```
#define     LIN_ID          0x31

u8 TX1_Cnt;                                         //发送计数
u8 RX1_Cnt;                                         //接收计数
u8 TX2_Cnt;                                         //发送计数
u8 RX2_Cnt;                                         //接收计数
bit B_TX1_Busy;                                     //发送忙标志
bit B_TX2_Busy;                                     //发送忙标志
u8 RX1_TimeOut;
u8 RX2_TimeOut;

u8 xdata RX1_Buffer[UART1_BUF_LENGTH];              //接收缓冲
u8 xdata RX2_Buffer[UART2_BUF_LENGTH];              //接收缓冲

void UART1_config(u8 brt);
void UART2_config(u8 brt);
void PrintString1(u8 *puts);
void delay_ms(u8 ms);
void UART1_TxByte(u8 dat);
void UART2_TxByte(u8 dat);
void Lin_Send(u8 *puts);
void SetTimer2Baudraye(u16 dat);

//========================================================================
// 函数: void main(void)
// 描述: 主函数。
// 参数: none.
// 返回: none.
// 版本: VER1.0
// 日期: 2014-11-28
// 备注:
//========================================================================
void main(void)
{
    u8 i;

    P0M1 = 0; P0M0 = 0;                             //设置为准双向口
    P1M1 = 0; P1M0 = 0;                             //设置为准双向口
    P2M1 = 0; P2M0 = 0;                             //设置为准双向口
    P3M1 = 0; P3M0 = 0;                             //设置为准双向口
    P4M1 = 0; P4M0 = 0;                             //设置为准双向口
    P5M1 = 0; P5M0 = 0;                             //设置为准双向口
    P6M1 = 0; P6M0 = 0;                             //设置为准双向口
    P7M1 = 0; P7M0 = 0;                             //设置为准双向口

    UART1_config(1);
    UART2_config(2);
    EA = 1;                                         //允许全局中断
    SLP_N =  1;

    PrintString1("STC8H8K64U UART1 Test Programme!\r\n"); //UART1 发送一个字符串

    while (1)
    {
        delay_ms(1);
        if(RX1_TimeOut > 0)
        {
            if(--RX1_TimeOut == 0)                  //超时,则串口接收结束
```

```
            {
                if(RX1_Cnt > 0)
                {
                    Lin_Send(RX1_Buffer);                   //将 UART1 收到的数据发送到 LIN 总线上
                }
                RX1_Cnt = 0;
            }
        }

        if(RX2_TimeOut > 0)
        {
            if(--RX2_TimeOut == 0)                          //超时,则串口接收结束
            {
                if(RX2_Cnt > 0)
                {
                    for (i=0; I < RX2_Cnt; i++)             //遇到停止符 0 结束
                    {
                        UART1_TxByte(RX2_Buffer[i]); //从 LIN 总线收到的数据发送到 UART1
                    }
                }
                RX2_Cnt = 0;
            }
        }
    }
}

//=========================================================================
// 函数: void delay_ms(unsigned  char ms)
// 描述:  延时函数。
// 参数: ms,要延时的 ms 数,  这里只支持 1~255ms.  自动适应主时钟.
// 返回: none.
// 版本: VER1.0
// 日期: 2013-4-1
// 备注:
//=========================================================================
void delay_ms(u8 ms)
{
    u16 i;
    do{
        i = MAIN_Fosc / 10000;
        while(--i);                                         //10T per loop
    }while(--ms);
}

//=========================================================================
// 函数: u8                Lin_CheckPID(u8     id)
// 描述: ID 码加上校验符,  转成 PID 码。
// 参数: ID 码.
// 返回: PID 码.
// 版本: VER1.0
// 日期: 2020-12-2
// 备注:
//=========================================================================
u8 Lin_CheckPID(u8 id)
{
    u8 returnpid ;
    u8 P0 ;
    u8 P1 ;
```

```
    P0 = (((id)^(id>>1)^(id>>2)^(id>>4))&0x01)<<6;
    P1 = ((~((id>>1)^(id>>3)^(id>>4)^(id>>5)))&0x01)<<7;

    returnpid = id|P0|P1 ;

    return returnpid ;
}

//==========================================================================
// 函数: u8 LINCalcChecksum(u8 *dat)
// 描述: 计算校验码。
// 参数: 数据场传输的数据.
// 返回: 校验码.
// 版本: VER1.0
// 日期: 2020-12-2
// 备注:
//==========================================================================
static u8 LINCalcChecksum(u8 *dat)
{
    u16 sum = 0;
    u8 i;

    for(I = 0; i < 8; i++)
    {
    sum += dat[i];
    if(sum & 0xFF00)
    {
        sum = (sum & 0x00FF) + 1;
    }
    }
    sum ^= 0x00FF;
    return (u8)sum;
}

//==========================================================================
// 函数: void Lin_SendBreak(void)
// 描述: 发送显性间隔信号。
// 参数: none.
// 返回: none.
// 版本: VER1.0
// 日期: 2020-12-2
// 备注:
//==========================================================================
void Lin_SendBreak(void)
{
    SetTimer2Baudraye(Baudrate_Break);
    UART2_TxByte(0);
    SetTimer2Baudraye(Baudrate2);
}

//==========================================================================
// 函数: void Lin_Send(u8 *puts)
// 描述: 发送 LIN 总线报文。
// 参数: 待发送的数据场内容.
// 返回: none.
// 版本: VER1.0
// 日期: 2020-12-2
// 备注:
//==========================================================================
```

```
void Lin_Send(u8 *puts)
{
    u8 i;

    Lin_SendBreak();                                    //Break
    UART2_TxByte(0x55);                                 //SYNC
    UART2_TxByte(Lin_CheckPID(LIN_ID));                 //LIN ID
    for(i=0;i<8;i++)
    {
        UART2_TxByte(puts[i]);
    }
    UART2_TxByte(LINCalcChecksum(puts));
}

//========================================================================
// 函数: void UART1_TxByte(u8 dat)
// 描述: 发送一个字节.
// 参数: 无.
// 返回: 无.
// 版本: V1.0, 2014-6-30
//========================================================================
void UART1_TxByte(u8 dat)
{
    SBUF = dat;
    B_TX1_Busy = 1;
    while(B_TX1_Busy);
}

//========================================================================
// 函数: void UART2_TxByte(u8 dat)
// 描述: 发送一个字节.
// 参数: 无.
// 返回: 无.
// 版本: V1.0, 2014-6-30
//========================================================================
void UART2_TxByte(u8 dat)
{
    S2BUF = dat;
    B_TX2_Busy = 1;
    while(B_TX2_Busy);
}

//========================================================================
// 函数: void PrintString1(u8 *puts)
// 描述: 串口 1 发送字符串函数。
// 参数: puts:                  字符串指针.
// 返回: none.
// 版本: VER1.0
// 日期: 2014-11-28
// 备注:
//========================================================================
void PrintString1(u8 *puts)
{
    for (; *puts != 0; puts++)                          //遇到停止符 0 结束
    {
        SBUF = *puts;
        B_TX1_Busy = 1;
        while(B_TX1_Busy);
    }
```

```
}

//========================================================================
// 函数: void PrintString2(u8 *puts)
// 描述: 串口 2 发送字符串函数。
// 参数: puts:                  字符串指针.
// 返回: none.
// 版本: VER1.0
// 日期: 2014-11-28
// 备注:
//========================================================================
//void PrintString2(u8 *puts)
//{
//      for (; *puts != 0;    puts++)                   //遇到停止符 0 结束
//      {
//          S2BUF = *puts;
//          B_TX2_Busy = 1;
//          while(B_TX2_Busy);
//      }
//}

//========================================================================
// 函数: SetTimer2Baudraye(u16 dat)
// 描述: 设置 Timer2 做波特率发生器。
// 参数: dat: Timer2 的重装值.
// 返回: none.
// 版本: VER1.0
// 日期: 2014-11-28
// 备注:
//========================================================================
void SetTimer2Baudraye(u16 dat)
{
    AUXR &= ~(1<<4);                                    //Timer stop
    AUXR &= ~(1<<3);                                    //Timer2 set As Timer
    AUXR |= (1<<2);                                     //Timer2 set as 1T mode
    TH2   = dat / 256;
    TL2 = dat % 256;
    IE2 &= ~(1<<2);                                     //禁止中断
    AUXR |= (1<<4);                                     //Timer run enable
}

//========================================================================
// 函数: void UART1_config(u8 brt)
// 描述: UART1 初始化函数。
// 参数: brt: 选择波特率, 2: 使用 Timer2 做波特率, 其它值: 使用 Timer1 做波特率.
// 返回: none.
// 版本: VER1.0
// 日期: 2014-11-28
// 备注:
//========================================================================
void UART1_config(u8 brt)
{
    /********** 波特率使用定时器 2 ****************/
    if(brt == 2)
    {
        AUXR |= 0x01;                                   //S1 BRT Use Timer2;
        SetTimer2Baudraye(Baudrate1);
    }
```

```
    /********** 波特率使用定时器 1 *****************/
    else
    {
        TR1  = 0;
        AUXR &= ~0x01;                  //S1 BRT Use Timer1;
        AUXR |= (1<<6);                 //Timer1 set as   1T mode
        TMOD &= ~(1<<6);                //Timer1 set As   Timer
        TMOD &= ~0x30;                  //Timer1_16bitAutoReload;
        TH1 = (u8)(Baudrate1 / 256);
        TL1 = (u8)(Baudrate1 % 256);
        ET1 = 0;                        //禁止中断
        INT_CLKO &= ~0x02;              //不输出时钟
        TR1 = 1;
    }
    /*************************************************/

    SCON = (SCON & 0x3f) | 0x40;        //UART1 模式:  0x00:  同步移位输出,
                                        //             0x40: 8 位数据,可变波特率,
                                        //             0x80: 9 位数据,固定波特率,
                                        //             0xc0: 9 位数据,可变波特率
//  PS  = 1;                            //高优先级中断
    ES  = 1;                            //允许中断
    REN = 1;                            //允许接收
    P_SW1 &= 0x3f;
//  P_SW1 |= 0x80;                      //UART1switch to:    0x00: P3.0 P3.1,
                                        //                   0x40: P3.6 P3.7,
                                        //                   0x80: P1.6 P1.7,
                                        //                   0xC0: P4.3 P4.4

    B_TX1_Busy = 0;
    TX1_Cnt = 0;
    RX1_Cnt = 0;
}

//========================================================================
// 函数: void UART2_config(u8 brt)
// 描述: UART2 初始化函数。
// 参数: brt: 选择波特率, 2: 使用 Timer2 做波特率, 其它值: 无效.
// 返回: none.
// 版本: VER1.0
// 日期: 2014-11-28
// 备注:
//========================================================================
void UART2_config(u8 brt)
{
    if(brt == 2)
    {
        SetTimer2Baudraye(Baudrate2);

        S2CON &= ~(1<<7);               //8 位数据,1 位起始位,1 位停止位,  无校验
        IE2 |= 1;                       //允许中断
        S2CON |= (1<<4);                //允许接收
        P_SW2 &= ~0x01;
//      P_SW2     |= 1;                 //UART2 switch to: 0: P1.0/P1.1, 1: P4.6/P4.7

        B_TX2_Busy = 0;
        TX2_Cnt = 0;
        RX2_Cnt = 0;
    }
```

```
}

//========================================================================
// 函数: void UART1_int     (void) interrupt UART1_VECTOR
// 描述: UART1 中断函数。
// 参数: nine.
// 返回: none.
// 版本: VER1.0
// 日期: 2014-11-28
// 备注:
//========================================================================
void UART1_int (void) interrupt   4
{
    if(RI)
    {
        RI = 0;
        if(RX1_Cnt >= UART1_BUF_LENGTH) RX1_Cnt = 0;
        RX1_Buffer[RX1_Cnt] = SBUF;
        RX1_Cnt++;
        RX1_TimeOut = 5;
    }

    if(TI)
    {
        TI = 0;
        B_TX1_Busy = 0;
    }
}

//========================================================================
// 函数: void UART2_int     (void) interrupt UART2_VECTOR
// 描述: UART2 中断函数。
// 参数: nine.
// 返回: none.
// 版本: VER1.0
// 日期: 2014-11-28
// 备注:
//========================================================================
void UART2_int (void) interrupt   8
{
    if((S2CON & 1) != 0)
    {
        S2CON &= ~1;                                    //Clear Rx flag
        if(RX2_Cnt >= UART2_BUF_LENGTH) RX2_Cnt = 0;
        RX2_Buffer[RX2_Cnt] = S2BUF;
        RX2_Cnt++;
        RX2_TimeOut = 5;
    }

    if((S2CON &     2) != 0)
    {
        S2CON &= ~2;                                    //Clear Tx flag
        B_TX2_Busy = 0;
    }
}
```

15 比较器，掉电检测，内部 1.19V 参考信号源

STC8A8K64D4 系列单片机内部集成了一个比较器。比较器的正极可以是 P3.7 端口、P5.0 端口、P5.1 端口或者 ADC 的模拟输入通道，负极可以是 P3.6 端口或者是内部 BandGap 经过 OP 后的 REFV 电压（内部固定比较电压）。**通过多路选择器和分时复用可实现多个比较器的应用**

比较器内部有可程序控制的两级滤波：模拟滤波和数字滤波。模拟滤波可以过滤掉比较输入信号中的毛刺信号，数字滤波可以等待输入信号更加稳定后再进行比较。比较结果可直接通过读取内部寄存器位获得，也可将比较器结果正向或反向输出到外部端口。将比较结果输出到外部端口可用作外部事件的触发信号和反馈信号，可扩大比较的应用范围。

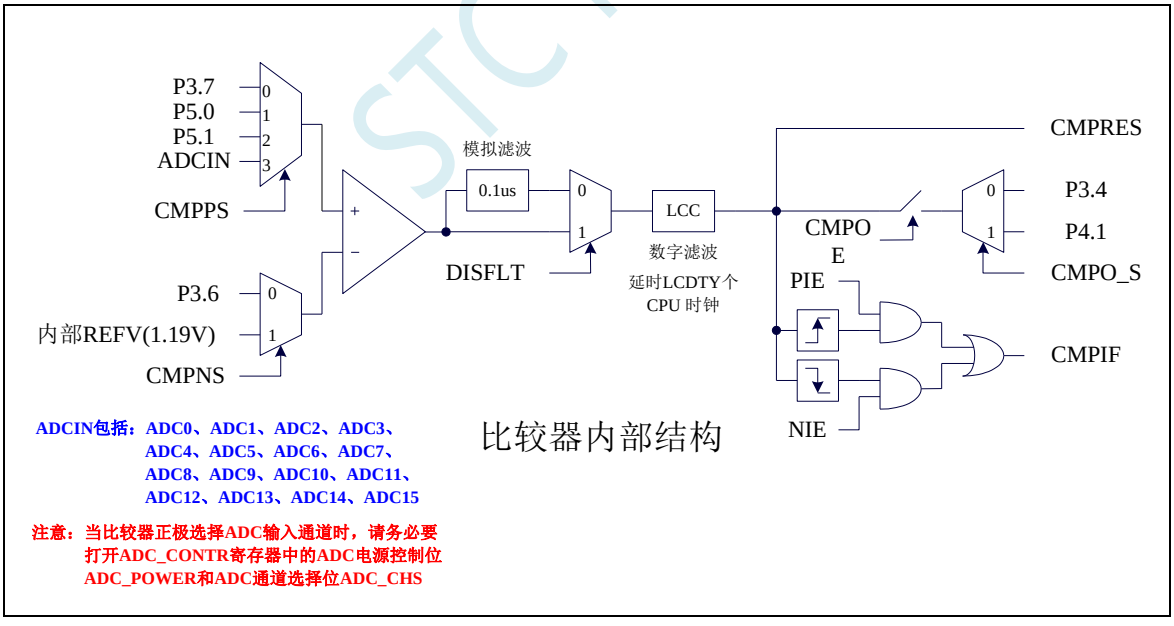
15.1 比较器输出功能脚切换

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
P_SW2	BAH	EAXFR	-	I2C_S[1:0]		CMPO_S	S4_S	S3_S	S2_S

CMPO_S：比较器输出脚选择位

CMPO_S	CMPO
0	P3.4
1	P4.1

15.2 比较器内部结构图



15.3 比较器相关的寄存器

符号	描述	地址	位地址与符号								复位值
			B7	B6	B5	B4	B3	B2	B1	B0	
CMPCR1	比较器控制寄存器 1	E6H	CMPEN	CMPIF	PIE	NIE	-	-	CMPOE	CMPRES	0000,xx00
CMPCR2	比较器控制寄存器 2	E7H	INVCMPO	DISFLT	LCDTY[5:0]						0000,0000
CMPEXCFG	比较器扩展配置寄存器	FEAEH	CHYS[1:0]		-	-	-	CMPNS	CMPPS[1:0]		00xx,x000

15.3.1 比较器控制寄存器 1（CMPCR1）

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
CMPCR1	E6H	CMPEN	CMPIF	PIE	NIE	-	-	CMPOE	CMPRES

CMPEN：比较器模块使能位

0：关闭比较功能

1：使能比较功能

CMPIF：比较器中断标志位。当 PIE 或 NIE 被使能后，若产生相应的中断信号，硬件自动将 CMPIF 置 1，并向 CPU 提出中断请求。此标志位必须用户软件清零。

（注意：没有使能比较器中断时，硬件不会设置此中断标志，即使用查询方式访问比较器时，不能查询此中断标志）

PIE：比较器上升沿中断使能位。

0：禁止比较器上升沿中断。

1：使能比较器上升沿中断。使能比较器的比较结果由 0 变成 1 时产生中断请求。

NIE：比较器下降沿中断使能位。

0：禁止比较器下降沿中断。

1：使能比较器下降沿中断。使能比较器的比较结果由 1 变成 0 时产生中断请求。

CMPOE：比较器结果输出控制位

0：禁止比较器结果输出

1：使能比较器结果输出。比较器结果输出到 P3.4 或者 P4.1（由 P_SW2 中的 CMPO_S 进行设定）

CMPRES：比较器的比较结果。此位为只读。

0：表示 CMP+的电平低于 CMP-的电平

1：表示 CMP+的电平高于 CMP-的电平

CMPRES 是经过数字滤波后的输出信号，而不是比较器的直接输出结果。

15.3.2 比较器控制寄存器 2（CMPCR2）

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
CMPCR2	E7H	INVCMPO	DISFLT	LCDTY[5:0]					

INVCMPO：比较器结果输出控制

0：比较器结果正向输出。若 CMPRES 为 0，则 P3.4/P4.1 输出低电平，反之输出高电平。

1：比较器结果反向输出。若 CMPRES 为 0，则 P3.4/P4.1 输出高电平，反之输出低电平。

DISFLT：模拟滤波功能控制

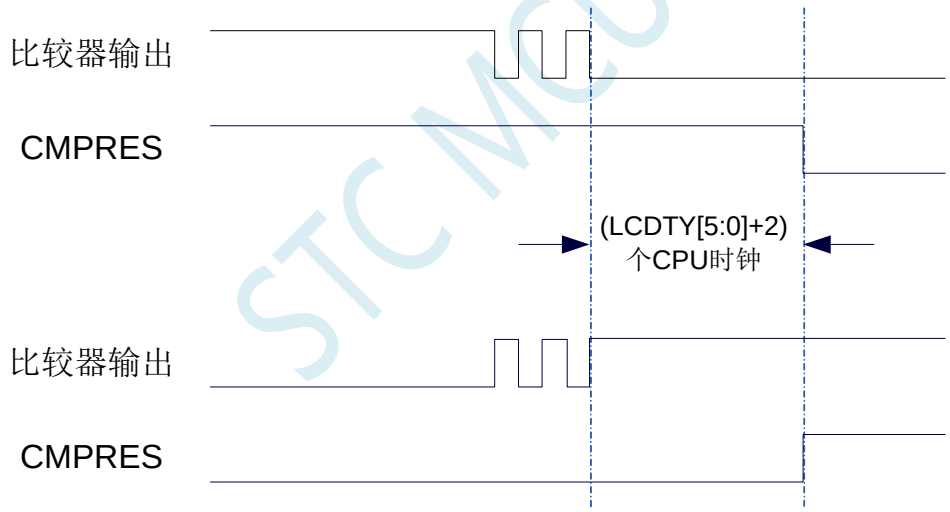
0：使能 0.1us 模拟滤波功能

1：关闭 0.1us 模拟滤波功能，可略微提高比较器的比较速度。

LCDTY[5:0]：数字滤波功能控制

数字滤波功能即为数字信号去抖动功能。当比较结果发生上升沿或者下降沿变化时，比较器侦测变化后的信号必须维持 LCDTY 所设置的 CPU 时钟数不发生变化，才认为数据变化是有效的；否则将视同信号无变化。

注意：当使能数字滤波功能后，芯片内部实际的等待时钟需额外增加两个状态机切换时间，即若 LCDTY 设置为 0 时，为关闭数字滤波功能；若 LCDTY 设置为非 0 值 n（n=1~63）时，则实际的数字滤波时间为（n+2）个系统时钟



15.3.3 比较器扩展配置寄存器（CMPEXCFG）

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
CMPEXCFG	FEAEH	CHYS[1:0]		-	-	-	CMPNS	CMPPS[1:0]	

CHYS[1:0]：比较器 DC 迟滞输入选择

CHYS [1:0]	比较器 DC 迟滞输入选择
00	0mV
01	10mV
10	20mV
11	30mV

CMPNS：比较器负端输入选择

0：P3.6

1：选择内部 BandGap 经过 OP 后的电压 REFV 作为比较器负极输入源 （芯片在出厂时，内部参考电压调整为 **1.19V**）

CMPPS[1:0]：比较器正端输入选择

CMPPS[1:0]	比较器正端
00	P3.7
01	P5.0
10	P5.1
11	ADCIN

15.4 范例程序

15.4.1 比较器的使用（中断方式）

C 语言代码

```
//测试工作频率为 11.0592MHz

#include "reg51.h"
#include "intrins.h"

#define   CMPEXCFG      (*(unsigned char volatile xdata *)0xfeae)
sfr       P_SW2     =   0xba;

sfr       CMPCR1    =   0xe6;
sfr       CMPCR2    =   0xe7;

sfr       P0M1      =   0x93;
sfr       P0M0      =   0x94;
sfr       P1M1      =   0x91;
sfr       P1M0      =   0x92;
sfr       P2M1      =   0x95;
sfr       P2M0      =   0x96;
sfr       P3M1      =   0xb1;
sfr       P3M0      =   0xb2;
sfr       P4M1      =   0xb3;
sfr       P4M0      =   0xb4;
sfr       P5M1      =   0xc9;
sfr       P5M0      =   0xca;

sbit      P10       =   P1^0;
sbit      P11       =   P1^1;

void CMP_Isr() interrupt 21
{
    CMPCR1 &= ~0x40;                        //清中断标志
    if (CMPCR1 & 0x01)
    {
        P10 = !P10;                         //上升沿中断测试端口
    }
    else
    {
        P11 = !P11;                         //下降沿中断测试端口
    }
}

void main()
{
    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
```

```
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    P_SW2 = 0x80;
    CMPEXCFG = 0x00;
    CMPEXCFG &= ~0x03;                  //P3.7 为 CMP+输入脚
//  CMPEXCFG |= 0x01;                   //P5.0 为 CMP+输入脚
//  CMPEXCFG |= 0x02;                   //P5.1 为 CMP+输入脚
//  CMPEXCFG |= 0x03;                   //ADC 输入脚为 CMP+输入脚
    CMPEXCFG &= ~0x04;                  //P3.6 为 CMP-输入脚
//  CMPEXCFG |= 0x04;                   //内部 1.19V 参考信号源为 CMP-输入脚
    P_SW2 = 0x00;

    CMPCR2 = 0x00;
    CMPCR2 &= ~0x80;                    //比较器正向输出
//  CMPCR2 |= 0x80;                     //比较器反向输出
    CMPCR2 &= ~0x40;                    //使能 0.1us 滤波
//  CMPCR2 |= 0x40;                     //禁止 0.1us 滤波
//  CMPCR2 &= ~0x3f;                    //比较器结果直接输出
    CMPCR2 |= 0x10;                     //比较器结果经过 16 个去抖时钟后输出
    CMPCR1 = 0x00;
    CMPCR1 |= 0x30;                     //使能比较器边沿中断
//  CMPCR1 &= ~0x20;                    //禁止比较器上升沿中断
//  CMPCR1 |= 0x20;                     //使能比较器上升沿中断
//  CMPCR1 &= ~0x10;                    //禁止比较器下降沿中断
//  CMPCR1 |= 0x10;                     //使能比较器下降沿中断
//  CMPCR1 &= ~0x02;                    //禁止比较器输出
    CMPCR1 |= 0x02;                     //使能比较器输出
    CMPCR1 |= 0x80;                     //使能比较器模块

    EA = 1;

    while (1);
}
```

汇编代码

```
;测试工作频率为 11.0592MHz

CMPEXCFG    EQU     0FEFAH

P_SW2       DATA    0BAH
CMPCR1      DATA    0E6H
CMPCR2      DATA    0E7H

P0M1        DATA    093H
P0M0        DATA    094H
P1M1        DATA    091H
P1M0        DATA    092H
P2M1        DATA    095H
P2M0        DATA    096H
P3M1        DATA    0B1H
P3M0        DATA    0B2H
P4M1        DATA    0B3H
P4M0        DATA    0B4H
P5M1        DATA    0C9H
P5M0        DATA    0CAH
```

```
                ORG     0000H
                LJMP    MAIN
                ORG     00ABH
                LJMP    CMPISR

                ORG     0100H
CMPISR:
                PUSH    ACC
                ANL     CMPCR1,#NOT 40H         ;清中断标志
                MOV     A,CMPCR1
                JB      ACC.0,RSING
FALLING:
                CPL     P1.0                    ;下降沿中断测试端口
                POP     ACC
                RETI
RSING:
                CPL     P1.1                    ;上升沿中断测试端口
                POP     ACC
                RETI

MAIN:
                MOV     SP, #5FH
                MOV     P0M0, #00H
                MOV     P0M1, #00H
                MOV     P1M0, #00H
                MOV     P1M1, #00H
                MOV     P2M0, #00H
                MOV     P2M1, #00H
                MOV     P3M0, #00H
                MOV     P3M1, #00H
                MOV     P4M0, #00H
                MOV     P4M1, #00H
                MOV     P5M0, #00H
                MOV     P5M1, #00H

                MOV     P_SW,#80H
                MOV     DPTR,#CMPEXCFG
                CLR     A
                ANL     A,#NOT 03H              ;P3.7 为CMP+输入脚
;               ORL     A,#01H                  ;P5.0 为CMP+输入脚
;               ORL     A,#02H                  ;P5.1 为CMP+输入脚
;               ORL     A,#03H                  ;ADC 输入脚为CMP+输入脚
                ANL     A,#NOT 04H              ;P3.6 为CMP-输入脚
;               ORL     A,#04H                  ;内部 1.19V 参考信号源为CMP-输入脚
                MOVX    @DPTR,A
                MOV     P_SW,#00H

                MOV     CMPCR2,#00H
                ANL     CMPCR2,#NOT 80H         ;比较器正向输出
;               ORL     CMPCR2,#80H             ;比较器反向输出
                ANL     CMPCR2,#NOT 40H         ;使能 0.1us 滤波
;               ORL     CMPCR2,#40H             ;禁止 0.1us 滤波
;               ANL     CMPCR2,#NOT 3FH         ;比较器结果直接输出
                ORL     CMPCR2,#10H             ;比较器结果经过 16 个去抖时钟后输出
                MOV     CMPCR1,#00H
                ORL     CMPCR1,#30H             ;使能比较器边沿中断
;               ANL     CMPCR1,#NOT 20H         ;禁止比较器上升沿中断
;               ORL     CMPCR1,#20H             ;使能比较器上升沿中断
```

```
;               ANL     CMPCR1,#NOT 10H         ;禁止比较器下降沿中断
;               ORL     CMPCR1,#10H             ;使能比较器下降沿中断
                ANL     CMPCR1,#NOT 08H         ;P3.7 为CMP+输入脚
;               ORL     CMPCR1,#08H             ;ADC 输入脚为CMP+输入脚
;               ANL     CMPCR1,#NOT 04H         ;内部1.19V 参考信号源为CMP-输入脚
                ORL     CMPCR1,#04H             ;P3.6 为CMP-输入脚
;               ANL     CMPCR1,#NOT 02H         ;禁止比较器输出
                ORL     CMPCR1,#02H             ;使能比较器输出
                ORL     CMPCR1,#80H             ;使能比较器模块
                SETB    EA

                JMP     $

                END
```

