

# 从单片机基础到程序框架（2019 版）

吴坚鸿 编著

捐权声明：

该教程免费授权给所有的出版社和做单片机学习板的厂家和各大培训机构以及全国各大院校，我本人不从中赢利也不收取任何版权费用，我本人也不卖书也不卖学习板也不搞线下培训。该教程的版权无偿捐给全社会。

## 第一节：我的价值观。

### 【1.1 我的价值观。】

我 2006 年毕业，2009 年就出来做自由职业者在深圳以接单片机项目谋生，到现在我有自己的机器人技术有限公司。目前公司的人机界面，运动算法，机器视觉，伺服驱动，ARM 单片机编程，DSP 编程，FPGA 编程，电路板设计，上位机软件都是由我带领的研发团队在做。我只专心做技术，而市场，生产，行政，资金，财务，采购都不用我分心去管，有我另外的合伙人雷总去打理，所以我常感恩能过上研发创作的日子，是因为有雷总的关照。

光有经济保障还是不够的，人最重要的是要找到自己的归宿自己的位置。我最爱看老子，庄子，孔子，王阳明的圣人书，王阳明说人人皆可成为圣人，所以我一直在追求圣人之道，我渴望成为圣人，圣人之道有真三不朽之说，立功，立言，立德。在立功层面，我这一辈子的定位就是做技术，我想做超级宇宙技术大牛，特别牛的那种牛，然后以我的技术协助雷总把我们的机器人公司做大做强。在立言层面，在不涉及我公司商业机密的前提下，我的天命和归宿就是做单片机技术分享，写一辈子源源不断的技术分享连载帖，然后写一本《从单片机基础到程序框架》的书，帮助更多单片机初学者，出书可以满足我在立言方面的追求。在立德层面，我平时信因果，在生活中多传播正能量。

我有自知之明，我的天命就是传播单片机技术。人最宝贵的东西是生命，生命属于人只有一次，人的一生应当这样度过：当他回首往事的时候，他不因虚度年华而悔恨，也不应碌碌无为而羞愧。在他临死的时候，他能够这样说：我的整个生命和全部精力，都献给了世界上最壮丽的事业——为传播单片机技术而奋斗。

### 【1.2 坚鸿单片机私人 QQ 群。】

这个是我私人的单片机 QQ 群，主要是用来交流工作上遇到的技术问题。群号是：184876577。群规如下：

- (1) 不许刚入群就发私信骚扰群主，有问题的请到群里聊。
- (2) 在群里提问问题时，不许点群主的名求解答，也不许用“@群主”的提问方式骚扰群主。比如，不能这样问“鸿哥，上拉电阻选多大？”，而应该去掉称呼这样问“上拉电阻选多大？”。
- (3) 不许刚入群就问关于“前途”和“发展前景”的问题。
- (4) 不许刚入群就发书本习题或者试题求助。
- (5) 不许刚入群就贴一大段代码刷屏。
- (6) 不许刚入群就求某项目全套源代码。
- (7) 早上自愿报数的时间是 6 点到 8 点，目的是活跃群氛围，其它时间段禁止报数。
- (8) 技术交流不许用语音。必须用文字，方便技术传播和交流。
- (9) 连续潜水 90 天没发言的非金星群友将会被移出群。
- (10) 在群里经常愤世嫉俗者会被移出群。

### 【1.3 相关资料下载网址。】

下载网址：<http://www.dumenmen.com>

## 第二节：初学者的疑惑。

### 【2.1 单片机应用的核心技术是什么？】

单片机应用的核心技术是什么？是按键，数码管，流水灯，串口。是它们的程序框架。按键和数码管是输入是人机界面，把它们程序框架研究透了，以后做彩屏或者更花销的显示界面，程序框架也可以通用。流水灯是应用程序是 APP，把它的程序框架研究透了，以后控制飞机大炮的程序框架也是一样。串口是通讯是接口，把它的程序框架研究透了，以后搞 SPI，CAN，USB 等通讯项目时，上层的程序框架也可以通用。如果某天你突然腰酸背痛可能是缺钙了，如果某天你第一次做项目时突然发现无从下手，你缺的可能是程序框架。

### 【2.2 跟我学单片机到底是学什么？】

跟我学单片机到底是学什么？我的回答是像驾驶汽车一样驾驭单片机。我教给大家的是驾驶汽车的技术而不是研发汽车的技术。因此每当别人问我学 51 单片机，PIC，AVR，stm32 哪个更加有前途，应该先学哪个再学哪个时，我的回答是既然你是学驾驶技术，那么用桑塔纳车来学还是用宝马车来学有差别吗？差别很小的，它们只是不同的厂家而已，只要会一种其它的就触类旁通了。把学单片机当作考驾照这是我常用的一个比喻。

### 【2.3 单片机神奇的工作原理是什么？】

单片机神奇的工作原理是什么？初学者不用纠结这个问题，这不是我们学习的方向。考驾照只要大概知道汽车是由四个轮，发动机，制动系统，离合器，方向盘等部分构成就够了，再深入的细节不用纠结。学单片机只要大概知道单片机内部由运算器，寄存器，IO 口，复位电路，晶振电路等部分组成就够了，再深入的不用纠结。说实话，我本人做单片机开发有很多年了，但是我对单片机的工作原理也很模糊，就像人为什么能通过大脑来灵活控制双手，对于我仍然是一个迷。有这样的疑惑时咋办？我建议用“游戏规则”这个概念去应付它。游戏规则是不需要解释的，只要遵守就可以了。在应用的技术领域，把暂时不解的东西当作游戏规则来解读和遵守是我常用的思维方式。

### 【2.4 很难记住繁杂的寄存器？】

很难记住繁杂的寄存器？寄存器不用死记硬背，只要知道它大概的操作流程，有哪几类就够了。配置寄存器时，可参考别人已经配置好的代码，这些代码都很容易通过网络或者书本获得。也可以查找芯片数据手册，有很多单片机厂家会给出各个功能的代码范例。

### 【2.5 汇编语言很难学怎么办？】

汇编语言很难学怎么办？我提个建议，对于初学者，一开始就学汇编语言确实难学，不如先学 C 语言，学会了 C 语言再学汇编，这样理解起来就容易多了。也可以把 C 语言列入必修课，汇编语言列入选修课，因为实际工作中也是 C 语言为主。

### 【2.6 很难记住各种繁杂的通信协议？】

很难记住繁杂的各种通信协议？IIC，SPI，232，CAN，USB 等等。不用记那么多，你只要理解串行和并

行通讯方式的基本原理就可以了，剩下的只是不同的协议而已，工作时再根据需要去看看相关资料就可以上手。不管世上有多少种通讯协议，物理世界上只有这两种通讯方式。

## 【2.7 很难写出短小精悍的程序？】

很难写出短小精悍的程序？初学者不用纠结于此。很多项目开发，程序容量不是刻意追求的目标，多一点少一点没关系，不会是寸土寸金的事情，现在大容量的单片机品种也非常多，反而更值得关注的是程序的运行效率，可读性和可修改性。当然，一些成本敏感的消费类电子不在此讨论范围，这类项目往往对程序容量也要求很苛刻。

### 第三节：单片机最重要的一个特性。

#### 【3.1 单片机的“一”。】

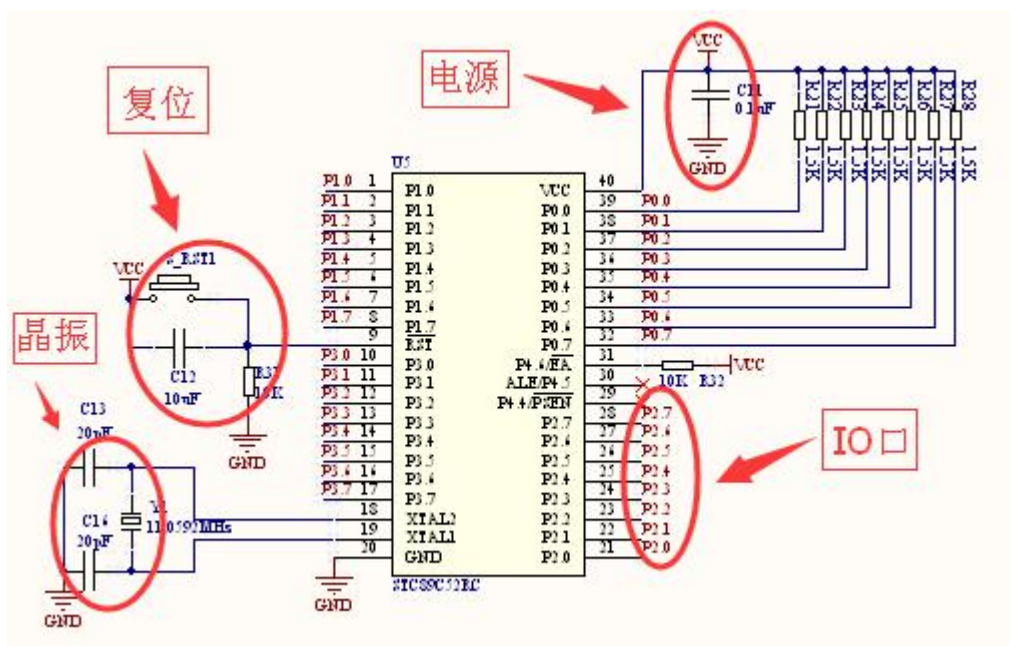


图 3.1 单片机工作的最小系统

“道生一，一生二，二生三，三生万物。”《道德经》认为，世间万物，缤纷多彩，都源自一个东西，这个“一”的东西就是“道”。电子世界也存在“一”，这个“一”繁衍出手机，电脑，电视机，机器人等丰富多彩的世界，这个“一”就是单片机最重要的一个特性：“程序下载进内存后，单片机既可以通过管脚识别外部输入的高低电平信号，也可以通过管脚对外部输出不同时间长度的高低电平。”这句话有 5 个关键词“程序，内存，管脚，电平，时间。”下面我详细解读每个关键词的含义，涉及到某些专用名词如果不理解也没关系，主要是让大家有个感性的认识就足矣。

#### 【3.2 程序。】

单片机程序有 3 种：C 程序，汇编程序，机器程序。能下载进单片机的只有机器程序，C 程序和汇编程序都不能直接下载进单片机，所以 C 程序和汇编程序都要经过编译软件翻译成机器程序后，才能下载进单片机。程序是语言，语言是用来交流，交流就必须存在两个对象，对象分别是程序员和单片机。程序员是人所以用 C 或者汇编语言，单片机是机器所以用机器语言，人和机是不同的世界，两者交流就需要一个翻译家，翻译家就是编译软件，俗称编译器，它能把 C 语言或者汇编语言翻译成单片机能识别的机器语言。机器语言就是 0101 的代码，一般以十六进制的形式呈现。

理论上，程序员也可以抛开 C 和汇编语言，直接用机器语言做项目。我读书时老师让我们做实验就是这么整。那时我还不知有烧录器，老师让我们先用汇编语言写好程序，然后自己充当编译器，对照汇编语言和机器语言的指令表，人工把汇编语言翻译成十六进制的机器语言，最后把机器语言按字节一个一个的输入到特定的实验设备来观察现象。现在回想起来，老师当时的初衷是让我们了解编程语言的本质。

既然可以直接用机器语言做项目，为什么还要 C 语言或者汇编语言？在 C 语言或者汇编语言没有诞生前，程序员就是通过在纸带上打孔来代表 01 的机器语言，此时相当于结绳记事的原始阶段。后来人类发明了汇编语言，通过英语单词来表示 01 机器语言特定的指令语句，此时开始诞生了汇编语言的编译器相当于进入象形文字的阶段。再到后来人类又发明了 C 语言，通过数学符号和英语单词来表达自己的逻辑，诞生了 C 语

言的编译器相当于进入了汉字白话文阶段，从此程序员写出来的 C 程序就非常方便移植，编辑，阅读，传播，继承。现在单片机开发的主流是 C 语言，我本人出来工作后就没有用过汇编做项目开发。C 语言是必修课，汇编语言是选修课；C 语言是白话文简单易懂，汇编语言是文言文繁琐难读。当然汇编也有它的应用场合，汇编的翻译效率高，往往是一句汇编语言对应一句机器语言，而一句 C 语言有可能对应几句机器语言，这样 C 程序的代码效率在很大程度上取决于编译器的水平，编译器能不能帮你翻译出高效的机器语言对于我们来说往往像黑盒子，不像汇编语言那么可控制。所以很多嵌入式系统某段要求简洁高效的源代码往往用汇编来写，也有少数一些很便宜的单片机不提供 C 编译器，只能用汇编语言开发。要不要学汇编，最好根据个人的工作需求来决定。

### 【3.3 内存。】

单片机就像 MP3，程序代码就像歌曲，把不同的歌曲下载到 MP3 里就可以听到不同的音乐，把不同的程序下载到单片机里，单片机就能做不同的事。能装程序的单片机必然有内存，内存由 ROM 和 RAM 组成，ROM 和 RAM 都能装东西，但各有不同。

ROM 的优点是存储的东西断电后不会丢失，缺点是存储的东西上电后不能更改，想要改变 ROM 的内容除非重新下载程序，而且下载次数有限制，FLASH 的 ROM 最大次数通常是 10 万次，而 OTP 的 ROM 只能下载 1 次，所以平时上电工作时 ROM 存储的东西是不能更改的，某些具有 IAP 功能的高级单片机不在此讨论范围内。而 RAM 恰好反过来，RAM 的优点是存储的东西上电后可以随时被单片机更改，更改次数没有限制，缺点是存储的东西断电后会丢失，没有记忆功能。

ROM 和 RAM 各有特点，单片机从中各取所长。ROM 用来存储不可更改的指令代码和常量数据，ROM 的容量往往相当于代码的容量。RAM 用来存储可以被更改的变量数据，RAM 的容量往往相当于全局变量和局部变量的容量。不管是用 C 语言还是汇编，所写的程序代码就自然包含了指令代码、常量数据、全局变量、局部变量，那么谁在幕后帮我们进行分类存储，谁把一个程序代码的一分为二让它们在 ROM 和 RAM 里各就各位？是编译器软件和下载器（烧录器），编译器除了把 C 语言翻译成机器语言之外，还帮我们分好了类，分配好了存储的地址和位置，下载器（烧录器）再根据这些信息把程序存储到内存中，这些工作一般不用程序员干预，它们自动完成。

### 【3.4 管脚。】

管脚是单片机与外部电路进行能量和信息交互的桥梁。有电源，复位，晶振和 I/O 口这 4 类管脚。

第一类电源管脚。是给单片机内部电路供电的接口。单片机有两种常用的供电电压，一般是 3.3V 或者 5V，有的单片机两种电压都兼容。

第二类复位管脚。单片机上电后需要外部电路给它一个瞬间高电平或者低电平的复位信号，才能启动工作。复位电路通常是由电容和电阻组成的充电电路来实现，也有一些系统是用专门的复位芯片来实现。

第三类晶振管脚。任何单片机想要工作必须要有晶振。单片机执行程序指令是按一个节拍一个节拍来执行的。这个节拍的时间就是由晶振产生，所以把晶振比喻成单片机的“心脏”是非常恰当的。现在也有很多单片机直接把晶振集成到内部，这类单片机不用外接晶振也可以。

第四类 I/O 口管脚。这是跟我们编写程序关联最密切的管脚。前面提到的电源，复位，晶振这 3 种管脚是为了让单片机能工作，俗称单片机工作的三要素。而单片机工作的具体内容就是通过 I/O 口管脚来体现的。比如，I/O 口能识别按键的输入，也能输出驱动继电器工作的开关信号，也能跟外围器件进行通信。

### 【3.5 电平。】

电平就是电压的两种状态，低或者高，低相当于程序里的 0，高相当于程序里的 1。单片机 I/O 口管脚检

测到的电压低于或等于 0.8V 时是低电平，程序里读取到的是 0 数字。单片机 I/O 口管脚检测到的电压高于或等于 2.4V 时是高电平，程序里读取到的是 1 数字。必须注意的是，I/O 口输入的最大电压不能超过单片机的供电电压。单片机输出的低电平是 0V，单片机输出的高电平等于它的供电电压值，往往是 3.3V 或者 5V。

### 【3.6 时间。】

时间是单片机程序必不可少的元素。跟外围芯片通信的时序节拍需要时间，驱动发光二极管闪烁需要时间，工控自动化的某些延时需要时间。单片机的时间来源自两方面。第一方面源自指令的周期时间。单片机是根据节拍来执行程序指令的，每执行一条指令都要消耗一点时间，只要让程序执行的指令数量越多，产生的时间就越长，通过调整所执行指令的数量就可以得到所需的时间长度。第二方面源自单片机内部自带的定时器。假如设置定时器每 20 毫秒产生一次中断，现在要获取 10 秒钟的时间，只需在程序里统计 500 次定时中断就可以了，因为 1 秒等于 1000 毫秒。指令和定时器这两者的时间最后都来源于晶振。

#### 第四节：平台软件和编译器软件的简介。

##### 【4.1 平台软件和编译器软件的各自分工。】

C 语言写在哪？谁来把 C 语言翻译成单片机可以识别的机器语言？这就是平台软件和编译器软件的分工。平台软件负责编辑 C 语言，编译软件负责把 C 语言翻译成单片机可以识别的机器语言。

##### 【4.2 每一种单片机的平台软件和编译器软件不一定是唯一的。】

C 语言在单片机的应用也是最近这些年发展起来的，早期做单片机的原厂更关注芯片硬件本身，配套的 C 语言开发软件方面涉入不深，他们往往只管把单片机芯片生产出来后，给大伙提供一个汇编语言的编译器软件就草率了事，所以给了很多第三方商家做平台软件和 C 编译器的机会，后来单片机原厂也乐意支持和配合这些第三方开发软件的厂商，也有一些单片机原厂直接收购这类第三方软件公司。因此，不同厂家的单片机，它所用的平台和编译器软件可能都不一样。即使是同样一个厂家的单片机，它也有可能存在多种不同的第三方平台软件和编译器软件，每一种单片机所用的平台软件和编译器不一定是唯一的。比如 stm8 单片机可以用 STVD 软件平台，也可以用 IAR 平台。stm32 单片机可以用 keil 平台，也可以用 IAR 平台。

##### 【4.3 平台软件和编译器软件的宿主与寄生关系。】

平台软件选定了之后，所用的编译器软件也可能存在多种选择，并不是一种平台软件就绑定一种编译器软件。生物学的比喻，平台软件是宿主，编译器软件是寄生在平台软件里的。一个平台软件可以嵌入多种不同的编译器软件，平台软件和编译器软件存在一对多的关系。比如，PIC 单片机的平台软件是 MPLAB，8 位 PIC 单片机是 PICC 编译器，12 位 PIC 单片机是 PIC18 编译器，16 位 PIC 单片机是 C30 编译器。而且 MPLAB 平台软件与上述各种编译器软件都要单独一个一个分开来安装，最后运行 MPLAB 平台软件，在里面操作某个菜单设置选项，把各种 C 编译器软件跟 MPLAB 平台软件关联起来。

##### 【4.4 51 单片机的平台软件和编译器软件。】

我后面的讲解，51 单片机的平台软件用 keil2，编译器软件用 C51。单片机程序开发需要用到这两种软件，但在项目开发的时候，只要跟平台软件打交道就可以了，因为编译器软件是当做一种独立配件嵌入到平台软件里，统一接受平台软件的控制。我在用 PIC 的 8 位单片机时，需要安装一次 MPLAB 平台软件，也需要独立再安装一次 PICC 编译器软件，然后运行 MPLAB 平台软件，在里面操作某个菜单设置选项，把 PICC 编译器跟 MPLAB 平台软件关联起来，也就是我所说的把 PICC 编译器嵌入到 MPLAB 平台软件里，统一接受平台软件的控制，但我平常写代码时只需要跟 MPLAB 平台软件打交道就可以了。我早期在做 51 单片机开发时，也是需要把 keil 平台软件和 C51 软件分开安装，然后再把它们关联起来，但是现在从 keil2 版本开始，在安装 keil 平台软件时就已经默认把 C51 安装好了，并且自动把 C51 嵌入到了 keil 平台软件。我现在用 keil2 这个版本的平台软件，只需要安装一次 keil2 平台软件就可以了，不需要像早期那样再单独安装 C51 编译器。



## 第五节：用 keil2 软件关闭，新建，打开一个工程的操作流程。

### 【5.1 本教程选择 keil2 软件版本的原因。】

Keil 软件目前有 Keil2, Keil4, Keil5 等版本。本教程之所以选用 keil2 版本，是因为 keil2 版本比较单纯，它本身内置了 C51 编译器，并且只适用于 51 单片机不能适用于 stm32 这类单片机。而 Keil4, Keil5 等版本不仅可以适用于 51 单片机的，还可以适用于 ARM 类的单片机，它们有 C51 编译器和 MDK-ARM 编译器两种选择，在同一个 keil4 或者 keil5 版本里，C51 和 MDK-ARM 两者往往只能二选一，MDK-ARM 编译器是针对 stm32 这类单片机，如果你电脑上用了 MDK-ARM 编译器想再切换到 C51 编译器就很麻烦了往往不兼容，为了电脑上既能用 C51 编译器，又能兼容 MDK-ARM 编译器，我的电脑上是同时安装了 C51 编译器的 keil2 和 MDK-ARM 编译器的 keil4，一台电脑同时安装 keil2 和 keil4 不会冲突，能兼容的。

### 【5.2 如何在不用关闭 keil2 软件的前提下又能关闭当前被打开的工程？】

要关闭当前工程，最简单的方法是直接点击 keil2 软件右上角的“X”直接把 keil2 软件也一起关了，这种方法不在讨论范围，现在要介绍的是如何在不关闭 keil2 软件的前提下又能关闭当前被打开的工程。



图 5.2.1 启动 keil2 软件

第一步：启动 keil2 软件。

双击桌面” keil uVision2” 的图标启动 keil2 软件。

-----步骤之间的分割线-----

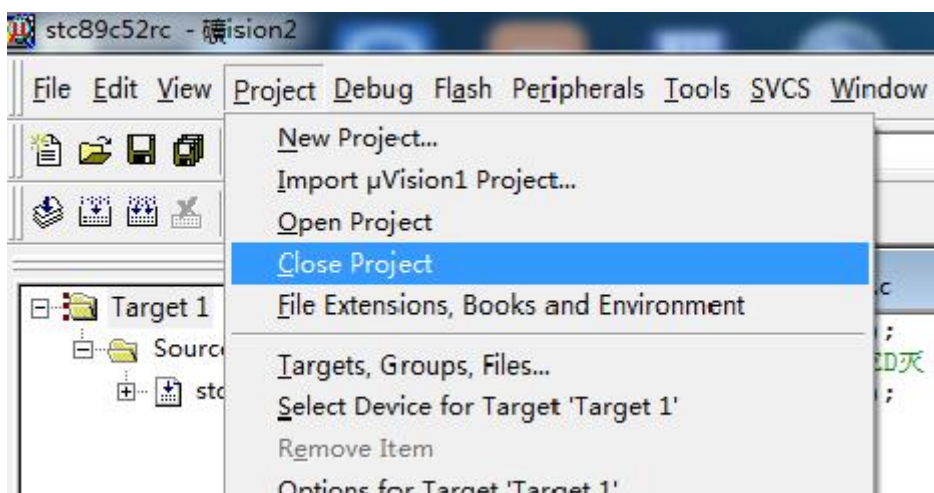


图 5.2.2 关闭被打开的已有工程

第二步：关闭被打开的已有工程。

启动 keil2 软件后，假设发现此软件默认打开了一个之前已经存在的工程。关闭已有工程的操作是这样子的：点击上面”Project”选项，在弹出的下拉菜单中选择”Close Project”即可。这时 keil2 软件处于”空”的状态，没有打开任何工程了。

### 【5.3 keil2 如何新建一个工程？】

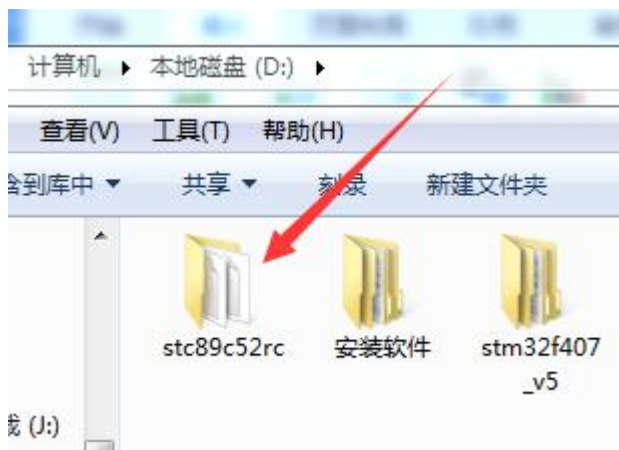


图 5.3.1 新建一个工程文件夹

第一步：新建一个工程文件夹。

在电脑 D 盘目录下新建一个文件夹，取名为”stc89c52rc”。

补充说明：

(1) 文件夹的命名以及后面涉及到的工程文件名统统都不要用中文，请全部用英文，数字，或者下划线这些字符。即使 keil 软件支持中文名，我建议也不要中文名，因为在单片机这个行业，有一些单片机厂家提供的平台软件，某些版本是不支持中文名的，所以大家从一开始就养成这个习惯，以后可以避免遇到一些不必要的麻烦。

(2) 新建的文件夹请直接放在某盘的根目录下，而不要放到某个已有文件夹的目录下。一方面是因为已

有的文件名目录往往带有中文单词，另外一方面是有一些单片机厂家的平台软件不支持嵌入层次太深的文件目录，所以大家从一开始就养成这个习惯，以后可以避免遇到一些不必要的麻烦。

-----步骤之间的分割线-----

第二步：启动 keil2 软件。

双击桌面” keil uVision2” 的图标启动 keil2 软件。

-----步骤之间的分割线-----

第三步：关闭默认被打开的已有工程。

启动 keil2 软件后，如果发现此软件默认打开了一个之前已经存在的工程，请先关闭此工程让 keil2 软件处于“空”的状态，如果没有发现此软件默认打开已有工程，这一步可以忽略跳过。关闭已有工程的操作是这样子的：点击上面” Project” 选项，在弹出的下拉菜单中选择” Close Project” 即可。这时 keil2 软件处于“空”的状态，没有打开任何工程了。

-----步骤之间的分割线-----

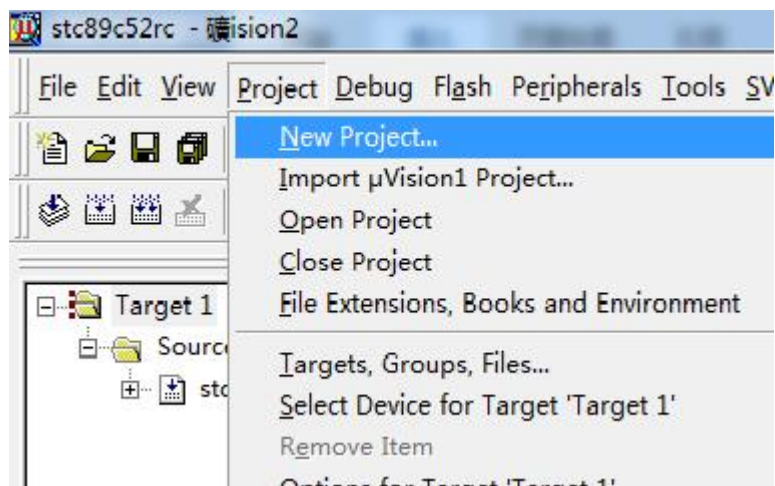


图 5.3.4.1 新建一个工程

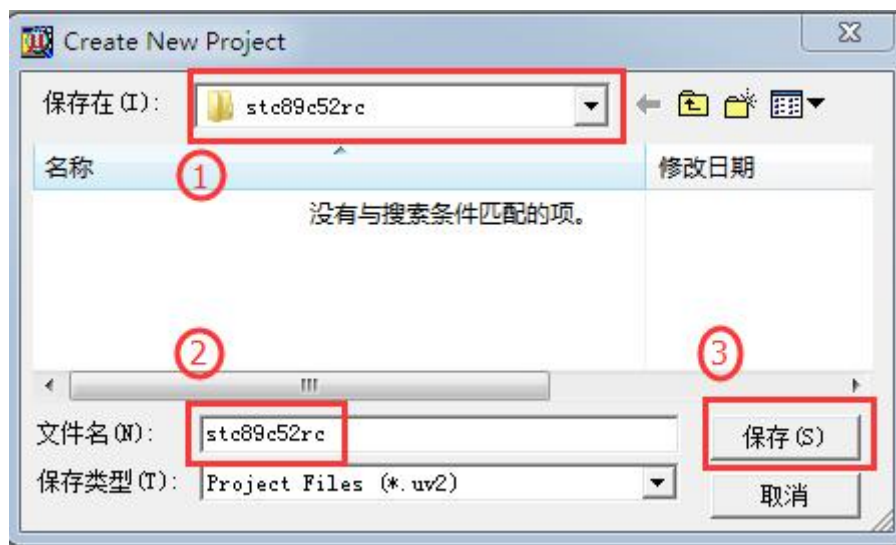


图 5.3.4.2 选择新建工程保存的位置

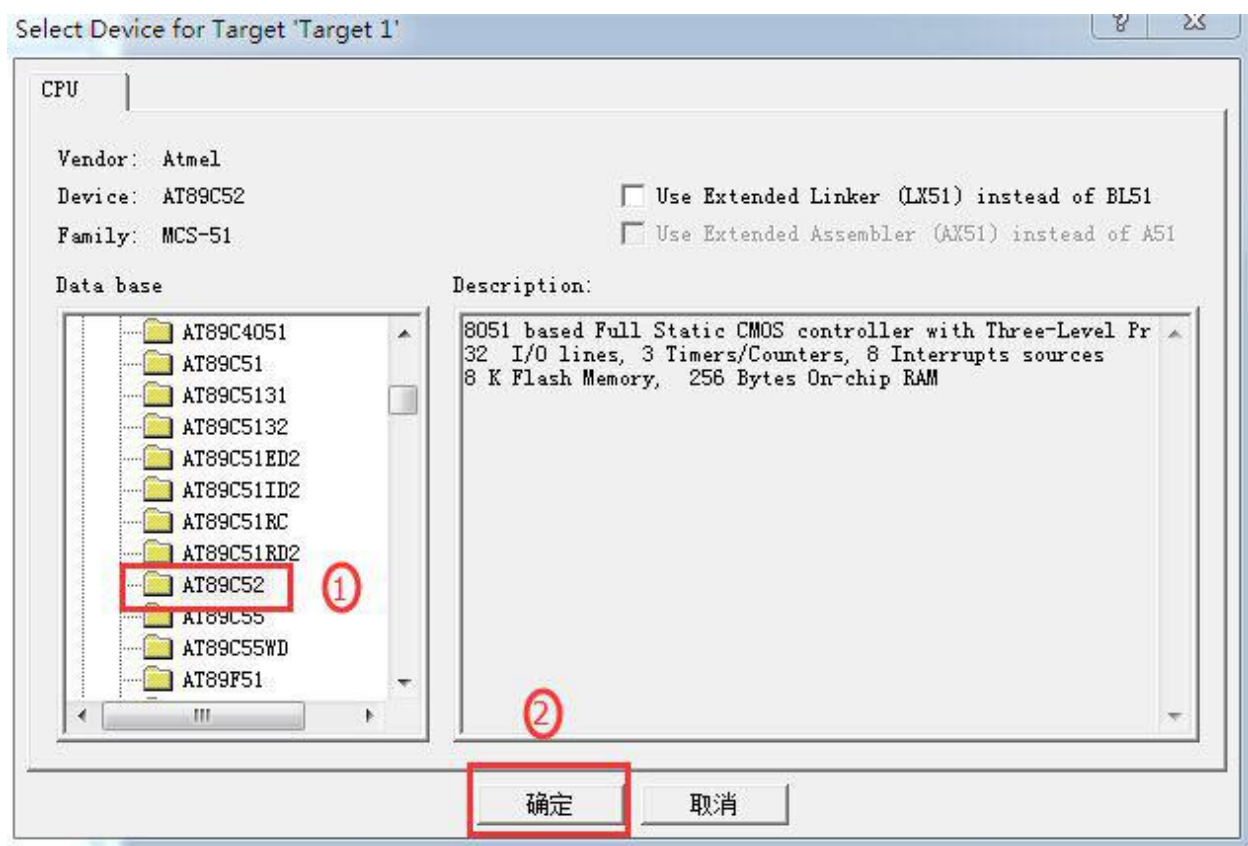


图 5.3.4.3 为当前工程选择编译器所支持的单片机型号

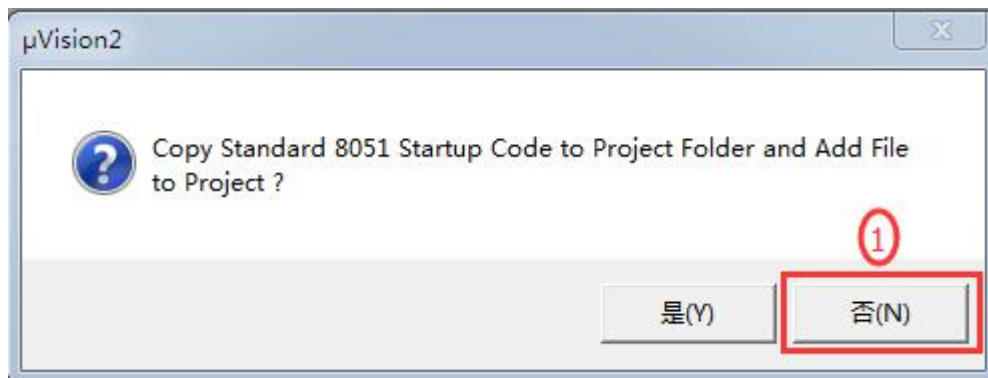


图 5.3.4.4 不需要把默认启动文件添加进来

第四步：利用工具向导新建一个工程。

点击上面”Project”选项，在弹出的下拉菜单中选择”New Project...”，在弹出的对话框中，选择保存的目录是刚才第一步新建的文件夹”stc89c52rc”，同时输入跟文件夹名称一样的工程文件名”stc89c52rc”，然后单击”保存”按钮（一个新工程模板就建成了），单击”保存”按钮后此时会弹出一个选择单片机型号的对话框，单击”Atmel”这个厂家前面的”+”号，在展开的下拉选项中选中”AT89C52”这个型号，然后点击”确定”，此时会弹出一个英文询问框，大概意思是”是否要复制 STARTUP.A51 这个文件到工程里？”我们单击”否”即可。

补充说明：

（1）以上新建的保存文件名应该跟我们第一步在 D 盘新建的文件夹名称一致，确保都是”stc89c52rc”，因为有一些单片机厂家的平台软件是有这个要求的，所以大家养成这个习惯，以后可以避免遇到一些不必要的麻烦。

（2）上面之所以选择 Atmel 厂家的 AT89C52 单片机，是因为本教程选用的单片机 STC89C52RC 跟 AT89C52 是兼容的。

（3）在弹出的英文询问框，大致意思是”是否要复制 STARTUP.A51 这个文件到工程里？”，那么 STARTUP.A51 这个文件有什么含义？STARTUP.A51 是一个启动程序文件，在单片机进入.c 程序执行 main 函数之前，先去执行这个启动程序，这个启动程序是专门用来初始化 RAM 和设置堆栈等，如果我们选”否”不添加这个启动程序，编译器也会自动加入一段我们不能更改的默认启动程序。如果选”是”，那么这个文件就会出现我们工程里，我们可以根据需要对它进行更改。但是大多数的情况下，我们都不会去更改此文件，所以无论你选”是”还是”否”，只要你不更改 START.A51 文件，对我们来说都是一样的。因此我本人一般情况下都是选”否”。

-----步骤之间的分割线-----

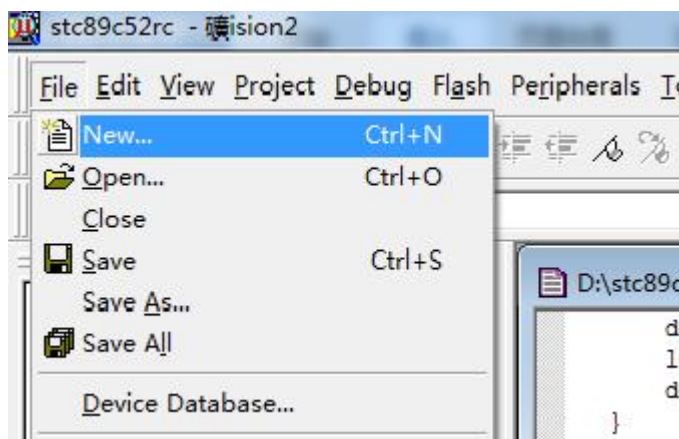


图 5.3.5.1 新建一个源文件

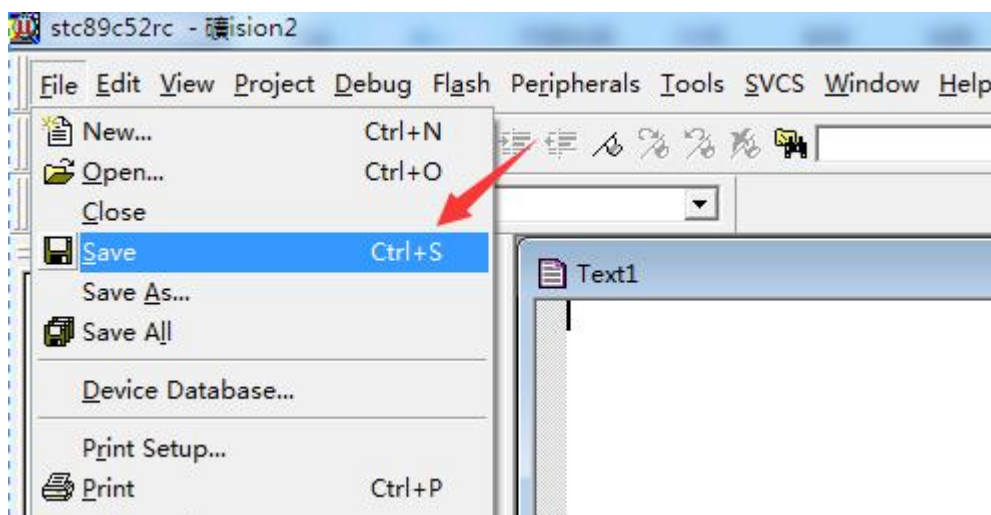


图 5.3.5.2 保存当前新建的源文件

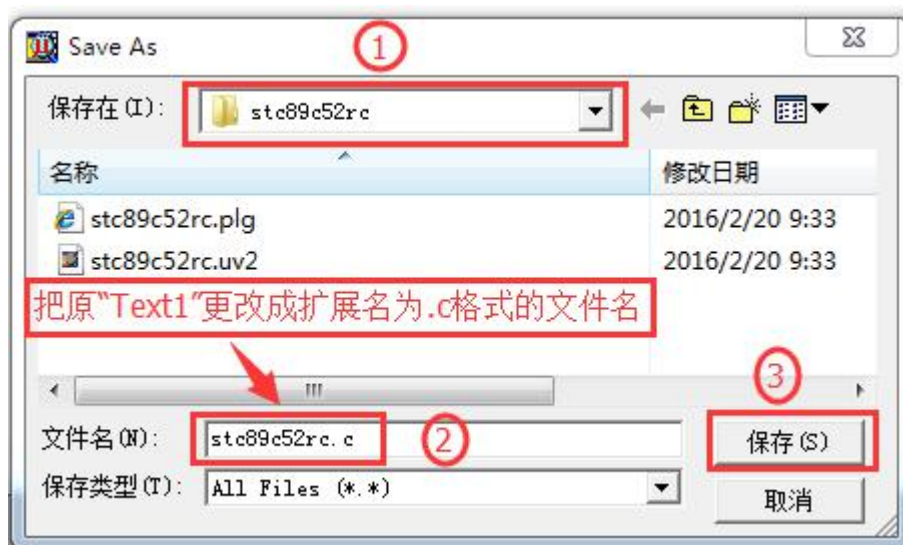


图 5.3.5.3 把当前源文件保存在指定的位置



第五步：新建一个.c 源文件。

点击上面”File”选项，在弹出的下拉菜单中选择”New...”，会看到弹出来一个名字为”Text1”的文件。再一次点击上面”File”选项，在弹出的下拉菜单中选择”Save”，会弹出一个保存的对话框，此时还是选择保存在第一步新建的文件夹目录下，并且把”Text1”文件名更改为”stc89c52rc.c”（注意后缀是.c 扩展名），单击”保存”。

补充说明：

（1）此时你如果打开 D 目录下”stc89c52rc”的文件夹，你会发现此文件夹有一个”stc89c52rc.c”的文件，这个文件就是在这一步被新建添加进来的，但是此文件”stc89c52rc.c”目前跟整个工程还没有关联，还需要在接下来的第六步那里进行关联操作。

（2）上面新建添加的文件，它的文件名必须是带.c 这个扩展名，表示此文件是 C 文件格式，这一个很重要不要搞错了。往后我们所写的 C 语言程序代码就是写在此 C 格式的文件里。此文件也俗称 C 源文件。

-----步骤之间的分割线-----

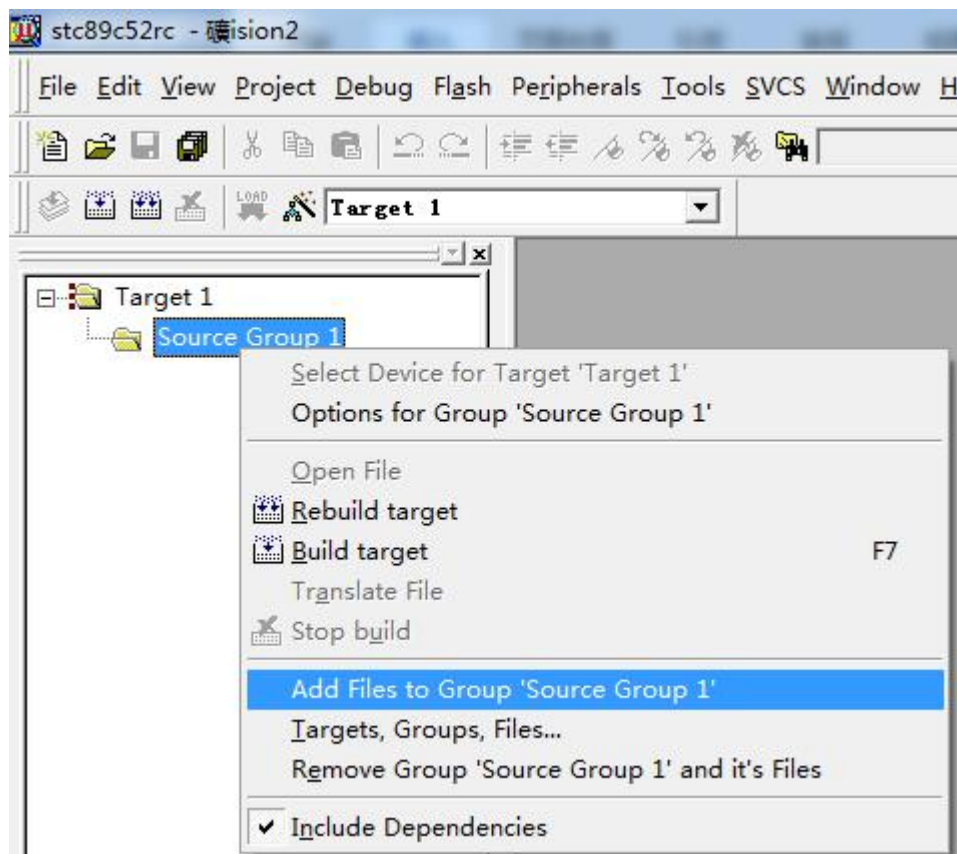


图 5.3.6.1 即将把源文件添加进工程里

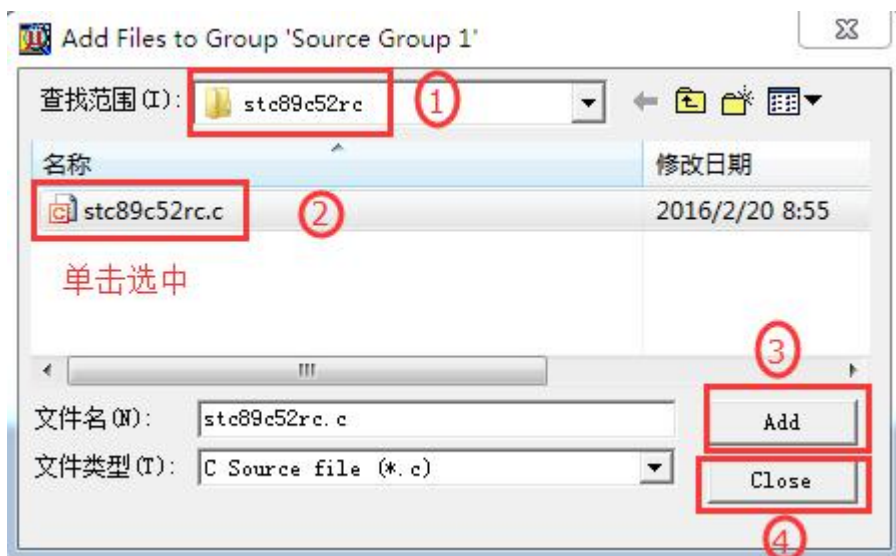


图 5.3.6.2 选择需要添加进工程里的源文件

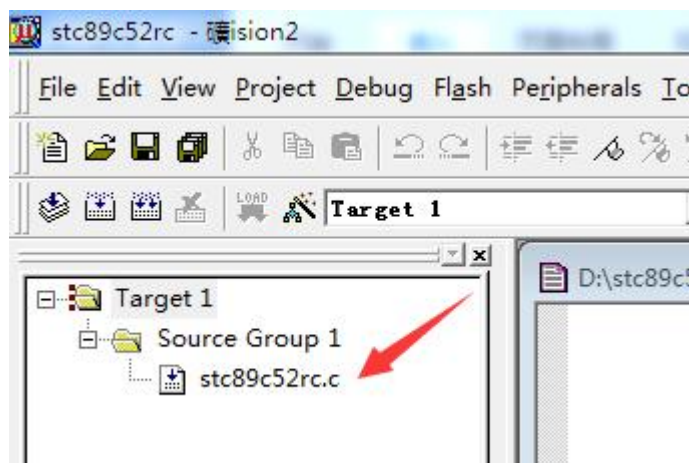


图 5.3.6.3 源文件添加成功

第六步：把刚才新建的.c源文件添加到工程里，跟当前工程关联起来。

点击左边竖着的选项框里面的”Target 1”前面的”+”号，在展开的下拉菜单下看到”Source Group 1”。右键单击”Source Group 1”选项，在下拉菜单中选择”Add Files to Group ‘Source Group 1’”选项，弹出一个文件选择对话框，单击选中刚才新建的.c源文件，然后单击一次”Add”按钮，此时虽然对话框没有关闭，但是已经悄悄地把.c源文件添加到工程里了（这个地方 keil 的用户体验设计得不够好，容易让人误解还没有把文件添加进来），这时再点击一次”Close”按钮先把此对话框关闭，然后发现左边的”Source Group 1”前面多了一个”+”号，单击此”+”号展开，发现下面的文件恰好是刚才新添加进去的.c源文件”stc89c52rc.c”。

补充说明：

（1）在刚才的操作中，我本人觉得 keil 软件有一个地方的用户体验做得不够好，容易引起误解。就是在弹出一个文件选择对话框时，先单击选中刚才新建的.c源文件，此时单击一次”Add”按钮，已经相当于把.c文件添加进工程了，但是此时 keil 软件并没有自动关闭对话框，这样很容易让初学者误以为.c源文件



还没有被添加进去。

-----步骤之间的分割线-----

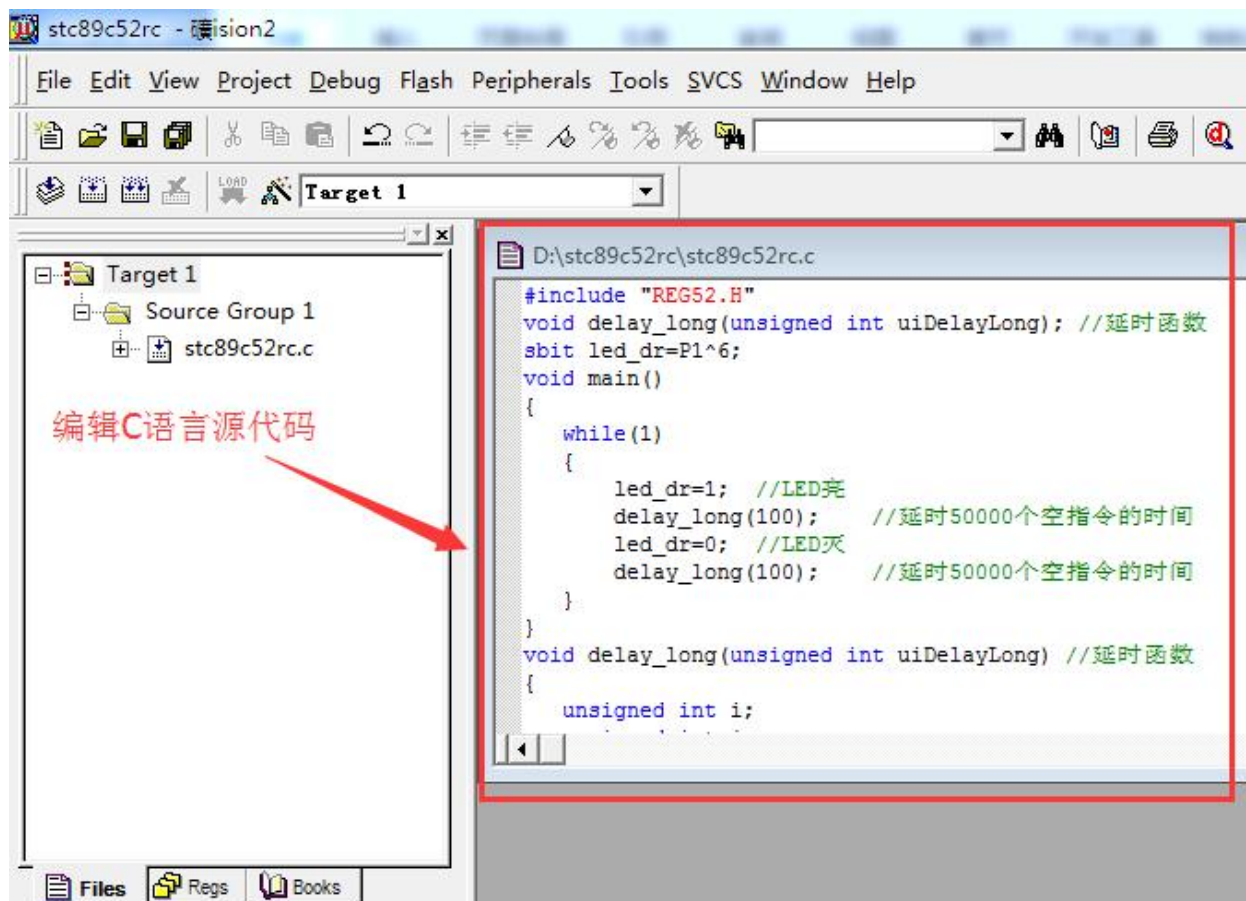


图 5.3.7 编辑 C 语言代码的区域

第七步：至此，可以正常的编辑 C 语言代码了。

双击打开左边 Target1 里面 Source Group1 下刚刚被添加进工程的“stc89c52rc.c”源文件，就可以在此“stc89c52rc.c”文件下输入 C 语言代码了，请把以下范例代码复制进去，然后再一次点击”File”选项，在弹出的下拉菜单中选择“Save”保存。此时，新建一个工程的步骤已经完成。供复制的范例代码如下：

```
#include "REG52.H"
void delay_long(unsigned int uiDelayLong); //延时函数
sbit led_dr=P1^6;
void main()
{
    while(1)
    {
        led_dr=1; //LED 亮
        delay_long(100); //延时 50000 个空指令的时间
        led_dr=0; //LED 灭
        delay_long(100); //延时 50000 个空指令的时间
```

```

    }
}
void delay_long(unsigned int uiDelayLong) //延时函数
{
    unsigned int i;
    unsigned int j;
    for(i=0;i<uiDelayLong;i++)
    {
        for(j=0;j<500;j++); //内嵌循环的空指令数量
    }
}

```

-----此处为分割线，上面的是代码的结束，下面的是补充说明的开始-----

补充说明：

(1) 可能有些朋友不是用 keil2 版本，如果他们是使用 keil4 的版本，当把代码复制到 keil4 时，如果中文注释出现乱码怎么办？解决办法是这样的：点击 keil4 软件的左上角“Edit”，在下拉菜单中选最后一项“Configuration”，在弹出的对话框中把 Encoding 的选项改成“Chinese GB2312(Simplified)”。然后删除所有 C 代码，重新复制一次代码进去就恢复正常了。当然，我们用 keil2 版本不会遇到这个问题，况且 keil2 版本的“Edit”下拉菜单也没有“Configuration”这个选项，所以 keil2 和 keil4 还是有一些差别的。

## 【5.4 keil2 如何打开一个现有的工程？】

第一步：启动 keil2 软件。

双击桌面“keil uVision2”的图标启动 keil2 软件。

-----步骤之间的分割线-----

第二步：关闭默认被打开的已有工程。

启动 keil2 软件后，如果发现此软件默认打开了一个之前已经存在的工程，请先关闭此工程让 keil2 软件处于“空”的状态，如果没有发现此软件默认打开已有工程，这一步可以忽略跳过。关闭已有工程的操作是这样子的：点击上面“Project”选项，在弹出的下拉菜单中选择“Close Project”即可。这时 keil2 软件处于“空”的状态，没有打开任何工程了。

-----步骤之间的分割线-----

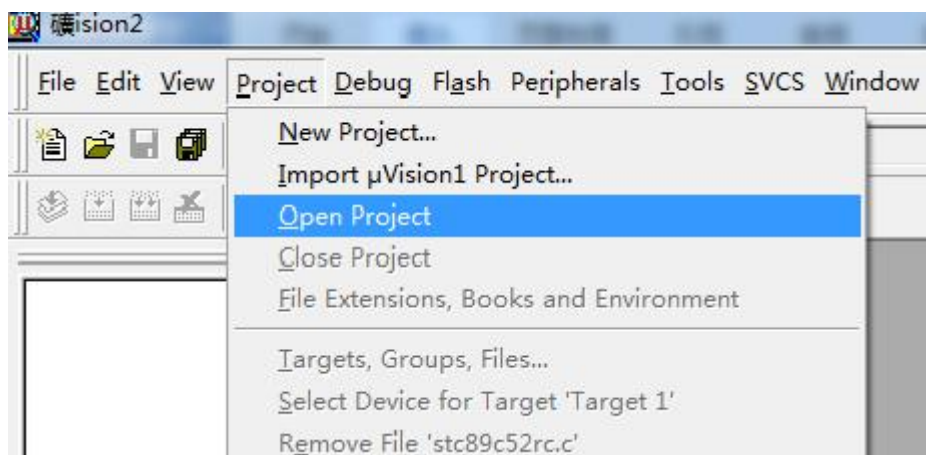


图 5.4.3.1 打开一个现有的工程

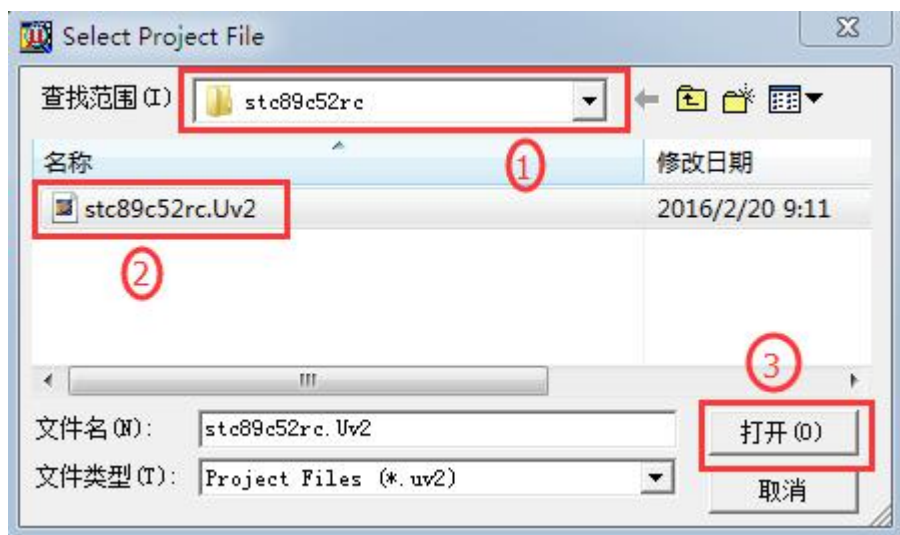


图 5.4.3.2 选择将要被打开的工程

第三步：打开一个现成的工程。

点击上面”Project”选项，在弹出的下拉菜单中选择“Open Project”，在弹出的文件对话框中，找到需要被打开工程文件夹（本例程是 D 盘下的“stc89c52rc”文件夹），在此文件夹目录下单击选中“stc89c52rc.Uv2”这个工程文件名，然后点击“打开”，就可以打开一个现有的工程文件了。

## 第六节：把.c 源代码编译成.hex 机器码的操作流程。

### 【6.1 把.c 源代码编译成.hex 机器码的详细步骤。】

第一步：启动 keil2 软件。

双击桌面” keil uVision2” 的图标启动 keil2 软件。

-----步骤之间的分割线-----

第二步：关闭默认被打开的已有工程。

启动 keil2 软件后，如果发现此软件默认打开了一个之前已经存在的工程，请先关闭此工程让 keil2 软件处于“空”的状态，如果没有发现此软件默认打开已有工程，这一步可以忽略跳过。关闭已有工程的操作是这样子的：点击上面” Project” 选项，在弹出的下拉菜单中选择” Close Project” 即可。这时 keil2 软件处于“空”的状态，没有打开任何工程了。

-----步骤之间的分割线-----

第三步：打开一个现成的工程。

点击上面” Project” 选项，在弹出的下拉菜单中选择” Open Project”，在弹出的文件对话框中，找到需要被打开工程文件夹（本例程是上一节在 D 盘下建的” stc89c52rc” 文件夹），在此文件夹目录下单击选中” stc89c52rc.Uv2” 这个工程文件名，然后点击” 打开”，就可以打开一个现有的工程文件了。

-----步骤之间的分割线-----

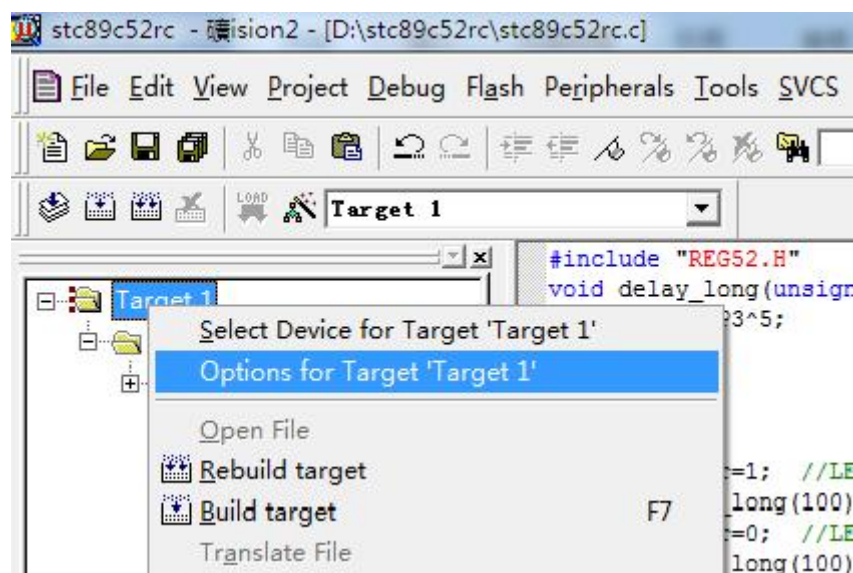


图 6.1.4.1 把设置窗口调出来

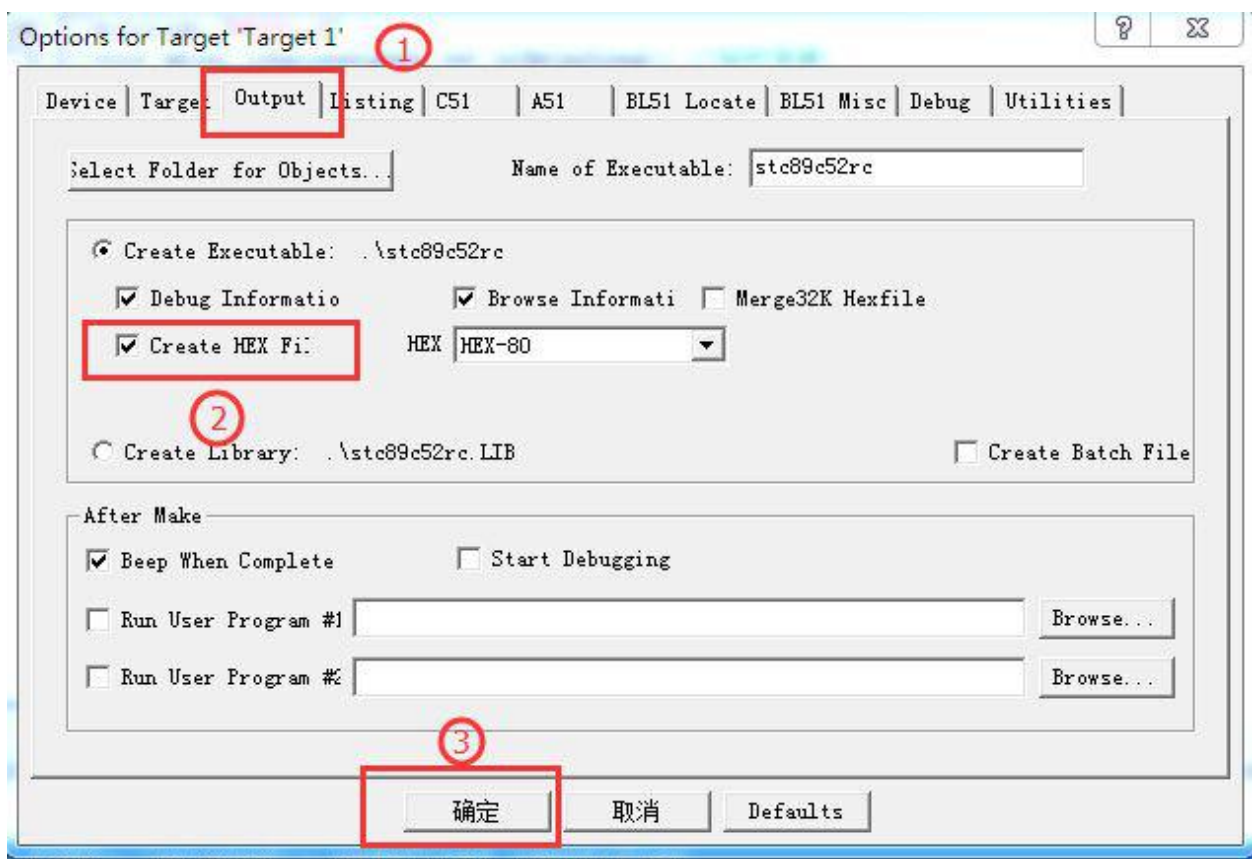


图 6.1.4.2 勾选上能生成 HEX 机器码的选项

第四步：设置编译环境让 keil2 软件允许产生 .hex 格式的机器码文件。

鼠标右键点击选中左边选项框里面的“Target 1”选项，在右键下拉菜单中选择“Options for Target ‘Target 1’”选项，弹出一个编译环境设置对话框，左键单击上面子菜单切换到“Output”窗口下，把“Create Hex File”勾选上。点击“确定”。

补充说明：

(1) 这个选项很重要，必须把“Create Hex File”选项勾上，否则后续的操作不能在工程文件夹的目录里生成 .Hex 的机器码文件。对于一个文件夹的工程模板，只需要设置一次就可以保存起来了，下次开电脑重新打开此工程模板时不需要再设置，这些被设置的参数都是能掉电保存起来的。

-----步骤之间的分割线-----

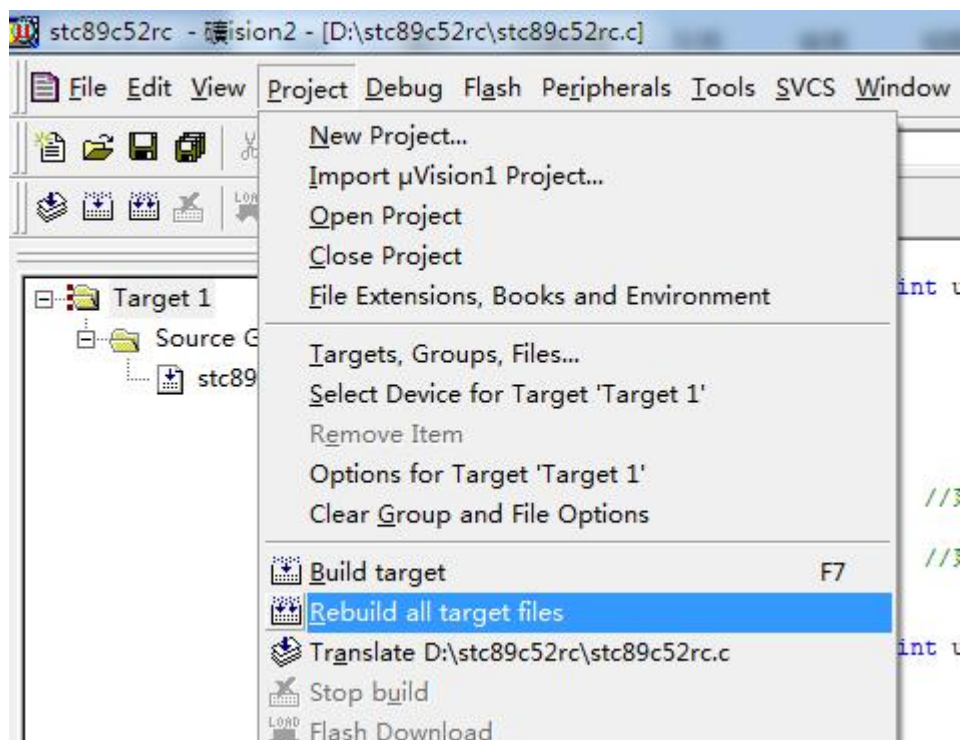


图 6.1.5.1 重新编译所有的文件

第五步：启动编译。

在确保 stc89c52rc.c 源文件里面有 C 语言源代码的情况下(如果没有, 请先复制上一节的例程源代码), 点击上面” Project”选项, 在弹出的下拉菜单中点击” Rebuild all target files”编译命令, 编译器开始编译工作。

-----步骤之间的分割线-----



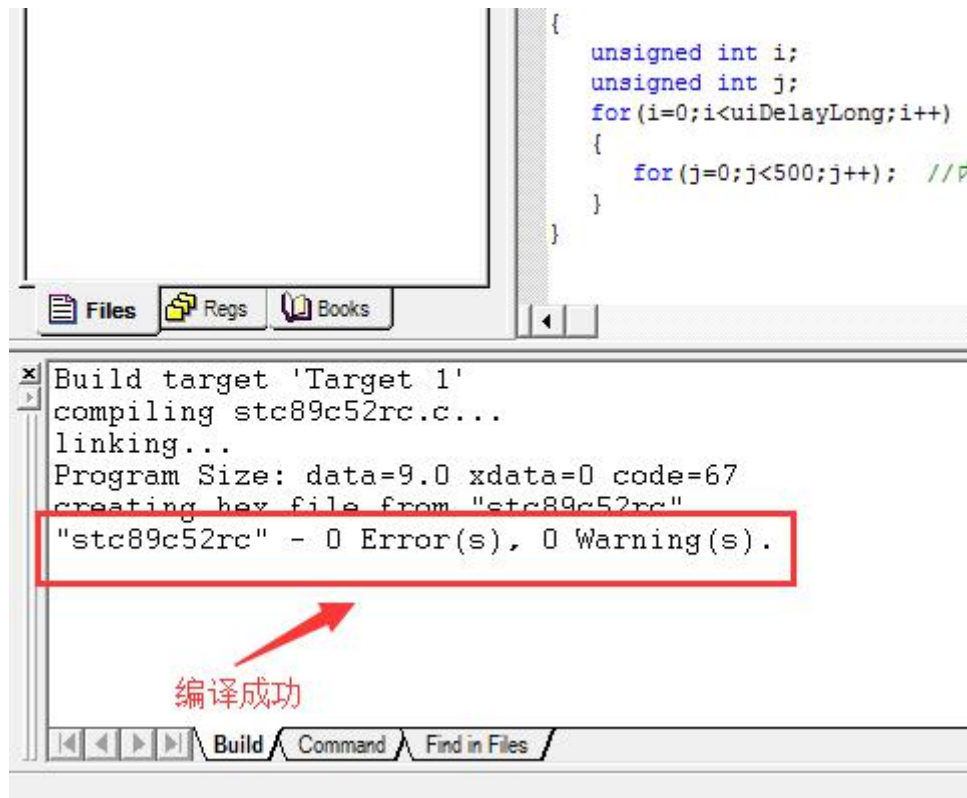


图 6.1.6.1 编译成功

第六步：在”Output Window”窗口下观察编译结果。

可以在最下方的”Output Window”窗口下观察到编译的过程提示。”Output Window”窗口默认出现在源代码区的最下方，如果没有发现”Output Window”窗口，请把鼠标的光标移动到最下方的滑动条下边，当它呈现移动光标的形状时，按住左键往上拖动就可以看到“Output Window”窗口了。当“Output Window”窗口最后一行显示“”stc89c52rc” - 0 Error(s), 0 Warning(s).”等信息时，表示翻译工程结束了。其中 0 Error(s)代表编译成功，没有任何错误。0 Warning(s)代表没有任何警告。

补充说明：

(1) 只要有一个错误 Error 产生，就说明编译不通过。如果没有任何错误 Error 产生，但是有几个警告 Warning 产生，在这种情况下很多时候都不影响程序的正常运行，只有少数情况下是会影响代码的正常运行的，因此我本人建议哪怕是一个警告，大家也不要放过它，也要找到产生这个警告的原因。

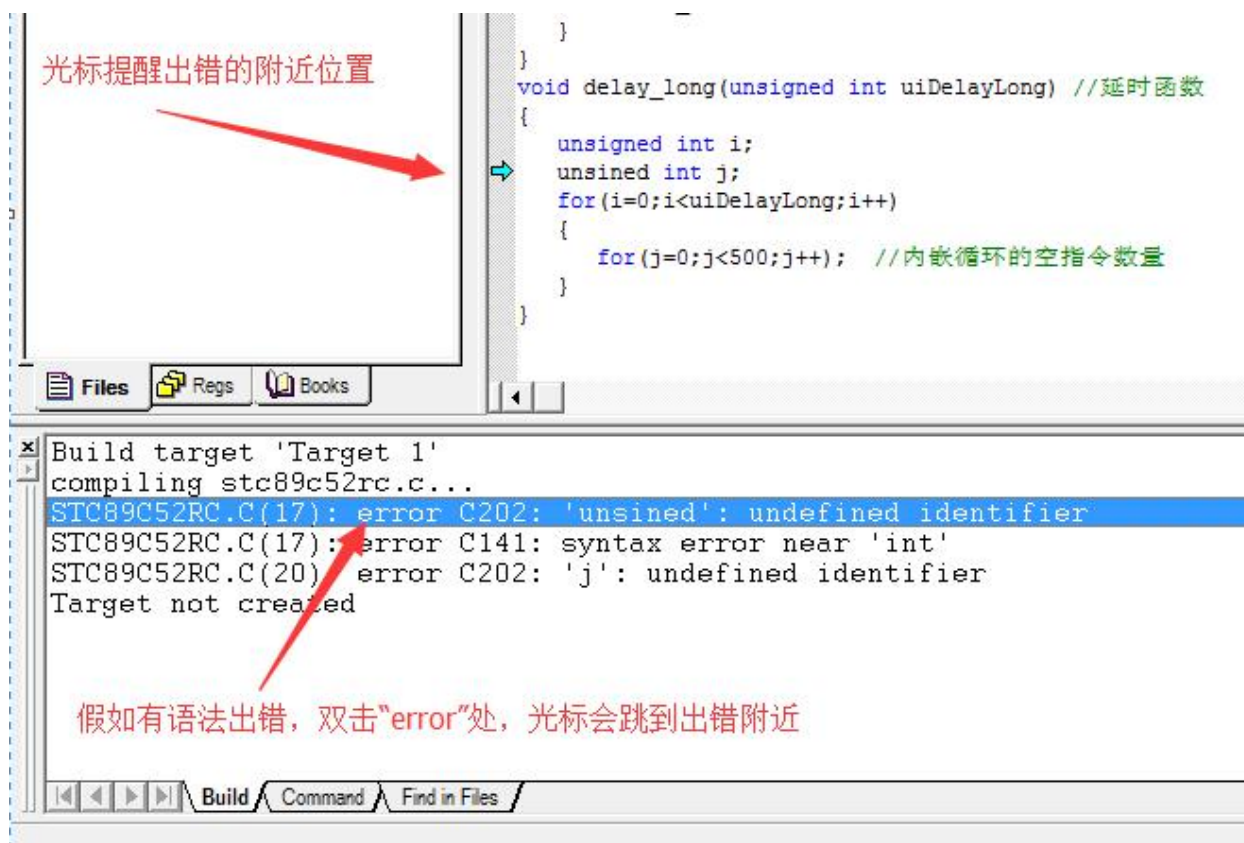


图 6.1.6.2 双击出错提示那行自动跳转到错误附近

(2) 查找错误的时候，只需要双击错误提示 error 那行内容，光标就会自动跳到源代码错误的附近，方便大家寻找语法错误。

(3) 还有一种很实用的方法，就是直接把提醒出错那一整行英文复制粘贴到网上去搜索，往往能搜索到所需的答案或者重要提示。

-----步骤之间的分割线-----



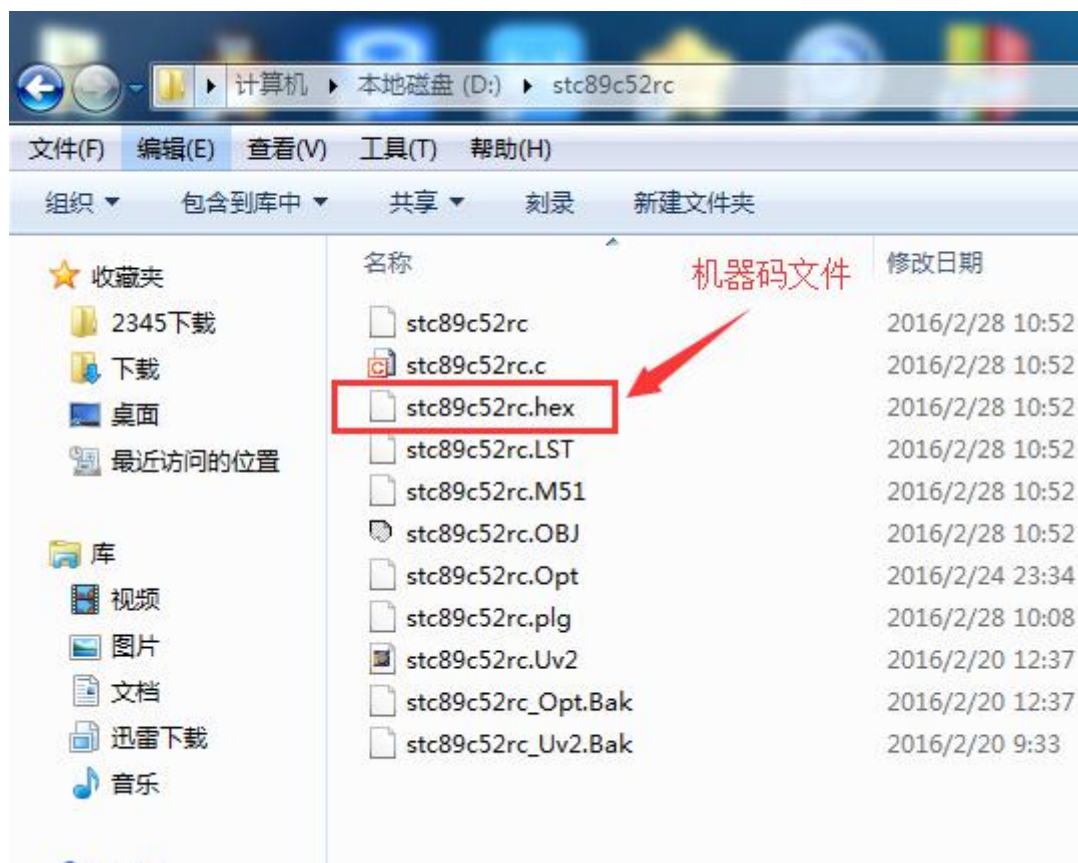


图 6.1.7.1 这个就是我们需要的 HEX 机器码烧录文件

第七步：编译后生成.hex 机器码文件的目录位置。

以上编译成功后，只要打开电脑 D 盘的 stc89c52rc 文件夹，就可以找到.hex 扩展名的机器码文件，这个文件就是我们要下载到单片机的机器码文件。

## 【6.2 注意！最后，还有一个非常重要的 keil 编译环境需要设置。】

STC89C52 单片机与 AT89C52 单片机是兼容的，它们的 ROM 程序容量都是 8K 字节，而它们的 RAM 数据容量是不一样的，STC89C52 的 RAM 是 512 字节，而 AT89C52 的 RAM 是 256 字节，尽管两者的 RAM 容量有一些小差异，但是对于我们用作入门学习来说，这些都是无所谓的，所以本教程硬件平台虽然是用 STC89C52 单片机，但是 keil 的编译环境其实是用 AT89C52 的芯片环境，因此本教程就以 AT89C52 为准。刚才提到 AT89C52 的程序容量 ROM 是 8K 字节，数据容量 RAM 是 256 字节，那么问题来了，很多初学者经常遇到，有一些程序代码 ROM 明明没有超过 8K，或者数据容量 RAM 明明还没超过 256 字节，编译器居然报错提醒容量不够！什么原因？怎么解决？

什么原因？是单片机的内存分配模式问题引起的，具体原因暂时不深入讲解。

怎么解决？有一个非常重要的 keil 编译环境需要设置一下，设置步骤是：

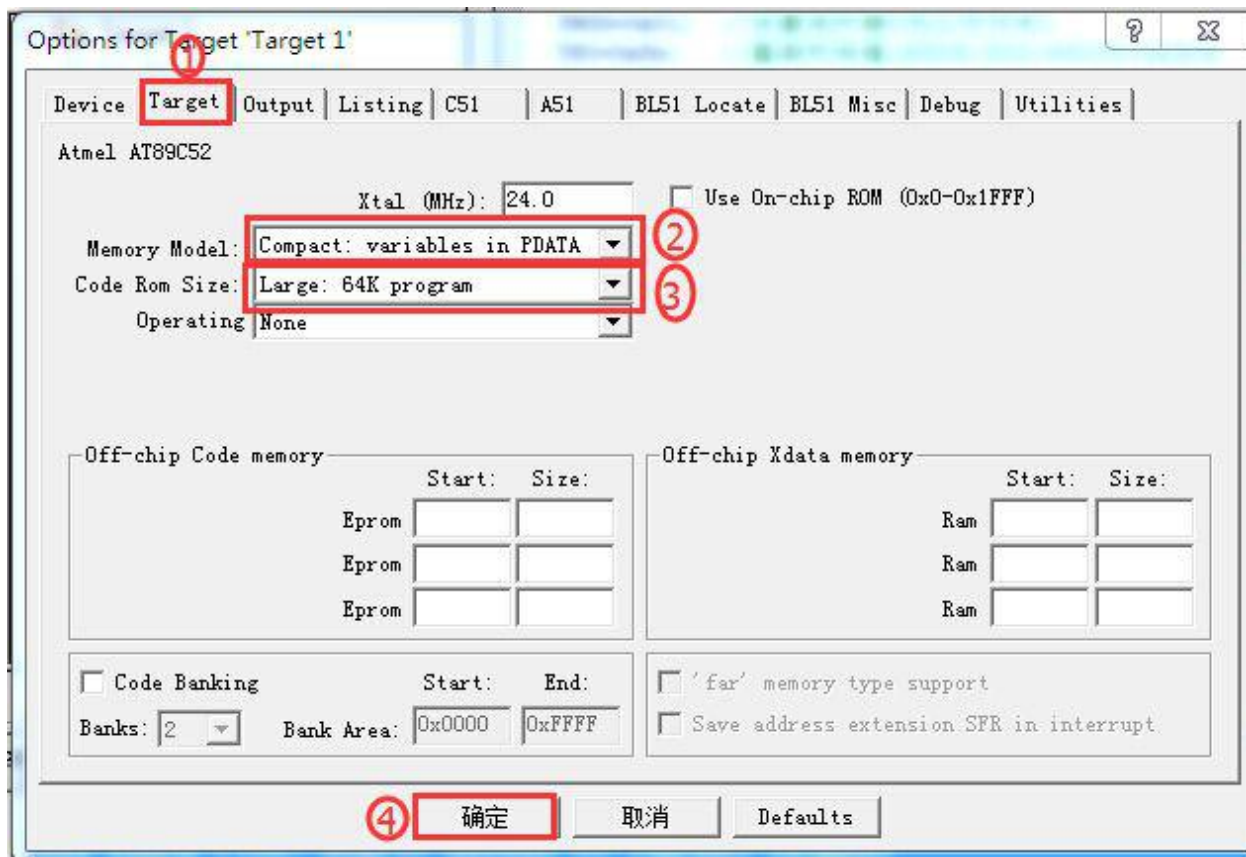


图 6.2.1 设置编译器的 RAM 和 ROM 容量大小的模式

如上图所示，在一个已经打开的工程里，鼠标右键点击选中左边选项框里面的”Target 1”选项，在右键下拉菜单里选择“Options for Target ‘Target 1’”选项，弹出一个编译环境设置对话框，第一步单击上面子菜单切换到“Target”窗口下，第二步在 Memory Model 选项的下拉菜单里选中“Compact: variables in PDATA”，第三步在 Code Rom Size 选项的下拉菜单里选中“Large: 64K program”，第四步点击“确定”。

## 第七节：本节预留。

本节预留。

## 第八节：把 .hex 机器码程序烧录到单片机的操作流程。

### 【8.1 烧录程序的本质。】

“烧录”是比较专业的说法，很多初学者第一次听这词还以为跟火有关，莫名其妙的“烧录”是啥意思？烧录其实就是下载，烧录程序就是下载程序。下载好理解了吧，下载电影，下载歌曲，让播放器去播放。此处的下载程序跟下载歌曲的“下载”完全是一回事。有人会问，下载歌曲到手机，手机是成品，下载程序到单片机，单片机也是成品？新买回来的单片机不是一张白纸的电子元件吗？其实，新买回来的单片机就是一个成品，它不是白纸，它内部已经嵌入了一段系统程序，这个系统程序就像你刚买回来的手机就帮你预装了安卓系统一样，只是它的用户存储区是空白的。比如手机，你往这个存储区里存电影就可以看电影，存音乐就可以听音乐。比如单片机，你往这个存储区存不同的程序就可以让单片机做不同的事。而预装在新单片机内部的系统程序就是专门负责跟外部接口通讯，同时负责把 hex 格式的程序代码存放在单片机内部正确的位置，这个就是烧录程序(下载程序)的本质。这样一比喻，所以 .hex 格式的烧录文件跟 .MP3 格式的音乐文件在存储本质上是是一样的。

再回顾总结一下，烧录程序的本质是：把单片机当做一个存储器，每一条程序指令都对应一个唯一的存储地址，把这些指令以字节为单位一条条存储到指定的存储地址中，这就是烧录程序的本质。对于 STC89C52RC 单片机，在下载程序时需要上位机软件和一根 USB 转串口线。上位机软件负责把 .hex 格式的机器码文件打开，机器码文件里面记录着每条程序指令所对应的地址信息，下载过程时，上位机软件根据 .hex 文件记录的指令内容和对应的地址信息，经过 USB 转串口线，跟单片机的预置系统程序进行串口通讯，从而把 .hex 记录的信息传输到单片机内部的 flash 存储器中，实现了程序的烧录下载。

### 【8.2 烧录程序所需的工具和软件。】

(1) 装有 XP 或者 WIN7 系统的电脑一台。

其它更高系统的我还没测试过，应该问题也不大。

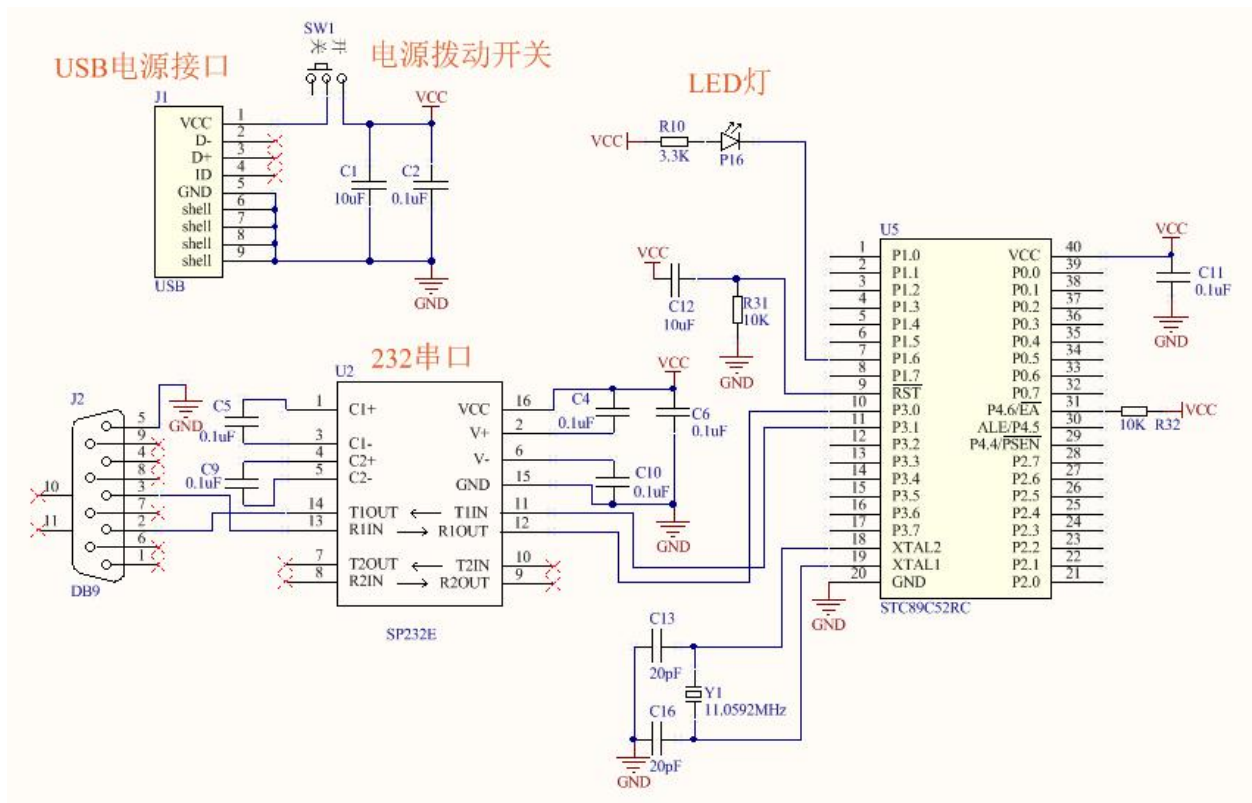


图 8.2.2 带串口的单片机最小系统

(2) 带 9 针串口、1 颗 LED 灯、电源拨动开关、能 5V 电源供电的 stc89c52rc 单片机核心板一块。

单片机的学习离不开硬件平台的编程练习，本教程用的学习板原理图已经分享到网上连载贴的附件资料里。大家也可以根据原理图自己焊接一块学习板来学习，或者用其它厂家带有串口的单片机学习板来学习。

串口是用来单片机跟电脑通讯的接口，是 STC89C52RC 单片机下载程序的通道。LED 灯用来观察单片机是否正常运行程序。电源拨动开关方便烧录程序时提供所需的断电和上电的操作。本单片机系统是 5V 供电。



图 8.2.3 USB 转串口线

(3) 主控芯片是 CH340 的 USB 转 RS232 串口线一条。

我之所以推荐主控芯片是 CH340 的 USB 转 RS232 串口线,因为 CH340 的下载线在烧录程序时很稳定可靠。这款 USB 转串口线可以在淘宝购买到。



图 8.2.4 USB 取电的电源线

(4) 5V 供电的 USB 电源线一条。

此 USB 线可以从电脑的 USB 口取电,也可以从输出 5V 的手机充电器处取电。但是我建议大家用输出 5V



的手机充电器来供电，因为很多电脑的 USB 口供电干扰比较大，会影响程序烧录。



图 8.2.5 USB 转串口的驱动安装程序

(5) 主控芯片是 CH340 的 USB 转 RS232 串口线驱动安装程序。

此驱动程序 USB 转 RS232 串口线的厂家通常都会提供，但是建议用我在附件资料里推荐给大家的驱动程序，毕竟这个程序经过我本人验证测试过。

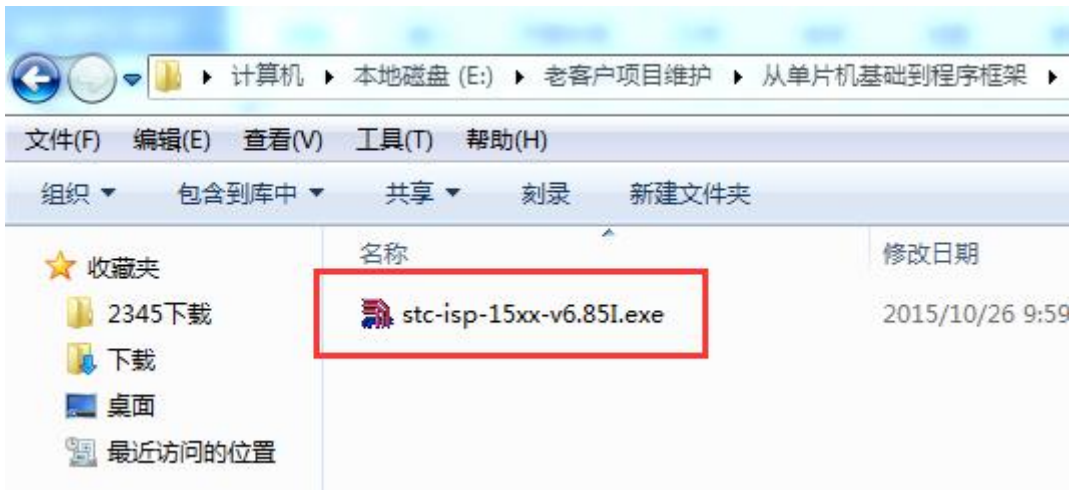


图 8.2.6 上位机软件

(6) 烧录程序和串口助手功能都具备的“stc-isp-15xx-v6.85I”上位机软件。

这是宏晶单片机官方免费提供的上位机软件，可以在宏晶单片机的官网上下下载获取。这款软件有很多功能，除了有下载程序和串口助手的功能外，还可以用来配置自动生成所需的初始化代码。当然，本教程后面主要是用到此软件的下载程序和串口助手这两个功能。所以大家所选的软件版本必须是 v6.85I 版本或者以上的版本，因为早些年有一些版本只有烧录功能但是没有串口助手的功能。

**【8.3 把 .hex 文件烧录到单片机的操作流程。】**

前面第 6 节内容已经教大家把一个驱动 LED 灯闪烁的 C 源代码编译成 .hex 文件的操作流程，同时在 D 盘的“stc89c52rc”文件夹里已经生成了一个“stc89c52rc.hex”的机器码文件，现在就要教大家如何烧录此文件到单片机内。此程序的功能是让单片机驱动一颗 LED 灯闪烁。

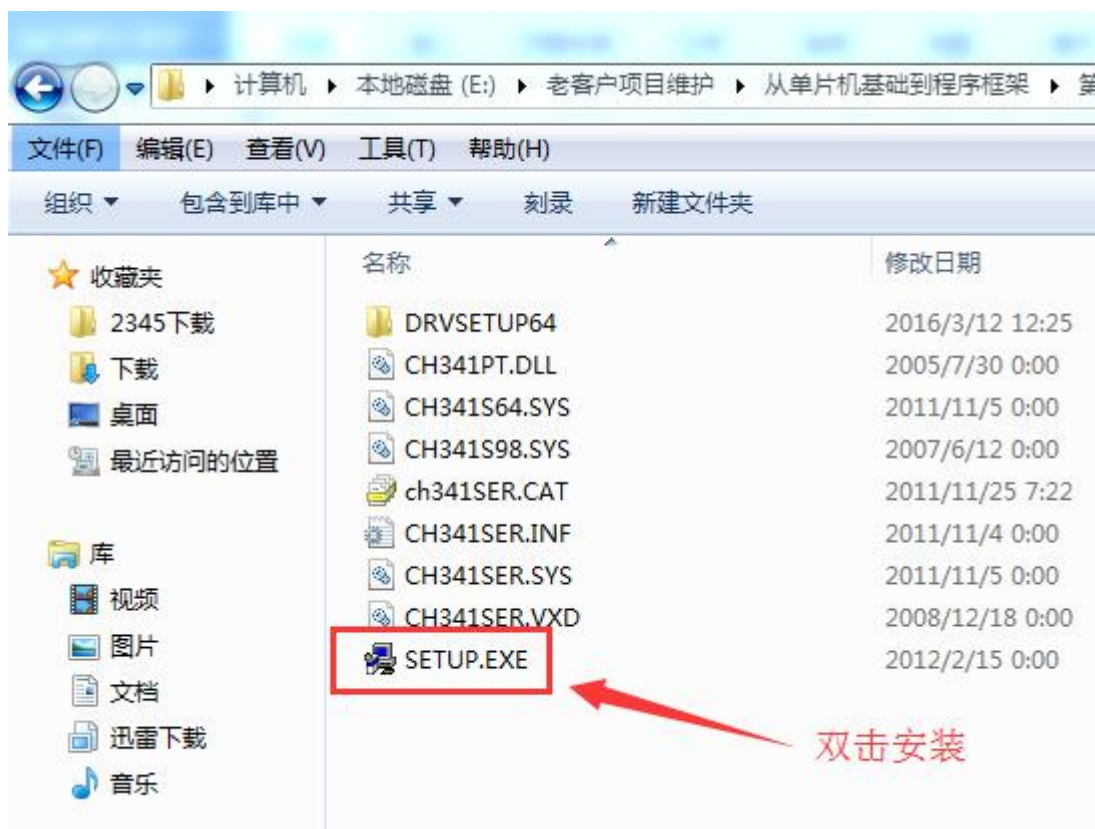


图 8.3.1.1 双击安装 USB 转串口驱动程序的启动图标



图 8.3.1.2 安装 USB 转串口驱动程序



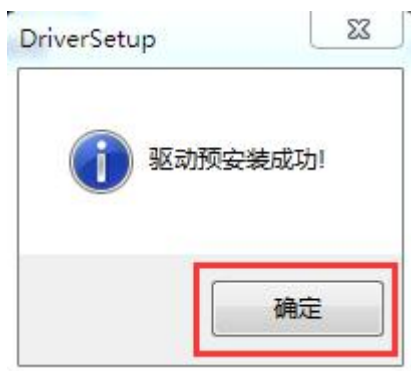


图 8.3.1.3 驱动程序安装成功

第一步：安装 USB 转串口驱动程序。

上位机“stc-isp-15xx-v6.85I”烧录软件就是安装在电脑端的用户软件，电脑跟单片机进行通讯，需要一根 USB 转串口线，欲使 USB 转串口线正常工作，必须预先安装 USB 转串口的驱动程序。具体的操作是这样的：在本连载贴附件资料处下载“USB 转串口的驱动程序 CH340.zip”文件压缩包，解压后打开此文件夹，找到“SETUP.EXE”这个安装应用程序，双击启动，在弹出的界面中，单击“安装”按钮即可完成驱动程序的安装。

-----步骤之间的分割线-----

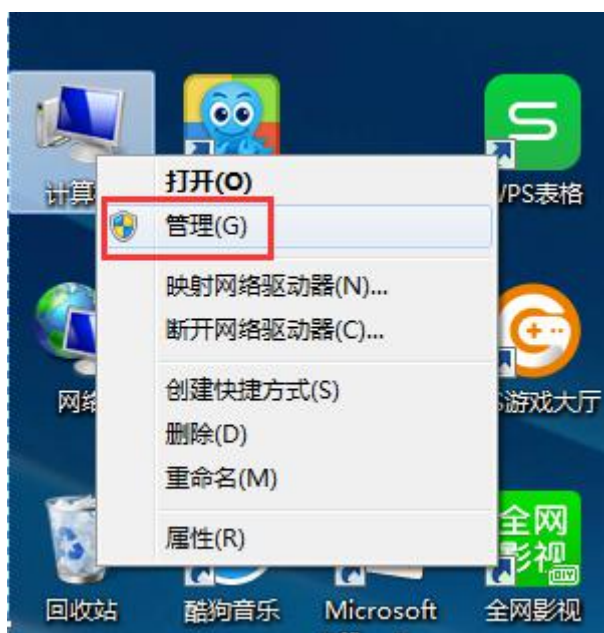


图 8.3.2.1 打开 WIND7 系统电脑的管理窗口

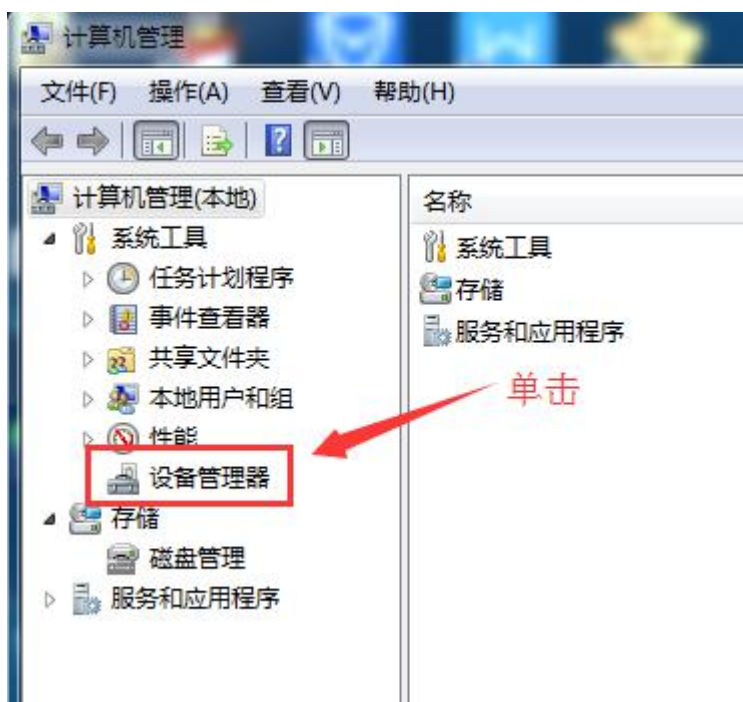


图 8.3.2.2 打开设备管理器窗口

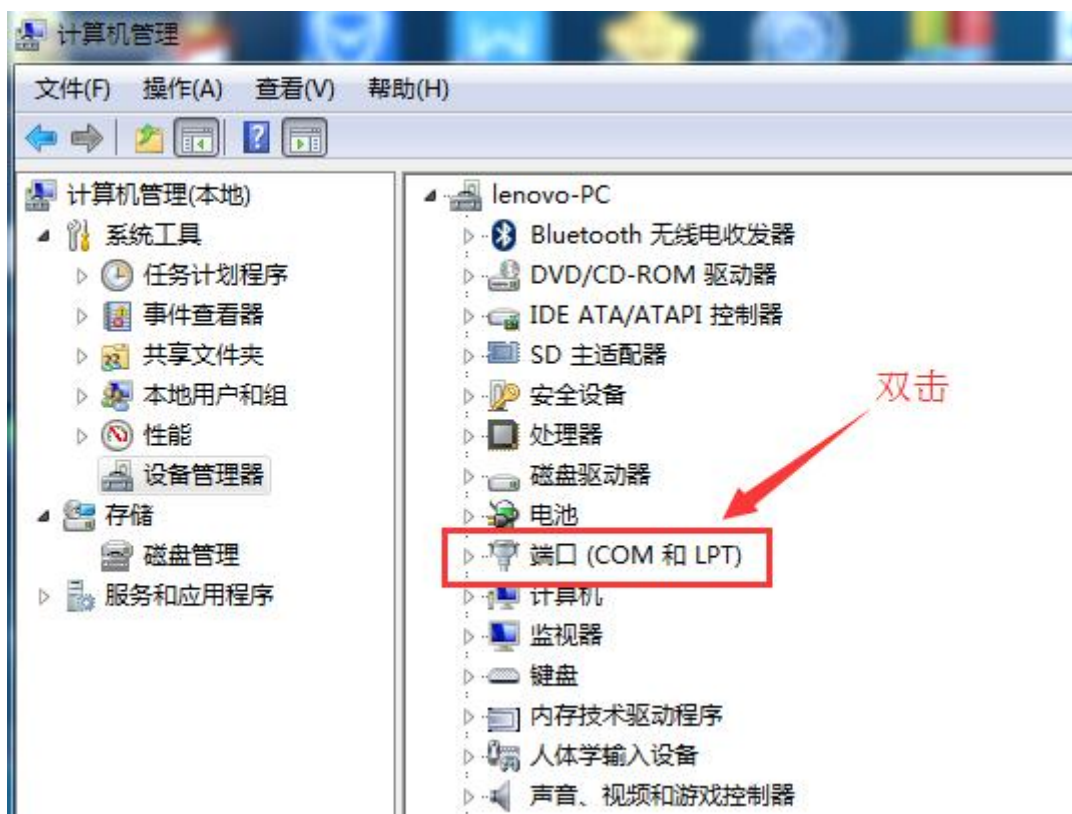


图 8.3.2.3 查看 COM 口号

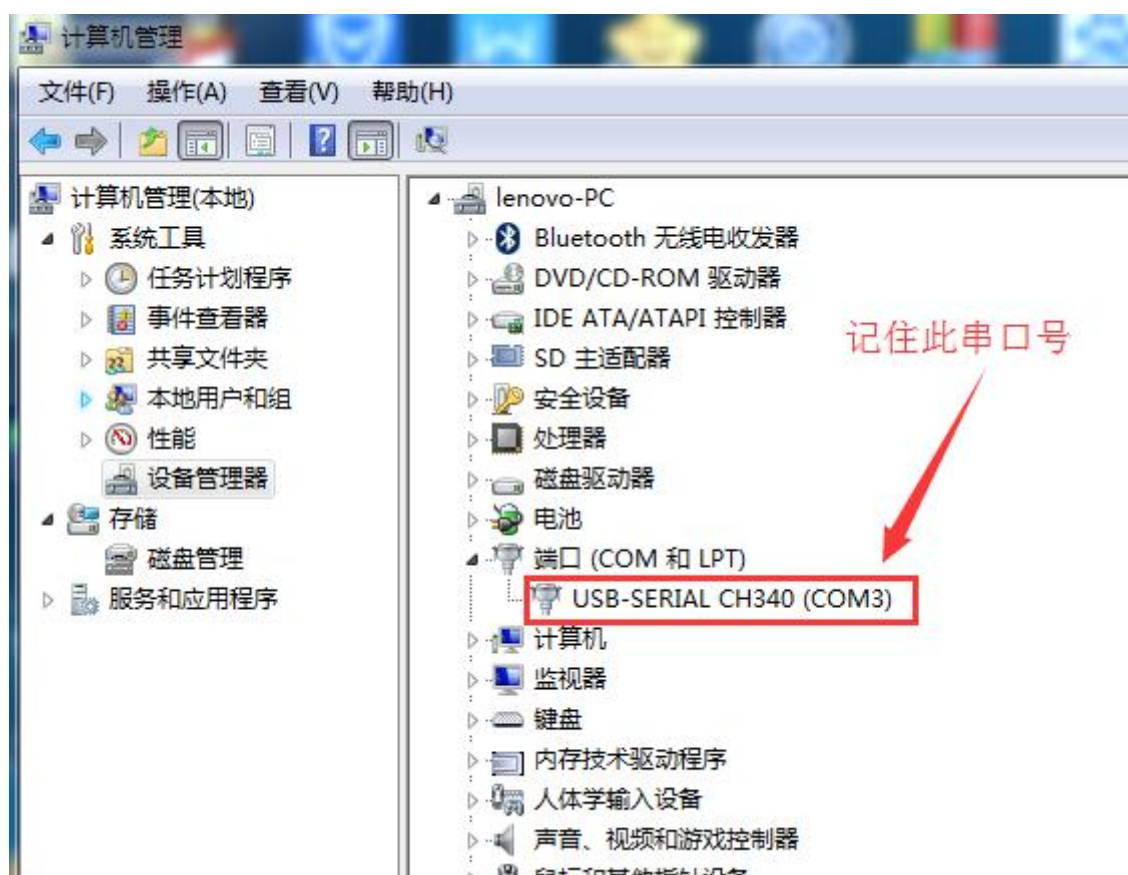


图 8.3.2.4 记录当前正用到的 COM 口号

第二步：硬件线路连接，同时记录串口号。

把 USB 转串口线插入电脑 USB 口，此时 USB 转串口线的另外一端连接 51 学习板的 9 针串口。同时，电源线一端用输出的 5V 手机充电器 USB 端口供电，电源线另一端连接 51 学习板的 USB 供电端口，此时可以通过 51 学习板的电源拨动开关来控制断电和上电。然后是查找串口号，方法是：以电脑 WIN7 系统为例，右击桌面“计算机”，单击选择下拉菜单的“管理”选项，在弹出的窗口中，点击“设备管理器”选项切换到对应的设置窗口，双击“端口（COM 和 LPT）”选项，在展开的下拉选项中，会看到“USB-SERTAL CH340 (COM3)”，这个 COM3 就是我们要记住的串口号，记住此串口号，后面的步骤要用到。你们的串口号不一定是 COM3，请以你们电脑显示的串口号为准。

-----步骤之间的分割线-----



图 8.3.3 双击打开上位机软件

第三步：打开上位机用户软件“stc-isp-15xx-v6.85I.exe”。

这个软件可以在宏晶单片机的官网下载获取，获取到的软件压缩包只需解压后就可以使用，不用安装，直接双击打开“stc-isp-15xx-v6.85I.exe”，此时会弹出“温馨提示”的窗口，我们按“确定”就可以进入到真正的工作界面了。

-----步骤之间的分割线-----





“最低波特率”设置为 2400，，“最高波特率”设置为 9600。波特率如果设置太高，可能会导致烧录（下载）不成功。

-----步骤之间的分割线-----

第七步：导入 .hex 格式的机器码文件。

点击“打开程序文件”的按钮，在弹出的对话框中，选择 D 盘下“stc89c52rc”文件夹目录下的“stc89c52rc.hex”，双击把“stc89c52rc.hex”导入到上位机用户软件。

-----步骤之间的分割线-----

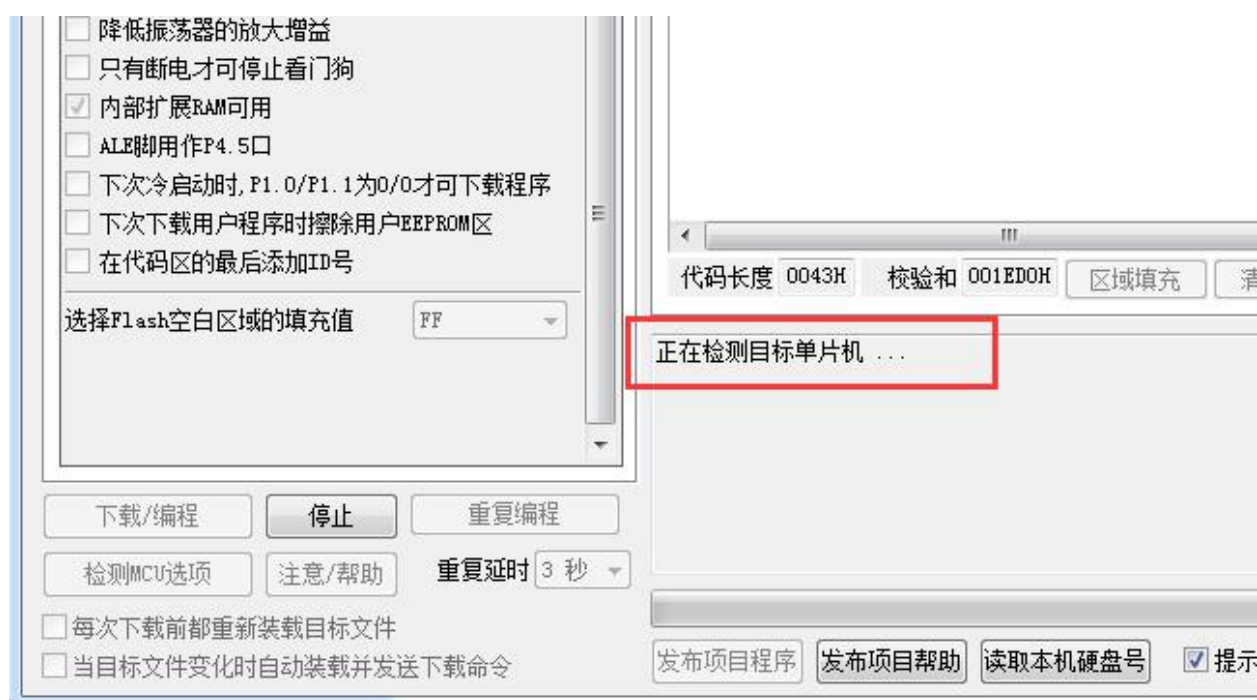


图 8.3.8 等待 51 学习板重新断电再上电

第八步：启动下载。

点击“下载/编程”的按钮，发现“正在检测目标单片机..”的提示信息，此时需要通过电源波动开关把 51 学习板重新断电然后再上电才能正常下载，很多人也把这个重新上电的过程称为“冷启动”。之所以要重新断电再上电，是因为单片机内部预置的系统程序只在上电短暂的瞬间才会检查一下是否接收到需要重新烧录程序的指令，如果没有接收到烧录指令，单片机整个话语权就由原来的系统程序转交给我们的用户程序来接管，所以此串口后面的时间就给我们用户程序来使用。因此每次烧录程序时，先启动上位机的下载命令，此时上位机不断发送请求下载的命令给单片机，但是此时单片机并不理会这些指令，因为此时单片机的话语权已经交给了我们的用户程序，此时并不是预置系统程序在掌控，所以除非重新断电然后再上电那一瞬间才会让系统内置程序去检测并且响应此下载命令。另外多说一句，其实不是所有厂家的单片机在烧录程序时都需要“冷启动”，也不是所有单片机都支持串口烧录，各厂家的单片机烧录程序方式会有一些差异，但基本原理是一样的，大同小异。

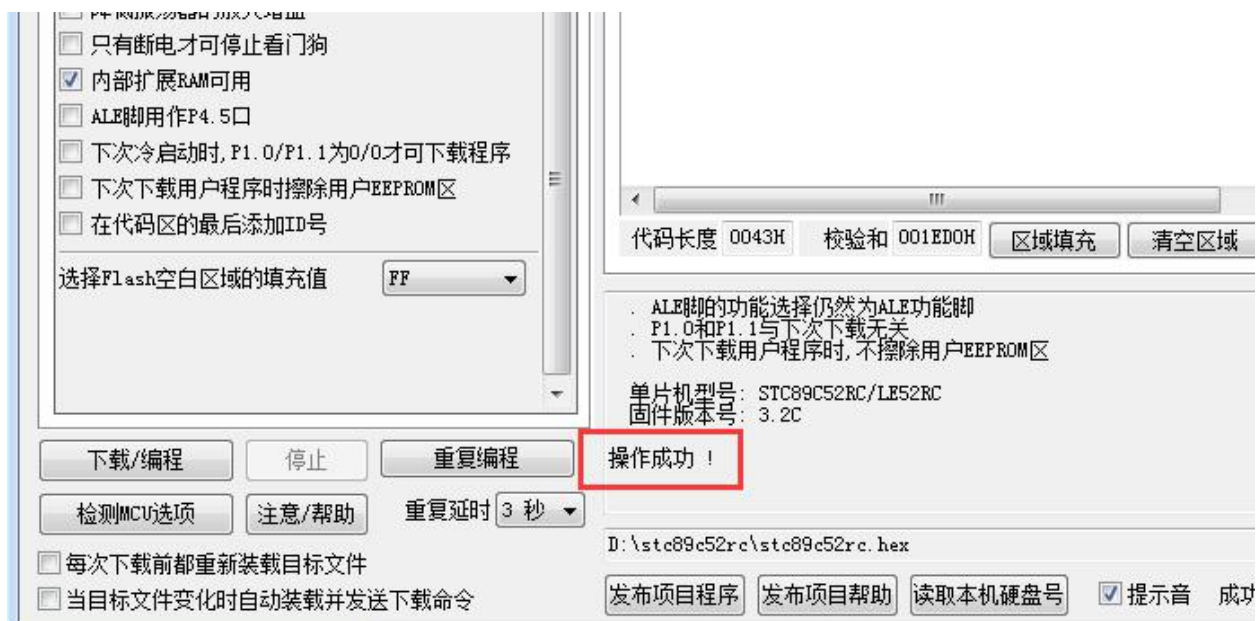


图 8.3.9 烧录（下载）成功

第九步：“冷启动”后观察是否操作成功的信息。

执行完前面第九步的“冷启动”后，如果发现有“... 操作成功!”的提示信息，就说明程序下载成功了。此时会发现 51 学习板上上面的一颗 LED 灯不断闪烁，是因为我们的 LED 灯程序驱动它才开始闪烁的，说明我们的程序在单片机上正常工作了。

补充说明：

(1) 以后只要每次重新编译了 C 源代码后，都会生成最新版本的 .hex 格式文件，所以每次烧录程序时，必须重新返回第七步，重新执行一次导入最新版本 .hex 格式文件的操作，确保被烧录的程序是最新版本的 .hex 烧录文件。

#### 【8.4 51 学习板下载程序失败时的解决办法。】

(1) 可以先松一下单片机卡座，稍微挪动一下单片机，然后再卡紧单片机。卡座必须卡紧单片机，避免接触不良。

(2) 改变供电电源，很多电脑的 USB 口供电电源干扰非常大，严重影响下载程序，请把 USB 电源线插入到手机充电器 5V 的 USB 接口，效果显著，明显提高了下载的成功率。

(3) 检查确保所选择的单片机型号是 STC89C/LE52RC，如果软件弹出推荐其它型号的单片机窗口，不用管它，我们就选 STC89C/LE52RC。

(4) 检查 STC-ISP 烧写软件是否选择匹配的 COM 口。

(5) 单片机是靠串口烧录程序进去的，单片机的串口是 P3.0, P3.1 两根线经过 232 转换芯片，然后才与 USB 转串口线连接的。因此，在烧录程序时，请确保 P3.0, P3.1 两个 I/O 口不能跳线连接到其它外围元器件上。

(6) 点击“下载/编程”后，记得再断电并且重新上电一次。看看是否烧录成功。

(7) 确保最低波特率一直设置为 2400，最高波特率为 9600。如果还不行再把最高波特率也改成 2400 试试。

(8) 如果还不行，就退出软件，拔掉 USB 转串口线，同时断电（必须把整根电源线拔出！），重新插入 USB 串口线，重新插入电源线开电，重新打开软件。

(9) 如果还不行，学习板先断电（必须把整根电源线拔出！），然后重启一次电脑。

(10) 总之：如果还不行，就按上述步骤多折腾几次。

(11) 最后实在不行，就尝试更换到其它 USB 口，或者尝试更换到其它电脑上试试。



## 第九节：本节预留。

本节预留。

## 第十节：程序从哪里开始，要到哪里去？

程序从哪里开始，要到哪里去？为了让初学者了解C语言程序的执行顺序，我把程序分成三个区域：进入主程序前的区域，主程序的初始化区域，主程序的循环区域。当然，这里三个区的分类暂时没有把中断程序的情况考虑进去，中断程序的内容我会在后面相关的章节中再详细介绍，这里暂时不考虑中断。

进入主程序前的区域。这是上电后，在单片机执行主程序代码之前就已经完成了的工作。包括头文件的包含，宏定义，内存分配这些工作。这部分的内容可以暂时不用去了解，我会在后面的一些章节中陆续深入讲解。

主程序的初始化区域。这是上电后，单片机进入主程序后马上就要执行的程序代码，这部分区域的代码有一个特点，大家也必须记住的，就是单片机只执行一次。只要单片机不重启，不复位，那么上电后这部分的代码只被执行一次。

主程序的循环区域。单片机在主程序中执行完了初始化区域的代码，紧接着就进入这片循环区域的代码。单片机一直在逐行循环执行这些代码，执行到末尾时又返回到循环区域的开始处继续开始新一轮的执行，周而复始，往复循环，这就是上电后单片机的最终归宿，一直处在循环的状态。

下面我跟大家分析一个程序源代码的三个区域和执行顺序，大家先看中文解释部分的内容，暂时不用理解每行指令的语法，有个整体的认识就可以了。此源代码实现的功能是：上电后，蜂鸣器鸣叫一声就停止（初始化区域），然后看到一个LED灯一直在不停的闪烁（循环区域）。

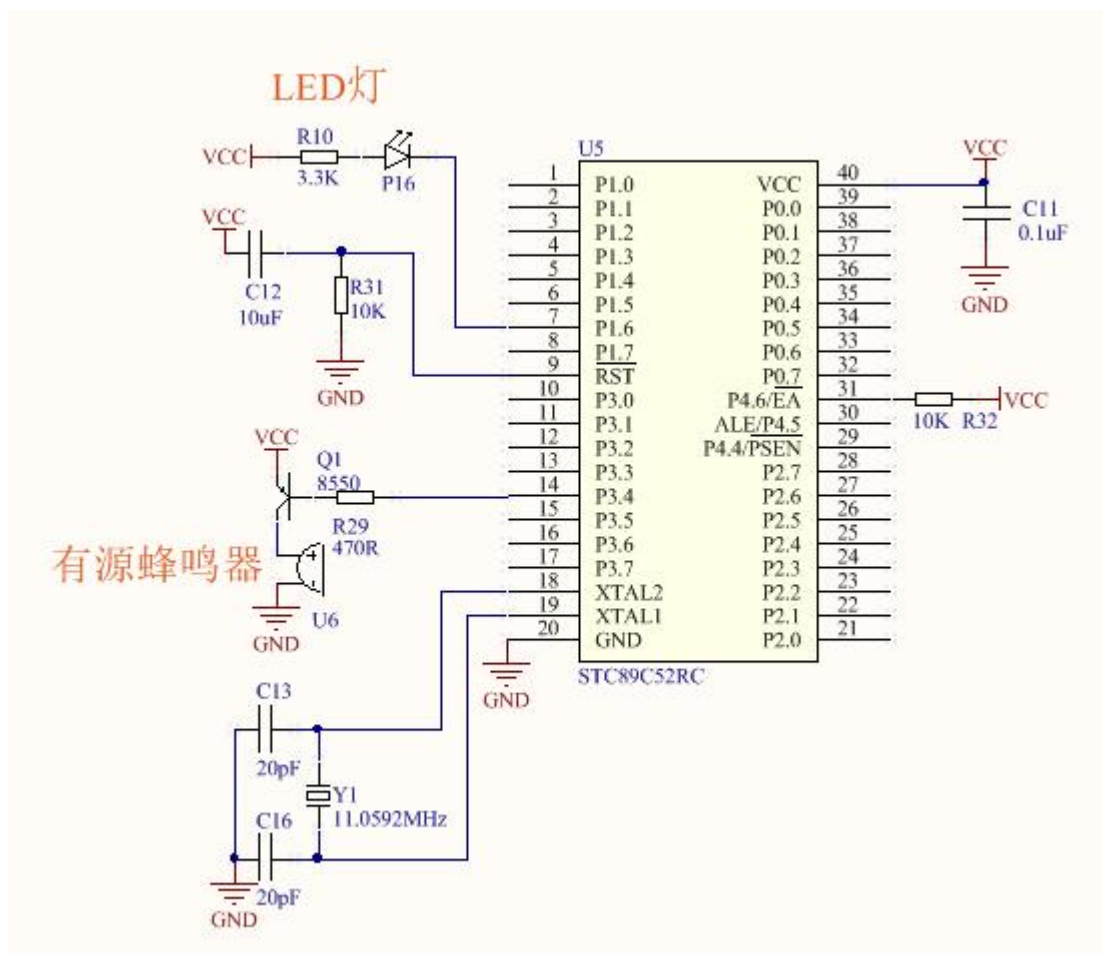


图 10.1 本节示例程序匹配的单片机驱动电路原理图

源代码如下：

```
#include "REG52.H"           //进入主程序前的区域：头文件包含

sbit beep_dr=P3^4;           //进入主程序前的区域：宏定义
sbit led_dr=P1^6;            //进入主程序前的区域：宏定义

unsigned long i;              //进入主程序前的区域：内存分配

void main()                   //主程序入口，即将进入初始化区域
{
    beep_dr=0;                //第一步：初始化区域：蜂鸣器开始鸣叫。
    for(i=0;i<6250;i++);      //第二步：初始化区域：延时 0.5 秒左右。也就是蜂鸣器鸣叫的持续时间。
    beep_dr=1;                //第三步：初始化区域：蜂鸣器停止鸣叫。
    while(1)                   //执行完上面的初始化区域，即将进入循环区域
    {
        led_dr=0;             //第四步：循环区域：LED 开始点亮。
        for(i=0;i<6250;i++);  //第五步：循环区域：延时 0.5 秒左右。也就是 LED 点亮的持续时间。
        led_dr=1;             //第六步：循环区域：LED 开始熄灭。
        for(i=0;i<6250;i++);  //第七步：循环区域：延时 0.5 秒左右。也就是 LED 熄灭的持续时间。
    }                           //执行完上面第七步后，单片机又马上返回到上面第四步继续往下执行。
}
```

上述代码执行顺序分析：

单片机进入主程序后，从第一步到第三步是属于初始化区域，只被执行一次。然后进入循环区域，从第四步执行到第七步，执行完第七步之后，马上又返回上面第四步继续循环往下执行，单片机一直处于第四步到第七步的往复循环中。可以很清晰的看到，上面的 main 和 while(1) 关键词就是三个区域的边界分割线。经过以上的分析，可以看出这三个区域的大概分布如下：

```
//... 进入主程序前的区域
void main()
{
    //... 初始化区域
    while(1)
    {
        //... 循环区域
    }
}
```

## 第十一节：一个在单片机上练习 C 语言的模板程序。

### 【11.1 一套完整的模板源代码。】

先给大家附上一套完整的模板源代码，后面章节练习 C 语言的模板程序就直接复制此完整的源代码，此源代码适合的单片机型号是 STC89C52RC，晶振是 11.0592MHz，串口波特率是 9600，初学者只需修改代码里从“C 语言学习区域的开始”到“C 语言学习区域的结束”的区域，其它部分不要更改。可复制的源代码请到网上论坛原贴处直接下载本教程的文件压缩包，解压文件压缩包后，直接用 WPS 办公软件打开“可编辑的 WPS 文档教程”这个文档，就可以复制里面相关章节的源代码。在网上搜索“从单片机基础到程序框架”就可以找到论坛原贴的出处，也可以直接到我的个人网站那里下载（[www.dumenmen.com](http://www.dumenmen.com)）。一套完整的模板源代码如下：

```
#include "REG52.H"
void View(unsigned long u32ViewData);
void to_BufferData(unsigned long u32Data,unsigned char *pu8Buffer,unsigned char u8Type);
void SendString(unsigned char *pu8String);

/*---C 语言学习区域的开始。-----*/

void main() //主函数
{
    unsigned char a; //定义一个变量 a。
    unsigned int  b; //定义一个变量 b。
    unsigned long c; //定义一个变量 c。
    a=100;           //给变量 a 赋值。
    b=10000;          //给变量 b 赋值。
    c=10000000000;    //给变量 c 赋值。
    View(a); //把第 1 个数 a 发送到电脑端的串口助手软件上观察。
    View(b); //把第 2 个数 b 发送到电脑端的串口助手软件上观察。
    View(c); //把第 3 个数 c 发送到电脑端的串口助手软件上观察。
    while(1)
    {
    }
}

/*---C 语言学习区域的结束。-----*/

void View(unsigned long u32ViewData)
{
    static unsigned char Su8ViewBuffer[43];
    code unsigned char Cu8_0D_0A[]={0x0d,0x0a,0x00};
    code unsigned char Cu8Start[]={"开始..."};
    static unsigned char Su8FirstFlag=0;
    static unsigned int Su16FirstDelay;
```

```

        if(0==Su8FirstFlag)
        {
            Su8FirstFlag=1;
            for(Su16FirstDelay=0;Su16FirstDelay<10000;Su16FirstDelay++);
            SendString(Cu8Start);
            SendString(Cu8_OD_0A);
            SendString(Cu8_OD_0A);
        }

        to_BufferData(u32ViewData, Su8ViewBuffer, 1);
        SendString(Su8ViewBuffer);
        to_BufferData(u32ViewData, Su8ViewBuffer, 2);
        SendString(Su8ViewBuffer);
        to_BufferData(u32ViewData, Su8ViewBuffer, 3);
        SendString(Su8ViewBuffer);
        to_BufferData(u32ViewData, Su8ViewBuffer, 4);
        SendString(Su8ViewBuffer);
        SendString(Cu8_OD_0A);
    }

void to_BufferData(unsigned long u32Data,unsigned char *pu8Buffer,unsigned char u8Type)
{
    code unsigned char Cu8Array1[]={0xB5, 0xDA, 0x4E, 0xB8, 0xF6, 0xCA, 0xFD, 0x00};
    code unsigned char Cu8Array2[]="十进制:";
    code unsigned char Cu8Array3[]="十六进制:";
    code unsigned char Cu8Array4[]="二进制:";
    static unsigned char Su8SerialNumber=1;
    static unsigned int  Su16BufferCnt;
    static unsigned int  Su16TempCnt;
    static unsigned int  Su16TempSet;
    static unsigned long Su32Temp1;
    static unsigned long Su32Temp2;
    static unsigned long Su32Temp3;
    static unsigned char Su8ViewFlag;
    if(1==u8Type)
    {
        for(Su16BufferCnt=0;Su16BufferCnt<7;Su16BufferCnt++)
        {
            pu8Buffer[Su16BufferCnt]=Cu8Array1[Su16BufferCnt];
        }
        pu8Buffer[2]=Su8SerialNumber+'0';
        pu8Buffer[Su16BufferCnt]=0x0d;
        pu8Buffer[Su16BufferCnt+1]=0x0a;
        pu8Buffer[Su16BufferCnt+2]=0;
        Su8SerialNumber++;
        return;
    }

```

```

    }
else if (2==u8Type)
{
    for(Su16BufferCnt=0;Su16BufferCnt<7;Su16BufferCnt++)
    {
        pu8Buffer[Su16BufferCnt]=Cu8Array2[Su16BufferCnt];
    }
    Su32Temp1=1000000000;
    Su32Temp2=10;
    Su16TempSet=10;
}
else if (3==u8Type)
{
    for(Su16BufferCnt=0;Su16BufferCnt<9;Su16BufferCnt++)
    {
        pu8Buffer[Su16BufferCnt]=Cu8Array3[Su16BufferCnt];
    }
    Su32Temp1=0x10000000;
    Su32Temp2=0x00000010;
    Su16TempSet=8;
}
else
{
    for(Su16BufferCnt=0;Su16BufferCnt<7;Su16BufferCnt++)
    {
        pu8Buffer[Su16BufferCnt]=Cu8Array4[Su16BufferCnt];
    }
    Su32Temp1=0x80000000;
    Su32Temp2=0x00000002;
    Su16TempSet=32;
}
Su8ViewFlag=0;
for(Su16TempCnt=0;Su16TempCnt<Su16TempSet;Su16TempCnt++)
{
    Su32Temp3=u32Data/Su32Temp1%Su32Temp2;
    if (Su32Temp3<10)
    {
        pu8Buffer[Su16BufferCnt]=Su32Temp3+'0';
    }
    else
    {
        pu8Buffer[Su16BufferCnt]=Su32Temp3-10+'A';
    }
    if (0==u32Data)

```

```

        {
            Su16BufferCnt++;
            break;
        }
        else if(0==Su8ViewFlag)
        {
            if('0'!=pu8Buffer[Su16BufferCnt])
            {
                Su8ViewFlag=1;
                Su16BufferCnt++;
            }
        }
        else
        {
            Su16BufferCnt++;
        }

        Su32Temp1=Su32Temp1/Su32Temp2;
    }
    pu8Buffer[Su16BufferCnt]=0x0d;
    pu8Buffer[Su16BufferCnt+1]=0x0a;
    pu8Buffer[Su16BufferCnt+2]=0;
}

void SendString(unsigned char *pu8String)
{
    static unsigned int Su16SendCnt;
    static unsigned int Su16Delay;
    SCON=0x50;
    TMOD=0X21;
    TH1=TL1=256-(11059200/12/32/9600);
    TR1=1;
    ES = 0;
    TI = 0;
    for (Su16SendCnt=0;Su16SendCnt<43;Su16SendCnt++)
    {
        if(0==pu8String[Su16SendCnt])
        {
            break;
        }
        else
        {
            SBUF =pu8String[Su16SendCnt];
            for (Su16Delay=0;Su16Delay<800;Su16Delay++);
            TI = 0;

```



```
}  
  
}  
  
}
```

## 【11.2 模板程序的使用说明。】

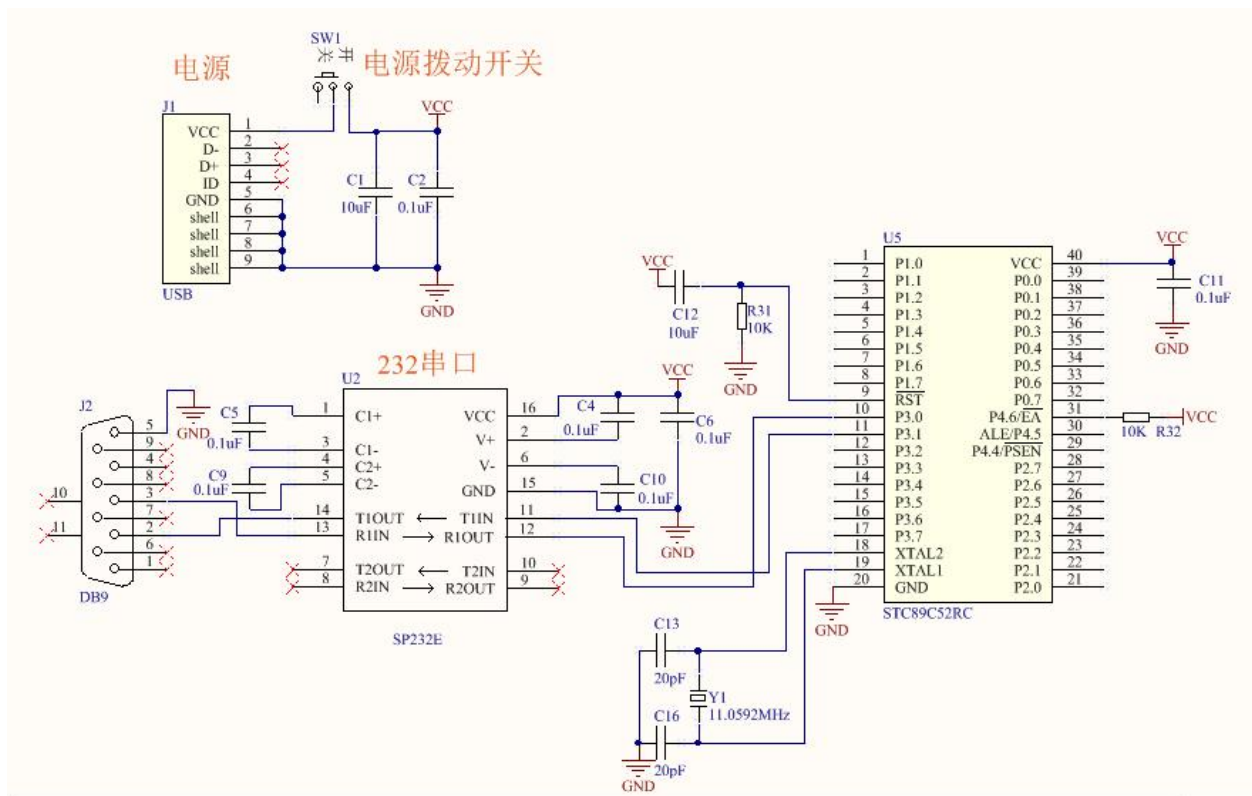


图 11.2.1 带串口的单片机最小系统

大多数初学者在学习 C 语言的时候，往往是在电脑端安装上 VC 平台软件来练习 C 语言，这种方法只要在代码里调用 printf 语句，编译后就可以看到被 printf 语句调用的变量，挺方便的。本教程没有用这种方法，既然本教程的 C 语言主要针对单片机，所以我想出了另外一种方法，这种方法就是直接在单片机上练习 C 语言，这样会让初学者体验更深刻。这种方法对硬件平台要求不高，只要 51 学习板上有一个 9 针的串口就可以，这个串口既可以用来烧录程序，也可以用来观察代码里的某个变量，只要在代码里调用 View 函数就可以达到类似 VC 平台软件下 printf 语句的效果，View 函数可以向串口输出某个变量的十进制，十六进制和二进制，大家只要在电脑端的串口助手软件就可以看到某个变量的这些信息，View 函数能查看的变量最大数值范围是 4 个字节的 unsigned long 变量，十进制的范围是从 0 到 4294967295，也可以查看 unsigned int 和 unsigned char 的类型变量（数据的进制以及 long, int, char 等知识点大家目前还没接触到，因此不懂也没关系，当前只要有个大概的认识就可以，暂时不用深入理解，后面章节还会详细介绍）。View 函数是我整个模板程序的其中一部分，所以要用这种方法就必须先复制我整个模板程序，初学者练习代码的活动范围仅仅局限于模板程序里的“C 语言学习区域”，在此区域里有一个 main 主函数，main 主函数内有一个初始化区域，初学者往往在这个初始化区域里练习 C 语言就够了，初学者最大的活动范围不能超过从“C 语言学习区域的开始”到“C 语言学习区域的结束”这个范围，这个范围之外其它部分的代码主要用来实现数据处理和串口发送的功能，大家暂时不用读懂它，直接复制过来就可以了。比如：

```

/*---C 语言学习区域的开始。-----*/

void main() //主函数
{
    //... 初始化区域，也就是主要用来给初学者学习 C 语言的区域。
    while(1)
    {

    }

}

/*---C 语言学习区域的结束。-----*/

```

上述例子中，初学者练习代码只能在从“C 语言学习区域的开始”到“C 语言学习区域的结束”这个范围，此范围外的代码直接复制过来不要更改。我们再来分析分析下面节选的 main 函数源代码：

```

/*---C 语言学习区域的开始。-----*/

void main() //主函数
{
    unsigned char a; //定义一个变量 a。
    unsigned int  b; //定义一个变量 b。
    unsigned long c; //定义一个变量 c。
    a=100;           //给变量 a 赋值。
    b=10000;         //给变量 b 赋值。
    c=1000000000;    //给变量 c 赋值。
    View(a);         //在电脑串口端查看第 1 个数 a。
    View(b);         //在电脑串口端查看第 2 个数 b。
    View(c);         //在电脑串口端查看第 3 个数 c。
    while(1)
    {
    }
}

/*---C 语言学习区域的结束。-----*/

```

上述节选的 main 函数代码里，比如“a=100; //给变量 a 赋值。”这行代码，所谓的“赋值”就是“=”这个语句，它表面上像我们平时用的等于号，实际上不是等于号，而是代表“给”的意思，把“=”符号右边的数复制一份给左边的变量，比如“a=100;”就是代表把 100 这个数值复制一份给变量 a，执行这条指令后，a 就等于 100 了。这里的分号“;”代表一条程序指令的结束。而双斜线“//”是注释语句，双斜线“//”这行后面的文字或字符都是用来注释用的，编译器会忽略双斜线“//”这一行后面的文字或字符，编译器不把注释文字或字符列入源代码，也就是“//”这一行中后面的文字或字符是不占单片机内存的。当然“//”仅仅局限于当前一行代码。上面除了“//”是注释语句外，上面的“/\*”和“\*/”之间也是注释语

接着在分析上述代码中最重要的函数，也是本节最核心最重要的函数 View(某个变量)。比如“View(a);”这行代码，View(a)就是要把变量 a 的十进制，十六进制和二进制的数值都发送到串口，我们通过 USB 转串口线让学习板连接上电脑，在电脑串口助手软件上就能看到被 View 函数调用的变量 a 的信息。

前面章节在讲烧录程序时提到一个叫“stc-isp-15xx-v6.85I”的上位机软件，这个软件除了用来烧录程序，还集成了串口助手软件的功能。所以本节直接共用烧录程序时的 USB 转串口线和“stc-isp-15xx-v6.85I”软件就可以了，无需额外再购买新的 USB 转串口线和下载其它串口助手软件，但是如何设置这个“stc-isp-15xx-v6.85I”上位机软件，还是有一些需要特别注意的地方的，现在把这个详细的步骤介绍给大家。

按前面章节介绍烧录程序时所需的步骤，用 USB 转串口线连接 51 学习板和电脑，记录 COM 号，打开“stc-isp-15xx-v6.85I”软件，选择单片机型号，选择对应的串口号（COM 号），设置最低波特率和最高波特率，这部分的内容跟烧录程序时的配置步骤是一样的，唯一必须要特别注意的是最高波特率必须选择 9600！最低波特率建议选择 2400。否则在烧录完程序后，当上位机集成软件自动切换到串口助手软件窗口时，接收区域显示的一些汉字信息可能会出现乱码。

STC-ISP (V6.851)

单片机型号: STC89C52RC/LE52RC 引脚数: Auto

串口号: USB-SERIAL CH340 (COM3) 扫描

最低波特率: 2400 最高波特率: 9600

起始地址: 0x0000 ☒ 清除代码缓冲区 打开程序文件

0x2000 ☒ 清除EEPROM缓冲区 打开EEPROM文件

硬件选项: 脱机下载/U8/U7 程序加密后传输 ID: 1

☐ 使能6T (双倍速)模式

☐ 降低振荡器的放大增益

☐ 只有断电才可停止看门狗

☒ 内部扩展RAM可用

☐ ALE脚用作P4.5口

☐ 下次冷启动时, P1.0/P1.1为0/0才可下载程序

☐ 下次下载用户程序时擦除用户EEPROM区

☐ 在代码区的最后添加ID号

选择Flash空白区域的填充值: FF

程序文件 EEPROM文件 **串口助手** Keil仿真设置 选型/价格/...

接收缓冲区

☒ 文本模式 ☐ HEX模式

清空接收区 保存接收数据

发送缓冲区

☐ 文本模式 ☒ HEX模式

清空发送区 保存发送数据

发送文件 发送数据 ☒ 自动发送 周期(ms) 100

串口: COM3 波特率: 9600 校验位: 无校验

☒ 编程完成后自动打开串口

打开串口 ☐ 将U8/U7设置为标准USB转串口

ALE脚的功能选择仍然为ALE功能脚

P1.0和P1.1与下次下载无关

下次下载用户程序时, 不擦除用户EEPROM区

图 11.3.2 设置上位机的串口助手选项

第二步：设置串口助手软件的选项。

先点击右上方选中“串口助手”选项切换到串口助手的窗口，接收缓冲区选择“文本模式”，串口选择匹配的 COM 号（跟烧录软件一致的 COM 号），波特率必须选择 9600，勾选上“编程完成后自动打开串口”选项，最后点击“打开串口”按钮使之切换到显示“关闭串口”的文字状态，至此串口助手软件的安装完毕。接下来就是按烧录程序的流程，打开新的 HEX 程序文件，程序烧录完成后上位机软件会自动切换到串口助手的串口，就可以观察到 View 函数从单片机上发送过来的某个变量的十进制，十六进制，二进制的信息了。接收缓冲区的窗口比较小，如果收到的信息比较多，只要在上下方向拖动窗口右边的滑块就可以依次看到全部的信息。如果想让单片机重新发送数据，只要让 51 学习板断电重启就可以重发一次数据，当串口助手的接收区接收的信息太多影响观察时，大家可以点击“清空接收区”的按钮来清屏，然后断电重启让它再重发一次数据。在电脑的串口助手软件里观察到的数据格式大概是什么样子的呢？比如编译完本章节上述完整的模板源代码程序后，会在串口助手软件里看到 a, b, c 三个变量的信息如下：

```
开始...

第 1 个数
十进制:100
十六进制:64
二进制:1100100

第 2 个数
十进制:10000
十六进制:2710
二进制:10011100010000

第 3 个数
十进制:1000000000
十六进制:3B9ACA00
二进制:111011100110101100101000000000
```

多说一句，烧录程序后，当软件自动切换到串口助手软件选项的窗口时，串口助手窗口显示单片机返回的信息，这时有可能第一行的文字“开始...”会丢失或者显示不出来，但是后面其它的关键信息不受影响，我猜测可能是串口助手软件本身的某个环节存在的小 bug，跟我们没关系，我们不用深究其原因，因为不会影响我们的使用，此时也有一种解决办法，就是只要让单片机断电重启重发一次数据就可以正确地看到第一行的文字“开始...”。

#### 【11.4 如何利用现有的工程编辑编译新的源代码？】

本教程后面有很多章节的源代码，是不是每个章节都要重新建一个工程？其实不用。我们只要用一个工程就可以编译编辑本教程所有章节的源代码。方法很简单，就是打开一个现有的工程，用快捷组合键“Ctrl+A”把原工程里面的 C 源代码全部选中，再按“Backspace”清空原来的代码，然后再复制本教程相关章节的代码粘贴到工程的 C 文档里，重新编译一次就可以得到对应的 Hex 格式的烧录文件。用这种方法的时候，建议大家做好每个程序代码的备份。每完成一个项目的小进度，都要及时把源代码存储到电脑硬盘里，电脑硬盘

里每个项目对应一个项目文件夹，每个项目文件夹里包含很多不同版本编号的源代码文件，每个源代码文件名都有流水编号，方便识别最新版本的程序，每天下班前都要把最新版本的源代码文件上传到自己的网盘里备份，在互联网时代，把源代码存到自己的网盘，可以随时异地存取，即使遇到电脑故障损坏也不担心数据永久丢失。

### 【11.5 编辑源代码的 5 个常用快捷键。】

介绍一下常用的快捷键，好好利用这 5 个快捷键，会让你在编辑源代码时效率明显提高。

- (1) 选中整篇所有的内容：组合键 Ctrl+A。
- (2) 把选中的内容复制到临时剪贴板：组合键 Ctrl+C。
- (3) 把临时剪贴板的内容粘贴到光标开始处：组合键 Ctrl+V。
- (4) 把选中的一行或者几行内容整体往右边移动：单键 Tab。每按一次就移动几个空格，很实用。
- (5) 把选中的一行或者几行内容整体往左边移动：组合键 Shift+Tab。每按一次就移动几个空格，很实用。

## 第十二节：变量的定义和赋值。

### 【12.1 学习 C 语言的建议和方法。】

先提一些学 C 语言的建议和方法，帮大家删繁就简，去掉一些初学者常见的思想包袱。现阶段我们的学习是使用单片机，把单片机当做一个成品，把单片机当做一个忠诚的士兵，学习 C 语言就是学习如何使用单片机，如何命令单片机，如何让单片机听懂我们的话并且听我们指挥。单片机内部太细节的构造原理暂时不用过多去关注，只要知道跟我们使用相关的几个特征就可以，这样初学者的学习包袱就没有那么重，就可以把重点放在使用上的，而不是好奇于根本原理的死磕到底。学 C 语言跟学习英语的性质是一样的，都是在学习一门外语，只是 C 语言比英语的语法要简单很多，非常容易上手，词汇量也没有英语那么多，C 语言常用单词才几十个而已。学习任何一门语言的秘诀在于练习，学习 C 语言的秘诀是多在单片机上练习编程。本教程后面几乎每个章节都有例题，这个例题很重要，初学者即使看懂了，我也强烈建议要把“C 语言学习区域”的那部分代码亲自上机敲键盘练习一遍，并且看看实验现象是否如你所愿。

### 【12.2 变量定义和赋值的感性认识。】

这些年我用过很多单片机，比如 51，PIC，LPC17 系列，STM8，STM32 等单片机。尽管各类单片机有一些差异，但是在选单片机时有 3 个参数我们一定会关注的，它们分别是：工作频率，数据存储器 RAM，程序存储器 ROM。工作频率跟晶振和倍频有关，决定了每条指令所要损耗的时间，从而决定了运算速度。RAM 跟代码里所定义变量的数量有关。ROM 跟程序代码量的大小有关。程序是什么？程序就是由对象和行为两者构成的。对象就是变量，就是变量的定义，就是 RAM，RAM 的大小决定了一个程序允许的对象数量。行为就是赋值，判断，跳转，运算等语法，就是 ROM，ROM 的大小决定了一个程序允许的行为程度。本节的标题是“变量的定义和赋值”，其中“定义”就是对象，“赋值”就是行为。

### 【12.3 变量的定义。】

变量的定义。一个程序最大允许有多少个对象，是由 RAM 的字节数决定的(字节是一种单位，后面章节会讲到)。本教程的编译环境是以 AT89C52 芯片为准，AT89C52 这个单片机有 256 个字节的 RAM，但是并不意味着程序就一定要全部占用这些 RAM。程序需要占用多少 RAM，完全是根据程序的实际情况来决定，需要多少就申请多少。这里的“对象”就是变量，这里的“申请”就是变量的定义。

定义变量的关键字。常用有 3 种容量的变量，每种变量的取值范围不一样。第一种是“unsigned char”变量，取值范围从 0 到 255，占用 RAM 一个字节，比喻成一房一厅。第二种是“unsigned int”变量，取值范围从 0 到 65535，占用 RAM 两个字节，比喻成两房一厅。第三种是“unsigned long”变量，取值范围从 0 到 4294967295，占用 RAM 四个字节，比喻成四房一厅。unsigned char, unsigned int 和 unsigned long 都是定义变量的关键字，所谓关键字也可以看成是某门外语的单词，需要大家记忆的，当然不用死记硬背，只要多上机练习就自然熟记于心，出口成章。多说一句，上述的变量范围是针对本教程所用的单片机，当针对不同的单片机时上述变量的范围可能会有一些小差异，比如在 stm32 单片机中，unsigned int 的字节数就不是两个字节，而是四个字节，这些都是由所选的编译器决定的，大家暂时有个初步了解就可以。

定义变量的语法格式。定义变量的语法格式由 3 部分组成：关键字，变量名，分号。比如：

```
unsigned char a;
```

其中 unsigned char 就是关键字，a 就是变量名，分号“;”就是一条语句的结束符号。

变量名的命名规则。变量名的第一个字符不能是数字，必须是字母或者下划线，字母或者下划线后面可以带数字，一个变量名之间的字符不能带空格，两个独立变量名之间也不能用空格隔开（但是两个独立变量名之间可以用逗号隔开）。变量名不能跟编译器已征用的关键字重名，不能跟函数名重名，这个现象跟古代

要求臣民避讳皇帝的名字有点像。哪些名字是合法的，哪些名字是不合法的？现在举一些例子说明：

```
unsigned char 3a; //不合法，第一个字符不能是数字。
unsigned char char; //不合法，char 是编译器已征用的关键字。
unsigned char a b; //不合法，ab 是一个变量名，a 与 b 的中间不能有空格。
unsigned char a,b; //合法，a 和 b 分别是一个独立的变量名，a 与 b 的中间可以用逗号隔开。
unsigned char a; //合法。
unsigned char abc; //合法。
unsigned char _ab; //合法。
unsigned char _3ab; //合法。
unsigned char a123; //合法。
unsigned char a12ced; //合法。
```

定义变量与 RAM 的内在关系。当我们定义一个变量时，相当于向单片机申请了一个 RAM 空间。C 编译器会自动为这个变量名分配一个 RAM 空间，每个字节的 RAM 空间都有一个固定唯一的地址。把每个字节的 RAM 空间比喻成房间，这个地址就是房号。地址是纯数字编号，不利于我们记忆，C 语言编译器为了降低我们的工作难度，不用我们记每个变量的地址，只需要记住这个变量的名称就可以了。操作某个变量名，就相当于操作某个对应地址的 RAM 空间。变量名与对应地址 RAM 空间的映射关系是 C 编译器暗中悄悄帮我们分配好的。比如：

```
unsigned char a;    //a 占用一个字节的 RAM 空间，这个空间的地址由 C 编译自动分配。
unsigned char b;    //b 占用一个字节的 RAM 空间，这个空间的地址由 C 编译自动分配。
unsigned char c;    //c 占用一个字节的 RAM 空间，这个空间的地址由 C 编译自动分配。
```

上述 a, b, c 三个变量各自占用一个字节的 RAM 空间，同时被 C 编译器分配了 3 个不同的 RAM 空间地址。

变量定义的初始化。变量定义之后，等于被 C 编译器分配了一个 RAM 空间，那么这个空间里面存储的数据是什么？如果没有刻意给它初始化，RAM 空间里面存储的数据是不太确定的，是默认的。有些场合，需要在给变量分配 RAM 空间时就给它一个固定的初始值，这就是变量定义的初始化。变量初始化的语法格式由 3 部分组成：关键字，变量名赋值，分号。比如：

```
unsigned char a=9;
```

其中 unsigned char 就是关键字。

其中 a=9 就是变量名赋值。a 从被 C 编译器分配 RAM 空间那一刻起，就默认是预存了一个 9 的数据。

分号 “;” 就是一条语句的结束符号。

## 【12.4 变量的赋值。】

赋值语句的含义。把右边对象的内容复制一份给左边对象。赋值语句有一个很重要的特性，就是覆盖性，左边对象原来的内容会被右边对象复制过来的新内容所覆盖。比如，左边对象是变量 a，假设原来 a 里面存的数据是 3，右边对象是数据 6，执行赋值语句后，会把右边的 6 赋值给了对象 a，那么 a 原来的数据 3 就被覆盖丢失了，变成了 6。

赋值语句的格式。赋值语句的语法格式由 4 部分组成：左边对象，关键字，右边对象，分号。比如：

```
a=b;
```

其中 a 就是左边对象。

其中 “=” 就是关键字。写法跟我们平时用的等于号是一样，但是在 C 语言里不是等于的意思，而是代表赋值的意思，它是代表中文含义的“给”，而不是用于判断的“等于”，跟等于号是两码事（C 语言的等于号是 “==”，这个后面章节会讲到）。

其中 b 就是右边对象。

其中分号 “;” 代表一条语句的结束符。



赋值语句与 ROM 的关系。赋值语句是行为的一种，所以编译会把赋值这个行为翻译成对应的指令，这些指令在下载程序时最终也是以数据的形式存储在 ROM 里，指令也是以字节为单位(字节是一种单位，后面章节会讲到)。本教程的编译环境是以 AT89C52 芯片为准，AT89C52 这个单片机有 8K 的 ROM 容量，也就是有 8192 个字节的 ROM (8 乘以 1024 等于 8192)，但是并不意味着程序就一定要全部占用这些 ROM。程序需要占用多少 ROM，完全是根据程序的行为程度决定，也就是通常所说的你的程序容量有多大，有多少行代码。多说一句，在单片机或者我们常说的计算机领域里，存储容量是以字节为单位，而每 K 之间的进制不是我们日常所用的 1000，而是 1024，所以刚才所说的 8K 不是 8000，而是 8192，这个是初学者很容易迷惑的地方。刚才提到，赋值语句是行为，凡是程序的行为指令都存储在单片机的 ROM 区。C 编译器会把一条赋值语句翻译成对应的一条或者几条机器码，机器码指令也是以字节为单位的。下载程序的时候，这些机器码就会被下载进单片机的 ROM 区。比如以下这行赋值语句：

```
unsigned char a;  
unsigned char b=3;  
a=b;
```

经过 C 编译器编译后会生成以字节为单位的机器码。这些机器码记录着这些信息：变量 a 的 RAM 地址，变量 b 的 RAM 地址和初始化时的预存数据 3，以及把 b 变量的内容赋值给 a 变量的这个行为。所有这些信息，不管是“数据”还是“行为”，本质都是以“数据”(或称数字，数码都可以)的形式存储记录的，单位是字节。

## 【12.5 例程的分析和练习。】

接下来练习一个程序实例。直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。本章节在“C 语言学习区域”练习的代码如下：

```
/*---C 语言学习区域的开始。-----*/  
  
void main() //主函数  
{  
    unsigned char a; //定义的变量 a 被分配了一个字节的 RAM 空间，保存的数据是不确定的默认值。  
    unsigned char b; //定义的变量 b 被分配了一个字节的 RAM 空间，保存的数据是不确定的默认值。  
    unsigned char c; //定义的变量 c 被分配了一个字节的 RAM 空间，保存的数据是不确定的默认值。  
    unsigned char d=9; //定义的变量 d 被分配了一个字节的 RAM 空间，保存的数据被初始化成 9。  
  
    b=3; //把 3 赋值给变量 b，b 由原来不确定的默认数据变成了 3。  
    c=b; //把变量 b 的内容复制一份赋值给左边的变量 c，c 从不确定的默认值变成了 3。  
    View(a); //把第 1 个数 a 发送到电脑端的串口助手软件上观察。  
    View(b); //把第 2 个数 b 发送到电脑端的串口助手软件上观察。  
    View(c); //把第 3 个数 c 发送到电脑端的串口助手软件上观察。  
    View(d); //把第 4 个数 d 发送到电脑端的串口助手软件上观察。  
    while(1)  
    {  
    }
```

```
}
```

```
/*---C 语言学习区域的结束。-----*/
```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第 1 个数

十进制:255

十六进制:FF

二进制:11111111

第 2 个数

十进制:3

十六进制:3

二进制:11

第 3 个数

十进制:3

十六进制:3

二进制:11

第 4 个数

十进制:9

十六进制:9

二进制:1001

分析：

第 1 个数 a 居然是 255，这个 255 从哪来？因为 a 我们一直没有给它初始值，也没有给它赋值，所以它是不确定的默认值，这个 255 就是所谓的不确定的默认值，是编译器在定义变量 a 时分配的，带有不确定的随机性，不同的编译器可能分配的默认值都会存在差异。根据我的经验，unsigned char 类型定义的默认值往往是 0 或者 255（255 是十六进制的 0xff，十六进制的内容后续章节会讲到）。

## 第十三节：赋值语句的覆盖性。

### 【13.1 什么是赋值语句的覆盖性？】

```
a=b;
```

上述代码，执行完这条赋值语句后，会把右边变量 b 的数值复制一份给左边变量 a，a 获得了跟 b 一样的数值，但是 a 原来自己的数值却丢失了，为什么会丢失？就是因为被 b 复制过来的新数据给覆盖了，这就是赋值语句的覆盖性。

### 【13.2 例程的分析和练习。】

既然赋值语句有覆盖性的特点，那么如何让两个变量相互交换数值？假设 a 原来的数据是 1，b 原来的数据是 5，交换数据后，a 的数据应该变为 5，b 的数据应该变为 1，怎么做？很多初学者刚看到这么简单的题目，会马上根据日常生活的思路，你把你的东西给我，我把我的东西给你，就两个步骤而已，看似很简单，现在按这个思路编写一段程序看看会出什么问题，代码如下：

```
/*---C 语言学习区域的开始。-----*/

void main() //主函数
{
    unsigned char a=1;    //定义的变量 a 被分配了 1 个字节的 RAM 空间，保存的数据被初始化成 1。
    unsigned char b=5;    //定义的变量 b 被分配了 1 个字节的 RAM 空间，保存的数据被初始化成 5。

    b=a; //第一步：为了交换，先把 a 的数赋值给 b。
    a=b; //第二步：为了交换，再把 b 的数赋值给 a。

    View(a);    //把第 1 个数 a 发送到电脑端的串口助手软件上观察。
    View(b);    //把第 2 个数 b 发送到电脑端的串口助手软件上观察。

    while(1)
    {
    }
}

/*---C 语言学习区域的结束。-----*/
```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第 1 个数  
十进制:1

十六进制:1

二进制:1

第 2 个数

十进制:1

十六进制:1

二进制:1

分析:

第 1 个数 a 和第 2 个数 b 居然都是 1! 这不是我们想要的结果。我们要的交换结果是: 交换后, a 变为 5, b 变为 1。在哪个环节出了问题? 把镜头切换到上述代码的“第一步”和“第二步”, 由于 b 的数据在执行完“第一步”后, b 自己原来的数据 5 被覆盖丢失了变成新的数据 1, 接着执行“第二步”后, 此时相当于把 b 的新数据 1 赋值给 a, 并没有 5! 所以 a 和 b 的数据都是 1, 不能达到交换后“a 为 5, b 为 1”的目的。其实就是赋值语句的覆盖性在作祟。

上述交换数据的程序宣告失败! 怎么办? 既然赋值语句具有覆盖性, 那么两变量想交换数据, 就必须借助第三方变量来寄存, 此时只需要多定义一个第三方变量 t。正确的代码如下:

```
/*---C 语言学习区域的开始。-----*/
```

```
void main() //主函数
```

```
{
```

```
    unsigned char a=1;    //定义的变量 a 被分配了 1 个字节的 RAM 空间, 保存的数据被初始化成 1。
```

```
    unsigned char b=5;    //定义的变量 b 被分配了 1 个字节的 RAM 空间, 保存的数据被初始化成 5。
```

```
    unsigned char t;      //定义一个第三方变量 t, 用来临时寄存数值。
```

```
    t=b; //第一步: 为了避免 b 的数据在赋值后被覆盖丢失, 先寄存一份在第三方变量 t 那里。
```

```
    b=a; //第二步: 把 a 的数赋值给 b, b 原来的数据虽然丢失, 但是 b 在 t 变量那里有备份。
```

```
    a=t; //第三步: 再把 b 在 t 变量里的备份赋值给 a。注意, 这里不能用 b, 因 b 原数据已被覆盖。
```

```
    View(a);    //把第 1 个数 a 发送到电脑端的串口助手软件上观察。
```

```
    View(b);    //把第 2 个数 b 发送到电脑端的串口助手软件上观察。
```

```
    while(1)
```

```
    {
```

```
    }
```

```
}
```

```
/*---C 语言学习区域的结束。-----*/
```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第 1 个数

十进制:5

十六进制:5

二进制:101

第 2 个数

十进制:1

十六进制:1

二进制:1

分析：

实验结果显示，两变量的数值交换成功。

### 【13.3 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。

## 第十四节：二进制与字节单位，以及常用三种变量的取值范围。

### 【14.1 为什么要二进制？】

为什么要二进制？我们日常生活明明是十进制的，为何数字电子领域偏要选择二进制？这是由数字硬件电路决定的。人有十个手指头，人可以直接发出十种不同声音来命名 0, 1, 2, 3...9 这些数字，人可以直接用眼睛识别出十种不同状态的信息，但是数字底层基础硬件电路要直接处理和识别十种状态却很难，相对来说，处理和识别两种状态就轻松多了，所以选择二进制。比如，一颗 LED 灯的亮或灭，一个 IO 口的输出高电平或低电平，识别某一个点的电压是高电平或低电平，只需要三极管等基础元器件就可把硬件处理电路搭建起来，二进制广泛应用在数字电路的存储，通讯和运算等领域，想学好单片机就必须掌握它。

### 【14.2 二进制如何表示成千上万的大数值？】

二进制如何表示成千上万的数值？现在用 LED 灯的亮和灭来跟大家讲解。

(1) 1 个 LED 灯：

灭      第 0 种状态

亮      第 1 种状态

合计：共 2 种状态。

(2) 2 个 LED 灯挨着：

灭灭      第 0 种状态

灭亮      第 1 种状态

亮灭      第 2 种状态

亮亮      第 3 种状态

合计：共 4 种状态。

(3) 3 个 LED 灯挨着：

灭灭灭      第 0 种状态

灭灭亮      第 1 种状态

灭亮灭      第 2 种状态

灭亮亮      第 3 种状态

亮灭灭      第 4 种状态

亮灭亮      第 5 种状态

亮亮灭      第 6 种状态

亮亮亮      第 7 种状态

合计：共 8 种状态。

(4) 8 个 LED 灯挨着：

灭灭灭灭灭灭灭灭      第 0 种状态

灭灭灭灭灭灭灭亮      第 1 种状态

.....      第 N 种状态

亮亮亮亮亮亮亮灭      第 254 种状态

亮亮亮亮亮亮亮亮      第 255 种状态



合计：共 256 种状态。

(5) 16 个 LED 灯挨着：

灭灭灭灭灭灭灭灭灭灭灭灭灭灭灭灭灭灭	第 0 种状态
灭灭灭灭灭灭灭灭灭灭灭灭灭灭灭灭灭亮	第 1 种状态
.....	第 N 种状态
亮亮亮亮亮亮亮亮亮亮亮亮亮亮亮亮灭	第 65534 种状态
亮亮亮亮亮亮亮亮亮亮亮亮亮亮亮亮亮	第 65535 种状态
合计：共 65536 种状态。	

(6) 32 个 LED 灯挨着：

灭灭灭	第 0 种状态
灭灭亮	第 1 种状态
.....	第 N 种状态
亮亮灭	第 4294967294 种状态
亮亮亮	第 4294967295 种状态
合计：共 4294967296 种状态。	

结论：  
连续挨着的 LED 灯越多，能表达的数值范围就越大。

【14.3 什么是位？】

什么是位？以上一个 LED 灯就代表 1 位，8 个 LED 灯就代表 8 位。位的英文名是用 bit 来表示。一个变量的位数越大就意味着这个变量的取值范围越大。一个单片机的位数越大，就说明这个单片机一次处理的数据范围就越大，意味着运算和处理速度就越快。我们日常所说的 8 位单片机，32 位单片机，就是这个位的概念。为什么 32 位的单片机比 8 位单片机的处理和运算能力强，就是这个原因。

【14.4 什么是字节？】

什么是字节？字节是计算机很重要的一个基本单位，一个字节有 8 位。8 个 LED 灯挨着能代表多少种状态，就意味着一个字节的取值范围有多大。从上面举的例子中，我们知道 8 个 LED 灯挨着，能表示从 0 到 255 种状态，所以一个字节的取值范围就是从 0 到 255。

【14.5 三种常用变量的取值范围是什么？】

前面章节曾提到三种常用的变量：unsigned char, unsigned int , unsigned long。现在有了二进制和字节的基础知识，就可以跟大家讲讲这三种变量的取值范围，而且很重要，这是我们写单片机程序必备的概念。

- unsigned char 的变量占用 1 个字节 RAM，共 8 位，根据前面 LED 灯的例子，取值范围是从 0 到 255。
- unsigned int 的变量占用 2 个字节 RAM，共 16 位，根据前面 LED 灯的例子，取值范围是从 0 到 65535。多说一句，对于 51 内核的单片机，unsigned int 的变量是占用 2 个字节。如果是在 32 位的 stm32 单片机，unsigned int 的变量是占用 4 个字节的，所以不同的单片机不同的编译器是会有一些差异的。
- unsigned long 的变量占用 4 个字节 RAM，共 32 位，根据前面 LED 灯的例子，取值范围是从 0 到 4294967295。

## 【14.6 例程练习和分析。】

现在我们编写一个程序来验证 unsigned char, unsigned int, unsigned long 的取值范围。

定义两个 unsigned char 变量 a 和 b, a 赋值 255, b 赋值 256, 255 和 256 恰好处于 unsigned char 的取值边界。

再定义两个 unsigned int 变量 c 和 d, c 赋值 65535, d 赋值 65536, 65535 和 65536 恰好处于 unsigned int 的取值边界。

最后定义两个 unsigned long 变量 e 和 f, e 赋值 4294967295, f 赋值 4294967296, 4294967295 和 4294967296 恰好处于 unsigned long 的取值边界。

程序代码如下：

```
/*---C 语言学习区域的开始。-----*/

void main() //主函数
{
    unsigned char a;    //定义一个变量 a, 并且分配了 1 个字节的 RAM 空间。
    unsigned char b;    //定义一个变量 b, 并且分配了 1 个字节的 RAM 空间。
    unsigned int c;     //定义一个变量 c, 并且分配了 2 个字节的 RAM 空间。
    unsigned int d;     //定义一个变量 d, 并且分配了 2 个字节的 RAM 空间。
    unsigned long e;    //定义一个变量 e, 并且分配了 4 个字节的 RAM 空间。
    unsigned long f;    //定义一个变量 f, 并且分配了 4 个字节的 RAM 空间。

    a=255;              //把 255 赋值给变量 a, a 此时会是什么数? 会超范围溢出吗?
    b=256;              //把 256 赋值给变量 b, b 此时会是什么数? 会超范围溢出吗?
    c=65535;            //把 65535 赋值给变量 c, c 此时会是什么数? 会超范围溢出吗?
    d=65536;            //把 65536 赋值给变量 d, d 此时会是什么数? 会超范围溢出吗?
    e=4294967295;       //把 4294967295 赋值给变量 e, e 此时会是什么数? 会超范围溢出吗?
    f=4294967296;       //把 4294967296 赋值给变量 f, f 此时会是什么数? 会超范围溢出吗?

    View(a);            //把第 1 个数 a 发送到电脑端的串口助手软件上观察。
    View(b);            //把第 2 个数 b 发送到电脑端的串口助手软件上观察。
    View(c);            //把第 3 个数 c 发送到电脑端的串口助手软件上观察。
    View(d);            //把第 4 个数 d 发送到电脑端的串口助手软件上观察。
    View(e);            //把第 5 个数 e 发送到电脑端的串口助手软件上观察。
    View(f);            //把第 6 个数 f 发送到电脑端的串口助手软件上观察。

    while(1)
    {
    }
}

/*---C 语言学习区域的结束。-----*/
```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第 1 个数

十进制:255

十六进制:FF

二进制:11111111

第 2 个数

十进制:0

十六进制:0

二进制:0

第 3 个数

十进制:65535

十六进制:FFFF

二进制:1111111111111111

第 4 个数

十进制:0

十六进制:0

二进制:0

第 5 个数

十进制:4294967295

十六进制:FFFFFFFF

二进制:11111111111111111111111111111111

第 6 个数

十进制:0

十六进制:0

二进制:0

分析：

通过实验结果，我们知道 unsigned char 变量最大能取值到 255，如果非要赋值 256 就会超出范围溢出后变成了 0。unsigned int 变量最大能取值到 65535，如果非要赋值 65536 就会超出范围溢出后变成了 0。unsigned long 变量最大能取值到 4294967295，如果非要赋值 4294967296 就会超出范围溢出后变成了 0。

## 【14.7 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，

其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。

## 第十五节：二进制与十六进制。

### 【15.1 十六进制是二进制的缩写。】

在我的印象中，C51 编译器好像并不支持二进制的书写格式，即使它能支持二进制的书写格式，二进制的书写还是有个弊端，就是数字太多太长了，写起来非常费劲不方便，怎么办？解决办法就是用十六进制。十六进制是二进制的缩写，之所以称它为二进制的缩写，是因为它们的转换关系非常简单直观，不需要借助计算器即可相互转换。

### 【15.2 何谓十六进制？】

何谓十六进制？欲搞清楚这个问题，还得先从十进制说起。所谓十进制，就是用一位字符可以表示从 0 到 9 这十个数字。所谓二进制，就是用一位字符可以表示从 0 到 1 这二个数字。所谓十六进制，当然也就是用一位字符可以表示从 0 到 15 这十六个数字。但是十六进制马上就会面临一个问题，十六进制的 10 到 15 这 6 个数其实是有两位字符组成的，并不是一位呀？于是 C 语言用这些字符 A, B, C, D, E, F 分别替代 10, 11, 12, 13, 14, 15 这 6 个数，10 前面的 0 到 9 还是跟十进制的字符一致。A, B, C, D, E, F 也可以用小写 a, b, c, d, e, f 来替代，在数值上不区分大小写，比如十六进制的 a 与 A 都是表示十进制的 10。

### 【15.3 二进制与十六进制是如何转换的？】

前面提到了十六进制是二进制的缩写，它们的转换关系非常简单直观，每 1 位十六进制的字符，对应 4 位二进制的字符。关系如下：

十进制	二进制	十六进制
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

二进制转换成十六进制的时候，如果不是 4 位的倍数，则最左边高位默认补上 0 凑合成 4 位的倍数。比如一个二进制的数 101001，可以在左边补上 2 个 0 变成 00101001，然后把每 4 位字符转成 1 个十六进制的字符。左边高 4 位 0010 对应十六进制的 2，右边低 4 位 1001 对应十六进制的 9，所以二进制的 101001 合起

来最终转换成十六进制的数是 29（实际上正确的写法是 0x29，为什么？请继续往下看。）。

#### 【15.4 十六进制数的标准书写格式是什么样子的？】

十六进制的标准书写格式是什么样子的？实际上，十六进制 29 并不能直接写成 29，否则就跟十进制的写法混淆了。为了把十六进制和十进制的书写格式进行区分，C 语言规定凡是十六进制必须加一个数字 0 和一个字母 x 作为前缀，也就是十六进制必须以 0x 作为前缀，所以刚才的十六进制 29 就应该写成 0x29，否则，如果直接写 29 编译器会认为是十进制的 29，而十进制的 29 转换成十六进制是 0x1D（十进制与十六进制之间如何转换在后面章节会讲到），0x29 与 0x1D 可见差别很大的，凡是不加前缀的都会被默认为十进制。多说一句，在 C 语言程序里，对于同样一个数值，既可以用十六进制，也可以用十进制，比如：d=0x2C 与 d=44 的含义是一样的，因为十六进制的 0x2C 和十进制的 44 最终都会被 C51 编译器翻译成二进制 00101100，是表示同样大小的数值。

#### 【15.5 例程练习和分析。】

现在我们编写一个程序来观察十六进制和二进制的关系。

程序代码如下：

```
/*---C 语言学习区域的开始。-----*/

void main() //主函数
{
    unsigned char a;    //定义一个变量 a，并且分配了 1 个字节的 RAM 空间。
    unsigned char b;    //定义一个变量 b，并且分配了 1 个字节的 RAM 空间。
    unsigned char c;    //定义一个变量 c，并且分配了 1 个字节的 RAM 空间。
    unsigned char d;    //定义一个变量 d，并且分配了 1 个字节的 RAM 空间。

    a=0x06;    //十六进制前记得加 0x 前缀，超过 9 部分的字母不分大小写。
    b=0x0A;    //十六进制前记得加 0x 前缀，超过 9 部分的字母不分大小写。
    c=0x0e;    //十六进制前记得加 0x 前缀，超过 9 部分的字母不分大小写。
    d=0x2C;    //十六进制前记得加 0x 前缀，超过 9 部分的字母不分大小写。

    View(a);    //把第 1 个数 a 发送到电脑端的串口助手软件上观察。
    View(b);    //把第 2 个数 b 发送到电脑端的串口助手软件上观察。
    View(c);    //把第 3 个数 c 发送到电脑端的串口助手软件上观察。
    View(d);    //把第 4 个数 d 发送到电脑端的串口助手软件上观察。

    while(1)
    {
    }
}

/*---C 语言学习区域的结束。-----*/
```

在电脑串口助手软件上观察到的程序执行现象如下：

```
开始...

第 1 个数
十进制:6
十六进制:6
二进制:110

第 2 个数
十进制:10
十六进制:A
二进制:1010

第 3 个数
十进制:14
十六进制:E
二进制:1110

第 4 个数
十进制:44
十六进制:2C
二进制:101100
```

分析：

通过实验结果，我们知道二进制与十六进制的转换关系确实非常清晰简单，所以十六进制也可以看作是二进制的缩写。

## 【15.6 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。

## 第十六节：十进制与十六进制。

### 【16.1 十进制与十六进制各自的应用场合。】

C 语言程序里只用了十进制和十六进制这两种书写格式，有的初学者会问，为什么没有用二进制？我的回答是：不是没有用二进制，而是十六进制已经代表了二进制，因为十六进制就是二进制的缩写形式，所以可以把十六进制和二进制看作是同一个东西。

十进制和十六进制各自有什么应用场合？十六进制方便人们理解机器，通常应用在配置寄存器，底层通讯驱动，底层 I/O 口驱动，以及数据的移位、转换、合并等场合，在底层驱动程序方面经常要用到。而十进制则方便人们直观理解数值的大小，在程序应用层要经常用到。总之，进制只是数据的表现形式而已，不管是什么进制的数，最终经过编译后都可以看做是二进制的数。

### 【16.2 十进制与十六进制相互转换的方法。】

十进制与十六进制如何相互转换？其实很多教科书上都有介绍它们之间如何通过手工计算进行转换的方法，这种方法当然是有助于我们深入理解数据的含义和转换关系，有兴趣的朋友可以自己找相关书籍来看看，但是在实际应用中，我本人是从来没有用过这种手工计算方法，而我用的方法是最简单直接的，就是借助电脑自带的计算器进行数制转换即可。现在把这种方法介绍给大家，以 WIN7 系统的电脑为例来讲解详细的操作步骤。

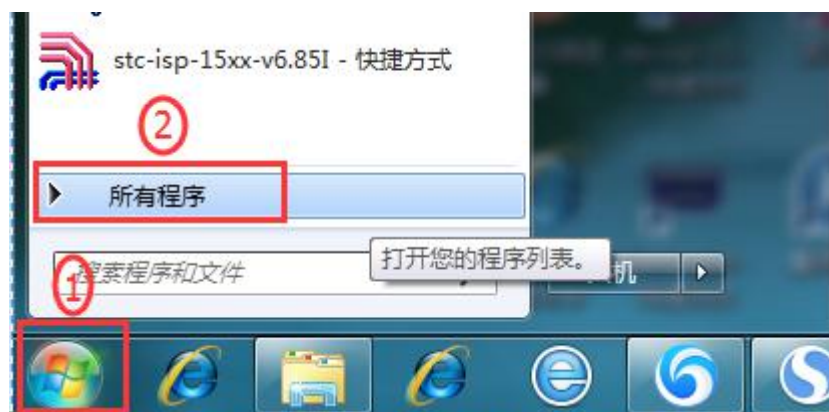


图 16.2.1.1 点击“所有程序”选项切换到系统自带程序的窗口



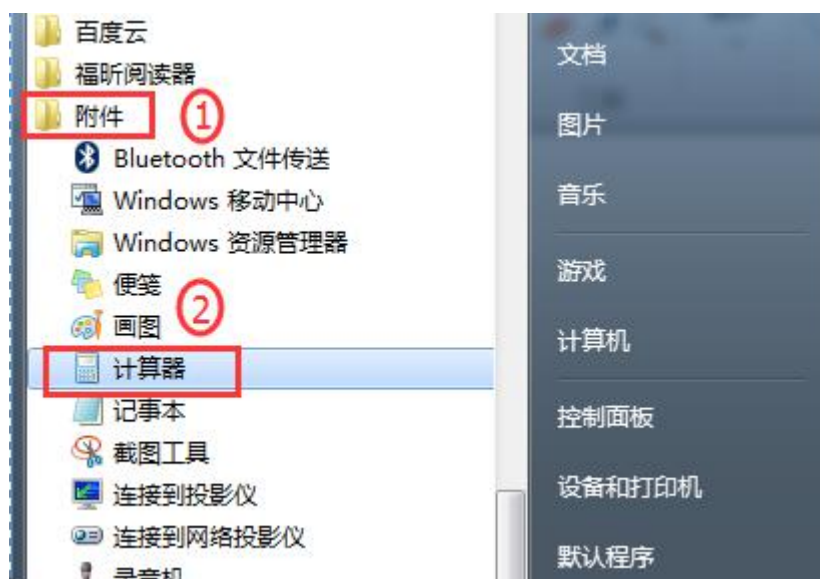


图 16.2.1.2 在“附件”子菜单下点击“计算器”启动此软件



图 16.2.1.3 已启动的“计算器”软件界面

第一步：打开电脑自带的计算器。

点击电脑左下角“开始”菜单，在菜单中点击“所有程序”选项切换到自带程序的窗口，在此窗口下，再点击“附件”的文件夹图标，在“附件”子菜单下点击“计算器”启动此软件。

-----步骤之间的分割线-----



图 16.2.2.1 把“计算器”的主界面切换到“程序员”界面



图 16.2.2.2 已打开的“程序员”界面

第二步：把“计算器”的主界面切换到“程序员”界面。

点击打开左上角“查看”的下拉菜单，在下拉菜单中选择“程序员”选项。

-----步骤之间的分割线-----



图 16.2.3.1 在十进制的选项下输入十进制的数据



图 16.2.3.2 把十进制的数据转换成十六进制的数据

第三步：十进制转换成十六进制的方法。

点击勾选中“十进制”选项，在此选项下输入十进制的数据，输入数据后，再切换点击勾选“十六进制”，

即可完成从十进制到十六进制的数据转换。比如输入十进制的“230”，切换到十六进制后就变成了“E6”。

-----步骤之间的分割线-----



图 16.2.4.1 在十六进制的选项下输入十六进制的数据



图 16.2.4.2 把十六进制的数据转换成十进制的数据

第四步：十六进制转换成十进制的方法。

点击勾选中“十六进制”选项，在此选项下输入十六进制的数据，输入数据后，再切换点击勾选“十进制”，即可完成从十六进制到十进制的数据转换。比如输入十六进制的“AC”，切换到十进制后就变成了“172”。

-----步骤之间的分割线-----

第五步：十六进制，十进制，八进制，二进制它们四者之间相互转换的方法。

我们看到“计算器”软件里已经包含了十六进制，十进制，八进制，二进制这四个选项，所以它们之间相互转换的方法跟上面介绍的步骤是一样的。

-----步骤之间的分割线-----

### 【16.3 例程练习和分析。】

现在我们编写一个程序来验证上面讲到的两个例子：

- (1) 输入十进制的 230，看看它的十六进制是什么样的。
- (2) 输入十六进制的 AC，看看它的十进制是什么样的。

程序代码如下：

```
/*---C 语言学习区域的开始。-----*/

void main() //主函数
{
    unsigned char a;    //定义一个变量 a，并且分配了 1 个字节的 RAM 空间。
    unsigned char b;    //定义一个变量 b，并且分配了 1 个字节的 RAM 空间。

    a=230;    //把十进制的 230 赋值给变量 a，在串口助手上观察一下它的十六进制是不是 E6。
    b=0xAC;   //把十六进制的 AC 赋值给变量 b，在串口助手上观察一下它的十进制是不是 172。
    View(a);  //把第 1 个数 a 发送到电脑端的串口助手软件上观察。
    View(b);  //把第 2 个数 b 发送到电脑端的串口助手软件上观察。

    while(1)
    {
    }
}

/*---C 语言学习区域的结束。-----*/
```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

```
第 1 个数
十进制:230
十六进制:E6
二进制:11100110
```

```
第 2 个数
十进制:172
十六进制:AC
二进制:10101100
```

分析：

通过实验结果，发现在单片机上转换的结果和在电脑自带“计算器”上转换的结果是一样的。

#### 【16.4 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。

## 第十七节：加法运算的 5 种常用组合。

### 【17.1 单片机本身具备基础的数学算术能力。】

单片机本身是一个成品，本身就具备了基础的加减乘除能力，把单片机当做一个大人，我们需要做的只是沟通而已，叫他做加法他就做加法，叫他做减法他就做减法，至于他是怎么计算出来的不用管，“他”本身内部的电路结构就具备了这种基础运算的能力。人机沟通依然是用 C 语言，本节讲的加法运算，用的 C 语言符号跟我们日常用的数学加法符号是一样的，都是符号“+”。多说一句，单片机这种内置的基础运算能力并不是无限大的，而是数值不能超过某个范围，如果在加数或者运算结果的数值范围超过 4294967295 的情况下，要继续实现这类加法运算，这个就需要我们在单片机本身基础的运算能力上专门去编写一套大数据算法的程序才能实现，这个大家暂时不用深入理解，先学好当前基础再说。

### 【17.2 加法语法格式。】

加法语法格式：

“保存变量” = “加数 1” + “加数 2” + ... + “加数 N” ;

含义：右边的“加数”与“加数”相加（这里暂时把平时所说的被加数也归类为加数），并且把最终的运算结果赋值给左边的“保存变量”。注意，这里的符号“=”不是等于号的意思，而是赋值的意思。左边的“保存变量”必须是变量，不能是常量，否则编译时会报错。而右边的“加数”既可以是变量，也可以是常量，也可以是“保存变量”本身自己。多说一句，什么是变量和什么是常量？变量就是可以在程序中被更改的，是分配的一个 RAM 空间。而常量往往就是常数值，或者是被分配在 ROM 空间的一个具体数值。下面根据右边“加数”与“加数”的不同组合，列出了加法运算的 5 种常用组合。

第 1 种：“加数 1”是常量，“加数 2”是常量。比如：

```
unsigned char a;  
a=3+15;
```

分析：数字“3”和“15”都是常量。执行上述语句后，保存变量 a 变成了 18。

第 2 种：“加数 1”是变量，“加数 2”是常量。比如：

```
unsigned char b;  
unsigned char x=10;  
b=x+15;
```

分析：x 是变量，“15”是常量。由于原来 x 变量里面的数值是 10，执行上述语句后，保存变量 b 变成了 25。而变量 x 则保持不变，执行完所有语句后 x 还是 10。

第 3 种：“加数 1”是变量，“加数 2”是变量。比如：

```
unsigned char c;  
unsigned char x=10;  
unsigned char y=6;  
c=x+y;
```

分析：x 是变量，y 也是变量。由于原来 x 变量里面的数值是 10，y 变量里面的数值是 6，执行上述语句后，保存变量 c 变成了 16。而变量 x 和 y 则保持不变，x 还是 10，y 还是 6。

第 4 种：“加数 1”是保存变量本身，“加数 2”是常量。比如：

```
unsigned char d=2;
d=d+18;
d=d+7;
```

分析：d 是保存变量本身，“18”是常量。这类语句有一个特点，具备了自加功能，可以更改自己本身的数值。比如原来保存变量 d 的数值是 2，执行“d=d+18;”语句后，d 变成了 20，接着再执行完“d=d+7;”语句后，d 最后变成了 27。

第 5 种：“加数 1”是保存变量本身，“加数 2”是变量。比如：

```
unsigned char e=2;
unsigned char x=10;
unsigned char y=6;
e=e+x;
e=e+y;
```

分析：e 是保存变量，x 与 y 都是变量。这类语句有一个特点，具备了自加功能，可以更改自己本身的数值。比如原来保存变量 e 的数值是 2，x 的数值是 10，执行“e=e+x;”语句后，e 变成了 12。由于 y 的数值是 6，接着再执行完“e=e+y;”语句后，所以 e 最后变成了 18。

### 【17.3 例程练习和分析。】

现在我们编写一个程序来验证上面讲到的 5 个加法例子：

程序代码如下：

```
/*---C 语言学习区域的开始。-----*/

void main() //主函数
{
    unsigned char a;    //定义一个变量 a，并且分配了 1 个字节的 RAM 空间。
    unsigned char b;    //定义一个变量 b，并且分配了 1 个字节的 RAM 空间。
    unsigned char c;    //定义一个变量 c，并且分配了 1 个字节的 RAM 空间。
    unsigned char d=2;  //定义一个变量 d，并且分配了 1 个字节的 RAM 空间。初始化默认为 2.
    unsigned char e=2;  //定义一个变量 e，并且分配了 1 个字节的 RAM 空间。初始化默认为 2.

    unsigned char x=10; //定义一个变量 x，并且分配了 1 个字节的 RAM 空间。初始化默认为 10.
    unsigned char y=6;  //定义一个变量 y，并且分配了 1 个字节的 RAM 空间。初始化默认为 6.

    //第 1 种：“加数 1”是常量，“加数 2”是常量。
    a=3+15;

    //第 2 种：“加数 1”是变量，“加数 2”是常量。
    b=x+15;

    //第 3 种：“加数 1”是变量，“加数 2”是变量。
    c=x+y;
```



```

//第4种：“加数1”是保存变量本身，“加数2”是常量。
d=d+18;
d=d+7;

//第5种：“加数1”是保存变量本身，“加数2”是变量。
e=e+x;
e=e+y;

View(a); //把第1个数a发送到电脑端的串口助手软件上观察。
View(b); //把第2个数b发送到电脑端的串口助手软件上观察。
View(c); //把第3个数c发送到电脑端的串口助手软件上观察。
View(d); //把第4个数d发送到电脑端的串口助手软件上观察。
View(e); //把第5个数e发送到电脑端的串口助手软件上观察。

while(1)
{
}
}

/*---C语言学习区域的结束。-----*/

```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第1个数

十进制:18

十六进制:12

二进制:10010

第2个数

十进制:25

十六进制:19

二进制:11001

第3个数

十进制:16

十六进制:10

二进制:10000

第4个数

十进制:27

十六进制:1B

二进制:11011

第 5 个数

十进制:18

十六进制:12

二进制:10010

分析:

通过实验结果，发现在单片机上的计算结果和我们的分析是一致的。

#### 【17.4 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。

## 第十八节：连加、自加、自加简写、自加 1。

### 【18.1 连加。】

上一节的加法例子中，右边的加数只有两个。实际上，C 语言规则没有限制加数的个数，它的通用格式如下：

“保存变量” = “加数 1” + “加数 2” + ... + “加数 N”；

当右边的加数个数超过两个的时候（这里暂时把平时所说的被加数也归类为加数），这种情况就是“连加”，每个加数的属性没有限定，可以是常量，也可以是变量。比如：

a=1+69+102;     //加数全部是常量。

b=q+x+y+k+r;     //加数全部是变量。

c=3+x+y+5+k;     //加数有的是常量，有的是变量。

连加的运行顺序是，赋值符号“=”右边的加数挨个相加，把每一次的运算结果放在一个临时的隐藏变量里，这个隐藏的变量我们看不到，是单片机系统内部参与运算时的专用寄存器，等右边所有的加数连加的计算结果出来后，再把这个隐藏变量所保存的计算结果赋值给左边的“保存变量”。

### 【18.2 自加、自加简写、自加 1。】

什么是自加？当赋值符号“=”右边的加数只要其中有一个是“保存变量”本身时，这种情况就是“自加”，自加在程序里有一个特点，只要加数不为 0，那么每执行一次这行代码，“保存变量”本身就会增大一次，不断执行这行代码，“保存变量”本身就会不断增大，而每次的增大量就取决于赋值符号“=”右边从第 2 个加数开始后面所有加数之和。自加的常见格式如下：

“保存变量” = “保存变量” + “加数 2”；

“保存变量” = “保存变量” + (“加数 2” + “加数 3” + ... + “加数 N”)；

在这类自加计算式中，当右边的加数有且仅有一个是“保存变量”本身时，那么上述自加计算式可以简写成如下格式：

“保存变量” += “加数 2”；

“保存变量” += “加数 2” + “加数 3” + ... + “加数 N”；

这种格式就是“自加简写”。现在举几个例子如下：

d+=6;     //相当于 d=d+6;

e+=x;     //相当于 e=e+x;

f+=18+y+k; //相当于 f=f+(18+y+k);

这些例子都是很常规的自加简写，再跟大家讲一种很常用的特殊简写。当右边只有两个加数，当一个加数是“保存变量”，另一个加数是常数 1 时，格式如下：

“保存变量” = “保存变量” + 1;

这时候，可以把上述格式简写成如下两种格式：

“保存变量” ++;

++ “保存变量”；

这两种格式也是俗称的“自加 1”操作。比如：

g++;     //相当于 g=g+1 或者 g+=1;

++h;     //相当于 h=h+1 或者 h+=1;

也就是说自加 1 符号“++”可以在变量的左边，也可以在变量的右边，它们在这里本质是一样的，没有差别，但是，如果是在某些特定情况下，这时自加 1 符号“++”在左边还是在右边是有差别的，有什么差别呢？这个内容以后再讲。

### 【18.3 例程练习和分析。】

现在我们编写一个程序来验证上面讲到的例子：

程序代码如下：

```
/*---C 语言学习区域的开始。-----*/

void main() //主函数
{
    unsigned char a;    //定义一个变量 a，并且分配了 1 个字节的 RAM 空间。
    unsigned char b;    //定义一个变量 b，并且分配了 1 个字节的 RAM 空间。
    unsigned char c;    //定义一个变量 c，并且分配了 1 个字节的 RAM 空间。
    unsigned char d=5;  //定义一个变量 d，并且分配了 1 个字节的 RAM 空间。初始化默认为 5.
    unsigned char e=5;  //定义一个变量 e，并且分配了 1 个字节的 RAM 空间。初始化默认为 5.
    unsigned char f=5;  //定义一个变量 f，并且分配了 1 个字节的 RAM 空间。初始化默认为 5.
    unsigned char g=5;  //定义一个变量 g，并且分配了 1 个字节的 RAM 空间。初始化默认为 5.
    unsigned char h=5;  //定义一个变量 h，并且分配了 1 个字节的 RAM 空间。初始化默认为 5.

    unsigned char q=1;  //定义一个变量 q，并且分配了 1 个字节的 RAM 空间。初始化默认为 1.
    unsigned char x=3;  //定义一个变量 x，并且分配了 1 个字节的 RAM 空间。初始化默认为 3.
    unsigned char y=6;  //定义一个变量 y，并且分配了 1 个字节的 RAM 空间。初始化默认为 6.
    unsigned char k=2;  //定义一个变量 k，并且分配了 1 个字节的 RAM 空间。初始化默认为 2.
    unsigned char r=8;  //定义一个变量 r，并且分配了 1 个字节的 RAM 空间。初始化默认为 8.

    //第 1 个知识点：连加。
    a=1+69+102;    //加数全部是常量。a 的结果为：172。
    b=q+x+y+k+r;    //加数全部是变量。b 的结果为：20。
    c=3+x+y+5+k;    //加数有的是常量，有的是变量。c 的结果为：19。

    //第 2 个知识点：自加。
    d+=6; //相当于 d=d+6; d 的结果为：11。
    e+=x; //相当于 e=e+x; e 的结果为：8。
    f+=18+y+k; //相当于 f=f+18+y+k; f 的结果为：31。

    //第 3 个知识点：自加 1。
    g++; //相当于 g=g+1 或者 g+=1; g 的结果为：6。
    ++h; //相当于 h=h+1 或者 h+=1; h 的结果为：6。

    View(a); //把第 1 个数 a 发送到电脑端的串口助手软件上观察。
    View(b); //把第 2 个数 b 发送到电脑端的串口助手软件上观察。
    View(c); //把第 3 个数 c 发送到电脑端的串口助手软件上观察。
    View(d); //把第 4 个数 d 发送到电脑端的串口助手软件上观察。
    View(e); //把第 5 个数 e 发送到电脑端的串口助手软件上观察。
```

```

    View(f);    //把第 6 个数 f 发送到电脑端的串口助手软件上观察。
    View(g);    //把第 7 个数 g 发送到电脑端的串口助手软件上观察。
    View(h);    //把第 8 个数 h 发送到电脑端的串口助手软件上观察。

    while(1)
    {
    }
}

/*---C 语言学习区域的结束。-----*/

```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第 1 个数

十进制:172

十六进制:AC

二进制:10101100

第 2 个数

十进制:20

十六进制:14

二进制:10100

第 3 个数

十进制:19

十六进制:13

二进制:10011

第 4 个数

十进制:11

十六进制:B

二进制:1011

第 5 个数

十进制:8

十六进制:8

二进制:1000

第 6 个数

十进制:31

十六进制:1F

二进制:11111

```
第 7 个数
十进制:6
十六进制:6
二进制:110
```

```
第 8 个数
十进制:6
十六进制:6
二进制:110
```

分析：

通过实验结果，发现在单片机上的计算结果和我们的分析是一致的。

#### 【18.4 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。

## 第十九节：加法运算的溢出。

### 【19.1 什么是加法运算的溢出？】

前面章节介绍的三种数据类型 `unsigned char` , `unsigned int` , `unsigned long`, 它们的数值都是有最大范围的, 分别是 255, 65535, 4294967295。如果运算结果超过了变量本身的最大范围, 会出现什么结果、有什么规律, 这就是本节要讲的溢出。

(1) 什么是溢出? 先看一个例子如下:

```
unsigned char a;  
a=0x8536;
```

分析:

因为 `a` 是 `unsigned char` 变量, 位数是 8 位, 也就是一个字节, 而 `0x8536` 是 16 位, 两个字节, 这种情况下, 把两字节的 `0x8536` 强行赋值给单字节变量 `a`, 变量 `a` 只能接收到最低 8 位的一个字节 `0x36`, 而高 8 位的一个字节 `0x85` 就被丢失了, 这个就是本节所说的溢出。

(2) 再看一个例子如下:

```
unsigned char b=0xff;  
b=b+1;
```

分析:

`b` 默认值是 `0xff`, 再加 1 后, 变成了 `0x0100` 保存在一个隐藏的中间变量, 然后再把这个中间变量赋值给单字节变量 `b`, `b` 只能接收到低 8 位的一个字节 `0x00`, 所以运算后 `b` 的数值由于溢出变成了 `0x00`。

(3) 再看一个例子如下:

```
unsigned char c=0xff;  
c=c+2;
```

分析:

`c` 默认值是 `0xff`, 再加 2 后, 变成了 `0x0101` 保存在一个隐藏中间变量, 然后再把这个中间变量赋值给单字节变量 `c`, `c` 只能接收到低 8 位的一个字节 `0x01`, 所以运算后 `c` 的数值由于溢出变成了 `0x01`。

(4) 再看一个例子如下:

```
unsigned int d=0xffff;  
d=d+5;
```

分析:

`d` 默认值是 `0xffff`, 再加 5 后, 变成了 `0x10003` 保存在一个隐藏中间变量, 由于这个隐藏的中间变量是 `unsigned int` 类型, 只能保存两个字节的的数据, 所以在中间变量这个环节就溢出了, 实际上隐藏的中间变量只保存了 `0x0003`, 然后再把这个中间变量赋值给 16 位的两字节变量 `d`, `d` 理所当然就是 `0x0003`。

(5) 再看一个例子如下:

```
unsigned long e=0xffffffff;  
e=e+5;
```

分析:

`e` 默认值是 `0xffffffff`, 再加 5 后, 变成了 `0x100000003` 保存在一个隐藏中间变量, 由于这个隐藏的中间变量是 `unsigned long` 类型, 只能保存四个字节的的数据, 所以在中间变量这个环节就溢出了, 实际上隐藏的中间变量只保存了 `0x00000003`, 然后再把这个中间变量赋值给 32 位的四字节变量 `e`, `e` 理所当然也是 `0x00000003`。

### 【19.2 例程练习和分析。】

现在我们编写一个程序来验证上面讲到的例子：  
程序代码如下：

```
/*---C 语言学习区域的开始。-----*/

void main() //主函数
{
    unsigned char a;          //一个字节
    unsigned char b=0xff;     //一个字节
    unsigned char c=0xff;     //一个字节
    unsigned int  d=0xfffe;    //两个字节
    unsigned long e=0xffffffff; //四个字节

    a=0x8536;
    b=b+1;
    c=c+2;
    d=d+5;
    e=e+5;

    View(a);  //把第 1 个数 a 发送到电脑端的串口助手软件上观察。
    View(b);  //把第 2 个数 b 发送到电脑端的串口助手软件上观察。
    View(c);  //把第 3 个数 c 发送到电脑端的串口助手软件上观察。
    View(d);  //把第 4 个数 d 发送到电脑端的串口助手软件上观察。
    View(e);  //把第 5 个数 e 发送到电脑端的串口助手软件上观察。

    while(1)
    {
    }
}

/*---C 语言学习区域的结束。-----*/
```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第 1 个数

十进制:54

十六进制:36

二进制:110110

第 2 个数

十进制:0

十六进制:0



二进制:0

第 3 个数

十进制:1

十六进制:1

二进制:1

第 4 个数

十进制:3

十六进制:3

二进制:11

第 5 个数

十进制:3

十六进制:3

二进制:11

分析:

通过实验结果，发现在单片机上的计算结果和我们的分析是一致的。

### 【19.3 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。

## 第二十章：隐藏中间变量为何物？

### 【20.1 隐藏中间变量为何物？】

“隐藏中间变量”虽然视之不见摸之不着，但是像空气一样无处不在。它有什么规律，是什么类型，数值范围是多大，研究它有什么实用价值？这就是本节要解开之谜。

前面章节提到，两个加数相加，其结果暂时先保存在一个“隐藏中间变量”里，运算结束后才把这个“隐藏中间变量”赋值给左边的“保存变量”。这里的“隐藏中间变量”到底是 unsigned int 类型还是 unsigned long 类型？为了研究它的规律，我在 keil 自带的 C51 编译环境下，专门编写了几个测试程序来观察实际运行的结果。

“保存变量” = “加数 1” + “加数 2”；

下面分别变换“保存变量”、“加数 1”、“加数 2”这三个元素的数据类型，来观察“隐藏中间变量”背后的秘密。

(1) “unsigned int” = “unsigned char” + “unsigned char”；

```
unsigned int a;
unsigned char x1=0x12;
unsigned char y1=0xfe;
a=x1+y1;
```

运算结果：a 等于 0x0110。

分析过程：两个 char 类型的数相加其运算结果暂时保存在“隐藏中间变量”，当运算结果大于两个“加数” unsigned char 本身时，并没有发生溢出现象，unsigned int 类型的“保存变量” a 最终得到了完整的结果 0x0110。

初步结论：这种情况，“隐藏中间变量”估计为 unsigned int 类型。

(2) “unsigned long” = “unsigned int” + “unsigned char”；

```
unsigned long b;
unsigned int x2=0xfffe;
unsigned char y2=0x12;
b=x2+y2;
```

运算结果：b 等于十六进制的 0x0010。

分析过程：一个 unsigned int 类型的数与一个 unsigned char 类型的数相加，当运算结果大于其中最大加数 unsigned int 类型本身时，因为左边的“保存变量”本来就是 unsigned long 类型，所以我本来以为运算结果应该是 unsigned long 类型的 0x00010010，但是实际结果出乎我的意料，最终结果是 unsigned int 类型的 0x0010，显然发生了溢出现象。

初步结论：这种情况，“隐藏中间变量”估计为 unsigned int 类型。

(3) “unsigned long” = “常量” + “常量”；

```
unsigned long c;
c=50000+50000;
```

运算结果：c 等于 100000。

分析过程：unsigned int 的最大数据范围是 65535，而两个常量相加，其结果超过了 65535 却还能完整保存下来。

初步结论：这种右边加数都是常量的情况下，“隐藏中间变量”估计等于左边的“保存变量”类型。

(4) “unsigned long” = “unsigned int” + “常量”;

```
unsigned long d;  
unsigned long e;  
unsigned int x3=50000;  
d=x3+30000;  
e=x3+50000;
```

运算结果: d 等于 14464, e 等于 100000。

分析过程:本来以为 d 应该等于 80000 的,结果却是 14464 显然发生了溢出。而同样的条件下,e 是 100000 却没有发生溢出。

**个人结论:**这个现象让我突然崩溃,实在研究不下去了。这是一种很怪异的现象,为什么同样的类型,因为常量的不同,一个发生了溢出,另外一个没有发生溢出?这时的“隐藏中间变量”到底是 unsigned int 类型还是 unsigned long 类型?我无法下结论。经过上述简单的测试,我发现规律是模糊的,模糊的规律就不能成为规律。如果真按这种思路研究下去,那真是没完没了,因为还有很多情况要研究,当超过 3 个以上加数相加,同时存在 unsigned long,unsigned int,unsigned char,以及“常量”这 4 种类型时又是什么规律?在不同的 C 编译器里又会是什么现象?即使把所有情况的规律摸清楚了又能怎么样,因为那么繁杂很容易忘记导致出错。有什么解决的办法吗?

