

# MSP430F5438A 操作手册

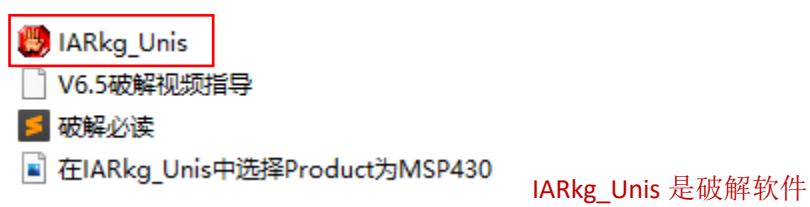
## 作者:向仔州

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
<b>问题 1 报错: Error[e183]: Static overlay map generation (-xo) is not supported for the MSP430 processor.....</b>	14
<b>问题 2: 工程路径存放太深, 会出现 The following path is too long:.....</b>	14
<b>问题 3: 将 IAR5.3 的工程用 IAR7.12 打开, 代码无法跳转.....</b>	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
<b>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.....</b>	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
<b>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.....</b>	31
定时器 B 使用.....	32
<b>定时器 B 指定计数器计数次数定时.....</b>	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
<b>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.....</b>	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。

EW430-7121-Autorun 这是 IAR7.12 安装软件



• Install IAR Embedded Workbench® for MSP430

选择第 1 项安装  
AR Embedded Workbench for MSP430 7.12.1

Welcome to the InstallShield Wizard for IAR Embedded Workbench for MSP430

The InstallShield Wizard will install IAR Embedded Workbench for MSP430 on your computer. To continue, click Next.

点击下一步

Next > Cancel

I accept the terms of the license agreement  
I do not accept the terms of the license agreement

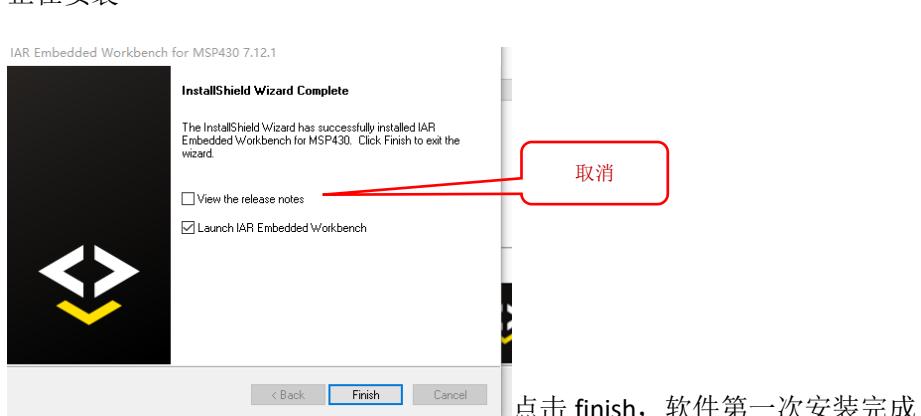
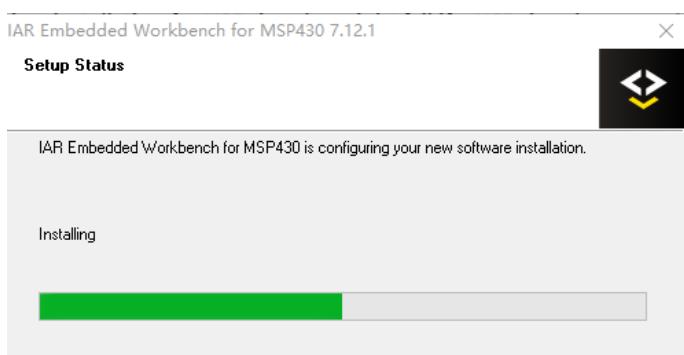
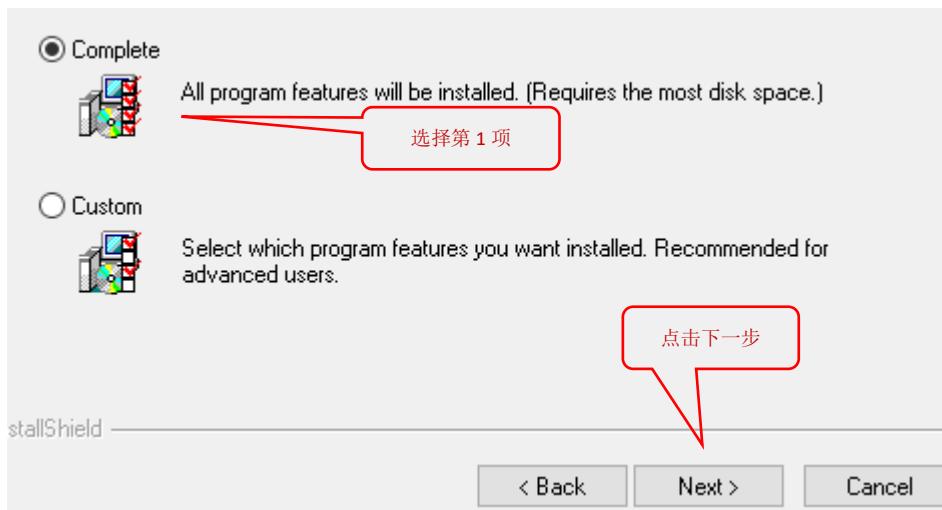
Print

InstallShield

Next > Cancel

Install IAR Embedded Workbench for MSP430 to:  
C:\...\IAR Systems\Embedded Workbench 8.0

Change... 记住安装路径



## Welcome



This wizard will help you to activate your IAR Embedded Workbench for Texas Instruments 430 license.

- If you have a license number, enter it here:

IAR 自动启动，会  
弹出注册窗口

- Use a network license
- Register with IAR Systems to get an evaluation license

- If you have a lic

- Use a network

- Register with IA

LicenseManager



Without an activated license you will  
not be able to use the product.  
You can restart license activation at  
any time using the IAR License Manager.

确定

不管警告，直接  
点击确定

- Don't run the Wizard for this product at startup.

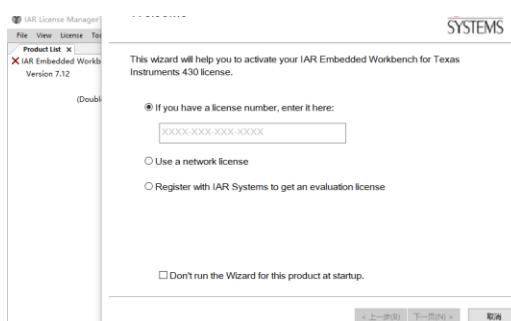
&lt; 上一步(B)

下一页(N) &gt;

取消

点击取消

点击 IAR → help → license Manager



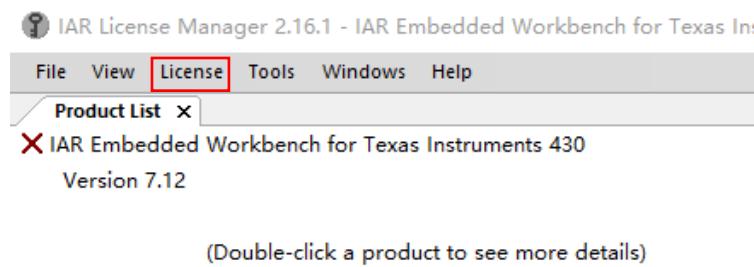
点击取消



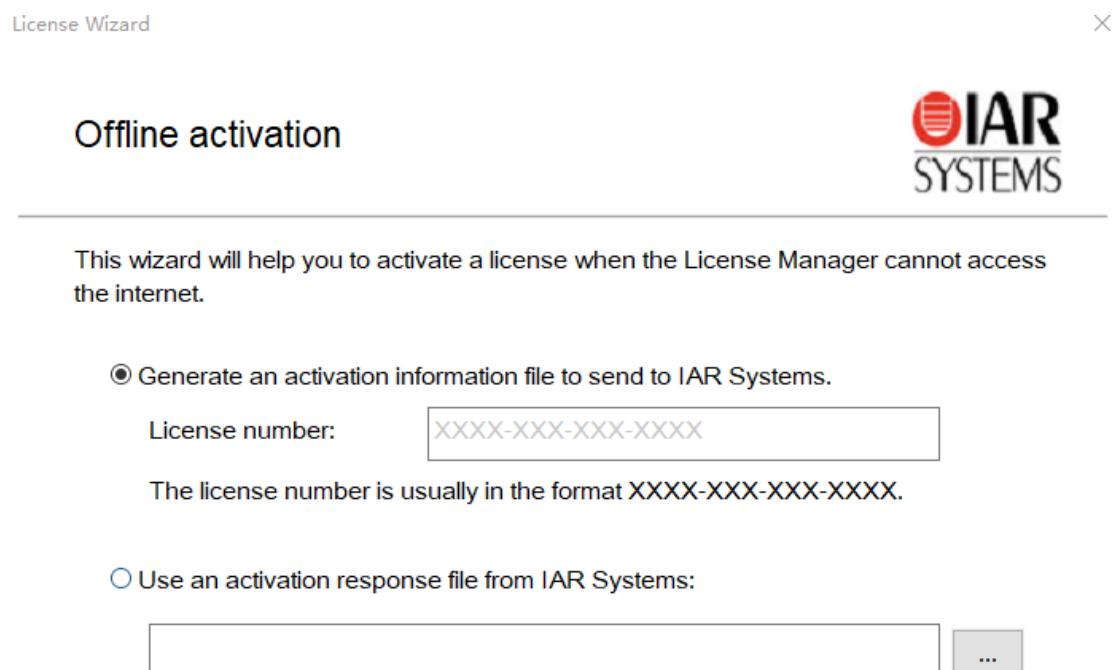
Don't run the Wizard for this product at startup.

