By 蒋晨辉
QQ：272402202

给单片机初学者的大餐

学习单片机也已经有几年了，藉此机会和大家聊一下我学习过程中的一些经历和想法吧。也感谢一线工人提供了这个机会。希望大家有什么好的想法和建议都直接跟帖说出来。毕竟只有交流才能够碰撞出火花来^_^。

几年前，和众多初学者一样，我接触到了单片机，立刻被其神奇的功能所吸引，从此不能自拔。很多个日夜就这样陪伴着它度过了。期间也遇到过非常多的问题，也一度被这些问题所困惑……等到回过头来，看到自己曾经走过的路，唏嘘不已。经常混迹于论坛里，也看到了很多初学者发的求助帖子，看到他们走在自己曾走过的弯路上，忽然想到了自己的那段日子，心里竟然莫名的冲动，凡此总总，我总是尽自己所能去回帖。很多时候，都想写一点什么东西出来，希望对广大的初学者有一点点帮助。但总是不知从何处写起。今天借一线工人的台，唱一唱我的戏☺。“卖弄”也好，“吹嘘”也罢，我只是想认真的写写我这一路走来历经的总总，把其中值得注意，以及经验的地方写出来，权当是我对自己的一个总结吧。而作为看官的你，如果看到了我的错误，还请一定指正，这样对我以及其它读者都有帮助，而至于你如果从中能够收获到些许，那便是我最大的欣慰了。姑妄言之，姑妄听之。如果有啥好的想法和建议一定要说出来。

一路学习过来的过程中，帮助最大之一无疑来自于网络了。很多时候，通过网络，我们都可以获取到所需要的学习资料。但是，随着我们学习的深入，我们会慢慢发现，网络提供的东西是有限度的，好像大部分的资料都差不多，或者说是适合大部分的初学者所需，而当我们想更进一步提高时，却发现能够获取到的资料越来越少，相信各位也会有同感，铺天盖地的单片机资料中大部分不是流水灯就是 LED，液晶，而且也只是仅仅作功能性的演示。于是有些人选择了放弃，或者是转移到其他兴趣上面去了，而只有少部分人选择了继续摸索下去，结合市面上的书籍，然后在网络上锲而不舍的搜集资料，再从牛人的只言片语中去体会，不断动手实践，慢慢的，也摸索出来了自己的一条路子。当然这个过程必然是艰辛的，而他学会了之后也不会在网络上轻易分享自己的学习成果。如此恶性循环下去，也就不难理解为什么初级的学习资料满天飞，而深入一点的学习资料却很少的原因了。相较于其他领域，单片机技术的封锁更加容易。尽管已经问世了很多年了，有价值的资料还是相当的欠缺，大部分的资料都是止于入门阶段或者是简单的演示实验。但是在实际工程应用中却是另外一回事。有能力的高手无暇或者是不愿公开自己的学习经验。

很多时候，我也很困惑，看到国外爱好者毫不保留的在网络上发布自己的作品，我忽然感觉到一丝丝的悲哀。也许，我们真的该转变一下思路了，帮助别人，其实也是在帮助自己。啰啰嗦嗦的说了这么多，相信大家能够明白说的是什么意思。在接下来的一段日子里，我将会结合电子工程师之家举办的主题周活动写一点自己的想法。尽可能从实用的角度去讲述。希望能够帮助更多的初学者更上一层楼。而关于这个主题周的最大主题我想了这样的一个名字“从单片机初学者迈向单片机工程师”。名字挺大挺响亮，给我的压力也挺大的，但我会努力，争取使这样的一系列文章能够带给大家一点帮助，而不是看后大跌眼镜。这样的一系列文章主要的对象是初学者，以及想从初学者更进一步提高的读者。而至于老手，以及那些牛 XX 的人，希望能够给我们这些初学者更多的一些指点哈~@_@~.

从这一章开始，我们开始迈入单片机的世界。在我们开始这一章具体的学习之前，有必要给大家先说明一下。在以后的系列文章中，我们将以 51 内核的单片机为载体，C 语言为编程语言，开发环境为 KEIL uv3。至于为什么选用 C 语言开发，好处不言而喻，开发速度快，效率高，代码可复用率高，结构清晰，尤其是在大型的程序中，而且随着编译器的不断升级，其编译后的代码大小与汇编语言的差距越来越小。而关于 C 语言和汇编之争，就像那个啥，每隔一段时间总会有人挑起这个话题，如果你感兴趣，可以到网上搜索相关的帖子自行阅读。不是说汇编不重要，在很多对时序要求非常高的场合，需要利用汇编语言和 C 语言混合编程才能够满足系统的需求。在我们学习掌握 C 语言的同时，也还需要利用闲余的时间去学习了解汇编语言。

1.从点亮 LED(发光二极管)开始

在市面上众多的单片机学习资料中，最基础的实验无疑于点亮 LED 了，即控制单片机的 I/O 的电平的变化。

如同如下实例代码一般

```
void main(void)
{
    LedInit() ;
    While(1)
    {
        LED = ON ;
        DelayMs(500) ;
        LED = OFF ;
        DelayMs(500) ;
    }
}
```

程序很简单，从它的结构可以看出，LED 先点亮 500MS，然后熄灭 500MS，如此循环下去，形成的效果就是 LED 以 1HZ 的频率进行闪烁。下面让我们分析上面的程序有没有什么问题。

看来看出，好像很正常的啊，能有什么问题呢？这个时候我们应该换一个思路去想了。试想，整个程序除了控制 LED = ON ； LED = OFF； 这两条语句外，其余的时间，全消耗在了 DelayMs(500)这两个函数上。而在实际应用系统中是没有哪个系统只闪烁一只 LED 就其它什么事情都不做了的。因此，在这里我们要想办法，把 CPU 解放出来，让它不要白白浪费 500MS 的延时等待时间。宁可让它一遍又一遍的扫描看有哪些任务需要执行，也不要让它停留在某个地方空转消耗 CPU 时间。

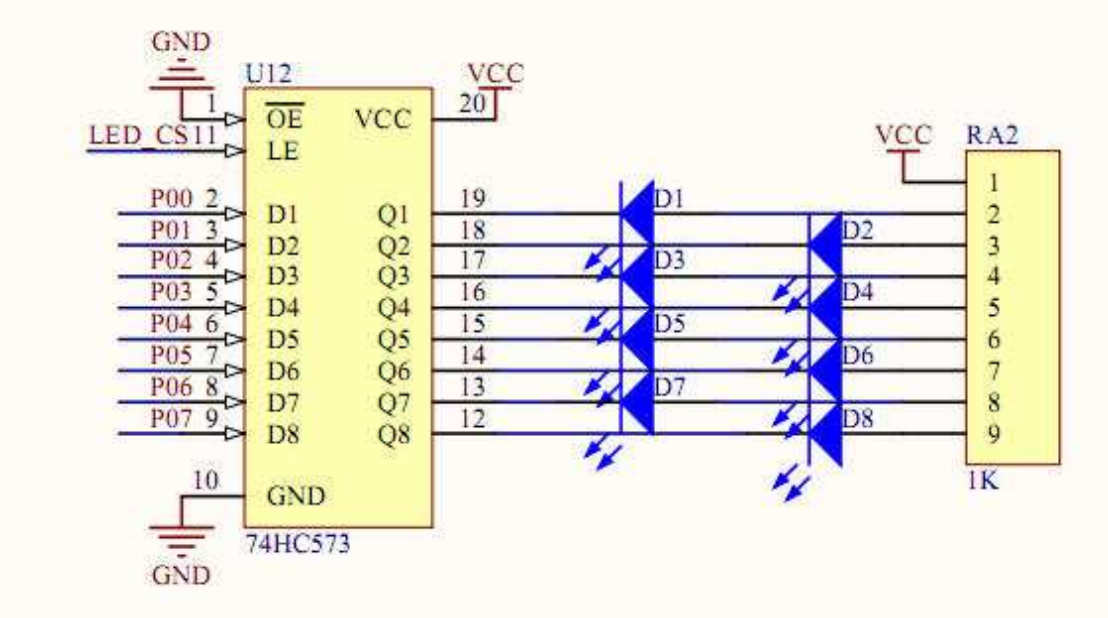
从上面我们可以总结出

(1) 无论什么时候我们都要以实际应用的角度去考虑程序的编写。

(2) 无论什么时候都不要让 CPU 白白浪费等待，尤其是延时(超过 1MS)这样的地方。

下面让我们从另外一个角度来考虑如何点亮一颗 LED。
先看看我们的硬件结构是什么样子的。

我手上的单片机板子是电子工程师之家的开发的学习板。就以它的实际硬件连接图来分析吧。如下图所示



一般的 LED 的正常发光电流为 10~20MA 而低电流 LED 的工作电流在 2mA 以下（亮度与普通发光管相同）。在上图中我们可知，当 Q1~Q8 引脚上面的电平为低电平时，LED 发光。通过 LED 的电流约为（VCC - Vd）/ RA2 。其中 Vd 为 LED 导通后的压降，约为 1.7V 左右。这个导通压降根据 LED 颜色的不同，以及工作电流的大小的不同，会有一定的差别。下面一些参数是网上有人测出来的，供大家参考。

红色的压降为 1.82-1.88V，电流 5-8mA，

绿色的压降为 1.75-1.82V，电流 3-5mA，

橙色的压降为 1.7-1.8V，电流 3-5mA

兰色的压降为 3.1-3.3V，电流 8-10mA，

白色的压降为 3-3.2V，电流 10-15mA，

(供电电压 5V，LED 直径为 5mm)

FUNCTION TABLE

Inputs			Output
Output Enable	Latch Enable	D	Q
L	H	H	H
L	H	L	L
L	L	X	no change
H	X	X	Z

X = don't care
Z = high impedance

74HC573 真值表如下：

通过这个真值表我们可以看出。当 OutputEnable 引脚接低电平的时候，并且 LatchEnable 引脚为高电平的时候，Q 端电平与 D 端电平相同。结合我们的 LED 硬件连接图可以知道 LED_CS 端为高电平时候，P0 口电平的变化即 Q 端的电平的变化，进而引起 LED 的亮灭变化。由于单片机的驱动能力有限，在此，74HC573 的主要作用就是起一个输出驱动的作用。需要注意的是，通过 74HC573 的最大电流是有限制的，否则可能会烧坏 74HC573 这个芯片。

I_{OUT}	DC Output Current, per Pin	±35	mA
I_{CC}	DC Supply Current, V_{CC} and GND Pins	±75	mA

上面这个图是从 74HC573 的 DATASHEET 中截取出来的，从上可以看出，每个引脚允许通过的最大电流为 35mA 整个芯片允许通过的最大电流为 75mA。在我们设计相应的驱动电路时候，这些参数是相当重要的，而且是最容易被初学者所忽略的地方。同时在设计的时候，要留出一定量的余量出来，不能说单个引脚允许通过的电流为 35mA，你就设计为 35mA，这个时候你应该把设计的上限值定在 20mA 左右才能保证能够稳定的工作。

（设计相应驱动电路时候，应该仔细阅读芯片的数据手册，了解每个引脚的驱动能力，以及整个芯片的驱动能力）

了解了相应的硬件后，我们再来编写驱动程序。

首先定义 LED 的接口

```
#define LED    P0
```

然后为亮灭常数定义一个宏，由硬件连接图可以，当 P0 输出为低电平时候 LED 亮，P0 输出为高电平时，LED 熄灭。

```
#define LED_ON()        LED = 0x00 ;   //所有 LED 亮
#define LED_OFF()       LED = 0xff ;   //所有 LED 熄灭
```

下面到了重点了，究竟该如何释放 CPU，避免其做延时空等待这样的事情呢。很简单，我们为系统产生一个 1MS 的时标。假定 LED 需要亮 500MS，熄灭 500MS，那么我们可以对这个 1MS 的时标进行计数，当这个计数值达到 500 时候，清零该计数值，同时把 LED 的状态改变。

```
unsigned int g_u16LedTimeCount = 0 ;       //LED 计数器
unsigned char g_u8LedState = 0 ;           //LED 状态标志, 0 表示亮，1 表示熄灭

void LedProcess(void)
{
    if(0 == g_u8LedState)   //如果 LED 的状态为亮，则点亮 LED
    {
```

```
            LED_ON() ;
        }
        else                        //否则熄灭 LED
        {
            LED_OFF() ;
        }
}


void LedStateChange(void)
{
        if(g_bSystemTime1Ms)                    //系统 1MS 时标到
        {
            g_bSystemTime1Ms = 0 ;
            g_u16LedTimeCount++ ;           //LED 计数器加一
            if(g_u16LedTimeCount >= 500)    //计数达到 500,即 500MS 到了,改变 LED 的状态。
            {
                g_u16LedTimeCount = 0 ;
                g_u8LedState    = ! g_u8LedState ;
            }
        }
}
```

上面有一个变量没有提到，就是 g_bSystemTime1Ms 。这个变量可以定义为位变量或者是其它变量，在我们的定时器中断函数中对其置位，其它函数使用该变量后，应该对其复位(清 0) 。

我们的主函数就可以写成如下形式(示意代码)

```
void main(void)
{
        while(1)
        {
            LedProcess() ;
            LedStateChange() ;
        }
}
```

因为 LED 的亮或者灭依赖于 LED 状态变量(g_u8LedState)的改变，而状态变量的改变，又依赖于 LED 计数器的计数值 g_u16LedTimeCount ，只有计数值达到一定后，状态变量才改变)所以，两个函数都没有堵塞 CPU 的地方。让我们来从头到尾分析一遍整个程序的流程。

程序首先执行 LedProcess() ;函数

因为 g_u8LedState 的初始值为 0 (见定义，对于全局变量，在定义的时候最好给其一个确定的值) 所以 LED 被点亮，然后退出 LedStateChange() 函数，执行下一个函数 LedStateChange()

在函数 LedStateChange()内部首先判断 1MS 的系统时标是否到了，如果没有到就直接退出函数，如果到了，就把时标清 0 以便下一个时标消息的到来，同时对 LED 计数器加一，然后再判断 LED 计数器是否到达我们预先想要的值 500，如果没有，则退出函数，如果有，对计数器清 0，以便下次重新计数，同时把 LED 状态变量取反，然后退出函数。

由上面整个流程可以知道，CPU 所做的事情，就是对一些计数器加一，然后根据条件改变状态，再根据这个状态来决定是否点亮 LED。这些函数执行所花的时间都是相当短的，如果主程序中还有其它函数，则 CPU 会顺次往下执行下去。对于其它的函数(如果有的话)也要采取同样的措施，保证其不堵塞 CPU，如果全部基于这种方法设计，那么对于不是非常庞大的系统，我们的系统依旧可以保证多个任务(多个函数)同时执行。系统的实时性得到了一定的保证，从宏观上看来，就是多个任务并发执行。

好了，这一章就到此为止，让我们总结一下，究竟有哪些需要注意的吧。

(1) 无论什么时候我们都要以实际应用的角度去考虑程序的编写。
(2) 无论什么时候都不要让 CPU 白白浪费等待，尤其是延时(超过 1MS)这样的地方。
(3) 设计相应驱动电路时候，应该仔细阅读芯片的数据手册，了解每个引脚的驱动能力，以及整个芯片的驱动能力
(4) 最重要的是，如何去释放 CPU(参考本章的例子)，这是写出合格程序的基础。

附完整程序代码(基于电子工程师之家的单片机开发板)

