“从单片机初学者迈向单片机工程师”之LED主题讨论周第一章----写在前面

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^_^。

。“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。ϑ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

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~@_@~.

“从单片机初学者迈向单片机工程师”之LED主题讨论周第二章----学会释放CPU

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

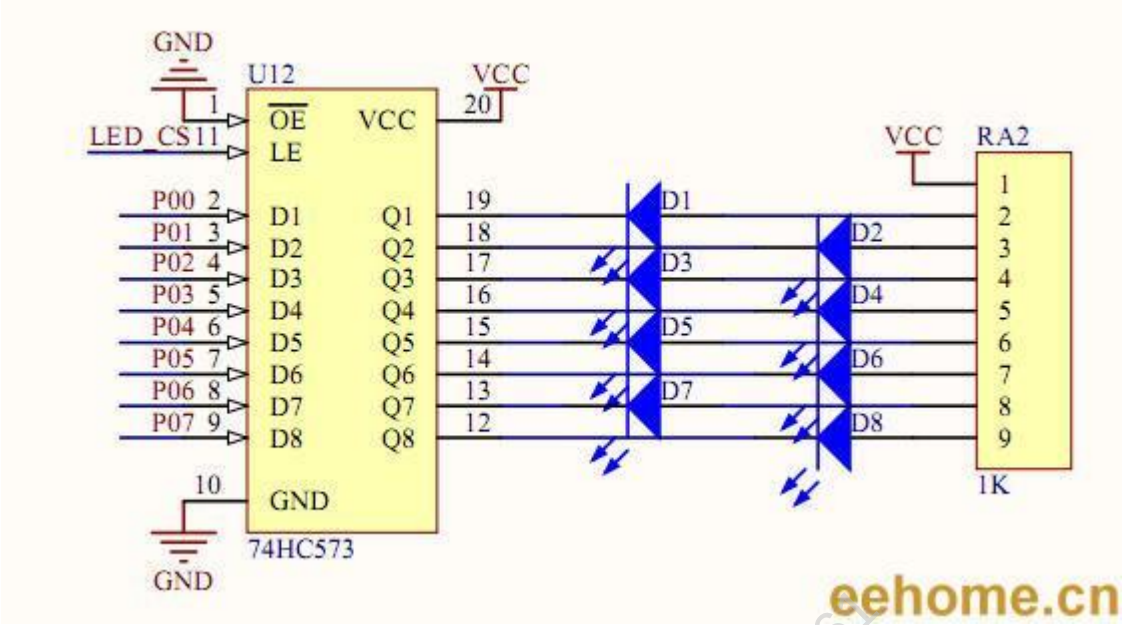
(1) 无论什么时候我们都要以实际应用的角度去考虑程序的编写。

(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)

74HC573 真值表如下：

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

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() ;
    }
}
```

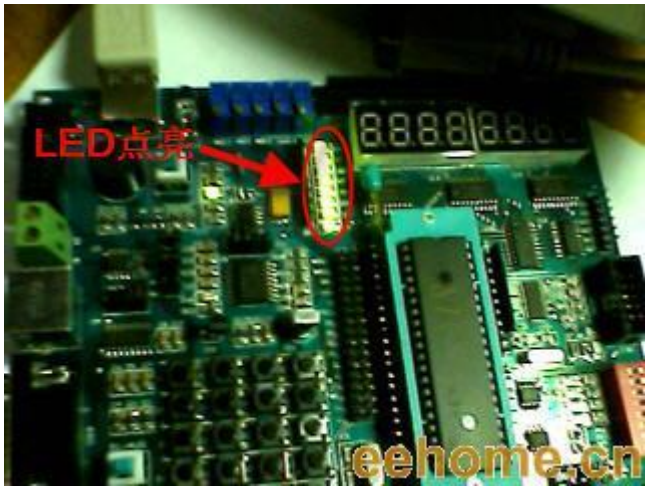
```
void LedStateChange(void)
{
    if(g_bSystemTime1Ms)                //系统 1MS 时标到
  {
            g_bSystemTime1Ms = 0 ;
            g_u16LedTimeCount++ ;       //LED 计数器加一
      if(g_u16LedTimeCount >= 500)   //计数达到 500,即 500MS 到了,改变 LED 的状态。
      {
      g_u16LedTimeCount = 0 ;
      g_u8LedState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

