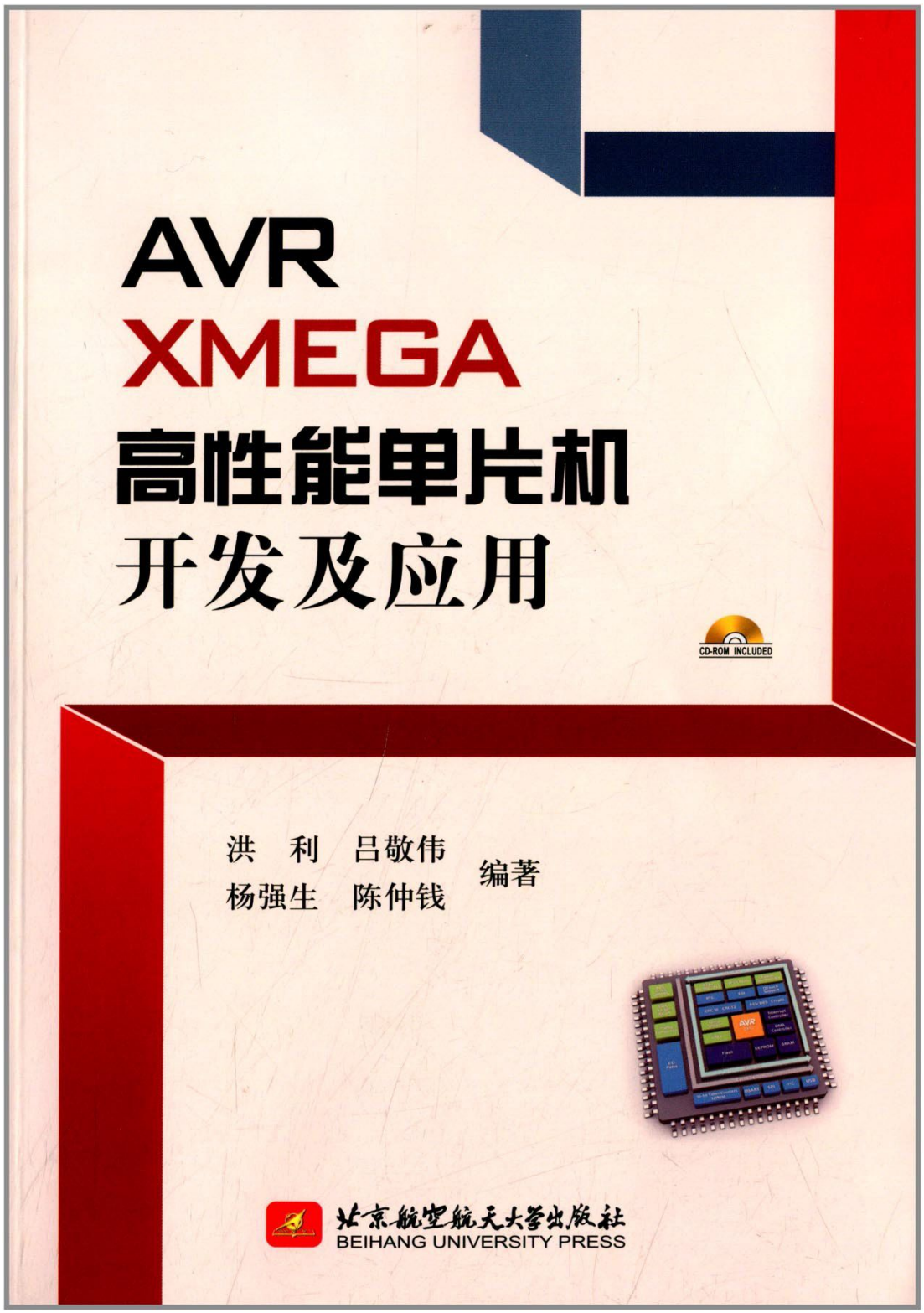
AVR XMEGA 片内外设应用

注：本内容出自《AVR XMEGA 高性能单片机开发及应用》一书的资料光盘，仅作技术交流之用，不得用于任何商业用途！



经过理论学习之后，从本章开始我们进入实践操练。本章从最基本的实例讲起，希望读者可以一步一步地跟着练习下去，通过本章学习对 AVR Xmega 单片机有更深的了解。

当然，要成功开发一个单片机系统首先要有相关的硬件设备，如计算机，仿真器等开发工具；其次还要有相关的软件配合，如 WINAVR,AVRSTUDIO,PROTEL 等。对于初学者来说，在学习本章内容的同时要不断地回顾前几章的内容，因为本章所有的实例全部要以前面几章为基础，只有学好了前面的内容再来学本章才会事半功倍。

由于 XMEGA 寄存器众多，采用 C 语言和汇编语言对 XMEGA 编程需要大量查找寄存器的配置说明，不但影响编程效率，同时也给编程人员带来诸多痛苦。为了解决这个问题，在底层寄存器与应用程序之间添加一层 XMEGA 片内外设驱动，这样编程人员在不了解底层寄存器配置说明的情况下，仍然可以很好的使用一些特定的功能。有关 XMEGA 片内外设驱动函数简介见光盘中附录 D，XMEGA 片内外设驱动源代码见附录 E 与附录 F。

5.1 I/O 基础应用实例

XMEGA 有灵活的通用 I/O（GPIO）端口。每个端口从引脚 0 到第 7 引脚共 8 个引脚，每个引脚可被配置为输入或输出。端口还有以下功能：中断，同步/异步输入检测和异步发送唤醒信号。可以单步配置使多个引脚具有相同的配置。所有端口作为通用 I/O 端口时都可以读-修改-写（RMW）。

1. 简易 I/O 引脚的控制

通过不断改变 PD4 和 PD5 两个引脚的电平来控制二极管的亮灭。

由于程序执行速度很快，如果在很短的时间内改变 PD4 和 PD5 的状态，人眼是看不出来的，所以中间必须有个延时程序。硬件连接见图 5-1-1：

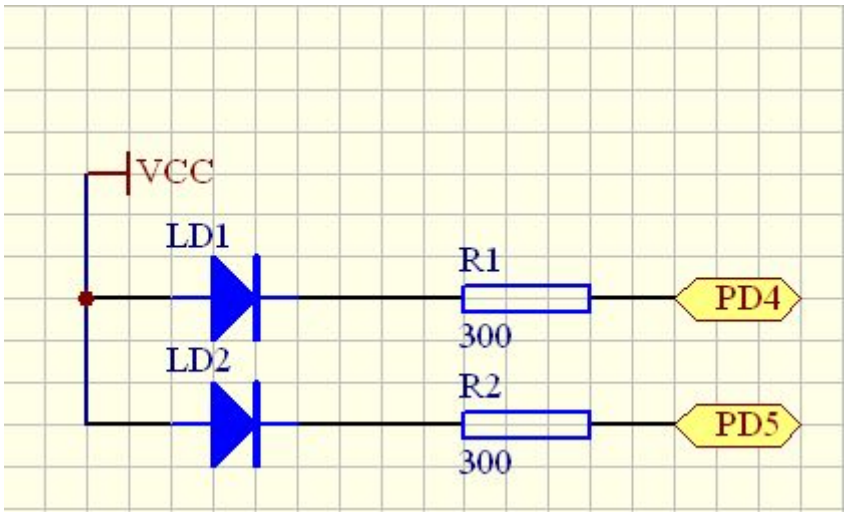


图 5-1-1 LED 发光管连接电路

C 语言代码：

```
//--------包含头文件-------//
    #include <avr/io.h>
```

```
   #include <util/delay.h>
//--------宏定义-------//
   #define LED1_ON()  PORTD_OUTCLR = 0x20
   #define LED1_OFF() PORTD_OUTSET = 0x20
   #define LED2_ON()  PORTD_OUTCLR = 0x10
   #define LED2_OFF() PORTD_OUTSET = 0x10
//--------main-------//
   int main()
   {
       PORTD_DIR = 0x30;//PD5，PD4 方向设为输出
       while(1)
       {
           LED1_ON();
           LED2_ON();
           _delay_ms(500);
           LED1_OFF();
           LED2_OFF();
           _delay_ms(500);
       }
    }
```

汇编代码：

```
//--------包含头文件-------//
.include "ATxmega128A1def.inc"//器件配置文件,决不可少,不然汇编通不过
.ORG 0
        RJMP RESET
.ORG 0X100          //跳过中断区 0x00-0x0F4
//-------RESET--------//
RESET:
         LDI R16,0x30
         STS PORTD_DIR,R16  //PD5，PD4 方向设为输出
REST_LOOP:
         LDI R16,0x30
         STS PORTD_OUTCLR,R16
         LDI R17,200           //设置延时参数
         CALL _delay_ms
         LDI R16,0x30
         STS PORTD_OUTSET,R16
         LDI R17,200            //设置延时参数
         CALL _delay_ms
         RJMP REST_LOOP
//--------_delay_ms -------//
_delay_ms:
L0:      LDI R18,250
```

```
L1:     DEC R18
        BRNE L1
        DEC R17
        BRNE L0
        RET
```

2．五维按键输入控制LED的亮灭

五维按键即五向导航键，集上下左右和确认键于一身，外观显得非常简洁，操作时通过一键实现五键的功能，灵活多变，避免了反复按键的单调，增加操作时的乐趣。

硬件设计采用PE口的PE0~PE4连接五维键盘的五个键，接收键的输入输入。程序中设置五位按键对应的E端口的低五位为输入，并使能上拉电阻。 PD4连接LED1,PD5连接LED2, 配置LED对应的端口为输出，并初始化PD4和PD5两个引脚输出为低电平,点亮LED1和LED2;

主程序判断用户按键输入，如果：

按中间的确定键 -- LED1、LED2端口取反，LED1、LED2交替亮灭 ；

按左键 -- LED1开 ；

按右键 -- LED1关 ；

按上键 -- LED2开 ；

按下键 -- LED2关 ；

硬件连接见图5-1-2：

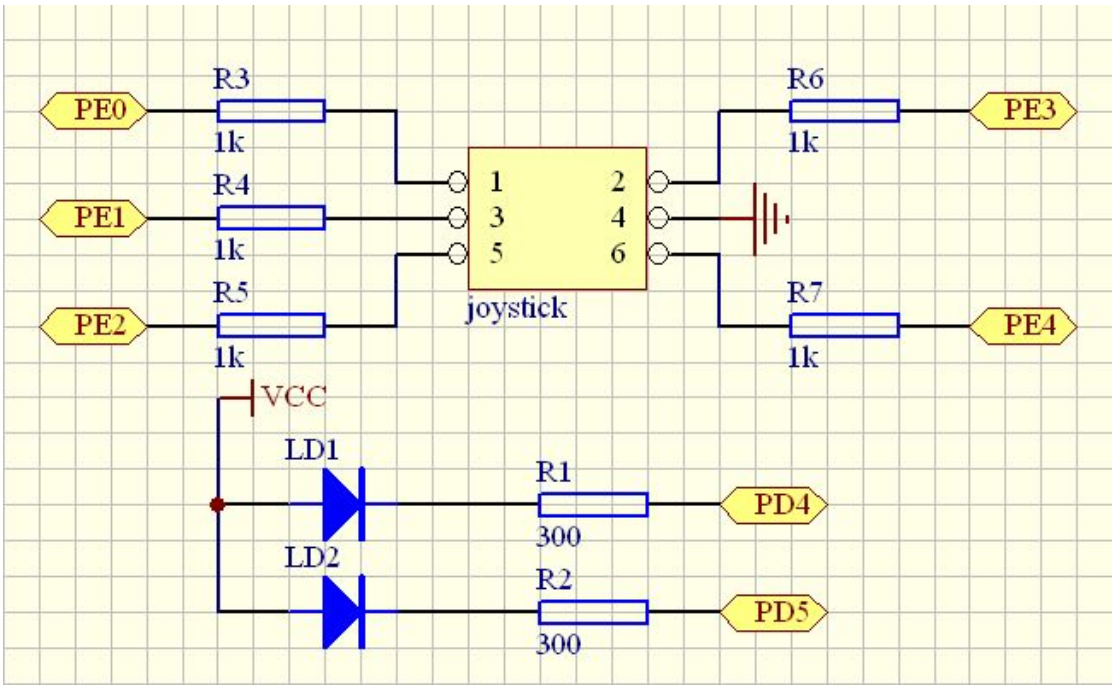


图5-1-2 五位按键控制LED连接电路

C 语言代码：

```
//--------包含头文件--------//
#include <avr/io.h>
//---------宏定义--------//
#define LED1_ON()  PORTD_OUTCLR = 0x20
#define LED1_OFF() PORTD_OUTSET = 0x20
```

```
#define LED1_T()   PORTD_OUTTGL = 0x20
#define LED2_ON()  PORTD_OUTCLR = 0x10
#define LED2_OFF() PORTD_OUTSET = 0x10
#define LED2_T()   PORTD_OUTTGL = 0x10
//--------按键返回值-------//
#define No_key   0x00
#define SELECT   0x01
#define LEFT     0x02
#define RIGHT    0x04
#define UP       0x08
#define DOWN     0x10
//-------- KEY_initial -------//
void KEY_initial(void)
{
    PORTE_DIRCLR = 0x1F;//设置按键引脚为输入
    /*
    PORTE_PIN0CTRL = PORT_OPC_PULLUP_gc;
    PORTE_PIN1CTRL = PORT_OPC_PULLUP_gc;
    PORTE_PIN2CTRL = PORT_OPC_PULLUP_gc;
    PORTE_PIN3CTRL = PORT_OPC_PULLUP_gc;
    PORTE_PIN4CTRL = PORT_OPC_PULLUP_gc;
    */
    //当有多个引脚的配置相同时，可以使用多引脚配置掩码寄存器一次配置多个引脚
    PORTCFG_MPCMASK = 0X1F;
    PORTE_PIN0CTRL = PORT_OPC_PULLUP_gc;
}
 //-------- char Get_Key -------//
unsigned char Get_Key(void)
{
    unsigned char Key=0,num_keypress = 0;
    if((PORTE_IN&(1<<1))==0)
        {
            Key|=SELECT;
            num_keypress++;
            }
    if((PORTE_IN&(1<<0))==0)
        {
            Key|=LEFT;
            num_keypress++;
            }
    if((PORTE_IN&(1<<4))==0)
        {
            Key|=RIGHT;
            num_keypress++;
```

```
            }
    if((PORTE_IN&(1<<2))==0)
        {
            Key|=UP;
            num_keypress++;
            }
    if((PORTE_IN&(1<<3))==0)
        {
            Key|=DOWN;
            num_keypress++;
            }
    if(num_keypress>1)
        Key=No_key;
    return Key;
}
//--------main-------//
int main(void)
{
    unsigned char Key_return = 0;
    PORTD_DIRSET = 0x30;//PD5，PD4方向设为输出
    LED1_ON();
    LED2_ON();
    KEY_initial();//初始化按键引脚
    while(1)
    {
        Key_return = Get_Key();
        if(Key_return)
        {
            switch(Key_return)
            {
                case SELECT:
                    LED1_T();
                    LED2_T();
                    break;
                case LEFT  :
                    LED1_ON();
                    break;
                case RIGHT :
                    LED1_OFF();
                    break;
                case UP    :
                    LED2_ON();
                    break;
                case DOWN  :
```

```
                    LED2_OFF();
                    break;
                default :break;
            }
            Key_return=0;
        }
    }
    return 0;
}
```

汇编代码：

```
//--------包含头文件-------//
.include "ATxmega128A1def.inc";器件配置文件,决不可少,不然汇编通不过
.ORG 0
        RJMP RESET
.ORG 0X100        ;跳过中断区0x00-0x0F4
//--------按键返回值-------//
.EQU No_key=0x00
.EQU SELECT=0x01
.EQU LEFT=0x02
.EQU RIGHT=0x04
.EQU UP=0x08
.EQU DOWN=0x10
//-------- KEY_initial -------//
KEY_initial:
            LDI R16,0X1F
            STS PORTE_DIRCLR,R16;//设置按键引脚为输入
            /*
            PORTE_PIN0CTRL = PORT_OPC_PULLUP_gc;
            PORTE_PIN1CTRL = PORT_OPC_PULLUP_gc;
            PORTE_PIN2CTRL = PORT_OPC_PULLUP_gc;
            PORTE_PIN3CTRL = PORT_OPC_PULLUP_gc;
            PORTE_PIN4CTRL = PORT_OPC_PULLUP_gc;
            */
           //当有多个引脚的配置相同时，可以使用多引脚配置掩码寄存器一次配置多个
引脚
            LDI R16,0X1F
            STS PORTCFG_MPCMASK,R16
            LDI R16,PORT_OPC_PULLUP_gc
            STS PORTE_PIN0CTRL,R16
            RET
 //-------- Get_Key -------//
Get_Key:
            EOR R18,R18
```

```
            EOR R17,R17
            LDS R16,PORTE_IN
            MOV R19,R16
            ANDI R19,0X01
            SBRS R19,0
            JMP Get_Key_1
Get_Key_11:
            MOV R19,R16
            ANDI R19,0X02
            SBRS R19,1
            JMP Get_Key_2
Get_Key_22:
            MOV R19,R16
            ANDI R19,0X04
            SBRS R19,2
            JMP Get_Key_3
Get_Key_33:
            MOV R19,R16
            ANDI R19,0X08
            SBRS R19,3
            JMP Get_Key_4
Get_Key_44:
            MOV R19,R16
            ANDI R19,0X10
            SBRS R19,4
            JMP Get_Key_5
            NOP
            JMP Get_Key_6
Get_Key_1:
            LDI R17,LEFT
           INC R18
           JMP Get_Key_11
Get_Key_2:
            LDI R17,SELECT
           INC R18
           JMP Get_Key_22
Get_Key_3:
           LDI R17,UP
           INC R18
           JMP Get_Key_33
Get_Key_4:
           LDI R17,DOWN
          INC R18
          JMP Get_Key_44
```

```
Get_Key_5:
           LDI R17,RIGHT
          INC R18
Get_Key_6:
          CLZ
          CPI R18,1
          BREQ Get_Key_END
          LDI R17,No_key
Get_Key_END:
          RET
 //--------宏定义 -------//
.MACRO LED1_ON
            LDI R16,@0
            STS PORTD_OUTSET,R16
.ENDMACRO
.MACRO LED2_ON
            LDI R16,@0
            STS PORTD_OUTSET,R16
.ENDMACRO
.MACRO LED1_OFF
            LDI R16,@0
            STS PORTD_OUTCLR,R16
.ENDMACRO
.MACRO LED2_OFF
            LDI R16,@0
            STS PORTD_OUTCLR,R16
.ENDMACRO
.MACRO LED1_T
            LDI R16,@0
            STS PORTD_OUTTGL,R16
.ENDMACRO
.MACRO LED2_T
            LDI R16,@0
            STS PORTD_OUTTGL,R16
.ENDMACRO
//--------RESET -------//
RESET:
           LDI R16,0x30
           STS PORTD_DIRSET,R16;//PD5，PD4方向设为输出
           LED1_ON 0X20
           LED2_ON 0X10
           CALL KEY_initial
RESET_LOOP:
           CALL  Get_Key
```

```
            CLZ
            CPI R17,0
            BREQ RESET_END
            CLZ
            CPI R17,SELECT
            BREQ RESET_1
            CPI R17,LEFT
            BREQ RESET_2
            CPI R17,RIGHT
            BREQ RESET_3
            CPI R17,UP
            BREQ RESET_4
            CPI R17,DOWN
            BREQ RESET_5
            LED2_OFF 0X10
            LDI R17,0
            JMP RESET_END
RESET_1:
            LED1_T 0X20
            LED2_T 0X10
            LDI R17,0
            JMP RESET_END
RESET_2:
            LED1_ON 0X20
            LDI R17,0
            JMP RESET_END
RESET_3:
            LED1_OFF 0X20
            LDI R17,0
            JMP RESET_END
RESET_4:
            LED2_ON 0X10
            LDI R17,0
            JMP RESET_END
RESET_5:
            LED2_OFF 0X10
            LDI R17,0
            JMP RESET_END
RESET_END:
            JMP RESET_LOOP
```

5.2 系统时钟实例

XMEGA 具有灵活的时钟系统，支持多种时钟源。高频锁相环和时钟分频器可以产生较宽范围的时钟频率。校准功能可自动校准内部振荡器。晶振失效监视可以发出非屏蔽中断，如果外部振荡器失效，自动切换到内部振荡器。

复位后，设备始终是从 2MHz 内部振荡器开始运行。正常操作过程中，系统时钟源和分频值可以在程序中随时修改。

1. 系统时钟的配置。时钟源被分为两类：内部振荡器和外部时钟源。内部振荡器有 32KHz 超低功耗振荡器；32.768KHz 校准内部振荡器；32MHz 运行校准内部振荡器；2MHz 运行校准内部振荡器。外部时钟源有 0.4 - 16MHz 晶体振荡器；外部时钟输入；32.768KHz 晶体振荡器。所有校准的内部振荡器、外部时钟源（XOSC）和锁相环 PLL 输出都可作为系统时钟源。程序中系统时钟源有三种选择。系统时钟源为外部晶体振荡器时 PLL 倍频系数设置为 6，预分频系数设置为 1；系统时钟选择为 2MHz 运行校准内部振荡器或 2MHz 运行校准内部振荡器时，预分频系数设置都为 1。

C 语言代码：

```
//---------包含头文件--------//
#include <avr/io.h>
#include "avr_compiler.h"
#include "clksys_driver.h"
//---------LED操作宏定义--------//
#define LED1_ON()  PORTD_OUTCLR = 0x20
#define LED1_OFF() PORTD_OUTSET = 0x20
#define LED1_T()   PORTD_OUTTGL = 0x20
#define LED2_ON()  PORTD_OUTCLR = 0x10
#define LED2_OFF() PORTD_OUTSET = 0x10
#define LED2_T()   PORTD_OUTTGL = 0x10
//--------- PLL_XOSC_Initial --------//
void PLL_XOSC_Initial(void)
{
    unsigned char factor = 6;
    /*设置晶振范围 启动时间*/
    CLKSYS_XOSC_Config( OSC_FRQRANGE_2TO9_gc,
false,OSC_XOSCSEL_XTAL_16KCLK_gc );
    CLKSYS_Enable( OSC_XOSCEN_bm  );//使能外部振荡器
    do {} while ( CLKSYS_IsReady( OSC_XOSCRDY_bm  ) == 0 );//等待外部振荡器准备好
    /*设置倍频因子并选择外部振荡器为PLL参考时钟*/
    CLKSYS_PLL_Config( OSC_PLLSRC_XOSC_gc, factor );
    CLKSYS_Enable( OSC_PLLEN_bm );//使能PLL电路
    do {} while ( CLKSYS_IsReady( OSC_PLLRDY_bm ) == 0 );//等待PLL准备好
    CLKSYS_Main_ClockSource_Select( CLK_SCLKSEL_PLL_gc);//选择系统时钟源
    /*设置预分频器A,B,C的值*/
    CLKSYS_Prescalers_Config( CLK_PSADIV_1_gc, CLK_PSBCDIV_1_1_gc );
```

```
}
//--------- RC32M_Initial --------//
void RC32M_Initial(void)
{
    CLKSYS_Enable( OSC_RC32MEN_bm  );//使能RC32M振荡器
    do {} while ( CLKSYS_IsReady( OSC_RC32MRDY_bm  ) == 0 );//等待RC32M振荡器准
备好
    CLKSYS_Main_ClockSource_Select( CLK_SCLKSEL_RC32M_gc);//选择系统时钟源
    /*设置预分频器A,B,C的值*/
    CLKSYS_Prescalers_Config( CLK_PSADIV_1_gc, CLK_PSBCDIV_1_1_gc );
}
//--------- RC2M_Initial --------//
void RC2M_Initial(void)
{
    CLKSYS_Enable( OSC_RC2MEN_bm  );//使能RC2M振荡器
    do {} while ( CLKSYS_IsReady( OSC_RC2MRDY_bm  ) == 0 );//等待RC2M振荡器准备
好
    CLKSYS_Main_ClockSource_Select( CLK_SCLKSEL_RC2M_gc);//选择系统时钟源
    /*设置预分频器A,B,C的值*/
    CLKSYS_Prescalers_Config( CLK_PSADIV_1_gc, CLK_PSBCDIV_1_1_gc );
}
 //---------main--------//
int main( void )
{
    PLL_XOSC_Initial();// 外部晶振8M，PLL输出8M*6=48M
    /*RC32M_Initial();//内部RC32M
        RC2M_Initial();内部RC2M*/
    PORTD_DIRSET = 0x30;//PD5，PD4方向设为输出
    TCC0.PER = 31250;
    TCC0.CTRLA = ( TCC0.CTRLA & ~TC0_CLKSEL_gm ) | TC_CLKSEL_DIV64_gc;
    TCC0.INTCTRLA = ( TCC0.INTCTRLA & ~TC0_OVFINTLVL_gm ) | TC_OVFINTLVL_MED_gc;
    /*使能低级中断和全局中断 */
    PMIC.CTRL |= PMIC_MEDLVLEN_bm;
    sei();
    while(1)
    {}
}
//-------- ISR （TCC0溢出中断函数）--------//
ISR(TCC0_OVF_vect)
{
    LED1_T();
}
```

汇编代码：

```
//---------包含头文件--------//
```

```
.include "ATxmega128A1def.inc";器件配置文件,决不可少,不然汇编通不过
.ORG 0
        RJMP RESET//复位
.ORG 0x01C          //TCC0溢出中断向量
        RJMP ISR
.ORG 0X100        ;跳过中断区0x00-0x0F4
//---------宏定义--------//
.MACRO LED_T1
           LDI R16,@0
           STS PORTD_OUTTGL,R16
.ENDMACRO
.MACRO LED_T2
           LDI R16,@0
           STS PORTD_OUTTGL,R16
.ENDMACRO

.MACRO CLKSYS_IsReady
 CLKSYS_IsReady_1:
           LDS R16,OSC_STATUS
           SBRS R16,@0
           JMP CLKSYS_IsReady_1//等待振荡器准备好
           NOP
.ENDMACRO
 //--------- PLL_XOSC_Initial（外部时钟选择）--------//
PLL_XOSC_Initial:
                 LDI R16,0X4B
                STS OSC_XOSCCTRL,R16//设置晶振范围 启动时间
                 LDI R16,OSC_XOSCEN_bm
                 STS OSC_CTRL,R16//使能外部振荡器
/*CLKSYS_IsReady_1:
                 LDS R16,OSC_STATUS
                 SBRS R16,OSC_XOSCRDY_bp
                 JMP CLKSYS_IsReady_1//等待外部振荡器准备好
                 NOP
*/
                 CLKSYS_IsReady OSC_XOSCRDY_bp
                 LDI R16,OSC_PLLSRC_XOSC_gc
                 ORI R16,0XC6
                  STS OSC_PLLCTRL,R16
                 LDS R16,OSC_CTRL//读取该寄存器的值到R16
                 SBR R16,OSC_PLLEN_bm//对PLLEN这一位置位，使能PLL
                 STS OSC_CTRL,R16
/*CLKSYS_IsReady_2:
                  LDS R16,OSC_STATUS
```

```
                SBRS R16,OSC_PLLRDY_bp
                JMP CLKSYS_IsReady_2//等待外部振荡器准备好
                NOP
*/
                 CLKSYS_IsReady OSC_PLLRDY_bp
                LDI R16,CLK_SCLKSEL_PLL_gc
                LDI R17,0XD8//密钥
                STS CPU_CCP,R17//解锁
                STS CLK_CTRL,R16//选择系统时钟源
                LDI R16,CLK_PSADIV_1_gc
                ORI R16,CLK_PSBCDIV_1_1_gc
                 LDI R17,0XD8//密钥
                STS CPU_CCP,R17//解锁
                STS CLK_PSCTRL,R16//设置预分频器A,B,C的值
                RET
//--------- RC32M_Initial（RC32M时钟选择）--------//
RC32M_Initial:
                LDI R16,OSC_RC32MEN_bm
                STS OSC_CTRL,R16//使能RC32M振荡器
                CLKSYS_IsReady OSC_RC32MRDY_bm//等待RC32M振荡器准备好
                LDI R16, CLK_SCLKSEL_RC32M_gc
                LDI R17,0XD8//密钥
                STS CPU_CCP,R17//解锁
                STS CLK_CTRL,R16//选择系统时钟源
                LDI R16,CLK_PSADIV_1_gc
                ORI R16,CLK_PSBCDIV_1_1_gc
                LDI R17,0XD8//密钥
                STS CPU_CCP,R17//解锁
                STS CLK_PSCTRL,R16//设置预分频器A,B,C的值
                RET
 //--------- RC2M_Initial（RC2M时钟选择）--------//
RC2M_Initial:
                LDI R16,OSC_RC2MEN_bm
                STS OSC_CTRL,R16// 使能RC2M振荡器
                CLKSYS_IsReady OSC_RC2MRDY_bm//等待RC2M振荡器准备好
                LDI R17,0XD8//密钥
                STS CPU_CCP,R17//解锁
                LDI R16, CLK_SCLKSEL_RC2M_gc
                STS CLK_CTRL,R16//选择系统时钟源
                    LDI R17,0XD8//密钥
                STS CPU_CCP,R17//解锁
                 LDI R16,CLK_PSADIV_1_gc
                ORI R16,CLK_PSBCDIV_1_1_gc
                STS CLK_PSCTRL,R16//设置预分频器A,B,C的值
```

```
                    RET
    //--------REST(函数入口)--------//
RESET:
                    CALL PLL_XOSC_Initial// PLL，外部晶振8M，输出8M*6=48M
                    /*
                    RC32M_Initial();//内部RC32M
                    RC2M_Initial();//内部RC2M
                     */
                     LDI R16,0X30
                     STS PORTD_DIRSET,R16;//PD5，PD4方向设为输出
                    LDI R16,0X0FF//设置定时器C0，计数周期65535
                     STS TCC0_PER,R16
                    STS TCC0_PER+1,R16
                    LDI R16,TC_CLKSEL_DIV64_gc
                     STS TCC0_CTRLA ,R16
                     LDI R16,TC_OVFINTLVL_MED_gc
                    STS TCC0_INTCTRLA,R16
                    LDI R16,PMIC_HILVLEN_bp//使能高级别中断，打开全局中断
                     STS PMIC_CTRL,R16
                     SEI
RESET_LOOP:
                     RJMP RESET_LOOP
//--------ISR(TCC0溢出中断)--------//
ISR:
                     LED_T1 0X20
                     RETI
```

5.3 异步串行接收/发送器实例

通用同步/异步串行接收/发送器（USART）是一个高度灵活的串行通信模块。USART 支持全双工通信，同步或者异步操作。通信是基于帧的，帧结构支持定制很宽的标准。USART 双向缓冲，帧与帧之间连续传输没有任何延时。接收和发送完毕都有单独的中断向量，支持全中断驱动的通信。帧错误和缓冲溢出都可以被硬件检测到并有独立的状态标志位。奇偶校验值的产生和校验也可以使能。

1.单片机串行口将 PC 端发送过来的数据接收并返回给 PC。

进行串口通信时要满足一定的条件，比如计算机的串口是 RS232 电平的，而单片机的串口电平是 TTL 电平的，两者之间必须要有一个电平转换电路，我们采用专用芯片 MAX232 进行转换；虽然可以用几个三极管进行电平转换，但是用专用专用芯片更简单可靠。MAX232 芯片是 MAXIM 公司生产的，包含两路接收器和驱动器的 IC 芯片。外部引脚见图 5-3-1

程序设置 USARTC0 的发送引脚 PC3 为输出，接收引脚 PC2 为输入；USARTC0 传输模式为异步；USARTC0 的帧格式为 8 位数据位，无校验，1 位停止；波特率为 9600bps；并使能 USARTC0 接收与发送。硬件连接见图 5-3-1：

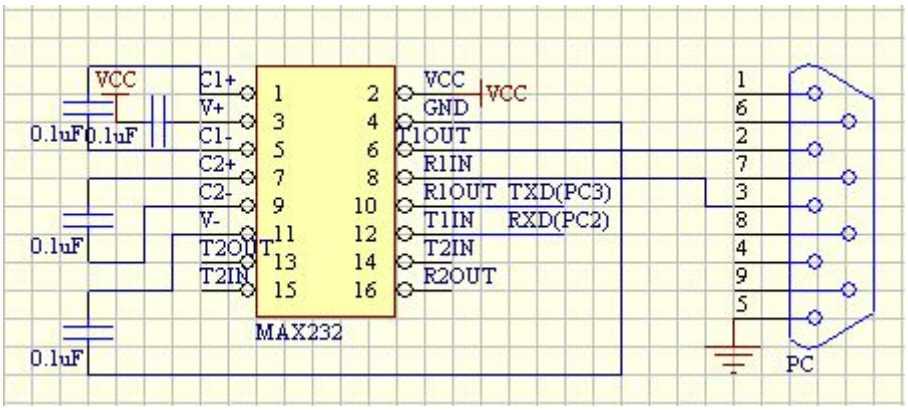


图 5-3-1 电平转换原理图

在实际运用中一定要保证上位机设置与单片机统一，否则数据将会出错。

C 语言代码：

```
//--------包含头文件--------//
#include <avr/io.h>
#include "avr_compiler.h"
#include <avr/interrupt.h>
#include "usart_driver.c"

//-------- uart_init (串口初始化)--------//
void uart_init(void)
{
    /* USARTC0 引脚方向设置*/
    PORTC.DIRSET   = PIN3_bm; // PC3 (TXD0) 输出
    PORTC.DIRCLR   = PIN2_bm; // PC2 (RXD0) 输入
    USART_SetMode(&USARTC0,USART_CMODE_ASYNCHRONOUS_gc); // USARTC0 模式 - 异步
    /*USARTC0帧结构, 8 位数据位, 无校验, 1停止位*/
    USART_Format_Set(&USARTC0, USART_CHSIZE_8BIT_gc,USART_PMODE_DISABLED_gc,
false);

    USART_Baudrate_Set(&USARTC0, 12 , 0); //设置波特率 9600
    USART_Tx_Enable(&USARTC0); // USARTC0 使能发送
    USART_Rx_Enable(&USARTC0); //USARTC0 使能接收
}
//--------main（主函数）--------//
int main(void)
{
    uart_init();
    uart_putc('\n');
    uart_putw_hex(0xFFFF);
    uart_putc('=');
    uart_putdw_dec(0xFFFF);
    uart_putc('\n');
```

```
    uart_putdw_hex(0xAABBCCDD);
    uart_putc('=');
    uart_putdw_dec(0xAABBCCDD);
    uart_putc('\n');
    uart_puts("www.upc.edu.cn");
    uart_putc('\n');
    //USARTC0 接收中断级别
    USART_RxdInterruptLevel_Set(&USARTC0,USART_RXCINTLVL_LO_gc);
    PMIC.CTRL |= PMIC_LOLVLEN_bm; //使能中断级别
    sei();
    while(1);
    return 0;
}
//-------- ISR (串口接收中断)--------//
ISR(USARTC0_RXC_vect)
{
    unsigned char buffer;
    buffer = USART_GetChar(&USARTC0);
    do{
        }while(!USART_IsTXDataRegisterEmpty(&USARTC0));
    USART_PutChar(&USARTC0,buffer);
}
```

