迈航科技

单片机那些事儿

初级篇——LED 小灯

残弈悟恩

2014

官 方 淘 宝 店 铺 ： HTTP://SHOP109195762.TAOBAO.COM

郑重声明

本资料以残弈悟恩开发的 MGMC-V1.0 实验板为硬件平台，以残弈悟恩编写的《深入浅出玩转 51 单片机》为辅助教程，以残弈悟恩录制的《31 天环游单片机》为基础视频。

本资料以个人学习和工作的经验为素材，以单片机初学者、单片机项目开发者为对象。教大家如何走进单片机的世界，如何开发工程项目。限于时间和水平关系，资料中难免有过失之处，望各位高手批评指教，多多拍砖，拍累了，你们休息，我继续上路。

现已连载的方式免费共享于各大电子网站，供单片机新手们参考学习，可以自由下载传阅，但未经残弈悟恩允许，不得用于任何商业目的，若经发现，残弈悟恩将以愚公移山的精神追究到底。

版本：20140416（V1.0）　　制造者：残弈悟恩

让爱充满大地——花 1 秒时间，拯救 1 个人，传递 1 份爱

声明：只是残弈悟恩爱心的喷发，我得不到一分钱，各位不要多想，谢谢！

你知道吗？在非洲北边的某个地区，每一秒都有许许多多的人正在挨饿，每一天至少有一位儿童死于营养不足。你的一次点击就能让某位穷人得到 1.1 杯食物。当然你可以不相信有这样的链接或者是骗点击什么的。事实上，网站确实是帮穷人得 1.1 杯食物的，只要你点进去单机一下中间的黄色按钮，就会出来一系列介绍各种商品的网页（绝对免费的并且不会下载任何软件，也不会有电脑病毒），同时也会有人因为您的一次点击而得到 1.1 杯食物，食物是由商家提拱的，但爱心却是您献出的。如果你觉得残弈悟恩在忽悠大家，你不妨可以在网上查一下是真与假。

看到这本资料的朋友多数都是电子爱好者、单片机初学者，或者干电子这一行的，管你穷学生还是穷工人，只要能上网，只要愿花一秒种就可以了。人生在世，有两件事不能等：一、孝顺；二、行善。无论你是 LED 小灯、普通灯泡也好，还是荧光灯也吧，最重要就是要懂得用自身的光去照耀别人，光的强度并不重要。

点击链接：

http://www.thehungersite.com/clickToGive/home.faces?siteId=1&link=ctg_ths_home_from_ths_thankyou_sitenav

第四章　LED 小灯

无论我们现在学习的单片机，还是以后即将学习的 ARM、FPGA、DSP 等技术，都是从控制最简单的 LED 小灯开始，一点一滴、一步一个脚印的去入门、学习。既然这样，我们就从 LED 小灯开始，来玩单片机。

4.1 各进制的换算

二进制、十进制、十六进制。八进制，不常用。敢不敢相信，大家看到电影、图片或听到歌曲在电脑中都是用二进制存储的，同理，学习单片机时，在 12864 上显示图片也是二进制存储的。因为电脑、单片机这些“大傻瓜”不认识别的，只认识“1 和 0”，所以得学好二进制，咋一听，是不是有些难啊？1、0 小学生都认识，我们“大学生”还怕吗？

十进制数有 0~9，共 10 个，逢十进一；二进制数 0、1 共两个，逢二进一；十六进制数有 0~9、外加 A~F（a~f），总共 16 个数，逢十六进一。二进制书写前需加 0b，十六进制需加 0x。十六进制数是和四为一，就是 4 个二进制组成一个十六进制数，于是它的每一位有 0b0000~0b1111 共计 16 个值。这三个数之间对于关系见表 4-1 所示。

借此为大家推荐一本王玮编著的《感悟设计——电子设计的经验与哲理》一书，我甚是喜欢，他里面写的好多东西，确实值得我们好好学习，虽然有些内容我还没涉及到，但这丝毫不影响我对本书的喜欢。里除了理论、经验，还有他发明的指算（二、十进制之间的转换）。我就引用到这里，希望能和大家共勉。

一只手掌 5 个手指，假设我们规定拇指、食指、中指、无名指、小指分别代表 1、2、4、8、16 这 5 个数（顺序倒过来或搅乱也可以，规定好就行），那么，在 0~31 以内的各个整数都可以通过手指的屈伸来表示了。例如划拳（民间喝酒的一种方法）出的二，就是十进制数 5（1+4）。通常做的“OK 手势”表示的就是 28（4+8+16），如此等等。这么一说大家可能会觉得没意思，但当大家用熟练了，就会觉得很好玩。

表 4-1 部分二进制、十进制、十六进制之间的对应关系

十进制	二进制	十六进制
0	0b0000 0000	0x00
1	0b0000 0001	0x01
2	0b0000 0010	0x02
3	0b0000 0011	0x03
4	0b0000 0100	0x04
5	0b0000 0101	0x05
6	0b0000 0110	0x06
7	0b0000 0111	0x07
8	0b0000 1000	0x08
9	0b0000 1001	0x09
10	0b0000 1010	0x0A
11	0b0000 1011	0x0B
12	0b0000 1100	0x0C
13	0b0000 1101	0x0D
14	0b0000 1110	0x0E
15	0b0000 1111	0x0F

……	……	……
255	0b1111 1111	0xFF

强势 X 入广告。本资料不做特殊说明，请大家记下“两个”等于（个人规定）：（1）高电平=逻辑“1”=VCC=5V；（2）低电平=逻辑“0”=GND=0V 。

4.2 数字电路和 C 语言中的逻辑运算

二进制的逻辑运算，又称其为布尔运算。无论 C 语言中，还是数字电路中，逻辑运算不可缺。在逻辑范畴中，只有“真”和“假”。先来目睹一下 C 语言中的逻辑运算，“0”为“假”，“非 0”为真，不要理解为只有 1 是“真”，2、-43、100 同样也是真。

（1）逻辑运算（是按整体运算），通常叫做逻辑运算符。

&&（and）：逻辑与，只有同为真时结果才为真，近似于乘法。

||（or）：逻辑或，只有同为假时结果才为假，近似于加法。

！（not）：逻辑非，条件为真，结果为假，近似于相反数。

（2）逻辑运算（按每个位来运算），通常叫做位运算符。

&：按位与，变量的每一位都参与（下同），例如：A = 0b0101 1010，B = 0b1010 1010，则 A & B = 0b0000 1010。

|：按位或。则 A | B = 0b1111 1010。

~：按位取反。则~A = 0b1010 0101。

^: 按位异或，异或的意思是，如果运算双方的值不同（即相异），则结果为真，双方值相同则结果为假。这样 A^B = 0b1111 0000。

4.3 C 语言之编程规范

刚开始，我准备不写这章，但考虑到新手或许对 C 语言方面的知识点有些模糊，最后决定还是写一写，就算抛 C 语言这块砖，引 C 语言这块玉吧。我写的这点 C 语言知识，只能算一点点皮毛而已，但有了这些“砖”，大家可以边玩单片机，边补充一下 C 语言知识，要是想深入学习，我强烈推荐大家看看这两本书，它们分别是：

《C Primer Plus 中文版》

《C 语言深度解剖》

C 语言究竟有多难，或者有多简单，这里引用几句《C 语言深度解剖》一书的作者陈正冲的话，望大家能重视 C 语言，将 C 语言学习列为一个长期计划。

-----版权声明：以下内容引用于陈正冲编著的《C 语言深度解剖》前言-----

我很无奈，也很无语。因为我完全在和一群业余者或者是 C 语言爱好者在对话。你们大学的计算机教育根本就是在浪费你们的时间，念了几年大学，连 C 语言的门都没摸着。现在大多数学校计算机系都开了 C、C++、Java、C#等语言，好像什么都学了，但是什么都不会，更可悲的是有些大学居然取消了 C 语言课程，认为其过时了。我个人的观点是“十鸟在林，不如一鸟在手”，真正把 C 语言整明白了再学别的语言也很简单，如果 C 语言都没整明白，别的语言学得再好也是花架子，因为你并不了解底层是怎么回事。当然我也从来不认为一个没学过汇编的人能真正掌握 C 语言的真谛。我个人一直认为，普通人用 C 语言在 3 年之下，一般来说，还没掌握 C 语言；5 年之下，一般来说还没熟悉 C 语言；10 年之下，谈不上精通。

记得我当初参加招聘会时，好多企业招聘要求是精通 C 语言、精通单片机、精通 ARM、精通 FPGA、精通 Verilog HDL 等，同学们为了应付招聘，也写上了精通什么、精通什么，现在想来，企业的这种要求很不实在，那么学生所写的精通也是为了“忽悠”人。结论是，人为了生存，必须得学会各种版本的忽悠啊。