## 【20.2 解决办法。】

“当遇到有争议的问题时,与其参与争议越陷越深,还不如想办法及时抽身绕开争议。”根据这个指导思想,我提出一种解决思路**“为了避免出现意想不到的溢出,在实际项目中,所有参与运算的变量都预先转化为 unsigned long 变量,再参与运算。”**

当然,也可能有人会问,如果计算结果超过了 unsigned long 最大范围时怎么办?我的回答是:首先,大多数项目的计算量都比较简单,一般情况下都不会超过 unsigned long 最大范围,但是,如果真遇到有可能超过 unsigned long 最大范围的运算项目时,那么就要用另外一种 BCD 码数组的运算算法来解决,而这个方法本节暂时不介绍,等以后再讲。

继续回到刚才的话题,“为了避免出现意想不到的溢出,在实际项目中,所有参与运算的变量都预先转化为 unsigned long 变量,再参与运算。”如何把所有的运算变量都转化为 unsigned long 变量?现在介绍一下这个方法。

第一个例子:比如上述第(2)个例子,其转换方法如下:

```
unsigned long f;  
unsigned int x2=0xfffe;  
unsigned char y2=0x12;  
unsigned long t; //多增加一个 long 类型的变量,用来变换类型。  
unsigned long r; //多增加一个 long 类型的变量,用来变换类型。  
t=0; //把变量的高位和低位全部清零。  
t=x2; //把 x2 的数值先放到一个 long 类型的变量里,让”加数”跟”保存变量”类型一致。  
r=0; //把变量的高位和低位全部清零。  
r=y2; //把 y2 的数值先放到一个 long 类型的变量里,让”加数”跟”保存变量”类型一致。  
f=t+r;
```

运算结果: f 等于十六进制的 0x00010010,没有发生溢出现象。

第二个例子:比如上述第(4)个例子,其转换方法如下:

```

unsigned long g;
unsigned long h;
unsigned int x3=50000;
unsigned long t; //多增加一个 long 类型的变量，用来变换类型
t=0; //把变量的高位和低位全部清零。
t=x3; //把 x3 的数值先放到一个 long 类型的变量里，让”加数”跟”保存变量”类型一致。
g=t+30000;
h=t+50000;

```

运算结果：g 等于 80000, h 等于 100000。都没有发生溢出。

### 【20.3 例程练习和分析。】

现在我们编写一个程序来验证上面讲到的例子：

程序代码如下：

```

/*---C 语言学习区域的开始。-----*/

void main() //主函数
{
    unsigned int a; //第（1）个例子
    unsigned char x1=0x12;
    unsigned char y1=0xfe;

    unsigned long b; //第（2）个例子
    unsigned int x2=0xfffe;
    unsigned char y2=0x12;

    unsigned long c; //第（3）个例子

    unsigned long d; //第（4）个例子
    unsigned long e;
    unsigned int x3=50000;

    unsigned long f; //第（2）个例子改进之后

    unsigned long g; //第（4）个例子改进之后
    unsigned long h;

    unsigned long t; //多增加一个 long 类型的变量，用来变换类型。
    unsigned long r; //多增加一个 long 类型的变量，用来变换类型。

    //第（1）个例子
    a=x1+y1;

```

```

//第（2）个例子
b=x2+y2;

//第（3）个例子
c=50000+50000;

//第（4）个例子
d=x3+30000;
e=x3+50000;

//第（2）个例子改进之后
t=0;    //把变量的高位和低位全部清零。
t=x2;   //把 x2 的数值先放到一个 long 类型的变量里，让”加数”跟”保存变量”类型一致。
r=0;    //把变量的高位和低位全部清零。
r=y2;   //把 y2 的数值先放到一个 long 类型的变量里，让”加数”跟”保存变量”类型一致。
f=t+r;

//第（4）个例子改进之后
t=0;    //把变量的高位和低位全部清零。
t=x3;   //把 x3 的数值先放到一个 long 类型的变量里，让”加数”跟”保存变量”类型一致。
g=t+30000;
h=t+50000;

View(a); //把第 1 个数 a 发送到电脑端的串口助手软件上观察。
View(b); //把第 2 个数 b 发送到电脑端的串口助手软件上观察。
View(c); //把第 3 个数 c 发送到电脑端的串口助手软件上观察。
View(d); //把第 4 个数 d 发送到电脑端的串口助手软件上观察。
View(e); //把第 5 个数 e 发送到电脑端的串口助手软件上观察。
View(f); //把第 6 个数 f 发送到电脑端的串口助手软件上观察。
View(g); //把第 7 个数 g 发送到电脑端的串口助手软件上观察。
View(h); //把第 8 个数 h 发送到电脑端的串口助手软件上观察。

while(1)
{
}
}

/*---C 语言学习区域的结束。-----*/

```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第 1 个数

十进制:272

十六进制:110

二进制:100010000

第 2 个数

十进制:16

十六进制:10

二进制:10000

第 3 个数

十进制:100000

十六进制:186A0

二进制:11000011010100000

第 4 个数

十进制:14464

十六进制:3880

二进制:11100010000000

第 5 个数

十进制:100000

十六进制:186A0

二进制:11000011010100000

第 6 个数

十进制:65552

十六进制:10010

二进制:10000000000010000

第 7 个数

十进制:80000

十六进制:13880

二进制:10011100010000000

第 8 个数

十进制:100000

十六进制:186A0

二进制:11000011010100000

分析:

通过实验结果,发现在单片机上的计算结果和我们的分析是一致的。

#### 【20.4 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。

## 第二十一节：减法运算的 5 种常用组合。

### 【21.1 减法语法格式。】

减法语法格式：

“保存变量” = “减数 1” - “减数 2” - ... - “减数 N”；

含义：右边的“减数”与“减数”相减（这里暂时把平时所说的被减数也归类为减数），并且把最终的运算结果赋值给左边的“保存变量”。注意，这里的符号“=”不是等于号的意思，而是赋值的意思。左边的“保存变量”必须是变量，不能是常量，否则编译时会报错。右边的“减数”既可以是变量，也可以是常量，也可以是“保存变量”本身自己。多说一句，什么是变量和常量？变量是可以在程序中被更改的，被分配的一个 RAM 空间。常量往往是数字，或者被分配在 ROM 空间的一个具体数值。下面根据右边“减数”与“减数”的不同组合，列出了减法运算的 5 种常用组合。

第 1 种：“减数 1”是常量，“减数 2”是常量。比如：

```
unsigned char a;  
a=15-3;
```

分析：数字“15”和“3”都是常量。执行上述语句后，保存变量 a 变成了 12。

第 2 种：“减数 1”是变量，“减数 2”是常量。比如：

```
unsigned char b;  
unsigned char x=15;  
b=x-10;
```

分析：x 是变量，“10”是常量。由于原来 x 变量里面的数值是 15，执行上述语句后，保存变量 b 变成了 5。而变量 x 则保持不变，x 还是 15。

第 3 种：“减数 1”是变量，“减数 2”是变量。比如：

```
unsigned char c;  
unsigned char x=15;  
unsigned char y=6;  
c=x-y;
```

分析：x 是变量，y 也是变量。由于原来 x 变量里面的数值是 15，y 变量里面的数值是 6，执行上述语句后，保存变量 c 变成了 9。而变量 x 和 y 则保持不变，x 还是 15，y 还是 6。

第 4 种：“减数 1”是保存变量本身，“减数 2”是常量。比如：

```
unsigned char d=18;  
d=d-2;  
d=d-7;
```

分析：d 是保存变量，“2”和“7”都是常量。这类语句有一个特点，具备了自减功能，可以更改自己本身的数值。比如原来保存变量 d 的数值是 18，执行“d=d-2;”语句后，d 变成了 16，接着再执行完“d=d-7;”语句后，d 最后变成了 9。

第 5 种：“减数 1”是保存变量本身，“减数 2”是变量。比如：

```
unsigned char e=28;  
unsigned char x=15;
```



```
unsigned char y=6;
e=e-x;
e=e-y;
```

分析：e 是保存变量，x 与 y 都是变量。这类语句有一个特点，具备了自减功能，可以更改自己本身的数值。比如原来保存变量 e 的数值是 28，执行“e=e-x;”语句后，e 变成了 13，接着再执行完“e=e-y;”语句后，e 最后变成了 7。

## 【21.2 例程练习和分析。】

现在我们编写一个程序来验证上面讲到的 5 个减法例子：

程序代码如下：

```
/*---C 语言学习区域的开始。-----*/

void main() //主函数
{
    unsigned char a;    //定义一个变量 a，并且分配了 1 个字节的 RAM 空间。
    unsigned char b;    //定义一个变量 b，并且分配了 1 个字节的 RAM 空间。
    unsigned char c;    //定义一个变量 c，并且分配了 1 个字节的 RAM 空间。
    unsigned char d=18; //定义一个变量 d，并且分配了 1 个字节的 RAM 空间。初始化默认为 18.
    unsigned char e=28; //定义一个变量 e，并且分配了 1 个字节的 RAM 空间。初始化默认为 28.

    unsigned char x=15; //定义一个变量 x，并且分配了 1 个字节的 RAM 空间。初始化默认为 15.
    unsigned char y=6;  //定义一个变量 y，并且分配了 1 个字节的 RAM 空间。初始化默认为 6.

    //第 1 种：“减数 1”是常量，“减数 2”是常量。
    a=15-3;

    //第 2 种：“减数 1”是变量，“减数 2”是常量。
    b=x-10;

    //第 3 种：“减数 1”是变量，“减数 2”是变量。
    c=x-y;

    //第 4 种：“减数 1”是保存变量本身，“减数 2”是常量。
    d=d-2;
    d=d-7;

    //第 5 种：“减数 1”是保存变量本身，“减数 2”是变量。
    e=e-x;
    e=e-y;

    View(a); //把第 1 个数 a 发送到电脑端的串口助手软件上观察。
    View(b); //把第 2 个数 b 发送到电脑端的串口助手软件上观察。
```

```

    View(c);    //把第 3 个数 c 发送到电脑端的串口助手软件上观察。
    View(d);    //把第 4 个数 d 发送到电脑端的串口助手软件上观察。
    View(e);    //把第 5 个数 e 发送到电脑端的串口助手软件上观察。

    while(1)
    {
    }
}

/*---C 语言学习区域的结束。-----*/

```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第 1 个数

十进制:12

十六进制:C

二进制:1100

第 2 个数

十进制:5

十六进制:5

二进制:101

第 3 个数

十进制:9

十六进制:9

二进制:1001

第 4 个数

十进制:9

十六进制:9

二进制:1001

第 5 个数

十进制:7

十六进制:7

二进制:111

分析：

通过实验结果，发现在单片机上的计算结果和我们的分析是一致的。

### 【21.3 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。

## 第二十二节：连减、自减、自减简写、自减 1。

### 【22.1 连减。】

上一节的减法例子中，右边的减数只有两个。实际上，C 语言规则没有限制减数的个数，它的通用格式如下：

“保存变量” = “减数 1” - “减数 2” - “减数 3” - ... - “减数 N”；

当右边的减数个数超过两个的时候（这里暂时把平时所说的被减数也归类为减数），这种情况就是“连减”。每个减数的属性没有限定，可以是常量，也可以是变量。比如：

```
a=68-3-15;    //减数全部是常量。
b=q-x-y-k;    //减数全部是变量。
c=63-x-5-k;    //减数有的是常量，有的是变量。
```

连减的运行顺序是，赋值符号“=”右边的减数挨个相减，把每一次的运算结果放在一个临时的隐藏中间变量里，这个隐藏的变量我们看不到，是单片机系统内部参与运算时的专用寄存器，等右边所有减数连减的计算结果出来后，再把隐藏变量所保存的计算结果赋值给左边的“保存变量”。

### 【22.2 自减、自减简写、自减 1。】

什么是自减？当赋值符号“=”右边的第 1 个减数是“保存变量”本身时（这里暂时把平时所说的被减数也归类为减数），这种情况就是“自减”。自减在程序里有一个特点，只要第 2 个减数不为 0，那么每执行一次这行代码，“保存变量”本身就会减小一次，不断执行这行代码，“保存变量”本身就会不断减小，而每次的减小量就取决于赋值符号“=”右边从第 2 个减数开始后面所有减数之和。自减的常见格式如下：

“保存变量” = “保存变量” - “减数 2”；  
“保存变量” = “保存变量” - (“减数 2” + “减数 3” + ... + “减数 N”)；

在这类自减计算式中，当只有右边的第 1 个减数是“保存变量”本身时，那么上述自减计算式可以简写成如下格式：

“保存变量” -= “减数 2”；  
“保存变量” -= “减数 2” + “减数 3” + ... + “减数 N”；

这种格式就是“自减简写”。现在举几个例子如下：

```
d-=6;    //相当于 d=d-6;
e-=x;    //相当于 e=e-x;
f-=18-y-k; //相当于 f=f-(18-y-k);
```

这些例子都是很常规的自减简写，再跟大家讲一种很常用的特殊简写。当右边只有两个减数，而第 1 个减数是“保存变量”，第 2 个减数是常数 1 时，格式如下：

“保存变量” = “保存变量” - 1；

这时候，可以把上述格式简写成如下两种格式：

“保存变量” --；  
-- “保存变量”；

这两种格式也是俗称的“自减 1”操作。比如：

```
g--;    //相当于 g=g-1 或者 g-=1;
--h;    //相当于 h=h-1 或者 h-=1;
```

自减 1 符号“--”可以在变量的左边，也可以在变量的右边，它们在这里本质是一样的，没有差别。当然，如果是在循环条件语句中，这时自减 1 符号“--”在左边还是在右边是有一点点微弱的差别，这方面的内容以后再讲。

### 【22.3 例程练习和分析。】

现在我们编写一个程序来验证上面讲到的例子：

程序代码如下：

```
/*---C 语言学习区域的开始。-----*/

void main() //主函数
{
    unsigned char a;    //定义一个变量 a, 并且分配了 1 个字节的 RAM 空间。
    unsigned char b;    //定义一个变量 b, 并且分配了 1 个字节的 RAM 空间。
    unsigned char c;    //定义一个变量 c, 并且分配了 1 个字节的 RAM 空间。
    unsigned char d=65; //定义一个变量 d, 并且分配了 1 个字节的 RAM 空间。初始化默认为 65.
    unsigned char e=38; //定义一个变量 e, 并且分配了 1 个字节的 RAM 空间。初始化默认为 38.
    unsigned char f=29; //定义一个变量 f, 并且分配了 1 个字节的 RAM 空间。初始化默认为 29.
    unsigned char g=5;  //定义一个变量 g, 并且分配了 1 个字节的 RAM 空间。初始化默认为 5.
    unsigned char h=5;  //定义一个变量 h, 并且分配了 1 个字节的 RAM 空间。初始化默认为 5.

    unsigned char q=50; //定义一个变量 q, 并且分配了 1 个字节的 RAM 空间。初始化默认为 50.
    unsigned char x=3;  //定义一个变量 x, 并且分配了 1 个字节的 RAM 空间。初始化默认为 3.
    unsigned char y=6;  //定义一个变量 y, 并且分配了 1 个字节的 RAM 空间。初始化默认为 6.
    unsigned char k=2;  //定义一个变量 k, 并且分配了 1 个字节的 RAM 空间。初始化默认为 2.

    //第 1 个知识点：连减。
    a=68-3-15;          //减数全部是常量。a 的结果为：50。
    b=q-x-y-k;          //减数全部是变量。b 的结果为：39。
    c=63-x-5-k;          //减数有的是常量，有的是变量。c 的结果为：53。

    //第 2 个知识点：自减简写。
    d-=6;                //相当于 d=d-6; d 的结果为：59。
    e-=x;                //相当于 e=e-x; e 的结果为：35。
    f-=18-y-k;           //相当于 f=f-(18-y-k); f 的结果为：19。

    //第 3 个知识点：自减 1。
    g--;                 //相当于 g=g-1 或者 g-=1; g 的结果为：4。
    --h;                 //相当于 h=h-1 或者 h-=1; d 的结果为：4。

    View(a);             //把第 1 个数 a 发送到电脑端的串口助手软件上观察。
    View(b);             //把第 2 个数 b 发送到电脑端的串口助手软件上观察。
    View(c);             //把第 3 个数 c 发送到电脑端的串口助手软件上观察。
    View(d);             //把第 4 个数 d 发送到电脑端的串口助手软件上观察。
    View(e);             //把第 5 个数 e 发送到电脑端的串口助手软件上观察。
    View(f);             //把第 6 个数 f 发送到电脑端的串口助手软件上观察。
}
```

```

    View(g);          //把第 7 个数 g 发送到电脑端的串口助手软件上观察。
    View(h);          //把第 8 个数 h 发送到电脑端的串口助手软件上观察。

    while(1)
    {
    }
}

/*---C 语言学习区域的结束。-----*/

```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第 1 个数

十进制:50

十六进制:32

二进制:110010

第 2 个数

十进制:39

十六进制:27

二进制:100111

第 3 个数

十进制:53

十六进制:35

二进制:110101

第 4 个数

十进制:59

十六进制:3B

二进制:111011

第 5 个数

十进制:35

十六进制:23

二进制:100011

第 6 个数

十进制:19

十六进制:13

二进制:10011

```
第 7 个数  
十进制:4  
十六进制:4  
二进制:100
```

```
第 8 个数  
十进制:4  
十六进制:4  
二进制:100
```

分析：

通过实验结果，发现在单片机上的计算结果和我们的分析是一致的。

#### 【22.4 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。

## 第二十三节：减法溢出与假想借位。

### 【23.1 减法溢出与假想借位。】

英文“unsigned”的中文意思就是“无符号的”，延伸含义是“无负号无负数”的意思，所以 unsigned char, unsigned int, unsigned long 这三种类型数据都是无负号无负数的，取值只能是 0 和正数，那么问题来了，当被减数小于减数的时候，运算结果会是什么样子，有什么规律？这就是本节要研究的减法溢出。

第一个例子：

```
unsigned char a;  
a=0-1;
```

分析：

左边的“保存变量”a 的数据长度是 1 个字节 8 位，a=0-1 可以看成是十六进制的 a=0x00-0x01。由于 0x00 比 0x01 小，所以假想一下需要向高位借位，借位后成了 a=0x100-0x01。所以 a 的最终结果是 0xff(十进制是 255)，这个“假想一下需要向高位借位”的过程就是本节制造的新概念“假想借位”。根据“假想借位”这个规律，如果是 b 也是 unsigned char 类型，那么 b=2-5 自然就相当于 b=0x102-0x05，运算结果 b 等于 0xfd(十进制是 253)。

第二个例子：

```
unsigned int c;  
c=0-1;
```

分析：

左边的“保存变量”c 的数据长度是 2 个字节 16 位，c=0-1 可以看成是十六进制的 c=0x0000-0x0001。由于 0x0000 比 0x0001 小，所以假想一下需要向高位借位，借位后成了 c=0x10000-0x0001。所以 c 的最终结果是 0xffff(十进制是 65535)。根据“假想借位”这个规律，如果是 d 也是 unsigned int 类型，那么 d=2-5 自然就相当于 d=0x10002-0x0005，运算结果 d 等于 0xfffd(十进制是 65533)。

综合分析：

为什么上述例子中会出现数据越减越大的奇葩现象？是因为减法溢出，是因为“假想借位”中的“借”是“光借不还”。一句话，根本问题就是溢出问题。

### 【23.2 因为减法溢出，所以加减顺序.....】

第三个例子：请分析下面例子中 e 和 f 因加减运算顺序不同而引发什么问题。

```
unsigned char e;  
unsigned char f;  
e=1-6+7;  
f=1+7-6;
```

用两种思路分析：

第一种思路：只看过程不看结果。加减法的运算优先级是从左到右，e 先减法后加法，1 减去 6 就有溢出了，所以过程有问题。而 f 先加法后减法，整个过程没有问题。

第二种思路：先看结果再分析过程。e 的运算结果居然是 2，f 的运算结果也是 2。好奇怪，既然 e 的过程有问题，为什么运算结果却没有问题？其实 e 发生两次溢出，第一次是减法溢出，第二次是加法溢出，所以“溢溢得正”（这句话是开玩笑的）。1-6“假想借位”后相当于 0x101-0x06，运算结果等于 0xfb（十进制是 251），然后 0xfb 再加上 0x07 等于 0x102，因为 e 是 unsigned char 类型只有 1 个字节，根据加法溢出的规律，最后只保留了低 8 位的一个字节 0x02，所以运算结果就是十进制的 2。



### 结论:

虽然 e 的运算结果侥幸是对的, 但是其运算过程发生了溢出是有问题的, 当运算式子更复杂一些, 比如有不同类型的变量时, 就有可能导致运算结果也出错。所以得出的结论是: 在加减法运算中, 为了减少出现减法溢出的现象, 建议先加法后减法。在后续章节讲到的乘除法运算中, 为了减小运算带来的误差也建议大家先乘法后除法。

## 【23.3 例程练习和分析。】

现在我们编写一个程序来验证上面讲到的例子:

程序代码如下:

```
/*---C 语言学习区域的开始。-----*/

void main() //主函数
{
    unsigned char a;        //定义一个变量 a, 并且分配了 1 个字节的 RAM 空间。
    unsigned char b;        //定义一个变量 b, 并且分配了 1 个字节的 RAM 空间。
    unsigned int c;         //定义一个变量 c, 并且分配了 2 个字节的 RAM 空间。
    unsigned int d;         //定义一个变量 d, 并且分配了 2 个字节的 RAM 空间。
    unsigned char e;        //定义一个变量 e, 并且分配了 1 个字节的 RAM 空间。
    unsigned char f;        //定义一个变量 f, 并且分配了 1 个字节的 RAM 空间。

    //第一个例子, 针对 a 与 b 都是 unsigned char 类型数据。
    a=0-1;
    b=2-5;

    //第二个例子, 针对 c 与 d 都是 unsigned int 类型的数据。
    c=0-1;
    d=2-5;

    //第三个例子, e 与 f 的加减顺序不一样。
    e=1-6+7;
    f=1+7-6;

    View(a);                //把第 1 个数 a 发送到电脑端的串口助手软件上观察。
    View(b);                //把第 2 个数 b 发送到电脑端的串口助手软件上观察。
    View(c);                //把第 3 个数 c 发送到电脑端的串口助手软件上观察。
    View(d);                //把第 4 个数 d 发送到电脑端的串口助手软件上观察。
    View(e);                //把第 5 个数 e 发送到电脑端的串口助手软件上观察。
    View(f);                //把第 6 个数 f 发送到电脑端的串口助手软件上观察。

    while(1)
    {
    }
```

```
}
```

```
/*---C 语言学习区域的结束。-----*/
```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第 1 个数

十进制:255

十六进制:FF

二进制:11111111

第 2 个数

十进制:253

十六进制:FD

二进制:11111101

第 3 个数

十进制:65535

十六进制:FFFF

二进制:1111111111111111

第 4 个数

十进制:65533

十六进制:FFFD

二进制:1111111111111101

第 5 个数

十进制:2

十六进制:2

二进制:10

第 6 个数

十进制:2

十六进制:2

二进制:10

分析：

通过实验结果，发现在单片机上的计算结果和我们的分析是一致的。

#### 【23.4 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，

其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。

## 第二十四节：借用 unsigned long 类型的中间变量可以减少溢出现象。

### 【24.1 为什么要借用 unsigned long 类型的中间变量？】

为什么要借用 unsigned long 类型的中间变量进行算术运算？其实就是为了减少溢出的问题。溢出是因为数据超过了它的最大范围，unsigned char, unsigned int, unsigned long 三种数据类型中，unsigned long 的取值是最大的。当参与运算变量中存在非 unsigned long 类型的时候，在运算前，先让每个非 unsigned long 类型的变量借用一个 unsigned long 类型的中间变量，然后才开始运算，可以大大减少运算中的溢出问题。unsigned long 的取值是从 0 到 4294967295，万一数据超过了 4294967295 怎么办？有两种办法，一种是换更加高级的 32 位单片机，比如 stm32 单片机就支持 64 位长度的“long long”数据类型，64 位长度的数据类型基本上可以满足绝大多数涉及运算的项目，还有一种方法思路是可以用 BCD 码的数组方式进行运算，这种数组运算的方法我以后会跟大家介绍，初学者现在暂时不用深入了解它。

### 【24.2 如何借用 unsigned long 类型的中间变量？】

借用中间变量的方法是引入中间变量，有多少个非 unsigned long 类型变量就引入多少个 unsigned long 中间变量，再借这个“壳”进行运算，最后再把中间变量的计算结果返回给实际变量。请看下面例子。

#### 转换之前：

```
unsigned int a;
unsigned char x=195;
unsigned long y=101;
a=x-y; //进行算术减法运算
```

#### 分析：

上述公式用到 3 个变量，其中 a 和 x 都不是 unsigned long 变量，因此需要为它们分别引入两个 unsigned long 类型的中间变量 t 和 s，于是乎，继续往下看.....

#### 转换之后：

```
unsigned int a;
unsigned char x=195;
unsigned long y=101;

unsigned long t; //引入的中间变量 t，用来给 a 借用。
unsigned long s; //引入的中间变量 s，用来给 x 借用。

//第一步：使用之前先清零
t=0; //t 在用之前，先把 t 的 32 位全部清零。
s=0; //s 在用之前，先把 s 的 32 位全部清零。

s=x; //s 接收 x 原数据，等效于 x 借用 unsigned long 中间变量 s 这个壳。
t=s-y; //此处 unsigned long 类型的 t 就默认代表了 unsigned int 类型的变量 a。

//第二步：因为其它的变量都是临时的，所以运算结束后再返回计算结果给原来的变量。
a=t; //运算结束后再把计算结果返回给原来的变量 a。
```

分析：

第一步：unsigned long 类型的中间变量在转换之前为什么要先赋值 0 进行清零，比如上述代码的“s=0;”？因为它是 32 位的数据类型，它也是一个随机数，如果不清零，后续的其它类型的变量可能是 16 位或者 8 位的类型变量，这些宽度不一的变量在给 32 位的变量赋值的时候，只能覆盖到 32 位变量的低 16 位或者低 8 位，无法等效于实际借用者变量的数值，所以有可能会出错。

第二步：因为其它的变量都是临时的，所以运算结束后应该再返回计算结果给原来的实际变量。在这里要多说一句，实际项目中，最后接收运算结果的变量应该根据项目所需去选择它的类型，建议尽量选择 unsigned long 类型吧，否则，如果中间变量的计算结果大于接收变量本身的类型范围，也会发生溢出。比如，上述最后一行代码 a=t，如果此时 t 的数值大于 65535，a 也会发生溢出现象。但是如果 a 本身是 unsigned long 类型，就不会发生这种现象。

加法，乘法，除法在借用中间变量的时候，跟本节减法例子中的思路也大同小异。

### 【24.3 建议在算术运算中确保所有的变量都是 unsigned long 类型。】

不管是以前讲的加法，现在讲的减法，还是未来讲的乘法和除法，我都会建议“在加减乘除四则运算中，凡是非 unsigned long 类型的变量，都应该借用 unsigned long 类型的中间变量进行运算，最后再返回计算结果给实际的变量。”unsigned long 变量是三种数据类型中取值范围最大的数，借用此类型的中间变量，可以减少在简单运算中可能出现的溢出问题。

## 第二十五节：乘法运算中的 5 种常用组合。

### 【25.1 乘法语法格式。】

乘法语法格式：

```
“保存变量” = “乘数 1” * “乘数 2” * . . * “乘数 N” ;
```

含义：为什么 C 语言的乘法符号并不是我们熟悉的“X”而是“\*”？我猜测是因为“X”跟键盘的大写字母“X”重复有冲突了，而“\*”轮廓跟“X”很相似，并且也可以在键盘上通过“Shift+8”的组合键直接键入“\*”，所以用“\*”作为乘法符号。上述乘法格式中，右边的“乘数”与“乘数”相乘（这里暂时把平时所说的被乘数也归类为乘数），并且把最终的运算结果赋值给左边的“保存变量”。注意，这里的符号“=”不是等于号的意思，而是赋值的意思。左边的“保存变量”必须是变量，不能是常量，否则编译时会报错。右边的“乘数”既可以是变量，也可以是常量，也可以是“保存变量”本身自己。多说一句，什么是变量和常量？变量是可以在程序中被更改的，被分配的一个 RAM 空间。常量往往是数字，或者被分配在 ROM 空间的一个具体数值。下面根据右边“乘数”与“乘数”的不同组合，列出了乘法运算的 5 种常用组合。

第 1 种：“乘数 1”是常量，“乘数 2”是常量。比如：

```
unsigned char a;  
a=15*3;
```

分析：数字“15”和“3”都是常量。执行上述语句后，保存变量 a 变成了 45。

第 2 种：“乘数 1”是变量，“乘数 2”是常量。比如：

```
unsigned char b;  
unsigned char x=15;  
b=x*10;
```

分析：x 是变量，“10”是常量。由于原来 x 变量里面的数值是 15，执行上述语句后，保存变量 b 变成了 150。而变量 x 则保持不变，x 还是 15。

第 3 种：“乘数 1”是变量，“乘数 2”是变量。比如：

```
unsigned char c;  
unsigned char x=15;  
unsigned char y=6;  
c=x*y;
```

分析：x 是变量，y 也是变量。由于原来 x 变量里面的数值是 15，y 变量里面的数值是 6，执行上述语句后，保存变量 c 变成了 90。而变量 x 和 y 则保持不变，x 还是 15，y 还是 6。

第 4 种：“乘数 1”是保存变量本身，“乘数 2”是常量。比如：

```
unsigned char d=18;  
d=d*2;  
d=d*7;
```

分析：d 是保存变量，“2”和“7”都是常量。这类语句有一个特点，具备了自乘功能，可以更改自己本身的数值。比如原来保存变量 d 的数值是 18，执行“d=d\*2;”语句后，d 变成了 36，接着再执行完“d=d\*7;”语句后，d 最后变成了 252。

第 5 种：“乘数 1”是保存变量本身，“乘数 2”是变量。比如：

```

unsigned char e=2;
unsigned char x=15;
unsigned char y=6;
e=e*x;
e=e*y;

```

分析：e 是保存变量，x 与 y 都是变量。这类语句有一个特点，具备了自乘功能，可以更改自己本身的数值。比如原来保存变量 e 的数值是 2，执行“e=e\*x;”语句后，e 变成了 30，接着再执行完“e=e\*y;”语句后，e 最后变成了 180。

## 【25.2 例程练习和分析。】

现在我们编写一个程序来验证上面讲到的 5 个乘法例子：

程序代码如下：

```

/*---C 语言学习区域的开始。-----*/

void main() //主函数
{
    unsigned char a;    //定义一个变量 a，并且分配了 1 个字节的 RAM 空间。
    unsigned char b;    //定义一个变量 b，并且分配了 1 个字节的 RAM 空间。
    unsigned char c;    //定义一个变量 c，并且分配了 1 个字节的 RAM 空间。
    unsigned char d=18; //定义一个变量 d，并且分配了 1 个字节的 RAM 空间。初始化默认为 18.
    unsigned char e=2;  //定义一个变量 e，并且分配了 1 个字节的 RAM 空间。初始化默认为 2.

    unsigned char x=15; //定义一个变量 x，并且分配了 1 个字节的 RAM 空间。初始化默认为 15.
    unsigned char y=6;  //定义一个变量 y，并且分配了 1 个字节的 RAM 空间。初始化默认为 6.

    //第 1 种：“乘数 1”是常量，“乘数 2”是常量。
    a=15*3;

    //第 2 种：“乘数 1”是变量，“乘数 2”是常量。
    b=x*10;

    //第 3 种：“乘数 1”是变量，“乘数 2”是变量。
    c=x*y;

    //第 4 种：“乘数 1”是保存变量本身，“乘数 2”是常量。
    d=d*2;
    d=d*7;

    //第 5 种：“乘数 1”是保存变量本身，“乘数 2”是变量。
    e=e*x;
    e=e*y;
}

```

```

    View(a);    //把第 1 个数 a 发送到电脑端的串口助手软件上观察。
    View(b);    //把第 2 个数 b 发送到电脑端的串口助手软件上观察。
    View(c);    //把第 3 个数 c 发送到电脑端的串口助手软件上观察。
    View(d);    //把第 4 个数 d 发送到电脑端的串口助手软件上观察。
    View(e);    //把第 5 个数 e 发送到电脑端的串口助手软件上观察。

    while(1)
    {
    }
}

/*---C 语言学习区域的结束。-----*/

```

在电脑串口助手软件上观察到的程序执行现象如下：

```

开始...

第 1 个数
十进制:45
十六进制:2D
二进制:101101

第 2 个数
十进制:150
十六进制:96
二进制:10010110

第 3 个数
十进制:90
十六进制:5A
二进制:1011010

第 4 个数
十进制:252
十六进制:FC
二进制:11111100

第 5 个数
十进制:180
十六进制:B4
二进制:10110100

```

分析：



通过实验结果，发现在单片机上的计算结果和我们的分析是一致的。

### 【25.3 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。

## 第二十六节：连乘、自乘、自乘简写，溢出。

### 【26.1 连乘。】

上一节的乘法例子中，右边的乘数只有两个。实际上，C 语言规则没有限制乘数的个数，它的通用格式如下：

“保存变量” = “乘数 1” \* “乘数 2” ... \* “乘数 N” ;

当右边的乘数个数超过两个的时候（这里暂时把平时所说的被乘数也归类为乘数），这种情况就是“连乘”。每个乘数的属性没有限定，可以是常量，也可以是变量。比如：

```
unsigned char x=3;    //定义一个变量 x，初始化默认为 3.
unsigned char y=6;    //定义一个变量 y，初始化默认为 6.
unsigned char k=2;    //定义一个变量 k，初始化默认为 2.
a=2*5*3;             //乘数全部是常量。a 的结果为 30。
b=k*x*y;             //乘全部是变量。b 的结果为 36。
c=x*5*y;             //乘数，有的是常量，有的是变量。c 的结果为 90。
```

连乘的运行顺序是，赋值符号“=”右边的乘数挨个相乘，把每一次的运算结果放在一个临时的隐蔽中间变量里，这个隐蔽的变量我们看不到，是单片机系统内部参与运算时的专用寄存器，等右边所有乘数连乘的计算结果出来后，再把隐蔽变量所保存的计算结果赋值给左边的“保存变量”。

### 【26.2 自乘与自乘简写。】

什么是自乘？当赋值符号“=”右边的乘数只要其中有一个是“保存变量”本身时，这种情况就是“自乘”，常见格式如下：

“保存变量” = “保存变量” \* “乘数 1”;

“保存变量” = “保存变量” \* ( “乘数 1” \* “乘数 2” ... \* “乘数 N” );

上述自乘计算式可以简写成如下格式：

“保存变量” \*= “乘数 1”;

“保存变量” \*= “乘数 1” \* “乘数 2” ... \* “乘数 N” ;

这种格式就是“自乘简写”。现在举几个例子如下：

```
unsigned char d=5;      //定义一个变量 d，初始化默认为 5.
unsigned char e=5;      //定义一个变量 e，初始化默认为 5.
unsigned char f=5;      //定义一个变量 f，初始化默认为 5.

unsigned char x=3;      //定义一个变量 x，初始化默认为 3.
unsigned char y=6;      //定义一个变量 y，初始化默认为 6.
unsigned char k=2;      //定义一个变量 k，初始化默认为 2.
d*=6;                  //相当于 d=d*6;最后 d 的结果为 30。
e*=x;                  //相当于 e=e*x;最后 e 的结果为 15。
f*=2*y*k;              //相当于 f=f*(2*y*k);最后 f 的结果为 120。
```

### 【26.3 有没有“自乘 1”的特殊写法？】

之前在讲加法的自加和减法的自减运算时，还给大家介绍了它们另外一种特殊的简写方式。比如减法运算，当右边只有 2 减数，当一个减数是“保存变量”，另一个是常数 1 时，格式如下：

“保存变量” = “保存变量” - 1;

这时候，可以把上述格式简写成如下两种格式：

“保存变量” --;

-- “保存变量”;

这两种格式也是俗称的“自减 1”操作。比如：

g--; //相当于 g=g-1 或者 g-=1;

--h; //相当于 h=h-1 或者 h-=1;

那么，本节所讲的自乘运算，有没有“g\*\*”或者“\*\*h”这种特殊的“自乘 1”写法？答案很明显，C 语言里没有“自乘 1”这种特殊写法。因为任何一个数“自乘 1”还是等于它本身，所以在乘法运算中这种特殊写法就没有存在的意义。多说一句，如果某天有朋友在某个地方看到“\*\*h”这类语句，它的本意跟“自乘”没关系，而是跟 C 语言的另一块知识点“指针”有关。

## 【26.4 乘法的溢出。】

乘法的溢出规律跟加减法的溢出规律是一样的。举一个例子如下：

```
unsigned char m=30;
unsigned char n=10;
unsigned char a;
a=m*n;
```

分析：m 与 n 相乘，相当于 30 乘以 10，运算结果是 300（十六进制是 0x012c）保存在一个隐藏中间变量，根据前面加减法运算的规律，我猜测这个隐藏中间变量可能是 unsigned int 类型，然后再把这个中间变量赋值给单字节变量 a，a 只能接收十六进制的低 8 位字节 0x2c，所以运算后 a 的数值由于溢出变成了十六进制的 0x2c（十进制是 44）。由于乘法的溢出规律跟加减法的溢出规律是一样的，所以不再多举例子。在实际项目中，为了减少溢出现象，我建议，不管加减乘除，凡是参与运算的变量全部都应该转化成 unsigned long 变量，转化的方法已经在前面章节讲过，不再重复讲解这方面的内容。

## 【26.5 例程练习和分析。】

现在编写一个程序来验证刚才讲到的连乘和自乘简写：

程序代码如下：

```
/*---C 语言学习区域的开始。-----*/

void main() //主函数
{
    unsigned char a;
    unsigned char b;
    unsigned char c;
    unsigned char d=5;    //定义一个变量 d，初始化默认为 5.
    unsigned char e=5;    //定义一个变量 e，初始化默认为 5.
    unsigned char f=5;    //定义一个变量 f，初始化默认为 5.

    unsigned char x=3;    //定义一个变量 x，初始化默认为 3.
    unsigned char y=6;    //定义一个变量 y，初始化默认为 6.
```

```

unsigned char k=2;    //定义一个变量 k，初始化默认为 2。

//第 1 个知识点：连乘。
a=2*5*3;              //乘数全部是常量。a 的结果为 30。
b=k*x*y;              //乘数全部是变量。b 的结果为 36。
c=x*5*y;              //乘数，有的是常量，有的是变量。c 的结果为 90。

//第 2 个知识点：自乘的简写。
d*=6;                 //相当于 d=d*6;最后 d 的结果为 30。
e*=x;                 //相当于 e=e*x;最后 e 的结果为 15。
f*=2*y*k;             //相当于 f=f*(2*y*k);最后 f 的结果为 120。

View(a);              //把第 1 个数 a 发送到电脑端的串口助手软件上观察。
View(b);              //把第 2 个数 b 发送到电脑端的串口助手软件上观察。
View(c);              //把第 3 个数 c 发送到电脑端的串口助手软件上观察。
View(d);              //把第 4 个数 d 发送到电脑端的串口助手软件上观察。
View(e);              //把第 5 个数 e 发送到电脑端的串口助手软件上观察。
View(f);              //把第 6 个数 f 发送到电脑端的串口助手软件上观察。

while(1)
{
}
}

/*---C 语言学习区域的结束。-----*/

```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第 1 个数

十进制:30

十六进制:1E

二进制:11110

第 2 个数

十进制:36

十六进制:24

二进制:100100

第 3 个数

十进制:90

十六进制:5A

二进制:1011010

第 4 个数

十进制:30

十六进制:1E

二进制:11110

第 5 个数

十进制:15

十六进制:F

二进制:1111

第 6 个数

十进制:120

十六进制:78

二进制:1111000

分析:

通过实验结果，发现在单片机上的计算结果和我们的分析是一致的。

## 【26.6 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。

## 第二十七节：整除求商。

### 【27.1 什么叫整除？】

最小的细分单位是“1”的除法运算就是整除，“1”不能再往下细分成小数点的除法运算就是整除。比如：

10 除以 4，商等于 2.5。-----（带小数点，这个不是整除）

10 除以 4，商等于 2，余数是 2。-----（这才是整除）

什么时候带小数点，什么时候是整除？取决于参与运算的变量类型。标准的 C 语言中，其实远远不止我前面所说的 unsigned char, unsigned int, unsigned long 这三种类型，比如还有一种叫浮点数类型的 float，当参与运算的变量存在 float 类型时，就可能存在小数点。关于小数点的问题以后再讲，现在暂时不深入讲解，现在要知道的是，unsigned char, unsigned int, unsigned long 这三种变量类型的除法都是属于整除运算，不带小数点的。

### 【27.2 整除的运算符号是什么样子的？】

10 除以 4，商等于 2，余数是 2，这个整除的过程诞生了两个结果，一个是商，一个是余数，与此对应，整除就诞生出两个运算符号，你如果想计算结果返回商就用“整除求商”的符号“/”，你如果想计算结果返回余数就用“整除求余”的符号“%”。咋一看，整除运算中用到的两个符号“/”和“%”都不是我们日常生活中熟悉的除号“÷”，我个人猜测是因为“÷”这个符号在电脑键盘上不方便直接输入，因此 C 语言的语法规则选用“/”和“%”作为整除的运算符号。

### 【27.3 整除求商“/”。】

整除求商的通用格式：

“保存变量” = “被除数” / “除数 1” / “除数 2” ... / “除数 N”；

跟之前讲的加减运算一样，赋值符号“=”左边的“保存变量”必须是变量，右边的可以是变量和常量的任意组合。如果右边只有两个参与运算的数据，就是整除求商的常见格式。

整除求商的常见格式：

“保存变量” = “被除数” / “除数”；

现在深入分析一下整除求商的运算规律。

（1）当除数等于 0 时。

我们都知道，数学运算的除数是不允许等于 0 的，如果在 51 单片机中非要让除数为 0，商会出现什么结果？我测试了一下，发现有一个规律：在 unsigned char 的变量类型下，如果“除数”是变量的 0，商等于十进制的 255（十六进制是 0xff）。如果“除数”是常量的 0，商等于十进制的 1。比如：

```
unsigned char a;
unsigned char b;
unsigned char y=0;
a=23/y; //除数变量 y 里面是 0，那么 a 的结果是 255(十六进制的 0xff)。
b=23/0; //除数是常量 0，那么 b 的结果是 1。
```

平时做项目要尽量避免“除数是 0”的情况，离它越远越好，但是既然除数不能为 0，为什么我非要做“除数为 0”时的实验呢？意义何在？这个实验的意义是，虽然我知道除数为 0 时会出错，但是我不知道这个错到底严不严重，会不会导致整个程序崩溃，当我做了这个实验后，我心中的石头才放下了，万一除数为

0 时，最多只是运算出错，但是不至于整个程序会崩溃，这样我心里就有了一个底，当哪天我某个程序崩溃跑飞时，我至少可以排除了“除数为 0”这种情况，引导我从其它方面去找 bug。

(2) 当被除数小于除数时。商等于 0。比如：

```
unsigned char c;  
c=7/10;    //c 的结果是 0。
```

(3) 当被除数等于除数时。商等于 1。比如：

```
unsigned char d;  
d=10/10;   //d 的结果是 1。
```

(4) 当被除数大于除数时。商大于 0。比如：

```
unsigned char e;  
unsigned char f;  
e=10/4;    //e 的结果是 2，大于 0。  
f=10/3;    //f 的结果是 3，大于 0。
```

#### 【27.4 整除求商的自除简写。】

当被除数是“保存变量”时，存在自除求商的简写。

“保存变量” = “保存变量” / “除数” ；

上述自除求商的简写如下：

“保存变量” / = “除数” ；

比如：

```
unsigned char g;  
g/=5;    //相当于 g=g/5;
```

#### 【27.5 整除求商有没有“自除 1”的特殊写法？】

加减法有自加 1 “++g” 和自减 1 “g--” 的特殊写法，但是求商的除法不存在这种自除 1 的特殊写法，因为一个数除以 1 的商还是等于它本身，所以求商的自除 1 没有任何意义，因此 C 语言语法中没有这种特殊写法。

#### 【27.6 整除求商的溢出。】

除法的溢出规律跟加法的溢出规律是一样的，所以不再多举例子。在实际项目中，为了避免一不小心就溢出的问题，我建议，不管加减乘除，凡是参与运算的变量全部都应该转化成 unsigned long 变量，转化的方法已经在前面章节讲过，不再重复讲解这方面的内容。

#### 【27.7 例程练习和分析。】

现在编写一个程序来验证刚才讲到的整除求商：

程序代码如下：

```

/*---C 语言学习区域的开始。-----*/

void main() //主函数
{
    unsigned char a;
    unsigned char b;
    unsigned char c;
    unsigned char d;
    unsigned char e;
    unsigned char f;
    unsigned char g=10; //初始化为 10
    unsigned char y=0;  //除数变量初始化为 0。

    //（1）当除数等于 0 时。
    a=23/y;
    b=23/0; //这行代码在编译时会引起一条警告“Warning”，暂时不用管它。

    //（2）当被除数小于除数时。
    c=7/10;

    //（3）当被除数等于除数时。
    d=10/10;

    //（4）当被除数大于除数时。
    e=10/4;
    f=10/3;

    //（5）整除求商的简写。
    g/=5; //相当于 g=g/5;

    View(a);          //把第 1 个数 a 发送到电脑端的串口助手软件上观察。
    View(b);          //把第 2 个数 b 发送到电脑端的串口助手软件上观察。
    View(c);          //把第 3 个数 c 发送到电脑端的串口助手软件上观察。
    View(d);          //把第 4 个数 d 发送到电脑端的串口助手软件上观察。
    View(e);          //把第 5 个数 e 发送到电脑端的串口助手软件上观察。
    View(f);          //把第 6 个数 f 发送到电脑端的串口助手软件上观察。
    View(g);          //把第 7 个数 g 发送到电脑端的串口助手软件上观察。

    while(1)
    {
    }
}

/*---C 语言学习区域的结束。-----*/

```



在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第 1 个数

十进制:255

十六进制:FF

二进制:11111111

第 2 个数

十进制:1

十六进制:1

二进制:1

第 3 个数

十进制:0

十六进制:0

二进制:0

第 4 个数

十进制:1

十六进制:1

二进制:1

第 5 个数

十进制:2

十六进制:2

二进制:10

第 6 个数

十进制:3

十六进制:3

二进制:11

第 7 个数

十进制:2

十六进制:2

二进制:10

分析：

通过实验结果，发现在单片机上的计算结果和我们的分析是一致的。

## 【27.8 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。

## 第二十八节：整除求余。

### 【28.1 整除求余 “%”。】

上一节讲到，求商求余都是属于整除运算，区别是：求商返回商，求余返回余，求商是“/”，求余是“%”。求余的运算符恰好就是我们平时常用的百分号“%”，之所以选择百分号作为求余的运算符，我猜测是因为，在小于 100% 的数据中，如果我们仔细回味一下百分号的分子与分母的关系，其实就隐含了一层淡淡的求余的味道。

整除求余的通用格式：

```
“保存变量” = “被除数” % “除数 1” % “除数 2” ... % “除数 N” ;
```

跟之前讲的加减运算一样，赋值符号“=”左边的“保存变量”必须是变量，右边的可以是变量和常量的任意组合。如果右边只有两个参与运算的数据，就是整除求余的常见格式。

整除求余的常见格式：

```
“保存变量” = “被除数” % “除数” ;
```

现在深入分析一下整除求余的运算规律。

(1) 当除数等于 0 时。

我们都知道，数学运算除数是不允许等于 0 的，如果在单片机中非要让除数为 0，余数会出现什么结果？我在 keil 的 C51 编译环境试过，发现有一个规律：如果除数是变量的 0，那么余数等于被除数。如果除数是常量的 0，那么余数等于 1。还有一种特殊的情况是编译不通过的，这种情况是“当被除数是变量，而除数是常量的 0”。比如：

```
unsigned char a;
unsigned char b;
unsigned char k=10;
unsigned char y=0; //除数初始化为 0

a=23%y; //除数变量 y 里面是 0，a 的结果等于被除数 23。
b=23%0; //除数是常量 0，b 的结果是 1。
b=k%0; //这种情况编译不通过：被除数是变量，而除数是常量的 0。
```

平时做项目要尽量避免“除数是 0”的情况，离它越远越好，但是既然除数不能为 0，为什么我非要做“除数为 0”时的实验呢？意义何在？这个实验的意义是，虽然我知道除数为 0 时会出错，但是我不知道这个错到底严不严重，会不会导致整个程序崩溃，当我做了这个实验后，我心中的石头才放下了，万一除数为 0 时，最多只是运算出错，但是不至于整个程序会崩溃，这样我心里就有了一个底，当哪天我某个程序崩溃跑飞时，我至少可以排除了“除数为 0”这种情况，引导我从其它方面去找 bug。

(2) 当被除数小于除数时。余数等于被除数本身。比如：

```
unsigned char c;
c=7%10; //c 的结果是 7。
```

(3) 当被除数等于除数时。余数等于 0。比如：

```
unsigned char d;
d=10%10; //d 的结果是 0。
```

(4) 当被除数大于除数时。余数必然小于除数。比如：

```
unsigned char e;
unsigned char f;
e=10%4;  //e 的结果是 2。
f=10%3;  //f 的结果是 1。
```

(5) 当除数等于 1 时。余数必然等于 0。

```
unsigned char g;
g=7%1;  //g 的结果是 0。
```

## 【28.2 整除求余的自除简写。】

当被除数是“保存变量”时，存在自除求余的简写。

“保存变量” = “保存变量” % “除数” ；

上述自除求余的简写如下：

“保存变量” % = “除数” ；

比如：

```
unsigned char h=9;
h%=5;  //相当于 h=h%5; 最后余数的计算结果是 4。
```

## 【28.3 整除求余有没有“自除 1”的特殊写法？】

加减法有自加 1 “++g” 和自减 1 “g--” 的特殊写法，但是求余的除法不存在这种自除 1 的特殊写法，因为任何一个数除以 1 的余数必然等于 0，所以求余的自除 1 没有任何意义，因此 C 语言语法中没有这种特殊写法。

## 【28.4 整除求余的溢出。】

不管是求商还是求余，除法的溢出规律跟加法的溢出规律是一样的，所以不再多举例子。在实际项目中，为了避免一不小心就溢出的问题，我建议，不管加减乘除，凡是参与运算的变量全部都应该转化成 unsigned long 变量，转化的方法已经在前面章节讲过，不再重复讲解这方面的内容。

## 【28.5 例程练习和分析。】

现在编写一个程序来验证刚才讲到的整除求余：

程序代码如下：

```
/*---C 语言学习区域的开始。-----*/

void main() //主函数
{
    unsigned char a;
    unsigned char b;
    unsigned char c;
    unsigned char d;
```

```

unsigned char e;
unsigned char f;
unsigned char g;
unsigned char h=9; //初始化为 9。

unsigned char k=10; //初始化为 10。
unsigned char y=0; //除数变量初始化为 0。

// (1) 当除数等于 0 时。
a=23%y;
b=23%0;
// b=k%0; //这种特殊情况编译不通过：“被除数”是变量，而“除数”是常量的 0。

// (2) 当被除数小于除数时。
c=7%10;

// (3) 当被除数等于除数时。
d=10%10;

// (4) 当被除数大于除数时。
e=10%4;
f=10%3;

// (5) 当除数等于 1 时。
g=7%1;

// (6) 自除求余的简写。
h%=5; //相当于 h=h%5;

View(a); //把第 1 个数 a 发送到电脑端的串口助手软件上观察。
View(b); //把第 2 个数 b 发送到电脑端的串口助手软件上观察。
View(c); //把第 3 个数 c 发送到电脑端的串口助手软件上观察。
View(d); //把第 4 个数 d 发送到电脑端的串口助手软件上观察。
View(e); //把第 5 个数 e 发送到电脑端的串口助手软件上观察。
View(f); //把第 6 个数 f 发送到电脑端的串口助手软件上观察。
View(g); //把第 7 个数 g 发送到电脑端的串口助手软件上观察。
View(h); //把第 8 个数 h 发送到电脑端的串口助手软件上观察。

while(1)
{
}
}

/*---C 语言学习区域的结束。-----*/

```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第 1 个数

十进制:23

十六进制:17

二进制:10111

第 2 个数

十进制:1

十六进制:1

二进制:1

第 3 个数

十进制:7

十六进制:7

二进制:111

第 4 个数

十进制:0

十六进制:0

二进制:0

第 5 个数

十进制:2

十六进制:2

二进制:10

第 6 个数

十进制:1

十六进制:1

二进制:1

第 7 个数

十进制:0

十六进制:0

二进制:0

第 8 个数

十进制:4

十六进制:4

二进制:100

分析：

通过实验结果，发现在单片机上的计算结果和我们的分析是一致的。

### 【28.6 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。

## 第二十九节：“先余后商”和“先商后余”提取数据某位，哪家强？

### 【29.1 先余后商。】

求商求余除了数学运算外，在实际单片机项目中还有一个很常用的功能，就是提取某个数的个十百千位。提取这些位有什么用呢？用途可大了，几乎凡是涉及界面显示的项目都要用到，比如数码管的显示，液晶屏的显示。提取某个数的个十百千位是什么意思呢？比如 8562 这个数，提取处理后，就可以得到千位的 8，百位的 5，十位的 6，个位的 2。这里提到的“个，十，百，千”位只是一个虚数，具体是多少应该根据实际项目而定，也有可能是“个，十，百，千，万，十万，百万...”等位，总之，提取的思路和方法都是一致的。下面以 8562 这个数为例开始介绍提取的思路和方法。

第一步：先把 8562 拆分成 8562, 562, 62, 2 这四个数。怎么拆分呢？用求余的算法。比如：

```
8562 等于 8562%10000;  
562 等于 8562%1000;  
62 等于 8562%100;  
2 等于 8562%10;
```

第二步：再从 8562, 562, 62, 2 这四个数中分别提取 8, 5, 6, 2 这四个数。怎么提取呢？用求商的算法。比如：

```
8 等于 8562/1000;  
5 等于 562/100;  
6 等于 62/10;  
2 等于 2/1;
```

第三步：最后，把第一步和第二步的处理思路连写在一起如下：

```
8 等于 8562%10000/1000;  
5 等于 8562%1000/100;  
6 等于 8562%100/10;  
2 等于 8562%10/1;
```

仔细观察，上述处理思路的规律感特别清晰，我们很容易发现其中的规律和原因，如果要提取“万，十万，百万...”的位数，也是用一样的思路。另外，多说一句，根据我的经验，有一些单片机的 C 编译器可能不支持 long 类型数据的求余求商连写在一起，那么就要分两步走“先求余，再求商”，分开来操作。比如：

```
unsigned char a;  
a=8562%10000/1000; //提取千位。
```

分成两步走之后如下：

```
unsigned char a;  
a=8562%10000;  
a=a/1000; //提取千位。
```

提取其它位分两步走的思路也是一样，不多说。

### 【29.2 先商后余。】



刚才讲到了“先余后商”的提取思路，其实也可以倒过来“先商后余”，也就是先求商再求余数。下面还是以 8562 这个数为例。

第一步：先把 8562 拆分成 8, 85, 856, 8562 这四个数。怎么拆分呢？用求商的算法。比如：

```
8 等于 8562/1000;  
85 等于 8562/100;  
856 等于 8562/10;  
8562 等于 8562/1;
```

第二步：再从 8, 85, 856, 8562 这四个数中分别提取 8, 5, 6, 2 这四个数。怎么提取呢？用求余的算法。比如：

```
8 等于 8%10;  
5 等于 85%10;  
6 等于 856%10;  
2 等于 8562%10;
```

第三步：最后，把第一步和第二步的处理思路连写在一起如下：

```
8 等于 8562/1000%10;  
5 等于 8562/100%10;  
6 等于 8562/10%10;  
2 等于 8562/1%10;
```

上述的规律感也是特别清晰的。

### 【29.3 “先余后商”和“先商后余”哪家强？】

上面讲了“先余后商”和“先商后余”这两种思路，到底哪种思路在实际项目中更好呢？其实我个人倾向于后者的“先商后余”，为什么呢？请看这个例子，以 3100000000 这个数为例，要提取该数的“十亿”位 3。

第一种：用“先余后商”的套路如下：

```
3 等于 3100000000%1000000000/1000000000;
```

这里出现了一个问题，我们知道，unsigned long 类型最大的数据是 0xffffffff，转换成十进制后最大的数是 4294967295，但是上面出现的 1000000000 这个数比 unsigned long 类型最大的数据 4294967295 还要大，这个就会引來我个人的担忧，C 编译器到底会怎么处理，很有可能会出现意想不到的错误，至少会让我感到心里不踏实。当然，也许会有一些朋友说，这个是多虑的，最高位完全可以把求余这一步省略，这个说法也对，但是作为一种“套路”，我还是喜欢“套路”的对称感，“套路”之所以成为“套路”，是因为有一种对称感。下面再看看如果用“先商后余”的思路来处理，会不会出现这个担忧。

第二种：用“先商后余”的套路如下：

```
3 等于 3100000000/1000000000%10;
```

这一次，上面出现的 1000000000 这个数比 unsigned long 类型最大的数据 4294967295 小，所以没有刚才那种担忧，也维护了“套路”的对称感。所以我在实际项目中喜欢用这种方法。

## 【29.4 例程练习和分析。】

现在编写一个程序来验证刚才讲到的两种思路：

程序代码如下：

```
/*---C 语言学习区域的开始。-----*/

void main() //主函数
{
    unsigned char a; //千位
    unsigned char b; //百位
    unsigned char c; //十位
    unsigned char d; //个位

    unsigned char e; //千位
    unsigned char f; //百位
    unsigned char g; //十位
    unsigned char h; //个位

    //x 初始化为 8562，必须是 unsignd int 类型以上，不能是 char 类型，char 最大范围是 255。
    unsigned int x=8562; //被提取的数

    //第一种：先余后商。
    a=x%10000/1000; //提取千位
    b=x%1000/100;    //提取百位
    c=x%100/10;      //提取十位
    d=x%10/1;        //提取个位

    //第二种：先商后余。
    e=x/1000%10;     //提取千位
    f=x/100%10;      //提取百位
    g=x/10%10;       //提取十位
    h=x/1%10;        //提取个位

    View(a);          //把第 1 个数 a 发送到电脑端的串口助手软件上观察。
    View(b);          //把第 2 个数 b 发送到电脑端的串口助手软件上观察。
    View(c);          //把第 3 个数 c 发送到电脑端的串口助手软件上观察。
    View(d);          //把第 4 个数 d 发送到电脑端的串口助手软件上观察。
    View(e);          //把第 5 个数 e 发送到电脑端的串口助手软件上观察。
    View(f);          //把第 6 个数 f 发送到电脑端的串口助手软件上观察。
    View(g);          //把第 7 个数 g 发送到电脑端的串口助手软件上观察。
    View(h);          //把第 8 个数 h 发送到电脑端的串口助手软件上观察。

    while(1)
```

```
    {  
    }  
}  
  
/*---C 语言学习区域的结束。-----*/
```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第 1 个数

十进制:8

十六进制:8

二进制:1000

第 2 个数

十进制:5

十六进制:5

二进制:101

第 3 个数

十进制:6

十六进制:6

二进制:110

第 4 个数

十进制:2

十六进制:2

二进制:10

第 5 个数

十进制:8

十六进制:8

二进制:1000

第 6 个数

十进制:5

十六进制:5

二进制:101

第 7 个数

十进制:6

十六进制:6

二进制:110

第 8 个数  
十进制:2  
十六进制:2  
二进制:10

分析:

通过实验结果，发现在单片机上的计算结果和我们的分析是一致的。

### 【29.5 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。

## 第三十节：逻辑运算符的“与”运算。

### 【30.1 “与”运算。】

不管是十进制还是十六进制，单片机底层的运算都是以二进制的形式进行的，包括前面章节的加减乘除运算，在单片机的底层处理也是以二进制形式进行。只不过加减乘除我们平时太熟悉了，以十进制的形式口算或者笔算也能得到正确的结果，所以不需要刻意把十进制的数据先转换成二进制，然后再模拟单片机底层的二进制运算。但是本节的逻辑“与”运算，在分析它的运算过程和规律的时候，必须把所有的数据都转化成二进制才能进行分析，因为它强调的是二进制的位与位之间的逻辑运算。我们知道，二进制中的每一位只能是 0 或者 1，两个数的“与”运算就是两个数被展开成二进制后的位与位之间的逻辑“与”运算。

“与”运算的运算符是“&”。运算规律是：两个位进行“与”运算，只有两个位都同时是 1 运算结果才能等于 1，否则，只要其中有一位是 0，运算结果必是 0。比如：

0&0 等于 0。  
0&1 等于 0。  
1&0 等于 0。  
1&1 等于 1。

注意，上述的 0 和 1 都是指二进制的 0 和 1。

现在举一个完整的例子来分析“与”运算的规律。有两个 unsigned char 类型的十进制数分别是 12 和 9，求 12&9 的结果是多少？分析步骤如下：

第一步：先把参与运算的两个数以二进制的格式展开。十进制转二进制的方法请参考前面第 14, 15, 16 节的内容。

十进制 12 的二进制格式是：00001100。  
十进制 9 的二进制格式是： 00001001。

第二步：二进制数右对齐，按上下每一位进行“与”运算。

十进制的 12	->	00001100
十进制的 9	->	&00001001
“与”运算结果是	->	00001000

第三步：把二进制的 00001000 转换成十六进制是：0x08。转换成十进制是 8。所以 12&9 的结果是 8。

上述举的例子只能分析“与”运算的规律，并没有看出“与”运算的意义所在。“与”运算有啥用途呢？其实用途很多，最常见的用途是可以指定一个变量二进制格式的某位清零，其它位保持不变。比如一个 unsigned char 类型的变量 b，数据长度一共是 8 位，从右往左：

想让第 0 位清零，其它位保持不变，只需跟十六进制的 0xfe 相“与”：b=b&0xfe。

想让第 1 位清零，其它位保持不变，只需跟十六进制的 0xfd 相“与”：b=b&0xfd。

想让第 2 位清零，其它位保持不变，只需跟十六进制的 0xfb 相“与”：b=b&0xfb。

想让第 3 位清零，其它位保持不变，只需跟十六进制的 0xf7 相“与”：b=b&0xf7。

想让第 4 位清零，其它位保持不变，只需跟十六进制的 0xef 相“与”：b=b&0xef。

想让第 5 位清零，其它位保持不变，只需跟十六进制的 0xdf 相“与”：b=b&0xdf。

想让第 6 位清零，其它位保持不变，只需跟十六进制的 0xbf 相“与”：b=b&0xbf。

想让第 7 位清零，其它位保持不变，只需跟十六进制的 0x7f 相“与”：b=b&0x7f。

根据上述规律，假设 b 原来等于十进制的 85（十六进制是 0x55，二进制是 01010101），要想把此数据的第 0 位清零，只需 b=b&0xfe。最终 b 的运算结果是十进制是 84（十六进制是 0x54，二进制是 01010100）。把它们展开成二进制格式的运算过程如下：

```
十进制的 85      ->    01010101
十六进制的 0xfe  ->    &11111110
“与” 运算结果是 ->    01010100
```

### 【30.2 与运算的“自与简写”。】

当被与数是“保存变量”时，存在“自与简写”。

“保存变量” = “保存变量” & “某数” ；

上述自与简写如下：

“保存变量” & = “某数” ；

比如：

```
unsigned char c=9;
c&=5; //相当于 c=c&5; 最后的计算结果 c 是 1。
```

### 【30.3 例程练习和分析。】

现在编写一个程序来验证刚才讲到的“与”运算：

程序代码如下：

```
/*---C 语言学习区域的开始。-----*/

void main() //主函数
{
    unsigned char a;
    unsigned char b=85; //十六进制是 0x55，二进制是 01010101。
    unsigned char c=9;

    a=12&9;
    b=b&0xfe;
    c&=5; //相当于 c=c&5; 最后的计算结果 c 是 1。

    View(a);           //把第 1 个数 a 发送到电脑端的串口助手软件上观察。
    View(b);           //把第 2 个数 b 发送到电脑端的串口助手软件上观察。
    View(c);           //把第 3 个数 c 发送到电脑端的串口助手软件上观察。

    while(1)
    {
    }
}

/*---C 语言学习区域的结束。-----*/
```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第 1 个数

十进制:8

十六进制:8

二进制:1000

第 2 个数

十进制:84

十六进制:54

二进制:1010100

第 3 个数

十进制:1

十六进制:1

二进制:1

分析：

通过实验结果，发现在单片机上的计算结果和我们的分析是一致的。

### 【30.4 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。

## 第三十一节：逻辑运算符的“或”运算。

### 【31.1 “或”运算。】

“或”运算也是以位为单位进行运算的。位是指二进制中的某一位，位只能是 0 或者 1。两个数的“或”运算就是转换成二进制后每一位的“或”运算。

“或”运算的符号是“|”。运算规律是：两个位的“或”运算，如果两个位都是 0，那么运算结果才是 0，否则只要其中有一位是 1，那么运算结果必定是 1。比如：

```
0|0 等于 0。
0|1 等于 1。
1|0 等于 1。
1|1 等于 1。
```

现在举一个完整的例子来分析“|”运算的规律。有两个 unsigned char 类型的十进制数分别是 12 和 9，求 12|9 的结果是多少？分析步骤如下：

第一步：先把参与运算的两个数以二进制的格式展开。十进制转二进制的方法请参考前面第 14, 15, 16 节的内容。

```
十进制 12 的二进制格式是：00001100。
十进制 9 的二进制格式是： 00001001。
```

第二步：二进制数右对齐，按上下每一位进行“或”运算。

```
十进制的 12    ->    00001100
十进制的 9     ->    |00001001
“或”运算结果是 ->    00001101
```

第三步：把二进制的 00001101 转换成十六进制是：0x0D。转换成十进制是 13。所以 12|9 的结果是 13。

上一节讲的“与”运算最常见的用途是可以指定一个变量的某位清 0，而本节的“或”运算刚好相反，“或”运算最常见的用途是可以指定一个变量的某位置 1，其它位保持不变。比如一个 unsigned char 类型的变量 b，数据长度一共是 8 位，从右往左：

```
想让第 0 位置 1，其它位保持不变，只需跟十六进制的 0x01 相“或”：b=b|0x01。
想让第 1 位置 1，其它位保持不变，只需跟十六进制的 0x02 相“或”：b=b|0x02。
想让第 2 位置 1，其它位保持不变，只需跟十六进制的 0x04 相“或”：b=b|0x04。
想让第 3 位置 1，其它位保持不变，只需跟十六进制的 0x08 相“或”：b=b|0x08。
想让第 4 位置 1，其它位保持不变，只需跟十六进制的 0x10 相“或”：b=b|0x10。
想让第 5 位置 1，其它位保持不变，只需跟十六进制的 0x20 相“或”：b=b|0x20。
想让第 6 位置 1，其它位保持不变，只需跟十六进制的 0x40 相“或”：b=b|0x40。
想让第 7 位置 1，其它位保持不变，只需跟十六进制的 0x80 相“或”：b=b|0x80。
```

根据上述规律，假设 b 原来等于十进制的 84（十六进制是 0x54，二进制是 01010100），要想把此数据的第 0 位置 1，只需 b=b|0x01。最终 b 的运算结果是十进制是 85（十六进制是 0x55，二进制是 01010101）。把它们展开成二进制格式的运算过程如下：

```
十进制的 84    ->    01010100
十六进制的 0x01 ->    |00000001
“或”运算结果是 ->    01010101
```



### 【31.2 或运算的“自或简写”。】

当被或数是“保存变量”时，存在“自或简写”。

“保存变量” = “保存变量” | “某数” ；

上述自或简写如下：

“保存变量” | = “某数” ；

比如：

```
unsigned char c=9;
c|=5; //相当于 c=c|5; 最后的计算结果 c 是 13。
```

### 【31.3 例程练习和分析。】

现在编写一个程序来验证刚才讲到的“或”运算：

程序代码如下：

```
/*---C 语言学习区域的开始。-----*/

void main() //主函数
{
    unsigned char a;
    unsigned char b=84; //十六进制是 0x54，二进制是 01010100。
    unsigned char c=9;

    a=12|9;
    b=b|0x01;
    c|=5; //相当于 c=c|5; 最后的计算结果 c 是 13。

    View(a);           //把第 1 个数 a 发送到电脑端的串口助手软件上观察。
    View(b);           //把第 2 个数 b 发送到电脑端的串口助手软件上观察。
    View(c);           //把第 3 个数 c 发送到电脑端的串口助手软件上观察。

    while(1)
    {
    }
}

/*---C 语言学习区域的结束。-----*/
```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第 1 个数  
十进制:13

```
十六进制:D  
二进制:1101  
  
第 2 个数  
十进制:85  
十六进制:55  
二进制:1010101
```

```
第 3 个数  
十进制:13  
十六进制:D  
二进制:1101
```

分析：

通过实验结果，发现在单片机上的计算结果和我们的分析是一致的。

#### 【31.4 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。

## 第三十二节：逻辑运算符的“异或”运算。

### 【32.1 “异或”运算。】

“异或”运算也是以位为单位进行运算的。位是指二进制中的某一位，位只能是 0 或者 1。两个数的“异或”运算就是转换成二进制后每一位的“异或”运算。

“异或”运算的符号是“ $\wedge$ ”。运算规律是：两个位的“异或”运算，如果两个位都相同，那么运算结果就是 0；如果两个位不同（相异），则运算结果是 1。比如：

0 $\wedge$ 0 等于 0。（两个位相同）  
0 $\wedge$ 1 等于 1。（两个位相异）  
1 $\wedge$ 0 等于 1。（两个位相异）  
1 $\wedge$ 1 等于 0。（两个位相同）

现在举一个完整的例子来分析“ $\wedge$ ”运算的规律。有两个 unsigned char 类型的十进制数分别是 12 和 9，求 12 $\wedge$ 9 的结果是多少？分析步骤如下：

第一步：先把参与运算的两个数以二进制的格式展开。十进制转二进制的方法请参考前面第 14, 15, 16 节的内容。

十进制 12 的二进制格式是：00001100。  
十进制 9 的二进制格式是： 00001001。

第二步：二进制数右对齐，按上下每一位进行“异或”运算。

十进制的 12	->	00001100
十进制的 9	->	$\wedge$ 00001001
“异或”运算结果是	->	00000101

第三步：把二进制的 00000101 转换成十六进制是：0x05。转换成十进制是 5。所以 12 $\wedge$ 9 的结果是 5。

### 【32.2 “异或”在项目中的应用。】

“异或”在哪些项目上经常应用？以我个人的项目经验，平时很少用“异或”，我本人在项目中用过两次“异或”，第一次是在某项目做串口通讯协议时，通过“异或”算法，增加一个校验字节，此校验字节是一串数据依次相“异或”的总结果，目的是为了增加数据传送时的抗干扰能力。第二次是把它用来对某变量的某个位进行取反运算，如何用“异或”来实现对某位进行取反的功能？要实现这个功能，首先要清楚“异或”运算有一个潜在的规律：任何一个位，凡是与 0 进行“异或”运算都保持不变，凡是与 1 进行“异或”运算都会达到取反的运算效果。因此，如果想某位实现取反的功能，只要把相关的位与“1”进行“异或”运算就可以实现取反的功能。二进制中的一个位要么是 0，要么是 1，不管是 0 还是 1，只要与 1 进行“异或”运算，是会达到取反的运算目的，0 的会变成 1，1 的会变成 0。请看以下这个例子：

0 $\wedge$ 1 等于 1。（两个位相异）  
1 $\wedge$ 1 等于 0。（两个位相同）

以上的例子只是列举了一个位，如果把一个字节的 8 位展开来，只要某位与“1”进行“异或”运算，都可以实现某位取反的功能。比如，一个十六进制的 0x55，如果要这个字节的低 4 位都取反，高 4 位不变，

只需要把该数据与十六进制的 0x0F 进行“异或”运算就可以达到目的。请看以下这个例子：

十六进制的 0x55	->	01010101
十六进制的 0x0F	->	^00001111
“异或”运算结果是	->	01011010

上述运算结果二进制的 01011010 转换成十六进制是 0x5A，转换成十进制是 90。

### 【32.3 异或运算的“自异或简写”。】

当被异或数是“保存变量”时，存在“自异或简写”。

“保存变量” = “保存变量” ^ “某数” ；

上述自异或简写如下：

“保存变量” ^ = “某数” ；

比如：

```
unsigned char c=9;
c^=5; //相当于 c=c^5; 最后的计算结果 c 是 12。
```

### 【32.4 例程练习和分析。】

现在编写一个程序来验证刚才讲到的“异或”运算：

程序代码如下：

```
/*---C 语言学习区域的开始。-----*/

void main() //主函数
{
    unsigned char a;
    unsigned char b;
    unsigned char c=9;

    a=12^9;
    b=0x55^0x0F;
    c^=5; //相当于 c=c^5; 最后的计算结果 c 是 12。

    View(a);           //把第 1 个数 a 发送到电脑端的串口助手软件上观察。
    View(b);           //把第 2 个数 b 发送到电脑端的串口助手软件上观察。
    View(c);           //把第 3 个数 c 发送到电脑端的串口助手软件上观察。

    while(1)
    {
    }
}

/*---C 语言学习区域的结束。-----*/
```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第 1 个数

十进制:5

十六进制:5

二进制:101

第 2 个数

十进制:90

十六进制:5A

二进制:1011010

第 3 个数

十进制:12

十六进制:C

二进制:1100

分析:

通过实验结果，发现在单片机上的计算结果和我们的分析是一致的。

### 【32.5 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。

### 第三十三节：逻辑运算符的“按位取反”和“非”运算。

#### 【33.1 “按位取反”运算。】

“按位取反”运算也是以位为单位进行运算的。位是指二进制中的某一位，位只能是 0 或者 1。跟前面“加、减、乘、除、与、或、异或”有点不一样的地方是，“按位取反”的运算只有 1 个对象，它不像加法运算那样可以与其它第 2 个对象产生关系，比如“a 加 b”这里有 2 个对象 a 和 b，而“a 按位取反”只有 1 个对象 a。一个数的“按位取反”运算就是把该数转换成二进制后对每一位的“取反”运算。

“按位取反”运算的符号是波浪符号“~”。运算规律是：针对一个数的“按位取反”，先将其展开成二进制的格式，然后每个位取反，所谓取反就是 1 的变成 0，0 的变成 1。

现在举一个完整的例子来分析“~”运算的规律。有两个 unsigned char 类型的十进制数分别是 5 和 0，求~5 和~0 的结果分别是多少？分析步骤如下：

第一步：先把参与运算的两个数以二进制的格式展开。十进制转二进制的方法请参考前面第 14, 15, 16 节的内容。

十进制 5 的二进制格式是： 00000101。

十进制 0 的二进制格式是： 00000000。

第二步：将它们二进制格式的每一位取反，1 的变成 0，0 的变成 1。

(a) 对 5 的按位取反。

十进制的 5                      ->     ~00000101

“按位取反”运算结果是 ->     11111010

(b) 对 0 的按位取反。

十进制的 0                      ->     ~00000000

“按位取反”运算结果是 ->     11111111

第三步：

(a) 把二进制的 11111010 转换成十六进制是：0xFA。转换成十进制是 250。所以~5 的结果是 250。

(b) 把二进制的 11111111 转换成十六进制是：0xFF。转换成十进制是 255。所以~0 的结果是 255。

#### 【33.2 “非”运算。】

注意，“非”运算不是以位为单位进行运算的。“非”跟“按位取反”有点相似，但是区别也明显。“按位取反”是以位为单位进行运算的，侧重在局部。而“非”是针对一个数的整体，侧重在全球。“非”只有两种状态“假”和“真”。0 代表假，大于 0 的数值代表真，也可以说“非”假即真，“非”真即假。不是假的就是真的，不是真的就是假的。强调的是两种状态的切换。在数值表示上，用 0 代表假的状态，用 1 代表真的状态。“非”的对象也只有 1 个，它不像加法运算那样可以与其它第 2 个对象产生关系，比如“a 加 b”这里有 2 个对象 a 和 b，而“a 的非”只有 1 个对象 a。

“非”运算的符号是感叹号“!”，注意输入这类运算符号的时候不能用汉字输入法，而是要切换到英文字符的输入法下再输入，否则编译不通过（其它运算符也一样，都要求在字符输入法下输入）。“非”运算的规律是：针对某个数的“非”，不管此数有多大，只要它大于 0，那么被“非”后就一定是 0。也不管此数是什么变量类型，只要它数值等于 0，那么被“非”后就一定是 1，而不是 0xff 或者 0xffff 之类。

现在举一个完整的例子来分析“!”运算的规律。有两个 unsigned char 类型的十进制数分别是 5 和 0，求!5 和!0 的结果分别是多少？分析思路如下：

(a) 针对 5 的“非”运算。

5 大于 0，是一个整体，被“非”后为 0.

(b) 针对 0 的“非”运算。

0 就是 0，是一个整体，被“非”后为 1.

### 【33.3 例程练习和分析。】

现在编写一个程序来验证刚才讲到的“按位取反”和“非”运算：

程序代码如下：

```
/*---C 语言学习区域的开始。-----*/

void main() //主函数
{
    unsigned char a=5;
    unsigned char b=5;
    unsigned char c=0;
    unsigned char d=0;

    a=~a;
    b=!b;

    c=~c;
    d=!d;

    View(a);          //把第 1 个数 a 发送到电脑端的串口助手软件上观察。
    View(b);          //把第 2 个数 b 发送到电脑端的串口助手软件上观察。
    View(c);          //把第 3 个数 c 发送到电脑端的串口助手软件上观察。
    View(d);          //把第 4 个数 d 发送到电脑端的串口助手软件上观察。

    while(1)
    {
    }
}

/*---C 语言学习区域的结束。-----*/
```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第 1 个数  
十进制:250  
十六进制:FA  
二进制:11111010

第 2 个数  
十进制:0  
十六进制:0  
二进制:0

第 3 个数  
十进制:255  
十六进制:FF  
二进制:11111111

第 4 个数  
十进制:1  
十六进制:1  
二进制:1

分析:

通过实验结果，发现在单片机上的计算结果和我们的分析是一致的。

### 【33.4 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。



## 第三十四节：移位运算的左移。

### 【34.1 “左移”运算。】

“左移”运算也是以位为单位进行运算的。位是指二进制中的某一位，位只能是 0 或者 1。欲理解某个数“左移”运算的内部规律，必先把该数展开成二进制的格式，然后才好分析。“左移”运算的符号是“<<”，它的通用格式如下：

“保存变量” = “被移数” <<n;

运算规律是：“被移数”先被复制一份放到某个隐蔽的临时变量（也称寄存器），然后对此临时变量展开成二进制的格式，左边是高位，右边是低位，此二进制格式的临时变量被整体由右往左移动了 n 位，原来左边的高 n 位数据被直接覆盖，而右边由于数据位移动而新空出的低 n 位数据被直接填入 0，最后再把移位运算的结果存入“保存变量”。多问一句，这行代码执行完毕后，“保存变量”和“被移数”到底哪个变量发生了变化，哪个变量维持不变？大家记住，只有赋值语句“=”左边的“保存变量”发生数值变化，而右边的“被移数”没有发生变化，因为“被移数”被操作的不是它自己本身，而是它的复制品替身（某个隐蔽的临时变量，也称寄存器）。这条规律对“加、减、乘、除、与、或、异或、非、取反”等运算都是适用的，重要的事情再重复一次，这条规律就是：只有赋值语句“=”左边的“保存变量”发生数值变化，而赋值语句“=”右边的“运算变量”本身不会发生变化，因为“运算变量”被操作的不是它自己本身，而是它的复制品替身（某个隐蔽的临时变量，也称寄存器）。

上述通用格式中的 n 代表被一次左移的位数，可以取 0，当 n 等于 0 的时候，代表左移 0 位，其实就是数值维持原来的样子没有发生变化。

现在举一个完整的例子来分析“<<”左移运算的规律。有两个 unsigned char 类型的变量 a 和 b，它们的数值都是十进制的 5，求  $a=a<<1$  和  $b=b<<2$  的结果分别是多少？分析步骤如下：

第一步：先把 a 和 b 变量原来的数值以二进制的格式展开。十进制转二进制的方法请参考前面第 14, 15, 16 节的内容。

a 变量是十进制 5，它的二进制格式是：00000101。

b 变量是十进制 5，它的二进制格式是：00000101。

第二步：将 a 左移 1 位，将 b 左移 2 位。

(1)  $a=a<<1$ ，就是将 a 左移 1 位。

a 左移前是       ->     00000101

a 左移 1 位后是  ->     00001010

结果分析：把二进制的 00001010 转换成十六进制是：0x0A。转换成十进制是 10。所以 a 初始值是 5，左移 1 位后的结果是 10。

(2)  $b=b<<2$ ，就是将 b 左移 2 位。

b 左移前是       ->     00000101

b 左移 2 位后是  ->     00010100

结果分析：把二进制的 00010100 转换成十六进制是：0x14。转换成十进制是 20。所以 b 初始值是 5，左移 2 位后的结果是 20。

### 【34.2 “左移”与乘法的关系。】

上面的例子，仔细观察，发现一个规律：5 左移 1 位就变成了 10（相当于 5 乘以 2），5 左移 2 位就变成

了 20（相当于 5 乘以 2 再乘以 2）。这个现象背后的规律是：在左移运算中，只要最高位不发生溢出的现象，那么每左移 1 位就相当于乘以 2，左移 2 位相当于乘以 2 再乘以 2，左移 3 位相当于乘以 2 再乘以 2 再乘以 2..... 以此类推。这个规律反过来从乘法的角度看，也是成立的：某个数乘以 2，就相当于左移 1 位，某个数乘以 2 再乘以 2 相当于左移 2 位，某个数乘以 2 再乘以 2 再乘以 2 相当于左移 3 位..... 以此类推。那么问题来了，同样是达到乘以 2 的运算结果，从运算速度的角度对比，“左移”和“乘法”哪家强？答案是：一条左移语句的运算速度比一条乘法语句的运算速度要快很多倍。

### 【34.3 “左移”的常见应用之一：不同数据类型之间的合并。】

比如有两个 unsigned char 单字节的类型数据 H 和 L，H 的初始值是十六进制的 0x12，L 的初始值是十六进制的 0x34，要将两个单字节的 H 和 L 合并成一个 unsigned int 双字节的数据 c，其中 H 是高 8 位字节，L 是低 8 位字节，合并成 c 后，c 的值应该是十六进制的 0x1234，此程序如何写？就需要用到左移。程序分析如下：

```
unsigned char H=0x12; //单字节
unsigned char L=0x34; //单字节
unsigned int c;       //双字节
c=H;                  //c 的低 8 位被 H 覆盖，也就是 c 的低 8 位得到了 H 的值。
c=c<<8;               //及时把 c 的低 8 位移动到高 8 位，同时 c 原来的低 8 位被填入 0
c=c+L;                //此时 c 再加 L，c 的低 8 位就 L 的值。
```

程序运行结果:c 就等于十六进制的 0x1234，十进制是 4660。

### 【34.4 “左移”的常见应用之二：聚焦在某个变量的某个位。】

前面第 31 节讲到“或”运算，其中讲到可以对某个变量的某个位置 1，当时是这样讲的，片段如下：

“或”运算最常见的用途是可以指定一个变量的某位置 1，其它位保持不变。比如一个 unsigned char 类型的变量 b，数据长度一共是 8 位，从右往左：

想让第 0 位置 1，其它位保持不变，只需跟十六进制的 0x01 相“或”：b=b|0x01。  
想让第 1 位置 1，其它位保持不变，只需跟十六进制的 0x02 相“或”：b=b|0x02。  
想让第 2 位置 1，其它位保持不变，只需跟十六进制的 0x04 相“或”：b=b|0x04。  
想让第 3 位置 1，其它位保持不变，只需跟十六进制的 0x08 相“或”：b=b|0x08。  
想让第 4 位置 1，其它位保持不变，只需跟十六进制的 0x10 相“或”：b=b|0x10。  
想让第 5 位置 1，其它位保持不变，只需跟十六进制的 0x20 相“或”：b=b|0x20。  
想让第 6 位置 1，其它位保持不变，只需跟十六进制的 0x40 相“或”：b=b|0x40。  
想让第 7 位置 1，其它位保持不变，只需跟十六进制的 0x80 相“或”：b=b|0x80。

但是这样写很多程序员会嫌它不直观，哪里不直观？就是 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80 这些数不直观，这些数只是代表了聚焦某个变量不同的位。如果把这些十六进制的数值换成左移的写法，在阅读上就非常清晰直观了。比如：0x01 可以用 1<<0 替代，0x02 可以用 1<<1 替代，0x04 可以用 1<<2 替代.....0x80 可以用 1<<7 替代。左移的 n 位，n 就恰好代表了某个变量的某个位。于是，我们把上面的片段更改成左移的写法后，如下：

“或”运算最常见的用途是可以指定一个变量的某位置 1，其它位保持不变。比如一个 unsigned char 类型的变量 b，数据长度一共是 8 位，从右往左：

想让第 0 位置 1，其它位保持不变，只需:b=b|(1<<0)。  
想让第 1 位置 1，其它位保持不变，只需:b=b|(1<<1)。  
想让第 2 位置 1，其它位保持不变，只需:b=b|(1<<2)。

想让第 3 位置 1，其它位保持不变，只需： $b=b|(1<<3)$ 。

想让第 4 位置 1，其它位保持不变，只需： $b=b|(1<<4)$ 。

想让第 5 位置 1，其它位保持不变，只需： $b=b|(1<<5)$ 。

想让第 6 位置 1，其它位保持不变，只需： $b=b|(1<<6)$ 。

想让第 7 位置 1，其它位保持不变，只需： $b=b|(1<<7)$ 。

分析：这样改进后，阅读就很清晰直观了，只是在程序代码的效率速度方面，因为多增加了一条左移指令，意味着要多消耗一条指令的时间，那么到底该选择哪种？其实各有利弊，应该根据个人的编程喜好和实际项目来取舍。很多 32 位的单片机在初始化寄存器的库函数里大量应用这种左移的方法来操作，目的是为了增加代码可读性。

根据上述规律，假设 d 原来等于十进制的 84（十六进制是 0x54，二进制是 01010100），要想把此数据的第 0 位置 1，只需： $d=d|(1<<0)$ 。最终 d 的运算结果是十进制是 85（十六进制是 0x55，二进制是 01010101）。

刚才上面讲到第 31 节的“或”运算，其实在第 30 节的“与”运算中也是可以用这种左移的方法来聚焦，只是要多配合一条“取反”的指令才可以。“与”运算跟“或”运算刚刚相反，它是对某个变量的某个位清零，当时是这样讲的，片段如下：

“与”运算最常见的用途是可以指定一个变量二进制格式的某位清零，其它位保持不变。比如一个 unsigned char 类型的变量 b，数据长度一共是 8 位，从右往左：

想让第 0 位清零，其它位保持不变，只需跟十六进制的 0xfe 相“与”： $b=b&0xfe$ 。

想让第 1 位清零，其它位保持不变，只需跟十六进制的 0xfd 相“与”： $b=b&0xfd$ 。

想让第 2 位清零，其它位保持不变，只需跟十六进制的 0xfb 相“与”： $b=b&0xfb$ 。

想让第 3 位清零，其它位保持不变，只需跟十六进制的 0xf7 相“与”： $b=b&0xf7$ 。

想让第 4 位清零，其它位保持不变，只需跟十六进制的 0xef 相“与”： $b=b&0xef$ 。

想让第 5 位清零，其它位保持不变，只需跟十六进制的 0xdf 相“与”： $b=b&0xdf$ 。

想让第 6 位清零，其它位保持不变，只需跟十六进制的 0xbf 相“与”： $b=b&0xbf$ 。

想让第 7 位清零，其它位保持不变，只需跟十六进制的 0x7f 相“与”： $b=b&0x7f$ 。

但是这样写很多程序员会嫌它不直观，哪里不直观？就是 0xfe, 0xfd, 0xfb, 0xf7, 0xef, 0xdf, 0xbf, 0x7f 这些数不直观，这些数只是代表了聚焦某个变量不同的位。如果把这些十六进制的数值换成左移的写法，在阅读上就非常清晰直观了，但是注意，这里左移之后还要配一条“取反”语句。比如：0xfe 可以用  $\sim(1<<0)$  替代，0xfd 可以用  $\sim(1<<1)$  替代，0xfb 可以用  $\sim(1<<2)$  替代..... 0x7f 可以用  $\sim(1<<7)$  替代。左移的 n 位后再取反，n 就恰好代表了某个变量的某个位。于是，我们把上面的片段更改成左移的写法后，如下：

“与”运算最常见的用途是可以指定一个变量二进制格式的某位清零，其它位保持不变。比如一个 unsigned char 类型的变量 b，数据长度一共是 8 位，从右往左：

想让第 0 位清零，其它位保持不变，只需： $b=b\&(\sim(1<<0))$ 。

想让第 1 位清零，其它位保持不变，只需： $b=b\&(\sim(1<<1))$ 。

想让第 2 位清零，其它位保持不变，只需： $b=b\&(\sim(1<<2))$ 。

想让第 3 位清零，其它位保持不变，只需： $b=b\&(\sim(1<<3))$ 。

想让第 4 位清零，其它位保持不变，只需： $b=b\&(\sim(1<<4))$ 。

想让第 5 位清零，其它位保持不变，只需： $b=b\&(\sim(1<<5))$ 。

想让第 6 位清零，其它位保持不变，只需： $b=b\&(\sim(1<<6))$ 。

想让第 7 位清零，其它位保持不变，只需： $b=b\&(\sim(1<<7))$ 。

分析：这样改进后，阅读就很清晰直观了，只是在程序代码的效率速度方面，因为多增加了一条左移指令和一条取反指令，意味着要多消耗两条指令的时间，那么到底该选择哪种？其实各有利弊，应该根据个人的编程喜好和实际项目来取舍。很多 32 位的单片机在初始化寄存器的库函数里大量应用这种左移的方法来

操作，目的就是增加了代码的可读性。

根据上述规律，假设 e 原来等于十进制的 85（十六进制是 0x55，二进制是 01010101），要想把此数据的第 0 位清零，只需  $e = e \& (\sim(1 \ll 0))$ 。最终 e 的运算结果是十进制的 84（十六进制是 0x54，二进制是 01010100）。

### 【34.5 左移运算的“左移简写”。】

当被移数是“保存变量”时，存在“左移简写”。

“保存变量” = “保存变量”  $\ll n$ ;

上述左移简写如下：

“保存变量”  $\ll n$ ;

比如：

```
unsigned char f=1;
unsigned char g=1;

f<<=1; //就相当于 f=f<<1;
g<<=2; //就相当于 g=g<<2;
```

### 【34.6 例程练习和分析。】

现在编写一个程序来验证刚才讲到的“左移”运算：

程序代码如下：

```
/*---C 语言学习区域的开始。-----*/

void main() //主函数
{
    unsigned char a=5;
    unsigned char b=5;

    unsigned char H=0x12; //单字节
    unsigned char L=0x34; //单字节
    unsigned int c;        //双字节

    unsigned char d=84;
    unsigned char e=85;

    unsigned char f=1;
    unsigned char g=1;

    //左移运算中蕴含着乘 2 的规律。
    a=a<<1; //a 左移 1 位，相当于 a=a*2，从原来的 5 变成了 10。
    b=b<<2; //b 左移 2 位，相当于 b=b*2*2，从原来的 5 变成了 20。

    //左移的应用之一：不同变量类型的合并。
```

```

c=H;    //c 的低 8 位被 H 覆盖，也就是此时 c 的低 8 位得到了 H 的各位值。
c=c<<8; //及时把 c 的低 8 位移动到高 8 位，同时 c 原来的低 8 位被填入 0
c=c+L;  //此时 c 再加 L，c 的低 8 位就 L 的值。此时 c 得到了 H 和 L 合并而来的值。

//左移的应用之二：聚焦在某个变量的某个位。
d=d|(1<<0);    //对第 0 位置 1。
e=e&(~(1<<0)); //对第 0 位清零。

//左移简写。
f<<=1;  //就相当于 f=f<<1;
g<<=2;  //就相当于 g=g<<2;

View(a);          //把第 1 个数 a 发送到电脑端的串口助手软件上观察。
View(b);          //把第 2 个数 b 发送到电脑端的串口助手软件上观察。
View(c);          //把第 3 个数 c 发送到电脑端的串口助手软件上观察。
View(d);          //把第 4 个数 d 发送到电脑端的串口助手软件上观察。
View(e);          //把第 5 个数 e 发送到电脑端的串口助手软件上观察。
View(f);          //把第 6 个数 f 发送到电脑端的串口助手软件上观察。
View(g);          //把第 7 个数 g 发送到电脑端的串口助手软件上观察。

while(1)
{
}
}

/*---C 语言学习区域的结束。-----*/

```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第 1 个数

十进制:10

十六进制:A

二进制:1010

第 2 个数

十进制:20

十六进制:14

二进制:10100

第 3 个数

十进制:4660

十六进制:1234

二进制:1001000110100

第 4 个数

十进制:85

十六进制:55

二进制:1010101

第 5 个数

十进制:84

十六进制:54

二进制:1010100

第 6 个数

十进制:2

十六进制:2

二进制:10

第 7 个数

十进制:4

十六进制:4

二进制:100

分析:

通过实验结果，发现在单片机上的计算结果和我们的分析是一致的。

### 【34.7 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。

## 第三十五节：移位运算的右移。

### 【35.1 “右移”运算。】

“右移”运算也是以位为单位进行运算的。位是指二进制中的某一位，位只能是 0 或者 1。欲理解某个数“右移”运算的内部规律，必先把该数展开成二进制的格式，然后才好分析。“右移”运算的符号是“>>”，它的通用格式如下：

“保存变量” = “被移数” >> n;

运算规律是：“被移数”先被复制一份放到某个隐蔽的临时变量（也称作寄存器），然后对此临时变量展开成二进制的格式，左边是高位，右边是低位，此二进制格式的临时变量被整体由左往右移动了 n 位，原来左边由于数据位移动而新空出的高 n 位数据被直接填入 0，而右边由于数据位移动而导致低 n 位数据被直接覆盖，最后再把移位运算的结果存入“保存变量”。多问一句，这行代码执行完毕后，“保存变量”和“被移数”到底哪个变量发生了变化，哪个变量维持不变？大家记住，只有赋值语句“=”左边的“保存变量”发生数值变化，而右边的“被移数”没有发生变化，因为“被移数”被操作的不是它自己本身，而是它的复制品替身（某个隐蔽的临时变量，也称作寄存器）。

上述通用格式中的 n 代表被一次右移的位数，可以取 0，当 n 等于 0 的时候，代表右移 0 位，其实就是数值维持原来的样子没有发生变化。

现在举一个完整的例子来分析“>>”右移运算的规律。有两个 unsigned char 类型的变量 a 和 b，它们的数值都是十进制的 5，求  $a=a>>1$  和  $b=b>>2$  的结果分别是多少？分析步骤如下：

第一步：先把 a 和 b 变量原来的数值以二进制的格式展开。十进制转二进制的方法请参考前面第 14, 15, 16 节的内容。

a 变量是十进制 5，它的二进制格式是：00000101。

b 变量是十进制 5，它的二进制格式是：00000101。

第二步：将 a 右移 1 位，将 b 右移 2 位。

(1)  $a=a>>1$ ，就是将 a 右移 1 位。

a 右移前是        ->    00000101

a 右移 1 位后是 ->    00000010

结果分析：把二进制的 00000010 转换成十六进制是：0x02。转换成十进制是 2。所以 a 初始值是 5，右移 1 位后的结果是 2。

(2)  $b=b>>2$ ，就是将 b 右移 2 位。

b 右移前是        ->    00000101

b 右移 2 位后是 ->    00000001

结果分析：把二进制的 00000001 转换成十六进制是：0x01。转换成十进制是 1。所以 b 初始值是 5，右移 2 位后的结果是 1。

### 【35.2 “右移”与除法的关系。】

左移一位相当于乘以 2，而右移跟左移恰恰相反，右移一位相当于除以 2，注意，这里的除法是整除，不带小数点的。比如上面例子，5 右移 1 位就变成了 2（相当于 5 整除 2 等于 2），5 右移 2 位就变成了 1（相当于 5 整除 2 再整除 2 等于 1）。这个现象背后的规律是：在右移运算中，每右移 1 位就相当于整除 2，右移 2 位相当于整除 2 再整除 2，右移 3 位相当于整除 2 再整除 2 再整除 2……以此类推。这个规律反过来从除

法的角度看，也是成立的：某个数整除 2，就相当于右移 1 位，某个数整除 2 再整除 2 相当于右移 2 位，某个数整除 2 再整除 2 再整除 2 相当于右 3 位..... 以此类推。那么问题来了，同样是达到整除 2 的运算结果，从运算速度的角度对比，“右移”和“整除”哪家强？答案是：一条右移语句的运算速度比一条整除语句的运算速度要快很多倍。

### 【35.3 “右移”的常见应用：不同数据类型之间的分解。】

比如有一个双字节 unsigned int 类型的变量 c，它的初始值是 0x1234，要把它分解成两个 unsigned char 单字节的类型数据 H 和 L，其中 H 是高 8 位字节，L 是低 8 位字节，分解后 H 应该等于 0x12，L 应该等于 0x34，此程序如何写？就需要用到右移。程序分析如下：

```
unsigned char H;          //单字节
unsigned char L;          //单字节
unsigned int c=0x1234;    //双字节
L=c;                      //c 的低 8 位直接赋值给单字节的 L
H=c>>8;                   //c 先把高 8 位右移到低 8 位，然后再把这 8 位数据赋值给 H
```

程序运行结果：H 就等于十六进制的 0x12，十进制是 18。L 就等于十六进制的 0x34，十进制是 52。提一个问题，请问执行完上述最后一条语句 H=c>>8 后，此时 c 的值是多少？答案是 c 仍然等于 0x1234，因为 c 本身没有发生变化，只要它没有赋值给它自己，执行完语句后就不会改变它自己本身，也就是本节开篇就提到的：“被移数”被操作的不是它自己本身，而是它的复制品替身（某个隐蔽的临时变量，也称寄存器）。

### 【35.4 右移运算的“右移简写”。】

当被移数是“保存变量”时，存在“右移简写”。

```
“保存变量” = “保存变量” >>n;
```

上述右移简写如下：

```
“保存变量” >>=n;
```

比如：

```
unsigned char d=8;
unsigned char e=8;

d>>=1; //就相当于 d=d>>1;
e>>=2; //就相当于 e=e>>2;
```

### 【35.5 例程练习和分析。】

现在编写一个程序来验证刚才讲到的“右移”运算：

程序代码如下：

```
/*---C 语言学习区域的开始。-----*/

void main() //主函数
{
    unsigned char a=5;
    unsigned char b=5;
```



```

unsigned char H;           //单字节
unsigned char L;           //单字节
unsigned int c=0x1234;     //双字节

unsigned char d=8;
unsigned char e=8;

//右移运算中蕴含着整除 2 的规律。
a=a>>1;                   //a 右移 1 位，相当于 a=a/2，从原来的 5 变成了 2。
b=b>>2;                   //b 右移 2 位，相当于 b=b/2/2，从原来的 5 变成了 1。

//右移的常见应用：不同变量类型的分解。
L=c;                      //c 的低 8 位直接赋值给单字节的 L
H=c>>8;                   //c 先把高 8 位右移到低 8 位，然后再把这 8 位数据赋值给 H

//右移简写。
d>>=1;                   //就相当于 d=d>>1;
e>>=2;                   //就相当于 e=e>>2;

View(a);                 //把第 1 个数 a 发送到电脑端的串口助手软件上观察。
View(b);                 //把第 2 个数 b 发送到电脑端的串口助手软件上观察。
View(H);                 //把第 3 个数 H 发送到电脑端的串口助手软件上观察。
View(L);                 //把第 4 个数 L 发送到电脑端的串口助手软件上观察。
View(d);                 //把第 5 个数 d 发送到电脑端的串口助手软件上观察。
View(e);                 //把第 6 个数 e 发送到电脑端的串口助手软件上观察。

while(1)
{
}
}

/*---C 语言学习区域的结束。-----*/

```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第 1 个数

十进制:2

十六进制:2

二进制:10

第 2 个数

```
十进制:1
十六进制:1
二进制:1

第 3 个数
十进制:18
十六进制:12
二进制:10010

第 4 个数
十进制:52
十六进制:34
二进制:110100

第 5 个数
十进制:4
十六进制:4
二进制:100

第 6 个数
十进制:2
十六进制:2
二进制:10
```

分析：

通过实验结果，发现在单片机上的计算结果和我们的分析是一致的。

### 【35.6 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。

## 第三十六节：括号的强制功能——改变运算优先级。

### 【36.1 括号的强制功能。】

C 语言中的括号有强制的功能，比如本节内容的强制改变优先级，以及以后将要讲到的数据变量类型的强制转换，指针类型的强制转换，都是要用到括号。括号就是强制，强制就是括号。

### 【36.2 括号强制改变运算优先级。】

C 语言的“加、减、乘、除、与、或、取反、左移、右移”等运算符是有严格优先级顺序的，但是我本人记忆力有限，做项目哪能记住这么多优先级的前后顺序，只是大概明白乘除的优先级比加减的优先级高，其它方面真的记不住那么多，怎么办？为了确保万一，我用到了“括号强制改变优先级”的功能，只要用了括号，就可以不按 C 语言默认的优先级顺序来出牌，可以人为的改变运算优先级，达到“随心所欲而不逾矩”的美妙境界。

括号的用法跟我们日常的数据运算公式的用法一致，先运行括号里面的运算，再执行其它运算。比如：

```
a=a<<2+5;
```

这行代码到底是先把变量 a 左移 2 位后再加 5，还是先 2 加 5 等于 7 再让变量 a 左移 7 位？对于像我这样不能熟记 C 语言运算优先级顺序的人，这条语句很容易让我搞混。但是加上括号就明了，添加括号后如下：

```
a=(a<<2)+5;
```

```
a=a<<(2+5);
```

不用多说，加上括号后，上述两行代码传递了清晰的优先级顺序。同理，再看一个例子：

```
c=1+3*c;
```

到底是 1 加 3 的结果再乘以变量 c，还是 3 乘以变量 c 的结果再加 1？因为我记得乘除法的优先级比加法的优先级高，所以答案是 3 乘以变量 c 的结果再加 1。但是对于初学者，为了避免出错，加上括号就显得更加清晰了，添加括号后如下：

```
c=(1+3)*c;
```

```
c=1+(3*c);
```

加括号后，优先级顺序一目了然。

### 【36.3 括号会不会带来额外的内存开销？】

有人会问，括号虽好，但是添加括号会不会带来额外的内存开销？答案是：不会。比如：

```
c=1+3*c;    //运算顺序：默认先乘，再加。
```

```
c=1+(3*c);  //运算顺序：强制先乘，再加。实现同样的功能，这里的括号也可以省略。
```

上面两行代码，它们的运算顺序一样的，第二行代码虽然添加了括号，但是不会带来额外的内存开销，这两行代码所占的内存大小是一样的。

括号不是鸡肋，括号应该是保健品，食之有味，又完全无副作用。用了括号可以使程序更加具有可读性，也可以让自己避开优先级顺序的大坑。

### 【36.4 例程练习和分析。】

现在编写一个程序来验证刚才讲到的主要内容：

程序代码如下：

```

/*---C 语言学习区域的开始。-----*/

void main() //主函数
{
    unsigned char a=0x01;
    unsigned char b=0x01;

    unsigned char c=0x02;
    unsigned char d=0x02;

    a=(a<<2)+5; //a 左移 2 位后变成 4，再加 5 等于 9
    b=b<<(2+5); //2 加 5 等于 7，b 再左移动 7 位等于 128

    c=(1+3)*c; //1 加 3 等于 4，再乘以变量 c 等于 8
    d=1+(3*d); //3 乘以 d 等于 6，再加 1 等于 7

    View(a);           //把第 1 个数 a 发送到电脑端的串口助手软件上观察。
    View(b);           //把第 2 个数 b 发送到电脑端的串口助手软件上观察。
    View(c);           //把第 3 个数 c 发送到电脑端的串口助手软件上观察。
    View(d);           //把第 4 个数 d 发送到电脑端的串口助手软件上观察。

    while(1)
    {
    }
}

/*---C 语言学习区域的结束。-----*/

```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第 1 个数

十进制:9

十六进制:9

二进制:1001

第 2 个数

十进制:128

十六进制:80

二进制:10000000

第 3 个数

十进制:8

十六进制:8  
二进制:1000

第 4 个数  
十进制:7  
十六进制:7  
二进制:111

分析:

通过实验结果，发现在单片机上的实验结果和我们的分析是一致的。

### 【36.5 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。

## 第三十七节：单字节变量赋值给多字节变量的疑惑。

### 【37.1 不同类型变量的赋值疑惑。】

之前讲过，多字节变量赋值给单字节变量时，多字节变量的低 8 位直接覆盖单字节变量，这个很容易理解，比如：

```
unsigned long a=0x12345678; //多字节变量
unsigned char t=0xab;      //单字节变量
t=a; //多字节赋值给单字节变量，t 的结果由原来的 0xab 变成了 0x78
```

那么，问题来了，如果调换过来，单字节赋值给多字节变量，多字节变量除了低 8 位被单字节变量所直接覆盖之外，其它剩余的位会是什么状态？会被 0 覆盖吗？还是会保持原来的数值不变？这个就是本节将要解开的疑惑。比如：

```
unsigned long a=0x12345678; //多字节变量
unsigned char t=0xab;      //单字节变量
a=t; //单字节赋值给多字节变量，此时，a 到底是 0x123456ab? 还是 0x000000ab? 疑惑中.....
```

想解开此疑惑，只要亲自上机测试一下就知道结果。经过在 keil 平台下的 C51 编译器测试后，发现结果是这样的：a 是 0x000000ab！也就是说，多字节变量其余高位是默认被 0 覆盖的。但是，我还有一个疑惑，是不是所有的 C 编译器都是这样默认处理，会不会在不同的 C 编译器平台下，会有不同的结论？所以，下面我再介绍两种比较可靠的办法给大家。

### 【37.2 我以前用的办法。】

我以前做项目的时候，每逢遇到这个疑惑，在不同变量赋值之前，我都多插入一行清零的代码，这行代码就是先把多字节变量通过直接赋值 0 来清零，因为我确信常量赋值都是直接覆盖的（其余高位都直接用 0 填充）。比如：

```
unsigned long a=0x12345678; //多字节变量
unsigned char t=0xab;      //单字节变量
a=0; //赋值之前先清零，这是我以前用的办法。
a=t; //单字节赋值给多字节变量
```

现在反省了一下，这种办法虽然可靠实用，但是显得过于保守。

### 【37.3 我现在用的办法：C 语言类型的强制转换。】

前面章节提到，括号在 C 语言中有强制的意思，可以强制改变优先级，也可以强制促进不同变量类型的匹配。比如：

```
unsigned long a=0x12345678; //多字节变量
unsigned char t=0xab;      //单字节变量
a=(unsigned long)t; //此处的括号就是强制把 t 先转变成 unsigned long 类型，然后再赋值。
这是我现在所使用的办法，推荐大家用这种。
```

### 【37.4 例程练习和分析。】

现在编写一个程序来验证刚才讲到的主要内容：

程序代码如下：

```
/*---C 语言学习区域的开始。-----*/

void main() //主函数
{

    unsigned long a=0x12345678; //多字节变量
    unsigned long b=0x12345678;
    unsigned long c=0x12345678;

    unsigned char t=0xab; //单字节变量

    a=t; //a 是 0x000000ab，其余高位默认被 0 覆盖。

    b=0; //这是我以前用的办法，显得过于保守
    b=t;

    c=(unsigned long)t; //C 语言的类型强制转换。现在推荐大家用这种。

    View(a); //把第 1 个数 a 发送到电脑端的串口助手软件上观察。
    View(b); //把第 2 个数 b 发送到电脑端的串口助手软件上观察。
    View(c); //把第 3 个数 c 发送到电脑端的串口助手软件上观察。

    while(1)
    {
    }
}

/*---C 语言学习区域的结束。-----*/
```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第 1 个数

十进制:171

十六进制:AB

二进制:10101011

第 2 个数

十进制:171

十六进制:AB

二进制:10101011

```
第 3 个数  
十进制:171  
十六进制:AB  
二进制:10101011
```

分析：

通过实验结果，发现在单片机上的实验结果和我们的分析是一致的。

### 【37.5 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。



## 第三十八节：第二种解决“运算过程中意外溢出”的便捷方法。

### 【38.1 意外溢出。】

运算过程中的意外溢出，稍不注意，就中招，不信，请看下面的例子：

```
/*---C 语言学习区域的开始。-----*/
unsigned long  a=0;
unsigned int x=1000;
unsigned int y=3000;
void main() //主函数
{
    a=x*y;    //猜猜 a 是多大？
    View(a); //把第 1 个数 a 发送到电脑端的串口助手软件上观察。
    while(1)
    {
    }
}
/*---C 语言学习区域的结束。-----*/
```

猜猜 a 是多大？很多人以为理所当然 3000000，但是实际上是 50880！中招了吧。莫名其妙的 50880，就是因为意外溢出所致。怎么办呢？请看下面介绍的两种解决办法。

### 【38.2 第一种办法：引入中间变量。】

我在前面章节中曾多次说过“为了避免运算过程中的意外溢出，建议大家把所有参与运算的变量都用 unsigned long 类型的变量，如果不是 unsigned long 类型的变量，就引入 unsigned long 类型的中间变量。”这种老方法如下：

```
/*---C 语言学习区域的开始。-----*/
unsigned long  a=0;
unsigned int x=1000;
unsigned int y=3000;
unsigned long  s; //引入的 unsigned long 中间变量。
unsigned long  t; //引入的 unsigned long 中间变量。
void main() //主函数
{
    s=x; //先把变量的数值搬到 unsigned long 中间变量。
    t=y;  //先把变量的数值搬到 unsigned long 中间变量。
    a=s*t; //中间变量代表原始变量进行运算。
    View(a); //把第 1 个数 a 发送到电脑端的串口助手软件上观察。
    while(1)
    {
    }
}
/*---C 语言学习区域的结束。-----*/
```

这一次，运算结果是正确的 3000000。

现在反省了一下，这种办法虽然可靠实用，但是显得有点罗嗦，而且引入的中间变量也无形中增加了一点内存。还有没有更好的办法？请看下面介绍的第二种办法。

### 【38.3 第二种办法：C 语言的类型强制转换。】

前面章节提到，括号在 C 语言中有强制的意思，可以强制改变优先级，在本节也可以临时强制改变运算过程中的变量类型。在运算过程中临时强制改变类型变量，就可以省去额外引入的中间变量，这种方法相比上面第一种老办法确实更便捷灵活。

```
/*---C 语言学习区域的开始。-----*/
unsigned long a=0;
unsigned int x=1000;
unsigned int y=3000;
void main() //主函数
{
    a=(unsigned long)x*(unsigned long)y; //添加的两个括号就是类型的强制转换。
    View(a); //把第 1 个数 a 发送到电脑端的串口助手软件上观察。
    while(1)
    {
    }
}
/*---C 语言学习区域的结束。-----*/
```

这一次，运算结果也是正确的 3000000。

多说一句，除了上述的乘法运算之外，其它的加、减、除法运算适不适用呢？虽然我还没有逐个测试，但是我感觉应该是都适用的。因此，在“加、减、除”等运算中，在必要的时候，也要在相关的变量的前缀加上类型的强制转换。

### 【38.4 全局变量和局部变量。】

先插入一个知识点，细心的朋友会发现，我上面的例子中，定义的变量都放在了 main 函数之外的上面，这种把变量定义在函数外面的变量叫全局变量，以前例子中定义在函数内的变量叫局部变量。

```
unsigned char a; //这个在函数之外，叫全局变量
void main() //主函数
{
    unsigned char b; //这个在函数之内，叫局部变量
    while(1)
    {
    }
}
```

上面例子中，a 定义在函数之外是全局变量，b 定义在函数之内是局部变量。全局变量与局部变量有什么不一样呢？以后的章节会仔细讲解这方面的知识，现在暂时不讲。之所以在这里提出这个知识点，是因为我今后的例子很多变量可能都会定义成全局变量，因此先在这里给大家打个招呼，知道 C 语言有这样一种语法就可以。

### 【38.5 例程练习和分析。】

现在编写一个程序来验证刚才讲到的主要内容：

程序代码如下：

```
/*---C 语言学习区域的开始。-----*/

unsigned long  a=0;
unsigned long  b=0;
unsigned long  c=0;
unsigned int  x=1000;
unsigned int  y=3000;

unsigned long  s; //中间变量
unsigned long  t;

void main() //主函数
{
    a=x*y; //意外溢出

    s=x;    //引入中间变量
    t=y;
    b=s*t;

    c=(unsigned long)x*(unsigned long)y;    //类型的强制转换

    View(a); //把第 1 个数 a 发送到电脑端的串口助手软件上观察。
    View(b); //把第 2 个数 a 发送到电脑端的串口助手软件上观察。
    View(c); //把第 3 个数 a 发送到电脑端的串口助手软件上观察。

    while(1)
    {
    }
}

/*---C 语言学习区域的结束。-----*/
```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第 1 个数

十进制:50880

十六进制:C6C0

二进制:1100011011000000

第 2 个数

十进制:3000000

十六进制:2DC6C0

二进制:1011011100011011000000

第 3 个数

十进制:3000000

十六进制:2DC6C0

二进制:1011011100011011000000

分析:

通过实验结果,发现在单片机上的实验结果和我们的分析是一致的。

### 【38.6 如何在单片机上练习本章节 C 语言程序?】

直接复制前面章节中第十一节的模板程序,练习代码时只需要更改“C 语言学习区域”的代码就可以了,其它部分的代码不要动。编译后,把程序下载进带串口的 51 学习板,通过电脑端的串口助手软件就可以观察到不同的变量数值,详细方法请看第十一节内容。

## 第三十九节：if 判断语句以及常量变量的真假判断。

### 【39.1 if 语句常规的书写格式。】

“if”在英文里的含义是“如果”的意思，在 C 语言里也是这个意思，是判断语句的专用关键词，也是平时做项目时应用的频率最高的语句之一。

如果 if 小括号里面的条件满足，就执行条件后面大括号里的语句；如果条件不满足，则直接跳过条件后面大括号里的语句。“if”语句的常见格式如下：

```
if(条件)
{
    语句 1;
    语句 2;
}
    语句 3;
    语句 4;
```

上述分析：

如果（条件）满足，就从“语句 1”开始往下执行，直到把大括号里面所有的语句执行完之后，才跳出大括号，接着从大括号之外的“语句 3”开始往下执行。

如果（条件）不满足，就直接跳过大括号里所有的语句，直接从大括号之外的“语句 3”处开始往后执行。

### 【39.2 if 语句省略大括号的用法。】

除了上述之外，还有一种省略大括号的书写格式，但是要注意，当 if 条件语句后面省略了大括号时，如果 if 小括号里面的条件满足，仅仅执行条件后面第一条语句，如果条件不满足，则跳过条件后面第一条语句。比如：

```
if(条件)
    语句 1;
    语句 2;
    语句 3;
    语句 4;
```

上述分析：

如果（条件）满足，就从语句 1 开始一直往下执行。

如果（条件）不满足，就直接跳过（条件）后的第一条语句“语句 1”，直接从（条件）后的第二条语句“语句 2”开始往后执行。

上述格式省略了大括号，实际上它等效于以下这种书写：

```
if(条件)
{
    语句 1;
}
    语句 2;
    语句 3;
    语句 4;
```

在实际项目中，为了阅读清晰，建议大家不要省略大括号。

### 【39.3 什么是真什么是假？】

刚才讲到，if 语句后面必备(条件)。那么，这个(条件)如何裁定“满足”和“不满足”？专业术语，我们用“真”表示“满足”，用“假”表示“不满足”。(条件)的真假判断，有两种：第一种是数值判断，第二种是关系判断。本节先讲第一种，数值判断。格式如下：

```
if(常量或者变量)
{
    语句 1;
    语句 2;
}
    语句 3;
    语句 4;
```

当小括号里面的(常量或者变量)不等于 0 时，就代表小括号里面的条件“满足”，是“真”；当小括号里面的(常量或者变量)等于 0 时，就代表小括号里面的条件“不满足”，是“假”。举个例子：

```
if(25)
{
    语句 1;
    语句 2;
}
    语句 3;
    语句 4;
```

上述分析：

因为“if(条件)”的“条件”是常量“25”，25 不等于 0，所以是“真”。因此，条件满足，直接从第一条语句“语句 1”处开始往下执行。

### 【39.4 例程练习和分析。】

现在编写一个程序，有 5 条 if 判断语句，如果条件为真，“统计变量 a”就会自动加 1，最后看看条件为真的语句有几条。

程序代码如下：

```
/*---C 语言学习区域的开始。-----*/

unsigned char x=2;
unsigned char y=0;
unsigned char a=0; // “统计变量 a”，此变量统计有多少条语句是真的

void main() //主函数
{
    if(1)      //常量不等于 0，因此为真
    {
        a=a+1; //a 由 0 自加 1 后变成 1。
    }
}
```

```

    }

    if(0)    //常量等于 0，因此为假
    {
        a=a+1; //由于条件为假，这条语句没有被执行，因此此时 a 仍然是 1
    }

    if(15)    //常量不等于 0，因此为真
    {
        a=a+1; //a 由 1 自加 1 后变成 2。
    }

    if(x)    //变量 x 为 2，不等于 0，因此为真
    {
        a=a+1; //a 由 2 自加 1 后变成 3。
    }

    if(y)    //变量 y 为 0，等于 0，因此为假
    {
        a=a+1; //由于条件为假，这条语句没有被执行，因此此时 a 仍然是 3
    }

    View(a); //把第 1 个数 a 发送到电脑端的串口助手软件上观察。
    while(1)
    {
    }
}

/*---C 语言学习区域的结束。-----*/

```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第 1 个数

十进制:3

十六进制:3

二进制:11

分析：

通过实验结果，发现在单片机上的实验结果和我们的分析是一致的。

### 【39.5 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，

其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。



## 第四十节：关系符的等于“==”和不等于“!=”。

### 【40.1 关系符的等于“==”和不等于“!=”。】

C 语言的“=”并不是等于号，而是赋值的意思，这点前面已讲过。为了跟赋值区分开来，C 语言用“==”来表示等于号的关系符，用“!=”表示不等于的关系符，之所以用“!=”表示不等于的关系，是因为 C 语言中的“!”就是“取非”的运算符，有否定之意。

等于关系符“==”语句的常见格式如下：

```
if(常量或变量==常量或变量)
{
    语句 1;
    语句 2;
}
语句 3;
语句 4;
```

不等于关系符“!=”语句的常见格式如下：

```
if(常量或变量!=常量或变量)
{
    语句 1;
    语句 2;
}
语句 3;
语句 4;
```

上一节讲到，常量或变量在 if 语句中的真假判断，不等于 0 就是真，等于 0 就是假。而本节关系运算符的真假判断也很简单清晰，满足条件就是真，不满足条件就是假。例如：

```
if(2==1)    //2 肯定不等于 1，所以不满足条件“等于的关系”，因此为假，不会执行大括号内的语句。
{
    语句 1;
    语句 2;
}
```

相反，请继续看下面不等于号“!=”这个例子：

```
if(2!=1)    //2 肯定不等于 1，所以满足条件“不等于的关系”，因此为真，会执行大括号内的语句。
{
    语句 1;
    语句 2;
}
```

### 【40.2 建议把常量放在“==”或“!=”关系符的左边】

“if(a==1)”和“if(1==a)”在实现的功能上是相同的。但是，在实际做项目的时候，还是建议大家采用后面这种写法“if(1==a)”，把常量放在左边，这样写有什么好处？好处是，如果我们不小心把等于号“==”或者“!=”误写成赋值符号“=”时，C编译器在编译时，它能及时发现错误并且报错告知我们，因为常量在左边是无法赋值的，编译器能及时发现错误。但是如果常量在右边而变量在左边，因为变量是允许赋值的，所以有一些C语言编译器未必会报错，就会留下不易察觉的程序隐患。比如：

```
if(a==5)
{
    语句 1;
}

if(b!=2)
{
    语句 2;
}
```

建议改成：

```
if(5==a)
{
    语句 1;
}

if(2!=b)
{
    语句 2;
}
```

### 【40.3 例程练习和分析。】

现在编写一个实验程序，一共有8个给定的数，要统计其中数值“等于85”的数有几个，统计其中数值“不等于75”的数有几个。

程序代码如下：

```
/*---C 语言学习区域的开始。-----*/

unsigned char x1=90; //给定的第 1 个数
unsigned char x2=65; //给定的第 2 个数
unsigned char x3=85; //给定的第 3 个数
unsigned char x4=79; //给定的第 4 个数
unsigned char x5=95; //给定的第 5 个数
unsigned char x6=65; //给定的第 6 个数
unsigned char x7=75; //给定的第 7 个数
unsigned char x8=85; //给定的第 8 个数
```

```
unsigned char a=0; //统计等于 85 的变量总数
unsigned char b=0; //统计不等于 75 的变量总数

void main() //主函数
{
    //第一部分:统计“等于 85”的总数有多少个。
    if(85==x1) //把常量 85 放在等于号的左边
    {
        a++; //相当于 a=a+1, 用来统计等于 85 的总数
    }

    if(85==x2) //把常量 85 放在等于号的左边
    {
        a++; //相当于 a=a+1, 用来统计等于 85 的总数
    }

    if(85==x3) //把常量 85 放在等于号的左边
    {
        a++; //相当于 a=a+1, 用来统计等于 85 的总数
    }

    if(85==x4) //把常量 85 放在等于号的左边
    {
        a++; //相当于 a=a+1, 用来统计等于 85 的总数
    }

    if(85==x5) //把常量 85 放在等于号的左边
    {
        a++; //相当于 a=a+1, 用来统计等于 85 的总数
    }

    if(85==x6) //把常量 85 放在等于号的左边
    {
        a++; //相当于 a=a+1, 用来统计等于 85 的总数
    }

    if(85==x7) //把常量 85 放在等于号的左边
    {
        a++; //相当于 a=a+1, 用来统计等于 85 的总数
    }

    if(85==x8) //把常量 85 放在等于号的左边
    {
        a++; //相当于 a=a+1, 用来统计等于 85 的总数
    }
}
```

```
}
```

```
//第二部分:统计“不等于 75”的总数有多少个。
```

```
if(75!=x1) //把常量 75 放在不等于号的左边
```

```
{
```

```
    b++; //相当于 b=b+1, 用来统计不等于 75 的总数
```

```
}
```

```
if(75!=x2) //把常量 75 放在不等于号的左边
```

```
{
```

```
    b++; //相当于 b=b+1, 用来统计不等于 75 的总数
```

```
}
```

```
if(75!=x3) //把常量 75 放在不等于号的左边
```

```
{
```

```
    b++; //相当于 b=b+1, 用来统计不等于 75 的总数
```

```
}
```

```
if(75!=x4) //把常量 75 放在不等于号的左边
```

```
{
```

```
    b++; //相当于 b=b+1, 用来统计不等于 75 的总数
```

```
}
```

```
if(75!=x5) //把常量 75 放在不等于号的左边
```

```
{
```

```
    b++; //相当于 b=b+1, 用来统计不等于 75 的总数
```

```
}
```

```
if(75!=x6) //把常量 75 放在不等于号的左边
```

```
{
```

```
    b++; //相当于 b=b+1, 用来统计不等于 75 的总数
```

```
}
```

```
if(75!=x7) //把常量 75 放在不等于号的左边
```

```
{
```

```
    b++; //相当于 b=b+1, 用来统计不等于 75 的总数
```

```
}
```

```
if(75!=x8) //把常量 75 放在不等于号的左边
```

```
{
```

```
    b++; //相当于 b=b+1, 用来统计不等于 75 的总数
```

```
}
```

```
View(a); //把第 1 个数 a 发送到电脑端的串口助手软件上观察。
```

```

    View(b); //把第 2 个数 b 发送到电脑端的串口助手软件上观察。

    while(1)
    {
    }
}

/*---C 语言学习区域的结束。-----*/

```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第 1 个数

十进制:2

十六进制:2

二进制:10

第 2 个数

十进制:7

十六进制:7

二进制:111

分析：

变量 a 为 2。（等于 85 的有 x3, x8 这 2 个）

变量 b 为 7。（不等于 75 的有 x1, x2, x3, x4, x5, x6, x8 这 7 个）

通过实验结果，发现在单片机上的实验结果和我们的分析是一致的。

#### 【40.4 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。

## 第四十一节：关系符的大于“>”和大于等于“>=”。

### 【41.1 大于“>”。】

大于关系符“>”语句的常见格式如下：

```
if(常量或变量>常量或变量)
{
    语句 1;
    语句 2;
}
语句 3;
语句 4;
```

上述 if 条件的真假判断规则是：如果左边的数大于右边的数，此条件为真（条件满足）。否则，为假（条件不满足）。例如：

```
if(2>1)    //2 肯定大于 1，所以满足条件“大于的关系”，因此为真，会执行大括号内的语句。
{
    语句 1;
    语句 2;
}
```

### 【41.2 大于等于“>=”。

大于关系符“>=”语句的常见格式如下：

```
if(常量或变量>=常量或变量)
{
    语句 1;
    语句 2;
}
语句 3;
语句 4;
```

上述 if 条件的真假判断规则是：如果左边的数大于或者等于右边的数，此条件为真（条件满足）。否则，为假（条件不满足）。例如：

```
if(2>=2)    //左边的 2 虽然不大于右边的 2，但是左边的 2 等于右边的 2，因此为真，满足条件。
{
    语句 1;
    语句 2;
}
```

### 【41.3 例程练习和分析。】

现在编写一个实验程序，一共有 8 个给定的数，要统计其中数值大于 79 的数有几个，同时，也统计其中数值大于等于 79 的数又有几个。

程序代码如下：

```

/*---C 语言学习区域的开始。-----*/
    unsigned char x1=90; //给定的第 1 个数
    unsigned char x2=65; //给定的第 2 个数
    unsigned char x3=85; //给定的第 3 个数
    unsigned char x4=79; //给定的第 4 个数
    unsigned char x5=95; //给定的第 5 个数
    unsigned char x6=65; //给定的第 6 个数
    unsigned char x7=75; //给定的第 7 个数
    unsigned char x8=85; //给定的第 8 个数

    unsigned char a=0; //统计大于 79 的变量总数
    unsigned char b=0; //统计大于等于 79 的变量总数

void main() //主函数
{
    //第一部分:统计“大于 79”的总数有多少个。

    if(x1>79) //如果条件为真,则执行下面大括号里面的语句。
    {
        a++; //相当于 a=a+1,用来统计大于 79 的总数
    }

    if(x2>79) //如果条件为真,则执行下面大括号里面的语句。
    {
        a++; //相当于 a=a+1,用来统计大于 79 的总数
    }

    if(x3>79) //如果条件为真,则执行下面大括号里面的语句。
    {
        a++; //相当于 a=a+1,用来统计大于 79 的总数
    }

    if(x4>79) //如果条件为真,则执行下面大括号里面的语句。
    {
        a++; //相当于 a=a+1,用来统计大于 79 的总数
    }

    if(x5>79) //如果条件为真,则执行下面大括号里面的语句。
    {
        a++; //相当于 a=a+1,用来统计大于 79 的总数
    }

    if(x6>79) //如果条件为真,则执行下面大括号里面的语句。
    {

```

```

    a++;    //相当于 a=a+1, 用来统计大于 79 的总数
}

if(x7>79)  //如果条件为真, 则执行下面大括号里面的语句。
{
    a++;    //相当于 a=a+1, 用来统计大于 79 的总数
}

if(x8>79)  //如果条件为真, 则执行下面大括号里面的语句。
{
    a++;    //相当于 a=a+1, 用来统计大于 79 的总数
}

//第二部分:统计“大于等于 79”的总数有多少个。

if(x1>=79) //如果条件为真, 则执行下面大括号里面的语句。
{
    b++;    //相当于 b=b+1, 用来统计大于等于 79 的总数
}

if(x2>=79) //如果条件为真, 则执行下面大括号里面的语句。
{
    b++;    //相当于 b=b+1, 用来统计大于等于 79 的总数
}

if(x3>=79) //如果条件为真, 则执行下面大括号里面的语句。
{
    b++;    //相当于 b=b+1, 用来统计大于等于 79 的总数
}

if(x4>=79) //如果条件为真, 则执行下面大括号里面的语句。
{
    b++;    //相当于 b=b+1, 用来统计大于等于 79 的总数
}

if(x5>=79) //如果条件为真, 则执行下面大括号里面的语句。
{
    b++;    //相当于 b=b+1, 用来统计大于等于 79 的总数
}

if(x6>=79) //如果条件为真, 则执行下面大括号里面的语句。
{
    b++;    //相当于 b=b+1, 用来统计大于等于 79 的总数
}

```



```

    if(x7>=79)  //如果条件为真，则执行下面大括号里面的语句。
    {
        b++;    //相当于 b=b+1，用来统计大于等于 79 的总数
    }

    if(x8>=79)  //如果条件为真，则执行下面大括号里面的语句。
    {
        b++;    //相当于 b=b+1，用来统计大于等于 79 的总数
    }

    View(a);    //把第 1 个数 a 发送到电脑端的串口助手软件上观察。
    View(b);    //把第 2 个数 b 发送到电脑端的串口助手软件上观察。

    while(1)
    {
    }
}

/*---C 语言学习区域的结束。-----*/

```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第 1 个数

十进制:4

十六进制:4

二进制:100

第 2 个数

十进制:5

十六进制:5

二进制:101

分析：

变量 a 为 4。（大于 79 的有 x1, x3, x5, x8 这 4 个）

变量 b 为 5。（大于等于 79 的有 x1, x3, x4, x5, x8 这 5 个）

通过实验结果，发现在单片机上的实验结果和我们的分析是一致的。

#### 【41.4 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。

## 第四十二节：关系符的小于“<”和小于等于“<=”。

### 【42.1 小于“<”。】

小于关系符“<”语句的常见格式如下：

```
if(常量或变量<常量或变量)
{
    语句 1;
    语句 2;
}
语句 3;
语句 4;
```

上述 if 条件的真假判断规则是：如果左边的数小于右边的数，此条件为真（条件满足）。否则，为假（条件不满足）。例如：

```
if(2<1)    //2 肯定不小于 1，所以不满足条件“小于的关系”，因此为假，不会执行大括号内的语句。
{
    语句 1;
    语句 2;
}
```

### 【42.2 小于等于“<=”。】

小于关系符“<=”语句的常见格式如下：

```
if(常量或变量<=常量或变量)
{
    语句 1;
    语句 2;
}
语句 3;
语句 4;
```

上述 if 条件的真假判断规则是：如果左边的数小于或者等于右边的数，此条件为真（条件满足）。否则，为假（条件不满足）。例如：

```
if(2<=2)    //左边的 2 虽然不小于右边的 2，但是左边的 2 等于右边的 2，因此为真，满足条件。
{
    语句 1;
    语句 2;
}
```

### 【42.3 例程练习和分析。】

现在编写一个实验程序，一共有 8 个给定的数，要统计其中数值小于 79 的数有几个，统计其中数值小于等于 79 的数有几个。

程序代码如下：

```

/*---C 语言学习区域的开始。-----*/
    unsigned char x1=90; //给定的第 1 个数
    unsigned char x2=65; //给定的第 2 个数
    unsigned char x3=85; //给定的第 3 个数
    unsigned char x4=79; //给定的第 4 个数
    unsigned char x5=95; //给定的第 5 个数
    unsigned char x6=65; //给定的第 6 个数
    unsigned char x7=75; //给定的第 7 个数
    unsigned char x8=85; //给定的第 8 个数

    unsigned char a=0; //统计小于 79 的变量总数
    unsigned char b=0; //统计小于等于 79 的变量总数

void main() //主函数
{
    //第一部分:统计“小于 79”的总数有多少个。

    if(x1<79) //如果条件为真,则执行下面大括号里面的语句。
    {
        a++; //相当于 a=a+1,用来统计小于 79 的总数
    }

    if(x2<79) //如果条件为真,则执行下面大括号里面的语句。
    {
        a++; //相当于 a=a+1,用来统计小于 79 的总数
    }

    if(x3<79) //如果条件为真,则执行下面大括号里面的语句。
    {
        a++; //相当于 a=a+1,用来统计小于 79 的总数
    }

    if(x4<79) //如果条件为真,则执行下面大括号里面的语句。
    {
        a++; //相当于 a=a+1,用来统计小于 79 的总数
    }

    if(x5<79) //如果条件为真,则执行下面大括号里面的语句。
    {
        a++; //相当于 a=a+1,用来统计小于 79 的总数
    }

    if(x6<79) //如果条件为真,则执行下面大括号里面的语句。
    {

```

```

    a++;    //相当于 a=a+1, 用来统计小于 79 的总数
}

if(x7<79)  //如果条件为真, 则执行下面大括号里面的语句。
{
    a++;    //相当于 a=a+1, 用来统计小于 79 的总数
}

if(x8<79)  //如果条件为真, 则执行下面大括号里面的语句。
{
    a++;    //相当于 a=a+1, 用来统计小于 79 的总数
}

//第二部分:统计“小于等于 79”的总数有多少个。

if(x1<=79) //如果条件为真, 则执行下面大括号里面的语句。
{
    b++;    //相当于 b=b+1, 用来统计小于等于 79 的总数
}

if(x2<=79) //如果条件为真, 则执行下面大括号里面的语句。
{
    b++;    //相当于 b=b+1, 用来统计小于等于 79 的总数
}

if(x3<=79) //如果条件为真, 则执行下面大括号里面的语句。
{
    b++;    //相当于 b=b+1, 用来统计小于等于 79 的总数
}

if(x4<=79) //如果条件为真, 则执行下面大括号里面的语句。
{
    b++;    //相当于 b=b+1, 用来统计小于等于 79 的总数
}

if(x5<=79) //如果条件为真, 则执行下面大括号里面的语句。
{
    b++;    //相当于 b=b+1, 用来统计小于等于 79 的总数
}

```

```

    if(x6<=79)  //如果条件为真，则执行下面大括号里面的语句。
    {
        b++;    //相当于 b=b+1，用来统计小于等于 79 的总数
    }

    if(x7<=79)  //如果条件为真，则执行下面大括号里面的语句。
    {
        b++;    //相当于 b=b+1，用来统计小于等于 79 的总数
    }

    if(x8<=79)  //如果条件为真，则执行下面大括号里面的语句。
    {
        b++;    //相当于 b=b+1，用来统计小于等于 79 的总数
    }

    View(a);    //把第 1 个数 a 发送到电脑端的串口助手软件上观察。
    View(b);    //把第 2 个数 b 发送到电脑端的串口助手软件上观察。

    while(1)
    {
    }
}

/*---C 语言学习区域的结束。-----*/

```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第 1 个数

十进制:3

十六进制:3

二进制:11

第 2 个数

十进制:4

十六进制:4

二进制:100

分析：

变量 a 为 3。（小于 79 的有 x2, x6, x7 这 3 个）

变量 b 为 4。（小于等于 79 的有 x2, x4, x6, x7 这 4 个）

通过实验结果，发现在单片机上的实验结果和我们的分析是一致的。

#### 【42.4 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。

## 第四十三节：关系符中的关系符:与“&&”，或“||”。

### 【43.1 关系符中的与“&&”。】

前面在讲关系符的时候，讲了只存在 1 个（判断条件）的情况下，根据这个判断为真还是为假再执行对应的操作，那么，当同时存在 2 个（判断条件）以上的情况下，该如何描述（判断条件）与（判断条件）之间的关系，这就涉及本节所讲的“关系符中的关系符”：与“&&”，或“||”。

先讲“&&”语句，符号“&&”称为“与”，它的含义是：假如有两个以上的（条件判断），当所有的（条件判断）都满足的时候，才认为这个整体判断是真，否则，只要有 1 个（条件判断）不满足，那么整体判断就是假。这个规律，有点像很多开关在电路回路中的串联关系，只有所有串联在回路中的开关都是闭合的状态，这个回路才是畅通的，否则，只要有一个开关是断开的，整个回路就是断开的。

与语句“&&”的常见格式如下：

```
if(第 1 个条件判断&&第 2 个条件判断...&&第 N 个条件判断)
{
    语句 1;
    语句 2;
}
语句 3;
语句 4;
```

在上述格式中，只有 if 语句后面小括号内所有的(条件判断)都满足的时候，整体判断才为真，才会执行到大括号内的“语句 1”和“语句 2”，否则，只要有 1 个不满足，就直接跳到“语句 3”处往下执行。

再举一个具体的例子，比如要取从 70 到 80 之间的所有数据，也就是说，既要大于等于 70，同时又要小于等于 80，程序代码可以这样书写：

```
if(a>=70&&a<=80)  //在 70 到 80 的区间范围（包括边界 70 和 80）
{
    语句 1;
    语句 2;
    .....
    语句 N;
}
```

### 【43.2 关系符中的或“||”。】

符号“||”称为“或”，它的含义是：假如有两个以上的（条件判断），只要有 1 个条件判断为真，则此整体判断裁定为真，否则，必须所有的（条件判断）都不满足，此整体判断才会裁定为假。这个规律，有点像很多开关在电路回路中的并联关系，并联在回路的多个开关，只要有 1 个开关是闭合的状态，那么这个回路肯定是畅通的，否则，必须全部开关都是断开的，整个回路才会是断开的。

或语句“||”的常见格式如下：

```
if(第 1 个条件判断||第 2 个条件判断...||第 N 个条件判断)
{
    语句 1;
    语句 2;
}
```

语句 3;

语句 4;

在上述格式中，只要 if 语句后面小括号内有 1 个(条件判断)是满足的时候，整体判断马上裁定为真，这时就会执行到大括号内的“语句 1”和“语句 2”，否则，必须全部的(条件判断)都不满足，整体判断才会裁定为假，这时就会直接跳到“语句 3”处往下执行。

再举一个具体的例子，比如要取除了 70 到 80 之间以外的所有数据，也就是说，要么小于 70，或者要么大于 80，可以这样写：

```
if(a<70||a>80) //在 70 到 80 的区间范围以外的数据（不包括边界 70 和 80）
{
    语句 1;
    语句 2;
    .....
    语句 N;
}
```

### 【43.3 “&”和“&&”，“|”和“||”的区别。】

前面章节讲过运算符的“&”和“|”，它们发音也是“与”和“或”，跟本节讲的关系符“&&”和“||”的发音是同音，因此很容易让初学者混淆。区别如下：

运算符的“&”和“|”，是属于运算符，是强调数与数，变量与变量，个体与个体之间的运算，而不是关系。它们之间的运算，是把一个数或一个变量转化成二进制后，进行二进制的 0 和 1 之间的“与”“或”运算。

关系符的“&&”和“||”，是属于关系符，是强调（条件判断）与（条件判断），关系与关系，整体与整体之间的关系判断，而不是运算。它们之间的关系，关键词是判断。

### 【43.4 “&&”和“||”的“短路”问题。】

关系符“&&”和“||”居然也有“短路”问题？大家不要惊异，这里所说的“短路”只是强调关系符内部判断的顺序和取舍。“短路”这个词在这里只是业内已经习惯了一种称谓，虽然我个人感觉有一点怪怪的不自然，但是我自己也想不出其它更好的词来描述这种关系，因此就跟业内已习惯的称谓保持一致。

“&&”的“短路”，它内部判断的顺序和取舍是这个样子的：在两个以上的判断中，从左边到右边，依次逐个判断，先判断第 1 个（条件判断），再第 2 个（条件判断）...再第 N 个（条件判断），但是，在此期间，只要发现有 1 个条件是不满足，就马上退出判断，不再继续判断后面的（条件判断），因为，对于“与”的关系符，只要有 1 个条件判断是不满足（假），就可以马上裁定整体判断为假了，没必要继续判断后面的（条件判断）。

“||”的“短路”，它内部判断的顺序和取舍是这个样子的：在两个以上的判断中，从左边到右边，依次逐个判断，先判断第 1 个（条件判断），再第 2 个（条件判断）...再第 N 个（条件判断），但是，在此期间，只要发现有 1 个条件是满足，就马上退出判断，不再继续判断后面的（条件判断），因为，对于“或”的关系符，只要有 1 个条件判断是满足（真），就可以马上裁定整体判断为真了，没必要继续判断后面的（条件判断）。



上述文字中的“从左到右”就是“顺序”，“马上退出”就是“取舍”。这种关系之所以称谓为“短路”，我猜测可能是把“&&”和“||”比喻成在电路的回路中，只要有个1个地方短路了，就可以马上裁定这个回路是短路的，就不用再判断其它地方了。

### 【43.5 例程练习和分析。】

现在编写一个实验程序，一共有8个给定的数，要统计其中数值从70到80之间的数有几个，统计其中取除了70到80之间以外的数有几个。

程序代码如下：

```
/*---C 语言学习区域的开始。-----*/
unsigned char x1=90; //给定的第1个数
unsigned char x2=65; //给定的第2个数
unsigned char x3=85; //给定的第3个数
unsigned char x4=79; //给定的第4个数
unsigned char x5=95; //给定的第5个数
unsigned char x6=65; //给定的第6个数
unsigned char x7=75; //给定的第7个数
unsigned char x8=85; //给定的第8个数

unsigned char a=0; //统计从70到80的变量总数
unsigned char b=0; //统计除了70到80以外的变量总数

void main() //主函数
{
    //第一部分:统计“从70到80之间的数有多少个。

    if(x1>=70&& x1<=80) //如果条件为真，则执行下面大括号里面的语句。
    {
        a++; //相当于 a=a+1，用来统计从70到80的总数
    }

    if(x2>=70&& x2<=80) //如果条件为真，则执行下面大括号里面的语句。
    {
        a++; //相当于 a=a+1，用来统计从70到80的总数
    }

    if(x3>=70&& x3<=80) //如果条件为真，则执行下面大括号里面的语句。
    {
        a++; //相当于 a=a+1，用来统计从70到80的总数
    }

    if(x4>=70&& x4<=80) //如果条件为真，则执行下面大括号里面的语句。
    {
```

```

    a++;    //相当于 a=a+1, 用来统计从 70 到 80 的总数
}

if (x5>=70&& x5<=80)    //如果条件为真, 则执行下面大括号里面的语句。
{
    a++;    //相当于 a=a+1, 用来统计从 70 到 80 的总数
}

if (x6>=70&& x6<=80)    //如果条件为真, 则执行下面大括号里面的语句。
{
    a++;    //相当于 a=a+1, 用来统计从 70 到 80 的总数
}

if (x7>=70&& x7<=80)    //如果条件为真, 则执行下面大括号里面的语句。
{
    a++;    //相当于 a=a+1, 用来统计从 70 到 80 的总数
}

if (x8>=70&& x8<=80)    //如果条件为真, 则执行下面大括号里面的语句。
{
    a++;    //相当于 a=a+1, 用来统计从 70 到 80 的总数
}

//第二部分:统计除了 70 到 80 之间以外的数有多少个。

if (x1<70|| x1>80)    //如果条件为真, 则执行下面大括号里面的语句。
{
    b++;    //相当于 b=b+1, 用来统计除了 70 到 80 以外的总数
}

if (x2<70|| x2>80)    //如果条件为真, 则执行下面大括号里面的语句。
{
    b++;    //相当于 b=b+1, 用来统计除了 70 到 80 以外的总数
}

if (x3<70|| x3>80)    //如果条件为真, 则执行下面大括号里面的语句。
{
    b++;    //相当于 b=b+1, 用来统计除了 70 到 80 以外的总数
}

if (x4<70|| x4>80)    //如果条件为真, 则执行下面大括号里面的语句。
{
    b++;    //相当于 b=b+1, 用来统计除了 70 到 80 以外的总数
}

```

```

    if(x5<70||x5>80) //如果条件为真，则执行下面大括号里面的语句。
    {
        b++; //相当于 b=b+1，用来统计除了 70 到 80 以外的总数
    }

    if(x6<70||x6>80) //如果条件为真，则执行下面大括号里面的语句。
    {
        b++; //相当于 b=b+1，用来统计除了 70 到 80 以外的总数
    }

    if(x7<70||x7>80) //如果条件为真，则执行下面大括号里面的语句。
    {
        b++; //相当于 b=b+1，用来统计除了 70 到 80 以外的总数
    }

    if(x8<70||x8>80) //如果条件为真，则执行下面大括号里面的语句。
    {
        b++; //相当于 b=b+1，用来统计除了 70 到 80 以外的总数
    }

    View(a); //把第 1 个数 a 发送到电脑端的串口助手软件上观察。
    View(b); //把第 2 个数 b 发送到电脑端的串口助手软件上观察。

    while(1)
    {
    }
}

/*---C 语言学习区域的结束。-----*/

```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第 1 个数

十进制:2

十六进制:2

二进制:10

第 2 个数

十进制:6

十六进制:6

二进制:110

分析：

变量 a 为 2。（数值从 70 到 80 之间的有 x4, x7 这 2 个）

变量 b 为 6。（除了 70 到 80 之间以外的有 x1, x2, x3, x5, x6, x8 这 6 个）

通过实验结果，发现在单片机上的实验结果和我们的分析是一致的。

#### 【43.6 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。

## 第四十四节：小括号改变判断优先级。

### 【44.1 小括号的概述。】

小括号在 C 语言里一直有“强制、改变优先级、明确顺序”这层含义。C 语言中，凡是在判断语句里插入了小括号，程序就会优先执行最里面小括号的判断语句，之后才会根据判断符的优先级执行其它相关语句。此功能很实用，因为 C 语言的判断符号众多，非常不利于程序员记忆各种不同符号的优先级顺序，而小括号却解决了这个问题。只要在合适的地方插入恰当的小括号，就可以强制改变判断的优先级，有了此项功能就不用程序员再刻意去记忆繁杂的优先级，同时，也为实际项目带来两个好处，一个是明确判断顺序，另一个是改变判断顺序。多说一句，哪怕添加的小括号是多余的重复的啰嗦的，也不会对程序带来副作用，反而，只会给程序员内心带来更大的确定和安全感。比如：

两个 if 条件判断语句：

```
if(a>=70&&a<=80) 和 if(a<70 || a>80)
```

有一些朋友喜欢插入两个小括号变成：

```
if((a>=70)&&(a<=80)) 和 if((a<70) || (a>80))
```

在这里插入的小括号是多余的重复的啰嗦的，但是还好，不会对程序有副作用。上述的修改，在不知道“>、>=、<、<=”这类语句跟“&&、||”这类语句哪个优先级更高的前提下，插入了小括号，可以更加明确判断的顺序，这种做法也值得肯定。

### 【44.2 小括号的具体应用。】

我个人平时在面对同时存在“>、>=、<、<=”和“&&、||”这些语句时，由于我很清楚“>、>=、<、<=”比“&&、||”这类语句的优先级更高，所以我不需要在此插入小括号来明确判断的顺序。但是遇到下面这种情况，我是一定会通过插入小括号的方式来明确判断的顺序。什么情况呢？如下：

```
if(“判断条件 1” || “判断条件 2 ” && “判断条件 3”)
```

这种情况下，就会很容易让我出现一个疑问，到底是先“判断条件 1”跟“判断条件 2”相“或”，最后再跟“判断条件 3”相“与”？还是先“判断条件 2”跟“判断条件 3”相“与”，最后再跟“判断条件 1”相“或”？如果此时果断插入小括号，就可以很容易明确它们的先后顺序，减少内心不必要的纠结。

插入小括号的第 1 种情况：

```
if((“判断条件 1” || “判断条件 2 ”) && “判断条件 3”)
```

插入小括号的第 2 种情况：

```
if(“判断条件 1” || (“判断条件 2 ” && “判断条件 3”))
```

上述两种情况，具体选择哪一种判断顺序要根据项目的需要来决定。同样的 3 个“判断条件”，如果插入的小括号的位置不一样，判断的顺序就不一样，那么结果也可能出现不一样，比如，上述判断条件：

假设“判断条件 1”为“真”，

假设“判断条件 2”为“真”，

假设“判断条件 3”为“假”，

等效成如下：

插入小括号的第 1 种情况：

```
if( (真||真) &&假)
{
    语句 1;
}
```

这种情况下，先判断最里面小括号的真假，(真||真)的结果是“真”，然后再把结果“真”和外面的“假”进行“与”判断，(真&&假)的结果是“假”，所以上述的最终判断是“假”，不能执行“语句 1”。

插入小括号的第 2 种情况：

```
if(真|| (真&&假))
{
    语句 1;
}
```

这种情况下，先判断最里面小括号的真假，(真&&假)的结果是“假”，然后再把结果“假”和外面的“真”进行“或”判断，(真||假)的结果是“真”，所以上述的最终判断是“真”，能执行“语句 1”。

综合上述两种情况，对比之后，得出这样的结论：在同样的条件和关系下，如果插入不同位置的小括号，就可以得出不同的结果。也就是说，小括号可以让关系判断变得丰富起来，可以实现各种复杂的逻辑判断功能。

### 【44.3 例程练习和分析。】

现在编写一个实验程序验证上述两种判断顺序。

程序代码如下：

```
/*---C 语言学习区域的开始。-----*/

//x, y 这三个变量作为条件判断的变量
unsigned char x=5;
unsigned char y=6;

//a, b 这两个变量作为输出判断结果的真假，0 代表假，1 代表真。
unsigned char a=0; //默认为 0，也就是默认为假
unsigned char b=0; //默认为 0，也就是默认为假

void main() //主函数
{
    if((x<y||y>x)&&x==y) //里面的条件是((真||真)&&假)，最终结果判断是假
    {
        a=1;
    }

    if(x<y|| (y>x&&x==y)) //里面的条件是(真||(真&&假))，最终结果判断是真
```

```

    {
        b=1;
    }

    View(a); //把第 1 个数 a 发送到电脑端的串口助手软件上观察。
    View(b); //把第 2 个数 b 发送到电脑端的串口助手软件上观察。

    while(1)
    {
    }
}

/*---C 语言学习区域的结束。-----*/

```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第 1 个数

十进制:0

十六进制:0

二进制:0

第 2 个数

十进制:1

十六进制:1

二进制:1

分析：

变量 a 为 0。（0 代表此条件判断结果为假）

变量 b 为 1。（1 代表此条件判断结果为真）

通过实验结果，发现在单片机上的实验结果和我们的分析是一致的。

#### 【44.4 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。

## 第四十五节： 组合判断 if...else if...else。

### 【45.1 三类组合判断语句的概述。】

if 的英文含义是“如果”，else 是“否则”，else if 是“否则如果”。在 C 语言里，if, else if, else 所表达的含义，跟英文也是一样的。

if, else if, else 可以组成三种组合判断语句，第一种是“二选一”，第二种是“多选一”，第三种是“多选一或者什么都不选”。

这类组合判断语句可以这样解读：在众多条件判断中，先从第一个 if 条件开始判断，如果第一个 if 条件是真，那么不管后面的条件是否为真，都不再判断，直接执行第一个 if 条件后面大括号的语句，组合语句中其它剩余的条件不再判断直接跳过，否则，就挨个条件往下判断，只要其中一个条件满足，就不再判断剩余的条件，也就是我们日常所说的多选一，甚至在某些组合语句如果所有条件都不满足，那么什么也不选。总之，在如此众多的条件中，最多只能执行一个条件后面大括号的语句。组合语句还有一个规律：if 语句只能出现在第一个条件判断，而且只能出现一次；else 只能出现在最后，而且也只能出现一次；而 else if 语句总是出现在中间，绝对不能出现在第一个条件判断，如果没有 else，也可以出现在最后的条件判断。多说一句，在上述所提到的“只能出现一次”的概念仅仅局限于在一个组合判断语句的范围内，而组合判断语句在整个程序的出现次数是不受限制的。

### 【45.2 二选一的组合判断。】

先讲第一种“二选一”的书写格式，如下：

书写格式如下：

```
if(条件 1)    //if 只能出现第一个条件，并且只能出现一次
{
    语句 1;
}
else          //else 只能出现最后，并且也只能出现一次。
{
    语句 2;
}
语句 3;
```

这类语句的书写特点是：第一个是 if 判断语句，最后一个是 else 语句，中间没有 else if 判断语句。

这类语句的执行顺序是：先判断第一个的 if 里面的（条件 1），如果（条件 1）满足而为真，就执行该（条件 1）后面紧跟的大括号里面的“语句 1”，执行完该大括号内的所有语句之后，就直接跳出整个组合判断的语句，不再判断也不再执行剩下来的 else 那部分的代码，直接跳到“语句 3”处，从“语句 3”处（包括“语句 3”）继续往下执行。但是，如果第一个的 if 里面的（条件 1）不满足而为假，那么就执行 else 后面大括号内的语句。也就是说，else 是在 if 条件不满足时才执行的，所以叫“二选一”，在 if 和 else 之间二选一。

### 【45.3 多选一的组合判断。】

接着讲第二种书写格式的“多选一”，这种书写格式，跟第一种对比，是在 if 与 else 的中间多插入了 N 个 else if 的判断语句。书写格式如下：

```
if(条件 1)    //if 只能出现第一个条件，并且只能出现一次
```



```

{
    语句 1;
}
else if(条件 2)    //else if 只能出现中间或最后，可以出现多次
{
    语句 2;
}
...
else if(条件 N)    //else if 只能出现中间或最后，可以出现多次
{
    语句 N;
}
else                //else 只能出现最后，并且也只能出现一次。
{
    语句 N+1;
}
语句 N+2;

```

这类语句的书写特点是：第一行是 if 开始，最后一行以 else 结束，中间是 N 个 else if 判断语句。

这类语句的执行顺序是：跟第一种“二选一”对比，判断顺序和规律大致也是一样的，也是从第一个 if 开始，往下逐个判断，然后到中间的 else if，只要发现一个条件满足，就执行该条件后面的大括号内的代码，之后就马上结束整个组合判断语句，不再判断剩下的组合判断语句。但是，如果万一前面第一个 if 和中间所有的 else if 的条件都不满足而为假，就直接执行最后一个 else 大括号内的语句。所以叫“多选一”，在“第一个 if、中间的 else if、最后一个 else”之间多选一。