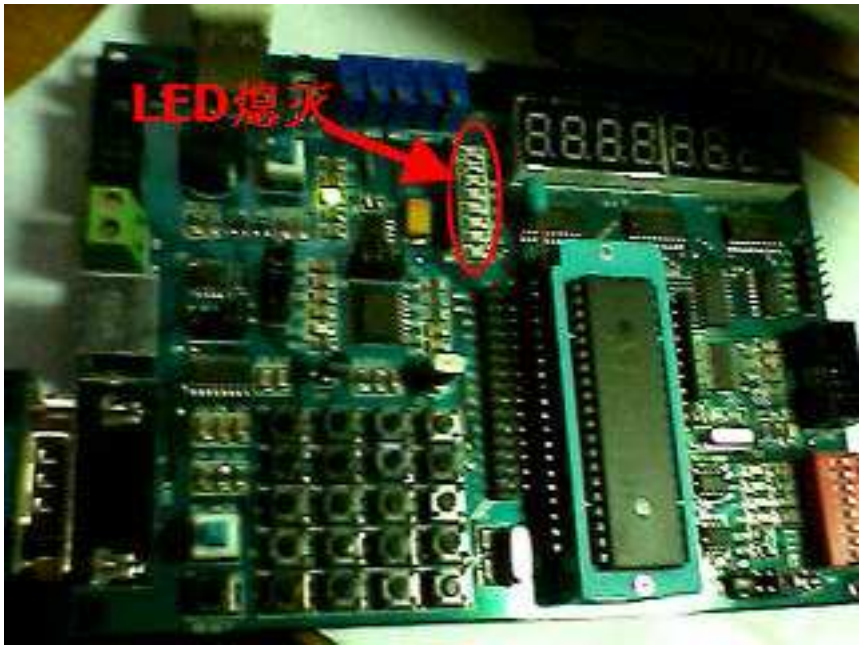
```
#include<reg52.h>

sbit LED_SEG   = P1^4;   //数码管段选
sbit LED_DIG   = P1^5;   //数码管位选
sbit LED_CS11 = P1^6;   //led 控制位
sbit ir=P1^7;
#define LED P0                  //定义 LED 接口
bit     g_bSystemTime1Ms = 0 ;                  // 1MS 系统时标
unsigned int    g_u16LedTimeCount = 0 ; //LED 计数器
unsigned char g_u8LedState = 0 ;          //LED 状态标志, 0 表示亮，1 表示熄灭

#define LED_ON()          LED = 0x00 ;   //所有 LED 亮
#define LED_OFF()         LED = 0xff ;   //所有 LED 熄灭

void Timer0Init(void)
{
    TMOD &= 0xf0 ;
    TMOD |= 0x01 ;        //定时器 0 工作方式 1
    TH0   =   0xfc ;        //定时器初始值
    TL0   =   0x66 ;
    TR0   = 1 ;
    ET0   = 1 ;
```

```
}
void LedProcess(void)
{
    if(0 == g_u8LedState)   //如果 LED 的状态为亮，则点亮 LED
    {
        LED_ON() ;
    }
    else                    //否则熄灭 LED
    {
        LED_OFF() ;
    }
}

void LedStateChange(void)
{
    if(g_bSystemTime1Ms)                //系统 1MS 时标到
    {
        g_bSystemTime1Ms = 0 ;
        g_u16LedTimeCount++ ;           //LED 计数器加一
        if(g_u16LedTimeCount >= 500)    //计数达到 500,即 500MS 到了,改变 LED 的状态。
        {
            g_u16LedTimeCount = 0 ;
            g_u8LedState    = ! g_u8LedState   ;
        }
    }
}

void main(void)
{
    Timer0Init() ;
    EA = 1 ;
    LED_CS11 = 1 ; //74HC595 输出允许
    LED_SEG = 0 ;   //数码管段选和位选禁止(因为它们和 LED 共用 P0 口)
    LED_DIG = 0 ;
    while(1)
    {
        LedProcess() ;
        LedStateChange() ;
    }
}

void Time0Isr(void) interrupt 1
{
    TH0   =   0xfc ;                //定时器重新赋初值
```

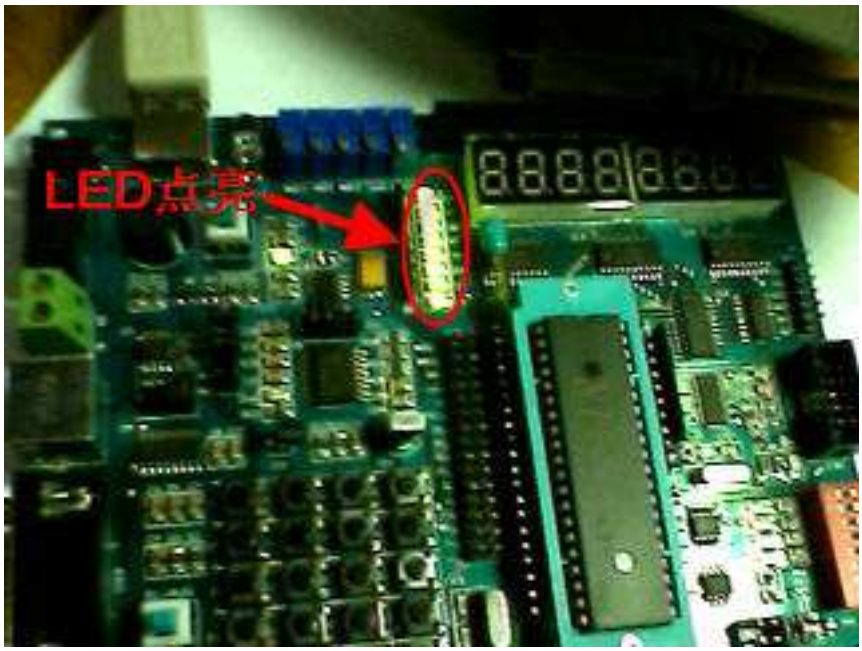
```
    TL0   =   0x66 ;
    g_bSystemTime1Ms = 1 ;      //1MS 时标标志位置位
}
```

实际效果图如下

熄灭时候



点亮时候



1. 好的开始是成功的一半

通过上一章的学习，我想你已经掌握了如何在程序中释放 CPU 了。希望能够继续坚持下去。一个良好的开始是成功的一半。我们今天所做的一切都是为了在单片机编程上做的更好。

在谈论今天的主题之前，先说下我以前的一些经历。在刚开始接触到 C 语言程序的时候，由于学习内容所限，写的程序都不是很大，一般也就几百行而矣。所以所有的程序都完成在一个源文件里面。记得那时候大一参加学校里的一个电子设计大赛，调试了一个多星期，所有程序加起来大概将近 1000 行，长长的一个文件，从上浏览下来都要好半天。出了错误简单的语法错误还好定位，其它一些错误，往往找半天才找的到。那个时候开始知道了模块化编程这个东西，也尝试着开始把程序分模块编写。最开始是把相同功能的一些函数(譬如 1602 液晶的驱动)全部写在一个头文件(.h)文件里面，然后需要调用的地方包含进去，但是很快发现这种方法有其局限性，很容易犯重复包含的错误。而且调用起来也很不方便。很快暑假的电子设计大赛来临了，学校对我们的单片机软件编程进行了一些培训。由于学校历年来参加国赛和省赛，因此积累了一定数量的驱动模块，那些日子，老师每天都会布置一定量的任务，让我们用这些模块组合起来，完成一定功能。而正是那些日子模块化编程的培训，使我对于模块化编程有了更进一步的认识。并且程序规范也开始慢慢注意起来。此后的日子，无论程序的大小，均采用模块化编程的方式去编写。很长一段时间以来，一直有单片机爱好者在 QQ 上和我一起交流。有时候，他们会发过来一些有问题的程序源文件，让我帮忙修改一下。同样是长长的一个文件，而且命名极不规范，从头看下来，着实是痛苦，说实话，还真不如我重新给他们写一个更快一些，此话到不假，因为手头积累了一定量的模块，在完成一个新的系统时候，只需要根据上层功能需求，在底层模块的支持下，可以很快方便的完成。而不需要从头

到尾再一砖一瓦的重新编写。藉此，也可以看出模块化编程的一个好处，就是可重复利用率高。下面让我们揭开模块化神秘面纱，一窥其真面目。

C 语言源文件 *.c

提到 C 语言源文件，大家都不会陌生。因为我们平常写的程序代码几乎都在这个 XX.C 文件里面。编译器也是以此文件来进行编译并生成相应的目标文件。作为模块化编程的组成基础，我们所要实现的所有功能的源代码均在这个文件里。理想的模块化应该可以看成是一个黑盒子。即我们只关心模块提供的功能，而不管模块内部的实现细节。好比我们买了一部手机，我们只需要会用手机提供的功能即可，不需要知晓它是如何把短信发出去的，如何响应我们按键的输入，这些过程对我们用户而言，就是是一个黑盒子。在大规模程序开发中，一个程序由很多个模块组成，很可能，这些模块的编写任务被分配到不同的人。而你在编写这个模块的时候很可能就需要利用到别人写好的模块的借口，这个时候我们关心的是，它的模块实现了什么样的接口，我该如何去调用，至于模块内部是如何组织的，对于我而言，无需过多关注。而追求接口的单一性，把不需要的细节尽可能对外部屏蔽起来，正是我们所需要注意的地方。

C 语言头文件 *.h

谈及到模块化编程，必然会涉及到多文件编译，也就是工程编译。在这样的一个系统中，往往会有多个 C 文件，而且每个 C 文件的作用不尽相同。在我们的 C 文件中，由于需要对外提供接口，因此必须有一些函数或者是变量提供给外部其它文件进行调用。

假设我们有一个 LCD.C 文件，其提供最基本的 LCD 的驱动函数

```
LcdPutChar(char cNewValue) ;   //在当前位置输出一个字符
```

而在我们的另外一个文件中需要调用此函数，那么我们该如何做呢？

头文件的作用正是在此。可以称其为一份接口描述文件。其文件内部不应该包含任何实质性的函数代码。我们可以把这个头文件理解成为一份说明书，说明的内容就是我们的模块对外提供的接口函数或者是接口变量。同时该文件也包含了一些很重要的宏定义以及一些结构体的信息，离开了这些信息，很可能就无法正常使用接口函数或者是接口变量。但是总的原则是：不该让外界知道的信息就不应该出现在头文件里，而外界调用模块内接口函数或者是接口变量所必须的信息就一定要出现在头文件里，否则，外界就无法正确的调用我们提供的接口功能。因而为了让外部函数或者文件调用我们提供的接口功能，就必须包含我们提供的这个接口描述文件----即头文件。同时，我们自身模块也需要包含这份模块头文件(因为其包含了模块源文件中所需要的宏定义或者是结构体)，好比我们平常所用的文件都是一式三份一样，模块本身也需要包含这个头文件。

下面我们来定义这个头文件，一般来说，头文件的名字应该与源文件的名字保持一致，这样我们便可以清晰的知道哪个头文件是哪个源文件的描述。

于是便得到了 LCD.C 的头文件 LCD.h 其内容如下。

```
#ifndef       _LCD_H_
#define       _LCD_H_
extern     LcdPutChar(char cNewValue) ;
#endif
```

这与我们在源文件中定义函数时有点类似。不同的是，在其前面添加了 extern 修饰符表明其是一个外部函数，可以被外部其它模块进行调用。

```
#ifndef       _LCD_H_
#define       _LCD_H_
#endif
```

这个几条条件编译和宏定义是为了防止重复包含。假如有两个不同源文件需要调用 LcdPutChar(char cNewValue)这个函数，他们分别都通过#include “Lcd.h”把这个头文件包含了进去。在第一个源文件进行编译时候，由于没有定义过 _LCD_H_ 因此 #ifndef _LCD_H_ 条件成立，于是定义_LCD_H_ 并将下面的声明包含进去。在第二个文件编译时候，由于第一个文件包含时候，已经将_LCD_H_定义过了。因此#ifndef _LCD_H_ 不成立，整个头文件内容就没有被包含。假设没有这样的条件编译语句，那么两个文件都包含了 extern LcdPutChar(char cNewValue)；就会引起重复包含的错误。

不得不说的 typedef

很多朋友似乎了习惯程序中利用如下语句来对数据类型进行定义

```
#define uint     unsigned int
#define uchar    unsigned char
```

然后在定义变量的时候 直接这样使用

```
uint    g_nTimeCounter = 0 ;
```

不可否认，这样确实很方便，而且对于移植起来也有一定的方便性。但是考虑下面这种情况你还会这么认为吗？

```
#define PINT unsigned int *     //定义 unsigned int 指针类型
PINT     g_npTimeCounter, g_npTimeState ;
```

那么你到底是定义了两个 unsigned int 型的指针变量，还是一个指针变量，一个整形变量呢？而你的初衷又是什么呢，想定义两个 unsigned int 型的指针变量吗？如果是这样，那么估计过不久就会到处抓狂找错误了。

庆幸的是 C 语言已经为我们考虑到了这一点。typedef 正是为此而生。为了给变量起一个别名我们可以用如下的语句

```
typedef   unsigned   int       uint16 ;      //给指向无符号整形变量起一个别名 uint16
typedef   unsigned   int   *   puint16 ;     //给指向无符号整形变量指针起一个别名 puint16
```

在我们定义变量时候便可以这样定义了：

```
uint16      g_nTimeCounter   =   0 ;   //定义一个无符号的整形变量
puint16     g_npTimeCounter   ;        //定义一个无符号的整形变量的指针
```

在我们使用 51 单片机的 C 语言编程的时候，整形变量的范围是 16 位，而在基于 32 的微处理下的整形变量是 32 位。倘若我们在 8 位单片机下编写的一些代码想要移植到 32 位的处理器上，那么很可能我们就需要在源文件中到处修改变量的类型定义。这是一件庞大的工作，为了考虑程序的可移植性，在一开始，我们就应该养成良好的习惯，用变量的别名进行定义。

如在 8 位单片机的平台下，有如下一个变量定义

```
uint16      g_nTimeCounter   =   0 ;
```

如果移植 32 单片机的平台下，想要其的范围依旧为 16 位。

可以直接修改 uint16 的定义，即

```
typedef   unsigned   short   int        uint16 ;
```

这样就可以了，而不需要到源文件处处寻找并修改。

将常用的数据类型全部采用此种方法定义，形成一个头文件，便于我们以后编程直接调用。

文件名 MacroAndConst.h

其内容如下：

```
#ifndef      _MACRO_AND_CONST_H_
#define      _MACRO_AND_CONST_H_

typedef       unsigned int       uint16;
typedef       unsigned int       UINT;
typedef       unsigned int       uint;
typedef       unsigned int       UINT16;
typedef       unsigned int       WORD;
typedef       unsigned int       word;
typedef          int             int16;
typedef          int             INT16;
typedef       unsigned long      uint32;

typedef       unsigned long        UINT32;
typedef       unsigned long        DWORD;
typedef       unsigned long        dword;
typedef       long                 int32;
typedef       long                 INT32;
typedef       signed     char      int8;
typedef       signed     char      INT8;
typedef       unsigned char         byte;
typedef       unsigned char         BYTE;
typedef       unsigned char        uchar;
typedef       unsigned char        UINT8;
typedef       unsigned char        uint8;
typedef       unsigned char        BOOL;

#endif
```

至此，似乎我们对于源文件和头文件的分工以及模块化编程有那么一点概念了。那么让我们趁热打铁，将上一章的我们编写的 LED 闪烁函数进行模块划分并重新组织进行编译。

在上一章中我们主要完成的功能是 P0 口所驱动的 LED 以 1Hz 的频率闪烁。其中用到了定时器，以及 LED 驱动模块。因而我们可以简单的将整个工程分成三个模块，定时器模块，LED 模块，以及主函数

