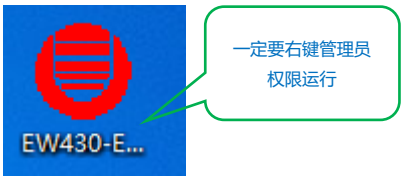
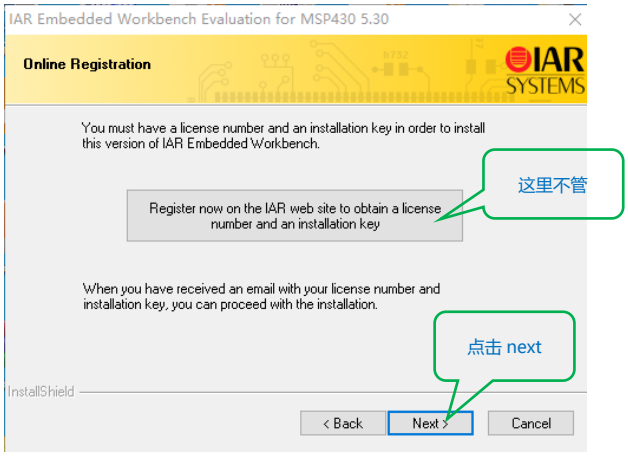
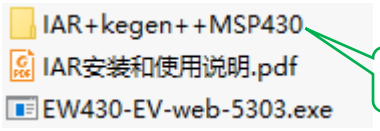
MSP430F149 操作指南