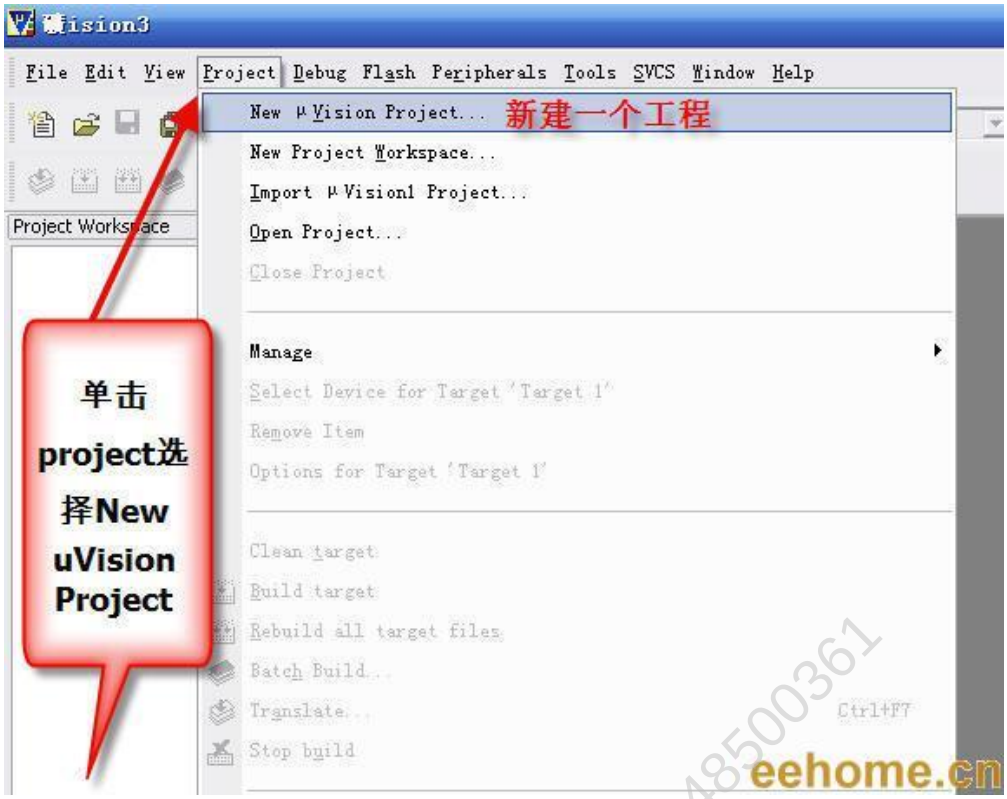
实际效果图如下

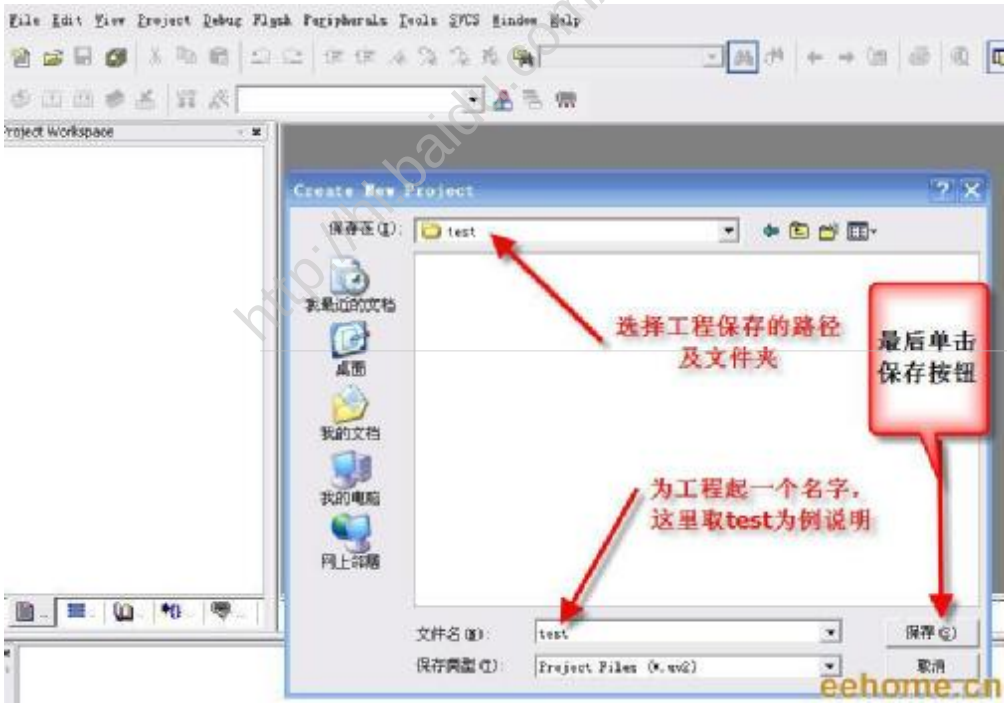
点亮

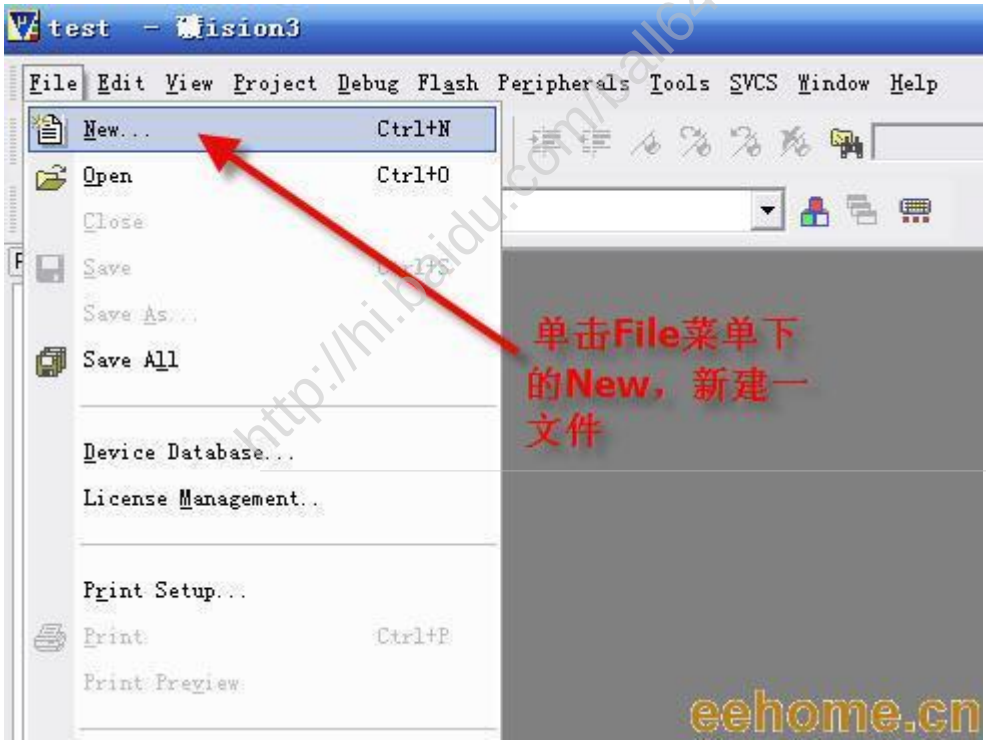


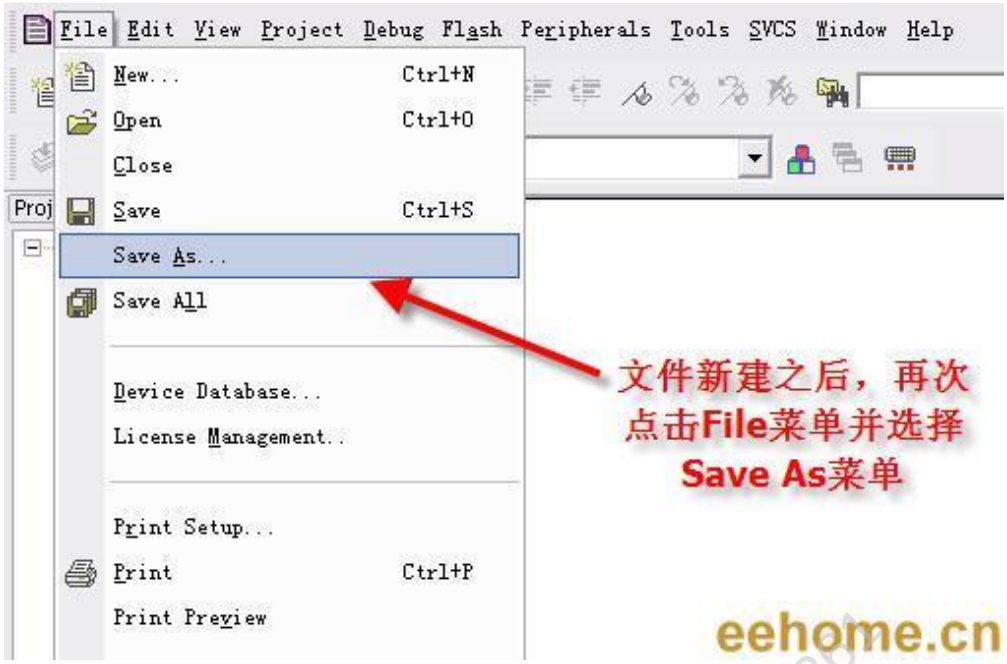
熄灭

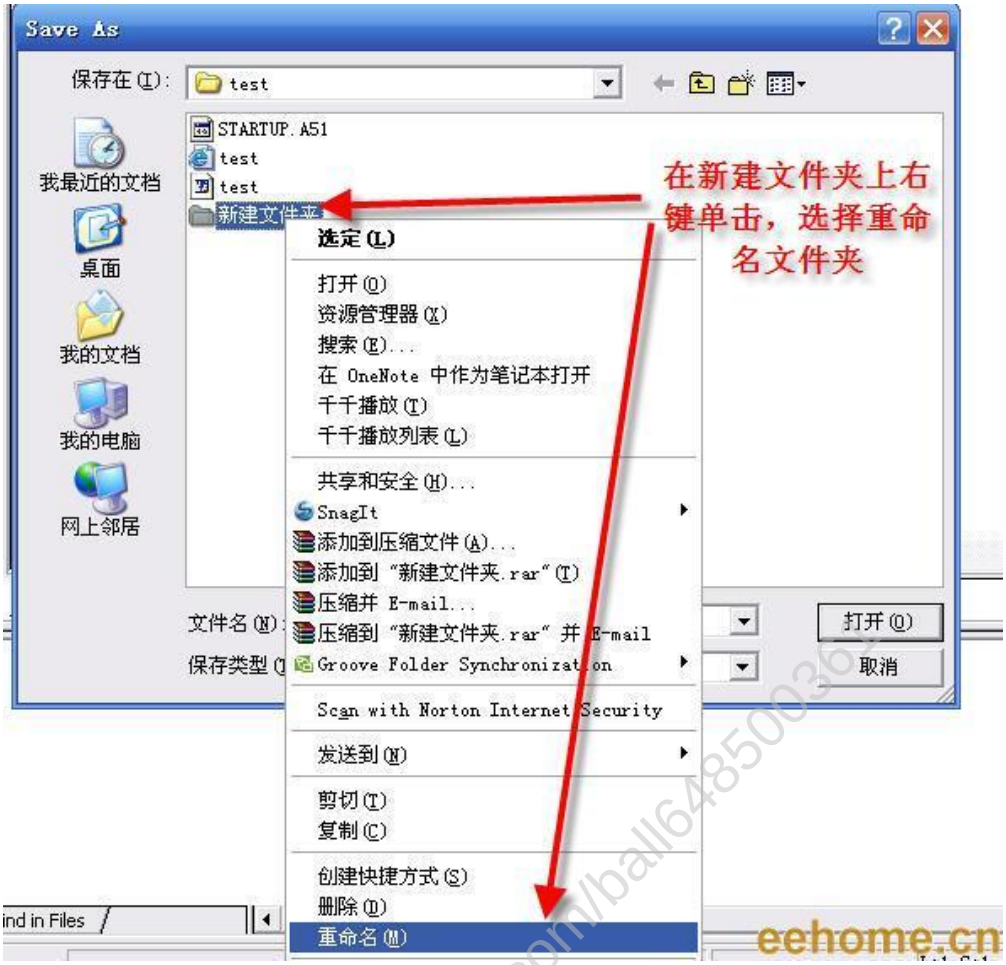
“从单片机初学者迈向单片机工程师”之LED主题讨论周第三章----模块化编程初识