< 上一步(B) 下一页(N) > 取消

点击顶部 License -> Offline activation

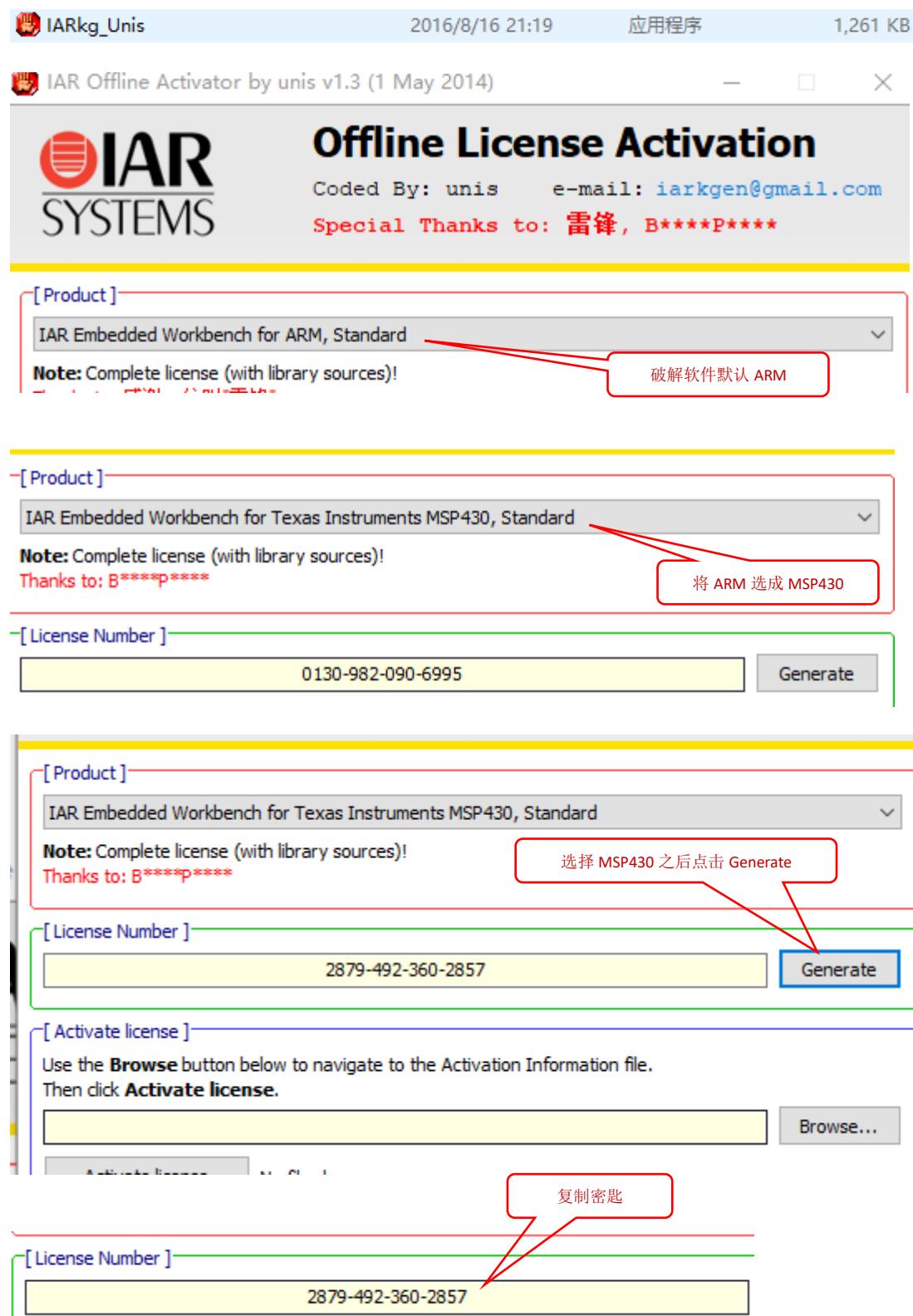


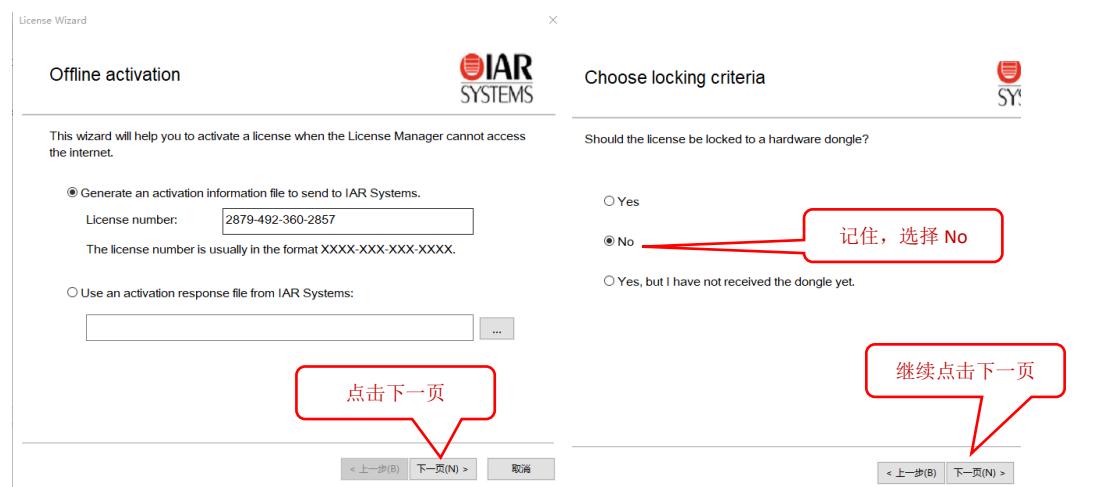
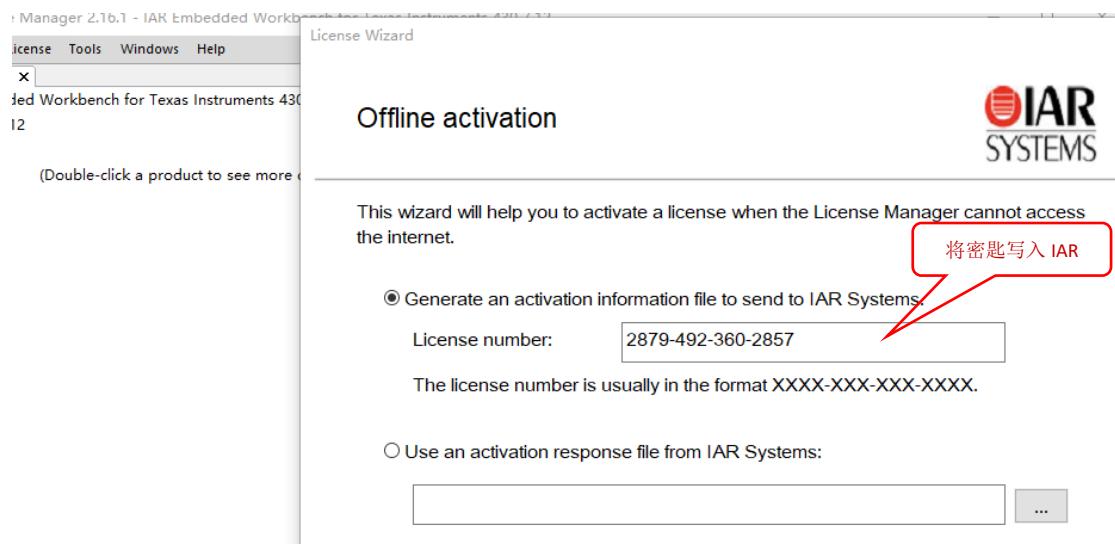
窗口停留在 product list



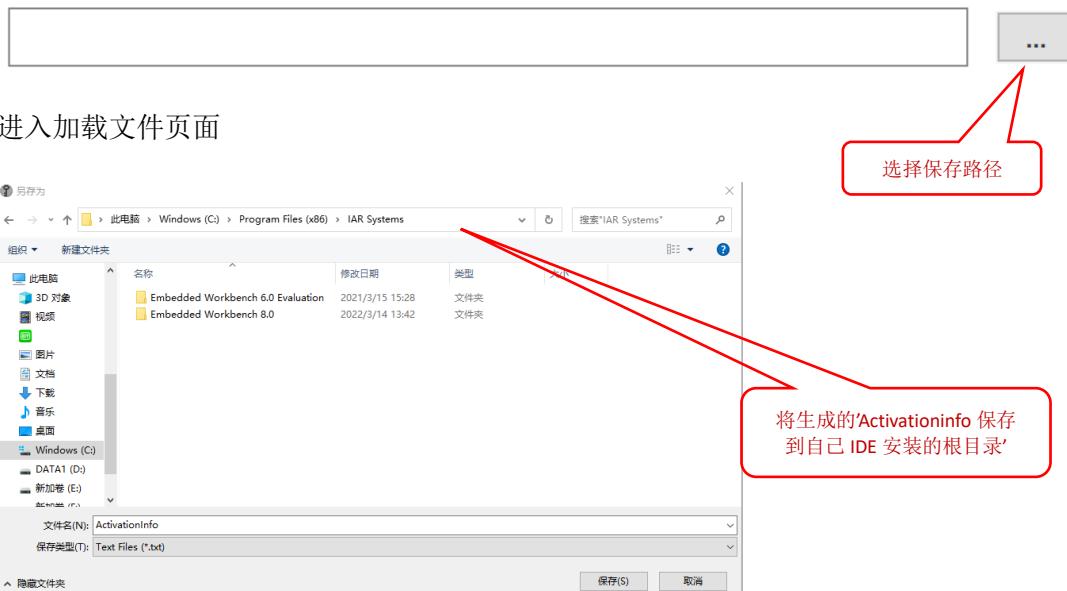
保持界面不动

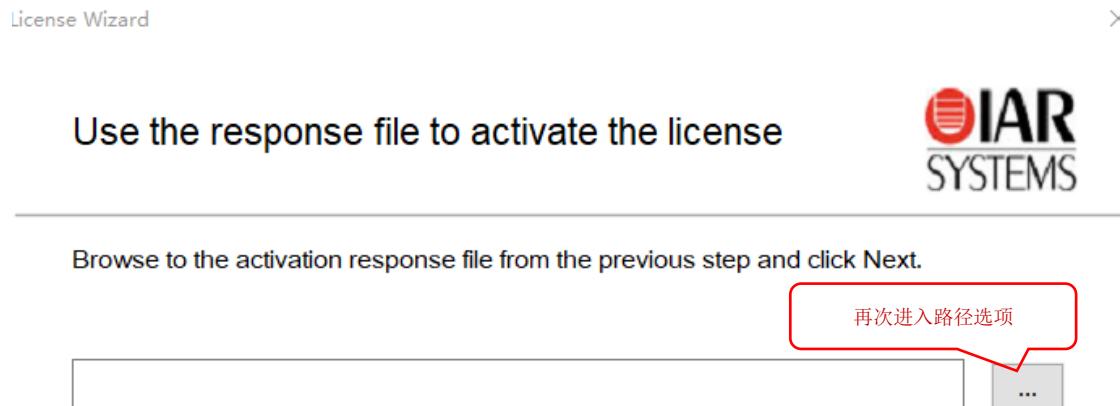
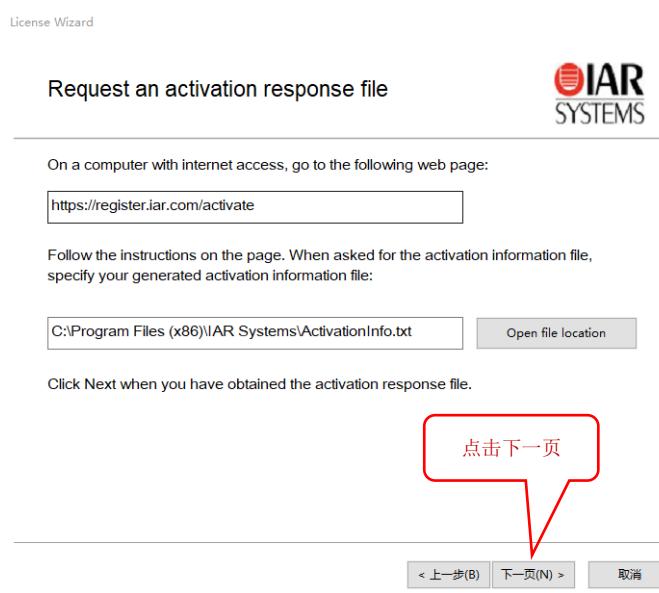
打开破解软件

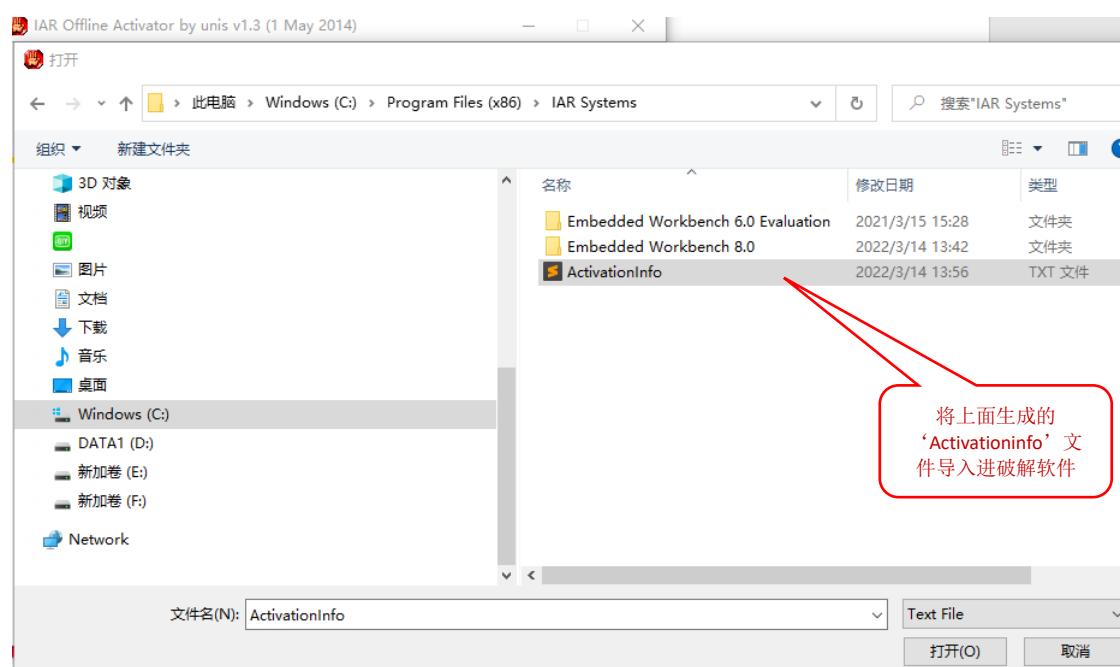
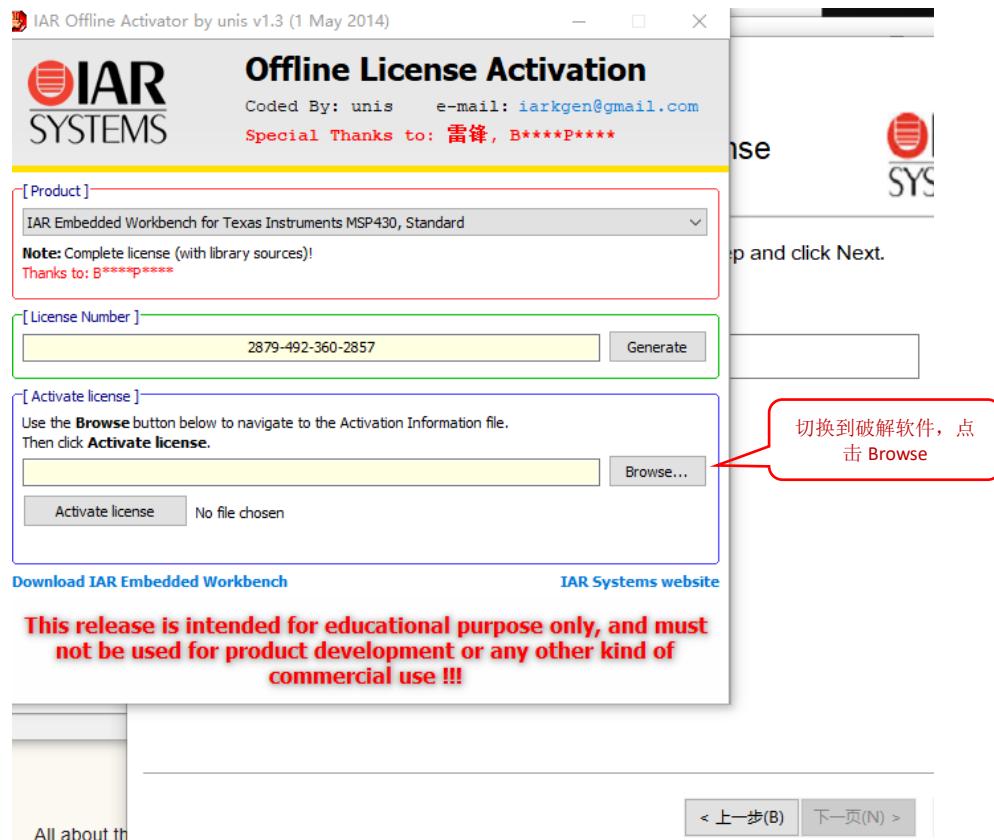


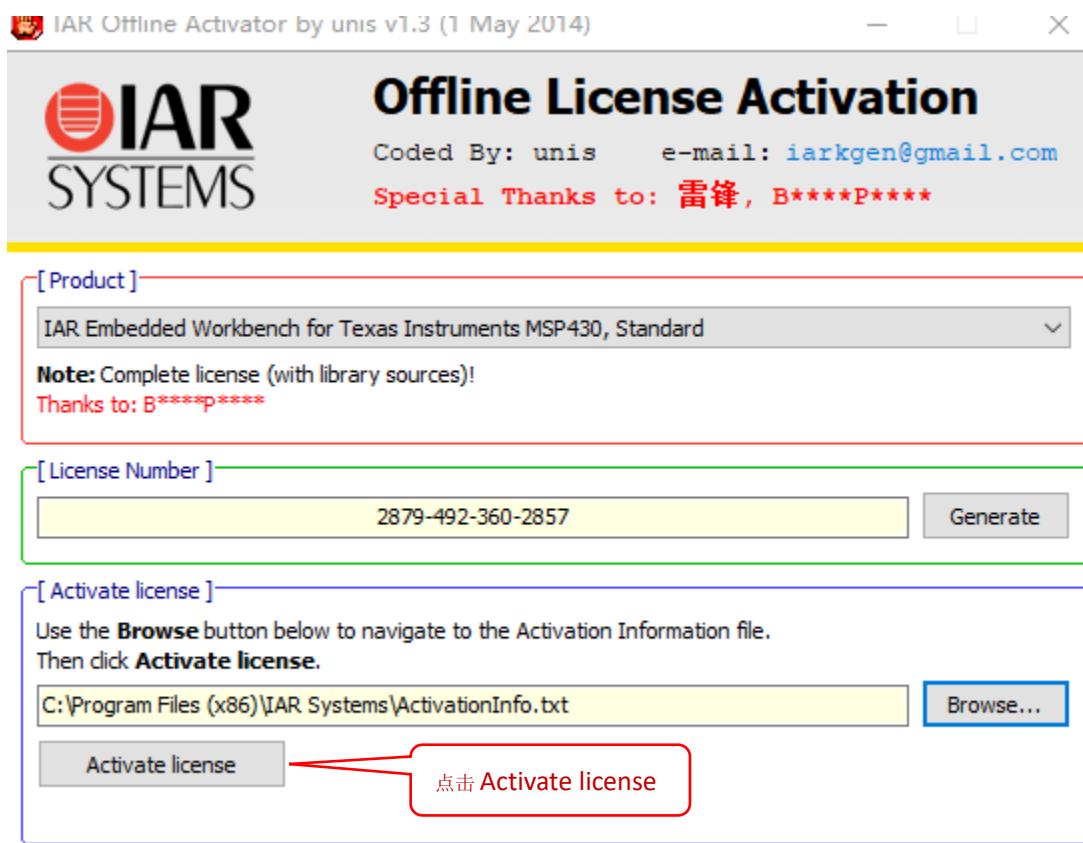


Choose where to save the activation information file.