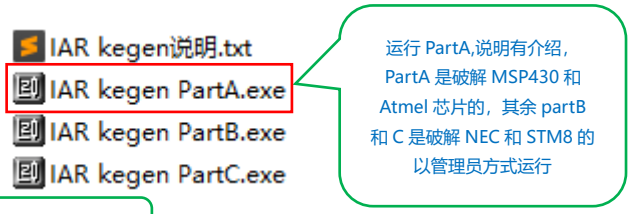
作者:向仔州

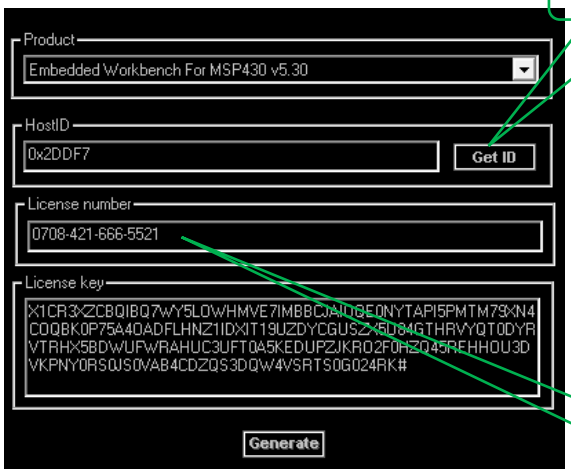
IAR EW430 5.3 版本 IDE 安装

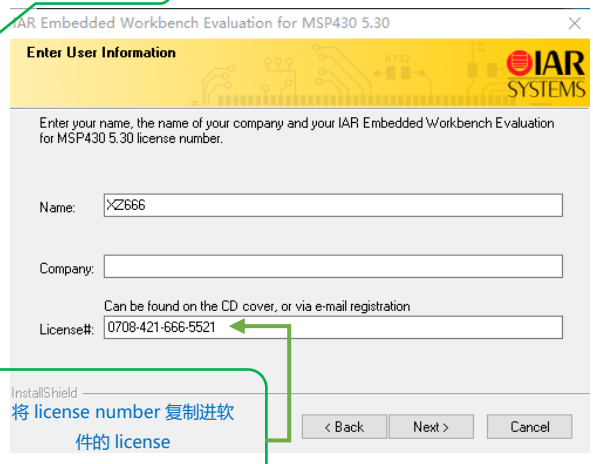




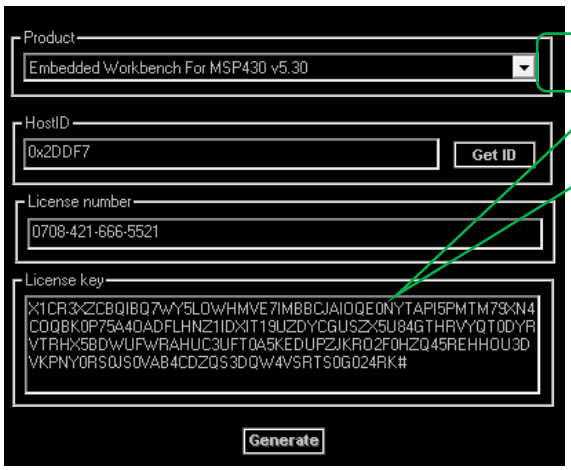




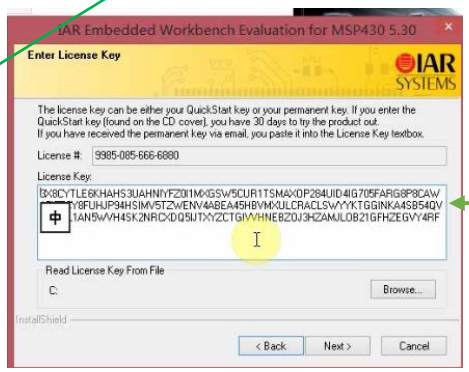


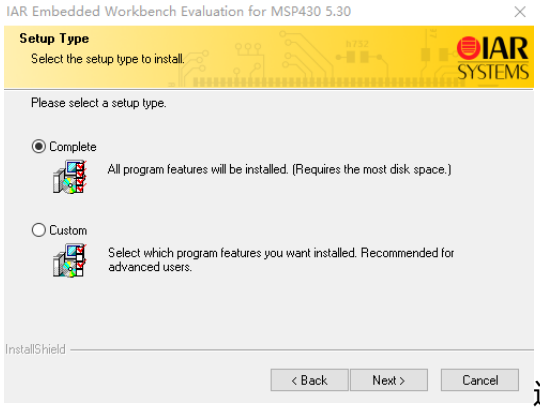


点击 Next

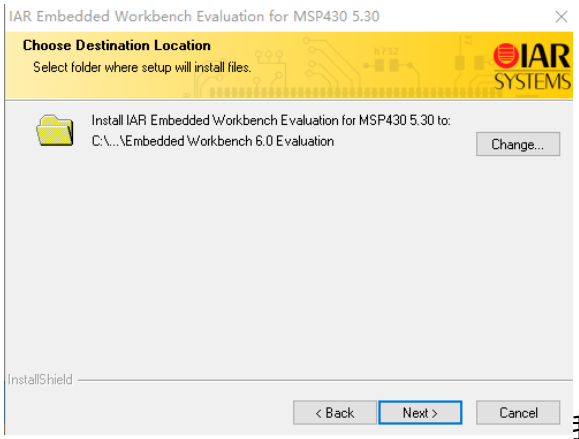


将 license key 复制进软件的 license key

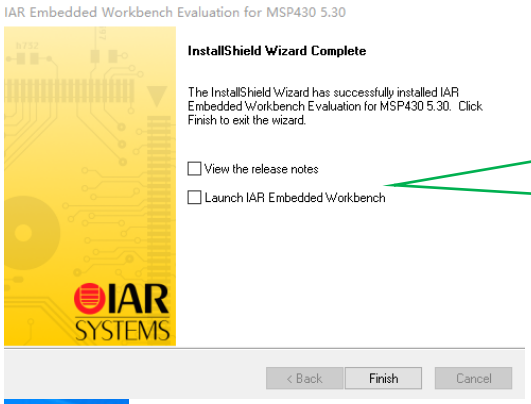




选择 Complete 点击 Next，然后点击 install 安装

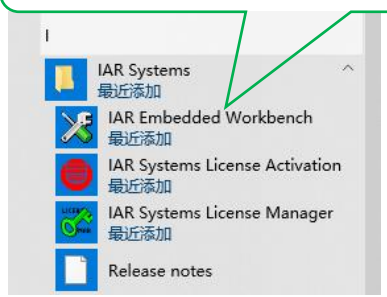