言归正传，开始讲述 C 语言的编程规范，其实这也是一个很广、很泛的话题，我这里要讲述的没有想象中的那么全面、复杂、权威，而只是大概提几点，望能和大家共勉。

4.3.1 程序的排版

4-1：程序块要采用缩进风格编写，缩进的空格数为 4 个。

说明：对于由开发工具自动生成的代码可以有不一致。为了能起到师范作用，我将所有的例程都采用程序块缩进 4 个空格的方式来编写。以后项目中，一个大程序有好多对“{、}”，若不采用这种方式，写完之后，一看肯定特别糟糕，这里就不举例了，以后我们慢慢认识。

4-2：相对独立的程序块之间、变量说明之后必须加空行。

4-3：不允许把多个短语句写在一行中，即一行只写一条语句。

4-4：if、for、do、while、case、default 等语句各自占一行，且 if、for、do、while、等语句的执行语句部分无论多少都要加括号{}。

长江后浪退前浪，残弈悟恩被拍死在沙滩上，大家尽量写得比我示范的还严格吧。

4.3.2 程序的注释

注释是程序可读性和可维护性的基石，如果不能在代码上做到顾名思义，那么就需要在注释上下大功夫。

-----版权声明：以下内容引用于陈正冲编著的《C 语言深度解剖》------

安息吧，路德维希.凡.贝多芬！

有位负责维护的程序员半夜被叫起来，去修复一个出了问题的程序。但是程序的原作者已经离职，没有办法联系上他。这个程序员从未接触过这个程序。在仔细检查所有的说明后，他发现了一条注释，如下：

```
MOV AX 723h ;R . I . P . L . V . B .
```

说明一点：这是汇编程序，并且汇编的注释以“;”开始。

这个维护程序员通宵研究这个程序，还是对注释百思不得其解。虽然最后他还是把程序的问题成功解决了，但是这个神秘的注释让他耿耿于怀。

几个月后，这名程序员在一个会议上遇到了注释的原作者。请教之后，才明白这条注释的意思：安息吧，路德维希.凡.贝多芬（Rest in peace,Ludwig Van Beethoven）。贝多芬于 1827 年逝世，而 1827 的十六进制正是 723。这样的注释确实很牛，这样的注释之人确实是高手，但惟独会让看的人哭笑不得啊！

注释的基本要求，现总结以下几点：

4-1：一般情况下，源程序有效注释量必须在 20%以上。

说明：注释的原则是有助于对程序的阅读理解，在该加的地方都必须加，注释不宜太多也不能太少，注释语言必须准确、易懂、简洁。

4-2：注释的内容要清楚、明了，含义准确，防止注释的二义性。

说明：错误的注释不但无益反而有害。

4-3：边写代码边注释，修改代码同时修改注释，以保证注释与代码的一致性。不再有用的注释要删除。

4-4：对于所有有物理含义的变量、常量，如果其命名起不到注释的作用，那么在声明时都必须加以注释，说明物理含义。变量、常量、宏的注释应放在其上方相邻位置或右方。

示例：

```
/* active statistic task num */
#define MAX_ACT_TASK_NUMBER 1000
#define MAX_ACT_TASK_NUMBER 1000      /* active statistic task num */
```

4-5：一目了然的语句不加注释。

例如：i++;　/* i 加 1，有意思吗？？ */

4-6：全局数据（变量、常量定义等）必须要加注释，并且要详细。包括对其功能、取值范围、那些函数或过程存取它，以及存取时该注意的事项等。

4-7：在代码的功能、意图层次上进行注释，提供有用、额外的信息。

说明：注释的目的是解释代码的目的、功能和采用的方法，提供代码以外的信息，帮助大家理解代码，防止没必要的重复注释。

示例：
```
if (receive_flag)
    /*如果 receive_flag 为真，有意义吗？NO*/
if (receive_flag)
    /*如果 xxx 收到了一个什么信息，则...，这就有了额外的信息*/
```

4-8：对一系列的数字编号给出注释，尤其在编写底层驱动程序的时候（比如管脚编号）。

示范：sbit　SCL = P3^6;　　/* P3.6 为 IIC 总线的时钟管脚 */

4-9：注释格式尽量统一，建议使用“/*……*/”。

4-10：注释应考虑程序易读及外观排版的因素，使用的语言若是中、英兼有的，建议多使用中文，除非能用非常流利准确的英文表达。

说明：注释语言不统一，影响程序易读性和外观排版，出于对维护人员的考虑，建议使用中文。

4.4 C 语言之数据

玩单片机，离不开编写程序，程序离不开数据。例如我们后面要写的程序，无论是简简单单的一个 LED 小灯，还是响不停的蜂鸣器，之后到数码管，再到定时器、计数器，都在与数据打交道，因此我们认识一下数据很有必要的。

4.4.1 变量与常量数据

首先看看什么是常量，因为变量是相对常量来说的。前面写过的程序中，用过的常量太多了，例如：1、10、0b1010 1101、0x3f，这些数据从程序执行开始到程序结束，数据一致没有发生变化，这种数据就叫做常量。相反，随程序执行而变化的数据就是变量了，例如 for 循环中的 i 变量，第一次执行是为：0，之后加加变为：1，再之后变为：5，等等。

既然是变量，那么就得有个范围，否则越界了怎么办。接下来看看 C51 中变量的范围，仅仅是 C51，这与 C 语言在别的编译器中有些区别。其 C51 数据类型如表 4-1 所示。

表 4-1 C51 数据类型的分类和数值范围

数据类型	符　合	范　围
字符型	unsigned char	0 ~ 255
	signed char	-128 ~127
整型	unsigned int	0 ~65535
	signed int	-32768 ~32767
长整型	unsigned long int	0 ~4294967295
	signed long int	-2147483648 ~2147483647
浮点型	float	-3.4×10^{-38} ~3.4×10^{38}
	double float	-3.4×10^{-38} ~3.4×10^{38}（C51 中）

最后总结一句：大家们以后编写程序时，对于变量只用小，不用大。什么意思，能用 char 解决的变量问题，何必用 long int 型呢，既浪费资源，又会使程序跑的比较慢。但不要为了节省资源而越界哦。例如，unsigned char i;for(i = 0;i < 1000;i++)，这样程序

会一直在 for 循环里跑，因为 i 加的累死也超不过 1000 啊，别小看这么个小问题，初学者经常在这点上被整的不知所措。

4.4.2 变量的作用域

C 语言中的每一个变量都有自己的生存周期和作用域，作用域是指可以引用该变量的代码区域，生命周期表示该变量在存储空间存在的时间。根据作用域来划分，C 语言变量可分为两类：全局变量和局部变量。根据生存周期又分为：动态存储和静态存储，这两者还可以细分，稍后见。

1.全局变量

全局变量也称为外部变量，它是在函数外部定义的变量，其作用域为当前源程序文件，即从定义该变量的当前行开始，直到该变量的源程序文件的结束，在这个区间内所有的函数都可以引用该变量。那别的源文件如何引用该变量了，若大家会，那当然很好，若不会，后面章节会给大家一个满意的答案。

大家以后在用全局变量时需要注意几点：

（1）对于局部变量的定义和说明，可以不加区分。而对于外部变量则不然，外部变量的定义和外部变量的说明并不是一回事。外部变量定义必须在所有的函数之外，且只能定义一次。

而外部变量说明出现在要使用该外部变量的各个函数内，在整个程序内，可能出现多次。外部变量在定义时就已分配了内存单元，外部变量定义可作初始赋值，外部变量说明不能再赋初始值，只是表明在函数内要使用某外部变量。大家是否记得用那个关键词来声明吗？答案稍后见。

（2）外部变量可加强函数模块之间的数据联系，但是又使函数要依赖这些变量，因而使得函数的独立性降低。从模块化程序设计的观点来看这是不利的， 因此能不用全局变量的地方就一定不要用哦。

（3）在同一源文件中，允许全局变量和局部变量同名。在局部变量的作用域内，全局变量不起作用。我想说的是，世间这么多字符组合，何必要用一样的来搞糊涂自己了。

2.局部变量

局部变量也称之为内部变量。局部变量是定义在函数内部的变量，其作用域仅限于函数或者复合语句内，离开该函数或复合语句后将无法再引用该变量。注意，这里所说的复合语句指包含在“{、}”内的语句，例如 if(条件 a){ int a = 0;}在该复合语句中变量 a 的作用域为定义 a 的那一行开始到大括号结束。

大家您请注意以下几点：

（1）主函数中定义的变量也只能在主函数中使用，不能在其它函数中使用。同时，主函数中也不能使用其它函数中定义的变量。因为主函数也是一个函数，它与其它函数是平行的关系。

（2）形参变量是属于被调函数的局部变量，实参变量是属于主调函数的局部变量。

（3）允许在不同的函数中使用相同的变量名，它们代表不同的对象，分配不同的单元，互不干扰，也不会发生混淆。虽然允许在不同的函数中使用相同的变量名，但是为了使程序明了易懂，不提倡在不同的函数中使用相同的变量名。