汇编代码：

```
 //--------包含头文件--------//
.include "ATxmega128A1def.inc";器件配置文件,决不可少,不然汇编通不过
.include "usart_driver.inc"
.ORG 0
        RJMP RESET
.ORG 0x032        //USARTC0数据接收完毕中断入口
        RJMP ISR
.ORG 0X100       ;跳过中断区0x00-0x0F4
.EQU ENTER  ='\n'
.EQU NEWLINE='\r'
.EQU EQUE   ='='
.EQU ZERO   ='0'
.EQU A_ASSCI='A'
STRING: .DB 'W','W','W','.','U','P','C','.','E','D','U','.','C','N',0
 //-------- uart_init (串口初始化)--------//
uart_init:
           /* USARTC0 引脚方向设置*/
           LDI R16,0X08 //PC3 (TXD0) 输出
           STS PORTC_DIRSET,R16
           LDI R16,0X04 //PC2 (RXD0) 输入
           STS PORTC_DIRCLR,R16
```

```
            // USARTC0 模式 - 异步 USARTC0帧结构, 8 位数据位, 无校验, 1停止位
            LDIR16,USART_CMODE_ASYNCHRONOUS_gc|USART_CHSIZE_8BIT_gc
                                              |USART_PMODE_DISABLED_gc
            STS USARTC0_CTRLC,R16
            LDI R16,12  //设置波特率 9600
            STS USARTC0_BAUDCTRLA,R16
            LDI R16,0
            STS USARTC0_BAUDCTRLB,R16
            LDI R16,USART_TXEN_bm|USART_RXEN_bm  //USARTC0 使能发送 USARTC0 使能
接收
            STS USARTC0_CTRLB,R16
            RET
  //-------- RESET(主函数入口)--------//
RESET:
              CALL uart_init
              LDI R16,ENTER
              PUSH R16
              uart_putc
              EOR XH,XH
              EOR XL,XL
              COM XH
              COM XL
              uart_putw_hex
              LDI R16,EQUE
              PUSH R16
              uart_putc
              EOR XH,XH
              EOR XL,XL
              COM XH
              COM XL
              uart_putw_dec
              LDI R16,ENTER
              PUSH R16
              uart_putc
              uart_puts_string STRING
              LDI R16,ENTER
              PUSH R16
              uart_putc
              LDI R16,USART_RXCINTLVL_LO_gc//USARTC0 接收中断级别
              STS USARTC0_CTRLA,R16
              LDI R16,PMIC_LOLVLEN_bm //使能中断级别
              STS PMIC_CTRL,R16
              SEI
 RESET_LOOP:
```

```
            NOP
            NOP
            JMP RESET_LOOP
//-------- ISR(串口中断)--------//
ISR:
             LDS R17,USARTC0_DATA
             USART_IsTXDataRegisterEmpty USART_DREIF_bp
             STS USARTC0_DATA,R17
             RETI
```

5.4 TC - 16-bit 定时器/计数器实例

XMEGA 有一组高级灵活的 16 位定时器/计数器（TC）。它的基本功能包括准确的计时、频率和波形的产生、事件管理以及数字信号的时间测量。高分辨率扩展（Hi-Res）和高级波形扩展（AWeX）可以和定时器/计数器配合使用，轻松地产生复杂和专门的频率、波形。

1.定时器的计数，捕获功能。分六个例子函数实现：

Example1：定时器TCC0基本计数。设置定时器TCC0计数周期为0X1000, 定时器TCC0时钟为系统时钟。

Example2：定时器TCC0通道A输入捕获。设置TCC0计数周期为255，TCC0时钟为系统时钟,使能TCC0的A通道,选择事件通道2为TCC0的事件源,PE3设置为下降沿触发 输入上拉，PE3作为通道2的事件源；当PE3产生下降沿则TCC0的A通道发生捕获，D端口LED显示捕获值的低八位，实现TCC0通道A输入捕获。

Example3：定时器TCC0测量PC0口信号频率。设置TCC0计数周期为0X7FFF（如果设置的周期寄存器的值比0x8000小，捕获完成后将把I/O引脚的电平变化存储在捕获寄存器的最高位），TCC0时钟为系统时钟，使能TCC0的A通道，选择事件通道0为TCC0的事件源，PC0设置为双沿触发 输入上拉，PC0作为通道0的事件源；当PC0产生下降沿或者上升沿时则TCC0的A通道发生捕获，进入捕获中断；通过检测最高位看是否是上升沿，两次上升沿捕获值相减，可得到周期；也可以通过检测最高位算出占空比。

Example4 ：定时器TCC0通道B占空比变化的脉宽调制输出。设置TCC0为单斜率PWM模式，TCC0的计数周期为512，时钟为系统时钟的64分之一，使能TCC0的B通道，每次在溢出时将新的值填到比较或捕获缓冲寄存器,在下次溢出时比较或捕获缓冲寄存器的值就会自动加载到比较或捕获寄存器，从而改变PC1引脚上的脉宽。

Example5：定时器TCC0对事件信号计数。设置TCC0计数周期为4，TCC0时钟为事件通道0，PE3设置为下降沿触发 输入上拉，PE3作为通道0的事件源；当PE3产生下降沿则定时器TCC0的计数寄存器加一，计数寄存器溢出时会进入溢出中断函数，对PD4和PD5电平取反。

Example6：定时器TCC0和TCC1实现32位计数。设置TCC0计数周期为1250，TCC0时钟为系统时钟的八分之一；TCC1计数周期为200，TCC1时钟为事件通道0；选择TCC0溢出为事件通道0的事件源，所以一旦TCC0溢出TCC1计数就会加一。

C语言代码：

```
//-------- CPU 时钟和分频值 --------//
#define F_CPU            2000000UL
#define CPU_PRESCALER    1
```

```
//--------包含头文件 --------//
#include "avr_compiler.h"
#include "usart_driver.c"
#include "TC_driver.c"
//-------函数申明--------//
void Example1( void );
void Example2( void );
void Example3( void );
void Example4( void );
void Example5( void );
void Example6( void );
//--------宏定义 --------//
#define LED1_ON()  PORTD_OUTCLR = 0x20
#define LED1_OFF() PORTD_OUTSET = 0x20
#define LED1_T()   PORTD_OUTTGL = 0x20
#define LED2_ON()  PORTD_OUTCLR = 0x10
#define LED2_OFF() PORTD_OUTSET = 0x10
#define LED2_T()   PORTD_OUTTGL = 0x10
 //-------- uart_init（串口初始化） --------//
void uart_init(void)
{
    /* USARTC0 引脚方向设置*/
    PORTC.DIRSET   = PIN3_bm; //PC3 (TXD0) 输出
    PORTC.DIRCLR   = PIN2_bm; //PC2 (RXD0) 输入
    USART_SetMode(&USARTC0,USART_CMODE_ASYNCHRONOUS_gc); //USARTC0 模式 - 异步
    // USARTC0帧结构, 8 位数据位, 无校验, 1停止位
    USART_Format_Set(&USARTC0, USART_CHSIZE_8BIT_gc,USART_PMODE_DISABLED_gc,
false);
    USART_Baudrate_Set(&USARTC0, 12 , 0); //设置波特率 9600
    USART_Tx_Enable(&USARTC0); //USARTC0 使能发送
    USART_Rx_Enable(&USARTC0); //USARTC0 使能接收
}
 //-------- main（主函数） --------//
int main( void )
{
    uart_init();
    //Example1();
    //Example2();
    //Example3();
    Example4();
    //Example5();
    //Example6();
}
 //-------- Example1--------//
```

```
void Example1( void )
{
      TC_SetPeriod( &TCC0, 0x1000 );
      TC0_ConfigClockSource( &TCC0, TC_CLKSEL_DIV1_gc );
      do {} while (1);
}
  //-------- Example2--------//
void Example2( void )
{
    uint16_t inputCaptureTime;
   //PE3设为输入，下降沿触发 输入上拉 当I/O引脚作为事件的捕获源，该引脚必须配置
   为边//沿检测
    PORTE.PIN3CTRL = PORT_ISC_FALLING_gc + PORT_OPC_PULLUP_gc;
    PORTE.DIRCLR = 0x08;
    PORTD.DIRSET = 0xFF; //Port D设为输出
    EVSYS.CH2MUX = EVSYS_CHMUX_PORTE_PIN3_gc; //PE0作为事件通道2的输入.
    //设置 TCC0 输入捕获使用事件通道2
    TC0_ConfigInputCapture( &TCC0, TC_EVSEL_CH2_gc );
    TC0_EnableCCChannels( &TCC0, TC0_CCAEN_bm ); //使能通道A
     TC_SetPeriod( &TCC0, 255 );
    TC0_ConfigClockSource( &TCC0, TC_CLKSEL_DIV1_gc ); //选择时钟，启动定时器
    do {
        do {} while ( TC_GetCCAFlag( &TCC0 ) == 0 );
        /*定时器把事件发生时计数寄存器的当前计数值拷贝到CCA寄存器*/
        inputCaptureTime = TC_GetCaptureA( &TCC0 );
        PORTD.OUT = (uint8_t) (inputCaptureTime);
         } while (1);
}
//-------- Example3--------//
void Example3( void )
{
    PORTD.DIRSET = 0xFF; //Port D设为输出 LED指示
    LED1_OFF();
    LED2_ON();
    PORTC.PIN0CTRL = PORT_ISC_BOTHEDGES_gc; //PC0设为输入，双沿触发
    PORTC.DIRCLR = 0x01;
    EVSYS.CH0MUX = EVSYS_CHMUX_PORTC_PIN0_gc; //PC0作为事件通道0的输入
    /*设置 TCC0 输入捕获使用事件通道0*/
    TC0_ConfigInputCapture( &TCC0, TC_EVSEL_CH0_gc );
    TC0_EnableCCChannels( &TCC0, TC0_CCAEN_bm ); //使能通道A
    //如果设置的周期寄存器的值比0x8000小，捕获完成后将
    //把I/O引脚的电平变化存储在捕获寄存器的最高位（MSB）。
    Clear MSB of PER[H:L] to allow for propagation of edge polarity. */
    TC_SetPeriod( &TCC0, 0x7FFF );
```

```
    TC0_ConfigClockSource( &TCC0, TC_CLKSEL_DIV1_gc ); //选择时钟，启动定时器
    TC0_SetCCAIntLevel( &TCC0, TC_CCAINTLVL_LO_gc ); //使能通道A 低级别中断
    PMIC.CTRL |= PMIC_LOLVLEN_bm;
    SEI();
    do {} while (1);
}
//--------ISR(TCC0捕获中断)--------//
ISR(TCC0_CCA_vect)
{
    LED1_T();
    LED2_T();
    static uint32_t frequency;
    static uint32_t dutyCycle;
    static uint16_t totalPeriod;
    static uint16_t highPeriod;
    uint16_t thisCapture = TC_GetCaptureA( &TCC0 );
    //按上升沿来保存总周期值
    if ( thisCapture & 0x8000 ) {
        //MSB=1,代表高电平，上升沿
        totalPeriod = thisCapture & 0x7FFF;
        TC_Restart( &TCC0 );
    }
    //下降沿保存高电平的周期
    else {
        highPeriod = thisCapture;
    }
    dutyCycle = ( ( ( highPeriod * 100 ) / totalPeriod ) + dutyCycle ) / 2;
    frequency = ( ( ( F_CPU / CPU_PRESCALER ) / totalPeriod ) + frequency ) / 2;
    //串口打印占空比和频率
    uart_puts("dutyCycle = ");
    uart_putdw_dec(dutyCycle);
    uart_putc('\n');
    uart_puts("frequency = ");
    uart_putdw_dec(frequency);
    uart_putc('\n');
}
//-------- Example4--------//
void Example4( void )
{
    uint16_t compareValue = 0x0000;
    PORTC.DIRSET = 0x02; //PC1输出
    TC_SetPeriod( &TCC0, 512 ); //设置计数周期
    TC0_ConfigWGM( &TCC0, TC_WGMODE_SS_gc ); //设置TC为单斜率模式
    TC0_EnableCCChannels( &TCC0, TC0_CCBEN_bm ); //使能通道B
```

```
    TC0_ConfigClockSource( &TCC0, TC_CLKSEL_DIV64_gc ); //选择时钟，启动定时器

    do {
        //新比较值
        compareValue += 31;
        if(compareValue>512)compareValue = 31;
        TC_SetCompareB( &TCC0, compareValue ); //设置到缓冲寄存器
        //溢出时比较值从CCBBUF[H:L] 传递到CCB[H:L]
        do {} while( TC_GetOverflowFlag( &TCC0 ) == 0 );
                TC_ClearOverflowFlag( &TCC0 ); //清除溢出标志
         } while (1);
}
 //-------- Example5--------//
void Example5( void )
{
    //PE3设为输入， 输入上拉，下降沿感知，DOWN键按下计数
    PORTE.PIN3CTRL = PORT_ISC_FALLING_gc + PORT_OPC_PULLUP_gc;
    PORTE.DIRCLR = 0x08;
    PORTD.DIRSET = 0x30; // PD4设为输出
    //选择PE3为事件通道0的输入, 使能数字滤波
    EVSYS.CH0MUX = EVSYS_CHMUX_PORTE_PIN3_gc;
    EVSYS.CH0CTRL = EVSYS_DIGFILT_8SAMPLES_gc;
    TC_SetPeriod( &TCC0, 4 ); //设置计数周期值-TOP
    //设置溢出中断为低级别中断
    TC0_SetOverflowIntLevel( &TCC0, TC_OVFINTLVL_LO_gc );
    PMIC.CTRL |= PMIC_LOLVLEN_bm;
    sei();
        TC0_ConfigClockSource( &TCC0, TC_CLKSEL_EVCH0_gc ); //启动定时器
    do {} while (1);
}
ISR(TCC0_OVF_vect)
{
    PORTD.OUTTGL = 0x10; //取反PD4
    PORTD.OUTCLR = 0x20;
}
//-------- Example6--------//
void Example6( void )
{
    PORTD.DIRSET = 0x10; // PD4设为输出
    EVSYS.CH0MUX = EVSYS_CHMUX_TCC0_OVF_gc; //TCC0溢出作为事件通道0的输入
    TC_EnableEventDelay( &TCC1 ); //使能TCC1传播时延
    TC_SetPeriod( &TCC0, 1250 ); // 设置计数周期
    TC_SetPeriod( &TCC1, 200 );
    TC1_ConfigClockSource( &TCC1, TC_CLKSEL_EVCH0_gc ); //使用通道0作为TCC1时钟
```

源

```
    //使用外设时钟8分频作为TCC0时钟源 启动定时器
    TC0_ConfigClockSource( &TCC0, TC_CLKSEL_DIV8_gc );
    do {
        do {} while( TC_GetOverflowFlag( &TCC1 ) == 0 );
        PORTD.OUTTGL = 0x10; //取反PD4
        TC_ClearOverflowFlag( &TCC1 ); //清除溢出标志
    } while (1);
}
```

汇编代码：

```
 //--------包含头文件--------//
.include "ATxmega128A1def.inc";器件配置文件,决不可少,不然汇编通不过
.include "usart_driver.inc"
.ORG 0
        RJMP RESET//复位
.ORG 0X01C
        RJMP ISR_OVFIF
.ORG 0X20
        RJMP ISR_CCA_vect

.ORG 0X100        ;跳过中断区0x00-0x0FF
.EQU ENTER  ='\n'
.EQU NEWLINE='\r'
.EQU EQUE   ='='
.EQU ZERO   ='0'
.EQU A_ASSCI='A'
 //--------宏定义--------//
.MACRO LED1_ON
           LDI R16,@0
           STS PORTD_OUTSET,R16
.ENDMACRO
.MACRO LED2_ON
           LDI R16,@0
           STS PORTD_OUTSET,R16
.ENDMACRO
.MACRO LED1_OFF
           LDI R16,@0
           STS PORTD_OUTCLR,R16
.ENDMACRO
.MACRO LED2_OFF
           LDI R16,@0
           STS PORTD_OUTCLR,R16
.ENDMACRO
.MACRO LED1_T
```

```
            LDI R16,@0
            STS PORTD_OUTTGL,R16
.ENDMACRO
.MACRO LED2_T
            LDI R16,@0
            STS PORTD_OUTTGL,R16
.ENDMACRO
 //-------- uart_init （串口初始化）--------//
uart_init:
            /* USARTC0 引脚方向设置*/
            LDI R16,0X08 //PC3 (TXD0) 输出
            STS PORTC_DIRSET,R16
            LDI R16,0X04 // PC2 (RXD0) 输入
            STS PORTC_DIRCLR,R16
           // USARTC0 模式 - 异步  USARTC0帧结构, 8 位数据位, 无校验, 1停止位
            LDI R16,USART_CMODE_ASYNCHRONOUS_gc|USART_CHSIZE_8BIT_gc
                                               |USART_PMODE_DISABLED_gc
            STS USARTC0_CTRLC,R16
            LDI R16,12 //设置波特率 9600
            STS USARTC0_BAUDCTRLA,R16
            LDI R16,0
            STS USARTC0_BAUDCTRLB,R16
            LDI R16,USART_TXEN_bm|USART_RXEN_bm//USARTC0 使能发送  USARTC0 使能
接收
            STS USARTC0_CTRLB,R16
            RET
//-------- Example1--------//
Example1:
    /* Set period/TOP value. */
    LDI XH,0X10
    EOR XL,XL
    STS TCC0_PER,XH
    STS TCC0_PER+1,XL
    LDI R16,TC_CLKSEL_DIV1_gc
    STS TCC0_CTRLA,R16
Example1_0:
    JMP Example1_0
    RET
 //-------- Example2--------//
Example2:
//PE3设为输入，下降沿触发 输入上拉 当I/O引脚作为事件的捕获源，
//该引脚必须配置为边沿检测。
    LDI R16,PORT_ISC_FALLING_gc|PORT_OPC_PULLUP_gc
    STS PORTE_PIN3CTRL,R16
```

```
    LDI R16,0X08
    STS PORTE_DIRCLR,R16
    LDI R16,0X0FF//Port D设为输出
    STS PORTD_DIRSET,R16
    LDI R16,EVSYS_CHMUX_PORTE_PIN3_gc//PE3作为事件通道2的输入
    STS EVSYS_CH2MUX,R16
    LDI R16,TC_EVSEL_CH2_gc |TC_EVACT_CAPT_gc//设置 TCC0 输入捕获使用事件通道2
    STS TCC0_CTRLD,R16
    LDI R16,TC0_CCAEN_bm  //使能通道A
    STS TCC0_CTRLB,R16

    EOR XH,XH
    LDI XL,0X0FF
    STS  TCC0_PER,XL
    STS  TCC0_PER+1,XH
    LDI R16,TC_CLKSEL_DIV1_gc//选择时钟，启动定时器
    STS TCC0_CTRLA,R16
Example2_1:
    LDS R16,TCC0_INTFLAGS
    SBRS R16,TC0_CCAIF_bp
    JMP Example2_1
    NOP
    //定时器把事件发生时计数寄存器的当前计数值拷贝到CCA寄存器
    LDS R16,TCC0_CCA
    STS PORTD_OUT,R16
    JMP  Example2_1
    RET
  //-------- Example3--------//
Example3:
    LDI R16,0X0FF//Port D设为输出 LED指示
    STS PORTD_DIRSET,R16
    LED1_OFF 0X20
    LED2_ON  0X10
    LDI R16,PORT_ISC_BOTHEDGES_gc//PC0设为输入，双沿触发
    STS PORTC_PIN0CTRL,R16
    LDI R16,0X01
    STS PORTC_DIRCLR,R16
    LDI R16,EVSYS_CHMUX_PORTC_PIN0_gc//PC0作为事件通道0的输入
    STS EVSYS_CH0MUX,R16
    LDI R16,TC_EVSEL_CH0_gc |TC_EVACT_CAPT_gc//设置 TCC0 输入捕获使用事件通道0
    STS TCC0_CTRLD,R16
    LDI R16,TC0_CCAEN_bm  //使能通道A
    STS TCC0_CTRLB,R16
```

```
    LDI XH,0X7F
    LDI XL,0X0FF
    STS  TCC0_PER,XL
    STS  TCC0_PER+1,XH
    LDI R16,TC_CLKSEL_DIV1_gc//选择时钟，启动定时器
    STS TCC0_CTRLA,R16
    LDI R16,TC_CCAINTLVL_LO_gc//使能通道A 低级别中断
    STS TCC0_INTCTRLB,R16
    LDI R16,PMIC_LOLVLEN_bm
    STS PMIC_CTRL,R16
    SEI
Example3_0:
    JMP Example3_0
    RET
 //--------捕获中断--------//
ISR_CCA_vect:
    LED1_T 0X20
    LED2_T 0X10
    LDS R17,TCC0_CCA //定时器把事件发生时计数寄存器的当前计数值拷贝到CCA寄存器
    LDS R16,TCC0_CCA+1
    //按上升沿来保存总周期值
    SBRS R16,7//MSB=1,代表高电平，上升沿
    JMP Example3_1
    NOP
    ANDI R16,0X80
    MOVW XH:XL,R16:R17
    uart_putw_dec
    JMP Example3_END
Example3_1:
    MOVW XH:XL,R16:R17
    uart_putw_dec
Example3_END:
    RETI
//------- Example4--------//
Example4:
    LDI R16,0X02 //PC1输出
    STS PORTC_DIRSET,R16
        LDI XH,0X02 //设置计数周期
    EOR XL,XL
    STS TCC0_PER,XL
    STS TCC0_PER+1,XH
    LDS R16,TCC0_CTRLB //设置TC为单斜率模式
    ORI R16,TC_WGMODE_SS_gc
    STS TCC0_CTRLB,R16
```

```
    LDI R16,TC0_CCBEN_bm  //使能通道B
    STS TCC0_CTRLB,R16
    LDI R16,TC_CLKSEL_DIV64_gc //选择时钟，启动定时器
    STS TCC0_CTRLA,R16
    EOR XH,XH//新比较值
    EOR XL,XL
    EOR YH,YH
    EOR YL,YL
    LDI YH,0X02
Example4_0:
    ADIW XH:XL,31
    CLZ
    CP XL,YL
    CPC XH,YH
    BRLO Example4_1
    EOR XH,XH
    EOR XL,XL
    ADIW XH:XL,31
Example4_1:
    STS TCC0_CCBBUF,XL //设置到缓冲寄存器
    STS TCC0_CCBBUF+1,XH
Example4_2:  //溢出时比较值从CCBBUF[H:L] 传递到CCB[H:L]
    LDS R16,TCC0_INTFLAGS
    SBRS R16,TC0_OVFIF_bp
    JMP Example4_2
    NOP
    LDI R16,TC0_OVFIF_bm //清除溢出标志
    STS TCC0_INTFLAGS,R16
    JMP Example4_0
    RET
 //------- Example5--------//
Example5:
   //PE3设为输入， 输入上拉，下降沿感知，DOWN键按下计数
    LDI R16,PORT_ISC_FALLING_gc|PORT_OPC_PULLUP_gc
    STS PORTE_PIN3CTRL,R16
    LDI R16,0X08
    STS PORTE_DIRCLR,R16
    LDI R16,0X30 //PD4设为输出
    STS PORTD_DIRSET,R16
    LDI R16,EVSYS_CHMUX_PORTE_PIN3_gc //选择PE3为事件通道0的输入，使能数字滤波
    STS EVSYS_CH0MUX,R16
    LDI R16,EVSYS_DIGFILT_8SAMPLES_gc
    STS EVSYS_CH0CTRL,R16
    LDI XL,0X04 //设置计数周期值-TOP
```

```
    EOR XH,XH
    STS TCC0_PER,XL
    STS TCC0_PER+1,XH
    LDI R16,USART_RXCINTLVL_LO_gc//设置溢出中断为低级别中断
    STS USARTC0_CTRLA,R16
    LDI R16,PMIC_LOLVLEN_bm
    STS PMIC_CTRL,R16
    SEI
    LDI R16,TC_CLKSEL_EVCH0_gc //启动定时器
    STS TCC0_CTRLA,R16
Example5_0:
    JMP Example5_0
    RET
ISR_OVFIF:
    LED1_T 0X20
    LED2_T 0X10
    RETI
  //------- Example6--------//
Example6:
    LDI R16,0x10//PD4设为输出
    STS PORTD_DIRSET,R16
    LDI R16,EVSYS_CHMUX_TCC0_OVF_gc//TCC0溢出作为事件通道0的输入
    STS EVSYS_CH0MUX,R16
    LDS R16,TCC0_CTRLD//使能TCC1传播时延
    ORI R16,TC0_EVDLY_bm
    STS TCC0_CTRLD,R16
    LDI XL,0X0E2 //设置计数周期
    LDI XH,0X04
    STS TCC0_PER,XL
    STS TCC0_PER+1,XH
    LDI XL,0X0C8
    EOR XH,XH
    STS TCC1_PER,XL
    STS TCC1_PER+1,XH
    LDI R16,TC_CLKSEL_EVCH0_gc //使用通道0作为TCC1时钟源
    STS TCC1_CTRLA,R16
    LDI R16,TC_CLKSEL_DIV8_gc//使用外设时钟8分频作为TCC0时钟源 启动定时器
    STS TCC0_CTRLA,R16
Example6_1:
    LDS R16,TCC1_INTFLAGS
    SBRS R16,TC1_OVFIF_bp
    JMP Example6_1
    NOP
    LED1_T 0X10//取反PD4
```

```
    LDI R16,TC1_OVFIF_bm//清除溢出标志
    STS TCC1_INTFLAGS,R16
    RET
//-------RESET--------//
RESET:
     CALL uart_init
     //CALL Example1
     //CALL Example2
     //CALL Example3
     CALL Example6
     //CALL Example5
     //CALL Example6
```

2. TCC0通道B占空比按照正弦样本值变化，PC1输出波形通过二阶或多阶低通滤波器输出可获得较平滑正弦曲线,频率越低，样点越密，输出波形越平滑。

C语言代码：

```
//-------包含头文件--------//
#include "avr_compiler.h"
#include "TC_driver.c"

#define LED1_T()   PORTD_OUTTGL = 0x20
unsigned char SineWaveTable128[128]  = {
128,134,140,147,153,159,165,171,177,182,188,193,199,204,209,213,
218,222,226,230,234,237,240,243,245,248,250,251,253,254,254,255,
255,255,254,254,253,251,250,248,245,243,240,237,234,230,226,222,
218,213,209,204,199,193,188,182,177,171,165,159,153,147,140,134,
128,122,116,109,103,97,91,85,79,74,68,63,57,52,47,43, 38,34,30,
26,22,19,16,13,11,8,6,5,3,2,2,1, 1,1,2,2,3,5,6,8,11,13,16,19,22
,26,30,34, 38,43,47,52,57,63,68,74,79,85,91,97,103,109,116,122}; //128点正弦波样本值

uint16_t compareValue = 0x0000;
//-------main（主函数）--------//
int main(void)
{
   PORTD_DIR = 0x20;//PD5方向设为输出
   PORTC.DIRSET = 0x02; //PC1输出
   TC_SetPeriod( &TCC0, 255 ); //设置计数周期
   TC0_ConfigWGM( &TCC0, TC_WGMODE_SS_gc ); //设置TC为单斜率模式
   TC0_EnableCCChannels( &TCC0, TC0_CCBEN_bm ); // 使能通道B
   TC0_SetOverflowIntLevel( &TCC0, TC_OVFINTLVL_LO_gc ); //设置溢出中断为低级别中断
   PMIC.CTRL |= PMIC_LOLVLEN_bm;
   sei();
   TC0_ConfigClockSource( &TCC0, TC_CLKSEL_DIV64_gc ); //选择时钟，启动定时器
```

```
   do {
    //溢出时比较值从CCBBUF[H:L] 传递到CCB[H:L]
      } while (1);
}
//------- ISR (TCC0溢出中断)--------//
ISR(TCC0_OVF_vect)
{
   LED1_T();//溢出指示灯
   compareValue ++; // 新比较值
   if(compareValue>128){compareValue = 0; }
   TC_SetCompareB( &TCC0, SineWaveTable128[compareValue] ); //设置到缓冲寄存器
  //溢出时比较值从CCBBUF[H:L] 传递到CCB[H:L]
}
```

汇编代码：

```
//------- 包含头文件--------//
.include "ATxmega128A1def.inc"//器件配置文件,决不可少,不然汇编通不过
.ORG 0
        RJMP RESET
.ORG 0X01C
        RJMP ISR_TCC0_OVF_vect
.ORG 0X100       //跳过中断区0x00-0x0FF
//------- 宏定义--------//
.MACRO LED1_T
           LDI R16,@0
           STS PORTD_OUTTGL,R16
.ENDMACRO
SineWaveTable128: .DB
128,134,140,147,153,159,165,171,177,182,188,193,199,204,209,213,218,222,226,230,
234,237,240,243,245,248,250,251,253,254,254,255,255,255,254,254,253,251,250,248,
245,243,240,237,234,230,226,222,218,213,209,204,199,193,188,182,177,171,165,159,
153,147,140,134,128,122,116,109,103,97,91,85,79,74,68,63,57,52,47,43,38,34,30,2
6,22,19,16,13,11,8,6,5,3,2,2,1,1,1,2,2,3,5,6,8,11,13,16,19,22,26,30,34,38,43,47,
52,57,63,68,74,79,85,91,97,103,109,116,122 // 128点正弦波样本值
 //-------RESET(主函数入口)--------//
RESET:
    LDI R16,0X20
    STS PORTD_DIR,R16//PD5方向设为输出
    LDI R16,0X02 //PC1输出
    STS PORTC_DIRSET,R16
    LDI XL,0X0FF//设置计数周期
    EOR XH,XH
    STS TCC0_PER,XL
    STS TCC0_PER+1,XH
    LDS R16,TCC0_CTRLB//设置TC为单斜率模式
```

```
    ORI R16,TC_WGMODE_SS_gc
    STS TCC0_CTRLB,R16
    LDI R16,TC0_CCBEN_bm //使能通道B
    STS TCC0_CTRLB,R16
    LDI R16,USART_RXCINTLVL_LO_gc//设置溢出中断为低级别中断
    STS USARTC0_CTRLA,R16
    /* 使能中断*/
    LDI R16,PMIC_LOLVLEN_bm
    STS PMIC_CTRL,R16
    SEI
    //选择时钟，启动定时器 溢出时比较值从CCBBUF[H:L] 传递到CCB[H:L]
    LDI R16,TC_CLKSEL_DIV64_gc
    STS TCC0_CTRLA,R16
    LDI ZH,HIGH(SineWaveTable128<<1)
    LDI ZL,LOW(SineWaveTable128<<1)
    EOR XH,XH;计数用
RESET_0:
    JMP  RESET_0
//------ ISR_TCC0_OVF_vect(TCC0溢出中断)--------//
ISR_TCC0_OVF_vect:
    LED1_T 0X20
    INC XH //新比较值
    SBRS XH,7
    JMP ISR_TCC0_OVF_vect_0
    NOP
    EOR XH,XH
ISR_TCC0_OVF_vect_0:
    //设置到缓冲寄存器 溢出时比较值从CCBBUF[H:L] 传递到CCB[H:L]
     LPM R16,Z+
     EOR R17,R17
     STS TCC0_CCBBUF,R16
     STS TCC0_CCBBUF+1,R17
     RETI
```

5.5 ADC 实例

ADC 将模拟电压转换为数字量。ADC 有 12 位精度，可以在每秒钟最高 2 百万次抽样。输入选择很灵活，可以是单端或是差分。差分输入可以通过增益来增加动态范围。另外可以选择一些内部输入信号。ADC 支持有符号和无符号转换。

ADC 是流水转换的，包含多个连续的阶段，每个阶段转换一部分结果。流水转换设计使的在低速时钟下也可以高抽样率，并消除抽样速率和传播时延的相关性。

ADC 测量可以从程序或设备的其他外设事件启动。4 个输入选择（MUX）以及结果寄存器使得程序可以轻松的获取数据。每个结果寄存器和 MUX 选择组成一个 ADC 通道。可以使用

DMA 直接将 ADC 结果转存到存储器或外设。

可以使用内部或者外部参考电压。内部提供一个精确的 1.00V 参考电压。

内部集成温度传感器，它的输出可以由 ADC 测量。DAC，VCC/ 10 和带隙电压都可使用 ADC 测量。

1. ADC将模拟电压转换为数字量。程序设置模拟电压转换器ADCA工作在有符号，12位分辨率的差分模式下，分别用查询标志位和中断函数两种方式来获得转换结果。四个通道的转换结果记录到数组adcSamples[4][10]中，USARTC0打印调试信息。