对应的文件关系如下

main.c

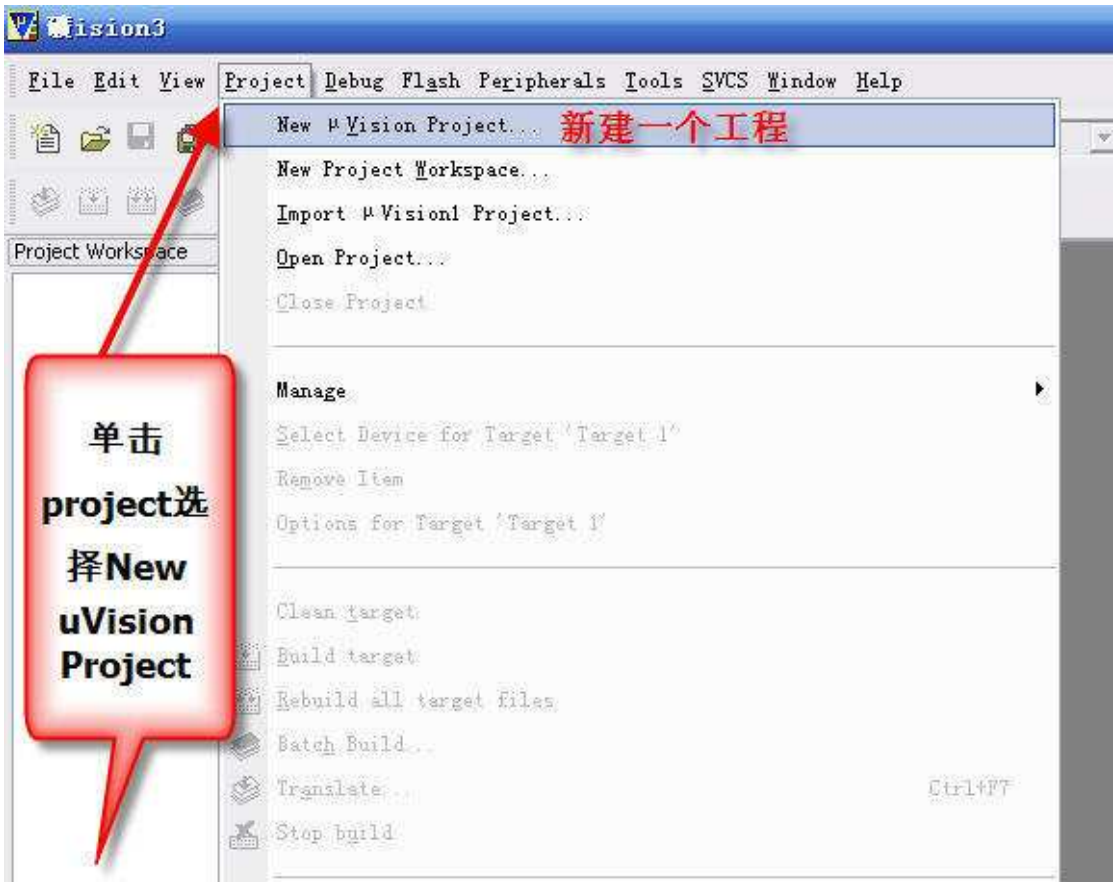
Timer.c --→Timer.h

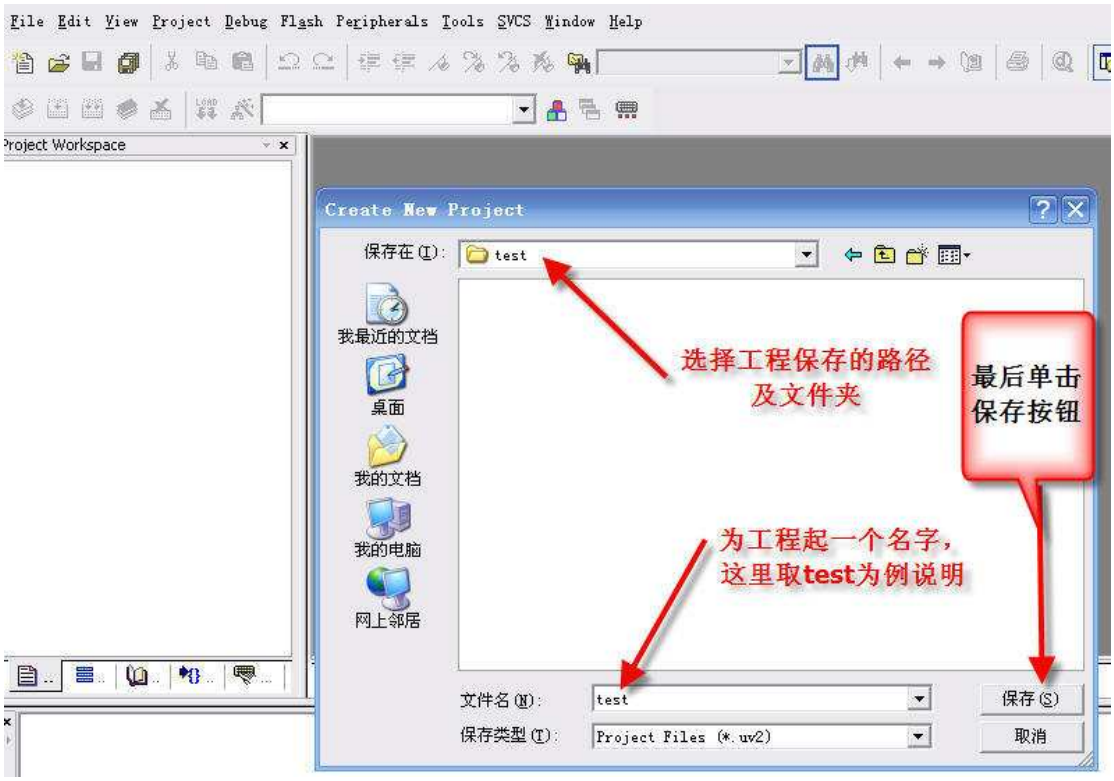
Led.c --→Led.h

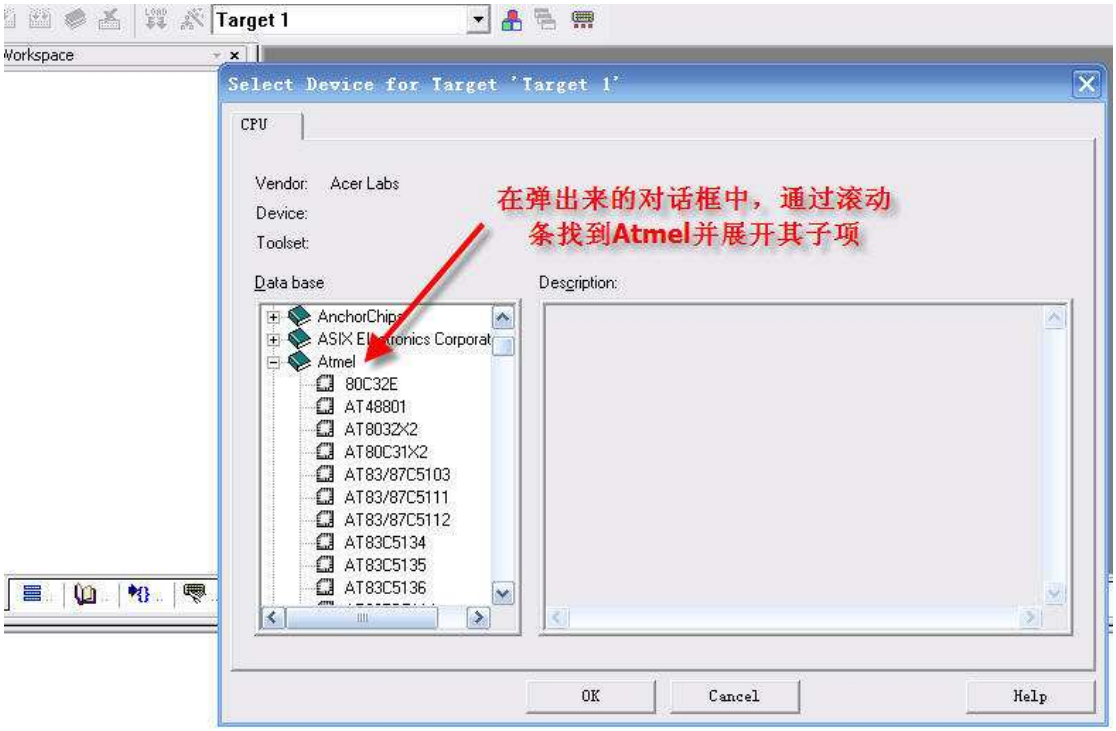
在开始重新编写我们的程序之前，先给大家讲一下如何在 KEIL 中建立工程模板吧，这个模板是我一直沿用至今。希望能够给大家一点启发。

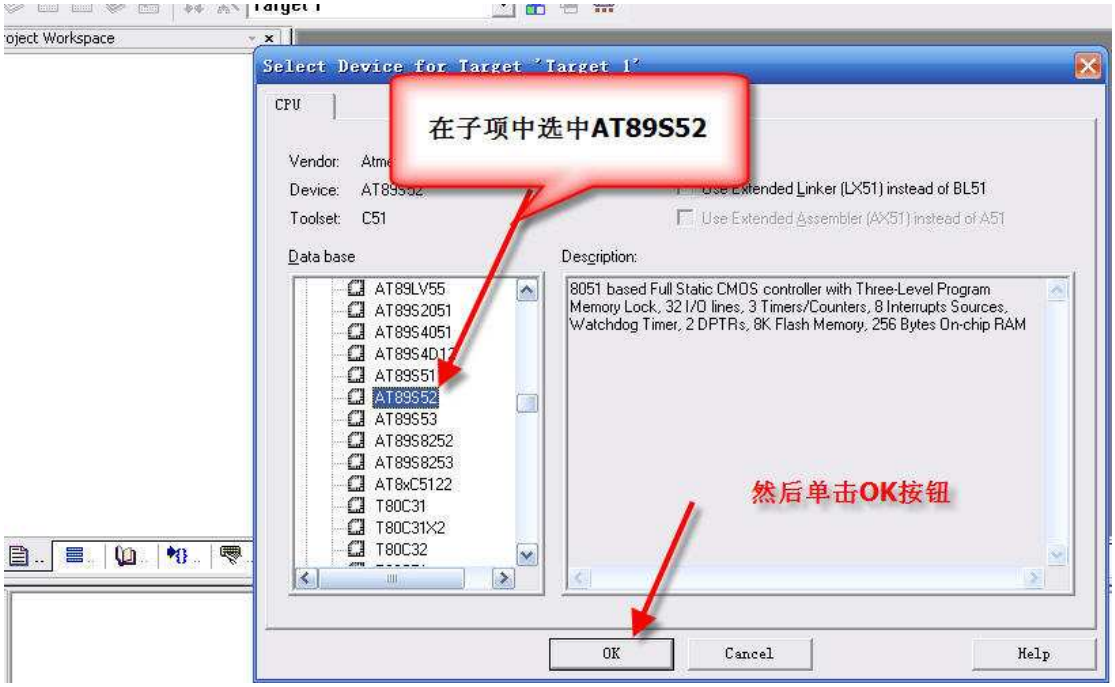
下面的内容就主要以图片为主了。同时辅以少量文字说明。

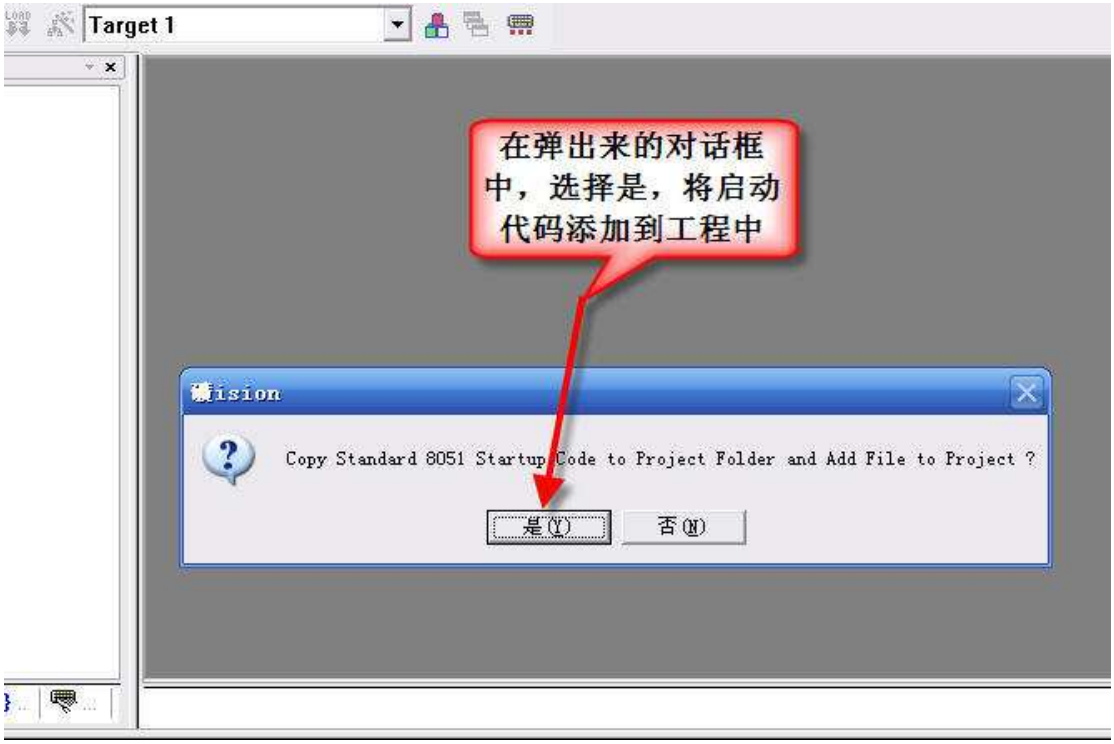
我们以芯片 AT89S52 为例。

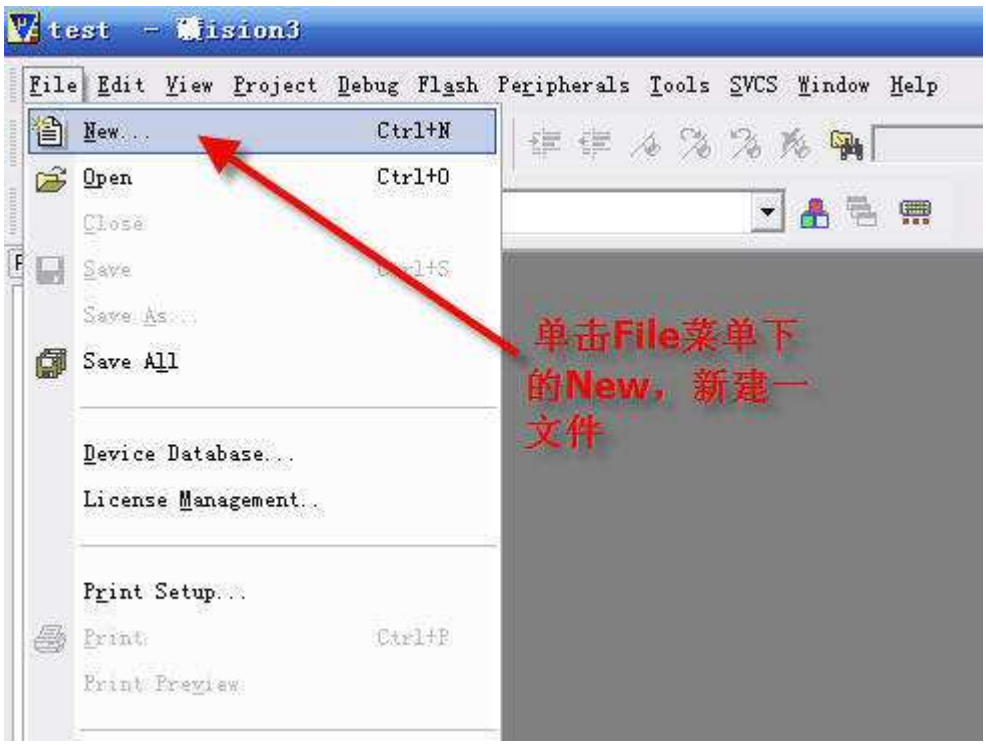


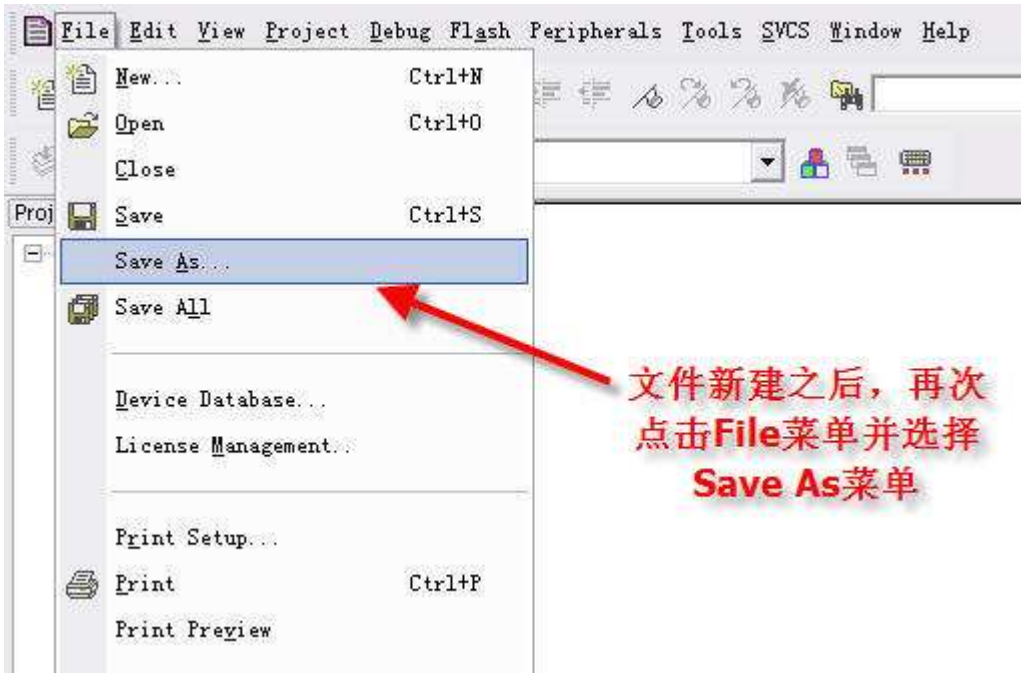

File Edit View Project Debug Flash Peripherals Tools SVCS Window Help
Project Workspace
Create New Project
保存在(I):
test
我最近的文档
桌面
我的文档
我的电脑
网上邻居
选择工程保存的路径
及文件夹
最后单击
保存按钮
为工程起一个名字，
这里取test为例说明
文件名(N):
test
保存(S)
保存类型(T):
Project Files (*.uv2)
取消

