MSP430F5438A操作手册

**作者:向仔州**

IAR7.12版本安装.............................................................................................................................2

IAR5.3版本工程移植到IAR7.12 IDE进行编译，出现问题汇总………………………………….…………...14

问题1报错: Error[e183]: Static overlay map generation (-xo) is not supported for the MSP430 processor………………………………………………………………………………………………………………………..………...14

问题2: 工程路径存放太深，会出现The following path is too long:…………………..………….14

问题3: 将IAR5.3 的工程用IAR7.12打开，代码无法跳转...............................................14

时钟初始化....................................................................................................................................15

串口0使用....................................................................................................................................17

串口中断接受数据…………………………………………………………………………………………………..……....19

串口Printf实现......................................................................................................................19

串口3使用……………………………………………………………………………………………………………..…….....20

串口3中断接收数据……………………………………………………………………………………………..………...21

程序运行正常。但是只能单字节收发，多字节收发就出问题了…………………………..……...21

GPIO输入输出操作........................................................................................................................22

如果想实现按键双边沿中断操作.........................................................................................24

GPIO外部中断...............................................................................................................................25

MSP430F5438A定时器使用..........................................................................................................26

定时器输入捕获比较，频率周期采集.................................................................................28

用P8.1/TA0.1对输入信号频率进行采集.............................................................................29

如果采集随机双脉冲之间的间隔，那么频率采集方式就不合适………………………….……….31

定时器B使用................................................................................................................................32

定时器B指定计数器计数次数定时.....................................................................................33

ADC12 模数转换器使用................................................................................................................33

ADC单次转换实验，每次转换后都用while轮询获取.......................................................34

中断方式单通道读取ADC数据............................................................................................37

ADC中断的数据如果放在定时器中断等待采集就会出BUG，注意.........................39

MSP430F5438A内部flash做eeprom存储..................................................................................40

模拟SPI操作OLED屏...................................................................................................................44

MSP430 标准modbus协议移植(支持poll软件)........................................................................51

对以上MODBUS代码进行逻辑解析....................................................................................56

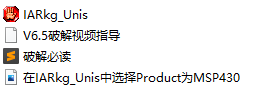
实现MODBUS地址越界判断................................................................................................59

**IAR7.12版本安装**

安装IAR7.12版本是为了解决仿真器固件升级，导致以前的IAR5.3 IDE无法下载问题。

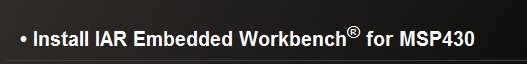
本来IAR5.3使用JTAG MSP-FET430UIF仿真器下载程序很正常，但是只要使用一次IAR7.12在MSP-FET430UIF仿真器上下载MSP430程序，那么MSP-FET430UIF里面固件就会升级，这时候再使用IAR5.3去下载就不行了，只要使用IAR7.12 IDE下载。

这是IAR7.12安装软件

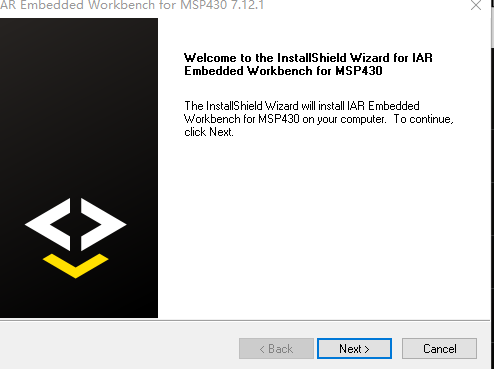
 IARkg\_Unis是破解软件

运行安装软件

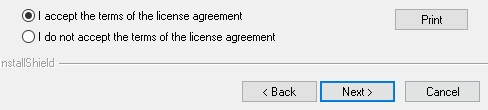




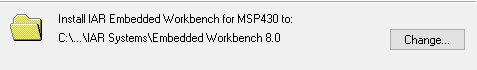
选择第1项安装



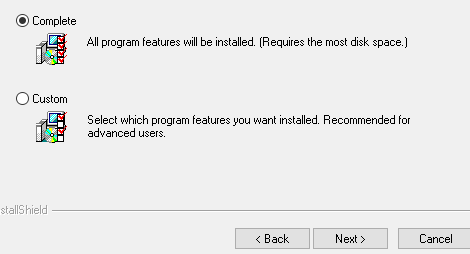
点击下一步



选择第1项

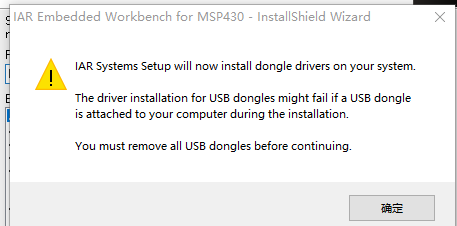


记住安装路径

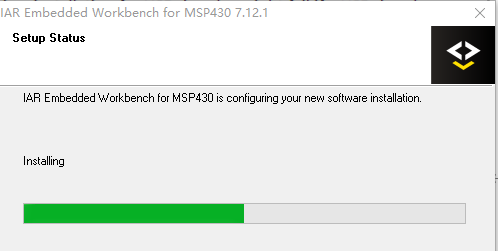


点击下一步

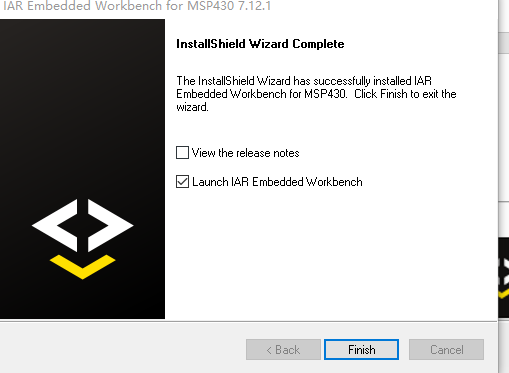
选择第1项



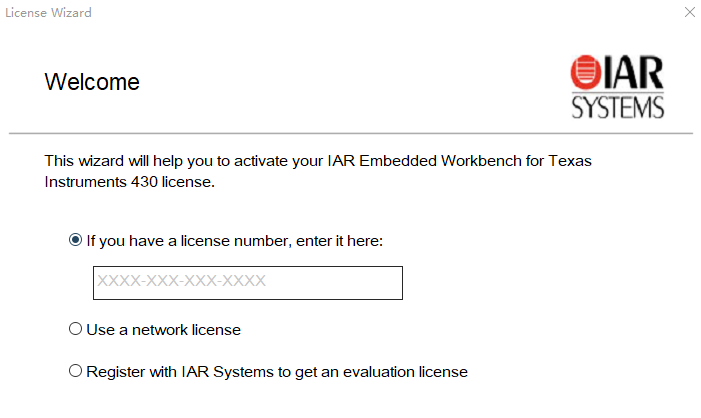
直接点击确定



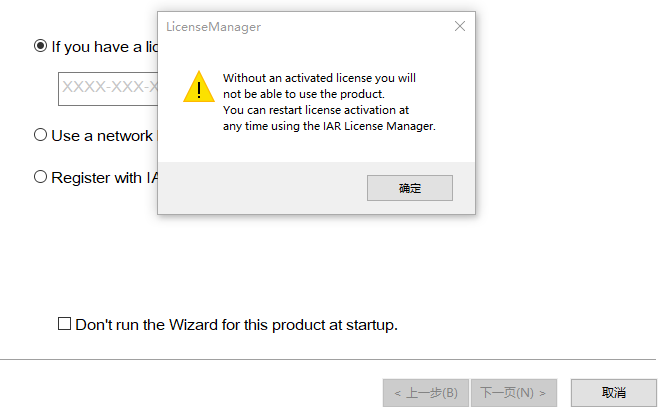
正在安装

点击finish，软件第一次安装完成

取消



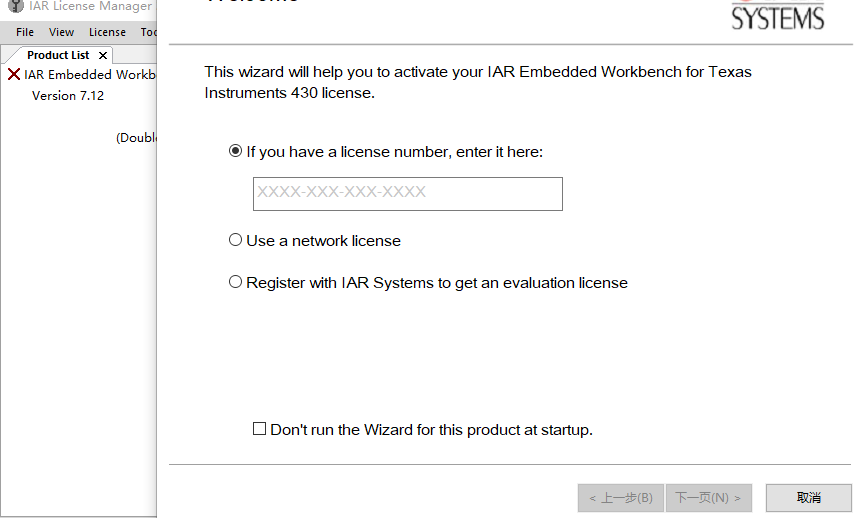
IAR自动启动，会弹出注册窗口



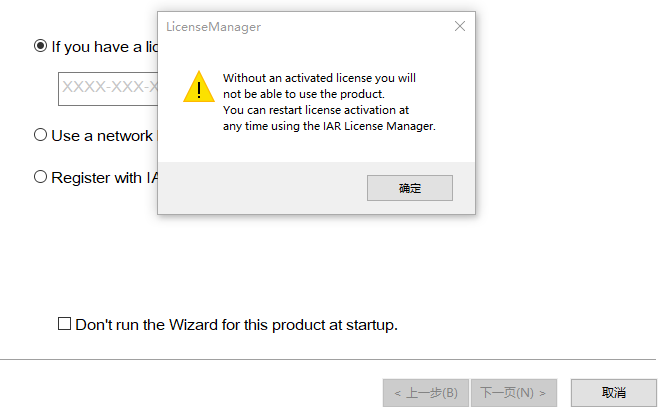
不管警告，直接点击确定

点击取消

点击IAR 🡪 help 🡪 license Manager

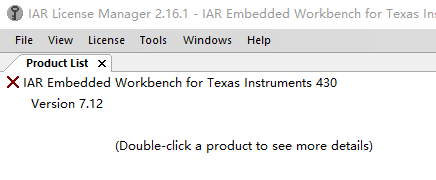


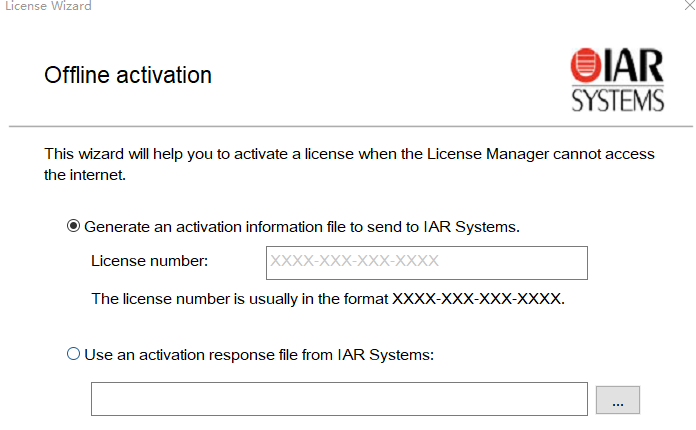
点击取消



点击确定

点击顶部License -> Offline activation

窗口停留在product list



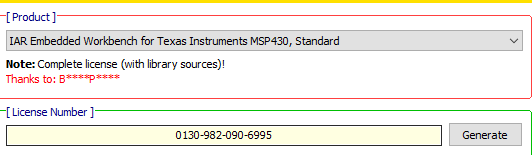
保持界面不动

打开破解软件

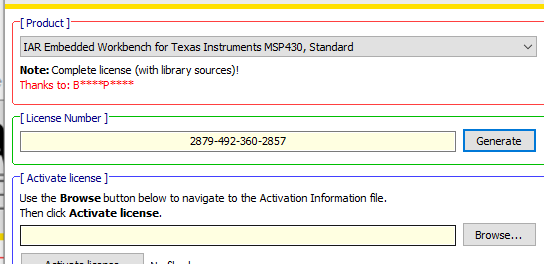




破解软件默认ARM



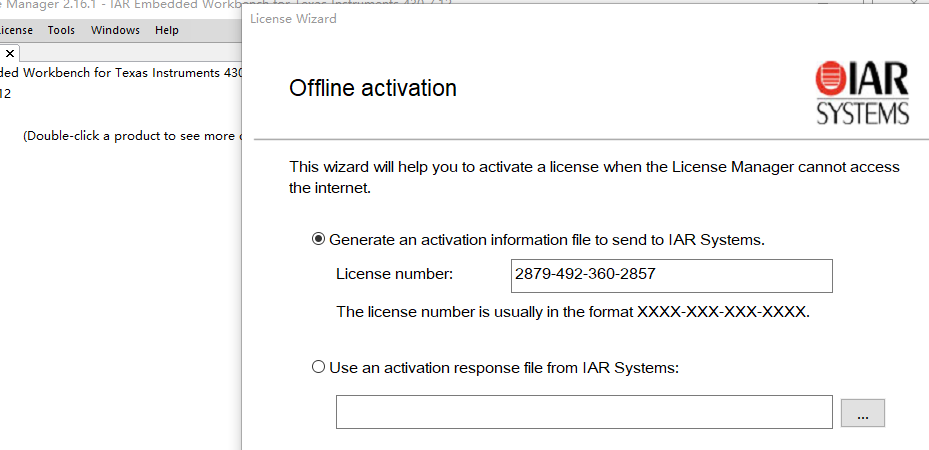
将ARM选成MSP430



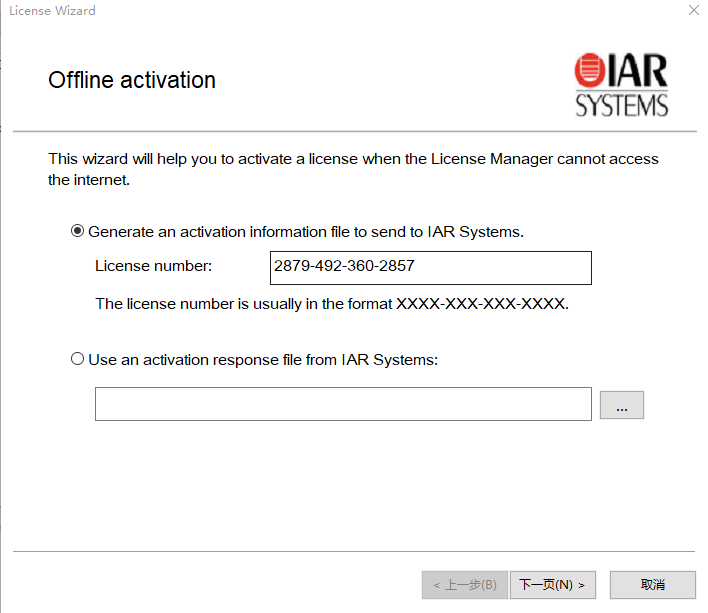
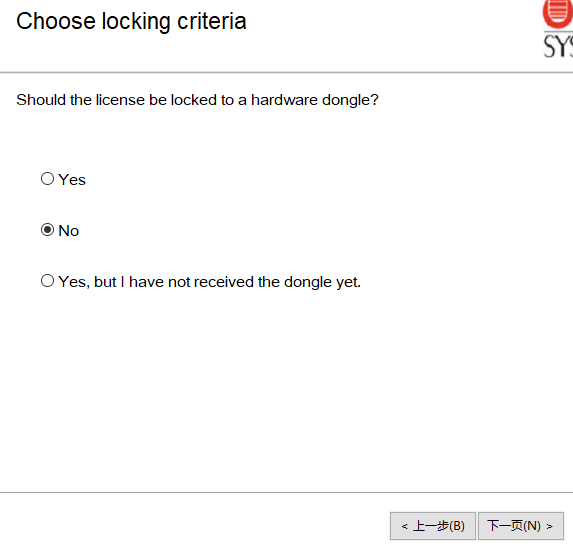
复制密匙

选择MSP430之后点击Generate





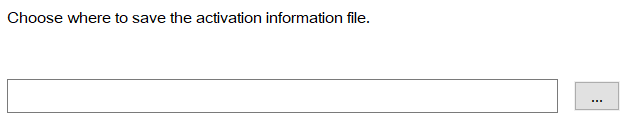
将密匙写入IAR

继续点击下一页

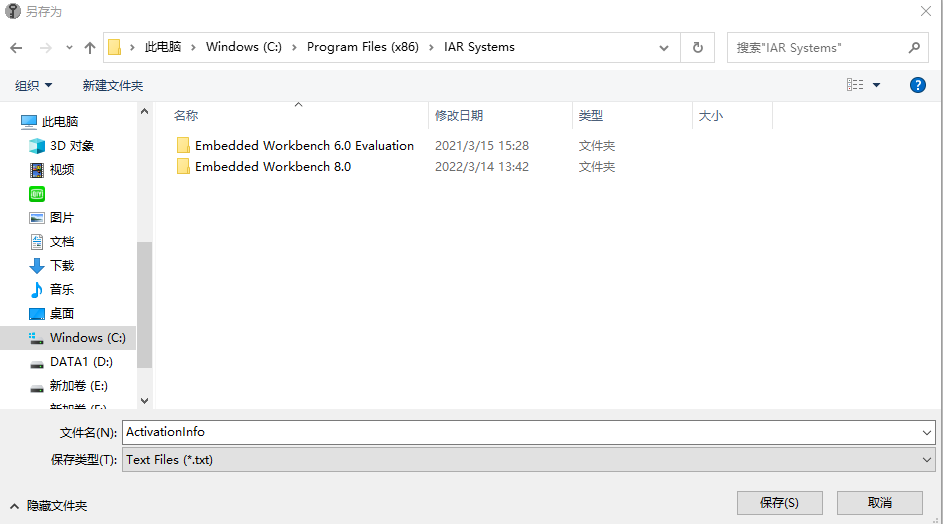
记住，选择No

点击下一页

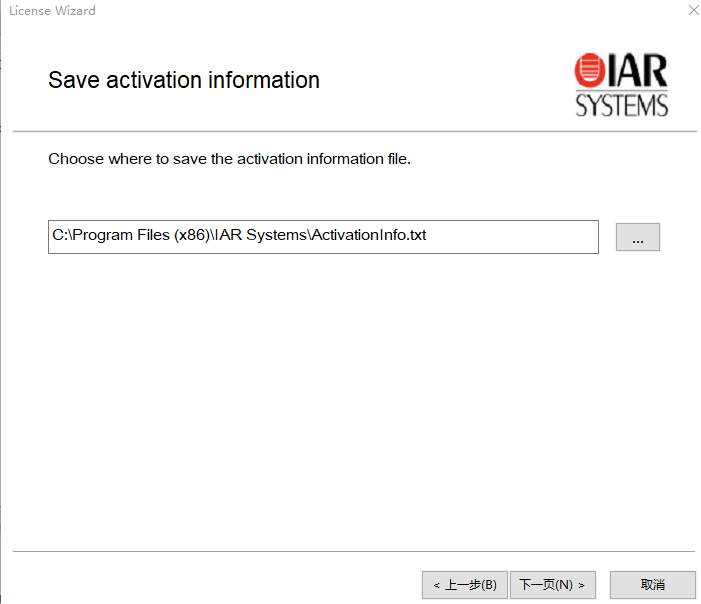


进入加载文件页面

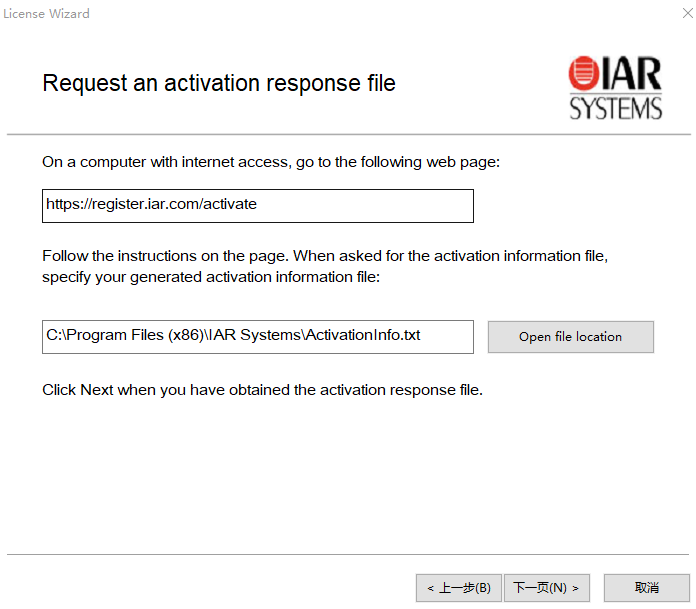
选择保存路径



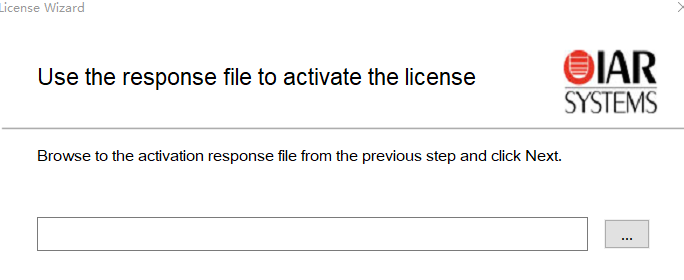
将生成的’Activationinfo保存到自己IDE安装的根目录’