C 语言代码：

```
//------ 包含头文件--------//
#include "avr_compiler.h"
#include "usart_driver.c"
#include "adc_driver.c"
#define SAMPLE_COUNT 10
int16_t adcSamples[4][SAMPLE_COUNT];
uint16_t interrupt_count = 0;
int8_t offset;
//------ uart_init (串口初始化)--------//
void uart_init(void)
{
    /* USARTC0 引脚方向设置*/
    PORTC.DIRSET   = PIN3_bm; //PC3 (TXD0) 输出
    PORTC.DIRCLR   = PIN2_bm; //PC2 (RXD0) 输入
    USART_SetMode(&USARTC0,USART_CMODE_ASYNCHRONOUS_gc); //USARTC0 模式 - 异步
    //USARTC0帧结构, 8 位数据位, 无校验, 1停止位
    USART_Format_Set(&USARTC0, USART_CHSIZE_8BIT_gc,USART_PMODE_DISABLED_gc,
false);
    USART_Baudrate_Set(&USARTC0, 12 , 0); //设置波特率 9600
    USART_Tx_Enable(&USARTC0); //USARTC0 使能发送
    USART_Rx_Enable(&USARTC0); //USARTC0 使能接收
}
//------ ADCA_CH0_1_2_3_Sweep_Interrupt --------//
void ADCA_CH0_1_2_3_Sweep_Interrupt(void)
{
    uart_puts("inside ADCA_CH0_1_2_3_Sweep_Interrupt");
    uart_putc('\n');
    ADC_CalibrationValues_Load(&ADCA); // 加载校准值
    ADC_ConvMode_and_Resolution_Config(&ADCA, ADC_ConvMode_Signed,
    ADC_RESOLUTION_12BIT_gc); // 设置 ADC A 有符号模式 12 位分辨率
    ADC_Prescaler_Config(&ADCA, ADC_PRESCALER_DIV32_gc); // 设置ADC分频
    ADC_Reference_Config(&ADCA, ADC_REFSEL_INT1V_gc); // 设置参考电压 VCC/1.6 V
    ADC_Ch_InputMode_and_Gain_Config(&ADCA.CH0,ADC_CH_INPUTMODE_DIFF_gc,ADC_DRI
    VER_CH_GAIN_NONE); //ADC A，差分输入，无增益
    ADC_Ch_InputMux_Config(&ADCA.CH0, ADC_CH_MUXPOS_PIN0_gc,
ADC_CH_MUXNEG_PIN0_gc);
```

```
    ADC_Enable(&ADCA);
    ADC_Wait_8MHz(&ADCA);
    offset = ADC_Offset_Get_Signed(&ADCA, &ADCA.CH0, false); //补偿值
    uart_puts(" ADC offset = ");
    uart_putc_hex(offset);
    uart_putc('\n');
    ADC_Disable(&ADCA);
    ADC_Ch_InputMode_and_Gain_Config(&ADCA.CH0,
                                     ADC_CH_INPUTMODE_SINGLEENDED_gc,
                                     ADC_DRIVER_CH_GAIN_NONE);
    ADC_Ch_InputMode_and_Gain_Config(&ADCA.CH1,
                                     ADC_CH_INPUTMODE_SINGLEENDED_gc,
                                     ADC_DRIVER_CH_GAIN_NONE);
    ADC_Ch_InputMode_and_Gain_Config(&ADCA.CH2,
                                     ADC_CH_INPUTMODE_SINGLEENDED_gc,
                                     ADC_DRIVER_CH_GAIN_NONE);
    ADC_Ch_InputMode_and_Gain_Config(&ADCA.CH3,
                                     ADC_CH_INPUTMODE_SINGLEENDED_gc,
                                     ADC_DRIVER_CH_GAIN_NONE);
    ADC_Ch_InputMux_Config(&ADCA.CH0, ADC_CH_MUXPOS_PIN0_gc, 0);
    ADC_Ch_InputMux_Config(&ADCA.CH1, ADC_CH_MUXPOS_PIN2_gc, 0);

    ADC_Ch_InputMux_Config(&ADCA.CH2, ADC_CH_MUXPOS_PIN4_gc, 0);
    ADC_Ch_InputMux_Config(&ADCA.CH3, ADC_CH_MUXPOS_PIN6_gc, 0);
    ADC_SweepChannels_Config(&ADCA, ADC_SWEEP_0123_gc);
    ADC_Ch_Interrupts_Config(&ADCA.CH2, ADC_CH_INTMODE_COMPLETE_gc,
    ADC_CH_INTLVL_LO_gc);
    PMIC.CTRL |= PMIC_LOLVLEX_bm;
    sei();
    ADC_Enable(&ADCA);
    ADC_Wait_8MHz(&ADCA);
    ADC_FreeRunning_Enable(&ADCA);//使能自由运行模式
}
//------ ADCA_CH0_1_2_3_Poll --------//
void ADCA_CH0_1_2_3_Poll(void)
{
    uart_puts("inside ADCA_CH0_1_2_3_Poll");
    uart_putc('\n');
    ADC_CalibrationValues_Load(&ADCA);
    ADC_ConvMode_and_Resolution_Config(&ADCA, ADC_ConvMode_Signed,
    ADC_RESOLUTION_12BIT_gc);
    ADC_Prescaler_Config(&ADCA, ADC_PRESCALER_DIV32_gc);
    ADC_Reference_Config(&ADCA, ADC_REFSEL_VCC_gc);
    ADC_Ch_InputMode_and_Gain_Config(&ADCA.CH0,
```

```
                                    ADC_CH_INPUTMODE_DIFF_gc,
                                    ADC_DRIVER_CH_GAIN_NONE);
  ADC_Ch_InputMode_and_Gain_Config(&ADCA.CH1,
                                    ADC_CH_INPUTMODE_INTERNAL_gc,
                                    ADC_DRIVER_CH_GAIN_NONE);
  ADC_Ch_InputMode_and_Gain_Config(&ADCA.CH2,
                                    ADC_CH_INPUTMODE_SINGLEENDED_gc,
                                    ADC_DRIVER_CH_GAIN_NONE);
  ADC_Ch_InputMode_and_Gain_Config(&ADCA.CH3,
                                    ADC_CH_INPUTMODE_SINGLEENDED_gc,
                                    ADC_DRIVER_CH_GAIN_NONE);
  ADC_Ch_InputMux_Config(&ADCA.CH0, ADC_CH_MUXPOS_PIN0_gc,
  ADC_CH_MUXNEG_PIN0_gc);
  ADC_Enable(&ADCA);
  ADC_Wait_8MHz(&ADCA);
  offset = ADC_Offset_Get_Signed(&ADCA, &ADCA.CH0, false);
  uart_puts(" ADC offset = ");
  uart_putc_hex(offset);
  uart_putc('\n');
  ADC_Disable(&ADCA);
  ADC_Ch_InputMux_Config(&ADCA.CH0, ADC_CH_MUXPOS_PIN0_gc,
ADC_CH_MUXNEG_PIN2_gc);
  ADC_Ch_InputMux_Config(&ADCA.CH1, ADC_CH_MUXINT_SCALEDVCC_gc, 0);
  ADC_Ch_InputMux_Config(&ADCA.CH2, ADC_CH_MUXPOS_PIN0_gc, 0);
  ADC_Ch_InputMux_Config(&ADCA.CH3, ADC_CH_MUXPOS_PIN2_gc, 0);
  ADC_SweepChannels_Config(&ADCA, ADC_SWEEP_0123_gc);//多通道转换
  ADC_Enable(&ADCA);//使能ADCA转换
  ADC_Wait_8MHz(&ADCA);
  ADC_FreeRunning_Enable(&ADCA);//使能自由运行模式
  for (uint16_t i = 0; i < SAMPLE_COUNT; ++i) {
      do{}while(!ADC_Ch_Conversion_Complete(&ADCA.CH0));
      adcSamples[0][i] = ADC_ResultCh_GetWord_Signed(&ADCA.CH0, offset);
       do{}while(!ADC_Ch_Conversion_Complete(&ADCA.CH1));
      adcSamples[1][i] = ADC_ResultCh_GetWord_Signed(&ADCA.CH1, offset);
      do{}while(!ADC_Ch_Conversion_Complete(&ADCA.CH2));
      adcSamples[2][i] = ADC_ResultCh_GetWord_Signed(&ADCA.CH2, offset);
       do{}while(!ADC_Ch_Conversion_Complete(&ADCA.CH3));
      adcSamples[3][i] = ADC_ResultCh_GetWord_Signed(&ADCA.CH3, offset);
  }
  ADC_FreeRunning_Disable(&ADCA);
  ADC_Disable(&ADCA);
  uart_puts(" ADCA.CH0 = ");uart_putc('\n');
  for (uint16_t i = 0; i < 10; ++i) {
      uart_putw_dec(adcSamples[0][i]);uart_puts(",");
```

```
        }uart_putc('\n');
        uart_puts(" ADCA.CH1 = ");uart_putc('\n');
    for (uint16_t i = 0; i < 10; ++i) {
        uart_putw_dec(adcSamples[1][i]);uart_puts(",");
        }uart_putc('\n');
        uart_puts(" ADCA.CH2 = ");uart_putc('\n');
    for (uint16_t i = 0; i < 10; ++i) {
        uart_putw_dec(adcSamples[2][i]);uart_puts(",");
        }uart_putc('\n');
        uart_puts(" ADCA.CH3 = ");uart_putc('\n');
    for (uint16_t i = 0; i < 10; ++i) {
        uart_putw_dec(adcSamples[3][i]);uart_puts(",");
        }uart_putc('\n');
}
 //------ main（主函数入口）--------//
int main(void)
{
    uart_init();
    ADCA_CH0_1_2_3_Sweep_Interrupt();
    //ADCA_CH0_1_2_3_Poll();
    while(true){}
}
//------ ISR（ADCA通道2中断） --------//
ISR(ADCA_CH2_vect)
{
    if(interrupt_count == 0)
    {
        uart_puts("inside Interrupt routine interrupt_count = 0");
        uart_putc('\n');
    }
    //读取转换结果清除标志位
    adcSamples[0][interrupt_count] = ADC_ResultCh_GetWord_Signed(&ADCA.CH0,
offset);
    adcSamples[1][interrupt_count] = ADC_ResultCh_GetWord_Signed(&ADCA.CH1,
offset);
    adcSamples[2][interrupt_count] = ADC_ResultCh_GetWord_Signed(&ADCA.CH2,
offset);
    adcSamples[3][interrupt_count] = ADC_ResultCh_GetWord_Signed(&ADCA.CH3,
offset);
    if(interrupt_count == SAMPLE_COUNT-1)
    {
         ADC_Ch_Interrupts_Config(&ADCA.CH3,ADC_CH_INTMODE_COMPLETE_gc,ADC_CH_I
          NTLVL_OFF_gc);
         ADC_FreeRunning_Disable(&ADCA);
```

```
         ADC_Disable(&ADCA);
         uart_puts("inside Interrupt routine interrupt_count = 99
");uart_putc('\n');
         uart_puts(" ADCA.CH0 = ");uart_putc('\n');
        for (uint16_t i = 0; i < 10; ++i) {
            uart_putw_dec(adcSamples[0][i]);uart_puts(",");
            }uart_putc('\n');
            uart_puts(" ADCA.CH1 = ");uart_putc('\n');
        for (uint16_t i = 0; i < 10; ++i) {
            uart_putw_dec(adcSamples[1][i]);uart_puts(",");
            }uart_putc('\n');
            uart_puts(" ADCA.CH2 = ");uart_putc('\n');
        for (uint16_t i = 0; i < 10; ++i) {
            uart_putw_dec(adcSamples[2][i]);uart_puts(",");
            }uart_putc('\n');
            uart_puts(" ADCA.CH3 = ");uart_putc('\n');
        for (uint16_t i = 0; i < 10; ++i) {
            uart_putw_dec(adcSamples[3][i]);uart_puts(",");
            }uart_putc('\n');
    }
    interrupt_count++;
}
```

汇编代码：

```
//------ 包含头文件--------//
.include "ATxmega128A1def.inc";器件配置文件,决不可少,不然汇编通不过
.include "usart_driver.inc"
.ORG 0
        RJMP RESET//复位
.ORG 0X094
        RJMP ISR_ADCA_CH3_vect
.ORG 0X100       ;跳过中断区0x00-0x0FF
.EQU ENTER  ='\n'
.EQU NEWLINE='\r'
.EQU EQUE   ='='
.EQU ZERO   ='0'
.EQU A_ASSCI='A'
STRING: .DB 'A','D','C','A','_','0','F','F','S','E','T','=',0
STRING0: .DB 'C','H','A','N','E','L','_','0','=',0
STRING1: .DB 'C','H','A','N','E','L','_','1','=',0
STRING2: .DB 'C','H','A','N','E','L','_','2','=',0
STRING3: .DB 'C','H','A','N','E','L','_','3','=',0
 //------ uart_init（串口初始化）--------//
uart_init:
        /* USARTC0 引脚方向设置*/
```

```
        LDI R16,0X08 //PC3 (TXD0) 输出
        STS PORTC_DIRSET,R16
        LDI R16,0X04//PC2 (RXD0) 输入
        STS PORTC_DIRCLR,R16
        // USARTC0 模式 - 异步 USARTC0帧结构, 8 位数据位, 无校验, 1停止位
         LDI  R16,USART_CMODE_ASYNCHRONOUS_gc|USART_CHSIZE_8BIT_gc
                                             |USART_PMODE_DISABLED_gc
        STS USARTC0_CTRLC,R16
        LDI R16,12//设置波特率 9600
        STS USARTC0_BAUDCTRLA,R16
        LDI R16,0
        STS USARTC0_BAUDCTRLB,R16
        LDI R16,USART_TXEN_bm//USARTC0 使能发送  USARTC0 使能接收
        STS USARTC0_CTRLB,R16
        RET
ADCA_CH0_1_2_3_Sweep_Interrupt:
        // 加载校准值
        LDI ZL,PROD_SIGNATURES_START + NVM_PROD_SIGNATURES_ADCACAL0_offset
        EOR ZH,ZH
        LDI R16,NVM_CMD_READ_CALIB_ROW_gc
        STS NVM_CMD,R16
        LPM R17,Z+
        LPM R18,Z
        LDI R16,NVM_CMD_NO_OPERATION_gc
        STS NVM_CMD,R16
        STS ADCA_CAL,R17
        STS ADCA_CAL+1,R18
         // 设置 ADC A 有符号模式 12 位分辨率
         LDI R16,ADC_CONMODE_bm|ADC_RESOLUTION_12BIT_gc
         STS ADCA_CTRLB,R16
         LDI R16,ADC_PRESCALER_DIV32_gc// 设置ADC分频
         STS ADCA_PRESCALER,R16
         LDI R16,ADC_REFSEL_INT1V_gc// 设置参考电压 VCC/1.6 V
        STS ADCA_REFCTRL,R16
         //ADC A，差分输入，无增益
         LDI R16,ADC_CH_INPUTMODE_DIFF_gc|ADC_CH_GAIN_1X_gc
         STS ADCA_CH0_CTRL,R16
         LDI R16,ADC_CH_MUXPOS_PIN0_gc|ADC_CH_MUXNEG_PIN0_gc
         STS ADCA_CH0_MUXCTRL,R16
         LDI R16,ADC_ENABLE_bm|ADC_CH0START_bm//启动ADC  转换使能
        STS ADCA_CTRLA,R16
         CALL _delay_ms
          uart_puts_string STRING
          LDS R16,ADCA_CH0_INTFLAGS
```

```
ADCA_CH0_1_2_3_Sweep_Interrupt_0:
          SBRS R16,ADC_CH_CHIF_bp
           JMP ADCA_CH0_1_2_3_Sweep_Interrupt_0
           NOP
           LDI R16,ADC_CH_CHIF_bm
           STS ADCA_CH0_INTFLAGS,R16
           LDS XL,ADCA_CH0RES
           LDS XH,ADCA_CH0RES+1
           uart_putw_dec
           LDI R16,0X00//关闭ADC
           STS ADCA_CTRLA,R16
          //设置通道0,1,2,3单端输入 且ADC增益因子为1
           LDI R16,ADC_CH_INPUTMODE_SINGLEENDED_gc|ADC_CH_GAIN_1X_gc
           STS ADCA_CH0_CTRL,R16
           LDI R16,ADC_CH_INPUTMODE_SINGLEENDED_gc|ADC_CH_GAIN_1X_gc
           STS ADCA_CH1_CTRL,R16
           LDI R16,ADC_CH_INPUTMODE_SINGLEENDED_gc|ADC_CH_GAIN_1X_gc
           STS ADCA_CH2_CTRL,R16
           LDI R16,ADC_CH_INPUTMODE_SINGLEENDED_gc|ADC_CH_GAIN_1X_gc
           STS ADCA_CH3_CTRL,R16
           //设置ADC各个通道的输入引脚
           LDI R16,ADC_CH_MUXPOS_PIN0_gc
           STS ADCA_CH0_MUXCTRL,R16
           LDI R16,ADC_CH_MUXPOS_PIN2_gc
           STS ADCA_CH1_MUXCTRL,R16
           LDI R16,ADC_CH_MUXPOS_PIN4_gc
           STS ADCA_CH2_MUXCTRL,R16
           LDI R16,ADC_CH_MUXPOS_PIN6_gc
           STS ADCA_CH3_MUXCTRL,R16
            LDI R16,ADC_SWEEP_0123_gc //设置扫描通道
           STS ADCA_EVCTRL,R16
           LDI R16,ADC_CH_INTMODE_COMPLETE_gc|ADC_CH_INTLVL_LO_gc
           STS ADCA_CH3_INTCTRL,R16
            LDI R16,PMIC_LOLVLEX_bm
            STS PMIC_CTRL,R16
            SEI
            LDI R16,ADC_ENABLE_bm
            STS ADCA_CTRLA,R16
            CALL _delay_ms
            LDI R16,ADC_FREERUN_bm//使能自由运行模式
            STS ADCA_CTRLB,R16
            RET
//------ RESET(主函数入口)--------//
RESET:
```

```
            CLR  R25
            CALL uart_init
            CALL ADCA_CH0_1_2_3_Sweep_Interrupt
RESET_0:
            JMP RESET_0
 //------ ISR_ADCA_CH3_vect (ADCA通道3中断)--------//
ISR_ADCA_CH3_vect:
             INC R25
             CPI R25,5
             BRNE ISR_ADCA_CH3_vect_1
             LDI R16,0X00
             STS PMIC_CTRL,R16
             LDI R16,0X00
             STS ADCA_CTRLB,R16
ISR_ADCA_CH3_vect_1:
         LDI R16,ADC_CH_CHIF_bm
         STS ADCA_CH0_INTFLAGS,R16
         STS ADCA_CH1_INTFLAGS,R16
         STS ADCA_CH2_INTFLAGS,R16
         LDI R16,ENTER
         PUSH R16
         uart_putc
         uart_puts_string STRING0
         LDS XL,ADCA_CH0RES
         LDS XH,ADCA_CH0RES+1
         uart_putw_dec
         LDI R16,ENTER
         PUSH R16
         uart_putc
         uart_puts_string STRING1
         LDS XL,ADCA_CH1RES
         LDS XH,ADCA_CH1RES+1
         uart_putw_dec
         LDI R16,ENTER
         PUSH R16
         uart_putc
         uart_puts_string STRING2
         LDS XL,ADCA_CH2RES
         LDS XH,ADCA_CH2RES+1
         uart_putw_dec
         LDI R16,ENTER
         PUSH R16
         uart_putc
         uart_puts_string STRING3
```

```
        LDS XL,ADCA_CH3RES
        LDS XH,ADCA_CH3RES+1
        uart_putw_dec
        LDI R16,ENTER
        PUSH R16
        uart_putc
        RETI
//------ _delay_ms (延时函数)--------//
_delay_ms:
        LDI R17,0xff
L0:     LDI R18,0xff
L1:     DEC R18
        BRNE L1
        DEC R17
        BRNE L0
        RET
```

5.6 I2C 实例

两线接口（TWI）是双向的 2 线总线通信，兼容 I^2C 和 SMBus。

连接到总线的设备必须作为一个主机或从机。主机通过总线向从机发送地址启动数据传输，并告诉从机主机是否发送或接收数据。一个总线可以有几个主机，如果两个或更多个主机尝试在同一时间传送数据，将由仲裁程序处理优先级。

在 XMEGA 中，TWI 模块同时具备主机和从机的功能。主机和从机功能上是独立的，可以单独启用。它们有独立的控制寄存器和状态寄存器以及中断向量。硬件可以检测到仲裁丢失，错误，总线上的碰撞和时钟保持，并且主从机各自的状态寄存器中指出。

两线接口(TWI)是两根双向总线它包括一个串行时钟线(SCL)和一个串行数据线(SDA)。这两条线是集电极开路的（线与），驱动总线的唯一外部元件是上拉电阻（Rp）。当没有 TWI 器件连接到总线时，上拉电阻将提供给总线一个高电平。一个恒流源可以选择作为上拉电阻。

1.TWIC自发自收。使能TWIC主机和从机，主程序中对发送缓存sendBuffer中的八位数据在D端口LED上面显示持续时间1s，然后TWIC主机发送数据给TWIC从机，TWIC从机接收数据并返还发送给TWIC主机，TWIC主机收到数据放到readdata数组中，sendBuffer中的数据与readdata中数据应该相同，否则传输出错。

C语言代码：

```
//------包含头文件--------//
#include "avr_compiler.h"
#include <util/delay.h>
#include "twi_master_driver.c"
#include "twi_slave_driver.c"
//从机地址
#define SLAVE_ADDRESS    0x55
//缓冲字节数
#define NUM_BYTES      6
```

```
// CPU 2MHz
#define CPU_SPEED   2000000
// 波特率100kHz
#define BAUDRATE    100000
#define TWI_BAUDSETTING TWI_BAUD(CPU_SPEED, BAUDRATE)
//结构体变量
TWI_Master_t twiMaster;    //TWI 主机
TWI_Slave_t twiSlave;      //TWI 从机
//测试数据
uint8_t sendBuffer[NUM_BYTES] = {0x01,0x02,0x03,0x04,0x05,0x0F};
void TWIC_SlaveProcessData(void)
{
    uint8_t bufIndex = twiSlave.bytesReceived;
    twiSlave.sendData[bufIndex] = (~twiSlave.receivedData[bufIndex]);
}
//-----mian（主函数入口）--------//
int main(void)
{
    // PORTD 设为输出 LED 显示数据
    PORTD.DIRSET = 0xFF;
    // 如果外部没有接上拉电阻，使能上拉 PC0(TWI-SDA), PC1(TWI-SCL)
    PORTCFG.MPCMASK = 0x03; // 一次配置多个引脚
    PORTC.PIN0CTRL = (PORTC.PIN0CTRL & ~PORT_OPC_gm) | PORT_OPC_PULLUP_gc;
    //初始化主机
    TWI_MasterInit(&twiMaster,
                   &TWIC,
                   TWI_MASTER_INTLVL_LO_gc,
                   TWI_BAUDSETTING);
    //初始化从机
    TWI_SlaveInitializeDriver(&twiSlave, &TWIC, TWIC_SlaveProcessData);
    TWI_SlaveInitializeModule(&twiSlave,
                              SLAVE_ADDRESS,
                              TWI_SLAVE_INTLVL_LO_gc);
    //使能低级别中断
    PMIC.CTRL |= PMIC_LOLVLEN_bm;
    sei();
    uint8_t BufPos = 0;
    while (1)
    {
        PORTD.OUT =0;
        for(BufPos=0;BufPos<6;BufPos++)
        {
            //LED 显示数据
            PORTD.OUT = sendBuffer[BufPos];
```

```
            _delay_ms(1000);
            //主机发送数据

    TWI_MasterWriteRead(&twiMaster,SLAVE_ADDRESS,&sendBuffer[BufPos],1,1);
            while (twiMaster.status != TWIM_STATUS_READY)
            {
                //等待传输完成
            }
            //LED 显示数据(已经取反)
            PORTD.OUT = (twiMaster.readData[0]);
            _delay_ms(1000);
        }
    }
}
//----- ISR（主机中断函数入口）--------//
ISR(TWIC_TWIM_vect)
{
    TWI_MasterInterruptHandler(&twiMaster);
}
 //----- ISR（从机中断函数入口）--------//
ISR(TWIC_TWIS_vect)
{
    TWI_SlaveInterruptHandler(&twiSlave);
}
```

汇编代码：

```
//----- 包含头文件--------//
.include "ATxmega128A1def.inc";器件配置文件,决不可少,不然汇编通不过
.include "usart_driver.inc"
.ORG 0
        RJMP RESET//复位
;.ORG 0x036        //USARTC0数据接收完毕中断入口
        ;RJMP ISR
.ORG  0X01A
        RJMP ISRT_TWIC_TWIM_vect
.ORG  0X018
        RJMP ISR_TWIC_TWIS_vect
.ORG 0X100       ;跳过中断区0x00-0x0FF
.EQU ENTER  ='\n'
.EQU NEWLINE='\r'
.EQU EQUE   ='='
.EQU ZERO   ='0'
.EQU A_ASSCI='A'
//从机地址
.equ SLAVE_ADDRESS=0x55
```

```
// CPU 2MHz
// 波特率400kHz
.equ TWI_BAUDSETTING=0
//测试数据
sendBuffer: .DB 0x01,0x02,0x03,0x04,0xbb,0xaa,0xdd,0xee,0x00
MAIN:       .DB 'I','N','T','O','M','A','I','N',0
 //----- uart_init （串口初始化）--------//
uart_init:
       /* USARTC0 引脚方向设置*/
         LDI R16,0X08 //PC3 (TXD0) 输出
         STS PORTC_DIRSET,R16
         LDI R16,0X04 //PC2 (RXD0) 输入
         STS PORTC_DIRCLR,R16
        //USARTC0 模式 - 异步  USARTC0帧结构, 8 位数据位, 无校验, 1停止位
         LDI R16,USART_CMODE_ASYNCHRONOUS_gc|USART_CHSIZE_8BIT_gc
                                            |USART_PMODE_DISABLED_gc
         STS USARTC0_CTRLC,R16
         LDI R16,12//设置波特率 9600
         STS USARTC0_BAUDCTRLA,R16
         LDI R16,0
         STS USARTC0_BAUDCTRLB,R16
         LDI R16,USART_TXEN_bm//USARTC0 使能发送  USARTC0 使能接收
         STS USARTC0_CTRLB,R16
          RET
//----- RESET(主函数入口)--------//
RESET:
         CALL uart_init
         LDI R16,ENTER
         PUSH R16
         uart_putc
         uart_puts_string MAIN
         LDI R16,ENTER
         PUSH R16
         uart_putc
         LDI R16,0X03
         STS PORTD_DIRSET,R16
        // 如果外部没有接上拉电阻，使能上拉 PC0(TWI-SDA), PC1(TWI-SCL)
         LDI R16,0X03
         STS PORTCFG_MPCMASK,R16 // 一次配置多个引脚
         LDI R16,PORT_OPC_PULLUP_gc
         STS PORTC_PIN0CTRL,R16
         LDI R16,TWI_BAUDSETTING
         STS TWIC_MASTER_BAUD,R16
         LDI R16,TWI_MASTER_INTLVL_LO_gc|TWI_MASTER_RIEN_bm |TWI_MASTER_WIEN_bm
```

```
|TWI_MASTER_ENABLE_bm
        STS TWIC_MASTER_CTRLA,R16
        ldi r16,TWI_MASTER_BUSSTATE_IDLE_gc
        sts twic_MASTER_STATUS,r16
         //初始化从机
        LDI R16,TWI_SLAVE_INTLVL_LO_gc |TWI_SLAVE_ENABLE_bm|TWI_SLAVE_DIEN_bm
                                       |TWI_SLAVE_APIEN_bm;|TWI_SLAVE_PIEN_bm

        STS TWIC_SLAVE_CTRLA,R16
        LDI R16,SLAVE_ADDRESS<<1
        STS TWIC_SLAVE_ADDR,R16
       //使能低级别中断
        LDI R16,PMIC_LOLVLEN_bm
        STS PMIC_CTRL,R16
        SEI
        //主机发送数据
        // 根据bytesToWrite判断是写, 发送START信号 + 7位地址 +  R/_W = 0
        LDI R16,SLAVE_ADDRESS<<1&0x0fe
        STS TWIC_MASTER_ADDR,R16
        ldi xh,0x21
        ldi xl,0x00
        ldi r16,0x00
        st x+,r16
        st x,r16//计数
RESET_2:
        JMP RESET_2
//----- ISRT_TWIC_TWIM_vect（主机中断）--------//
ISRT_TWIC_TWIM_vect:
          LDS  R16,TWIC_MASTER_STATUS
          SBRS R16,TWI_MASTER_WIF_bp
          JMP ISRT_TWIC_TWIM_vect_2//读中断
          nop
          sbrs r16,4
          jmp ISRT_TWIC_TWIM_vect_0
          nop
      // 根据bytesToWrite判断是读,以后交给读中断 发送START信号 + 7位地址 +
 R/_W = 1
          LDI R16,SLAVE_ADDRESS<<1|0X01
          STS TWIC_MASTER_ADDR,R16
          jmp ISRT_TWIC_TWIM_vect_1
          nop
ISRT_TWIC_TWIM_vect_0://写中断
         LDI ZH,HIGH(sendBuffer<<1)
         LDI ZL,LOW(sendBuffer<<1)
```

```
          ldi xh,0x21
          ldi xl,0x01
          ld r19,x
          add zl,r19
          lpm r16,z+
          inc r19
          st x,r19
          clz
          cpi r16,0x00
          breq ISRT_TWIC_TWIM_vect_1
          STS TWIC_MASTER_DATA,R16
          jmp ISRT_TWIC_TWIM_vect_3
ISRT_TWIC_TWIM_vect_1:
          STS TWIC_MASTER_DATA,R16
          lds r16,TWIC_MASTER_STATUS
          ori R16,TWI_MASTER_ARBLOST_bm|TWI_MASTER_RIEN_bm |TWI_MASTER_WIEN_bm
          sts TWIC_MASTER_STATUS,R16
          JMP ISRT_TWIC_TWIM_vect_3
ISRT_TWIC_TWIM_vect_2:
           LDS xh,TWIC_MASTER_DATA
           uart_putc_hex
           ldi xh,0x21
           ldi xl,0x01
           ld r16,x
           clz
           cpi r16,0x00
           breq  ISRT_TWIC_TWIM_vect_4
           LDI R16,TWI_MASTER_CMD_RECVTRANS_gc
           STS TWIC_MASTER_CTRLC,R16
           lds r16,TWIC_MASTER_STATUS
           ori R16,TWI_MASTER_ARBLOST_bm|TWI_MASTER_RIEN_bm |TWI_MASTER_WIEN_bm
           sts TWIC_MASTER_STATUS,R16
           jmp ISRT_TWIC_TWIM_vect_3
ISRT_TWIC_TWIM_vect_4:
           LDI R16,TWI_MASTER_ACKACT_bm|TWI_MASTER_CMD_STOP_gc
           STS TWIC_MASTER_CTRLC,R16
ISRT_TWIC_TWIM_vect_3:
            reti
//----- ISR_TWIC_TWIS_vect（从机中断）--------//
ISR_TWIC_TWIS_vect:
         LDS  R16,TWIC_SLAVE_STATUS
         sbrs r16,6
         jmp end_1
         nop
```

```
        LDI  R16,TWI_SLAVE_CMD_RESPONSE_gc
        STS  TWIC_SLAVE_CTRLB,R16
        jmp  end
end_1:
        LDS  R16,TWIC_SLAVE_STATUS
        sbrs r16,7
        jmp  end
        nop
        sbrs r16,1
        jmp  end_2
        nop
        jmp  end_5
        nop
end_2:
        LDS R16,TWIC_SLAVE_DATA
        clz
        cpi r16,0x00
        brne end_3
        nop
        jmp end_4
end_3:
        ldi yh,0x22
        eor yl,yl
        ldi xh,0x21
        ldi xl,0x01
        ld r17,x
        add yl,r17
        st y,r16
        mov xh,r16
        uart_putc_hex
        LDI  R16,TWI_SLAVE_CMD_RESPONSE_gc
        STS  TWIC_SLAVE_CTRLB,R16
        jmp end
end_4:
        ldi yh,0x22
        eor yl,yl
        ldi xh,0x21
        ldi xl,0x01
        ld r17,x
        add yl,r17
        st y,r16
        mov xh,r16
        uart_putc_hex
        LDI  R16,TWI_SLAVE_ACKACT_bm|TWI_SLAVE_CMD_COMPTRANS_gc
```

```
        STS  TWIC_SLAVE_CTRLB,R16
        jmp end
end_5:
        sbrc r16,4
        jmp end_6
        nop
        ldi yh,0x22
        eor yl,yl
        ldi xh,0x21
        ldi xl,0x01
        ld r17,x
        add yl,r17
        dec r17
        st x,r17
        ld r16,y
        STS TWIC_SLAVE_DATA,R16
        jmp end
end_6:
        LDI  R16,TWI_SLAVE_ACKACT_bm|TWI_SLAVE_CMD_COMPTRANS_gc
        STS  TWIC_SLAVE_CTRLB,R16
end:
        LDI R16,TWI_SLAVE_APIF_bm|TWI_SLAVE_DIF_bm
        STS TWIC_SLAVE_STATUS,r16
        RETI
```

2.TWIF对串行EEPROM AT24C02读写操作。

AT24C02是美国ATMEL公司的低功耗CMOS串行EEPROM，它是内含256×8位存储空间，具有工作电压宽（2.5～5.5V）、擦写次数多（大于10000次）、写入速度快（小于10ms）等特点。每写入或读出一个数据字节后，该地址寄存器自动加1，以实现对下一个存储单元的读写。所有字节均以单一操作方式读取。为降低总的写入时间，一次操作可写入多达8个字节的数据。

AT24C02引脚说明见表5-6-1：

表5-6-1 AT24C02引脚说明

符号	说明	方向
A0-A2	芯片地址	输入
SDA	串行数据/地址输入总线	输入/输出
SCL	时钟总线	输入
WP	写保护	输入
NC	没有定义	----
GND	接地	----
VCC	电源	----

AT24C02使用A0-A2三个引脚确定芯片地址，WP写保护，将该管教接VCC，EEPROM就实现写保护（只读）。将该管教接地或悬空，可以对器件进行读写操作。

功能描述：

AT24Cxxx支持I2C总线数据传输协议。I2C总线协议规定，任何将数据传送到总线的作为发送器，任何从总线接收数据的器件为接收器。数据传送由主器件控制，总线的串行时钟，起始停止条件均有主控制器产生。24Cxxx为从器件。主器件和从器件都可以作为发送器或接收器，但数据传送（接收或发送）模式由主器件控制。

I2C总线协议定义如下：

（1） 只有在总线非忙时才被允许进行数据传送。

（2） 在数据传送时，当时钟线为高电平，数据线必须为固定状态，不允许有跳变。时钟线为高电平时，数据线的任何电平变化将被当作总线的启动或停止。

启动条件：

起始条件必须在所有操作命令之前发送。时钟线保持高电平期间，数据线电平从高到底的跳变作为I2C总线的启动信号。AT24Cxxx一直监视SDA和SCL电平信号直到条件满足才响应。

停止条件：

时钟线保持高电平期间，数据线电平从低到高的跳变作为I2C总线的停止信号。操作结束时必须发送停止条件。

器件地址的约定：

主器件在发送启动命令后开始传送，主器件发送相应的从机器件的地址，8位从机器件地址的高4位固定为1010。接下来的3位用来定义存储器的地址，对于AT24C021/022这三位无意义。,对于AT24C041/042,接下来两位无意义，第三位是地址位高位；对于ATC24C081/082中第一位无意义，后两位表示地址高位。对于AT24C161/162这三位表示地址位高位。

最后一位位读写控制位。1 表示对从器件进行读操作，0 表示对从器件进行写操作。在主器件发送启动命令和发送一字节从器件地址后，如果从器件地址相吻合，AT24Cxxx发送一个应答信号（通过SDA）。然后AT24Cxxx再根据读/写控制进行读或者写操作。

应答信号：

每次数据传送成功后，接收器件发送一个应答信号。当第九个时钟产生时，产生应答信号的器件将SDA下拉为低，通知已经接收到8位数据。接收到起始条件和从地址后，AT24Cxxx发送一个应答信号；如果被选择为写操作，每接受到一字节数据，AT24Cxxx发送一个应答信号。

当接收到读命令后，AT24Cxxx发送一个字节数据，然后释放总线，等待应答信号。一旦接收到应答信号，它将继续发送数据。如果接收到主器件发送的非应答信号，它结束数据传送等待停止条件。

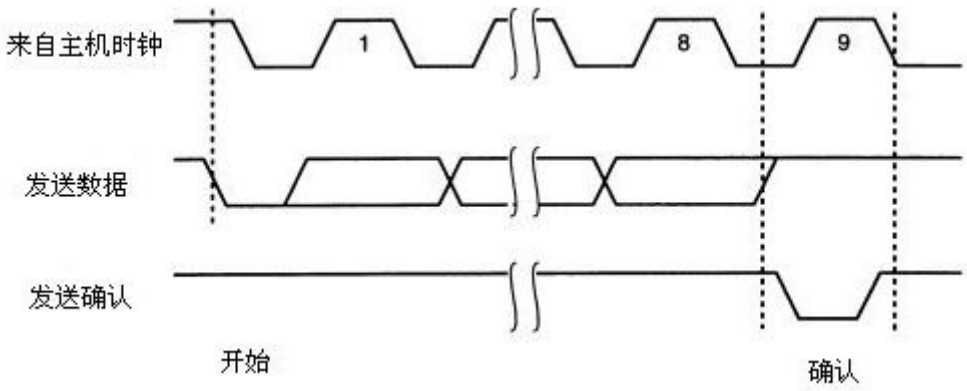


图5-6-1 应答时序

写操作：

字节写：

在字节写模式下，主器件发送起始命令和从器件地址信息给从器件。在从器件响应应答信号后，主器件将要写入数据的地址发送到AT24Cxxx的地址指针，主器件在收到从器件的应答信号后，再送数据到相应数据存储区地址。AT24Cxxx再响应一个应答信号，主器件产生一个停止信号；然后AT24Cxxx启动内部写周期。在内部写周期其间，AT24CxxxA不再响应主器件的任何请求。

页写：

使用页写操作时，做多可以一次向AT24Cxxx中写入16个字节的数据。页写操作的初始化字节写一样，区别在于传送了一个字节数据后，主器件发送15个字节的数据，每传送一个字节数据后，AT24Cxxx响应一个应答信号，寻址字节低位自动加1，而高位保持不变。

如果主器件在发送停止信号前发送的字节数超过16个字节的数据，地址计数器自动翻转，先前写入的数据被自动覆盖。接收到16字节数据后和主器件发送停止位后，AT24Cxxx启动内部写周期将数据写到数据区。