#### 【45.4 多选一或者什么都不选的组合判断。】

最后讲第三种书写格式的“多选一或者什么都不选”，这种书写格式，跟第二种对比，只有第一个 if 和其它的 else if 语句，没有最后那个 else 语句。书写格式如下：

```

if(条件 1)                //if 只能出现第一个条件，并且只能出现一次
{
    语句 1;
}
else if(条件 2)    //else if 只能出现中间或最后，可以出现多次
{
    语句 2;
}
...
else if(条件 N)    //else if 只能出现中间或最后，可以出现多次
{
    语句 N;
}
语句 N+1;

```

这类语句的书写特点是：第一行是 if 开始，中间是 N 个 else if 判断语句，没有最后一个 else 语句。

这类语句的执行顺序是：跟第二种“多选一”对比，判断顺序和规律大致也是一样的，也是从第一个 if

开始，往下逐个判断，然后到中间的 else if，只要发现一个条件满足，就执行该条件后面的大括号内的代码，之后就马上结束整个组合判断语句，不再判断剩余的组合判断语句。但是，如果万一前面第一个 if 和中间所有的 else if 的条件都不满足而为假，因为此时没有 else 语句，就意味着整个组合判断语句都没有条件满足，因此就没有相关满足的代码被执行到。所以把这种情况称为“多选一或者什么都不选”。

### 【45.5 例程练习和分析。】

现在编写一个实验程序。

程序代码如下：

```
/*---C 语言学习区域的开始。-----*/

//x 这个变量作为条件判断的变量
unsigned char x=5;

//a,b,c 这 3 个变量作为输出判断结果,0 代表什么语句都没执行，1 代表执行了语句 1，
//2 代表执行语句 2，3 代表执行语句 3。
unsigned char a=0;
unsigned char b=0;
unsigned char c=0;

void main() //主函数
{

    //第一种“二选一”
    if(x>6)
    {
        a=1;    //1 代表执行了“语句 1”
    }
    else
    {
        a=2;    //2 代表执行了“语句 2”
    }

    //第二种“多选一”
    if(x>6)
    {
        b=1;    //1 代表执行了“语句 1”
    }
    else if(7==x)
    {
        b=2;    //2 代表执行了“语句 2”
    }
    else
```

```

    {
        b=3;  //3 代表执行了 “语句 3”
    }

    //第三种 “多选一或者什么都不选”
    if(x>6)
    {
        c=1;  //1 代表执行了 “语句 1”
    }
    else if(7==x)
    {
        c=2;  //2 代表执行了 “语句 2”
    }
    else if(8==x)
    {
        c=3;  //3 代表执行了 “语句 3”
    }

    View(a);  //把第 1 个数 a 发送到电脑端的串口助手软件上观察。
    View(b);  //把第 2 个数 b 发送到电脑端的串口助手软件上观察。
    View(c);  //把第 3 个数 c 发送到电脑端的串口助手软件上观察。

    while(1)
    {
        }
    }

/*---C 语言学习区域的结束。-----*/

```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第 1 个数

十进制:2

十六进制:2

二进制:10

第 2 个数

十进制:3

十六进制:3

二进制:11

第 3 个数

十进制:0

十六进制:0

二进制:0

分析:

变量 a 为 2。(2 代表执行了语句 2)

变量 b 为 3。(3 代表执行了语句 3)

变量 c 为 0。(0 代表什么语句都没执行)

#### 【45.6 如何在单片机上练习本章节 C 语言程序?】

直接复制前面章节中第十一节的模板程序,练习代码时只需要更改“C 语言学习区域”的代码就可以了,其它部分的代码不要动。编译后,把程序下载进带串口的 51 学习板,通过电脑端的串口助手软件就可以观察到不同的变量数值,详细方法请看第十一节内容。

## 第四十六节： 一维数组。

### 【46.1 数组是什么？】

数组就是一堆变量或常量的集合。把一个数组里面某一个变量或者常量称为数组的元素，反过来也可以这么说，元素的集合就是数组。数组的最大特点就是内部所有的元素的地址都是挨家挨户相连的，同花顺似的，以第一个元素（下标是 0 的元素）为首地址，后来元素的地址挨个依次增大。首地址在 RAM 中的绝对地址往往是编译器自动分配的，我们不用管，可以看成是随机的。多说一句，在某些单片机，也可以通过特定的 C 语言关键词，强制要求编译器按我们的意愿，来分配到 RAM 中指定的某个绝对地址，这部分的内容这里暂时不讲。继续刚才的话题，首地址就像是一个坐标原点，一旦被编译器确定下来它在 RAM 中的地址，那么后面其它元素的地址都是在此基础上依次增大的，有规律的。正因为这个特点，数组在项目中往往起到缓存的作用。比如，在通信的项目中，用来作为一串数据的接收缓存。在界面显示的项目中，某个 16x16 点阵汉字的字模，需要一个内含 32 个元素的数组来作为缓存。在读写文件的项目中，也需要一个大数组来作为文件内容的缓存。在某些涉及复杂算法的项目，以数组作为缓存，并且通过配合循环语句或者指针，就可以快速批量的处理数据（循环语句和指针的相关知识后面章节会讲到）。总之，在项目应用中，数组无处不在。

数组分为一维数组，二维数组，三维数组。一维数组应用最广，二维数组其次，三维数组最少用。所以本教程只讲一维数组和二维数组，本节先讲一维数组。

### 【46.2 一维数组的书写格式和特点。】

一维数组不带初始化时候的定义格式如下：

```
数据类型 数组名[数组元素总数 N];
```

数据类型是指 unsigned char, unsigned int, unsigned long 这类关键词；数组名就是由字母和数字组合而成的字符串，遵循常用变量的命名规则；N 是数字，代表此数组内部有多少个元素。比如：

```
unsigned char x[3]; //这里的 3 是数组内部元素的总数，但不是下标。
```

上述这一行代码，就相当于一条语句定义了 3 个变量，这 3 个变量分别是 x[0], x[1], x[2]，但是不存在 x[3] 这个变量。这里，具体元素中括号内的“0, 1, 2”称为数组的下标，代表某个具体的元素。由此可见，数组有“批量定义”的特点。同时也要发现，此数组定义的 N 是 3，代表内含 3 个元素变量，但是具体到某个元素的时候，下标不是从 1 开始，而是从 0 开始，最后一个也不是 3 而是 2。可以这样描述，某个数组有 N 个元素，它具体元素的下标是从 0 开始，到 N-1 结束。那么问题来了，如果一个数组明明最大只有 N 个元素，但是我在操作某个具体的元素时，非要用下标 N 或者 N+1，也就是说，如果超过数组的范围的操作，会出现什么问题？后果严重吗？答案是：会导致数组越界出现异常或者编译不通过，可能会破坏其它数据，后果是严重的。因此大家使用数组的时候，要注意数组不能越界的问题。

刚刚讲了一维数组不带初始化的定义格式，现在接着讲带初始化的定义格式，如下：

```
数据类型 数组名[数组元素总数 N]={元素 0, 元素 1, ...元素 N-1};
```

比如：

```
unsigned char y[3]={10, 11, 12};
```

此数组一行代码定义了 3 个变量，分别是 y[0], y[1], y[2]。而 y[0] 初始化为 10, y[1] 初始化为 11, y[2] 初始化为 12。

在程序中，操作数组某个变量元素时，下标可以是常量，比如 y[0], 此时的 0 就是常量；下标也可以是变量，比如 y[i], 此时的 i 就是变量。再强调一次，作为下标的常量或者变量 i 的数值必须小于数组定义时的元素个数，否则就会导致数组越界出现异常或者编译不通过。

中括号内的 N 什么时候是“数组的元素总数”，什么时候是“数组的元素下标”，这个问题对初学者很容易混淆。其实很简单，定义的时候是“数组的元素总数”，操作调用具体某个元素的时候是“数组的元素下标”。

### 【46.3 什么情况下可以省略定义的元素总数？】

一维数组在定义时，如果预先给它填写若干个初始化的数据，在语法上，也可以省略中括号[N]里面的元素总数 N，这样编译器在编译时会根据初始化的总数来自动识别和定义此一维数组实际元素总数，分配对应数量的内存 RAM。比如：

```
unsigned char y[3]={10,11,12}; //没有省略元素总数的写法
```

跟

```
unsigned char y[]={10,11,12}; //在初始化的情况下，省略了元素总数的写法。
```

的意义是一样的，都是合法的，都是 C 语言所允许的。注意，省略元素个数时必须要有初始化的数据，否则，编译器不知道此数组的长度，可能导致编译出错。

这个功能在实际应用中有什么作用呢？在实际应用中，此项功能一般会用在常量数组里，而不是变量的数组里。当在数组定义的前面加上“const”或者“code”（针对 51 单片机）的关键词时，原来“变量”的数组就会变成“常量”的数组，这时，如果把常量的数组用来作为某个转换表格，此功能就很实用。因为作为转换表格的常量数组，我们在编程程序的过程中，有可能随时往里面添加数组，这个时候，不用我们刻意去计算和调整数组的元素总数 N，给我们写程序带来了便利。对于这个功能的应用，大家先有一个感性的认识即可，暂时不用深入去了解，因为后续的章节还会讲解这方面的内容。

### 【46.4 例程练习和分析。】

现在编写一个程序来熟悉一下一维数组的使用。

程序代码如下：

```
/*---C 语言学习区域的开始。-----*/

unsigned char x[3]; //此处的 3 不是下标，而是元素总数，里面的 3 个变量没有初始化
unsigned char y[3]={10,11,12}; //里面三个元素变量 y[0],y[1],y[2] 分别初始化为 10,11,12
unsigned char i=0; //定义和初始化一个变量。用来做 x 数组的下标。

void main() //主函数
{

    x[i]=25; //此时下标 i 为 0. 相当于把 25 赋值给 x[0]
    i=i+1;   //i 由 0 变成 1.
    x[i]=26; //此时下标 i 为 1. 相当于把 26 赋值给 x[1]
    i=i+1;   //i 由 1 变成 2.
    x[i]=27; //此时下标 i 为 2. 相当于把 27 赋值给 x[2]
    x[i]=x[i]+1; //此时 x[2] 自加 1 变成了 28
```

```
View(x[0]); //把第 1 个数 x[0]发送到电脑端的串口助手软件上观察。
View(x[1]); //把第 2 个数 x[1]发送到电脑端的串口助手软件上观察。
View(x[2]); //把第 3 个数 x[2]发送到电脑端的串口助手软件上观察。
View(y[0]); //把第 4 个数 y[0]发送到电脑端的串口助手软件上观察。
View(y[1]); //把第 5 个数 y[1]发送到电脑端的串口助手软件上观察。
View(y[2]); //把第 6 个数 y[2]发送到电脑端的串口助手软件上观察。

while(1)
{
}
}

/*---C 语言学习区域的结束。-----*/
```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第 1 个数

十进制:25

十六进制:19

二进制:11001

第 2 个数

十进制:26

十六进制:1A

二进制:11010

第 3 个数

十进制:28

十六进制:1C

二进制:11100

第 4 个数

十进制:10

十六进制:A

二进制:1010

第 5 个数

十进制:11

十六进制:B

二进制:1011

第 6 个数

十进制:12

十六进制:C  
二进制:1100

分析:

变量元素 x[0]为 25。

变量元素 x[1]为 26。

变量元素 x[2]为 28。

变量元素 y[0]为 10。

变量元素 y[1]为 11。

变量元素 y[2]为 12。

#### 【46.5 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。



## 第四十七节： 二维数组。

### 【47.1 二维数组的书写格式和特点。】

拿一维数组和二维数组来对比一下，一维数组只有一个下标，像由很多点连成的一条直线，整体给人的是一种“线”的观感。而二维数组有两个下标，这两个下标类似平面中的行与列，也类似平面中的 X 轴和 Y 轴的坐标，通过 y 轴和 x 轴坐标就可以找到所需的点，也就是二维数组的某个元素，因此，二维数组整体给人的是一种“面”的观感。

上述是对二维数组的感性描述，二维数组是由一维数组发展而来，所以继承了很多一维数组的特点。二维数组的所有“网点”元素的地址都是挨个相临的，先从第 0 行开始“扫描”当前行的列，第 0 行第 0 列，第 0 行第 1 列，第 0 行第 2 列.....再第 1 行第 0 列，第 1 行第 1 列，第 1 行第 2 列.....再第 2 行.....再第 N 行，上一行“尾”元素跟下一行“头”元素的地址也是相临连续的。

二维数组未带初始化时的通用定义格式如下：

```
数据类型 数组名[行数 Y][列数 X];
```

比如：

```
unsigned char a[2][3]; //此处的 2 代表有 2 行，3 代表有 3 列。
```

分析：此二维数组定义了 6 个变量，跟一维数组一样，下标都是从 0 开始，到(N-1)时结束，此处的 N 代表行数或者列数。所以 a[2][3]数组的元素挨个分别是 a[0][0], a[0][1], a[0][2], a[1][0], a[1][1], a[1][2]这 6 个变量。这 6 个变量的地址是顺序挨个相连的。

二维数组有两种常用初始化格式，一种是逐行初始化，一种是整体初始化。

第一种逐行初始化：

```
unsigned char a[2][3]=
{
    {0, 1, 2},
    {3, 4, 5}
};
```

在逐行初始化定义二维数组时，只要有初始化的数据，也可以省略行下标，但是列下标不能省略，比如：

```
unsigned char a[][3]=
{
    {0, 1, 2},
    {3, 4, 5}
};
```

此时编译器会根据元素的个数来确定行数是多少。

第二种整体初始化，跟一维数组一样，内部数据元素不需要额外增加大括号来分行。

```
unsigned char a[2][3]=
{
    0, 1, 2, 3, 4, 5
};
```

或者

```

unsigned char a[2][3]=
{
    0, 1, 2,
    3, 4, 5
};

```

都行。

C 语言是很丰富的语言，比如二维数组还允许不完全初始化的一些情况，这种情况我就不再深入讲解，我讲解的都是挑选一些针对以后单片机项目中可能会经常用到的语法。

二维数组我在很多项目上还是经常用到的，比如用在一些需要把所得的信息进行查表判断的项目，在每一行里放一条关键词字符串信息，利用循环语句进行逐行查找匹配信息。至于二维数组如何存放字符串的知识点以后再讲。这节的重点是让大家对二维数组有个初步的认识。

## 【47.2 例程练习和分析。】

现在编写一个程序来熟悉一下二维数组的书写和使用格式。

程序代码如下：

```

/*---C 语言学习区域的开始。-----*/

unsigned char  a[2][3]= //定义和初始化一个二维数组
{
    {0, 1, 2},
    {3, 4, 5}
};

void main() //主函数
{
    a[0][0]=8; //故意把第 0 行第 0 列的这个变量赋值 8，让大家看看如何直接操作二维数组某个元素。

    View(a[0][0]); //把第 1 个数 a[0][0]发送到电脑端的串口助手软件上观察。
    View(a[0][1]); //把第 2 个数 a[0][1]发送到电脑端的串口助手软件上观察。
    View(a[0][2]); //把第 3 个数 a[0][2]发送到电脑端的串口助手软件上观察。
    View(a[1][0]); //把第 4 个数 a[1][0]发送到电脑端的串口助手软件上观察。
    View(a[1][1]); //把第 5 个数 a[1][1]发送到电脑端的串口助手软件上观察。
    View(a[1][2]); //把第 6 个数 a[1][2]发送到电脑端的串口助手软件上观察。

    while(1)
    {
    }
}

/*---C 语言学习区域的结束。-----*/

```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第 1 个数

十进制:8

十六进制:8

二进制:1000

第 2 个数

十进制:1

十六进制:1

二进制:1

第 3 个数

十进制:2

十六进制:2

二进制:10

第 4 个数

十进制:3

十六进制:3

二进制:11

第 5 个数

十进制:4

十六进制:4

二进制:100

第 6 个数

十进制:5

十六进制:5

二进制:101

分析:

变量元素 a[0][0]为 8。从原来定义的 0 变成 8，因为被 main 函数里的第 1 行代码赋值了 8。

变量元素 a[0][1]为 1。

变量元素 a[0][2]为 2。

变量元素 a[1][0]为 3。

变量元素 a[1][1]为 4。

变量元素 a[1][2]为 5。

### 【47.3 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观

察到不同的变量数值，详细方法请看第十一节内容。

## 第四十八节： while 循环语句。

### 【48.1 程序的“跑道”。】

经常听到这句话“程序跑起来了吗？”，程序在哪里跑？有跑道吗？有的。循环语句就像一条椭圆的跑道，程序在跑道上不停的跑，不知疲倦的跑，永无止境，一秒钟几百万圈的速度。单片机的 main 主函数内往往有一条 while(1) 语句，这就是单片机的“循环跑道”，称之为主循环，主循环内还可以继续嵌套多层 while 循环语句。

### 【48.2 while 循环的常见格式。】

常见格式如下：

```
while(条件)
{
    语句 1;
    语句 2;
    .....
    语句 N;
}
语句 N+1;
```

上面的“花括号内”称为“循环体内”，“花括号外”称为“循环体外”，现在来分析一下上述代码的执行规律，如下：

(1) 像 if 语句一样，先判断 while 的(条件)是否为真。如果为“假”，就不执行循环体“内”的“语句 1”至“语句 N”，直接跳到循环体“外”的“语句 N+1”处开始往下执行。如果为“真”，才执行循环体“内”的“语句 1”至“语句 N”，当执行完循环体“内”最后的“语句 N”时，单片机会突然返回到第一行代码“while(条件)”处，继续判断循环的(条件)是否为真，如果为假就跳到循环体“外”的“语句 N+1”，表示结束了当前循环。如果为真就继续从“语句 1”执行到“语句 N”，然后再返回 while(条件)处，依次循环下去，直到条件为假时才罢休，否则一直循环下去。

(2) while(条件)语句中，条件判断真假的规则跟 if 语句一模一样，有两种类型：一种是纯常量或者变量类型的，只要此数值不等于 0 就认为是真，所以 while(1) 也称死循环语句，因为里面的条件永远不为 0。对于死循环这样的语句，如果不遇到 break, return, goto 这些语句，那么就永远也别想跳出这个循环；另外一种关系判断，以及关系语句之间的像“与或”关系这类的判断。这些条件判断的真假，跟 if 语句的规则是一样的，这里不再多讲。break, return, goto 这些语句后面章节会讲到。

### 【48.3 while 省略花括号，没带分号。】

```
while(条件)
    语句 1;
    语句 2;
    .....
    语句 N;
    语句 N+1;
```

上面的代码，居然没有了花括号，问题来了，此循环语句的“有效射程”究竟是多远，或者说，此循环语句的循环区域在哪里。现在跟大家解开这个谜团。第一行代码，while(条件)后面“没有分号”，接着第二行就是“语句 1”，所以，这种情况跟 if 语句省略花括号的写法是一样的，此时循环体默认只包含离它最近的一条且仅仅一条的“语句 1”，因此，上述的语句，等效于下面这种添加花括号的写法：

```
while(条件)
{
    语句 1;
}
语句 2;
.....
语句 N;
语句 N+1;
```

#### 【48.4 while 省略花括号，带分号。】

```
while(条件);
    语句 1;
    语句 2;
    .....
    语句 N;
    语句 N+1;
```

这次的代码跟刚才“48.3”的代码唯一的差别是，第一行代码，while(条件)后面“有分号”。所以它循环的有效范围就在第一行就结束了，不涉及“语句 1”。此时，等效于下面这种添加花括号的写法：

```
while(条件)
{
    ;    //这里的分号代表一条空语句
}
语句 1;
语句 2;
.....
语句 N;
语句 N+1;
```

如果 while 的（条件）一直为“真”，单片机就一直在循环体内执行一条“无意义”的空语句，相当于“耗着”的状态，执行不到后面“语句 1”的语句，除非，条件为“假”才罢休才会跳出循环体。

循环体内什么都没有，只写一条“空语句”，这种写法在实际项目中也是有用武之地的，比如，等待某件事是否满足条件，如果不满足，就一直死等死磕在这里，其它事情都干不了，这种“死等死磕”的做法，专业术语叫“阻塞”，与之反面相对应的是另外一个词叫“非阻塞”。对于循环的“阻塞”用法，老练的工程师通常会多加一个超时的判断，这些内容大家暂时不用深入了解，后续章节我会讲到。

#### 【48.5 例程练习和分析。】

现在编写一个程序来熟悉一下 while 语句的书写和使用格式。

程序代码如下：

```

/*---C 语言学习区域的开始。-----*/

unsigned char a=0; //观察这个数最后的变化
unsigned char b=0; //观察这个数最后的变化

unsigned char i;    //控制循环体的条件判断变量

void main() //主函数
{
    i=3;
    while(i) //i 不断减小，直到变为 0 时才跳出此循环体
    {
        a=a+1; //当 i 从 3 减少到 0 的时候，这条语句被循环执行了 3 次。
        i=i-1; //循环的条件不断发生变化，不断减小
    }

    i=0;
    while(i<3) //i 不断增大，当 i 大于或者等于 3 时才跳出此循环体
    {
        b=b+2; //当 i 从 0 增加到 3 的时候，这条语句被循环执行了 3 次。
        i=i+1; //循环的条件不断发生变化，不断增加
    }

    View(a); //把第 1 个数 a 发送到电脑端的串口助手软件上观察。
    View(b); //把第 2 个数 b 发送到电脑端的串口助手软件上观察。

    while(1)
    {
    }
}

/*---C 语言学习区域的结束。-----*/

```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第 1 个数

十进制:3

十六进制:3

二进制:11

第 2 个数

十进制:6

十六进制:6

分析:

变量 a 为 3。a 初始化为 0，进入循环体内后，a 每次加 1，循环加 3 次，因此从 0 变成了 3。

变量 b 为 6。b 初始化为 0，进入循环体内后，b 每次加 2，循环加 3 次，因此从 0 变成了 6。

#### 【48.6 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。



## 第四十九节： 循环语句 do while 和 for。

### 【49.1 do while 语句的常见格式。】

格式如下：

```
do
{
    语句 1;
    语句 2;
    .....
    语句 N;
} while(条件);
```

上述代码，单片机从上往下执行语句，先从 do 那里无条件进来，从“语句 1”开始往下执行，一直执行到“语句 N”，才开始判断 while(条件)的条件是否为真，如果为真继续返回到 do 的入口处，继续从“语句 1”开始往下执行，依次循环。大家留意到了吗，do while 和 while 语句有什么差别？差别是，do while 是先无条件进来执行一次循环体（花括号里所有的程序代码），执行到循环体最底部才判断 while(条件)的条件是否为真来决定是否继续循环，先上车再买票。而 while 语句是先判断条件是否为真再决定是否需要进入循环体，先买票再上车。

### 【49.2 for 语句的简介。】

for 语句也是循环语句，任何 for 语句能实现的功能都可以用 while 语句来实现同样的功能，for 语句和 while 语句有什么差别呢？for 语句把变量初始化，变量的条件判断，变量在执行循环体后的步进变化这三个常见要素集成在语句内部，以标准的格式书写出来。在很多场合下，for 在书写和表达方面比 while 语句显得更加简洁和直观。

### 【49.3 for 语句的自加格式。】

格式如下：

```
for(变量的初始化语句; 变量的条件判断;变量在执行一次循环体后自加的步进变化)
{
    语句 1;
    语句 2;
    .....
    语句 N;
}
```

在把上述变成更具体的代码例程如下：

```
for(i=0; i<3;i++)
{
    语句 1;
    语句 2;
    .....
```

```
    语句 N;  
}
```

上述代码，单片机从上往下，在进入循环体前，先把变量 *i* 初始化赋值 0（这行初始化代码在整个循环期间只被执行 1 次），然后判断 *i* 是否小于 3 这个条件，如果此条件为真，就开始正式进入循环体，从“语句 1”往下执行到“语句 N”，执行完一次循环体后，*i* 就自加 1（因为“*i++*”语句），此时 *i* 从原来初始化的 0 变成了 1，接着再返回来到 for 语句的条件判断“*i*<3”那里，判断 *i* 是否继续满足“小于 3”这个条件，如果此条件为真就继续往下执行，否则就跳过循环体结束当前循环。上述 for 语句实现的功能如果用 while 语句来写，等效于以下代码：

```
i=0; //进入循环体之前先初始化给予初值  
while(i<3)  
{  
    语句 1;  
    语句 2;  
    .....  
    语句 N;  
    i++; //执行一次循环体之后此变量自加发生变化  
}
```

上述的 while 循环语句只执行了 3 次循环体。

#### 【49.4 for 语句的自减格式。】

刚才讲的 for(*i*=0; *i*<3;*i++*)这种格式，它的变量 *i* 是不断自加的。还有一种比较常见的格式是 *i* 不断自减的，它的格式如下：

```
for(i=3; i>0;i--)  
{  
    语句 1;  
    语句 2;  
    .....  
    语句 N;  
}
```

上述自减的 for 语句功能如果用 while 语句来写，等效于以下代码：

```
i=3; //进入循环体之前先初始化给予初值  
while(i>0)  
{  
    语句 1;  
    语句 2;  
    .....  
    语句 N;  
    i--; //执行一次循环体之后此变量自减发生变化  
}
```

上述的 while 循环语句只执行了 3 次循环体。

#### 【49.5 for 省略花括号，没带分号。】

前面讲的 if 和 while 语句中，都提到了省略花括号的情况，for 语句也有这种写法，而且省略之后默认的有效范围都是一样的。请看例子如下：

```
for(i=0; i<3;i++)    //注意，这里没带分号。
    语句 1;
    语句 2;
    .....
    语句 N;
```

分析：上述代码，跟 if 语句一样，此时循环体默认只包含“语句 1”，等效于：

```
for(i=0; i<3;i++)    //注意，这里没带分号。
{
    语句 1;
}
    语句 2;
    .....
    语句 N;
```

#### 【49.6 for 省略花括号，带分号。】

```
for(i=0; i<3;i++);    //注意，这里带分号。
    语句 1;
    语句 2;
    .....
    语句 N;
```

分析：注意，此时循环体默认不包含“语句 1”，而是等效于：

```
for(i=0; i<3;i++)
{
    ;    //空语句。
}
    语句 1;
    语句 2;
    .....
    语句 N;
```

此时循环体内先循环执行三次空语句，然后才会结束 for 循环，接着才从“语句 1”开始往下执行。

#### 【49.7 for 循环语句的条件判断。】

上面举的例子中，仅仅列出了 for 语句条件判断的小于号关系符“<”，其实，for 语句条件判断的关系符跟 if 语句是一样通用的，凡是 if 语句能用的关系符都可以用在 for 语句上，比如“>”，“!=”，“==”，“<=”，“>=”等等。如下：

```
for(i=0;i<=3;i++); //小于等于的情况。这种写法是合法的。
for(i=0;i!=3;i++); //不等于的情况。这种写法是合法的。
for(i=0;i==3;i++); //等于的情况。这种写法是合法的。
```

## 【49.8 例程练习和分析。】

编写一个程序来熟悉一下 do while 和 for 语句的使用。

程序代码如下：

```
/*---C 语言学习区域的开始。-----*/

unsigned char a=0; //观察这个数最后的变化
unsigned char b=0; //观察这个数最后的变化
unsigned char c=0; //观察这个数最后的变化

unsigned char i; //控制循环体的条件判断变量

void main() //主函数
{
    i=3;
    do
    {
        a=a+1; //每执行一次循环体 a 就增加 1, 此行代码被循环执行了 3 次
        i=i-1; //i 不断变小
    }while(i); //i 不断变小，当 i 变为 0 时才跳出此循环体

    for(i=0;i<3;i++)
    {
        b=b+2; //此行代码被循环执行了 3 次
    }

    for(i=3;i>0;i--)
    {
        c=c+3; //此行代码被循环执行了 3 次
    }

    View(a); //把第 1 个数 a 发送到电脑端的串口助手软件上观察。
    View(b); //把第 2 个数 b 发送到电脑端的串口助手软件上观察。
    View(c); //把第 3 个数 c 发送到电脑端的串口助手软件上观察。
```

```
        while(1)
        {
        }
}

/*---C 语言学习区域的结束。-----*/
```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第 1 个数

十进制:3

十六进制:3

二进制:11

第 2 个数

十进制:6

十六进制:6

二进制:110

第 3 个数

十进制:9

十六进制:9

二进制:1001

分析：

变量 a 为 3。a 从 0 开始，循环加 1，一共 3 次，因此等于 3。

变量 b 为 6。b 从 0 开始，循环加 2，一共 3 次，因此等于 6。

变量 c 为 9。c 从 0 开始，循环加 3，一共 3 次，因此等于 9。

### 【49.9 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。

## 第五十节： 循环体内的 continue 和 break 语句。

### 【50.1 continue 语句。】

通常情况下，单片机在循环体里从第一行的“入口条件”开始往下执行，直至碰到循环体的边界“底部花括号”，才又折回到第一行的“入口条件”准备进行新一轮的循环。但是，若中途碰到 continue 语句，就会提前结束当前这一轮的循环，只要碰到 continue 语句，就立即折回到第一行的“入口条件”准备进行新一轮的循环。注意，continue 语句“结束”的对象仅仅是“当前这一轮的循环”，并没有真正结束这个循环的生命周期。它好像拦路虎，遇到它，它说“你回去，第二天再来。”这台词里的“第二天再来”就强调这个循环体的生命周期还没有真正结束。举一个具体的例子，如下：

```
while(...)或者 for(...)    //循环体的条件判断入口处
{    //循环体开始
    语句 1;
    语句 2;
    continue;
    语句 3;
    .....
    语句 N;
}    //循环体结束
```

分析：上述语句中，单片机从“循环体的条件判断入口处”开始往下执行，碰到 continue 就马上折回到“循环体的条件判断入口处”，继续开始新一轮的循环，因此，这段代码，continue 后面的“语句 3”至“语句 N”是永远也不会被执行到的。因为 continue 的拦截，上述语句等效于：

```
while(...)或者 for(...)    //循环体的条件判断入口处
{    //循环体开始
    语句 1;
    语句 2;
}    //循环体结束
```

问题来了，既然可以如此简化，还要 continue 干什么，不是多此一举？在实际应用中，continue 肯定不会像上面这样单独使用，continue 只有跟 if 语句结合，才有它存在的意义。例如：

```
while(...)或者 for(...)    //循环体的条件判断入口处
{    //循环体开始
    语句 1;
    语句 2;
    if(某条件)
    {
        continue;
    }
    语句 3;
    .....
    语句 N;
```

```
} //循环体结束
```

## 【50.2 break 语句。】

continue 语句提前结束当前这一轮的循环，准备进入下一轮的新循环，强调“某次结束”，但不是真结束。break 语句是直接跳出当前循环体，是真正的结束当前循环体，强调循环体的“生命结束”。举例如下：

```
while(...)或者 for(...) //循环体的条件判断入口处
{ //循环体开始
    语句 1;
    语句 2;
    break;
    语句 3;

    .....
    语句 N;
} //循环体结束
语句(N+1); //循环体之外语句
```

分析：上述语句中，单片机从“循环体的条件判断入口处”开始往下执行，突然碰到 break 语句，此时，立即无条件跳出当前循环体（无需判断 while 或者 for 的条件），直接执行到循环体之外的“语句(N+1)”，break 后面的“语句 3”至“语句 N”也没有被执行到。实际项目中，break 也往往会配合 if 一起使用，例如：

```
while(...)或者 for(...) //循环体的条件判断入口处
{ //循环体开始
    语句 1;
    语句 2;
    if(某条件)
    {
        break;
    }
    语句 3;

    .....
    语句 N;
} //循环体结束
语句(N+1); //循环体之外语句
```

## 【50.3 break 语句能跳多远？】

break 语句能跳多远？预知答案请先看以下例子：

```
while(...)
{
```

```

    语句 1;
    语句 2;
    while(...)
    {
        语句 3;
        break;
        语句 4;
    }
    语句 5;
}
语句 6;

```

分析：上述例子中，在 while 循环里面有藏着第二个 while 循环，像这种循环之中还有循环的情况，通常称为循环嵌套。单片机从上往下执行，当遇到 break 后，它会跳到“语句 5”那里呢，还是会跳到“语句 6”那里？正确答案是“语句 5”那里，这说明了 break 语句的“有效射程”仅仅刚好能跳出当前的循环体。也就是说，在上述循环嵌套的例子中，最内层的 break 只能跳出最内层的循环体，不能跳到最外层的“语句 6”那里，如果需要继续跳出最外层的“语句 6”那里，可以继续在外层的循环体内再增加一个 break 语句。

#### 【50.4 还有哪些语句可以无条件跳出循环体？】

除了 break 以外，还有 return 和 goto 语句可以跳出循环体。这部分的内容大家只需大概了解一下即可。return 语句比 break 语句还厉害，它不仅仅跳出当前循环体，还是跳出了当前函数，也就是提前结束了当前函数，这部分的内容后面章节会讲到，暂时不用管。而 goto 语句在 C 语言中大家都公认不建议用，因为它很容易扰乱大家常用的 C 语言编程结构，我本人也从来没有用过 goto 语句，因此不再深入讲解它。

#### 【50.5 例程练习和分析。】

编写一个程序来熟悉一下 continue 和 break 语句的使用。

程序代码如下：

```

/*---C 语言学习区域的开始。-----*/

unsigned char a=0; //观察这个数最后的变化
unsigned char b=0; //观察这个数最后的变化
unsigned char c=0; //观察这个数最后的变化
unsigned char d=0; //观察这个数最后的变化

unsigned char i; //控制循环体的条件判断变量
void main() //主函数
{
    //i<6 的条件判断是在进入循环体之前判断，而 i 的自加 1 是在执行完一次循环体之后才自加的。
    for(i=0;i<6;i++)
    {
        a=a+1; //被执行了 6 次，分别是第 0, 1, 2, 3, 4, 5 次
    }
}

```



```

        if(i>=3) //当 i 等于 3 的时候，开始“拦截”continue 后面的代码。
        {
            continue; //提前结束本次循环，准备进入下一次循环
        }
        b=b+1; //被执行了 3 次，分别是第 0, 1, 2 次
    }

//i<6 的条件判断是在进入循环体之前判断，而 i 的自加 1 是在执行完一次循环体之后才自加的。
for(i=0;i<6;i++)
{
    c=c+1; //被执行了 4 次，分别是第 0, 1, 2, 3 次
    if(i>=3) //当 i 等于 3 的时候，直接跳出当前循环体，结束此循环体的“生命周期”。
    {
        break; //马上跳出当前循环体
    }
    d=d+1; //被执行了 3 次，分别是第 0, 1, 2 次
}

View(a); //把第 1 个数 a 发送到电脑端的串口助手软件上观察。
View(b); //把第 2 个数 b 发送到电脑端的串口助手软件上观察。
View(c); //把第 3 个数 c 发送到电脑端的串口助手软件上观察。
View(d); //把第 4 个数 d 发送到电脑端的串口助手软件上观察。

while(1)
{
}
}

/*---C 语言学习区域的结束。-----*/

```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第 1 个数

十进制:6

十六进制:6

二进制:110

第 2 个数

十进制:3

十六进制:3

二进制:11

第 3 个数

十进制:4  
十六进制:4  
二进制:100

第 4 个数  
十进制:3  
十六进制:3  
二进制:11

分析:

变量 a 为 6。  
变量 b 为 3。  
变量 c 为 4。  
变量 d 为 3。

#### 【50.6 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。

## 第五十一节： for 和 while 的循环嵌套。

### 【51.1 循环的嵌套。】

大循环的内部又包含了小循环，称为循环嵌套。生活中，循环嵌套的现象很常见，一年 12 个月，假设每个月都是 30 天（仅仅假设而已），1 月份 30 天，2 月份 30 天……11 月份 30 天，12 月份 30，这里的年就是大循环，年内部的月就是小循环。一年 12 个月，大循环就是 12 次。一个月 30 天，小循环就是 30 次。用 for 语句来表达，大意如下：

```
for(m=1;m<=12;m++) //大循环。一年 12 个月。这里的 m 看作月，代表一年 12 个月的大循环。
{
    for(d=1;d<=30;d++) //内嵌小循环。一月 30 天。这里的 d 看作天，代表一个月 30 天的小循环。
    {

    }
}
```

### 【51.2 循环嵌套的执行顺序。】

例子如下：

```
for(i=0;i<2;i++) //大循环
{
    语句 1;
    for(k=0;k<3;k++) //内嵌的小循环
    {
        语句 2;
    }
    语句 3;
}
```

上述例子中，带 i 的 for 称为大循环，带 k 的 for 称为小循环，单片机从大循环入口进来，由上往下执行，执行第 1 次大循环，先执行 1 次“语句 1”，接着进入小循环，小循环要连续循环执行 3 次“语句 2”才跳出小循环，之后执行 1 次“语句 3”，然后再返回到大循环入口判断 i 条件是否满足，此时条件满足，继续执行第 2 次大循环，1 次“语句 1”，3 次“语句 2”，1 次“语句 3”，第 2 次循环结束后又返回到大循环入口判断 i 条件，此时 i 已经等于 2 不再小于 2 了，因此条件不满足，结束整个循环嵌套。上述执行的语句顺序如下：

```
语句 1; //第 1 次大循环开始
语句 2;
语句 2;
语句 2;
语句 3;
语句 1; //第 2 次大循环开始
语句 2;
语句 2;
语句 2;
语句 3;
```

根据此顺序，再看一个具体的程序例子：

```
a=0;
b=0;
for(i=0;i<2;i++) //大循环
{
    a=a+1; //被执行了 2 次
    for(k=0;k<3;k++) //内嵌的小循环
    {
        b= b+1; //被执行了 6 次
    }
}
```

上述例子中，执行完程序后，a 的值变成了 2，b 的值变成了 6。重点分析 b 的变化，“b=b+1”在内嵌循环体里被执行了 6 次，6 次从何而来？就是 i 乘以 k 等于 6。这个乘法次数是循环嵌套一个很重要的特性。上述程序如果用 while 语句来实现，等效如下：

```
a=0;
b=0;
i=0; //控制大循环的变量初始化
while(i<2) //大循环
{
    a=a+1; //被执行了 2 次
    k=0; //控制小循环的变量初始化
    while(k<3) //内嵌的小循环
    {
        b= b+1; //被执行了 6 次
        k=k+1;
    }
    i=i+1;
}
```

### 【51.3 循环嵌套的常见用途——二维数组的应用。】

二维数组 a[2][3]，它有 6 个变量，在没有学 for 语句之前，如果要依次把每个元素单独赋值清零真不容易，要写 6 次赋值语句如下：

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

自从懂了 for 嵌套语句之后，可以让同样功能的代码简洁许多。上述代码等效于如下：

```
for(i=0;i<2;i++) //大循环
{
    for(k=0;k<3;k++) //内嵌的小循环
```

```

    {
        a[i][k]=0;
    }
}

```

#### 【51.4 循环嵌套的常见用途——大延时。】

单片机项目经常会用到 delay 这个延时函数，大部分都是利用 for 循环来实现，小延时的函数往往不用嵌套，直接如下编写：

```
for(k=0;k<N;k++);
```

上述的 N 是控制循环次数，每次循环都要消耗单片机一点时间，如果 N 越大需要消耗的时间就越多，起到延时的作用。但是 N 所能取的最大值受它所定义的类型所限制，比如 unsigned char 类型最大范围是 255，unsigned int 类型最大范围是 65535，unsigned long 类型最大范围是 4294967295。如果要实现更大的延时怎么办？就可以用 for 的循环嵌套，利用循环嵌套可以使得循环总次数进行乘法翻倍的放大，很容易编写大延时的函数。比如：

```

for(i=0;i<M;i++) //大循环
{
    for(k=0;k<N;k++); //内嵌的小循环
}

```

此时循环的次数是 N 乘以 M 的乘积。如果 N 和 M 都是 unsigned long 类型，就意味着最大循环次数是 4294967295 的平方，次数大到惊人。

#### 【51.5 例程练习和分析。】

现在编写一个循环嵌套的练习程序。

程序代码如下：

```

/*---C 语言学习区域的开始。-----*/

unsigned char a=0; //观察这个数最后的变化
unsigned char b=0; //观察这个数最后的变化
unsigned char c=0; //观察这个数最后的变化

unsigned char i; //控制大循环体的条件判断变量
unsigned char k; //控制内嵌小循环体的条件判断变量
void main() //主函数
{
    for(i=0;i<2;i++) //大循环
    {
        a=a+1; //被执行了 2 次
        for(k=0;k<3;k++) //内嵌小循环
        {
            b=b+1; //被执行了 6 次，也就是 i 乘以 k，2 乘以 3 等于 6.
        }
    }
}

```

```

        c=c+1;    //被执行了 2 次
    }

    View(a); //把第 1 个数 a 发送到电脑端的串口助手软件上观察。
    View(b); //把第 2 个数 b 发送到电脑端的串口助手软件上观察。
    View(c); //把第 3 个数 c 发送到电脑端的串口助手软件上观察。

    while(1)
    {
    }
}

/*---C 语言学习区域的结束。-----*/

```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第 1 个数

十进制:2

十六进制:2

二进制:10

第 2 个数

十进制:6

十六进制:6

二进制:110

第 3 个数

十进制:2

十六进制:2

二进制:10

分析：

变量 a 为 2。

变量 b 为 6。

变量 c 为 2。

## 【51.6 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。

## 第五十二节： 支撑程序框架的 switch 语句。

### 【52.1 switch 的重要性。】

switch 是非常重要的语句，我所有的单片机项目都是用 switch 搭建程序主框架。如果说 while 和 for 是一对孪生兄弟，那么“if-else if”和 switch 也是一对孪生兄弟，凡是用“if-else if”能实现的功能都可以用 switch 实现。switch 有条件分支的功能，当条件的分支超过 3 个以上时，switch 会比“if-else if”更加直观清晰。

### 【52.2 switch 的语法。】

switch 常见的格式如下：

```
switch(变量)    //根据变量的数值大小从对应的 case 入口进来
{
    case 0:    //入口 0
        语句 0;
        break; //switch 程序体的出口之一
    case 1:    //入口 1
        语句 1;
        break; //switch 程序体的出口之一
    case 2:    //入口 2
        语句 2;
        break; //switch 程序体的出口之一
}    //最下面的花括号也是一个 switch 程序体的出口之一
```

分析：单片机从第一行的 switch(变量)进来，依次往下查询跟变量匹配的 case 入口，然后从匹配的 case 入口进来，往下执行语句，直到遇上 break 语句，或者 return 语句，或者“最下面的花括号”这三种情况之一，才跳出当前 switch 程序体。上述例子中，假如变量等于 3，单片机从 switch(变量)进来，往下查询跟 3 匹配的 case 入口，因为没有发现 case 3，最后遇到“最下面的花括号”于是结束 switch 程序体，像这种变量等于 3 的情况，就意味着 switch 里面的有效语句没有被执行到。多补充一句，在 case 2 选项中，“语句 2”后面紧跟的 break 可以省略，因为 case 2 是最后一个 case，即使没有遇到 break 也会遇到“最下面的花括号”而结束 switch 程序体。上述程序功能如果用“if-else if”语句来实现，等效于如下：

```
if(0==变量)
{
    语句 0;
}
else if(1==变量)
{
    语句 1;
}
else if(2==变量)
{
    语句 2;
}
```

### 【52.3 switch 的 break。】

刚才的例子中，可以看到三个关键字:switch, case, break。其实并不是每个 case 都必须要跟 break 配套,break 只是起到一个出口的功能。假如没有遇到 break,程序会一直往下执行,直到遇到 break 或者 switch “最下面的花括号”为止。比如:

```
switch(变量)    //根据变量的数值大小从对应的 case 入口进来
{
    case 0:    //入口 0
        语句 0;
        break;
    case 1:    //入口 1
        语句 1;
    case 2: //入口 2
        语句 2;
        break;
    case 3: //入口 3
        语句 3;
        break;
}    //最下面的花括号也是一个 switch 程序体的出口之一
```

分析:假如此时 switch(变量)的变量等于 1,单片机经过查询后,就从匹配的 case 1 入口进来,执行“语句 1”后,居然没有遇到 break 语句,于是紧接着碰到“case 2”入口的语句,现在问题来了,单片机此时是退出 switch 程序体还是忽略“case 2”入口语句而继续执行后面的“语句 2”? 答案是:忽略“case 2”入口语句而继续执行后面的“语句 2”。这里有点像坐地铁,你只关注一个入口和一个出口,进入地铁内之后,你中途再遇到无数个入口都可以忽略而继续前进,直到你到达目的地的出口才结束整个乘车过程。继续刚才的分析,单片机执行“语句 2”之后,紧接着遇到 break 语句,这时才跳出整个 switch 程序体。回顾一下整个流程,本例子中 case 1 没有 break 语句,就继续往下执行下面 case2 里面的语句,直到遇到 break 或者“最下面的花括号”为止。

### 【52.4 case 的变量有顺序要求吗?】

switch 语句内部的 case 有规定顺序吗? 必须连贯吗? switch 程序体内部可以写很多 case 入口,这些 case 入口是不是必须按从小到大的顺序? 是不是规定必须 case 数字连贯? 答案是:没有规定顺序,也没有规定 case 数字连贯。case 的数值只是代表入口,比如以下两种写法都是合法的:

第一种: case 不按从小到大的顺序(这种格式是合法的):

```
switch(变量)
{
    case 2:
        语句 2;
        break;
    case 0:
        语句 0;
        break;
```



```

    case 1:
        语句 1;
        break;
}

```

第二种：case 的数字不连贯（这种格式也是合法的）：

```

switch(变量)
{
    case 0:
        语句 0;
        break;
    case 3:
        语句 3;
        break;
    case 9:
        语句 9;
        break;
}

```

## 【52.5 switch 的 default.】

default 是入口语句，它在 switch 语句中也不是必须的，应根据程序需要来选择。default 相当于“if-else if-else ”组合语句中的 else，也就是当 switch 的入口变量没有匹配的 case 入口时，就会默认进入 default 入口，就像“if-else if-else ”语句中当前面所有的条件不满足时，就进入 else 语句的程序体，比如：

```

switch(变量)    //根据变量的数值大小从对应的 case 入口进来
{
    case 0:    //入口 0
        语句 0;
        break; //switch 程序体的出口之一
    case 1:    //入口 1
        语句 1;
        break; //switch 程序体的出口之一
    case 2:    //入口 2
        语句 2;
        break; //switch 程序体的出口之一
    default:    //当所有的 case 不满足，就从 default 的入口进来
        语句 3;
        break;
}    //最下面的花括号也是一个 switch 程序体的出口之一

```

分析：假如 switch 的入口变量等于 35，单片机从上往下查询，因为没有找到 case 35，所以就会从默认的 default 入口进来执行” 语句 3”，然后遇到 break 语句才跳出 switch 程序体。上述程序功能如果用“if-else if-else”组合语句来实现等效于如下：

```

if(0==变量)
{
    语句 0;
}
else if(1==变量)
{
    语句 1;
}
else if(2==变量)
{
    语句 2;
}
else    //相当于 switch 中的 default
{
    语句 3;
}

```

## 【52.6 switch 中内嵌 switch。】

if 语句可以内嵌 if 语句，while 语句也可以内嵌 while 语句，switch 语句当然也可以内嵌 switch。比如：

```

switch(a)
{
    case 1:
        switch(b)    //内嵌的 switch
        {
            case 1:
                Break;
            case 2:
                Break;
        }
        Break;
    case 2:
        Break;
}

```

分析：上述这种 switch 内嵌 switch 语句也是合法的，而且在实际项目中也很常用，大家目前先有个大概的了解即可，暂时不深入讲解。

## 【52.7 例程练习和分析。】

现在编写一个 switch 的练习程序。  
程序代码如下：

```

/*---C 语言学习区域的开始。-----*/

    unsigned char k;    //switch 的入口变量
    unsigned char a;    //观察此变量的变化来理解 switch 的执行顺序
void main() //主函数
{
    a=0;
    k=2;    //入口变量等于 2
    switch(k)
    {
        case 0:    //入口 0
            a++;
            break; //跳出 switch
        case 1:    //入口 1
            a++;
        case 2:    //入口 2, 上述 k 等于 2 所以从这里进来
            a++;
        case 3:    //入口 3
            a++;
        case 4:    //入口 4
            a++;
            break; //跳出 switch
        case 5:    //入口 5
            a++;
            break; //跳出 switch
        default:    //当前面没有遇到匹配的 case 入口时, 就从此 default 入口进来
            a++;
            break; //跳出 switch
    }    //最后一个 switch 的花括号也是跳出 switch

    View(a);    //把第 1 个数 a 发送到电脑端的串口助手软件上观察。

    while(1)
    {
    }
}

/*---C 语言学习区域的结束。-----*/

```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第 1 个数

十进制:3

十六进制:3

二进制:11

分析:

变量 a 为 3。单片机从 case 2 入口进来, 因为 case 2 和 case 3 都没有 break 语句, 直到遇到 case 4 的 break 语句才结束 switch 程序体, 因此整个过程遇到了 3 次 “a++” 语句, 因此变量 a 的 “自加一” 执行了 3 次后从 0 变成了 3。

### 【52.8 如何在单片机上练习本章节 C 语言程序?】

直接复制前面章节中第十一节的模板程序, 练习代码时只需要更改 “C 语言学习区域” 的代码就可以了, 其它部分的代码不要动。编译后, 把程序下载进带串口的 51 学习板, 通过电脑端的串口助手软件就可以观察到不同的变量数值, 详细方法请看第十一节内容。

## 第五十三节： 使用函数的三要素和执行顺序。

### 【53.1 函数的十大关联部件。】

函数是什么？我很难用一句话给它下定义，哪怕我真能用一句话定义了，初学者也很难从一句话的定义中“格”出函数之理。之所以函数有如此玄机，确实因为它包罗万象，涉及的内容非常多，就像要我去定义什么是中国，我也没法用一句话去定义，只有长大了慢慢了解它的地理文化历史，你才会对咱中国有深刻的认识。函数也是如此，虽然我不能用一句话定义函数，但是函数跟十大部件有关，只要今后不断学习和运用，对十大部件各个击破直到全部“通关”，总有一天你会感悟到函数的精髓。现在先把十大部件列出来，让大家有一个感性的认识，它们是：函数体，函数接口，return 语句，栈，全局变量，普通局部变量，静态局部变量，单个变量的指针，数组的指针，结构体的指针。本节讲的“使用函数的三要素和执行顺序”就是属于“函数体”这个部件的内容。

### 【53.2 使用函数的三要素。】

有的人习惯把函数称为程序，比如主程序，子程序，这时的主程序对应主函数，子程序对应子函数，是一回事，只是每个人的表达习惯不一样而已。使用函数的三要素是声明，定义，调用。每次新构造一个函数时，尽量遵守这个三个要素来做就可以减少一些差错。什么叫函数的声明，定义，调用？为了让大家有一个感性的认识，请先看下面这个例子：

```
/*---C 语言学习区域的开始。-----*/

void HanShu(void);    //子函数声明的第一区域

unsigned char  a;    //全局变量定义的第二区域
unsigned char  b;
unsigned char  c;

void HanShu(void)    //子函数定义的第三区域
{
    a++;    //子函数的代码语句
    b=b+5;
    c=c+6;
}

void main() //主函数
{
    a=0;
    b=0;
    c=0;
    HanShu() ;    //子函数被调用的第四区域
    c=a+b;
    while(1)
    {
```

```

    }
}
/*---C 语言学习区域的结束。-----*/

```

分析：上述例子中，从书写代码区域的角度来寻找函数的大概规律，从上往下：

第一区域：写子函数 HanShu 声明。

第二区域：全局变量的定义。

第三区域：子函数 HanShu 的定义。

第四区域：在 main 函数里对子函数 HanShu 的调用。

### 【53.3 子函数被其它函数调用时候的执行顺序。】

子函数被其它函数调用时，子函数的名字就相当于一个跳转地址，而子函数的定义部分就是要跳转的实际地址，单片机在主函数里遇到子函数名字，就直接跳转到子函数定义那里执行子函数内部的代码，执行完子函数后再返回到主函数，此时返回到主函数哪里呢？答：因为子函数已经被执行了一次，所以返回到主函数中的子函数名字后面，然后继续往下执行 main 函数其它剩余的代码。请看下面这个代码的执行顺序，一目了然：

```

/*---C 语言学习区域的开始。-----*/

void HanShu(void);    //子函数的声明
void HanShu(void)     //子函数的定义
{
    语句 1;
    语句 2;
}
void main() //主函数
{
    语句 3;
    HanShu() ;      //子函数的被调用
    语句 4;

    while(1)
    {

    }
}
/*---C 语言学习区域的结束。-----*/

```

执行顺序分析：单片机从主函数 main 那里进来往下执行，先执行“语句 3”，接着遇到 HanShu 名字的跳转地址，然后马上跳转到 HanShu 的定义部分，执行“语句 1”，“语句 2”，执行完子函数 HanShu 的定义部分，就马上返回到主函数，继续执行 HanShu 名字后面的“语句 4”。整个执行语句的先后顺序如下：

```

语句 3;
语句 1;
语句 2;

```

语句 4;

### 【53.4 例程练习和分析。】

现在编写一个练习程序来体验一下函数的使用。

程序代码如下：

```
/*---C 语言学习区域的开始。-----*/
void HanShu(void); //子函数声明的第一区域

unsigned char a; //全局变量定义的第二区域

void HanShu(void) //子函数定义的第三区域
{
    a++; //子函数的代码语句
}
void main() //主函数
{
    a=0;
    a++;
    HanShu() ; //子函数被调用的第四区域
    a++;
    View(a); //把第 1 个数 a 发送到电脑端的串口助手软件上观察。
    while(1)
    {

    }

}
/*---C 语言学习区域的结束。-----*/
```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第 1 个数

十进制:3

十六进制:3

二进制:11

分析：

变量 a 为 3。单片机从 main 主函数进来，主函数里有 2 条“a++”，再加上子函数里也有 1 条“a++”，因此累加了 3 次，从 0 变成了 3。

### 【53.5 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。



## 第五十四节： 从全局变量和局部变量中感悟“栈”为何物。

### 【54.1 本节阅读前的名词约定。】

变量可以粗略的分成两类，一类是全局变量，一类是局部变量。如果更深一步精细划分，全局变量还可以分成“普通全局变量”和“静态全局变量”，局部变量也可以分成“普通局部变量”和“静态局部变量”，也就是说，若精细划分，可以分成四类。其中“静态全局变量”和“静态局部变量”多了一个前缀“静态”，这个前缀“静态”是因为在普通的变量前面多加了一个修饰关键词“static”，这部分的内容后续章节会讲到。本节重点为了让大家理解内存模型的“栈”，暂时不考虑“静态变量”的情况，人为约定，本节所涉及的“全局变量”仅仅默认为“普通全局变量”，“局部变量”仅仅默认为“普通局部变量”。

### 【54.2 如何判定全局变量和局部变量？】

全局变量就是在函数外面定义的变量，局部变量就是在函数内部定义的变量，这是最直观的判定方法。下面的例子能很清晰地说明全局变量和局部变量的判定方法：

```
unsigned char a;    //在函数外面定义的，所以是全局变量。
void main()  //主函数
{
    unsigned char b; //在函数内部定义的，所以是局部变量。
    b=a;
    while(1)
    {

    }
}
```

### 【54.3 全局变量和局部变量的内存模型。】

单片机内存包括 ROM 和 RAM 两部分，ROM 存储的是单片机程序中的指令和一些不可更改的常量数据，而 RAM 存放的是可以被更改的变量数据，也就是说，全局变量和局部变量都是存放在 RAM，但是，虽然都是存放在 RAM，全局变量和局部变量之间的内存模型还是有明显的区别的，因此，分了两个不同的 RAM 区，全局变量占用的 RAM 区称为“全局数据区”，局部变量占用的 RAM 区称为“栈”，因为我后面会用宾馆来比喻“栈”，为了方便记忆，大家可以把“栈”想象成“客栈”来记忆。它们的内存模型到底有什么本质的区别呢？“全局数据区”就像你自己家的房间，是唯一的，一个房间的地址只能你一个人住（假设你还没结婚的时候），而且是永久的，所以说每个全局变量都有唯一对应的 RAM 地址，不可能重复的。而“栈”就像宾馆客栈，一年下来每天晚上住的人不一样，每个人在里面居住的时间是有期限的，不是长久的，一个房间的地址一年下来每天可能住进不同的人，不是唯一的。“全局数据区”的全局变量拥有永久产权，“栈”区的局部变量只能临时居住在宾馆客栈，地址不是唯一的，有期限的。全局变量像私人区，局部变量像公共区。“栈”的这片公共区，是给程序里所有函数内部的局部变量共用的，函数被调用的时候，该函数内部的每个局部变量就会被分配对应到“栈”的某个 RAM 地址，函数调用结束后，该局部变量就失效，因此它对应的“栈”的 RAM 空间就被收回以便给下一个被调用的函数的局部变量占用。请看下面这个例子，我借用“宾馆客栈”来比喻局部变量所在的“栈”。

```
void HanShu(void);  //子函数的声明
```

```

void HanShu(void)    //子函数的定义
{
    unsigned char a;    //局部变量
    a=1;
}
void main() //主函数
{
    HanShu() ;        //子函数的调用
}

```

分析：上述例子，单片机从主函数 main 往下执行，首先遇到 HanShu 子函数的调用，所以就跳到 HanShu 函数的定义那里开始执行，此时的局部变量 a 开始被分配在 RAM 的“栈区”的某个地址，相当于你入住宾馆被分配到某个房间。单片机执行完子函数 HanShu 后，局部变量 a 在 RAM 的“栈区”所分配的地址被收回，局部变量 a 消失，被收回的 RAM 地址可能会被系统重新分配给其它被调用的函数的局部变量，此时相当于你离开宾馆，从此你跟那个宾馆的房间没有啥关系，你原来在宾馆入住的那个房间会被宾馆老板重新分配给其他的客人入住。全局变量的作用域是永久性不受范围限制的，而局部变量的作用域就是它所在函数的内部范围。全局变量的“全局数据区”是永久的私人房子（这里的“永久”仅仅是举一个例子，别拿“70 年产权”来抬杠），局部变量的“栈”是临时居住的“客栈”。重要的事情说两遍，再次总结如下：

（1）每定义一个新的全局变量，就意味着多开销一个新的 RAM 内存。而每定义一个局部变量，只要在函数内部所定义的局部变量总数不超过单片机的“栈”区，此时的局部变量不开销新的 RAM 内存，因为局部变量是临时借用“栈”区的，使用后就还给“栈”，“栈”是公共区，可以重复利用，可以服务若干个不同的函数内部的局部变量。

（2）单片机每次进入执行函数时，局部变量都会被初始化改变，而全局变量则不会被初始化，全局变量是一直保存之前最后一次更改的值。

#### 【54.4 三个常见疑问。】

第一个疑问：

问：“全局数据区”和“栈区”是谁在幕后分配的，怎么分配的？

答：是 C 编译器自动分配的，至于怎么分配，谁分配多一点，谁分配少一点，C 编译器会有一个默认的比例分配，我们一般都不用管。

第二个疑问：

问：“栈”区是临时借用的，子函数被调用的时候，它内部的局部变量才会“临时”被分配到“栈”区的某个地址，那么问题来了，谁在幕后主持“栈区”这些分配的工作，难道也是 C 编译器？C 编译器不是在编译程序的时候一次性就做完了编译工作然后就退出历史舞台了吗？难道我们程序已经在单片机内部运转的时候，编译器此时还在幕后指手画脚的起作用？

答：单片机已经上电开始运行程序的时候，编译器是不可能起作用的。所以，真相只有一个，“栈区”分配给函数内部局部变量的工作，确实是 C 编译器做的，唯一需要注意的地方是，它不是“现炒现卖”，而是在单片机上电前，C 编译器就把所有函数内部的局部变量的分配工作就规划好了，都指定了如果某个函数一旦被调用，该函数内部的哪个局部变量应该分到“栈区”的哪个地址，C 编译器都是事先把这些“后事”都交代完毕了才“结束自己的生命”，后面，等单片机上电开始工作的时候，虽然 C 编译器此时“不在”了，但是单片机都是严格按照 C 编译器交代的“遗嘱”开始工作和分配“栈区”的。因此，“栈区”的“临时分配”非真正严格意义上的“临时分配”。

第三个疑问：

问：函数内部所定义的局部变量总数不超过单片机的“栈”区的 RAM 数量，那，万一超过了“栈”区的 RAM 数量，后果严重吗？

答：后果特别严重。这种情况，专业术语叫“爆栈”。程序会出现异常，而且是莫名其妙的异常。为了避免这种情况，一般在编写程序的时候，函数内部都不能定义大数组的局部变量，局部变量的数量不能定义太多太大，尤其要避免刚才所说的定义开辟大数组局部变量这种情况。大数组的定义应该定义成全局变量，或者定义成“静态的局部变量”（“静态”这部分相关的内容后面章节会讲到）。有一些 C 编译器，遇到“爆栈”的情况，会好心跟你提醒让你编译不过去，但是也有一些 C 编译器可能就不会给你提醒，所以大家以后做项目写函数的时候，要对“爆栈”心存敬畏。

### 【54.5 全局变量和局部变量的优先级。】

刚才说到，全局变量的作用域是永久性并且不受范围限制的，而局部变量的作用域就是它所在函数的内部范围，那么问题来了，假如局部变量和全局变量的名字重名了，此时函数内部执行的变量到底是局部变量还是全局变量？这个问题就涉及到优先级。注意，当面对同名的局部变量和全局变量时，函数内部执行的变量是局部变量，也就是局部变量在函数内部要比全局变量的优先级高。为了深刻理解“全局变量和局部变量的优先级”，强烈建议大家必须仔细看完下面列举的三个练习例子。

### 【54.6 例程练习和分析。】

请看下面第一个例子：

```
/*---C 语言学习区域的开始。-----*/

unsigned char a=5;      //此处第 1 个 a 是全局变量。

void main() //主函数
{
    unsigned char a=2; //此处第 2 个 a 是局部变量。跟上面全局变量的第 1 个 a 重名了！

    View(a); //把 a 发送到电脑端的串口助手软件上观察。
    while(1)
    {

    }

}

/*---C 语言学习区域的结束。-----*/
```

分析：

上述例子，有 2 个变量重名了！其中一个全局变量，另外一个局部变量。此时输出显示的结果是 5 还是 2？正确的答案是 2。因为在函数内部，函数内部的局部变量比全局变量的优先级更加高。此时 View(a) 是第 2 个局部变量的 a，而不是第 1 个全局变量的 a。虽然这里的两个 a 重名了，但是它们的内存模型不一样，第 1 个全局变量的 a 是分配在“全局数据区”是具有唯一的地址的，而第 2 个局部变量的 a 是被分配在

临时的“栈”区的，寄生在 main 函数内部。

再看下面第二个例子：

```
/*---C 语言学习区域的开始。-----*/
void HanShu(void); //函数声明
unsigned char a=5;    //此处第 1 个 a 是全局变量。
void HanShu(void)    //函数定义
{
    unsigned char a=3; //此处第 2 个 a 是局部变量。
}
void main() //主函数
{
    unsigned char a=2; //此处第 3 个 a 也是局部变量。
    HanShu(); //子函数被调用
    View(a); //把 a 发送到电脑端的串口助手软件上观察。
    while(1)
    {

    }

}
/*---C 语言学习区域的结束。-----*/
```

分析：

上述例子，有 3 个变量重名了！其中一个是全局变量，另外两个是局部变量。此时输出显示的结果是 5 还是 3 还是 2？正确的答案是 2。因为，HanShu 这个子函数是被调用结束之后，才执行 View(a) 的，就意味 HanShu 函数内部的局部变量(第 2 个局部变量 a)是在执行 View(a) 语句的时候就消亡不存在了，所以此时 View(a) 的 a 是第 3 个局部变量的 a（在 main 函数内部定义的局部变量的 a）。

再看下面第三个例子：

```
/*---C 语言学习区域的开始。-----*/
void HanShu(void); //函数声明
unsigned char a=5;    //此处第 1 个 a 是全局变量。
void HanShu(void)    //函数定义
{
    unsigned char a=3; //此处第 2 个 a 是局部变量。
}
void main() //主函数
{
    HanShu(); //子函数被调用
    View(a); //把 a 发送到电脑端的串口助手软件上观察。
```

```
while(1)
{

}

}

/*---C 语言学习区域的结束。-----*/
```

分析：

上述例子，有 2 个变量重名了！其中一个是全局变量，另外一个为局部变量。此时输出显示的结果是 5 还是 3？正确的答案是 5。因为，HanShu 这个子函数是被调用结束之后，才执行 View(a) 的，就意味 HanShu 函数内部的局部变量(第 2 个局部变量)是在执行 View(a) 语句的时候就消亡不存在了，同时，因为此时 main 函数内部也没有定义 a 的局部变量，所以此时 View(a) 的 a 是必然只能是第 1 个全局变量的 a (在 main 函数外面定义的全局变量的 a)。

### 【54.7 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。

## 第五十五节： 函数的作用和四种常见书写类型。

### 【55.1 函数和变量的命名规则。】

函数的名字和变量的名字一样，一般是由“字母，数字，下划线”三者组成。第1个字符不能是数字，必须是字母或者下划线“\_”，后面紧跟的第2个字符开始可以是数字。在C语言中名字所用的字母是区分大小写的。可以用下划线“\_”，但是不可以用横杠“-”。名字不能跟C编译系统已经征用的关键字重名，比如不能用“unsigned”，“char”，“static”等系统关键词，跟古代时不能跟皇帝重名一样，要避尊者讳。

### 【55.2 函数的作用和分类。】

函数的作用。通常把一些可能反复用到的算法或者过程封装成一个函数，函数就是一个模块，给它输入特定的参数，就可以输出想要的结果，比如一个加法函数，只要输入加数和被加数，然后就会输出相加结果之和，里面具体的算法过程只要写一次就可以重复调用，极大的节省单片机程序容量，也节省程序开发人员的工作量。还有一类函数，它从封装上看无所谓“输入输出”，这类函数往往是针对某一种可能重复使用的“过程”。

函数的分类。暂时排除指针的情况下（指针的内容后续章节会讲到），从输入输出的角度来看，有四种常见的书写类型。分别是“无输出无输入，无输出有输入，有输出无输入，有输出有输入”。“输出”是看函数名的前缀，前缀如果是void表示“无输出”，否则就是“有输出”。“输入”是看函数名括号里的内容，如果是void或者是空着就表示“无输入”，否则就是“有输入”。“输出”和“输入”是比较通俗的说法，专业一点的说法是，“有输出”表示函数“有返回”，“无输出”表示函数“无返回”。“有输入”表示函数“有形参”，“无输入”表示函数“无形参”。下面举一个加法函数的例子，分别用四种不同的函数类型来实现，通过对比它们之间的差别，来体会它们在书写方面有哪些不同，又有哪些规律。

### 【55.3 第1类：“无输出”“无输入”的函数。】

```
unsigned char a; //此变量用来接收最后相加结果的和。
unsigned char g=2;
unsigned char h=3;
void HanShu(void) //“无输出”“无输入”函数的定义。
{
    a=g+h;
}
main()
{
    HanShu(); //函数的调用。此处括号内的形参 void 要省略，否则编译不通过。
}
```

分析：void HanShu(void)，此函数名的前缀是void，括号内也是void，属于“无输出”“无输入”函数。这类函数表面看是“无输出”“无输入”，其实内部是通过全局变量来输入输出的，比如上面的例子就是靠a, g, h这三个全局变量来传递信息，只不过这类表达方式比较隐蔽，没有那么直观。

### 【55.4 第2类：“无输出”“有输入”的函数。】

```
unsigned char b; //此变量用来接收最后相加结果的和。
```

```

void HanShu(unsigned char i,unsigned char k)    // “无输出”“有输入”函数的定义。
{
    b=i+k;
}
main()
{
    HanShu(2,3);    //函数的调用。
}

```

分析：void HanShu(unsigned char i,unsigned char k)，此函数名的前缀是 void，括号内是(unsigned char i,unsigned char k)，属于“无输出”“有输入”的函数。括号的两个变量 i 和 k 是函数内的局部变量，也是跟对外的桥梁接口，它们有一个专业的名称叫形参。外部要调用此函数时，只要给括号填入对应的变量或者数值，这些变量和数值就会被复制一份传递给作为函数形参的局部变量（比如本例子中的 i 和 k），从而外部调用者跟函数内部就发生了数据信息的传递。这种书写方式的特点是把输入接口封装了出来。

### 【55.5 第3类：“有输出”“无输入”的函数。】

```

unsigned char c;    //此变量用来接收最后相加结果的和。
unsigned char m=2;
unsigned char n=3;
unsigned char HanShu(void)    // “有输出”“无输入”函数的定义。
{
    unsigned char p;
    p=m+n;
    return p;
}
main()
{
    c=HanShu();    //函数的调用。此处括号内的形参 void 要省略，否则编译不通过。
}

```

分析：unsigned char HanShu(void)，此函数名的前缀是 unsigned char 类型，括号内是 void，属于“有输出”“无输入”的函数。函数前缀的 unsigned char 表示此函数最后退出时会返回一个 unsigned char 类型的数据给外部调用者。而且这类函数内部必须有一个 return 语句配套，表示立即退出当前函数并且返回某个变量或者常量的数值给外部调用者。这种书写方式的特点是把输出接口封装了出来。

### 【55.6 第4类：“有输出”“有输入”的函数。】

```

unsigned char d;    //此变量用来接收最后相加结果的和。
unsigned char HanShu(unsigned char r,unsigned char s)    // “有输出”“有输入”函数的定义
{
    unsigned char t;
    t=r+s;
    return t;
}

```

```

}
main()
{
    d=HanShu(2,3); //函数的调用。
}

```

分析：unsigned char HanShu(unsigned char r,unsigned char s)，此函数名的前缀是 unsigned char 类型，括号内是(unsigned char r,unsigned char s)，属于“有输出”“有输入”的函数。输入输出的特点跟前面介绍的函数一样，不多讲。这种书写方式的特点是把输出和输入接口都封装了出来。

### 【55.7 函数在被“调用”时需要注意的地方。】

函数的三要素是“声明，定义，调用”。函数在被“调用”的时候，对于“无输入”的函数，形参的 void 关键词要省略，否则编译不通过，这里仅仅是指在函数在被“调用”的时候。

### 【55.8 例程练习和分析。】

现在编写一个练习程序，要求编写 4 个不同“输入输出”封装的函数，它们每个函数所实现的功能都是一样的，都是加法的算法函数，它们之间仅仅是外观的封装接口不同而已。

```

/*---C 语言学习区域的开始。-----*/

void hanshu_1(void);
void hanshu_2(unsigned char i,unsigned char k);
unsigned char hanshu_3(void);
unsigned char hanshu_4(unsigned char r,unsigned char s);

unsigned char a;    //此变量用来接收第 1 个函数最后相加结果的和。
unsigned char g=2;
unsigned char h=3;

unsigned char b;    //此变量用来接收第 2 个函数最后相加结果的和。

unsigned char c;    //此变量用来接收第 3 个函数最后相加结果的和。
unsigned char m=2;
unsigned char n=3;

unsigned char d;    //此变量用来接收第 4 个函数最后相加结果的和。

void hanshu_1(void) //第 1 类：“无输出”“无输入”。
{
    a=g+h;
}

void hanshu_2(unsigned char i,unsigned char k) //第 2 类：“无输出”“有输入”。

```



```

{
    b=i+k;
}

unsigned char hanshu_3(void)    //第3类：“有输出”“无输入”。
{
    unsigned char p;
    p=m+n;
    return p;
}

unsigned char hanshu_4(unsigned char r,unsigned char s)  //第4类：“有输出”“有输入”。
{
    unsigned char t;
    t=r+s;
    return t;
}

void main() //主函数
{
    hanshu_1();        //第1类：“无输出”“无输入”的函数调用。这里的形参的void要省略。
    hanshu_2(2,3);     //第2类：“无输出”“有输入”的函数调用。
    c=hanshu_3();      //第3类：“有输出”“无输入”的函数调用。这里的形参的void要省略。
    d=hanshu_4(2,3);   //第4类：“有输出”“有输入”的函数调用。
    View(a); //把a发送到电脑端的串口助手软件上观察。
    View(b); //把b发送到电脑端的串口助手软件上观察。
    View(c); //把c发送到电脑端的串口助手软件上观察。
    View(d); //把d发送到电脑端的串口助手软件上观察。
    while(1)
    {
    }
}

/*---C 语言学习区域的结束。-----*/

```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第1个数

十进制:5

十六进制:5

二进制:101

第2个数

十进制:5

十六进制:5

二进制:101

第 3 个数

十进制:5

十六进制:5

二进制:101

第 4 个数

十进制:5

十六进制:5

二进制:101

分析:

变量 a 为 5。

变量 b 为 5。

变量 c 为 5。

变量 d 为 5。

### 【55.9 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。

## 第五十六节： return 在函数中的作用以及四个容易被忽略的功能。

### 【56.1 return 深入讲解。】

return 在英语单词中有“返回”的意思，上一节提到，凡是“有输出”的函数，函数内部必须有一个“return+变量或者常量”与之配套，表示返回的结果给外部调用者接收，这个知识点很容易理解，但是容易被忽略的是另外四个功能：

第一个是 return 语句隐含了立即退出的功能。退出哪？退出当前函数。只要执行到 return 语句，就马上退出当前函数。即使 return 语句身陷多层 while 或者 for 的循环中，它也毫不犹豫立即退出当前函数。

第二个是 return 语句可以出现在函数内的任何位置。可以出现在第一行代码，也可以出现在中间的某行代码，也可以出现在最后一行的代码，它的位置不受限制。很多初学者有个错觉，以为 return 只能出现在最后一行，这是错的。

第三个是 return 语句不仅仅可以用在“有输出”的函数，也可以用在“无输出”的函数，也就是可以用在前缀是 void 的函数里。回顾上一节，在“有输出”的函数里，return 后面紧跟一个变量或者常量，表示返回的数，但是在“无输出”的函数里，因为是“无输出”，此时 return 后面不用跟任何变量或者常量，这种写法也是合法的，表示返回的是空的。此时 return 主要起到立即退出当前函数的作用。

第四个是 return 语句可以在一个函数里出现 N 多次，次数不受限制，不一定必须只能一次。不管一个函数内有多少个 return 语句，只要任何一个 return 语句被单片机执行到，就立即退出当前函数。

### 【56.2 中途立即退出的功能。】

下面的书写格式是合法的：

```
void HanShu(void) // “无输出”函数的定义。
{
    语句 1;
    return; //立即退出当前函数。对于这类“无输出”函数，return 后面没有跟任何变量或者常量。
    语句 2;
    return; //立即退出当前函数。对于这类“无输出”函数，return 后面没有跟任何变量或者常量。
    语句 3;
    return; //立即退出当前函数。对于这类“无输出”函数，return 后面没有跟任何变量或者常量。
}
```

分析：当 HanShu 此函数被调用时，单片机从“语句 1”往下执行，当遇到第一个 return 语句后，马上退出当前函数。后面的“语句 2”和“语句 3”等代码永远不会被执行到。多说一句，大家仔细看看 return 后面跟了什么数没有？什么都没有。因为此函数的前缀是 void 的，是“无输出”的。

### 【56.3 身陷多层 while 或者 for 的循环时的惊人表现。】

下面的书写格式是合法的：

```
void HanShu(void) // “无输出”函数的定义。
{
```

```

语句 1;
while(1)  //第一个循环
{
    while(1)  //第二个循环中的循环
    {
        return; //立即退出当前函数。
    }
    语句 2;
    return; //立即退出当前函数。
}
语句 3;
return; //立即退出当前函数。
}

```

分析：当 HanShu 此函数被调用时，单片机从“语句 1”往下执行，先进入第一个循环，接着进入第二个循环中的循环，然后遇到第一个 return 语句，于是马上退出当前函数。后面的“语句 2”和“语句 3”等代码永远不会被执行到。此函数中，虽然表面看起来有那么多可怕的循环约束着，但是一旦碰上 return 语句都是浮云，立刻退出当前函数。

#### 【56.4 在“有输出”函数里的书写格式。】

把上面例子中“无输出”改成“有输出”的函数后：

```

unsigned char HanShu(void)  // “有输出”函数的定义。
{
    unsigned char a=9;
    语句 1;
    while(1)  //第一个循环
    {
        while(1)  //第二个循环中的循环
        {
            return a; //返回 a 变量的值，并且立即退出当前函数。
        }
        语句 2;
        return a; //返回 a 变量的值，并且立即退出当前函数。
    }
    语句 3;
    return a; //返回 a 变量的值，并且立即退出当前函数。
}

```

分析：因为此函数是“有输出”的函数，所以 return 语句后面必须配套一个变量或者常量，此例子中配套的是 a 变量。当 HanShu 函数被调用时，单片机从“语句 1”往下执行，先进入第一个循环，接着进入第二个循环中的循环，然后遇到第一个“return a”语句，马上退出当前函数。而后面的“语句 2”和“语句 3”等代码是永远不会被执行到的。再一次说明了，return 语句不仅有返回某数的功能，还有立即退出的重

要功能。

### 【56.5 项目中往往是跟 if 语句搭配使用。】

前面的例子只是为了解释 return 语句的执行顺序和功能，实际项目中，如果中间有多个 return 语句，中间的 return 语句不可能像前面的例子那样单独使用，它往往是跟 if 语句一起搭配使用，否则单独用 return 就没有什么意义。比如：

```
void HanShu(void) // “无输出”函数的定义。
{
    语句 1;
    if(某条件满足)
    {
        return; //立即退出当前函数。
    }
    语句 2;
    if(某条件满足)
    {
        return; //立即退出当前函数。
    }
    语句 3;
}
```

分析：单片机从“语句 1”开始往下执行，至于在哪个“return”语句处退出当前函数，就要看哪个 if 的条件满不满足了，如果所有的 if 的条件都不满足，此函数会一直执行完最后的“语句 3”才退出当前函数。

### 【56.6 例程练习和分析。】

写一个简单的除法函数，在除法运算中，除数不能为 0，如果发现除数为 0，就立即退出当前函数，并且返回运算结果默认为 0。

```
/*---C 语言学习区域的开始。-----*/

//函数的声明。
unsigned int ChuFa(unsigned int BeiChuShu,unsigned int ChuShu);

//变量的定义。
unsigned int a;//此变量用来接收除法的运算结果。
unsigned int b;//此变量用来接收除法的运算结果。

//函数的定义。
unsigned int ChuFa(unsigned int BeiChuShu,unsigned int ChuShu)
{
    unsigned int Shang; //返回的除法运算结果：商。
```

```

    if(0==ChuShu)    //如果除数等于 0，就立即退出当前函数，并返回 0
    {
        return 0; // 退出当前函数并且返回 0. 此时后面的代码不会被执行。
    }

    Shang=BeiChuShu/ChuShu; //除法运算的算法
    return Shang; //返回最后的运算结果：商。并且退出当前函数。
}

void main() //主函数
{
    a=ChuFa(128,0); //函数调用。128 除以 0，把商返回给 a 变量。
    b=ChuFa(128,2); //函数调用。128 除以 2，把商返回给 b 变量。

    View(a); //把 a 发送到电脑端的串口助手软件上观察。
    View(b); //把 b 发送到电脑端的串口助手软件上观察。
    while(1)
    {
    }
}

/*---C 语言学习区域的结束。-----*/

```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第 1 个数

十进制:0

十六进制:0

二进制:0

第 2 个数

十进制:64

十六进制:40

二进制:1000000

分析：

变量 a 为 0。

变量 b 为 64。

## 【56.7 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观

察到不同的变量数值，详细方法请看第十一节内容。

## 第五十七节： static 的重要作用。

### 【57.1 变量前加入 static 后发生的“化学反应”。】

有两类变量，一类是全局变量，一类是局部变量。定义时，在任何一类变量前面加入 static 关键词，变量原有的特性都会发生某些变化，因此，static 像化学的催化剂，具有神奇的功能。加 static 关键词的书写格式如下：

```
static unsigned char a;      //这是在全局变量前加的 static 关键词
void HanShu(void)
{
    static unsigned char i;   //这是在局部变量前加的 static 关键词
}
```

### 【57.2 在全局变量前加 static。】

static 读作“静态”，全局变量前加 static，称为静态全局变量。静态全局变量和普通全局变量的功能大体相同，仅在有效范围(作用域)方面有差异。假设整个工程有多个文件组成，普通全局变量的有效范围能覆盖全部文件，在任何一个文件里，以及跨文件与文件之间，在传递信息的层面上都畅通无阻。而静态全局变量只能在当前定义的那个文件里起作用，活动范围完全被限定在一个文件，仿佛被加了紧箍咒，由不得你任性，在传递信息的层面上仅仅局限于定义变量时所在的那一个文件。这部分的内容有个大致印象就可以，暂时不用深入研究，等以后学到“多文件编程”时再关注，因为我当前的程序例子只有一个源文件，还没涉及“多文件编程”。

### 【57.3 在局部变量前加 static。】

这是本节重点。我常把局部变量比喻宾馆的客房，客人入住时被分配在哪间客房是随机临时安排的，第二天退房时宾馆会把客房收回继续分配给下一位其他的客人，是临时公共区。而加入 static 后的局部变量，发生了哪些变化？加入 static 后的局部变量，称为静态局部变量。静态局部变量就像宾馆的 VIP 客户，VIP 客户财大气粗，把宾馆分配的客房永远包了下来，永远不许再给其它客人入住。总结了静态局部变量的两个重要特性：

第一个，静态局部变量不会在函数调用时被初始化，它只在单片机刚上电时被初始化了一次，因为它的内存模型不是分配在“栈”，而是跟全局变量一样放在“全局数据区”，拥有自己唯一的地址。因此，静态局部变量的数值跟全局变量一样，具有“记忆”功能，你每次调用某个函数，函数内部的静态局部变量的数值是维持最后一次被更改的数值，不会被“清零”的。但是跟全局变量又有差别，全局变量的有效范围（作用域）是整个工程，而静态局部变量毕竟是“局部”，在传递信息的层面仅局限于当前函数内。而普通局部变量，众所周知，每次被函数调用时，都会被重新初始化，会被“清零”的，没有“记忆”功能的。

第二个，每次函数调用时，静态局部变量比普通局部变量少开销一条潜在的“初始化语句”，原因是普通局部变量每次被函数调用时都要重新初始化，而静态局部变量不用进行这个操作。也就是说，静态局部变量比普通局部变量的效率高一点，虽然这个“点”的时间开销微不足道，但是写程序时不能忽略这个“点”。静态局部变量用到好处之时，能体现一个工程师的功力。

### 【57.4 静态局部变量的应用场合。】

静态局部变量适用在那些“频繁调用”的函数，比如 main 函数主循环 while(1) 里直接调用的所有函数，



还有以后讲到的定时器中断函数，等等。因为静态局部变量每次被调用都不会被重新初始化，用在这类函数时就省去了每次初始化语句的时间。还有一类用途，就是那些规定不能被函数初始化的场合，比如在很多用 switch 搭建程序框架的函数里，这类 switch 程序框架俗称为状态机思路。

### 【57.5 能用全局变量替代静态局部变量吗？】

能用全局变量替代静态局部变量吗？能。哪怕在整个程序里全部用全局变量都可以。全局变量是一把牛刀，什么场合都用牛刀虽然也能解决问题，但是显得鲁莽没有条理。尽量把全局变量，普通局部变量，静态局部变量各自优势充分发挥出来才是编程之道。能用局部变量的尽量用局部变量，这样可以减少全局变量的使用。当局部变量帮分担一部分工作时，最后全局变量只起到一个作用，那就是在各函数之间传递信息。局部变量与全局变量的分工定位明确了，程序代码阅读起来就没有那么凌乱，思路也清晰很多。

### 【57.6 例程练习和分析。】

现在编写一个程序来熟悉 static 的性能。

```
/*---C 语言学习区域的开始。-----*/

//函数的声明。
unsigned char HanShu(void);
unsigned char HanShu_static(void);

//变量的定义。
unsigned char a; //用来接收函数返回的结果。
unsigned char b;
unsigned char c;
unsigned char d;
unsigned char e;
unsigned char f;

//函数的定义。
unsigned char HanShu(void)
{
    unsigned char i=0; //普通局部变量，每次函数调用都被初始化为 0.
    i++; //i 自加 1
    return i;
}

unsigned char HanShu_static(void)
{
    static unsigned char i=0; //静态局部变量，只在上电是此初始化语句才起作用。
    i++; //i 自加 1
    return i;
}
```

```

void main() //主函数
{
    //下面函数内的 i 是普通局部变量，每次调用都会被重新初始化。
    a=HanShu(); //函数内的 i 每次重新初始化为 0，再自加 1，所以 a 等于 1。
    b=HanShu(); //函数内的 i 每次重新初始化为 0，再自加 1，所以 b 等于 1。
    c=HanShu(); //函数内的 i 每次重新初始化为 0，再自加 1，所以 c 等于 1。

    //下面函数内的 i 是静态局部变量，第一次上电后默认为 0，就不会再被初始化，
    d=HanShu_static(); //d 由 0 自加 1 后等于 1。
    e=HanShu_static(); //e 由 1 自加 1 后等于 2。
    f=HanShu_static(); //f 由 2 自加 1 后等于 3。

    View(a); //把第 1 个数 a 发送到电脑端的串口助手软件上观察。
    View(b); //把第 2 个数 b 发送到电脑端的串口助手软件上观察。
    View(c); //把第 3 个数 c 发送到电脑端的串口助手软件上观察。
    View(d); //把第 4 个数 d 发送到电脑端的串口助手软件上观察。
    View(e); //把第 5 个数 e 发送到电脑端的串口助手软件上观察。
    View(f); //把第 6 个数 f 发送到电脑端的串口助手软件上观察。

    while(1)
    {
    }
}

/*---C 语言学习区域的结束。-----*/

```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第 1 个数

十进制:1

十六进制:1

二进制:1

第 2 个数

十进制:1

十六进制:1

二进制:1

第 3 个数

十进制:1

十六进制:1

二进制:1

第 4 个数  
十进制:1  
十六进制:1  
二进制:1

第 5 个数  
十进制:2  
十六进制:2  
二进制:10

第 6 个数  
十进制:3  
十六进制:3  
二进制:11

分析:

变量 a 为 1。

变量 b 为 1。

变量 c 为 1。

变量 d 为 1。

变量 e 为 2。

变量 f 为 3。

### 【57.7 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。

## 第五十八节： const (或 code) 在定义数据时的作用。

### 【58.1 const 与 code 的关系。】

const 与 code 都是语法的修饰关键词，放在所定义的数据前面时有“不可更改”之意。在 C 语言语法中，const 像普通话全国通用，是标准的语言；而 code 像地方的方言，仅仅适合针对 51 单片机的 C51 编译器环境。而其它大多数单片机的 C 编译器并不支持 code，只支持 const。比如 PIC，stm32 等单片机的 C 编译器都是只认 const 而不认 code 的。通常情况下，const 定义的数据都是放在 ROM 的，但是 51 单片机的 C51 编译器是例外，它并不把 const 定义的数据放在 ROM 区，只有用 code 关键词时它才会把数据放在 ROM 区，这一点相对其它大多数的单片机来说是不一样的。因为本教程是用 51 单片机的 C51 编译器，所以用 code 来替代 const。本节教程所提到的 const，在实际编程时都用 code 来替代。

### 【58.2 const (或 code) 在定义数据时的终极目的。】

在数据定义分配的应用中，const 的终极目的是为了节省 RAM 的开销。从“读”和“写”的角度分析，数据有两种：“能读能写”和“只能读”这两种。“能读能写”的数据占用 RAM 内存，叫变量，C 语言语法上定义此类数据时“无”const 前缀。“只能读”的数据占用 ROM 内存，叫常量，C 语言语法上定义此类数据时“有”const 前缀。单片机的 ROM 容量比 RAM 容量往往大几十倍甚至上百倍，相比之下，RAM 的资源显得比较稀缺。因此，把某些只需“读”而不需“写”的数据定义成 const 放在 ROM，就可以节省 RAM 的开销。

### 【58.3 const (或 code) 的应用场合。】

const 可以定义单个常量，也可以定义常量数组。定义单个常量时，通常应用在某个出现在程序多处并且需要经常调整的“阈值”参数，方便“一键更改”的操作。所谓“一键更改”，就是只要改一次 const 所定义初始化的某个常量，整个程序多次出现的此常量就自动更改了。定义常量数组时，通常应用在某个数据转换表，把某些固定的常量预先放到常量数组，通过数组的下标来“查表”。

### 【58.4 const (或 code) 的语法格式。】

定义单个常量和常量数组时的语法是以下这个样子的：

```
const unsigned char x=10; //定义单个常量。加了 const。
const unsigned char y[12]={31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}; //定义常量数组。加了 const。
```

### 【58.5 const (或 code) 的“能读”和“不可写”概念】

所谓“读”和“写”的能力，其实就是看某数能在赋值符号“=”的“右边”还是“左边”的能力。普通的变量，既可以在赋值符号“=”的“右边”（能读），也可以在赋值符号“=”的“左边”（能写）。比如，下面的写法是合法的：

```
unsigned char k=1; //这是普通的变量，无 const 前缀。
unsigned char n=2; //这是普通的变量，无 const 前缀。
n=k; //k 出现在赋值符号“=”的右边，表示能读。合法。
k=n; //k 出现在赋值符号“=”的左边，表示能写，可更改之意。合法。
```

但是如果一旦在普通的变量前面加了 `const` (或 `code`) 关键词, 就会发生“化学变化”, 原来的“变量”就变成了“常量”, 常量只能“读”, 不能“写”。比如:

```
const unsigned char c=1; //这是常量, 有 const 前缀。
unsigned char n=2; //这是普通的变量, 无 const 前缀。
n=c; //c 是常量, 能读, 这是合法的。这行代码是语法正确的。
c=n; //c 是常量, 不能写, 这是非法的, C 编译器不通过。这行代码是语法错误的。
```

## 【58.6 const (或 code) 能在函数内部定义吗?】

`const` (或 `code`) 能在函数内部定义吗? 能。语法是允许的。当在函数内部定义数据成 `const` (或者 `code`), 在数据的存储结构上, 数据也是放在 ROM 区的 (实际上在 51 单片机里想把数据放在 ROM 只能用 `code` 而不能用 `const`), 把数据定义在函数内部, 就只能在这个函数里面用, 不能被其它函数调用。在作用域的问题上, `const` (或者 `code`) 的常量数据跟其它变量的数据是一样的。比如:

```
void HanShu(void)
{
    const unsigned char c=1; //在函数内部定义的 const 常量也是放在 ROM 区存储。
    unsigned char n=2;
    n=c; //c 是常量, 在函数内部定义, 只能在当前这个 HanShu 函数里调用。
}
```

## 【58.7 例程练习和分析。】

本教程使用的是 51 单片机的 C51 编译器, 编写程序时为了让常量数据真正存储在 ROM 区, 因此, 本教程的程序例子都是用 `code` 替代 `const`。

本例程讲两个例子, 一个是单个常量, 一个是常量数组。

(1) 单个常量。举的例子是“阈值”的“一键更改”应用。根据考试的分数, 分两个等级。凡是大于或者等于 90 分的就是“优”, 串口助手输出显示“1”。凡是小于 90 分的就是“良”, 串口助手输出显示“0”。这里的“90 分”就是我所说的“阈值”概念, 只要用一个 `const` 定义一个常量数据来替代“90”, 当需要调整“阈值”时, 只要更改一次此定义的常量数值就可以达到“一键更改”之目的。

(2) 常量数组。举的例子是, 查询 2017 年 12 个月的某个月的总天数, 用两种思路实现, 一种是 `switch` 分支语句来实现, 另一种是 `const` 常量数组的“查表”思路来实现。通过对比这两种思路, 你会发现 `const` 常量数组在做“转换表”这类“查表”项目时的强大优越性。

```
/*---C 语言学习区域的开始。-----*/

//函数的声明。
unsigned char HanShu_switch(unsigned char u8Month);
unsigned char HanShu_const(unsigned char u8Month);

//数据的定义。
code unsigned char Cu8Level=90; //需要调整“阈值”时, 只需更改一次这里的“90”这个数值。
code unsigned char Cu8MonthBuffer[12]= //每个月对应的天数。从数组下标 0 开始, 0 代表 1 月...
{31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

```
unsigned char a; //用来接收函数返回的结果。
unsigned char b;
unsigned char c;
unsigned char d;
```

//函数的定义。

```
unsigned char HanShu_switch(unsigned char u8Month) //用 switch 分支来实现。
```

```
{
    switch(u8Month)
    {
        case 1: //1 月份的天数
            return 31;
        case 2: //2 月份的天数
            return 28;
        case 3: //3 月份的天数
            return 31;
        case 4: //4 月份的天数
            return 30;
        case 5: //5 月份的天数
            return 31;
        case 6: //6 月份的天数
            return 30;
        case 7: //7 月份的天数
            return 31;
        case 8: //8 月份的天数
            return 31;
        case 9: //9 月份的天数
            return 30;
        case 10: //10 月份的天数
            return 31;
        case 11: //11 月份的天数
            return 30;
        case 12: //12 月份的天数
            return 31;
        default: //万一输错了其它范围的月份，就默认返回 30 天。
            return 30;
    }
}
```

```
unsigned char HanShu_const(unsigned char u8Month) //用 const 常量数组的“查表”来实现。
```

```
{
    unsigned char u8GetDays;
    u8Month=u8Month-1; //因为数组下标是从 0 开始，0 代表 1 月份，1 代表 2 月份。所以减去 1。
```

```

    u8GetDays=Cu8MonthBuffer[u8Month]; //这就是查表，马上获取常量数组表格里固定对应的天数。
    return u8GetDays;
}

void main() //主函数
{
    //第（1）个例子
    if(89>=Cu8Level) //大于或者等于阈值，就输出 1。
    {
        a=1;
    }
    else //否则输出 0。
    {
        a=0;
    }

    if(95>=Cu8Level) //大于或者等于阈值，就输出 1。
    {
        b=1;
    }
    else //否则输出 0。
    {
        b=0;
    }

    //第（2）个例子
    c=HanShu_switch(2); //用 switch 分支的函数获取 2 月份的总天数。
    d=HanShu_const(2); //用 const 常量数组“查表”的函数获取 2 月份的总天数。

    View(a); //把第 1 个数 a 发送到电脑端的串口助手软件上观察。
    View(b); //把第 2 个数 b 发送到电脑端的串口助手软件上观察。
    View(c); //把第 3 个数 c 发送到电脑端的串口助手软件上观察。
    View(d); //把第 4 个数 d 发送到电脑端的串口助手软件上观察。

    while(1)
    {
    }
}

/*---C 语言学习区域的结束。-----*/

```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

```
第 1 个数
十进制:0
十六进制:0
二进制:0

第 2 个数
十进制:1
十六进制:1
二进制:1

第 3 个数
十进制:28
十六进制:1C
二进制:11100

第 4 个数
十进制:28
十六进制:1C
二进制:11100
```

分析：

- a 为 0。
- b 为 1。
- c 为 28。
- d 为 28。

### 【58.8 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。



## 第五十九节： 全局“一键替换”功能的#define。

### 【59.1 #define 作用和书写格式。】

上一节讲 const 的时候，讲到了当某个常量在程序中是属于需要频繁更改的“阈值”的时候，用 const 就可以提供“一键更改”的快捷服务。本节的#define 也具有此功能，而且功能比 const 更加强大灵活，它除了可以应用在常量，还可以应用在运算式以及函数的“一键更改”中。所谓“一键更改”，其实是说，#define 内含了“替换”的功能，此“替换”跟 word 办公软件的“替换”功能几乎是一模一样的。#define 的“替换”功能，除了在某些场合起到“一键更改”的作用，还可以在有些场合，把一些在字符命名上不方便阅读理解的常量、运算式或函数先“替换”成容易理解的字符串，让程序阅读起来更加清晰更加方便维护。#define 的常见三种书写格式如下：

```
#define 字符串 常量      //注意，这里后面没有分号“;”
#define 字符串 运算式    //注意，这里后面没有分号“;”
#define 字符串 函数      //注意，这里后面没有分号“;”
```

具体一点如下：

```
#define AA 1              //常量
#define BB (a+b+c)        //运算式
#define C add()           //函数
```

需要注意的时候，#define 后面没有分号“;”，因为它是 C 语言中的“预处理”的语句，不是单片机运行的程序指令语句。

### 【59.2 #define 的编译机制。】

#define 是属于“预编译”的指令，所谓“预编译”就是在“编译”之前就开始的准备工作。编译器在正式编译某个源代码的时候，先进行“预编译”的准备工作，对于#define 语句，编译器是直接把#define 要替换的内容先在“编辑层面”进行机械化替换，这个“机械化替换”纯粹是字符串的替换，可以理解成 word 办公软件的“替换”编辑功能。比如以下程序：

```
#define A 3
#define B (2+6)    //有括号
#define C 2+6      //无括号
unsigned long x=3;
unsigned long a;
unsigned long b;
unsigned long c;
void main() //主函数
{
    a=x*A;
    b=x*B;
    c=x*C;
```

```

    while(1)
    {
    }
}

```

经过编译器“预编译”的“机械化替换”后，等效于以下代码：

```

unsigned long x=3;
unsigned long a;
unsigned long b;
unsigned long c;
void main() //主函数
{
    a=x*3;
    b=x*(2+6);
    c=x*2+6;
    while(1)
    {
    }
}

```

### 【59.3 #define 在常量上的“一键替换”功能。】

上一节讲 const（或 code）的时候，举了一个“阈值”常量的例子，这个例子可以用#define 来替换等效。比如，原来 const(或 code)的例子如下：

```

code unsigned char Cu8Level=90; //需要调整“阈值”时，只需更改一次这里的“90”这个数值。

unsigned char a;
unsigned char b;
void main() //主函数
{
    if(89>=Cu8Level) //大于或者等于阈值，就输出 1。
    {
        a=1;
    }
    else //否则输出 0。
    {
        a=0;
    }
    if(95>=Cu8Level) //大于或者等于阈值，就输出 1。
    {
        b=1;
    }
}

```

```

    }
    else //否则输出 0。
    {
        b=0;
    }
    while(1)
    {
    }
}

```

上述程序现在用#define 来替换，等效如下：

```

#define Cu8Level 90 //需要调整“阈值”时，只需更改一次这里的“90”这个数值。

unsigned char a;
unsigned char b;
void main() //主函数
{
    if(89>=Cu8Level) //大于或者等于阈值，就输出 1。
    {
        a=1;
    }
    else //否则输出 0。
    {
        a=0;
    }
    if(95>=Cu8Level) //大于或者等于阈值，就输出 1。
    {
        b=1;
    }
    else //否则输出 0。
    {
        b=0;
    }
    while(1)
    {
    }
}

```

#### 【59.4 #define 在运算式上的“一键替换”功能。】

#define 在运算式上应用的时候，有一个地方要特别注意，就是必须加小括号“()”，否则容易出错。因为#define 的替换是很“机械呆板”的，它只管“字符编辑层面”的机械化替换，举一个例子如下：

```

#define B (2+6)    //有括号
#define C 2+6      //无括号
unsigned long x=3;
unsigned long b;
unsigned long c;
void main() //主函数
{
    b=x*B; //等效于 b=x*(2+6)，最终运算结果 b 等于 24。因为 3 乘以 8（2 加上 6 等于 8）。
    c=x*C; //等效于 c=x*2+6，最终运算结果 c 等于 12。因为 3 乘以 2 等于 6，6 再加 6 等于 12。
    while(1)
    {
    }
}

```

上述例子中，“有括号”与“没括号”的运算结果差别很大，第一个是 24，第二个是 12。具体的分析已经在源代码的注释了。

### 【59.5 #define 在函数上的“一键替换”功能。】

#define 的应用很广，也可以应用在函数的“替换”上。例子如下：

```

void add(void); //函数的声明。
void add(void)  //函数的定义。
{
    a++;
}

#define a_zi_jia add() //用字符串 a_zi_jia 来替代函数 add()。

unsigned long a=1;
void main() //主函数
{
    a_zi_jia; //这里相当于调用函数 add()。
    while(1)
    {
    }
}

```

### 【59.6 #define 在常量后面添加 U 或者 L 的特殊写法。】

有些初学者今后可能在工作中遇到#define 以下这种写法：

```
#define 字符串 常量U
#define 字符串 常量L
```

具体一点如下：

```
#define AA 6U
#define BB 6L
```

常量加后缀“U”或者“L”有什么含义呢？字面上理解，U表示该常量是无符号整型 unsigned int;L表示该常量是长整型 long。但是在实际应用中这样“多此一举”地去强调某个常量的数据类型有什么意义呢？我自己私下也做了一些测试，目前我本人暂时还没有发现这个秘密的答案。所以对于这个问题，初学者现在只要知道这种写法在语法上是合法的就可以，至于它背后有什么玄机，有待大家今后更深的发掘。

### 【59.7 #define 省略常量的特殊写法。】

有些初学者今后在多文件编程中，在某些头文件.h中，会经常遇到以下这类代码：

```
#ifndef _AAA_
#define _AAA_
#endif
```

其中第2行代码“#define \_AAA\_”后面居然没有常量，这样子的写法也行，到底是什么意思？在这类写法中，当字符串“\_AAA\_”后面省略了常量的时候，编译器默认会给\_AAA\_添加一个“非0”的常量，也许是1或者其它“非0”的值，多说一句，所谓“非0”值就是“肯定不是0”。上述代码等效于：

```
#ifndef _AAA_
#define _AAA_ 1 //编译器会在这类默认添加一个1 或者其它“非0”的常量
#endif
```

这个知识点大家只要先有一个感性的认识即可，暂时不用深入了解。

### 【59.8 例程练习和分析。】

现在编一个练习程序来熟悉#define 的用法。

```
/*---C 语言学习区域的开始。-----*/

//第1个：常量的例子
#define Cu8Level 90 //需要调整“阈值”时，只需更改一次这里的“90”这个数值。
unsigned char a;
unsigned char b;

//第2个：运算式的例子
#define C (2+6) //有括号
#define D 2+6 //无括号
unsigned char x=3;
```

```
unsigned char c;  
unsigned char d;
```

//第 3 个：函数的例子

```
unsigned char e=1;  
void add(void);  
void add(void)  
{  
    e++;  
}  
#define a_zi_jia add() //用字符串 a_zi_jia 来替代函数 add()。
```

```
void main() //主函数  
{
```

//第 1 个：常量的例子

```
if(89>=Cu8Level) //大于或者等于阈值，就输出 1。  
{  
    a=1;  
}  
else //否则输出 0。  
{  
    a=0;  
}  
if(95>=Cu8Level) //大于或者等于阈值，就输出 1。  
{  
    b=1;  
}  
else //否则输出 0。  
{  
    b=0;  
}
```

//第 2 个：运算式的例子

```
c=x*C; //等效于 c=x*(2+6)，最终运算结果 c 等于 24。因为 3 乘以 8（2 加上 6 等于 8）。  
d=x*D; //等效于 d=x*2+6，最终运算结果 d 等于 12。因为 3 乘以 2 等于 6，6 再加 6 等于 12。
```

//第 3 个：函数的例子

```
a_zi_jia; //这里相当于调用函数 add()。e 从 1 自加到 2。  
a_zi_jia; //这里相当于调用函数 add()。e 从 2 自加到 3。
```

```
View(a); //把第 1 个数 a 发送到电脑端的串口助手软件上观察。
View(b); //把第 2 个数 b 发送到电脑端的串口助手软件上观察。
View(c); //把第 3 个数 c 发送到电脑端的串口助手软件上观察。
View(d); //把第 4 个数 d 发送到电脑端的串口助手软件上观察。
View(e); //把第 5 个数 e 发送到电脑端的串口助手软件上观察。

while(1)
{
}
}
/*---C 语言学习区域的结束。-----*/
```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第 1 个数

十进制:0

十六进制:0

二进制:0

第 2 个数

十进制:1

十六进制:1

二进制:1

第 3 个数

十进制:24

十六进制:18

二进制:11000

第 4 个数

十进制:12

十六进制:C

二进制:1100

第 5 个数

十进制:3

十六进制:3

二进制:11

分析：

a 为 0。

b 为 1。  
c 为 24。  
d 为 12。  
e 为 3。

### 【59.9 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。



## 第六十节： 指针在变量(或常量)中的基础知识。

### 【60.1 指针与普通变量的对比。】

普通变量和指针都是变量，都要占用 RAM 资源。普通变量的 unsigned char 类型占用 1 个字节，unsigned int 类型占用 2 个字节，unsigned long 类型占用 4 个字节。但是指针不一样，指针是一种特殊的变量，unsigned char\*, unsigned int\*, unsigned long\* 这三类指针在 C51 编译器下都是一样占用 3 个字节。不同系统的指针到底占用多少个字节，是由 C 编译器根据芯片的硬件寻址范围决定的，比如 32 位单片机的指针往往都是 4 个字节，而某些 64 位的 PC 机，指针可能是 8 个字节，这些内容大家只要有大概的了解即可。指针是普通变量的载体，平时我们处理普通变量，都是可以“直接”操作普通变量本身。而学了指针之后，我们就多一种选择，可以通过指针这个载体来“间接”操作某个普通变量。“直接”不是比“间接”更好更高效吗？为什么要用“间接”？其实在某些场合，指针的“间接”操作更加灵活更加高效，这个要看具体的应用。

指针既然是普通变量的“载体”，那么普通变量就是“物”。“载体”与“物”之间可以存在一对多的关系。也就是说，一个篮子（载体），可以盛放鸡蛋（物），也可以盛放青菜（物），也可以盛放水果（物）。但是，在这里，一个篮子在一个时间段内，只能承载一种物品，如果想承载其它物品，必须先把当前物品“卸”下来，然后再“装”其它物品”。这里有两个关键动作“装”和“卸”，就是指针在处理普通变量时的“绑定”，某个指针与某个变量发生“绑定”，就已经包含了先“卸”后“装”这两个动作在其中。

题外话多说一句，刚才提到，unsigned int 类型占用 2 个字节，这个是在 C51 编译器下的情况。如果是在 stm32 单片机的编译器下，unsigned int 类型是占用 4 个字节。

### 【60.2 指针的定义。】

跟普通变量一样，指针也必须先定义再使用。为了与普通变量区分开来，指针在定义的时候多加了一个星号“\*”，例子如下：

```
unsigned char* pu8;    //针对 unsigned char 类型变量的指针。凡是指针都是占 4 个字节！
unsigned int* pu16;    //针对 unsigned int 类型变量的指针。凡是指针都是占 4 个字节！
unsigned long* pu32;   //针对 unsigned long 类型变量的指针。凡是指针都是占 4 个字节！
```

既然指针都是 4 个字节，为什么还要区分 unsigned char\*, unsigned int\* pu16, unsigned long\* pu32 这三种类型？因为指针是为普通变量（或常量）而生，所以要根据普通变量（或常量）的类型定义对应的指针。

### 【60.3 指针与普通变量是如何关联和操作的？】

指针在操作某个变量的时候，必须先跟某个变量关联起来，这里的关联就是“绑定”。“绑定”后，才可以通过指针这个“载体”来“间接”操作变量。指针与普通变量在“绑定”的时候，需要用到“&”这个符号。例子如下：

```
unsigned char* pu8;    //针对 unsigned char 类型变量的指针。凡是指针都是占 4 个字节！
unsigned char a=0;     //普通的变量。
pu8=&a;                //指针与普通变量发生关联（或者说绑定）。
*pu8=2;                //通过指针这个载体来处理 a 这个变量，此时 a 从原来的 0 变成了 2。
```

## 【60.4 指针处理“批量数据”的基础知识。】

之所以有通过载体来“间接”操作普通变量的存在价值，其中很重要的原因是指针在处理“批量数据”时特别给力，这里的“批量数据”是有条件的，要求这些数据的地址必须挨家挨户连起来的，不能是零零散散的“散户”，比如说，数组就是由一堆在 RAM 空间里地址连续的变量组合而成，指针在很多时候就是为数组而生的。先看一个例子如下：

```
unsigned char* pu8;    //针对 unsigned char 类型变量的指针。凡是指针都是占 4 个字节！
unsigned char Buffer[3];    //普通的数组，内含 3 个变量，它们地址是相连的。

pu8=&Buffer[0];    //指针与普通变量 Buffer[0]发生关联（或者说绑定）。
*pu8=1;            //通过指针这个载体来处理 Buffer[0]这个变量，此时 Buffer[0]变成了 1。

pu8=&Buffer[1];    //指针与普通变量 Buffer[1]发生关联（或者说绑定）。
*pu8=2;            //通过指针这个载体来处理 Buffer[1]这个变量，此时 Buffer[1]变成了 2。

pu8=&Buffer[2];    //指针与普通变量 Buffer[2]发生关联（或者说绑定）。
*pu8=3;            //通过指针这个载体来处理 Buffer[2]这个变量，此时 Buffer[2]变成了 3。
```

分析：上述例子中，并没有体现出指针的优越性，因为数组有 3 个元素，居然要绑定了 3 次，如果数组有 1000 个元素，难道要绑定 1000 次？显然这样是繁琐低效不可取的。而要发挥指针的优越性，我们现在必须深入了解一下指针的本质是什么，指针跟普通变量发生“绑定”的本质是什么。普通变量由“地址”和“地址所装的数据”构成，指针是特殊的变量，它是由什么构成呢？其实，指针是由“地址”和“地址所装的变量（或常量）的地址”组成。很明显，一个重要的区别是，普通变量装的数据，而指针装的是地址。正因为指针装的是地址，所以指针可以有两种选择，第一种可以处理“装的地址”，第二种可以处理“装的地址的所在数据”，这两种能力，就是指针的精华和本质所在，也是跟普通变量的区别所在。那么指针处理“装的地址”的语法是什么样子的？请看例子如下：

```
unsigned char* pu8;    //针对 unsigned char 类型变量的指针。凡是指针都是占 4 个字节！
unsigned char Buffer[3];    //普通的数组，内含 3 个变量，它们地址是相连的。

pu8=&Buffer[0];    //处理“装的地址”。把 Buffer[0]变量的地址装在指针这个载体里。
*pu8=1;            //处理“装的地址的所在数据”。此时 Buffer[0]变成了 1。

pu8++;            //处理“装的地址”。这里是“地址”自加 1，相当于指针此时装的是 Buffer[1]的地址。
*pu8=2;            //处理“装的地址的所在数据”。此时 Buffer[1]变成了 2。

pu8++;            //处理“装的地址”。这里是“地址”自加 1，相当于指针此时装的是 Buffer[2]的地址。
*pu8=3;            //处理“装的地址的所在数据”。此时 Buffer[2]变成了 3。
```

上述例子中，利用“地址”自加 1 的操作，省去了 2 条赋值式的“绑定”操作（比如像 `pu8=&Buffer[0]` 这类语句），因此“绑定”本质其实就是更改指针所装的“变量（或常量）的地址”的操作。此例子中虽然还没体现了出指针在数组处理时的优越性，但是利用指针处理“装的地址”这项功能，在实际项目中很容易

发现它的好处。

### 【60.5 指针与数组关联（绑定）时省略“&和下标[0]”的写法。】

指针与数组关联的时候，通常是跟数组的第 0 个元素的地址关联，此时，可以把数组的“&和下标[0]”省略，比如：

```
unsigned char* pu8;
unsigned char Buffer[3];
pu8=Buffer;      //此行代码省略了“&和下标[0]”，等效于 pu8=&Buffer[0];
```

### 【60.6 带 const 关键字的常量指针。】

指针也可以跟常量关联起来，处理常量，但是常量只能“读”不能“写”，所以通过指针操作常量的时候也是只能“读”不能“写”。操作常量的指针用 const 关键词修饰，强调此指针只有“读”的操作。例子如下：

```
const unsigned char* pCu8;    //常量指针
code char Cu8Buffer[3]={5,6,7}; //常量数组
unsigned char b;
unsigned char c;
unsigned char d;

pCu8=Cu8Buffer; //此行代码省略了“&和下标[0]”，等效于 pCu8=&Cu8Buffer[0];
b=*pCu8;        //读“装的地址的所在数据”。b 等于 5。

pCu8++;         //所装的地址自加 1，跟 Cu8Buffer[1]关联
c=*pCu8;        //读“装的地址的所在数据”。c 等于 6。

pCu8++;         //所装的地址自加 1，跟 Cu8Buffer[2]关联
d=*pCu8;        //读“装的地址的所在数据”。d 等于 7。
```

### 【60.7 例程练习和分析。】

现在编一个练习程序来熟悉指针的基础知识。

```
/*---C 语言学习区域的开始。-----*/

unsigned char* pu8;      //针对 unsigned char 类型变量的指针。凡是指针都是占 4 个字节！
unsigned char a=0;       //普通的变量。
unsigned char Buffer[3];  //普通的数组，内含 3 个变量，它们地址是相连的。

const unsigned char* pCu8; //常量指针
code char Cu8Buffer[3]={5,6,7}; //常量数组
```

```

unsigned char b;
unsigned char c;
unsigned char d;

void main() //主函数
{

    pu8=&a; //指针与普通变量发生关联（或者说绑定）。
    *pu8=2; //通过指针这个载体来处理 a 这个变量，此时 a 从原来的 0 变成了 2。

    pu8=&Buffer[0]; //处理“装的地址”。把 Buffer[0]变量的地址装在指针这个载体里。
    *pu8=1; //处理“装的地址的所在数据”。此时 Buffer[0]变成了 1。

    pu8++; //处理“装的地址”。这里是“地址”自加 1，相当于指针此时装的是 Buffer[1]的地址。
    *pu8=2; //处理“装的地址的所在数据”。此时 Buffer[1]变成了 2。

    pu8++; //处理“装的地址”。这里是“地址”自加 1，相当于指针此时装的是 Buffer[2]的地址。
    *pu8=3; //处理“装的地址的所在数据”。此时 Buffer[2]变成了 3。

    pCu8=Cu8Buffer; //此行代码省略了“&和下标[0]”，等效于 pCu8=&Cu8Buffer[0];
    b=*pCu8; //读“装的地址的所在数据”。b 等于 5。

    pCu8++; //所装的地址自加 1，跟 Cu8Buffer[1]关联
    c=*pCu8; //读“装的地址的所在数据”。c 等于 6。

    pCu8++; //所装的地址自加 1，跟 Cu8Buffer[2]关联
    d=*pCu8; //读“装的地址的所在数据”。d 等于 7。

    View(a); //把第 1 个数 a 发送到电脑端的串口助手软件上观察。
    View(b); //把第 2 个数 b 发送到电脑端的串口助手软件上观察。
    View(c); //把第 3 个数 c 发送到电脑端的串口助手软件上观察。
    View(d); //把第 4 个数 d 发送到电脑端的串口助手软件上观察。
    View(Buffer[0]); //把第 5 个数 Buffer[0]发送到电脑端的串口助手软件上观察。
    View(Buffer[1]); //把第 6 个数 Buffer[1]发送到电脑端的串口助手软件上观察。
    View(Buffer[2]); //把第 7 个数 Buffer[2]发送到电脑端的串口助手软件上观察。

    while(1)
    {
    }
}

/*---C 语言学习区域的结束。-----*/

```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第 1 个数

十进制:2

十六进制:2

二进制:10

第 2 个数

十进制:5

十六进制:5

二进制:101

第 3 个数

十进制:6

十六进制:6

二进制:110

第 4 个数

十进制:7

十六进制:7

二进制:111

第 5 个数

十进制:1

十六进制:1

二进制:1

第 6 个数

十进制:2

十六进制:2

二进制:10

第 7 个数

十进制:3

十六进制:3

二进制:11

分析：

a 为 2。

b 为 5。

c 为 6。

d 为 7。

Buffer[0]为 1。

Buffer[1]为 2。

Buffer[2]为 3。

### 【60.8 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。

## 第六十一节： 指针的中转站作用，地址自加法，地址偏移法。

### 【61.1 指针与批量数组的关系。】

指针和批量数据的关系，更像领导和团队的关系，领导是团队的代表，所以当需要描述某个团队的时候，为了表述方便，可以把由 N 个人组成的团队简化成该团队的一个领导，用一个领导来代表整个团队，此时，领导就是团队，团队就是领导。指针也一样，指针一旦跟某堆数据“绑定”了，那么指针就是这堆数据，这堆数据就是该指针，所以在很多 PC 上位机的项目中，往往也把指针称呼为“句柄”，字面上理解，就是一句话由 N 个文字组成，而“句柄”就是这句话的代表，实际上“句柄”往往是某一堆资源的代表。不管是把指针比喻成“领导”、“代表”还是“句柄”，指针在这里都有“中间站”这一层含义。

### 【61.2 指针在批量数据的“中转站”作用。】

指针在批量数据处理中，主要是能节省代码容量，而且是非常直观的节省代码容量。为什么能节省代码容量？是因为可以把某些重复性的具体实现的功能封装成指针来操作，请看下面的例子：

程序要求：根据一个选择变量 Gu8Sec 的值，要从三堆数据中选择对应的一堆数据放到数组 Gu8Buffer 里。当 Gu8Sec 等于 1 的时候选择第 1 堆，等于 2 的时候选择第 2 堆，等于 3 的时候选择第 3 堆。也就是“三选一”。

第 1 种实现的方法：没有用指针，最原始的处理方式。如下：

```
code unsigned char Cu8Memory_1[3]={1,2,3}; //第 1 堆数据
code unsigned char Cu8Memory_2[3]={4,5,6}; //第 2 堆数据
code unsigned char Cu8Memory_3[3]={7,8,9}; //第 3 堆数据

unsigned char Gu8Sec=2; //选择的变量
unsigned char Gu8Buffer[3]; //根据变量来存放对应的某堆数据的数组
unsigned char i; //for 循环用到的变量 i

switch(Gu8Sec) //根据此选择变量来切换到对应的操作上
{
    case 1: //第 1 堆
        for(i=0;i<3;i++) //第 1 次出现 for 循环，用来实现“赋值”的“搬运数据”的动作。
        {
            Gu8Buffer[i]=Cu8Memory_1[i];
        }
        break;

    case 2: //第 2 堆
        for(i=0;i<3;i++) //第 2 次出现 for 循环，用来实现“赋值”的“搬运数据”的动作。
        {
            Gu8Buffer[i]=Cu8Memory_2[i];
        }
}
```

```

        break;

    case 3: //第 3 堆
        for(i=0;i<3;i++) //第 3 次出现 for 循环，用来实现“赋值”的“搬运数据”的动作。
        {
            Gu8Buffer[i]=Cu8Memory_3[i];
        }
        break;
}

```

分析：上述程序中，没有用到指针，出现了 3 次 for 循环的“赋值”的“搬运数据”的动作。

第 2 种实现的方法：用指针作为“中间站”。如下：

```

code unsigned char Cu8Memory_1[3]={1,2,3}; //第 1 堆数据
code unsigned char Cu8Memory_2[3]={4,5,6}; //第 2 堆数据
code unsigned char Cu8Memory_3[3]={7,8,9}; //第 3 堆数据

unsigned char Gu8Sec=2; //选择的变量
unsigned char Gu8Buffer[3]; //根据变量来存放对应的某堆数据的数组
unsigned char i; //for 循环用到的变量 i
const unsigned char *pCu8; //引入一个指针作为“中间站”

switch(Gu8Sec) //根据此选择变量来切换到对应的操作上
{
    case 1: //第 1 堆
        pCu8=&Cu8Memory_1[0]; //跟第 1 堆数据“绑定”起来。
        break;

    case 2: //第 2 堆
        pCu8=&Cu8Memory_2[0]; //跟第 2 堆数据“绑定”起来。
        break;

    case 3: //第 3 堆
        pCu8=&Cu8Memory_3[0]; //跟第 3 堆数据“绑定”起来。
        break;
}

for(i=0;i<3;i++) //第 1 次出现 for 循环，用来实现“赋值”的“搬运数据”的动作。
{
    Gu8Buffer[i]=*pCu8; //把“指针所存的地址的数据”赋值给数组
    pCu8++; //“指针所存的地址”自加 1，为下一个数据的“赋值”的“搬运”作准备。
}

```



分析：上述程序中，用到了指针作为中间站，只出现了 1 次 for 循环的“赋值”的“搬运数据”的动作。对比之前第 1 种方法，在本例子中，用了指针之后，程序代码看起来更加高效简洁清爽省容量。在实际项目中，数据量越大的时候，指针这种“优越性”就越明显。

### 【61.3 指针在书写上另外两种常用写法。】

刚才 61.2 处第 2 个例子中，有一段代码如下：

```
for(i=0;i<3;i++) //第 1 次出现 for 循环，用来实现“赋值”的“搬运数据”的动作。
{
    Gu8Buffer[i]=*pCu8; //把“指针所存的地址的数据”赋值给数组
    pCu8++; //“指针所存的地址”自加 1，为下一个数据的“赋值”的“搬运”作准备。
}
```

很多高手，喜欢把上面 for 循环内部的那两行代码简化成一行代码，如下：

```
for(i=0;i<3;i++) //第 1 次出现 for 循环，用来实现“赋值”的“搬运数据”的动作。
{
    Gu8Buffer[i]=*pCu8++; //先把“数据”赋值给数组，然后“指针所存的地址”再自加 1。
}
```

上面这种写法也是合法的，而且在高手的代码中常见，据说也是最高效的写法。还有一种是利用“指针的偏移地址”的写法，我常用这种写法，因为感觉这种写法比较直观，而且跟数组的书写很像。如下：

```
for(i=0;i<3;i++) //第 1 次出现 for 循环，用来实现“赋值”的“搬运数据”的动作。
{
    Gu8Buffer[i]=pCu8[i]; //这类是“偏移地址”的写法，i 在这里相当于指针的偏移地址。
}
```

这种写法也是跟前面那两种写法在程序实现的功能上是一样的，是等效的，我常用这种写法。

### 【61.4 指针的“地址自加法”和“地址偏移法”的差别。】

刚才 61.3 处讲了 3 个例子，其中前面的两个例子都是属于“地址自加法”，而最后的那一个是属于“地址偏移法”。它们的根本差别是：“地址自加法”的时候，“指针所存的地址”是变动的；而“地址偏移法”的时候，“指针所存的地址”是不变的，“指针所存的地址”的“不变”的属性，就像某个原点，原点再加上偏移，就可以寻址到某个新的 RAM 地址所存的数据。例子如下：

第 1 种：“地址自加法”：

```
pCu8=&Cu8Memory_2[0]; //假设赋值后，此时“指针所存的地址”是 RAM 的地址 4。
for(i=0;i<3;i++)
{
```

```
    Gu8Buffer[i]=*pCu8++; //先把“数据”赋值给数组，然后“指针所存的地址”再自加1。  
}
```

分析：上述代码，等程序执行完 for 循环后，指针所存的地址还是 RAM 地址 4 吗？不是。因为它是变动的，经过 for 循环，“指针所存的地址”自加 3 次后，此时“所存的 RAM 地址”从原来的 4 变成了 7。

第 2 种：“地址偏移法”：

```
pCu8=&Cu8Memory_2[0]; //假设赋值后，此时“指针所存的地址”是 RAM 的地址 4。  
for(i=0;i<3;i++)  
{  
    Gu8Buffer[i]=pCu8[i]; //这类是“偏移地址”的写法，i 在这里相当于指针的偏移地址。  
}
```

分析：上述代码，等程序执行完 for 循环后，指针所存的地址还是 RAM 地址 4 吗？是的。因为它存的地址是不变的，变的只是偏移地址 i。此时“指针所存的地址”就像“原点”一样具有“绝对地址”的“参考点”的属性。

## 【61.5 例程练习和分析。】

现在编一个练习程序。

```
/*---C 语言学习区域的开始。-----*/  
  
code unsigned char Cu8Memory_1[3]={1,2,3}; //第 1 堆数据  
code unsigned char Cu8Memory_2[3]={4,5,6}; //第 2 堆数据  
code unsigned char Cu8Memory_3[3]={7,8,9}; //第 3 堆数据  
  
unsigned char Gu8Sec=2; //选择的变量  
unsigned char Gu8Buffer[3]; //根据变量来存放对应的某堆数据的数组  
unsigned char i; //for 循环用到的变量 i  
const unsigned char *pCu8; //引入一个指针作为“中间站”  
  
void main() //主函数  
{  
  
    switch(Gu8Sec) //根据此选择变量来切换到对应的操作上  
    {  
        case 1: //第 1 堆  
            pCu8=&Cu8Memory_1[0]; //跟第 1 堆数据“绑定”起来。  
            break;  
  
        case 2: //第 2 堆  
            pCu8=&Cu8Memory_2[0]; //跟第 2 堆数据“绑定”起来。
```

```

        break;

    case 3: //第 3 堆
        pCu8=&Cu8Memory_3[0]; //跟第 3 堆数据“绑定”起来。
        break;
}

// for(i=0;i<3;i++) //第 1 次出现 for 循环，用来实现“赋值”的“搬运数据”的动作。
// {
//     Gu8Buffer[i]=*pCu8++; //先把“数据”赋值给数组，然后“指针所存的地址”再自加 1。
// }

for(i=0;i<3;i++) //第 1 次出现 for 循环，用来实现“赋值”的“搬运数据”的动作。
{
    Gu8Buffer[i]=pCu8[i]; //这类是“偏移地址”的写法，i 在这里相当于指针的偏移地址。
}

View(Gu8Buffer[0]); //把第 1 个数 Gu8Buffer[0]发送到电脑端的串口助手软件上观察。
View(Gu8Buffer[1]); //把第 2 个数 Gu8Buffer[1]发送到电脑端的串口助手软件上观察。
View(Gu8Buffer[2]); //把第 3 个数 Gu8Buffer[2]发送到电脑端的串口助手软件上观察。

while(1)
{
}
}
/*---C 语言学习区域的结束。-----*/

```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第 1 个数

十进制:4

十六进制:4

二进制:100

第 2 个数

十进制:5

十六进制:5

二进制:101

第 3 个数

十进制:6

十六进制:6

二进制:110

分析:

Gu8Buffer[0]为 4。

Gu8Buffer[1]为 5。

Gu8Buffer[2]为 6。

### 【61.6 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。

## 第六十二节： 指针，大小端，化整为零，化零为整。

### 【62.1 内存的大小端。】

C51 编译器的 unsigned int 占 2 字节 RAM（也称为内存），unsigned long 占 4 字节 RAM，这两种数据类型所占的字节数都超过了 1 个字节，而 RAM 内存是每一个地址对应一个字节 RAM 内存，那么问题就来了，比如像 unsigned long 这种占 4 个字节 RAM 的数据变量，它这 4 个字节在 RAM 中的地址是“连续”的“挨家挨户”的“连号”的，这 4 个字节所存的一个数据，它的数据高低位在地址的排列上，到底是从低到高还是从高到低，到底是“正向”的还是“反向”？这两种不同的排列顺序，在 C 语言里用“大端”和“小端”这两个专业术语来描述。“大端”的方式是将高位存放在低地址，“小端”的方式是将低位存放在低地址。比如：

假设有一个 unsigned long 变量 a 等于 0x12345678，是存放在 RAM 内存中的 4, 5, 6, 7 这四个“连号”的地址里，现在看看它在“大端”和“小端”的存储方式里的差别。如下：

（1）在“大端”的方式里，将高位存放在低地址。

0x12 存在第 4 个地址，0x34 存在第 5 个地址，0x56 存在第 6 个地址，0x78 存在第 7 个地址。

（2）在“小端”的方式里，将低位存放在低地址。

0x78 存在第 4 个地址，0x56 存在第 5 个地址，0x34 存在第 6 个地址，0x12 存在第 7 个地址。

问题来了，在单片机里，内存到底是“大端”方式还是“小端”方式？答：这个跟 C 编译器有关。比如，在 51 单片机的 C51 编译环境里是“大端”方式，而在 stm32 单片机的 ARM\_MDK 编译环境里则是“小端”方式。那么问题又来了？如何知道一个 C 编译器是“大端”还是“小端”？答：有两种方式，一种是看 C 编译器的说明书，另一种是自己编写一个小程序测试一下就知道了（这种方法最简单可靠）。那么问题又来了？讲这个“大小端”有什么用？答：这个跟指针的使用密切相关。

### 【62.2 化整为零。】

在数据的存储和通信中，往往要先把数据转换成以字节为单位的数组，才能进行数据存储和通信。比如 unsigned long 这种类型的数据，就要先转换成 4 个字节，这种把某个变量转换成 N 个字节的过程，就是“化整为零”。“化整为零”的过程，在代码上，有两种常见的方式，一种是原始的“移位法”，另一种是极具优越性的“指针法”。比如，现在以“大端”方式为例（因为本教程是用 C51 编译器，C51 编译器是“大端”方式），有一个 unsigned long 变量 a 等于 0x12345678，要把这个变量分解成 4 个字节存放在一个数组 Gu8BufferA 中，现在跟大家分享和对比一下这两种方法。

（1）原始的“移位法”。

```
unsigned long a=0x12345678;
unsigned char Gu8BufferA[4];

Gu8BufferA[0]=a>>24;
Gu8BufferA[1]=a>>16;
Gu8BufferA[2]=a>>8;
Gu8BufferA[3]=a;
```

(2) 极具优越性的“指针法”。

```
unsigned long a=0x12345678;
unsigned char Gu8BufferA[4];
unsigned long *pu32;    //引入一个指针变量，注意，这里是 unsigned long 类型的指针。

pu32=(unsigned long *)&Gu8BufferA[0]; //指针跟数组“绑定”(也称为“关联”)起来。
*pu32=a; //这里仅仅 1 行代码就等效于上述 (1)“移位”例子中的 4 行代码，所以极具优越性。
```

多说一句，“pu32=(unsigned long \*)&Gu8BufferA[0]”这行代码中，其中小括号“(unsigned long \*)”是表示数据的强制类型转换，这里表示强制转换成 unsigned long 的指针方式，以后这类代码写多了，就会发现这种书写方法的规律。作为语言来解读先熟悉一下它的表达方式就可以了，暂时不用深究它的含义。

### 【62.3 化零为整。】

从数据存储中提取数据出来，从通讯端接收到一堆数据，这里的“提取”和“接收”都是以字节为单位的数据，所以为了“还原”成原来的类型变量，就涉及“化零为整”的过程。在代码上，有两种常见的方式，一种是原始的“移位法”，另一种是极具优越性的“指针法”。比如，现在以“大端”方式为例（因为本教程是用 C51 编译器，C51 编译器是“大端”方式），有一个数组 Gu8BufferB 存放了 4 个字节数据分别是：0x12, 0x34, 0x56, 0x78。现在要把这 4 个字节数据“合并”成一个 unsigned long 类型的变量 b，这个变量 b 等于 0x12345678。现在跟大家分享和对比一下这两种方法。

(1) 原始的“移位法”。

```
unsigned char Gu8BufferB[4]={0x12, 0x34, 0x56, 0x78};
unsigned long b;

b=Gu8BufferB[0];
b=b<<8;
b=b+Gu8BufferB[1];
b=b<<8;
b=b+Gu8BufferB[2];
b=b<<8;
b=b+Gu8BufferB[3];
```

(2) 极具优越性的“指针法”。

```
unsigned char Gu8BufferB[4]={0x12, 0x34, 0x56, 0x78};
unsigned long b;
unsigned long *pu32;    //引入一个指针变量，注意，这里是 unsigned long 类型的指针。

pu32=(unsigned long *)&Gu8BufferB[0]; //指针跟数组“绑定”(也称为“关联”)起来。
b=*pu32; //这里仅仅 1 行代码就等效于上述 (1)“移位”例子中的 7 行代码，所以极具优越性。
```

### 【62.4 “指针法”要注意的问题。】

“化整为零”和“化零为整”其实是一个“互逆”的过程，在使用“指针法”的时候，一定要注意“大

小端”的问题。“化整为零”和“化零为整”这两个“互逆”过程要么同时为“大端”，要么同时为“小端”，否则会因字节的排列顺序问题而引起数据的严重错误。

## 【62.5 例程练习和分析。】

现在编一个练习程序。

```
/*---C 语言学习区域的开始。-----*/

unsigned long a=0x12345678;
unsigned char Gu8BufferA[4];

unsigned char Gu8BufferB[4]={0x12, 0x34, 0x56, 0x78};
unsigned long b;

unsigned long *pu32;    //引入一个指针变量，注意，这里是 unsigned long 类型的指针。

void main() //主函数
{
    pu32=(unsigned long *)&Gu8BufferA[0]; //指针跟数组“绑定”（也称为“关联”）起来。
    *pu32=a; //化整为零

    pu32=(unsigned long *)&Gu8BufferB[0]; //指针跟数组“绑定”（也称为“关联”）起来。
    b=*pu32; //化零为整

    View(Gu8BufferA[0]); //把第 1 个数 Gu8BufferA[0]发送到电脑端的串口助手软件上观察。
    View(Gu8BufferA[1]); //把第 2 个数 Gu8BufferA[1]发送到电脑端的串口助手软件上观察。
    View(Gu8BufferA[2]); //把第 3 个数 Gu8BufferA[2]发送到电脑端的串口助手软件上观察。
    View(Gu8BufferA[3]); //把第 4 个数 Gu8BufferA[3]发送到电脑端的串口助手软件上观察。

    View(b); //把第 5 个数 b 发送到电脑端的串口助手软件上观察。

    while(1)
    {
    }
}
/*---C 语言学习区域的结束。-----*/
```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第 1 个数

十进制:18  
十六进制:12  
二进制:10010

第 2 个数  
十进制:52  
十六进制:34  
二进制:110100

第 3 个数  
十进制:86  
十六进制:56  
二进制:1010110

第 4 个数  
十进制:120  
十六进制:78  
二进制:1111000

第 5 个数  
十进制:305419896  
十六进制:12345678  
二进制:10010001101000101011001111000

分析:

Gu8BufferA[0]为 0x12。  
Gu8BufferA[1]为 0x34。  
Gu8BufferA[2]为 0x56。  
Gu8BufferA[3]为 0x78。  
b 为 0x12345678。

## 【62.6 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。



## 第六十三节： 指针“化整为零”和“化零为整”的“灵活”应用。

### 【63.1 化整为零的“灵活”应用。】

上一节讲“化整为零”的例子，指针是跟数组的首地址（下标是0）“绑定”的，这样，很多初学者就误以为指针跟数组“绑定”时，只能跟数组的“首地址”关联。其实，指针是可以跟数组的任何一个成员的地址“绑定”（只要不超过数组的长度导致越界），它不仅仅局限于首地址，指针的这个特征就是本节标题所说的“灵活”。请看下面这个例子：

有3个变量，分别是单字节 unsigned char a，双字节 unsigned int b，四字节 unsigned long c，它们加起来一共有7个字节，要把这7个字节放到一个7字节容量的数组里。除了用传统的“移位法”，还有一种更加便捷的“指针法”，代码如下：

```
unsigned char a=0x01;
unsigned int b=0x0203;
unsigned long c=0x04050607;

unsigned char Gu8BufferABC[7]; //存放3个不同长度变量的数组

unsigned char *pu8;   //引入的 unsigned char 类型指针
unsigned int *pu16;   //引入的 unsigned int 类型指针
unsigned long *pu32;  //引入的 unsigned long 类型指针

pu8=&Gu8BufferABC[0]; //指针跟数组的第0个位置“绑定”起来。
*pu8=a; //把a的1个字节放在数组第0个位置。

pu16=(unsigned int *)&Gu8BufferABC[1]; //指针跟数组的第1个位置“绑定”起来。
*pu16=b; //把b的2个字节放在数组第1、2这两个位置。

pu32=(unsigned long *)&Gu8BufferABC[3]; //指针跟数组的第3个位置“绑定”起来。
*pu32=c; //把c的4个字节放在数组第3、4、5、6这四个位置。
```

### 【63.2 化零为整的“灵活”应用。】

刚才讲的是“化整为零”，现在讲的是“化零为整”。刚才讲的是“分解”，现在讲的是“合成”。请看下面这个例子：

有一个容量为7字节数组，第0字节存放的是 unsigned char d 变量，第1、2字节存放的是 unsigned int e 变量，第3、4、5、6字节存放的是 unsigned long f 变量，现在要从数组中“零散”的字节里提取并且合成为“完整”的3个变量。代码如下：

```
unsigned char Gu8BufferDEF[7]={0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07}; //注意大小端的问题

unsigned char d;
```

```

unsigned int e;
unsigned long f;

unsigned char *pu8;    //引入的 unsigned char 类型指针
unsigned int *pu16;    //引入的 unsigned int 类型指针
unsigned long *pu32;   //引入的 unsigned long 类型指针

pu8=&Gu8BufferDEF[0]; //指针跟数组的第 0 个位置“绑定”起来。
d=*pu8;               //从数组第 0 位置提取单字节完整的 d 变量。

pu16=(unsigned int *)&Gu8BufferDEF[1]; //指针跟数组的第 1 个位置“绑定”起来。
e=*pu16;               //从数组第 1, 2 位置提取双字节完整的 e 变量。

pu32=(unsigned long *)&Gu8BufferDEF[3]; //指针跟数组的第 3 个位置“绑定”起来。
f=*pu32;               //从数组第 3, 4, 5, 6 位置提取四字节完整的 f 变量。

```

### 【63.3 例程练习和分析。】

现在编一个练习程序。

```

/*---C 语言学习区域的开始。-----*/

unsigned char a=0x01;
unsigned int b=0x0203;
unsigned long c=0x04050607;
unsigned char Gu8BufferABC[7]; //存放 3 个不同长度变量的数组

unsigned char Gu8BufferDEF[7]={0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07}; //注意大小端的问题
unsigned char d;
unsigned int e;
unsigned long f;

unsigned char *pu8;    //引入的 unsigned char 类型指针
unsigned int *pu16;    //引入的 unsigned int 类型指针
unsigned long *pu32;   //引入的 unsigned long 类型指针

void main() //主函数
{
    //第 1 类例子：化整为零。
    pu8=&Gu8BufferABC[0]; //指针跟数组的第 0 个位置“绑定”起来。
    *pu8=a; //把 a 的 1 个字节放在数组第 0 个位置。
}

```

```

    pu16=(unsigned int *)&Gu8BufferABC[1]; //指针跟数组的第 1 个位置“绑定”起来。
    *pu16=b;                               //把 b 的 2 个字节放在数组第 1、2 这两个位置。

    pu32=(unsigned long *)&Gu8BufferABC[3]; //指针跟数组的第 3 个位置“绑定”起来。
    *pu32=c;                               //把 c 的 4 个字节放在数组第 3、4、5、6 这四个位置。

    //第 2 类例子：化零为整。
    pu8=&Gu8BufferDEF[0]; //指针跟数组的第 0 个位置“绑定”起来。
    d=*pu8;               //从数组第 0 位置提取单字节完整的 d 变量。

    pu16=(unsigned int *)&Gu8BufferDEF[1]; //指针跟数组的第 1 个位置“绑定”起来。
    e=*pu16;                               //从数组第 1,2 位置提取双字节完整的 e 变量。

    pu32=(unsigned long *)&Gu8BufferDEF[3]; //指针跟数组的第 3 个位置“绑定”起来。
    f=*pu32;                               //从数组第 3,4,5,6 位置提取四字节完整的 f 变量。

    View(Gu8BufferABC[0]); //把第 1 个数 Gu8BufferABC[0]发送到电脑端的串口助手软件上观察。
    View(Gu8BufferABC[1]); //把第 2 个数 Gu8BufferABC[1]发送到电脑端的串口助手软件上观察。
    View(Gu8BufferABC[2]); //把第 3 个数 Gu8BufferABC[2]发送到电脑端的串口助手软件上观察。
    View(Gu8BufferABC[3]); //把第 4 个数 Gu8BufferABC[3]发送到电脑端的串口助手软件上观察。
    View(Gu8BufferABC[4]); //把第 5 个数 Gu8BufferABC[4]发送到电脑端的串口助手软件上观察。
    View(Gu8BufferABC[5]); //把第 6 个数 Gu8BufferABC[5]发送到电脑端的串口助手软件上观察。
    View(Gu8BufferABC[6]); //把第 7 个数 Gu8BufferABC[6]发送到电脑端的串口助手软件上观察。

    View(d);           //把第 8 个数 d 发送到电脑端的串口助手软件上观察。
    View(e);           //把第 9 个数 e 发送到电脑端的串口助手软件上观察。
    View(f);           //把第 10 个数 f 发送到电脑端的串口助手软件上观察。

    while(1)
    {
    }
}
/*---C 语言学习区域的结束。-----*/

```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第 1 个数

十进制:1

十六进制:1

二进制:1

第 2 个数

十进制:2

十六进制:2

二进制:10

第 3 个数

十进制:3

十六进制:3

二进制:11

第 4 个数

十进制:4

十六进制:4

二进制:100

第 5 个数

十进制:5

十六进制:5

二进制:101

第 6 个数

十进制:6

十六进制:6

二进制:110

第 7 个数

十进制:7

十六进制:7

二进制:111

第 8 个数

十进制:1

十六进制:1

二进制:1

第 9 个数

十进制:515

十六进制:203

二进制:1000000011

第:个数（这里是第 10 个数。本模块程序只支持显示第 1 到第 9 个,所以这里没有显示“10”）

十进制:67438087

十六进制:4050607

二进制:100000001010000011000000111

分析:

Gu8BufferABC[0]为 0x01。

Gu8BufferABC[1]为 0x02。

Gu8BufferABC[2]为 0x03。

Gu8BufferABC[3]为 0x04。

Gu8BufferABC[4]为 0x05。

Gu8BufferABC[5]为 0x06。

Gu8BufferABC[6]为 0x07。

d 为 0x01。

e 为 0x0203。

f 为 0x04050607。

#### 【63.4 如何在单片机上练习本章节 C 语言程序?】

直接复制前面章节中第十一节的模板程序,练习代码时只需要更改“C 语言学习区域”的代码就可以了,其它部分的代码不要动。编译后,把程序下载进带串口的 51 学习板,通过电脑端的串口助手软件就可以观察到不同的变量数值,详细方法请看第十一节内容。

## 第六十四节： 指针让函数具备了多个相当于 return 的输出口。

### 【64.1 函数的三类输出渠道。】

函数是模块，模块必须具备输入和输出的接口，从输入和输出的角度分析，函数对外部调用者传递信息主要有三类渠道，第一类是全局变量，第二类是 return 返回值，第三类是用指针。全局变量太隐蔽，没有那么直观，可读性稍差。return 可读性强，缺点是一个函数的 return 只能算一个数据的“出口”，如果一个函数要输出多个结果，return 就力不从心。指针作为函数的输出接口，就能随心所欲了，不但可读性强，而且输出的接口数量不受限制。

### 【64.2 只有一个输出接口的时候。】

现在举一个例子，要用函数实现一个加法运算，输出“一个”加法运算的和，求 3 加上 5 等于 8。下面三个例子中分别使用“全局变量，return, 指针”这三类输出接口。

第一类：全局变量。

```
unsigned char DiaoYongZhe; //调用者
unsigned char BeiJiaShu;   //被加数
unsigned char JiaShu;      //加数
unsigned char He;          //输出的接口，加法运算的“和”。

void JiaFa(void)
{
    He=BeiJiaShu+JiaShu;
}

void main()
{
    BeiJiaShu=3;           //填入被加数 3
    JiaShu=5;              //填入加数 5
    JiaFa();               //调用一次加法运算的函数
    DiaoYongZhe=He;        //把加法运算的“和”赋值给调用者。
}
```

第二类：return。

```
unsigned char DiaoYongZhe; //调用者

unsigned char JiaFa(unsigned char BeiJiaShu,unsigned char JiaShu)
{
    unsigned char He;
    He=BeiJiaShu+JiaShu;
    return He;
}
```

```
void main()
{
    DiaoyongZhe=JiaFa(3,5);    //把加法运算的“和”赋值给调用者，一气呵成。
}
```

第三类：指针。

```
unsigned char DiaoyongZhe;    //调用者

void JiaFa(unsigned char BeiJiaShu,unsigned char JiaShu,unsigned char *pu8He)
{
    *pu8He=BeiJiaShu+JiaShu;
}

void main()
{
    JiaFa(3,5,&DiaoyongZhe);    //通过指针这个输出渠道，把加法运算的“和”赋值给调用者，一气呵成。
}
```

### 【64.3 有多个输出接口的时候。】

现在举一个例子，要用函数实现一个除法运算，分别输出除法运算的商和余数这两个数，求 5 除以 3 等于 1 余 2。因为 return 只能输出一个结果，所以这里不列举 return 的例子，只使用“全局变量”和“指针”这两类输出接口。

第一类：全局变量。

```
unsigned char DiaoyongZhe_Shang; //调用者的商
unsigned char DiaoyongZhe_Yu;    //调用者的余数

unsigned char BeiChuShu;         //被除数
unsigned char ChuShu;           //除数
unsigned char Shang;            //输出的接口，除法运算的“商”。
unsigned char Yu;               //输出的接口，除法运算的“余”。

void ChuFa(void)
{
    Shang=BeiChuShu/ChuShu;      //求商。假设除数不会为 0 的情况。
    Yu=BeiChuShu%ChuShu;        //求余数。假设除数不会为 0 的情况。
}

void main()
{
    BeiChuShu=5;                //填入被除数 5
    ChuShu=3;                   //填入除数 3
}
```

```

ChuFa(); //调用一次除法运算的函数
DiaoYongZhe_Shang=Shang; //把除法运算的“商”赋值给调用者的商。
DiaoYongZhe_Yu=Yu; //把除法运算的“余数”赋值给调用者的余数。
}

```

第二类：return。

return 只能输出一个结果，力不从心，所以这里不列举 return 的例子。

第三类：指针。

```

unsigned char DiaoYongZhe_Shang; //调用者的商
unsigned char DiaoYongZhe_Yu; //调用者的余数

void ChuFa(unsigned char BeiChuShu,
            unsigned char ChuShu,
            unsigned char *pu8Shang,
            unsigned char *pu8Yu)
{
    *pu8Shang=BeiChuShu/ChuShu; //求商。假设除数不会为 0 的情况。
    *pu8Yu=BeiChuShu%ChuShu; //求余数。假设除数不会为 0 的情况。
}

void main()
{
    ChuFa(5, 3, &DiaoYongZhe_Shang, &DiaoYongZhe_Yu); //通过两个指针的输出接口，一气呵成。
}

```

## 【64.4 例程练习和分析。】

现在编一个练习程序。

```

/*---C 语言学习区域的开始。-----*/
void ChuFa(unsigned char BeiChuShu,
            unsigned char ChuShu,
            unsigned char *pu8Shang,
            unsigned char *pu8Yu); //函数声明

unsigned char DiaoYongZhe_Shang; //调用者的商
unsigned char DiaoYongZhe_Yu; //调用者的余数

void ChuFa(unsigned char BeiChuShu,
            unsigned char ChuShu,
            unsigned char *pu8Shang,
            unsigned char *pu8Yu) //函数定义
{

```



```

        *pu8Shang=BeiChuShu/ChuShu;    //求商。假设除数不会为 0 的情况。
        *pu8Yu=BeiChuShu%ChuShu;      //求余数。假设除数不会为 0 的情况。
    }

void main() //主函数
{
    ChuFa(5, 3, &DiaoYongZhe_Shang, &DiaoYongZhe_Yu); //函数调用。通过两个指针的输出接口，一气呵成。

    View(DiaoYongZhe_Shang); //把第 1 个数 DiaoYongZhe_Shang 发送到电脑端的串口助手软件上观察。
    View(DiaoYongZhe_Yu);    //把第 2 个数 DiaoYongZhe_Yu 发送到电脑端的串口助手软件上观察。
    while(1)
    {
    }
}
/*---C 语言学习区域的结束。-----*/

```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第 1 个数

十进制:1

十六进制:1

二进制:1

第 2 个数

十进制:2

十六进制:2

二进制:10

分析：

DiaoYongZhe\_Shang 为 1。

DiaoYongZhe\_Yu 为 2。

## 【64.5 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。

## 第六十五节： 指针作为数组在函数中的入口作用。

### 【65.1 函数的参数入口。】

要往函数内部传递信息，主要有两类渠道。第一类是全局变量。第二类是函数的参数入口，而参数入口可以分为“普通局部变量”和“指针”这两类。“普通局部变量”的参数入口一次只能传一个数据，如果一个数组有几十个甚至上百个数据，此时“普通局部变量”就无能为力，这时不可能也写几十个甚至上百个入口参数吧（这会累坏程序员），针对这种需要输入批量数据的场合，“指针”的参数入口就因此而生，完美解决了此问题，仅用一个“指针”参数入口就能解决一个数组N个数据的入口问题。那么，什么是函数的参数入口？例子如下：

```
//函数声明
unsigned long PingJunZhi(unsigned char a,unsigned char b,unsigned char c,unsigned char d);

//变量定义
unsigned char Gu8Buffer[4]={2,6,8,4}; //4个变量分别是 2, 6, 8, 4。
unsigned long Gu32PingJunZhi; //求平均值的结果

//函数定义
unsigned long PingJunZhi(unsigned char a,unsigned char b,unsigned char c,unsigned char d)
{
    unsigned long u32PingJunZhi;
    u32PingJunZhi=(a+b+c+d)/4;
    return u32PingJunZhi;
}

void main() //主函数
{
    //函数调用
    Gu32PingJunZhi=PingJunZhi(Gu8Buffer[0],Gu8Buffer[1],Gu8Buffer[2],Gu8Buffer[3]);
}
```

上面是一个求4个数据平均值的函数，在这个函数中，函数小括号的(unsigned char a,unsigned char b,unsigned char c,unsigned char d)就是4个变量的“普通局部变量”参数入口，刚才说到，如果一个数组有上百个变量，这种书写方式是很累的。如果改用“指针”入口参数的方式，例子如下：

```
//函数声明
unsigned long PingJunZhi(unsigned char *pu8Buffer);

//变量定义
unsigned char Gu8Buffer[4]={2,6,8,4}; //4个变量分别是 2, 6, 8, 4。
unsigned long Gu32PingJunZhi; //求平均值的结果
```

```

//函数定义
unsigned long PingJunZhi(unsigned char *pu8Buffer)
{
    unsigned long u32PingJunZhi;
    u32PingJunZhi=(pu8Buffer[0]+pu8Buffer[1]+pu8Buffer[2]+pu8Buffer[3])/4;
    return u32PingJunZhi;
}

void main() //主函数
{
    //函数调用
    Gu32PingJunZhi=PingJunZhi(&Gu8Buffer[0]); //等效于 Gu32PingJunZhi=PingJunZhi(Gu8Buffer)
}

```

上面例子中，仅用一个（unsigned char \*pu8Buffer）指针入口参数，就可以达到输入 4 个变量的目的（这 4 个变量要求是同在一个数组内）。

## 【65.2 const 在指针参数“入口”中的作用。】

指针在函数的参数入口中，既可以做“入口”，也可以做“出口”，而 C 语言为了区分这两种情况，提供了 const 这个关键字来限定权限。如果指针加了 const 前缀，就为指针的权限加了紧箍咒，限定了此指针只能作为“入口”，而不能作为“出口”。如果没有加了 const 前缀，就像本节的函数例子，此时指针参数既可以作为“入口”，也可以作为“出口”。加 const 关键字有两个意义，一方面是方便阅读，通过 const 就知道此接口的“入口”和“出口”属性，另一方面，是为了代码的安全，对于只能作为“入口”的指针参数一旦加了 const 限定，万一我们不小心在函数内部对 const 限定的指针所关联的数据进行了更改（“更改”就意味着“出口”），C 编译器在编译的时候就会有提醒或者报错，及时让我们发现程序的 bug（程序的漏洞）。这部分的内容后续章节会讲到，大家先有个大概的了解，本节暂时不深入讲。

## 【65.3 例程练习和分析。】

现在编一个练习程序。

```

/*---C 语言学习区域的开始。-----*/

//函数声明
unsigned long PingJunZhi(unsigned char *pu8Buffer);

//变量定义
unsigned char Gu8Buffer[4]={2,6,8,4}; //4 个变量分别是 2, 6, 8, 4。
unsigned long Gu32PingJunZhi; //求平均值的结果

//函数定义
unsigned long PingJunZhi(unsigned char *pu8Buffer)

```

```

    {
        unsigned long u32PingJunZhi;
        u32PingJunZhi=(pu8Buffer[0]+pu8Buffer[1]+pu8Buffer[2]+pu8Buffer[3])/4;
        return u32PingJunZhi;
    }

void main() //主函数
{
    //函数调用
    Gu32PingJunZhi=PingJunZhi(&Gu8Buffer[0]); //等效于 Gu32PingJunZhi=PingJunZhi(Gu8Buffer)
    View(Gu32PingJunZhi); //把第 1 个数 Gu32PingJunZhi 发送到电脑端的串口助手软件上观察。
    while(1)
    {
    }
}

/*---C 语言学习区域的结束。-----*/

```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第 1 个数  
 十进制:5  
 十六进制:5  
 二进制:101

分析：

平均值变量 Gu32PingJunZhi 为 5。

#### 【65.4 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。

## 第六十六节： 指针作为数组在函数中的出口作用。

### 【66.1 指针作为数组在函数中的出口。】

函数对外部调用者传递信息主要有三类渠道，第一类是全局变量，第二类是 return 返回值，第三类是指针。之前讲指针对外传递信息的时候，只讲了单个变量的情况，现在重点讲讲数组的情况。要把一个四位数的个，十，百，千位分别提取出来成为 4 个数，依次存放在一个包含 4 个字节的数组里，代码如下：

```
void TiQu(unsigned int ul6Data,unsigned char *pu8Buffer) // “提取” 函数
{
    unsigned char u8Ge; //个位
    unsigned char u8Shi; //十位
    unsigned char u8Bai; //百位
    unsigned char u8Qian; //千位

    u8Ge=ul6Data/1%10;        //提取个位
    u8Shi=ul6Data/10%10;      //提取十位
    u8Bai=ul6Data/100%10;     //提取百位
    u8Qian=ul6Data/1000%10;   //提取千位

    //最后，把所提取的数分别传输到“指针”这个“出口通道”
    pu8Buffer[0]=u8Ge;
    pu8Buffer[1]=u8Shi;
    pu8Buffer[2]=u8Bai;
    pu8Buffer[3]=u8Qian;
}
```

上述代码，为了突出“出口通道”，我刻意多增加了 u8Ge、u8Shi、u8Bai、u8Qian 这 4 个局部变量，其实，这 4 个局部变量还可以省略的，此函数简化后的等效代码如下：

```
void TiQu(unsigned int ul6Data,unsigned char *pu8Buffer) // “提取” 函数
{

    pu8Buffer[0]=ul6Data/1%10;        //提取个位
    pu8Buffer[1]=ul6Data/10%10;      //提取十位
    pu8Buffer[2]=ul6Data/100%10;     //提取百位
    pu8Buffer[3]=ul6Data/1000%10;   //提取千位
}
```

### 【66.2 例程练习和分析。】

现在编一个练习程序。

```

/*---C 语言学习区域的开始。-----*/

//函数声明
void TiQu(unsigned int ul6Data,unsigned char *pu8Buffer);

//全局变量定义
unsigned char Gu8Buffer[4]; //存放提取结果的数组

//函数定义
void TiQu(unsigned int ul6Data,unsigned char *pu8Buffer) //“提取”函数
{
    pu8Buffer[0]=ul6Data/1%10;        //提取个位
    pu8Buffer[1]=ul6Data/10%10;       //提取十位
    pu8Buffer[2]=ul6Data/100%10;      //提取百位
    pu8Buffer[3]=ul6Data/1000%10;     //提取千位
}

void main() //主函数
{
    TiQu(9876,&Gu8Buffer[0]); //把 9876 这个四位数分别提取 6、7、8、9 存放在数组 Gu8Buffer 里

    View(Gu8Buffer[0]); //把第 1 个数 Gu8Buffer[0]) 发送到电脑端的串口助手软件上观察
    View(Gu8Buffer[1]); //把第 2 个数 Gu8Buffer[1]) 发送到电脑端的串口助手软件上观察
    View(Gu8Buffer[2]); //把第 3 个数 Gu8Buffer[2]) 发送到电脑端的串口助手软件上观察
    View(Gu8Buffer[3]); //把第 4 个数 Gu8Buffer[3]) 发送到电脑端的串口助手软件上观察

    while(1)
    {
    }
}

/*---C 语言学习区域的结束。-----*/

```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第 1 个数

十进制:6

十六进制:6

二进制:110

第 2 个数

十进制:7

十六进制:7

二进制:111

第 3 个数

十进制:8

十六进制:8

二进制:1000

第 4 个数

十进制:9

十六进制:9

二进制:1001

分析:

Gu8Buffer[0]为 6。

Gu8Buffer[1]为 7。

Gu8Buffer[2]为 8。

Gu8Buffer[3]为 9。

### 【66.3 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。

## 第六十七节： 指针作为数组在函数中既“入口”又“出口”的作用。

### 【67.1 指针作为数组在函数中的“入口”和“出口”。】

前面分别讲了指针的入口和出口，很多初学者误以为指针是一个“单向”的通道，其实，如果指针前面没有加 const 这个“紧箍咒”限定它的属性，指针是“双向”的，不是“单向”的，也就是说，指针是可以同时具备“入口”和“出口”这两种属性的。现在讲一个程序例子，求一个数组（内含 4 元素）的每个元素变量的整数倍的一半，所谓整数倍的一半，就是除以 2，但是不带小数点，比如 4 的整数倍的一半是 2，7 的整数倍的一半是 3（不是 3.5），代码如下：

```
void Half(unsigned char *pu8Buffer) // “求一半”的函数
{
    unsigned char u8Data_0; //临时中间变量
    unsigned char u8Data_1; //临时中间变量
    unsigned char u8Data_2; //临时中间变量
    unsigned char u8Data_3; //临时中间变量

    //从指针这个“入口”里获取需要“被除以 2”的数据。
    u8Data_0=pu8Buffer[0];
    u8Data_1=pu8Buffer[1];
    u8Data_2=pu8Buffer[2];
    u8Data_3=pu8Buffer[3];

    //求数据的整数倍的一半的算法
    u8Data_0=u8Data_0/2;
    u8Data_1=u8Data_1/2;
    u8Data_2=u8Data_2/2;
    u8Data_3=u8Data_3/2;

    //最后，把计算所得的结果分别传输到指针这个“出口”
    pu8Buffer[0]=u8Data_0;
    pu8Buffer[1]=u8Data_1;
    pu8Buffer[2]=u8Data_2;
    pu8Buffer[3]=u8Data_3;
}
```

上述代码，为了突出“入口”和“出口”，我刻意多增加了 u8Data\_0, u8Data\_1, u8Data\_2, u8Data\_3 这 4 个临时中间变量，其实，这 4 个临时中间变量还可以省略的，此函数简化后的等效代码如下：

```
void Half(unsigned char *pu8Buffer) // “求一半”的函数
{
    pu8Buffer[0]=pu8Buffer[0]/2;
    pu8Buffer[1]=pu8Buffer[1]/2;
```



```

    pu8Buffer[2]=pu8Buffer[2]/2;
    pu8Buffer[3]=pu8Buffer[3]/2;
}

```

## 【67.2 例程练习和分析。】

现在编一个练习程序。

```

/*---C 语言学习区域的开始。-----*/

//函数声明
void Half(unsigned char *pu8Buffer);

//全局变量定义
unsigned char Gu8Buffer[4]={4, 7, 16, 25}; //需要“被除以 2”的数组

//函数定义
void Half(unsigned char *pu8Buffer) //“求一半”的函数
{
    pu8Buffer[0]=pu8Buffer[0]/2;
    pu8Buffer[1]=pu8Buffer[1]/2;
    pu8Buffer[2]=pu8Buffer[2]/2;
    pu8Buffer[3]=pu8Buffer[3]/2;
}

void main() //主函数
{
    Half(&Gu8Buffer[0]); //计算数组的整数倍的一半。这里的“入口”和“出口”是“同一个通道”。

    View(Gu8Buffer[0]); //把第 1 个数 Gu8Buffer[0] 发送到电脑端的串口助手软件上观察
    View(Gu8Buffer[1]); //把第 2 个数 Gu8Buffer[1] 发送到电脑端的串口助手软件上观察
    View(Gu8Buffer[2]); //把第 3 个数 Gu8Buffer[2] 发送到电脑端的串口助手软件上观察
    View(Gu8Buffer[3]); //把第 4 个数 Gu8Buffer[3] 发送到电脑端的串口助手软件上观察

    while(1)
    {
    }
}

/*---C 语言学习区域的结束。-----*/

```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

```
第 1 个数
十进制:2
十六进制:2
二进制:10

第 2 个数
十进制:3
十六进制:3
二进制:11

第 3 个数
十进制:8
十六进制:8
二进制:1000

第 4 个数
十进制:12
十六进制:C
二进制:1100
```

分析：

Gu8Buffer[0]为 2。  
Gu8Buffer[1]为 3。  
Gu8Buffer[2]为 8。  
Gu8Buffer[3]为 12。

### 【67.3 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。

## 第六十八节： 为函数接口指针“定向”的 const 关键词。

### 【68.1 为函数接口指针“定向”的 const 关键词。】

在函数接口处的指针，是一个双向口，既可以作为“输入”也可以作为“输出”，换句话说，既能“读”也能“写”（被更改），这样一来，当你把一个数组（或者某变量）通过指针引入到函数内部的时候，当执行完此函数，这个数组的数值可能已经悄悄发生了更改（“是否被更改”取决于函数内部的具体代码），进来时是“摩托”出来后可能已变成“单车”，而实际项目上，很多时候我们只想传递数组（或者某变量）的数值，并不想数组（或者某变量）本身发生变化，这个时候，本节的主角 const 关键词就派上用场了。

只要在函数接口的指针前面加上 const 关键词，原来双向的指针就立刻变成了单向，只能输入不能输出。这个 const 有两个好处。第一个好处是方便阅读，通过 const 就知道此接口的“入口”和“出口”属性，如果你是用别人已经封装好的函数，一旦发现接口指针带了 const 标签，就足以说明这个指针只能作为输入接口，不用担心输入数据被意外修改。第二个好处是确保数据的安全，函数接口指针一旦加了 const 限定，万一你不小心在函数内部对指针所关联的数据进行了更改（“更改”就意味着“出口”），C 编译器在编译的时候就会报错让你编译失败，及时让你发现程序的 bug（程序的漏洞），这是编译器层面的一道防火墙。例子如下：

```
unsigned char ShuRu(const unsigned char *pu8Data)
{
    unsigned char a;
    a=*pu8Data; //这行代码是合法的，是指针所关联数据的“读”操作。
    *pu8Data=a; //这行代码是非法的，是指针所关联数据的“写”操作，违背 const 的约束。
    return a;
}
```

### 【68.2 例程练习和分析。】

在前面第 65 节讲函数入口的时候，用到一个求数组平均值的程序例子，这个数组是仅仅作为输入用的，不需要被更改，因此，现在借本节讲 const 的机会，为此函数的接口指针补上一个 const 关键词，让该函数更加科学规范，程序如下：

```
/*---C 语言学习区域的开始。-----*/

unsigned long PinJunZhi(const unsigned char *pu8Buffer); //指针前增加一个 const 关键词
unsigned char Gu8Buffer[4]={2,6,8,4};
unsigned long Gu32PinJunZhi;

unsigned long PinJunZhi(const unsigned char *pu8Buffer) //指针前增加一个 const 关键词
{
    unsigned long u32PinJunZhi;
    u32PinJunZhi=(pu8Buffer[0]+pu8Buffer[1]+pu8Buffer[2]+pu8Buffer[3])/4; //求平均值
    return u32PinJunZhi;
}

void main() //主函数
{
```

```

    Gu32PinJunZhi=PinJunZhi (&Gu8Buffer[0]); //不用担心 Gu8Buffer 数组的数据被意外更改。

    View(Gu32PinJunZhi); //把第 1 个数 Gu32PinJunZhi 发送到电脑端的串口助手软件上观察。
    while(1)
    {
    }
}
/*---C 语言学习区域的结束。-----*/

```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第 1 个数  
十进制:5  
十六进制:5  
二进制:101

分析：

平均值变量 Gu32PinJunZhi 为 5。

### 【68.3 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。

## 第六十九节： 宏函数 sizeof()。

### 【69.1 宏函数 sizeof() 的基础知识。】

宏函数 sizeof() 是用来获取某个对象所占用的字节数。既然是“宏”，就说明它不是单片机执行的函数，而是单片机之外的 C 编译器执行的函数（像#define 这类宏语句一样），也就是说，在单片机上电之前，C 编译器在电脑端翻译我们的 C 语言程序的时候，一旦发现了这个宏函数 sizeof，它就会在电脑端根据 C 语言程序的一些关键字符（比如“unsigned char, [,]”这类字符）来自动计算这个对象所占用的字节数，然后再把我们 C 语言程序里所有的 sizeof 字符替换等效成一个“常量数字”，1 代表 1 个字节，5 代表 5 个字节，1000 代表 1000 个字节。所谓在单片机之外执行的宏函数，就是说，在“计算”这些对象所占的字节数的时候，这个“计算”的工作只占用电脑的内存（C 编译器是在电脑上运行的），并不占用单片机的 ROM 容量和内存。而其它在单片机端执行的“非宏”函数，是占用单片机的 ROM 容量和内存。比如：

```
unsigned char a;           //变量。占用 1 个字节
unsigned int b;            //变量。占用 2 个字节
unsigned long c;          //变量。占用 4 个字节
code unsigned char d[9];   //常量。占用 9 个字节

unsigned int Gul6GetBytes; //这个变量用来获取字节数

Gul6GetBytes=sizeof(a);    //单片机上电后，在单片机程序里等效于 Gul6GetBytes=1;
Gul6GetBytes=sizeof(b);    //单片机上电后，在单片机程序里等效于 Gul6GetBytes=2;
Gul6GetBytes=sizeof(c);    //单片机上电后，在单片机程序里等效于 Gul6GetBytes=4;
Gul6GetBytes=sizeof(d);    //单片机上电后，在单片机程序里等效于 Gul6GetBytes=9;
```

上述的“sizeof 字符”在进入单片机的层面的时候，已经被编译器预先替换成对应的“常量数字”的，这个“常量数字”就代表所占用的字节数。

### 【69.2 宏函数 sizeof() 的作用。】

在项目中，通常用在两个方面：一方面是用在求一个数组的大小尺寸，另一方面是用在计算内存分配时候的偏移量。当然，sizeof 并不是“刚需”，如果没有 sizeof 宏函数，我们也可以人工计算出一个对象所占用的字节数，只是，人工计算，一方面容易出错，另一方面代码往往“动一发而牵全身”，改一个变量往往就会涉及很多地方需要配合调整更改，没法做到“自由裁剪”的境界。下面举一个程序例子：要把 3 个不同长度的数组“合并”成 1 个数组。

第一种情况：在没有使用 sizeof 宏函数时，人工计算字节数和偏移量：

```
unsigned char a[2]={1,2}; //占用 2 个字节
unsigned char b[3]={3,4,5}; //占用 3 个字节
unsigned char c[4]={6,7,8,9}; //占用 4 个字节
unsigned char HeBing[9]; //合并 a,b,c 在一起的数组。这里的 9 是人工计算 a,b,c 容量累加所得。
unsigned char i; //循环变量 i

for(i=0;i<2;i++) //这里的 2，是人工计算出 a 占用 2 个字节
```

```

{
    HeBing[i+0]=a[i]; //从 HeBing 数组的偏移量第 0 个地址开始存放。
}

for(i=0;i<3;i++) //这里的 3，是人工计算出 b 占用 3 个字节
{
    HeBing[i+2]=b[i]; //这里的 2 是人工计算出的偏移量。a 占用了数组 2 个字节。
}

for(i=0;i<4;i++) //这里的 4，是人工计算出 c 占用 4 个字节
{
    HeBing[i+2+3]=c[i]; //这里的 2 和 3 是人工计算出的偏移量，a 和 b 占用了数组 2+3 个字节。
}

```

第二种情况：在使用 sizeof 宏函数时，利用 C 编译器自动来计算字节数和偏移量：

```

unsigned char a[2]={1,2}; //占用 2 个字节
unsigned char b[3]={3,4,5}; //占用 3 个字节
unsigned char c[4]={6,7,8,9}; //占用 4 个字节
unsigned char HeBing[sizeof(a)+sizeof(b)+sizeof(c)]; //C 编译器自动计算字节数
unsigned char i;

for(i=0;i<sizeof(a);i++) //C 编译器自动计算字节数
{
    HeBing[i+0]=a[i];
}

for(i=0;i<sizeof(b);i++) //C 编译器自动计算字节数
{
    HeBing[i+sizeof(a)]=b[i]; //C 编译器自动计算偏移量
}

for(i=0;i<sizeof(c);i++) //C 编译器自动计算字节数
{
    HeBing[i+sizeof(a)+sizeof(b)]=c[i]; //C 编译器自动计算偏移量
}

```

### 【69.3 例程练习和分析。】

现在编写一个练习的程序：

```

/*---C 语言学习区域的开始。-----*/

unsigned char a[2]={1,2}; //占用 2 个字节

```

```

unsigned char b[3]={3,4,5}; //占用 3 个字节
unsigned char c[4]={6,7,8,9}; //占用 4 个字节
unsigned char  HeBing[sizeof(a)+sizeof(b)+sizeof(c)];//C 编译器自动计算字节数
unsigned char i;

void main() //主函数
{
    for(i=0;i<sizeof(a);i++)  //C 编译器自动计算字节数
    {
        HeBing[i+0]=a[i];
    }

    for(i=0;i<sizeof(b);i++)  //C 编译器自动计算字节数
    {
        HeBing[i+sizeof(a)]=b[i];  //C 编译器自动计算偏移量
    }

    for(i=0;i<sizeof(c);i++)  //C 编译器自动计算字节数
    {
        HeBing[i+sizeof(a)+sizeof(b)]=c[i]; //C 编译器自动计算偏移量
    }

    for(i=0;i<sizeof(HeBing);i++)  //利用宏 sizeof 计算出 HeBing 数组所占用的字节数
    {
        View(HeBing[i]);  //把 HeBing 所有数据挨个依次全部发送到电脑端观察
    }

    while(1)
    {
    }
}

/*---C 语言学习区域的结束。-----*/

```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第 1 个数

十进制:1

十六进制:1

二进制:1

第 2 个数

十进制:2

十六进制:2

二进制:10

第 3 个数

十进制:3

十六进制:3

二进制:11

第 4 个数

十进制:4

十六进制:4

二进制:100

第 5 个数

十进制:5

十六进制:5

二进制:101

第 6 个数

十进制:6

十六进制:6

二进制:110

第 7 个数

十进制:7

十六进制:7

二进制:111

第 8 个数

十进制:8

十六进制:8

二进制:1000

第 9 个数

十进制:9

十六进制:9

二进制:1001

分析:

HeBing[0]为 1。

HeBing[1]为 2。

HeBing[2]为 3。

HeBing[3]为 4。

HeBing[4]为 5。



HeBing[5]为 6。

HeBing[6]为 7。

HeBing[7]为 8。

HeBing[8]为 9。

#### 【69.4 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。

## 第七十节：“万能数组”的结构体。

### 【70.1 结构体与数组。】

结构体是数组，但不是普通的数组，而是一种“万能数组”。普通数组，是依靠严格的数组下标（类似编号）来识别某个具体单元的（或者称“寻址”），期间，如果要往数组插入或者删除某些单元，后面所有单元的下标编号都会发生改变，牵一发而动全身，后面其它单元的下标序号自动重新排列，原来某个特定的单元的下标发生了改变，也就意味着“名字”发生了改变，这种情况在编写程序的时候，就意味着很多代码需要随着更改调整，给程序员带来很多不便。怎么办？结构体此时横空出世，扭转了这种“不便”的局面。之所以称结构体为“万能数组”，是因为结构体内部没有“下标编号”，只有名字。结构体与普通数组的本质区别是，结构体是靠“名字”来寻址的，不管你往结构体里插入或者删除某些单元，其它单元的“名字”不会发生改变，隔离效果好，左邻右舍不会受影响。除此之外，结构体内部的成员变量是允许出现不同的数据类型的，比如 unsigned char, unsigned int, unsigned long 这三种数据类型的变量都可以往同一个结构体里面“填充”，不受类型的局限，真正做到“万能”级。而普通数组就没有这个优越性，普通数组要么清一色都是 unsigned char，要么清一色都是 unsigned int，要么清一色都是 unsigned long，不能像结构体这么“混合型”的。结构体的这种优越性，在大型程序的升级和维护时体现得非常明显。

### 【70.2 “造模”和“生成”和“调用”。】

结构体的使用，有三道标准工序“造模”和“生成”和“调用”。塑胶外壳，必须先开模具（造模），然后再用模具印出外壳（生成），再把外壳应用于日常生活中（调用）。结构体也一样，先“造”结构体的“模”（造模），再根据这个“模”来“生成”一个结构体变量（生成），然后在某函数里使用此变量（调用）。例子如下：

```
struct StructMould    // “造模”
{
    unsigned char  u8Data_A;
    unsigned int   u16Data_B;
    unsigned long   u32Data_C;
};

struct StructMould  GtMould; // “生成”一个变量 GtMould。

void main()
{
    GtMould.u8Data_A=1;      //依靠成员的“名字”来“调用”
    GtMould.u16Data_B=2;     //依靠成员的“名字”来“调用”
    GtMould.u32Data_C=3;     //依靠成员的“名字”来“调用”

    while(1)
    {

    }

}
```

把上述程序转换成“普通数组”和“指针”的形式，给大家一个直观的对比，代码如下：

```
unsigned char Gu8MouldBuffer[7]; //相当于结构体变量 GtMould

unsigned char *pu8Data_A;
unsigned int *pu16Data_B;
unsigned long *pu32Data_C;

void main()
{
    pu8Data_A=(unsigned char *)&Gu8MouldBuffer[0]; //依靠数组的下标[0]来“调用”
    *pu8Data_A=1;

    pu16Data_B=(unsigned int *)&Gu8MouldBuffer[1]; //依靠数组的下标[1]来“调用”
    *pu16Data_B=2;

    pu32Data_C=(unsigned long *)&Gu8MouldBuffer[3]; //依靠数组的下标[3]来“调用”
    *pu32Data_C=3;

    while(1)
    {

    }
}
```

分析：上述两种代码，目标都是把 1, 2, 3 这三个数字存放在一个数组里。第一种用结构体的方式，第二种用普通数组的方式。

### 【70.3 例程练习和分析。】

现在编写一个练习的程序：

```
/*---C 语言学习区域的开始。-----*/

struct StructMould    // “造模”
{
    unsigned char  u8Data_A;
    unsigned int   u16Data_B;
    unsigned long  u32Data_C;
};

struct StructMould  GtMould; // “生成” 一个变量 GtMould。
```

```

void main() //主函数
{
    GtMould.u8Data_A=1;      //依靠成员的“名字”来“调用”
    GtMould.u16Data_B=2;    //依靠成员的“名字”来“调用”
    GtMould.u32Data_C=3;    //依靠成员的“名字”来“调用”

    View(GtMould.u8Data_A);  //把结构体成员 GtMould.u8Data_A 发送到电脑端观察
    View(GtMould.u16Data_B); //把结构体成员 GtMould.u16Data_B 发送到电脑端观察
    View(GtMould.u32Data_C); //把结构体成员 GtMould.u32Data_C 发送到电脑端观察

    while(1)
    {
    }
}
/*---C 语言学习区域的结束。-----*/

```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第 1 个数

十进制:1

十六进制:1

二进制:1

第 2 个数

十进制:2

十六进制:2

二进制:10

第 3 个数

十进制:3

十六进制:3

二进制:11

分析：

GtMould.u8Data\_A 为 1。

GtMould.u16Data\_B 为 2。

GtMould.u32Data\_C 为 3。

#### 【70.4 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，

其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。

## 第七十一节： 结构体的内存和赋值。

### 【71.1 结构体的内存生效。】

上一节讲到结构体有三道标准工序“造模”和“生成”和“调用”，那么，结构体在哪道工序的时候才会开始占用内存(或者说内存生效)？答案是在第二道工序“生成”（或者说定义）的时候才产生内存开销。第一道工序仅“造模”不“生成”是不会产生内存的。什么意思呢？请看下面的例子。

第一种情况：仅“造模”不“生成”。

```
struct StructMould    // “造模”
{
    unsigned char  u8Data_A;
    unsigned char  u8Data_B;
};
```

分析：这种情况是没有内存开销的，尽管你已经写下了数行代码，但是 C 编译器在翻译此代码的时候，它会识别到你偷工减料仅仅“造模”而不“生成”新变量，此时 C 编译器会把你这段代码忽略而过。

第二种情况：先“造模”再“生成”。