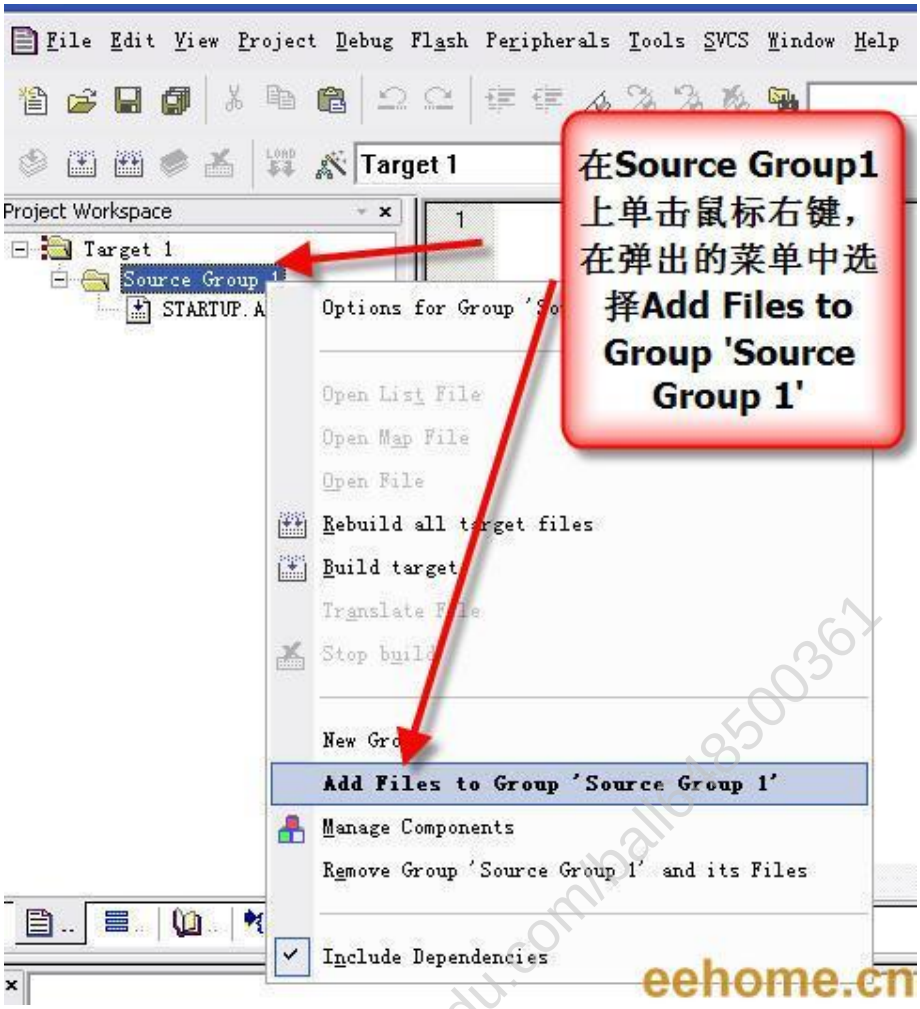


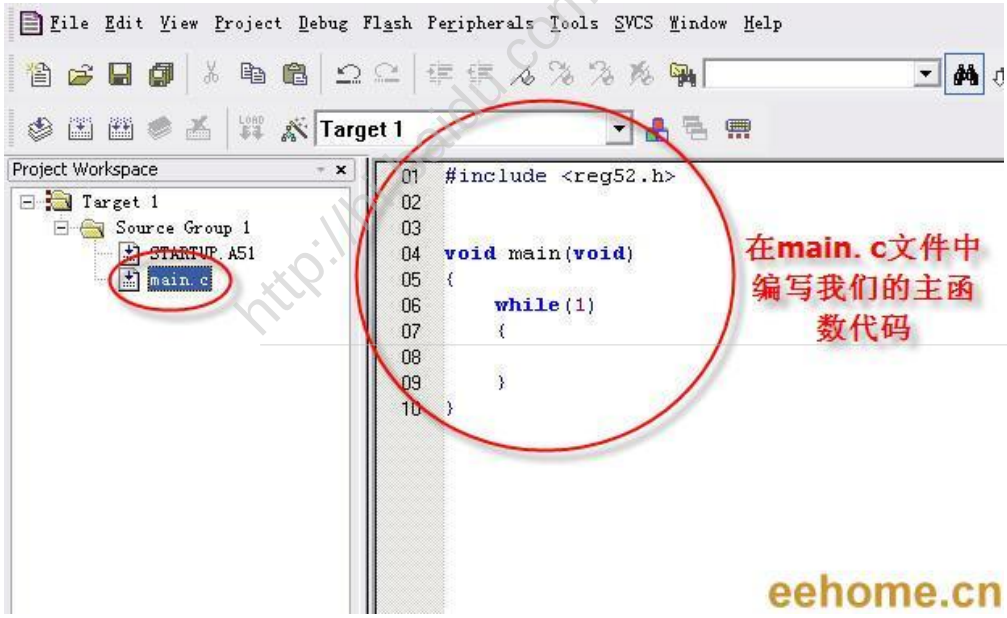


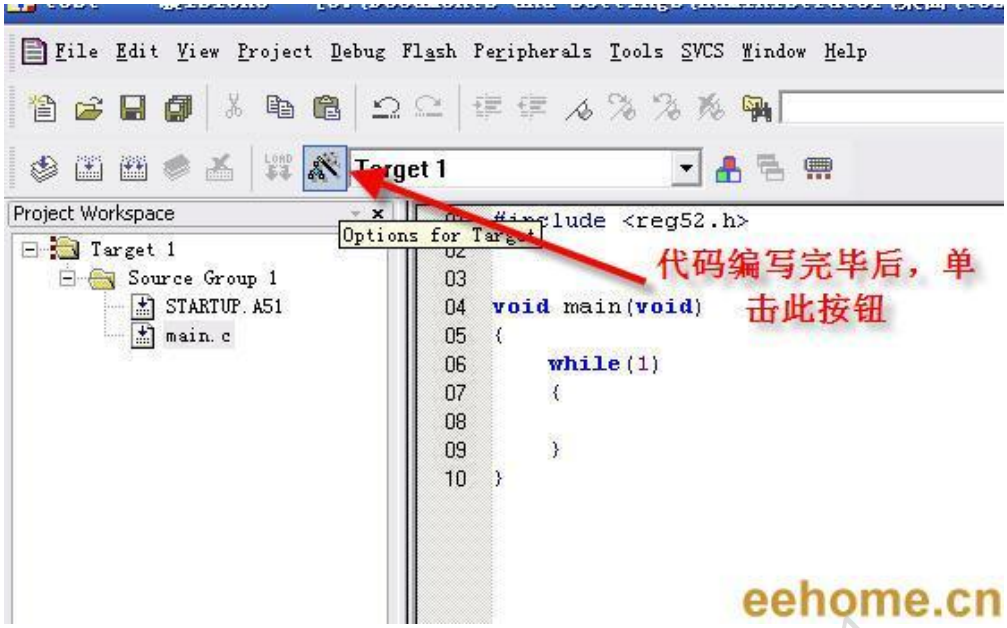


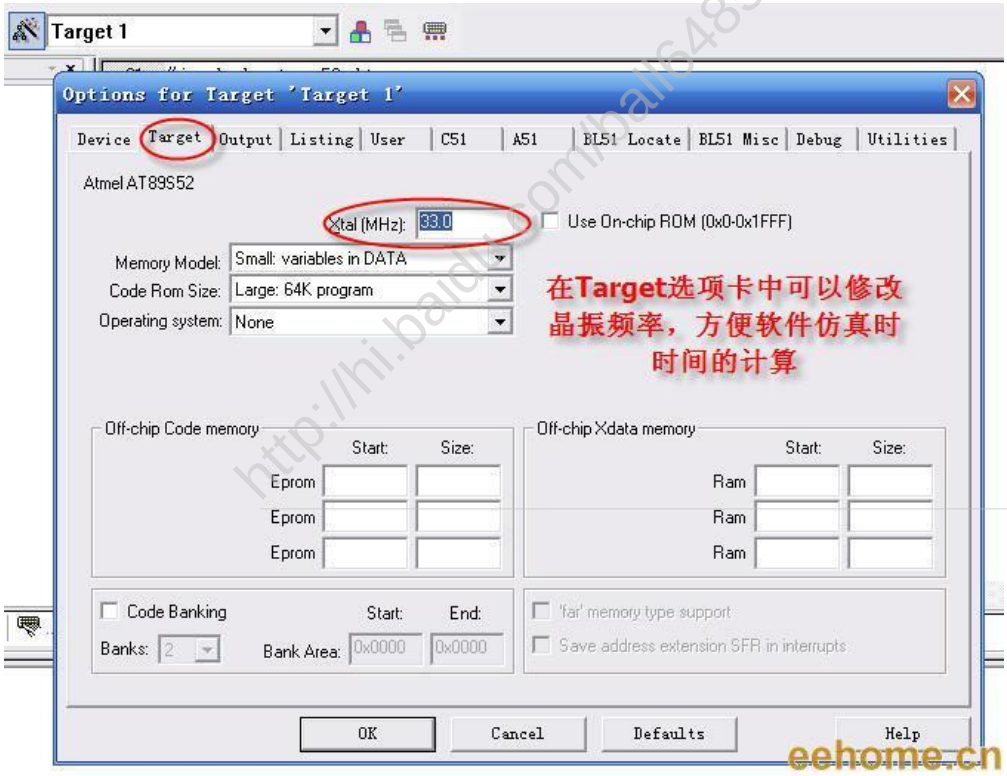


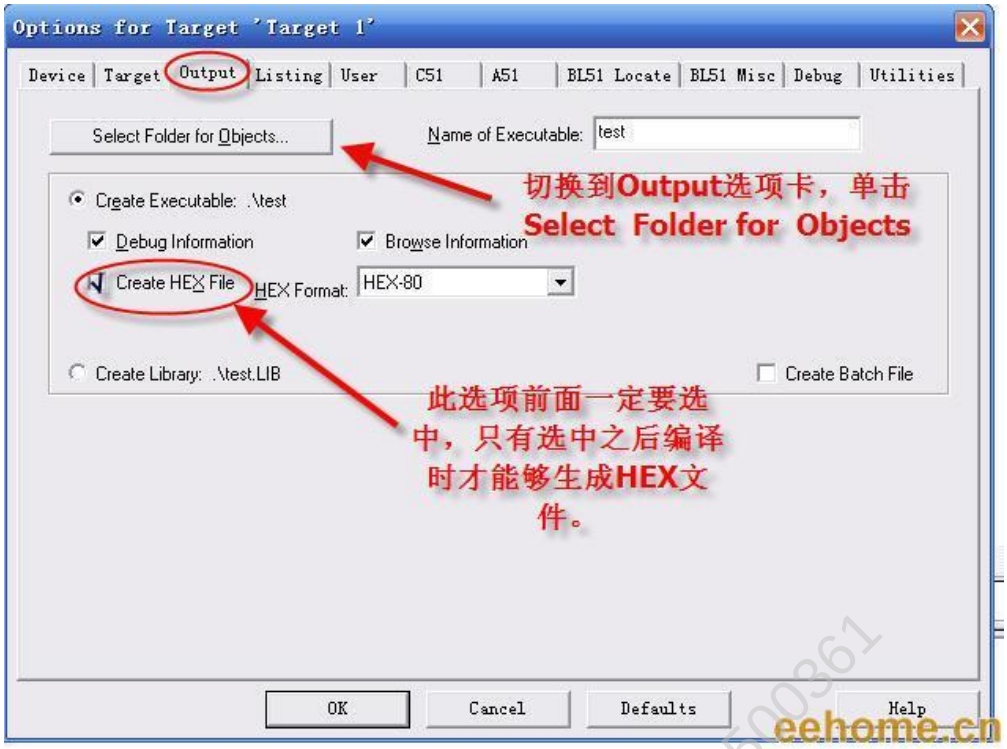