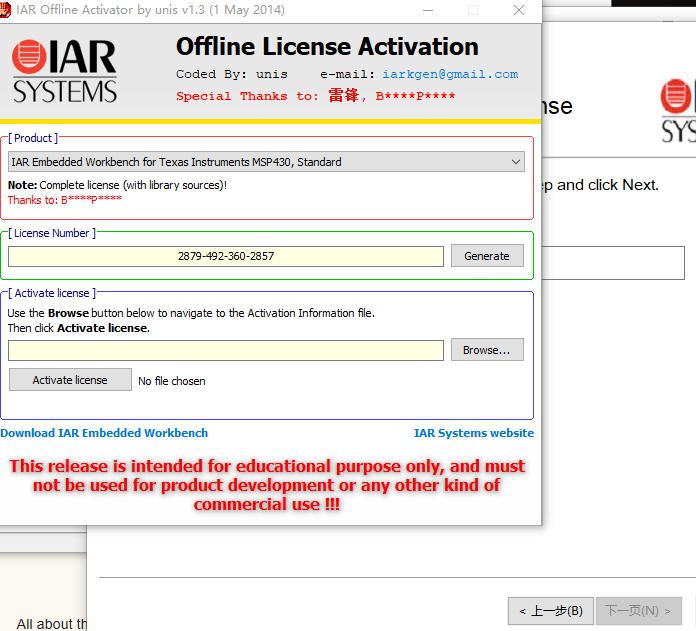
点击下一页



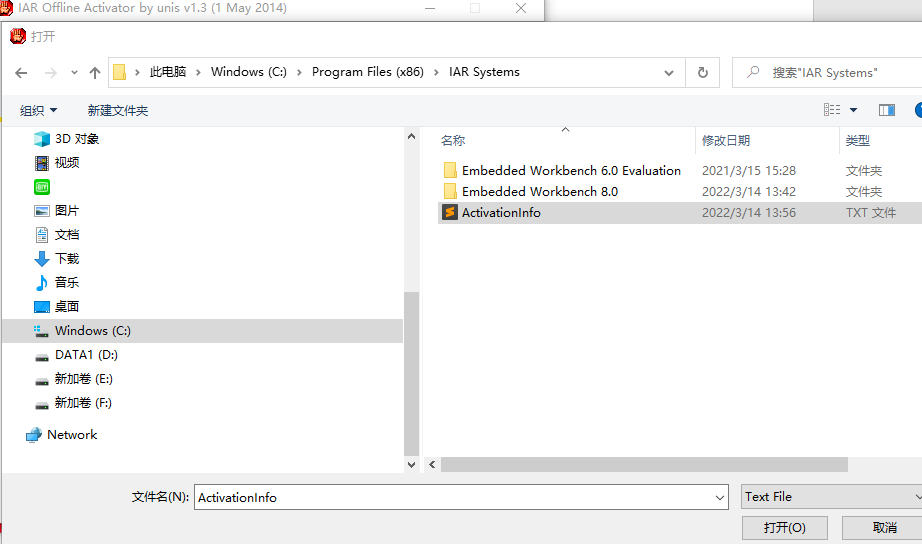
点击下一页



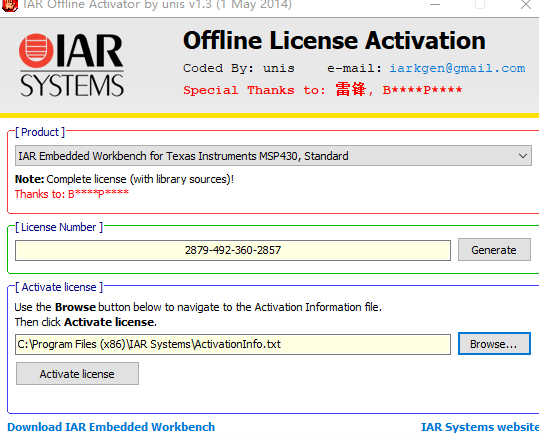
再次进入路径选项



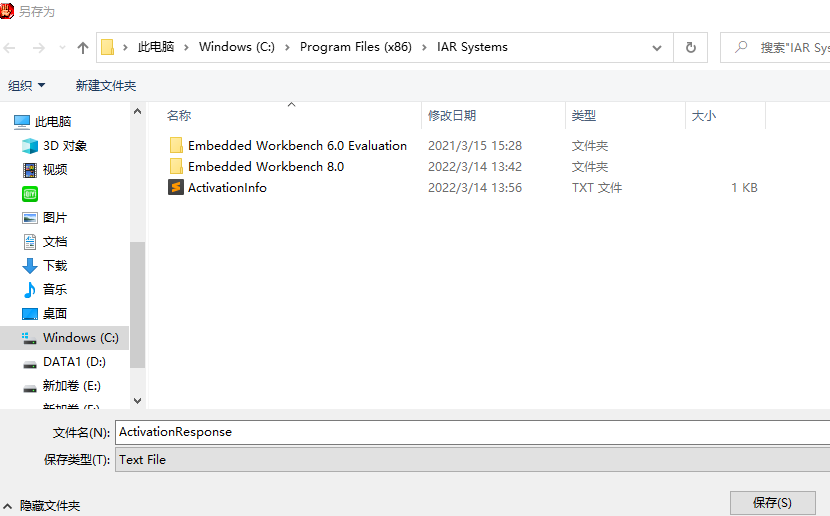
切换到破解软件，点击Browse



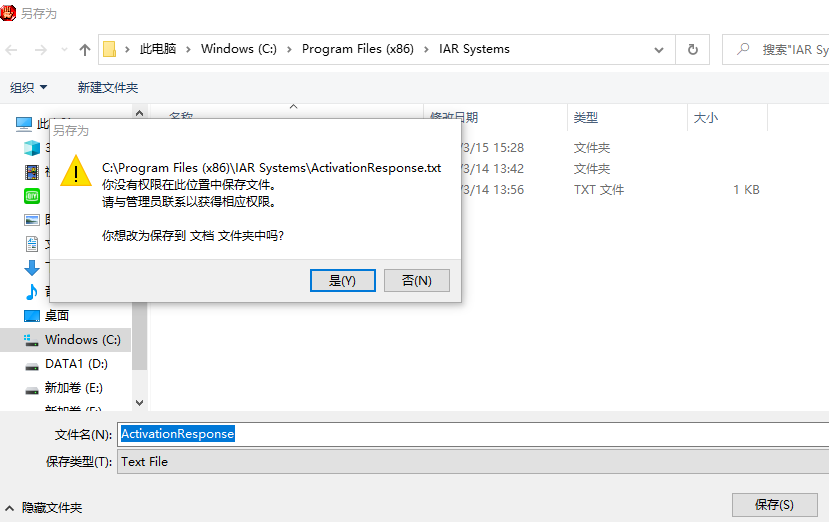
将上面生成的‘Activationinfo’文件导入进破解软件



点击Activate license

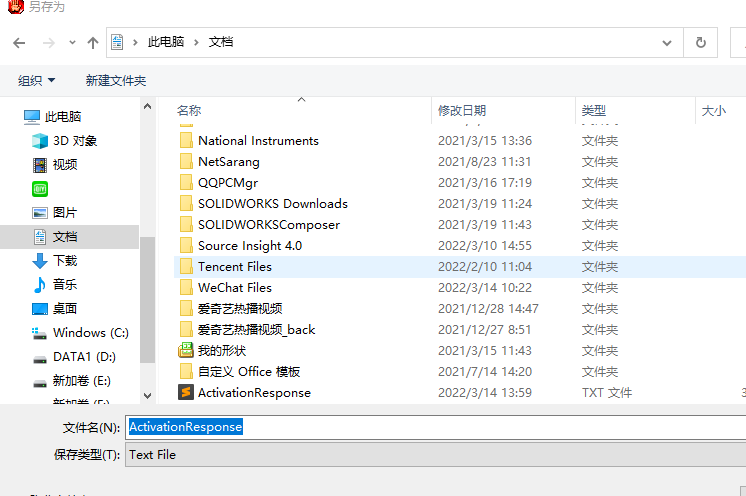


将’ActivationResponse’文件保存在安装根目录下

因为没有权限，所以无法保存到安装目录下，这是要注意

点击保存

强制点击‘是’

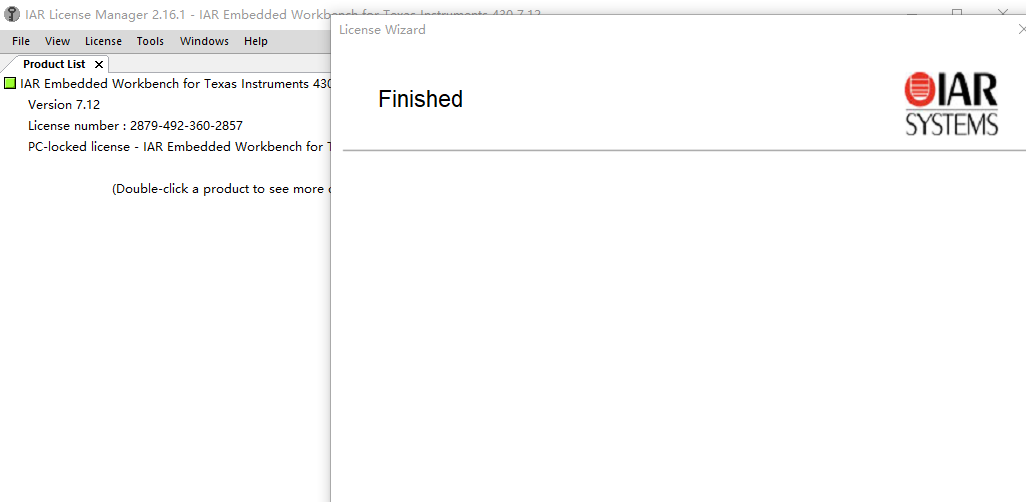
你会发现’ActivationResponse’默认保存在‘文档’，记住‘文档’路径

再次回到IAR IDE窗口中

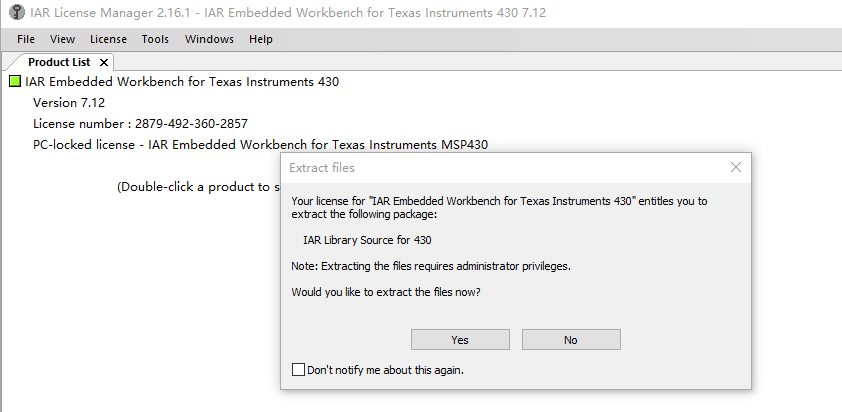


点击下一页

加载‘文档路径的’ActivationResponse

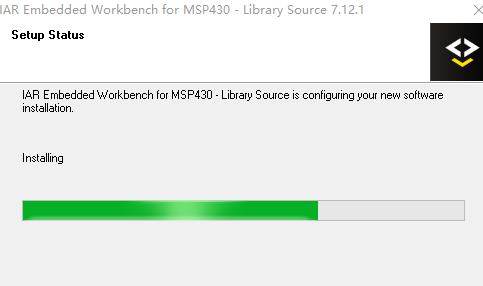


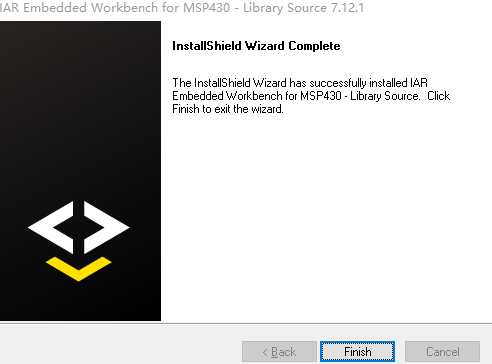
显示绿色，破解成功

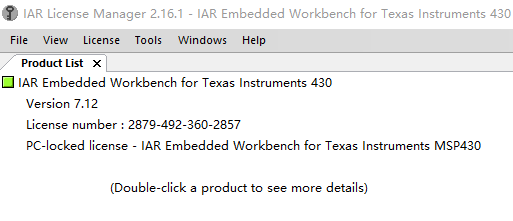


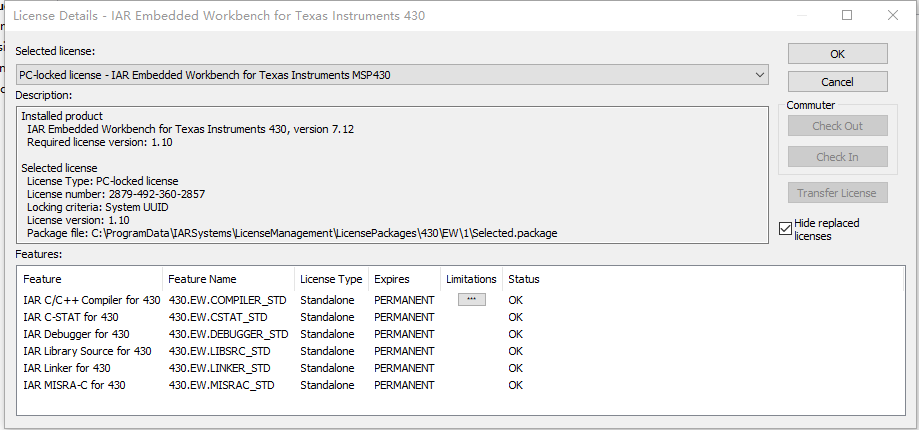
点击‘yes’

再次进入安装模式



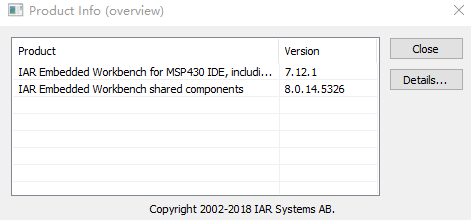
安装完成

双击Version 7.12



弹出该对话框内容，证明破解安装完成。

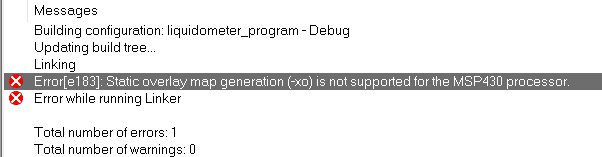
再次在IAR 软件点击help -> About -> Product Info

IAR激活完毕

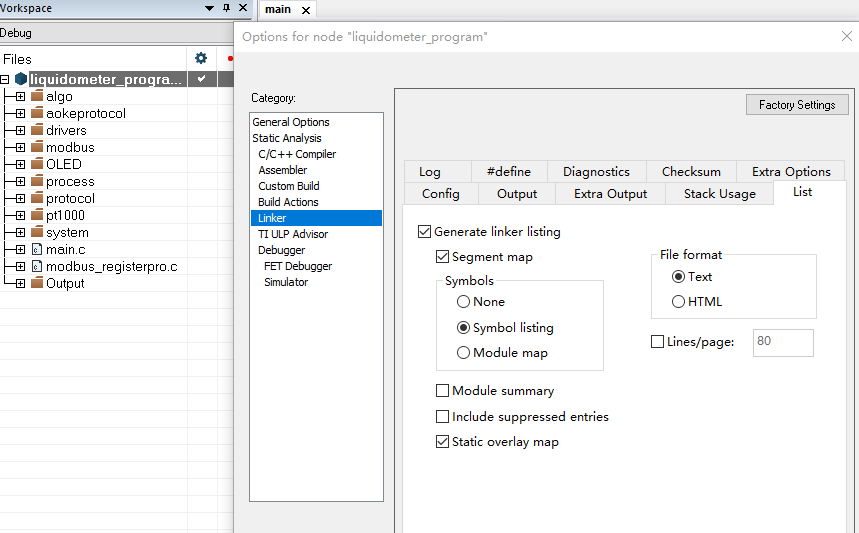
**IAR5.3版本工程移植到IAR7.12 IDE进行编译，出现问题汇总**

问题1报错: Error[e183]: Static overlay map generation (-xo) is not supported for the MSP430 processor

编译单独的C文件不会出现该问题，一定是点击Make，链接文件的时候出现。



打开工程的Options



取消Static Overlay map选项

再次make就正常了。

问题2: 工程路径存放太深，会出现The following path is too long:

D盘: \目录1\目录2\目录3\目录4\目录5\目录6\目录7\目录8\目录9\工程文件



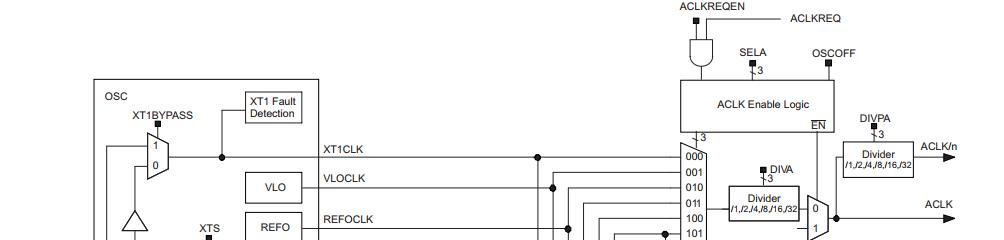
这种问题只有将工程拷贝到桌面进行修改，修改后又覆盖进原来深度目录下的工程做备份。

问题3: 将IAR5.3 的工程用IAR7.12打开，代码无法跳转

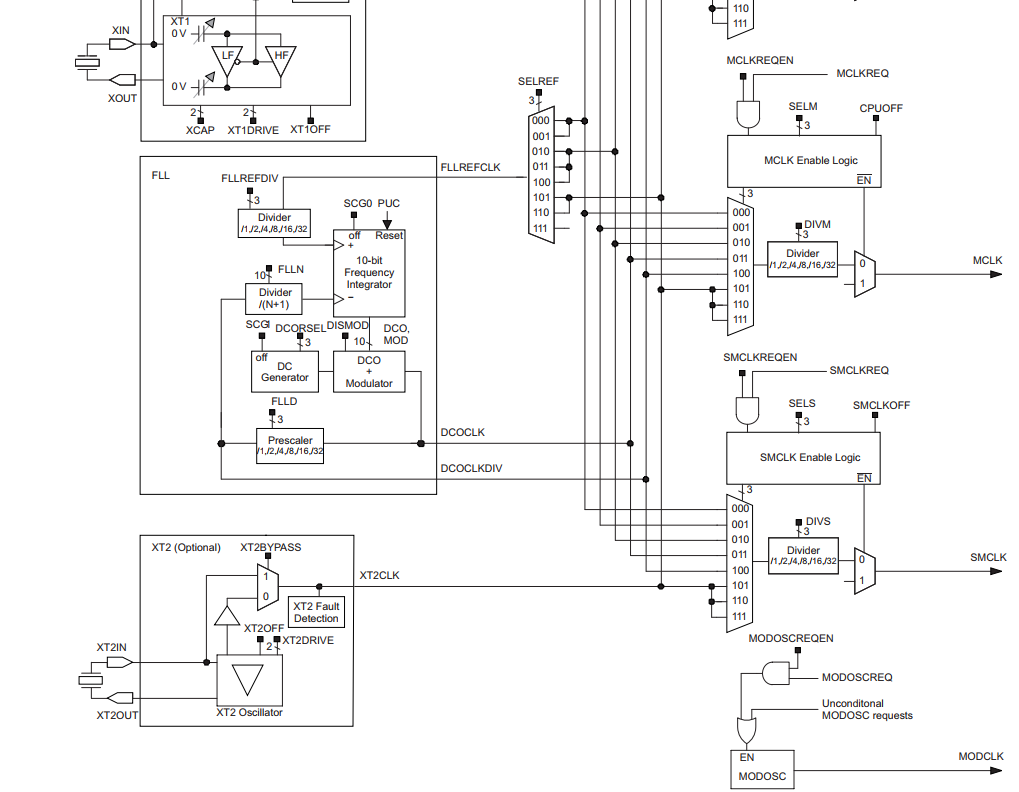
这种情况只有先clean清空工程，重新点击Make构建工程，就可以正常跳转了。

**时钟初始化**

3：ACLK时钟来源也选择REFOCLK



2: FLL时钟源选择REFOCLK

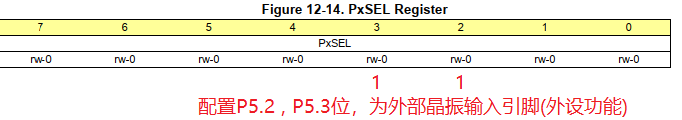


6:选择MCLK时钟源，为XT2外部时钟输入

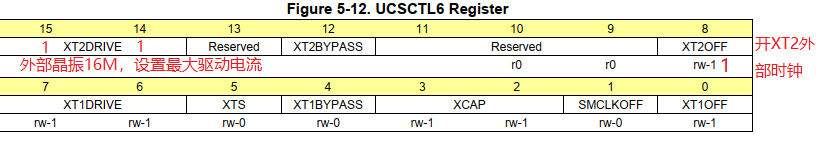
5:选择SMCLK时钟源，为XT2外部时钟输入

4.根据外部接入晶振频率，设置XT2驱动能力

1:开启XT2时钟，外部晶振输入

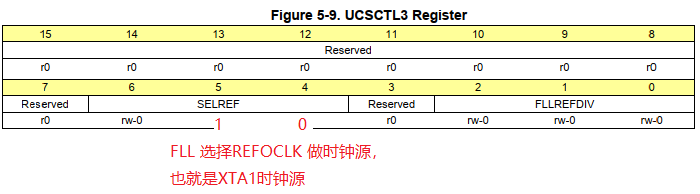


P5SEL |= BIT2+BIT3; // P5.2和P5.3是 XT2 脚，24MHz晶振

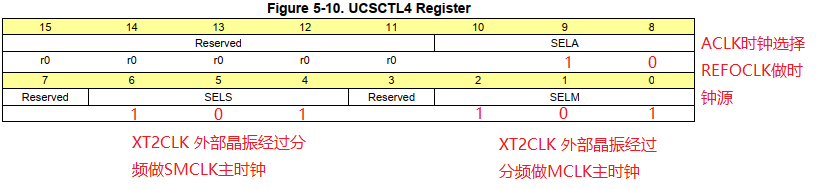


UCSCTL6 &= ~(XT2OFF); //开启XTO2时钟

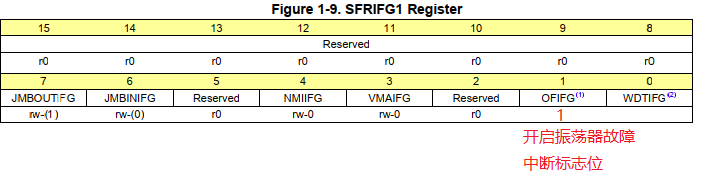
UCSCTL6 = XT2DRIVE1 + XT2DRIVE0; //驱动晶振的能力是24M到32M



UCSCTL3 |= SELREF\_2; // FLLref = REFO



UCSCTL4 |= SELA\_2;



UCSCTL4 |= SELS\_5+SELM\_5;

时钟代码例程

#include "msp430x54x.h"

/\*官方库自定义延时函数\*/

#define CPU\_F ((double)16000000) //外部高频晶振为16M

//#define CPU\_F ((double)32768) //如果用32.768khz做CPU时钟，就选这句

#define delay\_us(x) \_\_delay\_cycles((long)(CPU\_F \* (double)x/1000000.0)) //除以1000000微妙

#define delay\_ms(x) \_\_delay\_cycles((long)(CPU\_F \* (double)x/1000.0)) //除以1000毫秒

int main( void )

{

// Stop watchdog timer to prevent time out reset

WDTCTL = WDTPW + WDTHOLD;

unsigned char k = 0;

P5SEL |= BIT2+BIT3; // P5.2和P5.3是 XT2 脚，24MHz晶振

UCSCTL6 &= ~(XT2OFF); //开启XTO2时钟

UCSCTL3 |= SELREF\_2; // FLLref = REFO

UCSCTL4 |= SELA\_2; // ACLK=REFO,SMCLK=DCO,MCLK=DCO

do

{

UCSCTL7 &= ~(XT2OFFG + XT1LFOFFG + XT1HFOFFG + DCOFFG); //清除TX1,XT2和DCO振荡标志

SFRIFG1&=~OFIFG; //清除振荡器失效标志，清除中断标志

for(k=0xff;k>0;k--) //延时

{;}

}while((SFRIFG1&OFIFG)!=0);//判断XT2是否起振,等待时钟系统正常工作

UCSCTL6 = XT2DRIVE1 + XT2DRIVE0; //驱动晶振的能力是24M到32M

UCSCTL4 |= SELS\_5+SELM\_5;

/\*P1.7 LED IO 配置\*/

P1DIR |= 0x80; //1000 0000 P1.7输出模式

while(1)

{

P1OUT ^= BIT7;