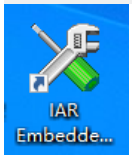
我一般默认安装在 C 盘



安装完成后一定要取消这两个勾选项

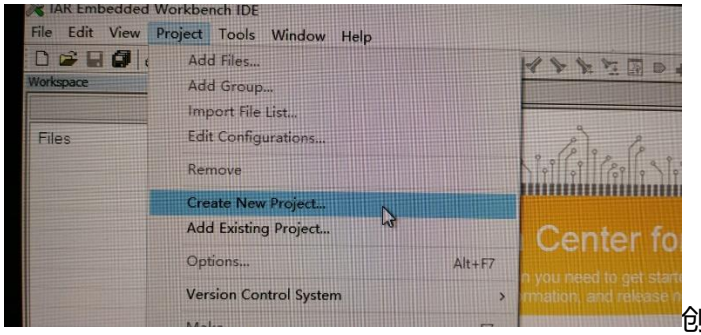
安装完的软件不会在桌面显示，一般都在软件启动菜单里显示，所以拖出快捷方式到桌面



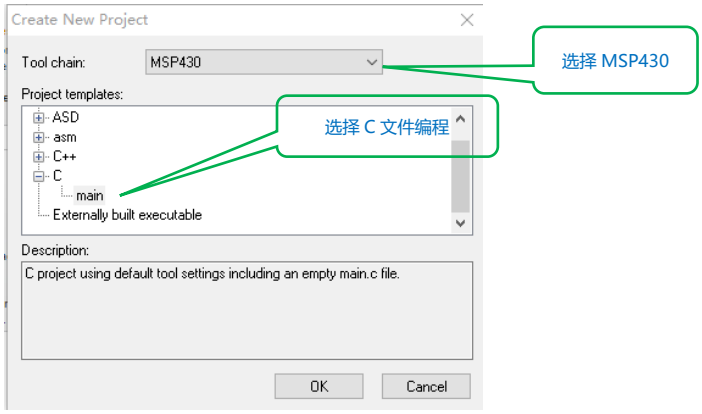


软件安装完成

IAR 创建 MSP430 工程

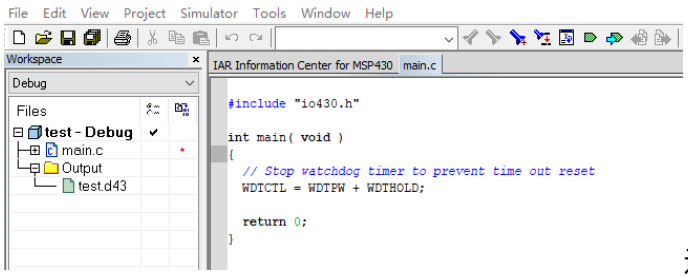


创建项目



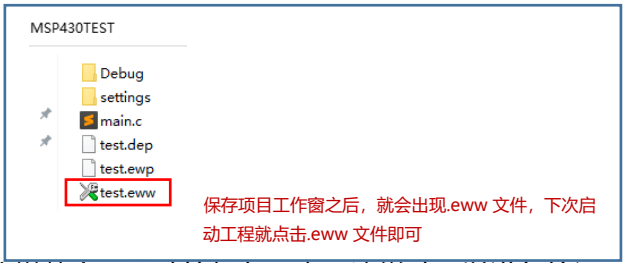
弹出保存工程路径对话框

然后保存



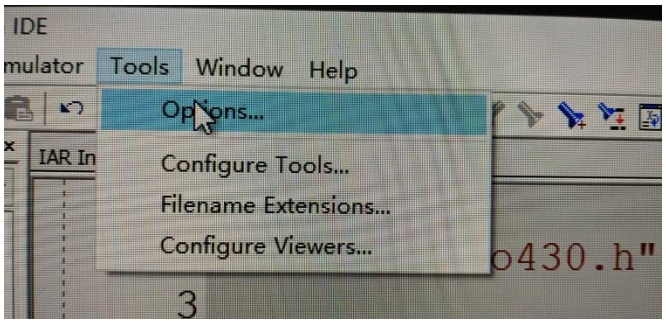
这就是创建的基本项目结构。

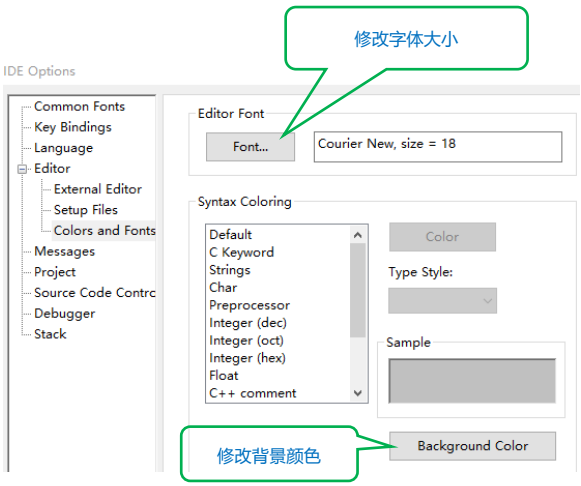
因为我们是先建立的项目，而没有去先建立工作窗，所以需要执行 save All 保存整个项目

