从 ATmega48/88/168 开始

IT IS ONLY A DEMO EDITION

宁可被读者认为是没有技术含量的简易入门读本，也不要被认作是技术高深但看不懂的『经典』。

中国电力出版社

前　　言

感谢您翻开这本与众不同的单片机书籍。

在本书中您无法找到下列内容：

1. 对英文版数据手册的照抄照翻。
2. 如何使用汇编语言编写单片机软件。
3. 对单片机各种资源事无巨细的讲解。

在其他书籍中您不一定能找到下列内容：

1. 以一种单片机为依托，学习各种单片机的通用法则。
2. 用生动的语言讲述各种原理。
3. 如何自己动手搭建单片机实验室。
4. 大量的应用实例，完整的程序代码。
5. 嵌入式系统开发和调试的工程思想。
6. 填补学校教学与工作需求之间的能力训练空白。

IT IS ONLY A DEMO EDITION

其他几个需要说明的问题：

1. 随书光盘在本书的阅读中非常重要，请注意保管。
2. 任何一种单片机都可以作为入门学习的机型，只是我们推荐 **AVR** 罢了。
3. 以 **C** 语言入门只是加速学习过程，入门后还需补习汇编。
4. 书中提及的各公司及其产品名称均属引用，作者不拥有其他权利。

感谢：
耿德根老师发起本书的编写，提供了开发器和样片并提供技术指导
西南科技大学机器人小组对实例进行的测试和回馈
张华、王姮、吴建、胡天涟、夏显峰在本书策划期间提出的宝贵意见
刘宏伟老师参与了本书的编写工作，对其辛勤劳动表示感谢
我的合作者王卓然的辛勤工作
韩名明同学参与的校对及内容测试工作
我们的父母对我们的理解和支持
网友彩虹、杨涛的测试回馈
出版社全体工作人员的大力支持
21ICBBS 和 **OurAVR** 上广大网友的支持和鼓励。

衷心希望您能够喜爱本书，并期待您的宝贵建议。

江海波
2007 年 **9** 月 **5** 日　于　四川·成都

第一篇

学会阅读 *Datasheet*

深入开发环境

IT IS ONLY A DEMO EDITION

对初学者来说，阅读英文资料确实有一定的难度。事实上，即便是有相当英文基础的人，要看懂芯片的DATASHEET这种专业性极强的资料仍需要一个过程。勤翻词典、注重积累、坚持阅读，相信一段时间以后，你就可以轻松地阅读DATASHEET等专业英文资料了。

实例 2 AVR最小系统DIY

大家知道，计算机学科是高度重视实际应用的，如果没有实践环节，一切学习都将变成纸上谈兵。如果您手中没有**ATmega48/88/168**的单片机实验板，也没有关系，我们一起来**DIY**一个，上面元件的总成本不到**100**元。另外一个好消息是，由于**ATmega8**的主要引脚与**ATmega48/88/168**是兼容的，这个实验板可以直接用于**ATmega8**的学习。

第 1 步 实验板准备工作

由于电路图太大，我们在这里就不给出他的全貌了，请读者在本书所附的光盘中阅读他。

—— DA895 注

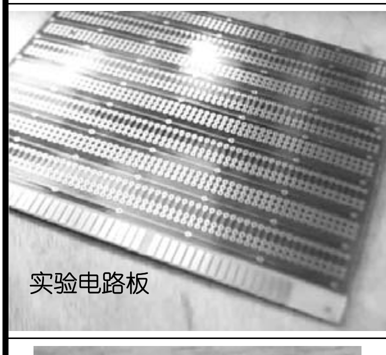
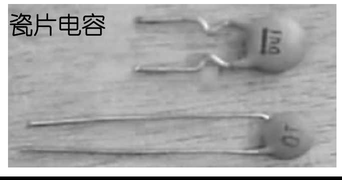
[准备工作]

先来熟悉一下**ATmega48/88/168**单片机实验板电路的基本组成：他由电源、单片机小系统、**ISP**接口、蜂鸣器、时钟发生器、**RS232**通讯电平转换、**I2C**接口的**E2PROM**共**6**个单元组成。为了方便起见，键盘和显示将做在另外的电路板上。

现在来看一下需要准备的元件和材料：

IT IS ONLY A DEMO EDITION

ATmega48/88/168单片机实验板电路原材料清单

	将**220V**市电转换成**9V-12V**的直流电，用于为实验板电路供电。注意输出的电流应大于**500mA**。输出插座的样式和极性见电源组装部分说明。也可以利用废旧游戏机（红白机）电源自己改装。
	大约**150x120mm**大小的实验电路板（也叫蛇目板）**1**张。注意他上面是有一部分铜箔走线的，不是光突突的焊盘，这种实验板使用起来要方便一些。
	**0.1uF 35V**瓷片电容**11**只
	0.01uF 16V瓷片电容**1**只
	20pF 16V瓷片电容**2**只

元件	规格
电解电容	**1000uF/35V CD11X**铝电解电容 **1**只
	470uF/16V CD11X铝电解电容 **1**只
	47uF/16V CD11X铝电解电容 **1**只
金膜电阻	**10Ohm w/4** 直插金膜电阻 **1**只
	10k w/4 直插金膜电阻 **4**只
	470Ohm w/4 直插金膜电阻 **23**只
集成电路	**ATmega48V-10PI**单片机 **1**只
	L7805CV 1只
	AT24C01A-10PU 1只
	M74HC04B1R 1只（可使用**CD4069**）
	MAX202ECPE 1只（可使用**MAX202CPP**）
蜂鸣器	**5V**单音蜂鸣器 **1**只。注意蜂鸣器分多种，这里使用接通电源后一直鸣叫的那种。
三极管	晶体三极管**9014C 1**只
二极管	整流二极管**1N4007 1**只
	开关二极管**1N4148 1**只
石英晶体	**8MHz**石英晶体 **1**只
电源插口	**5mm**电源插口 **1**个
开关	按钮开关 **1**个
LED	**5mm** 方形红色**LED 21**个
	5mm 圆形红色**LED 1**个
	5mm 圆形黄色**LED 1**个

IT IS ONLY A DEMO EDITION

图	说明
跳线帽	跳线帽若干。 这种跳线帽很便宜，可以直接买上**100**个。
集成电路插座	**DIP-8** 集成电路插座 **1**个
	DIP-14 集成电路插座 **1**个
	DIP-16 集成电路插座 **1**个
排针	**2.54mm**间距排针 **4-5**排
	2.54mm间距排针座 **2**排
	2.54mm间距圆形排针座 **2**排
导线	多股导线若干
线鼻子和热缩管	用来制作连接线的线鼻子和热缩套管若干
散热器	用于**7805**的散热器及紧固件
下载线插座	连接**ISP**下载线的插座

IT IS ONLY A DEMO EDITION

这些材料都是非常便宜并且容易购买到的，如果您在当地的电子元件市场买不到单片机，可以向代理商邮购。

好了，如果元器件准备好了，就让我们一起来搭建实验电路板吧！

第 2 步
规划电路板

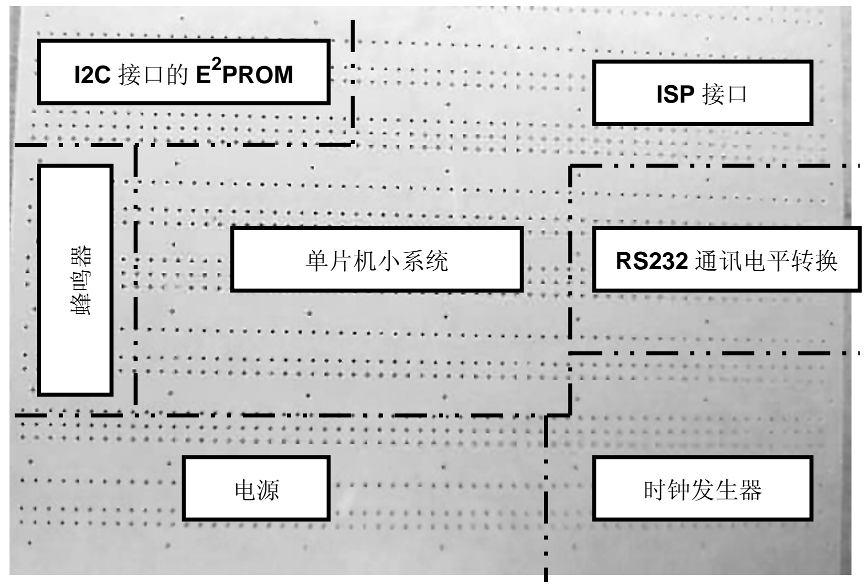
[组装顺序及规划]

前面已经说过，实验电路板由**6**部分组成，我们需要在实验电路板上规划出**6**个区域用于安装这些元器件，并且确定组装他们的先后顺序。

首先用记号笔在实验电路板上规划出各部分的分布情况，如**例图2.1**所示。本着“从电源入手、难易结合、边焊边测”的方法，焊接各部分的顺序为：

电源——单片机小系统及**ISP**接口——蜂鸣器——时钟发生

器——**RS232**通讯电平转换——**I2C**接口的**E²PROM**。



例图 2.1 实验电路板布局规划

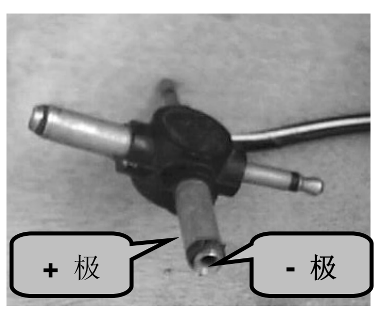
第 3 步
组装
调试

[电源的组装及调试]

电源部分电路图见**例图2.2**所示，其作用是将电源适配器输出的**9~12V**直流电压稳定成**5V**电压供各电路使用。**XS2**和**XS3**分别用于向其他电路板（例如我们后面要制作的键盘显示模块）提供**5V**电源和地。

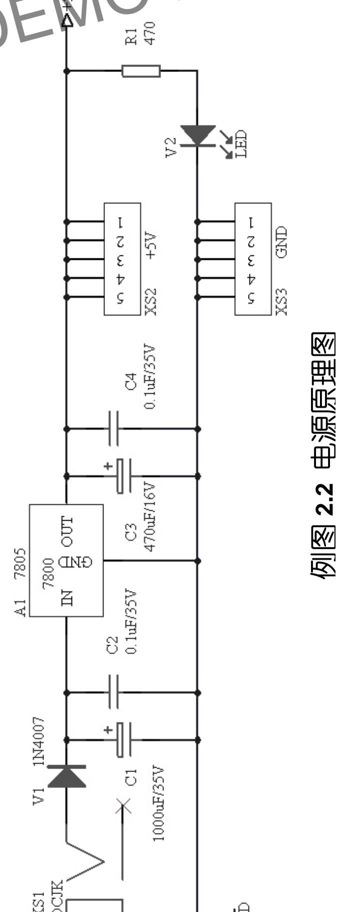
这个电路在设计上，加入了二极管**V1**，防止电源适配器极性不正确时烧毁实验板。**V2**用于指示电源是否正常工作。

用于我们实验板的电源适配器应该有**例图2.3**所示的输出接口和极性。



例图 2.3 电源适配器极性

焊接好电源部分的实验电路板参见**例图2.4**。由于使用的是实验电路板，我们并不要求大家的走线布局完全一样，可以根据自己理解进行操作。



例图 2.2 电源原理图

IT IS ONLY A DEMO EDITION

电源组装部分里，最容易出现错误的是电解电容极性装反。这种错误可能造成电容发热并导致爆炸，一定要仔细检查。

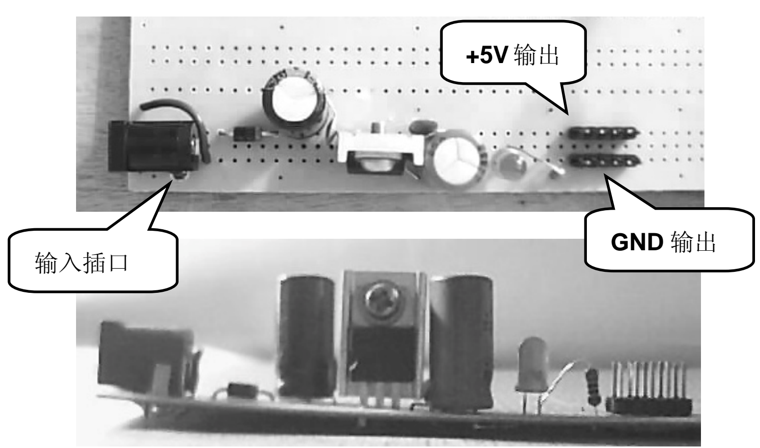
如果通电后 V2 不亮，电压表检测不到 5V 电压输出，应分部分检查是否有元件错装（例如 V1 极性焊反）或虚焊（例如 A1 的 OUT 端）。如果输出电压与输入电压相等，则应检查 A1 的 GND 端是否出现虚焊。

一般来说，只要组装上不出问题，电源部分的成功率非常高。

—— DA895 注

这里使用 5mm 圆形红色 LED 作为电源指示。

—— DA895 注



例图 2.4 电源部分-焊接完成后的效果

反复检查，确认无误后就可以将电源适配器连接到实验电源板上，开始调试电源部分。接通电源后，指示电源工作情况的发光二极管**V2**应该点亮，用万用表测试**XS2**、**XS3**两个输出点间的电压，应该为**5V**，误差应不超过**0.2V**。

[组装单片机小系统及 ISP 接口]

单片机小系统及**ISP**接口是实验板的核心部分，组装时一定要仔细。他的电路图见**例图2.5**。

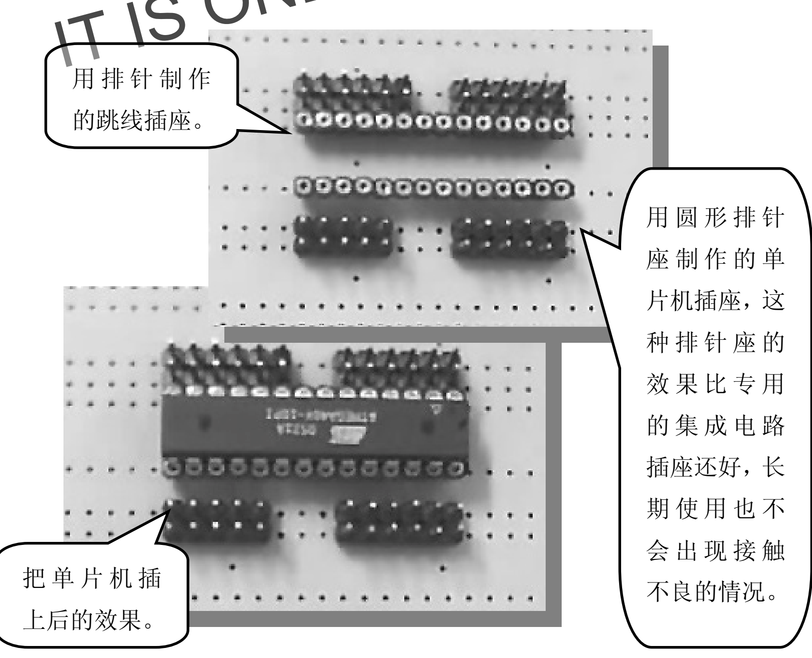
需要提醒的是，单片机是通过圆形排针插座插接在电路板上的，并且为了简洁起见，该图中我们没有画出各**I/O**口的跳线插座。读者可以参考**例图2.6**进行跳线的焊接。

IT IS ONLY A DEMO EDITION

只要排列整齐，实验电路板搭出来的电路也可以很考看哦。

半成品，拍照纪念，暗自窃喜下！

—— 傻孩子注

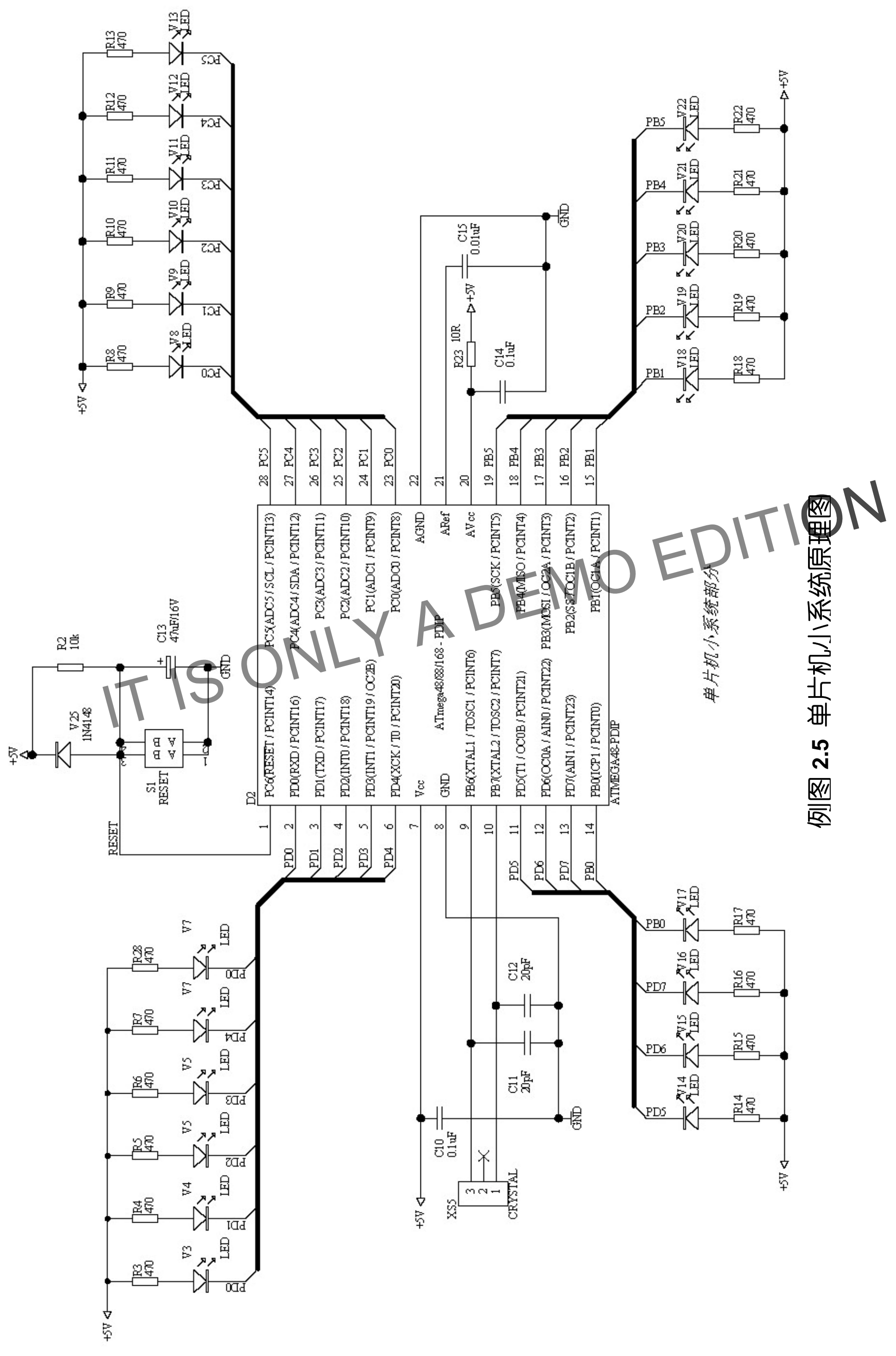


例图 2.6 小系统部分-焊接单片机插座及跳线

下面要焊接各个**I/O**口上的**LED**以及复位电路、石英晶体插座（**XS5**）、**A/D**电源滤波电路（**R23**、**C14**、**C15**）。焊接这部分时请特

别注意：

> 石英晶体插座XS5一定尽量靠近单片机XTAL1、XTAL2引脚，电容C11和C12接到地线的连接线应尽量的短，并且尽量使用独立的连接线连接到电源地的输出端。违反上面任何一个规则将可能导致石英晶体无法工作！



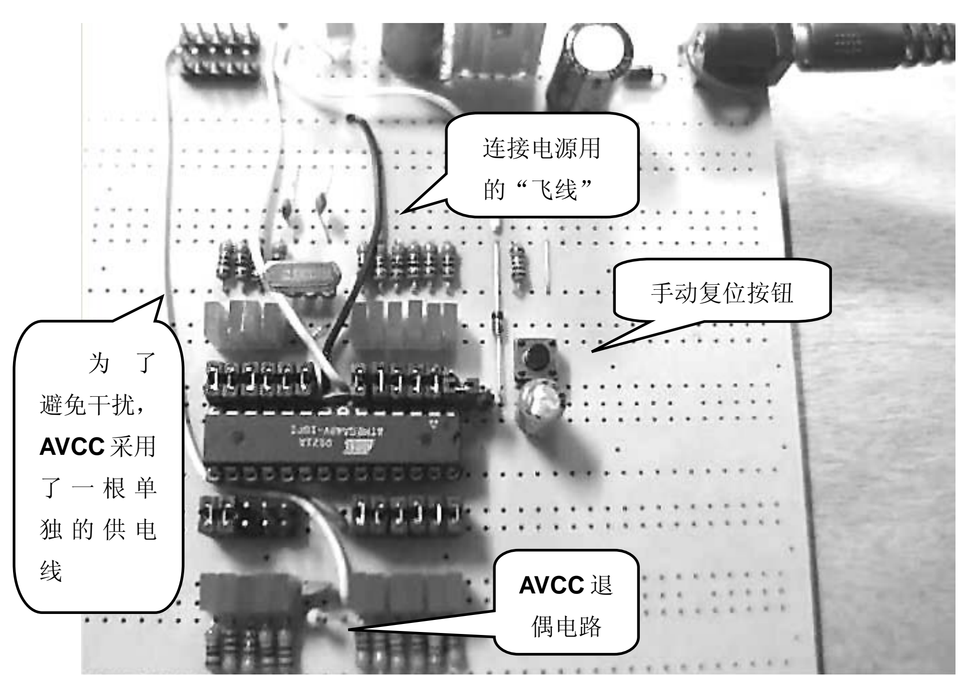
例图 **2.5** 单片机小系统原理图

IT IS ONLY A DEMO EDITION

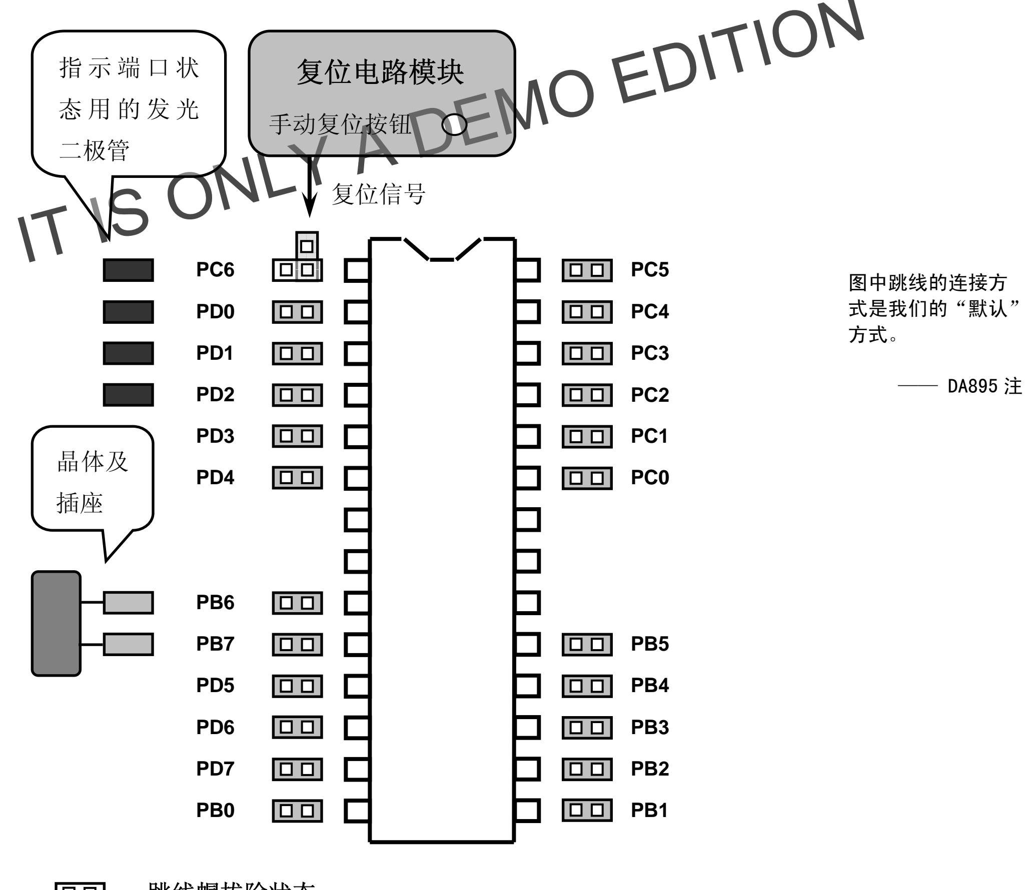
组装好单片机小系统后的实验板参考**例图2.7**所示。

图中的“跳线”部分，我们在电路图中没有画出来，他们的命名

方法及用途请参考例图**2.8**。



例图 **2.7** 小系统部分-焊接完成后的效果



IT IS ONLY A DEMO EDITION

图中跳线的连接方式是我们的“默认”方式。

—— DA895 注

例图 **2.8** 跳线帽的设置示意图

跳线的使用请按照下面的说明：

- 普通的**I/O**口：

I/O口输出状态，当需要观察其电平时，应将跳线帽插入，此时发光二极管（**LED**）在端口输出低电平时点亮。

这是一种非常有用的指示工具，当端口输出低频信号（数**Hz**）时，可以通过发光二极管直接看到电平的高低情况；当端口输出高频信号时，可以通过发光二极管的亮度定性地判断有无脉冲信号。

需要提醒的是，当端口作为输出时，由于**AVR**的端口负载能力很强，跳线帽的存在并不会影响输出信号；当端口作为输入时，需要考虑到其他芯片（例如我们后面将要使用的**MAX202**、**AT24C01A**等）的负载能力可能无法带动发光二极管这种重负载，一般情况下应将跳线帽拔除掉。

我们的实例 3 跑马灯 就是利用这些发光二极管来实现的。

—— DA895 注

- 带**A/D**输入的**I/O**口：

如果口线需要输入模拟电压，应该将该脚的跳线帽拔除，以免发光管这个“巨大”的负载影响输入信号的电平。

- **PC6**口：

PC6口与单片机的复位引脚**RESET**端复用。在默认状态下，应保持该引脚的跳线帽在**例图2.7**所示位置。

如果将该引脚当作端口**PC6**来使用时，则应将跳线帽插在与其他口线相同的位置。

严格来说，使用我们实验板的读者不可能将 RESET 端配置为 PC6 端口，因为这个设置熔丝只能在并行编程的模式下进行。

—— DA895 注

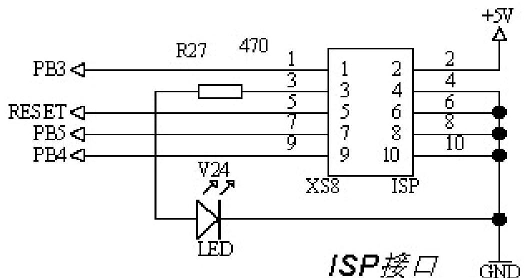
IT IS ONLY A DEMO EDITION

- **PB6**（**XTAL1**）**/ PB7**（**XTAL2**）口：

当使用内部**RC**振荡作为时钟源时，**XTAL1**和**ATAL2**端可以被配置成普通**I/O**口**PB6**和**PB7**来使用，此时应拔除这两个引脚上的跳线帽。靠近单片机一侧的排针可以用来引出这两个**I/O**口。

当使用外部石英晶体作为时钟源时，这两个跳线帽用于连接引脚和石英晶体，应予以保留。

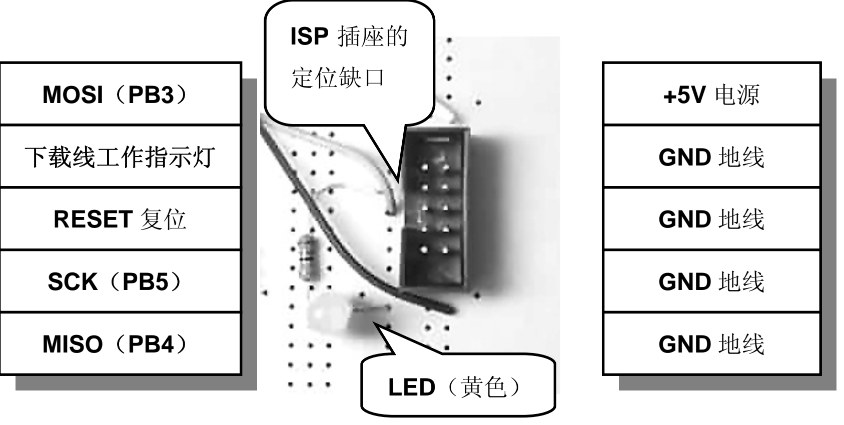
下面要重点介绍的是**ISP**下载线插座。他的电路图见**例图2.9**，效果图及引脚功能见**例图2.10**所示。这个插座的引脚顺序与**STK200/300**下载线是完全一致的，使用双龙下载线的读者也可以直接使用这个插座下载程序。



例图 2.9 ISP 下载线插座电路图

这里使用 5mm 圆形黄色 LED 作为下载线工作指示。

—— DA895 注



例图 2.10 ISP 下载线插座-焊接完成后的效果及引脚功能

图中框内的功能和ISP 插座引脚中的功能是对应的。

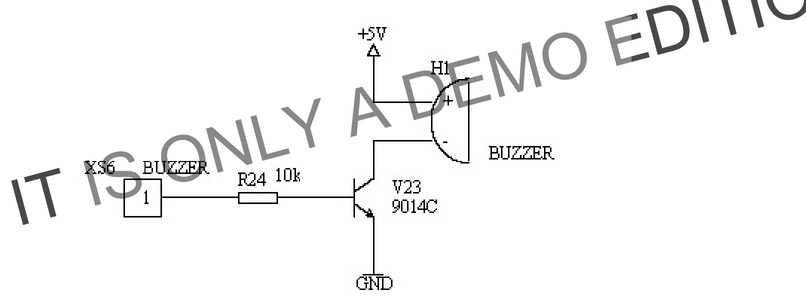
—— DA895 注

单片机小系统的调试需要借助软件进行，这就必须有下载线。为了保持章节的结构，下载线的制作放在了本实验的最后，读者可以继续按照书中的顺序组装实验板上的其他部分，也可以先跳到下载线组装部分，待验证了小系统搭建成功之后，带着喜悦的心情继续下面的工作。

[组装蜂鸣器单元]

蜂鸣器单元的电路图很简单，仅包含4个元件。

IT IS ONLY A DEMO EDITION



例图 2.11 蜂鸣器单元电路图

大家可能会问，在本书的实例中，并没有单独介绍驱动蜂鸣器的部分，为何在实验板上会安排这个单元呢？

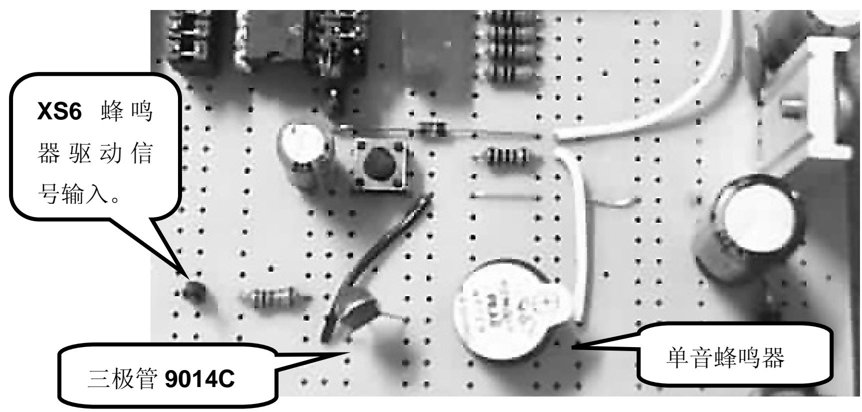
其实蜂鸣器是程序调试中一种非常有用的助手，特别是在没有仿真器，没有示波器的情况下，用来判断程序流向，估测脉冲频率等。

虽然本书的实例中没有必须要用蜂鸣器的地方，还是希望大家能够在实践中体会这种“调试工具”的用法。

图中的“BUZZER”是“蜂鸣器”的英文名称。

—— DA895 注

XS6是只有**1**根单独的插针，也是用排针来做的，效果图请参考**例图2.12**。



例图 2.12 蜂鸣器单元-焊接完成后的效果

蜂鸣器上粘贴了一张不干胶纸，是为了在清洗电路板时防止洗板水浸入到蜂鸣器内，一般应在焊接完成后将他撕掉，不过这张纸可以降低蜂器的音量，如果嫌他叫得太响，可以保留这张不干胶，不去撕毁。

—— DA895 注

蜂鸣器单元的调试非常简单，只需要用一根导线将**XS6**短接到电源（**+5V**）端，如果蜂鸣器鸣叫，即表示功能正确。

[组装时钟发生器单元]

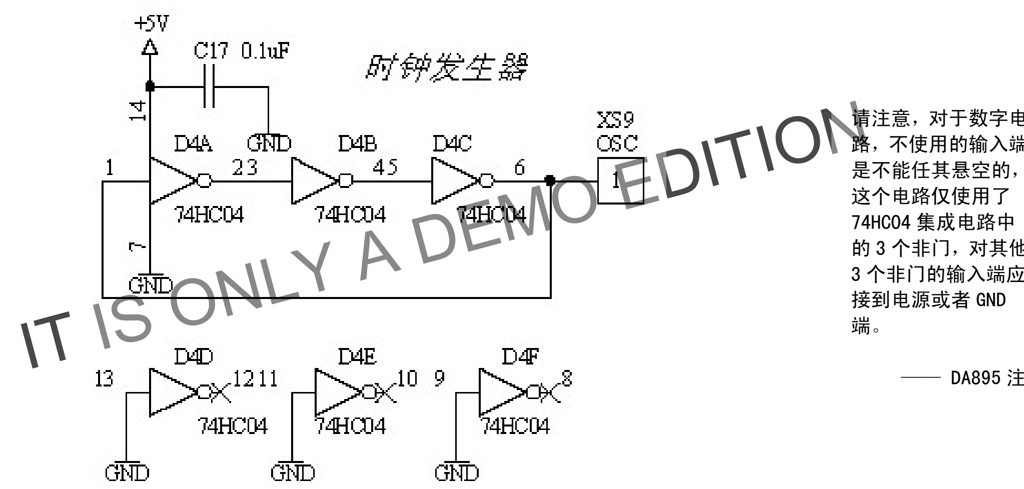
时钟发生器单元提供了一个简单的时钟源用于被错误配置为“外部时钟”模式的单片机解锁。详细的介绍请参看本篇**1.7**章节**“对误烧写为外部时钟模式的解锁方法”**的介绍。

这个时钟发生器实际上是一个由非门组成的“环形振荡器”，原理图见**例图2.13**。

时钟振荡由**XS9**输出，需要使用的时候将该端用导线连接到单片机**XTAL1**引脚即可。这个振荡器的频率约在数兆赫。

完成以后的效果图请参看**例图2.14**。

想看清输波形是比较麻烦的事情，必须使用示波器。如果没有条件，可以用万用表进行估测——如果万用表直流电压档测量**XS9**端时，得到的电压示数即不是电源电压（**5V**），也不是**GND**（**0V**），则说明组装成功，电路已经起振。

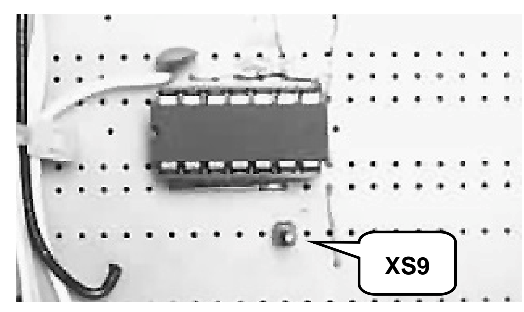


IT IS ONLY A DEMO EDITION

请注意，对于数字电路，不使用的输入端是不能任其悬空的，这个电路仅使用了 74HC04 集成电路中的 3 个非门，对其他 3 个非门的输入端应接到电源或者 GND 端。

—— DA895 注

例图 2.13 时钟发生器单元电路图



例图 2.14 时钟发生器单元-焊接完成后的效果

[组装 RS232 通讯电平转换单元]