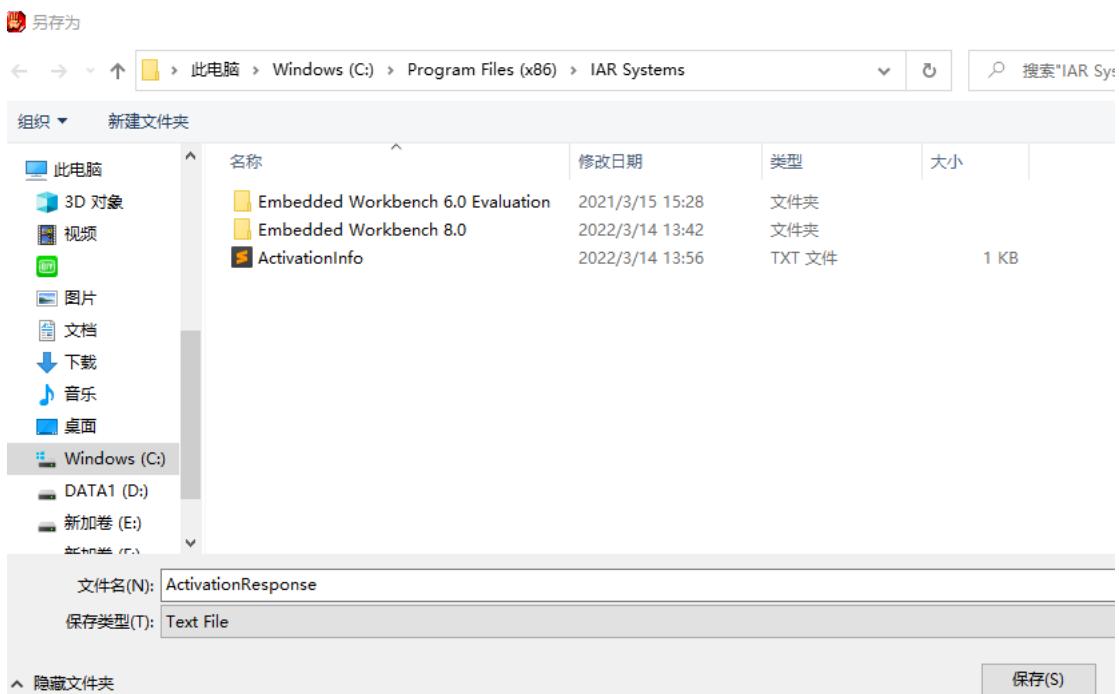




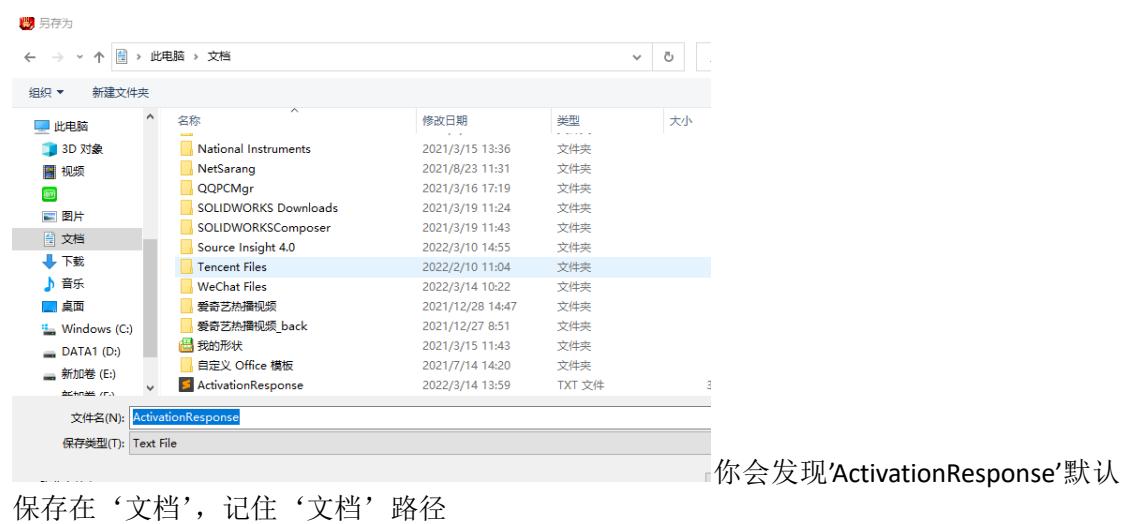
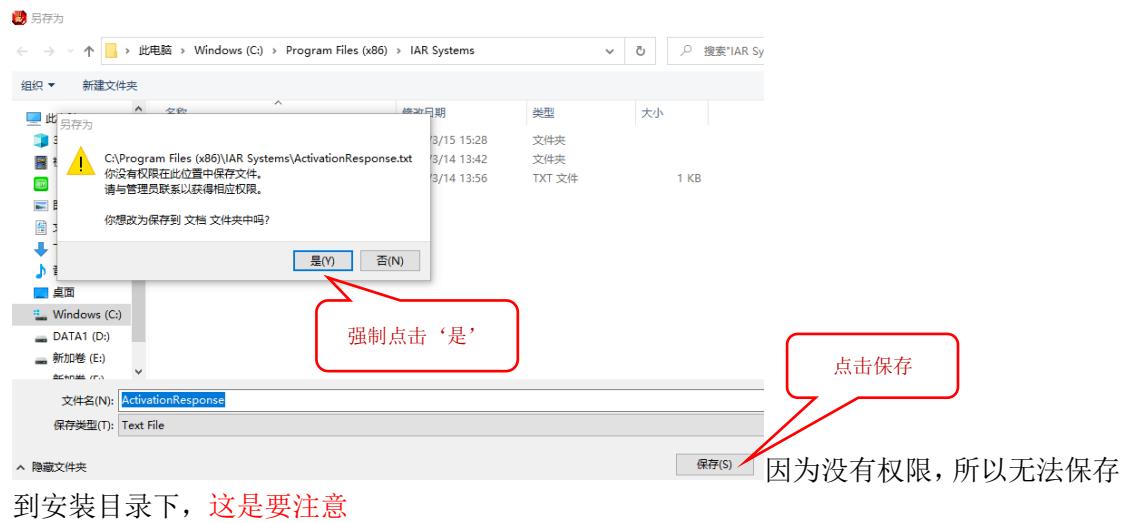


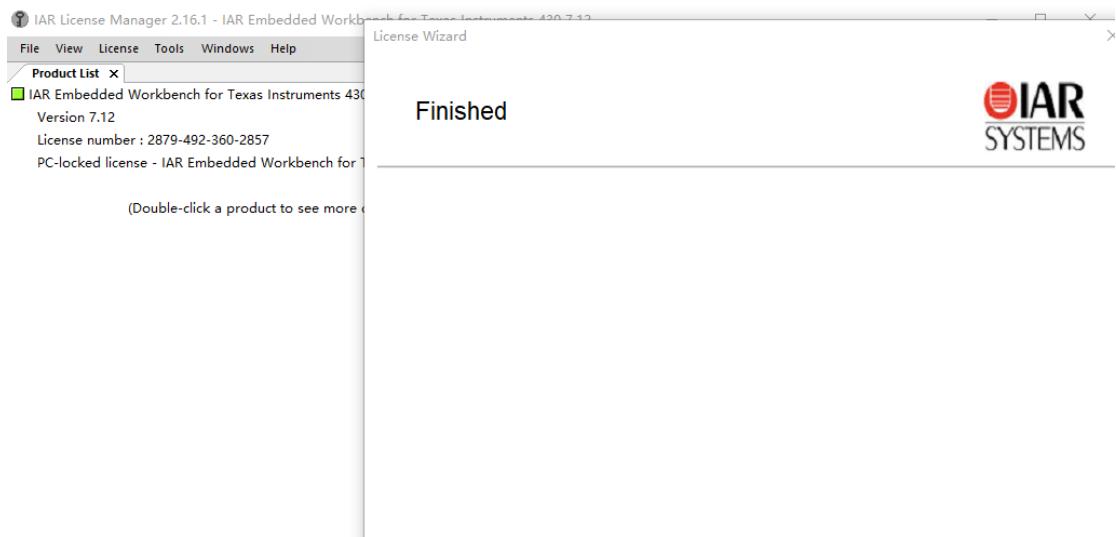
[Download IAR Embedded Workbench](#)

[IAR Systems website](#)

