说实话我能够使用的单片机不多，我总是以为无论什么单片机都能开发出好的产品。
前些年用 51，总是向各位大大学习，无休止的索取，在网上狂览一通。心里感激的同时也想奉献一些，可是我会什么？后来使用 avr（公司要求）还是向大大们学习，我又想奉献，
可是我会什么？我会的大大们都写了，我不会的大大们也写了。一个星期前花项目经费买了***的 kit 三合一板，最近几天闲了下来，便动手调试一下。算是有点心得，我又想奉献，可是我会什么？

我只是想和大大们交流一下，哪怕是对的或者是错的，大大们满足我的一点心愿吧。
唠叨了这么多，现在开始吧。

配置： stvd ， cosmic

我学单片机开门三砖总是要砸的。

第一砖： 电源系统，这没什么好说的，只是它是 stm8 工作的基础总是要提一下

第二砖： 时钟系统，这等下再说。

第三砖： 复位系统，stm8 只需要一只 104 电容从 reset 脚到地就可以了。

现在说说时钟系统，学习单片机无论 8 位的还是 32 位的，都要从时钟开始，下面是我一开始的时钟切换程序。

```
1    CLK_ECKR |=0X1;    //开启外部时钟

2    while(!(CLK_ECKR&0X2)); //等待外部时钟 rdy

3    CLK_CKDIVR &= 0XF8;       //CPU 无分频

4    CLK_SWR = 0XB4;    //选择外部时钟


5    CLK_SWCR |=0X2;    //使能外部时钟
```

上面的代码看起来没什么问题，可在调试过程中出现了有时能切换，有时有不能的情况，后来发现只要在第 5 行设上断点就能切换，我就想是不是得让 cpu 等一下，我又仔细的翻看下 rm0016 的时钟部分，发现得等待 CLK_SWCR 的标志位置位才能切换。

就变成了下面的代码

```
 CLK_ECKR |=0X1;    //开启外部时钟

while(!(CLK_ECKR&0X2)); //等待外部时钟 rdy

CLK_CKDIVR &= 0XF8;       //CPU 无分频

CLK_SWR = 0XB4;    //选择外部时钟

while(!(CLK_SWCR&0X8)); //这里要等


 CLK_SWCR |=0X2;    //使能外部时钟
```

现在一切 ok，是不是觉得看东西要仔细一下~~。顺便说一下，stm8 有三个时钟源的，hse 是外部时钟，hsi 是内部 16mhz 的时钟。Stm8 一启动默认为内部时钟，并且 8 分频。
其实这么处理不是最好的办法，如果外部时钟出了问题，stm8 要傻傻的等待到死。它可以有中断的，在中断中处理一切，包括恢复时钟源，这才是正道，只是我比较懒，不是做正规产品，想都不愿去想。

长长的一篇，没什么内容，请原谅我的唠叨吧。

又想起一句，仔细看手册里的时钟概略图吧，这对你有帮助。

第二节：傻的可爱—cosmic 和 time 的事情

使用单片机定时器总是用到的，无论是延时，键盘扫描，显示刷新，还是巨无霸的操作系统。Time1 太过复杂等过些天再说，我是从 time2 开始的，从简单的定时开始吧。
简单的解释一下，time2 是向上计数的，不像 time1 可以双向计数（这对我很有用，我可以使用它的正交编码功能，这正是我学 stm8 的初衷，它可以让我省下一片正交计数器或是一片 cpld，等过两天从公司借个编码器，调试一下），我们怎么可以达到定时 1ms 的目的哪？
关键是 TIM2_ARR 这个寄存器，TIM2_CNTR 是计数到 TIM2_ARR 就产生更新事件，然后清零从头开始的，看下面的代码。

1 CLK_PCKENR1 |=0X20;//开启时钟，stm8 的外设时钟可控

2 TIM2_PSCR |=0X3; //DIV8　1US ->　外部晶振 8mhz 除以 8 实现单位时间为 1us

3 TIM2_IER |= 0X1; //允许中断

4 TIM2_ARR = 0X3E7;　//关键是这里

5 TIM2_CR1 |= 0X1;　//开启定时器

这看起来没错，可就是不能实现定时效果，这是为什么？答案出乎我的意料，看汇编代码后才发现，comsic 使用了 ldw 指令，而 ldw 指令是先写低位再写高位的。ARR 寄存器是要求先写高位再写低位的，将第 4 行改为

TIM2_ARRH = 0X3;　//

TIM2_ARRL = 0XE7;
后，问题解决。用 avr 时 gcc 编译器都给做好了，comsic 很傻很强大。记住这个教训吧，要看编译器手册，不要偷懒，多写一行就多写一行吧。中断部分以后再说。就到这里，明天再聊，这耽误我看小说的时间了，哎，为了 stm8 我已经 4 晚上没看小说了。

第三节：ad 的单次转换

说起 ad 我是就头大，不是说 stm8 的 ad 让我头大，而是以前在产品中使用的 ad
老板总是要求越来越高，从 16bit 到 24bit，从逐渐逼近到 sigma ，在电路上克服小信号的采集实在是一件痛苦的事情，至今在 24bit 的采集上只能到 18bit 有效位，有经验的朋友一定要告诉我。
又扯远了，stm8 只是 10bit 的 ad，随便用用就可以了，我从来没指望它能给我出大力气，当然大大们做民品，或是别的要求不高的可以用用。
为什么说单次转换呢？因为简单，因为我懒。看下面的代码吧

//这里是初始化
CLK_PCKENR2 |=0X8; //使能 adc 时钟

```
ADC_CSR    |=0X3;       //选择通道 3 禁止中断

ADC_CR1 |=0X71;       //使能 ADC，18 分频

ADC_CR2 |=0X8;        //数据右对齐，low 8BIT    AT    ADC_DRL;
//这里是转换结果

unsigned int x;

   unsigned int x_h;

   ADC_CR1 |=0X1;    //启动转换

   while(!(ADC_CSR&0x80)); //等待转换结束    14 个时钟周期

ADC_CSR &= 0X7F;           //清除中断标志

x = ADC_DRL;               //READ DATA 因为是右对齐所以先读低位

x_h = ADC_DRH;

x_h =( x_h << 8 ) + x;

        return x_h;
```

这次没出什么错，大家失望了吧！哈哈，说点题外话，做 16bit 以上 ad 我认为要注意几点

1. 有一个好的基准
2. 传感器供电最好和基准联动
3. 要有效去除长线干扰，如加屏蔽网，做线阻平衡。
4. 使上两个好的电阻吧，会省很大力气
5. 布线要花大力气，不能瞎布。

其它的还有很多，大大们到网上看吧，前人栽树，后人乘凉。我们即要做前人，也要做后人。

第四节：中断系统和一杯热茶

最近喜欢喝茶，准备去买一套茶具，一个小壶，八个小杯那种。我喜欢

铁观音，浓浓的，滚烫的，直入喉咙。

中断就像一杯浓浓的铁观音，没有操作系统的时候，使用中断吧，一样可以达到实时响应。没有极品龙井，就喝铁观音吧，一样口齿留香。

Stm8 的中断是有优先级的，不是 avr 那种假优先级，是那种低级中断正在处理，高级中断可以终止它的优先级。

我们不说这些，它在不做项目时，离我还很遥远。

说说 comsic 的开中断手段吧看下面的语句

_asm("sim"); //这是关中断

_asm("rim");//这是开中断

我刚开始还以为 sim 是开中断，结果定时中断总是进不去。

_asm()插入汇编行，多行可以用\n 分割

汇编块可以使用下面格式

#asm

//汇编代码

#endasm

或者

#pragma asm

#pragma endasm

Stvd 自带了中断处理文件，在向量表里修改中断号处的函数名，来实现中断发生时程序跳到我们的中断处理程序。

我写了前面关于 time2 的更新中断。

向量表中 irq13 处改成这样 {0x82, TIME2_UIS}, /* irq13 */

```
@far @interrupt void TIME2_UIS ( void )
{
if( ++count>temp)
   {

count = 0;

PD_ODR ^=0X1; //LED 翻转

}

TIM2_SR1 &=0XFE; //中断标志位，它不会自动清零
```

```
return;


}
```

Temp 是前面 ad 转换的结果，这里来实现 led 的闪烁频率。@far 是指长指针，@interrupt 指示这是一个中断处理函数。

本来还想说 uart 的中断的，又一想明天我说 uart 的时候说啥。所以还是留在明天再说吧。

茶喝的多，睡眠质量受影响啊。

第五节： 永恒的串口和阶段感言

等说完串口，就要等一些天再和大家见面了，孩子总是和我捣乱，那是我的第一生命。

是我祖祖辈辈的延续。请原谅我的古老，我喜欢传统的，无论是京剧，大鼓还是快板。说起孩子，心情总是愉快的，有一天孩子感冒去医院，医生要验血，临近化验室时，孩子哭闹，妻子哄骗说是妻子验血，等抽完孩子的血孩子哇哇大哭并质问：“为什么你化验抽我的血”我和妻子苦笑。现在想来，孩子那时天真可爱，现在的孩子俨然一副大人麽样，他才 4 岁呀，是我做的不好吗？我从来不让他在家做和玩耍无关的事情，包括学习。别家的孩子大都报各种专长班，我从来都阻止妻子去给孩子增加负担。我要他的童年快快乐乐。我要让他童年充满童真，可是我做不到。孩子越来越聪明，越来越成熟，是我们老了吗？