```
struct StructMould    // “造模”
{
    unsigned char  u8Data_A;
    unsigned char  u8Data_B;
};

struct StructMould  GtMould_1; // “生成”一个变量 GtMould_1。占用 2 个字节内存
struct StructMould  GtMould_2; // “生成”一个变量 GtMould_2。占用 2 个字节内存
```

分析：这种情况才会占用内存。你“生成”变量越多，占用的内存就越大。像本例子，“生成”了两个变量 GtMould\_1 和 GtMould\_2，一个变量占用 2 个字节，两个就一共占用了 4 个字节。结论：内存的占用是跟变量的“生成”有关。

### 【71.2 结构体的内存对齐。】

什么是对齐？为了确保内存的地址能整除某个“对齐倍数”（比如 4）。比如以 4 为“对齐倍数”，在地址 0 存放一个变量 a, 因为地址 0 能整除“对齐倍数”4，所以符合“地址对齐”，接着往下再存放第二个变量 b，紧接着的地址 1 不能整除“对齐倍数”4，此时，为了内存对齐，本来打算把变量 b 放到地址 1 的，现在就要更改挪到地址 4 才符合“地址对齐”，这就是内存对齐的含义。“对齐倍数”是什么？“对齐倍数”就是单片机的位数除以 8。比如 8 位单片机的“对齐倍数”是 1（8 除以 8），16 位单片机是 2（16 除以 8），32 位单片机是 4（32 除以 8）。本教程所用的单片机是 8 位的 51 内核单片机，因此“对齐倍数”是 1。1 是可以被任何整数整除的，因此，8 位单片机在结构体的使用上被内存对齐的“干扰”是最小的。

为什么要对齐？单片机内部硬件层面一条指令处理的数据宽度是固定的，比如，因为一个字节是 8 位，所以，8 位的单片机一次处理的数据宽度是 1 个字节（8 除以 8 等于 1），16 位的单片机一次处理的数据宽度是 2 个字节（16 位除以 8 位等于 2），32 位的单片机一次处理的数据宽度是 4 个字节（32 位除以 8 位等于 4），如果字节不对齐，本来单片机一个指令能处理的数据可能就要分解成 2 个指令甚至更多的指令，所以 C 编译器为了让单片机处于最佳状态，在某些情况就会涉及内存对齐，结构体就涉及到内存对齐。

结构体的内存对齐表现在哪里呢？请看下面两个例子：

第一个例子：8 位单片机。

```
struct StructMould_1    // “造模”
{
    unsigned char  u8Data;    //一个 unsigned char 占用 1 个字节。
    unsigned long  u32Data;    //一个 unsigned long 占用 4 个字节。
};

struct StructMould_1  GtMould_1;  //占用多少个字节内存呢？
```

分析：GtMould\_1 这个变量占用多少个内存字节呢？假设 GtMould\_1 的首地址是 0，那么地址 0 就存放成员 u8Data，u8Data 占用 1 个字节，所以接下来的地址是 1（0+1），问题来了，地址 1 能直接存放占用 4 个字节的成员 u32Data 吗？因为 8 位单片机的“对齐倍数”是 1（8 除以 8），那么地址 1 显然是可以整除“对齐倍数”1 的，因此，地址 1 是可以果断存储 u32Data 成员的。因此，GtMould\_1 占用的总字节数是 5（1+4），也就是 u8Data 和 u32Data 两者所占字节数之和。

第二个例子：32 位单片机。

```
struct StructMould_1    // “造模”
{
    unsigned char  u8Data;    //一个 unsigned char 占用 1 个字节。
    unsigned long  u32Data;    //一个 unsigned long 占用 4 个字节。
};

struct StructMould_1  GtMould_1;  //占用多少个字节内存呢？
```

分析：GtMould\_1 这个变量占用多少个内存字节呢？假设 GtMould\_1 的首地址是 0，那么地址 0 就存放成员 u8Data，u8Data 占用 1 个字节，所以接下来的地址是 1（0+1），那么问题来了，地址 1 能直接存放占用 4 个字节的成员 u32Data 吗？不能。因为 32 位单片机的“对齐倍数”是 4（32 除以 8），那么地址 1 显然是不可以整除“对齐倍数”4 的，因此，就要把地址 1 更改挪到地址 4 这里才符合“地址对齐”，这样，就意味着多插入了 3 个“填充的字节”，因此，GtMould\_1 占用的总字节数是 8（1+3+4），也就是“1 个字节 u8Data，3 个填充字节，4 个 u32Data”三者所占字节数之和。那么问题又来了，如果把结构体内部成员 u8Data 和 u32Data 的位置顺序更改一下，内存容量会有所改变吗？位置顺序更改后如下。

```
struct StructMould_1    // “造模”
```

```

{
    unsigned long  u32Data;    //一个 unsigned long 占用 4 个字节。
    unsigned char  u8Data;     //一个 unsigned char 占用 1 个字节。
};

struct StructMould_1  GtMould_1; //占用多少个字节内存呢？

```

分析：更改 u8Data 和 u32Data 的位置顺序后，u32Data 在前 u8Data 在后，GtMould\_1 这个变量占用多少个内存字节呢？假设 GtMould\_1 的首地址是 0，那么地址 0 就存放成员 u32Data，u32Data 占用 4 个字节，所以接下来的地址是 4 (0+4)，那么问题来了，地址 4 能直接存放占用 1 个字节的成员 u8Data 吗？能。因为 32 位单片机的“对齐倍数”是 4 (32 除以 8)，那么地址 4 显然是可以整除“对齐倍数”4 的，因此，地址 4 是可以果断存储 u8Data 的。那么，是不是 GtMould\_1 就占用 5 个字节呢？不是。因为结构体的内存对齐，还包括另外一条规定，那就是“一个结构体变量所占的内存总容量必须能整除该单片机的“对齐倍数”（单片机的位数除以 8），如果不能，C 编译器就会擅自在最后一个成员的后面插入若干个“填充字节”来满足这个规则”，根据这条规定，计算所得的总容量 5 是不能整除“对齐倍数”4 的，必须再额外填充 3 个字节补足到 8，才能整除“对齐倍数”4，因此，更改顺序后，GtMould\_1 还是占用 8 个字节 (4+1+3)，前 4 个字节是 u32Data，中间 1 个字节是 u8Data，后 3 个字节是“填充字节”。

因为本教程采用的是 8 位的 51 内核单片机，因此，在上述这个例子中，GtMould\_1 所占的字节数是符合“第一个例子”的情况，也就是占用 5 个字节。内存对齐是遵守几条严格的规则的，我只列出其中最关键的两条给大家大致阅读一下，有一个印象即可，不强求死记硬背，只需知道“结构体因为存在内存对齐，所以实际内存容量是有可能大于内部各成员类型字节数相加之和，尤其是 16 位或者 32 位这类单片机”就可以了。

第（1）条：结构体内部某个成员相对结构体首地址的偏移地址必须能整除该单片机的“对齐倍数”（单片机的位数除以 8），如果不能，C 编译器就会擅自在各成员之间插入若干个“填充字节”来满足这个规则。

第（2）条：一个结构体变量所占的内存总容量必须能整除该单片机的“对齐倍数”（单片机的位数除以 8），如果不能，C 编译器就会擅自在最后一个成员的后面插入若干个“填充字节”来满足这个规则。

### 【71.3 如何获取某个结构体变量的内存容量？】

结构体存在内存对齐的问题，就说明它的内存占用情况不会像普通数组那样一目了然，那么，我们编写程序的时候怎么知道某个结构体变量占用了多少个字节数？答案是：用 sizeof 宏函数。比如：

```

struct StructMould_1
{
    unsigned long  u32Data;
    unsigned char  u8Data;
};

struct StructMould_1  GtMould_1;

unsigned long a; //此变量用来获取结构体变量 GtMould_1 所占用的字节总数
void main() //主函数
{

```



```
a=sizeof(GtMould_1); //利用宏函数 sizeof 获取结构体变量所占用的字节总数
}
```

#### 【71.4 结构体之间的赋值。】

结构体之间的赋值有两种，第一种是成员之间“一对一”的赋值，第二种是整个结构体之间“面对面”的整体赋值。第一种成员赋值像普通变量赋值那样，没有那么多套路和忌讳，数据传递安全可靠。第二种整个结构体之间赋值在编程体验上带有“一键操作”的快感，但是要注意避开一些“雷区”，首先，整体赋值的前提是必须保证两个结构体变量都是同一个“结构体模板”造出来的变量，不同“模板”的结构体变量之间禁止“整体赋值”，其次，哪怕是“同一个模板”的结构体变量，也并不是所有的“同模板结构体”变量都能实现整个结构体之间的直接赋值，只有在结构体内部成员比较简单的情况下才适合“整体赋值”，如果结构体内部包含有“指针”或者“字符串”或者“其它结构体中的结构体”，这类情况就比较复杂，这时建议大家绕开有“雷区”的“整体赋值”而直接选用安全可靠的“成员赋值”。什么是“成员赋值”什么是“整体赋值”？请看下面两个例子。

第一种：成员赋值。把结构体变量 GtMould\_2\_A 赋值给 GtMould\_2\_B。

```
struct StructMould_2    // “造模”
{
    unsigned long  u32Data;
    unsigned char  u8Data;
};

struct StructMould_2  GtMould_2_A; //生成第 1 个结构体变量
struct StructMould_2  GtMould_2_B //生成第 2 个结构体变量

void main() //主函数
{
    //先给 GtMould_2_A 赋初值。
    GtMould_2_A.u32Data=1;
    GtMould_2_A.u8Data=2;

    //通过“成员赋值”，把结构体变量 GtMould_2_A 赋值给 GtMould_2_B。
    GtMould_2_B.u32Data=GtMould_2_A.u32Data; //成员之间“一对一”的赋值
    GtMould_2_B.u8Data=GtMould_2_A.u8Data;   //成员之间“一对一”的赋值
}
```

第二种：整体赋值。把结构体变量 GtMould\_2\_A 赋值给 GtMould\_2\_B。

```
struct StructMould_2    // “造模”
{
    unsigned long  u32Data;
    unsigned char  u8Data;
};
```

```

struct StructMould_2  GtMould_2_A;  //生成第 1 个结构体变量
struct StructMould_2  GtMould_2_B   //生成第 2 个结构体变量

void main() //主函数
{
    //先给 GtMould_2_A 赋初值。
    GtMould_2_A.u32Data=1;
    GtMould_2_A.u8Data=2;

    //通过“整体赋值”，把结构体变量 GtMould_2_A 赋值给 GtMould_2_B。
    GtMould_2_B=GtMould_2_A;    //整体之间“一次性”的赋值
}

```

上述例子中的整体赋值，是因为结构体内部的数据比较“简单”，没有包含“指针”或者“字符串”或者“其它结构体中的结构体”这类数据成员，如果包含这类成员，建议大家不要用整体赋值。比如遇到以下这类结构体就建议大家直接用安全可靠的“成员赋值”：

```

struct StructMould    //“造模”
{
    unsigned char u8String[]=" String" ; //字符串
    unsigned char *pu8Data; //指针
    struct StructOtherMould GtOtherMould; //结构体中的结构体
};

```

## 【71.5 例程练习和分析。】

现在编写一个练习的程序：

```

/*---C 语言学习区域的开始。-----*/

struct StructMould_1    //“造模”
{
    unsigned long  u32Data;    //一个 unsigned long 占用 4 个字节。
    unsigned char  u8Data;     //一个 unsigned char 占用 1 个字节。
};

struct StructMould_2    //“造模”
{
    unsigned char  u8Data;
    unsigned long  u32Data;
};

struct StructMould_1  GtMould_1; //占用多少个字节内存呢？

```

```

    struct StructMould_2  GtMould_2_A;
    struct StructMould_2  GtMould_2_B;

    unsigned long a; //此变量用来获取结构体变量 GtMould_1 所占用的字节总数

void main() //主函数
{
    a=sizeof(GtMould_1); //利用宏函数 sizeof 获取结构体变量 GtMould_1 所占用的字节总数

    //先给 GtMould_2_A 赋初值。
    GtMould_2_A.u32Data=1;
    GtMould_2_A.u8Data=2;

    //通过“整体赋值”，把结构体变量 GtMould_2_A 赋值给 GtMould_2_B。
    GtMould_2_B=GtMould_2_A; //整体之间“一次性”的赋值

    View(a); //把 a 发送到电脑端观察
    View(GtMould_2_B.u32Data); //把结构体成员 GtMould_2_B.u32Data 发送到电脑端观察
    View(GtMould_2_B.u8Data); //把结构体成员 GtMould_2_B.u8Data 发送到电脑端观察

    while(1)
    {
    }
}

/*---C 语言学习区域的结束。-----*/

```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第 1 个数

十进制:5

十六进制:5

二进制:101

第 2 个数

十进制:1

十六进制:1

二进制:1

第 3 个数

十进制:2

十六进制:2

分析:

GtMould\_1 所占的字节数 a 为 5。

GtMould\_2\_B 的结构体成员 GtMould\_2\_B.u32Data 为 1。

GtMould\_2\_B 的结构体成员 GtMould\_2\_B.u8Data 为 2。

## 【71.6 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。

## 第七十二节： 结构体的指针。

### 【72.1 结构体指针的重要用途。】

结构体指针有两个重要用途，一个是结构体数据的传输存储和还原，另一个是作为结构体数据在涉及函数时的参数入口。

什么是“结构体数据的传输存储和还原”？结构体本质是一个数组，数组内可能包含了许多不同数据长度类型的成员，当整个结构体数据需要存储或者传输（通信）给另外一个单片机时，这时候有两种选择，一种是一个成员一个成员的挨个处理，这种“以成员为单位”的处理方式比较繁琐麻烦，另外一种是把整个结构体变量当作一个“以字节为单位”的普通数组来处理，但是关键的问题来了，假如我们把整个结构体数据以“字节为单位”的方式“整体打包”传递给另外一个单片机，当这个接收方的单片机接收到我们这一组数据后，如何把这“一包”以字节为单位的数组“还原”成相同的结构体变量，以便在程序处理中也能直接按“结构体的方式”处理某个具体的成员，这时就涉及到结构体指针的应用。

什么是“作为结构体数据在涉及函数时的参数入口”？结构体数据一般内部包含了很多成员，当要把这一包数据传递给某个函数内部时，这个函数要给结构体数据预留参数入口，这时，如果函数以结构体成员的角度来预留入口，那么有多少个成员就要预留多少个函数的参数入口，可阅读性非常差，操作起来也麻烦。但是，如果以指针的角度来预留入口，那么不管这个结构体内部包含多少个成员，只需要预留一个指针参数入口就够用了，这就是绝大多 32 位单片机的库函数都采样结构体指针作为函数的参数入口的原因。

结构体指针这两个重要用途后续章节会深入讲解，本节的重点是先让大家学会结构体指针的基础知识，为后续章节做准备。

### 【72.2 结构体指针的基础。】

操作结构体内部某个具体变量时，有两种方式，一种是成员调用的方式，另一种是指针调用的方式。C 语言语法为了区分这两种方式，专门设计了两种不同的操作符号。成员调用方式采样小数点“.”的符号，指针调用方式采用箭头“->”的符号。例子如下：

```
struct StructMould_1
{
    unsigned char  u8Data_A;
    unsigned long  u32Data_B;
};

struct StructMould_1  GtMould_1;  // “生成” 一个变量。    // 占用 5 个字节。
struct StructMould_1  *ptMould_1; // 定义一个结构体指针。  // 占用 3 个字节。

void main() // 主函数
{
    GtMould_1.u8Data_A=5;    // “成员调用” 的方式，用小数点符号 “.”

    ptMould_1=&GtMould_1;    // ptMould_1 指针与变量 GtMould_1 建立关联。
    ptMould_1->u8Data_A=ptMould_1->u8Data_A+5; // “指针调用” 的方式，用箭头符号 “->”
```

```

while(1)
{
}
}

```

分析：上述例子中，信息量很大，知识点有两个。

第一个知识点：为什么结构体变量 GtMould\_1 占用 5 个字节，而结构体指针 \*ptMould\_1 只占用 3 个字节？结构体变量 GtMould\_1 所占的内存是由结构体成员内部的数量决定的，而结构体指针 \*ptMould\_1 是由 C 编译器根据芯片硬件寻址范围而决定的，在一个相同的 C 编译器系统中，所有类型的指针所占用的字节数都是一样的，比如在本教程中所用 8 位单片机的 C51 编译器系统中，unsigned char \*, unsigned int \*, unsigned long \*, 以及本节的 struct StructMould\_1 \*, 都是占用 3 个字节。32 位单片机的指针往往都是 4 个字节，而某些 64 位的 PC 机，指针可能是 8 个字节，这些内容大家只要有个大概的了解即可。

第二个知识点：结构体成员 GtMould\_1.u8Data\_A 经过第一步的“成员调用”直接赋值 5，紧接着经过“指针调用”的累加 5 操作，最后 GtMould\_1.u8Data\_A 的数值是 10 (5+5)。

### 【72.3 例程练习和分析。】

现在编写一个练习的程序：

```

/*---C 语言学习区域的开始。-----*/

struct StructMould_1
{
    unsigned char  u8Data_A;
    unsigned long  u32Data_B;
};

struct StructMould_1  GtMould_1;  // “生成” 一个变量。    // 占用 5 个字节。
struct StructMould_1  *ptMould_1;  // 定义一个结构体指针。  // 占用 3 个字节。

void main() //主函数
{
    GtMould_1.u8Data_A=5;    // “成员调用” 的方式，用小数点符号 “.”

    ptMould_1=&GtMould_1;    //ptMould_1 指针与变量 GtMould_1 建立关联。
    ptMould_1->u8Data_A=ptMould_1->u8Data_A+5; // “指针调用” 的方式，用箭头符号 “->”

    View(sizeof(GtMould_1));    //在电脑端观察变量 GtMould_1 占用多少个字节。
    View(sizeof(ptMould_1));    //在电脑端观察指针 ptMould_1 占用多少个字节。
    View(GtMould_1.u8Data_A);    //在电脑端观察结构体成员 GtMould_1.u8Data_A 的最后数值。
    while(1)
    {
    }
}

```

```
}  
/*---C 语言学习区域的结束。-----*/
```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第 1 个数

十进制:5

十六进制:5

二进制:101

第 2 个数

十进制:3

十六进制:3

二进制:11

第 3 个数

十进制:10

十六进制:A

二进制:1010

分析：

变量 GtMould\_1 占用 5 个字节。

指针 ptMould\_1 占用 3 个字节。

结构体成员 GtMould\_1.u8Data\_A 的最后数值是 10。

#### 【72.4 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。

## 第七十三节： 结构体数据的传输存储和还原。

### 【73.1 结构体数据的传输存储和还原。】

结构体本质是一个数组，数组内可能包含了许多不同数据长度类型的成员，当整个结构体数据需要存储或者传输（通信）给另外一个单片机时，这时候有两种选择，一种是一个成员一个成员的挨个处理，这种“以成员为单位”的处理方式比较繁琐麻烦，另外一种是把整个结构体变量当作一个“以字节为单位”的普通数组来处理，但是有两个关键的问题来了，第一个问题是如何把结构体“拆分”成“以字节为单位”来进行搬运数据，第二个问题是假如我们把整个结构体数据以“字节为单位”的方式“整体打包”传递给另外一个单片机，当这个接收方的单片机接收到我们这一组数据后，如何把这“一包”以字节为单位的数组再“还原”成相同的结构体变量，以便在程序处理中也能直接按“结构体的方式”来处理某个具体的成员。其实，这两个问题都涉及到“指针的强制转换”。具体讲解的例子，请直接阅读下面 73.2 段落的源代码例子和注释。

### 【73.2 例程练习和分析。】

现在编写一个练习程序，把一个结构体变量“以字节的方式”存储到另外一个普通数组里，然后再把这个“以字节为单位”的普通数组“还原”成“结构体的方式”，以便直接操作内部某个具体的成员。

```
/*---C 语言学习区域的开始。-----*/

struct StructMould_1
{
    unsigned char  u8Data_A;
    unsigned long  u32Data_B;
    unsigned int   u16Data_C;
};

struct StructMould_1  GtMould_1;  // “生成” 一个变量。

unsigned char  Gu8Buffer[sizeof(GtMould_1)]; //定义一个内存跟结构体变量大小一样的普通数组
unsigned char *pu8;    //定义一个把结构体变量“拆分”成“以字节为单位”的指针
struct StructMould_1  *ptStruct; //定义一个结构体指针，用于“还原”普通数组为“结构体”
unsigned int i;        //定义一个用于 for 循环的变量

void main() //主函数
{
    //先把该结构体变量内部具体成员分别以“成员的方式”初始化为 5, 6, 7
    GtMould_1.u8Data_A=5;
    GtMould_1.u32Data_B=6;
    GtMould_1.u16Data_C=7;

    pu8=(unsigned char *)&GtMould_1;    //把结构体变量强制转换成“以字节为单位”的指针
    for(i=0;i<sizeof(GtMould_1);i++)
    {
```



```

        Gu8Buffer[i]=pu8[i];    //把结构体变量以字节的方式搬运并且存储到普通数组里。
    }

    ptStruct=(struct StructMould_1 *)&Gu8Buffer[0]; //再把普通数组强制“还原”成结构体指针
    ptStruct->u8Data_A=ptStruct->u8Data_A+1;    //该变量从 5 自加 1 后变成 6。
    ptStruct->u32Data_B=ptStruct->u32Data_B+1; //该变量从 6 自加 1 后变成 7。
    ptStruct->u16Data_C=ptStruct->u16Data_C+1; //该变量从 7 自加 1 后变成 8。

    View(ptStruct->u8Data_A); //在电脑端观察结构体成员 u8Data_A 的数值。
    View(ptStruct->u32Data_B); //在电脑端观察结构体成员 u32Data_B 的数值。
    View(ptStruct->u16Data_C); //在电脑端观察结构体成员 u16Data_C 的数值。

    while(1)
    {
    }
}
/*---C 语言学习区域的结束。-----*/

```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第 1 个数

十进制:6

十六进制:6

二进制:110

第 2 个数

十进制:7

十六进制:7

二进制:111

第 3 个数

十进制:8

十六进制:8

二进制:1000

分析：

结构体成员 u8Data\_A 的数值是 6。

结构体成员 u32Data\_B 的数值是 7。

结构体成员 u16Data\_C 的数值是 8。

### 【73.3 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。

## 第七十四节： 结构体指针在函数接口处的频繁应用。

### 【74.1 重温“函数的接口参数”。】

函数的接口参数主要起到标识的作用。比如：

一个加法函数：

```
unsigned char add(unsigned char a,unsigned char b)
{
    return (a+b);
}
```

这里的 a 和 b 就是接口参数，它的作用是告诉人们，你把两个加数分别代入 a 和 b，返回的就是你要的加法运算结果。这里的接口参数就起到入口标识的作用。注意，这句话的关键词是“标识”而不是“入口”，因为函数的“入口”不是唯一的，而是无数条路径。为什么这么说？我们把上面的例子改一下，改成全局变量，例子如下：

一个加法函数：

```
unsigned char a; //加数
unsigned char b; //加数
unsigned char c; //和
void add(void)
{
    c=a+b;
}
```

上述例子中，尽管我用“两个”void（空的）关键词把原来加法函数的入口（接口参数）和出口（return 返回）都堵得死死的，但是，全局变量是无法阻挡的，它进入一个函数的内部不受任何限制，也就是说，我们做项目的时候，如果把所有函数的接口参数和返回都改成 void 类型，所有的信息传递都改用全局变量，这样也是可以勉强把项目做完的。但是，如果真的把所有函数的接口参数都改成 void，全部靠全局变量来传递信息，那么最大的问题是函数多了之后，阅读非常不方便，你每看到一个被调用的函数，你不能马上猜出它大概跟哪些全局变量发生了关联，你必须一个一个的去查该函数的源代码才能理清楚，针对这个问题，C 语言的设计者，给了函数非常丰富的接口参数，最理想的函数是：你把凡是与此函数相关的全局变量都经过接口参数的入口才进入到函数内部，尽量把接口参数的入口看作是函数的唯一合法入口（尽管不是唯一也不是必须），这样只要看函数的接口参数就知道这个函数跟哪些全局变量有关，函数的输入输出就非常清晰明了。但是问题又来了，如果有多少个全局变量就开多少个接口参数，接口参数就会变得非常多，接口参数多了，函数的门面就非常难看，无异于把本来应该“小而窄”的接口设在“宽而广”的平原上，还不如直接用原来那种全局变量强行进入呢。那么，要解决这个问题怎么办？本节的主角“结构体指针”可以解决这个问题。

### 【74.2 结构体指针在函数接口处的频繁应用。】

当函数的接口参数非常多的时候，可以把 N 个相关的全局变量“打包”成一个结构体数据，碰到函数接口的时候，可以通过“结构体指针”以“包”为单位的方式进入，这样就可以让函数的接口参数看起来非常少，这种方法，是很多 32 位单片机的库函数一直在用的方法，它最重要的好处是简化入口的通道数量。你想想，32 位单片机有那么多寄存器，如果没有这种以“结构体指针”为接口参数的方式，它的入口可能需要

几十个接口参数，那岂不是非常麻烦？库函数设计的成败与否，本来就在于接口的设计合不合理，“结构体指针作为函数接口参数”在此场合就显得特别有价值，使用了这种方法，函数与全局变量之间，它们的关联脉络再也不用隐藏起来，并且可以很清晰的表达清楚。现在举一个例子，比如有一个函数，要实现把 5 个全局变量“自加 1”的功能，分别使用两种接口参数来实现，例子如下：

第一种方式：有多少个全局变量就开多少个接口参数。

```
//函数的声明
void Add_One( unsigned char *pu8Data_1, //第 1 个接口参数
             unsigned char *pu8Data_2, //第 2 个接口参数
             unsigned char *pu8Data_3, //第 3 个接口参数
             unsigned char *pu8Data_4, //第 4 个接口参数
             unsigned char *pu8Data_5); //第 5 个接口参数

//5 个全局变量的定义
unsigned char a;
unsigned char b;
unsigned char c;
unsigned char d;
unsigned char e;

//函数的定义
void Add_One( unsigned char *pu8Data_1, //第 1 个接口参数
             unsigned char *pu8Data_2, //第 2 个接口参数
             unsigned char *pu8Data_3, //第 3 个接口参数
             unsigned char *pu8Data_4, //第 4 个接口参数
             unsigned char *pu8Data_5) //第 5 个接口参数
{
    *pu8Data_1=(*pu8Data_1)+1; //实现自加 1 的功能
    *pu8Data_2=(*pu8Data_2)+1;
    *pu8Data_3=(*pu8Data_3)+1;
    *pu8Data_4=(*pu8Data_4)+1;
    *pu8Data_5=(*pu8Data_5)+1;
}

void main()
{
    //5 个全局变量都初始化为 0
    a=0;
    b=0;
```

```

c=0;
d=0;
e=0;

//函数的调用，实现 5 个变量都“自加 1”的功能。加“&”表示“传址”的方式进入函数内部。
Add_One(&a, //第 1 个接口参数
        &b, //第 2 个接口参数
        &c, //第 3 个接口参数
        &d, //第 4 个接口参数
        &e); //第 5 个接口参数
}

```

第二种方式：把 N 个全局变量打包成一个结构体，以“结构体指针”的方式进入函数内部。

```

//函数的声明
void Add_One(struct StructMould *ptMould); //只有 1 个结构体指针，大大减少了接口参数。

//结构体的“造模”
struct StructMould
{
    unsigned char a;
    unsigned char b;
    unsigned char c;
    unsigned char d;
    unsigned char e;
};

struct StructMould GtMould; //生成一个结构体变量，内部包含了 5 个全局变量 a, b, c, d, e。

//函数的定义
void Add_One(struct StructMould *ptMould) //只有 1 个结构体指针，大大减少了接口参数。
{
    ptMould->a=ptMould->a+1; //实现“自加 1”的功能。
    ptMould->b=ptMould->b+1;
    ptMould->c=ptMould->c+1;
    ptMould->d=ptMould->d+1;
    ptMould->e=ptMould->e+1;
}

void main()
{
    //5 个全局变量的结构体成员都初始化为 0
    GtMould.a=0;
    GtMould.b=0;

```

```

    GtMould.c=0;
    GtMould.d=0;
    GtMould.e=0;

    //函数的调用，实现 5 个变量都“自加 1”的功能。加“&”表示“传址”的方式进入函数内部。
    Add_One(&GtMould); //只有 1 个结构体指针，大大减少了接口参数。
}

```

### 【74.3 例程练习和分析。】

现在编写一个“以结构体指针为函数接口参数”的练习程序。

```

/*---C 语言学习区域的开始。-----*/

//函数的声明
void Add_One(struct StructMould *ptMould); //只有 1 个结构体指针，大大减少了接口参数。

//结构体的“造模”
struct StructMould
{
    unsigned char a;
    unsigned char b;
    unsigned char c;
    unsigned char d;
    unsigned char e;
};

struct StructMould GtMould; //生成一个结构体变量，内部包含了 5 个全局变量 a, b, c, d, e。

//函数的定义
void Add_One(struct StructMould *ptMould) //只有 1 个结构体指针，大大减少了接口参数。
{
    ptMould->a=ptMould->a+1; //实现“自加 1”的功能。
    ptMould->b=ptMould->b+1;
    ptMould->c=ptMould->c+1;
    ptMould->d=ptMould->d+1;
    ptMould->e=ptMould->e+1;
}

void main() //主函数
{
    //5 个全局变量的结构体成员都初始化为 0
    GtMould.a=0;
    GtMould.b=0;

```

```

    GtMould.c=0;
    GtMould.d=0;
    GtMould.e=0;

    //函数的调用，实现 5 个变量都“自加 1”的功能。加“&”表示“传址”的方式进入函数内部。
    Add_One(&GtMould); //只有 1 个结构体指针，大大减少了接口参数。

    View(GtMould.a); //在电脑端观察结构体成员 GtMould.a 的数值。
    View(GtMould.b); //在电脑端观察结构体成员 GtMould.b 的数值。
    View(GtMould.c); //在电脑端观察结构体成员 GtMould.c 的数值。
    View(GtMould.d); //在电脑端观察结构体成员 GtMould.d 的数值。
    View(GtMould.e); //在电脑端观察结构体成员 GtMould.e 的数值。

    while(1)
    {
    }
}
/*---C 语言学习区域的结束。-----*/

```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第 1 个数

十进制:1

十六进制:1

二进制:1

第 2 个数

十进制:1

十六进制:1

二进制:1

第 3 个数

十进制:1

十六进制:1

二进制:1

第 4 个数

十进制:1

十六进制:1

二进制:1

第 5 个数

十进制:1  
十六进制:1  
二进制:1

分析:

结构体成员 GtMould.a 的数值是 1。

结构体成员 GtMould.b 的数值是 1。

结构体成员 GtMould.c 的数值是 1。

结构体成员 GtMould.d 的数值是 1。

结构体成员 GtMould.e 的数值是 1。

#### 【74.4 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。



## 第七十五节： 指针的名义（例：一维指针操作二维数组）。

### 【75.1 指针的名义。】

刚开始接触指针往往有这种感觉，指针的江湖很乱，什么“乱七八糟”的指针都能冒出来，空指针，指针的指针，函数的指针，各种名目繁多的指针，似乎都可以打着指针的名义让你招架不住，而随着我们功力的提升，会逐渐拨开云雾，发现指针的真谛不外乎三个，第一个是所有的指针所占字节数都一样，第二个是所有指针的操作本质都是“取地址”，第三个是所有各种不同类型的指针之间的转换都可以用“小括号的类型强制转换”。

### 【75.2 一维指针操作二维数组。】

C 语言讲究门当户对，讲究类型匹配，什么类型的指针就操作什么类型的数据，否则 C 编译器在翻译代码的时候，会给予报错或者警告。如果想甩开因类型不匹配而导致的报错或者警告，就只能使用“小括号的类型强制转换”，这个方法在项目中的应用很频繁，也很实用。一维指针想直接操作二维数组也是必须使用“小括号的类型强制转换”。实际项目中为什么会涉及“一维指针想直接操作二维数组”？二维数组更加像一个由行与列组合而成的表格，而且每行单元的内存地址是连续的，并且上下每行与每行之间的首尾单元的内存地址也是连续的，凡是内存地址连续的都是指针的菜。我曾遇到这样一种情况，要从一个二维表格里提取某一行数据用来显示，而这个显示函数是别人封装好的一个库函数，库函数对外的接口是一维指针，这样，如何把二维表格（二维数组）跟一维指针在接口上兼容起来，就是一个要面临的问题，这时有两种思路，一种是把二维数组的某一行数据先用原始的办法提取出来存放在一个中间变量的一维数组，然后再把这个一维数组代入到一维指针接口的库函数里，另一种思路是绕开中间变量，直接把二维数组的某一行的地址强制转换成一维指针的类型，利用“类型强制转换”绕开 C 编译器的报错或警告，实现二维数组跟一维指针“直通”，经过实验，这种方法果然可以，从此对指针的感悟就又上了一层，原来，指针的“取地址”是不仅仅局限于某个数组的首地址，它完全可以利用类型强制转换的小括号“()”与取地址符号“&”结合起来，让指针跟一维数组或者二维数组里面任何一个单元直接关联起来。请看下面两个例子，用一维指针提取二维数组里面某一行的数据，第一个例子是在程序处理中的类型强制转换的应用，第二个例子是在函数接口中的类型强制转换的应用。

### 【75.3 在程序处理中的类型转换。】

```
unsigned char table[][3]= //二维数组
{
    {0x00, 0x01, 0x02}, //二维数组的第 0 行数据
    {0x10, 0x11, 0x12}, //二维数组的第 1 行数据
    {0x20, 0x21, 0x22}, //二维数组的第 2 行数据
};

unsigned char *pGu8; //一维指针
unsigned char Gu8Buffer[3]; //一维数组，存放从二维数组里提取出来的某一行数据
unsigned char i; // for 循环的变量
void main()
{
    pGu8=(unsigned char *)&table[2][0]; //利用类型强制转换使得一维指针跟二维数组关联起来。
```

```

    for(i=0;i<3;i++)
    {
        Gu8Buffer[i]=pGu8[i];    //提取二维数组的第 2 行数据，存入到一个一维数组里。
    }

    while(1)
    {

    }

}

```

#### 【75.4 在函数接口中的类型转换。】

在函数接口中，也可以利用类型强制转换来实现函数接口的匹配问题，比如，下面这个写法也是合法的。

```

void GetRowData(unsigned char *pu8); //函数的声明

unsigned char table[][3]= //二维数组
{
    {0x00, 0x01, 0x02},    //二维数组的第 0 行数据
    {0x10, 0x11, 0x12},    //二维数组的第 1 行数据
    {0x20, 0x21, 0x22},    //二维数组的第 2 行数据
};

unsigned char  Gu8Buffer[3];    //一维数组，存放从二维数组里提取出来的某一行数据

void GetRowData(unsigned char *pu8) //一维指针的函数接口
{
    unsigned char  i; // for 循环的变量
    for(i=0;i<3;i++)
    {
        Gu8Buffer[i]=pu8[i];    //提取二维数组的某行数据，存入到一个一维数组里。
    }
}

void main()
{
    GetRowData((unsigned char *)&table[2][0]); //利用类型强制转换来兼容一维指针的函数接口

    while(1)
    {

    }

}

```

```
}
```

### 【75.5 注意指针或者数组越界的问题。】

上述例子中，二维数组内部只有 9 个数据，如果指针操作的数据超过了这 9 个数据的地址范围，就会导致系统其它无辜的数据受到破坏，这个问题导致的后果是很严重的，这类指针或者数组越界的问题，大家平时做项目时必须留心注意。

### 【75.6 例程练习和分析。】

现在编写一个练习程序。

```
/*---C 语言学习区域的开始。-----*/

void GetRowData(unsigned char *pu8); //函数的声明

unsigned char table[][3]= //二维数组
{
    {0x00, 0x01, 0x02}, //二维数组的第 0 行数据
    {0x10, 0x11, 0x12}, //二维数组的第 1 行数据
    {0x20, 0x21, 0x22}, //二维数组的第 2 行数据
};

unsigned char Gu8Buffer[3]; //一维数组，存放从二维数组里提取出来的某一行数据

void GetRowData(unsigned char *pu8) //一维指针的函数接口
{
    unsigned char i; // for 循环的变量
    for(i=0;i<3;i++)
    {
        Gu8Buffer[i]=pu8[i]; //提取二维数组的某行数据，存入到一个一维数组里。
    }
}

void main() //主函数
{
    GetRowData((unsigned char *)&table[2][0]); //利用类型强制转换来兼容一维指针的函数接口

    View(Gu8Buffer[0]); //在电脑端观察存放二维数组某行数据的一维数组的内容
    View(Gu8Buffer[1]); //在电脑端观察存放二维数组某行数据的一维数组的内容
    View(Gu8Buffer[2]); //在电脑端观察存放二维数组某行数据的一维数组的内容
    while(1)
```

```
    {  
    }  
}  
/*---C 语言学习区域的结束。-----*/
```

在电脑串口助手软件上观察到的程序执行现象如下：

```
开始...  
  
第 1 个数  
十进制:32  
十六进制:20  
二进制:100000  
  
第 2 个数  
十进制:33  
十六进制:21  
二进制:100001  
  
第 3 个数  
十进制:34  
十六进制:22  
二进制:100010
```

分析：

Gu8Buffer[0]是十六进制的 0x20，提取了二维数组第 2 行中的某数据。

Gu8Buffer[1]是十六进制的 0x21，提取了二维数组第 2 行中的某数据。

Gu8Buffer[2]是十六进制的 0x22，提取了二维数组第 2 行中的某数据。

## 【75.7 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。

## 第七十六节： 二维数组的指针。

### 【76.1 二维数组指针的用途。】

前面章节讲了一维指针操作二维数组，本质是通过“类型强制转换”实现的，这种应用局限于某些特定的场合，毕竟一维有 1 个下标，二维有 2 个下标，一维和二维在队形感上是有明显差别的，强行用一维指针操作二维数组会破坏了代码原有的队形感，大多数的情况，还是用二维指针操作二维数组。

二维指针主要应用在两个方面，一方面是 N 个二维数组的“中转站”应用，另一方面是函数接口的应用。比如，当某项目有 N 个二维数组表格时，要通过某个变量来切换处理某个特定的表格，以便实现“N 选一”的功能，此时，二维指针在这 N 个二维数组之间就起到中转站的作用。又，当某个函数接口想输入或者输出一个二维数组时，就必然要用到二维指针作为函数的接口参数。

### 【76.2 二维指针的“中转站”应用。】

举一个例子，有 3 个现有的二维数组，通过某个变量来选择切换，把某个二维数组的数据复制到指定的一个缓存数组中。

```
code unsigned char table_1[3][3]= //第 1 个现有的二维数组
{
    {0x00, 0x01, 0x02},
    {0x10, 0x11, 0x12},
    {0x20, 0x21, 0x22},
};

code unsigned char table_2[3][3]= //第 2 个现有的二维数组
{
    {0xA0, 0xA1, 0xA2},
    {0xB0, 0xB1, 0xB2},
    {0xC0, 0xC1, 0xC2},
};

code unsigned char table_3[3][3]= //第 3 个现有的二维数组
{
    {0xD0, 0xD1, 0xD2},
    {0xE0, 0xE1, 0xE2},
    {0xF0, 0xF1, 0xF2},
};

unsigned char SaveBuffer[3][3]; //指定的一个缓存数组

unsigned char TableSec; //选择变量
const unsigned char (*pTable)[3]; //“中转站”的二维指针
unsigned char R,L; //复制数据时用到的 for 循环变量
void main()
```

```

{
    TableSec=2; //选择第 2 个现有的二维数组
    switch(TableSec) //根据选择变量来切换选择某个现有的二维数组
    {
        case 1: //选择第 1 个现有二维数组
            pTable=table_1; //二维指针 pTable 在这里关联指定的数组，起到中转站的作用。
            break;
        case 2: //选择第 2 个现有二维数组
            pTable=table_2; //二维指针 pTable 在这里关联指定的数组，起到中转站的作用。
            break;
        case 3: //选择第 3 个现有二维数组
            pTable=table_2; //二维指针 pTable 在这里关联指定的数组，起到中转站的作用。
            break;
    }

    //通过二维指针 pTable 来复制数据到指定的缓存数组 SaveBuffer
    for(R=0;R<3;R++) //行循环
    {
        for(L=0;L<3;L++) //列循环
        {
            SaveBuffer[R][L]=pTable[R][L]; //这里能看到，二维指针维护了二维数组的队形感
        }
    }

    while(1)
    {

    }
}

```

### 【76.3 二维指针在“函数接口”中的应用。】

把上述例子“复制过程”的代码封装成一个函数，实现的功能还是一样，有 3 个现有的二维数组，通过某个变量来选择切换，把某个二维数组的数据复制到指定的一个缓存数组中。

```

//函数声明
void CopyBuffer(const unsigned char (*pTable)[3], unsigned char (*pSaveBuffer)[3]);

code unsigned char table_1[3][3]= //第 1 个现有的二维数组
{
    {0x00, 0x01, 0x02},
    {0x10, 0x11, 0x12},
    {0x20, 0x21, 0x22},

```

```

};

code unsigned char table_2[3][3]= //第 2 个现有的二维数组
{
    {0xA0, 0xA1, 0xA2},
    {0xB0, 0xB1, 0xB2},
    {0xC0, 0xC1, 0xC2},
};

code unsigned char table_3[3][3]= //第 3 个现有的二维数组
{
    {0xD0, 0xD1, 0xD2},
    {0xE0, 0xE1, 0xE2},
    {0xF0, 0xF1, 0xF2},
};

unsigned char SaveBuffer[3][3]; //指定的一个缓存数组

unsigned char TableSec; //选择变量

//*pTable 是输入接口带 const 修饰, *pSaveBuffer 是输出结果的接口无 const。
void CopyBuffer(const unsigned char (*pTable)[3], unsigned char (*pSaveBuffer)[3])
{
    unsigned char R, L; //复制数据时用到的 for 循环变量

    for(R=0; R<3; R++) //行循环
    {
        for(L=0; L<3; L++) //列循环
        {
            pSaveBuffer[R][L]=pTable[R][L]; //这里能看到, 二维指针维护了二维数组的队形感
        }
    }
}

void main()
{
    TableSec=2; //选择第 2 个现有的二维数组
    switch(TableSec) //根据选择变量来切换选择某个现有的二维数组
    {
        case 1: //选择第 1 个现有二维数组
            CopyBuffer(table_1, SaveBuffer); //二维指针在这里分别体现了输入和输出接口作用
            break;
        case 2: //选择第 2 个现有二维数组
            CopyBuffer(table_2, SaveBuffer); //二维指针在这里分别体现了输入和输出接口作用
    }
}

```

```

        break;
    case 3:    //选择第 3 个现有二维数组
        CopyBuffer(table_3, SaveBuffer); //二维指针在这里分别体现了输入和输出接口作用
        break;
    }
    while(1)
    {

    }
}

```

#### 【76.4 二维指针“类型强制转换”的书写格式。】

unsigned char \*pu8, unsigned int \*pu16, unsigned int \*pu32 这些指针的书写定义都是很有规则感的，相比之下，二维指针的定义显得缺乏规则感，比如定义的二维指针变量 unsigned char (\*pTable)[3]，不规则在哪？就在于二维指针的变量 pTable 嵌入到了括号中去，跟符号“\*”捆绑在一起，这时就会冒出一个问题，如果我要强制某个指针变量为二维指针怎么办？下面的例子已经给出了答案。

```

unsigned char table[3][3]= //二维数组
{
    {0xD0, 0xD1, 0xD2},
    {0xE0, 0xE1, 0xE2},
    {0xF0, 0xF1, 0xF2},
};

unsigned char (*pTable)[3];

void main()
{
    pTable=(unsigned char (*)[3])table; //这里，强制类型转换用 unsigned char (*)[3]
}

```

总结：二维数组的强制类型转换用这种书写格式（unsigned char (\*)[N]），这里的 N 是代表实际项目中某数组的“列”数。

#### 【76.5 例程练习和分析。】

现在编写一个练习程序。

```

/*---C 语言学习区域的开始。-----*/
void CopyBuffer(const unsigned char (*pTable)[3], unsigned char (*pSaveBuffer)[3]);

code unsigned char table_1[3][3]= //第 1 个现有的二维数组
{
    {0x00, 0x01, 0x02},

```



```

    {0x10, 0x11, 0x12},
    {0x20, 0x21, 0x22},
};

code unsigned char table_2[3][3]= //第 2 个现有的二维数组
{
    {0xA0, 0xA1, 0xA2},
    {0xB0, 0xB1, 0xB2},
    {0xC0, 0xC1, 0xC2},
};

code unsigned char table_3[3][3]= //第 3 个现有的二维数组
{
    {0xD0, 0xD1, 0xD2},
    {0xE0, 0xE1, 0xE2},
    {0xF0, 0xF1, 0xF2},
};

unsigned char SaveBuffer[3][3]; //指定的一个缓存数组

unsigned char TableSec; //选择变量

//*pTable 是输入接口带 const 修饰, *pSaveBuffer 是输出结果的接口无 const。
void CopyBuffer(const unsigned char (*pTable)[3], unsigned char (*pSaveBuffer)[3])
{
    unsigned char R, L; //复制数据时用到的 for 循环变量

    for(R=0; R<3; R++) //行循环
    {
        for(L=0; L<3; L++) //列循环
        {
            pSaveBuffer[R][L]=pTable[R][L]; //这里能看到, 二维指针维护了二维数组的队形感
        }
    }
}

void main() //主函数
{
    TableSec=2; //选择第 2 个现有的二维数组
    switch(TableSec) //根据选择变量来切换选择某个现有的二维数组
    {
        case 1: //选择第 1 个现有二维数组
            CopyBuffer(table_1, SaveBuffer); //二维指针在这里分别体现了输入和输出接口作用

```

```

        break;
    case 2:    //选择第 2 个现有二维数组
        CopyBuffer(table_2, SaveBuffer); //二维指针在这里分别体现了输入和输出接口作用
        break;
    case 3:    //选择第 3 个现有二维数组
        CopyBuffer(table_3, SaveBuffer); //二维指针在这里分别体现了输入和输出接口作用
        break;
}

View(SaveBuffer[0][0]); //在电脑端观察某个二维数组第 0 行数据第 0 个元素的内容
View(SaveBuffer[0][1]); //在电脑端观察某个二维数组第 0 行数据第 1 个元素的内容
View(SaveBuffer[0][2]); //在电脑端观察某个二维数组第 0 行数据第 2 个元素的内容
while(1)
{
}
}
/*---C 语言学习区域的结束。-----*/

```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第 1 个数

十进制:160

十六进制:A0

二进制:10100000

第 2 个数

十进制:161

十六进制:A1

二进制:10100001

第 3 个数

十进制:162

十六进制:A2

二进制:10100010

分析：

SaveBuffer[0][0]是十六进制的 0xA0，提取了第 2 个二维数组的第 0 行第 0 个数据。

SaveBuffer[0][1]是十六进制的 0xA1，提取了第 2 个二维数组的第 0 行第 1 个数据。

SaveBuffer[0][2]是十六进制的 0xA2，提取了第 2 个二维数组的第 0 行第 2 个数据。

## 【76.6 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。

## 第七十七节： 指针唯一的“单向输出”通道 return。

### 【77.1 指针的“单向”输出通道。】

函数的接口有两个地方，一个是函数名“后面”的小括号所包含的接口参数，另一个是函数名“前面”通过函数内部 return 返回出来的“return 返回类型”。比如：

```
return 返回类型    函数名(接口参数, 接口参数...)

unsigned char HanShu(unsigned char a,unsigned char b) //a 和 b 是函数名“后面”的接口参数
{
    unsigned char c;
    c=a+b;
    return c;    //函数内部返回出来的“return 返回类型”
}
```

指针在“函数名后面小括号所包含的接口参数”的地方时，可以是一个“双向”口（输入和输出），如果在指针前面加上 const 关键字修饰，可以把“双向”改为只能输入的“单向”口，注意，这里所说的“单向”是指“输入的单向”，但是做不到“输出的单向”，指针如果想做到“输出的单向”，就必须通过 return 这个通道。return 返回指针这个功能很常用，比如用 32 位单片机想做比较漂亮的显示界面时，大家往往喜欢用到 emWIN 这个界面显示系统，而 emWIN 提供了很多库函数，这些库函数用了很多 return 返回的“句柄”，“句柄”其实就是指指针，比如类似以下这行代码：

```
hItem = WM_GetDialogItem(hWin_FrameWin_GetClientWindow, ID_LISTVIEW_0); //获取某个控件的句柄
```

其中 hItem 就是“句柄”，本质就是函数内部 return 返回出来的指针。

所以本节内容主要是想告诉大家，return 不仅可以返回普通的变量，也是可以返回指针的，而且还很常用。具体内容请看下面 77.2 例子中的讲解。

### 【77.2 例程练习和分析。】

编写一个函数，要从一个二维表格的数组中提取其中某一行的数据，用 return 这个返回输出的通道来接收该行数据的地址（指针），然后再通过这个指针的间接调用，把该行数据全部显示出来。

```
/*---C 语言学习区域的开始。-----*/

unsigned char *GetRowData(unsigned char (*pu8Table)[3],unsigned char u8RowSec); //函数声明

unsigned char table[][3]= //二维数组
{
    {0x00,0x01,0x02}, //二维数组的第 0 行数据
    {0x10,0x11,0x12}, //二维数组的第 1 行数据
}
```

```

{0x20, 0x21, 0x22}, //二维数组的第 2 行数据
};

//函数名前面是 unsigned char *, 代表内部 return 返回的是 unsigned char * 的指针。
unsigned char *GetRowData(unsigned char (*pu8Table)[3], unsigned char u8RowSec)
{
    unsigned char *pu8Row;
    pu8Row=(unsigned char *)&pu8Table[u8RowSec][0]; //提取某一行开始的地址（指针）
    return pu8Row; //经过 return 通道对外输出指针，pu8Row 是一个指针类型的变量。
}

unsigned char *pGu8Row; //接收 return 输出的指针
unsigned char Gu8Buffer[3]; //一维数组，存放从二维数组里提取出来的某一行数据
unsigned char i; // for 循环的变量

void main() //主函数
{
    pGu8Row=GetRowData(table, 0); //这里的 0 是表示选择二维表格的第 0 行数据
    for(i=0; i<3; i++)
    {
        Gu8Buffer[i]=pGu8Row[i]; //通过指针 pGu8Row 来搬运数据到一维数组 Gu8Buffer
    }
    View(Gu8Buffer[0]); //在电脑端观察存放二维数组某行数据的一维数组的内容
    View(Gu8Buffer[1]); //在电脑端观察存放二维数组某行数据的一维数组的内容
    View(Gu8Buffer[2]); //在电脑端观察存放二维数组某行数据的一维数组的内容
    while(1)
    {
    }
}

/*---C 语言学习区域的结束。-----*/

```

在电脑串口助手软件上观察到的程序执行现象如下：

开始...

第 1 个数

十进制:0

十六进制:0

二进制:0

第 2 个数

十进制:1

十六进制:1

二进制:1

第 3 个数

十进制:2

十六进制:2

二进制:10

分析:

Gu8Buffer[0]是 0，提取了二维数组的第 0 行第 0 个数据。

Gu8Buffer[1]是 1，提取了二维数组的第 0 行第 1 个数据。

Gu8Buffer[2]是 2，提取了二维数组的第 0 行第 2 个数据。

### 【77.3 如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。

## 第七十八节： typedef 和#define 和 enum。

### 【78.1 typedef 和#define 和 enum。】

typedef 称为“类型定义”，#define 称为“宏定义”，enum 称为“枚举”。三者都有“一键替换”的能力，但是应用的侧重点各有不同。请看下面的例子，要写一个函数，把学生的分数分为 3 个等级，第 1 等级是“优”（范围：“优”>=90 分），第 2 等级是“中”（范围：70 分<=“中”<90 分），第 3 等级是“差”（范围：“差”<70 分），实现此算法的函数需要一个输入口和一个输出口，用来输入分数和输出判断结果，判断的结果用三个数字常量 0, 1, 2 来表示，0 代表“优”，1 代表“中”，2 代表“差”。代码如下：

```
unsigned char GetGrade(unsigned char u8Score)
{
    if(u8Score<70)
    {
        return 2; //2 代表“差”
    }
    else if(u8Score>=70&&u8Score<90)
    {
        return 1; //1 代表“中”
    }
    else
    {
        return 0; //0 代表“优”
    }
}
```

上述代码没有添加任何“typedef，#define，enum”，是“素颜照”级别的原始代码。现在对上述代码做一些美容，加入“typedef，#define，enum”的元素，代码如下：

```
#define BAD_MEDIUM    70 //宏定义。用 BAD_MEDIUM 来表示“差”和“中”分数的分界线
#define MEDIUM_GOOD  90 //宏定义。用 MEDIUM_GOOD 来表示“中”和“优”分数的分界线

typedef unsigned char u8; //用 typedef 为类型“unsigned char”增加一个名为“u8”的代言人

enum {GOOD = 0, MEDIUM, BAD}; //用 enum 把“0, 1, 2”三个常量转换为“GOOD, MEDIUM, BAD”

u8 GetGrade(u8 u8Score)
{
    if(u8Score<BAD_MEDIUM) //等级分数分界线的判断
    {
        return BAD; //BAD 就是常量 2，代表“差”。
    }
    else if(u8Score>=BAD_MEDIUM&&u8Score<MEDIUM_GOOD) //等级分数分界线的判断
    {
```

```

        return MEDIUM; //MEDIUM 就是常量 1，代表“中”
    }
    else
    {
        return GOOD;    //GOOD 就是常量 0，代表“优”
    }
}

```

代码赏析：

赏析片段一：

```

#define BAD_MEDIUM 70 //宏定义。用 BAD_MEDIUM 来表示“差”和“中”分数的分界线
#define MEDIUM_GOOD 90 //宏定义。用 MEDIUM_GOOD 来表示“良”和“优”分数的分界线

```

这里，用宏定义#define 来关联分界线判断的分数，给后续代码的升级维护带来了便捷，因为用户有可能会要求把“差”“中”“优”三者的分数线进行调整，这时直接更改 70 和 90 这个数值就可以实现分数线的调整。可见，宏定义#define 经常用在涉及“分界线”判断的场合。

赏析片段二：

```

typedef unsigned char u8; //用 typedef 为类型“unsigned char”增加一个名为“u8”的代言人

```

用类型定义 typedef 为类型“unsigned char”增加一个名为“u8”的代言人，u 代表 unsigned 的 u，8 代表此类型占用 8 位，比如 unsigned char 就是占用 8 位的 unsigned 类型，所以用 u8。如果是 16 位的 unsigned 类型就用 u16，32 位则用 u32，这都是单片机界的常用命名习惯。上述代码用了类型定义，今后代码中凡是想定义一个 unsigned char 变量，都可以直接用 u8 来替代。这样有两个好处：第一个好处，u8 的字符个数明显比 unsigned char 少，省了敲代码的力气。第二个好处，方便代码在各种不同硬件平台上的移植，因为不同的单片机不同的编译器对 unsigned char，unsigned int，unsigned long 翻译所得的结果是不一样的，比如，51 单片机的 unsigned int 是占用 16 位的，而很多 32 位单片机的 unsigned int 是占用 32 位的，它们的 16 位则用 unsigned short int 类型，而不是 unsigned int。

当我们用 51 单片机写代码的时候，可以如下类型定义：

```

typedef unsigned char    u8;
typedef unsigned int     u16;
typedef unsigned long    u32;

```

当我们用 32 位的单片机写代码的时候，可以如下类型定义：

```

typedef unsigned char    u8;
typedef unsigned short int u16;
typedef unsigned int     u32;

```

这样，当我们想把 51 单片机的代码移到 32 位的单片机上时，只需要修改类型定义 typedef 这部分的代码，就可以快速做到代码在不同编译器平台上的类型兼容。



赏析片段三：

```
enum {GOOD = 0, MEDIUM, BAD}; //用 enum 把 “0, 1, 2” 三个常量转换为 “GOOD, MEDIUM, BAD”
```

用枚举 enum 把 “0, 1, 2” 三个常量转换为 “GOOD, MEDIUM, BAD” 英文单词，最大的好处就是方便代码的阅读和修改。再多补充一点枚举的基础知识，上述代码中，第一个英文单词 GOOD，经过 “GOOD = 0” 这条初始化的语句后，等效于常量 0，后面的 MEDIUM 和 BAD 则 C 编译器自动对它们进行“累加 1”排序，所以 MEDIUM 和 BAD 分别为常量 1, 2，这是 C 语言的语法规则。枚举 enum 的应用侧重在某些涉及到“状态”的数据类型，但是也不绝对。

## 【78.2 enum 和 typedef 的结合。】

enum 一旦搭载上 typedef 后，可以把各自的特性发挥得淋漓尽致，产生另外一种常见的用途，那就是“人造”数据类型的用途，这里的“人造”解读为“人为制造”之意。比如上述 78.1 的函数 u8 GetGrade(u8 u8Score)，输出接口接收的是 u8 类型，但是内部 return 返回的是枚举类型的 “GOOD, MEDIUM, BAD” 其中之一，而 u8 虽然也能接收和兼容常量 “GOOD, MEDIUM, BAD”，但是总是感觉有点“类型不匹配”的“不适感”，如果想消除这点“不适感”，可以用 enum 和 typedef 相结合的办法，修改后代码如下：

```
#define BAD_MEDIUM 70 //宏定义。用 BAD_MEDIUM 来表示“差”和“中”分数的分界线
#define MEDIUM_GOOD 90 //宏定义。用 MEDIUM_GOOD 来表示“良”和“优”分数的分界线

typedef unsigned char u8; //用 typedef 为类型“unsigned char”增加一个名为“u8”的代言人

typedef enum {
    GOOD = 0,
    MEDIUM,
    BAD
} Grade; //通过 typedef 和 enum 的结合，“人造”出一个新的数据类型 Grade。

Grade GetGrade(u8 u8Score) //这里返回的类型是 Grade，而“GOOD, MEDIUM, BAD”就是属于 Grade
{
    if(u8Score < BAD_MEDIUM) //等级分数分界线的判断
    {
        return BAD; //BAD 就是常量 2，代表“差”。
    }
    else if(u8Score >= BAD_MEDIUM && u8Score < MEDIUM_GOOD) //等级分数分界线的判断
    {
        return MEDIUM; //MEDIUM 就是常量 1，代表“中”
    }
    else
    {
        return GOOD; //GOOD 就是常量 0，代表“优”
    }
}
```

### 【78.3 例程练习和分析。】

为了熟悉 typedef, #define, enum 的用法, 现在要写一个函数, 把学生的分数分为 3 个等级, 第 1 等级是“优”(范围: “优” >=90 分), 第 2 等级是“中”(范围: 70 分 <= “中” <90 分), 第 3 等级是“差”(范围: “差” <70 分), 实现此算法的函数需要一个输入口和一个输出口, 用来输入分数和输出判断结果, 判断的结果用三个数字常量 0, 1, 2 来表示, 0 代表“优”, 1 代表“中”, 2 代表“差”。

```
/*---C 语言学习区域的开始。-----*/

#define BAD_MEDIUM 70 //宏定义。用 BAD_MEDIUM 来表示“差”和“中”分数的分界线
#define MEDIUM_GOOD 90 //宏定义。用 MEDIUM_GOOD 来表示“良”和“优”分数的分界线

typedef unsigned char u8; //用 typedef 为类型“unsigned char”增加一个名为“u8”的代言人

typedef enum {
    GOOD = 0,
    MEDIUM,
    BAD
} Grade; //通过 typedef 和 enum 的相结合, “人造”出一个新的数据类型 Grade。

Grade GetGrade(u8 u8Score); //函数声明

Grade a; //“人造”出 Grade 类型的变量 a, 用来接收函数的判断结果。
Grade b; //“人造”出 Grade 类型的变量 b, 用来接收函数的判断结果。
Grade c; //“人造”出 Grade 类型的变量 c, 用来接收函数的判断结果。

Grade GetGrade(u8 u8Score) //这里返回的类型是 Grade, 而“GOOD, MEDIUM, BAD”就是属于 Grade
{
    if (u8Score < BAD_MEDIUM) //等级分数分界线的判断
    {
        return BAD; //BAD 就是常量 2, 代表“差”。
    }
    else if (u8Score >= BAD_MEDIUM && u8Score < MEDIUM_GOOD) //等级分数分界线的判断
    {
        return MEDIUM; //MEDIUM 就是常量 1, 代表“中”
    }
    else
    {
        return GOOD; //GOOD 就是常量 0, 代表“优”
    }
}

void main() //主函数
```

```

{
    a=GetGrade(98); //输入 98 分, a 来接收判断的结果
    b=GetGrade(88); //输入 88 分, b 来接收判断的结果
    c=GetGrade(68); //输入 68 分, c 来接收判断的结果

    View(a); //在电脑端观察 98 分的判断结果 a
    View(b); //在电脑端观察 88 分的判断结果 b
    View(c); //在电脑端观察 68 分的判断结果 c
    while(1)
    {
    }
}
/*---C 语言学习区域的结束。-----*/

```

在电脑串口助手软件上观察到的程序执行现象如下:

开始...

第 1 个数

十进制:0

十六进制:0

二进制:0

第 2 个数

十进制:1

十六进制:1

二进制:1

第 3 个数

十进制:2

十六进制:2

二进制:10

分析:

98 分的判断结果 a 为 0, 0 代表“优”。

88 分的判断结果 b 为 1, 1 代表“中”。

68 分的判断结果 c 为 2, 2 代表“差”。

#### 【78.4 如何在单片机上练习本章节 C 语言程序?】

直接复制前面章节中第十一节的模板程序, 练习代码时只需要更改“C 语言学习区域”的代码就可以了, 其它部分的代码不要动。编译后, 把程序下载进带串口的 51 学习板, 通过电脑端的串口助手软件就可以观察到不同的变量数值, 详细方法请看第十一节内容。

## 第七十九节： 各种变量常量的命名规范。

### 【79.1 命名规范的必要。】

一个大型的项目程序，涉及到的变量常量非常多，各种变量常量眼花缭乱，名字不规范就无法轻松掌控全局。若能一开始就遵守特定的命名规范，则普天之下，率土之滨，都被你牢牢地掌控在手里，天下再也没有难维护的代码。本节教给大家的是我多年实践所沿用的命名规范和习惯，它不是唯一绝对的，只是给大家参考，大家今后也可以在自己的实践中慢慢总结出一套适合自己的命名规范和习惯。

### 【79.2 普通变量常量的命名规范和习惯。】

在 C51 编译器的平台下，unsigned char , unsigned int , unsigned long 三类常用的变量代表了“无符号的 8 位，16 位，32 位”，这类型的变量前缀分别加“u8, u16, u32”来表示。但是这种类型的变量还分全局变量和局部变量，为了有所区分，就在全局变量前加“G”来表示，不带“G”的就默认是局部变量。比如：

```
unsigned char Gu8Number;    //Gu8 就代表全局的 8 位变量
unsigned int  Gu16Number;   //Gu16 就代表全局的 16 位变量
unsigned long Gu32Number;   //Gu32 就代表全局的 32 位变量

void HanShu(unsigned char u8Data) //u8 就代表局部的 8 位变量
{
    unsigned char u8Number;    //u8 就代表局部的 8 位变量
    unsigned int  u16Number;   //u16 就代表局部的 16 位变量
    unsigned long u32Number;   //u32 就代表局部的 32 位变量
}
```

全局变量和局部变量继续往下细分，还分“静态”和“非静态”，为了有所区分，就在前面增加“ES”或“S”来表示，“ES”代表全局的静态变量，“S”代表局部的静态变量。比如：

```
static unsigned char ESu8Number;    //ESu8 就代表全局的 8 位静态变量
static unsigned int  ESu16Number;   //ESu16 就代表全局的 16 位静态变量
static unsigned long ESu32Number;   //ESu32 就代表全局的 32 位静态变量

void HanShu(unsigned char u8Data) //u8 就代表局部的 8 位变量
{
    static unsigned char Su8Number;    //Su8 就代表局部的 8 位静态变量
    static unsigned int  Su16Number;   //Su16 就代表局部的 16 位静态变量
    static unsigned long Su32Number;   //Su32 就代表局部的 32 位静态变量
}
```

刚才讲的只是针对“变量”，如果是“常量”，则前缀加“C”来表示，不管是全局的常量还是局部的常量，都统一用“C”来表示，不再刻意区分“全局常量”和“静态常量”，比如：

```
const unsigned char Cu8Number=1;    //Cu8 就代表 8 位常量，不刻意区分“全局”和“局部”
```

```

const unsigned int Cu16Number=1;    //Cu16 就代表 16 位常量，不刻意区分“全局”和“局部”
const unsigned long Cu32Number=1;   //Cu32 就代表 32 位常量，不刻意区分“全局”和“局部”

void HanShu(unsigned char u8Data) //u8 就代表局部的 8 位变量
{
    const unsigned char Cu8Number=1; //Cu8 就代表 8 位常量，不刻意区分“全局”和“局部”
    const unsigned int Cu16Number=1; //Cu16 就代表 16 位常量，不刻意区分“全局”和“局部”
    const unsigned long Cu32Number=1; //Cu32 就代表 32 位常量，不刻意区分“全局”和“局部”
}

```

### 【79.3 循环体变量的命名规范和习惯。】

循环体变量是一个很特定场合用的变量，为了突出它的特殊，这类变量在命名上用单个字母，可以不遵守命名规范，这里的“不遵守命名规范”就是它的“命名规范”，颇有道家“无为就是有为”的韵味，它是命名界的另类。比如：

```

unsigned char i; //超越了规则约束的循环体变量，用单个字母来表示。
unsigned long k; //超越了规则约束的循环体变量，用单个字母来表示。
void HanShu(unsigned char u8Data) //u8 就代表局部的 8 位变量
{
    unsigned int c; //超越了规则约束的循环体变量，用单个字母来表示。
    for(c=0;c<5;c++) //用在循环体的变量
    {
        u8Data=u8Data+1; //u8 就代表局部的 8 位变量
    }

    for(i=0;i<5;i++) //用在循环体的变量
    {
        u8Data=u8Data+1; //u8 就代表局部的 8 位变量
    }

    for(k=0;k<5;k++) //用在循环体的变量
    {
        u8Data=u8Data+1; //u8 就代表局部的 8 位变量
    }
}

```

### 【79.4 数组的命名规范和习惯。】

数组有四种应用场合，一种是普通数组，一种是字符串，一种是表格，一种是信息。在命名上分别加入后缀“Buffer,String,Table,Message”来区分，但是它们都是数组。比如：

```

unsigned int  Gu16NumberBuffer[5]; //后缀是 Buffer。16 位的全局变量数组。用在普通数组。
unsigned char Gu8NumberString[5];  //后缀是 String。8 位的全局变量数组。用在字符串。

//根据原理图得出的共阴数码管字模表
code unsigned char Cu8DigTable[]={//后缀是 Table。这里的 code 是代表 C51 的常量(类似 const)。
{
0x3f,  //0      序号 0
0x06,  //1      序号 1
0x5b,  //2      序号 2
0x4f,  //3      序号 3
0x66,  //4      序号 4
0x6d,  //5      序号 5
0x7d,  //6      序号 6
0x07,  //7      序号 7
0x7f,  //8      序号 8
0x6f,  //9      序号 9
0x00,  //不显示 序号 10
};

void HanShu(unsigned char u8Data) //u8 就代表局部的 8 位变量
{
    unsigned char u8NumberMessage[5]; //后缀是 Message。8 位的局部变量数组。用在信息。
}

```

### 【79.5 指针的命名规范和习惯。】

指针的前缀加“p”来区分。再往下细分，指针有全局和局部，有“静态”和“非静态”，有“8 位宽度”和“16 位宽度”和“32 位宽度”，有变量指针和常量指针。比如：

```

unsigned char *pGu8NumberString; //pGu8 代表全局的 8 位变量指针
void HanShu(const unsigned char *pCu8Data) //pCu8 代表局部的 8 位常量指针
{
    unsigned char *pu8NumberBuffer;          //pu8 代表局部的 8 位变量指针
    static unsigned int *pSu16NumberBuffer;    //pSu16 代表局部的 16 位静态变量指针
    static unsigned long *pSu32NumberBuffer;   //pSu32 代表局部的 32 位静态变量指针
}

```

### 【79.6 结构体的命名规范和习惯。】

结构体的前缀加“t”来区分。再往下细分，指针有全局和局部，有“静态”和“非静态”，有结构体变量和结构体指针。比如：

```

struct StructSignData //带符号的数
{

```

```

    unsigned char  u8Sign; //符号 0 为正数 1 为负数
    unsigned long  u32Data; //数值
};

struct StructSignData GtNumber; //Gt 代表全局的结构体变量。
void HanShu(struct StructSignData *ptData) //pt 代表局部的结构体指针
{
    struct StructSignData tNumber; //t 代表局部的结构体变量。
    static struct StructSignData StNumber; //St 代表局部的静态结构体变量。
}

```

### 【79.7 宏常量的命名规范和习惯。】

所谓“宏常量”往往是指用#define 语句定义的常量。宏常量的所有字符都用大写字母。比如：

```

#define DELAY_TIME 30 //宏常量所有字符都用大写字母。DELAY_TIME 代表延时的时间。
void HanShu(void)
{
    delay(DELAY_TIME); //相当于 delay(30), 这里的 delay 代表某个延时函数(这里没有具体写出来)
}

```

### 【79.8 首字符用大写字母以及下划线“\_”的灵活运用。】

两个以上的英文单词连在一起命名时，每个单词的首字符用大写，其余用小写，这样可以把每个单词“断句”开来，方便阅读。如果遇到两个英文单词连在一起不好“断句”的情况（比如某个英文单词全部是大写字母的专用名词），只要在两个英文单词之间插入下划线“\_”就可以清晰的“断句”了。比如：

```

unsigned long Gu32GetFileLength; //GetFileLength 寓意“获取某个文件的长度”。
unsigned char Gu8ESD_Flag; //ESD 是专业用名词，代表“静电释放”的意思。用下划线“_”断句。

```

## 第八十节： 单片机 IO 口驱动 LED。

### 【80.1 不再依赖第 11 节模板程序。】

前面大量的章节主要是讲 C 语言本身的基础知识，因此每次的练习例程都要依赖第 11 节的模板程序。从本节开始，正式进入到单片机主题，如果没有特殊说明，以后的练习程序就不再需要依赖第 11 节模板程序，可以脱离模板单飞了。

### 【80.2 寄存器。】

寄存器是跨越在软件与硬件之间的桥梁，单片机的 C 语言想控制单片机引脚输出 0V 或者 5V 的物理电压，本质就是通过往寄存器里填数字，往哪个寄存器填数字，填什么样的数字，对应的引脚就输出什么样的电压。至于“为什么往寄存器填数字就会在引脚上输出对应的电压”这个问题，对于我们“应用级”工程师来说是一个黑匣子。我们写软件的最底层就是操作到“寄存器”这个层面，至于“寄存器与物理电压之间是如何关联如何实现”这个问题，其实是“芯片级”半导体工程师所研究的事，因为单片机本身其实就是一个成品，我们从“芯片级”半导体工程师那里拿到这个成品，这个成品的说明书告诉了我们该成品的每个寄存器的作用，我们只能在这个基础上去做更上层的应用。该说明书其实就是大家通常所说的芯片的 datasheet。

寄存器在单片机 C 语言层面，是一个全局变量，是一个具备特定名字的全局变量，是一个被系统征用的全局变量。寄存器的名字就像古代皇帝的名字，所有普通老百姓的变量名字都要“避尊者讳”，不能跟寄存器的名字重名，否则 C 编译器就编译不通过。

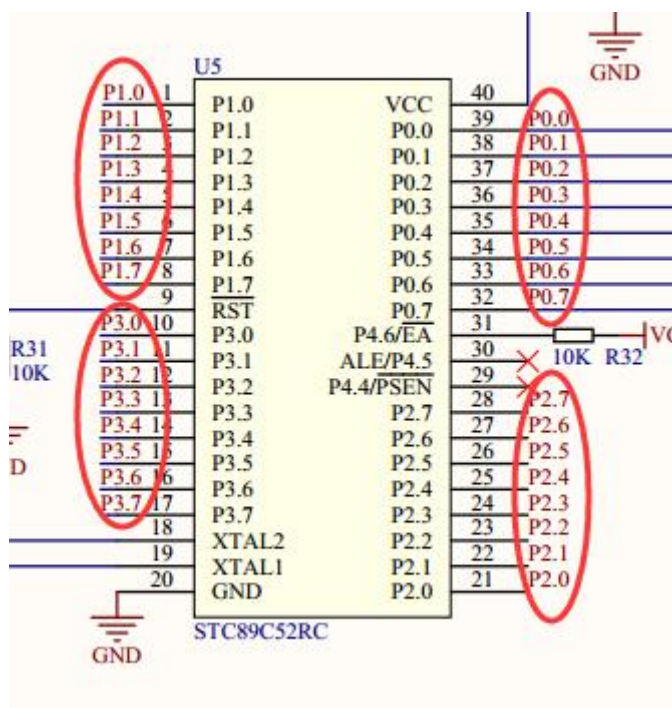


图 80.2.1 单片机的 32 个 IO 口引脚

本教程用的 STC89C52 单片机 IO 口寄存器有 4 个，分别是 P0, P1, P2, P3 这 4 个寄存器，每个寄存器都是一个 8 位的全局变量，每一位代表控制一个单片机的 IO 口引脚，因此，该单片机一共有 32 个（4 乘以 8）IO 口引脚，每个引脚都是可以单独控制的（俗称位操作）。往该位填入 0，对应的引脚就输出 0V 的物理电压。



往该位填入 1，对应的引脚就输出 5V 的物理电压。

### 【80.3 C 语言操作 IO 口寄存器。】

C 语言操作单片机 IO 口寄存器，以便在对应的引脚上输出对应的物理电压，有两种方式。一种是并口的方式，另外一种为位操作的方式。并口方式，一次操作 8 个位（8 个引脚），往往用在并口数据总线上。位操作方式，一次操作 1 个位（1 个引脚），该方式因为单独控制到某个引脚，所以应用更加灵活广泛。

并口方式。并口方式的时候，可以直接对 P0, P1, P2, P3 这 4 个寄存器赋值，就像对一个 unsigned char 的全局变量赋值一样。比如：

```
#include "REG52.H"
void main()
{
    P0=0xF0; //直接对 P0 赋值 0xF0，意味着 P0 口的 8 个引脚，高 4 位全部输出 5V，低 4 位全部输出 0V。
    while(1)
    {
    }
}
```

“P0=0xF0”这行代码，把十六进制的 0xF0 分解成二进制 11110000 来理解，P0.7，P0.6，P0.5，P0.4 这 4 个引脚分别输出 5V 物理电压，而 P0.3，P0.2，P0.1，P0.0 这 4 个引脚分别输出 0V 物理电压。

位操作方式。并口方式因为一次操作就绑定了 8 个引脚，是非常不方便的，因此，位操作就显得特别灵活实用，你可以直接操作 P0，P1，P2，P3 这 4 组引脚中（共 32 个）的某 1 个引脚，而不影响其它引脚的状态。比如，P1.4 引脚是属于 P1 组的 8 个引脚中的某 1 个引脚，如果想直接位操作 P1.4 引脚，要用到特定的关键词 sbit 和符号“^”这个组合，sbit 和符号“^”的组合类似宏定义，使用方式如下。

```
#include "REG52.H"
sbit P1_4=P1^4; //利用 sbit 和符号“^”的组合，把变量名字 P1_4 与 P1.4 引脚关联起来
void main()
{
    P1_4=0; //P1.4 引脚输出 0V 物理电压，而不影响其它 P1 口引脚的状态。
    while(1)
    {
    }
}
```

### 【80.4 点亮 LED。】

LED 灯要有电流通过，才会发光。要有电流通过，必须要有电压的“正压差”，“压差”可以用水压来比喻。

比如在 2 楼的水，对于 1 楼来说，它就有“正压差”（2 减去 1 等于“正 1”），因此只要构成回路（有水管），2 楼的水是可以往 1 楼流动的。

比如在 2 楼的水，对于 3 楼来说，它虽然有压差，但是有的只是“负压差”（2 减去 3 等于“负 1”），因此哪怕构成回路（有水管），2 楼的水也是不可以往 3 楼流动的。

比如在 2 楼的水，对于同楼层的 2 楼来说，它的压差是 0 压差（2 减去 2 等于“0 压差”），因此哪怕构成回路（有水管），2 楼的水也是不可以在 2 楼之间流动的。

上面三个比喻很关键，精髓在于是否有“正压差”。要点亮一个 LED 灯，并不是说你单片机引脚直接输出一个 5V 的物理电压就能点亮的，还要看它构成的整个 LED 灯回路，也就是实际的电路图是什么样的。在本教程的原理图中，我们点亮 LED 灯是采样“灌入式”的电路，也就是单片机输出 5V 电压的时候 LED 灯是熄灭的，而输出 0V 物理电压时 LED 灯反而是被点亮的。如下两个图：

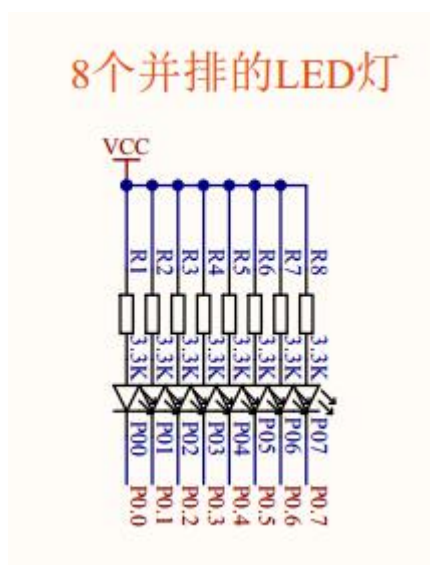


图 80. 4. 1 灌入式驱动 8 个 LED

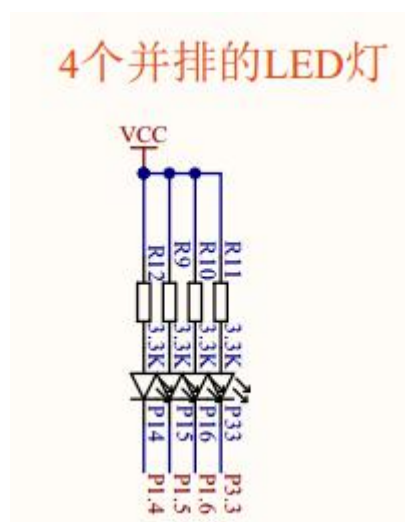


图 80. 4. 2 灌入式驱动 4 个 LED

现在根据这原理图，编写一个并口和位操作的练习例子，直接把程序烧录进开发板，就可以看到对应的 LED 灯的状态。

```
#include "REG52.H"
sbit P1_4=P1^4; //利用 sbit 和符号“^”的组合，把变量名字 P1_4 与 P1.4 引脚关联起来
void main()
{
    P0=0xF0; //直接对 P0 赋值 0xF0，意味着 P0 口的 8 个引脚，高 4 位全部输出 5V，低 4 位全部输出 0V。
    P1_4=0; //P1.4 引脚输出 0V 物理电压，而不影响其它 P1 口引脚的状态。
    while(1)
    {
    }
}
```

现象分析：

“P0=0xF0”直接对 P0 赋值 0xF0，意味着 P0 口的 8 个引脚，高 4 位全部输出 5V（LED 灯反而灭），低 4 位全部输出 0V（LED 灯反而被点亮）。

“P1\_4=0”P1.4 引脚输出 0V 物理电压（LED 灯反而被点亮）。

## 第八十一节： 时间和速度的起源（指令周期和晶振频率）。

### 【81.1 节拍。】

单片机的 C 语言经过 C 编译器后，翻译成很多条机器指令，单片机逐条执行这些指令，每执行一条指令都是按照固定的节奏进行的，两条指令之间是存在几乎固定的时间间隔（实际上不是所有指令的间隔时间都绝对一致，这里方便理解暂时看作是一致），这就是节拍，每个节拍之间的时间间隔其实就是指令周期，因此，指令周期越短，节拍就越短，单片机的运算速度就越快。指令周期是由什么决定的呢？指令周期是由“心跳速度”和“心跳个数”决定的。指令周期都是由固定的 N 个“心跳个数”组成的，指令周期到底由多少个“心跳个数”组成？每种单片机每类指令各不一样。我们用的 51 系列单片机，最短的单周期指令是由 12 个“心跳个数”组成，依次类推，双周期指令由 24 个“心跳个数”组成，4 周期指令由 48 个“心跳个数”组成。但是光有“心跳个数”还不够，还必须搭配知道“心跳速度”才能最终计算出指令周期。这里的“心跳速度”就是晶振的频率，“心跳个数”就是累计晶振的起振次数。比如，假设我们用的 51 单片机是 12MHz（本教程实际用的是 11.0592MHz），那么每个单周期的指令执行的时间是： $12 \times (1/12000000)$  秒 = 1 微秒。这个公式左边的“12”代表“12 个晶振起振的次数”，这个公式右边的“(1/12000000)”代表晶振每起振 1 次所需要的单位时间。二者结合，刚好就是“心跳个数”乘以“单个心跳周期”等于指令周期，而指令周期就是节拍的时间。

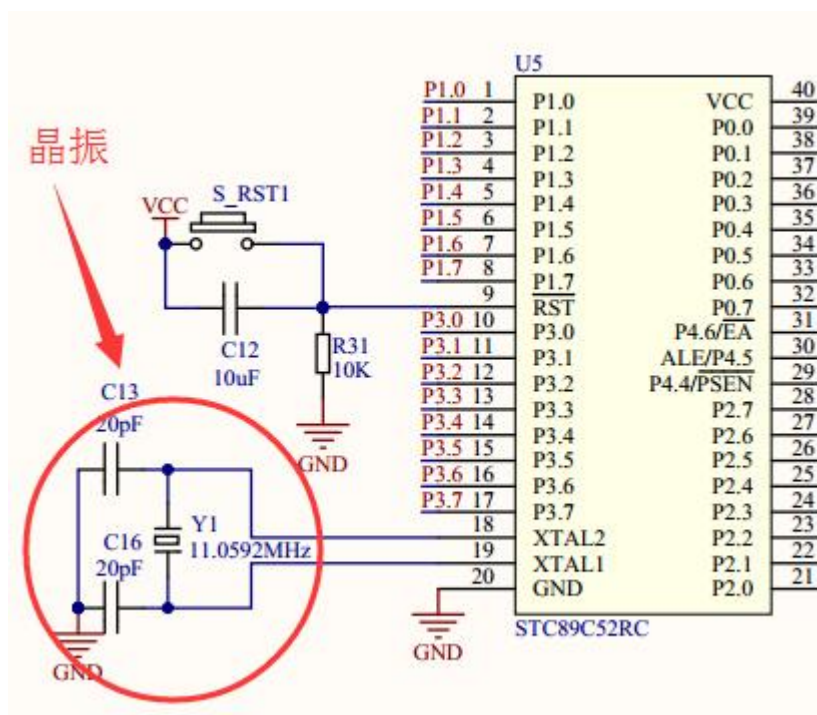


图 81.1.1 单片机的晶振

### 【81.2 累计节拍次数产生延时时间。】

有了这个最原始的“节拍”概念，现在开始编写一个练习程序，让一个 LED 灯闪烁，闪烁的本质，就是让一个 LED 灯先亮一会（“一会”就是延时），然后紧接着让 LED 灯熄灭一会（“一会”就是延时），依次循环，在视觉上看到的连贯动作就是 LED 闪烁。这里的关键是如何产生这个“一会”的延时，本节教程所用的就是一个 for 循环来执行 N 条空指令，每执行一条空指令就需要消耗掉 1 个左右的指令周期的时间（大概 1 微秒左右），空指令执行的循环次数越多，产生的延时时间就越长。例子如下：

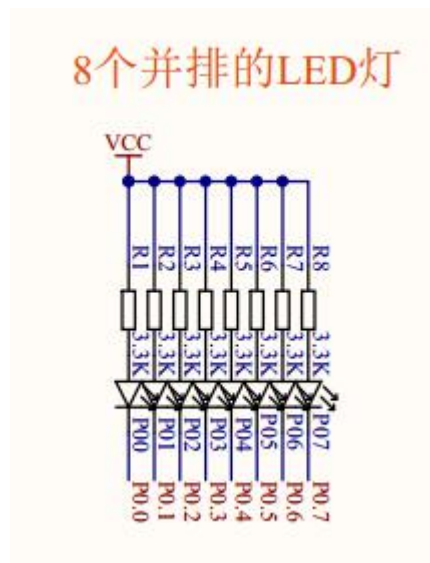


图 81.2.1 灌入式驱动 8 个 LED

```
#include "REG52.H"
sbit P0_0=P0^0; //利用 sbit 和符号“^”的组合，把变量名字 P0_0 与 P0.0 引脚关联起来
unsigned long i; //for 循环用的累计变量
//unsigned int i; //如果把 for 循环的变量 i 改成 unsigned int 类型，闪烁的频率会加快。
void main()
{
    while(1)
    {
        //第（1）步
        P0_0=0; //LED 灯亮。

        //第（2）步
        for(i=0;i<5000;i++) //累计的循环次数越大，这里的延时就越长，“亮”持续的时间就越长。
        {
            ; //分号代表一条空指令
        }

        //第（3）步
        P0_0=1; //LED 灯灭。

        //第（4）步
        for(i=0;i<5000;i++) //累计的循环次数越大，这里的延时就越长，“灭”持续的时间就越长。
        {
            ; //分号代表一条空指令
        }

        //第（5）步：这里已经触碰到主循环 while(1)的“底线”，所以接着跳转到第（1）步继续循环
    }
}
```

```
}  
}
```

#### 现象分析：

理论上，每执行 1 条指令大概 1 微秒左右，但是实际上，我们看到的实验现象，发现累计循环才 5000 次，按理论计算，应该产生 0.005 秒左右的延时才合理，但是实际上居然能产生类似 0.5 秒的闪烁效果，中间相差 100 倍！为什么？C 语言跟机器指令之间是存在翻译的“中间商”环节，一条 C 指令并不代表一条机器指令，往往一条 C 指令翻译后产生 N 条机器指令，比如上面的代码，用到 for 循环变量 i，用的是 unsigned long 变量，意味着 4 个字节，即使一条 C 语言赋值指令估计可能也要消耗 4 条单周期指令，在加上 for 循环的判断指令，和累加指令，以及跳转指令，所以我们看到的 for(i=0;i<5000;i++)并不代表是真正仅仅执行了 5000 个指令周期，而是有可能执行了 500000 条指令周期！假如我们把上述代码中的 i 改成 unsigned int 变量（2 字节），是会看到闪烁的速度明显加快的，其中原因就是 C 编译器与机器指令之间存在翻译后的“1 对 N”的关系。

## 第八十二节： Delay “阻塞” 延时控制 LED 闪烁。

### 【82.1 “阻塞” 与 “非阻塞”。】

做项目写程序，大框架大思路就是在“阻塞”与“非阻塞”这两种模式下不断切换。“阻塞”可以理解成“单任务处理”模式，“非阻塞”可以理解成“多任务并行处理”模式。“阻塞”的优点是它全神贯注不分心地专注于当下这一件事，它等待某个事件的响应速度是最快的，同时省去了“来回切换、反复扫描”的额外开销，而且在编程思路不用太费脑力只需“记流水账式”的编程即可，但是它的缺点是当下只能干一件事，其它事情无法兼顾，做不到多任务并行处理。而“非阻塞”恰恰相反，它的有优点就是“阻塞”的缺点，它的缺点就是“阻塞”的优点，对于“非阻塞”本节暂时不多讲。在实际项目中，有时候“大阻塞”中分支了N个“小非阻塞”，也有时候“大非阻塞”中分支了N个“小阻塞”。能在“阻塞”与“非阻塞”之间运用自如者，谓之神。

“阻塞等待”是指单片机在某个死循环里（比如“while(1)”这类）一直不断循环地在等待某个标志变量的状态，如果这个标志变量满足条件才会跳出这个死循环然后才能干其它的事情，否则一直在死循环里死等，给人一种全神贯注心无旁骛的感觉，

“阻塞延时”是指单片机在产生“延时时间”的时候做不了别的事，延时多久它就要被“阻塞”多久，只有延时过后它才能解脱去干别的事。比如，在编程上，常用 for 循环产生N个空指令来达到产生“延时时间”的目的，这种编程方式就是最常见的“阻塞延时”。

### 【82.2 Delay 阻塞延时的一个例子。】

现在利用“Delay 阻塞延时”编写一个练习程序，让一个 LED 灯闪烁。例子如下：

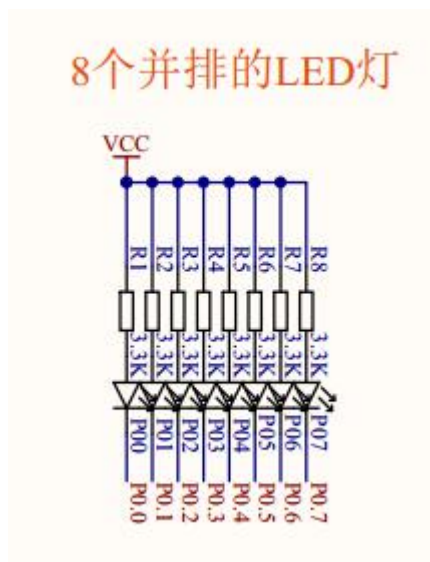


图 82.2.1 灌入式驱动 8 个 LED

```
#include "REG52.H"
void Delay(unsigned long u32DelayTime); //函数的声明
sbit P0_0=P0^0; //利用 sbit 和符号“^”的组合，把变量名字 P0_0 与 P0.0 引脚关联起来
```

```

void Delay(unsigned long u32DelayTime) //产生“阻塞延时”的延时函数
{
    static unsigned long i; //函数在频繁调用时，加 static 可以省去一条额外的初始化语句的开销。
    for(i=0;i<u32DelayTime;i++);
}

void main()
{
    while(1)
    {
        //第（1）步
        P0_0=0; //LED 灯亮。

        //第（2）步
        Delay(5000); //这里就是阻塞延时，时间就越长，“亮”持续的时间就越长。

        //第（3）步
        P0_0=1; //LED 灯灭。

        //第（4）步
        Delay(5000); //这里就是阻塞延时，时间就越长，“灭”持续的时间就越长。

        //第（5）步：这里已经触碰到主循环 while(1)的“底线”，所以接着跳转到第（1）步继续循环
    }
}

```

### 【82.3 累加型和累减型的两种 Delay 函数，哪家强？】

上述 82.2 例子中，用到一个 Delay 函数，该函数内部的 for 循环用的是“累加型”的，比如：

```

void Delay(unsigned long u32DelayTime)
{
    static unsigned long i; // “累加型”函数内部多开销了一个变量 i。
    for(i=0;i<u32DelayTime;i++); //因为这里的“i++”是加法运算，所以称为“累加型”。
}

```

现在在跟大家分享一种“累减型”的 Delay 函数，例子如下：

```

void Delay(unsigned long u32DelayTime)
{
    // “累减型”函数内部节省了一个变量 i。
    for(;u32DelayTime>0;u32DelayTime--); // “u32DelayTime--”意味着“累减型”。
}

```

仔细对比“累加型”和“累减型”，会发现在实现同样“阻塞延时”的功能下，因为“累减型”巧妙的



借用了函数入口的局部变量 u32DelayTime 来充当 for 循环的变量，而省去了一个 i 变量。因此，“累减型”比“累加型”强一点。

#### 【82.4 Delay 函数让初学者容易犯的错误。】

初学者刚接触 Delay 函数，常常容易犯的错误就是忽略了 for 循环变量的类型，for 循环变量的类型决定了你能输入的数值范围，比如上面例子中用到的是 unsigned long 变量，因此可以最大输入 Delay(4294967295)。如果是 unsigned int 变量，最大可以输入 Delay(65535)。如果是 unsigned char 变量，最大可以输入 Delay(255)。

#### 【82.5 Delay 内部的 for 循环嵌套可产生无穷长的时间。】

刚才讲到，如果用最大的变量类型 unsigned long，最大的输入是 Delay(4294967295)，那么问题来，难道 Delay 函数的阻塞延时的时间有最大极限？其实不存在最大极限，理论上，你要多大的延时都可以，只需要在 Delay 函数内部用上 for 循环的嵌套，就可以产生“乘法级”的无穷长的时间，例子如下：

```
void Delay(unsigned long u32DelayTime)
{
    static unsigned long i;
    static unsigned long k;
    for(i=0;i<u32DelayTime;i++)
    {
        for(k=0;k<5000;k++); //内部嵌套的 for 循环，意味着乘法的关系 u32DelayTime 的 5000 倍！
    }
}
```

#### 【82.6 “阻塞延时”与“非阻塞延时”的各自应用范围。】

“阻塞延时”一般应用在两个地方，一个是上电初始化进入主循环之前的延时，另一个是进入主循环之后，跟外部驱动芯片通信时候产生的时钟节拍小延时，而这个类延时一般是低于 1ms 的小延时。

“非阻塞延时”在项目中是被大量应用的，进入主循环之后，只要大于或等于 1ms 的延时，大多数都采样“非阻塞延时”，因为进入“任务框架级”的层面，只有“非阻塞延时”才能保证项目可以继续“多任务并行处理”。“非阻塞延时”的方式后续章节会讲到。

综上所述，1ms 是“阻塞延时”与“非阻塞延时”的一个分界线，1ms 这个时间不是绝对的，只是一个经验值。

## 第八十三节： 累计主循环的“非阻塞”延时控制 LED 闪烁。

### 【83.1 累计主循环的“非阻塞”。】

上一节提到，当 Delay 的“阻塞”时间超过 1ms 并且被频繁调用的时候，由于 Delay 做“独占式无用功”而消耗的延时太长，会影响其它任务的并行处理，整个系统给人的感觉非常卡顿不流畅。为了解决此问题，本节引入累计主循环的“非阻塞”，同时，希望通过此例子，让大家第一次感受到 switch 语句在多任务并行处理时候的优越性。switch 的精髓在于“根据某个判断条件实现步骤之间的灵活跳转”，这个思路是以后做所有大项目的框架性思路。

为什么“累计主循环”可以兼顾到其它任务的并行处理？因为单片机进入 main 函数以后，在一个主循环里要扫描 N 个任务，从头到尾，把 N 个任务扫描一遍，每扫描一遍算“一次主循环”，每一次“主循环”都是要消耗一点时间，累计的“主循环”次数越多，所要消耗的时间就越长，但是跟 Delay 唯一的差别是，Delay 做延时的时候没有办法扫描其它任务，而“累计主循环”内部本身就是在不断扫描其它任务，产生时间越长扫描其它任务的次数就越多，两者是完全相互促进而没有矛盾的。具体内容，请看下面的例子。

### 【83.2 累计主循环“非阻塞”的一个例子。】

现在利用“累计主循环非阻塞”编写一个练习程序，让一个 LED 灯闪烁。例子如下：

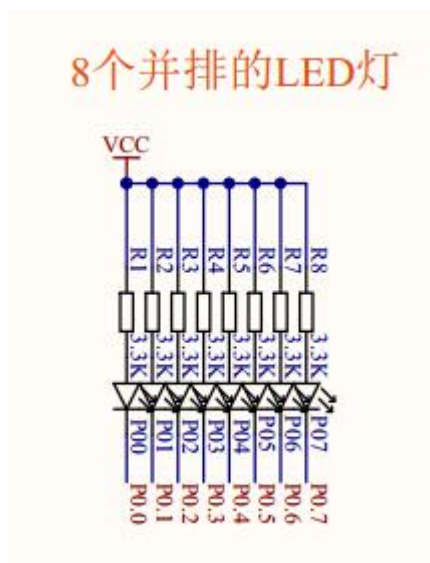


图 83.2.1 灌入式驱动 8 个 LED

```
#include "REG52.H"

#define CYCLE_SUM 5000 //累计主循环次数的设定阈值，该值决定了 LED 闪烁频率

sbit P0_0=P0^0; //利用 sbit 和符号“^”的组合，把变量名字 P0_0 与 P0.0 引脚关联起来

unsigned char Gu8CycleStep=0; //switch 的跳转步骤
unsigned long Gu32CycleCnt=0; //累计主循环的计数器
```

```

void main()
{
    while(1)
    {
        switch(Gu8CycleStep)
        {
            case 0:
                Gu32CycleCnt++;    //这里就是累计 main 函数内部的主循环 while(1)的次数
                if(Gu32CycleCnt>=CYCLE_SUM) //累计的次数达到设定值 CYCLE_SUM 就跳到下一步骤
                {
                    Gu32CycleCnt=0;    //及时清零计数器，为下一步骤的新一轮计数准备
                    P0_0=0;    //LED 灯亮。
                    Gu8CycleStep=1;    //跳到下一步骤
                }
                break;

            case 1:
                Gu32CycleCnt++;    //这里就是累计 main 函数内部的主循环 while(1)的次数
                if(Gu32CycleCnt>=CYCLE_SUM) //累计的次数达到设定值 CYCLE_SUM 就返回上一步骤
                {
                    Gu32CycleCnt=0;    //及时清零计数器，为返回上一步骤的新一轮计数准备
                    P0_0=1;    //LED 灯灭。
                    Gu8CycleStep=0;    //返回到上一个步骤
                }
                break;
        }
    }
}

```

### 【83.3 累计主循环的不足。】

上述 83.2 例子中，“累计主循环次数”实现时间延时是一个不错的选择。这种方法能胜任多任务处理的程序框架，但是本身也有一个小小的不足，比如“阈值 CYCLE\_SUM 到底应该取多少才能产生多长的时间”是没有标准的，只能依靠不断上机实验来拿到一个你所需要的数值，这种“不规范”，当程序要移植到其它单片机平台上的时候就特别麻烦，需要重新修正阈值 CYCLE\_SUM。除此之外，哪怕在同样的一个单片机里，随着主函数里任务量的增加，累计一次主循环所消耗的时间长度也会发生变化，意味着靠“累计主循环次数”所获得的时间也会发生变化而导致不准确，此时，为了保证延时时间的准确性，必须要做的就是再一次修正“设定累计主循环次数”的阈值 CYCLE\_SUM，这样显然给我们带来了一丝不便，怎么办？假设单片机没有“定时中断”这个资源，那么这种“累计主循环次数”在多任务处理中确实是不二之选，但是，因为现在几乎所有的单片机内部都有“定时中断”这个资源，所以，大家不用为这个“不足”而烦恼，我们只要用上本节的 switch 思路，再外加一个“定时中断”，就可以轻松解决此问题，下一节就跟大家讲“定时中断”的内容。

## 第八十四节： 中断与中断函数。

### 【84.1 中断。】

单片机的“中断”跟日常生活的“中断”差不多，我正在做“常事”的时候，突然遇到优先级更高的“急事”，这时我必须先暂停手上的“常事”，马上去处理突如其来的“急事”，处理完“急事”再返回来继续做“常事”。要理解单片机的“中断”，有六个关键点，第一点是“配置中断”，第二点是“做常事”，第三点是“中断请求”，第四点是“保护中断现场”，第五点是“处理急事”，第六点是“返回中断现场”。举一个我生活的例子如下：

第一点：我老婆随时都会打电话给我，所以我的手机 24 小时都打开处于待机的状态。（配置中断）

第二点：我正在读一本书《道德经》（做常事）。

第三点：当我读到第 18 页的时候，老婆突然给我打电话，让我去幼儿园帮接一下小孩（中断请求）。

第四点：我在第 18 页里夹了一张书签做标记（保护中断现场）。

第五点：我放下手上的书去幼儿园接小孩（处理急事）。

第六点：接了小孩，我回来继续打开《道德经》，找到书签标记的第 18 页（返回中断现场），继续阅读。

上述六点，在单片机的 C 语言里，“配置中断”放在主函数的初始化那里，“做常事”放在主函数的主循环里（main 函数内部的 while(1) 循环），“中断请求”单片机内部硬件检测到符合发生中断的条件，“保护中断现场”是单片机内部硬件电路自动处理的（不需要我们软件干涉），“处理急事”是单片机自动跳转到另外开辟的一个特殊中断函数处理（自动跳转是单片机的硬件自动完成不需要我们软件干涉），执行完一次中断函数后单片机再自动跳转到主函数的主循环的现场点继续从现场点开始继续做常事（返回中断现场）。在这六点中，其中第四点的“保护中断现场”与第六点的“返回中断现场”是要特别强调的，单片机从 main 函数的主循环 while(1) 准备跳转到中断函数之前，它会自动记录当前的位置（做好路标），以便处理完中断函数后再返回 main 函数的主循环 while(1) 时，能找到之前的被中断跳转前的位置，这样就可以接上原来的步骤去处理原来的“常事”，在步骤上既不提前也不滞后恰到好处，中断就不会影响到常事的完整性。代码分布图的模板描述如下：

```
void main()
{
    配置中断;
    while(1)
    {
        处理常事;
    }
}

void 中断函数() interrupt 序号    //中断函数后缀带“interrupt 序号”特别修饰
{
    急事;
}
```

奇怪！上述代码，为什么“main 函数”与“中断函数”在软件上看不到任何关联，既不存在“main 函数”调用“中断函数”，也不存在“中断函数”调用“main 函数”的情况，在观感上，“main 函数”与“中

断函数”仿佛是隔离的毫无“物理连接”的，为什么单片机还能在“main 函数”与“中断函数”两者中切换自如？没错，确实，“main 函数”与“中断函数”在书写上是隔离的毫无关联的，但是它们之间之所以能相互切换，是因为背后有一只无形的手在自动操控这一切，这只手就是单片机硬件自身，这是一种特殊机制，也可以理解成一种特殊的游戏规则，我们只要遵守就好了，除了普通函数，其它凡是中断函数的，都不用跟 main 函数发生软件上的关联调用，它们之间的切换都是硬件自动完成的，这就是 main 函数与中断函数的特殊跳转机制（或者称为游戏规则也可以）。

## 【84.2 常用的中断函数有哪三类？】

单片机的中断有很多，但常用在项目上的有三类：

第一类是定时中断。配置中断后，使其每隔一段时间就产生一次中断，比如“1ms 一次的定时中断”几乎是所有的系统里的标配，因为它对程序框架起到一个时间节拍的作用。

第二类是通讯中断。比如串口每接收完一个字节就会产生一个中断通知我们去做处理。

第三类是电平变化的中断。下降沿或者上升沿的中断，常常用在采集高速的脉冲信号。

## 【84.3 我们如何操控中断？】

刚才 84.1 提到“单片机硬件自动”这个概念，但是说它“硬件自动”并不意味着它不可控。单片机本身出厂的时候内部就携带了很多种类的中断，这些中断是否开启取决于你的“配置中断”代码，你要开启或者关闭某类中断，只需编写对应的“配置中断”代码就可以，而“配置中断”的代码本质就是填写某些寄存器数值。

## 第八十五节： 定时中断的寄存器配置。

### 【85.1 寄存器配置的本质。】

单片机内部可供我们选择的资源非常丰富，有定时器，有串口，有外部中断，等等。这些丰富的资源，就像你进入一家超市，你只需选择你所需要的东西就可以了，所以配置寄存器的关键在于选择，所谓选择就是往寄存器里面做填空题，单片机系统内部再根据你的“选择清单”，去启动对应的资源。那么我们怎么知道某个型号的单片机内部有哪些资源呢？看该型号“单片机的说明书”呀，“单片机的说明书”就是我们通常所说的“芯片的 datasheet”，或者说是“芯片的数据手册”，这些资料单片机厂家会提供的。

跟单片机打交道，其实跟人打交道没什么区别，你要让单片机按照你的“意愿”走，你首先要把你的“意愿”表达清楚，这个“意愿”就是信息，信息要具备准确性和唯一性，不能模棱两可。比如，现在要让单片机“每 1ms 产生一次中断”，你想想你可能需要给单片机提供哪些信息？

(1) 51 单片机有 2 个定时器，一个是 0 号定时器，一个是 1 号定时器，我们要面临“二选一”的选择，本例子中用的是“0 号定时器”。

(2) 0 号定时器内部又有 4 种工作方式：方式 0，方式 1，方式 2，方式 3，本例子中用的是“方式 1”。

(3) 定时器到底多长时间中断一次，这就涉及到填充与中断时间有关的寄存器的数值，该数值是跟时间成比例关系，本例子中配置的是 1ms 中断，就要填充对应的数值。

(4) 默认状态下，定时器是不会被开启的，如果要开启，这里就是涉及到定时器的“开关”，本例子要开启此开关。

(5) 定时器时间到了就要产生中断，中断也有“总开关”和“定时器的局部开关”，这两个开关都必须同时打开，中断才会有效。

要配置定时器“每 1ms 产生一次中断”，大概就上述这些信息，根据这些信息提示，下面开始讲解一下寄存器的具体内容。

### 【85.2 定时器/计数器的模式控制寄存器 TMOD。】

寄存器 TMOD 是一个 8 位的特殊变量，里面每一位都代表了不同的功能选择。根据芯片的说明书，TMOD 的 8 位从左到右依次对应从 D7 到 D0（左高位，右低位），定义如下：

GATE	C/T	M1	M0	GATE	C/T	M1	M0
------	-----	----	----	------	-----	----	----

仔细观察，发现左 4 位与右 4 位是对称的，分别都是“GATE, C/T, M1, M0”，左 4 位控制的是“定时器 1”，右 4 位控制的是“定时器 0”，因为本例子用的是“定时器 0”，因此“定时器 1”的左 4 位都设置为 0 的默认数值，我们只需重点关注右 4 位的“定时器 0”即可。

GATE: 定时器是否受“其它外部开关”的影响的标志位。定时器的开启或者停止，受到两个开关的影响，第一个开关是“自身原配开关”，第二个开关是“其它外部开关”。GATE 取 1 代表定时器受“其它外部开关”的影响，取 0 代表定时器不受“其它外部开关”的影响。本例子中，定时器只受到“自身原配开关”的影响，而不受到“其它外部开关”的影响，因此，GATE 取 0。

C/T: 定时器有两种模式，该位取 1 代表“计数器模式”，取 0 代表“定时器模式”。本例子是“定时器模式”，因此，C/T 取 0。

M1 与 M0: 工作方式的选择。M1 与 M0 这两位的 01 搭配，可以有 4 种组合（00, 01, 10, 11），每一种组合就代表一种工作方式。本例子选用“方式 1”，因此 M1 与 M0 取“01”的组合。

综上所述，TMOD 的配置代码是：TMOD=0x01;

### 【85.3 决定时间长度的寄存器 TH0 与 TL0。】

TH0 与 TL0, T 代表定时器英文单词 TIME 的 T, H 代表高位, L 代表低位, 0 代表定时器 0。

TH0 是一个 8 位宽度的寄存器, TL0 也是一个 8 位宽度的寄存器, 两者合并起来成为一个整体, 实际上就是一个 16 位宽度的寄存器, TH0 是高 8 位, TL0 是低 8 位, 它们合并后的数值范围是: 0 到 65535。该 16 位寄存器取值越大, 定时中断一次的时间反倒越小, 为什么? TH0 与 TL0 的初始值, 就像一个水桶里装的水。如果这个桶是空桶 (取值为 0), “雨水”想把这个桶“滴满溢出”所需要的时间就很大。如果里面已经装了大半的水 (取值为大于 32767), “雨水”想把这个桶“滴满溢出”所需要的时间就很小。这里的关键词“滴满溢出”的“滴”与“满溢出”, “滴”的速度是由单片机晶振决定的, 而“满溢出”一次就代表产生一次中断, 执行完中断函数在即将返回主函数之前, 我们重新装入特定容量的水 (重装初值), 为下一次的“滴满溢出”做准备, 依次循环, 从而连续不断地产生间歇的定时中断。

配置中断时间的大小是需要经验的, 因为, 每次定时中断的时间太长, 就意味着时间的可分度太粗, 而如果每次定时中断的时间太短, 则会产生很频繁的中断, 势必会影响主函数 main() 的执行效率, 而且累记中断次数的时间误差也会增大。因此, 配置中断时间是需要经验的, 根据经验, 定时中断取 1ms 一次, 是几乎所有单片机项目的最佳选择, 按我的理解, “1ms 定时中断一次”已经是单片机界公认的一种“标配”。

要配置 1ms 定时中断, TH0 与 TL0 如何取值? 刚才提到一个形象的例子“桶, 滴, 满溢出”。TH0 与 TL0 的最大取值范围是 65535, 可以理解成为最大 65535“滴”, 如果超过 65535“滴” (比如加 1“滴”后变成 65536“滴”)就会“满溢出”, 从而产生一次中断 (65536 是中断发生的临界值)。而“滴一次的时间”就刚好是单片机执行“一次单指令的时间”, “一次单指令的时间”等于 12 个晶振周期, 比如 12MHz 的晶振, 晶振周期是 (1/12000000) 秒, 而“一次单指令的时间”就等于 12 乘以 (1/12000000) 秒, 等于 0.000001 秒, 也就是 1us。1us“滴”一次, 要产生 1ms 的时间就需要“滴”1000 次。“满溢出”的前提条件是“桶里”一共需要装入 65536 滴才溢出, 因此, 在 12MHz 的晶振下要产生 1ms 的定时中断, TH0 与 TL0 的初值应该是 64536 (65536 减去 1000 等于 64536), 而 64536 变成十六进制 0xfc17, 再分解到高 8 位 TH0 为 0xfc, 低 8 位 TL0 为 0x17。

刚才的例子是假如晶振在 12MHz 的情况下所计算出来的结果, 而本教程所用的晶振是 11.0592MHz, 根据 11.0592MHz 产生 1ms 的定时中断, TH0 与 TL0 应该取值多少? 根据刚才的计算方式:

```
初值=[溢出值]-([0.001 秒]/([晶振周期的 12 个]*([1 秒]/[晶振频率])))  
初值=65536-(0.001/(12*(1/11059200)))  
初值=65536-922      (注: 922 是 921.6 的四舍五入)  
初值=64614  
初值=0xfc66  
初值 TH0=0xfc  
初值 TL0=0x66
```

### 【85.4 中断的总开关 EA 与局部开关 ET0。】

EA: 中断的总开关。宽度是 1 位的位变量。此开关如果取 0, 就会强行屏蔽所有的中断, 因此, 只要用到中断, 此开关必须取 1。

ET0: 专门针对定时器 0 中断的局部开关。宽度是 1 位的位变量。此开关如果取 0, 则会屏蔽定时器 0 的中断, 如果取 1 则允许定时器 0 中断。如果要定时器 0 能产生中断, 那么总开关 EA 与 ET0 必须同时都打开 (都取 1), 两者缺一不可。

### 【85.5 定时器 0 的“自身原配开关” TR0。】

TR0：定时器的“自身原配开关”。宽度是 1 位的位变量。很多初学者会把 EA，ET0，TR0 三者搞不清。定时器可以工作在“查询标志位”和“中断”这两种状态，也就是说在没有中断的情况下定时器也可以单独使用的。TR0 是定时器 0 自身的发动引擎，要不要把这个发动引擎所产生的能量传输到中断的渠道，则取决于中断开关 EA 和 ET0。TR0 是源头开关，EA 是中断总渠道开关，ET0 是中断分支渠道的定时器 0 开关。TR0 取 1 表示启动定时器 0，取 0 表示关闭定时器 0。

### 【85.6 定时器 0 的中断函数的书写格式。】

```
void 函数名() interrupt 1
{
    ... 中断程序内容;
    ... 此处省去若干代码
    ... 中断程序内容;
    ... 最后面的代码，要记得重装 TH0 与 TL0 的初值;
}
```

函数名可以随便取，只要不是编译器已经征用的关键字。这里的 1 是定时器 0 的中断号。不同的中断号代表不同类型的中断，至于哪类中断对应哪个中断号，大家可以查找相关书籍和资料。本节用的定时器 0 处于工作方式 1 的情况下，在即将退出中断之前，需要重装 TH0 与 TL0 的初始值。

### 【85.7 寄存器的名字来源。】

前面讲的寄存器都有固定的名字，而且这些名字都是唯一的，拼写的时候少一个字母或者多一个字母，C 编译器都会报错不让你通过，因此问题来了，初学者刚接触一款单片机的时候，如何知道某个寄存器它特定的唯一的名字？有两个来源。

第一个来源，可以打开 C 编译器的某个头文件（.h 格式）查看这些寄存器的名字。比如 51 单片机可以查看 REG52.H 这个头文件。如何打开 REG52.H 这个文件？在 keil 源代码编辑器界面下，选中上面 REG52.H 这几个字符，在右键弹出的菜单下点击 Open document “REG52.H” 即可。

第二个来源是直接参考一些现成的范例程序，这些范例程序网上很多，有的是原厂提供的，有的是热心网友分享，有的是技术书籍或者学习板开发板厂家提供的。

### 【85.8 如何快速配置寄存器。】

建议一边阅读芯片的数据手册，一边参考一些现成的范例程序，这些范例程序网上很多，有的是原厂提供的，有的是热心网友分享，有的是技术书籍或者学习板开发板厂家提供的。

### 【85.9 练习例程。】

现在编写一个定时中断程序，让两个 LED 灯闪烁，一个是在主函数里用累计主循环次数的方式实现（P0.0 控制），另一个是在定时中断函数里用累计定时中断次数的方式实现（P0.1 控制）。这两个闪烁的 LED 灯，一个在 main 函数，一个是在中断函数，两路任务互不干涉独立运行，并行处理的“雏形”略显出来。



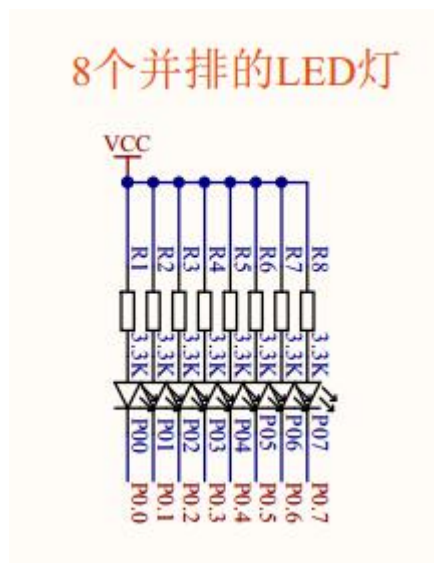


图 85.9.1 灌入式驱动 8 个 LED

```
#include "REG52.H"

#define CYCLE_SUM    5000    //主循环的次数

#define INTERRUPT_SUM    500    //中断的次数

sbit P0_0=P0^0;    //在主循环里的 LED 灯
sbit P0_1=P0^1;    //在定时中断里的 LED 灯

unsigned char Gu8CycleStep=0;
unsigned long Gu32CycleCnt=0;    //累计主循环的计数器

unsigned char Gu8InterruptStep=0;
unsigned long Gu32InterruptCnt=0;    //累计定时中断次数的计数器

void main()
{
    TMOD=0x01;    //设置定时器 0 为工作方式 1
    TH0=0xfc;    //产生 1ms 中断的 TH0 初始值
    TL0=0x66;    //产生 1ms 中断的 TL0 初始值
    EA=1;    //开总中断
    ET0=1;    //允许定时 0 的中断
    TR0=1;    //启动定时 0 的中断

    while(1)    //主循环
    {
        switch(Gu8CycleStep)

```

```

{
    case 0:
        Gu32CycleCnt++;
        if(Gu32CycleCnt>=CYCLE_SUM)
        {
            Gu32CycleCnt=0;
            P0_0=0;  //主循环的 LED 灯亮。
            Gu8CycleStep=1;
        }
        break;

    case 1:
        Gu32CycleCnt++;
        if(Gu32CycleCnt>=CYCLE_SUM)
        {
            Gu32CycleCnt=0;
            P0_0=1;  //主循环的 LED 灯灭。
            Gu8CycleStep=0;
        }
        break;
}
}
}

void TO_time() interrupt 1    //定时器 0 的中断函数，每 1ms 单片机自动执行一次此函数
{
    switch(Gu8InterruptStep)
    {
        case 0:
            Gu32InterruptCnt++;  //累计中断次数的次数
            if(Gu32InterruptCnt>=INTERRUPT_SUM) //次数达到设定值就跳到下一步骤
            {
                Gu32InterruptCnt=0;    //及时清零计数器，为下一步骤的新一轮计数准备
                P0_1=0;  //定时中断的 LED 灯亮。
                Gu8InterruptStep=1;  //跳到下一步骤
            }
            break;

        case 1:
            Gu32InterruptCnt++;  //累计中断次数的次数
            if(Gu32InterruptCnt>=INTERRUPT_SUM) //次数达到设定值就返回上一步骤
            {
                Gu32InterruptCnt=0;    //及时清零计数器，为返回上一步骤的新一轮计数准备

```

```
        P0_1=1;  //定时中断的 LED 灯灭。
        Gu8InterruptStep=0;  //返回到上一个步骤
    }
    break;
}

TH0=0xfc;  //重装初值，不能忘。
TL0=0x66;  //重装初值，不能忘。
}
```

## 第八十六节： 定时中断的“非阻塞”延时控制 LED 闪烁。

### 【86.1 定时中断应用的四大关键词。】

本节主要内容有四大个关键词：1ms，互斥量，volatile，switch。

(1) 1ms。把定时中断设置为 1ms 中断一次，几乎是单片机界公认的“标配”。这个 1ms 是系统时间的节拍来源，有了 1ms “标配”意识，你的程序在不同单片机平台上移植的时候会得心应手运用自如。

(2) 互斥量。“主函数”与“定时中断函数”，本质上是两个独立进程在不断切换并行运行，两个进程之间不断切换，就会涉及到数据的安全保护，数据的安全保护主要是针对多字节的变量，比如 int 类型(2 个字节)，long 类型(4 个字节)。但是单字节的 char 变量不用额外保护，因为“字节”是变量中的最小单位（在不考虑“位”的情况下），这里的“最小单位不可分”就像“原子是最小单位不可分”一样，因此也有很多前辈把“互斥量”称为“原子锁”。为什么要用互斥量？因为，在多个线程同时访问同一个全局变量的时候，如果双方都是“读操作”，则不会出现问题，但是，如果双方都是“既有写操作也有读操作”的情况下，比如，我在主函数里正在修改（写操作）一个 unsigned int 类型的变量，unsigned int 类型的变量占用 2 个字节，在更改数据的时候至少需要 2 条指令，当我刚执行完第 1 条指令还没来得及执行第 2 指令的时候，突然来了一个定时中断，并且在定时中断函数里也对这个变量进行了修改（写操作）并且还进行了读取判断操作，这个瞬间就可能给程序带来了隐患。话说回来，互斥量到底有没有必要，其实还是有点争议的，我曾经为这个问题纠结过很久，毕竟，如果不用互斥量，这么微观的隐患到底存不存在，目前很难做一个“让故障重现”的实验去证明，最后，我是本着“宁可信其有不可信其无”的态度，把互斥量应用在了我的工作中。

(3) volatile。volatile 是一个前缀的修饰关键词，也是用来保护主函数与中断函数共用的全局变量的，只不过，volatile 是针对 C 编译器的，预防“C 编译器在优化代码的时候误伤一些重要的共享数据”，就像预防杀毒软件用力过猛把一些合法软件当作病毒而误杀。加了 volatile 修饰的全局变量，就能提醒 C 编译器不要对这类特殊变量擅作主张去优化。

(4) switch。switch 是“非阻塞程序框架”的核心语句，在以 switch 为核心的框架下，进行不同步骤之间的程序跳转，是做大型裸机程序的常态。

### 【86.2 主函数与定时中断函数的程序框架。】

主函数与定时中断函数之间相互配合，主函数负责做什么，中断函数负责做什么，对于初学者来说可能是一头雾水，但是对于像我这种在单片机界深耕多年即将修炼成精的工程师来说，我心中是有很清晰的模板和套路的，这种模板和套路是经过多年沉淀下来的经验。比如，定时中断函数尽量放一些精简的计时器代码，一般不调用函数，但是“输入 I/O 口的消抖动”（按键扫描）以及“蜂鸣器鸣叫”这两类特殊函数我是喜欢破例放在定时中断函数里调用的。定时中断如何产生时间，这个时间如何跟主函数关联起来，请看下面的框架代码：

```
volatile unsigned char vGu8TimeFlag=0; //互斥量变量标志
volatile unsigned int vGu16TimeCnt=0; //计时器变量

void main()
{
    vGu8TimeFlag=0; //在“写操作”vGu16TimeCnt 全局变量之前，互斥量 vGu8TimeFlag 的“加锁”
    vGu16TimeCnt=1000; //全局变量的赋值，就是“写操作”
```

```

vGu8TimeFlag=1; //互斥量 vGu8TimeFlag 的“解锁”。同时也起到“启动计时器”的开关作用

while(1) //主循环
{
    if(0==vGu16TimeCnt) //时间变量为 0 则表示时间到了
    {
        ... 在这里执行具体的功能代码
    }
}

void TO_time() interrupt 1 //每 1ms 中断一次的定时中断函数
{
    if(1==vGu8TimeFlag&&vGu16TimeCnt>0) //判断 vGu8TimeFlag 是否等于 1，就是互斥量的判断。
    {
        vGu16TimeCnt--; //“自减一”的操作
    }
}

```

分析：上述代码中，vGu8TimeFlag 是一箭双雕，既起到互斥量的作用，也起到了计数器 vGu16TimeCnt 开始计时的启动开关作用。

### 【86.3 练习例程。】

现在根据上述程序框架，编写一个 LED 灯闪烁的程序。

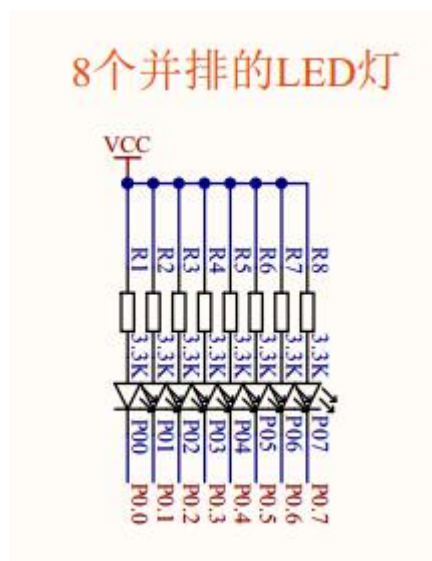


图 86.3.1 灌入式驱动 8 个 LED

```

#include "REG52.H"

#define BLINK_TIME    500    //时间是 500ms

sbit P0_0=P0^0;

volatile unsigned char vGu8TimeFlag=0; //互斥量变量标志
volatile unsigned int vGu16TimeCnt=0; //计时器变量

unsigned char Gu8Step=0; //switch 的切换步骤
void main()
{
    TMOD=0x01; //设置定时器 0 为工作方式 1
    TH0=0xfc;   //产生 1ms 中断的 TH0 初始值
    TL0=0x66;   //产生 1ms 中断的 TL0 初始值
    EA=1;       //开总中断
    ET0=1;      //允许定时 0 的中断
    TR0=1;      //启动定时 0 的中断

    while(1) //主循环
    {
        switch(Gu8Step)
        {
            case 0:
                if(0==vGu16TimeCnt) //时间到
                {
                    P0_0=0; //LED 灯亮
                    vGu8TimeFlag=0; //互斥量“加锁”
                    vGu16TimeCnt=BLINK_TIME; //计时器的写操作。设定计时的长度
                    vGu8TimeFlag=1; //互斥量“解锁”，同时蕴含了计时器“启动”的动作

                    Gu8Step=1; //切换到 case 1 这个步骤
                }
                break;

            case 1:

                if(0==vGu16TimeCnt) //时间到
                {
                    P0_0=1; //LED 灯灭。
                    vGu8TimeFlag=0; //互斥量“加锁”
                    vGu16TimeCnt=BLINK_TIME; //计时器的写操作。设定计时的长度
                    vGu8TimeFlag=1; //互斥量“解锁”，同时蕴含了计时器“启动”的动作

```

```

        Gu8Step=0; //切换到 case 0 这个步骤，依次循环
    }
    break;
}
}

void TO_time() interrupt 1 //定时器 0 的中断函数，每 1ms 单片机自动执行一次此函数
{
    if(1==vGu8TimeFlag&&vGu16TimeCnt>0) //判断 vGu8TimeFlag 是否等于 1，就是互斥量的判断
    {
        vGu16TimeCnt--; //“自减一”的操作
    }

    TH0=0xfc; //重装初值，不能忘
    TL0=0x66; //重装初值，不能忘
}

```

#### 【86.4 解决闪烁出现不规则“非对称感”现象的方法。】

上述例子，实验现象应该是 LED 闪烁很有规则的每 1s 闪烁一次，但是也有一部分初学者可能会遇到闪烁出现不规则“非对称感”的现象，这个问题的解决办法如下：在 keil2 的 project 下拉菜单下，选择 Options for Target 选项，弹出的窗口中，切换到 Target 选项，在 Memory Model 选项中选择 small:variables in Data。

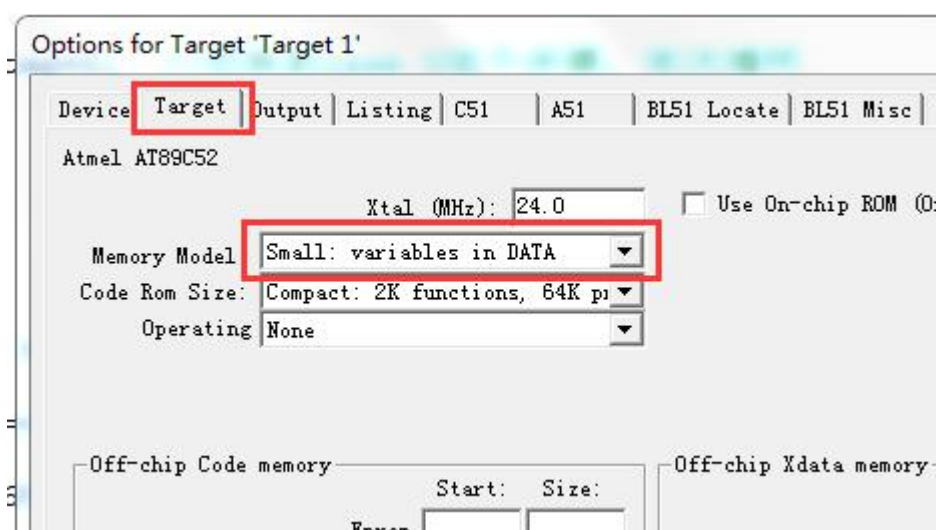


图 86.4.1 设置窗口

## 第八十七节： 一个定时中断产生 N 个软件定时器。

### 【87.1 信手拈来的软件定时器。】

初学者会疑惑，51 单片机只有 2 个定时器 T0 和 T1，是不是太少了一点？2 个定时器怎能满足实际项目的需要，很多项目涉及到的定时器往往十几个，怎么办？这个问题的奥秘就在本节的内容。

51 单片机内置的 2 个定时器 T0 和 T1，是属于硬件定时器，硬件定时器是一个母体，它可以孕育出 N 个软件定时器，实际项目中，我们需要多少个定时器只需要从同一个硬件定时器中断里构造出对应数量的软件定时器即可。构造 N 个软件定时器的框架如下：

```
// “软件定时器 1” 的相关变量
volatile unsigned char vGu8TimeFlag_1=0;
volatile unsigned int vGu16TimeCnt_1=0;

// “软件定时器 2” 的相关变量
volatile unsigned char vGu8TimeFlag_2=0;
volatile unsigned int vGu16TimeCnt_2=0;

// “软件定时器 3” 的相关变量
volatile unsigned char vGu8TimeFlag_3=0;
volatile unsigned int vGu16TimeCnt_3=0;

void main()
{
    vGu8TimeFlag_1=0;
    vGu16TimeCnt_1=1000; // “软件定时器 1” 的定时时间是 1000ms
    vGu8TimeFlag_1=1;

    vGu8TimeFlag_2=0;
    vGu16TimeCnt_2=500; // “软件定时器 2” 的定时时间是 500ms
    vGu8TimeFlag_2=1;

    vGu8TimeFlag_3=0;
    vGu16TimeCnt_3=250; // “软件定时器 3” 的定时时间是 250ms
    vGu8TimeFlag_3=1;

    while(1) //主循环
    {
        if(0==vGu16TimeCnt_1) // “软件定时器 1” 的时间到了
        {
            ... 在这里执行具体的功能代码
        }

        if(0==vGu16TimeCnt_2) // “软件定时器 2” 的时间到了
```



```

    {
        ... 在这里执行具体的功能代码
    }

    if(0==vGu16TimeCnt_3  // “软件定时器 3” 的时间到了
    {
        ... 在这里执行具体的功能代码
    }

}

}

void TO_time() interrupt 1    //每 1ms 中断一次的定时中断函数
{
    if(1==vGu8TimeFlag_1&&vGu16TimeCnt_1>0) //在定时中断里衍生出 “软件定时器 1”
    {
        vGu16TimeCnt_1--;
    }

    if(1==vGu8TimeFlag_2&&vGu16TimeCnt_2>0) //在定时中断里衍生出 “软件定时器 2”
    {
        vGu16TimeCnt_2--;
    }

    if(1==vGu8TimeFlag_3&&vGu16TimeCnt_3>0) //在定时中断里衍生出 “软件定时器 3”
    {
        vGu16TimeCnt_3--;
    }

    //按上面的套路继续写，可以衍生出 N 个 “软件定时器”，只要不超过单片机的 RAM 和 ROM。
}

```

## 【87.2 练习例程。】

现在根据上述程序框架，编写 3 个 LED 灯闪烁的程序。第 1 个 LED 灯的一闪一灭的周期是 2 秒，第 2 个 LED 灯的一闪一灭的周期是 1 秒，第 3 个 LED 灯一闪一灭的周期是 0.5 秒。这 3 个灯的闪烁频率是不一样的，因此需要 3 个软件定时器。该例子其实也是一个多任务并行处理的典型例子，这 3 个 LED 灯就代表 3 个不同的任务，它们之间是通过 switch 这个关键语句进行多任务并行处理的。switch 的精髓在于根据某个特定条件切换到对应的步骤（或称“跳转到对应的步骤”）。

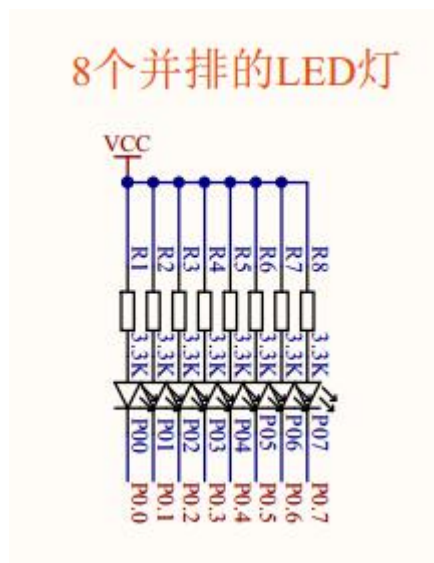


图 87.2.1 灌入式驱动 8 个 LED

```
#include "REG52.H"

#define BLINK_TIME_1    1000    //时间是 1000ms
#define BLINK_TIME_2    500     //时间是 500ms
#define BLINK_TIME_3    250     //时间是 250ms

sbit P0_0=P0^0;
sbit P0_1=P0^1;
sbit P0_2=P0^2;

// “软件定时器 1” 的相关变量
volatile unsigned char vGu8TimeFlag_1=0;
volatile unsigned int vGu16TimeCnt_1=0;

// “软件定时器 2” 的相关变量
volatile unsigned char vGu8TimeFlag_2=0;
volatile unsigned int vGu16TimeCnt_2=0;

// “软件定时器 3” 的相关变量
volatile unsigned char vGu8TimeFlag_3=0;
volatile unsigned int vGu16TimeCnt_3=0;

unsigned char Gu8Step_1=0; //软件定时器 1 的 switch 切换步骤
unsigned char Gu8Step_2=0; //软件定时器 2 的 switch 切换步骤
unsigned char Gu8Step_3=0; //软件定时器 3 的 switch 切换步骤

void main()
```

```

{
    TMOD=0x01; //设置定时器 0 为工作方式 1
    TH0=0xfc;  //产生 1ms 中断的 TH0 初始值
    TL0=0x66;  //产生 1ms 中断的 TL0 初始值
    EA=1;      //开总中断
    ET0=1;     //允许定时 0 的中断
    TR0=1;     //启动定时 0 的中断

    while(1) //主循环
    {

        //软件定时器 1 控制的 LED 灯闪烁
        switch(Gu8Step_1)
        {
            case 0:
                if(0==vGu16TimeCnt_1)
                {
                    P0_0=0;
                    vGu8TimeFlag_1=0;
                    vGu16TimeCnt_1=BLINK_TIME_1;
                    vGu8TimeFlag_1=1;

                    Gu8Step_1=1;
                }
                break;

            case 1:

                if(0==vGu16TimeCnt_1)
                {
                    P0_0=1;
                    vGu8TimeFlag_1=0;
                    vGu16TimeCnt_1=BLINK_TIME_1;
                    vGu8TimeFlag_1=1;

                    Gu8Step_1=0;
                }
                break;
        }

        //软件定时器 2 控制的 LED 灯闪烁
        switch(Gu8Step_2)
        {

```

```

        case 0:
            if(0==vGu16TimeCnt_2)
            {
                P0_1=0;
                vGu8TimeFlag_2=0;
                vGu16TimeCnt_2=BLINK_TIME_2;
                vGu8TimeFlag_2=1;

                Gu8Step_2=1;
            }
            break;

        case 1:

            if(0==vGu16TimeCnt_2)
            {
                P0_1=1;
                vGu8TimeFlag_2=0;
                vGu16TimeCnt_2=BLINK_TIME_2;
                vGu8TimeFlag_2=1;

                Gu8Step_2=0;
            }
            break;
    }

//软件定时器 3 控制的 LED 灯闪烁
switch(Gu8Step_3)
{
    case 0:
        if(0==vGu16TimeCnt_3)
        {
            P0_2=0;
            vGu8TimeFlag_3=0;
            vGu16TimeCnt_3=BLINK_TIME_3;
            vGu8TimeFlag_3=1;

            Gu8Step_3=1;
        }
        break;

    case 1:

        if(0==vGu16TimeCnt_3)

```

```

        {
            P0_2=1;
            vGu8TimeFlag_3=0;
            vGu16TimeCnt_3=BLINK_TIME_3;
            vGu8TimeFlag_3=1;

            Gu8Step_3=0;
        }
        break;
    }
}

void TO_time() interrupt 1    //定时器 0 的中断函数，每 1ms 单片机自动执行一次此函数
{
    if(1==vGu8TimeFlag_1&&vGu16TimeCnt_1>0) //在定时中断里衍生出“软件定时器 1”
    {
        vGu16TimeCnt_1--;
    }

    if(1==vGu8TimeFlag_2&&vGu16TimeCnt_2>0) //在定时中断里衍生出“软件定时器 2”
    {
        vGu16TimeCnt_2--;
    }

    if(1==vGu8TimeFlag_3&&vGu16TimeCnt_3>0) //在定时中断里衍生出“软件定时器 3”
    {
        vGu16TimeCnt_3--;
    }

    TH0=0xfc;    //重装初值，不能忘
    TL0=0x66;    //重装初值，不能忘
}

```

## 第八十八节： 两大核心框架理论（四区一线，switch 外加定时中断）。

### 【88.1 四区一线。】

提出“四区一线”理论，主要方便初学者理解单片机程序大概的“空间分区”。

“四区”代表四大主流函数，分别是：系统初始化函数，外设初始化函数，主程序的任务函数，定时中断函数。

“一线”是指“系统初始化函数”与“外设初始化函数”的“分割线”，这个“分割线”是一个 delay 的延时函数。

“四区一线”的布局如下：

```
void main()
{
    SystemInitial();           // “四区一线”的“第一区”
    Delay(10000);              // “四区一线”的“一线”
    PeripheralInitial();        // “四区一线”的“第二区”
    while(1) //主循环
    {
        LedTask();             // “四区一线”的“第三区”
        KeyTask();             // “四区一线”的“第三区”
        UsartTask();           // “四区一线”的“第三区”
        ...                    //凡是在主循环里的函数都是属于“第三区”
    }
}

void TO_time() interrupt 1     // “四区一线”的“第四区”
{
}
```

“第一区”的函数 SystemInitial(), 是一个系统的初始化函数，专门用来初始化单片机自己的寄存器以及个别外围要求响应速度快的输出设备，防止刚上电之后，由于输出 I/O 口电平状态不确定而导致外围设备误动作，比如驱动继电器的误动作等等。

“一线”的函数 Delay(10000)，是一个延时函数，为什么这里要插入一个延时函数？主要目的是为接下来的 PeripheralInitial() 做准备的。上电后先延时一段时间，再执行 PeripheralInitial() 函数，因为 PeripheralInitial() 函数专门用来初始化不要求上电立即处理的外设芯片和模块。比如液晶模块，AT24C02 存储芯片，DS1302 时钟芯片，等等。这些芯片在上电的瞬间，内部自身的复位需要一点时间，以及外部电压稳定也需要一点时间，只有过了这一点时间，这些芯片才处于正常的工作状态，这个时候单片机才能跟它正常通信，所以“一线”函数 Delay(10000) 的意义就在这里。

“第二区”的函数 PeripheralInitial(), 是一个外设的初始化函数。专门用来初始化不要求上电立即处理的外设芯片和模块。

“第三区”的函数 LedTask(), KeyTask(), UsartTask(), 等等，是一些在主循环里不断扫描的任务函数。

“第四区”的函数 void TO\_time() interrupt 1, 是一个定时中断函数，一个系统必须标配一个定时中

断函数才算完美齐全，这个中断函数提供系统的节拍时间，以及处理扫描一些跟 I/O 口消抖动相关的函数，以及跟蜂鸣器驱动相关的函数。

### 【88.2 switch 外加定时中断。】

提出“switch 外加定时中断”理论，主要方便初学者理解单片机程序大概的“逻辑框架”。

switch 是一个万能语句，它外加 while 与 for 循环就可以做任何复杂的算法，比如，搜索算法，运动算法，提取关键词算法，等等。它外加定时中断，就可以搭建一个系统的基本框架。比如，做通信的程序框架，人机界面的程序框架，按键服务的程序框架，等等。switch 的精髓在于“根据条件进行步骤的灵活切换”。具体内容请看本节的练习程序。

### 【88.3 练习例程。】

根据上述的两大核心框架理论，编写 1 个 LED 灯闪烁的程序。



图 88.3.1 灌入式驱动 8 个 LED

```
#include "REG52.H"

void TO_time();
void SystemInitial(void) ;
void Delay(unsigned long u32DelayTime) ;
void PeripheralInitial(void) ;
void LedTask(void);

#define BLINK_TIME_1 1000

sbit P0_0=P0^0;

volatile unsigned char vGu8TimeFlag_1=0;
```

```

volatile unsigned int vGul6TimeCnt_1=0;

void main()
{
    SystemInitial();          // “四区一线”的“第一区”
    Delay(10000);             // “四区一线”的“一线”
    PeripheralInitial();       // “四区一线”的“第二区”
    while(1) //主循环
    {
        LedTask();            // “四区一线”的“第三区”
    }
}

void T0_time() interrupt 1    // “四区一线”的“第四区”
{
    if(1==vGu8TimeFlag_1&&vGul6TimeCnt_1>0)
    {
        vGul6TimeCnt_1--;
    }

    TH0=0xfc;
    TL0=0x66;
}

void SystemInitial(void)
{
    TMOD=0x01;
    TH0=0xfc;
    TL0=0x66;
    EA=1;
    ET0=1;
    TR0=1;
}

void Delay(unsigned long u32DelayTime)
{
    for(;u32DelayTime>0;u32DelayTime--);
}

void PeripheralInitial(void)
{
}

```



```

void LedTask(void)
{
    static unsigned char Su8Step=0; //加 static 修饰的局部变量，每次进来都会保留上一次值。

    switch(Su8Step)
    {
        case 0:
            if(0==vGu16TimeCnt_1) //时间到
            {
                P0_0=0;
                vGu8TimeFlag_1=0;
                vGu16TimeCnt_1=BLINK_TIME_1; //重装定时的时间
                vGu8TimeFlag_1=1;

                Su8Step=1; //切换到下一个步骤，精髓语句！
            }
            break;

        case 1:

            if(0==vGu16TimeCnt_1) //时间到
            {
                P0_0=1;
                vGu8TimeFlag_1=0;
                vGu16TimeCnt_1=BLINK_TIME_1; //重装定时的时间
                vGu8TimeFlag_1=1;

                Su8Step=0; //返回到上一个步骤，精髓语句！
            }
            break;
    }
}

```

## 第八十九节： 跑马灯的三种境界。

### 【89.1 跑马灯的三种境界。】

跑马灯也称为流水灯，排列的几个 LED 依次循环的点亮和熄灭，给人“跑动起来”的感觉，故称为“跑马灯”。实现跑马灯的效果，编程上有三种思路，分别代表了跑马灯的三种境界，分别是：移位阻塞，移位非阻塞，状态切换非阻塞。

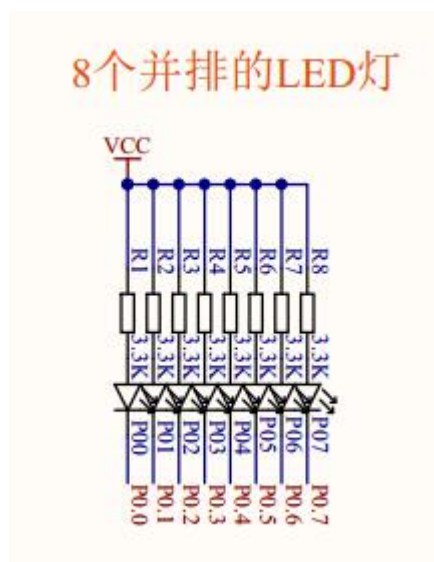


图 89.1.1 灌入式驱动 8 个 LED

本节用的是 8 个 LED 灯依次挨个熄灭点亮，如上图所示。

### 【89.2 移位阻塞。】

移位阻塞，“移位”用的是 C 语言的左移或者右移语句，“阻塞”用的是 delay 延时。代码如下：

```
#include "REG52.H"

void TO_time();
void SystemInitial(void) ;
void Delay(unsigned long u32DelayTime) ;
void PeripheralInitial(void) ;
void LedTask(void);

void main()
{
    SystemInitial();
    Delay(10000);
    PeripheralInitial();
    while(1)
```

```

    {
        LedTask();
    }
}

void T0_time() interrupt 1
{
    TH0=0xfc;
    TL0=0x66;
}

void SystemInitial(void)
{
    TMOD=0x01;
    TH0=0xfc;
    TL0=0x66;
    EA=1;
    ET0=1;
    TR0=1;
}

void Delay(unsigned long u32DelayTime)
{
    for(;u32DelayTime>0;u32DelayTime--);
}

void PeripheralInitial(void)
{
}

//跑马灯的任务程序
void LedTask(void)
{
    static unsigned char Su8Data=0x01; //加 static 修饰的局部变量，每次进来都会保留上一次值。
    static unsigned char Su8Cnt=0;      //加 static 修饰的局部变量，每次进来都会保留上一次值。

    P0=Su8Data; //Su8Data 的 8 个位代表 8 个 LED 的状态，0 为点亮，1 为熄灭。
    Delay(10000); //阻塞延时
    Su8Data=Su8Data<<1; //左移一位
    Su8Cnt++; //计数器累加 1
    if(Su8Cnt>=8) //移位大于等于 8 次后，重新赋初值

```

```

    {
        Su8Cnt=0;
        Su8Data=0x01; //重新赋初值，继续下一次循环移动
    }
}

```

分析总结：这是第 1 种境界的跑马灯，这种思路虽然实现了跑马灯的效果，但是因为“阻塞延时”，整个程序显得僵硬机械，缺乏多任务并行的框架。

### 【89.3 移位非阻塞。】

移位非阻塞，“移位”用的是 C 语言的左移或者右移语句，“非阻塞”用的是定时中断衍生出来的软件定时器。代码如下：

```

#include "REG52.H"

void T0_time();
void SystemInitial(void) ;
void Delay(unsigned long u32DelayTime) ;
void PeripheralInitial(void) ;
void LedTask(void);

#define BLINK_TIME_1 1000

volatile unsigned char vGu8TimeFlag_1=0;
volatile unsigned int vGu16TimeCnt_1=0;

void main()
{
    SystemInitial();
    Delay(10000);
    PeripheralInitial();
    while(1)
    {
        LedTask();
    }
}

void T0_time() interrupt 1
{
    if(1==vGu8TimeFlag_1&&vGu16TimeCnt_1>0) //软件定时器
    {
        vGu16TimeCnt_1--;
    }
}

```

```

    }

    TH0=0xfc;
    TL0=0x66;
}

void SystemInitial(void)
{
    TMOD=0x01;
    TH0=0xfc;
    TL0=0x66;
    EA=1;
    ET0=1;
    TR0=1;
}

void Delay(unsigned long u32DelayTime)
{
    for(;u32DelayTime>0;u32DelayTime--);
}

void PeripheralInitial(void)
{
}

//跑马灯的任务程序
void LedTask(void)
{
    static unsigned char Su8Data=0x01; //加 static 修饰的局部变量，每次进来都会保留上一次值。
    static unsigned char Su8Cnt=0;      //加 static 修饰的局部变量，每次进来都会保留上一次值。

    if(0==vGu16TimeCnt_1)      //时间到
    {
        vGu8TimeFlag_1=0;
        vGu16TimeCnt_1=BLINK_TIME_1; //重装定时的时间
        vGu8TimeFlag_1=1;

        P0=Su8Data; //Su8Data 的 8 个位代表 8 个 LED 的状态，0 为点亮，1 为熄灭。
        Su8Data=Su8Data<<1; //左移一位
        Su8Cnt++; //计数器累加 1
        if(Su8Cnt>=8) //移位大于等于 8 次后，重新赋初值
        {

```

```

        Su8Cnt=0;
        Su8Data=0x01; //重新赋初值，继续下一次循环移动
    }
}
}

```

分析总结：这是第 2 种境界的跑马灯，这种思路虽然实现了跑马灯的效果，也用到了多任务并行处理的基本元素“软件定时器”，但是因为还停留在“移位”语句的阶段，此时的程序并没有超越跑马灯本身，跑马灯还是跑马灯，处于“看山还是山”的境界。

#### 【89.4 状态切换非阻塞。】

状态切换非阻塞，“状态切换”用的是 switch 语句中根据特定条件进行步骤切换，“非阻塞”用的是定时中断衍生出来的软件定时器。代码如下：

```

#include "REG52.H"

void T0_time();
void SystemInitial(void) ;
void Delay(unsigned long u32DelayTime) ;
void PeripheralInitial(void) ;
void LedTask(void);

#define BLINK_TIME_1 1000

sbit P0_0=P0^0;
sbit P0_1=P0^1;
sbit P0_2=P0^2;
sbit P0_3=P0^3;
sbit P0_4=P0^4;
sbit P0_5=P0^5;
sbit P0_6=P0^6;
sbit P0_7=P0^7;

volatile unsigned char vGu8TimeFlag_1=0;
volatile unsigned int vGu16TimeCnt_1=0;

void main()
{
    SystemInitial();
    Delay(10000);
    PeripheralInitial();
    while(1)

```

```

    {
        LedTask();
    }
}

void T0_time() interrupt 1
{
    if(1==vGu8TimeFlag_1&&vGu16TimeCnt_1>0) //软件定时器
    {
        vGu16TimeCnt_1--;
    }

    TH0=0xfc;
    TL0=0x66;
}

void SystemInitial(void)
{
    TMOD=0x01;
    TH0=0xfc;
    TL0=0x66;
    EA=1;
    ET0=1;
    TR0=1;
}

void Delay(unsigned long u32DelayTime)
{
    for(;u32DelayTime>0;u32DelayTime--);
}

void PeripheralInitial(void)
{
}

//跑马灯的任务程序
void LedTask(void)
{
    static unsigned char Su8Step=0;    //加 static 修饰的局部变量，每次进来都会保留上一次值。

    switch(Su8Step)
    {

```

```

case 0:
    if(0==vGu16TimeCnt_1)  //时间到
    {

        vGu8TimeFlag_1=0;
        vGu16TimeCnt_1=BLINK_TIME_1;  //重装定时的时间
        vGu8TimeFlag_1=1;

        P0_0=1;  //第 0 个灯熄灭
        P0_1=0;
        P0_2=0;
        P0_3=0;
        P0_4=0;
        P0_5=0;
        P0_6=0;
        P0_7=0;

        Su8Step=1;  //切换到下一个步骤，精髓语句！
    }
    break;

case 1:
    if(0==vGu16TimeCnt_1)  //时间到
    {

        vGu8TimeFlag_1=0;
        vGu16TimeCnt_1=BLINK_TIME_1;  //重装定时的时间
        vGu8TimeFlag_1=1;

        P0_0=0;
        P0_1=1;  //第 1 个灯熄灭
        P0_2=0;
        P0_3=0;
        P0_4=0;
        P0_5=0;
        P0_6=0;
        P0_7=0;

        Su8Step=2;  //切换到下一个步骤，精髓语句！
    }
    break;

case 2:
    if(0==vGu16TimeCnt_1)  //时间到

```



```

{

    vGu8TimeFlag_1=0;
    vGu16TimeCnt_1=BLINK_TIME_1;    //重装定时的时间
    vGu8TimeFlag_1=1;

    P0_0=0;
    P0_1=0;
    P0_2=1;    //第 2 个灯熄灭
    P0_3=0;
    P0_4=0;
    P0_5=0;
    P0_6=0;
    P0_7=0;

    Su8Step=3;    //切换到下一个步骤，精髓语句！
}
break;

case 3:
    if(0==vGu16TimeCnt_1)    //时间到
    {

        vGu8TimeFlag_1=0;
        vGu16TimeCnt_1=BLINK_TIME_1;    //重装定时的时间
        vGu8TimeFlag_1=1;

        P0_0=0;
        P0_1=0;
        P0_2=0;
        P0_3=1;    //第 3 个灯熄灭
        P0_4=0;
        P0_5=0;
        P0_6=0;
        P0_7=0;

        Su8Step=4;    //切换到下一个步骤，精髓语句！
    }
    break;

case 4:
    if(0==vGu16TimeCnt_1)    //时间到
    {

```

```

        vGu8TimeFlag_1=0;
        vGu16TimeCnt_1=BLINK_TIME_1;    //重装定时的时间
        vGu8TimeFlag_1=1;

        P0_0=0;
        P0_1=0;
        P0_2=0;
        P0_3=0;
        P0_4=1;    //第 4 个灯熄灭
        P0_5=0;
        P0_6=0;
        P0_7=0;

        Su8Step=5;    //切换到下一个步骤，精髓语句！
    }
    break;

case 5:
    if(0==vGu16TimeCnt_1)    //时间到
    {

        vGu8TimeFlag_1=0;
        vGu16TimeCnt_1=BLINK_TIME_1;    //重装定时的时间
        vGu8TimeFlag_1=1;

        P0_0=0;
        P0_1=0;
        P0_2=0;
        P0_3=0;
        P0_4=0;
        P0_5=1;    //第 5 个灯熄灭
        P0_6=0;
        P0_7=0;

        Su8Step=6;    //切换到下一个步骤，精髓语句！
    }
    break;

case 6:
    if(0==vGu16TimeCnt_1)    //时间到
    {

        vGu8TimeFlag_1=0;
        vGu16TimeCnt_1=BLINK_TIME_1;    //重装定时的时间

```

```

        vGu8TimeFlag_1=1;

        P0_0=0;
        P0_1=0;
        P0_2=0;
        P0_3=0;
        P0_4=0;
        P0_5=0;
        P0_6=1;    //第 6 个灯熄灭
        P0_7=0;

        Su8Step=7;    //切换到下一个步骤，精髓语句！
    }
    break;

case 7:

    if(0==vGu16TimeCnt_1)    //时间到
    {
        vGu8TimeFlag_1=0;
        vGu16TimeCnt_1=BLINK_TIME_1;    //重装定时的时间
        vGu8TimeFlag_1=1;

        P0_0=0;
        P0_1=0;
        P0_2=0;
        P0_3=0;
        P0_4=0;
        P0_5=0;
        P0_6=0;
        P0_7=1;    //第 7 个灯熄灭

        Su8Step=0;    //返回到第 0 个步骤重新开始往下走，精髓语句！
    }
    break;
}
}

```

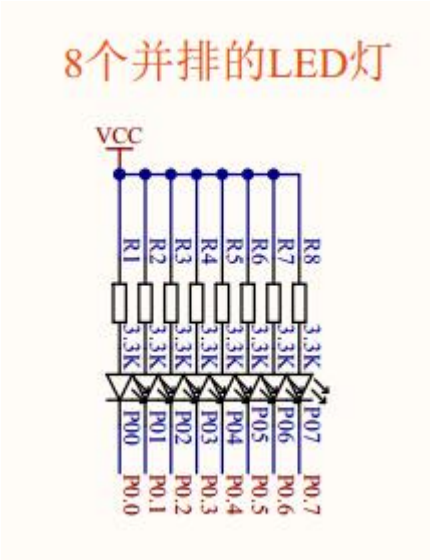
分析总结：这是第 3 种境界的跑马灯，很多初学者咋看此程序，表示不理解，人家一条赋值语句就解决 8 个 LED 一次性显示的问题，你非要拆分成 8 条按位赋值的语句，人家只用一个判断就实现了 LED 灯移动显示的功能，你非要整出 8 个步骤的切换，况且，整个程序的代码量明显增加了很多，这个程序好在哪？其实，我这么做是用心良苦呀。这个程序的代码量虽然增多了，但是仔细一看，并没有影响运行的效率。之所以把 8 个 LED 灯拆分成一个一个的 LED 灯单独赋值显示，是因为，在我眼里，这个 8 个 LED 灯代表的不仅仅是 LED 灯，而是 8 个输出信号！这 8 个输出信号未来驱动的可能是不同的继电器，气缸，电机，大炮，导弹，以及

它们的各种千变万化的组合逻辑，拆分之后程序框架就有了无限可能的扩展性。之所以整出 8 个步骤的切换，也是同样的道理，为了增加程序框架无限可能的扩展性。这个程序虽然表面看起来繁琐，但是仔细一看它是“多而不乱”，非常富有“队形感”。因此可以这么说，这个看似繁琐的跑马灯程序，其实背后蕴藏了编程界的大智慧，它已经突破了“看山还是山”的境界。

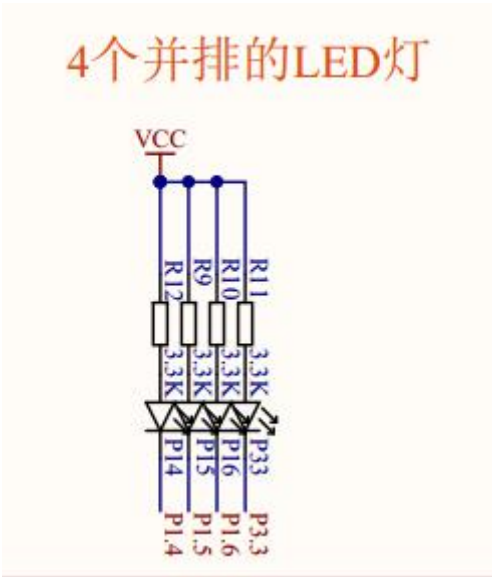
第九十节： 多任务并行处理两路跑马灯。

【90.1 多任务并行处理。】

两路速度不同的跑马灯，代表了两路独立运行的任务，单片机如何“并行”处理这两路任务，就涉及到“多任务并行处理的编程思路”。



上图 90.1.1 灌入式驱动 8 个 LED 第 1 路跑马灯



上图 90.1.2 灌入式驱动 4 个 LED 新增加的第 2 路跑马灯

如上图，本节特别值得一提的是，新增加的第 2 路跑马灯用的是 4 个 LED，这 4 个 LED 的驱动 IO 口是“散装的”，因为，前面 3 个是 P1 口的（P1.4, P1.5, P1.6），最后 1 个是 P3 口的（P3.3），这种情况下，肯定用不了“移位”的处理思路，只能用跑马灯第 3 种境界里所介绍的“状态切换非阻塞”思路，可见，“IO 口拆分”和“switch 状态切换”又一次充分体现了它们“程序框架万能扩展”的优越性。代码如下：

```

#include "REG52.H"

void T0_time();
void SystemInitial(void) ;
void Delay(unsigned long u32DelayTime) ;
void PeripheralInitial(void) ;
void Led_1_Task(void);
void Led_2_Task(void);

#define BLINK_TIME_1    1000 //控制第1路跑马灯的速度，数值越大“跑动”越慢。
#define BLINK_TIME_2    200  //控制第2路跑马灯的速度，数值越大“跑动”越慢。

sbit P0_0=P0^0;
sbit P0_1=P0^1;
sbit P0_2=P0^2;
sbit P0_3=P0^3;
sbit P0_4=P0^4;
sbit P0_5=P0^5;
sbit P0_6=P0^6;
sbit P0_7=P0^7;

sbit P1_4=P1^4;
sbit P1_5=P1^5;
sbit P1_6=P1^6;
sbit P3_3=P3^3;

volatile unsigned char vGu8TimeFlag_1=0;
volatile unsigned int vGu16TimeCnt_1=0;

volatile unsigned char vGu8TimeFlag_2=0;
volatile unsigned int vGu16TimeCnt_2=0;

void main()
{
    SystemInitial();
    Delay(10000);
    PeripheralInitial();
    while(1)
    {
        Led_1_Task(); //第1路跑马灯
        Led_2_Task(); //第2路跑马灯
    }
}

```

```

void TO_time() interrupt 1
{
    if(1==vGu8TimeFlag_1&&vGu16TimeCnt_1>0) //软件定时器 1
    {
        vGu16TimeCnt_1--;
    }

    if(1==vGu8TimeFlag_2&&vGu16TimeCnt_2>0) //软件定时器 2
    {
        vGu16TimeCnt_2--;
    }

    TH0=0xfc;
    TL0=0x66;
}

void SystemInitial(void)
{
    TMOD=0x01;
    TH0=0xfc;
    TL0=0x66;
    EA=1;
    ET0=1;
    TR0=1;
}

void Delay(unsigned long u32DelayTime)
{
    for(;u32DelayTime>0;u32DelayTime--);
}

void PeripheralInitial(void)
{
}

//第 1 路跑马灯
void Led_1_Task(void)
{
    static unsigned char Su8Step=0;    //加 static 修饰的局部变量，每次进来都会保留上一次值。

```

```

switch(Su8Step)
{
    case 0:
        if(0==vGu16TimeCnt_1) //时间到
        {

            vGu8TimeFlag_1=0;
            vGu16TimeCnt_1=BLINK_TIME_1; //重装定时的时间
            vGu8TimeFlag_1=1;

            P0_0=1; //第 0 个灯熄灭
            P0_1=0;
            P0_2=0;
            P0_3=0;
            P0_4=0;
            P0_5=0;
            P0_6=0;
            P0_7=0;

            Su8Step=1; //切换到下一个步骤，精髓语句！
        }
        break;

    case 1:
        if(0==vGu16TimeCnt_1) //时间到
        {

            vGu8TimeFlag_1=0;
            vGu16TimeCnt_1=BLINK_TIME_1; //重装定时的时间
            vGu8TimeFlag_1=1;

            P0_0=0;
            P0_1=1; //第 1 个灯熄灭
            P0_2=0;
            P0_3=0;
            P0_4=0;
            P0_5=0;
            P0_6=0;
            P0_7=0;

            Su8Step=2; //切换到下一个步骤，精髓语句！
        }
        break;

```



```

case 2:
    if(0==vGu16TimeCnt_1)  //时间到
    {

        vGu8TimeFlag_1=0;
        vGu16TimeCnt_1=BLINK_TIME_1;  //重装定时的时间
        vGu8TimeFlag_1=1;

        P0_0=0;
        P0_1=0;
        P0_2=1;  //第 2 个灯熄灭
        P0_3=0;
        P0_4=0;
        P0_5=0;
        P0_6=0;
        P0_7=0;

        Su8Step=3;  //切换到下一个步骤，精髓语句！
    }
    break;

case 3:
    if(0==vGu16TimeCnt_1)  //时间到
    {

        vGu8TimeFlag_1=0;
        vGu16TimeCnt_1=BLINK_TIME_1;  //重装定时的时间
        vGu8TimeFlag_1=1;

        P0_0=0;
        P0_1=0;
        P0_2=0;
        P0_3=1;  //第 3 个灯熄灭
        P0_4=0;
        P0_5=0;
        P0_6=0;
        P0_7=0;

        Su8Step=4;  //切换到下一个步骤，精髓语句！
    }
    break;

case 4:
    if(0==vGu16TimeCnt_1)  //时间到

```

```

{

    vGu8TimeFlag_1=0;
    vGu16TimeCnt_1=BLINK_TIME_1;    //重装定时的时间
    vGu8TimeFlag_1=1;

    P0_0=0;
    P0_1=0;
    P0_2=0;
    P0_3=0;
    P0_4=1;    //第 4 个灯熄灭
    P0_5=0;
    P0_6=0;
    P0_7=0;

    Su8Step=5;    //切换到下一个步骤，精髓语句！
}
break;

case 5:
    if(0==vGu16TimeCnt_1)    //时间到
    {

        vGu8TimeFlag_1=0;
        vGu16TimeCnt_1=BLINK_TIME_1;    //重装定时的时间
        vGu8TimeFlag_1=1;

        P0_0=0;
        P0_1=0;
        P0_2=0;
        P0_3=0;
        P0_4=0;
        P0_5=1;    //第 5 个灯熄灭
        P0_6=0;
        P0_7=0;

        Su8Step=6;    //切换到下一个步骤，精髓语句！
    }
    break;

case 6:
    if(0==vGu16TimeCnt_1)    //时间到
    {

```

```

        vGu8TimeFlag_1=0;
        vGu16TimeCnt_1=BLINK_TIME_1;    //重装定时的时间
        vGu8TimeFlag_1=1;

        P0_0=0;
        P0_1=0;
        P0_2=0;
        P0_3=0;
        P0_4=0;
        P0_5=0;
        P0_6=1;    //第 6 个灯熄灭
        P0_7=0;

        Su8Step=7;    //切换到下一个步骤，精髓语句！
    }
    break;

case 7:

    if(0==vGu16TimeCnt_1)    //时间到
    {
        vGu8TimeFlag_1=0;
        vGu16TimeCnt_1=BLINK_TIME_1;    //重装定时的时间
        vGu8TimeFlag_1=1;

        P0_0=0;
        P0_1=0;
        P0_2=0;
        P0_3=0;
        P0_4=0;
        P0_5=0;
        P0_6=0;
        P0_7=1;    //第 7 个灯熄灭

        Su8Step=0;    //返回到第 0 个步骤重新开始往下走，精髓语句！
    }
    break;
}

//第 2 路跑马灯
void Led_2_Task(void)
{
/*

```

疑点讲解(1):

这里第 2 路跑马灯的“Su8Step”与第 1 路跑马灯的“Su8Step”虽然同名，但是，因为它们是静态的局部变量，在两个不同的函数内部，是两个不同的变量，这两个变量所分配的 RAM 内存地址是不一样的，因此，它们虽然同名，但是不矛盾不冲突。

\*/

```
static unsigned char Su8Step=0;    //加 static 修饰的局部变量，每次进来都会保留上一次值。
```

```
switch(Su8Step)
```

```
{
```

```
case 0:
```

```
    if(0==vGu16TimeCnt_2) //时间到
```

```
    {
```

```
        vGu8TimeFlag_2=0;
```

```
        vGu16TimeCnt_2=BLINK_TIME_2;    //重装定时的时间
```

```
        vGu8TimeFlag_2=1;
```

```
        P1_4=1; //第 0 个灯熄灭
```

```
        P1_5=0;
```

```
        P1_6=0;
```

```
        P3_3=0;
```

```
        Su8Step=1; //切换到下一个步骤，精髓语句！
```

```
    }
```

```
    break;
```

```
case 1:
```

```
    if(0==vGu16TimeCnt_2) //时间到
```

```
    {
```

```
        vGu8TimeFlag_2=0;
```

```
        vGu16TimeCnt_2=BLINK_TIME_2;    //重装定时的时间
```

```
        vGu8TimeFlag_2=1;
```

```
        P1_4=0;
```

```
        P1_5=1; //第 1 个灯熄灭
```

```
        P1_6=0;
```

```
        P3_3=0;
```

```
        Su8Step=2; //切换到下一个步骤，精髓语句！
```

```
    }
```

```
    break;
```

```
case 2:
```

```

        if(0==vGu16TimeCnt_2) //时间到
        {

            vGu8TimeFlag_2=0;
            vGu16TimeCnt_2=BLINK_TIME_2; //重装定时的时间
            vGu8TimeFlag_2=1;

            P1_4=0;
            P1_5=0;
            P1_6=1; //第 2 个灯熄灭
            P3_3=0;

            Su8Step=3; //切换到下一个步骤，精髓语句！
        }
        break;

    case 3:
        if(0==vGu16TimeCnt_2) //时间到
        {

            vGu8TimeFlag_2=0;
            vGu16TimeCnt_2=BLINK_TIME_2; //重装定时的时间
            vGu8TimeFlag_2=1;

            P1_4=0;
            P1_5=0;
            P1_6=0;
            P3_3=1; //第 3 个灯熄灭

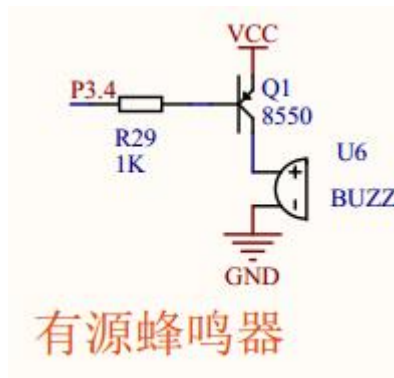
            Su8Step=0; //返回到第 0 个步骤重新开始往下走，精髓语句！
        }
        break;

    }
}

```

## 第九十一节：蜂鸣器的“非阻塞”驱动。

### 【91.1 蜂鸣器的硬件电路简介。】



上图 91.1.1 PNP 三极管驱动有源蜂鸣器

蜂鸣器有两种，一种是有源蜂鸣器，一种是无源蜂鸣器。有源蜂鸣器的驱动最简单，只要通电就一直响，断电就停，跟驱动 LED 灯一样。无源蜂鸣器则不一样，无源蜂鸣器一直断电不响，奇怪的是一直通电也不响，只有“通，关，通，关...”反复通电关电的状态，才会持续发生稳定的声音，此方式称为脉冲驱动方式，或者 PWM 驱动方式。本教程用的是有源蜂鸣器。

蜂鸣器的驱动电路也有两种常用的方式，一种是 NPN 三极管驱动，一种是 PNP 三极管驱动。NPN 三极管驱动电路，单片机输出“1”（高电平）蜂鸣器导通，输出“0”（低电平）蜂鸣器关闭。而 PNP 三极管驱动电路恰恰相反，单片机输出“0”（低电平）蜂鸣器导通，输出“1”（高电平）蜂鸣器关闭。本教程所用的是 PNP 三极管驱动电路，如上图。

### 【91.2 “非阻塞”驱动程序。】

“驱动层”是相对“应用层”而言。“应用层”发号施令，“驱动层”负责执行。一个好的“驱动层”必须给“应用层”提供快捷便利的调用接口，此接口可以是函数或者全局变量。本节驱动蜂鸣器所用的是全局变量 `vGu16BeepTimerCnt` 和 `vGu8BeepTimerFlag`。“应用层”只需给 `vGu16BeepTimerCnt` 赋值，给 `vGu8BeepTimerFlag` 置 1，就可以控制蜂鸣器发声，赋值越大，发声越长，500 代表发声 500ms，1000 代表发声 1000ms，具体细节实现，则由“驱动层”的驱动函数负责执行，驱动函数放在定时中断函数里定时扫描。为什么不把驱动函数放到 `main` 函数的循环里去？因为放在定时中断里，能保证蜂鸣器的声音长度是一致的，如果放在 `main` 循环里，声音的长度有可能在某些项目中受到某些必须一气呵成的任务干扰，得不到及时响应，影响声音长度的一致性。下面代码实现的功能是，单片机只要一上电，蜂鸣器就发出一次 1000ms 长度的“嘀”声音。

```
#include "REG52.H"

#define BEEP_TIME 1000 //控制蜂鸣器发声的长度，此处是 1000ms

void TO_time();
void SystemInitial(void);
void Delay(unsigned long u32DelayTime);
```

```

void PeripheralInitial(void) ;

void BeepOpen(void);    //蜂鸣器发声
void BeepClose(void);   //蜂鸣器关闭
void VoiceScan(void);   //蜂鸣器的驱动函数，放在定时中断里

sbit P3_4=P3^4;  //控制蜂鸣器的 I/O 口。0 代表发声，1 代表关闭。

volatile unsigned char vGu8BeepTimerFlag=0;
volatile unsigned int vGu16BeepTimerCnt=0;  //控制蜂鸣器发声长度的计时器

void main()
{
    SystemInitial();
    Delay(10000);
    PeripheralInitial();    //此函数内部有“应用层”的赋值操作，控制上电的声音长度。
    while(1)
    {
        ;
    }
}

void T0_time() interrupt 1
{
    VoiceScan();  //蜂鸣器的驱动函数

    TH0=0xfc;
    TL0=0x66;
}

void SystemInitial(void)
{
    TMOD=0x01;
    TH0=0xfc;
    TL0=0x66;
    EA=1;
    ET0=1;
    TR0=1;
}

void Delay(unsigned long u32DelayTime)
{
    for(;u32DelayTime>0;u32DelayTime--);
}

```

```

}

void PeripheralInitial(void)
{
    vGu8BeepTimerFlag=0;
    vGu16BeepTimerCnt=BEEP_TIME; // “应用层” 只需赋值，一上电，蜂鸣器发出 1000ms 长度的声音。
    vGu8BeepTimerFlag=1;
}

//蜂鸣器发声
void BeepOpen(void)
{
    P3_4=0; //0 代表发声
}

//蜂鸣器关闭
void BeepClose(void)
{
    P3_4=1; //1 代表关闭
}

//蜂鸣器的驱动函数，放在定时中断函数里每定时 1ms 扫描一次。
void VoiceScan(void)
{
    //Su8Lock 的作用是避免 BeepOpen() 被重复扫描影响效率，发声时只执行一次此函数即可。
    //同时，也巧妙借用 else 结构，实现逻辑顺序分解成“先发声，下一次再开始定时”的两个步骤。

    static unsigned char Su8Lock=0;

    if(1==vGu8BeepTimerFlag&&vGu16BeepTimerCnt>0)
    {
        if(0==Su8Lock)
        {
            Su8Lock=1; //进入触发声音后就自锁起来
            BeepOpen(); //发声，此处封装成函数，为了今后代码的移植性。
        }
        else //巧妙借用 else 结构，实现先发声，下一次中断再开始计时的逻辑顺序。比如，
        { //如果赋值 1，就能确保有 1ms 的计时发声。

            vGu16BeepTimerCnt--; //定时器自减，控制蜂鸣器发声的时间长度

            if(0==vGu16BeepTimerCnt)

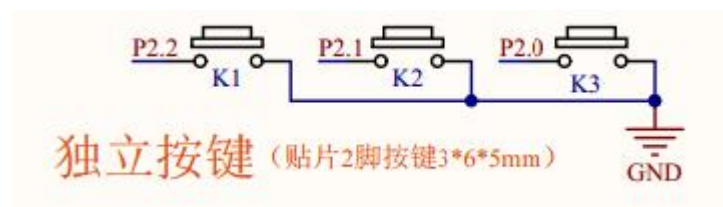
```



```
        {  
            Su8Lock=0;    //关闭声音后，及时解锁，为下一次触发做准备  
            BeepClose();  //关闭声音，此处封装成函数，为了今后代码的移植性。  
        }  
    }  
}
```

## 第九十二节： 独立按键的四大要素（自锁，消抖，非阻塞，清零式滤波）。

### 【92.1 独立按键的硬件电路简介。】



上图 92.1.1 独立按键电路

按键有两种驱动方式，一种是独立按键，一种是矩阵按键。1 个独立按键要占用 1 个 I/O 口，I/O 口不能共用。而矩阵按键的 I/O 口是分时分片选复用的，用少量的 I/O 口就可以驱动翻倍级别的按键数量。比如，用 8 个 I/O 口只能驱动 8 个独立按键，但是却可以驱动 16 个矩阵按键（4x4）。因此，按键少的时候就用独立按键，按键多的时候就用矩阵按键。这两种按键的驱动本质是一样的，都是靠识别输入信号的下降沿（或上升沿）来识别按键的触发。

独立按键的硬件原理基础，如上图，P2.2 这个 I/O 口，在按键 K1 没有被按下的时候，P2.2 口因为单片机内部自带上拉电阻把电平拉高，此时 P2.2 口是高电平的输入状态。当按键 K1 被按下的时候，按键 K1 左右像一根导线连接到电源的负极（GND），直接把原来 P2.2 口的电平拉低，此时 P2.2 口变成了低电平的输入状态。编写按键驱动程序，就是要识别这个电平从高到低的过程，这个过程也叫下降沿。多说一句，51 单片机的 P1, P2, P3 口是内部自带上拉电阻的，而 P0 口是内部没有上拉电阻的，需要外接上拉电阻。除此之外，很多单片机内部其实都没有上拉电阻的，因此，建议大家在独立按键电路的时候，养成一个习惯，凡是按键输入状态都外接上拉电阻。

识别按键的下降沿触发有四大要素：自锁，消抖，非阻塞，清零式滤波。

“自锁”，按键一旦进入到低电平，就要“自锁”起来，避免不断触发按键，只有当按键被松开变成高电平的时候，才及时“解锁”为下一次触发做准备。

“消抖”，按键是一个机械触点器件，在接触的瞬间必然存在微观上的机械抖动，反馈到电平的瞬间就是“高，低，高，低...”这种不稳定的电平状态是一种干扰，但是，按键一旦按下去稳定了之后，这种状态就消失，电平就一直保持稳定的低电平。消抖的本质就是滤波，要把这种接触的瞬间抖动过滤掉，避免按键的“一按多触发”。

“非阻塞”，在处理消抖的时候，必须用到延时，如果此时用阻塞的 delay 延时就会影响其它任务的运行效率，因此，用非阻塞的定时延时更加有优越性。

“清零式滤波”，在消抖的时候，有两种境界，第一种境界是判断两次电平的状态，中间插入“固定的时间”延时，这种方法前后一共判断了两次，第一次是识别到低电平就进入延时的状态，第二次是延时后再确认一次是否继续是低电平的状态，这种方法的不足是，“固定的时间”全凭经验值，但是不同的按键它们的抖动时间长度是不同的，除此之外，前后才判断了两次，在软件的抗干扰能力上也弱了很多，“密码等级”不够高。第二种境界就是“清零式滤波”，“清零式滤波”非常巧妙，抗扰能力超强，它能自动过滤不同按键的“抖动时间”，然后再进入一个“稳定时间”的“N 次识别判断”，更加巧妙的是，在“抖动时间”和“稳定时间”两者时间内，只要发现一次是高电平的干扰，就马上自动清零计时器，重新开始计时。“稳定时间”一般取 20ms 到 30ms 之间，而“抖动时间”是隐藏的，在代码上并没有直接描写出来，但是却无形地融入了代码之中，只有慢慢体会才能发现它的存在。

具体的代码如下，实现的功能是按一次 K1 或者 K2 按键，就触发一次蜂鸣器鸣叫。

```

#include "REG52.H"

#define KEY_VOICE_TIME    50 //按键触发后发出的声音长度
#define KEY_FILTER_TIME  25  //按键滤波的“稳定时间”25ms

void T0_time();
void SystemInitial(void) ;
void Delay(unsigned long u32DelayTime) ;
void PeripheralInitial(void) ;

void BeepOpen(void);
void BeepClose(void);
void VoiceScan(void);
void KeyScan(void);    //按键识别的驱动函数，放在定时中断里
void KeyTask(void);    //按键任务函数，放在主函数内

sbit P3_4=P3^4;
sbit KEY_INPUT1=P2^2; //K1 按键识别的输入口。
sbit KEY_INPUT2=P2^1; //K2 按键识别的输入口。

volatile unsigned char vGu8BeepTimerFlag=0;
volatile unsigned int vGu16BeepTimerCnt=0;

volatile unsigned char vGu8KeySec=0; //按键的触发序号，全局变量意味着是其它函数的接口。

void main()
{
    SystemInitial();
    Delay(10000);
    PeripheralInitial();
    while(1)
    {
        KeyTask();    //按键任务函数
    }
}

void T0_time() interrupt 1
{
    VoiceScan();
    KeyScan();    //按键识别的驱动函数

    TH0=0xfc;
    TL0=0x66;
}

```

```

}

void SystemInitial(void)
{
    TMOD=0x01;
    TH0=0xfc;
    TL0=0x66;
    EA=1;
    ET0=1;
    TR0=1;
}

void Delay(unsigned long u32DelayTime)
{
    for(;u32DelayTime>0;u32DelayTime--);
}

void PeripheralInitial(void)
{
}

void BeepOpen(void)
{
    P3_4=0;
}

void BeepClose(void)
{
    P3_4=1;
}

void VoiceScan(void)
{
    static unsigned char Su8Lock=0;

    if(1==vGu8BeepTimerFlag&&vGu16BeepTimerCnt>0)
    {
        if(0==Su8Lock)
        {
            Su8Lock=1;
            BeepOpen();
        }
    }
}

```