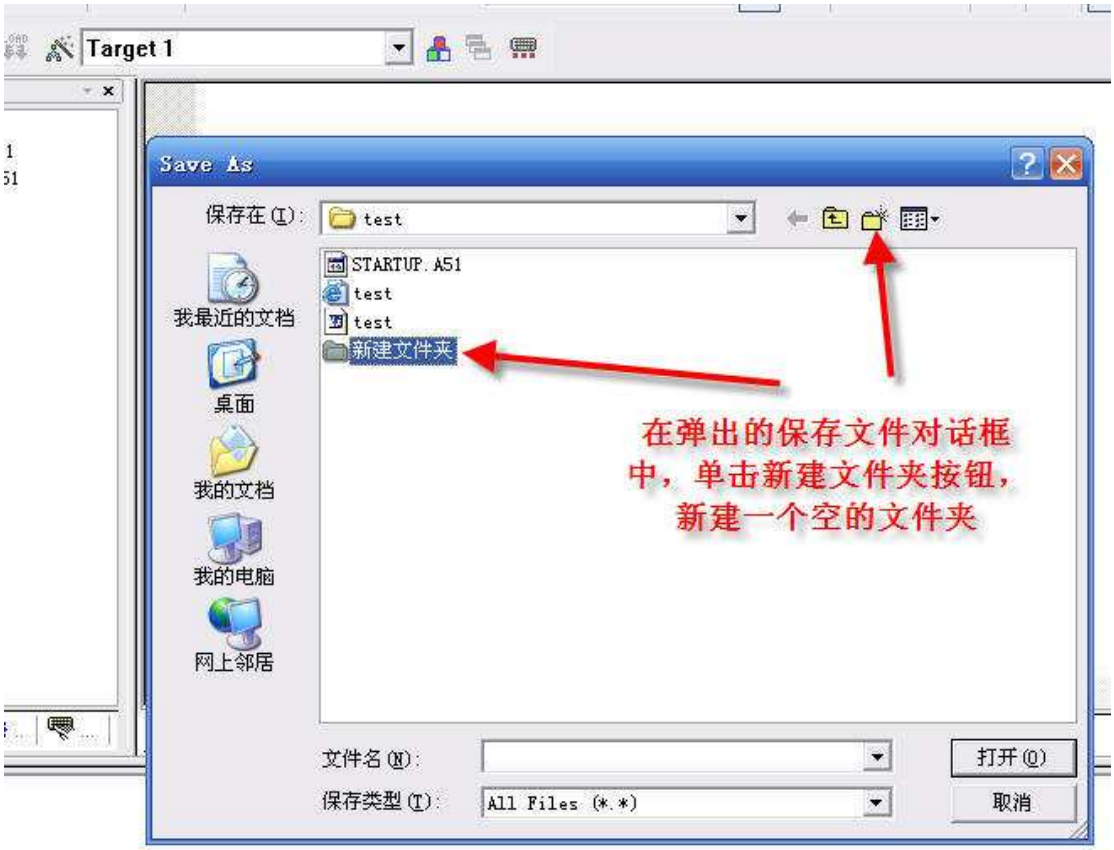

Target 1
Workspace
Select Device for Target 'Target 1'
CPU
Vendor: Acer Labs
Device:
Toolset:
在弹出来的对话框中，通过滚动
条找到Atmel并展开其子项
Data base
Description:
AnchorChips
ASIX El tronics Corporat
Atmel
80C32E
AT48801
AT8032X2
AT80C31X2
AT83/87C5103
AT83/87C5111
AT83/87C5112
AT83C5134
AT83C5135
AT83C5136
OK
Cancel
Help

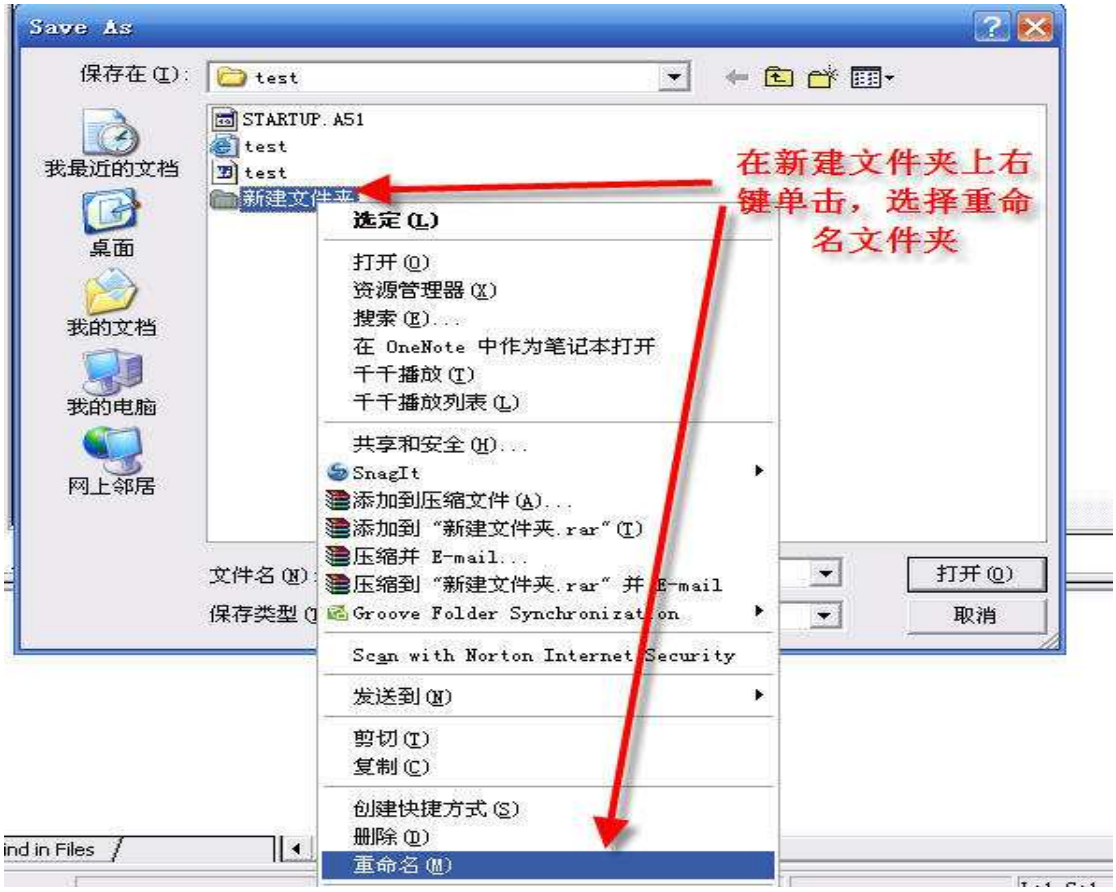

Select Device for Target 'Target 1'
CPU
在子项中选中AT89S52
Vendor:
Device:
Toolset: C51
Use Extended Linker (LX51) instead of BL51
Use Extended Assembler (AX51) instead of A51
Data base
Description:
AT89LV55
AT89S2051
AT89S4051
AT89S53
AT89S8252
AT89S8253
AT8xC5122
T80C31
T80C31X2
T80C32
8051 based Full Static CMOS controller with Three-Level Program Memory Lock, 32 I/O lines, 3 Timers/Counters, 8 Interrupts Sources, Watchdog Timer, 2 DPTRs, 8K Flash Memory, 256 Bytes On-chip RAM
然后单击OK按钮
OK
Cancel
Help


Target 1
在弹出来的对话框中，选择是，将启动代码添加到工程中
Copy Standard 8051 Startup Code to Project Folder and Add File to Project ?
是(Y)
否(N)


test - Vision3
File Edit View Project Debug Flash Peripherals Tools SVCS Window Help
New... Ctrl+N
Open Ctrl+O
Close
Save Ctrl+S
Save As...
Save All
Device Database...
License Management...
Print Setup...
Print Ctrl+P
Print Preview
单击File菜单下的New，新建一文件

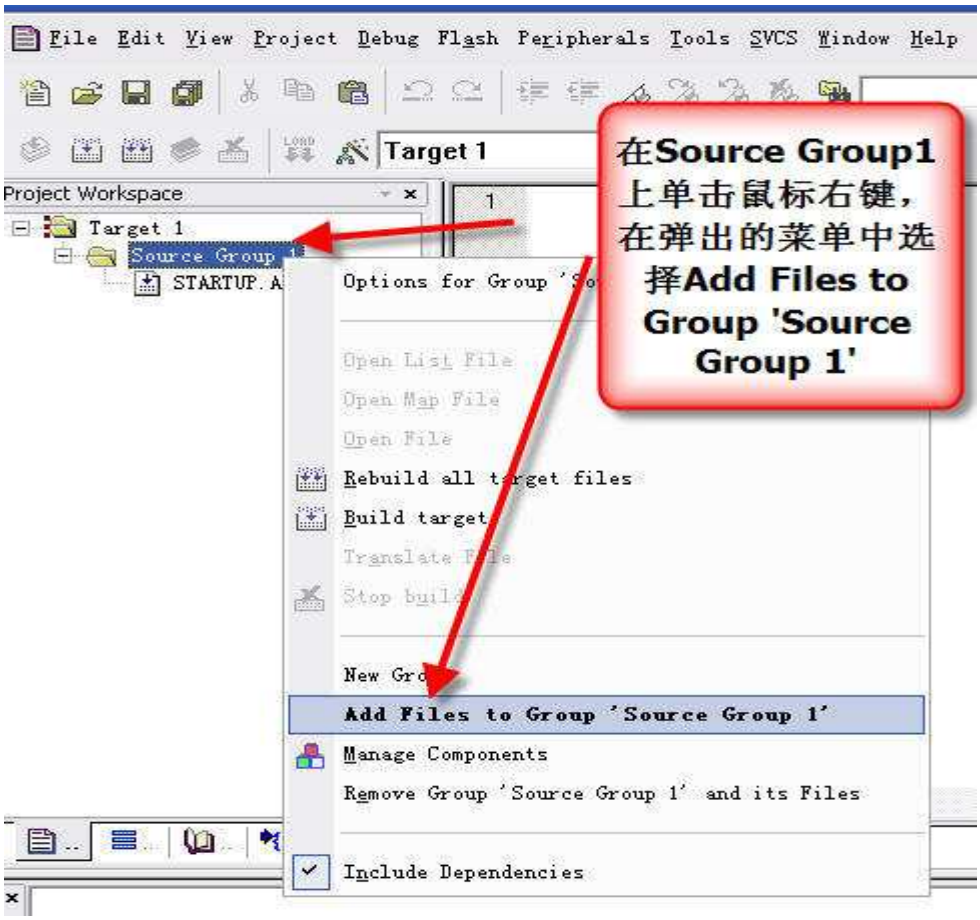

File Edit View Project Debug Flash Peripherals Tools SVCS Window Help
New... Ctrl+N
Open Ctrl+O
Close
Save Ctrl+S
Save As...
Save All
Device Database...
License Management...
Print Setup...
Print Ctrl+P
Print Preview
文件新建之后，再次点击File菜单并选择Save As菜单


Target 1
Save As
保存在(I):
test
STARTUP.A51
test
test
新建文件夹
我最近的文档
桌面
我的文档
我的电脑
网上邻居
文件名(N):
保存类型(T):
All Files (*.*)
打开(O)
取消
在弹出的保存文件对话框
中，单击新建文件夹按钮，
新建一个空的文件夹

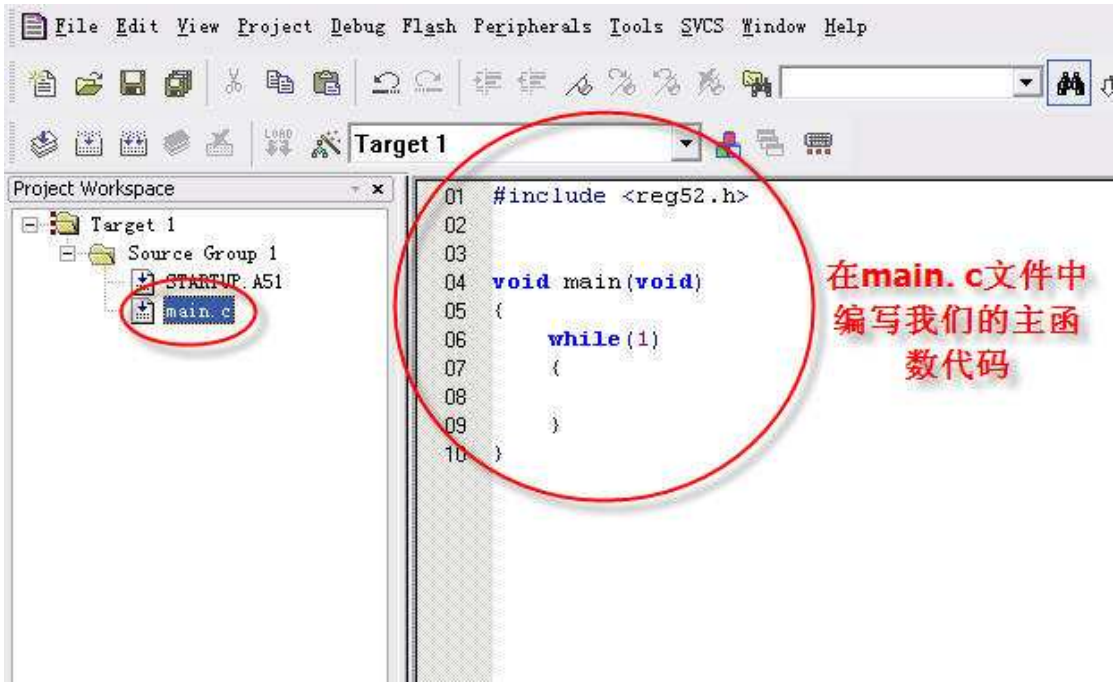

Save As
保存在(I): test
STARTUP.A51
test
test
新建文件夹
我最近的文档
桌面
我的文档
我的电脑
网上邻居
选定(L)
打开(O)
资源管理器(X)
搜索(E)...
在 OneNote 中作为笔记本打开
千千播放(T)
千千播放列表(L)
共享和安全(H)...
SnagIt
添加到压缩文件(A)...
添加到"新建文件夹.rar"(T)
压缩并 E-mail...
压缩到"新建文件夹.rar"并 E-mail
Groove Folder Synchronization
Scan with Norton Internet Security
发送到(N)
剪切(T)
复制(C)
创建快捷方式(S)
删除(D)
重命名(M)
文件名(N):
保存类型(T):
打开(O)
取消
在新建文件夹上右键单击，选择重命名文件夹