15.4.2 比较器的使用（查询方式）

C 语言代码

```
//测试工作频率为11.0592MHz

#include "reg51.h"
#include "intrins.h"

#define   CMPEXCFG      (*(unsigned char volatile xdata *)0xfeae)
sfr       P_SW2     =   0xba;

sfr       CMPCR1    =   0xe6;
sfr       CMPCR2    =   0xe7;

sfr       P0M1      =   0x93;
sfr       P0M0      =   0x94;
sfr       P1M1      =   0x91;
sfr       P1M0      =   0x92;
sfr       P2M1      =   0x95;
sfr       P2M0      =   0x96;
sfr       P3M1      =   0xb1;
sfr       P3M0      =   0xb2;
sfr       P4M1      =   0xb3;
sfr       P4M0      =   0xb4;
sfr       P5M1      =   0xc9;
sfr       P5M0      =   0xca;

sbit      P10       =   P1^0;
sbit      P11       =   P1^1;

void main()
{
    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
```

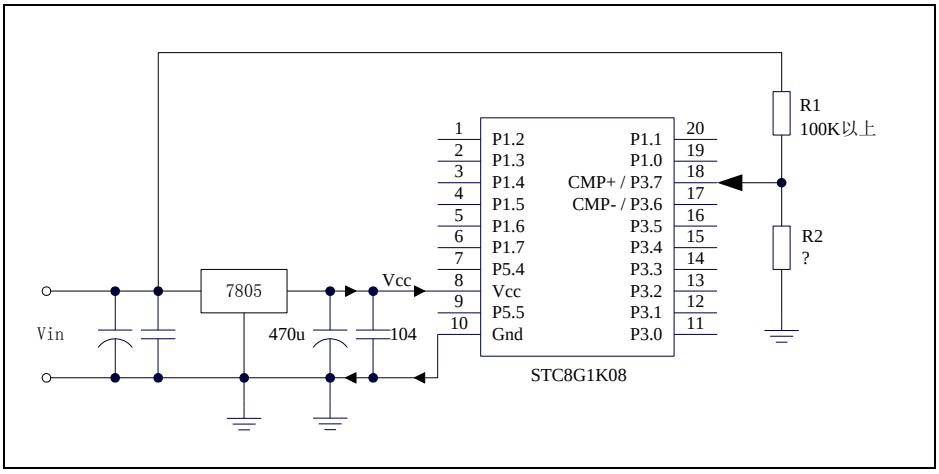
```
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    P_SW2 = 0x80;
    CMPEXCFG = 0x00;
    CMPEXCFG &= ~0x03;                  //P3.7 为 CMP+输入脚
//  CMPEXCFG |= 0x01;                   //P5.0 为 CMP+输入脚
//  CMPEXCFG |= 0x02;                   //P5.1 为 CMP+输入脚
//  CMPEXCFG |= 0x03;                   //ADC 输入脚为 CMP+输入脚
    CMPEXCFG &= ~0x04;                  //P3.6 为 CMP-输入脚
//  CMPEXCFG |= 0x04;                   //内部 1.19V 参考信号源为 CMP-输入脚
    P_SW2 = 0x00;

    CMPCR2 = 0x00;
    CMPCR2 &= ~0x80;                    //比较器正向输出
//  CMPCR2 |= 0x80;                     //比较器反向输出
    CMPCR2 &= ~0x40;                    //使能 0.1us 滤波
//  CMPCR2 |= 0x40;                     //禁止 0.1us 滤波
//  CMPCR2 &= ~0x3f;                    //比较器结果直接输出
    CMPCR2 |= 0x10;                     //比较器结果经过 16 个去抖时钟后输出
    CMPCR1 = 0x00;
    CMPCR1 |= 0x30;                     //使能比较器边沿中断
//  CMPCR1 &= ~0x20;                    //禁止比较器上升沿中断
//  CMPCR1 |= 0x20;                     //使能比较器上升沿中断
//  CMPCR1 &= ~0x10;                    //禁止比较器下降沿中断
//  CMPCR1 |= 0x10;                     //使能比较器下降沿中断
//  CMPCR1 &= ~0x02;                    //禁止比较器输出
    CMPCR1 |= 0x02;                     //使能比较器输出
    CMPCR1 |= 0x80;                     //使能比较器模块

    while (1)
    {
        P10 = CMPCR1 & 0x01;            //读取比较器比较结果
    }
}
```

汇编代码

```
;测试工作频率为 11.0592MHz

CMPEXCFG    EQU     0FEFAH

P_SW2       DATA    0BAH
CMPCR1      DATA    0E6H
CMPCR2      DATA    0E7H

P0M1        DATA    093H
P0M0        DATA    094H
P1M1        DATA    091H
P1M0        DATA    092H
P2M1        DATA    095H
P2M0        DATA    096H
P3M1        DATA    0B1H
P3M0        DATA    0B2H
P4M1        DATA    0B3H
P4M0        DATA    0B4H
P5M1        DATA    0C9H
```

```
P5M0        DATA    0CAH

            ORG     0000H
            LJMP    MAIN

            ORG     0100H
MAIN:
            MOV     SP, #5FH
            MOV     P0M0, #00H
            MOV     P0M1, #00H
            MOV     P1M0, #00H
            MOV     P1M1, #00H
            MOV     P2M0, #00H
            MOV     P2M1, #00H
            MOV     P3M0, #00H
            MOV     P3M1, #00H
            MOV     P4M0, #00H
            MOV     P4M1, #00H
            MOV     P5M0, #00H
            MOV     P5M1, #00H

            MOV     P_SW,#80H
            MOV     DPTR,#CMPEXCFG
            CLR     A
            ANL     A,#NOT 03H          ;P3.7 为 CMP+输入脚
;           ORL     A,#01H              ;P5.0 为 CMP+输入脚
;           ORL     A,#02H              ;P5.1 为 CMP+输入脚
;           ORL     A,#03H              ;ADC 输入脚为 CMP+输入脚
            ANL     A,#NOT 04H          ;P3.6 为 CMP-输入脚
;           ORL     A,#04H              ;内部 1.19V 参考信号源为 CMP-输入脚
            MOVX    @DPTR,A
            MOV     P_SW,#00H

            MOV     CMPCR2,#00H
            ANL     CMPCR2,#NOT 80H     ;比较器正向输出
;           ORL     CMPCR2,#80H         ;比较器反向输出
            ANL     CMPCR2,#NOT 40H     ;使能 0.1us 滤波
;           ORL     CMPCR2,#40H         ;禁止 0.1us 滤波
;           ANL     CMPCR2,#NOT 3FH     ;比较器结果直接输出
            ORL     CMPCR2,#10H         ;比较器结果经过 16 个去抖时钟后输出
            MOV     CMPCR1,#00H
            ORL     CMPCR1,#30H         ;使能比较器边沿中断
;           ANL     CMPCR1,#NOT 20H     ;禁止比较器上升沿中断
;           ORL     CMPCR1,#20H         ;使能比较器上升沿中断
;           ANL     CMPCR1,#NOT 10H     ;禁止比较器下降沿中断
;           ORL     CMPCR1,#10H         ;使能比较器下降沿中断
;           ANL     CMPCR1,#NOT 02H     ;禁止比较器输出
            ORL     CMPCR1,#02H         ;使能比较器输出
            ORL     CMPCR1,#80H         ;使能比较器模块

LOOP:
            MOV     A,CMPCR1
            MOV     C,ACC.0
            MOV     P1.0,C              ;读取比较器比较结果
            JMP     LOOP

            END
```

15.4.3 比较器作外部掉电检测（掉电过程中应及时保存用户数据到 EEPROM 中）

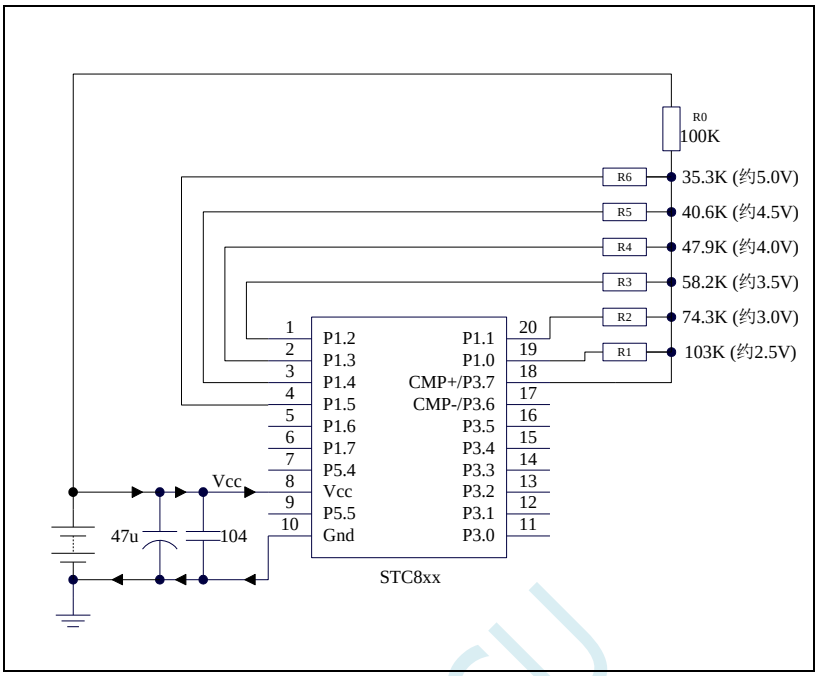


上图中电阻 R1 和 R2 对稳压块 7805 的前端电压进行分压，分压后的电压作为比较器 CMP+的外部输入与内部 1.19V 参考信号源进行比较。

一般当交流电在 220V 时，稳压块 7805 前端的直流电压为 11V，但当交流电压降到 160V 时，稳压块 7805 前端的直流电压为 8.5V。当稳压块 7805 前端的直流电压低于或等于 8.5V 时，该前端输入的直流电压被电阻 R1 和 R2 分压到比较器正极输入端 CMP+，CMP+端输入电压低于内部 1.19V 参考信号源，此时可产生比较器中断，这样在掉电检测时就有充足的时间将数据保存到 EEPROM 中。当稳压块 7805 前端的直流电压高于 8.5V 时，该前端输入的直流电压被电阻 R1 和 R2 分压到比较器正极输入端 CMP+，CMP+端输入电压高于内部 1.19V 参考信号源，此时 CPU 可继续正常工作。

内部 1.19V 参考信号源即为内部 BandGap 经过 OP 后的电压 REFV（**芯片在出厂时，内部参考电压调整为 1.19V**）。具体的数值要通过读取内部 1.19V 参考信号源在内部 RAM 区或者 Flash 程序存储器（ROM）区所占用的地址的值获得。对于 STC8 系列，内部 1.19V 参考信号源值在 RAM 和 Flash 程序存储器（ROM）中的存储地址请参考“存储器中的特殊参数”章节

15.4.4 比较器检测工作电压（电池电压）



上图中，利用电阻分压的原理可以近似的测量出 MCU 的工作电压（选通的通道，MCU 的 I/O 口输出低电平，端口电压值接近 Gnd，未选通的通道，MCU 的 I/O 口输出开漏模式的高，不影响其他通道）。

比较器的负端选择内部 1.19V 参考信号源，正端选择通过电阻分压后输入到 CMP+管脚的电压值。

初始化时 P1.5~P1.0 口均设置为开漏模式，并输出高。首先 P1.0 口输出低电平，此时若 Vcc 电压低于 2.5V 则比较器的比较值为 0，反之若 Vcc 电压高于 2.5V 则比较器的比较值为 1；

若确定 Vcc 高于 2.5V，则将 P1.0 口输出高，P1.1 口输出低电平，此时若 Vcc 电压低于 3.0V 则比较器的比较值为 0，反之若 Vcc 电压高于 3.0V 则比较器的比较值为 1；

若确定 Vcc 高于 3.0V，则将 P1.1 口输出高，P1.2 口输出低电平，此时若 Vcc 电压低于 3.5V 则比较器的比较值为 0，反之若 Vcc 电压高于 3.5V 则比较器的比较值为 1；

若确定 Vcc 高于 3.5V，则将 P1.2 口输出高，P1.3 口输出低电平，此时若 Vcc 电压低于 4.0V 则比较器的比较值为 0，反之若 Vcc 电压高于 4.0V 则比较器的比较值为 1；

若确定 Vcc 高于 4.0V，则将 P1.3 口输出高，P1.4 口输出低电平，此时若 Vcc 电压低于 4.5V 则比较器的比较值为 0，反之若 Vcc 电压高于 4.5V 则比较器的比较值为 1；

若确定 Vcc 高于 4.5V，则将 P1.4 口输出高，P1.5 口输出低电平，此时若 Vcc 电压低于 5.0V 则比较器的比较值为 0，反之若 Vcc 电压高于 5.0V 则比较器的比较值为 1。

C 语言代码

```
//测试工作频率为11.0592MHz

#include "reg51.h"
#include "intrins.h"

#define   CMPEXCFG      (*(unsigned char volatile xdata *)0xfeae)
sfr       P_SW2     =   0xba;
```

```
sfr         CMPCR1      =   0xe6;
sfr         CMPCR2      =   0xe7;

sfr         P0M1        =   0x93;
sfr         P0M0        =   0x94;
sfr         P1M1        =   0x91;
sfr         P1M0        =   0x92;
sfr         P2M1        =   0x95;
sfr         P2M0        =   0x96;
sfr         P3M1        =   0xb1;
sfr         P3M0        =   0xb2;
sfr         P4M1        =   0xb3;
sfr         P4M0        =   0xb4;
sfr         P5M1        =   0xc9;
sfr         P5M0        =   0xca;

sfr         P2M0        =   0x96;
sfr         P2M1        =   0x95;

void delay ()
{
    char i;

    for (i=0; i<20; i++);
}

void main()
{
    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    unsigned char v;

    P1M0 = 0x3f;                                //P1.5~P1.0 初始化为开漏模式
    P1M1 = 0x3f;
    P1 = 0xff;

    P_SW2 = 0x80;
    CMPEXCFG = 0x00;
    CMPEXCFG &= ~0x03;                          //P3.7 为 CMP+输入脚
    CMPEXCFG |= 0x04;                           //内部 1.19V 参考信号源为 CMP-输入脚
    P_SW2 = 0x00;

    CMPCR2 = 0x10;                              //比较器结果经过 16 个去抖时钟后输出
    CMPCR1 = 0x00;
    CMPCR1 &= ~0x02;                            //禁止比较器输出
    CMPCR1 |= 0x80;                             //使能比较器模块

    while (1)
```

```
    {
        v = 0x00;                                   //电压<2.5V
        P1 = 0xfe;                                  //P1.0 输出 0
        delay();
        if (!(CMPCR1 & 0x01)) goto ShowVol;
        v = 0x01;                                   //电压>2.5V
        P1 = 0xfd;                                  //P1.1 输出 0
        delay();
        if (!(CMPCR1 & 0x01)) goto ShowVol;
        v = 0x03;                                   //电压>3.0V
        P1 = 0xfb;                                  //P1.2 输出 0
        delay();
        if (!(CMPCR1 & 0x01)) goto ShowVol;
        v = 0x07;                                   //电压>3.5V
        P1 = 0xf7;                                  //P1.3 输出 0
        delay();
        if (!(CMPCR1 & 0x01)) goto ShowVol;
        v = 0x0f;                                   //电压>4.0V
        P1 = 0xef;                                  //P1.4 输出 0
        delay();
        if (!(CMPCR1 & 0x01)) goto ShowVol;
        v = 0x1f;                                   //电压>4.5V
        P1 = 0xdf;                                  //P1.5 输出 0
        delay();
        if (!(CMPCR1 & 0x01)) goto ShowVol;
        v = 0x3f;                                   //电压>5.0V
ShowVol:
        P1 = 0xff;
        P0 = ~v;
    }
}
```

汇编代码

```
;测试工作频率为 11.0592MHz

CMPEXCFG    EQU     0FEFAH

P_SW2       DATA    0BAH
CMPCR1      DATA    0E6H
CMPCR2      DATA    0E7H

P2M0        DATA    096H
P2M1        DATA    095H
P0M1        DATA    093H
P0M0        DATA    094H
P1M1        DATA    091H
P1M0        DATA    092H
P2M1        DATA    095H
P2M0        DATA    096H
P3M1        DATA    0B1H
P3M0        DATA    0B2H
P4M1        DATA    0B3H
P4M0        DATA    0B4H
P5M1        DATA    0C9H
P5M0        DATA    0CAH

            ORG     0000H
            LJMP    MAIN
```

```
        ORG     0100H
MAIN:
        MOV     SP, #5FH
        MOV     P0M0, #00H
        MOV     P0M1, #00H
        MOV     P1M0, #00H
        MOV     P1M1, #00H
        MOV     P2M0, #00H
        MOV     P2M1, #00H
        MOV     P3M0, #00H
        MOV     P3M1, #00H
        MOV     P4M0, #00H
        MOV     P4M1, #00H
        MOV     P5M0, #00H
        MOV     P5M1, #00H

        MOV     P_SW,#80H
        MOV     DPTR,#CMPEXCFG
        CLR     A
        ANL     A,#NOT 03H          ;P3.7 为CMP+输入脚
        ORL     A,#04H              ;内部 1.19V 参考信号源为CMP-输入脚
        MOVX    @DPTR,A
        MOV     P_SW,#00H

        MOV     P1M0,#00111111B     ;P1.5~P1.0 初始化为开漏模式
        MOV     P1M1,#00111111B
        MOV     P1,#0FFH
        MOV     CMPCR2,#10H         ;比较器结果经过 16 个去抖时钟后输出
        MOV     CMPCR1,#00H
        ANL     CMPCR1,#NOT 02H     ;禁止比较器输出
        ORL     CMPCR1,#80H         ;使能比较器模块
LOOP:
        MOV     R0,#00000000B       ;电压<2.5V
        MOV     P1,#11111110B       ;P1.0 输出 0
        CALL    DELAY
        MOV     A,CMPCR1
        JNB     ACC.0,SKIP
        MOV     R0,#00000001B       ;电压>2.5V
        MOV     P1,#11111101B       ;P1.1 输出 0
        CALL    DELAY
        MOV     A,CMPCR1
        JNB     ACC.0,SKIP
        MOV     R0,#00000011B       ;电压>3.0V
        MOV     P1,#11111011B       ;P1.2 输出 0
        CALL    DELAY
        MOV     A,CMPCR1
        JNB     ACC.0,SKIP
        MOV     R0,#00000111B       ;电压>3.5V
        MOV     P1,#11110111B       ;P1.3 输出 0
        CALL    DELAY
        MOV     A,CMPCR1
        JNB     ACC.0,SKIP
        MOV     R0,#00001111B       ;电压>4.0V
        MOV     P1,#11101111B       ;P1.4 输出 0
        CALL    DELAY
        MOV     A,CMPCR1
        JNB     ACC.0,SKIP
        MOV     R0,#00011111B       ;电压>4.5V
```

```
        MOV     P1,#11011111B       ;P1.5 輸出 0
        CALL    DELAY
        MOV     A,CMPCR1
        JNB     ACC.0,SKIP
        MOV     R0,#00111111B       ;电压>5.0V
SKIP:
        MOV     P1,#11111111B
        MOV     A,R0
        CPL     A
        MOV     P0,A                ;P0.5~P0.0 口显示电压
        JMP     LOOP

DELAY:
        MOV     R0,#20
        DJNZ    R0,$
        RET

        END
```

16 IAP/EEPROM/DATA-FLASH

STC8A8K64D4 系列单片机内部集成了大容量的 EEPROM。利用 ISP/IAP 技术可将内部 Data Flash 当 EEPROM，擦写次数在 10 万次以上。EEPROM 可分为若干个扇区，每个扇区包含 512 字节。

注意：EEPROM 的写操作只能将字节中的 1 写为 0，当需要将字节中的 0 写为 1，则必须执行扇区擦除操作。EEPROM 的读/写操作是以 1 字节为单位进行，而 EEPROM 擦除操作是以 1 扇区（512 字节）为单位进行，在执行擦除操作时，如果目标扇区中有需要保留的数据，则必须预先将这些数据读取到 RAM 中暂存，待擦除完成后再将保存的数据和需要更新的数据一起再写回 EEPROM/DATA-FLASH。

所以在使用时，建议同一次修改的数据放在同一个扇区，不是同一次修改的数据放在不同的扇区，不一定要用满。数据存储器的擦除操作是按扇区进行的（每扇区 512 字节）。

EEPROM 可用于保存一些需要在应用过程中修改并且掉电不丢失的参数数据。在用户程序中，可以对 EEPROM 进行字节读/字节编程/扇区擦除操作。在工作电压偏低时，建议不要进行 EEPROM 操作，以免发送数据丢失的情况。

16.1 EEPROM 操作时间

- 读取 1 字节：4 个系统时钟（使用 MOVC 指令读取更方便快捷）
- 编程 1 字节：约 30～40us（实际的编程时间为 6~7.5us，但还需要加上状态转换时间和各种控制信号的 SETUP 和 HOLD 时间）
- 擦除 1 扇区（512 字节）：约 4～6ms

EEPROM 操作所需时间是硬件自动控制的，用户只需要正确设置 IAP_TPS 寄存器即可。

IAP_TPS＝系统工作频率/1000000（小数部分四舍五入进行取整）

例如：系统工作频率为 12MHz，则 IAP_TPS 设置为 12

又例如：系统工作频率为 22.1184MHz，则 IAP_TPS 设置为 22

再例如：系统工作频率为 5.5296MHz，则 IAP_TPS 设置为 6

16.2 EEPROM 相关的寄存器

符号	描述	地址	位地址与符号								复位值
			B7	B6	B5	B4	B3	B2	B1	B0	
IAP_DATA	IAP 数据寄存器	C2H									1111,1111
IAP_ADDRH	IAP 高地址寄存器	C3H									0000,0000
IAP_ADDRL	IAP 低地址寄存器	C4H									0000,0000
IAP_CMD	IAP 命令寄存器	C5H	-	-	-	-	-	-	CMD[1:0]		xxxx,xx00
IAP_TRIG	IAP 触发寄存器	C6H									0000,0000
IAP_CONTR	IAP 控制寄存器	C7H	IAPEN	SWBS	SWRST	CMD_FAIL	-	-	-	-	0000,xxxx
IAP_TPS	IAP 等待时间控制寄存器	F5H	-	-	IAPTPS[5:0]						xx00,0000

16.2.1 EEPROM 数据寄存器（IAP_DATA）

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
IAP_DATA	C2H								

在进行 EEPROM 的读操作时，命令执行完成后读出的 EEPROM 数据保存在 IAP_DATA 寄存器中。

在进行 EEPROM 的写操作时，在执行写命令前，必须将待写入的数据存放在 IAP_DATA 寄存器中，再发送写命令。擦除 EEPROM 命令与 IAP_DATA 寄存器无关。

16.2.2 EEPROM 地址寄存器（IAP_ADDR）

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
IAP_ADDRH	C3H								
IAP_ADDRL	C4H								

EEPROM 进行读、写、擦除操作的目标地址寄存器。IAP_ADDRH 保存地址的高字节，IAP_ADDRL 保存地址的低字节

16.2.3 EEPROM 命令寄存器（IAP_CMD）

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
IAP_CMD	C5H	-	-	-	-	-	-	CMD[1:0]	

CMD[1:0]：发送EEPROM操作命令

00：空操作

01：读 EEPROM 命令。读取目标地址所在的 1 字节。

10：写 EEPROM 命令。写目标地址所在的 1 字节。注意：写操作只能将目标字节中的 1 写为 0，而不能将 0 写为 1。一般当目标字节不为 FFH 时，必须先擦除。

11：擦除 EEPROM。擦除目标地址所在的 1 页（1 扇区/512 字节）。注意：擦除操作会一次擦除 1 个扇区（512 字节），整个扇区的内容全部变成 FFH。

16.2.4 EEPROM 触发寄存器（IAP_TRIG）

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
IAP_TRIG	C6H								

设置完成 EEPROM 读、写、擦除的命令寄存器、地址寄存器、数据寄存器以及控制寄存器后，需要向触发寄存器 IAP_TRIG 依次写入 5AH、A5H（顺序不能交换）两个触发命令来触发相应的读、写、擦除操作。操作完成后，EEPROM 地址寄存器 IAP_ADDRH、IAP_ADDRL 和 EEPROM 命令寄存器 IAP_CMD 的内容不变。如果接下来要对下一个地址的数据进行操作，需手动更新地址寄存器 IAP_ADDRH 和寄存器 IAP_ADDRL 的值。

注意：每次 EEPROM 操作时，都要对 IAP_TRIG 先写入 5AH，再写入 A5H，相应的命令才会生效。写完触发命令后，CPU 会处于 IDLE 等待状态，直到相应的 IAP 操作执行完成后 CPU 才会从 IDLE 状态返回正常状态继续执行 CPU 指令。

16.2.5 EEPROM 控制寄存器（IAP_CONTR）

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
IAP_CONTR	C7H	IAPEN	SWBS	SWRST	CMD_FAIL	-	-	-	-

IAPEN：EEPROM操作使能控制位

0：禁止 EEPROM 操作

1：使能 EEPROM 操作

SWBS：软件复位选择控制位，（需要与SWRST配合使用）

0：软件复位后从用户代码开始执行程序

1：软件复位后从系统 ISP 监控代码区开始执行程序

SWRST：软件复位控制位

0：无动作

1：产生软件复位

CMD_FAIL：EEPROM操作失败状态位，需要软件清零

0：EEPROM 操作正确

1：EEPROM 操作失败

16.2.6 EEPROM 等待时间控制寄存器（IAP_TPS）

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
IAP_TPS	F5H	-	-	IAPTPS[5:0]					

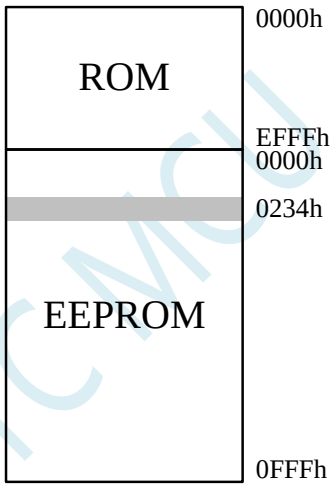
需要根据工作频率进行设置

若工作频率为12MHz，则需要将IAP_TPS设置为12；若工作频率为24MHz，则需要将IAP_TPS设置为24，其他频率以此类推。

16.3 EEPROM 大小及地址

STC8A8K64D4 系列单片机内部均有用于保存用户数据的 EEPROM。内部的 EEPROM 有 3 操作方式：读、写和擦除，其中擦除操作是以扇区为单位进行操作，每扇区为 512 字节，即每执行一次擦除命令就会擦除一个扇区，而读数据和写数据都是以字节为单位进行操作的，即每执行一次读或者写命令时只能读出或者写入一个字节。

STC8A8K64D4 系列单片机内部的 EEPROM 的访问方式有两种：IAP 方式和 MOVC 方式。IAP 方式可对 EEPROM 执行读、写、擦除操作，但 MOVC 只能对 EEPROM 进行读操作，而不能进行写和擦除操作。无论是使用 IAP 方式还是使用 MOVC 方式访问 EEPROM，首先都需要设置正确的目标地址。IAP 方式时，目标地址与 EEPROM 实际的物理地址是一致的，均是从地址 0000H 开始访问，但若要使用 MOVC 指令进行读取 EEPROM 数据时，目标地址必须是在 EEPROM 实际的物理地址的基础上还有加上程序大小的偏移。下面以 STC8A8K60D4 这个型号为例，对目标地址进行详细说明：