又跑题了，串口，自从我开始开发产品从来没离开过串口。因为我总要和计算机或其他的 mcu 说话，而串口是最简单和经济的方式。

传统的也是最难舍弃，stm8 的串口资源很丰厚，都有两个。好些年前，要用双串口除了使用专业芯片外只能选择华邦的芯片，说实话它那时真的很贵。Avr 也有双串口的，所以我一见双串口的芯片，总是兴奋。大概得了串口恐惧症了。

看代码：

```
    CLK_PCKENR1 |= 0X08; //开启时钟

    LINUART_BRR2 = 0X1;

LINUART_BRR1 = 0X1A;   //19200BPS
```

```
LINUART_CR2 = 0XAC;    //8，n，1 开启发送和接受中断
```

上面是初始化部分，很是简单自己看看吧。

我接下来要用串口中断做的事情很无聊，我要实现无论串口接收到一个什么数据，都要返回该数据并加发 0x55，0xaa。实时上这个协议一点用处都没有，我希望大家开发产品的时候有串口协议时，如果资源够用，又不愿自己写时，使用 modobus 协议吧，真的很好用。
下面是中断程序

```
@far @interrupt void USART_TX( void )
{


     switch( status )
{

case 0:

LINUART_DR = 0X55;

status = 1;

break;
case 1:

LINUART_DR = 0XAA;

status = 2;
break;
case 2:

LINUART_CR2 = 0X2C;    //数据空中断只能写 dr 清除，所以只能禁止它

status = 0;
break;

}
```

```
     return;
}

@far @interrupt void USART_RX(void )
{

unsigned char x;

x = LINUART_DR ;   //读数据自动清除中断标志



LINUART_DR = x;   //同时清除发送空中断标志

LINUART_CR2 = 0XAC;//所以可以打开发送空中断了

status = 0;

     return;
}
```

同样在向量表中改成这样

```
     {0x82, USART_TX}, /* irq20 */
     {0x82, USART_RX}, /* irq21 */
```

在这个简单的基础上，就可以开发自己的协议了。我用串口只使用这么多功能，别的如 lin，idra，或是别的都是以后的事了。

和兄弟们说声再见，下次在写时就是正交编码和 spi 了。

第五节： 正交编码和疑惑

今天去公司，找遍了废品堆都没有找到一只编码器，没办法只好从半成

品上 拆下来一个，大家不要说是我做的，不然老板会很生气。

正交计数方法很多，软件的，cpld 的，芯片的都可以，但 cpu 上集成了我们为什么不用，我没理由不选带正交功能的 stm8，因为他是 8bit 的，因为他价格据说很便宜，32bit 的 cpu 大多是带这个功能的包括 dsp，我总是说在我的产品里他是大马，我的产品是小车，其实是我不愿去啃 32bit 的大部头。写完这篇我下定决心要使用 stm32 了
到时候兄弟们一定要帮助我，就当是扶贫吧。

***的三合一板使用的芯片是 s207s8t6，44 脚的，time1 的两个输入段为 pc1，和 pc2，我将编码器的 a，b 相分别接在 PC1,PC2 上。接上 VCC 和 gnd ，电路的工作
算是完成，接下来都是软件的工作。

在此之前看看 stm32 的正交编码接口应用笔记吧，上面对原理描述的很清楚，比我说的要有条理，我就不说了。看下面的代码

```
//下面是初始化部分
CLK_PCKENR1 |=0X80; //开启 time1 时钟

TIM1_SMCR |= 0X3;     //工作在编码器模式 3

TIM1_CCMR1 |= 0X1;    //CC1 MAP TI1FP1    CH1

TIM1_CCMR2 |= 0X1; //CC2 MAP    TI2FP2    CH2

TIM1_ARRH = 0XEA;    // 60000 产生溢出

TIM1_ARRL = 0X60;

TIM1_IER |=0X1;    //开中断

TIM1_CNTR = cnt_start = 30000; //我要有个大的初始化值
                                //正好是满量程的一半

TIM1_CR1 =0X1;     //启动计数
```

通过上面简单的配置，time1 正是工作了，旋动编码器，可以看到 TIM 1_CNTR 的数据变动，我的 1000 线编码每转一圈产生 4000 个数。

在我的中断和主程序里做了处理，可计数范围扩展到 32bit，算是基本

达到了我的要求。有一件事要说一下，读 TIM1_CNTR 时要先读高位，再读低位。

Stm8 的工作告一段落，本来还要写 spi 的，可是还要搭外围电路，等一些天吧，我把 ad7705 接上，手中有十几片闲置的。

今天无意中在中断里做了 long 型数据加法，编译时居然出错，翻了翻编译器手册，没找到原因，希望知道的朋友告诉我一声。

没有使用意法的库，是因为我觉得使用它不利于入门，虽然它结构优美。做项目的时候再用吧。

STM8 的 C 语言编程（1）——基本程序与启动代码分析

现在几乎所有的单片机都能用 C 语言编程了，采用 C 语言编程确实能带来很多好处，至少可读性比汇编语言强多了。
在 STM8 的开发环境中，可以通过新建一个工程，自动地建立起一个 C 语言的框架，生成后开发环境会自动生成 2 个 C 语言的程序，一个是 main.c，另一个是 stm8_interrupt_vector.c。main.c 中就是一个空的 main()函数，如下所示：

```
/* MAIN.C file
 *
 * Copyright (c) 2002-2005 STMicroelectronics
 */


main()
{
      while (1);
}
```

而在 stm8_interrupt_vector.c 中，就是声明了对应该芯片的中断向量，如下所示：

```
/*    BASIC INTERRUPT VECTOR TABLE FOR STM8 devices
 *    Copyright (c) 2007 STMicroelectronics
 */

typedef void @far (*interrupt_handler_t)(void);
```

```
struct interrupt_vector {
    unsigned char interrupt_instruction;
    interrupt_handler_t interrupt_handler;
};

@far @interrupt void NonHandledInterrupt (void)
{
    /* in order to detect unexpected events during development,
       it is recommended to set a breakpoint on the following instruction
    */
    return;
}

extern void _stext();     /* startup routine */

struct interrupt_vector const _vectab[] = {
    {0x82, (interrupt_handler_t)_stext}, /* reset */
    {0x82, NonHandledInterrupt}, /* trap  */
    {0x82, NonHandledInterrupt}, /* irq0  */
    {0x82, NonHandledInterrupt}, /* irq1  */
    {0x82, NonHandledInterrupt}, /* irq2  */
    {0x82, NonHandledInterrupt}, /* irq3  */
    {0x82, NonHandledInterrupt}, /* irq4  */
    {0x82, NonHandledInterrupt}, /* irq5  */
    {0x82, NonHandledInterrupt}, /* irq6  */
    {0x82, NonHandledInterrupt}, /* irq7  */
    {0x82, NonHandledInterrupt}, /* irq8  */
    {0x82, NonHandledInterrupt}, /* irq9  */
    {0x82, NonHandledInterrupt}, /* irq10 */
    {0x82, NonHandledInterrupt}, /* irq11 */
    {0x82, NonHandledInterrupt}, /* irq12 */
    {0x82, NonHandledInterrupt}, /* irq13 */
    {0x82, NonHandledInterrupt}, /* irq14 */
    {0x82, NonHandledInterrupt}, /* irq15 */
    {0x82, NonHandledInterrupt}, /* irq16 */
    {0x82, NonHandledInterrupt}, /* irq17 */
    {0x82, NonHandledInterrupt}, /* irq18 */
```

```
    {0x82, NonHandledInterrupt}, /* irq19 */
    {0x82, NonHandledInterrupt}, /* irq20 */
    {0x82, NonHandledInterrupt}, /* irq21 */
    {0x82, NonHandledInterrupt}, /* irq22 */
    {0x82, NonHandledInterrupt}, /* irq23 */
    {0x82, NonHandledInterrupt}, /* irq24 */
    {0x82, NonHandledInterrupt}, /* irq25 */
    {0x82, NonHandledInterrupt}, /* irq26 */
    {0x82, NonHandledInterrupt}, /* irq27 */
    {0x82, NonHandledInterrupt}, /* irq28 */
    {0x82, NonHandledInterrupt}, /* irq29 */
};
```

在 stm8_interrupt_vector.c 中，除了定义了中断向量表外，还定义了空的中断服务程序，用于那些不用的中断。当然在自动建立时，所有的中断服务都是空的，因此，除了第 1 个复位的向量外，其它都指向那个空的中断服务函数。

生成框架后，就可以用 Build 菜单下的 Rebuild All 对项目进行编译和连接，生成所需的目标文件，然后就可以加载到 STM8 的芯片中，这里由于 main()函数是一个空函数，因此没有任何实际的功能。不过我们可以把这个框架对应的汇编代码反出来，看看 C 语言生成的代码，这样可以更深入地了解 C 语言编程的特点。

