcspn 函数相同，但它返回的是在 s1 匹配字符的地址指针，否则返回 NULL 指针。

char *strrchr(char *s,int c)：在字符串 s 中搜索最后出现的 c，并返回它的指针，否则返回 NULL。

size_t strspn(char *s1,char *s2)：在字符串 s1 搜索与字符串 s2 不匹配的第一个字符，包括结束 NULL 字符，其返回 s1 中找到的第一个不匹配字符的索引。

char *strstr(char *s1,char *s2)：在字符串 s1 中找到与 s2 匹配的子字符串，如果成功，返回 s1 中匹配字符串的地址指针，否则返回 NULL。

“const char *”支持函数，下面这三个函数的操作对象只能是在 Flash 中常数字符串外，其余与相应的函数相同。

size_t cstrlen(const char *s)

char *cstrcpy(char *dst, const char *src)

int cstrcmp(const char *s1, char *s2)

3.6.10　变量参数函数

"stdarg.h"提供再入式函数的变量参数处理，它定义了不确定的类型 va_list 和三个宏，使用这些函数之前必须用#include "stdarg.h"预处理。

va_start(va_list foo,<last-arg>)：初始化变量 foo。

va_arg(va_list foo,<promoted type>)：访问下一个参数，分派指定的类型。注意，那个类型必须是高级类型，如 int、long 或 double。小的类型如“char”不被支持。

va_end(va_list foo)：结束变量参数处理。

例如：printf ( )可以使用 vprintf ( )来实现。

```
#include "stdarg,h"
int printf(char *fmt,…)
     {
     va_list ap;
     va_start(ap,fmt);
     vfprintf(fmt,ap);
     va_end(ap);
     }
```

3.6.11　堆栈检查函数

ICCAVR 使用两个堆栈：一个用于子程序调用和中断操作的硬件堆栈，一个用于传递参数、临时变量和局部变量的软件堆栈。硬件堆栈是从数据内存的顶部开始分配的，在硬件堆栈下面再分配一定数量的字节作为软件堆栈。硬件堆栈和软件堆栈均为向下生长型的堆栈(注意：这与 51 单片机相反)，堆栈的使用请参见图 3.7。

如果函数的调用层次太深，有可能会发生硬件堆栈溢出到软件堆栈中，改变了软件堆栈中数据的内容。同样，当定义了太多的局部变量或一个局部集合变量太多也有可能出现软件堆栈溢出到动态分配的数据区，两个堆栈都有可能溢出，如果堆栈溢出，会引起不可预测的错误。可以使用堆栈检查函数检测两个堆栈是否溢出。在 Target 的页面中有一个 Return Stack Size 选项，用于指定硬件堆栈(保存函数返回值)的大小，通常如果子程序调用嵌套不深(不超过 4 层)，那么使用默认的 16 字节就足够了，如果使用了浮点函数，则至少应设定为 30 个字节。在一般情况下，除了层次很深的递归调用及使用了%f 格式说明符外，设定为 40 个字节就足够了。

启动代码在硬件堆栈和软件堆栈的最低字节分别写进一个代码(0xaa)，把这个代码称为警戒线。如果硬件堆栈和软件堆栈溢出过，则警戒线字节的代码(0xaa)就会改变，堆栈检查函数就是通过检查这两个堆栈的最低字节的代码是否被改变来判断两个堆栈是否溢出。通过调用_StackCheck(void)函数来检查堆栈溢出，如果警戒线字节中的代码仍然保持正确的值，那么函数检查通过，没有溢出。如果堆栈溢出，那么警戒线字节将可能被破坏，_StackCheck(void)函数检查到警戒线判断字节中的代码被改变，就判断相应的堆栈溢出(当程序堆栈溢出，程序可能运行不正常或偶然崩溃)，该函数再调用函数_StackOverflowed(char c)，如果参数是 1，那么硬件堆栈有过溢出；如果参数是 0，那么软件堆栈曾经溢出。

在使用堆栈检查函数时应注意以下几点：

(1) 在使用堆栈检查函数时，必须先用#include "macros.h"预处理。

(2) 如果使用自己的启动文件，在 ICCAVR6.20 以后的版本中，如果使用的启动文件中没有警戒线的内容，ICCAVR 也会自动添加警戒线。而在 ICCAVR6.20 以前的版本中，必须自己添加该部分内容，否则生成的代码中堆栈分配将不带警戒线。

(3) 如果使用动态内存分配，必须跳过警戒线字节_bss_end 来分配堆(即增加一个字节)，详见内存分配函数说明。

(4) 当_StackCheck(void)函数检测到警戒线字节被改变，则会调用一个默认的_StackOverflowed 函数来跳转到程序存储器 0 的位置(复位向量地址)。可以指定或重新编写一个新的函数来代替它，例如可以用新函数来指示是哪个堆栈溢出等，但这个函数也不可能执行太多的功能或让程序恢复到正常状态。因为堆栈溢出后，会更改掉一些有用的数据，引起不可预测的错误，甚至使程序死机。

下面用一个简单的实例来说明堆栈检查函数的作用：

```
main()
  {
  init()                                  //调用初始化程序
  float a,b;
  a=1.0;
  b=1.0;
  printf("a=%f\n", a);
  printf("b=%f\n", b);
  _StackCheck();                          //调用堆栈检查函数
  }
_StackOverflowed(char c)
  {
  if (c==1)
      puts("trashed HW stack");           //硬件堆栈溢出
  else
      puts("trashed SW stack");           //软件堆栈溢出
  }
```

3.6.12　双龙电子增补的库函数

需要增补库函数的读者可以到双龙电子的网站(http://www.sl.com.cn)下载，这些函数的硬件连接均针对双龙电子的 SLAVR 下载实验器，请参见附录 A 中 SLAVR 的原理图。如果读者需要在其他工程中使用这些增补的库函数时要注意，因为部分增补的库函数是专门针对 SL-AVR 硬件链接方式，读者需要进行相应的修改才能适用其他的硬件链接方式，本书在第 5 章的应用实例中提供部分增补库函数的源代码。

需要使用这些库函数前，应将库文件 libslavr.a 复制到 ICCAVR 安装目录下的库文件目录中(默认安装路径：c:\icc\LIB)，将头文件 slavr.h 复制到头文件目录中(默认安装路径：c:\icc\include)；在工程选项中(Project->Options->Target)的附加库(Additional LIB)中填入“slavr”。注意，引号不需要输入，不要加扩展名，而且在文件头必须使用#include "slavr.h" 预处理。

void write_lcd(unsigned char data,unsigned char data_type)：写命令或数据到 LCD 模块，可直接使用控制命令对 LCD 模块操作，其中 data:命令或数据字节，data_type 为 0 表示命令，1 表示数据。

unsigned char read_lcd(unsigned char data_type)：从 LCD 模块中读取数据或地址，其中 data_type 为 0 表示读取地址，1 表示读取数据。

void lcd_init(void)：LCD 模块初始化。

void lcd_putc(unsigned char c)：单字符输出至 LCD 函数。

void lcd_puts(unsigned char *s)：字符串输出至 LCD 函数。

void lcd_clear(void)：清除 LCD 显示函数。

void lcd_write(unsigned char adder,unsigned char data)：写数据到 LCD 的指定 DDRAM/CGRAM 位置。

void lcd_write_data(unsigned char data)：写数据到 LCD 的 DDRAM 当前位置。

unsigned char lcd_read(unsigned char adder)：从指定 DDRAM/CGRAM 位置读取数据。

void lcd_gotoxy(unsigned char x,unsigned char y)：将光标转移到 LCD 的 DDRAM 指定位置处，其中 x 值的范围为 0~39，y 的值的范围为 0~1。

void lcd_shift(unsigned char n,unsigned char p)：LCD 字符移位函数，n 为移动的次数，p 为移动方向，'l'或'L'表示左移，'r'或'R'表示右移。

void delay_1ms(void)：延时 1 毫秒函数，注意：对应的晶振为 8MHz，在“5.2 延时函数”一节中给出源代码。

void delay_ms(unsigned int n)：延时 n 毫秒函数，注意：对应的晶振为 8MHz，在“5.2 延时函数”一节中给出源代码。

unsigned char scan_key(void)：键盘扫描函数，注意：本函数不作按键释放检查，在“5.5 键盘扫描程序”一节中给出源代码。

返回值：

(1) 没有键按下则返回 0x7f；

(2) 如果先按 shift 键，再按相应的键，返回值最高位为 1，后 7 位为键名对应的数值；

(3) 直接按下除 shift 以外的键，返回键名对应数值。

unsigned char keypad(void)：键盘扫描函数，注意：本函数作按键释放检查，在“5.5 键盘扫描程序”一节中给出源代码。

返回值与 scan_key 相同。

void Start(void)：I^2C 总线启动。

void Stop(void)：I^2C 总线停止。

void Ack(void)：发送应答信号，I^2C 总线用。

void NoAck(void)：发送非应答信号，I^2C 总线用。

unsigned char TestAck(void)：测试应答信号，有应答信号返回 0，无应答信号返回 1。

void Write8Bit(unsigned char input)：写一个字节数据到 I^2C 总线。

unsigned char Read8Bit(void)：从 I^2C 总线读取一个字节数据。

void Write24c02(unsigned char *Wdata,unsigned char RomAddress,unsigned char number)：对串行 EEPROM 存贮器 AT24C02 采用页写模式连续写多个字节。

参数说明：

(1) unsigned char *Wdata：指向源数据块首地址的指针；

(2) unsigned char RomAddress：EEPROM 中的目标地址；

(3) unsigned char number：连续写的字节数。

void Read24c02(unsigned char *RamAddress,unsigned char RomAddress,unsigned char bytes)：对串行 EEPROM 存贮器 AT24C02 连续读多个字节。

参数说明：

(1) unsigned char *RamAddress：指向存放读出数据的变量的指针；

(2) unsigned char RomAddress：EEPROM 中的目标地址；

(3) unsigned char bytes：连续读的字节数。

void delay_us(int time)：微秒延时程序，在“5.2 延时函数”一节中给出源代码。

unsigned char ds1820_ack(void)：检查 DS1820 是否存在，如果 DS1820 不存在返回 1，否则返回 0，在“5.8 利用单总线访问 DS18B20”一节中给出源代码。

void ds1820_write(unsigned char cmd)：写 ROM 或存贮器命令到 DS1820，在“5.8 利用单总线访问 DS18B20”一节中给出源代码。

unsigned char ds1820_read(void)：从 DS1820 读 1 字节数据，在“5.8 利用单总线访问 DS18B20”一节中给出源代码。

unsigned char crccheck(unsigned char *p,unsigned char len)：CRC 校验，在“5.8 利用单总线访问 DS18B20”一节中给出源代码。

第 4 章　用 ICCAVR C 操作硬件资源

4.1　访问 AVR 的硬件

AVR 单片机的存储器在物理结构上可分为五个部分，访问它们需要使用不同的汇编指令。以 AT90S8515 为例：

1．程序存储器(0x000~0xFFF)，访问时用 LPM 指令访问；

2．片内数据存储器(0x0260~0x025F)，访问时用 STS、LDS 和 ST、LD 指令访问；

3．片外数据存储器(0x0260~0xFFFF)，访问时用 STS、LDS 和 ST、LD 指令访问；

4．32 个通用寄存器(R0~R31)，他们之间数据传送可使用 MOVE 指令；

5．I/O 寄存器(0x00~0x3F)，使用 IN、OUT 指令访问。

AVR 有丰富的外设资源，如 I/O 寄存器、EEPROM、UART、SPI、TIMER/COUNTER、ISP、PWM 和 A/D 变换等。C 语言是一种功能强大的语言，不但能很容易地实现各种高级算法，如排序、队列、软堆栈等，而且对单片机的硬件资源操作也很简单。本章将主要以 AT90S8515 为例，说明如何用 ICCAVR 编译器操作其各种硬件资源。

4.2　位　操　作

4.2.1　位操作的特点

ICCAVR 继承了 ANSI C 的强大位操作功能，并在 AVR 芯片相应的头文件中定义了单片机内部的寄存器(CPU 寄存器和 I/O 寄存器)和位，因此在 ICCAVR 中，对单片机的位操作还是很方便的，ICCAVR 中常用位操作符及含义如表 4.1 所示。

表 4.1　ICCAVR C 位操作定义

表　达　式	位　操　作	含　　义
A|B	按位或	表达式 A 与表达式 B 按位进行或运算
A&B	按位与	表达式 A 与表达式 B 按位进行与运算
A^B	按位异或	表达式 A 与表达式 B 按位进行异或运算
~A	按位取反	表达式 A 按位进行取反运算

说明：A、B 为两个表达式

以 AT90S8515 为例，根据位操作对象的存储位置，ICCAVR C 的位操作可分为以下几种不同的操作对象：

1．通用寄存器(R0~R31)；

2．I/O 寄存器(0x20~0x5f)；

3．内部数据存储器(0x060~0x25f)；

4．外部数据存储器(0x0260~0xFFFF)。

其中对于通用寄存器和 I/O 寄存器可以使用 io8515v.h 头文件中的宏定义来访问，对于内部数据寄存器和外部数据存储器，就只能使用 ANSI C 中的位操作实现访问。

如置位 I/O 寄存器中的 PA7 位，可采用如下方式：

```
PORTA |= BIT(PA7);
```

如果需置位数据寄存器中的第 7 位(char 类型变量 a)，可采用以下方式：

```
a |= (1<<7)
```

或：

```
a |= BIT(7)
```

在 macros.h 中，也定义了一些常用的位操作宏定义，如：

```
#define WDR()  asm("wdr")
#define SEI()  asm("sei")
#define CLI()  asm("cli")
#define NOP()  asm("nop")
#define _WDR() asm("wdr")
#define _SEI() asm("sei")
#define _CLI() asm("cli")
#define _NOP() asm("nop")
```

在程序中需要开全局中断，可以使用以下指令：

```
SEI();
```

或

```
_SEI();
```

4.2.2 位操作的 C 源程序实例及剖析

四种不同的位操作通常用于四种不同的目的，先看下面的一段程序：

[例 4.1]

```
1   #include "io8515v.h"
2   unsigned char gtmpA;          //声明全局变量
3   void main (void)
      {
4     unsigned char tmpB;         //声明局部变量 tmpB
5                                 //初始化系统
6     PORTA |=0x80;               //打开端口 A 的第 7 位(MSB)
7     tmpB |=0x80;                //打开局部变量 tmpB 的第 7 位(MSB)
8     gtmpA |=0x80;               //打开全局变量 gtmpA 的第 7 位(MSB)
9                                 //其他程序语句
      }
```

在上例的第 2 行 unsigned char gtmpA，声明全局变量，变量存储在数据存储器区。如果没有扩展外部 RAM，gtmpA 将被编译器分配在内部数据区 0x060~0x25F；否则，将被分配在外部数据区 0x0260~0xFFFF。

第 4 行 unsigned char tmpB，声明局部变量 tmpB，编译器把局部变量分配到一个通用寄存器(如 R20)，对该局部变量的操作，将被当作对这个通用寄存器的操作。

第 6、7 和 8 行分别进行不同类型变量的“按位或”操作。不管该位原来是“0”或“1”，语句执行后该位都会变成“1”：

```
PORTA |=0x80;                    //将端口 A 的第 7 位(MSB)置“1”
tmpB |=0x80;                        //将局部变量 tmpB 的第 7 位(MSB)置“1”
gtmpA |=0x80;                   //将全局变量 gtmpA 的第 7 位(MSB)置“1”
```

下面将对三种变量类型按不同的位操作及其常用功能分别进行说明，并假设局部变量分配到 R20 通用寄存器。

1.“按位或”用于打开某一位或几位，即置位，针对不同类型的变量，在 ICCAVR 中“按位或”操作与生成的汇编代码比较如表 4.2 所示。

表 4.2　“按位或”操作的 C 源程序与汇编代码比较

变 量 类 型	ICCAVR C 语言源程序	生成相应的汇编代码
I/O 寄存器	PORTA |= 0x80;	SBI　PORTA,7
局部变量	TmpB |= 0x80;	ORI　R20,128
全局变量	GtmpA |= 0x80;	LDS　R24,_gtmpA ORI　R24,128 STS　_gtmpA,R24

从上表的比较可以看出，ICCAVR C 语言源程序与生成的相应汇编代码的比较可以看出，ICCAVR C 的编译效率和语言密度是相当高的。对于 I/O 寄存器的位操作就生成 SBI PORTA,7 指令，对于局部变量的位操作，也能生成 ORI R20,128 指令，均用一条指令完成

操作；对于全局变量的位操作，ICCAVR 处理得也很巧妙，编译器先把全局变量装载到(LDS 指令)一个通用寄存器里(这里是 R24)，然后执行“立即或”操作 ORI R24,128，再把执行的结果存储到(STS 指令)全局变量 gtmpA 中。

2.“按位取反”用于关闭某一位或几位，即清位。

[例 4.1]中，将 6、7、8 行的“按位或”操作改写为“按位取反”操作：

```
PORTA &=~0x80;                    //将端口 A 的第 7 位(MSB)清零
tmpB &=~0x80;                     //将局部变量 tmpB 的第 7 位(MSB)清零
gtmpA &=~0x80;                    //将全局变量 gtmpA 的第 7 位(MSB)清零
```

针对不同类型的变量，在 ICCAVR 中“按位取反”操作与生成的汇编代码比较如表 4.3 所示。

表 4.3 “按位取反”操作的 C 源程序与汇编代码比较

变量类型	ICCAVR C 语言源程序	生成相应的汇编代码
I/O 寄存器	PORTA &= ~0x80;	CBI PORTA,7
局部变量	tmpB &= ~0x80;	ANDI R20,127
全局变量	gtmpA &= ~0x80;	LDS R24,_gtmpA ANDI R24, 127 STS gtmpA, R24

说明：0x80(0b1000000)按位取反后是 0x7F(0b01111111)

I/O 寄存器的按位取反操作编译为 CBI PORTA,7 指令，局部变量储存在 R20 中，按位与生成的汇编代码为 ANDI R20,127，全局变量的按位取反操作相对于局部变量来说，增加了加载(LDS)和存储(STS)语句，总的来说生成代码的密度是相当高的。

注意：AVR 中有些寄存器位的清除是通过写“1”完成的，下面给出使用查询方式向 UART 发送一个字节的例子。

[例 4.2]

```
void sendchar(unsigned char c)
   {
   UDR=c;                         //向 UART 数据寄存器写一个字节 c
   while (!(USR & 0x40));         //等待发送完标志位置“1”
   USR|=0x40;                     //清除发送完标志
   }
```

这个函数把从实参中传递到形参的c字符写到UART数据寄存器UDR，然后等待UART发送完成标志位有效(TXC 置“1”)，最后用写“1”的方式将发送完标志位清零：

```
USR|=0x40;                        //“按位或”操作，将 USR 第 6 位 TXC 置“1”
```

生成的汇编代码为：

```
sbi USR, 6
```

while 语句等待发送完标志位 TXC 置“1”，使用了“按位与”运算和逻辑非“!”运算，生成的汇编语句为：

```
wTXC:                                   //标号为编者加
sbis USR, 6
rjmp wTXC
```

3.“按位异或”用于翻转(取反)某一位或几位。

我们还是看[例 4.1]，将 6、7、8 行的“按位或” 操作改写为“按位异或”操作：

```
PORTA^=0x80;                            //将端口 A 的第 7 位(MSB)翻转
tmpB^=0x80;                             //将局部变量 tmpB 的第 7 位(MSB)翻转
gtmpA^=0x80;                            //将全局变量 gtmpA 的第 7 位(MSB)翻转
```

针对不同类型的变量，在 ICCAVR 中“按位异或”操作与生成的汇编代码比较如表 4.4 所示。

表 4.4　“按位异或”操作的 C 源程序与汇编代码比较

变 量 类 型	ICCAVR C 语言源程序	生成相应的汇编代码
I/O 寄存器	PORTA^=0x80;	LDI R24, 128 IN R2, PORTA EOR R2, R24 OUT PORTA, R2
局部变量	tmpB^=0x80;	LDI R24, 128 EOR R20, R24
全局变量	gtmpA^=0x80;	LDI R24, 128 LDS R2, _gtmpA EOR R2, R24 STS _gtmpA, R2

I/O 寄存器的按位翻转用到了 I/O 存取指令(IN,OUT)，执行异或操作的两个参数必须是通用寄存器，而且还用到临时寄存器 R24 存放 0x80，这种翻转操作有时是很方便的。

如下例中，在 PA.7 连接一个发光二极管，每次执行位翻转操作都会使发光二极管的状态发生变化(点亮/熄灭)，我们将位操作程序放在 1 秒定时的 TIMER1 中断服务程序中进行：

[例 4.3]

```
#pragma interrupt_handler timer1_ovf_isr:7
```

```
void timer1_ovf_isr(void)
    {
    TCNT1H = 0xE3;              //重新装载计数器高字节
    TCNT1L = 0xE0;              //重新装载计数器低字节
    PORTA ^= 0x80;              //翻转 PA.7 状态
    }
```

在上例中，晶振的频率为 7.3728MHz，TIMER1 的中断服务程序每秒执行一次，连接在 PA.7 脚的 LED 就按亮一秒、灭一秒的规则闪动。

4.“按位与”用于检查某一位或几位是否为 1。

“按位与”操作常用在条件或判断语句中，用于检查某一位是否为“1”，见下例：

[例 4.4]

```
#include "io8515v.h"
unsigned char gtmpA;                    //声明全局变量
void main(void)
     {
     unsigned char tmpB;                //声明局部变量 tmpB
                                        //初始化系统，语句略
     while(tmpB&0x80)                   //检查 tmpB 的第 7 位是否为 1
          {
          if(PINA&0x80)                 //检查 PA.7 是否为 1
               PORTA&=~0x80;            //关闭端口 A 的第 7 位(MSB)
          else
               PORTA|=0x80;             //打开端口 A 的第 7 位(MSB)
          if(gtmpA &= 0x80)             //检查 gtmpA 的第 7 位是否为 1
               {
               UDR = 'k';               //向串口发送字符“k”
               while(!(USR & 0x40));    //等待发送完标志位置“1”
               USR|=0x40;               //清除发送完标志
               }
          }
     }
```

假定局部变量 tmpB 存储在 R20，[例 4.4]中生成的汇编代码如下（省略有关初始化的语句）：

```
    RJMP L13
L8:
    SBIS PINA, 7                ; 检查 PA.7 是否为 1
    RJMP L11
```

```
    CBI PORTA, 7                ; 关闭端口 A 的第 7 位(MSB)
    RJMP L12
L11:
    SBI PORTA, 7                ; 打开端口 A 的第 7 位(MSB)
L12:
    LDS R2, gtmpA
    SBRS R2, 7                  ; 检查 gtmpA 的第 7 位是否为 1
    RJMP L13
    LDI R24, 107                ; k 的 ASCII 值是 107
    OUT UDR, R24                ; 向串口发送字符 "k"
L15:
    SBIS USR, 6                 ; 等待发送完标志位置 "1"
    RJMP L15
    SBI USR, 6                  ; 清除发送完标志
L13:
    SBRC R20,7                  ; 检查 tmpB 的第 7 位是否为 1
    RJMP L8
```

相对于汇编而言，C 语言简洁清晰的程序控制流程优势一览无余，同时 ICCAVR 用一系列的测试、分支和跳转指令提高了 C 语言的编译效率和代码密度。对应于 while(tmpB&0x80)的是最后两行，位为 0(因为有个非操作符"!")，则跳过下一条语句(SBRC)和一个跳转语句 RJMP　L8。表示如果 while 条件为假(0)就跳出 while 循环；否则继续执行循环内的指令(跳转到 L8)。

if(PINA&0x80)...else...语句实际上完成 PA.7 位翻转的功能，也可以用上面提到的按位异或解决：

```
PORTA^=0x80;
```

if(gtmpA&=0x80)检查 gtmpA 的第 7 位如果为 1，就向 UART 发送字符 "k"，并等待发送完成标志，最后清除发送完成标志，准备下一次的发送操作，这里清除标志用的是置 "1" 操作(SBI)。

4.2.3　使用单总线访问 DS18B20

1-WIRE 网络(MicroLAN，也称为单总线，是 Maxim 旗下的 Dallas 公司的专利技术)具有严谨的控制结构，其结构示意图如 4.1 所示，一般通过双绞线与 1-WIRE 元件进行数据通信，他们通常被定义为漏极开路端点，主/从式多点结构，而且一般都在主机端接一个上拉电阻至+5V 电源。通常为了给 1-WIRE 设备提供足够的电流，需要一个 MOSFET 管将 1-WIRE 总线上拉至+5V 电源，下面以 Maxim 的 DS18B20X 温度传感器为例说明如何使用 1-WIRE 网络。

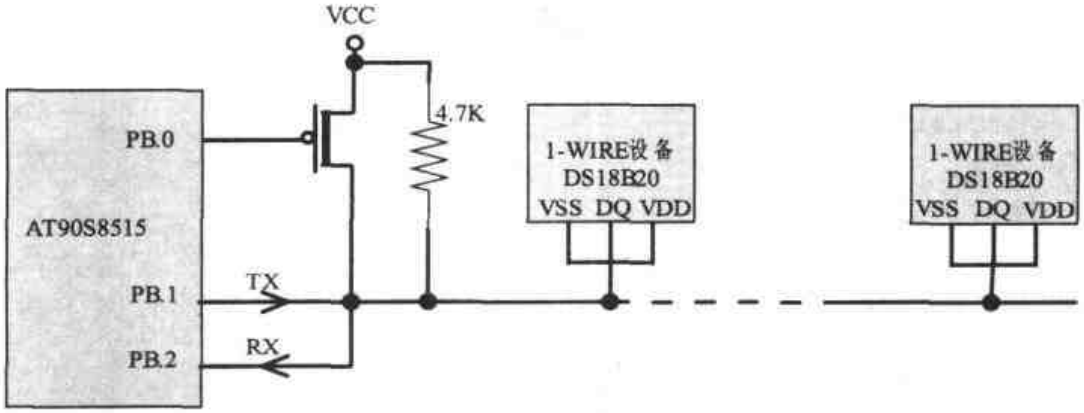


图 4.1 DS18B20X 组成的 1-WIRE 网络

1-WIRE 网络通信协议是分时定义的，有严格的时隙概念，图 4.2 是复位脉冲的时隙。

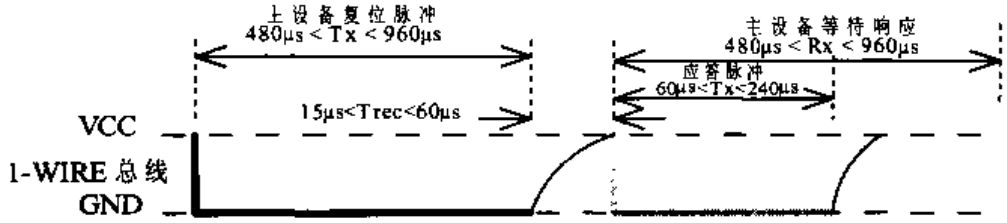


图 4.2 1-WIRE 协议的复位脉冲时隙

1-WIRE 协议定义了几种信号类型：复位脉冲、应答脉冲、读写 0/1 时隙。这些信号除了应答脉冲外，都有主机发出同步信号，并且发送所有的命令和数据都是低位在前、高位在后的传输形式，如图 4.3 所示。

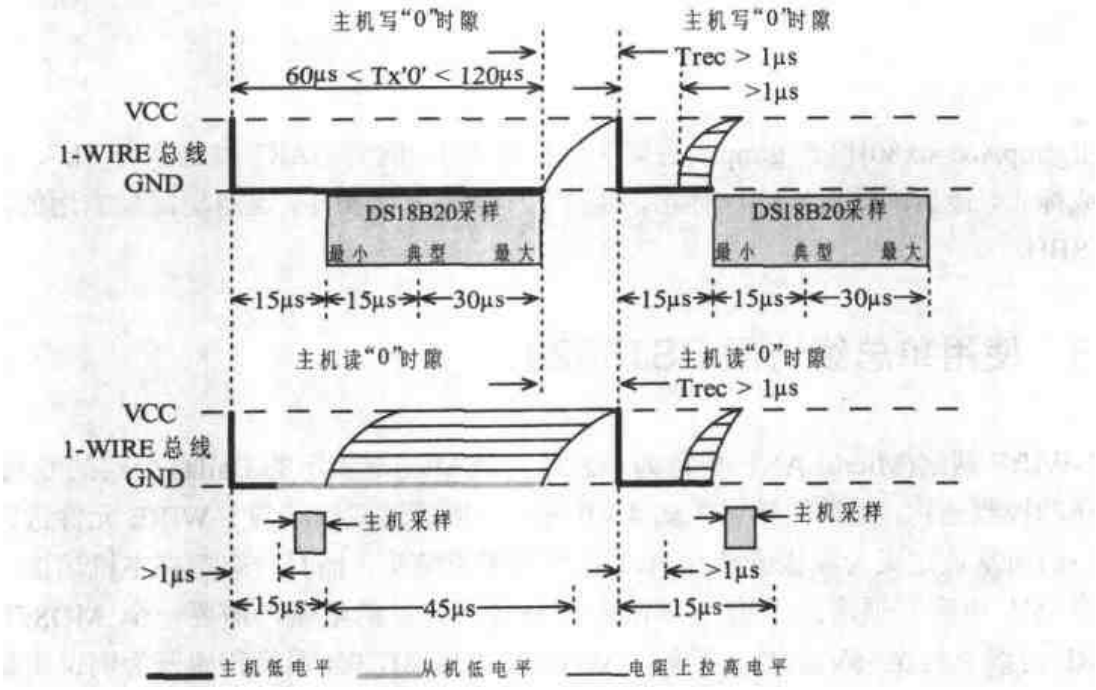


图 4.3 1-WIRE 读写“0/1”时隙

为实现 1-WIRE 协议严格的时隙要求，我们先定义一个微秒级的延时函数。AT90S8515 使用 4MHz 晶振，下面的函数将延时(dt + 2)μs：

[例 4.5]

```
#pragma ctask delay
void delay(unsigned char dt)
    {
    while(dt)
        dt--;
asm("nop");
}
```

由于延时函数并不影响可变寄存器(volatile registers)和 SREG 寄存器，因此在 delay 函数前使用了 ctask 伪指令，说明该函数是 c 任务函数，在调用该函数时不必插入保护和恢复可变寄存器的代码。关于可变寄存器的概念，请参见“2.11.4 编译附注和扩充”一节。

ICCAVR 编译后的汇编程序：

```
rjmp L3;                    2 个时钟
L2:dec R16;                 1 个时钟
L3:tst R16;                 1 个时钟
brne L2;                    2(1)个时钟
nop;                        1 个时钟
ret;                        4 个时钟
```

加上函数调用时需要给 R16 赋值指令(LDI 或 MOV)和 rcall 指令，延时总时间计算公式：

延时时间 = 0.25 * ((dt*4 - 1) + 9) = (dt+2)μs

其中 dt 是函数调用时的参数。

[例 4.6] 使用单总线访问 DS18B20。

说明：本例中使用 AT90S8515 的两个普通 I/O 脚(还有一个上拉控制脚)与 DS18B20 通信。在这里仅列出单总线网络应用的基本函数，只适用于单个单总线从设备的网络，单总线的具体应用实例，请参见第 5 章。

```
/************************************
函数名：reset_test
功　能：复位所有的单总线从器件
输　入：无
返　回：1　没有设备：
        0　检测到设备：
备　注：单总线上多于 480μs 的低电平将会复位总线上的所有单总线从器件
************************************/
```

```
unsigned char reset_test (void)
    {
    unsigned char presence;
    PORTB &=0x02;                  //1-WIRE Bus 为低电平，开始一个时隙
    delay (250);                   //共需延时 480μs
    delay (226);                   //因为函数形参为 unsigned cha，所以分两次延时
    PORTB|=0x02;                   //释放 1-WIRE 总线，上拉电阻将总线拉为高电平
    delay(60);                     //等待从机响应
    presence = (PINB & 0x40);      //读取 1-WIRE 总线
    delay(250);                    //等待 1-WIRE 总线恢复
    delay(170);
    return presence;
    }
/***********************************
函数名：read_bit
功  能：从单总线读取一个位
输  入：无
返  回：读到字节的最低位 LSB
备  注：
***********************************/
unsigned char read_bit (void)
    {
    unsigned char r_bit;
    PORTB&=0x02;                   //开始一个时隙
    delay (4);                     //延时 6μs
    PORTB|=0x02;                   //释放 1-WIRE bus
    delay (7);                     //延时 9μs
    r_bit=(PINB&0X04);             //采样 1-WIRE 总线
    delay (53);                    //等待 45μs 后允许总线浮空；等待 10μs 总线恢复时间
    return (r_bit);
    }
/***********************************
函数名：rite_bit
功  能：写一个位到 1-WIRE 总线
输  入：要写到 1-WIRE 的位，在字节的最低位 LSB
返  回：0  写操作成功
        1  写操作失败
备  注：  写一个位，等待 10μs 后读出，如果两次相同则成功，否则失败
***********************************/
unsigned char rite_bit (unsigned char bitval)
    {
    unsigned char r_bit;
```

```
    PORTB&=0x02;                    //开始一个时隙
    delay (4);                      //延时 6μs
    if (bitval)
    PORTB|=0x02;
    delay (8);                      //延时 10μs
    r_bit=(PINB&0x04);
    delay (53);                     //等待 45μs 后允许总线浮空;等待 10μs 总线恢复时间
    return (r_bit^bitval);
    }
/***********************************
函数名: read_byte
功　能: 从 1-WIRE 总线读一个字节
输　入: 无
返　回: 读出的字节
备　注: 调用读一个位 read_bit 函数, 低位在前
***********************************/
unsigned char read_byte (vaid)
    {
    unsigned char i;
    unsigned char value=0;
    for (i=0; i<8; i++)
          {                         //每次发送一个位, 低位 LSB 在前
          if(read_bit())            // 读取一个位
          value |= (0x01<<i);       //如果该位是"1", 则右移"i"位
          }
    delay(58);
    return (value);
    }
/***********************************
函数名: rite_byte
功　能: 向 1-WIRE 总线写一个字节
输　入: 要写的字节
返　回: 无
备　注: 调用写一个位 rite_bit 函数, 低位在前
***********************************/
void rite_byte (unsigned char value)
    {
    unsigned char i;
    unsigned char temp;
    for (i=0; i<8; i++)
          {                         //每次发送一个位, 低位 LSB 在前
          temp=value>>i;            //右移"i"位
```

```
            temp &=0x01                          //取得要发送的位
            while (!rite_bit(temp));             //1-WIRE 发送一个位，不正确则重发
            }
    delay(58);
    }
/***********************************
函数名：rite_command
功  能：向 DS18B20 发送 ROM 命令
输  入：命令字节
返  回：0  成功
        1  失败
备  注：
***********************************/
unsigned char rite_command(unsigned char c)
    {
    unsigned char val=1;
    if (!reset_test())                           //检测到 1-WIRE 设备
          {
          rite_byte(0xCC);                       //SKIP ROM，1-WIRE 总线上只有一个设备
          rite_byte(c);
          val=0;
          }
    return val;
    }
/***********************************
函数名：read_temperature
功  能：读取 DS18B20 温度寄存器，并转换为整数温度值
输  入：无
返  回：测得温度的整数值
备  注：9 位精度模式
***********************************/
int  read_temperature(void)
    {
    union
         {
         unsigned char c[2];
         int I;
         } temp;
    if(!rite_command(0x44))              //CONVERT_T，命令 DS18B20 开始温度采样变换
         {
         delay(98);                      //延时 100μs
         if(!rite_command(0xBE))         // READ_RAM，读取 DS18B20 寄存器
```

```
            {
            temp.c[0]=read_byte();      //温度寄存器低字节
            temp.c[1]=read_byte();      //温度寄存器高字节
            }
        }
    if(temp.c[1] & 0x80)                    //如果是负温度
            {
            temp.I=(~temp.I) + 1;           //得到温度的绝对值
            temp.I=temp.I>>4;               //取得温度的整数部分
            temp.I=temp.I * (-1);           //加上符号
            }
        else
            {
            temp.I=temp.I >> 4;             //取得温度的整数部分
            }
        return (temp.I) ;                   //返回温度值
        }
```

4.3　程序存储器和常量数据

4.3.1　程序存储器和常量数据的特点

AVR 单片机是哈佛结构(Harvard)的 MCU，对程序和数据带有不同的存储器和总线，通过单一级的流水线可对程序存储器进行访问，当执行某一指令时，下一指令被预先从程序存储器中取出，这使得每一个时钟周期都可以执行一条指令。另外，哈佛总线结构使得 AVR 单片机能比传统结构(冯诺依曼结构)有更多的存储器空间。尽管 AVR 单片机不支持扩充程序存储器，但其宽广的产品线(从 1K 字节到 8M 字节程序存储器)，可以满足绝大部分的嵌入式应用场合。

C 语言并不是专门为哈佛结构 MCU 所设计的，以指针为例，ANSI C 规定指向数据或函数的 C 指针不能指向各自所在以外的存储空间，但是哈佛结构的 AVR 要求数据指针既能指向数据存储器空间，也能指向程序存储器空间。比如既要能访问数据存储器中的变量数据，也要能够访问程序存储器中的常量数据，只能使用非标准 C 解决这个问题，ICCAVR C 中对标准 C 中的“const”限定词进行扩充，用“const”表示目标在程序存储器中。

AVR 读取程序存储器中的数据使用 LPM 指令(如图 4.4 所示)，而 LPM 指令使用 16 位 Z 指针作为访问程序存储器地址，并且将返回 8 位数据值存放在 R0 中。LPM 指令能寻址程序存储器第一个 64K 字节(32K 字)的空间。

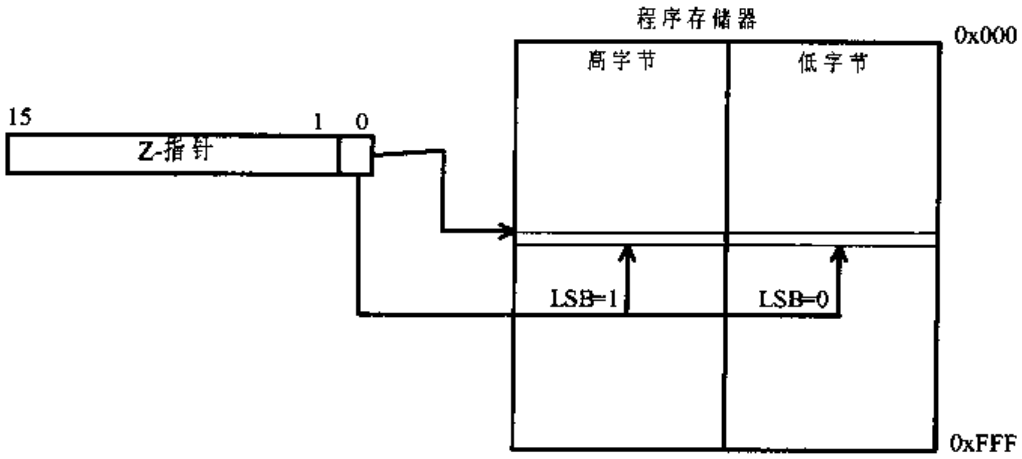


图 4.4 LPM 指令使用

要注意：AVR 程序存储器的指令总线为 16 位，即指令寄存器访问程序存储器时是以“字”(两个字节)为单位进行的，一个程序地址对应两个字节，而数据总线是 8 位的，一次读写是以一个字节为单位进行的，即用 Z 寄存器访问时，一次只能读写一个字节的数据。因此在使用 LPM 指令时，Z 指针(16 位)的最低位(LSB)为 0 时选择地址总线的低字节，为 1 时选择地址总线的高字节。

使用汇编时要注意地址总线与数据总线之间的换算关系，但是使用 C 语言后，这些换算关系由 C 编译器代劳了，编程者甚至不用去关心 16 位的指令总线与 8 位的数据总线之间的区别。

4.3.2 程序存储器和常量数据的 C 语言源程序及剖析

const 限定词用法十分灵活，适用于常量、表格数据以及指针，如下：

[例 4.7]

```
1   #include "io8515v.h"
2   const int itable[]={1,2,3};
3   const char ctable[]={4,5,6};
4   const char *ptr1;
5   char *const ptr2;
6   const char *const ptr3;
7   void main(void)
       {
8       int tmpInt;
9       char tmpChr;
10      tmpInt=itable[1];
11      tmpChr=ctable[2];
```

```
12      ptr1=ctable;
13      tmpChr=* ptr1++;
14      tmpChr=*ptr1;
}
```

第 2 行 const int itable[] = {1,2,3}，定义 itable 是分配在程序空间中的 3 个整型表格常量，占据三个字的位置。如果编译器定义 itable 从 0x001A 开始，则他们在程序空间存放形式如图 4.5 所示。

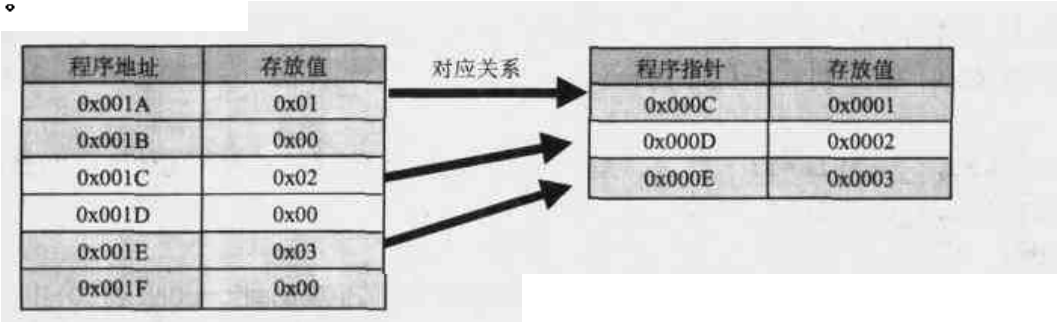


图 4.5　程序地址与程序指针的关系

“程序地址”指向程序空间中的“字节”(LPM 指令就使用程序地址，每次读取程序空间的 8 位数据)，“程序指针”指向程序空间的“字”。两者的换算关系为：

程序地址=程序指针*2

在 ICCAVR C 中不必担心这个问题，编译器已经自动地作了相应的变换，如第 10 行 tmpInt = itable[1]读取整型表格量 itable 的第 2 个值，如果 tmpInt 分配了 R23、R22 两个通用寄存器，对应的汇编程序为：

```
ST R30,-Y;                  保存当前 Z 指针
ST R31,-Y
ST R0,-Y;                   保存当前 R0
LDI R30,<_itable+2;         加载 itable[1]在程序空间的地址到 Z(指向 8 位字节)
LDI R31,>_itable+2
LPM;                        读取整型量的低字节
MOV R22,R0;                 将读取到 R0 的值复制到 tmpInt 的低字节中
ADIW R30,1;                 程序地址加 1
LPM;                        读取整型量的高字节
MOV R23,R0;                 将读取到 R0 的值复制到 tmpInt 的高字节中
LD R0,Y+;                   恢复原来 R0 值
LD R31,Y+;                  恢复原来 Z 指针值
LD R30,Y+
```

第 3 行 const char ctable[]={4,5,6}定义 ctable 是分配在程序空间中的 3 个字节型表格常量，占据两个字的位置。如果编译器定义 itable 从 0x0020 开始，则他们在程序空间的存放

形式如图 4.6 所示。

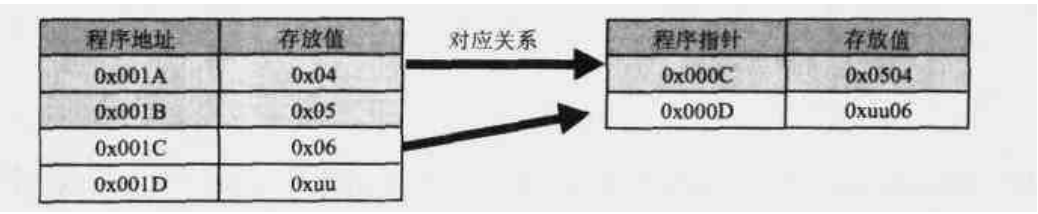


图 4.6 程序地址与程序指针的对应关系

第 11 行 tmpChr =ctable[2]读取字节型表格数据 ctable 的第 3 个值，tmpChr 被分配到通用寄存器 R20，相应的汇编程序为：

```
LDI R30,<_ctable+1;          加载 ctable[2]在程序空间的地址到 Z(指向 8 位字节)
LDI R31,>_ctable+1
LPM;                         读取字节量
MOV R20,R0;                  将读取到 R0 的值复制到 tmpChr(R20)中
```

第 4 行 const char *ptr1，表示 ptr1 是一个存放在数据空间中而指向程序空间数据的指针；第 5 行 char *const ptr2，表示 ptr2 是一个存放在程序空间中而指向数据空间数据的指针；第 6 行 const char *const ptr3，表示 ptr3 是一个存放在程序空间中并且指向程序空间数据的指针。比较常用的是像 itable、ctable 一类存放在程序空间中的表格数据和像 ptr1 一类存放在数据空间但指向程序空间数据的指针。

第 12~14 行是指针操作，ptr1=ctable 初始化指针，tmpChr =*ptr1++读取 ptr1 指向地址的值(ctable[0])并使 ptr1 值加 1，tmpChr =*ptr1 读取 ptr1 指向地址的值(ctable[1])。相应的汇编程序为：

```
LDI R24,<_ctable
LDI R25,>_ctable
STS _ptr1+1,R25;             初始化指针 ptr1
STS _ptr1, R24
MOV R2, R24
MOVE R3, R25
ADIW R24,1;                  指针值加 1
STS _ptr1+1,R25;             更新指针值
STS _ptr1, R24
MOV R30, R2;                 加载 Z 指针(Z 指针值还是加 1 前的 ptr1)
MOV R31, R3
LPM;                         读取字节数据
MOV R20, R0;                 将读取到 R0 的值 COPY 到 tmpChr(R20)中
MOV  R30, R 24;              加载 Z 指针
MOV  R31, R25
```

```
LPM;                                    读取字节数据
MOV  R20, R0;                           将读取到 R0 的值 COPY 到 tmpChr(R20)中
```

4.3.3　利用程序空间常量表实现 16 位快速 CRC

CRC(循环冗余校验)在数据传输中具有广泛的应用，本节将介绍一种面向字节的 16 位 CRC(下面简称 CRC16)的快速算法，与面向位的算法相比，能提高效率并节省 CPU 的时间。校验程序利用放在程序空间的常量数据进行 CRC 计算，如果使用扩展 32K 或 64K 的外部数据存储器(SRAM)，而为了节省程序运行时间，也可以把查找表放在扩展的外部数据空间。

CRC16 的产生多项式为：

$$G(x)=x^{16}+x^{15}+x^2+1$$

我们把 CRC16 计算需要的查找表放在程序空间中，CRC16 常数为 0xA001，保存在程序空间中包含 256 个整型数的数组 mtab 为：

```
const unsigned int mtab[256]=
   {
   0000 , C1C0 , 81C1 , 4001 , 01C3 , C003 , 8002 , 41C2 ,
   01C6 , C006 , 8007 , 41C7 , 0005 , C1C5 , 81C4 , 4004,
   01CC , C00C , 800D , 41CD , 000F , C1CF , 81CE , 400E ,
   000A , C1CA , 81CB , 400B , 01C9 , C009 , 8008 , 41C8,
   01D8 , C018 , 8019 , 41D9 , 001B , C1DB , 81DA , 401A ,
   001E , C1DE , 81DF , 401F , 01DD , C01D , 801C , 41DC,
   0014 , C1D4 , 81D5 , 4015 , 01D7 , C017 , 8016 , 41D6 ,
   01D2 , C012 , 8013 , 41D3 , 0011 , C1D1 , 81D0 , 4010,
   01F0 , C030 , 8031 , 41F1 , 0033 , C1F3 , 81F2 , 4032 ,
   0036 , C1F6 , 81F7 , 4037 , 01F5 , C035 , 8034 , 41F4,
   003C , C1FC , 81FD , 403D , 01FF , C03F , 803E , 41FE ,
   01FA , C03A , 803B , 41FB , 0039 , C1F9 , 81F8 , 4038,
   0028 , C1E8 , 81E9 , 4029 , 01EB , C02B , 802A , 41EA ,
   01EE , C02E , 802F , 41EF , 002D , C1ED , 81EC , 402C,
   01E4 , C024 , 8025 , 41E5 , 0027 , C1E7 , 81E6 , 4026 ,
   0022 , C1E2 , 81E3 , 4023 , 01E1 , C021 , 8020 , 41E0,
   01A0 , C060 , 8061 , 41A1 , 0063 , C1A3 , 81A2 , 4062 ,
   0066 , C1A6 , 81A7 , 4067 , 01A5 , C065 , 8064 , 41A4,
   006C , C1AC , 81AD , 406D , 01AF , C06F , 806E , 41AE ,
   01AA , C06A , 806B , 41AB , 0069 , C1A9 , 81A8 , 4068,
   0078 , C1B8 , 81B9 , 4079 , 01BB , C07B , 807A , 41BA ,
   01BE , C07E , 807F , 41BF , 007D , C1BD , 81BC , 407C,
   01B4 , C074 , 8075 , 41B5 , 0077 , C1B7 , 81B6 , 4076 ,
```

```
0072 , C1B2 , 81B3 , 4073 , 01B1 , C071 , 8070 , 41B0,
0050 , C190 , 8191 , 4051 , 0193 , C053 , 8052 , 4192 ,
0196 , C056 , 8057 , 4197 , 0055 , C195 , 8194 , 4054,
019C , C05C , 805D , 419D , 005F , C19F , 819E , 405E ,
005A , C19A , 819B , 405B , 0199 , C059 , 8058 , 4198,
0188 , C048 , 8049 , 4189 , 004B , C18B , 818A , 404A ,
004E , C18E , 818F , 404F , 018D , C04D , 804C , 418C,
0044 , C184 , 8185 , 4045 , 0187 , C047 , 8046 , 4186 ,
0182 , C042 , 8043 , 4183 , 0041 , C181 , 8180 , 4040
};
```

CRC16 算法遵循下面的步骤：

(1) 初始化 CRC 寄存器 FCS(一般设为 FCS = 0xFFFF)；

(2) 将下一个 8 位数据与 FCS 高 8 位数据异或的结果作为查找表的 8 位指针；

(3) 根据修改后的指针取得查找表中的值；

(4) 将查找表中高 8 位数据与 FCS 低 8 位数据异或后的值作为 FCS 值。重复步骤 2~4 直到计算到最后一个字节为止；

(5) 将 FCS 与 0xFFFF 异或值作为 CRC 计算结果返回。

下面的函数 fcs_calc 计算 CRC16，有三个输入参数：

const unsigned int *mtab：指向程序空间中的查找表；

unsigned char *buff：指向需要进行 CRC 校验的数组；

unsigned int len：需要进行 CRC 校验的数组长度。

函数返回 16 位的 CRC16 结果。

[例 4.8]

```
unsigned int fcs_calc(const unsigned int *mtab , unsigned char *buff , unsigned int len)
    {
    unsigned int fcs;
    unsigned int q;                 //临时寄存器
    fcs=0xffff;                     //初始化 CRC 寄存器 FCS
    while(len--)
        {
        q=*(mtab+(*buff++^(fcs>>8)));
        fcs=((q&0xff00)^(fcs<<8))|(q&0x00ff);
        }
    return (fcs^0xffff);            //将 FCS 与 0xFFFF 异或值作为 CRC 计算结果返回
    }
```

4.4　C 任务(Tasks)

当调用一个函数时，ICCAVR 编译器都需要生成代码来保存和恢复可变寄存器(volatile registers)和 SREG 寄存器。在一些特别情况下，这些操作可能是多余的。例如，当我们使用 RTOS(实时操作系统)，RTOS 管理着寄存器的保存和恢复，并作为任务切换处理的一部分，编译器如果再插入这些代码就没有必要了，另外还有一种情况就是，在调用某些函数时，我们可以肯定该函数并不会使用任何的寄存器，或者该函数在调用前后，可变寄存器(volatile registers)的内容保持不变，因此在调用该函数时插入保护和恢复可变寄存器(volatile registers)的指令就是多余的。如在[例 4.5]中用到的 delay 函数，在该函数中并没有使用任何寄存器，因此不需要插入保存和恢复可变寄存器的代码。

说明：不同的编译器对可变寄存器(volatile registers)的定义是不同的，ICCAVR 中与可变寄存器的有关内容，请参考“2.12.2 在线汇编中函数调用规则”一节。被调用的函数还必须保证保护寄存器(Preserved Registers)中的内容在调用前后不发生改变，如果需要使用这些保护寄存器，被调函数还必须自行插入保护和恢复所用到的保护寄存器的代码。

ICCAVR 使用 #pragma ctack 伪指令来说明某一函数为 C 任务，例如：

```
#pragma ctask fun1,fun2
void fun1(void)
    {
    //应用程序
    }
void fun2(void)
    {
    //应用程序
    }
void fun3(void)
    {
    //应用程序
}
```

在上面定义了三个函数，其中 fun1 和 fun2 为 C 任务函数，一个普通的函数 fun3。

C 任务通常都是与实时操作系统一起使用的，ICCAVR6.26C 中已附带了 UCOSII 实时操作系统的源代码及其部分应用实例，有关 UCOSII 的内容将放到编者的《AVR 单片机 C 语言开发应用与提高》一书中讲述。

4.5 I/O 寄存器

4.5.1 I/O 寄存器操作的特点

AT90S8515 拥有 64 个 I/O 寄存器(如图 4.7 所示)，通过这些寄存器可以：

1. 了解 CPU 及其外设的运行状态(查看不同的状态标志)；

2. 控制 CPU 及其外设的动作和行为。

在 AVR 单片机中，I/O 寄存器均有两种编址方式，以 AT90S8515 为例，其编址如图 4.7 所示。

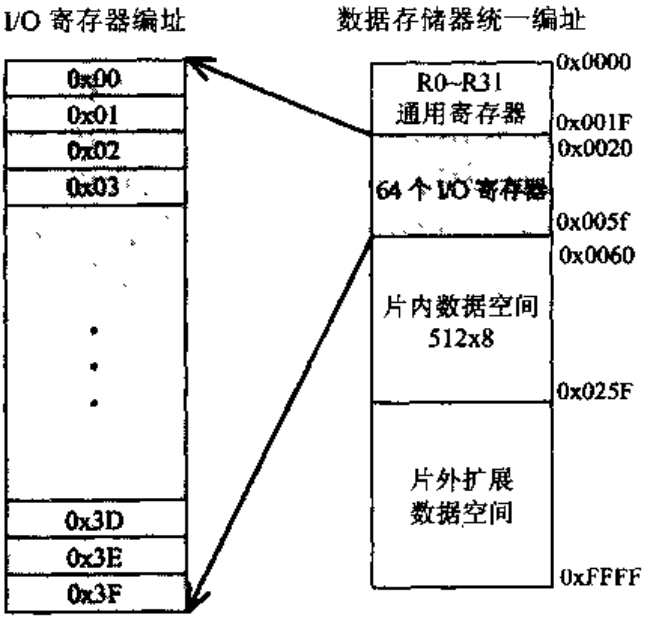


图 4.7 AT90S8515 的 I/O 寄存器

这 64 个 I/O 寄存器既有自己的独立编址 0x00~0x3F，也可以看作是在数据空间的映射 0x20~0x5F。对于寄存器的两种不同的编址方式，分别使用不同的指令来访问：

1. 访问 0x00~0x3F 的 I/O 寄存器空间，使用 IN、OUT 指令；

2. 访问 0x20~0x5F 的数据存储器空间，汇编中使用 LD、ST 和 LDS、STS 指令。

4.5.2 I/O 寄存器的 C 语言源程序及剖析

在 ICCAVR 中访问 0x00~0x3F 的寄存器空间，可以使用内汇编和预处理宏。

一个数据内存地址可以通过加指针类型符号直接访问。例如，SREG 寄存器的地址是

0x5F，对它的访问操作：

```
unsigned char c;
c=*(volatile unsigned char *)0x5F;          //读取 SREG 寄存器
*(volatile unsigned char *)0x5F|=0x80;      //打开全局中断允许位
```

提示：数据内存地址 0x00~0x31 指向 CPU 通用寄存器 R0~R31，不能随意修改它们的地址(或者输入错误的地址)，否则可能会造成致命的错误。

ICCAVR C 在 io*v.h 文件中用 define 伪指令给大部分的寄存器定义了指针形式符号，如定义 SREG：

```
#define SREG *(volatile unsigned char *)0x5F
```

上面对 SREG 的操作也可以改为：

```
unsigned char c;
c=SREG;                  //读取 SREG 寄存器
SREG|=0x80;              //打开全局中断允许位
```

说明：

1．在 ICCAVR 中对 I/O 寄存器进行操作，编译器自动生成用于访问 I/O 寄存器的专用指令，如 IN、OUT、BSET、BCLR 等，而且已经将地址 0x5F 变换为 0x3F。

2．#define 语句也可以定义绝对内存空间，用以指向一个固定地址的字节，如：

```
#define LED_OUT *(volatile unsigned char *)0x80AA
```

具体内容我们将在下一节讨论。

4.5.3　实现 1×8 键盘和 LED 显示

下面以一个简单的 1×8 键盘和 LED 显示来说明如何访问 I/O 寄存器。在这个例子中使用 AT90S8515、4MHz 晶振、PORTA 口接 8 个按键、PORTC 口接 8 个 LED，如图 4.8 所示，这个例子很简单，一个按键对应一个 LED，按一个键，对应的 LED 就会亮，而且支持多键组合。

说明：

(1) 下面的程序仅仅演示 I/O 口简单的输入和输出功能。

(2) 在图 4.8 中并没有画出 AT90S8515 的电源及外接的晶振。

(3) 要注意 LED 的连接方式，对于使用 CMOS 工艺的 51 系列的单片机，如 ATMEL 的 89C51，也不能按图 4.8 来接 LED，因为 89C51 吸入的电流大，输出的电流小，不足以点亮 LED，因此对于 51 系列单片机，通常 LED 的接法相反。即使一些近几年推出 51 内核改进的 MCU，有足够的吸入/输出电流，还是不能按图 4.8 所示的方法连接 LED，因为所

有的 51 内核的单片机，在复位时各端口均输出高电平，在开机一瞬间，LED 会点亮(MCU 复位时)。

(4) 图 4.8 中，仅仅是为了说明 AVR 单片机芯片的特性(与 51 系列单片机端口的差异)，所以 LED 的阳极接 AVR 单片机的端口，在实际使用时，如果有大量的 LED 都这样接，AVR 单片机功耗增加且不稳定，因此通常应将 LED 的阴极接单片机的端口，即端口输出低电平，LED 点亮，端口输出高电平，LED 熄灭。

AVR 系列单片机所有 I/O 端口具有真正的“读-修改-写”功能。以 PORTA 口为例，它是 8 位双向 I/O 端口，有三个 I/O 地址：数据锁存器 PORTA、数据方向寄存器 DDRA 和输入引脚地址 PINA，其中 PORTA 和 DDRA 可以读写，PINA 只可读。PORTA 口所有的管脚都有可以独立选择的上拉电阻，每个引脚缓冲器可以吸收 20mA 的电流，能够直接驱动 LED 显示，若内部拉高被触发，这些引脚可以成为电流源，对外提供弱的电流。

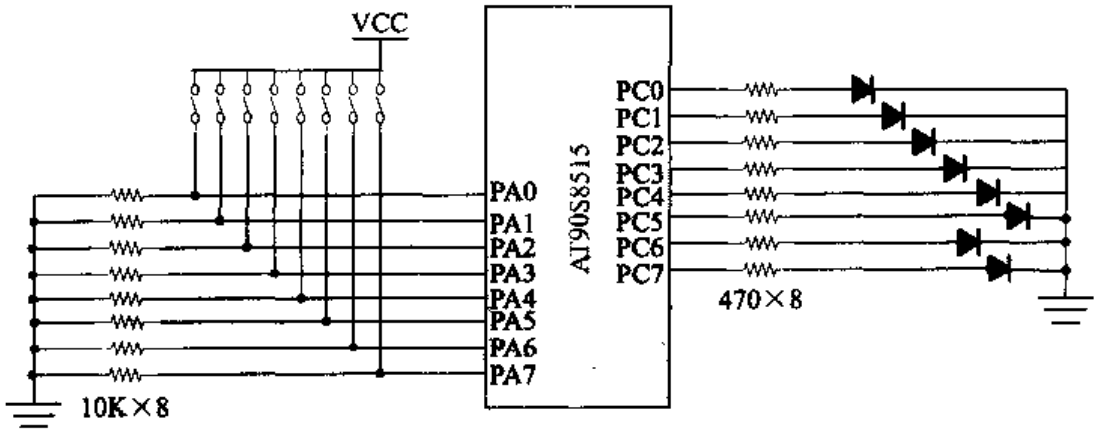


图 4.8 1×8 键盘和 LED 显示

PORTA 端口作为通用数字 I/O 口，8 个管脚具有相同的功能。DDRA 中的 DDRAn 选择引脚的方向，DDRAn 为“1”，相应的引脚为输出脚；DDRAn 为“0”，相应的引脚为输入脚；复位期间，PORTA 口为输入无上拉的高阻态，PORTA 口电平由外电路决定，这在强调复位电平的电路中特别有用。

51 系列单片机的各 I/O 口在复位时均为高电平，在某些场合中应用不方便，并且 51 系列单片机为准双向口，在作输入时，必须先向相应的端口锁存器中写入“1”，才能够读入引脚电平，而在 AVR 中就不必如此操作，直接读 PINn 口地址就可以了，以 PA 口为例，PA 口引脚 DDAn 和 PORTAn 的相互作用见表 4.5。

表 4.5 AVR 单片机 PA 口的配置

DDAn	PORTAn	I/O	上 拉	注 释
0	0	输入	N	高阻态
0	1	输入	Y	外部拉低时会输出电流
1	0	输出	N	推挽“0”输出
1	1	输出	N	推挽“1”输出

注：表中 n 为 0~7，引脚号

提示：PINA 不是一个寄存器，该 I/O 地址用来访问 PA 口的物理值。当读取 PORTA 时，读到的是 A 口锁存的数据；而读取 PINA 时，读到的是引脚上的逻辑电平值：高电平=1，低电平=0，这两者是不同的，要注意区别。

[例 4.9]

我们先要初始化使用到的端口方向：

```
void port_init(void)
    {
    PORTA=0x00;
    DDRA=0x00;                    //端口 A 初始化为输入
    PORTB=0xFF;
    DDRB=0x00;
    PORTC=0x00;                   //关闭所有的 LED
    DDRC=0xFF;                    //端口 C 初始化为输出
    PORTD=0xFF;
    DDRD=0x00;
    }
```

延迟函数：

```
void delay_1ms(void)
    {
    unsigned int i;
    for (i=1;i<570;i++)
         ;
    }
void delay(unsigned int n)
    {
    unsigned int i=0;
    while (i<n)
        {
        delay_1ms();
        i++;
        }
    }
```

程序流程如图 4.9 所示，系统初始化后，进入一个死循环，20 毫秒的防抖延时后，再判断两次读数是否相同，如果相同，则说明是同一按键按下，然后点亮相应的 LED。

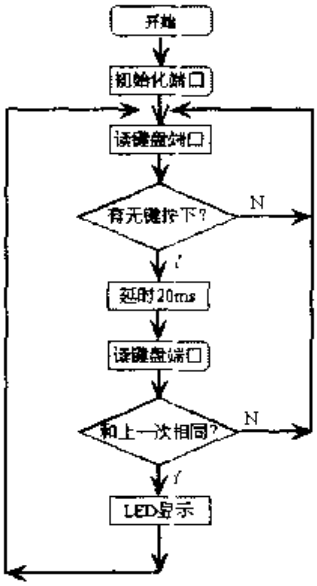


图 4.9 程序流程图

为增加程序稳定性，我们还可以使用 AT90S8515 内置的 watchdog。在+5V 系统下，采用 1.9 秒的 watchdog，初始化如下：

```
void watchdog_init(void)
    {
    WDR();                          //防止设置时 WATCHDOG 超时
    WDTCR=0x0F;                   //使能 WATCHDOG，采用 1MHz 的 2048 分频
    }
```

下面给出完整的程序清单，在 ICCAVR 6.26C 环境中编译通过：

```
#include <io8515v.h>
#include <macros.h>
void main(void)
    {
    unsigned char oldP , newP;        //定义临时按键值
    port_init();                      //初始化端口
    watchdog_init();                  //初始化 watchdog
    newP=0;                           //初始化临时变量
    oldP=0;
    for(;;)                           //进入死循环，等待按键发生
         {
         oldP=PINA;
         if( oldP^newP)               //有按键按下，oldP != newP
              {
              delay(20);              //延时 20ms，消除按键抖动
```

```
            newP = PINA;
            if(oldP & newP)          //延时前后的按键相同
                  {
                  newP &= oldP;      //取得新的按键值
                  PORTC^= newP;      //更新 led 显示
                  }
      }
      WDR( );                        //“喂狗”
      }
}
```

4.6　数据存储器的绝对寻址

4.6.1　数据存储器绝对寻址的操作特点

我们的应用中可能用到外部的 I/O 设备，如 LCD 接口或串口扩展等，对这些 I/O 设备内部寄存器的访问，可以映射为内存空间的绝对 I/O 地址。因为 AVR 单片机有 16 根地址线，可线性定位 64K 范围内的地址。

AVR 单片机有部分型号可以扩展 64K 的外部数据空间，如图 4.10 所示(以 AT90S8515 为例)，外部数据空间的低 512 字节与 AVR 单片机内的 512 字节 RAM 地址是重合的。如果单片机设定使用外部 SRAM，地址 0~511 所指的是内部的 512 字节，当访问大于 511 的地址，AVR 自动访问外部扩展的 SRAM 空间。无论访问内部或外部 RAM，都是用 LD、ST、STS、LDS 命令，只有访问外部数据存储器时 AVR 才会产生外部 RD 和 WR 信号。

提示：对 AT90S515 芯片而言，外部程序空间中的 0~511 地址不能使用。AVR 数据存储空间如图 4.10 所示。

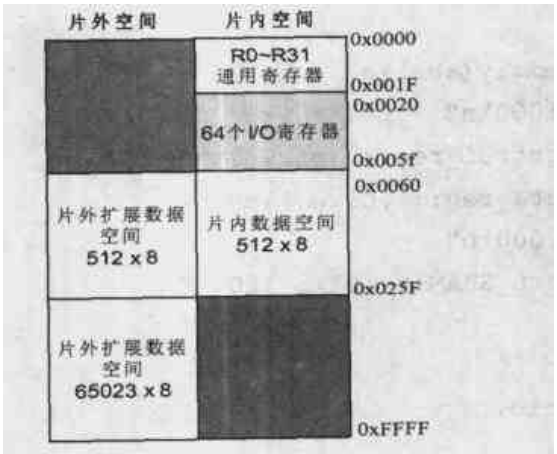


图 4.10　AVR 数据存器空间

4.6.2 绝对寻址数据存储器 C 语言源程序及剖析

假设有一个双字节的 LCD 控制寄存器定位在 0x1000 地址，一个双字节的 LCD 数据寄存器定位在 0x1002 地址，并且有一个 100 字节的双口 SRAM 定位在 0x2000 的地址。使用在线汇编定义：

```
asm(".area memory(abs)\n"          //在线汇编定义需要的绝对地址
".org 0x1000\n"
" _LCD_control_reg:: .blkw 1\n"
" _LCD_data_reg:: .blkw 1\n"
".org 0x2000\n"
"_dual_port_SRAM:: .blkb 100\n");
```

在上面的定义中，“.area memory(abs)”伪指令表示使用数据空间的绝对地址；“.org 0x1000”伪指令定位绝对地址，并且以此为起始地址分配变量。LCD_control_reg 定义在 0x1000，_LCD_data_reg 定义在 0x1002。“.blkw”伪指令定义双字节空间，“.blkb”伪指令定义单字节空间。“_dual_port_SRAM:: .blkb 100”定义 0x2000~0x2063 的 100 字节的数据空间。

如果在 ICCAVR 的 C 中要决定地址空间，需要声明他们为外部变量(汇编中定义的变量)，如果 C 中没有声明，ICCAVR 将会报错(未定义变量)。这是因为在汇编中定义的这些绝对地址空间的地址变量，即使在同一源文件中，C 编译器也不知道汇编中定义的变量。

```
extern unsigned int LCD_control_reg;
extern unsigned int LCD_data_reg;
extern unsigned char dual_port_SRAM[100];
void mapping_init(void)
   {
   asm(
      ".area memory(abs)\n"
      ".org 0x1000\n"
      " _LCD_control_reg:: .blkw 1\n"
      " _LCD_data_reg:: .blkw 1\n"
      ".org 0x2000\n"
      "_dual_port_SRAM:: .blkb 100\n"
      );
   }
void init_LCD(void)
   {
   union Datas
```

```
    {
    unsigned int tmINT;
    unsigned char tmChar[2];
     }tmData;
LCD_control_reg=0x8000;
                                    //用户程序
tmData.tmINT=LCD_data_reg;
dual_port_SRAM[0]=tmData.tmChar[0];
dual_port_SRAM[1]=tmData.tmChar[1];
tmData.tmChar[0]=0x80;
tmData.tmChar[1]=0x00;
                                    //实际上是 LCD_control_reg=0x0080
LCD_control_reg = tmData.tmINT;
}
```

内部的联合体变量使用 y 指针，在上例中生成的汇编代码为：

```
.area memory(abs)              ;绝对地址空间变量定义
.org 0x1000
 _LCD_control_reg:: .blkw 1
.org 0x1002
 _LCD_data_reg:: .blkw 1
.org 0x2000
 _dual_port_SRAM:: .blkb 100
.text;                              程序空间开始
_init_LCD:
   sbiw R28,2;                      设置 y 指针
   ldi R24,32768
   ldi R25,128
   sts _LCD_control_reg+1,R25
   sts _LCD_control_reg,R24
   lds R2,_LCD_data_reg;            读取 LCD_data_reg 低字节
   lds R3,_LCD_data_reg+1;          读取 LCD_data_reg 高字节
   std y+0,R2
   std y+1,R3
   ldd R2,y+0
;LCD_data_reg 低字节保存在 dual_port_SRAM[0]
   sts _dual_port_SRAM,R2
   ldd R2,y+1
;LCD_data_reg 高字节保存在 dual_port_SRAM[1]
   sts _dual_port_SRAM+1,R2
```

```
ldi R24,128;                    tmData.tmChar[0] = 0x80
std y+0,R24
clr R2;                         tmData.tmChar[1] = 0x00
std y+1,R2
ldd R2,y+0
ldd R3,y+1;                     使用 tmData.tmINT 初始化 LCD_control_reg = 0x0080
sts _LCD_control_reg+1,R3
sts _LCD_control_reg,R2
adiw R28,2;                     设置 y 指针
ret
```

在 ICCAVR 中，使用强制指针变量的方式定义 I/O 寄存器，如在 io8515v.h 中定义的串口数据寄存器 UDR：

```
#define UDR (*(volatile unsigned char *)0x2C)
```

0x2C 是串口数据寄存器 UDR 在 I/O 空间的绝对地址。
因此，定义绝对地址空间的字节变量也可以使用类似的方式：

```
#defin LCD_DATA (*(volatile unsigned char *)0x8000)
#define LCD_ISR (*(volatile unsigned char *)0xA000)
```

在上述定义中，LCD_DATA 定位在数据存储空间的 0x8000，LCD_ISR 定位在数据存储空间的 0xA000。在 C 语言中，就可以很方便的使用它们：

```
unsinged char tmISR;
LCD_DATA=0x80;
tmISR=LCD_ISR;
```

ICCAVR 编译器将上面的两句赋值语句编译出的汇编语句如下，其中临时变量 tmISR 分配到寄存器 R20：

```
ldi R24,128;                    R24=0x80
sts 32768,R24;
lds R20,40960;                  tmISR=LCD_ISR
```

同样，定义绝对地址空间的字(两个字节)也可以如下定义：

```
#defin LCD_DATA (*(volatile unsigned int *)0x8000)
#define LCD_ISR (*(volatile unsigned int *)0xA000)
```

使用强制指针变量的方式可以不必使用在线汇编，提高了程序的通用性和移植性，而且生成的代码更精炼。

4.6.3　使用 ST16C550 扩展串口

下面我们以使用 ST16C550 扩展串口芯片来说明如何使用数据空间绝对地址定位。

4.6.3.1　ST16C550 简介

ST16C550 是全双工串口收发器，用于单片机系统通过 8 位数据总线扩展串口。其原理如图 4.11 所示。

8 位数据总线用于读写数据，并且与 MCU 的 8 位数据总线相连，CS#是片选信号，低电平有效，RD# 和 WR#是读和写信号，都是低电平有效，右边的 8 个信号是可与 modem 连接的串口信号。通常如果不是直接与 modem 连接，一般只使用其中的 RX 和 TX 作为串口数据的收发信号。

提示：ST16C550 右边的 8 个信号都是 TTL 电平接口，如果要和 modem 等 RS232C 接口的设备连接，需要增加一块电平转换芯片(如 MAX202 等)。

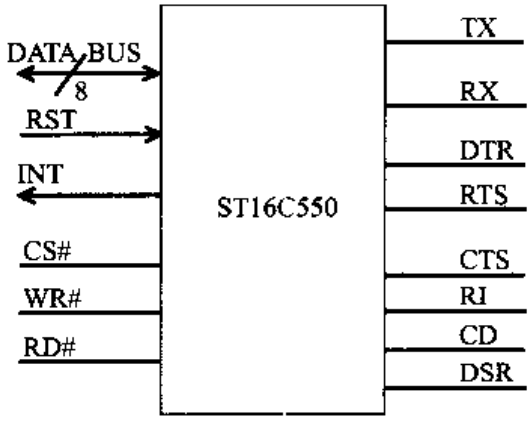


图 4.11　ST16C550 串口收发器

4.6.3.2　ST16C550 内部寄存器绝对地址空间的定义

ST16C550 内部有 7 个寄存器：存储接收/发送的数据寄存器 RHR/THR、中断控制寄存器 IER、状态寄存器 ISR、波特率寄存器 DLL 和 DLM 等。ST16C550 与 AT90S8515 的连接如图 4.12 所示，第 16 位地址(A15)反相后作为 ST16C550 的片选信号，片内寄存器选择地址 A2~A0 由 AVR 的 A14~A12 生成。ST16C550 的 TX 和 RX 通过电平转化(MAX232)后直接与 DTE 设备进行通信。

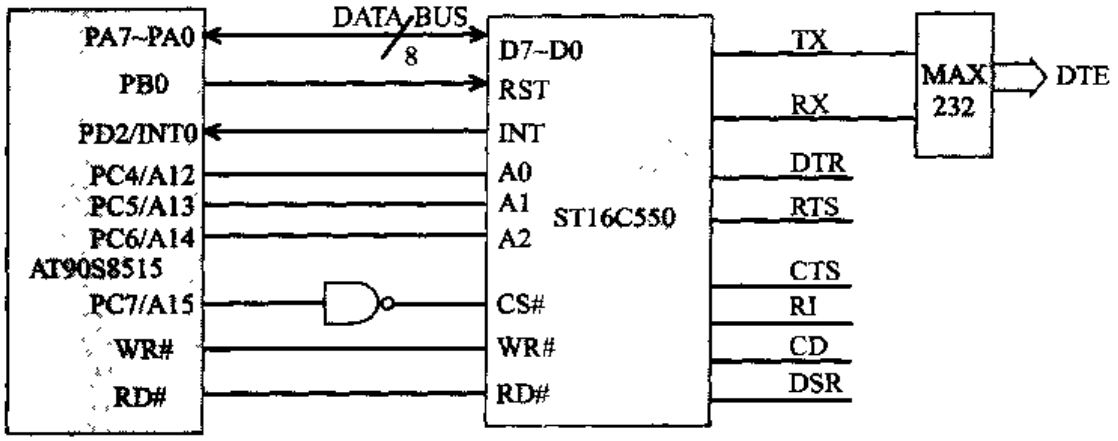


图 4.12 使用 ST16C550 扩展串口

这样，ST16C550 的几个内部寄存器可以映射为 AT90S8515 的绝对数据地址空间：

```
#define ST_RHR (*(volatile unsigned char *)0x8000)      //接收数据寄存器
#define ST_THR (*(volatile unsigned char *)0x8000)      //发送数据寄存器
#define ST_IER (*(volatile unsigned char *)0x9000)      //中断控制寄存器
#define ST_ISR (*(volatile unsigned char *)0xA000)      //中断状态寄存器
#define ST_FCR (*(volatile unsigned char *)0xA000)      //FIFO 控制寄存器
#define ST_LCR (*(volatile unsigned char *)0xB000)      //Line 控制寄存器
#define ST_MCR (*(volatile unsigned char *)0xC000)      //Modem 控制寄存器
#define ST_LSR (*(volatile unsigned char *)0xD000)      //Line 状态寄存器
#define ST_MSR (*(volatile unsigned char *)0xE000)      //Modem 状态寄存器
#define ST_SPR (*(volatile unsigned char *)0xF000)      //用户数据寄存器
#define ST_DLL (*(volatile unsigned char *)0x8000)      //波特率发生器低字节
#define ST_DLM (*(volatile unsigned char *)0x9000)      //波特率发生器高字节
```

波特率产生器计算公式：

BAND=晶振频率/(波特率×16)

使用 1.8432MHz 晶振，可以产生 50~115200bps 的任意标准的波特率，如表 4.6 所示。

表 4.6 ST16C550 波特率计算

波 特 率	分频 (10 进制)	分频 (16 进制)	DLM (16 进制)	DLL (16 进制)
50	2304	900	09	00
75	1536	600	06	00
150	768	300	03	00
300	384	180	01	80
600	192	C0	00	C0
1200	96	60	00	60

续表

波　特　率	分频 (10 进制)	分频 (16 进制)	DLM (16 进制)	DLL (16 进制)
2400	48	30	00	30
4800	24	18	00	18
7200	16	10	00	10
9600	12	0C	00	0C
19200	6	06	00	06
38400	3	03	00	03
57600	2	02	00	02
115200	1	01	00	01

4.6.3.3　ST16C550 使用

1. ST16C554 复位

```
void delay(unsigned char d)
    {
    while(d!=0)
        {
        asm("nop");
        d--;
        asm("nop");
        }
    return;
    }
void reset_st550(void)
    {
    PORTB&=~0x01;                    //RST=0
    delay(100);                      //延时
    PORTB|=0x01;                     //RST=1
    delay(80);                       //延时
    PORTB&=~0x01;                    //RST=0
    }
```