4.4.3 变量的存储类别

上面提到变量按作用的时间（生存周期）又可以分为：静态变量和动态变量。

1. auto 自动变量，默认的存储类别。根据变量的定义位置决定变量的生命周期和作用域，如果定义在函数外，则为全局变量，定义在函数或复合语句内，则为局部变量。C 语言

中如果忽略变量的存储类别，则编译器自动将其存储类型定义为自动变量。自动变量用关键字 auto 作存储类别的声明。关键字 auto 可以省略，auto 不写则隐含定为“自动存储类别”，属于动态存储方式。

2. static 静态变量。用于限制作用域，无论该变量是全局还是局部变量，该变量都存储在数据段上。静态全局变量的作用域仅限于该文件，而静态局部变量的作用域限于定义该变量的复合语句内。静态局部变量可以延长变量的生命周期，其作用域没有改变，而静态全局变量则生命周期没有改变，但其作用域却减小至当前文件内。有时希望函数中的局部变量的值在函数调用结束后不消失而保留原值，这时就应该指定局部变量为“静态局部变量”，用关键字 static 进行声明。

最后对静态局部变量做几点小结，望大家以后多加注意。

（1）静态局部变量属于静态存储类别，在静态存储区内分配存储单元。在程序整个运行期间都不释放。而自动变量（即动态局部变量）属于动态存储类别，占用动态存储空间，函数调用结束后立即释放。

（2）静态局部变量在编译时赋初值，即只赋初值一次。而对自动变量赋初值是在函数调用时进行，每调用一次函数重新赋一次初值，相当于执行一次赋值语句。

（3）如果在定义局部变量时不赋初值的话，则对静态局部变量来说，编译时自动赋初值 0（对数值型变量）或空字符（对字符变量）。而对自动变量来说，如果不赋初值则它的值是一个不确定的值。

强势 X 入一点点广告：变量的初始化。

在 C51（也即 Keil4 编译器）中，无论全局变量还是局部变量，在定义时即使未初始化，编译器也将会自动将其初始化为 0，因此在使用这两种变量时，不用再考虑它的初始化问题。但为了防止在一些别的编译中出现不确定值，或为了规范编程，笔者建议大家，无论是全局还是局部变量，定义之后顺便赋于初值：0，这样或许能在以后的编程路上少遇点麻烦。

3. register 变量。为了提高效率，C 语言允许将局部变量的值放在 CPU 中的寄存器中，这种变量叫“寄存器变量”，用关键字 register 作声明。关于 register 变量，大家需要三点，具体如下：

1)只有局部自动变量和形式参数可以作为寄存器变量；

2)一个计算机系统中的寄存器数目有限，不能定义任意多个寄存器变量；

3)局部静态变量不能定义为寄存器变量。

4. extern 外部变量（全局变量）：该关键字扩展全局变量的作用域，让其它文件中的程序也可以引用该变量，并不会改变改变量的生命周期。

它的作用域为从变量定义处开始，到本程序文件的末尾。如果在定义点之前的函数想引用外部变量，则应该在引用之前用关键字 extern 对该变量作“外部变量声明”。表示该变量是一个已经定义的外部变量。有了此声明，就可以从“声明”处起，合法地使用该外部变量。

可能这么解释有些模糊，那就先铭记，到后面编写程序时，我再给大家详细举例、并深入讲解。

这里还有一个名词，寄存器，稍后我们再来讲述，大家耐心等待吧。

4.4.4 变量的命名规则

首先向大家声明两点：变量的命名好坏当然与程序的好坏没有直接的关系；残弈悟恩在本书中的命名也不是太规范。但残弈还是衷心希望大家尽量将变量、函数名（后面讲述）写的规范一点。

一、命名的分类

在说变量命名之前，不得不提两种法则：匈牙利命名法、驼峰式大小写法。可能有人会说还有帕斯卡命名法，严格的说，帕斯卡命名属于驼峰式大小的子集，因此这里就不做说明了。无论哪种命名，在我看来，任何一个命名应该主要包含两层含义：望文知义、简单却信息量大。

1．驼峰命名法（Camel-Case）。该方法是电脑程序编写时的一套命名规则（惯例）。

程序员们为了自己的代码能更容易的在同行之间交流，所以才取统一的可读性比较好的命名方式。例如：有些程序员喜欢全部小写，有些程序员喜欢用下划线，所以如果要写一个 my name 的变量，一般写法有 myname、my_name、MyName 或者 myName。这样的命名规则不适合所有程序员阅读，而利用驼峰命名法来表示，可以增加程序可读性。

驼峰命名法就是当变量名或函数名是由一个或多个单字连结在一起，而构成的唯一识别字时，第一个单字以小写字母开始，第二个单字的首字母大写，这种方法统称为“小驼峰式命名法”，如：myFirstName；或每一个单字的首字母都采用大写字母，这种称之为“大驼峰式命名法”，例如：MyLastName。

这样的变量名看上去就像骆驼峰一样此起彼伏，故得名。驼峰命名法的命名规则可视为一种惯例，并无绝对与强制，只是为了增加识别和可读性。

2．匈牙利（Hungary）命名法。同样也是一种编程时的命名规范，

匈牙利命名法是一种编程时的命名规范，又称为 HN 命名法。基本原则是：变量名=属性+类型+对象描述，其中每一对象的名称都要求有明确含义，可以取对象名字全称或名字的一部分。命名要基于容易记忆容易理解的原则。保证名字的连贯性是非常重要的。

据说这种命名法是一位叫 Charles Simonyi 的匈牙利程序员发明的，后来他在微软呆了几年，于是这种命名法就通过微软的各种产品和文档资料向世界传播开了。现在，大部分程序员不管自己使用什么软件进行开发，或多或少都使用了这种命名法。这种命名法的出发点是把变量名按：属性+类型+对象描述的顺序组合起来，以使程序员作变量时对变量的类型和其它属性有直观的了解，下面是 HN 变量命名规范。

（1）属性部分

全局变量用 g_开头。如一个全局的长型变量定义为 g_lFailCount。

静态变量用 s_开头。如一个静态的指针变量定义为 s_plPerv_Inst。

成员变量用 m_开头。如一个长型成员变量定义为 m_lCount

（2）类型部分

指针：p；函数：fn；长整型：l；布尔：b；浮点型（有时也指文件）：f；双字：dw；字符串：sz；短整型：n；双精度浮点；计数：c（通常用 cnt）；字符：ch（通常用 c）；整型：i（通常用 n）；字节：by；字：w；无符号：u；位：bt；

（3）对象描述

采用英文单词或其组合，不允许使用拼音。程序中的英文单词一般不要太复杂，用词应当准确。英文词尽量不缩写，特别是非常用专业名词，如果有缩写，在同一系统中对同一单词必须使用相同的表示法，并且注明其意思。

二、命名的补充规则

1. 变量命名使用名词性词组，函数命名使用动词性词组（后面讲述）。

变量含义表示符构成：目标词 + 动词（的过去分词） + [状语] + [目的地]；

例如：DataGotFromSD、DataDeletedFromSD。

2. 所有宏定义、枚举常数、只读变量全用大写字母命名，用下划线分割单词。

例如：const int MIN_LENGTH =10；

4.5 C 语言之条件判断

该节内容或许简单，或者重要。

4.5.1 if 语句

与 if 语句有关的关键字就两个：if 和 else，翻译成中文就是“如果”和“否则”。if 语句有三种格式，分别如下：

一、if 语句的默认形式

```
if（条件表达式）
{
    语句 A;
}
```

其执行过程是，if（如果）条件表达式的值为“真”，则执行语句 A；如果条件表达式的值为“假”，则不执行语句 A。

二、if...else 语句

有些情况下，除了 if 的条件满足以后执行相应的语句以外，还需执行条件不满足情况下的相应语句，这时候就要用 if...else 语句了，它的基本语法形式是：

```
if（条件表达式）
{
    语句 A;
}
else
{
    语句 B;
}
```

三、if...else if 语句

if...else 语句是一个二选一的语句，或者执行 if 条件下的语句，或者执行 else 条件下的语句。还有一种多选一的用法就是 if...else if 语句。它的基本语法格式是：

```
if（条件表达式 1）
{
    语句 A;
}
else if（条件表达式 2）
{
    语句 B;
}
else if（条件表达式 3）
{
    语句 C;
}
……
else
{
```

```
    语句 N;
}
```

它的执行过程是：依次判断条件表达式的值，当出现某个值为“真”时，则执行相应的语句，然后跳出整个 if 的语句，执行“语句 N”后边的程序。如果所有的表达式都为“假”，则执行“语句 N”后，再执行“语句 N”后边的程序。