生成的代码包括 4 个部分，如图 1、图 2、图 3、图 4 所示。

```
0x8000 <__vectab>          0x82008083     INT   0x008083          INT   __stext
0x8004 <__vectab+4>        0x820080D0     INT   0x0080d0          INT   NonHandledInterrupt
0x8008 <__vectab+8>        0x820080D0     INT   0x0080d0          INT   NonHandledInterrupt
0x800c <__vectab+12>       0x820080D0     INT   0x0080d0          INT   NonHandledInterrupt
0x8010 <__vectab+16>       0x820080D0     INT   0x0080d0          INT   NonHandledInterrupt
0x8014 <__vectab+20>       0x820080D0     INT   0x0080d0          INT   NonHandledInterrupt
0x8018 <__vectab+24>       0x820080D0     INT   0x0080d0          INT   NonHandledInterrupt
0x801c <__vectab+28>       0x820080D0     INT   0x0080d0          INT   NonHandledInterrupt
0x8020 <__vectab+32>       0x820080D0     INT   0x0080d0          INT   NonHandledInterrupt
0x8024 <__vectab+36>       0x820080D0     INT   0x0080d0          INT   NonHandledInterrupt
0x8028 <__vectab+40>       0x820080D0     INT   0x0080d0          INT   NonHandledInterrupt
0x802c <__vectab+44>       0x820080D0     INT   0x0080d0          INT   NonHandledInterrupt
0x8030 <__vectab+48>       0x820080D0     INT   0x0080d0          INT   NonHandledInterrupt
0x8034 <__vectab+52>       0x820080D0     INT   0x0080d0          INT   NonHandledInterrupt
0x8038 <__vectab+56>       0x820080D0     INT   0x0080d0          INT   NonHandledInterrupt
0x803c <__vectab+60>       0x820080D0     INT   0x0080d0          INT   NonHandledInterrupt
0x8040 <__vectab+64>       0x820080D0     INT   0x0080d0          INT   NonHandledInterrupt
0x8044 <__vectab+68>       0x820080D0     INT   0x0080d0          INT   NonHandledInterrupt
0x8048 <__vectab+72>       0x820080D0     INT   0x0080d0          INT   NonHandledInterrupt
0x804c <__vectab+76>       0x820080D0     INT   0x0080d0          INT   NonHandledInterrupt
0x8050 <__vectab+80>       0x820080D0     INT   0x0080d0          INT   NonHandledInterrupt
0x8054 <__vectab+84>       0x820080D0     INT   0x0080d0          INT   NonHandledInterrupt
0x8058 <__vectab+88>       0x820080D0     INT   0x0080d0          INT   NonHandledInterrupt
0x805c <__vectab+92>       0x820080D0     INT   0x0080d0          INT   NonHandledInterrupt
0x8060 <__vectab+96>       0x820080D0     INT   0x0080d0          INT   NonHandledInterrupt
0x8064 <__vectab+100>      0x820080D0     INT   0x0080d0          INT   NonHandledInterrupt
0x8068 <__vectab+104>      0x820080D0     INT   0x0080d0          INT   NonHandledInterrupt
0x806c <__vectab+108>      0x820080D0     INT   0x0080d0          INT   NonHandledInterrupt
0x8070 <__vectab+112>      0x820080D0     INT   0x0080d0          INT   NonHandledInterrupt
0x8074 <__vectab+116>      0x820080D0     INT   0x0080d0          INT   NonHandledInterrupt
0x8078 <__vectab+120>      0x820080D0     INT   0x0080d0          INT   NonHandledInterrupt
0x807c <__vectab+124>      0x820080D0     INT   0x0080d0          INT   NonHandledInterrupt
```

图 1

```
0x8080 <__idesc__>          0x80          DATA
0x8081 <__idesc__+1>        0x83          DATA
0x8082 <__idesc__+2>        0x00          DATA
```

图 2

```
0x8083 <__stext>          0xAE0FFF      LDW   X,#0x0fff      LDW   X,#0x0fff
0x8086 <__stext+3>        0x94          LDW   SP,X           LDW   SP,X
0x8087 <__stext+4>        0x90CE8080    LDW   Y,0x8080       LDW   Y,__idesc__
0x808b <__stext+8>        0xAE8082      LDW   X,#0x8082      LDW   X,#0x8082
0x808e <__stext+11>       0xF6          LD    A,(X)          LD    A,(X)
0x808f <__stext+12>       0x2720        JREQ  0x80b1         JREQ  0x80b1
0x8091 <__stext+14>       0xA560        BCP   A,#0x60        BCP   A,#0x60
0x8093 <__stext+16>       0x2717        JREQ  0x80ac         JREQ  0x80ac
0x8095 <__stext+18>       0xBF00        LDW   0x00,X         LDW   0x00,X
0x8097 <__stext+20>       0xEE03        LDW   X,(0x03,X)     LDW   X,(0x03,X)
0x8099 <__stext+22>       0xBF03        LDW   0x03,X         LDW   c_y,X
0x809b <__stext+24>       0xBE00        LDW   X,0x00         LDW   X,0x00
0x809d <__stext+26>       0xEE01        LDW   X,(0x01,X)     LDW   X,(0x01,X)
0x809f <__stext+28>       0x90F6        LD    A,(Y)          LD    A,(Y)
0x80a1 <__stext+30>       0xF7          LD    (X),A          LD    (X),A
0x80a2 <__stext+31>       0x5C          INCW  X              INCW  X
0x80a3 <__stext+32>       0x905C        INCW  Y              INCW  Y
0x80a5 <__stext+34>       0x90B303      CPW   Y,0x03         CPW   Y,c_y
0x80a8 <__stext+37>       0x26F5        JRNE  0x809f         JRNE  0x809f
0x80aa <__stext+39>       0xBE00        LDW   X,0x00         LDW   X,0x00
0x80ac <__stext+41>       0x1C0005      ADDW  X,#0x0005      ADDW  X,#0x0005
0x80af <__stext+44>       0x20DD        JRT   0x808e         JRT   0x808e
0x80b1 <__stext+46>       0xAE0000      LDW   X,#0x0000      LDW   X,#0x0000
0x80b4 <__stext+49>       0x2002        JRT   0x80b8         JRT   0x80b8
0x80b6 <__stext+51>       0xF7          LD    (X),A          LD    (X),A
0x80b7 <__stext+52>       0x5C          INCW  X              INCW  X
0x80b8 <__stext+53>       0xA30006      CPW   X,#0x0006      CPW   X,#0x0006
0x80bb <__stext+56>       0x26F9        JRNE  0x80b6         JRNE  0x80b6
0x80bd <__stext+58>       0xAE0100      LDW   X,#0x0100      LDW   X,#0x0100
0x80c0 <__stext+61>       0x2002        JRT   0x80c4         JRT   0x80c4
0x80c2 <__stext+63>       0xF7          LD    (X),A          LD    (X),A
0x80c3 <__stext+64>       0x5C          INCW  X              INCW  X
0x80c4 <__stext+65>       0xA30100      CPW   X,#0x0100      CPW   X,#0x0100
0x80c7 <__stext+68>       0x26F9        JRNE  0x80c2         JRNE  0x80c2
0x80c9 <__stext+70>       0xCD80CE      CALL  0x80ce         CALL  main
0x80cc <_exit>            0x20FE        JRT   0x80cc         JRT   _exit
```

图 3

```
main.c:9        while (1);
0x80ce <main>               0x20FE        JRT   0x80ce             JRT   main
rupt_vector.c:17        return;
0x80d0 <.nHandledInterrupt> 0x80          IRET
```

图 4

图 1 显示的是从内存地址 8000H 开始的中断向量表，中断向量表中的第 1 行 82008083 H 为复位后单片机运行的第 1 跳指令的地址。从表中可以看出，单片机复位后，将从 8083 H 开始运行。其它行的中断向量都指向同一个位置的中断服务程序 80D0H。

图 2 显示的是 3 个字节，前 2 个字节 8083H 为复位后的第 1 条指令的地址，第 3 个字节是一个常量 0，后面的启动代码要用到。

图 3 显示的是启动代码，启动代码中除了初始化堆栈指针外，就是初始化 RAM 单元。由于目前是一个空的框架，因此在初始化完堆栈指针（设置成 0FFFH）后，由于 8082H 单元

的内容为 0，因此程序就跳到了 80B1H，此处是一个循环，将 RAM 单元从 0 到 5 初始化成 0。然后由于寄存器 X 设置成 0100H，就直接通过 CALL main 进入 C 的 main()函数。

图 4 显示的是 main()函数和中断服务函数，main()函数对应的代码就是一个无限的循环，而中断服务函数就一条指令，即中断返回指令。

通过分析，可以看出用 C 语言编程时，比汇编语言编程时，就是多出了一段启动代码。

STM8 的 C 语言编程（2）－－ 变量空间的分配

采用 C 这样的高级语言，其实可以不用关心变量在存储器空间中是如何具体分配的。但如果了解如何分配，对编程还是有好处的，尤其是在调试时。

例如下面的程序定义了全局变量数组 buffer 和一个局部变量 i，在 RAM 中如何分配的呢？

```
/*  MAIN.C  file
 *
 *  Copyright  (c)  2002-2005  STMicroelectronics
 */

unsigned  char  buffer[10];      //  定义全局变量

main()
{
    unsigned  char  i;          //  定义局部变量

    for(i=0;i<10;i++)
    {
       buffer[i]  =  0x55;
    }
}
```

我们可以通过 DEBUG 中的反汇编窗口，看到如下的对应代码：

```
0x80ce <main>               0x88          PUSH  A                  PUSH  A
main.c:12       for(i=0;i<10;i++)
0x80cf <main+1>             0x0F01        CLR   (0x01,SP)          CLR   (i,SP)
main.c:14           buffer[i] = 0x55;
0x80d1 <main+3>             0x7B01        LD    A,(0x01,SP)        LD    A,(0x01,SP)
0x80d3 <main+5>             0x5F          CLRW  X                  CLRW  X
0x80d4 <main+6>             0x97          LD    XL,A               LD    XL,A
0x80d5 <main+7>             0xA655        LD    A,#0x55            LD    A,#0x55
0x80d7 <main+9>             0xE700        LD    (0x00,X),A         LD    (0x00,X),A
main.c:12       for(i=0;i<10;i++)
0x80d9 <main+11>            0x0C01        INC   (0x01,SP)          INC   (i,SP)
main.c:12       for(i=0;i<10;i++)
0x80db <main+13>            0x7B01        LD    A,(0x01,SP)        LD    A,(0x01,SP)
0x80dd <main+15>            0xA10A        CP    A,#0x0a            CP    A,#0x0a
0x80df <main+17>            0x25F0        JRC   0x80d1             JRC   0x80d1
main.c:16   }
0x80e1 <main+19>            0x84          POP   A                  POP   A
0x80e2 <main+20>            0x81          RET                      RET
```