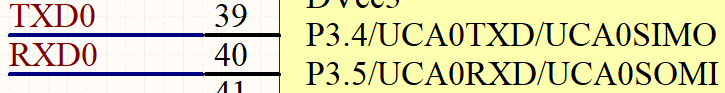
delay\_ms(1000); // 软件延时

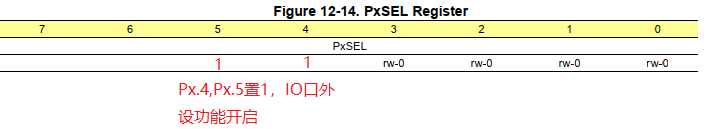
}

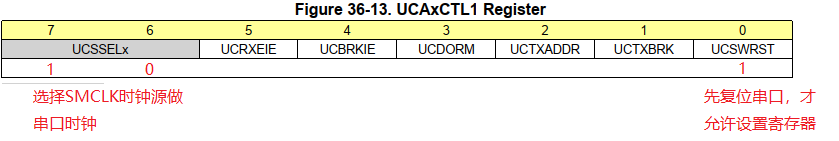
}

**串口0使用**

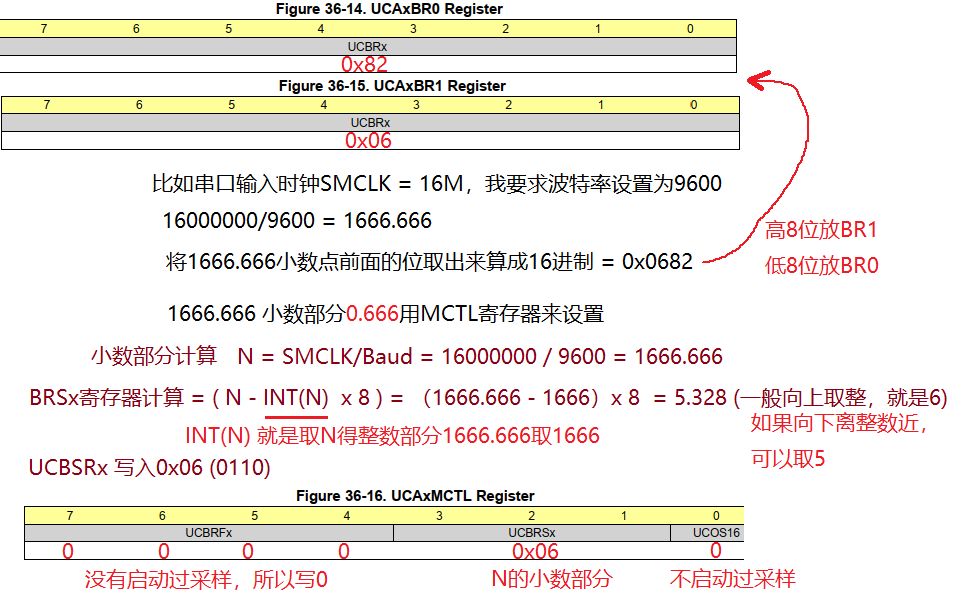
串口发送







UCA0CTL1 |= UCSSEL\_2;



UCA0BR0 = 0x82;

UCA0BR1 = 0x06;

UCA0MCTL = UCBRS\_3+UCBRF\_0;

#include "msp430x54x.h"

/\*官方库自定义延时函数\*/

#define CPU\_F ((double)16000000) //外部高频晶振为16M

//#define CPU\_F ((double)32768) //如果用32.768khz做CPU时钟，就选这句

#define delay\_us(x) \_\_delay\_cycles((long)(CPU\_F \* (double)x/1000000.0)) //除以1000000微妙

#define delay\_ms(x) \_\_delay\_cycles((long)(CPU\_F \* (double)x/1000.0)) //除以1000毫秒

/\*时钟初始化\*/

void SysInit(void)

{

unsigned char k = 0;

P5SEL |= BIT2+BIT3; // P5.2和P5.3是 XT2 脚，24MHz晶振

UCSCTL6 &= ~(XT2OFF); //开启XTO2时钟

UCSCTL3 |= SELREF\_2; // FLLref = REFO

UCSCTL4 |= SELA\_2; // ACLK=REFO,SMCLK=DCO,MCLK=DCO

do

{

UCSCTL7 &= ~(XT2OFFG + XT1LFOFFG + XT1HFOFFG + DCOFFG); //清除TX1,XT2和DCO振荡标志

SFRIFG1&=~OFIFG; //清除振荡器失效标志，清除中断标志

for(k=0xff;k>0;k--) //延时

{;}

}while((SFRIFG1&OFIFG)!=0);//判断XT2是否起振,等待时钟系统正常工作

UCSCTL6 = XT2DRIVE1 + XT2DRIVE0; //驱动晶振的能力是24M到32M

UCSCTL4 |= SELS\_5+SELM\_5;

}

/\*\*\*\*\*\*\*\*\*\*串口0初始化函数\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void usartInit(void)

{

P3SEL = BIT4 + BIT5; // 选择端口的第二功能，P3.4,P3.5 = USCI\_A0 TXD/RXD

UCA0CTL1 |= UCSWRST; // 状态机复位

UCA0CTL1 |= UCSSEL\_2; // 选择串口时钟源，UCSSEL\_1(CLK = ACLK) UCSSEL\_2(CLK = SMCLK)

UCA0BR0 = 0x82; // 两个寄存器配置串口的波特率，16M/9600=1666.66 (1666十六进制0x0682)

UCA0BR1 = 0x06; //也可以用专用波特率计算器，计算波特率得到的值

UCA0MCTL = UCBRS\_3+UCBRF\_0; // UCBRSx=3, UCBRFx=0

UCA0CTL1 &= ~UCSWRST; // 状态机置位

}

int main( void )

{

// Stop watchdog timer to prevent time out reset

WDTCTL = WDTPW + WDTHOLD;

SysInit();

usartInit();

while(1)

{

UCA0TXBUF = 0xA1; //串口发送数据

while(!(UCA0IFG&UCTXIFG)); //等待发送完成

delay\_ms(1000); // 软件延时

}

}

串口发送成功

串口中断接受数据

/\*\*\*\*\*\*\*\*\*\*串口0初始化函数\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void usartInit(void)

{

P3SEL = BIT4 + BIT5; // 选择端口的第二功能，P3.4, P3.5 = USCI\_A0 TXD/RXD

UCA0CTL1 |= UCSWRST; // 状态机复位

UCA0CTL1 |= UCSSEL\_2; // 选择串口时钟源，UCSSEL\_1(CLK = ACLK)UCSSEL\_2(CLK = SMCLK)

UCA0BR0 = 0x82; // 两个寄存器配置串口的波特率，16M/9600=1666.66 (1666十六进制0x0682)

UCA0BR1 = 0x06;

UCA0MCTL = UCBRS\_3+UCBRF\_0; // UCBRSx=3, UCBRFx=0

UCA0CTL1 &= ~UCSWRST; // 状态机置位

UCA0IE |= UCRXIE; // 使能 USCI\_A0 RX接收中断

}

int main( void )

{

// Stop watchdog timer to prevent time out reset

WDTCTL = WDTPW + WDTHOLD;

SysInit();

usartInit();

\_EINT(); //开总中断

while(1)

{

delay\_ms(1000); // 软件延时

}

}

#pragma vector=USCI\_A0\_VECTOR //串口0接收中断服务程序

\_\_interrupt void USCI\_A0\_ISR(void)

{

unsigned char rxData;

switch(\_\_even\_in\_range(UCA0IV,4))

{

case 0:break; // Vector 0 - no interrupt

case 2:

rxData = UCA0RXBUF; // Vector 2 - RXIFG 接收中断

UCA0TXBUF = rxData; //接受的数据转发

while(!(UCA0IFG&UCTXIFG));

break;

case 4:break; // Vector 4 - TXIFG 发送中断

default: break;

}

}

串口Printf实现

#include "msp430x54x.h"

#include <stdio.h>

int main( void )

{

// Stop watchdog timer to prevent time out reset

WDTCTL = WDTPW + WDTHOLD;

SysInit();

usartInit(); //串口初始化

\_EINT(); //开总中断

while(1)

{

printf("MSP430F5438A TEST ..\n");

delay\_ms(1000); // 软件延时

}

}

/\*

\* 主要实现printf底层的putchar函数重写

\*/

int putchar(int ch)

{

if(ch == '\n')

{

while(!(UCA0IFG&UCTXIFG)); // 判断:发送是否完成

UCA0TXBUF = 0x0d; // 装入发送寄存器

}

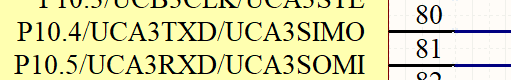
while(!(UCA0IFG&UCTXIFG)); // 判断:发送是否完成

UCA0TXBUF = ch; // 装入发送寄存器

return (ch);

}

串口3使用



/\*

\* 串口3初始化 9600

\*/

void uart3\_init(void)

{

P10SEL = BIT4 + BIT5; // 选择端口的第二功能，P10.4,P10.5 = TXD3 RXD3

UCA3CTL1 |= UCSWRST; // 状态机复位

UCA3CTL1 |= UCSSEL\_2; // 选择串口时钟源，UCSSEL\_1(CLK = ACLK) UCSSEL\_2(CLK = SMCLK)

UCA3BR0 = 0x82; // 两个寄存器配置串口的波特率，16M/9600=1666.66 (1666十六进制0x0682)

UCA3BR1 = 0x06; //也可以用专用波特率计算器，计算波特率得到的值

UCA3MCTL = UCBRS\_3+UCBRF\_0; // UCBRSx=3, UCBRFx=0

UCA3CTL1 &= ~UCSWRST; // 状态机置位

}

int main( void )

{

// Stop watchdog timer to prevent time out reset

WDTCTL = WDTPW + WDTHOLD;

clock\_init(); //时钟初始化

P10DIR |=BIT6; //将P10.6配置成输出 因为电路有硬件看门狗所以需要喂狗

uart3\_init();

P1DIR |= 0x80; //1000 0000 P1.7输出模式

while(1)

{

P10OUT |= BIT6; // P10.6输出高电平 因为电路有硬件看门狗所以需要喂狗

delay\_ms(1);

P10OUT &= ~BIT6; // P10.6输出低电平 因为电路有硬件看门狗所以需要喂狗

delay\_ms(1);

UCA3TXBUF = 0xA1; //串口发送数据

while(!(UCA3IFG&UCTXIFG)); //等待发送完成

}

}

PC成功接收到串口3的数据

串口3中断接收数据

int main( void )

{

// Stop watchdog timer to prevent time out reset

WDTCTL = WDTPW + WDTHOLD;

clock\_init(); //时钟初始化

P10DIR |=BIT6; //将P10.6配置成输出 因为电路有硬件看门狗所以需要喂狗

P10SEL = BIT4 + BIT5; // 选择端口的第二功能，P10.4,P10.5 = TXD3 RXD3

UCA3CTL1 |= UCSWRST; // 状态机复位

UCA3CTL1 |= UCSSEL\_2; // 选择串口时钟源，UCSSEL\_1(CLK = ACLK) UCSSEL\_2(CLK = SMCLK)

UCA3BR0 = 0x82; // 两个寄存器配置串口的波特率，16M/9600=1666.66 (1666十六进制0x0682)

UCA3BR1 = 0x06; //也可以用专用波特率计算器，计算波特率得到的值

UCA3MCTL = UCBRS\_3+UCBRF\_0; // UCBRSx=3, UCBRFx=0

UCA3CTL1 &= ~UCSWRST; // 状态机置位

UCA3IE |= UCRXIE; // 使能 USCI\_A3 RX接收中断

\_EINT();

while(1)

{

P10OUT |= BIT6; // P10.6输出高电平 因为电路有硬件看门狗所以需要喂狗

delay\_ms(1);

P10OUT &= ~BIT6;// P10.6输出低电平 因为电路有硬件看门狗所以需要喂狗

delay\_ms(1);

}

#pragma vector=USCI\_A3\_VECTOR

\_\_interrupt void USCI\_A3\_ISR(void)

{

unsigned char rxData;

switch(\_\_even\_in\_range(UCA3IV,4))

{

case 0:break; // Vector 0 - no interrupt

case 2:

rxData = UCA3RXBUF; // Vector 2 - RXIFG 接收中断

UCA3TXBUF = rxData; //接受的数据转发

while(!(UCA3IFG&UCTXIFG));

break;

case 4:break; // Vector 4 - TXIFG 发送中断

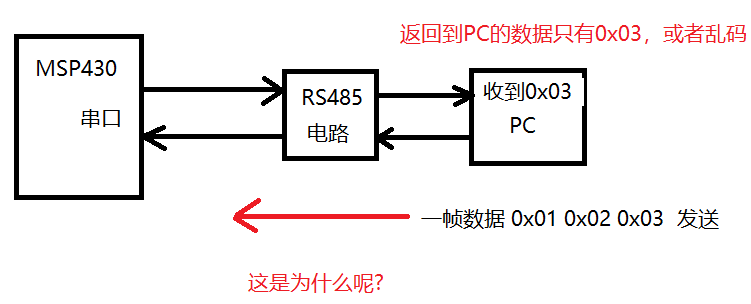
default: break;

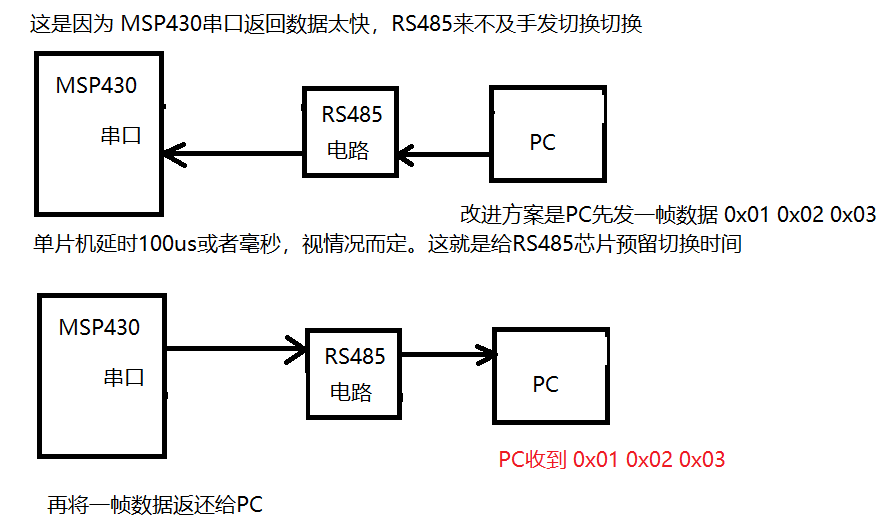
}

}

程序运行正常。但是只能单字节收发，多字节收发就出问题了。

起始软件本身没有问题，是外接RS485 半双工收发的问题。





unsigned char rxData[20];

unsigned int i = 0;;

#pragma vector=USCI\_A3\_VECTOR

\_\_interrupt void USCI\_A3\_ISR(void)

{

switch(\_\_even\_in\_range(UCA3IV,4))

{

case 0:break; // Vector 0 - no interrupt

case 2:

rxData[i] = UCA3RXBUF; // Vector 2 - RXIFG 接收中断

i++;

if(i == 10) //接收完一帧数据，比如10个字节，在转发

{

for(i = 0;i<10;i++)

{

UCA3TXBUF = rxData[i];

while(!(UCA3IFG&UCTXIFG));

}

i = 0;

}

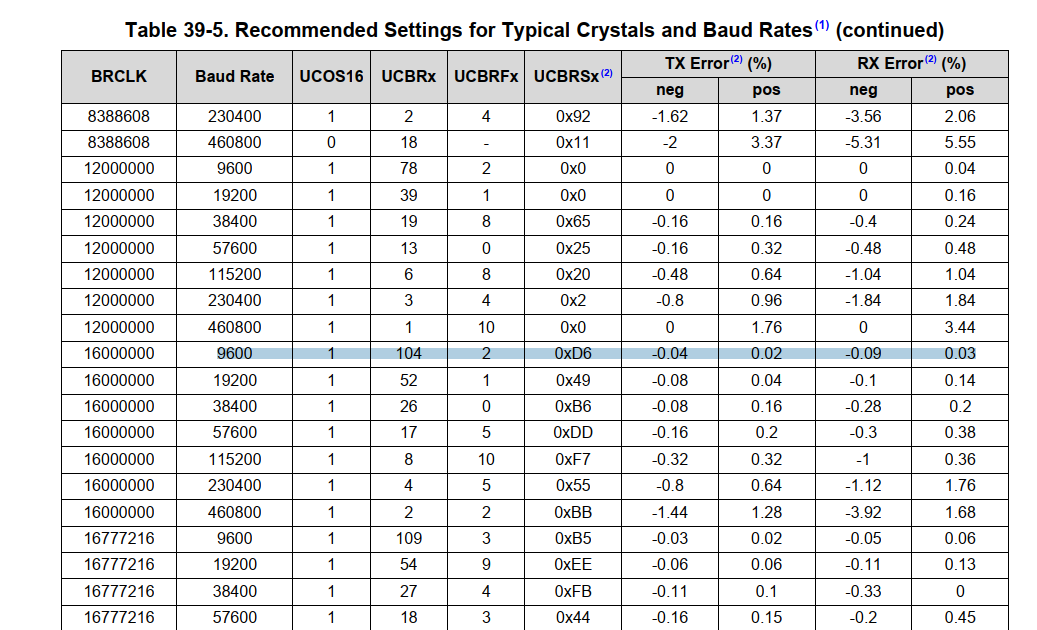
break;

case 4:break; // Vector 4 - TXIFG 发送中断

default: break;

}

} //这样串口转发就没有问题了



这是波特率误差校准表，可以参考优化串口。

**GPIO输入输出操作**

IO口输出模式，输出电平

/\*\*\*\*先将P2.7,P3.0,P3.1,P3.2,P3.3方向寄存器设置成输出模式，对应的位置1\*\*\*\*\*\*\*\*\*\*\*\*/

P2DIR |= BIT7; //P2.7输出模式

P3DIR |= BIT0; //P3.0输出模式

P3DIR |= BIT1; //P3.1输出模式

P3DIR |= BIT2; //P3.2输出模式

P3DIR |= BIT3; //P3.3输出模式

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*P2.7输出高低电平\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

P2OUT |= BIT7 //P2.7输出高电平

P2OUT &= ~BIT7 //P2.7输出低电平

其它IO以此类推

IO口输入操作



P7DIR &= ~BIT2; //P7.2默认为0 输入模式

P7DIR &= ~BIT3; //P7.3默认为0 输入模式

while(1)

{

if((P7IN&BIT2) == 0) //判断P7.2是否为低电平

{

/\*\*\*\*P7.2低电平 操作\*\*\*\*/

}

else

{

/\*\*\*\*P7.2高电平 操作\*\*\*\*/

}

}

P7.3操作方式一样

如果想实现按键双边沿中断操作

P7DIR &= ~BIT2; //P7.2默认为0 输入模式

P7IE |= BIT2; //开启P7.2外部中断

P7IES |= BIT2;//P7.2下降沿触发中断 (如果按键按下是低电平，那么初始化先设置下降沿)

P7IFG &= ~BIT2; //软件清除P7.2中断标志位

P7REN |= BIT2; //启动IO内部上下拉功能

#pragma vector = PORT1\_VECTOR

\_\_interrupt void P1\_IRQ(void)

{

if((P1IN & BIT6) == 0) //如果P1.6 == 0

{

printf("P1.6 IRQ falling...\n"); //按键按下进入中断，发现IO电平为0，执行下降沿程序，然后记得开上升沿模式，后面按键松开是上升沿状态

P1IES &= ~BIT6;//P1.6上升沿触发中断

}

else

{

printf("P1.6 IRQ rising...\n");//按键松开，上升沿,执行上升沿程序

P1IES |= BIT6;//第3次进入中断必须是按键再次按下，所以又设置回下降沿中断

}

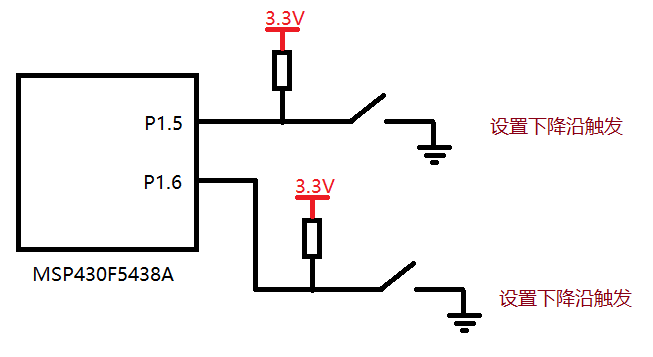
P1IFG &= ~BIT6; //清除中断标志位

}

**GPIO外部中断**

MSP430F5438A只有P1口和P2口这两组端口可以做外部中断。其余端口无法做外部中断。

其它一些型号可能还有其它中断口。



int main( void )

{

// Stop watchdog timer to prevent time out reset

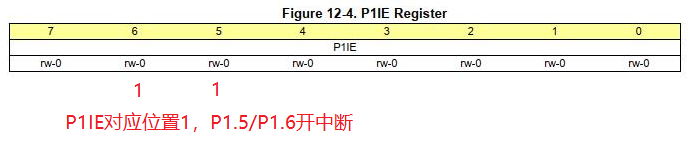
WDTCTL = WDTPW + WDTHOLD;

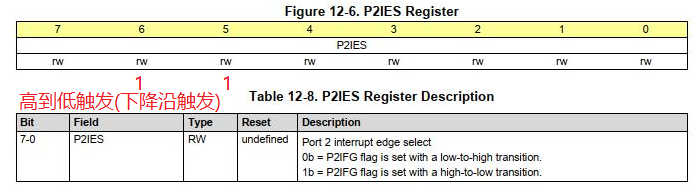
SysInit(); //时钟初始化，自定义

usartInit(); //串口初始化，自定义

P10DIR |=BIT6; //将P10.6配置成输出 因为电路有硬件看门狗所以需要喂狗

P1DIR &= ~BIT5; //P1.5默认为0 输入模式





P1DIR &= ~BIT6; //P1.6默认为0 输入模式

P1IE |= BIT5; //开启P1.5外部中断

P1IES |= BIT5;//P1.5下降沿触发中断

P1IFG &= ~BIT5; //软件清除P1.5中断标志位

P1REN |= BIT5; //启动IO内部上下拉功能

P1IE |= BIT6; //开启P1.6外部中断

P1IES |= BIT6;//P1.6下降沿触发中断

P1IFG &= ~BIT6; //软件清除P1.6中断标志位

P1REN |= BIT6; //启动IO内部上下拉功能

\_EINT(); //开总中断

printf("1111111...\n");

while(1)

{

P10OUT |= BIT6; // P10.6输出高电平 因为电路有硬件看门狗所以需要喂狗

delay\_ms(1);

P10OUT &= ~BIT6;// P10.6输出低电平 因为电路有硬件看门狗所以需要喂狗

delay\_ms(1);

}

}

#pragma vector = PORT1\_VECTOR //IO输入中断函数

\_\_interrupt void P1\_IRQ(void)

{

if(P1IFG & BIT5) //如果P1.5 == 1

{

printf("P1.5 IRQ...\n");

P1IFG &= ~BIT5; //一定要清除中断标志位，不然会频繁进中断

}

if(P1IFG & BIT6) //如果P1.6 == 1

{

printf("P1.6 IRQ...\n");