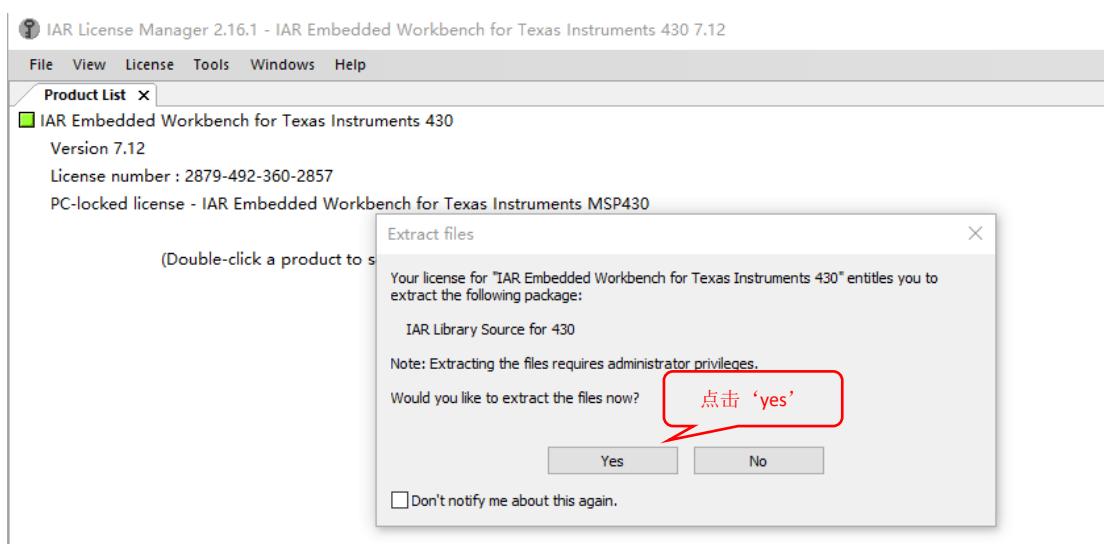


将'ActivationResponse'文件保存在安装根目录下

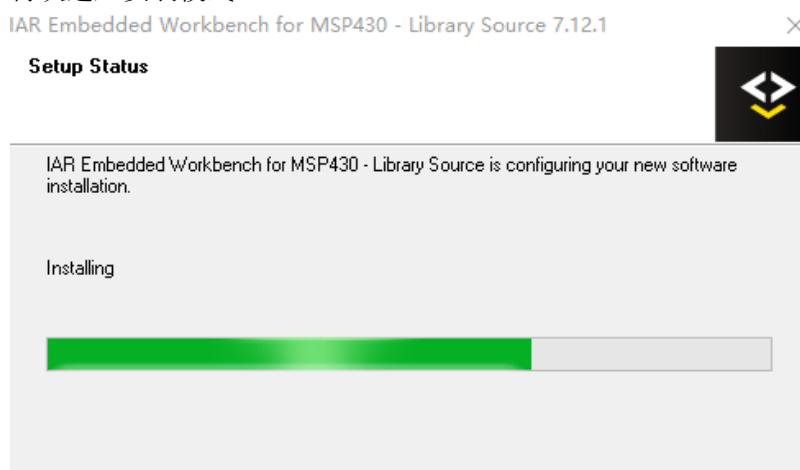


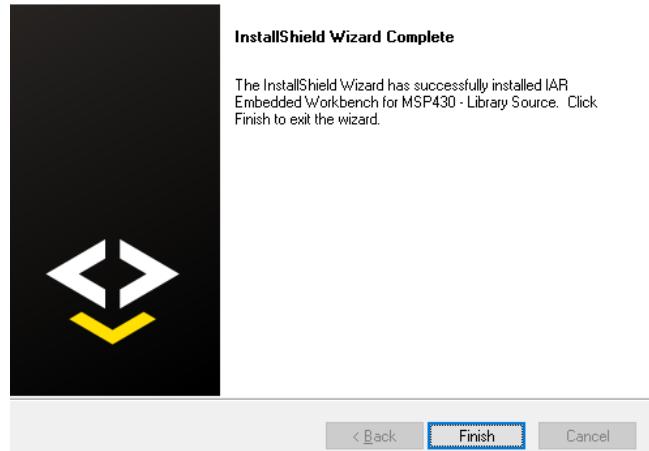


显示绿色，破解成功



再次进入安装模式





安装完成

IAR License Manager 2.16.1 - IAR Embedded Workbench for Texas Instruments 430

File View License Tools Windows Help

Product List X

IAR Embedded Workbench for Texas Instruments 430

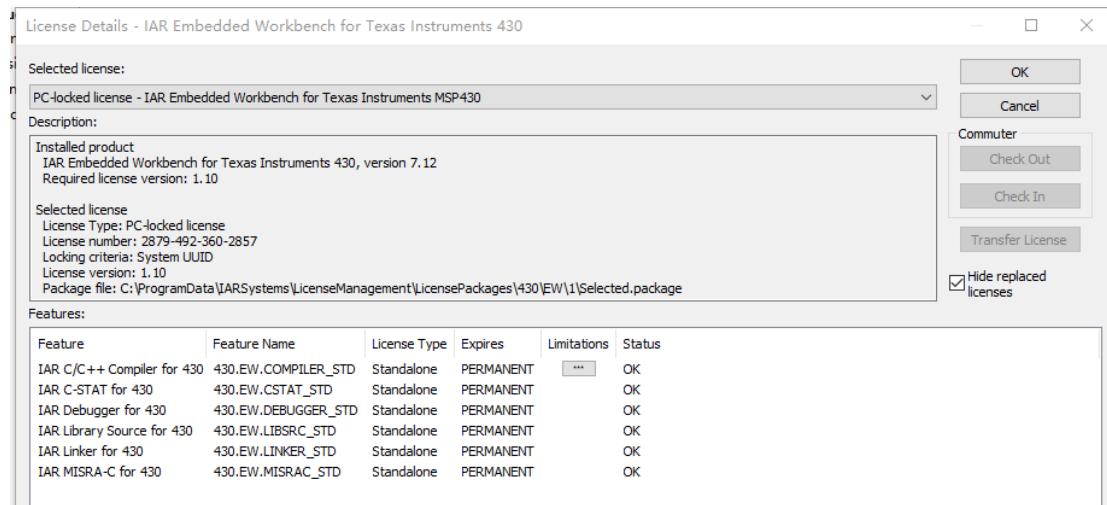
Version 7.12

License number : 2879-492-360-2857

PC-locked license - IAR Embedded Workbench for Texas Instruments MSP430

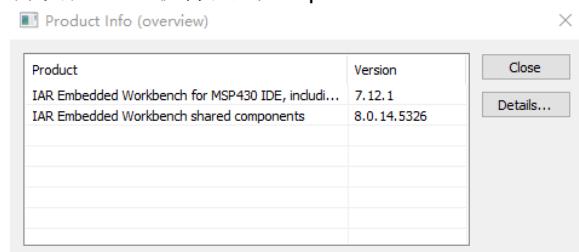
(Double-click a product to see more details)

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

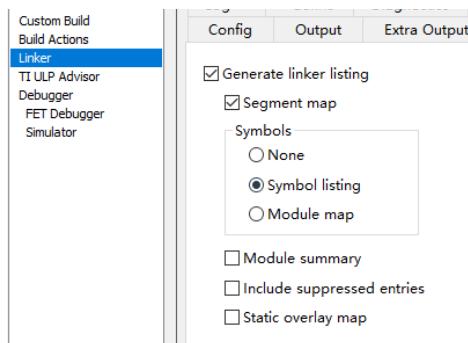
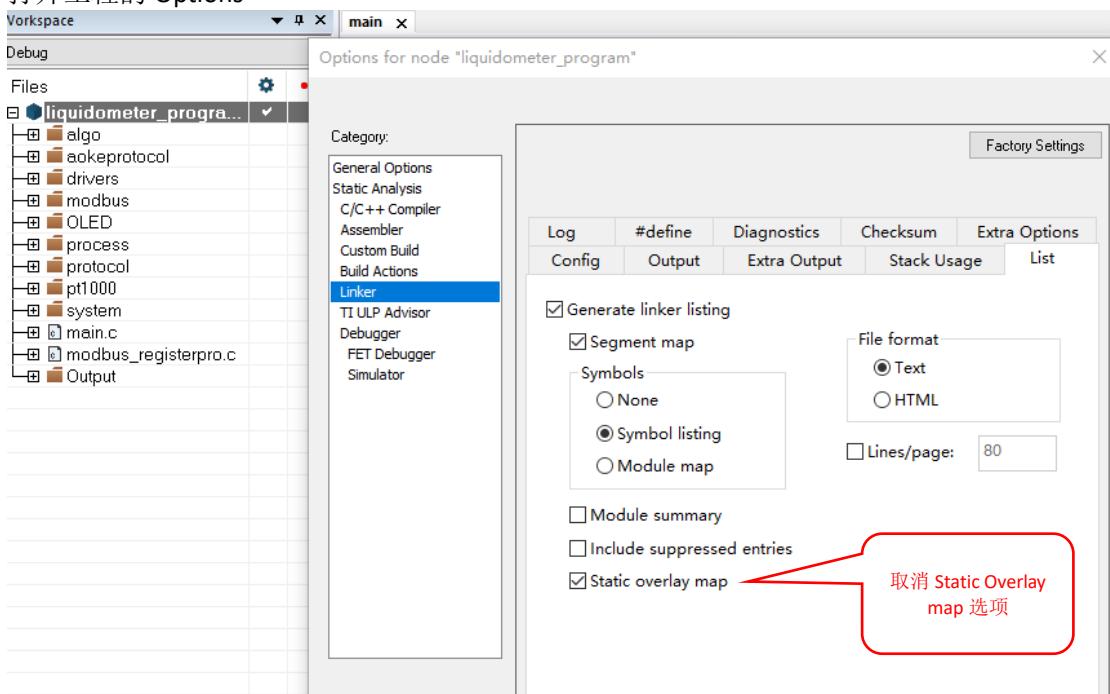
问题 1 报错: Error[e183]: Static overlay map generation (-xo) is not supported for the MSP430 processor

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。

```
Messages
Building configuration: liquidometer_program - Debug
Updating build tree...
Linking
✖ Error[e183]: Static overlay map generation (-xo) is not supported for the MSP430 processor.
✖ Error while running Linker

Total number of errors: 1
Total number of warnings: 0
```

打开工程的 Options



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

✖ The following path is too long:  
"E:\储德项目\磁敏液位计模块项目\CEAS RD-CP-CX-02 产品设计阶段V20 CEAS RD-CP-CX-02.006 软件详细设计\磁敏液位计正式应用程序\磁敏液位计硬壳\应用程序\型\liquidometer\_program.ewp".

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

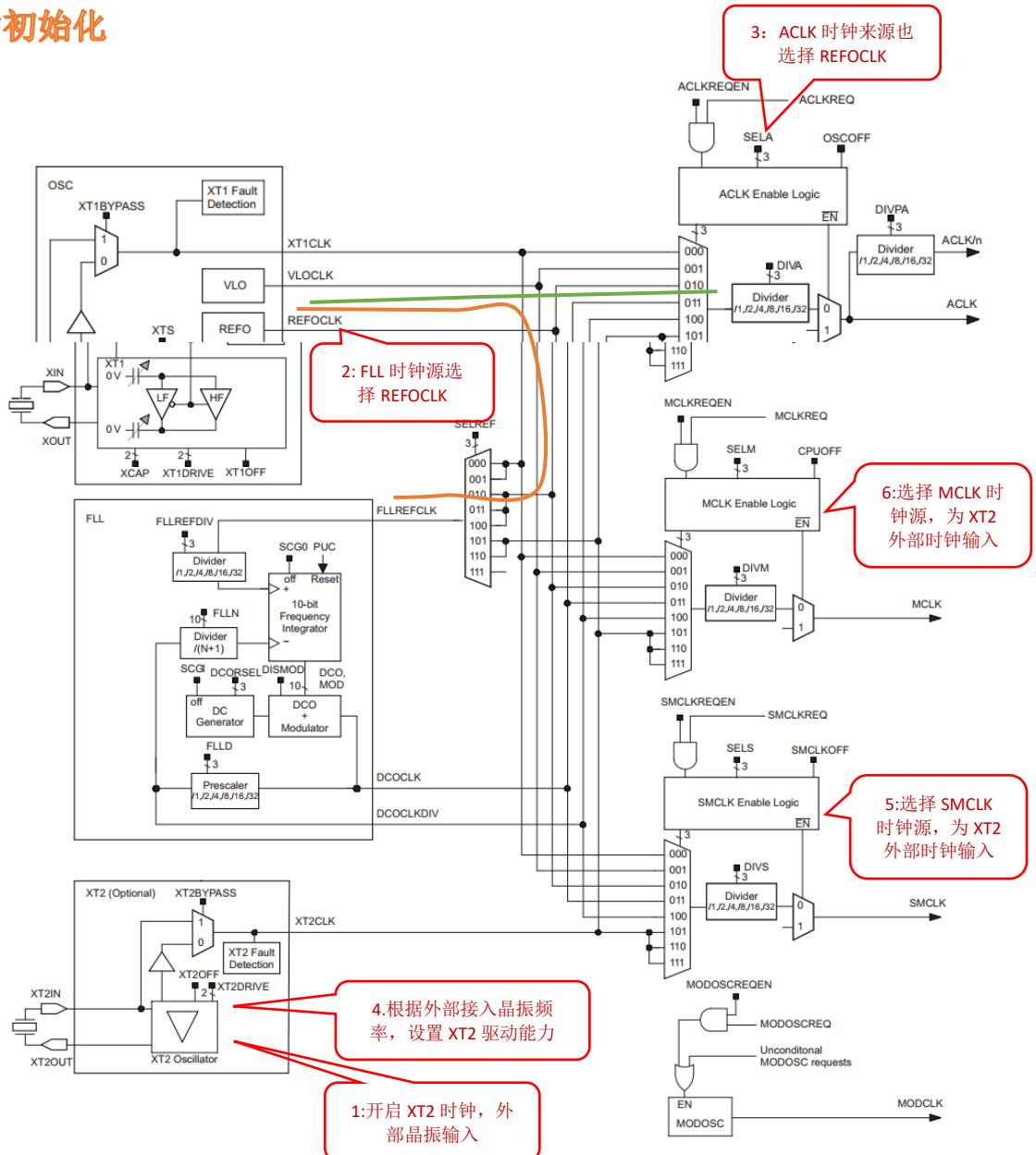


Figure 12-14. PxSEL Register

7	6	5	4	3	2	1	0
PxSEL							
rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0

1 1

配置P5.2 , P5.3位, 为外部晶振输入引脚(外设功能)

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

Figure 5-12. UCSCTL6 Register

15	14	13	12	11	10	9	8
1	XT2DRIVE	1	Reserved	XT2BYPASS		Reserved	XT2OFF
外部晶振16M, 设置最大驱动电流							
7	6	5	4	3	2	1	0
XT1DRIVE	XTS	XT1BYPASS		XCAP	SMCLKOFF	XT1OFF	
rw-1	rw-1	rw-0	rw-0	rw-1	rw-1	rw-0	rw-1

开XT2外部时钟

UCSCTL6 &= ~(XT2OFF); //开启 XT2 时钟  
 UCSCTL6 = XT2DRIVE1 + XT2DRIVE0; //驱动晶振的能力是 24M 到 32M

Figure 5-9. UCSCTL3 Register

15	14	13	12	11	10	9	8
Reserved							
r0	r0	r0	r0	r0	r0	r0	r0
7	6	5	4	3	2	1	0
Reserved	SELREF			Reserved	FLLREFDIV		
r0	rw-0	1	0	r0	rw-0	rw-0	rw-0

FLL 选择REFOCLK 做时钟源,

也就是XTA1时钟源

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

Figure 5-10. UCSCTL4 Register

15	14	13	12	11	10	9	8
Reserved							
r0	r0	r0	r0	r0	1	0	
7	6	5	4	3	2	1	0
Reserved	SELS			Reserved	SELM		
1	0	1			1	0	1

XT2CLK 外部晶振经过分

频做SMCLK主时钟

XT2CLK 外部晶振经过

分频做MCLK主时钟

UCSCTL4 |= SELA\_2;

Figure 1-9. SFRIFG1 Register

15	14	13	12	11	10	9	8
Reserved							
r0	r0	r0	r0	r0	r0	r0	r0
7	6	5	4	3	2	1	0
JMBOUTIFG	JMBINIFG	Reserved	NMIIIFG	VMAIFG	Reserved	OFIFG <sup>(1)</sup>	WDTIFG <sup>(2)</sup>
rw-(1)	rw-(0)	r0	rw-0	rw-0	r0	1	

开启振荡器故障

中断标志位

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) //外部高頻晶振为 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 XT02 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

TXD0	39	P3.4/UCA0TXD/UCA0SIMO
RXD0	40	P3.5/UCA0RXD/UCA0SOMI

Figure 12-14. PxSEL Register							
7	6	5	4	3	2	1	0
			PxSEL				
1	1		rw-0	rw-0	rw-0	rw-0	rw-0

Px.4,Px.5置1，IO口外

设功能开启

Figure 36-13. UCAxCTL1 Register							
7	6	5	4	3	2	1	0
UCSSELx	UCRXIE	UCBRKIE	UCDORM	UCTXADDR	UCTXBRK	UCSWRST	
1	0						1

选择SMCLK时钟源做

串口时钟

先复位串口，才

允许设置寄存器

UCA0CTL1 |= UCSSEL\_2;

Figure 36-14. UCAxBR0 Register							
7	6	5	4	3	2	1	0
			UCBRx	0x82			

Figure 36-15. UCAxBR1 Register

Figure 36-15. UCAxBR1 Register							
7	6	5	4	3	2	1	0
			UCBRx	0x06			

比如串口输入时钟SMCLK = 16M，我要求波特率设置为9600

$$16000000 / 9600 = 1666.666$$

将1666.666小数点前面的位取出来算成16进制 = 0x0682

高8位放BR1  
低8位放BR0

1666.666 小数部分0.666用MCTL寄存器来设置

小数部分计算 N = SMCLK/Baud = 16000000 / 9600 = 1666.666

BRSx寄存器计算 = (N - INT(N) \* 8) = (1666.666 - 1666) \* 8 = 5.328 (一般向上取整，就是6)