对于没有接触过**RS232**通讯口的初学者，这部分电路可能相对陌生。这种电路的任务是在**5V**的数字电平和±**10V**左右的**RS232**电平之间建立起转换的“桥梁”。这种芯片的正式名称叫“**RS232**

收发器”（**RS-232 Transceivers**）种类比较繁多，通常按“桥梁”的数量进行划分。最常见的芯片包括**MAX202**（**2**路发送、**2**路接收，工作于**5V**电压），**MAX3232**（**2**路发送、**2**路接收，工作于**3V**电压）。**TI**、**INTERSIL**、**SIPEX**、**ADI**等公司都有兼容的产品。

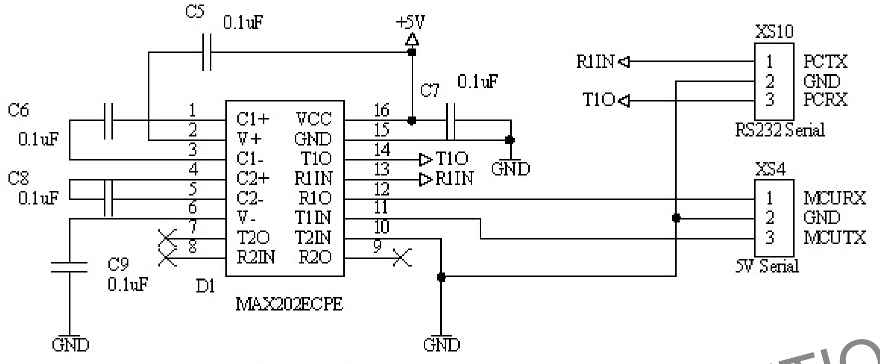
这种电路主要依靠电容型**DC-DC**提供工作所需要的±**10V**电压，所以芯片的外部元件基本都是电容。根据其工作原理，应该选择瓷片或者独石电容。这些电容的引线应尽量短，并安装在靠近芯片的位置。

作为接口电路，厂商可能会提供静电抑制和不带静电抑制两种版本例如**MAX202ECPE**和**MAX202CPP**，两者价差较大，如果实验场所静电比较严重（如北方地区），请选择带静电抑制的产品。

电容型 DC-DC 是一种使用电容作为储能元件的开关型直流-直流稳压电源架构，可以实现降压、升压、隔离、电压反向等稳压功能。

这种电路要求所使用的电容必须有尽可能小的等效串联电阻（ESR），通常应选用瓷片或独石电容，不益使用铝电解电容。

—— DA895 注



例图 2.14 RS232 通讯电平转换单元电路图

IT IS ONLY A DEMO EDITION

RS232通讯电平转换单元的调试稍微复杂一点，首先要确认芯片的**DC-DC**电路工作正常，用万用表测试**MAX202**第**2**脚与地之间应该有**10V**左右电压，否则应检查芯片的供电是否正常，各电容有无虚焊。

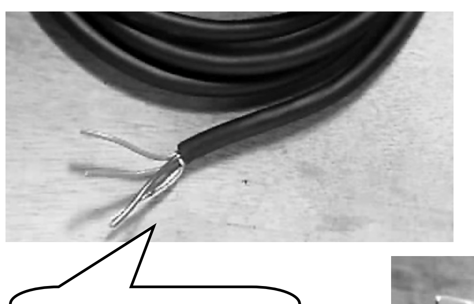
焊接**1**根如例图**2.16**所示的串口接线，用于连接**RS232**通讯电平转换单元和**PC**机串行口。

现在要介绍一个调试中经常使用到的工具——串口调试助手。这个软件很容易在网上找到，可以自行搜索下载。

接口电路一般会强调对静电的抑制能力（ESD 能力），静电放电实验模型分人体模型（Human body mode）和机器模型（Machine Mode）两种。

静电对芯片的威胁可能不会立即表现出来，随着时间的推移，这种“暗伤”比立即损坏的“明伤”更具破坏力。

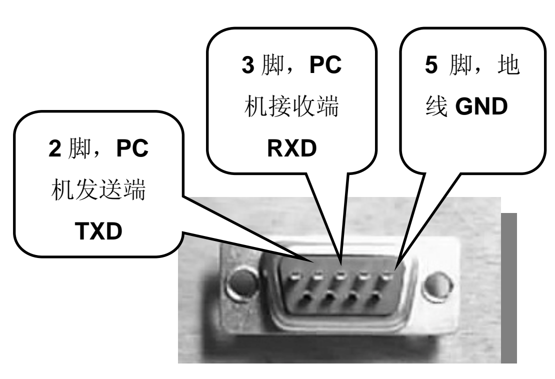
—— DA895 注

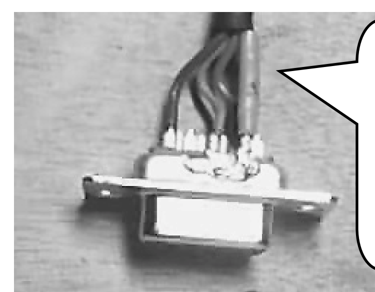


连接计算机主板上串行口的 **9** 针 **D** 型接口以及塑料外壳，注意不是针状的“插头”。

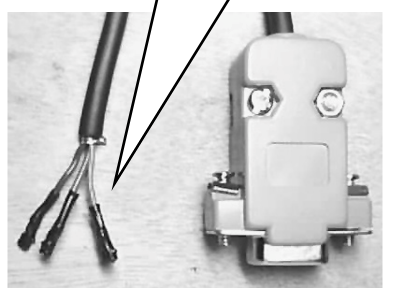
3 芯带屏蔽的电缆线，一般需要 **1~2m** 长。

例图 2.15 串口连接线的制作原材料

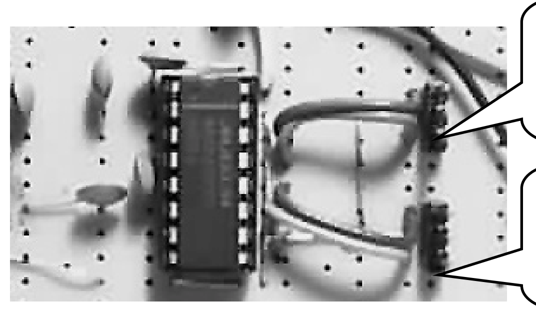




IT IS ONLY A DEMO EDITION



例图 2.16 串口接线制作示意图



例图 2.17 RS232 通讯电平转换单元-焊接完成后的效果

这种软件可以从串口发送数据，并监听、记录输入到计算机串口上的数据。

首先需要验证串口线制作正确，短路串口线插线端**TXD**、**RXD**两根线，用串口调试助手发送一串数据，应该能够在接收窗口中看到发送出的数据。再将这种检测方式延伸到**MAX202**芯片之后，用跳线将**XS10**的**1**、**3**脚短路，再次用串口调试助手发送一串数据，并观察接收结果。

如果能够正确接收所发送的数据，那么恭喜恭喜，您的串口模块已经能够正常工作了！

这种自发自收的串口检测方式通常被称为“自环线”

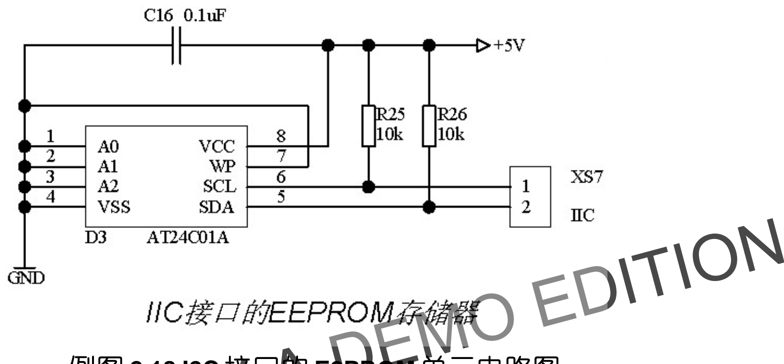
如果您的“自环线”检测没有通过，也请不要着急，只要前面所说的 10V 正常，出现这种问题一般都是虚焊所致。

用于测试串口自环线的软件可以在本书光盘中实例 16 的相关后台软件中找到。

—— DA895 注

[组装 I2C 接口的 E2PROM 单元]

这是我们实验板上组装的最后一个电路单元，核心元件是**AT24C01A**芯片。

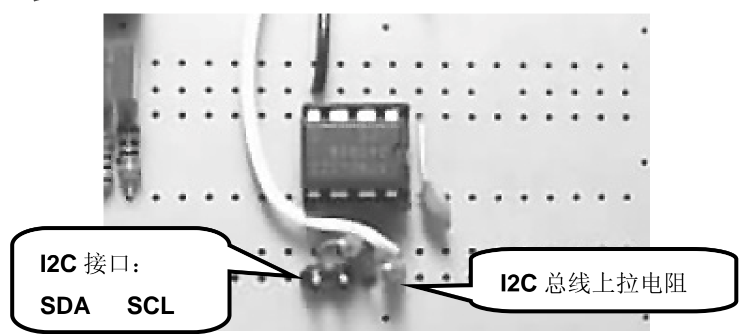


IIC接口的EEPROM存储器

例图 2.18 I2C 接口的 E2PROM 单元电路图

IT IS ONLY A DEMO EDITION

对这个芯片，我们同样用集成电路插座将他固定在电路板上。

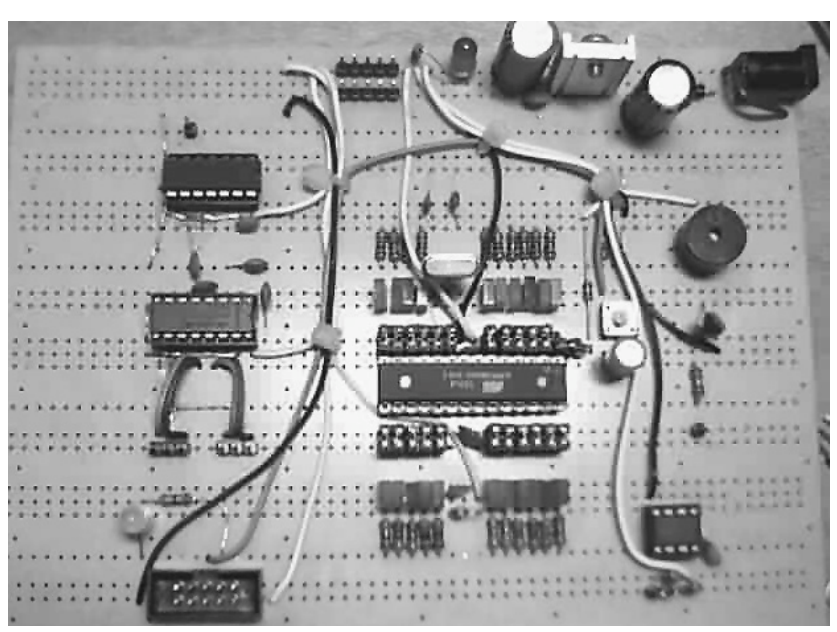


例图 2.19 I2C 接口的 E2PROM 单元-焊接完成后的效果

这个电路单元的调试工作，只能暂时放到本书实例16 **“存储器24C01的读写”**中再来介绍。

到现在为止，我们已经完成了实验板上所有电路单元的组装工作，来看一下他的全貌。

下面让我们进行最后的冲刺，焊接一根**STK-200/300**下载线。



例图 **2.20** 实验板完成后的全貌

[STK-200/300 下载线原理图简介]

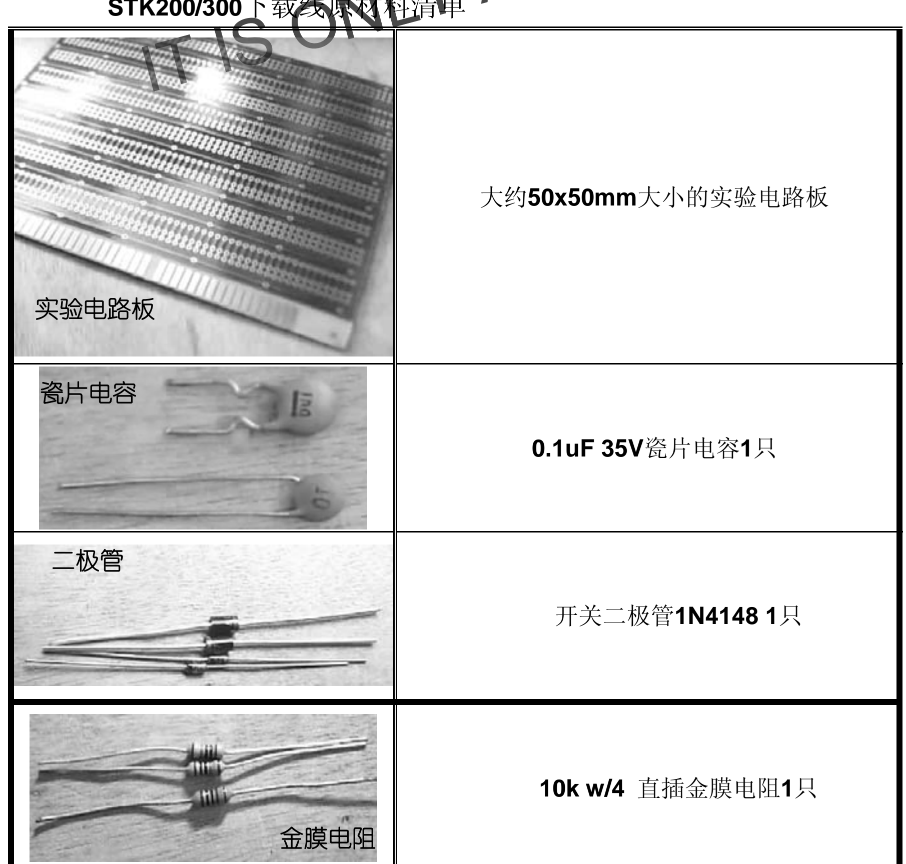
第 **4** 步
下载线
准备工作

STK-200/300是一种通过**PC**机并行口模拟**SPI**接口，对**AVR**单片机进行操作的下载线，电路图见**例图2.21**所示。

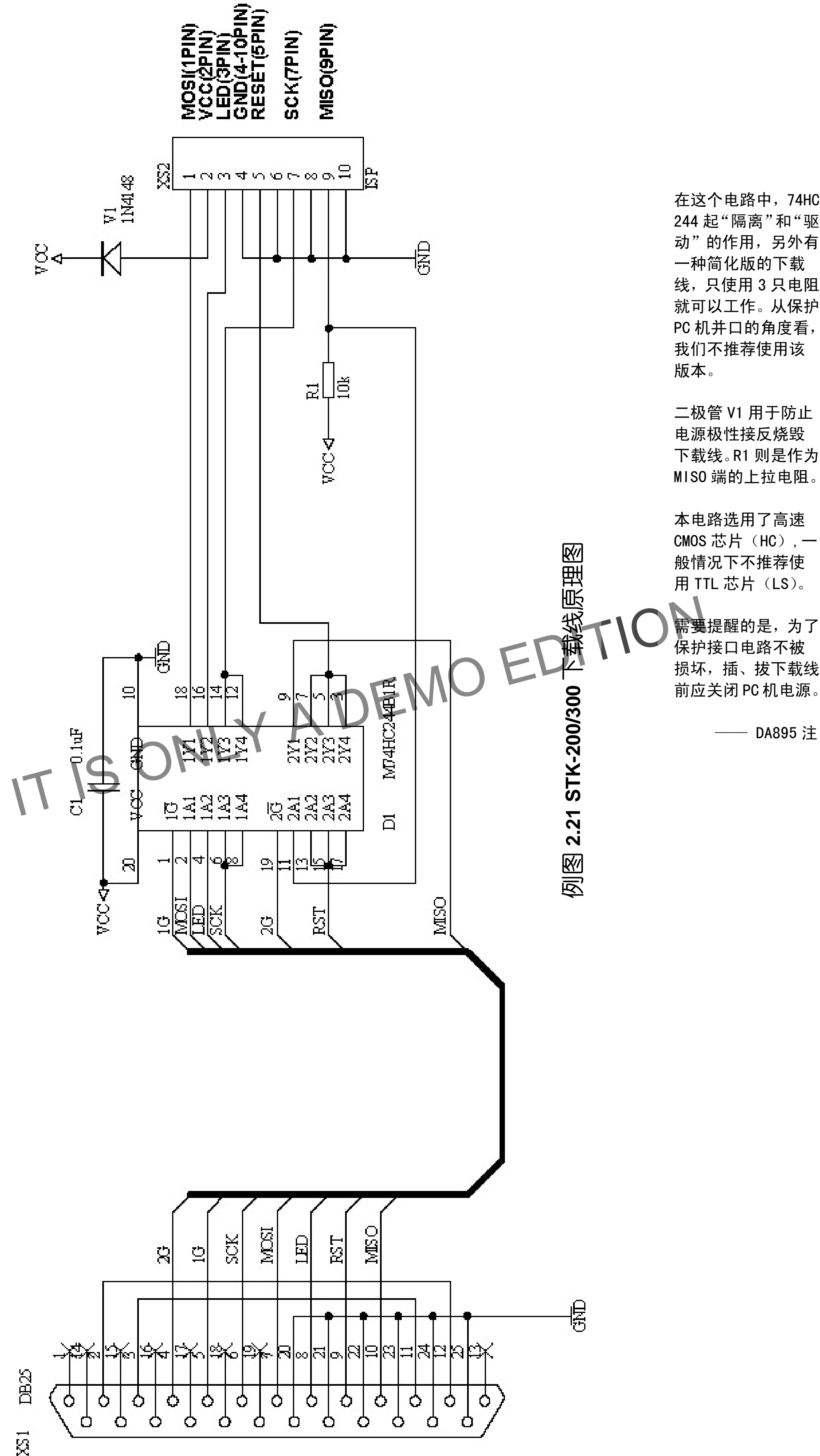
下载线的元件清单见下表：

IT IS ONLY A DEMO EDITION

STK200/300下载线原材料清单



实验电路板	大约**50x50mm**大小的实验电路板
瓷片电容	**0.1uF 35V**瓷片电容**1**只
二极管	开关二极管**1N4148 1**只
金膜电阻	**10k w/4** 直插金膜电阻**1**只



例图 2.21 STK-200/300 下载线原理图

在这个电路中，74HC244 起“隔离”和“驱动”的作用，另外有一种简化版的下载线，只使用 3 只电阻就可以工作。从保护 PC 机并口的角度看，我们不推荐使用该版本。

二极管 V1 用于防止电源极性接反烧毁下载线。R1 则是作为 MISO 端的上拉电阻。

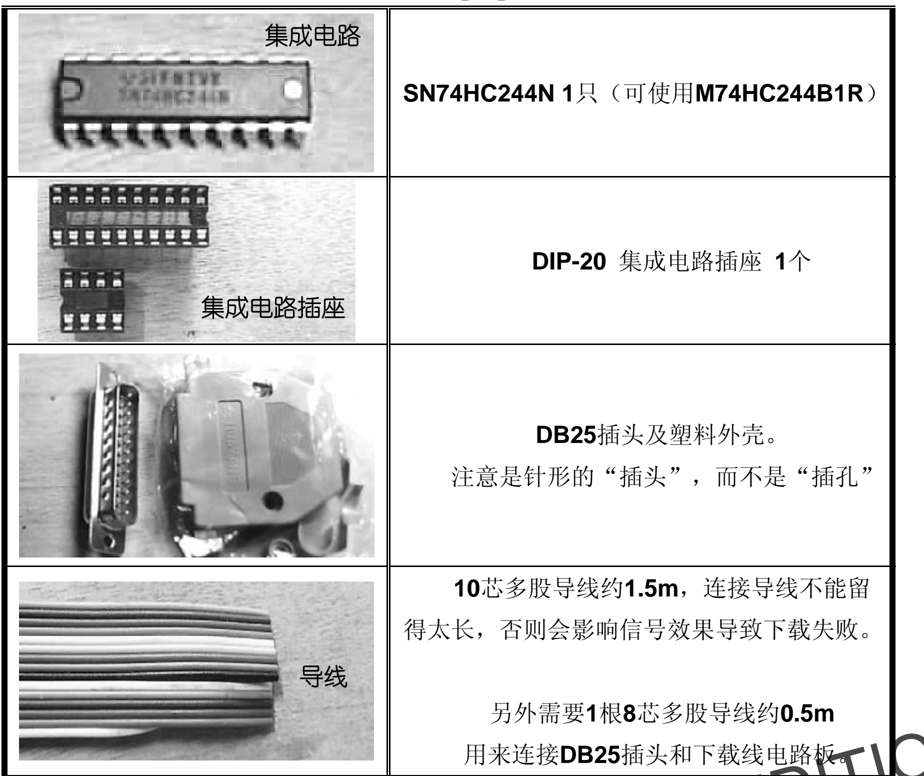
本电路选用了高速 CMOS 芯片（HC），一般情况下不推荐使用 TTL 芯片（LS）。

需要提醒的是，为了保护接口电路不被损坏，插、拔下载线前应关闭 PC 机电源。

—— DA895 注

IT IS ONLY A DEMO EDITION

STK200/300下载线原材料清单**[续]**



集成电路	**SN74HC244N 1**只（可使用**M74HC244B1R**）
集成电路插座	**DIP-20** 集成电路插座 **1**个
	DB25插头及塑料外壳。 注意是针形的“插头”，而不是“插孔”
导线	**10**芯多股导线约**1.5m**，连接导线不能留得太长，否则会影响信号效果导致下载失败。 另外需要**1**根**8**芯多股导线约**0.5m** 用来连接**DB25**插头和下载线电路板。

IT IS ONLY A DEMO EDITION

[组装 STK-200/300 下载线]

第 5 步

DB25

接口连线

下面我们就按图施工，组装下载线。

首先用**8**芯多股导线将**DB25**插头需要接出的引脚顺次接出。接地的引脚很多，只需要接出**1**根即可，其他的在插头上焊在一起。

别忘了这两根短接线哦，否则下载线会不工作的。

把线卡在外壳的空袭里，避免不小心拉断焊点。

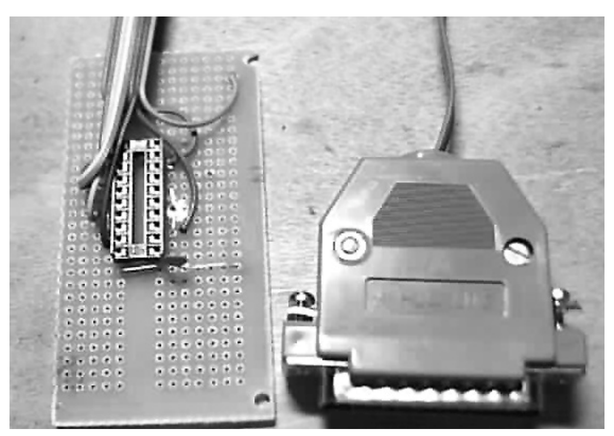
例图 **2.22 DB25** 插头连线完成

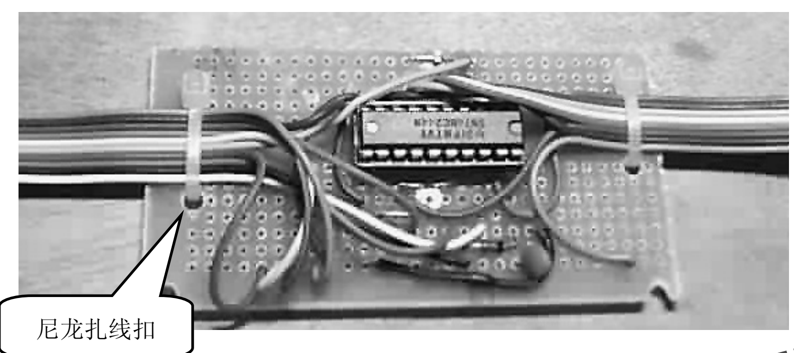
第 6 步

焊接

板上元件

下面来焊接电路板上的集成电路插座、**DB25**插头引出的接线、二极管、电阻、电容等。我用了两个尼龙扎线扣，把排线捆紧在电路板上，以防线头被扯掉。





例图 **2.23 ISP** 电路板焊接完成

将电路板引出到**ISP**插头的**10**芯排线压入插头。

IT IS ONLY A DEMO EDITION

第 **7** 步
制作 **ISP** 插头

这个接头压线的工作可不是很容易哦，建议多买几个接头备用，压好后要用万用表仔细检查各脚有没有接好，脚间有没有短路。

—— 傻孩子注

例图 **2.24 ISP** 插头压线

好了，下载线就算装完了！下面我们进入调试环节。

[STK-200/300 下载线及实验板小系统调试]

有了下载线，就可以下载任意一款**AVR**单片机了！

现在来解决实验板调试中最后一个悬疑——小系统的调试。我们通过编写一个操纵端口上发光二极管闪烁的程序来验证单片机中的程序可以正常运行，外部晶体工作正常。

由于大家现在还没有接触到编程相关知识，我们为大家事先编写了一个测试用的工程，可以在本书光盘内的“测试工程”文件夹下找到，文件名为“**test.c**”，编译好的目标代码名称为“**test.hex**”，该目标代码可以直接下载。

- 下载目标代码，确认单片机内部时钟工作正常：

刚买回的**ATmega48**单片机，其时钟源选择熔丝**CKSEL3..0**是配置在“校准的内部**RC**振荡器”状态的，只要给单片机提供电源，就可以下载和运行程序。鉴于外部晶体不一定能够稳定起振，在确认程序可以运行之前，不可冒然修改时钟源选择熔丝，以防止调试陷入困境。

打开双龙“**MCU**下载程序”软件，照例图**2.25**进行设置。

第 **8** 步 装载目标代码

IT IS ONLY A DEMO EDITION

例图 **2.25** 双龙“**MCU** 下载程序”软件的设置

单击“**FLASH**存储器”框架中的“**FLASH**”按钮 在弹出的“打开**FLASH**存储器数据文件”对话框中选中目标文件（上面所说的“**test.hex**”文件），单击“打开”按钮后，会弹出“空闲存储器填充”窗口，可以选择“填充**FF**”单选框，确定后即完成了文件的载入。

一般来说，目标代码不会占满整个单片机的程序存储器空间，如何对待剩余的空间就是“空闲存储器填充”的问题，一般来说安全起见，可以将其填充为 0xFF。

需要提醒的是，对于存储器而言，一般来说，所谓的“空”是指存储器中数据为 0xFF 的状态。

—— DA895 注

例图 **2.26** 打开 **FLASH** 存储器数据文件及空闲存储器填充对话框

第 **9** 步 连接下载

在主界面的“编程选项”中仅选择例图**2.25**中所选的项目，注意千万不要选择“配置熔丝”选项。

将**STK200/300**下载线的插头插入实验板**ISP**插口（**XS8**）。单击“编程”按钮，应看到主界面下方进度条开始指示下载进度。如果软件报告“进入编程模式失败！请检查**FUSE**设置、电源、时钟和**ISP**电缆连接”，说明在线下载的硬件通道不正确，应仔细检查下载线的焊接是否正确、**74HC244**芯片上有无电源供应、下载线插头引脚是否与单片机对应引脚可靠地连接。

如果您看到发光管闪烁，那么恭喜您，已经成功了一大半了！

—— DA895 注

下载完成后，应看到单片机各端口的发光二极管闪烁发光。

第 10 步

测试外部石英晶体

最后要测试的是外部石英晶体能否工作。**注意PB6和PB7引脚上的跳线帽务必处于插入状态!**

单击双龙“**MCU**下载程序”软件主界面中“配置熔丝”复选框将弹出“配置熔丝设置”对话框，可以通过单击“设置导航”按钮，切换到如**例图2.27**所示的“设置导航模式”。

选择如图中所示的时钟选项后单击“写入”按钮，软件将提示确认熔丝写入，确认后将把熔丝配置成外部晶体模式。

IT IS ONLY A DEMO EDITION

如果不做说明，这种熔丝设置将作为本书后面实例中的“默认”熔丝设置。

—— 老编注

例图 2.27 配置熔丝设置

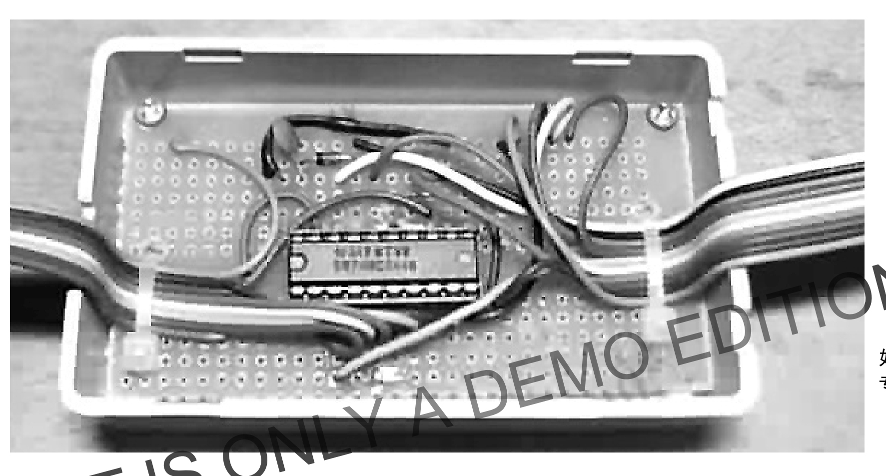
配置熔丝后，各端口的发光二极管应继续闪烁。如果发光二极管

停止了闪烁，说明晶体没有起振。

对于晶体不起振的情况，应先检查**PB6**、**PB7**引脚是否与晶体两端可靠连接，晶体两端的电容**C11**、**C12**容量是否装错。如果还不能起振，就试着用一根单独的导线将两个电容的接地端直接连接到**7805**的地线端。一般来说处理了上面的几种情况后，晶体会很容易地起振。

高兴之余，我们将下载线的电路板装在小塑料盒里，这样看起来更加专业。

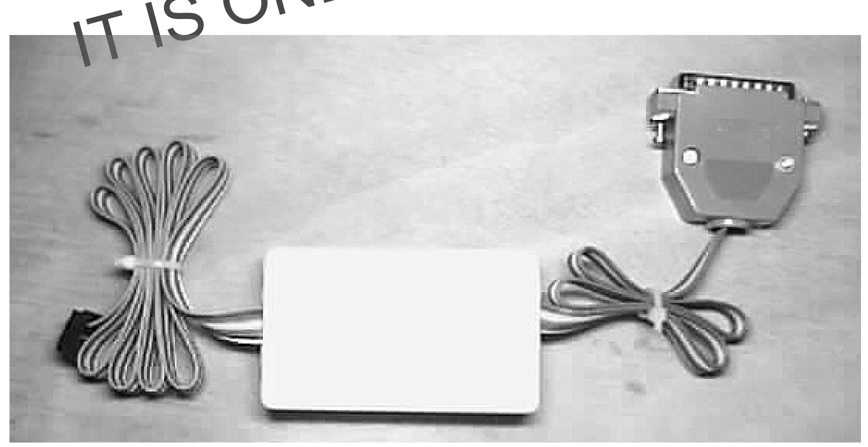
恭喜恭喜！现在您已经拥有了一套完整的**ATmega8/48/88/168**实验板和一根能够下载任意**AVR**单片机程序的下载线，现在您可以翻到第二篇，开始体验单片机学习的快乐了！



好像看起来还比较专业呢。不错，不错！

—— 老编注

IT IS ONLY A DEMO EDITION



例图 **2.28** 下载线完成

第二篇

从跑马灯开始　　对不起 接个电话

一秒钟究竟有多长　　电压低？

正在过收费站　　包装的学问

傻孩子求职记　　*MISSION Update*

IT IS ONLY A DEMO EDITION

人们常说：『艺术来源于生活、高于生活』，知识也是这样。在计算机类学科中，几乎所有概念都来源于对生活的抽象。不夸张地说：如果我们无法从生活中找寻出所学知识的原形，就不能认为完全掌握了知识点。

第四章 电量低！

本章引言

老编的 MP3 电池没电了，他想让我们帮他做一个可以测电池电量的装置。让我们一起用 ATmega48 单片机的 A/D 转换器实现。

IT IS ONLY A DEMO EDITION

本章牵涉知识点

- 什么是 **A/D** 转换器
- **AVR** 的 **A/D** 是什么结构
- **AVR** 的 **A/D** 是如何工作的
- 在使用 **A/D** 时需要注意些什么

……

原理 解析 <<阅读提示：原理简析部分试图将艰涩难懂的理论做浅显化的讲解，高手可以跳过

4.1 从猜数游戏到 A/D 转换器

为了帮助理解什么是**A/D**转换器，让我们先来做一个游戏。

首先老编在纸上写一个数字，这个数字必须在**0**到**60**之间，然后让傻孩子和**DA895**来猜这个数是多少。游戏规定，他们两个给出的数必须是**5**的整数倍，而老编只告诉他们说出的数跟他所写下的数字相比是偏大还是偏小，以最少次数猜到的人胜出。

A/D 是“Analog To Digital Converter”的缩写，中文名称是“模拟 / 数字转换器”，它的职责是将模拟信号量按照一定的规则转换为数字信号值，以使得数字电路或者单片机能够处理模拟信号量。

——DA895 注

首先老编写下**35**。

傻孩子说：“你写的是**40**吧？”

老编说：“偏大了！”

傻孩子说：“那就是**30**吧？”

老编又说：“偏小了！”

傻孩子说：“那肯定是**35**了。”

老编：“恭喜你，猜中了！”

第二次老编不厚道了，他钻游戏规则的空子，写下了**24**。

IT IS ONLY A DEMO EDITION

DA895开始猜：“我猜是**25**。”

老编：“大了！”

DA895又猜：“**20**！”

老编：“小了！”

DA895郁闷了：“我只能猜**5**的整数倍数字，怎么会出现比**20**大，又比**25**小的数呢？”

老编：“没有规定我给的数字也必须是**5**的倍数啊。”

DA895：“那我怎么可能猜得中呢？”

老编：“那就按照最接近的来算吧，你猜**25**也算对。”

这个游戏和**A/D**“量化”的概念是很相似的，老编写的数就像是模拟信号的电压值，他可以是任意的数值；傻孩子和**DA895**就像**A/D**转换器，他们用来“衡量”的只能是固定的几个数字。“**5**”在这里可以看成“步进量”，当定好“数字必须在**0**到**60**之间”和“猜的时候给出的数必须是**5**的整数倍”两个规定后，两个猜数人所能给出的数必定是**0**，**5**，**10**，**15**，**20**，**25**，**30**，**35**，**40**，**45**，**50**，**55**，**60**中的一个，但是他们不可能猜到老编“钻空子”给出的**24**，所以只有再定立一条规定“按照最接近的来算”。我们再加上一条“在两个**5**的整数倍数字中间的数据都做‘入’的处理，例如**27**，做‘入’的处理变成**30**”。

这样一来游戏规则就完善了。老编给出的任何数据，即使带小数位，也可以归结到**0**，**5**，**10**，**15**，**20**，**25**，**30**，**35**，**40**，**45**，**50**，**55**，**60**中的某一个数上。例如**35.2267**在傻孩子他们看来就是**35**。

现在我们再来看看傻孩子他们如何用最快的方法猜到老编写的数。

老编出的数字是**47**。

傻孩子可以从**5**开始说起，每次递增**5**，直到猜到为止，这种方法虽然简单，但是用来猜**47**这个数却需要用**9**次，很麻烦。

傻孩子动脑筋了：我先猜出大致的范围，然后再在这个范围里面猜不是简单了很多吗？

于是他先猜**20**，老编说小；他猜**40**，老编还说小；傻孩子就知道了：这个数是在**40**到**60**之间的。

现在他把“步进量”缩小为**10**，猜**50**，老编说大了，傻孩子马上说出数字是**45**！

傻孩子通过先定大范围再逐步缩小步进量的方法，只用了**3**次就猜到了老编的**47**，这和**逐次比较式A/D**的转换方法是类似的。

逐次比较式 A/D 是 A/D 转换器中的一种类型，ATmega48/88/168 的 A/D 就是这种类型。

——DA895 注

至此，我们对逐次比较式**A/D**已经有了一个基本的概念。读者可能会问：“为什么反复强调是逐次比较式**A/D**呢？”因为**A/D**转换器有多种类型，从工作原理上分，**A/D**转换器可分为**直接A/D**转换器和**间接A/D**转换器，从量化原理上又可分为**并行比较式、逐次比较式、双积分式**等类型。

直接 A/D 将模拟信号（例如电压）直接转换为数字信号值，速度较快。

间接 A/D 先将模拟信号转换为某种中间量（例如时间或频率），再将中间量转换为数字信号值，速度较慢，但抗工频干扰能力较直接 A/D 好。

关于 A/D 转换器更详细的知识，请读者查阅数字电路或 A/D 转换器的专门著作。

——DA895 注

IT IS ONLY A DEMO EDITION

4.2 ATmega48/88/168 的 A/D 转换器

单片机集成**A/D**转换器就像计算机主板通常要集成声卡一样——**ATmega48/88/168**也不例外。它集成了一个**6**通道（**TQFP**与**MLF**封装为**8**通道）、**10**位精度、提供内部电压基准源的逐次比较型**A/D**转换器（后面如果没有特殊说明，将简称其为**A/D**）。下面，我们跟随着**Datasheet**来简单看一下它的各项特性：

- **10**位精度：

让我们再次回到猜数游戏。现在老编又把游戏规则改了，老编不仅规定了数值的上限，而且限定傻孩子他们所猜数的个数只能有**11**个。例如老编将数值的上限定为**60**，傻孩子他们就只能从**0**、**6**、**12**、**18**、**24**、**30**、**36**、**42**、**48**、**54**、**60**这**11**个数字里猜一个；如果老编将数值的上限定为**100**，则他们只能从**0**、**10**、**20**、**30**、**40**、**50**、**60**、**70**、**80**、**90**、**100**里做出选择。老编这里规定的“所猜的数的个数”在**A/D**中被称为“**分辨率**”。与游戏中的情形类似，标题中所提到的“**10位精度**”，是指该**A/D**使用了**10**位的二进制数来表示输入电压的量化值，其最小值为**0x000**（**0b00,0000,0000**），最大值为**0x3FF**（**0b11,1111,1111**）——也就限定了总共只有**1024**（**0x3FF + 1**）个数可以选择。而“数值的上限”在**A/D**中被称为“**参考电压**”，输入的电压值应该在**0**至参考电压值之间，因为**A/D**转换器无法辨认超出参考电压范围的电压值。

这个参数的准确名称应称为“分辨率”（Resolution）。

——DA895 注

当“参考电压”和“分辨率”被确定后，每两个数值间的差值，即“步进量”就确定下来了。例如在猜数游戏中，规定数值上限为**60**，

可选的个数为**11**个，则“步进量”为**6**；当规定数值上限为**100**，可选数的个数同样为**11**个时，步进量就变为**10**。对于**A/D**来说，精度一般都是给定的常数，而参考电压却是可以选择的。上面的“步进量”在**A/D**中称为**1LSB**（最低有效位，**Least Significant Bit**）所代表的电压值。以**5V**参考电压、**10**位精度的**A/D**为例**1LSB**能够表示的电压值为：

1LSB 所表示的电压值 **=** 参考电压**5V /** （**0x3FF + 1**） **= 4.88mV**

由于步进量是固定的，当老编不厚道，给出不足一个步进量整数倍的数值时，傻孩子他们就只能猜到与此数最接近的数值，而始终无法准确地猜到这个数。对任何**A/D**来说，量化后输出的数字信号值都是以**1LSB**的电压值“步进”的，介于**1LSB**之间的电压将按照一定的规则进行入位或舍弃，这个过程中造成的误差被称为**“量化误差”**。

A/D对小于1LSB的输入电压信号，通常有两种处理方法：四舍五入或只舍不入。

从 ATmega48/88/168 的 DATASHEET 看出，它们属于四舍五入的处理方式。

——DA895 注

量化误差属于原理性误差，是无法消除的，因为傻孩子他们只能用“有限”个数字去逼近老编给出的“无限”种可能。从原理上讲，在参考电压一定的情况下，位数越高的**A/D**，其**1LSB**所代表的电压值越小，**A/D**的分辨率（精度）也越高。

- **0.5 LSB** 的非线性度：

在修正了偏移误差和增益误差后，所有转换后得到的数字信号值与实际输入电压信号间的误差，以**1LSB**的倍数为计量单位来度量。

IT IS ONLY A DEMO EDITION

- **± 2 LSB** 的绝对精度：

将所有的误差因素计入后，所有转换后得到的数字信号值与实际输入电压信号间的误差，以**1LSB**的倍数为计量单位来度量。

关于“偏移误差”与“增益误差”的解释，请参阅 DATASHEET 中“ADC 精度定义”部分.