上面的函数调用了一个延时函数 delay，调用它将使 ST16C550 的复位电平保持一段时间。

2. ST16C550 初始化

```
void init_st550(void)
    {
```

```
    reset_st550();                    //先复位
    ST_LCR=0x80;                      //配置波特率寄存器
    ST_DLL=0x0C;                      //使用 1.8432MHz 的晶振，9600bps
    ST_DLM=0x00;
    ST_LCR=0x03;                      //8 位数据，无校验，1 位停止位
    ST_FCR=0x00;                      //没有使用 FIFO
    ST_MCR=0x02;
    ST_IER=0x01;                      //接收中断使能
    ST_SPR=0xAA;                      //用户状态寄存器
    }
```

3. 查询方式收发数据

```
unsigned char Q_recvChar(void)        //查询方式接收一个字节
    {
    while(!(ST_ISR&0x04));            //等待接收完成标志位置位
    return ST_RHR;                    //返回接收到的字节
    }
void Q_sendChar(unsigned char d)      //查询方式发送一个字节
    {
    ST_THR = d;
    while(!(ST_LSR&0x20));            //等待发送完成标志位置位
    }
```

4. 中断方式收发数据

ST16C550 的 INT 中断信号接到 AT90S8515 的 PD0 引脚(INT0)，设置为高电平中断触发。中断触发时，有可能是接收到一个字节数据，也有可能刚发送完成一个字节。假定接收到的字节放在 tmRecv 中，需要发送的字节在 tmSend 中。

```
#pragma interrupt_handler int0_isr:2
void int0_isr(void)
    {
    if(ST_ISR&0x04)                   //判断接收完成标志位置位
    tmRecv = ST_RHR;                  //返回接收到的字节
    if(ST_LSR&0x20)                   //判断发送完成标志位置位
    ST_THR = tmSend;
    }
```

下面给出使用 ST16C550 查询方式收发数据的完整程序清单，在 ICCAVR 6.26C 环境中编译通过。

```
#include <io8515v.h>
#include <macros.h>
#define ST_RHR (*(volatile unsigned char *)0x8000)      //接收数据寄存器
#define ST_THR (*(volatile unsigned char *)0x8000)      //发送数据寄存器
#define ST_IER (*(volatile unsigned char *)0x9000)      //中断控制寄存器
#define ST_ISR (*(volatile unsigned char *)0xA000)      //中断状态寄存器
#define ST_FCR (*(volatile unsigned char *)0xA000)      //FIFO 控制寄存器
#define ST_LCR (*(volatile unsigned char *)0xB000)      //Line 控制寄存器
#define ST_MCR (*(volatile unsigned char *)0xC000)      //Modem 控制寄存器
#define ST_LSR (*(volatile unsigned char *)0xD000)      //Line 状态寄存器
#define ST_MSR (*(volatile unsigned char *)0xE000)      //Modem 状态寄存器
#define ST_SPR (*(volatile unsigned char *)0xF000)      //用户数据寄存器
#define ST_DLL (*(volatile unsigned char *)0x8000)      //波特率发生器低字节
#define ST_DLM (*(volatile unsigned char *)0x9000)      //波特率发生器高字节
void delay(unsigned char d)                  //延时函数
  {
  while(d!=0)
     {
     asm("nop");
     d--;
     asm("nop");
     }
  return;
  }
void reset_st550(void)                       //复位 ST16C550
  {
  PORTB&=~0x01;                              //RST=0
  delay(100);                                //延时
  PORTB|=0x01;                               //RST=1
  delay(80);                                 //延时
  PORTB&=~0x01;                              //RST=0
  }
void init_st550(void)                        //初始化 ST16C550
  {
  reset_st550();                             //先复位
  ST_LCR=0x80;                               //配置波特率寄存器
  ST_DLL=0x0C;                               //使用 1.8432MHz 的晶振，9600bps
  ST_DLM=0x00;
  ST_LCR=0x03;                               //8 位数据，无校验，1 位停止位
  ST_FCR=0x00;                               //没有使用 FIFO
  ST_MCR=0x02;
  ST_IER=0x01;                               //接收中断使能
```

```
    ST_SPR=0xAA;                          //用户状态寄存器
    }
unsigned char Q_recvChar(void)            //查询方式接收一个字节
    {
    while(!(ST_ISR&0x04));                //等待接收完成标志位置位
    return ST_RHR;                        //返回接收到的字节
    }
void Q_sendChar(unsigned char d)          //查询方式发送一个字节
    {
    ST_THR=d;
    while(!(ST_LSR&0x20));                //等待发送完成标志位置位
    }
void main(void)
    {
    unsigned char tchr;
    GIMSK=0x00;                           //关闭外部中断
    CLI();
    reset_st550();
    for(;;)
        {
        tchr = Q_recvChar();              //查询方式等待接收一个字符
        Q_sendChar(tchr);                 //查询方式把收到的字符发送出去
        }
    }
```

4.6.4 程序存储器的绝对定位

在 ICCAVR 中，可以使用下面的编译附注实现将指定的代码定位于程序存储器中的一个指定的位置。

```
#pragma abs_address:<address>
```

函数与全局数据不使用浮动定位，而是从<address>开始分配绝对地址。

```
#pragma end_abs_address
```

结束绝对定位，使目标程序使用正常的浮动定位。

例如，我们可以将 LED 字形表放置于 0x500 开始的程序存储器地址，可以设置如下：

```
#pragma abs_address:0x500          //从 0x500 处开始分配地址
const unsigned char tabel[]={0x3f,0x06,0x5b,0x4f,0x66,0x6d,0x7d,0x07,0x7f,0x6f,0x77,
                             0x7c,0x39,0x5e,0x79,0x71};
#pragma end_abs_address            //恢复正常的浮动定位
```

4.6.5　EEPROM 的绝对定位

ICCAVR 实际上并不提供在 EEPROM 中定位的指令，下面先看一个例子：

```
#pragma data:eeprom
int foo=0x1234;
char table[]={0,1,2,3,4,5};
#pragma data:data
```

在 ICCAVR 中编译后生成的汇编代码为(.s 文件中)：

```
.area eeprom(rom, con, rel)
_foo::
     .word 4660
     .dbfile 333.c
     .dbsym e foo _foo I
_table::
     .byte 0,1
     .byte 2,3
     .byte 4,5
     .dbsym e table _table A[6:6]c
     .area data(ram, con, rel)
```

生成的 INTEL HEX 文件的代码为(.eep 文件中)：

```
:0800010034120001020304 05A2
:00000001FF
```

生成的 EEPROM 文件(.eep)的读法与 INTEL HEX 格式文件的读法相同，其中 0 地址分配数据 00(不使用)，从 01 开始顺序分配数据。

假设在某一个工程应用中，EEPROM 中的空间有剩余，为了提高保存于 EEPROM 中数据的安全性，需要从 0x10 和 0x30 各存放一套数据，互为备份。可以采用如下方式：

```
#pragma data:eeprom
char a[]={1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
int table_one[]={0,1,2,3,4,5,6,7};
int b[]={0,1,2,3,4,5,6,7};
int table_two[]={0,1,2,3,4,5,6,7};
#pragma data:data
```

在上例中，数组 table_one 和数组 table_two 为需保存在 EEPROM 中的两套数据，而数组 a 和数组 b 为无效的数据，仅作为填充不需使用的 EEPROM 空间中。

编译得到的结果为：

```
:0E0001000102030405060708090A0B0C0D0E88
:0D000F000F000001000200030004000500C6
:0E001C000600070000000100020003000400BF
:0E002A000500060007000000010002000300B0
:0800380004000500060007 00AA
:00000001FF
```

实践也证明用这种方式在 EEPROM 中进行定位是可行的。

使用中要注意：

(1) 数组 a 和数组 b 不能为空，必须填入数字，否则该数组会被 ICCAVR 优化掉，使空间安排失效。

(2) 由于 AVR 单片机硬件设计上的缺陷，ATMEL 公司建议 EEPROM 中地址为 0 的存储空间尽量不要使用。支持 AVR 的所有编译器，如 ICCAVR、GCCAVR、IAR 和 CVAVR 等都不会使用地址为 0 的 EEPROM 空间，数据将从地址为 1 的位置开始放置，因此数组 a 只有 15 个元素。

4.7 中 断 操 作

AT90S8515 有 12 个中断源和一个复位中断。每个中断源在程序空间都有一个独立的中断向量地址，如表 4.7 所示。所有的中断事件都有独立的中断使能位，当使能位和全局中断使能都置“1”的情况下，中断才可以发生，相应的中断服务程序才会执行。

4.7.1 中断操作的特点(外部中断和定时/计数器中断)

不同型号的 AVR 单片机，其中断向量和中断源的数量是不同的，对于用 ICCAVR 编写的源程序，在不同型号的单片机之间移植时应注意。以 AT90S8515 为例，其中断向量和中断源如表 4.7 所示。

表 4.7 中断源和中断向量

向 量	程 序 地 址	中 断 源	中 断 定 义
1	0x000	RESET	复位中断
2	0x001	INT0	外部中断 0

续表

向　量	程 序 地 址	中　断　源	中 断 定 义
3	0x002	INT1	外部中断 1
4	0x003	TIMER1 CAPT	定时/计数器 1 捕获事件
5	0x004	TIMER1 COMPA	定时/计数器 1 比较匹配 A
6	0x005	TIMER1 COMPB	定时/计数器 1 比较匹配 B
7	0x006	TIMER1 OVF	定时/计数器 1 溢出
8	0x007	TIMER0 OVF	
9	0x008	SPI,STC	SPI 传输完成
10	0x009	UART,RX	UART 正确收到一个字节
11	0x00A	UART,UDRE	UART 发送寄存器空
12	0x00B	UART,TX	UART 成功发送一个字节
13	0x00C	ANA_COMP	模拟比较器事件

在表 4.7 中断向量中，处于低地址的中断具有高的优先级，所以在表中 RESET 具有最高的优先级，而模拟比较器的优先级最低。

上电复位、外部复位和 Watchdog 复位：在复位期间，所有的 I/O 寄存器被设置为初始值，程序从地址 0x000 开始执行。关于系统复位，将在下面详细介绍。

外部中断 INT0 和 INT1：由通用中断屏蔽寄存器 GIMSK 和一个通用中断标志寄存器 GIFR 进行控制，其中 GIMSK 寄存器的定义如表 4.8 所示，GIFR 寄存器的定义如表 4.9 所示。

表 4.8　通用中断屏蔽寄存器—GIMSK

	7	6	5	4	3	2	1	0
GIMSK(0x3B)	INT1	INT0	-	-	-	-	-	-
读/写	R/W	R/W	R	R	R	R	R	R
初始值	0	0	0	0	0	0	0	0

表 4.9　通用中断标志寄存器—GIFR

	7	6	5	4	3	2	1	0
GIFR(0x3A)	INTF1	INTF0	-	-	-	-	-	-
读/写	R/W	R/W	R	R	R	R	R	R
初始值	0	0	0	0	0	0	0	0

INT0/INT1 是外部中断使能位，INTF0/INTF1 使中断事件请求标志位。INT0/INT1 使能位和全局中断使能位 I 都为“1”时，外部中断使能。以 INT0 为例，外部中断 INT0 触发中断请求，INTF0 位置“1”。如果 INT0 位和 I 为 “1”，相应的服务中断程序会执行。中断服务程序执行时，INTF0 位和 I 位被硬件自动清“0”，这可以防止中断嵌套，中断服务程序完成，以 RETI 返回时，I 位被重新置“1”。复位后 INT0/INT1 位和 INTF0/INTF1 位都为“0”。

提示：INTF0/INTF1 的清“0”也可以通过直接写“1”来完成：

```
GIFR|=0x80;                    //将 INT1 清“0”
```

外部中断 INT0/INT1 的引脚可由编程设置为 3 种触发方式：上升沿触发、下降沿触发和低电平触发。触发方式的选择由 MCUCR 中的中断检测控制位 ISC00、ISC01 和 ISC11、ISC10 控制。当设置为低电平触发时，只要 MCU 一检测相应在引脚的电平为低，就可以触发中断并且一直保持中断。另外，即使 INT0 引脚设置为输出，其引脚上的电平变化也可以引发一个中断请求。

提示：外部中断提供一种软件触发中断的方法。还是以 INT0 为例，中断使能并且 INT0 引脚配置为输出，中断 IS01：IS00=11。

```
DDRD|=0x04;                    //INT0(PD2)配置为输出
GIMSK|=0x80;                   //使能 INT0
SEI();                         //全局中断使能
PORTD&=~0x04;                  //设 INT0(PD2)为低电平
PORTD|=0x04;                   //高电平，产生上升沿并触发中断
```

4.7.2 中断操作的 C 语言源程序及剖析

用 ICCAVR 编程，在 C 中只要用#pragma 伪指令和中断向量说明中断服务程序的入口地址即可。

```
#pragma interrupt_handler <函数名>:<中断向量>
```

如定义使用 INT0 中断服务程序：

```
#pragma interrupt_handler int0_isr:2
void int0_isr(void)
{
//INT0 中断服务程序
}
```

中断向量号如表 4.7 所示，要注意，中断向量号是从“1”开始的。C 编译器会根据中断向量号自动生成程序中的中断向量，并且自动保存和恢复在函数中用到的全部寄存器。而如果用汇编语言编写中断处理函数，我们必须自己完成这些工作。编译后生成的汇编程序：

```
.area vector(rom, abs);        程序空间说明
.org 2;                        中断向量地址=(中断向量-1)*2
rjmp _int0_isr;                跳转到中断服务程序
```

```
_int0_isr:;                    中断服务程序入口
     st -y,R0
     in R0,0x3f;               保存 SREG
     st -y,R0;                 注意堆栈增长方向：由顶向下
     ;                         用户程序
     ld R0,y+
     out 0x3f,R0;              恢复 SREG
     ld R0,y+
     reti;                     中断返回
```

提示：如果希望多个中断源使用同一个中断服务程序，可以用不同的中断向量声明多次，如 INT0 和 INT1 使用同一个中断服务函数：

```
#pragma interrupt_handler int0_isr:2
#pragma interrupt_handler int0_isr:3
void int0_isr(void)
   {
   //INT0 中断服务程序
   }
```

编译后生成的汇编程序为：

```
.area vector(rom, abs)
.org 4
rjmp _int1_isr;                跳转到中断服务程序 int1_isr
.area vector(rom, abs)
.org 6
rjmp _int1_isr;                跳转到同一个中断服务程序 int1_isr
_int0_isr:;                    中断服务程序入口
    ;                          中断服务程序内容
    reti;                      中断返回
```

4.7.3　4×4 按键唤醒电路

我们这里讨论如何用 C 实现 4×4 按键唤醒，原理图如图 4.13 所示，当没有键按下时，AVR 单片机处于 POWER-DOWN 模式，一旦有按键按下，AVR 将结束 POWER-DOWN 状态进入正常运行模式。如果“0”键被按下，红色 LED1 将闪动 10 次，当其他键按下时，绿色 LED2 将闪动相应的次数，如按下 5 键，LED2 闪动 5 次；按下 9 键，LED2 将闪动 9 次。

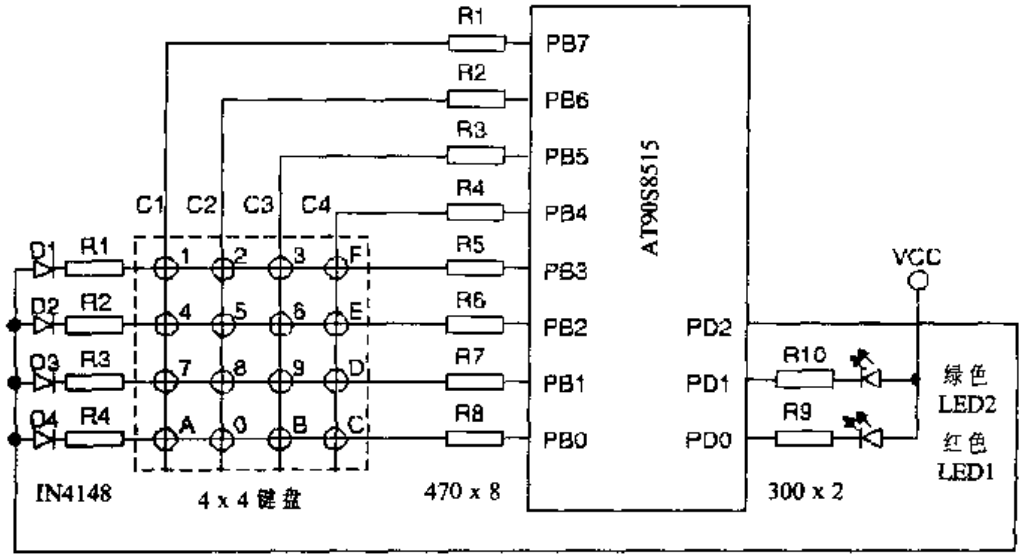


图 4.13 4×4 按键唤醒电路

在这个系统中，可以将之分解为初始化、中断服务和闪动 LED 三个模块编程。

1. 初始化模块

初始化程序完成以下工作：

(1) 初始化 I/O 端口方向寄存器；

(2) 配置外部中断 INT0 为低电平触发并使能；

(3) 设置 POWER-DOWN 模式允许位(MCUCR 中的 SE 和 SM)；

(4) 进入 SLEEP 模式。

主程序的工作是循环等待中断发生，因此将初始化程序写在 main 程序里。

```
#include <io8515v.h>
#include <macros.h>
void main(void)
    {
    DDRD=0xFB;                    //PD2(INT0) IN
    MCUCR=0X30;                   //打开 SE 和 SM 位，允许 SLEEP 和 POWER-DOMN 模式
    GIMSK=0x40;                   //允许外部中断 INT0,低电平触发
    for(;;)
        {
        CLI();
        DDRB=0xF0;                //PB.0~3 输入，PB.4~7 输出
        PORTB=0x0F;               //键盘行线内部上拉，列线低电平
        PORTD=0x07;               //关闭两个 LED
        SEI();
        asm("sleep");
        led_flash();              //中断服务返回后开始调用闪动 LED 程序
```

```
        }
    }
```

2．中断服务程序

AT90S8515 进入 SLEEP 模式后，一旦有按键按下，将触发外部中断 INT0，单片机被唤醒，开始执行中断服务程序。它通过键盘扫描来确定是哪个按键被按下，并存储按键值，程序流程如图 4.14 所示。

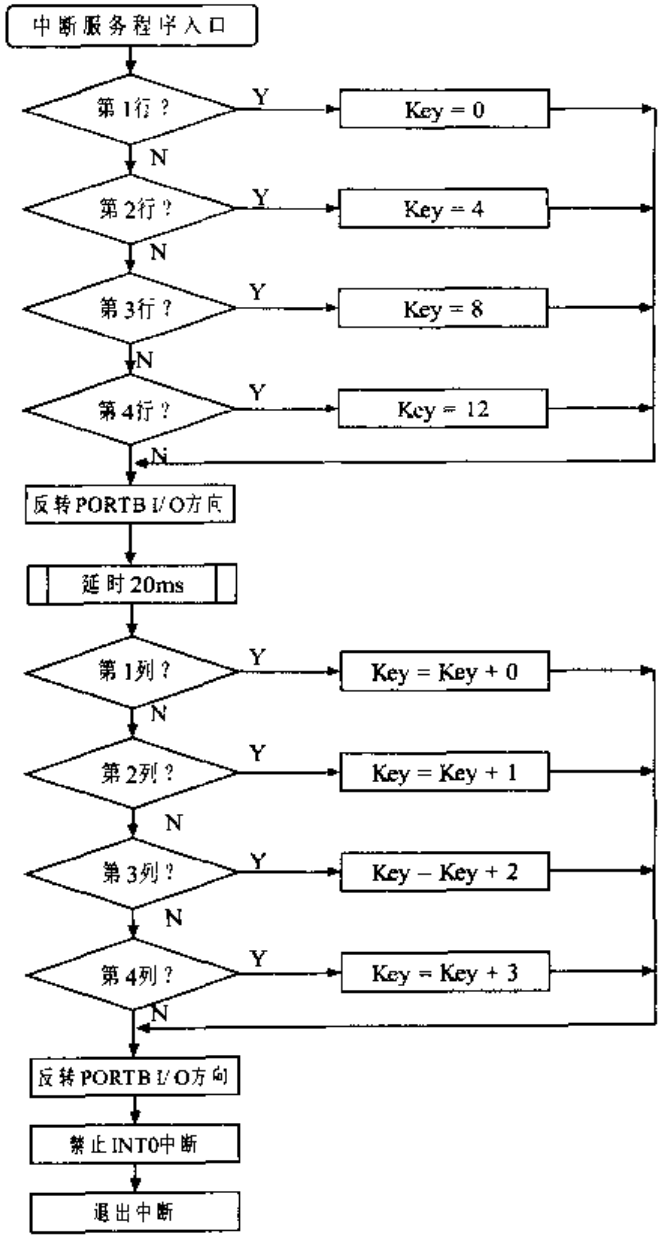


图 4.14　中断服务流程图

```
/*  定义 Key 为全局变量，用以记录按键的数值，再定义一个一维数组，用 Key 来做索引，就能得到按键的数值  */
unsigned char Key;
```

```
unsigned char KeySet[ ]=
    {
    1, 2, 3, 15,
    4, 5, 6, 14,
    7, 8, 9, 13,
    10,0, 11,12
    };
/*  如按下“8”键，第 3 行第 2 列，中断服务程序扫描的 Key=9，则按键值为
KeySet[Key]=KeySet[9]=8     */
/*  为去除键盘扫描抖动，扫描过程需要一个延时程序，作为按键消抖用  */
void delay_1ms(void)
    {
    u0nsigned int i;
    for (i=1;i<570;i++)
         ;
    }

void delay(unsigned int n)
    {
    unsigned int i=0;
    while (i<n)
        {
        delay_1ms();
        i++;
        }
    }
/*  INT0 的中断向量为 2，下面定义中断服务程序  */
#pragma interrupt_handler int0_isr:2
void int0_isr(void)
    {
    GIMSK=0;                    //禁止外部中断
    if(PINB&0x08)               //第 1 行
        Key=0;
    else if(PINB&0x04)          //第 2 行
        Key=4;
    else if(PINB&0x02)          //第 3 行
        Key=8;
    else if(PINB & 0x01)        //第 4 行
        Key=12;
    DDRB=0x0F;                  //反转 PORTB I/O 方向
    PORTB=0xF0;                 //置行为低电平
    delay(20);                  //延时约 20ms
```

```
if(PINB&0x80)                    //第 1 列
    Key+=0;
else if(PINB & 0x40)             //第 2 列
    Key+=1;
else if(PINB & 0x20)             //第 3 列
    Key+=2;
else if(PINB & 0x10)             //第 4 列
    Key+=3;
DDRB=0xF0;                       //再次反转 PORTB I/O 方向
PORTB=0x0F;                      //置列为低电平
GIMSK=0x40;                      //允许外部中断
}
```

3. LED 闪动程序

闪动 LED 程序需要一个参数，即存在全局变量 Key 中的按键数值，闪动函数根据 Key 中的参数，闪动相应的次数。程序流程图如图 4.15 所示。

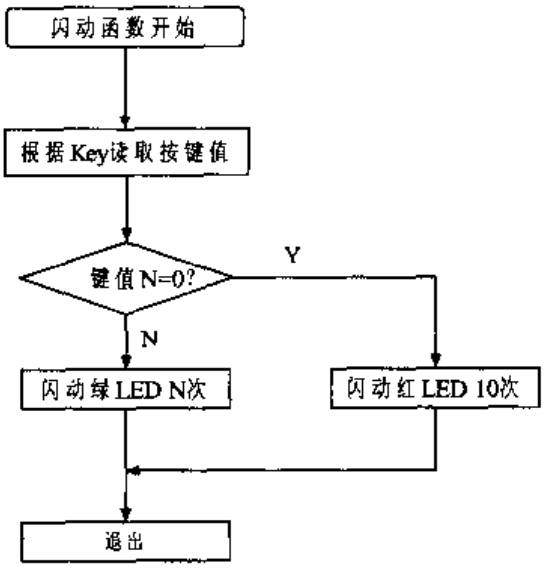


图 4.15　闪动 LED 流程图

```
void led_flash(unsigned char temp)
   {
   if(!temp)
        {
        temp=10;                 //如果键值=0,循环次数为 10
        while(temp)              //如果循环值!=0
            {
            PORTD|=0x01;         //打开红色 LED
            delay(250);          //延时 500ms
            delay(250);
```

```
            PORTD &=0x01;          //关闭红色 LED
            delay(250);
            delay(250);
            temp--;
            }
        }
    else
        {
        while(temp)
            {
            PORTD|=0x02;           //打开绿色 LED
            delay(250);            //延时 500ms
            delay(250);
            PORTD&=0x02;           //关闭绿色 LED
            delay(250);
            delay(250);
            temp--;
            }
        }
}
```

4.8 定时/计数器

AT90S8515 提供一个 8 位定时/计数器和一个 16 位定时/计数器，共两个定时/计数器，它们可以独立的从 10 位预分频器取得预分频时钟，并且既可以作为使用片内预分频时钟的定时器，也可作为对外部触发信号计数的计数器，其结构如图 4.16 所示。

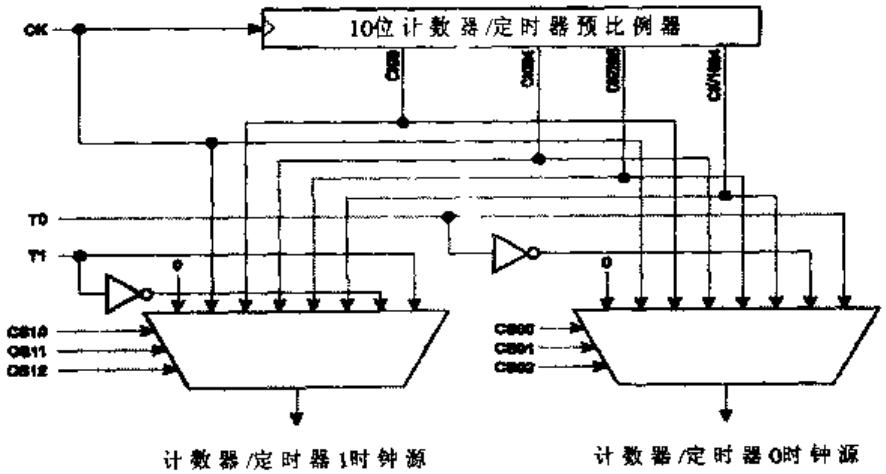


图 4.16 定时/计数器的预分频器

4.8.1　定时/计数器操作的特点

4.8.1.1　定时/计数器 0

定时/计数器 0 的时钟源可选择 CK、预分频比例器或外部引脚输入。TIFR 为状态标志寄存器，TCCR0 为控制寄存器，而 TIMSK 控制 T/C0 的中断屏蔽。

当 T/C0 由外部时钟信号驱动时，为了保证 CPU 对信号的正确采样(MCU 在内部 CPU 时钟的上升沿对外部信号进行采样)，要保证外部信号的转换时间至少为一个 CPU 时钟周期。

1．计数器/定时器 0 的控制寄存器—　TCCR0，其定义如表 4.10 所示。

表 4.10　定时/计数器 0 的控制寄存器—TCCR0

位	7	6	5	4	3	2	1	0
TCCR0(0x33)	-	-	-	-	-	CS02	CS01	CS00
R/W	R	R	R	R	R	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

TCCR0 中的 CS02~CS00 定义定时/计数器 0 的时钟源，默认值为“0”。

2．计数器/定时器 0 计数寄存器——TCNT0，其时钟源选择方式如表 4.11 所示，控制寄存器的定义如表 4.12 所示。

表 4.11　定时/计数器 0 的时钟源选择

CS02	CS01	CS00	说　明
0	0	0	停止，定时/计数器 0 被停止
0	0	1	CK
0	1	0	CK/8
0	1	1	CK/64
1	0	0	CK/256
1	0	1	CK/1024
1	1	0	外部 T0 引脚，下降沿触发
1	1	1	外部 T0 引脚，上升沿触发

时钟分频的模式从晶振时钟直接换算，停止条件提供定时器触发/禁止功能。如果使用外部引脚模式，相应设置必须在数据方向控制寄存器中完成。

提示：当定时/计数器 0 由外部引脚 T0 驱动时，即使 PB0(T0)配置为输出，管脚上的信号变化照样可以使计数器发生相应的变化，这就为用户提供了一个软件控制的方法。

定时/计数器 0 的 TCNT0 是可以进行读/写访问的向上计数器。若定时/计数器 0 被写入，同时时钟源有时钟输入，定时/计数器 0 就会在写入的值的基础上继续计数。

表 4.12 定时/计数器 0 的计数寄存器—TCNT0

	7	6	5	4	3	2	1	0
TCNT0(0x32)	MSB							LSB
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

4.8.1.2 16 位定时/计数器 1

16 位定时/计数器 1 的时钟可以从 CK、预分频比例器或外部引脚输入三种时钟源中选择。TIFR 寄存器为状态标志寄存器，而 TIMSK 寄存器控制定时/计数器 1 的中断屏蔽，定时/计数器 1 的控制寄存器 TCCR1A 和 TCCR1B 中提供溢出、比较匹配和捕获事件等标志以及控制信号。

当定时器/计数器 1 使用外部时钟信号驱动时，为了保证 CPU 对信号的正确采样，要保证外部信号的转换时间至少为一个 CPU 时钟周期，MCU 在内部 CPU 时钟的上升沿对外部信号进行采样。

定时/计数器 1 提供输出比较寄存器 OCR1A 和 OCR1B 作为与定时/计数器 1 内容进行比较的数据源，用来实现两种输出比较功能，如比较 A 匹配发生时清除计数器和比较输出引脚的动作等。

定时/计数器 1 还可以用作 8、9 或 10 位的脉冲调制器(PWM)，在此模式下计数器和 OCR1A/OCR1B 寄存器具有集中脉冲双抗误操作独立的 PWM 能力。

定时/计数器 1 提供输入捕获(ICP)功能，当输入捕捉引脚 ICP 发生相应事件时定时/计数器 1 的值将被传到输入捕捉寄存器 ICR1。实际的捕获事件设置由定时/计数器 1 的控制寄存器 TCCR1B 来定义。另外，模拟比较器也可以触发该输入捕获事件。ICP 的引脚逻辑图如图 4.17 所示。

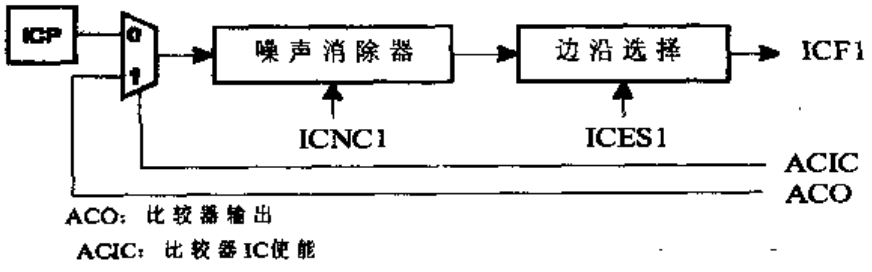


图 4.17 ICP 引脚原理图

1．定时/计数器1的控制寄存器—TCCR1A，其定义如表4.13所示。

表 4.13 定时/计数器 1 的控制寄存器—TCCR1A

	7	6	5	4	3	2	1	0
TCCR1A(0x2F)	COM1A1	COM1A0	COM1B1	COM1B0	-	-	PWM11	PWM10
R/W	R/W	R/W	R/W	R/W	R	R	R/W	R/W
初始值	0	0	0	0	0	0	0	0

COM1A1、COM1A0 比较输出模式 1A：

COM1A1 和 COM1A0 控制位决定定时/计数器 1 中比较匹配之后的输出引脚事件，输出引脚事件影响 OC1A，即输出比较 A 引脚。由于比较匹配发生时输出引脚 OC1A 的动作是 I/O 口的第二功能，相应的方向控制位要设置为 1，以便将其配置为输出。表 4.14 为比较 1 模式选择。

COM1B1、COM1B0 比较输出模式 1B，其定义如表 4.14 所示。

COM1B1 和 COM1B0 控制位决定定时/计数器 1 中比较匹配之后的输出引脚事件。输出引脚事件影响 OC1B，即输出比较 B 引脚。由于比较匹配发生时输出引脚 OC1B 的动作是 I/O 口的第二功能，相应的方向控制位要设置为 1，以便将其配置为输出。

表 4.14　比较 1 模式选择

COM1X1	COM1X0	说　明
0	0	定时/计数器 1 与输出引脚 OC1X 不连接
0	1	触发 OC1X 输出线
1	0	清除 OC1X 输出线(为 0)
1	1	设置 OC1X 输出线(为 1)

其中，X=A 或 B。

提示：在 PWM 模式这些位具有不同的功能。表 4.15 为 PWM 模式选择。

表 4.15　PWM 模式选择

PWM11	PWM10	说　明
0	0	禁止定时/计数器 1 的 PWM 操作
0	1	定时/计数器 1 为 8 位 PWM
1	0	定时/计数器 1 为 9 位 PWM
1	1	定时/计数器 1 为 10 位 PWM

2．定时/计数器 1 的控制寄存器—TCCR1B，其定义如表 4.16 所示。

表 4.16　定时/计数器 1 的控制寄存器—TCCR1B

	7	6	5	4	3	2	1	0
TCCR1B(0x2E)	ICNC1	ICES1	-	-	CTC1	CS12	CS11	CS10
R/W	R/W	R/W	R	R	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

ICNC1，输入捕获噪音清除器(4 CK)：

当 ICNC1 位被清“0”时，输入捕获触发噪音清除功能被禁止，有效的输入捕捉在 ICP 输入捕捉引脚上采样的第一个上升/下降边沿触发；当 ICNC1 位置“1”时，ICP 信号要进行 4 次连续采样，只有 4 个采样值都有效时输入捕捉标志才置位。实际的采样频率为系统

晶振 XTAL 时钟。

ICES1，输入捕捉 1 边沿选择：

当 ICES1 位为 0 时，定时/计数器 1 的值在 ICP 引脚电平的下降沿被传送到输入捕捉寄存器 ICR1；

若 ICES1 位为 1 时，定时/计数器 1 的值在 ICP 引脚电平的上升沿被传送到输入捕捉寄存器 ICR1。

CTC1，在比较匹配后清除定时/计数器 1：

当 CTC1 位被设为“1”时，比较 A 匹配事件发生后，定时/计数器 1 将被复位到 0x0000。若 CTC1 位是“0”，则定时/计数器 1 将继续计数而不受比较匹配的影响，并且直到它被停止、清除、溢出或改变方向。表 4.17 为定时/计数器 1 的预分频选择。

由于比较匹配事件的检测发生在匹配发生之后的一个 CPU 时钟，故而定时器预分频比率的不同将引起此功能有不同的表现。当预分频为 1 A 比较匹配寄存器的值设置为 C 时定时器的记数方式为：

...| C-2 | C-1 | C | 0 | 1 |...

而当预分频为 8 时定时器的计数方式则为：

...| C-2, C-2, C-2, C-2, C-2, C-2, C-2, C-2 | C-1, C-1, C-1, C-1, C-1, C-1, C-1, C-1 | C, 0, 0, 0, 0, 0, 0, 0 |...

在 PWM 模式下这几位没有作用。

CS12，CS11，CS10 选择定时/计数器 1 的时钟源，其定义如表 4.17 所示。

表 4.17 定时/计数器 1 的预分频选择

CS12	CS11	CS10	说 明
0	0	0	停止，定时/计数器 0 被停止
0	0	1	CK
0	1	0	CK/8
0	1	1	CK/64
1	0	0	CK/256
1	0	1	CK/1024
1	1	0	外部 T0 引脚，下降沿触发
1	1	1	外部 T0 引脚，上升沿触发

当定时/计数器 1 由外部引脚 T1 驱动时，即使 PB1(T1)配置为输出，管脚上的信号变化照样可以使计数器发生相应的变化，这就为用户提供了一个软件控制的方法。

3．计数器/定时器 1 ——TCNT1H 和 TCNT1L，其定义如表 4.18 所示。

表 4.18 定时/计数器 1 TCNT1H 和 TCNT1L

	7	6	5	4	3	2	1	0
TCNT1H(0x2D)	MSB							
TCNT1L(0x2C)								LSB

TCNT 寄存器(16 位)的值是可读写的，初始值为 0x0000。

此 16 位寄存器包含了 16 位定时/计数器 1 的值。当 CPU 访问这两个寄存器时为了保证高字节和低字节能够同时读写，要用到一个 8 位的临时寄存器 TEMP。此寄存器在访问 OCR1A、OCR1B 和 ICR1 的时候也要用到。

注意：如果主程序和中断程序在访问寄存器时都要用到 TEMP，那么在适当的时候需要关闭中断使能，防止出错。

写定时/计数器 1 TCNT1：

当 CPU 写高字节 TCNT1H 时，数据将被放置在 TEMP 寄存器，当 CPU 写低字节 TCNT1L 时，此数据及 TEMP 中的字节数据组合，且全部的 16 位数据被同步的写入 TCNT1。因此在写 16 位寄存器 TCNT1 时，首先要写高字节 TCNT1H。

读定时/计数器 1 TCNT1：

当 CPU 读取低位字节 TCNT1L 时，低字节 TCNT1L 的数据将送入 CPU，同时高字节 TCNT1H 将被送入 TEMP 寄存器，等到 CPU 读取 TCNT1H 时，TEMP 寄存器中的数据送入 CPU，因此在读 16 位的 TCNT1 时，首先要读低字节 TCNT1L。

4. 计数器/定时器 1 输出比较寄存器 OCR1AH 和 OCR1AL，其定义如表 4.19 所示。

表 4.19　定时/计数器 1 输出比较寄存器 OCR1AH 和 OCR1AL

	7	6	5	4	3	2	1	0
OCR1AH(0x2B)	MSB							
OCR1AL(0x2A)								LSB

5. 计数器/定时器 1 输出比较寄存器 OCR1BH 和 OCR1BL，其定义如表 4.20 所示。

表 4.20　定时/计数器 1 输出比较寄存器 OCR1BH 和 OCR1BL

	7	6	5	4	3	2	1	0
OCR1BH(0x29)	MSB							
OCR1BL(0x28)								LSB

OCR1B 寄存器的初始值为 0x0000。

定时/计数器 1 输出比较寄存器是一个 16 位的读/写寄存器，初始值都是 0x0000。定时/计数器 1 输出比较寄存器包括将要连续的与定时/计数器 1 的值相比较的数据。比较匹配的操作在定时/计数器 1 的控制和状态寄存器中定义。

由于输出比较寄存器 OCR1A/OCR1B 为 16 位寄存器，当 OCR1A/OCR1B 被写入时，需使用临时寄存器 TEMP 来确保全部的字节被同时写入，当 CPU 写高位字节时，OCR1AH/OCR1BH 数据被临时的存储在寄存器 TEMP 中；当 CPU 写低字节时，OCR1AL/OCR1BL 和 TEMP 寄存器被同步的写入 OCR1A/OCR1B。

6. 计数器/定时器 1 输入捕捉寄存器 ICR1H 和 ICR1L，其定义如表 4.21 所示。

表 4.21 定时/计数器 1 输入捕捉寄存器 ICR1H 和 ICR1L

	7	6	5	4	3	2	1	0
ICR1H(0x25)	MSB							
ICR1L(0x24)								LSB

输入捕获寄存器是一个 16 位的只读寄存器。按照 ICES1 的设定，输入捕捉引脚 ICP 上信号发生上跳变或下跳变时，定时/计数器 1 的 TCNT1 值被送入输入捕获寄存器 ICR1，同时，输入捕获标志位 ICF1 被置位。

由于输入捕获寄存器 ICR1 为一个 16 位寄存器，当 ICR1 被读出时，使用一个临时寄存器 TEMP，以确保全部的字节被同时读出。当 CPU 读取低位字节 ICR1L 时，数据被送入 CPU 且高位字节 ICR1H 的数据被放置在 TEMP 寄存器中。当 CPU 读取高位字节 ICR1H 中的数据时，CPU 接收 TEMP 寄存器中的数据。所以在访问时要用到 TEMP 寄存器以保证同时读取两个字节，读写过程与读写 TCNT1 相同。

7. PWM 模式下的定时/计数器 1

当选择 PWM 模式时，定时/计数器 1 以及输出比较寄存器 OCR1A/OCR1B 共同组成两个 8、9 或 10 位无尖峰自由运行的 PWM，其 TOP 的值与频率的关系如表 4.22 所示，且在引脚 PD5(OC1A)和 OC1B 上带有输出的节拍修正 PWM。定时/计数器 1 作为向上/向下的计数器，从 0x0000 向上记数到 TOP，在重复循环之前，它反转，并向下记数到 0x0000。

表 4.22 TOP 值及 PWM 频率

PWM 分辨率	TOP 的值	频 率
8 位	0x00FF(255)	fTC1/510
9 位	0x01FF(511)	fTC1/1022
10 位	0x03FF(1023)	fTC1/2046

当计数器中的数值和 OCR1A/OCR1B 的数值低 8、9 或 10 位一致时，PD5(OC1A)/OC1B 引脚根据在定时/计数器 1 控制寄存器 TCCR1A 中 COM1A0/COM1A1 和 COM1B0/COM1B1 的设置被设置或清除，表 4.23 为 COM1X1 和 COM1X0 的设置与 OCX1 引脚的关系。

表 4.23 PWM 模式下的比较 1 模式选择

COM1X1	COM1X0	在 OCX1 上的作用
0	0	不用作 PWM 功能
0	1	不用作 PWM 功能
1	0	清比较匹配，向上计数；置比较匹配，向下计数(PWM 不反转)
1	1	清比较匹配，向下计数；置比较匹配，向上计数(PWM 反转)

其中，X = A 或 B。

当输出比较寄存器 OCR1A 的值为 0x0000 或 TOP 时(如表 4.24 所示)，输出 OC1A/OC1B

根据 COM1X1/COM1X0 的设置保持低或高，其关系如表 4.24 所示。

表 4.24　OCR1X=$0000 或 TOP 时的 PWM 输出

COM1X1	COM1X0	OCR1X	OCR1X 输出
1	0	0x0000	L
1	0	TOP	H
1	1	0x0000	H
1	1	TOP	L

X = A 或 B，H=高电平，L=低电平

下面介绍与中断有关的两个数据寄存器：定时/计数器中断屏蔽寄存器 TIMSK 和定时/计数器中断标志寄存器 TIFR。

8．中断屏蔽寄存器——TIMSK，其定义如表 4.25 所示。

表 4.25　中断屏蔽寄存器 TIMSK

	7	6	5	4	3	2	1	0
TIMSK(0x39)	TOIE1	OCIE1A	OCIE1B	-	TICIE1	-	TOIE0	-
R/W	R/W	R/W	R/W	R	R/W	R	R/W	R-
初始值	0	0	0	0	0	0	0	0

9．中断标志寄存器 TIFR，其定义如表 4.26 所示。

表 4.26　中断标志寄存器 TIFR

	7	6	5	4	3	2	1	0
TIFR(0x38)	TOV1	OCF1A	0CF1B	-	ICF1	-	TOV0	-
R/W	R/W	R/W	R/W	R	R/W	R	R/W	R-
初始值	0	0	0	0	0	0	0	0

TOIE0/TOIE1，定时/计数器溢出中断触发使能

TOV0/TOV1，定时/计数器溢出中断标志

以定时/计数器 1 为例，当 TOIE1 和全局中断使能位 I 都为“1”，定时/计数器溢出中断事件触发中断，中断标志寄存器 TIFR 中溢出标志 TOV1 被置“1”，相应的中断服务程序将被触发。执行中断服务程序时，硬件将自动消零溢出中断标志 TOV1，TOV0/TOV1 中断标志也可通过写“1”来清零。

OCIE1A/OCIE1B，定时/计数器 1 输出比较匹配(A/B)中断使能

OCF1A/OCF1B，定时/计数器 1 输出比较匹配(A/B)标志

以定时/计数器比较匹配 A 中断为例，当 OCIE1A 和全局中断允许位 I 都为“1”，比较匹配 A 中断事件触发中断，中断标志寄存器 TIFR 中比较 A 标志 OCF1A 被置“1”，相应的中断服务程序将被触发。执行中断服务程序时，硬件将自动清零溢出中断标志 OCF1A，

OCF1A/OCF1B 中断标志也可通过写“1”来清零。

TICIE1，定时/计数器 1 输入捕获中断使能

ICF1，定时/计数器 1 输入捕获中断标志

当 TICIE1 和全局中断允许位 I 都为“1”，输入捕获中断事件触发中断，输入捕获中断标志位 ICF1 被置“1”(表明定时/计数器的值已被输入到捕获寄存器 ICR1)，相应的中断服务程序将被触发。执行中断服务程序时，硬件将自动清零溢出中断标志 ICF1。ICF1 中断标志也可通过写“1”来清零。

其他位保留，只读并且总为“0”。

4.8.2 定时/计数器操作的 C 语言源程序及剖析

4MHz 晶振下，使用定时器 1 的 1s 定时中断，选择 1024 分频，计数初始值为 0xF0BE，先进行初始化：

```
void timer1_init(void)
      {
      TCCR1B=0x00;              //先停止定时器 1
      TCNT1H=0xF0;              //设置计数初始值
      TCNT1L=0xBE;
      OCR1AH=0x00;              //不使用输出比较匹配 A
      OCR1AL=0x00;
      OCR1BH=0x00;              //不使用输出比较匹配 B
      OCR1BL=0x00;
      TCCR1A=0x00;
      TCCR1B=0x05;              //启动定时器 1，使用 1024 分频
      }
```

用编译附注说明下面的函数为中断函数：

```
#pragma interrupt_handler timer1_ovf_isr:7
void timer1_ovf_isr(void)
      {
      TCNT1H=0xF0;
      TCNT1L=0xBE;
      //完成每秒一次的工作
      }
```

注意：CPU 访问定时/计数器 1 的 TCNT1H 和 TCNT1L 等 16 位寄存器时，需使用一个 8 位暂存寄存器 TEMP，以确保 16 位寄存器的高低字节同时被读写。无论读写，CPU 都是

将 16 位的高字节暂存在 TEMP 中，所以读取时，先读低字节；而写入时，应先写高字节。如上面初始化 TCNT1H 和 TCNT1L：

```
TCNT1H=0xF0;                    //写向 TCNT1H 的值被暂存在 TEMP 中，TEMP=0xF0
TCNT1L=0xBE;                    //写低字节 TCNT1L 时，TEMP 的值被同时写入高字节中
                                //TEMP(0xF0) ->  TCNT1H
                                // 0xBE          -> TCNT1L
```

4.8.3　60Hz 时钟发生器

下例用 AT90S8515 作一个秒表，其原理图如图 4.18 所示，我们用 LED 数码管显示秒和十分之一秒，按下“复位”键清除 LED 显示；按下“开始/停止”键开始计时，并在 LED 上动态显示；再次按下“开始/停止”停止计时，LED 显示静止；再次按下“开始/停止”将继续上一次计时，LED 动态显示。最高计时时间为 99.9 秒，超过后从“0”开始计时。

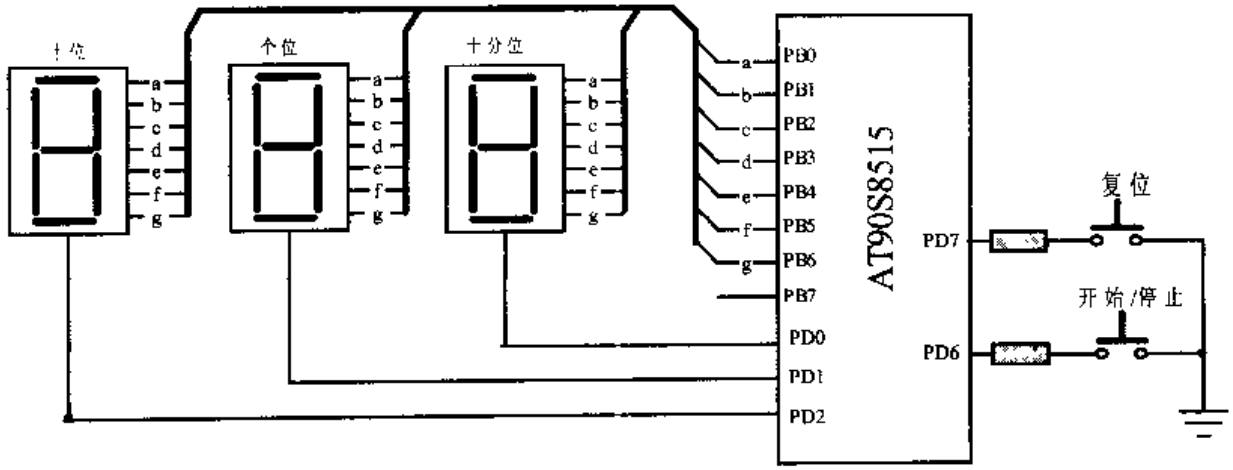


图 4.18　60Hz 时钟发生器

原理图说明：

(1) 原理图中没有画出电源及晶振管脚。

(2) 由于仅仅作为演示，只是为了说明如何通过扫描的方式使用数码管及如何编写中断定时程序，因此在这里直接使用 PD0~PD2 作为位选，用以驱动小型的数码管。如果数码接得较多或使用耗电量大(体积大)的数码管，应在位选(PD0~PD2)的每个管脚上加入三极管进行驱动。

共阴极七段数码 LED 显示 0~9 10 个数字，使用 PB0~PB6 驱动，PD0、PD1 和 PD2 选择位码，即要刷新的 LED。两个按键由 PD6 和 PD7 引脚检测。

为获得较准确的时间定时，我们使用 16 位定时器 TIMER1。系统使用 4MHz 晶振，采用 256 分频，100ms(1/10s)定时，我们可算得 TCHNT = 0xF9E6。

我们需要定义一个七段译码字形表，显示数字 0~9，可以定义在程序空间：

```
#pragma data:code
const unsigned tabs[] =
    {
    0x3F,  //"0"
    0x06,  //"1"
    0x5B,  //"2"
    0x4F,  //"3"
    0x66,  //"4"
    0x6D,  //"5"
    0x7D,  //"6"
    0x07,  //"7"
    0x7F,  //"8"
    0x6F  //"9"
    };                              //只要显示 0~9 即可
#pragma data:data
/*  需要三个 LED 显示数据寄存器，存储每个 LED 显示数据(BCD 码)  */
static unsigned char ledbuf[3];
/*  定义为全局静态变量，并且在启动时自动初始化  */
/*  秒和 1/10 秒存储定时更新的时间参数，它们都是全局变量  */
unsigned char sec;
unsigned char dsec;
/*  还需要一个全局变量，来打开/关闭定时器  */
unsigned char flag;
```

整个程序的流程如图 4.19 所示，流程可以分为初始化、中断服务程序、LED 扫描程序、按键扫描和主程序共五个功能模块，我们下面分别介绍这些模块：

1. 初始化

```
/*  初始化系统，打开 TIMER1 溢出中断使能，使用 100ms 定时  */
void sys_init(void)
    {
    0DDRD=0x3F;
    DDRB=0xFF;
    PORTD=0x07;
    PORTB=0x3F;
    TIMSK=0x80;                 //TIMER1 溢出中断使能
    TCCR1B=0x00;                //停止 TIMER1
    TCNT1H=0xF9;                //使用 100ms 定时
    TCNT1L=0xE6;
    TCCR1A=0x00;
    TCCR1B=0x04;                //使用 256 分频，启动定时器
```

```
SEI();                          //打开全局中断允许位
}
```

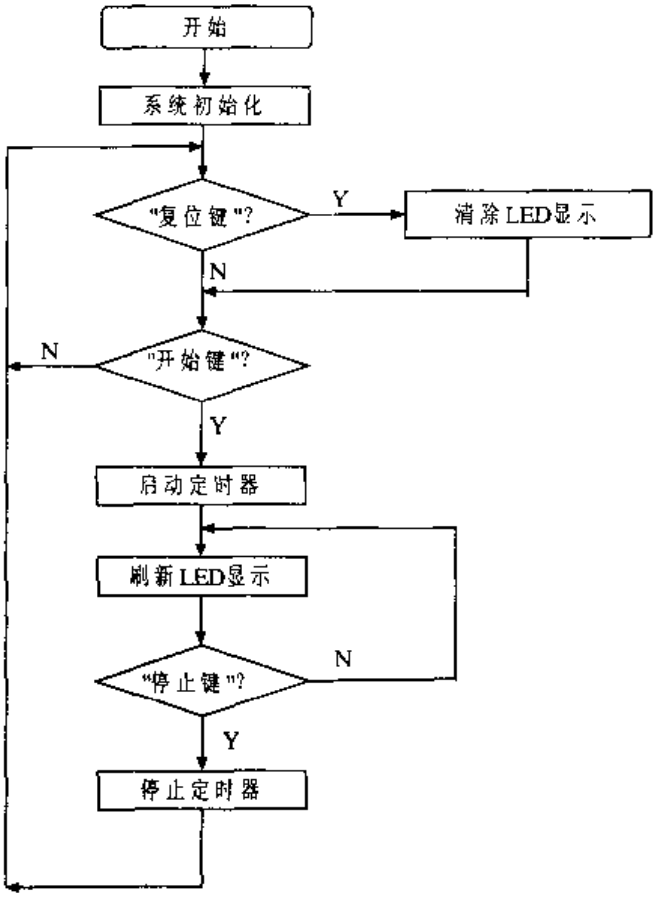


图 4.19　60Hz 秒表软件流程图

2. 中断服务程序

```
/*  TIMER1 溢出中断服务程序，负责刷新时间参数，刷新 LED 显示寄存器，并把计时参数变换到三个 LED 寄存器联 */
#pragma interrupt_handler timer1_ovf_isr:7
void timer1_ovf_isr(void)
      {
      TCNT1H=0xF9;              //重新装载定时器
      TCNT1L=0xE6;
      if(flag)
      {                                 //如果定时器允许
            dsec++;
            if(dsec>9)
                  {                     //十分位达到 10 后进位
                  dsec=0;
                  sec++;                //不必关心 sec 超过 255
```

```
            }
        ledbuf[0]=dsec;        //十分秒位
        ledbuf[1]=sec%10;          //个位
        ledbuf[2]=sec/10;          //十位
        }
    }
```

3. LED 扫描程序

/* LED 动态扫描显示，循环扫描时每一位显示要保持一定时间(通常为 2ms~5ms，视具体的应用而定)，我们可以使用前面的 delay 函数，在本例中计划延时 5ms，因为本例很简单，延时长一些并不影响其他的工作，选用 5ms，每个数码管刷新频率约为 60Hz，不会有闪烁感 */

```
void led_display(void)
    {
    unsigned char i;
    for(i=0;i<3;i++)
        {
        PORTB=ledbuf[i];            //将显示缓冲区数据送到 PORTB 口
        PORTD=~(1<<i);          //开始显示
        delay(2);                   //延时(2+2)ms
        PORTD|=(1<<i);          //关闭显示
        }
    }
```

4. 按键扫描

/* 按键扫描，返回参数区分按键的类型：无按键(0)、复位键(0x05)和开始/停止键(0x50) */

```
unsigned char key_scan(void)
   {
   unsigned char ret_key;
   ret_key = 0;                     //无按键
   if(!(PIND & 0x80))               //按下复位键，返回 0x05
   ret_key = 0x05;
   if(!(PIND & 0x40))               //按下开始/停止键，返回 0x50
   ret_key = 0x50;
   return ret_key;
   }
```

5. 主程序

```
void main(void)
```

```
{
unsigned char tmp key;
MCUCR=0;
sys_init();
flag=0;
for(;;)
     {
     tmp_key=scan_key();
     if(tmp_key)
          {                          //有按键按下
          if(tmp_key&0x0F)
               {                     //开始/停止键
               flag^= 1;      //第一次按，flag =1;第二次按，flag = 0
               }
          else if(tmp_key&0xF0)
               {                     //复位键
               ledbuf[0]=0x3F;       //清零所有位
               ledbuf[1]=0x3F;
               ledbuf[2]=0x3F;
               }
          }
          led_display();
     }
}
```

4.9　访问 UART

4.9.1　访问 UART 操作的特点

AT90S8515 具有一个全双工的通用异步收发器，它能在使用较低频率的时钟(3.6864MHz)时产生 115200bps 的波特率。UART 可设置为 8 位或 9 位数据，具有帧错误检测和噪声滤波功能。与 51 系列不同的是，AT90S8515 UART 有 3 个独立的中断源：发送结束、发送数据寄存器空和接收结束。图 4.20 为 UART 发送器的方框示意图。

4.9.1.1　数据发送

把待发送的数据写入 UART 数据寄存器就开始发送数据，若连续发送，只有前一个字符的停止位移出移位寄存器，UDR 的数据才送入移位寄存器。UART 数据发送先是起始位，

然后是数据，低位在前，高位在后。如果 UCR 中的 CHR9 为“1”，即选择了 9 位数据格式，则 UCR 中的 TXB8 将送到移位寄存器的位 9。

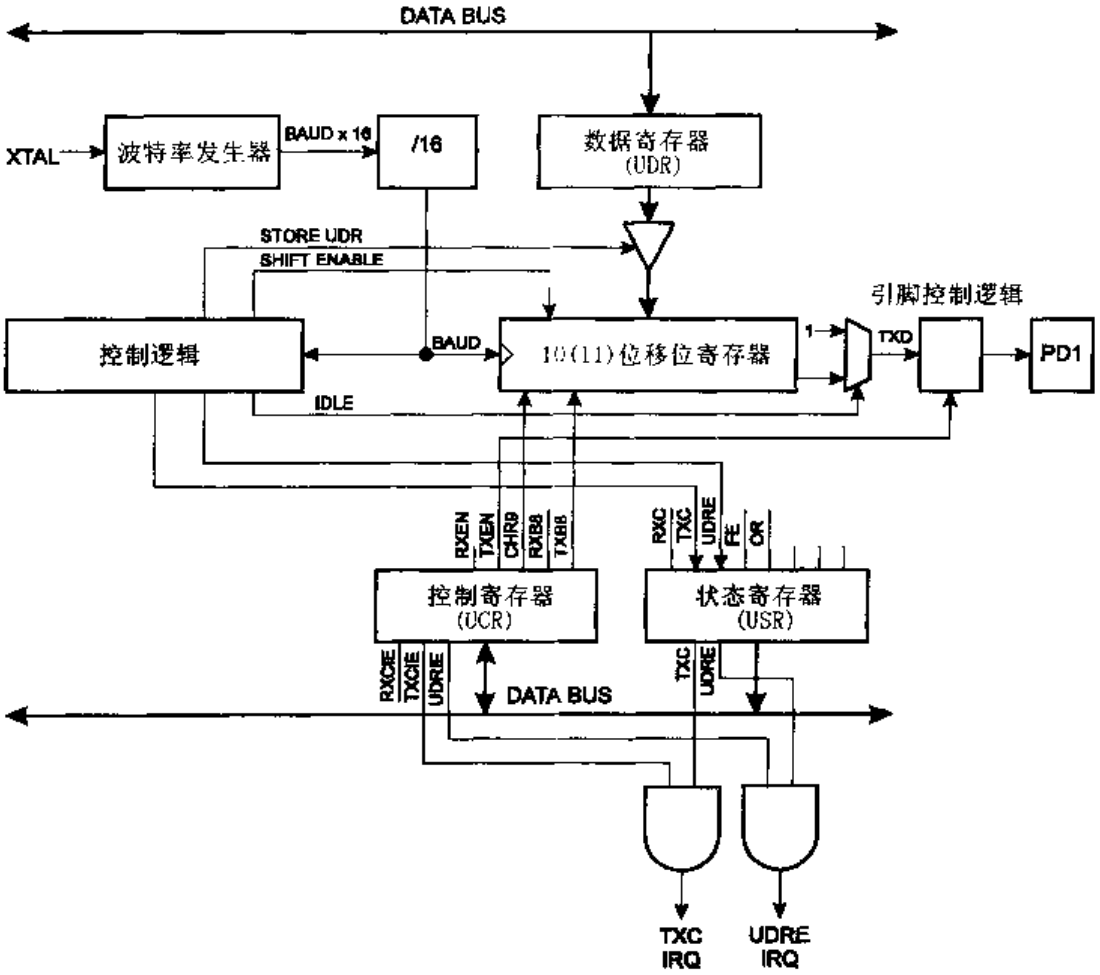


图 4.20 UART 发送器方框示意图

使能 UART 后，状态寄存器 USR 中 UDRE 位置“1”，数据从 UDR 中传送到移位寄存器后也会使 UDRE 置位。如果在前一个字节的停止位从移位寄存器移出，UDR 中没有新的数据写入，UDRE 标志位将保持为“1”，直到新的数据写入 UDR。当没有新的数据写入 UDR，而且停止位在 TXD 上保持了一位的长度，状态寄存器 USR 中 TXC 位被置位，表示 TX 传送完成。

UCR 寄存器中的 TXEN 位使能 UART 发送器。当 TXEN 为“0”时，PD1(TX)可作为普通 I/O 口。当 TXEN 为“1”时，UART 输出自动连接到 PD1，强制其为输出，而不管方向寄存器 DDRD.1 的设置。

4.9.1.2 数据接收

UART 接收器的前端以 16 倍于波特率的频率对 RXD 引脚采样。当线路空闲时(为 TTL 高电平)，一个逻辑“0”的采样将被转换为起始位的下降沿，并且初始起始位的探测序列。

跟随“1”到“0”的转换之后，接收器在第 8、9 和 10 个采样点采样 RXD 引脚。如果 3 个采样中两个或两个以上是逻辑“1”，则起始位被判定为噪声而被拒绝，接收器继续探测一个“1”到“0”的转换。图 4.21 为 UART 接收器方框示意图。

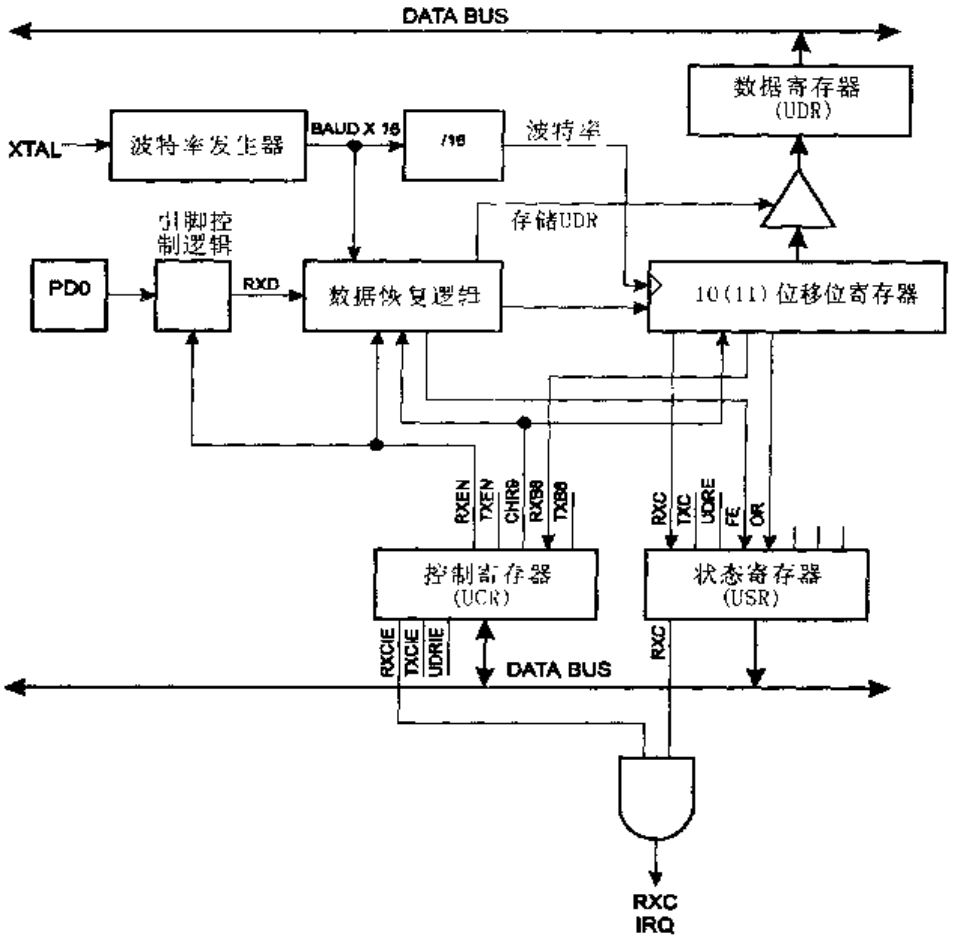


图 4.21　UART 接收器方框示意图

如果检测到一个有效的起始位，MCU 会开始数据位采样，这些位也在第 8、9 和 10 位处采样，3 取 2 作为该位的逻辑值。在采样的同时，这些位被移入移位寄存器。8 位数据的采样时序图如图 4.22 所示。

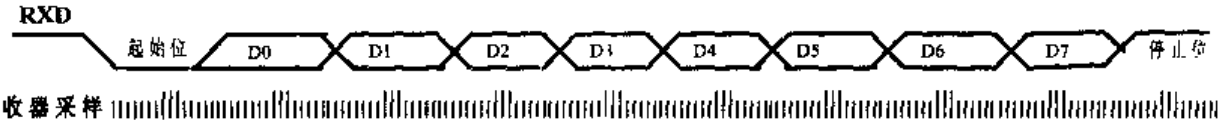


图 4.22　UART 接收器数据采样示意图

停止位到来时，3 个采样中的至少有两个为“1”接收器才认为停止位有效。但不管停止位有效与否，收到的数据都将被送入 UDR，并且置位 USR 的 RXC。但如果停止位无效，USR 中帧错误 FE 位将置“1”。

如果在读取 UDR 前，UART 又接收到一个数据，则 USR 中的过速标志位 OR 置“1”，表示最近一次移入移位寄存器的数据不能传送到 UDR 而丢失，OR 标志位也将一直保持到

UDR 被读取。9 位数据模式下，接收的第 9 位数据在 UCR 的 RXB8 位。

提示：

(1) 建议读取 UDR 前检查 FE，避免读取帧错误的 UDR;

(2) 如果 UART 速率较高或 MCU 负载比较重，建议读 UDR 前先检查 OR。

4.9.1.3 UART 寄存器

1. UART 数据寄存器—UDR，其定义如表 4.27 所示。

表 4.27 UART 数据寄存器—UDR

BIT	7	6	5	4	3	2	1	0
UDR(0x0C)	MSB							LSB
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

UDR 虽然是只有一个地址，但是实际上是物理上分离的二个寄存器，一个用于发送，另一个用于接收。读取 UDR 时，访问的是接收 UDR，而在写 UDR 时，访问的是发送 UDR。

2. UART 状态寄存器—USR，其定义如表 4.28 所示。

状态寄存器 USR 反映 UART 工作时的收发状态和异常信息，是一个只读寄存器。

表 4.28 UART 状态寄存器—USR

BIT	7	6	5	4	3	2	1	0
USR(0x0B)	RXC	TXC	UDRE	FE	OR	-	-	-
读/写	R	R	R	R	R	-	-	-
初始值	0	0	0	0	0	0	0	0

RXC：UART 接收结束。RXC 置位表示接收到的数据已经从接收移位寄存器传送到 UDR，但不管数据是否有误。如果 RXCIE 和全局中断允许位 I 都为“1”时，将引起接收结束中断，在读 UDR 时 RXC 被自动清除。如果采用中断方式接收，则中断服务程序必须读一次 UDR 以清除 RXC，否则中断结束后又会引发中断。

TXC：UART 发送结束。TXC 置位表示数据已经从发送移位寄存器发送出去，且 UDR 中没有新的要发送的数据。如果 TXCIE 已经置位，则 TXC 置位将引起发送结束中断，进入中断服务程序后 TXC 自动清零。另一种清零的方法是向 TXC 写“1”，如下：

```
USR|=0x40;
```

提示：在半双工通信应用中，TXC 标志可作为发送模式向接收模式转换的条件。

```
UDR=0x0A;                    //发送一个字符
While(!(USR & 0x40));        // 等待 TXC 置位
                             // 转入接收模式
```

```
while(!(USR & 0x80));          //等待 RXC 置位
TMPB=UDR;                      //接收一个字符
```

UDRE：UART 数据寄存器空。当数据从 UDR 传送到发送移位寄存器后，UDRE 置位，表明发送器已经准备好接收新的要发送的数据。如 UDRIE 和全局中断允许位 I 都置位，则要 UDRE 为“1”，UART 数据寄存器空中断服务程序就可以执行，写 UDR 将复位 UDRE。

如果利用中断方式发送数据，则在 UART 数据寄存器空中断服务程序里必须写 UDR 以清除 UDRE，否则中断将连续发生。

在复位时，UDRE 被设置为“1”表示已经准备好数据发送。

FE：帧异常。MCU 检测到帧错误(如停止位“0”)时 FE 置位。当停止位为“1”时，FE 复位。

OR：数据过速。如果程序读 UDR 前又有新的数据进入移位寄存器，则 OR 置位，表示数据无法转移到 URD 而丢失了，并且 OR 将一直保持到读取 UDR。

提示：如果波特率较高，或者 CPU 负载比较重，我们应该在读 UDR 前先检测 OR 标志：

```
if (!(USR&0x10))
tmpR=UDR;
```

3．UART 控制寄存器 UCR，其定义如表 4.29 所示。

UART 控制寄存器控制 UART 的收发使能和 9 位模式的第 9 位收发。

表 4.29　UAR 的控制寄存器—UCR

BIT	7	6	5	4	3	2	1	0
USR(0x0A)	RXCIE	TXCIE	UDRIE	RXEN	XEN	CHR9	RXB8	TXB8
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R	W
初始值	0	0	0	0	0	0	1	0

RXCIE/TXCIE/UDRIE：控制 UART 的三种中断使能，分别是发送、接收和发送寄存器空。

RXEN 使能 UART 接收，接收禁止将导致 USR 中的 TXC、OR 和 FE 无法置位，同时也不能复位已经置位的标志位。

TXEN 使能 UART 发送，但如果发送器当前正在发送，TXEN 清“0”不能立即关闭发送器，只有等到发送完毕后，发送器才真正关闭。

CHR9 置位使 UART 工作于 9 位模式，将发送数据的第 9 位在 TXB8 中，接收到数据的第 9 位在 RXB8 中。另外，RXB8 位是只读的，而 TXB8 是只写的。

4．波特率控制器—UBRR，其定义如表 4.30 所示。

表 4.30 UART 波特率控制器—UBRR

BIT	7	6	5	4	3	2	1	0
UBRR(0x09)	MSB							LSB
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

8 位波特率控制器 UBRR 通过对系统时钟进行分频，得到 16 倍于波特率的 UART 工作时钟。波特率的计算公式为：

$$BAUD = \frac{Fck}{16*(UBRR+1)}$$

式中，BAUD=波特率；

FCK=晶振频率；

UBRR=波特率控制器的内容。

如果使用 7.3728MHz 的晶振，需要 9600bps 的波特率，则有：

$$BAUD = \frac{Fck}{16*(UBRR+1)} = \frac{7.3728MHz}{16*(9600+1)} \approx 47$$

计算得 UBRR = 47。其他的速率可以查表或由 ICCAVR 内置的波特率和时钟计算器计算得到。

4.9.2 访问 UART 操作的 C 语言源程序及剖析

UART 有中断和查询两种工作方式，UART 初始化需要设置相关的几个控制寄存器，但是我们不用担心 PD0 和 PD1 的端口方向设置，因为在 UART 方式下，单片机将忽略 RXD 和 TXD 两个引脚的方向寄存器设置，即与 PD0 和 PD1 设置无关。使用 7.3728MHz 晶振，速率 9600bps。

1. 初始化函数

```
void uart0_init(void)          //使用 7.3728MHz 晶振，初始化速率为 9600bps
{
UCR=0x00;                      //设置波特率时要先关闭 UART
UBRR=0x2F;                     // 设置波特率为 9600bps
UCR=0xD8;                      //打开串口：8 位数据，
//使能收、发，使能接收终端、发送中断
}
```

上面的子程序在 ICCAVR 汇编后的代码：

```
_uart0_init:
clr R2
out 0xa,R2                    ; 设置波特率时要先关闭 UART
ldi R24,47
out 0x9,R24                   ; 设置波特率为 9600bps
ldi R24,216
out 0xa,R24                   ;打开串口：8 位数据，使能收、发，使能接收终端、发送中断
ret
```

2. 查询方式发送一个字节

```
void sendchar(unsigned char c)
    {
    UDR=c;
    while(!(USR & 0x40));
        USR|=0x40;
    }
```

ICCAVR 汇编后的代码,参数 c 通过 R16 传递：

```
_sendchar:
    out 0xc,R16               ;发送存在 R16 中的值
wait_txRDY:
    sbis 0xb,6                ;测试状态寄存器 USR 中 TXC 发送完成标志位
    rjmp wait_txRDY
    sbi 0xb,6                 ;通过写"1"清除 TXC 标志位
    ret
```

3. 查询方式接收一个字节

```
unsigned char  recchar(void)
    {
    while(!(USR&0x80));
    return UDR;
    }
```

ICCAVR 汇编后的代码，返回值保存在 R16 中：

```
_recchar:
    sbis 0xb,7                ;测试状态寄存器 USR 中 RXC 发送完成标志位
    rjmp _recchar
```

```
in R16,0xc                ;在 R16 中返回 UDR 中接收到的值
ret
```

4. 中断方式发送/接收

中断方式发送程序使用“TXC 发送完成”或“UDRE 发送寄存器空”中断服务程序，接收程序可以使用“RXC 接收完成”中断服务程序。需要发送的字符在 tmpSend 中，接收到的字符暂存在 tmpRecv 中。

```
unsigned char tmpSend,tmpRecv;
/*  用中断方式处理 uart 收发  */
#pragma interrupt_handler uart0_rx_isr:10
void uart0_rx_isr(void)                    //中断向量 10，UART 接收中断
     {
     tmpRecv=UDR;                          //uart has received a character in UDR
     }
#pragma interrupt_handler uart0_tx_isr:12
void uart0_tx_isr(void)                    //中断向量 10，UART 发送完成中断
     {
     UDR=tmpSend;                          //character has been transmitted
     }
```

提示：如果在发送完成中断或 UDRE 发送寄存器空中断服务程序中没有写数据寄存器 UDR，也没有清除相关标志，中断程序将会引起新的中断。

4.9.3 UART 速率自适应检测

UART 通信用途广泛，但它有一个前提是发送/接收方都设定在同样的波特率，如果使用的波特率不同，采样点发生偏移，接收方收到的将是乱码。下面介绍一种能够自动检测传输波特率的方法。

UART 接收器以设定波特率的 16 倍速率采样，并且取第 8、9 和 10 三个点的多数值(三选二)作为接收到的位。包括起始位和停止位，以回车符(0x0D)为例，传输时低位在前，高位在后，如图 4.23 所示。

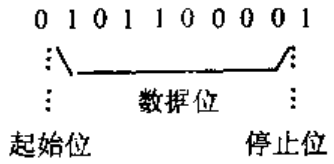


图 4.23 UART 接收时序图

串行通信中一个二进制位的传输时间(记为 T)取决于通信的波特率，9600bps 时一个二进制位的传输时间是 19200bps 时一个二进制位传输时间的两倍，即：$2*T_{19200}=T_{9600}$。因此，

9600bps 时一个位的传输时间，19200bps 时可以传输两个位。同样，9600bps 传输两个位的时间在 4800bps 时只能传送一个位。接收端设定接收波特率为 9600bps，发送端只有也以 9600bps 发送的字符，接收端才能正确地接收。发送波特率高于或低于 9600bps 都会使接收端接收到的字符发生错误。

接收端波特率固定为 9600bps，发送端以不同的波特率发送回车符时，接收方接收到的二进制序列如表 4.31 所示。

表 4.31　不同波特率下的二进制序列

波　特　率	接收到的二进制位序列	字 节 表 示
19200	0101100001111111111	0xF?
9600	0 1 0 1 1 0 0 0 0 1	0x0D
4800	0011001111000000011	0xE6
2400	0000111100001111111	0x78
1800	00000x1111x000001111	0xE0
1800	00000x1111x000001111	0xF0
1200	0000000011111110000	0x80
600	0000000000000001111	0x00
300	0000000000000000000	0x00
150	0000000000000000000	0x00
110	0000000000000000000	0x00

从上表中可以看出，除了 19200bps 和 1800bps 时两种特例情况，其他情形的二进制序列都是 9600bps 时二进制序列的变换。取前十个二进制位与 9600bps 时的二进制位相对应，并且忽略缺少停止位“1”引发的数据帧错误，把接收到的字符表示成字节方式(如上表的最右列所示)。例如：在发送速率为 1200bps，接收速率为 9600bps 时，接收端得到的字节是 0x80，而不是正确的回车符 0x0D。因为在不同的发送速率下(9600，4800，2400，1200)得到的字节不同，所以通过接收字符的判定就可以确定发送波特率。

发送波特率为 19200bps 时，其发送速度正好是接收速度(9600bps)的两倍，因此发送端的两个二进制位会被接收端看作一个。取决于不同的串行接口硬件，‘01’和‘10’这两种二进制位组合可能被认为是“1”或者“0”。幸运的是，只有 0～4 位存在这样的歧义问题，后面的位因为都是停止位，所以都是“1”。因此，发送速率为 19200bps 时接收到的字符的高半个字节为 0xF。低半个字节可能是多个值中的一个，但不会是 0x0，因为 0x0D 中有相邻的两个“1”，这就会至少在低半个字节中产生一个“1”。因此，整个字节的形式为 0xF?，且低半个字节不为 0。

当发送速率为 1800bps 时：

$T_{1800}=T_{9600}*16/3$

因为 16/3 不是整数，接收端二进制位的状态转换时刻和 9600 波特不一一对应，引起在接收端的一个位接收周期内有状态发生变化的可能。上表中给出的第六个位(表示为 x)

就是这种情况。因为 x 有可能被看作“1”，也有可能被看作“0”，所以发送速率为 1800bps 时接收到的字节可能是 0xE0 或者 0xF0。波特率为 3600bps 和 7200bps 时也有同样的问题，也可以采用同样的方法，但不确定的位数会增加，需要检测的字节种类也会更多。

当发送速率低于 1200bps 时，接收端收到的字节都是 0x00，因此只能确定其速率低于 1200bps，而不可能再得到更多的信息。为了解决这个问题，可以在 9600bps 的速率下继续接收下一个字节信息。发送速率为 600bps 或更低时，一个位的发送时间要大于 9600bps 时整个字节的接收时间。因此，发送端每一个从“1”(终止位)到“0”(起始位)的跳变都会让接收端认为一个新的字节开始了。低于 1200bps 的测量可通过测定第一个和第二个字节的接收时间差，然后根据时间差常数确定发送端的发送波特率，详细的数据这里不再列出，读者可自行分析。