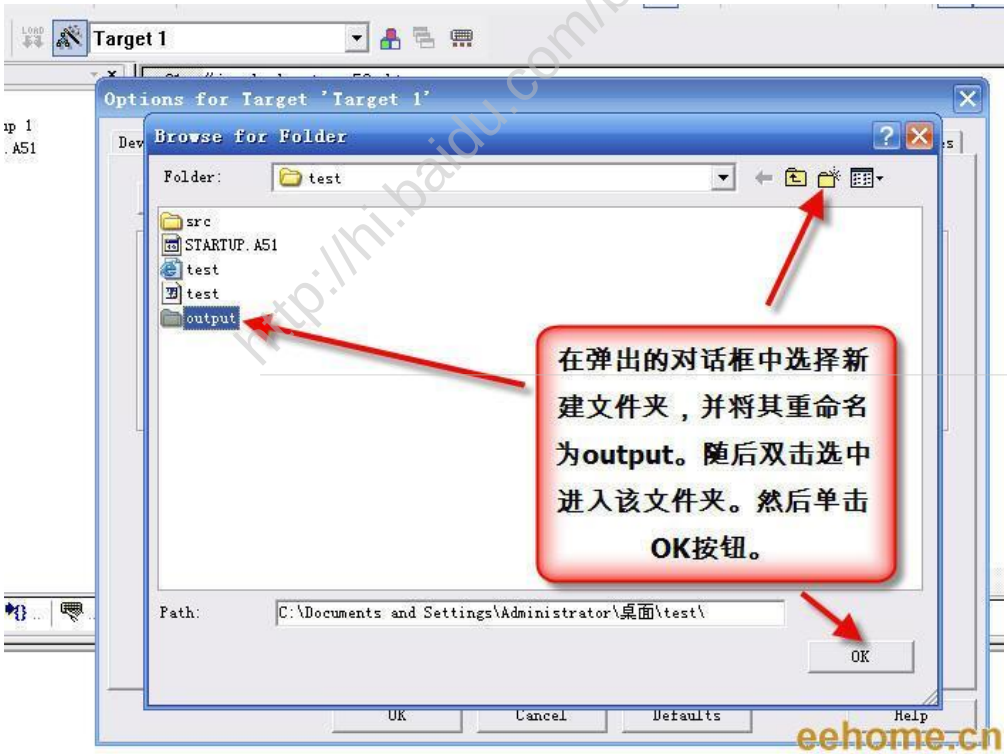


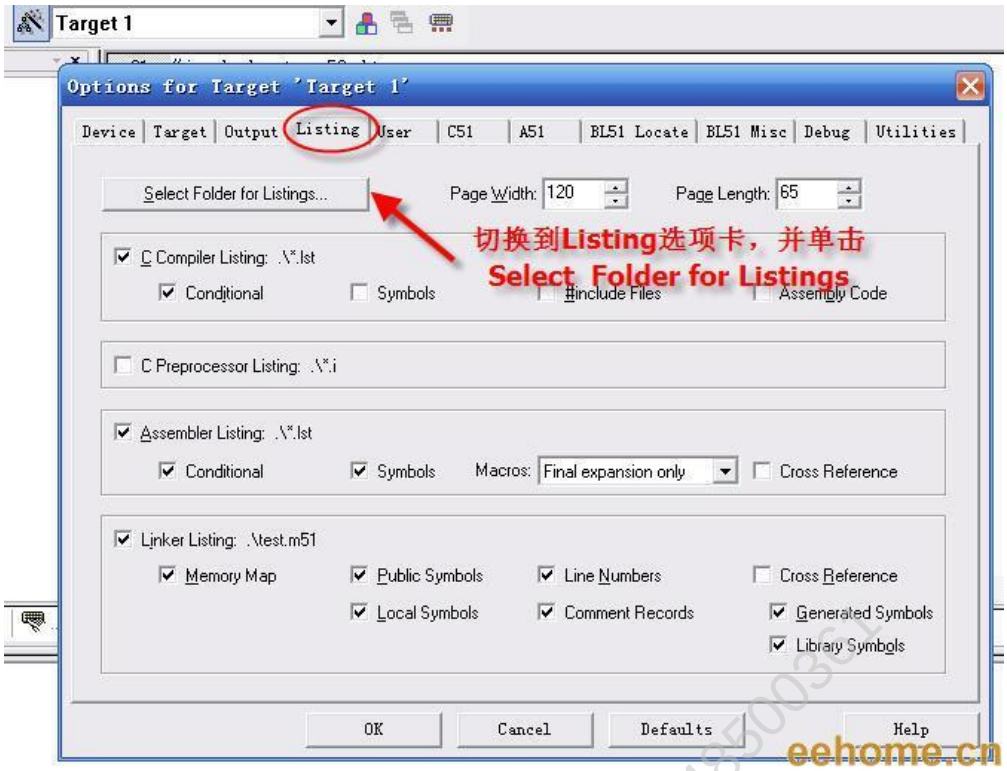


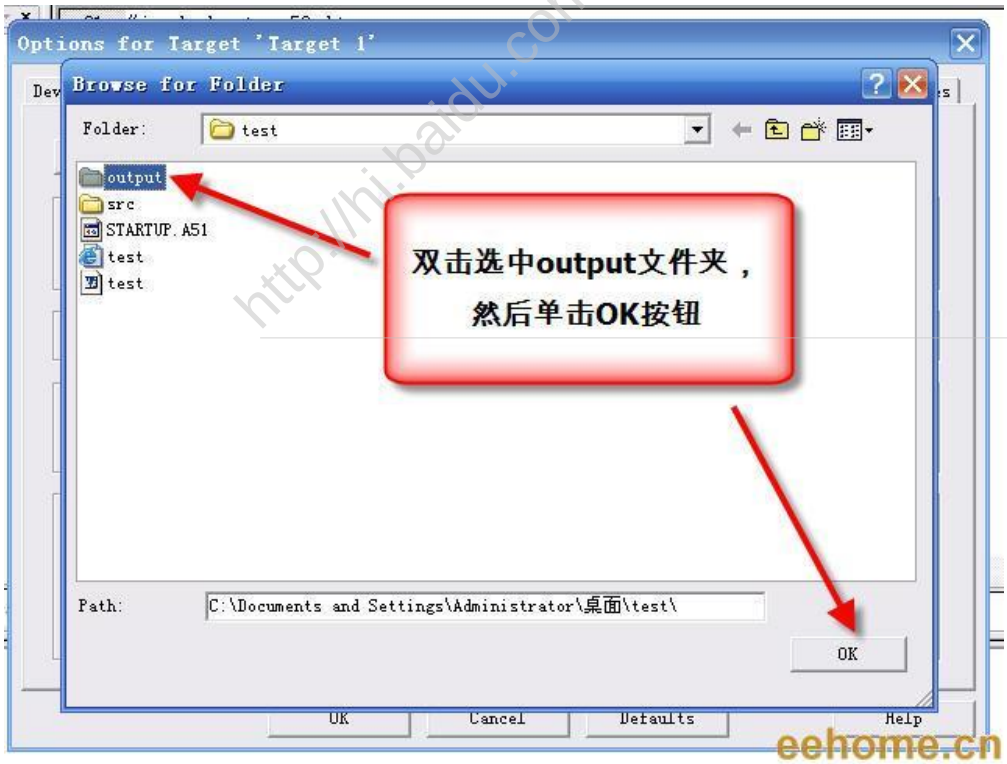


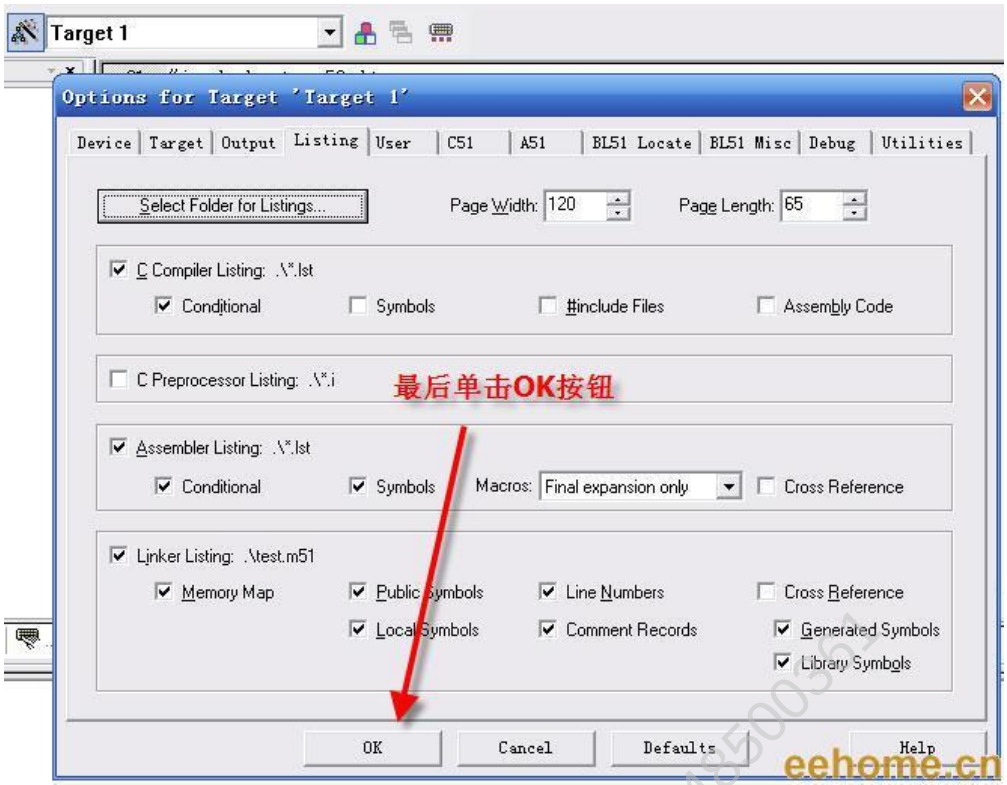


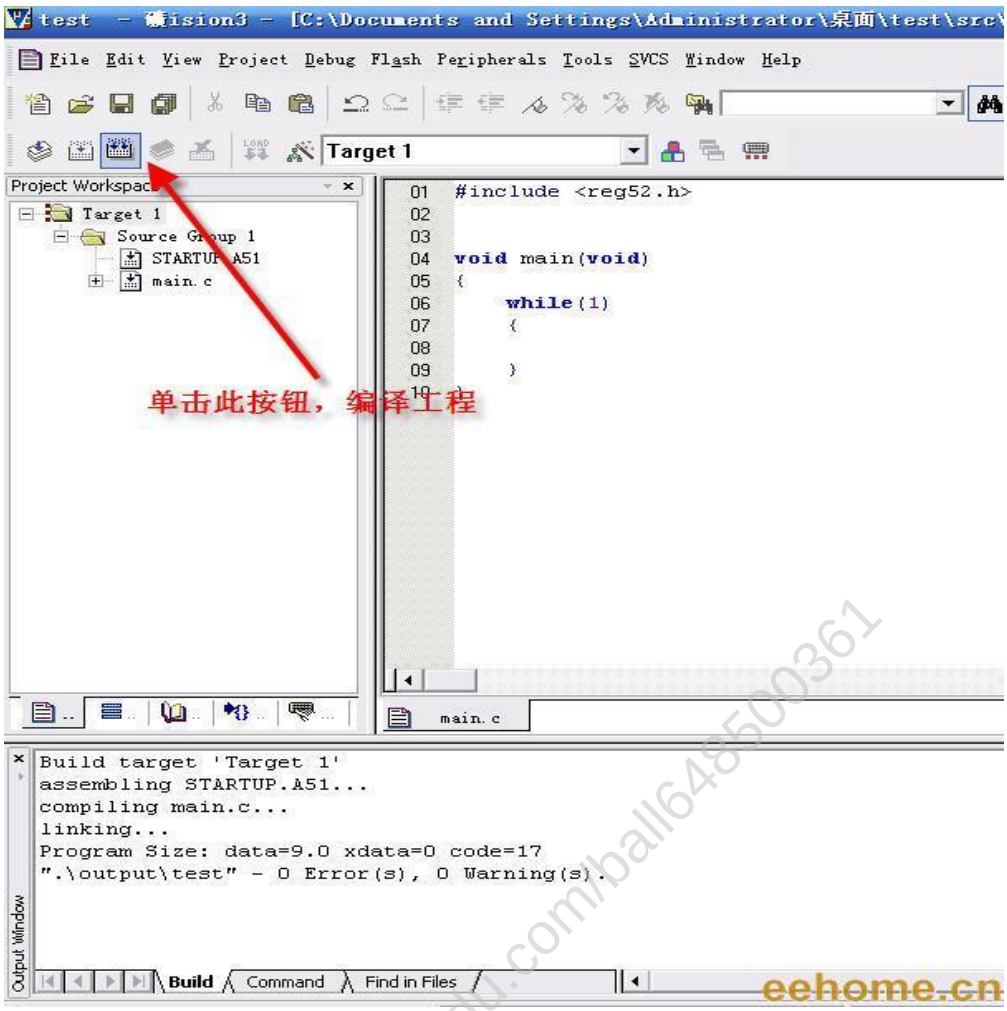