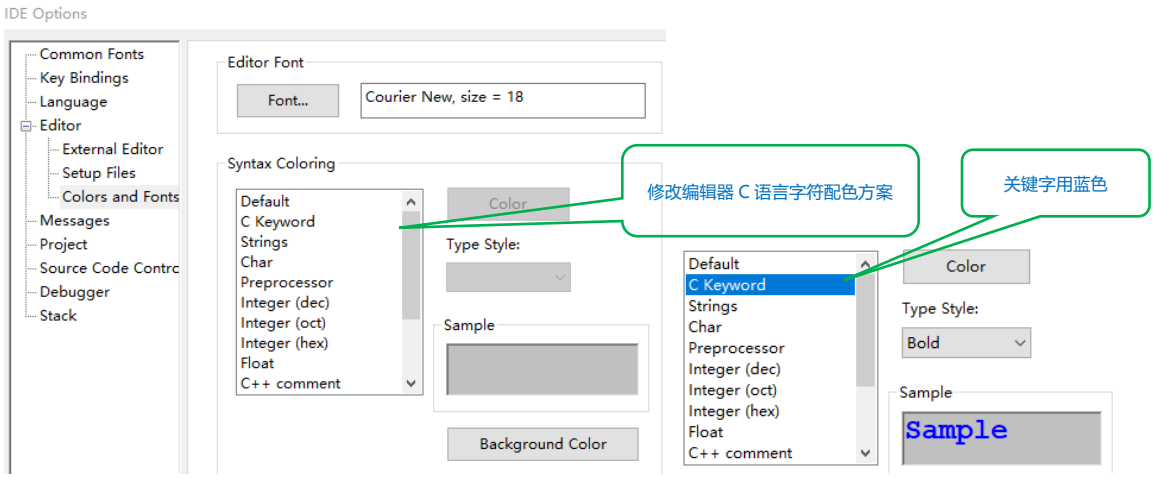


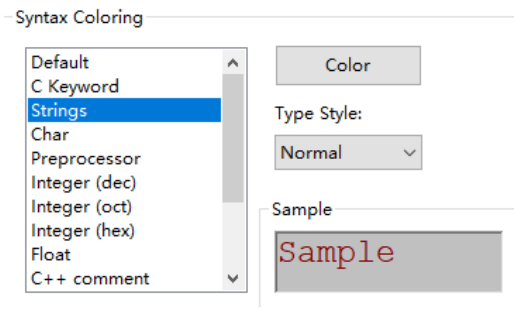
这样整个项目才算保存下来。这样才可以进行编译。

IAR 字体配色方案

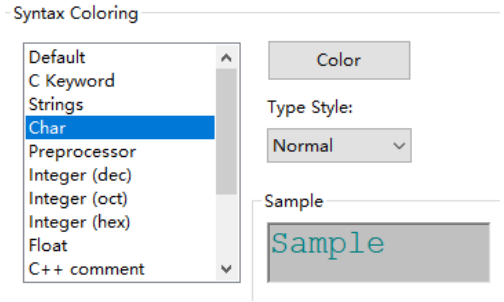




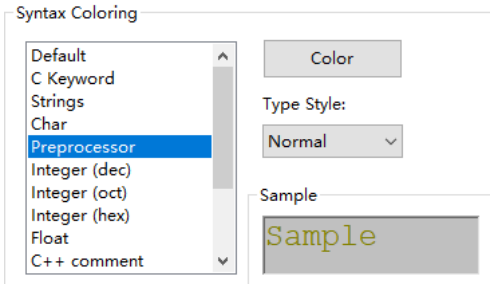




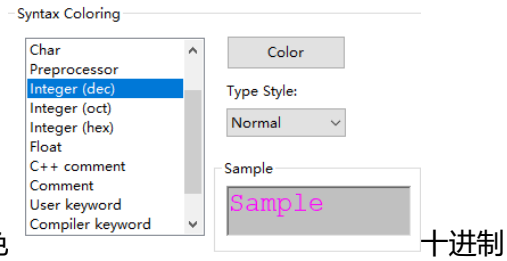
字符串用棕色



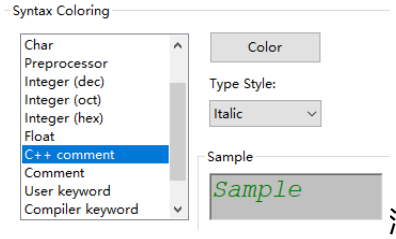