```

    }
    else
    {

        vGu16BeepTimerCnt--;

        if(0==vGu16BeepTimerCnt)
        {
            Su8Lock=0;
            BeepClose();
        }

    }
}
}
}

```

/\* 注释一：

\* 独立按键扫描的详细过程，以按键 K1 为例，如下：

\* 第一步：平时没有按键被触发时，按键的自锁标志，去抖动延时计数器一直被清零。

\* 第二步：一旦有按键被按下，去抖动延时计数器开始在定时中断函数里累加，在还没累加到

\* 阈值 KEY\_FILTER\_TIME 时，如果在这期间由于受外界干扰或者按键抖动，而使

\* IO 口突然瞬间触发成高电平，这个时候马上把延时计数器 Su16KeyCnt1 清零了，这个过程

\* 非常巧妙，非常有效地去除瞬间的杂波干扰。以后凡是用到开关感应器的时候，

\* 都可以用类似这样的方法去干扰。

\* 第三步：如果按键按下的时间达到阈值 KEY\_FILTER\_TIME 时，则触发按键，把编号 vGu8KeySec 赋值。

\* 同时，马上把自锁标志 Su8KeyLock1 置 1，防止按住按键不松手后一直触发。

\* 第四步：等按键松开后，自锁标志 Su8KeyLock1 及时清零（解锁），为下一次自锁做准备。

\* 第五步：以上整个过程，就是识别按键 IO 口下降沿触发的过程。

\*/

void KeyScan(void) //此函数放在定时中断里每 1ms 扫描一次

```

{
    static unsigned char Su8KeyLock1; //1 号按键的自锁
    static unsigned int Su16KeyCnt1; //1 号按键的计时器
    static unsigned char Su8KeyLock2; //2 号按键的自锁
    static unsigned int Su16KeyCnt2; //2 号按键的计时器

    //1 号按键
    if(0!=KEY_INPUT1)//IO 是高电平，说明按键没有被按下，这时要及时清零一些标志位
    {
        Su8KeyLock1=0; //按键解锁
        Su16KeyCnt1=0; //按键去抖动延时计数器清零，此行非常巧妙，是全场的亮点。
    }
    else if(0==Su8KeyLock1)//有按键按下，且是第一次被按下。这行很多初学者有疑问，请看专题分析。
    {

```

```

    Su16KeyCnt1++; //累加定时中断次数
    if (Su16KeyCnt1>=KEY_FILTER_TIME) //滤波的“稳定时间” KEY_FILTER_TIME，长度是 25ms。
    {
        Su8KeyLock1=1; //按键的自锁, 避免一直触发
        vGu8KeySec=1;   //触发 1 号键
    }
}

//2 号按键
if (0!=KEY_INPUT2)
{
    Su8KeyLock2=0;
    Su16KeyCnt2=0;
}
else if (0==Su8KeyLock2)
{
    Su16KeyCnt2++;
    if (Su16KeyCnt2>=KEY_FILTER_TIME)
    {
        Su8KeyLock2=1;
        vGu8KeySec=2;   //触发 2 号键
    }
}
}

void KeyTask(void)    //按键任务函数，放在主函数内
{
    if (0==vGu8KeySec)
    {
        return; //按键的触发序号是 0 意味着无按键触发，直接退出当前函数，不执行此函数下面的代码
    }

    switch(vGu8KeySec) //根据不同的按键触发序号执行对应的代码
    {
        case 1:        //1 号按键

            vGu8BeepTimerFlag=0;
            vGu16BeepTimerCnt=KEY_VOICE_TIME; //触发按键后，发出固定长度的声音
            vGu8BeepTimerFlag=1;
            vGu8KeySec=0; //响应按键服务处理程序后，按键编号必须清零，避免一直触发
            break;
    }
}

```

```

        case 2:      //2 号按键

            vGu8BeepTimerFlag=0;
            vGu16BeepTimerCnt=KEY_VOICE_TIME; //触发按键后，发出固定长度的声音
            vGu8BeepTimerFlag=1;
            vGu8KeySec=0; //响应按键服务处理程序后，按键编号必须清零，避免一直触发
            break;

    }
}

```

## 【92.2 专题分析：else if(0==Su8KeyLock1)】

疑问：

```

if(0!=KEY_INPUT1)
{
    Su8KeyLock1=0;
    Su16KeyCnt1=0;
}
else if(0==Su8KeyLock1)//有按键按下，且是第一次被按下。为什么？为什么？为什么？
{
    Su16KeyCnt1++;
    if(Su16KeyCnt1>KEY_FILTER_TIME)
    {
        Su8KeyLock1=1;
        vGu8KeySec=1;
    }
}
}

```

解答：

首先，我们要明白 C 语言的语法中，

```

if(条件 1)
{

}

else if(条件 2)
{

}

```

以上语句是一对组合语句，不能分开来看。当（条件 1）成立的时候，它是绝对不会判断（条件 2）的。当（条件 1）不成立的时候，才会判断（条件 2）。

回到刚才的问题，当程序执行到（条件 2）else if(0==Su8KeyLock1)的时候，就已经默认了（条件 1）if(0!=KEY\_INPUT1)不成立，这个条件不成立，就意味着 0==KEY\_INPUT1，也就是有按键被按下，因此，这里的 else if(0==Su8KeyLock1)等效于 else if(0==Su8KeyLock1&&0==KEY\_INPUT1)，而 Su8KeyLock1 是一个自

锁标志位，一旦按键被触发后，这个标志位会变 1，防止按键按住不松手的时候不断触发按键。这样，按键只能按一次触发一次，松开手后再按一次，又触发一次。

### 【92.3 专题分析：if(0!=KEY\_INPUT1)】

疑问：为什么不用 if(1==KEY\_INPUT1) 而用 if(0!=KEY\_INPUT1)？

解答：其实两者在功能上是完全等效的，在这里都可以用。之所以本教程优先选用后者 if(0!=KEY\_INPUT1)，是因为考虑到了代码在不同单片机平台上的可移植性和兼容性。很多 32 位的单片机提供的是库函数，库函数返回的按键状态是一个字节变量来表示，当被按下的时候是 0，但是，当没有按下的时候并不一定等于 1，而是一个“非 0”的数值。

### 【92.4 专题分析：把 KeyScan 函数放在定时器中断里】

疑问：为什么把 KeyScan 函数放在定时器中断里？

解答：中断函数里放的函数或者代码越少越好，但是 KeyScan 函数是特殊的函数，是涉及到 IO 口输入信号的滤波，滤波就涉及到时间的及时性与均匀性，放在定时中断函数里更加能保证时间的一致性。比如，蜂鸣器驱动，动态数码管驱动，按键扫描驱动，我个人都习惯放在定时中断函数里。

### 【92.5 专题分析：if(0==vGu8KeySec)return】

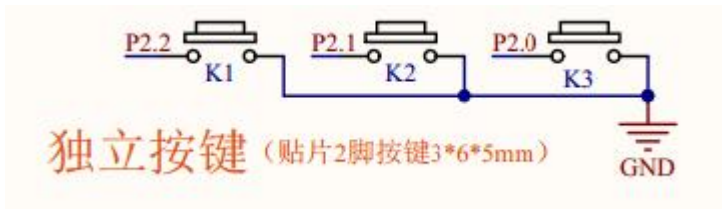
疑问：if(0==vGu8KeySec)return 是不是多此一举？

解答：在 KeyTask 函数这里，if(0==vGu8KeySec)return 这行代码删掉，对程序功能是没有影响的，这里之所以多插入这行判断语句，是因为，当按键多达几十个的时候，避免主函数每次进入 KeyTask 函数，都挨个扫描判断 switch 的状态进行多次判断，如果增加了这行 if(0==vGu8KeySec)return 代码，就可以直接退出省事，在理论上感觉更加运行高效。其实，不同单片机不同的 C 编译器可能对 switch 语句的翻译不一样，因此，这里的是不是更加高效我不敢保证。但是可以保证的是，加了这行代码也没有其它副作用。

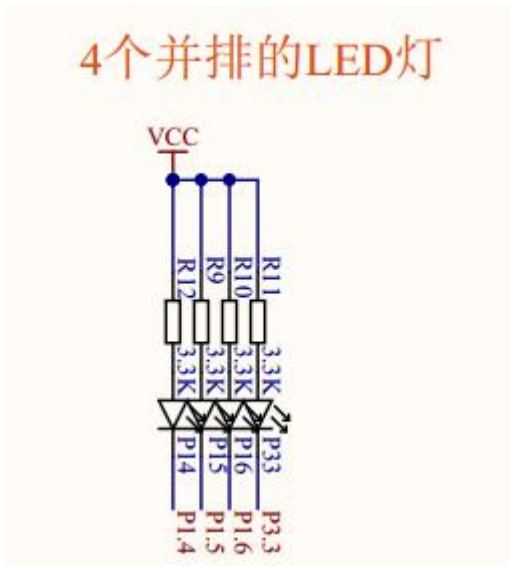


第九十三节： 独立按键鼠标式的单击与双击。

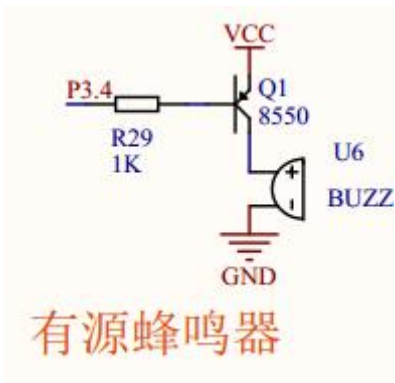
【93.1 鼠标式的单击与双击。】



上图 93. 1. 1 独立按键电路



上图 93. 1. 2 LED 电路



上图 93. 1. 3 有源蜂鸣器电路

鼠标的左键，可以触发单击，也可以触发双击。双击的规则是这样的，两次单击，如果第 1 次单击与第 2 次单击的时间比较“短”的时候，则这两次单击就构成双击。编写这个程序的最大亮点是如何控制好第 1

次单击与第 2 次单击的时间间隔。

程序例程要实现的功能是：(1) 单击改变 LED 灯的显示状态，单击一次 LED 从原来“灭”的状态变成“亮”的状态，或者从原来“亮”的状态变成“灭”的状态，依次循环切换。(2) 双击则蜂鸣器发出“嘀”的一声。代码如下：

```
#include "REG52.H"

#define KEY_VOICE_TIME    50      //按键触发后发出的声音长度
#define KEY_FILTER_TIME   25      //按键滤波的“稳定时间”25ms
#define KEY_INTERVAL_TIME 250    //连续两次单击之间的最大有效时间 250ms

void T0_time();
void SystemInitial(void) ;
void Delay(unsigned long u32DelayTime) ;
void PeripheralInitial(void) ;

void BeepOpen(void);
void BeepClose(void);
void LedOpen(void);
void LedClose(void);

void VoiceScan(void);
void KeyScan(void);    //按键识别的驱动函数，放在定时中断里
void SingleKeyTask(void); //单击按键任务函数，放在主函数内
void DoubleKeyTask(void); //双击按键任务函数，放在主函数内

sbit P3_4=P3^4;        //蜂鸣器
sbit P1_4=P1^4;        //LED

sbit KEY_INPUT1=P2^2;  //K1 按键识别的输入口。

volatile unsigned char vGu8BeepTimerFlag=0;
volatile unsigned int vGu16BeepTimerCnt=0;

unsigned char Gu8LedStatus=0; //记录 LED 灯的状态，0 代表灭，1 代表亮
volatile unsigned char vGu8SingleKeySec=0; //单击按键的触发序号
volatile unsigned char vGu8DoubleKeySec=0; //双击按键的触发序号

void main()
{
    SystemInitial();
    Delay(10000);
    PeripheralInitial();
    while(1)
```

```

    {
        SingleKeyTask();    //单击按键任务函数
        DoubleKeyTask();    //双击按键任务函数
    }
}

void TO_time() interrupt 1
{
    VoiceScan();
    KeyScan();    //按键识别的驱动函数

    TH0=0xfc;
    TL0=0x66;
}

void SystemInitial(void)
{
    TMOD=0x01;
    TH0=0xfc;
    TL0=0x66;
    EA=1;
    ET0=1;
    TR0=1;
}

void Delay(unsigned long u32DelayTime)
{
    for(;u32DelayTime>0;u32DelayTime--);
}

void PeripheralInitial(void)
{
    /* 注释一：
    * 把 LED 的初始化放在 PeripheralInitial 而不是放在 SystemInitial，是因为 LED 显示内容对上电
    * 瞬间的要求不高。但是，如果是控制继电器，则应该把继电器的输出初始化放在 SystemInitial。
    */

    //根据 Gu8LedStatus 的值来初始化 LED 当前的显示状态，0 代表灭，1 代表亮
    if(0==Gu8LedStatus)
    {
        LedClose();    //关闭 LED
    }
    else

```

```

    {
        LedOpen();    //点亮 LED
    }
}

void BeepOpen(void)
{
    P3_4=0;
}

void BeepClose(void)
{
    P3_4=1;
}

void LedOpen(void)
{
    P1_4=0;
}

void LedClose(void)
{
    P1_4=1;
}

void VoiceScan(void)
{
    static unsigned char Su8Lock=0;

    if(1==vGu8BeepTimerFlag&&vGu16BeepTimerCnt>0)
    {
        if(0==Su8Lock)
        {
            Su8Lock=1;
            BeepOpen();
        }
        else
        {
            vGu16BeepTimerCnt--;

            if(0==vGu16BeepTimerCnt)
            {
                Su8Lock=0;
            }
        }
    }
}

```

```

        BeepClose();
    }

}

}

}

}

/* 注释二：
* 双击按键扫描的详细过程：
* 第一步：平时没有按键被触发时，按键的自锁标志，去抖动延时计数器一直被清零。
*     如果之前已经有按键触发过 1 次单击，那么启动时间间隔计数器 Su16KeyIntervalCnt1，
*     在 KEY_INTERVAL_TIME 这个允许的时间差范围内，如果一直没有第 2 次单击触发，
*     则把累加按键触发的次数 Su8KeyTouchCnt1 也清零，上一次累计的单击数被清零，
*     就意味着下一次新的双击必须重新开始累加两次单击数。
* 第二步：一旦有按键被按下，去抖动延时计数器开始在定时中断函数里累加，在还没累加到
*     阈值 KEY_FILTER_TIME 时，如果在这期间由于受外界干扰或者按键抖动，而使
*     IO 口突然瞬间触发成高电平，这个时候马上把延时计数器 Su16KeyTimeCnt1
*     清零了，这个过程非常巧妙，非常有效地去除瞬间的杂波干扰，以后凡是用到开关感应器的时候，
*     都可以用类似这样的方法去干扰。
* 第三步：如果按键按下的时间超过了阈值 KEY_FILTER_TIME，马上把自锁标志 Su8KeyLock1 置 1，
*     防止按住按键不松手后一直触发。与此同时，累加 1 次按键次数，如果按键次数累加有 2 次，
*     则认为触发双击按键，并把编号 vGu8DoubleKeySec 赋值。
* 第四步：等按键松开后，自锁标志 Su8KeyLock1 及时清零解锁，为下一次自锁做准备。并且累加间隔时间，
*     防止两次按键的间隔时间太长。如果连续 2 次单击的间隔时间太长达到了 KEY_INTERVAL_TIME
*     的长度，立即清零当前按键次数的计数器，这样意味着上一次的累加单击数无效，下一次双击
*     必须重新累加新的单击数。
*/

void KeyScan(void) //此函数放在定时中断里每 1ms 扫描一次
{
    static unsigned char Su8KeyLock1;           //1 号按键的自锁
    static unsigned int  Su16KeyCnt1;           //1 号按键的计时器
    static unsigned char Su8KeyTouchCnt1;       //1 号按键的次数记录
    static unsigned int  Su16KeyIntervalCnt1;   //1 号按键的间隔时间计数器

    //1 号按键
    if(0!=KEY_INPUT1)//IO 是高电平，说明按键没有被按下，这时要及时清零一些标志位
    {
        Su8KeyLock1=0; //按键解锁
        Su16KeyCnt1=0; //按键去抖动延时计数器清零，此行非常巧妙。
        if (Su8KeyTouchCnt1>=1) //之前已经有按键触发过一次，启动间隔时间的计数器
        {
            Su16KeyIntervalCnt1++; //按键间隔的时间计数器累加
            if(Su16KeyIntervalCnt1>=KEY_INTERVAL_TIME) //达到最大允许的间隔时间，溢出无效

```

```

        {
            Su16KeyIntervalCnt1=0; //时间计数器清零
            Su8KeyTouchCnt1=0;      //清零按键的按下的次数
        }
    }
}
else if(0==Su8KeyLock1)//有按键按下，且是第一次被按下。此行如有疑问，请看第 92 节的讲解。
{
    Su16KeyCnt1++; //累加定时中断次数
    if(Su16KeyCnt1>=KEY_FILTER_TIME) //滤波的“稳定时间” KEY_FILTER_TIME，长度是 25ms。
    {
        Su8KeyLock1=1; //按键的自锁, 避免一直触发
        Su16KeyIntervalCnt1=0; //按键有效间隔的时间计数器清零
        Su8KeyTouchCnt1++; //记录当前单击的次数
        if(1==Su8KeyTouchCnt1) //只按了 1 次
        {
            vGu8SingleKeySec=1; //单击任务
        }
        else if(Su8KeyTouchCnt1>=2) //连续按了两次以上
        {
            Su8KeyTouchCnt1=0; //统计按键次数清零
            vGu8SingleKeySec=1; //单击任务
            vGu8DoubleKeySec=1; //双击任务
        }
    }
}
}

void SingleKeyTask(void) //单击按键任务函数，放在主函数内
{
    if(0==vGu8SingleKeySec)
    {
        return; //按键的触发序号是 0 意味着无按键触发，直接退出当前函数，不执行此函数下面的代码
    }

    switch(vGu8SingleKeySec) //根据不同的按键触发序号执行对应的代码
    {
        case 1: //单击任务
            //通过 Gu8LedStatus 的状态切换，来反复切换 LED 的“灭”与“亮”的状态
            if(0==Gu8LedStatus)
            {
                Gu8LedStatus=1; //标识并且更改当前 LED 灯的状态。0 就变成 1。
                LedOpen(); //点亮 LED
            }
        }
    }
}

```

```

    }
    else
    {
        Gu8LedStatus=0; //标识并且更改当前 LED 灯的状态。1 就变成 0。
        LedClose(); //关闭 LED
    }

    vGu8SingleKeySec=0; //响应按键服务处理程序后，按键编号必须清零，避免一直触发
    break;
}

}

void DoubleKeyTask(void) //双击按键任务函数，放在主函数内
{
    if(0==vGu8DoubleKeySec)
    {
        return; //按键的触发序号是 0 意味着无按键触发，直接退出当前函数，不执行此函数下面的代码
    }

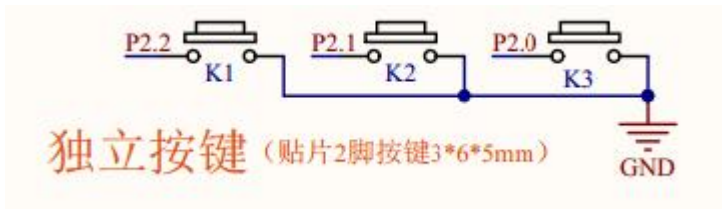
    switch(vGu8DoubleKeySec) //根据不同的按键触发序号执行对应的代码
    {
        case 1: //双击任务

            vGu8BeepTimerFlag=0;
            vGu16BeepTimerCnt=KEY_VOICE_TIME; //触发双击后，发出“嘀”一声
            vGu8BeepTimerFlag=1;
            vGu8DoubleKeySec=0; //响应按键服务处理程序后，按键编号必须清零，避免一直触发
            break;
    }
}

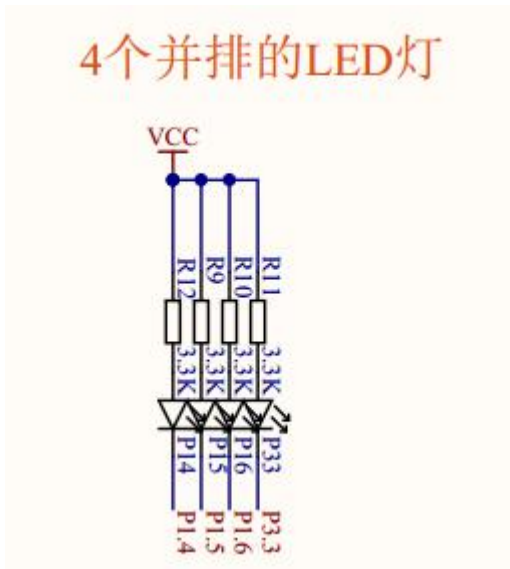
```

第九十四节： 两个独立按键构成的组合按键。

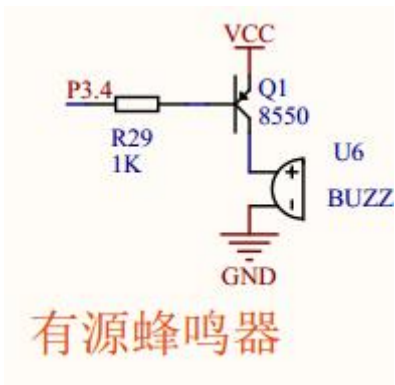
【94.1 组合按键。】



上图 94. 1. 1 独立按键电路



上图 94. 1. 2 LED 电路



上图 94. 1. 3 有源蜂鸣器电路

组合按键的触发，是指两个按键同时按下时的“非单击”触发。一次组合按键的产生，必然包含了三类按键的触发。比如，K1 与 K2 两个独立按键，当它们产生一次组合按键的操作时，就包含了三类触发：K1 单



击触发，K2 单击触发，K1 与 K2 的组合触发。这三类触发可以看作是底层的按键驱动程序，在按键应用层的任务函数 SingleKeyTask 和 CombinationKeyTask 中，可以根据项目的实际需要进行响应。本节程序例程要实现的功能是：（1）K1 单击让 LED 变成“亮”的状态。（2）K2 单击让 LED 变成“灭”的状态。（3）K1 与 K2 的组合按键触发让蜂鸣器发出“嘀”的一声。代码如下：

```
#include "REG52.H"

#define KEY_VOICE_TIME    50      //组合按键触发后发出的声音长度
#define KEY_FILTER_TIME  25      //按键滤波的“稳定时间”25ms

void T0_time();
void SystemInitial(void) ;
void Delay(unsigned long u32DelayTime) ;
void PeripheralInitial(void) ;

void BeepOpen(void);
void BeepClose(void);
void LedOpen(void);
void LedClose(void);

void VoiceScan(void);
void KeyScan(void);    //按键识别的驱动函数，放在定时中断里
void SingleKeyTask(void);    //单击按键任务函数，放在主函数内
void CombinationKeyTask(void);    //组合按键任务函数，放在主函数内

sbit P3_4=P3^4;        //蜂鸣器
sbit P1_4=P1^4;        //LED

sbit KEY_INPUT1=P2^2;  //K1 按键识别的输入口。
sbit KEY_INPUT2=P2^1;  //K2 按键识别的输入口。

volatile unsigned char vGu8BeepTimerFlag=0;
volatile unsigned int vGu16BeepTimerCnt=0;

volatile unsigned char vGu8SingleKeySec=0; //单击按键的触发序号
volatile unsigned char vGu8CombinationKeySec=0; //组合按键的触发序号

void main()
{
    SystemInitial();
    Delay(10000);
    PeripheralInitial();
    while(1)
    {
```

```

        CombinationKeyTask(); //组合按键任务函数
        SingleKeyTask();      //单击按键任务函数
    }
}

void T0_time() interrupt 1
{
    VoiceScan();
    KeyScan(); //按键识别的驱动函数

    TH0=0xfc;
    TL0=0x66;
}

void SystemInitial(void)
{
    TMOD=0x01;
    TH0=0xfc;
    TL0=0x66;
    EA=1;
    ET0=1;
    TR0=1;
}

void Delay(unsigned long u32DelayTime)
{
    for(;u32DelayTime>0;u32DelayTime--);
}

void PeripheralInitial(void)
{
    LedClose(); //初始化关闭 LED
}

void BeepOpen(void)
{
    P3_4=0;
}

void BeepClose(void)
{
    P3_4=1;
}

```

```

void LedOpen(void)
{
    P1_4=0;
}

void LedClose(void)
{
    P1_4=1;
}

void VoiceScan(void)
{
    static unsigned char Su8Lock=0;

    if(1==vGu8BeepTimerFlag&&vGu16BeepTimerCnt>0)
    {
        if(0==Su8Lock)
        {
            Su8Lock=1;
            BeepOpen();
        }
        else
        {

            vGu16BeepTimerCnt--;

            if(0==vGu16BeepTimerCnt)
            {
                Su8Lock=0;
                BeepClose();
            }

        }
    }
}

```

/\* 注释一：

\* 组合按键扫描的详细过程：

\* 第一步：平时只要 K1 与 K2 两个按键中有一个没有被按下时，按键的自锁标志，去抖动延时计数器一直被清零。

\* 第二步：一旦两个按键都处于被按下的状态，去抖动延时计数器开始在定时中断函数里累加，在还没累加到阈值 KEY\_FILTER\_TIME 时，如果在这期间由于受外界干扰或者按键抖动，而使其中一个

```

*      IO 口突然瞬间触发成高电平，这个时候马上把延时计数器 Su16CombinationKeyTimeCnt
*      清零了, 这个过程非常巧妙，非常有效地去除瞬间的杂波干扰。
* 第三步：如果两个按键按下的时间超过了阈值 KEY_FILTER_TIME，马上把自锁标志 Su8CombinationKeyLock
*      置 1，防止按住两个按键不松手后一直触发。并把按键编号 vGu8CombinationKeySec 赋值，
*      触发一次组合按键。
* 第四步：等其中一个按键松开后，自锁标志 Su8CombinationKeyLock 及时清零，为下一次自锁做准备。
*/

void KeyScan(void)  //此函数放在定时中断里每 1ms 扫描一次
{
    static unsigned char Su8KeyLock1;
    static unsigned int  Su16KeyCnt1;
    static unsigned char Su8KeyLock2;
    static unsigned int  Su16KeyCnt2;

    static unsigned char Su8CombinationKeyLock;  //组合按键的自锁
    static unsigned int  Su16CombinationKeyCnt;  //组合按键的计时器

    //K1 按键与 K2 按键的组合触发
    if(0!=KEY_INPUT1 || 0!=KEY_INPUT2) //两个按键只要有一个按键没有按下，处于“非组合按键”的状态。
    {
        Su8CombinationKeyLock=0; //组合按键解锁
        Su16CombinationKeyCnt=0;  //组合按键去抖动延时计数器清零，此行非常巧妙，是全场的亮点。
    }
    else if(0==Su8CombinationKeyLock) //两个按键被同时按下，且是第一次被按下。此行请看专题分析。
    {
        Su16CombinationKeyCnt++; //累加定时中断次数
        if(Su16CombinationKeyCnt>=KEY_FILTER_TIME) //滤波的“稳定时间”KEY_FILTER_TIME。
        {
            Su8CombinationKeyLock=1;  //组合按键的自锁, 避免一直触发
            vGu8CombinationKeySec=1;   //触发 K1 与 K2 的组合键操作
        }
    }

    //K1 按键的单击
    if(0!=KEY_INPUT1)
    {
        Su8KeyLock1=0;
        Su16KeyCnt1=0;
    }
    else if(0==Su8KeyLock1)
    {
        Su16KeyCnt1++;
    }
}

```

```

        if(Su16KeyCnt1>=KEY_FILTER_TIME)
        {
            Su8KeyLock1=1;
            vGu8SingleKeySec=1;    //触发 K1 的单击键
        }
    }

    //K2 按键的单击
    if(0!=KEY_INPUT2)
    {
        Su8KeyLock2=0;
        Su16KeyCnt2=0;
    }
    else if(0==Su8KeyLock2)
    {
        Su16KeyCnt2++;
        if(Su16KeyCnt2>=KEY_FILTER_TIME)
        {
            Su8KeyLock2=1;
            vGu8SingleKeySec=2;    //触发 K2 的单击键
        }
    }
}

void CombinationKeyTask(void)    //组合按键任务函数，放在主函数内
{
    if(0==vGu8CombinationKeySec)
    {
        return; //按键的触发序号是 0 意味着无按键触发，直接退出当前函数，不执行此函数下面的代码
    }

    switch(vGu8CombinationKeySec) //根据不同的按键触发序号执行对应的代码
    {
        case 1:    //K1 与 K2 的组合按键任务

            vGu8BeepTimerFlag=0;
            vGu16BeepTimerCnt=KEY_VOICE_TIME; //触发一次组合按键后，发出“嘀”一声
            vGu8BeepTimerFlag=1;
            vGu8CombinationKeySec=0; //响应按键服务处理程序后，按键编号必须清零，避免一直触发
            break;
    }
}

```

```

void SingleKeyTask(void)    //单击按键任务函数，放在主函数内
{
    if(0==vGu8SingleKeySec)
    {
        return; //按键的触发序号是 0 意味着无按键触发，直接退出当前函数，不执行此函数下面的代码
    }

    switch(vGu8SingleKeySec) //根据不同的按键触发序号执行对应的代码
    {
        case 1:    //K1 单击任务
            LedOpen();    //LED 亮

            vGu8SingleKeySec=0; //响应按键服务处理程序后，按键编号必须清零，避免一直触发
            break;

        case 2:    //K2 单击任务
            LedClose();    //LED 灭

            vGu8SingleKeySec=0; //响应按键服务处理程序后，按键编号必须清零，避免一直触发
            break;

    }
}

```

## 【94.2 专题分析：else if(0==Su8CombinationKeyLock)。】

疑问：

```

if(0!=KEY_INPUT1 || 0!=KEY_INPUT2)
{
    Su8CombinationKeyLock=0;
    Su16CombinationKeyCnt=0;
}
else if(0==Su8CombinationKeyLock) //两个按键被同时按下，且是第一次被按下。为什么？
{
    Su16CombinationKeyCnt++;
    if(Su16CombinationKeyCnt>=KEY_FILTER_TIME)
    {
        Su8CombinationKeyLock=1;
        vGu8CombinationKeySec=1;
    }
}
}

```

解答：

首先，我们要明白 C 语言的语法中，

```
if(条件 1)
{

}
else if(条件 2)
{

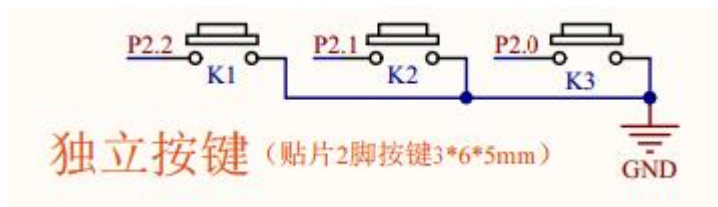
}
```

以上语句是一对组合语句，不能分开来看。当（条件 1）成立的时候，它是绝对不会判断（条件 2）的。当（条件 1）不成立的时候，才会判断（条件 2）。

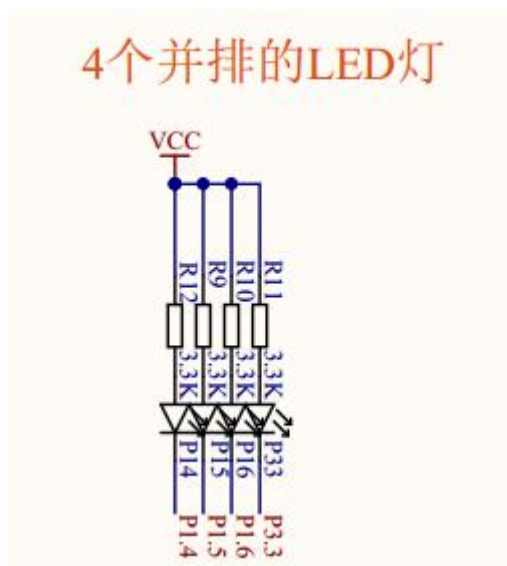
回到刚才的问题，当程序执行到（条件 2）`else if(0==Su8CombinationKeyLock)`的时候，就已经默认了（条件 1）`if(0!=KEY_INPUT1||0!=KEY_INPUT2)`不成立，这个条件不成立，就意味着 `0==KEY_INPUT1` 和 `0==KEY_INPUT2`，也就是有两个按键被同时按下，因此，这里的 `else if(0==Su8CombinationKeyLock)` 等效于 `else if(0==Su8CombinationKeyLock&&0==KEY_INPUT1&&0==KEY_INPUT2)`，而 `Su8CombinationKeyLock` 是一个自锁标志位，一旦组合按键被触发后，这个标志位会变 1，防止两个按键按住不松手的时候不断触发组合按键。这样，组合按键只能同时按下一次触发一次，任意松开其中一个按键后再同时按下一次两个按键，又触发一次新的组合按键。

第九十五节： 两个独立按键的“电脑键盘式”组合按键。

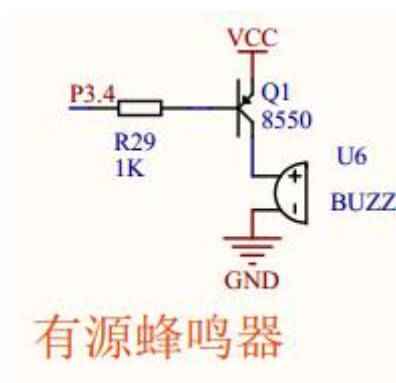
【95.1 “电脑键盘式”组合按键。】



上图 95.1.1 独立按键电路



上图 95.1.2 LED 电路



上图 95.1.3 有源蜂鸣器电路

上一节也讲了由 K1 和 K2 构成的组合按键，但是这种组合按键是普通的组合按键，因为它们的 K1 和 K2 是不分先后顺序的，你先按住 K1 然后再按 K2，或者你先按住 K2 然后再按 K1，效果都一样。本节讲的组合



按键是分先后顺序的，比如，像电脑的复制快捷键（Ctrl+C），你必须先按住 Ctrl 再按住 C 此时“复制快捷键”才有效，如果你先按住 C 再按住 Ctrl 此时“复制快捷键”无效。本节讲的例程就是要实现这个功能，用 K1 代表 C 这类“字符数字键”，用 K2 代表 Ctrl 这类“辅助按键”，因此，要触发组合键（K2+K1），必须先按住 K2 再按 K1 才有效。本节讲的例程功能如下：（1）K1 每单击一次，LED 要么从“灭”变成“亮”，要么从“亮”变成“灭”，在两种状态之间切换。（2）如果先按住 K2 再按 K1，就认为构造了“电脑键盘式”组合键，蜂鸣器发出“嘀”的一声。代码如下：

```
#include "REG52.H"

#define KEY_VOICE_TIME    50      //组合按键触发后发出的声音长度 50ms
#define KEY_FILTER_TIME   25      //按键滤波的“稳定时间” 25ms

void T0_time();
void SystemInitial(void) ;
void Delay(unsigned long u32DelayTime) ;
void PeripheralInitial(void) ;

void BeepOpen(void);
void BeepClose(void);
void LedOpen(void);
void LedClose(void);

void VoiceScan(void);
void KeyScan(void);    //按键识别的驱动函数，放在定时中断里
void SingleKeyTask(void);    //单击按键任务函数，放在主函数内
void CombinationKeyTask(void);    //组合按键任务函数，放在主函数内

sbit P3_4=P3^4;        //蜂鸣器
sbit P1_4=P1^4;        //LED

sbit KEY_INPUT1=P2^2;  //K1 按键识别的输入口。
sbit KEY_INPUT2=P2^1;  //K2 按键识别的输入口。

volatile unsigned char vGu8BeepTimerFlag=0;
volatile unsigned int vGu16BeepTimerCnt=0;

unsigned char Gu8LedStatus=0; //记录 LED 灯的状态，0 代表灭，1 代表亮

volatile unsigned char vGu8SingleKeySec=0;    //单击按键的触发序号
volatile unsigned char vGu8CombinationKeySec=0;    //组合按键的触发序号

void main()
{
    SystemInitial();
```

```

    Delay(10000);
    PeripheralInitial();
    while(1)
    {
        CombinationKeyTask(); //组合按键任务函数
        SingleKeyTask();      //单击按键任务函数
    }
}

void TO_time() interrupt 1
{
    VoiceScan();
    KeyScan(); //按键识别的驱动函数

    TH0=0xfc;
    TL0=0x66;
}

void SystemInitial(void)
{
    TMOD=0x01;
    TH0=0xfc;
    TL0=0x66;
    EA=1;
    ET0=1;
    TR0=1;
}

void Delay(unsigned long u32DelayTime)
{
    for(;u32DelayTime>0;u32DelayTime--);
}

void PeripheralInitial(void)
{
    if(0==Gu8LedStatus)
    {
        LedClose();
    }
    else
    {
        LedOpen();
    }
}

```

```

}

void BeepOpen(void)
{
    P3_4=0;
}

void BeepClose(void)
{
    P3_4=1;
}

void LedOpen(void)
{
    P1_4=0;
}

void LedClose(void)
{
    P1_4=1;
}

void VoiceScan(void)
{
    static unsigned char Su8Lock=0;

    if(1==vGu8BeepTimerFlag&&vGu16BeepTimerCnt>0)
    {
        if(0==Su8Lock)
        {
            Su8Lock=1;
            BeepOpen();
        }
        else
        {
            vGu16BeepTimerCnt--;

            if(0==vGu16BeepTimerCnt)
            {
                Su8Lock=0;
                BeepClose();
            }
        }
    }
}

```

```

    }
}

/* 注释一：
* “电脑键盘式”组合按键扫描的详细过程：
* 第一步：K2 与 K1 构成的组合按键触发是融合在 K1 单击按键程序里的，只需稍微更改一下 K1 单击的程序
*      ，就可以兼容到 K2 与 K1 构成的“电脑键盘式”组合按键。平时只要 K1 没有被按下时，按
*      键的自锁标志 Su8KeyLock1 和去抖动延时计数器 Su16KeyCnt1 一直被清零。
* 第二步：一旦 K1 按键被按下，去抖动延时计数器 Su16KeyCnt1 开始在定时中断函数里累加，在还没
*      累加到阈值 KEY_FILTER_TIME 时，如果在这期间由于受外界干扰或者按键抖动，而使
*      IO 口突然瞬间触发成高电平，这个时候马上把延时计数器 Su16KeyCnt1 清零了，
*      这个过程非常巧妙，非常有效地去除瞬间的杂波干扰。
* 第三步：如果 K1 按键按下的时间超过了阈值 KEY_FILTER_TIME，马上把自锁标志 Su8KeyLock1 置 1，
*      防止按住按键不松手后一直触发，此时才开始判断一次 K2 按键的电平状态，如果 K2 为低电
*      平就认为是组合按键，并给按键编号 vGu8CombinationKeySec 赋值，否则，就认为是 K1 的单击
*      按键，并给按键编号 vGu8SingleKeySec 赋值。
* 第四步：等 K1 按键松开后，自锁标志 Su8KeyLock1 及时清零，为下一次自锁做准备。
*/

void KeyScan(void) //此函数放在定时中断里每 1ms 扫描一次
{
    static unsigned char Su8KeyLock1;
    static unsigned int  Su16KeyCnt1;

    //K1 的单击，或者 K2 与 K1 构成的“电脑键盘式组合按键”。
    if(0!=KEY_INPUT1)//单个 K1 按键没有按下，及时清零一些标志。
    {
        Su8KeyLock1=0; //按键解锁
        Su16KeyCnt1=0; //去抖动延时计数器清零，此行非常巧妙，是全场的亮点。
    }
    else if(0==Su8KeyLock1)//单个按键 K1 被按下
    {
        Su16KeyCnt1++; //累加定时中断次数
        if(Su16KeyCnt1>=KEY_FILTER_TIME) //滤波的“稳定时间”KEY_FILTER_TIME。
        {
            if(0==KEY_INPUT2) //此时才开始判断一次 K2 的电平状态，为低电平则是组合按键。
            {
                Su8KeyLock1=1;
                vGu8CombinationKeySec=1; //组合按键的触发
            }
            else

```

```

        {
            Su8KeyLock1=1;
            vGu8SingleKeySec=1;    //K1 单击按键的触发
        }
    }
}

void CombinationKeyTask(void)    //组合按键任务函数，放在主函数内
{
    if(0==vGu8CombinationKeySec)
    {
        return; //按键的触发序号是 0 意味着无按键触发，直接退出当前函数，不执行此函数下面的代码
    }

    switch(vGu8CombinationKeySec) //根据不同的按键触发序号执行对应的代码
    {
        case 1:    //K1 与 K2 的组合按键任务

            vGu8BeepTimerFlag=0;
            vGu16BeepTimerCnt=KEY_VOICE_TIME; //触发一次组合按键后，发出“嘀”一声
            vGu8BeepTimerFlag=1;
            vGu8CombinationKeySec=0; //响应按键服务处理程序后，按键编号必须清零，避免一直触发
            break;
    }
}

void SingleKeyTask(void)    //单击按键任务函数，放在主函数内
{
    if(0==vGu8SingleKeySec)
    {
        return; //按键的触发序号是 0 意味着无按键触发，直接退出当前函数，不执行此函数下面的代码
    }

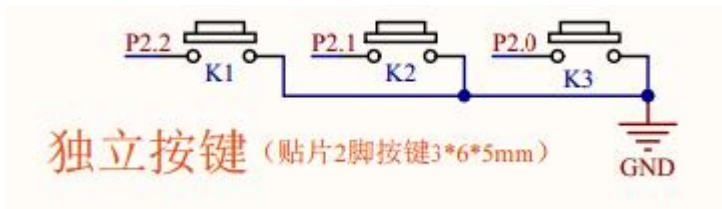
    switch(vGu8SingleKeySec) //根据不同的按键触发序号执行对应的代码
    {
        case 1:    //K1 单击任务
            if(0==Gu8LedStatus)
            {
                Gu8LedStatus=1;
                LedOpen();    //LED 亮
            }
            else

```

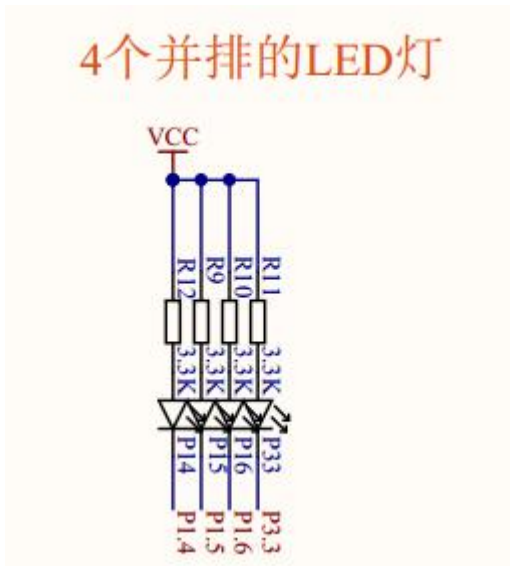
```
{  
    Gu8LedStatus=0;  
    LedClose();    //LED 灭  
}  
  
vGu8SingleKeySec=0; //响应按键服务处理程序后，按键编号必须清零，避免一直触发  
break;  
}  
}
```

第九十六节： 独立按键“一键两用”的短按与长按。

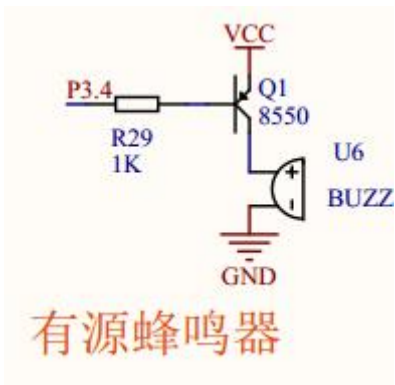
【96.1 “一键两用”的短按与长按。】



上图 96. 1. 1 独立按键电路



上图 96. 1. 2 LED 电路



上图 96. 1. 3 有源蜂鸣器电路

某些项目，当外部按键的资源比较少的时候，一个按键也可以“一键多用”。“一键多用”有很多种玩法，比如，谍战片的无线电通信，依赖一个按键的“不同敲击频率”就可以发送内容丰富的情报。本节“一键两

用”也是属于“一键多用”的众多玩法之一。“短按与长按”的原理是依赖“按键按下的时间长度”来区分识别。“短按”是指从按下的“下降沿”到松手的“上升沿”时间，“长按”是指从按下的“下降沿”到一直按住不松手的“低电平持续时间”。本节的例程功能如下：（1）K1 每“短按”一次（25ms），LED 要么从“灭”变成“亮”，要么从“亮”变成“灭”，在两种状态之间切换。（2）K1 每“长按”一次（500ms），蜂鸣器发出“嘀”的一声。代码如下：

```
#include "REG52.H"

#define KEY_VOICE_TIME    50      //按键“长按”触发后发出的声音长度 50ms

#define KEY_SHORT_TIME    25      //按键的“短按”兼“滤波”的“稳定时间” 25ms
#define KEY_LONG_TIME    500     //按键的“长按”兼“滤波”的“稳定时间” 500ms

void T0_time();
void SystemInitial(void) ;
void Delay(unsigned long u32DelayTime) ;
void PeripheralInitial(void) ;

void BeepOpen(void);
void BeepClose(void);
void LedOpen(void);
void LedClose(void);

void VoiceScan(void);
void KeyScan(void);    //按键识别的驱动函数，放在定时中断里
void SingleKeyTask(void);    //单击按键任务函数，放在主函数内

sbit P3_4=P3^4;        //蜂鸣器
sbit P1_4=P1^4;        //LED

sbit KEY_INPUT1=P2^2;  //K1 按键识别的输入口。

volatile unsigned char vGu8BeepTimerFlag=0;
volatile unsigned int vGu16BeepTimerCnt=0;

unsigned char Gu8LedStatus=0; //记录 LED 灯的状态，0 代表灭，1 代表亮

volatile unsigned char vGu8SingleKeySec=0; //单击按键的触发序号

void main()
{
    SystemInitial();
    Delay(10000);
    PeripheralInitial();
```



```

while(1)
{
    SingleKeyTask();    //单击按键任务函数
}

void TO_time() interrupt 1
{
    VoiceScan();
    KeyScan();    //按键识别的驱动函数

    TH0=0xfc;
    TL0=0x66;
}

void SystemInitial(void)
{
    TMOD=0x01;
    TH0=0xfc;
    TL0=0x66;
    EA=1;
    ET0=1;
    TR0=1;
}

void Delay(unsigned long u32DelayTime)
{
    for(;u32DelayTime>0;u32DelayTime--);
}

void PeripheralInitial(void)
{
    if(0==Gu8LedStatus)
    {
        LedClose();
    }
    else
    {
        LedOpen();
    }
}

void BeepOpen(void)

```

```

{
    P3_4=0;
}

void BeepClose(void)
{
    P3_4=1;
}

void LedOpen(void)
{
    P1_4=0;
}

void LedClose(void)
{
    P1_4=1;
}

void VoiceScan(void)
{
    static unsigned char Su8Lock=0;

    if(1==vGu8BeepTimerFlag&&vGu16BeepTimerCnt>0)
    {
        if(0==Su8Lock)
        {
            Su8Lock=1;
            BeepOpen();
        }
        else
        {
            vGu16BeepTimerCnt--;

            if(0==vGu16BeepTimerCnt)
            {
                Su8Lock=0;
                BeepClose();
            }
        }
    }
}

```

```

/* 注释一：
* “长按”与“短按”的识别过程：
* 第一步：平时只要 K1 没有被按下，按键的自锁标志 Su8KeyLock1 和去抖动延时计数器 Su16KeyCnt1
* 一直被清零。此时属于按键“松手时间”，因此同时检测“短按”标志 Su8KeyShortFlag
* 是否有效，如果有效就触发一次“短按”。
* 第二步：一旦 K1 按键被按下，去抖动延时计数器 Su16KeyCnt1 开始在定时中断函数里累加，在还没
* 累加到阈值 KEY_SHORT_TIME 和 KEY_LONG_TIME 时，如果在这期间由于受外界干扰或者
* 按键抖动，而使 IO 口突然瞬间触发成高电平，这个时候马上把延时计数器 Su16KeyCnt1 清零，
* 这个过程非常巧妙，非常有效地去除瞬间的杂波干扰。
* 第三步：如果 K1 按键按下的时间超过了“短按”阈值 KEY_SHORT_TIME，马上把“短按”标志
* Su8KeyShortFlag 置 1，如果此时还没有松手，直到发现按下的时间超过“长按”阈值
* KEY_LONG_TIME 时，先把“短按”标志 ucShortTouchFlag1 清零，然后触发“长按”，同时，为
* 了防止按住按键不松手后一直触发，要及时把 Su8KeyLock1 置 1 “自锁”。
* 第四步：等 K1 按键松手后，自锁标志 Su8KeyLock1 及时清零，为下一次自锁做准备，同时，也检测
* “短按”标志 Su8KeyShortFlag 是否有效，如果有效就触发一次“短按”。
*/

void KeyScan(void) //此函数放在定时中断里每 1ms 扫描一次
{
    static unsigned char Su8KeyLock1;
    static unsigned int Su16KeyCnt1;
    static unsigned char Su8KeyShortFlag=0; //按键“短按”触发的标志

    if(0!=KEY_INPUT1)//单个 K1 按键没有按下，及时清零一些标志。
    {
        Su8KeyLock1=0; //按键解锁
        Su16KeyCnt1=0; //去抖动延时计数器清零，此行非常巧妙，是全场的亮点。
        if(1==Su8KeyShortFlag) //松手的时候，如果“短按”标志有效就触发一次“短按”
        {
            Su8KeyShortFlag=0; //先清零“短按”标志避免一直触发。
            vGuSingleKeySec=1; //触发 K1 的“短按”
        }
    }
    else if(0==Su8KeyLock1)//单个按键 K1 被按下
    {
        Su16KeyCnt1++; //累加定时中断次数

        if(Su16KeyCnt1>=KEY_SHORT_TIME) // “短按”兼“滤波”的“稳定时间” KEY_SHORT_TIME
        {
            //注意，这里不能“自锁”。后面“长按”触发的时候才“自锁”。
            Su8KeyShortFlag=1; //K1 的“短按”标志有效，待松手时触发。
        }
    }
}

```

```

        if(Su16KeyCnt1>=KEY_LONG_TIME) // “长按”兼“滤波”的“稳定时间” KEY_LONG_TIME
        {
            Su8KeyLock1=1;          //此时“长按”触发才“自锁”
            Su8KeyShortFlag=0;      //既然此时“长按”有效，那么就要废除潜在的“短按”。
            vGu8SingleKeySec=2; //触发 K1 的“长按”
        }
    }
}

void SingleKeyTask(void)    //单击按键任务函数，放在主函数内
{
    if(0==vGu8SingleKeySec)
    {
        return; //按键的触发序号是 0 意味着无按键触发，直接退出当前函数，不执行此函数下面的代码
    }

    switch(vGu8SingleKeySec) //根据不同的按键触发序号执行对应的代码
    {
        case 1:          //K1 “短按”触发的任务
            if(0==Gu8LedStatus)
            {
                Gu8LedStatus=1;
                LedOpen();    //LED 亮
            }
            else
            {
                Gu8LedStatus=0;
                LedClose();    //LED 灭
            }
            vGu8SingleKeySec=0; //响应按键服务处理程序后，按键编号必须清零，避免一直触发
            break;

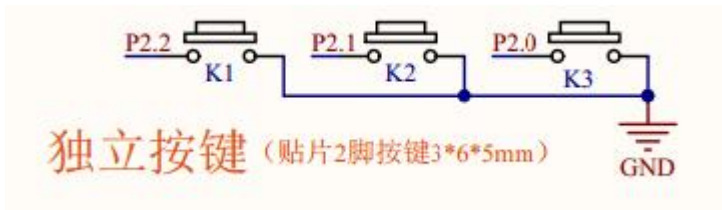
        case 2:          //K1 “长按”触发的任务
            vGu8BeepTimerFlag=0;
            vGu16BeepTimerCnt=KEY_VOICE_TIME; //触发一次“长按”后，发出“嘀”一声
            vGu8BeepTimerFlag=1;
            vGu8SingleKeySec=0; //响应按键服务处理程序后，按键编号必须清零，避免一直触发
            break;

    }
}

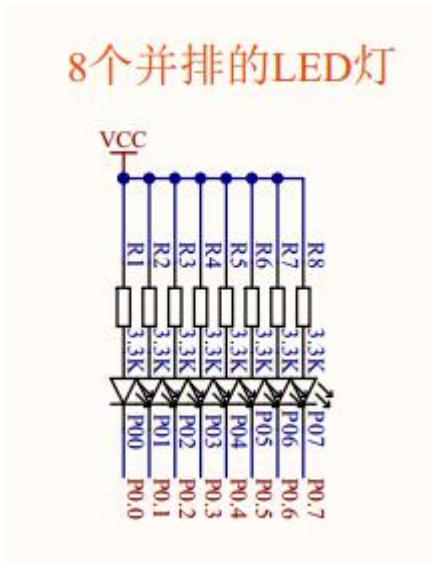
```

第九十七节： 独立按键按住不松手的连续均匀触发。

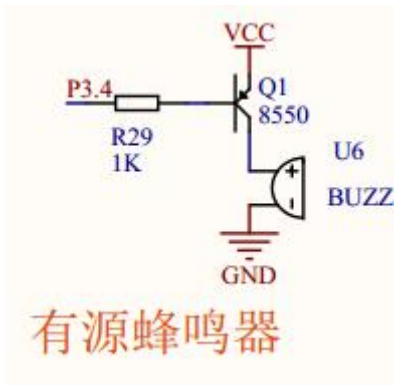
【97.1 按住不松手的连续均匀触发。】



上图 97. 1. 1 独立按键电路



上图 97. 1. 2 灌入式驱动 8 个 LED



上图 97. 1. 3 有源蜂鸣器电路

在电脑上删除某个文件某行文字的时候，单击一次“退格按键[Backspace]”，就删除一个文字，如果按

住“退格按键[Backspace]”不松手，就会“连续均匀”的触发“删除”的功能，自动逐个把整行文字删除清空，这就是“按住不松手的连续均匀触发”应用案例之一。除此之外，在很多需要人机交互的项目中都有这样的功能，为了快速加减某个数值，按住某个按键不松手，某个数值有节奏地快速往上加或者快速往下减。这种“按住不松手连续均匀触发”的按键识别，在程序上有“3个时间”需要留意，第1个是按键单击的“滤波”时间，第2个是按键“从单击进入连击”的间隔时间（此时间是“单击”与“连击”的分界线），第3个是按键“连击”的间隔时间，

本节例程实现的功能如下：（1）8个受按键控制的跑马灯在某一时刻只有1个LED亮，每触发一次K1按键，“亮的LED”就“往左边跑一步”；相反，每触发一次K2按键，“亮的LED”就“往右边跑一步”。如果按住K1或者K2不松手就连续触发，“亮的LED”就“连续跑”，一直跑到左边或者右边的尽头。（2）按键每“单击”一次蜂鸣器就鸣叫一次，但是，当按键“从单击进入连击”后，蜂鸣器就不鸣叫。

```
#include "REG52.H"

#define KEY_VOICE_TIME    50

#define KEY_SHORT_TIME    25        //按键单击的“滤波”时间 25ms
#define KEY_ENTER_CONTINUITY_TIME    300 //按键“从单击进入连击”的间隔时间 300ms
#define KEY_CONTINUITY_TIME    80    //按键“连击”的间隔时间 80ms

#define BUS_P0    P0        //8个LED灯一一对应单片机的P0口总线

void T0_time();
void SystemInitial(void) ;
void Delay(unsigned long u32DelayTime) ;
void PeripheralInitial(void) ;

void BeepOpen(void);
void BeepClose(void);

void VoiceScan(void);
void KeyScan(void);
void KeyTask(void);
void DisplayTask(void);    //显示的任务函数（LED显示状态）

sbit P3_4=P3^4;           //蜂鸣器

sbit KEY_INPUT1=P2^2;    //K1 按键识别的输入口。
sbit KEY_INPUT2=P2^1;    //K2 按键识别的输入口。

volatile unsigned char vGu8BeepTimerFlag=0;
volatile unsigned int vGu16BeepTimerCnt=0;

unsigned char Gu8LedStatus=0; //LED灯的状态
unsigned char Gu8DisplayUpdate=1; //显示的刷新标志
```

```

volatile unsigned char vGu8KeySec=0; //按键的触发序号
volatile unsigned char vGu8ShieldVoiceFlag=0; //屏蔽声音的标志

void main()
{
    SystemInitial();
    Delay(10000);
    PeripheralInitial();
    while(1)
    {
        KeyTask(); //按键的任务函数
        DisplayTask(); //显示的任务函数（LED 显示状态）
    }
}

/* 注释一：
* Gu8DisplayUpdate 这类“显示刷新变量”在“显示框架”里是很常见的，而且屡用屡爽。
* 目的是，既能及时刷新显示，又能避免主函数“不断去执行显示代码”而影响程序效率。
*/

void DisplayTask(void) //显示的任务函数（LED 显示状态）
{
    if(1==Gu8DisplayUpdate) //需要刷新一次显示
    {
        Gu8DisplayUpdate=0; //及时清零，避免主函数“不断去执行显示代码”而影响程序效率

        //Gu8LedStatus 是左移的位数，范围（0 至 7），决定了跑马灯的显示状态。
        BUS_P0=~(1<<Gu8LedStatus); //“左移<<”之后的“取反~”，因为 LED 电路是灌入式驱动方式。
    }
}

/* 注释二：
* 按键“连续均匀触发”的识别过程：
* 第一步：平时只要 K1 没有被按下，按键的自锁标志 Su8KeyLock1、去抖动延时计数器 Su16KeyCnt1、
* 连击计数器 Su16KeyContinuityCnt1，一直被清零。
* 第二步：一旦 K1 按键被按下，去抖动延时计数器 Su16KeyCnt1 开始在定时中断函数里累加，在还没
* 累加到阈值 KEY_SHORT_TIME 时，如果在这期间由于受外界干扰或者按键抖动，
* 而使 IO 口突然瞬间触发成高电平，这个时候马上把延时计数器 Su16KeyCnt1 清零，
* 这个过程非常巧妙，非常有效地去除瞬间的杂波干扰。
* 第三步：如果 K1 按键按下的时间超过了阈值 KEY_SHORT_TIME，则触发一次“单击”，同时，马上把自锁
* 标志 Su8KeyLock1 置 1 防止按住按键不松手后一直触发，并且把计数器 Su16KeyCnt1 清零为了下
* 一步用来累加“从单击进入连击的间隔时间 1000ms”。如果此时还没有松手，直到发现按下的时

```

```

*      间超过“从单击进入连击的间隔时间”阈值 KEY_ENTER_CONTINUITY_TIME 时，从此进入“连击”
*      的模式，连击计数器 Su16KeyContinuityCnt1 开始累加，每到达一次阈值
*      KEY_CONTINUITY_TIME 就触发 1 次按键，为了屏蔽按键声音及时把 vGu8ShieldVoiceFlag 也置 1，
*      同时，Su16KeyContinuityCnt1 马上清零为继续连击作准备。
* 第四步：等 K1 按键松手后，自锁标志 Su8KeyLock1、去抖动延时计数器 Su16KeyCnt1、
*      连击计数器 Su16KeyContinuityCnt1，及时清零，为下一次按键触发做准备。
*/

void KeyScan(void)  //此函数放在定时中断里每 1ms 扫描一次
{
    static unsigned char Su8KeyLock1;
    static unsigned int  Su16KeyCnt1;
    static unsigned int  Su16KeyContinuityCnt1;  //连击计数器

    static unsigned char Su8KeyLock2;
    static unsigned int  Su16KeyCnt2;
    static unsigned int  Su16KeyContinuityCnt2;  //连击计数器

    //K1 按键
    if(0!=KEY_INPUT1)//单个 K1 按键没有按下，及时清零一些标志。
    {
        Su8KeyLock1=0; //按键解锁
        Su16KeyCnt1=0;  //去抖动延时计数器清零，此行非常巧妙，是全场的亮点。
        Su16KeyContinuityCnt1=0;  //连击计数器
    }
    else if(0==Su8KeyLock1)//单个按键 K1 被按下
    {
        Su16KeyCnt1++; //累加定时中断次数，每一次累加额度是 1ms
        if(Su16KeyCnt1>=KEY_SHORT_TIME) //按键的“滤波”时间 25ms
        {
            Su8KeyLock1=1;      //“自锁”
            vGu8KeySec=1;      //触发一次 K1 按键
            Su16KeyCnt1=0;      //清零，为了下一步用来累加“从单击进入连击的间隔时间 300ms”
        }
    }
    else if(Su16KeyCnt1<=KEY_ENTER_CONTINUITY_TIME)//按住不松手累加到 300ms
    {
        Su16KeyCnt1++; //累加定时中断次数，每一次累加额度是 1ms
    }
    else //按住累加到 300ms 后仍然不放手，这个时候进入有节奏的连续触发
    {
        Su16KeyContinuityCnt1++; //连击计数器开始累加，每一次累加额度是 1ms
        if(Su16KeyContinuityCnt1>=KEY_CONTINUITY_TIME) //按住没松手，每 0.08 秒就触发一次
        {

```



```

        Su16KeyContinuityCnt1=0; //清零，为了继续连击。
        vGu8KeySec=1;           //触发一次 K1 按键
        vGu8ShieldVoiceFlag=1;  //把当前按键触发的声音屏蔽掉
    }

}

//K2 按键
if(0!=KEY_INPUT2)
{
    Su8KeyLock2=0;
    Su16KeyCnt2=0;
    Su16KeyContinuityCnt2=0;
}
else if(0==Su8KeyLock2)
{
    Su16KeyCnt2++;
    if(Su16KeyCnt2>=KEY_SHORT_TIME)
    {
        Su8KeyLock2=1;
        vGu8KeySec=2;           //触发一次 K2 按键
        Su16KeyCnt2=0;
    }
}
else if(Su16KeyCnt2<=KEY_ENTER_CONTINUITY_TIME)
{
    Su16KeyCnt2++;
}
else
{
    Su16KeyContinuityCnt2++;
    if(Su16KeyContinuityCnt2>=KEY_CONTINUITY_TIME)
    {
        Su16KeyContinuityCnt2=0;
        vGu8KeySec=2;           //触发一次 K2 按键
        vGu8ShieldVoiceFlag=1;  //把当前按键触发的声音屏蔽掉
    }
}
}

void KeyTask(void)    //按键任务函数，放在主函数内
{
    if(0==vGu8KeySec)

```

```

{
    return; //按键的触发序号是 0 意味着无按键触发，直接退出当前函数，不执行此函数下面的代码
}

switch(vGu8KeySec) //根据不同的按键触发序号执行对应的代码
{
    case 1: //K1 触发的任务
        if(Gu8LedStatus>0)
        {
            Gu8LedStatus--; //控制 LED “往左边跑”
            Gu8DisplayUpdate=1; //刷新显示
        }

        if(0==vGu8ShieldVoiceFlag) //声音没有被屏蔽
        {
            vGu8BeepTimerFlag=0;
            vGu16BeepTimerCnt=KEY_VOICE_TIME; //发出“嘀”一声
            vGu8BeepTimerFlag=1;
        }

        vGu8ShieldVoiceFlag=0; //及时把屏蔽标志清零，避免平时正常的单击声音也被淹没。
        vGu8KeySec=0; //响应按键服务处理程序后，按键编号必须清零，避免一直触发
        break;

    case 2: //K2 触发的任务
        if(Gu8LedStatus<7)
        {
            Gu8LedStatus++; //控制 LED “往右边跑”
            Gu8DisplayUpdate=1; //刷新显示
        }

        if(0==vGu8ShieldVoiceFlag) //声音没有被屏蔽
        {
            vGu8BeepTimerFlag=0;
            vGu16BeepTimerCnt=KEY_VOICE_TIME; //发出“嘀”一声
            vGu8BeepTimerFlag=1;
        }

        vGu8ShieldVoiceFlag=0; //及时把屏蔽标志清零，避免平时正常的单击声音也被淹没。
        vGu8KeySec=0; //响应按键服务处理程序后，按键编号必须清零，避免一直触发
        break;

}
}

```

```
void T0_time() interrupt 1
{
    VoiceScan();
    KeyScan();    //按键识别的驱动函数

    TH0=0xfc;
    TL0=0x66;
}

void SystemInitial(void)
{
    TMOD=0x01;
    TH0=0xfc;
    TL0=0x66;
    EA=1;
    ET0=1;
    TR0=1;
}

void Delay(unsigned long u32DelayTime)
{
    for(;u32DelayTime>0;u32DelayTime--);
}

void PeripheralInitial(void)
{
}

void BeepOpen(void)
{
    P3_4=0;
}

void BeepClose(void)
{
    P3_4=1;
}

void VoiceScan(void)
```

```
{

    static unsigned char Su8Lock=0;

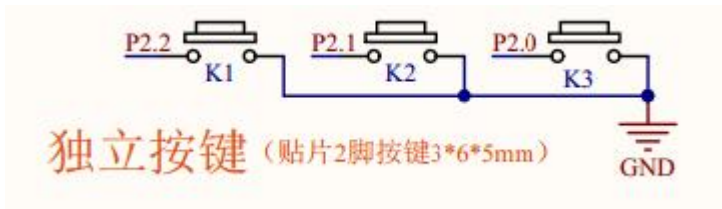
    if(1==vGu8BeepTimerFlag&&vGu16BeepTimerCnt>0)
    {
        if(0==Su8Lock)
        {
            Su8Lock=1;
            BeepOpen();
        }
        else
        {

            vGu16BeepTimerCnt--;

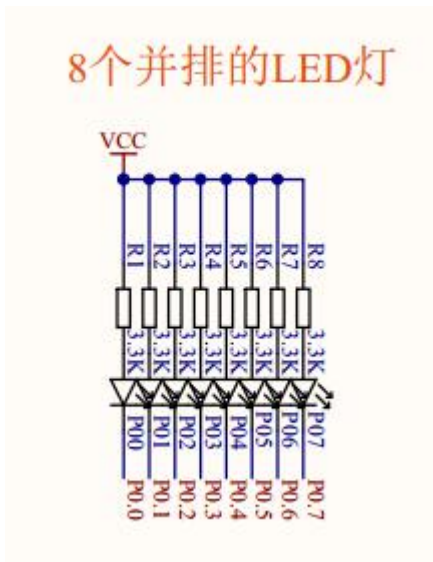
            if(0==vGu16BeepTimerCnt)
            {
                Su8Lock=0;
                BeepClose();
            }
        }
    }
}
```

第九十八节： 独立按键按住不松手的“先加速后匀速”的触发。

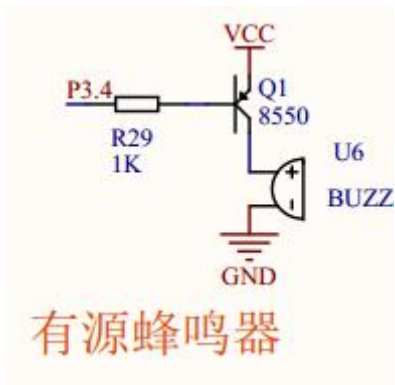
【98.1 “先加速后匀速”的触发。】



上图 98. 1. 1 独立按键电路



上图 98. 1. 2 灌入式驱动 8 个 LED



上图 98. 1. 3 有源蜂鸣器电路

当“连续加”或者“连续减”的数据范围很大的时候，就需要按键的加速与匀速相结合的触发方式。“加速”是指按住按键不松手，按键刚开始触发是从慢到快的渐进过程，当“加速”到某个特别快的速度的时候，

就“不再加速”，而是以该“恒定高速”进行“连续匀速”触发。这种触发方式，“加速”和“匀速”是相辅相成缺一不可的，为什么？假如没有“加速”只有“匀速”，那么刚按下按键就直接以最高速的“匀速”进行，就会跑过头，缺乏微调功能；而假如没有“匀速”只有“加速”，那么按下按键不松手后，速度就会一直不断飙升，最后失控过冲。

本节例程实现的功能如下：

(1) 要更改一个“设置参数”（一个全局变量），参数的范围是 0 到 800。

(2) 8 个受“设置参数”控制的跑马灯在某一时刻只有 1 个 LED 亮，每触发一次 K1 按键，该“设置参数”就自减 1，最小值为 0；相反，每触发一次 K2 按键，该“设置参数”就自加 1，最大值为 800。

(3) LED 灯实时显示“设置参数”的范围状态：

只有第 0 个 LED 灯亮：0<= “设置参数” <100。

只有第 1 个 LED 灯亮：100<= “设置参数” <200。

只有第 2 个 LED 灯亮：200<= “设置参数” <300。

只有第 3 个 LED 灯亮：300<= “设置参数” <400。

只有第 4 个 LED 灯亮：400<= “设置参数” <500。

只有第 5 个 LED 灯亮：500<= “设置参数” <600。

只有第 6 个 LED 灯亮：600<= “设置参数” <700。

只有第 7 个 LED 灯亮：700<= “设置参数” <=800。

(4) 按键每“单击”一次蜂鸣器就鸣叫一次，但是，当按键“从单击进入连击”后，蜂鸣器就不鸣叫。

```
#include "REG52.H"

#define KEY_VOICE_TIME    50

#define KEY_SHORT_TIME    25        //按键单击的“滤波”时间
#define KEY_ENTER_CONTINUITY_TIME    300 //按键“从单击进入连击”的间隔时间
#define KEY_CONTINUITY_INITIAL_TIME    80 //按键“连击”起始的预设间隔时间
#define KEY_SUB_DT_TIME    8        //按键在“加速”时每次减小的时间。
#define KEY_CONTINUITY_MIN_TIME    10 //按键时间减小到最后的“匀速”间隔时间。

#define BUS_P0    P0        //8 个 LED 灯一一对应单片机的 P0 口总线

void T0_time();
void SystemInitial(void) ;
void Delay(unsigned long u32DelayTime) ;
void PeripheralInitial(void) ;

void BeepOpen(void);
void BeepClose(void);

void VoiceScan(void);
void KeyScan(void);
void KeyTask(void);
void DisplayTask(void);    //显示的任务函数（LED 显示状态）
```

```

sbit P3_4=P3^4;          //蜂鸣器

sbit KEY_INPUT1=P2^2;    //K1 按键识别的输入口。
sbit KEY_INPUT2=P2^1;    //K2 按键识别的输入口。

volatile unsigned char vGu8BeepTimerFlag=0;
volatile unsigned int vGu16BeepTimerCnt=0;

unsigned int Gu16SetData=0; //“设置参数”。范围从 0 到 800。LED 灯反映该当前值的范围状态
unsigned char Gu8DisplayUpdate=1; //显示的刷新标志

volatile unsigned char vGu8KeySec=0; //按键的触发序号
volatile unsigned char vGu8ShieldVoiceFlag=0; //屏蔽声音的标志

void main()
{
    SystemInitial();
    Delay(10000);
    PeripheralInitial();
    while(1)
    {
        KeyTask();          //按键的任务函数
        DisplayTask();       //显示的任务函数（LED 显示状态）
    }
}

/* 注释一：
* Gu8DisplayUpdate 这类“显示刷新变量”在“显示框架”里是很常见的，而且屡用屡爽。
* 目的是，既能及时刷新显示，又能避免主函数“不断去执行显示代码”而影响程序效率。
*/
void DisplayTask(void) //显示的任务函数（LED 显示状态）
{
    if(1==Gu8DisplayUpdate) //需要刷新一次显示
    {
        Gu8DisplayUpdate=0; //及时清零，避免主函数“不断去执行显示代码”而影响程序效率

        if(Gu16SetData<100)
        {
            BUS_P0=~(1<<0); //第 0 个灯亮
        }
        else if(Gu16SetData<200)
        {

```

```

        BUS_P0=~(1<<1); //第 1 个灯亮
    }
    else if(Gu16SetData<300)
    {
        BUS_P0=~(1<<2); //第 2 个灯亮
    }
    else if(Gu16SetData<400)
    {
        BUS_P0=~(1<<3); //第 3 个灯亮
    }
    else if(Gu16SetData<500)
    {
        BUS_P0=~(1<<4); //第 4 个灯亮
    }
    else if(Gu16SetData<600)
    {
        BUS_P0=~(1<<5); //第 5 个灯亮
    }
    else if(Gu16SetData<700)
    {
        BUS_P0=~(1<<6); //第 6 个灯亮
    }
    else
    {
        BUS_P0=~(1<<7); //第 7 个灯亮
    }
}
}

```

/\* 注释二:

\* 按键“先加速后匀速”的识别过程:

\* 第一步: 每次按一次就触发一次“单击”, 如果按下去到松手的时间不超过 1 秒, 则不会进入  
\* “连击”模式。

\* 第二步: 如果按下去不松手的时间超过 1 秒, 则进入“连击”模式。按键触发的节奏

\* 不断加快, 直至到达某个极限值, 然后以此极限值间隔匀速触发。这就是“先加速后匀速”。

\*/

void KeyScan(void) //此函数放在定时中断里每 1ms 扫描一次

```

{
    static unsigned char Su8KeyLock1;
    static unsigned int Su16KeyCnt1;
    static unsigned int Su16KeyContinuityCnt1; //连击计数器
    static unsigned int Su16KeyContinuityTime1=KEY_CONTINUITY_INITIAL_TIME; //动态时间阈值

```



```

static unsigned char Su8KeyLock2;
static unsigned int Su16KeyCnt2;
static unsigned int Su16KeyContinuityCnt2; //连击计数器
static unsigned int Su16KeyContinuityTime2=KEY_CONTINUITY_INITIAL_TIME; //动态时间阈值

//K1 按键
if(0!=KEY_INPUT1)//单个 K1 按键没有按下，及时清零一些标志。
{
    Su8KeyLock1=0; //按键解锁
    Su16KeyCnt1=0; //去抖动延时计数器清零，此行非常巧妙，是全场的亮点。
    Su16KeyContinuityCnt1=0; //连击计数器
    Su16KeyContinuityTime1=KEY_CONTINUITY_INITIAL_TIME; //动态时间阈值。重装初始值。
}
else if(0==Su8KeyLock1)//单个按键 K1 被按下
{
    Su16KeyCnt1++; //累加定时中断次数，每一次累加额度是 1ms
    if(Su16KeyCnt1>=KEY_SHORT_TIME) //按键的“滤波”时间 25ms
    {
        Su8KeyLock1=1; // “自锁”
        vGu8KeySec=1; //触发一次 K1 按键
        Su16KeyCnt1=0; //清零，为了下一步用来累加 “从单击进入连击的间隔时间 300ms”
    }
}
else if(Su16KeyCnt1<=KEY_ENTER_CONTINUITY_TIME)//按住不松手累加到 300ms
{
    Su16KeyCnt1++; //累加定时中断次数，每一次累加额度是 1ms
}
else //按住累加到 300ms 后仍然不放手，这个时候进入有节奏的连续触发
{
    Su16KeyContinuityCnt1++; //连击计数器开始累加，每一次累加额度是 1ms
    if(Su16KeyContinuityCnt1>=Su16KeyContinuityTime1) //按住没松手，每隔一会就触发一次
    {
        Su16KeyContinuityCnt1=0; //清零，为了继续连击。
        vGu8KeySec=1; //触发一次 K1 按键
        vGu8ShieldVoiceFlag=1; //把当前按键触发的声音屏蔽掉
        if(Su16KeyContinuityTime1>=KEY_SUB_DT_TIME)
        {
            //此数值不断被减小，按键的触发速度就不断变快
            Su16KeyContinuityTime1=Su16KeyContinuityTime1-KEY_SUB_DT_TIME;//变快节奏
        }

        if(Su16KeyContinuityTime1<KEY_CONTINUITY_MIN_TIME) //最小间隔时间就是“高速匀速”
        {
            Su16KeyContinuityTime1=KEY_CONTINUITY_MIN_TIME; //最后以此最高速进行“匀速”
        }
    }
}

```

```

    }

}

}

//K2 按键
if(0!=KEY_INPUT2)
{
    Su8KeyLock2=0;
    Su16KeyCnt2=0;
    Su16KeyContinuityCnt2=0;
    Su16KeyContinuityTime2=KEY_CONTINUITY_INITIAL_TIME;
}
else if(0==Su8KeyLock2)
{
    Su16KeyCnt2++;
    if(Su16KeyCnt2>=KEY_SHORT_TIME)
    {
        Su8KeyLock2=1;
        vGu8KeySec=2;          //触发一次 K2 按键
        Su16KeyCnt2=0;
    }
}
else if(Su16KeyCnt2<=KEY_ENTER_CONTINUITY_TIME)
{
    Su16KeyCnt2++;
}
else
{
    Su16KeyContinuityCnt2++;
    if(Su16KeyContinuityCnt2>=Su16KeyContinuityTime2)
    {
        Su16KeyContinuityCnt2=0;
        vGu8KeySec=2;          //触发一次 K2 按键
        vGu8ShieldVoiceFlag=1;
        if(Su16KeyContinuityTime2>=KEY_SUB_DT_TIME)
        {
            Su16KeyContinuityTime2=Su16KeyContinuityTime2-KEY_SUB_DT_TIME;
        }

        if(Su16KeyContinuityTime2<KEY_CONTINUITY_MIN_TIME)
        {
            Su16KeyContinuityTime2=KEY_CONTINUITY_MIN_TIME;
        }
    }
}

```

```

    }
}

void KeyTask(void)    //按键任务函数，放在主函数内
{
    if(0==vGu8KeySec)
    {
        return; //按键的触发序号是 0 意味着无按键触发，直接退出当前函数，不执行此函数下面的代码
    }

    switch(vGu8KeySec) //根据不同的按键触发序号执行对应的代码
    {
        case 1:        //K1 触发的任务
            if(Gu16SetData>0)
            {
                Gu16SetData--;    // “设置参数”
                Gu8DisplayUpdate=1; //刷新显示
            }

            if(0==vGu8ShieldVoiceFlag) //声音没有被屏蔽
            {
                vGu8BeepTimerFlag=0;
                vGu16BeepTimerCnt=KEY_VOICE_TIME; //发出“滴”一声
                vGu8BeepTimerFlag=1;
            }

            vGu8ShieldVoiceFlag=0; //及时把屏蔽标志清零，避免平时正常的单击声音也被淹没。
            vGu8KeySec=0; //响应按键服务处理程序后，按键编号必须清零，避免一直触发
            break;

        case 2:        //K2 触发的任务
            if(Gu16SetData<800)
            {
                Gu16SetData++;    // “设置参数”
                Gu8DisplayUpdate=1; //刷新显示
            }

            if(0==vGu8ShieldVoiceFlag) //声音没有被屏蔽
            {
                vGu8BeepTimerFlag=0;
                vGu16BeepTimerCnt=KEY_VOICE_TIME; //发出“滴”一声
                vGu8BeepTimerFlag=1;
            }
    }
}

```

```

    }

    vGu8ShieldVoiceFlag=0; //及时把屏蔽标志清零，避免平时正常的单击声音也被淹没。
    vGu8KeySec=0; //响应按键服务处理程序后，按键编号必须清零，避免一直触发
    break;

}
}

void T0_time() interrupt 1
{
    VoiceScan();
    KeyScan(); //按键识别的驱动函数

    TH0=0xfc;
    TL0=0x66;
}

void SystemInitial(void)
{
    TMOD=0x01;
    TH0=0xfc;
    TL0=0x66;
    EA=1;
    ET0=1;
    TR0=1;
}

void Delay(unsigned long u32DelayTime)
{
    for(;u32DelayTime>0;u32DelayTime--);
}

void PeripheralInitial(void)
{
}

void BeepOpen(void)
{
    P3_4=0;

```

```

}

void BeepClose(void)
{
    P3_4=1;
}

void VoiceScan(void)
{

    static unsigned char Su8Lock=0;

    if(1==vGu8BeepTimerFlag&&vGu16BeepTimerCnt>0)
    {
        if(0==Su8Lock)
        {
            Su8Lock=1;
            BeepOpen();
        }
        else
        {

            vGu16BeepTimerCnt--;

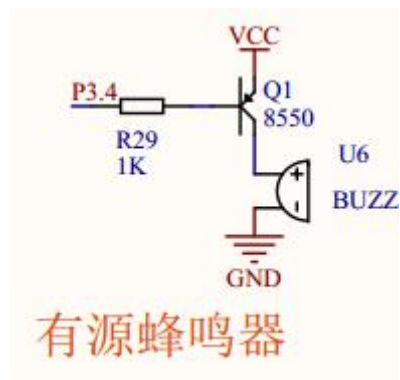
            if(0==vGu16BeepTimerCnt)
            {
                Su8Lock=0;
                BeepClose();
            }

        }
    }
}

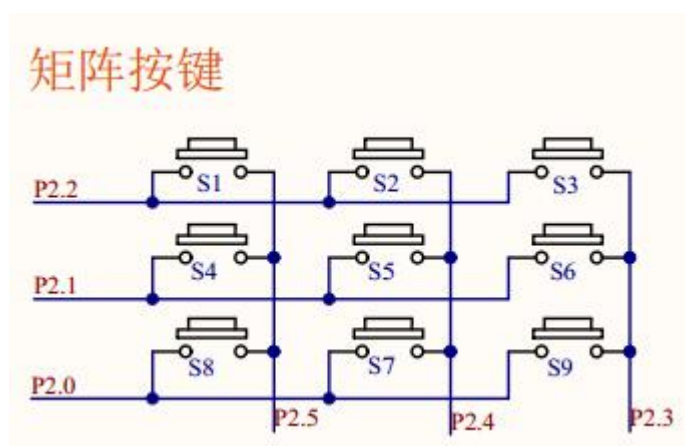
```

## 第九十九节：“行列扫描式”矩阵按键的单个触发（原始版）。

### 【99.1 “行列扫描式”矩阵按键。】



上图 99.1.1 有源蜂鸣器电路



上图 99.1.2 3\*3 矩阵按键的电路

上图是 3\*3 的矩阵按键电路，其它 4\*4 或者 8\*8 的矩阵电路原理是一样的，编程思路也是一样的。相对独立按键，矩阵按键因为采用动态行列扫描的方式，能更加节省 I/O 口，比如 3\*3 的 3 行 3 列，1 行占用 1 根 I/O 口，1 列占用 1 根 I/O 口，因此 3\*3 矩阵按键占用 6 个 I/O 口（3+3=6），但是能识别 9 个按键（3\*3=9）。同理，8\*8 矩阵按键占用 16 个 I/O 口（8+8=16），但是能识别 64 个按键（8\*8=64）。

矩阵按键的编程原理。如上图 3\*3 矩阵按键的电路，行 I/O 口（P2.2, P2.1, P2.0）定为输入，列 I/O 口（P2.5, P2.4, P2.3）定为输出。同一时刻，列输出的 3 个 I/O 口只能有 1 根是输出 L（低电平），其它 2 根必须全是 H（高电平），然后依次轮番切换输出状态，列输出每切换一次，就分别读取一次行输入的 3 个 I/O 口，这样一次就能识别到 3 个按键的状态，如果列连续切换 3 次就可以读取全部 9 个按键的状态。列的 3 种输出状态分别是：（P2.5 为 L，P2.4 为 H，P2.3 为 H），（P2.5 为 H，P2.4 为 L，P2.3 为 H），（P2.5 为 H，P2.4 为 H，P2.3 为 L）。为什么列输出每切换一次就能识别到 3 个按键的状态？因为，首先要明白一个前提，在没有任何按键“被按下”的时候，行输入的 3 个 I/O 口因为内部上拉电阻的作用，默认状态都是 H 电平。并且，H 与 H 相互短接输出为 H，H 与 L 相互短接输出 L，也就是，L（低电平）的优先级最大，任何 H（高电平）碰到 L（低电平）输出的结果都是 L（低电平）。L（低电平）就像数学乘法运算里的数字 0，任何数跟 0 相乘必然等于 0。多说一句，这个“L 最高优先级”法则是有前提的，就是 H（高电平）的产生必须是纯粹靠

上拉电阻拉高的 H（高电平）才行，比如刚好本教程所用的 51 单片机内部 IO 口输出的 H（高电平）是依靠内部的上拉电阻产生，如果是其它“非上拉电阻产生的高电平”与“低电平”短接就有“短路烧坏芯片”的风险，这时就需要额外增加“三极管开漏式输出”电路或者外挂“开漏式输出集成芯片”电路。继续回到正题，为什么列输出每切换一次就能识别到 3 个按键的状态？举个例子，比如当列输出状态处于（P2.5 为 L，P2.4 为 H，P2.3 为 H）下，我们读取行输入的 P2.2 口，行输入的 P2.2 与列输出 P2.5，P2.4，P2.3 的“交叉处”有 3 个按键 S1，S2，S3，此时，如果 P2.2 口是 L（低电平），那么必然是 S1 “被按下”，因为想让 P2.2 口是 L，只有 S1 有这个能力，而如果 S1 没有“被按下”，另外两个 S2，S3 即使“被按下”，P2.2 口也是 H 而绝对不会为 L，因为 S2，S3 的列输出 P2.4 为 H，P2.3 为 H，H 与 H 相互短接输出的结果必然为 H。

本节例程实现的功能：9 个矩阵按键，每按下 1 个按键都触发一次蜂鸣器鸣叫。

```
#include "REG52.H"

#define KEY_VOICE_TIME    50

#define KEY_SHORT_TIME    20    //按键去抖动的“滤波”时间

void T0_time();
void SystemInitial(void) ;
void Delay(unsigned long u32DelayTime) ;
void PeripheralInitial(void) ;

void BeepOpen(void);
void BeepClose(void);

void VoiceScan(void);
void KeyScan(void);
void KeyTask(void);

sbit P3_4=P3^4;    //蜂鸣器

sbit ROW_INPUT1=P2^2; //第 1 行输入口。
sbit ROW_INPUT2=P2^1; //第 2 行输入口。
sbit ROW_INPUT3=P2^0; //第 3 行输入口。

sbit COLUMN_OUTPUT1=P2^5; //第 1 列输出口。
sbit COLUMN_OUTPUT2=P2^4; //第 2 列输出口。
sbit COLUMN_OUTPUT3=P2^3; //第 3 列输出口。

volatile unsigned char vGu8BeepTimerFlag=0;
volatile unsigned int vGu16BeepTimerCnt=0;

volatile unsigned char vGu8KeySec=0; //按键的触发序号
```

```

void main()
{
    SystemInitial();
    Delay(10000);
    PeripheralInitial();
    while(1)
    {
        KeyTask();      //按键的任务函数
    }
}

/* 注释一：
* 矩阵按键扫描的详细过程：
* 先输出某 1 列低电平，其它 2 列输出高电平，延时等待 2ms 后（等此 3 列输出同步稳定），
* 再分别判断 3 行的输入 I/O 口， 如果发现哪一行是低电平，就说明对应的某个按键被触发。
* 依次循环切换输出的 3 种状态，并且分别判断输入的 3 行，就可以检测完 9 个按键。矩阵按键的
* 去抖动处理方法跟我前面讲的独立按键去抖动方法是一样的，不再重复多讲。
*/

void KeyScan(void)  //此函数放在定时中断里每 1ms 扫描一次
{
    static unsigned char Su8KeyLock=0;
    static unsigned int  Su16KeyCnt=0;
    static unsigned char Su8KeyStep=1;

    switch(Su8KeyStep)
    {
        case 1:  //按键扫描输出第一列低电平
            COLUMN_OUTPUT1=0;
            COLUMN_OUTPUT2=1;
            COLUMN_OUTPUT3=1;

            Su16KeyCnt=0;  //延时计数器清零
            Su8KeyStep++;  //切换到下一个运行步骤
            break;

        case 2:  //延时等待 2ms 后（等此 3 列输出同步稳定）。不是按键的去抖动延时。
            Su16KeyCnt++;
            if(Su16KeyCnt>=2)
            {
                Su16KeyCnt=0;
                Su8KeyStep++;  //切换到下一个运行步骤
            }
            break;
    }
}

```



```

case 3:
    if (1==ROW_INPUT1&&1==ROW_INPUT2&&1==ROW_INPUT3)
    {
        Su8KeyStep++; //如果没有按键按下，切换到下一个运行步骤
        Su8KeyLock=0; //按键自锁标志清零
        Su16KeyCnt=0; //按键去抖动延时计数器清零，此行非常巧妙
    }
    else if (0==Su8KeyLock) //有按键按下，且是第一次触发
    {
        if (0==ROW_INPUT1&&1==ROW_INPUT2&&1==ROW_INPUT3)
        {
            Su16KeyCnt++; //去抖动延时计数器
            if (Su16KeyCnt>=KEY_SHORT_TIME)
            {
                Su16KeyCnt=0;
                Su8KeyLock=1; //自锁置 1, 避免一直触发, 只有松开按键, 此标志位才会被清零
                vGu8KeySec=1; //触发 1 号键 对应 S1 键
            }
        }
        else if (1==ROW_INPUT1&&0==ROW_INPUT2&&1==ROW_INPUT3)
        {
            Su16KeyCnt++; //去抖动延时计数器
            if (Su16KeyCnt>=KEY_SHORT_TIME)
            {
                Su16KeyCnt=0;
                Su8KeyLock=1; //自锁置 1, 避免一直触发, 只有松开按键, 此标志位才会被清零
                vGu8KeySec=2; //触发 2 号键 对应 S2 键
            }
        }
        else if (1==ROW_INPUT1&&1==ROW_INPUT2&&0==ROW_INPUT3)
        {
            Su16KeyCnt++; //去抖动延时计数器
            if (Su16KeyCnt>=KEY_SHORT_TIME)
            {
                Su16KeyCnt=0;
                Su8KeyLock=1; //自锁置 1, 避免一直触发, 只有松开按键, 此标志位才会被清零
                vGu8KeySec=3; //触发 3 号键 对应 S3 键
            }
        }
    }
}
break;

```

```

case 4:    //按键扫描输出第二列低电平
    COLUMN_OUTPUT1=1;
    COLUMN_OUTPUT2=0;
    COLUMN_OUTPUT3=1;

    Su16KeyCnt=0; //延时计数器清零
    Su8KeyStep++; //切换到下一个运行步骤
    break;

case 5:    //延时等待 2ms 后（等此 3 列输出同步稳定）。不是按键的去抖动延时。
    Su16KeyCnt++;
    if (Su16KeyCnt>=2)
    {
        Su16KeyCnt=0;
        Su8KeyStep++; //切换到下一个运行步骤
    }
    break;

case 6:
    if (1==ROW_INPUT1&&1==ROW_INPUT2&&1==ROW_INPUT3)
    {
        Su8KeyStep++; //如果没有按键按下，切换到下一个运行步骤
        Su8KeyLock=0; //按键自锁标志清零
        Su16KeyCnt=0; //按键去抖动延时计数器清零，此行非常巧妙
    }
    else if (0==Su8KeyLock) //有按键按下，且是第一次触发
    {
        if (0==ROW_INPUT1&&1==ROW_INPUT2&&1==ROW_INPUT3)
        {
            Su16KeyCnt++; //去抖动延时计数器
            if (Su16KeyCnt>=KEY_SHORT_TIME)
            {
                Su16KeyCnt=0;
                Su8KeyLock=1; //自锁置 1, 避免一直触发, 只有松开按键, 此标志位才会被清零
                vGu8KeySec=4; //触发 4 号键 对应 S4 键
            }
        }
        else if (1==ROW_INPUT1&&0==ROW_INPUT2&&1==ROW_INPUT3)
        {
            Su16KeyCnt++; //去抖动延时计数器
            if (Su16KeyCnt>=KEY_SHORT_TIME)
            {
                Su16KeyCnt=0;

```

```

        Su8KeyLock=1;//自锁置 1, 避免一直触发, 只有松开按键, 此标志位才会被清零
        vGu8KeySec=5;  //触发 5 号键 对应 S5 键
    }
}
else if(1==ROW_INPUT1&&1==ROW_INPUT2&&0==ROW_INPUT3)
{
    Su16KeyCnt++;  //去抖动延时计数器
    if(Su16KeyCnt>=KEY_SHORT_TIME)
    {
        Su16KeyCnt=0;
        Su8KeyLock=1;//自锁置 1, 避免一直触发, 只有松开按键, 此标志位才会被清零
        vGu8KeySec=6;  //触发 6 号键 对应 S6 键
    }
}

}
break;
case 7:  //按键扫描输出第三列低电平
    COLUMN_OUTPUT1=1;
    COLUMN_OUTPUT2=1;
    COLUMN_OUTPUT3=0;

    Su16KeyCnt=0;  //延时计数器清零
    Su8KeyStep++;  //切换到下一个运行步骤
    break;

case 8:  //延时等待 2ms 后（等此 3 列输出同步稳定）。不是按键的去抖动延时。
    Su16KeyCnt++;
    if(Su16KeyCnt>=2)
    {
        Su16KeyCnt=0;
        Su8KeyStep++;  //切换到下一个运行步骤
    }
    break;

case 9:
    if(1==ROW_INPUT1&&1==ROW_INPUT2&&1==ROW_INPUT3)
    {
        Su8KeyStep=1;  //如果没有按键按下, 返回到第一步, 重新开始扫描!!!!!!
        Su8KeyLock=0;  //按键自锁标志清零
        Su16KeyCnt=0;  //按键去抖动延时计数器清零, 此行非常巧妙
    }
    else if(0==Su8KeyLock)  //有按键按下, 且是第一次触发
    {

```

```

        if (0==ROW_INPUT1&&1==ROW_INPUT2&&1==ROW_INPUT3)
        {
            Su16KeyCnt++; //去抖动延时计数器
            if (Su16KeyCnt>=KEY_SHORT_TIME)
            {
                Su16KeyCnt=0;
                Su8KeyLock=1;//自锁置 1, 避免一直触发, 只有松开按键, 此标志位才会被清零
                vGu8KeySec=7; //触发 7 号键 对应 S7 键
            }
        }
        else if (1==ROW_INPUT1&&0==ROW_INPUT2&&1==ROW_INPUT3)
        {
            Su16KeyCnt++; //去抖动延时计数器
            if (Su16KeyCnt>=KEY_SHORT_TIME)
            {
                Su16KeyCnt=0;
                Su8KeyLock=1;//自锁置 1, 避免一直触发, 只有松开按键, 此标志位才会被清零
                vGu8KeySec=8; //触发 8 号键 对应 S8 键
            }
        }
        else if (1==ROW_INPUT1&&1==ROW_INPUT2&&0==ROW_INPUT3)
        {
            Su16KeyCnt++; //去抖动延时计数器
            if (Su16KeyCnt>=KEY_SHORT_TIME)
            {
                Su16KeyCnt=0;
                Su8KeyLock=1;//自锁置 1, 避免一直触发, 只有松开按键, 此标志位才会被清零
                vGu8KeySec=9; //触发 9 号键 对应 S9 键
            }
        }
    }
    break;
}

void KeyTask(void) //按键任务函数, 放在主函数内
{
    if (0==vGu8KeySec)
    {
        return; //按键的触发序号是 0 意味着无按键触发, 直接退出当前函数, 不执行此函数下面的代码
    }
}

```

```
switch(vGu8KeySec) //根据不同的按键触发序号执行对应的代码
{
    case 1:        //S1 触发的任务

        vGu8BeepTimerFlag=0;
        vGu16BeepTimerCnt=KEY_VOICE_TIME; //发出“嘀”一声
        vGu8BeepTimerFlag=1;

        vGu8KeySec=0; //响应按键服务处理程序后，按键编号必须清零，避免一直触发
        break;

    case 2:        //S2 触发的任务

        vGu8BeepTimerFlag=0;
        vGu16BeepTimerCnt=KEY_VOICE_TIME; //发出“嘀”一声
        vGu8BeepTimerFlag=1;

        vGu8KeySec=0; //响应按键服务处理程序后，按键编号必须清零，避免一直触发
        break;

    case 3:        //S3 触发的任务

        vGu8BeepTimerFlag=0;
        vGu16BeepTimerCnt=KEY_VOICE_TIME; //发出“嘀”一声
        vGu8BeepTimerFlag=1;

        vGu8KeySec=0; //响应按键服务处理程序后，按键编号必须清零，避免一直触发
        break;

    case 4:        //S4 触发的任务

        vGu8BeepTimerFlag=0;
        vGu16BeepTimerCnt=KEY_VOICE_TIME; //发出“嘀”一声
        vGu8BeepTimerFlag=1;

        vGu8KeySec=0; //响应按键服务处理程序后，按键编号必须清零，避免一直触发
        break;

    case 5:        //S5 触发的任务

        vGu8BeepTimerFlag=0;
        vGu16BeepTimerCnt=KEY_VOICE_TIME; //发出“嘀”一声
        vGu8BeepTimerFlag=1;
```

```

        vGu8KeySec=0; //响应按键服务处理程序后，按键编号必须清零，避免一直触发
        break;

case 6: //S6 触发的任务

    vGu8BeepTimerFlag=0;
    vGu16BeepTimerCnt=KEY_VOICE_TIME; //发出“嘀”一声
    vGu8BeepTimerFlag=1;

    vGu8KeySec=0; //响应按键服务处理程序后，按键编号必须清零，避免一直触发
    break;

case 7: //S7 触发的任务

    vGu8BeepTimerFlag=0;
    vGu16BeepTimerCnt=KEY_VOICE_TIME; //发出“嘀”一声
    vGu8BeepTimerFlag=1;

    vGu8KeySec=0; //响应按键服务处理程序后，按键编号必须清零，避免一直触发
    break;

case 8: //S8 触发的任务

    vGu8BeepTimerFlag=0;
    vGu16BeepTimerCnt=KEY_VOICE_TIME; //发出“嘀”一声
    vGu8BeepTimerFlag=1;

    vGu8KeySec=0; //响应按键服务处理程序后，按键编号必须清零，避免一直触发
    break;

case 9: //S9 触发的任务

    vGu8BeepTimerFlag=0;
    vGu16BeepTimerCnt=KEY_VOICE_TIME; //发出“嘀”一声
    vGu8BeepTimerFlag=1;

    vGu8KeySec=0; //响应按键服务处理程序后，按键编号必须清零，避免一直触发
    break;

    }
}

```

```

void T0_time() interrupt 1
{
    VoiceScan();
    KeyScan();    //按键识别的驱动函数

    TH0=0xfc;
    TL0=0x66;
}

void SystemInitial(void)
{
    TMOD=0x01;
    TH0=0xfc;
    TL0=0x66;
    EA=1;
    ET0=1;
    TR0=1;
}

void Delay(unsigned long u32DelayTime)
{
    for(;u32DelayTime>0;u32DelayTime--);
}

void PeripheralInitial(void)
{
}

void BeepOpen(void)
{
    P3_4=0;
}

void BeepClose(void)
{
    P3_4=1;
}

void VoiceScan(void)
{

```

```
static unsigned char Su8Lock=0;

if(1==vGu8BeepTimerFlag&&vGu16BeepTimerCnt>0)
{
    if(0==Su8Lock)
    {
        Su8Lock=1;
        BeepOpen();
    }
    else
    {

        vGu16BeepTimerCnt--;

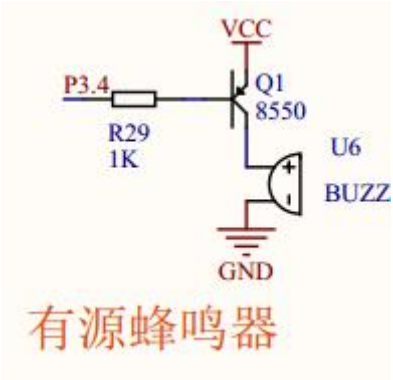
        if(0==vGu16BeepTimerCnt)
        {
            Su8Lock=0;
            BeepClose();
        }

    }
}
}
```

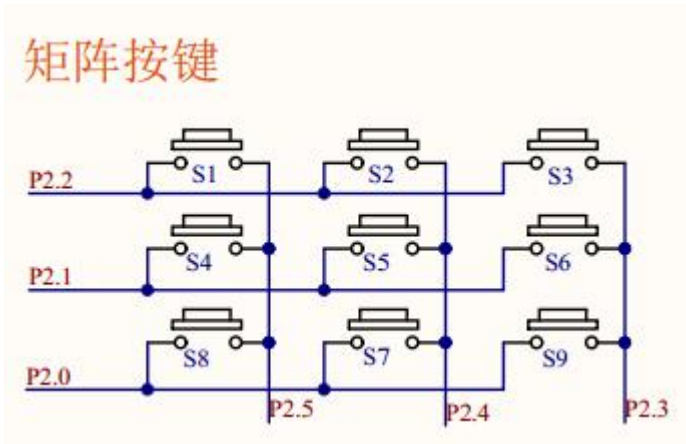


第一百节： “行列扫描式” 矩阵按键的单个触发（优化版）。

【100.1 “行列扫描式” 矩阵按键。】



上图 100.1.1 有源蜂鸣器电路



上图 100.1.2 3\*3 矩阵按键的电路

写程序，凡是出现“重复性、相似性”的代码，都可以加入“循环，判断，数组”这类语句对代码进行压缩优化。上一节讲的矩阵按键，代码是记流水账式的，出现很多“重复性、相似性”的代码，是没有经过优化的“原始版”，本节的目的是对上一节的代码进行优化，让大家从中发现一些技巧。

多说一句，我一直认为，只要单片机容量够，代码多一点少一点并不重要，只要不影响运行效率就行。而且有时候，代码写多一点，可读性非常强，修改起来也非常方便。如果一味的追求压缩代码，就会刻意用很多“循环，判断，数组”等元素，代码虽然紧凑了，但是可分离性，可更改性，可阅读性就没那么强。因此，做项目的时候，某些代码要不要进行压缩，是没有绝对标准的，能因敌而取胜者谓之神。

本节例程实现的功能：9 个矩阵按键，每按下 1 个按键都触发一次蜂鸣器鸣叫。

```
#include "REG52.H"

#define KEY_VOICE_TIME    50

#define KEY_SHORT_TIME    20    //按键去抖动的“滤波”时间
```

```

void TO_time();
void SystemInitial(void) ;
void Delay(unsigned long u32DelayTime) ;
void PeripheralInitial(void) ;

void BeepOpen(void);
void BeepClose(void);

void VoiceScan(void);
void KeyScan(void);
void KeyTask(void);

sbit P3_4=P3^4;          //蜂鸣器

sbit ROW_INPUT1=P2^2;    //第 1 行输入口。
sbit ROW_INPUT2=P2^1;    //第 2 行输入口。
sbit ROW_INPUT3=P2^0;    //第 3 行输入口。

sbit COLUMN_OUTPUT1=P2^5; //第 1 列输出口。
sbit COLUMN_OUTPUT2=P2^4; //第 2 列输出口。
sbit COLUMN_OUTPUT3=P2^3; //第 3 列输出口。

volatile unsigned char vGu8BeepTimerFlag=0;
volatile unsigned int vGu16BeepTimerCnt=0;

volatile unsigned char vGu8KeySec=0; //按键的触发序号

void main()
{
    SystemInitial();
    Delay(10000);
    PeripheralInitial();
    while(1)
    {
        KeyTask();      //按键的任务函数
    }
}

/* 注释一：
* 矩阵按键扫描的详细过程：
* 先输出某 1 列低电平，其它 2 列输出高电平，延时等待 2ms 后（等此 3 列输出同步稳定），
* 再分别判断 3 行的输入 I/O 口， 如果发现哪一行是低电平，就说明对应的某个按键被触发。

```

```
* 依次循环切换输出的 3 种状态，并且分别判断输入的 3 行，就可以检测完 9 个按键。矩阵按键的  
* 去抖动处理方法跟我前面讲的独立按键去抖动方法是一样的，不再重复多讲。  
*/
```

```
void KeyScan(void) //此函数放在定时中断里每 1ms 扫描一次  
{  
    static unsigned char Su8KeyLock=0;  
    static unsigned int  Su16KeyCnt=0;  
    static unsigned char Su8KeyStep=1;  
  
    static unsigned char Su8ColumnRecord=0; //本节多增加此变量用来切换当前列的输出  
  
    switch(Su8KeyStep)  
    {  
        case 1:  
            if(0==Su8ColumnRecord) //按键扫描输出第一列低电平  
            {  
                COLUMN_OUTPUT1=0;  
                COLUMN_OUTPUT2=1;  
                COLUMN_OUTPUT3=1;  
            }  
            else if(1==Su8ColumnRecord) //按键扫描输出第二列低电平  
            {  
                COLUMN_OUTPUT1=1;  
                COLUMN_OUTPUT2=0;  
                COLUMN_OUTPUT3=1;  
            }  
            else //按键扫描输出第三列低电平  
            {  
                COLUMN_OUTPUT1=1;  
                COLUMN_OUTPUT2=1;  
                COLUMN_OUTPUT3=0;  
            }  
            Su16KeyCnt=0; //延时计数器清零  
            Su8KeyStep++; //切换到下一个运行步骤  
            break;  
  
        case 2: //延时等待 2ms 后（等此 3 列输出同步稳定）。不是按键的去抖动延时。  
            Su16KeyCnt++;  
            if(Su16KeyCnt>=2)  
            {  
                Su16KeyCnt=0;  
                Su8KeyStep++; //切换到下一个运行步骤  
            }  
    }  
}
```

```

break;

case 3:
    if (1==ROW_INPUT1&&1==ROW_INPUT2&&1==ROW_INPUT3)
    {
        Su8KeyStep=1; //如果没有按键按下，返回到第一个运行步骤重新开始扫描!!!!!!
        Su8KeyLock=0; //按键自锁标志清零
        Su16KeyCnt=0; //按键去抖动延时计数器清零，此行非常巧妙
        Su8ColumnRecord++; //输出下一列
        if (Su8ColumnRecord>=3)
        {
            Su8ColumnRecord=0; //依次输出完第 3 列之后，继续从第 1 列开始输出低电平
        }
    }
    else if (0==Su8KeyLock) //有按键按下，且是第一次触发
    {
        if (0==ROW_INPUT1&&1==ROW_INPUT2&&1==ROW_INPUT3)
        {
            Su16KeyCnt++; //去抖动延时计数器
            if (Su16KeyCnt>=KEY_SHORT_TIME)
            {
                Su16KeyCnt=0;
                Su8KeyLock=1; //自锁置 1，避免一直触发，只有松开按键，此标志位才会被清零

                if (0==Su8ColumnRecord) //第 1 列输出低电平
                {
                    vGu8KeySec=1; //触发 1 号键 对应 S1 键
                }
                else if (1==Su8ColumnRecord) //第 2 列输出低电平
                {
                    vGu8KeySec=2; //触发 2 号键 对应 S2 键
                }
                else if (2==Su8ColumnRecord) //第 3 列输出低电平
                {
                    vGu8KeySec=3; //触发 3 号键 对应 S3 键
                }
            }
        }
        else if (1==ROW_INPUT1&&0==ROW_INPUT2&&1==ROW_INPUT3)
        {
            Su16KeyCnt++; //去抖动延时计数器
            if (Su16KeyCnt>=KEY_SHORT_TIME)
            {

```

```

        Su16KeyCnt=0;
        Su8KeyLock=1;//自锁置 1, 避免一直触发, 只有松开按键, 此标志位才会被清零

        if(0==Su8ColumnRecord)  //第 1 列输出低电平
        {
            vGu8KeySec=4;  //触发 4 号键 对应 S4 键
        }
        else if(1==Su8ColumnRecord)  //第 2 列输出低电平
        {
            vGu8KeySec=5;  //触发 5 号键 对应 S5 键
        }
        else if(2==Su8ColumnRecord)  //第 3 列输出低电平
        {
            vGu8KeySec=6;  //触发 6 号键 对应 S6 键
        }
    }
}
else if(1==ROW_INPUT1&&1==ROW_INPUT2&&0==ROW_INPUT3)
{
    Su16KeyCnt++;  //去抖动延时计数器
    if(Su16KeyCnt>=KEY_SHORT_TIME)
    {
        Su16KeyCnt=0;
        Su8KeyLock=1;//自锁置 1, 避免一直触发, 只有松开按键, 此标志位才会被清零
        if(0==Su8ColumnRecord)  //第 1 列输出低电平
        {
            vGu8KeySec=7;  //触发 7 号键 对应 S7 键
        }
        else if(1==Su8ColumnRecord)  //第 2 列输出低电平
        {
            vGu8KeySec=8;  //触发 8 号键 对应 S8 键
        }
        else if(2==Su8ColumnRecord)  //第 3 列输出低电平
        {
            vGu8KeySec=9;  //触发 9 号键 对应 S9 键
        }
    }
}

}

break;
}
}

```

```

void KeyTask(void)    //按键任务函数，放在主函数内
{
    if(0==vGu8KeySec)
    {
        return; //按键的触发序号是 0 意味着无按键触发，直接退出当前函数，不执行此函数下面的代码
    }

    switch(vGu8KeySec) //根据不同的按键触发序号执行对应的代码
    {
        case 1:        //S1 触发的任务

            vGu8BeepTimerFlag=0;
            vGu16BeepTimerCnt=KEY_VOICE_TIME; //发出“嘀”一声
            vGu8BeepTimerFlag=1;

            vGu8KeySec=0; //响应按键服务处理程序后，按键编号必须清零，避免一直触发
            break;

        case 2:        //S2 触发的任务

            vGu8BeepTimerFlag=0;
            vGu16BeepTimerCnt=KEY_VOICE_TIME; //发出“嘀”一声
            vGu8BeepTimerFlag=1;

            vGu8KeySec=0; //响应按键服务处理程序后，按键编号必须清零，避免一直触发
            break;

        case 3:        //S3 触发的任务

            vGu8BeepTimerFlag=0;
            vGu16BeepTimerCnt=KEY_VOICE_TIME; //发出“嘀”一声
            vGu8BeepTimerFlag=1;

            vGu8KeySec=0; //响应按键服务处理程序后，按键编号必须清零，避免一直触发
            break;

        case 4:        //S4 触发的任务

            vGu8BeepTimerFlag=0;
            vGu16BeepTimerCnt=KEY_VOICE_TIME; //发出“嘀”一声
            vGu8BeepTimerFlag=1;

            vGu8KeySec=0; //响应按键服务处理程序后，按键编号必须清零，避免一直触发

```

```
        break;

case 5:    //S5 触发的任务

    vGu8BeepTimerFlag=0;
    vGu16BeepTimerCnt=KEY_VOICE_TIME; //发出“嘀”一声
    vGu8BeepTimerFlag=1;

    vGu8KeySec=0; //响应按键服务处理程序后，按键编号必须清零，避免一直触发
    break;

case 6:    //S6 触发的任务

    vGu8BeepTimerFlag=0;
    vGu16BeepTimerCnt=KEY_VOICE_TIME; //发出“嘀”一声
    vGu8BeepTimerFlag=1;

    vGu8KeySec=0; //响应按键服务处理程序后，按键编号必须清零，避免一直触发
    break;

case 7:    //S7 触发的任务

    vGu8BeepTimerFlag=0;
    vGu16BeepTimerCnt=KEY_VOICE_TIME; //发出“嘀”一声
    vGu8BeepTimerFlag=1;

    vGu8KeySec=0; //响应按键服务处理程序后，按键编号必须清零，避免一直触发
    break;

case 8:    //S8 触发的任务

    vGu8BeepTimerFlag=0;
    vGu16BeepTimerCnt=KEY_VOICE_TIME; //发出“嘀”一声
    vGu8BeepTimerFlag=1;

    vGu8KeySec=0; //响应按键服务处理程序后，按键编号必须清零，避免一直触发
    break;

case 9:    //S9 触发的任务

    vGu8BeepTimerFlag=0;
    vGu16BeepTimerCnt=KEY_VOICE_TIME; //发出“嘀”一声
    vGu8BeepTimerFlag=1;
```

```

        vGu8KeySec=0; //响应按键服务处理程序后，按键编号必须清零，避免一直触发
        break;

    }
}

void T0_time() interrupt 1
{
    VoiceScan();
    KeyScan(); //按键识别的驱动函数

    TH0=0xfc;
    TL0=0x66;
}

void SystemInitial(void)
{
    TMOD=0x01;
    TH0=0xfc;
    TL0=0x66;
    EA=1;
    ET0=1;
    TR0=1;
}

void Delay(unsigned long u32DelayTime)
{
    for(;u32DelayTime>0;u32DelayTime--);
}

void PeripheralInitial(void)
{
}

void BeepOpen(void)
{
    P3_4=0;
}

void BeepClose(void)

```



```
{
    P3_4=1;
}

void VoiceScan(void)
{

    static unsigned char Su8Lock=0;

    if(1==vGu8BeepTimerFlag&&vGu16BeepTimerCnt>0)
    {
        if(0==Su8Lock)
        {
            Su8Lock=1;
            BeepOpen();
        }
        else
        {

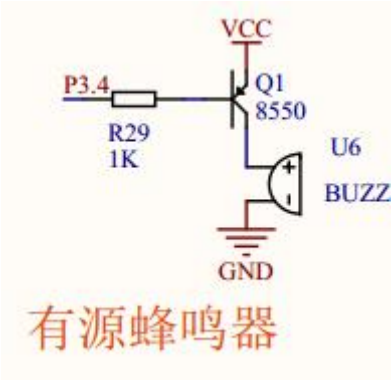
            vGu16BeepTimerCnt--;

            if(0==vGu16BeepTimerCnt)
            {
                Su8Lock=0;
                BeepClose();
            }

        }
    }
}
```

第一百零一节： 矩阵按键鼠标式的单击与双击。

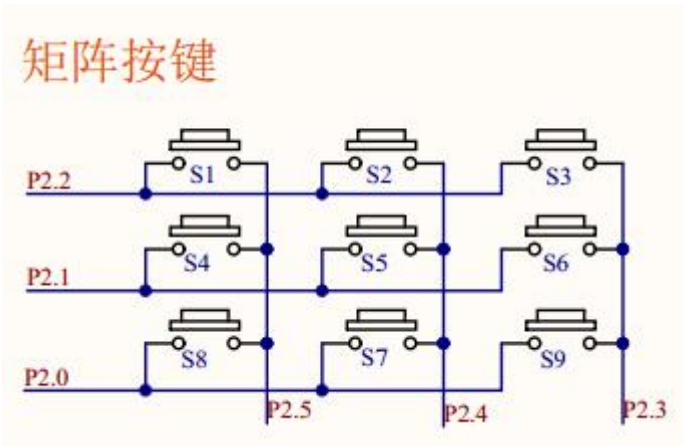
【101.1 矩阵按键鼠标式的单击与双击。】



上图 101.1.1 有源蜂鸣器电路



上图 101.1.2 LED 电路



上图 101.1.3 3\*3 矩阵按键的电路

矩阵按键与前面章节独立按键的单击与双击的处理思路是一样的，本节讲矩阵按键的单击与双击，也算是重温之前章节讲的内容。

鼠标的左键，可以触发单击，也可以触发双击。双击的规则是这样的，两次单击，如果第 1 次单击与第 2 次单击的时间比较“短”的时候，则这两次单击就构成双击。编写这个程序的最大亮点是如何控制好第 1 次单击与第 2 次单击的时间间隔。程序例程要实现的功能是：以 S1 按键为例，（1）单击改变 LED 灯的显示状态。单击一次 LED 从原来“灭”的状态变成“亮”的状态，或者从原来“亮”的状态变成“灭”的状态，依次循环切换。（2）双击则蜂鸣器发出“嘀”的一声。代码如下：

```
#include "REG52.H"

#define KEY_VOICE_TIME    50
#define KEY_SHORT_TIME    20    //按键去抖动的“滤波”时间
#define KEY_INTERVAL_TIME 80    //连续两次单击之间的最大有效时间。因为是矩阵，80 不一定是 80ms

void T0_time();
void SystemInitial(void) ;
void Delay(unsigned long u32DelayTime) ;
void PeripheralInitial(void) ;

void BeepOpen(void);
void BeepClose(void);
void LedOpen(void);
void LedClose(void);

void VoiceScan(void);
void KeyScan(void);
void SingleKeyTask(void);    //单击按键任务函数，放在主函数内
void DoubleKeyTask(void);    //双击按键任务函数，放在主函数内

sbit P3_4=P3^4;            //蜂鸣器
sbit P1_4=P1^4;            //LED

sbit ROW_INPUT1=P2^2;    //第 1 行输入口。
sbit ROW_INPUT2=P2^1;    //第 2 行输入口。
sbit ROW_INPUT3=P2^0;    //第 3 行输入口。

sbit COLUMN_OUTPUT1=P2^5;    //第 1 列输出口。
sbit COLUMN_OUTPUT2=P2^4;    //第 2 列输出口。
sbit COLUMN_OUTPUT3=P2^3;    //第 3 列输出口。

volatile unsigned char vGu8BeepTimerFlag=0;
volatile unsigned int vGu16BeepTimerCnt=0;
```

```

unsigned char Gu8LedStatus=0; //记录 LED 灯的状态，0 代表灭，1 代表亮
volatile unsigned char vGu8SingleKeySec=0; //单击按键的触发序号
volatile unsigned char vGu8DoubleKeySec=0; //双击按键的触发序号

void main()
{
    SystemInitial();
    Delay(10000);
    PeripheralInitial();
    while(1)
    {
        SingleKeyTask(); //单击按键任务函数
        DoubleKeyTask(); //双击按键任务函数
    }
}

/* 注释一：
* 矩阵按键扫描的详细过程：
* 先输出某 1 列低电平，其它 2 列输出高电平，延时等待 2ms 后（等此 3 列输出同步稳定），
* 再分别判断 3 行的输入 IO 口， 如果发现哪一行是低电平，就说明对应的某个按键被触发。
* 依次循环切换输出的 3 种状态，并且分别判断输入的 3 行，就可以检测完 9 个按键。矩阵按键的
* 去抖动处理方法跟我前面讲的独立按键去抖动方法是一样的。
*/

/* 注释二：
* 双击按键扫描的详细过程：
* 第一步：平时没有按键被触发时，按键的自锁标志, 去抖动延时计数器一直被清零。
* 如果之前已经有按键触发过 1 次单击，那么启动时间间隔计数器 Su16KeyIntervalCnt1，
* 在 KEY_INTERVAL_TIME 这个允许的时间差范围内，如果一直没有第 2 次单击触发，
* 则把累加按键触发的次数 Su8KeyTouchCnt1 也清零，上一次累计的单击数被清零，
* 就意味着下一次新的双击必须重新开始累加两次单击数。
* 第二步：一旦有按键被按下，去抖动延时计数器开始在定时中断函数里累加，在还没累加到
* 阈值 KEY_SHORT_TIME 时，如果在这期间由于受外界干扰或者按键抖动，而使
* IO 口突然瞬间触发成高电平，这个时候马上把延时计数器 Su16KeyCnt
* 清零了, 这个过程非常巧妙，非常有效地去除瞬间的杂波干扰，以后凡是用到开关感应器的时候，
* 都可以用类似这样的方法去干扰。
* 第三步：如果按键按下的时间超过了阈值 KEY_SHORT_TIME，马上把自锁标志 Su8KeyLock 置 1，
* 防止按住按键不松手后一直触发。与此同时，累加 1 次按键次数，如果按键次数累加有 2 次，
* 则认为触发双击按键，并把编号 vGu8DoubleKeySec 赋值。
* 第四步：等按键松开后，自锁标志 Su8KeyLock 及时清零解锁，为下一次自锁做准备。并且累加间隔时间，
* 防止两次按键的间隔时间太长。如果连续 2 次单击的间隔时间太长达到了 KEY_INTERVAL_TIME
* 的长度，立即清零当前按键次数的计数器，这样意味着上一次的累加单击数无效，下一次双击
* 必须重新累加新的单击数。

```

```
*/
```

```
void KeyScan(void) //此函数放在定时中断里每 1ms 扫描一次
{
    static unsigned char Su8KeyLock=0;
    static unsigned int  Su16KeyCnt=0;
    static unsigned char Su8KeyStep=1;

    static unsigned char Su8ColumnRecord=0; //用来切换当前列的输出

    static unsigned char Su8KeyTouchCnt1;      //S1 按键的次数记录
    static unsigned int  Su16KeyIntervalCnt1;   //S1 按键的间隔时间计数器

    switch(Su8KeyStep)
    {
        case 1:
            if(0==Su8ColumnRecord) //按键扫描输出第一列低电平
            {
                COLUMN_OUTPUT1=0;
                COLUMN_OUTPUT2=1;
                COLUMN_OUTPUT3=1;
            }
            else if(1==Su8ColumnRecord) //按键扫描输出第二列低电平
            {
                COLUMN_OUTPUT1=1;
                COLUMN_OUTPUT2=0;
                COLUMN_OUTPUT3=1;
            }
            else //按键扫描输出第三列低电平
            {
                COLUMN_OUTPUT1=1;
                COLUMN_OUTPUT2=1;
                COLUMN_OUTPUT3=0;
            }
            Su16KeyCnt=0; //延时计数器清零
            Su8KeyStep++; //切换到下一个运行步骤
            break;

        case 2: //延时等待 2ms 后（等此 3 列输出同步稳定）。不是按键的去抖动延时。
            Su16KeyCnt++;
            if(Su16KeyCnt>=2)
            {
                Su16KeyCnt=0;
                Su8KeyStep++; //切换到下一个运行步骤
            }
        }
    }
}
```

```

    }
    break;

case 3:
    if (1==ROW_INPUT1&&1==ROW_INPUT2&&1==ROW_INPUT3)
    {
        Su8KeyStep=1; //如果没有按键按下，返回到第一个运行步骤重新开始扫描!!!!!!
        Su8KeyLock=0; //按键自锁标志清零
        Su16KeyCnt=0; //按键去抖动延时计数器清零，此行非常巧妙

        if (Su8KeyTouchCnt1>=1) //之前已经有按键触发过一次，启动间隔时间的计数器
        {
            Su16KeyIntervalCnt1++; //按键间隔的时间计数器累加
            if (Su16KeyIntervalCnt1>=KEY_INTERVAL_TIME) //达到最大允许的间隔时间，溢出无效
            {
                Su16KeyIntervalCnt1=0; //时间计数器清零
                Su8KeyTouchCnt1=0; //清零按键的按下的次数，因为间隔时间溢出无效
            }
        }

        Su8ColumnRecord++; //输出下一列
        if (Su8ColumnRecord>=3)
        {
            Su8ColumnRecord=0; //依次输出完第 3 列之后，继续从第 1 列开始输出低电平
        }
    }
    else if (0==Su8KeyLock) //有按键按下，且是第一次触发
    {
        if (0==ROW_INPUT1&&1==ROW_INPUT2&&1==ROW_INPUT3)
        {
            Su16KeyCnt++; //去抖动延时计数器
            if (Su16KeyCnt>=KEY_SHORT_TIME)
            {
                Su8KeyLock=1; //自锁置 1, 避免一直触发, 只有松开按键, 此标志位才会被清零

                if (0==Su8ColumnRecord) //第 1 列输出低电平
                {
                    Su16KeyIntervalCnt1=0; //按键有效间隔的时间计数器清零
                    Su8KeyTouchCnt1++; //记录当前单击的次数
                    if (1==Su8KeyTouchCnt1) //只按了 1 次
                    {
                        vGu8SingleKeySec=1; //单击任务，触发 1 号键 对应 S1 键
                    }
                    else if (Su8KeyTouchCnt1>=2) //连续按了两次以上

```

```

        {
            Su8KeyTouchCnt1=0;    //统计按键次数清零
            vGu8SingleKeySec=1;    //单击任务，触发 1 号键 对应 S1 键
            vGu8DoubleKeySec=1;    //双击任务，触发 1 号键 对应 S1 键
        }
    }
    else if(1==Su8ColumnRecord) //第 2 列输出低电平
    {
        vGu8SingleKeySec=2; //触发 2 号键 对应 S2 键
    }
    else if(2==Su8ColumnRecord) //第 3 列输出低电平
    {
        vGu8SingleKeySec=3; //触发 3 号键 对应 S3 键
    }
}

}
else if(1==ROW_INPUT1&&0==ROW_INPUT2&&1==ROW_INPUT3)
{
    Su16KeyCnt++; //去抖动延时计数器
    if(Su16KeyCnt>=KEY_SHORT_TIME)
    {
        Su8KeyLock=1;//自锁置 1, 避免一直触发, 只有松开按键, 此标志位才会被清零

        if(0==Su8ColumnRecord) //第 1 列输出低电平
        {
            vGu8SingleKeySec=4; //触发 4 号键 对应 S4 键
        }
        else if(1==Su8ColumnRecord) //第 2 列输出低电平
        {
            vGu8SingleKeySec=5; //触发 5 号键 对应 S5 键
        }
        else if(2==Su8ColumnRecord) //第 3 列输出低电平
        {
            vGu8SingleKeySec=6; //触发 6 号键 对应 S6 键
        }
    }
}
else if(1==ROW_INPUT1&&1==ROW_INPUT2&&0==ROW_INPUT3)
{
    Su16KeyCnt++; //去抖动延时计数器
    if(Su16KeyCnt>=KEY_SHORT_TIME)
    {
        Su8KeyLock=1;//自锁置 1, 避免一直触发, 只有松开按键, 此标志位才会被清零
    }
}

```

```

        if(0==Su8ColumnRecord) //第 1 列输出低电平
        {
            vGu8SingleKeySec=7; //触发 7 号键 对应 S7 键
        }
        else if(1==Su8ColumnRecord) //第 2 列输出低电平
        {
            vGu8SingleKeySec=8; //触发 8 号键 对应 S8 键
        }
        else if(2==Su8ColumnRecord) //第 3 列输出低电平
        {
            vGu8SingleKeySec=9; //触发 9 号键 对应 S9 键
        }
    }
}

}

break;
}

}

}

void SingleKeyTask(void) //按键单击的任务函数，放在主函数内
{
    if(0==vGu8SingleKeySec)
    {
        return; //按键的触发序号是 0 意味着无按键触发，直接退出当前函数，不执行此函数下面的代码
    }

    switch(vGu8SingleKeySec) //根据不同的按键触发序号执行对应的代码
    {
        case 1: //S1 按键的单击任务

            //通过 Gu8LedStatus 的状态切换，来反复切换 LED 的“灭”与“亮”的状态
            if(0==Gu8LedStatus)
            {
                Gu8LedStatus=1; //标识并且更改当前 LED 灯的状态。0 就变成 1。
                LedOpen(); //点亮 LED
            }
            else
            {
                Gu8LedStatus=0; //标识并且更改当前 LED 灯的状态。1 就变成 0。
                LedClose(); //关闭 LED
            }
        }
    }
}

```



```

        vGu8SingleKeySec=0; //响应按键服务处理程序后，按键编号必须清零，避免一直触发
        break;

    default: //其它按键触发的单击

        vGu8SingleKeySec=0; //响应按键服务处理程序后，按键编号必须清零，避免一直触发
        break;

    }
}

void DoubleKeyTask(void) //双击按键任务函数，放在主函数内
{
    if (0==vGu8DoubleKeySec)
    {
        return; //按键的触发序号是 0 意味着无按键触发，直接退出当前函数，不执行此函数下面的代码
    }

    switch(vGu8DoubleKeySec) //根据不同的按键触发序号执行对应的代码
    {
        case 1: //S1 按键的双击任务

            vGu8BeepTimerFlag=0;
            vGu16BeepTimerCnt=KEY_VOICE_TIME; //触发双击后，发出“嘀”一声
            vGu8BeepTimerFlag=1;

            vGu8DoubleKeySec=0; //响应按键服务处理程序后，按键编号必须清零，避免一致触发
            break;

    }
}

void T0_time() interrupt 1
{
    VoiceScan();
    KeyScan(); //按键识别的驱动函数

    TH0=0xfc;
    TL0=0x66;
}

void SystemInitial(void)
{

```

```

    TMOD=0x01;
    TH0=0xfc;
    TL0=0x66;
    EA=1;
    ET0=1;
    TR0=1;
}

void Delay(unsigned long u32DelayTime)
{
    for(;u32DelayTime>0;u32DelayTime--);
}

void PeripheralInitial(void)
{
    /* 注释三:
    * 把 LED 的初始化放在 PeripheralInitial 而不是放在 SystemInitial, 是因为 LED 显示内容对上电
    * 瞬间的要求不高。但是, 如果是控制继电器, 则应该把继电器的输出初始化放在 SystemInitial。
    */

    //根据 Gu8LedStatus 的值来初始化 LED 当前的显示状态, 0 代表灭, 1 代表亮
    if(0==Gu8LedStatus)
    {
        LedClose(); //关闭 LED
    }
    else
    {
        LedOpen(); //点亮 LED
    }
}

void BeepOpen(void)
{
    P3_4=0;
}

void BeepClose(void)
{
    P3_4=1;
}

void LedOpen(void)
{
    P1_4=0;
}

```

```

}

void LedClose(void)
{
    P1_4=1;
}

void VoiceScan(void)
{

    static unsigned char Su8Lock=0;

    if(1==vGu8BeepTimerFlag&&vGu16BeepTimerCnt>0)
    {
        if(0==Su8Lock)
        {
            Su8Lock=1;
            BeepOpen();
        }
        else
        {

            vGu16BeepTimerCnt--;

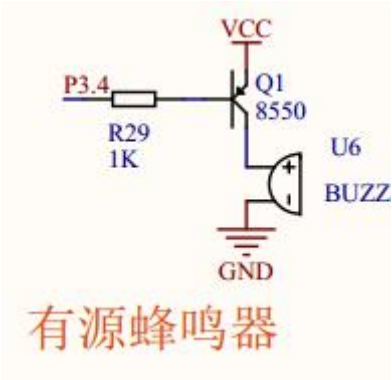
            if(0==vGu16BeepTimerCnt)
            {
                Su8Lock=0;
                BeepClose();
            }

        }
    }
}

```

第一百零二节： 两个“任意行输入” 矩阵按键的“有序” 组合触发。

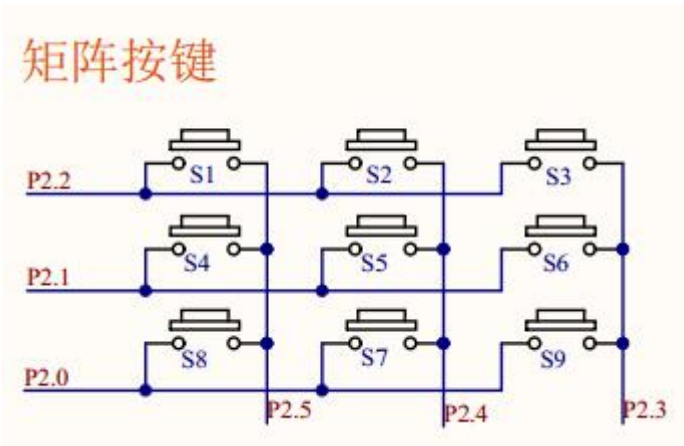
【102.1 “异行输入” “同行输入” “有序”。】



上图 102.1.1 有源蜂鸣器电路



上图 102.1.2 LED 电路



上图 102.1.3 3\*3 矩阵按键的电路

“任意行输入”是指能兼容“异行输入”与“同行输入”这两种按键状态。

何谓“异行输入”何谓“同行输入”？如上图矩阵按键，P2.2，P2.1，P2.0 是输入行，P2.5，P2.4，P2.3 是输出列。以 S1 按键为例，很明显，S2 和 S3 都是属于 S1 的“同行输入”，都是属于 P2.2 的输入行。除了 S2 和 S3 以外，其它所有的按键都是 S1 的“异行输入”，比如 S5 按键就是 S1 的“异行输入”，因为 S1 是属于 P2.2 的输入行，而 S5 是属于 P2.1 的输入行。

何谓“有序”组合触发？就是两个按键的触发必须遵守“先后顺序”才能构成“组合触发”。比如，像电脑的复制快捷键（Ctrl+C），你必须先按住 Ctrl 再按住 C 此时“复制快捷键”才有效，如果你先按住 C 再按住 Ctrl 此时“复制快捷键”无效。

“异行输入”与“同行输入”，相比之下，“同行输入”更难更有代表性，如果把“同行输入”的程序写出来了，那么完全按“同行输入”的思路，就可以把“异行输入”的程序写出来。因此，只要把“同行输入”的程序写出来了，也就意味着“任意行输入”的程序也就实现了。本节以 S1 和 S2 的“同行输入”按键为例，S1 是主键，类似复制快捷键的 Ctrl 键；S2 是从键，类似复制快捷键的 C 键。要触发组合键（S1+S2），必须先按住 S1 再按 S2 才有效。功能如下：（1）S1 每单击一次，LED 要么从“灭”变成“亮”，要么从“亮”变成“灭”，在两种状态之间切换。（2）如果先按住 S1 再按 S2，就认为构造了“有序”组合键，蜂鸣器发出“嘀”的一声。

```
#include "REG52.H"

#define KEY_VOICE_TIME    50
#define KEY_SHORT_TIME    20

void T0_time();
void SystemInitial(void) ;
void Delay(unsigned long u32DelayTime) ;
void PeripheralInitial(void) ;

void BeepOpen(void);
void BeepClose(void);
void LedOpen(void);
void LedClose(void);

void VoiceScan(void);
void KeyScan(void);
void SingleKeyTask(void);
void DoubleKeyTask(void);

sbit P3_4=P3^4;
sbit P1_4=P1^4;

sbit ROW_INPUT1=P2^2; //第 1 行输入口。
sbit ROW_INPUT2=P2^1; //第 2 行输入口。
sbit ROW_INPUT3=P2^0; //第 3 行输入口。
```

```
sbit COLUMN_OUTPUT1=P2^5; //第 1 列输出口。
sbit COLUMN_OUTPUT2=P2^4; //第 2 列输出口。
sbit COLUMN_OUTPUT3=P2^3; //第 3 列输出口。
```

```
volatile unsigned char vGu8BeepTimerFlag=0;
volatile unsigned int vGu16BeepTimerCnt=0;
```

```
unsigned char Gu8LedStatus=0;
volatile unsigned char vGu8SingleKeySec=0;
volatile unsigned char vGu8DoubleKeySec=0;
```

```
void main()
{
    SystemInitial();
    Delay(10000);
    PeripheralInitial();
    while(1)
    {
        SingleKeyTask();
        DoubleKeyTask();
    }
}
```

/\* 注释一：

```
* 两个“任意行输入”矩阵按键“有序”触发的两个最关键地方：
* （1）当 S1 按键被按下单击触发之后，“马上更新输出列的信号状态”，然后切换到后面的步骤。
* （2）在后面的步骤里，进入到 S1 和 S2 两个按键的轮番循环监控之中，如果发现 S1 按键率先
* 被松开了，就把步骤切换到开始的第一步，重新开始新一轮的按键扫描。
* （3）按照这个模板，只需“更改不同的列输出，判断不同的行输入”，就可以实现“任意行输入”
* 矩阵按键的“有序”组合触发。
*/
```

```
void KeyScan(void) //此函数放在定时中断里每 1ms 扫描一次
```

```
{
    static unsigned char Su8KeyLock=0;
    static unsigned int Su16KeyCnt=0;
    static unsigned char Su8KeyStep=1;

    static unsigned char Su8ColumnRecord=0;

    switch(Su8KeyStep)
    {
        case 1:
```

```

        if (0==Su8ColumnRecord)
        {
            COLUMN_OUTPUT1=0;
            COLUMN_OUTPUT2=1;
            COLUMN_OUTPUT3=1;
        }
        else if (1==Su8ColumnRecord)
        {
            COLUMN_OUTPUT1=1;
            COLUMN_OUTPUT2=0;
            COLUMN_OUTPUT3=1;
        }
        else
        {
            COLUMN_OUTPUT1=1;
            COLUMN_OUTPUT2=1;
            COLUMN_OUTPUT3=0;
        }
        Su16KeyCnt=0;
        Su8KeyStep++;
        break;

case 2:      //等待列输出稳定，但不是去抖动延时
        Su16KeyCnt++;
        if (Su16KeyCnt>=2)
        {
            Su16KeyCnt=0;
            Su8KeyStep++;
        }
        break;

case 3:
        if (1==ROW_INPUT1&&1==ROW_INPUT2&&1==ROW_INPUT3)
        {
            Su8KeyStep=1;
            Su8KeyLock=0;
            Su16KeyCnt=0;

            Su8ColumnRecord++;
            if (Su8ColumnRecord>=3)
            {
                Su8ColumnRecord=0;
            }
        }
    }

```

```

else if(0==Su8KeyLock)
{
    if(0==ROW_INPUT1&&1==ROW_INPUT2&&1==ROW_INPUT3)
    {
        Su16KeyCnt++;
        if(Su16KeyCnt>=KEY_SHORT_TIME)
        {
            Su8KeyLock=1;

            if(0==Su8ColumnRecord)
            {
                vGu8SingleKeySec=1;    //单击任务，触发 1 号键 对应 S1 键

                // “马上更新输出列的信号状态”
                COLUMN_OUTPUT1=1;
                COLUMN_OUTPUT2=0;    //列 2 也输出 0，非常关键的代码！
                COLUMN_OUTPUT3=1;

                Su16KeyCnt=0; //去抖动延时清零，为下一步计时做准备
                Su8KeyStep++; //切换到下一步步骤
            }
            else if(1==Su8ColumnRecord)
            {
                vGu8SingleKeySec=2;
            }
            else if(2==Su8ColumnRecord)
            {
                vGu8SingleKeySec=3;
            }
        }
    }

}
else if(1==ROW_INPUT1&&0==ROW_INPUT2&&1==ROW_INPUT3)
{
    Su16KeyCnt++;
    if(Su16KeyCnt>=KEY_SHORT_TIME)
    {
        Su8KeyLock=1;

        if(0==Su8ColumnRecord)
        {
            vGu8SingleKeySec=4;
        }
        else if(1==Su8ColumnRecord)

```



```

        {
            vGu8SingleKeySec=5;
        }
        else if (2==Su8ColumnRecord)
        {
            vGu8SingleKeySec=6;
        }
    }
}
else if (1==ROW_INPUT1&&1==ROW_INPUT2&&0==ROW_INPUT3)
{
    Su16KeyCnt++;
    if (Su16KeyCnt>=KEY_SHORT_TIME)
    {
        Su8KeyLock=1;
        if (0==Su8ColumnRecord)
        {
            vGu8SingleKeySec=7;
        }
        else if (1==Su8ColumnRecord)
        {
            vGu8SingleKeySec=8;
        }
        else if (2==Su8ColumnRecord)
        {
            vGu8SingleKeySec=9;
        }
    }
}

}

break;
case 4:          //等待列输出稳定，但不是去抖动延时
    Su16KeyCnt++;
    if (Su16KeyCnt>=2)
    {
        Su16KeyCnt=0;
        Su8KeyLock=0;    //关键语句！自锁清零，为下一步自锁组合按键做准备
        Su8KeyStep++;
    }
    break;

case 5:    //判断 S2 按键
    if (1==ROW_INPUT1&&1==ROW_INPUT2&&1==ROW_INPUT3) //S2 按键没有被按下

```

```

{
    Su8KeyLock=0;
    Su16KeyCnt=0;

    // “马上更新输出列的信号状态”
    COLUMN_OUTPUT1=0;    //列 1 输出 0，非常关键的代码！
    COLUMN_OUTPUT2=1;
    COLUMN_OUTPUT3=1;

    Su8KeyStep++;    //切换到下一个步骤，监控 S1 是否率先已经松开
}
else if(0==Su8KeyLock)
{
    if(0==ROW_INPUT1&&1==ROW_INPUT2&&1==ROW_INPUT3) //S2 按键被按下
    {
        Su16KeyCnt++;
        if(Su16KeyCnt>=KEY_SHORT_TIME)
        {
            Su8KeyLock=1;    //组合按键的自锁
            vGu8DoubleKeySec=1;    //触发组合按键(S1+S2)
        }
    }
}

break;

case 6:    //等待列输出稳定，但不是去抖动延时
    Su16KeyCnt++;
    if(Su16KeyCnt>=2)
    {
        Su16KeyCnt=0;
        Su8KeyLock=0;    //关键语句！自锁清零，为下一步自锁组合按键做准备
        Su8KeyStep++;
    }
    break;

case 7:    //监控 S1 按键是否率先已经松开
    if(1==ROW_INPUT1&&1==ROW_INPUT2&&1==ROW_INPUT3)
    {
        Su16KeyCnt=0;
        Su8KeyLock=0;
        Su8KeyStep=1;    //如果 S1 按键已经松开，返回到第一个运行步骤重新开始扫描
    }
}

```

```

        Su8ColumnRecord++;
        if (Su8ColumnRecord>=3)
        {
            Su8ColumnRecord=0;
        }
    }
    else
    {
        // “马上更新输出列的信号状态”
        COLUMN_OUTPUT1=1;
        COLUMN_OUTPUT2=0;    //列 2 输出 0，非常关键的代码！
        COLUMN_OUTPUT3=1;
        Su8KeyStep=4;    //如果 S1 按键没有松开，继续返回判断 S2 是否已按下
    }
    break;
}

}

void SingleKeyTask(void)
{
    if (0==vGu8SingleKeySec)
    {
        return;
    }

    switch(vGu8SingleKeySec)
    {
        case 1:    //S1 按键的单击任务，更改 LED 灯的显示状态

            if (0==Gu8LedStatus)
            {
                Gu8LedStatus=1;
                LedOpen();
            }
            else
            {
                Gu8LedStatus=0;
                LedClose();
            }

            vGu8SingleKeySec=0;
            break;

```

```

        default:

            vGu8SingleKeySec=0;
            break;

    }
}

void DoubleKeyTask(void)
{
    if(0==vGu8DoubleKeySec)
    {
        return;
    }

    switch(vGu8DoubleKeySec)
    {
        case 1:    //S1 与 S2 的组合按键触发，发出“嘀”一声

            vGu8BeepTimerFlag=0;
            vGu16BeepTimerCnt=KEY_VOICE_TIME;
            vGu8BeepTimerFlag=1;

            vGu8DoubleKeySec=0;
            break;

    }
}

void T0_time() interrupt 1
{
    VoiceScan();
    KeyScan();

    TH0=0xfc;
    TL0=0x66;
}

void SystemInitial(void)
{
    TMOD=0x01;
    TH0=0xfc;

```

```

    TL0=0x66;
    EA=1;
    ET0=1;
    TR0=1;
}

void Delay(unsigned long u32DelayTime)
{
    for(;u32DelayTime>0;u32DelayTime--);
}

void PeripheralInitial(void)
{
    if(0==Gu8LedStatus)
    {
        LedClose();
    }
    else
    {
        LedOpen();
    }
}

void BeepOpen(void)
{
    P3_4=0;
}

void BeepClose(void)
{
    P3_4=1;
}

void LedOpen(void)
{
    P1_4=0;
}

void LedClose(void)
{
    P1_4=1;
}

void VoiceScan(void)

```

```
{

    static unsigned char Su8Lock=0;

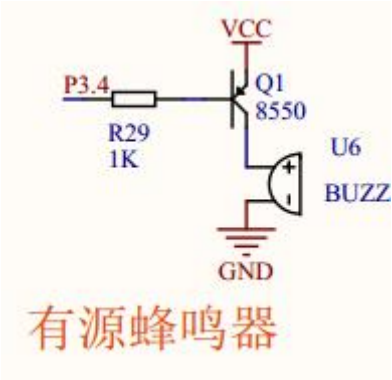
    if(1==vGu8BeepTimerFlag&&vGu16BeepTimerCnt>0)
    {
        if(0==Su8Lock)
        {
            Su8Lock=1;
            BeepOpen();
        }
        else
        {

            vGu16BeepTimerCnt--;

            if(0==vGu16BeepTimerCnt)
            {
                Su8Lock=0;
                BeepClose();
            }
        }
    }
}
```

第一百零三节： 两个“任意行输入” 矩阵按键的“无序” 组合触发。

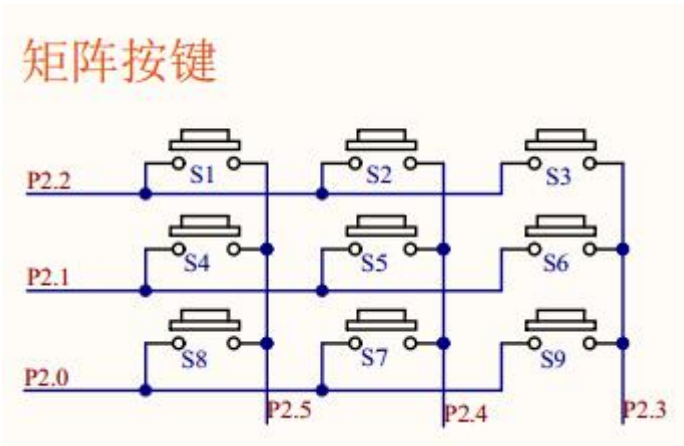
【103.1 “无序” 组合触发。】



上图 103.1.1 有源蜂鸣器电路



上图 103.1.2 LED 电路



上图 103.1.3 3\*3 矩阵按键的电路

“无序”是指两个组合按键不分先后顺序，都能构成组合触发。比如，要触发组合键（S1+S2），先按 S1 再按 S2，或者先按 S2 再按 S1，功能都是一样的。

本节程序功能如下：（1）S1 每单击一次，P1.4 所在的 LED 要么从“灭”变成“亮”，要么从“亮”变成“灭”，在两种状态之间切换。（2）S2 每单击一次，P1.5 所在的 LED 要么从“灭”变成“亮”，要么从“亮”变成“灭”，在两种状态之间切换。（3）如果先按住 S1 再按 S2，或者先按住 S2 再按 S1，都认为构造了“无序”组合键，蜂鸣器发出“嘀”的一声。

```
#include "REG52.H"

#define KEY_VOICE_TIME    50
#define KEY_SHORT_TIME    20

void TO_time();
void SystemInitial(void) ;
void Delay(unsigned long u32DelayTime) ;
void PeripheralInitial(void) ;

void BeepOpen(void);
void BeepClose(void);
void LedOpen_P1_4(void);
void LedClose_P1_4(void);
void LedOpen_P1_5(void);
void LedClose_P1_5(void);

void VoiceScan(void);
void KeyScan(void);
void SingleKeyTask(void);
void DoubleKeyTask(void);

sbit P3_4=P3^4;
sbit P1_4=P1^4;    //P1.4 所在的 LED
sbit P1_5=P1^5;    //P1.5 所在的 LED

sbit ROW_INPUT1=P2^2; //第 1 行输入口。
sbit ROW_INPUT2=P2^1; //第 2 行输入口。
sbit ROW_INPUT3=P2^0; //第 3 行输入口。

sbit COLUMN_OUTPUT1=P2^5; //第 1 列输出口。
sbit COLUMN_OUTPUT2=P2^4; //第 2 列输出口。
sbit COLUMN_OUTPUT3=P2^3; //第 3 列输出口。

volatile unsigned char vGu8BeepTimerFlag=0;
```



```

volatile unsigned int vGu16BeepTimerCnt=0;

unsigned char Gu8LedStatus_P1_4=0; //P1.4 所在的 LED 的状态
unsigned char Gu8LedStatus_P1_5=0; //P1.5 所在的 LED 的状态
volatile unsigned char vGu8SingleKeySec=0;
volatile unsigned char vGu8DoubleKeySec=0;

void main()
{
    SystemInitial();
    Delay(10000);
    PeripheralInitial();
    while(1)
    {
        SingleKeyTask();
        DoubleKeyTask();
    }
}

/* 注释一：
* 矩阵按键“无序”触发的两个最关键地方：
* （1）如果是 S1 按键先被按下并且单击触发之后，“马上更新输出列的信号状态”，然后切换到
* “S1 后面所在的步骤里”，进入到 S1 和 S2 两个按键的轮番循环监控之中，如果发现 S1 按键率先
* 被松开了，就把步骤切换到开始的第一步，重新开始新一轮的按键扫描。
* （2）如果是 S2 按键先被按下并且单击触发之后，“马上更新输出列的信号状态”，然后切换到
* “S2 后面所在的步骤里”，进入到 S1 和 S2 两个按键的轮番循环监控之中，如果发现 S2 按键率先
* 被松开了，就把步骤切换到开始的第一步，重新开始新一轮的按键扫描。
* （3）上面两个描述中的两种步骤，“S1 后面所在的步骤里”和“S2 后面所在的步骤里”是分开的，
* 不共用的，这是本节破题的关键。
*/

void KeyScan(void) //此函数放在定时中断里每 1ms 扫描一次
{
    static unsigned char Su8KeyLock=0;
    static unsigned int Su16KeyCnt=0;
    static unsigned char Su8KeyStep=1;

    static unsigned char Su8ColumnRecord=0;

    switch(Su8KeyStep)
    {
        case 1:
            if(0==Su8ColumnRecord)
            {

```

```

        COLUMN_OUTPUT1=0;
        COLUMN_OUTPUT2=1;
        COLUMN_OUTPUT3=1;
    }
    else if (1==Su8ColumnRecord)
    {
        COLUMN_OUTPUT1=1;
        COLUMN_OUTPUT2=0;
        COLUMN_OUTPUT3=1;
    }
    else
    {
        COLUMN_OUTPUT1=1;
        COLUMN_OUTPUT2=1;
        COLUMN_OUTPUT3=0;
    }
    Su16KeyCnt=0;
    Su8KeyStep++;
    break;

case 2:        //等待列输出稳定，但不是去抖动延时
    Su16KeyCnt++;
    if (Su16KeyCnt>=2)
    {
        Su16KeyCnt=0;
        Su8KeyStep++;
    }
    break;

case 3:
    if (1==ROW_INPUT1&&1==ROW_INPUT2&&1==ROW_INPUT3)
    {
        Su8KeyStep=1;
        Su8KeyLock=0;
        Su16KeyCnt=0;

        Su8ColumnRecord++;
        if (Su8ColumnRecord>=3)
        {
            Su8ColumnRecord=0;
        }
    }
    else if (0==Su8KeyLock)
    {

```

```

if (0==ROW_INPUT1&&1==ROW_INPUT2&&1==ROW_INPUT3)
{
    Su16KeyCnt++;
    if (Su16KeyCnt>=KEY_SHORT_TIME)
    {
        Su8KeyLock=1;

        if (0==Su8ColumnRecord)
        {
            vGu8SingleKeySec=1;    //单击任务，触发 1 号键 对应 S1 键

            // “马上更新输出列的信号状态”
            COLUMN_OUTPUT1=1;
            COLUMN_OUTPUT2=0;    //列 2 也输出 0，下一步监控 S2，非常关键的代码！
            COLUMN_OUTPUT3=1;

            Su16KeyCnt=0;    //去抖动延时清零，为下一步计时做准备
            Su8KeyStep=4;    //切换到 “S1 后面所在的步骤里”，破题的关键!!!
        }
        else if (1==Su8ColumnRecord)
        {
            vGu8SingleKeySec=2;    //单击任务，触发 2 号键 对应 S2 键

            // “马上更新输出列的信号状态”
            COLUMN_OUTPUT1=0;    //列 1 也输出 0，下一步监控 S1，非常关键的代码！
            COLUMN_OUTPUT2=1;
            COLUMN_OUTPUT3=1;

            Su16KeyCnt=0;    //去抖动延时清零，为下一步计时做准备
            Su8KeyStep=8;    //切换到 “S2 后面所在的步骤里”，破题的关键!!!
        }
        else if (2==Su8ColumnRecord)
        {
            vGu8SingleKeySec=3;
        }
    }
}

else if (1==ROW_INPUT1&&0==ROW_INPUT2&&1==ROW_INPUT3)
{
    Su16KeyCnt++;
    if (Su16KeyCnt>=KEY_SHORT_TIME)
    {
        Su8KeyLock=1;
    }
}

```

```

        if (0==Su8ColumnRecord)
        {
            vGu8SingleKeySec=4;
        }
        else if (1==Su8ColumnRecord)
        {
            vGu8SingleKeySec=5;
        }
        else if (2==Su8ColumnRecord)
        {
            vGu8SingleKeySec=6;
        }
    }
}
else if (1==ROW_INPUT1&&1==ROW_INPUT2&&0==ROW_INPUT3)
{
    Su16KeyCnt++;
    if (Su16KeyCnt>=KEY_SHORT_TIME)
    {
        Su8KeyLock=1;
        if (0==Su8ColumnRecord)
        {
            vGu8SingleKeySec=7;
        }
        else if (1==Su8ColumnRecord)
        {
            vGu8SingleKeySec=8;
        }
        else if (2==Su8ColumnRecord)
        {
            vGu8SingleKeySec=9;
        }
    }
}

}
break;

```

/\*----- “S1 后面所在的步骤里” -----\*/

```

case 4:          //等待列输出稳定，但不是去抖动延时
    Su16KeyCnt++;
    if (Su16KeyCnt>=2)
    {

```

```

        Su16KeyCnt=0;
        Su8KeyLock=0;    //关键语句！自锁清零，为下一步自锁组合按键做准备
        Su8KeyStep++;
    }
    break;

case 5:    //判断 S2 按键
    if (1==ROW_INPUT1&&1==ROW_INPUT2&&1==ROW_INPUT3) //S2 按键没有被按下
    {
        Su8KeyLock=0;
        Su16KeyCnt=0;

        // “马上更新输出列的信号状态”
        COLUMN_OUTPUT1=0;    //列 1 输出 0，下一步监控 S1，非常关键的代码！
        COLUMN_OUTPUT2=1;
        COLUMN_OUTPUT3=1;

        Su8KeyStep++;    //切换到下一个步骤，监控 S1 是否率先已经松开
    }
    else if (0==Su8KeyLock)
    {
        if (0==ROW_INPUT1&&1==ROW_INPUT2&&1==ROW_INPUT3) //S2 按键被按下
        {
            Su16KeyCnt++;
            if (Su16KeyCnt>=KEY_SHORT_TIME)
            {
                Su8KeyLock=1;    //组合按键的自锁
                vGu8DoubleKeySec=1;    //触发组合按键 (S1+S2)
            }
        }
    }

}
break;

case 6:    //等待列输出稳定，但不是去抖动延时
    Su16KeyCnt++;
    if (Su16KeyCnt>=2)
    {
        Su16KeyCnt=0;
        Su8KeyLock=0;    //关键语句！自锁清零，为下一步自锁组合按键做准备
        Su8KeyStep++;
    }
}

```

```

        break;

case 7:    //监控 S1 按键是否率先已经松开
    if (1==ROW_INPUT1&&1==ROW_INPUT2&&1==ROW_INPUT3)
    {
        Su16KeyCnt=0;
        Su8KeyLock=0;
        Su8KeyStep=1;    //如果 S1 按键已经松开，返回到第一个运行步骤重新开始扫描

        Su8ColumnRecord++;
        if (Su8ColumnRecord>=3)
        {
            Su8ColumnRecord=0;
        }
    }
    else
    {
        // “马上更新输出列的信号状态”
        COLUMN_OUTPUT1=1;
        COLUMN_OUTPUT2=0;    //列 2 输出 0，下一步监控 S2，非常关键的代码！
        COLUMN_OUTPUT3=1;
        Su8KeyStep=4;    //如果 S1 按键没有松开，继续返回判断 S2 是否已按下
    }
    break;

/*----- “S2 后面所在的步骤里” -----*/
case 8:    //等待列输出稳定，但不是去抖动延时
    Su16KeyCnt++;
    if (Su16KeyCnt>=2)
    {
        Su16KeyCnt=0;
        Su8KeyLock=0;    //关键语句！自锁清零，为下一步自锁组合按键做准备
        Su8KeyStep++;
    }
    break;

case 9:    //判断 S1 按键
    if (1==ROW_INPUT1&&1==ROW_INPUT2&&1==ROW_INPUT3) //S1 按键没有被按下
    {
        Su8KeyLock=0;
        Su16KeyCnt=0;

        // “马上更新输出列的信号状态”
        COLUMN_OUTPUT1=1;

```

```

        COLUMN_OUTPUT2=0; //列 2 输出 0，下一步监控 S2，非常关键的代码！
        COLUMN_OUTPUT3=1;

        Su8KeyStep++; //切换到下一个步骤，监控 S2 是否率先已经松开
    }
    else if(0==Su8KeyLock)
    {
        if(0==ROW_INPUT1&&1==ROW_INPUT2&&1==ROW_INPUT3) //S1 按键被按下
        {
            Su16KeyCnt++;
            if(Su16KeyCnt>=KEY_SHORT_TIME)
            {
                Su8KeyLock=1; //组合按键的自锁
                vGu8DoubleKeySec=1; //触发组合按键(S1+S2)
            }
        }
    }

    break;

case 10: //等待列输出稳定，但不是去抖动延时
    Su16KeyCnt++;
    if(Su16KeyCnt>=2)
    {
        Su16KeyCnt=0;
        Su8KeyLock=0; //关键语句！自锁清零，为下一步自锁组合按键做准备
        Su8KeyStep++;
    }
    break;

case 11: //监控 S2 按键是否率先已经松开
    if(1==ROW_INPUT1&&1==ROW_INPUT2&&1==ROW_INPUT3)
    {
        Su16KeyCnt=0;
        Su8KeyLock=0;
        Su8KeyStep=1; //如果 S2 按键已经松开，返回到第一个运行步骤重新开始扫描

        Su8ColumnRecord++;
        if(Su8ColumnRecord>=3)
        {
            Su8ColumnRecord=0;
        }
    }

```

```

    }
    else
    {
        // “马上更新输出列的信号状态”
        COLUMN_OUTPUT1=0;    //列 1 输出 0，下一步监控 S1，非常关键的代码！
        COLUMN_OUTPUT2=1;
        COLUMN_OUTPUT3=1;
        Su8KeyStep=8;    //如果 S2 按键没有松开，继续返回判断 S1 是否已按下
    }
    break;

}

}

void SingleKeyTask(void)
{
    if(0==vGu8SingleKeySec)
    {
        return;
    }

    switch(vGu8SingleKeySec)
    {
        case 1:    //S1 按键的单击任务，更改 P1.4 所在的 LED 灯的显示状态

            if(0==Gu8LedStatus_P1_4)
            {
                Gu8LedStatus_P1_4=1;
                LedOpen_P1_4();
            }
            else
            {
                Gu8LedStatus_P1_4=0;
                LedClose_P1_4();
            }

            vGu8SingleKeySec=0;
            break;

        case 2:    //S2 按键的单击任务，更改 P1.5 所在的 LED 灯的显示状态

            if(0==Gu8LedStatus_P1_5)

```



```

        {
            Gu8LedStatus_P1_5=1;
            LedOpen_P1_5();
        }
        else
        {
            Gu8LedStatus_P1_5=0;
            LedClose_P1_5();
        }

        vGu8SingleKeySec=0;
        break;

    default:

        vGu8SingleKeySec=0;
        break;

}

}

void DoubleKeyTask(void)
{
    if(0==vGu8DoubleKeySec)
    {
        return;
    }

    switch(vGu8DoubleKeySec)
    {
        case 1:        //S1 与 S2 的组合按键触发，发出“嘀”一声

            vGu8BeepTimerFlag=0;
            vGu16BeepTimerCnt=KEY_VOICE_TIME;
            vGu8BeepTimerFlag=1;

            vGu8DoubleKeySec=0;
            break;

    }

}

void T0_time() interrupt 1
{

```

```

    VoiceScan();
    KeyScan();

    TH0=0xfc;
    TL0=0x66;
}

void SystemInitial(void)
{
    TMOD=0x01;
    TH0=0xfc;
    TL0=0x66;
    EA=1;
    ET0=1;
    TR0=1;
}

void Delay(unsigned long u32DelayTime)
{
    for(;u32DelayTime>0;u32DelayTime--);
}

void PeripheralInitial(void)
{
    if(0==Gu8LedStatus_P1_4)
    {
        LedClose_P1_4();
    }
    else
    {
        LedOpen_P1_4();
    }

    if(0==Gu8LedStatus_P1_5)
    {
        LedClose_P1_5();
    }
    else
    {
        LedOpen_P1_5();
    }
}

```

```

void BeepOpen(void)
{
    P3_4=0;
}

void BeepClose(void)
{
    P3_4=1;
}

void LedOpen_P1_4(void)
{
    P1_4=0;
}

void LedClose_P1_4(void)
{
    P1_4=1;
}

void LedOpen_P1_5(void)
{
    P1_5=0;
}

void LedClose_P1_5(void)
{
    P1_5=1;
}

void VoiceScan(void)
{
    static unsigned char Su8Lock=0;

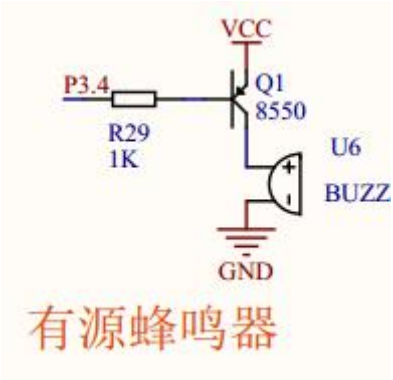
    if(1==vGu8BeepTimerFlag&&vGu16BeepTimerCnt>0)
    {
        if(0==Su8Lock)
        {
            Su8Lock=1;
            BeepOpen();
        }
        else

```

```
{  
  
    vGu16BeepTimerCnt--;  
  
    if(0==vGu16BeepTimerCnt)  
    {  
        Su8Lock=0;  
        BeepClose();  
    }  
  
}  
}
```

第一百零四节： 矩阵按键“一键两用” 的短按与长按。

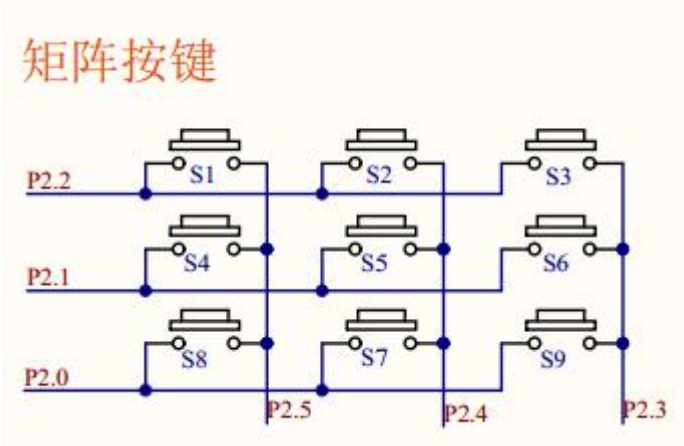
【104.1 “一键两用” 的短按与长按。】



上图 104.1.1 有源蜂鸣器电路



上图 104.1.2 LED 电路



上图 104.1.3 3\*3 矩阵按键的电路

矩阵按键与前面章节独立按键的“短按与长按”的处理思路是一样的，本节讲矩阵按键的“短按与长按”，也算是重温之前章节讲的内容。“短按与长按”的原理是依赖“按键按下的时间长度”来区分识别。“短按”是指从按下的“下降沿”到松手的“上升沿”时间，“长按”是指从按下的“下降沿”到一直按住不松手的“低电平持续时间”。本节的例程功能如下：（1）S1 每“短按”一次，LED 要么从“灭”变成“亮”，要么从“亮”变成“灭”，在两种状态之间切换。（2）S1 每“长按”一次，蜂鸣器发出“嘀”的一声。代码如下：

```
#include "REG52.H"

#define KEY_VOICE_TIME    50

#define KEY_SHORT_TIME    20    //按键的“短按”兼“滤波”的“稳定时间”
#define KEY_LONG_TIME    400    //按键的“长按”兼“滤波”的“稳定时间”

void TO_time();
void SystemInitial(void) ;
void Delay(unsigned long u32DelayTime) ;
void PeripheralInitial(void) ;

void BeepOpen(void);
void BeepClose(void);
void LedOpen_P1_4(void);
void LedClose_P1_4(void);

void VoiceScan(void);
void KeyScan(void);
void KeyTask(void);

sbit P3_4=P3^4;
sbit P1_4=P1^4;

sbit ROW_INPUT1=P2^2; //第 1 行输入口。
sbit ROW_INPUT2=P2^1; //第 2 行输入口。
sbit ROW_INPUT3=P2^0; //第 3 行输入口。

sbit COLUMN_OUTPUT1=P2^5; //第 1 列输出口。
sbit COLUMN_OUTPUT2=P2^4; //第 2 列输出口。
sbit COLUMN_OUTPUT3=P2^3; //第 3 列输出口。

volatile unsigned char vGu8BeepTimerFlag=0;
volatile unsigned int vGu16BeepTimerCnt=0;

unsigned char Gu8LedStatus_P1_4=0;
```

```
volatile unsigned char vGu8KeySec=0; //短按与长按共用一个全局变量 vGu8KeySec 来传递按键信息
```

```
void main()
{
    SystemInitial();
    Delay(10000);
    PeripheralInitial();
    while(1)
    {
        KeyTask();
    }
}
```

/\* 注释一：

\* 本节破题的关键：

\* 矩阵按键涉及的按键数量很多，但是实际项目上一般只需要少数个别按键具备这种

\* “短按”与“长按”的特殊技能，因此，在代码上，必须把这类“特殊技能按键”与

\* “大众按键”区分开来，才能相互清晰互不干扰。本节的“特殊技能按键”是 S1。

\*/

```
void KeyScan(void) //此函数放在定时中断里每 1ms 扫描一次
```

```
{
    static unsigned char Su8KeyLock=0;
    static unsigned int Su16KeyCnt=0;
    static unsigned char Su8KeyStep=1;

    static unsigned char Su8ColumnRecord=0;

    static unsigned char Su8KeyShortFlag_S1=0; //S1 按键专属的“短按”触发标志

    switch(Su8KeyStep)
    {
        case 1:
            if(0==Su8ColumnRecord)
            {
                COLUMN_OUTPUT1=0;
                COLUMN_OUTPUT2=1;
                COLUMN_OUTPUT3=1;
            }
            else if(1==Su8ColumnRecord)
            {
                COLUMN_OUTPUT1=1;
                COLUMN_OUTPUT2=0;
                COLUMN_OUTPUT3=1;
            }
        }
    }
```

```

    }
    else
    {
        COLUMN_OUTPUT1=1;
        COLUMN_OUTPUT2=1;
        COLUMN_OUTPUT3=0;
    }
    Su16KeyCnt=0;
    Su8KeyStep++;
    break;

case 2:        //等待列输出稳定，但不是去抖动延时
    Su16KeyCnt++;
    if (Su16KeyCnt>=2)
    {
        Su16KeyCnt=0;
        Su8KeyStep++;
    }
    break;

case 3:
    if (1==ROW_INPUT1&&1==ROW_INPUT2&&1==ROW_INPUT3)
    {
        Su8KeyStep=1;
        Su8KeyLock=0;
        Su16KeyCnt=0;

        if (1==Su8KeyShortFlag_S1)  //松手的时候，如果“短按”标志有效就触发一次“短按”
        {
            Su8KeyShortFlag_S1=0;    //先清零“短按”标志避免一直触发。
            vGu8KeySec=1;    //触发 S1 的“短按”
        }

        Su8ColumnRecord++;
        if (Su8ColumnRecord>=3)
        {
            Su8ColumnRecord=0;
        }
    }
    else if (0==Su8KeyLock)
    {
        //以下第 1 行，直接把 S1 按键单独扣出来，用“&&0==Su8ColumnRecord”作为筛选条件
        if (0==ROW_INPUT1&&1==ROW_INPUT2&&1==ROW_INPUT3&&0==Su8ColumnRecord)
        {

```



```

        Su16KeyCnt++;
        if (Su16KeyCnt >= KEY_SHORT_TIME) // “短按” 兼 “滤波” 的 “稳定时间”
        {
            //注意，这里不能“自锁”。后面“长按”触发的时候才“自锁”。
            Su8KeyShortFlag_S1=1;    //S1 的“短按”标志有效，待松手时触发。
        }

        if (Su16KeyCnt >= KEY_LONG_TIME) // “长按” 兼 “滤波” 的 “稳定时间”
        {
            Su8KeyLock=1;    //此时“长按”触发才“自锁”
            Su8KeyShortFlag_S1=0; //既然此时“长按”有效，那么就要废除潜在的“短按”。
            vGu8KeySec=21; //触发 S1 的“长按”
        }
    }
    else if (0==ROW_INPUT1&&1==ROW_INPUT2&&1==ROW_INPUT3)
    {
        Su16KeyCnt++;
        if (Su16KeyCnt >= KEY_SHORT_TIME)
        {
            Su8KeyLock=1;

            //既然 S1 按键已经被上面几行代码单独扣出来，这里就直接从 S2 按键开始判断
            if (1==Su8ColumnRecord)
            {
                vGu8KeySec=2;
            }
            else if (2==Su8ColumnRecord)
            {
                vGu8KeySec=3;
            }
        }
    }

    }
    else if (1==ROW_INPUT1&&0==ROW_INPUT2&&1==ROW_INPUT3)
    {
        Su16KeyCnt++;
        if (Su16KeyCnt >= KEY_SHORT_TIME)
        {
            Su8KeyLock=1;

            if (0==Su8ColumnRecord)
            {
                vGu8KeySec=4;
            }
        }
    }
}

```

```

        }
        else if(1==Su8ColumnRecord)
        {
            vGu8KeySec=5;
        }
        else if(2==Su8ColumnRecord)
        {
            vGu8KeySec=6;
        }
    }
}
else if(1==ROW_INPUT1&&1==ROW_INPUT2&&0==ROW_INPUT3)
{
    Su16KeyCnt++;
    if(Su16KeyCnt>=KEY_SHORT_TIME)
    {
        Su8KeyLock=1;
        if(0==Su8ColumnRecord)
        {
            vGu8KeySec=7;
        }
        else if(1==Su8ColumnRecord)
        {
            vGu8KeySec=8;
        }
        else if(2==Su8ColumnRecord)
        {
            vGu8KeySec=9;
        }
    }
}

}

break;

}

}

void KeyTask(void)
{
    if(0==vGu8KeySec)
    {
        return;
    }
}

```

```

}

switch(vGu8KeySec)
{
    case 1:        //S1 按键的“短按”任务，更改 P1.4 所在的 LED 灯的显示状态

        if(0==Gu8LedStatus_P1_4)
        {
            Gu8LedStatus_P1_4=1;
            LedOpen_P1_4();
        }
        else
        {
            Gu8LedStatus_P1_4=0;
            LedClose_P1_4();
        }

        vGu8KeySec=0;
        break;

    //以下 S1 按键的“长按”直接选择 case 21 的“21”，是为了不占用前排其它按键的编号。
    case 21:        //S1 按键的“长按”任务，蜂鸣器发出“嘀”一声
        vGu8BeepTimerFlag=0;
        vGu16BeepTimerCnt=KEY_VOICE_TIME; //蜂鸣器发出“嘀”一声
        vGu8BeepTimerFlag=1;

        vGu8KeySec=0;
        break;

    default:

        vGu8KeySec=0;
        break;

}

}

void T0_time() interrupt 1
{
    VoiceScan();
    KeyScan();

    TH0=0xfc;
    TL0=0x66;
}

```

```

}

void SystemInitial(void)
{
    TMOD=0x01;
    TH0=0xfc;
    TL0=0x66;
    EA=1;
    ET0=1;
    TR0=1;
}

void Delay(unsigned long u32DelayTime)
{
    for(;u32DelayTime>0;u32DelayTime--);
}

void PeripheralInitial(void)
{
    if(0==Gu8LedStatus_P1_4)
    {
        LedClose_P1_4();
    }
    else
    {
        LedOpen_P1_4();
    }
}

void BeepOpen(void)
{
    P3_4=0;
}

void BeepClose(void)
{
    P3_4=1;
}

void LedOpen_P1_4(void)
{
    P1_4=0;
}

```

```

}

void LedClose_P1_4(void)
{
    P1_4=1;
}

void VoiceScan(void)
{

    static unsigned char Su8Lock=0;

    if(1==vGu8BeepTimerFlag&&vGu16BeepTimerCnt>0)
    {
        if(0==Su8Lock)
        {
            Su8Lock=1;
            BeepOpen();
        }
        else
        {

            vGu16BeepTimerCnt--;

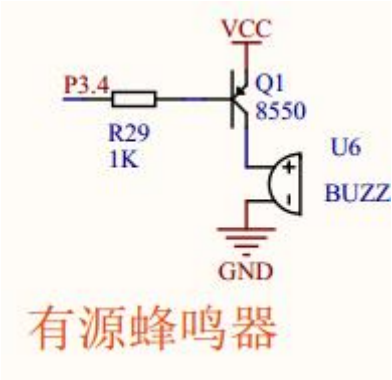
            if(0==vGu16BeepTimerCnt)
            {
                Su8Lock=0;
                BeepClose();
            }

        }
    }
}

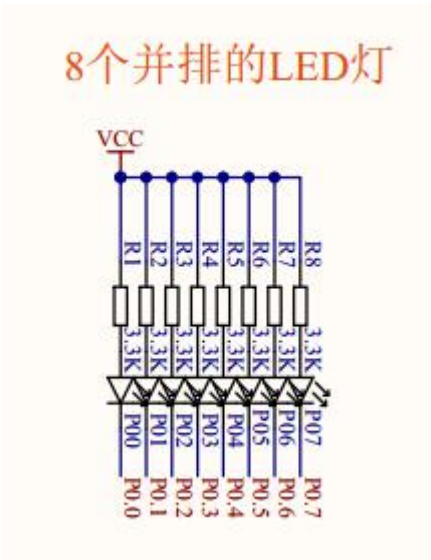
```

第一百零五节： 矩阵按键按住不松手的连续均匀触发。

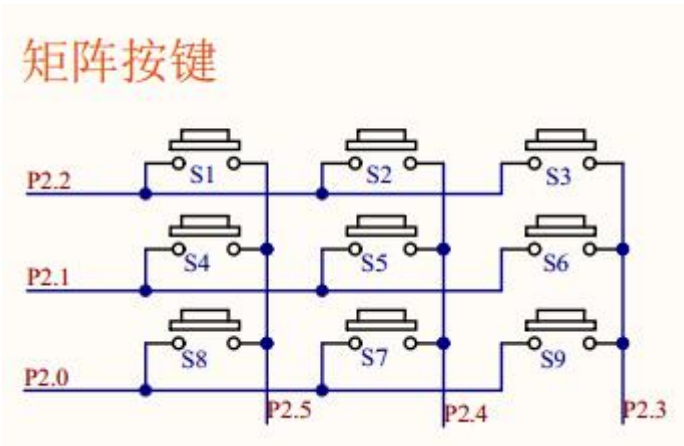
【105.1 按住不松手的连续均匀触发。】



上图 105.1.1 有源蜂鸣器电路



上图 105.1.2 LED 电路



上图 105.1.3 3\*3 矩阵按键的电路

矩阵按键与前面章节独立按键的“按住不松手的连续均匀触发”的处理思路是一样的。在电脑上删除某个文件某行文字的时候，单击一次“退格按键[Backspace]”，就删除一个文字，如果按住“退格按键[Backspace]”不松手，就会“连续均匀”的触发“删除”的功能，自动逐个把整行文字删除清空，这就是“按住不松手的连续均匀触发”应用案例之一。除此之外，在很多需要人机交互的项目中都有这样的功能，为了快速加减某个数值，按住某个按键不松手，某个数值有节奏地快速往上加或者快速往下减。这种“按住不松手连续均匀触发”的按键识别，在程序上有“3 个时间”需要留意，第 1 个是按键单击的“滤波”时间，第 2 个是按键“从单击进入连击”的间隔时间（此时间是“单击”与“连击”的分界线），第 3 个是按键“连击”的间隔时间，

本节例程实现的功能如下：（1）8 个受按键控制的跑马灯在某一时刻只有 1 个 LED 亮，每触发一次 S1 按键，“亮的 LED”就“往左边跑一步”；相反，每触发一次 S9 按键，“亮的 LED”就“往右边跑一步”。如果按住 S1 或者 S9 不松手就连续触发，“亮的 LED”就“连续跑”，一直跑到左边或者右边的尽头。（2）按键每“单击”一次 S1 或者 S9 蜂鸣器就鸣叫一次，但是，当按键“从单击进入连击”后，蜂鸣器就不鸣叫。代码如下：

```
#include "REG52.H"

#define KEY_VOICE_TIME    50

#define KEY_SHORT_TIME    20    //按键单击的“滤波”时间

#define KEY_ENTER_CONTINUITY_TIME    240    //按键“从单击进入连击”的间隔时间
#define KEY_CONTINUITY_TIME    64    //按键“连击”的间隔时间

#define BUS_P0    P0    //8 个 LED 灯一一对应单片机的 P0 口总线

void TO_time();
void SystemInitial(void) ;
void Delay(unsigned long u32DelayTime) ;
void PeripheralInitial(void) ;

void BeepOpen(void);
void BeepClose(void);

void VoiceScan(void);
void KeyScan(void);
void KeyTask(void);
void DisplayTask(void);    //显示的任务函数（LED 显示状态）

sbit P3_4=P3^4;

sbit ROW_INPUT1=P2^2;    //第 1 行输入口。
```

```

sbit ROW_INPUT2=P2^1; //第 2 行输入口。
sbit ROW_INPUT3=P2^0; //第 3 行输入口。

sbit COLUMN_OUTPUT1=P2^5; //第 1 列输出口。
sbit COLUMN_OUTPUT2=P2^4; //第 2 列输出口。
sbit COLUMN_OUTPUT3=P2^3; //第 3 列输出口。

volatile unsigned char vGu8BeepTimerFlag=0;
volatile unsigned int vGu16BeepTimerCnt=0;

unsigned char Gu8LedStatus=0; //LED 灯的状态
unsigned char Gu8DisplayUpdate=1; //显示的刷新标志

volatile unsigned char vGu8KeySec=0; //按键的触发序号
volatile unsigned char vGu8ShieldVoiceFlag=0; //屏蔽声音的标志

void main()
{
    SystemInitial();
    Delay(10000);
    PeripheralInitial();
    while(1)
    {
        KeyTask();
        DisplayTask(); //显示的任务函数（LED 显示状态）
    }
}