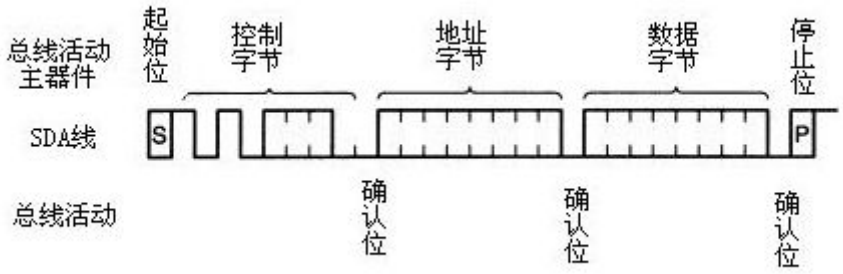


图5-6-2 字节写时序

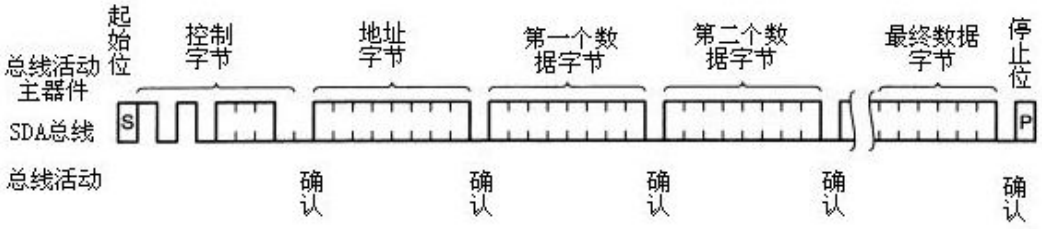


图5-6-3 页写时序

应答查询：

可以利用内部写周期时禁止数据输入这一特性。一旦主机发送停止位指示主机操作结束时，AT24Cxxx启动内部写周期。应答查询立即启动。包括发送一个起始信号和进行写操作的从器件地址。如果AT24Cxxx已经完成了内部自写周期，将发送一个应答信号，主器件可以继续对AT24Cxxx进行读写操作。

写保护：

写保护操作特性可使用户避免由于不当操作而造成对存储区域内部数据的改写，当WP

管教高时，整个寄存器区全部被保护起来而变为只可读取。AT24Cxxx可以接收从机地址和字节地址，但是装置在接收到第一个数据字节后不发送应答信号从而避免寄存器区域被编程改写。

读操作：

对AT24Cxxx读操作的初始化方式和写操作时一样，仅把/R/W位置为1，有三种可能的读操作方式：立即地址读；选择地址读；立即/选择地址连续读。

立即地址读：

AT24Cxxx的地址计数器内容为最后操作字节的地址加1。也就是说，如果上次读/写的操作地址为N，则立即读的地址从地址N+1开始。如果N=E（AT24C021/022中E=255, AT24C041/042中E=511, AT24C081/082中E=1023, AT24C161/162中E=2047）,则寄存器将会翻转到地址0继续输出数据。主机设备不需要发送一个应答信号，但是要产生一个停止信号。

选择地址读：

选择/随机读操作允许主机对寄存器的任意字节进行读操作。主机首先进行一次空写操作，发送起始条件，从机地址和它想读取的字节数据的地址，在AT24Cxxx应答以后，主机重新发送起始条件位和从机地址位，此时R/W为1。AT24CXXX响应并发送应答信号然后输出要求的8位字节数据。主机不发送信号应答，但是产生一个停止位。

连续读：

在连续读方式中，首先执行立即读或选择字节读操作。在AT24Cxxx发送完8位一字节数据后，主机产生一个应答信号来响应，告知AT24Cxxx主机要求更多的数据，对应每个主机产生的应答信号AT24Cxxx将发送一个8位的数据字节。当主机发送非应答信号时结束读操作，然后主机发送一个停止信号。

从AT24Cxxx输出的数据按顺序输出，由N到N+1。读操作时的地址计数器在AT24Cxxx整个寄存器区域增加，这样整个寄存器区域可在一个读操作内全部读出。当超过E（AT24C021/022中E=255, AT24C041/042中E=511, AT24C081/082中E=1023, AT24C161/162中E=2047）字节数据被读出时，计数器将循环计数继续输出数据。

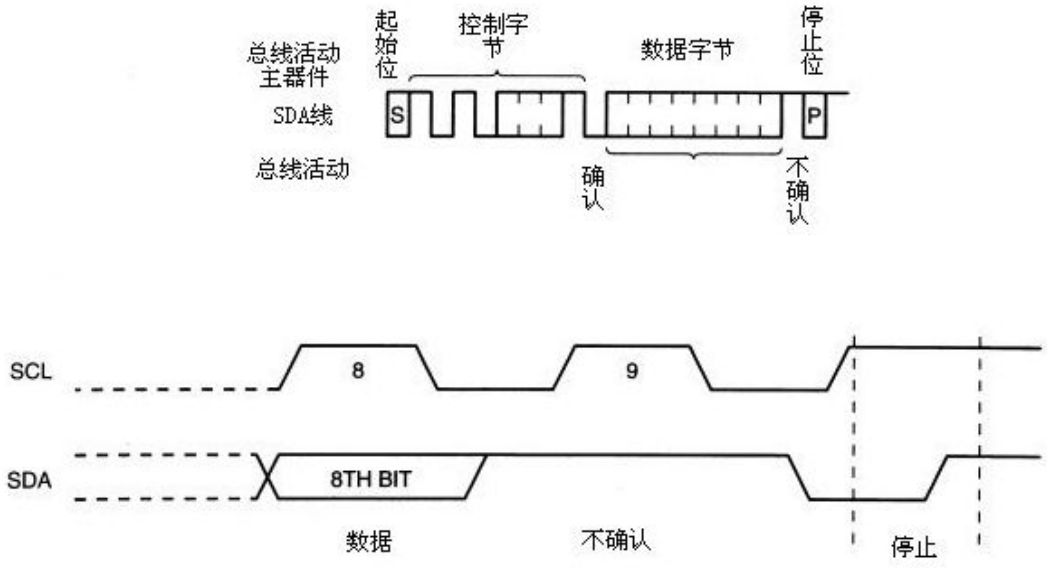


图5-6-4 立即地址读时序

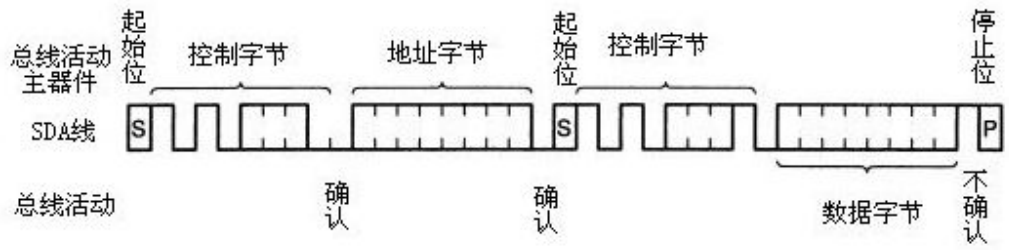


图5-6-5 选择地址读

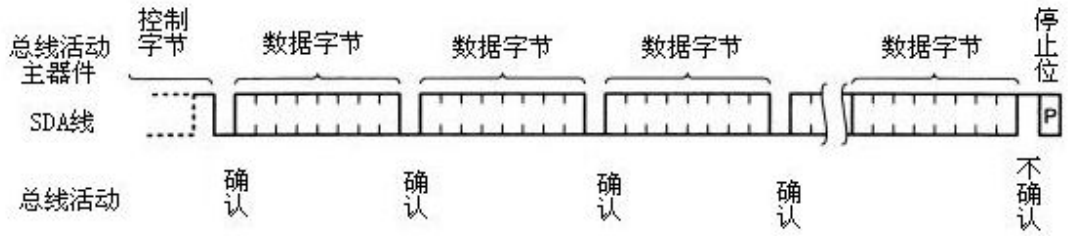


图5-6-6 连续读时序

TWIF设置为主模式，通过PC0向AT24C02写入8个字节，TWIF再通过数据总线PC0读出刚才写入的数据。

硬件连接见图5-6-7：

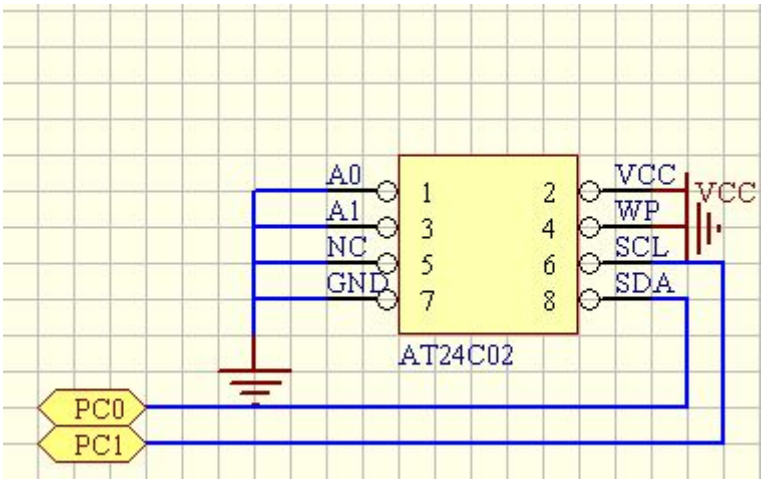


图5-6-7 AT24C02连接原理图

C 语言代码：

```
//-----包含头文件--------//
#include "avr_compiler.h"
```

```
#include <util/delay.h>
#include "twi_master_driver.c"
//器件地址 0 1010 000
#define DEVICE_ADDRESS    0x50
//缓冲字节数
#define NUM_BYTES        9
// CPU 2MHz
#define CPU_SPEED   2000000
// 波特率 100kHz
#define BAUDRATE    100000
#define TWI_BAUDSETTING TWI_BAUD(CPU_SPEED, BAUDRATE)
//片内字节地址
uint8_t WORD_ADDRESS = 0x00;
//结构体变量
TWI_Master_t twiMaster;    //TWI 主机
//测试数据
uint8_t sendBuffer[NUM_BYTES] =
{
0x00,
0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07
};
//-----main（主函数入口）--------//
int main(void)
{
    PORTD.DIRSET = 0xFF; // PORTD 设为输出 LED 显示数据
    // 如果外部没有接上拉电阻，使能上拉 PC0(TWI-SDA), PC1(TWI-SCL)
    PORTCFG.MPCMASK = 0x03; // 一次配置多个引脚
    PORTF.PIN0CTRL=(PORTF.PIN0CTRL&~PORT_OPC_gm) | PORT_OPC_PULLUP_gc;

    //初始化主机
    TWI_MasterInit(&twiMaster,&TWIF,TWI_MASTER_INTLVL_LO_gc,TWI_BAUDSETTING);
    //使能低级别中断
    PMIC.CTRL |= PMIC_LOLVLEN_bm;
    sei();
    uint8_t BufPos = 0;
    while (1)
    {
        PORTD.OUT =0;
        //主机发送数据，一次写 8 个字节
        sendBuffer[0] = WORD_ADDRESS;
        TWI_MasterWrite(&twiMaster,DEVICE_ADDRESS,&sendBuffer[0],9);
        while (twiMaster.status != TWIM_STATUS_READY)
        {
            //等待传输完成
```

```
        }
        //PORTD.OUT = twiMaster.result;
        _delay_ms(600);
        //主机读取数据，先写片内字节地址 再一次读 8 个字节
        TWI_MasterWriteRead(&twiMaster,DEVICE_ADDRESS,WORD_ADDRESS,1,8);
        while (twiMaster.status != TWIM_STATUS_READY)
        {
            //等待传输完成
        }
        for(BufPos=0;BufPos<8;BufPos++)
        {
            //LED 显示取反数据
            PORTD.OUT = ~twiMaster.readData[BufPos];
            _delay_ms(600);
        }
    }
}
//----- ISR（TWIF中断函数入口）--------//
ISR(TWIF_TWIM_vect)
{
    TWI_MasterInterruptHandler(&twiMaster);
}
```

汇编代码：

```
//----- 包含头文件--------//
.include "ATxmega128A1def.inc";器件配置文件,决不可少,不然汇编通不过
.include "usart_driver.inc"
.ORG 0
        RJMP RESET//复位
.ORG  0X01A
        RJMP ISRT_TWIC_TWIM_vect

.ORG 0X100        ;跳过中断区0x00-0x0FF

.EQU ENTER  ='\n'
.EQU NEWLINE='\r'
.EQU EQUE   ='='
.EQU ZERO   ='0'
.EQU A_ASSCI='A'
//器件地址 0B0 1010 000
.equ DEVICE_ADDRESS=0x50
//缓冲字节数
.equ NUM_BYTES=9
// CPU 2MHz
// 波特率100kHz
```

```
.equ TWI_BAUDSETTING=5
//片内字节地址
.equ WORD_ADDRESS=0x00
//测试数据
sendBuffer: .DB 0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x80//0x00写时字节
地址 //0x80是结束标志
MAIN:       .DB 'I','N','T','O','M','A','I','N',0
writedata:  .DB 'w','r','i','t','e','d','a','t','a',0
readdata:   .DB 'r','e','a','d','d','a','t','a',0
//----- uart_init (串口初始化)--------//
uart_init:
       /* USARTC0 引脚方向设置*/
         LDI R16,0X08//PC3 (TXD0) 输出
         STS PORTC_DIRSET,R16
         LDI R16,0X04  //PC2 (RXD0) 输入
         STS PORTC_DIRCLR,R16
       //USARTC0 模式 - 异步  USARTC0帧结构, 8 位数据位, 无校验, 1停止位
         LDI R16,USART_CMODE_ASYNCHRONOUS_gc|USART_CHSIZE_8BIT_gc
                                            |USART_PMODE_DISABLED_gc
         STS USARTC0_CTRLC,R16
         LDI R16,12 //设置波特率 9600
         STS USARTC0_BAUDCTRLA,R16
         LDI R16,0
         STS USARTC0_BAUDCTRLB,R16
         LDI R16,USART_TXEN_bm//USARTC0 使能发送
         STS USARTC0_CTRLB,R16
         RET
 //----- RESET (主函数入口)--------//
RESET:
        CALL uart_init
        LDI R16,ENTER
        PUSH R16
        uart_putc
        uart_puts_string MAIN
        LDI R16,0X03
        STS PORTD_DIRSET,R16
       // 如果外部没有接上拉电阻，使能上拉 PC0(TWI-SDA), PC1(TWI-SCL)
        LDI R16,0X03
        STS PORTCFG_MPCMASK,R16 // 一次配置多个引脚
        LDI R16,PORT_OPC_PULLUP_gc
        STS PORTC_PIN0CTRL,R16
        LDI R16,TWI_BAUDSETTING
        STS TWIC_MASTER_BAUD,R16
         LDI R16,TWI_MASTER_INTLVL_LO_gc|TWI_MASTER_RIEN_bm |TWI_MASTER_WIEN_bm
```

```
                                                    |TWI_MASTER_ENABLE_b
                                                    m
        STS TWIC_MASTER_CTRLA,R16
        ldi r16,TWI_MASTER_BUSSTATE_IDLE_gc
        sts twic_MASTER_STATUS,r16
        LDI R16,PMIC_LOLVLEN_bm//使能低级别中断
        STS PMIC_CTRL,R16
        SEI
RESET_1:
        LDI R16,ENTER
        PUSH R16
        uart_putc
        uart_puts_string writedata
        LDI R16,ENTER
        PUSH R16
        uart_putc
        //主机发送数据
        // 根据bytesToWrite判断是写, 发送START信号 + 7位地址 +  R/_W = 0
        LDI R16,DEVICE_ADDRESS<<1&0x0fe
        STS TWIC_MASTER_ADDR,R16
        ldi r16,0x00
        sts  GPIO_GPIOR0,r16//标志位
        sts  GPIO_GPIOR1,r16
        sts  GPIO_GPIOR2,r16//计数
RESET_2:
        lds r16,GPIO_GPIOR0
        cpi r16,0x01
        brne RESET_2
//必须有个延迟 因为芯片在收到stop信号时才开始写入之前发给他的八个字节
        LDI R16,ENTER
        PUSH R16
        uart_putc
        uart_puts_string readdata
        LDI R16,ENTER
        PUSH R16
        uart_putc
        //主机伪写指令目的是确定要读取数据的地址
        LDI R16,DEVICE_ADDRESS<<1&0x0fe//伪写指令 目的是为了确定读取的地址（发送
         器件// 地址 字节地址0x00 以确定读取数据的地址）
        STS TWIC_MASTER_ADDR,R16
        ldi r16,0x00
        sts  GPIO_GPIOR0,r16
        ldi r16,0x01
        sts  GPIO_GPIOR1,r16//标志位
```

```
RESET_4:
        lds r16,GPIO_GPIOR1
        cpi r16,0x00
        brne RESET_4
        LDI R16,DEVICE_ADDRESS<<1|0x01
        STS TWIC_MASTER_ADDR,R16
        ldi r16,0x00
        sts  GPIO_GPIOR0,r16
RESET_5:
        lds r16,GPIO_GPIOR0
        cpi r16,0x01
        brne RESET_5
RESET_6:
        jmp RESET_6
//----- ISRT_TWIC_TWIM_vect (TWIF中断函数入口)--------//
ISRT_TWIC_TWIM_vect:
          LDS  R16,TWIC_MASTER_STATUS
          SBRS R16,TWI_MASTER_WIF_bp
          JMP ISRT_TWIC_TWIM_vect_2//读中断
          nop
          sbrs r16,4
          jmp ISRT_TWIC_TWIM_vect_0
          NOP
          jmp ISRT_TWIC_TWIM_vect_5
ISRT_TWIC_TWIM_vect_0://写中断
          lds r16,GPIO_GPIOR1
          sbrc r16,0
          jmp ISRT_TWIC_TWIM_vect_1
          LDI ZH,HIGH(sendBuffer<<1)
          LDI ZL,LOW(sendBuffer<<1)
          lds r19,GPIO_GPIOR2
          add zl,r19
          lpm r16,z+
          inc r19
          sts GPIO_GPIOR2,r19
          sbrc r16,7
          jmp ISRT_TWIC_TWIM_vect_4
          mov xh,r16
          uart_putc_hex
          nop
          STS TWIC_MASTER_DATA,R16
          lds r16,TWIC_MASTER_STATUS
          ori R16,TWI_MASTER_ARBLOST_bm|TWI_MASTER_RIEN_bm |TWI_MASTER_WIEN_bm
          sts TWIC_MASTER_STATUS,R16
```

```
          jmp ISRT_TWIC_TWIM_vect_3
ISRT_TWIC_TWIM_vect_1:
         lds r16,GPIO_GPIOR0
         sbrc r16,0
         jmp ISRT_TWIC_TWIM_vect_4
         nop
         ldi r16,0x00
         STS TWIC_MASTER_DATA,R16
         lds r16,TWIC_MASTER_STATUS
         ori R16,TWI_MASTER_ARBLOST_bm|TWI_MASTER_RIEN_bm |TWI_MASTER_WIEN_bm
         sts TWIC_MASTER_STATUS,R16
         ldi r16,0x01
         sts  GPIO_GPIOR0,r16
         JMP ISRT_TWIC_TWIM_vect_3
ISRT_TWIC_TWIM_vect_2:
          LDS xh,TWIC_MASTER_DATA
          uart_putc_hex
          lds r19,GPIO_GPIOR2
          dec r19
          clz
          cpi r19,0x02
          breq  ISRT_TWIC_TWIM_vect_4
          sts GPIO_GPIOR2,r19
          LDI R16,TWI_MASTER_CMD_RECVTRANS_gc
          STS TWIC_MASTER_CTRLC,R16
ISRT_TWIC_TWIM_vect_5:
          lds r16,TWIC_MASTER_STATUS
          ori R16,TWI_MASTER_ARBLOST_bm|TWI_MASTER_RIEN_bm |TWI_MASTER_WIEN_bm
          sts TWIC_MASTER_STATUS,R16
          jmp ISRT_TWIC_TWIM_vect_3
ISRT_TWIC_TWIM_vect_4:
          LDI R16,TWI_MASTER_ACKACT_bm|TWI_MASTER_CMD_STOP_gc
          STS TWIC_MASTER_CTRLC,R16
          lds r16,TWIC_MASTER_STATUS
          ori R16,TWI_MASTER_ARBLOST_bm|TWI_MASTER_RIEN_bm |TWI_MASTER_WIEN_bm
          sts TWIC_MASTER_STATUS,R16
          ldi r16,0x01
          sts GPIO_GPIOR0,r16
          ldi r16,0x00
          sts GPIO_GPIOR1,r16
ISRT_TWIC_TWIM_vect_3:
           reti
//-----_delay_ms (延迟函数)--------//
_delay_ms:
```

```
L0:     LDI R18,250
L1:     DEC R18
        BRNE L1
        DEC R17
        BRNE L0
        RET
```

5.7 SPI 实例

串行外设接口（SPI）是一种使用 3 或 4 线的高速同步传输接口。它支持 XMEGA 设备和外设或者 AVR 设备之间的高速通信。SPI 支持全双工传输。

连接到总线的设备必须作为主机或者从机。主机通过拉低目标从机片选线（SS）初始化通信。主从机在移位寄存器内准备好将要传输的数据。主机产生 SCK 线上需要的时钟。数据在 MOSI 线上总是从主机移位到从机，在 MISO 线上数据总是从从机移位到主机。数据包过后，主机通过拉高 SS 线和从机同步。

1.SPI自发自收。SPIE设为主模式，SPIF设为从模式，SPIE向SPIF发送数据，SPIF接收到数据将其返回，SPIE收到返回的数据通过USARTC0串口打印出来，发送数据方式分两种：按字节发送；按数据包发送。具体实现参考代码，硬件连接见下面说明

```
 *    - PE4 与 PF4连接(SS)
 *    - PE5 与 PF5 连接(MOSI)
 *    - PE6 与 PF6连接(MISO)
 *    - PE7 与 PF7 连接(SCK)
```

C 语言代码：

```
//-----包含头文件--------//
#include "avr_compiler.h"
#include "usart_driver.c"
#include "spi_driver.c"
//测试数据字节数
#define NUM_BYTES     4
//端口E的SPI模块作为主机
SPI_Master_t spiMasterE;
//端口F的SPI模块作为从机
SPI_Slave_t spiSlaveF;
// SPI 数据包
SPI_DataPacket_t dataPacket;
//主机要发送的数据
uint8_t masterSendData[NUM_BYTES] = {0x11, 0x22, 0x33, 0x44};
//从机接收的数据
uint8_t masterReceivedData[NUM_BYTES];
//测试结果
bool success = true;
//----- uart_init（串口初始化）--------//
void uart_init(void)
```

```
{
    /* USARTC0 引脚方向设置*/
    PORTC.DIRSET   = PIN3_bm; // PC3 (TXD0) 输出
    PORTC.DIRCLR   = PIN2_bm; //PC2 (RXD0) 输入
    //USARTC0 模式 - 异步
    USART_SetMode(&USARTC0,USART_CMODE_ASYNCHRONOUS_gc);
    //USARTC0帧结构, 8 位数据位, 无校验, 1停止位
    USART_Format_Set(&USARTC0, USART_CHSIZE_8BIT_gc,USART_PMODE_DISABLED_gc,
false);
    USART_Baudrate_Set(&USARTC0, 12 , 0); //设置波特率 9600
    USART_Tx_Enable(&USARTC0); //USARTC0 使能发送
    USART_Rx_Enable(&USARTC0); //USARTC0 使能接收
}
 //-----main（主函数入口）--------//
int main(void)
{
    uart_init();
    uart_puts("inside main");uart_putc('\n');
    PORTE.DIRSET = PIN4_bm; //SS引脚上拉，方向输出
    PORTE.PIN4CTRL = PORT_OPC_WIREDANDPULL_gc;
    PORTE.OUTSET = PIN4_bm; //SS拉高
    PORT_t *ssPort = &PORTE; //SS引脚所在端口
    //初始化端口E上的SPI主机
    SPI_MasterInit(&spiMasterE,
                   &SPIE,
                   &PORTE,
                   false,
                   SPI_MODE_0_gc,
                   SPI_INTLVL_OFF_gc,
                   false,
                   SPI_PRESCALER_DIV4_gc);
    //初始化端口F上的SPI从机
    SPI_SlaveInit(&spiSlaveF,
                  &SPIF,
                  &PORTF,
                  false,
                  SPI_MODE_0_gc,
                  SPI_INTLVL_OFF_gc);
    SPI_MasterSSLow(ssPort, PIN4_bm); // 主机: 拉低SS
    uint8_t result = 0;
    /*1.单个字节方式传输*/
    for(uint8_t i = 0; i < NUM_BYTES; i++)
    {
        //主机: 主机传输数据到从机
```

```
    SPI_MasterTransceiveByte(&spiMasterE, masterSendData[i]);
    uart_puts("SPI_MasterTransceiveByte() return result = ");
    uart_putc_hex(result);
    uart_putc('\n');
    //从机: 等待数据可用
    while (SPI_SlaveDataAvailable(&spiSlaveF) == false){}
    uart_puts("SPI_SlaveDataAvailable");
    uart_putc('\n');
        uint8_t slaveByte = SPI_SlaveReadByte(&spiSlaveF); //从机: 取数据
    uart_puts("slaveByte = ");
    uart_putc_hex(slaveByte);
    uart_putc('\n');
    //从机: 收到数据加1, 发送回去
    slaveByte++;
    SPI_SlaveWriteByte(&spiSlaveF, slaveByte);
    //主机: 发送空字节读数据
    uint8_t masterReceivedByte = SPI_MasterTransceiveByte(&spiMasterE, 0x00);
    uart_puts("masterReceivedByte = ");
    uart_putc_hex(masterReceivedByte);
    uart_putc('\n');
    //主机: 检查数据是否正确
    if (masterReceivedByte != (masterSendData[i] + 1) )
    {
        success = false;
        uart_puts("success = false");
        uart_putc('\n');
    }
    uart_putc('\n');
    uart_putc('\n');
}
SPI_MasterSSHigh(ssPort, PIN4_bm); //主机: 释放SS
 /*2: 传输数据包*/
// 创建数据包(SS 在PE4)
SPI_MasterCreateDataPacket(&dataPacket,
                           masterSendData,
                           masterReceivedData,
                           NUM_BYTES,
                           &PORTE,
                           PIN4_bm);
// 传输数据包
uint8_t SPI_MTP_FLAG = SPI_MasterTransceivePacket(&spiMasterE, &dataPacket);
uart_puts("SPI_MTP_FLAG = ");
uart_putc_hex(SPI_MTP_FLAG);
uart_putc('\n');
```

```
    // 检查接收数据的正确性，假设接收正确
    for (uint8_t i = 0; i < NUM_BYTES - 1; i++)
    {
        if (masterReceivedData[i + 1] != masterSendData[i])
        {
            success = false;
            uart_puts("success = false");
            uart_putc('\n');
        }
        uart_puts("masterReceivedData[");uart_putw_dec(i);uart_puts("+ 1] = ");
        uart_putc_hex(masterReceivedData[i + 1]);uart_putc('\n');
    }
    while(true)
    {
        nop();
    }
}
```

汇编代码：

```
//-----包含头文件--------//
.include "ATxmega128A1def.inc";器件配置文件,决不可少,不然汇编通不过
.include "usart_driver.inc"
.ORG 0
        RJMP RESET//复位
.ORG 0X100        ;跳过中断区0x00-0x0FF
.EQU ENTER  ='\n'
.EQU NEWLINE='\r'
.EQU EQUE   ='='
.EQU ZERO   ='0'
.EQU A_ASSCI='A'
MAIN: .DB 'I','N','T','0','M','A','I','N',0
//主机要发送的数据
masterSendData: .db 0x11, 0x22, 0x33, 0x44
 //---------uart_init(串口初始化) --------//
uart_init:
        /* USARTC0 引脚方向设置*/
         LDI R16,0X08//PC3 (TXD0) 输出
         STS PORTC_DIRSET,R16
         LDI R16,0X04 //PC2 (RXD0) 输入
         STS PORTC_DIRCLR,R16
        //USARTC0 模式 - 异步 USARTC0帧结构, 8 位数据位, 无校验, 1停止位
         LDIR16,USART_CMODE_ASYNCHRONOUS_gc|USART_CHSIZE_8BIT_gc
                                           |USART_PMODE_DISABLED_gc
```

```
        STS USARTC0_CTRLC,R16
        LDI R16,12 //设置波特率 9600
        STS USARTC0_BAUDCTRLA,R16
        LDI R16,0
        STS USARTC0_BAUDCTRLB,R16
        LDI R16,USART_TXEN_bm // USARTC0 使能发送
        STS USARTC0_CTRLB,R16
        RET
//--------RESET (主函数入口) --------//
RESET:
     call uart_init
     uart_puts_string  MAIN;uart_putc('\n');
     LDI R16,0x10//SS引脚上拉，方向输出
     STS PORTE_DIRSET,R16
     LDI R16,PORT_OPC_WIREDANDPULL_gc
     STS PORTE_PIN4CTRL,R16
     LDI R16,0x10//SS拉高
     STS PORTE_OUTSET,R16
    //初始化端口E上的SPI主机
     LDI R16,SPI_PRESCALER_DIV4_gc|SPI_ENABLE_bm |SPI_MASTER_bm|SPI_MODE_0_gc
     STS SPIE_CTRL,R16
     LDI R16,SPI_INTLVL_OFF_gc
     STS SPIE_INTCTRL,R16
     LDI R16,0XB0
     STS PORTE_DIRSET,R16
    //初始化端口F上的SPI从机
     LDI R16,SPI_ENABLE_bm|SPI_MODE_0_gc
     STS SPIF_CTRL,R16
     LDI R16,SPI_INTLVL_OFF_gc //中断级别
     STS SPIF_INTCTRL,R16
     LDI R16,0X40
     STS PORTF_DIRSET,R16
     LDI R16,0X10// 主机: 拉低SS
     STS PORTE_OUTCLR,R16
     //主机: 主机传输数据到从机
     //发送数据0XAA
     LDI R16,0XAA
     STS SPIE_DATA,R16
    //等待传输完成
RESET_1:
     LDS R16,SPIE_STATUS
     SBRS R16,SPI_IF_bp
     JMP RESET_1
     nop
```

```
     //从机: 等待数据可用
RESET_2:
     LDS R16,SPIF_STATUS
     SBRS R16,SPI_IF_bp
     jmp RESET_2
     nop
    //从机: 取数据
     LDI R16,ENTER //回车换行
     PUSH R16
     uart_putc
     LDS XH,SPIF_DATA
     uart_putc_hex
     LDI R16,ENTER //回车换行
     PUSH R16
     uart_putc
    //发送数据0XBB
     LDI R16,0XBB
     STS SPIF_DATA,R16
     LDI R16,0X00
     STS SPIE_DATA,R16
RESET_3:
     LDS R16,SPIE_STATUS
     SBRS R16,SPI_IF_bp
     JMP RESET_3
     nop
     LDI R16,ENTER //回车换行
     PUSH R16
     uart_putc
     LDS XH,SPIE_DATA
     uart_putc_hex
     LDI R16,ENTER //回车换行
     PUSH R16
     uart_putc
RESET_4:
     JMP RESET_4
     RET
```

2.SPID对外部器件FM25V10读写操作。

由于PC默认的只带有RS-232接口,有两种方式可以得到PC上位机的RS-485电路:1)通过RS-232/RS-485转换电路将PC机串口RS-232信号转换成RS-232信号转换成RS-485信号，对于情况比较复杂的工业环境，最好是选用防浪带隔离栅的产品。2）通过PCI多串口卡,可以直接选用输出信号为RS-485类型的扩展卡。

在单片机应用系统中，常使用3.3V供电芯片MAX3485来完成TTL和RS-485的半双工转换。见图5-7-1，第一脚（RO）与XMEGA的USARTC0的PC2（RXD）相连，作为通信电路数据接收。第七脚（DI）与XMEGA的USARTC0的PC3（TXD）相连,作为通信电路的数据发送。/RE/DE短

接并一同与PD0相连，作为MAX3485接收和发送的使能信号，当准备发送时，应将PD0置为高电平，此时接收被禁止，发送被允许。当准备接收数据时，应将PD0置为低电平，此时发送被禁止，接收被允许。需要注意的是，在接收和发送之前，应先将使能端置为合适的电平，并延时一小段时间，否则可能导致第一个字符接收或发送异常；在接收和发送完成之后如果要切换使能状态，也应延时一小段时间，否则会导致数据帧的最后一个字符发送或接收异常。

FM25V10:结构容量为128Kx8位；读/写次数达到100万亿（10^{14}）次；掉电数据保存10年；写数据无延迟；采用先进的高可靠性铁电制造工艺；高速串行外设接口- SPI总线；频率可达40MHz；硬件上可直接替换串行Flash；写保护机制：硬件保护，软件保护；器件ID能读出制造商、产品密度和版本信息；工作电压：2.0V~3.6V；待机电流（典型值）：90μA，睡眠模式电流（典型值）：5μA；工业级温度：-40℃~+85℃；8脚环保/RoHS，SO封装。引脚说明见表5-7-1：

表5-7-1 FM25V10引脚说明

符号	说明	方向
/S	片选	输入
Q	数据输出	输出
/W	写保护	输入
VSS	接地	----
D	数据输入	输入
C	时钟	----
/HOLD	允许中断位	输入
VDD	电源	----

FM25V10仅仅支持SPI的模式0和模式3，图5-7-1显示了SPI模式0和模式3的时序，两种模式下数据位都是在上升沿被采样。

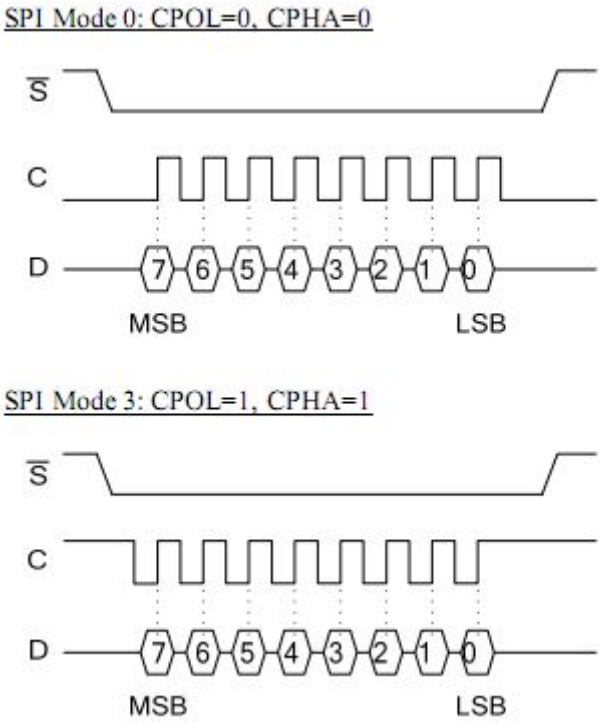


图5-7-1 SPI模式0和3时序

FM25V10控制命令字说明见表5-7-2

表5-7-2 FM25V10命令字

符号	说明	命令字
WREN	写使能	00000110
WRDI	禁止写	00000100
RDSR	读状态寄存器	00000101
WRSR	写状态寄存器	00000001
READ	读存储区	00000011
FSTRD	快速度存储区	00001011
WRITE	写存储区	00000010
SLEEP	进入睡眠	10111001
RDID	读器件ID	10011111
SNR	读器件序列号	11000011

FM25V10状态寄存器：

状态寄存器位说明见表5-7-2：

表5-7-2 FM25V10状态寄存器位功能

位	符号	说明
0	0	----
1	WEL	写使能位，此位只能通过WREN与WRDI命令字改写
2	BP0	存储器保护位
3	BP1	存储器保护位
4	0	----
5	0	----
6	1	----
7	WPEN	为低时/W脚忽略，为高时/W脚控制是否可以写状态寄存器

保护存储区通过BP0和BP1设置见表5-7-3：

BP1	BP0	保护地址范围
0	0	----
0	1	18000H-1FFFFH
1	0	10000H-1FFFFH
1	1	00000H-1FFFFH

写保护设置见表5-7-4：

WEL	WPEN	/W	保护存储区	未被保护存储区	状态寄存器
0	X	X	保护	保护	保护
1	0	X	保护	不保护	不保护
1	1	0	保护	不保护	保护
1	1	1	保护	不保护	不保护