通过上面的分析，各种波特率都可以通过回车符的发送和接收信息来测定。

下面给出 ICCAVR 实现的串口检测算法。函数返回值标志不同的串口速率。

```
unsigned char ComPortTest(void)
    {
    unsigned char recvByte;
    unsigned char returnByte;
    recvByte=recvchar();
    if(recvByte>=0xF1)          //19200bps
         returnByte=1;
    else if(recvByte==0x0D)     //9600bps
         returnByte=2;
    else if(recvByte==0xE6)     //4800bps
         returnByte=3;
    else if(recvByte==0x78)     //2400bps
         returnByte=4;
    else if((recvByte==0xF0)||(recvByte==0xE0))
         returnByte=5;
    else if(recvByte==0x80)     //1200bps
         returnByte=6;
    else
         returnByte=0;          //error
    return returnByte;
    }
```

4.10　访问内置的 EEPROM

4.10.1　访问单片机内置 EEPROM 操作的特点

AT90S8515 片内集成 512 字节 EEPROM，与访问 EEPROM 有关的寄存器为：EEAR、EEDR、EECR。根据系统电压的不同，EEPROM 的写时间在 2.5~4ms 之间，电源电压越低，写周期越长。

提示：写 EEPROM 时，在下一条指令被执行前 CPU 将挂起 2 个系统时钟周期，而读 EEPROM 时 CPU 将挂起 4 个系统时钟周期后执行下一条指令。

1．EEPROM 地址寄存器—EEAR，其定义如表 4.32 所示。

AT90S8515 有 512 字节的 EEPROM，在 0~511 之间线性寻址，用 9 位地址。AVR 各种单片机有不同容量的 EEPROM，为了兼容不同类型的 AVR 单片机，采用双字节地址寄存器：EEARH 和 EEARL。AT90S8515 使用 9 位地址，EEARH 的高 7 位保留。

表 4.32　EEPROM 地址寄存器—EEAR

	7	6	5	4	3	2	1	0
EEARH(0x3F)	-	-	-	-	-	-	-	EEAR8
EEARL(0x3E)	EEAR7	EEAR6	EEAR5	EEAR4	EEAR3	EEAR2	EEAR1	EEAR0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

2．EEPROM 数据寄存器—EEDR，其定义如表 4.33 所示。

表 4.33　EEPROM 数据寄存器—EEDR

	7	6	5	4	3	2	1	0
EEDR(0x3D)	EEDR7	EEDR6	EEDR5	EEDR4	EEDR3	EEDR2	EEDR1	EEDR0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

对于 EEPROM 写入操作，EEDR 寄存器包含了写入 EEPROM 的数据(表 4.33 为 EEPROM 数据寄存器—EEDR)，由 EEAR 寄存器给出其地址；对于 EEPROM 读取操作，EEDR 包含由 EEAR 给出的 EEPROM 地址，数据将从这一地址中读出。

3．EEPROM 控制寄存器—EECR，其定义如表 4.34 所示。

表 4.34 EEPROM 控制寄存器—EECR

	7	6	5	4	3	2	1	0
EECR(0x3C)	-	-	-	-	-	EEMWE	EEWE	EERE
R/W	R	R	R	R	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

EEMWE，EEPROM 读写使能：

EEMWE 位决定了设置 EEWE 为 1 是否导致 EEPROM 被写入。当 EEMWE 位为“1”时，置位 EEWE 将把数据写入 EEPROM 的指定地址；若 EEMWE 为“0”，则设置 EEWE 无效。当 EEMWE 被软件置位后 4 个时钟周期内被硬件清零。

EEWE，EEPROM 写触发：

EEPROM 写入触发信号 EEWE 是对 EEPROM 的写入使能。若地址 EEAR 和数据设置好之后，需置位 EEWE 以便将数据写入 EEPROM。当 EEWE 被置“1”时，EEMWE 必须被置“1”，否则不会发生 EEPROM 的写操作。写时序如下所示，其中第 2 和第 3 步不是必须的。

(1) 等待 EEWE 位变为“0”；

(2) 把新的 EEPROM 地址写入 EEAR(可选)；

(3) 把新的数据写入 EEDR(可选)；

(4) 在 EECR 中的 EEMWE 位写逻辑“1”；

(5) 在置位 EEMWE 位后的 4 个周期内对 EEWE 写逻辑“1”。

在写入时间(2.7V 时为 4ms 左右，5V 时为 2.5ms 左右)之后 EEWE 位被硬件清“0”。用户可以凭此位判断写时序是否已经完成，并借此判断上一个字节写入结束而开始下一个字节的写操作。注意：EEWE 置位后，CPU 在执行下一条指令前要停止 2 个周期。

注意：在写 EEPROM 时，建议关闭全局中断标志位 I。如果在步骤 4 和 5 之间响应中断将导致写操作失败。

EERE，EEPROM 读使能：

EEPROM 读取触发信号 EERE 是对 EEPROM 的读取使能。当 EEPROM 地址 EEAD 寄存器被正确设置后，需置位 EERE 以便将数据读入 EEDR 寄存器。当 EERE 被硬件清零时，表示 EEPROM 的数据已经读入 EEDR 寄存器。EEPROM 数据的读取只需要一条指令且无需等待，CPU 无需查询 EERE 位。一旦 EERE 置位后，CPU 在执行下一条指令前要停止 2 个周期。

注意：用户在读取 EEPROM 时应该检测 EEWE 位，如果一个写操作正在进行，则 EEAR 和 EEDR 将可能导致写失败。

4．防止 EEPROM 数据毁坏

由于 AVR 单片机硬件设计上的缺陷，ATMEL 公司建议 EEPROM 中地址为 0 的存储空间尽量不要使用。支持 AVR 的所有编译器，如 ICCAVR、GCCAVR、IAR 和 CVAVR 等都

不会使用地址为 0 的 EEPROM 空间，所有的变量或常量均从地址为 1 的空间开始存放，用户在使用 C 编译时可以不必考虑这个情况，但使用汇编时要注意这一点。

EEPROM 空间易丢失数据主要出现在早期的 AT90 系列芯片中，单片机内部的 POR(上电复位)电路很不可靠，如果电源供电质量不高，由于电源电压波动，特别是上电和掉电期间，会导致程序跑飞而误改了 EEPROM 中的数据。对这类芯片只要使用较高品质的电源，并且增加 MAX809 或与其兼容的电源监控芯片，则 EEPROM 中数据基本不会丢失。

近期生产 AVR 芯片的 EEPROM 数据丢失现象已大大改善，特别是对于 ATmega 系列的单片机，内部均带有 POR(上电复位)和 BOD(电源电压监测)两个电路，直接将复位端通过一个上拉电阻接正电源也不易丢失 EEPROM 数据，但使用时别忘了要使能 BOD 编程熔丝位。

由于电源电压过低，CPU 和 EEPROM 有可能工作不正常，造成 EEPROM 数据的毁坏。这种情况在使用独立的 EEPROM 器件时也会遇到，常有以下两种可能：

(1) 电源异常使 EEPROM 的数据彻底丢失；

(2) 复位不好或软件跑飞可能会导致 EEPROM 的数据被改写。

针对这两个问题，可以有以下的解决方案：

(1) 选用比 MCU 的电源范围宽并有 WP 引脚的 EEPROM 芯片。当电源较长时间低于 EEPROM 芯片的最低工作电压时非常容易丢失全部数据；或者 MCU 还能工作，但外置 EEPROM 芯片已不能工作时，EEPROM 中的数据会全部丢失。可以让电源电压过低时使 AVR 处于掉电休眠状态，这也可以防止 CPU 对 EEPROM 的误操作。

(2) 做好电源滤波，要使用高品质的电源，而且要等电源开机稳定后才去读写 EEPROM。

(3) 做好 MCU 复位电路，特别是使用 AVR 单片机内置的 EEPROM，复位不好，非常容易丢失数据，特别是地址 0 中的数据，建议早期生产的 AT90 系列单片机均应加上 MAX809 或与其兼容的电源监控芯片。

(4) 做好软件跑飞的处理。

(5) SDA 和 SCK 的上拉最好用 I/O 口控制，既可省电，也可在一定情况下保护 EEPROM。

(6) 外接 EEPROM 芯片的 WP 接 MCU 的 RESET。由于 AVR 单片机复位时为高阻态，可用 AVR 的 I/O 口接 WP 做软件保护，只在读写 EEPROM 时才给器件供电，不仅能提高可靠性，而且能省电。但在应用时应注意，EEPROM 芯片给电后需要有大于写周期的延时才能读写。

(7) EEPROM 空间富余时考虑双备份或多备份数据，且每份数据都应有校验。

(8) 将那些不需修改的常数存储于 FLASH 之中。

4.10.2　访问内置 EEPROM 操作的 C 源程序实例及剖析

下面给出读/写 AVR 单片机内置的 EEPROM 操作的例程，用 ICCAVR 读取指定地址的 EEPROM 数据，函数需要两个参数：一个 16 位地址和一个 8 位的字节数据。

```
void EEPROM_rite(unsigned int uAddr,unsigned char uData)
    {
    while(EECR & (1<<EEWE));    //等待前一次写操作完成
    EEAR=uAddr;                 //设置地址到 EEAR 寄存器
    EEDR=uData;                 //写数据到 EEDR 寄存器
    EECR|=(1<<EEMWE);           //写 EEMWE 位，允许 EEPROM 操作
    EECR|=(1<<EEWE);            //写 EEWE 位，开始 EEPROM 写操作
    }
```

ICCAVR 编译后的汇编代码如下，16 位地址传递到 r16,r17，8 位数据传递到 r18：

```
EEPROM_rite:
    sbic 0x1c,1;                等待前一次写操作完成
    rjmp EEPROM_rite
    sts EEARH,R17;              设置地址到 EEAR 寄存器
    sts EEARL,R16
    out 0x1d,R18;               写数据到 EEDR 寄存器
    sbi 0x1c,2;                 写 EEMWE 位，允许 EEPROM 操作
    sbi 0x1c,1;                 写 EEWE 位，开始 EEPROM 写操作
    ret
```

读取 EEPROM 中指定地址的数据只需要给出 EEPROM 地址，函数返回字节数据：

```
unsigned char EEPROM_read(unsigned int uAddr)
    {
    while(EECR & (1<<EEWE));    //等待前一次写操作完成
    EEAR=uAddr;                 //设置地址到 EEAR 寄存器
    EECR|=(1<<EERE);            //写 EEWE 位，开始 EEPROM 写操作
    return EEDR;
    }
```

ICCAVR 编译后的汇编代码如下，其中 16 位地址传递到 r16、r17，返回的 8 位数据传递到 r16。

```
EEPROM_read::
    sbic 0x1c,1;                等待前一次写操作完成
    rjmp EEPROM_read
    sts EEARH,R17;              设置地址到 EEAR 寄存器
    sts EEARL,R16
    sbi 0x1c,0;                 写 EERE 位，开始 EEPROM 读操作
    in R16,0x1d
```

```
    ret
```

上面的 EEPROM 读/写函数可在 C 中直接调用，方式如下：

```
void main(void)
    {
    unsigned char tEED;
    //用户程序
    tEED=EEPROM_read(0x0021);
    tEED++;
    EEPROM_rite(0x0021,tEED);
    //用户程序
    }
```

另外，ICCAVR 也以库函数的形式提供了访问 EEPROM 的函数，在使用之前要包含 eeprom.h 头文件：

```
#include "eeprom.h"
```

在 eeprom.h 文件定义了几个 eeprom 操作函数。

(1) 访问容量 256 字节或更少 EEPROM 的 AVR 单片机：

```
int _256EEPROMwrite( int location, unsigned char);
unsigned char _256EEPROMread( int);
```

(2) 访问容量大于 256 字节 EEPROM 的 AVR 单片机：

```
int EEPROMwrite( int location, unsigned char);
unsigned char EEPROMread( int);
```

(3) ICCAVR 还提供了两个比较通用的函数和宏，这些宏适用于任何 AVR 单片机器件。

```
void EEPROMReadBytes(int addr, void *ptr, int size);
void EEPROMWriteBytes(int addr, void *ptr, int size);
#define EEPROM_READ(addr, dst) EEPROMReadBytes(addr, &dst, sizeof (dst))
#define EEPROM_WRITE(addr, src) EEPROMWriteBytes(addr, &src, sizeof (src))
```

因此，在程序中可以直接利用下面的函数读写 EEPROM。

```
EEPROM_READ(int location,object)
EEPROM_WRITE(int location,object)
```

其中“object”可以是任意程序变量（包括结构和数组）。

例如：

```
void EEPROM_test(void)
     {
     int i;
     EEPROM_READ(0x0001,i);          //读取 2 个字节给 I
     EEPROM_WRITE(0x0001,i);         //写 2 个字节到 0x0001
     }
```

4.10.3 初始化内置的 EEPROM 数据

EEPROM 数据也可以在程序源文件中初始化，在 C 源文件中，它作为一个全局变量被分配到特殊调用区域“EEPROM”中。这是可以用伪指令实现的，其结果是产生扩展名为.eep 的输出文件。如：

```
struct WorkFrame
     {
     unsigned char Status;
     unsigned int  Length;
     unsigned char Sort;
     unsigned char Datas[100];
     };
#pragma data:eeprom
int uiData=0x1234;
struct WorkFrame tmFrame;
char table[]={0,1,2,3,4,5};
#pragma data:data
void main (void)
     {
     int I;
     struct WorkFrame wFrame;
     //…
     EEPROM_READ((int)&uiData, I);      //读出 uiData 到 I,I=0x1234
     EEPROM_READ((int)&tmFrame, wFrame);
                                        //读出结构体 tmFrame 中的数据到结构体
                                          wFrame 中
     }
```

4.11　访问同步串行接口 SPI

4.11.1　访问 SPI 操作的特点

同步串行接口(SPI)能使 AVR 单片机和外设或其他 AVR 单片机在不长的距离内进行高速同步数据传输。

AT90S8515 的 SPI 采用全双工 3 线同步模式，最高可达 5Mb/S 的速率，并且可编程为主机或从机。图 4.24 为 AT90S8515 的 SPI 结构示意图，图 4.25 所示为主从机之间的连接方法。

主机和从机的两个移位寄存器可以被认为是一个分开的 16 位环行移位寄存器，当数据从主机移向从机，同时数据也移向相反的方向。这就意味着在一个移位周期内，主机和从机数据进行了交换。在一个字节从主机移出后，传送停止标志位(SPIF)将被设置。如果此时 SPCR(SPI 控制寄存器)中的中断允许位 SPIE 为“1”，将生成一个中断请求；如全局中断允许位 I 也为“1”，则 SPI 中断服务程序会被执行。

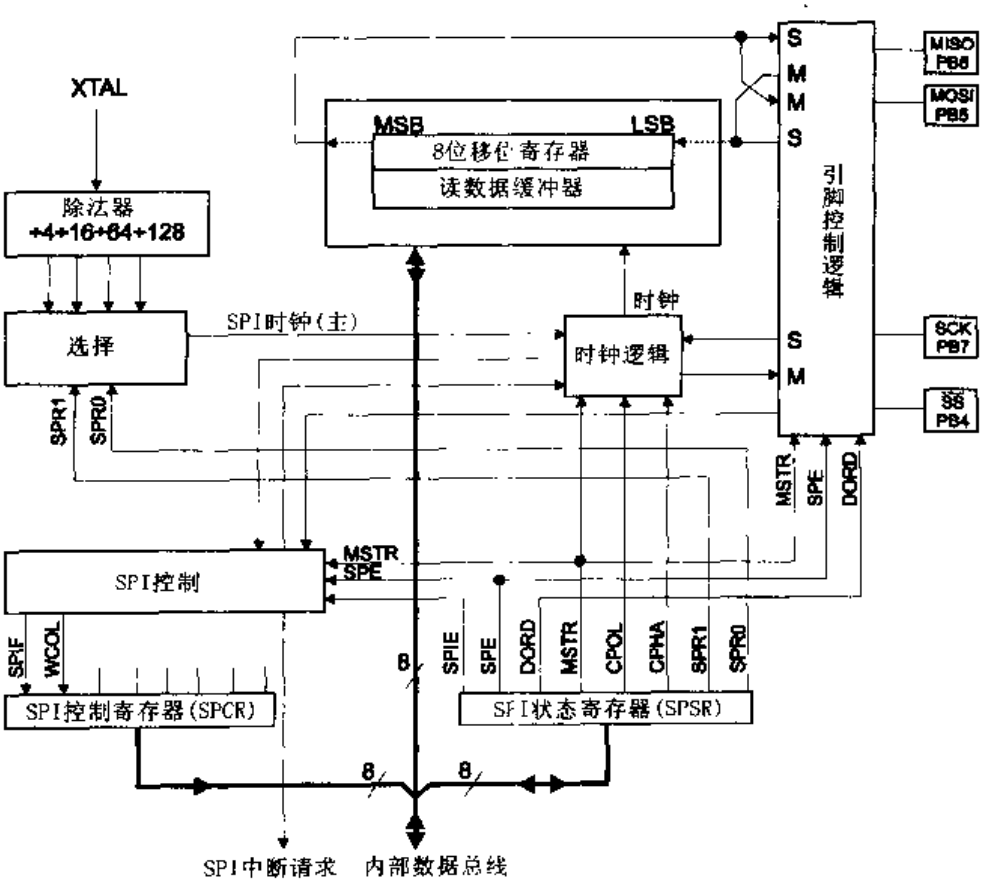


图 4.24　SPI 原理图

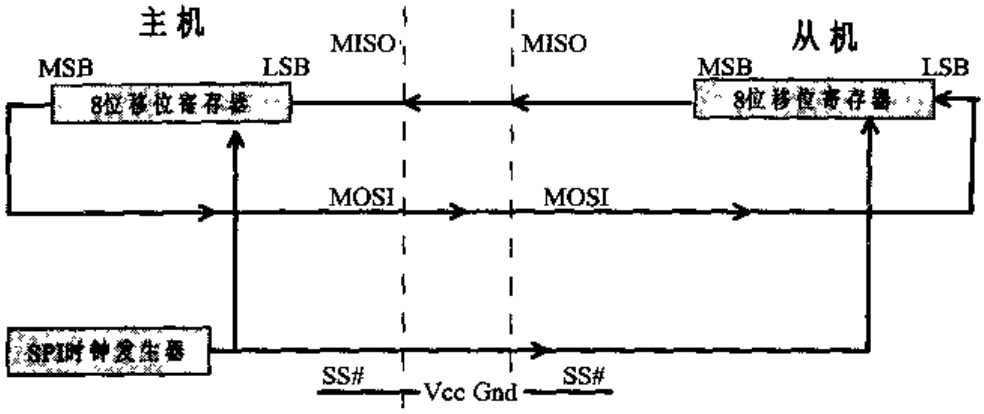


图 4.25 SPI 主从机连接

当 SPI 工作于主机模式时，SS#引脚(#表示低电平有效，等同于 SS)可由用户自己设置。如果设置为输入，在使用 SPI 发送数据前应检查 SS#引脚电平和 SPI 控制寄存器 SPCR 中的 MSTR 位，SS#引脚为输出时对 SPI 主机无影响。当 SPI 工作于从机模式时，SS#引脚为输入，低电平时使能从机 SPI。

根据 SPI 串行时钟 SCR 相位和极性，SPI 有四种工作模式，这由 SPCR 中的控制位 CPHA 时钟相位和 CPOL 时钟极性来决定，SPI 工作模式选择如表 4.35 所示，SPI 四种工作模式的时序如图 4.26 所示。

表 4.35 SPI 工作模式选择

CPOL	CPHA	模 式
0	0	0
0	1	1
1	0	2
1	1	3

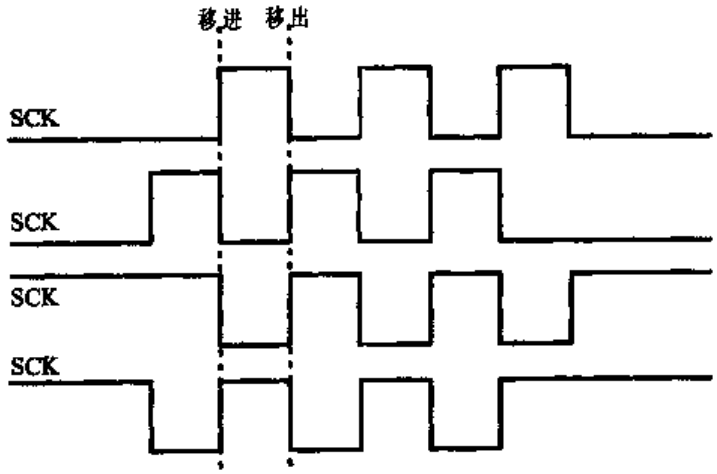


图 4.26 SPI 四种工作模式的时序

1. SPI 数据寄存器—SPDR

SPI 数据寄存器可以读/写，用于在寄存器和 SPI 移位寄存器之间传送数据。写入该寄存器时，初始化数据传送；读 SPDR 时，读到的是移位寄存器接收缓冲区的值。

2．SPI 控制寄存器—SPCR，其定义如表 4.36 所示。

表 4.36　SPI 控制寄存器—SPCR

BIT	7	6	5	4	3	2	1	0
SPCR(0x0D)	SPIE	SPE	DORD	MSTR	CPOL	CPHAP	SPR1	SPR0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

SPIE，SPI 中断允许。SPIE 置位将使 SPI 中断事件发生时置位 SPSR 中的 SPIF 中断标志位。

SPE，开始允许 SPI 工作。

DORD，SPI 数据发送/接收的顺序。DORD=1，发送/接收时数据的 LSB(低位)在前；DORD=0 则 MSB 高位在前。

MSTR，主从机选择。MSTR=1 为主机模式，而 MSTR=0 则为从机模式。

提示：若 SS# 为输入且在 MSTR=1 时被置低，则 MSTR 将被清零。同时 SPSR 中 SPIF 将被置位，从而触发中断服务程序。这时，MSTR 必须重新置位以激活 SPI 主机模式。

SPR1:SPR0，SPI 时钟速率选择位。这两位控制主机模式下 SPI 的速率，即 SCK 的时钟速率，SPR1:SPR0 对从机没有影响。SCK 和系统振荡频率 f_{ck} 之间关系如表 4.37 所示。

表 4.37　SCK 和振荡器频率之间的关系

SPR1	SPR0	SCK 频率	SPR1	SPR0	SCK 频率
0	0	$F_{CK}/4$	1	0	$F_{CK}/64$
0	1	$F_{CK}/16$	1	1	$F_{CK}/128$

3、SPI 状态寄存器—SPSR，其定义如表 4.38 所示。

表 4.38　SPI 状态寄存器—SPSR

	7	6	5	4	3	2	1	0
SPSR(0x0E)	SPIF	WCOL	-	-	-	-	-	-
R/W	R/W	R	R	R	R	R	R	R
初始值	0	0	0	0	0	0	0	0

SPIF，SPI 中断标志位。当数据传送完成时，SPIF 位被置位，若 SPCR 中 SPIE 和全局中断允许位 I 都为“1”，则 SPI 中断服务程序被触发。中断服务程序执行时硬件会自动把 SPIF 清零。另一种方法是如 SPIF 为“1”，在读取 SPSR 时 SPIF 位会被清零，然后读出 SPDR 数据寄存器的值。

WCOL，写冲突标志位。如果在数据传送中 SPI 数据寄存器被写入，则写冲突标志位

被置位。

提示：在数据传送过程中，SPDR 寄存器读出/写入的结果无效，所以在 WCOL 为“1”时，先读一次 SPSR，以清除 WCOL，再读/写 SPDR。

4.11.2 访问 SPI 操作 C 源程序实例及剖析

1. 初始化 SPI

AT90S8515 使用 4MHz 晶振，SPI 工作于模式 0，主设备，字节 MSB(最高位)在前，f_{ck}/16 时钟：

```
void SPI_MasterInit(void)
{
DDRB=0xB0;  // SPI 端口初始化：CS、SCK 和 MOSI 输出；MISO 输入。
            // SCK, MISO, MOSI,  CS
            // PB7, PB6,  PB5,  PB4, PB3, PB2 , PB1,  PB0
            //  O    I     O     O    I    I     I     O
            //  1    0     1     1    0    0     0     0
SPCR=0x51;  // SPI 控制寄存器初始化：SPI 允许，主模式，fck/16
            // SPIE, SPE , DORD, MSTR, CPOL, CPHA, SPR1, SPR0
            //   0    1     0     1     0     0     0     1
}
```

用 ICCAVR 编译后汇编代码为：

```
_spi_init:
    ldi R24,176 ; SPI 端口初始化：CS、SCK 和 MOSI 输出；MISO 输入
    out 0x17,R24
    ldi R24,80 ; SPI 控制寄存器初始化：SPI 允许，主模式，fck/16
    out 0xd,R24
    ret
```

另一片 AT90S8515 工作在从模式下，其他条件同主模式，但需要改变端口方向定义和 MSTR 模式选择位：

```
void SPI_SlaveInit(void)
   {
   DDRB=0x40;       // SPI 端口初始化：CS、SCK 和 MOSI 输入；MISO 输出。
                    // SCK, MISO, MOSI,  CS
                    // PB7, PB6,  PB5,  PB4, PB3, PB2 , PB1,  PB0
                    //  O    I     O     O    I    I     I     O
```

```
                    //  0     1     0     0     0     0     0     0
SPCR=0x41;          // SPI 控制寄存器初始化：SPI 允许，主模式，fck/16
                    // SPIE, SPE , DORD, MSTR, CPOL, CPHA, SPR1, SPR0
                    //  0     1     0     0     0     0     0     1
}
```

2. 使用 SPI

如果把 SPI 看作主从设备间一个 16 位循环移位器，主模式下发送一个字节和读取一个字节的 SPI 的操作可统一为：

```
unsigned char SPI_TranByte(unsigned char cData)
    {
    SPDR=cData;
    while(!(SPSR&0x80));          //如果 SPIF=1，表示 SPI 传输(一个字节)操作完成
    return SPDR;
    }
```

用 ICCAVR 编译后汇编代码为：

```
_SPI_TranByte:
    out 0xf,R16
wait_SPI:
    sbis 0xe,7
    rjmp wait_SPI
    in R16,0xf
    ret
```

从模式下 AT90S8515 不能主动发送/接收数据，可采用查询式或中断式发送/接收数据。查询方式接收一个字节。

```
unsigned char SPI_RecvByte(void)
    {
    while(!(SPSR&0x80);           //如果 SPIF=1，表示 SPI 传输(一个字节)操作完成
    retuen SPDR;
    }
```

中断方式发送一个字节，设变量已经存放在字节变量 tmp 中：

```
#pragma interrupt_handler spi_stc_isr:9    //声明中断向量
void spi_stc_isr(void)                     //SPDR 中的数据已经传输完成
    {
```

```
    SPDR=tmp;                          //新的数据送入 SPDR,等待主设备读取
    }
```

使用中断方式，可以设计一个 FIFO 缓冲器，在中断服务程序中读/写 FIFO。下面分别定义 100 字节的接收和发送 FOFI 缓冲器及其指针，在 SPI 中断服务程序中先从 SPDR 读一个字节放在接收 FIFO 中，然后再从发送 FIFO 中读一个字节写到 SPDR，最后移动指针并检查 FIFO 边界。

```
unsigned char SPI_RecvFifo[100];           //定义 100 字节的接收 FIFO
unsigned char SPI_TranFifo[100];           //定义 100 字节的发送 FIFO
unsigned char * pin_SPI_RecvFifo;          //定义接收/发送 FIFO 的进/出指针
unsigned char * pout_SPI_RecvFifo;
unsigned char * pin_SPI_TranFifo;
unsigned char * pout_SPI_TranFifo;

#pragma interrupt_handler spi_stc_isr:9    //声明中断向量
void spi_stc_isr(void)                     //SPDR 中的数据已经传输完成
    {
    (*pin_SPI_RecvFifo)=SPDR;              //读取一个字节
    SPDR=(* pout_SPI_TranFifo);            //新的数据送入 SPDR，等待主设备读取
    pin_SPI_RecvFifo++;                    //指针指向下一个字节
    pout_SPI_TranFifo++;
    if(pin_SPI_RecvFifo=SPI_RecvFifo +100)
                                           //检查指针是否指向最后一个字节
    pin_SPI_RecvFifo=SPI_RecvFifo;
    if(pout_SPI_TranFifo=SPI_TranFifo +100)
                                           //检查指针是否指向最后一个字节
    pout_SPI_TranFifo=SPI_TranFifo;
    }
```

提示：FIFO 缓冲器及其指针应该定义为全局变量。

下面给出完整的程序清单，在 ICCAVR6.26C 环境下编译通过：

```
#include <io8515v.h>
#include <macros.h>
#include <eeprom.h>
unsigned char SPI_RecvFifo[100];           //定义 100 字节的接收 FIFO
unsigned char SPI_TranFifo[100];           //定义 100 字节的发送 FIFO
unsigned char *pin_SPI_RecvFifo;           //定义接收/发送 FIFO 的进/出指针
unsigned char *pout_SPI_RecvFifo;
unsigned char *pin_SPI_TranFifo;
```

```
unsigned char *pout_SPI_TranFifo;
void SPI_SlaveInit()
    {
    DDRB=0x40;
    SPCR=0x41;
    }
#pragma interrupt_handler spi_stc_isr:9      //声明中断向量
void spi_stc_isr(void)                       //SPDR中的数据已经传输完成
    {
    (* pin_SPI_RecvFifo)=SPDR;               //读取一个字节
    SPDR=(* pout_SPI_TranFifo);              //新的数据送入SPDR，等待主设备读取
    pin_SPI_RecvFifo++;                      //指针指向下一个字节
    pout_SPI_TranFifo++;
    if(pin_SPI_RecvFifo==(SPI_RecvFifo +100))
                                             //检查指针是否指向最后一个字节
    pin_SPI_RecvFifo=SPI_RecvFifo;
    if(pout_SPI_TranFifo==(SPI_TrarFifo +100))
                                             //检查指针是否指向最后一个字节
    pout_SPI_TranFifo=SPI_TranFifo;
    }
void main(void)
    {
    SPI_SlaveInit();
    for(;;)                                  //等待SPI接收中断
        ;
    }
```

4.11.3　使用 DataFlash 存储器

FLASH存储器具有掉电数据不丢失、数据存取速度快、电可擦除、在线可编程、容量大、价格低廉以及足够多的擦写次数和较高的可靠性等诸多优点，因而已成为新一代嵌入式应用(如数码照相机和MP3)的首选存储器。SPI接口的FLASH使用较少的I/O线(4~5根)，下面将简单介绍ATmel公司带SPI接口的AT45DB161 FLASH芯片。

1. AT45DB161组织结构

ATmel公司生产的串行SPI接口的DataFlash AT45系列存储器容量已达到64Mbits，本节以16Mbits的AT45DB161为例，介绍用AVR8515的SPI接口存取大容量的FLASH的方法。AT45DB161为5V供电，SPI数据接口，其引脚结构如图4.27所示，下面给出SOIC28封装形式的主要管脚排列和说明。

引脚名称	功能
CS#	片选，低有效
SCK	SPI 时钟
SI	SPI 输入
SO	SPI 输出
WP#	硬件写保护，低有效
RESET#	芯片复位，低有效
RDY/BUSY#	芯片准备好/忙指示

表示低电平有效

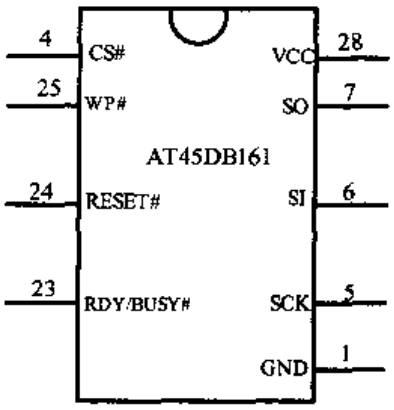


图 4.27 AT45DB161 结构图

提示：WP#写保护意味着当引脚为低电平时，对 FLASH 的前 256 页读/写操作无效。AT45DB161 的内部结构如图 4.28 所示，它有两个 528 字节 SRAM 的缓冲器。

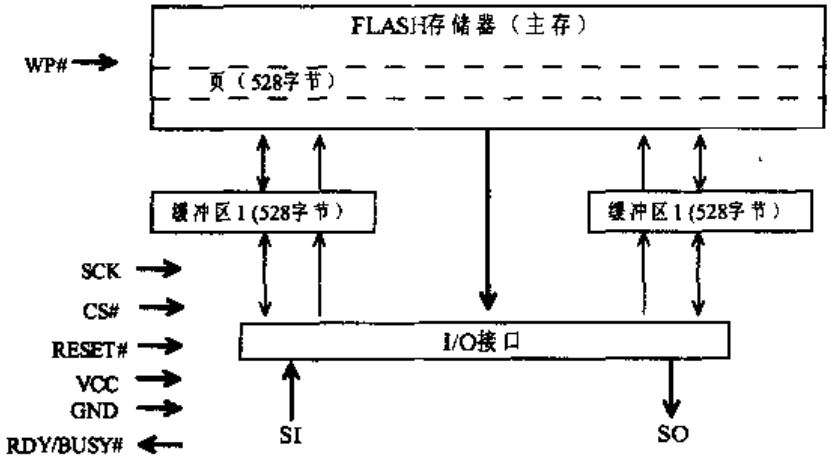


图 4.28 使用 ST16C550 扩展串口

当使用其中一个缓冲器编程主存时(即 FLASH MEMRY)，另一个缓冲器可同时用于接收要写入的数据或读出数据，实现“虚拟连续写”或 READ-WHILE-WRITE 操作，这就相对提高了 FLASH 的访问效率。

AT45DB161 主存按三级模式组织：节(SECTION)、块(BLOCK)、页(PAGE)。它的 17,301,504 位被分为 16 节，每节有 32 块，每块有 8 页，每页 528 字节，即共计 4096 页，2M 字节。

2. AT45DB161 指令，指令如表 4.39 所示。

表 4.39 AT45DB161 指令表

操 作	命 令	保 留 位	页 地 址	页内/缓冲偏移地址	
读主存	0x52	rr	PA11-PA0	BF9-BF0	32 个 x
读 Buffer1	0x54	xx	12 个 x	BFA9-BFA0	8 个 x
读 Buffer2	0x56	xx	12 个 x	BFA9-BFA0	8 个 x

续表

操　作	命　令	保 留 位	页 地 址	页内/缓冲偏移地址
主存传送到 Buffer1	0x53	rr	PA11-PA0	10 个 x
主存传送到 Buffer2	0x55	rr	PA11-PA0	10 个 x
主存与 Buffer1 比较	0x60	rr	PA11-PA0	10 个 x
主存与 Buffer2 比较	0x61	rr	PA11-PA0	10 个 x
写 Buffer1	0x84	xx	12 个 x	BFA9-BFA0
写 Buffer2	0x87	xx	12 个 x	BFA9-BFA0
带擦除 Buffer1 到主存	0x83	rr	PA11-PA0	10 个 x
带擦除 Buffer2 到主存	0x86	rr	PA11-PA0	10 个 x
不带擦除 Buffer1 到主存	0x88	rr	PA11-PA0	10 个 x
不带擦除 Buffer2 到主存	0x89	rr	PA11-PA0	10 个 x
主存页擦除	0x81	rr	PA11-PA0	10 个 x
主存块擦除	0x50	rr	PA11-PA3,xxx	10 个 x
Buffer1 为缓冲写主存	0x82	rr	PA11-PA0	BF9-BF0
Buffer2 为缓冲写主存	0x85	rr	PA11-PA0	BF9-BF0
Buffer1 为缓冲自动重写主存	0x56	rr	PA11-PA0	10 个 x
Buffer2 为缓冲自动重写主存	0x59	rr	PA11-PA0	10 个 x
读状态寄存器	0x57			

包括“读状态寄存器”指令在内，AT45DB161 共有 20 条指令，为了适应不同容量的 FLASH，这些指令中包含不同数量的填充位。读主存的指令最长，达 64 位，读缓冲的指令占 40 位，而其他(除读状态寄存器位外)占 32 位。AT45DB161 读写操作命令序列如图 4.29 所示(读状态寄存器指令除外)，传输时 MSB 在前，AT45DB161 的操作指令如表 4.39 所示。

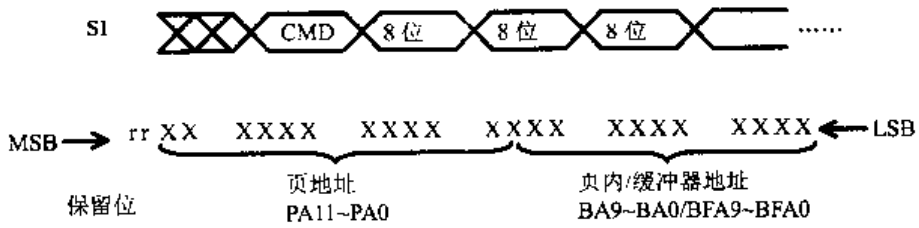


图 4.29　AT45DB161 读写操作命令序列

其中：

r 为保留位，建议以“0”填充；

x 为填充位，可以为“0”或“1”。

PA11-PA0，用来指定页地址，对 AT45DB161 来说，一共有 4096 页，占 12 位；

BA9-BA0，用来指定页内字节的地址，每页 528 字节，占 9 位；

BFA9-BFA0，用来指定缓冲区内字节地址，缓冲区为 528 字节，占 9 位。

AT45DB161 支持 SPI 的模式 0 和模式 3，本节主要采用模式 3 进行说明：上升沿移进

FLASH，下降沿移出 FLASH。读主存的指令有 64 位(8 字节)，下面仅以读主存指令为例进行时序说明，其指令格式如图 4.30 所示。

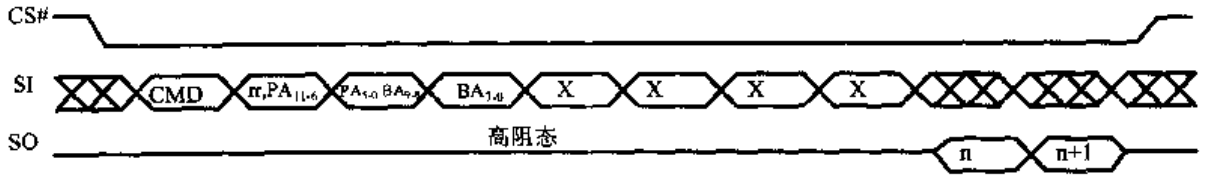


图 4.30 AT45DB161 读主存指令格式

其中：

r 为保留位，建议以“0”填充

x 为填充位，建议为“01”序列

n 为读到的第一个字节

n+1 为读的第二个字节

发送完 64 位主存指令，FLASH 内指定地址的数据会在 SCK 信号下降沿移出。每读出一个字节，字节地址自动加 1，如果已经到该页的末尾，只要有 SCK，页内地址会重新指向该页的起始处。在整个操作过程中，CS#脚始终有效(低电平)。CS#脚由“0”到“1”的跳变将会使本次操作终止，SO 脚恢复为高阻态。

读取状态寄存器的指令不需要地址信息，发送 0x57 指令后，状态寄存器值就会在 SCK 下降沿移出，状态寄存器的 MSB 在前，其指令格式如图 4.31 所示。

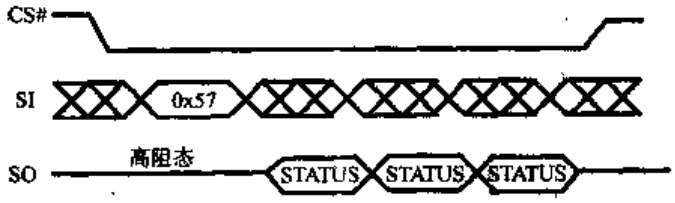


图 4.31 AT45DB161 读状态寄存器指令格式

但如果 CS#有效，只要有 SCK 工作，状态寄存器会在 SCK 作用下连续地移出，图中表示状态寄存器值连续移出 3 次。

3．AT45DB161 操作

AT45DB161 与 AT90S8515 采用四线 SPI 方式：CS#、SCK、SI、SO。另外还需要 WP#、RESET#和 RDY/BUSY#三条控制线。

我们将从 UART 接收的数据存放在 AT45DB161 中，还可以将存在 AT45DB161 中的数据读出送到 UART，图 4.32 为两者的连接示意图。

AT45DB161 内部集成上电复位电路，上电后会自动复位。如果不需要外部复位操作，它的 RESET#引脚需要上拉一个 10K 的电阻。低电平在引脚 RESET#上保持 10μs，将触发 AT45DB161 复位。外部复位会终止它正在进行的工作，并且使内部状态机返回到空闲状态，被外部复位终止的工作会在复位后(RESET#引脚恢复为高电平)继续进行。

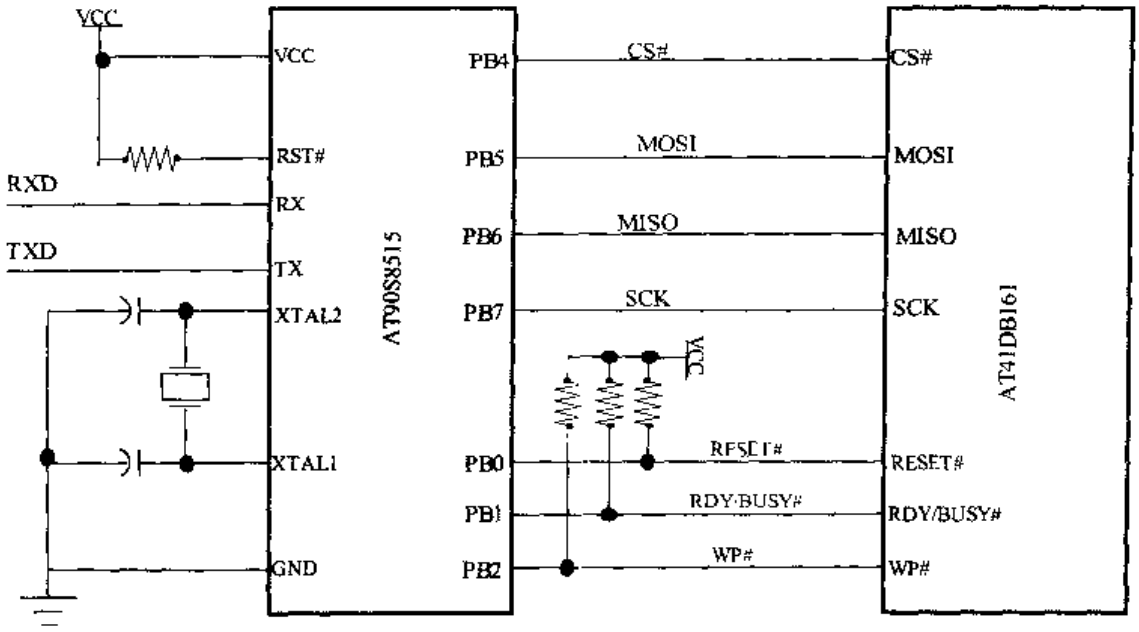


图 4.32　AT45DB161 与 AT900S8515 的连接

提示：如果 RESET#引脚一直为低电平，芯片将一直处于复位状态。

利用 4.5 节的 delay 延时函数，实现 at45db161 外部复位操作

```
void df_reset(void)
    {
    PORTB &=~0x01;                    //置 RESET#引脚为低电平
    delay(10);                        //延时(10+2)μs
    PORTB |=0x01;                     //置 RESET#引脚为高电平
    }
```

由 AT45DB161 指令表可以看出，有关 buffer1 和 buffer2 的操作是对偶的，区别在于不同的指令。这样，对两个不同 buffer 的操作函数可统一为包含一个指令参数的函数。写 buffer 的函数需要三个参数：命令字(选择 buffer)、buffer 偏移地址、要写入的字节数据。

```
    void  rite_buffer(unsigned char riteCMD,unsigned int buffer_offset,
unsigned char df_data)
        {
        while(!(PINB & 0x02));
        PORTB &=~0x10;                                  //使能 AT45DB161
        SPI_TranByte(riteCMD);                          //写 Buffer 指令,0x84 选 buffer1,
                                                          0x87 选 buffer2
        SPI_TranByte(0x00);                             //无关位
        SPI_TranByte((char)(buffer_offset>>8));//buffer 偏移地址 BFA9-BFA8
        SPI_TranByte((char)buffer_offset);              //buffer 偏移地址 BFA7-BFA0
```

```
        SPI_TranByte(df_data);                    //写字节数据
        PORTB|=0x10;
        }
```

我们来分析一下写 buffer 的时序，CS#有效时，N+1 个字节可以连续写入 dataflash，而不必写一个字节发送一次写 buffer 指令，指令时序如图 4.33 所示。

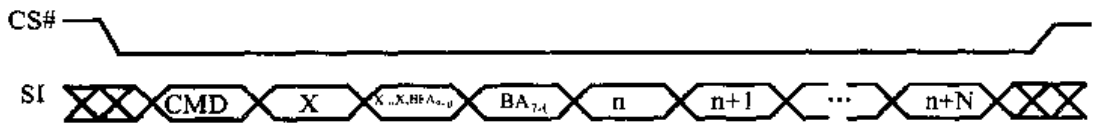


图 4.33 AT45DB161 写 buffer 的时序

图中：

(1) x 为无关位；

(2) BFA9-0 为 buffer 偏移地址；

(3) n 为写入的第一个字节，n+1 为写入的第二个字节，n+N 为写入的第 N+1 个字节。

我们重写上面写 buffer 的函数：

```
    void  rite_buffer(unsigned char riteCMD,unsigned int buffer_offset,unsigned
char * df_data,unsigned int length)
        {
        while(!(PINB & 0x02));     //等待 AT45DB161 空闲
        PORTB &=~0x10;             //使能 AT45DB161
        SPI_TranByte(riteCMD);     //写 buffer 指令,0x84 选 buffer1,0x87 选 buffer2
        SPI_TranByte(0x00);        //无关位
        SPI_TranByte((char)(buffer_offset>>8));    //buffer 偏移地址 BFA9-BFA8
        SPI_TranByte((char)buffer_offset);         //buffer 偏移地址 BFA7-BFA0
        while(length)
            {
            SPI_TranByte(* df_data);               //写字节数据
            length--;
            df_data ++;                            //指针移动
            }
        PORTB|=0x10;
        }
```

如果其中一个 buffer 已经写满数据，就可以使用“带擦除 buffer 写到主存”指令(buffer1:0x83/buffer2:0x86)把数据写到主存中了。

```
    void buffer_rtie_flash(unsigned char riteCMD,unsigned int page_counter)
       {
```

```
    while(!(PINB & 0x02));              //等待 AT45DB161 空闲
    PORTB &=~0x10;                      //使能 AT45DB161
    SPI_TranByte(riteCMD);              //带擦除 Buffer 写到主存，0x83 选 buffer1
          //0x86 选 buffer2；主存传送到 buffer，0x53 选 buffer1，0x55 选 buffer2
    SPI_TranByte((char)(page_counter>>6));  //页地址 PA11~PA6
    SPI_TranByte((char)(page_counter<<2));  //页地址 PA5-PA0
    SPI_TranByte(0x00);                     //无关位
    PORTB|=0x10;
    }
```

“主存传送到 buffer”操作与“带擦除 buffer 写到主存”操作的区别仅在于操作指令。相对应的，我们给出“读 Buffer”的操作函数：

```
    unsigned char  read_buffer(unsigned char riteCMD,unsigned int
buffer_offset)
    {
    while(!(PINB & 0x02));
    PORTB &=~0x10;                              //使能 AT45DB161
    SPI_TranByte(riteCMD);                      //写 Buffer 指令，0x84 选 buffer1，
                                                  0x87 选 buffer2
    SPI_TranByte(0x00);                         //无关位
    SPI_TranByte((char)(buffer_offset>>8));     //buffer 偏移地址 BFA9-BFA8
    SPI_TranByte((char)buffer_offset);          //buffer 偏移地址 BFA7-BFA0
    SPI_TranByte(0x00);                         //无关位
    SPI_TranByte(0xFF);                         //写字节数据，使字节数据从 SO 移出
    PORTB|=0x10;
    return SPDR;
    }
```

页擦除使用 12 位页地址，块擦除使用 12 位页地址的高 9 位，他们的操作可以统一为：

```
    void page_erasing(unsigned char riteCMD,unsigned int page_counter)
    {
    while(!(PINB & 0x02));
    PORTB &=~0x10;                          //使能 AT45DB161
    SPI_TranByte(riteCMD);                  //0x81 擦除页，0x50 擦除块
    SPI_TranByte((char)(page_counter>>6));  //页地址 PA11-PA6
    SPI_TranByte((char)(page_counter<<2));  //页地址 PA5-PA0
    SPI_TranByte(0x00);                     //无关位
    PORTB|=0x10;
    }
```

为了实现随机访问 FLASH 的内容，我们也可以直接对主存进行读写操作。直接对主存的读操作前面已经给出时序图，需要给出字节在主存中的确定位置，即页号和页内偏移。直接对主存写的操作实际上是以一个 buffer 为缓冲进行的，操作时需要选择一个 buffer 充当缓冲，它也需要给出字节存放在主存中的准确位置(包括页号和页内偏移地址)。

```
unsigned char read_memory(unsigned int page_counter, unsigned int page_offset)
    {
    while(!(PINB & 0x02));
    PORTB &=~0x10;                                  //使能 AT45DB161
    SPI_TranByte(0x52);                             //直接读取主存
    SPI_TranByte((char)(page_counter>>6));  //页地址 PA11-PA6
    SPI_TranByte((char)((page_counter<<2)||(page_offset>>8)));
                                                    //页地址 PA5-PA0,页内偏移 BA9-BA8
    SPI_TranByte((char)page_offset);
    SPI_TranByte(0x00);                             //无关位
    SPI_TranByte(0x00);                             //无关位
    SPI_TranByte(0x00);                             //无关位
    SPI_TranByte(0x00);                             //无关位
    SPI_TranByte(0xFF);                             //写字节数据，使字节数据从 SO 移出
    PORTB|=0x10;
    return SPDR;
    }
void rite_memory(unsigned char rite_CMD, unsigned int page_counter, unsigned
int page_offset, unsigned char df_data)
    {
    while(!(PINB & 0x02));
    PORTB &=~0x10;                          //使能 AT45DB161
    SPI_TranByte(rite_CMD);                 //直接写主存,0x82 选 buffer1,0x85 选 buffer2
    SPI_TranByte((char)(page_counter>>6));  //页地址 PA11-PA6
    SPI_TranByte((char)((page_counter<<2)||(page_offset>>8)));
                                                    //页地址 PA5-PA0,页内偏移 BA9-BA8
    SPI_TranByte((char)page_offset);
    SPI_TranByte(df_data);                          //写字节数据
    PORTB|=0x10;
    }
```

4.12　复位和 Watchdog

4.12.1　复位和 Watchdog 操作的特点

AT90S4414/8515 有 3 个复位源，复位逻辑如图 4.34 所示，复位特性如表 4.40 所示。

(1) 上电复位：当电源电压低于上电门限 VPOT 时，MCU 复位；

(2) 外部复位：当/RESET 引脚上的低电平超过 50ns 时，MCU 复位；

(3) 看门狗复位：看门狗定时器超时后，MCU 复位。

在复位期间，所有的 I/O 寄存器被设置为初始值，程序从地址 0x0000 开始执行。0x0000 地址中放置的指令必须为相对跳转指令(RJMP)，跳转到复位处理程序(开始程序)。若程序不需使用中断，则中断向量处也可以放置普通的程序代码。

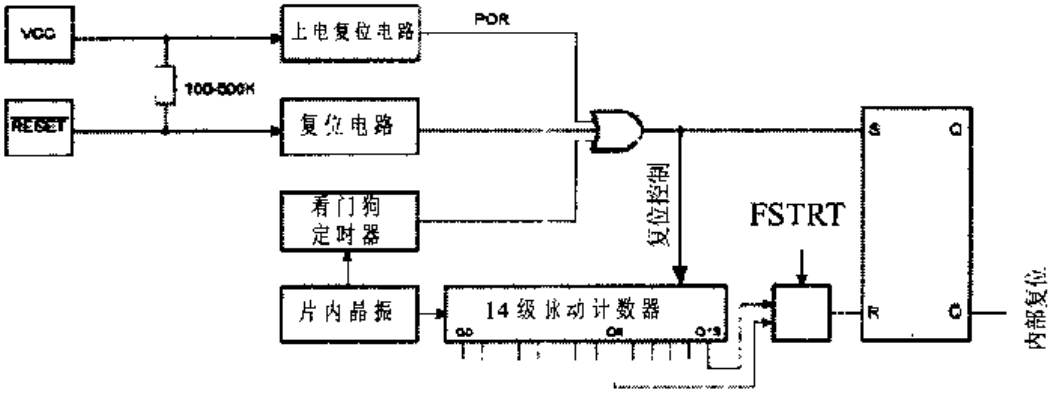


图 4.34　复位逻辑

表 4.40　复位特性(VCC = 5.0V)

符　号	参　数	最 小 值	典 型 值	最 大 值	单　位
V POT	上电复位电压门限(上升)	0.8	1.2	1.6	V
	上电复位电压门限(下降)	0.2	0.4	0.6	V
VVRST	复位引脚门限电压	–	–	0.9Vcc	V
t_{TOUT}	复位延迟定时输出，周期 FSTRT 未编程	11	16	21	ms
t_{TOUT}	复位延迟定时输出，周期 FSTRT 编程	0.25	0.28	0.31	ms

可以按照典型振荡器起振特性来选择启动时间。

用于时间溢出的看门狗 WDT 与振荡器的频率与工作电压有关，具体如表 4.41 所示。

表 4.41 看门狗振荡器周期数

FSTRT	溢出时间 VCC=5V	WDT 周期数
编程	1.1ms	1K
未编程	16.0ms	16K

4.12.1.1 上电复位

上电复位(POR)保证器件在上电时正确复位，看门狗定时器 WDT 驱动一个内部定时器，此定时器保证 MCU 只有在 V_{CC} 达到 V_{POT} 且过了一定时间之后才启动。全部的复位时间为 POWER_ON 复位时间 t_{POR} 加上延时时间 t_{TOUT}，上电复位时序如图 4.35 和图 4.36 所示，其中图 4.35 为复位引脚通过一个上拉电阻直接与 VCC 连接，而图 4.36 的复位引脚由外电路控制。

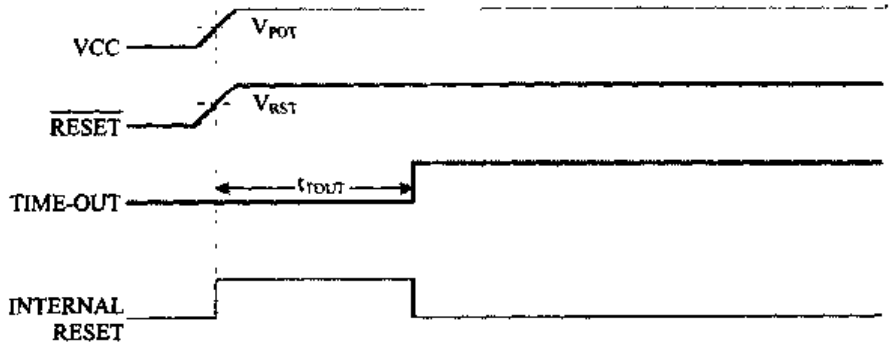


图 4.35 MCU 启动，RESET#与 VCC 相连

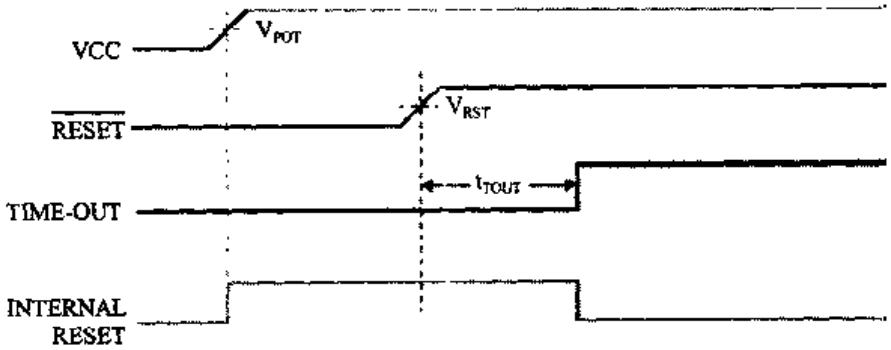


图 4.36 MCU 启动，RESET#由外部控制

如果您使用了陶瓷振荡器或是其他快速启动的振荡器，编程位于 FLASH 内的 FSTRT 熔丝位，可以使 MCU 在较短的时间启动。如果内置于片内的启动时间足够的话，RESET#引脚可以与 VCC 直接相连或是外接上拉电阻；如果在加上 VCC 的同时保持/RESET 为低，则可以延长复位周期。

4.12.1.2　外部复位

RESET#复位引脚上的低电压引发外部复位，低电压至少保持两个晶振时钟周期(20ns 左右)。当外加信号达到复位门限电压 V_{RST}(上升沿)时，t_{TOUT} 延时周期开始，然后，MCU 开始启动，工作期间的外部复位时序如图 4.37 所示。

注意：短脉冲不能保证可靠复位。

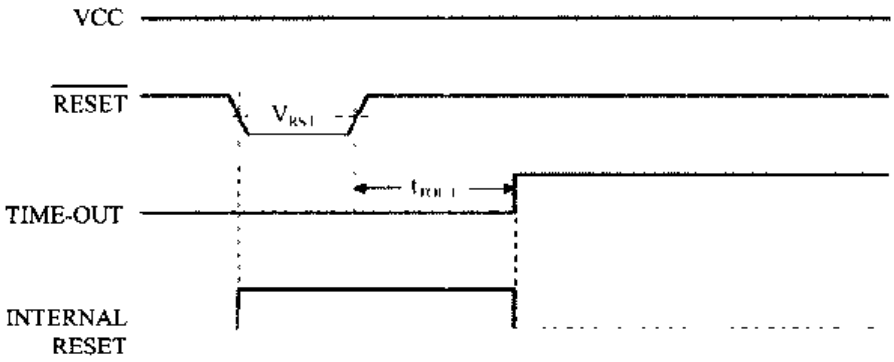


图 4.37　工作期间的外部复位

4.12.1.3　看门狗复位

当看门狗定时器溢出时，将产生 1 个晶振时钟周期的复位脉冲。在脉冲的下降沿，延时定时器开始对 t_{TOUT} 计数，如图 4.38 所示。

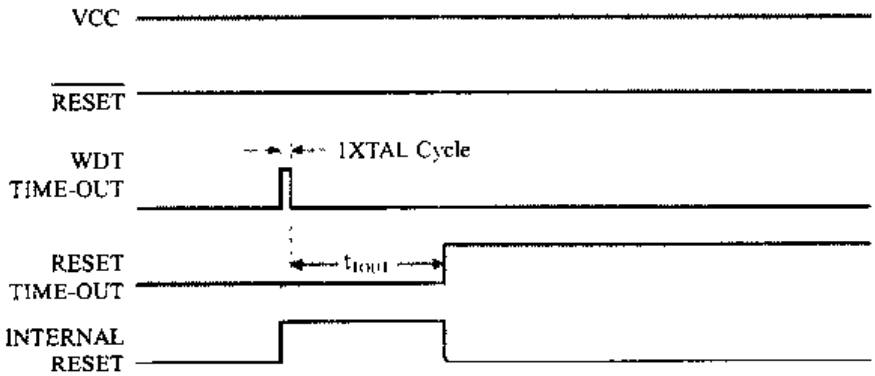


图 4.38　工作期间的看门狗复位

4.12.1.4　看门狗 WDT

看门狗定时器由片内独立的振荡器驱动，在 VCC=5V 的条件下典型振荡频率为 1MHz。通过调整定时器的预分频因数(8 种)可以改变看门狗复位时间间隔。

看门狗复位指令是 WDT，如果定时时间已到且没有执行 WDT 指令，则看门狗将复位 MCU，使 CPU 从复位地址(0x0000)重新开始执行。

为了防止不小心关闭看门狗，需要有一个特定的关闭程序。

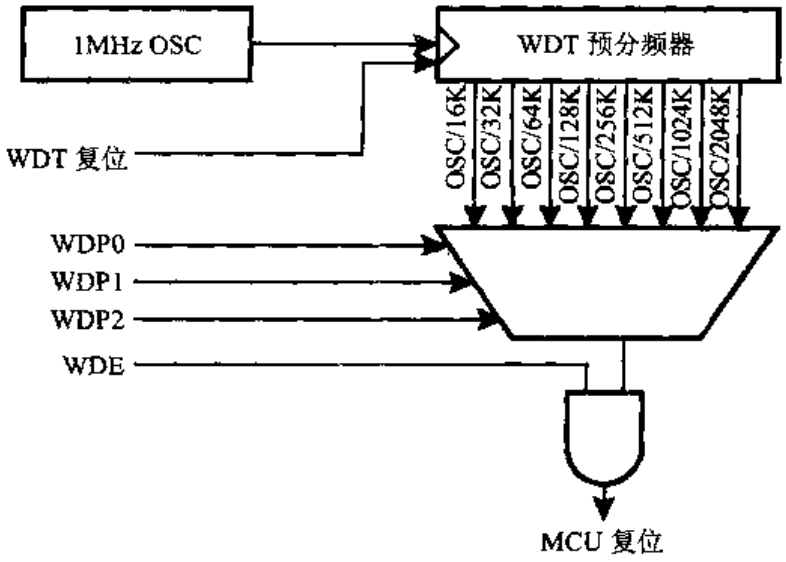


图 4.39 看门狗定时器

看门狗控制寄存器—WDTCR，其定义如表 4.42 所示。

表 4.42 看门狗定时器控制寄存器—WDTCR

BIT	7	6	5	4	3	2	1	0
WDTCR(0x21)	-	-	-	WDTOE	WDE	WDP2	WDP1	WDP0
读/写	R	R	R	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

WDTOE，看门狗关闭使能：

当 WDE 清零时，此位必须为“1”才能关闭看门狗。在置位的 4 个时钟周期后硬件对其清零。

WDE，看门狗使能：

WDE 为“1”时，看门狗使能。只有在 WDTOE 为“1”时，WDE 才能清零。以下为关闭看门狗的步骤：

(1) 在同一个指令内对 WDTOE 和 WDE 写逻辑“1”(即使 WDE 已经为“1”)；

(2) 在 4 个时钟之内对 WDE 写逻辑“0”。

WDP2/ WDP1/ WDP0 预分频器：

当使用看门狗 WDT 时，WDP2/ WDP1/ WDP0 决定它的分频选择，其控制作用如表 4.43 所示。不同的分频系数，可以选择不同的 WDT 定时。

表 4.43 看门狗定时器预分频选择

WDP2	WDP1	WDP0	振 荡 周 期	典型溢出时间 VCC=3V	典型溢出时间 VCC=5V
0	0	0	16K	47ms	15ms
0	0	1	32K	94ms	30ms
0	1	0	64K	0.19s	60ms

续表

WDP2	WDP1	WDP0	振 荡 周 期	典型溢出时间 VCC=3V	典型溢出时间 VCC=5V
0	1	1	128K	0.38s	0.12s
1	0	0	256K	0.75s	0.24s
1	0	1	512K	1.5s	0.49s
1	1	0	1024K	3.0s	0.97s
1	1	1	2048K	6.0s	1.9s

注意：

(1) 看门狗的振荡频率和系统电压有关。

(2) 应该在看门狗使能之前执行一次喂狗指令 WDT，如果看门狗在复位之前使能，则看门狗定时器有可能不是从 0 开始计数。

4.12.2　复位和 WDT 的 C 源程序实例及剖析

看门狗 WDT 的使用比较简单，当 Vcc = +5V 时，使用最大的 WDT 周期。以 AT90S8515 为例，初始化并打开 WDT，使用 2048K 预分频(约 2.48 秒复位)：

```
void WDT_init(void)
    {
    WDR();                          //初始化前先喂狗
    WDTCR=0x0F;                     //WDT 使能，使用 2048K 分频
    }
```

ICCAVR 编译后的汇编代码：

```
WDT_init:
wdr                                 ;初始化前先喂狗
ldi R24,15                          ;WDT 使能，使用 2048K 分频
out 0x21,R24
ret
```

关闭看门狗 WDT：

```
void WDT_off(void)
    {
    WDTCR=0x18;                     //写逻辑"1"到 WDTOE 和 WDE
    WDTCR=0x00;                     //关闭 WDT
    }
```

在程序里就可以使用看门狗 WDT 了，注意喂狗指令 WDR 放置的位置，要保证程序执行过程中不会复位，又保证当程序陷入死循环后，在允许的时间内复位。

ICCAVR 编译后的汇编代码为：

```
WDT_off:
    ldi R24,24
    out 0x21,R24                ;写逻辑"1"到 WDTOE 和 WDE
    clr R2
    out 0x21,R2                 ;关闭 WDT
    ret
```

第 5 章　ICCAVR 应用实例

5.1　C 程序优化

对程序进行优化，通常是指优化生成目标文件代码或程序执行速度，源程序编译时对代码进行优化和对速度进行优化实际上是一个矛盾的统一，一般是优化了代码的大小，就会延长执行时间，如果优化了程序的执行速度，则会带来代码增加的副作用，很难兼顾，只能在编译时掌握一个平衡点。但在编写源程序时，如果选用适当的表示方法及算法，也可能实现代码和速度一起优化。

5.1.1　程序结构的优化

1．程序的书写格式

虽然书写格式并不会影响生成的代码质量，但是在实际编写程序时还是应该遵循一定的书写规则，一个书写清晰、明了的程序，有利于以后的升级和维护。在书写程序时，特别是对于 While、for、do...while、if...elst、switch...case 这些语句或这些语句的嵌套组合，应采用“缩格”的书写形式。本书对源程序的书写均采用“缩格”的格式，可以参考本书的书写格式。

2．标识符

程序中使用的用户标识符除要遵循标识符的命名规则以外，一般不要用代数符号(如 a、b、x1、y1)作为变量名，应选取具有相关含义的英文单词或汉语拼音作为标识符(也可以用其缩写)，以增加程序的可读性，如：count、number1、red、work 等。

3．程序结构

C 语言是一种高级程序设计语言，提供了十分完备的规范化流程控制结构。因此在采用 C 语言设计单片机应用系统程序时，首先要注意尽可能采用结构化的程序设计方法，这样可使整个应用系统程序结构清晰，便于调试和维护。对于一个较大的应用程序，通常将整个程序按功能分成若干个模块，不同模块完成不同的功能，一般单个模块完成的功能较为简单，设计和调试也相对容易一些，而且各个模块可以分别编写，甚至还可以由不同的程序员编写。在 C 语言中，一个函数就可以认为是一个模块。所谓程序模块化，不仅是要将整个程序划分成若干个功能模块，更重要的是，还应该注意保持各个模块之间变量的相对独立性，即保持模块的独立性，尽量少使用全局变量等。对于一些常用的功能模块，还

可以封装为一个应用程序库，以便需要时可以直接调用。但是在使用模块化时，如果将模块分得太细太小，又会导致程序的执行效率变低(进入和退出一个函数时需要保护和恢复寄存器，这会占用一些运行时间)。

4．定义常数

在程序设计过程中，对于经常使用的一些常数，如果将它直接写到程序中去，一旦常数的数值发生变化，就必须逐个找出程序中所有的常数，并逐一进行修改，这样必然会降低程序的可维护性。因此，应尽量采用预处理命令方式来定义常数，而且还可以避免输入错误。

5．使用条件编译

能够使用条件编译(ifdef)的地方就使用条件编译，而不使用 if 语句，有利于减小编译生成的代码的长度，并且能够提高执行速度。

6．表达式

对于一个表达式中各种运算执行的优先顺序不太明确或容易混淆的地方，应当采用圆括号明确指定它们的优先顺序。一个表达式通常不能写得太复杂，如果表达式太复杂，时间久了以后，自己也不容易看得懂，不利于以后的维护。

7．函数

对于程序中的函数在使用之前，应对函数的类型进行说明，对函数类型的说明必须保证它与原来定义的函数类型一致，对于没有参数和没有返回值类型的函数应加上“void”说明。如果需要缩短代码的长度，可以将程序中一些公共的程序段定义为函数；如果需要缩短程序的执行时间，在程序调试结束后，将部分函数用宏定义来代替。注意，应该在程序调试结束后再定义宏，因为大多数编译系统在宏展开之后才会报错，这样会增加排错的难度。

8．尽量少用全局变量，多用局部变量。因为全局变量是放在数据存储器中，定义一个全局变量，MCU 就少一个可以利用的数据存储器空间，如果定义了太多的全局变量，会导致编译器无足够的内存可以分配。而局部变量大多定位于 MCU 内部的寄存器中，也有部分定位在数据存储器中，在绝大多数 MCU 中，使用寄存器操作速度比数据存储器快，指令也更多更灵活，有利于生成质量更高的代码，而且被局部变量所占用的寄存器和数据存储器在不同的模块中可以重复利用。

9．设定合适的编译程序选项

许多编译程序有几种不同的优化选项，在使用前应理解各优化选项的含义，然后选用最合适的一种优化方式。通常情况下一旦选用最高级优化，编译程序会近乎病态地追求代码或速度的优化，可能会影响程序的正确性，导致程序运行出错。下面介绍几种常用的 C 编译器可供选择的优化选项：

在 Keil 中，有速度优先和代码优先两种模式，每种又有 10 级优化可供选择。

在 ICCAVR 中，有“Default”和“Enable Code Compression”两个优化选项。

在 CodeVisionAVR 中，有“Tiny”和“small”两种内存模式。

在 IAR 中，共有 7 种不同的内存模式选项。

在 GCCAVR 中优化选项更多，一不小心更容易选到不恰当的选项。

因此应熟悉所使用的编译器，知道哪些参数在优化时会受到影响，哪些参数不会受到影响，这样才能够做到选择最合适的优化选项。

5.1.2　源程序中代码的优化

1．选择合适的算法和数据结构

应该熟悉算法语言，知道各种算法的优缺点，很多计算机书籍上都有算法介绍，具体资料请参见相应的参考资料。

将比较慢的顺序查找法用较快的二分查找或乱序查找法代替，将插入排序或冒泡排序法用快速排序法、合并排序法或根排序法代替，这些都可以大大提高程序执行的效率。选择一种合适的数据结构也很重要，比如你在一堆随机存放的数中使用了大量的插入和删除指令，那使用链表就比较快。

数组与指针具有十分密切的关系，一般来说，指针比较灵活简洁，而数组则比较直观，容易理解。对于大部分的 C 编译器，使用指针比使用数组生成的代码更短，执行效率更高。但是由于 Keil 编译器的特殊性，在 C51 中使用数组比使用指针生成的代码更短，效率更高。

2．使用尽量小的数据类型

能够使用字符型(char)定义的变量，就不要使用整型(int)变量来定义；能够使用整型变量定义的变量就不要用长整型(long int)，特别是能不用浮点型(float)变量就尽量不要使用浮点型变量，使用浮点型变量会使程序代码量增加很多，执行速度明显降低。当然，在定义变量后不能超过变量的有效范围，如果超过变量的范围赋值，大部分的 C 编译器并不报错，但程序运行结果却不符合设计意图，而且这样的错误很难排查。

在 keil 中，除了上面介绍的在定义变量时尽量选择小的数据类型外，在定义指针时应说明指针所指向的对象类型，少用通用型指针，以提高编译效率和运行速度。

3．使用自加、自减指令

通常使用自加、自减指令和复合赋值表达式(如 a++及 a---=1 等)都能够生成高质量的程序代码，编译器通常都能够直接生成 inc 和 dec 之类的指令，而使用 a=a+1 或 a=a-1 之类的指令，很多 C 编译器都会生成二到三条的指令来完成加 1 或减 1 的操作。在 AVR 单片机适用的 ICCAVR、GCCAVR、IAR 等几种常见的 C 编译器中，以上几种书写方式生成的代码是一样的，都直接生成 inc 和 dec 指令。

4．减少运算的强度

可以使用运算量小但功能相同的表达式替换原来复杂的表达式。

(1) 求余运算。

```
a=a%8;
```

可以改为：

```
a=a&7;
```

说明：位操作只需一个指令周期即可完成，而大部分的 C 编译器的求余(%)运算均是调用子程序来完成，代码长，执行速度慢。通常，只要是求 2^n 的余数，均可使用位操作的方法来代替。

(2) 平方运算

```
a=pow(a,2.0);
```

可以改为：

```
a=a*a;
```

说明：在有内置硬件乘法器的单片机中(如 51 系列)，乘法运算比求平方运算快得多，因为求平方是通过调用子程序来实现的，其中还使用到大量的浮点运算，生成的代码量都很大。在自带硬件乘法器的 AVR 单片机中，如 ATmega128 中，乘法运算只需 2 个时钟周期就可以完成，即使在没有内置硬件乘法器的 AVR 单片机中，乘法运算的子程序比平方运算的子程序代码也短得多，执行速度也快很多。

如果是求 3 次方或者是 3 次方以上时，如：

```
a=pow(a,3.0);
```

更改为：

```
a=a*a*a;
```

代码效率和执行效率均有明显的改善。

(3) 用移位实现乘除法运算

```
a=a*4;
b=b/4;
```

可以改为：

```
a=a<<2;
b=b>>2;
```

说明：通常如果需要乘以或除以 2^n，都可以用移位的方法代替。在 ICCAVR 中，如果乘以 2^n，编译器不是调用乘法子程序而是直接生成左移 n 位的代码，但乘以非 2^n 的整数或除以任何整数(包含 2^n)，需调用乘除法子程序运算。其他的 C 编译器，在乘以或除以 2^n 时，通常都是通过调用相应的子程序实现的。

用移位的方法得到代码比调用乘除法子程序生成的代码效率高得多，实际上，只要是乘以或除以一个整数，均可以用移位的方法得到结果，如：

```
a=a*9
```

可以改为：

```
a=(a<<3)+a
```

(4) 少用浮点运算

```
int a=200;
float b;
b=a*89.65
```

在上例中，如果变量 b 不定义为浮点型，而改为长整型，如下：

```
int a=200;
long int b;
b=a*8965/100;
```

计算结果基本相同(可能使用浮点数会略有一点小误差)，但是在改用长整型后，生成的代码却少了很多，执行效率得到很大的提高。

在很多情况下，如果忽略小数点部分对整个数值的影响，就省略小数点部分，改为定义整型或长整型。如果中间变量为浮点型且不能忽略小数点，也可以将该中间变量乘以 2^n 或 10^n 后，转换为长整型数，其中对于乘以 2^n 可以使用移位的方式实现，但不直观，而乘以 10^n 生成的代码量较大，但更直观。在最后运算时应记着除去 2^n 或 10^n，恢复该变量的原值。

5. 循环

(1) 循环语句

对于一些不需要循环变量参加运算的任务可以把它们放到循环外面，这里的任务包括表达式、函数的调用、指针运算、数组访问等。应该将没有必要执行多次的操作全部集合在一起，放到一个 init 的初始化程序中进行。

(2) 延时函数

大多数设计人员使用的延时函数均采用自加的形式：

```
void delay (void)
     {
     unsigned int i;
     for (i=0;i<1000;i++)
          ;
     }
```

将其改为自减的形式：

```
void delay (void)
     {
     unsigned int i;
     for (i=1000;i>0;i--)
          ;
     }
```

两个函数的延时效果相同，但几乎所有的 C 编译对后一种函数生成的代码均比前一种代码少 1~3 个字节，因为几乎所有的 MCU 均有 0 转移的指令，采用后一种方式能够生成这类指令，缩短代码长度。在使用 while 循环时也一样，使用自减指令控制循环会比使用自加指令控制循环生成的代码更少 1~3 个字母。

但是在循环中，通过循环变量“i”读写数组的指令时，读出数组中的数据先后次序与数组顺序相反，并且可能在读数组时超界，这两点在设计时要引起足够的注意。

(3) while 循环和 do...while 循环

用 while 循环时有以下两种循环形式：

```
unsigned int i;
i=0;
while (i<1000)
    {
    i++;
    //用户程序
    }
```

或：

```
unsigned int i;
i=1000;
do
    i--;
    //用户程序
while (i>0);
```

在这两种循环中，使用 do...while 循环编译后生成的代码的长度短于 while 循环。

6. 查表

在程序中一般不进行非常复杂的运算，如浮点数的乘除及开方以及一些复杂数学模型的插补运算等，对这些既消耗时间又消耗资源的运算，应尽量使用查表的方式，并且将数据表置于程序存储区。如果在编程时生成所需的数据表比较困难，也应改为在启动时先计算数值，然后在数据存储器中生成所需的数据表，以后在程序运行过程中直接查表就可以了，减少了程序执行过程中重复计算的工作量，并且能够大大提高程序的执行速度。

7. 其他

使用在线汇编以及将字符串和一些常量保存在程序存储器中，均能够提高生成代码的质量。

5.2 延 时 函 数

1. 毫秒级的延时

延时 1ms：

```
void delay_1ms(void)
    {
    unsigned int i;
    for(i=1;i<(unsigned int)(xtal*143-2);i++)
      ;
    }
```

在上式中，xtal 为晶振频率，单位为 MHz。

当晶振频率为 8M 时，延时函数软件仿真的结果为 1000.25μs。当晶振频率为 4M 时，延时函数软件仿真的结果为 999.5μs。

如果需要准确的 1ms 延时时间，则本计算公式只供参考，应通过软件仿真后，再确定循环的次数及循环初值，并且循环中还必须关闭全局中断，防止中断影响延时函数的延时时间。

下面的函数可以获得 1ms 的整数倍延时时间：

```
void delay(unsigned int n)
    {
    unsigned int i=0;
    for (i=0;i<n;i++)
    delay_1ms();
    }
```

如果需要准确的延时时间，则该函数只供参考，应通过软件仿真后，再确定循环的次数及初值。假设计某个程序需要 500ms 的延时，可以使用如下延时函数实现：

```
delay(500);
```

2. 微秒级延时

晶振频率为 8M 时 1μs 延时函数：

```
void delay_1us(void)
    {
    asm("nop");
    }
```

当晶振频率为 8M 时，软件仿真的结果为 1.0μs。

当然也可以使用宏定义实现 1.0μs：

```
#define delay_1us() asm("nop");asm("nop");asm("nop");asm("nop");asm("nop");
asm("nop");\asm("nop");asm("nop")
```