其实以上写的，不是我要说明的重点，真正要说的是 if 语句究竟该如何写，或者说该注意哪些。

（1）if（i == 100）与 if（100 == i）的区别？建议用后者，具体见实例 11。

（2）bool 变量与“零值”的比较该如何写，下面那种写法好？定义：bool bTestFlag = FALSE; 为何一般初始化为 FALSE 比较好？

A）if（0 == bTestFlag）；　　if（1 == bTestFlag）；

B）if（TRUE == bTestFlag）；　　if（FLASE == bTestFlag）；

C）if（bTestFlag）；　　if（！bTestFlag）；

现来分析一下这三种写法的好坏。

A）写法：bTestFlag 是什么？整型变量？如果不是这个名字遵循了前面的命名规范，恐怕很容易让人误会成整型变量。所以这种写法不怎么好。

B）写法：FLASE 的值大伙都知道，在编译器里被定义为 0；但是 TRUE 的值呢？都是 1 吗？很不幸，不都是 1。Visual C++定义为 1，而它的同胞兄弟 Visual Basic 就把 TRUE 定义为 -1。那很显然，这种写法也不怎么好。

C）写法：关于 if 的执行机理，上面说的很清楚了。那显然，本组的写法很好，既不会引起误会，也不会由于 TRUE 或 FLASE 的不同定义值而出错。记住：以后代码就这样写。

（3）if...else 的匹配不仅要做到心中有数，还要做到胸有成竹。幸好 C 语言规定：else 始终与同一括号内最近的未匹配的 if 语句结合。但大家写的程序，一定要层次分明，让自己、别人一看就知道那个 if 和那个 else 相对应。

（4）先处理正常情况，再处理异常情况。

在编写代码是，要使得正常情况的执行代码清晰，确认那些不常发生的异常情况处理代码不会遮掩正常的执行路径。这样对于代码的可读性和性能都很重要。因为，if 语句总是需要做判断，而正常情况一般比异常情况发生的概率更大（否则就应该把异常和正常颠倒过来了），如果把执行概率更大的代码放到后面，也就意味着 if 语句将进行多次无谓的比较。另外，非常重要的一点是，把正常情况的处理放在 if 后面，而不要放在 else 后面。当然这也符合把正常情况的处理放在前面的要求。

4.5.2 switch...case 语句

在这里单独提出 switch 语句是因为 switch 语句作为分支结构中的一种，使用方式及执行效果上与 if…else 语句完全不同。这种特殊的分支结构作用也是实现程序的条件跳转，不同的是其执行效率要比 if…else 语句快很多，原因在于 switch 语句更具条件实现程序跳转，而不是一次判断每个条件，由于 switch 条件表达式为常量，所以在程序运行时其表达是的值为确定值，因此就会根据确定的值来执行特定条件，而无需再去判断其他情况。由于这种特殊的结构，提倡大家们在自己的程序中尽量采用 switch 而避免过多使用 if…else 结构。switch...case 的格式如下：

```
switch（表达式）
{
    case 常量表达式 1：执行语句 A;break;
    case 常量表达式 2：执行语句 B;break;
```

```
    ……                    ……
    case 常量表达式 n：执行语句 N;break;
    default：执行语句 N+1;
}
```

在用 switch...case 语句时需要注意以下几点。

1）break 一定不能少，否则麻烦重重（除非有意使多个分支重叠）；

2）一定要加 default 分支，不要理解为画蛇添足，即使真的不需要，也应该保留，不要让别人误解为你是傻瓜。

3）case 后面只能是整型或字符型的常量或常量表达式。像 0.1、3/2 等都是不行，大家当然可以上机亲自调试一下。

4）case 语句排列顺序有关吗？若语句比较少，可以不予考虑。若语句较多时，就不得不考虑这个问题了。一般遵循以下三条原则。

①按字母或数字顺序排列各条 case 语句。例如 A、B...Z，1、2..55 等，好处大家慢慢体会。

②把正常情况放在前面，而把异常情况放在后面。

③按执行频率排列 case 语句。即执行越频繁的越往前放，越不频繁执行的越往后放。

4.6 C 语言之循环

C 语言中的循环分为三种形式，分别是：while 循环、do…while 循环和 for 循环。这三种循环在功能上存在细微的差别，但共同的特点是实现一个循环体，可以使程序反复执行一段代码。

4.6.1 while 循环

while 循环：执行循环之前，先判断条件的真假，条件为真，则执行循环体内的语句，为假则不执行循环体内的语句，直接结束该循环。大家需要注意的是什么数为真，什么数为假，不要以为只有 1 为真哦。

```
while(条件表达式)
{
    语句;
}
```

While 我们在后面的程序中要大量的用到，大伙必须掌握哈。

4.6.2 do...while 循环

do...while 循环：先执行一次循环体，再判断条件真假，为真则继续执行循环体内的语句，为假则循环结束。

```
do
{
    语句;
}
while(条件表达式);
```

大家注意区别 while，do...while 若条件表达式为假，会至少执行一次循环体，而 while 若条件为假就连一次都不执行。

4.6.3 for 循环

for 循环：先求解表达式 1，再判断表达式 2 的真假，若为真，则执行 for 循环的内部语句，再执行表达式 3，第一次循环结束(若为假，则整个循环结束，执行 for 循环之后的语句)。第二次循环开始时不再求解表达式 1，直接判断条件表达式，再执行循环体内的语句。之后再执行表达式 3，这样依次循环。

```
for(表达式 1;表达式 2;表达式 3)
{
    语句;
}
```

单片机中，只需注意两点，高手不要咬文嚼字或者拍砖，^_^。

a) while(1)等价于 for(;;)。

b) for 循环的三点说明。

1.建议 for 语句的循环控制变量的取值采用“半开半闭区间”写法。原因在于这种写法比“闭区间”直观，如表 4-2 所示。

表 4-2 for 循环区间写法区别

半开半闭的写法	闭区间写法

for(i = 0; i < 10; i++)
{
语句;
}
	for(i = 0; i <= 9; i++)
	{
	语句;
	}

2.在多重循环中，将最长的循环放在最内层，最短的循环放在最外层，以减少 CPU 跨切循环层的次数，如表 4-3 所示。

表 4-3 for 循环层写法区别

长循环在最内层（效率高）	长循环在最外层（效率低）

for(i = 0; i < 10; i++)
{
for(j = 0; j < 100; j++)
{
语句;
}
}
	for(j = 0; j < 100; j++)
	{
	for(i = 0; i < 10; i++)
	{
	语句;
	}
	}

3.不能在 for 循环体内修改循环变量，防止循环失控。

```
for(iVal = 0; iVal < 10; iVal++)
{
    …
    iVal = 6;  //千万不可，可能会违背自己的意愿
}
```

4.7 话说 LED 小灯

4.7.1 原理说明

LED(Light Emitting Diode)，发光二极管，简称 LED，是一种能将电能转换为可见光的固态半导体器件。LED 的核心是一个半导体的晶片，晶片的一端附在一个支架上，连接电

源的正极使整个晶片被环氧树脂封装起来,另一端是负极。半导体晶片由两部分组成，一部分是 P 型半导体，在它里面空穴占主导地位，另一端是 N 型半导体，在这边主要是电子。但这两种半导体连接起来的时候，它们之间就形成了一个“P-N 结”。当电流通过导线作用于这个晶片的时候，电子就会被推向 P 区，在 P 区里电子跟空穴复合，然后就会以光子的形式发出能量，这就是 LED 发光的原理。至于颜色，大家可以自行查阅资料，概况说，电子与空穴复合时释放出的能量决定了光的波长，波长决定了发光的颜色。

LED 品种繁多，如何识别正负，常见的有直插式和贴片式两种，如何区分 LED 的正负极是很多初学者经常问到的问题，常见的方法有观察法和万用表测量法。

1.观察法

直插式 LED：直插式的 LED 如果是全新的，可以通过引脚长短来判别发光二极管的正负极，引脚长的为正极，短的为负极。还有就是拿手上，一抹就知道（发光二极管的环氧树脂封装上有个缺口的是负极）。

贴片 LED：俯视，一边带彩色线的是负极，另一边是正极。

2.万用表测量法

这里讲述数字式（不是指针式的）。将万用板打到二极管测试档，两表笔接触二极管的两个脚，若二极管发光，说明红表笔接的是正极、黑表笔接的是负极。若不亮，情况刚好相反。当然东西都必须是好的。

4.7.2 硬件分析

对于 LED 来说，总共两个引脚，在电路设计上没有难度，这里以 MGMC-V1.0 实验板上的原理图来做讲解。通常情况下，LED 内部需要通过一定的电流且存在一定的压差（也即压降）才能使得其发光。通常使用的 LED 的工作电流为 3~20mA 左右，但二极管本身的内阻又比较小，所以不能直接将两端接电源和 GND，而需要加一个限流电阻（阻值如何计算，请看后面分解），限制通过 LED 的电流不要太大。LED 原理图如图 4-1 所示。