INT(N) 就是取N得整数部分1666.666取1666

如果向下离整数近，

可以取5

UCBSRx 写入0x06 (0110)

Figure 36-16. UCAxMCTL Register							
7	6	5	4	3	2	1	0
0	0	0	0	UCBRSx	0x06	UCOS16	0

没有启动过采样，所以写0

N的小数部分

不启动过采样

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 XT2 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
    }
}

```

A1 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)
    UCA0BRO = 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
    }
}
```

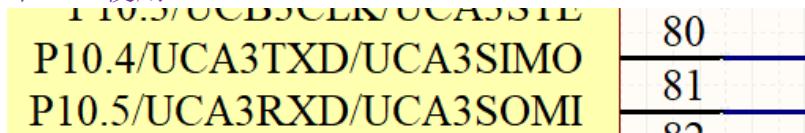
```

/*
 * 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
    }
}

A1 A1 A1 A1 A1 A1 A1
A1 A1 A1 A1 A1 A1 A1

```

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

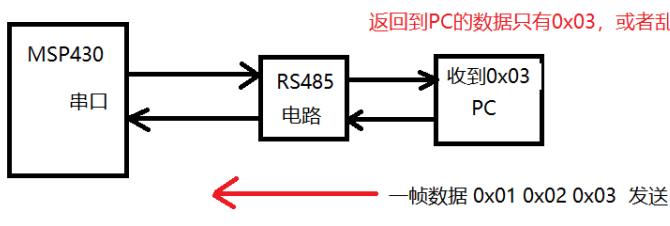
    _EINT();

    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);
    }

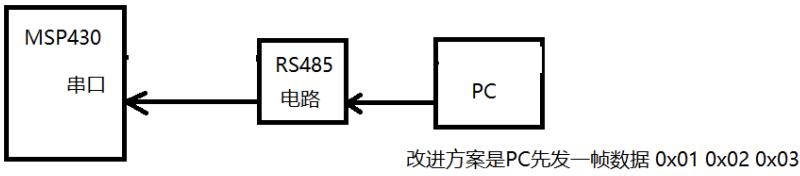
    #pragma vector=USCI_A3_VECTOR
    __interrupt void USCI_A3_ISR(void)
    {
        unsigned char rxData;
        switch(__even_in_range(UCA3IV,4))
        {
            case 0:break; // Vector 0 - no interrupt
            case 2:
                rxData = UCA3RXBUF; // Vector 2 - RXIFG 接收中断
                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。

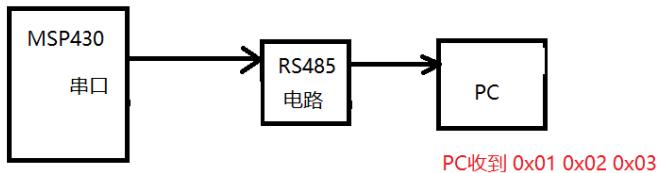


这是因为 MSP430串口返回数据太快，RS485来不及手发切换切换



改进方案是PC先发一帧数据 0x01 0x02 0x03

单片机延时100us或者毫秒，视情况而定。这就是给RS485芯片预留切换时间



PC收到 0x01 0x02 0x03

再将一帧数据返还给PC

```

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

```

Table 39-5. Recommended Settings for Typical Crystals and Baud Rates<sup>(1)</sup> (continued)

BRCLK	Baud Rate	UCOS16	UCBRx	UCBRFx	UCBRSS <sup>(2)</sup>	TX Error <sup>(3)</sup> (%)		RX Error <sup>(3)</sup> (%)	
						neg	pos	neg	pos
8388608	230400	1	2	4	0x92	-1.62	1.37	-3.56	2.06
8388608	460800	0	18	-	0x11	-2	3.37	-5.31	5.55
12000000	9600	1	78	2	0x0	0	0	0	0.04
12000000	19200	1	39	1	0x0	0	0	0	0.16
12000000	38400	1	19	8	0x65	-0.16	0.16	-0.4	0.24
12000000	57600	1	13	0	0x25	-0.16	0.32	-0.48	0.48
12000000	115200	1	6	8	0x20	-0.48	0.64	-1.04	1.04
12000000	230400	1	3	4	0x2	-0.8	0.96	-1.84	1.84
12000000	460800	1	1	10	0x0	0	1.76	0	3.44
16000000	9600	1	104	2	0xD6	-0.04	0.02	-0.09	0.03
16000000	19200	1	52	1	0x49	-0.08	0.04	-0.1	0.14
16000000	38400	1	26	0	0xB6	-0.08	0.16	-0.28	0.2
16000000	57600	1	17	5	0xDD	-0.16	0.2	-0.3	0.38
16000000	115200	1	8	10	0xF7	-0.32	0.32	-1	0.36
16000000	230400	1	4	5	0x55	-0.8	0.64	-1.12	1.76
16000000	460800	1	2	2	0xBB	-1.44	1.28	-3.92	1.68
16777216	9600	1	109	3	0xB5	-0.03	0.02	-0.05	0.06
16777216	19200	1	54	9	0xEE	-0.06	0.06	-0.11	0.13
16777216	38400	1	27	4	0xFB	-0.11	0.1	-0.33	0
16777216	57600	1	18	3	0x44	-0.16	0.15	-0.2	0.45

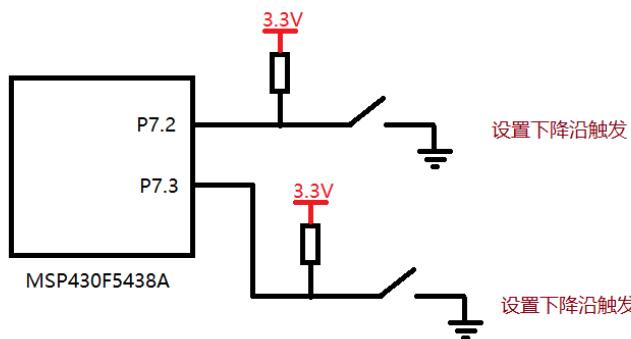
这是波特率误差校准表，可以参考优化串口。

## 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



为了不与P1组和P2组IO口外部中断功能搞混，我选用不具备外部中断功能的P7组IO口来测试IO输入

```
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
```

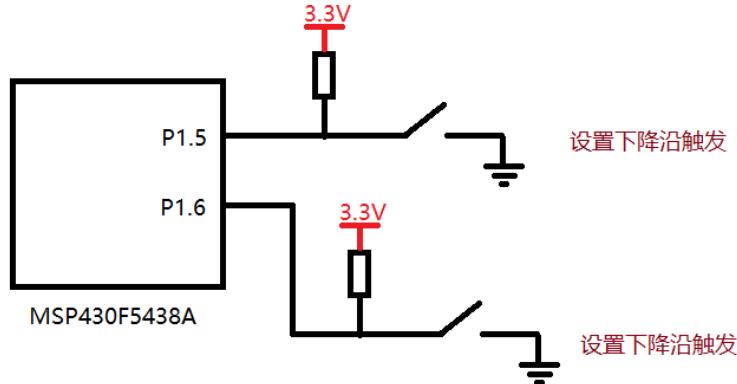
```
#pragma vector = PORT1_VECTOR
__interrupt void P1_IRQ(void)
{
    if((P1IN & BIT6) == 0) //如果 P1.6 == 0
    {
        printf("P1.6 IRQ falling...\n"); //按键按下进入中断, 发现 IO 电平为 0, 执行下降沿程序, 然后记得开上升沿模式, 后面按键松开是上升沿状态

        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。



```

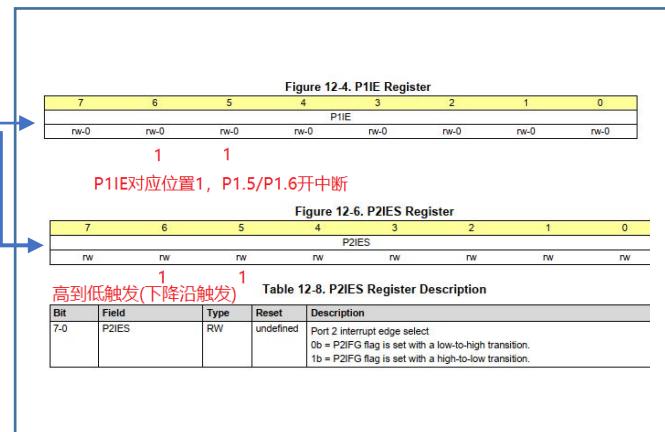
int main( void )
{
    // Stop watchdog timer to prevent time out reset
    WDTCTL = WDTPW + WDTHOLD;
    SysInit();           //时钟初始化，自定义
    usartInit();         //串口初始化，自定义
    P10DIR |=BIT6; //将 P10.6 配置成输出 因为电路有硬件看门狗所以需要喂狗
    P1DIR &= ~BIT5; //P1.5 默认为 0 输入模式
    P1DIR &= ~BIT6; //P1.6 默认为 0 输入模式

    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 CCO */

#define TIMER0_A1_VECTOR      (53 * 2u) /* 0xFFEA Timer0_A5 CC1-4, TA0 */
#define TIMER0_A0_VECTOR      (54 * 2u) /* 0xFFEC Timer0_A5 CCO */

#define TIMER0_B1_VECTOR      (59 * 2u) /* 0xFFFF Timer0_B7 CC1-6, TB */
#define TIMER0_B0_VECTOR      (60 * 2u) /* 0xFFFF8 Timer0_B7 CCO */
```

4. 编写代码时可参考下表

中断矢量名称	中断向量	定时器	含义	操作寄存器
TIMER1_A1_VECTOR	0xFFE0	TA1	TA1CCR1-2, TA1IFG 比较匹配通道1和2，溢出中断	TA1CCTL1, TA1CCR1 TA1CCTL2, TA1CCR2 TA1CTL(溢出中断)
TIMER1_A0_VECTOR	0xFFE2	TA1	TA1CCR0 比较匹配通道0	TA1CCTL0, TA1CCR0
TIMER0_A1_VECTOR	0xFFEA	TA0	TA0CCR1-4, TA0IFG 比较匹配通道1到4 溢出中断	TA0CCTL1, TA0CCR1 TA0CCTL2, TA0CCR2 TA0CCTL3, TA0CCR3 TA0CCTL4, TA0CCR4 TA0CTL(溢出中断)
TIMER0_A0_VECTOR	0xFFEC	TA0	TA0CCR0 比较匹配通道0	TA0CCTL0, TA0CCR0
TIMER0_B1_VECTOR	0xFFFF6	TB	TBCCR1-6, TBIFG 比较匹配通道1到6 溢出中断	
TIMER0_B0_VECTOR	0xFFFF8	TB	TBCCR0 比较匹配通道0	

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

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。

Figure 17-16. TAxCTL Register

15	14	13	12	11	10	9	8
Reserved						TASSEL	
						1	0
7	6	5	4	3	2	1	0
ID		MC		Reserved	TACLR	TAIE	TAIFG
1	1	1	0		1	1	

8分频

连续模式

定时器设置清0 开定时器中断

SMCLK时钟源

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  $1 \approx 32\text{ms} * 30 = 0.98$  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;
        }
    }
}

```

中断矢量名称	中断向量	定时器	含义	操作寄存器
TIMER1_A1_VECTOR	0xFFE0	TA1	TA1CCR1-2, TA1IFG 比较匹配通道1和2, 溢出中断	TA1CCTL1,TA1CCR1 TA1CCTL2,TA1CCR2 TA1CTL(溢出中断)

## 定时器输入捕获比较，频率周期采集

P8.0/TA0.0	57	60	I/O	General-purpose digital I/O Timer0_A5 CCR0 capture: CCI0B input, compare: Out0 output
P8.1/TA0.1	58	61	I/O	General-purpose digital I/O Timer0_A5 CCR1 capture: CCI1B input, compare: Out1 output
P8.2/TA0.2	59	62	I/O	General-purpose digital I/O Timer0_A5 CCR2 capture: CCI2B input, compare: Out2 output
P8.3/TA0.3	60	63	I/O	General-purpose digital I/O Timer0_A5 CCR3 capture: CCI3B input, compare: Out3 output
P8.4/TA0.4	61	64	I/O	General-purpose digital I/O Timer0_A5 CCR4 capture: CCI4B input, compare: Out4 output

TA0.0 意思是 TIMERO，也就是 TAO 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 TAO 定时器，CCR1 捕获比较通道

TA0.2 意思就是 TAO 定时器，CCR2 捕获比较通道

我们使用 P8.1/TA0.1 引脚，那么就是 TAO 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。

Port P8 (P8.0 to P8.7) Pin Functions

PIN NAME (P8.x)	x	FUNCTION	CONTROL BITS/SIGNALS	
			P8DIR.x	P8SEL.x
P8.0/TA0.0	0	P8.0 (I/O)	I: 0; O: 1	0
		Timer0_A5.CCI0B	0	1
		Timer0_A5.TA0	1	1
P8.1/TA0.1	1	P8.1 (I/O)	I: 0; O: 1	0
		Timer0_A5.CCI1B	0	1
		Timer0_A5.TA1	1	1
P8.2/TA0.2	2	P8.2 (I/O)	I: 0; O: 1	0
		Timer0_A5.CCI2B	0	1
		Timer0_A5.TA2	1	1
P8.3/TA0.3	3	P8.3 (I/O)	I: 0; O: 1	0
		Timer0_A5.CCI3B	0	1
		Timer0_A5.TA3	1	1
P8.4/TA0.4	4	P8.4 (I/O)	I: 0; O: 1	0
		Timer0_A5.CCI4B	0	1
		Timer0_A5.TA4	1	1
P8.5/TA1.0	5	P8.5 (I/O)	I: 0; O: 1	0
		Timer1_A3.CCI0B	0	1
		Timer1_A3.TA0	1	1
P8.6/TA1.1	6	P8.6 (I/O)	I: 0; O: 1	0
		Timer1_A3.CCI1B	0	1
		Timer1_A3.TA1	1	1
P8.7	7	P8.7 (I/O)	I: 0; O: 1	0

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 + TACL0 + 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 TAO,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 TAO,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 = TIMERO_A1_VECTOR //TIMERO 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 CCRO
通道需要查看宏， TIMERO_A0_VECTOR 就是 TIMERO CCRO 的通道入口