从这段代码中可以看到，全局变量 buffer 被分配到空间从地址 0000H 到 0009H。而局部变量 i 则在堆栈空间中分配，通过 PUSH A 指令，将堆栈指针减 1，腾出一个字节的空间，而 SP+1 指向的空间就是分配给局部变量使用的空间。

由此可以得出初步的结论，对于全局变量，内存分配是从低地址 0000H 开始向上分配的。而局部变量则是在堆栈空间中分配。

另外从上一篇文章中，可以知道堆栈指针初始化时为 0FFFH。而根据 PUSH 指令的定义，当压栈后堆栈指针减 1。因此堆栈是从上往下使用的。

因此根据内存分配和堆栈使用规则，我们在程序设计时，不能定义过多的变量，免得没有空间给堆栈使用。换句话说，当定义变量时，一定要考虑到堆栈空间，尤其是那些复杂的系统，程序调用层数多，这样就会占用大量的堆栈空间。

总之，在单片机的程序设计时，由于 RAM 空间非常有限，要充分考虑到全局变量、局部变量、程序调用层数和中断服务调用对空间的占用。

STM8 的 C 语言编程（3） —— GPIO 输出

与前些日子写的用汇编语言进行的实验一样，从今天开始，要在 ST 的三合一开发板上，用 C 语言编写程序，进行一系列的实验。

首先当然从最简单的 LED 指示灯闪烁的实验开始。

开发板上的 LED1 接在 STM8 的 PD3 上，因此要将 PD3 设置成输出模式，为了提高高电平时的输出电流，要将其设置成推挽输出方式。这主要通过设置对应的 DDR/CR1/CR2 寄存器实现。

利用 ST 的开发工具，先生成一个 C 语言程序的框架，然后修改其中的 main.c，修改后的代码如下。

编译通过后，下载到开发板，运行程序，可以看到 LED1 在闪烁，且闪烁的频率为 5HZ。

```
/*  MAIN.C  file
 *
```

```
 * Copyright (c) 2002-2005 STMicroelectronics
 */

#include "STM8S207C_S.h"

// 函数功能：延时函数
// 输入参数：ms -- 要延时的毫秒数，这里假设 CPU 的主频为 2MHZ
// 输出参数：无
// 返 回 值：无
// 备    注：无
void DelayMS(unsigned int ms)
{
  unsigned char i;
  while(ms != 0)
  {
   for(i=0;i<250;i++)
   {
   }
   for(i=0;i<75;i++)
   {
   }
   ms--;
  }
}

// 函数功能：主函数
//         初始化 GPIO 端口 PD3，驱动 PD3 为高电平和低电平
// 输入参数：ms -- 要延时的毫秒数，这里假设 CPU 的主频为 2MHZ
// 输出参数：无
// 返 回 值：无
// 备    注：无
main()
{
  PD_DDR = 0x08;
  PD_CR1 = 0x08;            // 将 PD3 设置成推挽输出
  PD_CR2 = 0x00;
  while(1)
  {
```

```
    PD_ODR = PD_ODR | 0x08;   // 将 PD3 的输出设置成 1
    DelayMS(100);             // 延时 100MS
    PD_ODR = PD_ODR & 0xF7;   // 将 PD3 的输出设置成 0
    DelayMS(100);             // 延时 100MS
  }
}
```

需要注意的是，当生成完框架后，为了能方便使用 STM8 的寄存器名字，必须包括 STM8S207C_S.h，最好将该文件拷贝到 C:\Program Files\STMicroelectronics\st_toolset\include 目录下，拷贝到工程目录下。或者将该路径填写到该工程的 Settings…中的 C Compiler 选项 Preprocessor 的 Additional include 中，这样编译时才会找到该文件。

STM8 的 C 语言编程（4） —— GPIO 输出和输入

今天要进行的实验，是利用 GPIO 进行输入和输出。在 ST 的三合一开发板上，按键接在 GPIO 的 PD7 上，LED 接在 GPIO 的 PD3 上，因此我们要将 GPIO 的 PD7 初始化成输入，PD3 初始化成输出。

关于 GPIO 的引脚设置，主要是要初始化方向寄存器 DDR，控制寄存器 1（CR1）和控制寄存器 2（CR2），寄存器的每一位对应 GPIO 的每一个引脚。具体的设置功能定义如下：

DDR	CR1	CR2	引脚设置
0	0	0	悬浮输入
0	0	1	上拉输入
0	1	0	中断悬浮输入
0	1	1	中断上拉输入
1	0	0	开漏输出
1	1	0	推挽输出
1	X	1	输出（最快速度为 10MHZ）

另外，输出引脚对应的寄存器为 ODR，输入引脚对应的寄存器为 IDR。

下面的程序是检测按键的状态，当按键按下时，点亮 LED，当按键抬起时，熄灭 LED。

同样也是利用 ST 的开发工具，先生成一个 C 语言程序的框架，然后修改其中的 main.c，修改后的代码如下。

编译通过后，下载到开发板，运行程序，按下按键，LED 就点亮，抬起按键，LED 就熄灭了。
另外，要注意，将 STM8S207C_S.h 拷贝到当前项目的目录下。

```
// 程序描述：检测开发板上的按键，若按下，则点亮 LED，若抬起，则熄灭 LED
//          按键接在 MCU 的 GPIO 的 PD7 上
//          LED 接在 MCU 的 GPIO 的 PD3 上

#include "STM8S207C_S.h"

main()
{
  PD_DDR = 0x08;
  PD_CR1 = 0x08;            // 将 PD3 设置成推挽输出
  PD_CR2 = 0x00;

  while(1)                  // 进入无限循环
  {
    if((PD_IDR & 0x80) == 0x80)   // 读入 PD7 的引脚信号
    {
      PD_ODR = PD_ODR & 0xF7;   // 如果 PD7 为 1，则将 PD3 的输出设置成 0，熄灭 LED
    }
    else
    {
      PD_ODR = PD_ODR | 0x08;   // 否则，将 PD3 的输出设置成 1，点亮 LED
    }
  }
}
```

STM8 的 C 语言编程（5）——8 位定时器应用之一

在 STM8 单片机中，有多种定时器资源，既有 8 位的定时器，也有普通的 16 位定时器，还有高级的定时器。今天的实验是用最简单的 8 位定时器 TIM4 来进行延时，然后驱动 LED 闪烁。为了简单起见，这里是通过程序查询定时器是否产生更新事件，来判断定时器的延时是否结束。

同样还是利用 ST 的开发工具，生成一个 C 程序的框架，然后修改其中的 main.c，修改后的代码如下。

编译通过后，下载到开发板，运行程序，可以看到 LED 在闪烁，或者用示波器可以在 LED 引脚上看到方波。

在这里要特别提醒的是，从 ST 给的手册上看，这个定时器中的计数器是一个加 1 计数器，但本人在实验过程中感觉不太对，经过反复的实验，我认为应该是一个减 1 计数器（也许是我拿的手册不对，或许是理解上有误）。例如，当给定时器中的自动装载寄存器装入 255 时，产生的方波频率最小，就象下面代码中计算的那样，产生的方波频率为 30HZ 左右。若初始化时给自动装载寄存器装入 1，则产生的方波频率最大，大约为 3.9K 左右。也就是说实际的分频数为 ARR 寄存器的值+1。

```
//  程序描述：通过初始化定时器 4，进行延时，驱动 LED 闪烁
//          LED 接在 MCU 的 GPIO 的 PD3 上

#include "STM8S207C_S.h"

main()
{
  // 首先初始化 GPIO
  PD_DDR = 0x08;
  PD_CR1 = 0x08;          // 将 PD3 设置成推挽输出
  PD_CR2 = 0x00;

  // 然后初始化定时器 4
  TIM4_IER = 0x00;        // 禁止中断
  TIM4_EGR = 0x01;        // 允许产生更新事件
  TIM4_PSCR = 0x07;       // 计数器时钟=主时钟/128=2MHZ/128
                          // 相当于计数器周期为 64uS
        TIM4_ARR = 255;         // 设定重装载时的寄存器值，255 是最大值
  TIM4_CNTR = 255;        // 设定计数器的初值
                  // 定时周期=(ARR+1)*64=16320uS
  TIM4_CR1 = 0x01;        // b0 = 1,允许计数器工作
                  // b1 = 0,允许更新
                  // 设置控制器，启动定时器
```

```
  while(1)                  // 进入无限循环
  {
    while((TIM4_SR1 & 0x81) == 0x00);      // 等待更新标志
    TIM4_SR1 = 0x00;                       // 清除更新标志
    PD_ODR = PD_ODR ^ 0x08;                // LED 驱动信号取反
                                  // LED 闪烁频率=2MHZ/128/255/2=30.63
  }
}
```

STM8 的 C 语言编程（6）——8 位定时器应用之二

今天进行的实验依然是用定时器 4，只不过改成了用中断方式来实现，由定时器 4 的中断服务程序来驱动 LED 的闪烁。

实现中断方式的关键点有几个，第一个关键点就是要打开定时器 4 的中断允许位，在定时器 4 的 IER 寄存器中有定义。第二个关键点，就是打开 CPU 的全局中断允许位，在汇编语言中，就是执行 RIM 指令，在 C 语言中，用下列语句实现：

```
_asm("rim");
```