如果需要小于 1μs 的延时，只有使用宏定义实现，当然，也可以直接插入在线汇编

```
asm("nop");
```

语句实现延时。

在程序中需要微秒级别的延时，可以用以下函数实现。

```
void delay_us(unsigned int n)
    {
    unsigned int i=0;
    for (i=0;i<n;i++)
    delay_1us();
    }
```

说明：在微秒级别的延时中，如果需要精确的延时时间，还必须关闭中断，并且应通过软件仿真后再确定循环的次数及初值。

强调：在实际应用中一般不直接使用软件进行长时间延时，因为 MCU 一直停留延时函数中(称为阻断)，不能再干其他的事情(除了中断外)，只有非常简单的应用或者简单的演示时才能使用延时函数实现长时间延时。实际应用中，对长时间(较简单任务一般指几十毫秒以上，对于复杂的应用，一般指几毫秒以上)的延时，应采用非阻断式的延时方式，或者使用定时器中断来完成延时。

5.3 读/写片内 EEPROM

```
/*********************************************
Project              :读/写 AT90S8515 片内的 EEPROM
Chip type            :AT90S8515
Clock frequency      :8.000000 MHz
Author               :沈文
Comments             :
使用了 eeprom.h 头文件中的读写 EEPROM 函数
*********************************************/
#include "io8515v.h"
#include "eeprom.h"
void main(void)
     {
     unsigned char i;
     i=EEPROMread(0x10);          //读 EEPROM 地址 0x10 的数据
     EEPROMwrite(0x20,i);         //将 i 写入到 EEPROM 地址 0x20 处
     }
```

5.4　信号周期测量程序

```
/*********************************************
Project              :信号周期测量演示程序
Chip type            :AT90S8515
Clock frequency      :8.000000 MHz
Author               :詹卫前
Rework               :沈文
From                 :广州双龙电子
Comments             :
1. 学习定时器 T1 捕获中断的使用
2. 6 路 LED 动态扫描程序的演示
3. 本程序在 SLAVR 上测试通过，硬件连接原理图请参见附录 A
*********************************************/
#include "io8515v.h"
#pragma interrupt_handler Icp_timer1:4      //说明 ICP 为中断处理函数
#pragma data:code                           //设置数据区为程序存储器
const unsigned char tabel[]={0x3f,0x06,0x5b,0x4f,0x66,0x6d,0x7d,0x07,0x7f,0x6f,0x77,
                             0x7c,0x39,0x5e,0x79,0x71};          //七段译码字形表
#pragma data:data                           //设置数据区回到数据存储器
unsigned char ledbuff[]={0x3f,0x3f,0x3f,0x3f,0x3f,0x3f};   //显示缓冲区
unsigned int oldcount;
unsigned int newcount;
/*******************************
        1ms 延时程序
*******************************/
void delay_1ms(void)
    {
    unsigned int i;
    for(i=1;i<1142;i++)
    ;
}
/*******************************
     六路动态扫描显示电路
*******************************/
void display(void)
    {
    unsigned char i;
```

```
    for (i=0;i<6;i++)
         {
         PORTB=ledbuff[i];              //将显示缓冲区数据送 PORTB 口
         PORTD=~(1<<i);                 //开始显示
         delay_1ms();                   //每一位显示保持一定时间
         PORTD|=(1<<i);                 //关闭显示
         }
    }
/************************************
  将 count 十六进制数据转换为 LED 七段码
************************************/
void hextobcd(unsigned int count)
    {
    unsigned char i,temp;
    for (i=0;i<6;i++)
         {
         temp=count%10;
         ledbuff[i]=tabel[temp];
         count=count/10;
         }
    }
/*******************************
        MCU 初始化
*******************************/
void mcu_init(void)
    {
    DDRD=0x3f;
    DDRB=0xff;
    PORTD=0xff;
    PORTB=0xff;                         //端口初始化
    TIMSK=0x08;                         //使能 T1 捕捉中断
    TCCR1A=0x00;
    TCCR1B=0xC2;                        //CK/8，捕捉周期的单位为 1μs
    ICR1=0;
    TCNT1=0;
    }
/************************************************
       主程序：测量 ICP 引脚上信号的周期
************************************************/
void main()
    {
    mcu_init();
```

```
    SREG|=0x80;                          //使能全局中断
    for(;;)
         {
         if ((newcount&0xfff8)!=(oldcount&0xfff8))
              {
              oldcount=newcount;
              }
         hextobcd(oldcount);
         display();                      //显示测量值
         }
    }
/*************************************
           捕捉中断处理程序
*************************************/
void Icp_timer1(void)
    {
    newcount=ICR1;
    ICR1=0;
    TCNT1=0;
    TCCR1B=0xC2;
    }
```

5.5　键盘扫描程序

```
/*********************************************
Project              :键盘扫描程序
Chip type            :AT90S8515
Clock frequency      :8.000000 MHz
Author               :詹卫前
Rework               :沈文
From                 :广州双龙电子
Comments             :
1. 学习键盘动态扫描程序
2. 学习 LED 动态扫描显示
3. 本程序在 SLAVR 上测试通过，硬件连接原理图请参见附录 A。
说明:
键盘扫描函数,没有键按下返回 0x7f、如果 shift 按下，返回值最高位为 1
*********************************************/
#include "io8515v.h"
```

```
#pragma data:code                          //设置数据区为程序存储器
const unsigned char tabel[]={0x3f,0x06,0x5b,0x4f,0x66,0x6d,0x7d,0x07,0x7f,0x6f,0x77,
                             0x7c,0x39,0x5e,0x79,0x71};          //七段译码字形表
#pragma data:data                          //设置数据区回到数据存储器
unsigned char ledbuff[]={0x3f,0x3f,0x3f,0x3f,0x3f,0x3f};    //显示缓冲区
void delay_us(int time)                    //微秒级延时程序
    {
    do
    time--;
    while (time>1);
    }
void delay_ms(unsigned int time)           //毫秒级延时程序
    {
    while(time!=0)
        {
        delay_us(1000);
        time--;
        }
    }
unsigned char scan_key(void)               //不作按键释放检查的键盘扫描函数
    {
    unsigned char i,temp;
    unsigned shift=0;
    DDRC=0x0f;
    PORTC=0xff;
    for (i=0;i<4;i++)
        {
        PORTC=~(1<<i);                     //键盘线扫描
        delay_us(100);                     //等待低电平稳定
        temp=PINC&0xf0;                    //读取键盘扫描信号,屏蔽低四位
        if ((PIND&0x80)==0)
             temp|=01;                     //检查 shift 键
        if (temp!=0xf0)                    //如果有键按下，延时 15ms 消抖
             {
             delay_ms(15);                 //可消除显示抖动
             temp=PINC&0xf0;               //再读键盘
             if ((PIND&0x80)==0)           //检查 shift 键
                  {
                  temp|=01;
                  shift=1;
                  }
             if (temp!=0xf0)
```

```
                {
                temp&=0xf0;
                switch (temp)       //计算键值
                    {
                    case 0x70:temp=15-(3-i)*4;break;
                    case 0xb0:temp=14-(3-i)*4;break;
                    case 0xd0:temp=13-(3-i)*4;break;
                    case 0xe0:temp=12-(3-i)*4;break;
                    default:temp=0x7f;
                    }
                if (shift==1) temp =0x80;
                return temp;
                }
            }
        PORTC=0xff;
        }
    return 0x7f;
    }
unsigned char keypad(void)          //有按键释放检查的键盘扫描函数
    {
    unsigned char temp1,temp2;
    temp1=scan_key();
    if (temp1==0x7f)
        return 0x7f;
    do
        temp2=scan_key();
    while (temp1==temp2);
    return temp1;
    }
    void port_init(void)            //端口初始化
        {
        DDRC=0x0f;
        DDRD=0x3f;
        DDRB=0xff;
        PORTD=0xff;
        PORTB=0xff;
        PORTC=0xff;
        }
    void display(void)              //六路动态扫描显示
        {
        unsigned char i;
        for (i=0;i<6;i++)
```

```
            {
            PORTB=ledbuff[i];
            PORTD=~(1<<i);
            delay_ms(1);
            PORTD|=(1<<i);
            }
        }
    /*    主程序的功能：读取键盘扫描码，转换成相应的字形码后送入显示缓冲区进行显示，如果
shift 键按下，则全部清 0   */
       void main(void)
        {
        unsigned char keyval1,keyval2;
        unsigned char i=0;
        MCUCR=0;
        port_init();
        while(1)
            {
            keyval1=scan_key();              //键盘扫描函数，不作按键释放检查
            if (keyval1!=0x7f)
                {
                do
                  {
                  keyval2=scan_key();
                  display();
                  }
                while (keyval1==keyval2);   //作按键释放检查
                if ((keyval1&0x80)==0x80)
                  {
                  for(i=0;i<6;i++)
                  ledbuff[i]=tabel[0];
                  }
                else
                  {
                  for (i=0;i<5;i++)
                       ledbuff[5-i]=ledbuff[4-i];  //每按一次键，左移一位
                  ledbuff[0]=tabel[keyval1&0x7f];
                  }
                }
            display();
            }
        }
```

5.6　生成模拟音乐

```
/*********************************************
Project              :利用 I/O 口生成模拟音乐
Chip type            :AT90S8515
Clock frequency      :8.000000 MHz
Author               :詹卫前
Rework               :沈文
From                 :广州双龙电子
Comments             :
1. 学习定时器 T0 溢出中断的使用
2. 学习定时器 T1 比较中断的使用
3. 学习计算机音乐的产生
4. 本程序在 SLAVR 上测试通过，硬件连接原理图请参见附录 A
*********************************************/
#include "io8515v.h"
#include "macros.h"
#pragma interrupt_handler timer0:8
#pragma interrupt_handler timer1:5
//#pragma data:code                          //设置数据区为程序存储器
flash unsigned int music_data[][2]=
/*******************************************
       卡秋莎音乐数据{x,y}
       x:对应音符音阶(频率),0 表示休止符
       y:对应音符节拍
*******************************************/
{
{440,600},{494,200},{523,600},{440,200},
{523,400},{494,200},{440,200},{494,400},
{330,400},{494,600},{523,200},{578,600},
{494,200},{578,400},{523,200},{494,200},
{440,800},{659,400},{880,400},{784,400},
{880,200},{784,200},{698,400},{659,200},
{578,200},{659,400},{440,400},{  0,200},      //休止符
{698,400},{578,200},{659,600},{523,200},
{494,200},{330,200},{523,200},{494,200},
{440,800},{659,400},{880,400},{784,400},
{880,200},{784,200},{698,400},{659,200},
```

```
{578,200},{659,400},{440,400},{  0,200},      //休止符
{698,400},{578,200},{659,600},{523,200},
{494,200},{330,200},{523,200},{494,200},
{440,800},{0,0}
};
#pragma data:data                              //设置数据区回到数据存储器
unsigned int delay=0;
/*******************************
      MCU初始化
*******************************/
void music_init(void)
    {
    MCUCR=0x00;
    DDRC=0x01;
    TCCR1A=0x00;
    TCCR1B=0x09;
    TCCR0=0x03;
    TCNT0=0x19;
    TIMSK=0x42;
    }
/*************************************************
     T0中断程序，产生音乐节拍
*************************************************/
void timer0(void)
    {
    delay++;
    TCNT0=0x19;
    }
/*************************************************
     T1中断程序，根据SOUND函数输出一定频率的方波
*************************************************/
void timer1(void)
    {
    PORTC^=0x01;
    }
/*************************************************
     SOUND程序，输出频率为 x Hz 的方波,延时 y ms
      x:100~20000 Hz, 0表示不发声
      y:0~65536 ms
*************************************************/
void sound(unsigned int x,unsigned int y)
    {
```

```
    SEI();
    delay=0;
    if (x!=0)
         {
         x=4000000/x;
         OCR1A=x;
         TCNT1=0x00;
         TIMSK|=0x40;
         while(delay<y)
             ;
         TIMSK&=0xbf;
         }
    else
         {
         TIMSK&=0xbf;
         while(delay<y)
             ;
         }
    CLI();
    }
/*********************************
         主程序
*********************************/
void main(void)
    {
    unsigned char i=0;
    music_init();
    while(1)
         {
         while(music_data[i][1]!=0)
             {
             sound(music_data[i][0],music_data[i][1]);
             i++;
             }
         i=0;
         }
    }
```

5.7 利用 I²C 总线读写 AT24C02

```
/*********************************************
Project              :利用 I2C 总线读写 AT24C02
Chip type            :AT90S8515
Clock frequency      :8.000000 MHz
Author               :詹卫前
Rework               :沈文
From                 :广州双龙电子
Comments             :
1. 学习 I2C 总线的读写(利用 SLAVR 库)。
2. 应包含头文件 slavr.h，在工程选项中的 Additional Lib 中填入 slavr。
3. 本程序在 SLAVR 上测试通过，硬件连接原理图请参见附录 A。其中：
   SDA   PA0
   SCL   PA1
*********************************************/
#include "io8515v.h"
#include "slavr.h"
#pragma data:code                          //设置数据区为程序存储器
const unsigned char tabel[]={0x3f,0x06,0x5b,0x4f,0x66,0x6d,0x7d,0x07,0x7f,0x6f,0x77,
                             0x7c,0x39,0x5e,0x79,0x71};          //七段译码字形表
#pragma data:data                          //设置数据区回到数据存储器
unsigned char ledbuff[]={0x3f,0x3f,0x3f,0x3f,0x3f,0x3f};   //显示缓冲区
unsigned char at24_add,at24_data;
/*****************************************
            I2C 总线写一个字节
*****************************************/
void i2c_Write(unsigned char Wdata,unsigned char RomAddress)
   {
   Start();
   Write8Bit(0xa0);
   TestAck();
   Write8Bit(RomAddress);
   TestAck();
   Write8Bit(Wdata);
   TestAck();
   Stop();
   delay_ms(10);
```

```
    }
/*******************************************
            I2C 总线读一个字节
*******************************************/
unsigned char i2c_Read(unsigned char RomAddress)
    {
    unsigned char temp;
    Start();
    Write8Bit(0xa0);
    TestAck();
    Write8Bit(RomAddress);
    TestAck();
    Start();
    Write8Bit(0xa1);
    TestAck();
    temp=Read8Bit();
    NoAck();
    Stop();
    return temp;
    }
/*******************************************
            端口初始化
*******************************************/
void port_init(void)
    {
    DDRD=0x3f;
    DDRB=0xff;
    PORTD=0xff;
    PORTB=0xff;
    }
/*******************************************
        六路动态扫描显示电路
*******************************************/
void display(void)
    {
    unsigned char i;
    i=at24_data%16;
    ledbuff[0]=tabel[i];
    i=at24_data/16;
    ledbuff[1]=tabel[i];
    i=at24_add%16;
    ledbuff[2]=tabel[i];
```

```
    i=at24_add/16;
    ledbuff[3]=tabel[i];
    for (i=0;i<6;i++)
        {
        PORTB=ledbuff[i];                    //将显示缓冲区数据送 PORTB 口
        PORTD=~(1<<i);                       //开始显示
        delay_1ms();                         //每一位显示保持一定时间
        PORTD|=(1<<i);                       //关闭显示
        }
    }
/*******************************************
主程序功能：演示了读写 AT24C02 的过程
1. 键盘上 0~F 键用于输入地址或数据，SHIFT 键用于切换状态
2. 上电复位后，按数字键只可以修改地址，程序自动显示 AT24C02 单元内容
3. 按一次 SHIFT 键，最高位 LED 小数点亮，此时可以修改数字，再按一次 SHIFT 键
程序将修改后的数字写入 AT24C02，然后回到上电复位后的状态
*******************************************/
void main(void)
    {
    unsigned char key1,key2;
    unsigned char index,flag;
    MCUCR=0;
    at24_add=0;
    at24_data=0;
    index=0;
    flag=0;
    port_init();                             //端口初始化
    while(1)
        {
        key1=scan_key();
        if (key1!=0x7f)
            {
            do
               {
               key2=scan_key();              //检查按键释放
               display();
               }
            while(key1==key2);
            if (key1>=0x80)
               {
               if (flag!=0x00)               //SHIFT 键切换数据和地址修改
                    {
```

```
            i2c_Write(at24_data,at24_add);
            flag=0x00;
            }
        else
            flag=0x80;
        ledbuff[5]=0x00^flag;
        index=0;
        }
    else
        {
        switch (index)
            {
            case 0:
                {
                if (flag==0x00)
                        {
                        key2=at24_add&0x0f;
                        at24_add=key2|(key1<<4);
                        }
                else
                        {
                        key2=at24_data&0x0f;
                        at24_data=key2|(key1<<4);
                        }
                index=01;
                break;
                }
            case 1:
                {
                if (flag==0x00)
                        {
                        key2=at24_add&0xf0;
                        at24_add=key2|(key1&0x0f);
                        }
                else
                        {
                        key2=at24_data&0xf0;
                        at24_data=key2|(key1&0x0f);
                        }
                index=00;
                break;
            }
```

```
                }
            }
        }
        if (flag==0x00)
        at24_data=i2c_Read(at24_add);
        display();
    }
}
```

5.8 利用单总线访问 DS18B20

```
/*********************************************
Project          :利用单总线访问 DS18B20
Chip type        :AT90S8515
Clock frequency :8.000000 MHz
Author           :詹卫前
Rework           :沈文
From             :广州双龙电子
Comments         :
1. 学习单总线的读写
2. 学习 2*16LCD 液晶屏的使用
3. 本程序在 SLAVR 上测试通过，硬件连接原理图请参见附录 A
4. 应包含头文件 slavr.h，在工程选项中的 Additional Lib 中填入 slavr
5. 硬件连接:
DS18B20 的 DQ 引脚连接 PA0 引脚，加 4.7K 上拉电阻
*********************************************/
#include "io8515v.h"
#include "slavr.h"
#pragma data:code            //设置数据区为程序存储器
const unsigned char tabel[]="0123456789ABCDEF";
#pragma data:data            //设置数据区回到数据存储器
/*******************************************
            延时函数
*******************************************/
void delay_us(int time)
   {
   do
      time--;
   while (time>1);
```

```
    }
/*********************************************
        检查 DS1820 是否存在:
如果 DS1820 不存在返回 1，否则返回 0
*********************************************/
unsigned char ds1820_ack(void)
    {
    unsigned char ack;
    DDRA|=0x01;                          //PORTA.0 输出
    PORTA&=0xfe;                         //输出 LOW
    delay_us(500);
    PORTA|=0x01;                         //输出 HIGH
    DDRA&=0xfe;                          //PORTA.0 输入
    delay_us(45);
    ack=PINA;
    ack&=0x01;
    delay_us(500);
    return ack;
    }
/*********************************************
        从 DS1820 读 1 字节数据
*********************************************/
unsigned char ds1820_read(void)
    {
    unsigned char i,byte,temp;
    byte=0x00;
    for (i=0;i<8;i++)
        {
        DDRA|=0x01;                      //PORTA.0 输出
        PORTA&=0xfe;                     //输出 LOW
        delay_us(3);
        PORTA|=0x01;                     //输出 HIGH
        DDRA&=0xfe;                      //PORTA.0 输入
        delay_us(10);
        temp=(PINA&0x01);
        if (temp!=0x00)
            byte|=0x80;
        if (i<7)
            byte=byte>>1;
        delay_us(100);
        DDRA|=0x01;                      //PORTA.0 输出
        delay_us(5);
```

```
        }
    return byte;
    }
/*********************************************
        写 ROM 或存储器命令到 DS1820
*********************************************/
void ds1820_write(unsigned char cmd)
    {
    unsigned char i,j;
    DDRA|=0x01;                          //PORTA.0 输出
    for (j=0;j<=7;j++)
        {
        i=cmd&0x01;
        if (i==0x01)
            {
            PORTA&=0xfe;                 //输出 LOW
            delay_us(10);
            PORTA|=0x01;                 //输出 HIGH
            delay_us(100);
            }
        else
            {
            PORTA&=0xfe;                 //输出 LOW
            delay_us(100);
            PORTA|=0x01;                 //输出 HIGH
            delay_us(10);
            }
        cmd=cmd>>1;
        }
    }
/*********************************************
            CRC 校验
*********************************************/
unsigned char crccheck(unsigned char *p,unsigned char len)
    {
    unsigned char bit0,cbit,r,temp,i,j,byte;
    temp=0;
    for (j=0;j<len;j++)
        {
        byte=p[j];
        for(i=0;i<8;i++)
            {
```

```
            cbit=temp&0x01;
            bit0=byte&0x01;
            temp=temp>>1;
            r=cbit^bit0;
            if(r==1)
            temp=temp^0x8c;
            byte=byte>>1;
            }
        }
    return temp;
    }
/*********************************************
            匹配DS1820
*********************************************/
void ds1820_match(unsigned char *p)
    {
    unsigned char i;
    do
        i=ds1820_ack();
    while (i==0x01);
    ds1820_write(0x55);
    for(i=0;i<8;i++)
        ds1820_write(p[i]);
    }
/*********************************************
  将十六进制数转换成ASCII码,并送LCD显示
*********************************************/
void asc2con(unsigned char *p,unsigned char len)
    {
    unsigned char i,temp;
    for (i=0;i<len;i++)
        {
        temp=p[len-1-i]&0xf0;
        temp=temp>>4;
        lcd_putc(tabel[temp]);
        temp=p[len-1-i]&0x0f;
        lcd_putc(tabel[temp]);
        }
    }
/*********************************************
    演示单总线设备工作过程
*********************************************/
```

```
void main(void)
    {
    unsigned char i,crc;
    unsigned char ds1820[9];          //存放从 DS18B20 读取的 9 字节数据
    unsigned char rom_code[8];        //存放 DS18B20 的 64 位序列号
    unsigned char *asc2p;
    unsigned int ds1820_temp;         //存放温度读数
    MCUCR=0x00;
    lcd_init();                       //LCD 初始化
    lcd_clear();                      //LCD 清屏
    lcd_gotoxy(0,0);
    lcd_puts("SERIAL NUMBER IS");     //在(0,0)处输出字符串
    do
        i=ds1820_ack();               //对 DS18B20 进行复位
    while (i==0x01);
    ds1820_write(0x33);               //写入 0x33ROM 命令
    for (i=0;i<8;i++)
        rom_code[i]=ds1820_read();    //读出 64 位序列号
    asc2p=&rom_code[0];
    crc=crccheck(asc2p,8);            //计算 CRC 校验和
    if (crc==0x00)                    //CRC 校验正确，显示序列号
        {
        lcd_gotoxy(0,1);
        asc2p=&rom_code[0];
        asc2con(asc2p,8);
        }
    delay_ms(2000);                   //延时
    while (1)
        {
        asc2p=&rom_code[0];
        ds1820_match(asc2p);          //匹配 DS1820
        ds1820_write(0x44);           //启动 DS1820 转换
        delay_ms(900);                //延时等转换结束，12 位格式约需 750ms
        asc2p=&rom_code[0];
        ds1820_match(asc2p);          //匹配 DS1820
        ds1820_write(0xbe);           //读转换后温度值
        for (i=0;i<9;i++)
            ds1820[i]=ds1820_read();
        lcd_clear();
        lcd_puts("Thermometer ");
        asc2p=&ds1820[0];
        crc=crccheck(asc2p,9);        //计算 CRC 校验和
```

```
        if (crc==0x00)                          //CRC 校验正确，输出温度读数并显示 HEX 数
            {
            ds1820_temp=ds1820[1];
            ds1820_temp=ds1820_temp<<8;
            ds1820_temp+=ds1820[0];
            asc2p=&ds1820[0];
            asc2con(asc2p,2);
            }
        lcd_gotoxy(0,1);
        if ((ds1820_temp&0xf800)!=0)            //判断是否小于 0 度
            {
            ds1820_temp=0-ds1820_temp;
            lcd_puts("-");
            }
        else
            lcd_puts("+");
        crc=ds1820_temp/16;                     //温度读数转换
        for(i=0;i<3;i++)
            {
            ds1820[2-i]=tabel[crc%10];
            crc=crc/10;
            }
        ds1820[3]='.';
        crc=ds1820_temp%16;
        crc=crc*10;
        ds1820[4]=tabel[crc/16];
        ds1820[5]=0xdf;
        ds1820[6]='C';
        for (i=0;i<7;i++)                       //显示温度
        lcd_putc(ds1820[i]);
        }
    }
```

5.9　用 LCD 显示中文及图形

```
/*********************************************
Project              :利用 LCD 显示中文及图形
Chip type            :AT90S8515
Clock frequency      :8.000000 MHz
```

```
Author              :詹卫前
Rework              :沈文
From                :广州双龙电子
Comments            :
1. 学习在普通的 LCD 上显示图形及中文点阵
2. 学习使用液晶模块自带的点阵汉字库(OCMJ4X8 液晶模块)
3. 为了节约篇幅，本例省略了图形和汉字点阵数据，如果读者需要点阵数据，可到互联网上下载生成点阵字模的软件，用该软件生成所需的字模
4. 本程序在 SLAVR 上测试通过，硬件连接原理图请参见附录 A
5. 应包含头文件 slavr.h，在工程选项中的 Additional Lib 中填入 slavr
6. 硬件连接:
数据线 D0~D7 接 PORTA
      ASK=PORTD.2
      ANSWER=PORTD.3
      RESET=PORTC.3
*********************************************/
#include "io8515v.h"
#include "slavr.h"
/*   定义 OCMJ4X8 端口   */
#define data_port PORTA                          //DATA_PORT=PORTA
#define ask (PIND&0x04)                          //ASK=PORTD.2
#define set_answer asm("sbi 0x12,3")             //ANSWER=PORTD.3
#define clr_answer asm("cbi 0x12,3")
#define set_reset asm("sbi 0x15,3")              //RESTE=PORTC.3
#define clr_reset asm("cbi 0x15,3")
#define wait asm("nop\n nop")
#pragma data:code                                //设置数据区为程序存储器
const unsigned char sl_ico[128]={};              //双龙公司图标，数据略，32×32 点阵
const unsigned char atmel_bmp[637]={};           // ATMEL 图标，数据略，104×49 点阵
const unsigned char hz1[480]={};                 //“和”字，数据略，64×64 点阵
const unsigned char hz2[132]={};                 //“欢”字，数据略，32×32 点阵
const unsigned char hz3[132]={};                 //“迎”字，数据略，32×32 点阵
const unsigned char hz4[132]={};                 //“您”字，数据略，32×32 点阵
#pragma data:data                                //设置数据区回到数据存储器
void ocmj_init(void)                             //LCD 模块初始化
   {
   MCUCR=0;
   DDRA=0xff;
   PORTA=0xff;
   DDRC=0b00001000;
   PORTC=0xff;
   DDRD=0b00001000;
```

```
    PORTD=0xff;
    clr_reset;                              //LCD 复位
    delay_ms(10);
    set_reset;
    clr_answer;
    delay_ms(10);
    }
void ocmj_write(unsigned char data)     //写数据到 LCD
    {
    while(ask!=0)
    ;
    data_port=data;
    wait;
    set_answer;
    wait;
    while(ask==0)
    ;
    clr_answer;
}
/*  传送 32*32 点阵数据到 LCD 模块，x:0~15(字节为单位)，y:0~64，ico_p:指向图形数据
的指针  */
void ico_tran(unsigned char x,unsigned char y,const unsigned char *ico_p)
  {
  unsigned char i,j;
  x+=4;
  for(j=0;j<32;j++)
      {
      for(i=0;i<4;i++)
          {
          ocmj_write(0xf3);
          ocmj_write(x);
          ocmj_write(y);
          ocmj_write(*ico_p);
          ico_p++;
          x++;
          }
      x-=4;
      y++;
      }
      }
/*  传送 bmp 点阵数据到 LCD，x:0~15(字节为单位)，y:0~64，bmp_p:指向图形数据的指针，
high:位图高度(点阵行为单位)，width:位图宽度(以字节为单位)    */
```

```
void bmp_tran(unsigned char x,unsigned char y,unsigned char high,unsigned char width,
              const unsigned char *bmp_p)
    {
    unsigned char i,j;
    x+=4;
    for(j=0;j<high;j++)
        {
        for(i=0;i<width;i++)
            {
            ocmj_write(0xf3);
            ocmj_write(x);
            ocmj_write(y);
            ocmj_write(*bmp_p);
            bmp_p++;
            x++;
            }
        x-=width;
        y++;
        }
    }
/*   传送 LCD 内部汉字数据到 LCD, x:0x00~0x07, y:0x00~0x03   */
void hz_tran(unsigned char x,unsigned char y,unsigned char *hz_p)
    {
    x+=2;
    while((*hz_p)!=0)
        {
        ocmj_write(0xf0);
        ocmj_write(x);
        ocmj_write(y);
        ocmj_write(*hz_p-0xa0);
        hz_p++;
        ocmj_write(*hz_p-0xa0);
        hz_p++;
        if(x<0x09)
            x++;
        else
            {
            x=0x02;
            y++;
            }
        }
    }
```

```
/*    传送ASCII字母到LCD，x:0x00~0x0f，y:0~64    */
void asc_tran(unsigned char x,unsigned char y,unsigned char *asc_p)
    {
    x+=4;
    while((*asc_p)!=0)
        {
        ocmj_write(0xf1);
        ocmj_write(x);
        ocmj_write(y);
        ocmj_write(*asc_p);
        asc_p++;
        if (x<0x13)
            x++;
        else
            {
            x=0x04;
            y+=8;
            }
        }
    }
/*          画点函数              */
void pset(unsigned char x,unsigned char y)
    {
    ocmj_write(0xf2);
    ocmj_write(x+32);
    ocmj_write(y);
    }
/*          画线函数          */
void line(unsigned char x1,unsigned char y1,unsigned char x2,unsigned char y2)
    {
    unsigned char t;
    signed int xerr=0,yerr=0;
    signed int delta_x,delta_y,distance;
    signed char incx,incy;
    /* 计算两个方向的长度 */
    delta_x=x2-x1;
    delta_y=y2-y1;
    /* 计算增量的方向,增量为"0"表示为垂直或水平线 */
    if(delta_x>0)
        incx=1;
    else
        {
```

```
        if( delta_x==0 )
        incx=0;
    else
        {
        incx=-1;
        delta_x =-delta_x;
        }
    }
if(delta_y>0)
    incy=1;
else
    {
    if( delta_y==0 )
        incy=0;
    else
        {
        incy=-1;
        delta_y =-delta_y;
        }
    }
/*  确定画线的范围 */
if( delta_x > delta_y )
     distance=delta_x;
else
     distance=delta_y;
/* 画线 */
for(t=0;t<= distance+1;t++)
     {
     ocmj_write(0xf2);
     ocmj_write(x1+32);
     ocmj_write(y1);
     xerr+=delta_x ;
     yerr+=delta_y ;
     if( xerr > distance )
          {
          xerr-=distance;
          x1+=incx;
          }
     if( yerr > distance )
          {
          yerr-=distance;
          y1+=incy;
```

```
            }
        }
    }
/*              画圆函数                */
void circle(unsigned char x0,unsigned char y0,unsigned char r)
    {
    unsigned char x,y;
    unsigned int xx,rr,xt,yt,rs;
    yt=r;
    rr=r*r+1;                           //补偿 1 修正方形
    rs=yt*3/4;                          //画 8 分之 一圆弧
    for (xt=0;xt<=rs;xt++)
         {
         xx=xt*xt;
         while ((yt*yt)>(rr-xx))
         yt--;
         x=x0+xt;                       //第一象限
         y=y0-yt;
         pset(x,y);
         x=x0-xt;                       //第二象限
         pset(x,y);
         y=y0+yt;                       //第三象限
         pset(x,y);
         x=x0+xt;                       //第四象限
         pset(x,y);
         /******45 度镜像画另外 8 分之 一圆弧******/
         x=x0+yt;                       //第一象限
         y=y0-xt;
         pset(x,y);
         x=x0-yt;                       //第二象限
         pset(x,y);
         y=y0+xt;                       //第三象限
         pset(x,y);
         x=x0+yt;                       //第四象限
         pset(x,y);
         }
    }
/*      MAIN 入口          */
void main(void)
    {
    unsigned char i;
    ocmj_init();                        //初始化
```

```
while(1)
    {
    ocmj_write(0xf4);                    //清屏
    ico_tran(0,0,sl_ico);                //传送图标,也可用 bmp_tran()传送
    hz_tran(3,0,"广州双龙");             //传送内部汉字
    hz_tran(2,1,"电子有限公司");
    asc_tran(5,39,"HTTP://");            //传送 ASCII 字符
    asc_tran(2,50,"WWW.SL.COM.CN");
    delay_ms(2000);                      //延时
    ocmj_write(0xf4);                    //清屏
    bmp_tran(4,2,60,8,hz1);              //传送汉字"和"
    delay_ms(3000);
    ocmj_write(0xf4);
    bmp_tran(1,3,49,13,atmel_bmp); //传送 ATMEL 标志图形
    asc_tran(2,54,"WWW.ATMEL.COM");
    delay_ms(3000);
    ocmj_write(0xf4);
    bmp_tran(2,15,33,4,hz2);             //传送汉字"欢"
    bmp_tran(6,15,33,4,hz3);             //传送汉字"迎"
    bmp_tran(10,15,33,4,hz4);            //传送汉字"您"
    hz_tran(0,0,"恭");
    hz_tran(0,1,"祝");
    hz_tran(0,2,"各");
    hz_tran(0,3,"位");
    hz_tran(7,0,"万");
    hz_tran(7,1,"事");
    hz_tran(7,2,"如");
    hz_tran(7,3,"意");
    delay_ms(3000);
    for(i=0;i<32;i++)
        {
        ocmj_write(0xf4);
        line(i*2,32,64,i);
        line(64,i,(127-i*2),32);
        line((127-i*2),32,64,(63-i));
        line(64,(63-i),i*2,32);
        delay_ms(100);
        }
    for(i=0;i<32;i++)
        {
        ocmj_write(0xf4);
        circle(63,31,i);
```

```
            delay_ms(100);
            }
        }
    }
```

5.10　多通道 A/D 变换

```
/*********************************************
Project              :多通道 A/D 转换演示程序
Chip type            :ATmega8
Clock frequency      :内部 RC(INT) 8.000000 MHz
Author               :詹卫前
Rework               :沈文
From                 :广州双龙电子
Comments             :
1. 学习使用 ATmega8 内部的 A/D 转换器
2. 学习使用动态扫描数码管
3. 本程序在 SL-MEGA8 上测试通过
4. 硬件连接:
使用内部 RC 振荡，PB6-G，PB7-DP 短路块连接，INT0/INT1 按键切换 ADC 通道
详见附录 B
*********************************************/
#include "iom8v.h"
#include "macros.h"
#define osccal 0x7d                          //内部 RC 校正常数
#define Vref   500                           //参考电压值
unsigned int adc_rel;                        //AD 转换结果
unsigned char adc_mux;                       //AD 通道
unsigned char led_buff[4]={0,0,0,0};
#pragma data:code                            //设置数据区为程序存储器
const unsigned char seg_table[16]={0xc0,0xf9,0xa4,0xb0,0x99,0x92,0x82,0xf8,0x80,
                                   0x90,0x88,0x83,0xc6,0xa1,0x86,0x8e};
#pragma data:data                            //设置数据区回到数据存储器
void delay_us(int time)                      //微秒级延时程序
    {
    do
    time--;
    while (time>1);
    }
```

```
void delay_ms(unsigned int time)          //毫秒级延时程序
    {
    while(time!=0)
        {
        delay_us(1000);
        time--;
        }
    }
void display(void)
    {
    unsigned char i;
    DDRB=0xff;
    PORTB=0xff;
    DDRD|=0xf0;
    PORTD|=0xf0;
    for(i=0;i<4;i++)
        {
        PORTB=led_buff[i];
        PORTD&=~(1<<(i+4));
        delay_ms(1);
        PORTD|=0xf0;
        }
    }
void adc_init(void)                       //ADC 初始化
    {
    DDRC=0x00;
    PORTC=0x00;
    ADCSRA=0x00;
    ADMUX=(1<<REFS0)|(adc_mux&0x0f);      //选择内部 AVCC 为基准
    ACSR=(1<<ACD);                        //关闭模拟比较器
    ADCSRA=(1<<ADEN)|(1<<ADSC)|(1<<ADIE)|(1<<ADPS2)|(1<<ADPS1) ;//64 分频
    }
#pragma interrupt_handler adc_isr:15      //ADC 完成中断
void adc_isr(void)
    {
    adc_rel=ADC&0x3ff;
    ADMUX=(1<<REFS0)|(adc_mux&0x0f);      //选择内部 AVCC 为基准
    ADCSRA|=(1<<ADSC);                    //启动 AD 转换
    }
void ADCtoBCD(unsigned int temp)          //ADC 结果转换成电压值
    {
    unsigned char i;
```

```
    temp=(unsigned int)(((unsigned long)((unsigned long)temp*Vref))/0x3ff);
    for(i=0;i<4;i++)
        {
        led_buff[i]=seg_table[temp%10];
        temp=temp/10;
        }
    led_buff[2]&=0x7f;
    }
void main(void)
    {
    unsigned char i;
    unsigned int adc_old;
    DDRD=0xf0;
    PORTD=0xff;
    OSCCAL=osccal;                          //校正振荡频率
    adc_mux=0;
    adc_init();
    SEI();
    for(i=0;i<4;i++)
        led_buff[i]=seg_table[8];
    for(i=0;i<200;i++)
        display();
    adc_old=0;
    adc_rel=0;
    while(1)
        {
        if(adc_old!=adc_rel)
            {
            adc_old=adc_rel;
            ADCtoBCD(adc_old);
            }
        display();
        i=PIND&0x0c;                        //判断按键
        if(i!=0x0c)
            {
            display();                      //按键消抖，用显示函数代延时函数
            if(i==(PIND&0x0c))
                {
                CLI();
                adc_rel=0;
                adc_old=0;
                if (i==0x08)                //INT0 键按下
```

```
                    {
                    if(adc_mux<3)
                        adc_mux++;    //递增
                    else
                        adc_mux=0;
                    }
                if (i==0x04)              //INT1 键按下
                    {
                    if(adc_mux>0)
                        adc_mux--;    //递减
                    else
                        adc_mux=3;
                    }
                led_buff[0]=seg_table[adc_mux];
                led_buff[1]=0xbf;         //-
                led_buff[2]=0b10001001;  //h
                led_buff[3]=0b11000110;  //C
                while((PIND&0x0c)!=0x0c)  //检查按键释放
                display();
                SEI();
                }
            }
        }
    }
```

5.11 A/D 和 D/A 变换

```
/*********************************************
Project              :用 ADC 和 DAC 演示程序
Chip type            :ATmega8
Clock frequency      :内部 RC(INT) 8.000000 MHz
Author               :詹卫前
Rework               :沈文
From                 :广州双龙电子
Comments             :
```

1. 学习使用 ATmega8 内部的 PWM 功能
2. 学习使用通过 RS232 与 PC 机进行通信
3. 本程序在 SL-MEGA8 上测试通过
4. 硬件连接：

```
PB6-G，PB7-DP 短路块连接
使用 INT0/INT1 按键改变 DAC 输出电压大小
使用 ADC0 通道测量 DAC 的输出电压
SL-MEGA8 原理图详见附录 B
5. PC 机设置说明
(1) 使用 ICCAVR 的终端调试窗口(Terminal)，进行通信调试
(2) 设置串口为 com1(或 com2)，通信波特率为 19200(Tools->Environment Options)
(3) 将 PC 屏幕光标定位于调试窗口中
*********************************************/
#include "iom8v.h"
#include "macros.h"
#define fosc 8000000                          //晶振 8MHz
#define baud 19200                            //波特率
#define osccal 0x7d                           //内部 RC 校正常数
#define Vref  500                             //参考电压值
unsigned int adc_rel;                         //AD 转换结果
unsigned char adc_mux;                        //AD 通道
unsigned int adc_old;
void delay_us(int time)                       //微秒级延时程序
    {
    do
    time--;
    while (time>1);
    }
void delay_ms(unsigned int time)              //毫秒级延时程序
    {
    while(time!=0)
        {
        delay_us(1000);
        time--;
        }
    }
void adc_init(void)                           //ADC 初始化
    {
    DDRC=0x00;
    PORTC=0x00;
    ADCSRA=0x00;
    ADMUX=(1<<REFS0)|(adc_mux&0x0f);          //选择内部 AVCC 为基准
    ACSR=(1<<ACD);                            //关闭模拟比较器
    ADCSRA=(1<<ADEN)|(1<<ADSC)|(1<<ADIE)|(1<<ADPS2)|(1<<ADPS1) ;//64 分频
    }
#pragma interrupt_handler adc_isr:15          //ADC 完成中断
```

```
void adc_isr(void)
    {
    static unsigned i;
    adc_rel+=ADC&0x3ff;
    ADMUX=(1<<REFS0)|(adc_mux&0x0f);      //选择内部 AVCC 为基准
    ADCSRA|=(1<<ADSC);                    //启动 AD 转换
    if (i<63)
         i++;
    else
         {
         adc_old=adc_rel/64;              //64 点平均滤波
         i=0;
         adc_rel=0;
         }
    }
void putchar(unsigned char c)             //字符输出函数
    {
    while (!(UCSRA&(1<<UDRE)));
    UDR=c;
    }
unsigned char getchar(void)               //字符输入函数
    {
    while(!(UCSRA& (1<<RXC)));
    return UDR;
    }
void puts(char *s)                        //字符串输出函数
    {
    while (*s)
       {
       putchar(*s);
       s++;
       }
    putchar(0x0a);                        //回车换行
    putchar(0x0d);
    }
void uart_init(void)                      //UART 初始化
    {
    UCSRB=(1<<RXEN)|(1<<TXEN);            //允许发送和接收
    UBRRL=(fosc/16/(baud+1))%256;
    UBRRH=(fosc/16/(baud+1))/256;
    UCSRC=(1<<URSEL)|(1<<UCSZ1)|(1<<UCSZ0);      //8 位数据+1 位 STOP 位
    }
```

```
void timer1_init(void)                          //T1 初始化程序
    {
    TCCR1B=0;                                   //stop
    TCNT1H=0;
    OCR1A=0;
    TCCR1A=(1<<WGM11)|(1<<WGM10)|(1<<COM1A1);
    TCCR1B=(1<<CS10);//(1<<WGM13)|(1<<WGM12)|(1<<CS10);
    }
//两个中断向量使用同一个中断处理函数，实现对 INT0/INT1 按键的处理
#pragma interrupt_handler int_isr:2 int_isr:3
void int_isr(void)
    {
    unsigned char i;
    delay_ms(10);
    i=PIND&0x0c;                                //键盘消抖动
    if(i==0x0c)
        return;
    else
        {
        if (i==0x08)                            //INT0 键按下
            {
            if(OCR1A<0x3ef)
                OCR1A+=0x10;                    //递增
            else
                OCR1A=0x3ff;
            }
        if (i==0x04)                            //INT1 键按下
            {
            if(OCR1A>0x10)
                OCR1A-=0x10;                    //递减
            else
                OCR1A=0;
            }
        while((PIND&0x0c)!=0x0c)                //检查按键释放
        ;
        }
    }
void main(void)                                 //MAIN 程序
    {
    unsigned char i;
    unsigned int temp;
    DDRB=(1<<PB1);
```

```
DDRD=0xf0;
PORTD=0xff;
OSCCAL=osccal;
adc_mux=0;
uart_init();
timer1_init();
adc_init();
GICR=0xC0;                              //int0、int1 中断使能
SEI();
adc_rel=0;
while(1)
    {
    i=getchar();
    switch (i)
        {
        case 'S':                       //MEGA8 DAC 程序 ID
            {
            puts("DAC TEST ver1.0");
            break;
            }
        case 'T':                       //调节输出电压
            {
            temp=getchar();
            temp=temp<<8;
            temp|=getchar();
            if (temp<0x400)
                 OCR1A=temp;
            else
                 OCR1A=0x3ff;
            break;
            }
        case 'O':                       //读取 OCR1A 寄存器值
            {
            temp=OCR1A;
            putchar(temp/256);
            putchar(temp%256);
            break;
            }
        case 'M':                   //设置 ADC 工作通道
            {
            adc_mux=getchar()&0x03;
            break;
```

```
            }
        case 'R':                          //读取 ADC 通道及 ADC 转换结果
            {
            putchar(adc_mux);
            putchar(adc_old/256);
            putchar(adc_old%256);
            }
        }
    }
}
```

5.12　利用 PWM 方式产生双音频信号

```
/*********************************************
Project              :利用 PWM 方式产生双音频信号(DTMF)
Chip type            :AT90S8515
Clock frequency      :8.000000 MHz
Author               :詹卫前
Rework               :沈文
From                 :广州双龙电子
Comments             :
1. 学习使用 PWM 产生所需的双音频信号
2. 学习按键扫描方式编程
3. 本程序在 SLAVR 上测试通过，硬件连接原理图请参见附录 A
4. 硬件连接：
喇叭连接：PD5(OC1A)--1K 电阻--喇叭
键盘连接：
  1     2     3     A  --  PB4
  4     5     6     B  --  PB5
  7     8     9     C  --  PB6
  *     0     #     D  --  PB7
  |     |     |     |
 PB0   PB1   PB2   PB3
*********************************************/
#include "io8515v.h"
#include "macros.h"
#define Xtal 8000000                       //系统时钟频率
#define prescaler 1                        //T1 预分频系数
#define N_samples 128                      //在查找表中的样本数
```

```
#define Fck Xtal/prescaler                    //T1 工作频率
#define delaycyc 10                           //读取 port C 口延时循环数
/*************************** 正弦表 ****************************
      样本表：一个周期分成 128 个点，每点按 7 位进行量化
*****************************************************************/
#pragma data:code                             //设置数据区为程序存储器
const unsigned char auc_SinParam [128]={
64,67,70,73,76,79,82,85,88,91,94,96,99,102,104,106,109,111,113,115,117,118,
120,121,123,124,125,126,126,127,127,127,127,127,127,127,126,126,125,124,
123,121,120,118,117,115,113,111,109,106,104,102,99,96,94,91,88,85,82,79,
76,73,70,67,64,60,57,54,51,48,45,42,39,36,33,31,28,25,23,21,18,16,14,12,10,9,
7,6,4,3,2,1,1,0,0,0,0,0,0,0,1,1,2,3,4,6,7,9,10,12,14,16,18,21,23,25,28,31,33,36,
39,42,45,48,51,54,57,60};
/*************************** x_SW ***************************
x_SW 表(8 倍)：x_SW=ROUND(8*N_samples*f*510/Fck)
*****************************************************************/
//高频(列)
//1209Hz  ---> x_SW=79
//1336Hz  ---> x_SW=87
//1477Hz  ---> x_SW=96
//1633Hz  ---> x_SW=107
const unsigned char auc_frequencyH [4]={107,96,87,79};
//低频(行)
//697Hz  ---> x_SW=46
//770Hz  ---> x_SW=50
//852Hz  ---> x_SW=56
//941Hz  ---> x_SW=61
const unsigned char auc_frequencyL [4]={61,56,50,46};
#pragma data:data                             //设置数据区回到数据存储器
//*************************** 全局变量 ****************************
unsigned char x_SWa=0x00;                     //高频信号脉冲宽度
unsigned char x_SWb=0x00;                     //低频信号脉冲宽度
unsigned int  X_LUTaExt=0;
unsigned int  X_LUTbExt=0;
unsigned int  X_LUTa;
unsigned int  X_LUTb;
/*****************************************************************
                  定时器溢出中断服务程序
*****************************************************************/
#pragma interrupt_handler ISR_T1_Overflow:7
void ISR_T1_Overflow (void)
   {
```

```
    X_LUTaExt+=x_SWa;
    X_LUTbExt+=x_SWb;
    X_LUTa=(char)(((X_LUTaExt+4)>>3)&(0x007F)); //数据规格化
    X_LUTb=(char)(((X_LUTbExt+4)>>3)&(0x007F));
                                                //计算 PWM 值:高频值+3/4 低频值
    OCR1A=(auc_SinParam[X_LUTa] + (auc_SinParam[X_LUTb]-
          (auc_SinParam[X_LUTb]>>2)));
    }
/*************************************************************
                        初始化
*************************************************************/
void init (void)
    {
    MCUCR=0x00;
    TIMSK=0x80;                          //T1 溢出中断使能
    TCCR1A=(1<<COM1A1)+(1<<PWM10);       //不翻转、8 位 PWM
    TCCR1B=(1<<CS10);                    //预分频系数为 1、即 CLK/1
    DDRD=(1 <<PD5);                      //PD5(OC1A)用作输出
    SEI();                               //全局中断使能
    }
/*************************************************************************
     为从 PORT C 口读取稳定的按键数据，所必须的延时程序(消抖延时)
*************************************************************************/
void Delay (void)
    {
    int i;
    for (i = 0; i < delaycyc; i++)
         NOP();
    }
/*************************************************************************
                主程序从 PORT C 口读取按键数据，来确定产生哪个
          高频(列)和低频(行)信号的混合信号，并且修正 x_SWa 和 x_SWb。
                              行  -> PINC 高四位
                              列  -> PINC 低四位
*************************************************************************/
void main (void)
    {
    unsigned char uc_Input;
    unsigned char uc_Counter=0;
    init();
    for(;;)
        {                                //高四位 - 行
```

```
        DDRC=0x0F;                          //高四位输入、低四位输出
        PORTC=0xF0;                         //高四位打开上位、低四位输出低电平
        uc_Counter=0;
        Delay();                            //延时等待 Port C 电平稳定
        uc_Input=PINC;                      //读取 Port C
        do
          {
          if(!(uc_Input & 0x80))            //检查 MSB 是否为低
               {                            //取低音脉冲宽度并结束循环
               x_SWb=auc_frequencyL[uc_Counter];
               uc_Counter=4;
               }
          else
               x_SWb=0;                     //没有频率调制要求
          uc_Counter++;
          uc_Input=uc_Input<<1;             //左移一位
          }
        while ((uc_Counter<4));             //低四位-列
        DDRC=0xF0;                          //高四位输出、低四位输入
        PORTC=0x0F;                         //高四位输出低电平、低四位打开上拉
        uc_Counter=0;
        Delay();                            //延时等待 Port C 电平稳定
        uc_Input=PINC;
        uc_Input=uc_Input<<4;
        do
          {
          if(!(uc_Input & 0x80))            //检查 MSB 是否为低
               {
               x_SWa=auc_frequencyH[uc_Counter];//取高音脉冲宽度并结束循环
               uc_Counter=4;
               }
          else
               x_SWa=0;
          uc_Counter++;
          uc_Input=uc_Input<<1;
          }
        while (uc_Counter<4);
    }
}
```

5.13　通过 UART 使用 PC 机键盘

```
/*********************************************
Project                 :通过 UART 使用 PC 机键盘
Chip type               :AT90S8515
Clock frequency         :8.000000 MHz
Author                  :詹卫前
Rework                  :沈文
From                    :广州双龙电子
Comments                :
```

1. 学习使用 AT90S8515 的 UART 与 PS/2 键盘通信
2. 本程序在 SLAVR 上测试通过，硬件连接原理图请参见附录 A
3. 硬件连接：

键盘的 data	PD3
键盘的 clock	PD2
D232.T	PD1
D232.R	PD0

注意：PD0 和 PD1 不能直接与计算机的 RS232 口相连，应进行电平转换(如 MAX232)。

4. PC 机键盘简介

(1) 键盘插头：键盘插头有两种：大插头(5 芯)和小插头(6 芯)。两种键盘插头的外形图及各插头接线定义如图 5.1 和 5.2 所示。

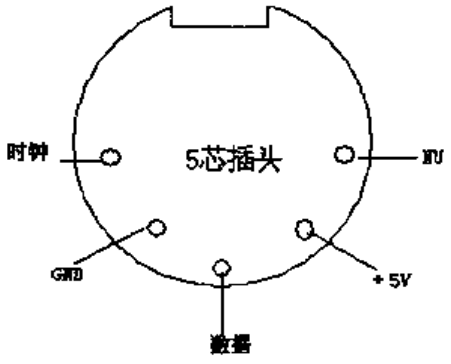


图 5.1　芯 PC 键盘插头

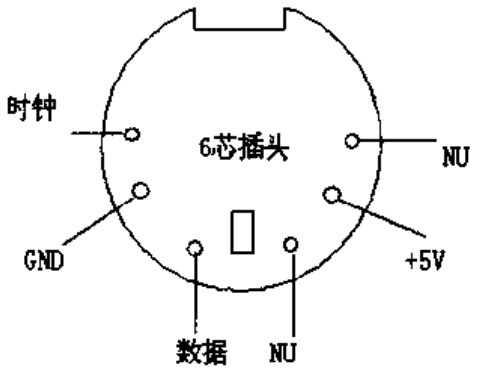


图 5.2　芯 PC 键盘插头

(2) 协议：标准键盘与计算机采用双向通信方式，键盘可以发送数据给计算机，计算机也可以发送命令给键盘，计算机有最高优先权，可以在任何时候发命令给键盘。通信协议是标准的串行异步通信格式。1 位起始位，8 位数据位(LSB 在先)，1 位奇校验位 P，最后是 1 位停止位。

(3) 扫描码：键盘电路正常工作时不断地扫描键盘矩阵。有按键，则确定按键位置之后以串行数据形式发送按键的扫描码给系统板键盘接口电路。按键的扫描码由接通扫描码和断开扫描码两部分组成。断开扫描码是在接通扫描码前加一个断开标志 F0 字节形成。键按下时，发送接通扫描码，简称通码；键松

开时，则发送该键的断开扫描码，简称断码。若一直按下某键，则以按键重复率连续发送该键的接通扫描码。例如“c”键的接通扫描码是 0x1c，断开扫描码是 0xF01c。“c“键被按下时，0x1c 被发送出去，如果按住不放，则以按键重复率连续发送 0x1c，直到该键释放，才发出断开扫描码 0xF01c。扫描码与按键的位置有关，与该键的 ASCII 码并无对应关系。

(4) 命令

PC 机发送给键盘的命令

0xFF	KB_RESET	复位键盘
0xFE	KB_RESEND	重新发送
0xF7~0xFD		空操作(不使用)
0xF6		设置默认值
0xF5		设置默认值和停止键盘
0xF4	KB_ENABLE	启动键盘
0xF3	KB_MAKE_BREAK	设置拍发速率/延时参数
0xEF~F2H		空操作(不使用)
0xEE	KB_ECHO	回响命令
0xED	LED_CMD	置位/复位 LED 指示器

键盘的发送/回应 PC 机的命令

0xFE	KB_RESEND_ACK	重新发送响应
0xFA	KB_ACK	正常应答
0x00	KB_OVER_RUN	超限应答
0xFD	KB_FAL_ACK	诊断故障应答
0xAA	KB_OK	诊断正常应答
0xEE	KB_ECHO_ACK	对 ECHO 命令的应答
0xF0	KB_BREAK	断开扫描码前缀字节。断开扫描码占 2 个字节，第一个字节总是 0xF0，第二个字节与接通扫描码相同

```
**********************************************/
#include "io8515v.h"
#include "macros.h"
#define PIN_DIR DDRD
#define PIN_KB PIND
#define PORT_KB PORTD
#define CLOCK 3
#define DATAPIN 2
#define BUFF_SIZE 64
#pragma data:code                                    //设置数据区为程序存储器
const unsigned char unshifted[][2] = {  //没有按下 shifted 键的 PC 机键盘扫描码
0x0d,9,0x0e,'`',0x15,'q',0x16,'1',0x1a,'z',0x1b,'s',0x1c,'a',0x1d,'w',0x
1e,'2',0x21,'c',
0x22,'x',0x23,'d',0x24,'e',0x25,'4',0x26,'3',0x29,' ',0x2a,'v',0x2b,'f',0
x2c,'t',0x2d,'r',
0x2e,'5',0x31,'n',0x32,'b',0x33,'h',0x34,'g',0x35,'y',0x36,'6',0x39,',',
0x3a,'m',0x3b,'j',
```

```
    0x3c,'u',0x3d,'7',0x3e,'8',0x41,',',0x42,'k',0x43,'i',0x44,'o',0x45,'0',
0x46,'9',0x49,'.',
    0x4a,'/',0x4b,'l',0x4c,';',0x4d,'p',0x4e,'-',0x52,0x27,0x54,'[',0x55,'='
,0x5a,13, 0x5b,']',
    0x5d,0x5c,0x61,'<',0x66,8,0x69,'1',0x6b,'4',0x6c,'7',0x70,'0',0x71,',',0
x72,'2',0x73,'5',
    0x74,'6',0x75,'8',0x79,'+',0x7a,'3',0x7b,'-',0x7c,'*',0x7d,'9',0,0};
    const unsigned char shifted[][2]={ //按下 shifted 键的 PC 机键盘扫描码
    0x0d,9,0x0e,'~',0x15,'Q',0x16,'!',0x1a,'Z',0x1b,'S',0x1c,'A',0x1d,'W',0x
1e,'@',
    0x21,'C',0x22,'X',0x23,'D',0x24,'E',0x25,'$',0x26,'#',0x29,'',0x2a,'V',0
x2b,'F',
    0x2c,'T',0x2d,'R',0x2e,'%',0x31,'N',0x32,'B',0x33,'H',0x34,'G',0x35,'Y',
0x36,'^',
    0x39,'L',0x3a,'M',0x3b,'J',0x3c,'U',0x3d,'&',0x3e,'*',0x41,'<',0x42,'K',
0x43,'I',
    0x44,'O',0x45,')',0x46,'(',0x49,'>',0x4a,'?',0x4b,'L',0x4c,':',0x4d,'P',
0x4e,'_',
    0x52,'"',0x54,'{',0x55,'+',0x5a,13,0x5b,'}',0x5d,'|',0x61,'>',0x66,8,0x6
9,'1',
    0x6b,'4',0x6c,'7',0x70,'0',0x71,',',0x72,'2',0x73,'5',0x74,'6',0x75,'8',
0x79,'+',
    0x7a,'3',0x7b,'-',0x7c,'*',0x7d,'9',0,0};
    #pragma data:data                          //设置数据区回到数据存储器
    unsigned char bitcount;                    //PC 键盘数据长度计数
    unsigned char kb_buffer[BUFF_SIZE];        //键盘缓冲区
    unsigned char input=0;                     //缓冲区读指针
    unsigned char output=0;                    //缓冲区写指针
    //   键盘缓冲区使用软件模拟 FIFO
    void put_kbbuff(unsigned char c)           //送键盘按键 ASCII 码到键盘缓冲区
       {
       kb_buffer[input]=c;
       if (input<(BUFF_SIZE-1))
            input++;
       else
            input=0;
       }
    unsigned char get_char(void)               //从键盘缓冲区读取按键的 ASCII 码
       {
       unsigned char temp;
       if(output==input)
            return 0;
```

```
    else
        {
        temp=kb_buffer[output];
        if(output<(BUFF_SIZE-1))
            output++;
        else
            output=0;
        return temp;
        }
    }
void uart0_init(void)
    {
    UCR=0x00;
    UBRR=0x19;                          //BAUD=19200
    UCR|=(1<<TXEN) ;                    //使能发送
    }
void port_init(void)                    //端口初始化
    {
    DDRD=0x02;
    PORTD=0xff;
    }
void init_kb(void)                      //为运行读取 PC 键盘程序进行初始化
    {
    MCUCR=0x02;                         //设置 8515 的 INT0 为下降沿触发中断
    GIMSK|=(1<<INT0);                   //使能 INT0 中断
    SEI();                              //开中断
    bitcount=11;
    }
void decode(unsigned char sc)           //对 PC 键盘的扫描码进行解码
    {
    static unsigned char shift,up,shiftup;
    unsigned char i;
    if (sc==0xf0)                       //按键释放
        {
        up=1;
        return;
        }
    if (up==1)                          //SHIFT 键开关
        {
        up=0;
        if ((sc==0x12)|(sc==0x59))
            shift=0;
```

```
        return;
        }
switch (sc)
        {
        case 0x12:                    //检测左 SHIFT 键
            {
            shift=1;
            shiftup=1;
            }
        case 0x59:                    //检测右 SHIFT 键
            {
            shift=1;
            shiftup=1;
            }
        default:
            {
            if (shift==0)
                {
                for(i=0;unshifted[i][0]!=sc && unshifted[i][0]; i++)
                    ;
                if (unshifted[i][0]==sc)
                    put_kbbuff(unshifted[i][1]);
                }
            else
                {
                for(i=0;shifted[i][0]!=sc && shifted[i][0]; i++)
                    ;
                if (shifted[i][0]==sc)
                put_kbbuff(shifted[i][1]);
                }
            }
        }
#pragma interrupt_handler int0_isr:2    //键盘数据读取中断程序
void int0_isr(void)
    {
    static unsigned char data;
    switch (bitcount)
        {
        case 11:
            {
            if ((PIN_KB&(1<<DATAPIN))!=0)
```

```
                return;
            else
                bitcount--;
            break;
            }
        case 2:
            {
            bitcount--;
            break;
            }
        case 1:
            {
            bitcount--;
            if ((PIN_KB&(1<<DATAPIN))==0)
                {
                bitcount=11;
                return;
                }
            else
                {
                bitcount=11;
                decode(data);
                }
            break;
            }
        default:
            {
            data=(data>>1);
            if((PIN_KB&(1<<DATAPIN))!=0)
            data|=0x80;
            bitcount--;
            }
        }
    }
//PC 键盘演示程序，SHIFT 键和按键同时按下时为大写
void main(void)
    {
    unsigned char key;
    port_init();
    uart0_init();
    init_kb();
    puts("PC 键盘演示程序");
```

```
    putchar(0x0d);
    while(1)
        {
        key=get_char();
        if(key!=0)
            {
            if (key==0x0d)              //回车键处理
                {
                putchar('\n');
                putchar(key);
                }
            else
                putchar(key);
            }
        }
    }
```

5.14　ATmega8 的 boot 引导 IAP 应用

```
/*********************************************
Project              :ATmega8 的 boot 引导 IAP 应用
Chip type            :ATmega8
Clock frequency      :外部(EXT) 8.000000 MHz
Author               :詹卫前
Rework               :沈文
From                 :广州双龙电子
Comments             :
1. 学习使用 ATmega8 的 boot 引导 IAP 的应用
2. 学习使用 ATmega8 的 UART 通过 RS232 与 PC 机进行通信
3. 本程序在 SL-MEGA8 上测试通过
4. 硬件连接：SL-MEGA8 原理图详见附录 B
5. PC 机运行 AVRprog.exe 软件
6. 数据格式：8 位数据，无奇偶校验，1 个停止位，19.2Kbps 速率
*********************************************/
#include "iom8v.h"
#define fosc 8000000                    //晶振 8MHz
#define baud 19200                      //波特率
#define device 0x77                     //芯片代码
#define sig_byte1 0x1E                  //信号字节
```

```
#define sig_byte2 0x93
#define sig_byte3 0x07
#define PAGESIZE 64                                    //IAP 编程页面数量
#define APP_PAGES ((8192/PAGESIZE)-(1024/ PAGESIZE ))
#define APP_END APP_PAGES * PAGESIZE
/*由于 ATMEL 没有提供以下四个函数的源代码，因此本书只能给出下面四个函数的汇编代码，但并不影响对这四个函数的使用*/
void write_page (unsigned int adr, unsigned char function);
void fill_temp_buffer (unsigned int data,unsigned int adr);
unsigned int read_program_memory (unsigned int adr,unsigned char cmd);
void write_lock_bits (unsigned char val);
/*  由于在 boot 编程时，可能无中断，因此采用查询方式接收和发送数据  */
void sendchar(unsigned char c)                         //字符输出函数
    {
    while (!(UCSRA&(1<<UDRE)));
    UDR=c;
    }
unsigned char recchar(void)                            //字符输入函数
    {
    while(!(UCSRA& (1<<RXC)));
    return UDR;
    }
void uart_init(void)                                   //UART 初始化
    {
    OSCCAL=0x7d;                                       //对内部 RC OSC 调整
    UCSRB=(1<<RXEN)|(1<<TXEN);                         //允许发送和接收
    UBRRL=(fosc/16/(baud+1))%256;
    UBRRH=(fosc/16/(baud+1))/256;
    UCSRC=(1<<URSEL)|(1<<UCSZ1)|(1<<UCSZ0);            //8 位数据+1 位 STOP 位
    }
void iap(void)
    {
    unsigned int intval,address,data;
    unsigned char val, ldata;
    for(;;)
        {
        val=recchar();
        if(val=='a')                                   //自动地址增量
            sendchar('Y');                             //采用快速自动增量模式
        else if(val=='A')                              //设置地址
                {
                address=recchar();                     //读地址
```

```
        address=(address<<8)|recchar();
        address=address<<1;                    //将字地址转为字节地址
        sendchar('\r');
        }
else if(val=='c')                              //发送低位数据字节
        {
        ldata=recchar();
        sendchar('\r');
        }
else if(val=='C')                              //发送高位数据字节
        {
        data=ldata|(recchar()<<8);
        fill_temp_buffer(data,(address));
        address+=2;
        sendchar('\r');
        }
else if(val=='e')                              //擦除程序区
        {
                                               //程序区有 60 页
        for(address=0;address<APP_END;address+=PAGESIZE)
            write_page(address,0x03);          //擦除完成
        sendchar('\r');
        }
else if(val=='l')                              //写 lock 位
        {
        write_lock_bits(recchar());
        sendchar('\r');
        }
else if(val=='m')                              //写一页
        {
        write_page((address),0x05);
        sendchar('\r');
        }
else if((val=='P')||(val=='L'))                //进入编程模式
        sendchar('\r');
else if (val=='p')                             //允许编程
        sendchar('S');
else if(val=='R')                              //读程序存储器
        {
        if (address>=APP_END)
            {
            sendchar(0xff);
```

```
            sendchar(0xff);
            address+=2;
            }
        else
            {
            intval=read_program_memory(address,0x00);
            sendchar((char)(intval>>8));    //发送高位
            sendchar((char)intval);         //发送低位
            address+=2;
            }
        }
else if (val=='D')
        {
        EEARL=address;
        EEARH=(address >> 8);
        address++;
        EEDR=recchar();
        EECR|=(1<<EEMWE);
        EECR|=(1<<EEWE);
        while(EECR&(1<<EEWE))
           ;
        sendchar('\r');
        }
else if (val=='d')
        {
        EEARL=address;
        EEARH=(address>>8);
        address++;
        EECR|=(1<<EERE);
        sendchar(EEDR);
        }
else if(val=='F')                               //读 fuse 位
        sendchar(read_program_memory(0x0000,0x09));
else if(val=='r')                               //读 lock 位
        sendchar(read_program_memory(0x0001,0x09));
else if(val=='N')                               //读 fuse 高位
        sendchar(read_program_memory(0x0003,0x09));
else if(val=='t')                               //返回
        {
        sendchar(device);
        sendchar(0);
        }
```

```
        else if ((val=='x')||(val=='y')||(val=='T'))
                {
                recchar();
                sendchar('\r');
                }
        else if (val=='S')                          //返回软件标识
                {
                sendchar('A');
                sendchar('V');
                sendchar('R');
                sendchar('B');
                sendchar('O');
                sendchar('O');
                sendchar('T');
                }
        else if (val=='V')                          //返回软件版本
                {
                sendchar('1');
                sendchar('0');
                }
        else if (val=='s')                          //返回签名
                {
                sendchar(sig_byte3);
                sendchar(sig_byte2);
                sendchar(sig_byte1);
                }
        else if(val!=0x1b)
                {
                sendchar('?');
                }
        }
    }
void main(void)
    {
    void (*funcptr)( void )=0x0000;                 //设置函数指针
    DDRD=0;
    PORTD=0xff;
    uart_init();
    if ((PIND&0x04)==0)
        {
        PORTD=0;
        iap();
```

```
        }
    else
        funcptr();
    }
/*注意：后面四个函数并非 C 源程序，而是汇编语言程序，应另建一个新文件(.s 扩展名)并加入工程中一起编译*/
; void write_page (unsigned int adr, unsigned char function);
_write_page::
    in r20,0x37
    sbrc r20,0
    rjmp _write_page
    MOV R31,R17
    MOV R30,R16;              move adress to z pointer (R31=ZH R30=ZL)
    OUT 0x37,R18;             argument 2 decides function
    SPM;                      perform pagewrite
    .dw $ffff
    NOP
    RET
; void fill_temp_buffer (unsigned int data, unsigned int adr);
_fill_temp_buffer::
    mov r31,r19
    mov r30,r18;              move adress to z pointer (R31=ZH R30=ZL)
    mov r1,r17
    mov r0,r16;               move data to reg 0 and 1
    ldi r18,0x01
    out 0x37,r18
    spm;                      Store program memory
    ret
;unsigned int read_program_memory (unsigned int adr ,unsigned char cmd);
_read_program_memory::
    ldi r20,0x11
    out 0x37,r20
    SPM
    .dw $ffff
    nop;                      重新使能 RWW 扇区
    mov r31,r17;              R31=ZH R30=ZL
    mov r30,r16;              move adress to z pointer
    sbrc r18,0;               read lockbits? (second argument=0x09)
    out 0x37,r18;             if so, place second argument in SPMEN register
    lpm;                      read LSB
    mov r16,r0
    inc R30
```

```
    lpm
    mov r17,r0;                read MSB (ignored when reading lockbits)
    ret
;void write_lock_bits (unsigned char val);
_write_lock_bits::
    in r20,0x37
    sbrc r20,0
    rjmp _write_lock_bits
    mov r0,r16
    ldi r17,0x09
    out 0x37,r17
    spm;                       write lockbits
    .dw $ffff
    nop
    ret
```

5.15　ATmega8 内置 RTC 的应用

```
/*********************************************
Project              :ATmega8 内置 RTC 的应用
Chip type            :ATmega8
Clock frequency      :外部(EXT) 8.000000 MHz
Author               :詹卫前
Rework               :沈文
From                 :广州双龙电子
Comments             :
1. 学习使用 ATmega8 内置的实时时钟的使用
2. 学习使用 ATmega8 的 UART 通过 RS232 与 PC 机进行通信
3. 本程序在 SL-MEGA8 上测试通过
4. 说明:
ATMEGA8 只有在使用内部 RC 振荡器，T2 使用异步时钟的情况下，外接的 32.768KHz 的晶振才起作用。
5. PC 机设置说明
(1) 使用 ICCAVR 的终端调试窗口(Terminal)，进行通信调试。
(2) 设置串口为 com1(或 com2)，通信波特率为 19200(Tools->Environment Options)。
(3) 将 PC 屏幕光标定位于调试窗口中。
*********************************************/
#include "iom8v.h"
#include "macros.h"
```

```
#define fosc 8000000                                    //晶振 8MHz
#define baud 19200                                      //波特率
unsigned char hour;
unsigned char minute;
unsigned char second;
void putchar(unsigned char c)                           //字符输出函数
    {
    while (!(UCSRA&(1<<UDRE)));
    UDR=c;
    }
unsigned char getchar(void)                             //字符输入函数
    {
    while(!(UCSRA& (1<<RXC)));
    return UDR;
    }
void puts(char *s)                                      //字符串输出函数
    {
    while (*s)
        {
        putchar(*s);
        s++;
        }
    putchar(0x0a);                                      //回车换行
    putchar(0x0d);
    }
void uart_init(void)                                    //UART 初始化
    {
    UCSRB=(1<<RXEN)|(1<<TXEN)|(1<<RXCIE);               //允许发送和接收
    UBRRL=(fosc/16/(baud+1))%256;
    UBRRH=(fosc/16/(baud+1))/256;
    UCSRC=(1<<URSEL)|(1<<UCSZ1)|(1<<UCSZ0);             //8 位数据+1 位 STOP 位
    }
void time(void)
    {
    if (second<60)
         return;
    else
         {
         second-=60;
         if (minute<59)
              minute++;
         else
```

```
            {
            minute=0;
            if (hour<23)
            hour++;
            else
            hour=0;
            }
        }
    }
void timer2_init(void)                          //初始化
    {
    TCCR2=0x00;
    ASSR=1<<AS2;                                //异步时钟
    TCNT2=0xE0;                                 //定时时间 1 秒
    TCCR2=(1<<CS22)|(1<<CS21)|(1<<CS20);        //分频比 1024,0x07
    }
#pragma interrupt_handler timer2_ovf_isr:5 //定时器 2 溢出中断
void timer2_ovf_isr(void)
    {
    TCNT2=0xE0;
    second++;
    }
void put_time(void)
    {
    putchar(hour/10+0x30);
    putchar(hour%10+0x30);
    putchar(':');
    putchar(minute/10+0x30);
    putchar(minute%10+0x30);
    putchar(':');
    putchar(second/10+0x30);
    putchar(second%10+0x30);
    }
#pragma interrupt_handler uart0_rx_isr:12  //UART 接收中断
void uart0_rx_isr(void)
    {
    unsigned i;
    i=UDR;
    if (i=='t')
        {
        hour=(getchar()-0x30)*10;
        hour=hour+(getchar()-0x30);
```

```
        minute=(getchar()-0x30)*10;
        minute=minute+(getchar()-0x30);
        second=(getchar()-0x30)*10;
        second=second+(getchar()-0x30);
        }
    }
void main(void)                                    //main 程序
    {
    unsigned char second_old;
    uart_init();
    timer2_init();
    TIMSK=1<<TOIE2;
    SEI();
    while(1)
        {
        if (second!=second_old)
            {
            time();
            second_old=second;
            put_time();
            }
        }
    }
```

第 6 章　GCCAVR 软件使用初步

GCCAVR 全称为 GUN C FOR AVR，是公开源代码的自由软件，因此使用它完全不必考虑价格因素。并且，由于有大量的高手参与开发，所以无论是 GCC 本身，还是与 GCC 配套的 AVR 编译包，其更新速度和效率都是其他开发工具所不能比拟的，这就极大地减少了因为开发工具本身的缺陷引起程序错误的概率，降低了维护成本。用 GCCAVR 编写源程序的可读性和可移植性都很高，只要是符合 GNU 规范的源程序，除了与硬件相关的代码外，其他的基本上不用改动，就可以实现在不同种类的 MCU 中移植，甚至可以顺利地移植到 DSP、ARM 等平台上。需要特别注意的是：GCCAVR(包括所有的 GNU 程序)来自 Unix 或者 Linux 平台，现在被移植到了 Windows 平台上，在移植后还保留了类似 Unix 的工作环境，如果读者不熟悉 Unix 环境可能会觉得软件很难用，熟练以后才会觉得功能的强大。

GCCAVR 支持 AT90 以及 ATmega 系列全部型号的 AVR 单片机，对于 ATtiny 系列的单片机，只直接支持 ATtiny22 和 ATtiny26 两种型号(实际上 GCCAVR 也提供了对 ATtiny 系列进行开发的手段，只是要注意技巧而已，到目前为止，支持没有 RAM 的 ATtiny 系列的只有 ICCTINY 编译器)。

大多数商业软件开发工具都有一个集成开发环境(Intefrated Development Environment，IDE)，如 ICCAVR、IAR 等编译器，这个集成开发环境提供一个图形用户接口(Graphical User Interface，GUI)，其中包括了程序编辑器和图形化的前端程序用来和编译器、汇编器、连接器、标准 C 库、建库工具接口，这些前端工具用对话框让用户能够设定各种选项，并将一组文件加入某个工程(Project)中，而且在前端隐藏和封装了真正的命令行编译器、汇编器、连接器和标准库。但是目前，GCC 还没有提供类似商业工具软件那样的 IDE 工作环境，因此学习在 GCCAVR 下开发软件，最好首先学习好如何编写 makefile 来使用 make 工具，然后学习控制 GCC 的通用命令行选项，再进一步学会控制 gas(汇编器)和 ld(连接器)。读者在使用 GCCAVR 以前还应尽可能多的阅读帮助文档，尤其要仔细阅读其中的 FAQ。

本章简单地介绍 GCCAVR 的安装和使用，并将 GCCAVR 与 ICCAVR 作对比。

6.1 GCCAVR 安装

6.1.1 下载

GCCAVR 是 AVR 单片机专用的 C 编译软件，可以在 http://sourceforge.net/projects/winavr 和 http://www.avrfreaks.net 上免费下载，并且 avrfreaks 网站上还对 GCCAVR 的各个组件进行整合，制作了便于安装和使用的 Windows 版(32 位)GCCAVR，并且提供了完整的使用手册，还在论坛上提供了强有力的技术支持。

GCCAVR 现在最新的版本是 2003-1-15，在 avrfreaks 网站上 GCCAVR 的 Windows 版有 Winavr-20030115-bin-install.exe 和 Winavr-20030115-src-install.exe 两个安装文件，其中后面一个安装文件是编译器的源程序，如果读者准备加入编译器的开发，或者想看看这么强大的软件内部，可以下载这个文件并研究它，而前面一个安装文件是后面一个安装文件在 Win32 平台上编译出来的结果，可以直接安装使用，在这里我们只需下载和使用它。要注意：在 Unix/Linux 和 WINDOWS 环境中 GCCAVR 的安装程序不一样，下载时应注意。

6.1.2 安装

1．系统需求

GCCAVR 既可以在 Unix、Linux 系统上使用，也可以在 Win98/Win2000/WinXP 操作系统下使用，当选用 Win98/Win2000/WinXP 操作系统下使用的版本时，必须满足以下的最低要求：

(1) X386 以上计算机；

(2) 50M 以上硬盘空间；

(3) 64M 内存。

2．安装

下面再介绍最新的 20020115 版的的安装，先双击下载的 Winavr-20030115-bin-install 文件，提示是否需要安装，如图 6.1 所示。