这三个参数对初学者意义不大，若不易理解可暂时跳过。

——DA895 注

- **13 - 260 µs** 的转换时间：
- 最高分辨率时的采样率高达**15 kSPS**：

这两项都是描述**A/D**的转换速度的。逐次比较型**A/D**的转换工作在时钟的指导下进行。时钟速度越高，对应的转换时间越短，但要牺牲转换精度作为代价。**13 - 260 µs** 的转换时间是指连续转换模式下，通过时钟选择所能得到的转换时间。

- **6** 路复用的单端输入通道：
- **2** 路附加的复用单端输入通道(**TQFP**与**MLF**封装)：

系统中常常不止一个模拟电压需要被单片机读取，而为了适应这一需要，单片机制造商们一般也不会只做**1**路**A/D**就“善罢甘休”。例如**ATmega48/88/168**就配备了**6~8**路的**A/D**输入通道，所谓“**6~8**路”实际上是因为**PDIP**封装的**ATmega48/88/168**提供了**6**路**A/D**输入，而**TQFP**和**MLF**封装的提供了**8**路**A/D**输入。

您可以参照附录八的 AVR 端口封装图。

——DA895 注

从**Datasheet**的“引脚配置”中我们可以看出，**ATmega48/88/168**的**ADC0**到**ADC5**输入功能被复用在**PC0**到**PC5**引脚上，而**TQFP**和**MLF**封装附带的**ADC6**和**ADC7**有单独的输入引脚。

需要注意的是，为了节约成本，这里多路**A/D**的输入实际上在共享

同一个**A/D**——**AVR**单片机通过一个模拟电子开关来决定哪一路输入与**A/D**相连接。同一时刻，**A/D**只为其中一路服务。

- 可选的左对齐**ADC**读数：

前面我们知道**A/D**转换所得结果占据**10**个二进制位，单纯使用一个八位寄存器不足以存储和表达，所以**AVR**使用了**ADCH**和**ADCL**两个寄存器来保存转换结果。平时（右对齐）**ADCH**中**Bits 7..2** 实际上是保留不用的。左对齐为**A/D**转换结果提供了一种灵活的读取方式，我们先来看一下左对齐会带来怎样的结果。

在Datasheet中提到AD转换器有一个对寄存器ADC进行保护的功能。即，在用户没有读取寄存器ADC中数值的时候，正在进行的采样得到的结果将不会破坏寄存器ADC中的数值。简单来说，我们不读取，数值就不会被更新。一旦我们进行了读取操作，新的数据就可以进入寄存器ADC了。

事实上，这一功能是通过简单地检测寄存器ADC的高字节ADCH来实现的。我们不读取寄存器ADCH，寄存器ADC就不会被更新；一旦我们读取了寄存器ADCH，那么新的数据就有机会进入寄存器ADC了。

正因为这样，Datasheet才一直强调我们应该先读取寄存器ADCL再读取寄存器ADCH。根据这一要点，我们可以直接读取寄存器ADCH而不用读取寄存器ADCL。这种读取方式在左对齐模式下非常有意义——它使我们能够直接获得8位精度的AD采样值。

——DA895 汴

例 A/D转换结果的左对齐：

10位分辨率的**A/D**转换得到的结果原为：

ADCH = 0b0000,0010

ADCL = 0b1010,1100

将**ADMUX**寄存器的**ADLAR**位设置为“**1**”，就选择了**A/D**结果左对齐，则**A/D**转换得到的结构将被调整为：

ADCH = 0b1010,1011

ADCL = 0b1100,0000

这个左对齐调整相当于将**ADCH**和**ADCL**相串接后向左移位**6**位，如图**2.4.1**。若在转换中只需要**8**位的精度，就可以利用左对齐，在转换后仅取**ADCH**中的数据。由于越靠近低位的数据所代表的电压值越小，左对齐、只取最高**8**位就相当于忽略了最低**2**位结果所表征的“细节”。

IT IS ONLY A DEMO EDITION

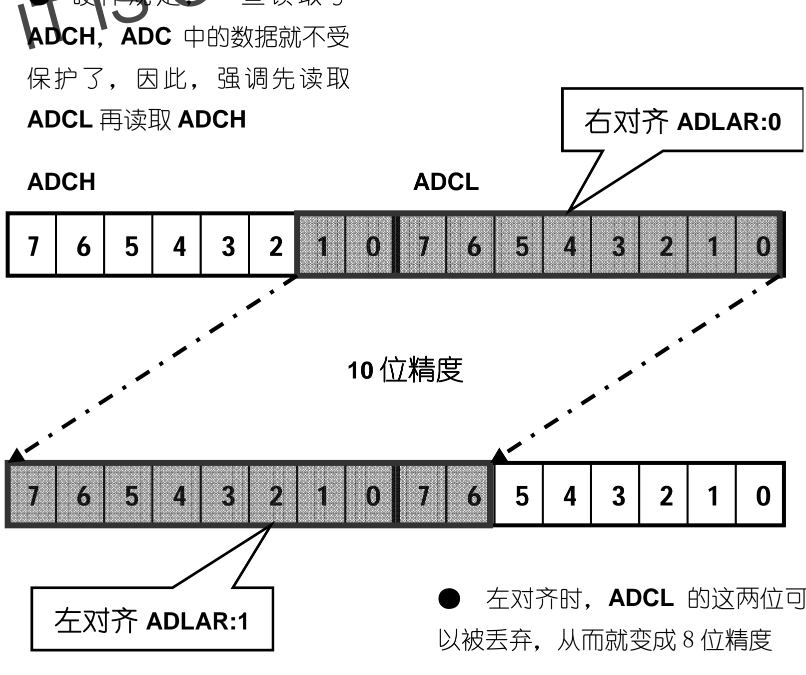


图 **2.4.1** **AD** 转换对齐结构示意图

- **0 - VCC** 的 **ADC** 输入电压范围：

● 可选的**1.1V ADC** 参考电压：

就像我们在猜数游戏中的规则一样，**A/D**通常是规定了可选数的个数，而它的上限在一定范围内是可选的，这个“上限”被称为“参考电压”。作为转换的基准，输入电压的范围一般处于**0V**到参考电压之间。**AVR**的参考电压可以通过**3**种途径获得：**AVCC**引脚上的电压、**AREF**引脚上的电压、片内自带的**1.1V**电压基准。**作为“基准”，参考电压的精确性直接影响了A/D转换的精确度。**

片内自带的电压基准源给一些用电池供电的手持式设备提供了便利，在使用**A/D**时，不需要额外添加电压基准源芯片作为参考。由于参考电压可以达到电源电压，故输入电压的范围是**0 - VCC**。

AVR 很多其他型号的芯片提供的内部参考源是 2.56V 的，使用内部参考源时，请一定要仔细阅读 Datasheet 进行确认。

——傻孩子注

● **ADC** 转换结束中断：

与单片机其他模块相似，**A/D**的转换结束事件也可以触发**CPU**中断。一旦单片机设置并启动了**A/D**，随后的转换过程就不需要**CPU**操心了，它可以放手去做其它工作；当转换工作完成，开启了中断模式的**A/D**转换器模块就会发出中断请求，通知**CPU**执行相应的中断处理程序。

● 基于睡眠模式的噪声抑制器：

为了进一步降低**A/D**转换时的噪声，**AVR**允许在**A/D**转换时将**CPU**转入睡眠模式，本章进阶阅读中将讨论这种抗噪声的工作方式。

IT IS ONLY A DEMO EDITION

实际 应用 ◁阅读提示：实际应用部分着重介绍该资源在实际应用过程中常见的方法，入门必读

4.3 ATmega48/88/168单片机中与A/D相关的引脚

● **A/D**输入引脚**ADC0~7**：

作为信号的输入端，**ADC0**至**ADC5**分别与引脚**PC0**至**PC5**复用。对于**TQFP**封装和**MLF**封装的**ATmega48/88/168**单片机才有**ADC6**和**ADC7**这两个独立的输入引脚，况且不与其它端口复用。

● **A/D**模拟供电端**AVCC**：

这是一个电源输入引脚，为**A/D**模块（包括**ADC7**和**ADC6**）和**PC0**至**PC3**端口提供电源。在不使用**A/D**模块的情况下，该引脚应接至**VCC**（单片机**VCC**引脚所接的电源）引脚上；当使用**A/D**模块时，应通过一个低通滤波器将该引脚与**VCC**连接起来。

低通滤波器用于阻断数字电路运行时所引入的噪声通过电源线串入到敏感的 A/D 模块中。只要能保证直流压降满足要求，这个低通滤波器可以用任何方式实现。

硬件的详细内容请读者阅读 Datasheet 中的相关部分，这里只做粗略介绍。

——DA895 注

● **A/D**参考电压**AREF**：

这个引脚上所接入的电压将会被作为的**参考电压**，成为**A/D**转换的基准源。**ATmega48/88/168**的**A/D**模块可以使用**AVCC**或内部的**1.1V**电压基准源作为参考电压。他们的选择关系如图**2.3.3**所示。**AREF**引脚直接

与最终选定的参考源相连，当选择AVCC或者内部1.1V电压作为参考电压时，AREF引脚上的电压就是AVCC的电压值或1.1V；当选择AREF引脚上的电压作为参考电压时，该引脚可以接入任何满足要求的参考源。

4.4 ATmega48/88/168单片机中与A/D相关的寄存器

- **ADC多路复用选择器ADMUX：**

ADMUX寄存器用于选择参考电压、转换结果的对齐方式和输入通道。对于具体的设置，这里不再分析，请读者参阅Datasheet中“ADC多路复用选择寄存器－ADMUX”部分的描述。

很多人阅读这一部分内容都会觉得相当吃力和枯燥，这是由于你没有打开DATASHEET进行比照阅读的缘故。其实，你不必过于在意这些内容，ICC中有一个专门的代码生成器可以帮助我们配制好这些烦人的寄存器。随后的内容中，我们会详细介绍如何使用代码生成器获取我们所需的ADC代码。

——傻孩子注

- **ADC控制及状态寄存器ADCSRA /ADCSRB：**

ADCSRA和ADCSRB寄存器用于设置A/D的启动条件、中断状态、转换速度等运行行为。请读者参阅Datasheet中“ADC控制和标志寄存器A－ADCSRA”及“ADC控制和标志寄存器B－ADCSRB”部分的描述。

- **ADC数据寄存器ADCL /ADCH：**

ADCL和ADCH分别用于保存A/D转换结果的低半字节和高半字节。对它们的操作可参考本章“4.7 读取A/D转换的结果”部分的讨论。

IT IS ONLY A DEMO EDITION

- **数字输入禁止寄存器DIDR0：**

DIDR0用于关闭模拟信号接口上的数字电路接口功能，请大家参考本章“4.5 在使用A/D时需要注意些什么”中的相关讨论。

- **功耗抑制寄存器PRR：**

需要注意，PRR寄存器控制着“是否向单片机各主要模块供应电源”，其中的PRADC位与我们讨论的A/D模块直接相关。在默认情况下，PRR总是会向A/D转换器供应电源的

4.5 使用 A/D 时需要注意些什么

- **AREF参考电压端连接的注意事项：**

由于A/D的参考电压可以从AVCC、AREF和内部1.1V电压基准源上任选其一，而AREF引脚与最终被选定的参考电压始终是相连的，如图2.4.2所示。

当AREF上连接有其它电压基准源时，绝对不要选择内部1.1V电压基准源，否则将导致两个电压基准源短路，烧毁器件！

在选择内部1.1V电压基准源时，仅能在AREF端接一个高频特性好、漏电流小的电容器（例如瓷片电容）以帮助稳定内部电压基准源的输出电压。

- **输入端串接电阻：**

为了避免电平冲突，**A/D**的输入电压和引脚间应该串接一个阻值适当的电阻。由于该电阻的阻值还会影响采样/保持电路的稳定时间，所以这个电阻也不能取得太大，可根据情况在**1k~10k**的范围内选取。

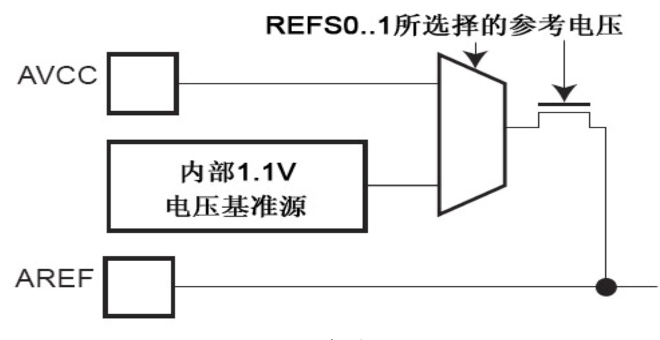


图 **2.4.2 A/D** 参考电压的选择

● 禁止输入端的数字缓冲器：

从原理上说，普通**CMOS**数字电路的输入端是不能接模拟信号电平的，否则将造成其输入级过载。由于数字输入禁止寄存器**DIDR0**在上电后被初始化为**0x00**，即所有带有**A/D**输入功能的引脚都被开启了数字输入缓冲器，因此需要输入模拟信号的引脚应在上电后的器件初始化中应用**DIDR0**寄存器禁止该引脚的数字输入缓冲器。

例如我们想使用**ADC3**引脚作为模拟信号输入端，就应在初始化中加入如下代码：

```
DIDR0 | = 0x08;                    //关闭PC3的数字输入缓冲
```

注意，这里我们关闭PC3 的数字输入功能时用了"或"的位操作，与后面程序代码中直接赋值的方法有所不同。

之所以使用位操作，是想提醒读者，在软件设计中，对于不被本软件使用，又不清楚当前状态的寄存器位一定不要去随意修改，换句话说就是"不要动别人的奶酪"。

——傻孩子注

IT IS ONLY A DEMO EDITION

● 电源退偶：

A/D属于容易受到噪声干扰的单元，在线路设计和绘制**PCB**时应格外注意。对于参考电压，若选用内部**1.1V**电压基准源时，在**AREF**端与地之间接入一个**0.1uF**或**0.01uF**的瓷片电容器，对改善电压基准源的噪声有好处；若选用**AVCC**或**AREF**端，则应做好该端与单片机供电端**VCC**间的退偶工作，推荐使用独立的电源供电或使用**LC**低通滤波器进行滤波，至少要用低通的**RC**滤波器将其与**VCC**隔离开，参见图**2.4.3**所示。

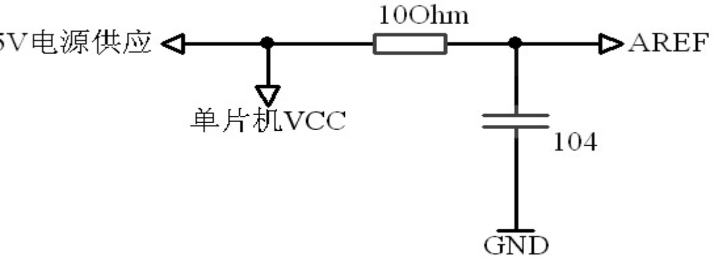


图 **2.4.3 A/D** 参考电源的退偶

● 输入通道与参考电压转换等其它注意事项：

在**A/D**的使用过程中，如果需要转化**A/D**的输入通道、参考电压等因素，当前**A/D**转换的结果可能会随设置的改变而被破坏，在应用时应注意**Datasheet**中给出的边界条件。

4.6 怎样知道A/D转换完成

当设置好A/D模块的相关寄存器并启动转换后，A/D模块将自动完成一次转换过程（单次转换模式）。在转换期间，CPU可以去处理其它任务，但是它又如何知道什么时候去读取A/D转换的结果、执行相应的处理程序呢？

光盘里的本章文件夹下有查询、中断、ADC噪声抑制模式唤醒方式实现电平表的完整源代码，供大家参考。

——傻孩子注

一般来说，CPU可以通过查询和中断两种方法得知A/D转换工作已经完成。“ADC噪声抑制模式”唤醒CPU的方式从本质上讲还是中断模式。

- 查询方式：

查询方式说得通俗一些就是“死等”，即CPU什么事情也不做，一直守着系统提供的标志位，等待A/D转换完成。查询方式的好处是代码简单、容易理解；缺点是CPU在转换期间一直处于空转状态，白白浪费了宝贵的时间。当实际应用中CPU不需要同时处理A/D转换和其它事务时，可以考虑使用查询方式。

我们来看一个查询方式的代码片段。

事实上“死等”只是实现查询的一种最简单模式。认为中断方式一定比查询方式要“有效率、高级“是错误的。很多高档的实时系统中，查询被认为是比中断更为有效的信息采集和处理方式。

在这些系统中，查询真正的目的只是“查询”，如果所期待的事情没有完成，就继续做其它事情；而随后每隔几乎相同的一段时间，系统又会回来“查询”一下。

——傻孩子注

```
ADCSRA |= 1<<ADSC;                //置位ADSC位,启动一次转换
while(!(ADCSRA&(1<<ADIF)));       //查询方式等待转换完成
voltage = ADC;                    //读入ADCH和ADCL中的转换结果
```

IT IS ONLY A DEMO EDITION

在设置完ADSC位并启动当前转换后，程序将使用while循环用以等待ADIF位被硬件设置为“1”，仅当这个条件成立时，while循环才会结束，CPU继续读取ADCH和ADCL中的转换结果。

我们来分析一下while(!(ADCSRA&(1<<ADIF)));语句。这个while循环的循环体是空的，只要它的条件!(ADCSRA&(1<<ADIF))不为“假”，这个空循环就会一直执行下去。ADIF是一个在头文件iom88v.h中定义了的一个值为4的常量。(1<<ADIF)在编译器看来也是一个常量，它表示把一个字节BIT4（第5位）设置为“1”，其它位保持为“0”，即0b00010000。逻辑与运算(ADCSRA&(1<<ADIF))相当于(ADCSRA&0x10)，用于从寄存器ADCSRA中“取出” ADIF标志位。注意括号外的逻辑非运算符“!”，它并非使运算后的值直接“颠倒”（那是运算符“~”的工作）。在这里，当ADIF位为1时，则与运算的结果就是一个非0的值，经过非运算“!”后，就会变成0；同样，当ADIF位为0时，与运算的结果就是0，经过非运算后，就变成一个非0的数值。最终的结果是：若ADIF位为0，则while循环会一直执行下去，实现了“查询等待转换完成”的功能；当转换完成，ADIF位被硬件置为1时，while循环的条件被破坏，CPU继续执行while(!(ADCSRA&(1<<ADIF)));之后的语句。

- 中断方式：

采用中断方式时，CPU在启动转换后，可以转而去做其它的处理工作，当转换完成后，A/D模块通过中断的方式通知CPU。与查询方式相比，中断方式下的CPU效率可能更高。

大家可能会注意到：整个程序实际上只是在循环进行A/D转换。这是因为电平表的程序只需要进行两个实际操作：

- 用A/D读取电压值；
- 将读到的电压值处理成光带形式并将结果通过端口在LED上显示出来。

在没有得到A/D转换结果之前就谈不上将转换结果转换并显示在端口上，而除此以外单片机似乎也没有其它任务需要执行。事实上，对于中断方式，CPU所需要处理的决不仅仅是如此简单的两个操作。

为了讲解中断方式下A/D转换的整个流程，我们在程序中加入了一个周期为1ms秒的定时中断，在定时中断中启动A/D转换，当一次转换完成后，A/D会向CPU发出“ADC转换完成中断”的报告，CPU会停下手头的工作直接跳转到“ADC转换完成中断服务程序”中处理数据，并将结果送到指定的端口通过LED显示出来。

采用中断方式的时候，处理器还需要额外处理“保存现场”和“恢复现场”的工作。这一细节，我们在前面第二篇第二章中已经介绍过了。

——傻孩子注

- ADC噪声抑制模式唤醒方式：

A/D转换器与CPU共同“居住”在同一块芯片内，共享电源和地线（即使在片内分开了，在片外还是会接在一起），CPU的数字电路不可避免地会影响敏感的A/D转换器。为了较彻底地消除它们之间的干扰，AVR引入了“ADC噪声抑制模式”，该模式下的A/D转换在CPU休眠期间进行，从很大程度上避免了数字信号对转换结果的干扰。在本章的进阶阅读部分，我们将详细讲述这种转换方式。

IT IS ONLY A DEMO EDITION

4.7 读取A/D的转换结果

当A/D转换器完成一次转换后，转换得到的结果存放在ADCH和ADCL两个寄存器内，程序需要将ADC转换所得到的数据读入到某一无符号整形变量（unsigned int）中等待进一步的处理。

为了防止在上一次转换结果还没有被读走，下一次的转换结果就将其“冲刷”掉，造成数据丢失，AVR的A/D采取了一个保护策略：

当程序语句读取ADCL寄存器时，ADCH的值将被锁定，直到程序读取了ADCH寄存器后，A/D转化器硬件才能将下一次的转换数据写入到ADCH中。

让我们来看一个代码示例。

```
adc_l = ADCL;        //读取ADCL寄存器中的转换结果,本语句执行后ADCH寄
                     //存器中的值已经被锁定
delay_ms(1);         //调用1mS延时函数,即使在此时另一次A/D转换完成,
                     // ADCH的值还是保持上次转换的数据
……                   //其中包括“启动一次A/D采样”在内的可能的操作
adc_h = ADCH;        //ADCH仍然保持上次转换的结果,该数据与adc_l变量中
                     //保存的数据仍然能组成正确的转换结果
```

当仅需要8位转换精度时，可以选择“转换结果左对齐”，每次读取转换结果时，仅需要读取寄存器ADCH而不用理会寄存器ADCL。

使用ICC AVR编写C语言代码，编译器会自动处理读取顺序的问题，用户可以绕过寄存器ADCH和ADCL而通过一个宏ADC直接将数据读取到一个无符号整形变量中。

```
unsigned int adc_result;      //定义一个自整型变量用于保存A/D转换结果
adc_result = ADC;             //读取A/D转换的结果,直接以整型变量的形式存放
```

AVR 单片机中，并不存在ADC这样一个寄存器，这是由ICC编译器在头文件iomXXv.h中定义的，和寄存器ADCL的地址相同。

——傻孩子注

这是使用C语言编写程序的好处之一，“智能”化的编译器为读者省去了很多麻烦，但是并不是所有编译器都如此“聪明”，对于使用汇编的用户则更需要注意读取顺序。

本章的进阶阅读中，我们将详细剖析几种常见的转换结果读取方式，从汇编的角度分析它们的执行效率。

4.8 使用代码生成器生成ADC初始化代码

到目前为止，大家已经了解了使用A/D模块所需要的一些基本知识，本节中我们来学习一下如何通过ICC编译器自带的**代码生成向导**生成我们所需ADC初始化程序。

IT IS ONLY A DEMO EDITION

第一步，点击我们已经熟悉的“魔法帽子”图标，打开代码生成器窗口。在本章实例中AVR芯片型号选择“M48”，时钟频率Xtal speed(MHz)选择“1.0000”，如图2.4.4所示。

第二步，单击位于代码生成器窗口上部的“Analog”标签，生成器将自动切换到ADC和模拟比较器设置窗口，如图2.4.5所示。在这里，我们只讨论ADC的设置。应该指出，这个设置界面并不完整，例如对通道号的选择就没有体现，这就需要我们在代码生成器的基础上再对初始化代码作少许的修改。让我们先来介绍一下这个界面中的各个选项。

● 复选框 Use ADC

给该复选框打勾，意思就是“我们要使用 ADC 模块”。如果不理睬该选项，无论后面我们做了怎样正确的设置，也不会得到任何代码。**推荐设置：打勾。**

● 复选框 Power off

这是一个和节能有关的设置，可以暂时不用理它。**推荐设置：不打勾。**

● 复选框 ADC enable

A/D转换器的使能开关。不选中的话，模块是不会运行的。**推荐设置：打勾。**

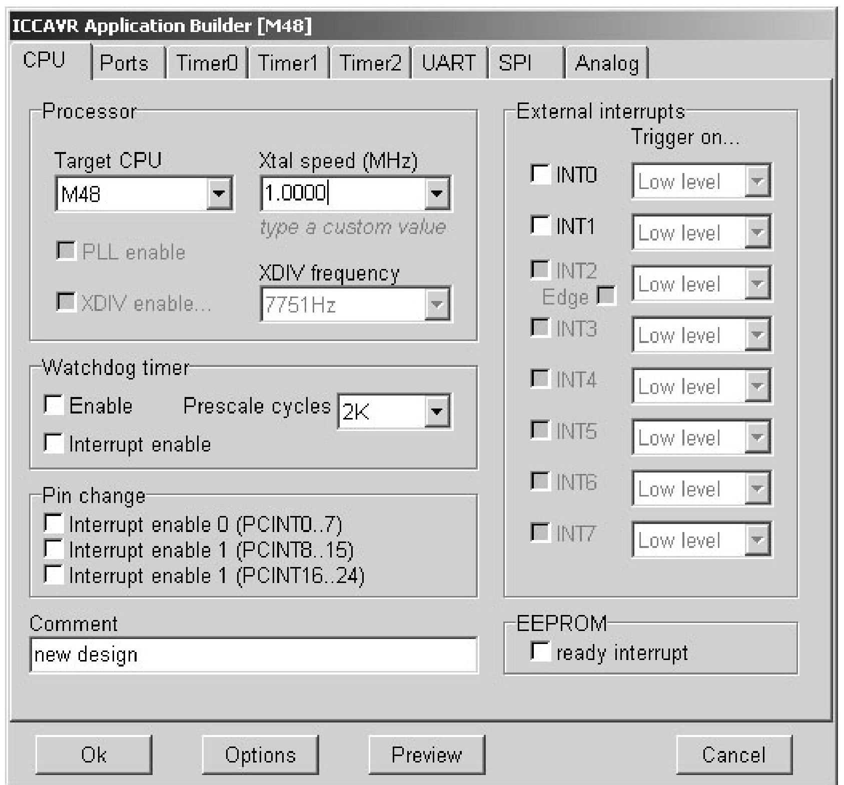


图 2.4.4 设置处理器和频率

IT IS ONLY A DEMO EDITION

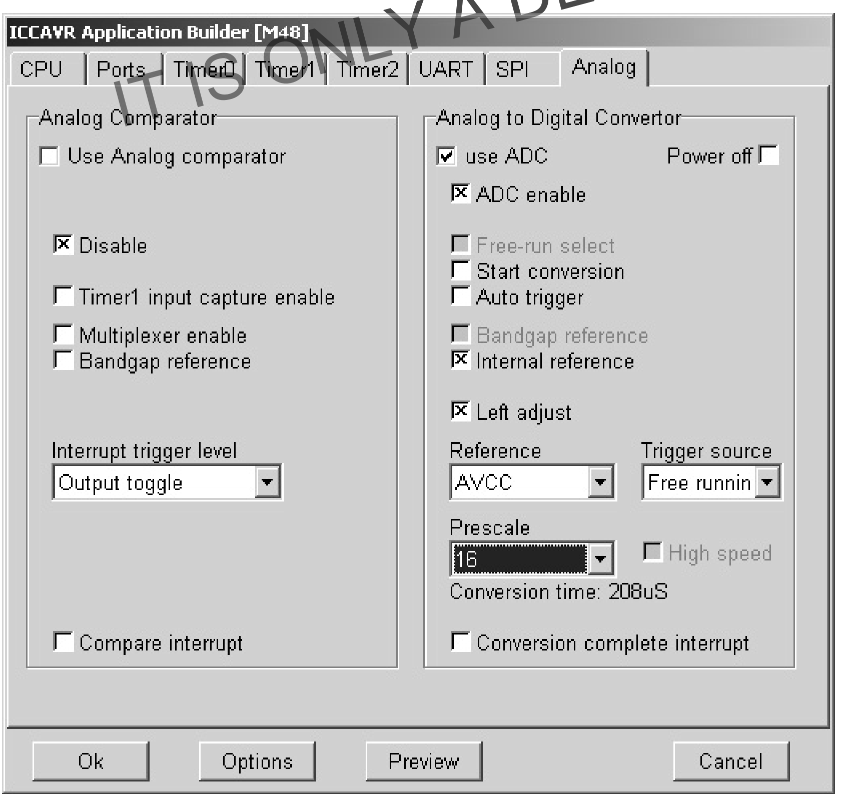


图 2.4.5 ADC 设置窗口

ICC6.31A 中，这里的 Internal reference 复选框和 Reference 下拉列表是无效的。这一 BUG 在随后的 ICC7.0 中得到修正。

——傻孩子注

需要强调，这里的启动指令，在连续模式下将启动连续的采样；而在单次采样模式中只会启动一次转换，该次转换完成以后，模块会停止工作，直到下次采样开始标志被置位。

——傻孩子注

● 复选框 **Start conversion**

A/D转换启动位。设置该位将启动一次**A/D**转换，在单次转换的初始化中不需要设置该位。**推荐设置：不打勾。**

● 复选框 **Auto trigger**

自动触发使能。如果该位被设置，则**A/D**模块将被由"**ADC**自动触发事件选择位"（**ADCSRB** 中**ADTS2..0**）所选定的事件自动触发启动。如果选择了该选项，**Start conversion**选项实际就失去了意义。**推荐设置：根据实际需要设定。**

● 复选框 **Internal reference**

使用内部电压基准源，该选项对应**ADMUX**寄存器的**RESF1..0**位，用于指定内部电压基准源作为**A/D**的参考基准源。在**ICC6.31A**中，该选项无效。这一错误在随后的版本中得到了纠正。**推荐设置：根据实际需要设定。**

● 复选框 **Left adjust**

A/D转换结果左对齐，打钩后设置**ADMUX**寄存器**ADLAR**位。**推荐设置：需要8位精度的时候，请使用左对齐。**

● 下拉列表框 **Reference**

用于选择**A/D**模块的参考基准源。在**ICC6.31A**中，该选项无效。这一错误在随后的版本中得到纠正。**推荐设置：根据实际需要设定。**

● 下拉列表框 **Trigger source**

该选项对应于**ADCSRB**寄存器中**ADTS2..0**位的设置，用于指定自动触发模式中触发**A/D**转换的事件源。当前面的**Auto trigger**复选框被选中时，该下拉列表用于选择自动触发的事件源。**推荐设置：根据实际需要设定。过于频繁的A/D采样会消耗大量的处理器时间。**

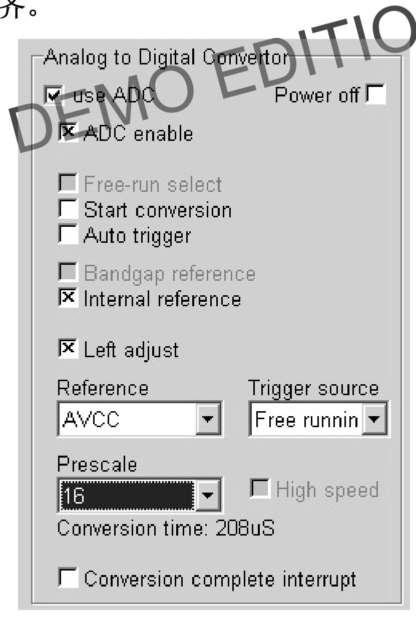


图 2.4.6 ADC 选项卡

IT IS ONLY A DEMO EDITION

● 下拉列表框 **Prescale**

对应于**ADCSRA**寄存器中的**ADPS2..0**位，预分频的大小决定了转换速度和精度。在本实例中，对速度要求不高，故选择了**16**分频。**推荐设置：当列表下方的"Conversion time"字体不为红色时为好。**

● 复选框 **Conversion complete interrupt**

转换完成中断。选择该复选框将使能"**ADC**转换完成中断"，代码

生成器将在代码中加入"**ADC**转换完成中断服务函数"（默认为**adc_isr()**函数），并设置**ADCSRA**寄存器中的**ADIE**位。鉴于本章实例中使用的是中断方式来读取**A/D**转换结果，因此选中该复选框。**推荐设置：根据需要设定。**

第三步，设置完成后单击代码生成器的**Preview**按钮，从弹出的窗口中，查找到我们上面设置所对应的**A/D**的初始化代码；或者直接单击**OK**按钮，直接生成一个**.c**文件。

我们可以发现一个有趣的现象：在**ICC6.31A**版**ICCAVR**中不但代码生成器没有提供**A/D**通道的设置，在**ADC**选项卡中，与**ADMUX**寄存器相关的**Internal reference**、**Left adjust**和**Reference**选项等无论怎样设置，在所生成的代码中，**ADMUX**的值始终都是**0x00**，这可能是**ICC**的作者不给我们这个偷懒的机会吧。没有关系，我们稍后再修改一次生成的代码。

另外**ACSR = 0x80;**语句用于关闭模拟比较器的使能，它等效于**ACSR = BIT(ACD);**，这是由于**Analog Comparator**选项卡中的**Disable**复选框默认处于选中状态造成的。

```
//ADC initialize
// Conversion time: 208uS
void adc_init(void)
{
 ADCSRA = 0x00; //disable adc
 ADMUX = 0x00;   //select adc input 0
 ACSR   = 0x80;
 ADCSRB = 0x00;
 ADCSRA = 0x8C;
 }
```