第 3 个关键点就是中断服务程序的框架或写法，中断服务程序的写法如下：

```
@far @interrupt void TIM4_UPD_OVF_IRQHandler (void)
{
   // 下面是中断服务程序的实体
}
```

第 4 个关键点就是要设置中断向量，即将中断服务程序的入口填写到中断向量表中，如下所示，将 IRQ23 对应的中断服务程序的入口填写成 TIM4_UPD_OVF_IRQHandler

```
struct interrupt_vector const _vectab[] = {
    {0x82, (interrupt_handler_t)_stext}, /* reset */
    {0x82, NonHandledInterrupt}, /* trap  */
    {0x82, NonHandledInterrupt}, /* irq0  */
    {0x82, NonHandledInterrupt}, /* irq1  */
    {0x82, NonHandledInterrupt}, /* irq2  */
    {0x82, NonHandledInterrupt}, /* irq3  */
    {0x82, NonHandledInterrupt}, /* irq4  */
    {0x82, NonHandledInterrupt}, /* irq5  */
    {0x82, NonHandledInterrupt}, /* irq6  */
```

```
    {0x82, NonHandledInterrupt}, /* irq7  */
    {0x82, NonHandledInterrupt}, /* irq8  */
    {0x82, NonHandledInterrupt}, /* irq9  */
    {0x82, NonHandledInterrupt}, /* irq10 */
    {0x82, NonHandledInterrupt}, /* irq11 */
    {0x82, NonHandledInterrupt}, /* irq12 */
    {0x82, NonHandledInterrupt}, /* irq13 */
    {0x82, NonHandledInterrupt}, /* irq14 */
    {0x82, NonHandledInterrupt}, /* irq15 */
    {0x82, NonHandledInterrupt}, /* irq16 */
    {0x82, NonHandledInterrupt}, /* irq17 */
    {0x82, NonHandledInterrupt}, /* irq18 */
    {0x82, NonHandledInterrupt}, /* irq19 */
    {0x82, NonHandledInterrupt}, /* irq20 */
    {0x82, NonHandledInterrupt}, /* irq21 */
    {0x82, NonHandledInterrupt}, /* irq22 */
  {0x82, TIM4_UPD_OVF_IRQHandler},/* irq23 */
    {0x82, NonHandledInterrupt}, /* irq24 */
    {0x82, NonHandledInterrupt}, /* irq25 */
    {0x82, NonHandledInterrupt}, /* irq26 */
    {0x82, NonHandledInterrupt}, /* irq27 */
    {0x82, NonHandledInterrupt}, /* irq28 */
    {0x82, NonHandledInterrupt}, /* irq29 */
};
```

解决了以上 4 个关键点，我们就能很轻松地用 C 语言实现中断服务了。

同样还是利用 ST 的开发工具，生成一个 C 程序的框架，然后修改其中的 main.c，修改后的代码如下。另外还要修改 stm8_interrupt_vector.c。

编译通过后，下载到开发板，运行程序，可以看到 LED 在闪烁，或者用示波器可以在 LED 引脚上看到方波。

修改后的 main.c 如下：

```
//  程序描述：通过初始化定时器 4，以中断方式驱动 LED 闪烁
//            LED 接在 MCU 的 GPIO 的 PD3 上

#include  "STM8S207C_S.h"

main()
{
```

```
  // 首先初始化 GPIO
  PD_DDR = 0x08;
  PD_CR1 = 0x08;          // 将 PD3 设置成推挽输出
  PD_CR2 = 0x00;

  // 然后初始化定时器 4
  TIM4_IER = 0x00;        // 禁止中断
  TIM4_EGR = 0x01;        // 允许产生更新事件
  TIM4_PSCR = 0x07;       // 计数器时钟=主时钟/128=2MHZ/128
                          // 相当于计数器周期为 64uS
        TIM4_ARR = 255;   // 设定重装载时的寄存器值，255 是最大值
  TIM4_CNTR = 255;        // 设定计数器的初值
                  // 定时周期=(ARR+1)*64=16320uS
  TIM4_CR1 = 0x01;        // b0 = 1,允许计数器工作
                  // b1 = 0,允许更新
                  // 设置控制器，启动定时器
  TIM4_IER = 0x01;        // 允许更新中断
  _asm("rim");            // 允许 CPU 全局中断

  while(1)                // 进入无限循环
  {
  }
}

// 函数功能：定时器 4 的更新中断服务程序
// 输入参数：无
// 输出参数：无
// 返 回 值：无
@far @interrupt void TIM4_UPD_OVF_IRQHandler (void)
{
   TIM4_SR1 = 0x00;       // 清除更新标志
   PD_ODR = PD_ODR ^ 0x08;  // LED 驱动信号取反
                  //LED 闪烁频率=2MHZ/128/255/2=30.63
}
```

修改后的 stm8_interrupt_vector.c 如下：

```
/*  BASIC INTERRUPT VECTOR TABLE FOR STM8 devices
 *  Copyright (c) 2007 STMicroelectronics
 */
```

```
typedef void @far (*interrupt_handler_t)(void);

struct interrupt_vector {
	unsigned char interrupt_instruction;
	interrupt_handler_t interrupt_handler;
};

@far @interrupt void NonHandledInterrupt (void)
{
	/* in order to detect unexpected events during development,
	   it is recommended to set a breakpoint on the following instruction
	*/
	return;
}

extern void _stext();	/* startup routine */
extern @far @interrupt void TIM4_UPD_OVF_IRQHandler (void);

struct interrupt_vector const _vectab[] = {
	{0x82, (interrupt_handler_t)_stext}, /* reset */
	{0x82, NonHandledInterrupt}, /* trap  */
	{0x82, NonHandledInterrupt}, /* irq0  */
	{0x82, NonHandledInterrupt}, /* irq1  */
	{0x82, NonHandledInterrupt}, /* irq2  */
	{0x82, NonHandledInterrupt}, /* irq3  */
	{0x82, NonHandledInterrupt}, /* irq4  */
	{0x82, NonHandledInterrupt}, /* irq5  */
	{0x82, NonHandledInterrupt}, /* irq6  */
	{0x82, NonHandledInterrupt}, /* irq7  */
	{0x82, NonHandledInterrupt}, /* irq8  */
	{0x82, NonHandledInterrupt}, /* irq9  */
	{0x82, NonHandledInterrupt}, /* irq10 */
	{0x82, NonHandledInterrupt}, /* irq11 */
	{0x82, NonHandledInterrupt}, /* irq12 */
	{0x82, NonHandledInterrupt}, /* irq13 */
	{0x82, NonHandledInterrupt}, /* irq14 */
	{0x82, NonHandledInterrupt}, /* irq15 */
```

```
    {0x82, NonHandledInterrupt}, /* irq16 */
    {0x82, NonHandledInterrupt}, /* irq17 */
    {0x82, NonHandledInterrupt}, /* irq18 */
    {0x82, NonHandledInterrupt}, /* irq19 */
    {0x82, NonHandledInterrupt}, /* irq20 */
    {0x82, NonHandledInterrupt}, /* irq21 */
    {0x82, NonHandledInterrupt}, /* irq22 */
  {0x82, TIM4_UPD_OVF_IRQHandler},/* irq23 */
    {0x82, NonHandledInterrupt}, /* irq24 */
    {0x82, NonHandledInterrupt}, /* irq25 */
    {0x82, NonHandledInterrupt}, /* irq26 */
    {0x82, NonHandledInterrupt}, /* irq27 */
    {0x82, NonHandledInterrupt}, /* irq28 */
    {0x82, NonHandledInterrupt}, /* irq29 */
};
```

STM8 的 C 语言编程（7）——16 位定时器的中断应用

在 STM8 中，除了有 8 位的定时器外，还有 16 位的定时器。今天进行的实验就是针对 16 位定时器 2 来进行的。除了计数单元为 16 位的，其它设置与前面 8 位的定时器基本一样。下面的程序也是采样中断方式，由定时器 2 的中断服务程序来驱动 LED 的闪烁。
具体的程序代码如下，其它注意点见上一篇，另外要注意别忘了修改相应的中断向量。

```
//  程序描述：通过初始化定时器 2，以中断方式驱动 LED 闪烁
//          LED 接在 MCU 的 GPIO 的 PD3 上

#include "STM8S207C_S.h"

main()
{
  // 首先初始化 GPIO
  PD_DDR = 0x08;
  PD_CR1 = 0x08;          // 将 PD3 设置成推挽输出
  PD_CR2 = 0x00;

  // 然后初始化定时器 4
  TIM2_IER = 0x00;        // 禁止中断
```

```
  TIM2_EGR = 0x01;          // 允许产生更新事件
  TIM2_PSCR = 0x01;         // 计数器时钟=主时钟/128=2MHZ/2
                                    // 相当于计数器周期为 1uS

      // 设定重装载时的寄存器值

      // 注意必须保证先写入高 8 位，再写入低 8 位
         TIM2_ARRH = 0xEA;          // 设定重装载时的寄存器的高 8 位
  TIM2_ARRL = 0x60;

  TIM2_CNTRH = 0xEA;         // 设定计数器的初值
  TIM2_CNTRL = 0x60;
                    // 定时周期=1*60000=60000uS=60ms

  TIM2_CR1 = 0x01;          // b0 = 1,允许计数器工作
                    // b1 = 0,允许更新
                    // 设置控制器，启动定时器
  TIM2_IER = 0x01;          // 允许更新中断
  _asm("rim");              // 允许 CPU 全局中断

  while(1)                  // 进入无限循环
  {
  }
}

// 函数功能：定时器 4 的更新中断服务程序
// 输入参数：无
// 输出参数：无
// 返 回 值：无
@far @interrupt void TIM2_UPD_IRQHandler (void)
{
   TIM2_SR1 = 0x00;         // 清除更新标志
   PD_ODR = PD_ODR ^ 0x08;  // LED 驱动信号取反
                    //LED 闪烁频率=2MHZ/2/60000/2=8.3
}
```