P1IFG &= ~BIT6; //一定要清除中断标志位，不然会频繁进中断

}

}

**MSP430F5438A定时器使用**

MSP430F5438有三个定时器 TA0 TA1和TB，定时器的功能略有区别

对于定时器TA1而言，有两个中断向量地址，其中比较匹配通道0具有单独的中断向量

MSP430中断向量的名称和TA0 TA1很难对应起来，需要通过中断向量地址来确认。

#define TIMER1\_A1\_VECTOR (48 \* 2u) /\* 0xFFE0 Timer1\_A3 CC1-2, TA1 \*/

#define TIMER1\_A0\_VECTOR (49 \* 2u) /\* 0xFFE2 Timer1\_A3 CC0 \*/

#define TIMER0\_A1\_VECTOR (53 \* 2u) /\* 0xFFEA Timer0\_A5 CC1-4, TA0 \*/

#define TIMER0\_A0\_VECTOR (54 \* 2u) /\* 0xFFEC Timer0\_A5 CC0 \*/

注意既然是做定时溢出中断，那么中断函数就必须用TIMER\_A1而不是TIMER\_A0，因为A0是给TA1做捕获比较0通道中断的

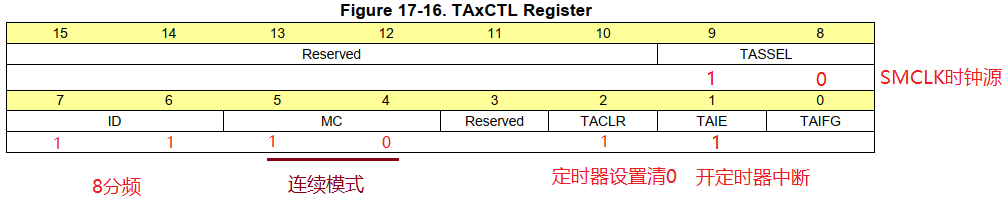
#define TIMER0\_B1\_VECTOR (59 \* 2u) /\* 0xFFF6 Timer0\_B7 CC1-6, TB \*/

#define TIMER0\_B0\_VECTOR (60 \* 2u) /\* 0xFFF8 Timer0\_B7 CC0 \*/

使用TA1做定时溢出中断



注意: 使用TA1定时器的CCR1,CCR2，还有定时器溢出中断，那么中断函数就对应#pragma vector = TIMER0\_A1\_VECTOR。如果使用CCR0就是定时器自定义定时时间，那么CCR0溢出后中断函数在vector = TIMER0\_A0\_VECTOR，而不是TIMER0\_A1\_VECTOR所以不要搞错了，一个定时器有两个中断函数。CCR0中断具有最高优先级。



16M/8分频 = 2M 一个时钟0.0000005(0.5us)，连续计数模式0.5us \* 65536 = 0.032(32ms)

证明32ms中断一次，那么1≈ 32ms \* 30 = 0.98秒

/\*定时器A初始化\*/

void TIMRA\_Init(void)

{

TA1CTL = TASSEL\_2 + TACLR + ID\_3 + MC\_2 + TAIE ; // SMCLK, (ID3)8分频，(MC2)连续模式, clear TAR

}

int Count = 0;

int main( void )

{

// Stop watchdog timer to prevent time out reset

WDTCTL = WDTPW + WDTHOLD;

SysInit();

P10DIR |=BIT6; //将P10.6配置成输出 因为电路有硬件看门狗所以需要喂狗

usartInit(); //串口初始化

TIMRA\_Init(); //定时器初始化

\_EINT(); //开总中断

while(1)

{

P10OUT |= BIT6; // P10.6输出高电平 因为电路有硬件看门狗所以需要喂狗

delay\_ms(1);

P10OUT &= ~BIT6;// P10.6输出低电平 因为电路有硬件看门狗所以需要喂狗

delay\_ms(1);

if(Count > 30) //32ms \* 30 = 0.98s

{

Count = 0;

printf("MSP430F5438A TEST ..\n");

}



}

}

#pragma vector=TIMER1\_A1\_VECTOR

\_\_interrupt void TIMER1\_A1\_ISR(void)

{

switch(\_\_even\_in\_range(TA1IV,14))

{

case 0: break; // No interrupt

case 2: break; // CCR1 not used

case 4: break; // CCR2 not used

case 6: break; // reserved

case 8: break; // reserved

case 10: break; // reserved

case 12: break; // reserved

case 14: Count++; // overflow 定时器65536溢出中断

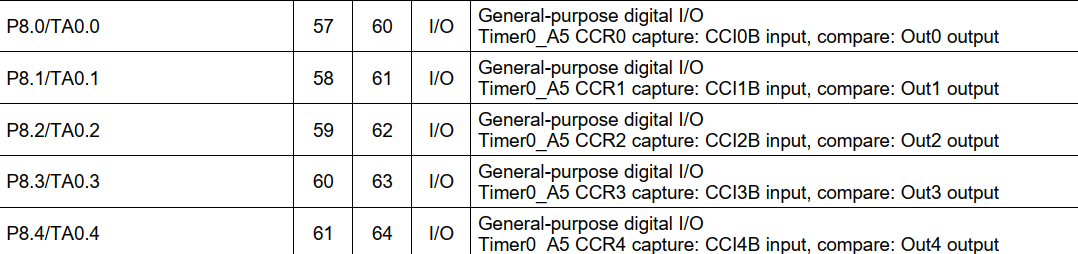
break;

default: break;

}

}

定时器输入捕获比较，频率周期采集

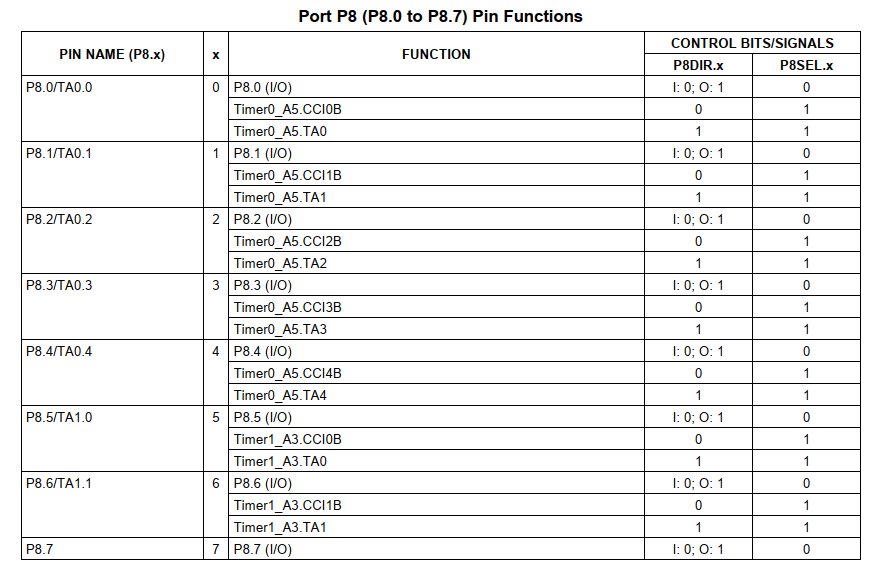


TA0.0 意思是TIMER0，也就是TA0定时器的意思。小数点后面的.0就是CCR0捕获比较通道

TA0.1 意思就是TA0定时器，CCR1捕获比较通道

TA0.2 意思就是TA0定时器，CCR2捕获比较通道

我们使用P8.1/TA0.1引脚，那么就是TA0定时器，CCR1通道进行捕获采集，PA8.1的CCR1通道要选择CCI1B这一路开关通道。



记住就算单独使用CCR输入捕获功能，也必须把相应的定时器打开

/\*定时器A初始化,P8.0输入捕获功能,CCI0B,CCR0\*/

void TIMRA\_Init(void)

{

P8DIR &= ~BIT1; //P8.1配置输入模式

P8SEL = BIT1; //P8.1配置成外设功能

TA0CTL = TASSEL\_2 + ID\_3 + MC\_2 + TACLR + TAIE ; // SMCLK, (ID3)8分频，(MC2)连续模式, clear TAR 开定时器溢出中断(TAIE)

TA0CCTL1|= CM0+SCS+CAP; //上升沿捕获，同步捕获，工作在捕获模式，TA0CCTL1就是定时器TA0,CCR1通道

TA0CCTL1|= CCIS\_1; //选择CCI0B输入，因为P8.1引脚在CCI0B位置的 TA0CCTL0就是定时器TA0,CCR0通道

TA0CCTL1|= CCIE; //允许捕获比较模块提出中断请求

TA0CCR1 = 0; //清0 CCR1

}

int main( void )

{

// Stop watchdog timer to prevent time out reset

WDTCTL = WDTPW + WDTHOLD;

SysInit();

P10DIR |=BIT6; //将P10.6配置成输出 因为电路有硬件看门狗所以需要喂狗

usartInit(); //串口初始化

TIMRA\_Init(); //定时器初始化

\_EINT(); //开总中断

while(1)

{

P10OUT |= BIT6; // P10.6输出高电平 因为电路有硬件看门狗所以需要喂狗

delay\_ms(1);

P10OUT &= ~BIT6;// P10.6输出低电平 因为电路有硬件看门狗所以需要喂狗

delay\_ms(1);

printf("while ..\n");

delay\_ms(300); // 软件延时

}

}

#pragma vector = TIMER0\_A1\_VECTOR //TIMER0中断， 也就是TA0, CCR1~CCR4,四个输入捕获通道的中断入口，如果用的CCR0通道需要查看宏，TIMER0\_A0\_VECTOR就是TIMER0 CCR0的通道入口

#pragma vector = TIMER0\_A1\_VECTOR

\_\_interrupt void capture\_CCR0\_IRQ(void)

{

switch(\_\_even\_in\_range(TA0IV,14))

{

case 0: break; // No interrupt

case 2: printf("TIMER0 CCR1 ..\n");break; // CCR1 通道，输入引脚信号触发中断

case 4: printf("TIMER0 CCR2 ..\n");break; // CCR2 not used

case 6: break; // reserved

case 8: break; // reserved

case 10: break; // reserved

case 12: break; // reserved

case 14: // overflow 定时器65536溢出中断,只要使用输入捕获，就会打开定时器，那么定时器溢出中断就必须有

break;

default: break;

}

}

用P8.1/TA0.1对输入信号频率进行采集

采集1Khz信号没问题

/\*定时器A初始化,P8.0输入捕获功能,CCI0B,CCR0\*/

void TIMRA\_Init(void)

{

P8DIR &= ~BIT1; //P8.1配置输入模式

P8SEL = BIT1; //P8.1配置成外设功能

TA0CTL = TASSEL\_2 + ID\_3 + MC\_2 + TACLR + TAIE ; // SMCLK = 16M,

//(ID3)8分频，16M/8=2M，定时器0.5us累加1

//(MC2)连续模式, clear TAR

//开定时器溢出中断(TAIE)

TA0CCTL1|= CM0+SCS+CAP; //上升沿捕获，同步捕获，工作在捕获模式

TA0CCTL1|= CCIS\_1; //CCI0B

TA0CCTL1|= CCIE; //允许捕获比较模块提出中断请求

TA0CCR1 = 0; //清0 CCR1

}

unsigned int old\_cap = 0; //上一次TACCR1的值

unsigned int period = 0; //计算出的周期

unsigned int TA\_ov\_num = 0; //定时器溢出一次累加1,也就是65535\*0.5us = 32768us就溢出了

int main( void )

{

// Stop watchdog timer to prevent time out reset

WDTCTL = WDTPW + WDTHOLD;

SysInit();

P10DIR |=BIT6; //将P10.6配置成输出 因为电路有硬件看门狗所以需要喂狗

usartInit(); //串口初始化

TIMRA\_Init(); //定时器初始化

\_EINT(); //开总中断

while(1)

{

P10OUT |= BIT6; // P10.6输出高电平 因为电路有硬件看门狗所以需要喂狗

delay\_ms(1);

P10OUT &= ~BIT6;// P10.6输出低电平 因为电路有硬件看门狗所以需要喂狗

delay\_ms(1);

printf("period = %d \n",period);

delay\_ms(300); // 软件延时

}

}

#pragma vector = TIMER0\_A1\_VECTOR

\_\_interrupt void capture\_CCR0\_IRQ(void)

{

switch(\_\_even\_in\_range(TA0IV,14))

{

case 0: break; // No interrupt

case 2: period = ((TA\_ov\_num\*65536) + TA0CCR1-old\_cap)\*0.5; // 乘以0.5是因为我们是2M的输入时钟，也就是0.5us计满TAR。所以最后算出来的结果要高于实际频率一半，乘以0.5就是/2的意思。修正回实际频率

old\_cap = TA0CCR1;

TA\_ov\_num = 0;

break; // CCR1 not used

case 4: printf("TIMER0 CCR2 ..\n");break; // CCR2 not used

case 6: break; // reserved

case 8: break; // reserved

case 10: break; // reserved

case 12: break; // reserved

case 14: TA\_ov\_num++; // overflow 定时器65536溢出中断

break;

default: break;

}

}

我发现，频率上了20Khz 误差就开始变大了，而且频率越高，数值越小，无法测量高频

下面对频率计算程序进行优化，这个问题是主循环printf消耗时间造成的。

char flag = 0; //第1次边沿捕获和第2次边沿捕获标记

unsigned long long capture1\_value = 0;

unsigned long long capture2\_value = 0;

unsigned long long capture\_diff = 0;

unsigned long long frequency = 0; //得到频率

#pragma vector = TIMER0\_A1\_VECTOR

\_\_interrupt void capture\_CCR0\_IRQ(void)

{

switch(\_\_even\_in\_range(TA0IV,14))

{

case 0: break; // No interrupt

case 2:

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*频率计算\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

if(flag == 0) //获取CCR1通道当前计数值

{

flag = 1;

capture1\_value = (TA\_ov\_num\*65535)+ TA0CCR1;

}

else if(flag == 1)//获取CCR1通道第2次计数值

{

flag = 0;

capture2\_value = (TA\_ov\_num\*65535)+ TA0CCR1;

if(capture2\_value > capture1\_value)

capture\_diff = (capture2\_value - capture1\_value);

else if(capture2\_value < capture1\_value)

capture\_diff = ((65535 - capture1\_value)+capture2\_value);

else

capture\_diff = 0;

frequency = (unsigned long long)(16000000/capture\_diff);//得到频率

}

TA\_ov\_num = 0; //清0本次累加的计数值

break; // CCR1 not used

case 4: printf("TIMER0 CCR2 ..\n");break; // CCR2 not used

case 6: break; // reserved

case 8: break; // reserved

case 10: break; // reserved

case 12: break; // reserved

case 14: TA\_ov\_num++; // overflow 定时器65536溢出中断

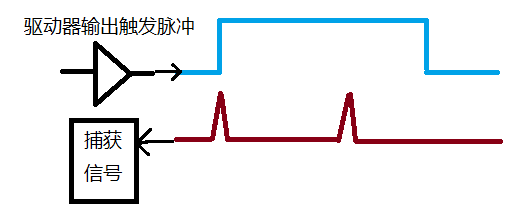
break;

default: break;

}

}

如果采集随机双脉冲之间的间隔，那么频率采集方式就不合适



main(){

TA0CTL = TASSEL\_2 + ID\_2 + MC\_2 + TACLR + TAIE ; // SMCLK = 16M,

TA0CCTL1|= CM1+SCS+CAP; //下降沿捕获，同步捕获，工作在捕获模式

TA0CCTL1|= CCIS\_1; //CCI0B

// TA0CCTL1|= CCIE; //允许捕获比较模块提出中断请求 屏蔽

TA0CCTL1 &= ~CCIE; //定时器初始化的时候一定要关闭捕获比较模块，像这种非周期型脉冲，必须捕获一次计算一次

TA0CCR1 = 0; //清0 CCR1

While(1){

gBDSechoTime.TimeOverflowNum = 0; //每次采集脉冲之前全局变量清0

gBDSechoTime.flag = 0; //每次采集脉冲之前进入中断的标志清0

P6OUT |= BIT5; //P6.5 = 1 驱动器输出高脉冲

TA0CTL = TASSEL\_2 + ID\_2 + MC\_2 + TACLR + TAIE ; //必须马上清0计数器的值，让计数器重新计数，一定要在捕获开启之前执行

TA0CCTL1|= CCIE; //开捕获比较模块

delay\_ms(1);

P6OUT &= ~BIT5; //P6.5 = 0 波导丝输出低

delay\_ms(10);

}

}

#pragma vector = TIMER0\_A1\_VECTOR

第1个脉冲触发第1次中断

\_\_interrupt void capture\_CCR0\_IRQ(void)

{



switch(\_\_even\_in\_range(TA0IV,14))

{

case 0: break; // No interrupt

case 2:

if(gBDSechoTime.flag == 0)

{

gBDSechoTime.flag = 1; //第1个脉冲让标志位置1

//TA0CTL = TASSEL\_2 + ID\_2 + MC\_2 + TACLR + TAIE ; //注意，第1个脉冲收到，进入中断后再清计数器误差大

gBDSechoTime.TimeOverflowNum = 0; //如果脉冲间隔超过65535 那么TimeOverflowNum+1

第2个脉冲触发第2次中断，发现标志位置1，采集第1个脉冲和第2个脉冲的间隔时间

}

else if(gBDSechoTime.flag == 1 )

{

delay\_us(700); //注意，在第2个脉冲之后，读取TA0CCR1寄存器的时候要延时，不是因为脉冲数据没采集到，而是TAR计数器的值复制给TA0CCR1需要时间。如果直接不延时瞬间读取，发现TA0CCR1的值还是第1个脉冲的值。(重点)

gBDSechoTime.TotalTime = (unsigned long)((gBDSechoTime.TimeOverflowNum\*655356)+TA0CCR1); //获取两个脉冲间隔时间

TA0CCTL1 &= ~CCIE; //关闭捕获比较模块

gBDSechoTime.flag = 0; //清0这一次双脉冲的采集标志

}

else

{

}

break;

case 4: printf("TIMER0 CCR2 ..\n");break; // CCR2 not used

case 6: break; // reserved

case 8: break; // reserved

case 10: break; // reserved

case 12: break; // reserved

case 14:

gBDSechoTime.TimeOverflowNum++; // overflow 定时器65536溢出中断,只要使用输入捕获，就会打开定时器，那么定时器溢出中断就必须有

break;

default: break;

}

}

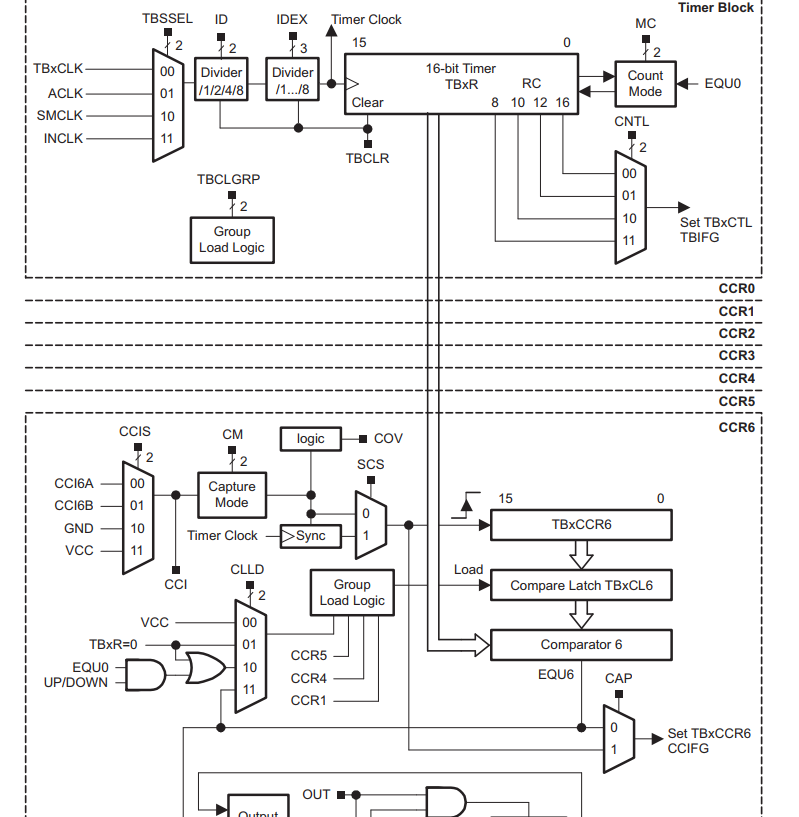
**定时器B使用**

TB的计数长度可以选择（8、10、12、16BITS），而TA只有16位；

TB0CCRn寄存器是双缓冲的，且可以分组。TA CCR无法分组

所有的TB输出可以被设为高阻状态

TB没有SCCI，即捕获器输入信号CCI没有被锁存



CCI没有锁存，直接输出

TA计数器只能计数到65535才能溢出。TB就不一样。计数器可以设置最大计数到16位(0xffff)，12位(0xfff)，10位(0x3ff)，8位(0xff)

TBCCR分组

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* TB0定时器初始化

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void TimerB0Init(void)

{

TBCTL = TBSSEL\_2 + ID\_3 + MC\_2 + CNTL\_0 + TBCLR + TBIE;

//SMCLK时钟

//ID\_3 8分频

//MC\_2 连续计数模式

//CNTL\_0 : 计数器长度16位，最大65565溢出中断

//TBCLR :清0计数器

//TBIE: 开启定时器B中断

}

#pragma vector=TIMERB1\_VECTOR

\_\_interrupt void TIMERB1\_ISR(void)

{

/\* Any access, read or write, of the TBIV register automatically resets the

highest "pending" interrupt flag. \*/

switch( \_\_even\_in\_range(TBIV,14) )

{

case 0: break; // No interrupt

case 2: break; // CCR1 not used

case 4: break; // CCR2 not used

case 6: break; // CCR3 not used

case 8: break; // CCR4 not used

case 10: break; // CCR5 not used

case 12: break; // CCR6 not used

case 14: printf("timeB++ \n"); // overflow 根据计算TIMB 32ms中断一次

break;

default: break;

}

}

定时器B指定计数器计数次数定时

中断向量 #pragma vector=TIMERB1\_VECTOR 是TB0定时器溢出中断和输入捕获比较中断

而#pragma vector=TIMERB0\_VECTOR 是CCR0，比较定时中断入口

所以要搞清楚，TB0定时器也是有两个中断入口的

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* TB0定时器初始化

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void TimerB0Init(void)

{

TBCCTL0 = CCIE; // CCR0 interrupt enabled

TBCCR0 = 55000; //连续计数模式，计数到55000产生TIMERB0中断

TBCTL = TBSSEL\_2 + MC\_2 + TBCLR; // SMCLK, contmode, clear TBR

//SMCLK时钟

//MC\_2 连续计数模式

//TBCLR :清0计数器

}

#pragma vector=TIMERB0\_VECTOR

\_\_interrupt void TIMERB0\_ISR (void)

{

/\*TB0计数器计数55000时间到,执行该中断\*/

TBCCR0 += 55000; //重新向CCR0赋值，重新进行定时计数

}

**ADC12 模数转换器使用**

采用P6.1/A1管脚进行测试

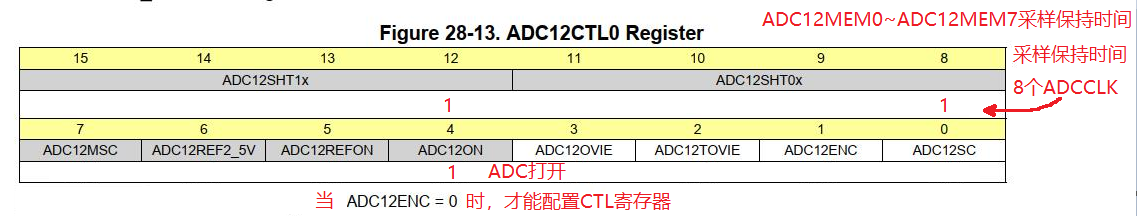
注意MSP430F54xx系列的ADC 没有内部REF基准源模块，需要外接基准源。