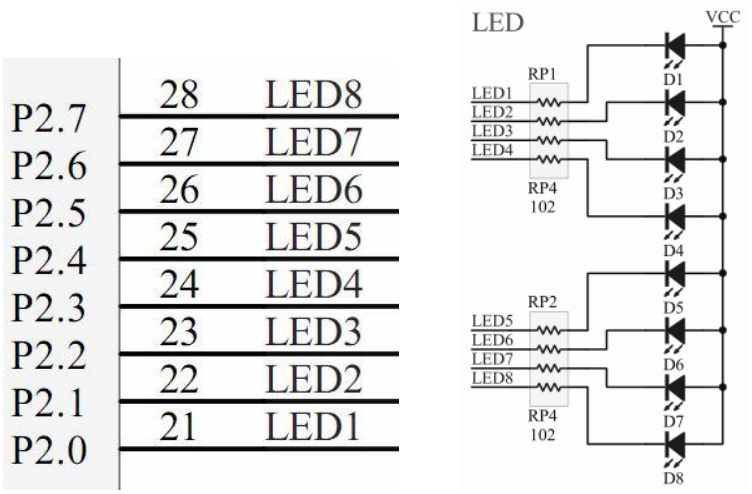

图 4-1 LED 原理图

该方式的 LED 驱动电路是将正极接在 VCC（高电平）上，负极再串联一个 1K（102=10×10²）的限流电阻，再接到单片机的 I/O 口上，这样，只需给 LED1~LED8 所对应的 I/O 口上低电平，就可以点亮 LED；还有一种接法是将 LED 的正极接单片机的 I/O 口，在通过一限流电阻，将负极接地，鉴别一下两种原因，这种方式不常用。在理论上，只要单片机输出高电平，就可以点亮 LED，但事实上，由于单片机的驱动电流（100~200uA）比较小，无法驱动 LED（对于某些高级单片机来说，因为端口可配置为强推挽输出，所以是完全可以驱动 LED 小灯，这里不赘）。还有不选择此种电路的原因是，单片机上电之后 I/O 口默认电平为高电

平，这样，从工程的角度考虑，单片机上电以后，LED 就会工作，这并不是我们想要的结果，所以一般不建议这么设计电路，当然具体设计依情况而定。

经上述分析，LED 两端只要有一合适的压差和电流，就会点亮 LED。那么单片机又是如何控制的呢。举个例子，要有水流，必须有水压差，同样，要有电流，就得有电压差，结合上图，LED 的正极已经接了高电平，那如果 LED0~LED7 所对应的单片机端口也为高电平，这样，LED 的左右电平都是高电平（5V），则就没有压差，所以就没有电流，则 LED 就不会亮，相反若单片机端口为低电平，从而 LED 两端则会有压差（左低、右高），这样 LED 灯就会被点亮（有合适的电压、电流），接着看看如何计算该限流电阻，一般贴片红色 LED 的压降是 1.82~1.88V，那么电阻两端的电压就为：5 - 1.85（中间值）= 3.15V，为了 LED 有合适的亮度和长寿命，一般让其工作电流为：3mA，有欧姆定律可知，限流电阻：R = 3.15V / 3mA = 1.05KΩ，所以用了 1K 的限流电阻，No 问题吧。

实践是检验真理的唯一标准！

4.8 例说 LED 小灯

接下来，我们就从点亮一个小小的 LED 灯开始，玩转单片机，玩转 C 语言，玩转寄存器，玩转所有改玩转的，有没有？（夸张了，别介意哈）

实验 1 点亮希望之灯

通过以上学习，大家已经扫除了 LED 原理和硬件障碍，现在来看看如何借助单片机用软件来控制 LED，由原理图可知，8 个 LED 接在单片机的 P2 口上，这里只需控制 P2 口的电平高低就可以实现控制 LED 的亮灭。为了不把新手们扼杀在萌芽阶段，先不讲解单片机的原理，所以只需依葫芦画瓢，先熟悉开发流程、点亮 LED，点亮一个 LED 灯的效果如图 2 所示，这样会让你有种成就感、喜悦感，可以增加大家的学习信心，接下来我们就开始实验吧！

- **实验目的：**懂得点亮开发板上 D1 号灯
- **实验器材：**电脑、麦光开发板、Keil4 编译软件、ISP 烧录软件
- **实验原理：**编写程序，使得单片机 P2^0 端口为低电平
- **实验步骤：**

1.使用 Keil4 编译软件，新建工程，编写程序，并生成 hex 文件。

```
1.  #include <reg52.h>
2.  sbit LED1 = P2^0;
3.  void main(void)
4.  {
5.      LED1 = 0;
6.  }
```

程序总共 6 行，1、2 行后面再做说明，剩下的就是一个主函数了，在这个主函数体(大括号)中就包含了一句动作代码“LED1 = 0;”它代表的意思就是将 LED 所在的单片机端口电平拉低，进而点亮 LED 灯。是不是特简单？

2.使用 ISP 烧录，将 hex 文件烧录到开发板中。

3.观察实验现象，如图 4-2 所示。

图 4-2 点亮一个 LED 的效果图

实验总结：

其实质只用了一行代码就点亮了一个 LED，是比较简单，不过这只是万里长城的第一步，后面的路还很漫长，且行且坚持。

接下来的实验，步骤、过程大都类似，这里就不赘。

实验 2 一闪一闪亮晶晶

上例已经点亮了一个 LED，那如何让 LED 闪烁起来呢？答案见此实例，同时再详细探讨一下其工作的奥秘。先来实例源代码。

```
1.  #include <reg52.h>
2.  #define uChar8 unsigned char
3.  #define uInt16 unsigned int
4.  sbit LED1 = P2^0;
5.  /* ************************************************ */
6.  // 函数名称：DelayMS()
7.  // 函数功能：毫秒延时
8.  // 入口参数：延时毫秒数（ValMS）
9.  // 出口参数：无
10. /* ************************************************ */
11. void DelayMS(uInt16 ValMS)
12. {
13.     uInt16 uiVal,ujVal;
14.     for(uiVal = 0; uiVal < ValMS; uiVal++)
15.         for(ujVal = 0; ujVal < 113; ujVal++);
16. }
17. void main(void)
18. {
19.     while(1)
20.     {
21.         LED1 = 0;
22.         DelayMS(1000);
23.         LED1 = 1;
24.         DelayMS(1000);
```

```
25.     }
26. }
```

从此实例开始，残弈悟恩将带领大家一行、一句的研读代码，并要弄清单片机内部是如何执行的，硬件上又是如何体现的。

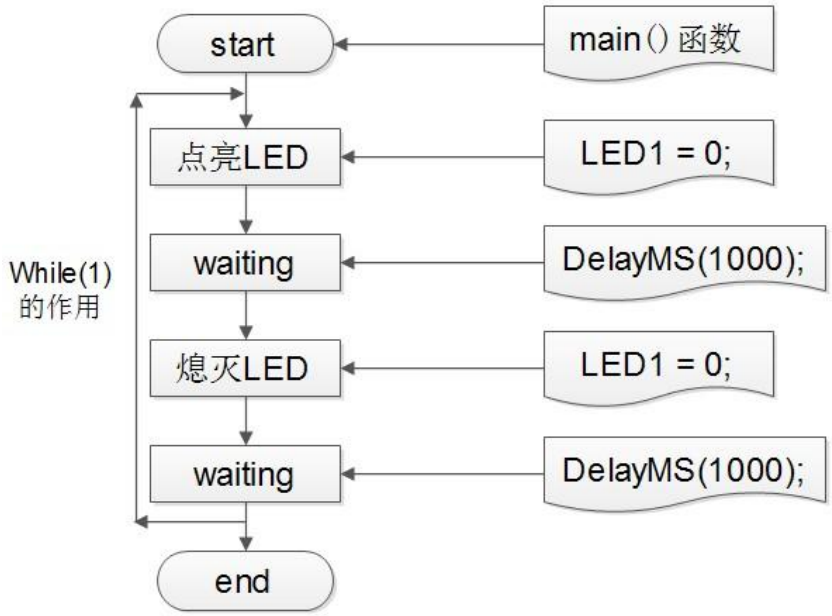

图 4-3 LED 亮灭流程图

相信，有了这张山寨版（以后所有的流程图，残弈悟恩都不会拘于形式，而是会尽量想办法用最简单、形象的方式来呈现给大家）的流程图，具体如图 4-3 所示，则 LED 亮灭的过程应该很清楚了吧，现在估计大家模糊的是代码具体如何实现，单片机内如何工作，为何两个 for 循环就会起到延时的作用，不急，让我慢慢道来。

第 1 行#include <reg52.h>，是为了包含单片机的头文件啊。否则下文用到的 P2 编译器怎么知道“他”来自哪个星球啊！大家学 C 语言时，开头也会写上#include<stdio.h>吧，例如写字，总的有一支笔吧，若大家将手指头咬破，用血来写，那残弈悟恩还能说啥呢？在代码中引用头文件，其实际意义就是将这个头文件中的全部内容放在引用头文件的位置处，避免每次编写同类程序都要将头文件中的语句重复编写。