STM8 的 C 语言编程（8）－－ UART 应用

串口通讯也是单片机应用中经常要用到，今天的实验就是利用 STM8 的 UART 资源，来进行串口通讯的实验。

实验程序的功能是以中断方式接收串口数据，然后将接收到的数据以查询方式发送到串口。

程序代码如下，首先要对 STM8 的 UART 进行初始化，初始化时要注意的是波特率寄存器的设置，当求出一个波特率的分频系数（一个 16 位的数）后，要将高 4 位和低 4 位写到 BRR2 中，而将中间的 8 位写到 BRR1 中，并且必须是先写 BRR2，再写 BRR1。

同样也是利用 ST 的开发工具，生成一个 C 语言的框架，然后修改其中的 main.c，同时由于需要用到中断服务，因此还要修改 stm8_interrupt_vector.c。

修改后，编译连接，然后下载到开发板上，再做一根与 PC 机相连的线，把开发板的串口与 PC 机的串口连接起来，注意，2、3 脚要交叉。

在 PC 机上运行超级终端，设置波特率为 9600，然后每按下一个按键，屏幕上就显示对应的字符。

修改后的 main.c 和 stm8_interrupt_vector.c 如下：

```
//  程序描述：初始化 UART，以中断方式接收字符，以查询方式发送
//            UART 通讯参数：9600bps,8 位数据，1 位停止位，无校验

#include  "STM8S207C_S.h"


// 函数功能：初始化 UART
// 输入参数：无
// 输出参数：无
// 返 回 值：无
// 备    注：无
void  UART3_Init(void)
{

  LINUART_CR2  =  0;          // 禁止 UART 发送和接收
  LINUART_CR1  =  0;          // b5  =  0,允许 UART
                    // b2  =  0,禁止校验

  LINUART_CR3  =  0;          // b5,b4  =  00,1 个停止位
```

```
// 设置波特率，必须注意以下几点：
//   (1) 必须先写 BRR2
//   (2) BRR1 存放的是分频系数的第 11 位到第 4 位，
//   (3) BRR2 存放的是分频系数的第 15 位到第 12 位，和第 3 位到第 0 位
// 例如对于波特率位 9600 时，分频系数=2000000/9600=208
// 对应的十六进制数为 00D0，BBR1=0D,BBR2=00
  LINUART_BRR2 = 0;
  LINUART_BRR1 = 0x0d;          // 实际的波特率分频系数为 00D0(208)
                                // 对应的波特率为 2000000/208=9600

  LINUART_CR2 = 0x2C;           // b3 = 1,允许发送
                                // b2 = 1,允许接收
                                // b5 = 1,允许产生接收中断
}

// 函数功能：从 UART3 发送一个字符
// 输入参数：ch -- 要发送的字符
// 输出参数：无
// 返 回 值：无
// 备   注：无
void UART3_SendChar(unsigned char ch)
{
  while((LINUART_SR & 0x80) == 0x00); // 若发送寄存器不空，则等待
  LINUART_DR = ch;                    // 将要发送的字符送到数据寄存器
}

main()
{
  // 首先初始化 UART3
  UART3_Init();
  _asm("rim");              // 允许 CPU 全局中断

  while(1)                  // 进入无限循环
  {
  }
}
```

```
// 函数功能：UART3 的接收中断服务程序
// 输入参数：无
// 输出参数：无
// 返 回 值：无
@far @interrupt void UART3_Recv_IRQHandler (void)
{
  unsigned char ch;

  ch = LINUART_DR;          // 读入接收到的字符
  UART3_SendChar(ch);       // 将字符发送出去
}



/*    BASIC INTERRUPT VECTOR TABLE FOR STM8 devices
 *    Copyright (c) 2007 STMicroelectronics
 */

typedef void @far (*interrupt_handler_t)(void);

struct interrupt_vector {
      unsigned char interrupt_instruction;
      interrupt_handler_t interrupt_handler;
};

@far @interrupt void NonHandledInterrupt (void)
{
      /* in order to detect unexpected events during development,
         it is recommended to set a breakpoint on the following instruction
      */
      return;
}

extern void _stext();     /* startup routine */
extern @far @interrupt void UART3_Recv_IRQHandler();

struct interrupt_vector const _vectab[] = {
      {0x82, (interrupt_handler_t)_stext}, /* reset */
```

```
    {0x82, NonHandledInterrupt}, /* trap  */
    {0x82, NonHandledInterrupt}, /* irq0  */
    {0x82, NonHandledInterrupt}, /* irq1  */
    {0x82, NonHandledInterrupt}, /* irq2  */
    {0x82, NonHandledInterrupt}, /* irq3  */
    {0x82, NonHandledInterrupt}, /* irq4  */
    {0x82, NonHandledInterrupt}, /* irq5  */
    {0x82, NonHandledInterrupt}, /* irq6  */
    {0x82, NonHandledInterrupt}, /* irq7  */
    {0x82, NonHandledInterrupt}, /* irq8  */
    {0x82, NonHandledInterrupt}, /* irq9  */
    {0x82, NonHandledInterrupt}, /* irq10 */
    {0x82, NonHandledInterrupt}, /* irq11 */
    {0x82, NonHandledInterrupt}, /* irq12 */
    {0x82, NonHandledInterrupt}, /* irq13 */
    {0x82, NonHandledInterrupt}, /* irq14 */
    {0x82, NonHandledInterrupt}, /* irq15 */
    {0x82, NonHandledInterrupt}, /* irq16 */
    {0x82, NonHandledInterrupt}, /* irq17 */
    {0x82, NonHandledInterrupt}, /* irq18 */
    {0x82, NonHandledInterrupt}, /* irq19 */
    {0x82, NonHandledInterrupt}, /* irq20 */
    {0x82, UART3_Recv_IRQHandler}, /* irq21 */
    {0x82, NonHandledInterrupt}, /* irq22 */
    {0x82, NonHandledInterrupt}, /* irq23 */
    {0x82, NonHandledInterrupt}, /* irq24 */
    {0x82, NonHandledInterrupt}, /* irq25 */
    {0x82, NonHandledInterrupt}, /* irq26 */
    {0x82, NonHandledInterrupt}, /* irq27 */
    {0x82, NonHandledInterrupt}, /* irq28 */
    {0x82, NonHandledInterrupt}, /* irq29 */
};
```

STM8 与汇编语言（9）——EEPROM 应用

EEPROM 是单片机应用系统中经常会用到的存储器，它主要用来保存一些掉电后需要保持不变的数据。在以前的单片机系统中，通常都是在单片机外面再扩充一个 EEPROM 芯片，这种方法除了增加成本外，也降低了可靠性。现在，很多单片机的公司都推出了集成有小容量 EEPROM 的单片机，这样就方便了使用，降低了成本，提高了可靠性。
STM8 单片机芯片内部也集成有 EEPROM，容量从 640 字节到 2K 字节。最为特色的是，在 STM8 单片机中，对 EEPROM 的访问就象常规的 RAM 一样，非常方便。EEPROM 的地址空间与内存是统一编址的，地址从 004000H 开始，大小根据不同的芯片型号而定。
下面的实验程序，就是先给 EEPROM 中的第一个单元 004000H 写入 55H，然后再读到全局变量 ch 中。
同样还是利用 ST 的开发工具，生成一个 C 语言程序的框架，然后修改其中的 main.c，修改后的代码如下。

```
//  程序描述：对芯片内部的 EEPROM 存储单元进行实验

#include "STM8S207C_S.h"

unsigned char ch;

main()
{
  unsigned char *p;

  p = (unsigned char *)0x4000;         // 指针 p 指向芯片内部的 EEPROM 第一个单元

  // 对数据 EEPROM 进行解锁
  do
  {
    FLASH_DUKR = 0xae;                 // 写入第一个密钥
    FLASH_DUKR = 0x56;                 // 写入第二个密钥
  } while((FLASH_IAPSR & 0x08) == 0);      // 若解锁未成功，则重新再来

  *p = 0xaa;                           // 写入第一个字节
  while((FLASH_IAPSR & 0x04) == 0);        // 等待写操作成功

  ch = *p;                             // 将写入的内容读到变量 ch 中

  while(1)
  {
```

```
    ;
  }
}
```

这里要注意的是，2 个密钥的顺序，与 STM8 的用户手册上是相反的，如果按照手册上的顺序，就会停留在 do…while 循环中。具体原因，也不是很清楚，也可能是我拿到的手册（中文和英文的都一样）太旧了，或者是理解有误。

另外，上面的实验程序中，ch 不能为局部变量，否则的话，在调试环境中跟踪 ch 变量时，显示的结果就不对，通过反汇编，我觉得是编译有问题，当定义成局部变量时，ch = *p 的汇编代码如下：

```
main.c:23        ch = *p;                        // 将写入的内容读到变量 ch 中
0x80f0 <main+34>   0x7B01       LD    A,(0x01,SP)       LD    A,(0x01,SP)
0x80f2 <main+36>   0x97         LD    XL,A              LD    XL,A
0x80f3 <main+37>   0x1E02       LDW   X,(0x02,SP)       LDW   X,(0x02,SP)
0x80f5 <main+39>   0xF6         LD    A,(X)             LD    A,(X)
0x80f6 <main+40>   0x97         LD    XL,A              LD    XL,A
```