STC8A8K60D4

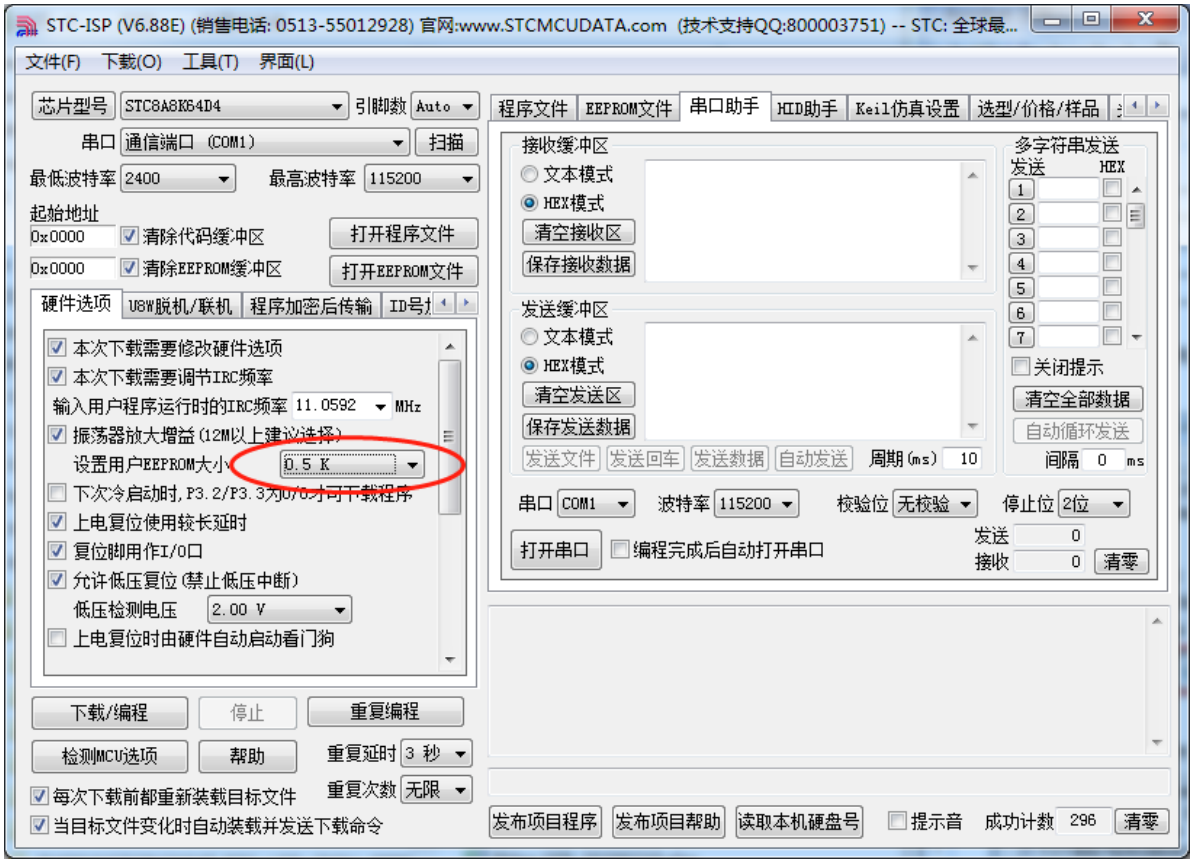
STC8A8K60D4 的程序空间为 60K 字节（0000h~EFFFh），EEPROM 空间为 4K（0000h~0FFFh）。当需要对 EEPROM 物理地址 0234h 的单元进行读、写、擦除时，若使用 IAP 方式进行访问时，设置的目标地址为 0234h，即 IAP_ADDRH 设置 02h，IAP_ADDRL 设置 34h，然后设置相应的触发命令即可对 0234h 单元进行正确操作了。但若是使用 MOVC 方式读取 EEPROM 的 0234h 单元，则必须在 0234h 的基础上还有加上 Flash 程序存储器（ROM）空间的大小 F000h，即必须将 DPTR 设置为 F234h，然后才能使用 MOVC 指令进行读取。

注意：由于擦除是以 512 字节为单位进行操作的，所以执行擦除操作时所设置的目标地址的低 9 位是无意义的。例如：执行擦除命令时，设置地址 0234H/0200H/0300H/03FFH，最终执行擦除的动作都是相同的，都是擦除 0200H~03FFH 这 512 字节。

不同型号内部 EEPROM 的大小及访问地址会存在差异，针对各个型号 EEPROM 的详细大小和地址请参考下表

型号	大小	扇区	IAP 方式读/写/擦除		MOVC 读取	
			起始地址	结束地址	起始地址	结束地址
STC8A8K16D4	48K	96	0000h	BFFFh	4000h	FFFFh
STC8A8K32D4	32K	64	0000h	7FFFh	8000h	FFFFh
STC8A8K60D4	4K	8	0000h	0FFFh	F000h	FFFFh
STC8A8K64D4	用户自定义[1]					

[1]：这个为特殊型号，这个型号的 EEPROM 大小是可用在 ISP 下载时用户自己设置的。如下图所示：



用户可用根据自己的需要在整个 FLASH 空间中规划出任意不超过 FLASH 大小的 EEPROM 空间，但需要注意：**EEPROM 总是从后向前进行规划的**。

例如：STC8A8K64D4 这个型号的 FLASH 为 64K，此时若用户想分出其中的 4K 作为 EEPROM 使用，则 EEPROM 的物理地址即为 64K 的最后 4K，物理地址为 F000h~FFFFh，当然，用户若使用 IAP 的方式进行访问，目标地址仍然从 0000h 开始，到 0FFFh 结束，当使用 MOVC 读取则需要从 F000h 开始，到 FFFFh 结束。

16.4 范例程序

16.4.1 EEPROM 基本操作

C 语言代码

```
//测试工作频率为 11.0592MHz

#include "reg51.h"
#include "intrins.h"

sfr         P0M1        =   0x93;
sfr         P0M0        =   0x94;
sfr         P1M1        =   0x91;
sfr         P1M0        =   0x92;
sfr         P2M1        =   0x95;
sfr         P2M0        =   0x96;
sfr         P3M1        =   0xb1;
sfr         P3M0        =   0xb2;
sfr         P4M1        =   0xb3;
sfr         P4M0        =   0xb4;
sfr         P5M1        =   0xc9;
sfr         P5M0        =   0xca;

sfr         IAP_DATA    =   0xC2;
sfr         IAP_ADDRH   =   0xC3;
sfr         IAP_ADDRL   =   0xC4;
sfr         IAP_CMD     =   0xC5;
sfr         IAP_TRIG    =   0xC6;
sfr         IAP_CONTR   =   0xC7;
sfr         IAP_TPS     =   0xF5;

void IapIdle()
{
    IAP_CONTR = 0;                  //关闭 IAP 功能
    IAP_CMD = 0;                    //清除命令寄存器
    IAP_TRIG = 0;                   //清除触发寄存器
    IAP_ADDRH = 0x80;               //将地址设置到非 IAP 区域
    IAP_ADDRL = 0;
}

char IapRead(int addr)
{
    char dat;

    IAP_CONTR = 0x80;               //使能 IAP
    IAP_TPS = 12;                   //设置等待参数 12MHz
    IAP_CMD = 1;                    //设置 IAP 读命令
    IAP_ADDRL = addr;               //设置 IAP 低地址
    IAP_ADDRH = addr >> 8;          //设置 IAP 高地址
    IAP_TRIG = 0x5a;                //写触发命令(0x5a)
    IAP_TRIG = 0xa5;                //写触发命令(0xa5)
    _nop_();
    dat = IAP_DATA;                 //读 IAP 数据
    IapIdle();                      //关闭 IAP 功能
```

```
    return dat;
}

void IapProgram(int addr, char dat)
{
    IAP_CONTR = 0x80;                   //使能 IAP
    IAP_TPS = 12;                       //设置等待参数 12MHz
    IAP_CMD = 2;                        //设置 IAP 写命令
    IAP_ADDRL = addr;                   //设置 IAP 低地址
    IAP_ADDRH = addr >> 8;              //设置 IAP 高地址
    IAP_DATA = dat;                     //写 IAP 数据
    IAP_TRIG = 0x5a;                    //写触发命令(0x5a)
    IAP_TRIG = 0xa5;                    //写触发命令(0xa5)
    _nop_();
    IapIdle();                          //关闭 IAP 功能
}

void IapErase(int addr)
{
    IAP_CONTR = 0x80;                   //使能 IAP
    IAP_TPS = 12;                       //设置等待参数 12MHz
    IAP_CMD = 3;                        //设置 IAP 擦除命令
    IAP_ADDRL = addr;                   //设置 IAP 低地址
    IAP_ADDRH = addr >> 8;              //设置 IAP 高地址
    IAP_TRIG = 0x5a;                    //写触发命令(0x5a)
    IAP_TRIG = 0xa5;                    //写触发命令(0xa5)
    _nop_();                            //
    IapIdle();                          //关闭 IAP 功能
}

void main()
{
    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    IapErase(0x0400);
    P0 = IapRead(0x0400);               //P0=0xff
    IapProgram(0x0400, 0x12);
    P1 = IapRead(0x0400);               //P1=0x12

    while (1);
}
```

汇编代码

```
;测试工作频率为 11.0592MHz

IAP_DATA    DATA    0C2H
IAP_ADDRH   DATA    0C3H
```

```
IAP_ADDRL   DATA    0C4H
IAP_CMD     DATA    0C5H
IAP_TRIG    DATA    0C6H
IAP_CONTR   DATA    0C7H
IAP_TPS     DATA    0F5H

P0M1        DATA    093H
P0M0        DATA    094H
P1M1        DATA    091H
P1M0        DATA    092H
P2M1        DATA    095H
P2M0        DATA    096H
P3M1        DATA    0B1H
P3M0        DATA    0B2H
P4M1        DATA    0B3H
P4M0        DATA    0B4H
P5M1        DATA    0C9H
P5M0        DATA    0CAH

            ORG     0000H
            LJMP    MAIN

            ORG     0100H

IAP_IDLE:
            MOV     IAP_CONTR,#0        ;关闭 IAP 功能
            MOV     IAP_CMD,#0          ;清除命令寄存器
            MOV     IAP_TRIG,#0         ;清除触发寄存器
            MOV     IAP_ADDRH,#80H      ;将地址设置到非 IAP 区域
            MOV     IAP_ADDRL,#0
            RET

IAP_READ:
            MOV     IAP_CONTR,#80H      ;使能 IAP
            MOV     IAP_TPS,#12         ;设置等待参数 12MHz
            MOV     IAP_CMD,#1          ;设置 IAP 读命令
            MOV     IAP_ADDRL,DPL       ;设置 IAP 低地址
            MOV     IAP_ADDRH,DPH       ;设置 IAP 高地址
            MOV     IAP_TRIG,#5AH       ;写触发命令(0x5a)
            MOV     IAP_TRIG,#0A5H      ;写触发命令(0xa5)
            NOP
            MOV     A,IAP_DATA          ;读取 IAP 数据
            LCALL   IAP_IDLE            ;关闭 IAP 功能
            RET

IAP_PROGRAM:
            MOV     IAP_CONTR,#80H      ;使能 IAP
            MOV     IAP_TPS,#12         ;设置等待参数 12MHz
            MOV     IAP_CMD,#2          ;设置 IAP 写命令
            MOV     IAP_ADDRL,DPL       ;设置 IAP 低地址
            MOV     IAP_ADDRH,DPH       ;设置 IAP 高地址
            MOV     IAP_DATA,A          ;写 IAP 数据
            MOV     IAP_TRIG,#5AH       ;写触发命令(0x5a)
            MOV     IAP_TRIG,#0A5H      ;写触发命令(0xa5)
            NOP
            LCALL   IAP_IDLE            ;关闭 IAP 功能
            RET

IAP_ERASE:
```

```
        MOV     IAP_CONTR,#80H          ;使能 IAP
        MOV     IAP_TPS,#12             ;设置等待参数 12MHz
        MOV     IAP_CMD,#3              ;设置 IAP 擦除命令
        MOV     IAP_ADDRL,DPL           ;设置 IAP 低地址
        MOV     IAP_ADDRH,DPH           ;设置 IAP 高地址
        MOV     IAP_TRIG,#5AH           ;写触发命令(0x5a)
        MOV     IAP_TRIG,#0A5H          ;写触发命令(0xa5)
        NOP
        LCALL   IAP_IDLE                ;关闭 IAP 功能
        RET

MAIN:
        MOV     SP, #5FH
        MOV     P0M0, #00H
        MOV     P0M1, #00H
        MOV     P1M0, #00H
        MOV     P1M1, #00H
        MOV     P2M0, #00H
        MOV     P2M1, #00H
        MOV     P3M0, #00H
        MOV     P3M1, #00H
        MOV     P4M0, #00H
        MOV     P4M1, #00H
        MOV     P5M0, #00H
        MOV     P5M1, #00H

        MOV     DPTR,#0400H
        LCALL   IAP_ERASE
        MOV     DPTR,#0400H
        LCALL   IAP_READ
        MOV     P0,A                    ;P0=0FFH
        MOV     DPTR,#0400H
        MOV     A,#12H
        LCALL   IAP_PROGRAM
        MOV     DPTR,#0400H
        LCALL   IAP_READ
        MOV     P1,A                    ;P1=12H

        SJMP    $

        END
```

16.4.2 使用 MOVC 读取 EEPROM

C 语言代码

```
//测试工作频率为 11.0592MHz

#include "reg51.h"
#include "intrins.h"

sfr     P0M1    =   0x93;
sfr     P0M0    =   0x94;
sfr     P1M1    =   0x91;
sfr     P1M0    =   0x92;
sfr     P2M1    =   0x95;
sfr     P2M0    =   0x96;
```

```
sfr         P3M1        =   0xb1;
sfr         P3M0        =   0xb2;
sfr         P4M1        =   0xb3;
sfr         P4M0        =   0xb4;
sfr         P5M1        =   0xc9;
sfr         P5M0        =   0xca;

sfr         IAP_DATA    =   0xC2;
sfr         IAP_ADDRH   =   0xC3;
sfr         IAP_ADDRL   =   0xC4;
sfr         IAP_CMD     =   0xC5;
sfr         IAP_TRIG    =   0xC6;
sfr         IAP_CONTR   =   0xC7;
sfr         IAP_TPS     =   0xF5;

#define     IAP_OFFSET      0xF000H             //STC8A8K60S4
void IapIdle()
{
    IAP_CONTR = 0;                              //关闭 IAP 功能
    IAP_CMD = 0;                                //清除命令寄存器
    IAP_TRIG = 0;                               //清除触发寄存器
    IAP_ADDRH = 0x80;                           //将地址设置到非 IAP 区域
    IAP_ADDRL = 0;
}

char IapRead(int addr)
{
    addr += IAP_OFFSET;                         //使用 MOVC 读取 EEPROM 需要加上相应的偏移
    return *(char code *)(addr);                //使用 MOVC 读取数据
}

void IapProgram(int addr, char dat)
{
    IAP_CONTR = 0x80;                           //使能 IAP
    IAP_TPS = 12;                               //设置等待参数 12MHz
    IAP_CMD = 2;                                //设置 IAP 写命令
    IAP_ADDRL = addr;                           //设置 IAP 低地址
    IAP_ADDRH = addr >> 8;                      //设置 IAP 高地址
    IAP_DATA = dat;                             //写 IAP 数据
    IAP_TRIG = 0x5a;                            //写触发命令(0x5a)
    IAP_TRIG = 0xa5;                            //写触发命令(0xa5)
    _nop_();
    IapIdle();                                  //关闭 IAP 功能
}

void IapErase(int addr)
{
    IAP_CONTR = 0x80;                           //使能 IAP
    IAP_TPS = 12;                               //设置等待参数 12MHz
    IAP_CMD = 3;                                //设置 IAP 擦除命令
    IAP_ADDRL = addr;                           //设置 IAP 低地址
    IAP_ADDRH = addr >> 8;                      //设置 IAP 高地址
    IAP_TRIG = 0x5a;                            //写触发命令(0x5a)
    IAP_TRIG = 0xa5;                            //写触发命令(0xa5)
    _nop_();                                    //
    IapIdle();                                  //关闭 IAP 功能
}

void main()
```

```
{
    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    IapErase(0x0400);
    P0 = IapRead(0x0400);                   //P0=0xff
    IapProgram(0x0400, 0x12);
    P1 = IapRead(0x0400);                   //P1=0x12

    while (1);
}
```

汇编代码

```
;测试工作频率为 11.0592MHz

IAP_DATA    DATA    0C2H
IAP_ADDRH   DATA    0C3H
IAP_ADDRL   DATA    0C4H
IAP_CMD     DATA    0C5H
IAP_TRIG    DATA    0C6H
IAP_CONTR   DATA    0C7H
IAP_TPS     DATA    0F5H

IAP_OFFSET  EQU     0F000H              ;STC8A8K60S4

P0M1        DATA    093H
P0M0        DATA    094H
P1M1        DATA    091H
P1M0        DATA    092H
P2M1        DATA    095H
P2M0        DATA    096H
P3M1        DATA    0B1H
P3M0        DATA    0B2H
P4M1        DATA    0B3H
P4M0        DATA    0B4H
P5M1        DATA    0C9H
P5M0        DATA    0CAH

            ORG     0000H
            LJMP    MAIN

            ORG     0100H

IAP_IDLE:
            MOV     IAP_CONTR,#0        ;关闭 IAP 功能
            MOV     IAP_CMD,#0          ;清除命令寄存器
            MOV     IAP_TRIG,#0         ;清除触发寄存器
            MOV     IAP_ADDRH,#80H      ;将地址设置到非 IAP 区域
```

```
        MOV     IAP_ADDRL,#0
        RET

IAP_READ:
        MOV     A,#LOW IAP_OFFSET       ;使用 MOVC 读取 EEPROM 需要加上相应的偏移
        ADD     A,DPL
        MOV     DPL,A
        MOV     A,@HIGH IAP_OFFSET
        ADDC    A,DPH
        MOV     DPH,A
        CLR     A
        MOVC    A,@A+DPTR               ;使用 MOVC 读取数据
        RET

IAP_PROGRAM:
        MOV     IAP_CONTR,#80H          ;使能 IAP
        MOV     IAP_TPS,#12             ;设置等待参数 12MHz
        MOV     IAP_CMD,#2              ;设置 IAP 写命令
        MOV     IAP_ADDRL,DPL           ;设置 IAP 低地址
        MOV     IAP_ADDRH,DPH           ;设置 IAP 高地址
        MOV     IAP_DATA,A              ;写 IAP 数据
        MOV     IAP_TRIG,#5AH           ;写触发命令(0x5a)
        MOV     IAP_TRIG,#0A5H          ;写触发命令(0xa5)
        NOP
        LCALL   IAP_IDLE                ;关闭 IAP 功能
        RET

IAP_ERASE:
        MOV     IAP_CONTR,#80H          ;使能 IAP
        MOV     IAP_TPS,#12             ;设置等待参数 12MHz
        MOV     IAP_CMD,#3              ;设置 IAP 擦除命令
        MOV     IAP_ADDRL,DPL           ;设置 IAP 低地址
        MOV     IAP_ADDRH,DPH           ;设置 IAP 高地址
        MOV     IAP_TRIG,#5AH           ;写触发命令(0x5a)
        MOV     IAP_TRIG,#0A5H          ;写触发命令(0xa5)
        NOP
        LCALL   IAP_IDLE                ;关闭 IAP 功能
        RET

MAIN:
        MOV     SP, #5FH
        MOV     P0M0, #00H
        MOV     P0M1, #00H
        MOV     P1M0, #00H
        MOV     P1M1, #00H
        MOV     P2M0, #00H
        MOV     P2M1, #00H
        MOV     P3M0, #00H
        MOV     P3M1, #00H
        MOV     P4M0, #00H
        MOV     P4M1, #00H
        MOV     P5M0, #00H
        MOV     P5M1, #00H

        MOV     DPTR,#0400H
        LCALL   IAP_ERASE
        MOV     DPTR,#0400H
        LCALL   IAP_READ
        MOV     P0,A                    ;P0=0FFH
```

```
        MOV     DPTR,#0400H
        MOV     A,#12H
        LCALL   IAP_PROGRAM
        MOV     DPTR,#0400H
        LCALL   IAP_READ
        MOV     P1,A                    ;P1=12H

        SJMP    $

        END
```

16.4.3 使用串口送出 EEPROM 数据

C 语言代码

```
//测试工作频率为 11.0592MHz

#include "reg51.h"
#include "intrins.h"

#define   FOSC          11059200UL
#define   BRT           (65536 - FOSC / 115200 / 4)

sfr       P0M1          =   0x93;
sfr       P0M0          =   0x94;
sfr       P1M1          =   0x91;
sfr       P1M0          =   0x92;
sfr       P2M1          =   0x95;
sfr       P2M0          =   0x96;
sfr       P3M1          =   0xb1;
sfr       P3M0          =   0xb2;
sfr       P4M1          =   0xb3;
sfr       P4M0          =   0xb4;
sfr       P5M1          =   0xc9;
sfr       P5M0          =   0xca;

sfr       AUXR          =   0x8e;
sfr       T2H           =   0xd6;
sfr       T2L           =   0xd7;

sfr       IAP_DATA      =   0xC2;
sfr       IAP_ADDRH     =   0xC3;
sfr       IAP_ADDRL     =   0xC4;
sfr       IAP_CMD       =   0xC5;
sfr       IAP_TRIG      =   0xC6;
sfr       IAP_CONTR     =   0xC7;
sfr       IAP_TPS       =   0xF5;

void UartInit()
{
    SCON = 0x5a;
    T2L = BRT;
    T2H = BRT >> 8;
    AUXR = 0x15;
}

void UartSend(char dat)
```

```
{
    while (!TI);
    TI = 0;
    SBUF = dat;
}

void IapIdle()
{
    IAP_CONTR = 0;                  //关闭 IAP 功能
    IAP_CMD = 0;                    //清除命令寄存器
    IAP_TRIG = 0;                   //清除触发寄存器
    IAP_ADDRH = 0x80;               //将地址设置到非 IAP 区域
    IAP_ADDRL = 0;
}

char IapRead(int addr)
{
    char dat;

    IAP_CONTR = 0x80;               //使能 IAP
    IAP_TPS = 12;                   //设置等待参数 12MHz
    IAP_CMD = 1;                    //设置 IAP 读命令
    IAP_ADDRL = addr;               //设置 IAP 低地址
    IAP_ADDRH = addr >> 8;          //设置 IAP 高地址
    IAP_TRIG = 0x5a;                //写触发命令(0x5a)
    IAP_TRIG = 0xa5;                //写触发命令(0xa5)
    _nop_();
    dat = IAP_DATA;                 //读 IAP 数据
    IapIdle();                      //关闭 IAP 功能

    return dat;
}

void IapProgram(int addr, char dat)
{
    IAP_CONTR = 0x80;               //使能 IAP
    IAP_TPS = 12;                   //设置等待参数 12MHz
    IAP_CMD = 2;                    //设置 IAP 写命令
    IAP_ADDRL = addr;               //设置 IAP 低地址
    IAP_ADDRH = addr >> 8;          //设置 IAP 高地址
    IAP_DATA = dat;                 //写 IAP 数据
    IAP_TRIG = 0x5a;                //写触发命令(0x5a)
    IAP_TRIG = 0xa5;                //写触发命令(0xa5)
    _nop_();
    IapIdle();                      //关闭 IAP 功能
}

void IapErase(int addr)
{
    IAP_CONTR = 0x80;               //使能 IAP
    IAP_TPS = 12;                   //设置等待参数 12MHz
    IAP_CMD = 3;                    //设置 IAP 擦除命令
    IAP_ADDRL = addr;               //设置 IAP 低地址
    IAP_ADDRH = addr >> 8;          //设置 IAP 高地址
    IAP_TRIG = 0x5a;                //写触发命令(0x5a)
    IAP_TRIG = 0xa5;                //写触发命令(0xa5)
    _nop_();                        //
    IapIdle();                      //关闭 IAP 功能
}
```

```
void main()
{
    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    UartInit();
    IapErase(0x0400);
    UartSend(IapRead(0x0400));
    IapProgram(0x0400, 0x12);
    UartSend(IapRead(0x0400));

    while (1);
}
```

汇编代码

```
;测试工作频率为 11.0592MHz

AUXR        DATA    8EH
T2H         DATA    0D6H
T2L         DATA    0D7H

IAP_DATA    DATA    0C2H
IAP_ADDRH   DATA    0C3H
IAP_ADDRL   DATA    0C4H
IAP_CMD     DATA    0C5H
IAP_TRIG    DATA    0C6H
IAP_CONTR   DATA    0C7H
IAP_TPS     DATA    0F5H

P0M1        DATA    093H
P0M0        DATA    094H
P1M1        DATA    091H
P1M0        DATA    092H
P2M1        DATA    095H
P2M0        DATA    096H
P3M1        DATA    0B1H
P3M0        DATA    0B2H
P4M1        DATA    0B3H
P4M0        DATA    0B4H
P5M1        DATA    0C9H
P5M0        DATA    0CAH

            ORG     0000H
            LJMP    MAIN

            ORG     0100H
```

```
UART_INIT:
        MOV     SCON,#5AH
        MOV     T2L,#0E8H       ;65536-11059200/115200/4=0FFE8H
        MOV     T2H,#0FFH
        MOV     AUXR,#15H
        RET

UART_SEND:
        JNB     TI,$
        CLR     TI
        MOV     SBUF,A
        RET

IAP_IDLE:
        MOV     IAP_CONTR,#0    ;关闭 IAP 功能
        MOV     IAP_CMD,#0      ;清除命令寄存器
        MOV     IAP_TRIG,#0     ;清除触发寄存器
        MOV     IAP_ADDRH,#80H  ;将地址设置到非 IAP 区域
        MOV     IAP_ADDRL,#0
        RET

IAP_READ:
        MOV     IAP_CONTR,#80H  ;使能 IAP
        MOV     IAP_TPS,#12     ;设置等待参数 12MHz
        MOV     IAP_CMD,#1      ;设置 IAP 读命令
        MOV     IAP_ADDRL,DPL   ;设置 IAP 低地址
        MOV     IAP_ADDRH,DPH   ;设置 IAP 高地址
        MOV     IAP_TRIG,#5AH   ;写触发命令(0x5a)
        MOV     IAP_TRIG,#0A5H  ;写触发命令(0xa5)
        NOP
        MOV     A,IAP_DATA      ;读取 IAP 数据
        LCALL   IAP_IDLE        ;关闭 IAP 功能
        RET

IAP_PROGRAM:
        MOV     IAP_CONTR,#80H  ;使能 IAP
        MOV     IAP_TPS,#12     ;设置等待参数 12MHz
        MOV     IAP_CMD,#2      ;设置 IAP 写命令
        MOV     IAP_ADDRL,DPL   ;设置 IAP 低地址
        MOV     IAP_ADDRH,DPH   ;设置 IAP 高地址
        MOV     IAP_DATA,A      ;写 IAP 数据
        MOV     IAP_TRIG,#5AH   ;写触发命令(0x5a)
        MOV     IAP_TRIG,#0A5H  ;写触发命令(0xa5)
        NOP
        LCALL   IAP_IDLE        ;关闭 IAP 功能
        RET

IAP_ERASE:
        MOV     IAP_CONTR,#80H  ;使能 IAP
        MOV     IAP_TPS,#12     ;设置等待参数 12MHz
        MOV     IAP_CMD,#3      ;设置 IAP 擦除命令
        MOV     IAP_ADDRL,DPL   ;设置 IAP 低地址
        MOV     IAP_ADDRH,DPH   ;设置 IAP 高地址
        MOV     IAP_TRIG,#5AH   ;写触发命令(0x5a)
        MOV     IAP_TRIG,#0A5H  ;写触发命令(0xa5)
        NOP
        LCALL   IAP_IDLE        ;关闭 IAP 功能
        RET
```

```
MAIN:
        MOV     SP, #5FH
        MOV     P0M0, #00H
        MOV     P0M1, #00H
        MOV     P1M0, #00H
        MOV     P1M1, #00H
        MOV     P2M0, #00H
        MOV     P2M1, #00H
        MOV     P3M0, #00H
        MOV     P3M1, #00H
        MOV     P4M0, #00H
        MOV     P4M1, #00H
        MOV     P5M0, #00H
        MOV     P5M1, #00H

        LCALL   UART_INIT
        MOV     DPTR,#0400H
        LCALL   IAP_ERASE
        MOV     DPTR,#0400H
        LCALL   IAP_READ
        LCALL   UART_SEND
        MOV     DPTR,#0400H
        MOV     A,#12H
        LCALL   IAP_PROGRAM
        MOV     DPTR,#0400H
        LCALL   IAP_READ
        LCALL   UART_SEND

        SJMP    $

        END
```