#pragma vector = TIMERO_A1_VECTOR
__interrupt void capture_CCRO_IRQHandler(void)
{

    switch(__even_in_range(TA0IV,14))
    {
        case 0: break;           // No interrupt
        case 2: printf("TIMERO CCR1 ..\n");break;      // CCR1 通道， 输入引脚信号触发中断
        case 4: printf("TIMERO 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,CCIOB,CCRO*/
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; //CCIOB
    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 = TIMERO_A1_VECTOR
__interrupt void capture_CCR0_IRQHandler(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("TIMERO 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 = TIMERO_A1_VECTOR
__interrupt void capture_CCR0_IRQHandler(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)

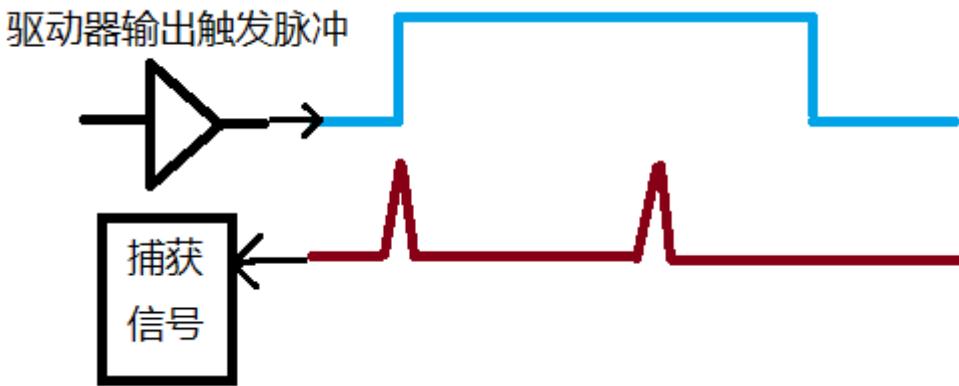
```

```

        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("TIMERO 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; //CCIOB
    // 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 = TIMERO_A1_VECTOR
__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
            }
            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("TIMERO 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;
    }
}

```

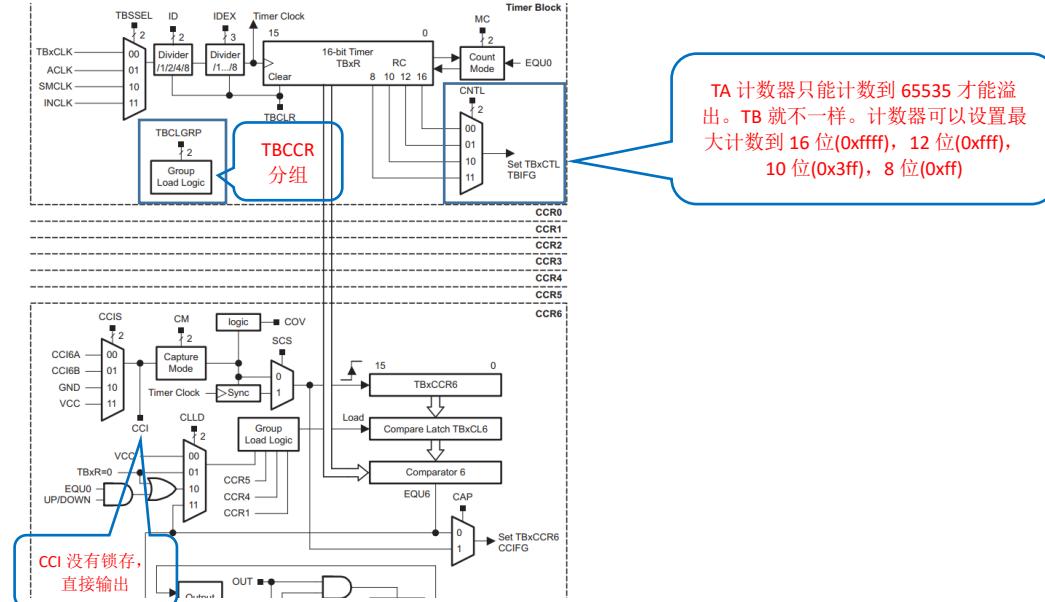
## 定时器 B 使用

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



```

*****
* TBO 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 TBO 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, TBO 定时器也是有两个中断入口的

```

*****
* TBO 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)
{
    /*TBO 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

Figure 28-13. ADC12CTL0 Register

15	14	13	12	11	10	9	8
ADC12SHT1x						ADC12SHT0x	
			1				1
7	6	5	4	3	2	1	0
ADC12MSC	ADC12REF2_5V	ADC12REFON	ADC12ON	ADC12OVIE	ADC12TOVIE	ADC12ENC	ADC12SC
1 ADC 打开							
当 ADC12ENC = 0 时，才能配置 CTL 寄存器							

ADC12MEM0~ADC12MEM7采样保持时间

采样保持时间  
8个ADCCLK

Figure 28-14. ADC12CTL1 Register

15	14	13	12	11	10	9	8
ADC12CSTARTADDx						ADC12SHPx	ADC12SSH
ADC12CSTARTADDx 默认为0						1	
7	6	5	4	3	2	1	0
ADC12DIVx		ADC12SSELx		ADC12CONSEQx		ADC12BUSY	
Can be modified only when ADC12ENC = 0							

采样脉冲信号来自于  
采样定时器

ADC12CSTARTADDx 选择很关键，如果默认为0，那么单通道ADC采集，不管使用哪个通道的ADC，采集出

来的值都在ADC12MEM0

同时ADC12IFG 也只能判断BIT0来确定ADC通道是否采集完成。记住在不设置ADC12CSTARTADDx情况下，使用任何通道都是用 BIT0来确定

Figure 28-17. ADC12MCTLx Register

7	6	5	4	3	2	1	0
ADC12EOS		ADC12SREFx		ADC12INCHx			
0	0	1	0	0	0	0	1

Can be modified only when ADC12ENC = 0

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

选择VREF+ and  
AVSS 就是ADC VR+  
接外部基准源+。  
ADC VR- 接GND

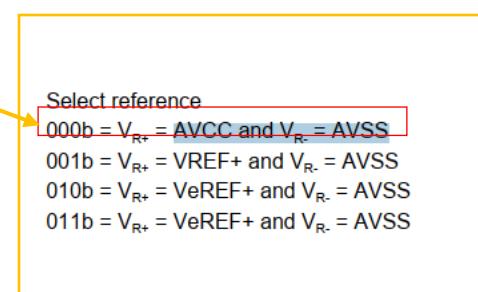
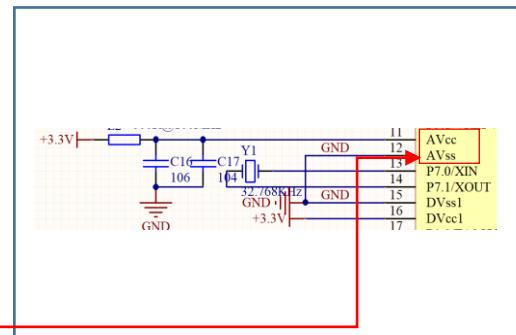
采集外部通道1  
也就是P6.1/A1

```
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 = ADC12SHPx;
    // 关闭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();
uartInit(); //串口初始化
ADC_A1_Init(); //开总中断
_EINT();

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();

    uartInit(); //串口初始化
    ADC_A1_Init(); //开总中断
    _EINT();
    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
    }
}

```

P5.0/VREF+/VeREF+	<input type="checkbox"/>	9
P5.1/VREF-/VeREF-	<input type="checkbox"/>	10
AV <sub>cc</sub>	<input type="checkbox"/>	11
AV <sub>ss</sub>	<input type="checkbox"/>	12

如果选择外接的基准源, 那就是 VeREF  
 $010b = V_{R+} = VeREF+$  and  $V_{R-} = AVSS$   
 那么 ADC12MCTL0 =  
 $ADC12SREF\_2 + ADC12INCH\_1$

```

    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
}
}

```

```

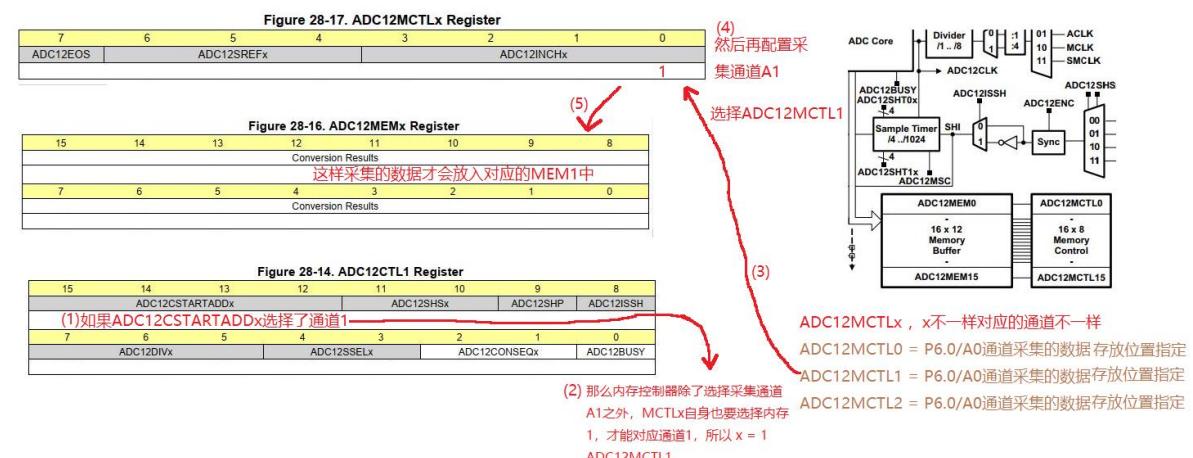
MSP430F5438A TEST ..
ADC Value = 2375
ADC Value = 2437
ADC Value = 2298
ADC Value = 2524
ADC Value = 2562
ADC Value = 2357
ADC Value = 2374

```

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
    }
}
ADC Value = 294
ADC Value = 293
ADC Value = 296
ADC Value = 296
ADC Value = 294
ADC Value = 294
ADC Value = 296