写使能：

FM25V10上电初始时禁止写的，在写之前必须要写使能。发送写使能命令字允许用户写操作，写操作包括写状态寄存器和写存储区。发送WREN命令字会使状态寄存器中WEL置位，此位标志写使能，人工写状态寄存器不会影响此位。每次在写操作完成后，写使能位自动置0，如果想再次执行写操作必须重新发送WEN命令字，时序图见图5-7-2：

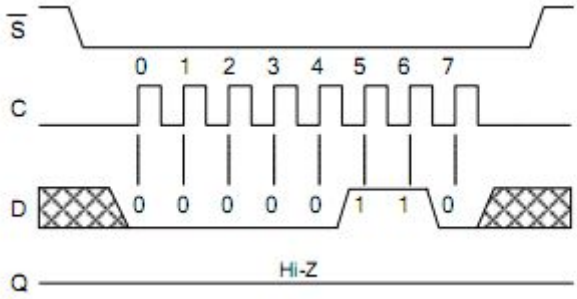


图5-7-2 写使能时序

禁止写：

发送禁止写命令字，状态寄存器中WEL会置0，时序见图5-7-3：

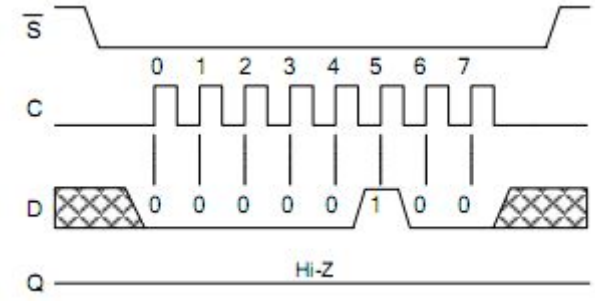


图5-7-3 禁止写时序

读状态寄存器：

发送读状态寄存器命令字，FM25V10将会响应发回状态寄存器的内容。时序见图5-7-4：

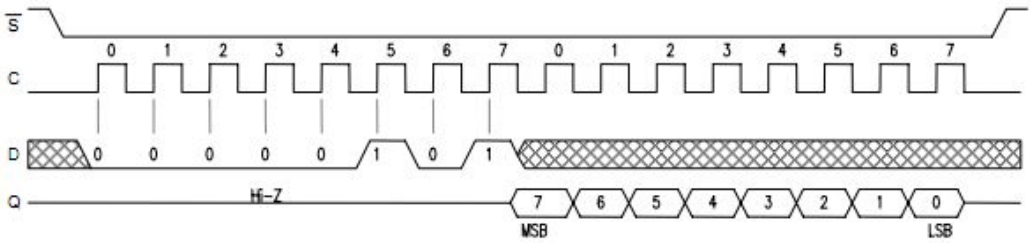


图5-7-4 读状态寄存器时序

写状态寄存器：

在没有被保护的情况下，FM25V10写状态寄存之前，必须要写使能，写使能以后发送写状态寄存器命令字，当命令字发送完成开始发送要写入状态寄存器的内容。时序见图5-7-5：

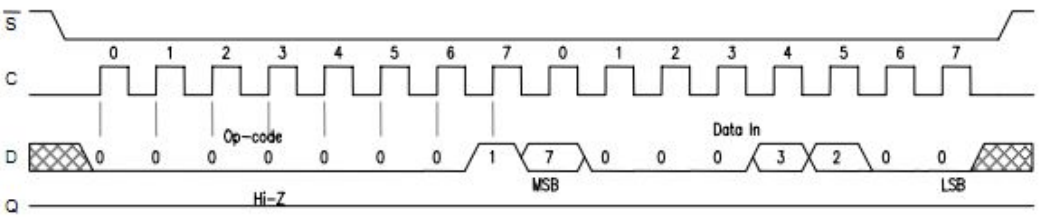


图5-7-5 写状态寄存器时序

写存储区：

在没有被保护的情况下，FM25V10写存储区之前，必须要写使能，写使能以后发送写存储区命令字，紧接其后是三个字节的存储区地址，发送完成以后发送要写入的数据。数据可以是单个字节也可以是多个字节，内部地址会自动递加，当地址为1FFFH时地址会从0开始。时序见图5-7-6：

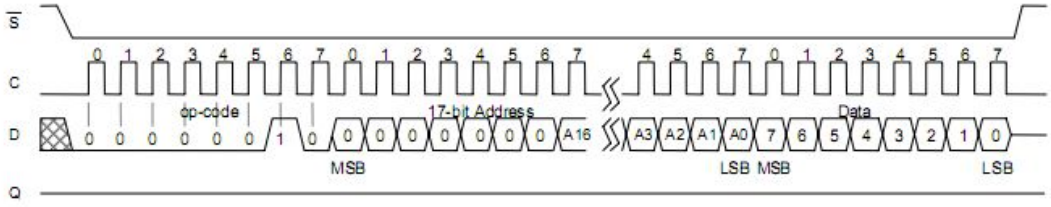


图5-7-6 写存储区时序（包含三个地址字节）

读存储区：

发送读存储区地址，发送完成以后发送三个字节地址，FM25V10接收到地址就会返回相应地址的内容，FM25V10地址会自动加1。读存储区时序见图5-7-7：

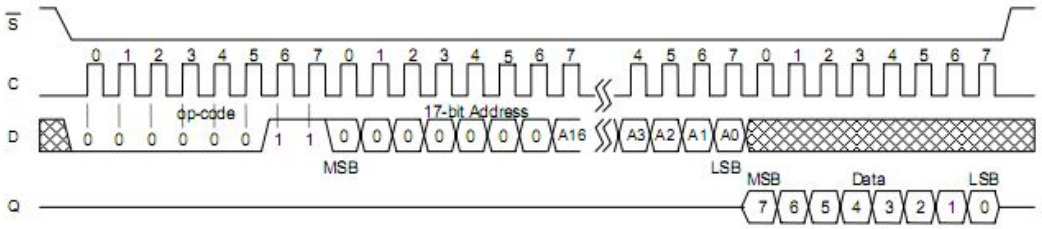


图5-7-7 读存储区时序（包含三个地址字节）

快速读存储区：

为了兼容串行FLASH，FM25V10支持快速读存储区。快速读存储区与读存储区有点相似，不同点1)发送的命令字不同；2）在发送完三个字节地址以后，快速读存储区方式下FM25V10不会返回要读取地址的内容，必须在三个地址字节后面加一个虚拟字节（内容不限），FM25V10在收到快速读存储区命令字，三个字节地址和虚拟字节后FM25V10会返回相应地址的内容，如果想读取多个字节，继续发送虚拟字节就可以读取。快速读存储区时序见图5-7-8：

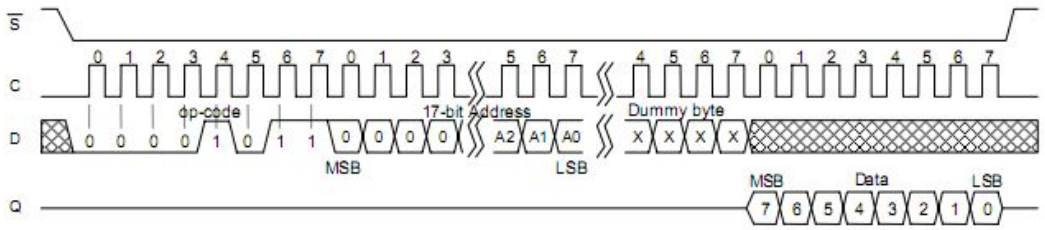


图5-7-8 快速读存储区时序（包含三个地址字节）

睡眠模式：

发送进入睡眠命令字，FM25V10会进入睡眠状态，/S引脚电平置低可以唤醒FM25V10，发送进入睡眠命令字时序见图5-7-9：

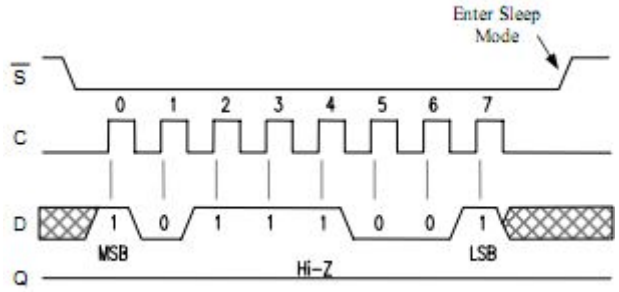


图5-7-9 进入睡眠时序

FM25V10和FM25VN10有/HOLD引脚。此引脚必须在C引脚为低电平时改变，/HOLD变为低电平时中断FM25V10当前操作；在/HOLD变为高电平时重新开始被中断的操作。如果不使用这个功能，请将/HOLD连接到VCC上。

FM25V10与XMEGA128A1的连接说明如下：

```
*    |---XMEGA128A1--|---------FM25V10---------------|
*    |---PD4---SS----|---1---/S---（片选）-----------|
*    |---PD6---MISO--|---3----Q---（数据输出）-------|
*    |---VCC---------|---5---/W---（写保护）---------|
*    |---GND---------|---7----VSS-------（接地）-----|
*    |---PD5---MOSI--|---8----D---（数据输入）-------|
*    |---PD7---SCK---|---6----C---（时钟）-----------|
*    |---VCC---------|---4---/HOLD--------(HOLD)-----|
*    |——VCC---------|---2----VDD---(电源)-----------|
```

硬件连接见图5-7-10：

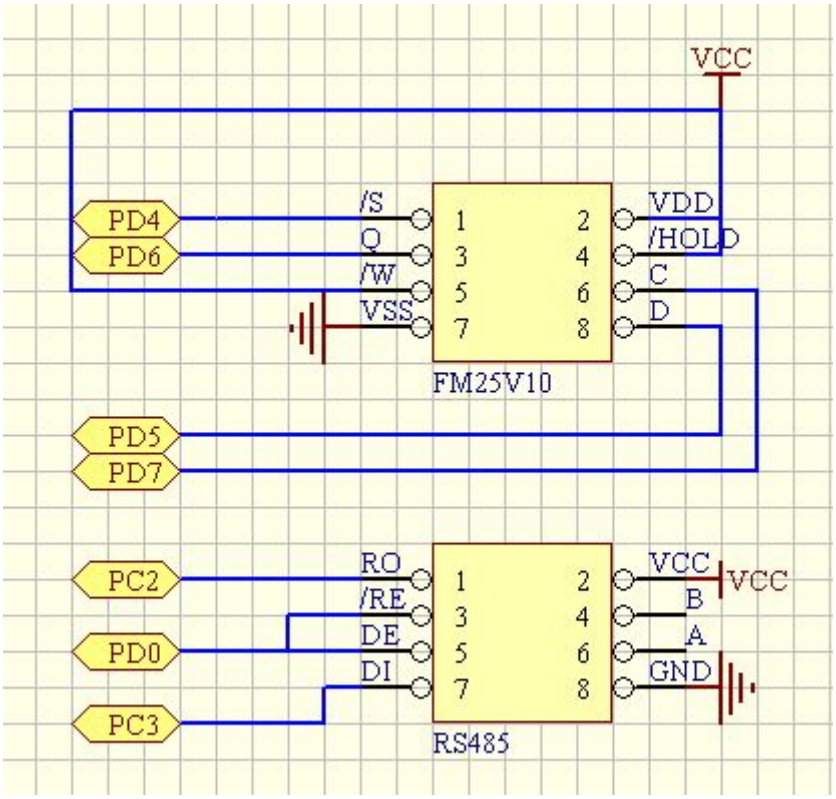


图 5-7-10 FM25V10 连接原理图

C 语言代码：

;发送写数据格式 ：F9(写标志)00 00 00（三字节地址） 02（要写数据个数） 01 02（要写的数据）

;发送读数据格式： FA(读标志)00 00 00 （要读的起始地址）02（要读的数据个数）

C 语言代码：

```
//--------包含头文件--------//
#include <avr/io.h>
#include "avr_compiler.h"
#include "clksys_driver.c"
#include "usart_driver.c"
#include "TC_driver.c"
uint8_t receivedata[100];
uint8_t writedata[100];
//---------PLL_XOSC_Initial (时钟设置函数)--------//
void PLL_XOSC_Initial(void)
{
    unsigned char factor =3;
    //设置晶振范围 启动时间
    CLKSYS_XOSC_Config( OSC_FRQRANGE_2TO9_gc, false,OSC_XOSCSEL_XTAL_16KCLK_gc );
    CLKSYS_Enable( OSC_XOSCEN_bm  );//使能外部振荡器
    do {} while ( CLKSYS_IsReady( OSC_XOSCRDY_bm  ) == 0 );//等待外部振荡器准备好
    //设置倍频因子并选择外部振荡器为PLL参考时钟
```

```
    CLKSYS_PLL_Config( OSC_PLLSRC_XOSC_gc, factor );
    CLKSYS_Enable( OSC_PLLEN_bm );//使能PLL电路
    do {} while ( CLKSYS_IsReady( OSC_PLLRDY_bm ) == 0 );//等待PLL准备好
    CLKSYS_Main_ClockSource_Select( CLK_SCLKSEL_PLL_gc);//选择系统时钟源
    //设置预分频器A,B,C的值
    CLKSYS_Prescalers_Config( CLK_PSADIV_1_gc, CLK_PSBCDIV_1_1_gc );
}
 //---------SPI_MasterInit (SPI设置函数)--------//
void SPI_MasterInit(void)
 {
        PORTCFG.VPCTRLA=0x10;//;PORTB映射到虚拟端口1，PORTA映射到虚拟端口0
        PORTCFG.VPCTRLB=0x32;//;PORTC映射到虚拟端口2，PORTD映射到虚拟端口3
        //SPI初始化
        VPORT3_DIR=0x10; //SS片选引脚设为方向输出
        VPORT3_DIR|=0x20;//MOSI引脚设为方向输出
        VPORT3_DIR|=0x80;//SCK引脚设为方向输出
        VPORT3_DIR|=0x01;//485控制引脚
        /*0,1,0,1,00,00;SPI Clock Double DISABLE,SPI module Enable, Data
         Order=MSB，Master Select, SPI Mode=0, SPI Clock
         Prescaler=CLKper/4=0.5MHZ*/
        SPID_CTRL=0x50;//SPI控制寄存器
        SPID_INTCTRL=0x00;//SPI 中断 关闭
 }
//---------uart_init (串口初始化)--------//
void uart_init(void)
{
    /* USARTC0 引脚方向设置*/
    PORTC.DIRSET   = PIN3_bm; // PC3 (TXD0) 输出
    PORTC.DIRCLR   = PIN2_bm; //PC2 (RXD0) 输入
    USART_SetMode(&USARTC0,USART_CMODE_ASYNCHRONOUS_gc); //USARTC0 模式 - 异步
    /* USARTC0帧结构, 8 位数据位, 无校验, 1停止位 */
    USART_Format_Set(&USARTC0, USART_CHSIZE_8BIT_gc,USART_PMODE_DISABLED_gc,
false);
    USART_Baudrate_Set(&USARTC0, 77 , 0); //设置波特率 9600
    USART_Tx_Enable(&USARTC0); //USARTC0 使能发送
    USART_Rx_Enable(&USARTC0); //USARTC0 使能接收
}
//---------main(主函数入口)--------//
void main(void)
{
        PLL_XOSC_Initial();
        SPI_MasterInit();
        uart_init();
        //指示灯
```

```
        VPORT0_DIR=0x04;//PORTA2输出
        VPORT0_DIR|=0x08;//PORTA3输出
        VPORT0_DIR|=0x10;//PORTA4输出
        //计数器时钟源为24MHZ/1024=23437.5Hz
        TC_SetPeriod( &TCC0, 0x1000 );
        TC0_ConfigClockSource( &TCC0, TC_CLKSEL_DIV1024_gc);
        USARTC0_CTRLA=0X38;
        PMIC.CTRL |=PMIC_MEDLVLEN_bm+PMIC_LOLVLEN_bm+PMIC_HILVLEN_bm;
            sei();
        while(1)
        VPORT3_OUT&=0xFE;//PORTD0输出低 使能485接收
 }
 //---------SPI_READ_DATA (SPI读函数)--------//
void SPI_READ_DATA(void) //读数据流程
{
        VPORT0_OUT=0x08;//接收过程亮灯
        //发送操作码之前片选产生下降沿
        VPORT3_OUT|=0x10;
        for(uint8_t i=0;i<=5;i++);
        VPORT3_OUT&=0xef;
        SPID_DATA=0x03; //发送读操作码，操作码 0X03
        while((SPID_STATUS&0x80)!=0x80);
        //数据移位完成，写3个字节的地址
        SPI_WRITE_ADDRESS();
        for(uint8_t i=0;i<receivedata[4];i++)
         {
          SPID_DATA=0x00;
          //读写数据都要等待SPI移位完成标志位置位
           while((SPID_STATUS&0x80)!=0x80);
           writedata[i]=SPID_DATA;
          }
         VPORT3_OUT|=0x01;//PORTD0输出高 使能485发送
         for(uint8_t j=0;j<receivedata[4];j++)
          {
           while(!(USARTC0.STATUS & USART_DREIF_bm));
            USART_PutChar(&USARTC0,writedata[j]);
           }
         while(!(USARTC0.STATUS & USART_TXCIF_bm));
         VPORT3_OUT|=0x10;//读操作完成，片选SS拉高
         VPORT0_OUT&=0xF7;//灯灭
}
//---------SPI_WRITE_DATA (SPI写函数)--------//
void SPI_WRITE_DATA(void)//写数据流程
{
```

```
        //发送操作码之前片选产生下降沿
        VPORT0_OUT=0x04;//接收过程亮灯
        VPORT3_OUT|=0x10;
        for(uint8_t i=0;i<=5;i++);
        VPORT3_OUT&=0xef;
        //发送写使能操作码，写操作码 0X06
        SPID_DATA=0X06;
        while((SPID_STATUS&0x80)!=0x80);
        VPORT3_OUT|=0x10;//发送写使能操作码 完成，片选SS拉高
        VPORT3_OUT|=0x10; //发送操作码之前片选产生下降沿
        for(uint8_t i=0;i<=5;i++);
        VPORT3_OUT&=0xef;
        SPID_DATA=0X02; //发送写操作码，WRITE 0X02
        while((SPID_STATUS&0x80)!=0x80);
        SPI_WRITE_ADDRESS();//数据移位完成，写3个字节的地址
        for(uint8_t i=0;i<receivedata[4];i++)
         {
         SPID_DATA=receivedata[i+5];
         while((SPID_STATUS&0x80)!=0x80);
         }
         VPORT3_OUT|=0x10;//写操作完成，片选SS拉高
         VPORT0_OUT&=0xFB;//灯灭
}
 //---------SPI_WRITE_ADDRESS (SPI写三个字节地址)--------//
void SPI_WRITE_ADDRESS(void)
{
        for(uint8_t i=0;i<3;i++)
        {
        SPID_DATA=receivedata[i+1];
        while((SPID_STATUS&0x80)!=0x80);
        }
}
 //---------main(主函数入口)--------//
ISR(USARTC0_RXC_vect)
{
       int count_num=0;
       VPORT0_OUT|=0x10;//接收过程亮灯
       receivedata[count_num]=USARTC0_DATA;//读Data Register
       TCC0_CNT=0;
       while(TCC0_CNT<=20) //接收间隔大于20，接收结束
      {
        if((USARTC0.STATUS&0x80)==0x80)
        {
        count_num++;
```

```
        receivedata[count_num]=USARTC0_DATA;//读Data Register
        TCC0_CNT=0;
         }
        }
        VPORT0_OUT&=0xef;//接收结束灯灭
        if(receivedata[0]==0xf9)//自定义SPI写命令
        SPI_WRITE_DATA();
        if(receivedata[0]==0xfa)//自定义SPI读命令
        SPI_READ_DATA();

}
ISR(USARTC0_TXC_vect) //串口发送中断，作用可以是发送标志位置位
{}
```

汇编代码：

```
//--------包含头文件--------//
.include "Atxmega32A4def.inc"//器件配置文件,决不可少,不然汇编通不过
.ORG 0
        RJMP RESET//复位
.ORG 0x032
        JMP USARTC0_INT_RXC//跳到串口C0接收完毕中断子程序

.ORG 0X100        //跳过中断区0x00-0x0F4
//--------宏定义--------//
.MACRO CLKSYS_IsReady
 CLKSYS_IsReady_1:
           LDS R16,OSC_STATUS
           SBRS R16,@0
           JMP CLKSYS_IsReady_1//等待外部振荡器准备好
           NOP
.ENDMACRO
//--------PLL_XOSC_Initial (时钟初始化)--------//
PLL_XOSC_Initial:
       LDI R16,0X10
       STS PORTCFG_VPCTRLA,R16;PORTB映射到虚拟端口1，PORTA映射到虚拟端口0
       LDI R16,0X32
       STS PORTCFG_VPCTRLB,R16;PORTC映射到虚拟端口2，PORTD映射到虚拟端口3
       LDI R16,0X4B
       STS OSC_XOSCCTRL,R16//设置晶振范围 启动时间
       LDI R16,OSC_XOSCEN_bm
       STS OSC_CTRL,R16//使能外部振荡器
       CLKSYS_IsReady OSC_XOSCRDY_bp
       LDI R16,OSC_PLLSRC_XOSC_gc
       ORI R16,0XC3
       STS OSC_PLLCTRL,R16
```

```
        LDS R16,OSC_CTRL//读取该寄存器的值到R16
        SBR R16,OSC_PLLEN_bm//对PLLEN这一位置位，使能PLL
        STS OSC_CTRL,R16
        CLKSYS_IsReady OSC_PLLRDY_bp
        LDI R16,CLK_SCLKSEL_PLL_gc
        LDI R17,0XD8//密钥
        STS CPU_CCP,R17//解锁
        STS CLK_CTRL,R16//选择系统时钟源
        LDI R16,CLK_PSADIV_1_gc
        ORI R16,CLK_PSBCDIV_1_1_gc
        LDI R17,0XD8//密钥
        STS CPU_CCP,R17//解锁
        STS CLK_PSCTRL,R16//设置预分频器A,B,C的值
        RET
//---------SPI_MasterInit (SPI初始化)--------//
SPI_MasterInit:
         SBI VPORT3_DIR,4 //SS片选引脚设为方向输出
         SBI VPORT3_DIR,5//MOSI引脚设为方向输出
         SBI VPORT3_DIR,7//SCK引脚设为方向输出
         LDI R16,0x50//0,1,0,1,00,00;SPI Clock Double DISABLE,SPI module Enable,
Data Order=MSB，Master Select, SPI Mode=0, SPI Clock Prescaler=CLKper/4=0.5MHZ
         STS SPID_CTRL,R16//SPI控制寄存器
         LDI R16,0x00
         STS SPID_INTCTRL,R16//SPI 中断 关闭
         RET
//---------uart_init (串口初始化)--------//
uart_init:
        /* USARTC0 引脚方向设置*/
         LDI R16,0X08 //PC3 (TXD0) 输出
         STS PORTC_DIRSET,R16
         LDI R16,0X04//PC2 (RXD0) 输入
         STS PORTC_DIRCLR,R16
        /* USARTC0 模式 - 异步 USARTC0帧结构, 8 位数据位, 无校验, 1停止位 */
         LDI R16,USART_CMODE_ASYNCHRONOUS_gc|USART_CHSIZE_8BIT_gc
                                            |USART_PMODE_DISABLED_gc
         STS USARTC0_CTRLC,R16
         LDI R16,77//设置波特率19200
         STS USARTC0_BAUDCTRLA,R16
         LDI R16,0
         STS USARTC0_BAUDCTRLB,R16
         //000:Reserved,1:接收使能,1:发送使能,0:波特率不加倍,0:单机模式，0:TXB8
不使用
         LDI R16,0X18
         STS USARTC0_CTRLB,R16
```

```
        LDI R16,0X30
        STS USARTC0_CTRLA,R16
        RET
//--------RESET(主函数入口)--------//
RESET:
        CALL PLL_XOSC_Initial
        CALL SPI_MasterInit
        CALL uart_init
        SBI VPORT3_DIR,0//PORTD0输出
        CBI VPORT3_OUT,0//PORTD0输出低 使能485接收
        //指示灯
        SBI VPORT0_DIR,2//PORTA2输出
        SBI VPORT0_DIR,3//PORTA3输出
        SBI VPORT0_DIR,4//PORTA4输出

TimerC0:
        //2.selecting a clock source
        LDI R16,0X07//计数器时钟源为24MHZ/1024=23437.5Hz
        STS TCC0_CTRLA,R16
//可编程多层中断控制寄存器高 中 低层使能，循环调度关闭，中断向量未移至 Boot
section
        LDI R16,PMIC_HILVLEN_bm + PMIC_MEDLVLEN_bm + PMIC_LOLVLEN_bm;
        STS PMIC_CTRL,R16
        SEI//全局中断使能置位
LOOP:
        NOP
        CBI VPORT3_OUT,0//PORTD0输出低 使能485接收
        JMP LOOP
 //---------SPI_READ_DATA (SPI读数据流程)--------//
SPI_READ_DATA:

        SBI VPORT3_OUT,4//发送操作码之前片选产生下降沿
        NOP
        CBI VPORT3_OUT,4

        LDI R16,0X03 //发送读操作码 READ 0X03
        STS SPID_DATA,R16
        CALL SPI_WAITING_IF
        CALL SPI_WRITE_ADDRESS//数据移位完成，写3个字节的地址
        LD  R17,Y+
LOOP_READ_DATA:
        LDI  R16,0X00
        STS SPID_DATA,R16
        CALL SPI_WAITING_IF
```

```
        LDS R16,SPID_DATA //读入数据移位完成,取出数据
        ST Y+,R16

        DEC R17//计数减1，读下一个字节的数据
        BRNE LOOP_READ_DATA
        SBI VPORT3_OUT,4//读操作完成，片选SS拉高
        RET
//--------SPI_WRITE_DATA (SPI写数据流程)--------//
SPI_WRITE_DATA:

        SBI VPORT3_OUT,4 //发送操作码之前片选产生下降沿
        NOP
        CBI VPORT3_OUT,4

        LDI R16,0X06 //发送写使能操作码WREN 0X06
        STS SPID_DATA,R16
        CALL SPI_WAITING_IF
        SBI VPORT3_OUT,4//发送写使能操作码 完成，片选SS拉高
        SBI VPORT3_OUT,4//发送操作码之前片选产生下降沿
        NOP
        CBI VPORT3_OUT,4

        LDI R16,0X02 //发送写操作码 WRITE 0X02
        STS SPID_DATA,R16
        CALL SPI_WAITING_IF
        CALL SPI_WRITE_ADDRESS //数据移位完成，写3个字节的地址
        LD  R17,Y+
LOOP_WRITE_DATA:
        LD  R16,Y+
        STS SPID_DATA,R16
        CALL SPI_WAITING_IF
        //数据移位完成，计数减1，写下一个字节的数据
        DEC R17
        BRNE LOOP_WRITE_DATA
        SBI VPORT3_OUT,4//写操作完成，片选SS拉高
        RET
 //--------SPI_WAITING_IF--------//
SPI_WAITING_IF: //读写数据都要等待SPI移位完成标志位置位
        LDS R16,SPID_STATUS
        SBRS R16,7//Skip if bit 7(IF) in SPID_STATUS set
        RJMP SPI_WAITING_IF//数据移位没有完成继续等待
        RET
//--------SPI_WRITE_ADDRESS --------//
SPI_WRITE_ADDRESS: //读写数据都要写三个字节的地址
```

```
        LDI R17,0X03
        LOOP_ADDRESS:
        LD  R16,X+
        STS SPID_DATA,R16
        CALL SPI_WAITING_IF
        DEC R17//数据移位完成，计数减1，写下一个字节的地址
        BRNE LOOP_ADDRESS
        RET
//---------USARTC0_INT_RXC（串口接收中断）--------//
USARTC0_INT_RXC:
        LDI R18,0X00//记录接收字节数
        LDI R31,0x20//串口数据起始储存地址Z=0x2000
        LDI R30,0x00
        SBI VPORT0_OUT,4//接收过程亮灯
STORE_DATA:
        LDS R16,USARTC0_DATA//读Data Register
        ST  Z+,R16
        INC R18//接收字符数加1
        LDI R16,0X00//计数器清零
        STS TCC0_CNT,R16
        STS TCC0_CNT+1,R16
WAITING_RECEIVE:
        LDS R16,TCC0_CNT
        CPI R16,20
        BRSH RECEIVE_END //接收间隔大于20，接收结束
        LDS R16,USARTC0_STATUS
        SBRS R16,7 //有数据来了就存
        JMP WAITING_RECEIVE
        JMP STORE_DATA
RECEIVE_END:
        CBI VPORT0_OUT,4//接收结束灯灭
        //X指针赋地址
        LDI R27,0X20
        LDI R26,0X00
        LD  R16,X+
        CPI R16,0XF9 //自定义SPI写命令
        BREQ SPI_WRITE
        CPI R16,0XFA //自定义SPI读命令
        BREQ SPI_READ
SPI_WRITE:
        /*X指针指向的数据区存储三个字节的地址
          写的数据存在Y指针指向的数据区域，第1个字节表明字节数目，第2个字节及其后
          是需要写的数据*/
        SBI VPORT0_OUT,3//SPI开始写灯亮
```

```
        LDI R27,0X20//X指针赋地址,SPI地址存储开始
        LDI R26,0X01
        //Y指针赋地址,SPI存储数据开始地址
        LDI R29,0X20
        LDI R28,0X04
        CALL SPI_WRITE_DATA
        CBI VPORT0_OUT,3//SPI结束写灯灭
        RETI
SPI_READ:
        /*X指针指向的数据区存储三个字节的地址
        读的数据存在Y指针指向的数据区域，第1个字节事先指明读取的字节数目，
        第2个字节之后是存放读出的数据*/
        SBI VPORT0_OUT,2//SPI开始读灯亮
        //X指针赋地址,SPI读取地址
        LDI R27,0X20
        LDI R26,0X01
        //Y指针赋地址,SPI读取数据个数
        LDI R29,0X20
        LDI R28,0X04
        CALL SPI_READ_DATA
        CBI VPORT0_OUT,2//SPI结束读灯灭
        //Y指针赋地址,SPI读取数据个数
        LDI R29,0X20
        LDI R28,0X04
        LD  R25,Y+//数据个数
        SBI VPORT3_OUT,0//使能485发送,为数据可以发送到串口助手
        SBI VPORT0_OUT,4//发送过程亮灯
AA_C0:
        LD  R16,Y+
        STS USARTC0_DATA,R16//串口C0发送
BB_C0:
        LDS R16,USARTC0_STATUS
        SBRS    R16,5
        RJMP    BB_C0
        DEC     R25
        CALL DELAY
        CPI     R25,0//检查字节数是否全部发送完成
        BRNE AA_C0
        CBI VPORT0_OUT,4//接收结束灯灭
        RETI
//--------——DELAY（延迟函数）--------//
DELAY:
        LDI R18,0X55
TT:
```

```
        LDI R19,0XFF
HH:
        CPI R19,0
        DEC R19
        BRNE HH
        DEC R18
        CPI R18,0
        BRNE TT
        RET
```

5.8 EEPROM 实例

Flash 存储器是按页更新的。EEPROM 可以是按字节或按页更新。Flash 和 EEPROM 页编程的步骤是：首先加载页缓存，然后将缓存写到选定的 Flash 或 EEPROM 的页。

EEPROM 页缓存每次载入一个字节，载入前必须先擦除。如果在已经加载过缓存的位置处再次写入，会破坏原来的内容。

EEPROM 页缓存在以下操作后自动清除:

- ✓ 系统复位
- ✓ 执行写 EEPROM 页命令
- ✓ 执行擦除-写 EEPROM 页命令
- ✓ 执行写锁定位和熔丝位命令

1. 片内EEPROM的读写操作。EEPROM的读写有四种方法：1.EEPROM不映射到RAM,按字节写入；2.EEPROM不映射到RAM,按页写入；3.EEPROM映射到RAM,按字节写入；4.EEPROM映射到RAM,按页写入；USARTC0负责打印调试信息。

C语言代码：

```
//--------包含头文件--------//
#include "avr_compiler.h"
#include "usart_driver.c"
#include "eeprom_driver.c"
//--------宏定义--------//
#define TEST_BYTE_1 0x55
#define TEST_BYTE_2 0xAA
#define TEST_BYTE_ADDR_1 0x00
#define TEST_BYTE_ADDR_2 0x08
#define TEST_PAGE_ADDR_1    0  //页地址总是在页的边界
#define TEST_PAGE_ADDR_2    2  //页地址总是在页的边界
#define TEST_PAGE_ADDR_3    5  //页地址总是在页的边界
//写入EEPROM的缓存数据
uint8_t testBuffer[EEPROM_PAGESIZE] = {"Accessing Atmel AVR XMEGA EEPROM"};
//--------uart_init（串口初始化）--------//
void uart_init(void)
{
    /* USARTC0 引脚方向设置*/
```

```
    PORTC.DIRSET   = PIN3_bm; //PC3 (TXD0) 输出
    PORTC.DIRCLR   = PIN2_bm; // PC2 (RXD0) 输入
    USART_SetMode(&USARTC0,USART_CMODE_ASYNCHRONOUS_gc); //USARTC0 模式 - 异步
  // USARTC0帧结构, 8 位数据位, 无校验, 1停止位
    USART_Format_Set(&USARTC0, USART_CHSIZE_8BIT_gc,USART_PMODE_DISABLED_gc,
false);
    USART_Baudrate_Set(&USARTC0, 12 , 0); //设置波特率 9600
    USART_Tx_Enable(&USARTC0); //USARTC0 使能发送
    USART_Rx_Enable(&USARTC0); //USARTC0 使能接收
}
//---------main（主函数入口）--------//
int main( void )
{
    uart_init();
    uart_puts("inside main");uart_putc('\n');
    bool test_ok = true; //测试结果
    uint8_t temp = 0;
    EEPROM_FlushBuffer(); //清空缓存
    //1.使用IO寄存器方式写入和读取2个字节
    EEPROM_DisableMapping(); //关闭EEPROM映射到内存空间
    //写单个字节
    EEPROM_WriteByte(TEST_PAGE_ADDR_1, TEST_BYTE_ADDR_1, TEST_BYTE_1);
    EEPROM_WriteByte(TEST_PAGE_ADDR_1, TEST_BYTE_ADDR_2, TEST_BYTE_2);
    temp = EEPROM_ReadByte(TEST_PAGE_ADDR_1, TEST_BYTE_ADDR_1); //读取写入的字节
    if ( temp != TEST_BYTE_1)
    {
        test_ok = false;
    }
    uart_puts("1 Read TEST_BYTE_1 = ");
    uart_putc_hex(temp);
    uart_putc('\n');
    temp = EEPROM_ReadByte(TEST_PAGE_ADDR_1, TEST_BYTE_ADDR_2);
    if ( temp != TEST_BYTE_2)
    {
        test_ok = false;
    }
    uart_puts("1 Read TEST_BYTE_2 = ");
    uart_putc_hex(temp);
    uart_putc('\n');
    //2.分离操作写一整页
    EEPROM_LoadPage(testBuffer); //加载页缓存，先擦页后写页
    EEPROM_ErasePage(TEST_PAGE_ADDR_2);
    EEPROM_SplitWritePage(TEST_PAGE_ADDR_2);
    uart_puts("2 Read testBuffer = ");//读取数据
```

```
uart_putc('\n');
for (uint8_t i = 0; i < EEPROM_PAGESIZE; ++i)
{
    temp = EEPROM_ReadByte(TEST_PAGE_ADDR_2, i );
    if ( temp != testBuffer[i] )
    {
        test_ok = false;
        break;
    }
    uart_putc(temp);
}
uart_putc('\n');
//3.使用内存映射写入并读取两个字节
EEPROM_EnableMapping();
EEPROM_WaitForNVM(); //写2个字节
EEPROM(TEST_PAGE_ADDR_1, TEST_BYTE_ADDR_1) = TEST_BYTE_1;
EEPROM_WaitForNVM();
EEPROM(TEST_PAGE_ADDR_1, TEST_BYTE_ADDR_2) = TEST_BYTE_2;
EEPROM_WaitForNVM(); //读取2个字节
temp = EEPROM(TEST_PAGE_ADDR_1, TEST_BYTE_ADDR_1);
if ( temp != TEST_BYTE_1)
{
    test_ok = false;
}
uart_puts("3 Read TEST_BYTE_1 = ");
uart_putc_hex(temp);
uart_putc('\n');
temp = EEPROM(TEST_PAGE_ADDR_1, TEST_BYTE_ADDR_2);
if ( temp != TEST_BYTE_2)
{
    test_ok = false;
}
uart_puts("3 Read TEST_BYTE_2 = ");
uart_putc_hex(temp);
uart_putc('\n');
//4.使用内存映射方式，分离操作，写一页缓存到EEPROM
EEPROM_EnableMapping();
//加载缓存
EEPROM_WaitForNVM();
for (uint8_t i = 0; i < EEPROM_PAGESIZE; ++i)
{
    EEPROM(TEST_PAGE_ADDR_3, i) = testBuffer[i];
}
//擦除EEPROM页，页缓存并没有被擦除
```

```
    EEPROM_ErasePage(TEST_PAGE_ADDR_3);
    //分离写操作，缓存在操作完成后清空
    EEPROM_SplitWritePage(TEST_PAGE_ADDR_3);
    EEPROM_WaitForNVM();//读取数据，串口显示
    uart_puts("4 Read testBuffer = ");
    uart_putc('\n');
    for (uint8_t i = 0; i < EEPROM_PAGESIZE; ++i)
    {
        temp = EEPROM(TEST_PAGE_ADDR_3, i);
        if ( temp!= testBuffer[i] )
        {
            test_ok = false;
            break;
            uart_puts("test_ok = false");
            uart_putc('\n');
        }
        uart_putc(temp);
    }
    uart_putc('\n');
    while(1)
        {
            nop();
        }
}
```