字符浅绿



预编译黄色

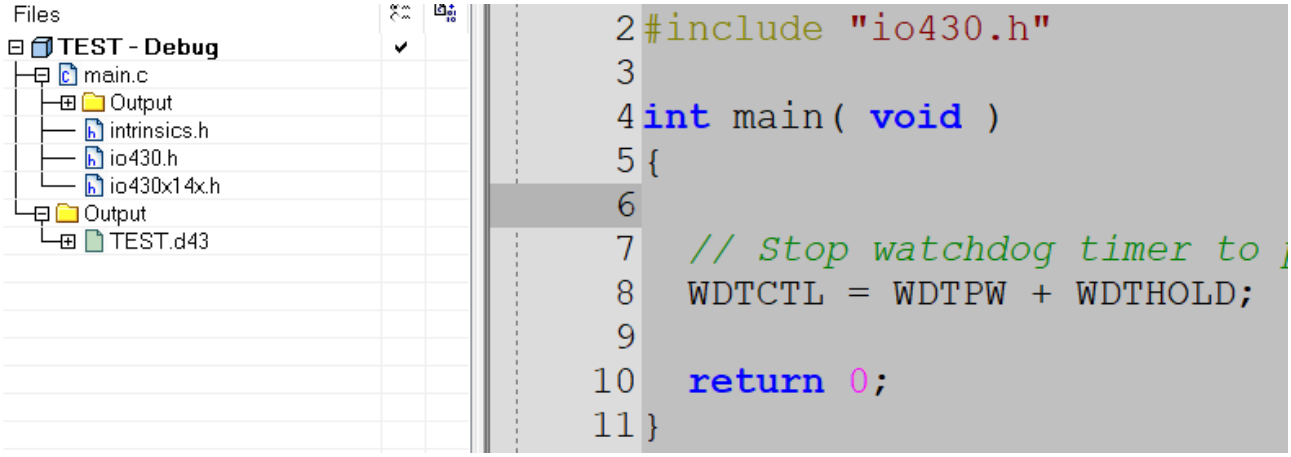


十进制

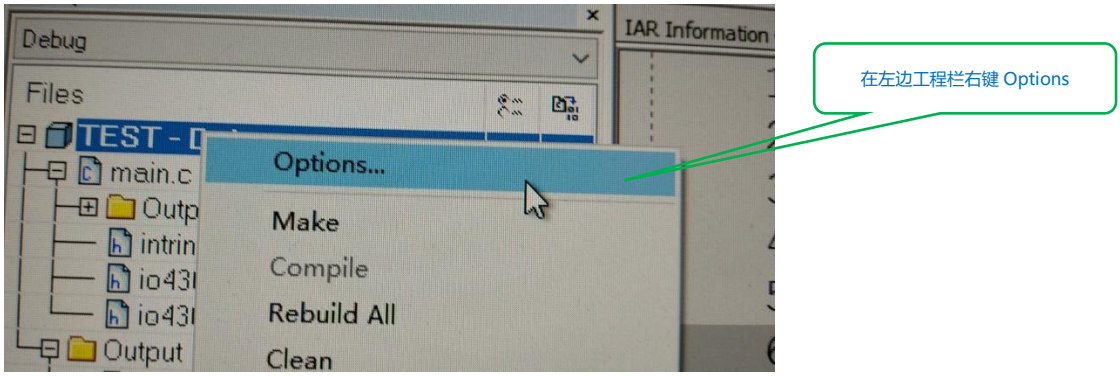


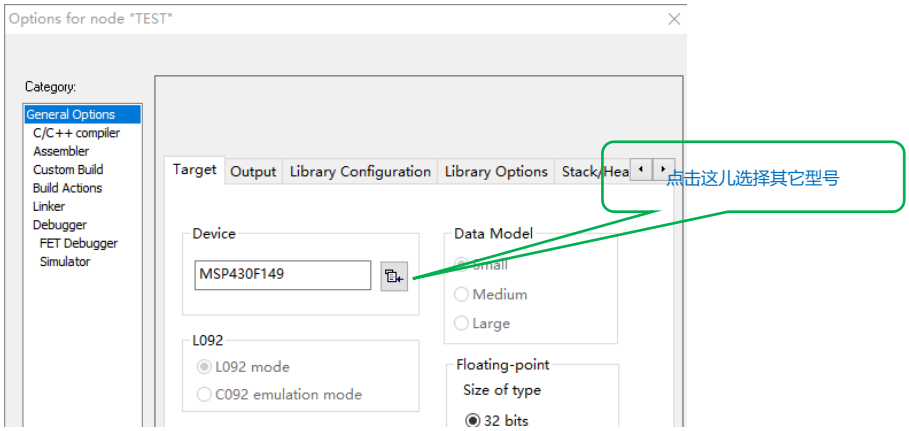
注释用绿色

工程默认是 MSP430F149,如果要用其它型号的芯片，必须在工程里面的 OPTION 上修改

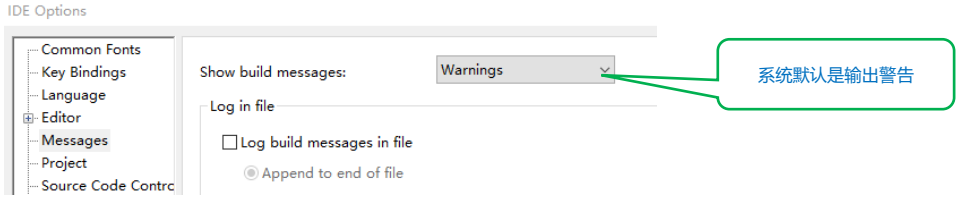


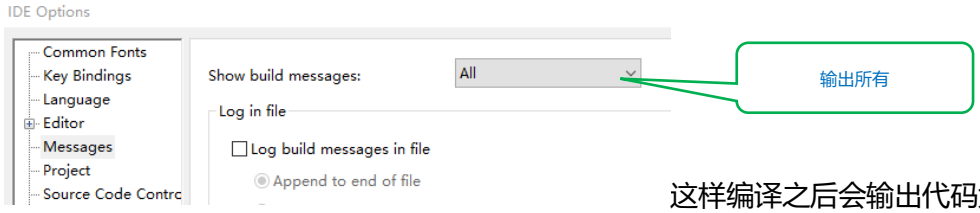
如果要编译其它芯片型号，需要修改工程选项