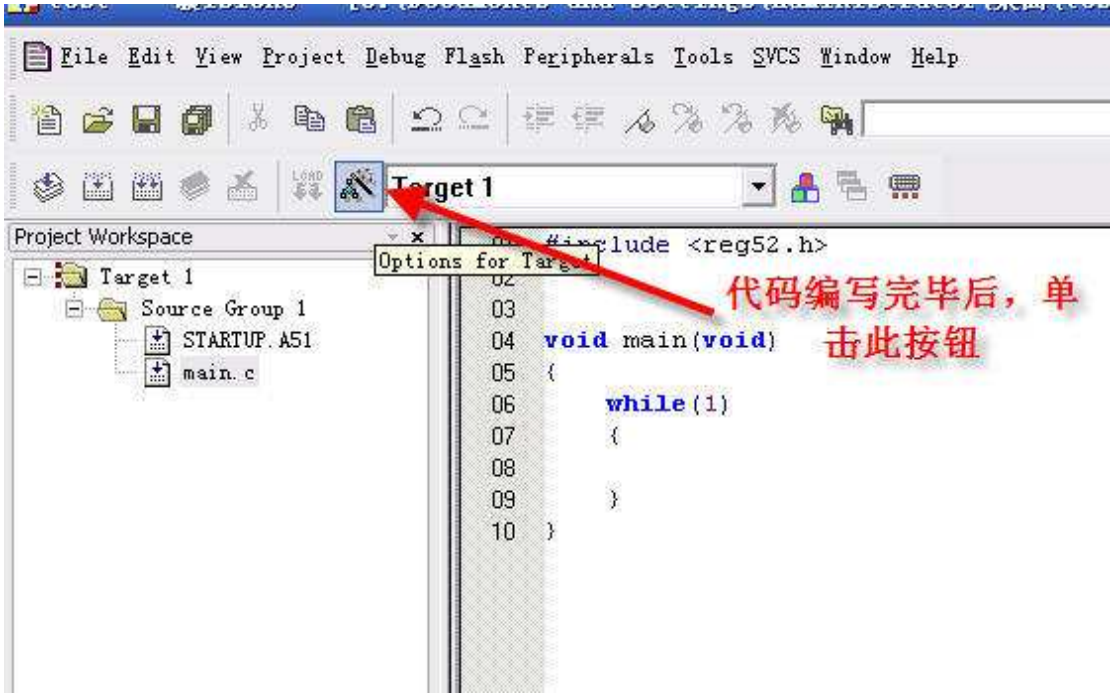

Save As
保存在(I): test
STARTUP.A51
test
test
src
我最近的文档
桌面
我的文档
我的电脑
网上邻居
文件名(N):
保存类型(T): All Files (*.*)
打开(O)
取消
将文件夹重命名为src
双击进入文件夹

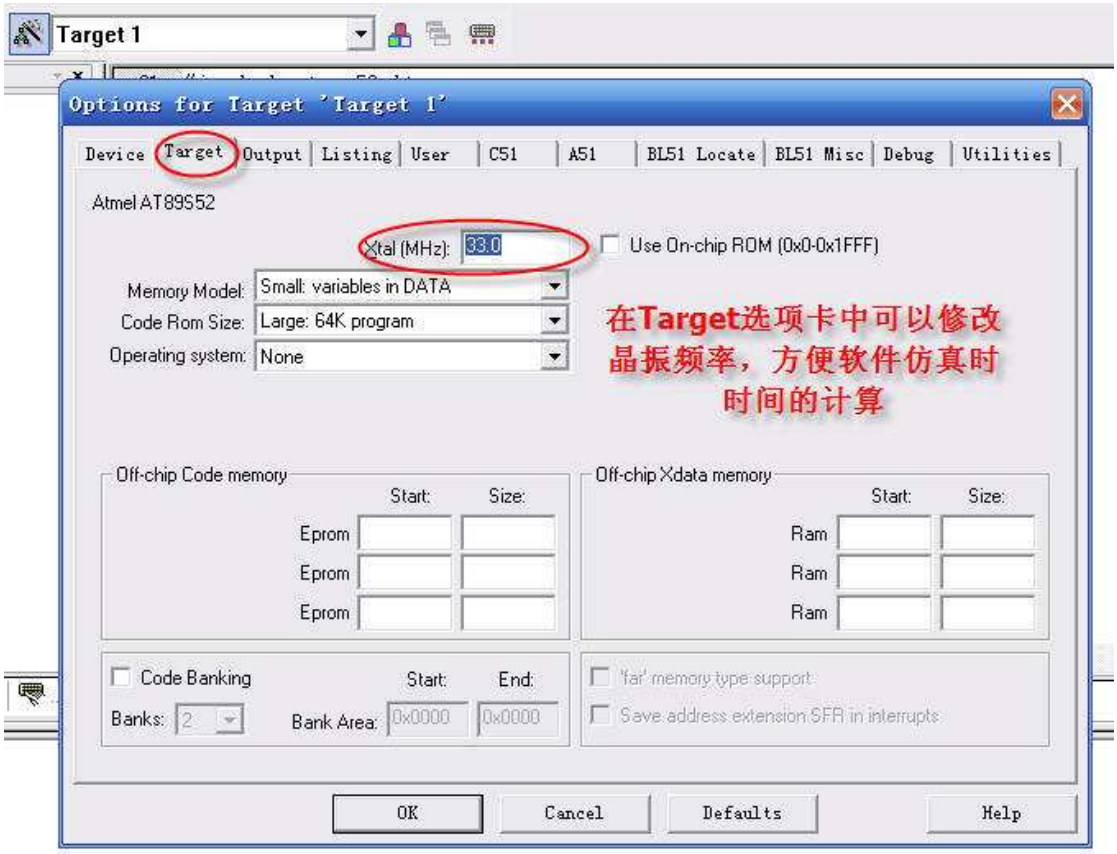

Save As
保存在(I):
src
我最近的文档
桌面
我的文档
我的电脑
网上邻居
给文件取一个名字为main. c, 然后单击保存按钮
文件名(N):
main.c
保存(S)
保存类型(T):
All Files (*.*)
取消

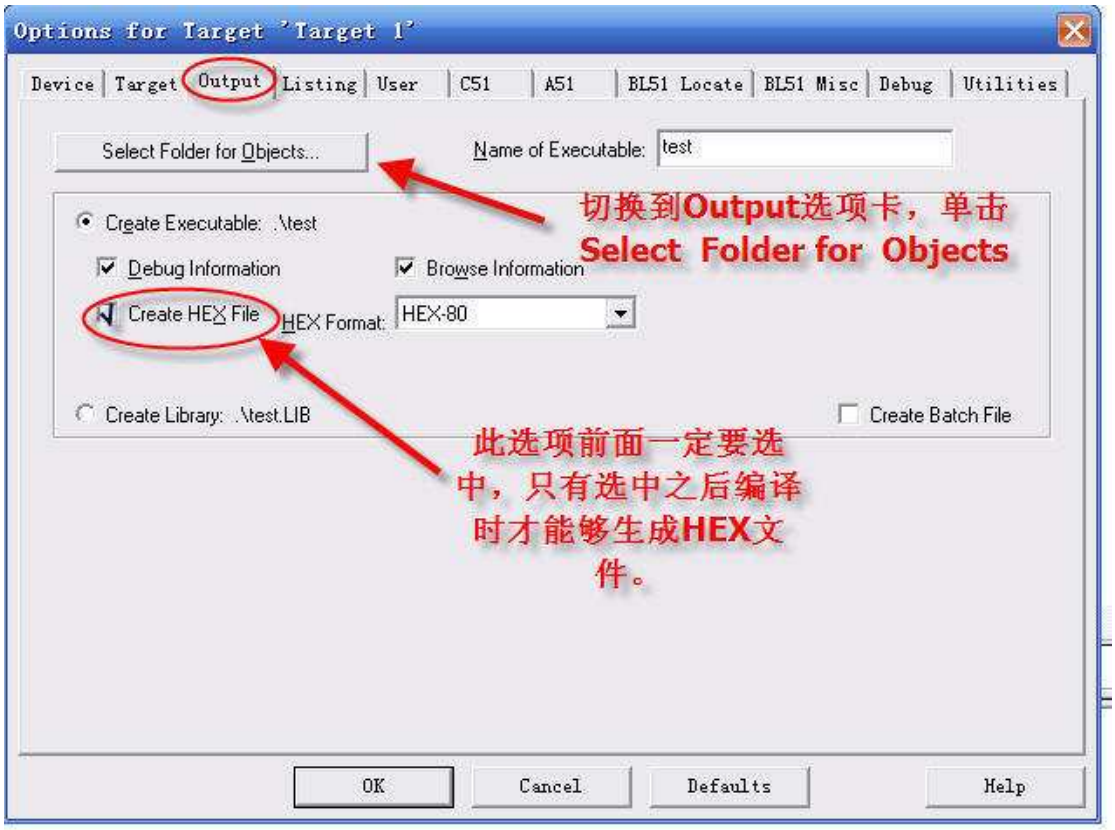

File Edit View Project Debug Flash Peripherals Tools SVCS Window Help
Target 1
Project Workspace
Target 1
Source Group 1
STARTUP.A
在Source Group1上单击鼠标右键，在弹出的菜单中选择Add Files to Group 'Source Group 1'
Options for Group 'So
Open List File
Open Map File
Open File
Rebuild all target files
Build target
Translate File
Stop build
New Gro
Add Files to Group 'Source Group 1'
Manage Components
Remove Group 'Source Group 1' and its Files
Include Dependencies

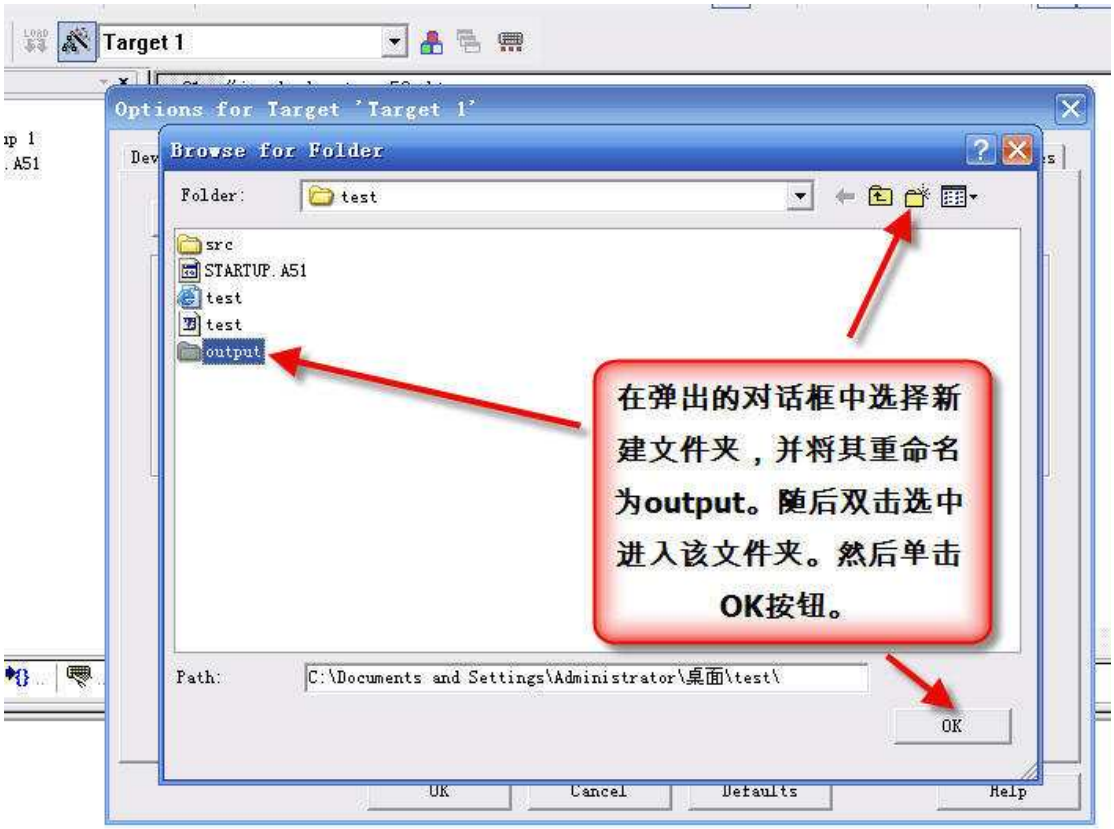

Add Files to Group 'Source Group 1'
查找范围(I):
src
main.c
选中我们之前新建的src文件夹，然后选中main.c文件，单击Add按钮后，再单击Close按钮
文件名(N):
main
文件类型(T):
C Source file (*.c)
Add
Close

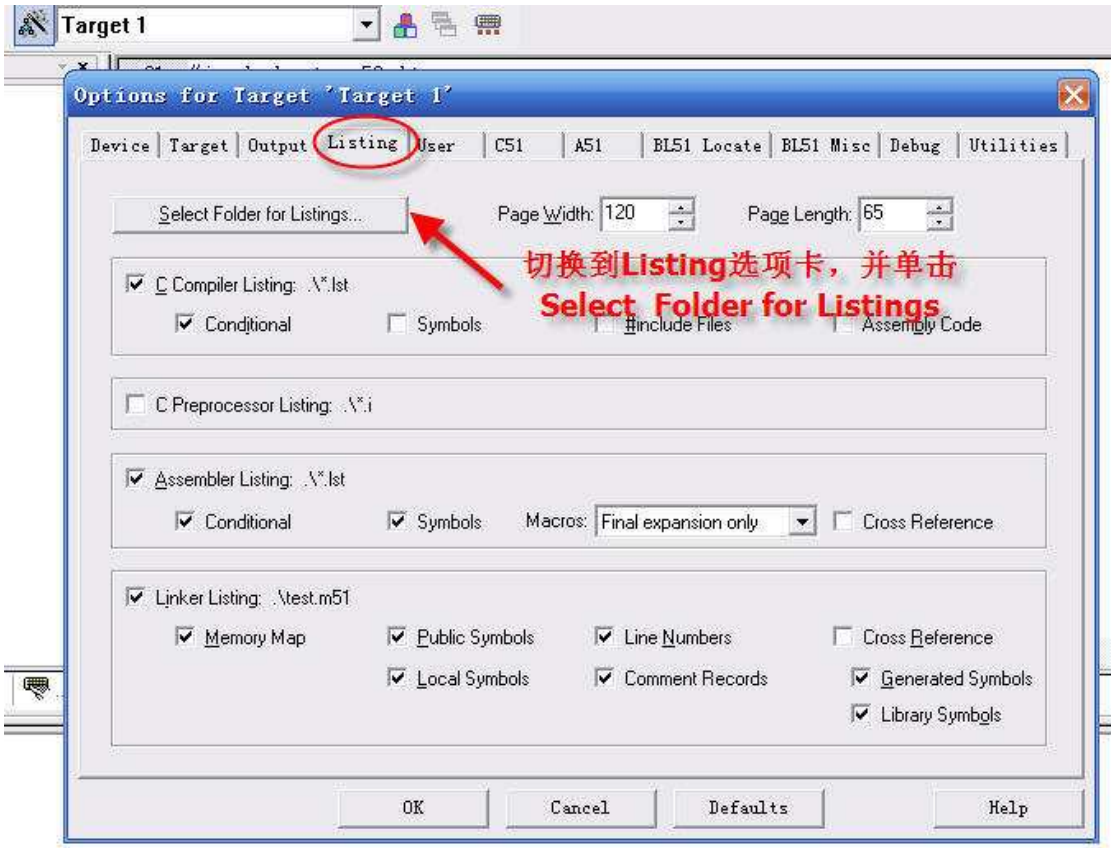

File Edit View Project Debug Flash Peripherals Tools SVCS Window Help
Target 1
Project Workspace
Target 1
Source Group 1
STARTUP.A51
main.c
01 #include <reg52.h>
02
03
04 void main(void)
05 {
06 while(1)
07 {
08
09 }
10 }
在main.c文件中编写我们的主函数代码