在代码中加入头文件有两种书写方法，分别是#include <reg52.h>与#include "reg52.h",那这两种形式有何区别，当使用“<xx.h>”包含头文件时，编译器先进入到软件安装文件夹处开始搜索这个头文件，也就是 keil\C51\INC 这个文件夹下，如果这个文件夹下没有引用的头文件，则编译器会报错。当使用“"xx.h"”包含头文件时，编译器先进入当前工程所在的文件夹开始搜索该头文件，如果当前工程所在文件夹下没有该头文件，编译器将继续回到软件安装文件夹处搜索这个头文件，若还是找不到，怎编译器会报错。因而一般将该头文件写成#include <reg52.h>的形式，而以后进行模块化编程时，写成“"xx.h"”的形式，例如自己编写的头文件“LED.h”，则可以写成#include“LED.h”，这个具体到后面模块化编程时，将详细讲解。那么这里包含头文件，主要是为了引用单片机的 P2 口，其实单片机中并没有什么 P0~P3 口，只是为了便于操作，给单片机起了四个别名 P0~P3 口，为了深入了解，可以将鼠标放到 keil4 中的#include<reg52.h>处，单击鼠标右键并选择“Open document<reg52.h>”打开该头文件，具体操作如图 4-4 所示。

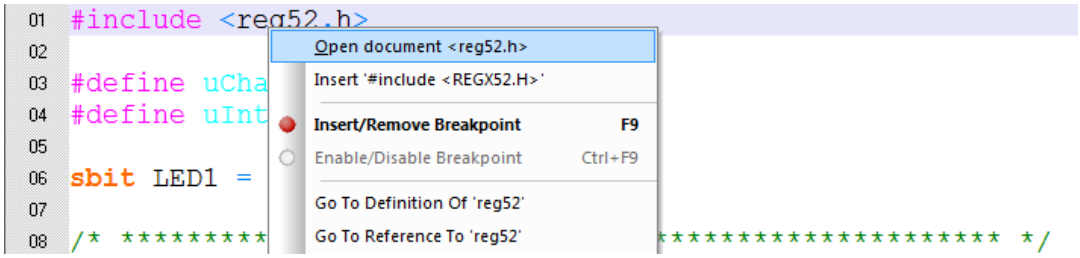

图 4-4 打开 reg52.h 头文件

其内容如下，鉴于篇幅原因，这里只留了一小部分做讲解用，具体内容大家自己可以打开该头文件去参阅。

```
1.  REG52.H
2.  #ifndef __REG52_H__
3.  #define __REG52_H__
4.  /*  BYTE Registers  */
5.  sfr P0      = 0x80;
6.  sfr P1      = 0x90;
7.  sfr P2      = 0xA0;
8.  sfr P3      = 0xB0;
9.  /*  BIT Registers  */
10. /*  IE  */
11. sbit EA     = IE^7;
12. sbit ET2    = IE^5;       //8052 only
13. sbit ES     = IE^4;
14. sbit ET1    = IE^3;
15. sbit EX1    = IE^2;
16. sbit ET0    = IE^1;
17. sbit EX0    = IE^0;
```

从上面代码中可以看到，该头文件定义了 52 系列单片机内部所有的功能寄存器，用到了两个关键字 sfr 和 sbit，如第 7 行的 sfr P2 = 0xA0。意思是把单片机内部地址 0xA0 处的这个寄存器重新起名 P2，P2 口有 8 位（0xA0~0xA7），大家就理解为 8 个房间好了。但这 8 位（0xA0~0xA7）与 P2 毫无关系，当操作 P2 口时，实质是在操作（0xA0~0xA7）这 8 位寄存器，形象点，大家叫张三，笔者给张三一个苹果，事实上苹果给到了张三这个人，而不是张三这两个字。这样，如果写一句：P2 = 0x00，就等价于将从地址 0xA0 开始的 8 个寄存器全部清零，之后单片机内部又通过数据总线（不管他是一根钢管还是一根毛线，反正是一种线）将这 8 位寄存器与 I/O 口相连，最后操作这些寄存器就可达到控制 I/O 的目的。

为了让大家再明白点（笔者培训时发现，同学们在这里很是不理解啊），笔者举个形象的例子。P0~P4 就相当于 0 层楼（地下室）、一、二、三层，一层楼房又分 8 个房间，例如房号有：001、103、205、307 等，这些房号就对应到单片机中就是 0xA0、0xA6 等，或者是所取的别名 P2.0、P2.6 等，接着房间里面可以住男的（高电平），也可以住女的（低电平），同理，这些寄存器中可以存“1”或者“0”。这样 32 个房间（4 层*8 个房间）刚好就对应 32 个寄存器，最后将这 32 个寄存器用某种特殊的线连接到 32 个 I/O 口上，继而实现了通过控制寄存器达到控制 I/O 口的目的。

接着再看看 sbit，例如第 11 行 sbit EA = IE^7。就是将 IE 寄存器（他也是对应一个地址，单片机也不认 PSW）的最高位重新命名为 EA，以后要开总中断时，就直接可以写 EA =

1；意思是将 EA 所对应的的最高位置 1（写高电平）。再不用写地址什么 0xXX 了。这样头文件就讲完了，不知大伙们什么感觉，迷迷糊糊还是清清楚楚？若还不懂，请来麦光论坛发帖吧。

第 2~3 行 C 语言中常用的宏定义，我们国家的全名是：中华人民共和国，可是书写时，一般写：中国，为何？长了书写麻烦、不习惯，所以简写呗。同样在编写程序时，写 unsigned char 明显比写 uChar8 麻烦，所以给 unsigned char 来了一种简写的方法 uChar8，当程序运行时遇到 uChar8 时，就会用 unsigned char 替换掉，这样就方便了编写程序。可能有人问，那为何不写个 u（h），这样写是很简单，但是在 C 语言中，无论定义一个变量，还是写一个函数，一定要做到顾名思义，一个 u（h）能达到吗？

第 4 行，sbit 我们前面已经说过了，也就是给 P2 的最低位起个别名 LED1,说白了就是为了简化编程，人就是这么聪明，没办法。这里的“^”理解为“的”好了，不懂就先记下。

第 5~10 行，若有人说没必要，笔者保持沉默，建议大家还是从一开始养成一个良好的编程习惯，等到以后编写复杂程序时，将会起到事半功倍的效果。

这里笔者说明一点，以后的程序，若以前写过的子函数，笔者可能会全部删掉，若没写过的笔者将保留，这样做主要是为了既规范编程，又节省篇幅，但随书、或实验板附带的源代码中将继续保留。

第 11~16 行，一个延时子函数。子函数具体如何编写，如何声明，如何调用，大家先可以去读读笔记 11 的夯实基础部分关于函数的介绍部分。这个函数名称为：DelayMS()，里面有个形式参数 ValMS，就是你想延时的毫秒数，范围由前面的 uInt16 决定，那么就是 $2^{16}-1$（也即 0~65535 毫秒）。如果写个 1000，意味着是 1000 毫秒也即 1 秒，接着说说内部是如何执行的，怎么做到延时的。

函数内部定义了两个局部变量 uiVal,ujVal，用于 for 循环，第一个 for 循环后面应有一对“{}”，这里没写。分析可知，每调用一次 DelayMS 函数，则执行 ValMS*121 次，这里的 121 是实验测得大概数据，那么程序运行到这里，单片机在干什么，其实单片机只是在无谓地做着重复运动“++”，以这样方式来“浪费”单片机的运行时间，从而达到延时的作用。关于延时先说这么多，以后慢慢道来，在此新手们、初学者，一定心中要铭记，刚开始、大学期间或者简单的工程中 DelayMS()函数可以随便用，但在以后真正做工程时，这个函数是一定要避开，具体以后慢慢说。

第 17~26，主函数。应该不陌生吧，关于主函数，百度一搜，一大堆。我们主要说一下 19 行。它是一个 while()循环，具体细节参阅笔记 9 的夯实基础部分，这里 while 的条件是 1（为真），这样就会进入 while 并执行里面的语句，里面总共四句 21、22、23、24，按顺序执行完再去判断 while 的真假，此时还是为真，程序就会继续跑里面的 4 句，完了再判断，还是真，再执行，就这样程序就会一直在 while 中运行，从而有了想要的大循环或者死循环，只要大家不断电，单片机没问题，他就会子子孙孙无穷尽也的跑下去，还有一种替换写法 for(;;)，道理一样，自行分析。

再来细说 LED1 = 0。前面已经说过 LED1 是为第一个 LED 起的别名，那我们倒着一步一步推理，LED1 = 0 等价于 P2^0 = 0，而 P2^0 的意思是 P2 口的最低位，那就等价给 P2 口的最低位赋值为“0”，再走，单片机中 P2 的最低位又对应的是一个地址（0xA0），那就是给地址 0xA0 赋值为“0”，也即给该地址对应的寄存器赋值为“0”，前面说过，该寄存器通过某种线和 I/O 相连，从而单片机 P2.0 口就是低电平，若赋值“1”，则 P2.0 口就是高电平，这样就可以通过操作单片机内部的寄存器值来间接的操作单片机的 I/O 口，最后起到了控制 LED 的作用。

好吧，这样代码就讲完了，最后编译生成 HEX 文件，下载到单片机中，慢慢欣赏吧，恭喜大家，已经是控制 LED 的高手了。

实验 3 跑马的汉子—LED 跑马灯(傻瓜版)