编译的时候，设置输出代码大小和所有信息

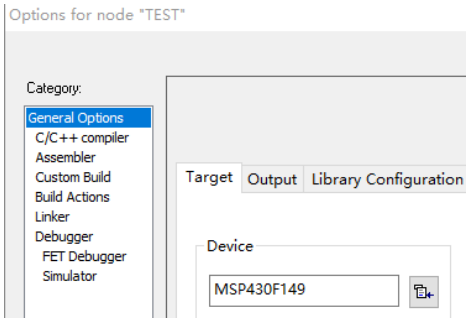


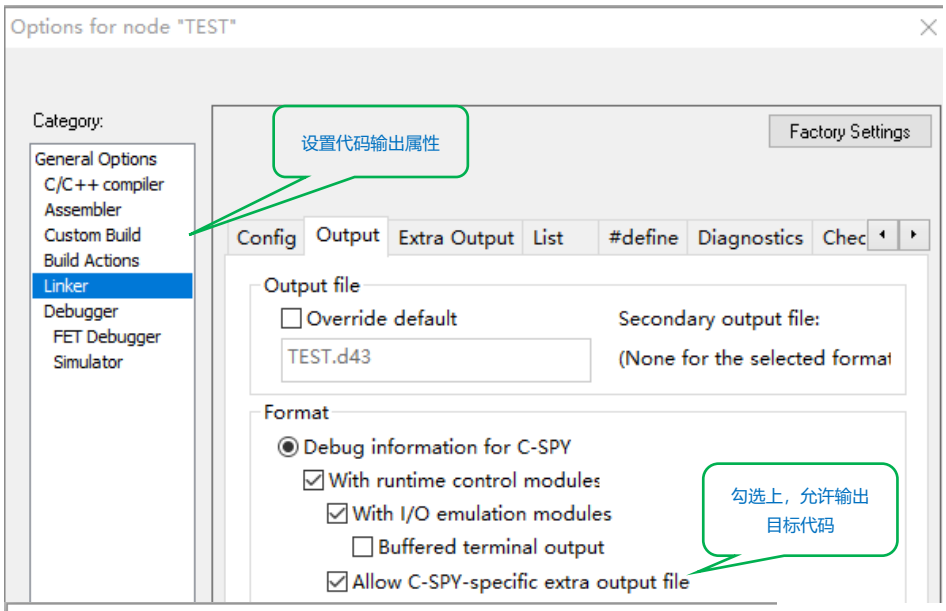


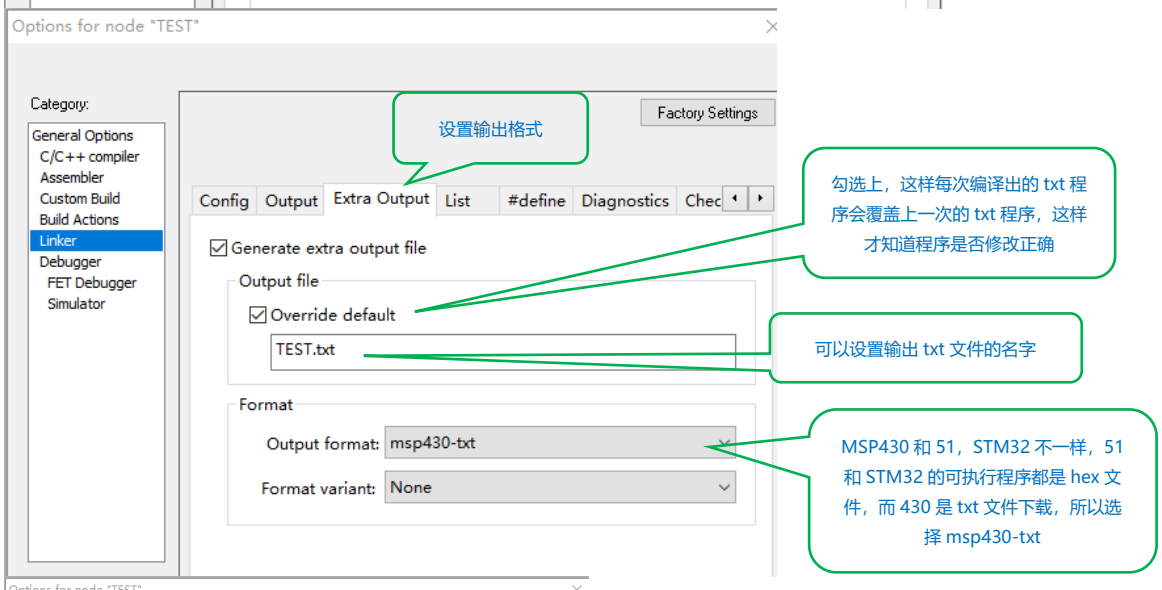
这样编译之后会输出代码大小。

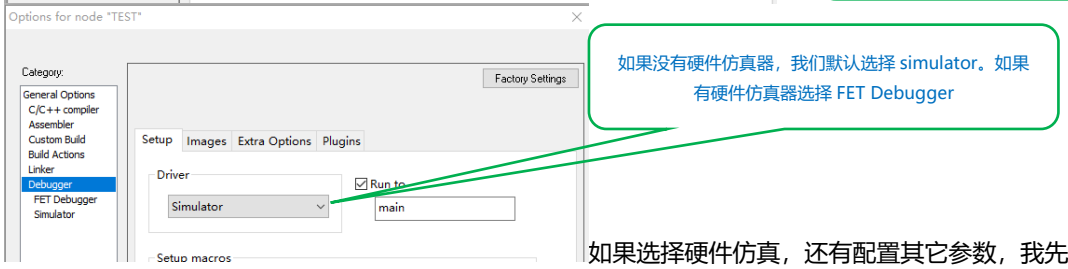
设置 IAR 工程输出程序供芯片下载

选择顶部菜单栏 options



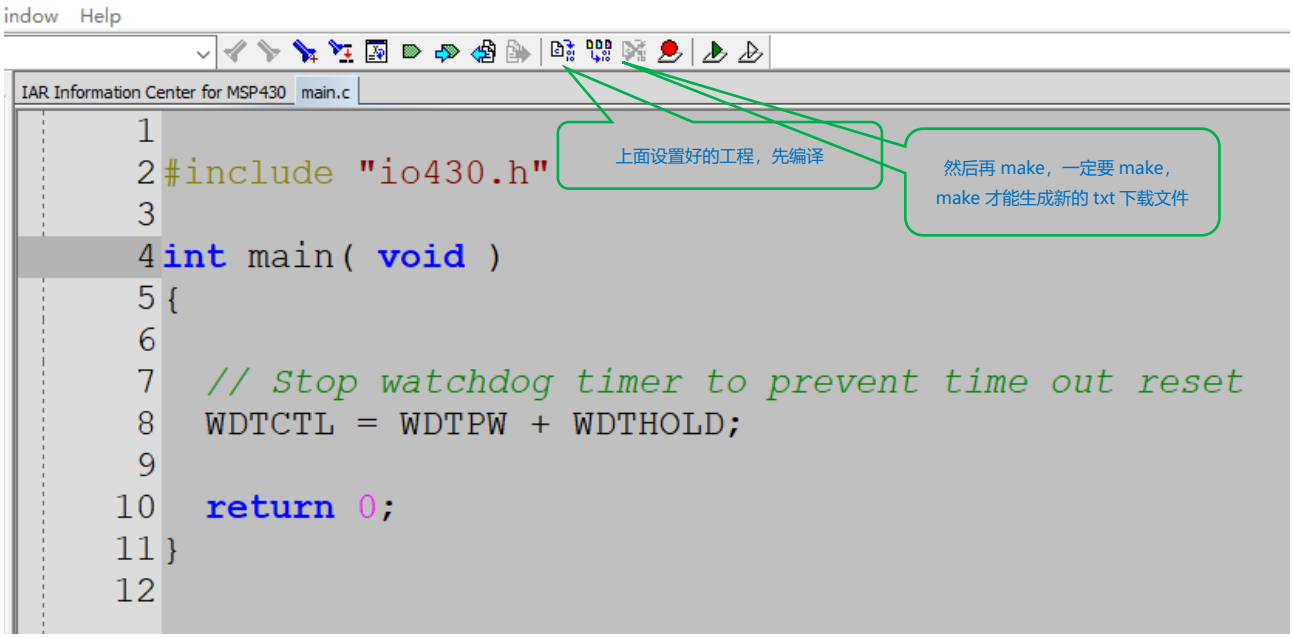