汇编代码：

```
//---------包含头文件--------//
.include "ATxmega128A1def.inc";器件配置文件,决不可少,不然汇编通不过
.include "usart_driver.inc"
.ORG 0
        RJMP RESET//复位
.ORG 0X100        ;跳过中断区0x00-0x0FF
.EQU ENTER  ='\n'
.EQU NEWLINE='\r'
.EQU EQUE   ='='
.EQU ZERO   ='0'
.EQU A_ASSCI='A'
.equ TEST_BYTE_1=0x55
.equ TEST_BYTE_2=0xAA
.equ TEST_BYTE_ADDR_1=0x00
.equ TEST_BYTE_ADDR_2=0x08
.equ TEST_PAGE_ADDR_1=0  //页地址总是在页的边界
.equ TEST_PAGE_ADDR_2=2  //页地址总是在页的边界
.equ TEST_PAGE_ADDR_3=5  //页地址总是在页的边界
//写入EEPROM的缓存数据
```

```
MAIN: .DB 'I','N','T','O','M','A','I','N',0
testBuffer: .db
'A','c','c','e','s','s','i','n','A','c','c','e','s','s','i','n','A','c','c','e',
's','s','i','n','A','c','c','e','s','s','i',0
//--------uart_init（串口初始化）--------//
uart_init:
        /* USARTC0 引脚方向设置*/
         LDI R16,0X08//PC3 (TXD0) 输出
         STS PORTC_DIRSET,R16
         LDI R16,0X04 //PC2 (RXD0) 输入
         STS PORTC_DIRCLR,R16
        //USARTC0 模式 - 异步 USARTC0帧结构, 8 位数据位, 无校验, 1停止位
         LDIR16,USART_CMODE_ASYNCHRONOUS_gc|USART_CHSIZE_8BIT_gc
                                           |USART_PMODE_DISABLED_gc
         STS USARTC0_CTRLC,R16
         LDI R16,12 //设置波特率 9600
         STS USARTC0_BAUDCTRLA,R16
         LDI R16,0
         STS USARTC0_BAUDCTRLB,R16
         LDI R16,USART_TXEN_bm // USARTC0 使能发送
         STS USARTC0_CTRLB,R16
         RET
//--------NVM_EXEC（非易失性存储控制器执行命令）--------//
NVM_EXEC:
    push r30
    push r31
    push r16
    push r18
    ldi r30, 0xCB
    ldi r31, 0x01
    ldi r16, 0xD8
    ldi r18, 0x01
    out 0x34, r16
    st Z, r18
     pop r18
     pop r16
     pop r31
     pop r30
     ret
//--------EEPROM_FlushBuffer（BUFFER擦除）--------//
EEPROM_FlushBuffer:
    call EEPROM_WaitForNVM
    nop
EEPROM_FlushBuffer_1:
```

```
    LDS r16,NVM_STATUS
    SBRC r16,NVM_EELOAD_bp
    jmp EEPROM_FlushBuffer_1
    nop
    ldi r16,NVM_CMD_ERASE_EEPROM_BUFFER_gc
    sts NVM_CMD,r16
    call NVM_EXEC
    ret
//---------EEPROM_WaitForNVM （等待NVM不忙）--------//
EEPROM_WaitForNVM:
EEPROM_WaitForNVM_1:
    lds r16,NVM_STATUS
    sbrc r16,NVM_NVMBUSY_bp
    JMP EEPROM_WaitForNVM_1
    nop
    ret
//---------宏定义（读取给定地址的eeprom数据）--------//
.MACRO EEPROM_ReadByte
    call EEPROM_WaitForNVM
    pop r16
    pop r17
    pop r18
    //写地址
    sts NVM_ADDR0,r16
    sts NVM_ADDR1,r17
    sts NVM_ADDR2,r18
    //执行读命令
    ldi r16,NVM_CMD_READ_EEPROM_gc
    sts NVM_CMD,r16
    call NVM_EXEC
    lds xh,NVM_DATA0
    PUSH XH
    uart_putc
.ENDMACRO
//---------EEPROM_WriteByte（写EEPROMbuffer）--------//
.MACRO EEPROM_WriteByte
    call EEPROM_FlushBuffer
    pop r16
    pop r17
    pop r18
    pop r19
    LDI r20,NVM_CMD_LOAD_EEPROM_BUFFER_gc
    sts NVM_CMD,r20
    //写地址
```

```
    sts NVM_ADDR0,r17
    sts NVM_ADDR1,r18
    sts NVM_ADDR2,r19
    //加载数据触发命令执行
     sts NVM_DATA0,r16
    //触发原子操作（擦&写）写签名，执行命令
     ldi r16,NVM_CMD_ERASE_WRITE_EEPROM_PAGE_gc
     sts NVM_CMD,r16
     call NVM_EXEC
.ENDMACRO
//---------EEPROM_ErasePage（擦除EEPROM）--------//
.MACRO EEPROM_ErasePage
    call EEPROM_WaitForNVM//等待NVM不忙
    pop r16
    pop r17
    pop r18
    sts NVM_ADDR0,r16
    sts NVM_ADDR2,r18
    sts NVM_ADDR1,r17
    ldi r16,NVM_CMD_ERASE_EEPROM_PAGE_gc
    sts NVM_CMD,r16
    call NVM_EXEC
.ENDMACRO
//---------RESET（主函数入口）--------//
RESET:
    call uart_init
    LDI R16,ENTER//回车换行
    PUSH R16
    uart_putc
    uart_puts_string  MAIN;uart_putc('\n');
    LDI R16,ENTER
    PUSH R16
    uart_putc
    call EEPROM_FlushBuffer//清空缓存
    ldi r16,~NVM_EEMAPEN_bm//关闭EEPROM映射到内存空间
    sts NVM_CTRLB,r16
    ldi r16,0x00
    push r16
    push r16
    push r16
    ldi r16,0x61
    push r16
    EEPROM_WriteByte //0x00 0x00 0x00 0x61
    ldi r16,0x00
```

```
    push r16
    push r16
    ldi r16,0x08
    push r16
    ldi r16,0x62
    push r16
    EEPROM_WriteByte //0x00 0x00 0x08 0x62
    ldi r16,0x00
    push r16
    push r16
    push r16
    EEPROM_ReadByte  //读取写入的字节
    ldi r16,0x00
    push r16
    push r16
    ldi r16,0x08
    push r16
    EEPROM_ReadByte
    LDI R16,ENTER //回车换行
    PUSH R16
    uart_putc
    //2.分离操作写一整页
    call EEPROM_WaitForNVM//加载页缓存，先擦页后写页
    ldi r16,NVM_CMD_LOAD_EEPROM_BUFFER_gc
    sts NVM_CMD,r16
    //地址清零，只有低几位使用，在循环内改变
    LDI r16,0x00
    STS NVM_ADDR1,r16
    LDI r16,0x00
    sts NVM_ADDR2,r16
    //加载多个字节到缓存
    eor r20,r20
    LDI ZH,HIGH(testBuffer<<1)
    LDI ZL,LOW(testBuffer<<1)
RESET_1:
    sts NVM_ADDR0,r20
    inc r20
    LPM R16,Z+
    sts NVM_DATA0,r16
    clz
    cpi r16,0
    brne RESET_1
    ldi r16,0x00
    push r16
```

```
    ldi r16,0x02
    push r16
    ldi r16,0x00
    push r16
    EEPROM_ErasePage
        //等待NVM不忙
    call EEPROM_WaitForNVM
    //写地址
    ldi r16,0x00
    sts NVM_ADDR0,r16
    sts NVM_ADDR2,r16
    ldi r16,0x02
    sts NVM_ADDR1,r16
    //触发写EEPROM页命令
    ldi r16,NVM_CMD_WRITE_EEPROM_PAGE_gc
    sts NVM_CMD,r16
    call NVM_EXEC
    eor r20,r20
RESET_4:
    ldi r16,0x00
    push r16
    ldi r16,0x02
    push r16
    mov r16,r20
    INC R20
    push r16
    EEPROM_ReadByte
    CPI XH,0
    BRNE RESET_4
    LDI R16,ENTER//回车换行
    PUSH R16
    uart_putc
    //3 EEPROM映射到内存空间
    ldi r16,NVM_EEMAPEN_bm
    sts NVM_CTRLB,r16
    //写2个字节
    call EEPROM_WaitForNVM
    ldi zh,0x10
    ldi zl,0x00
    ldi r16,0xdc
    st z+,r16
    call EEPROM_WaitForNVM
    ldi r16,0x00
    sts NVM_ADDR0,r16
```

```
    sts NVM_ADDR2,r16
    ldi r16,0x10
    sts NVM_ADDR1,r16
    ldi r16,NVM_CMD_ERASE_WRITE_EEPROM_PAGE_gc
    sts NVM_CMD,r16
    call NVM_EXEC
    call EEPROM_WaitForNVM
    ldi r16,0xaa
    st z,r16
    ldi r16,0x01
    sts NVM_ADDR0,r16
    ldi r16,0x00
    sts NVM_ADDR2,r16
    ldi r16,0x10
    sts NVM_ADDR1,r16
    ldi r16,NVM_CMD_ERASE_WRITE_EEPROM_PAGE_gc
    sts NVM_CMD,r16
    call NVM_EXEC
    //读取2个字节
    call EEPROM_WaitForNVM
    ldi zh,0x10
    ldi zl,0x00
    ld xh,z+
    ld xl,z
    uart_putw_hex
    LDI R16,ENTER//回车换行
    PUSH R16
    uart_putc
    //4.使用内存映射方式，分离操作，写一页缓存到EEPROM
    //地址清零，只有低几位使用，再循环内会改变
    ldi r16,NVM_EEMAPEN_bm
    sts NVM_CTRLB,r16
    ldi r16,NVM_CMD_LOAD_EEPROM_BUFFER_gc
    sts NVM_CMD,r16
    call EEPROM_WaitForNVM
    //加载多个字节到缓存
    LDI ZH,HIGH(testBuffer<<1)
    LDI ZL,LOW(testBuffer<<1)
    ldi xh,0x13
    ldi xl,0x00
RESET_2:
    LPM R16,Z+
    st x+,r16
    clz
```

```
    cpi r16,0
    brne RESET_2
    ldi r16,0x00
    push r16
    ldi r16,0x03
    push r16
    ldi r16,0x00
    push r16
    EEPROM_ErasePage
    call EEPROM_WaitForNVM//等待NVM不忙
    //写地址
    ldi r16,0x00
    sts NVM_ADDR0,r16
    ldi r16,0x00
    sts NVM_ADDR2,r16
    ldi r16,0x13
    sts NVM_ADDR1,r16
    //触发写EEPROM页命令
    ldi r16,NVM_CMD_WRITE_EEPROM_PAGE_gc
    sts NVM_CMD,r16
    call NVM_EXEC
    call EEPROM_WaitForNVM
    eor r20,r20
    ldi zh,0x13
    ldi zl,0x00
RESET_5:
    ld xh,z+
    PUSH XH
    uart_putc
    CPI XH,0
    BRNE RESET_5
RESET_3:
    JMP RESET_3
```

5.9 WDT - 看门狗定时器实例

看门狗定时器（WDT）是用来监控程序的正确执行的，使系统可以从错误状态中恢复，如：代码跑飞。WDT 是个定时器，用预先设定的超时周期配置，启动后会不断运行。如果 WDT 在超时周期内没有清零，它会触发系统复位。WDT 是通过执行 WDR 指令实现复位的。

WDT 在窗口模式下可以让使用者定义一个时隙，WDT 必须在时隙内复位。如果 WDT 复位太早或太晚，系统会触发复位操作。

WDT 可以在所有电源模式下运行。它从独立于 CPU 的时钟源运行，即使系统时钟不可用时，它仍能触发系统复位。

配置更改保护机制可以确保WDT设置不会随意改变。另外看门狗设置可以通过熔丝位锁定。1.看门狗定时器的使用。

看门狗有两种模式：正常模式，窗口模式。程序中有两个函数：

wdt_sw_enable_example函数是在熔丝位没有置位的情况下对看门狗进行两种模式的设置，在这种情况下可以通过代码对看门狗正常超时周期和窗口超时周期进行设置；

wdt_fuse_enable_example函数是在熔丝位置位的情况下，在这种情况下正常模式已经使能，窗口模式需要代码使能，但是两种模式的周期都不能改变，由熔丝位启动时自动加载。

熔丝位置位在软件中设置不需要代码置位。

C语言代码：

```
//--------包含头文件--------//
#include "avr_compiler.h"
#include "usart_driver.c"
#include "wdt_driver.c"
#define F_CPU (2000000UL) //定义CPU时钟
void wdt_fuse_enable_example( void );
void wdt_sw_enable_example( void );
#define TO_WD     128//看门狗超时周期毫秒数
#define TO_WDW    64//看门狗关闭周期毫秒数
//窗口模式下看门狗重置间隔时间
#define WINDOW_MODE_DELAY   ( (TO_WDW) +  (TO_WD / 2) )
//普通模式下看门狗重置间隔时间
#define NORMAL_MODE_DELAY   ( TO_WD / 2 )
//--------uart_init（串口初始化）--------//
void uart_init(void)
{
    /* USARTC0 引脚方向设置*/
    PORTC.DIRSET   = PIN3_bm; //PC3 (TXD0) 输出
    PORTC.DIRCLR   = PIN2_bm; // PC2 (RXD0) 输入
    USART_SetMode(&USARTC0,USART_CMODE_ASYNCHRONOUS_gc); //USARTC0 模式 - 异步
  // USARTC0帧结构, 8 位数据位, 无校验, 1停止位
    USART_Format_Set(&USARTC0, USART_CHSIZE_8BIT_gc,USART_PMODE_DISABLED_gc,
false);
    USART_Baudrate_Set(&USARTC0, 12 , 0); //设置波特率 9600
    USART_Tx_Enable(&USARTC0); //USARTC0 使能发送
    USART_Rx_Enable(&USARTC0); //USARTC0 使能接收
}
//--------main（主函数入口）--------//
int main( void )
{
    uart_init();
    uart_puts("inside main");uart_putc('\n');
    //wdt_sw_enable_example();
    wdt_fuse_enable_example();
```

```
}

//--------——wdt_sw_enable_example（WDLOCK熔丝位没有置位）--------//
void wdt_sw_enable_example( void )
{
    uart_puts("inside void wdt_sw_enable_example( void )");uart_putc('\n');
    //使能看门狗普通模式，超时周期为32CLK=32 ms
    WDT_EnableAndSetTimeout( WDT_PER_32CLK_gc );
    if(WDT.CTRL & WDT_ENABLE_bm)
    {
        uart_puts("WDT Is  Enabled");uart_putc('\n');
    }
    else
    {
        uart_puts("WDT Is not Enabled");uart_putc('\n');
    }
    //如果需要重新配置看门狗需要先关闭看门狗
    WDT_Disable();
    if(WDT.CTRL & WDT_ENABLE_bm)
    {
        uart_puts("WDT Is  Enabled");uart_putc('\n');
    }
    else
    {
        uart_puts("WDT Is not Enabled");uart_putc('\n');
    }
    if(WDT_IsWindowModeEnabled())
    {
        uart_puts("WDT Is WindowMode Enabled");uart_putc('\n');
    }
    else
    {
        uart_puts("WDT Is not WindowMode Enabled");uart_putc('\n');
    }
    /*在配置窗口模式前需要进行新的看门狗普通模式配置，
      因为窗口模式只有在普通模式使能时才能开启*/
    WDT_EnableAndSetTimeout( WDT_PER_128CLK_gc );
    WDT_EnableWindowModeAndSetTimeout( WDT_WPER_64CLK_gc ); //配置窗口模式
    if(WDT_IsWindowModeEnabled())
    {
        uart_puts("WDT Is WindowMode Enabled");uart_putc('\n');
    }
    else
    {
```

```
        uart_puts("WDT Is not WindowMode Enabled");uart_putc('\n');
    }
    while(true)
    {
        uint16_t repeatCounter;
        //确保不要过早重置看门狗，否则会导致系统复位，插入延时要大于开放的窗口周
期
        for (repeatCounter = WINDOW_MODE_DELAY; repeatCounter > 0;
--repeatCounter )
        {
            delay_us( 1000 ); // 1ms 延迟 @ 2MHz
        }
        WDT_Reset();//重置看门狗
    }
}
//---------wdt_fuse_enable_example（WDLOCK熔丝位置位，周期通过熔丝位设定）
--------//
void wdt_fuse_enable_example( void )
{
    uart_puts("inside void wdt_fuse_enable_example( void )");uart_putc('\n');
    //当看门狗通过熔丝位设定后，需要重置看门狗
    WDT_Reset();
    if(WDT.CTRL & WDT_ENABLE_bm)
    {
        uart_puts("WDT Is  Enabled");uart_putc('\n');
    }
    else
    {
        uart_puts("WDT Is not Enabled");uart_putc('\n');
    }
    if(WDT_IsWindowModeEnabled())
    {
        uart_puts("WDT Is WindowMode Enabled");uart_putc('\n');
    }
    else
    {
        uart_puts("WDT Is not WindowMode Enabled");uart_putc('\n');
    }
    if (true == WDT_IsWindowModeEnabled()){
        while(1){
            //如果窗口模式使能，系统将复位
        }
    }
    /*如果窗口模式未通过熔丝使能，需要在程序中使能，但是周期数不能设定，因为熔丝
```

```
    位已经置位*/
    WDT_EnableWindowMode();
    // 看门狗重置最好在(TO_WDW * 1.3) 和((TO_WDW + TO_WD)*0.7) 之间
    delay_us( 90000 );
    WDT_Reset();
    if(WDT_IsWindowModeEnabled())
    {
        uart_puts("WDT Is WindowMode Enabled");uart_putc('\n');
    }
    else
    {
        uart_puts("WDT Is not WindowMode Enabled");uart_putc('\n');
    }
    WDT_DisableWindowMode();
    if(WDT_IsWindowModeEnabled())
    {
        uart_puts("WDT Is WindowMode Enabled");uart_putc('\n');
    }
    else
    {
        uart_puts("WDT Is not WindowMode Enabled");uart_putc('\n');
    }
    while(true)
    {
        uint16_t repeatCounter;
        for (repeatCounter = NORMAL_MODE_DELAY; repeatCounter > 0;
--repeatCounter ) {
            delay_us( 1000 );
        }
        WDT_Reset();
    }
}
```

汇编代码：

```
//---------包含头文件--------//
.include "ATxmega128A1def.inc";器件配置文件,决不可少,不然汇编通不过
.include "usart_driver.inc"
.ORG 0
        RJMP RESET
.ORG 0X100        ;跳过中断区0x00-0x0FF
.EQU ENTER  ='\n'
.EQU NEWLINE='\r'
.EQU EQUE   ='='
.EQU ZERO   ='0'
.EQU A_ASSCI='A'
```

```
.equ TO_WD=128//看门狗超时周期毫秒数
.equ TO_WDW=64//看门狗关闭周期毫秒数
.equ WINDOW_MODE_DELAY=128//窗口模式下看门狗重置间隔时间
.equ NORMAL_MODE_DELAY=64 //普通模式下看门狗重置间隔时间
MAIN: .DB 'I','N','T','O','M','A','I','N',0
sw: .db 'I','N','T','O','s','w',0
fuse: .db 'I','N','T','O','f','u','s','e',0
WDTenabled: .db  'w','d','T','e','n','a','b','l','e','d',0
WDTunenabled: .db  'w','d','T','u','n','e','n','a','b','l','e','d',0
//--------uart_init（串口初始化）--------//
uart_init:
        /* USARTC0 引脚方向设置*/
         LDI R16,0X08//PC3 (TXD0) 输出
         STS PORTC_DIRSET,R16
         LDI R16,0X04 //PC2 (RXD0) 输入
         STS PORTC_DIRCLR,R16
        //USARTC0 模式 - 异步 USARTC0帧结构, 8 位数据位, 无校验, 1停止位
         LDIR16,USART_CMODE_ASYNCHRONOUS_gc|USART_CHSIZE_8BIT_gc
                                           |USART_PMODE_DISABLED_gc
         STS USARTC0_CTRLC,R16
         LDI R16,12 //设置波特率 9600
         STS USARTC0_BAUDCTRLA,R16
         LDI R16,0
         STS USARTC0_BAUDCTRLB,R16
         LDI R16,USART_TXEN_bm // USARTC0 使能发送
         STS USARTC0_CTRLB,R16
         RET
//--------RESET（主函数入口）--------//
RESET:
    call uart_init
    LDI R16,ENTER//回车换行
    PUSH R16
    uart_putc
    uart_puts_string  MAIN;uart_putc('\n');
    LDI R16,ENTER
    PUSH R16
    uart_putc
    call wdt_sw_enable_example
//--------wdt_sw_enable_example（WDLOCK熔丝位没有置位）--------//
wdt_sw_enable_example:
    uart_puts_string  sw
    LDI R16,ENTER
    PUSH R16
```

```
    uart_putc
    //使能看门狗普通模式，超时周期为32CLK=32 ms
    ldi r16,WDT_ENABLE_bm | WDT_CEN_bm | WDT_PER_32CLK_gc
    ldi R17,CCP_IOREG_gc
    STS CPU_CCP,R17
    STS WDT_CTRL,R16
wdt_sw_enable_example_1:
    lds r16,WDT_STATUS
    sbrc r16,WDT_SYNCBUSY_bp
    jmp wdt_sw_enable_example_1
    nop
    lds r16,WDT_CTRL
    sbrs r16,WDT_ENABLE_bp
    jmp wdt_sw_enable_example_2
    nop
    uart_puts_string  WDTenabled
    jmp wdt_sw_enable_example_3
wdt_sw_enable_example_2:
    uart_puts_string  WDTunenabled
wdt_sw_enable_example_3:
    wdr
    jmp wdt_sw_enable_example_3
```

5.10 RTC - 实时计数器实例

实时计数器（RTC）分 16 位计数器和 32 位计数器两种，当它达到已配置的比较值或 TOP 值时，将产生一个事件或一个中断请求。16 位实时计数器 RTC 的参考时钟，可使用 32.768 kHz 或 1.024 kHz 作为输入。外部 32.768 kHz 晶体振荡器或内部 RC 振荡器 32 kHz 都可以选择作为时钟源。32 位实时计数器 RTC 的参考时钟可使用 1.024 kHz 或 1Hz（32.768kHz 分频）作为输入，但必须使用外部 32.768 kHz 的晶振作为时钟源。

1. 实时计数器（RTC）实现秒计数器。程序设置 RTC 时钟为 1.024 kHz（由内部 32.768 kHz RC 振荡器产生），RTC 计数周期为 1023（1S 钟溢出一次），比较寄存器的值为 512；使能溢出中断和比较中断。在中断中使用 7 个 LED 灯显示时间的 BCD 码，端口 D 的 PIN0-3 显示秒数的个位，PIN4-6 显示秒数的十位，PIN7 上的 LED 间隔 1 秒闪烁一次。

C 语言代码：

```
//--------包含头文件--------//
#include "avr_compiler.h"
#include "rtc_driver.c"
#define LED_PORT          PORTD
#define RTC_CYCLES_1S     1024
//bcd 码秒计数
typedef struct RTC_BCD_struct{
    uint8_t sec_ones;   //个位
```

```
    uint8_t sec_tens;   //十位
} RTC_BCD_t;
 //--------main（主函数入口）--------//
int main(void)
{
    //打开内部 32.768 kHz RC 振荡器产生 1.024 kHz
    OSC.CTRL |= OSC_RC32KEN_bm;
    do {//等待时钟稳定
    } while ( ( OSC.STATUS & OSC_RC32KRDY_bm ) == 0);
    //设置由内部 32.768 kHz RC 振荡器产生 1.024 kHz 为 RTC 时钟源
    CLK.RTCCTRL = CLK_RTCSRC_RCOSC_gc | CLK_RTCEN_bm;
    //设置 LED 端口为输出
    LED_PORT.DIR = 0xFF;
    do {//检查 RTC 忙否
    } while ( RTC_Busy() );
    //配置 RTC 周期为 1S
    RTC_Initialize( RTC_CYCLES_1S, 0, 0, RTC_PRESCALER_DIV1_gc );
    //设置间隔 0.5 秒 PIN7 上的 LED 闪烁
    RTC_SetAlarm(RTC_CYCLES_1S/2);
    //使能溢出中断
    RTC_SetOverflowIntLevel( RTC_OVFINTLVL_LO_gc);
    //使能比较中断
    RTC_SetCompareIntLevel( RTC_COMPINTLVL_LO_gc);
    PMIC.CTRL |= PMIC_LOLVLEN_bm;
    sei();
    do {
        nop();
       } while (1);
}
//--------ISR（溢出中断服务程序）--------//
ISR(RTC_OVF_vect)
{
    static RTC_BCD_t rtcTime;
    if ( ++rtcTime.sec_ones > 9 )
    {
        rtcTime.sec_ones = 0;
        rtcTime.sec_tens++;
    }
    if ( rtcTime.sec_tens > 5 )
    {
        rtcTime.sec_tens = 0;
    }
    LED_PORT.OUT = ( ( rtcTime.sec_tens << 4 ) | rtcTime.sec_ones );
}
```

```
//--------ISR（比较中断服务程序）--------//
ISR(RTC_COMP_vect)
   {
      LED_PORT.OUTTGL = 0X80;
      RTC_SetAlarm(RTC_CYCLES_1S/2);
   }
```

汇编代码：

```
//---------包含头文件--------//
.include "ATxmega128A1def.inc";器件配置文件,决不可少,不然汇编通不过
.ORG 0
        RJMP RESET//复位
.ORG 0x014
        RJMP ISR_RTC_OVF_vect
.ORG 0x016
        RJMP ISR_RTC_COMP_vect
.ORG 0X100        ;跳过中断区0x00-0x0FF
//---------宏定义--------//
.MACRO LED
           LDI R16,@0
           STS PORTD_OUTTGL,R16
.ENDMACRO
//---------RESET（主函数入口）--------//
RESET:
     //打开内部 32.768 kHz RC振荡器产生1Khz
     ldi r16,OSC_RC32KEN_bm
     sts OSC_CTRL,r16
     //等待时钟稳定
RESET_1:
     lds r16,OSC_STATUS
     sbrs r16,OSC_RC32KRDY_bp
     jmp RESET_1
     ldi r16,CLK_RTCSRC_RCOSC_gc | CLK_RTCEN_bm //设置内部32kHz为RTC时钟源
     sts CLK_RTCCTRL,r16
     ldi r16,0xff //设置LED端口为输出
      sts PORTD_DIR,r16
RESET_2:          //检查RTC忙否
     lds r16,RTC_STATUS
     sbrc r16,RTC_SYNCBUSY_bp
     jmp RESET_2
      nop
     //配置RTC周期为1S
      ldi r16,0xff
      sts RTC_PER,r16
      ldi r16,0x03
```

```
      sts RTC_PER+1,r16
      ldi r16,0
      sts RTC_CNT,r16
      ldi r16,0
      sts RTC_COMP,r16
      ldi r16,RTC_PRESCALER_DIV1_gc
      sts RTC_CTRL,r16
      ldi r16,RTC_OVFINTLVL_LO_gc|RTC_COMPINTLVL_LO_gc //使能溢出中断，比较中断
      sts RTC_INTCTRL ,r16
      ldi r16,PMIC_LOLVLEN_bm
      sts PMIC_CTRL,r16
      sei
RESET_3:
    jmp RESET_3
//---------ISR_RTC_OVF_vect（溢出中断服务程序）--------//
ISR_RTC_OVF_vect:
       LED 0x10
       reti
//---------比较中断服务程序（比较中断服务程序）--------//
ISR_RTC_COMP_vect:
        LED 0x20
        Reti
```

5.11 DAC - 数模转换器实例

DAC将数字信号转换为模拟电压。DAC的精度12位，可以每秒转换1百万次。DAC可以连续输出到一个引脚或者通过使用抽样/保持（S/H）电路输出到2个不同的引脚。

输出信号摆幅由参考电压（V_{REF}）决定。以下为 DAC 参考源 V_{REF}：

- ✓ AVCC
- ✓ 内部 1.00V
- ✓ 外部参考电压，PORTA 或 PORTB 的 AREF 引脚

1. DACB通道0和通道1产生相位差180°的三角波。C语言程序设置DACB的CH0和CH1输出使能，通过示波器可以看到从PB2和PB3输出的相位相反的三角波；汇编代码只使能DACB的CH0通道输出，可以通过示波器看到PB2输出的三角波。

C语言代码：

```
//---------包含头文件--------//
#include "dac_driver.c"
//---------main（主函数入口）--------//
int main( void )
{
    uint16_t angle;
    DAC_DualChannel_Enable( &DACB,//端口B
                            DAC_REFSEL_INT1V_gc,//内部1V
```

```
                            false,//左对齐
                            DAC_CONINTVAL_4CLK_gc,// 4 CLK / 6 CLK  S/H
                            DAC_REFRESH_32CLK_gc );//刷新率
    while (1)
    {
        for ( angle = 0; angle < 0x1000; ++angle )
        {
            while ( DAC_Channel_DataEmpty( &DACB, CH0 ) == false )
            {
            }
            DAC_Channel_Write( &DACB, angle, CH0 );
            while ( DAC_Channel_DataEmpty( &DACB, CH1 ) == false )
            {
            }
            DAC_Channel_Write( &DACB, 0xFFF - angle, CH1 );
        }
        for ( angle = 0; angle < 0x1000; ++angle )
        {
            while ( DAC_Channel_DataEmpty( &DACB, CH0 ) == false )
            {}
            DAC_Channel_Write( &DACB, 0xFFF - angle, CH0 );
            while ( DAC_Channel_DataEmpty( &DACB, CH1 ) == false )
            {}
            DAC_Channel_Write( &DACB, angle, CH1 );
        }
    }
}
```

汇编代码：

```
//--------包含头文件--------//
.include "ATxmega128A1def.inc";器件配置文件,决不可少,不然汇编通不过
.ORG 0
        RJMP RESET
.ORG 0X100        ;跳过中断区0x00-0x0FF
//---------RESET （主函数入口）--------//
RESET:
    ldi R16,DAC_CHSEL_SINGLE_gc
    sts DACB_CTRLB,R16
    ldi R16,~( DAC_REFSEL_gm | DAC_LEFTADJ_bm )|DAC_REFSEL_INT1V_gc
    sts DACB_CTRLC,R16
    ldi r16,DAC_CONINTVAL_4CLK_gc|DAC_REFRESH_32CLK_gc
    sts DACB_TIMCTRL,r16
    ldi r16,DAC_CH0EN_bm | DAC_ENABLE_bm
    sts DACB_CTRLA,r16
    ldi xh,0xff
```

```
RESET_1:
      lds r16,DACB_STATUS
      SBRS R16,DAC_CH0DRE_bp
      jmp RESET_1
      nop
      sts DACB_CH0DATA,xh
      dec xh
      jmp RESET_1
```

5.12 AC - 模拟比较器实例

模拟比较器（AC）比较 2 个输入引脚上的电压值，基于该比较上给出数字输出。模拟比较器可根据多种输入变化产生中断请求和事件。

模拟比较器的 2 个重要的动态特性是磁滞和传播时延。这些参数都可以调整。

模拟比较器在每个模拟端口上成对出现（AC0 和 AC1）。它们性能相同，但控制寄存器是分开的。

1. A 端口上的模拟比较器 AC0 与 AC1 根据输出电平变化产生中断，控制 LED 灯的闪烁。AC0 的正引脚为 PA0，负引脚为内部的分压；AC1 的正引脚为 PA1，负引脚为内部的分压；定时器 TCC0 的通道 A 与 B 单斜率模式产生方波作为 AC 引脚 PA0 与 PA1 的输入；AC0 为上升沿中断，AC1 为下降沿中断；在 AC0 中断中控制 PD4 取反，在 AC1 中断中控制 PD5 取反。硬件电路连接说明如下：

```
 *    - PC0 与 PA0连接
 *    - PC1 与 PA1 连接
```

C 语言代码：

```
//--------包含头文件--------//
#include "ac_driver.c"
#include "TC_driver.c"
#define AC ACA
#define LED1_T()   PORTD_OUTTGL = 0x20
#define LED2_T()   PORTD_OUTTGL = 0x10
//---------main（主函数入口）--------//
int main(void)
{
      PORTCFG.VPCTRLA=0x10; //PORTB 映射到虚拟端口 1，PORTA 映射到虚拟端口 0
      PORTCFG.VPCTRLB=0x32;//;PORTC 映射到虚拟端口 2，PORTD 映射到虚拟端口 3
      VPORT0_DIR=0x00; //PORTA 引脚输入
      VPORT2_DIR=0xFF; //PORTC 引脚输出
      VPORT3_DIR=0xFF; //PORTD 引脚输出
      /* 设置计数周期 */
      TC_SetPeriod( &TCC0,4000 );
      TC_SetCompareA( &TCC0,1000);
      TC_SetCompareB( &TCC0,2000);
      /* 设置 TC 为单斜率模式 */
```

```
    TC0_ConfigWGM( &TCC0, TC_WGMODE_SS_gc );
    /* 使能通道 A B */
    TC0_EnableCCChannels( &TCC0, TC0_CCAEN_bm );
    TC0_EnableCCChannels( &TCC0, TC0_CCBEN_bm );
    /* 选择时钟，启动定时器 */
    TC0_ConfigClockSource( &TCC0, TC_CLKSEL_DIV1024_gc );
    /*使能模拟比较器 AC0 AC1*/
    AC_Enable(&AC, ANALOG_COMPARATOR0, false);
    AC_Enable(&AC, ANALOG_COMPARATOR1, false);
    /*模拟比较器输入电压比例因子*/
    AC_ConfigVoltageScaler(&AC,0);
    /* 设置模拟比较器 0 的引脚是 pin 0 and 1. */
    AC_ConfigMUX(&AC,ANALOG_COMPARATOR0,AC_MUXPOS_PIN0_gc,
                                                    AC_MUXNEG_SCALER_gc);
    AC_ConfigMUX(&AC,ANALOG_COMPARATOR1,AC_MUXPOS_PIN2_gc,
                                                    AC_MUXNEG_SCALER_gc);
    /*设置 AC0 AC1 的磁滞 */
    AC_ConfigHysteresis(&AC, ANALOG_COMPARATOR0, AC_HYSMODE_SMALL_gc);
    AC_ConfigHysteresis(&AC, ANALOG_COMPARATOR1, AC_HYSMODE_SMALL_gc);
    AC_ConfigInterrupt(&AC,ANALOG_COMPARATOR0,AC_INTMODE_RISING_gc,
                                                        AC_INTLVL_LO_gc)
;
    AC_ConfigInterrupt(&AC,ANALOG_COMPARATOR1,AC_INTMODE_FALLING_gc,
                                                        AC_INTLVL_LO_gc)
;
    PMIC.CTRL|=PMIC_MEDLVLEN_bm+PMIC_LOLVLEN_bm+PMIC_HILVLEN_bm;
    sei();
    while (1);
}
//---------ISR（AC0中断）--------//
 ISR(ACA_AC0_vect)
{
  LED2_T();
}
//---------ISR（AC1中断）--------//
ISR(ACA_AC1_vect)
{
  LED1_T();
}
```