OK ，到此一个简单的工程模板就建立起来了，以后我们再新建源文件和头文件的时候，就可以直接保存到 src 文件目录下面了。

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
```

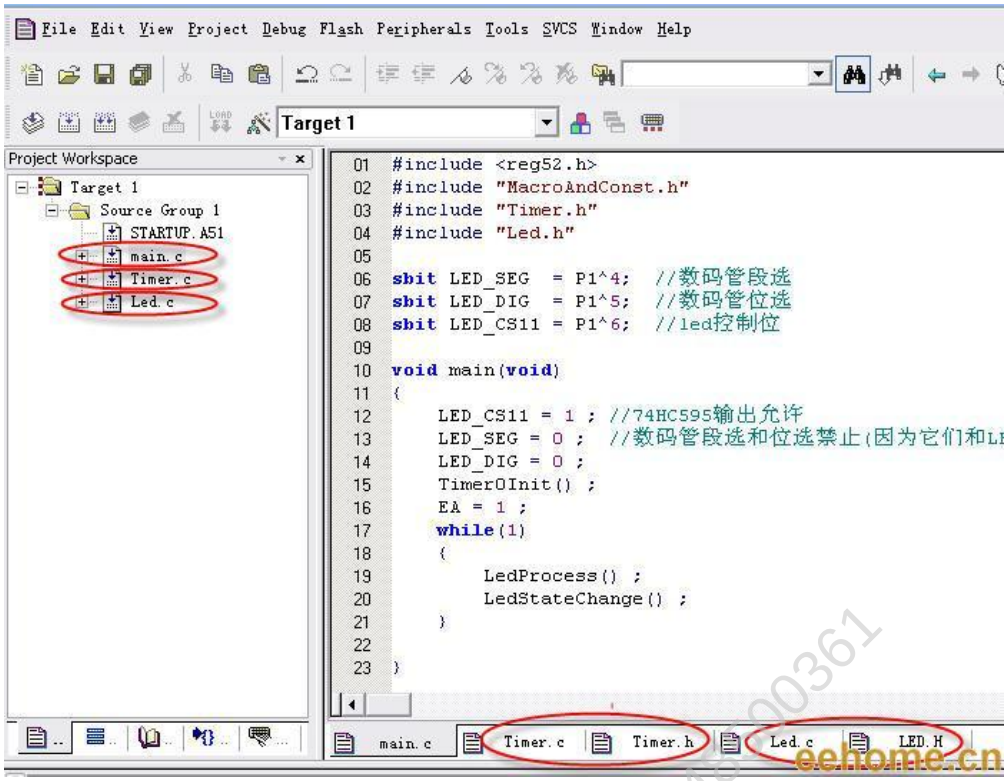
这两个模块完成后，我们将其 C 文件添加到工程中。然后开始编写主函数里的代码。
如下所示：