如果将 ch 定义成全局变量，则汇编代码为：

```
main.c:22        ch = *p;                        // 将写入的内容读到变量 ch 中
0x80ef <main+33>   0x1E01       LDW   X,(0x01,SP)       LDW   X,(0x01,SP)
0x80f1 <main+35>   0xF6         LD    A,(X)             LD    A,(X)
0x80f2 <main+36>   0xB700       LD    0x00,A            LD    0x00,A
```

这一段代码的分析仅供参考，本人使用的开发环境为 STVD4.1.0，编译器版本号为：COSMIC 的 CxSTM84.2.4。

STM8 的 C 语言编程（10）—— 修改 CPU 的时钟

在有些单片机的应用系统中，并不需要 CPU 运行在多高的频率。在低频率下运行，芯片的功耗会大大下降。STM8 单片机在运行过程中，可以随时修改 CPU 运行时钟频率，非常方便。实现这一功能，主要涉及到时钟分频寄存器（CLK_CKDIVR）。

时钟分频寄存器是一个 8 位的寄存器，高 3 位保留，位 4 和位 3 用于定义高速内部时钟的预分频，而位 2 到位 0 则用于 CPU 时钟的分频。这 5 位的详细定义如下：

位 4　位 3　高速内部时钟的分频系数

0　0　1

0　1　2

1　0　4

1 1 8

位 2 位 1 位 0 CPU 时钟的分频系数

0	0	0	1
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

假设我们使用内部的高速 RC 振荡器，其频率为 16MHZ，当位 4 为 0，位 3 为 1 时，则内部高速时钟的分频系数为 2，因此输出的主时钟为 8MHZ。当位 2 为 0，位 1 为 1，位 0 为 0 时，CPU 时钟的分频系数为 4，即 CPU 时钟=主时钟/4=2MHZ。

下面的实验程序首先将 CPU 的运行时钟设置在 8MHZ，然后快速闪烁 LED 指示灯。接着，通过修改主时钟的分频系数和 CPU 时钟的分频系数，将 CPU 时钟频率设置在 500KHZ，然后再慢速闪烁 LED 指示灯。通过观察 LED 指示灯的闪烁频率，可以看到，同样的循环代码，由于 CPU 时钟频率的改变，闪烁频率和时间长短都发生了变化。

同样还是利用 ST 的开发工具，生成一个 C 语言程序的框架，然后修改其中的 main.c，修改后的代码如下。修改后的代码编译连接后，就可以下载到开发板上，运行后会看到 LED 的闪烁频率有明显的变化。

```
//  程序描述：通过修改 CPU 时钟的分频系数，来改变 CPU 的运行速度

#include "STM8S207C_S.h"

// 函数功能：延时函数
// 输入参数：ms -- 要延时的毫秒数，这里假设 CPU 的主频为 2MHZ
// 输出参数：无
// 返 回 值：无
// 备    注：无
void DelayMS(unsigned int ms)
{
  unsigned char i;
  while(ms != 0)
  {
    for(i=0;i<250;i++)
```

```
    {
    }
    for(i=0;i<75;i++)
    {
    }
    ms--;
  }
}

main()
{
  int  i;

  PD_DDR  =  0x08;
  PD_CR1  =  0x08;            // 将 PD3 设置成推挽输出
  PD_CR2  =  0x00;

  CLK_SWR  =  0xE1;           // 选择芯片内部的 16MHZ 的 RC 振荡器为主时钟

  for(;;)                     // 进入无限循环
  {
    // 下面设置 CPU 时钟分频器，使得 CPU 时钟=主时钟
    // 通过发光二极管，可以看出，程序运行的速度确实明显提高了
    CLK_CKDIVR  =  0x08;      // 主时钟  =  16MHZ  /  2
                              // CPU 时钟  =  主时钟  =  8MHZ
    for(i=0;i<10;i++)
    {
      PD_ODR  =  0x08;
      DelayMS(100);
      PD_ODR  =  0x00;
      DelayMS(100);
    }

    // 下面设置 CPU 时钟分频器，使得 CPU 时钟=主时钟/4
    // 通过发光二极管，可以看出，程序运行的速度确实明显下降了
    CLK_CKDIVR  =  0x1A;      // 主时钟  =  16MHZ  /  8
                              // CPU 时钟  =  主时钟  /  4   =  500KHZ
    for(i=0;i<10;i++)
```

```
    {
      PD_ODR = 0x08;
      DelayMS(100);
      PD_ODR = 0x00;
      DelayMS(100);
    }
  }
}
```

STM8 的 C 语言编程（11）—— 切换时钟源

STM8 单片机的时钟源非常丰富，芯片内部既有 16MHZ 的高速 RC 振荡器，也有 128KHZ 的低速 RC 振荡器，外部还可以接一个高速的晶体振荡器。在系统运行过程中，可以根据需要，自由地切换。单片机复位后，首先采用的是内部的高速 RC 振荡器，且分频系数为 8，因此 CPU 的上电运行的时钟频率为 2MHZ。

切换时钟源，主要涉及到的寄存器有：主时钟切换寄存器 CLK_SWR 和切换控制寄存器 CLK_SWCR。

主时钟切换寄存器的复位值为 0xe1，表示切换到内部的高速 RC 振荡器上。当往该寄存器写入 0xb4 时，表示切换到外部的高速晶体振荡器上。

在实际切换过程中，应该先将切换控制寄存器中的 SWEN（第 1 位）设置成 1，然后设置 CLK_SWCR 的值，最后要判断切换控制寄存器中的 SWIF 标志是否切换成功。

下面的实验程序首先将主时钟源切换到外部的晶体振荡器上，振荡频率为 8MHZ，然后，然后快速闪烁 LED 指示灯。接着，将主时钟源又切换到内部的振荡器上，振荡频率为 2MHZ，然后再慢速闪烁 LED 指示灯。通过观察 LED 指示灯的闪烁频率，可以看到，同样的循环代码，由于主时钟源的改变的改变，闪烁频率和时间长短都发生了变化。

同样还是利用 ST 的开发工具，生成一个 C 语言程序的框架，然后修改其中的 main.c，修改后的代码如下。

```
//  程序描述：通过切换 CPU 的主时钟源，来改变 CPU 的运行速度

#include "STM8S207C_S.h"

// 函数功能：延时函数
// 输入参数：ms -- 要延时的毫秒数，这里假设 CPU 的主频为 2MHZ
// 输出参数：无
// 返 回 值：无
```

```
// 备    注：无
void DelayMS(unsigned int ms)
{
  unsigned char i;
  while(ms != 0)
  {
   for(i=0;i<250;i++)
   {
   }
   for(i=0;i<75;i++)
   {
   }
   ms--;
  }
}

main()
{
  int i;

  // 将 PD3 设置成推挽输出，以便推动 LED
  PD_DDR = 0x08;
  PD_CR1 = 0x08;
  PD_CR2 = 0x00;

  // 启动外部高速晶体振荡器
  CLK_ECKR = 0x01;              // 允许外部高速振荡器工作
  while((CLK_ECKR & 0x02) == 0x00); // 等待外部高速振荡器准备好

  // 注意，复位后 CPU 的时钟源来自内部的 RC 振荡器

  for(;;)                 // 进入无限循环
  {

   // 下面将 CPU 的时钟源切换到外部的高速晶体振荡器上，在开发板上的频率为 8MHZ
   // 通过发光二极管，可以看出，程序运行的速度确实明显提高了
   CLK_SWCR = CLK_SWCR | 0x02;   // SWEN <- 1
   CLK_SWR = 0xB4;             // 选择芯片外部的高速振荡器为主时钟
```

```
    while((CLK_SWCR & 0x08) == 0); // 等待切换成功

    CLK_SWCR = CLK_SWCR & 0xFD;    // 清除切换标志

    for(i=0;i<10;i++)              // LED 高速闪烁 10 次
    {
      PD_ODR = 0x08;
      DelayMS(100);
      PD_ODR = 0x00;
      DelayMS(100);
    }

    // 下面将 CPU 的时钟源切换到内部的 RC 振荡器上，由于 CLK_CKDIVR 的复位值为 0
x18
    // 所以 16MHZ 的 RC 振荡器要经过 8 分频后才作为主时钟，因此频率为 2MHZ
    // 通过发光二极管，可以看出，程序运行的速度确实明显下降了
    CLK_SWCR = CLK_SWCR | 0x02;    // SWEN <- 1
    CLK_SWR = 0xE1;                // 选择 HSI 为主时钟源
    while((CLK_SWCR & 0x08) == 0); // 等待切换成功

    CLK_SWCR = CLK_SWCR & 0xFD;    // 清除切换标志

    for(i=0;i<10;i++)              // LED 低速闪烁 10 次
    {
      PD_ODR = 0x08;
      DelayMS(100);
      PD_ODR = 0x00;
      DelayMS(100);
    }
  }
}
```

STM8 的 C 语言编程（12）—— AD 转换

在许多的单片机应用系统中，都需要 A/D 转换器，将模拟量转换成数字量。在 STM8 单片机中，提供的是 10 位的 A/D，通道数随芯片不同而不同，少的有 4 个通道，多的则有 16 个通道。