汇编代码：

```
//---------包含头文件--------//
.include "ATxmega128A1def.inc";器件配置文件,决不可少,不然汇编通不过
.ORG 0
        RJMP RESET//复位
```

```
.ORG 0x88
        RJMP ISR_ACA_AC0_vect
.ORG 0x8A
        RJMP ISR_ACA_AC1_vect
.ORG 0X100        ;跳过中断区0x00-0x0FF
//--------宏定义--------//
.MACRO LED1_T
           LDI R16,@0
           STS PORTD_OUTTGL,R16
.ENDMACRO
//---------RESET（主函数入口）--------//
RESET:
     ldi r16,0x10
     sts PORTCFG_VPCTRLA,r16//;PORTB映射到虚拟端口1，PORTA映射到虚拟端口0
     ldi r16,0x32
     sts PORTCFG_VPCTRLB,r16//;PORTC映射到虚拟端口2，PORTD映射到虚拟端口3
     ldi r16,0x00
     sts VPORT0_DIR,r16 //PORTA引脚输入
     ldi r16,0xff
     sts VPORT2_DIR,r16 //PORTC引脚输出
     ldi r16,0xff
     sts VPORT3_DIR,r16 //PORTD引脚输出
     /* 设置计数周期 */
     ldi XH,0X0F
     ldi XL,0X0a0
     sts TCC0_PER,XL
     sts TCC0_PER+1,XH
     ldi XH,0X03
     ldi XL,0X0e8
     sts TCC0_CCABUF,XL
     sts TCC0_CCABUF+1,XH
     ldi XH,0X07
     ldi XL,0X0D0
     sts TCC0_CCBBUF,XL
     sts TCC0_CCBBUF+1,XH
    /* 设置TC为单斜率模式 使能通道A B */
     LDS R16,TCC0_CTRLB
     ORI R16,TC_WGMODE_SS_gc|TC0_CCAEN_bm | TC0_CCBEN_bm
     STS TCC0_CTRLB,R16
     /* 选择时钟，启动定时器 */
     LDI R16,TC_CLKSEL_DIV1024_gc
     STS TCC0_CTRLA,R16
     /*模拟比较器输入电压比例因子*/
     LDI R16,0
```

```
    STS ACA_CTRLB,R16
    /* 设置模拟比较器0的引脚是 pin 0 and 1. */
    LDI R16,AC_MUXPOS_PIN0_gc|AC_MUXNEG_SCALER_gc
    STS ACA_AC0MUXCTRL,R16
    LDI R16,AC_MUXPOS_PIN2_gc|AC_MUXNEG_SCALER_gc
    STS ACA_AC1MUXCTRL,R16
    /*设置AC0 AC1的磁滞;低级中断级别;使能模拟比较器AC0 AC1 */
    LDI R16,AC_HYSMODE_SMALL_gc|AC_INTMODE_RISING_gc|AC_INTLVL_LO_gc
|AC_ENABLE_bm
    STS ACA_AC0CTRL,R16
    LDI R16,AC_HYSMODE_SMALL_gc|AC_INTMODE_FALLING_gc|AC_INTLVL_LO_gc
                                                          |AC_ENABL
E_bm
    STS ACA_AC1CTRL,R16
    LDI R16,PMIC_HILVLEN_bm + PMIC_MEDLVLEN_bm + PMIC_LOLVLEN_bm;
   /*可编程多层中断控制寄存器高 中 低层使能，循环调度关闭，中断向量未移至 Boot
   section*/
    STS PMIC_CTRL,R16
    SEI                  //全局中断使能置位
LOOP:
       NOP
       JMP LOOP
//---------ISR_ACA_AC0_vect（AC0中断函数入口）--------//
ISR_ACA_AC0_vect:
   LED1_T 0X10
   RETI
//---------ISR_ACA_AC1_vect（AC1中断函数入口）--------//
ISR_ACA_AC1_vect:
   LED1_T 0X20
   RETI
```

5.13 事件系统实例

事件系统用来进行外设间通信的。它可以使一个外设状态的改变自动触发其他外设的行为。这是个简单但强大的系统，可以允许外设自主控制而不需要中断、CPU 或 DMA。

事件分为两类：信号事件和数据事件。信号事件只会指示状态的改变，数据事件包含额外的信息。

产生事件的外设称为事件生成器。每个外设，例如定时器/计数器可以有多个事件源，如定时器比较匹配或定时器溢出。使用事件的外设称为事件使用者，触发的行为称为事件行为。

1. 事件系统的设置与运用。

Example1 函数选择 PD0 为通道 0 事件输入，通道 0 作为 TCC0 的事件源 并且事件行为是输入捕获，当 PD0 上面有电平变化时捕获标志位置位，当检测到捕获标志位置位时

我们使 PD5 上面的电平取反，会看到 LED 闪烁。

Example2 函数是选择 TCC0 益处作为事件通道 0 的事件，通道 0 触发 SWEEP 中定义的 ADC 通道的一次扫描，又因为 ADC 设置为自由模式，所以一旦 TCC0 溢出就会触发 ADC 通道的不断扫描。

Example3 函数实现 32 位计数，并且 TCC0 和 TCC1 对通道 1 捕获,当 PD0 有电平变化时 TCC0 和 TCC1 比较捕获标志就会置位。

Example4 是选择 PD0 为通道 0 事件输入,TCC0 时钟源为事件通道 0,也就是说当 PD0 电平变化时 TCC0 计数就会增加 1。

C 语言代码：

```
//--------包含头文件--------//
#include "avr_compiler.h"
#include "event_system_driver.c"
#include "TC_driver.c"
void Example1( void );
void Example2( void );
void Example3( void );
void Example4( void );
#define LED1_T()   PORTD_OUTTGL = 0x20
#define LED1_ON()  PORTD_OUTSET = 0x20
#define LED1_OFF() PORTD_OUTSET = 0x20
//--------main（主函数入口）--------//
int main( void )
{
    Example1();
    /*Example2();*/
    /*Example3();*/
    /*Example4();*/
    do {}while (1);
}
//--------Example1--------//
void Example1( void )
{
    /* PD.0 输入/双沿感知*/
    PORTD.PIN0CTRL |= PORT_ISC_BOTHEDGES_gc;
    PORTD.DIRCLR = 0x01;
    PORTD_DIRSET = 0x20;
    /*选择 PD0 为 channel 0 事件输入*/
    EVSYS_SetEventSource( 0, EVSYS_CHMUX_PORTD_PIN0_gc );
   //选择通道 0 作为 TCC0 的事件源 并且事件行为是输入捕获
    TCC0.CTRLD = (uint8_t) TC_EVSEL_CH0_gc | TC_EVACT_CAPT_gc;
    /* 使能 TCC0 比较捕获通道 A */
    TCC0.CTRLB |= TC0_CCAEN_bm;
    /*设置 TCC0 计数周期 */
    TCC0.PER = 0xFFFF;
```

```
    TCC0.CTRLA = TC_CLKSEL_DIV1_gc; //TCC0 时钟源
    while (1) {
        if ( TCC0.INTFLAGS & TC0_CCAIF_bm ) {
           /* 当捕获发生时标志位置位，清除标志位*/
            TCC0.INTFLAGS |= TC0_CCAIF_bm;
            LED1_T();
        }
    }
}
//---------Example2--------//
void Example2( void )
{
    PORTD_DIRSET = 0x20;
    /*选择 TCC0 益处作为事件通道 0 的事件*/
    EVSYS_SetEventSource( 0, EVSYS_CHMUX_TCC0_OVF_gc );
    /*ADC 被选择的通道 0 1 2 3，事件通道 0，1 ，2,3 作为所选事件通道；
    EVSEL 定义的通道号最小的事件通道将触发 SWEEP 中定义的 ADC 通道的一次扫描*/
    ADCA.EVCTRL = (uint8_t) ADC_SWEEP_0123_gc |
                  ADC_EVSEL_0123_gc |
                  ADC_EVACT_SWEEP_gc;
    /* 通道0,1,2,3配置为单端正向输入信号 并且设置0,1,2,3的各个通道的正向输入引
脚*/
    ADCA.CH0.MUXCTRL = (uint8_t) ADC_CH_MUXPOS_PIN4_gc |0;
    ADCA.CH0.CTRL = ADC_CH_INPUTMODE_SINGLEENDED_gc;
    ADCA.CH1.MUXCTRL = (uint8_t) ADC_CH_MUXPOS_PIN5_gc |0;
    ADCA.CH1.CTRL = ADC_CH_INPUTMODE_SINGLEENDED_gc;
    ADCA.CH2.MUXCTRL = (uint8_t) ADC_CH_MUXPOS_PIN6_gc |0;
    ADCA.CH2.CTRL = ADC_CH_INPUTMODE_SINGLEENDED_gc;
    ADCA.CH3.MUXCTRL = (uint8_t) ADC_CH_MUXPOS_PIN7_gc |0;
    ADCA.CH3.CTRL = ADC_CH_INPUTMODE_SINGLEENDED_gc;
    /*  设置 ADC 时钟预分频为 DIV8；精度为 12 位；ADC 自由运行模式
       ADC 参考电压为内部 VCC / 1.6V；使能 ADC*/
    ADCA.PRESCALER = ( ADCA.PRESCALER & ~ADC_PRESCALER_gm ) |
                     ADC_PRESCALER_DIV8_gc;
    ADCA.CTRLB = ( ADCA.CTRLB & ~ADC_RESOLUTION_gm ) |
                 ADC_RESOLUTION_12BIT_gc;
    ADCA.CTRLB = ( ADCA.CTRLB & ~( ADC_CONMODE_bm | ADC_FREERUN_bm ) );
    ADCA.REFCTRL = ( ADCA.REFCTRL & ~ADC_REFSEL_gm ) |
                   ADC_REFSEL_VCC_gc;
    ADCA.CTRLA |= ADC_ENABLE_bm;
    /*设置 TCC0 计数周期 */
    TCC0.PER = 0x0FFF;
    /* 设置溢出中断为低级别中断 */
    TC0_SetOverflowIntLevel( &TCC0, TC_OVFINTLVL_LO_gc );
```

```
    PMIC.CTRL |= PMIC_LOLVLEN_bm;
    sei();
    TCC0.CTRLA |= TC_CLKSEL_DIV256_gc; //TCC0 时钟源
    while (1) {
    /*当 TCC0 溢出时 ADC 通道将不断的被扫描
    PD5 上面的灯会不断的闪烁*/
               }
}
ISR(TCC0_OVF_vect)
{
 LED1_T();
}
//--------Example3--------//
void Example3( void )
{
    /* PD.0 输入/双沿感知*/
    PORTD.PIN0CTRL = PORT_ISC_BOTHEDGES_gc;
    PORTD.DIRCLR = 0x01;
    /*TCC0 溢出作为通道 0 的事件源*/
    EVSYS_SetEventSource( 0, EVSYS_CHMUX_TCC0_OVF_gc );
    /*选择 PD0 为通道 1 事件输入 */
    EVSYS_SetEventSource( 1, EVSYS_CHMUX_PORTD_PIN0_gc );
    /* 选择通道 0 作为 TCC1 的时钟源 TCC0 与 TCC1 级联成了 32 为计数器*/
    TCC1.CTRLA = TC_CLKSEL_EVCH0_gc;
    /* 设置通道 TCC0 为捕获模式 */
    TCC0.CTRLD = (uint8_t) TC_EVSEL_CH1_gc | TC_EVACT_CAPT_gc;
    /* 设置通道 TCC1 为捕获模式并且添加事件的延迟来弥补插入的传播时延 */
    TCC1.CTRLD = (uint8_t) TC_EVSEL_CH1_gc | TC0_EVDLY_bm | TC_EVACT_CAPT_gc;
    /* 使能 TCC0,TCC1 的 A 通道 */
    TCC0.CTRLB = TC0_CCAEN_bm;
    TCC1.CTRLB = TC1_CCAEN_bm;
    TCC0.CTRLA = TC_CLKSEL_DIV1_gc; //TCC0 时钟源
     while (1) {
        if ( TCC0.INTFLAGS & TC0_CCAIF_bm ) {
            /* 当捕获发生时标志位置位，清除标志位*/
            TCC0.INTFLAGS |= TC0_CCAIF_bm;
            TCC1.INTFLAGS |= TC1_CCAIF_bm;
        }
    }
}
//--------Example4--------//
void Example4( void )
{
    /* PD.0 输入/双沿感知*/
```

```
    PORTD.PIN0CTRL = PORT_ISC_RISING_gc;
    PORTD.DIRCLR = 0x01;
    /* PC 引脚全部输出*/
    PORTC.DIRSET = 0xFF;
    /*选择 PD0 为通道 0 事件输入*/
    EVSYS_SetEventSource( 0, EVSYS_CHMUX_PORTD_PIN0_gc );
    /*  在 TCC0 通道 A 上加数字滤波器*/
    EVSYS_SetEventChannelFilter( 0, EVSYS_DIGFILT_8SAMPLES_gc );
    /*设置 TCC0 计数周期 */
    TCC0.PER = 0xFFFF;
    //TCC0 时钟源为事件通道 0
    TCC0.CTRLA = TC_CLKSEL_EVCH0_gc;
    while (1) {
        /* 输出 TCC0 计数器中数值的相反*/
        PORTC.OUT = ~TCC0.CNT;
    }
}
```

汇编代码：

```
//--------包含头文件--------//
.include "ATxmega128A1def.inc";器件配置文件,决不可少,不然汇编通不过
.ORG 0
        RJMP RESET//复位
.ORG 0X100        ;跳过中断区 0x00-0x0FF
//--------RESET（主函数入口）--------//
RESET:
    call Example1
    /*call Example2
    call Example3
    call Example4 */
RESET_1:
    JMP RESET_1
//--------Example1--------//
Example1:
    /* PD.0 输入/双沿感知*/
    ldi r16,PORT_ISC_BOTHEDGES_gc
    sts PORTD_PIN0CTRL,r16
    ldi r16,0x01
    sts PORTD_DIRCLR,r16
    /*选择 PD0 为 channel 0 事件输入*/
    LDI R16,EVSYS_CHMUX_PORTD_PIN0_gc
    STS EVSYS_CH0MUX,R16

    /*选择通道 0 作为 TCC0 的事件源 并且事件行为是输入捕获*/
    LDI R16,TC_EVSEL_CH0_gc |TC_EVACT_CAPT_gc
```

```
    STS TCC0_CTRLD,R16
    /* 使能 TCC0 比较捕获通道 A */
    LDI R16,TC0_CCAEN_bm
    STS TCC0_CTRLB,R16
    /*设置 TCC0 计数周期 */
    LDI XL,0X0FF
    STS  TCC0_PER,XL
    STS  TCC0_PER+1,XL
    LDI R16,TC_CLKSEL_DIV1_gc //TCC0 时钟源
    STS TCC0_CTRLA,R16
    /* 当捕获发生时标志位置位，清除标志位*/
Example1_1:
    LDS R16,TCC0_INTFLAGS
    SBRS R16,TC0_CCAIF_bp
    JMP  Example1_1
    LDI R16,TC0_CCAIF_bm
    STS TCC0_INTFLAGS,R16
    JMP Example1_1
    RET
//---------Example1--------//
Example2:
    /*选择 TCC0 益处作为事件通道 0 的事件*/
    LDI R16,EVSYS_CHMUX_TCC0_OVF_gc
    STS EVSYS_CH0MUX,R16
    /*ADC 被选择的通道 0 1 2 3，事件通道 0，1 ，2,3 作为所选事件通道；
    EVSEL 定义的通道号最小的事件通道将触发 SWEEP 中定义的 ADC 通道的一次扫描*/
    LDI R16,ADC_SWEEP_0123_gc |ADC_EVSEL_0123_gc |ADC_EVACT_SWEEP_gc
    STS ADCA_EVCTRL,R16
    /* 通道0,1,2,3配置为单端正向输入信号 并且设置0,1,2,3的各个通道的正向输入引
脚*/
    LDI R16,ADC_CH_MUXPOS_PIN4_gc |0
    STS ADCA_CH0_MUXCTRL,R16
    LDI R16,ADC_CH_INPUTMODE_SINGLEENDED_gc
    STS ADCA_CH0_CTRL,R16
    LDI R16,ADC_CH_MUXPOS_PIN5_gc |0
    STS ADCA_CH1_MUXCTRL,R16
    LDI R16,ADC_CH_INPUTMODE_SINGLEENDED_gc
    STS ADCA_CH1_CTRL,R16
    LDI R16,ADC_CH_MUXPOS_PIN6_gc |0
    STS ADCA_CH2_MUXCTRL,R16
    LDI R16,ADC_CH_INPUTMODE_SINGLEENDED_gc
    STS ADCA_CH2_CTRL,R16
    LDI R16,ADC_CH_MUXPOS_PIN7_gc |0
    STS ADCA_CH3_MUXCTRL,R16
```

```
    LDI R16,ADC_CH_INPUTMODE_SINGLEENDED_gc
    STS ADCA_CH3_CTRL,R16
    /*设置 ADC 时钟预分频为 DIV8；精度为 12 位；ADC 自由运行模式
     ADC 参考电压为内部 VCC / 1.6V；使能 ADC*/
    LDI R16,ADC_PRESCALER_DIV8_gc
    STS ADCA_PRESCALER,R16
    LDI R16,ADC_RESOLUTION_12BIT_gc|ADC_FREERUN_bm
    STS ADCA_CTRLB,R16
    LDI R16,ADC_REFSEL_VCC_gc
    STS ADCA_REFCTRL,R16
    LDI R16,ADC_ENABLE_bm
    STS ADCA_CTRLA,R16
    /*设置 TCC0 计数周期 */
    LDI XL,0X0FF
    STS  TCC0_PER,XL
    STS  TCC0_PER+1,XL
    LDI R16,TC_CLKSEL_DIV1_gc//TCC0 时钟源
    STS TCC0_CTRLA,R16
Example2_1:
    /*当 TCC0 溢出时 ADC 通道将不断的被扫描 */
    JMP Example2_1
    RET
//--------Example3--------//
Example3:
    /* PD.0 输入/双沿感知*/
    ldi r16,PORT_ISC_BOTHEDGES_gc
    sts PORTD_PIN0CTRL,r16
    ldi r16,0x01
    sts PORTD_DIRCLR,r16
    /*TCC0 溢出作为通道 0 的事件源*/
    LDI R16,EVSYS_CHMUX_TCC0_OVF_gc
    STS EVSYS_CH0MUX,R16
    /*选择 PD0 为通道 1 事件输入 */
    LDI R16,EVSYS_CHMUX_PORTD_PIN0_gc
    STS EVSYS_CH1MUX,R16
    /* 选择通道 0 作为 TCC1 的时钟源 TCC0 与 TCC1 级联成了 32 为计数器*/
    LDI R16,TC_CLKSEL_EVCH0_gc
    STS TCC1_CTRLA,R16
    /* 设置通道 TCC0 为捕获模式 */
    LDI R16,TC_EVSEL_CH1_gc | TC_EVACT_CAPT_gc
    STS TCC0_CTRLD,R16

    /* 设置通道 TCC1 为捕获模式并且添加事件的延迟来弥补插入的传播时延 */
    LDI R16,TC_EVSEL_CH1_gc|TC_EVACT_CAPT_gc| TC0_EVDLY_bm
```

```
    STS TCC1_CTRLD,R16
    /* 使能 TCC0,TCC1 的 A 通道 */
    LDI R16,TC0_CCAEN_bm
    STS TCC0_CTRLB,R16
    LDI R16,TC1_CCAEN_bm
    STS TCC1_CTRLB,R16
    LDI R16,TC_CLKSEL_DIV1_gc  //TCC0 时钟源
    STS TCC0_CTRLA,R16
    /* 当捕获发生时标志位置位，清除标志位*/
Example3_1:
    LDS R16,TCC0_INTFLAGS
    SBRS R16,TC0_CCAIF_bp
    JMP  Example3_1
    LDI R16,TC0_CCAIF_bm
    STS TCC0_INTFLAGS,R16
    LDI R16,TC1_CCAIF_bm
    STS TCC1_INTFLAGS,R16
    JMP Example3_1
    RET
//---------Example4--------//
Example4:
    /* PD.0 输入/双沿感知*/
    ldi r16,PORT_ISC_BOTHEDGES_gc
    sts PORTD_PIN0CTRL,r16
    ldi r16,0x01
    sts PORTD_DIRCLR,r16
    /* PC 引脚全部输出*/
    LDI R16,0x0ff
    STS PORTC_DIRSET,R16
    /*选择 PD0 为通道 0 事件输入*/
    LDI R16,EVSYS_CHMUX_PORTD_PIN0_gc
    STS EVSYS_CH0MUX,R16
    /* 在 TCC0 通道 A 上加数字滤波器*/
    LDI R16,EVSYS_DIGFILT_8SAMPLES_gc
    STS EVSYS_CH0CTRL,R16
    /*设置 TCC0 计数周期 */
    LDI r16,0x0ff
    STS  TCC0_PER,r16
    STS  TCC0_PER+1,r16
    LDI R16,TC_CLKSEL_EVCH0_gc //TCC0 时钟源
    STS TCC0_CTRLA,R16
Example4_1:
    /* 输出 TCC0 计数器中数值*/
    LDS R16,TCC0_CNT
```

```
STS PORTC_OUT,R16
JMP Example4_1
RET
```

5.14 EBI - 外部总线接口实例

外部总线接口是用于连接外设和存储器到数据存储空间的接口。当 EBI 使能时，内部 SRAM 之外的数据地址空间可以通过 EBI 引脚使用。

通过 EBI 可以连接 SRAM，SDRAM 或 LCD 显示器等内存映射的设备。

从 256 字节（8bit）至 16M 字节（24bit），地址空间和使用的引脚数量是可选的。无论引脚或多或少都可应用于 EBI，为了达到最佳使用，地址和数据总线有多种复用模式。外部总线将线性映射到内部 SRAM 的尾部。

1.EBI 三端口总线接口与 IS61LV6416sram 相连，EBI 通过总线读写 IS61LV6416sram。

74LVC573 是三态输出的八路透明 D 类锁存器。工作电压是 2.0V 到 5.5V;有 20 个引脚;引脚说明见表 5-14-1：

表 5-14-1 74LVC573 引脚说明

符号	说明	方向
1D-8D	数据输入总线	输入
1Q-8Q	数据输出总线	输出
/OE	输出使能位	输入
LE	锁定位	输入
VCC	电源	----
GND	接地	----

当/OE 输入为高电平时，数据输入总线对数据输出总线不起作用，数据输出总线呈现高阻态;
当/OE 输入为低电平，LE 输入高电平时，输入总线数据与输出总线数据保持一致;
当/OE 输入为低电平，LE 输入低电平时，输入总线数据将被保持在输出总线上，直到 LE 输入高电平为止。

IS61LV6416sram 是高速的，拥有 64K x 16 位的 SRAM 芯片，工作电压：3.3v；最大工作电流：120MA；温度级别：-65 到 150；工作功耗：75mw；引脚说明见表 5-14-2：

表 5-14-2 IS61LV6416 引脚说明

符号	说明	方向
A0-A15	地址总线	输入
I/O0-I/O15	数据总线	输入/输出
/CE	片选使能	输入
/OE	输出使能	输入
/WE	写使能	输入
/LB	低字节控制位（I/O0-I/O7）	输入
/UB	高字节控制位（I/O8-I/O15）	输入
NC	没有定义引脚	----
VDD	电源	----
GND	接地	----