```

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
WhADC12 A1 Value = 297
ile
WhADC12 A1 Value = 297
ile
WhADC12 A1 Value = 294
ile
WhADC12 A1 Value = 297
ile
WhADC12 A1 Value = 297
ile

```

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;                      // Vector 8: ADC12IFG1
                break;
        case 10: break;                               // Vector 10: ADC12IFG2
        case 12: break;                               // Vector 12: ADC12IFG3
        case 14: break;                               // Vector 14: ADC12IFG4
        .....
    }
}

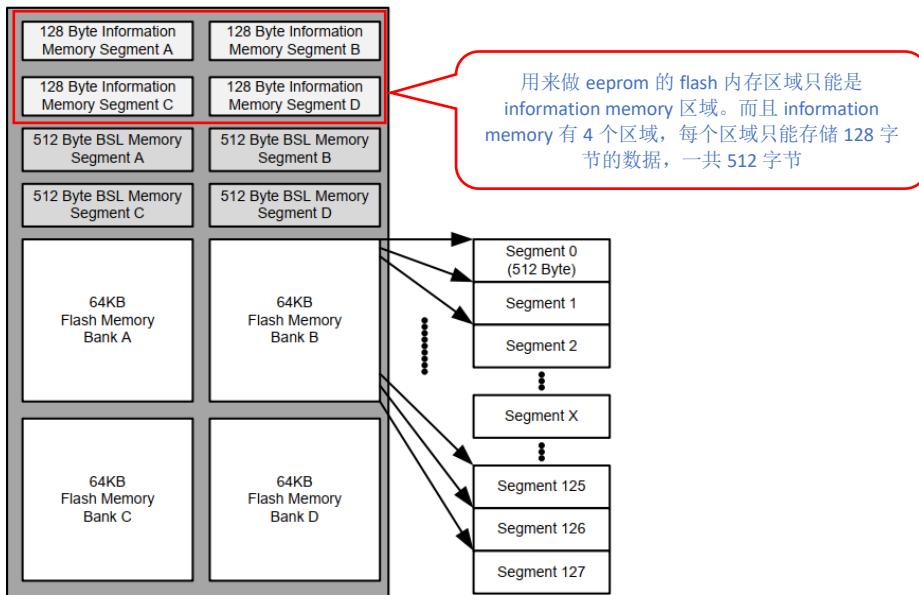
/****************
* 启动 ADC 采集
* 返回 ADC 采集的值
*****************/
unsigned int StartADCgetData(void) //该程序放在定时器任务中断里面处理，出现问题
{
    ADC12CTL0 |= ADC12SC;

    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



### Memory Organization

		MSP430F5419A MSP430F5418A	MSP430F5436A MSP430F5435A	MSP430F5438A MSP430F5437A
<b>Memory (flash)</b> Main: interrupt vector Main: code memory	Total Size Flash Flash	128 KB 00FFFFh–00FF80h 025BFh–005C00h	192 KB 00FFFFh–00FF80h 035BFh–005C00h	256 KB 00FFFFh–00FF80h 045BFh–005C00h
<b>Main: code memory</b>	Bank D	N/A	23 KB 035BFh–030000h	64 KB 03FFFh–030000h
	Bank C	23 KB 025BFh–020000h	64 KB 02FFFh–020000h	64 KB 02FFFh–020000h
	Bank B	64 KB 01FFFh–010000h	64 KB 01FFFh–010000h	64 KB 01FFFh–010000h
	Bank A	41 KB 00FFFh–005C00h	41 KB 00FFFh–005C00h	64 KB 045BFh–040000h 00FFFh–005C00h
<b>RAM</b>	Size	16 KB	16 KB	16 KB
	Sector 3	4 KB 005BFh–004C00h	4 KB 005BFh–004C00h	4 KB 005BFh–004C00h
	Sector 2	4 KB 004BFh–003C00h	4 KB 004BFh–003C00h	4 KB 004BFh–003C00h
	Sector 1	4 KB 003BFh–002C00h	4 KB 003BFh–002C00h	4 KB 003BFh–002C00h
	Sector 0	4 KB 002BFh–001C00h	4 KB 002BFh–001C00h	4 KB 002BFh–001C00h
<b>Information memory (flash)</b>	Info A	128 B 0019FFh–001980h	128 B 0019FFh–001980h	128 B 0019FFh–001980h
	Info B	128 B 00197Fh–001900h	128 B 00197Fh–001900h	128 B 00197Fh–001900h
	Info C	128 B 0018FFh–001880h	128 B 0018FFh–001880h	128 B 0018FFh–001880h
	Info D	128 B 00187Fh–001800h	128 B 00187Fh–001800h	128 B 00187Fh–001800h
<b>Bootstrap loader (BSL) memory (Flash)</b>	BSL 3	512 B 0017FFh–001600h	512 B 0017FFh–001600h	512 B 0017FFh–001600h
	BSL 2	512 B 0015FFh–001400h	512 B 0015FFh–001400h	512 B 0015FFh–001400h
	BSL 1	512 B 0013FFh–001200h	512 B 0013FFh–001200h	512 B 0013FFh–001200h
	BSL 0	512 B 0011FFh–001000h	512 B 0011FFh–001000h	512 B 0011FFh–001000h
<b>Peripherals</b>	Size	4KB 000FFh–000000h	4KB 000FFh–000000h	4KB 000FFh–000000h

MSP430F5438A 就只有这块区域的地址可  
以做 eeprom

因为 ABCD 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

uartInit();      //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
        delay_ms(10);
        P10OUT &= ~BIT6;// P10.6 输出低电平 因为电路有硬件看门狗

        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

    uartInit();      //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
            delay_ms(10);
            P10OUT &= ~BIT6;// P10.6 输出低电平 因为电路有硬件看门狗

            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");
    }

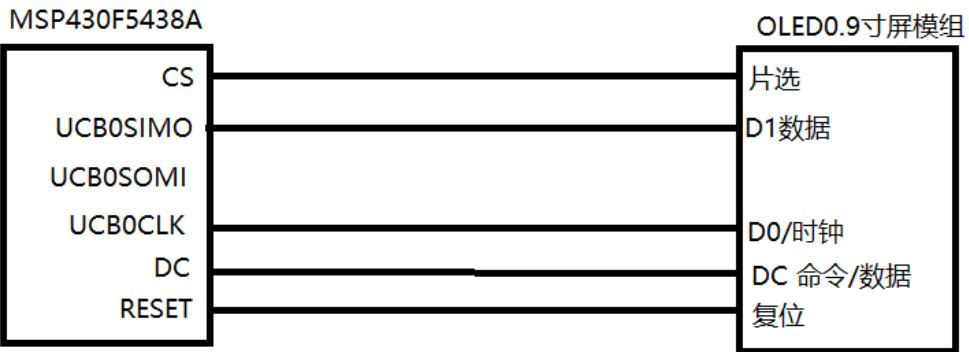
    while(1)
    {
        for(i = 0; i < 2000; i++)
        {
            P10OUT |= BIT6; // P10.6 输出高电平 因为电路有硬件看门狗所以需要喂狗
            delay_ms(10);
            P10OUT &= ~BIT6;// P10.6 输出低电平 因为电路有硬件看门狗所以需要喂狗

            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。

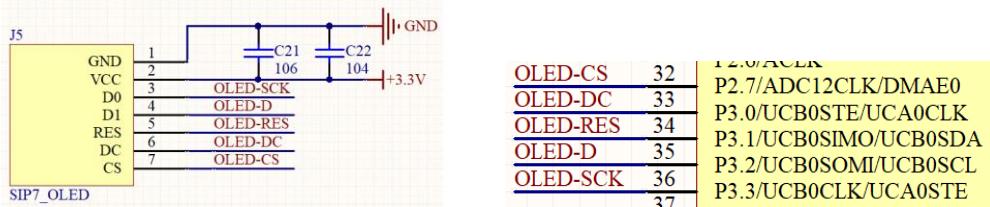
## 模拟 SPI 操作 OLED 屏



MSP430F系列采用  
UCB控制器做SPI串  
行总线

OLED只有SPI输入，没有输出

因为我是模拟 SPI，所以 IO 口接法不是按照标准 UCB SPI 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。

Table 8-4 : Control pins of 4-wire Serial interface

Function	E	R/W#	CS#	D/C#
Write command	Tie LOW	Tie LOW	L	L
Write data	Tie LOW	Tie LOW	L	H

向 OLED 写命令时，  
CS=0,DC=0

向 OLED 写数据时，  
CS=0,DC=1

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

Figure 8-13 : GDDRAM pages structure of SSD1306

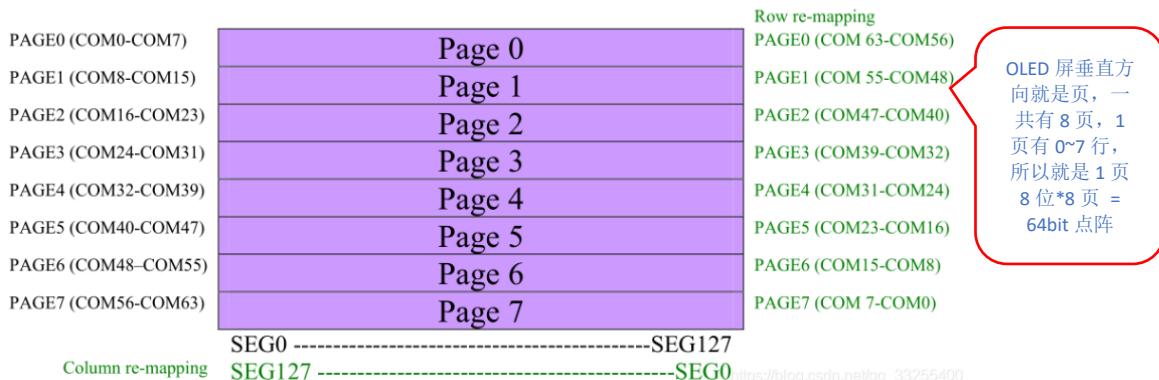
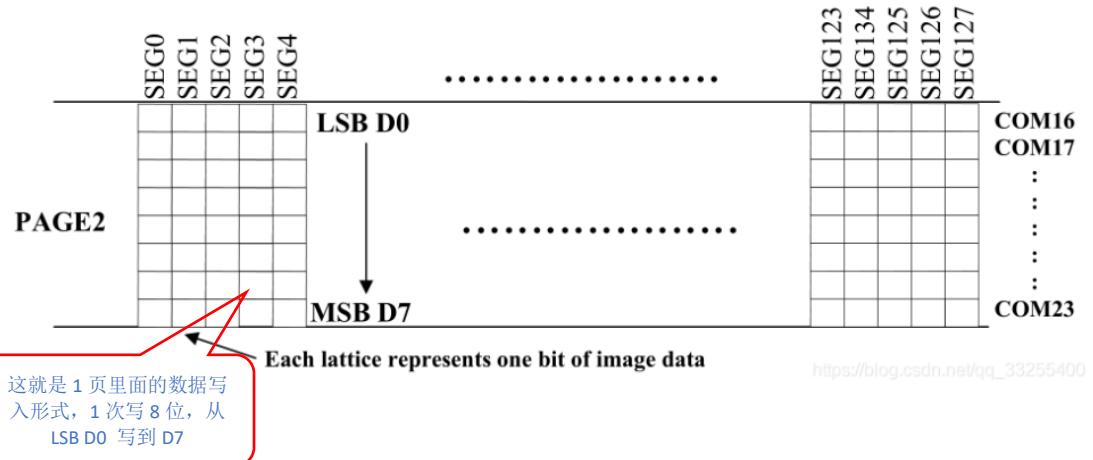


Figure 8-14 : Enlargement of GDDRAM (No row re-mapping and column-remapping)

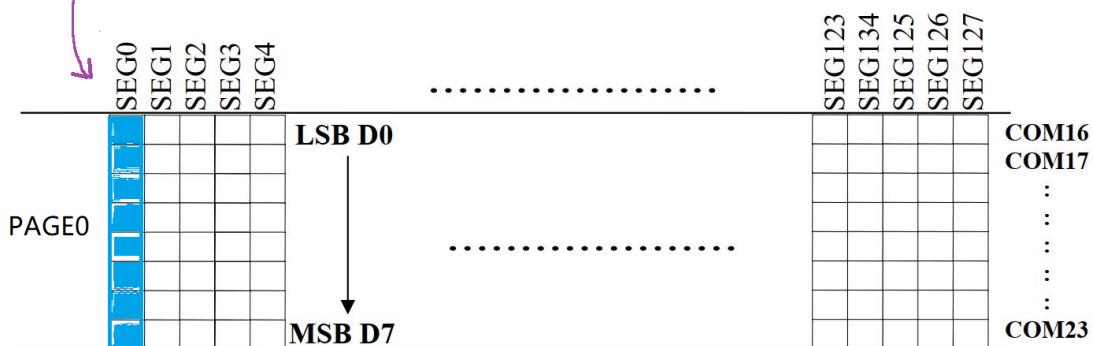


### 1. 通过命令 B0h 到 B7h 选择显示页面的起始地址

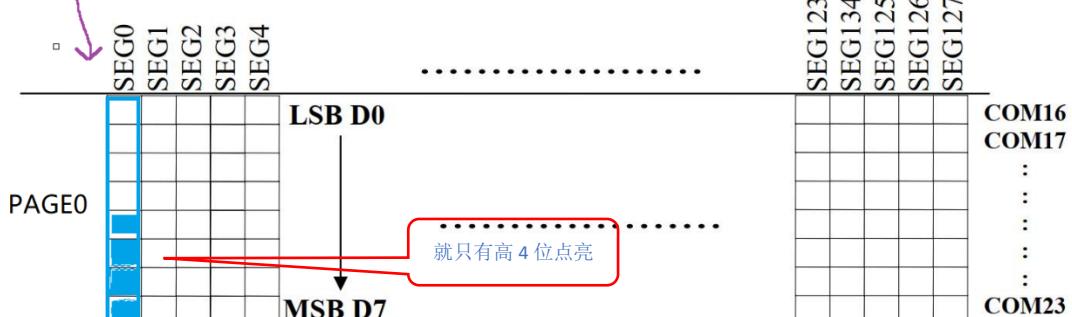
```
OLED_WR_Byte(0xb0+0,0); //表示选择y方向选择第0页, 命令模式写入
OLED_WR_Byte(((0 & 0xf0)>>4)|0x10, 0); //表示列方向从0开始, 也就是SEG0开始, 命令模式写入
OLED_WR_Byte((0 & 0x0f)|0x01, 0); //表示列方向从0开始, 也就是SEG0开始, 命令模式写入
```

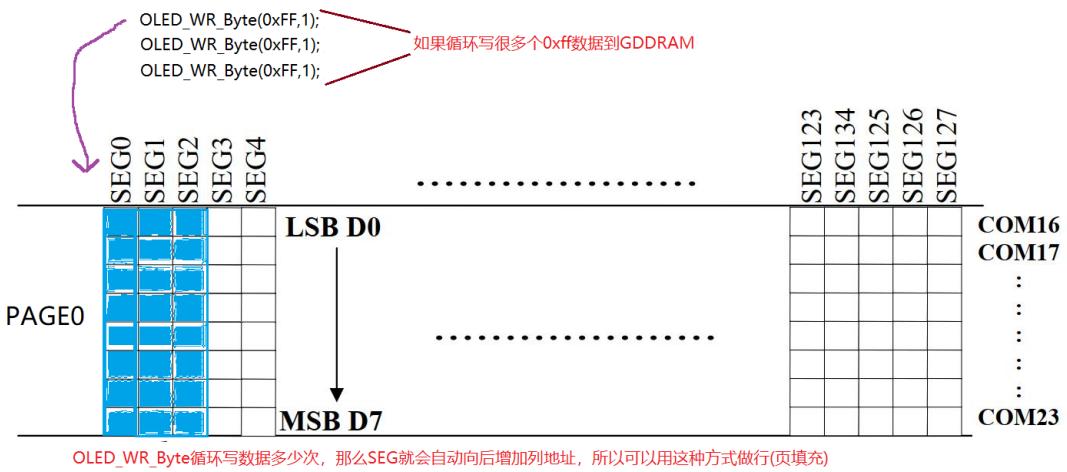
以上坐标其实位置设置完之后, 马上写数据

```
OLED_WR_Byte(0xFF,1); //向第0页, 第0列写数据0xff, 数据模式写入
```

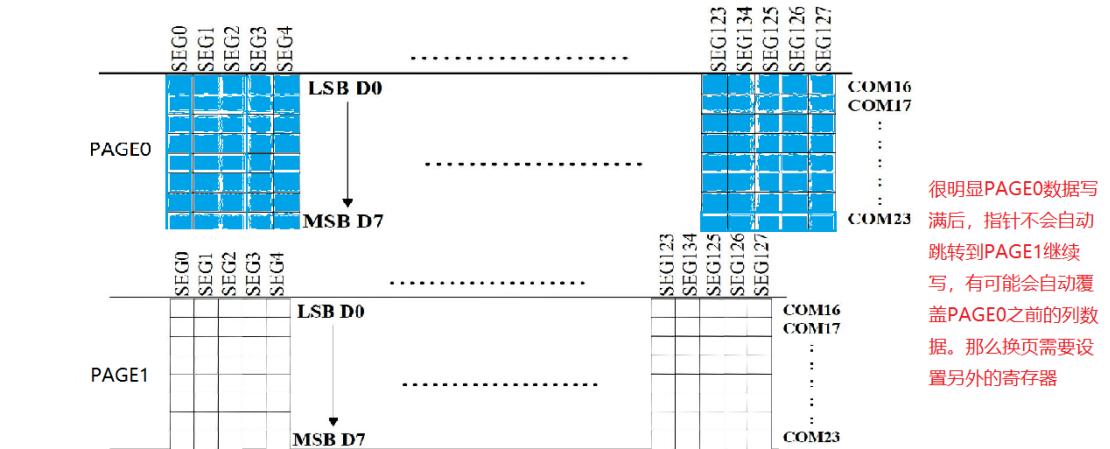


OLED\_WR\_Byte(0xF0,1); 如果改成写F0





`for(int i = 0; i<无穷大, i++)  
 OLED_WR_Byte(0xFF,1);` 这种无限循环写数据的方式, PAGE0  
写满之后会自动换到PAGE1继续写吗?

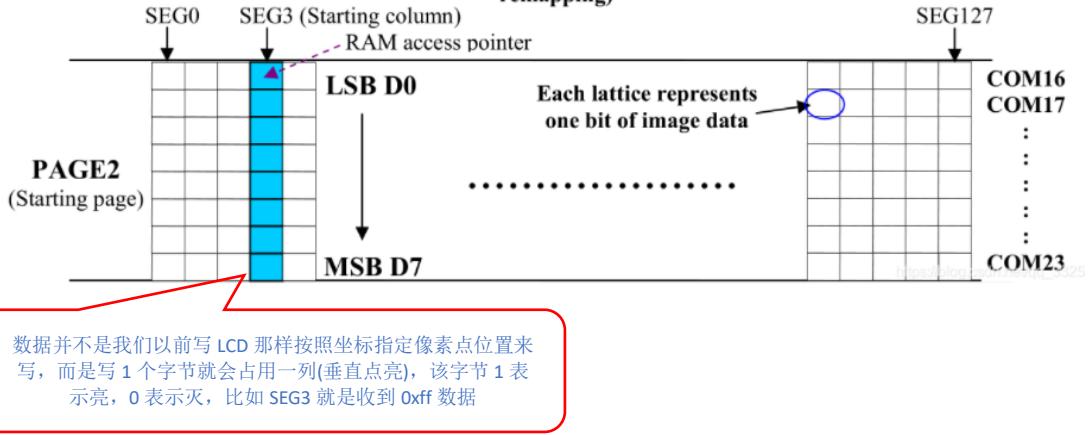


2. 通过命令 00h~0Fh 设置指针的下起始列地址
3. 通过 10h~1Fh 命令设置指针的上起始列地址

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

Figure 10-2 : Example of GDDRAM access pointer setting in Page Addressing Mode (No row and column remapping)



## 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); //设置显示位置一列低地址
        OLED_WR_Byte (0x10,OLED_CMD); //设置显示位置一列高地址
        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(0x8A,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*/
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},/*"!",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*/
{0xF,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, 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, 0xA0, 0x60, 0x00,, ,
0x00, 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
字符串显示使用方式 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 rdbuf[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 rdbuf 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