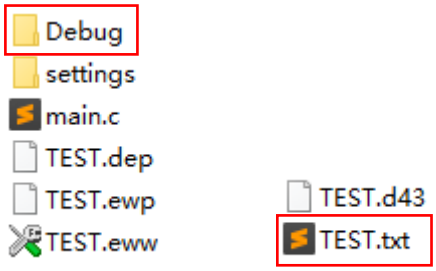




如果选择硬件仿真，还有配置其它参数，我先使用软件仿真

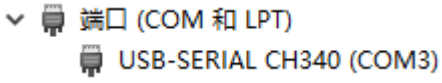
BSL 软件下载 txt 程序进单片机



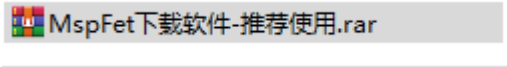


要下载的程序就在 Debug/Exe/TEST.txt 中

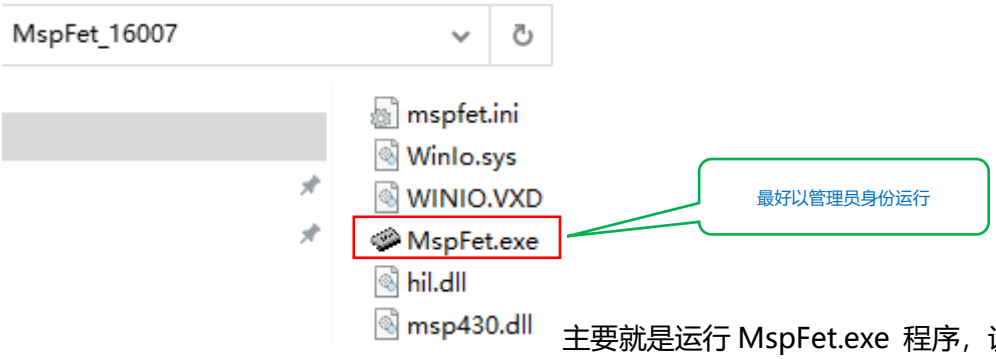
BSL 下载程序是通过串口下载，一定要安装 x64 版本的 CH340,或者 CH341 芯片驱动。