16.4.4 串口 1 读写 EEPROM-带 MOVC 读

C 语言代码（main.c）

```
//测试工作频率为 11.0592MHz

/*      本程序经过测试完全正常，不提供电话技术支持，如不能理解，请自行补充相关基础。  */

/************ 本程序功能说明      **************
STC8G 系列 EEPROM 通用测试程序.
请先别修改程序，直接下载"02-串口 1 读写 EEPROM-带 MOVC 读"里面的"UART-EEPROM.hex"测试。下载时选择主频 11.0592MHZ.
PC 串口设置：波特率 115200,8,n,1.
对 EEPROM 做扇区擦除、写入 64 字节、读出 64 字节的操作。

命令例子:
E 0     对 EEPROM 进行扇区擦除操作，E 表示擦除，数字 0 为 0 扇区(十进制，0~126，看具体 IC).
W 0     对 EEPROM 进行写入操作，W 表示写入，数字 0 为 0 扇区(十进制，0~126，看具体 IC). 从扇区的开始地址连续写 64 字节.
R 0     对 EEPROM 进行 IAP 读出操作，R 表示读出，数字 0 为 0 扇区(十进制，0~126，看具体 IC). 从扇区的开始地址连续读 64 字节.
M 0     对 EEPROM 进行 MOVC 读出操作(操作地址为扇区*512+偏移地址) 数字 0 为 0 扇区(十进制，0~126，看具体 IC). 从扇区的开始地址连续读 64 字节.
```

```
注意：为了通用，程序不识别扇区是否有效，用户自己根据具体的型号来决定。

日期: 2019-6-10
*****************************************/

#include   "config.H"
#include   "EEPROM.h"

#define    Baudrate1                115200L
#define    UART1_BUF_LENGTH         10
#define    EEADDR_OFFSET            (8 * 1024)      //定义 EEPROM 用 MOVC 访问时加的偏移量,
                                                    //等于 FLASH ROM 的大小,对于 IAP 或 IRC 开头的,
                                                    //则偏移量必须为 0
#define    TimeOutSet1              5

/************* 本地常量声明  **************/
u8    code  T_Strings[]={"去年今日此门中，人面桃花相映红。人面不知何处去，桃花依旧笑春风。"};

/************* 本地变量声明  **************/
u8    xdatatmp[70];
u8    xdataRX1_Buffer[UART1_BUF_LENGTH];
u8    RX1_Cnt;
u8    RX1_TimeOut;
bit   B_TX1_Busy;

/************* 本地函数声明  **************/
void  UART1_config(void);
void  TX1_write2buff(u8 dat);                       //写入发送缓冲
void  PrintString1(u8 *puts);                       //发送一个字符串

/*************  外部函数和变量声明 *****************/

/*****************************************/

u8    CheckData(u8 dat)
{
      if((dat >= '0') && (dat <= '9'))    return (dat-'0');
      if((dat >= 'A') && (dat <= 'F'))    return (dat-'A'+10);
      if((dat >= 'a') && (dat <= 'f'))    return (dat-'a'+10);
      return 0xff;
}

u16   GetAddress(void)
{
      u16   address;
      u8    i;

      address = 0;
      if(RX1_Cnt <   3)      return 65535;            //error
      if(RX1_Cnt <= 5)                                //5 个字节以内是扇区操作，十进制,
                                                      //支持命令:      E 0, E 12, E 120
                                                      //               W 0, W 12, W 120
                                                      //               R 0, R 12, R 120
      {
            for(i=2; i<RX1_Cnt; i++)
            {
                  if(CheckData(RX1_Buffer[i]) > 9)
                        return 65535;                 //error
                  address = address * 10 + CheckData(RX1_Buffer[i]);
```

```
        }
        if(address < 124)                               //限制在 0~123 扇区
        {
            address <<= 9;
            return (address);
        }
    }
    else if(RX1_Cnt == 8)                               //8 个字节直接地址操作，十六进制,
                                                        //支持命令: E 0x1234, W 0x12b3, R 0x0A00
    {
        if((RX1_Buffer[2] == '0') && ((RX1_Buffer[3] == 'x') || (RX1_Buffer[3] == 'X')))
        {
            for(i=4; i<8; i++)
            {
                if(CheckData(RX1_Buffer[i]) > 0x0F)
                    return 65535;                       //error
                address = (address << 4) + CheckData(RX1_Buffer[i]);
            }
            if(address < 63488)
                return (address);                       //限制在 0~123 扇区
        }
    }

     return     65535;                                  //error
}

//========================================================================
// 函数: void    delay_ms(u8 ms)
// 描述:  延时函数。
// 参数: ms,要延时的 ms 数,  这里只支持 1~255ms.  自动适应主时钟.
// 返回: none.
// 版本: VER1.0
// 日期: 2013-4-1
// 备注:
//========================================================================
void    delay_ms(u8 ms)
{
    u16 i;
    do
    {
        i = MAIN_Fosc / 10000;
        while(--i)   ;
    }while(--ms);
}

//使用 MOVC 读 EEPROM
void EEPROM_MOVC_read_n(u16 EE_address, u8 *DataAddress, u16 number)
{
    u8    code  *pc;

    pc = EE_address + EEADDR_OFFSET;
    do
    {
        *DataAddress = *pc;                             //读出的数据
        DataAddress++;
        pc++;
    }while(--number);
}
```

```
/********************** 主函数 *************************/
void main(void)
{
    u8    i;
    u16 addr;

    UART1_config();                                 // 选择波特率, 2: 使用 Timer2 做波特率,
                                                    //其它值: 使用 Timer1 做波特率.
    EA = 1;                                         //允许总中断

    PrintString1("STC8 系列 MCU 用串口 1 测试 EEPROM 程序!\r\n");   //UART1 发送一个字符串

    while(1)
    {
        delay_ms(1);
        if(RX1_TimeOut > 0)                         //超时计数
        {
            if(--RX1_TimeOut == 0)
            {
                if(RX1_Buffer[1] == ' ')
                {
                    addr = GetAddress();
                    if(addr < 63488)                //限制在 0~123 扇区
                    {
                        if(RX1_Buffer[0] == 'E')    //PC 请求擦除一个扇区
                        {
                            EEPROM_SectorErase(addr);
                            PrintString1("扇区擦除完成!\r\n");
                        }

                        else if(RX1_Buffer[0] == 'W')           //PC 请求写入 EEPROM 64 字节数据
                        {
                            EEPROM_write_n(addr,T_Strings,64);
                            PrintString1("写入操作完成!\r\n");
                        }

                        else if(RX1_Buffer[0] == 'R')           //PC 请求返回 64 字节 EEPROM 数据
                        {
                            PrintString1("IAP 读出的数据如下：\r\n");
                            EEPROM_read_n(addr,tmp,64);
                            for(i=0; i<64; i++)
                                TX1_write2buff(tmp[i]);     //将数据返回给串口
                            TX1_write2buff(0x0d);
                            TX1_write2buff(0x0a);
                        }
                        else if(RX1_Buffer[0] == 'M')           //PC 请求返回 64 字节 EEPROM 数据
                        {
                            PrintString1("MOVC 读出的数据如下：\r\n");
                            EEPROM_MOVC_read_n(addr,tmp,64);
                            for(i=0; i<64; i++)
                                TX1_write2buff(tmp[i]);     //将数据返回给串口
                            TX1_write2buff(0x0d);
                            TX1_write2buff(0x0a);
                        }
                        else   PrintString1("命令错误!\r\n");
                    }
                    else   PrintString1("命令错误!\r\n");
                }
```

```
                    RX1_Cnt = 0;
                }
            }
        }
}
/*********************************************/

/*************** 发送一个字节 *****************************/
void TX1_write2buff(u8 dat)                     //写入发送缓冲
{
    B_TX1_Busy = 1;                             //标志发送忙
    SBUF = dat;                                 //发送一个字节
    while(B_TX1_Busy);                          //等待发送完毕
}

//========================================================================
// 函数: void PrintString1(u8 *puts)
// 描述: 串口 1 发送字符串函数。
// 参数: puts:   字符串指针.
// 返回: none.
// 版本: VER1.0
// 日期: 2014-11-28
// 备注:
//========================================================================
void PrintString1(u8 *puts)                     //发送一个字符串
{
    for (; *puts != 0;  puts++)                 //遇到停止符 0 结束
    {
        TX1_write2buff(*puts);
    }
}

//========================================================================
// 函数: void    UART1_config(void)
// 描述: UART1 初始化函数。
// 参数: none.
// 返回: none.
// 版本: VER1.0
// 日期: 2014-11-28
// 备注:
//========================================================================
void  UART1_config(void)
{
    TR1 = 0;
    AUXR &= ~0x01;                              //S1 BRT Use Timer1;
    AUXR |=  (1<<6);                            //Timer1 set as 1T mode
    TMOD &= ~(1<<6);                            //Timer1 set As Timer
    TMOD &= ~0x30;                              //Timer1_16bitAutoReload;
    TH1 = (u8)((65536L-(MAIN_Fosc / 4) / Baudrate1) >> 8);
    TL1 = (u8)(65536L-(MAIN_Fosc / 4) / Baudrate1);
    ET1 = 0;                                    // 禁止 Timer1 中断
    INT_CLKO &= ~0x02;                          // Timer1 不输出高速时钟
    TR1  = 1;                                   // 运行 Timer1

    S1_USE_P30P31();    P3n_standard(0x03);     //切换到 P3.0 P3.1
    //S1_USE_P36P37();  P3n_standard(0xc0);     //切换到 P3.6 P3.7
    //S1_USE_P16P17();  P1n_standard(0xc0);     //切换到 P1.6 P1.7

    SCON = (SCON & 0x3f) | 0x40;                //UART1 模式, 0x00: 同步移位输出,
```

```
                                                //                  0x40: 8 位数据,可变波特率
                                                //                  0x80: 9 位数据,固定波特率
                                                //                  0xc0: 9 位数据,可变波特率
//      PS    = 1;                              //高优先级中断
        ES    = 1;                              //允许中断
        REN = 1;                                //允许接收

        B_TX1_Busy = 0;
        RX1_Cnt = 0;
}

//========================================================================
// 函数: void UART1_int (void) interrupt UART1_VECTOR
// 描述: UART1 中断函数。
// 参数: nine.
// 返回: none.
// 版本: VER1.0
// 日期: 2014-11-28
// 备注:
//========================================================================
void UART1_int (void) interrupt 4
{
        if(RI)
        {
                RI = 0;
                if(RX1_Cnt >= UART1_BUF_LENGTH)
                        RX1_Cnt = 0;                    //防溢出
                RX1_Buffer[RX1_Cnt++] = SBUF;
                RX1_TimeOut = TimeOutSet1;
        }

        if(TI)
        {
                TI = 0;
                B_TX1_Busy = 0;
        }
}
```

C 语言代码（EEPROM.c）

```
//测试工作频率为 11.0592MHz

//      本程序是 STC 系列的内置 EEPROM 读写程序。

#include "config.h"
#include "eeprom.h"

//========================================================================
// 函数: void      ISP_Disable(void)
// 描述:  禁止访问 ISP/IAP.
// 参数: non.
// 返回: non.
// 版本: V1.0, 2012-10-22
//========================================================================
void  DisableEEPROM(void)
{
        ISP_CONTR = 0;                          //禁止 ISP/IAP 操作
        IAP_TPS     = 0;
        ISP_CMD     = 0;                        //去除 ISP/IAP 命令
```

```
    ISP_TRIG   = 0;                         //防止 ISP/IAP 命令误触发
    ISP_ADDRH = 0xff;                       //清 0 地址高字节
    ISP_ADDRL = 0xff;                       //清 0 地址低字节，指向非 EEPROM 区，防止误操作
}

//=========================================================================
// 函数: void EEPROM_read_n(u16 EE_address,u8 *DataAddress,u16 number)
// 描述: 从指定 EEPROM 首地址读出 n 个字节放指定的缓冲.
// 参数: EE_address:   读出 EEPROM 的首地址.
//       DataAddress:  读出数据放缓冲的首地址.
//       number:       读出的字节长度.
// 返回: non.
// 版本: V1.0, 2012-10-22
//=========================================================================
void EEPROM_read_n(u16 EE_address,u8 *DataAddress,u16 number)
{
    EA = 0;                                 //禁止中断
    ISP_CONTR = ISP_EN;                     //允许 ISP/IAP 操作
    IAP_TPS = (u8)(MAIN_Fosc / 1000000L);   //工作频率设置
    ISP_READ();                             //送字节读命令，命令不需改变时，不需重新送命令
    do
    {
        ISP_ADDRH = EE_address / 256;       //送地址高字节（地址需要改变时才需重新送地址）
        ISP_ADDRL = EE_address % 256;       //送地址低字节
        ISP_TRIG();                         //先送 5AH，再送 A5H 到 ISP/IAP 触发寄存器，
                                            //每次都需要如此
                                            //送完 A5H 后，ISP/IAP 命令立即被触发启动
                                            //CPU 等待 IAP 完成后，才会继续执行程序。
        _nop_();
        _nop_();
        _nop_();
        *DataAddress = ISP_DATA;            //读出的数据送往
        EE_address++;
        DataAddress++;
    }while(--number);

    DisableEEPROM();
    EA = 1;                                 //重新允许中断
}

/********************* 扇区擦除函数 *****************/
//=========================================================================
// 函数: void EEPROM_SectorErase(u16 EE_address)
// 描述: 把指定地址的 EEPROM 扇区擦除.
// 参数: EE_address:   要擦除的扇区 EEPROM 的地址.
// 返回: non.
// 版本: V1.0, 2013-5-10
//=========================================================================
void EEPROM_SectorErase(u16 EE_address)
{
    EA = 0;                                 //禁止中断
                                            //只有扇区擦除，没有字节擦除，512 字节/扇区。
                                            //扇区中任意一个字节地址都是扇区地址。
    ISP_ADDRH = EE_address / 256;           //送扇区地址高字节（地址需要改变时才需重新送地址）
    ISP_ADDRL = EE_address % 256;           //送扇区地址低字节
    ISP_CONTR = ISP_EN;                     //允许 ISP/IAP 操作
    IAP_TPS = (u8)(MAIN_Fosc / 1000000L);   //工作频率设置
    ISP_ERASE();                            //送扇区擦除命令，命令不需改变时，不需重新送命令
    ISP_TRIG();
```

```
    _nop_();
    _nop_();
    _nop_();
    DisableEEPROM();
    EA = 1;                                     //重新允许中断
}

//=========================================================================
// 函数: void EEPROM_write_n(u16 EE_address,u8 *DataAddress,u16 number)
// 描述: 把缓冲的 n 个字节写入指定首地址的 EEPROM.
// 参数: EE_address:  写入 EEPROM 的首地址.
//       DataAddress: 写入源数据的缓冲的首地址.
//       number:      写入的字节长度.
// 返回: non.
// 版本: V1.0, 2012-10-22
//=========================================================================
void EEPROM_write_n(u16 EE_address,u8 *DataAddress,u16 number)
{
    EA = 0;                                     //禁止中断

    ISP_CONTR = ISP_EN;                         //允许 ISP/IAP 操作
    IAP_TPS = (u8)(MAIN_Fosc / 1000000L);       //工作频率设置
    ISP_WRITE();                                //送字节写命令，命令不需改变时，不需重新送命令
    do
    {
        ISP_ADDRH = EE_address / 256;           //送地址高字节（地址需要改变时才需重新送地址）
        ISP_ADDRL = EE_address % 256;           //送地址低字节
        ISP_DATA  = *DataAddress;               //送数据到 ISP_DATA，只有数据改变时才需重新送
        ISP_TRIG();
        _nop_();
        _nop_();
        _nop_();
        EE_address++;
        DataAddress++;
    }while(--number);

    DisableEEPROM();
    EA = 1;                                     //重新允许中断
}
```

16.4.5 口令擦除写入-多扇区备份-串口 1 操作

C 语言代码（main.c）

```
//测试工作频率为 11.0592MHz

/*      本程序经过测试完全正常，不提供电话技术支持，如不能理解，请自行补充相关基础。  */

/************* 本程序功能说明      **************
STC8G 系列 STC8H 系列 STC8C 系列 EEPROM 通用测试程序，演示多扇区备份、有扇区错误则用正确扇区数据写入、
全部扇区错误(比如第一次运行程序)则写入默认值.

每次写都写入 3 个扇区，即冗余备份.
每个扇区写一条记录，写入完成后读出保存的数据和校验值跟源数据和校验值比较，并从串口 1(P3.0  P3.1)返回结果(正确
或错误提示).
每条记录自校验, 64 字节数据, 2 字节校验值，校验值 = 64 字节数据累加和 ^ 0x5555. ^0x5555 是为了保证写入的 66 个数
据不全部为 0.
如果有扇区错误，则将正确扇区的数据写入错误扇区，如果 3 个扇区都错误，则均写入默认值.
```

```
擦除、写入、读出操作前均需要设置口令, 如果口令不对则退出操作, 并且每次退出操作都会清除口令.

请先别修改程序, 直接下载"03-口令擦除写入-多扇区备份-串口1 操作"里面的"UART-EEPROM.hex"测试. 下载时选择主频 11.0592MHZ.

PC 串口设置: 波特率 115200,8,n,1.
对 EEPROM 做扇区擦除、写入 64 字节、读出 64 字节的操作。

命令例子:
使用串口助手发单个字符, 大小写均可.

发 E 或 e:  对 EEPROM 进行扇区擦除操作，E 表示擦除, 会擦除扇区 0、1、2.

发 W 或 w:  对 EEPROM 进行写入操作，W 表示写入, 会写入扇区 0、1、2, 每个扇区连续写 64 字节, 扇区 0 写入 0x0000~0x003f, 扇区 1 写入 0x0200~0x023f, 扇区 0 写入 0x0400~0x043f.

发 R 或 r:  对 EEPROM 进行读出操作，R 表示读出, 会读出扇区 0、1、2, 每个扇区连续读 64 字节, 扇区 0 读出 0x0000~0x003f, 扇区 1 读出 0x0200~0x023f, 扇区 0 读出 0x0400~0x043f.

注意：为了通用，程序不识别扇区是否有效，用户自己根据具体的型号来决定。

日期: 2021-11-5
*****************************************/

#include   "config.H"
#include   "EEPROM.h"

#define          Baudrate1          115200L

/************* 本地常量声明  **************/
u8   code  T_StringD[]={"去年今日此门中，人面桃花相映红。人面不知何处去，桃花依旧笑春风。"};
u8   code  T_StringW[]={"横看成岭侧成峰，远近高低各不同。不识庐山真面目，只缘身在此山中。"};


/************* 本地变量声明  **************/
u8   xdatatmp[70];                              //通用数据
u8   xdataSaveTmp[70];                          //要写入的数组

bit  B_TX1_Busy;
u8   cmd;                                       //串口单字符命令


/************* 本地函数声明  **************/
void  UART1_config(void);
void  TX1_write2buff(u8 dat);                   //写入发送缓冲
void  PrintString1(u8 *puts);                   //发送一个字符串


/*************  外部函数和变量声明 *****************/

/*************  读取 EEPROM 记录, 并且校验, 返回校验结果, 0 为正确, 1 为错误 *****************/
u8   ReadRecord(u16 addr)
{
    u8   i;
    u16  ChckSum;                               //计算的累加和
    u16  j;                                     //读取的累加和

    for(i=0; i<66; i++)     tmp[i] = 0;         //清除缓冲
    PassWord = D_PASSWORD;                      //给定口令
```

```
    EEPROM_read_n(addr,tmp,66);                 //读出扇区 0
    for(ChckSum=0, i=0; i<64; i++)
        ChckSum += tmp[i];                      //计算累加和
    j = ((u16)tmp[64]<<8) + (u16)tmp[65];       //读取记录的累加和
    j ^= 0x5555;                                //隔位取反，避免全 0
    if(ChckSum != j)        return 1;           //累加和错误，返回 1
    return      0;                              //累加和正确，返回 0
}

/*************  写入 EEPROM 记录，并且校验，返回校验结果，0 为正确，1 为错误  *****************/
u8  SaveRecord(u16 addr)
{
    u8  i;
    u16 ChckSum;                                //计算的累加和

    for(ChckSum=0, i=0; i<64; i++)
        ChckSum += SaveTmp[i];                  //计算累加和
    ChckSum ^= 0x5555;                          //隔位取反，避免全 0
    SaveTmp[64] = (u8)(ChckSum >> 8);
    SaveTmp[65] = (u8)ChckSum;

    PassWord = D_PASSWORD;                      //给定口令
    EEPROM_SectorErase(addr);                   //擦除一个扇区
    PassWord = D_PASSWORD;                      //给定口令
    EEPROM_write_n(addr, SaveTmp, 66);          //写入扇区

    for(i=0; i<66; i++)
        tmp[i] = 0;                             //清除缓冲
    PassWord = D_PASSWORD;                      //给定口令
    EEPROM_read_n(addr,tmp,66);                 //读出扇区 0
    for(i=0; i<66; i++)                         //数据比较
    {
        if(SaveTmp[i] != tmp[i])
            return 1;                           //数据有错误，返回 1
    }
    return 0;                                   //累加和正确，返回 0
}

/**********************  主函数  *************************/
void main(void)
{
    u8  i;
    u8  status;                                 //状态

    UART1_config();                             //  选择波特率，2：使用 Timer2 做波特率，
                                                //其它值：使用 Timer1 做波特率.
    EA = 1;                                     //允许总中断

    PrintString1("STC8G-8H-8C 系列 MCU 用串口 1 测试 EEPROM 程序!\r\n");     //UART1 发送一个字符串

                                                //上电读取 3 个扇区并校验，如果有扇区错误则将正确的
                                                //扇区写入错误扇区，如果 3 个扇区都错误，则写入默认值
    status = 0;
    if(ReadRecord(0x0000) == 0)                 //读扇区 0
    {
        status |= 0x01;                         //正确则标记 status.0=1
        for(i=0; i<64; i++)
            SaveTmp[i] = tmp[i];                //保存在写缓冲
    }
```

```
    if(ReadRecord(0x0200) == 0)                 //读扇区 1
    {
        status |= 0x02;                         //正确则标记 status.1=1
        for(i=0; i<64; i++)
            SaveTmp[i] = tmp[i];                //保存在写缓冲
    }
    if(ReadRecord(0x0400) == 0)                 //读扇区 2
    {
        status |= 0x04;                         //正确则标记 status.2=1
        for(i=0; i<64; i++)
            SaveTmp[i] = tmp[i];                //保存在写缓冲
    }

    if(status == 0)                             //所有扇区都错误, 则写入默认值
    {
        for(i=0; i<64; i++)
            SaveTmp[i] = T_StringD[i];          //读取默认值
    }
    else PrintString1("上电读取 3 个扇区数据均正确!\r\n");    //UART1 发送一个字符串提示

    if((status & 0x01) == 0)                    //扇区 0 错误, 则写入默认值
    {
        if(SaveRecord(0x0000) == 0)
            PrintString1("写入扇区 0 正确!\r\n");       //写入记录 0 扇区正确
        else
            PrintString1("写入扇区 0 错误!\r\n");       //写入记录 0 扇区错误
    }
    if((status & 0x02) == 0)                    //扇区 1 错误, 则写入默认值
    {
        if(SaveRecord(0x0200) == 0)
            PrintString1("写入扇区 1 正确!\r\n");       //写入记录 1 扇区正确
        else
            PrintString1("写入扇区 1 错误!\r\n");       //写入记录 1 扇区错误
    }
    if((status & 0x04) == 0)                    //扇区 2 错误, 则写入默认值
    {
        if(SaveRecord(0x0400) == 0)
            PrintString1("写入扇区 2 正确!\r\n");       //写入记录 2 扇区正确
        else
            PrintString1("写入扇区 2 错误!\r\n");       //写入记录 2 扇区错误
    }


    while(1)
    {
        if(cmd != 0)                            //有串口命令
        {
            if((cmd >= 'a') && (cmd <= 'z'))
                cmd -= 0x20;                    //小写转大写

            if(cmd == 'E')                      //PC 请求擦除一个扇区
            {
                PassWord = D_PASSWORD;          //给定口令
                EEPROM_SectorErase(0x0000);     //擦除一个扇区
                PassWord = D_PASSWORD;          //给定口令
                EEPROM_SectorErase(0x0200);     //擦除一个扇区
                PassWord = D_PASSWORD;          //给定口令
                EEPROM_SectorErase(0x0400);     //擦除一个扇区
                PrintString1("扇区擦除完成!\r\n");
```

```
            }

            else if(cmd == 'W')                         //PC 请求写入 EEPROM 64 字节数据
            {
                for(i=0; i<64; i++)
                    SaveTmp[i] = T_StringW[i];          //写入数值
                if(SaveRecord(0x0000) == 0)
                    PrintString1("写入扇区 0 正确!\r\n");   //写入记录 0 扇区正确
                else
                    PrintString1("写入扇区 0 错误!\r\n");   //写入记录 0 扇区错误
                if(SaveRecord(0x0200) == 0)
                    PrintString1("写入扇区 1 正确!\r\n");   //写入记录 1 扇区正确
                else
                    PrintString1("写入扇区 1 错误!\r\n");   //写入记录 1 扇区错误
                if(SaveRecord(0x0400) == 0)
                    PrintString1("写入扇区 2 正确!\r\n");   //写入记录 2 扇区正确
                else
                    PrintString1("写入扇区 2 错误!\r\n");   //写入记录 2 扇区错误
            }

            else if(cmd == 'R')                         //PC 请求返回 64 字节 EEPROM 数据
            {
                if(ReadRecord(0x0000) == 0)             //读出扇区 0 的数据
                {
                    PrintString1("读出扇区 0 的数据如下：\r\n");
                    for(i=0; i<64; i++)
                        TX1_write2buff(tmp[i]);         //将数据返回给串口
                    TX1_write2buff(0x0d);               //回车换行
                    TX1_write2buff(0x0a);
                }
                else   PrintString1("读出扇区 0 的数据错误!\r\n");

                if(ReadRecord(0x0200) == 0)             //读出扇区 1 的数据
                {
                    PrintString1("读出扇区 1 的数据如下：\r\n");
                    for(i=0; i<64; i++)
                        TX1_write2buff(tmp[i]);         //将数据返回给串口
                    TX1_write2buff(0x0d);               //回车换行
                    TX1_write2buff(0x0a);
                }
                else   PrintString1("读出扇区 1 的数据错误!\r\n");

                if(ReadRecord(0x0400) == 0)             //读出扇区 2 的数据
                {
                    PrintString1("读出扇区 2 的数据如下：\r\n");
                    for(i=0; i<64; i++)
                        TX1_write2buff(tmp[i]);         //将数据返回给串口
                    TX1_write2buff(0x0d);               //回车换行
                    TX1_write2buff(0x0a);
                }
                else   PrintString1("读出扇区 2 的数据错误!\r\n");
            }
            else   PrintString1("命令错误!\r\n");
            cmd = 0;
        }
    }
}
/*********************************************/
```

```
/*************  发送一个字节 ********************************/
void TX1_write2buff(u8 dat)                             //写入发送缓冲
{
    B_TX1_Busy = 1;                                     //标志发送忙
    SBUF = dat;                                         //发送一个字节
    while(B_TX1_Busy);                                  //等待发送完毕
}

//========================================================================
// 函数: void PrintString1(u8 *puts)
// 描述:  串口 1 发送字符串函数。
// 参数: puts:   字符串指针.
// 返回: none.
// 版本: VER1.0
// 日期: 2014-11-28
// 备注:
//========================================================================
void PrintString1(u8 *puts)                             //发送一个字符串
{
    for (; *puts != 0;  puts++)                         //遇到停止符 0 结束
    {
        TX1_write2buff(*puts);
    }
}

//========================================================================
// 函数: void    UART1_config(void)
// 描述: UART1 初始化函数。
// 参数: none.
// 返回: none.
// 版本: VER1.0
// 日期: 2014-11-28
// 备注:
//========================================================================
void  UART1_config(void)
{
    TR1 = 0;
    AUXR &= ~0x01;                          //S1 BRT Use Timer1;
    AUXR |=  (1<<6);                        //Timer1 set as 1T mode
    TMOD &= ~(1<<6);                        //Timer1 set As Timer
    TMOD &= ~0x30;                          //Timer1_16bitAutoReload;
    TH1 = (u8)((65536L-(MAIN_Fosc / 4) / Baudrate1) >> 8);
    TL1 = (u8)(65536L-(MAIN_Fosc / 4) / Baudrate1);
    ET1 = 0;                                // 禁止 Timer1 中断
    INT_CLKO &= ~0x02;                      // Timer1 不输出高速时钟
    TR1  = 1;                               // 运行 Timer1

    S1_USE_P30P31();   P3n_standard(0x03);  //切换到 P3.0 P3.1
    //S1_USE_P36P37();  P3n_standard(0xc0);  //切换到 P3.6 P3.7
    //S1_USE_P16P17();  P1n_standard(0xc0);  //切换到 P1.6 P1.7

    SCON = (SCON & 0x3f) | 0x40;            //UART1 模式,  0x00: 同步移位输出,
                                            //             0x40: 8 位数据,可变波特率,
                                            //             0x80: 9 位数据,固定波特率,
                                            //             0xc0: 9 位数据,可变波特率
//  PS  = 1;                                //高优先级中断
    ES  = 1;                                //允许中断
    REN = 1;                                //允许接收
```

```
        B_TX1_Busy = 0;
}

//========================================================================
// 函数: void UART1_int (void) interrupt UART1_VECTOR
// 描述: UART1 中断函数。
// 参数: nine.
// 返回: none.
// 版本: VER1.0
// 日期: 2014-11-28
// 备注:
//========================================================================
void UART1_int (void) interrupt 4
{
        if(RI)
        {
                RI = 0;
                cmd = SBUF;
        }

        if(TI)
        {
                TI = 0;
                B_TX1_Busy = 0;
        }
}
```