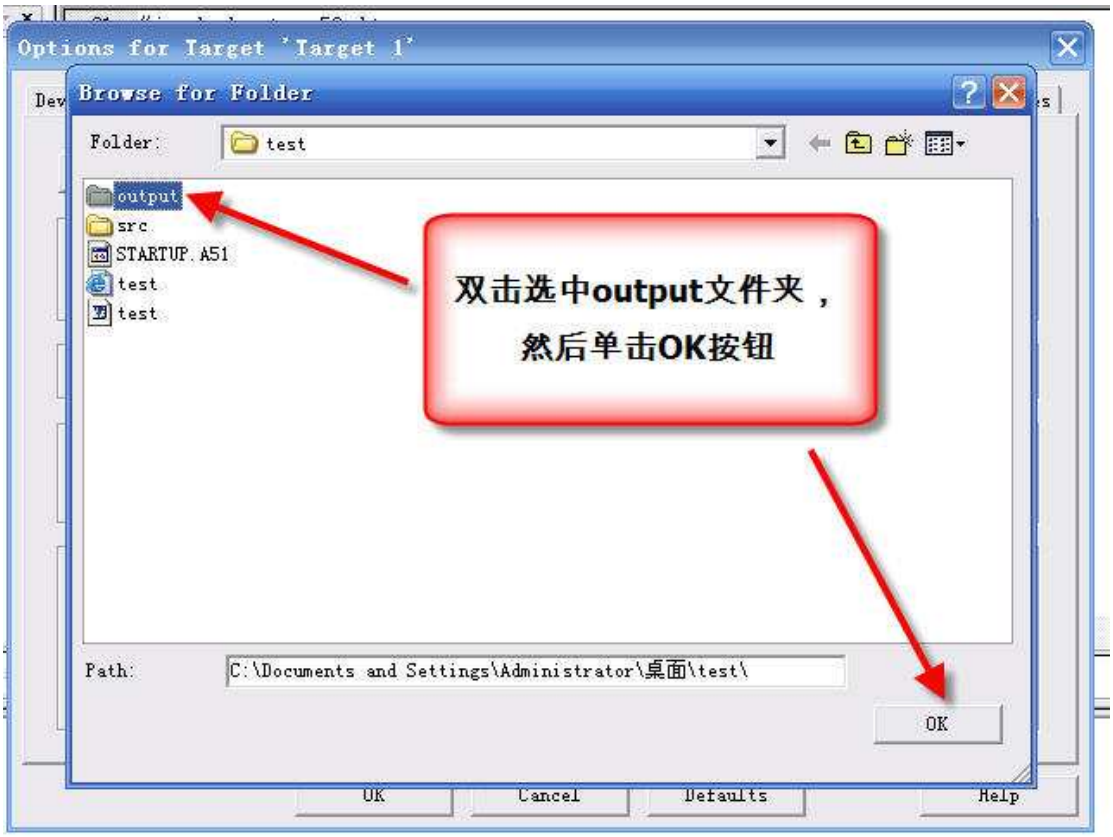

File Edit View Project Debug Flash Peripherals Tools SVCS Window Help
Target 1
Project Workspace
Options for Target
Target 1
Source Group 1
STARTUP.A51
main.c
#include <reg52.h>
void main(void)
{
while(1)
{
}
}
代码编写完毕后，单击此按钮

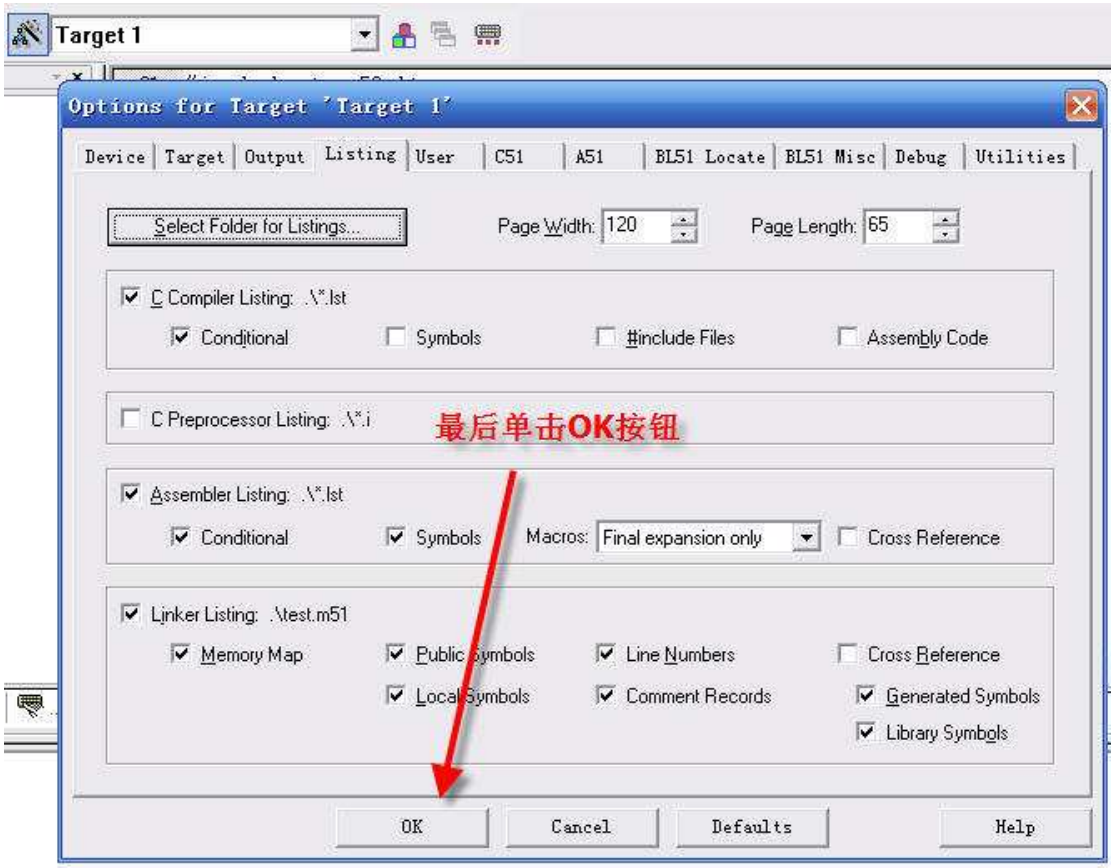

Target 1
Options for Target 'Target 1'
Device Target Output Listing User C51 A51 BL51 Locate BL51 Misc Debug Utilities
Atmel AT89S52
Xtal (MHz): 33.0
Use On-chip ROM (0x0-0x1FFF)
Memory Model: Small: variables in DATA
Code Rom Size: Large: 64K program
Operating system: None
在Target选项卡中可以修改晶振频率，方便软件仿真时时间的计算
Off-chip Code memory
Start: Size:
Eprom
Eprom
Eprom
Off-chip Xdata memory
Start: Size:
Ram
Ram
Ram
Code Banking
Start: End:
Banks: 2
Bank Area: 0x0000 0x0000
'far' memory type support
Save address extension SFR in interrupts
OK
Cancel
Defaults
Help

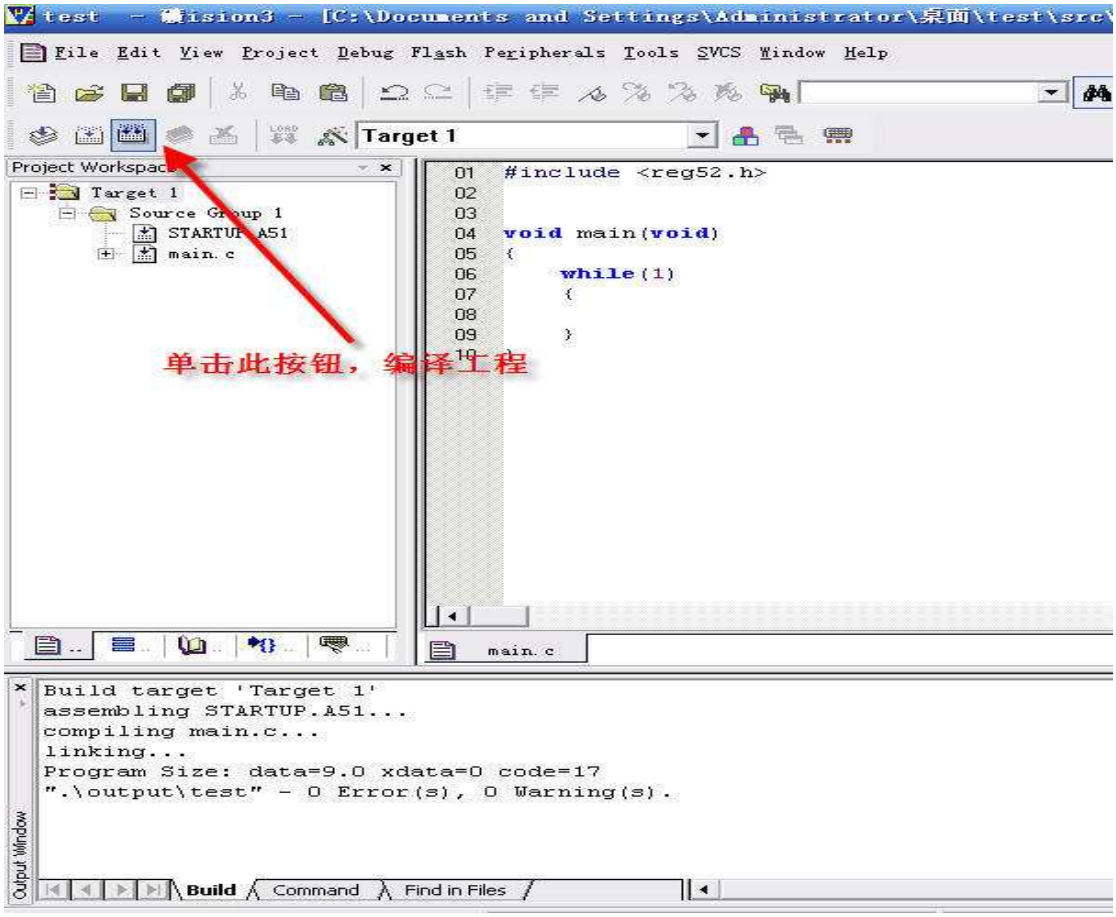

Options for Target 'Target 1'
Device
Target
Output
Listing
User
C51
A51
BL51 Locate
BL51 Misc
Debug
Utilities
Select Folder for Objects...
Name of Executable: test
Create Executable: .\test
Debug Information
Browse Information
Create HEX File
HEX Format: HEX-80
Create Library: .\test.LIB
Create Batch File
OK
Cancel
Defaults
Help
切换到Output选项卡，单击Select Folder for Objects
此选项前面一定要选中，只有选中之后编译时才能够生成HEX文件。


Target 1
Options for Target 'Target 1'
Browse for Folder
Folder: test
src
STARTUP.A51
test
test
output
在弹出的对话框中选择新建文件夹，并将其重命名为output。随后双击选中进入该文件夹。然后单击OK按钮。
Path: C:\Documents and Settings\Administrator\桌面\test\
OK
OK
Cancel
Defaults
Help


Target 1
Options for Target 'Target 1'
Device
Target
Output
Listing
User
C51
A51
BL51 Locate
BL51 Misc
Debug
Utilities
Select Folder for Listings...
Page Width: 120
Page Length: 65
切换到Listing选项卡，并单击
Select Folder for Listings
C Compiler Listing: .*.lst
Conditional
Symbols
#Include Files
Assembly Code
C Preprocessor Listing: .*.i
Assembler Listing: .*.lst
Conditional
Symbols
Macros: Final expansion only
Cross Reference
Linker Listing: .\test.m51
Memory Map
Public Symbols
Line Numbers
Cross Reference
Local Symbols
Comment Records
Generated Symbols
Library Symbols
OK
Cancel
Defaults
Help


Options for Target 'Target 1'
Browse for Folder
Folder:
test
output
src
STARTUP.A51
test
test
双击选中output文件夹，
然后单击OK按钮
Path:
C:\Documents and Settings\Administrator\桌面\test\
OK
OK
Cancel
Defaults
Help


Target 1
Options for Target 'Target 1'
Device
Target
Output
Listing
User
C51
A51
BL51 Locate
BL51 Misc
Debug
Utilities
Select Folder for Listings...
Page Width: 120
Page Length: 65
C Compiler Listing: .*.lst
Conditional
Symbols
#include Files
Assembly Code
C Preprocessor Listing: .*.i
最后单击OK按钮
Assembler Listing: .*.lst
Conditional
Symbols
Macros: Final expansion only
Cross Reference
Linker Listing: .\test.m51
Memory Map
Public Symbols
Line Numbers
Cross Reference
Local Symbols
Comment Records
Generated Symbols
Library Symbols
OK
Cancel
Defaults
Help


File Edit View Project Debug Flash Peripherals Tools SVCS Window Help
Target 1
Project Workspace
Target 1
Source Group 1
STARTUP.A51
main.c
单击此按钮，编译工程
#include <reg52.h>
void main(void)
{
while(1)
{
}
main.c
Build target 'Target 1'
assembling STARTUP.A51...
compiling main.c...
linking...
Program Size: data=9.0 xdata=0 code=17
".\output\test" - 0 Error(s), 0 Warning(s).
Output Window
Build
Command
Find in Files

OK ，到此一个简单的工程模板就建立起来了，以后我们再新建源文件和头文件的时候，就可以直接保存到 src 文件目录下面了。

下面我们开始编写各个模块文件。

首先编写 Timer.c 这个文件主要内容就是定时器初始化，以及定时器中断服务函数。其内容如下。

```
#include <reg52.h>

bit g_bSystemTime1Ms = 0 ;                  // 1MS 系统时标


void Timer0Init(void)
{
    TMOD &= 0xf0 ;
    TMOD |= 0x01 ;      //定时器 0 工作方式 1
    TH0   =   0xfc ;      //定时器初始值
    TL0   =   0x66 ;
    TR0   = 1 ;
    ET0   = 1 ;
}

void Time0Isr(void) interrupt 1
{
    TH0   =   0xfc ;                //定时器重新赋初值
    TL0   =   0x66 ;
    g_bSystemTime1Ms = 1 ;      //1MS 时标标志位置位
}
```

由于在 Led.c 文件中需要调用我们的 g_bSystemTime1Ms 变量。同时主函数需要调用 Timer0Init()初始化函数，所以应该对这个变量和函数在头文件里作外部声明。以方便其它函数调用。

Timer.h 内容如下。

```
#ifndef _TIMER_H_
#define _TIMER_H_

extern void Timer0Init(void) ;
extern bit g_bSystemTime1Ms ;

#endif
```

完成了定时器模块后，我们开始编写 LED 驱动模块。
Led.c 内容如下：

```
#include <reg52.h>
#include "MacroAndConst.h"
#include "Led.h"
#include "Timer.h"

static uint16   g_u16LedTimeCount = 0 ; //LED 计数器
static uint8    g_u8LedState = 0 ;       //LED 状态标志, 0 表示亮，1 表示熄灭

#define LED P0                  //定义 LED 接口
#define LED_ON()        LED = 0x00 ;   //所有 LED 亮
#define LED_OFF()       LED = 0xff ;   //所有 LED 熄灭

void LedProcess(void)
{
    if(0 == g_u8LedState)   //如果 LED 的状态为亮，则点亮 LED
    {
        LED_ON() ;
    }
    else                    //否则熄灭 LED
    {
        LED_OFF() ;
    }
}


void LedStateChange(void)
{
    if(g_bSystemTime1Ms)                //系统 1MS 时标到
    {
        g_bSystemTime1Ms = 0 ;
        g_u16LedTimeCount++ ;           //LED 计数器加一
        if(g_u16LedTimeCount >= 500) //计数达到 500,即 500MS 到了,改变 LED 的状态。
        {
```

```
                g_u16LedTimeCount = 0 ;
                g_u8LedState    = ! g_u8LedState   ;
            }
        }
}
```