```

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.timeout++;
        if(modbus.timeout >=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()
{
    uart0Init(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.timeout = 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;
}
}

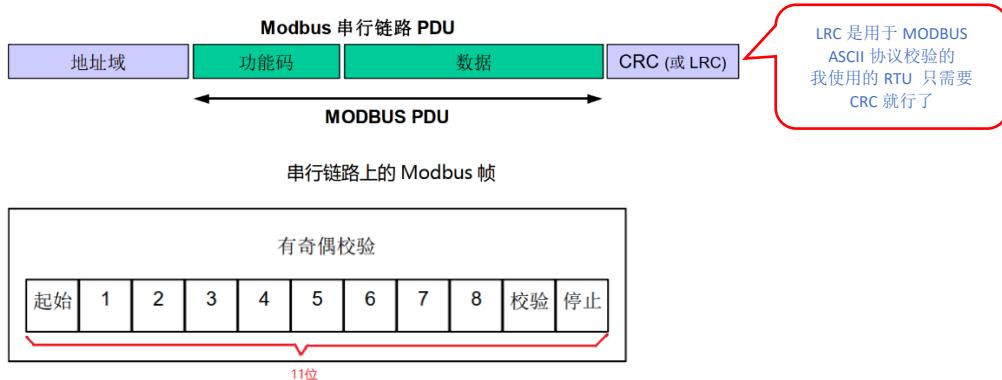
```

### 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：

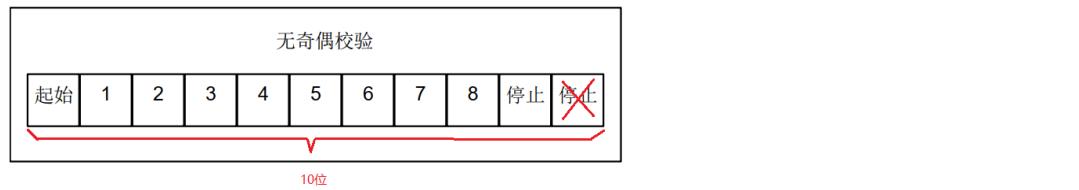
1. 主机发送设备没有定义的寄存器地址 → 设备没有这个地址 → 设备返回寄存器地址错误
2. 主机发送数据给设备 → 设备检测 CRC 校验 → 如果 CRC 校验错误，设备返回给主机错误信息
3. 主机发送功能码给设备 → 设备没有实现该功能码 → 设备返回功能码错误给主机

为了解决以上内容，我们必须熟悉 MODBUS 协议的细节

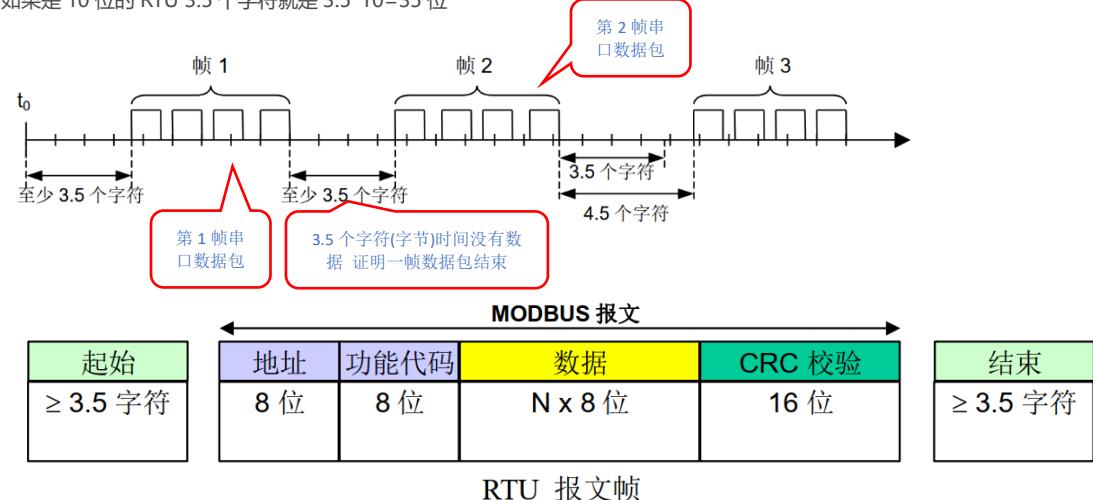


带奇偶校验的 RTU 字节(串口字节)为 11 位

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  $3.5 \times 11 = 38.5$  位

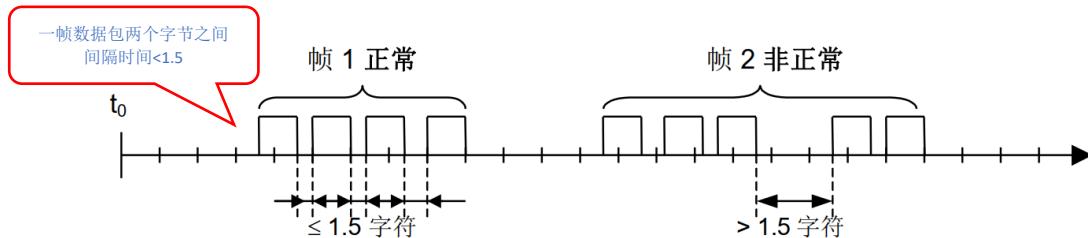


如果是 10 位的 RTU 3.5 个字符就是  $3.5 \times 10 = 35$  位



整个报文帧必须以连续的字符流发送。

如果两个字符之间的空闲间隔大于 1.5 个字符时间，则报文帧被认为不完整应该被接收节点丢弃。



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

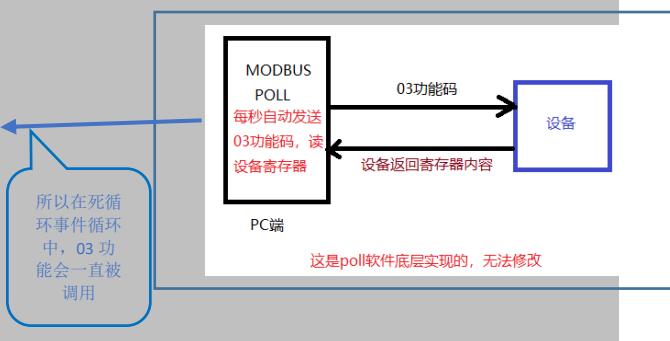
在波特率为 9600 的情况下，只要大于 4.0104167 毫秒即可！

### 功能码 03 和 06 解析

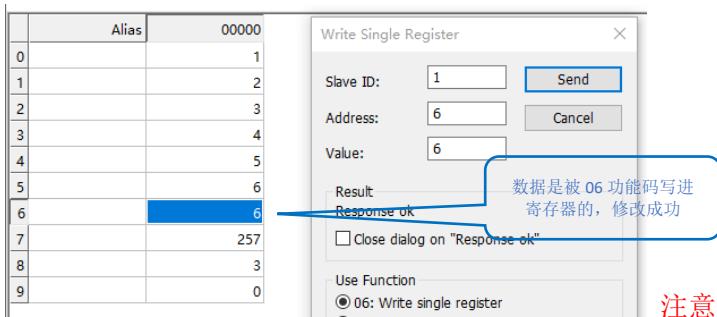
```
//收到数据包
//通过读到的数据帧计算CRC
crc = Modbus_CRC16(&modbus.rcbuf[0],modbus.recount-2);
// 读取数据帧的CRC
rccrc = modbus.rcbuf[modbus.recount-2]*256+modbus.rcbuf[modbus.recount-1];
if(crc == rccrc) //CRC检验成功 开始分析包
{
    if(modbus.rcbuf[0] == modbus.myadd) // 检查地址是否时自己的地址
    {
        switch(modbus.rcbuf[1]) //分析modbus功能码
        {
            case 0:break;
            case 1:break;
            case 2:break;
            case 3:
                Modbus_Func3();
                break;

            case 4:break;
            case 5:break;
            case 6:
                Modbus_Func6();
                break;

            case 7:break;
        }
    }
}
```



The screenshot shows two windows of the Mbpoll2 software. The left window displays a table of registers with aliases and values. A callout box points to a specific entry at address 8 with value 257, stating: '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，因为在用 03 功能码不停的循环读取' (If the device's internal program automatically changes register values, this table will automatically update because it is constantly being read by the 03 function code in a loop). The right window shows a 'Write Single Register' dialog with fields for Slave ID (1), Address (6), and Value (6). A note next to it says: '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' (If you modify a register value in the 03 function interface, you will find it is modified according to the 06 function code).



注意 06 功能码只能写单个寄存器

### 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
}
```

Mbpoll1

Tx = 5: Err = 0: ID = 1: F = 03: SR = 1000ms

	Alias	00000
0		1
1		2
2		3
3		4
4		5
5		6
6		8

开机 03 功能码， 默认读取数据是 Reg 数组的值

Mbpoll1

Tx = 12: Err = 0: ID = 1: F = 16: SR = 1000ms

	Alias	00000	Communication Traffic
0		8888	Ext Stop Clear Save Copy Log <input type="checkbox"/> Stop on Error
1		7777	Px:000001-01 10 00 00 00 05 00 0A Tx:000002-01 10 00 00 00 05 00 0A 22 B8 1E 61 1A 0A 15 B3 11 5C 60 A0
2		6666	Rx:000003-01 10 00 00 00 05 00 0A Tx:000004-01 10 00 00 00 05 00 0A 22 B8 1E 61 1A 0A 15 B3 11 5C 60 A0
3		5555	Px:000005-01 10 00 00 00 05 00 0A Tx:000006-01 10 00 00 00 05 00 0A 22 B8 1E 61 1A 0A 15 B3 11 5C 60 A0
4		4444	Rx:000007-01 10 00 00 00 05 00 0A Tx:000008-01 10 00 00 00 05 00 0A 22 B8 1E 61 1A 0A 15 B3 11 5C 60 A0
5			Rx:000009-01 10 00 00 00 05 00 0A
6			
7			

10 功能码，写数据没有问题

Modbus Poll - Mbpoll1

File Edit Connection Setup Functions Display View Window Help

Mbpoll1

Tx = 5: Err = 0: ID = 1: F = 03: SR = 1000ms

	Alias	00000
0		8888
1		7777
2		6666
3		5555
4		4444

03 功能码读取这几个寄存器的值，没问题

Tx = 4: Err = 0: ID = 1: F = 03: SR = 1000ms

	Alias	00000
0		8888
1		7777
2		6666
3		5555
4		4444
5		6
6		-23873
7		257
8		3
9		0

如果 03 功能码读越界

Mbpoll1

Tx = 19: Err = 0: ID = 1: F = 16: SR = 1000ms

	Alias	00000
0		8888
1		7777
2		6666
3		5555
4		4444
5		6
6		-23873
7		257
8		3
9		0

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

Mbpoll1

Tx = 6: Err = 6: ID = 1: F = 03: SR = 1000ms

Timeout error

	Alias	00000
0		0
1		0
2		0
3		0
4		0
5		0
6		0
7		0
8		0
9		0

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];

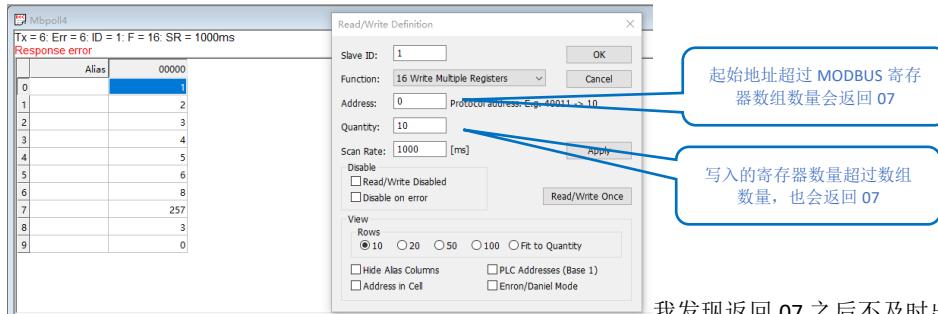
for(i = 0; i < regNum+1;i++)
{
    Count++;
    Reg[startAddr + i] = (modbus.rcbuf[6+Count]<<8); //数据高字节
    Count++;
    Reg[startAddr + i] = (Reg[startAddr + i]) | (modbus.rcbuf[6+Count]); //数据低字节
}

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 功能码报错*/
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。

## RTC 时钟使用

未完待续.....