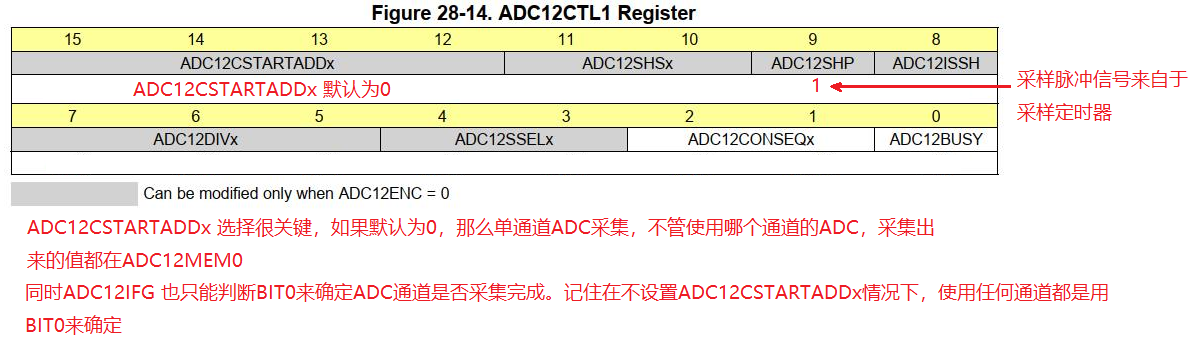
MSP430F5438A和MSP4305438是有区别的

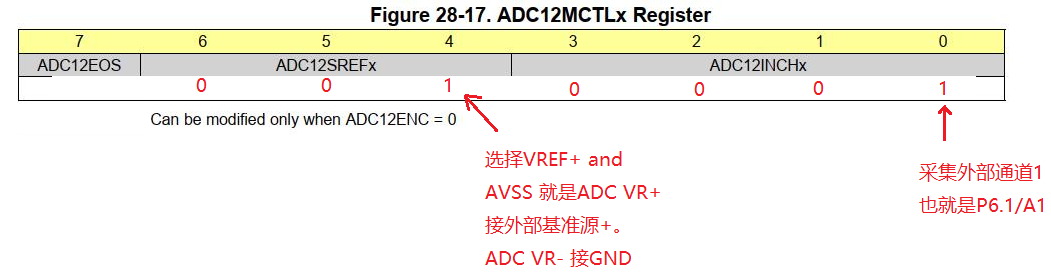
MSP430F5438没有REF模块, MSP430F5438A带有内部REF模块

MSP430F5438A在没有CPU主程序介入下ADC外设可以同时转换16路模拟信号。

ADC单次转换实验，每次转换后都用while轮询获取







注意ADC12MCTLx 这个x是有通道选择的，下面代码就是因为这个MCTLx没选对，导致出问题

void ADC\_A1\_Init(void)

{

P6SEL|=BIT1; //选择ADC通道1,P6.1/A1功能模式

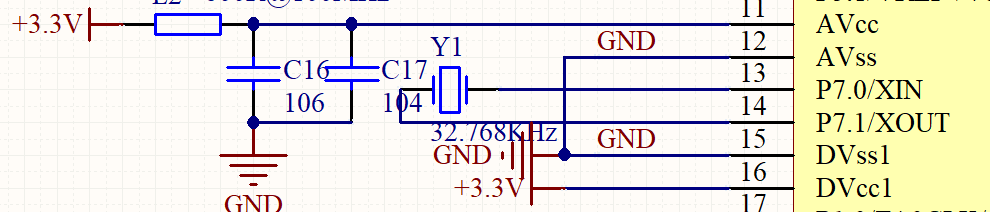
// 只有在ADC12ENC复位的情况下才可以操作

ADC12CTL0 &= ~ADC12ENC;

// 设置采样保持时间，最大时间周期以提高转换精度

// 注意MSP430F5438没有REF模块，片内基准无效

// 操作ADC12REF2\_5V ，ADC12REFON并无意义



ADC12CTL0 = ADC12SHT0\_15 + ADC12SHT1\_15 + ADC12ON;

/\* 如果是5438A内部自带REF模块，操作方式

ADC12CTL0 = ADC12SHT0\_15 + ADC12SHT1\_15 + ADC12ON +

ADC12REF2\_5V + ADC12REFON;\*/

// 采样保持脉冲来自采样定时器

ADC12CTL1 = ADC12SHP;

// 关闭内部内部温度检测以降低功耗，注意或操作否则修改转换精度

ADC12CTL2 |= ADC12TCOFF ;

// 基准电压选择AVCC，并选择1通道——(AVCC-AVSS)/2

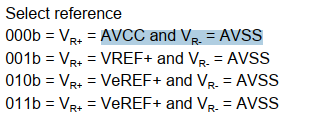
ADC12MCTL0 = ADC12SREF\_0 + ADC12INCH\_1;

\_\_delay\_cycles(75);

// ADC12使能

ADC12CTL0 |= ADC12ENC;

}



int main( void )

{

int i = 0;

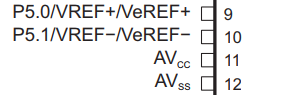
// Stop watchdog timer to prevent time out reset

WDTCTL = WDTPW + WDTHOLD;

SysInit();

usartInit(); //串口初始化

ADC\_A1\_Init();



如果选择外接的基准源，那就是VeREF



那么ADC12MCTL0 = ADC12SREF\_2+ADC12INCH\_1

\_EINT(); //开总中断

printf("MSP430F5438A TEST ..\n");

while(1)

{

ADC12CTL0 |= ADC12SC; // 启动转换

while ( !(ADC12IFG & BIT0) ); // ADC12IFG.x 理论上应该是不同的AD采集通道，

//对应不同的IFG位，但是因为 ADC12CSTARTADDx 初始化配置位0，

//那么不管任何通道采集是否完成标志位都只有用IFG0来判断

printf("ADC Value = %d\n",ADC12MEM0); //和IFG情况一样理论上应该是不同的AD采集通道，对应不同的ADC12MEM

//但是因为 ADC12CSTARTADDx 初始化配置为0，

//所以所有的AD通道采集的值都放在ADC12MEM0，其实这只适合单通道转换场景

}

}

ADC12CTL1寄存器增加ADC12CSTARTADD参数

void ADC\_A1\_Init(void)

{

P6SEL|=BIT1; //选择ADC通道1,P6.1/A1功能模式

// 只有在ADC12ENC复位的情况下才可以操作

// ADC12SHT1X ADC12SHT0X ADC12MSC ADC12REF2\_5V ADC12REFON ADC12ON

ADC12CTL0 &= ~ADC12ENC;

// 设置采样保持时间，最大时间周期以提高转换精度

// 注意MSP430F5438没有REF模块，片内基准无效

// 操作ADC12REF2\_5V ，ADC12REFON并无意义

ADC12CTL0 = ADC12SHT0\_15 + ADC12SHT1\_15 + ADC12ON;

/\* 如果是5438A内部自带REF模块，操作方式

ADC12CTL0 = ADC12SHT0\_15 + ADC12SHT1\_15 + ADC12ON +

ADC12REF2\_5V + ADC12REFON;\*/

// 采样保持脉冲来自采样定时器，增加ADC转换起始地址1ADC12CSTARTADD\_1;

ADC12CTL1 = ADC12SHP | ADC12CSTARTADD\_1;

// 关闭内部内部温度检测以降低功耗，注意或操作否则修改转换精度

ADC12CTL2 |= ADC12TCOFF ;

// 基准电压选择AVCC，并选择1通道——(AVCC-AVSS)/2

ADC12MCTL0 = ADC12SREF\_0 + ADC12INCH\_1;

\_\_delay\_cycles(75);

// ADC12使能

ADC12CTL0 |= ADC12ENC;

}

int main( void )

{

int i = 0;

// Stop watchdog timer to prevent time out reset

WDTCTL = WDTPW + WDTHOLD;

SysInit();

usartInit(); //串口初始化

ADC\_A1\_Init();

\_EINT(); //开总中断

printf("MSP430F5438A TEST ..\n");

while(1)

{

ADC12CTL0 |= ADC12SC; // 启动转换

while ( !(ADC12IFG & BIT1) ); //这次改为IFG1标志位判断，通道1标志位，

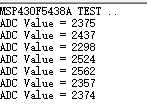
//因为在CTL1设置了ADC12CSTARTADD

printf("ADC Value = %d\n",ADC12MEM1); //这次改为ADC12MEM1读取，读取ADC通道1的数据

//因为在CTL1设置了ADC12CSTARTADD

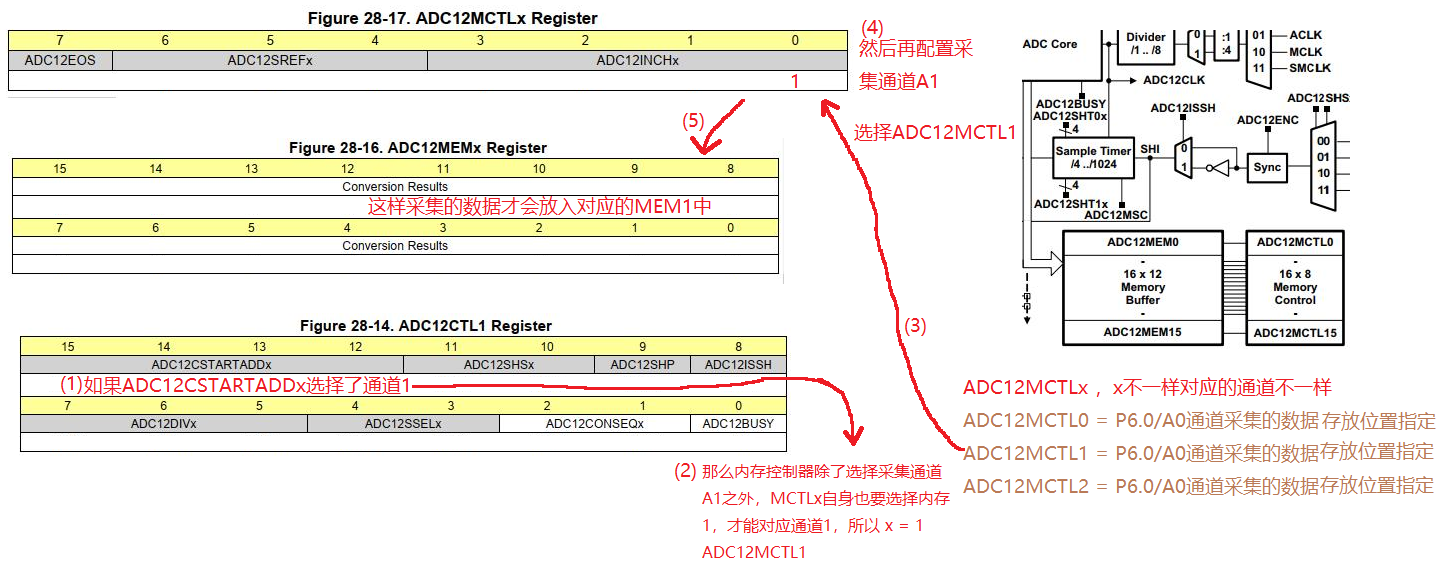
}

}

 测试发现程序不会卡在ADC12IFG BIT1 通道1的位置，但是读出来的A1通道数据没对。这是为什么呢?

这其实就是因为ADC12MCTLx的x通道没选对。

下面对ADC12MCTLx ，ADC12CTL1，ADC12MEMx这三个寄存器关系做下描述。



void ADC\_A1\_Init(void)

{

P6SEL|=BIT1; //选择ADC通道1,P6.1/A1功能模式

// 只有在ADC12ENC复位的情况下才可以操作

ADC12CTL0 &= ~ADC12ENC;

ADC12CTL0 = ADC12SHT0\_15 + ADC12SHT1\_15 + ADC12ON;

// 采样保持脉冲来自采样定时器，增加ADC转换起始地址1ADC12CSTARTADD\_1;

ADC12CTL1 = ADC12SHP | ADC12CSTARTADD\_1;

// 关闭内部内部温度检测以降低功耗，注意或操作否则修改转换精度

ADC12CTL2 |= ADC12TCOFF ;

// 基准电压选择AVCC，并选择1通道——(AVCC-AVSS)/2

ADC12MCTL1 = ADC12SREF\_0 + ADC12INCH\_1; //MCTLx改为1，选择内存控制器1接收A1采集的数据，这样IFG也就是判断BIT1, 数据存放在ADC12MEM1

\_\_delay\_cycles(75);

// ADC12使能

ADC12CTL0 |= ADC12ENC;

}

int main( void )

{

int i = 0;

WDTCTL = WDTPW + WDTHOLD;

SysInit();

usartInit(); //串口初始化

ADC\_A1\_Init();

\_EINT(); //开总中断

printf("MSP430F5438A TEST ..\n");

while(1)

{

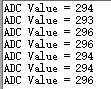
ADC12CTL0 |= ADC12SC; // 启动转换

while ( !(ADC12IFG & BIT1) ); //MCTLx设置的内存1，所以这次改为IFG1标志位判断，通道1标志位，成功

printf("ADC Value = %d\n",ADC12MEM1); // MCTLx设置的内存1，这次改为ADC12MEM1读 ADC通道1的数据成功

}

}

A1通道模拟值读取成功。这就是轮询读取ADC数据的方法

中断方式单通道读取ADC数据

ADC中断主要就是要开启ADC12CTL1里面的ADC12CSTARTADD参数

ADC12IE中断寄存器要打开

void ADC\_A1\_Init(void)

{

P6SEL|=BIT1; //选择ADC通道1,P6.1/A1功能模式

// 只有在ADC12ENC复位的情况下才可以操作

ADC12CTL0 &= ~ADC12ENC;

ADC12CTL0 = ADC12SHT0\_15 + ADC12SHT1\_15 + ADC12ON;

// 采样保持脉冲来自采样定时器

ADC12CTL1 = ADC12SHP | ADC12CSTARTADD\_1; //不同的ADC通道中断，必须要设定ADC12CSTARTADDx参数对应的通道，不然的话中断函数无法启动

// 关闭内部内部温度检测以降低功耗，注意或操作否则修改转换精度

ADC12CTL2 |= ADC12TCOFF ;

// 基准电压选择AVCC，并选择1通道——(AVCC-AVSS)/2

ADC12MCTL1 = ADC12SREF\_0 + ADC12INCH\_1; //因为采集的是A1通道，所以MCTLx还是要选择1，如果选择0，那么ADC中断中用ADC12MEM1得到的数据还是在内存0里面

ADC12IE |= 0x02; //打开ADC 通道1中断，A1引脚

\_\_delay\_cycles(75);

// ADC12使能

ADC12CTL0 |= ADC12ENC;

}

/\*

\* ADC中断服务函数

\*/

#pragma vector = ADC12\_VECTOR

\_\_interrupt void ADC12\_ISR(void)

{

switch(\_\_even\_in\_range(ADC12IV,34))

{

case 0: break; // Vector 0: No interrupt

case 2: break; // Vector 2: ADC overflow

case 4: break; // Vector 4: ADC timing overflow

case 6: break; // Vector 6: ADC12IFG0

case 8: printf("ADC12 A1 Value = %d \n",ADC12MEM1);break; // Vector 8: ADC12IFG1

case 10: break; // Vector 10: ADC12IFG2

case 12: break; // Vector 12: ADC12IFG3

case 14: break; // Vector 14: ADC12IFG4

case 16: break; // Vector 16: ADC12IFG5

case 18: break; // Vector 18: ADC12IFG6

case 20: break; // Vector 20: ADC12IFG7

case 22: break; // Vector 22: ADC12IFG8

case 24: break; // Vector 24: ADC12IFG9

case 26: break; // Vector 26: ADC12IFG10

case 28: break; // Vector 28: ADC12IFG11

case 30: break; // Vector 30: ADC12IFG12

case 32: break; // Vector 32: ADC12IFG13

case 34: break; // Vector 34: ADC12IFG14

default: break;

}

} //不同的IFGx对应不同的ADC通道中断，这里case 8 IFG1 就是A1通道采集数据中断

int main( void )

{

int i = 0;

WDTCTL = WDTPW + WDTHOLD;

SysInit();

usartInit(); //串口初始化

ADC\_A1\_Init();

\_EINT(); //开总中断

printf("MSP430F5438A TEST ..\n");

while(1)

{

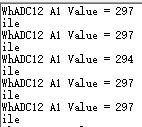
ADC12CTL0 |= ADC12SC;// 启动转换，启的转换一次，ADC中断一次，如果中断后不重复启的ADC转换，那么ADC中断是不会再发生的，因为设置的是ADC单次转换

printf("While\n");

}

}

再单次转换中不需要关注清除ADC中断标志位操作

 ADC中断采集数据正确，因为printf在中断消耗时间的原因，打印起来看起很乱。

ADC中断的数据如果放在定时器中断等待采集就会出BUG，注意

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* ADC中断服务函数

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#pragma vector = ADC12\_VECTOR

\_\_interrupt void ADC12\_ISR(void)

{

switch(\_\_even\_in\_range(ADC12IV,34))

{

case 0: break; // Vector 0: No interrupt

case 2: break; // Vector 2: ADC overflow

case 4: break; // Vector 4: ADC timing overflow

case 6: break; // Vector 6: ADC12IFG0

case 8: ADCvalue = ADC12MEM1;

ADCsampleStaus = 1;

break; // Vector 8: ADC12IFG1

case 10: break; // Vector 10: ADC12IFG2

case 12: break; // Vector 12: ADC12IFG3

case 14: break; // Vector 14: ADC12IFG4

………………….

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 启动ADC采集

\* 返回ADC采集的值

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

unsigned int StartADCgetData(void) //该程序放在定时器任务中断里面处理，出现问题

{

ADC12CTL0 |= ADC12SC;

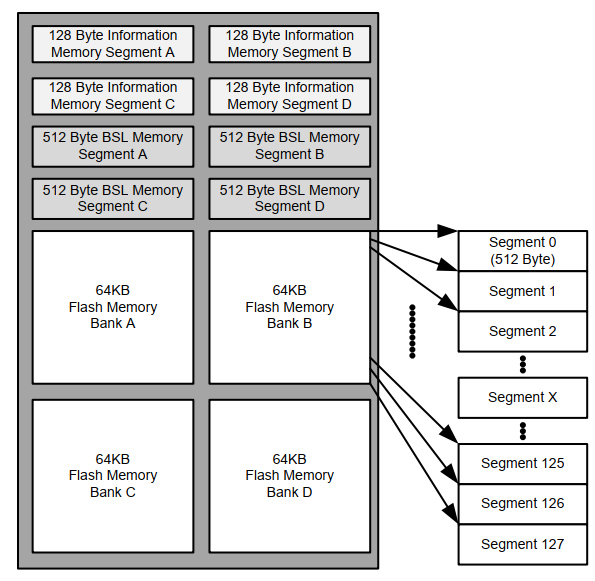
while(ADCsampleStaus != 1); //问题就是while循环等待ADC中断太久

ADCsampleStaus = 0;

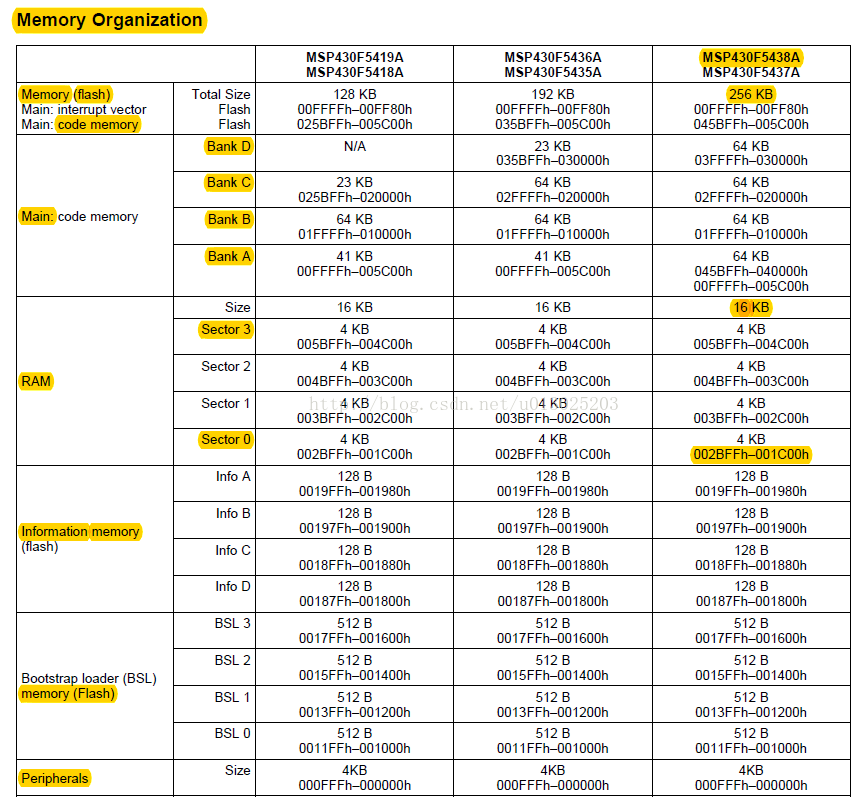
return ADCvalue;

}

**MSP430F5438A内部flash做eeprom存储**



用来做eeprom的flash内存区域只能是information memory区域。而且information memory有4个区域，每个区域只能存储128字节的数据，一共512字节



MSP430F5438A就只有这块区域的地址可以做eeprom

因为A B C D端都是挨着连续的，最低的地址端是D，如果你要存放大于128 B字节的数据，那么就取地址D段(0x1800h)，开始向上累加。

例如: char \* Flash\_ptr;

Flash\_ptr = (char \*) 0x1800;

\*Flash\_ptr++ = 0xA1;

这样操作

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 向flash information区写数据，使用0x1880地址来测试

\* 1880地址在C段

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void write\_SegC()

{

char \* Flash\_ptr; // Initialize Flash pointer

Flash\_ptr = (char \*) 0x1880; //存储起始地址

\_\_disable\_interrupt(); // 5xx Workaround: 关闭全局中断，如果开了看门狗，也必须关闭看狗

// interrupt while erasing. Re-Enable // GIE if needed

FCTL3 = FWKEY; // Clear Lock bit

FCTL1 = FWKEY+ERASE; // 擦除information 区域,flash写之前需要擦除，因为flash只能写0，擦除后全部为1

\*Flash\_ptr = 0; // Dummy write to erase Flash seg 这句初始化指针数据为0必须要写，不然会出现存储BUG

FCTL1 = FWKEY+WRT; // Set WRT bit for write operation

\*Flash\_ptr++ = 0xA1; // Write value to flash

\*Flash\_ptr++ = 0xA2;

\*Flash\_ptr++ = 0xA3;

\*Flash\_ptr++ = 0xA4;

\*Flash\_ptr++ = 0xA5;

\*Flash\_ptr++ = 0xA6;

FCTL1 = FWKEY; // Clear WRT bit

FCTL3 = FWKEY+LOCK; // Set LOCK bit

\_\_enable\_interrupt(); //再次打开中断

}

/\*

\* 读取information区，C段1880地址上的数据

\*/

void Read\_Flash(void)

{

char \*pFlash=(char\*)0x1880; //读取0x1880地址，看看是否数据成功写入

uint8\_t data[6] = 0;

uint8\_t i = 0;

data[0] = \*pFlash++;

data[1] = \*pFlash++;

data[2] = \*pFlash++;

data[3] = \*pFlash++;

data[4] = \*pFlash++;

data[5] = \*pFlash++;

printf("data = %x %x %x %x %x %x\n",data[0],data[1],data[2],data[3],data[4],data[5]);

}

int main( void )

{

int i = 0;

char \*xpFlash = (char \*)0x1880;