/* 注释一：
* Gu8DisplayUpdate 这类“显示刷新变量”在“显示框架”里是很常见的，而且屡用屡爽。
* 目的是，既能及时刷新显示，又能避免主函数“不断去执行显示代码”而影响程序效率。
*/

void DisplayTask(void) //显示的任务函数（LED 显示状态）
{
    if(1==Gu8DisplayUpdate) //需要刷新一次显示
    {
        Gu8DisplayUpdate=0; //及时清零，避免主函数“不断去执行显示代码”而影响程序效率

        //Gu8LedStatus 是左移的位数，范围（0 至 7），决定了跑马灯的显示状态。
        BUS_P0=~(1<<Gu8LedStatus); //“左移<<”之后的“取反~”，因为 LED 电路是灌入式驱动方式。
    }
}

```



```

/* 注释二：
* 本节破题的关键：
* 矩阵按键涉及的按键数量很多，但是实际项目上一般只需要少数个别按键具备这种
* “单击”与“连续均匀触发”的特殊技能，因此，在代码上，必须把这类“特殊技能按键”与
* “大众按键”区分开来，才能相互清晰互不干扰。本节的“特殊技能按键”是 S1 和 S9。
* 如果觉得本节的讲解不够详细具体，请先阅读一下前面章节“独立按键按住不松手的连续均匀触发”。
*/

void KeyScan(void) //此函数放在定时中断里每 1ms 扫描一次
{
    static unsigned char Su8KeyLock=0;
    static unsigned int Su16KeyCnt=0;
    static unsigned char Su8KeyStep=1;

    static unsigned char Su8ColumnRecord=0;

    switch(Su8KeyStep)
    {
        case 1:
            if(0==Su8ColumnRecord)
            {
                COLUMN_OUTPUT1=0;
                COLUMN_OUTPUT2=1;
                COLUMN_OUTPUT3=1;
            }
            else if(1==Su8ColumnRecord)
            {
                COLUMN_OUTPUT1=1;
                COLUMN_OUTPUT2=0;
                COLUMN_OUTPUT3=1;
            }
            else
            {
                COLUMN_OUTPUT1=1;
                COLUMN_OUTPUT2=1;
                COLUMN_OUTPUT3=0;
            }
            Su16KeyCnt=0;
            Su8KeyStep++;
            break;

        case 2: //等待列输出稳定，但不是去抖动延时
            Su16KeyCnt++;
            if(Su16KeyCnt>=2)

```

```

    {
        Su16KeyCnt=0;
        Su8KeyStep++;
    }
    break;

case 3:
    if (1==ROW_INPUT1&&1==ROW_INPUT2&&1==ROW_INPUT3)
    {
        Su8KeyStep=1;    //返回步骤 1 继续扫描
        Su8KeyLock=0;
        Su16KeyCnt=0;

        Su8ColumnRecord++;
        if (Su8ColumnRecord>=3)
        {
            Su8ColumnRecord=0;
        }
    }
    else if (0==Su8KeyLock)
    {
        if (0==ROW_INPUT1&&1==ROW_INPUT2&&1==ROW_INPUT3)
        {
            Su16KeyCnt++;
            if (Su16KeyCnt>=KEY_SHORT_TIME)
            {
                Su8KeyLock=1;

                if (0==Su8ColumnRecord)
                {
                    vGu8KeySec=1;    //触发一次单击
                    Su16KeyCnt=0;    //计时器清零，为即将来临的计时做准备
                    Su8KeyStep=4;    //跳到 S1 按键的专属区，脱离大众按键
                }
                else if (1==Su8ColumnRecord)
                {
                    vGu8KeySec=2;
                }
                else if (2==Su8ColumnRecord)
                {
                    vGu8KeySec=3;
                }
            }
        }
    }
}

```

```

}
else if (1==ROW_INPUT1&&0==ROW_INPUT2&&1==ROW_INPUT3)
{
    Su16KeyCnt++;
    if (Su16KeyCnt>=KEY_SHORT_TIME)
    {
        Su8KeyLock=1;

        if (0==Su8ColumnRecord)
        {
            vGu8KeySec=4;
        }
        else if (1==Su8ColumnRecord)
        {
            vGu8KeySec=5;
        }
        else if (2==Su8ColumnRecord)
        {
            vGu8KeySec=6;
        }
    }
}
else if (1==ROW_INPUT1&&1==ROW_INPUT2&&0==ROW_INPUT3)
{
    Su16KeyCnt++;
    if (Su16KeyCnt>=KEY_SHORT_TIME)
    {
        Su8KeyLock=1;
        if (0==Su8ColumnRecord)
        {
            vGu8KeySec=7;
        }
        else if (1==Su8ColumnRecord)
        {
            vGu8KeySec=8;
        }
        else if (2==Su8ColumnRecord)
        {
            vGu8KeySec=9;    //触发一次单击
            Su16KeyCnt=0;    //计时器清零，为即将来临的计时做准备
            Su8KeyStep=6;    //跳到 S9 按键的专属区，脱离大众按键
        }
    }
}
}

```

```

    }
    break;

/*-----S1 按键的专属区-----*/
case 4:
    if (0==ROW_INPUT1&&1==ROW_INPUT2&&1==ROW_INPUT3) //仅判断 S1 按键，避免交叉影响
    {
        Su16KeyCnt++;
        if (Su16KeyCnt>=KEY_ENTER_CONTINUITY_TIME) //该时间是“单击”与“连击”的分界线
        {
            Su16KeyCnt=0;    //计时器清零，为即将来临的计时做准备
            Su8KeyStep=5;    //S1 按键进入有节奏的连续触发
        }
    }
    else //如果期间检查到 S1 按键已经松手
    {
        Su8KeyStep=1;    //返回步骤 1 继续扫描
        Su8KeyLock=0;
        Su16KeyCnt=0;

        Su8ColumnRecord++;
        if (Su8ColumnRecord>=3)
        {
            Su8ColumnRecord=0;
        }
    }

    break;
case 5: //S1 按键进入有节奏的连续触发
    if (0==ROW_INPUT1&&1==ROW_INPUT2&&1==ROW_INPUT3) //仅判断 S1 按键，避免交叉影响
    {
        Su16KeyCnt++;
        if (Su16KeyCnt>=KEY_CONTINUITY_TIME) //该时间是“连击”的时间
        {
            Su16KeyCnt=0;    //清零，为了继续连击。
            vGu8KeySec=1;    //触发一次 S1 按键
            vGu8ShieldVoiceFlag=1; //因为连击，把当前按键触发的声音屏蔽掉
        }
    }
    else //如果期间检查到 S1 按键已经松手
    {
        Su8KeyStep=1;    //返回步骤 1 继续扫描
        Su8KeyLock=0;

```

```

        Su16KeyCnt=0;

        Su8ColumnRecord++;
        if(Su8ColumnRecord>=3)
        {
            Su8ColumnRecord=0;
        }
    }
    break;

/*-----S9 按键的专属区-----*/
case 6:
    if(1==ROW_INPUT1&&1==ROW_INPUT2&&0==ROW_INPUT3) //仅判断 S9 按键，避免交叉影响
    {
        Su16KeyCnt++;
        if(Su16KeyCnt>=KEY_ENTER_CONTINUITY_TIME)//该时间是“单击”与“连击”的分界线
        {
            Su16KeyCnt=0;    //计时器清零，为即将来临的计时做准备
            Su8KeyStep=7;    //S9 按键进入有节奏的连续触发
        }
    }
    else //如果期间检查到 S9 按键已经松手
    {
        Su8KeyStep=1;    //返回步骤 1 继续扫描
        Su8KeyLock=0;
        Su16KeyCnt=0;

        Su8ColumnRecord++;
        if(Su8ColumnRecord>=3)
        {
            Su8ColumnRecord=0;
        }
    }

    break;
case 7: //S9 按键进入有节奏的连续触发
    if(1==ROW_INPUT1&&1==ROW_INPUT2&&0==ROW_INPUT3) //仅判断 S9 按键，避免交叉影响
    {
        Su16KeyCnt++;
        if(Su16KeyCnt>=KEY_CONTINUITY_TIME) //该时间是“连击”的时间
        {
            Su16KeyCnt=0;    //清零，为了继续连击。
            vGu8KeySec=9;    //触发一次 S9 按键
            vGu8ShieldVoiceFlag=1; //因为连击，把当前按键触发的声音屏蔽掉
        }
    }

```

```

    }
}
else //如果期间检查到 S9 按键已经松手
{
    Su8KeyStep=1;    //返回步骤 1 继续扫描
    Su8KeyLock=0;
    Su16KeyCnt=0;

    Su8ColumnRecord++;
    if (Su8ColumnRecord>=3)
    {
        Su8ColumnRecord=0;
    }
}
break;

}

}

void KeyTask(void)
{
    if (0==vGu8KeySec)
    {
        return;
    }

    switch(vGu8KeySec)
    {
        case 1:    //S1 按键的任务
            if (Gu8LedStatus>0)
            {
                Gu8LedStatus--; //控制 LED “往左边跑”
                Gu8DisplayUpdate=1; //刷新显示
            }

            if (0==vGu8ShieldVoiceFlag) //声音没有被屏蔽
            {
                vGu8BeepTimerFlag=0;
                vGu16BeepTimerCnt=KEY_VOICE_TIME; //发出 “嘀” 一声
                vGu8BeepTimerFlag=1;
            }

```

```

vGu8ShieldVoiceFlag=0; //及时把屏蔽标志清零，避免平时正常的单击声音也被淹没。
vGu8KeySec=0; //响应按键服务处理程序后，按键编号必须清零，避免一致触发
break;

case 9: //S9 按键的任务
    if (Gu8LedStatus<7)
    {
        Gu8LedStatus++; //控制 LED “往右边跑”
        Gu8DisplayUpdate=1; //刷新显示
    }

    if (0==vGu8ShieldVoiceFlag) //声音没有被屏蔽
    {
        vGu8BeepTimerFlag=0;
        vGu16BeepTimerCnt=KEY_VOICE_TIME; //发出“嘀”一声
        vGu8BeepTimerFlag=1;
    }

    vGu8ShieldVoiceFlag=0; //及时把屏蔽标志清零，避免平时正常的单击声音也被淹没。
    vGu8KeySec=0; //响应按键服务处理程序后，按键编号必须清零，避免一致触发
    break;

default:

    vGu8KeySec=0;
    break;

}
}

void T0_time() interrupt 1
{
    VoiceScan();
    KeyScan();

    TH0=0xfc;
    TL0=0x66;
}

void SystemInitial(void)
{
    TMOD=0x01;
    TH0=0xfc;

```

```

    TL0=0x66;
    EA=1;
    ET0=1;
    TR0=1;
}

void Delay(unsigned long u32DelayTime)
{
    for(;u32DelayTime>0;u32DelayTime--);
}

void PeripheralInitial(void)
{
}

void BeepOpen(void)
{
    P3_4=0;
}

void BeepClose(void)
{
    P3_4=1;
}

void VoiceScan(void)
{
    static unsigned char Su8Lock=0;

    if(1==vGu8BeepTimerFlag&&vGu16BeepTimerCnt>0)
    {
        if(0==Su8Lock)
        {
            Su8Lock=1;
            BeepOpen();
        }
        else
        {

            vGu16BeepTimerCnt--;

            if(0==vGu16BeepTimerCnt)

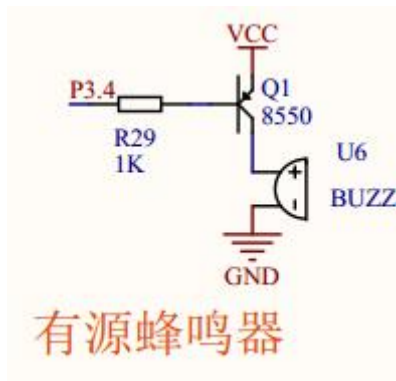
```



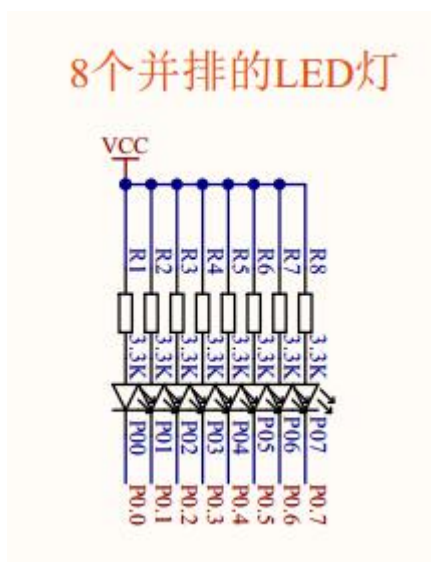
```
        {  
            Su8Lock=0;  
            BeepClose();  
        }  
    }  
}
```

第一百零六节： 矩阵按键按住不松手的“先加速后匀速”触发。

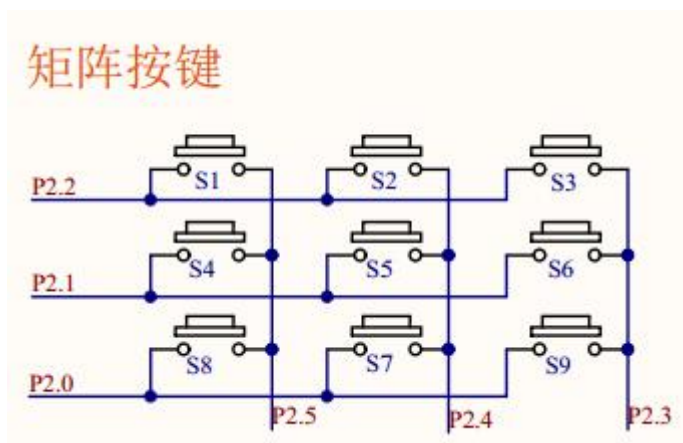
【106.1 按住不松手的先加速后匀速触发。】



上图 106.1.1 有源蜂鸣器电路



上图 106.1.2 LED 电路



上图 106.1.3 3\*3 矩阵按键的电路

矩阵按键与前面章节“独立按键按住不松手的先加速后匀速的触发”的处理思路是一样的。当“连续加”或者“连续减”的数据范围很大的时候，就需要按键的加速与匀速相结合的触发方式。“加速”是指按住按键不松手，按键刚开始触发是从慢到快的渐进过程，当“加速”到某个特别快的速度的时候，就“不再加速”，而是以该“恒定高速”进行“连续匀速”触发。这种触发方式，“加速”和“匀速”是相辅相成缺一不可的，为什么？假如没有“加速”只有“匀速”，那么刚按下按键就直接以最高速的“匀速”进行，就会跑过头，缺乏微调功能；而假如没有“匀速”只有“加速”，那么按下按键不松手后，速度就会一直不断飙升，最后失控过冲。

本节例程实现的功能如下：

- (1) 要更改一个“设置参数”（一个全局变量），参数的范围是 0 到 800。
- (2) 8 个受“设置参数”控制的跑马灯在某一时刻只有 1 个 LED 亮，每触发一次 S1 按键，该“设置参数”就自减 1，最小值为 0；相反，每触发一次 S9 按键，该“设置参数”就自加 1，最大值为 800。
- (3) LED 灯实时显示“设置参数”的范围状态：
  - 只有第 0 个 LED 灯亮：0 ≤ “设置参数” < 100。
  - 只有第 1 个 LED 灯亮：100 ≤ “设置参数” < 200。
  - 只有第 2 个 LED 灯亮：200 ≤ “设置参数” < 300。
  - 只有第 3 个 LED 灯亮：300 ≤ “设置参数” < 400。
  - 只有第 4 个 LED 灯亮：400 ≤ “设置参数” < 500。
  - 只有第 5 个 LED 灯亮：500 ≤ “设置参数” < 600。
  - 只有第 6 个 LED 灯亮：600 ≤ “设置参数” < 700。
  - 只有第 7 个 LED 灯亮：700 ≤ “设置参数” ≤ 800。
- (4) 按键每“单击”一次蜂鸣器就鸣叫一次，但是，当按键“从单击进入连击”后，蜂鸣器就不鸣叫。

```
#include "REG52.H"

#define KEY_VOICE_TIME    50

#define KEY_SHORT_TIME    20    //按键单击的“滤波”时间

#define KEY_ENTER_CONTINUITY_TIME    240    //按键“从单击进入连击”的间隔时间
#define KEY_CONTINUITY_INITIAL_TIME    64    //按键“连击”起始的预设间隔时间
#define KEY_SUB_DT_TIME    6    //按键在“加速”时每次减小的时间。
#define KEY_CONTINUITY_MIN_TIME    8    //按键时间减小到最后的“匀速”间隔时间。

#define BUS_P0    P0    //8 个 LED 灯一一对应单片机的 P0 口总线

void T0_time();
void SystemInitial(void) ;
void Delay(unsigned long u32DelayTime) ;
void PeripheralInitial(void) ;

void BeepOpen(void);
void BeepClose(void);
```

```

void VoiceScan(void);
void KeyScan(void);
void KeyTask(void);
void DisplayTask(void);    //显示的任务函数（LED 显示状态）

sbit P3_4=P3^4;

sbit ROW_INPUT1=P2^2;    //第 1 行输入口。
sbit ROW_INPUT2=P2^1;    //第 2 行输入口。
sbit ROW_INPUT3=P2^0;    //第 3 行输入口。

sbit COLUMN_OUTPUT1=P2^5;    //第 1 列输出口。
sbit COLUMN_OUTPUT2=P2^4;    //第 2 列输出口。
sbit COLUMN_OUTPUT3=P2^3;    //第 3 列输出口。

volatile unsigned char vGu8BeepTimerFlag=0;
volatile unsigned int vGu16BeepTimerCnt=0;

unsigned int Gu16SetData=0; //“设置参数”。范围从 0 到 800。LED 灯反映该当前值的范围状态
unsigned char Gu8DisplayUpdate=1; //显示的刷新标志

volatile unsigned char vGu8KeySec=0; //按键的触发序号
volatile unsigned char vGu8ShieldVoiceFlag=0; //屏蔽声音的标志

void main()
{
    SystemInitial();
    Delay(10000);
    PeripheralInitial();
    while(1)
    {
        KeyTask();
        DisplayTask();    //显示的任务函数（LED 显示状态）
    }
}

/* 注释一：
* Gu8DisplayUpdate 这类“显示刷新变量”在“显示框架”里是很常见的，而且屡用屡爽。
* 目的是，既能及时刷新显示，又能避免主函数“不断去执行显示代码”而影响程序效率。
*/

void DisplayTask(void)    //显示的任务函数（LED 显示状态）

```

```

{
    if(1==Gu8DisplayUpdate) //需要刷新一次显示
    {
        Gu8DisplayUpdate=0; //及时清零，避免主函数“不断去执行显示代码”而影响程序效率

        if(Gu16SetData<100)
        {
            BUS_P0=~(1<<0); //第 0 个灯亮
        }
        else if(Gu16SetData<200)
        {
            BUS_P0=~(1<<1); //第 1 个灯亮
        }
        else if(Gu16SetData<300)
        {
            BUS_P0=~(1<<2); //第 2 个灯亮
        }
        else if(Gu16SetData<400)
        {
            BUS_P0=~(1<<3); //第 3 个灯亮
        }
        else if(Gu16SetData<500)
        {
            BUS_P0=~(1<<4); //第 4 个灯亮
        }
        else if(Gu16SetData<600)
        {
            BUS_P0=~(1<<5); //第 5 个灯亮
        }
        else if(Gu16SetData<700)
        {
            BUS_P0=~(1<<6); //第 6 个灯亮
        }
        else
        {
            BUS_P0=~(1<<7); //第 7 个灯亮
        }
    }
}

```

/\* 注释二：

\* 本节破题的关键：

\* 矩阵按键涉及的按键数量很多，但是实际项目上一般只需要少数个别按键具备这种

\* “单击”与“先加速后均匀触发”的特殊技能，因此，在代码上，必须把这类“特殊技能按键”与

\* “大众按键”区分开来，才能相互清晰互不干扰。本节的“特殊技能按键”是 S1 和 S9。  
\* 如果觉得本节的讲解不够详细具体，请先阅读一下前面章节“独立按键按住不松手的先加速后匀速触发”。  
\*/

```
void KeyScan(void) //此函数放在定时中断里每 1ms 扫描一次
{
    static unsigned char Su8KeyLock=0;
    static unsigned int  Su16KeyCnt=0;
    static unsigned char Su8KeyStep=1;
    static unsigned int  Su16KeyContinuityTime=KEY_CONTINUITY_INITIAL_TIME; //动态时间阈值

    static unsigned char Su8ColumnRecord=0;

    switch(Su8KeyStep)
    {
        case 1:
            if(0==Su8ColumnRecord)
            {
                COLUMN_OUTPUT1=0;
                COLUMN_OUTPUT2=1;
                COLUMN_OUTPUT3=1;
            }
            else if(1==Su8ColumnRecord)
            {
                COLUMN_OUTPUT1=1;
                COLUMN_OUTPUT2=0;
                COLUMN_OUTPUT3=1;
            }
            else
            {
                COLUMN_OUTPUT1=1;
                COLUMN_OUTPUT2=1;
                COLUMN_OUTPUT3=0;
            }
            Su16KeyCnt=0;
            Su8KeyStep++;
            break;

        case 2: //等待列输出稳定，但不是去抖动延时
            Su16KeyCnt++;
            if(Su16KeyCnt>=2)
            {
                Su16KeyCnt=0;
                Su8KeyStep++;
            }
        }
    }
```

```

    }
    break;

case 3:
    if (1==ROW_INPUT1&&1==ROW_INPUT2&&1==ROW_INPUT3)
    {
        Su8KeyStep=1;    //返回步骤 1 继续扫描
        Su8KeyLock=0;
        Su16KeyCnt=0;
        Su16KeyContinuityTime=KEY_CONTINUITY_INITIAL_TIME; //动态时间阈值。重装初始值。

        Su8ColumnRecord++;
        if (Su8ColumnRecord>=3)
        {
            Su8ColumnRecord=0;
        }
    }
    else if (0==Su8KeyLock)
    {
        if (0==ROW_INPUT1&&1==ROW_INPUT2&&1==ROW_INPUT3)
        {
            Su16KeyCnt++;
            if (Su16KeyCnt>=KEY_SHORT_TIME)
            {
                Su8KeyLock=1;

                if (0==Su8ColumnRecord)
                {
                    vGu8KeySec=1;    //触发一次单击
                    Su16KeyCnt=0;    //计时器清零，为即将来临的计时做准备
                    Su8KeyStep=4;    //跳到 S1 按键的专属区，脱离大众按键
                }
                else if (1==Su8ColumnRecord)
                {
                    vGu8KeySec=2;
                }
                else if (2==Su8ColumnRecord)
                {
                    vGu8KeySec=3;
                }
            }
        }
        else if (1==ROW_INPUT1&&0==ROW_INPUT2&&1==ROW_INPUT3)

```

```

{
    Su16KeyCnt++;
    if (Su16KeyCnt>=KEY_SHORT_TIME)
    {
        Su8KeyLock=1;

        if (0==Su8ColumnRecord)
        {
            vGu8KeySec=4;
        }
        else if (1==Su8ColumnRecord)
        {
            vGu8KeySec=5;
        }
        else if (2==Su8ColumnRecord)
        {
            vGu8KeySec=6;
        }
    }
}
else if (1==ROW_INPUT1&&1==ROW_INPUT2&&0==ROW_INPUT3)
{
    Su16KeyCnt++;
    if (Su16KeyCnt>=KEY_SHORT_TIME)
    {
        Su8KeyLock=1;
        if (0==Su8ColumnRecord)
        {
            vGu8KeySec=7;
        }
        else if (1==Su8ColumnRecord)
        {
            vGu8KeySec=8;
        }
        else if (2==Su8ColumnRecord)
        {
            vGu8KeySec=9;    //触发一次单击
            Su16KeyCnt=0;    //计时器清零，为即将来临的计时做准备
            Su8KeyStep=6;    //跳到 S9 按键的专属区，脱离大众按键
        }
    }
}
}
}

```



```

        break;

/*-----S1 按键的专属区-----*/
case 4:
    if (0==ROW_INPUT1&&1==ROW_INPUT2&&1==ROW_INPUT3) //仅判断 S1 按键，避免交叉影响
    {
        Su16KeyCnt++;
        if (Su16KeyCnt>=KEY_ENTER_CONTINUITY_TIME)//该时间是“单击”与“连击”的分界线
        {
            Su16KeyCnt=0;    //计时器清零，为即将来临的计时做准备
            Su8KeyStep=5;    //S1 按键进入有节奏的连续触发
        }
    }
    else //如果期间检查到 S1 按键已经松手
    {
        Su8KeyStep=1;    //返回步骤 1 继续扫描
        Su8KeyLock=0;
        Su16KeyCnt=0;
        Su16KeyContinuityTime=KEY_CONTINUITY_INITIAL_TIME; //动态时间阈值。重装初始值。

        Su8ColumnRecord++;
        if (Su8ColumnRecord>=3)
        {
            Su8ColumnRecord=0;
        }
    }

    break;
case 5: //S1 按键进入有节奏的连续触发
    if (0==ROW_INPUT1&&1==ROW_INPUT2&&1==ROW_INPUT3) //仅判断 S1 按键，避免交叉影响
    {
        Su16KeyCnt++;
        if (Su16KeyCnt>=Su16KeyContinuityTime) //该时间是“刚开始不断减小，最后不变”
        {
            Su16KeyCnt=0;    //清零，为了继续连击。
            vGu8KeySec=1;    //触发一次 S1 按键
            vGu8ShieldVoiceFlag=1; //因为连击，把当前按键触发的声音屏蔽掉
            if (Su16KeyContinuityTime>=KEY_SUB_DT_TIME)
            {
                //Su16KeyContinuityTime 数值不断被减小，按键的触发速度就不断变快
                Su16KeyContinuityTime=Su16KeyContinuityTime-KEY_SUB_DT_TIME;//变快节奏
            }

            //最小间隔时间 KEY_CONTINUITY_MIN_TIME 就是“高速匀速”

```

```

        if(Su16KeyContinuityTime<KEY_CONTINUITY_MIN_TIME)
        {
            //最后以 KEY_CONTINUITY_MIN_TIME 时间为最高速进行“匀速”
            Su16KeyContinuityTime=KEY_CONTINUITY_MIN_TIME;
        }

    }
}
else //如果期间检查到 S1 按键已经松手
{
    Su8KeyStep=1;    //返回步骤 1 继续扫描
    Su8KeyLock=0;
    Su16KeyCnt=0;
    Su16KeyContinuityTime=KEY_CONTINUITY_INITIAL_TIME; //动态时间阈值。重装初始值。

    Su8ColumnRecord++;
    if(Su8ColumnRecord>=3)
    {
        Su8ColumnRecord=0;
    }
}
break;

/*-----S9 按键的专属区-----*/
case 6:
    if(1==ROW_INPUT1&&1==ROW_INPUT2&&0==ROW_INPUT3) //仅判断 S9 按键，避免交叉影响
    {
        Su16KeyCnt++;
        if(Su16KeyCnt>=KEY_ENTER_CONTINUITY_TIME)//该时间是“单击”与“连击”的分界线
        {
            Su16KeyCnt=0;    //计时器清零，为即将来临的计时做准备
            Su8KeyStep=7;    //S9 按键进入有节奏的连续触发
        }
    }
else //如果期间检查到 S9 按键已经松手
{
    Su8KeyStep=1;    //返回步骤 1 继续扫描
    Su8KeyLock=0;
    Su16KeyCnt=0;
    Su16KeyContinuityTime=KEY_CONTINUITY_INITIAL_TIME; //动态时间阈值。重装初始值。

    Su8ColumnRecord++;
    if(Su8ColumnRecord>=3)

```

```

        {
            Su8ColumnRecord=0;
        }
    }

    break;

case 7: //S9 按键进入有节奏的连续触发
    if (1==ROW_INPUT1&&1==ROW_INPUT2&&0==ROW_INPUT3) //仅判断 S9 按键，避免交叉影响
    {
        Su16KeyCnt++;
        if (Su16KeyCnt>=Su16KeyContinuityTime) //该时间是“刚开始不断减小，最后不变”
        {
            Su16KeyCnt=0; //清零，为了继续连击。
            vGu8KeySec=9; //触发一次 S9 按键
            vGu8ShieldVoiceFlag=1; //因为连击，把当前按键触发的声音屏蔽掉
            if (Su16KeyContinuityTime>=KEY_SUB_DT_TIME)
            {
                //Su16KeyContinuityTime 数值不断被减小，按键的触发速度就不断变快
                Su16KeyContinuityTime=Su16KeyContinuityTime-KEY_SUB_DT_TIME;//变快节奏
            }

            //最小间隔时间 KEY_CONTINUITY_MIN_TIME 就是“高速匀速”
            if (Su16KeyContinuityTime<KEY_CONTINUITY_MIN_TIME)
            {
                //最后以 KEY_CONTINUITY_MIN_TIME 时间为最高速进行“匀速”
                Su16KeyContinuityTime=KEY_CONTINUITY_MIN_TIME;
            }
        }
    }

else //如果期间检查到 S9 按键已经松手
{
    Su8KeyStep=1; //返回步骤 1 继续扫描
    Su8KeyLock=0;
    Su16KeyCnt=0;
    Su16KeyContinuityTime=KEY_CONTINUITY_INITIAL_TIME; //动态时间阈值。重装初始值。

    Su8ColumnRecord++;
    if (Su8ColumnRecord>=3)
    {
        Su8ColumnRecord=0;
    }
}

break;

```

```

    }
}

void KeyTask(void)
{
    if(0==vGu8KeySec)
    {
        return;
    }

    switch(vGu8KeySec)
    {
        case 1:    //S1 按键的任务
            if(Gu16SetData>0)
            {
                Gu16SetData--;    // “设置参数”
                Gu8DisplayUpdate=1; //刷新显示
            }

            if(0==vGu8ShieldVoiceFlag) //声音没有被屏蔽
            {
                vGu8BeepTimerFlag=0;
                vGu16BeepTimerCnt=KEY_VOICE_TIME; //发出“嘀”一声
                vGu8BeepTimerFlag=1;
            }

            vGu8ShieldVoiceFlag=0; //及时把屏蔽标志清零，避免平时正常的单击声音也被淹没。
            vGu8KeySec=0; //响应按键服务处理程序后，按键编号必须清零，避免一致触发
            break;

        case 9:    //S9 按键的任务

            if(Gu16SetData<800)
            {
                Gu16SetData++;    // “设置参数”
                Gu8DisplayUpdate=1; //刷新显示
            }

            if(0==vGu8ShieldVoiceFlag) //声音没有被屏蔽
            {
                vGu8BeepTimerFlag=0;
                vGu16BeepTimerCnt=KEY_VOICE_TIME; //发出“嘀”一声
                vGu8BeepTimerFlag=1;
            }
    }
}

```

```

    }

    vGu8ShieldVoiceFlag=0; //及时把屏蔽标志清零，避免平时正常的单击声音也被淹没。
    vGu8KeySec=0; //响应按键服务处理程序后，按键编号必须清零，避免一致触发
    break;

default:

    vGu8KeySec=0;
    break;

}
}

void T0_time() interrupt 1
{
    VoiceScan();
    KeyScan();

    TH0=0xfc;
    TL0=0x66;
}

void SystemInitial(void)
{
    TMOD=0x01;
    TH0=0xfc;
    TL0=0x66;
    EA=1;
    ET0=1;
    TR0=1;
}

void Delay(unsigned long u32DelayTime)
{
    for(;u32DelayTime>0;u32DelayTime--);
}

void PeripheralInitial(void)
{
}

```

```

void BeepOpen(void)
{
    P3_4=0;
}

void BeepClose(void)
{
    P3_4=1;
}

void VoiceScan(void)
{
    static unsigned char Su8Lock=0;

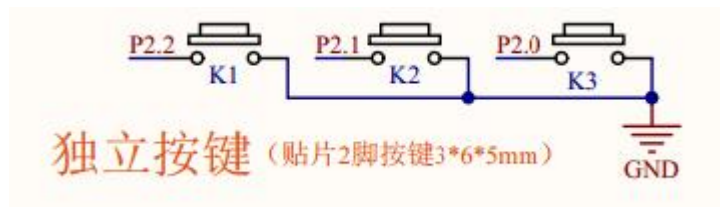
    if(1==vGu8BeepTimerFlag&&vGu16BeepTimerCnt>0)
    {
        if(0==Su8Lock)
        {
            Su8Lock=1;
            BeepOpen();
        }
        else
        {
            vGu16BeepTimerCnt--;

            if(0==vGu16BeepTimerCnt)
            {
                Su8Lock=0;
                BeepClose();
            }
        }
    }
}

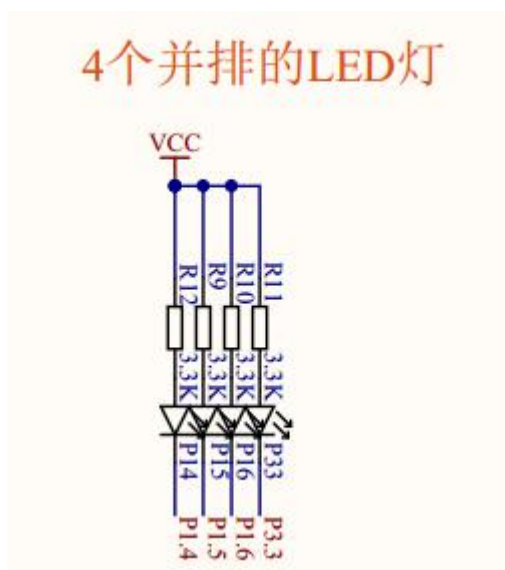
```

## 第一百零七节： 开关感应器的识别与软件滤波。

### 【107.1 开关感应器的识别与软件滤波。】



上图 107.1.1 独立按键模拟开关感应器



上图 107.1.2 LED 电路

什么叫开关感应器？凡是只能输出 0 和 1 这两种状态的感应器都可以统称为开关感应器。前面花了大量的章节讲按键，按键的识别主要是识别电平变化状态的“下降沿”，程序代码中有 1 个特别的变量标志叫“自锁标志”，还有 1 个用来消除抖动的“计时器”。本节讲的开关感应器跟按键很相似，差别在于，开关感应器是识别电平变化状态的“电平”，程序代码中没有“自锁标志”，但是多增加了 1 个用来消除抖动的“计时器”，也就是一共有两个用来消除抖动的“计时器”，这两个“计时器”相互“清零”相互“抗衡”，从而实现了开关感应器的“消抖”处理，专业术语也叫“软件滤波”。消抖的时间跟按键差不多，我的经验值是 20ms 到 30ms 之间，我平时在项目中喜欢用 20ms。

在显示框架方面，除了之前讲过 `Gu8DisplayUpdate` 这类“显示刷新变量”，本节介绍另外一种常用的显示框架，原理是“某数值跟上一次对比，如果发生了变化（两数值不一样），则自动刷新显示，并及时记录当前值”。

本节例程实现的功能如下：用 K1 独立按键模拟开关感应器，K1 独立按键“没有被按下”时是高电平，单片机识别到这种“高电平”，就让 P1.4 所在的 LED 灯发亮；K1 独立按键“被按下”时是低电平，单片机识别到这种“低电平”，就让 P1.4 所在的 LED 灯熄灭。

```
#include "REG52.H"
```

```

#define SENSOR_TIME 20    //开关感应器的“滤波”时间

void T0_time();
void SystemInitial(void) ;
void Delay(unsigned long u32DelayTime) ;
void PeripheralInitial(void) ;

void VoiceScan(void);
void SensorScan(void);
void DisplayTask(void);    //显示的任务函数（LED 显示状态）

sbit P1_4=P1^4;
sbit Sensor_K1_sr=P2^2;    //开关感应器 K1 所在的引脚

volatile unsigned char vGu8Sensor_K1=0;    //K1 开关感应器的当前电平状态。

void main()
{
    SystemInitial();
    Delay(10000);
    PeripheralInitial();
    while(1)
    {
        DisplayTask();    //显示的任务函数（LED 显示状态）
    }
}

/* 注释一：
* 后缀为_Last 这类“对比上次数值发生变化而自动刷新显示”在“显示框架”里是很常见的，
* 目的是，既能及时刷新显示，又能避免主函数“不断去执行显示代码”而影响程序效率。
*/

void DisplayTask(void)    //显示的任务函数（LED 显示状态）
{
    // Su8Sensor_K1_Last 初始化取值 255，只要不为 0 或者 1 就行，目的是让上电就发生第一次刷新。
    static unsigned char Su8Sensor_K1_Last=255;    //记录 K1 开关感应器上一次的电平状态。

    if(Su8Sensor_K1_Last!=vGu8Sensor_K1)    //如果当前值与上一次值不一样，就自动刷新
    {
        Su8Sensor_K1_Last=vGu8Sensor_K1;    //及时记录最新值，避免主函数“不断去执行显示代码”

        if(0==vGu8Sensor_K1)    //如果当前电平状态为“低电平”，LED 熄灭
        {
            P1_4=1;    //LED 熄灭

```



```

    }
    else //如果当前电平状态为“高电平”，LED 发亮
    {
        P1_4=0; //LED 发亮
    }
}
}

```

/\* 注释二：

\* 本节破题的关键：

\* 两个“计时器”相互“清零”相互“抗衡”，从而实现了开关感应器的“消抖”处理，

\* 专业术语也叫“软件滤波”。这种滤波方式，不管是从“高转成低”，还是“低转成高”，

\* 如果在某个瞬间出现干扰抖动，某个计数器都会及时被“清零”，从而起到非常高效的消抖滤波作用。

\*/

void SensorScan(void) //此函数放在定时中断里每 1ms 扫描一次，用来识别和滤波开关感应器

```

{
    static unsigned int Su16Sensor_K1_H_Cnt=0; //判断高电平的计时器
    static unsigned int Su16Sensor_K1_L_Cnt=0; //判断低电平的计时器

    if(0==Sensor_K1_sr)
    {
        Su16Sensor_K1_H_Cnt=0; //在判断低电平的时候，高电平的计时器被清零，巧妙极了！
        Su16Sensor_K1_L_Cnt++;
        if(Su16Sensor_K1_L_Cnt>=SENSOR_TIME)
        {
            Su16Sensor_K1_L_Cnt=0;
            vGu8Sensor_K1=0; //此全局变量反馈当前电平的状态
        }

    }
    else
    {
        Su16Sensor_K1_L_Cnt=0; //在判断高电平的时候，低电平的计时器被清零，巧妙极了！
        Su16Sensor_K1_H_Cnt++;
        if(Su16Sensor_K1_H_Cnt>=SENSOR_TIME)
        {
            Su16Sensor_K1_H_Cnt=0;
            vGu8Sensor_K1=1; //此全局变量反馈当前电平的状态
        }

    }
}

```

void TO\_time() interrupt 1

```
{  
    SensorScan(); //开关感应器的识别与软件滤波处理  
  
    TH0=0xfc;  
    TL0=0x66;  
}
```

```
void SystemInitial(void)
```

```
{  
    TMOD=0x01;  
    TH0=0xfc;  
    TL0=0x66;  
    EA=1;  
    ET0=1;  
    TR0=1;  
}
```

```
void Delay(unsigned long u32DelayTime)
```

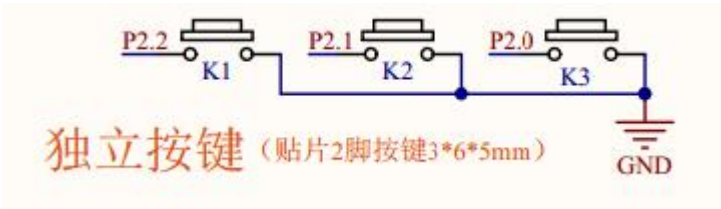
```
{  
    for(;u32DelayTime>0;u32DelayTime--);  
}
```

```
void PeripheralInitial(void)
```

```
{  
  
}
```

第一百零八节： 按键控制跑马灯的启动和暂停和停止。

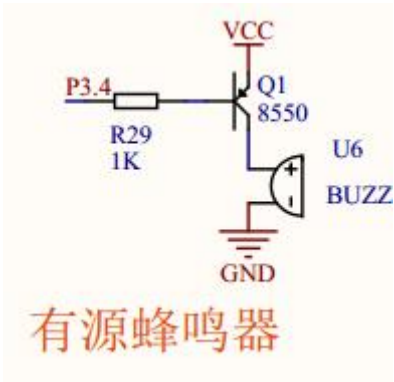
【108.1 按键控制跑马灯的启动和暂停和停止。】



上图 108.1.1 独立按键



上图 108.1.2 LED 电路



上图 108.1.3 有源蜂鸣器的电路

在我眼里，按键不仅仅是按键，跑马灯不仅仅是跑马灯。按键是输入设备，跑马灯是应用程序。本节表面上讲按键控制跑马灯的简单项目，实际上作者用心良苦立意深远，试图通过按键与跑马灯，来分享一种输入设备如何关联应用程序的程序框架。

本节例程实现的功能如下：

(1)【启动暂停】按键 K1。按下【启动暂停】按键 K1 启动之后，跑马灯处于“启动”状态，4 个 LED 灯从左到右依次循环的变亮，给人“跑”起来的感觉。此时如果再按一次【启动暂停】按键 K1，则跑马灯处于“暂停”状态，如果再按一次【启动暂停】按键 K1，跑马灯又变回“启动”状态。因此，【启动暂停】按键 K1 是专门用来切换“启动”和“暂停”这两种状态。

(2)【停止】按键 K2。当跑马灯处于“启动”或者“暂停”或者“停止”的状态时，只要按下【停止】按键 K2，当前的运动状态就终止，强制变回初始的“停止”状态，类似“复位”按键的作用。当跑马灯处于“停止”状态时，此时再按下【启动暂停】按键 K1 之后，跑马灯又处于“启动”状态。

```
#include "REG52.H"

#define KEY_VOICE_TIME    50
#define KEY_FILTER_TIME  25
#define RUN_TIME    200    //跑马灯的跑动速度的时间参数

void TO_time();
void SystemInitial(void) ;
void Delay(unsigned long u32DelayTime) ;
void PeripheralInitial(void) ;

void BeepOpen(void);
void BeepClose(void);
void VoiceScan(void);
void KeyScan(void);
void KeyTask(void);
void RunTask(void);    //跑马灯的任务函数

//4 个跑马灯的输出口
sbit P1_4=P1^4;
sbit P1_5=P1^5;
sbit P1_6=P1^6;
sbit P3_3=P3^3;

//蜂鸣器的输出口
sbit P3_4=P3^4;

sbit KEY_INPUT1=P2^2;    //【启动暂停】按键 K1 的输入口。
sbit KEY_INPUT2=P2^1;    //【停止】按键 K2 的输入口。

volatile unsigned char vGu8BeepTimerFlag=0;
volatile unsigned int vGu16BeepTimerCnt=0;

volatile unsigned char vGu8KeySec=0;

unsigned char Gu8RunStart=0;    //控制跑马灯启动的总开关
```

```
unsigned char Gu8RunStatus=0; //标识跑马灯当前的状态。0 代表停止，1 代表启动，2 代表暂停。
```

```
volatile unsigned char vGu8RunTimerFlag=0; //用于控制跑马灯跑动速度的定时器
```

```
volatile unsigned int vGu16RunTimerCnt=0;
```

```
void main()
```

```
{
    SystemInitial();
    Delay(10000);
    PeripheralInitial();
    while(1)
    {
        KeyTask(); //按键的任务函数
        RunTask(); //跑马灯的任务函数
    }
}
```

```
void TO_time() interrupt 1
```

```
{
    VoiceScan();
    KeyScan();

    if(1==vGu8RunTimerFlag&&vGu16RunTimerCnt>0) //用于控制跑马灯跑动速度的定时器
    {
        vGu16RunTimerCnt--;
    }

    TH0=0xfc;
    TL0=0x66;
}
```

```
void SystemInitial(void)
```

```
{
    TMOD=0x01;
    TH0=0xfc;
    TL0=0x66;
    EA=1;
    ET0=1;
    TR0=1;
}
```

```
void Delay(unsigned long u32DelayTime)
```

```
{
```

```

        for(;u32DelayTime>0;u32DelayTime--);
    }

void PeripheralInitial(void)
{
    //跑马灯处于初始化的状态
    P1_4=0;    //第 1 个灯亮
    P1_5=1;    //第 2 个灯灭
    P1_6=1;    //第 3 个灯灭
    P3_3=1;    //第 4 个灯灭
}

void BeepOpen(void)
{
    P3_4=0;
}

void BeepClose(void)
{
    P3_4=1;
}

void VoiceScan(void)
{
    static unsigned char Su8Lock=0;

    if(1==vGu8BeepTimerFlag&&vGu16BeepTimerCnt>0)
    {
        if(0==Su8Lock)
        {
            Su8Lock=1;
            BeepOpen();
        }
        else
        {

            vGu16BeepTimerCnt--;

            if(0==vGu16BeepTimerCnt)
            {
                Su8Lock=0;
                BeepClose();
            }
        }
    }
}

```

```

    }

    }

}

void KeyScan(void) //此函数放在定时中断里每 1ms 扫描一次
{
    static unsigned char Su8KeyLock1;
    static unsigned int  Su16KeyCnt1;
    static unsigned char Su8KeyLock2;
    static unsigned int  Su16KeyCnt2;

    // 【启动暂停】按键 K1 的扫描识别
    if(0!=KEY_INPUT1)
    {
        Su8KeyLock1=0;
        Su16KeyCnt1=0;
    }
    else if(0==Su8KeyLock1)
    {
        Su16KeyCnt1++;
        if(Su16KeyCnt1>=KEY_FILTER_TIME)
        {
            Su8KeyLock1=1;
            vGu8KeySec=1;    //触发 1 号键
        }
    }

    // 【停止】按键 K2 的扫描识别
    if(0!=KEY_INPUT2)
    {
        Su8KeyLock2=0;
        Su16KeyCnt2=0;
    }
    else if(0==Su8KeyLock2)
    {
        Su16KeyCnt2++;
        if(Su16KeyCnt2>=KEY_FILTER_TIME)
        {
            Su8KeyLock2=1;
            vGu8KeySec=2;    //触发 2 号键
        }
    }
}

```

```

}

/* 注释一：
* 本节破题的关键：
* 在 KeyTask 和 RunTask 两个任务函数之间，主要是靠 Gu8RunStart 和 Gu8RunStatus 这两个
* 全局变量来传递信息。
*/

void KeyTask(void)    //按键的任务函数，放在主函数内
{
    if(0==vGu8KeySec)
    {
        return; //按键的触发序号是 0 意味着无按键触发，直接退出当前函数，不执行此函数下面的代码
    }

    switch(vGu8KeySec) //根据不同的按键触发序号执行对应的代码
    {
        case 1:        //1 号按键。【启动暂停】按键 K1
            if(0==Gu8RunStatus) //当跑马灯处于“停止”状态时
            {
                Gu8RunStart=1;    //总开关“打开”。
                Gu8RunStatus=1;    //状态切换到“启动”状态
            }
            else if(1==Gu8RunStatus) //当跑马灯处于“启动”状态时
            {
                Gu8RunStatus=2;    //状态切换到“暂停”状态
            }
            else //当跑马灯处于“暂停”状态时
            {
                Gu8RunStatus=1;    //状态切换到“启动”状态
            }

            vGu8BeepTimerFlag=0;
            vGu16BeepTimerCnt=KEY_VOICE_TIME; //触发按键后，发出固定长度的声音
            vGu8BeepTimerFlag=1;
            vGu8KeySec=0; //响应按键服务处理程序后，按键编号必须清零，避免一直触发
            break;

        case 2:        //2 号按键。【停止】按键 K2

            Gu8RunStart=0;    //总开关“关闭”。

```



```

        Gu8RunStatus=0; //状态切换到“停止”状态

        vGu8BeepTimerFlag=0;
        vGu16BeepTimerCnt=KEY_VOICE_TIME; //触发按键后，发出固定长度的声音
        vGu8BeepTimerFlag=1;
        vGu8KeySec=0; //响应按键服务处理程序后，按键编号必须清零，避免一直触发
        break;
    }
}

void RunTask(void) //跑马灯的任务函数，放在主函数内
{
    static unsigned char Su8RunStep=0; //运行的步骤

    //当总开关处于“停止”并且“步骤不为0”时，强制把步骤归零, 跑马灯初始化。
    if (0!=Su8RunStep&&0==Gu8RunStart)
    {
        Su8RunStep=0; //步骤归零

        //跑马灯处于初始化的状态
        P1_4=0; //第1个灯亮
        P1_5=1; //第2个灯灭
        P1_6=1; //第3个灯灭
        P3_3=1; //第4个灯灭
    }

    switch(Su8RunStep) //屡见屡爱的 switch 又来了
    {
        case 0:
            if(1==Gu8RunStart) //总开关“打开”
            {
                vGu8RunTimerFlag=0;
                vGu16RunTimerCnt=0; //定时器清零
                Su8RunStep=1; //切换到下一步，启动
            }
            break;
        case 1:
            if(1==Gu8RunStatus&&0==vGu16RunTimerCnt) //当前处于“启动”状态，并且定时器等于0
            {
                P1_4=0; //第1个灯亮
                P1_5=1; //第2个灯灭
            }
        }
    }
}

```

```

        P1_6=1;    //第 3 个灯灭
        P3_3=1;    //第 4 个灯灭

        vGu8RunTimerFlag=0;
        vGu16RunTimerCnt=RUN_TIME;    //用于控制跑马灯跑动速度的定时器
        vGu8RunTimerFlag=1;    //启动定时器
        Su8RunStep=2;    //切换到下一步
    }

    break;
case 2:
    if(1==Gu8RunStatus&&0==vGu16RunTimerCnt) //当前处于“启动”状态，并且定时器等于 0
    {
        P1_4=1;    //第 1 个灯灭
        P1_5=0;    //第 2 个灯亮
        P1_6=1;    //第 3 个灯灭
        P3_3=1;    //第 4 个灯灭

        vGu8RunTimerFlag=0;
        vGu16RunTimerCnt=RUN_TIME;    //用于控制跑马灯跑动速度的定时器
        vGu8RunTimerFlag=1;    //启动定时器
        Su8RunStep=3;    //切换到下一步
    }

    break;
case 3:
    if(1==Gu8RunStatus&&0==vGu16RunTimerCnt) //当前处于“启动”状态，并且定时器等于 0
    {
        P1_4=1;    //第 1 个灯灭
        P1_5=1;    //第 2 个灯灭
        P1_6=0;    //第 3 个灯亮
        P3_3=1;    //第 4 个灯灭

        vGu8RunTimerFlag=0;
        vGu16RunTimerCnt=RUN_TIME;    //用于控制跑马灯跑动速度的定时器
        vGu8RunTimerFlag=1;    //启动定时器
        Su8RunStep=4;    //切换到下一步
    }

    break;
case 4:
    if(1==Gu8RunStatus&&0==vGu16RunTimerCnt) //当前处于“启动”状态，并且定时器等于 0
    {
        P1_4=1;    //第 1 个灯灭

```

```
P1_5=1;    //第 2 个灯灭
P1_6=1;    //第 3 个灯灭
P3_3=0;    //第 4 个灯亮

vGu8RunTimerFlag=0;
vGu16RunTimerCnt=RUN_TIME;    //用于控制跑马灯跑动速度的定时器
vGu8RunTimerFlag=1;    //启动定时器
Su8RunStep=1;    //返回到第 1 步，重新开始下一轮的循环!!!
}

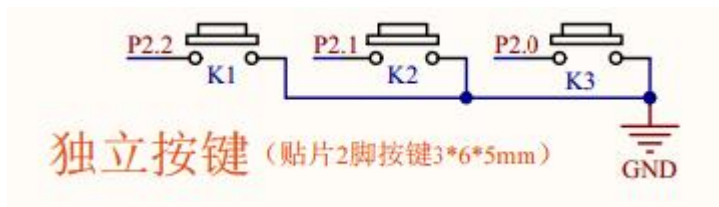
break;

}

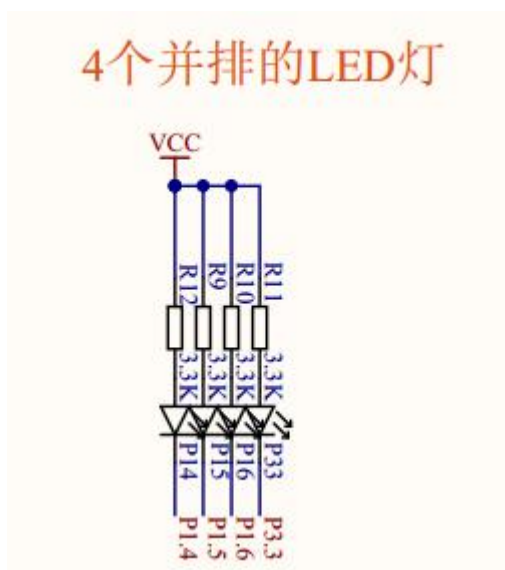
}
```

第一百零九节： 按键控制跑马灯的方向。

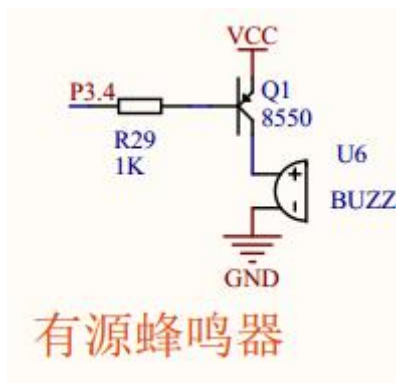
【109.1 按键控制跑马灯的方向。】



上图 109.1.1 独立按键



上图 109.1.2 LED 电路



上图 109.1.3 有源蜂鸣器的电路

之前 108 节讲到跑马灯的启动、暂停、停止，本节在此基础上，增加一个“方向”的控制，除了加深理解输入设备如何关联应用程序的程序框架之外，还有一个知识点值得一提，就是如何通过灵活切换 switch 的“步骤变量”来达到随心所欲的过程控制，本节的“方向”的控制就用到这个方法。

本节例程的功能如下：

(1) **【启动暂停】** 按键 K1。按下 **【启动暂停】** 按键 K1 启动之后，跑马灯处于“启动”状态，4 个 LED 灯挨个依次循环的变亮，给人“跑”起来的感觉。此时如果再按一次 **【启动暂停】** 按键 K1，则跑马灯处于“暂停”状态，如果再按一次 **【启动暂停】** 按键 K1，跑马灯又变回“启动”状态。因此，**【启动暂停】** 按键 K1 是专门用来切换“启动”和“暂停”这两种状态。

(2) **【停止】** 按键 K2。当跑马灯处于“启动”或者“暂停”或者“停止”的状态时，只要按下 **【停止】** 按键 K2，当前的运动状态就终止，强制变回初始的“停止”状态，类似“复位”按键的作用。当跑马灯处于“停止”状态时，此时再按下 **【启动暂停】** 按键 K1 之后，跑马灯又处于“启动”状态。

(3) **【方向】** 按键 K3。跑马灯上电后默认处于“往右跑”的方向。每按一次 **【方向】** 按键 K3，跑马灯就在“往右跑”与“往左跑”两个方向之间切换。

```
#include "REG52.H"

#define KEY_VOICE_TIME    50
#define KEY_FILTER_TIME  25
#define RUN_TIME    200    //跑马灯的跑动速度的时间参数

void T0_time();
void SystemInitial(void) ;
void Delay(unsigned long u32DelayTime) ;
void PeripheralInitial(void) ;

void BeepOpen(void);
void BeepClose(void);
void VoiceScan(void);
void KeyScan(void);
void KeyTask(void);
void RunTask(void);    //跑马灯的任务函数

//4 个跑马灯的输出口
sbit P1_4=P1^4;
sbit P1_5=P1^5;
sbit P1_6=P1^6;
sbit P3_3=P3^3;

//蜂鸣器的输出口
sbit P3_4=P3^4;

sbit KEY_INPUT1=P2^2;    //【启动暂停】按键 K1 的输入口。
sbit KEY_INPUT2=P2^1;    //【停止】按键 K2 的输入口。
sbit KEY_INPUT3=P2^0;    //【方向】按键 K3 的输入口。

volatile unsigned char vGu8BeepTimerFlag=0;
volatile unsigned int vGu16BeepTimerCnt=0;
```

```

volatile unsigned char vGu8KeySec=0;

unsigned char Gu8RunStart=0;      //控制跑马灯启动的总开关
unsigned char Gu8RunStatus=0;     //标识跑马灯当前的状态。0 代表停止，1 代表启动，2 代表暂停。
unsigned char Gu8RunDirection=0;  //标识跑马灯当前的方向。0 代表往右跑，1 代表往左跑。

volatile unsigned char vGu8RunTimerFlag=0;  //用于控制跑马灯跑动速度的定时器
volatile unsigned int vGu16RunTimerCnt=0;

void main()
{
    SystemInitial();
    Delay(10000);
    PeripheralInitial();
    while(1)
    {
        KeyTask();    //按键的任务函数
        RunTask();    //跑马灯的任务函数
    }
}

void T0_time() interrupt 1
{
    VoiceScan();
    KeyScan();

    if(1==vGu8RunTimerFlag&&vGu16RunTimerCnt>0)  //用于控制跑马灯跑动速度的定时器
    {
        vGu16RunTimerCnt--;
    }

    TH0=0xfc;
    TL0=0x66;
}

void SystemInitial(void)
{
    TMOD=0x01;
    TH0=0xfc;
    TL0=0x66;
    EA=1;
    ET0=1;
    TR0=1;
}

```

```

}

void Delay(unsigned long u32DelayTime)
{
    for(;u32DelayTime>0;u32DelayTime--);
}

void PeripheralInitial(void)
{
    //跑马灯处于初始化的状态
    P1_4=0;    //第 1 个灯亮
    P1_5=1;    //第 2 个灯灭
    P1_6=1;    //第 3 个灯灭
    P3_3=1;    //第 4 个灯灭
}

void BeepOpen(void)
{
    P3_4=0;
}

void BeepClose(void)
{
    P3_4=1;
}

void VoiceScan(void)
{
    static unsigned char Su8Lock=0;

    if(1==vGu8BeepTimerFlag&&vGu16BeepTimerCnt>0)
    {
        if(0==Su8Lock)
        {
            Su8Lock=1;
            BeepOpen();
        }
        else
        {

            vGu16BeepTimerCnt--;
        }
    }
}

```

```

        if(0==vGu16BeepTimerCnt)
        {
            Su8Lock=0;
            BeepClose();
        }
    }
}

void KeyScan(void)  //此函数放在定时中断里每 1ms 扫描一次
{
    static unsigned char Su8KeyLock1;
    static unsigned int  Su16KeyCnt1;
    static unsigned char Su8KeyLock2;
    static unsigned int  Su16KeyCnt2;
    static unsigned char Su8KeyLock3;
    static unsigned int  Su16KeyCnt3;

    // 【启动暂停】按键 K1 的扫描识别
    if(0!=KEY_INPUT1)
    {
        Su8KeyLock1=0;
        Su16KeyCnt1=0;
    }
    else if(0==Su8KeyLock1)
    {
        Su16KeyCnt1++;
        if(Su16KeyCnt1>=KEY_FILTER_TIME)
        {
            Su8KeyLock1=1;
            vGu8KeySec=1;    //触发 1 号键
        }
    }

    // 【停止】按键 K2 的扫描识别
    if(0!=KEY_INPUT2)
    {
        Su8KeyLock2=0;
        Su16KeyCnt2=0;
    }
    else if(0==Su8KeyLock2)
    {

```



```

        Su16KeyCnt2++;
        if (Su16KeyCnt2>=KEY_FILTER_TIME)
        {
            Su8KeyLock2=1;
            vGu8KeySec=2;    //触发 2 号键
        }
    }

    // 【方向】 按键 K3 的扫描识别
    if (0!=KEY_INPUT3)
    {
        Su8KeyLock3=0;
        Su16KeyCnt3=0;
    }
    else if (0==Su8KeyLock3)
    {
        Su16KeyCnt3++;
        if (Su16KeyCnt3>=KEY_FILTER_TIME)
        {
            Su8KeyLock3=1;
            vGu8KeySec=3;    //触发 3 号键
        }
    }
}

/* 注释一：
*   本节破题的关键：
*   在 KeyTask 和 RunTask 两个任务函数之间，主要是靠 Gu8RunStart、Gu8RunStatus、Gu8RunDirection
*   这三个全局变量来传递信息。
*/

void KeyTask(void)    //按键的任务函数，放在主函数内
{
    if (0==vGu8KeySec)
    {
        return; //按键的触发序号是 0 意味着无按键触发，直接退出当前函数，不执行此函数下面的代码
    }

    switch (vGu8KeySec) //根据不同的按键触发序号执行对应的代码
    {
        case 1:        //1 号按键。【启动暂停】 按键 K1
            if (0==Gu8RunStatus) //当跑马灯处于“停止”状态时
            {
                Gu8RunStart=1;    //总开关“打开”。
            }
        }
    }

```

```

        Gu8RunStatus=1; //状态切换到“启动”状态
    }
    else if(1==Gu8RunStatus) //当跑马灯处于“启动”状态时
    {
        Gu8RunStatus=2; //状态切换到“暂停”状态
    }
    else //当跑马灯处于“暂停”状态时
    {
        Gu8RunStatus=1; //状态切换到“启动”状态
    }

    vGu8BeepTimerFlag=0;
    vGu16BeepTimerCnt=KEY_VOICE_TIME; //触发按键后，发出固定长度的声音
    vGu8BeepTimerFlag=1;
    vGu8KeySec=0; //响应按键服务处理程序后，按键编号必须清零，避免一直触发
    break;

case 2: //2号按键。【停止】按键 K2

    Gu8RunStart=0; //总开关“关闭”。
    Gu8RunStatus=0; //状态切换到“停止”状态

    vGu8BeepTimerFlag=0;
    vGu16BeepTimerCnt=KEY_VOICE_TIME; //触发按键后，发出固定长度的声音
    vGu8BeepTimerFlag=1;
    vGu8KeySec=0; //响应按键服务处理程序后，按键编号必须清零，避免一直触发
    break;

case 3: //3号按键。【方向】按键 K3
    //每按一次 K3 按键，Gu8RunDirection 就在 0 和 1 之间切换,从而控制方向
    if(0==Gu8RunDirection)
    {
        Gu8RunDirection=1;
    }
    else
    {
        Gu8RunDirection=0;
    }

    vGu8BeepTimerFlag=0;
    vGu16BeepTimerCnt=KEY_VOICE_TIME; //触发按键后，发出固定长度的声音
    vGu8BeepTimerFlag=1;
    vGu8KeySec=0; //响应按键服务处理程序后，按键编号必须清零，避免一直触发

```

```

        break;

    }
}

/* 注释二：
* “方向”的控制，是通过 Gu8RunDirection 的判断，来灵活切换 switch 的“步骤变量”来达到
* 随心所欲的过程控制。
*/

void RunTask(void)    //跑马灯的任务函数，放在主函数内
{
    static unsigned char Su8RunStep=0; //运行的步骤

    //当总开关处于“停止”并且“步骤不为0”时，强制把步骤归零,跑马灯初始化。
    if (0!=Su8RunStep&&0==Gu8RunStart)
    {
        Su8RunStep=0; //步骤归零

        //跑马灯处于初始化的状态
        P1_4=0;    //第1个灯亮
        P1_5=1;    //第2个灯灭
        P1_6=1;    //第3个灯灭
        P3_3=1;    //第4个灯灭

    }

    switch(Su8RunStep) //屡见屡爱的 switch 又来了
    {
        case 0:
            if(1==Gu8RunStart) //总开关“打开”
            {
                vGu8RunTimerFlag=0;
                vGu16RunTimerCnt=0; //定时器清零
                Su8RunStep=1; //切换到下一步，启动
            }
            break;
        case 1:
            if(1==Gu8RunStatus&&0==vGu16RunTimerCnt) //当前处于“启动”状态，并且定时器等于0
            {
                P1_4=0;    //第1个灯亮
                P1_5=1;    //第2个灯灭
                P1_6=1;    //第3个灯灭
            }
        }
    }
}

```

```

        P3_3=1;    //第 4 个灯灭

        vGu8RunTimerFlag=0;
        vGu16RunTimerCnt=RUN_TIME;    //用于控制跑马灯跑动速度的定时器
        vGu8RunTimerFlag=1;    //启动定时器

        //灵活切换“步骤变量”
        if(0==Gu8RunDirection) //往右跑
        {
            Su8RunStep=2;
        }
        else //往左跑
        {
            Su8RunStep=4;
        }
    }

    break;
case 2:
    if(1==Gu8RunStatus&&0==vGu16RunTimerCnt) //当前处于“启动”状态，并且定时器等于 0
    {
        P1_4=1;    //第 1 个灯灭
        P1_5=0;    //第 2 个灯亮
        P1_6=1;    //第 3 个灯灭
        P3_3=1;    //第 4 个灯灭

        vGu8RunTimerFlag=0;
        vGu16RunTimerCnt=RUN_TIME;    //用于控制跑马灯跑动速度的定时器
        vGu8RunTimerFlag=1;    //启动定时器

        //灵活切换“步骤变量”
        if(0==Gu8RunDirection) //往右跑
        {
            Su8RunStep=3;
        }
        else //往左跑
        {
            Su8RunStep=1;
        }
    }

    break;
case 3:

```

```

if(1==Gu8RunStatus&&0==vGu16RunTimerCnt) //当前处于“启动”状态，并且定时器等于 0
{
    P1_4=1;    //第 1 个灯灭
    P1_5=1;    //第 2 个灯灭
    P1_6=0;    //第 3 个灯亮
    P3_3=1;    //第 4 个灯灭

    vGu8RunTimerFlag=0;
    vGu16RunTimerCnt=RUN_TIME;    //用于控制跑马灯跑动速度的定时器
    vGu8RunTimerFlag=1;    //启动定时器

    //灵活切换“步骤变量”
    if(0==Gu8RunDirection) //往右跑
    {
        Su8RunStep=4;
    }
    else //往左跑
    {
        Su8RunStep=2;
    }
}

break;

case 4:
if(1==Gu8RunStatus&&0==vGu16RunTimerCnt) //当前处于“启动”状态，并且定时器等于 0
{
    P1_4=1;    //第 1 个灯灭
    P1_5=1;    //第 2 个灯灭
    P1_6=1;    //第 3 个灯灭
    P3_3=0;    //第 4 个灯亮

    vGu8RunTimerFlag=0;
    vGu16RunTimerCnt=RUN_TIME;    //用于控制跑马灯跑动速度的定时器
    vGu8RunTimerFlag=1;    //启动定时器

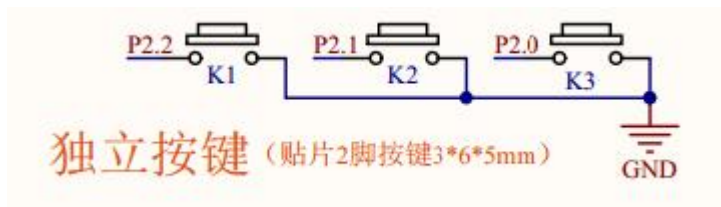
    //灵活切换“步骤变量”
    if(0==Gu8RunDirection) //往右跑
    {
        Su8RunStep=1;
    }
    else //往左跑
    {
        Su8RunStep=3;
    }
}

```

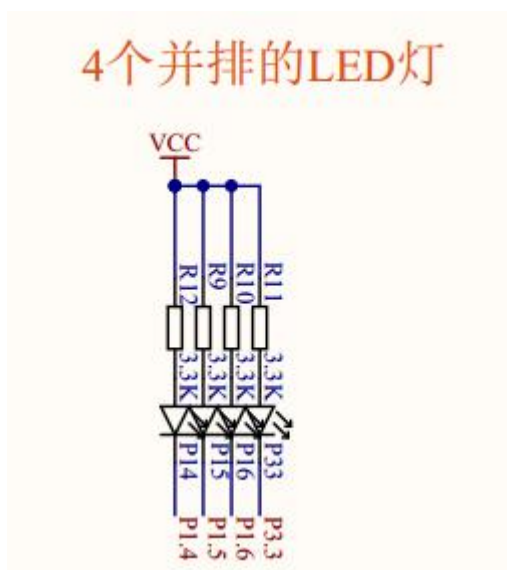
```
    }  
  
    break;  
  
}  
  
}
```

## 第一百一十节： 按键控制跑马灯的速度。

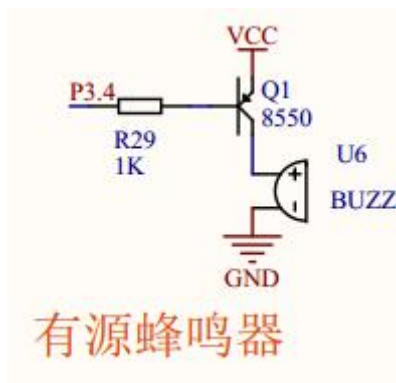
### 【110.1 按键控制跑马灯的速度。】



上图 110.1.1 独立按键



上图 110.1.2 LED 电路



上图 110.1.3 有源蜂鸣器的电路

之前 109 节讲到跑马灯的启动、暂停、停止、方向，本节在此基础上，把原来的“停止”更改为“速度”，加深理解输入设备如何关联应用程序的程序框架。

本节例程的功能如下：

- (1) 【启动暂停】按键 K1。上电后，按下【启动暂停】按键 K1 启动之后，跑马灯处于“启动”状态，4

个LED灯挨个依次循环的变亮，给人“跑”起来的感觉。此时如果再按一次【启动暂停】按键 K1，则跑马灯处于“暂停”状态，如果再按一次【启动暂停】按键 K1，跑马灯又变回“启动”状态。因此，【启动暂停】按键 K1 是专门用来切换“启动”和“暂停”这两种状态。

(2)【速度】按键 K2。每按一次【速度】按键 K2，跑马灯就在“慢”、“中”、“快”三档速度之间切换。

(3)【方向】按键 K3。跑马灯上电后默认处于“往右跑”的方向。每按一次【方向】按键 K3，跑马灯就在“往右跑”与“往左跑”两个方向之间切换。

```
#include "REG52.H"

#define KEY_VOICE_TIME    50
#define KEY_FILTER_TIME  25

#define RUN_TIME_SLOW     500    // “慢”档速度的时间参数
#define RUN_TIME_MIDDLE   300    // “中”档速度的时间参数
#define RUN_TIME_FAST     100    // “快”档速度的时间参数

void T0_time();
void SystemInitial(void) ;
void Delay(unsigned long u32DelayTime) ;
void PeripheralInitial(void) ;

void BeepOpen(void);
void BeepClose(void);
void VoiceScan(void);
void KeyScan(void);
void KeyTask(void);
void RunTask(void);    //跑马灯的任务函数

//4个跑马灯的输出口
sbit P1_4=P1^4;
sbit P1_5=P1^5;
sbit P1_6=P1^6;
sbit P3_3=P3^3;

//蜂鸣器的输出口
sbit P3_4=P3^4;

sbit KEY_INPUT1=P2^2;    //【启动暂停】按键 K1 的输入口。
sbit KEY_INPUT2=P2^1;    //【速度】按键 K2 的输入口。
sbit KEY_INPUT3=P2^0;    //【方向】按键 K3 的输入口。

volatile unsigned char vGu8BeepTimerFlag=0;
volatile unsigned int vGu16BeepTimerCnt=0;
```



```

volatile unsigned char vGu8KeySec=0;

unsigned char Gu8RunStart=0;      //控制跑马灯启动的总开关
unsigned char Gu8RunStatus=0;     //标识跑马灯当前的状态。0 代表停止，1 代表启动，2 代表暂停。
unsigned char Gu8RunDirection=0;  //标识跑马灯当前的方向。0 代表往右跑，1 代表往左跑。
unsigned char Gu8RunSpeed=0;      //当前的速度档位。0 代表“慢”，1 代表“中”，2 代表“快”。
unsigned int  Gu16RunSpeedTimeDate=0; //承接各速度档位的时间参数的变量

volatile unsigned char vGu8RunTimerFlag=0;  //用于控制跑马灯跑动速度的定时器
volatile unsigned int vGu16RunTimerCnt=0;

void main()
{
    SystemInitial();
    Delay(10000);
    PeripheralInitial();
    while(1)
    {
        KeyTask();    //按键的任务函数
        RunTask();    //跑马灯的任务函数
    }
}

void T0_time() interrupt 1
{
    VoiceScan();
    KeyScan();

    if(1==vGu8RunTimerFlag&&vGu16RunTimerCnt>0) //用于控制跑马灯跑动速度的定时器
    {
        vGu16RunTimerCnt--;
    }

    TH0=0xfc;
    TL0=0x66;
}

void SystemInitial(void)
{
    TMOD=0x01;
    TH0=0xfc;
    TL0=0x66;
    EA=1;
}

```

```

    ET0=1;
    TR0=1;
}

void Delay(unsigned long u32DelayTime)
{
    for(;u32DelayTime>0;u32DelayTime--);
}

void PeripheralInitial(void)
{
    //跑马灯处于初始化的状态
    P1_4=0;    //第 1 个灯亮
    P1_5=1;    //第 2 个灯灭
    P1_6=1;    //第 3 个灯灭
    P3_3=1;    //第 4 个灯灭

    //根据当前的速度档位 Gu8RunSpeed，来初始化速度时间参数 Gu16RunSpeedTimeDate
    if(0==Gu8RunSpeed)
    {
        Gu16RunSpeedTimeDate=RUN_TIME_SLOW; //赋值“慢”档的时间参数
    }
    else if(1==Gu8RunSpeed)
    {
        Gu16RunSpeedTimeDate=RUN_TIME_MIDDLE; //赋值“中”档的时间参数
    }
    else
    {
        Gu16RunSpeedTimeDate=RUN_TIME_FAST; //赋值“快”档的时间参数
    }
}

void BeepOpen(void)
{
    P3_4=0;
}

void BeepClose(void)
{
    P3_4=1;
}

```

```

void VoiceScan(void)
{

    static unsigned char Su8Lock=0;

    if(1==vGu8BeepTimerFlag&&vGu16BeepTimerCnt>0)
    {
        if(0==Su8Lock)
        {
            Su8Lock=1;
            BeepOpen();
        }
        else
        {

            vGu16BeepTimerCnt--;

            if(0==vGu16BeepTimerCnt)
            {
                Su8Lock=0;
                BeepClose();
            }

        }
    }
}

```

```

void KeyScan(void)  //此函数放在定时中断里每 1ms 扫描一次
{
    static unsigned char Su8KeyLock1;
    static unsigned int  Su16KeyCnt1;
    static unsigned char Su8KeyLock2;
    static unsigned int  Su16KeyCnt2;
    static unsigned char Su8KeyLock3;
    static unsigned int  Su16KeyCnt3;

    // 【启动暂停】按键 K1 的扫描识别
    if(0!=KEY_INPUT1)
    {
        Su8KeyLock1=0;
        Su16KeyCnt1=0;
    }
    else if(0==Su8KeyLock1)

```

```

{
    Su16KeyCnt1++;
    if (Su16KeyCnt1>=KEY_FILTER_TIME)
    {
        Su8KeyLock1=1;
        vGu8KeySec=1;    //触发 1 号键
    }
}

// 【速度】 按键 K2 的扫描识别
if (0!=KEY_INPUT2)
{
    Su8KeyLock2=0;
    Su16KeyCnt2=0;
}
else if (0==Su8KeyLock2)
{
    Su16KeyCnt2++;
    if (Su16KeyCnt2>=KEY_FILTER_TIME)
    {
        Su8KeyLock2=1;
        vGu8KeySec=2;    //触发 2 号键
    }
}

// 【方向】 按键 K3 的扫描识别
if (0!=KEY_INPUT3)
{
    Su8KeyLock3=0;
    Su16KeyCnt3=0;
}
else if (0==Su8KeyLock3)
{
    Su16KeyCnt3++;
    if (Su16KeyCnt3>=KEY_FILTER_TIME)
    {
        Su8KeyLock3=1;
        vGu8KeySec=3;    //触发 3 号键
    }
}
}

/* 注释一：
* 本节破题的关键：

```

```

* 在 KeyTask 和 RunTask 两个任务函数之间，主要是靠 Gu8RunStart、Gu8RunStatus、Gu8RunDirection、
* Gu16RunSpeedTimeDate 这四个全局变量来传递信息。
*/

void KeyTask(void)    //按键的任务函数，放在主函数内
{
    if(0==vGu8KeySec)
    {
        return; //按键的触发序号是 0 意味着无按键触发，直接退出当前函数，不执行此函数下面的代码
    }

    switch(vGu8KeySec) //根据不同的按键触发序号执行对应的代码
    {
        case 1:        //1 号按键。【启动暂停】按键 K1
            if(0==Gu8RunStatus) //当跑马灯处于“停止”状态时
            {
                Gu8RunStart=1;    //总开关“打开”。
                Gu8RunStatus=1;    //状态切换到“启动”状态
            }
            else if(1==Gu8RunStatus) //当跑马灯处于“启动”状态时
            {
                Gu8RunStatus=2;    //状态切换到“暂停”状态
            }
            else //当跑马灯处于“暂停”状态时
            {
                Gu8RunStatus=1;    //状态切换到“启动”状态
            }

            vGu8BeepTimerFlag=0;
            vGu16BeepTimerCnt=KEY_VOICE_TIME; //触发按键后，发出固定长度的声音
            vGu8BeepTimerFlag=1;
            vGu8KeySec=0; //响应按键服务处理程序后，按键编号必须清零，避免一直触发
            break;

        case 2:        //2 号按键。【速度】按键 K2

            //每按一次 K2 按键，Gu8RunSpeed 就在 0、1、2 三者之间切换，并且根据 Gu8RunSpeed 的数值，
            //对 Gu16RunSpeedTimeDate 赋值不同的速度时间参数，从而控制速度档位
            if(0==Gu8RunSpeed)
            {
                Gu8RunSpeed=1;    //“中”档
                Gu16RunSpeedTimeDate=RUN_TIME_MIDDLE; //赋值“中”档的时间参数
            }
            else if(1==Gu8RunSpeed)

```

```

    {
        Gu8RunSpeed=2;    // “快” 档
        Gu16RunSpeedTimeDate=RUN_TIME_FAST; //赋值 “快” 档的时间参数
    }
    else
    {
        Gu8RunSpeed=0;    // “慢” 档
        Gu16RunSpeedTimeDate=RUN_TIME_SLOW; //赋值 “慢” 档的时间参数
    }

    vGu8BeepTimerFlag=0;
    vGu16BeepTimerCnt=KEY_VOICE_TIME; //触发按键后，发出固定长度的声音
    vGu8BeepTimerFlag=1;
    vGu8KeySec=0; //响应按键服务处理程序后，按键编号必须清零，避免一直触发
    break;

case 3:    //3 号按键。【方向】按键 K3
    //每按一次 K3 按键，Gu8RunDirection 就在 0 和 1 之间切换, 从而控制方向
    if(0==Gu8RunDirection)
    {
        Gu8RunDirection=1;
    }
    else
    {
        Gu8RunDirection=0;
    }

    vGu8BeepTimerFlag=0;
    vGu16BeepTimerCnt=KEY_VOICE_TIME; //触发按键后，发出固定长度的声音
    vGu8BeepTimerFlag=1;
    vGu8KeySec=0; //响应按键服务处理程序后，按键编号必须清零，避免一直触发
    break;

}
}

/* 注释二：
* “速度” 是受 Gu16RunSpeedTimeDate 具体数值大小的影响
*/

void RunTask(void)    //跑马灯的任务函数，放在主函数内
{
    static unsigned char Su8RunStep=0; //运行的步骤

```

```

//当总开关处于“停止”并且“步骤不为0”时，强制把步骤归零,跑马灯初始化。
if(0!=Su8RunStep&&0==Gu8RunStart)
{
    Su8RunStep=0; //步骤归零

    //跑马灯处于初始化的状态
    P1_4=0;    //第1个灯亮
    P1_5=1;    //第2个灯灭
    P1_6=1;    //第3个灯灭
    P3_3=1;    //第4个灯灭

}

switch(Su8RunStep) //屡见屡爱的 switch 又来了
{
    case 0:
        if(1==Gu8RunStart) //总开关“打开”
        {
            vGu8RunTimerFlag=0;
            vGu16RunTimerCnt=0; //定时器清零
            Su8RunStep=1; //切换到下一步，启动
        }
        break;
    case 1:
        if(1==Gu8RunStatus&&0==vGu16RunTimerCnt) //当前处于“启动”状态，并且定时器等于0
        {
            P1_4=0;    //第1个灯亮
            P1_5=1;    //第2个灯灭
            P1_6=1;    //第3个灯灭
            P3_3=1;    //第4个灯灭

            vGu8RunTimerFlag=0;
            vGu16RunTimerCnt=Gu16RunSpeedTimeDate; //速度时间参数变量的大小，决定了速度
            vGu8RunTimerFlag=1;    //启动定时器

            //灵活切换“步骤变量”
            if(0==Gu8RunDirection) //往右跑
            {
                Su8RunStep=2;
            }
            else //往左跑
            {
                Su8RunStep=4;
            }
        }
    }
}

```

```

    }

}

break;
case 2:
    if (1==Gu8RunStatus&&0==vGu16RunTimerCnt) //当前处于“启动”状态，并且定时器等于 0
    {
        P1_4=1;    //第 1 个灯灭
        P1_5=0;    //第 2 个灯亮
        P1_6=1;    //第 3 个灯灭
        P3_3=1;    //第 4 个灯灭

        vGu8RunTimerFlag=0;
        vGu16RunTimerCnt=Gu16RunSpeedTimeDate; //速度时间参数变量的大小，决定了速度
        vGu8RunTimerFlag=1;    //启动定时器

        //灵活切换“步骤变量”
        if (0==Gu8RunDirection) //往右跑
        {
            Su8RunStep=3;
        }
        else //往左跑
        {
            Su8RunStep=1;
        }
    }

    break;
case 3:
    if (1==Gu8RunStatus&&0==vGu16RunTimerCnt) //当前处于“启动”状态，并且定时器等于 0
    {
        P1_4=1;    //第 1 个灯灭
        P1_5=1;    //第 2 个灯灭
        P1_6=0;    //第 3 个灯亮
        P3_3=1;    //第 4 个灯灭

        vGu8RunTimerFlag=0;
        vGu16RunTimerCnt=Gu16RunSpeedTimeDate; //速度时间参数变量的大小，决定了速度
        vGu8RunTimerFlag=1;    //启动定时器

        //灵活切换“步骤变量”
        if (0==Gu8RunDirection) //往右跑
        {

```



```

        Su8RunStep=4;
    }
    else //往左跑
    {
        Su8RunStep=2;
    }
}

break;
case 4:
    if(1==Gu8RunStatus&&0==vGu16RunTimerCnt) //当前处于“启动”状态，并且定时器等于0
    {
        P1_4=1; //第1个灯灭
        P1_5=1; //第2个灯灭
        P1_6=1; //第3个灯灭
        P3_3=0; //第4个灯亮

        vGu8RunTimerFlag=0;
        vGu16RunTimerCnt=Gu16RunSpeedTimeDate; //速度时间参数变量的大小，决定了速度
        vGu8RunTimerFlag=1; //启动定时器

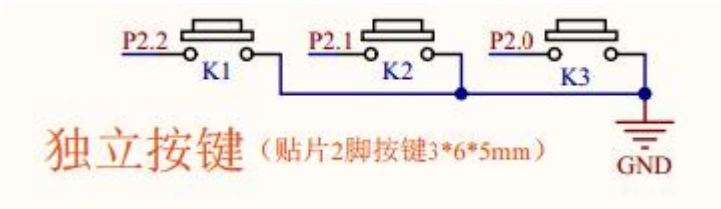
        //灵活切换“步骤变量”
        if(0==Gu8RunDirection) //往右跑
        {
            Su8RunStep=1;
        }
        else //往左跑
        {
            Su8RunStep=3;
        }
    }

    break;
}
}
}

```

第一百一十一节： 工业自动化设备的开关信号的运动控制。

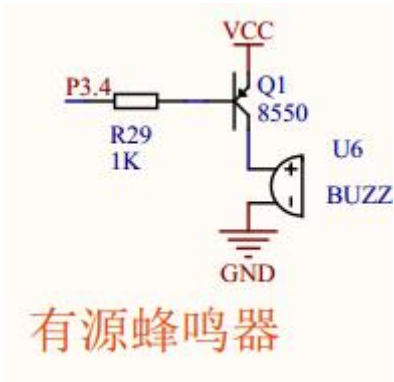
【111.1 开关信号的运动控制。】



上图 111.1.1 独立按键



上图 111.1.2 LED 电路



上图 111.1.3 有源蜂鸣器的电路

本节涉及的知识点有，switch 的过程控制，时间延时，开关感应器的软件滤波，工件计数器，以及整体的软件框架。

现在有一台设备，水平方向有一个滑块，能左右移动，滑块上安装了一个能垂直伸缩的“机械手”。按下启动按键后，滑块先从左边往右边移动，移到最右边碰到“右感应器”后，滑块上的“机械手”开始往下

移动 2 秒，移动 2 秒后开始原路返回，“机械手”向上移动，碰到“上感应器”后，滑块开始往左边移动，移动 3 秒后默认已经回到原位最左边，此时“计数器”累加 1，完成一次过程，如果再按下启动按键，继续重复这个过程。

这个设备用了 2 个气缸。1 个“水平气缸”驱动滑块水平方向的左右移动，当控制“水平气缸”的输出信号为 0 时往左边跑，当控制“水平气缸”的输出信号为 1 时往右边跑。另 1 个“垂直气缸”驱动“机械手”的上下移动，当控制“垂直气缸”的输出信号为 0 时往上边跑，当控制“垂直气缸”的输出信号为 1 时往下边跑。

这个设备用了 2 个开关感应器。分别是“右感应器”和“上感应器”。当感应器没有被碰到的时候信号为 1，当感应器被碰到的时候信号为 0。

这个设备用了 1 个独立按键。控制运动的启动。

2 个气缸是输出信号，用 P1.4 和 P1.5 所控制的两个 LED 模拟。2 个开关感应器是输入信号，用 K2 和 K3 这两个独立按键模拟。1 个独立按键用 K1 按键。如上图。

```
#include "REG52.H"

#define KEY_VOICE_TIME    50
#define KEY_FILTER_TIME   25
#define SENSOR_TIME       20      //开关感应器的“滤波”时间

void TO_time();
void SystemInitial(void) ;
void Delay(unsigned long u32DelayTime) ;
void PeripheralInitial(void) ;

void BeepOpen(void);
void BeepClose(void);

void GoLeft(void) ; //“水平气缸”往左跑
void GoRight(void); //“水平气缸”往右跑
void GoUp(void);    //“垂直气缸”往上跑
void GoDown(void);  //“垂直气缸”往下跑

void VoiceScan(void);
void SensorScan(void); //开关感应器的消抖，在定时中断里调用处理
void KeyScan(void);
void KeyTask(void);
void RunTask(void);    //运动控制的任务函数

sbit P1_4=P1^4; //水平气缸的输出
sbit P1_5=P1^5; //垂直气缸的输出

sbit P3_4=P3^4; //蜂鸣器的输出口
```

```

sbit KEY_INPUT1=P2^2; //【启动】按键 K1 的输入口。

sbit SensorRight_sr=P2^1; //右感应器的输入口
sbit SensorUp_sr=P2^0; //上感应器的输入口

volatile unsigned char vGu8SensorRight=0; //右感应器经过滤波后的当前电平状态。
volatile unsigned char vGu8SensorUp=0; //上感应器经过滤波后的当前电平状态。

volatile unsigned char vGu8BeepTimerFlag=0;
volatile unsigned int vGu16BeepTimerCnt=0;

volatile unsigned char vGu8KeySec=0;

unsigned char Gu8RunStart=0; //启动的总开关
unsigned char Gu8RunStatus=0; //运动的状态，0 为停止，1 为运行

unsigned int Gul6RunCnt=0; //计数器
unsigned int Gul6ReturnLeftTime=3000; //水平往左跑的延时变量，默认为 3 秒
unsigned int Gul6GoDownTime=2000; //垂直往下跑的延时变量，默认为 2 秒

volatile unsigned char vGu8RunTimerFlag=0; //用于控制运动过程中的延时的定时器
volatile unsigned int vGu16RunTimerCnt=0;

void main()
{
    SystemInitial();
    Delay(10000);
    PeripheralInitial();
    while(1)
    {
        KeyTask(); //按键的任务函数
        RunTask(); //运动控制的任务函数
    }
}

/* 注释一：
* 两个“计时器”相互“清零”相互“抗衡”，从而实现了开关感应器的“消抖”处理，
* 专业术语也叫“软件滤波”。这种滤波方式，不管是从“高转成低”，还是“低转成高”，
* 如果在某个瞬间出现干扰抖动，某个计数器都会及时被“清零”，从而起到非常高效的消抖滤波作用。
*/

void SensorScan(void) //此函数放在定时中断里每 1ms 扫描一次，用来识别和滤波开关感应器
{
    static unsigned int Sul6SensorRight_H_Cnt=0; //判断高电平的计时器

```

```

static unsigned int Sul6SensorRight_L_Cnt=0; //判断低电平の计时器

static unsigned int Sul6SensorUp_H_Cnt=0; //判断高电平の计时器
static unsigned int Sul6SensorUp_L_Cnt=0; //判断低电平の计时器

//右感应器の滤波
if(0==SensorRight_sr)
{
    Sul6SensorRight_H_Cnt=0; //在判断低电平的时候，高电平の计时器被清零，巧妙极了！
    Sul6SensorRight_L_Cnt++;
    if(Sul6SensorRight_L_Cnt>=SENSOR_TIME)
    {
        Sul6SensorRight_L_Cnt=0;
        vGu8SensorRight=0; //此全局变量反馈经过滤波后“右感应器”当前电平の状态
    }
}
else
{
    Sul6SensorRight_L_Cnt=0; //在判断高电平的时候，低电平の计时器被清零，巧妙极了！
    Sul6SensorRight_H_Cnt++;
    if(Sul6SensorRight_H_Cnt>=SENSOR_TIME)
    {
        Sul6SensorRight_H_Cnt=0;
        vGu8SensorRight=1; //此全局变量反馈经过滤波后“右感应器”当前电平の状态
    }
}

//上感应器の滤波
if(0==SensorUp_sr)
{
    Sul6SensorUp_H_Cnt=0;
    Sul6SensorUp_L_Cnt++;
    if(Sul6SensorUp_L_Cnt>=SENSOR_TIME)
    {
        Sul6SensorUp_L_Cnt=0;
        vGu8SensorUp=0; //此全局变量反馈经过滤波后“上感应器”当前电平の状态
    }
}
else
{
    Sul6SensorUp_L_Cnt=0;
    Sul6SensorUp_H_Cnt++;
}

```

```

        if (Su16SensorUp_H_Cnt>=SENSOR_TIME)
        {
            Su16SensorUp_H_Cnt=0;
            vGu8SensorUp=1;  //此全局变量反馈经过滤波后“上感应器”当前电平的状态
        }
    }

}

void T0_time() interrupt 1
{
    VoiceScan();
    KeyScan();
    SensorScan();  //用来识别和滤波开关感应器

    if (1==vGu8RunTimerFlag&&vGu16RunTimerCnt>0)  //用于控制运动延时的定时器
    {
        vGu16RunTimerCnt--;
    }

    TH0=0xfc;
    TL0=0x66;
}

void SystemInitial(void)
{
    TMOD=0x01;
    TH0=0xfc;
    TL0=0x66;
    EA=1;
    ET0=1;
    TR0=1;
}

void Delay(unsigned long u32DelayTime)
{
    for(;u32DelayTime>0;u32DelayTime--);
}

void PeripheralInitial(void)
{
    //上电初始化气缸的开机位置

```

```

    GoLeft() ;    // “水平气缸” 往左跑，上电初始化时滑块处于左边
    GoUp() ;      // “垂直气缸” 往上跑，上电初始化时 “机械臂” 处于上方
}

void BeepOpen(void)
{
    P3_4=0;
}

void BeepClose(void)
{
    P3_4=1;
}

void GoLeft(void) // “水平气缸” 往左跑
{
    P1_4=0;
}

void GoRight(void) // “水平气缸” 往右跑
{
    P1_4=1;
}

void GoUp(void) // “垂直气缸” 往上跑
{
    P1_5=0;
}

void GoDown(void) // “垂直气缸” 往下跑
{
    P1_5=1;
}

void VoiceScan(void)
{
    static unsigned char Su8Lock=0;

    if(1==vGu8BeepTimerFlag&&vGu16BeepTimerCnt>0)
    {
        if(0==Su8Lock)

```

```

        {
            Su8Lock=1;
            BeepOpen();
        }
    else
    {

        vGu16BeepTimerCnt--;

        if(0==vGu16BeepTimerCnt)
        {
            Su8Lock=0;
            BeepClose();
        }

    }
}
}

```

void KeyScan(void) //此函数放在定时中断里每 1ms 扫描一次

```

{
    static unsigned char Su8KeyLock1;
    static unsigned int Su16KeyCnt1;

    // 【启动】 按键 K1 的扫描识别
    if(0!=KEY_INPUT1)
    {
        Su8KeyLock1=0;
        Su16KeyCnt1=0;
    }
    else if(0==Su8KeyLock1)
    {
        Su16KeyCnt1++;
        if(Su16KeyCnt1>=KEY_FILTER_TIME)
        {
            Su8KeyLock1=1;
            vGu8KeySec=1;    //触发 1 号键
        }
    }
}
}

```

/\* 注释二：

\* 在 KeyTask 中只改变 Gu8RunStart 的值，用于总启动开关。而运动状态 Gu8RunStatus 是在运动函数



```

* RunTask 中改变，用于对外反馈实时的运动状态。
*/

void KeyTask(void)    //按键的任务函数，放在主函数内
{
    if(0==vGu8KeySec)
    {
        return; //按键的触发序号是 0 意味着无按键触发，直接退出当前函数，不执行此函数下面的代码
    }

    switch(vGu8KeySec) //根据不同的按键触发序号执行对应的代码
    {
        case 1:        //1 号按键。【启动】按键 K1

            if(0==Gu8RunStatus) //根据当前运动的状态来决定“总开关”是否能受按键的控制
            {
                Gu8RunStart=1;    //总开关“打开”。
            }

            vGu8BeepTimerFlag=0;
            vGu16BeepTimerCnt=KEY_VOICE_TIME; //触发按键后，发出固定长度的声音
            vGu8BeepTimerFlag=1;
            vGu8KeySec=0; //响应按键服务处理程序后，按键编号必须清零，避免一直触发
            break;

        default:
            vGu8KeySec=0; //响应按键服务处理程序后，按键编号必须清零，避免一直触发
            break;

    }
}

/* 注释三：
* 本节故意引入三个变量：计数器 Gu16RunCnt，左延时 Gu16ReturnLeftTime，下延时 Gu16GoDownTime。
* 在人机界面的场合，这三个变量可以用来扩展实现设置参数的功能。比如，如果有数码管，可以通过
* 显示 Gu16RunCnt 的数值来让客户看到当前设备的计数器。如果有数码管和按键，可以通过切换到某个
* 界面下，修改 Gu16ReturnLeftTime 和 Gu16GoDownTime 的数值，让客户对设备进行延时参数的设置。
*/

void RunTask(void)    //运动控制的任务函数，放在主函数内
{
    static unsigned char Su8RunStep=0; //运行的步骤

```

```

//当总开关处于“停止”并且“步骤不为0”时，强制把步骤归零。
if(0!=Su8RunStep&&0==Gu8RunStart)
{
    Su8RunStep=0; //步骤归零
}

switch(Su8RunStep) //屡见屡爱的 switch 又来了
{
    case 0:
        if(1==Gu8RunStart) //总开关“打开”
        {
            Gu8RunStatus=1; //及时设置 Gu8RunStatus 的运动状态为“运行”

            GoRight() ;      //“水平气缸”往右跑。P1.4 的 LED 灯“灭”。

            Su8RunStep=1; //切换到下一步
        }
        break;
    case 1:
        if(0==vGu8SensorRight) //直到碰到了“右感应器”（按下 K2），“机械臂”才往下移动。
        {
            GoDown(); //“垂直气缸”往下跑。P1.5 的 LED 灯“灭”。
            vGu8RunTimerFlag=0;
            vGu16RunTimerCnt=Gu16GoDownTime; //向下移动 3 秒的延时赋值
            vGu8RunTimerFlag=1; //启动定时器

            Su8RunStep=2; //切换到下一步
        }
        break;
    case 2:
        if(0==vGu16RunTimerCnt) //当定时的 3 秒时间到，“机械臂”才往上移动，开始原路返回。
        {
            GoUp(); //“垂直气缸”往上跑。P1.5 的 LED 灯“亮”。
            Su8RunStep=3; //切换到下一步
        }
        break;
    case 3:
        if(0==vGu8SensorUp) //直到碰到了“上感应器”（按下 K3），滑块才往左移动。
        {
            GoLeft(); //“水平气缸”往左跑。P1.4 的 LED 灯“亮”。
            vGu8RunTimerFlag=0;
            vGu16RunTimerCnt=Gu16ReturnLeftTime; //向左移动 2 秒的延时赋值
            vGu8RunTimerFlag=1; //启动定时器
        }

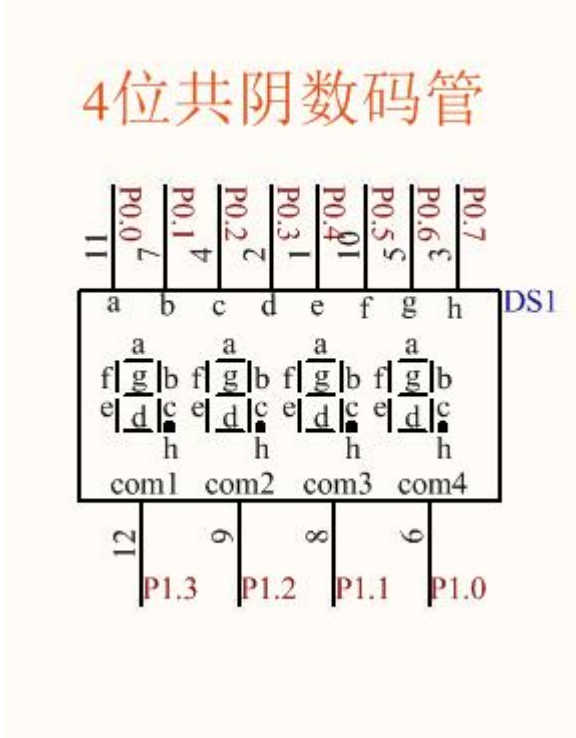
```

```
        Su8RunStep=4; //切换到下一步
    }

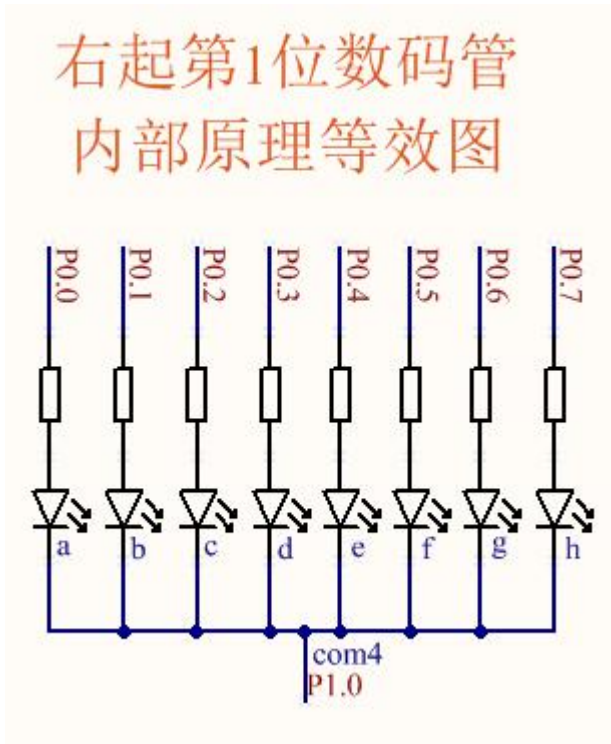
    break;
case 4:
    if (0==vGu16RunTimerCnt) //当定时的 2 秒时间到，完成一次过程。
    {
        Gu16RunCnt++; //计数器加 1，统计设备运行的次数
        Gu8RunStatus=0; //及时设置 Gu8RunStatus 的运动状态为“停止”
        Gu8RunStart=0; //总开关“关闭”，为下一次启动作准备
        Su8RunStep=0; //步骤变量清零，为下一次启动作准备
    }
    break;
}
}
```

第一百一十二节： 数码管显示的基础知识。

【112.1 数码管显示的基础知识。】



上图 112. 1. 1 数码管



上图 112. 1. 2 等效图

如上图 112. 1. 1，一个数码管内部有 8 个段码，每个段码内部对应一颗能发光的 LED 灯，把相关位置的

段码点亮或熄灭就可以显示出不同的数字或者小数点。比如，要显示一个数字“1”，只需要点亮 b 和 c 这两个段码 LED 即可，其它 6 个 a, d, e, f, g, h 段码 LED 熄灭，就可以显示一个数字“1”。再进一步深入分析数码管内部的等效图（上图 112.1.2），com4 是右起第 1 位数码管内部 8 个段码 LED 的公共端，要点亮任何一个段码 LED 的前提必须是公共端 com4 为低电平（P1.0 输出 0 信号）。如果公共端 com4 为高电平（P1.0 输出 1 信号），则不管段码端 P0 口的 8 个 I/O 口输出什么信号，8 个段码 LED 都是熄灭的（无正压差，则无电流无回路）。因此，公共端（比如 com4, com3, com2, com1）就是某个数码管的“总开关”。比如，右起第 1 位数码管要显示数字“1”，要点亮 b 和 c，则 P0.1 和 P0.2 必须输出“1”高电平，其它 P0.0, P0.3, P0.4, P0.5, P0.6, P0.7 必须输出“0”低电平，把这 8 个 I/O 口二进制的信号转换成十六进制，则整个 P0 口总线只需输出一个十六进制的 0x06，最后，“总开关”打开，公共端 com4 输出“0”，即可显示一个数字“1”。如果需要显示其它的不同数字，只需要改变段码端 P0 口的十六进制输出数值即可，如果提前把要显示的数字放在一个数组里，这个数组就是编码转换表，类似于一个字库表。现在编写一个程序例子，右起第 1 个和第 3 个数码管循环显示从 0 到 9 的数字，另外右起第 2 个和第 4 个数码管则关闭不显示，程序代码如下：

```
#include "REG52.H"

#define CHANGE_TIME 1000    //数码管切换显示数字的时间

void T0_time();
void SystemInitial(void) ;
void Delay(unsigned long u32DelayTime) ;
void PeripheralInitial(void) ;

void DisplayTask(void);    //数码管显示的任务函数

sbit P1_0=P1^0; //右起第 1 位数码管的公共端 com4
sbit P1_1=P1^1; //右起第 2 位数码管的公共端 com3
sbit P1_2=P1^2; //右起第 3 位数码管的公共端 com2
sbit P1_3=P1^3; //右起第 4 位数码管的公共端 com1

//根据原理图得出的共阴数码管编码转换表，类似于一个字库表
code unsigned char Cu8DigTable[]=
{
0x3f, //0      序号 0
0x06, //1      序号 1
0x5b, //2      序号 2
0x4f, //3      序号 3
0x66, //4      序号 4
0x6d, //5      序号 5
0x7d, //6      序号 6
0x07, //7      序号 7
0x7f, //8      序号 8
0x6f, //9      序号 9
0x00, //不显示 序号 10
};
```

```

volatile unsigned char vGu8ChangeTimerFlag=0; //控制切换数字的时间的定时器
volatile unsigned int vGu16ChangeTimerCnt=0;

unsigned char Gu8Number=0; //从 0 到 9 依次循环显示的数字

void main()
{
    SystemInitial();
    Delay(10000);
    PeripheralInitial();
    while(1)
    {
        DisplayTask();    //数码管显示的任务函数
    }
}

void DisplayTask(void)    //数码管显示的任务函数
{
    static unsigned char Su8GetCode; //从编码转换表中提取出来的编码。

    if(0==vGu16ChangeTimerCnt) //定时的时间到，更新显示下一个数字，依次循环显示
    {
        Su8GetCode=Cu8DigTable[Gu8Number]; //从编码转换表中提取出来的编码。

        P0=Su8GetCode; //段码端输出需要显示的编码
        P1_0=0; //右起第 1 位数码管的公共端 com4，“总开关”打开，输出低电平 0
        P1_1=1; //右起第 2 位数码管的公共端 com3，“总开关”关闭，输出高电平 1
        P1_2=0; //右起第 3 位数码管的公共端 com2，“总开关”打开，输出低电平 0
        P1_3=1; //右起第 4 位数码管的公共端 com1，“总开关”关闭，输出高电平 1

        Gu8Number++; //显示的数字不断从 0 到 9 累加
        if(Gu8Number>9)
        {
            Gu8Number=0;
        }

        vGu8ChangeTimerFlag=0;
        vGu16ChangeTimerCnt=CHANGE_TIME;
        vGu8ChangeTimerFlag=1; //启动新一轮的定时器
    }
}

void T0_time() interrupt 1
{

```

```

        if(1==vGu8ChangeTimerFlag&&vGu16ChangeTimerCnt>0) //数码管显示切换时间的定时器
        {
            vGu16ChangeTimerCnt--;
        }

        TH0=0xfc;
        TL0=0x66;
    }

void SystemInitial(void)
{
    //初始化上电瞬间数码管的状态
    P1_0=1; //右起第1位数码管的公共端 com4,“总开关”关闭,输出低电平 1
    P1_1=1; //右起第2位数码管的公共端 com3,“总开关”关闭,输出高电平 1
    P1_2=1; //右起第3位数码管的公共端 com2,“总开关”关闭,输出低电平 1
    P1_3=1; //右起第4位数码管的公共端 com1,“总开关”关闭,输出高电平 1

    TMOD=0x01;
    TH0=0xfc;
    TL0=0x66;
    EA=1;
    ET0=1;
    TR0=1;
}

void Delay(unsigned long u32DelayTime)
{
    for(;u32DelayTime>0;u32DelayTime--);
}

void PeripheralInitial(void)
{
}

```

【113.1 动态扫描的数码管。】

P0.7	P0.6	P0.5	P0.4	P0.3	P0.2	P0.1	P0.0
11	7	6	5	4	3	2	1
a	b	c	d	e	f	g	h
DS1							
a	b	c	d	e	f	g	h
f	g	b	e	a	d	c	h
e	d	c	h	a	b	g	f
com1	com2	com3	com4				
12	6	8	9				
P1.3	P1.2	P1.1	P1.0				

上一节，看到打开显示的数码管右起第 1 个 (com4) 和第 3 个 (com2) 在任意时刻显示的数字是一样的，为什么？因为四个数码管的 8 个段码 a, b, c, d, e, f, g, h 所连接的单片机 I/O 口是共用的，如果把四个数码管全部打开 (com1, com2, com3, com4 全部输出低电平)，会发现四个数码管在任意时刻显示的四个数字也是一样的！实际应用中，要四个数码管能各自独立显示不同的数字，就需要用到“分时动态扫描”的方式。所谓分时，就是在任意时刻只能显示其中一个数码管（某个 com 输出低电平），其它三个数码管关闭（其它三个 com 输出高电平），每个数码管显示停留的时间固定一致并且非常短暂，四个数码管依次循环的切换显示，只要切换画面的速度足够快，人的视觉就分辨不出来，感觉八个数码管是同时亮的（实际不是同时亮），跟动画片“1 秒钟动态切换显示多少幅画面”的原理一样。现在编写一个程序例子，四个数码管要显示四个不同的数字“1234”，程序代码如下：

```
#include "REG52.H"

/* 注释一：
*   SCAN_TIME 是每个数码管停留显示的短暂时间。这里称为“扫描时间”。这个时间既不能太长也不能
*   太短，要调试到恰到好处。太长，则影响其它数码管的显示，会让人觉得画面不连贯不是同时亮；
*   太短，又会影响显示的亮度。具体的时间应该根据实际项目不断调试修正而得到最佳显示的数值。
*/

#define SCAN_TIME 1
```



```

void T0_time();
void SystemInitial(void) ;
void Delay(unsigned long u32DelayTime) ;
void PeripheralInitial(void) ;

void DisplayScan(void);    //数码管的动态扫描函数，放在定时中断里。

sbit P1_0=P1^0; //右起第 1 位数码管的公共端 com4
sbit P1_1=P1^1; //右起第 2 位数码管的公共端 com3
sbit P1_2=P1^2; //右起第 3 位数码管的公共端 com2
sbit P1_3=P1^3; //右起第 4 位数码管的公共端 com1

//根据原理图得出的共阴数码管编码转换表，类似于一个字库表
code unsigned char Cu8DigTable[]=
{
0x3f, //0      序号 0
0x06, //1      序号 1
0x5b, //2      序号 2
0x4f, //3      序号 3
0x66, //4      序号 4
0x6d, //5      序号 5
0x7d, //6      序号 6
0x07, //7      序号 7
0x7f, //8      序号 8
0x6f, //9      序号 9
0x00, //不显示 序号 10
};

volatile unsigned char vGu8ScanTimerFlag=0; //动态扫描的定时器
volatile unsigned int vGu16ScanTimerCnt=0;

/* 注释二：
* vGu8Display_Righ_4, vGu8Display_Righ_3, vGu8Display_Righ_2, vGu8Display_Righ_1, 这四个
* 全局变量用来传递每位数码管需要显示的数字，作为对上面应用层调用的接口变量。
*/

volatile unsigned char vGu8Display_Righ_4=1; //右起第 4 位数码管显示的变量。这里显示 “1”
volatile unsigned char vGu8Display_Righ_3=2; //右起第 3 位数码管显示的变量。这里显示 “2”
volatile unsigned char vGu8Display_Righ_2=3; //右起第 2 位数码管显示的变量。这里显示 “3”
volatile unsigned char vGu8Display_Righ_1=4; //右起第 1 位数码管显示的变量。这里显示 “4”

void main()
{
    SystemInitial();

```

```

    Delay(10000);
    PeripheralInitial();
    while(1)
    {
    }
}

/* 注释三：
* DisplayScan 数码管的动态扫描函数，之所以放在定时中断里，是因为动态扫描数码管对时间均匀度
* 要求很高，如果放在 main 主函数中，期间稍微出现一些延时滞后或者超前执行的情况，都会导致
* 数码管出现“闪烁”或者“忽暗忽亮”的显示效果。
*/

void DisplayScan(void)
{
    static unsigned char Su8GetCode; //从编码转换表中提取出来的编码。
    static unsigned char Su8ScanStep=1; //扫描步骤

    if(0==vGu16ScanTimerCnt) //定时的时间到，切换显示下一个数码管，依次动态快速循环切换显示
    {

/* 注释四：
* 在即将切换显示到下一个新的数码管之前，应该先关闭显示所有的数码管，避免因关闭不彻底而导致
* 数码管某些段位出现“漏光”，也就是数码管因程序处理不善而出现常见的“鬼影”显示情况。
*/

        P0=0x00; //输出显示先清零，先关闭显示所有的数码管
        //先关闭所有的 com 口，先关闭显示所有的数码管
        P1_0=1; //右起第 1 位数码管的公共端 com4，“总开关”关闭，输出低电平 1
        P1_1=1; //右起第 2 位数码管的公共端 com3，“总开关”关闭，输出高电平 1
        P1_2=1; //右起第 3 位数码管的公共端 com2，“总开关”关闭，输出低电平 1
        P1_3=1; //右起第 4 位数码管的公共端 com1，“总开关”关闭，输出高电平 1

        switch(Su8ScanStep)
        {
            case 1: //显示右起第 1 个数码管
                Su8GetCode=Cu8DigTable[vGu8Display_Righ_1]; //从编码转换表中提取出来的编码。
                P0=Su8GetCode; //段码端输出需要显示的编码
                P1_0=0; //右起第 1 位数码管的公共端 com4，“总开关”打开，输出低电平 0
                P1_1=1; //右起第 2 位数码管的公共端 com3，“总开关”关闭，输出高电平 1
                P1_2=1; //右起第 3 位数码管的公共端 com2，“总开关”关闭，输出高电平 1
                P1_3=1; //右起第 4 位数码管的公共端 com1，“总开关”关闭，输出高电平 1
                break;

```

```

        case 2: //显示右起第 2 个数码管
            Su8GetCode=Cu8DigTable[vGu8Display_Righ_2]; //从编码转换表中提取出来的编码。
            P0=Su8GetCode; //段码端输出需要显示的编码
            P1_0=1; //右起第 1 位数码管的公共端 com4, “总开关” 关闭, 输出高电平 1
            P1_1=0; //右起第 2 位数码管的公共端 com3, “总开关” 打开, 输出低电平 0
            P1_2=1; //右起第 3 位数码管的公共端 com2, “总开关” 关闭, 输出高电平 1
            P1_3=1; //右起第 4 位数码管的公共端 com1, “总开关” 关闭, 输出高电平 1
            break;

        case 3: //显示右起第 3 个数码管
            Su8GetCode=Cu8DigTable[vGu8Display_Righ_3]; //从编码转换表中提取出来的编码。
            P0=Su8GetCode; //段码端输出需要显示的编码
            P1_0=1; //右起第 1 位数码管的公共端 com4, “总开关” 关闭, 输出高电平 1
            P1_1=1; //右起第 2 位数码管的公共端 com3, “总开关” 关闭, 输出高电平 1
            P1_2=0; //右起第 3 位数码管的公共端 com2, “总开关” 打开, 输出低电平 0
            P1_3=1; //右起第 4 位数码管的公共端 com1, “总开关” 关闭, 输出高电平 1
            break;

        case 4: //显示右起第 4 个数码管
            Su8GetCode=Cu8DigTable[vGu8Display_Righ_4]; //从编码转换表中提取出来的编码。
            P0=Su8GetCode; //段码端输出需要显示的编码
            P1_0=1; //右起第 1 位数码管的公共端 com4, “总开关” 关闭, 输出高电平 1
            P1_1=1; //右起第 2 位数码管的公共端 com3, “总开关” 关闭, 输出高电平 1
            P1_2=1; //右起第 3 位数码管的公共端 com2, “总开关” 关闭, 输出高电平 1
            P1_3=0; //右起第 4 位数码管的公共端 com1, “总开关” 打开, 输出低电平 0
            break;

    }

    Su8ScanStep++;
    if(Su8ScanStep>4) //如果扫描步骤大于 4, 继续从第 1 步开始扫描
    {
        Su8ScanStep=1;
    }

    vGu8ScanTimerFlag=0;
    vGu16ScanTimerCnt=SCAN_TIME;
    vGu8ScanTimerFlag=1; //启动新一轮的定时器
}

}

void T0_time() interrupt 1
{

```

```

DisplayScan(); //数码管的动态扫描函数

if(1==vGu8ScanTimerFlag&&vGu16ScanTimerCnt>0) //数码管显示切换时间的定时器
{
    vGu16ScanTimerCnt--;
}

TH0=0xfc;
TL0=0x66;
}

void SystemInitial(void)
{
    //初始化上电瞬间数码管的状态，关闭显示所有的数码管
    P0=0x00;
    P1_0=1; //右起第1位数码管的公共端 com4，“总开关”关闭，输出低电平 1
    P1_1=1; //右起第2位数码管的公共端 com3，“总开关”关闭，输出高电平 1
    P1_2=1; //右起第3位数码管的公共端 com2，“总开关”关闭，输出低电平 1
    P1_3=1; //右起第4位数码管的公共端 com1，“总开关”关闭，输出高电平 1

    TMOD=0x01;
    TH0=0xfc;
    TL0=0x66;
    EA=1;
    ET0=1;
    TR0=1;
}

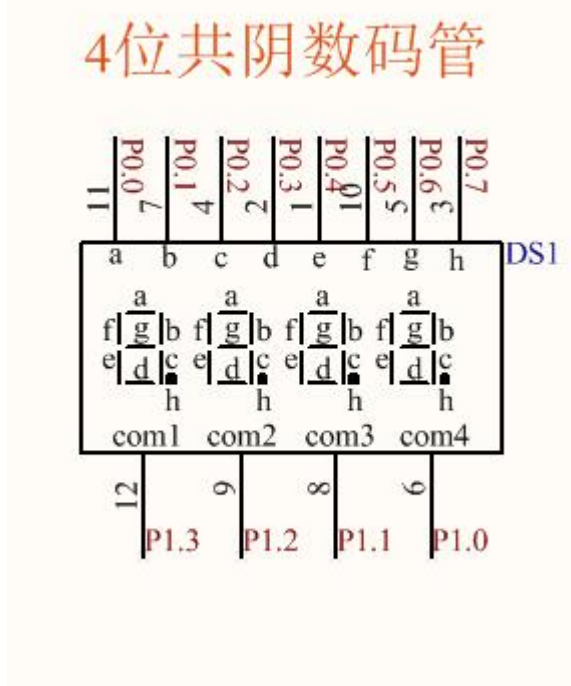
void Delay(unsigned long u32DelayTime)
{
    for(;u32DelayTime>0;u32DelayTime--);
}

void PeripheralInitial(void)
{
}

```

## 第一百一十四节： 动态扫描的数码管显示小数点。

### 【114.1 动态扫描的数码管显示小数点。】



上图 114.1.1 数码管

如上图，小数点的段码是 h，对应单片机的 P0.7 口。数码管编码转换表（类似字库）的 11 个以字节为单位的数据，把它们从十六进制转换成二进制后，可以发现第 7 位（对应 P0.7 口）都是 0。因此，从转换表里取数据后，得到的数据默认是让数码管的小数点不显示的。如果想显示这个小数点，就需要用到“或（|）”语句操作，把第 7 位改为 1。比如，本节程序需要显示“1.234”这个带小数点的数值，代码如下：

```
#include "REG52.H"

#define SCAN_TIME 1

void TO_time();
void SystemInitial(void) ;
void Delay(unsigned long u32DelayTime) ;
void PeripheralInitial(void) ;

void DisplayScan(void);

sbit P1_0=P1^0;
sbit P1_1=P1^1;
sbit P1_2=P1^2;
sbit P1_3=P1^3;
```

//转换表，里面的 11 个数据，转换成二进制后，第 7 位数据都是 0 默认不显示小数点

```
code unsigned char Cu8DigTable[]=
```

```
{
0x3f, //0      序号 0
0x06, //1      序号 1
0x5b, //2      序号 2
0x4f, //3      序号 3
0x66, //4      序号 4
0x6d, //5      序号 5
0x7d, //6      序号 6
0x07, //7      序号 7
0x7f, //8      序号 8
0x6f, //9      序号 9
0x00, //不显示 序号 10
};
```

```
volatile unsigned char vGu8ScanTimerFlag=0;
```

```
volatile unsigned int vGu16ScanTimerCnt=0;
```

```
volatile unsigned char vGu8Display_Righ_4=1; //右起第 4 位数码管显示的变量。这里显示 “1”
volatile unsigned char vGu8Display_Righ_3=2; //右起第 3 位数码管显示的变量。这里显示 “2”
volatile unsigned char vGu8Display_Righ_2=3; //右起第 2 位数码管显示的变量。这里显示 “3”
volatile unsigned char vGu8Display_Righ_1=4; //右起第 1 位数码管显示的变量。这里显示 “4”
```

/\* 注释一：

```
* vGu8Display_Righ_Dot_4, vGu8Display_Righ_Dot_3, vGu8Display_Righ_Dot_2,
* vGu8Display_Righ_Dot_1, 这四个全局变量用来传递每位数码管是否需要显示它的小数点，如果是 1
* 代表需要显示其小数点，如果是 0 则不显示小数点。这四个变量作为对上面应用层调用的接口变量。
*/
```

```
volatile unsigned char vGu8Display_Righ_Dot_4=1; //右起第 4 位数码管的小数点。1 代表打开显示。
volatile unsigned char vGu8Display_Righ_Dot_3=0; //右起第 3 位数码管的小数点。0 代表关闭显示。
volatile unsigned char vGu8Display_Righ_Dot_2=0; //右起第 2 位数码管的小数点。0 代表关闭显示。
volatile unsigned char vGu8Display_Righ_Dot_1=0; //右起第 1 位数码管的小数点。0 代表关闭显示。
```

```
void main()
```

```
{
    SystemInitial();
    Delay(10000);
    PeripheralInitial();
    while(1)
    {
    }
}
```

```
}
```

```
void DisplayScan(void)
```

```
{
```

```
    static unsigned char Su8GetCode;
```

```
    static unsigned char Su8ScanStep=1;
```

```
    if(0==vGu16ScanTimerCnt)
```

```
    {
```

```
        P0=0x00;
```

```
        P1_0=1;
```

```
        P1_1=1;
```

```
        P1_2=1;
```

```
        P1_3=1;
```

```
        switch(Su8ScanStep)
```

```
        {
```

```
            case 1:
```

```
                Su8GetCode=Cu8DigTable[vGu8Display_Righ_1];
```

```
/* 注释二：
```

```
* 这里是本节的关键。通过判断全局的接口变量的数值，来决定是否打开显示小数点。
```

```
* 从转换表取出字模数据后再跟 0x80 进行“或”运算即可把第 7 位数据改为 1。
```

```
*/
```

```
        if(1==vGu8Display_Righ_Dot_1) //如果打开了需要显示第 1 个数码管的小数点
```

```
        {
```

```
            Su8GetCode=Su8GetCode|0x80; //把第 7 位数据改为 1，显示小数点
```

```
        }
```

```
        P0=Su8GetCode;
```

```
        P1_0=0;
```

```
        P1_1=1;
```

```
        P1_2=1;
```

```
        P1_3=1;
```

```
        break;
```

```
            case 2:
```

```
                Su8GetCode=Cu8DigTable[vGu8Display_Righ_2];
```

```
                if(1==vGu8Display_Righ_Dot_2) //如果打开了需要显示第 2 个数码管的小数点
```

```
                {
```

```
                    Su8GetCode=Su8GetCode|0x80; //把第 7 位数据改为 1，显示小数点
```

```

    }
    P0=Su8GetCode;
    P1_0=1;
    P1_1=0;
    P1_2=1;
    P1_3=1;
    break;

case 3:
    Su8GetCode=Cu8DigTable[vGu8Display_Righ_3];
    if(1==vGu8Display_Righ_Dot_3) //如果打开了需要显示第 3 个数码管的小数点
    {
        Su8GetCode=Su8GetCode|0x80; //把第 7 位数据改为 1，显示小数点
    }
    P0=Su8GetCode;
    P1_0=1;
    P1_1=1;
    P1_2=0;
    P1_3=1;
    break;

case 4:
    Su8GetCode=Cu8DigTable[vGu8Display_Righ_4];
    if(1==vGu8Display_Righ_Dot_4) //如果打开了需要显示第 4 个数码管的小数点
    {
        Su8GetCode=Su8GetCode|0x80; //把第 7 位数据改为 1，显示小数点
    }
    P0=Su8GetCode;
    P1_0=1;
    P1_1=1;
    P1_2=1;
    P1_3=0;
    break;

}

Su8ScanStep++;
if (Su8ScanStep>4)
{
    Su8ScanStep=1;
}

vGu8ScanTimerFlag=0;
vGu16ScanTimerCnt=SCAN_TIME;

```



```

        vGu8ScanTimerFlag=1;
    }
}

void T0_time() interrupt 1
{
    DisplayScan();

    if(1==vGu8ScanTimerFlag&&vGu16ScanTimerCnt>0)
    {
        vGu16ScanTimerCnt--;
    }

    TH0=0xfc;
    TL0=0x66;
}

```

```

void SystemInitial(void)
{
    P0=0x00;
    P1_0=1;
    P1_1=1;
    P1_2=1;
    P1_3=1;

    TMOD=0x01;
    TH0=0xfc;
    TL0=0x66;
    EA=1;
    ET0=1;
    TR0=1;
}

```

```

void Delay(unsigned long u32DelayTime)
{
    for(;u32DelayTime>0;u32DelayTime--);
}

```

```

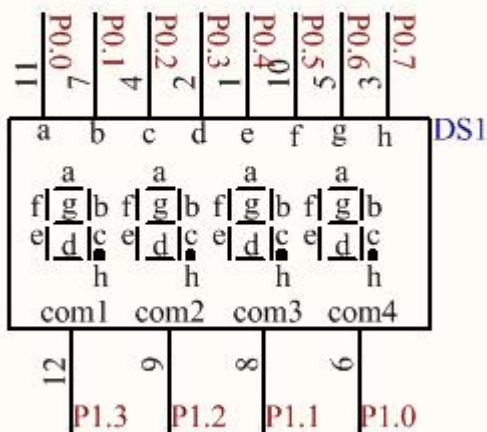
void PeripheralInitial(void)
{
}

```

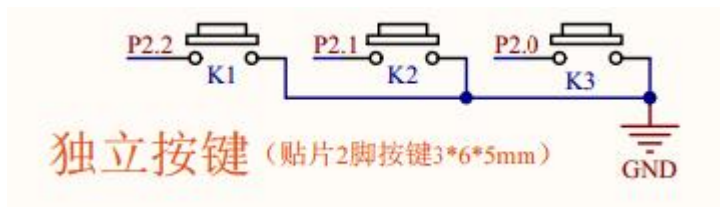
## 第一百一十五节： 按键控制数码管的秒表。

### 【115.1 按键控制数码管的秒表。】

#### 4位共阴数码管



上图 115.1.1 数码管



上图 115.1.2 独立按键

本节通过一个秒表的小项目，让大家学会以下 4 个知识点：

(1) 上层的界面显示框架几乎都要用到更新变量，更新变量包括整屏更新和局部更新，本节只用到整屏更新。更新变量是用全局变量在函数之间传递信息。作用是，当有某个需要显示的数据发生改变的时候，就要给更新变量置 1，让显示函数重新更新一次显示，确保最新的数据能及时显示出来，平时没有数据更新改变的时候不用频繁更新显示避免占用 CPU 过多的时间。

(2) 凡是需要显示数字的地方，都必须涉及如何把一个数据按“个十百千万...”的位逐个提取出来的算法。这个算法比较简单，主要用“求余”和“求商”这两个运算语句就可以随心所欲的把数据位提取出来。除此之外，还要学会如何用 if 语句判断数据的范围，来把高位尚未用到的某个数码管屏蔽，让该位数码管只显示一个“不显示”的数据（避免直接显示一个 0）。

(3) 我常把单片机程序简化成 4 个代表：按键（人机输入），数码管（人机界面），跑马灯（应用程序），串口（通信）。本节的“应用程序”不是跑马灯，而是秒表。不管是跑马灯，还是秒表，都要用到一个总启动 Gu8RunStart 和一个总运行步骤 Gu8RunStep。建议大家，总启动 Gu8RunStart 和总运行步骤 Gu8RunStep

应该成双成对的出现（这样关断响应更及时，并且结构更紧凑，漏洞更少），比如，凡是总启动 Gu8RunStart 发生改变的时候，总运行步骤 Gu8RunStep 都复位归零一下。

（4）一个硬件的定时器中断，可以衍生出 N 个软件定时器，之前跟大家介绍的是“递减式”的软件定时器，而且实际应用中，“递减式”的软件定时器也是用得最多。本节因为项目的需要，需要用到的是“累加式”的软件定时器。不管是哪种软件定时器，大家都要注意定时器变量在定义时所用到的数据类型，这个数据类型决定了定时时间的长度，比如在 51 单片机中，unsigned int 的范围是 0 到 65535，最大一次性定时 65.535 秒。而 unsigned long 的范围是 0 到 4294967295，最大一次性定时 4294967.295 秒。本节秒表的时间超过 65.535 秒，因此需要用到 unsigned long 类型的定时器变量。

本节秒表程序的功能：K1 按键是复位按键，每按一次，秒表都停止并且重新归零。K2 按键是启动和暂停按键，当秒表处于复位后停止的状态时按一次则开始启动，当秒表处于正在工作的状态时按一次则处于暂停状态，当秒表处于暂停的状态时按一次则继续处于工作的状态。本节 4 位数数码管，显示的时间是带 2 位小数点的，能显示的时间范围是：0.00 秒到 99.99 秒。代码如下：

```
#include "REG52.H"

#define KEY_FILTER_TIME 25

#define SCAN_TIME 1

void T0_time();
void SystemInitial(void);
void Delay(unsigned long u32DelayTime);
void PeripheralInitial(void);

void KeyScan(void);
void KeyTask(void);

void DisplayScan(void); //底层显示的驱动函数
void DisplayTask(void); //上层显示的任务函数

void RunTask(void); //秒表的应用程序

sbit KEY_INPUT1=P2^2;
sbit KEY_INPUT2=P2^1;

sbit P1_0=P1^0;
sbit P1_1=P1^1;
sbit P1_2=P1^2;
sbit P1_3=P1^3;

//数码管转换表
code unsigned char Cu8DigTable[]=
{
0x3f, //0      序号 0
```

```

0x06, //1      序号 1
0x5b, //2      序号 2
0x4f, //3      序号 3
0x66, //4      序号 4
0x6d, //5      序号 5
0x7d, //6      序号 6
0x07, //7      序号 7
0x7f, //8      序号 8
0x6f, //9      序号 9
0x00, //不显示 序号 10
};

//数码管底层驱动扫描的软件定时器
volatile unsigned char vGu8ScanTimerFlag=0;
volatile unsigned int vGu16ScanTimerCnt=0;

//秒表的软件定时器，注意，这里是 unsigned long 类型，范围是 0 到 4294967295 毫秒
volatile unsigned char vGu8StopWatchTimerFlag=0;
volatile unsigned long vGu32StopWatchTimerCnt=0;

//数码管上层每 10ms 就定时刷新一次显示的软件定时器。用于及时更新显示秒表当前的实时数值
volatile unsigned char vGu8UpdateTimerFlag=0;
volatile unsigned int vGu16UpdateTimerCnt=0;

unsigned char Gu8RunStart=0; //应用程序的总启动
unsigned char Gu8RunStep=0; //应用程序的总运行步骤。建议跟 vGu8RunStart 成双成对出现
unsigned char Gu8RunStatus=0; //当前秒表的状态。0 代表停止，1 代表正在工作中，2 代表暂停

unsigned char Gu8WdUpdate=1; //开机默认整屏更新一次显示。此变量在显示框架中是非常重要的变量

volatile unsigned char vGu8Display_Righ_4=10; //开机默认最高位数码管显示一个“不显示”数据
volatile unsigned char vGu8Display_Righ_3=0;
volatile unsigned char vGu8Display_Righ_2=0;
volatile unsigned char vGu8Display_Righ_1=0;

volatile unsigned char vGu8Display_Righ_Dot_4=0;
volatile unsigned char vGu8Display_Righ_Dot_3=1; //开机默认保留显示 2 个小数点
volatile unsigned char vGu8Display_Righ_Dot_2=0;
volatile unsigned char vGu8Display_Righ_Dot_1=0;

volatile unsigned char vGu8KeySec=0;

void main()

```

```

{
    SystemInitial();
    Delay(10000);
    PeripheralInitial();
    while(1)
    {
        KeyTask();      //按键的任务函数
        DisplayTask();  //数码管显示的上层任务函数
        RunTask();      //秒表的应用程序
    }
}

void KeyTask(void)      //按键的任务函数
{
    if(0==vGu8KeySec)
    {
        return;
    }

    switch(vGu8KeySec)
    {
        case 1:        //复位按键

            Gu8RunStatus=0; //秒表返回停止的状态

            Gu8RunStart=0; //秒表停止
            Gu8RunStep=0; //总运行步骤归零。建议跟 vGu8RunStart 成双成对出现

            vGu8StopWatchTimerFlag=0;
            vGu32StopWatchTimerCnt=0; //秒表的软件定时器清零

            Gu8WdUpdate=1; //整屏更新一次显示

            vGu8KeySec=0;
            break;

        case 2:        //启动与暂停的按键
            if(0==Gu8RunStatus) //在停止状态下
            {
                Gu8RunStatus=1; //秒表处于工作状态

                vGu8StopWatchTimerFlag=0;
                vGu32StopWatchTimerCnt=0;
                vGu8StopWatchTimerFlag=1; //启动秒表的软件定时器
            }
        }
    }
}

```

```

        Gu8RunStart=1;    //秒表总开关启动
        Gu8RunStep=0;    //总运行步骤归零。建议跟 vGu8RunStart 成双成对出现
    }
    else if(1==Gu8RunStatus) //在工作状态下
    {
        Gu8RunStatus=2; //秒表处于暂停状态
    }
    else //在暂停状态下
    {
        Gu8RunStatus=1; //秒表处于工作状态
    }

    Gu8WdUpdate=1; //整屏更新一次显示，确保在暂停的时候能显示到最新的数据

    vGu8KeySec=0;
    break;
}
}

void DisplayTask(void) //数码管显示的上层任务函数
{
    //需要借用的中间变量，用来拆分数数据位
    static unsigned char Su8Temp_4, Su8Temp_3, Su8Temp_2, Su8Temp_1; //需要借用的中间变量

    /* 注释一：
    * 此处为什么要多加 4 个中间过渡变量 Su8Temp_X? 是因为 vGu32StopWatchTimerCnt 分解数据的时候
    * 需要进行除法和求余数的运算，就会用到好多条指令，就会耗掉一点时间，类似延时了一会。我们
    * 的定时器每隔一段时间都会产生中断，然后在中断里驱动数码管显示，当 vGu32StopWatchTimerCnt
    * 还没完全分解出 4 位有效数据时，这个时候来的定时中断，就有可能导致显示的数据瞬间产生不完整，
    * 影响显示效果。因此，为了把需要显示的数据过渡最快，所以采取了先分解，再过渡显示的方法。
    */

    if(1==Gu8WdUpdate) //如果需要整屏更新
    {
        Gu8WdUpdate=0; //及时清零，只更新一次显示即可，避免一直进来更新显示

        //先分解数据

        //Su8Temp_4 提取“十秒”位。
        Su8Temp_4=vGu32StopWatchTimerCnt/10000%10; //实际精度是 0.001 秒,但显示精度是 0.01 秒
    }
}

```

```

//Su8Temp_3 提取“个秒”位。
Su8Temp_3=vGu32StopWatchTimerCnt/1000%10; //实际精度是 0.001 秒,但显示精度是 0.01 秒

//Su8Temp_2 提取“百毫秒”位。
Su8Temp_2=vGu32StopWatchTimerCnt/100%10; //实际精度是 0.001 秒,但显示精度是 0.01 秒

//Su8Temp_1 提取“十毫秒”位。
Su8Temp_1=vGu32StopWatchTimerCnt/10%10; //实际精度是 0.001 秒,但显示精度是 0.01 秒

//判断数据范围,来决定最高位数码管是否需要显示。
if(vGu32StopWatchTimerCnt<10000) //10.000 秒。实际 4 位数码管最大只能显示 99.99 秒
{
    Su8Temp_4=10; //在数码管转换表里,10 代表一个“不显示”的数据
}

//上面先分解数据之后,再过渡需要显示的数据到底层驱动变量里,让过渡的时间越短越好
vGu8Display_Righ_4=Su8Temp_4; //过渡需要显示的数据到底层驱动变量
vGu8Display_Righ_3=Su8Temp_3;
vGu8Display_Righ_2=Su8Temp_2;
vGu8Display_Righ_1=Su8Temp_1;

vGu8Display_Righ_Dot_4=0;
vGu8Display_Righ_Dot_3=1; //保留显示 2 位小数点
vGu8Display_Righ_Dot_2=0;
vGu8Display_Righ_Dot_1=0;

}
}

void RunTask(void) //秒表的应用程序
{
    if(0==Gu8RunStart)
    {
        return; // 如果秒表处于停止状态,则直接退出当前函数,不执行该函数以下的其它代码
    }

    switch(Gu8RunStep)
    {
        case 0: //在这个步骤里,主要用来初始化一些参数

            vGu8UpdateTimerFlag=0;
            vGu16UpdateTimerCnt=10; //每 10ms 更新显示一次当前秒表的时间
            vGu8UpdateTimerFlag=1;

```

```

        Gu8RunStep=1; //跳转到每 10ms 更新显示一次的步骤里
        break;

    case 1: //每 10ms 更新一次显示，确保实时显示秒表当前的时间
        if(0==vGu16UpdateTimerCnt) //每 10ms 更新显示一次当前秒表的时间
        {
            vGu8UpdateTimerFlag=0;
            vGu16UpdateTimerCnt=10; //重置定时器，为下一个 10ms 更新做准备
            vGu8UpdateTimerFlag=1;

            Gu8WdUpdate=1; //整屏更新一次显示当前秒表的时间
        }
        break;
    }
}

void KeyScan(void) //按键底层的驱动扫描函数，放在定时中断函数里
{
    static unsigned char Su8KeyLock1;
    static unsigned int Su16KeyCnt1;
    static unsigned char Su8KeyLock2;
    static unsigned int Su16KeyCnt2;

    if(0!=KEY_INPUT1)
    {
        Su8KeyLock1=0;
        Su16KeyCnt1=0;
    }
    else if(0==Su8KeyLock1)
    {
        Su16KeyCnt1++;
        if(Su16KeyCnt1>=KEY_FILTER_TIME)
        {
            Su8KeyLock1=1;
            vGu8KeySec=1;
        }
    }
}

if(0!=KEY_INPUT2)
{
    Su8KeyLock2=0;

```



```

        Su16KeyCnt2=0;
    }
    else if(0==Su8KeyLock2)
    {
        Su16KeyCnt2++;
        if(Su16KeyCnt2>=KEY_FILTER_TIME)
        {
            Su8KeyLock2=1;
            vGu8KeySec=2;
        }
    }
}

```

void DisplayScan(void) //数码管底层的驱动扫描函数，放在定时中断函数里

```

{
    static unsigned char Su8GetCode;
    static unsigned char Su8ScanStep=1;

    if(0==vGu16ScanTimerCnt)
    {

        P0=0x00;
        P1_0=1;
        P1_1=1;
        P1_2=1;
        P1_3=1;

        switch(Su8ScanStep)
        {
            case 1:
                Su8GetCode=Cu8DigTable[vGu8Display_Righ_1];

                if(1==vGu8Display_Righ_Dot_1)
                {
                    Su8GetCode=Su8GetCode|0x80;
                }
                P0=Su8GetCode;
                P1_0=0;
                P1_1=1;
                P1_2=1;
                P1_3=1;
                break;

```

```

case 2:
    Su8GetCode=Cu8DigTable[vGu8Display_Righ_2];
    if(1==vGu8Display_Righ_Dot_2)
    {
        Su8GetCode=Su8GetCode|0x80;
    }
    P0=Su8GetCode;
    P1_0=1;
    P1_1=0;
    P1_2=1;
    P1_3=1;
    break;

case 3:
    Su8GetCode=Cu8DigTable[vGu8Display_Righ_3];
    if(1==vGu8Display_Righ_Dot_3)
    {
        Su8GetCode=Su8GetCode|0x80;
    }
    P0=Su8GetCode;
    P1_0=1;
    P1_1=1;
    P1_2=0;
    P1_3=1;
    break;

case 4:
    Su8GetCode=Cu8DigTable[vGu8Display_Righ_4];
    if(1==vGu8Display_Righ_Dot_4)
    {
        Su8GetCode=Su8GetCode|0x80;
    }
    P0=Su8GetCode;
    P1_0=1;
    P1_1=1;
    P1_2=1;
    P1_3=0;
    break;
}

Su8ScanStep++;
if (Su8ScanStep>4)
{

```

```

        Su8ScanStep=1;
    }

    vGu8ScanTimerFlag=0;
    vGu16ScanTimerCnt=SCAN_TIME;
    vGu8ScanTimerFlag=1;
}

void TO_time() interrupt 1
{
    KeyScan();      //按键底层的驱动扫描函数
    DisplayScan();  //数码管底层的驱动扫描函数

    if(1==vGu8ScanTimerFlag&&vGu16ScanTimerCnt>0)
    {
        vGu16ScanTimerCnt--; //递减式的软件定时器
    }

    //每 10ms 就定时更新一次显示的软件定时器
    if(1==vGu8UpdateTimerFlag&&vGu16UpdateTimerCnt>0)
    {
        vGu16UpdateTimerCnt--; //递减式的软件定时器
    }

    //秒表实际走的时间的软件定时器，注意，这里是“累加式”的软件定时器。
    //当秒表处于工作的状态 1==Gu8RunStatus
    if(1==vGu8StopWatchTimerFlag&&1==Gu8RunStatus&&vGu32StopWatchTimerCnt<0xffffffff)
    {
        vGu32StopWatchTimerCnt++; //累加式的软件定时器
    }

    TH0=0xfc;
    TL0=0x66;
}

void SystemInitial(void)
{
    P0=0x00;
    P1_0=1;
    P1_1=1;
    P1_2=1;

```

```
P1_3=1;

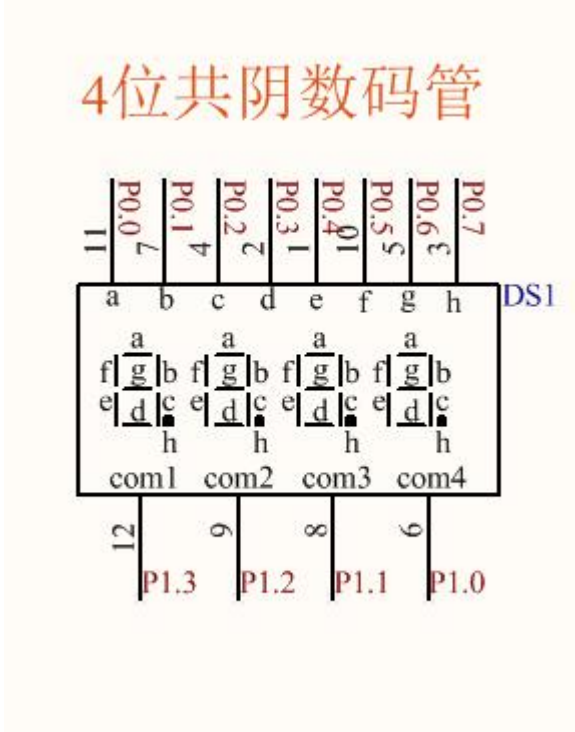
TMOD=0x01;
TH0=0xfc;
TL0=0x66;
EA=1;
ET0=1;
TR0=1;
}

void Delay(unsigned long u32DelayTime)
{
    for(;u32DelayTime>0;u32DelayTime--);
}

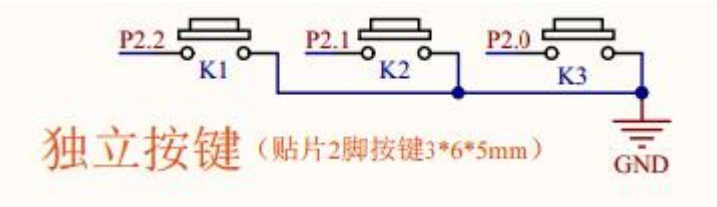
void PeripheralInitial(void)
{
}
}
```

第一百一十六节： 按键控制数码管的倒计时。

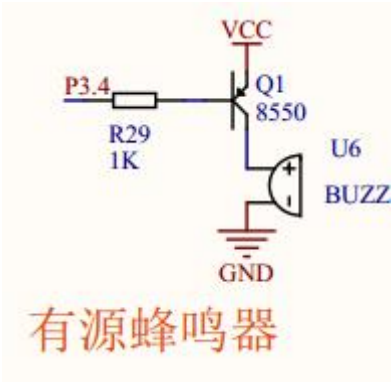
【116.1 按键控制数码管的倒计时。】



上图 116. 1. 1 数码管



上图 116. 1. 2 独立按键



上图 116. 1. 3 有源蜂鸣器

上一节讲“累加式”的秒表，这一节讲“递减式”的倒计时。通过此小项目，加深理解在显示框架中常用的更新变量（整屏更新和局部更新），以及应用程序与按键是如何关联的框架。同时，在拆分“个十百千万...”的时候，有一处地方必须注意，“先整除后求余”必须用一行代码一气呵成，不能拆分成两行代码“先整除；后求余”，否则会有隐患会有 bug。除非，把四个临时变都改成 unsigned long 类型。比如：

以下这样是对的：

```
static unsigned char Su8Temp_1;
Su8Temp_1=vGu32CountdownTimerCnt/10%10; //一气呵成，没毛病。
```

以下这样是有隐患有 bug 的（除非把 Su8Temp\_1 改成 unsigned long 类型）：

```
static unsigned char Su8Temp_1;
Su8Temp_1=vGu32CountdownTimerCnt/10;
Su8Temp_1=Su8Temp_1%10; //拆分成两行代码后，有隐患有 bug。数据溢出的原因引起。
```

本节倒计时程序的功能：K1 按键是复位按键，每按一次，倒计时停止并且重新恢复初始值 10.00 秒。K2 按键是启动按键，当秒表处于复位后停止的状态时按一次则开始启动倒计时，当倒计时变为 0.00 秒的时候，蜂鸣器发出一次“滴”的提示声。此时，如果想启动新一轮的倒计时，只需按一次 K1 复位键，然后再按 K2 启动按键，就可以启动新一轮 10.00 秒的倒计时。代码如下：

```
#include "REG52.H"

#define KEY_FILTER_TIME  25

#define SCAN_TIME  1

#define VOICE_TIME  50    //蜂鸣器一次“滴”的声音长度 50ms

void TO_time();
void SystemInitial(void) ;
void Delay(unsigned long u32DelayTime) ;
void PeripheralInitial(void) ;

void KeyScan(void);
void KeyTask(void);

void VoiceScan(void); //蜂鸣器的驱动函数
void DisplayScan(void); //底层显示的驱动函数
void DisplayTask(void); //上层显示的任务函数

void RunTask(void); //倒计时的应用程序

void BeepOpen(void);
void BeepClose(void);
```

```

sbit KEY_INPUT1=P2^2;
sbit KEY_INPUT2=P2^1;

sbit P1_0=P1^0;
sbit P1_1=P1^1;
sbit P1_2=P1^2;
sbit P1_3=P1^3;

sbit P3_4=P3^4;

//数码管转换表
code unsigned char Cu8DigTable[]=
{
0x3f,  //0      序号 0
0x06,  //1      序号 1
0x5b,  //2      序号 2
0x4f,  //3      序号 3
0x66,  //4      序号 4
0x6d,  //5      序号 5
0x7d,  //6      序号 6
0x07,  //7      序号 7
0x7f,  //8      序号 8
0x6f,  //9      序号 9
0x00,  //不显示 序号 10
};

//数码管底层驱动扫描的软件定时器
volatile unsigned char vGu8ScanTimerFlag=0;
volatile unsigned int vGu16ScanTimerCnt=0;

//倒计时的软件定时器，注意，这里是 unsigned long 类型，范围是 0 到 4294967295 毫秒
volatile unsigned char vGu8CountdownTimerFlag=0;
volatile unsigned long vGu32CountdownTimerCnt=10000;  //开机默认显示初始值 10.000 秒

//数码管上层每 10ms 就定时刷新一次显示的软件定时器。用于及时更新显示秒表当前的实时数值
volatile unsigned char vGu8UpdateTimerFlag=0;
volatile unsigned int vGu16UpdateTimerCnt=0;

//蜂鸣器的软件定时器
volatile unsigned char vGu8BeepTimerFlag=0;
volatile unsigned int vGu16BeepTimerCnt=0;

unsigned char Gu8RunStart=0;  //应用程序的总启动
unsigned char Gu8RunStep=0;  //应用程序的总运行步骤。建议跟 vGu8RunStart 成双成对出现

```

```

unsigned char Gu8RunStatus=0; //当前倒计时的状态。0 代表停止，1 代表正在工作中

unsigned char Gu8WdUpdate=1; //开机默认整屏更新一次显示。此变量在显示框架中是非常重要的变量

volatile unsigned char vGu8Display_Righ_4=1; //显示“1”，跟 vGu32CountdownTimerCnt 高位一致
volatile unsigned char vGu8Display_Righ_3=0;
volatile unsigned char vGu8Display_Righ_2=0;
volatile unsigned char vGu8Display_Righ_1=0;

volatile unsigned char vGu8Display_Righ_Dot_4=0;
volatile unsigned char vGu8Display_Righ_Dot_3=1; //开机默认保留显示 2 个小数点
volatile unsigned char vGu8Display_Righ_Dot_2=0;
volatile unsigned char vGu8Display_Righ_Dot_1=0;

volatile unsigned char vGu8KeySec=0;

void main()
{
    SystemInitial();
    Delay(10000);
    PeripheralInitial();
    while(1)
    {
        KeyTask(); //按键的任务函数
        DisplayTask(); //数码管显示的上层任务函数
        RunTask(); //倒计时的应用程序
    }
}

void KeyTask(void) //按键的任务函数
{
    if(0==vGu8KeySec)
    {
        return;
    }

    switch(vGu8KeySec)
    {
        case 1: //复位按键

            Gu8RunStatus=0; //倒计时返回停止的状态

            Gu8RunStart=0; //倒计时的运行步骤的停止
            Gu8RunStep=0; //总运行步骤归零。建议跟 vGu8RunStart 成双成对出现
    }
}

```



```

    vGu8CountdownTimerFlag=0;    //禁止倒计时开始(而 Gu8RunStart 是启动开关, 不矛盾)
    vGu32CountdownTimerCnt=10000; //倒计时的软件定时器恢复初始值 10.000 秒

    Gu8WdUpdate=1; //整屏更新一次显示

    vGu8KeySec=0;
    break;

case 2:    //启动的按键
    if(0==Gu8RunStatus) //在停止状态下
    {

        vGu8CountdownTimerFlag=0;
        vGu32CountdownTimerCnt=10000; //初始值是 10.000 秒
        vGu8CountdownTimerFlag=1;    //允许倒计时的软件定时器的启动

        Gu8RunStatus=1; //倒计时处于工作状态(并且, 这一瞬间才正式启动倒计时)

        Gu8RunStart=1;    //倒计时的运行步骤的总开关开启
        Gu8RunStep=0;    //总运行步骤归零。建议跟 vGu8RunStart 成双成对出现

        Gu8WdUpdate=1; //整屏更新一次显示, 确保在启动的时候能显示到最新的数据
    }

    vGu8KeySec=0;
    break;
}
}

void DisplayTask(void) //数码管显示的上层任务函数
{
    //需要借用的中间变量, 用来拆分数数据位。
    static unsigned char Su8Temp_4, Su8Temp_3, Su8Temp_2, Su8Temp_1; //需要借用的中间变量

    if(1==Gu8WdUpdate) //如果需要整屏更新
    {
        Gu8WdUpdate=0; //及时清零, 只更新一次显示即可, 避免一直进来更新显示

        //先分解数据, 注意, 这里分解的时候, “先整除后求余” 必须用一行代码一气呵成, 不能拆
        //分成两行代码, 否则会有隐患会有 bug。除非, 把四个临时变都改成 unsigned long 类型。
    }
}

```

```

//Su8Temp_4 提取“十秒”位。
Su8Temp_4=vGu32CountdownTimerCnt/10000%10; //实际精度是 0.001 秒,但显示精度是 0.01 秒

//Su8Temp_3 提取“个秒”位。
Su8Temp_3=vGu32CountdownTimerCnt/1000%10; //实际精度是 0.001 秒,但显示精度是 0.01 秒

//Su8Temp_2 提取“百毫秒”位。
Su8Temp_2=vGu32CountdownTimerCnt/100%10; //实际精度是 0.001 秒,但显示精度是 0.01 秒

//Su8Temp_1 提取“十毫秒”位。
Su8Temp_1=vGu32CountdownTimerCnt/10%10; //实际精度是 0.001 秒,但显示精度是 0.01 秒

//判断数据范围,来决定最高位数码管是否需要显示。
if(vGu32CountdownTimerCnt<10000) //10.000 秒。实际 4 位数码管最大只能显示 99.99 秒
{
    Su8Temp_4=10; //在数码管转换表里,10 代表一个“不显示”的数据
}

//上面先分解数据之后,再过渡需要显示的数据到底层驱动变量里,让过渡的时间越短越好
vGu8Display_Righ_4=Su8Temp_4; //过渡需要显示的数据到底层驱动变量
vGu8Display_Righ_3=Su8Temp_3;
vGu8Display_Righ_2=Su8Temp_2;
vGu8Display_Righ_1=Su8Temp_1;

vGu8Display_Righ_Dot_4=0;
vGu8Display_Righ_Dot_3=1; //保留显示 2 位小数点
vGu8Display_Righ_Dot_2=0;
vGu8Display_Righ_Dot_1=0;

}
}

void RunTask(void) //倒计时的应用程序
{
    if(0==Gu8RunStart)
    {
        return; // 如果总开关处于停止状态,则直接退出当前函数,不执行该函数以下的其它代码
    }

    switch(Gu8RunStep)
    {
        case 0: //在这个步骤里,主要用来初始化一些参数

            vGu8UpdateTimerFlag=0;

```

```

vGu16UpdateTimerCnt=10; //每 10ms 更新显示一次当前倒计时的时间
vGu8UpdateTimerFlag=1;

Gu8RunStep=1; //跳转到每 10ms 更新显示一次的步骤里
break;

case 1: //每 10ms 更新一次显示，确保实时显示当前倒计时的时间
    if(0==vGu16UpdateTimerCnt) //每 10ms 更新显示一次当前倒计时的时间
    {

        vGu8UpdateTimerFlag=0;
        vGu16UpdateTimerCnt=10; //重置定时器，为下一个 10ms 更新做准备
        vGu8UpdateTimerFlag=1;

        Gu8WdUpdate=1; //整屏更新一次显示当前倒计时的时间

        if(0==vGu32CountdownTimerCnt) //如果倒计时的时间到，则跳转到结束的步骤
        {
            Gu8RunStep=2; //跳转到倒计时结束的步骤
        }

    }
    break;

case 2: //倒计时结束的步骤
    //Gu8RunStatus=0; //这行代码注释掉，让每次新启动之前都必须按一次 K1 复位按键才有效

    Gu8RunStart=0; //倒计时的运行步骤的停止
    Gu8RunStep=0; //总运行步骤归零。建议跟 vGu8RunStart 成双成对出现

    vGu8BeepTimerFlag=0;
    vGu16BeepTimerCnt=VOICE_TIME; //蜂鸣器发出“滴”一声
    vGu8BeepTimerFlag=1;

    Gu8WdUpdate=1; //整屏更新一次显示当前倒计时的时间

    break;

}
}

```

```
void KeyScan(void) //按键底层的驱动扫描函数，放在定时中断函数里
```

```
{
    static unsigned char Su8KeyLock1;
    static unsigned int  Su16KeyCnt1;
    static unsigned char Su8KeyLock2;
    static unsigned int  Su16KeyCnt2;

    if(0!=KEY_INPUT1)
    {
        Su8KeyLock1=0;
        Su16KeyCnt1=0;
    }
    else if(0==Su8KeyLock1)
    {
        Su16KeyCnt1++;
        if(Su16KeyCnt1>=KEY_FILTER_TIME)
        {
            Su8KeyLock1=1;
            vGu8KeySec=1;
        }
    }

    if(0!=KEY_INPUT2)
    {
        Su8KeyLock2=0;
        Su16KeyCnt2=0;
    }
    else if(0==Su8KeyLock2)
    {
        Su16KeyCnt2++;
        if(Su16KeyCnt2>=KEY_FILTER_TIME)
        {
            Su8KeyLock2=1;
            vGu8KeySec=2;
        }
    }
}
```

```
void DisplayScan(void) //数码管底层的驱动扫描函数，放在定时中断函数里
```

```
{
    static unsigned char Su8GetCode;
    static unsigned char Su8ScanStep=1;

    if(0==vGu16ScanTimerCnt)
```

```

{

P0=0x00;
P1_0=1;
P1_1=1;
P1_2=1;
P1_3=1;

switch(Su8ScanStep)
{
    case 1:
        Su8GetCode=Cu8DigTable[vGu8Display_Righ_1];

        if(1==vGu8Display_Righ_Dot_1)
        {
            Su8GetCode=Su8GetCode|0x80;
        }
        P0=Su8GetCode;
        P1_0=0;
        P1_1=1;
        P1_2=1;
        P1_3=1;
        break;

    case 2:
        Su8GetCode=Cu8DigTable[vGu8Display_Righ_2];
        if(1==vGu8Display_Righ_Dot_2)
        {
            Su8GetCode=Su8GetCode|0x80;
        }
        P0=Su8GetCode;
        P1_0=1;
        P1_1=0;
        P1_2=1;
        P1_3=1;
        break;

    case 3:
        Su8GetCode=Cu8DigTable[vGu8Display_Righ_3];
        if(1==vGu8Display_Righ_Dot_3)
        {
            Su8GetCode=Su8GetCode|0x80;
        }
}

```

```

        P0=Su8GetCode;
        P1_0=1;
        P1_1=1;
        P1_2=0;
        P1_3=1;
        break;

    case 4:
        Su8GetCode=Cu8DigTable[vGu8Display_Righ_4];
        if(1==vGu8Display_Righ_Dot_4)
        {
            Su8GetCode=Su8GetCode|0x80;
        }
        P0=Su8GetCode;
        P1_0=1;
        P1_1=1;
        P1_2=1;
        P1_3=0;
        break;

    }

    Su8ScanStep++;
    if(Su8ScanStep>4)
    {
        Su8ScanStep=1;
    }

    vGu8ScanTimerFlag=0;
    vGu16ScanTimerCnt=SCAN_TIME;
    vGu8ScanTimerFlag=1;
}
}

```

void VoiceScan(void) //蜂鸣器的驱动函数

```

{

    static unsigned char Su8Lock=0;

    if(1==vGu8BeepTimerFlag&&vGu16BeepTimerCnt>0)
    {
        if(0==Su8Lock)
        {

```

```

        Su8Lock=1;
        BeepOpen();
    }
    else
    {

        vGu16BeepTimerCnt--;

        if(0==vGu16BeepTimerCnt)
        {
            Su8Lock=0;
            BeepClose();
        }

    }
}

void BeepOpen(void)
{
    P3_4=0;
}

void BeepClose(void)
{
    P3_4=1;
}

void T0_time() interrupt 1
{
    VoiceScan();    //蜂鸣器的驱动函数
    KeyScan();      //按键底层的驱动扫描函数
    DisplayScan();  //数码管底层的驱动扫描函数

    if(1==vGu8ScanTimerFlag&&vGu16ScanTimerCnt>0)
    {
        vGu16ScanTimerCnt--; //递减式的软件定时器
    }

    //每 10ms 就定时更新一次显示的软件定时器
    if(1==vGu8UpdateTimerFlag&&vGu16UpdateTimerCnt>0)
    {
        vGu16UpdateTimerCnt--; //递减式的软件定时器
    }
}

```

```

//倒计时实际走的时间的软件定时器，注意，这里还附加了启动状态的条件“&&1==Gu8RunStatus”
if (1==vGu8CountdownTimerFlag&&vGu32CountdownTimerCnt>0&&1==Gu8RunStatus)
{
    vGu32CountdownTimerCnt--; //递减式的软件定时器
}

    TH0=0xfc;
    TL0=0x66;
}

void SystemInitial(void)
{
    P0=0x00;
    P1_0=1;
    P1_1=1;
    P1_2=1;
    P1_3=1;

    TMOD=0x01;
    TH0=0xfc;
    TL0=0x66;
    EA=1;
    ET0=1;
    TR0=1;
}

void Delay(unsigned long u32DelayTime)
{
    for(;u32DelayTime>0;u32DelayTime--);
}

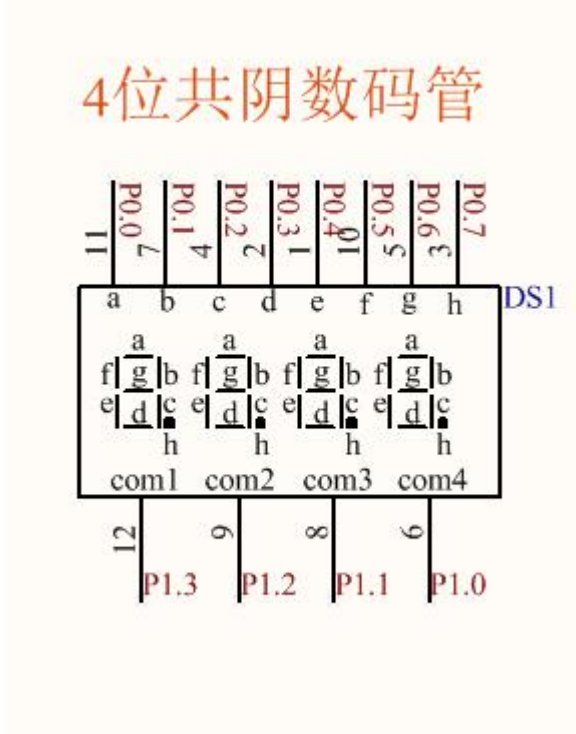
void PeripheralInitial(void)
{
}

```

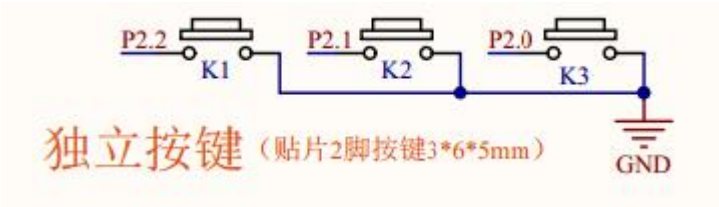


第一百一十七节： 按键切换数码管窗口来设置参数。

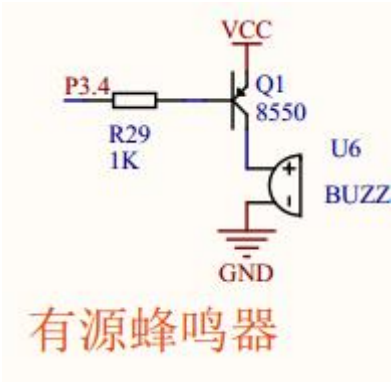
【117.1 按键切换数码管窗口来设置参数。】



上图 117.1.1 数码管



上图 117.1.2 独立按键



上图 117.1.3 有源蜂鸣器

单片机是“数据”驱动型的。按什么逻辑跑，以什么方式跑，都是“数据”决定的。人机交互的核心就是“人”以什么渠道去更改“机”内部的某些“数据”。在程序框架层面，按键更改或者编辑某些数据，我的核心思路都是“在某个窗口下去更改某个特定的数据”，如果某个窗口的数据很多，就需要在此窗口下进一步细分，细分为“某个窗口下的某个局部（子菜单、光标选择）”。可见，“窗口”是支点，“局部”是支点中再细分出来的支点。窗口对应一个名叫“窗口选择”的全局变量 Gu8Wd，局部（子菜单、光标选择）对应一个名叫“局部选择”的全局变量 Gu8Part。数据发生变化的时候，才需要更新显示到数码管上，平时不用一直更新显示，因此，与“窗口选择”Gu8Wd 还对应一个名叫“整屏更新”的全局变量 Gu8WdUpdate，与“局部选择”Gu8Part 还对应一个名叫“局部更新”的全局变量 Gu8PartUpdate。本节的小项目程序只用到“窗口”，没有用到“局部”。

本节小项目的程序功能，利用按键与数码管的人机交互，可以对单片机内部三个参数 Gu8SetData\_1，Gu8SetData\_2，Gu8SetData\_3 进行编辑。这三个参数分别在三个窗口下进行编辑，这三个窗口是数码管显示“1-XX”，“2-YY”，“3-ZZ”。其中，XX 代表 Gu8SetData\_1 数据，YY 代表 Gu8SetData\_2 数据，ZZ 代表 Gu8SetData\_3 数据，这三个数据的范围是从 0 到 99。K1 是窗口切换按键，每按一次，窗口会在“1-XX”，“2-YY”，“3-ZZ”三个窗口之间进行切换。K2 是数字累加按键，每按一次，显示的数字会累加 1。K3 是数字递减按键，每按一次，显示的数字会递减 1。代码如下：

```
#include "REG52.H"

#define KEY_FILTER_TIME  25
#define SCAN_TIME  1
#define VOICE_TIME  50

void TO_time();
void SystemInitial(void) ;
void Delay(unsigned long u32DelayTime) ;
void PeripheralInitial(void) ;

void KeyScan(void);
void KeyTask(void);

void VoiceScan(void);
void DisplayScan(void);
void DisplayTask(void); //上层显示的任务函数
void Wd1(void); //窗口 1 显示函数
void Wd2(void); //窗口 2 显示函数
void Wd3(void); //窗口 3 显示函数

void BeepOpen(void);
void BeepClose(void);

sbit KEY_INPUT1=P2^2;
sbit KEY_INPUT2=P2^1;
sbit KEY_INPUT3=P2^0;
```

```

sbit P1_0=P1^0;
sbit P1_1=P1^1;
sbit P1_2=P1^2;
sbit P1_3=P1^3;

sbit P3_4=P3^4;

//数码管转换表
code unsigned char Cu8DigTable[]=
{
0x3f,  //0      序号 0
0x06,  //1      序号 1
0x5b,  //2      序号 2
0x4f,  //3      序号 3
0x66,  //4      序号 4
0x6d,  //5      序号 5
0x7d,  //6      序号 6
0x07,  //7      序号 7
0x7f,  //8      序号 8
0x6f,  //9      序号 9
0x00,  //不显示 序号 10
0x40,  //横杠-   序号 11
};

volatile unsigned char vGu8ScanTimerFlag=0;
volatile unsigned int vGu16ScanTimerCnt=0;

volatile unsigned char vGu8BeepTimerFlag=0;
volatile unsigned int vGu16BeepTimerCnt=0;

unsigned char Gu8SetData_1=0; //单片机内部第 1 个可编辑的参数
unsigned char Gu8SetData_2=0; //单片机内部第 2 个可编辑的参数
unsigned char Gu8SetData_3=0; //单片机内部第 3 个可编辑的参数

unsigned char Gu8Wd=1;  //窗口选择变量。人机交互程序框架的支点。初始化开机后显示第 1 个窗口。
unsigned char Gu8WdUpdate=1;  //整屏更新变量。初始化为 1 开机后整屏更新一次显示。

volatile unsigned char vGu8Display_Righ_4=1;  //显示窗口 “1”
volatile unsigned char vGu8Display_Righ_3=11; //显示横杠 “-”
volatile unsigned char vGu8Display_Righ_2=0;  //显示十位数值 “0”
volatile unsigned char vGu8Display_Righ_1=0;  //显示个位数值 “0”

volatile unsigned char vGu8Display_Righ_Dot_4=0;
volatile unsigned char vGu8Display_Righ_Dot_3=0;

```

```

volatile unsigned char vGu8Display_Righ_Dot_2=0;
volatile unsigned char vGu8Display_Righ_Dot_1=0;

volatile unsigned char vGu8KeySec=0;

void main()
{
    SystemInitial();
    Delay(10000);
    PeripheralInitial();
    while(1)
    {
        KeyTask();    //按键的任务函数
        DisplayTask(); //数码管显示的上层任务函数
    }
}

void KeyTask(void)    //按键的任务函数
{
    if(0==vGu8KeySec)
    {
        return;
    }

    switch(vGu8KeySec)
    {
        case 1:    //窗口切换的按键
            Gu8Wd++; //窗口切换到下一个窗口
            if(Gu8Wd>3) //一共 3 个窗口。切换第 3 个窗口之后，继续返回到第 1 个窗口
            {
                Gu8Wd=1; //返回到第 1 个窗口
            }
            Gu8WdUpdate=1; //整屏更新一次显示

            vGu8BeepTimerFlag=0;
            vGu16BeepTimerCnt=VOICE_TIME; //蜂鸣器发出“滴”一声
            vGu8BeepTimerFlag=1;

            vGu8KeySec=0;
            break;

        case 2:    //累加的按键
            switch(Gu8Wd) //以窗口选择 Gu8Wd 为支点，去编辑对应的数据。又一次用到 switch 语句
            {

```

```

        case 1:    //在第 1 个窗口下编辑 Gu8SetData_1 数据
            Gu8SetData_1++;
            if(Gu8SetData_1>99) //把最大范围限定在 99
            {
                Gu8SetData_1=99;
            }
            Gu8WdUpdate=1; //整屏更新一次显示
            break;

        case 2:    //在第 2 个窗口下编辑 Gu8SetData_2 数据
            Gu8SetData_2++;
            if(Gu8SetData_2>99) //把最大范围限定在 99
            {
                Gu8SetData_2=99;
            }
            Gu8WdUpdate=1; //整屏更新一次显示
            break;

        case 3:    //在第 3 个窗口下编辑 Gu8SetData_3 数据
            Gu8SetData_3++;
            if(Gu8SetData_3>99) //把最大范围限定在 99
            {
                Gu8SetData_3=99;
            }
            Gu8WdUpdate=1; //整屏更新一次显示
            break;
    }

    vGu8BeepTimerFlag=0;
    vGu16BeepTimerCnt=VOICE_TIME; //蜂鸣器发出“滴”一声
    vGu8BeepTimerFlag=1;

    vGu8KeySec=0;
    break;

case 3:    //递减的按键
    switch(Gu8Wd) //以窗口选择 Gu8Wd 为支点，去编辑对应的数据。又一次用到 switch 语句
    {
        case 1:    //在第 1 个窗口下编辑 Gu8SetData_1 数据
            if(Gu8SetData_1>0) //把最小范围限定在 0
            {
                Gu8SetData_1--;
            }
            Gu8WdUpdate=1; //整屏更新一次显示

```

```

        break;

    case 2:    //在第 2 个窗口下编辑 Gu8SetData_2 数据
        if(Gu8SetData_2>0) //把最小范围限定在 0
        {
            Gu8SetData_2--;
        }
        Gu8WdUpdate=1; //整屏更新一次显示
        break;

    case 3:    //在第 3 个窗口下编辑 Gu8SetData_3 数据
        if(Gu8SetData_3>0) //把最小范围限定在 0
        {
            Gu8SetData_3--;
        }
        Gu8WdUpdate=1; //整屏更新一次显示
        break;
    }

    vGu8BeepTimerFlag=0;
    vGu16BeepTimerCnt=VOICE_TIME; //蜂鸣器发出“滴”一声
    vGu8BeepTimerFlag=1;

    vGu8KeySec=0;
    break;
}
}

void DisplayTask(void) //数码管显示的上层任务函数
{
    switch(Gu8Wd) //以窗口选择 Gu8Wd 为支点，去执行对应的窗口显示函数。又一次用到 switch 语句
    {
        case 1:
            Wd1(); //窗口 1 显示函数
            break;
        case 2:
            Wd2(); //窗口 2 显示函数
            break;
        case 3:
            Wd3(); //窗口 3 显示函数
            break;
    }
}
}

```

```

void Wd1(void)    //窗口 1 显示函数
{
    //需要借用的中间变量，用来拆分数数据位。
    static unsigned char Su8Temp_4, Su8Temp_3, Su8Temp_2, Su8Temp_1; //需要借用的中间变量

    if(1==Gu8WdUpdate) //如果需要整屏更新
    {
        Gu8WdUpdate=0; //及时清零，只更新一次显示即可，避免一直进来更新显示

        Su8Temp_4=1; //窗口 “1”
        Su8Temp_3=11; //横杠 “-”
        Su8Temp_2=Gu8SetData_1/10%10; //十位数值
        Su8Temp_1=Gu8SetData_1/1%10; //个位数值

        //上面先分解数据之后，再过渡需要显示的数据到底层驱动变量里，让过渡的时间越短越好
        vGu8Display_Righ_4=Su8Temp_4; //过渡需要显示的数据到底层驱动变量
        vGu8Display_Righ_3=Su8Temp_3;
        vGu8Display_Righ_2=Su8Temp_2;
        vGu8Display_Righ_1=Su8Temp_1;

        //不显示任何一个小数点
        vGu8Display_Righ_Dot_4=0;
        vGu8Display_Righ_Dot_3=0;
        vGu8Display_Righ_Dot_2=0;
        vGu8Display_Righ_Dot_1=0;
    }
}

void Wd2(void)    //窗口 2 显示函数
{
    //需要借用的中间变量，用来拆分数数据位。
    static unsigned char Su8Temp_4, Su8Temp_3, Su8Temp_2, Su8Temp_1; //需要借用的中间变量

    if(1==Gu8WdUpdate) //如果需要整屏更新
    {
        Gu8WdUpdate=0; //及时清零，只更新一次显示即可，避免一直进来更新显示

        Su8Temp_4=2; //窗口 “2”
        Su8Temp_3=11; //横杠 “-”
        Su8Temp_2=Gu8SetData_2/10%10; //十位数值
        Su8Temp_1=Gu8SetData_2/1%10; //个位数值

        //上面先分解数据之后，再过渡需要显示的数据到底层驱动变量里，让过渡的时间越短越好

```

```

        vGu8Display_Righ_4=Su8Temp_4; //过渡需要显示的数据到底层驱动变量
        vGu8Display_Righ_3=Su8Temp_3;
        vGu8Display_Righ_2=Su8Temp_2;
        vGu8Display_Righ_1=Su8Temp_1;

        //不显示任何一个小数点
        vGu8Display_Righ_Dot_4=0;
        vGu8Display_Righ_Dot_3=0;
        vGu8Display_Righ_Dot_2=0;
        vGu8Display_Righ_Dot_1=0;
    }
}

void Wd3(void) //窗口 3 显示函数
{
    //需要借用的中间变量，用来拆分数数据位。
    static unsigned char Su8Temp_4, Su8Temp_3, Su8Temp_2, Su8Temp_1; //需要借用的中间变量

    if(1==Gu8WdUpdate) //如果需要整屏更新
    {
        Gu8WdUpdate=0; //及时清零，只更新一次显示即可，避免一直进来更新显示

        Su8Temp_4=3; //窗口 “3”
        Su8Temp_3=11; //横杠 “-”
        Su8Temp_2=Gu8SetData_3/10%10; //十位数值
        Su8Temp_1=Gu8SetData_3/1%10; //个位数值

        //上面先分解数据之后，再过渡需要显示的数据到底层驱动变量里，让过渡的时间越短越好
        vGu8Display_Righ_4=Su8Temp_4; //过渡需要显示的数据到底层驱动变量
        vGu8Display_Righ_3=Su8Temp_3;
        vGu8Display_Righ_2=Su8Temp_2;
        vGu8Display_Righ_1=Su8Temp_1;

        //不显示任何一个小数点
        vGu8Display_Righ_Dot_4=0;
        vGu8Display_Righ_Dot_3=0;
        vGu8Display_Righ_Dot_2=0;
        vGu8Display_Righ_Dot_1=0;
    }
}

void KeyScan(void) //按键底层的驱动扫描函数，放在定时中断函数里
{
    static unsigned char Su8KeyLock1;

```



```
static unsigned int  Su16KeyCnt1;
static unsigned char Su8KeyLock2;
static unsigned int  Su16KeyCnt2;
    static unsigned char Su8KeyLock3;
static unsigned int  Su16KeyCnt3;

if(0!=KEY_INPUT1)
{
    Su8KeyLock1=0;
    Su16KeyCnt1=0;
}
else if(0==Su8KeyLock1)
{
    Su16KeyCnt1++;
    if(Su16KeyCnt1>=KEY_FILTER_TIME)
    {
        Su8KeyLock1=1;
        vGu8KeySec=1;
    }
}

if(0!=KEY_INPUT2)
{
    Su8KeyLock2=0;
    Su16KeyCnt2=0;
}
else if(0==Su8KeyLock2)
{
    Su16KeyCnt2++;
    if(Su16KeyCnt2>=KEY_FILTER_TIME)
    {
        Su8KeyLock2=1;
        vGu8KeySec=2;
    }
}

if(0!=KEY_INPUT3)
{
    Su8KeyLock3=0;
    Su16KeyCnt3=0;
}
else if(0==Su8KeyLock3)
{
    Su16KeyCnt3++;
```

```

        if(Su16KeyCnt3>=KEY_FILTER_TIME)
        {
            Su8KeyLock3=1;
            vGu8KeySec=3;
        }
    }
}

void DisplayScan(void)    //数码管底层的驱动扫描函数，放在定时中断函数里
{
    static unsigned char Su8GetCode;
    static unsigned char Su8ScanStep=1;

    if(0==vGu16ScanTimerCnt)
    {

        P0=0x00;
        P1_0=1;
        P1_1=1;
        P1_2=1;
        P1_3=1;

        switch(Su8ScanStep)
        {
            case 1:
                Su8GetCode=Cu8DigTable[vGu8Display_Righ_1];

                if(1==vGu8Display_Righ_Dot_1)
                {
                    Su8GetCode=Su8GetCode|0x80;
                }
                P0=Su8GetCode;
                P1_0=0;
                P1_1=1;
                P1_2=1;
                P1_3=1;
                break;

            case 2:
                Su8GetCode=Cu8DigTable[vGu8Display_Righ_2];
                if(1==vGu8Display_Righ_Dot_2)
                {

```