IT IS ONLY A DEMO EDITION

根据本章实例的需要，参照**Datasheet**，我们将**A/D**的初始化代码修改为：

由于排版的原因，这里所书写的代码与在编辑器里看到的格式可能不太一样。

——老编注

```
ADMUX = 3;          /*选择 AREF,结果右对齐,选择 ADC 通道 3*/
ADCSRA = 0x84;      /*ADC 模块使能,未开始转换,自动触发关闭,中断使能
                      预分频比 16*/
ADCSRB = 0x00;      /*ACME 用于模拟比较器,这里忽略.由于没有使用自
                      动触发功能,对 ADTS 的设置没有意义*/
DIDR0 = 0x08;       /*关闭 PC3 口的数字输入缓冲*/
```

代码生成器同时还生成了"**ADC**转换完成中断服务函数"

```
#pragma interrupt_handler adc_isr:22
void adc_isr(void)
```

```
{
  //conversion complete, read value (int) using...
  // value=ADCL;                 //Read 8 low bits first (important)
  // value|=(int)ADCH << 8;      //read 2 high bits and shift into top byte
}
```

我们可以在这个函数中加入自己需要的中断服务函数代码。代码生成器还给我们建议了一种读取转换结果的方式，这里我们就不再探讨了。

4.9 书写具有工程结构的初始化代码

在前面的章节中，我们曾经介绍了一种工程结构：将硬件资源的初始化函数以及相关的中断处理函数放在一个**.c**文件中，又将所有外部需要调用的函数和变量的声明部分放在一个同名的**.h**文件中，并通过定义宏的方法实现在程序编译时将一些代码或者函数插入到指定的中断处理程序中。这种方式成功地将相对固定的程序硬件初始化部分和相对改动频繁的中断处理程序部分进行了隔离，从模块化和代码重用的角度来说，好处是显而易见的。

仿照前面的方法，我们对代码生成器生成的内容进一步深加工，我们后面的进阶部分的讲解都建立在这一工程结构的基础之上。

IT IS ONLY A DEMO EDITION

第一步，我们使用代码生成器生成我们所需要的代码。在配置界面中，我们直接单击生成器窗口下面的“**OK**”按钮，系统会自动生成一个未保存的文本文件“**Untitled – n**”，这里的**n**由文件的顺序决定。

第二步，我们通过选择菜单**File→New**新建一个空的文本文件，以一个能够说明该文件在工程中结构特征的名称来命名，比如说：**HD_Support.c**，这里**HD**是**Hardware**的缩写。从第一步生成的文件中粘贴函数**void adc_init(void)**、**void adc_isr(void)**。如果我们的**A/D**采用了中断模式，那么还需要粘贴中断向量声明部份：

```
#pragma interrupt_handler adc_isr:22
```

最好再将中断向量号用**iom48v.h**中定义过的宏来替代，提高代码的可移植性：

```
#pragma interrupt_handler adc_isr:iv_ADC
```

一个模块通常由一个.c和一个相关的声明库.h组成。具体内容参见第三篇第一章。

——傻孩子 注

第三步，新建一个与前面的**.c**同名的**.h**文件，并在**.c**文件的首行将该头文件包含进去，以后我们暂时性地将所有的函数和变量都在这个头文件中声明一下，方便工程中其他部分对该模块的调用。例如：

```
# include “HD_Support.h”
```

其实，我们最好把它们连同前面提到过的中断向量声明一起，放到.h文件中。本书后面的章节，就会这样做。

——傻孩子 注

同时，在**.h**文件的首部加上器件的相关声明“**iom48v.h**”，以及包含了系统基本宏的头文件“**macros.h**”。

```
# include “iom48v.h”
```

```
# include “macros.h”
```

第四步，在新建的.h文件中，添加一个宏，名称只要能表达明确的意义即可，单词要记得大写并用下划线隔开。例如：

```
# define INSERT_ADC_ISR_CODE    NOP();
```

复制该宏，将它粘贴到.c文件的ADC中断处理函数中。这样做的意义很明确，就是要在不编辑.c文件的情况下，将一些代码（例如“NOP();”）插入到中断处理程序里面。很容易想到，只要替换.h中宏声明部分“NOP()”所在位置的内容，就可以实现将任意的代码插入到中断处理函数中。我们顺便将采集到的A/D值保存到一个固定的局部无符号整形变量中：

```
void adc_isr(void)
{
    unsigned int ADCValue = ADC & 0x03FF;    //10 位精度的情况
    INSERT_ADC_ISR_CODE
}
```

第五步，进行其它相关的声明函数的编写，例如我们还需要编写一个端口初始化函数，一个总的设备初始化函数，例如：

```
void port_init(void)
{
    //这里需要将 A/D 采样输入端口设置为输入状态，
    //不要开上拉电阻
}

void device_init(void)            //其实只有这个函数才是外部需要调用的
{
    CLI();
    port_init();
    adc_init();
    ……
    SEI();
}
```

IT IS ONLY A DEMO EDITION

第六步，将.c文件中所有函数的声明部分都粘贴到.h文件中；对于全局变量，则需要删除初始化部分，同样粘贴一个到.h中。记得要添加一个extern修饰符。

关于变量的声明和引用，请参照第三篇第一章的解释。

——傻孩子 注

```
extern void port_init(void);         //引用 port_init();
extern void adc_init(void);          //引用 adc_init();
extern void adc_isr(void);           //引用 adc_isr();
extern device_init(void);            //引用 device_init();
```

如果采用的是单次A/D转换模式，我们还可以编写一个宏来专门启动一次A/D采样，看着START_ADC这样的语句总比看到寄存器设置要亲切吧：

```
# define START_ADC      ADCSRA |= BIT(ADSC);   //启动 AD 转换
```

到这里我们就基本完成了A/D相关的初始化代码，并且提供了一个将代码插入到A/D中断处理程序中的接口、一个启动A/D转换的宏（类伪码）。万事俱备只欠东风，使用的时候，只需要通过#include宏来包含上面编写的头文件.h，例如：在包含了main()函数的.c文件里面增加如下的代码：

所为类伪码就是采用各种方式（例如宏）实现的一种编码风格，这种风格中代码看起来很像伪代码。

——傻孩子 注

```
# include "HD_Support.h"
```

同时，将HD_Support.h和HD_Support.c与调用它们的文件放在同一个文件夹下，采用工程编译的方式（ICC中按快捷键F9），就能比较规范地使用A/D资源了。

进阶 阅读

<<阅读提示：进阶阅读着重从软件和工程的角度提供一些阅读材料，众口难调，请酌量添加

4.10 电量计原理概述

对于使用电池供电的微型设备，例如手机和MP3播放器通常需要向使用者提供电池剩余电量的信息，实现这种功能的电路被称为电量计。

在实际应用中，更有效的电量测试方案是使用专用的电量计芯片，例如有专门为锂离子电池配备的，集电量计和保护功能于一体的芯片。读者可在网上搜索到相关信息。

— — DA895

IT IS ONLY A DEMO EDITION

- 电池的容量是如何表示的：

电池的容量（C）是以其能放出的电量来度量的，单位是安·时（AH）。1安·时的容量表示电池以1安培的电流连续放电1小时所释放出的电量。例如5号碱性电池的容量约为2安·时，这表明以1安培的电流持续放电，该电池可以连续使用两个小时。

一块电池以不同的电流进行放电，得到的放电时间长度将大相径庭。一般来说，用小电流或断续放电的方式来使用电池，将大大延长电池的使用时间。放电电流的“大”或“小”是相对于电池的容量而言的，例如1安·时容量的电池以1A的电流进行放电，放电速率就是1C；以0.5A的电流进行放电，放电速率为0.5C。对于可充电电池，其放电电流不能过大，否则将对电池造成伤害。

- 采用查表法修正电压型电量计：

电池的资料来源于超霸（GP）公司网站 http://www.gpbatteries-cn.com.hk

——老编注

电池内所剩余的电量在一定程度体现在电池的电压上，但是电压不是随着电量的减少而线性下降的，图2.4.7为超霸（GP）GP15AU碱性电池以10欧姆负载、连续放电模式下的放电曲线。从图中可以看出，在放电初期，电压很快从1.6V下降到1.4V左右，在大部分放电时间内，电池电压保持在1.4V到1.1V之间，在接近终止电压0.9V时电压的下降速度加快，当电压下降到0.9V以下时，电池的绝大部分能量已经被放出。这就是说，电压与电池电量间的关系，不是简单的线性对应关系，直接用电压线性换算电池的剩余电量并不科学。

为了修正电压与剩余电量间的这种非线性关系，可以使用查表法。

我们在单片机中事先构建一张如下所示电压与剩余电量之间的关系表，通过用电压查表确定剩余的电量。

注意，查表法也是单片机处理非线性关系时常用的方法之一。

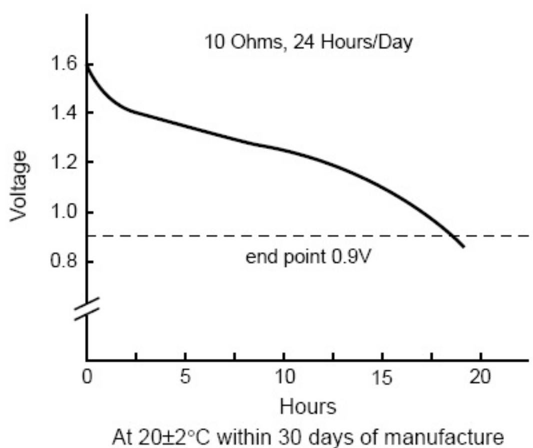


图 2.4.7 GP15AU 电池的放电曲线

电压与剩余电量的关系表

IT IS ONLY A DEMO EDITION

电压值（V）	剩余的电量（%）
1.60	100
1.35	75
1.25	50
1.15	25
0.90	0

本表仅用于说明查表法的基本原理，并非实际测试后的结果。

——DA895 注

4.11 转换结果读取方式的比较

在前面的章节中我们已经了解到：**A/D**转换完成后的结果是分高、低字节分别存放在**ADCH**和**ADCL**两个寄存器中的，我们对**A/D**转换数据的处理通常需要整个转换结果。显然，最适合保存转换结果的数据类型是无符号整型（**unsigned int**型，**2**字节长）。对于**AVR**单片机来说，出于对数据的保护，**ADCH**和**ADCL**这两个寄存器的读取顺序有特殊的要求。下面我们将针对这一问题进行讨论：如何将两字节的转换结果存放进最终的无符号整型变量中。

对于使用汇编编写的程序，不存在整型变量的说法，因此我们就不再赘述了。

——傻孩子注

- 利用编译器直接读入到无符号整型变量：

利用编译器直接读入是最简单的方法，我们来看如下的代码：

```
unsigned int voltage;
```

数据手册没有明确说明 ADCH 的高 6 位是否保持为 0。但从实际测试的结果看，是这样的，然而为了代码的严谨性，我们加上了“&0x3FF”的操作。

——DA895 注

```
……
voltage = ADC&0x3FF;                    //读入ADCH和ADCL中的转换结果
```

打开头文件“iom48v.h”可以看到“ADC”被定义在与寄存器ADCL相同的地址上。那么编译器是如何将两个寄存器内的数据“转化”为一个整型数据的呢？为了揭开这个秘密，我们打开编译器输出的汇编文件。（汇编文件即*.s文件，“*”为所写的C代码文件名）

在分配变量时，编译器将变量voltage分配到R20和R21中：

阅读编译器输出的汇编文件有一定的难度，初学者如掌握起来暂时有困难，可以先不深究，直接关心分析后得到的结论。

——傻孩子注

```
;           voltage -> R20,R21
```

并将“voltage = ADC;”一句编译为：

```
lds R20,120                  ;ADCL的地址为0x78，即120---老编注
lds R21,120+1                ;ADCH的地址为0x79，即121---老编注
andi R21,3                   ;只取ADCH的低2位
```

可见编译器已经为使用者绕过了变量组装的过程，这也是**效率最高**的读取方法！

- 用共用体组装ADC转换结果：

IT IS ONLY A DEMO EDITION

首先我们来回忆一下C语言中关于共用体的知识。共用体是指若干个变量共享同一段内存空间。如果改变其中的一个变量的值，与它共享内存的其它变量的值也会随之改变。

先来看一个简单的例子。定义如下的共用体：

```
union simple
{
       unsigned char a;
       unsigned char b;
  } value;
```

同样是无符号字符型变量的a和b两个变量却占据同一个字节的RAM空间，这就像我们有的网友在论坛上喜欢使用多个帐号发帖子，其实都是他一个人所为。

如果遇到如下语句：

```
       b = 0x01;
       a = 0x02;
```

由于a和b共享同一个存储空间，改变a变量值的同时，也把b的值给改变了，所以第二句话执行完毕后，b的值实际上也是0x02了。

我们定义如下的共用体：

```
union char_to_int
```

```
{
        unsigned char ad_result[2];
        unsigned int voltage;
}voltage_value;
```

数组**ad_result[ ]** 和无符号整形变量**voltage**之间有着如图**2.4.8**所示的存储空间共享关系，**ad_result[1]**和**ad_result[0]**就像房屋**char_to_int**的两个房间，如果和在一起，就称为**voltage**房间。

我们没有对所有的AVR单片机C编译器做测试，对于是否有不支持直接读取的编译器不得而知，但使用共用体进行数据类型转换，也是一种重要的软件方法。

——老编注

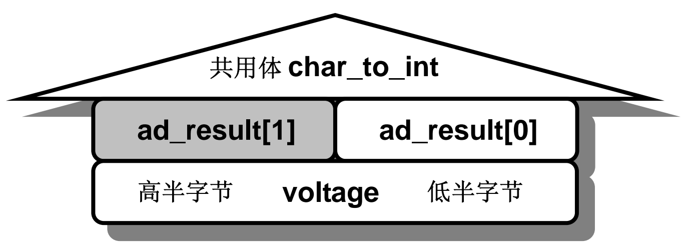


图 **2.4.8** 用共用体组装 **A/D** 转换结果

在从**A/D**读取转换结果的时候，将**ADCL**寄存器的数据放入到变量**voltage_value .ad_result[0]**中，再将**ADCH**寄存器的数据放入到变量**voltage_value .ad_result[1]**中。注意**ADCH**只有低**2**位有效，我们用**ADCH**与上**0x03**（**ADCH&0x03**）来提取它们。

IT IS ONLY A DEMO EDITION

在取**ADC**转换数据时则使用**voltage_value.voltage**变量，将组装成无符号整型的转换结果一次性取出。代码片段如下：

```
voltage_value.ad_result[0] = ADCL;              //读入ADC转换结果低位
voltage_value.ad_result[1] = ADCH&0x03;         //读入ADC转换结果高位
```

所谓"间址寻址"就是间接地访问某一地址。这就好比要打开一个抽屉A还需要从另外一个抽屉B中找到这个抽屉A的钥匙——好在我们手上有另外一个抽屉B的钥匙——这个钥匙就是间址寻址的关键参数。

——傻孩子注

在处理数据时则使用**voltage_value.voltage**变量。让我们再来关心一下编译器是如何处理这两行语句的：

```
;       voltage_value.ad_result[0] = ADCL;          //读入ADC转换结果低位
 lds R2,120
 std y+0,R2
;       voltage_value.ad_result[1] = ADCH&0x03;     //读入ADC转换结果高位
 lds R24,121
 andi R24,3
 std y+1,R24
```

显然，由于使用了数组，编译器使用了**Y**指针来做寄存器**间址寻址**，这实在是"小题大做"，既浪费了资源，又降低了速度。

- 强制类型转换和位操作组装**ADC**转换结果：

在前面我们已经看到，代码生成器在生成"**ADC**转换完成中断服

务函数"时建议了一种用强制类型转换和位操作来实现的结果组装方式，我们对它稍微做一点修改。

```
unsigned int voltage;
……
voltage = ADCL;
voltage |= (unsigned int)ADCH<<8;
```

首先将**ADCL**直接赋值给两字节长的无符号整型变量**voltage**，这相当于填充了变量的低字节。对**ADCH**，首先用**(unsigned int)ADCH**语句将它"当做"一个无符号整型来操作，再用"**<<8**"将它移位到高字节去。我们同样来看一下编译器对这种写法是如何处理的：

因为寄存器 ADCH 值是一个 unsigned char 型数据，如果直接进行移位，就会溢出了。书写表达式时，进行类型对齐是一个好习惯。

——傻孩子注

```
;       voltage = ADCL;
 lds R20,120                    ;ADCL的地址为0x78，即120---笔者加注
 clr R21                        ;清空高半字节
 .dbline 37                     ;这是编译器定义的标号
;       voltage |= (unsigned int)ADCH<<8;
 lds R2,121                     ;ADCH的地址为0x79，即121
 clr R3                         ; R3被用作变量
                                ;(unsigned int)ADCH高半字节
 mov R3,R2
 clr R2
 or R20,R2
 or R21,R3                      ;用或逻辑方式组装高、低字节
```

IT IS ONLY A DEMO EDITION

不同的编译器甚至对于编译器不同的版本，对同一句 C 语句的编译可能不甚相同，我们这里仅就当前的 ICC 编译器进行讨论，这些讨论结果不能简单地"推广"到其它的编译器。

大家可以按照这里介绍的方法，分析其它编译器的编译特点，进而选择效率最高的代码书写方式。

——傻孩子注

由上面的分析我们可以看出，如果编译器能够直接将转化结果输出到整型变量中，这无疑是最方便快捷的方式；如果编译器不能识别这种操作，则可以用共用体来实现转换结果组装，但程序效率大为下降，至于用强制类型转换和位操作的方式，由于编译器将变量扩展为**int**型进行运算，因此效率更低。

4.12 ADC噪声抑制模式

"数字"和"模拟"是一对兄弟，虽然各有所长，但是放到一起难免会"磕磕碰碰"。单片机片内的**ADC**是这两兄弟"面对面" 的典型例子。既然大家在空间上已经分不开了，那么让我们在时间上给他们一点自由，使得**ADC**能够工作得更好。

- 休眠功能的开启和关闭：

为了介绍**ADC**噪声抑制模式，我们先要简单介绍一下**AVR**的休眠模式。

休眠是指单片机停止全部或者部分外设，并暂时停止运行程序，直到有某种事件（例如外中断或串口中断）发生，再将 CPU 唤醒，以处理该事件的过程。

——DA895 注

在低功耗要求特别突出的今天，除非特别需要，单片机不会始终在全速模式下运行，因为高速运行会消耗更多的电能，这就像我们PC机的CPU，随着运算速度的提高，功耗也越来越大一样。现代的单片机几乎都支持休眠功能。

SMCR寄存器控制着AVR单片机CPU的休眠模式。当设置好SMCR寄存器的值后只要一执行SLEEP指令，CPU就会转入相应的休眠模式、关断部分资源的驱动时钟并停止程序的运行。这样可以大大减少消耗的电流。从Datasheet中“电源管理及休眠模式”一章Table18的表中可以查到SMCR寄存器设置所对应的休眠模式，我们将该表重绘如下：

表2.4.2 SMCR寄存器设置与休眠模式

SM2	SM1	SM0	休眠模式
0	0	0	空闲模式
0	0	1	ADC噪声抑制模式
0	1	0	掉电模式
0	1	1	省电模式
1	0	0	保留
1	0	1	保留
1	1	0	Standby模式（注）
1	1	1	保留

注：仅在使用外部晶体或谐振器时Standby模式才可用。

IT IS ONLY A DEMO EDITION

例如某CPU仅在串口收到数据后对数据进行处理而没有其它工作需要做，它可以启动“空闲模式”，在没有数据到来时，CPU处于“睡眠”状态，当串口接收到数据后，串口接收完成中断会唤醒CPU来处理数据，当CPU处理完当前数据后，继续低功耗的“睡眠”状态。

老编语：起来起来工作了，读者反馈回来了，赶快修改稿子……
众小编：怎么又来了……才睡一小会……
大家打着哈欠，七手八脚把文件分类、修改妥当，不一会，编辑部里又鼾声一片了。

——跑龙套注

在这里，我们使用了一种特殊的休眠状态“ADC噪声抑制模式”，在这种模式中，已设置好的A/D转换过程可以继续进行，而CPU停止运行程序，ADC、外部中断、两线串行口（TWI）的地址匹配、定时器/计数器2和看门狗也可以继续工作。注意这里所说的“继续工作”是基于相关的模块被打开的情况而言的，模块的使能由“功耗抑制寄存器”（PRR）和其各自的控制寄存器控制。

为防止CPU意外进入休眠模式，SMCR寄存器中设置了“休眠使能”（SE）位，当且仅当该位被设置为“1”时，SLEEP指令才能让CPU转入休眠模式。这就好比“不见鬼子不挂弦”，多了一重保险。

我们使用如下的代码，仅当执行SLEEP语句前才设置SE位。

```
SMCR |= 0x01;            //使能休眠功能
asm("SLEEP");            //休眠CPU,进入ADC噪声抑制模式
```

而在CPU被唤醒后立即清除SE位。

```
SMCR &= 0xFE;            //关闭休眠功能
```

注意：因为休眠是依靠中断唤醒的，所以在进入休眠模式前，必须使能ADC完成中断（ADCSRA中ADIE位）和总中断（可使用SEI();语句），否则CPU在进入休眠模式后就再也不会醒过来了。

● ADC噪声抑制模式下的转换启动：

ADCSRA寄存器中的ADSC位用于启动一次A/D转换，在连续转换模式下，置位该位将启动第一次转换，而在ADC噪声抑制模式下，A/D的转换并不需要由设置该位来启动。

在设置ADCSRA时仅设置了ADC使能、中断使能和预分频而不设置该位。

```
ADCSRA |= 0x8F;
        //ADC模块使能,未开始转换,自动触发关闭,中断使能,预分频比128
```

A/D转换将在执行了SLEEP指令进入休眠模式后自动开始。

4.13 A/D的软件滤波技术

如果我们将采样得到的A/D值用数码管显示出来，通常会发现，它的最后两位会飞快地跳动以至于看起来就像是两个8字，而前面几位则较为稳定，非常奇怪是吗？

对我们来说，看到数字的跳动，除了让第一次使用A/D的你兴奋得高呼“我成功啦！”以外，是否还会让你对看不清最后几位的确切数值感到恼火？电压表明明告诉了我们被测点有一个稳定的电压，为什么采用A/D却无法获得同样稳定的结果呢？

为什么末尾数字会发生跳动呢？原因很简单，在我们A/D的输入端上除了需要测量的电压之外，还有不受欢迎的朋友——干扰。本节所介绍的滤波算法，会把这些朋友从我们的转换结果中“请”出去。

IT IS ONLY A DEMO EDITION

滤波可以分为两大类：硬件滤波和软件滤波。其中，软件滤波具有一些硬件滤波所没有的有趣特性：

- 他们通常不需要复杂的外围电路支持，但是，很多经典的算法都来源于对外围电路功能的模拟。例如，平均值滤波技术就是对电容滤波的简单模拟。
- 因为脱离了硬件的束缚，软件滤波的很多算法要灵活许多，原理简单，容易理解。但往往随着滤波程度的加深，处理速度和资源开销都会成倍地增长。
- 软件滤波技术通常建立在数据结构元素“队列”的基础之上。队列的使用及使用技巧，往往决定了软件滤波算法的效率和效果。

在介绍常见的滤波算法之前，我们先来复习一下队列。所谓队列，是一种数据结构。它模拟了人们“排队”的过程，因此也具有“排队”才有的特性，例如：

- 讲究先来后到：排队是绝对不允许插队的，讲究先来的人先

排入队。简单地说，就是“先进先出”，对应的英文缩写为FIFO（First In First Out）。

- 排队只能从后往前排：一个队伍，总有一个排头、一个排尾。排队的人从队伍的尾部开始排队，最后从队伍的头部完成排队过程。因此，队列数据结构中，有且只有一个头部Head和一个尾部Tail。数据从尾部进入，从头部被取出。

不过，数据结构中的队列毕竟是对现实生活中的排队现象进行了一些抽象和改造。因此，二者也存在一些不同点，只要我们注意比较一下它们的不同点，队列结构也就不再是难点了。

- 数据结构中，队列的长度既可以是固定的，也可以是变长的；实现这两种不同特性的具体方法并不相同；而现实中，排队往往是没有限制的，从几米到几公里的长龙我们都司空见惯。
- 数据结构中，逻辑上的队列可以和现实中的队列一样是一字摆开的（线性的）；也可以是环形的，称为环形队列。将一个普通队列首尾相接，就是环形队列。
- 数据结构中，绝对不允许插队（普通队列）；现实生活中有些人排队时愣是装作不知道规则而去故意违反。

有一种队列叫做加权队列，按照一定顺序插队就是它的特征。大家可以参照相关的数据结构书籍。

——傻孩子 注

以上我们简单介绍了一下队列的概念，更多内容将在本书**第三篇第二章（“来自身边的启示”）**中作详细的介绍。如果您是第一次接触这一概念，或者对这一概念很感兴趣，可以先阅读这一章节。

IT IS ONLY A DEMO EDITION

好了，言归正传。下面我们从C语言的角度介绍几种常见的软件滤波方法。

首先，我们来看看几种滤波算法的一些特点：

- 滤波算法只有“队列类”和“过滤类”两大基本类别。
- 将以上的两大类别进行结合，将产生新的算法。
- 设计算法，请依照以上两条。滤波算法是无穷尽的。只需要掌握两大类别算法中的一些基本算法即可。

- 滤波算法的两大基本类别

众多的滤波算法其实都可以简单地划分为两大类，或者可以认为是这两大类当中某些基本方法的结合。这两类算法间有着显著的区别。

“队列类” 算法。它着重于：将收到的数据以队列的形式组织起来进行存储和加工。在这里，**队列对于数据的存储和缓冲作用并不是“队列类”算法的独有特征。利用队列的组织形式对一组数据进行加工才是该算法的关键**。这类算法通常采用一个定长的队列来存放数据，一旦从数列的尾部“添加”一个新数据，就会从头部“挤”出一个旧数据。（如图2.3.9所示）加入新数据后，整个队列会进行一次运算，例如求取队列的平均值、进行队列排序并选取中间的数值等等。经过

计算后得到的数值可以作为新的结果被存放到专门的变量中或添加到另外一个用于记录变化趋势的队列中。这种算法通常用于滤除周期性的干扰，系统开支大、速度慢、系统灵敏度低是这类算法通常需要面对的问题。

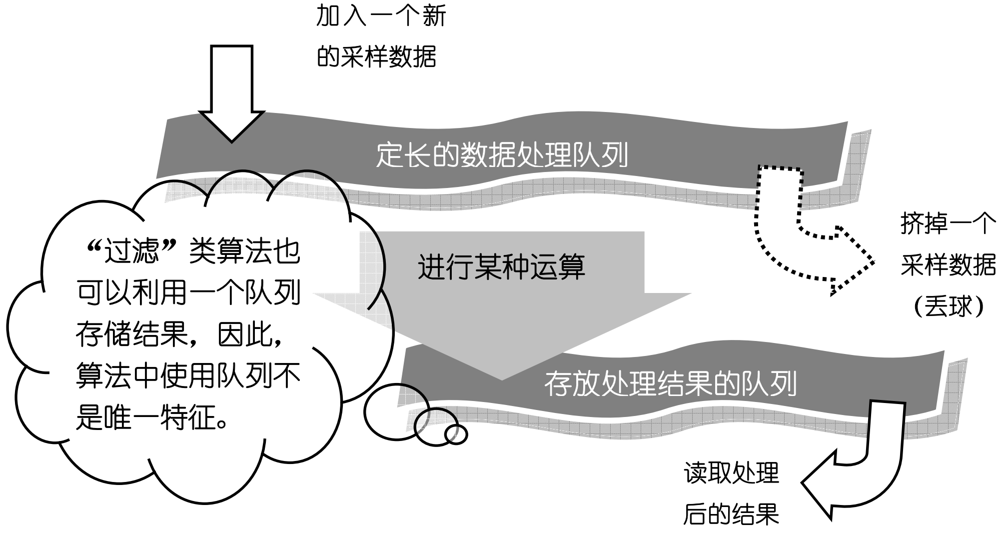


图 2.4.9 “队列类”滤波算法特征示意图

“过滤类”。顾名思义，会根据事先设定的一些条件，对输入的数据进行筛选、过滤，剔除某些数据。常用的过滤条件有：数值的范围前后两个数值的最大差值、某种特征序列等等。这是一类基于条件判断的算法。当指定的条件满足时，数据被接受并得到处理；条件不满足时，则根据一定原则选取替代品——比如利用上一个合格的数据来替代等等。这类算法根据过滤条件的不同可以继续划分为“数据挑选类”和“数据排除类”等等。

IT IS ONLY A DEMO EDITION

● “队列类”算法中常见的基本算法

■ 算术均值滤波

［算法描述］

所谓算术均值滤波，就是在每收到了固定数量的采样数据以后就计算一次这些数据的平均值，并将该平均值作为处理后得到的结果输出。完成一次计算以后，通常将存储采样数值的队列清空，并将计数器清零，开始下一轮的处理。

［C语言实现］

在前面的**4.9**节“书写具有工程结构的初始化代码”中，我们已经可以通过定义宏**INSERT_ADC_ISR_CODE**将我们需要的代码插入到中断处理程序中。同时，中断处理程序中也为我们提供了一个变量**ADCValue**，其中保存着读取到的采样数据。基于以上事实，我们可以编写一个函数来放置我们的数据处理代码：

```
void Insert_ADC_ISR_Code(unsigned int ADCValue);
```

并通过定义宏：

```
# define INSERT_ADC_ISR_CODE Insert_ADC_ISR_Code(ADCValue);
```

将该函数插入到中断处理程序中。今后如果没有特殊说明，我们所使用的代码或者函数就是按照上面的方式进行组织的。

下面我们来看看如何使用C语言实现“算术均值滤波”。

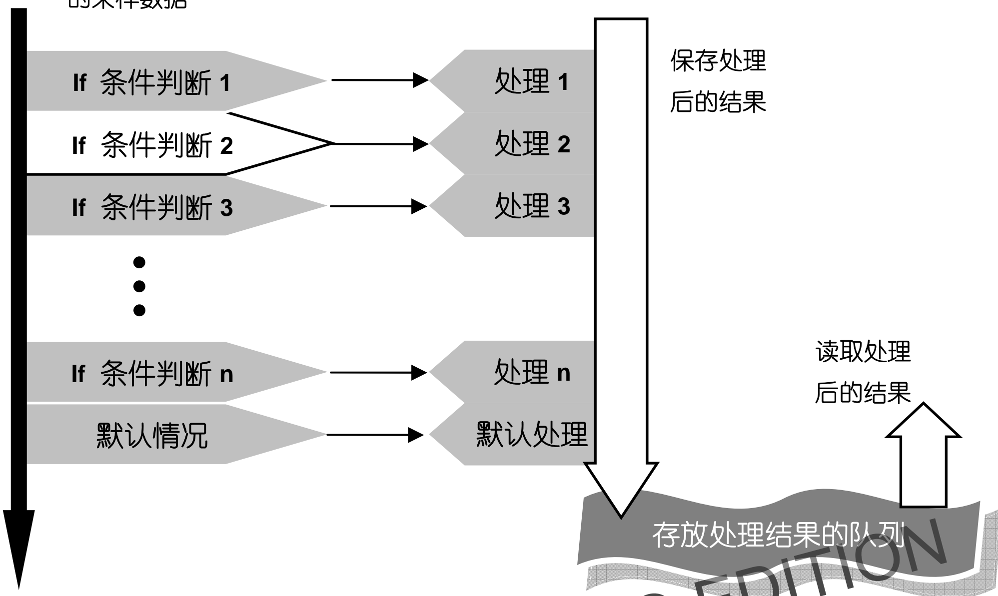


图 2.4.10 “过滤类”滤波算法特征示意图

IT IS ONLY A DEMO EDITION

```
//全局变量声明
unsigned int g_wADValue;            //用于保存结果的变量

void Insert_ADC_Isr_Code(unsigned int ADCValue)
{
  static unsigned int s_wADCBuffer[ADC_BUFF_SIZE];
  static unsigned char s_cBufferCounter = 0;
  static unsigned long s_dTotal = 0;

  //将A/D采样值加入到数组中
  s_wADCBuffer[s_cBufferCounter] = ADCValue;
  s_dTotal = s_dTotal + ADCValue;

  s_cBufferCounter++;
  //判断是否已经获得足够数量的数据
  if (s_cBufferCounter == ADC_BUFF_SIZE)
  {
      //如果已经获得足够数量的数据
      //进行一次平均值计算，并对计数器清零
      //将计算结果保存在一个全局变量中
      g_wADValue = (unsigned long)s_dTotal / (unsigned
```

阅读这里的代码，你也许会发出疑问“如果每次都将 s_dTotal 初始化为 0，那么 s_dTotal 还怎样记录总数？”，

注意到前面的 static 了么？你可以先阅读一下第三篇第一章中关于静态局部变量的内容。

——傻孩子 注

注意这里的斜杠是除号哦。

——傻孩子 注

```
long)ADC_BUFF_SIZE;
        s_cBufferCounter = 0;
        s_dTotal = 0;
  }
  } //End Of Insert_ADC_Isr_Code
```

［算法优化］

从代码执行的效率上看，随着所设定的数组尺寸的增大，输出一次结果的时间间隔就会增长——但是对于执行一次该函数所用的时间开销来说（时间复杂度）却是恒定的。我们无法从数组的尺寸上做文章来提高代码的执行效率。

在整个函数中，最复杂的运算就是求平均值的除法运算。对8位单片机来说，如果没有硬件除法器，就需要庞大的额外的代码来完成一个哪怕最简单的、能够整除的整数除法运算。

有没有办法能够取代该运算呢？答案是肯定的。

首先，需要明确这样一个事实：对于一个二进制数，将它向右移动1位（低位丢弃，高位补零）就相当于进行了一次除以2的运算。以此类推，移动n次就可以完成除以2^n的操作。查阅指令表会发现，对一个8位二进制数（1字节）进行右移只消耗1个指令周期。虽然不能简单地认为对于占两个字节的无符号整型数（unsigned int）进行一次移位操作消耗2个指令周期，但是这种操作肯定比进行一次除法运算效率要高得多。这个问题就简单了：我们只需要将ADC_BUFF_SIZE定义为2的n次幂，进行平均值计算的时候，只要将总和向右移动n位，就完成了等效的除法运算。

在附录五中你同样也可以找到移位操作的相关说明

——DA895 注

IT IS ONLY A DEMO EDITION

说干就干。我们先修改宏ADC_BUFF_SIZE：

```
//这里定义的数字为2的n次幂的n的大小
# define ADC_BUFF_SIZE_BIT_COUNT        0
# define ADC_BUFF_SIZE (1<<ADC_BUFF_SIZE_BIT_COUNT)
```

这样，当我们定义ADC_BUFF_SIZE_BIT_COUNT为3的时候，ADC_BUFF_SIZE的值就相当于2^2，即8。

接下来，我们对函数体进行优化，将求平均值部分的代码修改为：

```
g_wADValue = (unsigned long)s_dTotal >> ADC_BUFF_SIZE_BIT_COUNT;
```

大功告成！重新编译代码，你会发现编译后的代码尺寸小了很多。另外，如果限定ADC_BUFF_SIZE的大小不会大于64的话，我们还可以进一步优化代码，将保存总和的变量s_dTotal改为无符号整型，这样又能节省很大一部分系统开支。需要注意一点：根据匈牙利命名法，变量名也要相应调整为s_wTotal了。剩下的代码优化工作就留给大家作为练习，这里就不再赘述。

关于匈牙利命名法，大家可以从第三篇第四章中找到较为详细的介绍。

——傻孩子 注

■ 滑动窗口均值滤波

［算法描述］

所谓的滑动窗口均值滤波，实际上是对算术均值滤波的一种改进。我们注意到，算术均值滤波中，每采集一固定数量的数据才进行一次

处理、输出一次数据，显然原本就很离散的数据经过这样处理以后，输出的结果在时间上的间隔就更大了。为了修正这种扩大的时间间隔，我们对算术均值滤波进行小小的改动：将“每采集固定数量的数据就进行一次处理”修改为“每采集一个数据就对整个队列进行一次处理”。也就是说，每次处理以后我们并不清空队列。这种处理方式，仿佛是有一个的“窗口”在由采样数据组成的“风景”上划过——就和我们坐火车一般，所以被形象地称为滑动窗口均值滤波。