// Stop watchdog timer to prevent time out reset

WDTCTL = WDTPW + WDTHOLD;

SysInit();

P10DIR |=BIT6; //将P10.6配置成输出 因为电路有硬件看门狗所以需要喂狗

usartInit(); //串口初始化

\_EINT(); //开总中断

/\*P1.7 LED IO 配置\*/

// P1DIR |= 0x80; //1000 0000 P1.7输出模式

printf("MSP430F5438A TEST ..\n");

write\_SegC(); //向information 区域的flash 写数据

delay\_ms(10);

printf(" write\_SegC..\n");

while(1)

{

for(i = 0; i < 2000; i++)

{

P10OUT |= BIT6; // P10.6输出高电平 因为电路有硬件看门狗所以需要喂狗

delay\_ms(10);

P10OUT &= ~BIT6;// P10.6输出低电平 因为电路有硬件看门狗所以需要喂狗

Read\_Flash();

delay\_ms(10);

}

// 软件延时

}

}

int main( void )

{

int i = 0;

char \*xpFlash = (char \*)0x1880;

// Stop watchdog timer to prevent time out reset

WDTCTL = WDTPW + WDTHOLD;

SysInit();

P10DIR |=BIT6; //将P10.6配置成输出 因为电路有硬件看门狗所以需要喂狗

usartInit(); //串口初始化

\_EINT(); //开总中断

printf("MSP430F5438A TEST ..\n");

write\_SegC();

delay\_ms(10);

while(1)

{

for(i = 0; i < 2000; i++)

{

P10OUT |= BIT6; // P10.6输出高电平 因为电路有硬件看门狗所以需要喂狗

delay\_ms(10);

P10OUT &= ~BIT6;// P10.6输出低电平 因为电路有硬件看门狗所以需要喂狗

Read\_Flash();

delay\_ms(10);

}

// 软件延时

}

}

测试发现，数据写入和读出没有问题，但是从新下载程序后，如果不写入flash 0x1880段数据，那么再次读取0x1880段数据就全部是FF，这是为什么呢? 其实这是因为我们重新下载程序后，软件自动将flash所有区域都擦除了。所有我用来保存数据的0x1880段也被下载软件擦除了。(MSP430F149UserGuide)我已经给出了在下载程序通过设置IAR，来保存EEPROM的方法。下面是默认下载程序会擦除information区的暂时解决方案。

在实际应用中，程序一直在运行，不会去用下载软件擦除flash，如果每次板子复位，就会再去写一次flash

write\_SegC(); //向information 区域的flash 写数据

delay\_ms(10);

printf(" write\_SegC..\n");

这是因为该程序在初始化阶段。这就导致了不管存不存储，都会去写flash，这样就不对了，不是按照我程序要求得时间去写flash。

int main( void )

{

int i = 0;

char \*xpFlash = (char \*)0x1880;

// Stop watchdog timer to prevent time out reset

WDTCTL = WDTPW + WDTHOLD;

SysInit();

P10DIR |=BIT6; //将P10.6配置成输出 因为电路有硬件看门狗所以需要喂狗

usartInit(); //串口初始化

\_EINT(); //开总中断

printf("MSP430F5438A TEST ..\n");

if(\*xpFlash != 0xA1) //加入判断语句，可以确认已经写过一次flash，不需要再写。当然这是测试flash存储功能才用的方法。

{

write\_SegC();

delay\_ms(10);

printf(" write\_SegC..\n");

}

while(1)

{

for(i = 0; i < 2000; i++)

{

P10OUT |= BIT6; // P10.6输出高电平 因为电路有硬件看门狗所以需要喂狗

delay\_ms(10);

P10OUT &= ~BIT6;// P10.6输出低电平 因为电路有硬件看门狗所以需要喂狗

Read\_Flash();

delay\_ms(10);

}

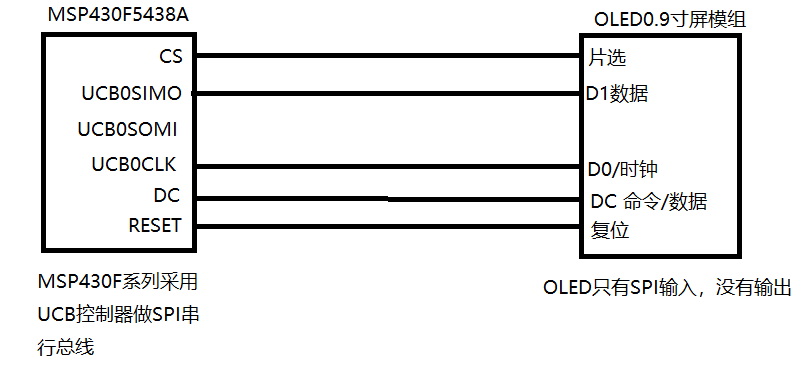
// 软件延时

}

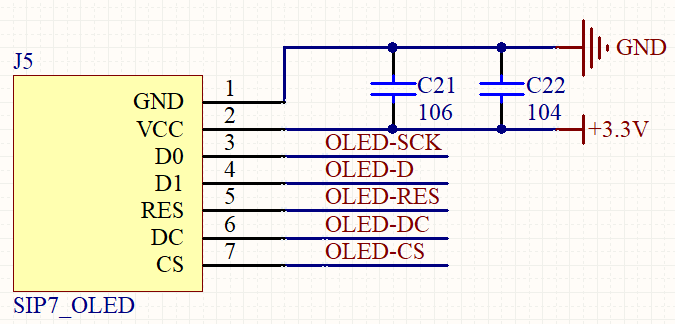
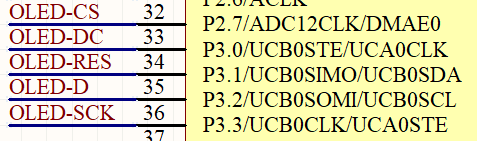
}

程序执行成功。正常情况下是需要写flash的时候才去执行write\_SegC();，平时不会执行。

**模拟SPI操作OLED屏**



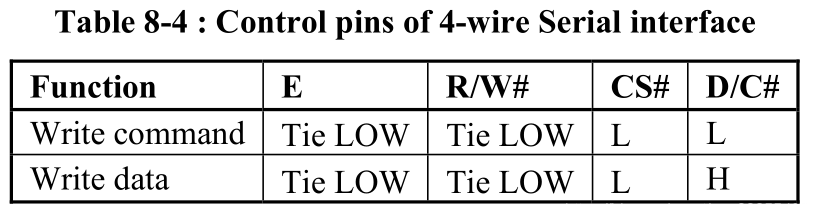
因为我是模拟SPI，所以IO口接法不是按照标准UCBSPI控制器接的，有点不同。

这是实际模拟SPI的接法，随意找了几个IO口来接。

SSD1306LCD控制器4线SPI接口包括：SCLK(时钟D0)，SDIN(数据D1)，D/C(命令数据控制)，片选(CS)。

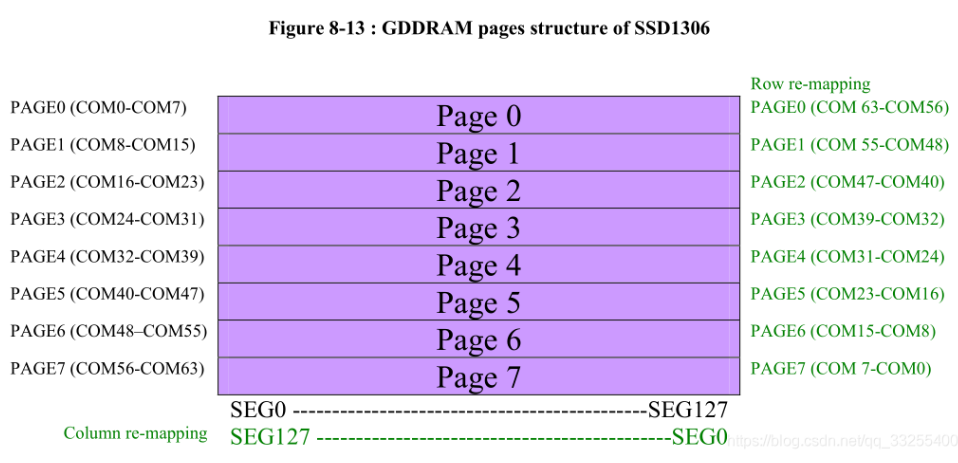
D0充当SCLK，D1充当SDIN。对于未使用的数据引脚，D2应保持打开状态。从D3到D7、E和R/W（WR）可连接到外部接地。



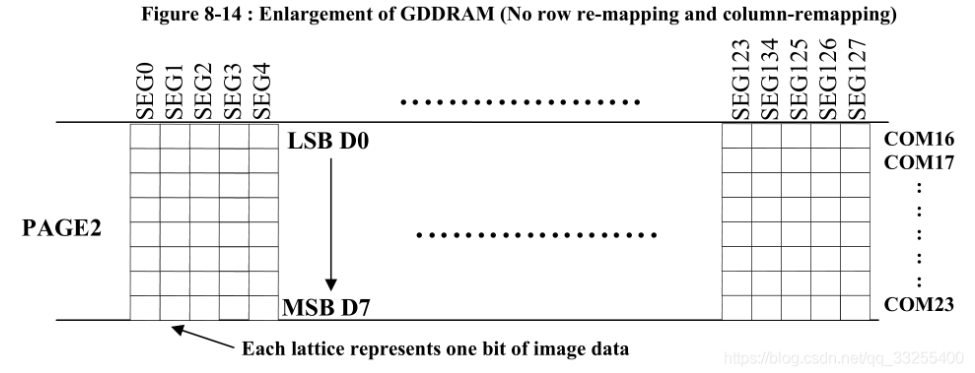
向OLED写数据时，CS=0,DC=1

向OLED写命令时，CS=0,DC=0

GDDRAM是一个位映射的静态RAM，保存要显示的位模式。RAM的大小是128 x 64位，RAM分为8页，从第0页到第7页，用于单色128x64点阵显示器

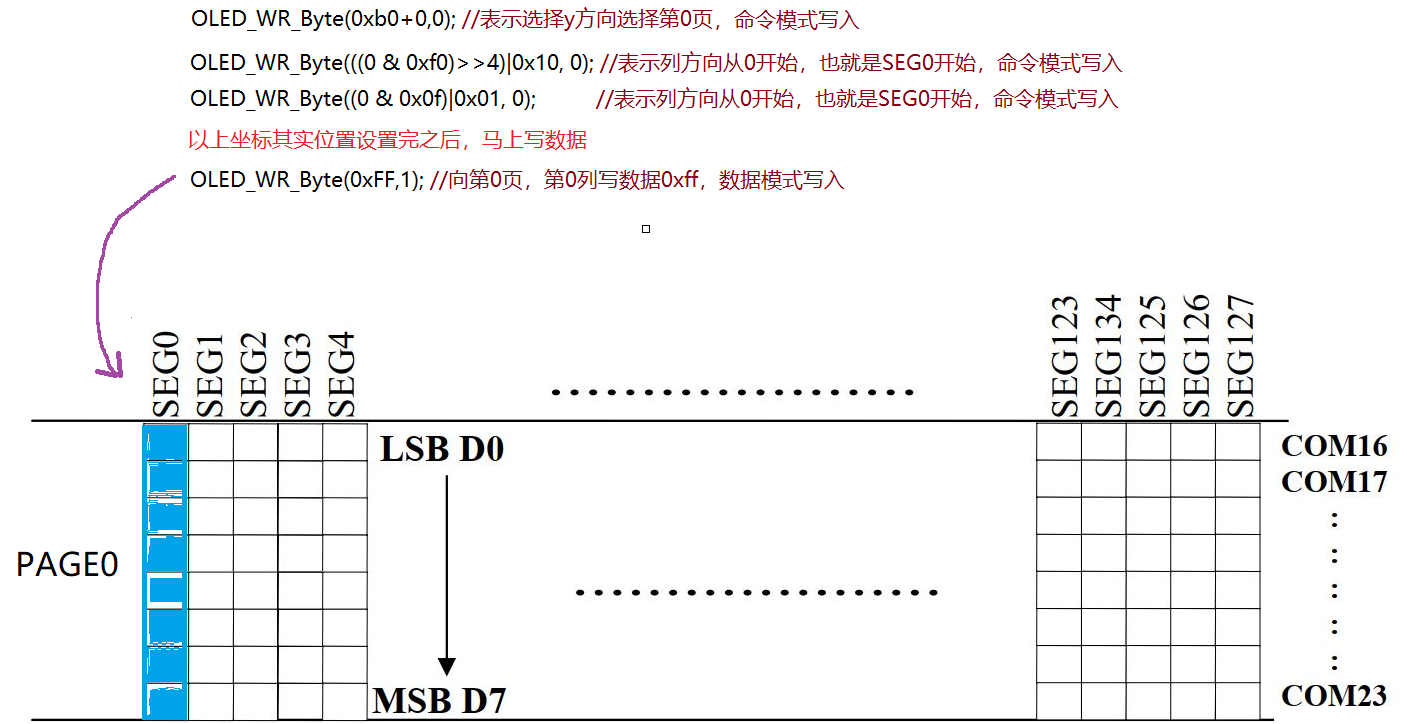


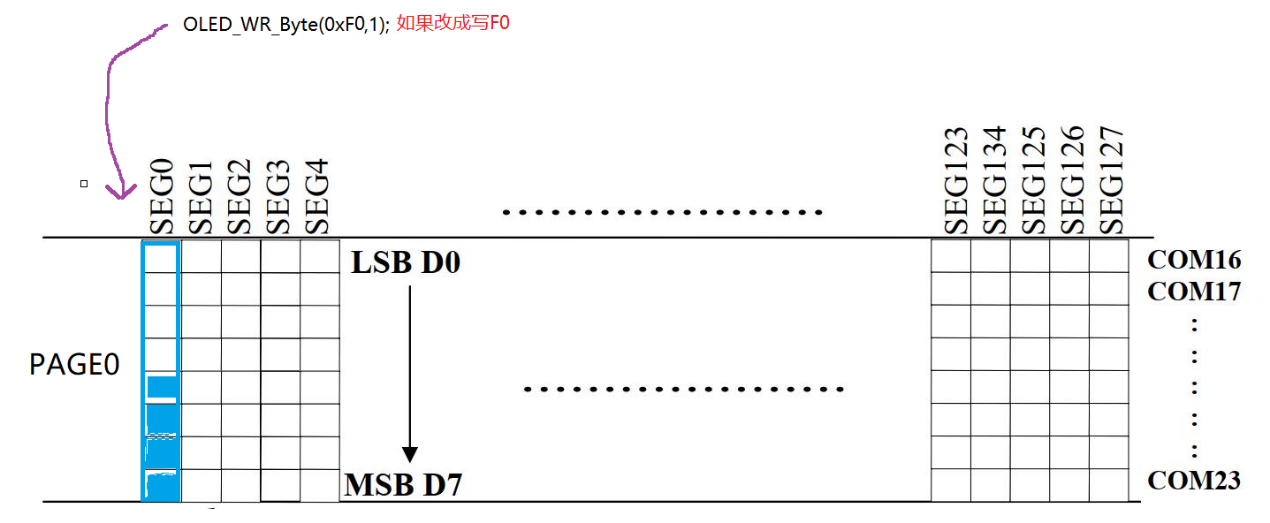
OLED屏垂直方向就是页，一共有8页，1页有0~7行，所以就是1页8位\*8页 = 64bit点阵



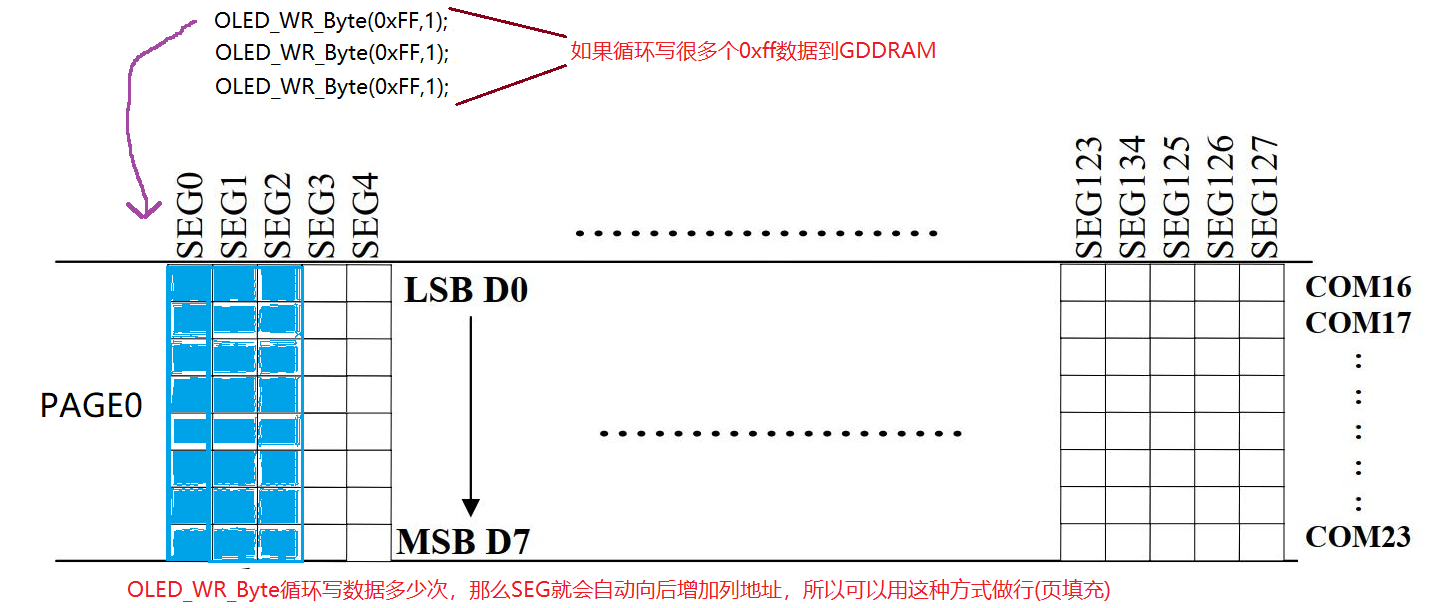
这就是1页里面的数据写入形式，1次写8位，从LSB D0 写到D7

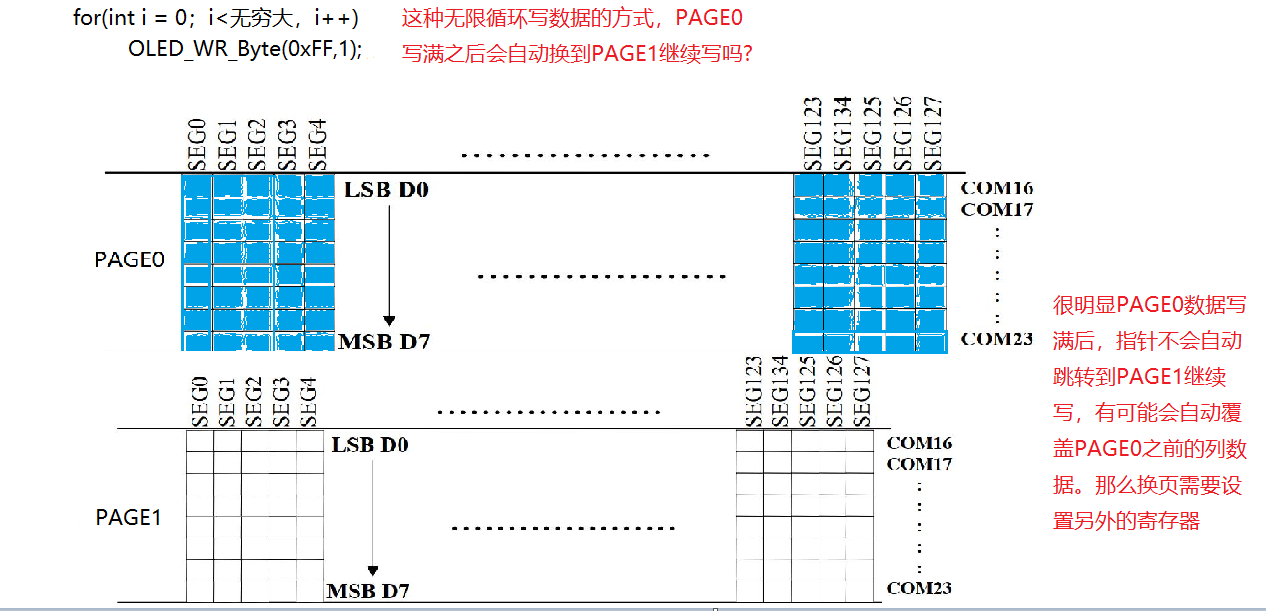
1. 通过命令B0h到B7h选择显示页面的起始地址





就只有高4位点亮





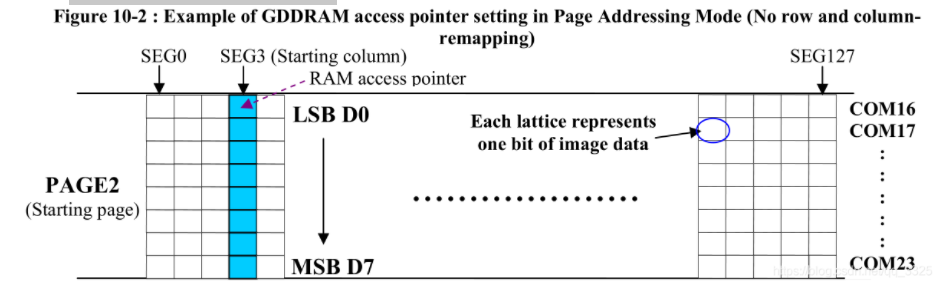
2. 通过命令00h~0Fh设置指针的下起始列地址

3. 通过10h~1Fh命令设置指针的上起始列地址

例如，如果页地址设置为B2h，则下列地址为03h，上列地址为00h

这意味着起始列是第2页的SEG3。RAM访问指针位于中所示的位置

输入数据字节将写入第3列的RAM位置



数据并不是我们以前写LCD那样按照坐标指定像素点位置来写，而是写1个字节就会占用一列(垂直点亮)，该字节1表示亮，0表示灭，比如SEG3就是收到0xff数据

MSP430模拟SPI OLED驱动移植

#define OLED\_CS\_H() (P2OUT |= BIT7) //引脚定义

#define OLED\_CS\_L() (P2OUT &= ~BIT7)

#define OLED\_DC\_H() (P3OUT |= BIT0)

#define OLED\_DC\_L() (P3OUT &= ~BIT0)

#define OLED\_CLK\_H() (P3OUT |= BIT3)

#define OLED\_CLK\_L() (P3OUT &= ~BIT3)

#define OLED\_D1\_H() (P3OUT |= BIT2)

#define OLED\_D1\_L() (P3OUT &= ~BIT2)

#define OLED\_RST\_H() (P3OUT |= BIT1)

#define OLED\_RST\_L() (P3OUT &= ~BIT1)

#define OLED\_CMD 0 //写命令

#define OLED\_DATA 1 //写数据

//向SSD1306写入一个字节。

//dat:要写入的数据/命令

//cmd:数据/命令标志 0,表示命令;1,表示数据;

void OLED\_WR\_Byte(unsigned char dat,unsigned char cmd)

{

unsigned char i;

if(cmd)

OLED\_DC\_H(); //高电平=数据

else

OLED\_DC\_L(); //低电平=命令

OLED\_CS\_L(); //CS = 0

for(i=0;i<8;i++)

{

OLED\_CLK\_L(); //CLK=0

if(dat&0x80)