如果你累了、倦了，骑着一匹骏马驰骋于茫茫大草原，那是多么惬意的一件事，前一秒你在这里，下一秒不见了，因为马一直在跑。那如何让 8 个 LED 也像马一样跑起来呢？8 个 LED 模拟跑马的效果，说难也难，说简单也简单，先上源码，之后详细分析。

先看看傻瓜式的跑马灯代码，以便与后面程序对比。

```
1.  #include <reg52.h>
2.  #define uInt16 unsigned int
3.  sbit LED1 = P2^0;
4.  sbit LED2 = P2^1;
5.  sbit LED3 = P2^2;
6.  sbit LED4 = P2^3;
7.  sbit LED5 = P2^4;
8.  sbit LED6 = P2^5;
9.  sbit LED7 = P2^6;
10. sbit LED8 = P2^7;
11. /* ******************************************************* */
12. // 函数名称：DelayMS()
13. // 函数功能：毫秒延时
14. // 入口参数：延时毫秒数（ValMS）
15. // 出口参数：无
16. /* ******************************************************* */
17. void DelayMS(uInt16 ValMS)
18. {
19.     uInt16 uiVal,ujVal;
20.     for(uiVal = 0; uiVal < ValMS; uiVal++)
21.         for(ujVal = 0; ujVal < 113; ujVal++);
22. }
23. void main(void)
24. {
25.     while(1)
26.     {
27.         LED1 = 0;DelayMS(100);
28.         LED1 = 1; LED2 = 0; DelayMS(100);
29.         LED2 = 1; LED3 = 0; DelayMS(100);
30.         LED3 = 1; LED4 = 0; DelayMS(100);
31.         LED4 = 1; LED5 = 0; DelayMS(100);
32.         LED5 = 1; LED6 = 0; DelayMS(100);
33.         LED6 = 1; LED7 = 0; DelayMS(100);
34.         LED7 = 1; LED8 = 0; DelayMS(100);
35.         LED8 = 1;
36.     }
37. }
```

这个程序乍一看很长，是不是很难，不，一点都不，执行过程就是先点亮第一个 LED，

延时一会（这里延时的长度就看马是否吃饱了，饱了就跑地快（延时短），饿了就跑地慢（延时长）），再熄灭，之后点亮第二个，过会再灭，之后的之后...依次循环，1→2…7→8,看跑马的汉子，就是很威武、很雄壮，一直跑，跑不死…

这样的程序也太傻瓜了吧，能不能用简单的方法编写跑马灯了，答案当然是肯定的，接下来看看“高级”写法。

实验 4 跑马的汉子—LED 跑马灯(高级版)

```
1.  #include <reg52.h>
2.  #include <intrins.h>
3.  #define uChar8 unsigned char
4.  #define uInt16 unsigned int
5.  void DelayMS(uInt16 ValMS)
6.  {    /* 同上 */   }
7.  void main(void)
8.  {
9.      uChar8 uTempVal;
10.     uTempVal = 0xfe;
11.     while(1)
12.     {
13.         P2 = uTempVal;
14.         uTempVal = _crol_(uTempVal,1);
15.         DelayMS(100);
16.     }
17. }
```

这里很巧妙的用了 KeilC51 自带的函数库，里面有个_crol_（）函数，利用这个函数就可以大大简化程序,代码由原先的 40 多行变成了现在的 20 多行,接下来详细说说这个函数，它是包含在“intrins.h”头文件中，所以需要写一句#include<intrins.h>包含该头文件。_crol_（）函数的功能是循环左移，什么是循环左移，什么是左移，什么又是右移、或者循环右移，结合图示来说明，如图 4-5、4-6、4-7、4-8 所示。

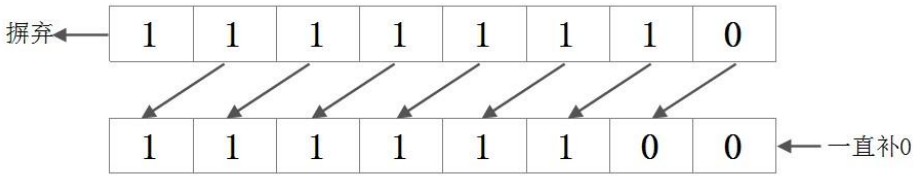

图 4-5 左移示意图

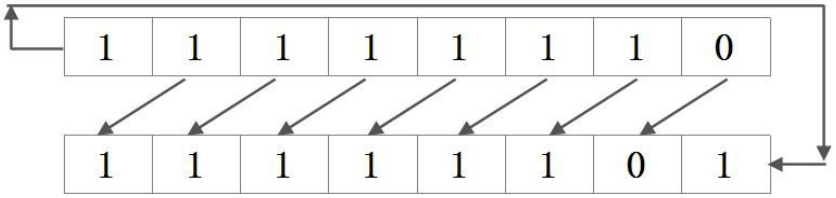

图 4-6 循环左移示意图

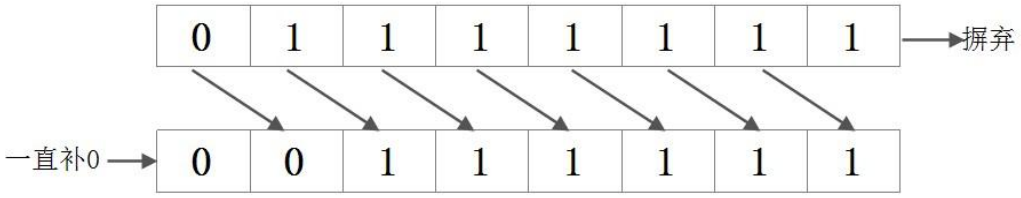

图 4-7 右移示意图

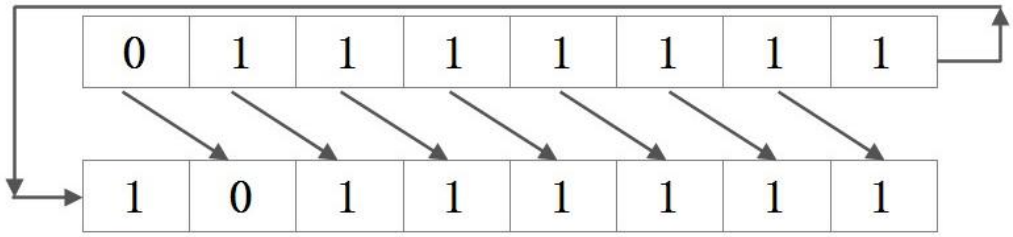

图 4-8 循环右移示意图

看着这些图，笔者相信，C 语言中的“<<”、“>>”都应该能熟练的掌握了吧，还有上面的循环左右移动应该也不难了吧。

程序分析如下，分析前先看个山寨图，如图 4-9 所示，这样更能帮助大家形象的理解此函数的执行过程。

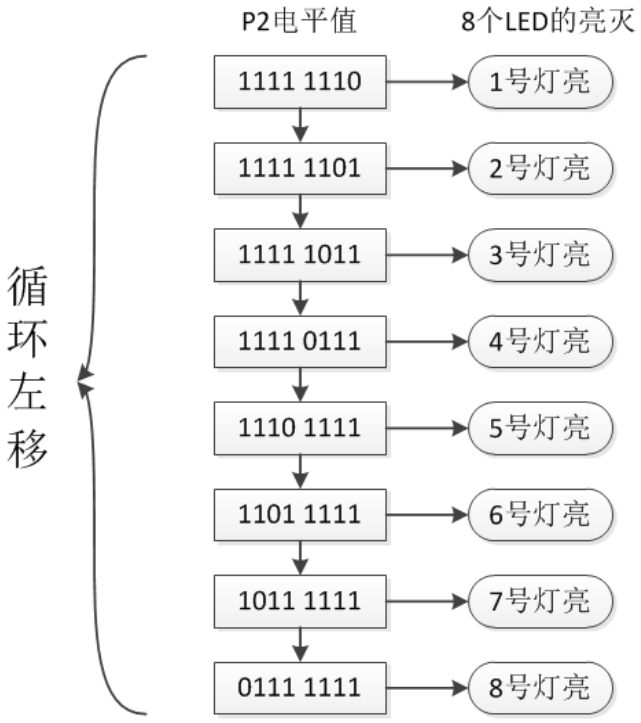

图 4-9 跑马灯示意图

第 1~8，笔者就再不多说了，否则大家都嫌烦。第 9、10 行就是一个变量定义和简单的赋值语句，程序每当执行完_crol_（）函数时能实时的将每一位的值赋给 P2。该程序的核心是 14 行，_crol_（）函数有两个形式参数，第一个是要操作的变量，第二个是一次移位的个数，这里 uTempVal 就是要操作的变量，之后的 1 表示每执行一次该函数，将 uTempVal 变量循环左移一位，uTempVal 的初值是 0xfe(0b1111 1110)，所以上电之后首先 P2.0 对应的 LED 点亮，之后执行一次移位操作，移位后的数据则为：0xfd(0b1111 1101)，直到最后 0x7f(0b0111 1111),这样就会依次点亮这 8 个 LED，从而形成了跑马灯的效果。注意，这里