下面的实验程序首先对 A/D 输入进行采样，然后将采样结果的高 8 位（丢弃最低的 2 位），作为延时参数去调用延时子程序，然后再去驱动 LED 控制信号。因此不同的采样值，决定了 LED 的闪烁频率。当旋转 ST 三合一开发板上的电位器时，可以看到 LED 的闪烁频率发生变化。

同样还是利用 ST 的开发工具，生成一个 C 语言程序的框架，然后修改其中的 main.c，修改后的代码如下。

```
//  程序描述：通过 AD 模块，采样电位器的电压，改变 LED 的闪烁频率

#include "STM8S207C_S.h"

// 函数功能：延时函数
// 输入参数：ms -- 要延时的毫秒数，这里假设 CPU 的主频为 2MHZ
// 输出参数：无
// 返 回 值：无
// 备    注：无
void DelayMS(unsigned int ms)
{
  unsigned char i;
  while(ms != 0)
  {
    for(i=0;i<250;i++)
    {
    }
    for(i=0;i<75;i++)
    {
    }
    ms--;
  }
}

main()
{
  int i;
```

```
  // 将 PD3 设置成推挽输出，以便推动 LED
  PD_DDR = 0x08;
  PD_CR1 = 0x08;
  PD_CR2 = 0x00;

  // 初始化 A/D 模块
  ADC_CR2 = 0x00;          // A/D 结果数据左对齐
  ADC_CR1 = 0x00;          // ADC 时钟=主时钟/2=1MHZ
                           // ADC 转换模式=单次
                           // 禁止 ADC 转换
  ADC_CSR = 0x03;          // 选择通道 3
  ADC_TDRL = 0x20;

  for(;;)                  // 进入无限循环
  {
    ADC_CR1 = 0x01;        // CR1 寄存器的最低位置 1，使能 ADC 转换
    for(i=0;i<100;i++);    // 延时一段时间，至少 7uS，保证 ADC 模块的上电完成
    ADC_CR1 = ADC_CR1 | 0x01; // 再次将 CR1 寄存器的最低位置 1
                           // 使能 ADC 转换
    while((ADC_CSR & 0x80) == 0); // 等待 ADC 结束

    i = ADC_DRH;           // 读出 ADC 结果的高 8 位
    DelayMS(i);            // 延时一段时间

    PD_ODR = PD_ODR ^ 0x08;  // 将 PD3 反相
  }
}
```

STM8 的 C 语言编程（13）—— 蜂鸣器

蜂鸣器是现在单片机应用系统中很常见的，常用于实现报警功能。为此 STM8 特别集成了蜂鸣器模块，应用起来非常方便。

在应用蜂鸣器模块时，首先要打开片内的低速 RC 振荡器（当然也能使用外部的高速时钟），其频率为 128KHZ。然后通过设置蜂鸣器控制寄存器 BEEP_CSR 中的 BEEPDIV[4:0]来获取 8KHZ 的时钟，再通过 BEEPSEL 最终产生 1KHZ 或 2KHZ 或 4KHZ 的蜂鸣器时钟，最后使能该寄存器中的 BEEPEN 位，产生蜂鸣器的输出。
下面的实验程序首先初始化低速振荡器，然后启动蜂鸣器，再延时 2.5 秒，然后关闭蜂鸣器。同样还是利用 ST 的开发工具，生成一个汇编程序的框架，然后修改其中的 main.c，修改后的代码如下。

```
//  程序描述：启动单片机中的蜂鸣器模块

#include "STM8S207C_S.h"

// 函数功能：延时函数
// 输入参数：ms -- 要延时的毫秒数，这里假设 CPU 的主频为 2MHZ
// 输出参数：无
// 返 回 值：无
// 备    注：无
void DelayMS(unsigned int ms)
{
  unsigned char i;
  while(ms != 0)
  {
   for(i=0;i<250;i++)
   {
   }
   for(i=0;i<75;i++)
   {
   }
   ms--;
  }
}

main()
{
  int i;

  CLK_ICKR = CLK_ICKR | 0x08;     // 打开芯片内部的低速振荡器 LSI
  while((CLK_ICKR & 0x10) == 0);  // 等待振荡器稳定
```

```
  // 通过设置蜂鸣器控制寄存器，来打开蜂鸣器的功能
  // 蜂鸣器控制寄存器的设置：
  // BEEPDIV[1:0] = 00
  // BEEPDIV[4:0] = 0e
  // BEEPEN       = 1
  // 蜂鸣器的输出频率 = Fls / ( 8 * (BEEPDIV + 2) )= 128K / (8 * 16) = 1K
  BEEP_CSR = 0x2e;

  for(i=0;i<10;i++)
  {
    DelayMS(250);
  }

  BEEP_CSR = 0x1e;          // 关闭蜂鸣器

  while(1);
}
```

STM8 的 C 语言编程（14）—— PWM

在单片机应用系统中，也常常会用到 PWM 信号输出，例如电机转速的控制。现在很多高档的单片机也都集成了 PWM 功能模块，方便用户的应用。

对于 PWM 信号，主要涉及到两个概念，一个就是 PWM 信号的周期或频率，另一个就是 PWM 信号的占空比。例如一个频率为 1KHZ，占空比为 30%，有效信号为 1 的 PWM 信号，在用示波器测量时，就是高电平的时间为 300uS，低电平的时间为 700uS 的周期波形。

在单片机中实现 PWM 信号的功能模块，实际上就是带比较器的计数器模块。首先该计数器循环计数，例如从 0 到 N，那么这个 N 就决定了 PWM 的周期，PWM 周期=（N+1）*计数器时钟的周期。在计数器模块中一定还有一个比较器，比较器有 2 个输入，一个就是计数器的当前值，另一个是可以设置的数，这个数来自一个比较寄存器。当计数器的值小于比较寄存器的值时，输出为 1（可以设置为 0），当计数器的值大于或等于比较寄存器的值时，输出为 0（也可设置为 1，与前面对应）。

了解了这个基本原理后，我们就可以使用 STM8 单片机中的 PWM 模块了。下面的实验程序首先将定时器 2 的通道 2 设置成 PWM 输出方式，然后通过设置自动装载寄存器 TIM2_CCR2，决定 PWM 信号的周期。在程序的主循环中，循环修改占空比，先是从 0 逐渐递增到 128，然后再从 128 递减到 0。

当把下面的程序在 ST 的三合一板上运行时，可以看到发光二极管 LD1 逐渐变亮，然后又逐渐变暗，就这样循环往复。如果用示波器看，可以看到驱动 LD1 的信号波形的占空比从 0 变到 50%，然后又从 50%变到 0。
同样还是利用 ST 的开发工具，生成一个 C 语言程序的框架，然后修改其中的 main.c，修改后的代码如下。

```
//  程序描述：用 PWM 输出驱动 LED

#include "STM8S207C_S.h"

void CLK_Init(void);
void TIM_Init(void);

// 函数功能：延时函数
// 输入参数：ms -- 要延时的毫秒数，这里假设 CPU 的主频为 2MHZ
// 输出参数：无
// 返 回 值：无
// 备    注：无
void DelayMS(unsigned int ms)
{
  unsigned char i;
  while(ms != 0)
  {
    for(i=0;i<250;i++)
    {
    }
    for(i=0;i<75;i++)
    {
    }
    ms--;
  }
}

// 函数功能：初始化时钟
// 输入参数：无
// 输出参数：无
// 返 回 值：无
// 备    注：无
```

```
void CLK_Init()
{
  CLK_CKDIVR = 0x11;        // 10: fHSI = fHSI RC output/ 4
                    //        = 16MHZ / 4 =4MHZ
                    // 001: fCPU=fMASTER/2. = 2MHZ
}

// 函数功能：初始化定时器 2 的通道 2，用于控制 LED 的亮度
// 输入参数：无
// 输出参数：无
// 返 回 值：无
// 备   注：无
void TIM_Init()
{
  TIM2_CCMR2 = TIM2_CCMR2 | 0x70;// Output mode PWM2.
                    // 通道 2 被设置成比较输出方式
                    // OC2M = 111,为 PWM 模式 2，
                    // 向上计数时，若计数器小于比较值，为无效电平
                    // 即当计数器在 0 到比较值时，输出为 1，否则为 0
  TIM2_CCER1 = TIM2_CCER1 | 0x30;// CC polarity low,enable PWM output    */
                    // CC2P = 1，低电平为有效电平
                    // CC2E = 1，开启输出引脚

//初始化自动装载寄存器，决定 PWM 方波的频率，Fpwm=4000000/256=15625HZ
  TIM2_ARRH = 0;
  TIM2_ARRL = 0xFF;

//初始化比较寄存器，决定 PWM 方波的占空比
  TIM2_CCR2H = 0;
  TIM2_CCR2L = 0;

// 初始化时钟分频器为 1，即计数器的时钟频率为 Fmaster=4MHZ
  TIM2_PSCR = 0;
// 启动计数
  TIM2_CR1 = TIM2_CR1 | 0x01;
}

main()
```

```
{
  unsigned  char  i;

  CLK_Init();                 // 初始化时钟
  TIM_Init();                 // 初始化定时器

  while(1)                    // 进入无限循环
  {
   // 下面的循环将占空比逐渐从 0 递增到 50%
   for(i=0;i<128;i++)
   {
     TIM2_CCR2H  =  0;
     TIM2_CCR2L  =  i;
     DelayMS(5);
   }

   // 下面的循环将占空比逐渐从 50%递减到 0
   for(i=128;i>0;i--)
   {
     TIM2_CCR2H  =  0;
     TIM2_CCR2L  =  i;
     DelayMS(5);
   }
 }
}
```