图 6.1 GCCAVR 安装界面

单击“是”的按钮，出现如图 6.2 所示的安装界面。

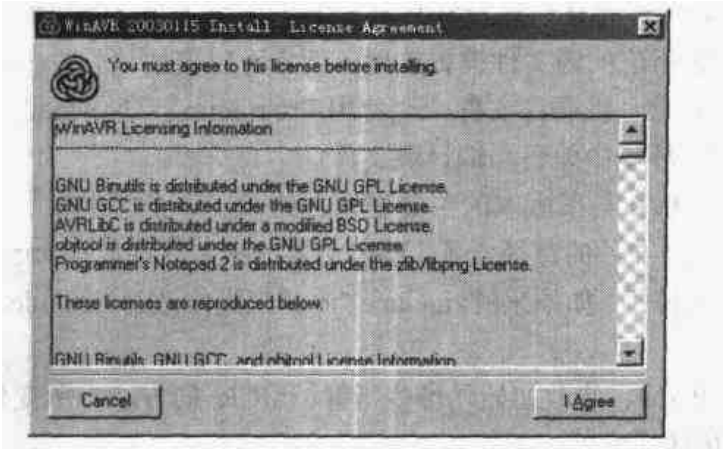


图 6.2　GCCAVR 安装界面

单击“I Agree”按钮，同意 WinAVR 的使用许可，出现如 6.3 所示路径选择界面，让用户选择安装路径。

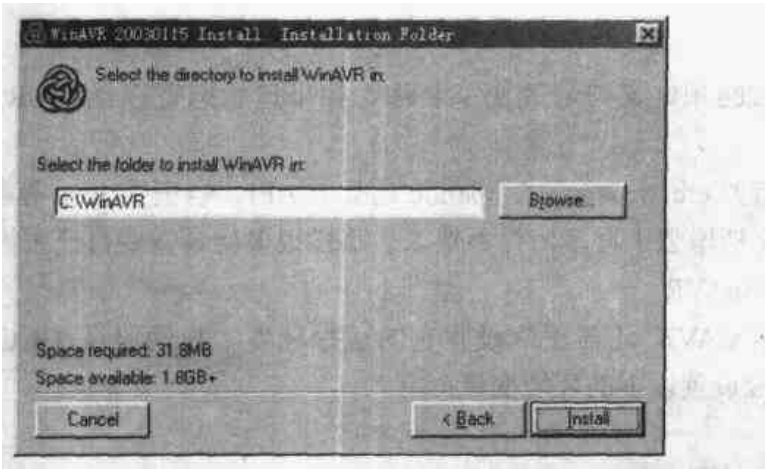


图 6.3　GCCAVR 安装路径设置界面

一般选择默认的 c:\WinAVR 路径，按下“Install”按钮，出现图 6.4 所示的安装过程。

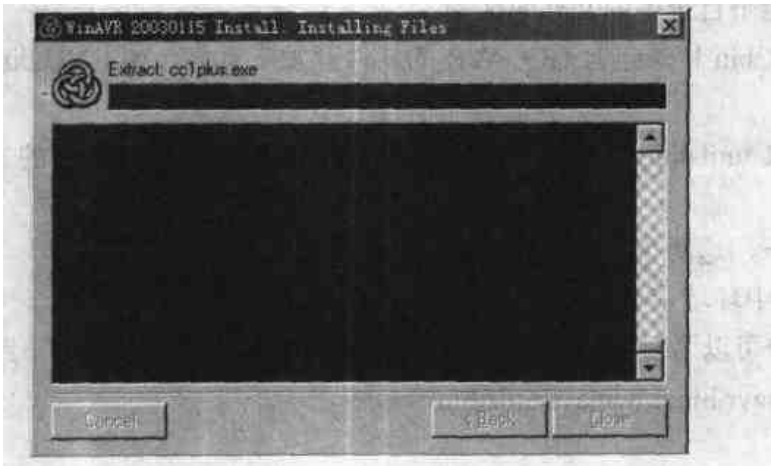


图 6.4　GCCAVR 安装过程

约等待一分钟的时间，就可以完成安装。在完成安装后，在桌面上会多一个“Programmers Notepad”图标并自动用记事本打开“Readme”文件，其中 Programmers Notepad 是 GCCAVR 所配的源文件编辑软件，读者可以使用该软件编辑源文件，但编者通常都是使用 UltraEdit 软件编辑源文件，比使用 Programmers Notepad 更方便，后面会详细介绍如何在 UltraEdit 环境中编辑和编译源文件。

3．新版本 GCCAVR 与旧版本的主要更新之处

(1) AVR 特定包含文件的路径变了，现在主要的头文件都放在 avr 目录下，因此有一些包含文件的名称也改掉了，如原来的#include "io.h"应当修改为#include "avr/io.h"，这一点要引起注意。

(2) 使用外部 RAM、修改初始化堆栈指针、MCR 的方法有所变化，还允许用户在程序运行之前插入初始化代码。

(3) 新版本建议使用直接赋值的方法操作特殊功能寄存器(SFR)。如：

```
DDRA=0x00;
temp=PINA;
```

(4) 有少数的预定义符号改变了名称，如串口状态寄存器 USR 中的 OVR 变成了 DOR 等。

(5) 增加诸如 crc16、sprintf、malloc 之类的 API，修正了 acos 等函数的缺点。

(6) 帮助文档增强了许多，再不像以前那样很多问题需要自己去搜索和琢磨了。

4．设置 WinAVR

要使用 GCCAVR 还需正确设置它的运行环境。我们假设 GCCAVR 的安装目录是 C:\WinAVR，应设置以下的环境变量：

```
AVR=C:\WinAVR\bin
AVR=C:\WinAVR\utils\bin
```

要注意这两个目录中的文件的区别：

C:\WinAVR\bin 目录包含 GCCAVR 的软件开发工具集，如 GNU binutils，GNU GCC，objtool 等。

C:\WinAVR \utils\bin 包含从 Unix 或 GNU 程序移植到 Windows 的文件，包括 sh(bash)和 maket 等。

一般情况下，这两个目录都是必需的。

(1) Win98 中环境变量的设置：

在 win98 中可以用 UltraEdit 打开 c:\autoexec.bat，并在其中加入以下路径，如图 6.5 所示：

path=c:\winavr\bin;c:\winavr\utils\bin

图 6.5　AUTOEXEC 中添加路径

要注意：如果在 autoexec.bat 中原先有一些 path 设置，应该将以上的路径添加到 path 路径的最前端，以防止其他的 make 文件插在前面，导致不能正确启动 GCCAVR。

(2) Win2000 和 WinXP 中环境变量的设置

图 6.6　Win2000 和 WinXP 中环境变量的设置

对于 Win2000 和 WinXP 中需在控制面板中修改环境变量，打开控制面版->系统->高级->环境变量->系统变量，编辑 Path 的值并添加以下路径，如图 6.6 所示。

c:\winavr\bin;c:\winavr\utils\bin;

要注意，应以系统管理员或自己的用户名登录进 Win2000 或 WinXP，并且应将以上路径添加到 Path 的最前面。

5．编译和链接 demo 程序

GCCAVR 目前还只是从类似 DOS 下的命令行方式，打开 MS-DOS 窗口(Win2000/WinXP 下叫“命令提示符”)，进入 C:\WinAVR\doc\examples\demo 目录，在命令行上输入 make，然后按回车键，是否出现如图 6.7 的提示？

图 6.7　Make 提示信息

如果没有出现该提示，一般是系统没有重新启动，可重新启动系统，也可以直接输入以下路径：

path=c:\winavr\bin;c:\winavr\utils\bin

再次输入 make 命令就可以出现图 6.7 中所示的提示信息。如果还没有看到以上信息，那就是环境变量设置错误所致，请重新设置环境变量。

退回 Windows 环境中(可直接利用键盘上的 Windows 快捷键退回，即标准 103 键盘中 Ctrl 和 Alt 之间的那个键)，使用 UltraEdit 打开 C:\WinAVR\doc\examples\demo\demo.c 文件，在源文件中添加一个空格然后删除空格并保存(只是为了改变一下该文件的生成时间)。重新回到 MS-DOS 窗口，再次执行 make，如果能够看到图 6.8 所示的提示信息，则说明您已经成功地编译了第一个 GCCAVR 程序。

图 6.8 编译成功后的提示信息

这个 demo 编译后的最终结果是 demo.hex/demo.srec 和 demo_eeprom.hex/demo_eeprom.srec，这是 Intel 十六进制/Motorola S 格式的烧片子文件，分别对应 flash 中的程序和 eeprom 中的数据，不过演示文件的 eeprom 是空的。

将 C:\WinAVR\sample 下的 makefile 复制到刚才的 demo 目录下，然后用 UltraEdit 打开 makefile，并修改如下两行：

```
MCU=atmega128        修改为=>      MCU=at90s2313
TARGET=main          修改为=>      TARGET=demo
```

再对 demo.c 作一下修改并存盘(再次改变文件的生成时间)，然后再回到 MS-DOS 窗口执行 make 命令，执行后应出现如图 6.9 所示的信息。

细心的读者看到图 6.9 的编译结果，会有些奇怪，为什么 text 为 0 而 data 为 244，这是由于 makefile 中有以下两处错误，这两处错误影响显示结果，但不影响编辑结果，用 UltraEdit 打开该 makefile 文件并修改这两处错误：

```
# Display size of file.
.PHONY : sizebefore
sizebefore:
    @echo Size before:
    -$(HEXSIZE)          =>本行更改为-$( ELFSIZE)
.PHONY : sizeafter
sizeafter:
```

```
@echo Size after:
$(HEXSIZE)          =>本行更改为-$( ELFSIZE)
```

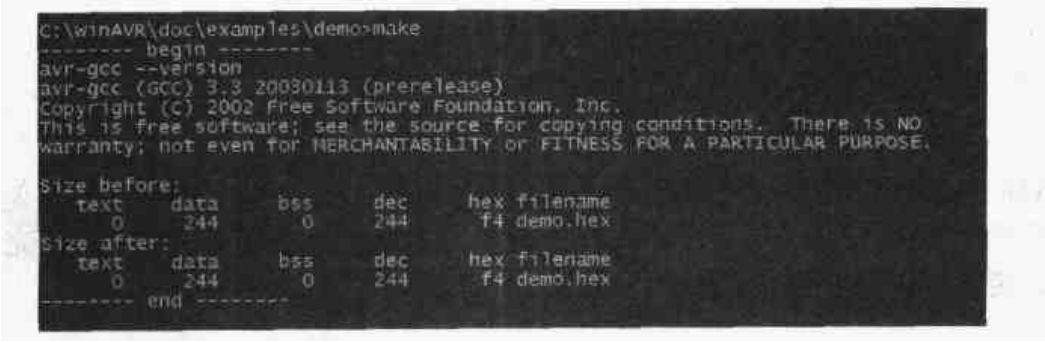


图 6.9　再次编译成功后的提示信息

修改完成以后，再回到 MS-DOS 窗口执行 make 命令，执行后出现如图 6.10 所示的编译信息。

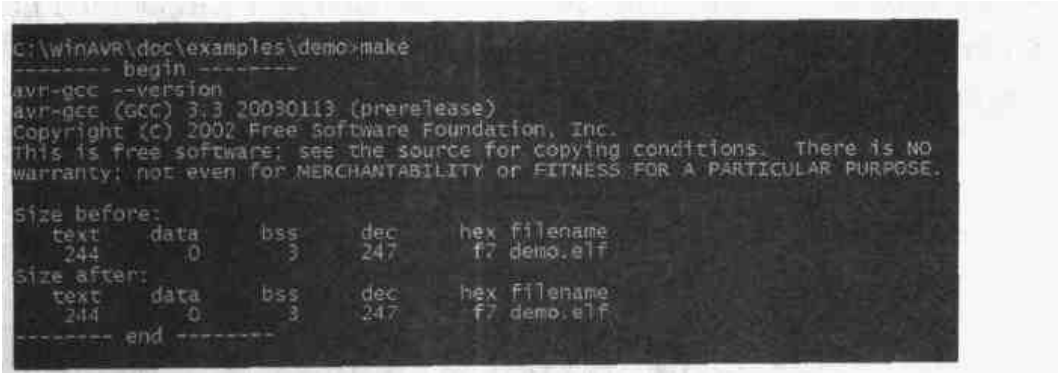


图 6.10　改错以后编译成功的提示信息

这次我们使用新的 makefile 模板文件编译后，又多生成以下 3 个有用的文件：

demo.cof：可以在 AVR studio 中进行源码级调试的目标文件，本文件是使用 objtool 生成的，应使用 3.55 版以上的 AVR studio 进行仿真和调试。

demo.lss：列表文件，列举了和启动文件一起编译生成的最终汇编代码，是整个工程绝对定位后的完整列表文件，整个工程编译后只生成一个与工程同名的.lst 文件，是新版本新增的输出文件。

demo.map：给出各存储区变量分配情况的文本文件。

注意，要将.lss 文件与.lst 文件区别开来，其中生成的.lst 文件为：

demo.lst：列表文件，列举出了源文件中的全部语句对应的汇编代码，变量和代码没完成绝对定位。每一个源文件均生成一个与该源文件同名的.lis 文件。

6.2 使用 GCCAVR 工具

我们将从创建一个简单的 AVR 应用实例开始介绍 GCCAVR，并分别说明如何使用命令行格式对源文件进行编译、链接和调试，最后介绍利用 makefile 文件来一次完成全部的编译和链接工作。

6.2.1 建立一个项目

本节我们将创建一个简单的 AVR 应用，利用 PWM(脉冲调制)控制一个 LED，这个 LED 将实现 1 秒点亮 1 秒熄灭的交替闪动，使用 AT90S8515 芯片，4MHz 晶振，用 10 位 PWM 输出控制 LED，如图 6.11 所示。

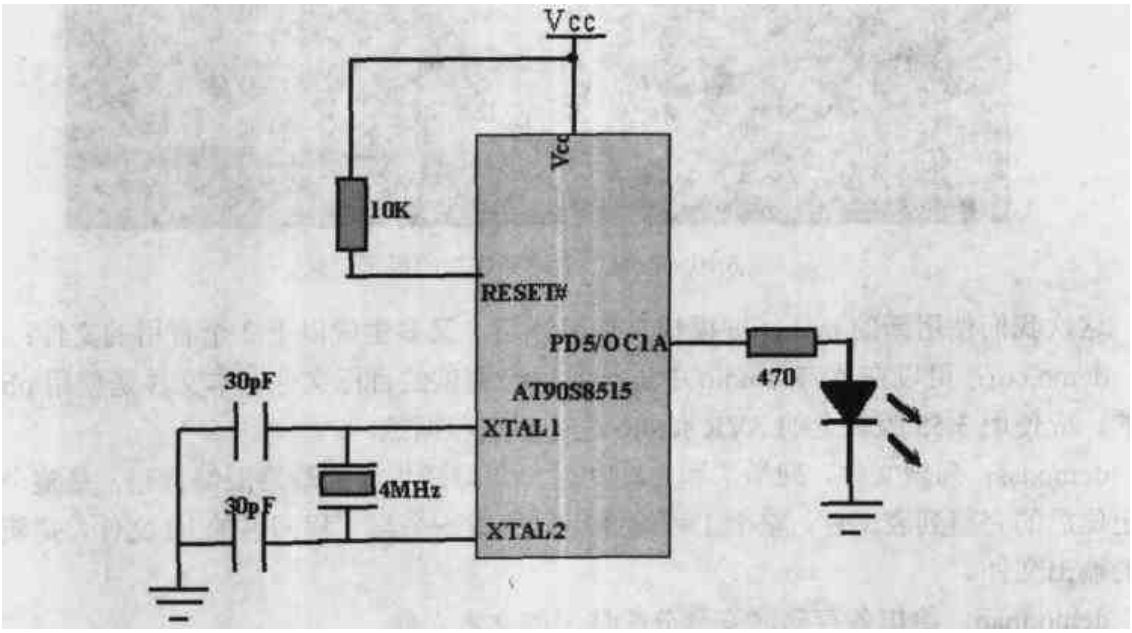


图 6.11 PWM 控制的 LED

下面我们先给出 GCCAVR 的源程序：

```
#include "avr/io.h"
#include "avr/interrupt.h"
#include "avr/signal.h"
#define OC1 PD5
#define DDROC DDRD
#define OCR OCR1AL
```

```
/* 使用 10 位 PWM，需要一个 16 位变量记录其当前值 */
volatile unsigned int pwm;
volatile enum
    {
    UP,DOWN
    } direction;
/*
**SIGNAL()是用于中断服务程序的程序入口，如果 TIMER 溢出将触发 SIGNAL()
**所定义的中断服务函数。
*/
SIGNAL(SIG_OVERFLOW1)
    {
    switch(direction)
        {
        case UP:
        if(++pwm==1023)
        direction=DOWN;
        break;
        case DOWN:
        if(--pwm==0)
        direction=UP;
        break;
        }
/**在中断服务程序中使用 outw()函数是安全的，因为中断服务函数执行时，CPU 已经关闭全局中
断，否则必须使用 outw_atomic()函数。**/
    outw(pwm,OCR);
    }
/*系统复位后需要完成初始化 PWM 和使能中断的工作*/
void ioinit(void)
    {
    outp(BV(PWM10)|BV(PWM11)|BV(COM1A1),TCCR1A);    /*使用 10 PWM*/
    outp(BV(CS10),TCCR1B);                          /* 使用 fck 作为定时器时钟 */
    outw(OCR,0);                                    /* PD5(OC1A)为输出 */
    outp(BV(OC1),DDROC);                            /* 允许 TIMER1 溢出中断 */
    timer_enable_int(BV(TOIE1));                    /* 打开全局中断允许 */
    sei();
    }
void main(void)
    {
    ioinit();
    for(;;)      /* Main 函数中的 loop 循环不作任何事情，只是等待中断的发生 */
        ;
```

```
    }
```

作为对比，我们也给出 ICCAVR 的源程序，读者可以比较两者的区别：

```
#include "io8515v.n"
#include "macros.h"
#define OC1 PD5
#define DDROC DDRD
/* 定义 OCR 为 16 位寄存器 OCR1A */
#define OCR OCR1A
volatile unsigned int pwm;
volatile enum
    {
    UP,DOWN
    } direction;
/* 中断函数声明，中断向量为 7 */
#pragma interrupt_handler timer1_ovf_isr:7
void timer1_ovf_isr(void)
    {
    switch(direction)
        {
        case UP:
            if(++pwm==1023)
                direction=DOWN;
            break;
        case DOWN:
            if(--pwm==0)
                direction=UP;
            break;
        }
    pwm = OCR;
    }
void ioinit(void)
    {
    TCCR1A=(1<<PWM10)|(1<<PWM11)|(1<<COM1A1);    /* 使用 10 PWM */
    TCCR1B=(1<<CS10);                            /* 使用 fck 作为定时器时钟 */
    OCR=0;                                       /* 初始化 PWM 寄存器为“0” */
    DDROC=(1<<OC1);                              /* PD5(OC1A)为输出 */
    TIMSK=(1<<TOIE1);                            /* 允许 TIMER1 溢出中断 */
    SEI();                                       /* 打开全局中断允许 */
    }
void main(void)
```

```
{
ioinit();
for(;;)
    ;
}
```

6.2.2　编译和链接

我们先在 C:\WinAVR\doc\examples 文件夹中生成一个 CH6_1 的文件夹，并将上面 GCCAVR 的源程序命名为 CH6_1.c，并保存到新生成的 CH6_1 文件夹中。要注意：扩展名必须为小写的.c，而不能为大写的.C，因为使用.C 的扩展名时，编译器将认为这是 C++编写的源程序，将调用 avr-c++对源文件进行编译而导致出错。

要对 CH6_1.c 进行编译，首先要告诉编译器我们所使用 MCU 的型号，可以使用-mmcu 参数定义 MCU 的型号，如果不指定 MCU 的型号，则编译器默认为 AT90S8515。

-mmcu：定义 MCU 的型号

在编译行中常用以下参数：

-Os：让编译器对代码在可能的范围内进行优化，其中参数-Ox(x=1,2,3,s)就是控制编译器的优化方式和等级，1~3 代表优化方式，s 代表按代码长度优化，在这里我们使用-Os 优化方式。

-g：参数产生调试信息；

-c：参数告诉编译器编译完成后停止，不进行链接操作。在 c 源程序规模不大时，我们可以一次性完成编译和链接工作，当 C 源程序规模较大，实际应用中一般将其分解成几个源文件，先逐个编译后，再一起链接。

在命令行中输入以下指令：

```
avr-gcc  -g  -Os  -mmcu=at90s8515  -c  CH6_1.c
```

如果 c 源程序没有错误，编译器将生成 CH6_1.o 文件，如图 6.12 所示。

图 6.12　生成的 CH6_1.o 文件

我们再把上面生成的 CH6_1.o 链接为一个二进制文件 CH6_1.out，在命令提示行输出以下命令：

```
avr-gcc  -g  -mmcu= at90s8515  -o CH6_1.out  CH6_1.o
```

如果没有输入错误，则可以生成 CH6_1.out 文件，如图 6.13 所示。

图 6.13　生成的 CH6_1.out 文件

编译完成后，我们可以从屏幕上得到许多有用的信息和结果，但是并没有产生可以烧写芯片的数据文件。GCCAVR 还提供一个有用的工具 avr-objdump，使用它能从.out 文件中提取出许多有用的信息。avr-objdump 工具有许多参数项，输入单个的命令可以查看其详细信息，使用-h 参数可以查看其代码信息。

我们输入以下的命令：

```
avr-objdump -h CH6_1.o
```

将产生程序各部分占用空间的详细信息，如图 6.14 所示。

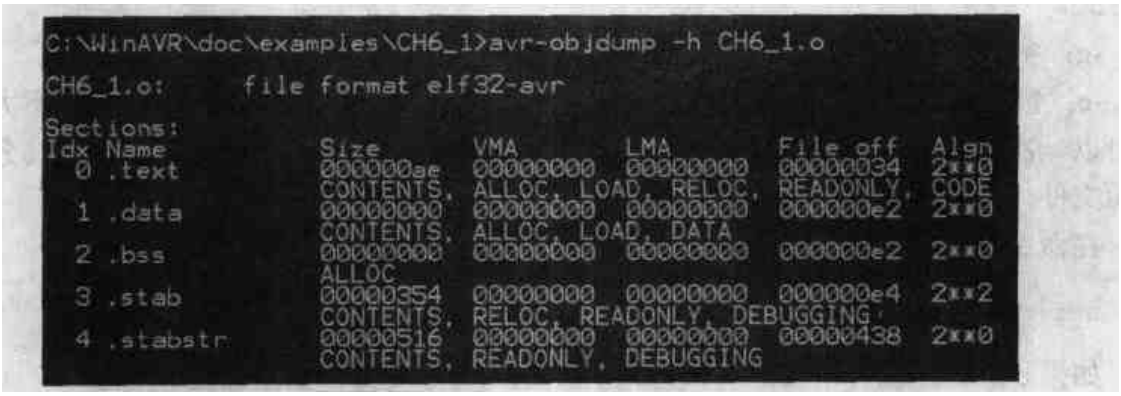


图 6.14　程序各部分占用空间的详细信息

编者经常使用-S 参数，它将反汇编.o 文件，指令如下：

```
avr-objdump -S CH6_1.o
```

6.2.3　使用“MAP”文件

尽管 avr-objdump 工具很有用，但有时我们需要查看链接产生的信息。“MAP”文件可以显示数据和程序空间的大小，还能显示模块的使用以及加载的库函数信息。在输入链接命令时，我们可以加上-Wl, -Map 参数来产生 MAP 文件(注意：在 W 后面是小写字母 l，

不要写成数字 1)。输入以下命令重新链接 CH6_1.o 文件：

```
avr-gcc  -g  -mmcu=at90s8515  -Wl, -Map, CH6_1.map -o CH6_1.out  CH6_1.o
```

就可以生成 CH6_1.map 文件，如图 6.15 所示。

图 6.15　生成的 CH6_1.map 文件

6.2.4　产生 .hex 文件

到现在为止，我们仍然没有生成可以烧写芯片的输出文件，可以使用 GCCAVR 提供的另一个工具 avr-objcopy 来生成我们所需的文件，使用该命令加上-j 参数和-O ihex 参数，就可以从.out 文件生成.hex 文件，指令如下：

```
avr-objcopy -j  .text  -O  ihex  CH6_1.out  CH6_1.hex
```

可以得到 CH6_1.hex 的烧写文件，如图 6.16 所示。

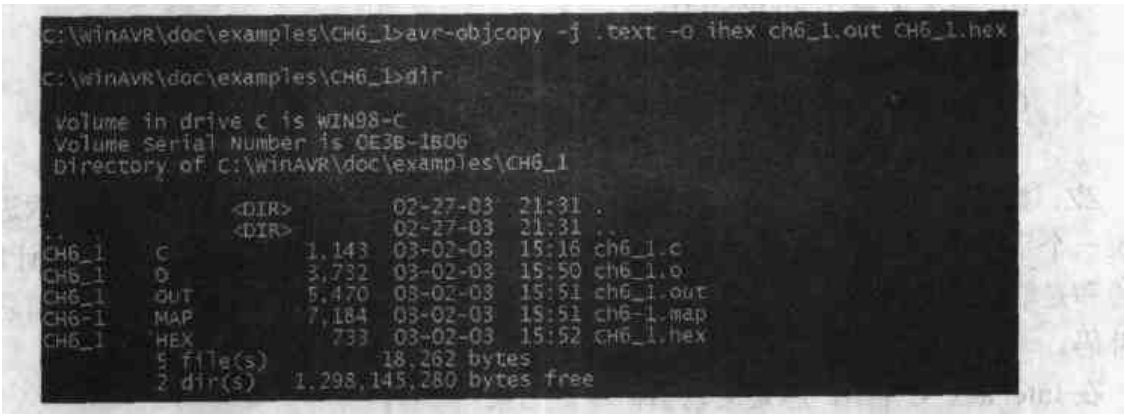


图 6.16　生成的 CH6_1.hex 文件

其中 CH_1.hex 文件的内容如图 6.17 所示，编者直接使用 DOS 中的 type 命令来显示 CH6_1.hex 文件中的内容，实际上该文件可以用任何源程序编辑软件或者文字编辑软件(如 Word 等)打开该文件。

图 6.17 生成的 CH6_1.hex 文件的内容

Intel hex 文件常用来保存单片机或其他处理器的目标程序代码，在该文件中保存的是物理程序存储区中的目标代码映象，大部分的编程器都支持这种格式。Intel hex 文件全部由可打印的 ASCII 字符组成，一个.hex 文件由一条或多条记录组成，每条记录都由一个冒号“:”打头，其格式如下：

```
:CCAAAARR......ZZ
```

其中：

CC：本条记录中的数据字节数

AAAA：本条记录中的数据在存储区中的起始地址

RR：记录类型：

00 数据记录 (data record)

01 结束记录 (end record)

02 段记录 (paragraph record)

03 转移地址记录 (transfer address record)

.....：数据域，即具体的数据信息

ZZ：数据域校验和 Intel hex 文件记录中的数字都是十六进制格式，两个十六进制数字代表一个字节。CC 是数据域中的实际字节数，地址、记录类型和校验和域没有计算在内，校验和是取记录中从数据字节计数域(CC)到数据域(.....)最后一个字节的所有字节总和的 2 的补码。

在 Intel hex 文件中，总是用以下命令行结束：

```
:00000001FF
```

在上面，我们使用-j 参数说明我们需要.text 空间的内容，也可以使用该参数来产生 EEPROM 空间的内容，命令如下：

```
avr-objcopy -j  .eeprom  -O  ihex  CH6_1.out  CH6_1.eep
```

上面指令产生的.eep 文件内容如图 6.18 所示。

图 6.18　生成的 CH6_1.eep 文件

生成的 CH6_1.eep 文件的内容如图 6.19 所示。

图 6.19　生成的 CH6_1.eep 文件的内容

由于演示中并没有对 eeprom 初始化，也没有使用 eeprom 的内容，因此生成的 CH6_1.eep 是一个空的烧写文件，只有.hex 格式的结束行。

6.2.5　使用 makefile 文件

1．makefile 模板介绍

可以看到，如果我们每次修改 c 源文件，都需要重新使用上面的编译、链接和产生.hex 文件命令，这些命令文件输入不方便且易错，为减少这些重复性工作，我们可以使用类似于“批处理”文件--makefile 文件。GCCAVR 也提供了 makefile 模板，只需将该模板作简单的修改就可以供我们在实际的工程中使用。将 C:\WinAVR\sample 下的 makefile 复制到我们的演示文件 CH6_1.c 所在的文件夹中(C:\WinAVR\doc\examples\CH6_1)，用 UltraEdit 打开该文件，并对以下两个部分进行修改并保存。

```
MCU=atmega128          修改为=>    MCU=at90s8515
TARGET=main            修改为=>    TARGET=CH6_1
```

在 MS-DOS 窗口中输入 make 命令，就可以直接生成我们所需的目标文件。

下面列举出 GCCAVR 提供 makefile 模板的内容，并对其中的命令作简要的说明：

```
# Sample makefile (c) 2002-2003 Eric B. Weddington
# Released to the Public Domain
# Define directories.    指定 WinAVR 的系统参数
   DIRAVR=c:/winavr
```

```
DIRAVRBIN=$(DIRAVR)/bin
DIRAVRUTILS=$(DIRAVR)/utils/bin
DIRINC=.
DIRLIB=$(DIRAVR)/avr/lib
# Define programs.  定义所使用的标识符
SHELL=sh
COMPILE=avr-gcc
ASSEMBLE=avr-gcc -x assembler-with-cpp
REMOVE=rm -f
COPY=cp
OBJCOPY=avr-objcopy
OBJDUMP=avr-objdump
ELFCOFF=objtool
HEXSIZE=@avr-size --target=$(FORMAT) $(TARGET).hex
ELFSIZE=@avr-size $(TARGET).elf
FINISH=@echo "Errors: none"
BEGIN=@echo "-------- begin --------"
END=@echo "--------  end  --------"
# MCU name  指明系统使用的AVR型号，如AT90s8515、AT90s8535、ATmega8、ATmega128等
MCU=atmega128
# Output format. Can be [srec|ihex].  指明输出文件的格式，共有srec(Motorola S格式)和ihex(Intel 16进制格式)两种输出格式可选择，一般选用ihex格式
FORMAT=ihex
# Target file name (without extension).  指明目标使用的文件名，注意文件名不带扩展名.c
TARGET=main
# List C source files here.  指明用到的所有c源程序文件
SRC=$(TARGET).c
# List Assembler source files here.  指明用到的所有汇编源程序文件
ASRC=
# Compiler flags.  编译参数
CPFLAGS=-g -Os -funsigned-char -funsigned-bitfields -fpack-struct
-fshort-enums -Wall -Wstrict-prototypes -Wa,-ahlms=$(<:.c=.lst)
# Assembler flags.  汇编参数
ASFLAGS=-Wa,-ahlms=$(<:.s=.lst), -gstabs
# Linker flags (passed via GCC).  链接参数
LDFLAGS=-Wl,-Map=$(TARGET).map,--cref
# Additional library flags (-lm=math library).  链接过程需要的其他库文件
LIBFLAGS=-lm
# Define all project specific object files.  定义OBJ文件
OBJ=$(SRC:.c=.o) $(ASRC:.s=.o)
# Define all listing files.  定义LST文件
```

```
    LST=$(ASRC:.s=.lst) $(SRC:.c=.lst)
# Add target processor to flags.    指明芯片型号
    CPFLAGS+=-mmcu=$(MCU)          编译过程所需的芯片型号
    ASFLAGS+=-mmcu=$(MCU)          汇编过程所需的芯片型号
    LDFLAGS+=-mmcu=$(MCU)          链接过程所需的芯片型号
# Default target.    make 的默认输出目标文件
all: begin gccversion sizebefore $(TARGET).elf $(TARGET).hex $(TARGET).eep
$(TARGET).lss $(TARGET).cof sizeafter finished end     说明应在此处增加
$(TARGET).map，生成.map 文件
# Default target.
.PHONY : all
all: begin gccversion sizebefore $(TARGET).elf $(TARGET).lss $(TARGET).map
$(TARGET).hex $(TARGET).cof $(TARGET).eep sizeafter finished end
# Eye candy. 显示信息
.PHONY : begin
begin:
$(BEGIN)
.PHONY : finish
finished:
    $(FINISH)
.PHONY : end
end:
$(END)
# Display size of file.
.PHONY : sizebefore
sizebefore:
@echo "Size before:"
-$(HEXSIZE)              =>注意：本行应更改为-$( ELFSIZE)
.PHONY : sizeafter
sizeafter:
@echo "Size after:"
$(HEXSIZE)               =>注意：本行应更改为-$( ELFSIZE)
# Display compiler version information.   显示编译器版本信息
.PHONY : gccversion
gccversion :
$(COMPILE) --version
# Create AVROBJ format file from ELF output file. (Future release)   从 elf
文件生成 avrobj 文件
obj: .elf
$(OBJCOPY) -O avrobj -R .eeprom $< $@
# Create COFF format file from ELF output file.   调用 ELFCOFF，从 elf 文件生
成 COFF 文件
```

```
    cof: .elf
    $(ELFCOFF) loadelf $< mapfile $*.map writecof $@
    # Create final output files (.hex, .eep) from ELF output file.  从 elf 文
件生成输出文件
    hex: .elf
    $(OBJCOPY) -O $(FORMAT) -R .eeprom $< $@
    eep: .elf
    -$(OBJCOPY) -j .eeprom --set-section-flags=.eeprom="alloc,load"
--change-section-lma .eeprom=0 -O $(FORMAT) $< $@
    # Create extended listing file from ELF output file.  从 elf 文件生成 lss 文件
    lss: .elf
    $(OBJDUMP) -h -S $< >$@
    # Link: create ELF output file from object files.  从 obj 文件生成 elf 文件
    SECONDARY : $(TRG).elf
    elf: $(OBJ)
    $(COMPILE) $(LDFLAGS) $(OBJ) $(LIBFLAGS) --output $@
    # Compile: create object files from C source files.  从.c 源文件生成 obj 文件
    o : .c
    $(COMPILE) -c $(CPFLAGS) -I$(DIRINC) $< -o $@
    # Assemble: create object files from assembler files.  从.s 汇编文件生成 obj 文件
    o : .s
    $(ASSEMBLE) -c $(ASFLAGS) -I$(DIRINC) $< -o $@
    # Target: clean project.  删除临时文件
    .PHONY : clean
    clean: begin clean_list finished end
    .PHONY : clean_list
    clean_list :
    $(REMOVE) $(TARGET).hex
    $(REMOVE) $(TARGET).eep
    $(REMOVE) $(TARGET).obj
    $(REMOVE) $(TARGET).cof
    $(REMOVE) $(TARGET).elf
    $(REMOVE) $(TARGET).map
    $(REMOVE) $(TARGET).obj
    $(REMOVE) $(TARGET).cof
    $(REMOVE) $(TARGET).a90
    $(REMOVE) $(TARGET).sym
    $(REMOVE) $(TARGET).lnk
    $(REMOVE) $(OBJ)
    $(REMOVE) $(LST)
    $(REMOVE) $(SRC:.c=.s)
    # List source file dependencies here:
```

```
    $(TARGET).o: $(TARGET).c
    # Automatically generate C source code dependencies. (Code taken fromthe GNU
make user manual.)
    # Note that this will work with sh (bash) and sed that is shipped withWinAVR
(see the SHELL variable defined above).
    # This may not work with other shells or other seds.
    #.d: .c
    # set -e; $(COMPILE) -MM $(CPFLAGS) $< \| sed 's/\($*\)\.o[ :]*/\1.o $@ :
/g' > $@; \[ -s $@ ] || rm -f $@
    # include $(SRC:.c=.d)
    # List assembly only source file dependencies here:
```

注意：

(1) 以$开始的命令语句前必须加 TAB 键产生的缩进，而不能用空格缩进。

(2) 在 GCCAVR 提供的 makefile 模板中有两处错误，请将其更正，如下：

```
    # Display size of file.
    .PHONY : sizebefore
    sizebefore:
       @echo Size before:
       -$(HEXSIZE)          =>注意：本行应更改为-$( ELFSIZE)

    .PHONY : sizeafter
    sizeafter:
       @echo Size after:
       $(HEXSIZE)           =>注意：本行应更改为-$( ELFSIZE)
```

(3) 在# Default target.语句中增加$(TARGET).map 说明，以生成.map 文件。

原模板为：

```
    # Default target.
    .PHONY : all
    all: begin gccversion sizebefore $(TARGET).elf $(TARGET).hex $(TARGET).eep
$(TARGET).lss $(TARGET).cof sizeafter finished end
```

改进后为：

```
    # Default target.
    .PHONY : all
    all: begin gccversion sizebefore $(TARGET).elf $(TARGET).lss $(TARGET).map
$(TARGET).hex $(TARGET).cof $(TARGET).eep sizeafter finished end
```

(4) 还可以在 makefile 中增加几个 target，如：

```
dump:
<tab> avr-objdump -S -h $(TARGRT).elf >dump.txt
<tab> notepad dump.txt
install :
```

<tab> 根据下载器指定 UISP 参数，可以用并口 STK 200 dongle、串口 AVRISP 等，甚至可以是 Altera ByteBlaste 下载电缆。

这样，编译完以后，还会执行以下操作：

make install 直接下载代码到 AVR 芯片中。

make dump 反汇编原代码到 dumop.txt，并自动用记事本打开。

(5) 在 makefile 文件的 CPFLAGS=行中，加入-Ox(x=1,2,3,s)就可以对编译生成的结果进行优化，其中 1~3 代表优化方式，s 代表按代码长度优化。但在调试时，如果使用最优化选项，则只要语义不变，编译器会自由重组语句的执行顺序，以便在一个条件运算中只使用一条分支语句。而分支语句只能指向一个很小的范围，因此如果一次条件运算语句不能直接使用一条分支语句，则编译器会在其周围使用一条跳跃指令及一条 rjmp 指令，这会给调试时带来一些麻烦。最优化选项还会带来另一个副作用，一个变量只在它实际被使用的代码中有效。因此如果在函数开始的时候，一个变量被放入寄存器，那么当编译器注意到该变量已经不再使用的时候，这个寄存器就给其他变量使用。此时，如果在调试时试图监视该变量的值，就有可能造成误解。因此为了避免这些副作用，建议在调试时关闭最优化选项，在调试完成后再用最优化选项生成所需的烧写文件。

这个问题并非是 GCCAVR 独有，几乎所有的编译器在选用最优化选项时均有这个问题，而且 GCCAVR 的最优方选项生成的代码比其他 C 编译器更稳定一些。

(6) 如果需要编译带浮点函数的库，应在 LDFLAGS=项目最后加上-lm 参数。

2．在 UltraEdit 环境中编辑源程序

我们可以利用 UltraEdit 软件自带的自动执行和输出窗口来编译源文件，比前面介绍的在 MS-DOS 窗口中会方便得多，因为我们的源程序也是在这个软件中编辑的。

对 UltraEdit 设置如下：

在高级->工具配置中输入如图 6.20 所示的内容，输入完成后，点击右边的“插入”按钮，并按“确定”按钮退出配置画面。

在设置完成后，在高级菜单上就会多出一个 WinAVR 选项，如图 6.21 所示。

当编辑完源程序后，只要单击菜单中的 WinAVR 一栏(或者使用 UltraEdit 提供的快捷键 ctrl+shift+1)，UltraEdit 软件就会自动调用 make 软件，并且按照 makefile 中的内容对源程序进行编译，将结果显示在 UltraEdit 的输出栏中。

我们用 UltraEdit 软件打开 C:\WinAVR\doc\examples\demo\demo.c 源程序，按下 ctrl+shift+1 快捷键，就可以在 UltraEdit 软件的输出窗口看到编译的结果，如图 6.22 所示。

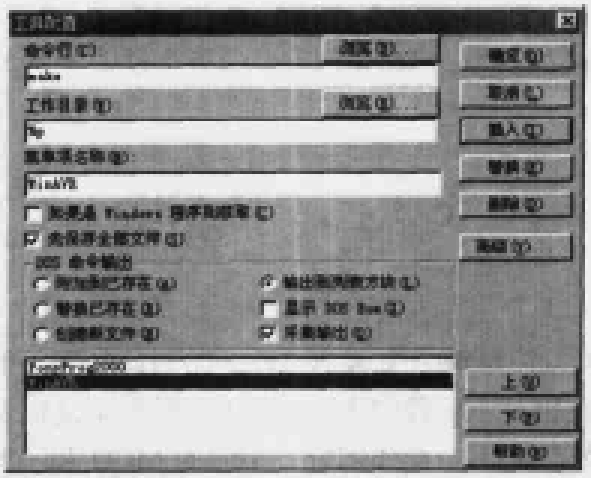


图 6.20　UltraEdit 中工具配置内容

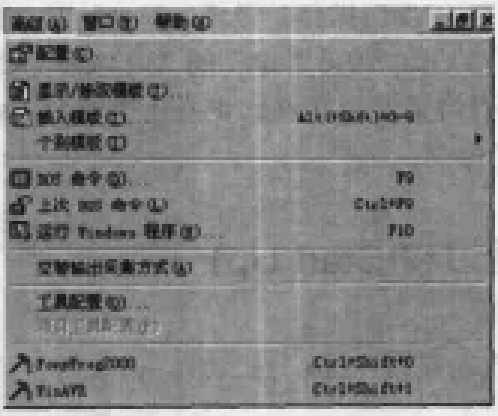


图 6.21　UltraEdit 的高级菜单中多出 WinAVR 选项

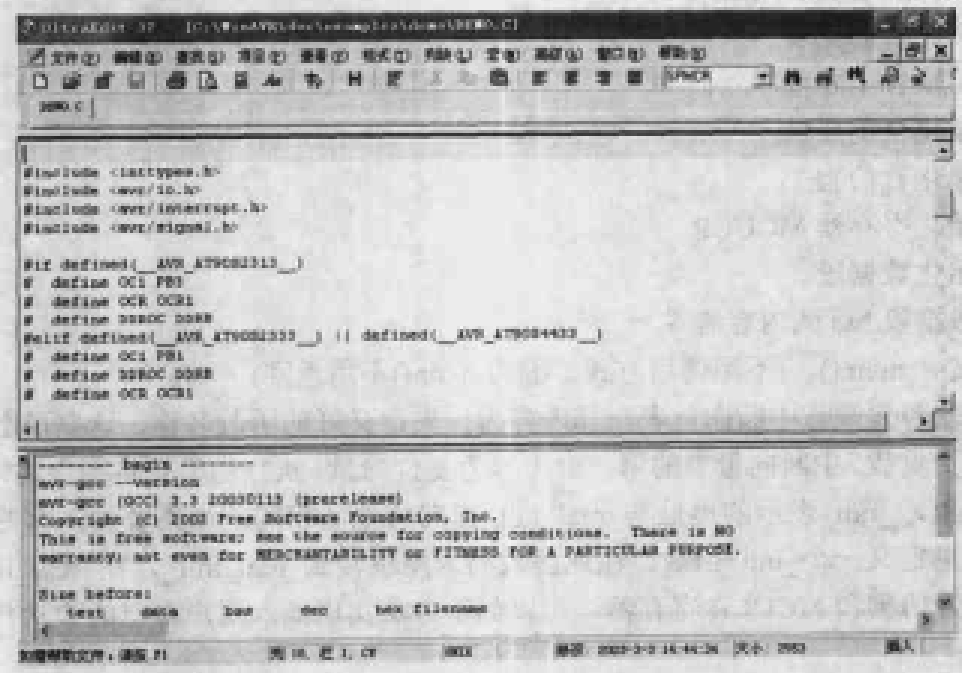


图 6.22　在 UltraEdit 环境中编译结果

编者通常将 PonyProg2000 也放置在 UltraEdit 的“高级”菜单中，这样在编译完成后就可以直接下载至目标电路板进行仿真和调试，设置方法与增加 WinAVR 选项相同。

6.3 应用 API

嵌入式编程的代码可以简单地分为两部分，一是与硬件无关的算法部分，对其编程与普通 C 编程没有区别，这也是不同种类 MCU 及不同编译器之间移植而基本不用修改的部分；二是与硬件相关的寄存器/端口操作部分，不同的 MCU 和不同的编译器之间实现方法各有不同，因此在移植时这一部分内容必须做相应的修改。在 GCCAVR 里则通过一系列的 API 来解决与硬件相关的寄存器/端口操作，当然，您也可以定义自己的 API。

我们在此简单地介绍目前 GCCAVR 里定义的 API 以及 GCCAVR 工作过程，在介绍与硬件操作相关的 API 时，我们将采用和 ICCAVR 对比说明的方式，并且为了在形式上区别，有关 ICCAVR 的介绍都是斜体字。

6.3.1 应用程序启动过程(Start Up)

标准库文件包含一个启动模块(Start Up Module)，用于为真正执行用户程序做环境设置。

启动模块完成的任务如下：

1. 提供缺省向量表
2. 提供缺省中断程序入口
3. 初始化全局变量
4. 初始化看门狗
5. 初始化寄存器 MCUCR
6. 初始化数据段
7. 将数据段.bss 的内容清零
8. 跳转到 main()。(不用调用方式，因为 main()不用返回)

启动模块包含缺省中断向量表，其内容为预先定义好的函数名称，这些函数名称可以由程序设计员重载。中断向量表的第一个内容为复位向量，执行结果是将程序跳转到_init_。在启动模块里，_init_表示的地址与_real_init_指向的地址相同，如果要加入客户代码，则需要在程序里定义一个_init_函数。在此函数的末尾跳转到_real_init_，在_real_init_部分，系统将设置看门狗和 MCUCR 寄存器。可以如下方法实现插入用户自己的启动代码：

```
void _real_init_(void);
void _init_(void)  __attribute__((naked));
```

```
void _init_(void)
    {
    // 用户代码
    // 最后的代码必须为：
    asm ("rjmp _real_init_");
    }
```

启动模块并没有真正取用相应寄存器的设置数值(以符号_init_wdctr_，_init_mcucr_，_init_emcucr_表示)，而是通过地址来取得其值，因用户可以通过链接器的--defsym 选项来设置这些符号的地址。如果用户没有定义，则启动模块将使用默认值。

接下来系统将从程序存储器里把具有初值的全局变量加载到数据存储器 SRAM。然后是将数据段.bbs 清零，此数据段包含所有没有初值的非 AUTO 变量(如字符串，静态变量等)，最后，系统跳转到 main()函数，用户代码开始执行。系统对此特殊函数加入一些特殊的处理，进入此函数后，堆栈指向 SRAM 的末尾。

在 GCCAVR 的新版本中，对于程序的前期初始化显得比以前的版本要有弹性，允许用户在初始化代码运行之前插入任何代码，写一小段如下的汇编代码就能实现这个目标：

```
;; begin xram.S
#include "avr/io.h"
          .section .init1,"ax",@progbits
          ldi r16,_BV(SRE) | _BV(SRW)
          out _SFR_IO_ADDR(MCUCR),r16
;; end xram.S
```

汇编以上代码段，然后把得到的 xram.o 连接到您的程序中。这样，这段代码将被附着在初始化代码中，并且在 reset 完成后开始执行。

这样做的好处是，可以在初始化代码的最开头(这是真正的最开头，没有堆栈、没有寄存器归零等操作)插入想要的任何代码。比如用户需要判断单片机是上电复位还是由其他原因引起复位(如看门狗等)，并且针对不同的复位情况采取不同的对策，这样就可以充分利用新版 GCCAVR 提供的修改启动代码的功能。除非遇到一些非常特殊的情况，基本上这种操作并不需要对连接器脚本进行任何修改(其实旧版本的 GCC 也可以做到这些，但是需要手工调整连接器脚本)。最好不要修改__statck 的缺省初始值(位于内部 SRAM 的末尾且向下生长，速度较快)，还应添加 Wl,-Tdata,0x801100 以保证数据段在堆栈上方。

6.3.2　存储器 API

AVR 具有三种存储器：SRAM、Flash 和 EEPROM。GCCAVR 将程序代码放在 FLASH 空间，数据放在 SRAM 空间。

6.3.2.1 SRAM 数据空间

定位在 SRAM 数据空间的数据有多种类型，在使用这些类型说明前还必须加入头文件 inttypes.h，但如果已经包含了 avr/io.h 文件，那么就会自动加入了 inttypes.h 文件。GCCAVR 提供的变量类型如表 6.1 所示。

表 6.1 提供的数据类型

类 型	占用字节数	值 范 围
int8_t	1	-128 ~ 127
uint8_t	1	0 ~ 255
int16_t	2	-32768 ~ 32768
uint16_t	2	0 ~ 65535
int32_t	4	-2147483648 ~ 2147483647
uint32_t	4	0 ~ 4294967295
int64_t	8	-9.22 * 10^18 ~ 9.22 * 10^18
uint64_t	8	0 ~ 1.844 * 10^19

在 GCCAVR 中，SRAM 数据空间变量定义和使用比较简单，不需要使用 API 库函数。表 6.2 将 GCCAVR 和 ICCAVR 定义和操作 SRAM 空间的数据方式作个对比：

表 6.2 GCCAVR 和 ICCAVR 提供的数据类型对比

操作定义	GCCAVR	ICCAVR
定义字节变量	uint8_t val1,val2;	unsigned char val1,val2;
变量赋值	val2 = 0x09;	val2 = 0x09;
变量读取	val1 = val2;	val1 = val2;

6.3.2.2 程序存储器

如果要将数据(如常量、字符串等)放在 Flash 里，用户需要用 type__attribute__((progmem)) 指明数据类型，为了方便使用，GCCAVR 定义了一些更直观的符号，如表 6.3 所示。使用程序空间 API 时需要包括 avr/pgmspace.h 文件。

表 6.3 程序存储器数据类型定义

类 型	定 义
prog_void	void __attribute__((progmem))
prog_char	char __attribute__((progmem))
prog_int	int __attribute__((progmem))
prog_long	long __attribute__((progmem))

续表

类　型	定　义
prog_long_long	long long __attribute__((progmem))
PGM_P	prog_char const*
PGM_VOID_P	prog_void const*

表 6.4 为 GCCAVR 和 ICCAVR 定义/操作程序存储器变量对比。

表 6.4　GCCAVR 和 ICCAVR 的操作对比

操作定义	GCCAVR	ICCAVR
8 位字节数据	prog_char LINE = {0};	*char　LINE = 0;*
读取字节数据	char res = PRG_RDB(&LINE);	*char res = LINE;*
字节数组	prog_char TEN[10] = {0,1,2,3,4,5,6,7,8,9};	*char TEN[] = {0,1,2,3,4,5,6,7,8,9};*
读取字节数组	char res = PRG_RDB(&value[5]);	*char res = TEN[5];*
字符串	char *LINE1 = PSTR("The first line");	*char *LINE1 = "The first line";*
读取字符串	char rec = PRG_RDB(LINE1+3);	*char rec = *(LINE1+3);*

GCCAVR 提供以下操作程序存储器空间数据的库函数：

1．__elpm_inline

用法：uint8_t __elpm_inline(uint32_t addr);

说明：执行 ELPM 指令从 FLASH 中取数，参数为 32 位地址，返回一个 8 位数据。

2．__lpm_inline

用法：uint8_t __elpm_inline(uint16_t addr);

说明：执行 LPM 指令从 FLASH 中取数，参数为 16 位地址，返回一个 8 位数据。

3．memcpy_P

用法：void* memcpy_P(void* dst, PGM_VOID_P src, size_t n);

说明：memcpy 的特殊版本，完成从 FLASH 取 n 个字节的任务。

4．PRG_RDB

用法：uint8_t PGR_RDB(uint16_t addr);

说明：此函数简单地调用__lpm_inline

5．PSTR

用法：PSTR(s);

说明：参数为字符串，功能是将其放在 FLASH 里并返回地址。

6．strcmp_P

用法：int strcmp(char const*, PGM_P);

说明：功能与 strcmp()类似，第二个参数指向程序存储器内的字符串。

7．strcpy_P

用法：char* strcpy_P(char*, PGM_P);

说明：功能与 strcpy()类似，第二个参数指向程序存储器内的字符串。

8．strlen_P

用法：size_t strlen_P(PGM_P);

说明：功能与 strlen()类似，第二个参数指向程序存储器内的字符串。

9．strncmp_P

用法：size_t strncmp_P(char const*, PGM_P, size_t);

说明：功能与 strncmp()类似，第二个参数指向程序存储器内的字符串。

10．strncpy_P

用法：size_t strncpy_P(char*, PGM_P, size_t);

说明：功能与 strncpy()类似，第二个参数指向程序存储器内的字符串。

6.3.2.3 EEPROM 数据空间

AVR 内部有 EEPROM，但其地址空间与 SRAM 不相同，在访问时必须通过三个与 EEPROM 有关的寄存器来进行。GCCAVR 的 EEPROM API 封装了这些功能，为用户提供了高级接口，使用时要包含 avr/eeprom.h 头文件。

GCCAVR 在程序中定义 EEPROM 数据的例子如下：

```
    static uint16_t uiData __attribute__((section(".eeprom")))=0;      /* 定义
无符号整型数 */
    static uint16_t table[6] __attribute__((section(".eeprom")));      /* 定义
数组 */
```

在 ICCAVR 中定义 EEPROM 空间数据，需要#pragma data:eeprom 伪指令指明数据存放位置：

```
#pragma data:eeprom                    //指明从 EEPROM 空间开始存放
unsigned int uiData=0x1234;            //定义整型数据
char table[6];无反顾                   //声明字节型数组
#pragma data:data                      //指明重新回到数据空间
```

表6.5为GCCAVR与ICCAVR操作EEPROM对比：

表 6.5 GCCAVR 与 ICCAVR 操作 EEPROM 举例

操作定义	GCCAVR	ICCAVR
读取变量 uiData	res = eeprom((unit16_t)&uiData);	*res = EEPROMread (uiData);*
写入数据	Eprom_wb((uint8_t)&table[0],0xAA);	*EEPROMwrite(table[0],0xAA);*

续表

操作定义	GCCAVR	ICCAVR
Copy 数组到 sram	uint8_t ramval[6]; eeprom_read_block(&ramval, (uint8_t)&eepromval, 5);	*unsigned char ramval[6];* *EEPROM_READ(*table,ramval);*

不同的 AVR 器件具有不同数目的 EEPROM，在 ICCAVR 中，针对不同 EEPROM 大小的芯片必须使用不同的函数访问，而 GCCAVR 中就没有这个限制，并提供 5 个与该 EEPROM 有关的 API 库函数：

1．eeprom_is_ready(void);

说明：此函数用于指示是否可以访问 EEPROM，如果 EEPROM 正在执行写操作，则在 4ms 内无法访问。函数通过查询相应的状态位来指示现在是否可以访问 EEPROM。

2．eeprom_rb

用法：uint8_t eeprom_rb(uint16_t addr);

说明：从 EEPROM 里读出一个字节的内容，参数 addr 用于指示要读出的地址，_EEGET(addr)调用此函数。

3．eeprom_read_block

用法：void eeprom_read_block(void* buf, uint16_t addr, size_t n);

说明：读出一块 EEROM 的内容，参数 addr 为起始地址，n 表明要读取的字节数，数据被读到 SRAM 的 buf 里。

4．eeprom_rw

用法：unint16_t eeprom_rw(uint16_t addr);

说明：从 EEPROM 里读出一个 16 位的数据，低字节为低 8 位，高字节为高 8 位，参数 addr 为地址。

5．eeprom_wb

用法：void eeprom_wb(uint16_t addr, uint8_t val);

说明：将 8 位数据 val 写入地址为 addr 的 EEPROM 存储器里。_EEPUT(addr，val)调用此函数。

6.3.3　中断 API

由于 C 语言设计目标与硬件无关，因此各种编译器在处理中断时使用的方法都是编译器设计者自己定义的方法。一般编译器都提供已经预先定义好的向量表，并指向具有预定义名称的中断例程。通过使用合适的名称和编译器指定的定义方法，实际中断时就能够调用用户定义相应的中断函数，如果用户没有定义自己的中断函数，则器件库的缺省函数被加入到生成的代码中。

除了中断向量表的问题，编译器还必须保护和恢复中断中使用过的寄存器。GCCAVR

中的中断 API 解决了中断处理的细节问题，只要将中断例程声明为 INTERRUPT()类型或 SIGAL()类型即可，在 GCCAVR 中，对于用户没有定义的中断，默认的处理是直接填入 reti 指令(中断返回)，使用中断 API 函数需要加入 avr/interrupt.h 文件和 avr/signal.h 文件。

表 6.6 AT90S8515 中断向量表

向量名称	向量	中断源	中断定义
--	1	RESET	复位中断
SIG_INTERUPT0	2	INT0	外部中断 0
SIG_INTERUPT1	3	INT1	外部中断 1
SIG_INPUT_CAPTURE1	4	TIMER1 CAPT	定时/计数器 1 捕获事件
SIG_OUTPUT_COMPARE1A	5	TIMER1 COMPA	定时/计数器 1 比较匹配 A
SIG_OUTPUT_COMPARE1B	6	TIMER1 COMPB	定时/计数器 1 比较匹配 B
SIG_OVERFLOW1	7	TIMER1 OVF	定时/计数器 1 溢出
SIG_OVERFLOW0	8	TIMER0 OVF	定时/计数器 0 溢出
SIG_SPI	9	SPI,STC	SPI 传输完成
SIG_UART_RECV	10	UART,RX	UART 正确收到一个字节
SIG_UART_DATA	11	UART,UDRE	UART 发送寄存器空
SIG_UART_TRANS	12	UART,TX	UART 成功发送一个字节
SIG_COMPARATOR	13	ANA_COMP	模拟比较器事件

在源程序中声明一个函数为中断服务函数有以下两种方式：INTERRUPT(SIGNAME)和 SIGNAL(SIGNAME)，其中前者在执行中断服务程序时，全局中断使能位有效，即还可以响应其他优先级高的中断请求；而后者执行中断服务程序时将关闭全局中断使能位，不能响应其他中断。以外部中断 INT1 为例，表 6.7 给出 GCCAVR 和 ICCAVR 中断声明的对比。

表 6.7 GCCAVR 和 ICCAVR 中断声明对比

GCCAVR	INTERRUPT(SIG_INTERRUPT1) { /* 中断服务程序 */ }	中断服务程序声明，全局中断允许位开，可以其他的中断请求
	SIGNAL(SIG_INTERRUPT1) { /* 中断服务程序 */ }	中断服务程序声明，全局中断允许位关，不能响应任何中断
ICCAVR	*#pragma interrupt_handler int0_isr:2* *void int0_isr(void)* { *//INT0 中断服务程序* }	*中断服务程序声明，全局中断允许位关，也不能响应任何中断，如果需响应其他中断，必须由用户在中断函数中开全局中断允许位 I*

下面提供几个中断 API 库函数：

1．cli

用法：void cli(void);

说明：通过清零全局中断触发寄存器 I 来停止中断响应，其编译结果仅为一条汇编指令。

2．enable_external_int

用法：void enable_external_int(uint8_t ints);

说明：此函数访问 GIMSK 寄存器(对于 ATMega 器件则是 EIMSK 寄存器)，功能与宏 outp()一样。

3．INTERRUPT

用法：INTERRUPT(signame)

说明：定义中断源 signame 对应的中断例程，在执行时，全局屏蔽位将清零，其他中断被使能。ADC 结束中断例程的例子如下所示：

```
INTERRUPT(SIG_ADC)
  {
  }
```

4．sei

用法：void sei(void);

说明：通过置位全局中断触发寄存器 I 来使能中断响应，其编译结果仅为一条汇编指令。

5．SIGNAL

用法：SIGNAL(signame)

说明：定义中断源 signame 对应的中断例程。在执行时，全局屏蔽位保持置位，其他中断被禁止。ADC 结束中断例程的例子如下所示：

```
SIGNAL(SIG_ADC)
  {
  }
```

6．timer_enable_int

用法：void timer_enable_int(uint8_t ints);

说明：此函数操作 TIMSK 寄存器，也可以通过 outp()来设置。

6.3.4　I/O 端口 API

本节介绍操作 I/O 寄存器的 API 函数和宏定义，它们大多是由在线汇编写的，因此不必担心其代码效率问题，而放心使用这些函数和宏。

1. _BV

用法：_BV(pos)；

说明：_BV()就是一次普通的左移运算，将位定义转换成屏蔽码(MASK)，与头文件 avr/io.h 里的位定义一起使用。

使用这个宏操作能够让代码更加易懂，而且由于编译器的作用，使用_BV()宏并不会导致任何性能上的降低或升高。

例如，设置定时器 2 为完全的 I/O 时钟、比较匹配时 OC2 输出、并且比较匹配时清空定时器，以及把 OC2 设定为输出端口，那么，既可以用：

```
CS2x=0b001;
COM2x=0b01;
CTC2=1;
DDRD=0x80;
```

也可以用：

```
TCCR2=_BV(COM20)|_BV(CTC2)|_BV(CS20);
DDRD=_BV(PD7);
```

像上例中，使用_BV()宏有利于提高源程序的可读性。

2. bit_is_clear

用法：uint8_t bit_is_clear(uint8_t port, uint8_t bit);

描述：如果 port 的 bit 位清零则返回 1。此函数调用 sbic 指令，故 port 应为有效地址。

例如，测试 PORTB 的第 3 位是否为“0”：

```
unit8_t result=bit_is_set(PORTB,PINB3);
```

如果 PORTB 的第 3 位为“0”，result=1；否则 result=0。

3. bit_is_set

用法：uint8_t bit_is_set(uint8_t port, uint8_t bit);

描述：如果 port 的 bit 位置位则返回 1。此函数调用 sbis 指令，故 port 应为有效地址。

4. cbi

用法：void cbi(uint8_t port, uint8_t bit);

说明：清零 port 的 bit 位。bit 的值为 0~7。如果 port 为实际 I/O 寄存器，则此函数生成一条 cbi 指令；否则，函数生成相应的优化代码。

例如，清零 PORTB 的第 3 位：

```
cbi(PORTB,PINB3)
```

5. inp

用法：uint8_t inp(uint8_t port);

说明：从端口 port 读入 8 比特的数值。如果 port 为常数，则函数生成一条 in 指令；若为变量，则函数用直接寻址指令。

6. __inw

用法：uint16_t __inw(uint8_t port);

说明：从 I/O 寄存器读入 16 位的数值。此函数用于读取 16 位寄存器(ADC，ICR1，OCR1，TCNT1)的值，因为读取这些寄存器需要合适的步骤。由于此函数只产生两条汇编指令，因此要在中断禁止时使用，否则有可能由于中断插入到指令之间造成读取错误。

7. __inw_atomic

用法：uint16_t __inw_atomic(uint8_t port);

说明：读取 16 位 I/O 寄存器的数值。此函数首先禁止中断，读取数据之后再恢复中断状态，因此可以安全地应用在各种系统状态。

8. loop_until_bit_is_clear

用法：oidoid loop_until_bit_is_clear (uint8_t port, uint8_t bit);

说明：此函数简单地调用 sbic 指令来测试端口 port 的 bit 位是否清零，port 必须为有效端口。

9. loop_until_bit_is_set

用法：oidoid loop_until_bit_is_set (uint8_t port, uint8_t bit);

说明：此函数简单地调用 sbis 指令来测试端口 port 的 bit 位是否置位，port 必须为有效端口。

10. outp

用法：void outp(uint8_t val, uint8_t port);

说明：将 val 写入端口 port，如果 port 为常数，则函数生成一条 out 指令；若为变量，则函数用直接寻址指令。

11. __outw

用法：void __outw(uint16_t val, uint8_t port);

说明：将 16 位的 val 写入端口 port。此函数适合于操作 16 位寄存器，如 ADC，ICR1，OCR1，TCNT1。由于此函数只产生两条汇编指令，因此要在中断禁止时使用，否则有可能由于中断插入到指令之间，改写了 Temp 寄存器的内容而导致写入错误。

12. __outw_atomic

用法：void __outw_atomic(uint16_t val, uint8_t port);

说明：将 16 位的 val 写入端口 port，此函数适合于操作 16 位寄存器，如 ADC、ICR1、OCR1、TCNT1。此函数首先禁止中断，读取数据之后再恢复中断状态，因此可以安全地应用在各种系统状态。

13. sbi

用法：void sbi(uint8_t port, uint8_t bit);

说明：置位 port 的 bit 位，bit 的值为 0~7，如果 port 为实际 I/O 寄存器，则此函数生成一条 sbi 指令；否则，函数生成相应的优化代码。

在新版的 GCCAVR 中，建议使用直接赋值的方法操作特殊功能寄存器(SFR)。如：

```
DDRA=0x00;
temp=PINA;
```

这种操作方法比较符合一般人的使用习惯，而且与 ICCAVR 的操作方法一致。

6.3.5 看门狗 WDT API

以下函数操作看门狗。宏定义参见 avr/wdt.h。

用户可以通过启动代码初始化看门狗。WDTCR 的缺省值为 0，如果希望将其设置为其他值，则需要在链接命令里加入相应的命令，使用的符号为__init_wdtcr__。如下为将 WDTCR 设置为 0x1f 的例子(表 6.8 为看门狗溢出时间选择)：

avr-ld -defsym __init_wdtcr__=0x1f

1．wdt_disable

用法：void wdt_disable(void);

说明：关闭看门狗。

2．wdt_enable

用法：void wdt_enable(unit8_t timeout);

说明：使能看门狗。看门狗溢出时间为 timeout。

3．wdt_reset

用法：void wdt_reset(void);

说明：产生喂狗指令 wdr。

表 6.8 看门狗溢出时间选择

Timeout	周　期
0	16K CLK
1	32K CLK
2	64K CLK
3	128K CLK
4	256K CLK
5	512K CLK
6	1024K CLK
7	2048K CLK

表 6.9 为 GCCAVR 与 ICCAVR 操作看门狗对比。

表 6.9　GCCAVR 与 ICCAVR 操作看门狗对比

操作定义	GCCAVR	ICCAVR
初始化并打开看门狗	wdt_enable(7);	*WDTCR = 0x0F; //WDT 使能，使 //用 2048K 分频*
关闭看门狗	wdt_disable();	*WDTCR = 0x18;　//写逻辑“1”到 //WDTOE 和 WDE WDTCR = 0x00;　//关闭 WDT*
喂狗	wdt_reset();	*WDR();　　//喂狗指令*

6.4　GCCAVR 使用在线汇编

本章介绍在 GCCAVR 中使用在线汇编，关于 ICCAVR 的在线汇编，读者可以参考“2.12 在线汇编”一节中的相关内容。

6.4.1　GCCAVR 的 ASM 声明

首先看用 GCCAVR 编写的一个从 PORTD 读入数据的例子：

```
asm("in %0,%1":"=r"(value):"I"(PORTD):)
```

由上可以看出 GCCAVR 的在线汇编由以下 4 个部分组成：

1．汇编指令本身，以字符串"in %0, %1"表示；
2．由逗号分隔的输出操作数，本例为"=r"(value)
3．由逗号分隔的输入操作数，本例为"I"(PORTD)
4．Clobber 寄存器

因此在 GCCAVR 中的在线汇编的通用格式为：

```
asm(代码 : 输出操作数列表 : 输入操作数列表 : clobber 列表);
```

例如：

```
asm(code:output operand list:input operand list:clobber list);
```

例子中%0 表示第一个操作数，%1 表示第二个操作数，即：

```
%0 -> "=r"(value)
%1 -> "I"(PORTD)
```

如果在后续的 C 代码中没有使用到汇编代码中使用的变量，则优化编译时会将这些语句删除，为了防止这种情况的发生，需要加入 volatile 属性：

```
asm volatile ("in %0,%1":"=4"(value):"I"(PORTD):);
```

嵌入汇编的 Clobber 寄存器部分可以忽略，而其他部分不能忽略，但可以为空，如下例：

```
asm volatile("cli"::)
```

6.4.2 汇编代码

用户可以在 GCCAVR C 代码里嵌入任意的汇编指令，就如同在汇编器里写程序一样。并且在一个 asm 指令中，您可以写任意多的汇编指令。为了增加程序的可读性，可以使用换行符“\n”：

```
asm volatile    ("nop\n\t"
                 "nop\n\t"
                 "nop\n\t"
                 "nop\n\t"
                 : :);              /* 注意：别忘了两个冒号 */
```

在 GCCAVR 中还提供了一些特殊的寄存器名称。

表 6.10 GCCAVR 提供几个特殊的寄存器名称

符　号	寄 存 器
__SREG__	状态寄存器 SREG(0x3F)
__SP_H__	堆栈指针高字节(0x3E)
__SP_L__	堆栈指针低字节(0x3D)
__tmp_reg__	r0，用于临时寄存器
__zero_reg__	r1，其值永远为 0
__PC__	程序指针

在应用程序里，可以任意使用寄存器 r0，而不用保存和恢复它。建议在程序里使用__tmp_reg__和__zero_reg__代替 r0 和 r1，因为我们不能确定在以后的版本里这两个寄存器的定义是否会改变。

6.4.3 输入/输出操作数

在线汇编中的每一个输入/输出操作数都需要一个限定符，GCCAVR 编译器可以支持表

6.11 所列的约束符。

表 6.11　支持的限定符

限　定　符	描　　述	范　　围
A	普通高部寄存器	r16～r23
B	指针	y,z
C		
D	高端寄存器	r16～r31
E	指针	x,y,z
G	浮点常数	0.0
I	6 位正整数	0～63
J	6 位负整数	-63～0
K	整形常数	2
L	整形常数	0
L	普通低部寄存器	r0～r15
M	8 位正整数	0～255
N	16 位正整数	
N	正整数	-1
O	正整数	8，16，24
P	正整数	1
Q	堆栈指针寄存器	sph:spl
R	任意一个寄存器	r0~r31
T	临时寄存器	r0
V	32 位正整数	
W		r24,r26,r28,r30
X	X 指针	x
Y	Y 指针	y
Z	Z 指针	z

要注意的是，在使用这些约束符号时要防止选择错误。例如，用户使用了“r”约束符号，而汇编语句则使用了“ori”指令，由于用户使用了“r”约束符，则编译器认为可以在 r0~r31 之间选择任意一个寄存器，若编译器选择了 r2~r15，则会由于选用的寄存器不适用 ori 而出现编译错误，因此必须使用正确的约束符“d”。

约束符号还可以有前置修饰符，如表 6.12 所示。

表 6.12　提供的 3 个前置修饰符

修　饰　符	指 定 意 义
=	只写操作数
+	读-写操作数(在线汇编不支持)
&	寄存器只能用作输出

要注意：输出操作数必须为只写操作数，C 表达式结果必须为 l(r0~r15)，编译器不检查汇编指令中的变量类型是否合适。输入操作数为只读，如果输入/输出使用同一个寄存器时，可以在输入操作数的约束字符里使用一个一位数字，这个数字告诉编译器使用与第 n 个(从 0 开始计数)操作数相同的寄存器。例如：

```
asm volatile("SWAP %0" : "=r"(value) : "0"(value));
```

这条语句的目的是交换变量 value 的高低 4 位。约束符号"0"告诉编译器使用与第一个操作数相同的寄存器作为输入寄存器。要注意的是，即使用户没有指定，编译器也有可能使用相同的寄存器作为输入/输出。在某些情况下会引发严重的问题，因此如果用户需要区分输入/输出寄存器，则必须为输出操作数增加修饰符"&"。如下例所示：

```
asm volatile("in %0, %1 \n\t"
             "out %1, %2 \n\t"
             : "=&r"(input)
             : "I"(port), "r"(output)
             );
```

此例的目的是读入端口数据，然后给端口写入另一个数据。若编译器使用了同一个寄存器作为参数 input 和 output 存储位置，则第一条指令执行后 output 的内容就被破坏了。而使用修饰符"&"之后，这个问题得以解决。

下面为一个高 16 位与低 16 位交换的 32 位数据操作的例子。

```
;16 位数据，高 8 位与低 8 位交换
asm volatile("mov  __tmp_reg__, %A0  \n\t"
             "mov  %A0, %B0 \n\t"
             "mov  %B0, __tmp_reg__ \n\t"
             : "=r"(value)
             : "0"(value)
             );
;32 位数据，高 16 位与低 16 位交换
asm volatile("mov  __tmp_reg__, %A0  \n\t"
             "mov  %A0, %D0 \n\t"
             "mov  %D0, __tmp_reg__ \n\t"
             "mov  __tmp_reg__, %B0 \n\t"
             "mov  %B0, %C0 \n\t"
             "mov  %C0, __tmp_reg__ \n\t"
             : "=r"(value)
             : "0"(value)
             );
```

其中“0”代表第一个操作数，“A”、“B”、“C”、“D”表示 32 位的数据：

```
位      31~24     23~16     15~8      7~0
          D         C         B         A
```

6.4.4　Clobber 寄存器

如前所述，asm 语句的最后一部分为 clobber，如果您在汇编代码里使用了没有作为操作数声明的寄存器，就需要在 clobber 里声明以通知编译器。例：

```
Asm volatile ("cli              \n\t"
              "ld r24, %A0      \n\t"
              "inc r24          \n\t"
              "st %A0, r24      \n\t"
              "sei              \n\t"
              :
              :"z"(ptr)
              :"r24"
              );
```

编译结果为：

```
CLI
LD R24, Z
INC R24
ST Z, R24
SEI
```

当然，也可以用__tmp_reg__来取代 r24，此时就没有 clobber 寄存器了。

```
Asm volatile ("cli                      \n\t"
              "ld __tmp_reg__, %A0      \n\t"
              "inc __tmp_reg__          \n\t"
              "st %A0, __tmp_reg__      \n\t"
              "sei                      \n\t"
              :
              :"z"(ptr)
              :"r24"
              );
```

下面我们再举一个更详细的例子：

```
c_func
   {
   uint8_t s;
   asm volatile("in %0, __SREG__        \n\t"
                "cli                    \n\t"
                "ld, __tmp_reg__, %A1   \n\t"
                "inc __tmp_reg__        \n\t"
                "st %a1, __tmp_reg__    \n\t"
                "out __SREG__, %0       \n\t"
                : "=&r"(t)
                :"z"(ptr)
                );
   }
```

在上例中看起来好像没问题，其实不尽然。由于优化的原因，编译器不会更新 C 代码里其他使用这个数值的寄存器，出于同样的原因，上述代码输入寄存器可能保持的不是当前最新的数值，对此，用户可以加入特殊的"memory" clobber 来强迫编译器及时更新所有的变量，如下所示：

```
c_func
   {
   uint8_t  s;
asm volatile("in %0, __SREG__        \n\t"
             "cli                    \n\t"
             "ld, __tmp_reg__, %A1   \n\t"
             "inc __tmp_reg__        \n\t"
             "st %a1, __tmp_reg__    \n\t"
             "out __SREG__, %0       \n\t"
             : "=&r"(t)
             :"z"(ptr)
             :"memory"
             );
}
```

更好的方法是将一个指针声明为 volatile，如下所示：

```
volatile uint8_t *ptr;
```

这样，一旦指针指向的变量发生变化，编译器就会重新加载最新的数值。下面再给出两个使用在线汇编 c 函数的例子，一个用于延时，另一个是读取 16 位寄存器的值。

```
/*  延时函数，在调用前必须初始化全局变量 delay_count  */
void delay(uint8_t ms)
    {
    uint16_t cnt;
    asm volatile(
                "\n"
                "L_dl1%=:" "\n\t"
                "mov %A0, %A2" "\n\t"
                "mov %B0, %B2" "\n"
                "L_dl2%=:" "\n\t"
                "sbiw %A0, 1" "\n\t"
                "brne L_dl2%=" "\n\t"
                "dec %2" "\n\t"
                "brne L_dl1%=" "\n\t"
                : "=&w" (cnt)
                : "r" (ms), "r" (delay_count)
                );
    }
/* 从 I/O 寄存器读取 16 位数据 */
uint16_t inw(uint8_t port)
    {
    uint16_t result;
    asm volatile(
                "in %A0,%1" "\n\t"
                "in %B0,(%1) + 1"
                : "=r" (result)
                : "I" (port)
                );
return result;
}
```

6.4.5　在线汇编中使用 #define 定义的常量

在线汇编中可以正常使用以下语句：

```
asm volatile("sbi 0x18,0x07;");
```

但将其改为以下语句：

```
asm volatile("sbi PORTB,0x07;");
```

编译器出现 Error: constant value required 错误，原因是 PORTB 是一个在头文件中定义的，在编译前由预处理器处理的符号，然而预处理器并不会对在线汇编的部分进行处理，

因此编译器无法识别 PORTB 这个符号，可以使用如下语句解决这个问题：

```
asm volatile("sbi %0, 0x07" : "I" (PORTB):);
```

6.4.6 混合编程的寄存器使用

如果用户需要进行汇编与 C 的混合编程，就必须了解寄存器的使用。

1．寄存器使用

R0：可用做暂时寄存器。如果用户汇编代码使用了 R0，且要调用 C 代码，则在调用之前必须保存 R0，C 中断例程不会自动保存和恢复 R0。

R1：C 编译器假定此寄存器内容为 0。如果用户使用了此寄存器，则在汇编代码返回之前需将其清零，C 中断例程会自动保存和恢复 R1。

R2~R17、R28、R29：C 编译器使用这些寄存器。如果用户汇编代码需要使用这些寄存器，则必须保存并恢复这些寄存器。

R18~R27、R30、R31：如果您的汇编代码不调用 C 代码则无需保存和恢复这些寄存器，如果用户要调用 C 代码，则在调用之前需保存。

2．函数调用规则

参数表：函数的参数由左至右分别分配给 R25 到 R8。每个参数占据数个寄存器，若参数太多以致 R25～R8 无法容纳，则多出来的参数将放入堆栈。

返回值：8 位返回值存放在 R24。16 位返回值存放在 R25:R24。32 位返回值存放在 R25:R24:R23:R22。64 位返回值存放在 R25:R24:R23:R22:R21:R20:R19:R18。

3．汇编代码中使用 C 名称

在默认情况下，在 C 和汇编中使用同一个变量名或函数名，如下面的例子，c 中定义的变量 val1 和 val2 都可以在 c 和汇编中使用。

```
c_func
    {
    uint8_t  val1;
    uint8_t  val2;
    val1 = 2;
    val2 = 5;
    asm volatile(
                "lds r30,val1    \n\t"
                "lds r31,val2    \n\t"
                "add r30,r31     \n\t"
                "sts val2,r30    \n\t"
                : :
                );
    }
```

不过，用户也可以为自己的 C 变量/函数重新定义一个名字(别名)在汇编中使用，例如：

```
unsigned long value asm("clock")=3686400;
```

这个语句告诉编译器，在汇编中使用别名 clock 而不是 value，可以使用 clock 对 C 中定义的全局变量 value 进行操作。要定义局部变量，如下例：

```
void Count(void)
     {
     register unsigned char counter asm("r3");
     /* ... 其他代码... */
     asm volatile("clr r3");
     /* ... 其他代码... */
     }
```

上面 C 中声明局部寄存器变量 counter，定义它的汇编别名为 r3。那么在线汇编“clr r3”实际将清零寄存器变量 counter。

当然，也可以为特定的汇编函数/变量声明一个 c 别名。如果使用汇编定义了一个“CALCULATE”函数，想在 C 中调用它，但是 C 编译器并不支持直接调用汇编，那么就可以使用下面的语句为函数 CALCULATE 起一个别名：

```
extern long Calc(void) asm ("CALCULATE");
```

在 C 中调用 Calc(void) 函数实际上就是调用汇编中的 CALCULATE 函数。

6.5　使用定时/计数器

定时/计数器的原理及寄存器在第 4 章的相关内容已经介绍，这里只作简单的说明，重点讲述如何用 GCCAVR 语言操作定时/计数器。

6.5.1　定时/计数器 0

Timer0 是一个从 0~0xff，8 位定时/计数器。相关的寄存器如表 6.13 所示。

表 6.13　与 TIMER0 有关的几个寄存器

类　别	名　称	名　称
定时器寄存器	定时/计数器控制寄存器	TCCR0 (Timer/Counter 0 Control Register)
	定时/计数器寄存器	TCNT0 (Timer/Counter 0 Value)