定义一个局部变量，是方便讲解，当然也可以将程序改写为：P2 = 0xfe，P2 = _crol_(P2, 1)。通过此程序我们可以发现，硬件相同，实验现象相同，可程序千差万别，这就有了高手与菜鸟的区别。因此大家要虚心，不能满足于现象，要多实践，多编程，慢慢积累属于自己经验。

实例 5 美女长发飘飘流—LED 流水灯

流水灯顾名思义就是让 LED 如同流水一般，第一个 LED 先亮，过会第二个亮，但第一还是亮的（区分跑马灯，跑马灯中的第一个这时已经熄灭了），以此类推，亮 3 个、4…8 个，最后全部熄灭，再周而复始的循环下去，具体看代码如何实现。

```
#include <reg52.h>
#define uInt16 unsigned int
void DelayMS(uInt16 ValMS)
{    /* 同上 */}
void main(void)
{
    int i;                          //循环变量
    while(1)
    {
        P2 = 0xff;                  //设定 LED 灯初始值
        for(i = 0;i < 8;i++)
        {
            P2 = P2 << 1;           //移位、依次点亮
            DelayMS(100);           //延时
        }
    }
}
```

此程序了，就不做详细说明了，大家只需结合上面的左移示意图和下面的山寨流程图，流程图如图 4-10 所示，肯定就能看懂，相信自己是最棒的，不信大家可以试一试，^_^。

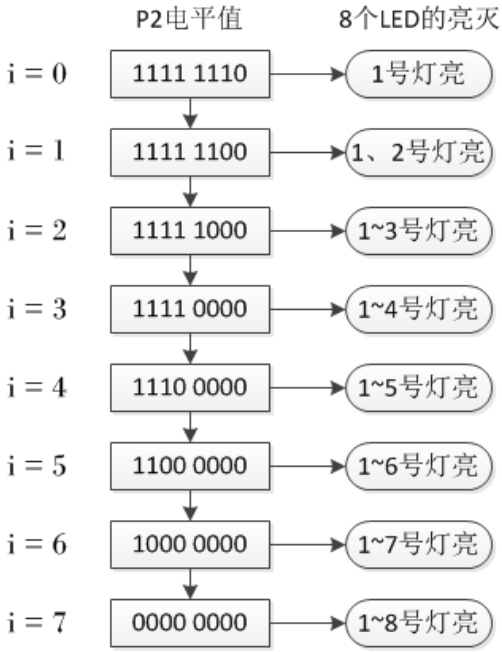

图 4-10 流水灯示意图

4.9 知识扩展——混合编程

本资料所有的程序是用 C 语言来编写，但是读者要知道汇编也是可以编写程序的，两者各有优缺点啊，但是 C 语言的优点更为突出罢了，所以现在用汇编来编写程序的人越来越少，我也是其中一个，可有一个概念，需要大家们了解，那就是混合编程，各取所长。例如用 C 语言写的延时函数，只是一个大概的时间，而用汇编可以写出较为精确的延时，这里笔者还是以跑马灯为例，来简简单单的说说 C 语言和汇编的混合编程，若看不懂就先放一放，若感兴趣自己可以去深入的学习一下。

建立工程、添加文件的过程请看上面 Keil4 简介，源码如下。这里需要对 Keil4 设置几点。

（1）如图 4-11 所示，右键单击你写的.c 文件，接着单击“Options for File ‘LED.c’…”打开如图 3-12 所示的对话框，在箭头所指的两项前打勾（默认是灰色的，要让其变成黑色的），最后单击“OK”按钮。

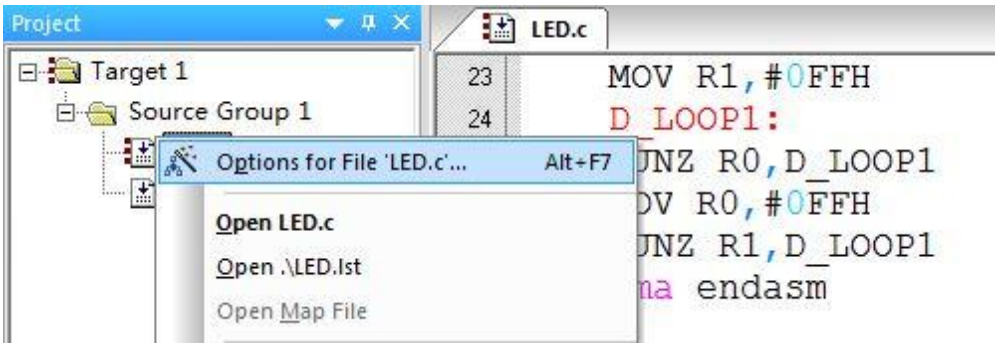

图 4-11 选择 Options for File ’LED.c’对话框

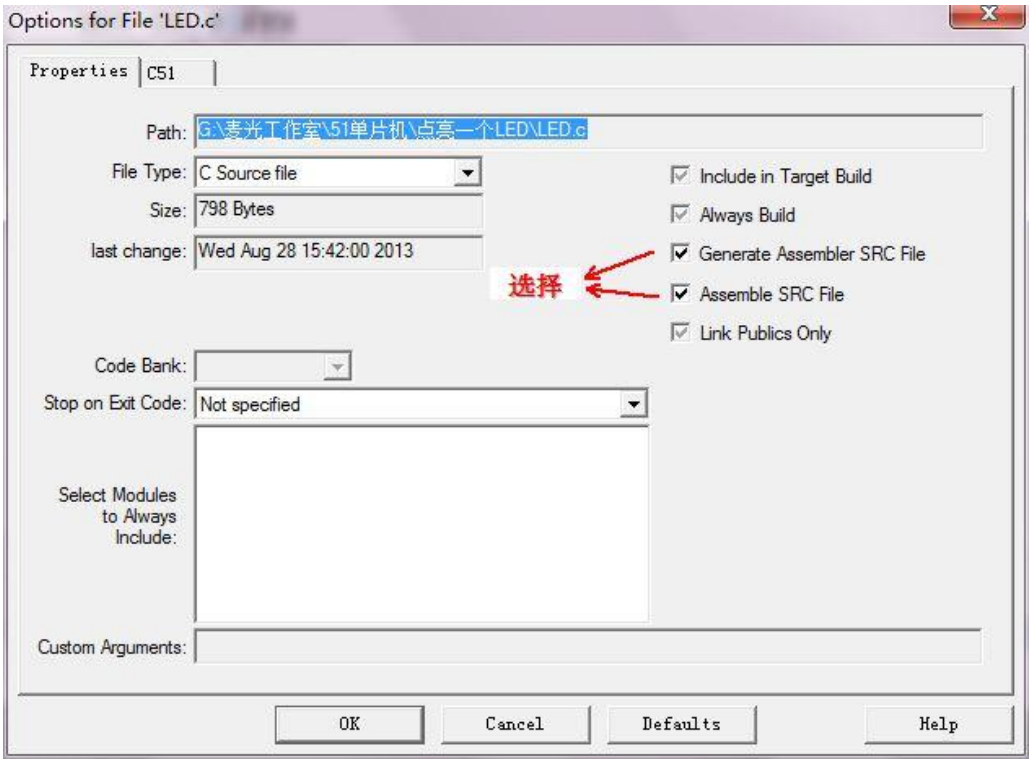

图 4-12 Options for File ’LED.c’设置对话框

（2）接着按 Keil4 操作步骤的第 7 步打开文件添加对话框，一直定位到大家您 Keil4 安

装路径的 LIB 文件下（笔者的路径：D:\PRO_XYMB\keil4\C51\LIB），选择 C51S（添加方法如图 2-8 所示），不同的是要先在“文件类型”中选择“.lib”的后缀名，并添加，这样在“Source Group 1”下会多出一个 C51S.LIB 的文件。接着编译，就会生成可执行文件，下载到单片机中，同样会看到跑马灯的效果。

实验 6 C 语言和汇编语言的混合编程

```
1.  #include <reg52.h>
2.  #include <intrins.h>
3.  #define uChar8 unsigned char
4.  #define uInt16 unsigned int
5.  void DelayMS(void)
6.  {
7.  #pragma asm
8.      MOV R0,#0FFH
9.      MOV R1,#0FFH
10.     D_LOOP1:
11.     DJNZ R0,D_LOOP1
12.     MOV R0,#0FFH
13.     DJNZ R1,D_LOOP1
14. #pragma endasm
15. }
16. void main(void)
17. {
18.     uChar8 uTempVal;
19.     uTempVal = 0xfe;
20.     while(1)
21.     {
22.         P2 = uTempVal;
23.         uTempVal = _crol_(uTempVal,1);
24.         DelayMS();
25.     }
```

程序这里不解释，大家自行查阅相关书籍，弄懂即可。