```
#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

[color=#FF0000]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

"从单片机初学者迈向单片机工程师"之 LED 主题讨论周第四章----渐明渐暗的灯

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。

“从单片机初学者迈向单片机工程师”之 LED 主题讨论周第五章--多任务环境下的数码管编程设计

[post]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：

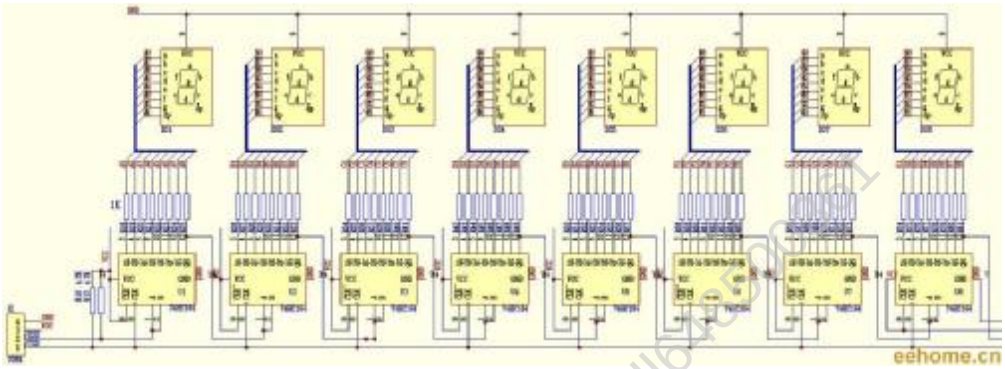
```
while(1)
{
    for(num=0;num<9;num++)
    {
            P0=table[num];
            P2=code[num] ;
            delayms(2) ;
    }
}
```

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 ϑms 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。

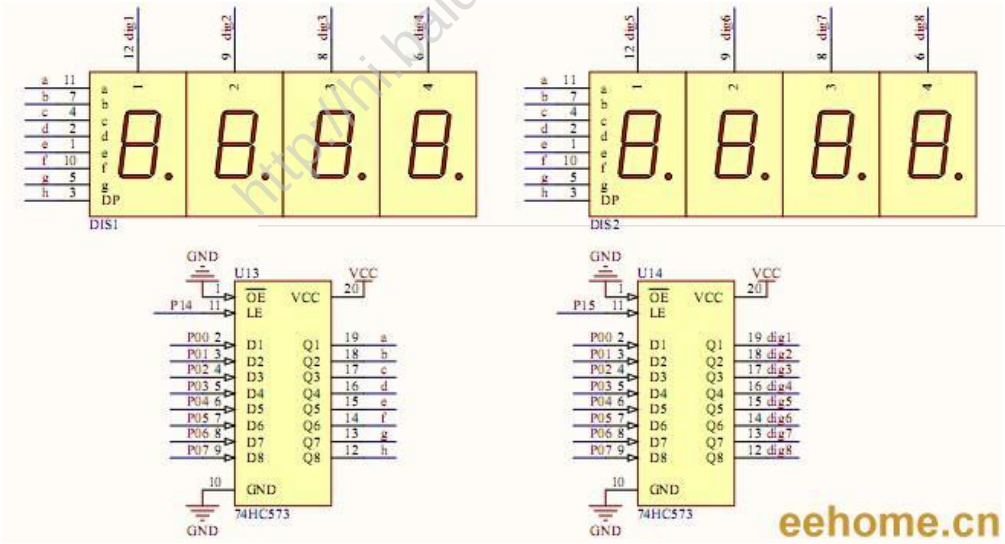
数码管的显示分为动态显示和静态显示两种。静态显示是每一位数码管都用一片独立的驱动芯片进行驱动。比较常见的有 74LS164，74HC595 等。利用这类芯片的好处就是可以级联，留给单片机的接口只需要时钟线，数据线，因此比较节省 I/O 口。如下图所示：



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) ;
        }
}
```

ϑ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。

[/post

本章所附例程在 EE21 学习板上调试通过，拥有板子的朋友可以直接下载附件对照学习

“从单片机初学者迈向单片机工程师”之 KEY 主题讨论第一章按键程序编写的基础

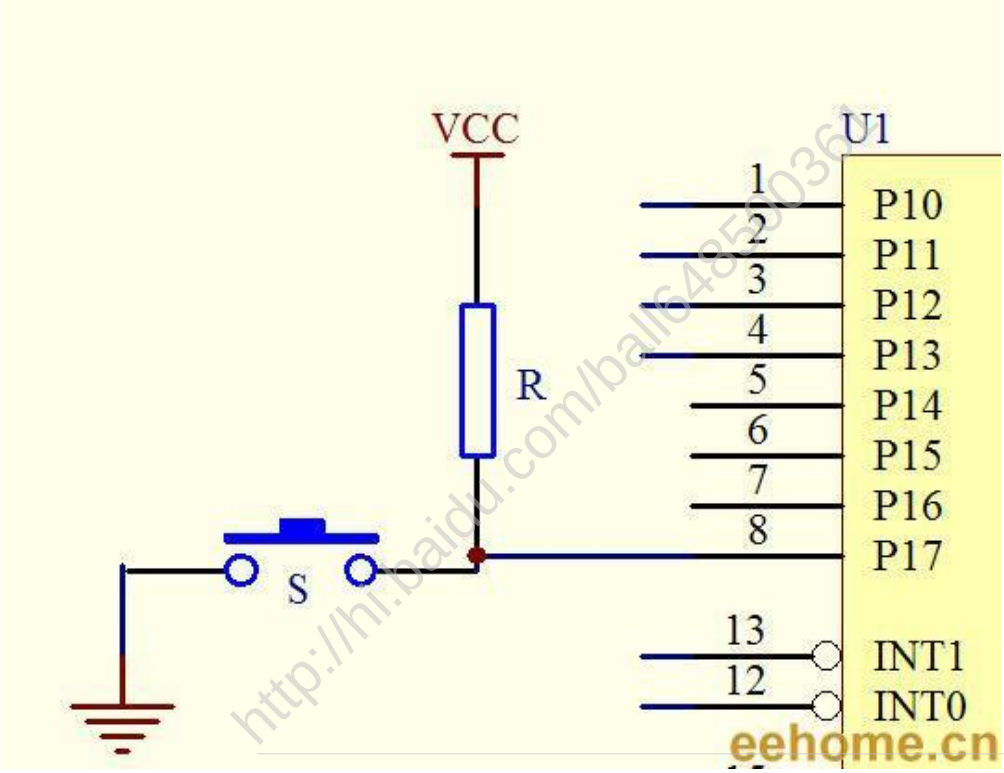
从这一章开始，我们步入按键程序设计的殿堂。在基于单片机为核心构成的应用系统中，用户输入是必不可少的一部分。输入可以分很多种情况，譬如有的系统支持 PS2 键盘的接口，有的系统输入是基于编码器，有的系统输入是基于串口或者 USB 或者其它输入通道等等。在各种输入途径中，更常见的是，基于单个按键或者由单个键盘按照一定排列构成的矩阵键盘(行列键盘)。我们这一篇章主要讨论的对象就是基于单个按键的程序设计，以及矩阵键盘的程序编写。

◎按键检测的原理

常见的独立按键的外观如下，相信大家并不陌生，各种常见的开发板学习板上随处可以看到他们的身影。

总共有四个引脚，一般情况下，处于同一边的两个引脚内部是连接在一起的，如何分辨两个引脚是否处在同一边呢？可以将按键翻转过来，处于同一边的两个引脚，有一条突起的线将他们连接一起，以标示它们俩是相连的。如果无法观察得到，用数字万用表的二极管挡位检测一下即可。搞清楚这点非常重要，对于我们画 PCB 的时候的封装很有益。

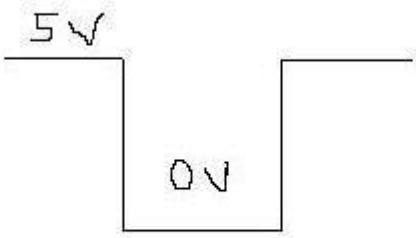
它们和我们的单片机系统的 I/O 口连接一般如下：



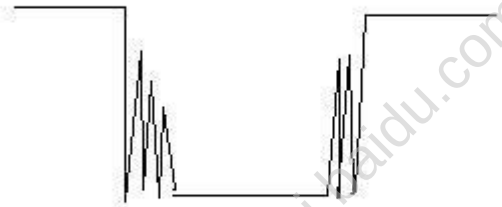
对于单片机 I/O 内部有上拉电阻的微控制器而言，还可以省掉外部的那个上拉电阻。简单分析一下按键检测的原理。当按键没有按下的时候，单片机 I/O 通过上拉电阻 R 接到 VCC，我们在程序中读取该 I/O 的电平的时候，其值为 1(高电平)；当按键 S 按下的时候，该 I/O 被短接到 GND，在程序中读取该 I/O 的电平的时候，其值为 0(低电平) 。这样，按键的按下与否，就和与该按键相连的 I/O 的电平的变化相对应起来。结论：我们在程序中通过检测到该 I/O 口电平的变化与否，即可以知道按键是否被按下，从而做出相应的响应。一切看起来很美好，是这样的吗？

◎现实并非理想

在我们通过上面的按键检测原理得出上述的结论的时候，其实忽略了一个重要的问题，那就是现实中按键按下时候的电平变化状态。我们的结论是基于理想的情况得出来的，就如同下面这幅按键按下时候对应电平变化的波形图一样：



而实际中，由于按键的弹片接触的时候，并不是一接触就紧紧的闭合，它还存在一定的抖动，尽管这个时间非常的短暂，但是对于我们执行时间以 us 为计算单位的微控制器来说，
它太漫长了。因而，实际的波形图应该如下面这幅示意图一样。



这样便存在这样一个问题。假设我们的系统有这样功能需求：在检测到按键按下的时候，将某个 I/O 的状态取反。由于这种抖动的存在，使得我们的微控制器误以为是多次按键的按下，从而将某个 I/O 的状态不断取反，这并不是我们想要的效果，假如该 I/O 控制着系统中某个重要的执行的部件，那结果更不是我们所期待的。于是乎有人便提出了软件消除抖动的思想，道理很简单：抖动的时间长度是一定的，只要我们避开这段抖动时期，检测稳定的时候的电平不久可以了吗？听起来确实不错，而且实际应用起来效果也还可以。于是，各种各样的书籍中，在提到按键检测的时候，总也不忘说道软件消抖。就像下面的伪代码所描述的一样。(假设按键按下时候，低电平有效)