```

        Su8GetCode=Su8GetCode|0x80;
    }
    P0=Su8GetCode;
    P1_0=1;
    P1_1=0;
    P1_2=1;
    P1_3=1;
    break;

case 3:
    Su8GetCode=Cu8DigTable[vGu8Display_Righ_3];
    if(1==vGu8Display_Righ_Dot_3)
    {
        Su8GetCode=Su8GetCode|0x80;
    }
    P0=Su8GetCode;
    P1_0=1;
    P1_1=1;
    P1_2=0;
    P1_3=1;
    break;

case 4:
    Su8GetCode=Cu8DigTable[vGu8Display_Righ_4];
    if(1==vGu8Display_Righ_Dot_4)
    {
        Su8GetCode=Su8GetCode|0x80;
    }
    P0=Su8GetCode;
    P1_0=1;
    P1_1=1;
    P1_2=1;
    P1_3=0;
    break;
}

Su8ScanStep++;
if(Su8ScanStep>4)
{
    Su8ScanStep=1;
}

vGu8ScanTimerFlag=0;

```

```

        vGu16ScanTimerCnt=SCAN_TIME;
        vGu8ScanTimerFlag=1;
    }
}

void VoiceScan(void) //蜂鸣器的驱动函数
{
    static unsigned char Su8Lock=0;

    if(1==vGu8BeepTimerFlag&&vGu16BeepTimerCnt>0)
    {
        if(0==Su8Lock)
        {
            Su8Lock=1;
            BeepOpen();
        }
        else
        {
            vGu16BeepTimerCnt--;

            if(0==vGu16BeepTimerCnt)
            {
                Su8Lock=0;
                BeepClose();
            }
        }
    }
}

void BeepOpen(void)
{
    P3_4=0;
}

void BeepClose(void)
{
    P3_4=1;
}

void T0_time() interrupt 1

```

```

{
    VoiceScan();    //蜂鸣器的驱动函数
    KeyScan();      //按键底层的驱动扫描函数
    DisplayScan();  //数码管底层的驱动扫描函数

    if(1==vGu8ScanTimerFlag&&vGu16ScanTimerCnt>0)
    {
        vGu16ScanTimerCnt--; //递减式的软件定时器
    }

    TH0=0xfc;
    TL0=0x66;
}

```

```

void SystemInitial(void)

```

```

{
    P0=0x00;
    P1_0=1;
    P1_1=1;
    P1_2=1;
    P1_3=1;

    TMOD=0x01;
    TH0=0xfc;
    TL0=0x66;
    EA=1;
    ET0=1;
    TR0=1;
}

```

```

void Delay(unsigned long u32DelayTime)

```

```

{
    for(;u32DelayTime>0;u32DelayTime--);
}

```

```

void PeripheralInitial(void)

```

```

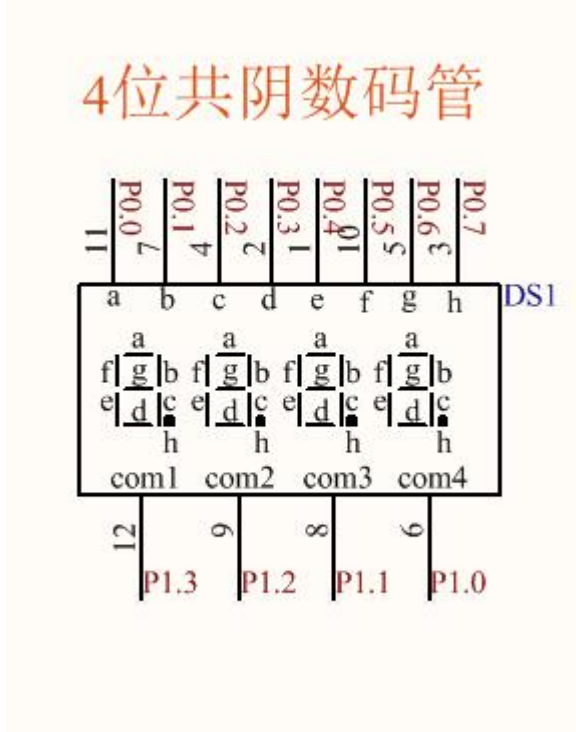
{

}

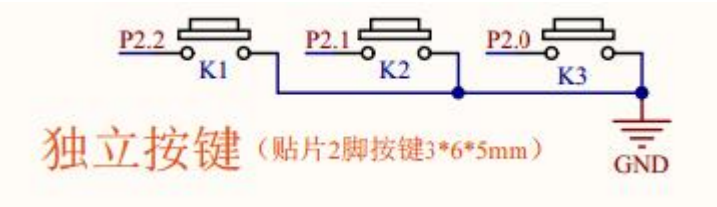
```

第一百一十八节： 按键让某位数码管闪烁跳动来设置参数。

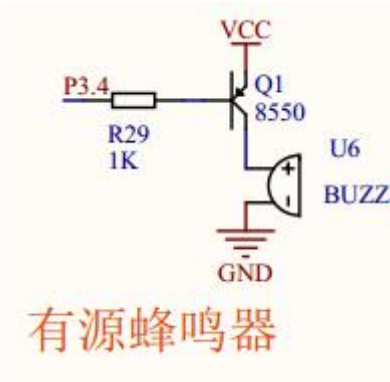
【118.1 按键让某位数码管闪烁跳动来设置参数。】



上图 118. 1. 1 数码管



上图 118. 1. 2 独立按键



上图 118. 1. 3 有源蜂鸣器

当一个窗口只有一个数据的时候，只需以“窗口”为支点，切换到某个窗口下去设置某个数据即可。但是，当某个窗口有几个数据时，就必须在以“窗口”为支点的前提下，再细分出一个二级的支点，这个二级支点就是“局部”（或者称为子菜单）。“窗口”对应一个“窗口选择”的全局变量 Gu8Wd，“局部”对应一个“局部选择”的全局变量 Gu8Part。数据需要更新显示输出到屏幕（数码管）时，有两种更新方式，一种是“整屏更新”，另一种是“局部更新”。“整屏更新”只有一个整屏的更新变量 Gu8WdUpdate，而“局部更新”有 N 个更新变量 Gu8PartUpdate\_x（Gu8PartUpdate\_1，Gu8PartUpdate\_2，Gu8PartUpdate\_3），一个窗口下有多少个数据就存在多少个局部的更新变量 Gu8PartUpdate\_x，这些局部的更新变量在不同的窗口下是可以共用的。当某个局部被选中的时候，可以有很多种表现方式，比如在液晶屏上，常见的有光标跳动，某行文字的底色变色（反显），本节例程用的数码管，当某个局部被选中的时候，用某位数码管闪烁跳动的方式。

本节小项目的程序功能，在一个窗口下，对单片机内部四个参数 Gu8SetData\_4，Gu8SetData\_3，Gu8SetData\_2，Gu8SetData\_1 进行编辑。这四个参数的范围是从 0 到 9，从左到右分别显示在四位数码管上，每一位数码管对应一个数据。比如左起第 1 位是 Gu8SetData\_4，左起第 2 位是 Gu8SetData\_3，左起第 3 位是 Gu8SetData\_2，左起第 4 位是 Gu8SetData\_1。K1 是局部选择的切换按键，每按一次，数码管从左到右，依次闪烁跳动，表示某个数据被选中。K2 是数字累加按键，每按一次，闪烁跳动的数字会累加 1。K3 是数字递减按键，每按一次，闪烁跳动的数字会递减 1。代码如下：

```
#include "REG52.H"

#define KEY_FILTER_TIME  25
#define SCAN_TIME      1
#define VOICE_TIME      50
#define BLINK_TIME      250    //数码管闪烁跳动的的时间的间隔

void T0_time();
void SystemInitial(void) ;
void Delay(unsigned long u32DelayTime) ;
void PeripheralInitial(void) ;

void KeyScan(void);
void KeyTask(void);

void VoiceScan(void);
void DisplayScan(void);
void DisplayTask(void); //上层显示的任务函数
void Wd1(void); //窗口 1 显示函数
void PartUpdate(unsigned char u8Part); //局部选择对应的某个局部变量更新显示输出

void BeepOpen(void);
void BeepClose(void);

sbit KEY_INPUT1=P2^2;
sbit KEY_INPUT2=P2^1;
sbit KEY_INPUT3=P2^0;
```

```

sbit P1_0=P1^0;
sbit P1_1=P1^1;
sbit P1_2=P1^2;
sbit P1_3=P1^3;

sbit P3_4=P3^4;

//数码管转换表
code unsigned char Cu8DigTable[]=
{
0x3f, //0      序号 0
0x06, //1      序号 1
0x5b, //2      序号 2
0x4f, //3      序号 3
0x66, //4      序号 4
0x6d, //5      序号 5
0x7d, //6      序号 6
0x07, //7      序号 7
0x7f, //8      序号 8
0x6f, //9      序号 9
0x00, //不显示 序号 10
0x40, //横杠-  序号 11
};

volatile unsigned char vGu8ScanTimerFlag=0;
volatile unsigned int vGu16ScanTimerCnt=0;

volatile unsigned char vGu8BeepTimerFlag=0;
volatile unsigned int vGu16BeepTimerCnt=0;

volatile unsigned char vGu8BlinkTimerFlag=0; //数码管闪烁跳动的定时器
volatile unsigned int vGu16BlinkTimerCnt=0;

unsigned char Gu8SetData_4=0; //单片机内部第 4 个可编辑的参数
unsigned char Gu8SetData_3=0; //单片机内部第 3 个可编辑的参数
unsigned char Gu8SetData_2=0; //单片机内部第 2 个可编辑的参数
unsigned char Gu8SetData_1=0; //单片机内部第 1 个可编辑的参数

unsigned char Gu8Wd=1; //窗口选择变量。人机交互程序框架的支点。初始化开机后显示第 1 个窗口。
unsigned char Gu8WdUpdate=1; //整屏更新变量。初始化为 1 开机后整屏更新一次显示。
unsigned char Gu8Part=0; //局部选择变量。0 代表当前窗口下没有数据被选中。
unsigned char Gu8PartUpdate_1=0; //局部 1 的更新变量,
unsigned char Gu8PartUpdate_2=0; //局部 2 的更新变量
unsigned char Gu8PartUpdate_3=0; //局部 3 的更新变量

```



```

unsigned char Gu8PartUpdate_4=0;    //局部 4 的更新变量

volatile unsigned char vGu8Display_Righ_4=0;    //左起第 1 位初始化显示数值 “0”
volatile unsigned char vGu8Display_Righ_3=0;    //左起第 2 位初始化显示数值 “0”
volatile unsigned char vGu8Display_Righ_2=0;    //左起第 3 位初始化显示数值 “0”
volatile unsigned char vGu8Display_Righ_1=0;    //左起第 4 位初始化显示数值 “0”

volatile unsigned char vGu8Display_Righ_Dot_4=0;
volatile unsigned char vGu8Display_Righ_Dot_3=0;
volatile unsigned char vGu8Display_Righ_Dot_2=0;
volatile unsigned char vGu8Display_Righ_Dot_1=0;

volatile unsigned char vGu8KeySec=0;

void main()
{
    SystemInitial();
    Delay(10000);
    PeripheralInitial();
    while(1)
    {
        KeyTask();    //按键的任务函数
        DisplayTask(); //数码管显示的上层任务函数
    }
}

void PartUpdate(unsigned char u8Part) //局部选择对应的某个局部变量更新显示输出
{
    switch(u8Part)
    {
        case 1:
            Gu8PartUpdate_1=1;
            break;
        case 2:
            Gu8PartUpdate_2=1;
            break;
        case 3:
            Gu8PartUpdate_3=1;
            break;
        case 4:
            Gu8PartUpdate_4=1;
            break;
    }
}

```

```

}

void KeyTask(void)    //按键的任务函数
{
    if(0==vGu8KeySec)
    {
        return;
    }

    switch(vGu8KeySec)
    {
        case 1:    //局部切换的按键
            switch(Gu8Wd) //在某个窗口下
            {
                case 1:    //在窗口 1 下
                    //以下之所以有两个 PartUpdate(Gu8Part)，是因为相邻的两个局部发生了变化。

                    PartUpdate(Gu8Part); //切换之前的局部进行更新。
                    Gu8Part++; //切换到下一个局部
                    if(Gu8Part>4)
                    {
                        Gu8Part=0;
                    }
                    PartUpdate(Gu8Part); //切换之后的局部进行更新。
                    break;
            }

            vGu8BeepTimerFlag=0;
            vGu16BeepTimerCnt=VOICE_TIME; //蜂鸣器发出“滴”一声
            vGu8BeepTimerFlag=1;

            vGu8KeySec=0;
            break;

        case 2:    //累加的按键
            switch(Gu8Wd) //在某个窗口下
            {
                case 1:    //在窗口 1 下
                    switch(Gu8Part) //二级支点的局部选择
                    {
                        case 1: //局部 1 被选中，代表左起第 1 位数据 Gu8SetData_4 被选中。
                            Gu8SetData_4++;
                            if(Gu8SetData_4>9)

```

```

        {
            Gu8SetData_4=9;
        }
        PartUpdate(Gu8Part); //当前局部更新显示输出到数码管
        break;

    case 2: //局部 2 被选中，代表左起第 2 位数据 Gu8SetData_3 被选中。
        Gu8SetData_3++;
        if(Gu8SetData_3>9)
        {
            Gu8SetData_3=9;
        }
        PartUpdate(Gu8Part); //当前局部更新显示输出到数码管
        break;

    case 3: //局部 3 被选中，代表左起第 3 位数据 Gu8SetData_2 被选中。
        Gu8SetData_2++;
        if(Gu8SetData_2>9)
        {
            Gu8SetData_2=9;
        }
        PartUpdate(Gu8Part); //当前局部更新显示输出到数码管
        break;

    case 4: //局部 4 被选中，代表左起第 4 位数据 Gu8SetData_1 被选中。
        Gu8SetData_1++;
        if(Gu8SetData_1>9)
        {
            Gu8SetData_1=9;
        }
        PartUpdate(Gu8Part); //当前局部更新显示输出到数码管
        break;

    }
    break;

    case 2: //在窗口 2 下（本节只用到窗口 1）
        break;
}

vGu8BeepTimerFlag=0;
vGu16BeepTimerCnt=VOICE_TIME; //蜂鸣器发出“滴”一声
vGu8BeepTimerFlag=1;

```

```

vGu8KeySec=0;
break;

case 3:    //递减的按键
switch(Gu8Wd) //在某个窗口下
{
    case 1:    //在窗口 1 下
        switch(Gu8Part) //二级支点的局部选择
        {
            case 1: //局部 1 被选中，代表左起第 1 位数据 Gu8SetData_4 被选中。
                if(Gu8SetData_4>0)
                {
                    Gu8SetData_4--;
                }
                PartUpdate(Gu8Part); //当前局部更新显示输出到数码管
                break;

            case 2: //局部 2 被选中，代表左起第 2 位数据 Gu8SetData_3 被选中。
                if(Gu8SetData_3>0)
                {
                    Gu8SetData_3--;
                }
                PartUpdate(Gu8Part); //当前局部更新显示输出到数码管
                break;

            case 3: //局部 3 被选中，代表左起第 3 位数据 Gu8SetData_2 被选中。
                if(Gu8SetData_2>0)
                {
                    Gu8SetData_2--;
                }
                PartUpdate(Gu8Part); //当前局部更新显示输出到数码管
                break;

            case 4: //局部 4 被选中，代表左起第 4 位数据 Gu8SetData_1 被选中。
                if(Gu8SetData_1>0)
                {
                    Gu8SetData_1--;
                }
                PartUpdate(Gu8Part); //当前局部更新显示输出到数码管
                break;

        }
    break;
}

```

```

        case 2:    //在窗口 2 下（本节只用到窗口 1）
            break;
    }

    vGu8BeepTimerFlag=0;
    vGu16BeepTimerCnt=VOICE_TIME; //蜂鸣器发出“滴”一声
    vGu8BeepTimerFlag=1;

    vGu8KeySec=0;
    break;
}
}

void DisplayTask(void) //数码管显示的上层任务函数
{
    switch(Gu8Wd) //以窗口选择 Gu8Wd 为支点，去执行对应的窗口显示函数。又一次用到 switch 语句
    {
        case 1:
            Wd1(); //窗口 1 显示函数
            break;
        case 2:    //窗口 2 显示选择（本节只用到窗口 1）
            break;
    }
}

void Wd1(void) //窗口 1 显示函数
{
    //需要借用的中间变量，用来拆分数数据位。
    static unsigned char Su8Temp_4, Su8Temp_3, Su8Temp_2, Su8Temp_1; //需要借用的中间变量
    static unsigned char Su8BlinkFlag=0; //两种状态的切换判断的中间变量

    if(1==Gu8WdUpdate) //如果需要整屏更新
    {
        Gu8WdUpdate=0; //及时清零，只更新一次显示即可，避免一直进来更新显示

        //不显示任何一个小数点
        vGu8Display_Righ_Dot_4=0;
        vGu8Display_Righ_Dot_3=0;
        vGu8Display_Righ_Dot_2=0;
        vGu8Display_Righ_Dot_1=0;

        Gu8PartUpdate_1=1; //局部 1 更新显示
    }
}

```

```

        Gu8PartUpdate_2=1 ;//局部 2 更新显示
        Gu8PartUpdate_3=1; //局部 3 更新显示
        Gu8PartUpdate_4=1; //局部 4 更新显示

    }

    if(1==Gu8PartUpdate_1) //局部 1 更新显示
    {
        Gu8PartUpdate_1=0; //及时清零，只更新一次显示即可，避免一直进来更新显示

        Su8Temp_4=Gu8SetData_4; //显示左起第 1 个数据 Gu8SetData_4

        //上面先分解数据之后，再过渡需要显示的数据到底层驱动变量里，让过渡的时间越短越好
        vGu8Display_Righ_4=Su8Temp_4; //过渡需要显示的数据到底层驱动变量
    }

    if(1==Gu8PartUpdate_2) //局部 2 更新显示
    {
        Gu8PartUpdate_2=0; //及时清零，只更新一次显示即可，避免一直进来更新显示

        Su8Temp_3=Gu8SetData_3; //显示左起第 2 个数据 Gu8SetData_3

        //上面先分解数据之后，再过渡需要显示的数据到底层驱动变量里，让过渡的时间越短越好
        vGu8Display_Righ_3=Su8Temp_3; //过渡需要显示的数据到底层驱动变量
    }

    if(1==Gu8PartUpdate_3) //局部 3 更新显示
    {
        Gu8PartUpdate_3=0; //及时清零，只更新一次显示即可，避免一直进来更新显示

        Su8Temp_2=Gu8SetData_2; //显示左起第 3 个数据 Gu8SetData_2

        //上面先分解数据之后，再过渡需要显示的数据到底层驱动变量里，让过渡的时间越短越好
        vGu8Display_Righ_2=Su8Temp_2; //过渡需要显示的数据到底层驱动变量
    }

    if(1==Gu8PartUpdate_4) //局部 4 更新显示
    {
        Gu8PartUpdate_4=0; //及时清零，只更新一次显示即可，避免一直进来更新显示

        Su8Temp_1=Gu8SetData_1; //显示左起第 4 个数据 Gu8SetData_1

        //上面先分解数据之后，再过渡需要显示的数据到底层驱动变量里，让过渡的时间越短越好
        vGu8Display_Righ_1=Su8Temp_1; //过渡需要显示的数据到底层驱动变量
    }

```

```

}

if (0==vGu16BlinkTimerCnt) //某位被选中的数码管跳动闪烁的定时器
{
    vGu8BlinkTimerFlag=0;
    vGu16BlinkTimerCnt=BLINK_TIME; //重设定时器的定时时间
    vGu8BlinkTimerFlag=1;

    switch(Gu8Part) //某个局部被选中，则闪烁跳动
    {
        case 1:
            if (0==Su8BlinkFlag) //两种状态的切换判断
            {
                Su8BlinkFlag=1;
                Su8Temp_4=10; //左起第 1 个显示“不显示”（10 代表不显示）
            }
            else
            {
                Su8BlinkFlag=0;
                Su8Temp_4=Gu8SetData_4; //左起第 1 个显示数据 Gu8SetData_4
            }

            break;

        case 2:
            if (0==Su8BlinkFlag) //两种状态的切换判断
            {
                Su8BlinkFlag=1;
                Su8Temp_3=10; //左起第 2 个显示“不显示”（10 代表不显示）
            }
            else
            {
                Su8BlinkFlag=0;
                Su8Temp_3=Gu8SetData_3; //左起第 2 个显示数据 Gu8SetData_3
            }

            break;

        case 3:
            if (0==Su8BlinkFlag) //两种状态的切换判断
            {
                Su8BlinkFlag=1;
                Su8Temp_2=10; //左起第 3 个显示“不显示”（10 代表不显示）
            }
    }
}

```

```

        else
        {
            Su8BlinkFlag=0;
            Su8Temp_2=Gu8SetData_2; //左起第 3 个显示数据 Gu8SetData_2
        }

        break;

    case 4:
        if (0==Su8BlinkFlag) //两种状态的切换判断
        {
            Su8BlinkFlag=1;
            Su8Temp_1=10; //左起第 3 个显示 “不显示” (10 代表不显示)
        }
        else
        {
            Su8BlinkFlag=0;
            Su8Temp_1=Gu8SetData_1; //左起第 4 个显示数据 Gu8SetData_1
        }

        break;

    default: //都没有被选中的时候
        Su8Temp_4=Gu8SetData_4; //左起第 1 个显示数据 Gu8SetData_4
        Su8Temp_3=Gu8SetData_3; //左起第 2 个显示数据 Gu8SetData_3
        Su8Temp_2=Gu8SetData_2; //左起第 3 个显示数据 Gu8SetData_2
        Su8Temp_1=Gu8SetData_1; //左起第 4 个显示数据 Gu8SetData_1
        break;
}

vGu8Display_Righ_4=Su8Temp_4; //过渡需要显示的数据到底层驱动变量
vGu8Display_Righ_3=Su8Temp_3; //过渡需要显示的数据到底层驱动变量
vGu8Display_Righ_2=Su8Temp_2; //过渡需要显示的数据到底层驱动变量
vGu8Display_Righ_1=Su8Temp_1; //过渡需要显示的数据到底层驱动变量

}

}

void KeyScan(void) //按键底层的驱动扫描函数，放在定时中断函数里
{
    static unsigned char Su8KeyLock1;
    static unsigned int Su16KeyCnt1;
    static unsigned char Su8KeyLock2;

```



```

static unsigned int  Su16KeyCnt2;
static unsigned char Su8KeyLock3;
static unsigned int  Su16KeyCnt3;

if(0!=KEY_INPUT1)
{
    Su8KeyLock1=0;
    Su16KeyCnt1=0;
}
else if(0==Su8KeyLock1)
{
    Su16KeyCnt1++;
    if(Su16KeyCnt1>=KEY_FILTER_TIME)
    {
        Su8KeyLock1=1;
        vGu8KeySec=1;
    }
}

if(0!=KEY_INPUT2)
{
    Su8KeyLock2=0;
    Su16KeyCnt2=0;
}
else if(0==Su8KeyLock2)
{
    Su16KeyCnt2++;
    if(Su16KeyCnt2>=KEY_FILTER_TIME)
    {
        Su8KeyLock2=1;
        vGu8KeySec=2;
    }
}

if(0!=KEY_INPUT3)
{
    Su8KeyLock3=0;
    Su16KeyCnt3=0;
}
else if(0==Su8KeyLock3)
{
    Su16KeyCnt3++;
    if(Su16KeyCnt3>=KEY_FILTER_TIME)
    {

```

```

        Su8KeyLock3=1;
        vGu8KeySec=3;
    }
}

void DisplayScan(void)    //数码管底层的驱动扫描函数，放在定时中断函数里
{
    static unsigned char Su8GetCode;
    static unsigned char Su8ScanStep=1;

    if(0==vGu16ScanTimerCnt)
    {

        P0=0x00;
        P1_0=1;
        P1_1=1;
        P1_2=1;
        P1_3=1;

        switch(Su8ScanStep)
        {
            case 1:
                Su8GetCode=Cu8DigTable[vGu8Display_Righ_1];

                if(1==vGu8Display_Righ_Dot_1)
                {
                    Su8GetCode=Su8GetCode|0x80;
                }
                P0=Su8GetCode;
                P1_0=0;
                P1_1=1;
                P1_2=1;
                P1_3=1;
                break;

            case 2:
                Su8GetCode=Cu8DigTable[vGu8Display_Righ_2];
                if(1==vGu8Display_Righ_Dot_2)
                {
                    Su8GetCode=Su8GetCode|0x80;
                }

```

```

        P0=Su8GetCode;
        P1_0=1;
        P1_1=0;
        P1_2=1;
        P1_3=1;
        break;

    case 3:
        Su8GetCode=Cu8DigTable[vGu8Display_Righ_3];
        if(1==vGu8Display_Righ_Dot_3)
        {
            Su8GetCode=Su8GetCode|0x80;
        }
        P0=Su8GetCode;
        P1_0=1;
        P1_1=1;
        P1_2=0;
        P1_3=1;
        break;

    case 4:
        Su8GetCode=Cu8DigTable[vGu8Display_Righ_4];
        if(1==vGu8Display_Righ_Dot_4)
        {
            Su8GetCode=Su8GetCode|0x80;
        }
        P0=Su8GetCode;
        P1_0=1;
        P1_1=1;
        P1_2=1;
        P1_3=0;
        break;

}

Su8ScanStep++;
if(Su8ScanStep>4)
{
    Su8ScanStep=1;
}

vGu8ScanTimerFlag=0;
vGu16ScanTimerCnt=SCAN_TIME;
vGu8ScanTimerFlag=1;

```

```

    }
}

void VoiceScan(void) //蜂鸣器的驱动函数
{
    static unsigned char Su8Lock=0;

    if(1==vGu8BeepTimerFlag&&vGu16BeepTimerCnt>0)
    {
        if(0==Su8Lock)
        {
            Su8Lock=1;
            BeepOpen();
        }
        else
        {

            vGu16BeepTimerCnt--;

            if(0==vGu16BeepTimerCnt)
            {
                Su8Lock=0;
                BeepClose();
            }

        }
    }
}

void BeepOpen(void)
{
    P3_4=0;
}

void BeepClose(void)
{
    P3_4=1;
}

void T0_time() interrupt 1
{
    VoiceScan(); //蜂鸣器的驱动函数
}

```

```

KeyScan();      //按键底层的驱动扫描函数
DisplayScan();  //数码管底层的驱动扫描函数

if(1==vGu8ScanTimerFlag&&vGu16ScanTimerCnt>0)
{
    vGu16ScanTimerCnt--; //递减式的软件定时器
}

if(1==vGu8BlinkTimerFlag&&vGu16BlinkTimerCnt>0) //数码管闪烁跳动的定时器
{
    vGu16BlinkTimerCnt--; //递减式的软件定时器
}

TH0=0xfc;
TL0=0x66;
}

void SystemInitial(void)
{
    P0=0x00;
    P1_0=1;
    P1_1=1;
    P1_2=1;
    P1_3=1;

    TMOD=0x01;
    TH0=0xfc;
    TL0=0x66;
    EA=1;
    ET0=1;
    TR0=1;
}

void Delay(unsigned long u32DelayTime)
{
    for(;u32DelayTime>0;u32DelayTime--);
}

void PeripheralInitial(void)
{
}

```

## 第一百一十九节： 一个完整的人机界面的程序框架的脉络。

### 【119.1 一个完整的人机界面的程序框架的脉络。】

前面两节例子告诉我们，一个完整的人机界面的程序框架包含两个要素，分别是“支点”与“更新”。“支点”包括“窗口选择”和“局部选择”，“更新”包括“整屏更新”和“局部更新”。

“支点”的作用是把显示函数与按键函数完美无缝的关联起来，两个函数同样的“支点”促使同样的“话语体系”，让“所见即所得”实时同步，确保按键操作的数据就是当前显示被选中的数据。

“静态数据”与“动态数据”的概念。被窗口显示的数据通常有两种：一种是静态数据，比如装饰门面的数据，只能显示不能更改的数据，以及图片图标这类数据；另外一种动态数据，这种数据在窗口显示上是活动的可编辑的，是需要经常修改的，往往也是系统核心的数据，需要保存或者需要跟某些关键运动密切相关的数据。比如，在前面章节中，数码管要显示三个窗口“1-XX”，“2-YY”，“3-ZZ”，其中“1-”、“2-”、“3-”是属于静态数据，它们是起“装饰”作用的。而“XX”、“YY”、“ZZ”则是动态数据，它们是可编辑的，也是单片机系统内部核心的数据。

“整屏更新”与“局部更新”的分工。“整屏更新”主要负责在切换新窗口时，把“静态数据”一次性显示到当前窗口。而“局部更新”主要负责在当前窗口下显示“动态数据”。

下面，我把一个完整的人机界面的程序框架的脉络勾勒出来，让大家有一个整体的观感，这种人机界面的程序框架放之四海而皆准，我已把它应用在各种数码管，单色液晶屏，彩屏，电脑上位机等项目上。假设某个项目中只有两个“窗口”只有两个“局部”，程序框架的脉络如下：

#### 显示部分：

```
void DisplayTask(void) //数码管显示的上层任务函数
{
    switch(Gu8Wd) //以“窗口选择” Gu8Wd 为支点
    {
        case 1:
            Wd1(); //窗口 1 显示函数
            break;
        case 2:
            Wd2(); //窗口 2 显示函数
            break;
    }
}

void Wd1(void) //窗口 1 显示函数
{
    if(1==Gu8WdUpdate) //整屏更新
    {
        Gu8WdUpdate=0; //及时清零，只更新一次显示即可，避免一直进来更新显示

        ..... //此处省略 N 行代码，用来显示静态的数据，比如图片图标，或者装饰的数据
    }
}
```

```

        //以下，“整屏更新” 必然是要把所有的“局部更新”都触发一次
        Gu8PartUpdate_1=1;  //局部 1 更新显示
        Gu8PartUpdate_2=1  ;//局部 2 更新显示
    }

    if(1==Gu8PartUpdate_1) //局部 1 更新显示
    {
        Gu8PartUpdate_1=0; //及时清零，只更新一次显示即可，避免一直进来更新显示

        .....    //此处省略 N 行代码，用来显示动态的数据。比如可编辑的数据，实时变化的数据
    }

    if(1==Gu8PartUpdate_2) //局部 2 更新显示
    {
        Gu8PartUpdate_2=0; //及时清零，只更新一次显示即可，避免一直进来更新显示

        .....    //此处省略 N 行代码，用来显示动态的数据。比如可编辑的数据，实时变化的数据
    }

    if(0==vGu16BlinkTimerCnt) //跳动的光标，或者动态闪烁的某位被选中的数据
    {
        vGu8BlinkTimerFlag=0;
        vGu16BlinkTimerCnt=BLINK_TIME; //重设定时器的定时时间
        vGu8BlinkTimerFlag=1;

        .....    //此处省略 N 行代码，用来制作跳动的光标或者某位被选中而闪烁的数据
    }
}

void Wd2(void)    //窗口 2 显示函数
{
    .....    //此处省略 N 行代码，窗口 2 显示函数的代码跟窗口 1 类似
}

```

**按键部分：**

```

void KeyTask(void)    //按键的任务函数

```

```

{
    if(0==vGu8KeySec)
    {
        return;
    }

    switch(vGu8KeySec)
    {
        case 1:    //1 号按键
            switch(Gu8Wd) //以 “窗口选择” Gu8Wd 为支点
            {
                case 1:    //在窗口 1 下
                    switch(Gu8Part) //以 “局部选择” Gu8Part 为支点
                    {
                        case 1:

                            .....    //此处省略 N 行代码

                            break;

                        case 2: //局部 2 被选中

                            .....    //此处省略 N 行代码

                            break;

                    }
                    break;

                case 2:    //在窗口 2 下
                    switch(Gu8Part) //以 “局部选择” Gu8Part 为支点
                    {
                        case 1:

                            .....    //此处省略 N 行代码

                            break;

                        case 2: //局部 2 被选中

                            .....    //此处省略 N 行代码

                            break;

                    }
                    break;
            }
    }
}

```



```
    }

    vGu8BeepTimerFlag=0;
    vGu16BeepTimerCnt=VOICE_TIME; //蜂鸣器发出“滴”一声
    vGu8BeepTimerFlag=1;

    vGu8KeySec=0;
    break;

case 2: //2 号按键

    ..... //此处省略 N 行代码，跟 1 号按键的代码类似

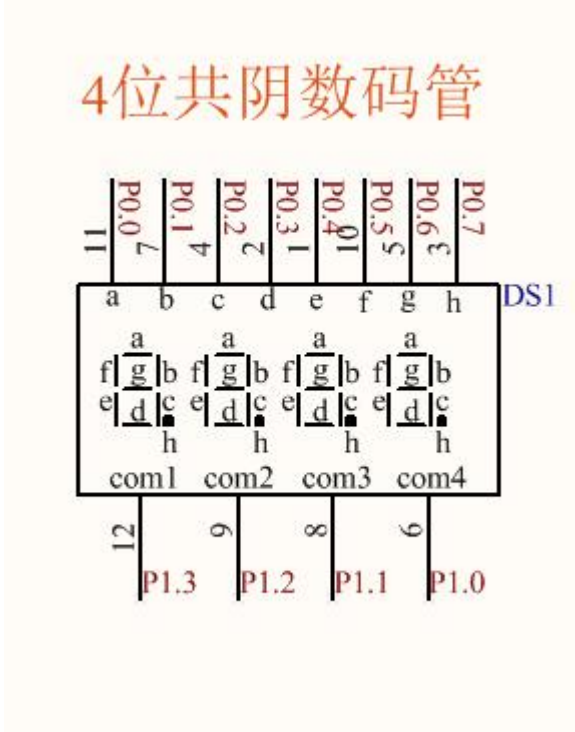
    vGu8BeepTimerFlag=0;
    vGu16BeepTimerCnt=VOICE_TIME; //蜂鸣器发出“滴”一声
    vGu8BeepTimerFlag=1;

    vGu8KeySec=0;
    break;

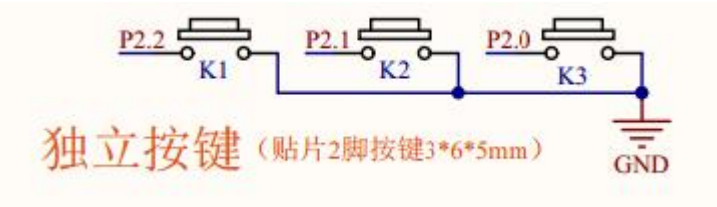
}
}
```

第一百二十节： 按键切换窗口切换局部来设置参数。

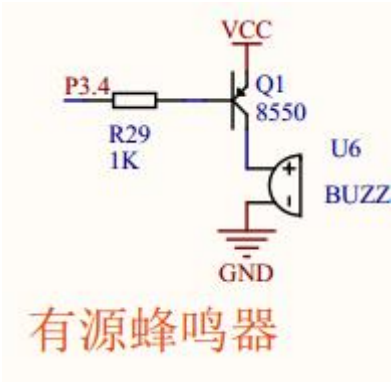
【120.1 按键切换窗口切换局部来设置参数。】



上图 120. 1. 1 数码管



上图 120. 1. 2 独立按键



上图 120. 1. 3 有源蜂鸣器

为了更好地理解上一节提出的人机界面程序框架的脉络，本节程序恰好包含了整屏更新与局部更新的应用，同时也引入了一个新的知识点：在人机界面的程序框架中，常常会遇到需要以“位”来编辑某个数据的情况，这种情况实际上是先把“待编辑数据”分解成几个“位”中间临时个体，然后显示并且编辑这些“位”中间临时个体，编辑结束后，再把这些“位”中间临时个体合并成一个完整的数据赋值给“待编辑数据”。

本节程序功能如下：

(1) 有 3 个窗口 1-XX，2-YY，3-ZZ，其中 XX，YY，ZZ 分别代表 3 个可编辑的数据 Gu8SetDate\_1，Gu8SetDate\_2，Gu8SetDate\_3。数据范围是从 0 到 99。

(2) K1 按键。含“短按”与“长按”复合双功能。当数码管“没有闪烁”时，“短按”K1 按键可以切换窗口，而“长按”K1 按键会使数码管从“没有闪烁”进入到“闪烁模式”。当数码管处于“闪烁模式”时，“短按”K1 可以使数码管在十位和个位之间切换“闪烁”的“局部位”，而“长按”K1 表示更改完毕当前窗口数据并从“闪烁模式”退出到“没有闪烁”。

(3) K2 按键。当数码管处于“闪烁模式”时，每按一次 K2 按键就可以使当前闪烁的某位数码管“递增 1”。

(4) K3 按键。当数码管处于“闪烁模式”时，每按一次 K2 按键就可以使当前闪烁的某位数码管“递减 1”。

上述功能，在窗口切换和退出“闪烁模式”时用到整屏更新，在闪烁的某位数码管切换“局部”时用到局部更新。代码如下：

```
#include "REG52.H"

#define KEY_FILTER_TIME 25    //按键的“短按”兼“滤波”的“稳定时间”
#define KEY_LONG_TIME 500    //按键的“长按”兼“滤波”的“稳定时间”

#define SCAN_TIME 1
#define VOICE_TIME 50
#define BLINK_TIME 250    //数码管闪烁跳动的时间的间隔

void TO_time();
void SystemInitial(void) ;
void Delay(unsigned long u32DelayTime) ;
void PeripheralInitial(void) ;

void KeyScan(void);
void KeyTask(void);

void VoiceScan(void);
void DisplayScan(void);
void DisplayTask(void); //上层显示的任务函数
void Wd1(void); //窗口 1 显示函数
void Wd2(void); //窗口 2 显示函数
void Wd3(void); //窗口 3 显示函数

void PartUpdate(unsigned char u8Part); //局部选择对应的某个局部变量更新显示输出
```

```

void BeepOpen(void);
void BeepClose(void);

sbit KEY_INPUT1=P2^2;
sbit KEY_INPUT2=P2^1;
sbit KEY_INPUT3=P2^0;

sbit P1_0=P1^0;
sbit P1_1=P1^1;
sbit P1_2=P1^2;
sbit P1_3=P1^3;

sbit P3_4=P3^4;

//数码管转换表
code unsigned char Cu8DigTable[]=
{
0x3f, //0      序号 0
0x06, //1      序号 1
0x5b, //2      序号 2
0x4f, //3      序号 3
0x66, //4      序号 4
0x6d, //5      序号 5
0x7d, //6      序号 6
0x07, //7      序号 7
0x7f, //8      序号 8
0x6f, //9      序号 9
0x00, //不显示 序号 10
0x40, //横杠-  序号 11
};

volatile unsigned char vGu8ScanTimerFlag=0;
volatile unsigned int vGu16ScanTimerCnt=0;

volatile unsigned char vGu8BeepTimerFlag=0;
volatile unsigned int vGu16BeepTimerCnt=0;

volatile unsigned char vGu8BlinkTimerFlag=0; //数码管闪烁跳动的定时器
volatile unsigned int vGu16BlinkTimerCnt=0;

unsigned char Gu8SetData_3=0; //单片机内部第 3 个可编辑的参数, 在窗口 3
unsigned char Gu8SetData_2=0; //单片机内部第 2 个可编辑的参数, 在窗口 2
unsigned char Gu8SetData_1=0; //单片机内部第 1 个可编辑的参数, 在窗口 1

```

```

/* 注释一：
*      在人机界面的程序框架中，常常会遇到需要以“位”来编辑某个数据的情况，这种情况
*      实际上是先把“待编辑数据”分解成几个“位”临时中间个体，然后显示并且编辑这些“位”
*      临时中间个体，编辑结束后，再把这些“位”临时中间个体合并成一个完整的数据赋值给
*      “待编辑数据”。以下 Gu8EditData_2 和 Gu8EditData_1 就是“位”临时中间个体的中间变量。
*/

unsigned char Gu8EditData_2=0; //对应显示左起第 3 位数码管的“位”数据，是中间变量。
unsigned char Gu8EditData_1=0; //对应显示左起第 4 位数码管的“位”数据，是中间变量。

unsigned char Gu8Wd=1; //窗口选择变量。人机交互程序框架的支点。初始化开机后显示第 1 个窗口。
unsigned char Gu8WdUpdate=1; //整屏更新变量。初始化为 1 开机后整屏更新一次显示。
unsigned char Gu8Part=0; //局部选择变量。0 代表当前窗口下没有数据被选中。
unsigned char Gu8PartUpdate_1=0; //局部 1 的更新变量，
unsigned char Gu8PartUpdate_2=0; //局部 2 的更新变量

volatile unsigned char vGu8Display_Righ_4=1; //左起第 1 位初始化显示窗口“1”
volatile unsigned char vGu8Display_Righ_3=11; //左起第 2 位初始化显示横杠“-”
volatile unsigned char vGu8Display_Righ_2=0; //左起第 3 位初始化显示数值“0”
volatile unsigned char vGu8Display_Righ_1=0; //左起第 4 位初始化显示数值“0”

//不显示小数点
volatile unsigned char vGu8Display_Righ_Dot_4=0;
volatile unsigned char vGu8Display_Righ_Dot_3=0;
volatile unsigned char vGu8Display_Righ_Dot_2=0;
volatile unsigned char vGu8Display_Righ_Dot_1=0;

volatile unsigned char vGu8KeySec=0;

void main()
{
    SystemInitial();
    Delay(10000);
    PeripheralInitial();
    while(1)
    {
        KeyTask(); //按键的任务函数
        DisplayTask(); //数码管显示的上层任务函数
    }
}

void PartUpdate(unsigned char u8Part) //局部选择对应的某个局部变量更新显示输出
{

```

```

switch(u8Part)
{
    case 1:
        Gu8PartUpdate_1=1;
        break;
    case 2:
        Gu8PartUpdate_2=1;
        break;
}
}

void KeyTask(void)    //按键的任务函数
{
    if(0==vGu8KeySec)
    {
        return;
    }

    switch(vGu8KeySec)
    {
        case 1:        //K1 按键的“短按”，具有“切换窗口”和“切换局部”的双功能。
            if(0==Gu8Part) //处于“没有闪烁”的时候，是“切换窗口”
            {
                switch(Gu8Wd) //在某个窗口下
                {
                    case 1:        //在窗口 1 下
                        Gu8Wd=2; //切换到窗口 2
                        Gu8EditData_2=Gu8SetData_2/10%10; //“待编辑数据”分解成中间个体
                        Gu8EditData_1=Gu8SetData_2/1%10; //“待编辑数据”分解成中间个体
                        Gu8WdUpdate=1; //整屏更新
                        break;

                    case 2:        //在窗口 2 下
                        Gu8Wd=3; //切换到窗口 3
                        Gu8EditData_2=Gu8SetData_3/10%10; //“待编辑数据”分解成中间个体
                        Gu8EditData_1=Gu8SetData_3/1%10; //“待编辑数据”分解成中间个体
                        Gu8WdUpdate=1; //整屏更新
                        break;

                    case 3:        //在窗口 3 下
                        Gu8Wd=1; //切换到窗口 1
                        Gu8EditData_2=Gu8SetData_1/10%10; //“待编辑数据”分解成中间个体
                        Gu8EditData_1=Gu8SetData_1/1%10; //“待编辑数据”分解成中间个体

```

```

        Gu8WdUpdate=1; //整屏更新
        break;

    }
}
else //处于“闪烁模式”的时候，是“切换局部”
{
    PartUpdate(Gu8Part); //切换之前的局部进行更新。
    Gu8Part++; //切换局部
    if(Gu8Part>2)
    {
        Gu8Part=1;
    }
    PartUpdate(Gu8Part); //切换之后的局部进行更新。
}

vGu8BeepTimerFlag=0;
vGu16BeepTimerCnt=VOICE_TIME; //蜂鸣器发出“滴”一声
vGu8BeepTimerFlag=1;

vGu8KeySec=0;
break;

case 2: //递增按键 K2
    switch(Gu8Wd) //在某个窗口下
    {
        case 1: //在窗口 1 下
        case 2: //在窗口 2 下，窗口 2 与窗口 1 的代码完全一模一样，因此可以这样共享
        case 3: //在窗口 3 下，窗口 3 与窗口 1 的代码完全一模一样，因此可以这样共享
            switch(Gu8Part) //二级支点的局部选择
            {
                case 1: //局部 1 被选中，代表左起第 3 位数码管被选中。
                    Gu8EditData_2++; //编辑“十位”个体的中间变量
                    if(Gu8EditData_2>9)
                    {
                        Gu8EditData_2=9;
                    }
                    PartUpdate(Gu8Part); //当前局部更新显示输出到数码管
                    break;

                case 2: //局部 2 被选中，代表左起第 4 位数码管被选中。
                    Gu8EditData_1++; //编辑“个位”个体的中间变量
                    if(Gu8EditData_1>9)
                    {

```

```

        Gu8EditData_1=9;
    }
    PartUpdate(Gu8Part); //当前局部更新显示输出到数码管
    break;
}
break;
}

vGu8BeepTimerFlag=0;
vGu16BeepTimerCnt=VOICE_TIME; //蜂鸣器发出“滴”一声
vGu8BeepTimerFlag=1;

vGu8KeySec=0;
break;

case 3: //递减按键 K3
    switch(Gu8Wd) //在某个窗口下
    {
        case 1: //在窗口 1 下
        case 2: //在窗口 2 下，窗口 2 与窗口 1 的代码完全一模一样，因此可以这样共享
        case 3: //在窗口 3 下，窗口 3 与窗口 1 的代码完全一模一样，因此可以这样共享
            switch(Gu8Part) //二级支点的局部选择
            {
                case 1: //局部 1 被选中，代表左起第 3 位数码管被选中。
                    if(Gu8EditData_2>0)
                    {
                        Gu8EditData_2--; //编辑“十位”个体的中间变量
                    }
                    PartUpdate(Gu8Part); //当前局部更新显示输出到数码管
                    break;

                case 2: //局部 2 被选中，代表左起第 4 位数码管被选中。
                    if(Gu8EditData_1>0)
                    {
                        Gu8EditData_1--; //编辑“个位”个体的中间变量
                    }
                    PartUpdate(Gu8Part); //当前局部更新显示输出到数码管
                    break;
            }
        break;
    }

    vGu8BeepTimerFlag=0;
    vGu16BeepTimerCnt=VOICE_TIME; //蜂鸣器发出“滴”一声

```



```

vGu8BeepTimerFlag=1;

vGu8KeySec=0;
break;

case 4:    //K1 按键的“长按”，具有进入和退出“闪烁模式”的功能。“退出”隐含“确定”

switch(Gu8Wd) //在某个窗口下
{
    case 1:    //在窗口 1 下
        if(0==Gu8Part) //处于“没有闪烁”的时候，将进入“闪烁模式”
        {
            Gu8EditData_2=Gu8SetData_1/10%10; //先把“待编辑数据”分解成中间个体
            Gu8EditData_1=Gu8SetData_1/1%10;    //先把“待编辑数据”分解成中间个体
            Gu8Part=1; //进入“闪烁模式”，从“局部 1”开始闪烁
        }
        else //处于“闪烁模式”的时候，将退出到“没有闪烁”，隐含“确定”功能
        {
            Gu8SetData_1=Gu8EditData_2*10+Gu8EditData_1; //把个体合并还原成数据
            Gu8Part=0; //退出“闪烁模式”
            Gu8WdUpdate=1; //整屏更新
        }
        break;

    case 2:    //在窗口 2 下
        if(0==Gu8Part) //处于“没有闪烁”的时候，将进入“闪烁模式”
        {
            Gu8EditData_2=Gu8SetData_2/10%10; //先把“待编辑数据”分解成中间个体
            Gu8EditData_1=Gu8SetData_2/1%10;    //先把“待编辑数据”分解成中间个体
            Gu8Part=1; //进入“闪烁模式”，从“局部 1”开始闪烁
        }
        else //处于“闪烁模式”的时候，将退出到“没有闪烁”，隐含“确定”功能
        {
            Gu8SetData_2=Gu8EditData_2*10+Gu8EditData_1; //把个体合并还原成数据
            Gu8Part=0; //退出“闪烁模式”
            Gu8WdUpdate=1; //整屏更新
        }
        break;

    case 3:    //在窗口 3 下
        if(0==Gu8Part) //处于“没有闪烁”的时候，将进入“闪烁模式”
        {
            Gu8EditData_2=Gu8SetData_3/10%10; //先把“待编辑数据”分解成中间个体
            Gu8EditData_1=Gu8SetData_3/1%10;    //先把“待编辑数据”分解成中间个体

```

```

        Gu8Part=1; //进入“闪烁模式”，从“局部1”开始闪烁
    }
    else //处于“闪烁模式”的时候，将退出到“没有闪烁”，隐含“确定”功能
    {
        Gu8SetData_3=Gu8EditData_2*10+Gu8EditData_1; //把个体合并还原成数据
        Gu8Part=0; //退出“闪烁模式”
        Gu8WdUpdate=1; //整屏更新
    }
    break;

}

vGu8BeepTimerFlag=0;
vGu16BeepTimerCnt=VOICE_TIME; //蜂鸣器发出“滴”一声
vGu8BeepTimerFlag=1;

vGu8KeySec=0;
break;

}
}

void DisplayTask(void) //数码管显示的上层任务函数
{
    switch(Gu8Wd) //以窗口选择 Gu8Wd 为支点，去执行对应的窗口显示函数。又一次用到 switch 语句
    {
        case 1:
            Wd1(); //窗口 1 显示函数
            break;
        case 2:
            Wd2(); //窗口 2 显示函数
            break;
        case 3:
            Wd3(); //窗口 3 显示函数
            break;
    }
}

void Wd1(void) //窗口 1 显示函数
{
    //需要借用的中间变量，用来拆分数据位。
    static unsigned char Su8Temp_4, Su8Temp_3, Su8Temp_2, Su8Temp_1; //需要借用的中间变量
    static unsigned char Su8BlinkFlag=0; //两种状态的切换判断的中间变量

```

```

if(1==Gu8WdUpdate) //如果需要整屏更新
{
    Gu8WdUpdate=0; //及时清零，只更新一次显示即可，避免一直进来更新显示

    Su8Temp_4=1; //左起第1位数码管，显示窗口“1”，属于静态数据，起“装饰”作用。
    Su8Temp_3=11; //左起第2位数码管，显示横杠“-”，属于静态数据，起“装饰”作用。

    vGu8Display_Righ_4=Su8Temp_4; //过渡需要显示的数据到底层驱动变量
    vGu8Display_Righ_3=Su8Temp_3; //过渡需要显示的数据到底层驱动变量

    //不显示任何一个小数点，属于静态数据，起“装饰”作用，切换窗口后只扫描一次的代码。
    vGu8Display_Righ_Dot_4=0;
    vGu8Display_Righ_Dot_3=0;
    vGu8Display_Righ_Dot_2=0;
    vGu8Display_Righ_Dot_1=0;

    Gu8PartUpdate_1=1; //局部1更新显示
    Gu8PartUpdate_2=1; //局部2更新显示
}

if(1==Gu8PartUpdate_1) //局部1更新显示
{
    Gu8PartUpdate_1=0; //及时清零，只更新一次显示即可，避免一直进来更新显示

    Su8Temp_2=Gu8EditData_2; //显示“十位”的临时中间个体，属于动态数据。

    vGu8Display_Righ_2=Su8Temp_2; //过渡需要显示的数据到底层驱动变量
}

if(1==Gu8PartUpdate_2) //局部2更新显示
{
    Gu8PartUpdate_2=0; //及时清零，只更新一次显示即可，避免一直进来更新显示

    Su8Temp_1=Gu8EditData_1; //显示“个位”的临时中间个体，属于动态数据。

    vGu8Display_Righ_1=Su8Temp_1; //过渡需要显示的数据到底层驱动变量
}

if(0==vGu16BlinkTimerCnt) //某位被选中的数码管跳动闪烁的定时器
{
    vGu8BlinkTimerFlag=0;
}

```

```

vGu16BlinkTimerCnt=BLINK_TIME; //重设定时器的定时时间
vGu8BlinkTimerFlag=1;

switch(Gu8Part) //某个局部被选中，则闪烁跳动
{
    case 1:
        if(0==Su8BlinkFlag) //两种状态的切换判断
        {
            Su8BlinkFlag=1;
            Su8Temp_2=10; //左起第3个显示“不显示”（10代表不显示）
        }
        else
        {
            Su8BlinkFlag=0;
            Su8Temp_2=Gu8EditData_2; //显示“十位”的临时中间个体，属于动态数据。
        }

        break;

    case 2:
        if(0==Su8BlinkFlag) //两种状态的切换判断
        {
            Su8BlinkFlag=1;
            Su8Temp_1=10; //左起第4个显示“不显示”（10代表不显示）
        }
        else
        {
            Su8BlinkFlag=0;
            Su8Temp_1=Gu8EditData_1; //显示“个位”的临时中间个体，属于动态数据。
        }

        break;

    default: //都没有被选中的时候
        Su8Temp_2=Gu8EditData_2; //显示“十位”的临时中间个体，属于动态数据。
        Su8Temp_1=Gu8EditData_1; //显示“个位”的临时中间个体，属于动态数据。
        break;
}

vGu8Display_Righ_2=Su8Temp_2; //过渡需要显示的数据到底层驱动变量
vGu8Display_Righ_1=Su8Temp_1; //过渡需要显示的数据到底层驱动变量

}
}

```

```

void Wd2(void)    //窗口 2 显示函数
{
    //需要借用的中间变量，用来拆分数数据位。
    static unsigned char Su8Temp_4, Su8Temp_3, Su8Temp_2, Su8Temp_1; //需要借用的中间变量
    static unsigned char Su8BlinkFlag=0; //两种状态的切换判断的中间变量

    if(1==Gu8WdUpdate) //如果需要整屏更新
    {
        Gu8WdUpdate=0; //及时清零，只更新一次显示即可，避免一直进来更新显示

        Su8Temp_4=2;    //左起第 1 位数码管，显示窗口“2”，属于静态数据，起“装饰”作用。
        Su8Temp_3=11;   //左起第 2 位数码管，显示横杠“-”，属于静态数据，起“装饰”作用。

        vGu8Display_Righ_4=Su8Temp_4; //过渡需要显示的数据到底层驱动变量
        vGu8Display_Righ_3=Su8Temp_3; //过渡需要显示的数据到底层驱动变量

        //不显示任何一个小数点，属于静态数据，起“装饰”作用，切换窗口后只扫描一次的代码。
        vGu8Display_Righ_Dot_4=0;
        vGu8Display_Righ_Dot_3=0;
        vGu8Display_Righ_Dot_2=0;
        vGu8Display_Righ_Dot_1=0;

        Gu8PartUpdate_1=1; //局部 1 更新显示
        Gu8PartUpdate_2=1 ;//局部 2 更新显示
    }

    if(1==Gu8PartUpdate_1) //局部 1 更新显示
    {
        Gu8PartUpdate_1=0; //及时清零，只更新一次显示即可，避免一直进来更新显示

        Su8Temp_2=Gu8EditData_2; //显示“十位”的临时中间个体，属于动态数据。

        vGu8Display_Righ_2=Su8Temp_2; //过渡需要显示的数据到底层驱动变量
    }

    if(1==Gu8PartUpdate_2) //局部 2 更新显示
    {
        Gu8PartUpdate_2=0; //及时清零，只更新一次显示即可，避免一直进来更新显示

        Su8Temp_1=Gu8EditData_1; //显示“个位”的临时中间个体，属于动态数据。

        vGu8Display_Righ_1=Su8Temp_1; //过渡需要显示的数据到底层驱动变量
    }
}

```

```

}

if(0==vGu16BlinkTimerCnt)  //某位被选中的数码管跳动闪烁的定时器
{
    vGu8BlinkTimerFlag=0;
    vGu16BlinkTimerCnt=BLINK_TIME;  //重设定时器的定时时间
    vGu8BlinkTimerFlag=1;

    switch(Gu8Part)  //某个局部被选中，则闪烁跳动
    {
        case 1:
            if(0==Su8BlinkFlag)  //两种状态的切换判断
            {
                Su8BlinkFlag=1;
                Su8Temp_2=10;  //左起第 3 个显示“不显示”（10 代表不显示）
            }
            else
            {
                Su8BlinkFlag=0;
                Su8Temp_2=Gu8EditData_2;  //显示“十位”的临时中间个体，属于动态数据。
            }

            break;

        case 2:
            if(0==Su8BlinkFlag)  //两种状态的切换判断
            {
                Su8BlinkFlag=1;
                Su8Temp_1=10;  //左起第 4 个显示“不显示”（10 代表不显示）
            }
            else
            {
                Su8BlinkFlag=0;
                Su8Temp_1=Gu8EditData_1;  //显示“个位”的临时中间个体，属于动态数据。
            }

            break;

        default:  //都没有被选中的时候
            Su8Temp_2=Gu8EditData_2;  //显示“十位”的临时中间个体，属于动态数据。
            Su8Temp_1=Gu8EditData_1;  //显示“个位”的临时中间个体，属于动态数据。
            break;
    }
}

```

```

        vGu8Display_Righ_2=Su8Temp_2; //过渡需要显示的数据到底层驱动变量
        vGu8Display_Righ_1=Su8Temp_1; //过渡需要显示的数据到底层驱动变量

    }
}

void Wd3(void) //窗口 3 显示函数
{
    //需要借用的中间变量，用来拆分数据位。
    static unsigned char Su8Temp_4, Su8Temp_3, Su8Temp_2, Su8Temp_1; //需要借用的中间变量
    static unsigned char Su8BlinkFlag=0; //两种状态的切换判断的中间变量

    if(1==Gu8WdUpdate) //如果需要整屏更新
    {
        Gu8WdUpdate=0; //及时清零，只更新一次显示即可，避免一直进来更新显示

        Su8Temp_4=3; //左起第 1 位数码管，显示窗口“3”，属于静态数据，起“装饰”作用。
        Su8Temp_3=11; //左起第 2 位数码管，显示横杠“-”，属于静态数据，起“装饰”作用。

        vGu8Display_Righ_4=Su8Temp_4; //过渡需要显示的数据到底层驱动变量
        vGu8Display_Righ_3=Su8Temp_3; //过渡需要显示的数据到底层驱动变量

        //不显示任何一个小数点，属于静态数据，起“装饰”作用，切换窗口后只扫描一次的代码。
        vGu8Display_Righ_Dot_4=0;
        vGu8Display_Righ_Dot_3=0;
        vGu8Display_Righ_Dot_2=0;
        vGu8Display_Righ_Dot_1=0;

        Gu8PartUpdate_1=1; //局部 1 更新显示
        Gu8PartUpdate_2=1 ;//局部 2 更新显示
    }

    if(1==Gu8PartUpdate_1) //局部 1 更新显示
    {
        Gu8PartUpdate_1=0; //及时清零，只更新一次显示即可，避免一直进来更新显示

        Su8Temp_2=Gu8EditData_2; //显示“十位”的临时中间个体，属于动态数据。

        vGu8Display_Righ_2=Su8Temp_2; //过渡需要显示的数据到底层驱动变量
    }

    if(1==Gu8PartUpdate_2) //局部 2 更新显示

```

```

{
    Gu8PartUpdate_2=0; //及时清零，只更新一次显示即可，避免一直进来更新显示

    Su8Temp_1=Gu8EditData_1; //显示“个位”的临时中间个体，属于动态数据。

    vGu8Display_Righ_1=Su8Temp_1; //过渡需要显示的数据到底层驱动变量
}

if(0==vGu16BlinkTimerCnt) //某位被选中的数码管跳动闪烁的定时器
{
    vGu8BlinkTimerFlag=0;
    vGu16BlinkTimerCnt=BLINK_TIME; //重设定时器的定时时间
    vGu8BlinkTimerFlag=1;

    switch(Gu8Part) //某个局部被选中，则闪烁跳动
    {
        case 1:
            if(0==Su8BlinkFlag) //两种状态的切换判断
            {
                Su8BlinkFlag=1;
                Su8Temp_2=10; //左起第3个显示“不显示”（10代表不显示）
            }
            else
            {
                Su8BlinkFlag=0;
                Su8Temp_2=Gu8EditData_2; //显示“十位”的临时中间个体，属于动态数据。
            }

            break;

        case 2:
            if(0==Su8BlinkFlag) //两种状态的切换判断
            {
                Su8BlinkFlag=1;
                Su8Temp_1=10; //左起第4个显示“不显示”（10代表不显示）
            }
            else
            {
                Su8BlinkFlag=0;
                Su8Temp_1=Gu8EditData_1; //显示“个位”的临时中间个体，属于动态数据。
            }

            break;
    }
}

```



```

        default:    //都没有被选中的时候
            Su8Temp_2=Gu8EditData_2; //显示“十位”的临时中间个体，属于动态数据。
            Su8Temp_1=Gu8EditData_1; //显示“个位”的临时中间个体，属于动态数据。
            break;
    }

    vGu8Display_Righ_2=Su8Temp_2; //过渡需要显示的数据到底层驱动变量
    vGu8Display_Righ_1=Su8Temp_1; //过渡需要显示的数据到底层驱动变量
}
}

void KeyScan(void) //按键底层的驱动扫描函数，放在定时中断函数里
{
    static unsigned char Su8KeyShortFlag=0; //按键“短按”触发的标志
    static unsigned char Su8KeyLock1;
    static unsigned int  Su16KeyCnt1;
    static unsigned char Su8KeyLock2;
    static unsigned int  Su16KeyCnt2;
    static unsigned char Su8KeyLock3;
    static unsigned int  Su16KeyCnt3;

    //需要详细分析以下这段“短按”与“长按”代码的朋友，请参考第 96 节。
    if(0!=KEY_INPUT1)
    {
        Su8KeyLock1=0;
        Su16KeyCnt1=0;
        if(1==Su8KeyShortFlag)
        {
            Su8KeyShortFlag=0;
            vGu8KeySec=1; //触发 K1 的“短按”
        }
    }
    else if(0==Su8KeyLock1)
    {
        Su16KeyCnt1++;

        if(Su16KeyCnt1>=KEY_FILTER_TIME)
        {
            Su8KeyShortFlag=1;
        }
    }
}

```

```

    if (Su16KeyCnt1>=KEY_LONG_TIME)
    {
        Su8KeyLock1=1;
        Su8KeyShortFlag=0;
        vGu8KeySec=4; //触发 K1 的“长按”
    }
}

if (0!=KEY_INPUT2)
{
    Su8KeyLock2=0;
    Su16KeyCnt2=0;
}
else if (0==Su8KeyLock2)
{
    Su16KeyCnt2++;
    if (Su16KeyCnt2>=KEY_FILTER_TIME)
    {
        Su8KeyLock2=1;
        vGu8KeySec=2;
    }
}

if (0!=KEY_INPUT3)
{
    Su8KeyLock3=0;
    Su16KeyCnt3=0;
}
else if (0==Su8KeyLock3)
{
    Su16KeyCnt3++;
    if (Su16KeyCnt3>=KEY_FILTER_TIME)
    {
        Su8KeyLock3=1;
        vGu8KeySec=3;
    }
}
}

void DisplayScan(void)    //数码管底层的驱动扫描函数，放在定时中断函数里
{
    static unsigned char Su8GetCode;
    static unsigned char Su8ScanStep=1;

```

```

if(0==vGu16ScanTimerCnt)
{

    P0=0x00;
    P1_0=1;
    P1_1=1;
    P1_2=1;
    P1_3=1;

    switch(Su8ScanStep)
    {
        case 1:
            Su8GetCode=Cu8DigTable[vGu8Display_Righ_1];

            if(1==vGu8Display_Righ_Dot_1)
            {
                Su8GetCode=Su8GetCode|0x80;
            }
            P0=Su8GetCode;
            P1_0=0;
            P1_1=1;
            P1_2=1;
            P1_3=1;
            break;

        case 2:
            Su8GetCode=Cu8DigTable[vGu8Display_Righ_2];
            if(1==vGu8Display_Righ_Dot_2)
            {
                Su8GetCode=Su8GetCode|0x80;
            }
            P0=Su8GetCode;
            P1_0=1;
            P1_1=0;
            P1_2=1;
            P1_3=1;
            break;

        case 3:
            Su8GetCode=Cu8DigTable[vGu8Display_Righ_3];
            if(1==vGu8Display_Righ_Dot_3)
            {

```

```

        Su8GetCode=Su8GetCode|0x80;
    }
    P0=Su8GetCode;
    P1_0=1;
    P1_1=1;
    P1_2=0;
    P1_3=1;
    break;

case 4:
    Su8GetCode=Cu8DigTable[vGu8Display_Righ_4];
    if(1==vGu8Display_Righ_Dot_4)
    {
        Su8GetCode=Su8GetCode|0x80;
    }
    P0=Su8GetCode;
    P1_0=1;
    P1_1=1;
    P1_2=1;
    P1_3=0;
    break;

}

Su8ScanStep++;
if(Su8ScanStep>4)
{
    Su8ScanStep=1;
}

vGu8ScanTimerFlag=0;
vGu16ScanTimerCnt=SCAN_TIME;
vGu8ScanTimerFlag=1;
}
}

```

void VoiceScan(void) //蜂鸣器的驱动函数

```

{

    static unsigned char Su8Lock=0;

    if(1==vGu8BeepTimerFlag&&vGu16BeepTimerCnt>0)
    {

```

```

        if (0==Su8Lock)
        {
            Su8Lock=1;
            BeepOpen();
        }
    else
    {

        vGu16BeepTimerCnt--;

        if (0==vGu16BeepTimerCnt)
        {
            Su8Lock=0;
            BeepClose();
        }

    }
}

void BeepOpen(void)
{
    P3_4=0;
}

void BeepClose(void)
{
    P3_4=1;
}

void T0_time() interrupt 1
{
    VoiceScan();    //蜂鸣器的驱动函数
    KeyScan();      //按键底层的驱动扫描函数
    DisplayScan();  //数码管底层的驱动扫描函数

    if (1==vGu8ScanTimerFlag&&vGu16ScanTimerCnt>0)
    {
        vGu16ScanTimerCnt--; //递减式的软件定时器
    }

    if (1==vGu8BlinkTimerFlag&&vGu16BlinkTimerCnt>0) //数码管闪烁跳动的定时器
    {
        vGu16BlinkTimerCnt--; //递减式的软件定时器
    }
}

```

```

    }

    TH0=0xfc;
    TL0=0x66;
}

void SystemInitial(void)
{
    P0=0x00;
    P1_0=1;
    P1_1=1;
    P1_2=1;
    P1_3=1;

    TMOD=0x01;
    TH0=0xfc;
    TL0=0x66;
    EA=1;
    ET0=1;
    TR0=1;
}

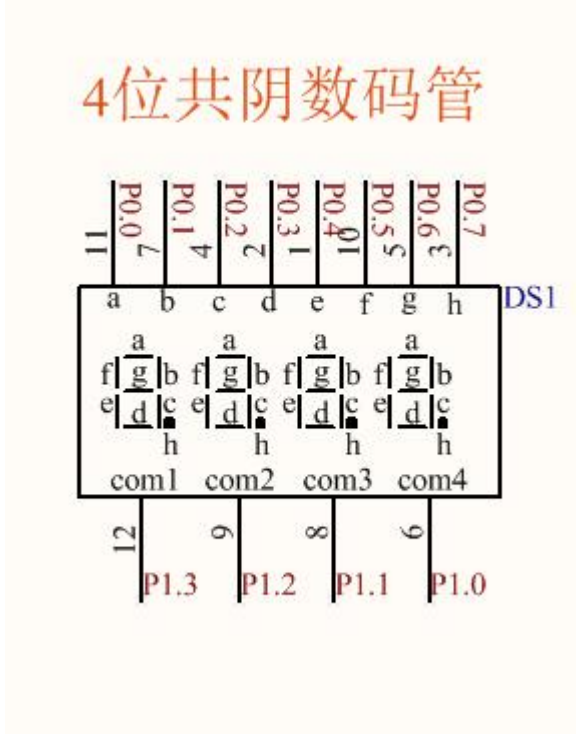
void Delay(unsigned long u32DelayTime)
{
    for(;u32DelayTime>0;u32DelayTime--);
}

void PeripheralInitial(void)
{
}

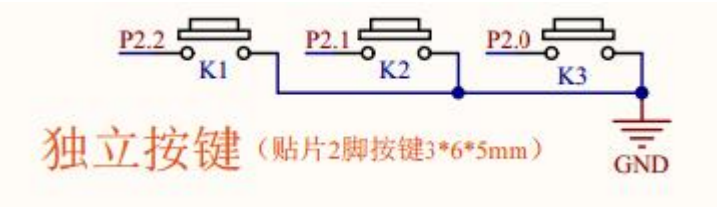
```

第一百二十一节： 可调参数的数码管倒计时。

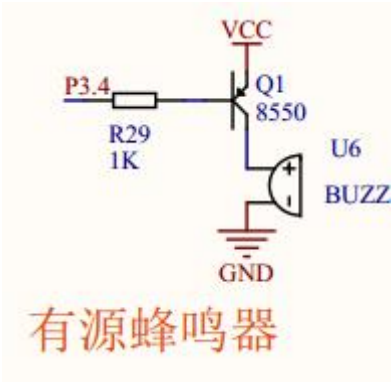
【121.1 可调参数的数码管倒计时。】



上图 121.1.1 数码管



上图 121.1.2 独立按键



上图 121.1.3 有源蜂鸣器

上节讲如何设置数据，本节讲“数据”如何关联“某种功能”，本节的“可调参数”就是“数据”，“倒计时”就是“某种功能”。程序功能如下：

(1) 倒计时范围从 0.00 秒到 99.99 秒，范围可调。开机默认是:10.00 秒。

(2) K1 [设置键]。当数码管“没有闪烁”时，“长按”K1 键则进入“闪烁模式”，某位数码管开始闪烁，闪烁的位代表可修改的位数据，此时再“短按”K1 按键可以使数码管在位之间切换闪烁。当数码管处于“闪烁模式”时，“长按”K1 按键，代表数据修改完成并停止闪烁。

(3) K2 [加键] 与 [复位键]。当数码管某位正在闪烁时，此时 K2 是 [加键]，按 K2 会使位数据“自加 1”。当数码管“没有闪烁”时，此时 K2 是 [复位键]，按 K2 会使当前倒计时数据恢复“设置值”。

(4) K3 [减键] 与 [开始键]。当数码管某位正在闪烁时，此时 K3 是 [减键]，按 K3 会使位数据“自减 1”。当数码管“没有闪烁”时，此时 K3 是 [开始键]，按 K3 开始倒计时。

代码如下：

```
#include "REG52.H"

#define KEY_FILTER_TIME  25    //按键的“短按”兼“滤波”的“稳定时间”
#define KEY_LONG_TIME    500  //按键的“长按”兼“滤波”的“稳定时间”

#define SCAN_TIME  1
#define VOICE_TIME  50
#define BLINK_TIME  250    //数码管闪烁跳动的的时间的间隔

void T0_time();
void SystemInitial(void) ;
void Delay(unsigned long u32DelayTime) ;
void PeripheralInitial(void) ;

void KeyScan(void);
void KeyTask(void);

void VoiceScan(void);
void DisplayScan(void);
void DisplayTask(void); //上层显示的任务函数
void RunTask(void);     //倒计时的应用程序

void Wd1(void); //窗口 1 显示函数。用来设置参数。
void Wd2(void); //窗口 2 显示函数。倒计时的运行显示窗口

void PartUpdate(unsigned char u8Part); //局部选择对应的某个局部变量更新显示输出

void BeepOpen(void);
void BeepClose(void);

sbit KEY_INPUT1=P2^2;
sbit KEY_INPUT2=P2^1;
```



```

sbit KEY_INPUT3=P2^0;

sbit P1_0=P1^0;
sbit P1_1=P1^1;
sbit P1_2=P1^2;
sbit P1_3=P1^3;

sbit P3_4=P3^4;

//数码管转换表
code unsigned char Cu8DigTable[]=
{
0x3f, //0      序号 0
0x06, //1      序号 1
0x5b, //2      序号 2
0x4f, //3      序号 3
0x66, //4      序号 4
0x6d, //5      序号 5
0x7d, //6      序号 6
0x07, //7      序号 7
0x7f, //8      序号 8
0x6f, //9      序号 9
0x00, //不显示 序号 10
0x40, //横杠-  序号 11
};

volatile unsigned char vGu8ScanTimerFlag=0;
volatile unsigned int vGu16ScanTimerCnt=0;

volatile unsigned char vGu8BeepTimerFlag=0;
volatile unsigned int vGu16BeepTimerCnt=0;

volatile unsigned char vGu8BlinkTimerFlag=0; //数码管闪烁跳动的定时器
volatile unsigned int vGu16BlinkTimerCnt=0;

//倒计时的软件定时器，注意，这里是 unsigned long 类型，范围是 0 到 4294967295 毫秒
volatile unsigned char vGu8CountdownTimerFlag=0;
volatile unsigned long vGu32CountdownTimerCnt=10000; //当前倒计时的计时值
unsigned long Gu32SetData_Countdown=10000; //倒计时的设置值

//数码管上层每 10ms 就定时刷新一次显示的软件定时器。用于及时更新显示秒表当前的实时数值
volatile unsigned char vGu8UpdateTimerFlag=0;
volatile unsigned int vGu16UpdateTimerCnt=0;

```

```

unsigned char Gu8RunStart=0; //应用程序的总启动
unsigned char Gu8RunStep=0; //应用程序的总运行步骤。建议跟 vGu8RunStart 成双成对出现
unsigned char Gu8RunStatus=0; //当前倒计时的状态。0 代表停止，1 代表正在工作中

unsigned char Gu8EditData_4=0; //对应显示右起第 4 位数码管的“位”数据，是中间变量。
unsigned char Gu8EditData_3=0; //对应显示右起第 3 位数码管的“位”数据，是中间变量。
unsigned char Gu8EditData_2=0; //对应显示右起第 2 位数码管的“位”数据，是中间变量。
unsigned char Gu8EditData_1=0; //对应显示右起第 1 位数码管的“位”数据，是中间变量。

unsigned char Gu8Wd=1; //窗口选择变量。人机交互程序框架的支点。初始化开机后显示第 1 个窗口。
unsigned char Gu8WdUpdate=1; //整屏更新变量。初始化为 1 开机后整屏更新一次显示。
unsigned char Gu8Part=0; //局部选择变量。0 代表当前窗口下没有数据被选中。
unsigned char Gu8PartUpdate_1=0; //局部 1 的更新变量，
unsigned char Gu8PartUpdate_2=0; //局部 2 的更新变量
unsigned char Gu8PartUpdate_3=0; //局部 3 的更新变量，
unsigned char Gu8PartUpdate_4=0; //局部 4 的更新变量

volatile unsigned char vGu8Display_Righ_4=1; //显示“1”，跟 vGu32CountdownTimerCnt 高位一致
volatile unsigned char vGu8Display_Righ_3=0;
volatile unsigned char vGu8Display_Righ_2=0;
volatile unsigned char vGu8Display_Righ_1=0;

volatile unsigned char vGu8Display_Righ_Dot_4=0;
volatile unsigned char vGu8Display_Righ_Dot_3=1; //开机默认保留显示 2 个小数点
volatile unsigned char vGu8Display_Righ_Dot_2=0;
volatile unsigned char vGu8Display_Righ_Dot_1=0;

volatile unsigned char vGu8KeySec=0;

void main()
{
    SystemInitial();
    Delay(10000);
    PeripheralInitial();
    while(1)
    {
        KeyTask(); //按键的任务函数
        DisplayTask(); //数码管显示的上层任务函数
        RunTask(); //倒计时的应用程序
    }
}

void PartUpdate(unsigned char u8Part) //局部选择对应的某个局部变量更新显示输出

```

```

{
    switch(u8Part)
    {
        case 1:
            Gu8PartUpdate_1=1;
            break;
        case 2:
            Gu8PartUpdate_2=1;
            break;
        case 3:
            Gu8PartUpdate_3=1;
            break;
        case 4:
            Gu8PartUpdate_4=1;
            break;
    }
}

void RunTask(void) //倒计时的应用程序
{
    if(0==Gu8RunStart)
    {
        return; // 如果总开关处于停止状态，则直接退出当前函数，不执行该函数以下的其它代码
    }

    switch(Gu8RunStep)
    {
        case 0: //在这个步骤里，主要用来初始化一些参数

            vGu8UpdateTimerFlag=0;
            vGu16UpdateTimerCnt=10; //每 10ms 更新显示一次当前倒计时的时间
            vGu8UpdateTimerFlag=1;

            Gu8RunStep=1; //跳转到每 10ms 更新显示一次的步骤里
            break;

        case 1: //每 10ms 更新一次显示，确保实时显示当前倒计时的时间
            if(0==vGu16UpdateTimerCnt) //每 10ms 更新显示一次当前倒计时的时间
            {

                vGu8UpdateTimerFlag=0;
                vGu16UpdateTimerCnt=10; //重置定时器，为下一个 10ms 更新做准备
            }
        }
    }
}

```

```

        vGu8UpdateTimerFlag=1;

        Gu8WdUpdate=1; //整屏更新一次显示当前倒计时的时间

        if(0==vGu32CountdownTimerCnt) //如果倒计时的时间到，则跳转到结束的步骤
        {
            Gu8RunStep=2; //跳转到倒计时结束的步骤
        }

    }
    break;

case 2: //倒计时结束的步骤
    //Gu8RunStatus=0; //这行代码注释掉，让每次新启动之前都必须按一次 K1 复位按键才有效

    Gu8RunStart=0; //倒计时的运行步骤的停止
    Gu8RunStep=0; //总运行步骤归零。建议跟 vGu8RunStart 成双成对出现

    vGu8BeepTimerFlag=0;
    vGu16BeepTimerCnt=VOICE_TIME; //蜂鸣器发出“滴”一声
    vGu8BeepTimerFlag=1;

    Gu8WdUpdate=1; //整屏更新一次显示当前倒计时的时间

    break;

}

}

void KeyTask(void) //按键的任务函数
{
    if(0==vGu8KeySec)
    {
        return;
    }

    if(0!=Gu8RunStatus) //在“非停止”状态下，用 return 来拦截一些“不该响应”的按键
    {
        if(2==vGu8KeySec) //在“非停止”状态下，只响应[复位]这个按键
        {
            ; //这里没有 return 语句，表示可以继续往下扫描本函数余下的代码，没有被拦截。
        }
    }
}

```

```

else
{
    return;    //其余的按键则拦截退出
}
}

switch(vGu8KeySec)
{
    case 1:    //按键 K1 的“短按”。切换数码管闪烁的位。
        switch(Gu8Wd) //在某个窗口下
        {
            case 1:    //在窗口 1 下

                if(0!=Gu8Part) //处于“闪烁模式”的时候，是“切换局部”
                {
                    PartUpdate(Gu8Part); //切换之前的局部进行更新。
                    Gu8Part++; //切换局部
                    if(Gu8Part>4)
                    {
                        Gu8Part=1;
                    }
                    PartUpdate(Gu8Part); //切换之后的局部进行更新。

                    vGu8BeepTimerFlag=0;
                    vGu16BeepTimerCnt=VOICE_TIME; //蜂鸣器发出“滴”一声
                    vGu8BeepTimerFlag=1;
                }
                break;
        }

        vGu8KeySec=0;
        break;

    case 2:    //按键 K2 [加键] 与 [复位键]
        if(0!=Gu8Part) //处于“闪烁模式”的时候，是 [加键]
        {
            switch(Gu8Wd) //在某个窗口下
            {
                case 1:    //在窗口 1 下
                    switch(Gu8Part) //二级支点的局部选择
                    {
                        case 1: //局部 1 被选中，代表右起第 4 位数码管被选中。
                            if(Gu8EditData_4<9)

```

```

        {
            Gu8EditData_4++; //编辑“千位”个体的中间变量
        }
        PartUpdate(Gu8Part); //当前局部更新显示输出到数码管
        break;

    case 2: //局部 2 被选中，代表右起第 3 位数码管被选中。
        if (Gu8EditData_3 < 9)
        {
            Gu8EditData_3++; //编辑“百位”个体的中间变量
        }
        PartUpdate(Gu8Part); //当前局部更新显示输出到数码管
        break;

    case 3: //局部 3 被选中，代表右起第 2 位数码管被选中。
        if (Gu8EditData_2 < 9)
        {
            Gu8EditData_2++; //编辑“十位”个体的中间变量
        }
        PartUpdate(Gu8Part); //当前局部更新显示输出到数码管
        break;

    case 4: //局部 4 被选中，代表右起第 1 位数码管被选中。
        if (Gu8EditData_1 < 9)
        {
            Gu8EditData_1++; //编辑“个位”个体的中间变量
        }
        PartUpdate(Gu8Part); //当前局部更新显示输出到数码管
        break;
    }
    break;
}

else //处于“没有闪烁”的时候，是[复位键]
{
    Gu8EditData_4 = Gu32SetData_Countdown / 10000 % 10; //分解成“十秒”个体
    Gu8EditData_3 = Gu32SetData_Countdown / 1000 % 10; //分解成“个秒”个体
    Gu8EditData_2 = Gu32SetData_Countdown / 100 % 10; //分解成“百毫秒”个体
    Gu8EditData_1 = Gu32SetData_Countdown / 10 % 10; //分解成“十毫秒”个体

    Gu8RunStatus = 0; //倒计时返回停止的状态

    Gu8RunStart = 0; //倒计时的运行步骤的停止
    Gu8RunStep = 0; //总运行步骤归零。建议跟 vGu8RunStart 成双成对出现
}

```

```

        Gu8Wd=1; //返回设置数据的窗口
        Gu8WdUpdate=1; //整屏更新一次显示
    }

    vGu8BeepTimerFlag=0;
    vGu16BeepTimerCnt=VOICE_TIME; //蜂鸣器发出“滴”一声
    vGu8BeepTimerFlag=1;

    vGu8KeySec=0;
    break;

case 3: //按键 K3 [减键] 与 [开始键]
    if(0!=Gu8Part) //处于“闪烁模式”的时候，是 [减键]
    {
        switch(Gu8Wd) //在某个窗口下
        {
            case 1: //在窗口 1 下
                switch(Gu8Part) //二级支点的局部选择
                {
                    case 1: //局部 1 被选中，代表右起第 4 位数码管被选中。
                        if(Gu8EditData_4>0)
                        {
                            Gu8EditData_4--; //编辑“十秒”个体的中间变量
                        }
                        PartUpdate(Gu8Part); //当前局部更新显示输出到数码管
                        break;

                    case 2: //局部 2 被选中，代表右起第 3 位数码管被选中。
                        if(Gu8EditData_3>0)
                        {
                            Gu8EditData_3--; //编辑“个秒”个体的中间变量
                        }
                        PartUpdate(Gu8Part); //当前局部更新显示输出到数码管
                        break;

                    case 3: //局部 3 被选中，代表右起第 2 位数码管被选中。
                        if(Gu8EditData_2>0)
                        {
                            Gu8EditData_2--; //编辑“百毫秒”个体的中间变量
                        }
                        PartUpdate(Gu8Part); //当前局部更新显示输出到数码管
                        break;
                }
            }
        }
    }

```

```

        case 4: //局部 4 被选中，代表右起第 1 位数码管被选中。
            if (Gu8EditData_1 > 0)
            {
                Gu8EditData_1--; //编辑“十毫位”个体的中间变量
            }
            PartUpdate(Gu8Part); //当前局部更新显示输出到数码管
            break;
        }
        break;
    }
}
else //处于“没有闪烁”的时候，是[开始键]
{
    if (0 == Gu8RunStatus) //在停止状态下
    {
        vGu8CountdownTimerFlag = 0;
        vGu32CountdownTimerCnt = Gu32SetData_Countdown; //从“设置值”开始倒计时
        vGu8CountdownTimerFlag = 1; //允许倒计时的软件定时器的启动

        Gu8RunStatus = 1; //倒计时处于工作状态（并且，这一瞬间才正式启动倒计时）

        Gu8RunStart = 1; //倒计时的运行步骤的总开关开启
        Gu8RunStep = 0; //总运行步骤归零。建议跟 vGu8RunStart 成双成对出现

        Gu8Wd = 2; //进入倒计时运行的窗口
        Gu8WdUpdate = 1; //整屏更新一次显示，确保在启动的时候能显示到最新的数据
    }
}

vGu8BeepTimerFlag = 0;
vGu16BeepTimerCnt = VOICE_TIME; //蜂鸣器发出“滴”一声
vGu8BeepTimerFlag = 1;

vGu8KeySec = 0;
break;

case 4: //K1 按键的“长按”，具有进入和退出“闪烁模式”的功能。“退出”隐含“确定”

    switch (Gu8Wd) //在某个窗口下
    {
        case 1: //在窗口 1 下
            if (0 == Gu8Part) //处于“没有闪烁”的时候，将进入“闪烁模式”
            {

```



```

        Gu8EditData_4=Gu32SetData_Countdown/10000%10; //分解成“十秒”个体
        Gu8EditData_3=Gu32SetData_Countdown/1000%10; //分解成“个秒”个体
        Gu8EditData_2=Gu32SetData_Countdown/100%10; //分解成“百毫秒”个体
        Gu8EditData_1=Gu32SetData_Countdown/10%10; //分解成“十毫秒”个体
        Gu8Part=1; //进入“闪烁模式”，从“局部1”开始闪烁
    }
    else //处于“闪烁模式”的时候，将退出到“没有闪烁”，隐含“确定”功能
    {
        //把个体合并还原成数据
        Gu32SetData_Countdown=Gu8EditData_4*10000+Gu8EditData_3*1000;
        Gu32SetData_Countdown=Gu32SetData_Countdown+Gu8EditData_2*100;
        Gu32SetData_Countdown=Gu32SetData_Countdown+Gu8EditData_1*10;

        Gu8Part=0; //退出“闪烁模式”
        Gu8WdUpdate=1; //整屏更新
    }

    break;

}

vGu8BeepTimerFlag=0;
vGu16BeepTimerCnt=VOICE_TIME; //蜂鸣器发出“滴”一声
vGu8BeepTimerFlag=1;

vGu8KeySec=0;
break;

}
}

void DisplayTask(void) //数码管显示的上层任务函数
{
    switch(Gu8Wd) //以窗口选择 Gu8Wd 为支点，去执行对应的窗口显示函数。又一次用到 switch 语句
    {
        case 1:
            Wd1(); //窗口 1 显示函数。用来设置参数。
            break;
        case 2:
            Wd2(); //窗口 2 显示函数。倒计时的运行显示窗口。
            break;
    }
}
}

```

```

void Wd1(void)    //窗口 1 显示函数。用来设置参数。
{
    //需要借用的中间变量，用来拆分数数据位。
    static unsigned char Su8Temp_4, Su8Temp_3, Su8Temp_2, Su8Temp_1; //需要借用的中间变量
    static unsigned char Su8BlinkFlag=0; //两种状态的切换判断的中间变量

    if(1==Gu8WdUpdate) //如果需要整屏更新
    {
        Gu8WdUpdate=0; //及时清零，只更新一次显示即可，避免一直进来更新显示

        //属于静态数据，起“装饰”作用，切换窗口后只扫描一次的代码。
        vGu8Display_Righ_Dot_4=0;
        vGu8Display_Righ_Dot_3=1;    //保留显示 2 位小数点
        vGu8Display_Righ_Dot_2=0;
        vGu8Display_Righ_Dot_1=0;

        Gu8PartUpdate_1=1; //局部 1 更新显示
        Gu8PartUpdate_2=1  ;//局部 2 更新显示
        Gu8PartUpdate_3=1  ;//局部 3 更新显示
        Gu8PartUpdate_4=1  ;//局部 4 更新显示

    }

    if(1==Gu8PartUpdate_1) //局部 1 更新显示
    {
        Gu8PartUpdate_1=0; //及时清零，只更新一次显示即可，避免一直进来更新显示

        if(Gu32SetData_Countdown<10000)
        {
            Su8Temp_4=10; //显示“无”
        }
        else
        {
            Su8Temp_4=Gu8EditData_4; //显示“十秒”的临时中间个体，属于动态数据。
        }

        vGu8Display_Righ_4=Su8Temp_4; //过渡需要显示的数据到底层驱动变量
    }

    if(1==Gu8PartUpdate_2) //局部 2 更新显示
    {
        Gu8PartUpdate_2=0; //及时清零，只更新一次显示即可，避免一直进来更新显示
    }
}

```

```

    Su8Temp_3=Gu8EditData_3; //显示“个秒”的临时中间个体，属于动态数据。

    vGu8Display_Righ_3=Su8Temp_3; //过渡需要显示的数据到底层驱动变量
}

if(1==Gu8PartUpdate_3) //局部3 更新显示
{
    Gu8PartUpdate_3=0; //及时清零，只更新一次显示即可，避免一直进来更新显示

    Su8Temp_2=Gu8EditData_2; //显示“百毫秒”的临时中间个体，属于动态数据。

    vGu8Display_Righ_2=Su8Temp_2; //过渡需要显示的数据到底层驱动变量
}

if(1==Gu8PartUpdate_4) //局部4 更新显示
{
    Gu8PartUpdate_4=0; //及时清零，只更新一次显示即可，避免一直进来更新显示

    Su8Temp_1=Gu8EditData_1; //显示“十毫秒”的临时中间个体，属于动态数据。

    vGu8Display_Righ_1=Su8Temp_1; //过渡需要显示的数据到底层驱动变量
}

if(0==vGu16BlinkTimerCnt) //某位被选中的数码管跳动闪烁的定时器
{
    vGu8BlinkTimerFlag=0;
    vGu16BlinkTimerCnt=BLINK_TIME; //重设定时器的定时时间
    vGu8BlinkTimerFlag=1;

    switch(Gu8Part) //某个局部被选中，则闪烁跳动
    {
        case 1:
            if(0==Su8BlinkFlag) //两种状态的切换判断
            {
                Su8BlinkFlag=1;
                Su8Temp_4=10; //右起第4个显示“不显示”（10代表不显示）
            }
            else
            {
                Su8BlinkFlag=0;
                Su8Temp_4=Gu8EditData_4; //显示“十秒”的临时中间个体，属于动态数据。
            }
        }
    }
}

```

```
        break;

    case 2:
        if (0==Su8BlinkFlag) //两种状态的切换判断
        {
            Su8BlinkFlag=1;
            Su8Temp_3=10; //右起第 3 个显示 “不显示”（10 代表不显示）
        }
        else
        {
            Su8BlinkFlag=0;
            Su8Temp_3=Gu8EditData_3; //显示 “个秒” 的临时中间个体，属于动态数据。
        }

        break;

    case 3:
        if (0==Su8BlinkFlag) //两种状态的切换判断
        {
            Su8BlinkFlag=1;
            Su8Temp_2=10; //右起第 2 个显示 “不显示”（10 代表不显示）
        }
        else
        {
            Su8BlinkFlag=0;
            Su8Temp_2=Gu8EditData_2; //显示 “百毫秒” 的临时中间个体，属于动态数据。
        }

        break;

    case 4:
        if (0==Su8BlinkFlag) //两种状态的切换判断
        {
            Su8BlinkFlag=1;
            Su8Temp_1=10; //右起第 1 个显示 “不显示”（10 代表不显示）
        }
        else
        {
            Su8BlinkFlag=0;
            Su8Temp_1=Gu8EditData_1; //显示 “十毫秒” 的临时中间个体，属于动态数据。
        }

        break;
```

```

        default:    //都没有被选中的时候
            if (Gu32SetData_Countdown<10000)
            {
                Su8Temp_4=10;  //显示“无”
            }
            else
            {
                Su8Temp_4=Gu8EditData_4;  //显示“十秒”的临时中间个体，属于动态数据。
            }
            Su8Temp_3=Gu8EditData_3;  //显示“个秒”的临时中间个体，属于动态数据。
            Su8Temp_2=Gu8EditData_2;  //显示“百毫秒”的临时中间个体，属于动态数据。
            Su8Temp_1=Gu8EditData_1;  //显示“十毫秒”的临时中间个体，属于动态数据。
            break;
        }

        vGu8Display_Righ_4=Su8Temp_4;  //过渡需要显示的数据到底层驱动变量
        vGu8Display_Righ_3=Su8Temp_3;  //过渡需要显示的数据到底层驱动变量
        vGu8Display_Righ_2=Su8Temp_2;  //过渡需要显示的数据到底层驱动变量
        vGu8Display_Righ_1=Su8Temp_1;  //过渡需要显示的数据到底层驱动变量
    }
}

void Wd2(void)    //窗口 2 显示函数。倒计时的运行显示窗口。
{
    //需要借用的中间变量，用来拆分数据位。
    static unsigned char Su8Temp_4, Su8Temp_3, Su8Temp_2, Su8Temp_1; //需要借用的中间变量

    if (1==Gu8WdUpdate) //如果需要整屏更新
    {
        Gu8WdUpdate=0; //及时清零，只更新一次显示即可，避免一直进来更新显示

        //先分解数据，注意，这里分解的时候，“先整除后求余”必须用一行代码一气呵成，不能拆
        //分成两行代码，否则会有隐患会有 bug。除非，把四个临时变都改成 unsigned long 类型。

        //Su8Temp_4 提取“十秒”位。
        Su8Temp_4=vGu32CountdownTimerCnt/10000%10; //实际精度是 0.001 秒,但显示精度是 0.01 秒

        //Su8Temp_3 提取“个秒”位。
        Su8Temp_3=vGu32CountdownTimerCnt/1000%10; //实际精度是 0.001 秒,但显示精度是 0.01 秒

        //Su8Temp_2 提取“百毫秒”位。
        Su8Temp_2=vGu32CountdownTimerCnt/100%10; //实际精度是 0.001 秒,但显示精度是 0.01 秒
    }
}

```

```

//Su8Temp_1 提取“十毫秒”位。
Su8Temp_1=vGu32CountdownTimerCnt/10%10; //实际精度是 0.001 秒,但显示精度是 0.01 秒

//判断数据范围,来决定最高位数码管是否需要显示。
if(vGu32CountdownTimerCnt<10000) //10.000 秒。实际 4 位数码管最大只能显示 99.99 秒
{
    Su8Temp_4=10; //在数码管转换表里,10 代表一个“不显示”的数据
}

//上面先分解数据之后,再过渡需要显示的数据到底层驱动变量里,让过渡的时间越短越好
vGu8Display_Righ_4=Su8Temp_4; //过渡需要显示的数据到底层驱动变量
vGu8Display_Righ_3=Su8Temp_3;
vGu8Display_Righ_2=Su8Temp_2;
vGu8Display_Righ_1=Su8Temp_1;

vGu8Display_Righ_Dot_4=0;
vGu8Display_Righ_Dot_3=1; //保留显示 2 位小数点
vGu8Display_Righ_Dot_2=0;
vGu8Display_Righ_Dot_1=0;

}
}

void KeyScan(void) //按键底层的驱动扫描函数,放在定时中断函数里
{
    static unsigned char Su8KeyShortFlag=0; //按键“短按”触发的标志
    static unsigned char Su8KeyLock1;
    static unsigned int Su16KeyCnt1;
    static unsigned char Su8KeyLock2;
    static unsigned int Su16KeyCnt2;
    static unsigned char Su8KeyLock3;
    static unsigned int Su16KeyCnt3;

    //需要详细分析以下这段“短按”与“长按”代码的朋友,请参考第 96 节。
    if(0!=KEY_INPUT1)
    {
        Su8KeyLock1=0;
        Su16KeyCnt1=0;
        if(1==Su8KeyShortFlag)
        {
            Su8KeyShortFlag=0;
            vGu8KeySec=1; //触发 K1 的“短按”
        }
    }
}

```

```
else if (0==Su8KeyLock1)
{
    Su16KeyCnt1++;

    if (Su16KeyCnt1>=KEY_FILTER_TIME)
    {
        Su8KeyShortFlag=1;
    }

    if (Su16KeyCnt1>=KEY_LONG_TIME)
    {
        Su8KeyLock1=1;
        Su8KeyShortFlag=0;
        vGu8KeySec=4; //触发 K1 的“长按”
    }
}

if (0!=KEY_INPUT2)
{
    Su8KeyLock2=0;
    Su16KeyCnt2=0;
}
else if (0==Su8KeyLock2)
{
    Su16KeyCnt2++;
    if (Su16KeyCnt2>=KEY_FILTER_TIME)
    {
        Su8KeyLock2=1;
        vGu8KeySec=2;
    }
}

if (0!=KEY_INPUT3)
{
    Su8KeyLock3=0;
    Su16KeyCnt3=0;
}
else if (0==Su8KeyLock3)
{
    Su16KeyCnt3++;
    if (Su16KeyCnt3>=KEY_FILTER_TIME)
    {
        Su8KeyLock3=1;
        vGu8KeySec=3;
    }
}
```

```

    }
}

void DisplayScan(void)    //数码管底层的驱动扫描函数，放在定时中断函数里
{
    static unsigned char Su8GetCode;
    static unsigned char Su8ScanStep=1;

    if(0==vGu16ScanTimerCnt)
    {

        P0=0x00;
        P1_0=1;
        P1_1=1;
        P1_2=1;
        P1_3=1;

        switch(Su8ScanStep)
        {
            case 1:
                Su8GetCode=Cu8DigTable[vGu8Display_Righ_1];

                if(1==vGu8Display_Righ_Dot_1)
                {
                    Su8GetCode=Su8GetCode|0x80;
                }
                P0=Su8GetCode;
                P1_0=0;
                P1_1=1;
                P1_2=1;
                P1_3=1;
                break;

            case 2:
                Su8GetCode=Cu8DigTable[vGu8Display_Righ_2];
                if(1==vGu8Display_Righ_Dot_2)
                {
                    Su8GetCode=Su8GetCode|0x80;
                }
                P0=Su8GetCode;
                P1_0=1;

```



```

        P1_1=0;
        P1_2=1;
        P1_3=1;
        break;

    case 3:
        Su8GetCode=Cu8DigTable[vGu8Display_Righ_3];
        if(1==vGu8Display_Righ_Dot_3)
        {
            Su8GetCode=Su8GetCode|0x80;
        }
        P0=Su8GetCode;
        P1_0=1;
        P1_1=1;
        P1_2=0;
        P1_3=1;
        break;

    case 4:
        Su8GetCode=Cu8DigTable[vGu8Display_Righ_4];
        if(1==vGu8Display_Righ_Dot_4)
        {
            Su8GetCode=Su8GetCode|0x80;
        }
        P0=Su8GetCode;
        P1_0=1;
        P1_1=1;
        P1_2=1;
        P1_3=0;
        break;

    }

    Su8ScanStep++;
    if(Su8ScanStep>4)
    {
        Su8ScanStep=1;
    }

    vGu8ScanTimerFlag=0;
    vGu16ScanTimerCnt=SCAN_TIME;
    vGu8ScanTimerFlag=1;
}
}

```

```

void VoiceScan(void) //蜂鸣器的驱动函数
{

    static unsigned char Su8Lock=0;

    if(1==vGu8BeepTimerFlag&&vGu16BeepTimerCnt>0)
    {
        if(0==Su8Lock)
        {
            Su8Lock=1;
            BeepOpen();
        }
        else
        {

            vGu16BeepTimerCnt--;

            if(0==vGu16BeepTimerCnt)
            {
                Su8Lock=0;
                BeepClose();
            }

        }
    }
}

```

```

void BeepOpen(void)
{
    P3_4=0;
}

```

```

void BeepClose(void)
{
    P3_4=1;
}

```

```

void T0_time() interrupt 1
{
    VoiceScan();    //蜂鸣器的驱动函数
    KeyScan();      //按键底层的驱动扫描函数
    DisplayScan();  //数码管底层的驱动扫描函数
}

```

```

if (1==vGu8ScanTimerFlag&&vGu16ScanTimerCnt>0)
{
    vGu16ScanTimerCnt--; //递减式的软件定时器
}

if (1==vGu8BlinkTimerFlag&&vGu16BlinkTimerCnt>0) //数码管闪烁跳动的定时器
{
    vGu16BlinkTimerCnt--; //递减式的软件定时器
}

//每 10ms 就定时更新一次显示的软件定时器
if (1==vGu8UpdateTimerFlag&&vGu16UpdateTimerCnt>0)
{
    vGu16UpdateTimerCnt--; //递减式的软件定时器
}

//倒计时实际走的时间的软件定时器，注意，这里还附加了启动状态的条件 “&&1==Gu8RunStatus”
if (1==vGu8CountdownTimerFlag&&vGu32CountdownTimerCnt>0&&1==Gu8RunStatus)
{
    vGu32CountdownTimerCnt--; //递减式的软件定时器
}

TH0=0xfd; //此参数可根据具体的时间来修改，尽量确保每定时中断一次接近 1ms
TL0=0x40; //此参数可根据具体的时间来修改，尽量确保每定时中断一次接近 1ms
}

void SystemInitial(void)
{
    P0=0x00;
    P1_0=1;
    P1_1=1;
    P1_2=1;
    P1_3=1;

    TMOD=0x01;
    TH0=0xfd; //此参数可根据具体的时间来修改，尽量确保每定时中断一次接近 1ms
    TL0=0x40; //此参数可根据具体的时间来修改，尽量确保每定时中断一次接近 1ms
    EA=1;
    ET0=1;
    TR0=1;

    //上电初始化开机显示的窗口
    Gu8EditData_4=Gu32SetData_Countdown/10000%10; //分解成“十秒”个体
    Gu8EditData_3=Gu32SetData_Countdown/1000%10; //分解成“个秒”个体

```

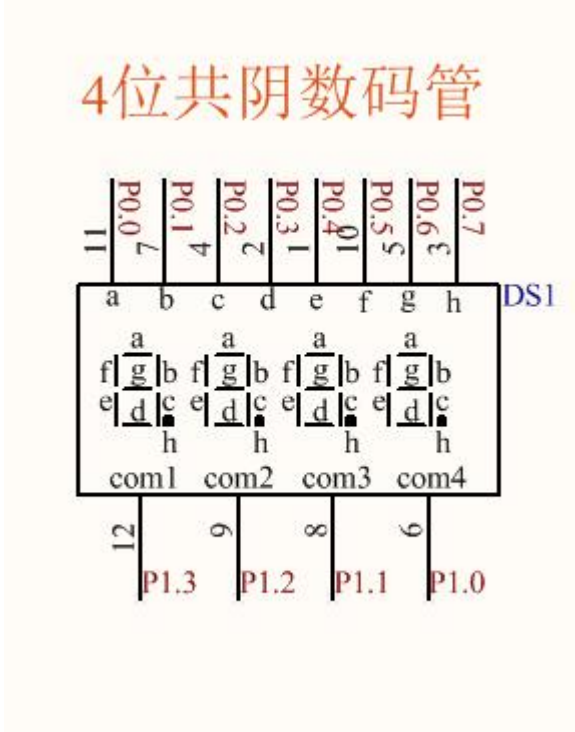
```
    Gu8EditData_2=Gu32SetData_Countdown/100%10; //分解成“百毫秒”个体
    Gu8EditData_1=Gu32SetData_Countdown/10%10;  //分解成“十毫秒”个体
    Gu8Wd=1; //返回设置数据的窗口
    Gu8WdUpdate=1;  //整屏更新一次显示
}

void Delay(unsigned long u32DelayTime)
{
    for(;u32DelayTime>0;u32DelayTime--);
}

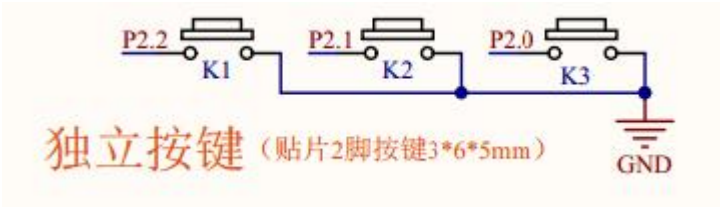
void PeripheralInitial(void)
{
}
}
```

第一百二十二节： 利用定时中断做的“时分秒”数显时钟。

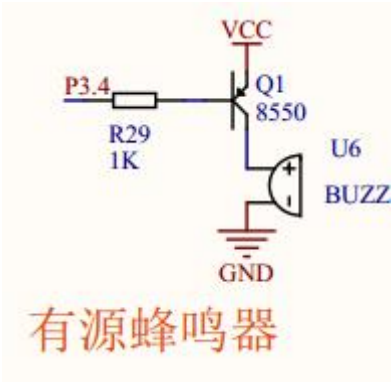
【122.1 利用定时中断做的“时分秒”数显时钟。】



上图 122. 1. 1 数码管



上图 122. 1. 2 独立按键



上图 122. 1. 3 有源蜂鸣器

本节的数显时钟小项目，意在人机界面程序框架的练习。程序功能如下：

(1) 只有“时分秒”，没有“年月日”。

(2) 平时时钟正常工作的时候，四位数码管的显示格式是这样的“HH.MM”，其中 HH 代表“时”，MM 代表“分”，而中间的小数点“.”每隔一秒闪烁一次。

(3) K1 [设置键] 与 [切换窗口键]。当数码管“没有闪烁”时（处于正常工作模式），“长按”K1 键则进入“闪烁模式”（修改时钟模式），“闪烁模式”一共有 3 个窗口，分别是“1-HH”，“2-MM”，“3-SS”。其中“HH”“MM”“SS”分别代表可修改的“时”“分”“秒”，它们处于“闪烁”的状态，代表可编辑。此时，“短按”K1 按键代表 [切换窗口键]，可以使数码管在“1-HH”，“2-MM”，“3-SS”三个窗口之间依次切换。修改完毕后，只需“长按”K1 键代表确定完成并且退出当前“闪烁模式”返回到时钟的“正常工作模式”。

(4) K2 [加键]。当数码管某位正在闪烁时，此时 K2 是 [加键]，按 K2 会使数据“自加 1”。

(5) K3 [减键]。当数码管某位正在闪烁时，此时 K3 是 [减键]，按 K3 会使数据“自减 1”。

(6) 处于“闪烁模式”时的 3 个窗口的数据范围。处于修改“时”的“1-HH”窗口时，HH 的范围是：0 到 23；处于修改“分”的“2-MM”窗口时，MM 的范围是：0 到 59；处于修改“秒”的“3-SS”窗口时，SS 的范围是：0 到 59。

代码如下：

```
#include "REG52.H"

#define KEY_FILTER_TIME  25
#define KEY_LONG_TIME    500

#define SCAN_TIME  1
#define VOICE_TIME  50
#define BLINK_TIME  250

void TO_time();
void SystemInitial(void) ;
void Delay(unsigned long u32DelayTime) ;
void PeripheralInitial(void) ;

void KeyScan(void);
void KeyTask(void);

void VoiceScan(void);
void DisplayScan(void);
void DisplayTask(void);
void Wd1(void);    //窗口 1。时钟正常工作的窗口“HH.MM”。小数点在闪烁跳动。
void Wd2(void);    //窗口 2。闪烁模式，修改“时”的“1-HH”的窗口。
void Wd3(void);    //窗口 3。闪烁模式，修改“分”的“2-MM”的窗口。
void Wd4(void);    //窗口 4。闪烁模式，修改“秒”的“3-SS”的窗口。

void BeepOpen(void);
void BeepClose(void);
```

```

sbit KEY_INPUT1=P2^2;
sbit KEY_INPUT2=P2^1;
sbit KEY_INPUT3=P2^0;

sbit P1_0=P1^0;
sbit P1_1=P1^1;
sbit P1_2=P1^2;
sbit P1_3=P1^3;

sbit P3_4=P3^4;

//数码管转换表
code unsigned char Cu8DigTable[]=
{
0x3f,  //0      序号 0
0x06,  //1      序号 1
0x5b,  //2      序号 2
0x4f,  //3      序号 3
0x66,  //4      序号 4
0x6d,  //5      序号 5
0x7d,  //6      序号 6
0x07,  //7      序号 7
0x7f,  //8      序号 8
0x6f,  //9      序号 9
0x00,  //不显示 序号 10
0x40,  //横杠-   序号 11
};

volatile unsigned char vGu8ScanTimerFlag=0;
volatile unsigned int vGu16ScanTimerCnt=0;

volatile unsigned char vGu8BeepTimerFlag=0;
volatile unsigned int vGu16BeepTimerCnt=0;

volatile unsigned char vGu8BlinkTimerFlag=0;
volatile unsigned int vGu16BlinkTimerCnt=0;

//时钟的软件定时器，注意，这里是 unsigned long 类型，范围是 0 到 4294967295 毫秒
volatile unsigned char vGu8ClockTimerFlag=0;
volatile unsigned long vGu32ClockTimerCnt=0;

//时钟正常工作的时候，每 500ms 更新显示一次
volatile unsigned char vGu8UpdateTimerFlag=0;
volatile unsigned int vGu16UpdateTimerCnt=0;

```

```

unsigned char Gu8EditData_1=0; //是中间变量，用于编辑窗口“1-HH”下的 HH 数据。
unsigned char Gu8EditData_2=0; //是中间变量，用于编辑窗口“2-MM”下的 MM 数据。
unsigned char Gu8EditData_3=0; //是中间变量，用于编辑窗口“3-SS”下的 SS 数据。

unsigned char Gu8Wd=0; //窗口选择变量。人机交互程序框架的支点。
unsigned char Gu8WdUpdate=0; //整屏更新变量。

unsigned char Gu8PartUpdate_1=0; //局部 1 的更新变量,
unsigned char Gu8PartUpdate_2=0; //局部 2 的更新变量
unsigned char Gu8PartUpdate_3=0; //局部 3 的更新变量,

volatile unsigned char vGu8Display_Righ_4=0;
volatile unsigned char vGu8Display_Righ_3=0;
volatile unsigned char vGu8Display_Righ_2=0;
volatile unsigned char vGu8Display_Righ_1=0;

volatile unsigned char vGu8Display_Righ_Dot_4=0;
volatile unsigned char vGu8Display_Righ_Dot_3=1; //开机默认保留显示 2 个小数点
volatile unsigned char vGu8Display_Righ_Dot_2=0;
volatile unsigned char vGu8Display_Righ_Dot_1=0;

volatile unsigned char vGu8KeySec=0;

void main()
{
    SystemInitial();
    Delay(10000);
    PeripheralInitial();
    while(1)
    {
        KeyTask(); //按键的任务函数
        DisplayTask(); //数码管显示的上层任务函数
    }
}

void KeyTask(void) //按键的任务函数
{
    if(0==vGu8KeySec)
    {
        return;
    }

    switch(vGu8KeySec)

```



```

{
case 1:    //按键 K1 的“短按”。在“闪烁模式”下切换数码管的窗口。
switch(Gu8Wd) //在某个窗口下
{
case 2:    //窗口 2。修改“时”的“1-HH”窗口。
Gu8Wd=3; //切换到窗口 3 的“2-MM”窗口
Gu8WdUpdate=1; //整屏更新
break;

case 3:    //窗口 3。修改“分”的“2-MM”窗口。
Gu8Wd=4; //切换到窗口 4 的“3-SS”窗口
Gu8WdUpdate=1; //整屏更新
break;

case 4:    //窗口 4。修改“秒”的“3-SS”窗口。
Gu8Wd=2; //切换到窗口 2 的“1-HH”窗口
Gu8WdUpdate=1; //整屏更新
break;
}
}

```

```

vGu8BeepTimerFlag=0;
vGu16BeepTimerCnt=VOICE_TIME; //蜂鸣器发出“滴”一声
vGu8BeepTimerFlag=1;

```

```

vGu8KeySec=0;
break;

```

```

case 2:    //按键 K2 [加键]

switch(Gu8Wd) //在某个窗口下
{
case 2:    //窗口 2。修改“时”的“1-HH”窗口。
if(Gu8EditData_1<23) //“时”的范围是 0 到 23
{
Gu8EditData_1++;
}
Gu8PartUpdate_1=1; //局部 1 更新显示
break;

case 3:    //窗口 3。修改“分”的“2-MM”窗口。
if(Gu8EditData_2<59) //“分”的范围是 0 到 59
{
Gu8EditData_2++;
}
}
}

```

```

        Gu8PartUpdate_1=1;    //局部 1 更新显示
        break;

    case 4:    //窗口 4。修改“秒”的“3-SS”窗口。
        if(Gu8EditData_3<59) //“秒”的范围是 0 到 59
        {
            Gu8EditData_3++;
        }
        Gu8PartUpdate_1=1;    //局部 1 更新显示
        break;
}

vGu8BeepTimerFlag=0;
vGu16BeepTimerCnt=VOICE_TIME; //蜂鸣器发出“滴”一声
vGu8BeepTimerFlag=1;

vGu8KeySec=0;
break;

case 3:    //按键 K3 [减键] 与 [开始键]
    switch(Gu8Wd) //在某个窗口下
    {
        case 2:    //窗口 2。修改“时”的“1-HH”窗口。
            if(Gu8EditData_1>0)
            {
                Gu8EditData_1--;
            }
            Gu8PartUpdate_1=1;    //局部 1 更新显示
            break;

        case 3:    //窗口 3。修改“分”的“2-MM”窗口。
            if(Gu8EditData_2>0)
            {
                Gu8EditData_2--;
            }
            Gu8PartUpdate_1=1;    //局部 1 更新显示
            break;

        case 4:    //窗口 4。修改“秒”的“3-SS”窗口。
            if(Gu8EditData_3>0)
            {
                Gu8EditData_3--;
            }
            Gu8PartUpdate_1=1;    //局部 1 更新显示

```

```

        break;
    }

    vGu8BeepTimerFlag=0;
    vGu16BeepTimerCnt=VOICE_TIME; //蜂鸣器发出“滴”一声
    vGu8BeepTimerFlag=1;

    vGu8KeySec=0;
    break;

case 4: //K1 按键的“长按”，具有进入和退出“闪烁模式”的功能。“退出”隐含“确定”

    switch(Gu8Wd) //在某个窗口下
    {
        case 1: //窗口 1。时钟正常工作的窗口。
            vGu8ClockTimerFlag=0; //停止时钟的定时器

            Gu8EditData_1=vGu32ClockTimerCnt/3600000; //分解成“时”个体
            Gu8EditData_2=vGu32ClockTimerCnt%3600000/60000; //分解成“分”个体
            Gu8EditData_3=vGu32ClockTimerCnt%3600000%60000/1000; //分解成“秒”个体

            Gu8Wd=2; //切换到窗口 2 的“1-HH”的闪烁窗口
            Gu8WdUpdate=1; //整屏更新
            break;

        case 2: //窗口 2。修改时钟时间的“1-HH”的闪烁窗口
        case 3: //窗口 3。修改时钟时间的“2-MM”的闪烁窗口
        case 4: //窗口 4。修改时钟时间的“3-SS”的闪烁窗口
            //把个体合并还原成当前时钟时间的数据
            vGu32ClockTimerCnt=Gu8EditData_1*3600000+Gu8EditData_2*60000+Gu8EditData_3*1000;
            vGu8ClockTimerFlag=1; //启动时钟的定时器

            Gu8Wd=1; //切换到窗口 1 的正常工作的窗口
            Gu8WdUpdate=1; //整屏更新
            break;
    }

    vGu8BeepTimerFlag=0;
    vGu16BeepTimerCnt=VOICE_TIME; //蜂鸣器发出“滴”一声
    vGu8BeepTimerFlag=1;

    vGu8KeySec=0;
    break;

```

```

    }
}

void DisplayTask(void) //数码管显示的上层任务函数
{
    switch(Gu8Wd) //以窗口选择 Gu8Wd 为支点，去执行对应的窗口显示函数。又一次用到 switch 语句
    {
        case 1:
            Wd1(); //窗口 1。时钟正常运行的窗口
            break;
        case 2:
            Wd2(); //窗口 2。修改“时”的“1-HH”窗口
            break;
        case 3:
            Wd3(); //窗口 3。修改“分”的“2-MM”窗口
            break;
        case 4:
            Wd4(); //窗口 4。修改“秒”的“3-SS”窗口
            break;
    }
}

void Wd1(void) //窗口 1。时钟正常运行的窗口
{
    //需要借用的中间变量，用来拆分数数据位。
    static unsigned char Su8Temp_4, Su8Temp_3, Su8Temp_2, Su8Temp_1; //需要借用的中间变量
    static unsigned char Su8BlinkFlag=0; //两种状态的切换判断的中间变量

    if(1==Gu8WdUpdate) //如果需要整屏更新
    {
        Gu8WdUpdate=0; //及时清零，只更新一次显示即可，避免一直进来更新显示

        //属于静态数据，起“装饰”作用，切换窗口后只扫描一次的代码。
        vGu8Display_Righ_Dot_4=0;
        vGu8Display_Righ_Dot_3=1; //保留显示 2 位小数点
        vGu8Display_Righ_Dot_2=0;
        vGu8Display_Righ_Dot_1=0;

        Gu8PartUpdate_1=1; //局部 1 更新显示
    }

    if(1==Gu8PartUpdate_1) //局部 1 更新显示，更新显示一次数据和闪烁的小数点
    {

```

```

    Gu8PartUpdate_1=0; //及时清零，只更新一次显示即可，避免一直进来更新显示

    Su8Temp_4=vGu32ClockTimerCnt/3600000/10; //时的十位
    Su8Temp_3=vGu32ClockTimerCnt/3600000%10; //时的个位
    Su8Temp_2=vGu32ClockTimerCnt%3600000/60000/10; //分的十位
    Su8Temp_1=vGu32ClockTimerCnt%3600000/60000%10; //秒的个位

    //小数点的闪烁
    if (0==Su8BlinkFlag)
    {
        Su8BlinkFlag=1;
        vGu8Display_Righ_Dot_3=1;    //显示第 2 位小数点。

    }
    else
    {
        Su8BlinkFlag=0;
        vGu8Display_Righ_Dot_3=0;    //不显示第 2 位小数点
    }

    vGu8Display_Righ_4=Su8Temp_4; //过渡需要显示的数据到底层驱动变量
    vGu8Display_Righ_3=Su8Temp_3; //过渡需要显示的数据到底层驱动变量
    vGu8Display_Righ_2=Su8Temp_2; //过渡需要显示的数据到底层驱动变量
    vGu8Display_Righ_1=Su8Temp_1; //过渡需要显示的数据到底层驱动变量
}

if (0==vGu16UpdateTimerCnt) //每隔 500ms 就更新显示一次数据和闪烁的小数点
{
    vGu8UpdateTimerFlag=0;
    vGu16UpdateTimerCnt=500; //重设定时器的定时时间
    vGu8UpdateTimerFlag=1;

    Gu8PartUpdate_1=1; //局部 1 更新显示
}

}

void Wd2(void) //窗口 2。修改“时”的“1-HH”窗口。
{
    //需要借用的中间变量，用来拆分数据位。
    static unsigned char Su8Temp_2, Su8Temp_1; //需要借用的中间变量
    static unsigned char Su8BlinkFlag=0; //两种状态的切换判断的中间变量

```

```

if(1==Gu8WdUpdate) //如果需要整屏更新
{
    Gu8WdUpdate=0; //及时清零，只更新一次显示即可，避免一直进来更新显示

    //属于静态数据，起“装饰”作用，切换窗口后只扫描一次的代码。
    vGu8Display_Righ_4=1; //显示数字“1”
    vGu8Display_Righ_3=11; //显示横杠“-”

    vGu8Display_Righ_Dot_4=0;
    vGu8Display_Righ_Dot_3=0;
    vGu8Display_Righ_Dot_2=0;
    vGu8Display_Righ_Dot_1=0;

    Gu8PartUpdate_1=1; //局部1更新显示
}

if(1==Gu8PartUpdate_1) //局部1更新显示
{
    Gu8PartUpdate_1=0; //及时清零，只更新一次显示即可，避免一直进来更新显示

    Su8Temp_2=Gu8EditData_1/10; //显示“时”的十位
    Su8Temp_1=Gu8EditData_1%10; //显示“时”的个位

    vGu8Display_Righ_2=Su8Temp_2; //过渡需要显示的数据到底层驱动变量
    vGu8Display_Righ_1=Su8Temp_1; //过渡需要显示的数据到底层驱动变量
}

if(0==vGu16BlinkTimerCnt) //某位被选中的数码管跳动闪烁的定时器
{
    vGu8BlinkTimerFlag=0;
    vGu16BlinkTimerCnt=BLINK_TIME; //重设定时器的定时时间
    vGu8BlinkTimerFlag=1;

    if(0==Su8BlinkFlag) //两种状态的切换判断
    {
        Su8BlinkFlag=1;
        Su8Temp_2=10; //10代表不显示
        Su8Temp_1=10; //10代表不显示
    }
    else
    {
        Su8BlinkFlag=0;
        Su8Temp_2=Gu8EditData_1/10; //显示“时”的十位
    }
}

```

```

        Su8Temp_1=Gu8EditData_1%10; //显示“时”的个位

    }

    vGu8Display_Righ_2=Su8Temp_2; //过渡需要显示的数据到底层驱动变量
    vGu8Display_Righ_1=Su8Temp_1; //过渡需要显示的数据到底层驱动变量
}

void Wd3(void) //窗口 3。修改“分”的“2-MM”窗口。
{
    //需要借用的中间变量，用来拆分数数据位。
    static unsigned char Su8Temp_2, Su8Temp_1; //需要借用的中间变量
    static unsigned char Su8BlinkFlag=0; //两种状态的切换判断的中间变量

    if(1==Gu8WdUpdate) //如果需要整屏更新
    {
        Gu8WdUpdate=0; //及时清零，只更新一次显示即可，避免一直进来更新显示

        //属于静态数据，起“装饰”作用，切换窗口后只扫描一次的代码。
        vGu8Display_Righ_4=2; //显示数字“2”
        vGu8Display_Righ_3=11; //显示横杠“-”

        vGu8Display_Righ_Dot_4=0;
        vGu8Display_Righ_Dot_3=0;
        vGu8Display_Righ_Dot_2=0;
        vGu8Display_Righ_Dot_1=0;

        Gu8PartUpdate_1=1; //局部 1 更新显示
    }

    if(1==Gu8PartUpdate_1) //局部 1 更新显示
    {
        Gu8PartUpdate_1=0; //及时清零，只更新一次显示即可，避免一直进来更新显示

        Su8Temp_2=Gu8EditData_2/10; //显示“分”的十位
        Su8Temp_1=Gu8EditData_2%10; //显示“分”的个位

        vGu8Display_Righ_2=Su8Temp_2; //过渡需要显示的数据到底层驱动变量
        vGu8Display_Righ_1=Su8Temp_1; //过渡需要显示的数据到底层驱动变量
    }

    if(0==vGu16BlinkTimerCnt) //某位被选中的数码管跳动闪烁的定时器

```

```

{
    vGu8BlinkTimerFlag=0;
    vGu16BlinkTimerCnt=BLINK_TIME; //重设定时器的定时时间
    vGu8BlinkTimerFlag=1;

    if(0==Su8BlinkFlag) //两种状态的切换判断
    {
        Su8BlinkFlag=1;
        Su8Temp_2=10; //10 代表不显示
        Su8Temp_1=10; //10 代表不显示
    }
    else
    {
        Su8BlinkFlag=0;
        Su8Temp_2=Gu8EditData_2/10; //显示“分”的十位
        Su8Temp_1=Gu8EditData_2%10; //显示“分”的个位
    }

    vGu8Display_Righ_2=Su8Temp_2; //过渡需要显示的数据到底层驱动变量
    vGu8Display_Righ_1=Su8Temp_1; //过渡需要显示的数据到底层驱动变量
}
}

void Wd4(void) //窗口 4。修改“秒”的“3-SS”窗口。
{
    //需要借用的中间变量，用来拆分数数据位。
    static unsigned char Su8Temp_2, Su8Temp_1; //需要借用的中间变量
    static unsigned char Su8BlinkFlag=0; //两种状态的切换判断的中间变量

    if(1==Gu8WdUpdate) //如果需要整屏更新
    {
        Gu8WdUpdate=0; //及时清零，只更新一次显示即可，避免一直进来更新显示

        //属于静态数据，起“装饰”作用，切换窗口后只扫描一次的代码。
        vGu8Display_Righ_4=3; //显示数字“3”
        vGu8Display_Righ_3=11; //显示横杠“-”

        vGu8Display_Righ_Dot_4=0;
        vGu8Display_Righ_Dot_3=0;
        vGu8Display_Righ_Dot_2=0;
        vGu8Display_Righ_Dot_1=0;

        Gu8PartUpdate_1=1; //局部 1 更新显示
    }
}

```



```

if(1==Gu8PartUpdate_1) //局部1 更新显示
{
    Gu8PartUpdate_1=0; //及时清零，只更新一次显示即可，避免一直进来更新显示

    Su8Temp_2=Gu8EditData_3/10; //显示“秒”的十位
    Su8Temp_1=Gu8EditData_3%10; //显示“秒”的个位

    vGu8Display_Righ_2=Su8Temp_2; //过渡需要显示的数据到底层驱动变量
    vGu8Display_Righ_1=Su8Temp_1; //过渡需要显示的数据到底层驱动变量
}

if(0==vGu16BlinkTimerCnt) //某位被选中的数码管跳动闪烁的定时器
{
    vGu8BlinkTimerFlag=0;
    vGu16BlinkTimerCnt=BLINK_TIME; //重设定时器的定时时间
    vGu8BlinkTimerFlag=1;

    if(0==Su8BlinkFlag) //两种状态的切换判断
    {
        Su8BlinkFlag=1;
        Su8Temp_2=10; //10 代表不显示
        Su8Temp_1=10; //10 代表不显示
    }
    else
    {
        Su8BlinkFlag=0;
        Su8Temp_2=Gu8EditData_3/10; //显示“秒”的十位
        Su8Temp_1=Gu8EditData_3%10; //显示“秒”的个位
    }

    vGu8Display_Righ_2=Su8Temp_2; //过渡需要显示的数据到底层驱动变量
    vGu8Display_Righ_1=Su8Temp_1; //过渡需要显示的数据到底层驱动变量
}
}

void KeyScan(void) //按键底层的驱动扫描函数，放在定时中断函数里
{
    static unsigned char Su8KeyShortFlag=0; //按键“短按”触发的标志
    static unsigned char Su8KeyLock1;
    static unsigned int Su16KeyCnt1;
    static unsigned char Su8KeyLock2;
    static unsigned int Su16KeyCnt2;

```

```

static unsigned char Su8KeyLock3;
static unsigned int  Su16KeyCnt3;

//需要详细分析以下这段“短按”与“长按”代码的朋友，请参考第 96 节。
if(0!=KEY_INPUT1)
{
    Su8KeyLock1=0;
    Su16KeyCnt1=0;
    if(1==Su8KeyShortFlag)
    {
        Su8KeyShortFlag=0;
        vGu8KeySec=1;    //触发 K1 的“短按”
    }
}
else if(0==Su8KeyLock1)
{
    Su16KeyCnt1++;

    if(Su16KeyCnt1>=KEY_FILTER_TIME)
    {
        Su8KeyShortFlag=1;
    }

    if(Su16KeyCnt1>=KEY_LONG_TIME)
    {
        Su8KeyLock1=1;
        Su8KeyShortFlag=0;
        vGu8KeySec=4; //触发 K1 的“长按”
    }
}

if(0!=KEY_INPUT2)
{
    Su8KeyLock2=0;
    Su16KeyCnt2=0;
}
else if(0==Su8KeyLock2)
{
    Su16KeyCnt2++;
    if(Su16KeyCnt2>=KEY_FILTER_TIME)
    {
        Su8KeyLock2=1;
        vGu8KeySec=2;
    }
}

```

```

    }

    if(0!=KEY_INPUT3)
    {
        Su8KeyLock3=0;
        Su16KeyCnt3=0;
    }
    else if(0==Su8KeyLock3)
    {
        Su16KeyCnt3++;
        if(Su16KeyCnt3>=KEY_FILTER_TIME)
        {
            Su8KeyLock3=1;
            vGu8KeySec=3;
        }
    }
}

void DisplayScan(void)    //数码管底层的驱动扫描函数，放在定时中断函数里
{
    static unsigned char Su8GetCode;
    static unsigned char Su8ScanStep=1;

    if(0==vGu16ScanTimerCnt)
    {

        P0=0x00;
        P1_0=1;
        P1_1=1;
        P1_2=1;
        P1_3=1;

        switch(Su8ScanStep)
        {
            case 1:
                Su8GetCode=Cu8DigTable[vGu8Display_Righ_1];

                if(1==vGu8Display_Righ_Dot_1)
                {
                    Su8GetCode=Su8GetCode|0x80;
                }

                P0=Su8GetCode;

```

```

        P1_0=0;
        P1_1=1;
        P1_2=1;
        P1_3=1;
        break;

    case 2:
        Su8GetCode=Cu8DigTable[vGu8Display_Righ_2];
        if(1==vGu8Display_Righ_Dot_2)
        {
            Su8GetCode=Su8GetCode|0x80;
        }
        P0=Su8GetCode;
        P1_0=1;
        P1_1=0;
        P1_2=1;
        P1_3=1;
        break;

    case 3:
        Su8GetCode=Cu8DigTable[vGu8Display_Righ_3];
        if(1==vGu8Display_Righ_Dot_3)
        {
            Su8GetCode=Su8GetCode|0x80;
        }
        P0=Su8GetCode;
        P1_0=1;
        P1_1=1;
        P1_2=0;
        P1_3=1;
        break;

    case 4:
        Su8GetCode=Cu8DigTable[vGu8Display_Righ_4];
        if(1==vGu8Display_Righ_Dot_4)
        {
            Su8GetCode=Su8GetCode|0x80;
        }
        P0=Su8GetCode;
        P1_0=1;
        P1_1=1;
        P1_2=1;
        P1_3=0;
        break;

```

```

    }

    Su8ScanStep++;
    if (Su8ScanStep>4)
    {
        Su8ScanStep=1;
    }

    vGu8ScanTimerFlag=0;
    vGu16ScanTimerCnt=SCAN_TIME;
    vGu8ScanTimerFlag=1;
}
}

```

void VoiceScan(void) //蜂鸣器的驱动函数

```

{

    static unsigned char Su8Lock=0;

    if(1==vGu8BeepTimerFlag&&vGu16BeepTimerCnt>0)
    {
        if(0==Su8Lock)
        {
            Su8Lock=1;
            BeepOpen();
        }
        else
        {

            vGu16BeepTimerCnt--;

            if(0==vGu16BeepTimerCnt)
            {
                Su8Lock=0;
                BeepClose();
            }
        }
    }
}

```

void BeepOpen(void)

```

{

```

```

    P3_4=0;
}

void BeepClose(void)
{
    P3_4=1;
}

void T0_time() interrupt 1
{
    VoiceScan();    //蜂鸣器的驱动函数
    KeyScan();      //按键底层的驱动扫描函数
    DisplayScan();  //数码管底层的驱动扫描函数

    if(1==vGu8ScanTimerFlag&&vGu16ScanTimerCnt>0)
    {
        vGu16ScanTimerCnt--; //递减式的软件定时器
    }

    if(1==vGu8BlinkTimerFlag&&vGu16BlinkTimerCnt>0) //数码管闪烁跳动的定时器
    {
        vGu16BlinkTimerCnt--; //递减式的软件定时器
    }

    //在正常工作的窗口下，每 500ms 就定时更新一次显示的软件定时器
    if(1==vGu8UpdateTimerFlag&&vGu16UpdateTimerCnt>0)
    {
        vGu16UpdateTimerCnt--; //递减式的软件定时器
    }

    //时钟实际走的时间的软件定时器，注意，这里是递增式的软件定时器
    if(1==vGu8ClockTimerFlag)
    {
        vGu32ClockTimerCnt++; //递增式的软件定时器
        if(vGu32ClockTimerCnt>=86400000) //86400000 毫秒代表 24 时
        {
            vGu32ClockTimerCnt=0;
        }
    }

    TH0=0xfd; //此参数可根据具体的时间来修改，尽量确保每定时中断一次接近 1ms
    TL0=0x40; //此参数可根据具体的时间来修改，尽量确保每定时中断一次接近 1ms
}

```

```

void SystemInitial(void)
{
    P0=0x00;
    P1_0=1;
    P1_1=1;
    P1_2=1;
    P1_3=1;

    TMOD=0x01;
    TH0=0xfd;    //此参数可根据具体的时间来修改，尽量确保每定时中断一次接近 1ms
    TL0=0x40;    //此参数可根据具体的时间来修改，尽量确保每定时中断一次接近 1ms
    EA=1;
    ET0=1;
    TR0=1;

    //上电初始化一些关键的数据

    Gu8Wd=1;    //窗口 1。开机默认处于正常工作的窗口
    Gu8WdUpdate=1;    //整屏更新变量

    vGu8ClockTimerFlag=0;
    vGu32ClockTimerCnt=43200000;    //43200000 毫秒开机默认 12:00 点。12 时就是 43200000 毫秒
    vGu8ClockTimerFlag=1;    //启动时钟的定时器

    //时钟正常工作的时候，每 500ms 更新显示一次
    vGu16UpdateTimerCnt=500;
    vGu8UpdateTimerFlag=1;    //启动小数点闪烁的定时器
}

void Delay(unsigned long u32DelayTime)
{
    for(;u32DelayTime>0;u32DelayTime--);
}

void PeripheralInitial(void)
{
}

```

## 第一百二十三节： 一种能省去一个 lock 自锁变量的按键驱动程序。

### 【123.1 一种能省去一个 lock 自锁变量的按键驱动程序。】

一位群友给我提到了一个按键的改进建议，能巧妙的省去一个 lock 自锁变量。这个建议引起了我对“变量的分工要专一，一个变量尽量只用在一类事物上，尽量不取巧兼容”的思考。

第一种：带 lock 自锁变量，也是我一直在用的代码。

```
if (0!=KEY_INPUT1) //IO 是高电平，说明按键没有被按下，这时要及时清零一些标志位
{
    Su8KeyLock1=0; //按键解锁
    Su16KeyCnt1=0; //按键去抖动延时计数器清零，此行非常巧妙，是全场的亮点。
}
else if (0==Su8KeyLock1) //有按键按下，且是第一次被按下。这行如果有疑问，请看 92 节的专题分析。
{
    Su16KeyCnt1++; //累加定时中断次数
    if (Su16KeyCnt1>=KEY_FILTER_TIME) //滤波的“稳定时间” KEY_FILTER_TIME，长度是 25ms。
    {
        Su8KeyLock1=1; //按键的自锁, 避免一直触发
        vGu8KeySec=1; //触发 1 号键
    }
}
```

第二种：省略掉一个 lock 自锁变量，群友提出的改进建议。

```
if (0!=KEY_INPUT1)
{
    Su16KeyCnt1=0;
}
else if (Su16KeyCnt1<KEY_FILTER_TIME) //巧妙的利用了 Su16KeyCnt1 等于滤波时间时，只执行一次
{
    Su16KeyCnt1++;
    if (KEY_FILTER_TIME==Su16KeyCnt1) //巧妙的利用了 Su16KeyCnt1 等于滤波时间时，只执行一次
    {
        vGu8KeySec=1;
    }
}
```

分析：

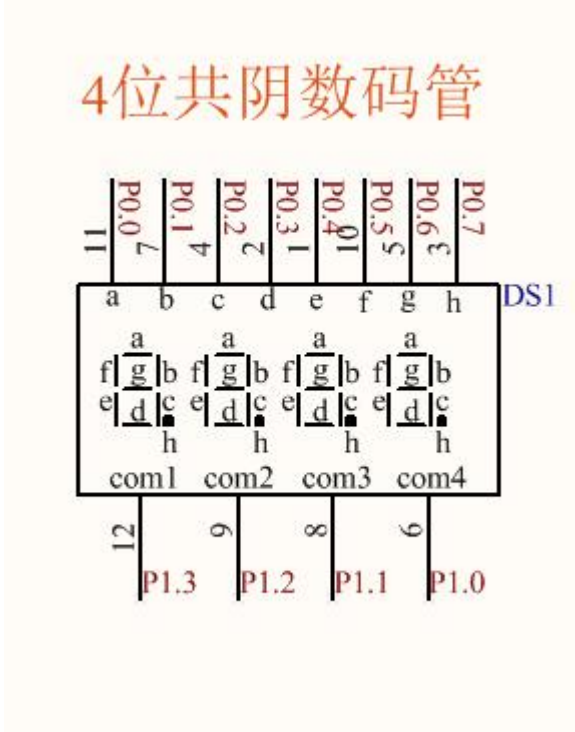
不得不佩服群友的智慧，第二种改进后看起来非常巧妙，犹如蜻蜓点水般轻盈洒脱。但是，为此代码狂欢片刻后，我又有了新的思考和看法。“计时器 Su16KeyCnt1”和“自锁变量 Su8KeyLock1”是两个不同的事物，是两个不同的范畴，就应该用两个不同的变量进行区分。如果逞一时之巧，把两种不同范畴的事物巧妙合并成一个变量，势必会导致程序的“易读性”和“后续维护的可扩展性”大打折扣。“自锁变量 Su8KeyLock1”真的是可有可无吗？假设，如果“计时器 Su16KeyCnt1”的消抖时间 KEY\_FILTER\_TIME 要求等于 0，那么第二种改进后的代码立刻暴露出了问题，行不通。而第一种代码，因为有“自锁变量 Su8KeyLock1”的存在，即使消抖时间 KEY\_FILTER\_TIME 等于 0，也不影响代码功能的完整性，因为第一种代码的理念是“自锁与计



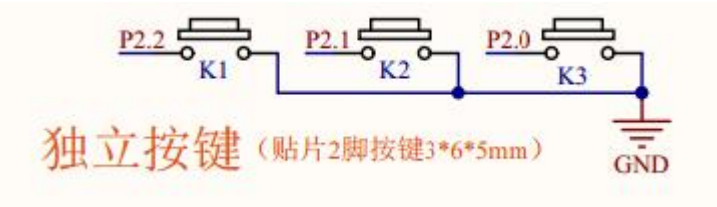
时器是两种不同的功能范畴，用两个不同的变量进行分开隔离，各自管理两种不同的事物，计时器即使为 0 也不影响代码本该有的自锁功能”。通过此例子，给初学者一个建议，在代码的“队形感，易读性，扩展性，分类清晰”和“巧妙，节省代码”两者之间，建议大家优先考虑“队形感，易读性，扩展性，分类清晰”，追求一种原则上的“工整，不出奇兵，扎硬寨，打呆仗，步步为营”，这样阵脚不易乱，能走得更远，驾驭更多千军万马的代码。

第一百二十四节： 数显仪表盘显示“速度、方向、计数器”的跑马灯。

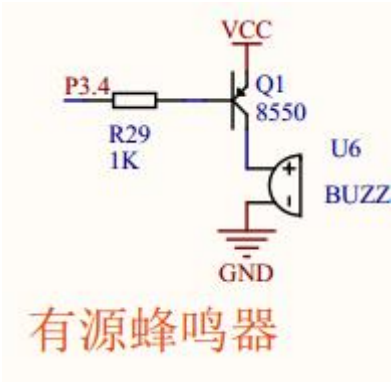
【124.1 数显仪表盘显示“速度、方向、计数器”的跑马灯。】



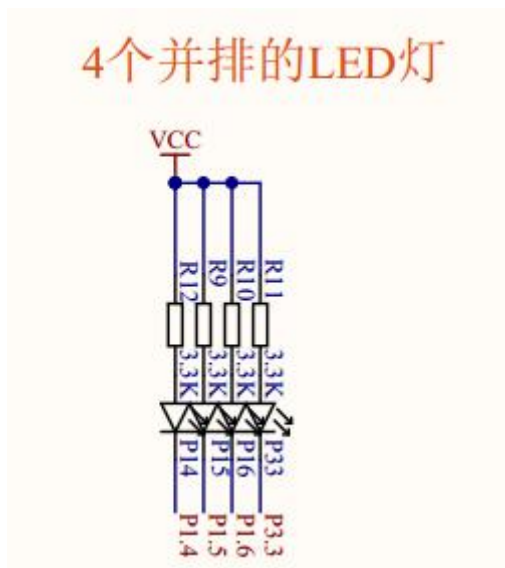
上图 124. 1. 1 数码管



上图 124. 1. 2 独立按键



上图 124. 1. 3 有源蜂鸣器



上图 124.1.4 LED 电路

本节小项目，意在“人机界面”与“过程控制”如何关联的练习。

程序功能如下：

(1) 数码管显示的格式是“S.D.CC”。其中 S 是代表 3 档速度，能显示的数字范围是“1、2、3”，分别代表“慢、中、快”3 档速度。D 代表方向，往右跑显示符号“r”(right 的首字母)，往左跑显示符号“L”(Left 的首字母)。CC 代表计数器，跑马灯每跑完一次，计数器自动加 1，范围是 0 到 99。

(2) 【速度】按键 K1。每按一次【速度】按键 K1，速度档位显示的数字在“1、2、3”之间切换。

(3) 【方向】按键 K2。跑马灯上电后默认处于“往右跑”的方向，默认显示字符“r”。每按一次【方向】按键 K2，跑马灯就在“往右跑”与“往左跑”两个方向之间切换，显示的字符在“r、L”之间切换。

(4) 【启动暂停】按键 K3。上电后，按下【启动暂停】按键 K3 启动之后，跑马灯处于“启动”状态，4 个 LED 灯挨个依次循环的变亮，给人“跑”起来的感觉，此时再按一次【启动暂停】按键 K3，则跑马灯处于“暂停”状态，接着又按一次【启动暂停】按键 K3，跑马灯又变回“启动”状态。因此，【启动暂停】按键 K3 是专门用来切换“启动”和“暂停”这两种状态。

代码如下：

```
#include "REG52.H"

#define KEY_FILTER_TIME  25

#define SCAN_TIME  1
#define VOICE_TIME  50

#define RUN_TIME_SLOW    500    // “慢”档速度的时间参数
#define RUN_TIME_MIDDLE  300    // “中”档速度的时间参数
#define RUN_TIME_FAST    100    // “快”档速度的时间参数

void T0_time();
void SystemInitial(void) ;
```

```

void Delay(unsigned long u32DelayTime) ;
void PeripheralInitial(void) ;

void KeyScan(void);
void KeyTask(void);
void RunTask(void);    //跑马灯的任务函数

void VoiceScan(void);
void DisplayScan(void);
void DisplayTask(void);
void Wd1(void);    //窗口 1。
void BeepOpen(void);
void BeepClose(void);

sbit KEY_INPUT1=P2^2;
sbit KEY_INPUT2=P2^1;
sbit KEY_INPUT3=P2^0;

sbit P1_0=P1^0;
sbit P1_1=P1^1;
sbit P1_2=P1^2;
sbit P1_3=P1^3;

sbit P3_4=P3^4;

//4 个跑马灯的输出口
sbit P1_4=P1^4;
sbit P1_5=P1^5;
sbit P1_6=P1^6;
sbit P3_3=P3^3;

//数码管转换表
code unsigned char Cu8DigTable[]=
{
0x3f,    //0        序号 0
0x06,    //1        序号 1
0x5b,    //2        序号 2
0x4f,    //3        序号 3
0x66,    //4        序号 4
0x6d,    //5        序号 5
0x7d,    //6        序号 6
0x07,    //7        序号 7
0x7f,    //8        序号 8

```

```

0x6f, //9      序号 9
0x00, //不显示 序号 10
0x40, //横杠-  序号 11
0x38, //字符 L  序号 12
0x70, //字符 r  序号 13
};

volatile unsigned char vGu8ScanTimerFlag=0;
volatile unsigned int vGu16ScanTimerCnt=0;

volatile unsigned char vGu8BeepTimerFlag=0;
volatile unsigned int vGu16BeepTimerCnt=0;

unsigned char Gu8Wd=0; //窗口选择变量。人机交互程序框架的支点。
unsigned char Gu8WdUpdate=0; //整屏更新变量。

unsigned char Gu8PartUpdate_1=0; //局部 1 的更新变量,
unsigned char Gu8PartUpdate_2=0; //局部 2 的更新变量
unsigned char Gu8PartUpdate_3=0; //局部 3 的更新变量,

volatile unsigned char vGu8Display_Righ_4=0;
volatile unsigned char vGu8Display_Righ_3=0;
volatile unsigned char vGu8Display_Righ_2=0;
volatile unsigned char vGu8Display_Righ_1=0;

volatile unsigned char vGu8Display_Righ_Dot_4=1; //需要显示的小数点
volatile unsigned char vGu8Display_Righ_Dot_3=1; //需要显示的小数点
volatile unsigned char vGu8Display_Righ_Dot_2=0;
volatile unsigned char vGu8Display_Righ_Dot_1=0;

volatile unsigned char vGu8KeySec=0;

unsigned char Gu8RunCounter=0; //计数器, 范围是 0 到 99

unsigned char Gu8RunStep=0; //运行的步骤
unsigned char Gu8RunStart=0; //控制跑马灯启动的总开关

unsigned char Gu8RunStatus=0; //标识跑马灯当前的状态。0 代表停止, 1 代表启动, 2 代表暂停。
unsigned char Gu8RunDirection=0; //标识跑马灯当前的方向。0 代表往右跑, 1 代表往左跑。
unsigned char Gu8RunSpeed=1; //当前的速度档位。1 代表“慢”, 2 代表“中”, 3 代表“快”。
unsigned int Gu16RunSpeedTimeDate=0; //承接各速度档位的时间参数的变量

volatile unsigned char vGu8RunTimerFlag=0; //用于控制跑马灯跑动速度的定时器
volatile unsigned int vGu16RunTimerCnt=0;

```

```

void main()
{
    SystemInitial();
    Delay(10000);
    PeripheralInitial();
    while(1)
    {
        KeyTask();      //按键的任务函数
        DisplayTask();  //数码管显示的上层任务函数
        RunTask();      //跑马灯的任务函数
    }
}

void RunTask(void)      //跑马灯的任务函数，放在主函数内
{
    if(0==Gu8RunStart) //如果是停止的状态
    {
        return; //如果是停止的状态，退出当前函数，不扫描余下代码。
    }

    switch(Gu8RunStep) //屡见屡爱的 switch 又来了
    {
        case 0:
            vGu8RunTimerFlag=0;
            vGu16RunTimerCnt=0; //定时器清零
            Gu8RunStep=1; //切换到下一步，启动

            break;
        case 1:
            if(1==Gu8RunStatus&&0==vGu16RunTimerCnt) //当前处于“启动”状态，并且定时器等于 0
            {
                P1_4=0; //第 1 个灯亮
                P1_5=1; //第 2 个灯灭
                P1_6=1; //第 3 个灯灭
                P3_3=1; //第 4 个灯灭

                vGu8RunTimerFlag=0;
                vGu16RunTimerCnt=Gu16RunSpeedTimeDate; //速度时间参数变量的大小，决定了速度
                vGu8RunTimerFlag=1; //启动定时器

                //灵活切换“步骤变量”
                if(0==Gu8RunDirection) //往右跑
                {

```

```

        Gu8RunStep=2;
    }
    else //往左跑
    {
        if(Gu8RunCounter<99)
        {
            Gu8RunCounter++; //往左边跑完一次，运行的计数器自加 1
        }
        Gu8PartUpdate_3=1; //局部 3 的更新变量，更新显示计数器

        Gu8RunStep=4;
    }

}

break;
case 2:
    if(1==Gu8RunStatus&&0==vGu16RunTimerCnt) //当前处于“启动”状态，并且定时器等于 0
    {
        P1_4=1; //第 1 个灯灭
        P1_5=0; //第 2 个灯亮
        P1_6=1; //第 3 个灯灭
        P3_3=1; //第 4 个灯灭

        vGu8RunTimerFlag=0;
        vGu16RunTimerCnt=Gu16RunSpeedTimeDate; //速度时间参数变量的大小，决定了速度
        vGu8RunTimerFlag=1; //启动定时器

        //灵活切换“步骤变量”
        if(0==Gu8RunDirection) //往右跑
        {
            Gu8RunStep=3;
        }
        else //往左跑
        {
            Gu8RunStep=1;
        }
    }

    break;
case 3:
    if(1==Gu8RunStatus&&0==vGu16RunTimerCnt) //当前处于“启动”状态，并且定时器等于 0
    {
        P1_4=1; //第 1 个灯灭
    }

```

```

P1_5=1;    //第 2 个灯灭
P1_6=0;    //第 3 个灯亮
P3_3=1;    //第 4 个灯灭

vGu8RunTimerFlag=0;
vGu16RunTimerCnt=Gu16RunSpeedTimeDate; //速度时间参数变量的大小，决定了速度
vGu8RunTimerFlag=1;    //启动定时器

//灵活切换“步骤变量”
if (0==Gu8RunDirection) //往右跑
{
    Gu8RunStep=4;
}
else //往左跑
{
    Gu8RunStep=2;
}
}

break;
case 4:
if (1==Gu8RunStatus&&0==vGu16RunTimerCnt) //当前处于“启动”状态，并且定时器等于 0
{
    P1_4=1;    //第 1 个灯灭
    P1_5=1;    //第 2 个灯灭
    P1_6=1;    //第 3 个灯灭
    P3_3=0;    //第 4 个灯亮

    vGu8RunTimerFlag=0;
    vGu16RunTimerCnt=Gu16RunSpeedTimeDate; //速度时间参数变量的大小，决定了速度
    vGu8RunTimerFlag=1;    //启动定时器

    //灵活切换“步骤变量”
    if (0==Gu8RunDirection) //往右跑
    {
        if (Gu8RunCounter<99)
        {
            Gu8RunCounter++; //往右边跑完一次，运行的计数器自加 1
        }
        Gu8PartUpdate_3=1;    //局部 3 的更新变量，更新显示计数器

        Gu8RunStep=1;
    }
    else //往左跑

```



```

        {
            Gu8RunStep=3;
        }
    }

    break;

}

}

void KeyTask(void)    //按键的任务函数
{
    if (0==vGu8KeySec)
    {
        return;
    }

    switch(vGu8KeySec)
    {
        case 1:    //【速度】按键 K1
            switch(Gu8Wd) //在某个窗口下
            {
                case 1:    //窗口 1。
                    //每按一次 K1 按键，Gu8RunSpeed 就在 1、2、3 三者之间切换，
                    //并且根据 Gu8RunSpeed 的数值，对 Gu16RunSpeedTimeDate 赋值
                    //不同的速度时间参数，从而控制速度档位。

                    if(1==Gu8RunSpeed)
                    {
                        Gu8RunSpeed=2;  //“中”档
                        Gu16RunSpeedTimeDate=RUN_TIME_MIDDLE; //赋值“中”档的时间参数
                    }
                    else if(2==Gu8RunSpeed)
                    {
                        Gu8RunSpeed=3;  //“快”档
                        Gu16RunSpeedTimeDate=RUN_TIME_FAST; //赋值“快”档的时间参数
                    }
                    else
                    {
                        Gu8RunSpeed=1;  //“慢”档
                        Gu16RunSpeedTimeDate=RUN_TIME_SLOW; //赋值“慢”档的时间参数
                    }
                }
            }
        }
    }
}

```

```

        Gu8PartUpdate_1=1;    //局部 1 的更新变量，更新显示“速度”

        vGu8BeepTimerFlag=0;
        vGu16BeepTimerCnt=VOICE_TIME;    //蜂鸣器发出“滴”一声
        vGu8BeepTimerFlag=1;
        break;
    }

    vGu8KeySec=0;
    break;

case 2:    //【方向】按键 K2

    switch(Gu8Wd) //在某个窗口下
    {
        case 1:    //窗口 1。
            //每按一次 K2 按键，Gu8RunDirection 就在 0 和 1 之间切换, 从而控制方向
            if(0==Gu8RunDirection)
            {
                Gu8RunDirection=1;
            }
            else
            {
                Gu8RunDirection=0;
            }

            Gu8PartUpdate_2=1;    //局部 2 更新显示，更新显示“方向”

            vGu8BeepTimerFlag=0;
            vGu16BeepTimerCnt=VOICE_TIME;    //蜂鸣器发出“滴”一声
            vGu8BeepTimerFlag=1;
            break;
    }

    vGu8KeySec=0;
    break;

case 3:    //【启动暂停】按键 K3
    switch(Gu8Wd) //在某个窗口下
    {
        case 1:    //窗口 1。
            if(0==Gu8RunStatus) //当跑马灯处于“停止”状态时
            {
                Gu8RunStep=0;    //运行步骤从 0 开始
                Gu8RunStart=1;    //总开关“打开”。
            }
    }

```

```

        Gu8RunStatus=1; //状态切换到“启动”状态
    }
    else if(1==Gu8RunStatus) //当跑马灯处于“启动”状态时
    {
        Gu8RunStatus=2; //状态切换到“暂停”状态
    }
    else //当跑马灯处于“暂停”状态时
    {
        Gu8RunStatus=1; //状态切换到“启动”状态
    }

    vGu8BeepTimerFlag=0;
    vGu16BeepTimerCnt=VOICE_TIME; //蜂鸣器发出“滴”一声
    vGu8BeepTimerFlag=1;
    break;
}

vGu8KeySec=0;
break;
}
}

void DisplayTask(void) //数码管显示的上层任务函数
{
    switch(Gu8Wd) //以窗口选择 Gu8Wd 为支点，去执行对应的窗口显示函数。又一次用到 switch 语句
    {
        case 1:
            Wd1(); //窗口 1。
            break;
    }
}

void Wd1(void) //窗口 1。
{
    //需要借用的中间变量，用来拆分数数据位。
    static unsigned char Su8Temp_4, Su8Temp_3, Su8Temp_2, Su8Temp_1; //需要借用的中间变量

    if(1==Gu8WdUpdate) //如果需要整屏更新
    {
        Gu8WdUpdate=0; //及时清零，只更新一次显示即可，避免一直进来更新显示

        //属于静态数据，起“装饰”作用，切换窗口后只扫描一次的代码。
        vGu8Display_Righ_Dot_4=1; //显示小数点
        vGu8Display_Righ_Dot_3=1; //显示小数点
    }
}

```

```

vGu8Display_Righ_Dot_2=0;
vGu8Display_Righ_Dot_1=0;

Gu8PartUpdate_1=1; //局部 1 更新显示
Gu8PartUpdate_2=1; //局部 2 更新显示
Gu8PartUpdate_3=1; //局部 3 更新显示
}

if(1==Gu8PartUpdate_1) //局部 1 更新显示，速度
{
    Gu8PartUpdate_1=0; //及时清零，只更新一次显示即可，避免一直进来更新显示

    Su8Temp_4=Gu8RunSpeed;

    vGu8Display_Righ_4=Su8Temp_4; //过渡需要显示的数据到底层驱动变量
}

if(1==Gu8PartUpdate_2) //局部 2 更新显示，方向
{
    Gu8PartUpdate_2=0; //及时清零，只更新一次显示即可，避免一直进来更新显示

    if(0==Gu8RunDirection) //往右跑
    {
        Su8Temp_3=13; //数码管的字模转换表序号 13 代表显示字符“r”
    }
    else
    {
        Su8Temp_3=12; //数码管的字模转换表序号 12 代表显示字符“L”
    }
    vGu8Display_Righ_3=Su8Temp_3; //过渡需要显示的数据到底层驱动变量
}

if(1==Gu8PartUpdate_3) //局部 3 更新显示，计数器
{
    Gu8PartUpdate_3=0; //及时清零，只更新一次显示即可，避免一直进来更新显示

    Su8Temp_2=Gu8RunCounter%100/10; //提取十位
    Su8Temp_1=Gu8RunCounter%10/1; //提取个位

    vGu8Display_Righ_2=Su8Temp_2; //过渡需要显示的数据到底层驱动变量
    vGu8Display_Righ_1=Su8Temp_1; //过渡需要显示的数据到底层驱动变量
}
}

```

```

void KeyScan(void)  //按键底层的驱动扫描函数，放在定时中断函数里
{
    static unsigned char Su8KeyLock1;
    static unsigned int  Su16KeyCnt1;
    static unsigned char Su8KeyLock2;
    static unsigned int  Su16KeyCnt2;
    static unsigned char Su8KeyLock3;
    static unsigned int  Su16KeyCnt3;

    if(0!=KEY_INPUT1)
    {
        Su8KeyLock1=0;
        Su16KeyCnt1=0;
    }
    else if(0==Su8KeyLock1)
    {
        Su16KeyCnt1++;
        if(Su16KeyCnt1>=KEY_FILTER_TIME)
        {
            Su8KeyLock1=1;
            vGu8KeySec=1;
        }
    }

    if(0!=KEY_INPUT2)
    {
        Su8KeyLock2=0;
        Su16KeyCnt2=0;
    }
    else if(0==Su8KeyLock2)
    {
        Su16KeyCnt2++;
        if(Su16KeyCnt2>=KEY_FILTER_TIME)
        {
            Su8KeyLock2=1;
            vGu8KeySec=2;
        }
    }

    if(0!=KEY_INPUT3)
    {

```

```

        Su8KeyLock3=0;
        Su16KeyCnt3=0;
    }
    else if(0==Su8KeyLock3)
    {
        Su16KeyCnt3++;
        if(Su16KeyCnt3>=KEY_FILTER_TIME)
        {
            Su8KeyLock3=1;
            vGu8KeySec=3;
        }
    }
}

void DisplayScan(void)    //数码管底层的驱动扫描函数，放在定时中断函数里
{
    static unsigned char Su8GetCode;
    static unsigned char Su8ScanStep=1;

    if(0==vGu16ScanTimerCnt)
    {

        P0=0x00;
        P1_0=1;
        P1_1=1;
        P1_2=1;
        P1_3=1;

        switch(Su8ScanStep)
        {
            case 1:
                Su8GetCode=Cu8DigTable[vGu8Display_Righ_1];

                if(1==vGu8Display_Righ_Dot_1)
                {
                    Su8GetCode=Su8GetCode|0x80;
                }
                P0=Su8GetCode;
                P1_0=0;
                P1_1=1;
                P1_2=1;
                P1_3=1;
            }
        }
    }
}

```

```

        break;

    case 2:
        Su8GetCode=Cu8DigTable[vGu8Display_Righ_2];
        if(1==vGu8Display_Righ_Dot_2)
        {
            Su8GetCode=Su8GetCode|0x80;
        }
        P0=Su8GetCode;
        P1_0=1;
        P1_1=0;
        P1_2=1;
        P1_3=1;
        break;

    case 3:
        Su8GetCode=Cu8DigTable[vGu8Display_Righ_3];
        if(1==vGu8Display_Righ_Dot_3)
        {
            Su8GetCode=Su8GetCode|0x80;
        }
        P0=Su8GetCode;
        P1_0=1;
        P1_1=1;
        P1_2=0;
        P1_3=1;
        break;

    case 4:
        Su8GetCode=Cu8DigTable[vGu8Display_Righ_4];
        if(1==vGu8Display_Righ_Dot_4)
        {
            Su8GetCode=Su8GetCode|0x80;
        }
        P0=Su8GetCode;
        P1_0=1;
        P1_1=1;
        P1_2=1;
        P1_3=0;
        break;
}

Su8ScanStep++;

```

```

        if (Su8ScanStep>4)
        {
            Su8ScanStep=1;
        }

        vGu8ScanTimerFlag=0;
        vGu16ScanTimerCnt=SCAN_TIME;
        vGu8ScanTimerFlag=1;
    }
}

```

void VoiceScan(void) //蜂鸣器的驱动函数

```

{

    static unsigned char Su8Lock=0;

    if(1==vGu8BeepTimerFlag&&vGu16BeepTimerCnt>0)
    {
        if(0==Su8Lock)
        {
            Su8Lock=1;
            BeepOpen();
        }
        else
        {

            vGu16BeepTimerCnt--;

            if(0==vGu16BeepTimerCnt)
            {
                Su8Lock=0;
                BeepClose();
            }

        }
    }
}

```

void BeepOpen(void)

```

{
    P3_4=0;
}

```

void BeepClose(void)



```

{
    P3_4=1;
}

void T0_time() interrupt 1
{
    VoiceScan();    //蜂鸣器的驱动函数
    KeyScan();      //按键底层的驱动扫描函数
    DisplayScan();  //数码管底层的驱动扫描函数

    if(1==vGu8ScanTimerFlag&&vGu16ScanTimerCnt>0)
    {
        vGu16ScanTimerCnt--;
    }

    if(1==vGu8RunTimerFlag&&vGu16RunTimerCnt>0) //用于控制跑马灯跑动速度的定时器
    {
        vGu16RunTimerCnt--;
    }

    TH0=0xf0;    //此参数可根据具体的时间来修改，尽量确保每定时中断一次接近 1ms
    TL0=0x40;    //此参数可根据具体的时间来修改，尽量确保每定时中断一次接近 1ms
}

void SystemInitial(void)
{
    P0=0x00;
    P1_0=1;
    P1_1=1;
    P1_2=1;
    P1_3=1;

    TMOD=0x01;
    TH0=0xf0;    //此参数可根据具体的时间来修改，尽量确保每定时中断一次接近 1ms
    TL0=0x40;    //此参数可根据具体的时间来修改，尽量确保每定时中断一次接近 1ms
    EA=1;
    ET0=1;
    TR0=1;

    //上电初始化一些关键的数据

    Gu8Wd=1;    //窗口 1。开机默认处于正常工作的窗口
    Gu8WdUpdate=1; //整屏更新变量
    //跑马灯处于初始化的状态

```

```

P1_4=0;    //第 1 个灯亮
P1_5=1;    //第 2 个灯灭
P1_6=1;    //第 3 个灯灭
P3_3=1;    //第 4 个灯灭

//根据当前的速度档位 Gu8RunSpeed, 来初始化速度时间参数 Gu16RunSpeedTimeDate
if (1==Gu8RunSpeed)
{
    Gu16RunSpeedTimeDate=RUN_TIME_SLOW; //赋值“慢”档的时间参数
}
else if (2==Gu8RunSpeed)
{
    Gu16RunSpeedTimeDate=RUN_TIME_MIDDLE; //赋值“中”档的时间参数
}
else
{
    Gu16RunSpeedTimeDate=RUN_TIME_FAST; //赋值“快”档的时间参数
}
}

void Delay(unsigned long u32DelayTime)
{
    for(;u32DelayTime>0;u32DelayTime--);
}

void PeripheralInitial(void)
{
}

```

## 第一百二十五节：“双线”的肢体接触通信。

### 【125.1 “双线”的肢体接触通信。】

芯片之间通信，都离不开“数据信号”和“时钟信号”，缺一不可。“数据信号”和“时钟信号”是什么关系，它们是怎样相互配合来实现通信的功能？其实原理也很简单。打个比喻，甲乙两个人，规定只能靠一只“手”和一只“脚”进行肢体接触的通信，他们之间如何传输数据？“手”可以产生“两种”状态“握紧”和“松开”，“脚”可以产生“一种”状态“踢一脚”。他们之间约定，甲发送数据给乙，乙每被甲“踢一脚”就去记录一次手的状态是“握紧”还是“松开”，“握紧”代表二进制的 0，“松开”代表二进制的 1，这样，如果他们之间想传输一个字节的十六进制数据 0x59，只需把十六进制的数据 0x59 展开成二进制 01011001，从右到左（从低位到高位）以“位”为单位挨个发送，过程如下：

第一次“踢一脚”：手的状态是“松开”，记录 1。

第二次“踢一脚”：手的状态是“握紧”，记录 0。

第三次“踢一脚”：手的状态是“握紧”，记录 0。

第四次“踢一脚”：手的状态是“松开”，记录 1。

第五次“踢一脚”：手的状态是“松开”，记录 1。

第六次“踢一脚”：手的状态是“握紧”，记录 0。

第七次“踢一脚”：手的状态是“松开”，记录 1。

第八次“踢一脚”：手的状态是“握紧”，记录 0。

上述肢体接触的通信过程，其实一只“手”就代表了一根“数据线”，可以产生高电平“1”和低电平“0”这两种状态，而一只“脚”代表了一根“时钟线”，但是“踢一脚”代表了“时钟线”上的一种什么状态呢？注意，“踢一脚”既不是高电平“1”也不是低电平“0”，而是瞬间只产生一次的“上升沿”或者“下降沿”。何谓“上升沿”何谓“下降沿”？“上升沿”是代表“时钟线从低电平跳变到高电平的瞬间”，“下降沿”是代表“时钟线从高电平跳变到低电平的瞬间”。“踢一脚”、“上升沿”、“下降沿”此三者都可以统一理解成“节拍”。

芯片之间通信，“时钟信号”只需 1 个足矣，而“数据信号”却可以不止 1 个。1 个“数据信号”往往叫“串行”通信，一个节拍只能传输 1 位数据。8 个以上并列的“数据信号”往往叫“并行”通信，一个节拍能传输 8 位以上的数据。可见，并行的“数据信号”越多，传输的速率越快。

常见的系统中，串口，IIC，SPI，USB，CAN 这类都是“串行”通信。而 32 位单片机与外部的 nandflash, norflash, sdram, sram 这些芯片通信往往是“并行”通信，并行的数据信号多达 8 个 16 个甚至 32 个。

本节标题之所以强调“双线”，是因为“手”代表数据线，“脚”代表时钟线，一共两条线因此为“双线”。现在把上述的肢体通信过程翻译成 C 语言代码，甲发送数据的代码如下：

```
sbit Hand_DATA=P2^6; //手的数据线
sbit Foot_CLK=P2^7;  //脚的时钟线

void SendByte(unsigned char u8Data) //甲发送数据的发送函数
{
    static unsigned char i;
    for(i=0;i<8;i++) //一个字节包含 8 个位数据，需要循环 8 次
    {
        if(0==(u8Data&0x01)) //根据数据的每一位状态，发送对应的位数据。
```

```
{
    Hand_DATA=0;  //0 代表“握紧”
}
else
{
    Hand_DATA=1;  //1 代表“松开”
}

Foot_CLK=1;
Delay(); //为产生均匀的脉冲节拍，时钟线的高电平先延时一会
Foot_CLK=0;  //从高电平跳变到低电平，产生瞬间的“下降沿”，代表“踢一脚”
Delay(); //为产生均匀的脉冲节拍，时钟线的低电平先延时一会

u8Data=u8Data>>1; //右移一位，为即将发送下一位做准备
}
}
```

## 第一百二十六节：“单线”的肢体接触通信。

### 【126.1 同步通信与异步通信。】

既然芯片之间通信离不开“数据”和“时钟”这两种信号，那么是不是说，通信必须至少两根线（双线）以上？不是。单线也可以通信，继续拿甲乙两人的肢体通信做比喻，这一次只允许用一只“手”不许用“脚”，“手”继续做数据信号，那么时钟信号在哪？时钟信号在甲乙两人各自的“心跳”。用两个人的“心跳”作为时钟信号就有两个时钟节拍，初学者可能在这里会有疑惑，这两人的“心跳”频率可能不一致，时钟节拍可能不同步，怎么能进行通信呢？说到这里，恰好通讯界有两个专业的概念，一个是“同步通信”另一个是“异步通信”。像上一节讲那种用脚的动作“踢一脚”作为时钟信号，这个时钟信号只有一个，对于通讯的甲乙双方是实时“同步的”时钟信号，因此这种通信叫做“同步通信”。而本节提到的用两个人各自的“心跳”做时钟信号，有两个时钟源，时钟信号是“不同步的”，这种通信叫做“异步通信”。

### 【126.2 异步通信的原理。】

既然两人各自的“心跳”不同步（异步），而且“心跳”是从甲乙两人出生开始就一直持续存在不停跳动的，那么发送一个字节的数据是从什么时候开始到什么时候结束就必须事先有一个约定。他们是这样约定的：

（一）平时的待命状态。甲是发送方，乙是接收方，平时待命没有发送数据的时候，甲手的状态一直是“松开”的（电平1）。

（二）1个开始位与8个数据位。当甲要发送数据给乙的时候，第1个心跳甲先“握紧”（电平0）代表“开始位”，“开始位”用来通知乙方请做好接收数据的准备，然后第2个到第9个心跳甲依次靠手的状态发送8个位的字节数据（数据位），乙方因为“甲的开始位”的存在已经做好了接收第2个心跳数据的准备，因此乙方能完全接收第2个心跳至第9个心跳的数据位的数据。

（三）1个停止位。甲发送了第9个心跳的数据后，必须马上恢复到待命的状态“松开”（电平1），以便为下一次发送数据时能正确发送“开始位”，但是这个待命的状态“松开”至少应该持续多长的时间呢？至少持续1个“心跳”的时间以上。这样，虽然两个人的“心跳”不同步并且频率也不一样，但是只要8个“心跳”的累加误差不超过1个“心跳”的停止位时间，数据就肯定不会错位。这个至少持续1个“心跳”的待命状态就起到消除累加误差的作用。

### 【126.3 异步的肢体通信的例子。】

“手”可以产生“两种”状态“握紧”和“松开”，甲发送数据给乙，乙每“心跳”一次就去判断一次手的状态，“握紧”代表二进制的0，“松开”代表二进制的1，这样，如果他们之间想传输一个字节的十六进制数据0x59，只需把十六进制的数据0x59展开成二进制01011001，从右到左（从低位到高位）以“位”为单位挨个发送，过程如下：

平时手的状态一直处于“松开”的待命状态，直到手第一次出现“握紧”的状态.....

第一次“心跳”：手的状态是“握紧”，开始位，通知乙作好接收即将过来的8个“心跳”数据位。

第二次“心跳”：手的状态是“松开”，数据位bit0，记录1。

第三次“心跳”：手的状态是“握紧”，数据位bit1，记录0。

第四次“心跳”：手的状态是“握紧”，数据位bit2，记录0。

第五次“心跳”：手的状态是“松开”，数据位bit3，记录1。

第六次“心跳”：手的状态是“松开”，数据位bit4，记录1。

第七次“心跳”：手的状态是“握紧”，数据位bit5，记录0。

第八次“心跳”：手的状态是“松开”，数据位 bit6，记录 1。

第九次“心跳”：手的状态是“握紧”，数据位 bit7，记录 0。

第十次“心跳”：手的状态是“松开”，停止位，至少持续 1 个“心跳”的待命状态。

现在把上述的“单线”（异步）的肢体通信过程翻译成 C 语言代码，甲发送数据的代码如下：

```
sbit Hand_DATA=P2^6; //手的数据线

void SendByte(unsigned char u8Data) //甲发送数据的发送函数
{
    static unsigned char i;

    Hand_DATA=0; //开始位。0 代表“握紧”
    Delay(); //甲的心跳间隔时间
    for(i=0;i<8;i++) //发送 8 个数据位
    {
        if(0==(u8Data&0x01)) //根据数据的每一位状态，发送对应的位数据。
        {
            Hand_DATA=0; //0 代表“握紧”
        }
        else
        {
            Hand_DATA=1; //1 代表“松开”
        }

        Delay(); //甲的心跳间隔时间

        u8Data=u8Data>>1; //右移一位，为即将发送下一位做准备
    }
    Hand_DATA=1; //停止位。1 代表“松开”
    Delay(); //甲的心跳间隔时间
}
```

## 第一百二十七节： 单片机串口接收数据的机制。

### 【127.1 单片机串口接收数据的底层时序。】

上一节“单线的肢体接触通信”其实是为本节打基础的，通信线只用了一根“数据”线，没有用到“时钟”线，属于异步通信方式，还分析时序中的“1 个开始位，8 个数据位，1 个停止位”等细节内容，这些时序其实就是本节单片机串口通信的底层时序，一模一样。继续上一节的内容（很有必要重新温习一次上一节的异步通信原理），继续沿用甲乙双方靠各自“心跳”的节拍来异步通信的例子，本节单片机串口接收数据是代表乙方，我把乙方串口接收数据的过程翻译成 C 语言，代码如下：

```
sbit USART_RX=P3^0; //用来接收串口数据的数据线
unsigned char Gu8ReceiveData=0; //串口接收到的 8 位数据
unsigned char i; //连续接收 8 位数据的循环变量
void main()
{
    Gu8ReceiveData=0;
    while(1)
    {
        USART_RX=1; //51 单片机的规则，每次读取数据前都执行一条“置 1”指令
        Delay(); //乙的心跳间隔时间，待机时，每一个节拍监控一次数据线的状态
        if(0==USART_RX) //如果监控到甲发送的“开始位 0”，从下一个节拍开始连续接收 8 位数据
        {
            for(i=0;i<8;i++) //连续循环接收 8 个“数据位”
            {
                USART_RX=1; //51 单片机的规则，每次读取数据前都执行一条“置 1”指令
                Delay(); //乙的心跳间隔时间，每个节拍判断读取一位数据
                if(1==USART_RX) //判断读取数据线上的状态
                {
                    Gu8ReceiveData=Gu8ReceiveData | 0x80;
                }
                else
                {
                    Gu8ReceiveData=Gu8ReceiveData & 0x7F;
                }
                Gu8ReceiveData=Gu8ReceiveData>>1; //右移一位，为即将接收下一位做准备
            }
            Delay(); //乙的心跳间隔时间，这里额外增加一个节拍，作为“停止位”的开销。
        }
    }
}
```

### 【127.2 单片机内置的“硬件串口模块”。】

很显然，上面【127.1】分享的时序代码会占用单片机大量的时间，单片机每接收一个字节的數據都会被束缚一次手脚，耽误了其它大事，怎么办？为了把单片机从底层繁琐的时序中解放出来，单片机内置了很多“硬货”，俗称“硬件资源”，“硬件串口模块”便是其中之一。何谓“硬件”，单片机内置的“硬件”可以看作是另外一个独立运行的“核”，这个“核”可以看作是另外一个 CPU，可以独立工作，相当于单片机主人在某个领域的一个专用助手。单片机只需要跟这个“核”通信发指令就可以，具体的执行过程由这个“核”独立去完成，这个“核”完成工作之后再把处理结果反馈给单片机。那么，单片机是如何跟这些内置“硬件资源”通信呢？其实它们的通信接口是“寄存器”，不管是单片机给“硬件资源”发送指令，还是单片机从“硬件资源”里读取所需要的结果数据，都是通过“寄存器”来完成。

### 【127.3 单片机与硬件串口通信的接口“寄存器”。】

硬件串口的寄存器主要涉及：串口的方式选择，波特率，允许串口接收数据，中断的优先级，中断的允许，等等。比如，51 单片机的串口是兼容很多种方式的，可以同步通信，也可以异步通信，异步通信还可以兼容 10 位（1 开始位、8 数据位、1 停止），11 位（1 开始位、8 数据位、1 校验位、1 停止），等等，这些就是多选题，我们要在某个特定的寄存器里面做出选择。波特率，是用来衡量通信的速度，比如波特率是 9600，就意味着 1 秒钟能收发 9600 个二进制的位数据，也就是 1 秒钟能产生 9600 个时钟节拍，波特率越高通信的速度越快，这些也需要我们往相关的寄存器填入相应的数据，来告知“硬件串口”以哪种波特率进行通信。

那么，对于初学者，寄存器如何配置呢？主要有这些思路：查看芯片手册（datasheet），产看 C 编译器的手册，查看芯片相关的 C 语言的头文件（比如 51 单片机的 REG.H），在网上参考别人已经配置好的代码，或者购买相关芯片的学习板时所配套的程序例程。

本节用到的串口，是 10 位数据长度的异步通信，波特率 9600，相关配置的代码如下：

```
unsigned char u8_TMOD_Temp=0;

//串口的波特率与内置的定时器 1 直接相关，因此配置此定时器 1 就等效于配置波特率。
u8_TMOD_Temp=0x20; //即将把定时器 1 设置为：工作方式 2，初值自动重装的 8 位定时器。
TMOD=TMOD&0x0f; //此寄存器低 4 位是跟定时器 0 相关，高 4 位是跟定时器 1 相关。先清零定时器 1。
TMOD=TMOD|u8_TMOD_Temp; //往高 4 位的定时器 1 填入 0x2，低 4 位的定时器 0 保持不变。
TH1=256-(11059200L/12/32/9600); //波特率为 9600。11059200 代表晶振 11.0592MHz，
TL1=256-(11059200L/12/32/9600); //L 代表 long 的长类型数据。根据芯片手册提供的计算公式。
TR1=1; //开启定时器 1

SM0=0;
SM1=1; //SM0 与 SM1 的设置：选择 10 位异步通信，波特率根据定时器 1 可变
REN=1; //允许串口接收数据

//为了保证串口中断接收的数据不丢失，必须设置 IP = 0x10，相当于把串口中断设置为最高优先级，
//这个时候，串口中断可以打断任何其他的中断服务函数，实现嵌套的功能，
IP =0x10; //把串口中断设置为最高优先级，必须的。

ES=1; //允许串口中断
EA=1; //允许总中断
```

### 【127.4 硬件串口的中断函数。】



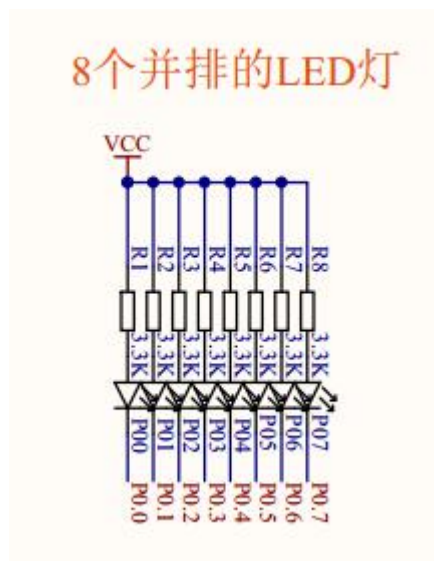
硬件串口接收完一个字节的数据之后，会及时产生一个串口中断去通知单片机接收数据。单片机在串口中断函数里直接读取“串口专用缓存寄存器”SBUF 的数据，就可以直接获得一个完整的 8 位宽度的数据，省去了繁琐的驱动时序底层。

串口的中断函数跟定时器的中断函数很类似，只不过中断号不一样而已，比如我们前面章节用的定时器 0 的中断号是“1”，而本节串口的中断号是“4”。这些其实是 C 编译器定的游戏规则，我们只要根据它提供的数据手册遵守它的游戏规则就好了。串口中断函数里还有一个地方要注意，硬件串口“接收完一个字节”的数据后产生一次中断，而硬件串口“发送完一个字节”的数据后也产生一次中断，这两个一“收”一“发”的中断都是共用中断号为“4”的中断函数，因此，我们必须在中断函数里通过判断寄存器的 RI 和 TI 的标志位来判断到底是“收”的中断，还是“发”的中断，并且软件上要及时把 RI 或者 TI 及时清零，避免不断进入中断的情况。参考代码如下：