IS61LV6416 芯片有 16 根数据总线和 16 根地址总线，可以访问 64K 大小的以字为单位

的空间，如图 5-14-1 低字节控制位置低，使能数据总线的低 8 位，每次读出或写入的是字节；即程序中芯片每个地址只用到低字节，高字节没有使用。

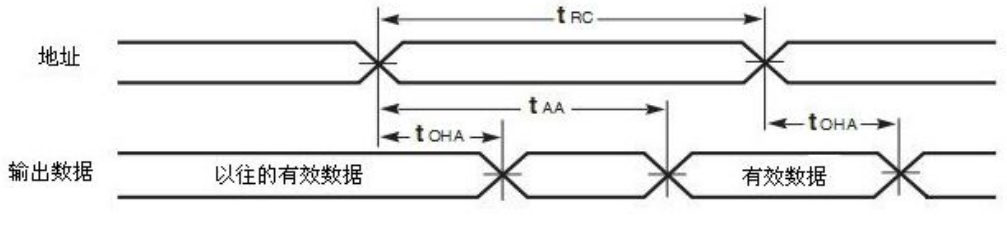


图 5-14-1 读时序 1 /CE 和/OE 均为低电平，/WE 为高电平

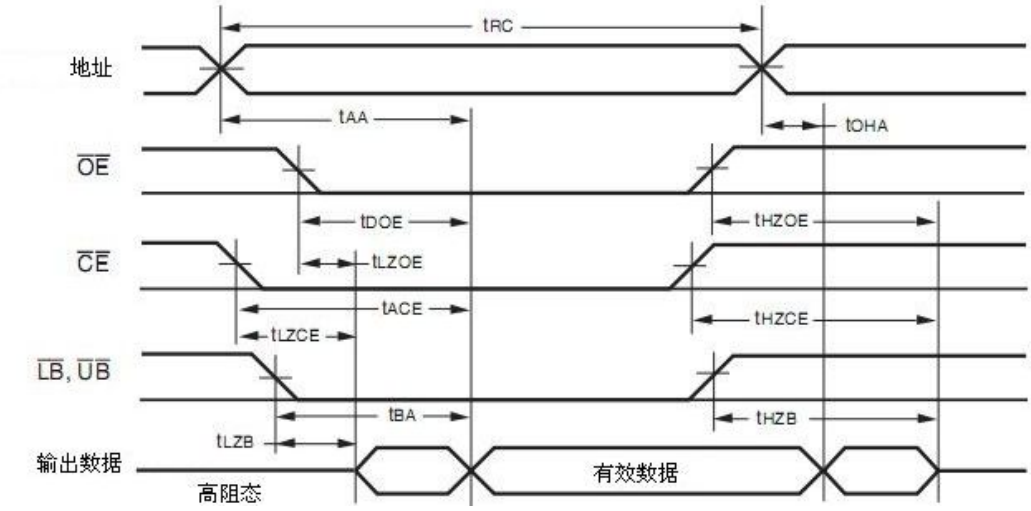


图 5-14-2 读时序 2 /WE 为高电平，由/CE 和/OE 控制数据读取

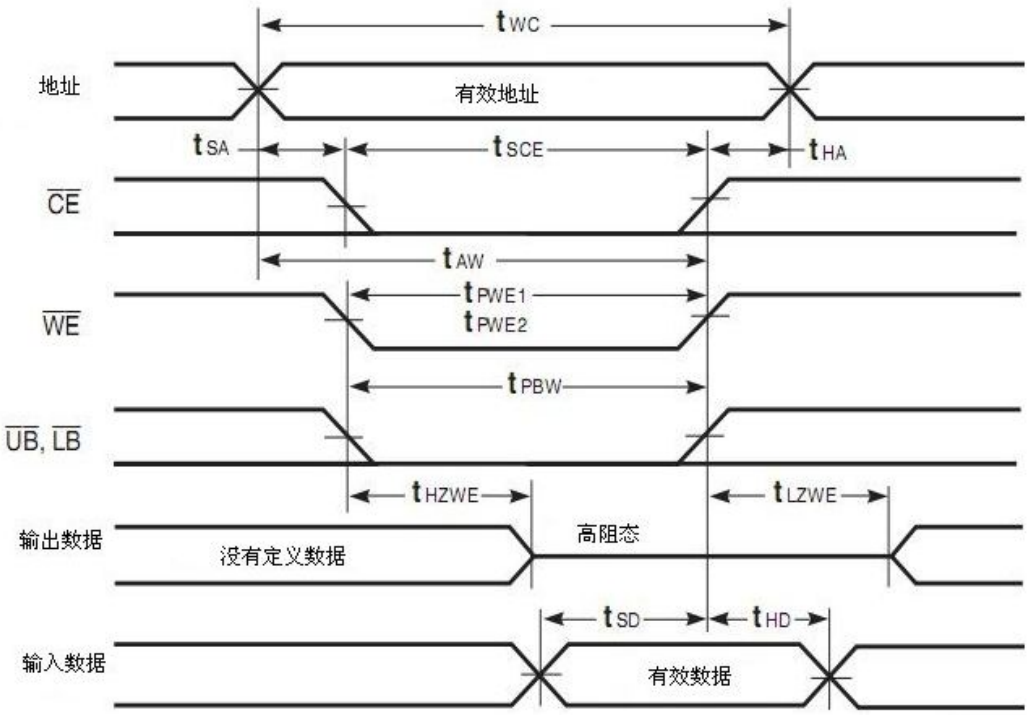


图 5-14-3 由/CE 控制数据写入 /OE 为高或者低电平

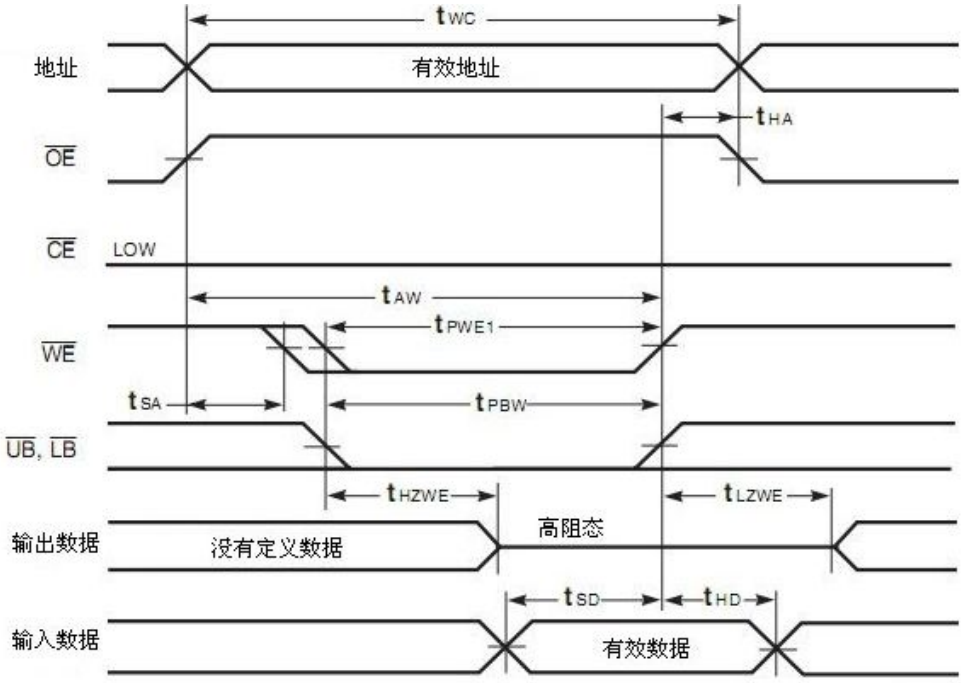


图 5-14-4 由/WE 控制数据写入 /OE 为高电平

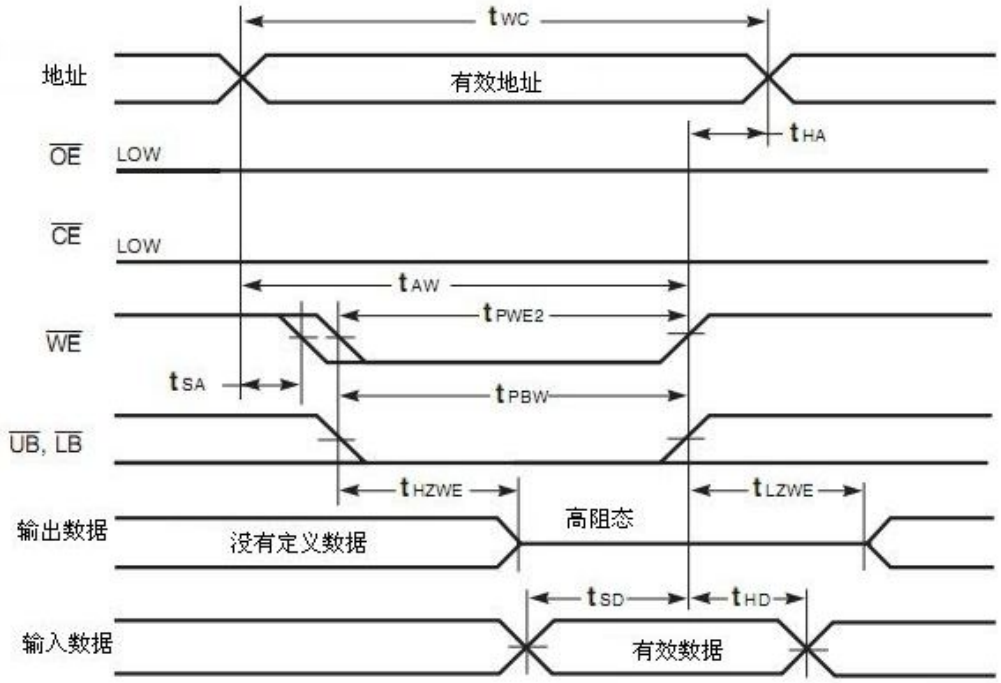


图 5-14-5 由/WE 控制数据写入 /OE 为低电平

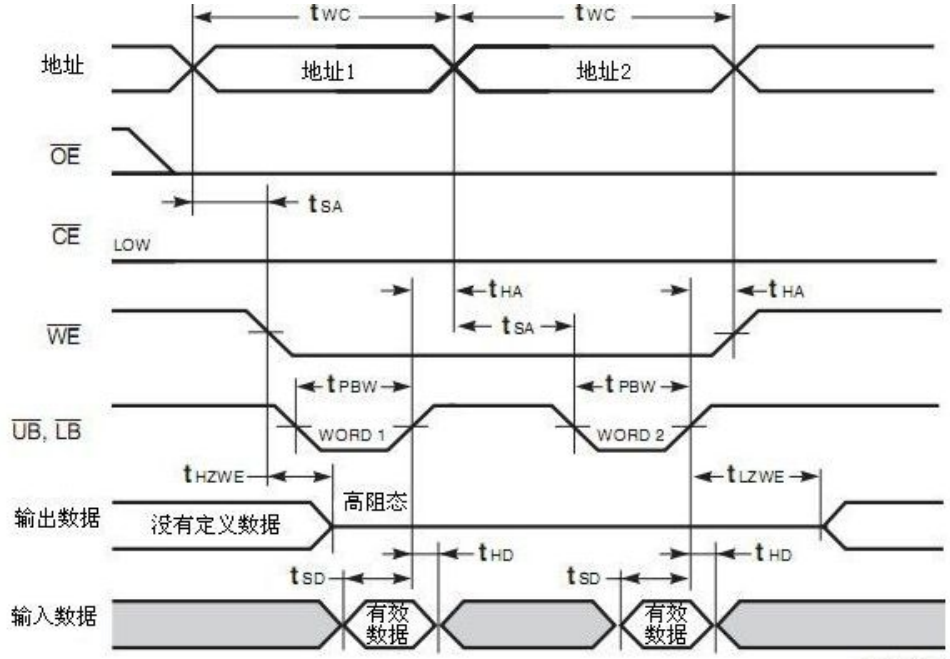


图 5-14-6 由/LB /UB 控制数据写入

程序中向 IS61LV6416sram 写入 10 个相同的数据，然后 EBI 通过数据总线读回来并通过串口打印。硬件连接见图 5-14-7：

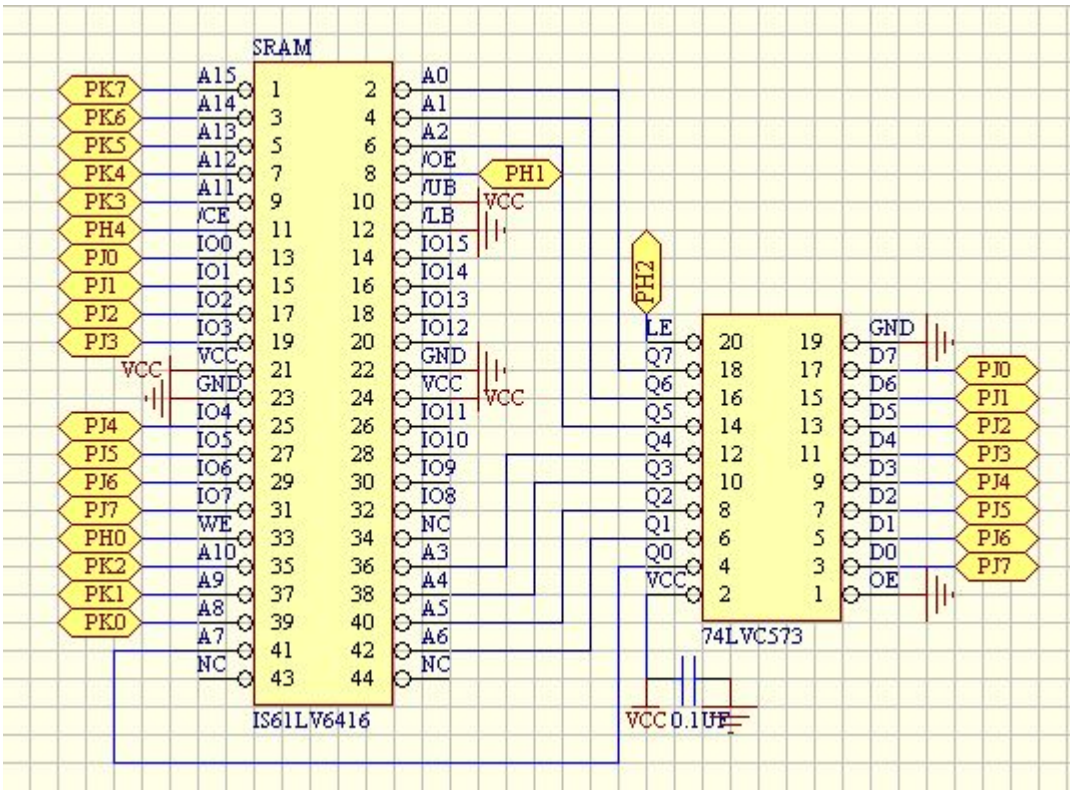


图 5-14-7 IS61LV6416 连接原理图

C 语言代码：

```
//--------包含头文件--------//
#include "avr_compiler.h"
#include "ebi_driver.c"
#include "usart_driver.c"
#define TESTBYTE   0xA5//测试数据
/*SRAM大小 UL=unsigned long*/
#define SRAM_SIZE 0x10000UL
#define SRAM_ADDR 0x4000  //外接sram的最低地址
#define WRITE_NUM 0x0A //写的数据个数
//--------uart_init(串口初始化)-------//
void uart_init(void)
{
    /* USARTC0 引脚方向设置*/
    PORTC.DIRSET   = PIN3_bm; //PC3 (TXD0) 输出
    PORTC.DIRCLR   = PIN2_bm; // PC2 (RXD0) 输入
    USART_SetMode(&USARTC0,USART_CMODE_ASYNCHRONOUS_gc); //USARTC0 模式 - 异步
  // USARTC0帧结构, 8 位数据位, 无校验, 1停止位
    USART_Format_Set(&USARTC0, USART_CHSIZE_8BIT_gc,USART_PMODE_DISABLED_gc,
false);
    USART_Baudrate_Set(&USARTC0, 12 , 0); //设置波特率 9600
    USART_Tx_Enable(&USARTC0); //USARTC0 使能发送
    USART_Rx_Enable(&USARTC0); //USARTC0 使能接收
```

```
}
//---------main（主函数入口）--------//
int main( void )
{
    /* 设置EBI总线的端口方向*/
    PORTH.DIR = 0xFF;
    PORTK.DIR = 0xFF;
    PORTJ.DIR = 0x00;
    uart_init();
    /* 初始化EBI */
    EBI_Enable( EBI_SDDATAW_8BIT_gc,
                EBI_LPCMODE_ALE1_gc,
                EBI_SRMODE_ALE2_gc,
                EBI_IFMODE_3PORT_gc );
    /*初始化SRAM*/
    EBI_EnableSRAM( &EBI.CS0,                /* 选择CS0 */
                    EBI_CS_ASPACE_64KB_gc,   /* 64 KB SRAM的大小 */
                    SRAM_ADDR,               /* 基地址 */
                    0 );
    /* 写数据*/
    for (uint32_t i = 0; i < WRITE_NUM; i++) {
        __far_mem_write(i+SRAM_ADDR, TESTBYTE);
    }
    /*读数据*/
    for (uint32_t i = 0; i < WRITE_NUM; i++) {
            uart_putc_hex(__far_mem_read(i+SRAM_ADDR));
        }
    while(1);
}
```

汇编代码：

```
//---------包含头文件--------//
.include "ATxmega128A1def.inc";器件配置文件,决不可少,不然汇编通不过
.include "usart_driver.inc"
.ORG 0
        RJMP RESET//复位
.ORG 0X100        ;跳过中断区0x00-0x0FF
.EQU ENTER  ='\n'
.EQU NEWLINE='\r'
.EQU EQUE   ='='
.EQU ZERO   ='0'
.EQU A_ASSCI='A'
.EQU TESTBYTE=0xA6 //测试数据
.EQU SRAM_ADDR=0x4000//外接sram的最低地址
.EQU WRITE_NUM=0x0A//写的数据个数
```

```
//----- uart_init （串口初始化）--------//
uart_init:
       /* USARTC0 引脚方向设置*/
         LDI R16,0X08 //PC3 (TXD0) 输出
         STS PORTC_DIRSET,R16
         LDI R16,0X04 //PC2 (RXD0) 输入
         STS PORTC_DIRCLR,R16
        //USARTC0 模式 - 异步  USARTC0帧结构, 8 位数据位, 无校验, 1停止位
         LDI R16,USART_CMODE_ASYNCHRONOUS_gc|USART_CHSIZE_8BIT_gc
|USART_PMODE_DISABLED_gc
         STS USARTC0_CTRLC,R16
         LDI R16,12//设置波特率 9600
         STS USARTC0_BAUDCTRLA,R16
         LDI R16,0
         STS USARTC0_BAUDCTRLB,R16
         LDI R16,USART_TXEN_bm//USARTC0 使能发送  USARTC0 使能接收
         STS USARTC0_CTRLB,R16
          RET
//----- RESET(主函数入口)--------//
RESET:
      /* 设置总线引脚 */
        LDI R16,0XFF
        STS PORTH_DIR,R16
        STS PORTK_DIR,R16
        LDI R16,0X00
        STS PORTJ_DIR,R16
        CALL uart_init
       /* 初始化EBI */
        LDI R16, EBI_SDDATAW_8BIT_gc|EBI_LPCMODE_ALE1_gc
                                    |EBI_SRMODE_ALE2_gc|EBI_IFMODE_3PORT_gc
        STS EBI_CTRL,R16
       /*初始化SRAM*/
        LDI R16,EBI_CS_SRWS_gm
        STS EBI_CS0_CTRLB,R16
        LDI R16,0x00
        STS EBI_CS0_BASEADDR,R16 //基地址必须是外接地址空间大小的倍数
        STS EBI_CS0_BASEADDR+1,R16
        LDI R16,EBI_CS_MODE_SRAM_gc|EBI_CS_ASPACE_64KB_gc
        STS EBI_CS0_CTRLA,R16
        /* 写数据*/
        LDI R30,0X00
        LDI R31,0X40
        in R17,CPU_RAMPZ
        LDI R16,0X00
```

```
        OUT CPU_RAMPZ,R16
RESET_1:
        LDI R16,TESTBYTE
        ST Z,R16
        INC R30
        CLZ
        CPI R30,10
        BRNE RESET_1
        OUT CPU_RAMPZ,R17
        /*读数据*/
        LDI R30,0X00
        LDI R31,0X40
        in R17, CPU_RAMPZ
        LDI R16,0X00
        OUT CPU_RAMPZ,R16
RESET_2:
        LD  XH,Z+
        uart_putc_hex
        CLZ
        CPI R30,10
        in R16,CPU_SREG
        SBRS R16,1
        JMP RESET_2
        NOP
        OUT CPU_RAMPZ,R17
RESET_3:JMP  RESET_3
```

5.15 DMA - 直接存储访问控制器实例

XMEGA DMA 控制器是高灵活的 DMA 控制器，可以在 CPU 介入最少的情况下在存储器和外设传输数据。DMA 控制器有灵活的通道优先权选择，多种寻址模式，双重缓冲能力和块容量较大。

DMA 共有 4 个通道，每个通道都有独立的源地址、目的地址、触发器和块大小。不同的通道还有独立的控制设置、中断设置和中断向量。当一个传输结束或 DMA 控制器发现错误时，会产生中断请求。当 DMA 通道需要数据传输时，总线仲裁允许 DMA 控制器在 AVR CPU 不使用数据总线的时候传输数据。突发传输大小为 1，2，4 或 8 个字节。寻址方式可以是静态的、递增的或递减的。在突发传输和块传输结束后，自动加载源地址和目的地址。程序中，外设和事件都可以触发 DMA 传输。

1. DMA数据块传输的操作。DMA块传输分单次块传输和重复块传输。程序中单次块传输将memoryBlockA数组里面的100个数据一次性的传输到memoryBlockB数组里。重复块传输将memoryBlockA数组里面的100个数据分成10个大小相等的数据块，启动一次重复块传输，使memoryBlockA数组里面数据移到memoryBlockB数组里。通过串口打印memoryBlockB数组里面的数据加以验证。

C语言代码：

```
//--------包含头文件--------//
#include "dma_driver.h"
#include "avr_compiler.h"
#include "usart_driver.c"
#define MEM_BLOCK_SIZE   (10) //存储块大小
#define MEM_BLOCK_COUNT (10) //存储块个数
/*存储总的大小 在demo中 MEM_BLOCK_SIZE * MEM_BLOCK_COUNT不能大于64K*/
#define MEM_BLOCK ( MEM_BLOCK_SIZE * MEM_BLOCK_COUNT )
uint8_t memoryBlockA[MEM_BLOCK];
uint8_t memoryBlockB[MEM_BLOCK];
volatile bool gInterruptDone;
volatile bool gStatus;
//--------uart_init(串口初始化)-------//
void uart_init(void)
{
    /* USARTC0 引脚方向设置*/
    PORTC.DIRSET   = PIN3_bm; //PC3 (TXD0) 输出
    PORTC.DIRCLR   = PIN2_bm; // PC2 (RXD0) 输入
    USART_SetMode(&USARTC0,USART_CMODE_ASYNCHRONOUS_gc); //USARTC0 模式 - 异步
  // USARTC0帧结构, 8 位数据位, 无校验, 1停止位
    USART_Format_Set(&USARTC0, USART_CHSIZE_8BIT_gc,USART_PMODE_DISABLED_gc,
false);
    USART_Baudrate_Set(&USARTC0, 12 , 0); //设置波特率 9600
    USART_Tx_Enable(&USARTC0); //USARTC0 使能发送
    USART_Rx_Enable(&USARTC0); //USARTC0 使能接收
}
//--------BlockMemCopy（单次块传输初始化）-------//
bool BlockMemCopy( const void * src,
                   void * dest,
                   uint16_t blockSize,
                   volatile DMA_CH_t * dmaChannel )
{
    DMA_EnableChannel( dmaChannel ); //使能DMA通道
    DMA_SetupBlock( dmaChannel,
                    src,
                    DMA_CH_SRCRELOAD_NONE_gc,
                    DMA_CH_SRCDIR_INC_gc,
                    dest,
                    DMA_CH_DESTRELOAD_NONE_gc,
                    DMA_CH_DESTDIR_INC_gc,
                    blockSize,
                    DMA_CH_BURSTLEN_8BYTE_gc,
                    0,
```

```
                    false );
    DMA_StartTransfer( dmaChannel ); //请求传输
    return true;
}
//--------MultiBlockMemCopy（重复块传输初始化）-------//
bool MultiBlockMemCopy( const void * src, void * dest, uint16_t blockSize,
                        uint8_t repeatCount, volatile DMA_CH_t * dmaChannel )
{
    uint8_t flags;
    DMA_EnableChannel( dmaChannel ); //使能DMA通道
    /*设置源地址目的地址，块大小，以及地址的重载设置、模式设置，突发传输8个字节，
      重*复传输*/
    DMA_SetupBlock( dmaChannel,
                    src,
                    DMA_CH_SRCRELOAD_NONE_gc,
                    DMA_CH_SRCDIR_INC_gc,
                    dest,
                    DMA_CH_DESTRELOAD_NONE_gc,
                    DMA_CH_DESTDIR_INC_gc,
                    blockSize,
                    DMA_CH_BURSTLEN_8BYTE_gc,
                    repeatCount,
                    true );
    DMA_StartTransfer( dmaChannel );//请求传输

    do {
    flags = DMA_ReturnStatus_non_blocking( dmaChannel ); //等待出现错误或传输结
束
    } while ( flags == 0);
    dmaChannel->CTRLB |= ( flags );
    /*检测错误标志*/
    if ( ( flags & DMA_CH_ERRIF_bm ) != 0x00 ) {
        return false;
    } else {
        return true;
    }
}
//--------main（主函数入口）-------//
void main( void )
{
    uint32_t index;
    uart_init();
    volatile DMA_CH_t * Channel;
    Channel = &DMA.CH0;
```

```
    DMA_Enable();//进行单通道传输
    for ( index = 0; index < MEM_BLOCK; ++index )
    {
        memoryBlockA[index] = ( (uint8_t) index & 0xff );//赋值
    }
    /*重复数据块传输*/
    gStatus = MultiBlockMemCopy( memoryBlockA,
                                 memoryBlockB,
                                 MEM_BLOCK_SIZE,
                                 MEM_BLOCK_COUNT,
                                 Channel );
    if ( gStatus )
       {
              for ( index = 0; index < MEM_BLOCK; ++index)
                  uart_putdw_dec(memoryBlockB[index]);//通过串口输出
        }
    /*数据块单次传输*/
    if ( gStatus )
        {
        /*使能低级中断（当传输完成或者传输错误）*/
        DMA_SetIntLevel( Channel, DMA_CH_TRNINTLVL_LO_gc,
DMA_CH_ERRINTLVL_LO_gc );
        PMIC.CTRL |= PMIC_LOLVLEN_bm;
        sei();
         uart_putc('\n');
        for ( index = 0; index < MEM_BLOCK; ++index ) //重新填写发送数组
        {
            memoryBlockA[index] = 0xff - ( (uint8_t) index & 0xff );
        }
         uart_putc('\n');
        gInterruptDone = false; //标志位，在中断函数里面变成true
        /*数据块单次传输*/
        gStatus = BlockMemCopy( memoryBlockA,
                                memoryBlockB,
                                MEM_BLOCK,
                                Channel);
        do {} while ( gInterruptDone == false ); //等待传输结束
         uart_putc('\n');
        if ( gStatus )
                {
                   for ( index = 0; index < MEM_BLOCK; ++index )
                   uart_putdw_dec(memoryBlockB[index]); //通过串口输出
                 }
         uart_putc('\n');
```

```
        }
        while(1);
    }
//--------ISR（DMA出错中断和传输完成中断）-------//
ISR(DMA_CH0_vect)
{
    if (DMA.CH0.CTRLB & DMA_CH_ERRIF_bm) {
        DMA.CH0.CTRLB |= DMA_CH_ERRIF_bm;
        gStatus = false;
    } else {
        DMA.CH0.CTRLB |= DMA_CH_TRNIF_bm;
        gStatus = true;
    }
    gInterruptDone = true;
}
```

汇编代码：

```
//--------包含头文件--------//
.include "ATxmega128A1def.inc";器件配置文件,决不可少,不然汇编通不过
.include "usart_driver.inc"
.ORG 0
        RJMP RESET//复位
.ORG 0X100        ;跳过中断区0x00-0x0FF
.EQU ENTER  ='\n'
.EQU NEWLINE='\r'
.EQU EQUE   ='='
.EQU ZERO   ='0'
.EQU A_ASSCI='A'
.EQU MEM_BLOCK_SIZE=10//块大小
.EQU MEM_BLOCK_COUNT=10 //存储块个数
.EQU MEM_BLOCK=100//数据个数
//----- uart_init （串口初始化）--------//
uart_init:
       /* USARTC0 引脚方向设置*/
         LDI R16,0X08 //PC3 (TXD0) 输出
         STS PORTC_DIRSET,R16
         LDI R16,0X04 //PC2 (RXD0) 输入
         STS PORTC_DIRCLR,R16
        //USARTC0 模式 - 异步  USARTC0帧结构, 8 位数据位, 无校验, 1停止位
         LDI R16,USART_CMODE_ASYNCHRONOUS_gc|USART_CHSIZE_8BIT_gc
|USART_PMODE_DISABLED_gc
         STS USARTC0_CTRLC,R16
         LDI R16,12//设置波特率 9600
         STS USARTC0_BAUDCTRLA,R16
         LDI R16,0
```

```
          STS USARTC0_BAUDCTRLB,R16
          LDI R16,USART_TXEN_bm//USARTC0 使能发送  USARTC0 使能接收
          STS USARTC0_CTRLB,R16
           RET
//----- RESET(主函数入口)--------//
RESET:
         CALL uart_init
         LDI R16,DMA_ENABLE_bm
         STS DMA_CTRL,R16
         LDS R16,DMA_CH0_CTRLA
         ORI R16,DMA_CH_ENABLE_bm
         STS DMA_CH0_CTRLA,R16
         LDI XH,0X20 //写入要传的数据 初始地址是0x2000
         LDI XL,0X00
         LDI ZH,0X00
RESET_1:
         ST X+,ZH
         INC ZH
         CLZ
         CPI XL,100
         BRNE RESET_1
        /*重复数据块传输*/
         LDI R16,0X00//初始化源地址和目的地址 赋值顺序不能颠倒
         STS DMA_CH0_SRCADDR0,R16
         LDI R16,0X20
         STS DMA_CH0_SRCADDR1,R16
         LDI R16,0X00
         STS DMA_CH0_SRCADDR2,R16
         LDI R16,0X00
         STS DMA_CH0_DESTADDR0,R16
         LDI R16,0X30
         STS DMA_CH0_DESTADDR1,R16
         LDI R16,0X00
         STS DMA_CH0_DESTADDR2,R16
         LDI R16,DMA_CH_SRCRELOAD_NONE_gc|DMA_CH_SRCDIR_INC_gc|
DMA_CH_DESTRELOAD_NONE_gc|DMA_CH_DESTDIR_INC_gc
         STS DMA_CH0_ADDRCTRL,R16
         LDI R16,MEM_BLOCK_SIZE//块大小
         STS DMA_CH0_TRFCNT,R16
         LDI R16,0X00
         STS DMA_CH0_TRFCNT+1,R16
         LDS R16,DMA_CH0_CTRLA
         ORI R16,DMA_CH_BURSTLEN_8BYTE_gc|DMA_CH_REPEAT_bm
         STS DMA_CH0_CTRLA,R16
```

```
        LDI R16,MEM_BLOCK_COUNT//块数
        STS DMA_CH0_REPCNT,R16
        LDI R16,0X00
        STS DMA_CH0_REPCNT+1,R16
        LDS R16,DMA_CH0_CTRLA //请求传输
        ORI R16,DMA_CH_TRFREQ_bm
        STS DMA_CH0_CTRLA,R16
RESET_2:
        LDS R16,DMA_CH0_CTRLB
        ANDI R16,DMA_CH_ERRIF_bm|DMA_CH_TRNIF_bm
        CPI R16,0
        BREQ RESET_2
        LDS R16,DMA_CH0_CTRLB
        ORI R16,DMA_CH_ERRIF_bm|DMA_CH_TRNIF_bm
        STS DMA_CH0_CTRLB,R16
        /*读出数据*/
        LDI ZH,0X30//目的地址
        LDI ZL,0X00
RESET_3:
        LD XH,Z+
        uart_putc_hex
        CLZ
        CPI ZL,100
        in R16,CPU_SREG
        SBRS R16,1
        JMP RESET_3
        NOP
        LDI R16,DMA_CH_ENABLE_bm
        STS DMA_CH0_CTRLA,R16
        /*数据块单次传输*/
        LDI R16,0X00//初始化源地址和目的地址 赋值顺序不能颠倒
        STS DMA_CH0_SRCADDR0,R16
        LDI R16,0X20
        STS DMA_CH0_SRCADDR1,R16
        LDI R16,0X00
        STS DMA_CH0_SRCADDR2,R16
        LDI R16,0X00
        STS DMA_CH0_DESTADDR0,R16
        LDI R16,0X35
        STS DMA_CH0_DESTADDR1,R16
        LDI R16,0X00
        STS DMA_CH0_DESTADDR2,R16
        LDI R16,DMA_CH_SRCRELOAD_NONE_gc|DMA_CH_SRCDIR_INC_gc
|DMA_CH_DESTRELOAD_NONE_gc|DMA_CH_DESTDIR_INC_gc
```

```
        STS DMA_CH0_ADDRCTRL,R16
        LDI R16,MEM_BLOCK//块大小
        STS DMA_CH0_TRFCNT,R16
        LDI R16,0X00
        STS DMA_CH0_TRFCNT+1,R16
        LDS R16,DMA_CH0_CTRLA
        ORI R16,DMA_CH_BURSTLEN_8BYTE_gc
        STS DMA_CH0_CTRLA,R16
        LDS R16,DMA_CH0_CTRLA    //请求传输
        ORI R16,DMA_CH_TRFREQ_bm
        STS DMA_CH0_CTRLA,R16
RESET_4:
        LDS R16,DMA_CH0_CTRLB
        ANDI R16,DMA_CH_ERRIF_bm|DMA_CH_TRNIF_bm
        CPI R16,0
        BREQ RESET_4
        /*读出数据*/
        LDI ZH,0X35//目的地址
        LDI ZL,0X00
RESET_5:
        LD XH,Z+
        uart_putc_hex
        CLZ
        CPI ZL,100
        in R16,CPU_SREG
        SBRS R16,1
        JMP RESET_5
        NOP
RESET_6:
        JMP RESET_6
```

5.16 Boot Loader 实例

Boot Loader（程序代码位于 Flash 的引导区）可以同时读取和写入 Flash 程序存储器，用户签名行和 EEPROM，写锁定位。实现远程更新应用程序。

1. Boot Loader 更新程序区。通过串口中断接收数据写入程序区，再从程序区读出来，通过串口显示，执行完一次串口中断退出 bootloader 程序，跳到程序区执行。

说明：在下载 boot 区程序区时必须在 project->configuration options->memory setting 点击 add 添加 memory type 为 flash name 为.text address 为 0x10000 (这是 atsmage128a1 的 boot 区起始地址)，如果想执行程序区代码，boot 在下载时不能擦除 flash，可以直接选择十六进制文件下载。（一定不能选取擦除项，不然程序区也会被擦除）

C 语言代码：

```
//--------包含头文件--------//
```

```
#include "avr_compiler.h"
#include "usart_driver.c"
#include "TC_driver.c"
#include "sp_driver.h"
#define PROG_START  PROGMEM_START
uint16_t receivedata[255];
uint16_t readdata[255];
bool  volatile bootapp;
unsigned long int FlashAddr;
//---------quit（跳出bootloader区）--------//
void quit (void)
{
  CPU_CCP=CCP_IOREG_gc;
  PMIC.CTRL = 0X00;//将向量表移到程序区
  EIND=0X00;
  (*((void(*)(void))PROG_START))();
}
//---------uart_init(串口初始化)-------//
void uart_init(void)
{
    /* USARTC0 引脚方向设置*/
    PORTC.DIRSET   = PIN3_bm; //PC3 (TXD0) 输出
    PORTC.DIRCLR   = PIN2_bm; // PC2 (RXD0) 输入
    USART_SetMode(&USARTC0,USART_CMODE_ASYNCHRONOUS_gc); //USARTC0 模式 - 异步
  // USARTC0帧结构, 8 位数据位, 无校验, 1停止位
    USART_Format_Set(&USARTC0, USART_CHSIZE_8BIT_gc,USART_PMODE_DISABLED_gc,
false);
    USART_Baudrate_Set(&USARTC0, 12 , 0); //设置波特率 9600
    USART_Tx_Enable(&USARTC0); //USARTC0 使能发送
    USART_Rx_Enable(&USARTC0); //USARTC0 使能接收
}
//---------write_one_page (flash写一页)-------//
void write_one_page(unsigned char *buf)
{
    FlashAddr=0x5000;
    SP_LoadFlashPage(buf); //数据填入Flash缓冲页
    SP_EraseWriteApplicationPage(FlashAddr); //将缓冲页数据写入一个Flash页
    SP_WaitForSPM();//等待页编程完成
}
//---------main (主函数入口)-------//
void main( void )
{
    uart_init();
    uart_puts((void *)"enter into main");
```

```
    uart_putc('\n');
    bootapp=true;//标志位 在中断里面置成false
    TC_SetPeriod( &TCC0, 0x1000 );
    TC0_ConfigClockSource( &TCC0, TC_CLKSEL_DIV1024_gc );
    USART_RxdInterruptLevel_Set(&USARTC0,USART_RXCINTLVL_LO_gc);
    CPU_CCP=CCP_IOREG_gc;//I/O寄存器受保护的签名
    /*使能低级中断并且将中断向量表移到boot区*/
    PMIC.CTRL = PMIC_LOLVLEN_bm|PMIC_IVSEL_bm;
    sei();
    while(bootapp);
    quit();//退出boot区
}
//---------ISR (串口中断)-------//
ISR(USARTC0_RXC_vect)
{      uint8_t count_num=0;
       bootapp=false;
       TCC0_CNT=0;
       receivedata[count_num]= USART_GetChar(&USARTC0);//读Data Register
       while(TCC0_CNT<=20) //接收间隔大于20，接收结束
      {
        if((USARTC0.STATUS&0x80)==0x80)
        {
        count_num++;
        receivedata[count_num]= USART_GetChar(&USARTC0);//读Data Register
        TCC0_CNT=0;
        }
      }
      write_one_page(receivedata);
      FlashAddr=0x5000;
      for(uint8_t i=0;i<=count_num;i++)
      {
      readdata[i]=SP_ReadWord(FlashAddr);
      FlashAddr++;
      FlashAddr++;
      }
      for(uint8_t i=0;i<=count_num;i++)
      {
        while(!USART_IsTXDataRegisterEmpty(&USARTC0));
        USART_PutChar(&USARTC0,readdata[i]);
      }
}
```

汇编代码：

```
//---------包含头文件-------//
.include "ATxmega128A1def.inc";器件配置文件,决不可少,不然汇编通不过
```

```
.include "usart_driver.inc"
.ORG 0X10000        //设置从boot区启动
        JMP RESET  //复位
.ORG 0x10032         //USARTC0数据接收完毕中断入口
        JMP ISR
.ORG 0X10100        ;跳过中断区0x00-0x0F4
.EQU ENTER  ='\n'
.EQU NEWLINE='\r'
.EQU EQUE   ='='
.EQU ZERO   ='0'
.EQU A_ASSCI='A'
//----- uart_init （串口初始化）--------//
uart_init:
       /* USARTC0 引脚方向设置*/
         LDI R16,0X08 //PC3 (TXD0) 输出
         STS PORTC_DIRSET,R16
         LDI R16,0X04 //PC2 (RXD0) 输入
         STS PORTC_DIRCLR,R16
        //USARTC0 模式 - 异步  USARTC0帧结构, 8 位数据位, 无校验, 1停止位
         LDI R16,USART_CMODE_ASYNCHRONOUS_gc|USART_CHSIZE_8BIT_gc
|USART_PMODE_DISABLED_gc
         STS USARTC0_CTRLC,R16
         LDI R16,12//设置波特率 9600
         STS USARTC0_BAUDCTRLA,R16
         LDI R16,0
         STS USARTC0_BAUDCTRLB,R16
         LDI R16,USART_TXEN_bm//USARTC0 使能发送  USARTC0 使能接收
         STS USARTC0_CTRLB,R16
          RET
//----- RESET(主函数入口)--------//
RESET:
             CALL uart_init
              LDI R16,'M'
             PUSH R16
             uart_putc
             LDI R16,'A'
             PUSH R16
             uart_putc
             LDI R16,'I'
             PUSH R16
             uart_putc
             LDI R16,'N'
             PUSH R16
             uart_putc
```

```
            LDI R16,ENTER
            PUSH R16
            uart_putc //这里不能调用uart_puts_string因为这个过程里面使用了LPM
指令
            LDI XH,0X10
            EOR XL,XL
            STS TCC0_PER,XH
            STS TCC0_PER+1,XL
            LDI R16,TC_CLKSEL_DIV1024_gc//串口超过一定时间认为已经接收完毕
            STS TCC0_CTRLA,R16
            LDI R16,USART_RXCINTLVL_LO_gc //USARTC0 接收低级断级别
            STS USARTC0_CTRLA,R16
            LDI R17,CCP_IOREG_gc
            LDI R16,PMIC_LOLVLEN_bm|PMIC_IVSEL_bm//将中断向量表移到boot区
            STS CPU_CCP,R17
            STS PMIC_CTRL,R16//Enable Low_Level interrupts
            SEI
            LDI R16,0X00
            STS GPIO_GPIOR0,R16//如果变成1则跳转到程序区
 RESET_LOOP:
            LDS R16,GPIO_GPIOR0
            CLZ
            CPI R16,1
            BRNE  RESET_LOOP
            NOP
            LDI R16,0X00
            STS GPIO_GPIOR0,R16//如果变成1则跳转到程序区
            JMP QUIT_BOOT
            JMP RESET_LOOP//永远也不会到这一步
//---------RECEIVE_DATA （串口接收数据）-------//
RECEIVE_DATA:
             LDI R18,0X00//记录接收字节数,其他地方不能使用以免破坏数据
             LDI R31,0x20//串口数据起始储存地址Z=0x2000
             LDI R30,0x00
STORE_DATA:
             LDS R16,USARTC0_DATA//读数据寄存器
             ST  Z+,R16
             INC R18//接收字符数加1
             LDI R16,0X00//计数器清零
             STS TCC0_CNT,R16
             STS TCC0_CNT+1,R16
WAITING_RECEIVE:
             LDS R16,TCC0_CNT
             CPI R16,5
```

```
                BRSH RECEIVE_END //接收间隔大于5，接收结束
                LDS R16,USARTC0_STATUS
                SBRS R16,7 //有数据来了就存
                JMP WAITING_RECEIVE
                JMP STORE_DATA
RECEIVE_END:
                RET
//---------SP_LoadFlashPage（写flash缓冲）-------//
SP_LoadFlashPage:
                IN R21,CPU_RAMPZ
                LDI R16,0X00
                OUT   CPU_RAMPZ,R16
                CLR   ZL
                CLR   ZH
                LDI   XH,0X20
                LDI   XL,0X00
                LDI   r20, NVM_CMD_LOAD_FLASH_BUFFER_gc
                STS   NVM_CMD, r20
                LDI   r19, CCP_SPM_gc
                MOV   R17,R18//要写入的数据个数
SP_LoadFlashPage_1:
                LD    R0, X+
                MOV   R1,R0
                STS   CPU_CCP, r19
                SPM
                ADIW  ZL, 2
                DEC R17
                CLZ
                CPI R17,0
                BRNE SP_LoadFlashPage_1
                OUT CPU_RAMPZ,R21
                CLR R1
                RET
 //---------SP_EraseWriteApplicationPage（擦除并写flash一页）-------//
SP_EraseWriteApplicationPage:
                IN    r21, CPU_RAMPZ
                LDI R16,0X00
                OUT   CPU_RAMPZ,R16
                LDI   ZH,0X50
                LDI   ZL,0X00//FLASH地址
                OUT   CPU_RAMPZ,ZL
                LDI   r20, NVM_CMD_ERASE_WRITE_APP_PAGE_gc
                STS   NVM_CMD, r20
                LDI   r19, CCP_SPM_gc
```

```
                STS   CPU_CCP, r19
                SPM
                CLR R1
                OUT   CPU_RAMPZ, r21
                RET
  //--------SP_WaitForSPM（等待写完成）-------//
SP_WaitForSPM:
                LDS   r16, NVM_STATUS
                SBRC  r16, NVM_NVMBUSY_bp
                RJMP  SP_WaitForSPM
                CLR   r16
                STS   NVM_CMD, r16
                RET
//---------SP_ReadWord（读取flash里面的数据）-------//
SP_ReadWord:
                IN  r21, CPU_RAMPZ
                LDI R16,0X00
                OUT CPU_RAMPZ,R16
                LDI ZH,0X50
                LDI ZL,0X00//读flash的地址
                LDI XH,0X30
                LDI XL,0X00//SDRAM中存放从flash读出来数据的地址
                LDI R19,0X00//计数
SP_ReadWord_1:
                ELPM  R24, Z
                ST X+,R24
                ADIW ZL,2
                INC R19
                CLZ
                CP R18,R19
                BRNE  SP_ReadWord_1
                OUT   CPU_RAMPZ, r21
                RET
//---------SEND_DATA（向串口发送读取的数据）-------//
SEND_DATA:
                LDI XH,0X30
                LDI XL,0X00//SDRAM中存放从flash读出来数据的地址
                LDI R17,0X00//计数
SEND_DATA_1:
                LD R16,X+
                USART_IsTXDataRegisterEmpty USART_DREIF_bp
                STS USARTC0_DATA,R16
                INC R17
                CLZ
```

```
                CP R17,R18
                BRNE SEND_DATA_1
                RET
//--------——QUIT_BOOT（退出BOOT区）-------//
QUIT_BOOT:
                LDI R17,CCP_IOREG_gc
                LDI R16,0X00
                STS CPU_CCP,R17
                STS PMIC_CTRL,R16//将中断向量表搬到程序区
                JMP 0X0000
//---------——ISR（串口接收中断）-------//
ISR:
                CALL RECEIVE_DATA//串口接收数据
                CALL SP_LoadFlashPage//写flash缓冲区
                CALL SP_EraseWriteApplicationPage//擦除并写一页
                CALL SP_WaitForSPM//等待写完
                CALL SP_ReadWord//读数据
                CALL SEND_DATA//串口返回读的数据
                 LDI R16,0X01
                STS GPIO_GPIOR0,R16//如果变成1则跳转到程序区
                RETI
```