C 语言代码（EEPROM.c）

```
//测试工作频率为 11.0592MHz

//      本程序是 STC 系列的内置 EEPROM 读写程序。

#include    "config.h"
#include    "EEPROM.h"

u32         PassWord;                           //擦除 写入时需要的口令

//========================================================================
// 函数: void      ISP_Disable(void)
// 描述: 禁止访问 ISP/IAP.
// 参数: non.
// 返回: non.
// 版本: V1.0, 2012-10-22
//========================================================================
void  DisableEEPROM(void)
{
        ISP_CONTR = 0;                          //禁止 ISP/IAP 操作
        IAP_TPS     = 0;
        ISP_CMD     = 0;                        //去除 ISP/IAP 命令
        ISP_TRIG    = 0;                        //防止 ISP/IAP 命令误触发
        ISP_ADDRH = 0xff;                       //清 0 地址高字节
        ISP_ADDRL = 0xff;                       //清 0 地址低字节，指向非 EEPROM 区，防止误操作
}

//========================================================================
// 函数: void EEPROM_read_n(u16 EE_address,u8 *DataAddress,u16 number)
// 描述: 从指定 EEPROM 首地址读出 n 个字节放指定的缓冲.
// 参数: EE_address:    读出 EEPROM 的首地址.
```

```
//          DataAddress:  读出数据放缓冲的首地址.
//          number:         读出的字节长度.
// 返回: non.
// 版本: V1.0, 2012-10-22
//========================================================================
void EEPROM_read_n(u16 EE_address,u8 *DataAddress,u16 number)
{
    if(PassWord == D_PASSWORD)                  //口令正确才会操作 EEPROM
    {
        EA = 0;                                 //禁止中断
        ISP_CONTR = ISP_EN;                     //允许 ISP/IAP 操作
        IAP_TPS = (u8)(MAIN_Fosc / 1000000L);   //工作频率设置
        ISP_READ();                             //送字节读命令，命令不需改变时，不需重新送命令
        do
        {
            ISP_ADDRH = EE_address / 256;       //送地址高字节（地址需要改变时才需重新送地址）
            ISP_ADDRL = EE_address % 256;       //送地址低字节
            if(PassWord == D_PASSWORD)          //口令口令正确才触发操作
            {
                ISP_TRIG = 0x5A;                //先送 5AH，再送 A5H 到 ISP/IAP 触发寄存器，
                                                //每次都需要如此
                ISP_TRIG = 0xA5;                //送完 A5H 后，ISP/IAP 命令立即被触发启动
            }                                   //CPU 等待 IAP 完成后，才会继续执行程序。
            _nop_();
            _nop_();
            _nop_();
            *DataAddress = ISP_DATA;            //读出的数据送往
            EE_address++;
            DataAddress++;
        }while(--number);

        DisableEEPROM();
        EA = 1;                                 //重新允许中断
    }
    PassWord = 0;                               //清除口令
}

/********************* 扇区擦除函数 *****************/
//========================================================================
// 函数: void EEPROM_SectorErase(u16 EE_address)
// 描述:  把指定地址的 EEPROM 扇区擦除.
// 参数: EE_address:    要擦除的扇区 EEPROM 的地址.
// 返回: non.
// 版本: V1.0, 2013-5-10
//========================================================================
void EEPROM_SectorErase(u16 EE_address)
{
    if(PassWord == D_PASSWORD)                  //口令正确才会操作 EEPROM
    {
        EA = 0;                                 //禁止中断
                                                //只有扇区擦除，没有字节擦除，512 字节/扇区。
                                                //扇区中任意一个字节地址都是扇区地址。
        ISP_ADDRH = EE_address / 256;           //送扇区地址高字节（地址需要改变时才需重新送地址）
        ISP_ADDRL = EE_address % 256;           //送扇区地址低字节
        ISP_CONTR = ISP_EN;                     //允许 ISP/IAP 操作
        IAP_TPS = (u8)(MAIN_Fosc / 1000000L);   //工作频率设置
        ISP_ERASE();                            //送扇区擦除命令，命令不需改变时，不需重新送命令
            if(PassWord == D_PASSWORD)          //口令口令正确才触发操作
            {
```

```
            ISP_TRIG = 0x5A;                    //先送 5AH，再送 A5H 到 ISP/IAP 触发寄存器，
                                                //每次都需要如此
            ISP_TRIG = 0xA5;                    //送完 A5H 后，ISP/IAP 命令立即被触发启动
        }                                       //CPU 等待 IAP 完成后，才会继续执行程序。
        _nop_();
        _nop_();
        _nop_();
        DisableEEPROM();
        EA = 1;                                 //重新允许中断
    }
    PassWord = 0;                               //清除口令
}

//=========================================================================
// 函数: void EEPROM_write_n(u16 EE_address,u8 *DataAddress,u16 number)
// 描述: 把缓冲的 n 个字节写入指定首地址的 EEPROM.
// 参数: EE_address:   写入 EEPROM 的首地址.
//       DataAddress:  写入源数据的缓冲的首地址.
//       number:       写入的字节长度.
// 返回: non.
// 版本: V1.0, 2012-10-22
//=========================================================================
void EEPROM_write_n(u16 EE_address,u8 *DataAddress,u16 number)
{
    if(PassWord == D_PASSWORD)                  //口令正确才会操作 EEPROM
    {
        EA = 0;                                 //禁止中断

        ISP_CONTR = ISP_EN;                     //允许 ISP/IAP 操作
        IAP_TPS = (u8)(MAIN_Fosc / 1000000L);   //工作频率设置
        ISP_WRITE();                            //送字节写命令，命令不需改变时，不需重新送命令
        do
        {
            ISP_ADDRH = EE_address / 256;       //送地址高字节（地址需要改变时才需重新送地址）
            ISP_ADDRL = EE_address % 256;       //送地址低字节
            ISP_DATA  = *DataAddress;           //送数据到 ISP_DATA，只有数据改变时才需重新送
            if(PassWord == D_PASSWORD)          //口令正确才触发操作
            {
                ISP_TRIG = 0x5A;                //先送 5AH，再送 A5H 到 ISP/IAP 触发寄存器，
                                                //每次都需要如此
                ISP_TRIG = 0xA5;                //送完 A5H 后，ISP/IAP 命令立即被触发启动
            }                                   //CPU 等待 IAP 完成后，才会继续执行程序。
            _nop_();
            _nop_();
            _nop_();
            EE_address++;
            DataAddress++;
        }while(--number);

        DisableEEPROM();
        EA = 1;                                 //重新允许中断
    }
    PassWord = 0;                               //清除口令
}
```

17 ADC 模数转换，内部 1.19V 参考信号源

STC8A8K64D4 系列单片机内部集成了一个 12 位高速 A/D 转换器。ADC 的时钟频率为系统频率 2 分频再经过用户设置的分频系数进行再次分频(ADC 的工作时钟频率范围为 SYSclk/2/1 到 SYSclk/2/16)。

STC8A8K64D4 系列的 ADC 最快速度： **800K（每秒进行 80 万次 ADC 转换）**

ADC 转换结果的数据格式有两种：左对齐和右对齐。可方便用户程序进行读取和引用。

注意：ADC 的第 15 通道只能用于检测内部参考信号源，参考信号源值出厂时校准为 1.19V，由于制造误差以及测量误差，导致实际的内部参考信号源相比 1.19V，大约有±1%的误差。如果用户需要知道每一颗芯片的准确内部参考信号源值，可外接精准参考信号源，然后利用 ADC 的第 15 通道进行测量标定。

如果芯片有 ADC 的外部参考电源管脚 ADC_VRef+，则一定不能浮空，必须接外部参考电源或者直接连到 VCC

17.1 ADC 相关的寄存器

符号	描述	地址	位地址与符号								复位值
			B7	B6	B5	B4	B3	B2	B1	B0	
ADC_CONTR	ADC 控制寄存器	BCH	ADC_POWER	ADC_START	ADC_FLAG	ADC_EPWMT	ADC_CHS[3:0]				000x,0000
ADC_RES	ADC 转换结果高位寄存器	BDH									0000,0000
ADC_RESL	ADC 转换结果低位寄存器	BEH									0000,0000
ADCCFG	ADC 配置寄存器	DEH	-	-	RESFMT	-	SPEED[3:0]				xx0x,0000

符号	描述	地址	位地址与符号								复位值
			B7	B6	B5	B4	B3	B2	B1	B0	
ADCTIM	ADC 时序控制寄存器	FEA8H	CSSETUP	CSHOLD[1:0]		SMPDUTY[4:0]					0010,1010
ADCEXCFG	ADC 扩展配置寄存器	FEADH	-	-	ADCETRS[1:0]		-	CVTIMESEL[2:0]			xx00,x000

17.1.1 ADC 控制寄存器（ADC_CONTR），PWM 触发 ADC 控制

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
ADC_CONTR	BCH	ADC_POWER	ADC_START	ADC_FLAG	ADC_EPWMT	ADC_CHS[3:0]			

ADC_POWER：ADC 电源控制位

0：关闭 ADC 电源

1：打开 ADC 电源。

建议进入空闲模式和掉电模式前将 ADC 电源关闭，以降低功耗

特别注意：

1、给 MCU 的 内部 ADC 模块电源打开后，需等待约 1ms, 等 MCU 内部的 ADC 电源稳定后再让 ADC 工作；

2、适当加长对外部信号的采样时间，就是对 ADC 内部采样保持电容的充电或放电时间,时间够，内部才能和外部电势相等。

ADC_START：ADC 转换启动控制位。写入 1 后开始 ADC 转换，转换完成后硬件自动将此位清零。

0：无影响。即使 ADC 已经开始转换工作，写 0 也不会停止 A/D 转换。

1：开始 ADC 转换，转换完成后硬件自动将此位清零。

ADC_FLAG：ADC 转换结束标志位。当 ADC 完成一次转换后，硬件会自动将此位置 1，并向 CPU 提出中断请求。此标志位必须软件清零。

ADC_EPWMT：使能 PWM 实时触发 ADC 功能。详情请参考 PWM 章节

ADC_CHS[3:0]：ADC 模拟通道选择位

（注意：被选择为 ADC 输入通道的 I/O 口，必须设置 PxM0/PxM1 寄存器将 I/O 口模式设置为高阻输入模式。另外如果 MCU 进入掉电模式/时钟停振模式后，仍需要使能 ADC 通道，则需要设置 PxIE 寄存器关闭数字输入通道，以防止外部模拟输入信号忽高忽低而产生额外的功耗）

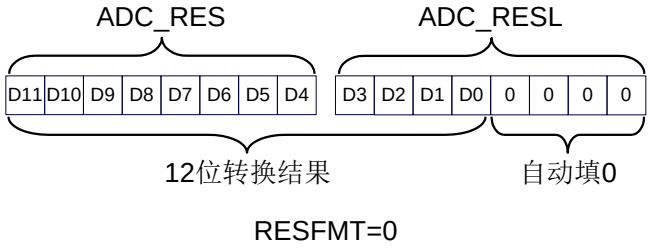
ADC_CHS[3:0]	ADC 通道
0000	P1.0/ADC0
0001	P1.1/ADC1
0010	P1.2/ADC2
0011	P1.3/ADC3
0100	P1.4/ADC4
0101	P1.5/ADC5
0110	P1.6/ADC6
0111	P1.7/ADC7
1000	P0.0/ADC8
1001	P0.1/ADC9
1010	P0.2/ADC10
1011	P0.3/ADC11
1100	P0.4/ADC12
1101	P0.5/ADC13
1110	P0.6/ADC14
1111	测试内部 1.19V

17.1.2 ADC 配置寄存器（ADCCFG）

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
ADCCFG	DEH	-	-	RESFMT	-	SPEED[3:0]			

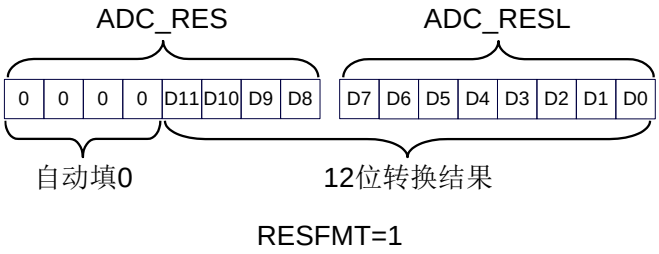
RESFMT：ADC 转换结果格式控制位

0：转换结果左对齐。ADC_RES 保存结果的高 8 位，ADC_RESL 保存结果的低 4 位。格式如下：



RESFMT=0

1：转换结果右对齐。ADC_RES 保存结果的高 4 位，ADC_RESL 保存结果的低 8 位。格式如下：



SPEED[3:0]：设置 ADC 工作时钟频率｛F_{ADC}＝SYSclk/2/(SPEED+1)｝

SPEED[3:0]	给 ADC 的工作时钟频率
0000	SYSclk/2/1
0001	SYSclk/2/2
0010	SYSclk/2/3
...	. . .
1101	SYSclk/2/14
1110	SYSclk/2/15
1111	SYSclk/2/16

17.1.3 ADC 转换结果寄存器（ADC_RES，ADC_RESL）

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
ADC_RES	BDH								
ADC_RESL	BEH								

当 A/D 转换完成后，10 位/12 位的转换结果会自动保存到 ADC_RES 和 ADC_RESL 中。保存结果的数据格式请参考 ADC_CFG 寄存器中的 RESFMT 设置。

17.1.4 ADC 时序控制寄存器（ADCTIM）

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
ADCTIM	FEA8H	CSSETUP	CSHOLD[1:0]		SMPDUTY[4:0]				

CSSETUP：ADC 通道选择时间控制 T_{setup}

CSSETUP	占用 ADC 工作时钟数
0	**1（默认值）**
1	2

CSHOLD[1:0]：ADC 通道选择保持时间控制 T_{hold}

CSHOLD[1:0]	占用 ADC 工作时钟数
00	1
01	**2（默认值）**
10	3
11	4

SMPDUTY[4:0]：ADC 模拟信号采样时间控制 T_{duty}（**注意：SMPDUTY 一定不能设置小于 01010B**）

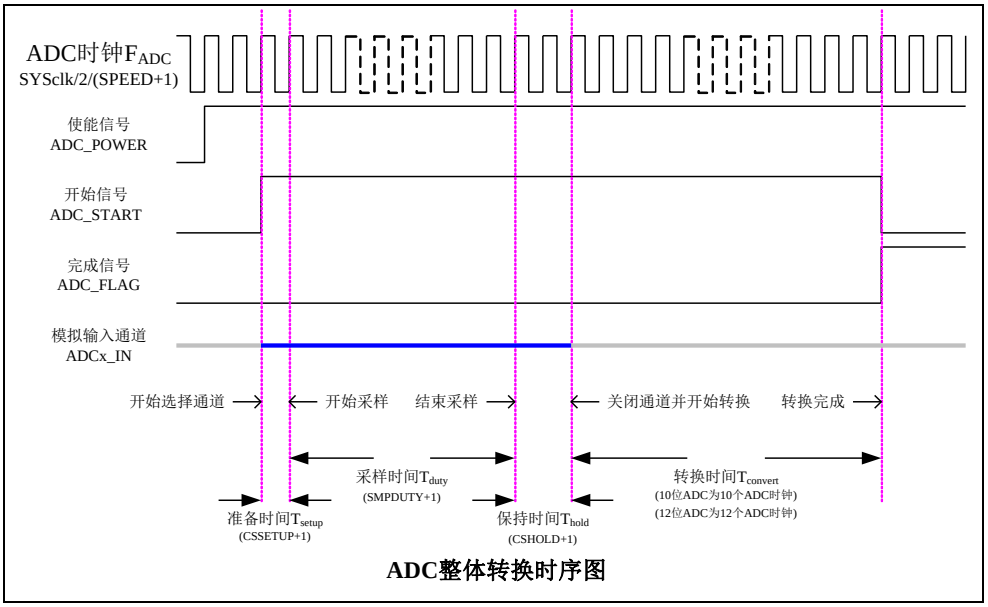
SMPDUTY[4:0]	占用 ADC 工作时钟数
00000	1
00001	2
...	...
01010	**11（默认值）**
...	...
11110	31
11111	32

ADC 数模转换时间：$T_{convert}$

10 位 ADC 的转换时间固定为 10 个 ADC 工作时钟

12 位 ADC 的转换时间固定为 12 个 ADC 工作时钟

一个完整的 ADC 转换时间为： $T_{setup} + T_{duty} + T_{hold} + T_{convert}$，如下图所示



ADC整体转换时序图

17.1.5 ADC 扩展配置寄存器（ADCEXCFG）

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
ADCEXCFG	FEADH	-	-	ADCETRS[1:0]		-	CVTIMESEL[2:0]		

ADCETRS[1:0]：ADC 外部触发脚 ADC_ETR 控制位

ADCETRS[1:0]	ADC_ETR 设置
0x	禁止 ETR 功能
10	使能 ADC_ETR 的上升沿触发 ADC
11	使能 ADC_ETR 的下降沿触发 ADC

注：使用此功能前，必须打开 ADC_CONTR 中的 ADC 电源开关，并设置好相应的 ADC 通道

CVTIMESEL[2:0]：ADC 自动转换次数选择

CVTIMESEL [2:0]	ADC 自动转换次数
0xx	转换 1 次
100	转换 2 次并取平均值
101	转换 4 次并取平均值
110	转换 8 次并取平均值
111	转换 16 次并取平均值

注：当使能 ADC 自动转换多次功能后，ADC 中断标志只会在 ADC 自动转换到设置的次数后，才会被置 1（例如：设置 CVTIMESEL 为 101B，即 ADC 自动转换 4 次并取平均值，则 ADC 中断标志位每完成 4 次 ADC 转换才会被置 1）

17.2 ADC 相关计算公式

17.2.1 ADC 速度计算公式

ADC 的转换速度由 ADCCFG 寄存器中的 SPEED 和 ADCTIM 寄存器共同控制。转换速度的计算公式如下所示：

$$12\text{位ADC转换速度} = \frac{\text{MCU工作频率SYSclk}}{2\times(\text{SPEED[3:0]}+1)\times[(\text{CSSETUP}+1)+(\text{CSHOLD}+1)+(\text{SMPDUTY}+1)+12]}$$

注意：

- 12 位 ADC 的速度不能高于 800KHz
- SMPDUTY 的值不能小于 10，建议设置为 15
- CSSETUP 可使用上电默认值 0
- CHOLD 可使用上电默认值 1（ADCTIM 建议设置为 3FH）

17.2.2 ADC 转换结果计算公式

$$12\text{位ADC转换结果} = 4096 \times \frac{\text{ADC被转换通道的输入电压Vin}}{\text{ADC外部参考源的电压}} \quad \text{（有独立ADC\_Vref+管脚）}$$

17.2.3 反推 ADC 输入电压计算公式

$$\text{ADC被转换通道的输入电压Vin} = \text{ADC外部参考源的电压} \times \frac{12\text{位ADC转换结果}}{4096} \quad \text{（有独立ADC\_Vref+管脚）}$$

17.2.4 反推工作电压计算公式

当需要使用 ADC 输入电压和 ADC 转换结果反推工作电压时，若目标芯片无独立的 ADC_Vref+管脚，则可直接测量并使用下面公式，若目标芯片有独立 ADC_Vref+管脚时，则必须将 ADC_Vref+管脚连接到 Vcc 管脚。

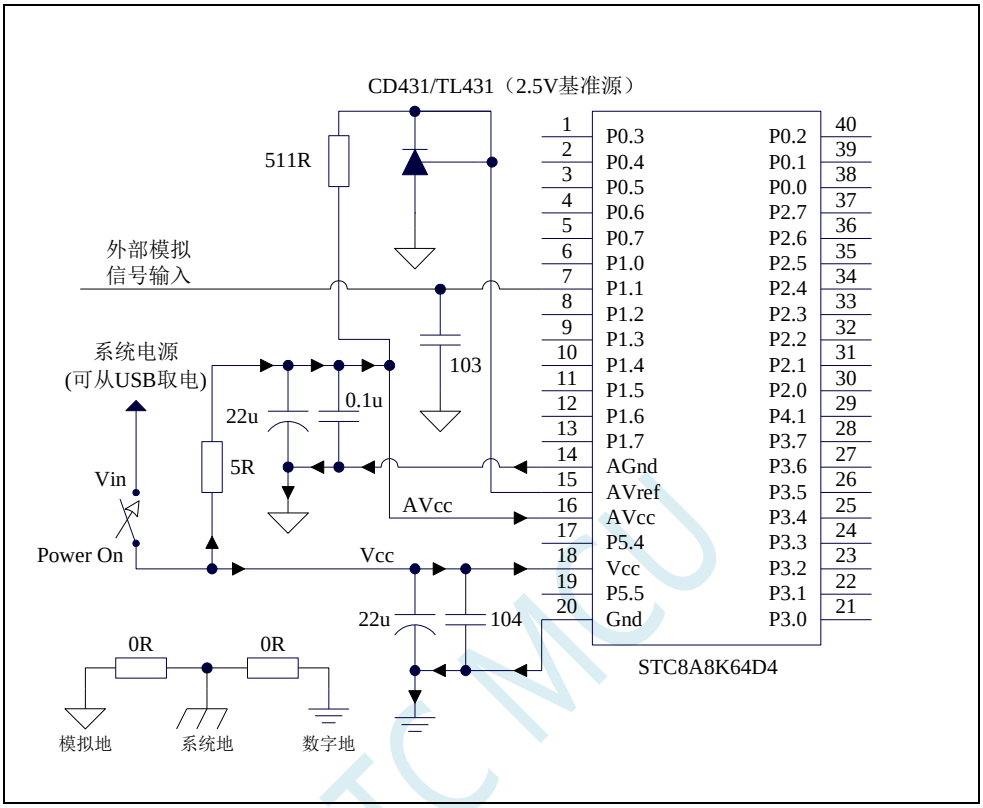
$$\text{MCU工作电压Vcc} = 4096 \times \frac{\text{ADC被转换通道的输入电压Vin}}{\text{12位ADC转换结果}}$$

17.3 12 位 ADC 静态特性

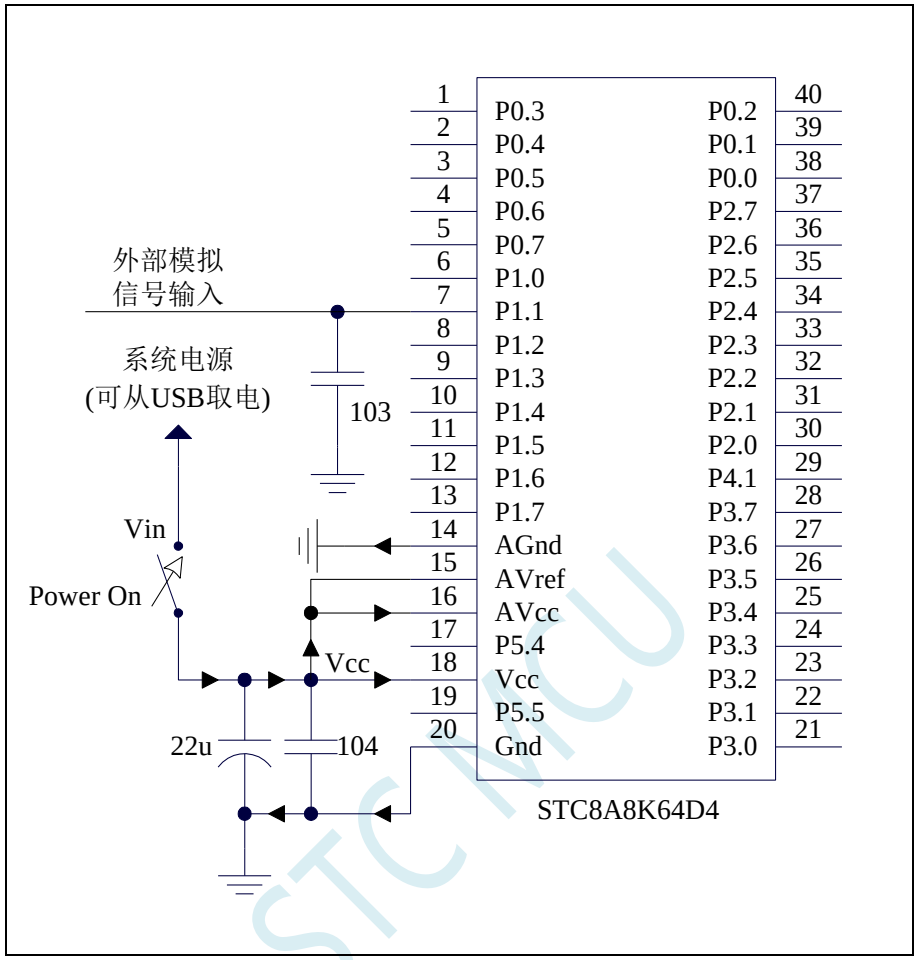
符号	描述	最小值	典型值	最大值	单位
RES	分辨率	-	12	-	Bits
E_T	整体误差	-	0.5	1	LSB
E_O	偏移误差	-	-0.1	1	LSB
E_G	增益误差	-	0	1	LSB
E_D	微分非线性误差	-	0.7	1.5	LSB
E_I	积分非线性误差	-	1	2	LSB
R_{AIN}	通道等效电阻	-	∞	-	ohm
R_{ESD}	采样保持电容前串接的抗静电电阻	-	700	-	ohm
C_{ADC}	内部采样保持电容	-	16.5	-	pF

17.4 ADC 应用参考线路图

17.4.1 高精度 ADC 应用



17.4.2 ADC 一般应用（对 ADC 精度要求不高的应用）



17.5 范例程序

17.5.1 ADC 基本操作（查询方式）

C 语言代码

```
//测试工作频率为 11.0592MHz

#include "reg51.h"
#include "intrins.h"

sfr         ADC_CONTR   =   0xbc;
sfr         ADC_RES     =   0xbd;
sfr         ADC_RESL    =   0xbe;
sfr         ADCCFG      =   0xde;

sfr         P_SW2       =   0xba;
#define     ADCTIM      (*(unsigned char volatile xdata *)0xfea8)

sfr         P1M1        =   0x91;
sfr         P1M0        =   0x92;
sfr         P0M1        =   0x93;
sfr         P0M0        =   0x94;
sfr         P2M1        =   0x95;
sfr         P2M0        =   0x96;
sfr         P3M1        =   0xb1;
sfr         P3M0        =   0xb2;
sfr         P4M1        =   0xb3;
sfr         P4M0        =   0xb4;
sfr         P5M1        =   0xc9;
sfr         P5M0        =   0xca;

void main()
{
    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    P1M0 = 0x00;                                //设置 P1.0 为 ADC 口
    P1M1 = 0x01;
    P_SW2 |= 0x80;
    ADCTIM = 0x3f;                              //设置 ADC 内部时序
    P_SW2 &= 0x7f;
    ADCCFG = 0x0f;                              //设置 ADC 时钟为系统时钟/2/16
    ADC_CONTR = 0x80;                           //使能 ADC 模块

    while (1)
```

```
    {
        ADC_CONTR |= 0x40;                  //启动 AD 转换
        _nop_();
        _nop_();
        while (!(ADC_CONTR & 0x20));        //查询 ADC 完成标志
        ADC_CONTR &= ~0x20;                 //清完成标志
        P2 = ADC_RES;                       //读取 ADC 结果
    }
}
```

汇编代码

```
;测试工作频率为 11.0592MHz

ADC_CONTR   DATA    0BCH
ADC_RES     DATA    0BDH
ADC_RESL    DATA    0BEH
ADCCFG      DATA    0DEH

P_SW2       DATA    0BAH
ADCTIM      XDATA   0FEA8H

P1M1        DATA    091H
P1M0        DATA    092H
P0M1        DATA    093H
P0M0        DATA    094H
P2M1        DATA    095H
P2M0        DATA    096H
P3M1        DATA    0B1H
P3M0        DATA    0B2H
P4M1        DATA    0B3H
P4M0        DATA    0B4H
P5M1        DATA    0C9H
P5M0        DATA    0CAH

            ORG     0000H
            LJMP    MAIN

            ORG     0100H
MAIN:
            MOV     SP, #5FH
            MOV     P0M0, #00H
            MOV     P0M1, #00H
            MOV     P1M0, #00H
            MOV     P1M1, #00H
            MOV     P2M0, #00H
            MOV     P2M1, #00H
            MOV     P3M0, #00H
            MOV     P3M1, #00H
            MOV     P4M0, #00H
            MOV     P4M1, #00H
            MOV     P5M0, #00H
            MOV     P5M1, #00H

            MOV     P1M0,#00H           ;设置 P1.0 为 ADC 口
            MOV     P1M1,#01H
            MOV     P_SW2,#80H
            MOV     DPTR,#ADCTIM        ;设置 ADC 内部时序
            MOV     A,#3FH
```

```
        MOVX    @DPTR,A
        MOV     P_SW2,#00H
        MOV     ADCCFG,#0FH         ;设置 ADC 时钟为系统时钟/2/16
        MOV     ADC_CONTR,#80H      ;使能 ADC 模块
LOOP:
        ORL     ADC_CONTR,#40H      ;启动 AD 转换
        NOP
        NOP
        MOV     A,ADC_CONTR         ;查询 ADC 完成标志
        JNB     ACC.5,$-2
        ANL     ADC_CONTR,#NOT 20H  ;清完成标志
        MOV     P2,ADC_RES          ;读取 ADC 结果

        SJMP    LOOP

        END
```