OLED\_D1\_H(); //DAT=1

else

OLED\_D1\_L(); //DAT=0

OLED\_CLK\_H(); //CLK=1

dat<<=1;

}

OLED\_CS\_H(); //CS=1

OLED\_DC\_H(); //DC=1

}

//清屏函数,清完屏,整个屏幕是黑色的!和没点亮一样!!!

void OLED\_Clear(void)

{

unsigned char i,n;

for(i=0;i<8;i++)

{

OLED\_WR\_Byte (0xb0+i,OLED\_CMD); //设置页地址（0~7）

OLED\_WR\_Byte (0x00,OLED\_CMD); //设置显示位置—列低地址

OLED\_WR\_Byte (0x10,OLED\_CMD); //设置显示位置—列高地址

for(n=0;n<128;n++)OLED\_WR\_Byte(0,OLED\_DATA);

} //更新显示

}

//设置坐标点

void OLED\_Set\_Pos(unsigned char x, unsigned char y)

{

OLED\_WR\_Byte(0xb0+y,OLED\_CMD);

OLED\_WR\_Byte(((x&0xf0)>>4)|0x10,OLED\_CMD);

OLED\_WR\_Byte((x&0x0f)|0x01,OLED\_CMD);

}

主要就是实现OLED\_WR\_Byte, OLED\_Clear, OLED\_Set\_Pos这三个函数。

//初始化SSD1306

void OLED\_Init(void)

{

OLED\_RST\_H();

delay\_ms(1);

OLED\_RST\_L();

delay\_ms(10);

OLED\_RST\_H();

OLED\_WR\_Byte(0xAE,OLED\_CMD);//--turn off oled panel

OLED\_WR\_Byte(0x00,OLED\_CMD);//---set low column address

OLED\_WR\_Byte(0x10,OLED\_CMD);//---set high column address

OLED\_WR\_Byte(0x40,OLED\_CMD);//--set start line address Set Mapping RAM Display Start Line (0x00~0x3F)

OLED\_WR\_Byte(0x81,OLED\_CMD);//--set contrast control register

OLED\_WR\_Byte(0xCF,OLED\_CMD); // Set SEG Output Current Brightness

OLED\_WR\_Byte(0xA1,OLED\_CMD);//--Set SEG/Column Mapping 0xa0左右反置 0xa1正常

OLED\_WR\_Byte(0xC8,OLED\_CMD);//Set COM/Row Scan Direction 0xc0上下反置 0xc8正常

OLED\_WR\_Byte(0xA6,OLED\_CMD);//--set normal display

OLED\_WR\_Byte(0xA8,OLED\_CMD);//--set multiplex ratio(1 to 64)

OLED\_WR\_Byte(0x3f,OLED\_CMD);//--1/64 duty

OLED\_WR\_Byte(0xD3,OLED\_CMD);//-set display offset Shift Mapping RAM Counter (0x00~0x3F)

OLED\_WR\_Byte(0x00,OLED\_CMD);//-not offset

OLED\_WR\_Byte(0xd5,OLED\_CMD);//--set display clock divide ratio/oscillator frequency

OLED\_WR\_Byte(0x80,OLED\_CMD);//--set divide ratio, Set Clock as 100 Frames/Sec

OLED\_WR\_Byte(0xD9,OLED\_CMD);//--set pre-charge period

OLED\_WR\_Byte(0xF1,OLED\_CMD);//Set Pre-Charge as 15 Clocks & Discharge as 1 Clock

OLED\_WR\_Byte(0xDA,OLED\_CMD);//--set com pins hardware configuration

OLED\_WR\_Byte(0x12,OLED\_CMD);

OLED\_WR\_Byte(0xDB,OLED\_CMD);//--set vcomh

OLED\_WR\_Byte(0x40,OLED\_CMD);//Set VCOM Deselect Level

OLED\_WR\_Byte(0x20,OLED\_CMD);//-Set Page Addressing Mode (0x00/0x01/0x02)

OLED\_WR\_Byte(0x02,OLED\_CMD);//

OLED\_WR\_Byte(0x8D,OLED\_CMD);//--set Charge Pump enable/disable

OLED\_WR\_Byte(0x14,OLED\_CMD);//--set(0x10) disable

OLED\_WR\_Byte(0xA4,OLED\_CMD);// Disable Entire Display On (0xa4/0xa5)

OLED\_WR\_Byte(0xA6,OLED\_CMD);// Disable Inverse Display On (0xa6/a7)

OLED\_WR\_Byte(0xAF,OLED\_CMD);//--turn on oled panel

OLED\_WR\_Byte(0xAF,OLED\_CMD); /\*display ON\*/

OLED\_Clear();

OLED\_Set\_Pos(0,0); //起始坐标设置为x=0，y=0(PAGE0)

}

/\*可以使用OLED\_on给屏幕全部写1来测试驱动是否移植成功，移植成功的话，屏幕会显示很多白色的杠杠\*/

void OLED\_On(void)

{

unsigned char i,n;

for(i=0;i<8;i++)

{

OLED\_WR\_Byte (0xb0+i,OLED\_CMD);

OLED\_WR\_Byte (0x00,OLED\_CMD);

OLED\_WR\_Byte (0x10,OLED\_CMD);

for(n=0;n<128;n++)OLED\_WR\_Byte(1,OLED\_DATA);

}

}

下面移植字符显示功能

/\*OLED

\* x:是SEG其实地址，也就是x坐标

\* y:是页地址，一般是0~7，也就是y坐标，只是这里的y坐标比较大

\* char:写入显示的字符ASCII码

\*/

void OLED\_ShowChar(unsigned char x,unsigned char y,unsigned char chr)

{

unsigned char c=0,i=0;

c=chr-' '; //得到偏移后的值

if(x>Max\_Column-1){x=0;y=y+2;}

if (SIZE == 16)

{

OLED\_Set\_Pos(x,y);

for(i=0;i<8;i++)

{

OLED\_WR\_Byte(oled\_asc2\_1608[c][i],OLED\_DATA);

}

OLED\_Set\_Pos(x,y+1);

for(i=8;i<16;i++)

{

OLED\_WR\_Byte(oled\_asc2\_1608[c][i],OLED\_DATA);

}

}

else {

OLED\_Set\_Pos(x, y + 1);

for (i = 0; i<6; i++)

OLED\_WR\_Byte(F6x8[c][i], OLED\_DATA);

}

}

const unsigned char oled\_asc2\_1608[60][16]={

{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},/\*" ",0\*/

{0x00,0x00,0x00,0x00,0x00,0x00,0x1F,0xCC,0x00,0x0C,0x00,0x00,0x00,0x00,0x00,0x00},/\*"!",1\*/

{0x00,0x00,0x08,0x00,0x30,0x00,0x60,0x00,0x08,0x00,0x30,0x00,0x60,0x00,0x00,0x00},/\*""",2\*/

{0x02,0x20,0x03,0xFC,0x1E,0x20,0x02,0x20,0x03,0xFC,0x1E,0x20,0x02,0x20,0x00,0x00},/\*"#",3\*/

{0x00,0x00,0x0E,0x18,0x11,0x04,0x3F,0xFF,0x10,0x84,0x0C,0x78,0x00,0x00,0x00,0x00},/\*"$",4\*/

{0x0F,0x00,0x10,0x84,0x0F,0x38,0x00,0xC0,0x07,0x78,0x18,0x84,0x00,0x78,0x00,0x00},/\*"%",5\*/………………………………………………该字库去网上找库这里就不用写完了………………../

unsigned char const F6x8[][6] =

{

0x00, 0x00, 0x00, 0x00, 0x00, 0x00,// sp

0x00, 0x00, 0x00, 0x2f, 0x00, 0x00,// !

0x00, 0x00, 0x07, 0x00, 0x07, 0x00,// "

0x00, 0x14, 0x7f, 0x14, 0x7f, 0x14,// #

0x00, 0x24, 0x2a, 0x7f, 0x2a, 0x12,// $

0x00, 0x62, 0x64, 0x08, 0x13, 0x23,// %

0x00, 0x36, 0x49, 0x55, 0x22, 0x50,// &

0x00, 0x00, 0x05, 0x03, 0x00, 0x00,// '

0x00, 0x00, 0x1c, 0x22, 0x41, 0x00,// (

0x00, 0x00, 0x41, 0x22, 0x1c, 0x00,// )

0x00, 0x14, 0x08, 0x3E, 0x08, 0x14,// \*

0x00, 0x08, 0x08, 0x3E, 0x08, 0x08,// +

0x00, 0x00, 0x00, 0xA0, 0x60, 0x00,// ,

0x00, 0x08, 0x08, 0x08, 0x08, 0x08,// -

0x00, 0x00, 0x60, 0x60, 0x00, 0x00,// .

0x00, 0x20, 0x10, 0x08, 0x04, 0x02,// /……………………..小尺寸字符库，去网上找

字符显示使用方式OLED\_ShowChar(100,0,'8'); //在SEG100位置，PAGE0页开始显示

/\*

\* 字符串显示

\* x起始SEG，y起始PAGE页，\*char写入字符串

\*/

void OLED\_ShowString(unsigned char x, unsigned char y, unsigned char \*chr)

{

unsigned char j = 0;

while (chr[j] != '\0')

{

OLED\_ShowChar(x, y, chr[j]);

x += 8;

if (x>120){ x = 0; y += 2; }

j++;

}

}

/\*

\* 16进制数字显示范围0~f

\* 可以用来做数字拼接显示

\*/

void OLED\_4\_8\_number(unsigned char page,unsigned char lie,unsigned char num)

{

unsigned char i;

OLED\_WR\_Byte(0xb0+page,OLED\_CMD); //选择页PAGE0

OLED\_WR\_Byte((lie>>4)|0x10,OLED\_CMD); //选择SEG0~127

OLED\_WR\_Byte(lie&0x0f,OLED\_CMD); //选择SEG0~127

for(i=0;i<4;i++)

OLED\_WR\_Byte(number[num\*4+i],OLED\_DATA);

}

const unsigned char number[]={

0x7C,0x82,0x82,0x7C,0x00,0x84,0xFE,0x80,0xC4,0xA2,0x92,0x8C,

0x44,0x92,0x92,0x6C,0x30,0xAC,0xFE,0xA0,0x4E,0x8A,0x8A,0x72,

0x7C,0x92,0x92,0x64,0x06,0xF2,0x0A,0x06,0x6C,0x92,0x92,0x6C,

0x4C,0x92,0x92,0x7C,0xFC,0x12,0x12,0xFC,0xFE,0x92,0x92,0x6C,

0x7C,0x82,0x82,0x44,0xFE,0x82,0x82,0x7C,0xFE,0x92,0x92,0x92,

0xFE,0x12,0x12,0x12

}; //number字库

for(char x = 0; x<16;x++){

OLED\_4\_8\_number(10,0,x); //这就是数字使用方式

delay\_ms(300);

}//0~f循环显示

**MSP430 标准modbus协议移植(支持poll软件)**

尝试在MSP430移植freemodbus协议栈，发现freemodbus在16位单片机上运行不正常，无法与poll进行连接，现在移植自己设计的modbus库。

modbus.h (中间件)

#ifndef \_modbus\_h

#define \_modbus\_h

#include "modbus\_crc.h"

#include "modbus\_uart.h"

#include "modbus\_485.h"

#include "modbus\_time.h"

#include "msp430x54x.h"

#include <stdio.h>

typedef struct

{

unsigned char myadd; //本设备从机地址

unsigned char rcbuf[100]; //modbus接受缓冲区

unsigned char timout; //modbus数据持续时间

unsigned char recount; //modbus端口接收到的数据个数

unsigned char timrun; //modbus定时器是否计时标志

unsigned char reflag; //modbus一帧数据接受完成标志位

unsigned char sendbuf[100]; //modbus接发送缓冲区

}MODBUS;

extern MODBUS modbus;

extern unsigned int Reg[];

void Modbus\_Init(void);

void Modbus\_Func3(void);

void Modbus\_Event(void);

#endif

modbus.c(中间件)

#include "modbus.h"

#include "msp430x54x.h"

#include <stdio.h>

MODBUS modbus;

unsigned int Reg[] ={ 0x0001,

0x0002,

0x0003,

0x0004,

0x0005,

0x0006,

0x0008,

}; //自己定义的MODBUS寄存器，上位机POLL就是操作该寄存器值。

// Modbus初始化函数

void Modbus\_Init()

{

modbus.myadd = 0x01; //从机设备地址为1

modbus.timrun = 0; //modbus定时器停止计算

//Modbus\_485\_Init(); //我是RS232 不需要初始化RS485 IO口

Modbus\_Uart\_Init();

Modbus\_Time\_Init();

}

// Modbus 03号功能码函数

// Modbus 主机读取寄存器值

void Modbus\_Func3()

{

unsigned int Regadd,Reglen,crc;

unsigned char i,j;

//得到要读取寄存器的首地址

Regadd = modbus.rcbuf[2]\*256+modbus.rcbuf[3];

//得到要读取寄存器的数据长度

Reglen = modbus.rcbuf[4]\*256+modbus.rcbuf[5];

//发送回应数据包

i = 0;

modbus.sendbuf[i++] = modbus.myadd; //发送本机设备地址

modbus.sendbuf[i++] = 0x03; //发送功能码

modbus.sendbuf[i++] = ((Reglen\*2)%256); //返回字节个数

for(j=0;j<Reglen;j++) //返回数据

{

modbus.sendbuf[i++] = Reg[Regadd+j]/256;

modbus.sendbuf[i++] = Reg[Regadd+j]%256;

}

crc = Modbus\_CRC16(modbus.sendbuf,i); //计算要返回数据的CRC

modbus.sendbuf[i++] = crc/256;

modbus.sendbuf[i++] = crc%256;

// 开始返回Modbus数据

//Modbus\_485\_TX\_Mode; //我这里是RS232不需要切换收发状态

for(j=0;j<i;j++)

{

Modbus\_Send\_Byte(modbus.sendbuf[j]);

}

//Modbus\_485\_RX\_Mode; //我这里是RS232不需要切换收发状态

}

// Modbus 06号功能码函数

// Modbus 主机写入寄存器值

void Modbus\_Func6()

{

unsigned int Regadd;

unsigned int val;

unsigned int i,crc,j;

i=0;

Regadd=modbus.rcbuf[2]\*256+modbus.rcbuf[3]; //得到要修改的地址

val=modbus.rcbuf[4]\*256+modbus.rcbuf[5]; //修改后的值

Reg[Regadd]=val; //修改本设备相应的寄存器

//以下为回应主机

modbus.sendbuf[i++]=modbus.myadd;//本设备地址

modbus.sendbuf[i++]=0x06; //功能码

modbus.sendbuf[i++]=Regadd/256;

modbus.sendbuf[i++]=Regadd%256;

modbus.sendbuf[i++]=val/256;

modbus.sendbuf[i++]=val%256;

crc=Modbus\_CRC16(modbus.sendbuf,i);

modbus.sendbuf[i++]=crc/256;

modbus.sendbuf[i++]=crc%256;

//Modbus\_485\_TX\_Mode; //我这里是RS232不需要切换收发状态

for(j=0;j<i;j++)

{

Modbus\_Send\_Byte(modbus.sendbuf[j]);

}

//Modbus\_485\_RX\_Mode; //我这里是RS232不需要切换收发状态

}

// Modbus事件处理函数

void Modbus\_Event()

{

unsigned int crc,rccrc;

//没有收到数据包

if(modbus.reflag == 0)

{

return;

}

/\* printf(" = %x %x %x %x %x %x %x %x \n", //测试是否接收成功

modbus.rcbuf[0],modbus.rcbuf[1],modbus.rcbuf[2],modbus.rcbuf[3],

modbus.rcbuf[4],modbus.rcbuf[5],modbus.rcbuf[6],modbus.rcbuf[7]);\*/

//有可能在RS485总线上，因用户主机的代码bug，会出现接收到1字节数据，这1字节数据，进入CRC校验程序会导致BUG，因为modbus.recount如果为1，那么1-2绝对是负数，导致CRC16程序里面死循环，无限循环，从而导致rcbuf溢出。

if(modbus.recount > 7) //如果收到的数据只有1字节，就不处理,必须是标准MODBUS 大于7字节才处理，这段if是容错设计

{

//收到数据包

//通过读到的数据帧计算CRC

crc = Modbus\_CRC16(&modbus.rcbuf[0],modbus.recount-2);

// 读取数据帧的CRC

rccrc = modbus.rcbuf[modbus.recount-2]\*256+modbus.rcbuf[modbus.recount-1];

if(crc == rccrc) //CRC检验成功 开始分析包

{

if(modbus.rcbuf[0] == modbus.myadd) // 检查地址是否时自己的地址

{

switch(modbus.rcbuf[1]) //分析modbus功能码

{

case 0:break;

case 1:break;

case 2:break;

case 3:

Modbus\_Func3(); //03功能码

break;

case 4:break;

case 5:break;

case 6:

Modbus\_Func6(); //06功能码

break;

case 7:break;

case 8:break;

case 9:break;

}

}

else if(modbus.rcbuf[0] == 0) //广播地址不予回应

{

}

}

}

//可以加入else 打印提示，收到数据错误

modbus.recount = 0; //不管MODBUS是否收到1字节，做容错处理，都必须将recount清0

modbus.reflag = 0; //不管MODBUS是否收到1字节，做容错处理，都必须将recount清0

}

modbus\_crc.c

#include "modbus\_crc.h"

#include "msp430x54x.h"

#include <stdio.h>

/\* CRC 高位字节值表 \*/

const unsigned char auchCRCHi[] = {

0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0,

0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,

0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0,

0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40,

0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1,

0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41,

0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1,

0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,

0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0,

0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40,

0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1,

0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40,

0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0,

0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40,

0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0,

0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40,

0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0,

0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,

0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0,

0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,

0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0,

0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40,

0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1,

0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,

0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0,

0x80, 0x41, 0x00, 0xC1, 0x81, 0x40

} ;

/\* CRC低位字节值表\*/

const unsigned char auchCRCLo[] = {

0x00, 0xC0, 0xC1, 0x01, 0xC3, 0x03, 0x02, 0xC2, 0xC6, 0x06,

0x07, 0xC7, 0x05, 0xC5, 0xC4, 0x04, 0xCC, 0x0C, 0x0D, 0xCD,

0x0F, 0xCF, 0xCE, 0x0E, 0x0A, 0xCA, 0xCB, 0x0B, 0xC9, 0x09,

0x08, 0xC8, 0xD8, 0x18, 0x19, 0xD9, 0x1B, 0xDB, 0xDA, 0x1A,

0x1E, 0xDE, 0xDF, 0x1F, 0xDD, 0x1D, 0x1C, 0xDC, 0x14, 0xD4,

0xD5, 0x15, 0xD7, 0x17, 0x16, 0xD6, 0xD2, 0x12, 0x13, 0xD3,

0x11, 0xD1, 0xD0, 0x10, 0xF0, 0x30, 0x31, 0xF1, 0x33, 0xF3,

0xF2, 0x32, 0x36, 0xF6, 0xF7, 0x37, 0xF5, 0x35, 0x34, 0xF4,

0x3C, 0xFC, 0xFD, 0x3D, 0xFF, 0x3F, 0x3E, 0xFE, 0xFA, 0x3A,

0x3B, 0xFB, 0x39, 0xF9, 0xF8, 0x38, 0x28, 0xE8, 0xE9, 0x29,

0xEB, 0x2B, 0x2A, 0xEA, 0xEE, 0x2E, 0x2F, 0xEF, 0x2D, 0xED,

0xEC, 0x2C, 0xE4, 0x24, 0x25, 0xE5, 0x27, 0xE7, 0xE6, 0x26,

0x22, 0xE2, 0xE3, 0x23, 0xE1, 0x21, 0x20, 0xE0, 0xA0, 0x60,

0x61, 0xA1, 0x63, 0xA3, 0xA2, 0x62, 0x66, 0xA6, 0xA7, 0x67,

0xA5, 0x65, 0x64, 0xA4, 0x6C, 0xAC, 0xAD, 0x6D, 0xAF, 0x6F,

0x6E, 0xAE, 0xAA, 0x6A, 0x6B, 0xAB, 0x69, 0xA9, 0xA8, 0x68,

0x78, 0xB8, 0xB9, 0x79, 0xBB, 0x7B, 0x7A, 0xBA, 0xBE, 0x7E,

0x7F, 0xBF, 0x7D, 0xBD, 0xBC, 0x7C, 0xB4, 0x74, 0x75, 0xB5,

0x77, 0xB7, 0xB6, 0x76, 0x72, 0xB2, 0xB3, 0x73, 0xB1, 0x71,

0x70, 0xB0, 0x50, 0x90, 0x91, 0x51, 0x93, 0x53, 0x52, 0x92,

0x96, 0x56, 0x57, 0x97, 0x55, 0x95, 0x94, 0x54, 0x9C, 0x5C,

0x5D, 0x9D, 0x5F, 0x9F, 0x9E, 0x5E, 0x5A, 0x9A, 0x9B, 0x5B,

0x99, 0x59, 0x58, 0x98, 0x88, 0x48, 0x49, 0x89, 0x4B, 0x8B,

0x8A, 0x4A, 0x4E, 0x8E, 0x8F, 0x4F, 0x8D, 0x4D, 0x4C, 0x8C,

0x44, 0x84, 0x85, 0x45, 0x87, 0x47, 0x46, 0x86, 0x82, 0x42,

0x43, 0x83, 0x41, 0x81, 0x80, 0x40

} ;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

功能: CRC16校验

输入:

输出:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

unsigned int Modbus\_CRC16( unsigned char \*puchMsg, unsigned int usDataLen )

{

unsigned char uchCRCHi = 0xFF ; // 高CRC字节初始化

unsigned char uchCRCLo = 0xFF ; // 低CRC 字节初始化

unsigned long uIndex ; // CRC循环中的索引

while ( usDataLen-- ) // 传输消息缓冲区

{

uIndex = uchCRCHi ^ \*puchMsg++ ; // 计算CRC

uchCRCHi = uchCRCLo ^ auchCRCHi[uIndex] ;

uchCRCLo = auchCRCLo[uIndex] ;

}

return ( uchCRCHi << 8 | uchCRCLo ) ;

}

定时器底层驱动modbus\_time.c

#include "modbus\_time.h"

#include "modbus.h"

#include "msp430x54x.h"

#include <stdio.h>

void Modbus\_Time\_Init()

{

TA1CTL = TASSEL\_2 + TACLR + ID\_3 + MC\_1 + TAIE;

//TASSEL\_2 使用SMCLK 16M

//TACLR清0 计数器

//(ID3)8分频 计数频率 = 2M 0.5us计数一次

//MC\_1 增量模式，定时到指定时间 Up to CCR0

//TAIE 开定时器中断

TA1CCR0 = 1000; //500us CCR0中断一次 计数器计数到CCR0 产生A0中断

TA1CCTL0 |= CCIE; //打开CCR0中断

}

#pragma vector=TIMER1\_A1\_VECTOR

\_\_interrupt void TIMER1\_TA1\_ISR(void)

{

switch(\_\_even\_in\_range(TA1IV,14))

{

case 0: break; // No interrupt

case 2: break; // CCR1 not used

case 4: break; // CCR2 not used

case 6: break; // reserved

case 8: break; // reserved

case 10: break; // reserved

case 12: break; // reserved

case 14: //printf("TIMEA1 IRQ...\n");

// overflow 定时器65536溢出中断

break;

default: break;

}

}

// Modbus 定时器中断函数 500us中断一次

#pragma vector=TIMER1\_A0\_VECTOR

\_\_interrupt void TIMER1\_TA1\_ISR0(void)

{

if(modbus.timrun != 0)