续表

类　别	名　称	名　称
中断寄存器	中断标志寄存器	TIFR (Timer Interrupt Flag Register)
	中断屏蔽寄存器	TIMSK (Timer Interrupt Mask Register)
	全局中断屏蔽寄存器	GIMSK (General Interrupt Mask Register)

6.5.1.1 定时器应用

定时/计数器 0 工作在定时器模式时，从系统时钟分频而得到定时时钟，每个时钟(分频后)周期 TCNT0 寄存器的值都要增加 1。分频系数可以选择 1、8、64、256 和 1024，详细内容参见第 4 章相关内容。

定时器的可以使用查询方式或中断方式，下面分别介绍。

1．查询方式

使用查询方式操作定时器的程序如下：

```
/*
** 使用查询方式使用 TIMER0 定时器
** 当 TIFR 寄存器溢出时，变量 led 的值增加 1，并且输出到 PORTB 驱动发光 led
*/
#include "avr/io.h"
uint8_t led;
uint8_t state;
void main( void )
    {
    outp(0xFF, DDRB);                    /* PORTB 所有端口设为输出 */
    outp(0, TCNT0);                      /* 定时/计数器 0 的初始值为 0 */
    outp(5, TCCR0);                      /* 使用 1024 分频( ck/1024) */
    led=0;
    for (;;)
         {
         /* 循环检测 TIFR 寄存器中溢出标志位是否有效 */
         do
            state=inp(TIFR) & 0x01;
         while (state != 0x01);
         outp(~led,PORTB);               /* 输出 led 的值到 PORTB */
         led++;
         if (led==255)
            led=0;
         /* 写逻辑"1"到 TIFR 的 TOV0 位来清楚溢出标志位 TOV0 */
         outp((1<<TOV0),TIFR);
         }
    }
```

本例中 TCCR0 初始化为 5(采用 1024 分频)，并且把 TCNT0 的起始值设为 0，每过 1024 个系统时钟，TCNT0 的值加 1。

for(;;)；语句定义了一个无限循环，它内部的 DO-WHILE 循环一直检测 TIFR 中的定时器溢出标志位 TOV0 是否有效(定时器是否溢出)。

当 TCNT0 的值达到 0xFF 后，加 1 操作将触发定时器溢出事件，TIFR 中的 TOV0 位会被硬件置位。

检测到溢出标志后，输出 led 的值驱动 PORTB 口的 LED。

写逻辑"1"到 TOV0 来清除 TOV0 位，重新启动定时器。

下面给出 ICCAVR 的查询方式操作程序：

```
/*** 使用查询方式使用 TIMER0 定时器。
** 当 TIFR 寄存器溢出时，变量 led 的值增加 1，并且输出到 PORTB 驱动发光 led。***/
#include "io8515v.h"
unsigned char led;
void  main(void)
    {
    DDRB=0Xff;                          /* PORTB 所有端口设为输出 */
    TCNT0=0;                            /* 定时/计数器 0 的初始值为 0 */
    TCCR0=5;                            /* 使用 1024 分频(ck/1024) */
    led=0;
    for (;;)
         {
         /* 循环检测 TIFR 寄存器中溢出标志位是否有效 */
         while(!(TIFR&0x01));
             PORTB=led;                 /* 输出 led 的值到 PORTB */
         led++;
         if (led==255)
             led=0;
         /* 写逻辑"1"到 TIFR 的 TOV0 位来清楚溢出标志位 TOV0 */
         TIFR|=(1<<TOV0);
         }
    }
```

2．中断方式

实际上，定时器的中断方式操作用的更加普遍。当定时/计数器工作在中断方式下时，一旦 TIMER0 溢出事件发生，CPU 程序指针会自动地跳到相应的中断地址，执行中断服务函数。当中断服务函数执行结束后，程序返回到发生中断的地方继续执行。仍然采用上面的例子，程序如下：

```
/**
```

```
* 使用中断方式使用 TIMER0 定时器。
** 当 TIFR 寄存器溢出时，变量 led 的值增加 1，并且输出到 PORTB 驱动发光 led
*/
#include "avr/io.h"
#include "avr/interrupt.h"
#include "avr/signal.h"
uint8_t led;
SIGNAL (SIG_OVERFLOW0)                    /* 定义定时器 0 溢出中断服务程序 */
  {
  outp(~led, PORTB);                      /* 输出 led 的值到 PORTB */
  led++;
  if (led==255)
  led=0;
  outp(0,TCNT0);                          /* 重新加载定时器寄存器 */
  }
void main(void)
     {
     outp(0xFF, DDRB);                    /* PORTB 所有端口设为输出 */
     outp((1<<TOIE0), TIMSK);             /* 允许 TIMER0 溢出中断 */
     outp(0, TCNT0);                      /* 定时/计数器 0 的初始值为 0 */
     outp(5, TCCR0);                      /* 使用 1024 分频(ck/1024) */
     led=0;
     sei();                               /* 全局中断允许 */
     for (;;)                             /* 循环等待中断处理 */
          ;
     }
```

中断服务程序由关键字"SIGNAL"声明，"SIG_OVERFLOW0"告诉编译器这里是 TIMER0 溢出中断向量。当溢出事件发生后，系统会自动执行中断服务程序。在初始化中，必须打开 TIMER0 溢出中断的使能位 TOIE0(TIMSK 寄存器)和全局中断使能位 I(SREG 寄存器)。

再给出 ICCAVR 中断操作程序：

```
/**
* 使用中断方式使用 TIMER0 定时器。
** 当 TIFR 寄存器溢出时，变量 led 的值增加 1，并且输出到 PORTB 驱动发光 led
*/
#include "io8515v.h"
unsigned char led;
#pragma interrupt_handler timer0_isr: 7
void timer0_isr(void)                     /* 定义定时器 0 溢出中断服务程序 */
```

```
    {
    PORTB=led;                          /* 输出 led 的值到 PORTB */
    led++;
    if (led==255)
    led=0;
    TCNT0=0  ;                          /* 重新加载定时器寄存器 */
    }
void main( void )
    {
    DDRB=0xFF;                          /* PORTB 所有端口设为输出 */
    TIMSK=(1<<TOIE0);                   /* 允许 TIMER0 溢出中断 */
    TCNT0=0;                            /* 定时/计数器 0 的初始值为 0 */
    TCCR0=5;                            /* 使用 1024 分频( ck/1024) */
    led=0;
    SEI();                              /* 全局中断允许 */
    for (;;)                            /* 循环等待中断处理 */
        ;
    }
```

6.5.1.2　计数器应用

在计数器模式下，TIMER0 计算 T0 引脚上的脉冲，当然，也可以利用这个功能使用外部的时钟源实现定时功能。下面分别给出查询和中断方式下计数器应用的例子，需要注意的是，T0(PB0)引脚的方向需要设置为输入，触发方式设为下降沿触发。

1．查询方式

使用查询方式操作计数器的程序如下：

```
/*
** 查询方式计数器 0 应用
** 如果计数器溢出，TIFR 寄存器中的溢出中断标志 TOV0 会被置位
** 溢出后 led 的值加 1，输出到 PORTB 驱动发光 led
*/
#include "avr/io.h"
uint8_t led;
uint8_t state;
void main( void )
    {
    outp(0xFE, DDRB);               /* PINB7-PINB6 用于输出，PINB0 用于输入(T0) */
    outp(0xFE, TCNT0);              /* 计数器初始值 */
    outp(6, TCCR0);                 /* 配置 TIMER0 为 T0 下降沿触发式计数器 */
    led=2;
```

```
    for (;;)
         {
         do                              /* 循环检测溢出是否发生  */
            state=inp(TIFR) & 0x01;
         while (state != 0x01);
         outp(0xFE, TCNT0);              /* 初始化计数器 */
         outp(~led,PORTB);
         led++;
         if (led==255)
            led=0;
         /* 写逻辑"1"到 TIFR 的 TOV0 位来清楚溢出标志位 TOV0 */
         outp((1<<TOV0),TIFR);
         }
    }
```

相应的，给出 ICCAVR 查询方式操作计数器的程序：

```
/*
** 查询方式计数器 0 应用
** 如果计数器溢出，TIFR 寄存器中的溢出中断标志 TOV0 会被置位
** 溢出后 led 的值加 1，输出到 PORTB 驱动发光 led
*/
#include "io8515v.h"
unsigned char led;
void main( void )
    {
    DDRB=0xFE;                       /* PINB7~PINB6 用于输出,PINB0 用于输入(T0) */
    TCNT0=0xFE;                      /* 计数器初始值 */
    TCCR0=6;                         /* 配置 TIMER0 为 T0 下降沿触发式计数器 */
    led=2;
    for (;;)
    {
         while (!(TIFR & 0x01));     /* 循环检测溢出是否发生  */
         TCNT0=0xFE;                 /* 初始化计数器 */
         PORTB=led;
         led++;
         if (led==255)
              led=0;
         /* 写逻辑"1"到 TIFR 的 TOV0 位来清楚溢出标志位 TOV0 */
         TIFR=(1<<TOV0);
         }
    }
```

2. 中断方式

使用中断方式操作计数器的程序如下：

```
/*
** 中断方式计数器 0 应用
** 如果计数器溢出，TIFR 寄存器中的溢出中断标志 TOV0 会被置位
** 溢出后 led 的值加 1，输出到 PORTB 驱动发光 led
*/
#include "avr/io.h"
#include "avr/interrupt.h"
#include "avr/signal.h"
uint8_t led;
SIGNAL (SIG_OVERFLOW0)
   {
   outp(~led, PORTB);                    /* 输出 led 的值到 PORTB */
   led++;
   if (led==255)
        led-2;
   outp(0xFE,TCNT0);                     /* 重新加载定时器寄存器 */
   }
void main( void )
   {
   outp(0xFE, DDRB);                     /* PINB7-PINB6 用于输出，PINB0 用于输入(T0) */
   outp((1<<TOIE0), TIMSK);              /* 允许 TIMER0 溢出中断 */
   outp(0xFE, TCNT0);                    /* 初始化计数器 */
   outp(6, TCCR0);                       /* 配置 TIMER0 为 T0 下降沿触发式计数器 */
   led=0;
   sei(); /* 全局中断允许 */
   for (;;)
        ;
   }
```

相应的，给出 ICCAVR 中断方式操作计数器的程序：

```
/*
** 中断方式计数器 0 应用
** 如果计数器溢出，TIFR 寄存器中的溢出中断标志 TOV0 会被置位
** 溢出后 led 的值加 1，输出到 PORTB 驱动发光 led
*/
#include "io8515v.h"
unsigned char  led;
```

```
#pragma interrupt_handler timer0_isr: 7
void timer0_isr(void)
    {
    PORTB=led;                          /* 输出 led 的值到 PORTB */
    led++;
    if (led==255)
    led=2;
    TCNT0=0xFE;                         /* 重新加载定时器寄存器 */
    }
void main( void )
    {
    DDRB=0xFE;                          /* PINB7-PINB6 用于输出,PINB0 用于输入(T0) */
    TIMSK=(1<<TOIE0);                   /* 允许 TIMER0 溢出中断 */
    TCNT0=0xFE;                         /* 初始化计数器 */
    TCCR0=6;                            /* 配置 TIMER0 为 T0 下降沿触发式计数器 */
    led=0;
    SRI();                              /* 全局中断允许 */
    for (;;)
        ;
    }
```

6.5.2 定时/计数器 1

定时/计数器 1 是 16 位的，寄存器值从 0x0000~0xFFFF，可以实现较长时间的定时和较大的计数。除了定时模式和计数模式，定时/计数器 1 还有比较模式、捕获模式和 PWM 模式。因为 TIMER1 的定时模式同 TIMER0 相似，读者可参考前面的叙述，本节只介绍 TIMER1 的后面三种工作模式。

表 6.14 与 TIMER1 有关的几个寄存器

类 别	名 称	名 称
定时器寄存器	定时/计数器控制寄存器	TCCR1A (Timer/Counter Control Register A)
		TCCR1B (Timer/Counter Control Register B)
	定时/计数器寄存器	TCCR1L (Timer/Counter Value Low Byte)
		TCCR1H (Timer/Counter Value High Byte)
	比较 A 寄存器	OCR1AL (Output Compare Register A Low Byte)
		OCR1AH (Output Compare Register A High Byte)
	比较 B 寄存器	OCR1BL (Output Compare Register B Low Byte)
		OCR1BH (Output Compare Register B High Byte)
	捕获寄存器	ICR1L (Input Capture Register Low Byte)
		ICR1H (Input Capture Register High Byte)

续表

类　别	名　称	名　称
中断寄存器	中断标志寄存器	TIFR (Timer Interrupt Flag Register)
	中断屏蔽寄存器	TIMSK (Timer Interrupt Mask Register)
	全局中断屏蔽寄存器	GIMSK (General Interrupt Mask Register)

6.5.2.1　比较模式

定时/计数器1通过比较OCR1A/OCR1B寄存器(16位)和TCNT1寄存器(16位)，提供两个输出比较应用OCR1A和OCR1B。输出比较功能包括比较A匹配发生时清除计数器和比较输出引脚，包括OC1A(PIND.5)和OC1B的动作。比较1模式选择如表6.15所示。

表 6.15　比较 1 模式选择

COM1X1	COM1X0	说　明
0	0	计数器/定时器 1 与输出引脚 OC1X 不连接
0	1	触发 OC1X 输出线
1	0	清除 OC1X 输出线(为 0)
1	1	设置 OC1X 输出线(为 1)

其中，X=A 或 B。

定时/计数器 1 分频选择和定时/计数器 0 相同，这里不再多讲。下面给出使用比较匹配的例子程序，它可以用在 STK-200 开发板上，也可以用在图 6.8 所示的系统中。

```
/*
** 比较匹配测试程序，定时/计数器 1 比较匹配中断将驱动 PORTB 口的发光 LED
** S1 增加延迟时间；S2 减少延迟时间
*/
#include "avr/io.h"
#include "avr/interrupt.h"
#include "avr/signal.h"
uint8_t delay;
SIGNAL (SIG_OUTPUT_COMPARE1A)          /*比较匹配中断服务程序 */
  {
  if (delay==0)
       outp(0XFF, PORTB);
  else
       outp(0XFE, PORTB);
  }
SIGNAL (SIG_OVERFLOW1)                 /* 定时/计数器 1 溢出中断服务程序 */
  {
  outp(0XFF, PORTB);
```

```
    outp(delay,TCNT1H);
    outp(0,TCNT1L);
    }
SIGNAL (SIG_INTERRUPT0)      /* PD2, INT0 中断，减少延迟时间*/
    {
    if (delay < 15)
    delay = 0;
    else
    delay=delay - 15;
    }
SIGNAL (SIG_INTERRUPT1)      /* PD3. INT1 中断，增加延迟时间 */
    {
    if (delay>235)
        delay=250;
    else
        delay=delay + 15;
    }
void main( void )
    {
    outp(0xFF, DDRB);          /* PORTB 所有位定义为输出 */
    outp(0x00, DDRD);          /* PORTD 所有位定义为输入 */
    delay=120;                 /* 定时/计数器 1 的高字节初始值 */
                               /* Switches PB3,PB4 for Interrupt 0 and 1 */
    outp((1<<TOIE1)|(1<<OCIE1A), TIMSK);     /*定时/计数器 1 溢出中断使能，比
                                              较 1A 中断使能 */
    outp((1<<INT0)|(1<<INT1), GIMSK);        /*外部中断 INT0,INT1 使能*/
    outp((1<<ISC01)|(1<<ISC10)|(1<<ISC11), MCUCR);   /*INT0 下降沿触发，
                                                       INT1 上升沿触发 */
    outp(delay, TCNT1H);                     /* 初始化 TCNT1 */
    outp(0, TCNT1L);
    outp(0XFF, OCR1AH);                      /* 初始化比较匹配 OCR1A */
    outp(0X10, OCR1AL);
    outp(0, TCCR1A);                         /* 比较匹配 OCR1A 部输出(不连接
                                                到 OC1A(PIND.5 引脚) */
    outp(1, TCCR1B);                         /* 选择系统时钟(CK)作为时钟源 */
    sei();
    for (;;)
        ;
    }
```

编者觉得这个例子很有意思，定时/计数器 1 比较匹配中断和溢出中断都可以驱动 PORTB 口的 LED。主程序里初始化定时/计数器 1 寄存器和比较匹配寄存器，在定时器溢

出以前，比较匹配中断触发，其中断服务程序打开 PORTB 口的 LED；定时器继续运行直到它溢出，定时器溢出中断服务程序关闭 PORTB 口的 LED，并重新初始化定时器。连在 PD2(INT0)和 PD3(INT1)上按键 S1 和 S2 分别触发 INT0 和 INT1 中断，按下 S1 使 delay 减少 15，而按下 S2 使 delay 增加 15。

作为对比，给出 ICCAVR 下的源程序：

```
/*
** 比较匹配测试程序，定时/计数器 1 比较匹配中断将驱动 PORTB 口的发光 LED
** S1 增加延迟时间；S2 减少延迟时间
*/
#include "io8515v.h"
unsigned char delay;
#pragma interrupt_handler timer1_compa_isr:5
void timer1_compa_isr(void)     /*比较匹配中断服务程序 */
    {
    if(delay==0)
    PORTB=0xFF;
    else
    PORTB=0xFE;
    }
#pragma interrupt_handler timer1_ovf_isr:7
void timer1_ovf_isr(void)       /* 定时/计数器 1 溢出中断服务程序 */
    {
    PORTB=0xFF;
    TCNT1H=delay;
    TCNT1L=0;
    }
#pragma interrupt_handler int0_isr:2
void int0_isr(void)             /* PD2, INT0 中断，减少延迟时间*/
    {
    if(delay<15)
         delay=0;
    else
         delay-=15;
    }
#pragma interrupt_handler int1_isr:3
void int1_isr(void)             /* PD3, INT1 中断，增加延迟时间 */
    {
    if (delay>235)
         delay=250;
    else
```

```
        delay+=15;
    }
void main( void )
    {
    DDRB=0xFF;                          /* PORTB 所有位定义为输出 */
    DDRD=0x00;                          /* PORTD 所有位定义为输入 */
    delay=120;                          /* 定时/计数器 1 的高字节初始值 */
                                        /* Switches PB3,PB4 for Interrupt 0 and 1 */
    TIMSK=(1<<TOIE1)|(1<<OCIE1A);/*定时/计数器 1 溢出中断使能，比较 1A 中断使能 */
    GIMSK=(1<<INT0)|(1<<INT1);  /*外部中断 INT0,INT1 使能*/
    MCUCR=(1<<ISC01)|(1<<ISC10)|(1<<ISC11);/*INT0 下降沿触发，INT1 上升沿触发 */
    TCNT1H=delay;                       /* 初始化 TCNT1 */
    TCNT1L=0;
    COR1AH=0xFF;                        /* 初始化比较匹配 OCR1A */
    OCR1AL=0x10;
    TCCR1A=0;                           /* 比较匹配OCR1A部输出(不连接到OC1A(PIND.5引脚) */
    TCCR1B=1;                           /* 选择系统时钟(CK)作为时钟源 */
    SEI();
    for (;;)
        ;
    }
```

6.5.2.2 捕获模式

计数器/定时器 1 提供输入捕获(ICP)功能，当输入捕捉引脚 ICP 发生相应事件时，计数器/定时器 1 的值将被传到输入捕捉寄存器 ICR1。实际的捕获事件设置由计数器/定时器 1 的控制寄存器 TCCR1B 来定义。另外，模拟比较器也可以触发该输入捕获事件，详细的描述请参考第 4 章的有关内容。

下面给出捕获模式应用的程序：

```
/*
** 定时/计数器 1 捕获中断测试程序
** 按下 PIND.6(ICP)按键，定时/计数器 1 寄存器值输出到 PORTB 驱动 LED
*/
#include "avr/io.h"
#include "avr/interrupt.h"
#include "avr/signal.h"
SIGNAL (SIG_OVERFLOW1)
  {
  outp(0,TCNT1H);                       /* 溢出后重新加载定时/计数器寄存器 */
  outp(0,TCNT1L);
```

```
    }
SIGNAL (SIG_INPUT_CAPTURE1)
    {
    /* 读出定时/计数器寄存器值并输出到 PORTB 口驱动 LED */
    outp(~inp(ICR1L),PORTB);
    outp(~inp(ICR1H),PORTB);
    }
void main( void )
    {
    outp(0xFF, DDRB);                    /* 定义 PORTB 所有端口为输出 */
    outp(0x00, DDRD);                    /* 定义 PORTB 所有端口为输入(按键) */
    outp(0xFF, PORTB);

outp((1<<TICIE1)|(1<<TOIE1), TIMSK);/*定时/计数器 1 捕获中断使能,溢出中断使能 */
outp(0, TCNT1H);                     /* 初始化定时/计数器 1 */
outp(0, TCNT1L);
outp(0, TCCR1A);
outp(5, TCCR1B);                     /*使用 1024 分频时钟源 */
sei();
for (;;)
      ;
}
```

同样给出使用 ICCAVR 的源程序：

```
/*
** 定时/计数器 1 捕获中断测试程序
** 按下 PIND.6(ICP)按键，定时/计数器 1 寄存器值输出到 PORTB 驱动 LED
*/
#include "io8515v.h"
#pragma interrupt_handler timer1_capt_isr:4
void timer1_capt_isr(void)
    {
    /* 读出定时/计数器寄存器值并输出到 PORTB 口驱动 LED */
    PORTB=~ICR1L;
    PORTB=~ICR1H;
    }
#pragma interrupt_handler timer1_ovf_isr:7
void timer1_ovf_isr(void)
    {
    TCNT1H=0x00;                     /* 溢出后重新加载定时/计数器寄存器 */
    TCNT1L=0x00;
```

```
    }
void main( void )
    {
    DDRB=0xFF;                          /* 定义 PORTB 所有端口为输出 */
    DDRD=0x00;                          /* 定义 PORTB 所有端口为输入(按键) */
    PORTB=0xFF;                         /* 关闭所有 LED */
    TIMSK=(1<<TICIE1)|(1<<TOIE1);       /*定时/计数器 1 捕获中断使能，溢出中断使能 */
    TCNT1H=0x00;                        /* 初始化定时/计数器 1 */
    TCNT1L=0x00;
    TCCR1A=0x00;
    TCCR1B=0x05;                        /*使用 1024 分频时钟源 */
    SEI();
    for (;;)
        ;
    }
```

6.5.2.3 PWM 模式

AT90S8515 的定时/计数器 1 工作在 PWM(脉宽调制)模式时，它和输出比较寄存器 OCR1A/OCR1B 共同组成两个 8、9 或 10 位自由运行的 PWM，且在引脚 PD5(OC1A)和 OC1B 上带有输出的节拍修正 PWM。计数器/定时器 1 作为向上/向下的计数器，从 0x0000 向上记数到 TOP，在重复循环之前，它反转，并向下记数到 0x0000。

当计数器中的数值和 OCR1A/OCR1B 的数值低 8、9 或 10 位一致时，PD5(OC1A)/OC1B 引脚根据在计数器/定时器 1 控制寄存器 TCCR1A 中 COM1A0/COM1A1 和 COM1B0/COM1B1 的设置被设置或清除。更加详细的内容请参考第 4 章的相关内容。

下面是 10 位 PWM 模式的实例：

```
/*
** 定时/计数器 1 PWM 模式测试程序
** 定时/计数器 1 工作于自由运行的 10 位 PWM 模式
** 从 0x000 到 0x3FF 加 1(向上)计数，当寄存器的值达到 0x3FF 时，减 1(向下)计数到 0x000，然后再加 1 计数到 0x3FF，如此循环
** 当向上计数到与比较寄存器 OC1A 匹配时，OC1A(PD5)脚电平被拉低；
** 当向下计数到与比较寄存器 OC1A 匹配时，OC1A(PD5)脚电平被拉高；
** 当向上计数到与比较寄存器 OC1B 匹配时，OC1B 脚电平被拉高；
** 当向下计数到与比较寄存器 OC1A 匹配时，OC1B 脚电平被拉低；
*/
#include "avr/io.h"
void main( void )
    {
    outp(0xFF, DDRD);                   /* PORTD 所有端口作为输出 */
```

```
    outp(0xB3, TCCR1A);                    /* 初始化定时器，1024 分频 */
    outp(0x5, TCCR1B);
    outp(0x00, TCNT1H);
    outp(0x00, TCNT1L);                    /* 计数器初始化 */
    outp(0x00, OCR1AH);
    outp(0xFF, OCR1AL);                    /* 初始化比较寄存器 OC1A */
    outp(0x00, OCR1BH);
    outp(0xFF, OCR1BL);                    /* 初始化比较寄存器 OC1B */
    for (;;)
        ;
    }
```

程序在 OC1A(PD5)和 OC1B 引脚上输出 10 位模式的 PWM 波形，OC1A 配置为 COM1A1:COM1A0=10，即向上计数匹配清零，向下计数匹配置位。而 OC1B 配置为 COM1B1:COM1B0 = 11，即向上计数匹配置位，向下计数匹配清零。

相应的，给出 ICCAVR 实现 10 位 PWM 的源程序：

```
/*
** 定时/计数器 1 PWM 模式测试程序
** 定时/计数器 1 工作于自由运行的 10 位 PWM 模式
** 从 0x000 到 0x3FF 加 1(向上)计数，当寄存器的值达到 0x3FF 时，减 1(向下)计数到 0x000，
然后再加 1 计数到 0x3FF，如此循环
** 当向上计数到与比较寄存器 OC1A 匹配时，OC1A(PD5)脚电平被拉低；
** 当向下计数到与比较寄存器 OC1A 匹配时，OC1A(PD5)脚电平被拉高；
** 当向上计数到与比较寄存器 OC1B 匹配时，OC1B 脚电平被拉高；
** 当向下计数到与比较寄存器 OC1A 匹配时，OC1B 脚电平被拉低；
*/
#include "io8515v.h"
void main( void )
    {
    DDRD=0xFF;                             /* PORTD 所有端口作为输出 */
    TCCR1A=0xB3;                           /* 初始化定时器，1024 分频 */
    TCCR1B=0x05;
    TCNT1H=0x00;
    TCNT1L=0x00;                           /* 计数器初始化 */
    OCR1AH=0x00;
    OCR1AL=0xFF;                           /* 初始化比较寄存器 OC1A */
    OCR1BH=0x00;
    OCR1BL=0xFF;                           /* 初始化比较寄存器 OC1B */
    for (;;)
;
}
```

6.6 通用异步串行通信 UART

AVR 可以通过异步串口 UART 与 PC 或其他设备进行通信，AT90S8515 提供一个全双工 UART，具有以下几个特点：

1．波特率发生器可以生成任何波特率；
2．在低频率的系统时钟下能得到较高的波特率；
3．8 位和 9 位数据；
4．噪声滤波；
5．过速误差检测；
6．帧错误检测；
7．错误起始位检测；
8．3 个独立的中断源：TX 完成，TX 数据寄存器空和 RX 完成。

与 UART 相关的几个寄存器如表 6.16 所示。

波特率的计算公式：

表 6.16 与 UART 相关的寄存器

	名 称	名 称
数据寄存器	UART 数据寄存器	UDR(UART I/O Register)
控制	状态寄存器	USR(UART Status Register)
	控制寄存器	UCR(UART Control Register)
	波特率寄存器	UBRR(BAND Rate Generater)

UBRR=fosc/(波特率*16)−1

其中，UBRR 是波特率寄存器的值；fosc 是系统时钟频率；波特率是需要的设定的波特率。比如使用 4MHz 晶振，需要 9600bps 的速率，即 fosc=4MHz，波特率=9600，则有：

UBRR=4M/(9600*16)−1=25

这就是说 UBRR 应该初始化为 25，UART 就在 9600bps 速率下工作。各个寄存器的详细使用请参考第 4 章的相关内容。

6.6.1 发送数据

下面分别给出 UART 中断和查询方式下操作的实例。

1．中断方式发送数据

```
/*
** UART 发送的测试程序
** 使用中断方式发送一串字符串到 UART
*/
#include "avr/io.h"
#include "ave/interrupt.h"
#include "avr/signal.h"
#include "avr/pgmspace.h"
/* 声明一个存放在程序空间的字符串 a[],长度为 23 */
char a[] __attribute__((progmem))="This is a test message!";
uint8_t pos=0;
SIGNAL(SIG_UART_TRANS)                    /*UART 发送完成中断 */
  {
  if (pos++<23)
       outp(PRG_RDB(&a[pos]), UDR);     /* 从程序空间读出数据送到 UART 串口 */
  }
void main(void)
  {
  outp((1<<TXCIE)|(1<<TXEN),UCR);/* 允许 UART 发送数据，UART 发送完成中断使能 */
  outp(25, UBRR);                  /* 初始化波特率产生器 */
  sei();                           /* 打开全局中断 */
  outp(PRG_RDB(&a[pos]), UDR); /* 先写一个字节到 UART，以触发 UART 发送完成中断 */
  for(;;)
      ;
  }
```

相对应的，给出 ICCAVR 中断方式发送 UART 数据的源程序：

```
/*
** UART 发送的测试程序
** 使用中断方式发送一串字符串到 UART
*/
#include "io8515v.h"
/* 声明一个存放在程序空间的字符串 a[],长度为 23 */
const char a[]="This is a test message!";
unsigned char pos=0;
#pragma interrupt_handler uart0_tx_isr:12
void uart0_tx_isr(void)
   {
   if (pos++<23)
        UDR=a[pos];          /* 从程序空间读出数据送到 UART 串口 */
   }
```

```
void main(void)
    {
    UCR=(1<<TXCIE)|(1<<TXEN);     /* 允许 UART 发送数据，UART 发送完成中断使能 */
    UBRR=25;                      /* 初始化波特率产生器 */
    SEI();                        /* 打开全局中断 */
    UDR=a[pos];                   /* 先写一个字节到 UART，以触发 UART 发送完成中断 */
    for(;;)
        ;
    }
```

2. 查询方式发送数据

```
/*
** UART 发送的测试程序
** 使用查询方式发送一串字符串到 UART
*/
#include "avr/io.h"
#include "avr/interrupt.h"
#include "avr/signal.h"
#include "avr/pgmspace.h"
/* 声明一个存放在程序空间的字符串 a[],长度为 23 */
char a[] __attribute__((progmem))="This is a test message!";
uint8_t pos=0;
void sendchar(unsigned char c)      /*查询方式发送数据 */
    {
    outp(c, UDR);
    while(!(inp(USR)&(1<<TXC)));
    sbi(USR,TXC);
    }
void main(void)
    {
    outp((1<<TXEN),UCR);            /* 允许 UART 发送数据 */
    outp(25, UBRR);                 /* 初始化波特率产生器 */
    while(pos<23)
        {
        sendchar(PRG_RDB(&a[pos]));
        pos++;
        }
    for(;;)
        ;
    }
```

同样的，给出 ICCAVR 的源程序：

```
/*
** UART 发送的测试程序
** 使用查询方式发送一串字符串到 UART
*/
#include "io8515v.h"
/* 声明一个存放在程序空间的字符串 a[],长度为 23 */
const char a[]="This is a test message!";
unsigned char pos=0;
void sendchar(unsigned char c)      /*查询方式发送数据 */
    {
    UDR=c;
    while(!(USR&0x40));
    USR|=0x40;
    }
void main(void)                     /* 允许 UART 发送数据 */
    {
    UCR=(1<<TXEN);
    UBRR=25;                        /* 初始化波特率产生器 */
    CLI();                          /* 关闭全局中断 */
    while(pos<23)
        {
        send(a[pos]);               /* 写一个字节到 UART */
        pos++;
        }
    for(;;)
        ;
    }
```

6.6.2　接收数据

1. 中断方式

```
/*
** UART 接收的测试程序
** 使用中断方式接收一个字符，并送到 PORTB 驱动发光 LED
*/
#include "avr/io.h"
#include "avr/interrupt.h"
#include "avr/signal.h"
```

```
SIGNAL(SIG_UART_RECV)
   {
   /* 从 UART 读取一个字节输出到 PORTB 驱动发光 LED */
   outp(inp(UDR),PORTB);
   }
void main(void)
   {
   outp(0Xff,DDRB);
   outp((1<<RXCIE)|(1<<RXEN),UCR);   /* UART 接收允许，接收完成中断使能 */
   outp(25, UBRR);                   /* 初始化波特率寄存器，9600bps */
   sei();                            /* 打开全局使能 */
   for(;;)
       ;
   }
```

给出 ICCAVR 的源程序：

```
/*
** UART 接收的测试程序
** 使用中断方式接收一个字符，并送到 PORTB 驱动发光 LED
*/
#include "io8515v.h"
#pragma interrupt_handler uart0_rx_isr:10
void uart0_rx_isr(void)
    {
    /* 从 UART 读取一个字节输出到 PORTB 驱动发光 LED */
    PORTB=UDR;
    }
void main(void)
    {
    DDRB=0xFF;
    UCR=(1<<RXCIE)|(1<<RXEN);        /* UART 接收允许，接收完成中断使能 */
    UBRR=25;                         /* 初始化波特率寄存器，9600bps */
    SEI();                           /* 打开全局使能 */
    for(;;)
        ;
    }
```

2. 查询方式

```
/*
** UART 接收的测试程序
```

```
** 使用查询方式接收一个字符，并送到 PORTB 驱动发光 LED
*/
#include "avr/io.h"
#include "avr/interrupt.h"
#include "avr/signal.h"
uint8_t  recvchar(void)
    {
    while(!(inp(USR)&(1<<RXC)));
    return inp(UDR);
    }
void main(void)
    {
    uint8_t t;
    outp(0Xff,DDRB);
    outp((1<<RXEN),UCR);                    /* UART 接收允许，接收完成中断使能 */
    outp(25, UBRR);                         /* 初始化波特率寄存器，9600bps */
    for(;;)
         {
         t=recvchar();
         outp(PORTB,t);                     /*从UART 读取一个字节输出到 PORTB 驱动发光 LED */
         }
    }
```

同样给出 ICCAVR 的源程序：

```
/*
** UART 接收的测试程序
** 使用查询方式接收一个字符，并送到 PORTB 驱动发光 LED
*/
#include "io8515v.h"
unsigned char  recvchar(void)
    {
    while(!(USR&0x80));
    return UDR;
    }
void main(void)
    {
    DDRB=0xFF;
    UCR=(1<<RXEN);                          /* UART 接收允许 */
    UBRR=25;                                /* 初始化波特率寄存器，9600bps */
    for(;;)
         {
```

```
        /*从 UART 读取一个字节输出到 PORTB 驱动发光 LED */
        PORTB=recvchar();
        }
}
```

6.7 库 函 数

本节所介绍的库函数是针对 20030115 版的，安装文件为 Winavr-20030115-bin-install，其他版本的可能会与本节介绍的库函数略有差别，但一般不会影响使用。

6.7.1 头文件介绍

库函数按不同的类别声明在不同的头文件中，以字母为序分别介绍头文件：

ctype.h：字符类型函数
eeprom.h：EEPROM 访问函数
errno.h：错误处理函数
ina90.h：与 IAR C 兼容的头文件
interrupt.h：中断处理函数
inttypes.h：定义不同的数据类型
io.h：包含寄存器定义和其他头文件
math.h：数学函数
pgmspace.h：与 IAR C 兼容的头文件
progmem.h：与 pgmspace.h 头文件相同
setjmp.h：长跳转函数
sig-avr.h：与 signal,h 相同，旧版头文件，建议不使用
signal.h：信号处理函数
stdlib.h：标准库函数
stdio.h：标准输入输出函数
string.h：字符串操作函数
timer.h：定时器控制函数
twi.h：针对 ATmega163 的 I2C 函数
wdt.h：看门狗定时器控制函数

6.7.2　库函数功能介绍

下面按头文件(即功能)介绍库函数。

1．字符类型函数 ctype.h

在使用前应包含头文件#include "ctype.h"

int isalnum(int c)：如果 c 为字母或数字则返回 1，否则返回 0。

int isalpha(int c)：如果 c 为字母则返回 1，否则返回 0。

int isascii(int c)：如果 c 为 ASCII 码则返回 1，否则返回 0。

int isblank(int c)：如果 c 为空格字符则返回 1，否则返回 0，可以是 space 键或 tab 键输入的空格字符。

int iscntrl(int c)：如果 c 为控制字符则返回 1，否则返回 0。

int isdigit(int c)：如果 c 为数字则返回 1，否则返回 0。

int isgraph(int c)：如果 c 为可打印字符(不包括空格)则返回 1，否则返回 0。

int islower(int c)：如果 c 为小写字母则返回 1，否则返回 0。

int isprint(int c)：如果 c 为可打印字符(包括空格)则返回 1，否则返回 0。

int ispunct(int c)：如果 c 为标点符号则返回 1，否则返回 0。

int isspace(int c)：如果 c 为空格、'\n'、'\f'、'\r'、'\t'、'v'之一则返回 1，否则返回 0。

int isupper(int c)：如果 c 为大写字母则返回 1，否则返回 0。

int isxdigit(int c)：如果 c 为 16 进制数则返回 1，否则返回 0。

int toascii(int c)：将 c 转换为 7 位 ASCII 字符，成功返回 1，否则返回 0。

int tolower(int c)：将 c 转换为小写字母，成功返回 1，否则返回 0。

int toupper(int c)：将 c 转换为大写字母，成功返回 1，否则返回 0。

2．EEPROM 访问函数 eeprom.h

新版本要求将文件包含改为：#include "\avr\eeprom.h"

void eeprom_read_block(void *buf, unsigned int addr, size_t n)：从 EEPROM 的 addr 地址开始，读取 n 个字节到 buf。

int eeprom_is_ready()：宏定义，EEPROM 准备好(EECR=0)返回非 0，否则返回 0。

unsigned char eeprom_read_byte (unsigned int addr)：从 EEPROM 读取 addr 地址处的字节数据，在以前的版本中为 unsigned char eeprom_rb(unsigned int addr)，新版也兼容旧版本的函数。

unsigned int eeprom_read_word(unsigned int addr)：从 EEPROM 读取一个整数，低字节地址 addr，高字节地址(addr+1)，在以前的版本中为 unsigned int eeprom_rw(unsigned int addr)，新版本也兼容旧版本的函数。

void eeprom_write_byte (unsigned int addr, unsigned char val)：向 EEPROM 的 addr 地址处写入字节数据 val，在以前的版本中为 void eeprom_wb(unsigned int addr, unsigned char

val)，新版本也兼容旧版本的函数。

_EEPUT(addr, val) eeprom_wb(addr, val)：与 IAR C 的兼容函数

_EEGET(var, addr) (var) = eeprom_rb(addr)：与 IAR C 的兼容函数

3．错误处理函数 errno.h

在使用前应包含头文件#include "errno.h"

```
#define  EDOM     33
#define  ERANGE   34
```

通常 EDOM 和 ERANGE 扩展成各种非零的整型常数表达式，适用于预处理指令#if。

int errno：检测系统的错误

在调用函数前应先将其置为零，并且应在调用下一个函数前检查其值，用以判断被调用的函数是否出错。

4．与 IAR C 兼容的头文件 ina90.h

由于 IAR 为适用于 AVR 单片机中最早推出的 C 编译器，后来推出的新编译器如 ICCAVR、CVAVR 等都支持(至少尽量支持)直接使用 IAR C 语言编写的源文件，因此在 GCCAVR 中提供本头文件也是为了这个目的。新版本要求将文件包含改为：#include "\avr\ina90.h"，包含这个头文件后，就可以直接使用 IAR 中的一些常用函数及宏定义，现将部分常用的宏定义列举如下：

```
#define _CLI() __asm__ __volatile__ ("cli")
#define _SEI() __asm__ __volatile__ ("sei")
#define _NOP() __asm__ __volatile__ ("nop")
#define _WDR() __asm__ __volatile__ ("wdr")
#define _SLEEP() __asm__ __volatile__ ("sleep")
#define _OPC(op) __asm__ __volatile__ (".word %0" : : "n" (op))
```

5．中断处理函数 interrupt.h

新版本要求将文件包含改为：#include "\avr\interrupt.h"

sei()：宏定义，中断使能

cli()：宏定义，关闭中断

void enable_external_int(unsigned char ints)：写 ints 值到 EIMSK 或 GIMSK

void timer_enable_int(unsigned char ints)：写 ints 值到 TIMSK

6．定义不同的数据类型 inttypes.h

在使用前应包含头文件#include "inttypes.h"，在本头文件中重新进行数据类型定义，将 GNU 的数据类型改为符合 ANSI C 的规范。

typedef signed char int8_t：8 位有符号数

typedef unsigned char uint8_t：8 位无符号数

typedef int int16_t：16 位有符号数

typedef unsigned int uint16_t：16 位无符号数

typedef long int32_t：32 位有符号数

typedef unsigned long uint32_t：32 位无符号数

typedef long long int64_t：64 位有符号数

typedef unsigned long long uint64_t：64 位无符号数

Typedef int16_t intptr_t：指针

Typedef uint16_t uintptr_t：指针

注意：在使用“-mint8”选项时，编译器将把 int 类型定义为 8 位。

7．包含寄存器定义和其他头文件 io.h

包含寄存器定义和其他 AVR 芯片的头文件，本头文件包含的内容较多，也较重要，请读者认真阅读。

8．数学函数 math.h

在使用前应包含头文件#include "math.h"，而且 math.h 数学函数库中定义的函数需要 libm.a 库文件支持，因此在连接时增加-lm 参数，或者在 makefile 的 LDFLAGS=后面追加-lm 参数。

M_PI：常数，为 3.14159265358979323846264 3

M_SQRT2：常数，为 1.41421356237309504880168872

double cos(double x)：返回以弧度形式表示 x 的余弦值

double fabs(double x)：返回 x 的绝对值

double fmod(double x, double y)：返回 x/y 的余数

double modf(double x, double *iptr)：把浮点数分解成整数部分和小数部分，整数部分存放到 pint 指向的变量，小数部分应当大于或等于 0 而小于 1，并且作为函数返回值返回

double sin(double x)：返回以弧度形式表示 x 的正弦值

double sqrt(double x)：返回 x 的平方根

double tan(double x)：返回以弧度形式表示 x 的正切值

double floor(double x)：返回不大于 x 的最大整数

double ceil(doublce x)：返回不小于 x 的最小整数

double frexp(double x, int *exp)：把浮点数 x 分解成数字部分 y(尾数)和以 2 为底的指数 n 两个部分，即 $x=y*2^n$，y 的范围为 $0.5 \leqslant y < 1$，y 值被函数返回，而 n 值存放在 pexp 指向的变量中

double ldexp(double x, int exp)：返回 $x*2^{exp}$

double exp(double x)：返回以 e 为底 x 的幂，即 e^x

double cosh(double x)：以弧度形式返回 x 的双曲余弦值

double sinh(double x)：以弧度形式返回 x 的双曲正弦值

double tanh(double x)：以弧度形式返回 x 的双曲正切值

double acos(double x)：以弧度形式返回 x 的余弦值

double asin(double x)：以弧度形式返回 x 的正弦值

double atan(double x)：以弧度形式返回 x 的正切值

double atan2(double x, double y)：返回 y/x 的反正切，其范围在$-\pi \sim +\pi$之间

double log(double x)：返回 x 的自然对数

double log10(double x)：返回以 10 为底的对数

double pow(double x, double y)：返回 x^y 值

int isnan(double x)：如果 x 为非负数(NAN，值为 0xffffffff)，返回 1，否则返回 0

int isnaf(double x)：如果 x 的值为正溢出(+INF，值大于 0x7f800000)和负溢出(-INF，值小于 0xff800000)则返回 0，否则返回 1

double strtod(const char *s, char **endptr)：将 ASCII 字符串转换为 double 数

double square(double x)：返回 x 的平方根

double inverse(double x)：返回 1/x

9．pgmspace.h：

使源程序能与 IAR C 兼容的头文件

10．progmem.h

使源程序能与 IAR C 兼容的头文件，与 pgmspace.h 相同

11．长跳转函数 setjmp.h

在使用前应包含头文件#include "setjmp.h"

定义了一个用于保存调用环境信息(以便恢复)的数组类型

jmp_buf:

offset	大小	描述
0	16	call-saved registers(r2~r17)
16	2	frame pointer (r29 : r28)
18	2	stack pointer (SPH : SPL)
20	1	status register (SREG)
21	3	return address (PC) (2 bytes used for <=128K flash)

24 = total size

int setjmp(jmp_buf env)：将调用环境保存到 jmp_buf 参数中，以供后续的函数 longjmp 恢复该环境时使用。如果直接从函数调用返回，setjmp 返回 0，如果从函数 longjmp 的调用返回，则返回值为非零值。

void longjmp(jmp_buf env, int val)：在程序的同一次运行过程中恢复由最近一次 setjmp 调用所保存的环境，该环境取自于参数 jmp_buf，如果程序的运行环境已经改变，或者包含宏 setjmp 调用的函数中途停止执行，其结果是不确定的。

由于以上的调用绕过了正常的函数调用和返回机制，因此 longjmp 在中断和信号以及与这两者相关的其他函数环境下都能够正确执行，如果函数 longjmp 被一个嵌套的信号处理函数所调用，其结果是不确定。

12．信号处理函数 signal.h

在使用前应包含头文件#include "signal.h"，改自旧版本的 sig-avr.h 头文件，定义存储在程序起始处的中断向量符号：

SIG_INTERRUPT0 ~ SIG_INTERRUPT7：外部中断 0~7 的中断向量

SIG_OUTPUT_COMPARE2：拟比较 2 中断
SIG_OVERFLOW2：定时/计数器 2 溢出中断
SIG_INPUT_CAPTURE1：捕获 1 中断
SIG_OUTPUT_COMPARE1A：比较匹配 1A 中断
SIG_OUTPUT_COMPARE1B：比较匹配 1B 中断
SIG_OVERFLOW1：定时/计数器 1 溢出中断
SIG_OUTPUT_COMPARE0：模拟比较 0 中断
SIG_OVERFLOW0：定时/计数器 0 溢出中断
SIG_SPI：SPI 中断
SIG_UART_RECV：UART0 接收完成中断
SIG_UART1_RECV：UART1 接收完成中断
SIG_UART_DATA：UART0 数据寄存器空中断
SIG_UART1_DATA：UART1 数据寄存器空中断
SIG_UART_TRANS：UART0 发送完成中断
SIG_UART1_TRANS：UART1 发送完成中断
SIG_ADC：A/D 中断
SIG_EEPROM：EEPROM 中断
SIG_COMPARATOR：比较器中断
另外，还定义了以下两个函数：
SIGNAL(signame)：为信号 signame 定义中断服务程序。中断服务程序执行过程中全局中断使能位为 0，中断不能嵌套。
INTERRUPT(signame)：为信号 signame 定义中断服务程序。中断服务程序执行过程中全局中断使能位为 1，中断可以嵌套。
13．标准库函数 stdlib.h
在使用前应包含头文件#include "stdlib.h"
标准库定义了以下三种数据结构：

```
typedef struct      //函数 div_t 返回值的结构体类型
    {
    int quot;       //商
    int rem;        //余数
    } div_t;
typedef struct      //函数 ldiv_t 返回值的结构体类型
    {
    long quot;      //商
    long rem;       //余数
    } ldiv_t;
```

标准库中提供以下的宏和函数：

void abort()：死循环函数。当程序需异常中止时调用本函数，程序进入死循环。

int abs(int x)：返回 x 的绝对值

long int labs(long int x)：返回 x 的绝对值

void *bsearch(const void *key, const void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *))：从含 nmemb 个对象的数组(base 指向其第一个元素)中查找与 key 所指向的对象匹配的元素，数组中每个元素的大小由 size 指定，compar 所指向的比较函数带有两个指针参数，分别指向 key 对象和数组元素。如果 key 对象小于、等于或大于数组元素，函数将分别返回小于、等于或大于 0 的整数。数组的组成应按顺序：所有小于 key 对象的元素、所有等于 key 对象的元素、所有大于 key 对象的元素。函数 bsearch 返回指向数组中与 key 匹配的元素指针，若无匹配元素则返回空指针。

div_t div(int x, int y)：返回 x/y 的值，其值为一个结构体，包含商和余数，如果是非精确除法，计算出的商是比最接近代数商稍小的整数。

ldiv_t ldiv(long x, long y)：返回 x/y 的值，其值为一个结构体，包含商和余数，如果是非精确除法，计算出的商是比最接近代数商稍小的整数。

void qsort(void *base, size_t nmemb, size_t size, compar_fn_t compar)：排序含 nmemb 个对象的数组。base 指向初始元素，每个对象的大小由 size 指定。数组内容通过 compar_fn_t 和 compar 所指向的比较函数按升序排序。如果第一个参数小于、等于或大于第二个参数，则函数分别返回小于、等于或大于零的值。

unsigned long strtoul(const char *nptr, char **endptr, int base)：将 nprt 所指向的字符串的初始部分转换成 long int 类型数值。它首先将输入串分解成三部分：一个初始的空白字符(由函数 isspace 指定)序列(可能没有)，一个形似 base 为基数的整数的主导序列，和一个含有一到多个不可识别字符的尾串(包括终止输入串的空字符)。然后它将主导序列试着转换成一个整数，并返回结果。

long int strtol(const char *nptr, char **endptr, int base)：将 nptr 所指向的字符串的初始部分转换成 long int 类型数值。它首先将输入串分解成三部分：一个初始的空白字符(由函数 isspace 指定)序列(也可能没有)，一个类似 base 为基数整数的主导序列，和一个含有一到多个不可识别字符的尾串(包括终止输入串的空字符)，然后它将主导序列试着转换成一个整数，并返回结果。

如果 base 值为 0，主导序列的期望格式为一整型常数，前面带可选的正负号，但不包括整数后缀。如果 base 的值在 2 和 36 之间，主导序列的期望格式是一个以 base 为基数整数的字母数字序列，前面加可选的正负号，但不包括整数后缀。字母 a(或 A)到 z(或 Z)表示值 10 到 35，只允许出现数字值小于 base 值的字母，如果 base 值是 16，字母和数字序列前可以加上可选项字符 0x 或 0X 以及符号。

主导序列应定义为输入串中最长的初始序列，从第一个非空白字符开始，如果输入串为空或全部由空白组成，或者第一个非空白字符既不是符号又不是允许的字母或数字，则主导序列不包含任何字符。

如果主导序列具有期望格式且 base 值为 0，以第一个数字打头的字符序列被转换为整型常数，如果主导序列具有期望格式且 base 值从 2 到 36，则它被用作转换的基数，并将相应值赋予各个字母。如果主导序列以负号开头，则转换结果取负，如果 endptr 不是空指针，则指向尾串部分的指针存入 endptr 所指向的对象中。

如果没有转换操作函数 strtol 返回 0，如果转换后的值在可表示值域之外的值，根据值的符号返回 LONG_MAX 或 LONG_Min，并将宏 ERANGE 的值存入 eerno。

long int atol(const char *p)：转换字符串 p 的起始部分为长整型并返回它，字符串 p 起始必须是长整型字符串，否则返回 0。

int atoi(const char *p)：转换字符串 p 的起始部分为整型并返回它，字符串 p 起始必须是整型字符串，否则返回 0。

void exit(int status)：正常终止程序运行。

void *malloc(size_t size)：分配 size 字节的存储区，如果分配成功则返回内存区地址，如内存不够分配则返回空指针。

void free(void *ptr)：释放 ptr 所指向的内存区，使该空间能够用于再分配。

void *calloc(size_t nele, size_t size)：分配 nelem 个数据项的内存连续空间，每个数据项的大小为 size 字节，并且所分配的空间初始化为 0。如果成功，返回分配内存单元的首地址，否则返回空指针。

double strtod(const char *nptr, char **endptr)：将 nptr 所指向的字符串的初始部分转换成 deuble 类型数值。它首先将输入串分解成三部分：一个可能为空的前导空白字符(由函数 isspace 指定)序列，一个形似浮点常数的主导序列，和一个含有一到多个不可识别字符的尾串(包括终止输入串的空字符)，然后它将主导序列试着转换成一个整数，并返回结果。

主导序列的扩充形式是可选的加号或减号，然后是一个包含十进制小数点字符(可选)的非空数字序列，然后是一个可选的指数部分，但没有浮点后缀。主导序列定义为输入串中最长的原始子序列，它以第一个非空白字符开头，这是期望的格式。如果输入串为空或全部由空白字符组成，或者第一个非空白字符不是一个符号、数字或十进制小数点字符，那么该主导序列不包含任何字符。

如果主导序列是期望格式，以第一个数字或十进制小数点开头的字符序列被解释为一个浮点常数(十进制小数点字符代替圆点的情况除外)，既没有指数部分又没有十进制小数点字符出现时，则假定串的最后一个数字之后紧跟着一个十进制小数点。如果主导序列以负号开头，则转换结果取负，如果 endptr 不是空指针，指向尾串部分的指针存入 endprt 所指向的对象中。

如果主导序列为空或不是期望格式，不进行转换操作，如果 endptr 不是空指针，则 nptr 的值存入 endptr 所指向的对象中。

如果没有转换操作函数 strtod 返回 0，如果转换后的值溢出，根据值的符号返回正或负 HUGE_VAL，并将宏 ERANGE 的值存入 errno，如果转换后的值会导致下溢，则返回零值，并将宏 ERANGE 的值存入 errno。

int rand(void)：返回一个在 0 和 RAND_MAX(0x7FFF)之间的随机数

void srand(unsigned int seed)：用参数 seed 作为随机数种子，初始化随后调用 rand 产生伪随机数

int rand_r(unsigned long *ctx)：从 ctx 中取出随机数的种子数

char *itoa(int val, char *s, int radix)：按照 radix 指定的格式转换整数 val 为 s 指向的 ASCII 的字符串。

char *ltoa(long int val, char *s, int radix)：按照 radix 指定的格式转换长整数 val 为 s 指向的 ASCII 的字符串。

char *utoa(unsigned int val, char *s, int radix)：转换无符号整数为字符串

long random(void)：返回一个在 0 和 RAND_MAX(0x7FFFFFFF)之间的随机数

void srandom(unsigned long seed)：用参数 seed 作为随机数种子，初始化后调用 random 产生伪随机数

long random_r(unsigned long *ctx)：从 ctx 中取出随机数的种子数

char *dtostre(double val, char *s, unsigned char prec,unsigned char flags)：把浮点数转为 ASCII 码格式的字符串。

char *dtostrf(double val, char width, char prec, char *s)：把浮点数转为 ASCII 码格式的字符串

14．字符串操作函数 string.h

在使用前应包含头文件#include "string.h"

void * memccpy (void *dest,const void *src,int val,size_t len)：从 scr 复制不超过 len 个字节到 dest，在复制的过程中如果存在 val，则停止复制并返回，如果不存在 val，则复制 len 个字符。

void * memchr(const void *src,int val,size_t len)：在字符串 src 中搜索 len 个字节长度以寻找与 val 相同的字符，如果成功返回匹配字符的地址指针，否则返回 NULL

int memcmp(const void *s1,const void *s2,size_t len)：对字符串 s1 和 s2 的前 len 个字符进行比较，如果相同返回 0，如果 s1 中字符串大于 s2 中字符串，则返回大于 0 的值，如果 s1 中字符串小于 s2 中字符串，则返回小于 0 的值。

void * memcpy(void *dest,const void *src,size_t len)：src 所指的对象中复制 len 个字符至 dest 所指的对象中，函数返回 dest 的值。

void * memmove(void *dest,const void *src,size_t len)：src 中 len 个字符至 dest，其他与 memcpy 基本相同，但复制区可以重叠，在复制过程中使用了长度为 len 的临时数组。

void * memset(void *dest,int val,size_t len)：在 dest 中从开始位置填充 len 个字节的 val，它返回 dest

int strcasecmp(const char *s1,const char *s2)：比较两个字符串，如果相同返回 0，如果 s1 中字符串大于 s2 中字符串，则返回大于 0 的值，如果 s1 中字符串小于 s2 中字符串，则返回小于 0 的值。

char * strcat(char *dest,const char *src)：将 src 所指向的字符串(包括终止字符串的空字符)复制到 dest 所指向的数组的末尾，并且 src 指向的字符串的初始字符将覆盖 dest 末尾的

空字符，函数返回 dest 的值。

char *strchr(const char *src,int val)：在字符串 src 中搜索最后出现的 val，并返回它的位置，终止字符串的空字符也作为串的一部分．函数返回指 src 中第一个 val 的位置，如果没有出现与 val 相同的字符，则返回一个空指针。

int strcmp(const char *s1,const char *s2)：比较两个字符串，如果相同返回 0，如果 s1 中字符串大于 s2 中字符串，则返回大于 0 的值，如果 s1 中字符串小于 s2 中字符串，则返回小于 0 的值。

char * strcpy(char *dest,const char *src)：将 src 所指向的字符串(包括终止字符串的空字符)复制到 dest 所指向的数组的末尾，函数返回 dest 的值。

size_t strlcat(char *dest,const char *src,size_t size)：基本和 strcat 函数相同，但只复制前 size 个字符，如果 size 大于 src 所指向字符串的长度，则将 src 复制到 dest 后就返回。

size_t strlcpy(char *dest,const char *src,size_t size)：基本和 strcpy 函数相同，但只复制前 size 个字符，如果 size 大于 src 所指向字符串的长度，则将 src 复制到 dest 后就返回。

size_t strlen(const char *src)：返回字符串 src 的长度，不包括结束 NULL 字符

char * strlwr(char *string)：将 src 字符串中的大写字母转换成相应的小写字母，其他字符保持不变

int strncasecmp(const char *s1,const char *s2,size_t len)：基本和 strcasecmp 函数相同

char * strncat(char *dest,const char *src,size_t len)：函数 strncat 将 src 所指向的字符串(包括终止字符和空字符)复制到 dest 所指向的数组的末尾，最多追回 len 个字符，空字符及空字符后的字符不予复制。src 的初始字符覆盖 dest 末尾的空字符。结果字符串的末尾通常追加一个空字符，函数返回 dest 的值。

int strncmp(const char *s1,const char *s2,size_t len)：函数比较 s1 和 s2 所指向的字符串，如果相同返回 0，如果 s1 中字符串大于 s2 中字符串，则返回大于 0 的值，如果 s1 中字符串小于 s2 中字符串，则返回小于 0 的值。

char *strncpy(char *dest,const char *src,size_t len)：基本和 strlcpy 函数相同

size_t strnlen(const char *src,size_t len)：基本和 strlen 函数相同，如果 src 小于 len，则返回 src 的长度(不包含结束 NULL 字符)，否则返回 len

char * strrchr(const char *src,int val)：基本和 strchr 函数相同，但返回的是最后一次与 val 匹配字符的位置

char * strrev(char *string)：将字符串 string 翻转，即第一个字符变成最后一个，最后一个变成第一个。

char * strstr(const char *s1,const char *s2)：在字符串 s1 中搜索字符串 s2，并返回找到的第一个相匹配的指针，否则返回 NULL

char * strupr(char *string)：将 src 字符串中的小写字母转换成相应的大写字母，其他字符保持不变

15．定时器控制函数 timer.h

新版本要求将文件包含改为：#include "\avr\timer.h"

timer.h 文件定义定时/计数器使用的一个枚举类型：

```
enum
   {
   STOP=0,
   CK=1,
   CK8=2,
   CK64=3,
   CK256=4,
   CK1024=5,
   T0_FALLING_EDGE=6,
   T0_RISING_EDGE=7
   };
```

枚举类型定义定时/计数器的控制寄存器的时钟源(分频系数)
void timer0_source(unsigned int src)：将 src 赋值给寄存器 TCCR0
void timer0_stop();：通过对寄存器 TCNT0 清零来停止定时器 0
void timer0_start();：通过对寄存器 TCNT0 写入 1 来启动定时器 0
16．针对 ATmega163 的 I2C 函数 twi.h
新版本要求将文件包含改为：#include "\avr\twi.h"
17．看门狗定时器控制函数 wdt.h
新版本要求将文件包含改为：#include "\avr\wdt.h"
wdt_reset()：复位看门狗(喂狗)
wdt_enable(timeout)：打开看门狗，定时为 timeout
wdt_disable()：关闭看门狗

第 7 章　CodeVisionAVR 集成环境

编者在本章中介绍的是 CodeVisionAVR 的 Version1.0.1.8dLight 版(DEMO)，并将 CodeVisionAVR 与 ICCAVR 作对比介绍，最后再简单介绍 CodeVisionAVR 的库函数并给出一些简单的应用实例。

其实从下面的介绍中，可以看出 CodeVisionAVR 还是比较易用的，特别是对于用惯了 Keil 的读者，很快就能够上手。但是 CodeVisionAVR 在国内没有技术支持(以前有)，用户又少，有了问题不知该去哪里问，而且最大的问题是 CodeVisionAVR 的升级速度慢，使用过程中发现的缺陷不能及时改进，因此现在用户越来越少。

7.1　CodeVisionAVR 编译器简介

7.1.1　标识符

CodeVisionAVR(以下简称为 CVAVR)的标识符可以由字母、数字和下划线组成，但是必须用字母或下划线开头，这与 ICCAVR 的规定是一样的。应注意：CVAVR 标识符最长只能是 32 个字符。

7.1.2　关键字

以下为 CVAVR 定义的关键字：

break、bit、case、char、const、continue、default、do、double、eeprom、else、enum、extern、flash、float、goto、if、int、interrupt、long、register、short、sizeof、sfrb、sfrw、static、struct、switch、typedef、union、unsigned、void、while。

7.1.3　数据类型

表 7.1 为 CVAVR 的数据类型。

表 7.1 CVAVR 的数据类型

类 型	长度(位)	范 围
位变量(bit)	1	0 或 1
无符号字符型(unsigned char)	8	0~255
有符号字符型(signed char)	8	-128~127
字符型(char)	8	-128~127
无符号短整型(unsigned short)	16	0~65535
(有符号)短整型((signed) short)	16	-32768~32767
无符号整型(unsigned int)	16	0~65535
(有符号)整型((signed) int)	16	-32768~32767
无符号长整型(unsigned long)	32	0~4294967295
(有符号)长整型((signed) long)	32	-2147483648~2147483647
(单精度)浮点型(float)	32	+/-1.175e-38~3.40e+38
双精度(浮点型)(double)	32	+/-1.175e-38~3.40e+38

要注意：在 CVAVR 中，char 默认为 signed char，这与 ANSI C 的默认情况是相同的，但与 ICCAVR 默认为 unsigned char 是不同的。

7.1.4 常量

1. 二进制：在数字的前面加上“0b”的标志，如 0b10010011。
2. 16 进制：在数字的前面加上“0x”或“0X”的标志，如 0x4f。
3. 无符号整数：在一个数字后面加上“u”的后缀，如 23u。
4. 长整型整数：在一个数字后面加上“l”的后缀，如 23l。
5. 浮点型：在一个数字后面加上“f”的后缀，如 23f。
6. 字符常数：用单引号把字符括起来，如'a'。
7. 字符串数常数：用双引号把字符括起来，如"CVAVR"。

7.1.5 变量

1. 全局变量和局部变量

如果未对一个全局变量或静态局部变量赋初值，相当于对该变量赋初值为 0，因为 CVAVR 在启动时对全部数据内存清零，但要注意，在 ICCAVR 中，只对使用到的那一部分内存清零，对未使用的内存不清零，两者都是符合 ANSI C 要求的。CVAVR 中，若未对一个局部变量赋初值，则该变量的值是不确定的. 这与 ICCAVR 是一致的。

在 CVAVR 中，可以用如下方法定义一个变量并赋初值：

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

用“@”符号可以将全局变量保存在一个指定的 SRAM 空间，如需将全局变量 a 保存在 0x100 处，可定义如下：

```
int a @0x100;
```

也可以将一个结构体保存在从 0x110 开始的位置，定义如下：

```
struct a {
int b;
char c;
} alfa @0x110;
```

还可以指定将一个变量分配至寄存器中，定义如下：

```
register int a @14;
```

在上例中，分配全局变量 a(int 型)至寄存器 R14、R15 中。

在 ICCAVR 中，要在 0x110 处定义一个双字节的变量就只能使用在线汇编，很不方便，如下：

```
asm(".area memory(abs)\n"
".org 0x110\n"
"a:: .blkw 1");                    //注意，不要忘了加分号
```

2．位变量

在 CVAVR 中可以直接定义位变量，如下：

```
bit on;
```

也可以在定义时赋初值：

```
bit on=1;
```

说明：CVAVR 定义的位变量保存在 R2~R15 的寄存器中，最多只能定义 112 个位变量(14 个寄存器)，需要定义多少的位变量，应在 bit Variables Size(Project->configure->C Compiler)中指定，定义的位变量从寄存器 R2 的最低位开始放置。

7.1.6　运算符

CVAVR 支持以下的运算符，这些运算符与 ANSI C 的用法一致：

+、-、*、/、%、++、--、=、==、~、!、!=、<、>、<=、>=、&、&&、|、||、^、?、<<、>>、-=、+=、/=、%=、&=、*=、^=、|=、>>=、<<=。

7.1.7 存储空间

由于 AVR 为哈佛结构的单片机，其内部有程序存储器、数据存储器和 EEPROM 三个独立的空间，而 C 语言是针对冯•诺依曼结构的处理器开发的，并不适合于描述单片机这三种不同的存储空间，因此，AVR 的 C 编译器均对此作了相应的扩充，CVAVR 引入了 flash 和 eeprom 两个关键字。

1．EEPROM 空间：

在 CVAVR 中，可以用 eeprom 关键词将全局变量分配至 EEPROM 中，如下：

```
eeprom int a;
```

也可以在定义时对变量初始化，如：

```
eeprom int a=1;
```

CVAVR 还可以将数组、字符串、结构体分配至 EEPROM 中。如：

```
eeprom char a[10]={0,1,2,3};                    //数组
char eeprom *ptr_to_eeprom="This string is placed in EEPROM";      //字符串
eeprom struct a {                               //定义结构体
                char b;
                int c;
                char e[15];
                } f;
```

在 CVAVR 可以直接访问 EEPROM 中的全局变量，与访问 SRAM 中的数据方式相同，这比 ICCAVR、GCCAVR、IAR 方便得多了：

```
eeprom int a;
a=1;                                    //在程序中给 a 赋值
```

或

```
eeprom int *ptr_to_eeprom,a;
ptr_to_eeprom=&a;                       //通过指针给变量 a 赋值
*ptr_to_eeprom=1;
```

在 ICCAVR 中，不支持将全局变量或局部变量分配至 EEPROM，通常只将常量分配至

EEPROM 中，一般定义如下：

```
#pragma data:eeprom
const unsigned char table[]={0x3f,0x06,0x5b,0x4f,0x66,0x6d,0x7d,0x07,0x7f,
0x6f,0x77,0x7c,0x39,0x5e,0x79,0x71,;          //字形表存放在 EEPROM 中
#pragma data:data
```

2．Flash 空间：

CVAVR 对字符串的默认处理方式与 ICCAVR 相同，即对于用户没有指定只能保存在 Flash 中的字符串，在启动时将字符串从程序存储区复制到数据存储区。

```
char *ptr_to_ram="this string is placed in RAM";
```

上面例子中没有指定字符串只存储于 Flash，因此启动时会将该字符串复制到数据存储空间。

```
char flash *ptr_to_flash="this string is placed in Flash";
```

在上例中，用户使用了 flash 关键词，因此字符串只存储于 Flash 空间。

注意：在 CVAVR 中有两种存储模式“TINY”和“SMALL”，其中“TINY”模式使用 1 个字节(8 位)的指针，因此只能访问不大于 256 字节的存储空间；“SMALL”模式使用 2 个字节(16 位)的指针，可以访问不大于 64K 字节的存储空间。当使用外部数据存储器时，必须选用“SMALL”模式，如果没有使用外部数据存储器，应优先选用“TINY”模式。

CVAVR 指向字符串的指针数组至多只支持 8 个元素，注意，是指向字符串的指针数组，普通数组则没有这个限制。如：

```
char flash *strings[8]={"a","b","c","d","e","f","g","h"};
```

如果数组元素(字符串)超过 8 个，CVAVR 会出错。

3．数据存储：

数据存储器是用于保存变量、堆栈结构和动态内存分配的堆。通常它们不出现在输出文件中，但在程序运行时使用。一个没有使用外部扩展数据存储器的程序使用数据内存如图 7.1 所示。

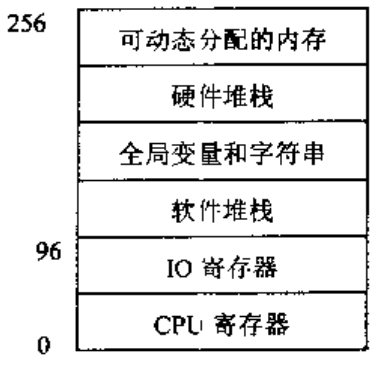


图 7.1　数据内存的使用

在图 7.1 中，从地址 0 开始的 96 个字节(0x60)是 CPU 寄存器和 I/O 寄存器，编译器从 0x60 往上放置软件堆栈、全局变量和字符串、硬件堆栈，在变量区域的顶部是可以分配动态的数据内存。

要注意，CVAVR 数据内存的使用方式与 ICCAVR 不同，ICCAVR 数据内存的使用方式如图 3.7 所示。

7.1.8 访问寄存器

CVAVR 使用“sfrb”和“sfrw”定义寄存器，如：

```
sfrb PINA=0x19;
sfrw TCNT1=0x2c;
```

则在 CVAVR 中，可以直接访问所需的寄存器：

```
unsigned char a;
a=PINA;
TCNT1=0x1111;
```

也可以直接访问寄存器中的某一位，比较方便，如：

```
PINA.7=0;
```

还可以用标准 C 的位变量实现：

```
PINA&=0b01111111;
```

在 ICCAVR 中，采用强制类型转换和指针的概念来实现访问 MCU 的寄存器，定义如下：

```
#define PINA (*(volatile unsigned char *)0x39)
```

则在 ICCAVR 中，可以使用如下方式访问：

```
PINA=0xff;
```

在 ICCAVR 中，要访问 I/O 寄存器的某一位，只能使用标准 C 的位变量实现：

```
PINA&=0b01111111;
```

7.1.9 中断服务函数

在 CVAVR 中使用扩展关键字 interrupt 说明某个函数为中断服务函数，如需要说明定时器 time1 溢出的函数为中断函数，可定义如下：

```
interrupt [TIM1_OVF] void timer1_overflow(void)
```

在 CVAVR 中，使用 TIM1_OVF 名称来代替中断向量号声明，比 ICCAVR 直接使用中断向量号更直观。

7.1.10　C 任务

在 CVAVR 中，可以使用“#pragma savereg”声明函数为 C 任务函数，让 CVAVR 生成该函数的代码中不必插入保存和恢复可变寄存器(Volatile Registers)的指令，让 RTOS 核来管理寄存器。CVAVR 中定义的可变寄存器为 R0、R1、R22、R23、R24、R25、R26、R27、R28、R29、R30、R31 和 SREG，与 ICCAVR 中定义的可变寄存器不相同。

7.2　CodeVisionAVR 菜单简介

1．CVAVR 集成工作环境(IDE)

CVAVR 的集成工作环境(IDE)与 ICCAVR 类似，如图 7.2 所示，其中①区为导航栏，②区为编辑区，③区为信息栏。

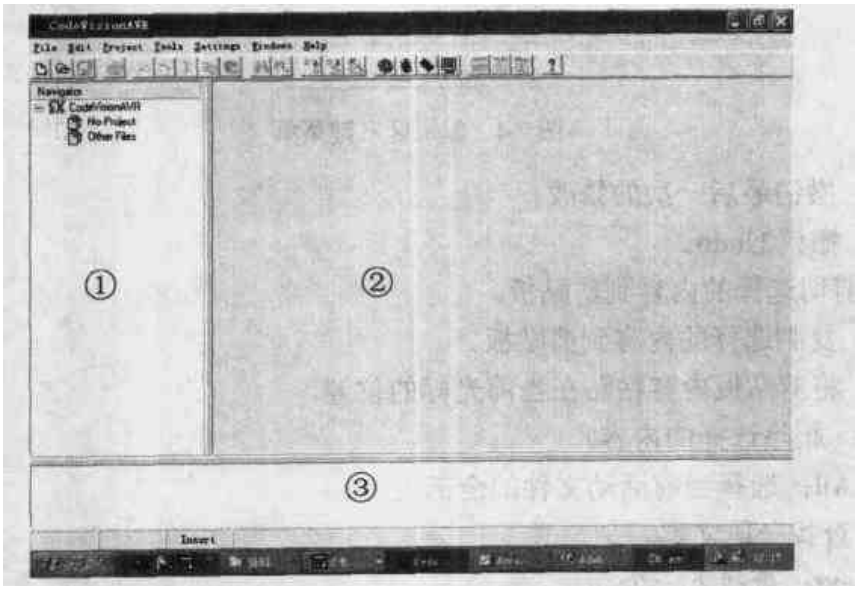


图 7.2　CVAVR 集成工作环境

2．右键菜单

在导航栏①中单击鼠标右键弹出的菜单，如图 7.3 所示，其中，

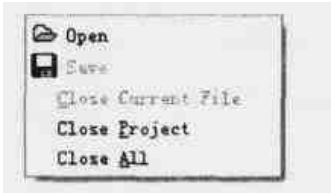


图 7.3 导航栏右键菜单

- Open：打开一个工程或源文件。
- Save：存盘。
- Close Current File：关闭当前文件。
- Close project：关闭工程文件。
- Close All：关闭全部文件。

在编辑区②单击鼠标右键弹出的菜单，如图 7.4 所示，其中，

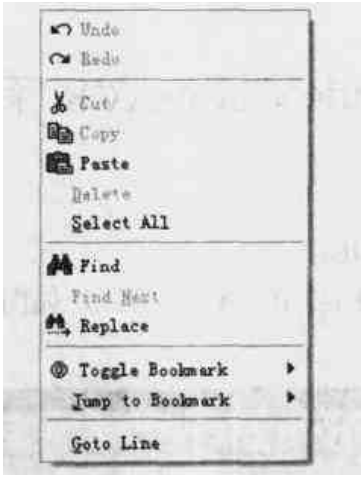


图 7.4 编辑区右键菜单

- Undo：撤销最后一次的修改。
- Redo：撤销 Undo。
- Cut：剪切选择的内容到剪贴板。
- Copy：复制选择的内容到剪贴板。
- Paste：将剪贴板内容粘贴在当前光标的位置。
- Delete：删除选择的内容。
- Selext All：选择当前活动文件的全部内容。
- Find：查找一段文本。
- Find Next：查找下一个。
- Replace：查找和替换。
- Toggle Bookmark：插入书签。
- Jump to Bookmark：跳到下一个书签。
- Gote Line：到指定的行。

3. File Menu 文件菜单

File Menu 文件菜单如图 7.5 所示，其中，

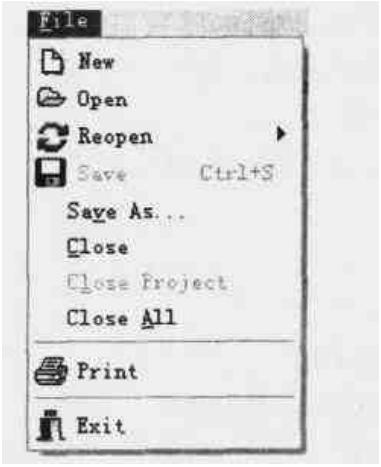


图 7.5　文件菜单

- New：新建一个源文件或工程文件。
- Open：打开一个源文件或工程文件。
- Reopen：重新打开历史文件，有关历史文件显示在子菜单中。
- Save：存盘。
- Save As...：换名存盘。
- Close：关闭打开的文件。
- Close Project：关闭一个工程文件。
- Close All：关闭所有打开的文件。
- Print：打印当前活动文件。
- Exit：退出 CVAVR。

4．Edit Menu 编辑菜单

File Menu 编辑菜单如图 7.6 所示，其中，

图 7.6　编辑菜单

- Undo：撤销最后一次的修改。
- Redo：撤销 Undo。

- Cut：剪切选择的内容到剪贴板。
- Copy：复制选择的内容到剪贴板。
- Paste：将剪贴板内容粘贴在当前光标的位置。
- Delete：删除选择的内容。
- Selext All：选择当前活动文件的全部内容。
- Find：查找一段文本。
- Find Next：查找下一个。
- Replace：查找和替换。
- Toggle Bookmark：插入书签。
- Jump to Bookmark：跳到下一个书签。
- Gote Line：到指定的行。

5. Project Menu 工程菜单

Project Menu 工程菜单如图 7.7 所示，其中，

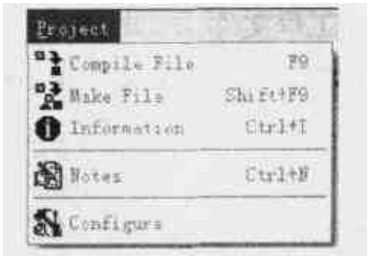


图 7.7 工程菜单

- Compile File：编译一个文件。
- Make file：编译一个工程。
- Information：工程信息。
- Notes：CVAVR 内置的记事本。
- Configure：配置工程文件，后面再详细介绍。

6. Tools Menu 工具菜单

Tools Menu 工具菜单如图 7.8 所示，其中，

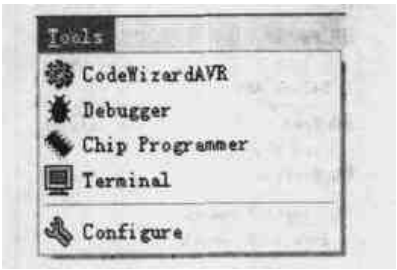


图 7.8 工具菜单

- CodewizardAVR：应用构筑向导程序，用于生成硬件的初始化代码。
- Debugger：调用仿真调试器，通常都是调用 AVRstudio。
- Chip Programmer：芯片编程。

- Terminal：内置仿真器。
- Configure：系统配置，可以添加其他工具(可执行文件)，添加后在 Tools 菜单中会增加相应的内容。

7. Setting Menu 设置菜单

Setting Menu 设置菜单如图 7.9 所示，其中，

General：一般设置内有三个选项，如图 7.10 所示。

图 7.9　设置菜单

图 7.10　一般设置对话框

- Shor Toolbar：显示工具栏。
- Show Navigator：显示导航栏。
- Show Information：显示编译信息。

Editor：CVAVR 集成环境设置。设置对话框如图 7.11 所示。

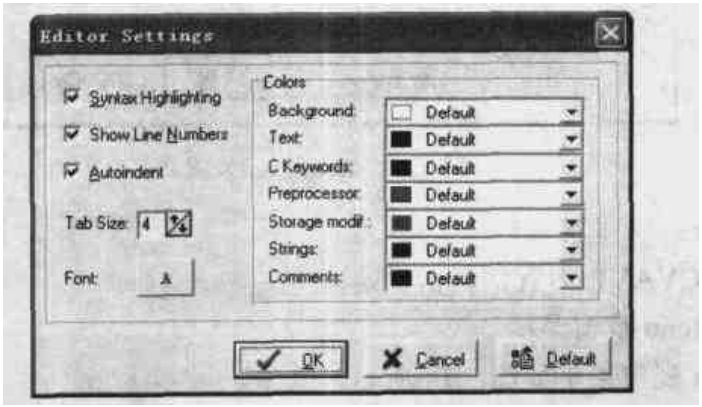


图 7.11　集成环境设置对话框

- SyntaxHighlighting：语法高亮显示。
- Show Line Numbers：显示行号。
- Autoindent：自动缩进。
- Tab Size：Tab 键跳跃设定。
- Font：字体设定。
- Background：背景颜色。
- Text：文本颜色。
- C Keywords：C 关键字颜色。
- Preprocessor：预处理字符颜色。