```
unsigned char Gu8ReceiveData=0; //接收到一个字节的数据

void usart(void) interrupt 4 //串口接发的中断，中断号为 4
{
    if(1==RI) //接收完一个字节后引起的中断
    {
        RI = 0; //及时清零，避免一直无缘无故的进入中断。
        Gu8ReceiveData=SBUF; //直接读取“串口专用缓存寄存器”SBUF 的 8 位数据。
    }
    else //发送数据引起的中断
    {
        TI = 0; //及时清除发送中断的标志，避免一直无缘无故的进入中断。
        //以下可以添加一个全局变量的标志位的相关代码，通知主函数已经发送完一个字节的数据了。
    }
}
```

## 【127.5 上位机与单片机的串口通信例程。】



上图 127.5.1 灌入式驱动 8 个 LED

程序功能如下：

波特率 9600，校验位 NONE（无），数据位 8，停止位 1。在上位机的串口助手里，发送一个十六进制（HEX 模式）的 01，让单片机的一颗 LED “亮”。发送一个十六进制（HEX 模式）的 00，让单片机的一颗 LED “灭”。上位机的串口助手的使用，请参考前面第 11 节的相关内容，或者自己在网上查找串口助手软件的使用方法，串口助手软件网上很多，我们的使用要求不高，随便选用一家都可以。代码如下：

```
#include "REG52.H"

void usart(void);

sbit P0_0=P0^0; //一颗 LED 灯

unsigned char Gu8ReceiveData=0; //接收到一个字节的数据

void main()
{
    unsigned char u8_TMOD_Temp=0;

    P0_0=1; //LED 灭。1 代表 LED 灭， 0 代表亮

    //串口的波特率与内置的定时器 1 直接相关，因此配置此定时器 1 就等效于配置波特率。
    u8_TMOD_Temp=0x20; //即将把定时器 1 设置为：工作方式 2，初值自动重装的 8 位定时器。
    TMOD=TMOD&0x0f; //此寄存器低 4 位是跟定时器 0 相关，高 4 位是跟定时器 1 相关。先清零定时器 1。
    TMOD=TMOD|u8_TMOD_Temp; //把高 4 位的定时器 1 填入 0x2，低 4 位的定时器 0 保持不变。
    TH1=256-(11059200L/12/32/9600); //波特率为 9600。11059200 代表晶振 11.0592MHz，
    TL1=256-(11059200L/12/32/9600); //L 代表 long 的长类型数据。根据芯片手册提供的计算公式。
    TR1=1; //开启定时器 1
```

```

SM0=0;
SM1=1; //SM0 与 SM1 的设置：选择 10 位异步通信，波特率根据定时器 1 可变
REN=1; //允许串口接收数据

//为了保证串口中断接收的数据不丢失，必须设置 IP = 0x10，相当于把串口中断设置为最高优先级，
//这个时候，串口中断可以打断任何其他的中断服务函数实现嵌套，
IP =0x10; //把串口中断设置为最高优先级，必须的。

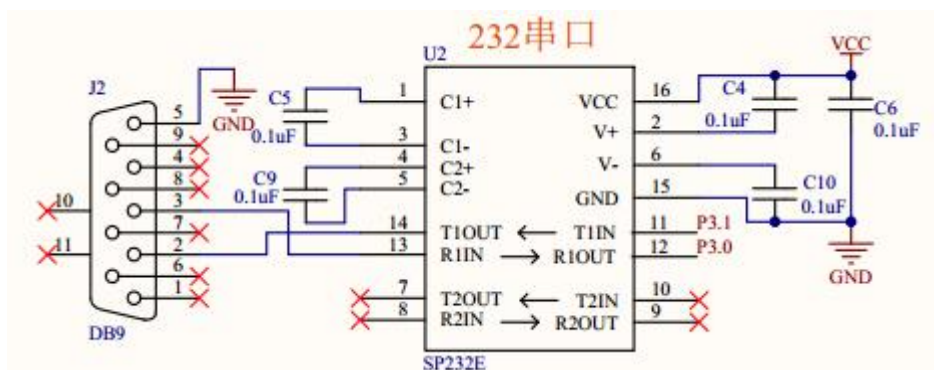
ES=1;          //允许串口中断
EA=1;          //允许总中断
while(1) //主循环
{
}

}

void usart(void) interrupt 4 //串口接发的中断函数，中断号为 4
{
    if(1==RI) //接收完一个字节后引起的中断
    {
        RI = 0; //及时清零，避免一直无缘无故的进入中断。
        Gu8ReceiveData=SBUF; //直接读取“串口专用缓存寄存器”SBUF 的 8 位数据。
        if(0x01==Gu8ReceiveData)
        {
            P0_0=0; //LED 亮。1 代表 LED 灭， 0 代表亮
        }
        else
        {
            P0_0=1; //LED 灭。1 代表 LED 灭， 0 代表亮
        }
    }
    else //发送数据引起的中断
    {
        TI = 0; //及时清除发送中断的标志，避免一直无缘无故的进入中断。
        //以下可以添加一个全局变量的标志位的相关代码，通知主函数已经发送完一个字节的数据了。
    }
}
}

```

## 【127.6 单片机串口电路的简易分析。】



上图 127.6.1 232 串口电路

单片机串口对外的引脚是与 I/O 口的“P3.1、P3.0”共用的。P3.1 是串口的 TX 引脚，即对外发送数据的引脚。P3.0 是串口的 RX 引脚，即接收外部数据的引脚。一旦项目中用了串口，那么这两个引脚就必须“专脚专用”，只给串口单独使用，不再做 I/O 口。平时通信的时候，跟其它单片机或者系统进行串口通信，除了接 TX 和 RX 这两根数据线之外，必须一定把双方的负极 GND 也“共地”接上，否则双方建立不了同样的电压参考点，通信毕然失败。因此，串口通信最低标配是 3 根线：RX, TX, GND。

如果两个甲乙单片机都布在一块板子上，距离不超过半米，他们两个要进行串口通信，怎么接线？把他们的 GND 连起来，然后 RX 与 TX “交叉”对接，甲的 RX 接到乙的 TX，甲的 TX 接到乙的 RX。这种在短距离通信的时候，不用增加任何外部辅助压差信号放大芯片，这种方式叫做“串口的 TTL”接线方式。

如果两个系统串口通信的距离比较远，比如在不同的板子上，1 米以上 10 米以下的距离，这时就不能采用原始的“串口的 TTL”接线方式，因为线缆越长电阻越大，本身就要消耗一些压降，而 3.3V 的压降很容易被消耗完，通信的可靠度和抗扰能力就会降低。为了解决这个问题，可以引用 232 标准的接线方式，外部需接一个压差放大的芯片，把从原来 3.3V 的压差放大到一两倍左右，通信的距离就大大提高。具体 232 的细节，大家可以网上搜搜“RS232”。注意，采用 232 协议通信，也要注意“共地”和数据线“交叉”的两个问题，232 通信的最低标配也是 3 根线：R, T, GND。上图 SP232E 就是一个压差信号放大的通信专用芯片。

## 第一百二十八节： 接收“固定协议”的串口程序框架。

### 【128.1 固定协议。】

实际项目中，串口一个回合的收发数据量远远不止 1 个字节，而是往往携带了某种“固定协议”的一串数据（专业术语称“一帧数据”）。一串数据的“固定协议”因为起到类似“校验”和“密码确认”的功能，因此在安全可靠性方面大大增强。但是上一节也提到，单片机利用最底层硬件的串口接口，一次收发的最小单位是“1 个字节”，那么，怎么样在此基础上搭建一个能快速收发并且能快速解析数据的程序框架就显得尤为重要。本节我跟大家分享我常用的串口程序框架，此框架主要包含“数据头，数据类型，数据长度，其它数据”这四部分。比如，为了通过串口去控制单片机的蜂鸣器发出不同长度的声音，我专门制定了一串十六进制的数据：EB 01 00 00 00 08 03 E8，下面以此串数据来跟大家详细分析。

数据头（EB）：占 1 个字节，作为“起始字节”，起到“接头暗号”的作用，平时用来过滤无关的数据。只有“接头暗号”吻合，单片机才会进入到接收其它有效数据的步骤，否则一直被“数据头”挡在门外视为无效数据。注意，数据头不能用十六进制的 00 或者 FF，因为 00 和 FF 的密码等级太弱，很多单片机一上电的瞬间因为硬件的某种不确定的原因，会直接误发送 00 或者 FF 这类干扰数据。

数据类型（01）：占用 1 个字节。数据类型是用来定义这串数据的用途，比如，01 代表用来控制蜂鸣器的，02 代表控制 LED 的，03 代表机器启动，等等功能，都可以用这个字节的数据进行分类定义。本例子用 01 代表控制蜂鸣器发出不同时间长度的声音。

数据长度（00 00 00 08）：占 4 个字节。用来告诉通信的对方，这串数据一共有多少个字节。本例子中，数据长度占用了 4 个字节，就意味着最大数据长度是一个 unsigned long 类型的数据范围，从 0 到 4294967295。比如，本例子中一串数据的长度是 8 个字节（EB 01 00 00 00 08 03 E8），因此这“数据长度”四个字节分别是 00 00 00 08，十六进制的 08 代表十进制的 8 字节。注意，51 单片机的内存是属于大端模式，因此十进制的 8 在四字节 unsigned long 的内存排列顺序是 00 00 00 08，也就是低位放在数组的高下标。如果是 stm32 的单片机，stm32 单片机的内存是属于小端模式，十进制的 8 在四字节 unsigned long 的内存排列顺序是 08 00 00 00，低位放在数组的低下标。为什么强调这个？因为主要方便我们用指针的方法实现数据的拆分和整合，这个知识点的内容我在前面第 62 节详细讲解过。

其它数据（03 E8）：此数据根据不同的“数据类型”可以用来做不同的用途，根据具体的项目而定。本例子十六进制的 03 E8，代表一个 unsigned int 的十进制数据 1000。此数据的大小用来控制蜂鸣器发声的长度，1000 代表长叫 1000ms。如果想让蜂鸣器短叫 100ms，只需把这两个字节改为：00 64。

### 【128.2 程序框架的四个要点分析。】

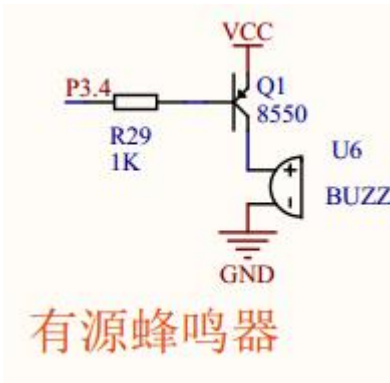
第一点：先接收后处理，开辟一块专用的内存数组。要处理一串数据，必须先征用一块内存数组专门用来缓存接收到的数据，等接收完此串数据再处理。

第二点：接头暗号。本节例子的数据头 EB 就是接头暗号。一旦接头暗号吻合，才会进入到下一步接收其它有效数据的步骤上。

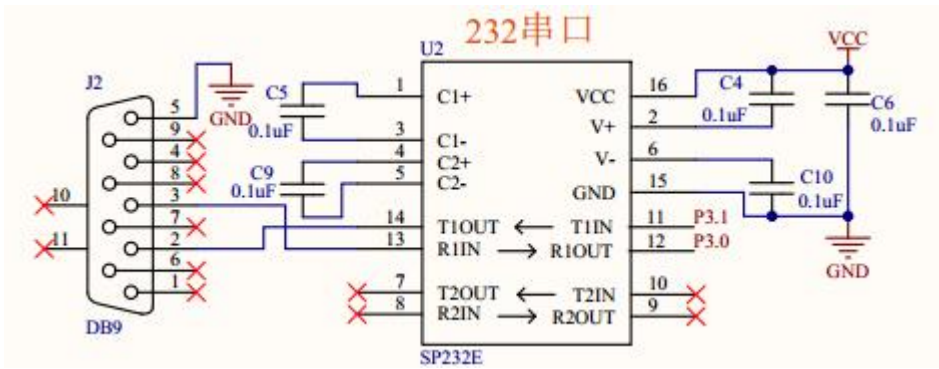
第三点：如何识别接收一串数据的完毕。本节例子中，是靠“固定协议”提供的“数据长度”来判别是否已经接收完一串数据。中断函数接收完一串数据后，应该用一个全局变量来给外部 main 函数一个通知，让 main 函数里面的相关函数来处理此串数据。

第四点：接收数据中相邻字节之间通信超时的异常处理。如果接头暗号吻合之后，马上切换到“接受其它有效数据”的步骤，但是，如果在此步骤的通信过程中一旦发现通信不连贯，就应该及时退出当下“接受其它有效数据”的步骤，继续返回到刚开始的“接头暗号”的步骤，为下一次接收新的一串数据做准备。那么，如何识别通信不连贯？靠判断接收数据中相邻字节之间的时间是否超时来决定，详细内容请看下面的程序例程。

【128.3 程序例程。】



上图 128.3.1 有源蜂鸣器电路



上图 128.3.2 232 串口电路

程序功能如下：

- (1) 在上位机的串口助手里，发送一串数据，控制蜂鸣器发出不同长度的声音。
- (2) 波特率 9600，校验位 NONE（无），数据位 8，停止位 1。
- (3) 十六进制的数据格式如下：

EB 01 00 00 00 08 XX XX

其中 EB 是数据头，01 是代表数据类型，00 00 00 08 代表数据长度是 8 个（十进制）。XX XX 代表一个 unsigned int 的数据，此数据的大小决定了蜂鸣器发出声音的长度。比如：

让蜂鸣器鸣叫 1000ms 的时间，发送十六进制的： EB 01 00 00 00 08 03 E8

让蜂鸣器鸣叫 100ms 的时间，发送十六进制的： EB 01 00 00 00 08 00 64

```
#include "REG52.H"

#define RECE_TIME_OUT    2000 //通信过程中字节之间的超时时间 2000ms
#define REC_BUFFER_SIZE  20   //接收数据的缓存数组的长度
```

```

void usart(void); //串口接收的中断函数
void TO_time(); //定时器的中断函数

void UsartTask(void); //串口接收的任务函数，放在主函数内

void SystemInitial(void) ;
void Delay(unsigned long u32DelayTime) ;
void PeripheralInitial(void) ;

void BeepOpen(void);
void BeepClose(void);
void VoiceScan(void);

sbit P3_4=P3^4;

volatile unsigned char vGu8BeepTimerFlag=0;
volatile unsigned int vGu16BeepTimerCnt=0;

unsigned char Gu8ReceBuffer[REC_BUFFER_SIZE]; //开辟一片接收数据的缓存
unsigned long Gu32ReceCnt=0; //接收缓存数组的下标
unsigned char Gu8ReceStep=0; //接收中断函数里的步骤变量
unsigned char Gu8ReceFeedDog=1; //“喂狗”的操作变量。
unsigned char Gu8ReceType=0; //接收的数据类型
unsigned long Gu32ReceDataLength=0; //接收的数据长度
unsigned char Gu8FinishFlag=0; //是否已接收完成一串数据的标志
unsigned long *pu32Data; //用于数据转换的指针
volatile unsigned char vGu8ReceTimeOutFlag=0; //通信过程中字节之间的超时定时器的开关
volatile unsigned int vGu16ReceTimeOutCnt=0; //通信过程中字节之间的超时定时器，“喂狗”的对象

void main()
{
    SystemInitial();
    Delay(10000);
    PeripheralInitial();
    while(1)
    {
        UsartTask(); //串口接收的任务函数
    }
}

void usart(void) interrupt 4 //串口接发的中断函数，中断号为 4
{

```

```

if(1==RI) //接收完一个字节后引起的中断
{
    RI = 0; //及时清零，避免一直无缘无故的进入中断。

/* 注释一：
* 以下 Gu8FinishFlag 变量的用途。
* 此变量一箭双雕，0 代表正处于接收数据的状态，1 代表已经接收完毕并且及时通知主函数中的处理函数
* UsartTask() 去处理新接收到的一串数据。除此之外，还起到一种“自锁自保护”的功能，在新数据还
* 没有被主函数处理完毕的时候，禁止接收其它新的数据，避免新数据覆盖了尚未处理的数据。
*/

    if(0==Gu8FinishFlag) //1 代表已经完成接收了一串新数据，并且禁止接收其它新的数据
    {

/* 注释二：
* 以下 Gu8ReceFeedDog 变量的用途。
* 此变量是用来检测并且识别通信过程中相邻的字节之间是否存在超时的情况。
* 如果大家听说过单片机中的“看门狗”这个概念，那么每接收到一个数据此变量就“置 1”一次，它的
* 作用就是起到及时“喂狗”的作用。每接收到一个数据此变量就“置 1”一次，在主函数里，相关
* 的定时器就会被重新赋值，只要这个定时器能不断及时的被补充新的“能量”新的值，那么这个定时器
* 就永远不会变成 0，只要不变成 0 就不会超时。如果两个字节之间通信时间超过了固定的长度，就意味
* 着此定时器变成了 0，这时就需要把中断函数里的接收步骤 Gu8Step 及时切换到“接头暗号”的步骤。
*/

        Gu8ReceFeedDog=1; //每接收到一个字节的数据，此标志就置 1 及时更新定时器的值。
        switch(Gu8ReceStep)
        {
            case 0: //接头暗号的步骤。判断数据头的步骤。
                Gu8ReceBuffer[0]=SBUF; //直接读取刚接收完的一个字节的数据。
                if(0xeb==Gu8ReceBuffer[0]) //等于数据头 0xeb，接头暗号吻合。
                {
                    Gu32ReceCnt=1; //接收缓存的下标
                    Gu8ReceStep=1; //切换到下一个步骤，接收其它有效的数据
                }
                break;

            case 1: //数据类型和长度
                Gu8ReceBuffer[Gu32ReceCnt]=SBUF; //直接读取刚接收完的一个字节的数据。
                Gu32ReceCnt++; //每接收一个字节，数组下标都自加 1，为接收下一个数据做准备
                if(Gu32ReceCnt>=6) //前 6 个数据。接收完了“数据类型”和“数据长度”。
                {
                    Gu8ReceType=Gu8ReceBuffer[1]; //提取“数据类型”
                    //以下的转换，在第 62 节讲解过的指针法
                    pu32Data=(unsigned long *)&Gu8ReceBuffer[2]; //数据转换
                    Gu32ReceDataLength=*pu32Data; //提取“数据长度”
                    if(Gu32ReceCnt>=Gu32ReceDataLength) //靠“数据长度”来判断是否完成

```



```

        {
            Gu8FinishFlag=1; //接收完成标志“置1”，通知主函数处理。
            Gu8ReceStep=0;    //及时切换回接头暗号的步骤
        }
        else    //如果还没结束，继续切换到下一个步骤，接收“其它数据”
        {
            Gu8ReceStep=2;    //切换到下一个步骤
        }
    }
    break;
case 2:    //其它数据
    Gu8ReceBuffer[Gu32ReceCnt]=SBUF; //直接读取刚接收完的一个字节的数据。
    Gu32ReceCnt++; //每接收一个字节，数组下标都自加1，为接收下一个数据做准备

    //靠“数据长度”来判断是否完成。也不允许超过数组的最大缓存的长度
    if (Gu32ReceCnt>=Gu32ReceDataLength || Gu32ReceCnt>=REC_BUFFER_SIZE)
    {
        Gu8FinishFlag=1; //接收完成标志“置1”，通知主函数处理。
        Gu8ReceStep=0;    //及时切换回接头暗号的步骤
    }
    break;
}
}
}
else //发送数据引起的中断
{
    TI = 0; //及时清除发送中断的标志，避免一直无缘无故的进入中断。
    //以下可以添加一个全局变量的标志位的相关代码，通知主函数已经发送完一个字节的数据了。
}
}

void UsartTask(void)    //串口接收的任务函数，放在主函数内
{
    static unsigned int *pSul6Data; //数据转换的指针
    static unsigned int Sul6Data;    //转换后的数据

    if (1==Gu8ReceFeedDog) //每被“喂一次狗”，就及时更新一次“超时检测的定时器”的初值
    {
        Gu8ReceFeedDog=0;

        vGu8ReceTimeOutFlag=0;
        vGu16ReceTimeOutCnt=RECE_TIME_OUT; //更新一次“超时检测的定时器”的初值
        vGu8ReceTimeOutFlag=1;
    }
}

```

```

}
else if (Gu8ReceStep>0&&0==vGu16ReceTimeOutCnt) //超时，并且步骤不在接头暗号的步骤
{
    Gu8ReceStep=0; //串口接收数据的中断函数及时切换回接头暗号的步骤
}

if (1==Gu8FinishFlag) //1 代表已经接收完毕一串新的数据，需要马上去处理
{
    switch (Gu8ReceType) //接收到的数据类型
    {
        case 0x01: //驱动蜂鸣器
            //以下的数据转换，在第 62 节讲解过的指针法
            pSul6Data=(unsigned int *)&Gu8ReceBuffer[6]; //数据转换。
            Sul6Data=*pSul6Data; //提取“蜂鸣器声音的长度”

            vGu8BeepTimerFlag=0;
            vGu16BeepTimerCnt=Sul6Data; //让蜂鸣器鸣叫
            vGu8BeepTimerFlag=1;
            break;
    }

    Gu8FinishFlag=0; //上面处理完数据再清零标志，为下一次接收新的数据做准备
}
}

void TO_time() interrupt 1
{
    VoiceScan();

    if (1==vGu8ReceTimeOutFlag&&vGu16ReceTimeOutCnt>0) //通信过程中字节之间的超时定时器
    {
        vGu16ReceTimeOutCnt--;
    }

    TH0=0xfc;
    TL0=0x66;
}

void SystemInitial(void)
{

```

```

unsigned char u8_TMOD_Temp=0;

//以下是定时器 0 的中断的配置
TMOD=0x01;
TH0=0xfc;
TL0=0x66;
EA=1;
ET0=1;
TR0=1;

//以下是串口接收中断的配置
//串口的波特率与内置的定时器 1 直接相关，因此配置此定时器 1 就等效于配置波特率。
u8_TMOD_Temp=0x20; //即将把定时器 1 设置为：工作方式 2，初值自动重装的 8 位定时器。
TMOD=TMOD&0x0f; //此寄存器低 4 位是跟定时器 0 相关，高 4 位是跟定时器 1 相关。先清零定时器 1。
TMOD=TMOD|u8_TMOD_Temp; //把高 4 位的定时器 1 填入 0x2，低 4 位的定时器 0 保持不变。
TH1=256-(11059200L/12/32/9600); //波特率为 9600。11059200 代表晶振 11.0592MHz，
TL1=256-(11059200L/12/32/9600); //L 代表 long 的长类型数据。根据芯片手册提供的计算公式。
TR1=1; //开启定时器 1

SM0=0;
SM1=1; //SM0 与 SM1 的设置：选择 10 位异步通信，波特率根据定时器 1 可变
REN=1; //允许串口接收数据

//为了保证串口中断接收的数据不丢失，必须设置 IP = 0x10，相当于把串口中断设置为最高优先级，
//这个时候，串口中断可以打断任何其他的中断服务函数实现嵌套，
IP =0x10; //把串口中断设置为最高优先级，必须的。

ES=1; //允许串口中断
EA=1; //允许总中断
}

void Delay(unsigned long u32DelayTime)
{
    for(;u32DelayTime>0;u32DelayTime--);
}

void PeripheralInitial(void)
{
}

void BeepOpen(void)
{
    P3_4=0;

```

```

}

void BeepClose(void)
{
    P3_4=1;
}

void VoiceScan(void)
{

    static unsigned char Su8Lock=0;

    if(1==vGu8BeepTimerFlag&&vGu16BeepTimerCnt>0)
    {
        if(0==Su8Lock)
        {
            Su8Lock=1;
            BeepOpen();
        }
        else
        {

            vGu16BeepTimerCnt--;

            if(0==vGu16BeepTimerCnt)
            {
                Su8Lock=0;
                BeepClose();
            }

        }
    }
}

```

## 第一百二十九节：接收带“动态密匙”与“累加和”校验数据的串口程序框架。

### 【129.1 “累加和”与“动态密匙”。】

上一节讲了串口基本的程序框架，但是没有讲到校验。校验在很多通信项目中是必不可少的。比如，在事关金融或者生命安全的项目，是不允许有任何的数据丢失或错误的；在容易受干扰的工业环境，或者在无线通信的项目中，这些项目往往容易丢失数据；还有一种常见的人为过失是，在编写程序的层面，因为超时重发的时间与从机不匹配，导致反馈的信息延时而造成数据丢失，如果这种情况也加上校验，通信会稳定可靠很多。

上一节讲到“数据头，数据类型，数据长度，其它数据”这四个元素，本节在此基础上，增加两个校验的元素，分别是“动态密匙”与“累加和”。“动态密匙”占用2个字节，“累加和”占用1个字节，因此，这两个元素一共占用最后面的3个字节。分析如下：

数据头（EB）：占1个字节，作为“起始字节”，起到“接头暗号”的作用，平时用来过滤无关的数据。

数据类型（01）：占用1个字节。数据类型是用来定义这串数据的用途。

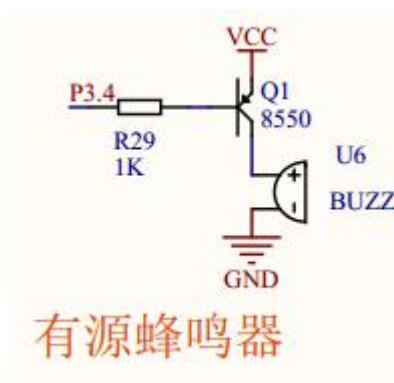
数据长度（00 00 00 0B）：占4个字节。用来告诉通信的对方，这串数据一共有多少个字节。

其它数据（03 E8）：此数据根据不同的“数据类型”可以用来做不同的用途，根据具体的项目而定。

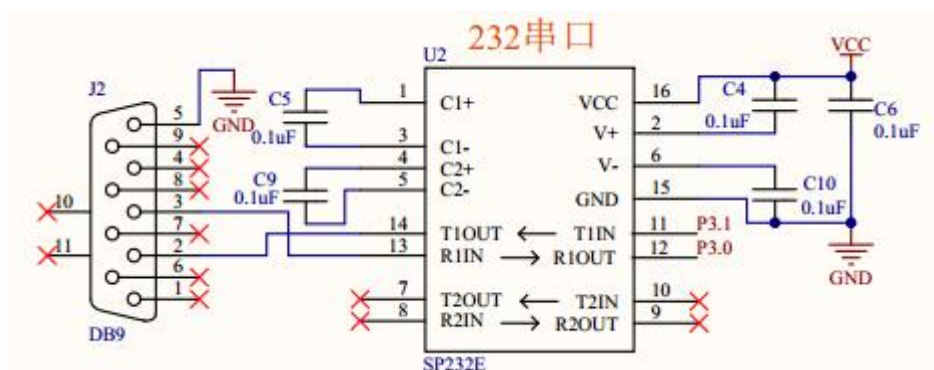
动态密匙（00 01）：这两个字节代表一个 unsigned int 类型的数据，数据范围是从0到65535，但是考虑到数据更加安全可靠，一般丢弃了首尾的0（十六进制的00 00）与65535（十六进制的FF FF），只保留从1到65534的变化。大部分的通信模型都是主机对从机的“一问一应答”模式，也就是，主机每发送一条指令给从机，从机才返回一条消息作为应答。如果主机发送了信息后，在规定的时间内，没有收到从机的应答指令，主机就继续发送信息给从机，但是此时，从机本来应该应答主机当前指令的，可能因为某种情况导致反馈的信息发生了延时，导致此时应答的数据是主机的上一条指令，从而造成“一问一应答”的数据帧发送了错位，这种情况加上“动态密匙”就能使问题得到有效的解决。主机每发送一条信息，信息里都携带了2个字节的“动态密匙”，从机每收到主机的一条信息，在应答此信息时都把收到的“动态密匙”原封不动的反馈给主机，主机再查看发送的“动态密匙”与接收到的“动态密匙”是否一致，以此来判断应答数据是否有效。“动态密匙”像流水号一样，每发送一次指令后都累加1，不断发生变化，从1到65534，依次循环。这是数据校验的一种方式。

累加和（E3）。“累加和”放在数据串的最后一个字节，是前面所有字节的累加之和（不包括自己本身的字节），累加的结果高于一个字节的那部分自动溢出丢掉，只保留低8位的一个字节的数据。比如：本例子中，数据串是：EB 01 00 00 00 0B 03 E8 00 01 E3。其中最后一个字节E3就是“累加和”，前面所有字节相加等于十六进制的0x1E3，只保留低8位的一个字节的数据，因此为十六进制的0xE3。验证“累加和”的方法，可以借用电脑“附件”自带的“计算器”软件来实现，打开“计算器”软件后，在“查看”的下拉菜单里，选择“程序员”，然后选择“十六进制”。不管是主机还是从机，每接收到一串数据后，都要自己计算一次“累加和”，把自己计算得到的“累加和”与接收到的最后一个字节的“累加和”进行对比，来判断接收到的数据是否发生了丢失或者错误。

### 【129.2 程序例程。】



上图 129.2.1 有源蜂鸣器电路



上图 129.2.2 232 串口电路

程序功能如下：

(1) 单片机模拟从机，上位机的串口助手模拟主机。在上位机的串口助手里，发送一串数据，控制蜂鸣器发出不同长度的声音。

(2) 本节因为还没有讲到数据发送的内容，因此应答“动态密匙”那部分的代码暂时不写，只写验证“累加和”那部分的代码。

(3) 波特率 9600，校验位 NONE（无），数据位 8，停止位 1。

(4) 十六进制的数据格式：EB 01 00 00 00 0B XX XX YY YY ZZ 。其中：

EB 是数据头。

01 是代表数据类型。

00 00 00 0B 代表数据长度是 11 个（十进制）。

XX XX 代表一个 unsigned int 的数据，此数据的大小决定了蜂鸣器发出声音的长度。

YY YY 代表一个 unsigned int 的动态密匙，每收发一条指令，此数据累加一次 1，范围从 1 到 65534。

ZZ 代表前面所有字节的累加和。

比如：

让蜂鸣器鸣叫 1000 毫秒，密匙为 00 01，发送十六进制的：EB 01 00 00 00 0B 03 E8 00 01 E3

让蜂鸣器鸣叫 100 毫秒，密匙为 00 02，发送十六进制的：EB 01 00 00 00 0B 00 64 00 02 5D

```

#include "REG52.H"

#define RECE_TIME_OUT    2000 //通信过程中字节之间的超时时间 2000ms
#define REC_BUFFER_SIZE  20   //接收数据的缓存数组的长度

void usart(void); //串口接收的中断函数
void TO_time();   //定时器的中断函数

void UsartTask(void); //串口接收的任务函数，放在主函数内

void SystemInitial(void) ;
void Delay(unsigned long u32DelayTime) ;
void PeripheralInitial(void) ;

void BeepOpen(void);
void BeepClose(void);
void VoiceScan(void);

sbit P3_4=P3^4;

volatile unsigned char vGu8BeepTimerFlag=0;
volatile unsigned int vGu16BeepTimerCnt=0;

unsigned char Gu8ReceBuffer[REC_BUFFER_SIZE]; //开辟一片接收数据的缓存
unsigned long Gu32ReceCnt=0; //接收缓存数组的下标
unsigned char Gu8ReceStep=0; //接收中断函数里的步骤变量
unsigned char Gu8ReceFeedDog=1; //“喂狗”的操作变量。
unsigned char Gu8ReceType=0; //接收的数据类型
unsigned int Gu16ReceYY=0; //接收的动态密钥
unsigned char Gu8ReceZZ=0; //接收的累加和，必须是 unsigned char 的数据类型
unsigned long Gu32ReceDataLength=0; //接收的数据长度
unsigned char Gu8FinishFlag=0; //是否已接收完成一串数据的标志
unsigned long *pu32Data; //用于数据转换的指针
volatile unsigned char vGu8ReceTimeOutFlag=0; //通信过程中字节之间的超时定时器的开关
volatile unsigned int vGu16ReceTimeOutCnt=0; //通信过程中字节之间的超时定时器，“喂狗”的对象

void main()
{
    SystemInitial();
    Delay(10000);
    PeripheralInitial();
    while(1)
    {

```

```

    UsartTask();    //串口接收的任务函数
}
}

void usart(void) interrupt 4    //串口接发的中断函数，中断号为 4
{
    if(1==RI)    //接收完一个字节后引起的中断
    {
        RI = 0; //及时清零，避免一直无缘无故的进入中断。

/* 注释一：
* 以下 Gu8FinishFlag 变量的用途。
* 此变量一箭双雕，0 代表正处于接收数据的状态，1 代表已经接收完毕并且及时通知主函数中的处理函数
* UsartTask() 去处理新接收到的一串数据。除此之外，还起到一种“自锁自保护”的功能，在新数据还
* 没有被主函数处理完毕的时候，禁止接收其它新的数据，避免新数据覆盖了尚未处理的数据。
*/
        if(0==Gu8FinishFlag)    //1 代表已经完成接收了一串新数据，并且禁止接收其它新的数据
        {

/* 注释二：
* 以下 Gu8ReceFeedDog 变量的用途。
* 此变量是用来检测并且识别通信过程中相邻的字节之间是否存在超时的情况。
* 如果大家听说过单片机中的“看门狗”这个概念，那么每接收到一个数据此变量就“置 1”一次，它的
* 作用就是起到及时“喂狗”的作用。每接收到一个数据此变量就“置 1”一次，在主函数里，相关
* 的定时器就会被重新赋值，只要这个定时器能不断及时的被补充新的“能量”新的值，那么这个定时器
* 就永远不会变成 0，只要不变成 0 就不会超时。如果两个字节之间通信时间超过了固定的长度，就意味
* 着此定时器变成了 0，这时就需要把中断函数里的接收步骤 Gu8Step 及时切换到“接头暗号”的步骤。
*/
        Gu8ReceFeedDog=1; //每接收到一个字节的数据，此标志就置 1 及时更新定时器的值。
        switch(Gu8ReceStep)
        {
            case 0:    //接头暗号的步骤。判断数据头的步骤。
                Gu8ReceBuffer[0]=SBUF; //直接读取刚接收完的一个字节的数据。
                if(0xeb==Gu8ReceBuffer[0])    //等于数据头 0xeb，接头暗号吻合。
                {
                    Gu32ReceCnt=1; //接收缓存的下标
                    Gu8ReceStep=1;    //切换到下一个步骤，接收其它有效的数据
                }
                break;

            case 1:    //数据类型和长度
                Gu8ReceBuffer[Gu32ReceCnt]=SBUF; //直接读取刚接收完的一个字节的数据。
                Gu32ReceCnt++; //每接收一个字节，数组下标都自加 1，为接收下一个数据做准备
                if(Gu32ReceCnt>=6)    //前 6 个数据。接收完了“数据类型”和“数据长度”。

```



```

        {
            Gu8ReceType=Gu8ReceBuffer[1]; //提取“数据类型”
            //以下的转换，在第 62 节讲解过的指针法
            pu32Data=(unsigned long *)&Gu8ReceBuffer[2]; //数据转换
            Gu32ReceDataLength=*pu32Data; //提取“数据长度”
            if (Gu32ReceCnt>=Gu32ReceDataLength) //靠“数据长度”来判断是否完成
            {
                Gu8FinishFlag=1; //接收完成标志“置 1”，通知主函数处理。
                Gu8ReceStep=0; //及时切换回接头暗号的步骤
            }
            else //如果还没结束，继续切换到下一个步骤，接收“其它数据”
            {
                Gu8ReceStep=2; //切换到下一个步骤
            }
        }
        break;
    case 2: //其它数据
        Gu8ReceBuffer[Gu32ReceCnt]=SBUF; //直接读取刚接收完的一个字节的数据。
        Gu32ReceCnt++; //每接收一个字节，数组下标都自加 1，为接收下一个数据做准备

        //靠“数据长度”来判断是否完成。也不允许超过数组的最大缓存的长度
        if (Gu32ReceCnt>=Gu32ReceDataLength || Gu32ReceCnt>=REC_BUFFER_SIZE)
        {
            Gu8FinishFlag=1; //接收完成标志“置 1”，通知主函数处理。
            Gu8ReceStep=0; //及时切换回接头暗号的步骤
        }
        break;
    }
}

else //发送数据引起的中断
{
    TI = 0; //及时清除发送中断的标志，避免一直无缘无故的进入中断。
    //以下可以添加一个全局变量的标志位的相关代码，通知主函数已经发送完一个字节的数据了。
}
}

void UsartTask(void) //串口接收的任务函数，放在主函数内
{
    static unsigned int *pSul6Data; //数据转换的指针
    static unsigned int Sul6Data; //转换后的数据
    static unsigned int i;
    static unsigned char Su8RecZZ=0; //计算的“累加和”，必须是 unsigned char 的数据类型

```

```

if (1==Gu8ReceFeedDog) //每被“喂一次狗”，就及时更新一次“超时检测的定时器”的初值
{
    Gu8ReceFeedDog=0;

    vGu8ReceTimeOutFlag=0;
    vGu16ReceTimeOutCnt=RECE_TIME_OUT;//更新一次“超时检测的定时器”的初值
    vGu8ReceTimeOutFlag=1;
}
else if (Gu8ReceStep>0&&0==vGu16ReceTimeOutCnt) //超时，并且步骤不在接头暗号的步骤
{
    Gu8ReceStep=0; //串口接收数据的中断函数及时切换回接头暗号的步骤
}

if (1==Gu8FinishFlag) //1 代表已经接收完毕一串新的数据，需要马上去处理
{
    switch (Gu8ReceType) //接收到的数据类型
    {
        case 0x01: //驱动蜂鸣器
            //以下的数据转换，在第 62 节讲解过的指针法

            pSu16Data=(unsigned int *)&Gu8ReceBuffer[Gu32ReceDataLength-3]; //数据转换
            Gu16ReceYY=*pSu16Data; //提取“动态密钥”。本例子中暂时不做返回应答的处理

            Gu8ReceZZ=Gu8ReceBuffer[Gu32ReceDataLength-1]; //提取“累加和”

            Su8RecZZ=0;
            for (i=0;i<(Gu32ReceDataLength-1);i++)
            {
                Su8RecZZ=Su8RecZZ+Gu8ReceBuffer[i]; //计算“累加和”
            }

            if (Su8RecZZ==Gu8ReceZZ) //验证“累加和”，“计算的”与“接收的”是否一致
            {
                pSu16Data=(unsigned int *)&Gu8ReceBuffer[6]; //数据转换。
                Su16Data=*pSu16Data; //提取“蜂鸣器声音的长度”

                vGu8BeepTimerFlag=0;
                vGu16BeepTimerCnt=Su16Data; //让蜂鸣器鸣叫
                vGu8BeepTimerFlag=1;
            }
    }
}

```

```

        break;
    }

    Gu8FinishFlag=0; //上面处理完数据再清零标志，为下一次接收新的数据做准备
}

void T0_time() interrupt 1
{
    VoiceScan();

    if(1==vGu8ReceTimeOutFlag&&vGu16ReceTimeOutCnt>0) //通信过程中字节之间的超时定时器
    {
        vGu16ReceTimeOutCnt--;
    }

    TH0=0xfc;
    TL0=0x66;
}

void SystemInitial(void)
{
    unsigned char u8_TMOD_Temp=0;

    //以下是定时器 0 的中断的配置
    TMOD=0x01;
    TH0=0xfc;
    TL0=0x66;
    EA=1;
    ET0=1;
    TR0=1;

    //以下是串口接收中断的配置
    //串口的波特率与内置的定时器 1 直接相关，因此配置此定时器 1 就等效于配置波特率。
    u8_TMOD_Temp=0x20; //即将把定时器 1 设置为：工作方式 2，初值自动重装的 8 位定时器。
    TMOD=TMOD&0x0f; //此寄存器低 4 位是跟定时器 0 相关，高 4 位是跟定时器 1 相关。先清零定时器 1。
    TMOD=TMOD|u8_TMOD_Temp; //把高 4 位的定时器 1 填入 0x2，低 4 位的定时器 0 保持不变。
    TH1=256-(11059200L/12/32/9600); //波特率为 9600。11059200 代表晶振 11.0592MHz，
    TL1=256-(11059200L/12/32/9600); //L 代表 long 的长类型数据。根据芯片手册提供的计算公式。
    TR1=1; //开启定时器 1

```

```

SM0=0;
SM1=1;  //SM0 与 SM1 的设置：选择 10 位异步通信，波特率根据定时器 1 可变
REN=1;  //允许串口接收数据

//为了保证串口中断接收的数据不丢失，必须设置 IP = 0x10，相当于把串口中断设置为最高优先级，
//这个时候，串口中断可以打断任何其他的中断服务函数实现嵌套，
IP =0x10;  //把串口中断设置为最高优先级，必须的。

ES=1;      //允许串口中断
EA=1;      //允许总中断
}

void Delay(unsigned long u32DelayTime)
{
    for(;u32DelayTime>0;u32DelayTime--);
}

void PeripheralInitial(void)
{
}

void BeepOpen(void)
{
    P3_4=0;
}

void BeepClose(void)
{
    P3_4=1;
}

void VoiceScan(void)
{
    static unsigned char Su8Lock=0;

    if(1==vGu8BeepTimerFlag&&vGu16BeepTimerCnt>0)
    {
        if(0==Su8Lock)
        {
            Su8Lock=1;
            BeepOpen();
        }
    }
}

```

```
    else
    {

        vGu16BeepTimerCnt--;

        if(0==vGu16BeepTimerCnt)
        {
            Su8Lock=0;
            BeepClose();
        }

    }

}
```

## 第一百三十节：接收带“动态密钥”与“异或”校验数据的串口程序框架。

### 【130.1 “异或”的校验。】

通信的校验常用有两种，一种是“累加和”，另一种是“异或”。“异或”算法的详细介绍请看前面章节的第 32 节。

上一节讲的“累加和”，放在数据串的最后一个字节，是前面所有字节的累加之和（不包括自己本身的字节），累加的结果高于一个字节的那部分自动溢出丢掉，只保留低 8 位的一个字节的数据。本节讲的“异或”，也是放在数据串的最后一个字节，是前面所有字节的异或结果（不包括自己本身的字节）。本节在上一节的基础上，只更改以下这段校验算法的代码即可。

上一节的“累加和”算法如下：

```
Gu8ReceZZ=Gu8ReceBuffer[Gu32ReceDataLength-1]; //提取“累加和”

Su8RecZZ=0;
for(i=0;i<(Gu32ReceDataLength-1);i++)
{
    Su8RecZZ=Su8RecZZ+Gu8ReceBuffer[i]; //计算“累加和”
}

if(Su8RecZZ==Gu8ReceZZ) //验证“累加和”，“计算的”与“接收的”是否一致
{
    //此处省去若干代码
}
```

本节的“异或”算法如下：

```
Gu8ReceZZ=Gu8ReceBuffer[Gu32ReceDataLength-1]; //提取接收到的“异或”

Su8RecZZ=Gu8ReceBuffer[0]; //提取数据串第“i=0”个数据作为异或的原始数据
for(i=1;i<(Gu32ReceDataLength-1);i++) //注意，这里是从第“i=1”个数据开始
{
    Su8RecZZ=Su8RecZZ^Gu8ReceBuffer[i]; //计算“异或”
}

if(Su8RecZZ==Gu8ReceZZ) //验证“异或”，“计算的”与“接收的”是否一致
{
    //此处省去若干代码
}
```

### 【130.2 通信协议。】

数据头（EB）：占 1 个字节，作为“起始字节”，起到“接头暗号”的作用，平时用来过滤无关的数据。

数据类型 (01): 占用 1 个字节。数据类型是用来定义这串数据的用途。

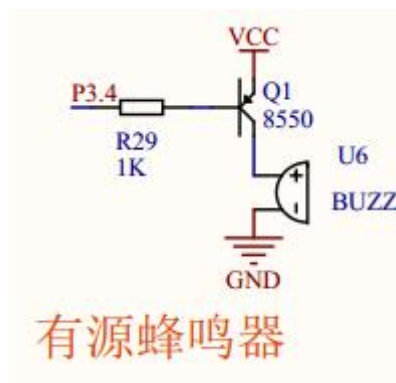
数据长度 (00 00 00 0B): 占 4 个字节。用来告诉通信的对方, 这串数据一共有多少个字节。

其它数据 (03 E8): 此数据根据不同的“数据类型”可以用来做不同的用途, 根据具体的项目而定。

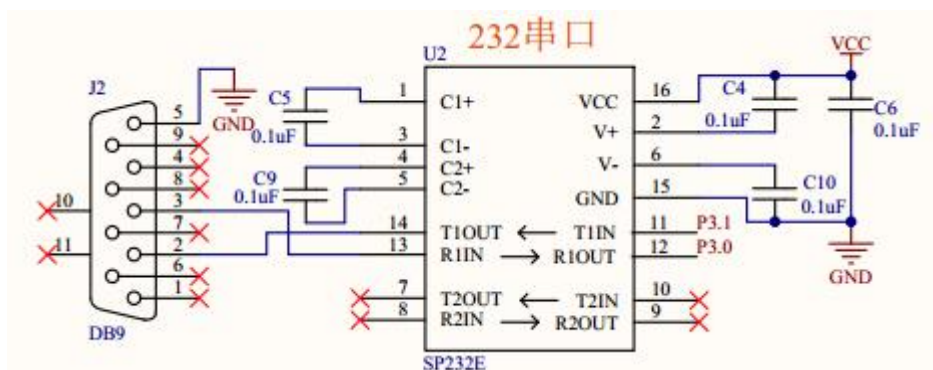
动态密匙 (00 01): 这两个字节代表一个 unsigned int 类型的数据, 数据范围是从 0 到 65535, 但是考虑到数据更加安全可靠, 一般丢弃了首尾的 0 (十六进制的 00 00) 与 65535 (十六进制的 FF FF), 只保留从 1 到 65534 的变化。大部分的通信模型都是主机对从机的“一问一应答”模式, 也就是, 主机每发送一条指令给从机, 从机才返回一条消息作为应答。如果主机发送了信息后, 在规定的时间内, 没有收到从机的应答指令, 主机就继续发送信息给从机, 但是此时, 从机本来应该应答主机当前指令的, 可能因为某种情况导致反馈的信息发生了延时, 导致此时应答的数据是主机的上一条指令, 从而造成“一问一应答”的数据帧发送了错位, 这种情况加上“动态密匙”就能使问题得到有效的解决。主机每发送一条信息, 信息里都携带了 2 个字节的“动态密匙”, 从机每收到主机的一条信息, 在应答此信息时都把收到的“动态密匙”原封不动的反馈给主机, 主机再查看发送的“动态密匙”与接收到的“动态密匙”是否一致, 以此来判断应答数据是否有效。“动态密匙”像流水号一样, 每发送一次指令后都累加 1, 不断发生变化, 从 1 到 65534, 依次循环。这是数据校验的一种方式。

异或 (0B)。“异或”放在数据串的最后一个字节, 是前面所有字节的异或结果 (不包括自己本身的字节)。比如: 本例子中, 数据串是: EB 01 00 00 00 0B 03 E8 00 01 0B。其中最后一个字节 0B 就是“异或”字节, 前面所有字节相“异或”等于十六进制的 0B。验证“异或”的方法, 可以借用电脑“附件”自带的“计算器”软件来实现, 打开“计算器”软件后, 在“查看”的下拉菜单里, 选择“程序员”, 然后选择“十六进制”, 该计算器软件的异或运算按键是“Xor”。不管是主机还是从机, 每接收到一串数据后, 都要自己计算一次“异或”, 把自己计算得到的“异或”与接收到的最后一个字节的“异或”进行对比, 来判断接收到的数据是否发生了丢失或者错误。

### 【130.3 程序例程。】



上图 130.3.1 有源蜂鸣器电路



上图 130.3.2 232 串口电路

程序功能如下：

(5) 单片机模拟从机，上位机的串口助手模拟主机。在上位机的串口助手里，发送一串数据，控制蜂鸣器发出不同长度的声音。

(6) 本节因为还没有讲到数据发送的内容，因此应答“动态密匙”那部分的代码暂时不写，只写验证“异或”那部分的代码。

(7) 波特率 9600，校验位 NONE（无），数据位 8，停止位 1。

(8) 十六进制的数据格式：EB 01 00 00 00 0B XX XX YY YY ZZ 。其中：

EB 是数据头。

01 是代表数据类型。

00 00 00 0B 代表数据长度是 11 个（十进制）。

XX XX 代表一个 unsigned int 的数据，此数据的大小决定了蜂鸣器发出声音的长度。

YY YY 代表一个 unsigned int 的动态密匙，每收发一条指令，此数据累加一次 1，范围从 1 到 65534。

ZZ 代表前面所有字节的异或结果。

比如：

让蜂鸣器鸣叫 1000 毫秒，密匙为 00 01，发送十六进制的：EB 01 00 00 00 0B 03 E8 00 01 0B

让蜂鸣器鸣叫 100 毫秒，密匙为 00 02，发送十六进制的：EB 01 00 00 00 0B 00 64 00 02 87

```
#include "REG52.H"

#define RECE_TIME_OUT    2000 //通信过程中字节之间的超时时间 2000ms
#define REC_BUFFER_SIZE  20   //接收数据的缓存数组的长度

void usart(void); //串口接收的中断函数
void TO_time();   //定时器的中断函数

void UsartTask(void); //串口接收的任务函数，放在主函数内

void SystemInitial(void) ;
void Delay(unsigned long u32DelayTime) ;
void PeripheralInitial(void) ;
```



```

void BeepOpen(void);
void BeepClose(void);
void VoiceScan(void);

sbit P3_4=P3^4;

volatile unsigned char vGu8BeepTimerFlag=0;
volatile unsigned int vGu16BeepTimerCnt=0;

unsigned char Gu8ReceBuffer[REC_BUFFER_SIZE]; //开辟一片接收数据的缓存
unsigned long Gu32ReceCnt=0; //接收缓存数组的下标
unsigned char Gu8ReceStep=0; //接收中断函数里的步骤变量
unsigned char Gu8ReceFeedDog=1; //“喂狗”的操作变量。
unsigned char Gu8ReceType=0; //接收的数据类型
unsigned int Gu16ReceYY=0; //接收的动态密钥
unsigned char Gu8ReceZZ=0; //接收的异或
unsigned long Gu32ReceDataLength=0; //接收的数据长度
unsigned char Gu8FinishFlag=0; //是否已接收完成一串数据的标志
unsigned long *pu32Data; //用于数据转换的指针
volatile unsigned char vGu8ReceTimeOutFlag=0; //通信过程中字节之间的超时定时器的开关
volatile unsigned int vGu16ReceTimeOutCnt=0; //通信过程中字节之间的超时定时器，“喂狗”的对象

void main()
{
    SystemInitial();
    Delay(10000);
    PeripheralInitial();
    while(1)
    {
        UsartTask(); //串口接收的任务函数
    }
}

void usart(void) interrupt 4 //串口接发的中断函数，中断号为4
{
    if(1==RI) //接收完一个字节后引起的中断
    {
        RI = 0; //及时清零，避免一直无缘无故的进入中断。
    }
}

/* 注释一：
* 以下 Gu8FinishFlag 变量的用途。
* 此变量一箭双雕，0 代表正处于接收数据的状态，1 代表已经接收完毕并且及时通知主函数中的处理函数
* UsartTask() 去处理新接收到的一串数据。除此之外，还起到一种“自锁自保护”的功能，在新数据还

```

```

* 没有被主函数处理完毕的时候，禁止接收其它新的数据，避免新数据覆盖了尚未处理的数据。
*/
    if(0==Gu8FinishFlag)  //1 代表已经完成接收了一串新数据，并且禁止接收其它新的数据
    {

/* 注释二：
* 以下 Gu8ReceFeedDog 变量的用途。
* 此变量是用来检测并且识别通信过程中相邻的字节之间是否存在超时的情况。
* 如果大家听说过单片机中的“看门狗”这个概念，那么每接收到一个数据此变量就“置 1”一次，它的
* 作用就是起到及时“喂狗”的作用。每接收到一个数据此变量就“置 1”一次，在主函数里，相关
* 的定时器就会被重新赋值，只要这个定时器能不断及时的被补充新的“能量”新的值，那么这个定时器
* 就永远不会变成 0，只要不变成 0 就不会超时。如果两个字节之间通信时间超过了固定的长度，就意味
* 着此定时器变成了 0，这时就需要把中断函数里的接收步骤 Gu8Step 及时切换到“接头暗号”的步骤。
*/

        Gu8ReceFeedDog=1; //每接收到一个字节的的数据，此标志就置 1 及时更新定时器的值。
        switch(Gu8ReceStep)
        {
            case 0:        //接头暗号的步骤。判断数据头的步骤。
                Gu8ReceBuffer[0]=SBUF; //直接读取刚接收完的一个字节的数据。
                if(0xeb==Gu8ReceBuffer[0])  //等于数据头 0xeb，接头暗号吻合。
                {
                    Gu32ReceCnt=1; //接收缓存的下标
                    Gu8ReceStep=1;  //切换到下一个步骤，接收其它有效的数据
                }
                break;

            case 1:        //数据类型和长度
                Gu8ReceBuffer[Gu32ReceCnt]=SBUF; //直接读取刚接收完的一个字节的数据。
                Gu32ReceCnt++; //每接收一个字节，数组下标都自加 1，为接收下一个数据做准备
                if(Gu32ReceCnt>=6)  //前 6 个数据。接收完了“数据类型”和“数据长度”。
                {
                    Gu8ReceType=Gu8ReceBuffer[1]; //提取“数据类型”
                    //以下的数据转换，在第 62 节讲解过的指针法
                    pu32Data=(unsigned long *)&Gu8ReceBuffer[2]; //数据转换
                    Gu32ReceDataLength=*pu32Data; //提取“数据长度”
                    if(Gu32ReceCnt>=Gu32ReceDataLength) //靠“数据长度”来判断是否完成
                    {
                        Gu8FinishFlag=1; //接收完成标志“置 1”，通知主函数处理。
                        Gu8ReceStep=0;  //及时切换回接头暗号的步骤
                    }
                    else  //如果还没结束，继续切换到下一个步骤，接收“其它数据”
                    {
                        Gu8ReceStep=2;  //切换到下一个步骤
                    }
                }
            }
        }
    }
}

```

```

        }
        break;
    case 2:    //其它数据
        Gu8ReceBuffer[Gu32ReceCnt]=SBUF; //直接读取刚接收完的一个字节的数据。
        Gu32ReceCnt++; //每接收一个字节，数组下标都自加 1，为接收下一个数据做准备

        //靠“数据长度”来判断是否完成。也不允许超过数组的最大缓存的长度
        if (Gu32ReceCnt>=Gu32ReceDataLength || Gu32ReceCnt>=REC_BUFFER_SIZE)
        {
            Gu8FinishFlag=1; //接收完成标志“置 1”，通知主函数处理。
            Gu8ReceStep=0;    //及时切换回接头暗号的步骤
        }
        break;
    }
}

else //发送数据引起的中断
{
    TI = 0; //及时清除发送中断的标志，避免一直无缘无故的进入中断。
    //以下可以添加一个全局变量的标志位的相关代码，通知主函数已经发送完一个字节的数据了。
}
}

void UsartTask(void)    //串口接收的任务函数，放在主函数内
{
    static unsigned int *pSul6Data; //数据转换的指针
    static unsigned int Sul6Data;    //转换后的数据
    static unsigned int i;
    static unsigned char Su8RecZZ=0; //计算的“异或”

    if (1==Gu8ReceFeedDog) //每被“喂一次狗”，就及时更新一次“超时检测的定时器”的初值
    {
        Gu8ReceFeedDog=0;

        vGu8ReceTimeOutFlag=0;
        vGu16ReceTimeOutCnt=RECE_TIME_OUT; //更新一次“超时检测的定时器”的初值
        vGu8ReceTimeOutFlag=1;
    }
    else if (Gu8ReceStep>0&&0==vGu16ReceTimeOutCnt) //超时，并且步骤不在接头暗号的步骤
    {
        Gu8ReceStep=0; //串口接收数据的中断函数及时切换回接头暗号的步骤
    }
}

```

```

}

if(1==Gu8FinishFlag) //1 代表已经接收完毕一串新的数据，需要马上去处理
{
    switch(Gu8ReceType) //接收到的数据类型
    {
        case 0x01: //驱动蜂鸣器
            //以下的数据转换，在第 62 节讲解过的指针法

            pSu16Data=(unsigned int *)&Gu8ReceBuffer[Gu32ReceDataLength-3]; //数据转换
            Gu16ReceYY=*pSu16Data; //提取“动态密钥”。本例子中暂时不做返回应答的处理

            Gu8ReceZZ=Gu8ReceBuffer[Gu32ReceDataLength-1]; //提取接收到的“异或”

            Su8RecZZ=Gu8ReceBuffer[0]; //提取数据串第“i=0”个数据作为异或的原始数据
            for(i=1;i<(Gu32ReceDataLength-1);i++) //注意，这里是从第“i=1”个数据开始
            {
                Su8RecZZ=Su8RecZZ^Gu8ReceBuffer[i]; //计算“异或”
            }

            if(Su8RecZZ==Gu8ReceZZ) //验证“异或”，“计算的”与“接收的”是否一致
            {
                pSu16Data=(unsigned int *)&Gu8ReceBuffer[6]; //数据转换。
                Su16Data=*pSu16Data; //提取“蜂鸣器声音的长度”

                vGu8BeepTimerFlag=0;
                vGu16BeepTimerCnt=Su16Data; //让蜂鸣器鸣叫
                vGu8BeepTimerFlag=1;
            }

            break;
        }

        Gu8FinishFlag=0; //上面处理完数据再清零标志，为下一次接收新的数据做准备
    }
}

void T0_time() interrupt 1
{
    VoiceScan();

    if(1==vGu8ReceTimeOutFlag&&vGu16ReceTimeOutCnt>0) //通信过程中字节之间的超时定时器

```

```

    {
        vGul6ReceTimeOutCnt--;
    }

    TH0=0xfc;
    TL0=0x66;
}

void SystemInitial(void)
{
    unsigned char u8_TMOD_Temp=0;

    //以下是定时器 0 的中断的配置
    TMOD=0x01;
    TH0=0xfc;
    TL0=0x66;
    EA=1;
    ET0=1;
    TR0=1;

    //以下是串口接收中断的配置
    //串口的波特率与内置的定时器 1 直接相关，因此配置此定时器 1 就等效于配置波特率。
    u8_TMOD_Temp=0x20; //即将把定时器 1 设置为：工作方式 2，初值自动重装的 8 位定时器。
    TMOD=TMOD&0x0f; //此寄存器低 4 位是跟定时器 0 相关，高 4 位是跟定时器 1 相关。先清零定时器 1。
    TMOD=TMOD|u8_TMOD_Temp; //把高 4 位的定时器 1 填入 0x2，低 4 位的定时器 0 保持不变。
    TH1=256-(11059200L/12/32/9600); //波特率为 9600。11059200 代表晶振 11.0592MHz，
    TL1=256-(11059200L/12/32/9600); //L 代表 long 的长类型数据。根据芯片手册提供的计算公式。
    TR1=1; //开启定时器 1

    SM0=0;
    SM1=1; //SM0 与 SM1 的设置：选择 10 位异步通信，波特率根据定时器 1 可变
    REN=1; //允许串口接收数据

    //为了保证串口中断接收的数据不丢失，必须设置 IP = 0x10，相当于把串口中断设置为最高优先级，
    //这个时候，串口中断可以打断任何其他的中断服务函数实现嵌套，
    IP =0x10; //把串口中断设置为最高优先级，必须的。

    ES=1; //允许串口中断
    EA=1; //允许总中断
}

void Delay(unsigned long u32DelayTime)
{

```

```

        for(;u32DelayTime>0;u32DelayTime--);
    }

void PeripheralInitial(void)
{

}

void BeepOpen(void)
{
    P3_4=0;
}

void BeepClose(void)
{
    P3_4=1;
}

void VoiceScan(void)
{

    static unsigned char Su8Lock=0;

    if(1==vGu8BeepTimerFlag&&vGu16BeepTimerCnt>0)
    {
        if(0==Su8Lock)
        {
            Su8Lock=1;
            BeepOpen();
        }
        else
        {

            vGu16BeepTimerCnt--;

            if(0==vGu16BeepTimerCnt)
            {
                Su8Lock=0;
                BeepClose();
            }
        }
    }
}

```

## 第一百三十一节： 灵活切换各种不同大小“接收内存”的串口程序框架。

### 【131.1 切换各种不同大小“接收内存”。】

很多 32 位的单片机，只要外挂 SRAM 或者 SDRAM 这类内存芯片，就可以轻松的把一个全局变量的数组开辟到几百 K 甚至几兆的容量。开辟这么大的数组，往往是用来处理一些文件类的大数据，比如串口接收一张 480x272 点阵大小的 BMP 格式的图片文件，就需要开辟一个几百 K 的全局变量大数组。串口通信中，从接收内存的容量来划分，常用有两种数据类型，一种是常规控制类（容量小），一种是文件类（容量大），要能做到在这两种“接收内存”中灵活切换，关键是用到“指针的中转切换”技术。

“常规控制类内存”负责两块事务，一块是接收“前部分的”[数据头，数据类型，数据长度]，另一块是“后部分的”[常规控制类的专用数据]。

“文件类内存”只负责“后部分的”[文件类的专用数据]，而“前部分的”[数据头，数据类型，数据长度]是需要借助“常规控制类内存”来实现的。

本节破题的关键在于，根据不同的数据类型，利用“指针的中转切换”实现不同接收内存的灵活切换。关键代码是串口中断函数这部分的处理，片段代码的讲解如下：

```
unsigned char Gu8ReceBuffer[20]; //常规控制类的小内存
unsigned char Gu8FileBuffer[40]; //文件类的大内存
unsigned char *pGu8ReceBuffer; //用来切换接收内存的“中转指针”
unsigned long Gu32ReceCntMax=20; //最大缓存（初始值 20 或者 40 都没关系，因为后续会动态改变）

void usart(void) interrupt 4
{
    if(1==RI)
    {
        RI = 0;

        if(0==Gu8FinishFlag)
        {
            Gu8ReceFeedDog=1;
            switch(Gu8ReceStep)
            {
                case 0: // “前部分的”数据头。接头暗号的步骤
                    Gu8ReceBuffer[0]=SBUF;
                    if(0xeb==Gu8ReceBuffer[0])
                    {
                        Gu32ReceCnt=1;
                        Gu8ReceStep=1;
                    }
                    break;

                case 1: // “前部分的”数据类型和长度
                    Gu8ReceBuffer[Gu32ReceCnt]=SBUF;
```

```

Gu32ReceCnt++;
if(Gu32ReceCnt>=6) //前 6 个数据。接收完了“数据类型”和“数据长度”。
{
    Gu8ReceType=Gu8ReceBuffer[1]; //提取“数据类型”
    pu32Data=(unsigned long *)&Gu8ReceBuffer[2];
    Gu32ReceDataLength=*pu32Data; //提取“数据长度”
    if(Gu32ReceCnt>=Gu32ReceDataLength) //靠“数据长度”来判断是否完成
    {
        Gu8FinishFlag=1; //接收完成标志“置 1”，通知主函数处理。
        Gu8ReceStep=0; //及时切换回接头暗号的步骤
    }
    else //如果还没结束，继续切换到下一个步骤，接收“有效数据”
    {
        //以下几行代码是本节的破题关键!!!
        if(0x02==Gu8ReceType) //如果是文件类，把指针关联到 Gu8FileBuffer
        {
            pGu8ReceBuffer=(unsigned char *)&Gu8FileBuffer[0]; //下标 0
            Gu32ReceCntMax=40+6; //最大缓存
        }
        else //如果是常规类，继续把指针关联到 Gu8ReceBuffer 本身的数组
        {
            pGu8ReceBuffer=(unsigned char *)&Gu8ReceBuffer[6]; //下标 6
            Gu32ReceCntMax=20; //最大缓存
        }

        Gu8ReceStep=2; //切换到下一个步骤
    }
}
break;
case 2: //“后部分的”数据
    pGu8ReceBuffer[Gu32ReceCnt-6]=SBUF; //这里的指针就是各种不同内存的化身!!!
    Gu32ReceCnt++; //每接收一个字节，数组下标都自加 1，为接收下一个数据做准备

    //靠“数据长度”来判断是否完成。也不允许超过数组的最大缓存的长度
    if(Gu32ReceCnt>=Gu32ReceDataLength || Gu32ReceCnt>=Gu32ReceCntMax)
    {
        Gu8FinishFlag=1; //接收完成标志“置 1”，通知主函数处理。
        Gu8ReceStep=0; //及时切换回接头暗号的步骤
    }
    break;
}
}
}
else //发送数据引起的中断

```



```

{
    TI = 0; //及时清除发送中断的标志，避免一直无缘无故的进入中断。
    //以下可以添加一个全局变量的标志位的相关代码，通知主函数已经发送完一个字节的數據了。
}
}

```

## 【131.2 通信协议。】

数据头 (EB)：占 1 个字节，作为“起始字节”，起到“接头暗号”的作用，平时用来过滤无关的数据。

数据类型 (01)：占用 1 个字节。数据类型是用来定义这串数据的用途。

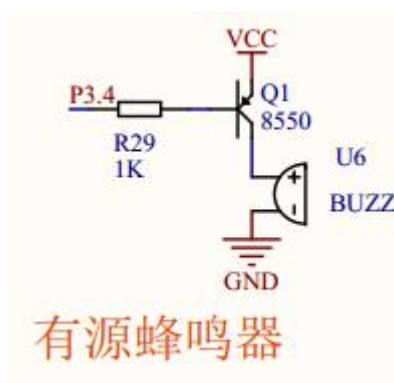
数据长度 (00 00 00 0B)：占 4 个字节。用来告诉通信的对方，这串数据一共有多少个字节。

其它数据 (03 E8)：此数据根据不同的“数据类型”可以用来做不同的用途，根据具体的项目而定。

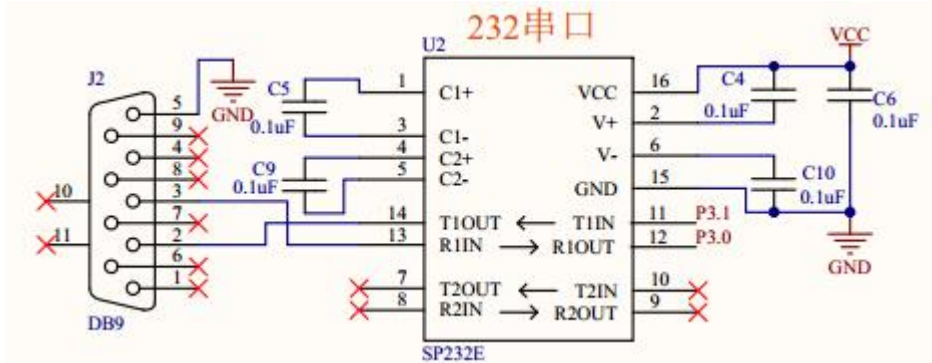
动态密匙 (00 01)：这两个字节代表一个 unsigned int 类型的数据，数据范围是从 0 到 65535，但是考虑到数据更加安全可靠，一般丢弃了首尾的 0（十六进制的 00 00）与 65535（十六进制的 FF FF），只保留从 1 到 65534 的变化。大部分的通信模型都是主机对从机的“一问一应答”模式，也就是，主机每发送一条指令给从机，从机才返回一条消息作为应答。如果主机发送了信息后，在规定的时间内，没有收到从机的应答指令，主机就继续发送信息给从机，但是此时，从机本来应该应答主机当前指令的，可能因为某种情况导致反馈的信息发生了延时，导致此时应答的数据是主机的上一条指令，从而造成“一问一应答”的数据帧发送了错位，这种情况加上“动态密匙”就能使问题得到有效的解决。主机每发送一条信息，信息里都携带了 2 个字节的“动态密匙”，从机每收到主机的一条信息，在应答此信息时都把收到的“动态密匙”原封不动的反馈给主机，主机再查看发送的“动态密匙”与接收到的“动态密匙”是否一致，以此来判断应答数据是否有效。“动态密匙”像流水号一样，每发送一次指令后都累加 1，不断发生变化，从 1 到 65534，依次循环。这是数据校验的一种方式。

异或 (0B)。“异或”放在数据串的最后一个字节，是前面所有字节的异或结果（不包括自己本身的字节）。比如：本例子中，数据串是：EB 01 00 00 00 0B 03 E8 00 01 0B。其中最后一个字节 0B 就是“异或”字节，前面所有字节相“异或”等于十六进制的 0B。验证“异或”的方法，可以借用电脑“附件”自带的“计算器”软件来实现，打开“计算器”软件后，在“查看”的下拉菜单里，选择“程序员”，然后选择“十六进制”，该计算器软件的异或运算按键是“Xor”。不管是主机还是从机，每接收到一串数据后，都要自己计算一次“异或”，把自己计算得到的“异或”与接收到的最后一个字节的“异或”进行对比，来判断接收到的数据是否发生了丢失或者错误。

## 【131.3 程序例程。】



上图 131.3.1 有源蜂鸣器电路



上图 131.3.2 232 串口电路

程序功能如下：

(9) 单片机模拟从机，上位机的串口助手模拟主机。在上位机的串口助手里，发送一串数据，控制蜂鸣器发出不同长度的声音。数据类型为 01 时，把“后部分的”数据发送给“常规控制类内存”；数据类型为 02 时，把“后部分的”数据发送给“文件类内存”。

(10) 本节因为还没有讲到数据发送的内容，因此应答“动态密匙”那部分的代码暂时不写，只写验证“异或”那部分的代码。

(11) 波特率 9600，校验位 NONE（无），数据位 8，停止位 1。

(12) 十六进制的数据格式：EB 01 00 00 00 0B XX XX YY YY ZZ 。其中：

EB 是数据头。

01 是代表数据类型。

00 00 00 0B 代表数据长度是 11 个（十进制）。

XX XX 代表一个 unsigned int 的数据，此数据的大小决定了蜂鸣器发出声音的长度。

YY YY 代表一个 unsigned int 的动态密匙，每收发一条指令，此数据累加一次 1，范围从 1 到 65534。

ZZ 代表前面所有字节的异或结果。

比如：

数据类型 01，“后部分的”数据发给“常规控制类内存”，让蜂鸣器鸣叫 1000 毫秒，密匙为 00 01，发送十六进制的：EB 01 00 00 00 0B 03 E8 00 01 0B

数据类型 02，“后部分的”数据发给“文件类内存”，让蜂鸣器鸣叫 100 毫秒，密匙为 00 02，发送十六进制的：EB 02 00 00 00 0B 00 64 00 02 84

```
#include "REG52.H"
```

```
#define RECE_TIME_OUT    2000 //通信过程中字节之间的超时时间 2000ms
```

```
#define REC_BUFFER_SIZE  20    //常规控制类数组的长度
```

```
#define FILE_BUFFER_SIZE 40    //文件类数组的长度
```

```

void usart(void); //串口接收的中断函数
void TO_time(); //定时器的中断函数

void UsartTask(void); //串口接收的任务函数，放在主函数内

void SystemInitial(void) ;
void Delay(unsigned long u32DelayTime) ;
void PeripheralInitial(void) ;

void BeepOpen(void);
void BeepClose(void);
void VoiceScan(void);

sbit P3_4=P3^4;

volatile unsigned char vGu8BeepTimerFlag=0;
volatile unsigned int vGu16BeepTimerCnt=0;

unsigned char Gu8ReceBuffer[REC_BUFFER_SIZE]; //常规控制类的小内存
unsigned char Gu8FileBuffer[FILE_BUFFER_SIZE]; //文件类的大内存
unsigned char *pGu8ReceBuffer; //用来切换接收内存的“中转指针”

unsigned long Gu32ReceCntMax=REC_BUFFER_SIZE; //最大缓存
unsigned long Gu32ReceCnt=0; //接收缓存数组的下标
unsigned char Gu8ReceStep=0; //接收中断函数里的步骤变量
unsigned char Gu8ReceFeedDog=1; //“喂狗”的操作变量。
unsigned char Gu8ReceType=0; //接收的数据类型
unsigned int Gu16ReceYY=0; //接收的动态密匙
unsigned char Gu8ReceZZ=0; //接收的异或
unsigned long Gu32ReceDataLength=0; //接收的数据长度
unsigned char Gu8FinishFlag=0; //是否已接收完成一串数据的标志
unsigned long *pu32Data; //用于数据转换的指针
volatile unsigned char vGu8ReceTimeOutFlag=0; //通信过程中字节之间的超时定时器的开关
volatile unsigned int vGu16ReceTimeOutCnt=0; //通信过程中字节之间的超时定时器，“喂狗”的对象

void main()
{
    SystemInitial();
    Delay(10000);
    PeripheralInitial();
    while(1)
    {
        UsartTask(); //串口接收的任务函数
    }
}

```

```

}

void usart(void) interrupt 4 //串口接发的中断函数，中断号为4
{
    if(1==RI) //接收完一个字节后引起的中断
    {
        RI = 0; //及时清零，避免一直无缘无故的进入中断。

/* 注释一：
* 以下 Gu8FinishFlag 变量的用途。
* 此变量一箭双雕，0 代表正处于接收数据的状态，1 代表已经接收完毕并且及时通知主函数中的处理函数
* UsartTask() 去处理新接收到的一串数据。除此之外，还起到一种“自锁自保护”的功能，在新数据还
* 没有被主函数处理完毕的时候，禁止接收其它新的数据，避免新数据覆盖了尚未处理的数据。
*/
        if(0==Gu8FinishFlag) //1 代表已经完成接收了一串新数据，并且禁止接收其它新的数据
        {

/* 注释二：
* 以下 Gu8ReceFeedDog 变量的用途。
* 此变量是用来检测并且识别通信过程中相邻的字节之间是否存在超时的情况。
* 如果大家听说过单片机中的“看门狗”这个概念，那么每接收到一个数据此变量就“置 1”一次，它的
* 作用就是起到及时“喂狗”的作用。每接收到一个数据此变量就“置 1”一次，在主函数里，相关
* 的定时器就会被重新赋值，只要这个定时器能不断及时的被补充新的“能量”新的值，那么这个定时器
* 就永远不会变成 0，只要不变成 0 就不会超时。如果两个字节之间通信时间超过了固定的长度，就意味
* 着此定时器变成了 0，这时就需要把中断函数里的接收步骤 Gu8Step 及时切换到“接头暗号”的步骤。
*/

        Gu8ReceFeedDog=1; //每接收到一个字节的数据，此标志就置 1 及时更新定时器的值。
        switch(Gu8ReceStep)
        {
            case 0: // “前部分的”数据头。接头暗号的步骤。
                Gu8ReceBuffer[0]=SBUF; //直接读取刚接收完的一个字节的数据。
                if(0xeb==Gu8ReceBuffer[0]) //等于数据头 0xeb，接头暗号吻合。
                {
                    Gu32ReceCnt=1; //接收缓存的下标
                    Gu8ReceStep=1; //切换到下一个步骤，接收其它有效的数据
                }
                break;

            case 1: // “前部分的”数据类型和长度
                Gu8ReceBuffer[Gu32ReceCnt]=SBUF; //直接读取刚接收完的一个字节的数据。
                Gu32ReceCnt++; //每接收一个字节，数组下标都自加 1，为接收下一个数据做准备
                if(Gu32ReceCnt>=6) //前 6 个数据。接收完了“数据类型”和“数据长度”。
                {
                    Gu8ReceType=Gu8ReceBuffer[1]; //提取“数据类型”

```

```

        //以下的数据转换，在第 62 节讲解过的指针法
        pu32Data=(unsigned long *)&Gu8ReceBuffer[2]; //数据转换
        Gu32ReceDataLength=*pu32Data; //提取“数据长度”
        if (Gu32ReceCnt>=Gu32ReceDataLength) //靠“数据长度”来判断是否完成
        {
            Gu8FinishFlag=1; //接收完成标志“置 1”，通知主函数处理。
            Gu8ReceStep=0;    //及时切换回接头暗号的步骤
        }
        else //如果还没结束，继续切换到下一个步骤，接收“有效数据”
        {
            //以下几行代码是本节的破题关键!!!
            if (0x02==Gu8ReceType) //如果是文件类，把指针关联到 Gu8FileBuffer
            {
                pGu8ReceBuffer=(unsigned char *)&Gu8FileBuffer[0]; //下标 0
                Gu32ReceCntMax=FILE_BUFFER_SIZE+6; //最大缓存
            }
            else //如果是常规类，继续把指针关联到 Gu8ReceBuffer 本身的数组
            {
                pGu8ReceBuffer=(unsigned char *)&Gu8ReceBuffer[6]; //下标 6
                Gu32ReceCntMax=REC_BUFFER_SIZE; //最大缓存
            }

            Gu8ReceStep=2; //切换到下一个步骤
        }
    }
    break;
case 2: // “后部分的”数据
    pGu8ReceBuffer[Gu32ReceCnt-6]=SBUF; //这里的指针就是各种不同内存的化身!!!
    Gu32ReceCnt++; //每接收一个字节，数组下标都自加 1，为接收下一个数据做准备

    //靠“数据长度”来判断是否完成。也不允许超过数组的最大缓存的长度
    if (Gu32ReceCnt>=Gu32ReceDataLength || Gu32ReceCnt>=Gu32ReceCntMax)
    {
        Gu8FinishFlag=1; //接收完成标志“置 1”，通知主函数处理。
        Gu8ReceStep=0;    //及时切换回接头暗号的步骤
    }
    break;
}
}
}
else //发送数据引起的中断
{
    TI = 0; //及时清除发送中断的标志，避免一直无缘无故的进入中断。
    //以下可以添加一个全局变量的标志位的相关代码，通知主函数已经发送完一个字节的數據了。

```

```

    }
}

void UsartTask(void)    //串口接收的任务函数，放在主函数内
{
    static unsigned int *pSul6Data; //数据转换的指针
    static unsigned int Sul6Data;   //转换后的数据
    static unsigned int i;
    static unsigned char Su8RecZZ=0; //计算的“异或”

    if(1==Gu8ReceFeedDog) //每被“喂一次狗”，就及时更新一次“超时检测的定时器”的初值
    {
        Gu8ReceFeedDog=0;

        vGu8ReceTimeOutFlag=0;
        vGu16ReceTimeOutCnt=RECE_TIME_OUT; //更新一次“超时检测的定时器”的初值
        vGu8ReceTimeOutFlag=1;
    }
    else if(Gu8ReceStep>0&&0==vGu16ReceTimeOutCnt) //超时，并且步骤不在接头暗号的步骤
    {
        Gu8ReceStep=0; //串口接收数据的中断函数及时切换回接头暗号的步骤
    }

    if(1==Gu8FinishFlag) //1 代表已经接收完毕一串新的数据，需要马上去处理
    {
        switch(Gu8ReceType) //接收到的数据类型
        {
            case 0x01: //常规控制类的小内存。驱动蜂鸣器
                //以下的数据转换，在第 62 节讲解过的指针法

                pSul6Data=(unsigned int *)&Gu8ReceBuffer[Gu32ReceDataLength-3]; //数据转换
                Gu16ReceYY=*pSul6Data; //提取“动态密钥”。本例子中暂时不做返回应答的处理

                Gu8ReceZZ=Gu8ReceBuffer[Gu32ReceDataLength-1]; //提取接收到的“异或”

                Su8RecZZ=Gu8ReceBuffer[0]; //提取数据串第“i=0”个数据作为异或的原始数据
                for(i=1;i<(Gu32ReceDataLength-1);i++) //注意，这里是从第“i=1”个数据开始
                {
                    Su8RecZZ=Su8RecZZ^Gu8ReceBuffer[i]; //计算“异或”
                }
            }
        }
    }
}

```

```

if(Su8RecZZ==Gu8ReceZZ) //验证“异或”，“计算的”与“接收的”是否一致
{
    pSu16Data=(unsigned int *)&Gu8ReceBuffer[6]; //数据转换。
    Su16Data=*pSu16Data; //提取“蜂鸣器声音的长度”

    vGu8BeepTimerFlag=0;
    vGu16BeepTimerCnt=Su16Data; //让蜂鸣器鸣叫
    vGu8BeepTimerFlag=1;
}

break;

case 0x02: //文件类的大内存。驱动蜂鸣器。
//以下的数据转换，在第 62 节讲解过的指针法

pSu16Data=(unsigned int *)&Gu8ReceBuffer[Gu32ReceDataLength-3]; //数据转换
Gu16ReceYY=*pSu16Data; //提取“动态密钥”。本例子中暂时不做返回应答的处理

//注意，请留意以下代码文件类内存数组 Gu8FileBuffer 的下标位置
Gu8ReceZZ=Gu8FileBuffer[Gu32ReceDataLength-1-6]; //提取接收到的“异或”

//前面 6 个字节是“前部分的”[数据头，数据类型，数据长度]
Su8RecZZ=Gu8ReceBuffer[0]; //提取数据串第“i=0”个数据作为异或的原始数据
for(i=1;i<6;i++) //注意，这里是从第“i=1”个数据开始
{
    Su8RecZZ=Su8RecZZ^Gu8ReceBuffer[i]; //计算“前部分的”“异或”
}

//6 个字节之后是“后部分的”“文件类专用的数据”
for(i=0;i<(Gu32ReceDataLength-1-6);i++)
{
    Su8RecZZ=Su8RecZZ^Gu8FileBuffer[i]; //计算“后部分的”“异或”
}

if(Su8RecZZ==Gu8ReceZZ) //验证“异或”，“计算的”与“接收的”是否一致
{
    pSu16Data=(unsigned int *)&Gu8FileBuffer[0]; //数据转换。此处下标 0!
    Su16Data=*pSu16Data; //提取“蜂鸣器声音的长度”

    vGu8BeepTimerFlag=0;
    vGu16BeepTimerCnt=Su16Data; //让蜂鸣器鸣叫
    vGu8BeepTimerFlag=1;
}

```

```

        break;

    }

    Gu8FinishFlag=0; //上面处理完数据再清零标志，为下一次接收新的数据做准备
}

void T0_time() interrupt 1
{
    VoiceScan();

    if(1==vGu8ReceTimeOutFlag&&vGu16ReceTimeOutCnt>0) //通信过程中字节之间的超时定时器
    {
        vGu16ReceTimeOutCnt--;
    }

    TH0=0xfc;
    TL0=0x66;
}

void SystemInitial(void)
{
    unsigned char u8_TMOD_Temp=0;

    //以下是定时器 0 的中断的配置
    TMOD=0x01;
    TH0=0xfc;
    TL0=0x66;
    EA=1;
    ET0=1;
    TR0=1;

    //以下是串口接收中断的配置
    //串口的波特率与内置的定时器 1 直接相关，因此配置此定时器 1 就等效于配置波特率。
    u8_TMOD_Temp=0x20; //即将把定时器 1 设置为：工作方式 2，初值自动重装的 8 位定时器。
    TMOD=TMOD&0x0f; //此寄存器低 4 位是跟定时器 0 相关，高 4 位是跟定时器 1 相关。先清零定时器 1。
    TMOD=TMOD|u8_TMOD_Temp; //把高 4 位的定时器 1 填入 0x2，低 4 位的定时器 0 保持不变。
    TH1=256-(11059200L/12/32/9600); //波特率为 9600。11059200 代表晶振 11.0592MHz，
    TL1=256-(11059200L/12/32/9600); //L 代表 long 的长类型数据。根据芯片手册提供的计算公式。

```



```

TR1=1; //开启定时器 1

SM0=0;
SM1=1; //SM0 与 SM1 的设置：选择 10 位异步通信，波特率根据定时器 1 可变
REN=1; //允许串口接收数据

//为了保证串口中断接收的数据不丢失，必须设置 IP = 0x10，相当于把串口中断设置为最高优先级，
//这个时候，串口中断可以打断任何其他的中断服务函数实现嵌套，
IP =0x10; //把串口中断设置为最高优先级，必须的。

ES=1;          //允许串口中断
EA=1;          //允许总中断
}

void Delay(unsigned long u32DelayTime)
{
    for(;u32DelayTime>0;u32DelayTime--);
}

void PeripheralInitial(void)
{
}

void BeepOpen(void)
{
    P3_4=0;
}

void BeepClose(void)
{
    P3_4=1;
}

void VoiceScan(void)
{
    static unsigned char Su8Lock=0;

    if(1==vGu8BeepTimerFlag&&vGu16BeepTimerCnt>0)
    {
        if(0==Su8Lock)
        {
            Su8Lock=1;

```

```
        BeepOpen() ;  
    }  
else  
{  
  
    vGu16BeepTimerCnt--;  
  
    if (0==vGu16BeepTimerCnt)  
    {  
        Su8Lock=0;  
        BeepClose() ;  
    }  
}  
}  
}
```

## 第一百三十二节：“转发、透传、多种协议并存”的双缓存串口程序框架。

### 【132.1 字节间隔时间、双缓存切换、指针切换关联。】

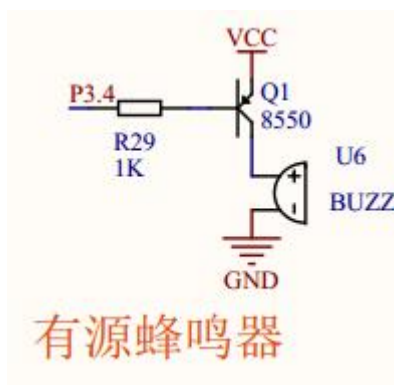
在一些通讯模块的项目中，常常涉及数据的转发，透传，提取关键字的处理，单片机接收到的数据不许随意丢失，必须全部暂存，然后提取关键字，再把整包数据“原封不动”或者“略作修改”转发给“下家”。这类项目的特点是，通讯协议不是固定唯一的，因此，前面章节那种接头暗号（数据头）的程序框架不适合这里，本节跟大家分享另外一种程序框架。

第一个要突破的技术难点是，既然通讯协议不是固定唯一的，那么，如何识别一串数据已经接收完毕？答案是靠接收每个字节之间的间隔时间来识别。当一串数据正在接收时，每个字节之间的间隔时间是“短暂的相对均匀的”。当一串数据已经接收完毕时，每个字节之间的间隔时间是“突然变长的”。代码的具体实现，是靠一个软件定时器，模拟单片机“看门狗”的“喂狗”原理。

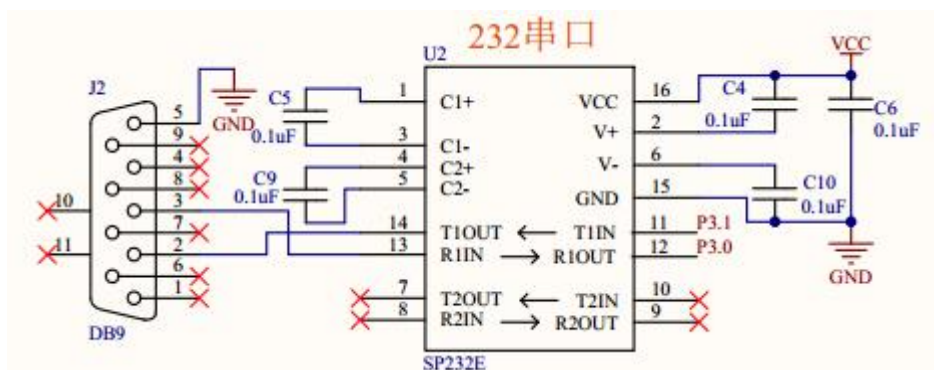
第二个要突破的技术难点是，既然通讯协议不是固定唯一的，数据内容带有随机性，甚至字节之间的间隔时间的长短也带有随机性和不确定性，那么，如何预防正在处理数据时突然“接收中断”又接收到的新数据覆盖了尚未来得及处理的旧数据，或者，如何预防正在处理旧数据时，丢失了“突然又新过来的本应该接收的新数据”？答案是用双缓存轮流切换的机制。双缓存，一个用在处理刚刚接收到的旧数据，另一个用在时刻准备着接收新数据，轮流切换，两不误。

第三个要突破的技术难点是，既然是用双缓存轮流切换的机制，那么，在主程序里如何统一便捷地处理两个缓存的数组？这里的“统一”是关键，要把两个数组“统一”成（看成是）一个数组，方法是，只需用“指针切换关联”的技术就可以了。

### 【132.2 程序例程。】



上图 132.2.1 有源蜂鸣器电路



上图 132.2.2 232 串口电路

程序功能如下：单片机接收任意长度（最大一次不超过 30 字节）的一串数据。如果发现连续有三个字节是 0x02 0x03 0x04，蜂鸣器则“短叫”100ms 提示；如果发现连续有四个字节是 0x06 0x07 0x08 0x09，蜂鸣器则“长叫”2000ms 提示。

比如测试“短叫”100ms，发送十六进制的数据串：05 02 00 00 02 03 04 09

比如测试“长叫”2000ms，发送十六进制的数据串：02 02 06 07 08 09 01 08 03 00 05

代码如下：

```
#include "REG52.H"

#define DOG_TIME_OUT 20 //理论上，9600 波特率的字节间隔时间大概 0.8ms 左右，因此取 20ms 足够
#define RECE_BUFFER_SIZE 30 //接收缓存的数组大小

void usart(void); //串口接收的中断函数
void T0_time(); //定时器的中断函数

void UsartTask(void); //串口接收的任务函数，放在主函数内

void SystemInitial(void) ;
void Delay(unsigned long u32DelayTime) ;
void PeripheralInitial(void) ;

void BeepOpen(void);
void BeepClose(void);
void VoiceScan(void);

sbit P3_4=P3^4;

unsigned char Gu8CurrentReceBuffer_Sec=0; //当前接收缓存的选择标志。0 代表缓存 A，1 代表缓存 B

unsigned char Gu8ReceBuffer_A[RECE_BUFFER_SIZE]; //双缓存其中之一的缓存 A
unsigned long Gu32ReceCnt_A=0; //缓存 A 的数组下标与计数器，必须初始化为 0，做好接收准备
```

```

unsigned char Gu8ReceBuffer_B[RECE_BUFFER_SIZE]; //双缓存其中之一的缓存 B
unsigned long Gu32ReceCnt_B=0;    //缓存 B 的数组下标与计数器，必须初始化为 0，做好接收准备

unsigned char Gu8ReceFeedDog=1; //“喂狗”的操作变量。
unsigned char Gu8FinishFlag=0; //接收完成标志。0 代表还没有完成，1 代表已经完成了一次接收

volatile unsigned char vGu8ReceTimeOutFlag=0; //通信过程中字节之间的超时定时器的开关
volatile unsigned int vGu16ReceTimeOutCnt=0; //通信过程中字节之间的超时定时器，“喂狗”的对象

volatile unsigned char vGu8BeepTimerFlag=0;
volatile unsigned int vGu16BeepTimerCnt=0;

void main()
{
    SystemInitial();
    Delay(10000);
    PeripheralInitial();
    while(1)
    {
        UsartTask();    //串口接收的任务函数
    }
}

void usart(void) interrupt 4
{
    if(1==RI)
    {
        RI = 0;

        Gu8FinishFlag=0; //此处也清零，意味深长，当主函数正在处理数据时，可以兼容多次接收完成
        Gu8ReceFeedDog=1; //看门狗的“喂狗”操作，给软件定时器继续“输血”
        if(0==Gu8CurrentReceBuffer_Sec)    //0 代表选择缓存 A
        {
            if(Gu32ReceCnt_A<RECE_BUFFER_SIZE)
            {
                Gu8ReceBuffer_A[Gu32ReceCnt_A]=SBUF;
                Gu32ReceCnt_A++; //记录当前缓存 A 的接收字节数
            }
        }
        else    //1 代表选择缓存 B
        {
            if(Gu32ReceCnt_B<RECE_BUFFER_SIZE)
            {

```

```

        Gu8ReceBuffer_B[Gu32ReceCnt_B]=SBUF;
        Gu32ReceCnt_B++; //记录当前缓存 B 的接收字节数
    }

}

else //发送数据引起的中断
{
    TI = 0; //及时清除发送中断的标志，避免一直无缘无故的进入中断。
    //以下可以添加一个全局变量的标志位的相关代码，通知主函数已经发送完一个字节的數據了。
}
}

void UsartTask(void) //串口接收的任务函数，放在主函数内
{
    static unsigned char *pSu8ReceBuffer; //“指针切换关联”中的指针，切换内存
    static unsigned char Su8Lock=0; //用来避免一直更新的临时变量
    static unsigned long i; //用在数据处理中的循环变量
    static unsigned long Su32ReceSize=0; //接收到的数据大小的临时变量

    if(1==Gu8ReceFeedDog) //每被“喂一次狗”，就及时更新一次“超时检测的定时器”的初值
    {
        Gu8ReceFeedDog=0;

        Su8Lock=0; //解锁。用来避免一直更新的临时变量

        //以下三行代码是看门狗中的“喂狗”操作。继续给软件定时器“输血”
        vGu8ReceTimeOutFlag=0;
        vGu16ReceTimeOutCnt=DOG_TIME_OUT; //正在通信时，两个字节间隔的最大时间，本节选用 20ms
        vGu8ReceTimeOutFlag=1;
    }
    else if(0==Su8Lock&&0==vGu16ReceTimeOutCnt) //超时，代表一串数据已经接收完成
    {
        Su8Lock=1; //避免一直进来更新
        Gu8FinishFlag=1; //两个字节之间的时间超时，因此代表了一串数据已经接收完成
    }

    if(1==Gu8FinishFlag) //1 代表已经接收完毕一串新的数据，需要马上去处理
    {
        if(0==Gu8CurrentReceBuffer_Sec)
        {
            Gu8CurrentReceBuffer_Sec=1; //以最快的速度先切换接收内存，避免丢失新发过来的数据

```

```

//Gu32ReceCnt_B=0; //这里不能清零缓存 B 的计数器，意味深长，避免此处临界点发生中断
Gu8FinishFlag=0; //尽可能以最快的速度清零本次完成的标志，为下一次新数据做准备
pSu8ReceBuffer=(unsigned char *)&Gu8ReceBuffer_A[0]; //关联刚刚接收的数据缓存
Su32ReceSize=Gu32ReceCnt_A; //记录当前缓存的有效字节数
Gu32ReceCnt_A=0; //及时把当前缓存计数清零，为一次切换接收缓存做准备。意味深长。
}
else
{
    Gu8CurrentReceBuffer_Sec=0; //以最快的速度先切换接收内存，避免丢失新发过来的数据
    //Gu32ReceCnt_A=0; //这里不能清零缓存 A 的计数器，意味深长，避免此处临界点发生中断
    Gu8FinishFlag=0; //尽可能以最快的速度清零本次完成的标志，为下一次新数据做准备
    pSu8ReceBuffer=(unsigned char *)&Gu8ReceBuffer_B[0]; //关联刚刚接收的数据缓存
    Su32ReceSize=Gu32ReceCnt_B; //记录当前缓存的有效字节数
    Gu32ReceCnt_B=0; //及时把当前缓存计数清零，为一次切换接收缓存做准备。意味深长。
}

//Gu8FinishFlag=0; //之所以不选择在这里清零，是因为在上面清零更及时快速。意味深长。

//开始处理刚刚接收到的一串新数据，直接“统一”处理 pSu8ReceBuffer 指针为代表的数据即可
for(i=0;i<Su32ReceSize;i++)
{
    if(0x02==pSu8ReceBuffer[i]&&
        0x03==pSu8ReceBuffer[i+1]&&
        0x04==pSu8ReceBuffer[i+2]) //连续三个数是 0x02 0x03 0x04
    {
        vGu8BeepTimerFlag=0;
        vGu16BeepTimerCnt=100; //让蜂鸣器“短叫”100ms
        vGu8BeepTimerFlag=1;
        return; //直接退出当前函数
    }

    if(0x06==pSu8ReceBuffer[i]&&
        0x07==pSu8ReceBuffer[i+1]&&
        0x08==pSu8ReceBuffer[i+2]&&
        0x09==pSu8ReceBuffer[i+3]) //连续四个数是 0x06 0x07 0x08 0x09
    {
        vGu8BeepTimerFlag=0;
        vGu16BeepTimerCnt=2000; //让蜂鸣器“长叫”2000ms
        vGu8BeepTimerFlag=1;
        return; //直接退出当前函数
    }
}
}

```

```

    }
}

void T0_time() interrupt 1
{
    VoiceScan();

    if(1==vGu8ReceTimeOutFlag&&vGu16ReceTimeOutCnt>0) //通信过程中字节之间的超时定时器
    {
        vGu16ReceTimeOutCnt--;
    }

    TH0=0xfc;
    TL0=0x66;
}

void SystemInitial(void)
{
    unsigned char u8_TMOD_Temp=0;

    //以下是定时器 0 的中断的配置
    TMOD=0x01;
    TH0=0xfc;
    TL0=0x66;
    EA=1;
    ET0=1;
    TR0=1;

    //以下是串口接收中断的配置
    //串口的波特率与内置的定时器 1 直接相关，因此配置此定时器 1 就等效于配置波特率。
    u8_TMOD_Temp=0x20; //即将把定时器 1 设置为：工作方式 2，初值自动重装的 8 位定时器。
    TMOD=TMOD&0x0f; //此寄存器低 4 位是跟定时器 0 相关，高 4 位是跟定时器 1 相关。先清零定时器 1。
    TMOD=TMOD|u8_TMOD_Temp; //把高 4 位的定时器 1 填入 0x2，低 4 位的定时器 0 保持不变。
    TH1=256-(11059200L/12/32/9600); //波特率为 9600。11059200 代表晶振 11.0592MHz，
    TL1=256-(11059200L/12/32/9600); //L 代表 long 的长类型数据。根据芯片手册提供的计算公式。
    TR1=1; //开启定时器 1

    SM0=0;
    SM1=1; //SM0 与 SM1 的设置：选择 10 位异步通信，波特率根据定时器 1 可变
    REN=1; //允许串口接收数据

    //为了保证串口中断接收的数据不丢失，必须设置 IP = 0x10，相当于把串口中断设置为最高优先级，
    //这个时候，串口中断可以打断任何其他的中断服务函数实现嵌套，

```



```

    IP =0x10;  //把串口中断设置为最高优先级，必须的。

    ES=1;      //允许串口中断
    EA=1;      //允许总中断
}

void Delay(unsigned long u32DelayTime)
{
    for(;u32DelayTime>0;u32DelayTime--);
}

void PeripheralInitial(void)
{
}

void BeepOpen(void)
{
    P3_4=0;
}

void BeepClose(void)
{
    P3_4=1;
}

void VoiceScan(void)
{
    static unsigned char Su8Lock=0;

    if(1==vGu8BeepTimerFlag&&vGu16BeepTimerCnt>0)
    {
        if(0==Su8Lock)
        {
            Su8Lock=1;
            BeepOpen();
        }
        else
        {

            vGu16BeepTimerCnt--;

            if(0==vGu16BeepTimerCnt)

```

```
        {  
            Su8Lock=0;  
            BeepClose();  
        }  
    }  
}
```

## 第一百三十三节：常用的三种串口发送函数。

### 【133.1 发送单字节的底层驱动函数。】

单片机内置的“独立硬件串口模块”能直接实现“发送一个字节数据”的基础功能，因此，发送单字节的函数是应用层与硬件层的最小单位的接口函数，也称为底层驱动函数。应用层再复杂的发送函数都基于此最小单位的接口函数来实现。单片机应用层与“独立硬件串口模块”之间的接口通信是靠寄存器 SBUF 作为中间载体的，要实现发送单字节的最小接口函数，有如下三个关键点。

第一个，单片机应用层如何知道“硬件模块”已经发送完了一个字节，靠什么来识别？答：在初始化函数里，可以把“硬件模块”配置成，每发送完一个字节后都产生一次发送中断，在发送中断函数里让一个全局变量从 0 变成 1，依此全局变量作为识别是否已经发送完一个字节的标志。

第二个，发送一个字节数据的时候，如果“硬件模块”通讯异常，没有按预期产生发送中断，单片机就会一直处于死循环等待“完成标志”的状态，怎么办？答：在等待“完成标志”的时候，加入超时处理的机制。

第三个，在连续发送一堆数据时，如果接收方（或者上位机）发现有丢失数据的时候，如何调节此发送函数？答：可以根据实际调试的结果，如果接收方发现丢失数据，可以尝试在每发送一个字节之后插入一个 Delay 延时，延时的时间长度根据实际调试为准。我个人的经验中，感觉 stm32 这类 M3 核或者 M4 核的单片机在发送一个字节的时候只需判断是否发送完成的标志位即可，不需要插入 Delay 延时。但是在其它某些个别厂家单片机的串口发送数据中，是需要插入 Delay 延时作为调节，否则在连续发送一堆数据时会丢失数据，这个，应该以实际调试项目为准。

片段的讲解代码如下：

```
unsigned char Gu8ReceData;
unsigned char Gu8SendByteFinish=0; //发送一个字节完成的标志
void usart(void) interrupt 4 //串口的中断函数
{
    if(1==RI)
    {
        RI = 0;
        Gu8ReceData=SBUF;
    }
    else //发送数据引起的中断
    {
        TI = 0; //及时清除发送中断的标志，避免一直无缘无故的进入中断。
        Gu8SendByteFinish=1; //从 0 变成 1 通知主函数已经发送完一个字节的的数据了。
    }
}

void UsartSendByteData(unsigned char u8SendData) //发送一个字节的底层驱动函数
{
    static unsigned int Su16TimeOutDelay; //超时处理的延时计时器

    Gu8SendByteFinish=0; //在发送一个字节之前，必须先把此全局变量的标志清零。
    SBUF =u8SendData; //依靠寄存器 SBUF 作为载体发送一个字节的的数据
```

```

    Su16TimeOutDelay=0xffff; //超时处理的延时计时器装载一个相对合理的计时初始值
    while(Su16TimeOutDelay>0) //超时处理
    {
        if(1==Gu8SendByteFinish)
        {
            break; //如果 Gu8SendByteFinish 为 1，则发送一个字节完成，退出当前循环等待。
        }
        Su16TimeOutDelay--; //超时计时器不断递减
    }

    //Delay(); //在实际应用中，当连续发送一堆数据时如果发现丢失数据，可以尝试在此增加延时
}

```

### 【133.2 发送任意起始位置任意长度的函数。】

要连续发送一堆数据，必须先把这堆数据封装成一个数组，然后编写一个发送数组的函数。该函数内部是基于“发送单字节的最小接口函数”来实现的。该函数对外通常需要两个接口，一个是数组的任意起始位置，一个发送的数据长度。数组的任意起始位置只需靠指针即可实现。片段的讲解代码如下：

```

//任意数组
unsigned char Gu8SendBuffer[11]={0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09,0x0A};

//发送任意起始位置任意长度的函数
void UsartSendBuffer(const unsigned char *pCu8SendBuffer,unsigned long u32SendSize)
{
    static unsigned long i;
    for(i=0;i<u32SendSize;i++) //u32SendSize 为发送的数据长度
    {
        UsartSendByteData(pCu8SendBuffer[i]); //基于“发送单字节的最小接口函数”来实现的
    }
}

void main()
{
    UsartSendBuffer((const unsigned char *)&Gu8SendBuffer[0],5);//从第 0 位置发送 5 个数据
    UsartSendBuffer((const unsigned char *)&Gu8SendBuffer[6],5);//从第 6 位置发送 5 个数据
    while(1)
    {

    }
}

```

### 【133.3 发送带协议的函数。】

前面章节中，我们讲过接收“带固定协议”的程序框架，这类“带固定协议”的数据串里本身就自带了“数据的长度”，因此，要编程一个发送带协议的函数，关键在于，在函数内部根据协议先提取整串数据的有效长度。该函数对外通常也需要两个接口，一个是数组的起始位置，一个发送数据的最大限制长度。最大限制长度的作用是用来防止数组越界，增强程序的安全性。片段的讲解代码如下：

```
// “固定协议”十六进制的数据格式：EB 01 00 00 00 0B 03 E8 00 01 0B 。其中：
// EB 是数据头。
// 01 是代表数据类型。
// 00 00 00 0B 代表数据长度是 11 个（十进制）。
// 03 E8 00 01 0B 代表其它数据

// “带固定协议”的数组
unsigned char Gu8SendMessage[11]={0xEB, 0x01, 0x00, 0x00, 0x00, 0x0B, 0x03, 0xE8, 0x00, 0x01, 0x0B};

//发送带协议的函数
void UsartSendMessage(const unsigned char *pCu8SendMessage,unsigned long u32SendMaxSize)
{
    static unsigned long i;
    static unsigned long *pSu32;
    static unsigned long u32SendSize;

    pSu32=(const unsigned long *)&pCu8SendMessage[2];
    u32SendSize=*pSu32; //从带协议的数组中提取整包数组的有效发送长度

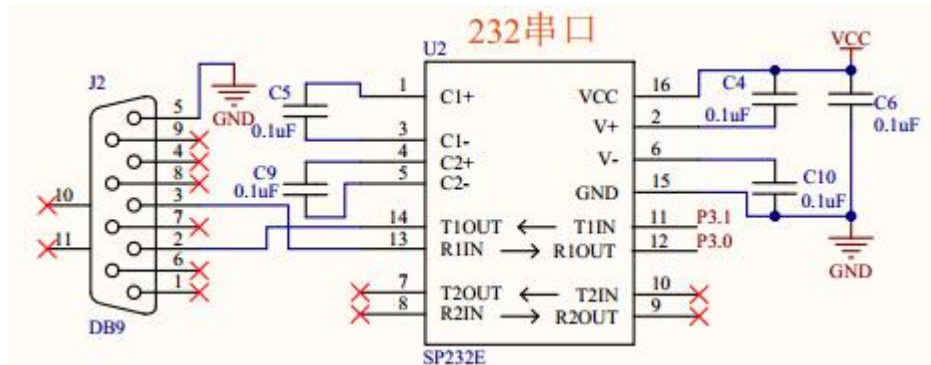
    if(u32SendSize>u32SendMaxSize) //如果“有效发送长度”大于“最大限制的长度”，数据异常
    {
        return; //数据异常，直接退出当前函数，预防数组越界
    }

    for(i=0;i<u32SendSize;i++) //u32SendSize 为发送的数据长度
    {
        UsartSendByteData(pCu8SendMessage[i]); //基于“发送单字节的最小接口函数”来实现的
    }
}

void main()
{
    UsartSendMessage((const unsigned char *)&Gu8SendMessage[0],100); //必须从第 0 位置发送
    while(1)
    {

    }
}
```

#### 【133.4 程序例程。】



上图 133.4.1 232 串口电路

程序功能如下：

单片机上电瞬间，直接发送三串数据。

第一串是十六进制的任意数据：00 01 02 03 04

第二串是十六进制的任意数据：06 07 08 09 0A

第三串是十六进制的“带协议”数据：EB 01 00 00 00 0B 03 E8 00 01 0B

波特率 9600，校验位 NONE（无），数据位 8，停止位 1。在电脑的串口助手软件里，设置接收显示的为“十六进制”（HEX 模式），即可观察到发送的三串数据。

代码如下：

```
#include "REG52.H"

void UartSendByteData(unsigned char u8SendData); //发送一个字节的底层驱动函数

//发送任意起始位置任意长度的函数
void UartSendBuffer(const unsigned char *pCu8SendBuffer,unsigned long u32SendSize);

//发送带协议的函数
void UartSendMessage(const unsigned char *pCu8SendMessage,unsigned long u32SendMaxSize);

void usart(void); //串口接收的中断函数
void SystemInitial(void);
void Delay(unsigned long u32DelayTime);
void PeripheralInitial(void);

unsigned char Gu8ReceData;
unsigned char Gu8SendByteFinish=0; //发送一个字节完成的标志
```

```

//任意数组
unsigned char Gu8SendBuffer[11]={0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09,0x0A};

// “固定协议”十六进制的数据格式：EB 01 00 00 00 0B 03 E8 00 01 0B 。其中：
// EB 是数据头。
// 01 是代表数据类型。
// 00 00 00 0B 代表数据长度是 11 个（十进制）。
// 03 E8 00 01 0B 代表其它数据
// “带固定协议”的数组
unsigned char Gu8SendMessage[11]={0xEB,0x01,0x00,0x00,0x00,0x0B,0x03,0xE8,0x00,0x01,0x0B};

void main()
{
    SystemInitial();
    Delay(10000);
    PeripheralInitial();    //在此函数内部调用了发送的三串数据
    while(1)
    {
    }
}

void usart(void) interrupt 4    //串口的中断函数
{
    if(1==RI)
    {
        RI = 0;
        Gu8ReceData=SBUF;
    }
    else //发送数据引起的中断
    {
        TI = 0; //及时清除发送中断的标志，避免一直无缘无故的进入中断。
        Gu8SendByteFinish=1; //从 0 变成 1 通知主函数已经发送完一个字节的數據了。
    }
}

void UsartSendByteData(unsigned char u8SendData) //发送一个字节的底层驱动函数
{
    static unsigned int Su16TimeOutDelay; //超时处理的延时计时器

    Gu8SendByteFinish=0; //在发送以字节之前，必须先把此全局变量的标志清零。
    SBUF =u8SendData; //依靠寄存器 SBUF 作为载体发送一个字节的數據
    Su16TimeOutDelay=0xffff; //超时处理的延时计时器装载一个相对合理的计时初始值
    while(Su16TimeOutDelay>0) //超时处理
    {

```

```

        if(1==Gu8SendByteFinish)
        {
            break; //如果 Gu8SendByteFinish 为 1，则发送一个字节完成，退出当前循环等待。
        }
        Su16TimeOutDelay--; //超时计时器不断递减
    }

    //Delay(); //在实际应用中，当连续发送一堆数据时如果发现丢失数据，可以尝试在此增加延时
}

//发送任意起始位置任意长度的函数
void UsartSendBuffer(const unsigned char *pCu8SendBuffer,unsigned long u32SendSize)
{
    static unsigned long i;
    for(i=0;i<u32SendSize;i++) //u32SendSize 为发送的数据长度
    {
        UsartSendByteData(pCu8SendBuffer[i]); //基于“发送单字节的最小接口函数”来实现的
    }
}

//发送带协议的函数
void UsartSendMessage(const unsigned char *pCu8SendMessage,unsigned long u32SendMaxSize)
{
    static unsigned long i;
    static unsigned long *pSu32;
    static unsigned long u32SendSize;

    pSu32=(const unsigned long *)&pCu8SendMessage[2];
    u32SendSize=*pSu32; //从带协议的数组中提取整包数组的有效发送长度

    if(u32SendSize>u32SendMaxSize) //如果“有效发送长度”大于“最大限制的长度”，数据异常
    {
        return; //数据异常，直接退出当前函数，预防数组越界
    }

    for(i=0;i<u32SendSize;i++) //u32SendSize 为发送的数据长度
    {
        UsartSendByteData(pCu8SendMessage[i]); //基于“发送单字节的最小接口函数”来实现的
    }
}

void SystemInitial(void)
{
    unsigned char u8_TMOD_Temp=0;

```



```

//以下是定时器 0 的中断的配置
TMOD=0x01;
TH0=0xfc;
TL0=0x66;
EA=1;
ET0=1;
TR0=1;

//以下是串口接收中断的配置
//串口的波特率与内置的定时器 1 直接相关，因此配置此定时器 1 就等效于配置波特率。
u8_TMOD_Temp=0x20; //即将把定时器 1 设置为：工作方式 2，初值自动重装的 8 位定时器。
TMOD=TMOD&0x0f; //此寄存器低 4 位是跟定时器 0 相关，高 4 位是跟定时器 1 相关。先清零定时器 1。
TMOD=TMOD|u8_TMOD_Temp; //把高 4 位的定时器 1 填入 0x2，低 4 位的定时器 0 保持不变。
TH1=256-(11059200L/12/32/9600); //波特率为 9600。11059200 代表晶振 11.0592MHz，
TL1=256-(11059200L/12/32/9600); //L 代表 long 的长类型数据。根据芯片手册提供的计算公式。
TR1=1; //开启定时器 1

SM0=0;
SM1=1; //SM0 与 SM1 的设置：选择 10 位异步通信，波特率根据定时器 1 可变
REN=1; //允许串口接收数据

//为了保证串口中断接收的数据不丢失，必须设置 IP = 0x10，相当于把串口中断设置为最高优先级，
//这个时候，串口中断可以打断任何其他的中断服务函数实现嵌套，
IP =0x10; //把串口中断设置为最高优先级，必须的。

ES=1; //允许串口中断
EA=1; //允许总中断
}

void Delay(unsigned long u32DelayTime)
{
    for(;u32DelayTime>0;u32DelayTime--);
}

void PeripheralInitial(void)
{
    //发送任意数组
    UsartSendBuffer((const unsigned char *)&Gu8SendBuffer[0],5); //从第 0 位置发送 5 个数据
    UsartSendBuffer((const unsigned char *)&Gu8SendBuffer[6],5); //从第 6 位置发送 5 个数据

    //发送带协议的数组
    UsartSendMessage((const unsigned char *)&Gu8SendMessage[0],100); //必须从第 0 位置发送
}

```

## 第一百三十四节：“应用层半双工”双机串口通讯的程序框架。

### 【134.1 应用层的“半双工”和“全双工”。】

应用层的“半双工”。主机与从机在程序应用层采用“一问一答”的查询模式，主机是主动方，从机是被动方，主机问一句从机答一句，“聊天对话”的氛围很无趣很呆板。从机没有发言权，当从机想主动给主机发送一些数据时就“憋得慌”。半双工适用于大多数单向通讯的场合。

应用层的“全双工”。主机与从机在程序应用层可以实现任意双向的通讯，这时从机也可变为主机，主机也可变为从机，就像两个人平时聊天，无所谓谁是从机谁是主机，也无所谓非要对方对我每句话都要应答附和（只要对方能听得清我讲什么就可以），“聊天对话”的氛围很生动很活泼。全双工适用于通讯更复杂的场合。

本节从“半双工”开始讲，让初学者先熟悉双机通讯的基本程序框架，下一节再讲“全双工”。

### 【134.2 双机通讯的三类核心函数。】

双机通讯在程序框架层面有三类核心的函数，它们分别是：通过程的控制函数，发送的队列驱动函数，接收数据后的处理函数。

“通过程的控制函数”的数量可以不止 1 个，每一个通讯事件都对应一个独立的“通过程的控制函数”，根据通讯事件的数量，一个系统往往有 N 个“通过程的控制函数”。顾名思义，它负责过程的控制，无论什么项目，凡是过程控制我都首选 switch 语句。此函数是属于上层应用的函数，它的基础底层是“发送的队列驱动函数”和“接收数据后的处理函数”这两个函数。

“发送的队列驱动函数”在系统中只有 1 个“发送的队列驱动函数”，负责“通讯管道的占用”的分配，负责数据的具体发送。当同时存在很多“待发送”的请求指令时，此函数会根据“if, else if...”的优先级，像队列一样安排各指令发送的先后顺序，确保各指令不会发生冲突。此函数属于底层的驱动函数。

“接收数据后的处理函数”在系统中只有 1 个，负责处理当前接收到的数据，它既属于“底层函数”也属于“应用层函数”，二者成分皆有。

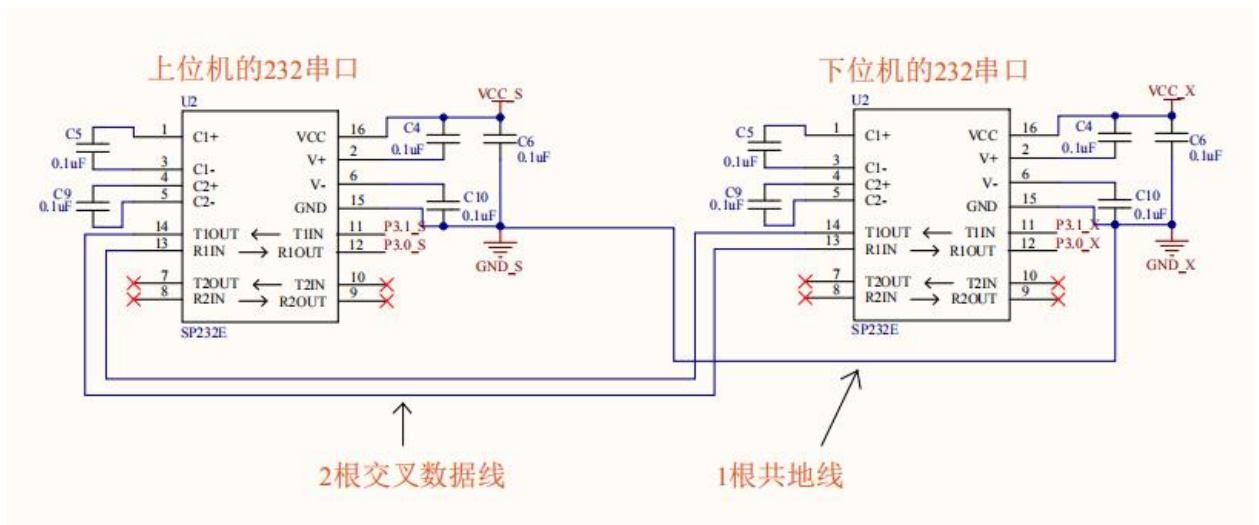
我们一旦深刻地领悟了这三类函数各自的分工与关联方式，将来应付再复杂的通讯系统都会脉络清晰，游刃有余。

### 【134.3 例程的功能需求。】

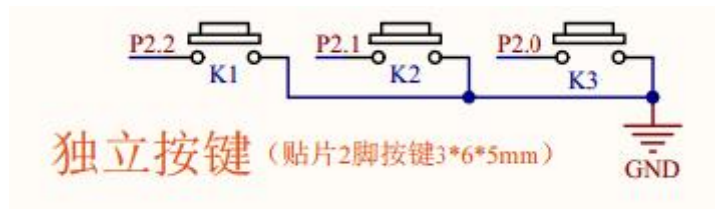
上位机与下位机都有一个一模一样的 57 个字节的大数组。在上位机端按下独立按键 K1 后，上位机开始与下位机建立通讯，上位机的目的是读取下位机的那 57 个字节的大数组，分批读取，每批读取 10 个字节，最后一批读取的是余下的 7 个字节。读取完毕后，上位机把读取到的大数组与自己的大数组进行对比：如果相等，表示通讯正确，蜂鸣器“长鸣”一声；如果不相等，表示通讯错误，蜂鸣器“短鸣”一声。在通讯过程中，如果出现通信异常（比如因为接收超时或者接收某批次数据错误而导致重发的次数超过最大限制的次数）也表示通讯错误，蜂鸣器也会发出“短鸣”一声的提示。

### 【134.4 例程的电路图。】

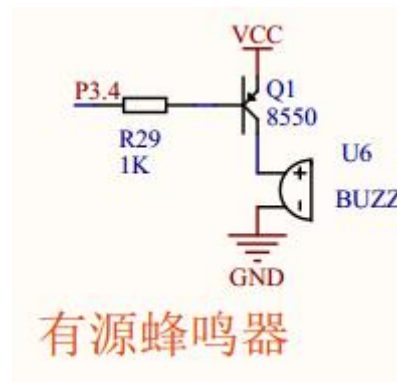
两个单片机进行 232 串口通讯，一共需要 3 根线：1 根作为共地线，其它 2 根是交叉的收发数据线（上位机的“接收线”连接下位机的“发送线”，上位机的“发送线”连接下位机的“接收线”），如下图所示：



上图 134. 4. 1 双机通讯的 232 串口接线图



上图 134. 4. 2 上位机的独立按键



上图 134. 4. 3 上位机的有源蜂鸣器

## 【134. 5 例程的通讯协议。】

(一) 通讯参数。波特率 9600，校验位 NONE（无），数据位 8，停止位 1。

(二) 上位机读取下位机的数组容量的大小的指令。

(1) 上位机发送十六进制的数据：EB 01 00 00 00 07 ED。

EB 是数据头。

01 是指令类型，01 代表请求下位机返回大数据的容量大小。

00 00 00 07 代表整个指令的数据长度。

ED 是前面所有字节数据的异或结果，用来作为校验数据。

(2) 下位机返回十六进制的数据：EB 01 00 00 00 0C XX XX XX XX ZZ。

EB 是数据头。

01 是指令类型，01 代表返回大数组的容量大小。

00 00 00 0B 代表整个指令的数据长度

XX XX XX XX 代表大数组的容量大小

ZZ 是前面所有字节数据的异或结果，用来作为校验数据。

(三) 上位机读取下位机的大数组的分段数据的指令。

(1) 上位机发送十六进制的数据：EB 02 00 00 00 0F RR RR RR RR YY YY YY YY ZZ

EB 是数据头

02 是指令类型，02 代表请求下位机返回当前分段的数据。

00 00 00 0F 代表整个指令的数据长度

RR RR RR RR 代表请求下位机返回的数据的“请求起始地址”

YY YY YY YY 代表请求下位机从“请求起始地址”一次返回的数据长度

ZZ 是前面所有字节数据的异或结果，用来作为校验数据。

(2) 下位机返回十六进制的数据：EB 02 TT TT TT TT RR RR RR RR YY YY YY YY HH ...HH ZZ

EB 是数据头

02 是指令类型，02 代表返回大数组当前分段的数据

TT TT TT TT 代表整个指令的数据长度

RR RR RR RR 代表下位机返回数据时的“请求起始地址”

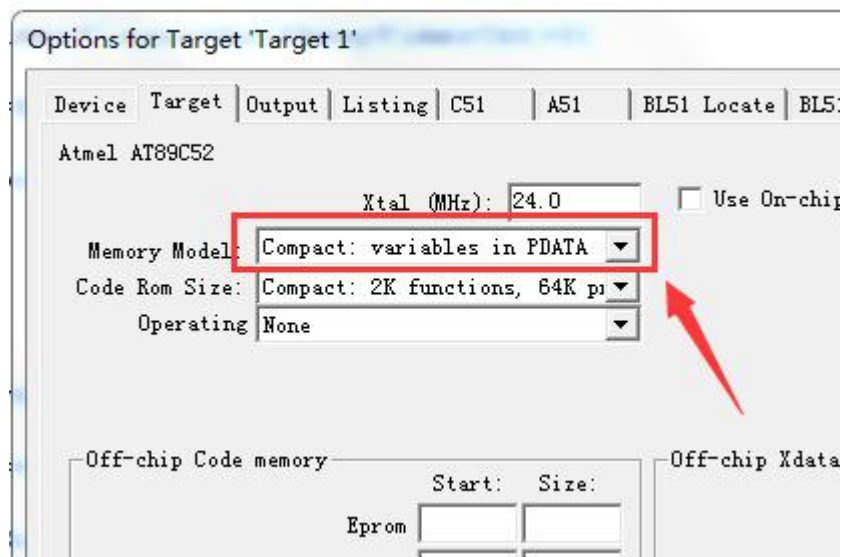
YY YY YY YY 代表下位机从“请求起始地址”一次返回的数据长度

HH ...HH 代表中间有效的数据内容

ZZ 是前面所有字节数据的异或结果，用来作为校验数据。

### 【134.6 解决本节例程编译不过去的方法。】

因为本节用到的全局变量比较多，如果有初学者在编译的时候出现“error C249: 'DATA': SEGMENT TOO LARGE”的提示，请按下图的窗口提示来设置一下编译的环境。



上图 134.5.1 设置编译的环境

### 【134.7 例程的上位机程序。】

```
#include "REG52.H"

#define RECE_TIME_OUT    2000 //通讯过程中字节之间的超时时间 2000ms
#define REC_BUFFER_SIZE  30    //常规控制类数组的长度
#define KEY_FILTER_TIME  25    //按键滤波的“稳定时间”

void usart(void); //串口接收的中断函数
void TO_time();  //定时器的中断函数

void BigBufferUsart(void); //读取下位机大数组的“通讯过程的控制函数”。三大核心函数之一
void QueueSend(void);      //发送的队列驱动函数。三大核心函数之一
void ReceDataHandle(void); //接收数据后的处理函数。三大核心函数之一

void UsartTask(void); //串口收发的任务函数，放在主函数内

unsigned char CalculateXor(const unsigned char *pCu8Buffer, //异或的算法函数
                           unsigned long u32BufferSize);

//比较两个数组的是否相等。返回 1 代表相等，返回 0 代表不相等
//u32BufferSize 是参与对比的数组的大小
unsigned char CmpTwoBufferIsSame(const unsigned char *pCu8Buffer_1,
                                  const unsigned char *pCu8Buffer_2,
                                  unsigned long u32BufferSize);

void UsartSendByteData(unsigned char u8SendData); //发送一个字节的底层驱动函数
```

```

//发送带协议的函数
void UsartSendMessage(const unsigned char *pCu8SendMessage,unsigned long u32SendMaxSize);

void SystemInitial(void) ;
void Delay(unsigned long u32DelayTime) ;
void PeripheralInitial(void) ;

void BeepOpen(void);
void BeepClose(void);
void VoiceScan(void);
void KeyScan(void);
void KeyTask(void);

sbit P3_4=P3^4;          //蜂鸣器的驱动输出口
sbit KEY_INPUT1=P2^2;    //K1 按键识别的输入口。

//下面表格数组的数据与下位机的表格数据一模一样，目的用来检测接收到的数据是否正确
code unsigned char Cu8TestTable[]=
{
0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09,0x0A,
0x11,0x12,0x13,0x14,0x15,0x16,0x17,0x18,0x19,0x1A,
0x21,0x22,0x23,0x24,0x25,0x26,0x27,0x28,0x29,0x2A,
0x31,0x32,0x33,0x34,0x35,0x36,0x37,0x38,0x39,0x3A,
0x41,0x42,0x43,0x44,0x45,0x46,0x47,0x48,0x49,0x4A,
0x51,0x52,0x53,0x54,0x55,0x56,0x57
};

unsigned char Gu8ReceTable[57]; //从下位机接收到的表格数据的数组

//把一些针对某个特定事件的全局变量放在一个结构体内，可以让全局变量的分类更加清晰
struct StructBigBufferUsart      //控制读取大数组的通讯过程的结构体
{
unsigned char u8Status; //通讯过程的状态 0 为初始状态 1 为通讯成功 2 为通讯失败
unsigned char u8ReSendCnt; //重发计数器
unsigned char u8Step;    //通讯过程的步骤
unsigned char u8Start;   //通讯过程的启动
unsigned long u32NeedSendSize; //一共需要发送的全部数据量
unsigned long u32AlreadySendSize; //实际已经发送的数据量
unsigned long u32CurrentAddr; //当前批次需要发送的起始地址
unsigned long u32CurrentSize; //当前批次从起始地址开始发送的数据量
unsigned char u8QueueSendTrig; //队列驱动函数的发送的启动
unsigned char u8QueueSendBuffer[30]; //队列驱动函数的发送指令的数组
unsigned char u8QueueStatus; //队列驱动函数的通讯状态 0 为初始状态 1 为通讯成功 2 为通讯失败

```

```

};

unsigned char Gu8QueueReceUpdate=0; //1 代表“队列发送数据后，收到了新的数据”

struct StructBigBufferUsart  GtBigBufferUsart;//此结构体变量专门用来控制读取大数组的通讯事件

volatile unsigned char vGu8BigBufferUsartTimerFlag=0; //过程控制的超时定时器
volatile unsigned int vGu16BigBufferUsartTimerCnt=0;

volatile unsigned char vGu8QueueSendTimerFlag=0; //队列发送的超时定时器
volatile unsigned int vGu16QueueSendTimerCnt=0;

volatile unsigned char vGu8BeepTimerFlag=0;
volatile unsigned int vGu16BeepTimerCnt=0;

volatile unsigned char vGu8KeySec=0;

unsigned char Gu8SendByteFinish=0; //发送一个字节完成的标志

unsigned char Gu8ReceBuffer[REC_BUFFER_SIZE]; //常规控制类的小内存
unsigned char *pGu8ReceBuffer; //用来切换接收内存的“中转指针”

unsigned long Gu32ReceCntMax=REC_BUFFER_SIZE; //最大缓存
unsigned long Gu32ReceCnt=0; //接收缓存数组的下标
unsigned char Gu8ReceStep=0; //接收中断函数里的步骤变量
unsigned char Gu8ReceFeedDog=1; //“喂狗”的操作变量。
unsigned char Gu8ReceType=0; //接收的数据类型
unsigned char Gu8Rece_Xor=0; //接收的异或
unsigned long Gu32ReceDataLength=0; //接收的数据长度
unsigned char Gu8FinishFlag=0; //是否已接收完成一串数据的标志
unsigned long *pu32Data; //用于数据转换的指针
volatile unsigned char vGu8ReceTimeOutFlag=0;//通讯过程中字节之间的超时定时器的开关
volatile unsigned int vGu16ReceTimeOutCnt=0; //通讯过程中字节之间的超时定时器，“喂狗”的对象

void main()
{
    SystemInitial();
    Delay(10000);
    PeripheralInitial();
    while(1)
    {
        UsartTask(); //串口收发的任务函数
        KeyTask();
    }
}

```

```

}

void KeyTask(void)    //按键任务函数，放在主函数内
{
    if(0==vGu8KeySec)
    {
        return; //按键的触发序号是 0 意味着无按键触发，直接退出当前函数，不执行此函数下面的代码
    }

    switch(vGu8KeySec) //根据不同的按键触发序号执行对应的代码
    {
        case 1:        //1 号按键。K1 的独立按键
            //GtBigBufferUsart.u8Start 在开机初始化函数里必须初始化为 0!这一步很关键!
            if(0==GtBigBufferUsart.u8Start) //只有在还没有启动的情况下，才能启动
            {
                GtBigBufferUsart.u8Status=0; //通讯过程的状态 0 为初始状态
                GtBigBufferUsart.u8Step=0;    //通讯过程的步骤 0 为从当前开始的步骤
                GtBigBufferUsart.u8Start=1;    //通讯过程的启动
            }
            vGu8KeySec=0; //响应按键服务处理程序后，按键编号必须清零，避免一致触发
            break;
    }
}

/* 注释一：
* 每一个通讯事件都对应的一个独立的“通讯过程的控制函数”，一个系统中有多少个通讯事件，就存在
* 多少个“通讯过程的控制函数”。该函数负责某个通讯事件从开始到结束的整个过程。比如本节项目，
* 在通讯过程中，如果发现接收到的数据错误，则继续启动重发的机制。当发现接收到的累加字节数等于
* 预期想要接收的数量时，则结束这个通讯的事件。
*/

void BigBufferUsart(void) //读取下位机大数组的“通讯过程的控制函数”
{
    static const unsigned char SCu8ReSendCntMax=3; //重发的次数
    static unsigned long *pSu32Data; //用于数据与数组转换的指针

    switch(GtBigBufferUsart.u8Step) //过程控制，我首选 switch 语句!
    {
        case 0:
            if(1==GtBigBufferUsart.u8Start) //通讯过程的启动
            {
                //根据实际项目需要，在此第 0 步骤里可以添加一些初始化相关的数据
                GtBigBufferUsart.u8ReSendCnt=0; //重发计数器清零
            }
        }
    }
}

```



```

        GtBigBufferUsart.u8Step=1;    //切换到下一步
    }
    break;

//-----先发送“读取下位机的数组容量的大小的指令”-----
//-----EB 01 00 00 00 07 ED-----
    case 1:
        GtBigBufferUsart.u8QueueSendBuffer[0]=0xeb; //数据头
        GtBigBufferUsart.u8QueueSendBuffer[1]=0x01; //数据类型 读取数组容量大小
        pSu32Data=(unsigned long *)&GtBigBufferUsart.u8QueueSendBuffer[2];
        *pSu32Data=7; //数据长度 本条指令的数据总长是 7 个字节
//异或算法的函数
        GtBigBufferUsart.u8QueueSendBuffer[6]=CalculateXor(GtBigBufferUsart.u8QueueSendBuffer,
                                                            6); //最后一个字节不纳入计算

//队列驱动函数的状态 0 为初始状态 1 为通讯成功 2 为通讯失败
        GtBigBufferUsart.u8QueueStatus=0; //队列驱动函数的通讯状态
        GtBigBufferUsart.u8QueueSendTrig=1; //队列驱动函数的发送的启动

        vGu8BigBufferUsartTimerFlag=0;
        vGu16BigBufferUsartTimerCnt=2000;
        vGu8BigBufferUsartTimerFlag=1; //过程控制的超时定时器的启动

        GtBigBufferUsart.u8Step=2;    //切换到下一步
    break;

case 2: //发送之后，等待下位机返回的数据的状态
    if(1==GtBigBufferUsart.u8QueueStatus) //当前批次的接收到的数据成功
    {
        GtBigBufferUsart.u8ReSendCnt=0; //重发计数器清零
        GtBigBufferUsart.u32AlreadySendSize=0; //实际已经发送的数据量清零
        GtBigBufferUsart.u32CurrentAddr=0; //当前批次需要发送的起始地址
        GtBigBufferUsart.u32CurrentSize=10; //从当前批次起始地址开始发送的数据量
        GtBigBufferUsart.u8Step=3;    //切换到下一步
    }
    else if(2==GtBigBufferUsart.u8QueueStatus) //当前批次的接收到的数据失败
    {
        GtBigBufferUsart.u8ReSendCnt++;
        if(GtBigBufferUsart.u8ReSendCnt>=SCu8ReSendCntMax) //大于最大的重发次数
        {
            GtBigBufferUsart.u8Step=0;
            GtBigBufferUsart.u8Start=0; //结束当前的过程通讯
            GtBigBufferUsart.u8Status=2; //对外宣布“通讯失败”
        }
    }
}

```

```

        vGu8BeepTimerFlag=0;
        vGu16BeepTimerCnt=30;    //让蜂鸣器“短鸣”一声
        vGu8BeepTimerFlag=1;
    }
    else
    {
        GtBigBufferUsart.u8Step=1;    //返回上一步，重发当前段的数据
    }
}
else if(0==vGu16BigBufferUsartTimerCnt) //当前批次在等待接收返回数据时，超时
{
    GtBigBufferUsart.u8ReSendCnt++;
    if(GtBigBufferUsart.u8ReSendCnt>=SCu8ReSendCntMax) //大于最大的重发次数
    {
        GtBigBufferUsart.u8Step=0;
        GtBigBufferUsart.u8Start=0; //结束当前的过程通讯
        GtBigBufferUsart.u8Status=2; //对外宣布“通讯失败”

        vGu8BeepTimerFlag=0;
        vGu16BeepTimerCnt=30;    //让蜂鸣器“短鸣”一声
        vGu8BeepTimerFlag=1;
    }
    else
    {
        GtBigBufferUsart.u8Step=1;    //返回上一步，重发当前段的数据
    }
}
break;

//-----接着发送“读取下位机的大数组的分段数据的指令”-----
//-----EB 02 00 00 00 0F RR RR RR RR YY YY YY ZZ -----
case 3:
    GtBigBufferUsart.u8QueueSendBuffer[0]=0xeb; //数据头
    GtBigBufferUsart.u8QueueSendBuffer[1]=0x02; //数据类型 读取分段数据
    pSu32Data=(unsigned long *)&GtBigBufferUsart.u8QueueSendBuffer[2];
    *pSu32Data=15; //数据长度 本条指令的数据总长是 15 个字节

    pSu32Data=(unsigned long *)&GtBigBufferUsart.u8QueueSendBuffer[2+4];
    *pSu32Data=GtBigBufferUsart.u32CurrentAddr; //当前批次需要发送的起始地址

    pSu32Data=(unsigned long *)&GtBigBufferUsart.u8QueueSendBuffer[2+4+4];
    *pSu32Data=GtBigBufferUsart.u32CurrentSize; //从当前批次起始地址发送的数据量

//异或算法的函数

```

```
GtBigBufferUsart.u8QueueSendBuffer[14]=CalculateXor(GtBigBufferUsart.u8QueueSendBuffer,
14); //最后一个字节不纳入计算
```

```
//队列驱动函数的状态 0 为初始状态 1 为通讯成功 2 为通讯失败
GtBigBufferUsart.u8QueueStatus=0; //队列驱动函数的通讯状态
GtBigBufferUsart.u8QueueSendTrig=1; //队列驱动函数的发送的启动
```

```
vGu8BigBufferUsartTimerFlag=0;
vGu16BigBufferUsartTimerCnt=2000;
vGu8BigBufferUsartTimerFlag=1; //过程控制的超时定时器的启动
```

```
GtBigBufferUsart.u8Step=4; //切换到下一步
break;
```

```
case 4: //发送之后，等待下位机返回的数据的状态
```

```
if(1==GtBigBufferUsart.u8QueueStatus) //当前批次的接收到的数据成功
{
```

```
    //更新累加当前实际已经发送的字节数
```

```
    GtBigBufferUsart.u32AlreadySendSize=GtBigBufferUsart.u32AlreadySendSize+
    GtBigBufferUsart.u32CurrentSize;
```

```
    //更新下一步起始的发送地址
```

```
    GtBigBufferUsart.u32CurrentAddr=GtBigBufferUsart.u32CurrentAddr+
    GtBigBufferUsart.u32CurrentSize;
```

```
    //更新下一步从起始地址开始发送的字节数
```

```
    if((GtBigBufferUsart.u32CurrentAddr+GtBigBufferUsart.u32CurrentSize)>
    GtBigBufferUsart.u32NeedSendSize) //最后一段数据的临界点的判断
    {
```

```
        GtBigBufferUsart.u32CurrentSize=GtBigBufferUsart.u32NeedSendSize-
        GtBigBufferUsart.u32CurrentAddr;
```

```
    }
```

```
    else
```

```
    {
```

```
        GtBigBufferUsart.u32CurrentSize=10;
```

```
    }
```

```
    //判断是否已经把整个大数组的 57 个字节都已经接收完毕。如果已经接收完毕，则
    //结束当前通信；如果还没结束，则继续请求下位机发送下一段新数据。
```

```
    if(GtBigBufferUsart.u32AlreadySendSize>=GtBigBufferUsart.u32NeedSendSize)
    {
```

```
        GtBigBufferUsart.u8Step=0;
```

```
        GtBigBufferUsart.u8Start=0; //结束当前的过程通讯
```

```

        if(1==CmpTwoBufferIsSame(Cu8TestTable,    //如果接收的数据与存储的相等
                                Gu8ReceTable,
                                57))
        {
            vGu8BeepTimerFlag=0;
            vGu16BeepTimerCnt=1000;    //让蜂鸣器“长鸣”一声
            vGu8BeepTimerFlag=1;
            GtBigBufferUsart.u8Status=1; //对外宣布“通讯成功”
        }
        else
        {
            vGu8BeepTimerFlag=0;
            vGu16BeepTimerCnt=30;    //让蜂鸣器“短鸣”一声
            vGu8BeepTimerFlag=1;
            GtBigBufferUsart.u8Status=2; //对外宣布“通讯失败”
        }

    }
    else
    {
        GtBigBufferUsart.u8ReSendCnt=0; //重发计数器清零
        GtBigBufferUsart.u8Step=3;    //返回上一步，继续发下一段的新数据
    }

}
else if(2==GtBigBufferUsart.u8QueueStatus) //当前批次的接收到的数据失败
{
    GtBigBufferUsart.u8ReSendCnt++;
    if(GtBigBufferUsart.u8ReSendCnt>=SCu8ReSendCntMax) //大于最大的重发次数
    {
        GtBigBufferUsart.u8Step=0;
        GtBigBufferUsart.u8Start=0; //结束当前的过程通讯
        GtBigBufferUsart.u8Status=2; //对外宣布“通讯失败”

        vGu8BeepTimerFlag=0;
        vGu16BeepTimerCnt=30;    //让蜂鸣器“短鸣”一声
        vGu8BeepTimerFlag=1;
    }
    else
    {
        GtBigBufferUsart.u8Step=3;    //返回上一步，重发当前段的数据
    }
}
else if(0==vGu16BigBufferUsartTimerCnt) //当前批次在等待接收返回数据时，超时

```

```
{
    GtBigBufferUsart.u8ReSendCnt++;
    if(GtBigBufferUsart.u8ReSendCnt>=SCu8ReSendCntMax) //大于最大的重发次数
    {
        GtBigBufferUsart.u8Step=0;
        GtBigBufferUsart.u8Start=0; //结束当前的过程通讯
        GtBigBufferUsart.u8Status=2; //对外宣布“通讯失败”

        vGu8BeepTimerFlag=0;
        vGu16BeepTimerCnt=30; //让蜂鸣器“短鸣”一声
        vGu8BeepTimerFlag=1;
    }
    else
    {
        GtBigBufferUsart.u8Step=3; //返回上一步，重发当前段的数据
    }
}
break;
}
```

/\* 注释二：

\* 整个项目中只有一个“发送的队列驱动函数”，负责“通讯管道的占用”的分配，负责数据的具体发

\* 送。当同时存在很多“待发送”的请求指令时，此函数会根据“if ,else if...”的优先级，像队列一

\* 样安排各指令发送的先后顺序，确保各指令不会发生冲突。

\*/

```
void QueueSend(void) //发送的队列驱动函数
{
    static unsigned char Su8Step=0;

    switch(Su8Step)
    {
        case 0: //分派即将要发送的任务
            if(1==GtBigBufferUsart.u8QueueSendTrig)
            {
                GtBigBufferUsart.u8QueueSendTrig=0; //及时清零。驱动层，不管结果，只发一次。

                Gu8QueueReceUpdate=0; //接收应答数据的状态恢复初始值

                //发送带指令的数据
                UsartSendMessage((const unsigned char*)&GtBigBufferUsart.u8QueueSendBuffer[0],
                                30);
```

```

        vGu8QueueSendTimerFlag=0;
        vGu16QueueSendTimerCnt=2000;
        vGu8QueueSendTimerFlag=1; //队列发送的超时定时器
        Su8Step=1;
    }
//    else if(...) //当有其它发送的指令时，可以在此处继续添加判断，越往下优先级越低
//    else if(...) //当有其它发送的指令时，可以在此处继续添加判断，越往下优先级越低

    break;

case 1: //发送之后，等待下位机的应答。驱动层，只管有没有应答，不管应答对不对。
    if(1==Gu8QueueReceUpdate) //如果“接收数据后的处理函数”接收到应答数据
    {
        Su8Step=0; //返回上一步继续处理其它“待发送的指令”
    }

    if(0==vGu16QueueSendTimerCnt) //发送指令之后，等待应答超时
    {
        Su8Step=0; //返回上一步继续处理其它“待发送的指令”
    }

    break;
}
}

/* 注释三：
* 整个项目中只有一个“接收数据后的处理函数”，负责即时处理当前接收到的数据。
*/

void ReceDataHandle(void) //接收数据后的处理函数
{
    static unsigned long *pSu32Data; //数据转换的指针
    static unsigned long i;
    static unsigned char Su8Rece_Xor=0; //计算的“异或”

    static unsigned long Su32CurrentAddr; //读取的起始地址
    static unsigned long Su32CurrentSize; //读取的发送的数据量

    if(1==Gu8ReceFeedDog) //每被“喂一次狗”，就及时更新一次“超时检测的定时器”的初值
    {
        Gu8ReceFeedDog=0;
    }
}

```

```

    vGu8ReceTimeOutFlag=0;
    vGu16ReceTimeOutCnt=RECE_TIME_OUT;//更新一次“超时检测的定时器”的初值
    vGu8ReceTimeOutFlag=1;
}
else if (Gu8ReceStep>0&&0==vGu16ReceTimeOutCnt) //超时，并且步骤不在接头暗号的步骤
{
    Gu8ReceStep=0; //串口接收数据的中断函数及时切换回接头暗号的步骤
}

if (1==Gu8FinishFlag) //1 代表已经接收完毕一串新的数据，需要马上去处理
{
    switch (Gu8ReceType) //接收到的数据类型
    {
        case 0x01: //读取下位机的数组容量的大小
            Gu8QueueReceUpdate=1; //告诉“队列驱动函数”收到了新的应答数据

            Gu8Rece_Xor=Gu8ReceBuffer[Gu32ReceDataLength-1]; //提取接收到的“异或”
            Su8Rece_Xor=CalculateXor (Gu8ReceBuffer, Gu32ReceDataLength-1); //计算“异或”

            if (Gu32ReceDataLength>=11&& //接收到的数据长度必须大于或者等于 11 个字节
                Su8Rece_Xor==Gu8Rece_Xor) //验证“异或”，“计算的”与“接收的”是否一致
            {
                pSu32Data=(unsigned long *)&Gu8ReceBuffer[6]; //数据转换。
                GtBigBufferUsart.u32NeedSendSize=*pSu32Data; //提取将要接收数组的大小
                GtBigBufferUsart.u8QueueStatus=1; //告诉“过程控制函数”，当前通讯成功
            }
            else
            {
                GtBigBufferUsart.u8QueueStatus=2; //告诉“过程控制函数”，当前通讯失败
            }

            break;

        case 0x02: //读取下位机的分段数据
            Gu8QueueReceUpdate=1; //告诉“队列驱动函数”收到了新的应答数据

            Gu8Rece_Xor=Gu8ReceBuffer[Gu32ReceDataLength-1]; //提取接收到的“异或”
            Su8Rece_Xor=CalculateXor (Gu8ReceBuffer, Gu32ReceDataLength-1); //计算“异或”

            pSu32Data=(unsigned long *)&Gu8ReceBuffer[6]; //数据转换。
            Su32CurrentAddr=*pSu32Data; //读取的起始地址

            pSu32Data=(unsigned long *)&Gu8ReceBuffer[6+4]; //数据转换。
            Su32CurrentSize=*pSu32Data; //读取的发送的数据量
    }
}

```

```

        if(Gu32ReceDataLength>=11&& //接收到的数据长度必须大于或者等于 11 个字节
        Su8Rece_Xor==Gu8Rece_Xor&& //验证“异或”，“计算的”与“接收的”是否一致
        Su32CurrentAddr==GtBigBufferUsart.u32CurrentAddr&& //验证“地址”，相当于验证“动态密钥”
        Su32CurrentSize==GtBigBufferUsart.u32CurrentSize) //验证“地址”，相当于验证“动态密钥”
        {
            for(i=0;i<Su32CurrentSize;i++)
            {
                //及时把接收到的数据存储到 Gu8ReceTable 数组
                Gu8ReceTable[Su32CurrentAddr+i]=Gu8ReceBuffer[6+4+4+i];
            }

            GtBigBufferUsart.u8QueueStatus=1; //告诉“过程控制函数”，当前通讯成功
        }
        else
        {
            GtBigBufferUsart.u8QueueStatus=2; //告诉“过程控制函数”，当前通讯失败
        }

        break;
    }

    Gu8FinishFlag=0; //上面处理完数据再清零标志，为下一次接收新的数据做准备
}

void UsartTask(void)    //串口收发的任务函数，放在主函数内
{
    BigBufferUsart(); //读取下位机大数组的“通讯过程的控制函数”
    QueueSend();      //发送的队列驱动函数
    ReceDataHandle(); //接收数据后的处理函数
}

void usart(void) interrupt 4 //串口接发的中断函数，中断号为 4
{
    if(1==RI) //接收完一个字节后引起的中断
    {
        RI = 0; //及时清零，避免一直无缘无故的进入中断。

        if(0==Gu8FinishFlag) //1 代表已经完成接收了一串新数据，并且禁止接收其它新的数据
        {
            Gu8ReceFeedDog=1; //每接收到一个字节的数据，此标志就置 1 及时更新定时器的值。
            switch(Gu8ReceStep)
            {

```



```

case 0:    // “前部分的”数据头。接头暗号的步骤。
    Gu8ReceBuffer[0]=SBUF; //直接读取刚接收完的一个字节的数据。
    if(0xeb==Gu8ReceBuffer[0]) //等于数据头 0xeb，接头暗号吻合。
    {
        Gu32ReceCnt=1; //接收缓存的下标
        Gu8ReceStep=1; //切换到下一个步骤，接收其它有效的数据
    }
    break;

case 1:    // “前部分的”数据类型和长度
    Gu8ReceBuffer[Gu32ReceCnt]=SBUF; //直接读取刚接收完的一个字节的数据。
    Gu32ReceCnt++; //每接收一个字节，数组下标都自加 1，为接收下一个数据做准备
    if(Gu32ReceCnt>=6) //前 6 个数据。接收完了“数据类型”和“数据长度”。
    {
        Gu8ReceType=Gu8ReceBuffer[1]; //提取“数据类型”
        //以下的转换，在第 62 节讲解过的指针法
        pu32Data=(unsigned long *)&Gu8ReceBuffer[2]; //数据转换
        Gu32ReceDataLength=*pu32Data; //提取“数据长度”
        if(Gu32ReceCnt>=Gu32ReceDataLength) //靠“数据长度”来判断是否完成
        {
            Gu8FinishFlag=1; //接收完成标志“置 1”，通知主函数处理。
            Gu8ReceStep=0; //及时切换回接头暗号的步骤
        }
        else //如果还没结束，继续切换到下一个步骤，接收“有效数据”
        {
            //本节只用到一个接收数组，把指针关联到 Gu8ReceBuffer 本身的数组
            pGu8ReceBuffer=(unsigned char *)&Gu8ReceBuffer[6];
            Gu32ReceCntMax=REC_BUFFER_SIZE; //最大缓存

            Gu8ReceStep=2; //切换到下一个步骤
        }
    }
    break;

case 2:    // “后部分的”数据
    pGu8ReceBuffer[Gu32ReceCnt-6]=SBUF; //这里的指针就是各种不同内存的化身!!!
    Gu32ReceCnt++; //每接收一个字节，数组下标都自加 1，为接收下一个数据做准备

    //靠“数据长度”来判断是否完成。也不允许超过数组的最大缓存的长度
    if(Gu32ReceCnt>=Gu32ReceDataLength || Gu32ReceCnt>=Gu32ReceCntMax)
    {
        Gu8FinishFlag=1; //接收完成标志“置 1”，通知主函数处理。
        Gu8ReceStep=0; //及时切换回接头暗号的步骤
    }
    break;

```

```

    }
}
else //发送数据引起的中断
{
    TI = 0; //及时清除发送中断的标志，避免一直无缘无故的进入中断。
    Gu8SendByteFinish=1; //从 0 变成 1 通知主函数已经发送完一个字节的數據了。
}
}

void UsartSendByteData(unsigned char u8SendData) //发送一个字节的底层驱动函数
{
    static unsigned int Su16TimeOutDelay; //超时处理的延时时器

    Gu8SendByteFinish=0; //在发送以字节之前，必须先把此全局变量的标志清零。
    SBUF =u8SendData; //依靠寄存器 SBUF 作为载体发送一个字节的數據
    Su16TimeOutDelay=0xffff; //超时处理的延时时器装载一个相对合理的计时初始值
    while(Su16TimeOutDelay>0) //超时处理
    {
        if(1==Gu8SendByteFinish)
        {
            break; //如果 Gu8SendByteFinish 为 1，则发送一个字节完成，退出当前循环等待。
        }
        Su16TimeOutDelay--; //超时时器不断递减
    }

    //Delay(); //在实际应用中，当连续发送一堆数据时如果发现丢失数据，可以尝试在此增加延时
}

//发送带协议的函数
void UsartSendMessage(const unsigned char *pCu8SendMessage,unsigned long u32SendMaxSize)
{
    static unsigned long i;
    static unsigned long *pSu32;
    static unsigned long u32SendSize;

    pSu32=(const unsigned long *)&pCu8SendMessage[2];
    u32SendSize=*pSu32; //从带协议的数组中提取整包数组的有效发送长度

    if(u32SendSize>u32SendMaxSize) //如果“有效发送长度”大于“最大限制的长度”，数据异常
    {
        return; //数据异常，直接退出当前函数，预防数组越界
    }
}

```

```

        for(i=0;i<u32SendSize;i++) //u32SendSize 为发送的数据长度
        {
            UsartSendByteData(pCu8SendMessage[i]); //基于“发送单字节的最小接口函数”来实现的
        }
    }

unsigned char CalculateXor(const unsigned char *pCu8Buffer, //此处加 const 代表数组“只读”
                          unsigned long u32BufferSize) //参与计算的数组的大小
{
    unsigned long i;
    unsigned char Su8Rece_Xor;
    Su8Rece_Xor=pCu8Buffer[0]; //提取数据串第“i=0”个数据作为异或的原始数据
    for(i=1;i<u32BufferSize;i++) //注意，这里是从第“i=1”个数据开始
    {
        Su8Rece_Xor=Su8Rece_Xor^pCu8Buffer[i]; //计算“异或”
    }
    return Su8Rece_Xor; //返回运算后的异或的计算结果
}

//比较两个数组的是否相等。返回 1 代表相等，返回 0 代表不相等
unsigned char CmpTwoBufferIsSame(const unsigned char *pCu8Buffer_1,
                                const unsigned char *pCu8Buffer_2,
                                unsigned long u32BufferSize) //参与对比的数组的大小
{
    unsigned long i;
    for(i=0;i<u32BufferSize;i++)
    {
        if(pCu8Buffer_1[i]!=pCu8Buffer_2[i])
        {
            return 0; //只要有一个不相等，则返回 0 并且退出当前函数
        }
    }
    return 1; //相等
}

void KeyScan(void) //此函数放在定时中断里每 1ms 扫描一次
{
    static unsigned char Su8KeyLock1; //1 号按键的自锁
    static unsigned int Su16KeyCnt1; //1 号按键的计时器

    //1 号按键
    if(0!=KEY_INPUT1)//IO 是高电平，说明按键没有被按下，这时要及时清零一些标志位
    {

```

```

        Su8KeyLock1=0; //按键解锁
        Su16KeyCnt1=0; //按键去抖动延时计数器清零，此行非常巧妙，是全场的亮点。
    }
    else if(0==Su8KeyLock1)//有按键按下，且是第一次被按下。
    {
        Su16KeyCnt1++; //累加定时中断次数
        if(Su16KeyCnt1>=KEY_FILTER_TIME) //滤波的“稳定时间” KEY_FILTER_TIME，长度是 25ms。
        {
            Su8KeyLock1=1; //按键的自锁, 避免一直触发
            vGu8KeySec=1;    //触发 1 号键
        }
    }
}

void TO_time() interrupt 1
{
    VoiceScan();
    KeyScan();

    if(1==vGu8BigBufferUsartTimerFlag&&vGu16BigBufferUsartTimerCnt>0) //过程控制的超时定时器
    {
        vGu16BigBufferUsartTimerCnt--;
    }

    if(1==vGu8QueueSendTimerFlag&&vGu16QueueSendTimerCnt>0) //队列发送的超时定时器
    {
        vGu16QueueSendTimerCnt--;
    }

    if(1==vGu8ReceTimeOutFlag&&vGu16ReceTimeOutCnt>0) //通讯过程中字节之间的超时定时器
    {
        vGu16ReceTimeOutCnt--;
    }

    TH0=0xfc;
    TL0=0x66;
}

void SystemInitial(void)
{
    unsigned char u8_TMOD_Temp=0;

    //以下是定时器 0 的中断的配置

```

```

TMOD=0x01;
TH0=0xfc;
TL0=0x66;
EA=1;
ET0=1;
TR0=1;

//以下是串口接收中断的配置
//串口的波特率与内置的定时器 1 直接相关，因此配置此定时器 1 就等效于配置波特率。
u8_TMOD_Temp=0x20; //即将把定时器 1 设置为：工作方式 2，初值自动重装的 8 位定时器。
TMOD=TMOD&0x0f; //此寄存器低 4 位是跟定时器 0 相关，高 4 位是跟定时器 1 相关。先清零定时器 1。
TMOD=TMOD|u8_TMOD_Temp; //把高 4 位的定时器 1 填入 0x2，低 4 位的定时器 0 保持不变。
TH1=256-(11059200L/12/32/9600); //波特率为 9600。11059200 代表晶振 11.0592MHz，
TL1=256-(11059200L/12/32/9600); //L 代表 long 的长类型数据。根据芯片手册提供的计算公式。
TR1=1; //开启定时器 1

SM0=0;
SM1=1; //SM0 与 SM1 的设置：选择 10 位异步通讯，波特率根据定时器 1 可变
REN=1; //允许串口接收数据

//为了保证串口中断接收的数据不丢失，必须设置 IP = 0x10，相当于把串口中断设置为最高优先级，
//这个时候，串口中断可以打断任何其他的中断服务函数实现嵌套，
IP =0x10; //把串口中断设置为最高优先级，必须的。

ES=1; //允许串口中断
EA=1; //允许总中断
}

void Delay(unsigned long u32DelayTime)
{
    for(;u32DelayTime>0;u32DelayTime--);
}

void PeripheralInitial(void)
{
    GtBigBufferUsart.u8Start=0; //通讯过程的启动变量必须初始化为 0!这一步很关键!
}

void BeepOpen(void)
{
    P3_4=0;
}

void BeepClose(void)

```

```

{
    P3_4=1;
}

void VoiceScan(void)
{

    static unsigned char Su8Lock=0;

    if(1==vGu8BeepTimerFlag&&vGu16BeepTimerCnt>0)
    {
        if(0==Su8Lock)
        {
            Su8Lock=1;
            BeepOpen();
        }
        else
        {

            vGu16BeepTimerCnt--;

            if(0==vGu16BeepTimerCnt)
            {
                Su8Lock=0;
                BeepClose();
            }

        }
    }
}
}

```

### 【134.8 例程的下位机程序。】

下位机作为从机应答上位机的指令，程序相对简化了很多。不需要“通讯过程的控制函数”，直接在“接收数据后的处理函数”里启动“发送的队列驱动函数”来发送应答的数据即可。发送应答数据后，也不用等待上位机的应答数据。

```

#include "REG52.H"

#define RECE_TIME_OUT    2000 //通讯过程中字节之间的超时时间 2000ms
#define REC_BUFFER_SIZE  30    //常规控制类数组的长度

void usart(void); //串口接收的中断函数

```

```

void TO_time();    //定时器的中断函数

void QueueSend(void);    //发送的队列驱动函数
void ReceDataHandle(void); //接收数据后的处理函数

void UsartTask(void);    //串口收发的任务函数，放在主函数内

unsigned char CalculateXor(const unsigned char *pCu8Buffer, //异或的算法的函数
                           unsigned long u32BufferSize);

void UsartSendByteData(unsigned char u8SendData); //发送一个字节的底层驱动函数

//发送带协议的函数
void UsartSendMessage(const unsigned char *pCu8SendMessage, unsigned long u32SendMaxSize);

void SystemInitial(void) ;
void Delay(unsigned long u32DelayTime) ;
void PeripheralInitial(void) ;

//下面表格数组的数据与上位机的表格数据一模一样，目的用来让上位机检测接收到的数据是否正确
code unsigned char Cu8TestTable[]=
{
0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0A,
0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17, 0x18, 0x19, 0x1A,
0x21, 0x22, 0x23, 0x24, 0x25, 0x26, 0x27, 0x28, 0x29, 0x2A,
0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37, 0x38, 0x39, 0x3A,
0x41, 0x42, 0x43, 0x44, 0x45, 0x46, 0x47, 0x48, 0x49, 0x4A,
0x51, 0x52, 0x53, 0x54, 0x55, 0x56, 0x57
};

//把一些针对某个特定事件的全局变量放在一个结构体内，可以让全局变量的分类更加清晰
struct StructBigBufferUsart    //应答读取大数组的通讯过程的结构体
{
unsigned char u8QueueSendTrig; //队列驱动函数的发送的启动
unsigned char u8QueueSendBuffer[30]; //队列驱动函数的发送指令的数组
unsigned char u8QueueStatus; //队列驱动函数的通讯状态 0 为初始状态 1 为通讯成功 2 为通讯失败
};

unsigned char Gu8QueueReceUpdate=0; //1 代表“队列发送数据后，收到了新的数据”

struct StructBigBufferUsart GtBigBufferUsart; //此结构体变量专门用来应答读取大数组的通讯事件

volatile unsigned char vGu8QueueSendTimerFlag=0; //队列发送的超时定时器
volatile unsigned int vGu16QueueSendTimerCnt=0;

```

```

unsigned char Gu8SendByteFinish=0; //发送一个字节完成的标志

unsigned char Gu8ReceBuffer[REC_BUFFER_SIZE]; //常规控制类的小内存
unsigned char *pGu8ReceBuffer; //用来切换接收内存的“中转指针”

unsigned long Gu32ReceCntMax=REC_BUFFER_SIZE; //最大缓存
unsigned long Gu32ReceCnt=0; //接收缓存数组的下标
unsigned char Gu8ReceStep=0; //接收中断函数里的步骤变量
unsigned char Gu8ReceFeedDog=1; //“喂狗”的操作变量。
unsigned char Gu8ReceType=0; //接收的数据类型
unsigned char Gu8Rece_Xor=0; //接收的异或
unsigned long Gu32ReceDataLength=0; //接收的数据长度
unsigned char Gu8FinishFlag=0; //是否已接收完成一串数据的标志
unsigned long *pu32Data; //用于数据转换的指针
volatile unsigned char vGu8ReceTimeOutFlag=0; //通讯过程中字节之间的超时定时器的开关
volatile unsigned int vGu16ReceTimeOutCnt=0; //通讯过程中字节之间的超时定时器，“喂狗”的对象

void main()
{
    SystemInitial();
    Delay(10000);
    PeripheralInitial();
    while(1)
    {
        UsartTask(); //串口收发的任务函数
    }
}

/* 注释一：
* 整个项目中只有一个“发送的队列驱动函数”，负责“通讯管道的占用”的分配，负责数据的具体发
* 送。当同时存在很多“待发送”的请求指令时，此函数会根据“if ,else if...”的优先级，像队列一
* 样安排各指令发送的先后顺序，确保各指令不会发生冲突。
*/

void QueueSend(void) //发送的队列驱动函数
{
    static unsigned char Su8Step=0;

    switch(Su8Step)
    {
        case 0: //分派即将要发送的任务
            if(1==GtBigBufferUsart.u8QueueSendTrig)
            {

```



```

        GtBigBufferUsart.u8QueueSendTrig=0; //及时清零。驱动层，不管结果，只发一次。

        Gu8QueueReceUpdate=0; //接收应答数据的状态恢复初始值

        //发送带指令的数据
        UsartSendMessage((const unsigned char *)&GtBigBufferUsart.u8QueueSendBuffer[0],
                        30);
        //注意，这里是从机应答主机的数据，不需要等待返回的数据，因此不需要切换 Su8Step
    }
    // else if(...) //当有其它发送的指令时，可以在此处继续添加判断，越往下优先级越低
    // else if(...) //当有其它发送的指令时，可以在此处继续添加判断，越往下优先级越低

    break;

case 1: //发送之后，等待下位机的应答。驱动层，只管有没有应答，不管应答对不对。
    if(1==Gu8QueueReceUpdate) //如果“接收数据后的处理函数”接收到应答数据
    {
        Su8Step=0; //返回上一步继续处理其它“待发送的指令”
    }

    if(0==vGu16QueueSendTimerCnt) //发送指令之后，等待应答超时
    {
        Su8Step=0; //返回上一步继续处理其它“待发送的指令”
    }

    break;
}
}

/* 注释二：
* 整个项目中只有一个“接收数据后的处理函数”，负责即时处理当前接收到的数据。
*/

void ReceDataHandle(void) //接收数据后的处理函数
{
    static unsigned long *pSu32Data; //数据转换的指针
    static unsigned long i;
    static unsigned char Su8Rece_Xor=0; //计算的“异或”

    static unsigned long Su32CurrentAddr; //读取的起始地址
    static unsigned long Su32CurrentSize; //读取的发送的数据量

    if(1==Gu8ReceFeedDog) //每被“喂一次狗”，就及时更新一次“超时检测的定时器”的初值

```

```

{
    Gu8ReceFeedDog=0;

    vGu8ReceTimeOutFlag=0;
    vGu16ReceTimeOutCnt=RECE_TIME_OUT;//更新一次“超时检测的定时器”的初值
    vGu8ReceTimeOutFlag=1;
}
else if (Gu8ReceStep>0&&0==vGu16ReceTimeOutCnt) //超时，并且步骤不在接头暗号的步骤
{
    Gu8ReceStep=0; //串口接收数据的中断函数及时切换回接头暗号的步骤
}

if(1==Gu8FinishFlag) //1 代表已经接收完毕一串新的数据，需要马上去处理
{
    switch(Gu8ReceType) //接收到的数据类型
    {
        case 0x01: //返回下位机的数组容量的大小

            Gu8Rece_Xor=Gu8ReceBuffer[Gu32ReceDataLength-1]; //提取接收到的“异或”
            Su8Rece_Xor=CalculateXor(Gu8ReceBuffer,Gu32ReceDataLength-1); //计算“异或”

            if(Su8Rece_Xor!=Gu8Rece_Xor) //验证“异或”，如果不相等，退出当前 switch
            {
                break; //退出当前 switch
            }

            GtBigBufferUsart.u8QueueSendBuffer[0]=0xeb; //数据头
            GtBigBufferUsart.u8QueueSendBuffer[1]=0x01; //数据类型 返回数组容量的大小
            pSu32Data=(unsigned long *)&GtBigBufferUsart.u8QueueSendBuffer[2];
            *pSu32Data=11; //数据长度 本条指令的数据总长是 11 个字节

            //提取数组容量的大小
            pSu32Data=(unsigned long *)&GtBigBufferUsart.u8QueueSendBuffer[2+4];
            *pSu32Data=sizeof(Cu8TestTable); //相当于*pSu32Data=57;sizeof 请参考第 69 节

            //异或算法的函数
            GtBigBufferUsart.u8QueueSendBuffer[10]=CalculateXor(GtBigBufferUsart.u8QueueSendBuffer,
                                                                    10); //最后一个字节不纳入计算

            //队列驱动函数的状态 0 为初始状态 1 为通讯成功 2 为通讯失败
            GtBigBufferUsart.u8QueueStatus=0; //队列驱动函数的通讯状态
            GtBigBufferUsart.u8QueueSendTrig=1; //队列驱动函数的发送的启动

            Gu8QueueReceUpdate=1; //告诉“队列驱动函数”此发送指令无需等待上位机的应答

```

```

        break;

case 0x02:    //返回下位机的分段数据

    Gu8Rece_Xor=Gu8ReceBuffer[Gu32ReceDataLength-1]; //提取接收到的“异或”
    Su8Rece_Xor=CalculateXor(Gu8ReceBuffer, Gu32ReceDataLength-1); //计算“异或”

    if(Su8Rece_Xor!=Gu8Rece_Xor) //验证“异或”，如果不相等，退出当前 switch
    {
        break;    //退出当前 switch
    }

    pSu32Data=(unsigned long *)&Gu8ReceBuffer[6]; //数据转换。
    Su32CurrentAddr=*pSu32Data; //读取的起始地址

    pSu32Data=(unsigned long *)&Gu8ReceBuffer[6+4]; //数据转换。
    Su32CurrentSize=*pSu32Data; //读取的发送的数据量

    GtBigBufferUsart.u8QueueSendBuffer[0]=0xeb; //数据头
    GtBigBufferUsart.u8QueueSendBuffer[1]=0x02; //数据类型 返回分段数据

    pSu32Data=(unsigned long *)&GtBigBufferUsart.u8QueueSendBuffer[2];
    *pSu32Data=6+4+4+Su32CurrentSize+1; //数据总长度

    pSu32Data=(unsigned long *)&GtBigBufferUsart.u8QueueSendBuffer[2+4];
    *pSu32Data=Su32CurrentAddr; //返回接收到的起始地址

    pSu32Data=(unsigned long *)&GtBigBufferUsart.u8QueueSendBuffer[2+4+4];
    *pSu32Data=Su32CurrentSize; //返回接收到的当前批次的数据量

    for(i=0;i<Su32CurrentSize;i++)
    {
        //装载即将要发送的分段数据
        GtBigBufferUsart.u8QueueSendBuffer[6+4+4+i]=Cu8TestTable[Su32CurrentAddr+i];
    }

    //异或算法的函数
    GtBigBufferUsart.u8QueueSendBuffer[6+4+4+Su32CurrentSize]=
    CalculateXor(GtBigBufferUsart.u8QueueSendBuffer, 6+4+4+Su32CurrentSize);

    //队列驱动函数的状态 0 为初始状态 1 为通讯成功 2 为通讯失败
    GtBigBufferUsart.u8QueueStatus=0; //队列驱动函数的通讯状态
    GtBigBufferUsart.u8QueueSendTrig=1; //队列驱动函数的发送的启动

```

```

        Gu8QueueReceUpdate=1; //告诉“队列驱动函数”此发送指令无需等待上位机的应答
        break;
    }

    Gu8FinishFlag=0; //上面处理完数据再清零标志，为下一次接收新的数据做准备
}
}

void UsartTask(void)    //串口收发的任务函数，放在主函数内
{
    QueueSend();        //发送的队列驱动函数
    ReceDataHandle();   //接收数据后的处理函数
}

void usart(void) interrupt 4 //串口接发的中断函数，中断号为 4
{
    if(1==RI) //接收完一个字节后引起的中断
    {
        RI = 0; //及时清零，避免一直无缘无故的进入中断。

        if(0==Gu8FinishFlag) //1 代表已经完成接收了一串新数据，并且禁止接收其它新的数据
        {
            Gu8ReceFeedDog=1; //每接收到一个字节的数据，此标志就置 1 及时更新定时器的值。
            switch(Gu8ReceStep)
            {
                case 0: // “前部分的”数据头。接头暗号的步骤。
                    Gu8ReceBuffer[0]=SBUF; //直接读取刚接收完的一个字节的数据。
                    if(0xeb==Gu8ReceBuffer[0]) //等于数据头 0xeb，接头暗号吻合。
                    {
                        Gu32ReceCnt=1; //接收缓存的下标
                        Gu8ReceStep=1; //切换到下一个步骤，接收其它有效的数据
                    }
                    break;

                case 1: // “前部分的”数据类型和长度
                    Gu8ReceBuffer[Gu32ReceCnt]=SBUF; //直接读取刚接收完的一个字节的数据。
                    Gu32ReceCnt++; //每接收一个字节，数组下标都自加 1，为接收下一个数据做准备
                    if(Gu32ReceCnt>=6) //前 6 个数据。接收完了“数据类型”和“数据长度”。
                    {
                        Gu8ReceType=Gu8ReceBuffer[1]; //提取“数据类型”
                        //以下的数据转换，在第 62 节讲解过的指针法
                        pu32Data=(unsigned long *)&Gu8ReceBuffer[2]; //数据转换
                        Gu32ReceDataLength=*pu32Data; //提取“数据长度”
                        if(Gu32ReceCnt>=Gu32ReceDataLength) //靠“数据长度”来判断是否完成

```

```

        {
            Gu8FinishFlag=1; //接收完成标志“置1”，通知主函数处理。
            Gu8ReceStep=0;    //及时切换回接头暗号的步骤
        }
    else    //如果还没结束，继续切换到下一个步骤，接收“有效数据”
    {
        //本节只用到一个接收数组，把指针关联到 Gu8ReceBuffer 本身的数组
        pGu8ReceBuffer=(unsigned char *)&Gu8ReceBuffer[6];
        Gu32ReceCntMax=REC_BUFFER_SIZE;    //最大缓存

        Gu8ReceStep=2;    //切换到下一个步骤
    }
}
break;
case 2:    //“后部分的”数据
    pGu8ReceBuffer[Gu32ReceCnt-6]=SBUF; //这里的指针就是各种不同内存的化身!!!
    Gu32ReceCnt++; //每接收一个字节，数组下标都自加1，为接收下一个数据做准备

    //靠“数据长度”来判断是否完成。也不允许超过数组的最大缓存的长度
    if (Gu32ReceCnt>=Gu32ReceDataLength || Gu32ReceCnt>=Gu32ReceCntMax)
    {
        Gu8FinishFlag=1; //接收完成标志“置1”，通知主函数处理。
        Gu8ReceStep=0;    //及时切换回接头暗号的步骤
    }
    break;
}
}
}
else //发送数据引起的中断
{
    TI = 0; //及时清除发送中断的标志，避免一直无缘无故的进入中断。
    Gu8SendByteFinish=1; //从0变成1通知主函数已经发送完一个字节的數據了。
}
}

void UsartSendByteData(unsigned char u8SendData) //发送一个字节的底层驱动函数
{
    static unsigned int Su16TimeOutDelay; //超时处理的延时时器

    Gu8SendByteFinish=0; //在发送以字节之前，必须先把此全局变量的标志清零。
    SBUF =u8SendData; //依靠寄存器 SBUF 作为载体发送一个字节的數據
    Su16TimeOutDelay=0xffff; //超时处理的延时时器装载一个相对合理的计时初始值
    while (Su16TimeOutDelay>0) //超时处理
    {

```

```

        if(1==Gu8SendByteFinish)
        {
            break; //如果 Gu8SendByteFinish 为 1，则发送一个字节完成，退出当前循环等待。
        }
        Su16TimeOutDelay--; //超时计时器不断递减
    }

    //Delay(); //在实际应用中，当连续发送一堆数据时如果发现丢失数据，可以尝试在此增加延时
}

//发送带协议的函数
void UsartSendMessage(const unsigned char *pCu8SendMessage,unsigned long u32SendMaxSize)
{
    static unsigned long i;
    static unsigned long *pSu32;
    static unsigned long u32SendSize;

    pSu32=(const unsigned long *)&pCu8SendMessage[2];
    u32SendSize=*pSu32; //从带协议的数组中提取整包数组的有效发送长度

    if(u32SendSize>u32SendMaxSize) //如果“有效发送长度”大于“最大限制的长度”，数据异常
    {
        return; //数据异常，直接退出当前函数，预防数组越界
    }

    for(i=0;i<u32SendSize;i++) //u32SendSize 为发送的数据长度
    {
        UsartSendByteData(pCu8SendMessage[i]); //基于“发送单字节的最小接口函数”来实现的
    }
}

unsigned char CalculateXor(const unsigned char *pCu8Buffer, //此处加 const 代表数组“只读”
                          unsigned long u32BufferSize) //参与计算的数组的大小
{
    unsigned long i;
    unsigned char Su8Rece_Xor;
    Su8Rece_Xor=pCu8Buffer[0]; //提取数据串第“i=0”个数据作为异或的原始数据
    for(i=1;i<u32BufferSize;i++) //注意，这里是从第“i=1”个数据开始
    {
        Su8Rece_Xor=Su8Rece_Xor^pCu8Buffer[i]; //计算“异或”
    }
    return Su8Rece_Xor; //返回运算后的异或的计算结果
}

```

```

void TO_time() interrupt 1
{

    if(1==vGu8QueueSendTimerFlag&&vGu16QueueSendTimerCnt>0) //队列发送的超时定时器
    {
        vGu16QueueSendTimerCnt--;
    }

    if(1==vGu8ReceTimeOutFlag&&vGu16ReceTimeOutCnt>0) //通讯过程中字节之间的超时定时器
    {
        vGu16ReceTimeOutCnt--;
    }

    TH0=0xfc;
    TL0=0x66;
}

```

```

void SystemInitial(void)
{
    unsigned char u8_TMOD_Temp=0;

    //以下是定时器 0 的中断的配置
    TMOD=0x01;
    TH0=0xfc;
    TL0=0x66;
    EA=1;
    ET0=1;
    TR0=1;

    //以下是串口接收中断的配置
    //串口的波特率与内置的定时器 1 直接相关，因此配置此定时器 1 就等效于配置波特率。
    u8_TMOD_Temp=0x20; //即将把定时器 1 设置为：工作方式 2，初值自动重装的 8 位定时器。
    TMOD=TMOD&0x0f; //此寄存器低 4 位是跟定时器 0 相关，高 4 位是跟定时器 1 相关。先清零定时器 1。
    TMOD=TMOD|u8_TMOD_Temp; //把高 4 位的定时器 1 填入 0x2，低 4 位的定时器 0 保持不变。
    TH1=256-(11059200L/12/32/9600); //波特率为 9600。11059200 代表晶振 11.0592MHz，
    TL1=256-(11059200L/12/32/9600); //L 代表 long 的长类型数据。根据芯片手册提供的计算公式。
    TR1=1; //开启定时器 1

    SM0=0;
    SM1=1; //SM0 与 SM1 的设置：选择 10 位异步通讯，波特率根据定时器 1 可变
    REN=1; //允许串口接收数据

    //为了保证串口中断接收的数据不丢失，必须设置 IP = 0x10，相当于把串口中断设置为最高优先级，

```

//这个时候，串口中断可以打断任何其他的中断服务函数实现嵌套，  
IP =0x10; //把串口中断设置为最高优先级，必须的。

ES=1; //允许串口中断  
EA=1; //允许总中断

}

```
void Delay(unsigned long u32DelayTime)
{
    for(;u32DelayTime>0;u32DelayTime--);
}
```

```
void PeripheralInitial(void)
{
}
}
```