```
If(0 == io_KeyEnter)              //如果有键按下了
{
    Delayms(20) ;              //先延时 20ms 避开抖动时期
    If(0 == io_KeyEnter)       //然后再检测，如果还是检测到有键按下
```

```
    {
        return KeyValue ;           //是真的按下了，返回键值
    }
    else
{
    return KEY_NULL         //是抖动，返回空的键值
}
    while(0 == io_KeyEnter) ;    //等待按键释放
}
```

乍看上去，确实挺不错，实际中呢？在实际的系统中，一般是不允许这么样做的。为什么呢？首先，这里的 Delayms(20) ，让微控制器在这里白白等待了 20 ms 的时间，啥也没干，考虑我在《学会释放 CPU》一章中所提及的几点，这是不可取的。其次 while(0 == io_KeyEnter) 所以合理的分配好微控制的处理时间，是编写按键程序的基础。ϑ;更是程序设计中的大忌(极少的特殊情况例外)。任何非极端情况下，都不要使用这样语句来堵塞微控制器的执行进程。原本是等待按键释放，结果 CPU 就一直死死的盯住该按键，其它事情都不管了，那其它事情不干了吗？你同意别人可不会同意

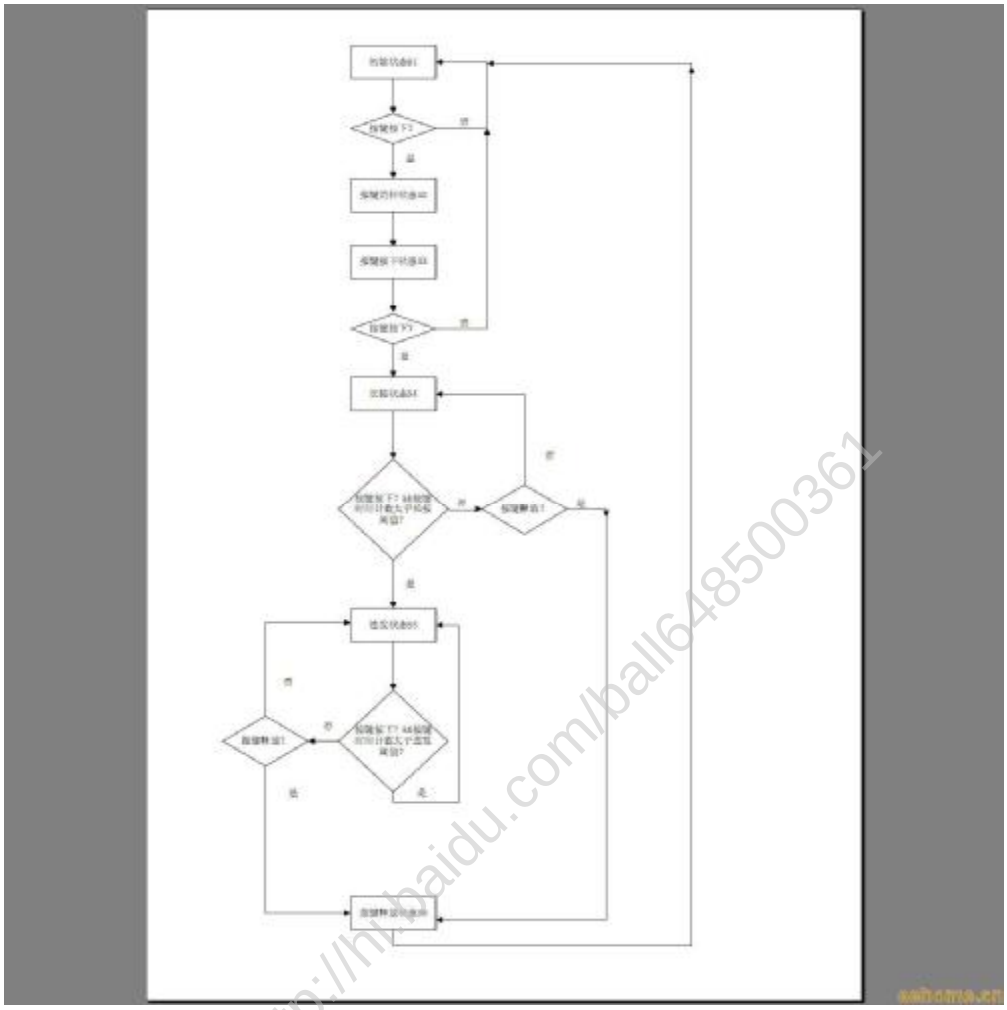
◎消除抖动有必要吗？

的确，软件上的消抖确实可以保证按键的有效检测。但是，这种消抖确实有必要吗？有人提出了这样的疑问。抖动是按键按下的过程中产生的，如果按键没有按下，抖动会产生吗？如果没有按键按下，抖动也会在 I/O 上出现，我会立刻把这个微控制器锤了，永远不用这样一款微控制器。所以抖动的出现即意味着按键已经按下，尽管这个电平还没有稳定。所以只要我们检测到按键按下，即可以返回键值，问题的关键是，在你执行完其它任务的时候，再次执行我们的按键任务的时候，抖动过程还没有结束，这样便有可能造成重复检测。所以，如何在返回键值后，避免重复检测，或者在按键一按下就执行功能函数，当功能函数的执行时间小于抖动时间时候，如何避免再次执行功能函数，就成为我们要考虑的问题了。这是一个仁者见仁，智者见智的问题，就留给大家去思考吧。所以消除抖动的目的是：防止按键一次按下，多次响应。

"从单片机初学者迈向单片机工程师"之 KEY 主题讨论第二章 基于状态转移的独立按键程序设计

本章所描述的按键程序要达到的目的：检测按键按下，短按，长按，释放。即通过按键的返回值我们可以获取到如下的信息：按键按下(短按)，按键长按，按键连发，按键释放。不知道大家还记得小时候玩过的电子钟没有，就是外形类似于 CALL 机(CALL)的那种，有一个小液晶屏，还有四个按键，功能是时钟，闹钟以及秒表。在调整时间的时候，短按+键每次调整值加一，长按的时候调整值连续增加。小的时候很好奇，这样的功能到底是如何实现的呢，今天就让我们来剖析它的原理吧。ϑ机，好像是很古老的东西了
状态在生活中随处可见。譬如早上的时候，闹钟把你叫醒了，这个时候，你便处于清醒的状态，马上你就穿衣起床洗漱吃早餐，这一系列事情就是你在这个状态做的事情。做完这些后你会去等车或者开车去上班，这个时候你就处在上班途中的状态.....中午下班时间到了，你就处于中午下班的状态，诸如此类等等，在每一个状态我们都会做一些不同的事情，而总会有外界条件促使我们转换到另外一种状态，譬如闹钟叫醒我们了，下班时间到了等等。对于状态的定义出发点不同，考虑的方向不同，或者会有些许细节上面的差异，但是大的状态总是相同的。生活中的事物同样遵循同样的规律，譬如，用一个智能充电器给你的手机电池充电，刚开始，它是处于快速充电状态，随着电量的增加，电压的升高，当达到规定的电压时候，它会转

换到恒压充电。总而言之，细心观察，你会发现生活中的总总都可以归结为一个个的状态，而状态的变换或者转移总是由某些条件引起同时伴随着一些动作的发生。我们的按键亦遵循同样的规律，下面让我们来简单的描绘一下它的状态流程转移图。

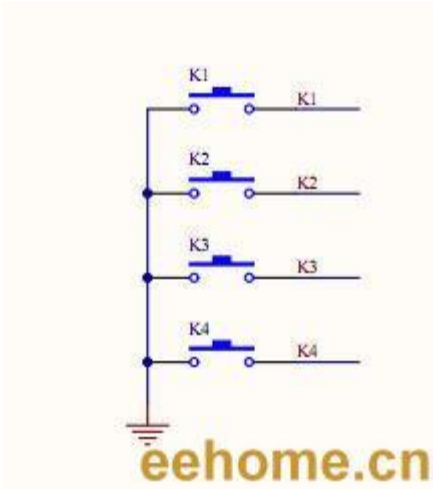


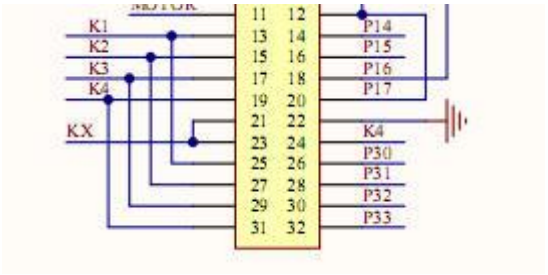
下面对上面的流程图进行简要的分析。

首先按键程序进入初始状态 S1，在这个状态下，检测按键是否按下，如果有按下，则进入按键消抖状态 2，在下一次执行按键程序时候，直接由按键消抖状态进入按键按下状态 3，在此状态下检测按键是否按下，如果没有按键按下，则返回初始状态 S1，如果有则可以返回键值，同时进入长按状态 S4，在长按状态下每次进入按键程序时候对按键时间计数，当计数值超过设定阈值时候，则表明长按事件发生，同时进入按键连发状态 S5。如果按键键值为空键，则返回按键释放状态 S6，否则继续停留在本状态。在按键连发状态下，如果按键键值为空键则返回按键释放状态 S6，如果按键时间计数超过连发阈值，则返回连发按键值，清零时间计数后继续停留在本状态。

看了这么多，也许你已经有一个模糊的概念了，下面让我们趁热打铁，一起来动手编写按键驱动程序吧。

下面是我使用的硬件的连接图。





硬件连接很简单，四个独立按键分别接在 P3^0------P3^3 四个 I/O 上面。

因为 51 单片机 I/O 口内部结构的限制，在读取外部引脚状态的时候，需要向端口写 1.在 51 单片机复位后，不需要进行此操作也可以进行读取外部引脚的操作。因此，在按键的端口没有复用的情况下，可以省略此步骤。而对于其它一些真正双向 I/O 口的单片机来说，将引脚设置成输入状态，是必不可少的一个步骤。

下面的程序代码初始化引脚为输入。