{

modbus.timout++;

if(modbus.timout >=8)

{

modbus.timrun = 0;

modbus.reflag = 1;

}

}

}

串口底层驱动 modbus\_uart.c

#include "modbus\_uart.h"

#include "uart0.h"

#include "msp430x54x.h"

#include <stdio.h>

// Modbus串口初始化

void Modbus\_Uart\_Init()

{

usart0Init(115200); //使能Moubud USART

}

//modbus串口发送一个字节数据

void Modbus\_Send\_Byte(unsigned char Modbus\_byte)

{

uart0SendByte(Modbus\_byte);

}

//串口接收中断服务程序

#pragma vector=USCI\_A0\_VECTOR

\_\_interrupt void USCI\_A0\_ISR(void)

{

unsigned char rxData;

switch(\_\_even\_in\_range(UCA0IV,4))

{

case 0:break; // Vector 0 - no interrupt

case 2:

rxData = UCA0RXBUF; // Vector 2 - RXIFG 接收中断

if( modbus.reflag==1) //有数据包正在处理

{

return ;

}

modbus.rcbuf[modbus.recount++] = rxData;

modbus.timout = 0;

if(modbus.recount == 1) //已经收到了第二个字符数据

{

modbus.timrun = 1; //开启modbus定时器计时

}

break;

case 4:break; // Vector 4 - TXIFG 发送中断

default: break;

}

}

对以上MODBUS代码进行逻辑解析

以上代码能正常对MODBUS POLL通信，但是没有实现安全检测功能: 安全检测功能如下:

1.主机发送设备没有定义的寄存器地址 🡪 设备没有这个地址 🡪 设备返回寄存器地址错误

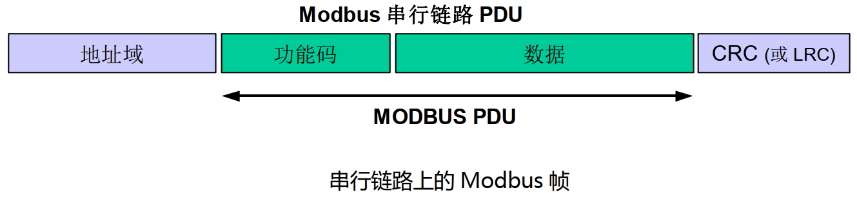
2.主机发送数据给设备🡪 设备检测CRC校验 🡪 如果CRC校验错误，设备返回给主机错误信息

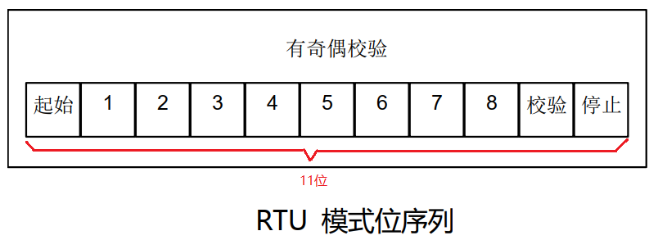
3.主机发送功能码给设备 🡪 设备没有实现该功能码 🡪 设备返回功能码错误给主机

为了解决以上内容，我们必须熟悉MODBUS协议的细节

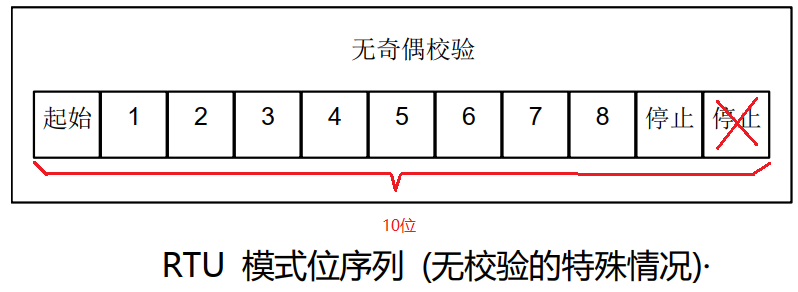
LRC是用于MODBUS ASCII协议校验的

我使用的RTU 只需要CRC就行了



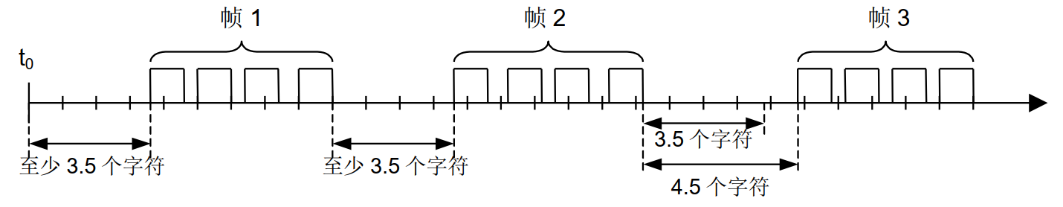
带奇偶校验的RTU 字节(串口字节)为11位

如果是11位RTU 情况下1个字符(字节)就包括11位，那么3.5个字符(字节)就是3.5\*11=38.5位

无奇偶校验的RTU 字节(串口字节)为10位

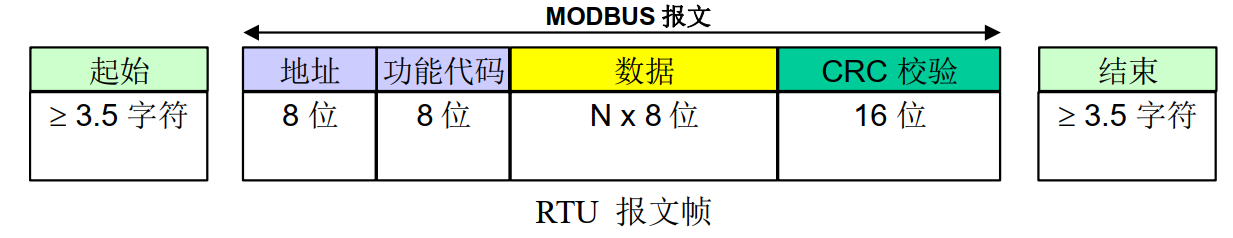
如果是10位的RTU 3.5个字符就是3.5\*10=35位

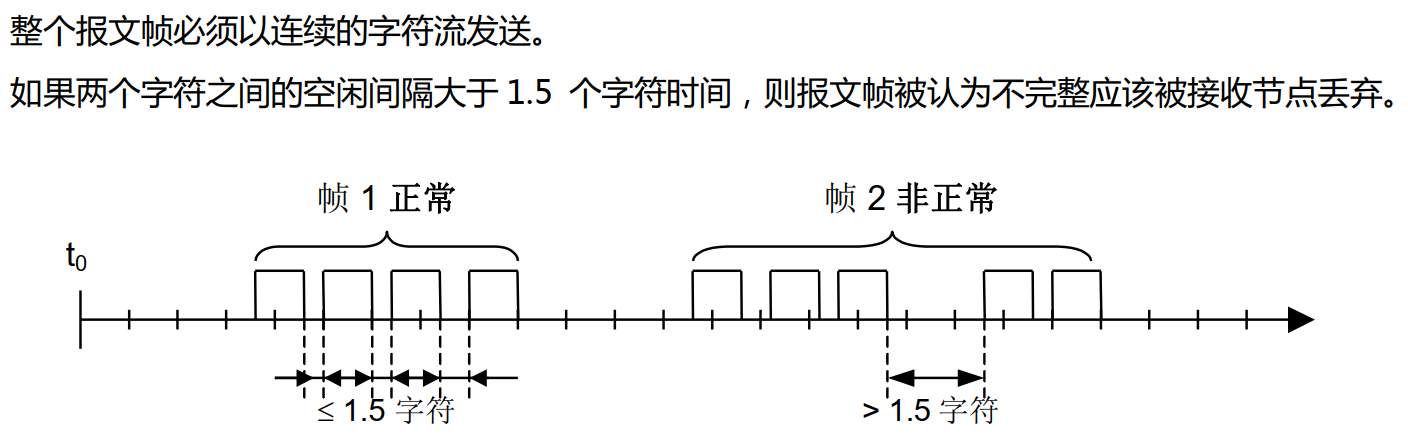
第2帧串口数据包



3.5个字符(字节)时间没有数据 证明一帧数据包结束

第1帧串口数据包





一帧数据包两个字节之间间隔时间<1.5

在通信速率<19200 bps 时，如9600bps，建议的字符间超时时间(t1.5)为 750µs

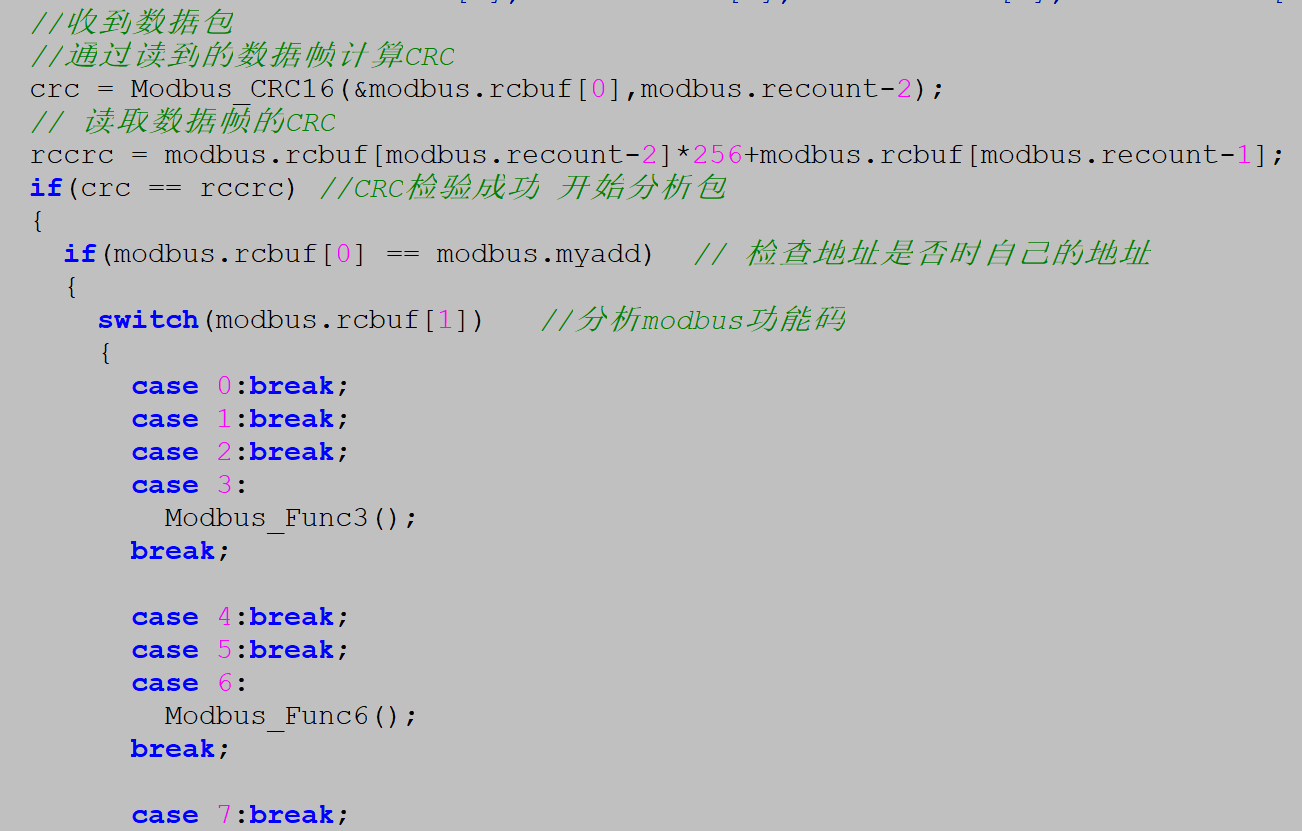
在通信速率>19200 bps 时，如115200bps, 帧间的超时时间 (t1.5) 为 1.750ms

比如9600bps，意思就是说每1秒（也就是1000毫秒）传输9600个位，  
反过来说传输9600个二进制位需要1000毫秒

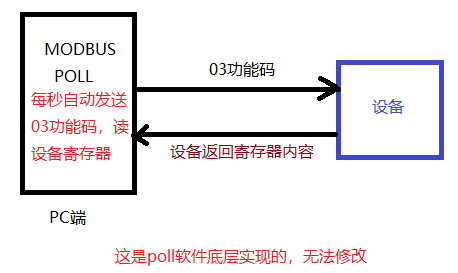
那么传输38.5个二进制位需要的时间就是：  
38.5/9.6=4.0104167毫秒

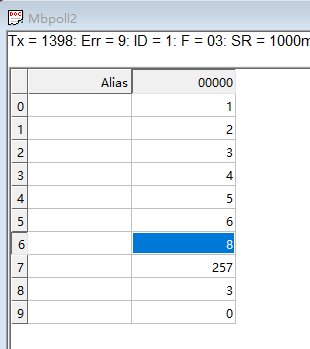
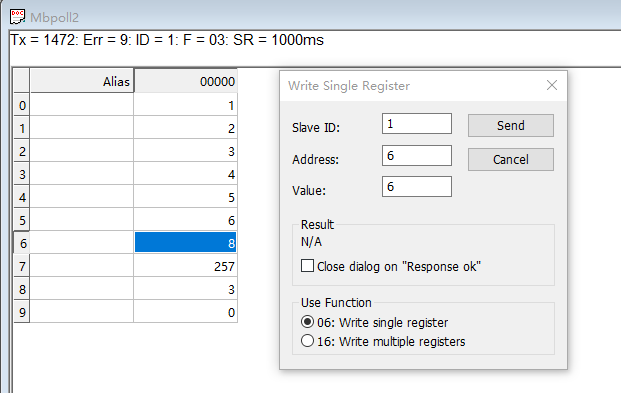
MODBUS RTU要求一帧数据起始和结束至少有大于等于3.5个字符的时间  
在波特率为9600的情况下，只要大于4.0104167毫秒即可！

功能码03和06解析



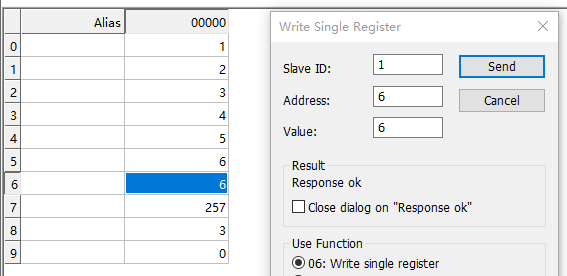
所以在死循环事件循环中，03功能会一直被调用



如果在03功能界面去修改某个寄存器的值，你会发现是按照06功能码去修改的

如果设备内部程序自动修改了寄存器值，那么这里会自动变化，因为poll在用03功能码不停的循环读取

 注意06功能码只能写单个寄存器

数据是被06功能码写进寄存器的，修改成功

下面实现10(十进制16)功能码

// Modbus 0x10(十进制16)号功能码函数

// Modbus 主机写入多个寄存器值

void Modbus\_Func10(void)

{

unsigned int startAddr = 0;

unsigned int regNum = 0; //寄存器数量

unsigned char ByteNum = 0; //后面跟的字节数 最多255

unsigned int i,crc,j,Count = 0; //串口接受的缓存单独用Count计算

i=0;

startAddr = modbus.rcbuf[2]\*256+modbus.rcbuf[3]; //得到要修改寄存器的起始地址

regNum = modbus.rcbuf[4]\*256+modbus.rcbuf[5]; //得到修改的寄存器数量

ByteNum = modbus.rcbuf[6];

for(i = 0; i < regNum+1;i++)

{

Count++;

Reg[startAddr + i] = (modbus.rcbuf[6+Count]<<8); //数据高字节

Count++;

Reg[startAddr + i] = (Reg[startAddr + i]) | (modbus.rcbuf[6+Count]); //数据低字节

}

// printf("%x %x %x %x %x %x \n",Reg[0],Reg[1],Reg[2],Reg[3],Reg[4],Reg[5]);//查看写入是否正确

i = 0;

//以下为回应主机

modbus.sendbuf[i++]=modbus.myadd;//本设备地址

modbus.sendbuf[i++]=0x10; //功能码16

modbus.sendbuf[i++]=startAddr/256;

modbus.sendbuf[i++]=startAddr%256;

modbus.sendbuf[i++]=regNum/256;

modbus.sendbuf[i++]=regNum%256;

crc=Modbus\_CRC16(modbus.sendbuf,i);

modbus.sendbuf[i++]=crc/256;

modbus.sendbuf[i++]=crc%256;

for(j=0;j<i;j++)

{

Modbus\_Send\_Byte(modbus.sendbuf[j]);

}

}

unsigned int Reg[] ={ 0x0001,

0x0002,

0x0003,

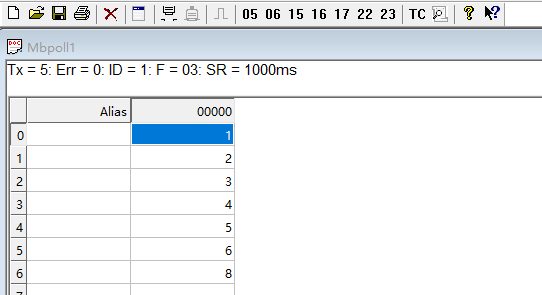
0x0004,

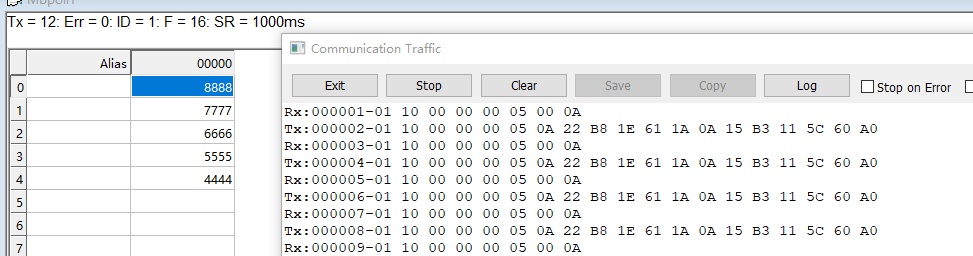
0x0005,

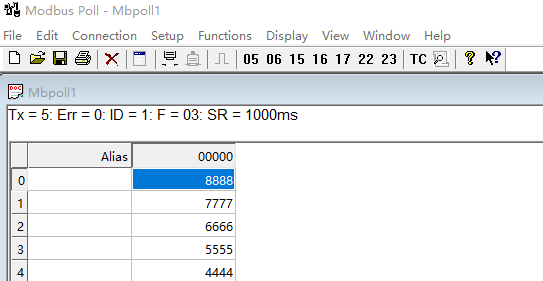
0x0006,

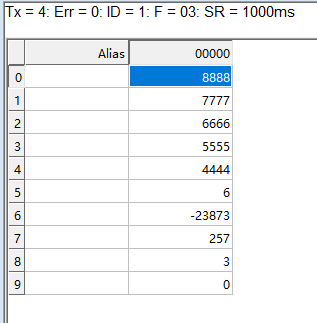
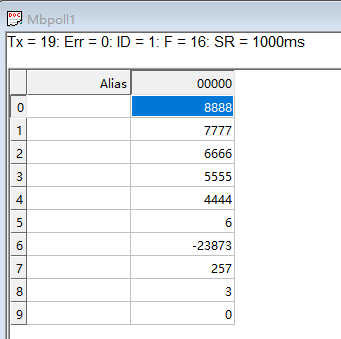
0x0008,

}; //自己定义的寄存器 开机poll默认读取该寄存器的值

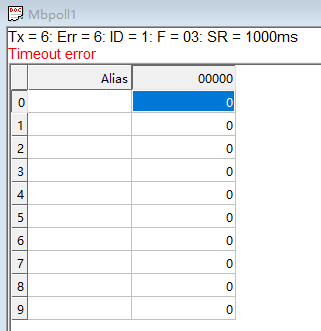
开机03功能码，默认读取数据是Reg数组的值

 10功能码，写数据没有问题

 03功能码读取这几个寄存器的值，没问题

如果03功能码读越界 或者10写越界

或者操作成04功能码了，那么再进行03功能码读，或者10功能码写，都会出现Timeout

因为，我并没有实现04功能码程序，所以使用的时候要注意。

实现MODBUS地址越界判断

/ /Modbus 0x10(十进制16)号功能码函数

// Modbus 主机写入多个寄存器值

void Modbus\_Func10(void)

{

unsigned int startAddr = 0;

unsigned int regNum = 0; //寄存器数量

unsigned char ByteNum = 0; //后面跟的字节数 最多255

unsigned int i,crc,j,Count = 0; //串口接受的缓存单独用Count计算

i=0;

startAddr = modbus.rcbuf[2]\*256+modbus.rcbuf[3]; //得到要修改寄存器的起始地址

regNum = modbus.rcbuf[4]\*256+modbus.rcbuf[5]; //得到修改的寄存器数量

ByteNum = modbus.rcbuf[6];

for(i = 0; i < regNum+1;i++)

{

Count++;

Reg[startAddr + i] = (modbus.rcbuf[6+Count]<<8); //数据高字节

Count++;

Reg[startAddr + i] = (Reg[startAddr + i]) | (modbus.rcbuf[6+Count]); //数据低字节

}

i = 0;

//以下为回应主机

modbus.sendbuf[i++]=modbus.myadd;//本设备地址

/\*如果起始地址越界，超过数组元素总量。或者起始地址正确，读取寄存器的时候，

寄存器数量超过元素总量，都会返回07功能吗报错\*/

if((startAddr > 8 || startAddr < 0) || (regNum > 8 || regNum < 0))

modbus.sendbuf[i++]=0x07; //功能码07 读取异常状态

else

modbus.sendbuf[i++]=0x10; //功能码16

modbus.sendbuf[i++]=startAddr/256;

modbus.sendbuf[i++]=startAddr%256;

modbus.sendbuf[i++]=regNum/256;

modbus.sendbuf[i++]=regNum%256;

crc=Modbus\_CRC16(modbus.sendbuf,i);

modbus.sendbuf[i++]=crc/256;

modbus.sendbuf[i++]=crc%256;

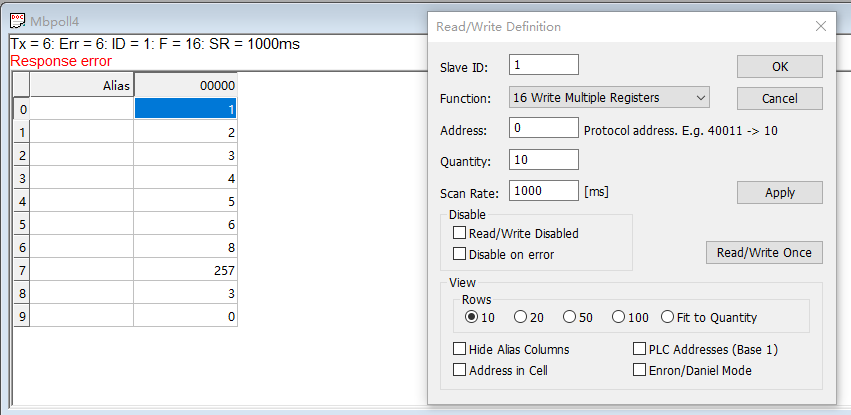
for(j=0;j<i;j++)

{

Modbus\_Send\_Byte(modbus.sendbuf[j]);

}

}

我发现返回07之后不及时出来，设备就一直Timeout，只有重启才能正常。这个问题只有后续来处理。

写入的寄存器数量超过数组数量，也会返回07

起始地址超过MODBUS寄存器数组数量会返回07

**RTC时钟使用**

未完待续…………………………….