［C语言实现］

基于上面的描述，我们对前面的代码做一下小小的修改：

```
//全局变量声明
unsigned int g_wADValue;            //用于保存结果的变量

void Insert_ADC_Isr_Code(unsigned int ADCValue)
{
   static unsigned int s_wADCBuffer[ADC_BUFF_SIZE];
   static unsigned char s_cBufferCounter = 0;
   unsigned long s_dTotal;                    //注意这里，不再需要静态类型了
   unsigned char n = 0;

   //将A/D采样值加入到数组中
   s_wADCBuffer[s_cBufferCounter] = ADCValue;

   //计算总和
   s_dTotal = 0;
   for (n = 0;n < ADC_BUFF_SIZE;n++)
   {
      s_dTotal += s_wADCBuffer[n];
   }
   //计算平均值，并输出到一个固定的变量中
g_wADValue = (unsigned long)s_dTotal >>ADC_BUFF_SIZE_BIT_COUNT;

   s_cBufferCounter++;
   //判断队列指针是否达到边界
   if (s_cBufferCounter == ADC_BUFF_SIZE)
   {
    //队列指针到达边界，清零
      s_cBufferCounter = 0;
   }
} //End Of Insert_ADC_Isr_Code
```

“+=”是一个双目运算符。A += 2 实际上就等同于 A = A + 2。

——傻孩子 注

IT IS ONLY A DEMO EDITION

［算法优化］

上面的算法已经对除法运算进行了优化，方法和算术均值滤波的优化方式是相同的。注意到该算法的时间复杂度仍然是常量：每次都

要对整个数据进行一次求和运算和一次取平均值的等效除法运算。这个过程中，对数组的求和运算，其时间开销会随着数组的尺寸增大而增长，不仅如此，耗费了大量时间后得到的结果，我们却将其丢弃了，实在可惜。如果能将计算的结果保存到下一次，并只针对变化的部分将该结果进行更新就好了。

单纯将 s_dTotal 修改为 static 类型是没有用的，因为下一次该变量的值会被清零

——傻孩子 注

还记得队列的特征么？除了先来后到，不允许插队之外，还有一个非常重要的特征：新的数据总会添加在尾部，而旧的数据将会从头部被取出。也就是说，位于中间的数据并没有改变，变化的只是头部和尾部的数据！这实在是一个令人振奋的发现。那么，新的优化算法也似乎明朗了。

将道理想明白和实际去做还是有很大差别的。当我们开始修改代码的时候会发现：程序实现的队列中，似乎并没有所谓的头部或者是尾部，那么针对头尾数据的操作就更无从谈起了。请等一下，你注意这样一个变量了么？**s_cBufferCounter**在程序中是依靠修改它的值来实现对数组元素的逐个访问的。到了**数组尾部**，它的值又会被修改为**0**。这个过程对大家来说并不难理解。问题在于，当我们利用这个变量作为数组下标访问数组的时候，实际发生了一些容易被忽略的事情。考察下面的代码：

```
s_wADCBuffer[s_cBufferCounter] = ADCValue;
```

它实际上等效于：

```
s_wADCBuffer[s_cBufferCounter] =0;                //(1)
s_wADCBuffer[s_cBufferCounter] = ADCValue;        //(2)
```

IT IS ONLY A DEMO EDITION

我们注意到，**(1)**式的功能是将**s_cBufferCounter**所指向的单元清空，也就是丢弃。这似乎和队列中头所做的操作是相同的；**(2)**式的功能是在**s_cBufferCounter**所指向的单元中放入新的数据，这也和队列尾部所要做的添加操作类似。难道这一句话就等同于两个队列操作？如果是这样，那么队列的头部和尾部总在一起，这又是为什么呢？

要解释这个现象，就牵涉到环形队列的知识了。其实在不知不觉中我们已经使用了环形队列——将一个队列的首尾相接就是环形队列，回忆代码：

这里介绍的队列知识在第三篇第三章都有详细讲解。

——傻孩子 注

```
//判断队列指针是否达到边界
if (s_cBufferCounter == ADC_BUFF_SIZE)
{
    //队列指针到达边界，清零
s_cBufferCounter = 0;
}
```

做的就是这样的工作。如果队列是环形的，什么情况下队列的头部和尾部是重合的呢？

第一，**当初始状态，队列中没有任何数据的时候，头部和尾部就是一体的**。因为队列的尾部总是指向第一个可以添加数据的单元，同时队列的头部总是指向第一个有数据的单元。队列尾部和头部之间的绝对差值实际上就反映了队列中现有数据的数量。当队列头部和尾部重合的时候，意味着什么呢？至少包含队列为空的情况（二者相等，

差值肯定为**0**）。

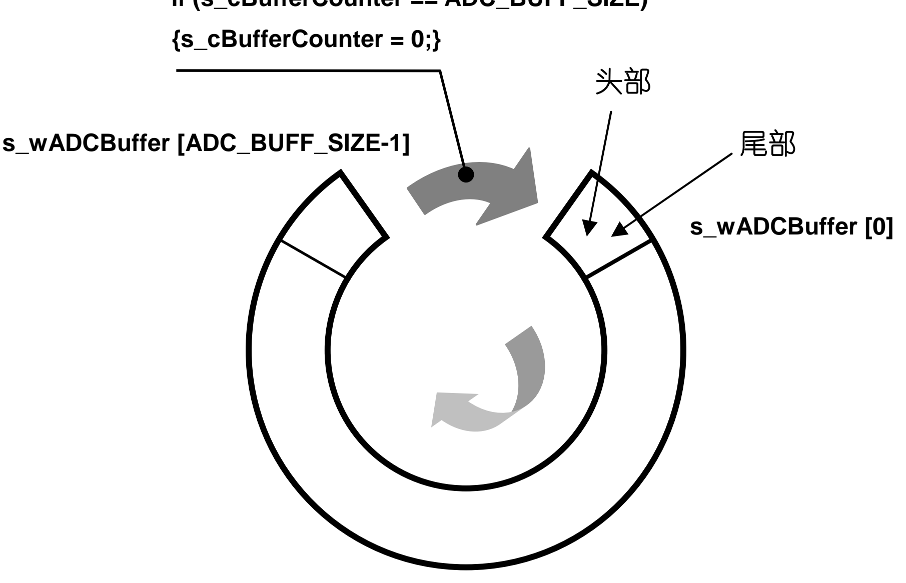


图 **2.4.11** 环形队列示意图（队列为空）

第二，**当队列为满的时候，头部和尾部也是重合的**。此时，虽然尾指针尝试指向尾部的第一个空的元素，但由于队列空间已满，没有空闲单元，同时考虑到队列的环形逻辑，这一指针只好与头指针相重合了。这个时候，队列中元素的个数为“队列的大小”即**ADC_BUFF_SIZE**是该情况区别于第一种情况的特征。

IT IS ONLY A DEMO EDITION

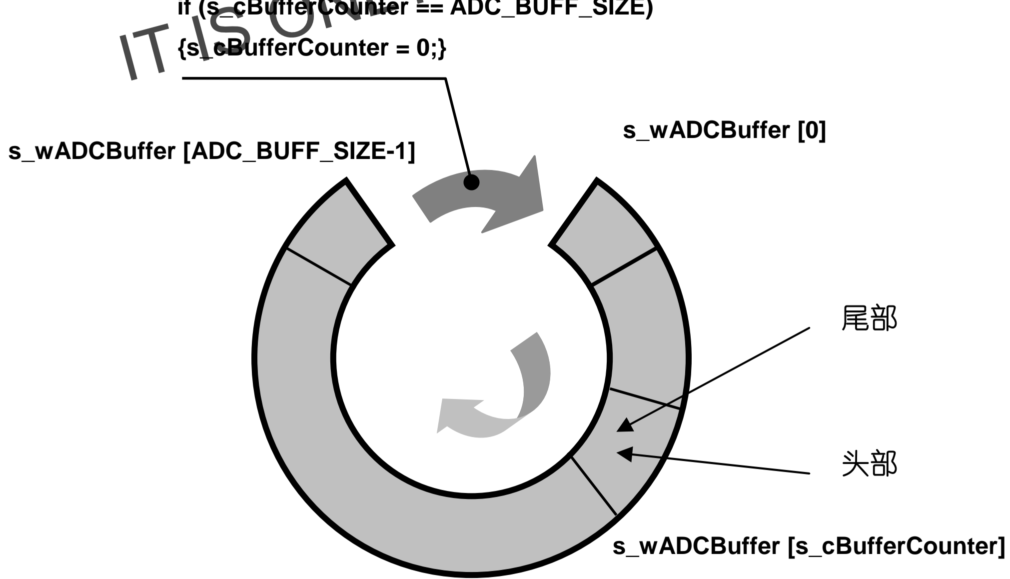


图 **2.4.12** 环形队列示意图（队列为满）

在本算法中，我们采用的是环形队列，并且假设队列总为满的，即便在初始状态下，队列中所有元素都为**0**，总和为**0**也是满足这种假设的。所以，我们只需要一个指针来同时代表头指针和尾指针。使用该指针时，心中要将其不同时刻履行那种指针的职责明确下来。基于

以上事实，优化代码如下：

```
//全局变量声明
unsigned int g_wADValue;          //用于保存结果的变量
static unsigned int s_wADCBuffer[ADC_BUFF_SIZE];

void Insert_ADC_Isr_Code(unsigned int ADCValue)
{
  static unsigned char s_cBufferCounter = 0;
  static unsigned long s_dTotal = 0;  //注意这里，再次变为静态

  //从总和中删除头部元素的值，履行头部指针职责
  s_dTotal -= s_wADCBuffer[s_cBufferCounter];

  //将A/D采样值加入到数组中，履行尾指针职责
  s_wADCBuffer[s_cBufferCounter] = ADCValue;
  //更新总和
  s_dTotal += ADCValue;

  //计算平均值，并输出到一个固定的变量中
  g_wADValue = (unsigned long)s_dTotal /
  (unsigned long)ADC_BUFF_SIZE;

  s_cBufferCounter++;
  //判断队列指针是否达到边界
  if (s_cBufferCounter == ADC_BUFF_SIZE)
  {
    //队列指针到达边界，清零
    s_cBufferCounter = 0;
  }
} //End Of Insert_ADC_Isr_Code
```

IT IS ONLY A DEMO EDITION

■ 中值滤波

［算法描述］

中值滤波实际上只是滑动窗口均值滤波的一个姊妹算法。他们都使用滑动窗口机制，唯一不同的是，中值滤波并不对队列中的元素进行算术平均，而是将其排序，取中间值作为结果输出。影响该算法效率的关键因素就是排序算法的效率。由于篇幅的限制，这里就不再赘述了。

互联网上有很多优秀的排序算法，有兴趣的话可以研究一下，本文就以冒泡算法为例进行说明了。

——傻孩子 注

［C语言实现］

在程序实现的结构上，它和滑动窗口均值滤波类似，都使用了环形队列作为处理手段。不同的是，中值滤波的队列长度（数组尺寸）

通常为奇数——这便于取出排序后的中间值。从演示的角度出发，我们简单地使用冒泡排序作为排序算法。

```
//全局变量声明
unsigned int g_wADValue;          //用于保存结果的变量

//冒泡排序函数，返回排序后的中间值
unsigned int ADCompositor(unsigned int *wBuffer)
{
BOOL bIfSwaped = TRUE;   // BOOL型实际上是unsigned char
unsigned int wTemp = 0;
unsigned char n = 0;

  while(bIfSwaped)             //检测上次是否发生了冒泡交换
  {
    bIfSwaped    = FALSE;   //复位交换标志位
    for (n = 0;n < ADC_BUFF_SIZE - 1;n++)
    {
        if (wBuffer[n] < wBuffer[n+1])
        {
            //对两个数值进行交换，冒泡
            wTemp = wBuffer[n];
            wBuffer[n] = wBuffer[n+1];
            wBuffer[n+1] = wTemp;

            //因为发生了交换，所以设置标志位
            bIfSwaped = TRUE;
        }
    } //End Of For
  //如果完成一轮冒泡都没有发生交换，表明已经排序完成
  } //End Of While

  return wBuffer[ADC_BUFF_SIZE>>1];
} //End Of ADCompositor

void Insert_ADC_Isr_Code(unsigned int ADCValue)
{
  static unsigned int s_wADCBuffer[ADC_BUFF_SIZE];
  static unsigned char s_cBufferCounter = 0;
  static unsigned long s_dTotal = 0;       //注意这里，再次变为静态

  //从总和中删除头部元素的值，履行头部指针职责
  s_dTotal -= s_wADCBuffer[s_cBufferCounter];
```

IT IS ONLY A DEMO EDITION

```
    //将A/D采样值加入到数组中，履行尾指针职责
    s_wADCBuffer[s_cBufferCounter] = ADCValue;

    //进行中值排序，并返回一个结果
    g_wADValue = ADCompositor(wADCBuffer);

    s_cBufferCounter++;
    //判断队列指针是否达到边界
    if (s_cBufferCounter == ADC_BUFF_SIZE)
    {
       //队列指针到达边界，清零
       s_cBufferCounter = 0;
    }
} //End Of Insert_ADC_Isr_Code
```

［算法优化］

中值滤波算法优化的关键在于排序算法的效率。因此，可以从排序算法入手进行优化，本书并非是讨论数据结构和算法的专业书籍，这里就不在深入的讨论了。

- **“过滤类”算法中常见的基本算法**

IT IS ONLY A DEMO EDITION

在本节的一开始，我们通过将A/D采样值显示在数码管上的例子说明了软件滤波的必要性。实际上，这个例子仅仅只说明了“队列类”算法的情况。实际应用当中，“过滤类”算法的特征和“队列类”算法并不相同，它们通常系统反映迅速，灵敏度较高，对“突发性”的随机噪声滤除效果好。合理选取滤波算法是取得良好效果的关键，但是本书并不是专门介绍这个领域的书籍，读者可以查阅其它资料以得到更多信息。

■ 限幅滤波

［算法描述］

限幅算法的基本思想是，对输入数据的大小设置一定的门限，可以是上限、下限也可以两者都有。对于超过限度的输入量，采用一定的替代策略，如截断饱和值（比方说大于0xFF就让它等于0xFF）、使用上一次有效值代替等等。

［C语言实现］

```
//全局变量声明
unsigned int g_wADValue;          //用于保存结果的变量

void Insert_ADC_Isr_Code(unsigned int ADCValue)
{
   if (ADCValue > AD_VALUE_UP_LINE)
```

```
    {
      //如果超过了上限
      ADCValue = AD_VALUE_UP_LINE;
    }
    else if (ADCValue < AD_VALUE_DOWN_LINE)
    {
      //如果超过了下限
      ADCValue = AD_VALUE_DOWN_LINE;
    }

    //返回一个结果
    g_wADValue = ADCValue;
  }
```

［算法优化］

（略）

■ K限幅滤波

［算法描述］

对于一个相对稳定变化缓慢的信号源，采样的结果也应该是相对稳定且变化缓慢的。但是，从采样的结果当中，我们往往发现偶尔会有一两个很强的噪声，在波形中就像小山一样拔地而起，很快又恢复稳定。对于这类噪声，如果应用当中不允许我们使用深度较大的均值滤波，最好的方法就是监测前后两个波形的变化率，也就是斜率K。对于造成过大斜率的数值，我们往往直接丢弃，采用上一次的有效结果作为结果输出。

IT IS ONLY A DEMO EDITION

［C语言实现］

```
void Insert_ADC_Isr_Code(unsigned int ADCValue)
{
    //这个静态局部变量用于保存上次的有效结果
    //需要注意，如果初始化为0，那么很难会有有效输出产生了，
    //所以这里应该填写一个采样电压的典型值
    static unsigned int s_wLastADCValue = 0x01FF;
    unsigned int s_nK = Abs(ADCValue – s_wLastADCValue);

    if (s_nK > AD_K_LIMIT_LINE)
    {
      //如果斜率超过了限度
      ADCValue = s_wLastADCValue;
    }
```

```
    s_wLastADCValue = ADCValue;
    //返回一个结果
    g_wADValue = ADCValue;
} //End Of Insert_ADC_Isr_Code
```

[算法优化]

（略）

■ 一阶滞后滤波

[算法描述]

所谓一阶滞后滤波，实际上就是一个针对前后两个采样结果的加权平均算法。理论上，应该划分进“队列类”滤波算法的范畴，但是从分阶加权求平均的数学计算角度来看，称为“过滤类”算法也不是不可以。实际上，“过滤类”算法的特性，它大都具有——关键就在于这个一阶上了。对于多阶的加权均值滤波，将在后面介绍。

关于权值，该算法是这样规定的：

- 新来的数据有一个较高的权值，上次的数据权值较低；
- 所有权值的和为1。

[C语言实现]

使用C语言处理该算法的时候，存在一个浮点数的处理问题。与除法的对待方法相似，我们采用位移的方法来绕过。

IT IS ONLY A DEMO EDITION

```
//全局变量声明
unsigned int g_wADValue;          //用于保存结果的变量

void Insert_ADC_Isr_Code(unsigned int ADCValue)
{
 //这个静态局部变量用于保存上次的有效结果
  static unsigned int s_wLastADCValue = 0;

  //实现一个1: 3的权值关系
  g_wADValue = (s_wLastADCValue * 2 + ADCValue * 6)>>3;

  s_wLastADCValue = ADCValue;
} //End Of Insert_ADC_Isr_Code
```

[算法优化]

（略）

● 一些常见的综合算法

■ 中值滑动窗口均值滤波

[算法描述]

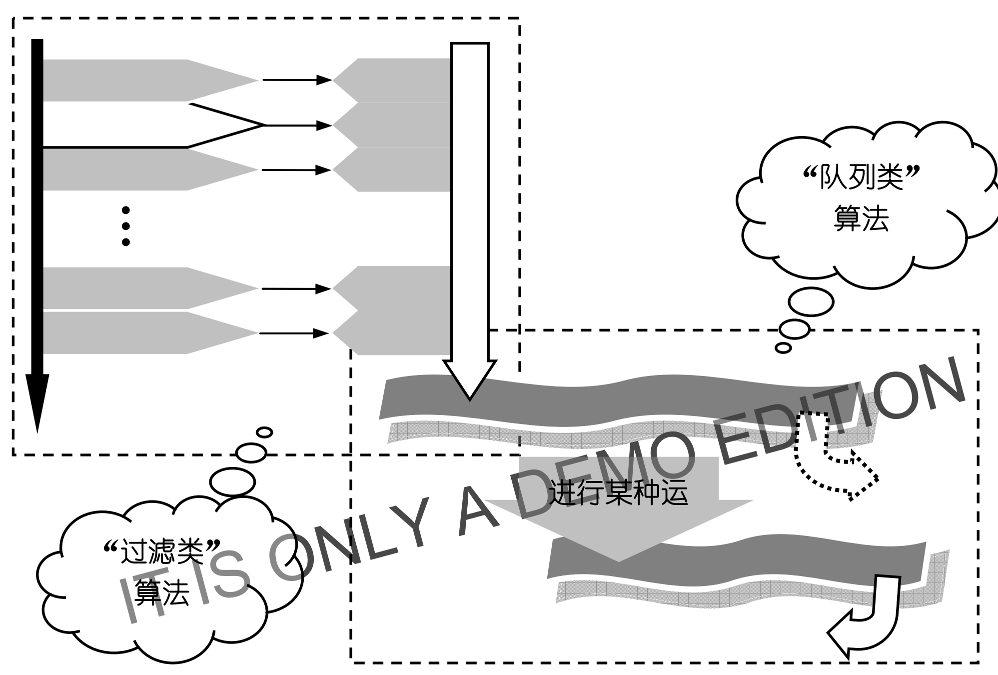
“去掉一个最高分，去掉一个最低分……最后的（平均）得分为**95.55**分”——这就是中值滑动窗口均值滤波算法。

[C语言实现]

（略）

[算法的优化]

（略）



IT IS ONLY A DEMO EDITION

图 2.4.13 综合滤波算法结构特征示意图

■ 限幅/K限幅均值滤波

[算法描述]

当我们先使用限幅或者K限幅算法对输入的数值进行加工以后再简单地使用任意均值滤波算法，就称为“限幅/K限幅均值滤波”算法。

[C语言实现]

（略）

[算法的优化]

（略）

■ 加权（多阶滞后）均值滤波

[算法描述]

该滤波算法就是前面介绍过的“一阶滞后滤波”算法的“多阶推广”。它要求我们在计算的时候，将不同位置的数值乘以不同的权值，从而获得一个新的结果。不同位置的权值不一定按照一定顺序递增或者递减，权值的安排策略不同又会派生出许多子算法。

[C语言实现]

（略）

[算法的优化]

通常，我们回避浮点数的运算，采用先给每个数乘以各自权值对应的整数，最后统一通过位移的方法实现一个等效的除法，获得计算结果。这就要求所有位置的整数权值的和一定要为2的整数次幂。

（示例程序略）

3.14 软件实现的斯密特触发器

在一些智能充电器中，单片机要一直监测电池的电压，一旦超过某一数值，就由恒流充电切换到恒压充电。撇开充电器制作相关的各种知识和概念不说我们集中注意力来考察：当电压接近那个设定的临界值时发生的一些事情。

不管你是否真的见过这种现象，请随我从一个相对较小的时间尺度内“俯下身子”从微观角度来看看发生了些什么：当电压接近临界值的时候，系统还处于恒流充电状态，于是电压开始朝突破临界值的方向上升……在某一时刻，电压突破了临界值，程序检测到了这一数值变化，立即将充电状态由恒流切换为恒压。然而，由于恒流状态时加载在电池上的电压要远远高于恒压时所保持的电压，所以几乎在同一瞬间，单片机监测到的电池电压又跳回了临界值以下——于是，系统又头脑简单地将工作模式切换回了恒流状态……如此反复，在电池电压处于该临界值附近的时候，这样的事情会频繁地发生许多次，直到彻底切换为恒压状态。

以上是从微观角度所假想发生的事情，实际情形也许要更为复杂一些。不过从宏观角度来看，可能发生的现象是：表明充电器充电模式的指示灯频繁地交替闪烁——这不是我们需要的效果。

有什么方法能避免这种令人头痛的事情发生呢？说来很简单，将切换工作模式的临界电压值由一个变为上下两个门限。从恒流切换到恒压状态需要一个较高的门限，这个“较高”是相对原先的单个门限值来说的；从恒压再跳回恒流状态，需要实际电压低于一个相对原先单门限更低的值——高低门限之间的差值至少要大于“当充电器在两种充电模式间切换时造成的前后电压差值的典型大小”。

这里应用的改变充电器充电状态的触发特性，叫做施密特特性。采用软件方式实现这种双门限的触发方式，就称为“软件施密特触发

IT IS ONLY A DEMO EDITION

器”。

编写“软件施密特触发器”的要点是：明确“两态”、“两限”。“两态”是指当前状态处于两种状态中的哪一种。数值当前正从小变大或是从大变小，这都是随机和局部的，并不能以此作为决定当前状态的依据。唯一有资格决定当前状态的就是上一次的状态。根据记录的当前状态决定下一步的监测对象，这是最可靠的方法。简单打个比方来说，假设我们处于恒压状态，那么现在我们需要监测的就是下门限；一旦电压低于了下门限，我们的当前状态就变成了恒流状态。于是，我们只要执着于上门限就可以了。而这里说的“两限”就是上下两个门限。

下面，我们就使用伪代码来描述一下上面例子中提到的解决方案：

```
void Insert_ADC_Isr_Code(unsigned int ADCValue)
{
//这个静态局部变量用于保存上次的有效结果
static BOOL s_blfVoltageMode = FALSE;
……

 if (s_blfVoltageMode)                           //检测当前工作模式
 {
   if (ADCValue < ADC_DOWN_LINE)
   {
     SET_CURRENT_MODEL                           //进入恒流模式
     s_blfVoltageMode = FALSE;                   //修改当前模式标志
    }
}
else
{
   if (ADCValue > ADC_UP_LINE)
   {
     SET_VOLTAGE_MODEL                           //进入恒压模式
     s_blfVoltageMode = TRUE;                    //修改当前模式标志
    }
  } //End Of Else

……
} //End Of Insert_ADC_Isr_Code
```

IT IS ONLY A DEMO EDITION

实例 12 电量计

[需求分析]

掌握了A/D的初步知识后，我们就可以来设计调试光带式电量计的硬件电路及软件了。

电量计通过测量电池的电压来告知用户电池的剩余容量。手机或MP3的电量指示器可以给我们一个很直观的光带式电量计的映像。通常来说，电池两端的电压在一定程度上表征了电池内剩余电量的多少，这里我们先对问题做一个粗略的近似，认为电压与电量呈简单的线性关系，测电量实际上就是测电池两端的电压，在本章进阶阅读部分，我们进一步讨论了这个问题。

针对简化后的问题，电压型光带显示电量计的任务是：

不断测量ADC3（即PC3）端上的电压值，并以光带的形式显示在LED上，电压越高，光带越长（即点亮的LED数目越多）。

电量计软件的工作比较简单，他只重复做两件事情：

- 转换接到端口上的电池电压。
- 处理转换得到的数据并将结果显示为光带。

为了简化起见，我们用一个电位器对电源电压进行分压来模拟电池两端的电压，通过调节电位器，可以看到光带随着分压电压的变化而变化。

应该说明，使用查询方式、中断、自动触发连续转换、噪声抑制模式都可以实现电量计软件的功能，但从任务分析和抗噪声的角度考虑，实现这个软件的最佳方法是使用噪声抑制模式。为了介绍最通用的A/D转换程序，我们在这里将使用中断方式来编写这个软件。在本章进阶阅读部分，我们介绍了噪声抑制模式的使用方法。

IT IS ONLY A DEMO EDITION

[硬件电路]

光带式电量计的电路图见例图12.1，其中与本实验不相关的部分未画出。本实例使用ADC3作为输入，我们用一个电位器RP1来产生一个变化的输入电压，这个电压也可以由其他可调电源代替，但应注意其电压不能高于单片机的电源电压。考虑到实验中可能出现的意外情况，AREF端没有同AVCC端接在一起，而是通过一个0.01uF的电容接到地。AVCC端在本实例中被当作参考电压来使用，在其与单片机的VCC端之间加入了一个RC低通滤波器。从PB0到PB5端口接有6个发光二极管组成光带指示器。

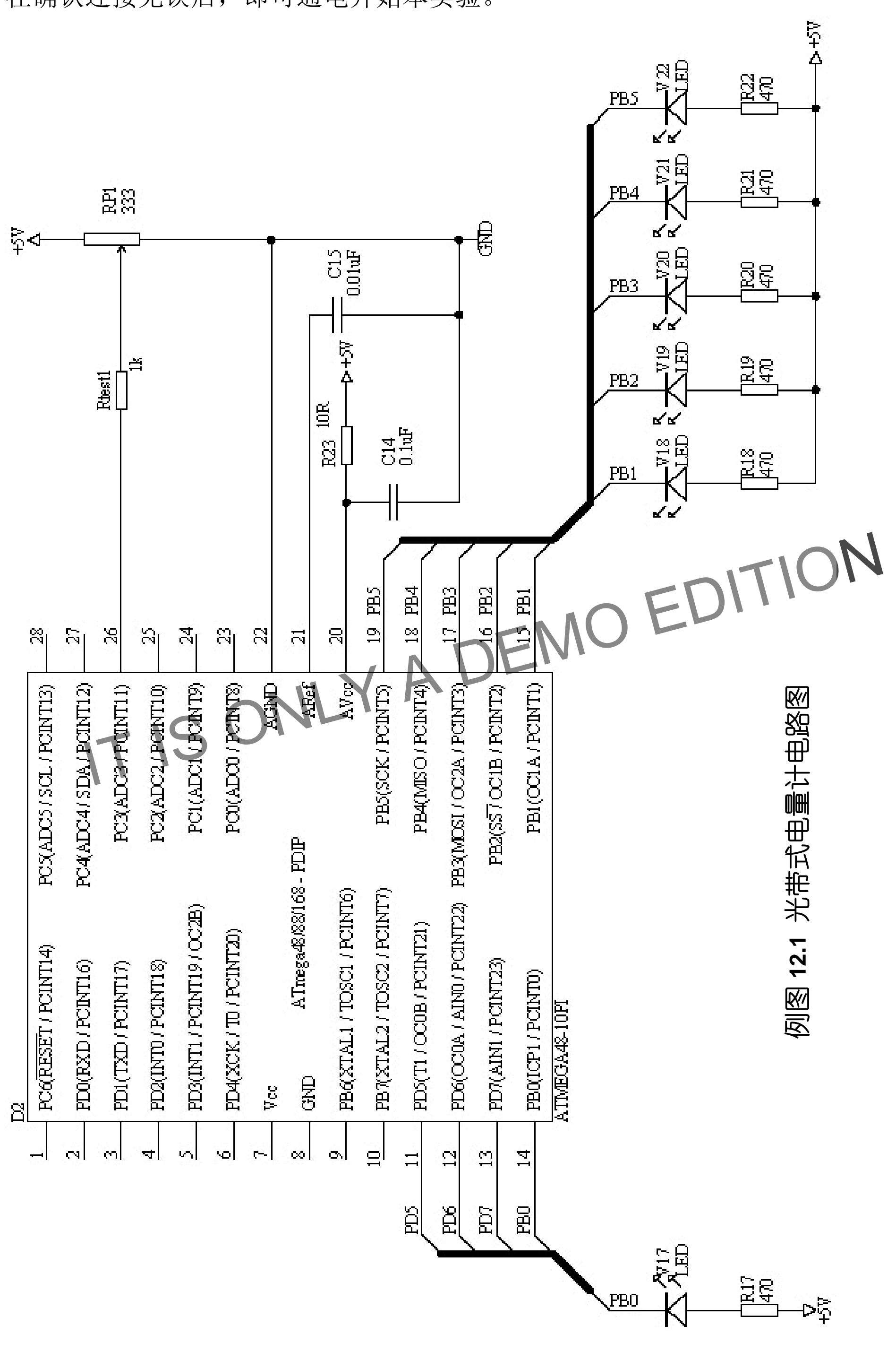
对于使用双龙SL_MEGA8开发实验器来做实验的读者，由于其管脚上没有直接连接发光二极管（LED），读者需要将PB0至PB5口通过导线连接至J6（LEDx8）的L8至L3端。对于跳线的设置，读者可将J9（ADC）

这个原理图是示意性的，有很多情况没有做考虑，例如没有使用复位电路，没有对未使用的空闲端口进行处理，没有对ADC 的输入加低通滤波器等。在正式的工程设计中，这些问题都应该得到重视，而在本书中为简明起见，没有说明这些问题。

——DA895 注

跳线的**ADC3**跳线帽保留，其他**5**个均拔除。保持**J8**（**REF**）跳线帽呈拔除状态，即可完成本实验的跳线连接，时钟等其他跳线的连接方式请参考前面的实例设置，这里不再赘述。

读者可能会发现，**例图12.1**中的参数选值与双龙**SL_MEGA8**开发实验器中的值有所不同，他们都在设计许可的范围之内，均可正常工作。在确认连接无误后，即可通电开始本实验。



IT IS ONLY A DEMO EDITION

例图 **12.1** 光带式电量计电路图

下面我们在前面介绍的实验电路板上实现这个电路：

第 1 步
准备分压电位器

我们采用电位器分压的方式的到一个在**0**到**5V**范围内连续可变的电压来“模拟”电池的电压。

用 **33k** 的电位器进行分压，用以模拟测量的电压。调节电位器即可调整电压值。

Rtest1 电阻，用于保护单片机口线。

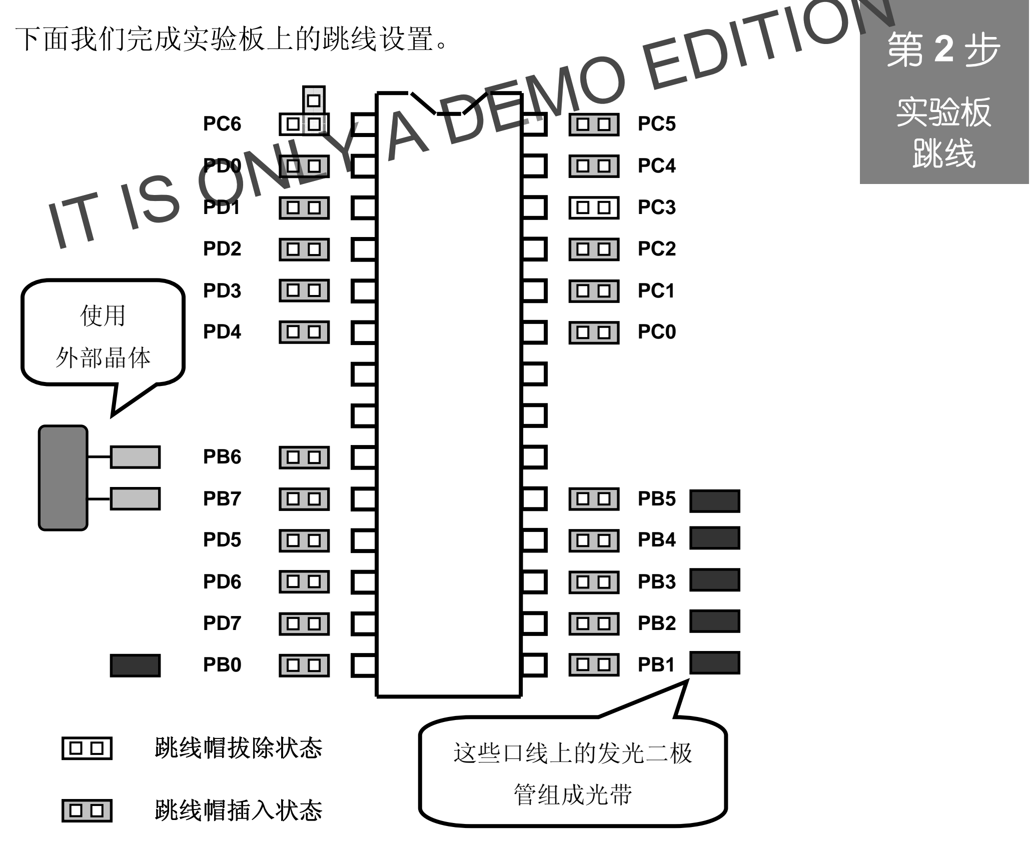
电位器两端分别接 **5V** 电源和地线，以取得分压用的电压。

例图 **12.2** 准备分压电位器

第 2 步
实验板跳线

下面我们完成实验板上的跳线设置。

IT IS ONLY A DEMO EDITION



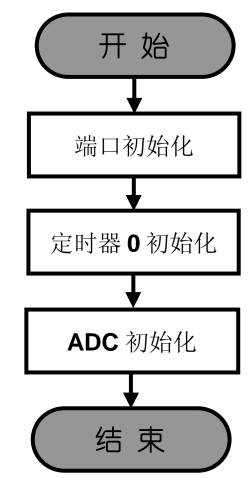
例图 **12.3** 跳线帽的设置

电位器中间的输出端连接到**PC3**跳线靠近单片机一侧的插针上。连接情况的图示请参考实验末尾的**例图12.7**。

[软件设计]

第 3 步
软件
设计

用中断方式实现的电压型光带显示电量计的软件主要分为**3**个模块：初始化模块、定时器中断服务模块、**ADC**转换完成中断服务模块。他们的流程图如**例图12.4~12.6**所示，其中初始化模块中的代码我们在本章“**4.8 使用代码生成器生成ADC初始化代码**”部分已经讨论过了，在稍后我们会对**ADC**转换完成中断服务模块的代码进行详细讲解。



例图 12.4 初始化模块流程图

请注意，在之前的几个实例中，我们都没有在 Device_INIT 函数中设置 MCUCR / DIDR0 及 PRR 寄存器，这是因为这些寄存器在复位后的初始化状态刚好可以为我们直接所用。为了突出重点，在代码中没有再对他们进行设置。