```
void KeyInit(void)
{
    io_key_1 = 1 ;
    io_key_2 = 1 ;
    io_key_3 = 1 ;
    io_key_4 = 1 ;
}
```

根据按键硬件连接定义按键键值

```
#define KEY_VALUE_1             0x0e
#define KEY_VALUE_2             0x0d
#define KEY_VALUE_3              0x0b
#define KEY_VALUE_4              0x07
#define KEY_NULL                 0x0f
```

下面我们来编写按键的硬件驱动程序。

根据第一章所描述的按键检测原理，我们可以很容易的得出如下的代码：

```
static uint8 KeyScan(void)
{
    if(io_key_1 == 0)return KEY_VALUE_1 ;
    if(io_key_2 == 0)return KEY_VALUE_2 ;
    if(io_key_3 == 0)return KEY_VALUE_3 ;
```

```
    if(io_key_4 == 0)return KEY_VALUE_4 ;
    return KEY_NULL ;
}
```

其中 io_key_1 等是我们按键端口的定义，如下所示：

```
sbit io_key_1 = P3^0 ;
sbit io_key_2 = P3^1 ;
sbit io_key_3 = P3^2 ;
sbit io_key_4 = P3^3 ;
```

KeyScan()作为底层按键的驱动程序，为上层按键扫描提供一个接口，这样我们编写的上层按键扫描函数可以几乎不用修改就可以拿到我们的其它程序中去使用，使得程序复用性大大提高。同时，通过有意识的将与底层硬件连接紧密的程序和与硬件无关的代码分开写，使得程序结构层次清晰，可移植性也更好。对于单片机类的程序而言，能够做到函数级别的代码重用已经足够了。

在编写我们的上层按键扫描函数之前，需要先完成一些宏定义。

```
//定义长按键的 TICK 数,以及连发间隔的 TICK 数
#define KEY_LONG_PERIOD          100
#define KEY_CONTINUE_PERIOD     25

//定义按键返回值状态(按下,长按,连发,释放)
#define KEY_DOWN                 0x80
#define KEY_LONG                 0x40
#define KEY_CONTINUE             0x20
#define KEY_UP                   0x10

//定义按键状态
#define KEY_STATE_INIT           0
#define KEY_STATE_WOBBLE         1
#define KEY_STATE_PRESS          2
#define KEY_STATE_LONG           3
#define KEY_STATE_CONTINUE       4
#define KEY_STATE_RELEASE        5
```


接着我们开始编写完整的上层按键扫描函数，按键的短按，长按，连按，释放等等状态的判断均是在此函数中完成。对照状态流程转移图，然后再看下面的函数代码，可以更容易的去理解函数的执行流程。完整的函数代码如下：

```
void GetKey(uint8 *pKeyValue)
{
    static uint8 s_u8KeyState = KEY_STATE_INIT ;
    static uint8 s_u8KeyTimeCount = 0 ;
    static uint8 s_u8LastKey = KEY_NULL ;  //保存按键释放时候的键值
    uint8 KeyTemp = KEY_NULL ;

    KeyTemp = KeyScan() ;          //获取键值
```

```
switch(s_u8KeyState)
{
    case KEY_STATE_INIT :
            {
                if(KEY_NULL != (KeyTemp))
                {
                    s_u8KeyState = KEY_STATE_WOBBLE ;
                }
            }
    break ;

    case KEY_STATE_WOBBLE :        //消抖
            {
                s_u8KeyState = KEY_STATE_PRESS ;
            }
    break ;

    case KEY_STATE_PRESS :
            {
                if(KEY_NULL != (KeyTemp))
                {
                    s_u8LastKey = KeyTemp ; //保存键值,以便在释放按键状态返回键值
                    KeyTemp |= KEY_DOWN ;   //按键按下
                    s_u8KeyState = KEY_STATE_LONG ;
                }
                else
                {
                    s_u8KeyState = KEY_STATE_INIT ;
                }
            }
    break ;

    case KEY_STATE_LONG :
            {
                if(KEY_NULL != (KeyTemp))
                {
                    if(++s_u8KeyTimeCount > KEY_LONG_PERIOD)
                    {
                        s_u8KeyTimeCount = 0 ;
                        KeyTemp |= KEY_LONG ;   //长按键事件发生
                        s_u8KeyState = KEY_STATE_CONTINUE ;
                    }
                }
```

```
                else
                {
                    s_u8KeyState = KEY_STATE_RELEASE ;
                }
            }
        break ;

        case KEY_STATE_CONTINUE :
            {
                if(KEY_NULL != (KeyTemp))
                {
                    if(++s_u8KeyTimeCount > KEY_CONTINUE_PERIOD)
                    {
                        s_u8KeyTimeCount = 0 ;
                        KeyTemp |= KEY_CONTINUE ;
                    }
                }
                else
                {
                    s_u8KeyState = KEY_STATE_RELEASE ;
                }
            }
        break ;

        case KEY_STATE_RELEASE :
            {
                s_u8LastKey |= KEY_UP ;
                KeyTemp = s_u8LastKey ;
                s_u8KeyState = KEY_STATE_INIT ;
            }
        break ;

        default : break ;
    }
    *pKeyValue = KeyTemp ; //返回键值
}
```

关于这个函数内部的细节我并不打算花过多笔墨去讲解。对照着按键状态流程转移图，然后去看程序代码，你会发现其实思路非常清晰。最能让人理解透彻的，莫非就是将整个程序自己看懂，然后想象为什么这个地方要这样写，抱着思考的态度去阅读程序，你会发现自己的程序水平会慢慢的提高。所以我更希望的是你能够认认真真的看完，然后思考。也许你会收获更多。

不管怎么样，这样的一个程序已经完成了本章开始时候要求的功能：按下，长按，连按，释放。事实上，如果掌握了这种基于状态转移的思想，你会发现要求实现其它按键功能，譬如，多键按下，功能键等等，亦相当简单，在下一章，我们就去实现它。

在主程序中我编写了这样的一段代码，来演示我实现的按键功能。

```
void main(void)
{
    uint8 KeyValue = KEY_NULL;
    uint8 temp = 0 ;
      LED_CS11 = 1 ; //流水灯输出允许
    LED_SEG = 0 ;
    LED_DIG = 0 ;
    Timer0Init() ;
    KeyInit() ;
    EA = 1 ;
    while(1)
    {
        Timer0MainLoop() ;
        KeyMainLoop(&KeyValue) ;

        if(KeyValue == (KEY_VALUE_1 | KEY_DOWN)) P0 = ~1 ;
        if(KeyValue == (KEY_VALUE_1 | KEY_LONG)) P0 = ~2 ;
        if(KeyValue == (KEY_VALUE_1 | KEY_CONTINUE)) { P0 ^= 0xf0;}
        if(KeyValue == (KEY_VALUE_1 | KEY_UP)) P0 = 0xa5 ;
    }

}
```

按住第一个键，可以清晰的看到 P0 口所接的 LED 的状态的变化。当按键按下时候，第一个 LED 灯亮，等待 2 S 后第二个 LED 亮，第一个熄灭，表示长按事件发生。再过 500 ms 第 5~8 个 LED 闪烁，表示连按事件发生。当释放按键时候，P0 口所接的 LED 的状态为：
灭亮灭亮亮灭亮灭，这也正是 P0 = 0xa5 这条语句的功能