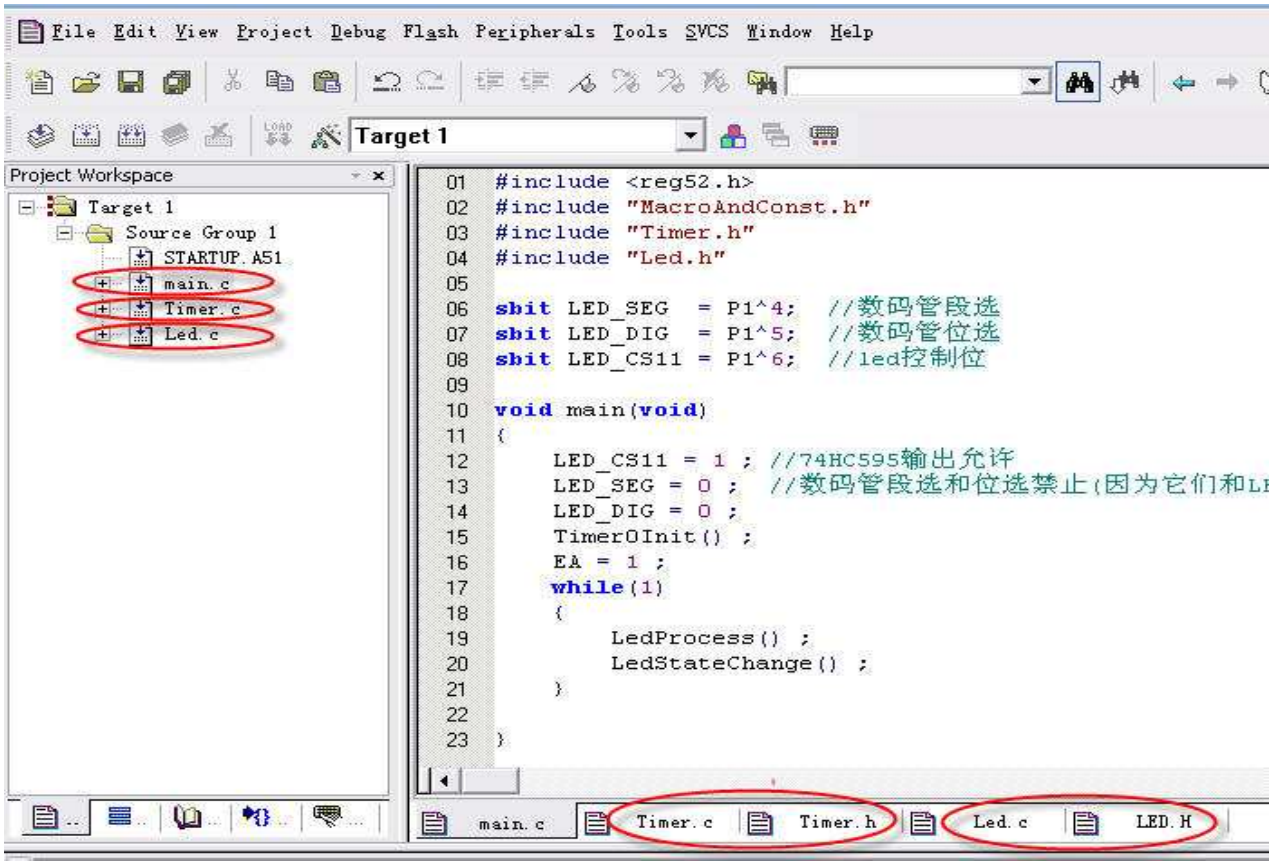
这个模块对外的借口只有两个函数，因此在相应的 Led.h 中需要作相应的声明。
Led.h 内容：

```
#ifndef _LED_H_
#define _LED_H_

extern void LedProcess(void) ;
extern void LedStateChange(void) ;

#endif
```

这两个模块完成后，我们将其 C 文件添加到工程中。然后开始编写主函数里的代码。
如下所示：

```
#include <reg52.h>
#include "MacroAndConst.h"
#include "Timer.h"
#include "Led.h"

sbit LED_SEG    = P1^4;   //数码管段选
sbit LED_DIG    = P1^5;   //数码管位选
sbit LED_CS11 = P1^6;   //led 控制位

void main(void)
{
    LED_CS11 = 1 ; //74HC595 输出允许
    LED_SEG = 0 ;   //数码管段选和位选禁止(因为它们和 LED 共用 P0 口)
    LED_DIG = 0 ;
    Timer0Init() ;
    EA = 1 ;
    while(1)
    {
        LedProcess() ;
        LedStateChange() ;
    }
}
```

整个工程截图如下：



至此，第三章到此结束。
一起来总结一下我们需要注意的地方吧

1. **C 语言源文件(*.c)的作用是什么**
2. **C 语言头文件(*.h)的作用是什么**
3. **typedef 的作用**
4. **工程模板如何组织**
5. **如何创建一个多模块(多文件)的工程**

看着学习板上的 LED 按照我们的意愿开始闪烁起来，你心里是否高兴了，我相信你会的。但是很快你就会感觉到太单调，总是同一个频率在闪烁，总是同一个亮度在闪烁。如果要是能够由暗逐渐变亮，然后再由亮变暗该多漂亮啊。嗯，想法不错，可以该从什么地方入手呢。

在开始我们的工程之前，首先来了解一个概念：PWM。

PWM(Pulse Width Modulation)是脉冲宽度调制的英文单词的缩写。下面这段话是通信百科中对其的定义：

脉冲宽度调制(PWM)是利用微处理器的数字输出来对模拟电路进行控制的一种非常有效的技术，广泛应用在从测量、通信到功率控制与变换的许多领域中。脉宽调制是开关型稳压电源中的术语。这是按稳压的控制方式分类的，除了 PWM 型，还有 PFM 型和 PWM、PFM 混合型。脉宽调制式开关型稳压电路是在控制电路输出频率不变的情况下，通过电压反馈调整其占空比，从而达到稳定输出电压的目的。

读起来有点晦涩难懂。其实简单的说来，PWM 技术就是通过调整一个周期固定的方波的占空比，来调节输出电压的平均当电压，电流或者功率等被控量。我们可以用一个水龙头来类比，把 1S 时间分成 50 等份，即每一个等份 20MS。在这 20MS 时间里如果我们把水龙头水阀一直打开，那么在这 20MS 里流过的水肯定是最多的，如果我们把水阀打开 15MS，剩下的 5MS 关闭水阀，那么流出的水相比刚才 20MS 全开肯定要小的多。同样的道理，我们可以通过控制 20MS 时间里水阀开启的时间的长短来控制流过的水的多少。那么在 1S 内平均流出的水流量也就可以被控制了。

当我们调整 PWM 的占空比时，就会引起电压或者电流的改变，LED 的明暗状态就会随之发生相应的变化，听起来好像可以通过这种方法来实现我们想要的渐明渐暗的效果。让我们来试一下吧。

大家都知道人眼有一个临界频率，当 LED 的闪烁频率达到一定的时候，人眼就分辨不出 LED 是否在闪烁了。就像我们平常看电视一样，看起来画面是连续的，实质不是这个样子，所有连续动作都是一帧帧静止的画面在 1S 的时间里快速播放出来，譬如每秒 24 帧的速度播放，由于人眼的视觉暂留效应，看起来画面就是连续的了。同样的道理，为了让我们的 LED 在变化的过程中，我们感觉不到其在闪烁，可以将其闪烁的频率定在 50Hz 以上。同时为了看起来明暗过渡的效果更加明显，我们在这里定义其变化范围为 0~99（100 等分）. 即最亮的时候其灰度等级为 99，为 0 的时候最暗，也就是熄灭了。

于是乎我们定义 PWM 的占空比上限为 99, 下限定义为 0

```
#define   LED_PWM_LIMIT_MAX        99
#define   LED_PWM_LIMIT_MIN         0
```

假定我们 LED 的闪烁频率为 50HZ，而亮度变化的范围为 0~99 共 100 等分。则每一等分所占用的时间为 1/(50*100) = 200us 即我们在改变 LED 的亮灭状态时，应该是在 200us 整数倍时刻时。在这里我们用单片机的定时器产生 200us 的中断，同时每 20MS 调整一次 LED 的占空比。这样在 20MS * 100 = 2S 的时间内 LED 可以从暗逐渐变亮，在下一个 2S 内可以从亮逐渐变暗，然后不断循环。

由于大部分的内容都可以在中断中完成，因此，我们的大部分代码都在 Timer.c 这个文件中编写，主函数中除了初始化之外，就是一个空的死循环。

Timer.c 内容如下。

```
#include <reg52.h>
#include "MacroAndConst.h"

#define LED P0              //定义 LED 接口
#define LED_ON()         LED = 0x00 ;   //所有 LED 亮
#define LED_OFF()        LED = 0xff ;   //所有 LED 熄灭

#define   LED_PWM_LIMIT_MAX        99
#define   LED_PWM_LIMIT_MIN         0

static uint8 s_u8TimeCounter = 0 ; //中断计数
static uint8 s_u8LedDirection = 0 ; //LED 方向控制 0 ： 渐亮 1 ： 渐灭
static int8   s_s8LedPWMCounter   = 0 ; //LED 占空比
void Timer0Init(void)
{
    TMOD &= 0xf0 ;
    TMOD |= 0x01 ;      //定时器 0 工作方式 1
    TH0   =  0xff ;        //定时器初始值(200us 中断一次)
    TL0   =   0x47 ;
    TR0   = 1 ;
    ET0   = 1 ;
}

void Time0Isr(void) interrupt 1
{
    static int8 s_s8PWMCounter = 0   ;
    TH0   =  0xff ;                  //定时器重新赋初值
    TL0   =   0x47 ;

    if(++s_u8TimeCounter >= 100) //每 20MS 调整一下 LED 的占空比
    {
        s_u8TimeCounter = 0 ;
        //如果是渐亮方向变化,则占空比递增
        if((s_s8LedPWMCounter <= LED_PWM_LIMIT_MAX)
                                        &&(0 == s_u8LedDirection))
        {
            s_s8LedPWMCounter++   ;
            if(s_s8LedPWMCounter > LED_PWM_LIMIT_MAX)
            {
```

```
                    s_u8LedDirection = 1 ;
                    s_s8LedPWMCounter =   LED_PWM_LIMIT_MAX ;

                }
            }
            //如果是渐暗方向变化,则占空比递渐
        if((s_s8LedPWMCounter >= LED_PWM_LIMIT_MIN)
                                &&(1 == s_u8LedDirection))
            {
                s_s8LedPWMCounter-- ;
                if(s_s8LedPWMCounter < LED_PWM_LIMIT_MIN)
                {
                    s_u8LedDirection = 0 ;
                    s_s8LedPWMCounter =   LED_PWM_LIMIT_MIN ;

                }
            }
            s_s8PWMCounter = s_s8LedPWMCounter ;   //获取 LED 的占空比
        }

        if(s_s8PWMCounter > 0) //占空比大于 0,则点亮 LED,否则熄灭 LED
        {
            LED_ON() ;
            s_s8PWMCounter-- ;
        }
        else
        {
            LED_OFF();
        }

    }
```

其实 PWM 技术在我们实际生活中应用的非常多。比较典型的应用就是控制电机的转速，控制充电电流的大小，等等。而随着技术的发展，也出现了其他类型的 PWM 技术，如相电压 PWM，线电压 PWM，SPWM 等等，如果有兴趣可以到网上去获取相应资料学习。

关于渐明渐暗的灯就简单的讲到这里。

数码管在实际应用中非常广泛，尤其是在某些对成本有限制的场合。编写一个好用的LED 程序并不是那么的简单。曾经有人这样说过，如果用数码管和按键，做一个简易的可以调整的时钟出来，那么你的单片机就算入门了 60%了。此话我深信不疑。我遇到过很多单片机的爱好者，他们问我说单片机我已经掌握了，该如何进一步的学习下去呢？我并不急于回答他们的问题，而是问他们：会编写数码管的驱动程序了吧？“嗯”。会编写按键程序了吧？“嗯”。好，我给你出一个小题目，你做一下。用按键和数码管以及单片机定时器实现一个简易的可以调整的时钟，要求如下：

8 位数码管显示，显示格式如下

时-分-秒

XX-XX-XX

要求：系统有四个按键，功能分别是 调整，加，减，确定。在按下调整键时候，显示时的两位数码管以 1 Hz 频率闪烁。如果再次按下调整键，则分开始闪烁，时恢复正常显示，依次循环，直到按下确定键，恢复正常的显示。在数码管闪烁的时候，按下加或者减键可以调整相应的显示内容。按键支持短按，和长按，即短按时，修改的内容每次增加一或者减小一，长按时候以一定速率连续增加或者减少。

结果很多人，很多爱好者一下子都理不清楚思路。其实问题的根源在于没有以工程化的角度去思考程序的编写。很多人在学习数码管编程的时候，都是照着书上或者网上的例子来进行试验。殊不知，这些例子代码仅仅只是具有一个演示性的作用，拿到实际中是很难用的。举一个简单的例子。

下面这段程序是在网上随便搜索到的：

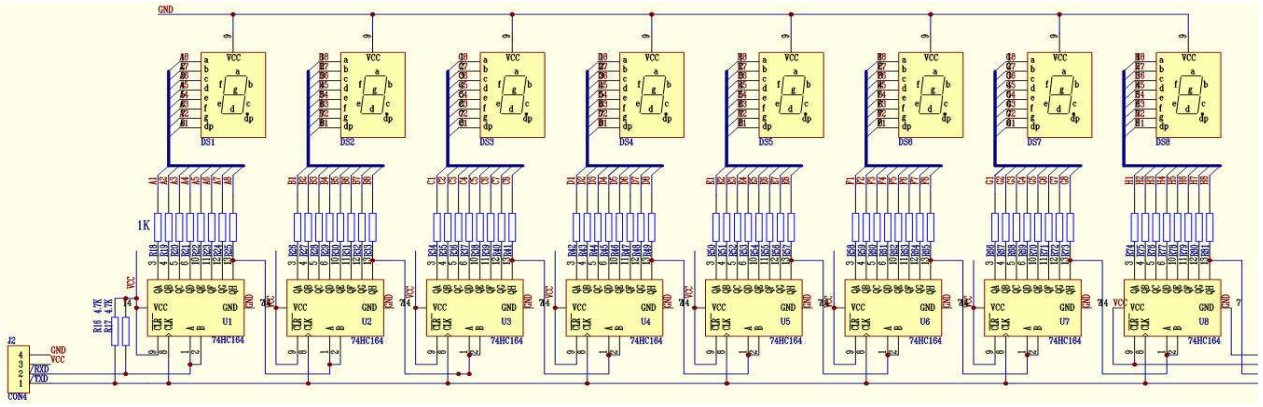
```
while(1)
{
    for(num=0;num<9;num++)
    {
            P0=table[num];
            P2=code[num] ;
            delayms(2) ;
    }
}
```

看出什么问题来了没有，如果没有看出来请仔细想一下，如果还没有想出来，请回过头去，认真再看一遍“学会释放 CPU”这一章的内容。这个程序作为演示程序是没有什么问题的，但是实际应用的时候，数码管显示的内容经常变化，而且还有很多其它任务需要执行，因此这样的程序在实际中是根本就无法用的，更何况，它这里也调用了 delayms(2)这个函数来延时 2 ms 这更是令我们深恶痛绝☺

本章的内容正是探讨如何解决多任务环境下(不带 OS)的数码管程序设计的编写问题。理解了其中的思想，无论要求我们显示的形式怎么变化(如数码管闪烁，移位等),我们都可以很方便的解决问题。