17.5.2 ADC 基本操作（中断方式）

C 语言代码

```
//测试工作频率为 11.0592MHz

#include "reg51.h"
#include "intrins.h"

sfr     ADC_CONTR   =   0xbc;
sfr     ADC_RES     =   0xbd;
sfr     ADC_RESL    =   0xbe;
sfr     ADCCFG      =   0xde;

sfr     P_SW2       =   0xba;
#define ADCTIM      (*(unsigned char volatile xdata *)0xfea8)

sbit    EADC        =   IE^5;

sfr     P1M1        =   0x91;
sfr     P1M0        =   0x92;
sfr     P0M1        =   0x93;
sfr     P0M0        =   0x94;
sfr     P2M1        =   0x95;
sfr     P2M0        =   0x96;
sfr     P3M1        =   0xb1;
sfr     P3M0        =   0xb2;
sfr     P4M1        =   0xb3;
sfr     P4M0        =   0xb4;
sfr     P5M1        =   0xc9;
sfr     P5M0        =   0xca;

void ADC_Isr() interrupt 5
{
    ADC_CONTR &= ~0x20;                 //清中断标志
    P2 = ADC_RES;                       //读取 ADC 结果
    ADC_CONTR |= 0x40;                  //继续 AD 转换
}

void main()
```

```
{
    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    P1M0 = 0x00;                    //设置 P1.0 为 ADC 口
    P1M1 = 0x01;
    P_SW2 |= 0x80;
    ADCTIM = 0x3f;                  //设置 ADC 内部时序
    P_SW2 &= 0x7f;
    ADCCFG = 0x0f;                  //设置 ADC 时钟为系统时钟/2/16
    ADC_CONTR = 0x80;               //使能 ADC 模块
    EADC = 1;                       //使能 ADC 中断
    EA = 1;
    ADC_CONTR |= 0x40;              //启动 AD 转换

    while (1);
}
```

汇编代码

```
;测试工作频率为 11.0592MHz

ADC_CONTR   DATA    0BCH
ADC_RES     DATA    0BDH
ADC_RESL    DATA    0BEH
ADCCFG      DATA    0DEH

P_SW2       DATA    0BAH
ADCTIM      XDATA   0FEA8H

EADC        BIT     IE.5

P1M1        DATA    091H
P1M0        DATA    092H
P0M1        DATA    093H
P0M0        DATA    094H
P2M1        DATA    095H
P2M0        DATA    096H
P3M1        DATA    0B1H
P3M0        DATA    0B2H
P4M1        DATA    0B3H
P4M0        DATA    0B4H
P5M1        DATA    0C9H
P5M0        DATA    0CAH

            ORG     0000H
            LJMP    MAIN
            ORG     002BH
            LJMP    ADCISR
```

```
            ORG     0100H
ADCISR:
            ANL     ADC_CONTR,#NOT 20H      ;清完成标志
            MOV     P2,ADC_RES              ;读取 ADC 结果
            ORL     ADC_CONTR,#40H          ;继续 AD 转换
            RETI

MAIN:
            MOV     SP, #5FH
            MOV     P0M0, #00H
            MOV     P0M1, #00H
            MOV     P1M0, #00H
            MOV     P1M1, #00H
            MOV     P2M0, #00H
            MOV     P2M1, #00H
            MOV     P3M0, #00H
            MOV     P3M1, #00H
            MOV     P4M0, #00H
            MOV     P4M1, #00H
            MOV     P5M0, #00H
            MOV     P5M1, #00H

            MOV     P1M0,#00H               ;设置 P1.0 为 ADC 口
            MOV     P1M1,#01H
            MOV     P_SW2,#80H
            MOV     DPTR,#ADCTIM            ;设置 ADC 内部时序
            MOV     A,#3FH
            MOVX    @DPTR,A
            MOV     P_SW2,#00H
            MOV     ADCCFG,#0FH             ;设置 ADC 时钟为系统时钟/2/16
            MOV     ADC_CONTR,#80H          ;使能 ADC 模块
            SETB    EADC                    ;使能 ADC 中断
            SETB    EA
            ORL     ADC_CONTR,#40H          ;启动 AD 转换

            SJMP    $

            END
```

17.5.3 格式化 ADC 转换结果

C 语言代码

```
//测试工作频率为 11.0592MHz

#include "reg51.h"
#include "intrins.h"

sfr         ADC_CONTR   =   0xbc;
sfr         ADC_RES     =   0xbd;
sfr         ADC_RESL    =   0xbe;
sfr         ADCCFG      =   0xde;

sfr         P_SW2       =   0xba;
#define     ADCTIM          (*(unsigned char volatile xdata *)0xfea8)
```

```
sfr     P1M1    =   0x91;
sfr     P1M0    =   0x92;
sfr     P0M1    =   0x93;
sfr     P0M0    =   0x94;
sfr     P2M1    =   0x95;
sfr     P2M0    =   0x96;
sfr     P3M1    =   0xb1;
sfr     P3M0    =   0xb2;
sfr     P4M1    =   0xb3;
sfr     P4M0    =   0xb4;
sfr     P5M1    =   0xc9;
sfr     P5M0    =   0xca;

void main()
{
    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    P1M0 = 0x00;                    //设置 P1.0 为 ADC 口
    P1M1 = 0x01;
    P_SW2 |= 0x80;
    ADCTIM = 0x3f;                  //设置 ADC 内部时序
    P_SW2 &= 0x7f;
    ADCCFG = 0x0f;                  //设置 ADC 时钟为系统时钟/2/16
    ADC_CONTR = 0x80;               //使能 ADC 模块
    ADC_CONTR |= 0x40;              //启动 AD 转换
    _nop_();
    _nop_();
    while (!(ADC_CONTR & 0x20));    //查询 ADC 完成标志
    ADC_CONTR &= ~0x20;             //清完成标志

    ADCCFG = 0x00;                  //设置结果左对齐
    ACC = ADC_RES;                  //A 存储 ADC 的 10 位结果的高 8 位
    B = ADC_RESL;                   //B[7:6]存储 ADC 的 10 位结果的低 2 位,B[5:0]为 0

//  ADCCFG = 0x20;                  //设置结果右对齐
//  ACC = ADC_RES;                  //A[1:0]存储 ADC 的 10 位结果的高 2 位,A[7:2]为 0
//  B = ADC_RESL;                   //B 存储 ADC 的 10 位结果的低 8 位

    while (1);
}
```

汇编代码

```
;测试工作频率为 11.0592MHz

ADC_CONTR   DATA    0BCH
ADC_RES     DATA    0BDH
ADC_RESL    DATA    0BEH
```

```
ADCCFG      DATA    0DEH

P_SW2       DATA    0BAH
ADCTIM      XDATA   0FEA8H

P1M1        DATA    091H
P1M0        DATA    092H
P0M1        DATA    093H
P0M0        DATA    094H
P2M1        DATA    095H
P2M0        DATA    096H
P3M1        DATA    0B1H
P3M0        DATA    0B2H
P4M1        DATA    0B3H
P4M0        DATA    0B4H
P5M1        DATA    0C9H
P5M0        DATA    0CAH

            ORG     0000H
            LJMP    MAIN

            ORG     0100H
MAIN:
            MOV     SP, #5FH
            MOV     P0M0, #00H
            MOV     P0M1, #00H
            MOV     P1M0, #00H
            MOV     P1M1, #00H
            MOV     P2M0, #00H
            MOV     P2M1, #00H
            MOV     P3M0, #00H
            MOV     P3M1, #00H
            MOV     P4M0, #00H
            MOV     P4M1, #00H
            MOV     P5M0, #00H
            MOV     P5M1, #00H

            MOV     P1M0,#00H           ;设置 P1.0 为 ADC 口
            MOV     P1M1,#01H
            MOV     P_SW2,#80H
            MOV     DPTR,#ADCTIM        ;设置 ADC 内部时序
            MOV     A,#3FH
            MOVX    @DPTR,A
            MOV     P_SW2,#00H
            MOV     ADCCFG,#0FH         ;设置 ADC 时钟为系统时钟/2/16
            MOV     ADC_CONTR,#80H      ;使能 ADC 模块

            ORL     ADC_CONTR,#40H      ;启动 AD 转换
            NOP
            NOP
            MOV     A,ADC_CONTR         ;查询 ADC 完成标志
            JNB     ACC.5,$-2
            ANL     ADC_CONTR,#NOT 20H  ;清完成标志

            MOV     ADCCFG,#00H         ;设置结果左对齐
            MOV     A,ADC_RES           ;A 存储 ADC 的 10 位结果的高 8 位
            MOV     B,ADC_RESL          ;B[7:6]存储 ADC 的 10 位结果的低 2 位,B[5:0]为 0

;           MOV     ADCCFG,#20H         ;设置结果右对齐
```

```
;               MOV     A,ADC_RES               ;A[3:0]存储 ADC 的 10 位结果的高 2 位,A[7:2]为 0
;               MOV     B,ADC_RESL              ;B 存储 ADC 的 10 位结果的低 8 位

                SJMP    $

                END
```

17.5.4 ADC 自动转换多次取平均值

C 语言代码

```
//测试工作频率为 11.0592MHz

#include "reg51.h"
#include "intrins.h"

sfr         ADC_CONTR   =       0xbc;
sfr         ADC_RES     =       0xbd;
sfr         ADC_RESL    =       0xbe;
sfr         ADCCFG      =       0xde;

sfr         P_SW2       =       0xba;
#define     ADCTIM      (*(unsigned char volatile xdata *)0xfea8)
#define     ADCEXCFG    (*(unsigned char volatile xdata *)0xfead)

sfr         P1M1        =       0x91;
sfr         P1M0        =       0x92;
sfr         P0M1        =       0x93;
sfr         P0M0        =       0x94;
sfr         P2M1        =       0x95;
sfr         P2M0        =       0x96;
sfr         P3M1        =       0xb1;
sfr         P3M0        =       0xb2;
sfr         P4M1        =       0xb3;
sfr         P4M0        =       0xb4;
sfr         P5M1        =       0xc9;
sfr         P5M0        =       0xca;

void main()
{
    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    P1M0 = 0x00;                                //设置 P1.0 为 ADC 口
    P1M1 = 0x01;
    P_SW2 |= 0x80;
    ADCTIM = 0x3f;                              //设置 ADC 内部时序
//  ADCEXCFG = 0x04;                            //设置 ADC 自动转换 2 此取平均值
```

```
//      ADCEXCFG = 0x05;                    //设置 ADC 自动转换 4 此取平均值
        ADCEXCFG = 0x06;                    //设置 ADC 自动转换 8 此取平均值
//      ADCEXCFG = 0x07;                    //设置 ADC 自动转换 16 此取平均值
        P_SW2 &= 0x7f;
        ADCCFG = 0x0f;                      //设置 ADC 时钟为系统时钟/2/16
        ADC_CONTR = 0x80;                   //使能 ADC 模块

        while (1)
        {
            ADC_CONTR |= 0x40;              //启动 AD 转换
            _nop_();
            _nop_();
            while (!(ADC_CONTR & 0x20));    //查询 ADC 完成标志
            ADC_CONTR &= ~0x20;             //清完成标志
            P2 = ADC_RES;                   //读取 ADC 结果
        }
}
```

汇编代码

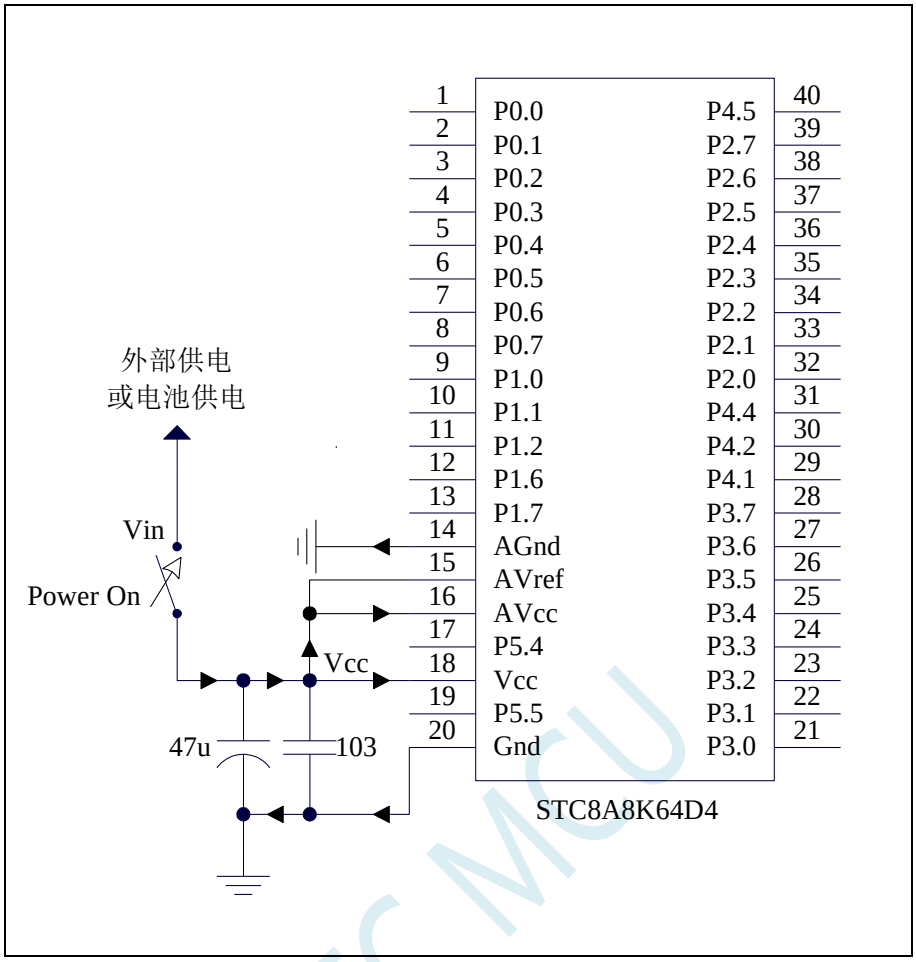
```
;测试工作频率为 11.0592MHz

ADC_CONTR   DATA    0BCH
ADC_RES     DATA    0BDH
ADC_RESL    DATA    0BEH
ADCCFG      DATA    0DEH

P_SW2       DATA    0BAH
ADCTIM      XDATA   0FEA8H
ADCEXCFG    XDATA   0FEAdH

P1M1        DATA    091H
P1M0        DATA    092H
P0M1        DATA    093H
P0M0        DATA    094H
P2M1        DATA    095H
P2M0        DATA    096H
P3M1        DATA    0B1H
P3M0        DATA    0B2H
P4M1        DATA    0B3H
P4M0        DATA    0B4H
P5M1        DATA    0C9H
P5M0        DATA    0CAH

            ORG     0000H
            LJMP    MAIN

            ORG     0100H
MAIN:
            MOV     SP, #5FH
            MOV     P0M0, #00H
            MOV     P0M1, #00H
            MOV     P1M0, #00H
            MOV     P1M1, #00H
            MOV     P2M0, #00H
            MOV     P2M1, #00H
            MOV     P3M0, #00H
            MOV     P3M1, #00H
            MOV     P4M0, #00H
```

```
        MOV     P4M1, #00H
        MOV     P5M0, #00H
        MOV     P5M1, #00H

        MOV     P1M0,#00H           ;设置 P1.0 为 ADC 口
        MOV     P1M1,#01H
        MOV     P_SW2,#80H
        MOV     DPTR,#ADCTIM        ;设置 ADC 内部时序
        MOV     A,#3FH
        MOVX    @DPTR,A
        MOV     DPTR,#ADCEXCFG
;       MOV     A,#04H              ;设置 ADC 自动转换 2 此取平均值
        MOV     A,#05H              ;设置 ADC 自动转换 4 此取平均值
;       MOV     A,#06H              ;设置 ADC 自动转换 8 此取平均值
;       MOV     A,#07H              ;设置 ADC 自动转换 16 此取平均值
        MOVX    @DPTR,A
        MOV     P_SW2,#00H
        MOV     ADCCFG,#0FH         ;设置 ADC 时钟为系统时钟/2/16
        MOV     ADC_CONTR,#80H      ;使能 ADC 模块
LOOP:
        ORL     ADC_CONTR,#40H      ;启动 AD 转换
        NOP
        NOP
        MOV     A,ADC_CONTR         ;查询 ADC 完成标志
        JNB     ACC.5,$-2
        ANL     ADC_CONTR,#NOT 20H  ;清完成标志
        MOV     P2,ADC_RES          ;读取 ADC 结果

        SJMP    LOOP

        END
```

17.5.5 利用 ADC 第 15 通道测量外部电压或电池电压

STC8A8K64D4 系列 ADC 的第 15 通道用于测量内部参考信号源，由于内部参考信号源很稳定，约为 1.19V，且不会随芯片的工作电压的改变而变化，所以可以通过测量内部 1.19V 参考信号源，然后通过 ADC 的值便可反推出外部电压或外部电池电压。

下图为参考线路图:



C 语言代码

```
//测试工作频率为 11.0592MHz

#include "reg51.h"
#include "intrins.h"

#define   FOSC        11059200UL
#define   BRT         (65536 - FOSC / 115200 / 4)

sfr       AUXR        =   0x8e;

sfr       ADC_CONTR   =   0xbc;
sfr       ADC_RES     =   0xbd;
sfr       ADC_RESL    =   0xbe;
sfr       ADCCFG      =   0xde;

sfr       P_SW2       =   0xba;
#define   ADCTIM          (*(unsigned char volatile xdata *)0xfea8)

sfr       P1M1        =   0x91;
sfr       P1M0        =   0x92;
sfr       P0M1        =   0x93;
sfr       P0M0        =   0x94;
sfr       P2M1        =   0x95;
```

```
sfr     P2M0    =   0x96;
sfr     P3M1    =   0xb1;
sfr     P3M0    =   0xb2;
sfr     P4M1    =   0xb3;
sfr     P4M0    =   0xb4;
sfr     P5M1    =   0xc9;
sfr     P5M0    =   0xca;

int     *BGV;                           //内部参考信号源值存放在 idata 中
                                        //idata 的 EFH 地址存放高字节
                                        //idata 的 F0H 地址存放低字节
                                        //电压单位为毫伏(mV)
bit     busy;

void UartIsr() interrupt 4
{
    if (TI)
    {
        TI = 0;
        busy = 0;
    }
    if (RI)
    {
        RI = 0;
    }
}

void UartInit()
{
    SCON = 0x50;
    TMOD = 0x00;
    TL1 = BRT;
    TH1 = BRT >> 8;
    TR1 = 1;
    AUXR = 0x40;
    busy = 0;
}

void UartSend(char dat)
{
    while (busy);
    busy = 1;
    SBUF = dat;
}

void ADCInit()
{
    P_SW2 |= 0x80;
    ADCTIM = 0x3f;                      //设置 ADC 内部时序
    P_SW2 &= 0x7f;

    ADCCFG = 0x2f;                      //设置 ADC 时钟为系统时钟/2/16
    ADC_CONTR = 0x8f;                   //使能 ADC 模块,并选择第 15 通道
}

int ADCRead()
{
    int res;
```

```
    ADC_CONTR |= 0x40;                          //启动 AD 转换
    _nop_();
    _nop_();
    while (!(ADC_CONTR & 0x20));                //查询 ADC 完成标志
    ADC_CONTR &= ~0x20;                         //清完成标志
    res = (ADC_RES << 8) | ADC_RESL;            //读取 ADC 结果

    return res;
}

void main()
{
    int res;
    int vcc;
    int i;

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    BGV = (int idata *)0xef;
    ADCInit();                                  //ADC 初始化
    UartInit();                                 //串口初始化

    ES = 1;
    EA = 1;

//  ADCRead();
//  ADCRead();                                  //前两个数据建议丢弃

    res = 0;
    for (i=0; i<8; i++)
    {
        res += ADCRead();                       //读取 8 次数据
    }
    res >>= 3;                                  //取平均值

    vcc = (int)(4096L * *BGV / res);            //(12 位 ADC 算法)计算 VREF 管脚电压,即电池电压
//  vcc = (int)(1024L * *BGV / res);            //(10 位 ADC 算法)计算 VREF 管脚电压,即电池电压
                                                //注意,此电压的单位为毫伏(mV)
    UartSend(vcc >> 8);                         //输出电压值到串口
    UartSend(vcc);

    while (1);
}
```

上面的方法是使用 ADC 的第 15 通道反推外部电池电压的。在 ADC 测量范围内，ADC 的外部测量电压与 ADC 的测量值是成正比例的，所以也可以使用 ADC 的第 15 通道反推外部通道输入电压，假设当前已获取了内部参考信号源电压为 BGV，内部参考信号源的 ADC 测量值为 res_{bg}，外部通道输入电压

的 ADC 测量值为 res_x，则外部通道输入电压 $V_x = BGV / res_{bg} * res_x$;

17.5.6 ADC 做电容感应触摸按键

按键是电路最常用的零件之一，是人机界面重要的输入方式，我们最熟悉的是机械式按键，但是机械按键有一个缺点（特别是便宜的按键），触点有寿命，很容易出现接触不良而失效。而非接触的按键则没有机械触点，寿命长，使用方便。

非接触的按键有多种方案，而电容感应按键则是低成本的方案，多年前一般是使用专门的 IC 来实现，随着 MCU 功能的加强，以及广大用户的实践经验，直接使用 MCU 来做电容感应按键的技术已经成熟，其中最典型最可靠的是使用 ADC 做的方案。

本文档详述使用 STC 带 ADC 的系列 MCU 做的方案，可以使用任何带 ADC 功能的 MCU 来实现。下面前 3 个图是用得最多的方式，原理都一样，本文使用第 2 个图。

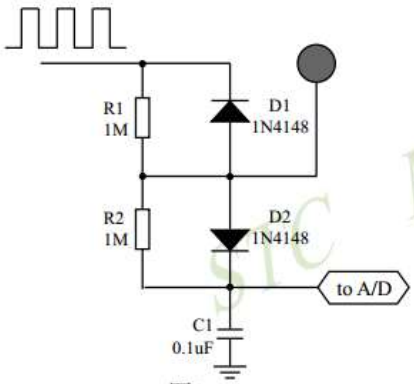


图1

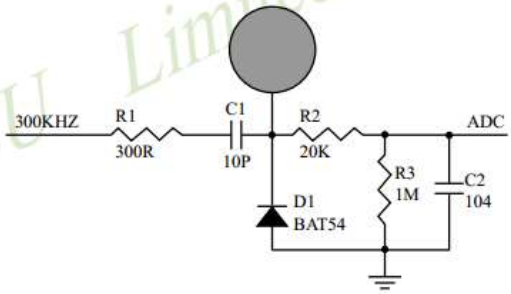


图2

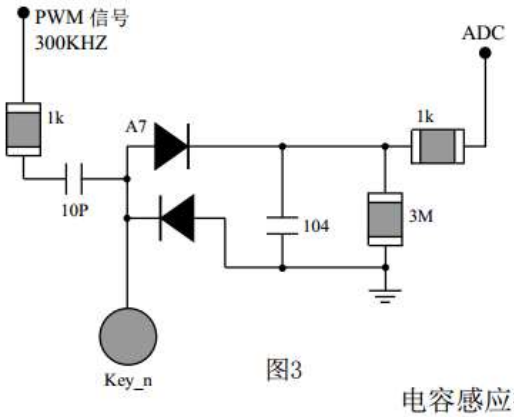


图3

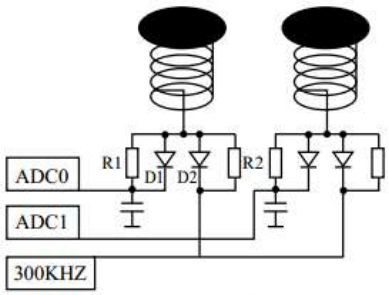
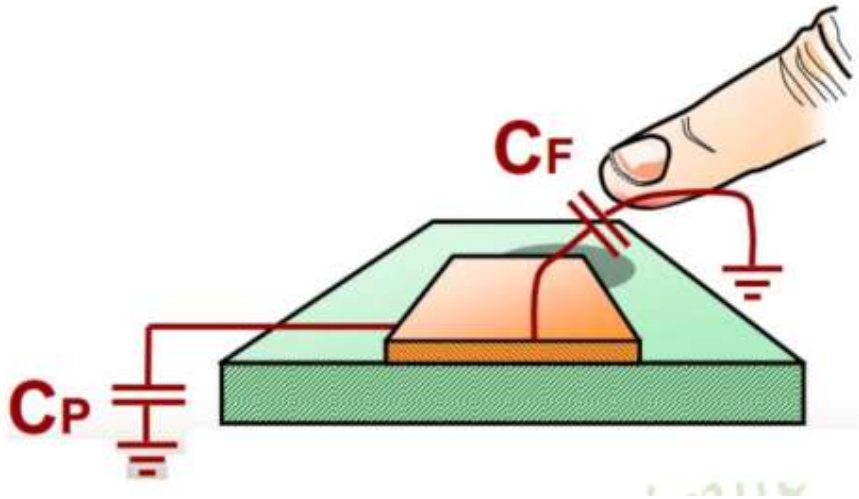


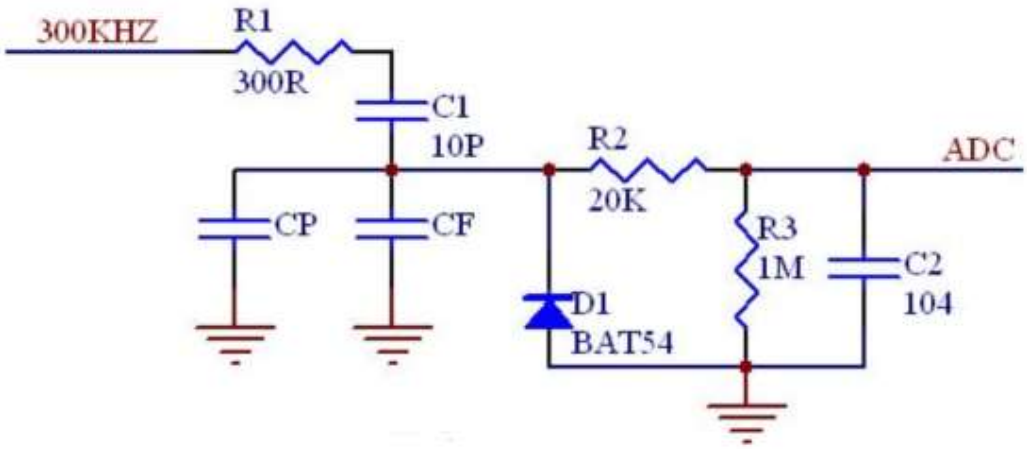
图4 加了感应弹簧

电容感应按键取样电路

一般实际应用时，都使用图 4 所示的感应弹簧来加大手指按下的面积。感应弹簧等效一块对地的金属板，对地有一个电容 CP，而手指按下后，则再并联一个对地的电容 CF，如下图所示。



下面为电路图的说明，CP 为金属板和分布电容，CF 为手指电容，并联在一起与 C1 对输入的 300KHZ 方波进行分压，经过 D1 整流，R2、C2 滤波后送 ADC，当手指压上去后，送去 ADC 的电压降低，程序就可以检测出按键动作。