严格说来在这个程序中，除了 DIDR0 寄存器的初始值与我们所需要的状态不符，必须修改之外，其他两个寄存器的复位初始值可以直接使用。

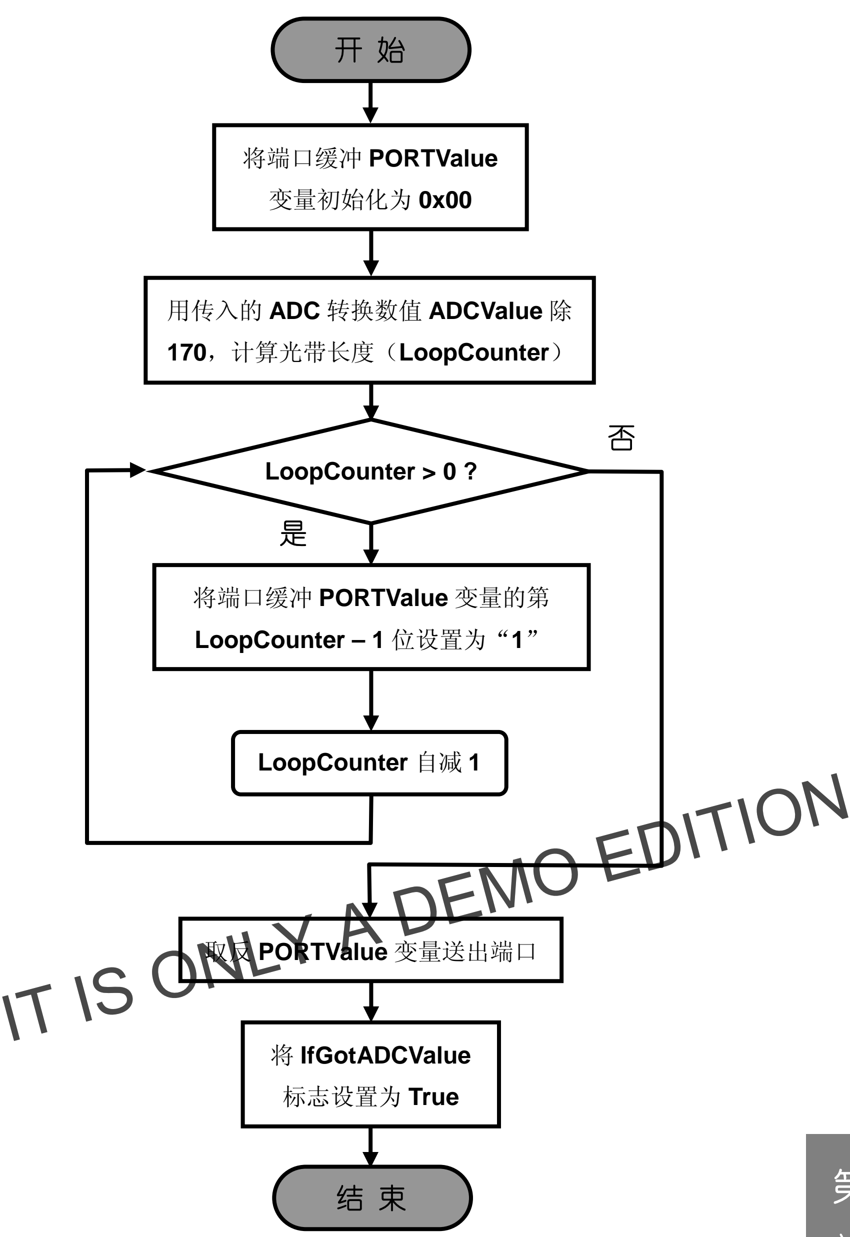
为了强调与单片机全局设置相关的寄存器，我们在这里对这 3 个寄存器进行了初始化设置。

——DA895 注

IT IS ONLY A DEMO EDITION

开 始
重新加载定时器 **0**
当前有 **A/D** 转换正在进行？
是
否
将 **A/D** 转换标志 **IfGotADCValue0** 设置为“假”
启动一次 **A/D** 转换
结 束

例图 **12.5**
定时器中断服务模块软件流程图



IT IS ONLY A DEMO EDITION

例图 12.6 ADC 转换完成中断服务模块流程图

第 4 步
突破关键
编写代码

● 代码编写：

完整的程序源代码请参考光盘中实例12 电量计文件夹下的文件，本程序的核心是“ADC转换完成中断服务模块”，他的代码如下：

```
void Insert_Adc_ISR_Code(void)
{
    unsigned int ADCValue = ADC;          //获取ADC10位精度的采样结果
    unsigned char LoopCounter = 0,PORTValue = 0;

    LoopCounter = ADCValue / 170;      //转换为端口光柱显示
```

由于拷贝的问题，这里的代码和直接用文本编辑器在计算机上看到的程序版式可能有差别。

最终的源代码请参见光盘中的文件，书中的代码仅供参考。

——老编注

```
    while(LoopCounter)
    {
        PORTValue |= 1<<(LoopCounter-1);
        LoopCounter--;
    }

    PORTB = PORTValue^0xFF;             //由于端口上的LED在低电平时点亮，
                                        //为了适应习惯在这里对端口数据做
                                        //了反相的处理
    IfGotADCValue = True;               //设置采样完成标志
}
```

该模块在读入了**A/D**转换结果后保存在变量**ADCValue**中，交送后面的程序进行计算处理。**IfGotADCValue**是一个标志变量，当启动一次**ADC**转换时将该标志设置为**False**，当完成一次转换后将其设置为**True**，用来防止在一次转换还未完成的情况下错误地启动下一次转换过程，虽然对于本程序**1mS**的中断频率来说这种错误并不会出现，但考虑到代码的规范性，我们仍然加入了这个标志。

限于篇幅，在这里我们不在介绍程序的编译和目标代码下载方式了，有疑问的读者请参考第一篇第二章。

——老编注

为了把**A/D**转换得出的数字量均匀地显示为光带，我们设计了一个除法运算，由于一共有**6**只发光管用于显示光带，每一只所代表的电压范围应为**1023 / 7 = 107.5**，故选择**107**。

PORTValue是一个临时变量，用于"组装"送往端口的光带数据，对他的操作是以位运算的方式进行的。例如当**LoopCounter**为**4**时，说明需要点亮**4**个发光二极管，此时**while**循环会被执行**4**次，在循环中依次将**PORTValue**的第**3**、**2**、**1**、**0**位设置为"**1**"。最后，由于在我们的实验板上，端口呈低电平时发光二极管点亮，为了使光带的显示符合习惯，**PORTValue**被取反后再送出到端口。

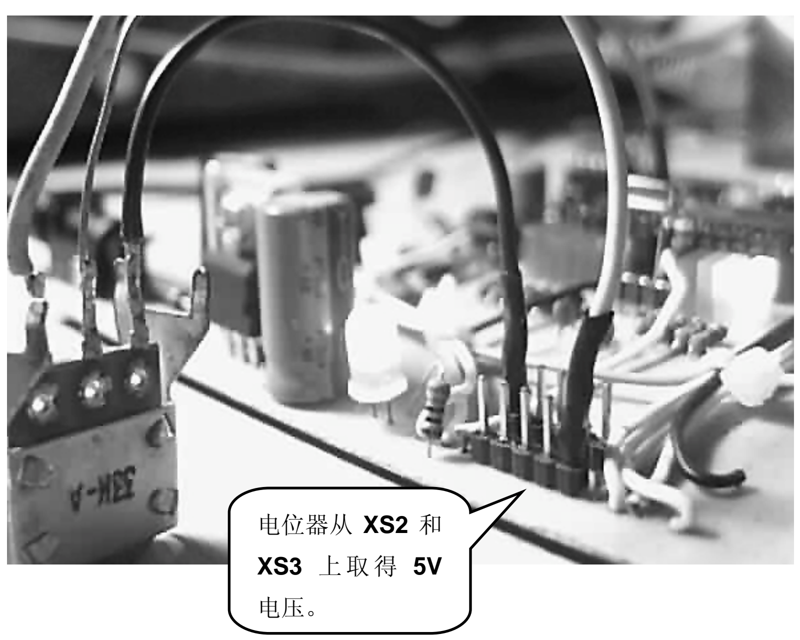
IT IS ONLY A DEMO EDITION

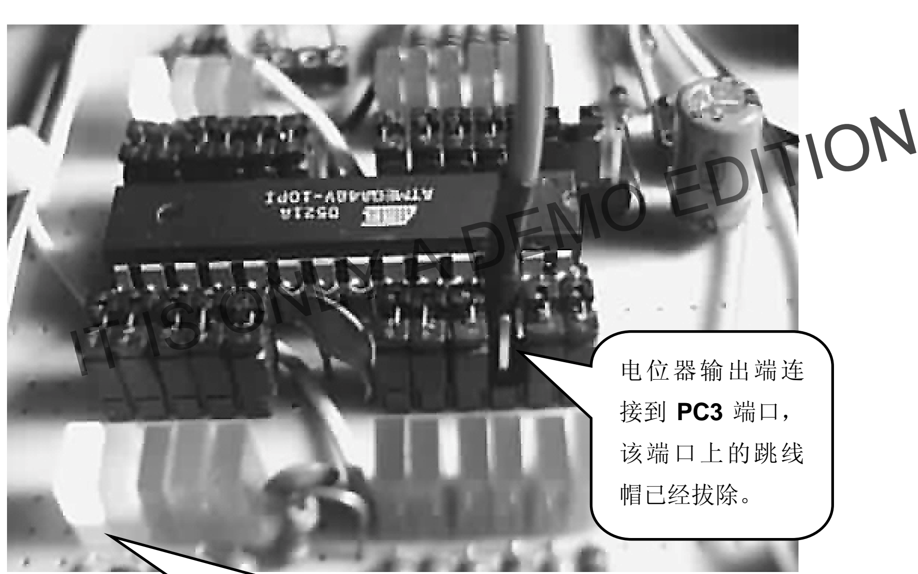
第 5 步 观察实验现象

现在来看一下演示的效果：旋转电位器的旋柄可以看见，**PB0**到**PB5**口线上发光二极管组成的光带长度随着旋柄的位置而改变长度。当输入电压最高时，**6**只**LED**全部点亮；当电压为**0**时，所有**LED**均熄灭。

例图**12.4**是实验的实际照片，图中可以看到连线方式和光带效果。

缓慢调节电位器的旋柄，使之达到"某一个发光二极管恰好点亮"的位置，会观察到该发光二极管出现闪烁现象，这是由于干扰引起的，欲知详情，请阅读第二篇第四章"进阶阅读"部分。





IT IS ONLY A DEMO EDITION

例图 12.7 电量计实验效果照片

第三篇

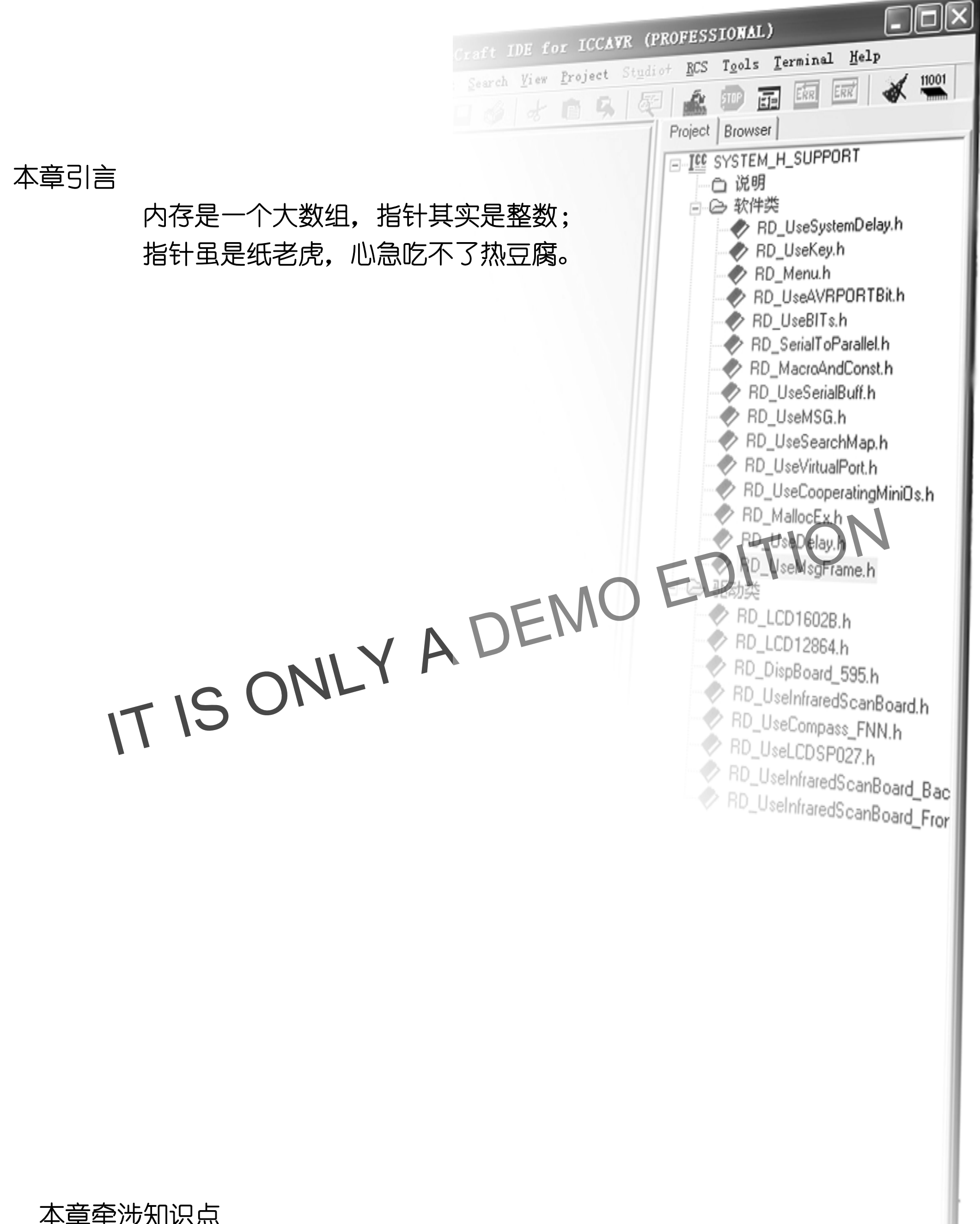
朝花夕拾　指针都是纸老虎

来自身边的启示　初识嵌入式

IT IS ONLY A DEMO EDITION

编写程序，甚至编写软件，不同于一般的应用文写作。它是一种创作，一种需要人们运用自己的聪明才智，经过一定的思维过程，通过动手实践才能完成的艺术。从某种意义上说，编写程序更像是文学创作——一个火花般闪耀的灵感，一种轻松屈驾语言的能力，再加上作者一颗对生活充满感悟的心。一杯咖啡抑或是一壶清茶，悠悠的散发着暖香……伴随着轻快而又坚定的键盘敲击声，屏幕上的光标流水般的跳动……不用多久，随着一个让人舒畅的懒腰，一段佳作跃然纸上。

第二章 指针都是纸老虎



本章引言

内存是一个大数组，指针其实是整数；
指针虽是纸老虎，心急吃不了热豆腐。

IT IS ONLY A DEMO EDITION

本章牵涉知识点

- 如何理解存储器地址空间的概念
- 指针的本质
- 地址绑定与指针欺骗
- 函数指针与消息地图
- 其他

……

原理 | 解析 <<阅读提示：原理简析部分试图将艰涩难懂的理论做浅显化的讲解，高手可以跳过

黔驴技穷的故事想必大家早有耳闻——老虎虽然强大，但是对于从未见过的“黔之驴”也心存几分畏惧。这一畏惧显然是暂时的，一旦弄清了对方“温顺食草动物”的本质，黔之驴不过是老虎盘中的一道珍奇美味而已。指针的学习，也是如此。本文无意于从编译器的角度为大家展示指针种种语法细节，我们的主旨是希望通过一个简化了的模型，帮助大家理解指针的一些行为和特征，最终跨过这道学习的门槛。

2.1 存储器是一个大数组

为了揭晓指针的本来面目，我们首先需要明确一个假设：存储器空间不过是一个简单的一维字节数组，数组的起始地址就是 **0x0000**；而数组的大小就是整个存储器所包含的字节总数。为了方便后面的讲解，我们不妨使用 **C** 语言建立下面的伪声明：

```
unsigned char MemorySpace[MEMORY_SIZE];
```

这里 **MEMORY_SIZE** 是一个宏，它以字节为单位描述了存储器的实际大小。**MemorySpace** 就是这个一维数组的名称，通常我们称之为存储器空间。由于先前有了“存储器就是一个大数组”的假设，因此，该数组的起始地址就是 **0x0000**，最后一个字节的地址为 (**MEMORY_SIZE-1**)。

IT IS ONLY A DEMO EDITION

细节 1
数组的最初内容

■ 该数组的初始内容是什么呢？

在 **C** 语言中，数组声明以后，编译器没有义务将数组中每一个元素都进行清零操作，这和我们习惯上认为的“新声明的数组就像一张白纸”的愿望是不同的。因此，我们的 **MemorySpace** 初始值同样无法确定。考虑到这样一个大数组中存放了程序所有需要使用的变量，而变量的内容直接影响程序的正常运行。存储器空间的**初值不确定性**会在程序运行时带来隐患——因此，我们编写代码时一定要养成好习惯：对于声明的变量一定要进行初始化，给与一个确定的初值：

```
unsigned int Numebrs;         //不好的习惯
unsigned int Numbers = 0;     //良好的习惯
```

2.2 “指鹿为马”说指针

如果存储器空间可以被看作是一个大的一维数组，那么指针又是什么呢？（为了方便讨论，以后的文章中，如果没有特殊说明，指针就表示指针变量）很遗憾，如果你将一个运行中的程序打断，观察

整个 **MemorySpace** 数组的内容，你会发现，没有任何一个数据在脑门子上写着“我是指针”几个大字。造成这种现象，有两种可能：

第一、 指针并不存放在 **MemorySpace** 数组中，而在别的地方专门有一个存放指针的存储器，我们姑且就称其为“指针存储器（**PointsMemory**）”；
第二、 指针隐藏在 **MemorySpace** 的数据中，只不过我们没有发现他们。

对于第一种假说，我们有很明确的证据：**Datasheet** 曾明确告诉我们 **AVR** 有一些专用的指针寄存器，例如 **PC** 指针，硬件堆栈指针等等。但是，我们在程序中大量使用了指针变量，如果这些指针真的全部存储在专用存储器里，会不会受到存储器大小的限制呢？为什么 **Datasheet** 对于这一重要的技术指标只字未提呢？

对于第二种假设，如果我们坚持认为“指针就是一种完全不同于其他变量类型的东西，至于它具体什么样子我们不知道而已”那么，我们就确实无法认同这种假设的存在——即便我们可能已经看到了事实的真相。如果我们放弃从前的偏见，大胆的假设，指针其实只是一种概念上的东西，它的存储本质与普通变量没有任何区别；换句话说，指针的概念就是我们强加在某些普通变量头上的一顶帽子，那么，眼前 **MemorySpace** 数组中的一切就可以解释了。

IT IS ONLY A DEMO EDITION

承认这种说法的存在，必须搞清楚“这种强加行为，对于普通变量的类型有什么要求”以及“究竟需要在普通变量上强加怎样的概念，才能被称之为指针”两大问题。

可喜的是，解释上面的疑问并非无章可循，相反在 **C** 语言中 **sizeof()**运算符为这种“假说”提供了强有力的证据。首先，让我们复习一下关于指针操作的两个运算符：“**&**”和“*****”：

“**&**”在这里并不是位运算符号，而是用来获取指定变量实际地址的运算符。例如：对于 **unsigned char** 型变量 **Number**，通过运算 **(&Number)**我们就可以知道变量 **Number** 在我们的 **MemorySpace** 中的实际位置——如果前面关于“存储器就是大数组”的假设成立的话，那么“**&**”运算返回的就应该是变量 **Number** 在 **MemorySpace** 中的“数组下标”。

“*”在这里也不代表乘法符号，它和“**&**”的功能相反，通过“*”我们可以由 **MemroySpace** 的数组下标直接访问到 **Number** 变量本身。例如：对于前面的例子，如果我们声明一个指针变量：

```
unsigned char *P = (&Number);
```

那么**(*P)**就完全等效于变量 **Number** 本身。

最后，我们来说一说 **sizeof()**运算符。首先，这是一个运算符，而不是包含在什么头文件中的某个库函数。其次，在 **C** 语言中，**sizeof()**

可以告诉我们某一变量甚至是变量类型“究竟占用了多大的存储器空间”。有了这样一个工具，我们首先看看各种不同类型的指针在空间占用上有什么不同：

对于指针变量

```
unsigned char *pCHAR = NULL;    //记得前面说过的，变量要初始化
unsigned int *pINT = NULL;      //指针是变量，当然不能例外
unsigned long *pLONG = NULL;    //指针甚至要求更严格，必须初始化
……
```

分别进行**sizeof()**运算，得到结果

```
计算 sizeof(pCHAR) 得到 2；
计算 sizeof(pINT) 得到 2；
计算 sizeof(pLONG) 得到 2；
……
```

显然，指针的大小是固定的，统一为两个字节，这一结果和指针的类型没有任何关系。那么，指针变量占用的存储空间为什么是**2**个字节，它是恒定不变的么？回答这个问题其实非常简单，根据本章一开始“存储器就是一个大数组”的假设，指针中所存放的内容应该是**MemorySpace**数组的下标，而下标的大小直接决定于数组的实际大小。

查阅**Datasheet**我们知道，**AVR**虽然是一种**8**位单片机，却有着直接访问**64K**存储器空间（**MemorySpace**）的能力，而**2**个字节的无符号整数恰好能表示**0~65534**（**64K**）的数值范围，从这个角度来讲，问题得到了圆满的答案：指针所占用的存储器大小直接决定于处理器能访问的存储器大小。由于**AVR**只能够访问**64K**存储器，其指针只需要**2**个字节就能保存；同样道理，对于**32**位**PC**机来说，保存一个指针就需要**4**个字节（**4*8**位=**32**位）；对于**64**位平台的**PC**机来说，保存一个指针的开销就是**8**个字节了。

IT IS ONLY A DEMO EDITION

通过上面的分析，我们做一个大胆的假设：对于**AVR**单片机来说，指针的实质就是占用两个字节的**unsigned int**型变量，其占用存储空间的大小是固定不变的。由此回答了前文提出的两大疑问之一：“这种强加行为，对于普通变量的类型有什么要求”。

如果上面的假设成立，那么普通**unsigned int**型变量和指针有什么区别呢？不同类型指针之间的区别又体现在什么地方呢？有一点是显而易见的，指针之间的区别就在于它们所指向的变量类型是不同的；而对于“普通**unsigned int**型变量和指针的区别”，由于两个字节的长度除了地址以外并不能容纳下额外的类型描述信息，我们也许只能使用“人们强加的概念”来解释了：

如果说，对于普通的**unsigned int**型变量，人们通过编译器，强行将指针所指向变量的类型信息与其打包在一起；那么在这一类型信息中，至少包含了指针所指向变量的长度信息。下面的代码实验可以证明这一推论：

对于

```
unsigned char Number = 0;
unsigned char *p = (&Number);
```

分别进行**sizeof()**运算，得到结果

```
计算 sizeof(Number) 得到 1；
计算 sizeof(*p) 得到 1；
计算 sizeof(p) 得到 2；
```

第二个运算中，**sizeof**括号里包含了一个进行“*”运算的指针。根据前面的说法，**(*p)**就完全等效于变量**Number**，**sizeof()**实际上是直接获得变量**Number**类型信息中的存储空间信息，因而，两个运算都获得了同样的计算结果。同理，第三个运算中，**sizeof()**直接返回指针的存储空间信息，而指针的存储本质就是一个**unsigned int**型变量，因此，我们得到了一个数值为“**2**”结果。

我们注意到，这个实验证明了指针中至少包含了其所要指向变量的长度信息，而这一信息明显是没有存放在**MemroySpace**存储器空间里面的。我们将这一现象解释为“人们强加在**unsigned int**型数据上的类型信息”。那么，这种强加行为究竟有没有存在的依据与合理性？有没有实现方法呢？找到这一问题的答案，就等于给这个假说找到了一个合理的解释。

IT IS ONLY A DEMO EDITION

2.3 空指针（void *）

由前面的假设，我们推断，如果指针的存储本质就是一个**unsigned int**型的变量，那么一个**unsigned int**型的常量应该也是指针常量的存储本质了。

例如，对于**unsigned int**型常数**0x0048**，如果我们认定它是一个地址，或者说，认定它就是一个**MemorySpace**数组的下标，那么，应该存在某种方法将其转化为一个指针常量。如果说，强制转换是我们第一个想到的方法；那么，将**0x0048**转换为什么类型的指针，就是我们需要考虑的第一个问题。

一个指针，一定需要指定类型信息么？当一个指针抛弃了类型信息，而保留存放有地址**unsigned int**型变量，它还是一个指针么？答案是肯定的。在**ANSI-C**的新规范中，增加了一种特殊指针——空指针，使用**(void *)**来表示。

空指针就是这样一种指针，由于其中只含有地址信息，没有指定任何该地址所包含内容的数据类型信息，因此，可以被视为“万能指针”或者“胚指针”——任何指针在丢掉了类型信息以后，都可以转化为空指针；同时，空指针在附加了类型信息以后，都可以成为某一数据类型的专用指针。例如：

```
unsigned char NumberA = 0;
unsigned int NumberB = 0;
void *SuperPoint = NULL;            //声明了一个空指针
```

```
unsigned char *pCHAR = (&NumberA);
unsigned int *pINT = (&NumberB);
……
SuperPoint = (void *)pCHAR;          //丢弃指针 pCHAR 的类型信息
SuperPoint = (void *)pINT;           //丢弃指针 pINT 的类型信息
```

在这个例子中，指针**pCHAR**和指针**pINT**在通过**(void *)**进行强制类型转换以后，丢弃了指针的类型信息，因而转化成了空指针，可以直接赋值给空指针变量**SuperPoint**。

细节 2 指针的整数运算

- 指针的整数运算：

指针的存储本质是整型变量的另外一个重要证据就是：指针也可以进行整型变量的数值运算。这同时也是指针与普通整型变量的一个重要区别：指针的数值运算，结果受到类型信息的影响。例如：

对于**MemorySpace**存储空间中某一个地址，我们设其下标为**Addr**（以字节为单位的下标）。设下面的指针初值都为**Addr**。则，对应的运算结果为：

```
对于 unsigned char 型指针 PointA，PointA+1 结果为 Addr+1；
对于 unsigned int 型指针 PointB，PointB+1 结果为 Addr+2；
对于 unsigned long 型指针 PointC，PointC+1 结果为 Addr+4；
……
```

IT IS ONLY A DEMO EDITION

通过观察，我们很容易得出这样的结论：

```
对于某一类型的指针 p，p+1 的结果为 Addr+sizeof(“对应类型”)。
```

如果**p**是一个**(void *)**型的指针，**p+1**的结果又会是多少呢？

指针能够进行数值运算，是其整数存储本质的体现；指针的数值运算要受到其内部类型信息的限定，这是其指针特征的体现——假设指针每次数值加**1**的结果仅仅只是让其记录的**MemorySpace**下标增加**1**个单位的话，那么通过指针加**1**的方式来遍历数组将会导致完全错误的结果——**C**语言经典教材上说**unsigned int**型指针**p**指向了一个**unsigned int**型的一维数组，我们可以通过**p+n**的方式来访问数组的第**n**个元素。对于这种说法，大家也许都觉得是理所当然；反过来，我们却对**p+1**等于**Addr+2**感到吃惊，实在是一件很滑稽的事情。

空指针(void *)没有类型信息，因此运算符sizeof()没有办法获得一个确定的结果，所以空指针(void *)不允许进行整数运算。这就是隐藏在法则背后的真相。

2.4 变量、指针变量、指针常量与“不应该被修改的指针变量”

由上一节的讨论我们知道，在将普通整型常量强制转化为指针常量时，如果没有明确的目标，可以忽略指针的类型，直接将其转化为空指针常量：**((void *)0x004B)**。通过这种方式，我们就获得了一个指向**MemorySpace**下标为**0x004B**的空指针常量。由于是指针常量，因此，该指针并不占用任何**MemorySpace**存储空间，也就是说，对其进行“**&**”运算，试图获得一个“莫须有”的下标是没有任何意义的：

```
//对一个指针常量进行“&”是没有任何意义的运算
void **p = &((void *)0x004B);
```

指针常量也可以是任何其它类型，例如：**((unsigned int *)0x004B)**就表示指针所指向的地址中存放的是一个**unsigned int**型数据，而指针常量**((unsigned long *)0x004B)**所指向的空间中保存的就是一个**unsigned long**型数据了。

对于一个指针常量来说，在整个程序的运行期间，其中包含的**MemorySpace**下标信息都不会改变，因此，<u>指针常量在任何场合下都不需要占用MemorySpace空间，而可以由编译器在编译过程中，像填写表格一样“填写”到程序中使用到该指针常量的地方</u>。因为内容固定，而可以由编译器在编译时进行填表处理，这是一个很重要的特性。虽然容易理解，但是如果没有注意到这一点，就会产生类似对指针常量求地址“**&**”这样的错误；也很难理解多维数组的一些概念。

IT IS ONLY A DEMO EDITION

相对“指针常量”，还有一种被**const**关键字修饰的“常量指针”。这又是一种绕口的概念，为了方便记忆、体现其本质，我们建议大家将后者称之为“不应该被修改的指针变量”。说它不应该被修改，是因为它被**const**关键字修饰；说它是变量，是因为其实实在在的占有了**MemorySpace**存储空间。因此，“指针常量”和“不应该被修改的指针变量”是两个完全不同的概念。需要明确的是，“指针常量”是绝对不能被修改的；而“不应该被修改的指针变量”，虽然“不应该”但是绝对可以“被修改”，这是其变量本质所决定的。在随后的章节中，我们将详细介绍，如何绕过**const**限定，“偷天换日”修改这一本不应该被修改的变量。

细节 3
指针常量
变量
指针变量
常量指针

- 指针常量、全局变量、指针变量、不应该被修改的指针变量

  - 指针常量 = 普通整数强制转化为某一个类型的指针
    例如：**((unsigned int *)0x004B)**

  - 全局变量 = (*(指针常量))
    如果试图采用这种方式强行将某一**MemorySpace**空间征用做变量，你需要首先确认，这一空间是系统没有占用的。例如：**(*((unsigned int *)0x004B))**就是一个**unsigned int**型的变量，如果

我们非要给这个变量加一个名字的话，你就会发现它和普通的变量没有任何区别了。

```
//人工强制设定的 unsigned int 型变量 NumberA
# define NumberA      (*((unsigned int *)0x004B))
```

如果你仔细观察过**iomXXXv.h**文件的话，会发现其实**ICC**就是通过这种方法声明寄存器变量的：

```
//定义寄存器 GPIOR2，这其实就是一个变量哦
#define GPIOR2      (*(volatile unsigned char *)0x4B)
```

- 指针变量 **= unsigned int**型变量 **+** 可选的类型信息
  对于空指针**(void *)**来说，类型信息就是可选的。<u>指针变量中保存的就是**MemorySpace**数组的下标</u>。例如：

```
unsigned long NumberA = 0;
unsigned int NumberB = 0;

//借用普通整型变量存储空间的指针
# define pNumberA      ((unsigned long *)NumberB)
……
//和普通指针一样的使用方法
pNumberA = &NumberA;
```

IT IS ONLY A DEMO EDITION

- 不应该被修改的指针变量 **= const +** 指针变量
  例如：

```
const unsigned long *p = NULL;
```

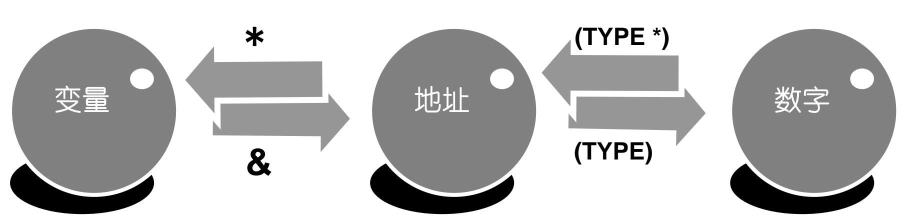


图 **3.2.1** 类型迁移示意图

2.5 数组、指针数组、数组指针

数组是一种特殊的数据类型，它的本质就是一块大小固定的连续存储器区域。这就好比 **MemorySpace** 这个数组，它实际上只代表一个固定长度的存储区域，至于这个存储区域内部如何组织，实际上就

要杀要剐随我们便了。

为了方便大家理解，我们不妨设定，一维数组由一段存储数据的内存区域和一个指向该内存区域起始地址的指针变量构成——虽然这和实际的情形有些许不同：

```
一维数组 = 一段连续的 MemorySpace 空间 + 一个指针变量
```

在这个假设中，连续的存储空间用于存储指定数量（相同数据类型）的数据；而指针指向这段空间的起始地址。如果用于存储实际数据的连续空间大小为 **n**（单位：字节），则整个一维数组应该占用 **n+2** 的空间。一个例子：

这里关于一维数组的组成说明只是一个假设、一个方便大家理解的模型。真实的一维数组结构和这一模型虽然存在类似的地方，却并不相同。我们会在后面的内容中利用这一模型，为大家揭示数组的真实本质。

——老编 注

对于数组：**unsigned int Array[10]**，Array 表示指向数组起始地址的针变量；Array[]表示数组的本体。数组用于存储数据的连续存储空间大小为 **2*10** 个字节。考虑到指针变量 **Array** 的存在，整个一维数组应该占用 **20+2** 个字节的存储空间。我们可以通过这个指向数组首地址的指针变量与其他指针变量进行赋值操作，例如：

```
unsigned int *p = Array; //这是我们常见的情形
```

也可以通过该指针或者 **p** 指针来访问数组元素：

```
//几个等效的一维数组操作
Array[1] = 100;
p[1] = 100;
(*(p+1)) = 100;
```

IT IS ONLY A DEMO EDITION

在这里，**Array** 和 **p** 是两个等效的指针，它们的操作没有任何区别。至于第三个表达式，我们注意到，**p** 是一个 **unsigned int** 型的指针，因此，**p+1** 实际上等于在 **p** 所记录的 **MemorySpace** 下标上增加 **2** 个单位（**unsigned int** 型数据占用 **2** 个字节）。因为 **p** 是一个指针，所以 **p+1** 仍然是一个指针。对指针进行“*”获取的是对应地址内存放的数据，因此，**(*(p+1))**实际上访问的是从 **p** 指向的地址开始向后数第二个 **unsigned int** 型数据，也就是数组的 **Array[1]**。这些都不难理解。

数组的连续存储空间中，可以存储任何数据，当然包括指针，所以“指针的数组”简称“指针数组”，理所当然就是一个存放了 **n** 多指针的数组。例如：

```
//一个指针数组的例子
unsigned long *p[10];           //由十个指针组成的数组
```

前面，我们提到过，数组的本质就是一段连续的存储区域，因此，存放了 **10** 个指针的数组和存放了 **10** 个 **unsigned int** 型变量的数组在存储本质上并没有任何区别。这个概念也不难理解。

“数组的指针”，简称“数组指针”，就是一个指向一维数组中那个指针变量（例如前面的指针 **Array**）的指针——一个指向指针的指针。只不过，这个指针有一个特殊之处，其包含的类型信息中，关于被指向对象所占用的存储空间大小并不是被指指针（例如前面的 **Array**）所占用的空间 **2**，而是一维数组实际用来保存数据的存储器空间（例如前面数组实际存放数据需要 **20** 个字节）。关于指向指针的指针，我们会在随后详细说明，这里我们只是想说明“数组指针”的本质。

数组指针在声明的时候，必须要指定其指向的数组最后一个维度占用存储器空间的大小。例如：

```
//一个不合法的数组指针，它没有指定所指向数组最后一个维度的大小
unsigned int (*p)[] = &Array;

//一个合法的数组指针
unsigned int (*p)[10] = &Array;
```

大家注意到，对于数组**Array[10]**，用数组指针**p**指向它时，使用了一个“**&**”符号。数组名称不就是数组的起始地址么？难道指向数组的指针需要的不是这个地址么？是的，数组指针本质是一个指向指针的指针（而不是普通指针），因此，它感兴趣的不是数组的起始地址，而是指向这个起始地址包含在数组中的这个指针**Array**。前面提到过**Array**是一个指针变量，但是它在**p**的眼中，就是比自己低一个等级的普通变量，因此，要获得这个“低级的普通变量的地址”当然需要一个“**&**”运算符号啦。正因为如此，指向指针的指针在表示其指向的数组时，需要使用一个“*****”运算符，例如：

IT IS ONLY A DEMO EDITION

```
(*p)[1] = 100;
```

因为，**p = &Array**，**(*p) = (*(&Array)) = Array**，所以，上面的代码等效于：

```
Array[1] = 100;
(*(Array+1)) = 100;
```

为什么数组的指针在声明时一定说明数组最后一个维度的大小呢？答案很简单。数组指针虽然是一种指向指针的指针，但是，既然我们称其为数组指针而不是普通的指向指针的指针，就是因为，他们之间有一个根本性的区别。普通指向指针的指针，在类型信息描述中，包含的是其指向的指针变量占用的空间（也就是说，恒为**2**）；而数组指针在类型信息里面包含了数组的大小信息（在多维数组指针中，可以忽略前面各个维度的大小，但是不能忽略最后一个维度的大小，在随后的章节中，我们将详细讲解其中的原因）。通俗的说：作为一个数组指针，要想与普通的“指向指针的指针”划清界限、标新立异，就必须要包含数组的大小信息，否则就“没有脸面被称之为数组指针”。