- storage modif：存储修饰符颜色。
- Strings：字符串颜色。
- Comments：注释颜色。
- Assembler：汇编出错设定，如图 7.12 所示。

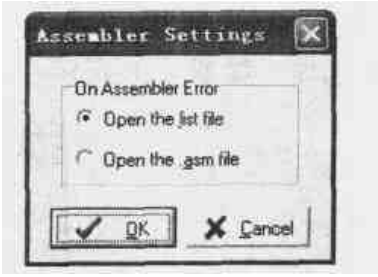


图 7.12 汇编出错设置

Debugger：调用其他仿真软件设置，通常都是调用 AVRstudio，其对话框如图 7.13 所示。单击“Browse”，通过对话框可以选择所需的文件，编者用的是 WinXP，在安装 AVRStudio4.04 时选用默认安装路径，因此“Debugger”中的对话框指向“D:\Program Files\Atmel\AVR Tools\AvrStudio4\AVRStudio.exe”，具体位置与用户的操作系统及安装 AVRStudio 的路径有关。

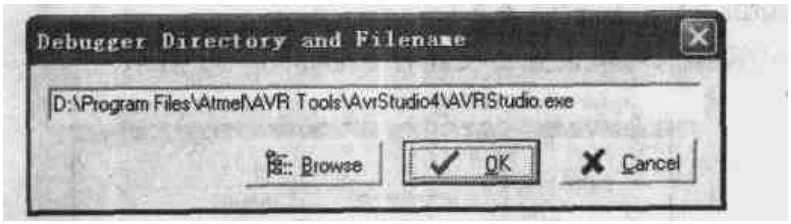


图 7.13 Debugger 设置

- Frogrammer：编程器设定。
- Terminal：CVAVR 内置仿真器设定。

8. Windows Menu 视窗菜单

Windows Menu 视窗菜单如图 7.14 所示，其中，

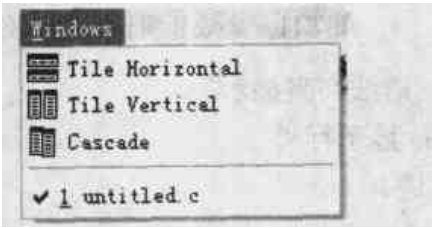


图 7.14 视窗菜单

- Tile Horizontal：水平分割。
- Tile Vertical：垂直分割。
- Cascade：层叠。

9. Help Menu 菜单

Help Menu 菜单如图 7.15 所示，其中，

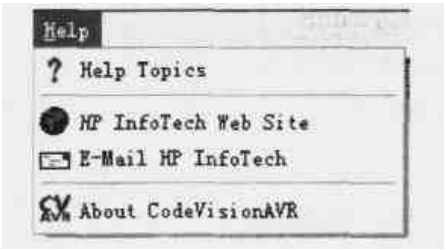


图 7.15　帮助菜单

- Help Topics：帮助主题。
- Hp InfoTech Web Site：进入技术支持主页。
- E-mail HP InfoTech：给技术支持发邮件。
- About CodeVisionAVR：关于 CVAVR。

10．快捷工具栏

快捷工具栏如图 7.16 所示。

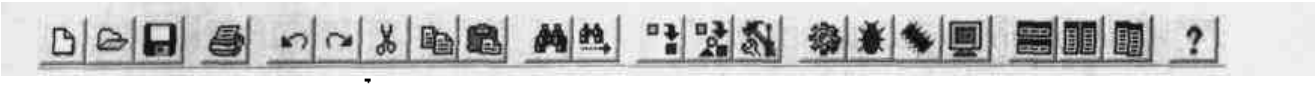


图 7.16　快捷工具栏

从左至右各图标功能分别如下：

新建一个源文件或工程文件，相当于“File->New”命令。

打开一个源文件或工程文件，相当于“File->Open”命令。

打印当前活动文件，相当于“File->Print”命令。

撤销最后一次的修改，相当于“Edit->Undo”命令。

撤销 Undo，相当于“Edit->Redo”命令。

剪切选择的内容到剪贴板，相当于“Edit->Cut”命令。

复制选择的内容到剪贴板，相当于“Edit->Copy”命令。

将剪贴板内容粘贴在当前光标的位置，相当于“Edit->Paste”命令。

删除选择的内容，相当于“Edit->Delete”命令。

查找一段文本，相当于“Edit->Find”命令。

查找下一个，相当于“Edit->Find Next”命令。

编译一个文件，相当于“Project->Compile File”命令。

编译一个工程，相当于“Project->Make file”命令。

配置工程文件，相当于“Project->Configure”命令。

应用构筑向导程序，相当于“Tools->CodewizardAVR”命令。

调用仿真调试器，相当于“Tools->Debugger”命令，。

芯片编程，相当于“Tools->Chip Programmer”命令。

内置仿真器，相当于“Tools->Terminal”命令。

水平分割，相当于“Windows->Tile Horizontal”命令。

垂直分割，相当于“Windows->Tile Vertical”命令。

层叠，相当于“Windows->Cascade”命令。

帮助主题，相当于“Helps->Help Topics”命令。

11．Configure 配置工程文件

共有以下三个页面：

“File”页面如图 7.17 所示，该页面可以向一个工程中添加或删除一个文件。单击“Add”，通过选择框选择文件添加进工程中，也可以单击工程中需要删除的文件，再单击“Remove”，可以从工程中除去该文件。

“C Compiler”页面如图 7.18 所示，该页面有以下功能：

- Chip：芯片型号选择。
- Clock：晶振频率。
- UART：UART 设置。
- Memory Model：编译模式。
- Ram：SRAM 设置。
- Complation：编译选项。
- Bit Variables Size：使用 Bit 位的数量设置。
- Automatic Register Allocation：自动寄存器分配。
- Use an External Startup Initialization File：使用外部的启动文件。

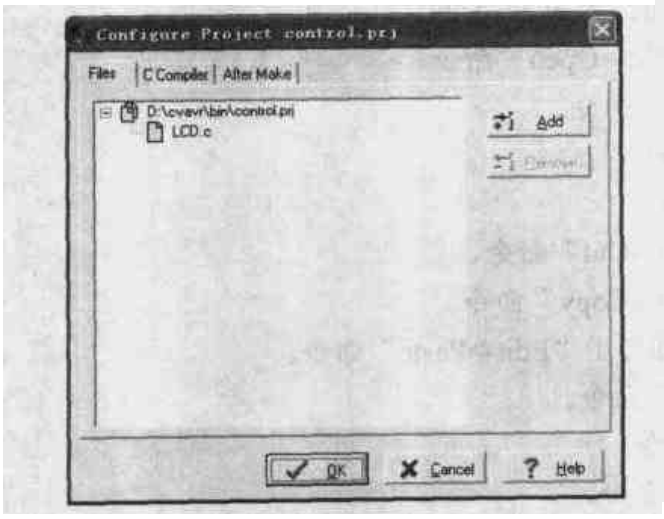


图 7.17 工程文件设置

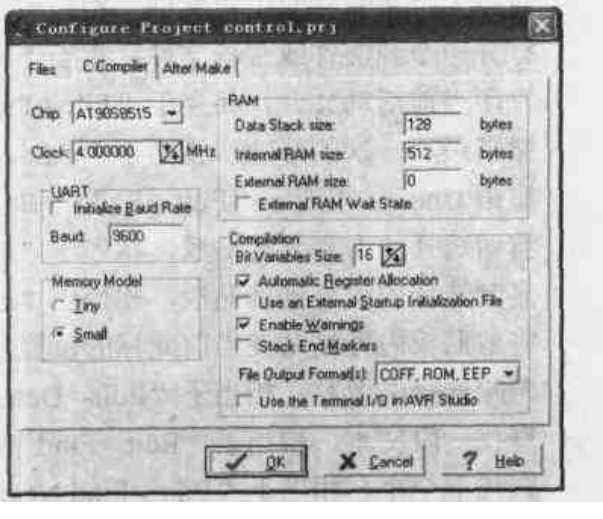


图 7.18 C 编译器设置选项

- Enable Warnings：允许告警。
- Stack End Markers：堆栈结束位置加上一个标志位(用于检测堆栈是否溢出)。
- File Output Format(s)：选择文件输出格式。
- Use the Teminal I/O in AVR Studio：如果选中，输出文件可以在 AVR Studio 的终端模拟仿真。
- “After Make”页面如图 7.19 所示，该页面主要设置在工程编译结束后 CVAVR 执行的动作。

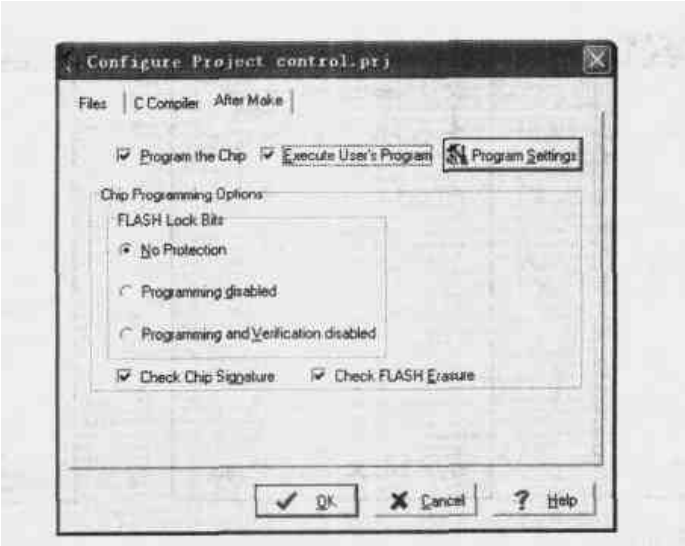


图 7.19　After Make 设置

12．应用构筑向导

该向导用于生成硬件的初始化代码。下面简单介绍各页面的功能。

1．芯片设置，可以设置所使用 AVR 芯片的型号及时钟频率等参数，如图 7.20 所示。

2．端口设置，可以设置各端口的初始化状态，如图 7.21 所示。

3．外部中断触发方式设定，如图 7.22 所示。

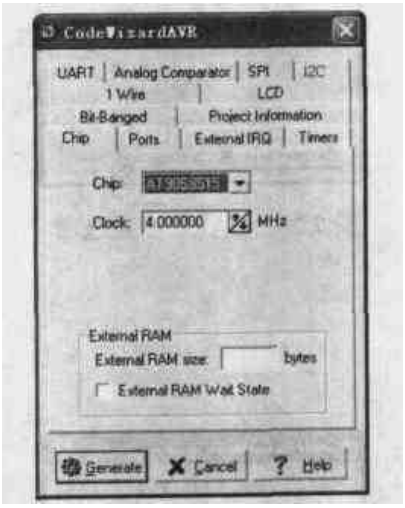


图 7.20　Chip 设置页面

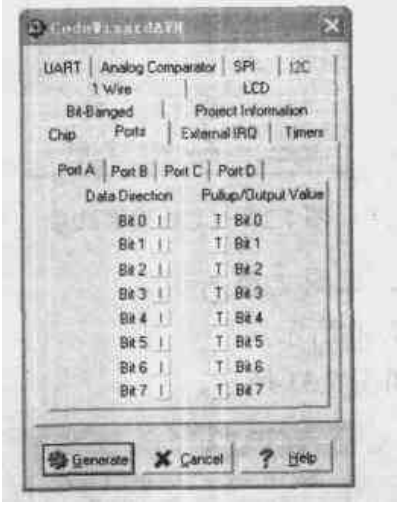


图 7.21　Ports 设置页面

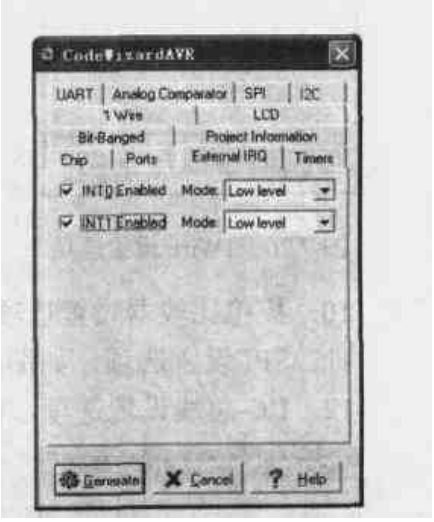


图 7.22　ExternalIPQ 设置页面

4．定时器及软件狗设置，如图 7.23 所示。

5．DS1302 设置，如图 7.24 所示。

6．工程信息页，可以填入与工程有关的信息，如图 7.25 所示。

7．单总线设置选项，如图 7.26 所示。

8．LCD 设置选项，如图 7.27 所示。

9．UART 设置选项，如图 7.28 所示。

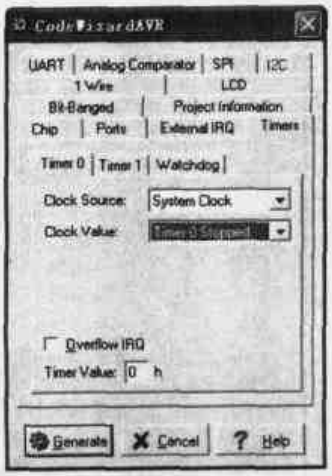


图 7.23 Time 设置页面

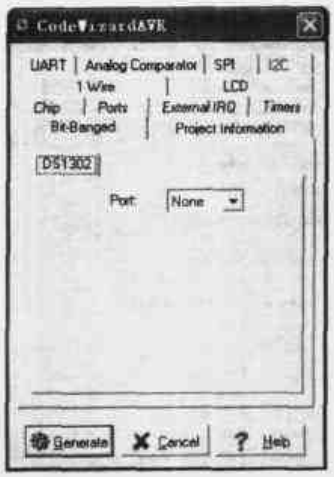


图 7.24 DS1302 设置页面

图 7.25 工程信息页面

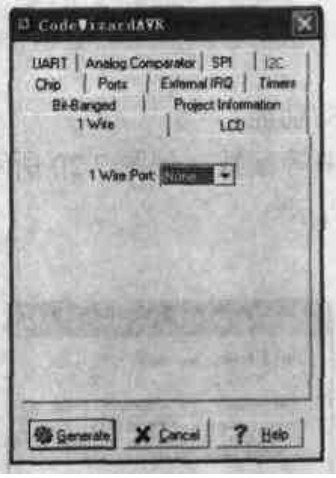


图 7.26 1Wire 设置选项

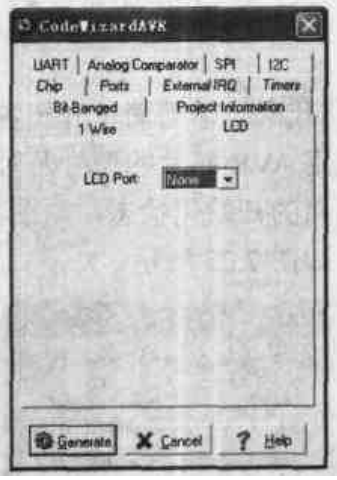


图 7.27 LCD 设置选项

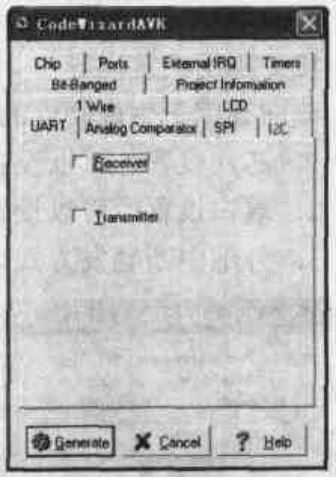


图 7.28 UART 设置选项

10. 模拟比较器设置选项，如图 7.29 所示。

11. SPI 设置选项，如图 7.30 所示。

12. I^2C 总线设置选项，如图 7.31 所示。

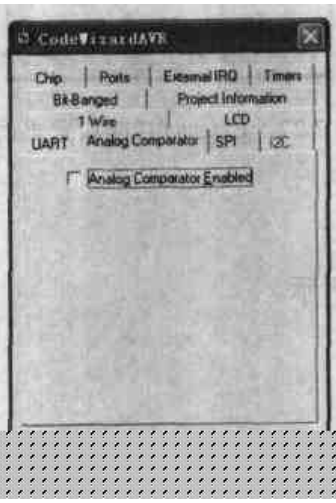


图 7.29 Comparator 设置选项

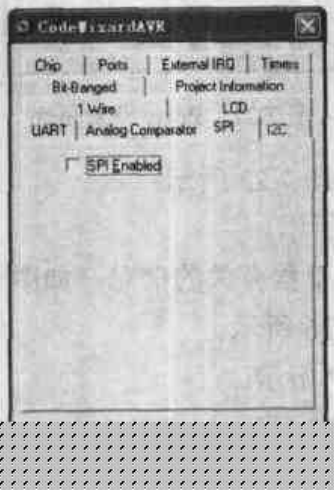


图 7.30 SPI 设置选项

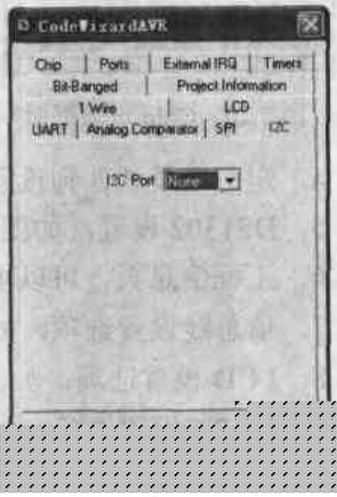


图 7.31 I^2C 设置选项

7.3　CodeVisionAVR 编译器常用库函数简介

7.3.1　字符类型函数

下列函数支持 ASCII 字符运算，使用这些函数之前应当用#include "ctype.h"预处理。

unsigned char isalnum(char c)：如果 c 是数字或字母则返回 1，否则返回 0。

unsigned char isalpha(char c)：如果 c 是字母则返回 1，否则返回 0。

unsigned char isascii(char c)：如果 c 是 ASCII 码(0~127)则返回 1，否则返回 0。

unsigned char iscntrl(char c)：如果 c 是控制字符(0~31 或 127)则返回 1，否则返回 0。

unsigned char isdigit(char c)：如果 c 是十进制数字则返回 1，否则返回 0。

unsigned char islower(char c)：如果 c 是小写字母则返回 1，否则返回 0。

unsigned char isprint(char c)：如果 c 是一个可打印字符(32~127)则返回 1，否则返回 0。

unsigned char ispunct(char c)：如果 c 是一个标点字符(ASCII 码中除了控制字符、数字和字母以外的字符)则返回 1，否则返回 0。

unsigned char isspace(char c)：如果 c 是空格、CR 和 HT 返回 1，否则返回 0。

unsigned char isupper(char c)：如果 c 是大写字母则返回 1，否则返回 0。

unsigned char isxdigit(char c)：如果 c 是 16 进制数字则返回 1，否则返回 0。

char toascii(char c)：返回字符 c 对应的 ASCII。

unsigned char toint(char c)：把 c 当做 16 进制字符并返回对应的 10 进制整数(范围：0~15)。

char tolower(char c)：如果 c 是大写字母则返回对应的小写字母，其他的字符保持不变。

char toupper(char c)：如果 c 是小写字母则返回对应的大写字母，其他的字符保持不变。

7.3.2　标准输入/输出函数

下列函数支持通过 UART 端口的输入/输出(I/O)，使用这些函数之前应当用#include "stdio.h"预处理，并对相应的端口初始化。

char getchar(void)：使用查询方式返回由 UART 接收的一个字符。

void putchar(char c)：使用查询方式由 UART 发送一个字符 c。

void puts(char *str)：使用 putchar 把 SRAM 中以空字符“\0”结束的字符串输出，并在输出结束后自动插入回车符。

void putsf(char flash *str)：使用 putchar 把 FLASH 中的以空字符“\0”结束的字符串输出，并在输出结束后自动插入换行符。

void printf(char flash *fmtstr [, arg1, arg2, ...])：使用 putchar 按格式说明符输出 fmtstr 字

符串。

fmtstr 指向的字符串是常量，必须放在 FLASH 中，本函数只支持如下 ANSI C 部分的格式化说明符：

%c：输出一个 ASCII 字符。

%d：输出有符号十进制整数。

%i：输出有符号十进制整数，与%d 相同。

%u：输出无符号十进制整数。

%x：输出以 0x 引导的十六进制整数。

%X：输出以 0X 引导的十六进制整数。

%s：输出 SRAM 中的以空字符结束的字符串。

%%：输出%字符。

说明：

所有输出的数据都是右对齐的，并在左侧加空格补齐。如果在%和 d、i、u、x 或 X 之间加入一个字符“0”，那么在输出数据的左侧加 0 补齐。如果在%和 d、i、u、x 或 X 之间加入宽度限制符(0~9),可以指定输出的数的最小宽度，如果在%和 d、i、u、x 或 X 之间加入一个字符“-”，那么输出的数左对齐(如果有宽度限定字符，应在限制符前加入字符“-”)。

void sprintf(char *str, char flash *fmtstr [, arg1, arg2, …])：这个函数与 printf 类似，只是它的格式化字符放在以空字符结尾的字符串 str 中。

char *gets(char *str, unsigned char len)：使用 getchar 接收以换行符结束的字符串 str。

说明：换行符会被"\0"替换。字符串的最大长度是 len，如果已经收到了 len 个字符后还没有收到换行符，那么自动加上一个结束标志 0，函数停止执行并退出。函数的返回值是指向 str 的指针。

char scanf(char flash *fmtstr [, arg1 address, arg2 address, …])：使用 getchar 按格式说明符接收格式化文本 fmtstr 字符串。

fmtstr 指向的字符串是常量，必须放在 FLASH 中，本函数只支持 ANSI C 部分的格式化说明符：

%c：输入一个 ASCII 字符。

%d：输入十进制整数。

%i：输入十进制整数，与%d 相同。

%u：输入无符号十进制整数。

%x：输入无符号十六进制整数。

%s：输入以空字符结束的字符串。

函数执行成功，返回接收的个数，如果接收出错，返回-1。

char sscanf(char *str, char flash *fmtstr [, arg1 address, arg2 address, …])：这个函数与 scanf 类似，只是 fmtstr 指向的字符串存放在 SRAM 中。

7.3.3　标准内存分配函数

使用这些函数之前应当用#include "stdlib.h"预处理。

int atoi(char *str)：转换字符串 str 为整型数并返回它的值，str 指向字符串的起始字符必须是数字字符或小数点，否则返回 0。当碰到字符串中第一个十进制数字和小数点以外的字符时，转换结束。

long int atol(char *str))：转换字符串 str 为长整型数并返回它的值，str 指向字符串的起始字符必须是数字字符或小数点，否则返回 0。当碰到字符串中第一个十进制数字和小数点以外的字符时，转换结束。

void itoa(int n, char *str)：转换整型数为字符型字符串。str 指向字符串的起始字符必须是数字字符或小数点，否则返回 0。当碰到字符串中第一个十进制数字和小数点以外的字符时，转换结束。

void ltoa(long int n, char *str)：转换长整型数为字符型字符串，这个函数的原型和源代码在 ltoa.h 文件中。

void ftoa(float n, unsigned char decimals, char *str)：转换浮点数为字符型字符串，由 decimals 指定字符串的长度，这个函数的原型和源代码在 ftoa.h 文件中。

void ftoe(float n, unsigned char decimals, char *str)：转换浮点数为字符型字符串，由 decimals 指定字符串的长度，字符串采用以 e 为底的方法，这个函数的原型和源代码在 ftoe.h 文件中。

float atof(char *str)：转换字符串为浮点数，字符串 str 起始字符必须是数字字符或小数点，否则返回 0。当碰到字符串中第一个十进制数字和小数点以外的字符时，转换结束。这个函数的原型和源代码在 atof.h 文件中。

int rand (void)：产生一个 0~32767 之间的随机数。

void srand(int seed)：设置随机数发生器的种子数。

7.3.4　数学函数

使用这些函数之前应当用#include "math.h"预处理。

unsigned char cabs(char n)：返回 n 的绝对值。

unsigned int abs(int n)：返回 n 的绝对值。

unsigned long labs(long int n)：返回 n 的绝对值。

float fabs(float n)：返回 n 的绝对值。

signed char cman(char a,char b)：返回 a 和 b 的最大值。

int man(int a, int b)：返回 a 和 b 的最大值。

long int lman(long int a, long int b)：返回 a 和 b 的最大值。

float fman(float a,float b)：返回 a 和 b 的最大值。

char cmin(char a,char b)：返回 a 和 b 的最小值。

int min(int a, int b)：返回 a 和 b 的最小值。

long int lmin(long int a, long int b)：返回 a 和 b 的最小值。

float fmin(float a, float b)：返回 a 和 b 的最小值。

char csign(char n)：当 n 分别为负数、0、正数时，返回 - 1、0、1。

char sign(int n)：当 n 分别为负数、0、正数时，返回 - 1、0、1。

char lsign(long int n)：当 n 分别为负数、0、正数时，返回 - 1、0、1。

char fsign(float n)：当 n 分别为负数、0、正数时，返回 - 1、0、1。

unsigned char sqrt(unsigned int n)：返回无符号整数 n 的平方根。

unsigned int lsqrt(unsigned long n)：返回无符号长整数 n 的平方根。

float sqrt(float n)：返回浮点数 n 的平方根。

float floor(float n)：返回不大于 n 的最大整数。

float ceil(float n)：返回对应 n 的整数，小数部分四舍五入。

float fmod(float n, float y)：返回 n/y 的余数。

float modf(float n, float *ipart)：把浮点数 n 分解成整数部分和小数部分。整数部分存放在 ipart 指向的变量中，小数部分应大于或等于 0.5 而小于 1，并作为函数的返回值。

float ldenp(float n, int enpn)：返回 $n\times2^{enpn}$ 。

float frenp(float n, int *enpn)：把浮点数 n 分解成数字部分 y(尾数)和以 2 为底的指数 n 两个部分，即 $n=y\times2^n$，y 要大于等于 0.5 小于 1，y 值被函数返回，而 n 值存放在 enpn 指向的变量中。

float enp(float n)：返回 e^n 的值。

float log(float n)：返回 n 的自然对数。

float log10(float n)：返回以 10 为底的 n 的对数。

float pow(float n, float y)：返回 n^y 的值。

float sin(float n)：返回 n 的正弦函数值，n 为弧度。

float cos(float n)：返回 n 的余弦函数值，n 为弧度。

float tan(float n)：返回 n 的正切函数值，n 为弧度。

float sinh(float n)：返回 n 的双曲正弦函数值，n 为弧度。

float cosh(float n)：返回 n 的双曲余弦函数值，n 为弧度。

float tanh(float n)：返回 n 的双曲正切函数值，n 为弧度。

float asin(float n)：返回 n 的反正弦函数值，返回值为弧度，范围在 $-\pi/2\sim\pi/2$ 之间，n 的值必须在-1~1 之间。

float acos(float n)：返回 n 的反余弦函数值，返回值为弧度，范围在 $0\sim\pi$ 之间，n 的值必须在-1~1 之间。

float atan(float n)：返回 n 的反正弦函数值，返回值为弧度，范围在 $-\pi/2\sim\pi/2$ 之间。

float atan2(float y, float n)：返回 y / n 的反正弦函数值，返回值为弧度，范围在-π ~π之间。

7.3.5　字符串函数

使用这些函数之前应当用#include "string.h"预处理。

char *strcat(char *str1, char *str2)：复制字符串 str2 到字符串 str1 的结尾，返回 str1 的指针。

char *strcatf(char *str1, char flash *str2)：复制 FLASH 中 str2 到 str1 的结尾，返回 str1 的指针。

char *strncat(char *str1, char *str2, unsigned char n)：复制 str2(不含结束符 NULL)的 n 个字符到 str1 的结尾，如果 str2 的长度比 n 小，则只复制 str2，返回 str1 的指针。

char *strncatf(char *str1, char flash *str2, unsigned char n)：复制 FLASH 中的字符串 str2(不含结束符 NULL)的 n 个字符到 str1 的结尾，如果 str2 的长度比 n 小，则只复制 str2，返回 str1 的指针。

char *strchr(char *str, char c)：在字符串 str 中搜索第一个出现的字符 c。如果成功，返回匹配字符的指针；否则返回 NULL。

char *strrchr(char *str, char c)：在字符串 str 中搜索最后一个出现的字符 c。如果成功，返回匹配字符的指针；否则返回 NULL。

signed char strpos(char *str, char c)：在字符串 str 中搜索第一个出现的字符 c。如果成功，返回匹配字符在字符串中的位置；否则返回 - 1。

signed char strrpos(char *str, char c)：在字符串 str 中搜索最后一个出现的 c。如果成功，返回匹配字符在字符串中的位置；否则返回 - 1。

signed char strcmp(char *str1, char *str2)：比较两个字符串。如果相同，返回 0；如果 str1>str2，返回值>0；如果 str1<str2，返回值<0。

signed char strcmpf(char *str1, char flash *str2)：比较 SRAM 中的字符串 str1 和 Flash 中的字符串 str2。如果相同，返回 0；如果 str1>str2，返回值>0；如果 str1<str2，返回值<0。

signed char strncmp(char *str1, char *str2, unsigned char n)：比较两个字符串的前 n 个字符。如果相同，返回 0；如果 str1>str2，返回值>0；如果 str1<str2，返回值<0。

signed char strncmpf(char *str1, char flash *str2, unsigned char n)：比较 SRAM 中的字符串 str1 和 Flash 中的字符串 str2 的前 n 个字符。如果相同，返回 0；如果 str1>str2，返回值>0；如果 str1<str2，返回值<0。

char *strcpy(char *dest, char *src)：复制字符串 src 到字符串 dest，返回 dest 的指针。

char *strcpyf(char *dest, char flash *src)：复制 Flash 中的字符串 src 到 SRAM 中的字符串 dest，返回 dest 的指针。

char *strncpy(char *dest, char *src, unsigned char n)：复制字符串 src 的前 n 个字符到字

符串 dest，返回 dest 的指针。

char *strncpyf(char *dest, char flash *src, unsigned char n)：复制 Flash 中的字符串 src 的前 n 个字符到 SRAM 中的字符串 dest，返回 dest 的指针。

unsigned char strspn(char *str, char *set)：在字符串 str 中搜索与字符串 set 不匹配的第一个字符。如果搜索到不匹配字符，返回不匹配字符在 str 的位置；如果 set 的所有字符都匹配，返回字符串 str 的长度。

unsigned char strspnf(char *str, char flash *set)：在 SRAM 中的字符串 str 中搜索与 Flash 中的字符串 set 不匹配的第一个字符。如果搜索到不匹配字符，返回不匹配字符在 str 的位置；如果 set 的所有字符都匹配，返回字符串 str 的长度。

unsigned char strcspn(char *str, char *set)：在字符串 str 中搜索与字符串 set 匹配的第一个字符。如果搜索到匹配字符，返回匹配字符在 str 的位置；否则返回字符串 str 的长度。

unsigned char strcspnf(char *str, char flash *set)：在字符串 str 中搜索与 Flash 中的字符串 set 匹配的第一个字符。如果搜索到匹配字符，返回匹配字符在 str 的位置；否则返回字符串 str 的长度。

char *strpbrk(char *str, char *set)：在字符串 str 中搜索与字符串 set 匹配的第一个字符。如果搜索到匹配字符，返回匹配字符的指针；否则返回 NULL。

char *strpbrkf(char *str, char flash *set)：在字符串 str 中搜索与 FLASH 中的字符串 set 匹配的第一个字符。如果搜索到匹配字符，返回匹配字符的指针；否则返回 NULL。

char *strrpbrk(char *str, char *set)：在字符串 str 中搜索与字符串 set 匹配的最后一个字符。如果搜索到匹配字符，返回匹配字符的指针；如果没有匹配字符，返回 NULL。

char *strrpbrkf(char *str, char flash *set)：在 SRAM 中的字符串 str 中搜索与 Flash 中的字符串 set 匹配的最后一个字符。如果搜索到匹配字符，返回匹配字符的指针；否则返回 NULL。

char *strstr(char *str1, char *str2)：在字符串 str1 中搜索与字符串 str2 匹配的子字符串。如果找到匹配的子字符串，返回 str1 中的子字符串的起始地址指针；否则返回 NULL。

char *strstrf(char *str1, char flash *str2)：在 SRAM 中的字符串 str1 中搜索与 Flash 中的字符串 str2 匹配的子字符串。如果找到匹配的子字符串，返回 str1 中的子字符串的起始地址指针；否则返回 NULL。

unsigned char strlen(char *str)：返回字符串 str 的长度(范围 0~255)。这个函数只能用在 TINY 模式下。

unsigned int _strlen(char *str)：返回字符串 str 的长度(范围 0~65535)。这个函数只能用在 SMALL 模式下。

unsigned int strlenf(char flash *str)：返回 Flash 中的字符串 str 的长度。

void *memcpy(void *dest,void *src, unsigned char n)：复制 src 的 n 个字节到 dest。dest 与 src 不能重叠。返回 dest 的指针。这个函数只能用在 TINY 模式下。

void *memcpy(void *dest,void *src, unsigned int n)：复制 src 的 n 个字节到 dest。dest 与 src 不能重叠。返回 dest 的指针。这个函数只能用在 SMALL 模式下。

void *memcpyf(void *dest,void flash *src, unsigned char n)：复制 Flash 中的字符串 src 的 n 个字节到 dest。dest 与 src 不能重叠。返回 dest 的指针。这个函数只能用在 TINY 模式。

void *memcpyf(void *dest,void flash *src, unsigned int n)：复制 Flash 中的字符串 src 的 n 个字节到 dest。dest 与 src 不能重叠。返回 dest 的指针。这个函数只能用在 SMALL 模式。

void *memccpy(void *dest,void *src, char c, unsigned char n)：复制字符串 src 的 n 个字节到 dest，如果碰到字符 c 就停止。Dest 与 src 不能重叠。如果最后一个复制的字符是 c 则返回 NULL，否则返回指向 dest+n+1 的指针。这个函数只能用在 TINY 模式。

void *memccpy(void *dest,void *src, char c, unsigned int n)：复制字符串 src 的 n 个字节到 dest，如果碰到字符 c 就停止。dest 与 src 不能重叠。如果最后一个复制的字符是 c 返回 NULL，否则返回指向 dest+n+1 的指针。这个函数只能用在 SMALL 模式。

void *memmove(void *dest,void *src, unsigned char n)：复制 src 的 n 个字节到 dest，dest 与 src 可以重叠，返回 dest 的指针。这个函数只能用在 TINY 模式。

void *memmove(void *dest,void *src, unsigned int n)：复制 src 的 n 个字节到 dest，dest 与 src 可以重叠，返回 dest 的指针。这个函数只能用在 SMALL 模式。

void *memchr(void *buf, unsigned char c, unsigned char n)：在 buf 的前 n 个字节中搜索字符 c，如果搜索到 c，就返回指向 c 的指针；否则返回 NULL。这个函数只能用在 TINY 模式。

void *memchr(void *buf, unsigned char c, unsigned int n)：在 buf 的前 n 个字节中搜索字符 c，如果搜索到 c，就返回指向 c 的指针；否则返回 NULL。这个函数只能用在 SMALL 模式。

signed char memcmp(void *buf1,void *buf2, unsigned char n)：比较字符串 buf1 和 buf2 的前 n 个字节。当 buf1<buf2，buf1=buf2，buf1>buf2 时分别返回<0，0，>0。这个函数只能用在 TINY 模式。

signed char memcmp(void *buf1,void *buf2, unsigned int n)：比较字符串 buf1 和 buf2 的前 n 个字节。当 buf1<buf2，buf1=buf2，buf1>buf2 时分别返回<0，0，>0。这个函数只能用在 SMALL 模式。

signed char memcmpf(void *buf1,void flash *buf2, unsigned char n)：比较 SRAM 中的字符串 buf1 和 FLASH 中的字符串 buf2，最多比较前 n 个字节。当 buf1<buf2，buf1=buf2，buf1>buf2 时分别返回<0，0，>0。这个函数只能用在 TINY 模式。

signed char memcmpf(void *buf1,void flash *buf2, unsigned int n)：比较 SRAM 中的字符串 buf1 和 FLASH 中的字符串 buf2，最多比较前 n 个字节。当 buf1<buf2，buf1=buf2，buf1>buf2 时分别返回<0，0，>0。这个函数只能用在 SMALL 模式。

void *memset(void *buf, unsigned char c, unsigned char n)：用字符 c 填充 buf 的前 n 个字节，返回指向 buf 的指针。这个函数只能用在 TINY 模式。

void *memset(void *buf, unsigned char c, unsigned int n)：用字符 c 填充 buf 的前 n 个字节，返回指向 buf 的指针。这个函数只能用在 SMALL 模式。

7.3.6 BCD 转换函数

使用这些函数之前应当用#include " bcd.h "预处理。

unsigned char bcd2bin(unsigned char n)：把 BCD 码数 n 转换为二进制。

unsigned char bin2bcd(unsigned char n)：把二进制数 n 转换为 BCD 码。n 必须为在 0~99 之间。

7.3.7 存储器访问函数

使用这些函数之前应当用#include " mem.h"预处理。

void pokeb(unsigned int addr, unsigned char data)：把一个字节 data 写在 SRAM 中指定的 addr 地址。

void pokew(unsigned int addr, unsigned int data)：把一个字 data 写在 SRAM 中指定的 addr 地址。低字节在 addr，高字节在 addr+1。

unsignad char peekb(unsigned int addr)：在 SRAM 中指定的 addr 地址读一个字节。

unsigned int peekw (unsigned int addr)：在 SRAM 中指定的 addr 地址读一个字节。低字节从 addr 读出，高字节从 addr+1 读出。

7.3.8 延时函数

使用这些函数之前应当用#include "delay.h"预处理。下面函数使用程序循环产生延时，调用这些函数之前要关闭中断，否则会比预期的延时要长，并且要在 Project->Configure->C Compiler 菜单中设定正确的时钟频率。

void delay_us(unsigned int n)：n 个微秒的延时。n 必须是常数表达式。

void delay_ms(unsigned int n)：n 个毫秒的延时。n 必须是常数表达式。这个函数会每毫秒清一次看门狗。

7.3.9 LCD 函数

1．LCD 字符型函数

使用这些函数之前应当用#include "lcd.h "预处理，在包含头文件之前，用户必须声明哪一个口与 LCD 模块通信。下面的函数针对由日立 HD44780 或兼容芯片控制的字符型 LCD 模块，支持以下类型：1×8、2×12、3×12、1×16、2×16、2×20、4×20、2×24、2×40。

底层 LCD 函数：

void _lcd_ready(void)：等待，直到 LCD 模块准备好接收数据。在使用_lcd_write_data 函数向 LCD 模块写数据前必须调用此函数。

void _lcd_write_data(unsigned char data)：向 LCD 模块的命令寄存器写一个字节 data。

void lcd_write_byte(unsigned char addr, unsigned char data)：向 LCD 模块的字符发生器或 RAM 中写一个字节 data。

unsigned char lcd_read_byte(unsigned char addr)：从 LCD 模块的字符发生器或 RAM 读出一个字节。

高级 LCD 函数：

void lcd_init(unsigned char lcd_columns)：初始化 LCD 模块，清屏并把显示坐标设定在 0 列 0 行，模块不显示光标，LCD 模块的列必须指定(例如 16)，在使用其他高级 LCD 函数前必须先调用此函数。

void lcd_clear(void)：清屏并把显示坐标设定在 0 列 0 行。

void lcd_gotoxy(unsigned char x, unsigned char y)：设定显示坐标在 x 列 y 行。列、行均由 0 开始。

void lcd_putchar(char c)：在当前坐标显示字符 c。

void lcd_puts(char *str)：在当前坐标显示 SRAM 中的字符串 str。

void lcd_putsf(char flash *str)：在当前坐标显示 FLASH 中的字符串 str。

2．4×40 字符型函数

只有商业版的 CodeVisionAVR C Compiler 才有这部分功能，使用这些函数之前应当用#include "lcd4x40.h"预处理，包含头文件之前用户必须声明哪一个口与 LCD 模块通信。下面的函数针对由日立 HD44780 或兼容芯片控制的字符型 40×4LCD 模块。

底层 LCD 函数：

void _lcd_ready(void)：等待，直到 LCD 模块准备好接收数据，在使用_lcd_write_data 函数向 LCD 模块写数据前必须调用此函数。

void _lcd_write_data(unsigned char data)：向 LCD 模块的命令寄存器写一个字节 data，这个函数可以用来修改 LCD 的配置。在调用_lcd_ready 和_lcd_write_data 函数之前，全局变量_en1_msk 必须设为 LCD_EN1(LCD_EN2)，以选择使用上半部(下半部)的 LCD 控制器。

void lcd_write_byte(unsigned char addr, unsigned char data)：向 LCD 模块的字符发生器或 RAM 写一个字节 data。

unsigned char lcd_read_byte(unsigned char addr)：从 LCD 模块的字符发生器或 RAM 读出一个字节。

高级 LCD 函数：

void lcd_init(void)：初始化 LCD 模块，清屏并把显示坐标设定在 0 列 0 行，不显示光标。LCD 模块的列必须指定(例如 16)，在使用其他高级 LCD 函数前，必须先调用此函数。

void lcd_clear(void)：清屏并把显示坐标设定在 0 列 0 行。

void lcd_gotoxy(unsigned char x, unsigned char y)：设定显示坐标在 x 列 y 行。列、行均由 0 开始。

void lcd_putchar(char c)：在当前坐标显示字符 c。

void lcd_puts(char *str)：在当前坐标显示 SRAM 中的字符串 str。

void lcd_putsf(char flash *str)：在当前坐标显示 FLASH 中的字符串 str。

3．以 8 位外部存储器模式接口的 LCD 显示函数

使用这些函数之前应当用#include " lcdstk.h "预处理。下面的函数针对由日立 HD44780 或兼容芯片控制的字符型 LCD 模块，支持以下类型：1×8、2×12、3×12、1×16、2×16、2×20、4×20、2×24、2×40。该模块作为一个 8 位的外设接在 AVR 的外部数据地址总线上，因此这些函数只能用于能外扩存储器的 AVR 芯片，如 AT90S4414、AT90S8515、ATmega603、ATmega103、ATmega161 和 ATmega128。这种接口方式也用在 STK200 和 STK300 开发板上，LCD 的连接方式参见开发板的说明。

底层 LCD 函数：

void _lcd_ready(void)：等待，直到 LCD 模块准备好接收数据。在使用宏_LCD_RS0 和_LCD_RS1_向 LCD 模块写数据前必须调用此函数。

void lcd_write_byte(unsigned char addr, unsigned char data)：向 LCD 的字符发生器或 RAM 写一个字节。

unsigned char lcd_read_byte(unsigned char addr)：从 LCD 模块的字符发生器或 RAM 中读出一个字节。

高级 LCD 函数：

void lcd_init(unsigned char lcd_columns)：初始化 LCD 模块，清屏并把显示坐标设定在 0 列 0 行，不显示光标，LCD 模块的列必须指定(例如 16)。在使用其他高级 LCD 函数前必须先调用此函数。

void lcd_clear(void)：清屏并把显示坐标设定在 0 列 0 行。

void lcd_gotoxy(unsigned char x, unsigned char y)：设定显示坐标在 x 列 y 行。列、行由 0 开始。

void lcd_putchar(char c)：在当前坐标显示字符 c。

void lcd_puts(char *str)：在当前坐标显示 SRAM 中的字符串 str。

void lcd_putsf(char flash *str)：在当前坐标显示 FLASH 中的字符串 str。

7.3.10 I^2C 总线函数

1．I^2C 总线通信协议函数

使用这些函数之前应当用#include "i2c.h"预处理，包含头文件之前用户必须先声明哪些口线用于 I^2C 总线。利用以下这些函数可以把单片机作为主机或从机。

void i2c_init(void)：初始化 I^2C 总线。调用其他 I^2C 函数之前必须先调用此函数。

unsigned char i2c_start(void)：发送 START 信号。如果总线空闲，返回 1；如果总线忙，返回 0。

void i2c_stop(void)：发送 STOP 信号。

unsigned char i2c_read(unsigned char ack)：读一个字节。

unsigned char i2c_write(unsigned char data)：写一个字节。如果从机有应答，返回 1；如果从机不应答，返回 0。

2．LM75 温度传感器函数

使用这些函数之前应当用#include "lm75.h "预处理。在 lm75.h 中自动包含 I^2C 总线函数，在包含头文件之前，必须先声明哪些口线通过 I^2C 总线与 LM75 通信。

void lm75_init(unsigned char chip,signed char thyst,signed char tos, unsigned char pol)：初始化 LM75，调用这个函数之前，必须调用函数 i2c_init 初始化 I^2C 总线，调用其他的 LM75 函数之前，必须先调用此函数，如果有多个 LM75 接在同一个 I^2C 总线上，则每个 LM75 都要初始化(一个 I^2C 总线上最多只能接 8 只 LM75，因为地址从 0~7)。

LM75 被设置为比较模式，O.S.输出脚在温度高于 tos 时激活，低于 thyst 时恢复高阻。thyst 和 tos 都是摄氏度。pol 设置 LM75 的 O.S.输出脚在激活时的极性，如果 pol 是 0，输出激活时为低；如果 pol 是 1，输出激活时为高。

int lm75_temperature_10(unsigned char chip)：读地址为 chip 的 LM75 的温度。温度是摄氏度，其值为返回值除以 10。

3．DS1621 温度计函数

使用这些函数之前应当用#include "ds1621.h"预处理。ds1621.h 中自动包含 I^2C 总线函数，包含头文件之前，必须先声明哪些口线通过 I^2C 总线与 DS1621 通信。

调用这个函数之前，必须调用函数 i2c_init 初始化 I^2C 总线，调用其他的 DS1621 函数之前，必须先调用此函数，如果有多个 DS1621 接在同一个 I^2C 总线上，则每个 DS1621 都要作初始化，同一个 I^2C 总线上最多只能接 8 只 DS1621。

DS1621 被设置于比较模式，Tout 输出脚在温度高于 thigh 时激活，低于 tlow 时恢复高阻，tlow 和 thigh 都是摄氏度。pol 设置 DS1621 的 Tout 输出脚在激活时的极性，如果 pol 是 0，输出激活时为低；如果 pol 是 1，输出激活时为高。

unsigned char ds1621_get_status(unsigned char chip)：读出对应地址 chip 的 DS1621 的配置/状态字节，其内容参看 DS1621 的数据手册。

void ds1621_set_status(unsigned char chip, unsigned char data)：设置对应地址 chip 的 DS1621 的配置/状态字节，其内容参看 DS1621 的数据手册。

void ds1621_start(unsigned char chip)：使地址为 chip 的 ds1621 从省电模式中退出，开始温度测量和比较。

void ds1621_stop(unsigned char chip)：使地址为 chip 的 DS1621 停止测量温度，进入省电模式。

int ds1621_temperature_10(unsigned char chip)：读地址为 chip 的 DS1621 的温度。温度是摄氏度，其值为返回值除以 10。

4．PCF8563 实时时钟函数

使用这些函数之前应当用#include "pcf8563.h"预处理。只有商业版的 CodeVisionAVR C

Compiler 才有这部分功能。在 pcf8563.h 中自动包含 I^2C 总线函数，包含头文件之前，必须先声明哪些口线通过 I^2C 总线与 PCF8563 通信。

void rtc_init(unsigned char ctrl2, unsigned char clkout, unsigned char timer_ctrl)：初始化 PCF8563。调用这个函数之前，必须调用函数 i2c_init 初始化 I^2C 总线，调用其他的 PCF8563 函数之前，必须先调用此函数，一个 I^2C 总线只能接一个 PCF8583。

参数 ctrl2 设定 PCF8563 的 Control/Status2(控制/状态 2)寄存器的初值。在 pcf8563.h 头文件定义了以下一些宏，以便设置 ctrl2 参数：

RTC_TIE_ON：置 Control/Status 2 寄存器的 TIE 位为 1。

RTC_AIE_ON：置 Control/Status 2 寄存器的 AIE 位为 1。

RTC_TP_ON：置 Control/Status 2 寄存器的 TI/TP 位为 1。

这些宏可以用“|”操作符连在一起使用，以便能同时设置多个位为 1。

参数 clkout 设定 PCF8563 CLKOUT Frequency(输出频率)寄存器的初值。在 pcf8563.h 头文件定义了以下一些宏，以便设置 clkout 参数：

RTC_CLKOUT_OFF：关闭 PCF8563 的脉冲输出。

RTC_CLKOUT_1：1Hz 脉冲输出。

RTC_CLKOUT_32：32Hz 脉冲输出。

RTC_CLKOUT_1024：1024Hz 脉冲输出。

RTC_CLKOUT_32768：32768Hz 脉冲输出。

参数 timer_ctrl 设定了 PCF8563 的 Timer Control(定时器控制)寄存器的初值，pcf8563.h 头文件定义了一些宏方便设置 the timer_ctrl 参数：

RTC_TIMER_OFF：关闭 PCF8563 倒计数定时器。

RTC_TIMER_CLK_1_60：设置 PCF8563 倒计数定时器的时钟频率为 1/60Hz。

RTC_TIMER_CLK_1：设置 PCF8563 倒计数定时器的时钟频率为 1Hz。

RTC_TIMER_CLK_64：设置 PCF8563 倒计数定时器的时钟频率为 64Hz。

RTC_TIMER_CLK_4096：设置 PCF8563 倒计数定时器的时钟频率为 4096Hz.。

unsigned char rtc_read(unsigned char address)：从 PCF8563 的 address 地址的寄存器读出一个字节。

void rtc_write(unsigned char address, unsigned char data)：写一个字节到 PCF8563 的 address 地址的寄存器。

unsigned char rtc_get_time(unsigned char *hour, unsigned char *min, unsigned char *sec)：返回 RTC(实时时钟)的时间。指针*hour、*min 和*sec 必须指向接收小时、分钟和秒的变量。如果函数返回 1，说明读的时间正确。如果函数返回 0，说明供电电压太低，时间不正确。

void rtc_set_time(unsigned char hour, unsigned char min, unsigned char sec)：设置 RTC 的时间，参数 hour、min 和 sec 对应时、分、秒。

void rtc_get_date(unsigned char *date, unsigned char *month, unsigned *year)：读取 RTC 的日历，指针*date、*month 和*year 指向接收日、月、年的变量。

void rtc_set_date(unsigned char date, unsigned char month, unsigned year)：设置 RTC 的日历。

void rtc_alarm_off(void)：关闭 RTC 的闹铃功能。

void rtc_alarm_on(void)：打开 RTC 的闹铃功能。

void rtc_get_alarm(unsigned char *date, unsigned char *hour, unsigned char *min)：读取 RTC 的闹铃的日期和时间。指针*date、*hour 和*min 指向接收日期、小时、分钟的变量。

void rtc_set_alarm(unsigned char date, unsigned char hour, unsigned char min)：设置 RTC 的闹铃的日期和时间。参数 date、hour 和 min 对应日期、小时、分钟。如果 date 是 0，这个参数将被忽略。调用这个函数后，闹铃被关闭，要调用 rtc_alarm_on 函数打开闹铃功能。

void rtc_set_timer(unsigned char val)：设置 PCF8563 定时器的值。

5．PCF8583 实时时钟函数

使用这些函数之前应当用#include "pcf8583.h"预处理。只有商业版的 CodeVisionAVR C Compiler 才有这部分功能。在 pcf8583.h 中自动包含 I^2C 总线函数，包含头文件之前，必须先声明哪些口线通过 I^2C 总线与 PCF8583 通信。

void rtc_init(unsigned char chip, unsigned char dated_alarm)：初始化 PCF8563。调用这个函数之前，必须调用函数 i2c_init 初始化 I^2C 线，调用其他的 PCF8583 函数之前，必须先调用此函数。如果 I^2C 总线上接有多个 PCF8583，必须通过参数 chip 对每一个 PCF8583 都初始化，一个 I^2C 总线上最多可以接 2 个 PCF8583，它们的地址是 0 或 1。

参数 dated_alarm 设定 RTC 闹铃的日期和时间，是两者同时起作用(dated_alarm=1)，还是只有时间起作用(dated_alarm=0)，调用这个函数后，闹铃被关闭，要调用 rtc_alarm_on 函数打开闹铃功能。

unsigned char rtc_read(unsigned char chip, unsigned char address)：读 PCF8583 的 SRAM 中的一个字节。

void rtc_write(unsigned char chip, unsigned char address, unsigned char data)：向 PCF8583 的 SRAM 中写一个字节。在向 SRAM 中写数据时，地址 10h 和 11h 存放的是年份的值。

unsigned char rtc_get_status(unsigned char chip)：返回 PCF8583 的控制/状态寄存器的值。调用这个函数时，全局变量__rtc_status 和__rtc_alarm 自动更新。变量__rtc_status 存放着控制/状态寄存器的值，如果变量__rtc_alarm 为 1，表示有闹铃。

void rtc_get_time(unsigned char chip, unsigned char *hour, unsigned char *min, unsigned char *sec, unsigned char *hsec)：读出 RTC 的当前时间。指针*hour、*min、*sec 和*hsec 指向接收小时、分钟、秒和百分之一秒的变量。

void rtc_set_time(unsigned char chip, unsigned char hour, unsigned char min, unsigned char sec, unsigned char hsec)：设置 RTC 的时间。参数 hour、min、sec 和 hsec 对应小时、分钟、秒和百分之一秒的值。

void rtc_get_date(unsigned char chip, unsigned char *date, unsigned char *month, unsigned *year)：读出 RTC 的当前日期。指针*date、*month、*year 指向接收日、月、年的变量。

void rtc_set_date(unsigned char chip, unsigned char date, unsigned char month, unsigned

year)：设置 RTC 的日期。

void rtc_alarm_off(unsigned char chip)：关闭 RTC 的闹铃。

void rtc_alarm_on(unsigned char chip)：打开 RTC 的闹铃。

void rtc_get_alarm_time(unsigned char chip, unsigned char *hour, unsigned char *min, unsigned char *sec, unsigned char *hsec)：读出 RTC 闹铃的时间设置值。指针*hour、*min、*sec 和*hsec 指向接收小时、分钟、秒和百分之一秒的变量。

void rtc_set_alarm_time(unsigned char chip, unsigned char hour, unsigned char min, unsigned char sec, unsigned char hsec)：设置 RTC 的闹铃的时间值。参数 hour、min、sec 和 hsec 对应小时、分钟、秒和百分之一秒的值。

void rtc_get_alarm_date(unsigned char chip, unsigned char *date, unsigned char *month)：读出 RTC 的闹铃的日期设置值。指针*date、*month、*year 指向接收日、月、年的变量。

void rtc_set_alarm_date(unsigned char chip, unsigned char date, unsigned char month)：设置 RTC 闹铃的日期值。

6．DS1302 实时时钟函数

使用这些函数之前应当用#include "ds1302.h"预处理。只有商业版的 CodeVisionAVR C Compiler 才有这部分功能。在 ds1302 中自动包含 I^2C 总线函数，包含头文件之前，必须先声明哪些口线通过 I^2C 总线与 DS1302 通信。

void rtc_init(unsigned char tc_on, unsigned char diodes, unsigned char res)：初始化 DS1302。调用其他的 DS1302 函数之前必须先调用此函数，如果参数 tc_on 为 1，则打开涓流充电，参数 diodes 设定了涓流充电时使用二极管的个数，可以是 1 或 2，参数 res 指明了涓流充电时的电阻值：

0：没有电阻

1：2kΩ

2：4kΩ

3：8kΩ

unsigned char ds1302_read(unsigned char addr)：在 DS1302 的 SRAM 中的地址 addr 读一个字节。

void ds1302_write(unsigned char addr, unsigned char data)：在 DS1302 的 SRAM 中的地址 addr 写一个字节 data。

void rtc_get_time(unsigned char *hour, unsigned char *min, unsigned char *sec)：读出 RTC 的时间。指针*hour、*min、*sec 指向接收小时、分钟、秒的变量。

void rtc_set_time(unsigned char hour, unsigned char min, unsigned char sec)：设置 RTC 的时间。参数 hour、min、sec 对应小时、分钟、秒的值。

void rtc_get_date(unsigned char *date, unsigned char *month, unsigned char *year)：读出 RTC 的当前日期。指针*date、*month、*year 指向接收日、月、年的变量。

void rtc_set_date(unsigned char date, unsigned char month, unsigned char year)：设置 RTC 的日期。

7．DS1307 实时时钟函数

使用这些函数之前应当用#include "ds1307.h"预处理。只有商业版的 CodeVisionAVR C Compiler 才有这部分功能。在 ds1307.h 中自动包含 I^2C 总线函数，包含头文件之前，必须先声明哪些口线通过 I^2C 总线与 DS1307 通信。

void rtc_init(unsigned char rs, unsigned char sqwe, unsigned char out)：初始化 DS1307。调用这个函数之前，必须调用函数 i2c_init 初始化 I^2C 总线，调用其他的 DS1307 函数之前，必须先调用此函数。参数 rs 设定了 SQW/OUT 引脚上输出方波的频率：

0：1Hz

1：4096Hz

2：8192Hz

3：32768Hz

如果参数 sqwe 设为 1，则允许 SQW/OUT 引脚上的方波输出，参数 out 设定了禁止 SQW/OUT 引脚上的方波输出(sqwe=0)时引脚上的逻辑电平。

void rtc_get_time(unsigned char *hour, unsigned char *min, unsigned char *sec)：读出 RTC 的当前时间。指针*hour、*min，*sec 指向接收小时、分钟、秒的变量。

void rtc_set_time(unsigned char hour, unsigned char min, unsigned char sec)：设置 RTC 的时间。参数 hour、min、sec 对应小时、分钟、秒的值。

void rtc_get_date(unsigned char *date, unsigned char *month, unsigned char *year)：读出 RTC 的当前日期。指针*date、*month、*year 指向接收日、月、年的变量。

void rtc_set_date(unsigned char date, unsigned char month, unsigned char year)：设置 RTC 的日期。

7.3.11　单总线函数

1．单线通信协议函数

使用这些函数之前应当用#include "1wire.h"预处理。只有商业版的 CodeVisionAVR C Compiler 才有这部分功能。在以下函数中，均以 MCU 为主机，外设(单总线器件)为从机，包含头文件之前，必须先声明哪些口线使用单线通讯协议与器件通信。

注意：由于单线协议函数使用时有严格的延时，所以操作期间要关闭中断。并且要在 Project->Configure->C Compiler 菜单设定正确的晶振频率。

unsigned char w1_init(void)：初始化总线上的器件。如果有器件返回 1，否则返回 0。

unsigned char w1_read(void)：从总线上读一个字节。

unsigned char w1_write(unsigned char data)：在总线上写一个字节。如果写过程正常完成返回 1，否则返回 0。

unsigned char w1_search(unsigned char cmd,void *p)：返回总线上器件的个数。如果没有器件，返回 0。参数 cmd 是发给器件的命令，如 DS1820/DS1822 的搜索 ROM(Search ROM)

为 F0h，报警搜索(Alarm Search)为 Ech。指针 p 指向存放器件返回的 8 字节 ROM 码的 SRAM 区域。8 字节后会存放某些器件的状态字节(如 DS2405)，所以必须给每个器件开辟 9 个字节的 SRAM。如果总线上有多个器件，那么首先要使用 w1_search 读出所有器件的 ROM 码，以便在后面的过程中对它们进行寻址。

unsigned char w1_crc8(void *p, unsigned char n)：返回从地址 p 开始的 n 个字节的 8 位 CRC 校验。

2．DS1820/1822 温度传感器函数

使用这些函数之前应当用#include "ds1820.h"预处理。只有商业版的 CodeVisionAVR C Compiler 才有这部分功能。在以下函数中，均以 MCU 为主机，外设(单总线器件)为从机，在 ds1820.h 中包含了单总线的函数。包含头文件之前，必须先声明哪些口线使用单线通信协议与器件通信。

int ds1820_temperature_10(unsigned char *addr)：返回 ROM 码存在地址 addr 处的数组中的 DS1820/DS1822 的温度，温度的单位为 0.1 摄氏度。如果有错误，返回值为-9999。如果只有 1 个 DS1820/DS1822，就不需要 ROM 码，指针 addr 要设为 NULL。如果有多个器件要首先读 ROM 码，对每一个器件进行识别，然后才能调用 ds1820_temperature_10 对需要的器件通过 ROM 码进行地址匹配。

unsigned char ds1820_set_alarm(unsigned char *addr,signed char temp_low,signed char temp_high)：设置 DS1820/DS1822 的低温、高温报警温度，如果设置成功返回 1，否则返回 0，报警温度存在 DS1820/DS1822 的暂存器 SRAM 和 EEPROM 中。用来寻址器件的 ROM 码放在 addr 指向的数组，如果只有 1 个 DS1820/DS1822，就不需要 ROM 码，指针 addr 要设为 NULL。DS1820/DS1822 的温度报警状态可以用 w1_search 函数发送报警搜索(Alarm Search) Ech 命令检测到。

7.3.12 SPI 函数

使用这些函数之前应当用#include "spi.h"预处理。

unsigned char spi(unsigned char data)：发送一个字节，同时接收一个字节。注意：spi 函数通信使用查询方式，所以不需要设置 SPI 中断允许标志位 SPIE，但在调用 spi 函数之前要先设置 SPI 控制寄存器 SPCR。

7.3.13 电源管理函数

使用这些函数之前应当用#include "sleep.h"预处理。

void sleep_enable(void)：允许进入低功耗模式。

void sleep_disable(void)：禁止进入低功耗模式，使用这可以防止意外进入低功耗模式。

void idle(void)：进入闲置模式。调用这个函数之前必须先调用 sleep_enable 函数，以允

许进入低功耗模式。在这种模式下，CPU 停止工作，但定时器(计数器)、看门狗和中断系统继续工作。MCU 可以由外部或内部中断唤醒。

void powerdown(void)：进入掉电模式。调用这个函数之前必须先调用 sleep_enable 函数，以允许进入低功耗模式。在这种模式下，外部晶振停振，看门狗和外部中断继续工作。只有外部复位、看门狗复位和外部中断可以唤醒 MCU。注意：外部中断的电平持续时间应长于晶振启动时间。

void powersave(void)：进入休眠模式。调用这个函数之前必须先调用 sleep_enable 函数，以允许进入低功耗模式。这种模式与掉电模式类似。

7.3.14　格雷码转换函数

使用这些函数之前应当用#include "gray.h"预处理。

unsigned char gray2binc(unsigned char n)：将格雷码形式的 n 转换到二进制形式。

unsigned char gray2bin(unsigned int n)：将格雷码形式的 n 转换到二进制形式。

unsigned char gray2binl(unsigned long n)：将格雷码形式的 n 转换到二进制形式。

unsigned char bin2grayc(unsigned char n)：将二进制形式的 n 转换到格雷码形式。

unsigned char bin2gray(unsigned int n)：将二进制形式的 n 转换到格雷码形式。

unsigned char bin2grayl(unsigned long n)：将二进制形式的 n 转换到格雷码形式。

7.4　CodeVisionAVR 应用实例

7.4.1　延迟函数

由于 CVAVR 提供了两个延迟函数，因此在 CVAVR 中使用软件延时只要直接调用延迟函数就可以了，如果需要精确的延时时间，必须关闭中断。

如需要延迟 100ms：

```
delay_ms(100);
```

如需要精确延时 100μs：

```
#asm("cli")                    //关全局中断
delay_us(100);
#asm("sei")                    //开全局中断
```

说明：

1．使用延时函数前必须设定准确晶振频率。

2．在源文件前面必须加上“#include "delay.h "”预处理行。

3．汇编伪指令后面不能加上“;”，这一点与 ICCAVR 是不同的。

4．在实际应用中一般不能直接使用软件进行长时间延时，因为 MCU 一直停留在延时函数中，不能再干其他的事情(除了中断外)，对长时间(一般几十到几百毫秒)的延时，一般使用定时器中断，如果需要更长时间的延时，必须使用定时器中断来完成。

7.4.2 字符型 LCD

```
/*********************************************
Project          :向 LCD 写入一个字符
Date             :2002-11-28
interpreter      :刘洲
Rework           :沈文
From             :广州双龙电子
Comments         :2x16 字符 LCD，LCD 连接至 PORTC

Chip type            :AT90S8515
Clock frequency      :4.000000 MHz
Memory model         :Small
Internal RAM size    :512
External RAM size    :0
Data Stack size      :128
*********************************************/

/*********************************************
    [LCD]     [PORTC]
1   GND  —  GND
2   +5V   —  VCC
3   VLC  —  LCD HEADER Vo
4   RS   —  PC0
5   RD   —  PC1
6   EN   —  PC2
11  D4   —  PC4
12  D5   —  PC5
13  D6   —  PC6
14  D7   —  PC7
*********************************************/

#include "90s8515.h"
#asm                                    //声明与 LCD 模块连接的端口
```

```
.equ __lcd_port=0x15
#endasm
#include "lcd.h"                          //包含头文件
typedef unsigned char byte;
/*  自定义字符的点阵数据，一个指向右上角的箭头  */
flash byte char0[8]={0b0000000, 0b0001111,0b0000011,0b0000101,0b0001001,
                     0b0010000,0b0100000,0b1000000};
void define_char(byte flash *pc,byte char_code)
    {
    byte i,a;
    a=(char_code<<3)|0x40;
    for (i=0; i<8; i++) lcd_write_byte(a++,*pc++);
    }
void main(void)
    {
    lcd_init(16);                         //初始化 2 行 16 列 LCD
    define _char (char0,0);               //自定义字符 0
    lcd_gotoxy(0,0);                      //设定显示坐标到 0 行 0 列
    lcd_putsf("User char 0:");            //显示 Flash 中的字符串
    lcd_putchar(0);                       //显示自定义字符 0
    while (1) ;                           //死循环
    }
```

7.4.3　访问 AT24C02

```
/*********************************************
Project          :访问 ATmel 24C02(256 字节 EEPROM)
Date             :2002-11-28
interpreter      :刘洋
Rework           :沈文
From             :广州双龙电子
Comments         :使用 PORTB 作 I2C 总线
                  SDA 为 PB3
                  SCL 为 PB4

Chip type           :AT90S8515
Clock frequency     :4.000000 MHz
Memory model        :Small
Internal RAM size   :512
External RAM size   :0
Data Stack size     :128
*********************************************/
```

```
#include "90s8515.h"
#asm                                    //声明 I2C 总线使用的端口
.equ __i2c_port=0x18
.equ __sda_bit=3
.equ __scl_bit=4
#endasm
#include "i2c.h"                        //包含头文件
#include "delay.h"
#define EEPROM_BUS_ADDRESS 0xa0
//从 EEPROM 中读一个字节
unsigned char eeprom_read(unsigned char address)
   {
   unsigned char data;
   i2c_start();
   i2c_write(EEPROM_BUS_ADDRESS);
   i2c_write(address);
   i2c_start();
   i2c_write(EEPROM_BUS_ADDRESS | 1);
   data=i2c_read(0);
   i2c_stop();
   return data;
   }
//向 EEPROM 写一个字节
void eeprom_write(unsigned char address, unsigned char data)
   {
   i2c_start();
   i2c_write(EEPROM_BUS_ADDRESS);
   i2c_write(address);
   i2c_write(data);
   i2c_stop();
   delay_ms(10);                        //延时 10ms 等待写操作完成
   }
void main(void)
   {
   unsigned char i;
   i2c_init();                          //初始化 I2C 总线
   eeprom_write(0xaa,0x55);             //在地址 AAh 写入 0x55
   i=eeprom_read(0xaa);                 //从地址 0xAA 读一个字节
   while (1);                           //死循环
   }
```

7.4.4　使用 I²C 总线访问 LM75

```
/*****************************************
Project          :使用 I2C 总线访问 LM75 温度传感器
Date             :2002-11-28
interpreter      :刘洲
Rework           :沈文
From             :广州双龙电子
Comments         :使用 PORTB 作 I2C 总线
                  SDA 为 PB3
                  SCL 为 PB4

Chip type           :AT90S8515
Clock frequency     :4.000000 MHz
Memory model        :Small
Internal RAM size   :512
External RAM size   :0
Data Stack size     :128
*****************************************/

#include "90s8515.h"
#asm                                    //声明 I2C 总线使用的端口
.equ __i2c_port=0x18
.equ __sda_bit=3
.equ __scl_bit=4
#endasm
#include "lm75.h"                       //包含 LM75 头文件
#asm                                    //声明 LCD 模块在 PORTC
.equ __lcd_port=0x15
#endasm
#include "lcd.h"                        //包含 LCD 头文件*/
#include <stdio.h>                      //包含 sprintf 的函数原型的头文件
#include <math.h>                       //包含 abs 函数原型的头文件
char display_buffer[33];
void main(void)
   {
   int t0,t1;
   lcd_init(16);                        //初始化 LCD，2 行 16 列
   i2c_init();                          //初始化 I2C 总线
   lm75_init(0,20,25,0);                //初始化地址 0 的 LM75，thyst=20 度，tos=25 度
```

```
lm75_init(1,30,35,0);                    //初始化地址 1 的 LM75，thyst=30 度，tos=35
while (1)                                //循环显示温度
  {
  t0=lm75_temperature_10(0);             //读地址 0 的温度
  t1=lm75_temperature_10(1);             //读地址 1 的温度
  //准备要显示的温度在 display_buffer
  sprintf(display_buffer,"t0=%-i.%-u%cC\nt1=%-i.%-u%cC",t0/10,abs(t0%10),
  0xdf,t1/10,abs(t1%10),0xdf);
  lcd_clear();
  lcd_puts(display_buffer);              //显示温度
  }
}
```

7.4.5 使用 I²C 总线访问 PCF8563

```
/*********************************************
Project          :使用 I2C 总线访问 PCF8563 实时时钟
Date             :2002-11-28
interpreter      :刘洋
Rework           :沈文
From             :广州双龙电子
Comments         :使用 PORTB 作 I2C 总线
                  SDA 为 PB3.
                  SCL 为 PB4

Chip type           :AT90S8515
Clock frequency     :4.000000 MHz
Memory model        :Small
Internal RAM size   :512
External RAM size   :0
Data Stack size     :128
*********************************************/

#asm                                     //声明 I2C 总线使用的端口
.equ __i2c_port=0x18
.equ __sda_bit=3
.equ __scl_bit=4
#endasm
#include <pcf8563.h>
void main(void)
  {
```

```
    unsigned char ok,h,m,s;
    i2c_init();                          //初始化 I2C 总线
    //初始化 RTC，定时器中断允许，闹铃中断允许
    //CLKOUT 频率=1Hz，定时器时钟频率=1Hz */
    rtc_init(RTC_TIE_ON|RTC_AIE_ON,RTC_CLKOUT_1,RTC_TIMER_CLK_1);
    ok=rtc_get_time(&h,&m,&s);           //从 RTC 读时间
    }
```

7.4.6　使用单总线访问 DS1820

```
/*********************************************
Project          :使用单总线访问 DS1820
Date             :2002-11-28
interpreter      :刘洪
Rework           :沈文
From             :广州双龙电子
Comments         :使用 PORTB 的 PB2 作数据线

Chip type            :AT90S8515
Clock frequency      :4.000000 MHz
Memory model         :Small
Internal RAM size    :512
External RAM size    :0
Data Stack size      :128
*********************************************/

#include <90s8515.h>
#asm                                 //声明使用单总线的端口
.equ __wl_port=0x18 ;PORTB
.equ __wl_bit=2
#endasm
#include "ds1820.h"                  //包含单总线的头文件
#include "stdio.h"                   //包含有 printf 函数原型的头文件
#include "math.h"                    //包含有 abc 函数原型的头文件
#define MAX_DEVICES 8                //单总线上 DS1820 的最大个数为 8 个
/*定义一个二维数组，保存 DS1820/DS1822 的 ROM 码存储区，每个 DS1820 需 9 字节，其中前 8
字节是器件的 ROM 码，最后一个字节是 CRC */
unsigned char rom_codes[MAX_DEVICES,9];
/*给发生温度报警的器件分配 ROM 码存储空间*/
unsigned char alarm_rom_codes[MAX_DEVICES,9];
main()
```

```
{
unsigned char i,j,devices;
int temp;
UBRR=xtal/16/baud-1;                    //初始化 UART 波特率
UCR=8;                                  //初始化 UART 控制寄存器
/*检测有多少个 DS1820/DS1822，并存放它们的 ROM 码到 rom_codes 数组*/
devices=w1_search(0xf0,rom_codes);
printf("%-u DEVICE(S) DETECTED\n\r",devices); //显示器件的个数
if (devices==0) while (1);              //如果没有器件则进入死循环
/* 设置所有器件低温报警为 25 摄氏度，高温报警为 35 摄氏度*/
for (i=0;i<devices;i++)
     {
     printf("INITIALIZING DEVICE #%-u ", i+1);
     if (ds1820_set_alarm(&rom_codes[i,0],25,35))
     putsf("OK"); else putsf("ERROR");
     }
while (1)
     {
     for (i=0;i<devices;)               //测量并显示温度
          {
          temp=ds1820_temperature_10(&rom_codes[i,0]);
          printf("t%-u=%-i.%-u\xf8C\n\r",++i,temp/10,abs(temp%10));
          }
          /*显示发生温度报警的器件的号码*/
          printf("ALARM GENERATED BY %-u DEVICE(S)\n\r",
          w1_search(0xec,alarm_rom_codes));
     }
}
```

7.4.7 使用 SPI 访问 AD7896

```
/*********************************************
Project          :使用 SPI 访问 AD7896
Date             :2002-11-28
interpreter      :刘汗
Rework           :沈文
From             :广州双龙电子
Comments         :使用 PORTB 的 PB2 作数据线

Chip type            :AT90S8515
Clock frequency      :4.000000 MHz
```

```
Memory model        :Small
Internal RAM size   :512
External RAM size   :0
Data Stack size     :128
*****************************************/

/*****************************************
AD7896 与 AT90S8515 的连接:
[AD7896]    —    [AT9S8515 DIP40]
1 Vin
2 Vref=5V
3 AGND    —     20 GND
4 SCLK    —     8 SCK
5 SDATA   —     7 MISO
6 DGND    —    20 GND
7 CONVST  —    2 PB1
8 BUSY    —    1 PB0

2x16 的字符型 LCD 接在 PORTC:
[LCD]      —    [AT90S8515 DIP40]
1 GND     —    20 GND
2 +5V      —    40 VCC
3 VLC
4 RS       —    21 PC0
5 RD       —    22 PC1
6 EN       —    23 PC2
11 D4       —   25 PC4
12 D5       —   26 PC5
13 D6       —   27 PC6
14 D7       —   28 PC7
*****************************************/

#include "90s8515.h"
#asm                                    //声明 LCD 接在端口 C
.equ __lcd_port=0x15
#endasm
#include "lcd.h"                        //包含 LCD 头文件
#include "spi.h"                        //包含 SPI 头文件
#include "stdio.h"                      //包含有 sprintf 函数原型的头文件
#include "delay.h"                      //包含有 delay 函数原型的头文件
#define VREF 5000L                      //AD7896 参考电压[mV]
#define ADC_BUSY PINB.0                 //定义 AD7896 控制信号
```

```
#define NCONVST PORTB.1
char lcd_buffer[33];                    //LCD 显示缓存
unsigned read_adc(void)
    {
    unsigned result;
    NCONVST=0;                          //使用采样模式 1(高速采样)
    NCONVST=1;
    while (ADC_BUSY);                   //等待采样完成
    result=(unsigned) spi(0)<<8;        //通过 SPI 读 MSB
    result|=spi(0);                     //通过 SPI 读 LSB 并与 MSB 合并
    result=(unsigned) (((unsigned long)result*VREF)/4096L);  //计算采样
                                                              电压[mV]
    return result;                      //返回测量值
    }
void main(void)
    {
    /*初始化 PORTB
    PB.0 输入，接 AD7896 忙信号(BUSY)
    PB.1 输出，接 AD7896 启动采样(/CONVST)
    PB.2、PB.3 输入
    PB.4 输出(SPI/SS)
    PB.5 输入
    PB.6 输入(SPI MISO)
    PB.7 输出(SPI SCLK)*/
    DDRB=0x92;
    /*初始化 SPI 在主机模式，不需要中断，MSB 先发送，时钟极性负，
    SCK 空闲时为低，SCK=fxtal/4 */
    SPCR=0x54;
    /*  AD7896 工作在模式 1(高速采样)，/CONVST=1, SCLK=0  */
    PORTB=2;
    lcd_init(16);                       //初始化 LCD
    lcd_putsf("AD7896 SPI bus\nVoltmeter");
    delay_ms(2000);
    lcd_clear();
    while (1)                           //读并显示 ADC 输入电压
        {
        sprintf(lcd_buffer,"Uadc=%4umV",read_adc());
        lcd_clear();
        lcd_puts(lcd_buffer);
        delay_ms(100);
        }
    }
```

7.4.8　8 路 A/D 自动巡测系统

```
/*********************************************
Project          :用 AT90S8535 制八路 A/D 自动巡测系统
Version          :1.0
Date             :2002-11-28
Author           :Will
Rework           :沈文
From             :广州双龙电子
Comments         :
```

AT90S8535 作 0~7 通道 A/D 转换，用 LED 显示：其中 D7、D6 两位显示通道号，D3~D0 四位显示转换值(十六进制数 0~3FFH)。程序下载自 SL-AVR-2 后就开始执行，自动从 0 通道到 7 通道 A/D 转换并扫描显示，当按住 0~7 任一位数字键，转换该道值并显示一段时间，然后又自动循环显示。

本程序在 SL-AVR-2 上调试通过。

```
Chip type           :AT90S8535
Clock frequency     :8.000000 MHz
Memory model        :Small
Internal RAM size   :512
External RAM size   :0
Data Stack size     :128
*********************************************/

/*********************************************
```