C 语言代码

```
//测试工作频率为24MHz

#include "reg51.h"
#include "intrins.h"

#define    MAIN_Fosc    24000000UL                  //定义主时钟
#define    Timer0_Reload  (65536UL -(MAIN_Fosc / 600000))   //Timer 0 重装值，对应300KHZ

typedef    unsigned char  u8;
typedef    unsigned int   u16;
typedef    unsigned long  u32;

sfr        P0M1     =   0x93;
sfr        P0M0     =   0x94;
sfr        P1M1     =   0x91;
sfr        P1M0     =   0x92;
sfr        P2M1     =   0x95;
sfr        P2M0     =   0x96;
```

```
sfr     P3M1    =   0xb1;
sfr     P3M0    =   0xb2;
sfr     P4M1    =   0xb3;
sfr     P4M0    =   0xb4;
sfr     P5M1    =   0xc9;
sfr     P5M0    =   0xca;

sfr     ADC_CONTR   =   0xBC;           //带 AD 系列
sfr     ADC_RES     =   0xBD;           //带 AD 系列
sfr     ADC_RESL    =   0xBE;           //带 AD 系列
sfr     AUXR    =   0x8E;
sfr     AUXR2   =   0x8F;

#define CHANNEL     8                   //ADC 通道数
#define ADC_90T     (3<<5)              //ADC 时间 90T
#define ADC_180T    (2<<5)              //ADC 时间 180T
#define ADC_360T    (1<<5)              //ADC 时间 360T
#define ADC_540T    0                   //ADC 时间 540T
#define ADC_FLAG    (1<<4)              //软件清 0
#define ADC_START   (1<<3)              //自动清 0

sbit    P_LED7  =   P2^7;
sbit    P_LED6  =   P2^6;
sbit    P_LED5  =   P2^5;
sbit    P_LED4  =   P2^4;
sbit    P_LED3  =   P2^3;
sbit    P_LED2  =   P2^2;
sbit    P_LED1  =   P2^1;
sbit    P_LED0  =   P2^0;

u16 idata adc[TOUCH_CHANNEL];           //当前 ADC 值
u16 idata adc_prev[TOUCH_CHANNEL];      //上一个 ADC 值
u16 idata TouchZero[TOUCH_CHANNEL];     //0 点 ADC 值
u8 idata TouchZeroCnt[TOUCH_CHANNEL];   //0 点自动跟踪计数
u8 cnt_250ms;

void delay_ms(u8 ms);
void ADC_init(void);
u16 Get_ADC10bitResult(u8 channel);
void AutoZero(void);
u8 check_adc(u8 index);
void ShowLED(void);

void main(void)
{
    u8 i;

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;
```

```
    delay_ms(50);
    ET0 = 0;                                    //初始化 Timer0 输出一个 300KHZ 时钟
    TR0 = 0;
    AUXR |= 0x80;                               //Timer0 set as 1T mode
    AUXR2 |= 0x01;                              //允许输出时钟
    TMOD = 0;                                   //Timer0 set as Timer, 16 bits Auto Reload.
    TH0 = (u8)(Timer0_Reload >> 8);
    TL0 = (u8)Timer0_Reload;
    TR0 = 1;
    ADC_init();                                 //ADC 初始化
    delay_ms(50);                               //延时 50ms
    for (i=0; i<TOUCH_CHANNEL; i++)             //初始化 0 点和上一个值和 0 点自动跟踪计数
    {
        adc_prev[i] = 1023;
        TouchZero[i] = 1023;
        TouchZeroCnt[i] = 0;
    }
    cnt_250ms = 0;
    while (1)
    {
        delay_ms(50);                           //每隔 50ms 处理一次按键
        ShowLED();
        if (++cnt_250ms >= 5)
        {
            cnt_250ms = 0;
            AutoZero();                         //每隔 250ms 处理一次 0 点自动跟踪
        }
    }
}

void delay_ms(u8 ms)
{
    unsigned int i;

    do
    {
        i = MAIN_Fosc / 10000;
        while(--i) ;
    } while(--ms);
}

void ADC_init(void)
{
    P1M0 = 0x00;                                //8 路 ADC
    P1M1 = 0xff;
    ADC_CONTR = 0x80;                           //允许 ADC
}

u16 Get_ADC10bitResult(u8 channel)
{
    ADC_RES = 0;
    ADC_RESL = 0;
    ADC_CONTR = 0x80 | ADC_90T | ADC_START | channel;   //触发 ADC
    _nop_();
    _nop_();
    _nop_();
    _nop_();
    while((ADC_CONTR & ADC_FLAG) == 0) ;        //等待 ADC 转换结束
```

```
    ADC_CONTR = 0x80;                           //清除标志
    return(((u16)ADC_RES << 2) | ((u16)ADC_RESL & 3));  //返回 ADC 结果
}

void AutoZero(void)                             //250ms 调用一次
                                                //这是使用相邻 2 个采样的差的绝对值之和来检测。
{
    u8 i;
    u16 j,k;

    for(i=0; i<TOUCH_CHANNEL; i++)              //处理 8 个通道
    {
        j = adc[i];
        k = j - adc_prev[i];                    //减前一个读数
        F0 = 0;                                 //按下
        if(k & 0x8000) F0 = 1, k = 0 - k;       //释放 求出两次采样的差值
        if(k >= 20)                             //变化比较大
        {
            TouchZeroCnt[i] = 0;                //如果变化比较大，则清 0 计数器
            if(F0) TouchZero[i] = j;            //如果是释放，并且变化比较大，则直接替代
        }
        else                                    //变化比较小，则蠕动，自动 0 点跟踪
        {
            if(++TouchZeroCnt[i] >= 20)         //连续检测到小变化 20 次/4 = 5 秒.
            {
                TouchZeroCnt[i] = 0;
                TouchZero[i] = adc_prev[i];     //变化缓慢的值作为 0 点            }
        }
        adc_prev[i] = j;                        //保存这一次的采样值
    }
}

u8 check_adc(u8 index)                          //获取触摸信息函数 50ms 调用 1 次
                                                //判断键按下或释放,有回差控制
{
    u16 delta;

    adc[index] = 1023 - Get_ADC10bitResult(index);  //获取 ADC 值, 转成按下键 ADC 值增加
    if(adc[index] < TouchZero[index]) return 0;     //比 0 点还小的值，则认为是键释放
    delta = adc[index] - TouchZero[index];
    if(delta >= 40) return 1;                   //键按下
    if(delta <= 20) return 0;                   //键释放
    return 2;                                   //保持原状态
}

void ShowLED(void)
{
    u8 i;

    i = check_adc(0);
    if(i == 0) P_LED0 = 1;                      //指示灯灭
    if(i == 1) P_LED0 = 0;                      //指示灯亮
    i = check_adc(1);
    if(i == 0) P_LED1 = 1;                      //指示灯灭
    if(i == 1) P_LED1 = 0;                      //指示灯亮
    i = check_adc(2);
    if(i == 0) P_LED2 = 1;                      //指示灯灭
    if(i == 1) P_LED2 = 0;                      //指示灯亮
    i = check_adc(3);
```

```
    if(i == 0) P_LED3 = 1;                  //指示灯灭
    if(i == 1) P_LED3 = 0;                  //指示灯亮
    i = check_adc(4);
    if(i == 0) P_LED4 = 1;                  //指示灯灭
    if(i == 1) P_LED4 = 0;                  //指示灯亮
    i = check_adc(5);
    if(i == 0) P_LED5 = 1;                  //指示灯灭
    if(i == 1) P_LED5 = 0;                  //指示灯亮
    i = check_adc(6);
    if(i == 0) P_LED6 = 1;                  //指示灯灭
    if(i == 1) P_LED6 = 0;                  //指示灯亮
    i = check_adc(7);
    if(i == 0) P_LED7 = 1;                  //指示灯灭
    if(i == 1) P_LED7 = 0;                  //指示灯亮
}
```

汇编代码

```
;测试工作频率为 24MHz

Fosc_KHZ    EQU     24000               ;定义主时钟 KHZ
Reload      EQU     (65536 - Fosc_KHZ/600)  ;Timer 0 重装值,对应 300KHZ

ADC_CONTR   DATA    0xBC                ;带 AD 系列
ADC_RES     DATA    0xBD                ;带 AD 系列
ADC_RESL    DATA    0xBE                ;带 AD 系列
AUXR        DATA    0x8E
AUXR2       DATA    0x8F

P0M1        DATA    093H
P0M0        DATA    094H
P1M1        DATA    091H
P1M0        DATA    092H
P2M1        DATA    095H
P2M0        DATA    096H
P3M1        DATA    0B1H
P3M0        DATA    0B2H
P4M1        DATA    0B3H
P4M0        DATA    0B4H
P5M1        DATA    0C9H
P5M0        DATA    0CAH

CHANNEL     EQU     8                   ;ADC 通道数
ADC_90T     EQU     (3 SHL 5)           ;ADC 时间 90T
ADC_180T    EQU     (2 SHL 5)           ;ADC 时间 180T
ADC_360T    EQU     (1 SHL 5)           ;ADC 时间 360T
ADC_540T    EQU     0                   ;ADC 时间 540T
ADC_FLAG    EQU     (1 SHL 4)           ;软件清 0
ADC_START   EQU     (1 SHL 3)           ;自动清 0

P_LED7      BIT     P2.7;
P_LED6      BIT     P2.6;
P_LED5      BIT     P2.5;
P_LED4      BIT     P2.4;
P_LED3      BIT     P2.3;
P_LED2      BIT     P2.2;
P_LED1      BIT     P2.1;
P_LED0      BIT     P2.0;
```

```
adc             EQU     30H             ;当前 ADC 值 30H~3FH,两字节一个值
adc_prev        EQU     40H             ;上一个 ADC 值 40H~4FH,两字节一个值
TouchZero       EQU     50H             ;0 点 ADC 值 50H~5FH,两字节一个值
TouchZeroCnt    EQU     60H             ;0 点自动跟踪计数 60H~67H
cnt_250ms       DATA    68H

                ORG     0000H
                LJMP    MAIN

                ORG     0100H
MAIN:
                MOV     SP,#0D0H
                MOV     P0M0,#00H
                MOV     P0M1,#00H
                MOV     P1M0,#00H
                MOV     P1M1,#00H
                MOV     P2M0,#00H
                MOV     P2M1,#00H
                MOV     P3M0,#00H
                MOV     P3M1,#00H
                MOV     P4M0,#00H
                MOV     P4M1,#00H
                MOV     P5M0,#00H
                MOV     P5M1,#00H

                MOV     R7,#50
                LCALL   F_delay_ms
                CLR     ET0             ;初始化 Timer0 输出一个 300KHZ 时钟
                CLR     TR0
                ORL     AUXR,#080H      ;Timer0 set as 1T mode
                ORL     AUXR2,#01H      ;允许输出时钟
                MOV     TMOD,#0         ;Timer0 set as Timer,16 bits Auto Reload.
                MOV     TH0,#HIGH Reload
                MOV     TL0,#LOW Reload
                SETB    TR0
                LCALL   F_ADC_init
                MOV     R7,#50
                LCALL   F_delay_ms
                MOV     R0,#adc_prev    ;初始化上一个 ADC 值
L_Init_Loop1:
                MOV     @R0,#03H
                INC     R0
                MOV     @R0,#0FFH
                INC     R0
                MOV     A,R0
                CJNE    A,#(adc_prev + CHANNEL * 2),L_Init_Loop1
                MOV     R0,#TouchZero   ;初始化 0 点 ADC 值
L_Init_Loop2:
                MOV     @R0,#03H
                INC     R0
                MOV     @R0,#0FFH
                INC     R0
                MOV     A,R0
                CJNE    A,#(TouchZero+CHANNEL * 2),L_Init_Loop2
                MOV     R0,#TouchZeroCnt ;初始化自动跟踪计数值
L_Init_Loop3:
                MOV     @R0,#0
                INC     R0
                MOV     A,R0
```

```
                CJNE    A,#(TouchZeroCnt + CHANNEL),L_Init_Loop3
                MOV     cnt_250ms,#5
L_MainLoop:
                MOV     R7,#50                  ;延时 50ms
                LCALL   F_delay_ms
                LCALL   F_ShowLED               ;处理一次触摸键值
                DJNZ    cnt_250ms,L_MainLoop
                MOV     cnt_250ms,#5            ;250ms 处理一次 0 点自动跟踪
                LCALL   F_AutoZero              ;自动跟踪零点
                SJMP    L_MainLoop

F_ADC_init:
                MOV     P1M0,#00H               ;8 路 ADC
                MOV     P1M1,#0FFH
                MOV     ADC_CONTR,#080H         ;允许 ADC
                RET

F_Get_ADC10bitResult:
                MOV     ADC_RES,#0
                MOV     ADC_RESL,#0
                MOV     A,R7
                ORL     A,#0E8H                 ;触发 ADC
                MOV     ADC_CONTR,A
                NOP
                NOP
                NOP
                NOP
L_10bitADC_Loop1:
                MOV     A,ADC_CONTR
                JNB     ACC.4,L_10bitADC_Loop1  ;等待 ADC 转换结束
                MOV     ADC_CONTR,#080H         ;清除标志
                MOV     A,ADC_RES
                MOV     B,#04H
                MUL     AB
                MOV     R7,A
                MOV     R6,B
                MOV     A,ADC_RESL
                ANL     A,#03H
                ORL     A,R7
                MOV     R7,A
                RET

F_AutoZero:                                     ;250ms 调用一次
                                                ;这是使用相邻 2 个采样的差的绝对值之和来检测。
                CLR     A
                MOV     R5,A
L_AutoZero_Loop:
                MOV     A,R5
                ADD     A,ACC
                ADD     A,#LOW (adc)
                MOV     R0,A
                MOV     A,@R0
                MOV     R6,A
                INC     R0
                MOV     A,@R0
                MOV     R7,A
                MOV     A,R5
                ADD     A,ACC
                ADD     A,#LOW (adc_prev+01H)
```

```
                MOV     R0,A
                CLR     C
                MOV     A,R7
                SUBB    A,@R0
                MOV     R3,A
                MOV     A,R6
                DEC     R0
                SUBB    A,@R0
                MOV     R2,A
                CLR     F0 ;按下
                JNB     ACC.7,L_AutoZero_1
                SETB    F0
                CLR     C
                CLR     A
                SUBB    A,R3
                MOV     R3,A
                MOV     A,R3
                CLR     A
                SUBB    A,R2
                MOV     R2,A
L_AutoZero_1:
                CLR     C               ;计算 [R2 R3] - #20,if(k >= 20)
                MOV     A,R3
                SUBB    A,#20
                MOV     A,R2
                SUBB    A,#00H
                JC      L_AutoZero_2    ;[R2 R3] ,20,转
                MOV     A,#LOW (TouchZeroCnt)   ;如果变化比较大，则清 0 计数器 TouchZeroCnt[i] = 0;
                ADD     A,R5
                MOV     R0,A
                MOV     @R0,#0
                JNB     F0,L_AutoZero_3
                MOV     A,R5
                ADD     A,ACC
                ADD     A,#LOW (TouchZero)
                MOV     R0,A
                MOV     @R0,6
                INC     R0
                MOV     @R0,7
                SJMP    L_AutoZero_3
L_AutoZero_2:                           ;变化比较小，则蠕动，自动 0 点跟踪
                                        ;连续检测到小变化 20 次/4 = 5 秒.
                MOV     A,#LOW (TouchZeroCnt)
                ADD     A,R5
                MOV     R0,A
                INC     @R0
                MOV     A,@R0
                CLR     C
                SUBB    A,#20
                JC      L_AutoZero_3    ;if(TouchZeroCnt[i] < 20),转
                MOV     @R0,#0          ;TouchZeroCnt[i]      = 0;
                MOV     A,R5            ;变化缓慢的值作为 0 点
                ADD     A,ACC
                ADD     A,#LOW (adc_prev)
                MOV     R0,A
                MOV     A,@R0
                MOV     R2,A
                INC     R0
                MOV     A,@R0
```

```
        MOV     R3,A
        MOV     A,R5
        ADD     A,ACC
        ADD     A,#LOW (TouchZero)
        MOV     R0,A
        MOV     @R0,2
        INC     R0
        MOV     @R0,3
L_AutoZero_3:                           ;保存采样值 adc_prev[i] = j;
        MOV     A,R5
        ADD     A,ACC
        ADD     A,#LOW (adc_prev)
        MOV     R0,A
        MOV     @R0,6
        INC     R0
        MOV     @R0,7
        INC     R5
        MOV     A,R5
        XRL     A,#08H
        JZ      $ + 5H
        LJMP    L_AutoZero_Loop
        RET

F_check_adc:                            ;判断键按下或释放,有回差控制
        MOV R4,7
        LCALL   F_Get_ADC10bitResult    ;返回的 ADC 值在 [R6 R7]
        CLR     C
        MOV     A,#0FFH
        SUBB    A,R7
        MOV     R7,A
        MOV     A,#03H
        SUBB    A,R6
        MOV     R6,A
        MOV     A,R4                    ;保存 adc[index]
        ADD     A,ACC
        ADD     A,#LOW (adc)
        MOV     R0,A
        MOV     @R0,6
        INC     R0
        MOV     @R0,7
        MOV     A,R4
        ADD     A,ACC
        ADD     A,#LOW (TouchZero+01H)
        MOV     R1,A
        MOV     A,R4
        ADD     A,ACC
        ADD     A,#LOW (adc)
        MOV     R0,A
        MOV     A,@R0
        MOV     R6,A
        INC     R0
        MOV     A,@R0
        CLR     C
        SUBB    A,@R1                   ;计算 adc[index] - TouchZero[index]
        MOV     A,R6
        DEC     R1
        SUBB    A,@R1
        JNC     L_check_adc_1
        MOV     R7,#00H
```

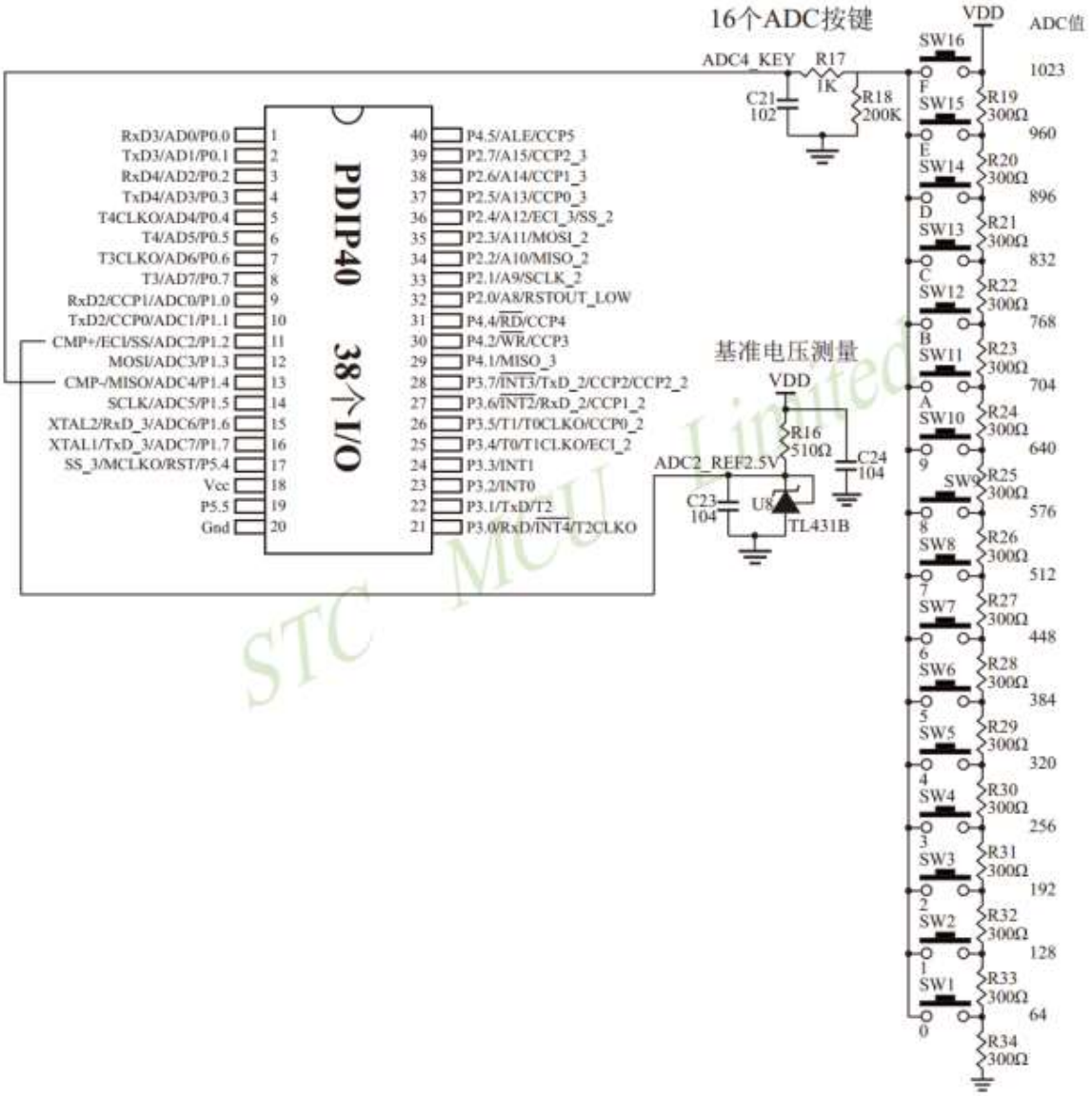
```
                RET
L_check_adc_1:
                MOV     A,R4
                ADD     A,ACC
                ADD     A,#LOW (TouchZero+01H)
                MOV     R1,A
                MOV     A,R4
                ADD     A,ACC
                ADD     A,#LOW (adc+01H)
                MOV     R0,A
                CLR     C
                MOV     A,@R0
                SUBB    A,@R1
                MOV     R7,A
                DEC     R0
                MOV     A,@R0
                DEC     R1
                SUBB    A,@R1
                MOV     R6,A
                CLR     C
                MOV     A,R7
                SUBB    A,#40
                MOV     A,R6
                SUBB    A,#00H
                JC      L_check_adc_2           ;if(delta < 40),转
                MOV     R7,#1                   ;if(delta >= 40) return 1; //键按下 返回 1
                RET
L_check_adc_2:
                SETB    C
                MOV     A,R7
                SUBB    A,#20
                MOV     A,R6
                SUBB    A,#00H
                JNC     L_check_adc_3
                MOV     R7,#0
                RET
L_check_adc_3:
                MOV     R7,#2
                RET

F_ShowLED:
                MOV     R7,#0
                LCALL   F_check_adc
                MOV     A,R7
                ANL     A,#0FEH
                JNZ     L_QuitCheck0
                MOV     A,R7
                MOV     C,ACC.0
                CPL     C
                MOV     P_LED0,C
L_QuitCheck0:
                MOV     R7,#1
                LCALL   F_check_adc
                MOV     A,R7
                ANL     A,#0FEH
                JNZ     L_QuitCheck1
                MOV     A,R7
                MOV     C,ACC.0
                CPL     C
```

```
                MOV     P_LED1,C
L_QuitCheck1:
                MOV     R7,#2
                LCALL   F_check_adc
                MOV     A,R7
                ANL     A,#0FEH
                JNZ     L_QuitCheck2
                MOV     A,R7
                MOV     C,ACC.0
                CPL     C
                MOV     P_LED2,C
L_QuitCheck2:
                MOV     R7,#3
                LCALL   F_check_adc
                MOV     A,R7
                ANL     A,#0FEH
                JNZ     L_QuitCheck3
                MOV     A,R7
                MOV     C,ACC.0
                CPL     C
                MOV     P_LED3,C
L_QuitCheck3:
                MOV     R7,#4
                LCALL   F_check_adc
                MOV     A,R7
                ANL     A,#0FEH
                JNZ     L_QuitCheck4
                MOV     A,R7
                MOV     C,ACC.0
                CPL     C
                MOV     P_LED4,C
L_QuitCheck4:
                MOV     R7,#5
                LCALL   F_check_adc
                MOV     A,R7
                ANL     A,#0FEH
                JNZ     L_QuitCheck5
                MOV     A,R7
                MOV     C,ACC.0
                CPL     C
                MOV     P_LED5,C
L_QuitCheck5:
                MOV     R7,#6
                LCALL   F_check_adc
                MOV     A,R7
                ANL     A,#0FEH
                JNZ     L_QuitCheck6
                MOV     A,R7
                MOV     C,ACC.0
                CPL     C
                MOV     P_LED6,C
L_QuitCheck6:
                MOV     R7,#7
                LCALL   F_check_adc
                MOV     A,R7
                ANL     A,#0FEH
                JNZ     L_QuitCheck7
                MOV     A,R7
                MOV     C,ACC.0
```

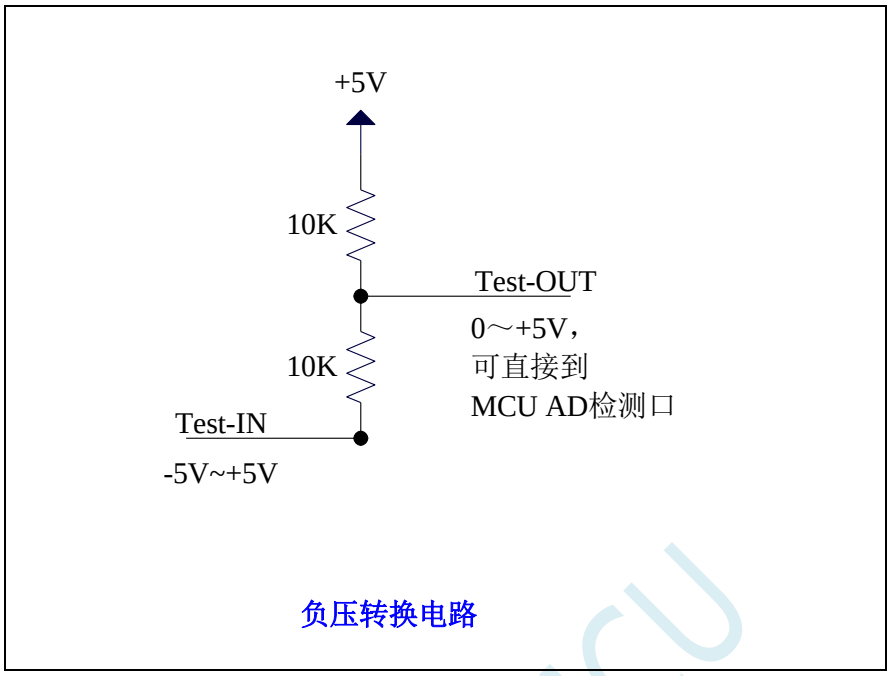
```
            CPL     C
            MOV     P_LED7,C
L_QuitCheck7:
            RET

F_delay_ms:
            PUSH    3
            PUSH    4
L_delay_ms_1:
            MOV     R3,#HIGH (Fosc_KHZ / 13)
            MOV     R4,#LOW (Fosc_KHZ / 13)
L_delay_ms_2:
            MOV     A,R4
            DEC     R4
            JNZ     L_delay_ms_3
            DEC     R3
L_delay_ms_3:
            DEC     A
            ORL     A,R3
            JNZ     L_delay_ms_2
            DJNZ    R7,L_delay_ms_1
            POP     4
            POP     3
            RET

            END
```

17.5.7 ADC 作按键扫描应用线路图

读 ADC 键的方法：每隔 10ms 左右读一次 ADC 值，并且保存最后 3 次的读数，其变化比较小时再判断键。判断键有效时，允许一定的偏差，比如±16 个字的偏差。