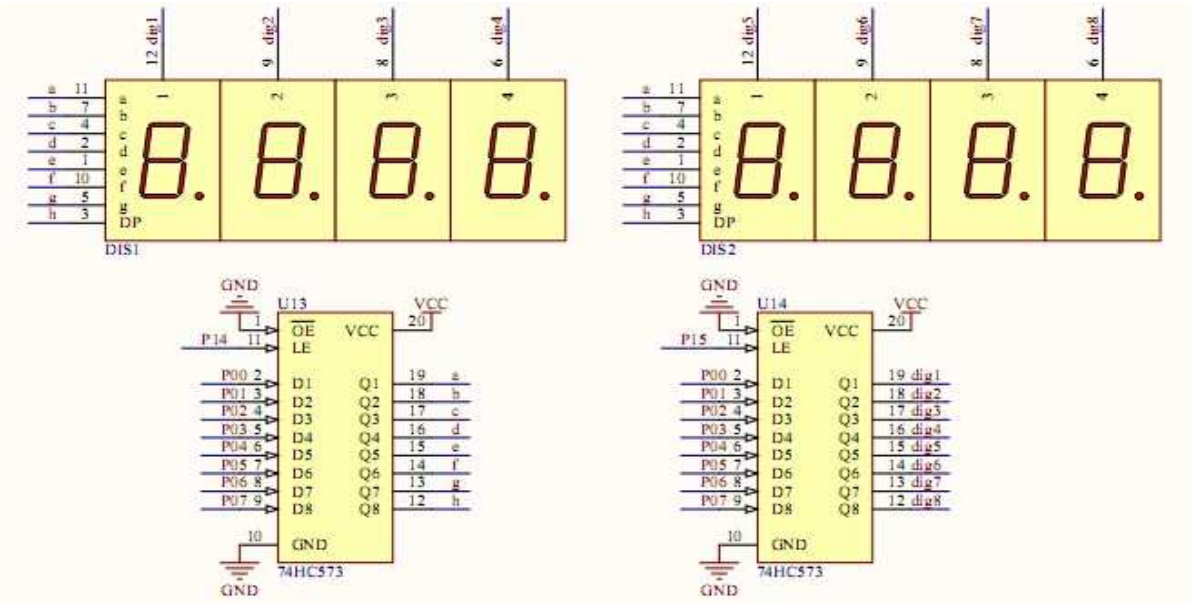
数码管的显示分为动态显示和静态显示两种。静态显示是每一位数码管都用一片独立的驱动芯片进行驱动。比较常见的有 74LS164，74HC595 等。利用这类芯片的好处就是可以级联，留给单片机的接口只需要时钟线，数据线，因此比较节省 I/O 口。如下图所示：

利用 74LS164 级联驱动 8 个单独的数码管



静态显示的优点是程序编写简单。但是由于涉及到的驱动芯片数量比较多，同时考虑到 PCB 的布线等等因素，在低成本要求的开发环境下，单纯的静态驱动并不合适。这个时候就可以考虑到动态驱动了。

动态驱动的图如下所示(以 EE21 开发板为例)



由上图可以看出。8 个数码管的段码由一个单独的 74HC573 驱动。同时每一个数码管的公共端连接在另外一个 74HC573 的输出上。当送出第一位数码管的段码内容时候，同时选通第一位数码管的位选，此时，第一位数码管就显示出相应的内容了。一段时间之后，送出第二位数码管段码的内容，选通第二位数码管的位选，这时显示的内容就变成第二位数码管的内容了……依次循环下去，就可以看到了所有数码管同时显示了。事实上，任意时刻，只有一位数码管是被点亮的。由于人眼的视觉暂留效应以及数码管的余辉效应，当数码管扫描

的频率非常快的时候，人眼已经无法分辨出数码管的变化了，看起来就是同时点亮的。我们假设数码管的扫描频率为 50 Hz, 则完成一轮扫描的时间就是 1 / 50 = 20 ms 。我们的系统共有 8 位数码管，则每一位数码管在一轮扫描周期中点亮的时间为 20 / 8 = 2.5 ms 。

动态扫描对时间要求有一点点严格，否则，就会有明显的闪烁。

假设我们程序 中所有任务如下：

```
while(1)
{
    LedDisplay() ;       //数码管动态扫描
    ADProcess() ;        //AD 采集处理
    TimerProcess() ;   //时间相关处理
    DataProcess() ;    //数据处理
}
```

LedDisplay() 这个任务的执行时间，如同我们刚才计算的那样，50 Hz 频率扫描，则该函数执行的时间为 20 ms 。 假设 ADProcess()这个任务执行的的时间为 2 ms ，TimerProcess() 这个函数执行的时间为 1 ms ，DataProcess() 这个函数执行的时间为 10 ms 。 那么整个主函数执行一遍的总时间为 20 + 2 + 1 + 10 = 33 ms 。即 LedDisplay() 这个函数的扫描频率已经不为 50 Hz 了，而是 1 / 33 = 30.3 Hz 。这个频率数码管已经可以感觉到闪烁了，因此不符合我们的要求。为什么会出现这种情况呢？ 我们刚才计算的 50 Hz 是系统只有 LedDisplay() 这一个任务的时候得出来的结果。当系统添加了其它任务后，当然系统循环执行一次的总时间就增加了。如何解决这种现象了，还是离不开我们第二章所讲的那个思想。

系统产生一个 2.5 ms 的时标消息。LedDisplay() , 每次接收到这个消息的时候, 扫描一位数码管。这样 8 个时标消息过后，所有的数码管就都被扫描一遍了。可能有朋友会有这样的疑问：ADProcess() 以及 DataProcess() 等函数执行的时间还是需要十几 ms 啊，在这十几 ms 的时间里，已经产生好几个 2.5 ms 的时标消息了，这样岂不是漏掉了扫描，显示起来还是会闪烁。能够想到这一点，很不错，这也就是为什么我们要学会释放 CPU 的原因。对于 ADProcess()，TimerProcess()，DataProcess()，等任务我们依旧要采取此方法对 CPU 进行释放，使其执行的时间尽可能短暂，关于如何做到这一点，在以后的讲解如何设计多任务程序设计的时候会讲解到。

下面我们基于此思路开始编写具体的程序。

首先编写 Timer.c 文件。该文件中主要为系统提供时间相关的服务。必要的头文件包含。

```
#include <reg52.h>
#include "MacroAndConst.h"
```

为了方便计算，我们取数码管扫描一位的时间为 2 ms。设置定时器 0 为 2 ms 中断一次。

同时声明一个位变量，作为 2 ms 时标消息的标志

```
bit g_bSystemTime2Ms = 0 ;                  // 2msLED 动态扫描时标消息
```

初始化定时器 0

```
void Timer0Init(void)
{
    TMOD &= 0xf0 ;
    TMOD |= 0x01 ;        //定时器 0 工作方式 1
    TH0   =   0xf8 ;        //定时器初始值
    TL0   =   0xcc ;
    TR0   = 1 ;
```

```
    ET0   = 1 ;
}
```

在定时器 0 中断处理程序中，设置时标消息。

```
void Time0Isr(void) interrupt 1
{
    TH0   =   0xf8 ;                  //定时器重新赋初值
    TL0   =   0xcc ;
    g_bSystemTime2Ms = 1 ;      //2MS 时标标志位置位
}
```

然后我们开始编写数码管的动态扫描函数。

新建一个 C 源文件，并包含相应的头文件。

```
#include <reg52.h>
#include "MacroAndConst.h"
#include "Timer.h"
```

先开辟一个数码管显示的缓冲区。动态扫描函数负责从这个缓冲区中取出数据，并扫描显示。而其它函数则可以修改该缓冲区，从而改变显示的内容。

```
uint8 g_u8LedDisplayBuffer[8] = {0} ; //显示缓冲区
```

然后定义共阳数码管的段码表以及相应的硬件端口连接。

```
code uint8 g_u8LedDisplayCode[]=
{
    0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,
    0x80,0x90,0x88,0x83,0xC6,0xA1,0x86,0x8E,
    0xbf, //'-'号代码
} ;

sbit io_led_seg_cs = P1^4 ;
sbit io_led_bit_cs = P1^5 ;

#define LED_PORT P0
```

再分别编写送数码管段码函数，以及位选通函数。

```
static void SendLedSegData(uint8 dat)
{
    LED_PORT          = dat ;
    io_led_seg_cs = 1 ;          //开段码锁存,送段码数据
    io_led_seg_cs = 0 ;
}

static void SendLedBitData(uint8 dat)
{
    uint8 temp ;
    temp = (0x01 << dat ) ;   //根据要选通的位计算出位码
    LED_PORT          = temp ;
    io_led_bit_cs = 1 ;            //开位码锁存,送位码数据
```

```
    io_led_bit_cs = 0 ;
}
```

下面的核心就是如何编写动态扫描函数了。
如下所示：

```
void LedDisplay(uint8 * pBuffer)
{
    static uint8 s_LedDisPos = 0 ;
    if(g_bSystemTime2Ms)
    {
        g_bSystemTime2Ms = 0 ;

        SendLedBitData(8) ;              //消隐，只需要设置位选不为 0~7 即可

        if(pBuffer[s_LedDisPos] == '-')    //显示'-'号
        {
            SendLedSegData(g_u8LedDisplayCode[16]) ;
        }
        else
        {
            SendLedSegData(g_u8LedDisplayCode[pBuffer[s_LedDisPos]]) ;
        }

        SendLedBitData(s_LedDisPos);

        if(++s_LedDisPos > 7)
        {
            s_LedDisPos = 0 ;
        }
    }
}
```

函数内部定义一个静态的变量 s_LedDisPos，用来表示扫描数码管的位置。每当我们执行该函数一次的时候，s_LedDisPos 的值会自加 1，表示下次扫描下一个数码管。然后判断 g_bSystemTime2Ms 时标消息是否到了。如果到了，就开始执行相关扫描，否则就直接跳出函数。SendLedBitData(8) ;的作用是消隐。因为我们的系统的段选和位选是共用 P0 口的。在送段码之前，必须先关掉位选，否则，因为上次位选是选通的，在送段码的时候会造成相应数码管的点亮，尽管这个时间很短暂。但是因为我们的数码管是不断扫描的，所以看起来还是会有些微微亮。为了消除这种影响，就有必要再送段码数据之前关掉位选。

if(pBuffer[s_LedDisPos] == '-')   //显示'-'号这行语句是为了显示’-’符号特意加上去的，大家可以看到在定义数码管的段码表的时候，我多加了一个字节的代码 0xbf：

```
code uint8 g_u8LedDisplayCode[]=
```

```
{
    0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,
    0x80,0x90,0x88,0x83,0xC6,0xA1,0x86,0x8E,
    0xbf, //'-'号代码
};
```

通过 `SendLedSegData(g_u8LedDisplayCode[pBuffer[s_LedDisPos]]) ;`送出相应的段码数据后，然后通过 `SendLedBitData(s_LedDisPos);`打开相应的位选。这样对应的数码管就被点亮了。

```
if(++s_LedDisPos > 7)
{
    s_LedDisPos = 0 ;
}
```

然后 `s_LedDisPos` 自加 1，以便下次执行本函数时，扫描下一个数码管。因为我们的系统共有 8 个数码管，所以当 `s_LedDisPos > 7` 后，要对其进行清 0 。否则，没有任何一个数码管被选中。这也是为什么我们可以用

```
SendLedBitData(8) ;              //消隐，只需要设置位选不为 0~7 即可
```

对数码管进行消隐操作的原因。

下面我们来编写相应的主函数，并实现数码管上面类似时钟的效果，如显示 10-20-30 即 10 点 20 分 30 秒。

Main.c

```
#include <reg52.h>
#include "MacroAndConst.h"
#include "Timer.h"
#include "Led7Seg.h"

sbit io_led = P1^6 ;


void main(void)
{
    io_led = 0 ;    //发光二极管与数码管共用 P0 口,这里禁止掉发光二极管的锁存输出
    Timer0Init() ;
    g_u8LedDisplayBuffer[0] = 1 ;
    g_u8LedDisplayBuffer[1] = 0 ;
    g_u8LedDisplayBuffer[2] = '-' ;
    g_u8LedDisplayBuffer[3] = 2 ;
    g_u8LedDisplayBuffer[4] = 0 ;
    g_u8LedDisplayBuffer[5] = '-' ;
    g_u8LedDisplayBuffer[6] = 3 ;
    g_u8LedDisplayBuffer[7] = 0 ;
    EA = 1 ;
    while(1)
    {
        LedDisplay(g_u8LedDisplayBuffer) ;
```

```
    }
}
```

将整个工程进行编译，看看效果如何☺

动起来

既然我们想要模拟一个时钟，那么时钟肯定是要走动的，不然还称为什么时钟撒。下面我们在前面的基础之上，添加一点相应的代码，让我们这个时钟走动起来。

我们知道，之前我们以及设置了一个扫描数码管用到的 2 ms 时标。 如果我们再对这个时标进行计数，当计数值达到 500，即 500 * 2 = 1000 ms 时候，即表示已经逝去了 1 S 的时间。我们再根据这个 1 S 的时间更新显示缓冲区即可。听起来很简单，让我们实现它吧。

首先在 Timer.c 中声明如下两个变量：

```
bit g_bTime1S = 0 ;                          //时钟 1S 时标消息
static uint16 s_u16ClockTickCount = 0 ;   //对 2 ms 时标进行计数
```

再在定时器中断函数中添加如下代码：

```
    if(++s_u16ClockTickCount == 500)
    {
        s_u16ClockTickCount = 0 ;
        g_bTime1S = 1 ;
    }
```

从上面可以看出，s_u16ClockTickCount 计数值达到 500 的时候，g_bTime1S 时标消息产生。然后我们根据这个时标消息刷新数码管显示缓冲区：

```
void RunClock(void)
{
    if(g_bTime1S )
    {
        g_bTime1S = 0 ;
        if(++g_u8LedDisplayBuffer[7] == 10)
        {
            g_u8LedDisplayBuffer[7] = 0 ;
            if(++g_u8LedDisplayBuffer[6] == 6)
            {
                g_u8LedDisplayBuffer[6] = 0 ;
                if(++g_u8LedDisplayBuffer[4] == 10)
                {
                    g_u8LedDisplayBuffer[4]= 0 ;
                    if(++g_u8LedDisplayBuffer[3] == 6)
                    {
                        g_u8LedDisplayBuffer[3] = 0 ;
                        if(    g_u8LedDisplayBuffer[0]<2)
```

```
                {
                    if(++g_u8LedDisplayBuffer[1]==10)
                    {
                        g_u8LedDisplayBuffer[1] = 0 ;
                        g_u8LedDisplayBuffer[0]++;
                    }
                }
                else
                {
                    if(++g_u8LedDisplayBuffer[1]==4)
                    {
                        g_u8LedDisplayBuffer[1] = 0 ;
                        g_u8LedDisplayBuffer[0] = 0 ;
                    }
                }
            }
        }
    }

    }
}
```

这个函数的作用就是对每个数码管缓冲位的值进行判断，判断的标准就是我们熟知的24小时制。如秒的个位到了10就清0，同时秒的十位加1….诸如此类，我就不一一详述了。

同时，我们再编写一个时钟初始值设置函数，这样，可以很方便的在主程序开始的时候修改时钟初始值。

```
void SetClock(uint8 nHour, uint8 nMinute, uint8 nSecond)
{
    g_u8LedDisplayBuffer[0] = nHour / 10 ;
    g_u8LedDisplayBuffer[1] = nHour % 10 ;
    g_u8LedDisplayBuffer[2] = '-' ;
    g_u8LedDisplayBuffer[3] = nMinute / 10 ;
    g_u8LedDisplayBuffer[4] = nMinute % 10 ;
    g_u8LedDisplayBuffer[5] = '-' ;
    g_u8LedDisplayBuffer[6] = nSecond / 10 ;
    g_u8LedDisplayBuffer[7] = nSecond % 10 ;
}
```

然后修改下我们的主函数如下：

```
void main(void)
{
    io_led = 0 ;    //发光二极管与数码管共用P0口,这里禁止掉发光二极管的锁存输出
    Timer0Init() ;
    SetClock(10,20,30) ;  //设置初始时间为10点20分30秒
    EA = 1 ;
```

```
    while(1)
    {
        LedDisplay(g_u8LedDisplayBuffer) ;
        RunClock();
    }
}
```

编译好之后，下载到我们的实验板上，怎么样，一个简单的时钟就这样诞生了。

至此，本章所诉就告一段落了。至于如何完成数码管的闪烁显示，就像本章开头所说的那个数码管时钟的功能，就作为一个思考的问题留给大家思考吧。

同时整个 LED 篇就到此结束了，在以后的文章中，我们将开始学习如何编写实用的按键扫描程序。