硬件接口：

AT90S8535 的 PC1~PC4 接 SL279，控制 LED 显示及键盘。

PA0~PA7 接模拟输入电压。

AGND 接地。

AVCC 与 VREF 间接 1k 电阻，VRBF 到地接 100μF 电解电容。

AVC 与 VCC 间接一只 100Ω 电阻，AVCC 通过 104 片电容连接。

```
*********************************************/

#include "90s8535.h"
#include "delay.h"
unsigned int adc_data,temp2;
#define ADC_VREF_TYPE 0x00
//********** ADC 中断服务程序 **********
#pragma savereg-
interrupt [ADC_INT] void adc_isr(void)
    {
```

```
    #asm
    push r30
    push r31
    #endasm
    adc_data=ADCW;                    //读取ADC 转换结果
    #asm
    pop r31
    pop r30
    #endasm
    }
//********** 读取ADC转换结果带噪声消除 **********
#pragma savereg+
unsigned int read_adc(unsigned char adc_input)
    {
    ADMUX=adc_input|ADC_VREF_TYPE;
    #asm
    in r30,mcucr
    sbr r30,__se_bit
    cbr r30,__sm_mask
    out mcucr,r30
    sleep
    cbr r30,__se_bit
    out mcucr,r30
    #endasm
    return adc_data;
    }
//********** 定义全局变量 **********
unsigned char temp,temp1,temp3,sl,sh,ml,mh,hl,hh;
//********** 发送子程序 **********
void transmit(unsigned char para)
    {
    unsigned char i,j;
    PORTC.4=0;                        //SL279片选，低电平有效
    delay_us(50);
    j=para;
    for(i=8;i>0;i--)
        {
        if((j&0x80)==0x80)
            PORTC.2=1;                //数据端
        else
            PORTC.2=0;
        PORTC.3=1;
```

```
        delay_us(10);
        PORTC.3=0;
        delay_us(10);
        j=j<<1;
        }
    }
//********** 接收子程序 **********
void receive()
    {
    unsigned char k;
    DDRC.2=0;
    delay_us(50);
    for(k=8;k>0;k--)
        {
        PORTC.3=1;                    //时钟端
        delay_us(10);
        if(PINC.2==1)
            temp|=0x01;
        else
            temp&=0xfe;
        PORTC.3=0;
        delay_us(10);
        if(k>1)
            temp=temp<<1;
        }
    DDRC.2=1;
    }
//********** 初始化子程序 **********
void init()
    {
    unsigned char l;
    PORTC.4=1;                        //SL279 片选无效
    PORTC.3=0;                        //时钟端
    delay_ms(20);
    transmit(0xc8);                   //D0 位->0
    delay_us(10);
    transmit(0x00);
    PORTC.4=1;
    delay_ms(10);
    transmit(0xc9);                   //D1 位->0
    delay_us(10);
    transmit(0x00);
```

```
    PORTC.4=1;
    delay_ms(10);
    transmit(0xca);                    //D2 位->0
    delay_us(10);
    transmit(0x00);
    transmit(0xcb);                    //D3 位->0
    delay_us(10);
    transmit(0x00);
    PORTC.4=1;
    delay_ms(10);
    transmit(0xe0);                    //D4 位->-
    delay_us(10);
    l=0x00;
    l+=(0x08+0x08+0x08+0x08);
    transmit(l);
    PORTC.4=1;
    delay_ms(10);
    l+=0x08;
    transmit(0xe0);                    //D5 位->-
    delay_us(10);
    transmit(l);
    PORTC.4=1;
    delay_ms(10);
    transmit(0xce);                    //D6 位->0
    delay_us(10);
    transmit(0x00);
    PORTC.4=1;
    delay_ms(10);
    transmit(0xcf);                    //D7 位->0
    delay_us(10);
    transmit(0x00);
    PORTC.4=1;
    delay_ms(10);
    transmit(0x88);                    //去闪烁
    delay_us(10);
    transmit(0xff);
    PORTC.4=1;
    delay_ms(10);
    }
//********** 显示子程序 **********
void disp(unsigned char ls,unsigned char hs, unsigned char lm,unsigned charhm,
          unsigned char lh,unsigned char h)
```

```
    {
    unsigned char sll,shh,mll,mhh,hll,hhh;
    sll=ls;
    shh=hs;
    mll=lm;
    mhh=hm;
    hll=lh;
    hhh=h;
    transmit(0xc8);                    //D0 位显示
    delay_us(10);
    transmit(sll);
    PORTC.4=1;
    delay_ms(10);
    transmit(0xc9);                    //D1 位显示
    delay_us(10);
    transmit(shh);
    PORTC.4=1;
    delay_ms(10);
    transmit(0xca);                    //D2 位显示
    delay_us(10);
    transmit(mll);
    PORTC.4=1;
    delay_ms(10);
    transmit(0xcb);                    //D3 位显示
    delay_us(10);
    transmit(mhh);
    PORTC.4=1;
    delay_ms(10);
    transmit(0xce);                    //D6 位显示
    delay_us(10);
    transmit(hll);
    PORTC.4=1;
    delay_ms(10);
    tran smit(0xcf);                   //D7 位显示
    delay_us(10);
    transmit(hhh);
    PORTC.4=1;
    delay_ms(10);
    }
//********** 转换子程序 **********
void conver(unsigned char mid)
    {
```

```
    unsigned char k,p;
    p=mid;
    read_adc(p);
    #asm("cli")
    temp1=(unsigned char)adc_data;          //低位字节;
    temp2=(0xff00&adc_data);                //int
    for(k=0;k<8;k++)
        temp2=temp2>>1;
    temp3=(unsigned char)temp2;             //高位字节;
    sl=(0x0f&temp1);                        //送 D0 位;
    sh=(0xf0&temp1);
    for(k=0;k<4;k++)
        sh=sh>>1;                           //送 D1 位;
    ml=(0x0f&temp3);                        //送 D2 位;
    mh=(0xf0&temp3);
    for(   k=0;k<4;k++)
        mh=mh>>1;                           //送 D3 位;
    hl=(0x0f&p);                            //送 D6 位;
    hh=(0xf0&p);
    for(k=0;k<4;k++)
        hh=hh>>1;                           //送 D7 位;
    disp(sl,sh,ml,mh,hl,hh);
    delay_ms(1000);
}
//********** 主程序 **********
void main(void)
    {
    unsigned char m,s;                      //定义局部变量
    DDRA=0x00;                              //初始化 PortA
    PORTA=0x00;
    DDRC=0xFF;                              //初始化 PortC
    PORTC=0x00;
    GIMSK=0x00;                             //外部中断初始化，关闭 INT0 和 INT1
    MCUCR=0x00;
    init( );                                //调用初始化函数
    delay_ms(500);
    while(1)
        {
        s=0x00;
        ADCSR=0x8B;                         //ADC 初始化，时钟频率:1000.000kHz
        for(m=0;m<8;m++,s++)                //自动巡测 8 路
            {
```

```
#asm("sei")                    //全局中断允许
conver(s);
if(PINC.1==0)                  //有键按下
    {
    transmit(0x15);            //读键
    receive();
    PORTC.4=1;
    delay_us(10);
    switch(temp)               //根据键号作转换
        {
        case 0:
            {
            #asm("sei")        //全局中断允许
            conver(temp);
            delay_ms(1500);
            };
            break;
        case 1:
            {
            #asm("sei")        //全局中断允许
            conver(temp);
            delay_ms(1500);
            };
            break;
        case 2:
            {
            #asm("sei")        //全局中断允许
            conver(temp);
            delay_ms(1500);
            };
            break;
        case 3:
            {
            #asm("sei")        //全局中断允许
            conver(temp);
            delay_ms(1500);
            };
            break;
        case 4:
            {
            #asm("sei")        //全局中断允许
            conver(temp);
```

```
                    delay_ms(1500);
                    };
                    break;
                case 5:
                    {
                    #asm("sei")          //全局中断允许
                    conver(temp);
                    delay_ms(1500);
                    };
                    break;
                case 6:
                    {
                    #asm("sei")          //全局中断允许
                    conver(temp);
                    delay_ms(1500);
                    };
                    break;
                case 7:
                    {
                    #asm("sei")          //全局中断允许
                    conver(temp);
                    delay_ms(1500);
                    };
                    break;
                default: break;
                }
            }
        }
    }
}
```

第 8 章　IAR 软件使用初步

总部设在瑞典的 IAR SYSTEMS 公司是世界著名的软件生产厂家之一，IAR 公司生产的 IAR Embedded Workbench(简称 IAR EW)是一整套集成开发环境(IDE)，包括嵌入式 C/C++编译器、汇编器、连接定位器、库管理、项目管理及调试器等。IAR 的 ICC90 编译器是 IAR Embedded Workbench 内的一个部分(插件)，是与 ATMEL 的 AT90 系列 AVR 单片机同步开发的一个老牌的 C 语言工具，它有自己的源程序调试工具软件 C-SPY，也可以生成.cof 文件，在 ATMEL 的 avrstudio 中调试。

编者使用的是 IAR 2.31E 版的 DEMO 版，在本章中不会详细地介绍 IAR 的集成环境 IDE，只给出几个需要配置的关键点，其他方面的内容读者可以在使用时自行体会。

8.1　IAR Embedded Workbench 简介

8.1.1　安装

1．系统需求

IAR 可以在 Win98/Win2000/WinXP 操作系统下使用，最低的硬件需求是：

(1) X386 以上计算机；

(2) 50MB 以上硬盘空间；

(3) 64MB 内存；

2．安装过程

IAR 的安装过程比较简单，按照安装向导提示，按“Next”就可以了。从光盘安装 IAR Embedded Workbench 2.31E，双击 ewavr-ev.exe 文件，出现如图 8.1 所示的界面。

按下“Next”键，出现软件安装协议界面，如图 8.2 所示。

按下“Accept”键，依次进入安装目录选择，安装项目选择定制等界面，如图 8.3、图 8.4 和图 8.5 所示。

IAR Embedded Workbench 安装完成后，在安装目录(C:\Program Files\IAR Systems)下生成如图 8.6 的系统目录。

IAR 的正版软件有加密狗，只有把加密狗插在打印口上，双击相应的图标才能进入工作窗口。编者使用的是 30 天的试用版，不需要加密狗就可以运行。

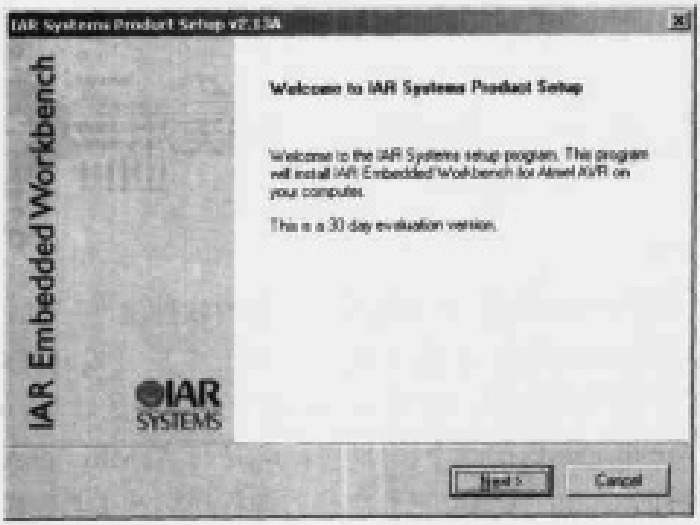


图 8.1 IAR Embedded Workbench 安装

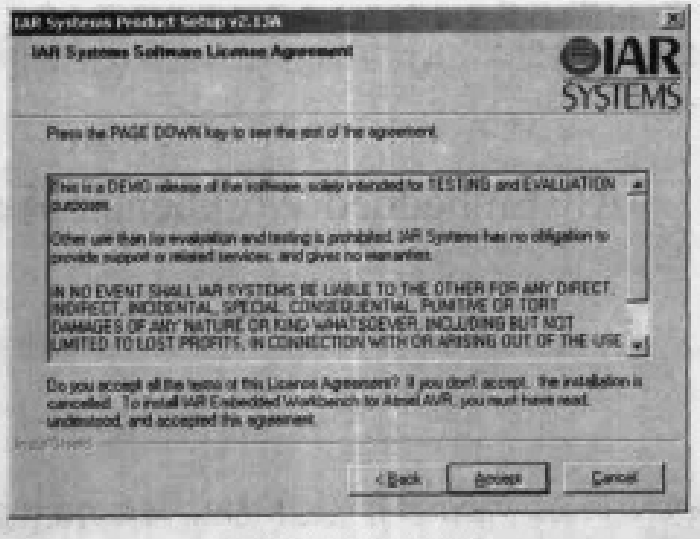


图 8.2 安装软件协议

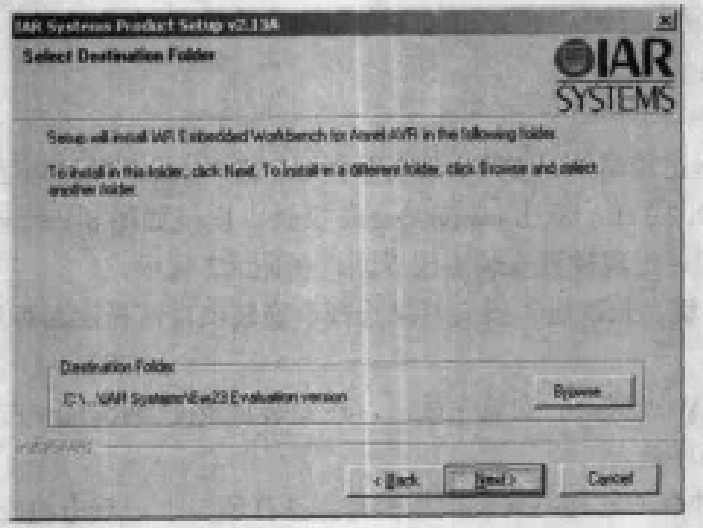


图 8.3 安装目录选择设定

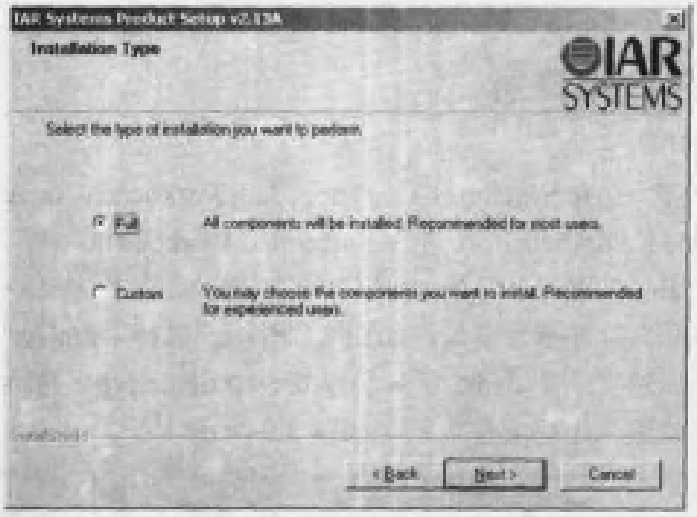


图 8.4　安装内容选择定制

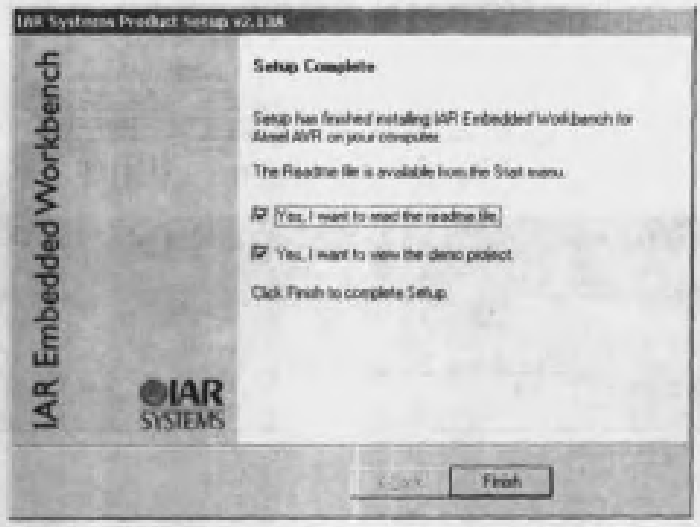


图 8.5　IAR Embedded Workbench 安装完成

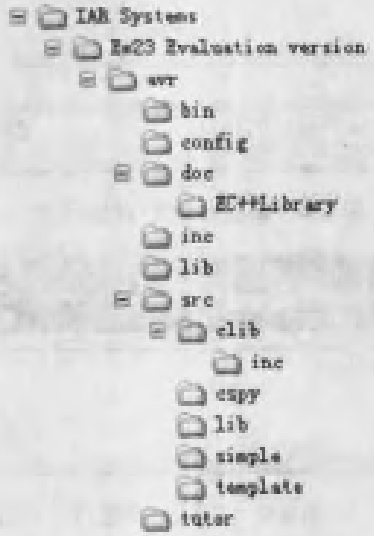


图 8.6　IAR Embedded Workbench 安装后的目录结构

8.1.2 配置 IAR C 编译器

选择“开始->程序->IAR System->IAR Embedded Workbench for Atmel AVR Evaluation version->IAR Embedded Workbench”，进入 Embedded Workbench。

1. 新建一个“demo”工程

选择“Files->New”打开新建窗口，如图 8.7 所示。选择“Project”，按下“确定”，进入文件存放目录窗口。建立一个 demo 目录和建立一个 demo.prj(名称为“demo”)工程文件，如图 8.8 所示。然后就进入工程文件窗口，如图 8.9 所示。

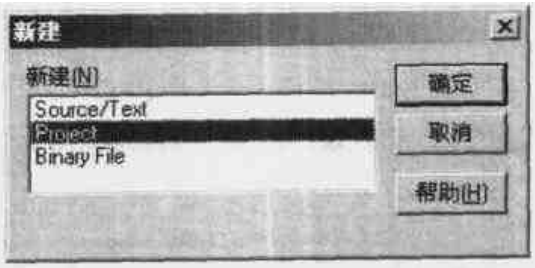


图 8.7 新建工程文件

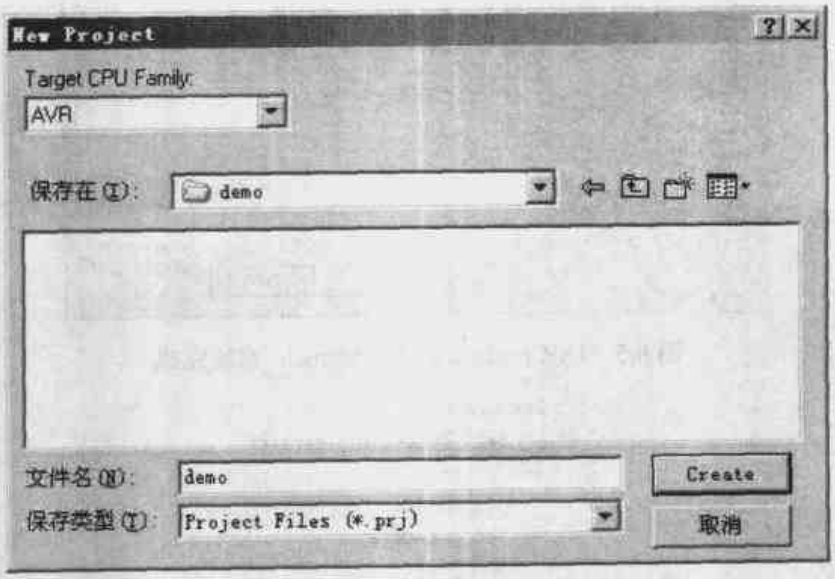


图 8.8 新建工程的存放目录

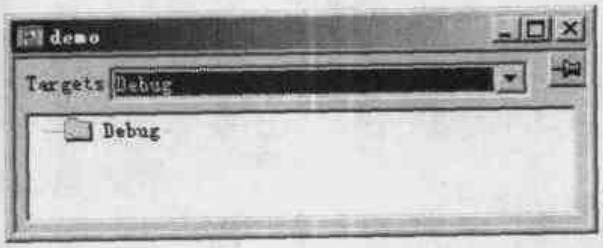


图 8.9 工程文件窗口

2. 编辑 C 源程序文件

选择“Files->New”打开新建窗口，如图 8.7 所示选择“Source/Text”，新建一个名为 untitled 的文件，输入 C 源程序，保存为 DEMO.C 文件，如图 8.10 所示。

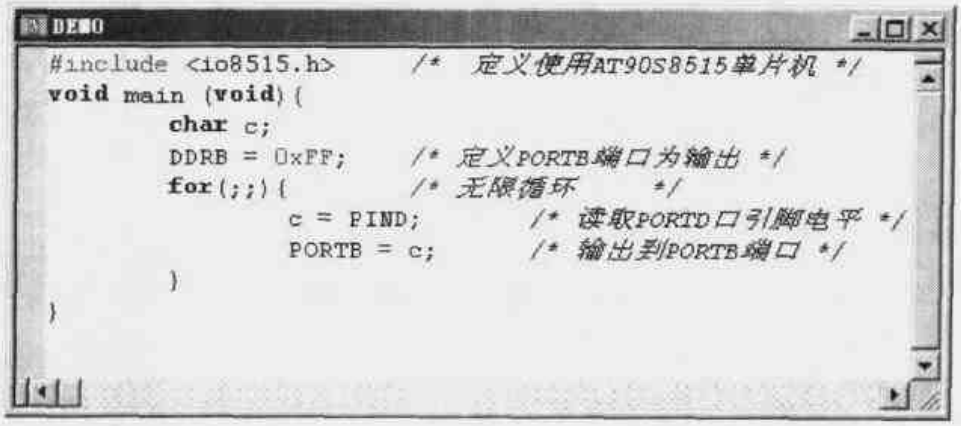


图 8.10　DEMO.C 源程序

3．配置 C 编译器

选择“Project -> Options”，打开“Options for target ‘debug’”窗口，在“Category”中选择“General”。在 target 选项卡下的“processor configuration”选择“-v1 Max 64 Kbyte data，8 Kbyte code”或者“—cpu= 8515，AT90S8515”。IAR 有七种处理器配置和多种 CPU 选择，表 8.1 给出其中处理器配置及其内存使用的应用。

表 8.1　处理器配置和内存使用

配　置	处理器名称	描　述
V0	Max 256 byte data，8 Kbyte code	编译时只可以使用 tiny 模式，IAR 中扩充的 near，far 和 huge 关键词不可用
V1	Max 64 Kbyte data，8 Kbyte code	编译时只可以使用 tiny 和 small 模式，tiny 模式默认使用 256 字节数据 SRAM，small 模式默认使用最多 64K 字节 SRAM。系统默认使用 tiny 内存模式，可以使用 tiny 和 near 关键词
V2	Max 256 byte data，128 Kbyte code	编译时只可以使用 tiny 模式
V3	Max 64 Kbyte data，128 Kbyte code	编译时只可以使用 tiny 和 small 模式
V4	Max 16 Mbyte data，128 Kbyte code	编译时只可以使用 small 和 large 模式
V5	Max 64 Kbyte data，8 Mbyte code	编译时只可以使用 tiny 和 samll 模式
V6	Max 16 Mbyte data，8 Mbyte code	编译时只可以使用 small 和 large 模式

另外，新版本的 IAR 编译器也支持 CPU 选择，从 tiny 系列到 S90 系列，再到 mega 系列，读者可以像在 ICCAVR 编译器中那样自己定义使用的 CPU，如使用 AT90S8515，可以选择“—cpu= 8515”。内存模式配置可以根据需要定制，决定使用哪一种指针类型。

根据表 8.1 所列出的模式，Memery model 选项选择“small”模式，如图 8.11 所示。Memery model 配置全局数据和变量(非堆栈)，局部变量(堆栈)数据和动态分配的变量(如使用 malloc 指令)在数据空间的存放方式。表 8.2 为从 Memery model 的角度给出的处理器模

式与 Memery model 的关系。

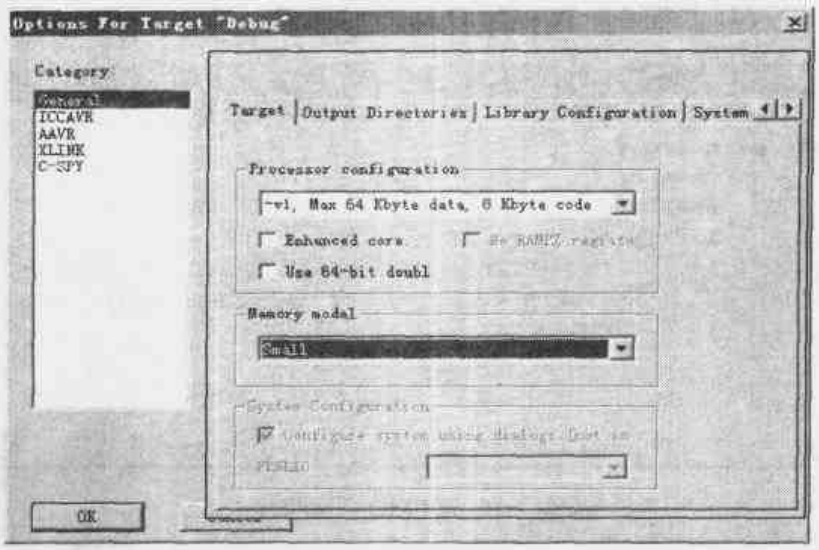


图 8.11 编译器目标器件设置

表 8.2 Memery model 与处理器模式关系

Memery model	程序空间指针	数据空间指针	堆栈最大空间	处理器模式
tiny	__tiny	__tiny	256 byte	V0,V1,V2,V3,V5
small	__near	__near	64 Kbyte	V1,V3,V4,V5,V6
large	__far	__far	64 Kbyte	V4,V6

其他的如“output directories”、“Library configuration”和“System configuration”选项可以采用默认的设置，不必更改。

4. 设置链接器

在“Category”下选择“XLINK”选项，出现如图 8.12 所示界面。

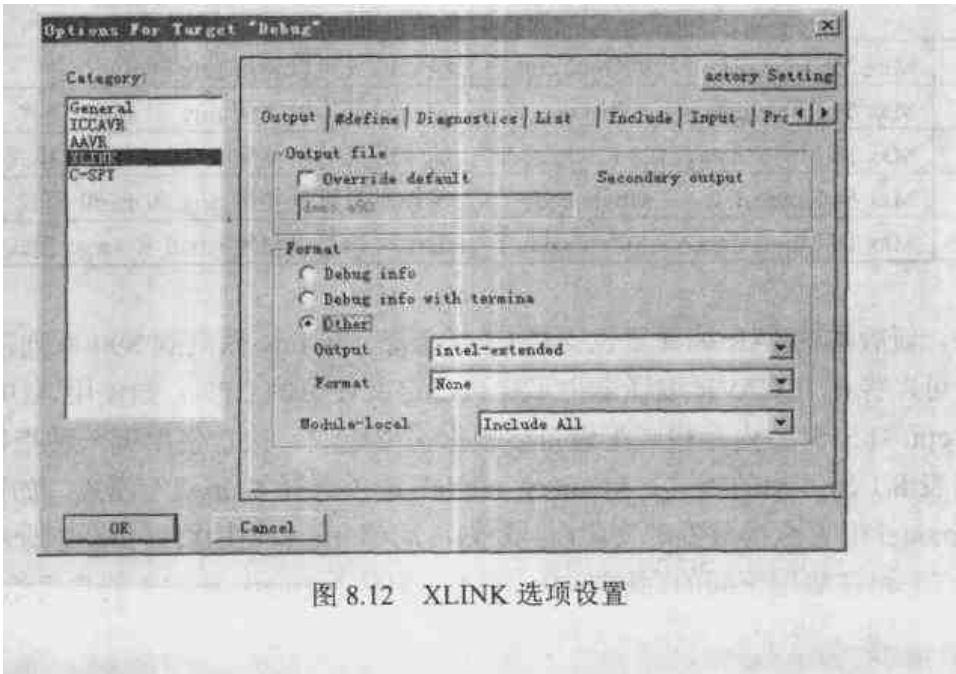


图 8.12 XLINK 选项设置

要生成可以下载的文件(Intel 格式的.hex 文件)，必须在“Format”选项里选择“Other”，并且在“Output”栏内选择“intel-extended”。当然，如果只需要 IAR 给出调试信息而不需要生成下载文件，可以在“Format”选项下选择“Debug info”或“Debug info with terminal”。

在 XLINK 选项下，可以通过“Include”选项卡中的参数选择处理器配置，设定内存模式，如图 8.13 所示。

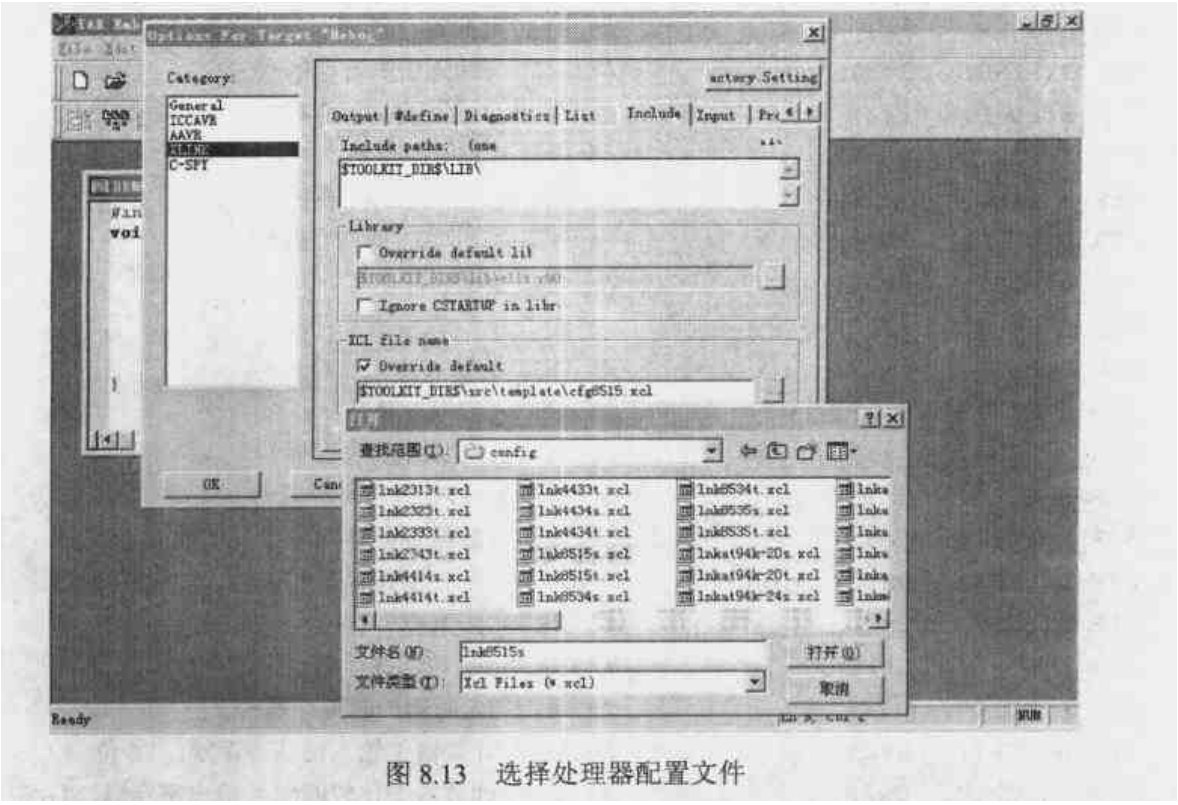


图 8.13　选择处理器配置文件

其他的配置选项都可以使用 IDE 的默认值，下面就可以按下“F9”键(或者“Project -> Make”)编译 project 了。编译成功后，在相应的目录下生成所需的文件(调试文件为*.d90，下载文件为*.a90)。

8.2　使用 IAR 寄存器和位操作

8.2.1　使用 IAR 寄存器

使用 IAR，将从了解寄存器开始。IAR 不同于 ICCAVR，IAR 进行了语法扩充——定义新的数据类型 sfrb 和 sfrw，在 C 中可以直接访问 MCU 的有关寄存器。打开目录 inc 下的 io8515.h 文件，可以看到 MCU 寄存器的定义，下面给出清单。

```
SFR_B(ACSR,    0x08)                    /* 模拟比较器控制状态寄存器*/
```

```
SFR_B(UBRR,   0x09)          /* UART 波特率寄存器*/
SFR_B(UCR,    0x0A)          /* UART 控制寄存器 */
SFR_B(USR,    0x0B)          /* UART 状态寄存器 */
SFR_B(UDR,    0x0C)          /* UART 数据寄存器 */
SFR_B(SPCR,   0x0D)          /* SPI 控制寄存器 */
SFR_B(SPSR,   0x0E)          /* SPI 状态寄存器 */
SFR_B(SPDR,   0x0F)          /* SPI 数据寄存器 */
SFR_B(PIND,   0x10)          /* 端口 D 输入引脚寄存器*/
SFR_B(DDRD,  0x11)           /* 端口 D I/O 方向寄存器*/
SFR_B(PORTD,  0x12)          /* 端口 D 数据寄存器 */
SFR_B(PINC,   0x13)          /* 端口 C 输入引脚寄存器*/
SFR_B(DDRC,   0x14)          /* 端口 C I/O 方向寄存器*/
SFR_B(PORTC,  0x15)          /* 端口 C 数据寄存器 */
SFR_B(PINB,   0x16)          /* 端口 B 输入引脚寄存器*/
SFR_B(DDRB,   0x17)          /* 端口 B I/O 方向寄存器*/
SFR_B(PORTB,  0x18)          /* 端口 B 数据寄存器 */
SFR_B(PINA,   0x19)          /* 端口 A 输入引脚寄存器*/
SFR_B(DDRA,   0x1A)          /* 端口 A I/O 方向寄存器*/
SFR_B(PORTA,  0x1B)          /* 端口 A 数据寄存器 */
SFR_B(EECR,   0x1C)          /* EEPROM 控制寄存器*/
SFR_B(EEDR,   0x1D)          /* EEPROM 数据寄存器*/
SFR_W(EEAR,   0x1E)          /* EEPROM 地址寄存器，16 位*/
SFR_B(WDTCR,  0x21)          /* 看门狗定时器控制寄存器*/
SFR_W(ICR1,   0x24)          /* 定时/计数器 1 输入捕获寄存器，16 位 */
SFR_W(OCR1B,  0x28)          /* 定时/计数器 1 比较匹配 B 输出寄存器，16 位*/
SFR_W(OCR1A,  0x2A)          /* 定时/计数器 1 比较匹配 A 输出寄存器，16 位*/
SFR_W(TCNT1,  0x2C)          /* 定时/计数器 1 寄存器，16 位*/
SFR_B(TCCR1B, 0x2E)          /* 定时/计数器 1 控制状态寄存器 1B*/
SFR_B(TCCR1A, 0x2F)          /* 定时/计数器 1 控制状态寄存器 1A*/
SFR_B(TCNT0,  0x32)          /* 定时/计数器 0 寄存器*/
SFR_B(TCCR0,  0x33)          /* 定时/计数器 0 控制状态寄存器*/
SFR_B(MCUCR,  0x35)          /* MCU 通用控制寄存器*/
SFR_B(TIFR,   0x38)          /* 定时/计数器中断状态寄存器*/
SFR_B(TIMSK,  0x39)          /* 定时/计数器中断屏蔽寄存器*/
SFR_B(GIFR,   0x3A)          /* 通用中断标志寄存器*/
SFR_B(GIMSK,  0x3B)          /* 通用中断屏蔽寄存器*/
SFR_W(SP,     0x3D)          /* 指针寄存器，16 位*/
SFR_B(SREG,   0x3F)          /* MCU 状态寄存器*/
```

以 SREG(MCU 状态寄存器)和 TCNT1(定时/计数器 1 寄存器)为例说明：

```
SFR_B(SREG,   0x3F)          /* 定义 8 位寄存器 SREG 在 IO 空间的 0x3F 地址 */
SFR_W(TCNT1,  0x2C)          /* 定义 16 位寄存器 TCNT1 在 IO 空间，低字节在 0x2C
                                地址，高字节在 0x2D 地址*/
```

这相当于 ICCAVR 中采用强制类型转换型指针的下列定义：

```
#define SREG (*(volatile unsigned char *)0x5F);     /* 定义 8 位寄存器在数据内
                                                       存中的映射地址 */
#define TCNT1 (*(volatile unsigned int *)0x4C);     /* 定义 16 位寄存器在数据
                                                       内存中的映射地址，低字
                                                       节在 0x4C 地址，高字节
                                                       在 0x4D 地址*/
```

写定时/计数器 1 的寄存器 TCNT1：

```
/*所有 I/O 寄存器都可以象普通变量一样访问 */
#include "io8515.h"                 /* 使用 AT90S8515 */
void main(void)
    {
    DDRD=0xFF;                      /* Port D 输出 */
    TCNT1=0xE080;                   /* 设置定时/计数器 1 寄存器 */
    /* 其他操作 */
    }
```

8.2.2　IAR 位操作

IAR 中的位定义与 ICCAVR 一样，没有扩充专门的位变量类型，而使用#define 伪指令定义寄存器，如 MCUCR 寄存器中的位：

```
/* MCU general Control Register */
#define    SRE          7
#define    SRW          6
#define    SE           5
#define    SM           4
#define    ISC11        3
#define    ISC10        2
#define    ISC01        1
#define    ISC00        0
```

其他寄存器的位定义可以参看 io8515.h 文件，这里不再列出。

IAR 的位操作与 ICCAVR 一样，只能使用 ANSI C 的位运算，幸好 ANSI C 的位运算功能足够强大，足以应付一般情况的需要。如设置/清除 SRE 位的操作：

```
/*IAR 的位操作*/
#include "io8515.h"                 /* 使用 AT90S8515 */
```

```
void main(void)
    {
    MCUCR|=(1<<SRE);              /* 设置 SRE 位 */
    MCUCR&=~(1<<SRE);             /* 清除 SRE 位 */
    if(PORTD&0x01)                /* 测试 PORTD 的第 0 位是否为 1 */
    PORTB|=(1<<PINB0);            /* 输出到 PORTB 的第 0 位*/
    /* 其他操作 */
    }
```

另外，也可像 ICCAVR 中一样，定义一些宏来进行位操作，生成的汇编代码具有很高的效率。如：

```
/*可定义位操作的宏*/
#define SET_BIT(x, y) (x|=(1<<y))        /* 置 x 中的第 y 位为 1 */
#define CLR_BIT(x, y) (x&=~(1<<y))       /* 清 x 中的第 y 位为 0 */
#define GET_BIT(x, y) (x&(1<<y))         /* 读取 x 中的第 y 位 */
```

这样，上面的位定义以后，在 C 中可以使用下面的代码操作：

```
/*IAR 的位操作*/
#include "io8515.h"                  /* 使用 AT90S8515 */
#define SET_BIT(x, y)  (x|=(1<<y))
#define CLR_BIT(x, y)  (x&=~(1<<y))
#define GET_BIT(x, y)  (x&(1<<y))
void main(void)
{
SET_BIT(MCUCR ,SRE);                 /* 设置 SRE 位 */
CLR_BIT(MCUCR,SRE);                  /* 清除 SRE 位 */
if(GET_BIT(PORTD ,0))                /* 测试 PORTD 的第 0 位是否为 1 */
SET_BIT(PORTB ,PINB0);               /* 输出到 PORTB 的第 0 位*/
/* 其他操作 */
}
```

8.3 IAR 中断向量和中断使用

1. IAR 中断向量定义

IAR 的头文件中定义 MCU 的中断向量，它使用#define 伪指令。以 AT90S8515 为例，定义 13 个中断向量。

```
#define RESET_vect          (0x00)      /* 上电复位中断*/
#define INT0_vect           (0x02)      /* 外部中断 INT0 中断 */
#define INT1_vect           (0x04)      /* 外部中断 INT1 中断 */
#define TIMER1_CAPT1_vect   (0x06)      /* 定时/计数器 1 捕获中断 */
#define TIMER1_COMPA_vect   (0x08)      /* 定时/计数器 1 比较匹配 A 中断 */
#define TIMER1_COMPB_vect   (0x0A)      /* 定时/计数器 1 比较匹配 B 中断 */
#define TIMER1_OVF1_vect    (0x0C)      /* 定时/计数器 1 溢出中断 */
#define TIMER0_OVF0_vect    (0x0E)      /* 定时/计数器 0 溢出中断 */
#define SPI_STC_vect        (0x10)      /* SPI 中断 */
#define UART_RX_vect        (0x12)      /* UART 接收完成中断 */
#define UART_UDRE_vect      (0x14)      /* UART 发送寄存器空中断 */
#define UART_TX_vect        (0x16)      /* UART 发送完成中断 */
#define ANA_COMP_vect       (0x18)      /* 模拟比较器中断 */
```

2．中断服务程序的定义

定义了中断向量，可以使用#prama vector=中断向量，利用这种方式声明定时/计数器 0 中断服务程序如下：

```
#pragma vector=TIMER0_OVF0_vect          /* 中断向量在 io8515.h 中有定义 */
__interrupt void timers0()               /* 中断服务程序在 2.26c 中的说明方式 */
   {
   /* 中断服务程序中的程序代码 */
   }
```

8.4　IAR 数据类型和数据空间

本节介绍 AVR IAR 编译器支持的数据类型，它支持 ISO/ANSI C 定义的所有基本数据类型。

8.4.1　数据类型及取值范围

整型数据类型：

表 8.3 给出整型数据类型的长度及其取值范围。

表 8.3　整型数据类型

数 据 类 型	长度/bits	取 值 范 围
char	8	0~255
signed char	8	-128~127

续表

数 据 类 型	长度/bits	取 值 范 围
unsigned char	8	0~255
short	16	-32768~32767
signed short	16	-32768~32767
unsigned short	16	0~65535
int	16	-32768~32767
signed int	16	-32768~32767
unsigned int	16	0~65535
long	32	-2^31~2^31-1
signed long	32	-2^31~2^31-1
unsigned long	32	0~2^32-1

注意：

(1) 枚举类型(Enum type)

在枚举类型下，“enum”关键词中应选用占用空间最小的数据类型，通常都是定义为 char 类型。

(2) 字节类型(Char type)

对字节类型(char type)，编译器默认为 unsigned 类型，与 ICCAVR 默认为无符号类型是相同的，但是与 ANSI C 及其他编译器(如 Keil、CVAVR)是不同的。当然，可以使用 --char_is_signed 参数将字节类型默认为有符号字节类型(signed)。

(3) 浮点数据类型

IAR 使用的浮点数据类型符合 IEEE-754 标准，其中浮点数据类型定义为 32 位的数据。表 8.4 给出浮点数据类型的长度和范围。

表 8.4 浮点数据类型

数 据 类 型	数 据 范 围	小 数 部 分	指 数 部 分	尾 数 部 分
float	±1.18E-38~±3.39E+38	7	8	23
double	±2.23E-308~±1.79E+308	15	11	52

注意：

64 位的“double”类型可以通过编译参数"--64bit_doubles"来定义。

8.4.2 数据空间

AVR 有三种不同类型的空间：程序空间 FLASH、数据空间 DATA 和 EEPROM 空间。表 8.5 给出 AVR IAR 编译器支持的数据空间类型，各种数据类型的声明使用"#pragma location"伪指令。

表 8.5　各种数据类型的声明使用的伪指令

数 据 空 间	描　　述
CODE	声明为程序空间 __nearfunc
CSTACK	使用内部数据堆栈
DIFUNCT	声明在 main 函数之前执行的构造函数 CSTARTUP
EEPROM_AN	存放本地初始化的 EEPROM 变量
EEPROM_I	存放下载时才初始化的 EEPROM 变量
EEPROM_N	存放已经初始化的 EEPROM 变量
FAR_C	用于存放 __far 类型常数数据
FAR_F	存放静态和全局的程序空间(__farflash)变量
FAR_I	存放静态和全局的声明为 non-zero 类型的数据和变量
FAR_ID	存放声明在 FAR_I 数据空间中的变量的初始化数值
FAR_N	存放静态和全局的 __far 变量(非易失性)
FAR_Z	存放静态和全局未初始化或 non-zero 型的 __far 变量
FARCODE	存放声明为 __farfunc 的程序代码
HEAP	存放用于 malloc, calloc 和 free 指令的堆栈数据
HUGE_C	用于存放 __huge 类型常量数据，包括字符串
HUGE_F	存放静态和全局的 __hugeflash 型变量
HUGE_I	存放静态和全局的__hugeflash 型的 non-zero 变量
HUGE_ID	存放声明在 HUGE_I 数据空间中的变量的初始化数值
HUGE_N	存放静态和全局的__huge 变量(非易失性)
HUGE_Z	存放静态和全局未初始化或 non-zero 型的 __huge 变量
INTVEC	存放复位和中断向量
NEAR_C	用于存放 __tiny 和 __near 类型常量数据，包括字符串
NEAR_F	存放静态和全局的 __flash 型变量
NEAR_I	存放静态和全局的__near 型的 non-zero 变量
NEAR_ID	存放声明在 NEAR_I 数据空间中的变量的初始化数值
NEAR_N	存放静态和全局的__near 变量(非易失性)
NEAR_Z	存放静态和全局未初始化或 non-zero 型的 __near 变量
RSTACK	存放用于函数返回数据的堆栈空间
TINY_F	存放静态和全局的 __tinyflash 型变量
TINY_I	存放静态和全局的__tiny 型的 non-zero 变量
TINY_ID	存放声明在 TINY_I 数据空间中的变量的初始化数值
TINY_N	存放静态和全局的__tiny 变量(非易失性)

8.5 IAR 操作 MCU 外设

本节介绍使用 IAR 操作 AVR 单片机的外设，在前面几章里已介绍过外设的原理，这里就不再赘述了，只给出各种操作的源程序。IAR 与 ICCAVR 的语法比较接近，请读者注意比较，与之对应的 ICCAVR 程序可以参考第 6 章的有关部分。

8.5.1 使用定时/计数器

AT90S8515 有一个 8 位定时/计数器 0 和一个 16 位的定时/计数器 1。定时/计数器 1 除了具有定时和计数两种功能外，还提供输出比较匹配、输入捕获和 PWM 三种功能。IAR 与 ICCAVR 对定时/计数器的操作基本相同，这里只给出定时/计数器 0 的基本用法，至于定时/计数器 1 的用法可参考第 4 章 ICCAVR 的相关部分。

1. 使用查询方式操作定时器

前面定义过的宏：

```
#define SET_BIT(x, y)      (x|=(1<<y))
#define CLR_BIT(x, y)      (x&=~(1<<y))
#define GET_BIT(x, y)      (x&(1<<y))
```

查询方式的源程序如下：

```
/** 查询方式使用 TIMER0 定时器。
** 当 TIFR 寄存器溢出时，变量 LED 的值增加 1，并且输出到 PORTB 驱动发光 LED。*/
#include "io8515.h"                    /* 使用 AT90S8515 */
unsigned char led;
void main( void )
    {
    DDRB=0xFF;                         /* PORTB 所有端口设为输出 */
    TCNT0=0;                           /* 定时/计数器 0 的初始值为 0 */
    TCCR0=5;                           /* 使用 1024 分频( ck/1024) */
    led=0;
    for (;;)
         {
         /* 循环检测 TIFR 寄存器中溢出标志位是否有效 */
         while(!GET_BIT(TIFR,TOV0));
         PORTB=led;                    /* 输出 led 的值到 PORTB */
```

```
        led++;
        if (led==255)
        led=0;
        /*写逻辑"1"到 TIFR 的 TOV0 位来清楚溢出标志位 TOV0*/
        SET_BIT(TIFR,TOV0);
        }
    }
```

2. 中断方式

实际上，定时器的中断方式操作更加普遍。还是使用上面的例子，IAR 中断函数声明使用“#pragma vector=”伪指令，源程序如下：

```
/** 使用中断方式使用 TIMER0 定时器。
** 当 TIFR 寄存器溢出时，变量 led 的值增加 1，并且输出到 PORTB 驱动发光 led。*/
#include "io8515.h"                    /* 使用 AT90S8515 */
unsigned char led;
#pragma vector=TIMER0_OVF0_vect        /* 中断向量在 io8515.h 中有定义 */
__interrupt void timer0_isr(void)      /* 定义定时器 0 溢出中断服务程序 */
    {
    PORTB=led;                         /* 输出 led 的值到 PORTB */
    led++;
    if (led==255)
    led=0;
    TCNT0=0  ;                         /* 重新加载定时器寄存器 */
    }
void main(void)
    {
    DDRB=0xFF;                         /* PORTB 所有端口设为输出 */
    SET_BIT(TIMSK,TOIE0);              /* 允许 TIMER0 溢出中断 */
    TCNT0=0;                           /* 定时/计数器 0 的初始值为 0 */
    TCCR0=5;                           /* 使用 1024 分频( ck/1024) */
    led=0;
    SEI();                             /* 全局中断允许 */
    for (;;){}                         /* 循环等待中断处理 */
    }
```

8.5.2　使用 UART

IAR 没有提供 UART 的库函数，在这里给出使用 UART 以查询方式收发数据的源程序。

1. 查询方式发送数据

```
/** UART 发送的测试程序
** 使用查询方式发送一串字符串到 UART*/
#include "io8515.h"
/* 声明一个存放在程序空间的字符串数组 a[] */
const char a[] ="This is a test message!";
unsigned char pos=0;
void sendchar(unsigned char c)
    {
    /*查询方式发送数据 */
    UDR=c;
    while(!GET_BIT(USR,TXC));
    SET_BIT(USR,TXC);
    }
int main(void)
    {
    SET_BIT(UCR,TXEN);                  /* 允许 UART 发送数据 */
    UBRR=25;                            /* 初始化波特率产生器 */
    CLI();                              /* 关闭全局中断 */
    while(pos<23)
        {
        send(a[pos]);                   /* 写一个字节到 UART */
        pos ++;
        }
    for(;;)
        ;
    }
```

2. 查询方式接收数据

```
/** UART 接收的测试程序
** 使用查询方式接收一个字符并送到 PORTB 驱动发光 LED */
#include "io8515.h"
unsigned char  recvchar(void)
    {
    while(!GET_BIT(USR,RXC))
        ;
    return UDR;
    }
void main(void)
    {
    DDRB=0xFF;
    SET_BIT(UCR,RXEN);                  /* UART 接收允许 */
```

```
    UBRR=25;                          /* 初始化波特率寄存器，9600bps */
    for(;;)
        {
        /* 从 UART 读取一个字节输出到 PORTB 驱动发光 LED */
        PORTB=recvchar();
        }
}
```

说明：上例中的发送和接收程序仅仅只是示例，一般不能应用在实际的工程中。在实际使用中，应加上延时判断，当在一定时间内发送失败或接收失败，应退出发送或接收程序，防止由于外部线路的故障而导致死循环。

8.5.3　使用 EEPROM

IAR 提供两个库函数来读写 EEPROM(在 inavr.h 文件中)：

```
#define __EEPUT(ADR,VAL) (*((unsigned char __eeprom *)ADR)=VAL)
#define __EEGET(VAR, ADR) (VAR=*((unsigned char __eeprom *)ADR))
```

其中，ADR 是需要访问的 EEPROM 8 位地址，VAR 是读/写的 8 位字节数据。
举例如下：

```
/* 利用 IAR 标准 I/O 函数来读/写 E2PROM */
#include "io8515.h"
#include "ina90.h"                    /* 包含 inavr.h 文件 */
void main(void)
    {
    char temp=0x55;
    _EEPUT(0x10,temp);                /* 写 E2PROM 地址 0x10 */
    _EEGET(temp,0x10);                /* 读 E2PROM 地址 0x10 */
    for(;;)
        ;
    }
```

8.5.4　使用数据空间绝对地址

下面介绍两种在 IAR C 语言中访问外部绝对地址空间的方法。以操作某 LCD 为例，先定义 LCD 的数据寄存器(LCMDW)和控制寄存器(LCMCW)。

1．使用指针变量赋值

定义两个指针变量，然后再分别让它们指向数据寄存器和控制寄存器的物理地址。

```
unsigned char *LCMCW,*LCMDW;      /* 定义两个指针变量 */
/* LCMCW 指向 LCD 控制寄存器的物理地址 */
LCMCW=(unsigned char *)0xf100;
/* LCMDW 指向 LCD 数据寄存器的物理地址 */
LCMDW=(unsigned char *)0xf200;
```

在程序中就可以这样使用上面定义的两个寄存器：

```
#include "io8515.h"
void main(void)
   {
   unsigned char cmd,chr;            /*定义局部变量 */
   unsigned char cbyte;
   cmd=0x80;
   chr=0x52;
   *LCMCW=cmd;                       /* 写 LCD 控制寄存器 */
   *LCMDW=chr;                       /* 写 LCD 数据寄存器 */
   if((*LCMCW & 0x03)==0x03)         /* 读取并判断状态 */
   cbyte = *LCMDW;                   /* 读 LCD 数据寄存器 */
   for(;;)
       ;
   }
```

2. 直接定义绝对地址

也可以使用“__no_init”伪指令来定义两个指针常量，并且让它们分别指向数据寄存器和控制寄存器。

```
/* LCMCW 指向 LCD 控制寄存器的物理地址 */
__no_init static uchar LCMCW @ 0xF100;
/* LCMDW 指向 LCD 数据寄存器的物理地址 */
__no_init static uchar LCMDW @ 0xF000;
```

在程序中可以有相同的调用方式：

```
#include "io8515.h"
void main(void)
   {
   unsigned char cmd,chr;            /*定义局部变量 */
   unsigned char cbyte;
```

```
cmd=0x80;
chr=0x52;
*LCMCW=cmd;                          /* 写 LCD 控制寄存器 */
*LCMDW=chr;                          /* 写 LCD 数据寄存器 */
if((*LCMCW & 0x03)==0x03)            /* 读取并判断状态 */
cbyte=*LCMDW;                        /* 读 LCD 数据寄存器 */
for(;;)
    ;
}
```

提示：以上两种方法 C 语言应用是一样的，但是观察他们所生成的汇编代码，就会发现两种方法将得到不同的效率。仅以 if((*LCMCW & 0x03)==0x03)读取判断为例进行比较：

```
;方法 1 生成的代码：
; if((*LCMCW & 0x03)==0x03) /* 读取并判断状态 */
LDI R26,LOW(LCMCW)
LDI R27,(LCMCW) >> 8
LD R30,X+
LD R31,X
SBIW R27:R26,1
LD R17,Z
ANDI R17,0x03
CPI R17,3
;方法 2 生成的代码：
;if((*LCMCW & 0x03)==0x03) /* 读取并判断状态 */
LDS R17,LCMCW
ANDI R17,0x03
CPI R17,3
```

比较发现：方法 1 使用指针变量将占用 ram 保存，每次都要首先寻址取数，然后再赋值给端口寄存器 Z；而方法 2 直接使用端口地址，将其作为一个常数(地址)处理，简洁明了，当然速度也就快了很多。

8.6　使用 IAR 模拟 I^2C 主模式程序实例

本节给出 IAR 实现的主模式下的 I^2C 程序，使用 AT90S8515，4MHz 晶振，如图 8.14 所示，在这里不再讲述 I^2C 的原理，只给出用 IAR C 操作 I^2C 的操作源代码。

```
/* ************************************************************
**  I2C 源程序
**  使用 AT90S8515 模拟 I2C 主模式，4MHz 晶振
**  PD.4 定义为 SDA；PD.5 定义为 SCL；总线需要 2K 的上拉电阻
**  I2C 协议需要各函数：
**  起始位：    void I2C_Start(void);
**  停止位：    void I2C_Stop(void);
**  读一个字节：unsigned char I2C_Rite(unsigned char wb);
**  写一个字节：unsigned char I2C_Read(void);
**  数据应答：  void I2C_Ack(void);
************************************************************/
```

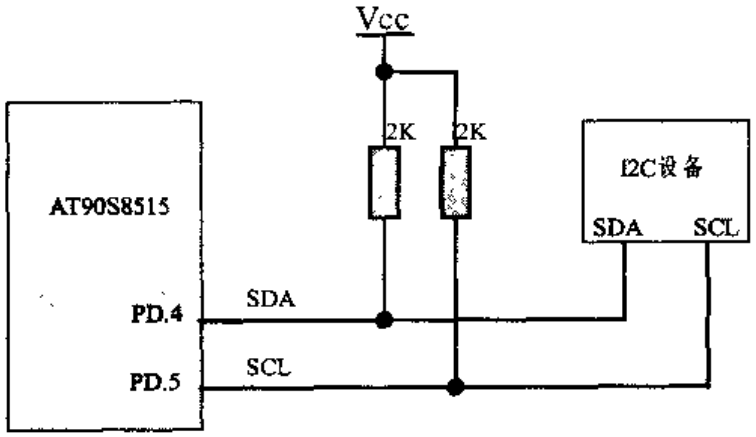


图 8.14　AT90S8515 与 I2C 设备连接

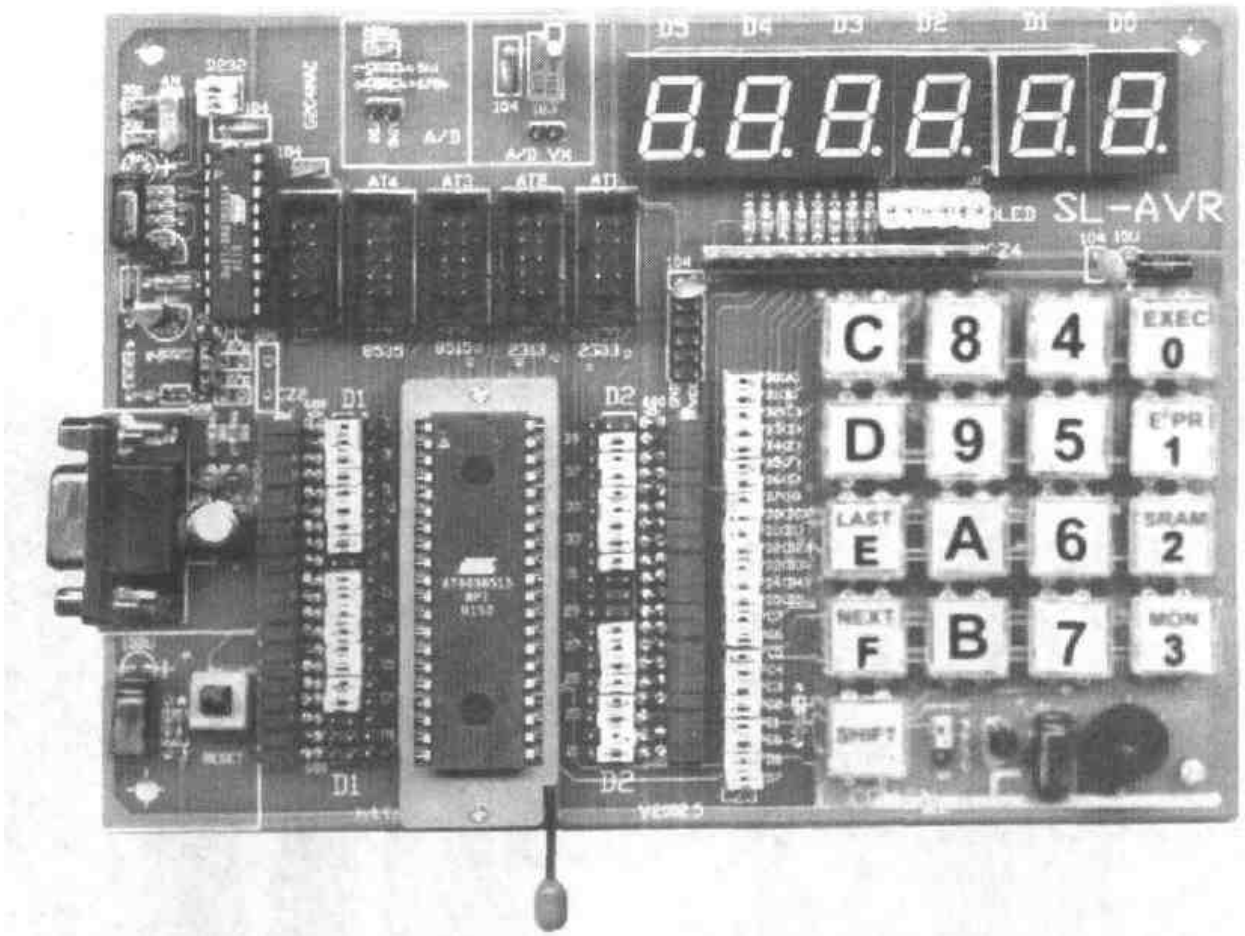
```
#include "io8515.h"
#define I2C_TIMER 100 /* I2C 延时常数，可以根据不同的晶振频率进行调整 */
#define SDA  PD4              /* 定义 PD.4 为 SDA */
#define SCL   PD5             /* 定义 PD.5 为 SCL */
#define SET_SDA      PORTD|=(1<<SDA)          /* 置 SDA 线为高电平 */
#define CLS_SDA      PORTD&=~(1<<SDA)         /* 清 SDA 线为低电平 */
#define SET_SCL      PORTD|=(1<<SCL)          /* 置 SCL 线为高电平 */
#define CLS_SCL      PORTD&=~(1<<SCL)         /* 清 SCL 线为低电平 */
#define ACK_FAIL     PIND&(1<<SDA)
#define SDA_HIGH     PIND&(1<<SDA)
/* 延时函数 */
void delay(unsigned int dt)
    {
    while(dt)
        {
        asm("nop");
        dt--;
        asm("nop");
```

```
        }
    }
/* 拉高时钟SCL电平，模拟时钟上升跳变，注意电平变化前后的延时 */
void clock_high(void)
    {
    delay(I2C_TIMER/2);                    /* 延时I2C_TIMER/2 */
    SET_SCL;
    delay(I2C_TIMER/2);                    /* 延时I2C_TIMER/2 */
    }
/* 拉低时钟SCL电平，模拟时钟下降跳变，注意电平变化前后的延时 */
void clock_low(void)
    {
    delay(I2C_TIMER/2);                    /* 延时I2C_TIMER/2 */
    CLS_SCL;
    delay(I2C_TIMER/2);                    /* 延时I2C_TIMER/2 */
    }
/* I2C起始位 */
void I2C_Start(void)
    {
    clock_low();
    SET_SDA;                               /*置数据线SDA为高电平*/
    delay(I2C_TIMER*2);                    /* 延时I2C_TIMER*2 */
    clock_high();
    delay(I2C_TIMER*2);                    /* 延时I2C_TIMER*2  */
    CLS_SDA;
    delay(I2C_TIMER*4);                    /* 延时I2C_TIMER*4  */
    clock_low();
    delay(I2C_TIMER*4);                    /* 延时I2C_TIMER*4  */
    }
/* I2C停止位 */
void I2C_Stop(void)
    {
    clock_low();
    CLS_SDA;                               /* 设置数据线SDA为低电平 */
    delay(I2C_TIMER*2);                    /* 延时I2C_TIMER*2  */
    clock_high();
    delay(I2C_TIMER*4);                    /* 延时I2C_TIMER*4  */
    SET_SDA;
    delay(I2C_TIMER*2);                    /* 延时I2C_TIMER*2  */
    }
/* 向I2C设备写一个字节，返回值为"0"表示操作成功，返回非零值表示操作失败 */
unsigned char I2C_Rite(unsigned char wb)
```

```
    {
    unsigned char i;
    for(i=0;i<8;i++)                    /*发送一个字节，高位在前 */
         {
         clock_low();
         if( wb&0x80 )
         /* 如果该位为"1"，设置数据线 SDA 为高电平 */
            SET_SDA;
         else
         /* 如果该位为"0"，设置数据线 SDA 为低电平 */
            CLS_SDA;
         clock_high();
         wb=wb<<1;                      /* 向左移位 */
         }
    /* 等待 I2C 设备的应答 */
    clock_low();
    delay(4*I2C_TIMER);                 /* 延时 I2C_TIMER*4  */
    clock_high();
    while(i>1)                          /* 利用上面的 i，从 8 开始 */
        {
        delay(I2C_TIMER);
        if(ACK_FAIL) i--;               /* 检测 I2C 器件的的数据应答 */
        else i=0;                       /* 如果成功，返回"0"值 */
        }
    clock_low();
    return i;
    }
/* 向 I2C 设备发送一个读取数据的应答信号*/
void I2C_Ack(void)
    {
    clock_low();
    CLS_SDA;
    delay(I2C_TIMER);
    clock_high();
    delay(4*I2C_TIMER);
    clock_low();
    SET_SDA;
    }
/* 从 I2C 设备读取一个字节，高位(MSB)在前 */
unsigned char I2C_Read(void)
    {
    unsigned char i;
```

```
unsigned char rByte = 0;
for(i=0;i<8;i++)
      {
      clock_low();
      clock_high();
      if(SDA_HIGH)                    /* 判断数据线 SDA 的电平 */
      rByte|=1<<(7-i);                /* 如果接收到"1"，设置 rByte 中的相应位 */
      clock_low();
      }
return rByte;
}
```

附录 A　SL-AVR 开发实验器简介



五合一的 SL-AVR 嵌入式单片机开发下载实验器(相当于 AVR 编程器+模拟仿真器+单片机组态开发+实验器+科研样机)提供的几十个实用实验程序，也可改变硬件接口，修改程序，实现原程序的功能。这对大专院校学生发挥其创造性思维及动手能力特别有用，可改善我国传统教育下的高分低能的状况。对初学 AVR 单片机的设计者，可以先使用 SL-AVR 实验器，暂时节省购买较昂贵的实时仿真器及万用编程器的费用，本实验器也可作科研样机使用。

SL-AVR 编程开发实验器硬件采用模块化设计，便于用户灵活组成科研项目所需的硬件结构，硬件有：

1. RS232 通信接口；
2. 串行下载监控；
3. *DIP8-DIP40* 通用锁紧插座；

4. 通用锁紧插座的每个端口均用短路块连接，实验器在出厂时各 I/O 端口均作为输出，用发光二极管(LED)显示器件引脚电平的高低，也改变短路块将该端口作输入或其他用途；

5．6 位 LED 数码管作输出显示；

6．配一个 2×16 点阵的 LCD 液晶显示器；

7．有一个 17 键的键盘扫描输入；

8．模拟比较输入电路；

9．音响电路、单片机复位电路、模拟电压输入电路等；

随机附 120×170mm 万用实验板及一片 AT90S8515 芯片，SL-AVR 适用于所有具有串行 ISP 下载编程功能的 AVR 单片机，用户板上的器件无需拆下即可编程，同时还可做 AVR 单片机的 I/O 口、A/D、D/A、LED、LCD、键盘输入、音频输出和模拟比较等开发实验。

10．具有单片机组态开发功能(另配组态监控及软件)；这样该机具有互动功能,从单片机上可指挥 PC 屏幕上的控件动作，PC 屏幕的控件可指挥单片机上的器件动作。他们又可独立工作，可在 PC 屏幕上自由组态设计，也可在单片机板上自由开发、下载、实验，可以生动形象直观地了解单片机 I/O 口的基本功能、I/O 口的扩展功能及单片机组态开发的复杂应用。

提供功能强大的 AVRstudio 集成开发环境(IDE)，包括以下内容：

1．AVR Assembler 编译器；

2．AVRstudio 调试功能，支持 C 源代码级调试；

3．AVR Prog 串行和并行下载功能；

4．JTAG ICE 仿真等功能。

新版 SL-AVR 对针对旧版的 SLAVR 缺陷进行以下改进：

1．下载插座 AT1、AT4 改为 DC10 插座，除了可以使用开发实验器配套的串行下载方式编程外，也可使用双龙的 ISP 并行高速下载线对大容量器件下载编程；

2．本机工作晶振可插拔更换，便于用户超频、降频实验；

SL-AVR 硬件连接说明：

1．D232：D232 通信线短路块，插上短路块，接到 AT90S1200 的 UART 端口，拔出短路块，可从 T、R 端口用插针线接到单片机通信口，可做单片机异步通信 UART、主从同步通信 SPI 或 ISP 下载通信；

2．AT90S1200：SL-AVR 开发实验器监控芯片；

3．ISP：SL 系列 DC10 的(ISP)插座，即 AVR 单片机的下载信号插座。本开发实验器配一片 AT90S8515 器件，绝大多数实验使用该器件，硬件(用短路块)连接出厂时也按该器件连接,其他器件作为选购件；

ISP 下载插座：如图 A.1 所示，引脚功能分别为 VCC、GND、XTAL2、XTAL1、MOSI、/RESET、SCK、MISO。随机附有一条 10 线信号线。由用户接插到对应 AVR 单片机(AT1-AT4)的信号脚上。ISP 也可直接连到用户板作 AVR 单片机(或用转接线)的串行下载编程用，如用户板有晶振，则 XTAL1 和 XTAL2 信号线无需接出。

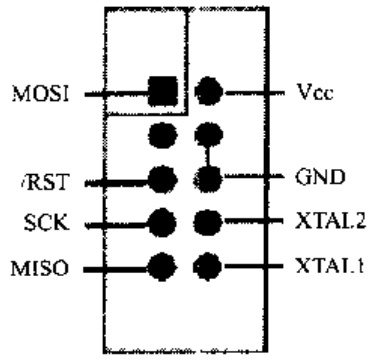


图 A.1 ISP 下载插座

4．有关 AVR 单片机的信号脚插座说明：

(1) AT1 插针座为 AT90S4433/AT90S2333/ATmega8 ISP 下载信号线插座；

(2) AT2 插针座为 AT90S1200/AT90S2313 ISP 下载信号线插座；

(3) AT3 插针座为 AT90S4414/AT90S8515/AT89S8252/AT89S53 ISP 下载信号线插座；

(4) AT4 插针座为 AT90S4443/AT90S8535 ISP 下载信号线插座；

(5) 如用于其他 AVR 单片机，可用转接线接到相应芯片的下载信号引脚上；

5．CZ1：电源及通信下载插座，电源线为地及+5V，通讯电缆一头接 CZ1，另一头接计算机 RS232 插座；

ERPOW：红色 LED 作电源指示；

BUSY：绿色 LED 作下载通信工作忙指示；

6．单片机复位按钮(RST)：作为人工强制启动复位用，在下载结束或开机通电时程序都会自动复位；

7．LED 发光二极管：低电平有效，作 I/O 口输出电平指示；

8．D1 和 D2 短路块，使单片机引脚作输出或输入用。插上短路块引脚作输出，用 LED 显示高低电平(低电平 LED 灯亮)；不插短路块，器件引脚可作输入用或作其他用途，如电源引线等；

警告：

(1) D1、D2 短路块中如有对应器件的晶振脚或复位脚，应拔出插短路块！

(2) AT90S4433/AT2333/AT90S4434/AT8535/ATmega103 等器件下载时把 DLED 短路块拔出，因为这些器件晶振脚或复位脚与 LED 相连，电平被 LED 电阻拉低了，会造成下载出错！

9．U4：无应力下载插座，可放置 AVR 单片机四种 DIP 封装(DIP8/20/28/40)器件下载用，放置器件时缺口应向上，与插座底部对齐；

10．CZ3 接线排针：为 AT90S8515 对应引脚 PB0-PB7、PD0-PD5、PC7-PC0，插上短路块接到键盘(17 键)和 LED/LCD 显示器，拔出短路块，用附机配的接插线可把键盘与显示改为其他 I/O 口，也可改用其他器件(如 AT90S2313/AT90S8535 等)，用接插线连接到器件相应引脚上(D1、D2 短路块处)；

11．SPEAKER：讯响器输入端，接相应器件的右下脚(5/11/15/21)，插上短路块，与 AT90S8515 接通的引脚相连通，也可用接插线接到单片机任何 I/O 脚上作音响输出；

12．无源音响器：由单片机控制发出声音；

13．17 键键盘模块：接 AT90S8515 的 PC 口，16 个数字键，另一个 SHIFT 换档键，SHIFT 接 AT90S8515 的 PD7，当先按下 SHIFT 键，再按数字键，即实现数字键上的命令功能，当然根据程序的要求，这些按键的功能可以重新定义；

14．LED 模块：由 6 只数码管组成，D5-D2 作地址，D1-D0 作数据，数码管字位口对应接 AT90S8515 的 PD5-PD0 口，数码管的字形(abcdefgh)对应接 AT90S8515 的 PB7-PB0 口，见后面的 CZ3 接线表；

15．DLED 短路块：LED 位线，用 DLED 短路块连接，断开短路块，位线也可作它用途。注意：在下载 AT90S4433/AT90S8535/ATmega103 等芯片时，应插出短路块！

16．CZ4 接线座：为 2 行×16 字 LCD 液晶显示模块插座，用 AT90S8515 的 PB、PD 口；

17．A/D VX：多圈电位器作为模拟信号输入用，两边分别接上 VCC、GND，中间头(VX)用连接线接到单片机作模拟信号输入脚(如 AIN1)；

18．A/D：为片内模拟比较器作 A/D 转换外部元件电路(最好改为电阻精度为 1%，电容精度为 5%，所接阻值仅供参考)，AIN0、PD2 为接线端；

19．AT1-AT4：对应器件下载插座，为了便于用户能使用自己工作样机相一致的晶振及提高 SL-AVR 工作可靠性，特别对晶振设置作了改进，在 SL-AVR 工作时，随机所附晶振接插小印板(含 8M 晶振)，应插在对应器件的晶振引脚上(对应器件旁的插针)，晶振数值可根据实际工作改变(如超频降频等)。

DIP40 插座接线图，如下表所示：

接 CZ3	DLED	DIP--40								接 CZ3
PB0--a		1	AVR AT90S 系列 U4 DIP 引脚排列图					40		
PB1--b		2						39		
PB2--c		3						38		
PB3--d		4						37		
PB4--e		5						36		
PB5--f		6	DIP--28					35		
PB6--g		7	1	蓝色框字为复位脚短路块位置，紫色为 DLED 短路块位置，会影响器件下载！			28	34		
PB7--h		8	2				27	33		
		9	3				26	32		
PD0--D0	D0	10	4	DIP--20			25	31		
PD1--D1	D1	11	5	1	红色字为晶振脚短路块位置	20	24	30		
PD2--D2	D2	12	6	2		19	23	29	PC7—键线 7	
PD3--D3	D3	13	7	3		18	22	28	PC6—键线 6	
PD4--D4	D4	14	8	4		17	21	27	PC5—键线 5	
PD5--D5	D5	15	9	5		16	20	26	PC4—键线 4	
PD6--D6		16	10	6	DIP--8	15	19	25	PC3—键线 3	
PD7--D7		17	11	7	1	8	14	18	24	PC2—键线 2
		18	12	8	2	7	13	17	23	PC1—键线 1
		19	13	9	3	6	12	16	22	PC0—键线 0
		20	14	10	4	5	11	15	21	

注意：CZ3 是专为 AT90S8515 而设计的，只需插上短路块，LED、LCD、键盘就能工作，方便快捷。而用其他器件做实验时，应改变短路块或用短接线连接：

(1) 如 DIP20 器件的 AT90S1200/AT90S2313 下载时，DIP-20-1/-4/-5 脚和 CZ3-PD4 及 CZ3-PD5 短路块应拔出；

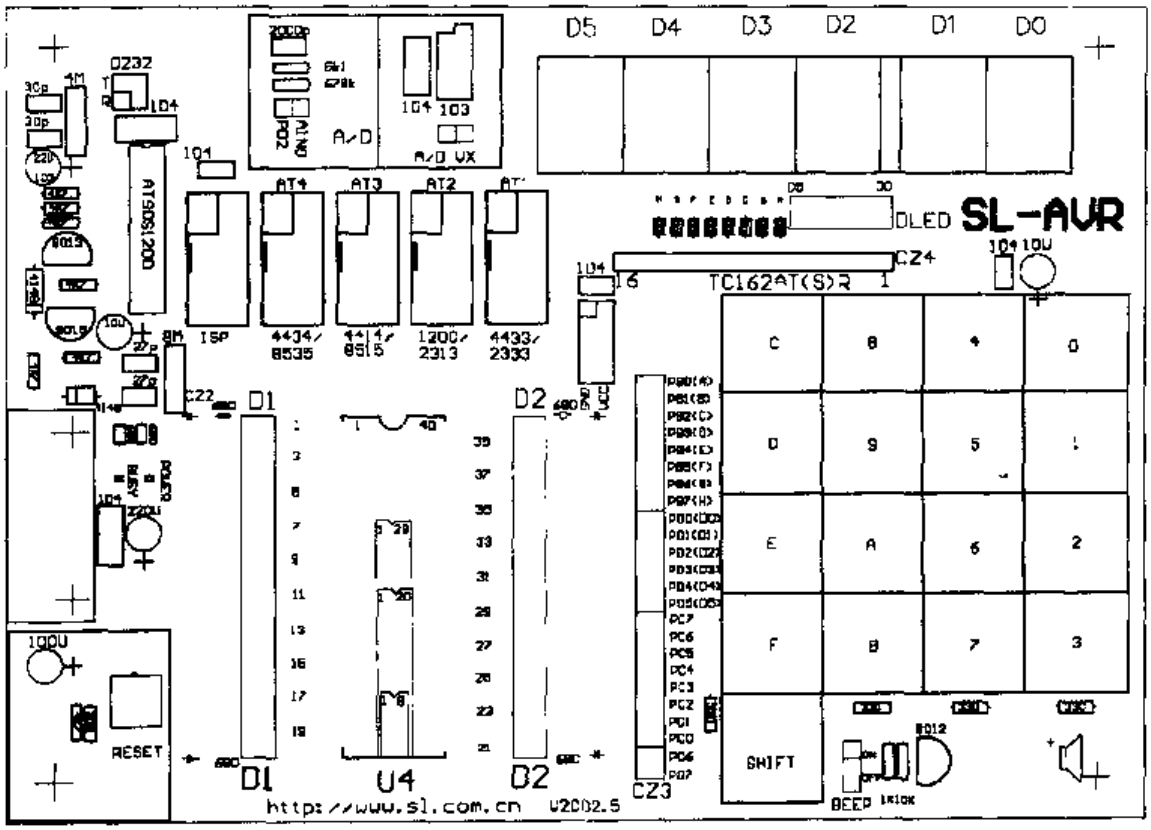
(2) 如 DIP28 器件的 AT90S4433/AT90S2333 下载时，DIP-28-1/-9/-10 脚 CZ3-PB6/PD5/PD6 短路块应拔出；

(3) 如 DIP8 器件的 AT90S2323/Attiny10~12 下载时，DIP-8-1/-2/-3 脚和 CZ3-PD7 短路块应拔出；

(4) 如 DIP40 器件的 AT90S4414/AT90S8515 下载时，DIP-40-9/-18/-19 脚短路块应拔出；

(5) 如 DIP40 器件的 AT90S4434/AT90S8535 下载时，DIP-40-9/-12/-13 脚和 CZ3-PD2/PD3 短路块应拔出；

SL-AVR 元器件排布示意图



SL-AVR 的原理图：