2.6 多维数组

在很多 **C** 语言经典中，对于多维数组，都有类似的描述："多维数组可以被看作是多个一维数组的叠加。第一个一维数组中所有元素都是指针，这些指针分别指向一个一维数组，这些一维数组中的所有元素也是指针，他们也分别指向一个一维数组……直到最后一个维度的一维数组，其中的数组元素不再是指针，而是实实在在的数据。……这种模型用来帮助我们理解多维数组的结构，但实际情形还是有一些差别的。"

撇开上面文字中的"差别"不管，我们采用前一小节中提到的一维数组模型来构建一个二维数组（三维数组的情形类似），仔细研究一下采用这种方法构建出来二维数组具有哪些特性：

首先，我们声明一个一维数组 **ArrayA[2]**作为第一维度的一维数组。按照文字描述，这应该是一个"指针的数组"：

```
unsigned int *ArrayA[2];
```

在这个数组中，包含了 **2** 个指向 **unsigned int** 型数据的指针，**Array[0]**和 **Array[1]**，占用 **2*2** 个字节的空间。**ArrayA** 是一个指向该数组起始地址的指针变量。

接下来，我们声明 **2** 个数组，它们都包含了 **3** 个 **unsigned int** 型数据：

```
unsigned int ArrayB[3];
unsigned int ArrayC[3];
```

IT IS ONLY A DEMO EDITION

在这两个数组中，分别包含了 **3** 个 **unsigned int** 型数据；**ArrayB** 和 **ArrayC** 是两个指针变量，分别指向两个数组的起始地址。接下来，我们将这三个数组组合起来，构成一个二维数组，并通过宏，将这个 **2*3** 的三维数组命名为 **Array**：

```
# define Array        ArrayA

//组合一个二维数组
ArrayA[0] = ArrayB;
ArrayA[1] = ArrayC;
```

二维数组的构建工作完成。我们可以通过宏 **Array** 对其中的任何一个元素进行访问。接下来，让我们一起来看看这样一个二维数组有哪些性质。

- **Array**：数组名，实际上是指向一维数组**ArrayA[]**起始地址的指针变量**ArrayA**。该指针所指向的空间，实际上就是**ArrayA[0]**。**ArrayA[]**是一个"指针的数组"，所以，**ArrayA[0]**表示一个指针，

它指向数组ArrayB[]的起始地址。也就是说，Array实际上是一个指针变量（ArrayA），它指向的存储空间中，保存了一个指针ArrayA[0]，该指针指向另外一个数组的起始地址——因此，相对最后的数组ArrayB来说，Array是一个指向指针的指针。

- Array[0]：指针。其中实际保存了数组ArrayB[]的起始地址。
- (*(Array+0))：Array实际上就是指针变量ArrayA，前面说过，它是一个指向指针的指针。已知指针所占用的存储空间恒为2，如果Array所指向的存储单元在MemorySpace数组中的下标为n，则表达式Array+1实际上相当于n+2（前面说过，指针变量保存的值就是MemorySpace数组的下标，而下标是以字节为单位的），Array+0就相当于n——数值保持不变。又因为，指针变量Array指向的存储单元为Array[0]，因此(*Array)就表示Array[0]，这是一个指针。

- *Array：解释同上。

- Array+1：这是一个指针。它指向指针变量Array[1]所在的存储单元，因此，是一个指向指针的指针。

- &Array[1]：Array[1]是一个指针，对其进行求地址运算，将获得该指针变量的地址，它在数值上等同于Array+1。

- Array[1]+2：Array[1]是一个指针，指向数组ArrayC[]的起始地址。由于ArrayC[]是一个unsigned int型的数组，因此，该指针中所包含的类型信息就是unsigned int。指针每增加1实际上就等于地址增加2个字节。Array[1]是指针，Array[1]+2仍然是指针。该指针记录的是ArrayC[]中第三个元素ArrayC[2]的地址。也就是“(Array[1]+2) == (&ArrayC[2])”。对于：

```
unsigned int *p = Array[1];
```

因为p等于指针变量ArrayC，所以p[2]等效于ArrayC[2]。注意上面的表达式中p实际上等于Array[1]。因此p[2]等效于Array[1][2]。这就是我们所熟悉的二维数组的表达形式。

- (*(Array+1)) +2：前面，我们已经证明过Array+1等效于(&Array[1])，所以(*(&Array[1]))实际上等效于Array[1]。又因为Array[1]+2等效于&ArrayC[2]进而等效于&Array[1][2]，所以(*(Array+1)) +2表示的仍然是一个指向ArrayC[2]的指针。

- &Array[1][2]：解释同上。

- *(Array[1]+2)：由Array[1]+2等同于&Array[1][2]可知*(Array[1]+2)等同于Array[1][2]。

IT IS ONLY A DEMO EDITION

- ***((*(Array+1))+2)**：由于**Array+1**等同于**&Array[1]**，因此***(Array+1)**等同于**Array[1]**，原式等效于***(Array[1]+2)**。

上面，我们通过一个“指针的数组”和一个 **unsigned int** 型的普通数组组建了一个 **unsigned int** 型的二维数组 **Array**，并对这一数组的性质进行了分析。采用这种方式构建的二维数组在性质上和实际的二维数组是非常类似的，但是，这种模型化的“人造二维数组”要比真正的二维数组容易理解的许多。那么，这种人造二维数组在性质上和真实的二维数组有什么区别呢？这一区别又是如何造成的呢？

不知大家是否记得，我们的模型中，一维数组由两个部分组成：一个是与该数组同名、指向数组起始地址的指针变量（例如前面的 **ArrayA**、**ArrayB**、**ArrayC**）；一个是数组实际用来存放数据的连续存储区域。我们构建的这个一维数组模型，占用的空间要比现实中同样条件的一维数组多 **2** 个字节——也就是指针变量所占用的空间大小。这个指针就是解释整个问题的关键。

首先，对于编译器来说，数组在声明时都规定了大小，其起始地址也是常量，并不会在程序的运行时刻发生变化。因此，这样一个指向数组起始地址的指针变量，虽然非常重要（只有它指示数组的起始地址，当然不可缺少），但这一不会变化的量却要占用有限的 **MemorySpace** 存储空间——实在不划算。一般认为，编译器就把这个指针变量按照指针常量的方法处理了：不占用实际的存储空间，只在编译的时候，使用填表的方法将指针所包含的地址信息填写到相关的位置。

事实上，这种“等效”的观点是不准确的。这个指向数组首地址的指针变量虽然在程序的运行时刻其值不会发生改变，可以被认为是一个常量。但是前面提到过：指针常量是不能进行“**&**”运算的——这是指针变量的专利。因此，更为准确地理解方式是：对于这个具有常量特征的指针变量，编译器在编译的某一时刻仍然假设它实际存在，一切操作都按照指针变量的语法进行，只不过假设该指针保存在 **MemorySpace** 以外的其他存储器中；在随后的某一时刻，编译器专门针对这个“常年不会变化的变量”进行了特别的处理：

首先、对这个 **MemorySpace** 中“不存在”的指针变量进行 **sizeof()** 运算得到的将不在是 **2**，而是整个数组实际占用的空间大小。在所有求取数组大小的运算 **sizeof()**中，都忽略其存在；或者说，只以数组中位于 **MemorySpace** 存储空间中的部分为运算对象。例如：对数组 **unsigned int Array[10]**进行“求大小”运算 **sizeof(Array)**，获得的并不是 **2** 而是 **20**，当然更不是我们前面模型中提到的 **22**。

其次、所有针对该指针的“**&**”运算，虽然被保留，但是运算结果一律以该指针所指向的地址替代。例如：原本意义和内容都不相同的**&Array** 和 **Array**，经过处理以后，将具有相同的数值；而两个表达式所代表意义的不同却得到了保留。

再次、与多维数组相关，所有包含了该指针“**&**”运算结果的表达式，其运算结果中**&Array** 部分都被 **Array** 所指向的地址所替代。例

IT IS ONLY A DEMO EDITION

如：对于前面模型中的二维数组，**&(ArrayA[1])**将被 **ArrayC[]**的首地址替换，因为 **ArrayA[1]**是一个指向 **ArrayC[]**首地址的指针，因此对该指针进行“**&**”运算，将直接得到 **ArrayC[]**的首地址。

最后、多维数组中，除去最后一个维度，前面各个一维数组中保存的都是下一维度数组的起始地址，因而，受到同样的处理——抹杀这些数组在 **MemorySpace** 空间中的存在，仅仅保留这些指针语义上功能，并对这些语义作了些许修改。这些修改，在前面三条中都已经加以描述过了。

经过以上 **4** 个步骤地处理，一个由一维数组复合而成的多维数组，就只剩下实际的数据存放部分，以及一些令人迷惑的指针运算法则。指针变量的存储本质就是 **unsigned int** 型变量，一旦否认了这一存储本质，而仅仅强调指针的功能，就好比空中楼阁一般，让人茫茫然找不到实体，更谈不上对其“深刻的理解”了——多维数组，就是一个很好的例子。我们希望通过还原多维数组的本来结构，讲述编译器在背后的所作所为，让大家对于多维数组，有一种豁然开朗的感觉。也许你会说：“原来是这样啊！编译器隐藏了一些指针，难怪我先前想不通呢。其实多维数组还是蛮简单的嘛！”

多维数组，其实真的不难。如果你认为自己理解了上面的模型，那我们就来看一个遗留的老问题：为什么指向多维数组的指针必须描述最后一个维度的大小呢？其实答案很简单，因为，经过处理的多维数组，实际上只有最后一个维度的数组占用 **MemorySpace** 存储空间，不描述它描述谁？比方说对于一个二维数组 **unsigned int Array[2][3]**，一个指向该二维数组的指针 **p** 在声明的时候，就必须指明最后一个维度的大小：

IT IS ONLY A DEMO EDITION

```
unsigned int (*p)[][3] = &Array;
```

如我我们只声明第一维度的大小而忽略最后一个维度的大小呢？

```
unsigned int (*p)[2][] = &Array;
```

编译器会像我们提出抗议，因为，经过它自作聪明的优化过后，整个二维数组中构成第一个维度的一维数组已经名存实亡了——它并不占用空间。而真正留在 **MemorySpace** 存储空间中的是第二个维度的一维数组。因此，对于一个指向数组的指针来说，要想证明自己是一个真正的“数组指针”而不是普通的“指向指针的指针”，就必须拥有数组大小“这一关键文凭”。而且，这一“文凭”在数组指针参与 **sizeof()**运算时还要“出具”。如果 **sizeof()**发现数组指针出具的文凭等级不够，甚至会发出警告。例如，对于前面提到的指针数组：

```
//不完整的声明方式
unsigned int (*p)[][3] = &Array;
printf(“%d”,sizeof(*p));  //数组大小信息不完全，编译器会报告错误


//完整的声明方式
```

```
unsigned int (*p)[2][3] = &Array;
printf("%d",sizeof(*p));   //信息完整，数组大小能够通过计算获得
```

留一个思考题给大家：数组的指针 **p**，经过上面的声明以后，其包含的地址是多少呢？为什么？记得结合模型来考虑哦。

2.7 指向指针的指针

指向指针的指针，其实是一个相对的概念，理论上，你永远无法判断一个指针是否是一个指向指针的指针，甚至无法确定它不是一个指向指针的指针的指针……

指针，一定是指向了某一个内存单元。如果这个内存单元用来存放某个指针，那么对于前者，我们可以说，这是一个"指向指针的指针"。问题在于，对后者来说，我们无法界定其指向的空间中是否保存有其他指针，因此无法界定后者是否是一个"指向指针的指针"，因而也不能确定前面的"指向指针的指针"是否有资格被称为"指向指针的指针的指针"……如此往复，永世不休……

换一个角度，如果我们无视指针指向存储单元中保存的内容——管他是指针还是数据——那么世界就清静了，成为一个只有指针、没有"指向指针的指针"、理想而简单的 **C** 语言世界。

IT IS ONLY A DEMO EDITION

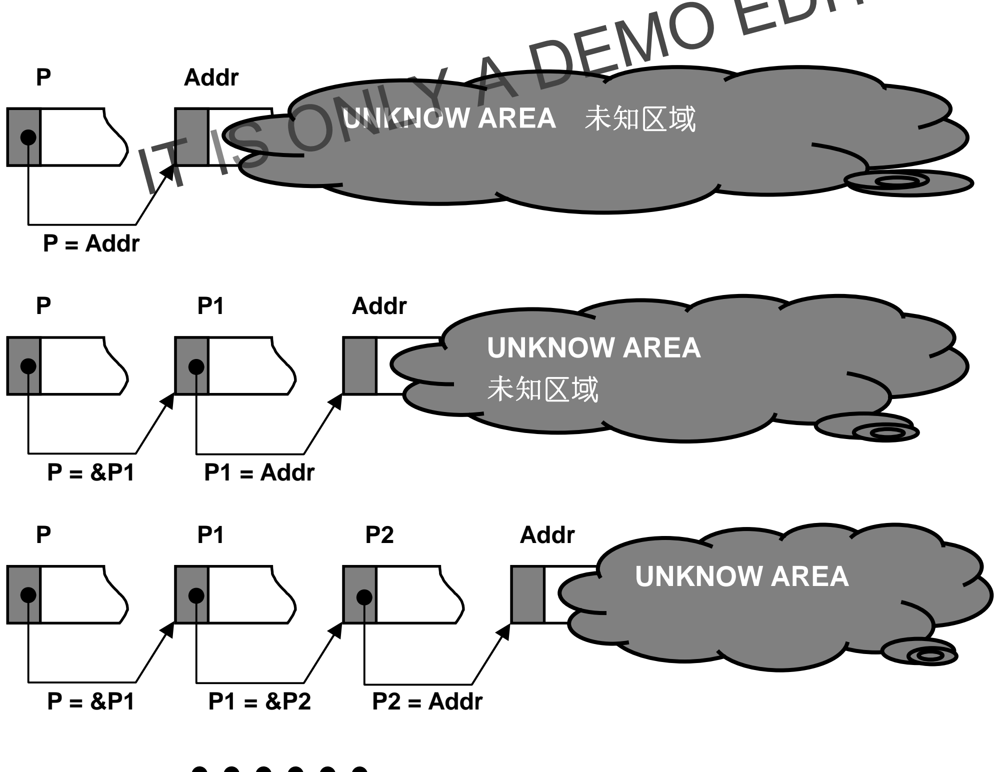


图 3.2.2 指针、指向指针的指针、指向指针的指针的指针

综上所述，指向指针的指针是一个相对概念，具有一定的不确定性。在悲观者的眼中，这一问题非常严重；在乐天派的眼中"只要

闭上眼睛，世界上就没有什么悬崖”。究竟怎样看待这个问题，视各人喜好而定啦。

实际 应用 <<阅读提示：实际应用部分着重介绍该资源在实际应用过程中常见的方法，入门必读

2.8 地址绑定与“偷天换日”

指针的存储本质是 **unsigned int** 型变量。因此，我们可以将一个指针还原为普通变量，甚至可以将一个普通整型变量强行转换为一个指针：

```
unsigned int Number = 0;            //普通整型变量
unsigned int *p = NULL;             //普通指针

p = (unsigned int *)Number;         //青蛙变王子
Number = (unsigned int)p;           //王子变青蛙
```

IT IS ONLY A DEMO EDITION

对于第一个“青蛙变王子”的表达式，我们称之为“地址绑定”。简而言之，就是将变量 **Number** 所代表的整数当作一个地址来看待。至于第二个“王子变青蛙”的表达式，嵌入式系统中除了有时候 **Debug** 需要，一般很少用到。显然，某一个指针所指向的实际地址数值对客户来说没有任何意义。

地址绑定相当灵活，你可以用任何形式来给定一个整数，然后自由地将该整数转化为任何类型的地址，甚至赋给一个指针。这里需要明确一个概念：<u>地址和指针是两回事。指针中保存地址。</u>例如，以下各种表达式形式都是地址绑定的典型实例：

```
//对寄存器地址进行绑定
#define GPIOR2      (*(volatile unsigned char *)0x4B)
```

这里，我们将常数 **0x4B** 强制转换为一个指针变量，并通过“*”运算访问地址 **0x4B**。回忆前面的内容我们知道，这种方法实际定义了一个名为 **GPIOR2** 的 **volatile unsigned char** 型的变量。

```
//对指定的内存区域进行绑定
float Number = 3.1415926;
unsigned char *p = (unsigned char *)(&Number);
```

通过指针 **p** 我们获得了存储 **float** 型变量的 **4** 个独立字节。这种方法通常用于拆解数据类型、方便串行数据通信。对于 **float** 型变量 **Number** 来说，表达式首先通过“**&**”运算获得了该变量的存储地址；接下来，通过强制类型转换，修改该地址的类型信息为 **unsigned char**，

并将修改后的地址赋给 **unsigned char** 型指针变量 **p**。这是一种更为常用的地址绑定方式。

```
//将普通变量绑定为位段的例子
typedef struct BYTE_DIV8           //一个字节拆分为 8 个二进制位
{
    unsigned BIT0:1;
    unsigned BIT1:1;
    unsigned BIT2:1;
    ……
    unsigned BIT7:1;
}BYTE_BIT;
……
unsigned char Status = 0;
……
(*((BYTE_BIT *)(&Status))).BIT3 = 1;      //通过位段将 BIT3 单独置位
(*((BYTE_BIT *)(&Status))).BIT6 = 0;      //通过位段将 BIT6 单独清零
```

在这个例子中，我们通过将一个变量的地址强行绑定在一个位段的指针上，并通过该指针来访问位段，实现对该变量任意二进制位的单独操作。对于 **unsigned char** 型变量 **Status**，表达式首先通过“**&**”运算获得其地址；接下来，将这一地址的类型强行绑定为指向位域 **BYTE_BIT** 的指针；最后，利用“*”运算，我们就可以使用这一指针来访问 **Status** 变量的任何一个二进制位了。这种技术在嵌入式系统中非常常见、有用，大家应该掌握并消化这一技术。

IT IS ONLY A DEMO EDITION

```
//对数组进行绑定的例子
unsigned char MyZone[10];
void *p = (void *)MyZone;
```

数组的本质，不过是一个指定大小的连续存储区域。一旦通过声明数组的方法获得了这一区域的指针，是蒸是煮就随我们的喜好而定了。想不想 **DIY** 一下动态内存分配？先用数组申请一段固定空间再说吧。

```
//C 语言指针最可怕的地方，幸好 ICC 不支持此语法
const unsigned int n = 1234;
unsigned int *p = (unsigned int)&n;
(*p) = 4321;             //利用指针绕过 const 限定，修改 n 的值
```

看到这里，大家应该知道为什么 **const** 修饰的变量只能被称之为“不应该被修改的变量”而不是“常量”了吧。只要你是变量，就有被修改的可能，原本的“常量”竟然可以被“偷天换日”，可怕吧？因此，这一特性甚至被作为 **C** 语言的一大缺陷而遭到批判。实际应用中，这一特性在有限的范围内，是非常有用的——前提是，你知道你正在做什么。

2.9 大端对齐与小端对齐

地址绑定技术，在概念上，等效于联合体 **union**，例如：

```
//联合体实现地址绑定
union Example
{
    unsigned long dNumber;
    unsigned char Array[4];
}Data;
```

大端（大尾，Big Endian）与小端（小尾，Little Endian）出自格列佛游记，Blefuscu 帝国的国民被根据吃鸡蛋的方式划分为两个部分：一部分在吃鸡蛋的时候从鸡蛋的大端（big end）开始，而另一部分则从鸡蛋的小端（little end）开始。

——傻孩子 注

在这个联合体 **Example** 中，**unsigned long** 型变量 **dNumebr** 和 **unsigned char** 型数组 **Array[]**拥有同样的起始地址。它等效于下面的地址绑定代码：

```
//与联合体等效的地址绑定
unsigned long dNumber = 0;
unsigned char *Array = (unsigned char *)(&dNumber);
```

于是，通过这两种方法，我们都能够利用数组（指针）**Array** 实现对 **unsigned long** 型数据逐个字节的访问。考虑在上面的情形中，下面的代码会有怎样的结果：

IT IS ONLY A DEMO EDITION

细节 4 字节的对齐顺序

```
//一个不能想当然的问题
dNumber = 0x01020304;           //占用 4 个字节的 unsigned long
printf("%d ",Array[0]);         //输出 dNumber 的第一个字节
printf("%d ",Array[1]);         //输出 dNumber 的第二个字节
printf("%d ",Array[2]);         //输出 dNumber 的第三个字节
printf("%d ",Array[3]);         //输出 dNumber 的第四个字节
```

我们将得到怎样的结果呢？

> “肯定是 **01 02 03 04** 啦！”傻孩子最性急。
> “也可能是 **04 02 03 01** 啊”**DA895** 想了想回答道。
> 出题的老耿笑着不说话。老编站在一旁皱眉头。

正像这类问题惯有的答案一样：两种答案都对，也都不对。这取决于所使用的计算机系统（单片机系统）采用哪种字节对齐方式。我们知道 **unsigned long** 型变量 **dNumber** 由从低到高四个字节组成。类似答案第一种：**4** 个字节中高字节放在低位；低字节放在高位的字节排列方式被称为“大端对齐”。类似第二种答案：**4** 个字节中低字节放在低位，高字节放在高位的字节排列方式被称为“小端对齐”。不同的计算机系统采用的字节对齐顺序不一定相同，编写代码时，如果牵涉到对大数据类型的拆分，就尤其需要关心这个问题。我们所使用的 AVR 单片机采用的是“小端对齐”，这点一定要记牢哦。

计算机系统中，数据类型相关的细节中，除去字节的对齐顺序以外，还有字节的对齐方式，例如 2 字节对齐、4 字节对齐、8 字节对齐等等。这类信息在网上非常普遍。有兴趣的读者可以配合 sizeof 运算符以及地址绑定技术自己研究一下。

——老编 注

2.10 内存入侵

(^_^DEMO^_^)

2.11 extern：“在想你的三百六十五天……”

(^_^DEMO^_^)

2.12 人去楼空的“野指针”

军事上，弹道导弹打不中目标，多半不是因为火控系统存在多大的误差，天气有多么的糟糕，而在于，目标可能是活动的——按图索骥找不到移动的堡垒；生意场上，投资存在巨大风险，通常不是因为合作方的运作有多糟糕、产品质量有多差、管理有多落后，而在于，皮包公司的存在——昨天还是经理、秘书、职员熙熙攘攘做一屋，交了货款以后，我们今天去提货就发现已经“人去楼空”了；程序运行结果总是不对，多半不是因为“使用指针有多么危险、多么不可靠、多么难以把握”，而在于，指针指向的存储空间可能是临时性的——一个函数内部的局部变量，在函数执行完毕以后，就会被自动释放，那么指向该局部变量的指针，在被作为参数返回以后，就只能是一个“无家可归”的“野指针”了。

IT IS ONLY A DEMO EDITION

```
//一个造成“野指针”的典型例子
unsigned char *Functions(void)
{
    unsigned char Number = 0;          //函数内的局部变量
    unsigned char *p = &(Number);      //指向该局部变量的指针
    ……
    return p;                          //一个野指针诞生了
}
```

很多原因都可以造成“野指针”，对于初学者来说，如果试图采用背公式的方法记录下所有可能造成野指针的代码形式，那么这种尝试往往是徒劳的，因为能记住的代码形式是有限的，而造成野指针的可能性是无穷尽的——“一个野指针倒下了，千百个野指针站起来了”。只有在编写代码时，仔细考察“指针的有效范围”以及“指针所指向变量的有效范围”避免指针范围超越变量生存范围的情况出现，才能从根本上降低“野指针”存在的可能。

上面的函数例子中，由于指针被作为函数的返回值，因此该指针的有效范围不仅仅局限于函数体内部；而指针所指向的变量是一个

局部变量，其有效范围仅限于函数执行期间的函数体内部，一旦函数执行结束，变量本身就被释放。在这种情况下，指针的有效范围大于其指向变量的有效范围，因而，导致了一个野指针的诞生。

不可否认，上面关于野指针的例子来源于 **C** 语言学习的经典，让人觉得仿佛野指针只有在这种非常罕见或者类似的情况下才会发生。实际应用中，情况要更糟。下面就是一个更为典型的例子，有理由相信，在指针学习中，没有任何一个错误比它还要“流行”：

```
int *p;
int a;
……
a = 100;
(*p) = a;
```

如果您一眼就看出了错误的所在并正准备捧腹大笑，我们还是希望有机会给那些仍然一脸迷惑的朋友们作一个明确的解释：上面的程序片断中，指针 **p** 并没有得到初始化（也就是说，指针没有指向实体），它的值可以是任意的（好心的编译器有时候会自作聪明将其置为 **NULL**），换句话说是它就是一个野指针。对一个野指针进行任何访问操作都是极其危险的。通常，程序在编译的时候我们并不会看见到任何警告，一旦程序得到运行，“**XX** 地址不可访问”的错误提示往往令我们无所适从。

对于这种错误，检查起来非常简单，记住一个原则：指针必须要指向实体——无论你是在初始化的时候给予，还是后来某个时刻给予——实体和指针的初始化一样都是必不可少的。

IT IS ONLY A DEMO EDITION

进阶 阅读 <<阅读提示：进阶阅读着重从软件和工程的角度提供一些阅读材料，众口难调，您请酌量添加

2.13 分支程序

也需你已经习惯于如下的代码结构：

```
//一个分支程序的例子
……
unsigned char cCurrentStatus = 0;
……
switch(cCurrentStatus)
{
    case 0:
```

```
            Process_A();                    //执行操作 A
            break;
        case 1:
            Process_B();                    //执行操作 B
            break;
        case 2:
        case 3:
            Process_C();                    //执行操作 C
            break;
        ……
        default:
            Process_Default();              //执行默认的处理函数
            break;
    };
```

在这段程序中，随着状态变量**cCurrentStatus**内容的变化，程序会在每次执行**switch**语句时转向不同的程序分支——很多状态机系统正是利用类似的结构实现的。如何使用**switch**语句实现分支程序同状态机的实现方法一样，都不是本节所要讨论的问题。通过上面的代码，我们希望大家认同这样一个事实：在这类结构中，一个状态总是对应一个唯一的处理函数；与此同时，一个处理函数必然有至少一个状态与之对应。也就是说，一个状态只能对应一个处理函数，但同一个处理函数却可以对应不同的状态（例如前面程序中的状态2和状态3，它们都唯一的对应着同一个处理函数**Process_C**）。理解这一事实是我们展开后续讨论的前提。

IT IS ONLY A DEMO EDITION

2.14 消息地图与函数指针

(^_^DEMO^_^)

2.15 态内存分配 ABC

- 在 ICC 中如何正常的进行动态内存分配

在 **ICC** 中，我们是无法直接使用 **C** 语言基本库函数 **malloc()**和 **free()**进行动态内存分配和释放——即便包含了 **C** 语言标准库"**stdlib.h**"也是一样。这是因为，这里的动态内存分配是一种堆式分配。为了节省空间开支，**ICC** 没有为我们开辟默认的堆空间用于动态内存分配。使用这一功能，我们需要人工添加一个（或多个）堆。

在 **ICC** 提供的 **C** 语言标准库函数 **stdlib.h** 中，有一个非标准的函数 **_NewHeap**，它的原形为：

```
void _NewHeap(void *start, void *end);
```

通过该函数，我们只需要提供堆的起始地址和终止地址就可以将堆添加到

系统中，所有动态分配的内存，都将从这一空间划分出去。例如：

```
extern char _bss_end;
_NewHeap(&_bss_end+1, &_bss_end + 201);        //添加 200 字节的堆空间
```

上面的代码中，我们添加了一个大小为 **200** 字节的堆。如果这是第一次向系统中添加堆空间，那么从此以后，我们便能正常地使用 **malloc** 函数动态的获取我们需要的存储空间了。需要补充说明的是，函数 **_NewHeap** 可以被多次使用，向系统中添加若干个堆，而这些堆并不要求是相互连续的。例如在上述代码的基础上，我们还可以追加一段数组空间作为堆，加入到系统中：

```
unsigned char OurArray[100] = {0};
_NewHeap(OurArray,OurArray+100);               //将数组作为堆，进行添加
```

这里，我们需要补充强调一下：调用函数 **_NewHeap()** 时必须传递两个参数：一个是堆的起始地址，一个是堆的终止地址。如果我们试图将某种类型的数组作为堆添加到系统中，那么第一个参数使用数组名（所代表的地址）是没有任何问题的。关键在于，函数要求我们同时给出数组的终止地址，这时我们就必须进行地址计算——由数组名代表的起始地址加上数组元素的个数来获取数组的终止地址，注意，这里加上的不是数组的实际字节数，而是数组元素的个数。根据本章前面指针介绍过的指针加减知识容易理解，数组名所代表的地址是有类型区别的，对该地址进行加减计算时，每加减 **n** 的数字量，其计算出的地址结果都相当于该地址加减了 **n * sizeof**(数组类型)个字节。此时，如果我们自作聪明的在数组名称代表的地址上增加数组的实际字节数作为偏移量，那么最终得到的结果，可能已近相差了十万八千里——对这一错误毫不知情的系统，在进行动态内存分配时，有发生内存入侵的可能。例如：

IT IS ONLY A DEMO EDITION

```
unsigned int OurArray[100] = {0};
_NewHeap(OurArray,OurArray+100);      //实际添加了 200 个字节
```

通过 **_NewHeap**，我们将指定了起始和终止地址的一段连续空间添加到系统中用于动态内存的分配。为了提高空间的利用率，对于已经不再使用的空间，我们应该及时通过 **free()** 函数进行释放。对 **free** 函数来说，他需要我们提供一个有效的地址——该地址属于一个已经被分配出去的动态内存。对相当一部分的 **C** 语言来说，向 **free** 函数传递一个已经被释放过了的空间地址，或者一个根本就不属于堆空间的地址并不一定会对程序的运行有任何的影响，但是对 ICC 目前版本（截止到 ICC7.13A）的 free 函数来说，这种"欺骗"是不被允许的。如果你为了保险在程序中不小心对于同一个空间释放了两次，那么最后一次调用 free 将导致死循环。如果你的程序中恰巧开启了看门狗，后果不言自明。

● 如何动态的建立大小已知的二维数组（多维数组）

动态创建一个大小已知的二位数组，首先，我们需要利用 **typedef** 建立一个自定义类型。通过该类型我们可以描述二维数组的信息。假设我们要创建一个 **4*10** 的 **int** 型二维数组，则对应的代码为：

```
//建立自定义类型，描述而维数组信息
typedef   int   ARRAY[4][10];
```

接下来，我们声明一个指向该类型二维数组的指针——也就是一个“指向数组的指针”：

```
//建立数组指针，用于保存分配到的空间地址
ARRAY      *pArr = NULL;                    //声明一个指针，并初始化为 NULL
```

接下来，我们通过 **malloc** 函数申请数组空间：

```
pArr = (ARRAY *)malloc(sizeof(ARRAY));
if (pArr == NULL)                           //检查空间是否分配成功
{
    //这里插入空间申请失败的处理代码
}
```

通过以上步骤，我们成功地实现了动态创建已知大小的二维数组。这种方法实际上也适用动态创建已知大小的多维数组。对于申请到的空间，我们可以利用指向数组地指针直接进行访问，例如：

IT IS ONLY A DEMO EDITION

```
//访问动态的二维数组的例子
(*pArr)[2][4] = -123;                   //利用指向函数的指针访问数组
```

- **如何动态的建立大小可变的一维数组**

动态创建可变大小的一维数组，其关键在于正确计算数组所需的空间大小。假设，我们要创建一个 **float** 型的数组，该数组的大小由变量 **n** 来动态设定：

```
//动态创建一维数组
float *pArr = (float)malloc(sizeof(float) * n);
if (pArr == NULL)                           //检查空间是否分配成功
{
    //这里插入空间申请失败的处理代码
}
```

访问数组时，直接通过指针 **pArr** 即可：

```
//访问动态一维数组的例子
pArr[1] = 3.1415926;                    //访问数组时需要注意不要溢出
```

- 如何动态的建立第二个维度大小可变的二维数组

(^_^DEMO^_^)

- 如何动态的建立两个维度同时可变的二维数组

(^_^DEMO^_^)

IT IS ONLY A DEMO EDITION

实例 20　端口位操作的实现

第 1 步
需求分析

[需求分析]

在第三篇第一章中，我们详细介绍了 C 语言位运算。通过这些运算，我们可以单独对 AVR 的某一个引脚进行控制而不影响同一端口的其他引脚。例如：

```
//使用位运算操作端口的例子
PORTB |= BIT(PB5);            //将 PB5 单独置为高电平
PORTB &= ~BIT(PB6);           //将 PB6 单独置为低电平
PORTB ^= BIT(PB7);            //将 PB7 单独取反
```

通过这一例子，我们可以看出：使用位运算操作端口时，置位和清零需要使用不同的运算符，难以采用统一的运算形式。理想状态下，我们希望通过下面的途径来进行端口操作：

```
//理想状态下的端口操作
PB5 = 1;                      //将 PB5 单独置为高电平
PB6 = 0;                      //将 PB6 单独置为低电平
PB7 =~PB7;                    //将 PB7 单独取反
```

IT IS ONLY A DEMO EDITION

习惯上，我们把这种操作称为端口位操作。令人惋惜的是，ICC 本身并不支持这种书写方式。所幸，借助 C 语言的一些语法和结构，在牺牲一定代码执行效率的情况下，我们可以自己实现端口位操作。

实际上 ICC 对位运算方式进行的端口位操作进行了优化，当 1 行语句中只修改端口的 1 个位时，编译器将直接使用 SBI、CBI 等位操作指令“翻译”C 语句，从而避免了“读-修改-写”操作的麻烦；如果所操作的位多于 1 个，则使用“读-修改-写”的策略编译。

为了使 C 语言的书写方式更加易读，我们加入了位操作的实现，实际上这种操作是以牺牲代码效率换取代码的可维护性。

——DA859 注

[原理解析]

实现端口位操作，其基本思想就是把端口寄存器通过内存入侵的办法强制绑定为一个位段。通过这个位段，我们就可以实现对该寄存器的位操作了。

AVR 中所有和端口有关的寄存器都直接映射在 SRAM 的地址空间中，简单的说，通过访问 SRAM 的某一个特殊的固定地址，我们就能直接访问寄存器。对这些寄存器来说，SRAM 中的地址就好比它的门牌号码。知道了门牌号码，就能够找到对应的寄存器。

在本章前面的论述中，我们介绍过一下的两个公式：

```
指针常量 = ((类型信息 *)数值常量);
全局变量 = (*(指针常量));
```

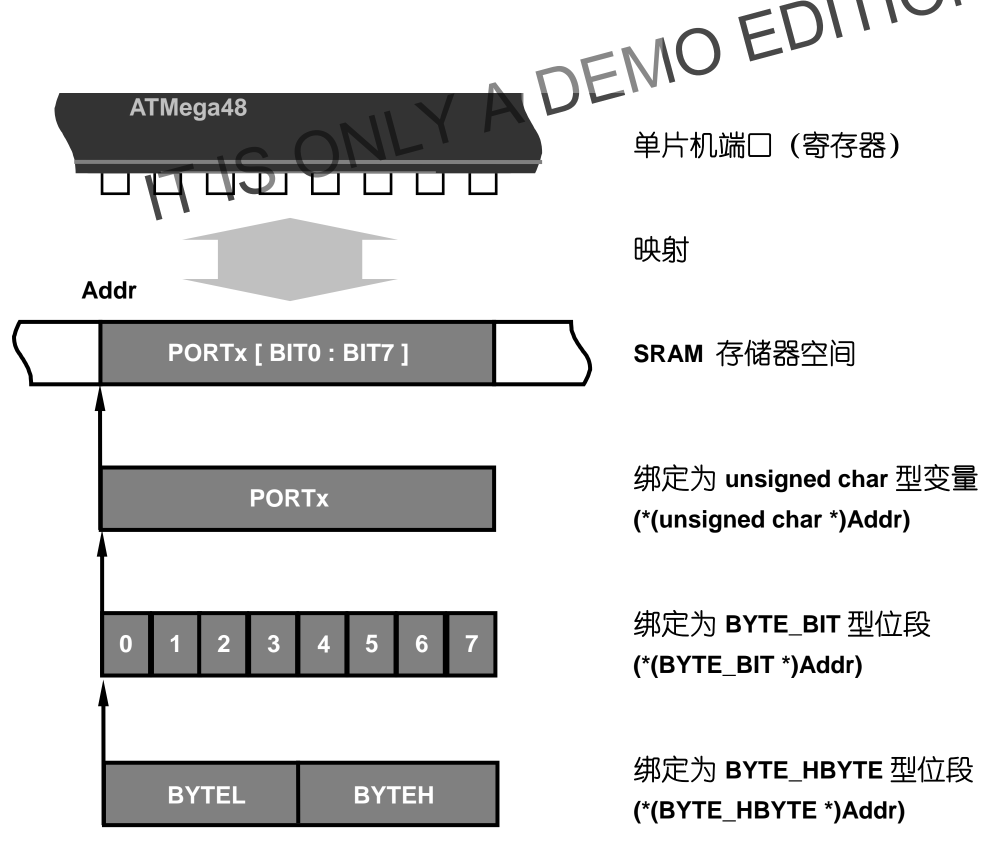
ICC 中，系统在头文件 **iomXX.h** 中利用上述公式，定义了一些地址常量，并通过这些地址常量进一步定义了寄存器变量。例如：

```
/* Port B */
#define PINB    (*(volatile unsigned char *)0x23)
#define DDRB    (*(volatile unsigned char *)0x24)
#define PORTB (*(volatile unsigned char *)0x25)
```