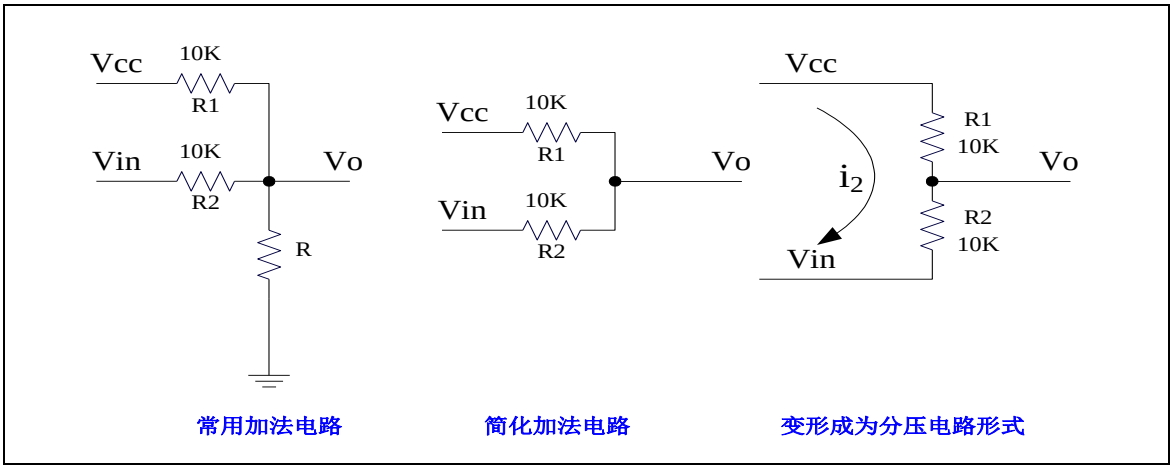

16个ADC按键
ADC4_KEY
R17
1K
C21
102
R18
200K
VDD
ADC值
SW16
1023
SW15
960
SW14
896
SW13
832
SW12
768
SW11
704
SW10
640
SW9
576
SW8
512
SW7
448
SW6
384
SW5
320
SW4
256
SW3
192
SW2
128
SW1
64
R19 300Ω
R20 300Ω
R21 300Ω
R22 300Ω
R23 300Ω
R24 300Ω
R25 300Ω
R26 300Ω
R27 300Ω
R28 300Ω
R29 300Ω
R30 300Ω
R31 300Ω
R32 300Ω
R33 300Ω
R34 300Ω
PDIP40
38个I/O
RxD3/AD0/P0.0
TxD3/AD1/P0.1
RxD4/AD2/P0.2
TxD4/AD3/P0.3
T4CLKO/AD4/P0.4
T4/AD5/P0.5
T3CLKO/AD6/P0.6
T3/AD7/P0.7
RxD2/CCP1/ADC0/P1.0
TxD2/CCP0/ADC1/P1.1
CMP+/ECI/SS/ADC2/P1.2
MOSI/ADC3/P1.3
CMP-/MISO/ADC4/P1.4
SCLK/ADC5/P1.5
XTAL2/RxD_3/ADC6/P1.6
XTAL1/TxD_3/ADC7/P1.7
SS_3/MCLKO/RST/P5.4
Vcc
P5.5
Gnd
P4.5/ALE/CCP5
P2.7/A15/CCP2_3
P2.6/A14/CCP1_3
P2.5/A13/CCP0_3
P2.4/A12/ECI_3/SS_2
P2.3/A11/MOSI_2
P2.2/A10/MISO_2
P2.1/A9/SCLK_2
P2.0/A8/RSTOUT_LOW
P4.4/RD/CCP4
P4.2/WR/CCP3
P4.1/MISO_3
P3.7/INT3/TxD_2/CCP2/CCP2_2
P3.6/INT2/RxD_2/CCP1_2
P3.5/T1/T0CLKO/CCP0_2
P3.4/T0/T1CLKO/ECI_2
P3.3/INT1
P3.2/INT0
P3.1/TxD/T2
P3.0/RxD/INT4/T2CLKO
基准电压测量
VDD
R16
510Ω
C24
104
ADC2_REF2.5V
C23
104
U8
TL431B

17.5.8 检测负电压参考线路图



负压转换电路

17.5.9 常用加法电路在 ADC 中的应用



常用加法电路　　简化加法电路　　变形成为分压电路形式

参照分压电路得到公式 1

公式 1：$Vo = Vin + i_2 * R2$

公式 2：$i_2 = (Vcc - Vin) / (R1 + R2)$｛条件：流向 Vo 的电流 ≈0｝

将 R1=R2 代入公式 2 得公式 3

公式 3：$i_2 = (Vcc - Vin) / 2R2$

将公式 3 代入公式 1 得公式 4

公式 4：$Vo = (Vcc + Vin) / 2$

根据公式 4，可以将以上电路看成加法电路。

在单片机的模数转换测量中，要求被测电压大于 0 并且小于 VCC。如果被测电压小于 0V，可以利用加法电路将被测电压提升到 0V 以上。此时对被测电压的变化范围有一定的要求：

把上述条件代入公式 4 可得到下面 2 式

$(Vcc + Vin) / 2 > 0$　　即 $Vin > -Vcc$

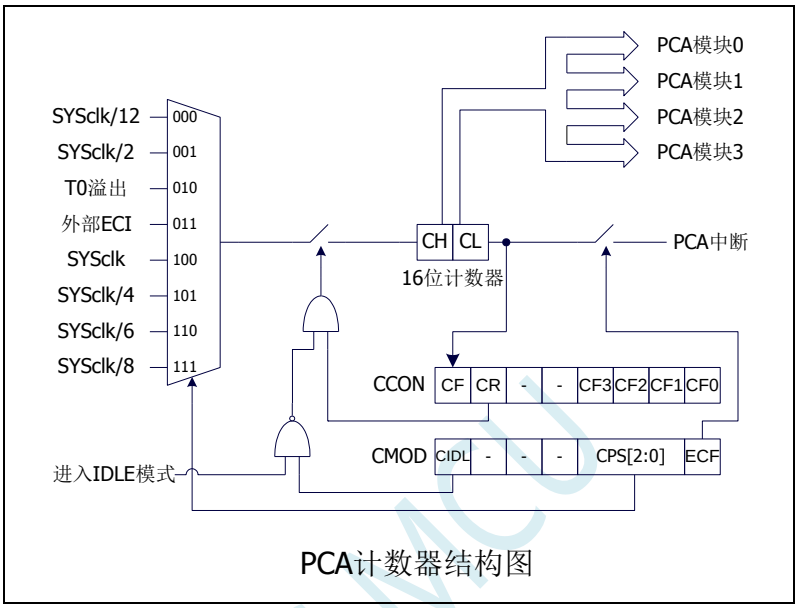
$(Vcc + Vin) / 2 < Vcc$　　即 $Vin < Vcc$

上面 2 式可以合起来：　　**$-Vcc < Vin < Vcc$**

18 PCA/CCP/PWM 应用

STC8A8K64D4 系列单片机内部集成了 4 组可编程计数器阵列（PCA/CCP/PWM）模块，可用于软件定时器、外部脉冲捕获、高速脉冲输出和 PWM 脉宽调制输出。

PCA 内部含有一个特殊的 16 位计数器，4 组 PCA 模块均与之相连接。PCA 计数器的结构图如下：



PCA计数器结构图

18.1 PCA 功能脚切换

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
P_SW1	A2H	S1_S[1:0]		CCP_S[1:0]		SPI_S[1:0]		0	-

CCP_S[1:0]：PCA 功能脚选择位

CCP_S[1:0]	ECI	CCP0	CCP1	CCP2	CCP3
00	P1.2	P1.7	P1.6	P1.5	P1.4
01	P2.2	P2.3	P2.4	P2.5	P2.6
10	P7.4	P7.0	P7.1	P7.2	P7.3
11	P3.5	P3.3	P3.2	P3.1	P3.0

18.2 PCA 相关的寄存器

符号	描述	地址	位地址与符号								复位值
			B7	B6	B5	B4	B3	B2	B1	B0	
CCON	PCA 控制寄存器	D8H	CF	CR	-	-	CCF3	CCF2	CCF1	CCF0	00xx,x000
CMOD	PCA 模式寄存器	D9H	CIDL	-	-	-	CPS[2:0]			ECF	0xxx,0000
CCAPM0	PCA 模块 0 模式控制寄存器	DAH	-	ECOM0	CCAPP0	CCAPN0	MAT0	TOG0	PWM0	ECCF0	x000,0000
CCAPM1	PCA 模块 1 模式控制寄存器	DBH	-	ECOM1	CCAPP1	CCAPN1	MAT1	TOG1	PWM1	ECCF1	x000,0000
CCAPM2	PCA 模块 2 模式控制寄存器	DCH	-	ECOM2	CCAPP2	CCAPN2	MAT2	TOG2	PWM2	ECCF2	x000,0000

CCAPM3	PCA 模块 3 模式控制寄存器	FD54H	-	ECOM3	CCAPP3	CCAPN3	MAT3	TOG3	PWM3	ECCF3	x000,0000
CL	PCA 计数器低字节	E9H									0000,0000
CCAP0L	PCA 模块 0 低字节	EAH									0000,0000
CCAP1L	PCA 模块 1 低字节	EBH									0000,0000
CCAP2L	PCA 模块 2 低字节	ECH									0000,0000
CCAP3L	PCA 模块 3 低字节	FD55H									0000,0000
PCA_PWM0	PCA0 的 PWM 模式寄存器	F2H	EBS0[1:0]		XCCAP0H[1:0]		XCCAP0L[1:0]		EPC0H	EPC0L	0000,0000
PCA_PWM1	PCA1 的 PWM 模式寄存器	F3H	EBS1[1:0]		XCCAP1H[1:0]		XCCAP1L[1:0]		EPC1H	EPC1L	0000,0000
PCA_PWM2	PCA2 的 PWM 模式寄存器	F4H	EBS2[1:0]		XCCAP2H[1:0]		XCCAP2L[1:0]		EPC2H	EPC2L	0000,0000
PCA_PWM3	PCA3 的 PWM 模式寄存器	FD57H	EBS3[1:0]		XCCAP3H[1:0]		XCCAP3L[1:0]		EPC3H	EPC3L	0000,0000
CH	PCA 计数器高字节	F9H									0000,0000
CCAP0H	PCA 模块 0 高字节	FAH									0000,0000
CCAP1H	PCA 模块 1 高字节	FBH									0000,0000
CCAP2H	PCA 模块 2 高字节	FCH									0000,0000
CCAP3H	PCA 模块 3 高字节	FD56H									0000,0000

18.2.1 PCA 控制寄存器（CCON）

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
CCON	D8H	CF	CR	-	-	CCF3	CCF2	CCF1	CCF0

CF：PCA 计数器溢出中断标志。当 PCA 的 16 位计数器计数发生溢出时，硬件自动将此位置 1，并向 CPU 提出中断请求。此标志位需要软件清零。

CR：PCA 计数器允许控制位。

0：停止 PCA 计数

1：启动 PCA 计数

CCFn（n=0,1,2,3）：PCA 模块中断标志。当 PCA 模块发生匹配或者捕获时，硬件自动将此位置 1，并向 CPU 提出中断请求。此标志位需要软件清零。

18.2.2 PCA 模式寄存器（CMOD）

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
CMOD	D9H	CIDL	-	-	-	CPS[2:0]			ECF

CIDL：空闲模式下是否停止 PCA 计数。

0：空闲模式下 PCA 继续计数

1：空闲模式下 PCA 停止计数

CPS[2:0]：PCA 计数脉冲源选择位

CPS[2:0]	PCA 的输入时钟源
000	系统时钟/12
001	系统时钟/2
010	定时器 0 的溢出脉冲
011	ECI 脚的外部输入时钟
100	系统时钟
101	系统时钟/4
110	系统时钟/6

111	系统时钟 8

ECF：PCA 计数器溢出中断允许位。

0：禁止 PCA 计数器溢出中断

1：使能 PCA 计数器溢出中断

18.2.3 PCA 计数器寄存器（CL，CH）

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
CL	E9H								
CH	F9H								

由 CL 和 CH 两个字节组合成一个 16 位计数器，CL 为低 8 位计数器，CH 为高 8 位计数器。每个 PCA 时钟 16 位计数器自动加 1。

18.2.4 PCA 模块模式控制寄存器（CCAPMn）

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
CCAPM0	DAH	-	ECOM0	CCAPP0	CCAPN0	MAT0	TOG0	PWM0	ECCF0
CCAPM1	DBH	-	ECOM1	CCAPP1	CCAPN1	MAT1	TOG1	PWM1	ECCF1
CCAPM2	DCH	-	ECOM2	CCAPP2	CCAPN2	MAT2	TOG2	PWM2	ECCF2
CCAPM3	FD54H	-	ECOM3	CCAPP3	CCAPN3	MAT3	TOG3	PWM3	ECCF3

ECOMn：允许 PCA 模块 n 的比较功能

CCAPPn：允许 PCA 模块 n 进行上升沿捕获

CCAPNn：允许 PCA 模块 n 进行下降沿捕获

MATn：允许 PCA 模块 n 的匹配功能

TOGn：允许 PCA 模块 n 的高速脉冲输出功能

PWMn：允许 PCA 模块 n 的脉宽调制输出功能

ECCFn：允许 PCA 模块 n 的匹配/捕获中断

18.2.5 PCA 模块模式捕获值/比较值寄存器（CCAPnL，CCAPnH）

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
CCAP0L	EAH								
CCAP1L	EBH								
CCAP2L	ECH								
CCAP3L	FD55H								
CCAP0H	FAH								
CCAP1H	FBH								
CCAP2H	FCH								
CCAP3H	FD56H								

当 PCA 模块捕获功能使能时，CCAPnL 和 CCAPnH 用于保存发生捕获时的 PCA 的计数值（CL 和 CH）；当 PCA 模块比较功能使能时，PCA 控制器会将当前 CL 和 CH 中的计数值与保存在 CCAPnL 和 CCAPnH 中的值进行比较，并给出比较结果；当 PCA 模块匹配功能使能时，PCA 控制器会将当前 CL 和 CH 中的计数值与保存在 CCAPnL 和 CCAPnH 中的值进行比较，看是否匹配（相等），并给出匹配结果。

18.2.6 PCA 模块 PWM 模式控制寄存器（PCA_PWMn）

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
PCA_PWM0	F2H	EBS0[1:0]		XCCAP0H[1:0]		XCCAP0L[1:0]		EPC0H	EPC0L
PCA_PWM1	F3H	EBS1[1:0]		XCCAP1H[1:0]		XCCAP1L[1:0]		EPC1H	EPC1L
PCA_PWM2	F4H	EBS2[1:0]		XCCAP2H[1:0]		XCCAP2L[1:0]		EPC2H	EPC2L
PCA_PWM3	FD57H	EBS3[1:0]		XCCAP3H[1:0]		XCCAP3L[1:0]		EPC3H	EPC3L

EBSn[1:0]：PCA 模块 n 的 PWM 位数控制

EBSn[1:0]	PWM 位数	重载值	比较值
00	8 位 PWM	{EPCnH, CCAPnH[7:0]}	{EPCnL, CCAPnL[7:0]}
01	7 位 PWM	{EPCnH, CCAPnH[6:0]}	{EPCnL, CCAPnL[6:0]}
10	6 位 PWM	{EPCnH, CCAPnH[5:0]}	{EPCnL, CCAPnL[5:0]}
11	10 位 PWM	{EPCnH, XCCAPnH[1:0], CCAPnH[7:0]}	{EPCnL, XCCAPnL[1:0], CCAPnL[7:0]}

XCCAPnH[1:0]：10 位 PWM 的第 9 位和第 10 位的重载值

XCCAPnL[1:0]：10 位 PWM 的第 9 位和第 10 位的比较值

EPCnH：PWM 模式下，重载值的最高位（8 位 PWM 的第 9 位，7 位 PWM 的第 8 位，6 位 PWM 的第 7 位，10 位 PWM 的第 11 位）

EPCnL：PWM 模式下，比较值的最高位（8 位 PWM 的第 9 位，7 位 PWM 的第 8 位，6 位 PWM 的第 7 位，10 位 PWM 的第 11 位）

注意：在更新 10 位 PWM 的重载值时，必须先写高两位 XCCAPnH[1:0]，再写低 8 位 CCAPnH[7:0]。

18.3 PCA 工作模式

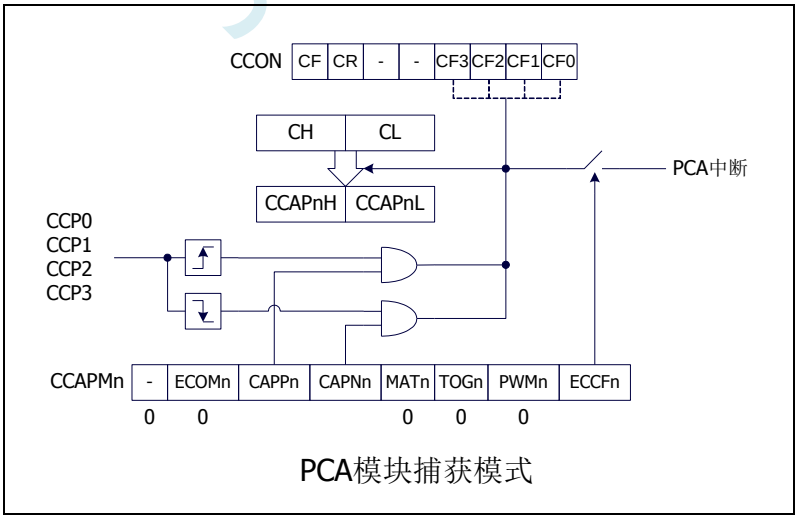
STC12H 系列单片机共有 4 组 PCA 模块，每组模块都可独立设置工作模式。模式设置如下所示：

CCAPMn								模块功能
-	ECOMn	CAPPn	CAPNn	MATn	TOGn	PWMn	ECCFn	
-	0	0	0	0	0	0	0	无操作
-	1	0	0	0	0	1	0	6/7/8/10 位 PWM 模式，无中断
-	1	1	0	0	0	1	1	6/7/8/10 位 PWM 模式，产生上升沿中断
-	1	0	1	0	0	1	1	6/7/8/10 位 PWM 模式，产生下降沿中断
-	1	1	1	0	0	1	1	6/7/8/10 位 PWM 模式，产生边沿中断
-	0	1	0	0	0	0	x	16 位上升沿捕获
-	0	0	1	0	0	0	x	16 位下降沿捕获
-	0	1	1	0	0	0	x	16 位边沿捕获
-	1	0	0	1	0	0	x	16 位软件定时器
-	1	0	0	1	1	0	x	16 位高速脉冲输出

18.3.1 捕获模式

要使一个 PCA 模块工作在捕获模式，寄存器 CCAPMn 中的 CAPNn 和 CAPPn 至少有一位必须置 1（也可两位都置 1）。PCA 模块工作于捕获模式时，对模块的外部 CCP0/CCP1/CCP2 管脚的输入跳变进行采样。当采样到有效跳变时，PCA 控制器立即将 PCA 计数器 CH 和 CL 中的计数值装载到模块的捕获寄存器中 CCAPnL 和 CCAPnH，同时将 CCON 寄存器中相应的 CCFn 置 1。若 CCAPMn 中的 ECCFn 位被设置为 1，将产生中断。由于所有 PCA 模块的中断入口地址是共享的，所以在中断服务程序中需要判断是哪一个模块产生了中断，并注意中断标志位需要软件清零。

PCA 模块工作于捕获模式的结构图如下图所示：



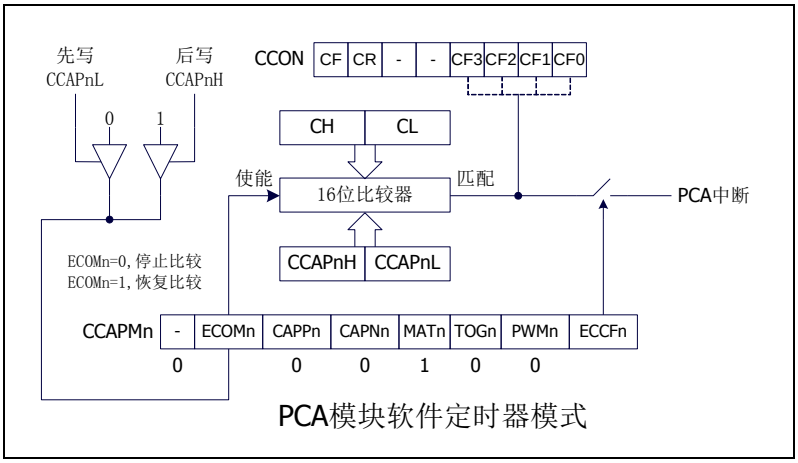
PCA模块捕获模式

18.3.2 软件定时器模式

通过置位 CCAPMn 寄存器的 ECOM 和 MAT 位，可使 PCA 模块用作软件定时器。PCA 计数器值 CL 和 CH 与模块捕获寄存器的值 CCAPnL 和 CCAPnH 相比较，当两者相等时，CCON 中的 CCFn 会被

置 1，若 CCAPMn 中的 ECCFn 被设置为 1 时将产生中断。CCFn 标志位需要软件清零。

PCA 模块工作于软件定时器模式的结构图如下图所示：

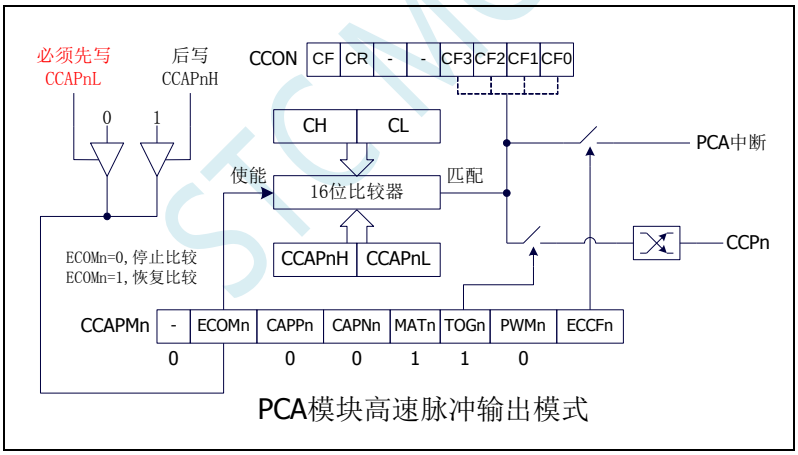


PCA模块软件定时器模式

18.3.3 高速脉冲输出模式

当 PCA 计数器的计数值与模块捕获寄存器的值相匹配时,PCA 模块的 CCPn 输出将发生翻转转。要激活高速脉冲输出模式，CCAPMn 寄存器的 TOGn、MATn 和 ECOMn 位必须都置 1。

PCA 模块工作于高速脉冲输出模式的结构图如下图所示：



PCA模块高速脉冲输出模式

18.3.4 PWM 脉宽调制模式及频率计算公式

18.3.4.1 8 位 PWM 模式

脉宽调制是使用程序来控制波形的占空比、周期、相位波形的一种技术，在三相电机驱动、D/A 转换等场合有广泛的应用。STC8 系列单片机的 PCA 模块可以通过设定各自的 PCA_PWMn 寄存器使其工作于 8 位 PWM 或 7 位 PWM 或 6 位 PWM 或 10 位 PWM 模式。要使能 PCA 模块的 PWM 功能，模块寄存器 CCAPMn 的 PWMn 和 ECOMn 位必须置 1。

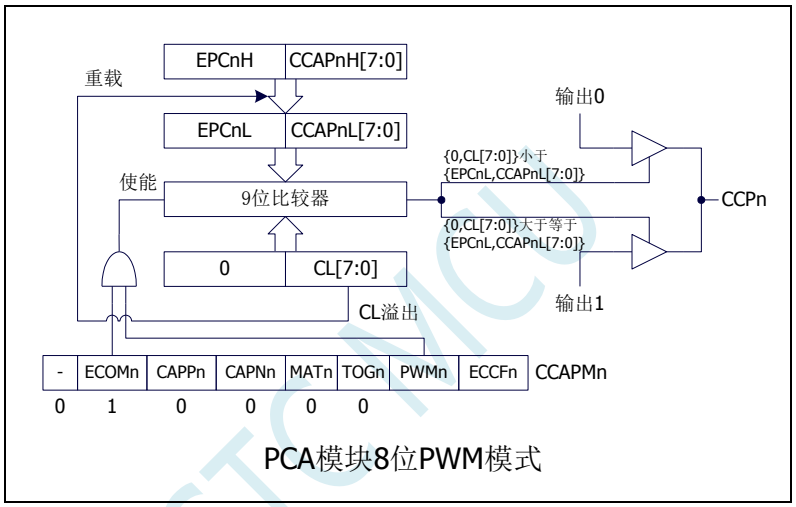
PCA_PWMn 寄存器中的 EBSn[1:0]设置为 00 时，PCA 模块 n 工作于 8 位 PWM 模式，此时将{0,CL[7:0]}与捕获寄存器{EPCnL,CCAPnL[7:0]}进行比较。当 PCA 模块工作于 8 位 PWM 模式时，由于所有模块共

用一个 PCA 计数器，所有它们的输出频率相同。各个模块的输出占空比使用寄存器{EPCnL,CCAPnL[7:0]}进行设置。当{0,CL[7:0]}的值小于{EPCnL,CCAPnL[7:0]}时，输出为低电平；当{0,CL[7:0]}的值等于或大于{EPCnL,CCAPnL[7:0]}时，输出为高电平。当 CL[7:0]的值由 FF 变为 00 溢出时，{EPCnH,CCAPnH[7:0]}的内容重新装载到{EPCnL,CCAPnL[7:0]}中。这样就可实现无干扰地更新 PWM。

$$8\text{位模式的PWM频率} = \frac{\text{PCA时钟输入源频率}}{256}$$

当EPCnH=0及CCAPnH=00H时，PWM固定输出高
当EPCnH=1及CCAPnH=FFH时，PWM固定输出低

PCA 模块工作于 8 位 PWM 模式的结构图如下图所示：



PCA模块8位PWM模式

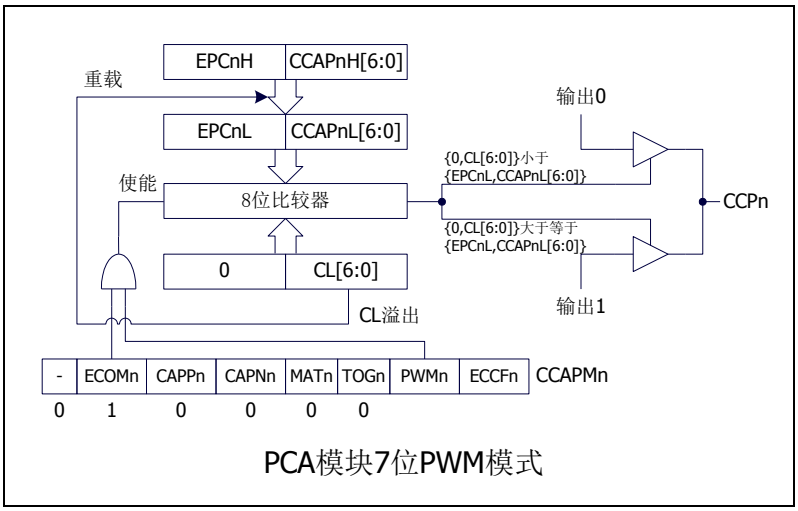
18.3.4.2 7 位 PWM 模式

PCA_PWMn 寄存器中的 EBSn[1:0]设置为 01 时，PCA 模块 n 工作于 7 位 PWM 模式，此时将{0,CL[6:0]}与捕获寄存器{EPCnL,CCAPnL[6:0]}进行比较。当 PCA 模块工作于 7 位 PWM 模式时，由于所有模块共用一个 PCA 计数器，所有它们的输出频率相同。各个模块的输出占空比使用寄存器{EPCnL,CCAPnL[6:0]}进行设置。当{0,CL[6:0]}的值小于{EPCnL,CCAPnL[6:0]}时，输出为低电平；当{0,CL[6:0]}的值等于或大于{EPCnL,CCAPnL[6:0]}时，输出为高电平。当 CL[6:0]的值由 7F 变为 00 溢出时，{EPCnH,CCAPnH[6:0]}的内容重新装载到{EPCnL,CCAPnL[6:0]}中。这样就可实现无干扰地更新 PWM。

$$7\text{位模式的PWM频率} = \frac{\text{PCA时钟输入源频率}}{128}$$

当EPCnH=0及CCAPnH=00H时，PWM固定输出高
当EPCnH=1及CCAPnH=FFH时，PWM固定输出低

PCA 模块工作于 7 位 PWM 模式的结构图如下图所示：



18.3.4.3 6 位 PWM 模式

PCA_PWMn 寄存器中的 EBSn[1:0]设置为 10 时，PCA 模块 n 工作于 6 位 PWM 模式，此时将{0,CL[5:0]}与捕获寄存器{EPCnL,CCAPnL[5:0]}进行比较。当 PCA 模块工作于 6 位 PWM 模式时，由于所有模块共用一个 PCA 计数器，所有它们的输出频率相同。各个模块的输出占空比使用寄存器{EPCnL,CCAPnL[5:0]}进行设置。当{0,CL[5:0]}的值小于{EPCnL,CCAPnL[5:0]}时，输出为低电平；当{0,CL[5:0]}的值等于或大于{EPCnL,CCAPnL[5:0]}时，输出为高电平。当 CL[5:0]的值由 3F 变为 00 溢出时，{EPCnH,CCAPnH[5:0]}的内容重新装载到{EPCnL,CCAPnL[5:0]}中。这样就可实现无干扰地更新 PWM。

$$8\text{位模式的PWM频率} = \frac{\text{PCA时钟输入源频率}}{64}$$

当EPCnH=0及CCAPnH=00H时，PWM固定输出高
当EPCnH=1及CCAPnH=FFH时，PWM固定输出低

PCA 模块工作于 6 位 PWM 模式的结构图如下图所示：