上面的例子中，系统利用地址绑定的方法为我们定义了三个寄存器变量 **PINB**、**DDRB** 和 **PORTB**。它们都是 **unsigned char** 型变量，并经过关键字 **volatile** 的修饰。由此我们便很容易联想：如果我们将这些地址绑定到位段而不是整型变量上，是否就意味着我们可以利用位段的访问方式，对该地址的数据进行位操作了呢？答案是肯定的。

借助位段、地址绑定技术和一些宏定义，我们很容易就可以实现端口位操作（如图 **20.1** 所示）。原理走通了，接下来就是如何编写一个通用头文件的问题了。首先，不同型号的 **AVR** 单片机的寄存器地址是不同的，因此，利用具体地址来进行绑定，势必要求我们为每一个 **AVR** 单片机型号都编写对应的头文件，工作量太大，不可取。其次，我们注意到，**AVR** 单片机的端口拥有相同的命名规则：控制端口方向的寄存器称为 **DDRx**、控制

IT IS ONLY A DEMO EDITION



例图 **20.1** 端口位操作原理示意图

单片机输出电平的寄存器为 **PORTx**，用于读取外部实际电平的寄存器则用 **PINx** 来表示。其中 **x** 为任意英文字母。**ICC** 实际上已经在头文件 **iomXX.h** 中为我们定义好了 **DDRx**、**PORTx** 和 **PINx**。也就是说，只要对这些变量进行取地址运算“**&**”就能获得所需的寄存器地址。由此，我们得出解决方案，利用系统提供的头文件 **iomXX.h**，以条件编译的方式，编写一个从 **A** 到 **G** 的端口描述文件，即可轻松实现端口位操作。

(^_^DEMO^_^)

依次对所有的端口寄存器都进行这样的改造，我们就获得了梦寐以求的端口位操作。

(^_^DEMO^_^)

[核心源代码]

(^_^DEMO^_^)

[应用举例]

第 5 步
结果验证

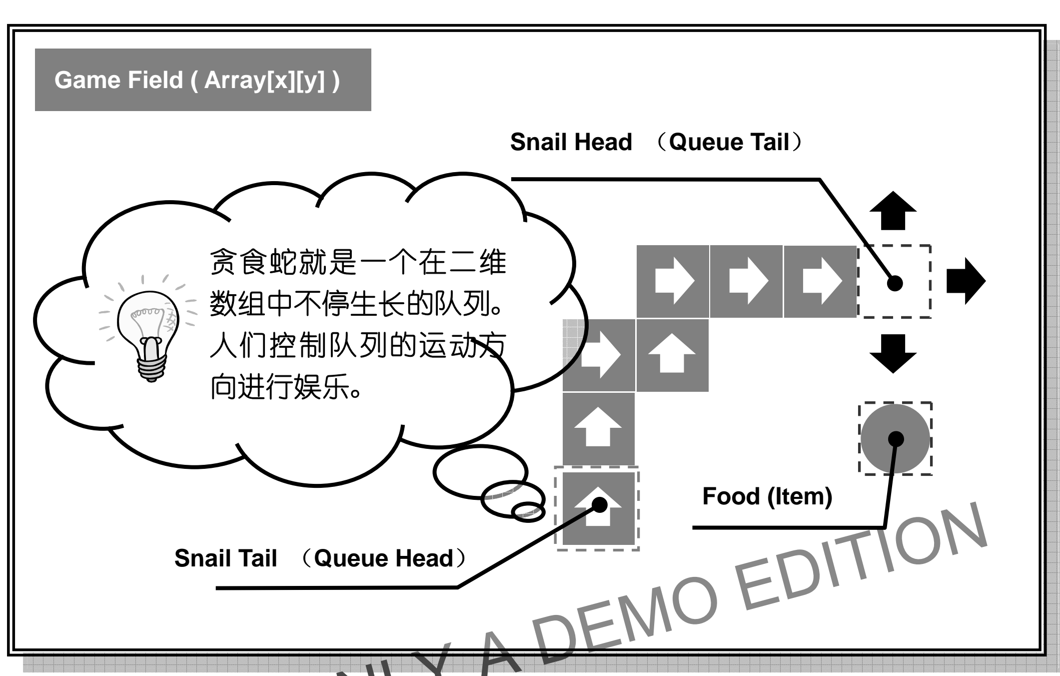
假设，我们现在要使用 **ATMega48** 单片机，为了支持端口位操作，我们可以在程序中包含上面的头文件 **UseAVRPortBit.h**：

IT IS ONLY A DEMO EDITION

```
//一个应用 AVR 端口位操作的例子
# include <iom48v.h>
# include “UseAVRPortBit.h”   //这句话一定要出现在文件 iomXXX.h 的后面
……
DDR_PC5 = 0;_PC5 = 1;         //PC5 置为输入状态，开启上拉电阻
DDR_PC6 = 1;_PC6 = 0;         //PC6 置为输出状态，低电平输出
……
if (PIN_PC5 == 1)             //读取 PC5 引脚上的电平
{
        ……
}
……
```

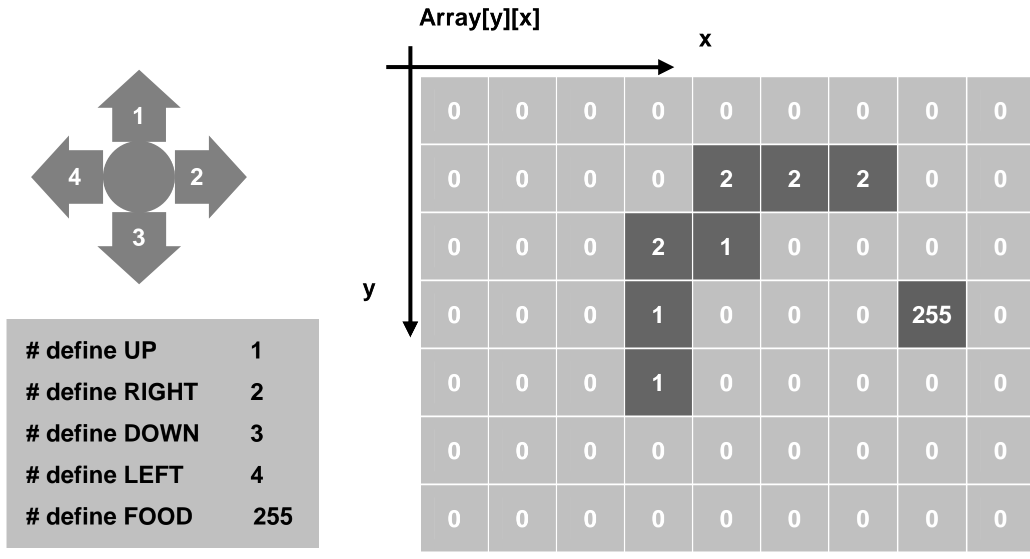
实例 21 贪食蛇

[原理图解]



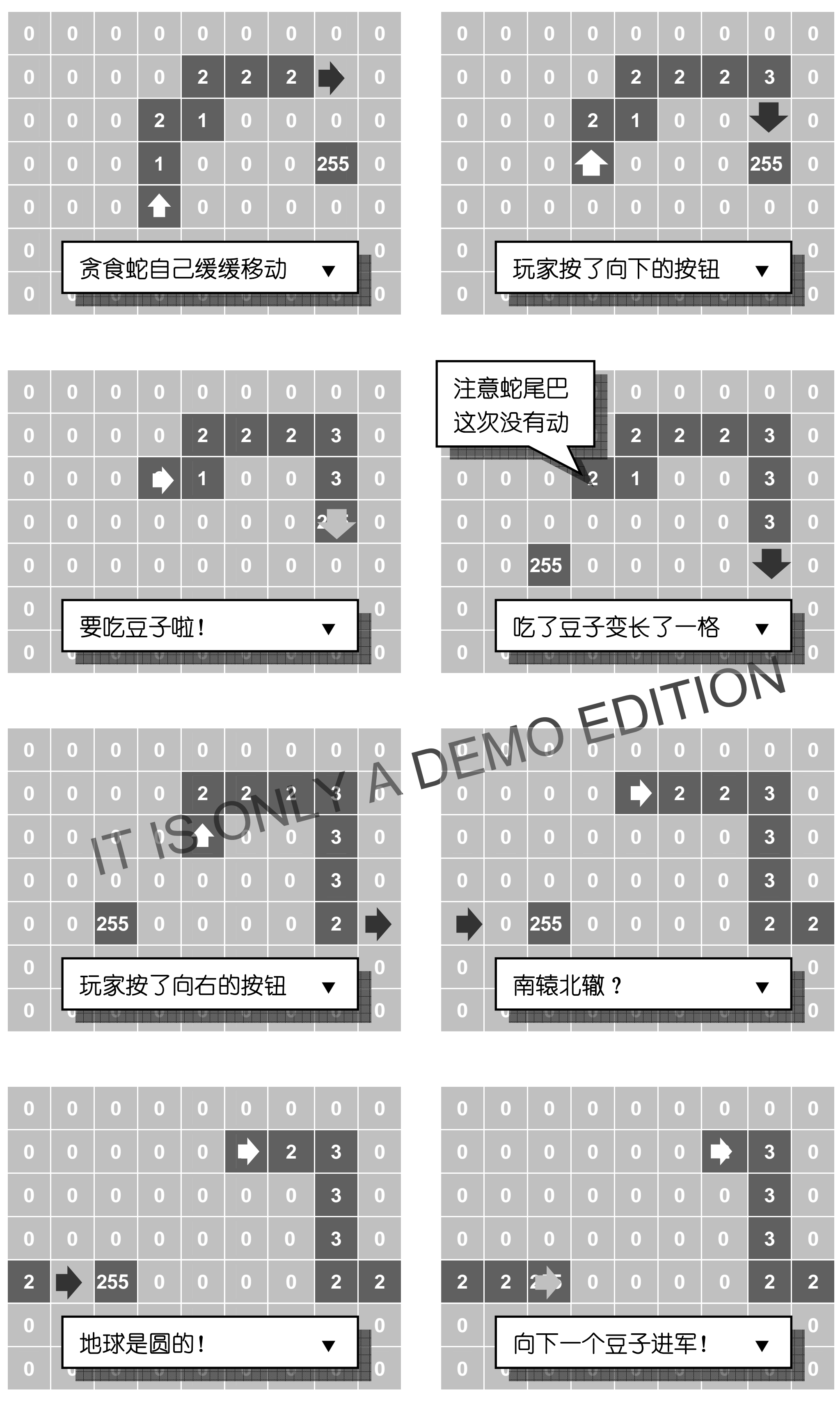
IT IS ONLY A DEMO EDITION

例图 21.1 贪食蛇数据结构原理示意图



0	0	0	0	0	0	0	0	0
0	0	0	0	2	2	2	0	0
0	0	0	2	1	0	0	0	0
0	0	0	1	0	0	0	255	0
0	0	0	1	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

例图 21.2 贪食蛇存储结构示意图



IT IS ONLY A DEMO EDITION

例图 21.3 图解贪食蛇的运动

[游戏流程图]

作为一个讲述编程方法的实例，这里没有加入额外的文字说明，希望大家喜欢这种图示+伪代码的说明方法。

——老编 注

开 始

游戏初始化：初始化环境变量、清空场地、绘制小蛇，产生第一个豆子

蛇头是否碰到了自己或者其他障碍物？

是

否

蛇是否吃到了豆子？

是

否

吃豆子计数器加一
产生一个新豆子

根据蛇尾上记录的蛇身运动方向，移动蛇的尾巴

获取用户按键信息，并根据这一信息，改变蛇头的运动方向

游戏延时

游戏主循环

结 束

IT IS ONLY A DEMO EDITION

例图 **21.4** 贪食蛇游戏流程示意图

[系统伪代码]

```
//游戏相关的一些宏
# define GROUND          0x00
# define UP              0x01
# define RIGHT           0x02
# define DOWN            0x03
# define LEFT            0x04
# define FOOD            0xff

# define GAME_FIELD_HEIGHT        7        //场地的总高度
# define GAME_FIELD_WIDTH         9        //场地的总宽度

//按键键值
# define KEY_NULL            0x00
# define KEY_UP              0x01
# define KEY_RIGHT           0x02
# define KEY_DOWN            0x03
# define KEY_LEFT            0x04
# define KEY_PAUSE           0x05
```

IT IS ONLY A DEMO EDITION

```
//游戏相关的全局声明
typedef struct SnakeBody
{
    unsigned char x;
    unsigned char y;
}SNAKE_BODY;

//声明一个数组用于描述场地
unsigned char Array[GAME_FIELD_HEIGHT][GAME_FIELD_WIDTH] = {0};

//声明蛇头和蛇尾
SNAKE_BODY      SnakeHead = {0};         //队列的尾部
SNAKE_BODY      SnakeTail  = {0};        //队列的头部

//游戏延迟，如果把它改为 0，你将获得一个风驰电掣的贪食蛇，单位毫秒
unsigned int GameDelay = 100;
//记录我们吃了多少个豆子
unsigned int FoodCounter = 0;
//当前蛇的运动方向
unsigned char SnakeGoDirection = RIGHT;
```

```
//系统用到的一些外部函数
```

```
//该函数用于在制定的位置（x,y）绘制黑色的小正方形
extern void Draw_Box(unsigned char x,unsigned char y);
//该函数用于擦除指定位置的(x,y)小正方形
extern void Clear_Box(unsigned char x,unsigned char y);
//该函数用于从键盘缓冲区中获取一个按键键值,NULL 表示没有按键
extern unsigned char Get_Key_Code(void);
//系统毫秒延迟函数
extern void Delay_MS(unsigned int DelayTime);
//随机函数，能够产生 0-n 之间的任意整数
extern unsigned char RND(unsigned char n);
```

```
//随机产生食物
void Put_New_Food(void)
{
    //获取一个空地上的(x,y)坐标
    do
    {
        x = RND(GAME_FIELD_WIDTH);
        y = RND(GAME_FIELD_WIDTH);
    }
    while(Array[y][x] != GROUND);
    Array[y][x] = FOOD;                    //在数组中标记食物
    Draw_Box(x,y);                         //画出食物
}
```

IT IS ONLY A DEMO EDITION

```
//游戏初始化函数
void Game_INIT(void)
{
    unsigned char x = 0, y = 0;

    //初始化游戏环境变量
    FoodCounter = 0;
    GameDelay = 100;
    SnakeGoDirection = RIGHT;

    //打扫游戏场地
    for (y = 0;y < GAME_FIELD_HEIGHT;y++)
    {
        for (x = 0;x < GAME_FIELD_WIDTH;x++)
        {
            Array[y][x] = GROUND;
        }
    }
```

```
        //设置蛇头的初始位置
        SnakeHead.x = 5;
        SnakeHead.y = 3;

        //设置蛇尾的初始位置
        SnakeTail.x = 3;
        SnakeTail.y = 3;

        //画出最初的蛇
        Array[3][4] = RIGHT;
        Draw_Box(4,3);                    //画出蛇的身体
        Array[3][3] = RIGHT;
        Draw_Box(3,3);                    //画出蛇的身体

        //随机产生一个食物的位置
        Put_New_Food();

    }
```

```
    //游戏的主程序
    void Game_Main(void)
    {
        unsigned char x = 0,y = 0;
        unsigned char UserPressKey = KEY_NULL;
        Game_INIT();                      //游戏初始化

        while(1)                          //游戏主循环
        {
            //获取蛇头位置
            x = SnakeHead.x;
            y = SnakeHead.y;
            //判断是否发生了碰撞
            if ((Array[y][x] != GROUND) && (Array[y][x] != FOOD))
            {
                //撞到了自己或者 UFO，GameOver
                return ;
            }

            if (Array[y][x] != FOOD)
            {
                //处理尾部运动方向
                x = SnakeTail.x;
                y = SnakeTail.y;
                switch (Array[y][x])      //判断尾部的运动方向
```

IT IS ONLY A DEMO EDITION

```
        {
            case UP:
                if (y == 0)
                {
                    y = GAME_FIELD_HEIGHT;
                }
                y--;
                break;
            case RIGHT:
                x++;
                if (x == GAME_FIELD_WIDTH)
                {
                    x = 0;
                }
                break;
            case DOWN:
                y++;
                if (y == GAME_FIELD_HEIGHT)
                {
                    y = 0;
                }
                break;
            case LEFT:
                if (x == 0)
                {
                    x = GAME_FIELD_WIDTH;
                }
                x--;
                break;
            default:
                break;
        }
        Array[SnakeTail.y][SnakeTail.x] = GROUND;
        Clear_Box(SnakeTail.x,SnakeTail.y);         //清除尾巴
        SnakeTail.x = x;
        SnakeTail.y = y;
    }
    else
    {
        FoodCounter++;                    //吃下的豆子增加了一个
        Put_New_Food();                   //再产生一个新豆子
    }

    //获取用户按键
```

IT IS ONLY A DEMO EDITION

```
UserPressKey = Get_Key_Code();
//贪食蛇肯定没有倒档，这里用了环形影射加定义域平移的方法
if (UserPressKey != ((SnakeGoDirection -1 + 2) % 4) + 1)
{
        //判断按键
        switch (UserPressKey)
        {
              case KEY_UP:
              case KEY_DOWN:
              case KEY_RIGHT:
              case KEY_LEFT:
                   SnakeGoDirection = UserPressKey;
                   break;
              case KEY_PAUSE:
                   //任意键继续
                   while(Get_Key_Code() == KEY_NULL);
                   break;
              case KEY_NULL:
              default:
                   break;
           }
}

//处理头部运动方向
x = SnakeHead.x;
y = SnakeHead.y;
switch(SnakeGoDirection)        //判断蛇头运动方向
{
      case UP:
            Array[y][x] = UP;
            Draw_Box(x,y);            //加入一个新元素到队列
            if (y == 0)
            {
                  y = GAME_FIELD_HEIGHT;
            }
            y--;
            SnakeHead.y = y;          //蛇头移动
            break;
      case RIGHT:
            Array[y][x] = RIGHT;
            Draw_Box(x,y);            //加入一个新元素到队列
            x++;
            if (x == GAME_FIELD_WIDTH)
            {
```

IT IS ONLY A DEMO EDITION

```
                    x = 0;
                }
                SnakeHead.x = x;        //蛇头移动
                break;
            case DOWN:
                Array[y][x] = DOWN;
                Draw_Box(x,y);          //加入一个新元素到队列
                y++;
                if (y == GAME_FIELD_HEIGHT)
                {
                    y = 0;
                }
                SnakeHead.y = y;        //蛇头移动
                break;
            case LEFT:
                Array[y][x] = LEFT;
                Draw_Box(x,y);          //加入一个新元素到队列
                if (x == 0)
                {
                    x = GAME_FIELD_WIDTH;
                }
                x--;
                SnakeHead.x = x;        //蛇头移动
                break;
        }

        //在这里进行游戏延时，修改 GameDelay 的值将改变游戏难度
        Delay_MS(GameDelay);
    }
}
```

IT IS ONLY A DEMO EDITION

第零篇 如何阅读本书

本书的结构

大家好！初次见面，请多多关照！感谢这本书能够让我们认识这么多新朋友，衷心祝愿大家能够和我们共同度过一段美好的单片机学习时光。

大家可能会注意到，这本书没有介绍微型计算机的基本概念、基本工作原理，甚至没有把 **ATmega48/88/168** 单片机的介绍放在最前面，没有介绍这种单片机的优点所在！

这的确是一本不同于传统观念的单片机书籍！

在这个单片机产业群雄争霸的时代，评价一种单片机优劣的权利在研发工程师手中而不在我们这些作者手中。我们所希望做的，是以 **ATmega48** 单片机为载体，向大家介绍学习单片机技术的一般性方法。

下面给大家介绍一下本书的基本结构和所建议的阅读方法。

IT IS ONLY A DEMO EDITION

- 篇章设置

第一篇：介绍学习单片机的准备知识。

如果您是单片机初学者，仅仅浅尝过 **51** 单片机的课程学习；如果您是单片机爱好者，却感觉无从入门。那么请您跟随本书一起，通过 **DATASHEET** 认识 **ATmega48/88/168** 单片机，亲自动手焊接一根下载线、一块实验板。这些自制的学习工具将伴随我们跨入单片机世界的大门。

读者在阅读本书的同时需要阅读 **DATASHEET**，本书中仅仅剖析了 **DATASHEET** 中的难点和重点。

第二篇：单片机重点外设的学习。

单片机最为有用的，不是他的计算能力，而是丰富的外设资源。在这一篇里我们介绍常用的单片机外设在实际中的应用。

在每一个外设的介绍之后，我们安排了若干个应用实例。这些实例非常详细，提供了方案分析、硬件电路图、流程图、重点难点代码的分析。在本书的光盘中给出了各个实例的全部源代码，这些源代码都是由作者调试过的，直接下载到单片机中就可以运行。

第三篇：单片机软件开发模式介绍。

在本书中，软件语言以 **C** 语言为主，但单片机上的 **C** 语言与 **PC** 机上的环境不甚相同。本篇的目的就是让大家能在已有的 **C** 语言基础上尽快进入单片机的软件开发模式。

在这一篇中，我们温习了 **C** 语言的部分知识点，但是这些知识点不是重复通常 **C** 语言教材中的内容，而是针对单片机的应用特点做了发散。

● 本书中的一些阅读提示

在本书中，你会经常看到以下几种阅读提示：

基础 知识 <<阅读提示：基础知识试图希望您拥有一个扎实的基础，这是以后学习的关键

或

原理 解析 <<阅读提示：原理简析部分试图将艰涩难懂的理论做浅显化的讲解，高手可以跳过

这些部分，将介绍本章中的基础知识，或者用简单的事例作为比喻来说明软硬件的工作原理。如果您已近有一定的单片机基础知识，可以跳过这些部分。

实际 应用 <<阅读提示：实际应用部分着重介绍该资源在实际应用过程中常见的方法，入门必读

具体到 **ATmega48** 单片机的介绍，例如阐述某个外设在 **ATmega48** 单片机上的具体操纵方法。在介绍中包括了许多实际的问题，这种先阐述原理，再具体到单片机介绍的方法也是本书“以 **ATmega48** 单片机为载体，向大家介绍学习单片机技术的方法”思路的体现，我们希望读者通过本书学习到的不仅是 **ATmega48/88/168** 单片机，而是一种学习单片机的通用方法。

IT IS ONLY A DEMO EDITION

进阶 阅读 <<阅读提示：进阶阅读着重从软件和工程的角度提供一些阅读材料，众口难调，您请酌量添加

留给初学者的上升空间。阅读这部分对初学者来说可能有一定的难度，可以暂时跳过，待他们对单片机技术学习有一定深入时，再来研究这些相对复杂的问题。

A/D 是“Analog To Digital Converter”的缩写，中文名称是“模拟 / 数字转换器”，他的职责是将模拟信号量按照一定的规则转换为数字信号值，以使得数字电路或者单片机能够处理模拟信号量。

——DA895 注

这是本书的旁注。大家可能会注意到，我们的文字并没有占满整个页

面，在两侧为大家留足了批注的空白区域，其中有的批注，我们在编写的时候已经帮大家加上去了，他们的落款分别是“**DA895**”和“傻孩子”两位同学或老编。

“**DA895**”主要负责批注硬件相关的问题，“傻孩子”则主要负责软件相关的问题。这些问题包括名词解释，额外的提示等。这些信息本身不是本书要讨论的问题，但是通过批注的形式写在页面旁边，可以节省读者查阅其他资料的时间。同时也让他们两位同学陪着大家一起完成学习过程。

本书中的虚拟人物介绍

为了让单片机的学习过程更加生动活泼，我们安排了三位虚拟人物穿插在全书中，他们分别是编辑部的“老编”和 **DA895**、傻孩子两位同学。他们将加入到本书的讲解和旁注工作中，希望他们的加入能让本书更好地为读者服务。

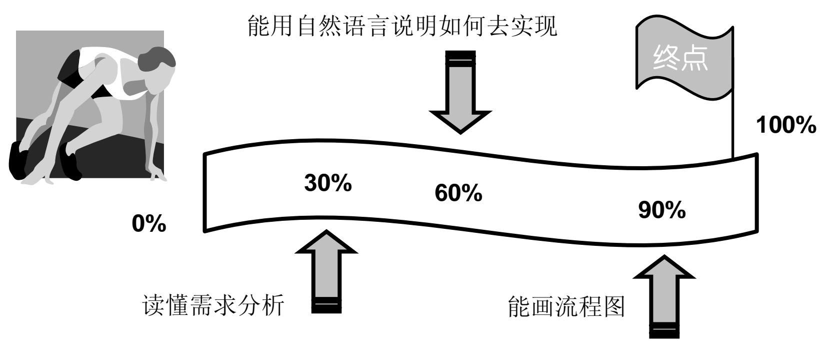
如何阅读实例部分

实例 1　阅读 74HC595 DATASHEET

IT IS ONLY A DEMO EDITION

如果您在本书中看到这样的标题，他就是实例了。详细的实例是本书的特点之一。在每个章节的中，您都会找到这种实例的标示。

众所周知，单片机学习重在实践。非常详细的实例是本书的特点之一。编写团队为了安排哪些试验以及他们的难度控制讨论了许久，最后按照“常用优先，先易后难，循序渐进”的原则精选出的书中的实例。我们给出了所有实例的电路图、流程图、难点源代码分析。虽然光盘中的源代码都经过了调试，直接下载到单片机中就可以运行，但是希望大家不要像学



您距离终点还有多远

校里对待单片机实验那样来对待他们，千万不能“一烧了事”。

推荐的实例阅读方法是先想后读，在阅读前先思考，如果没有后面的现成方案，我该如何实现命题中的要求；在阅读中寻找，我的思考方式和实例中的有什么区别；在阅读后反问，给出来的方法是实现命题的最佳方法吗？

最后提示大家的是，知道如何去解决一个问题比实际动手去解决这个问题更加可贵。

单片机硬件学习

单片机在经历了**20**年的辉煌发展之后，已经深入了今天大到飞机、汽车，小到自动豆浆机、电饭锅等各种电子设备之中。应该说单片机技术是当代电子设计师必备的设计技能之一。

与**80**年代的无线电爱好者不同，单片机的学习开发必须依赖于个人计算机（**PC**）。本书的作者也经历了大学里囊中羞涩的时代，好在单片机开发对 **PC** 机的要求不高，一台能够运行 **Windows98**、有串口和并口的二手电脑已经完全可以开发 **AVR** 单片机了。

除此之外就是基本的钳工工具：

IT IS ONLY A DEMO EDITION

由于单片机容易被静电损坏，焊接最好能用恒温电烙铁，这种烙铁温度可调，并且不会产生漏电损坏单片机。对于没有条件使用恒温烙铁的朋友，可以使用普通的**20W**内热式烙铁，但应该将烙铁外壳可靠接地。

有了这些工具，就可以跟着本书第一篇实例 **2** 的介绍一步一步地搭建属于自己的 **ATmega48/88/168** 单片机开发平台。

本书中没有包括 **DATASHEET** 内的详细介绍——如果把他们都写进来，本书将成为一本庞然大物，何况阅读 **DATASHEET** 最方便的地方是计算机上。读者在阅读本书的同时还需要一台 **PC** 机以阅读光盘中所附的 **DATASHEET** 及程序源代码文件。

另外您在阅读本书之外，可以再准备一本数字电路基础教材、一本 **51** 单片机基础书籍配合学习。

单片机软件学习

我们这本书的主要语言是 **C** 语言。诚然，用高级语言开发是现代单片机开发的趋势，本书也没有详细介绍 **AVR** 单片机汇编语言的指令、寻址方法等知识，这并不是说汇编语言过时了，而是因为汇编语言已经作为一种基本技能应用在底层驱动和简单系统的开发之中。

对于一个单片机开发者来说，只有学好了汇编，才能品尝到单片机的“满汉全席”，否则就只能吃 **C** 语言的“快餐”。

本书也不是一本专门讲 **C** 语言的书籍，在这个领域已经有很多教科书可以学习了。在这里我们仅仅结合单片机开发中的特点对 **C** 语言中的部分知识点作一个温习。

建议读者在阅读本书时，准备一本 **C** 语言的教材，例如谭浩强前辈的《**C** 程序设计》。

单片机的开发过程在我看来主要是软件的开发过程，读者可能会感觉到，本书中大部分篇幅在讲述软件设计。这是由于软件相对于硬件的灵活性决定的。从减少元件数量，提高可靠性的角度来看，同一个功能，用软件实现比用硬件实现更加可靠。

嵌入式系统开发，实际上就是为某一应用量身定做的“专用计算机”开发软硬件系统。牵涉到的知识除了基本的模拟电路、数字电路以外，更多的就是软件方面的知识。通常，一门精通的高级语言、初步的数据结构知识、初步的算法、一定的软件工程概念是对一个嵌入式系统从业者的最基本要求。本书定位于单片机的初学者、爱好者，因此也是以这一要求来安排软件相关章节内容的。说起玄乎，做起来不难。对于所学的知识，善于从生活中寻找它们的原形，是我们学习计算机学科的心得。

IT IS ONLY A DEMO EDITION

怎样阅读本书

如果你是一个高手，或者是一个自称的老手

请选择其它更适合您阅读的书籍。如果你真的想学习 AVR 单片机，我们认为阅读官方的 Datasheet 更直接，更经济。如果您是想为初学者寻找一本入门书籍——我们相信您的选择。

如果你是一个电子类专业的在校学生

我们推荐你先仔细阅读第一篇第一章，熟悉和了解数据手册 Datasheet 的使用方法。紧接着，您可以跳过第二篇，直接进入第三篇，温习和补充一些 C 语言知识。这一篇章中，由于省略了大部分经典 C 语言教材的语法介绍，因此，我们推荐您还需要将一本谭浩强前辈的《C 程序设计》放在手边，以备随时查阅。在此之后，你就可以自由的按照喜好，阅读第二篇中的内容了。不用过于担心章节之间的穿插关系，书的旁批中会给您指出相关知识所在的位置。

如果你是一个电子爱好者

动手实践，是我们共同的爱好。在每一个章节中，都有一个对应的实例，即便章节中的解说没有看懂，通常，实践的过程都会使我们恍然大悟。本书偏重于嵌入式系统的软件设计，这并不是说，作为一个电子爱好者，就很难读懂，相反，这本书在最初确立原则时，就是设计用来辅助软件功底稍差的电子爱好者，使大家有机会成为"软硬兼得"的高手。本书的第二篇应该是您下功夫阅读的对象。

IT IS ONLY A DEMO EDITION

如果你是一个偏向于软件的 AVR 爱好者

作为偏向于软件的 AVR 爱好者，我们最关心的是如何使用手中的语言工具最直接的去操纵硬件资源。硬件细节，往往是我们想忽略的，因此本书的代码编写是建立在"ICC 代码生成向导"之上的。通过第一篇第一章关于 Datasheet 的介绍，您应该能最快的找到您感兴趣的寄存器。同时，本书介绍了大量嵌入式系统中常见的算法，包含的内容从 C 语言到数据结构；从算法设计到操作系统；从编码规范到软件工程……您可以直接阅读本书的第三篇，相信一定有可以与您共鸣的地方。

如果你是一个在职的工程师，想接触一下 AVR 单片机

本书在编写之时，是以辅助 Datasheet 阅读为蓝本的。学习 AVR 单片机，对您来说，应该以 Datasheet 为主，在理解出现障碍或者需要实例作为参照时，可以本书作为有限的参考。

如果你曾经学习过 51 单片机

如果您习惯于用汇编语言开发 51 单片机，那么 C 语言可能是您最大的障碍。首先，快速阅读本书第三篇的内容，跳过难以一时理解的部分，迅速进入第二篇章。在这一篇章中，您可以跳过"原理解析"直接进入"实际应用"部分。俗话说，一通百通，对比曾经的学习的 51 知识，

熟悉 **C** 语言环境，对您来说，也许并不是一件难事。

如果你喜欢系统地学习一门知识

您可以按部就班的依次阅读各个章节。这对您系统掌握一种单片机、熟悉一种开发环境、了解一种工程思想、学习一类应用算法是非常有好处的。

如果你喜欢开门见山

您是一个急性子的人，凡事喜欢一针见血，透过现象看本质。在阅读时，您可以首先从各章的实例出发，遇到您感兴趣的内容时，再回头阅读章节的详细说明。跳过无用、冗繁的比喻、讲解，直接阅读如何使用代码生成向导获得所需代码、直接通过 **Datasheet** 中的寄存器说明掌握某一硬件资源的操作方式，对您来说，可能是一种更有效的方式。本书为了顾及大部分人的口味，可谓“灌水颇多”，还请您自己原谅我们这种做法，挤干“水分”，获取您最感兴趣的部分。

如果你曾经阅读过类似的技术类书籍

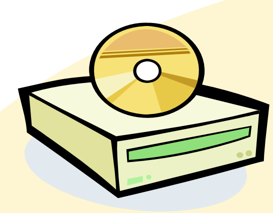
您最好找一本同类的书籍放在手边，以备不时的对照，补充需要的内容。本书在编写之时，参考了大量同类书籍，有意避开了可能“雷同”的部分。我们虽然不会是“**Datasheet** 的翻译”，但是，也因此造成了很多内容的缺失，典型的例子就是本书的“偏向软件处理的特征”。嵌入式系统设计同时包含软/硬两部分的设计，而本书没有去详细介绍专门针对 **ATmega48/88/168** 的硬件组成。配合光盘内的数据手册阅读，可能是您最好的选择。

IT IS ONLY A DEMO EDITION

如果你只是想消遣一下

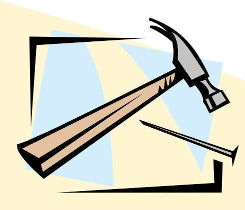
技术来源于生活，看看您身边的技术都可以用哪些身边的例子来进行类比——这通常是一件非常有趣的事情。所谓言者无心，听者有意。我们一个很不经意且并不恰当的比喻，也许能燃起您的一丝灵感。如果我们真的起到了抛砖引玉的作用，还请您一定要告诉我们、告诉身边搞技术的朋友——原来技术可以这么有趣。在本书的很多章节中，原理解析部分通常包含了很多有趣的模型和比喻，有空的话，不妨还请您驻足浏览一番。

购买本书 de 十大理由



随书光盘 内容丰富

光盘中包括所有实例的源程序代码，可直接下载运行。VB 编写的后台软件使单片机学习不再与 PC 绝缘。



大量实例 真实可用

21 个生动的实例为您展示核心外设的使用方法。为您逐项讲解 ICC 代码生成器的使用——写程序原来也可以如此享受。

以人为本 de 策划思想

从身边的事物入手，原理的讲述不再枯燥无味。在不知不觉中迈入单片机世界的大门。

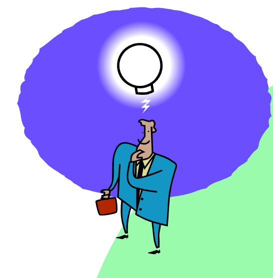
IT IS ONLY A DEMONSTRITION

知识结构 深入浅出

从最简单的端口操作爬升到复杂的 I2C 接口和 BOOT 编程，不知不觉中迈入高手行列。

DIY 实验室 图文并茂

手把手搭建自己的下载线、实验板，教你 DIY 自己的实验平台，与高昂入门成本挥手说 ByeBye！



独一无二的软件视角

本书首次站在软件的角度，为您讲述嵌入式系统开发的常见处理方法。真正告别“调通硬件资源就算入门”的老套路。

详尽的技术细节

首次详细披露 BOOT 程序细节，给出完整代码及后台软件。首次为您详尽披露 ICC 汇编接口规范。

补充课堂教学 提升就业竞争力

针对用人市场之所需，选择重点编写，助您求职就业一臂之力。

IT IS ONLY A DEMO EDITION

直面技术难题 分析客观深入

毫不隐讳 ICC 中的 BUG 及常见的编码失误，让他们成为初学者的“前车之鉴”。用 C 还是汇编？用中断还是用查询？我们逐个为您客观分析。

3 位虚拟人物 语言幽默生动

幽默生动的比喻贯穿讲解过程，即使没有单片机基础也能轻松入门。3 位虚拟人物贯穿全书，学习变得生动有趣。