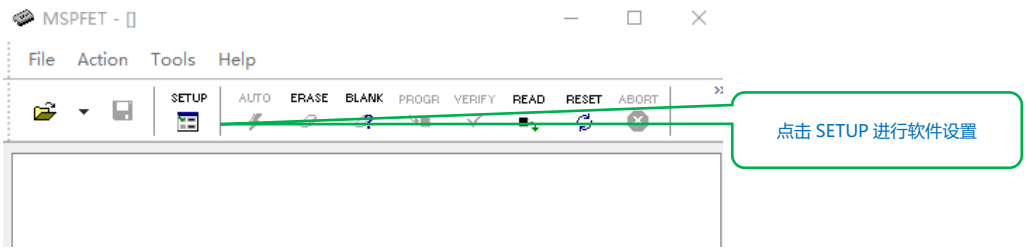
PC 插上开发板有端口显示，我们就是用这个端口下载程序。

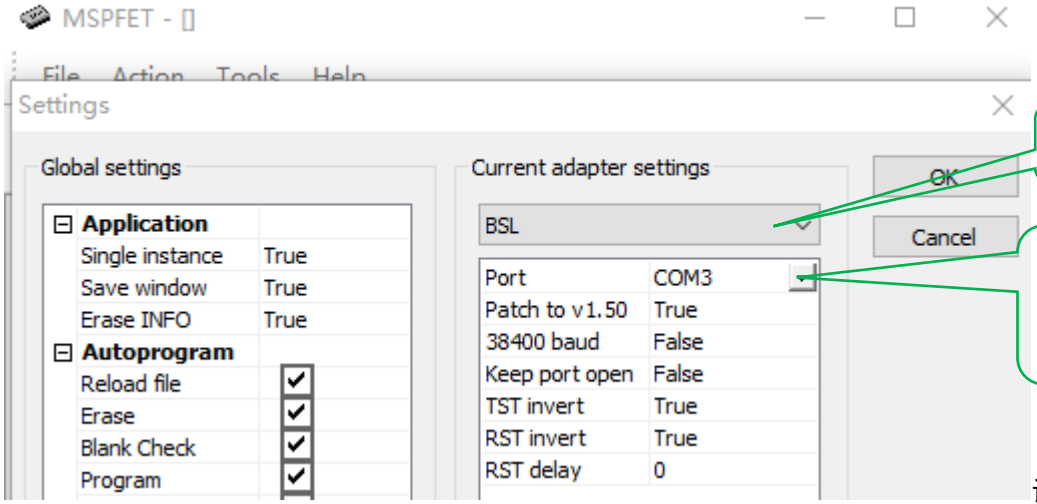


解压 TI 官方的 MspFet 软件下载程序

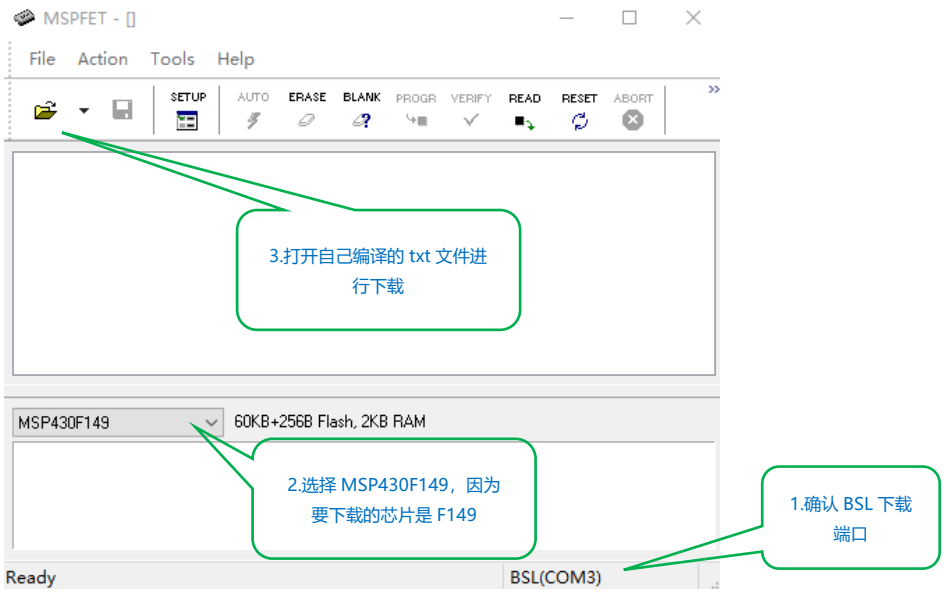


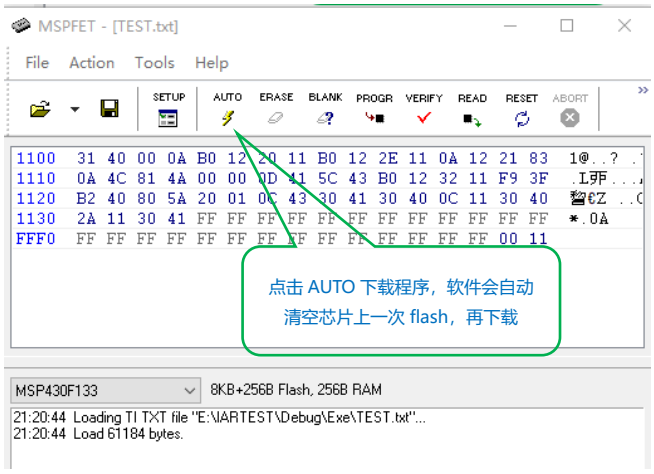
主要就是运行 MspFet.exe 程序，设置，下载。

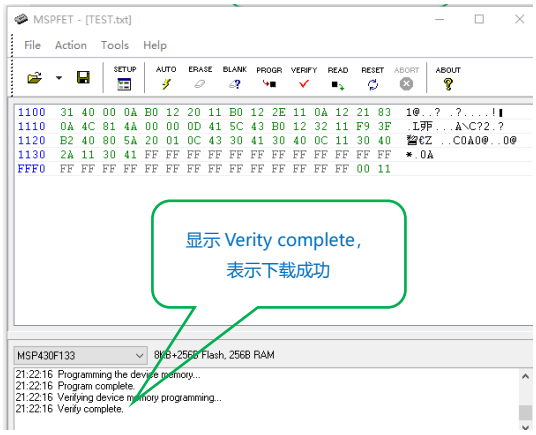




设置完点击 OK







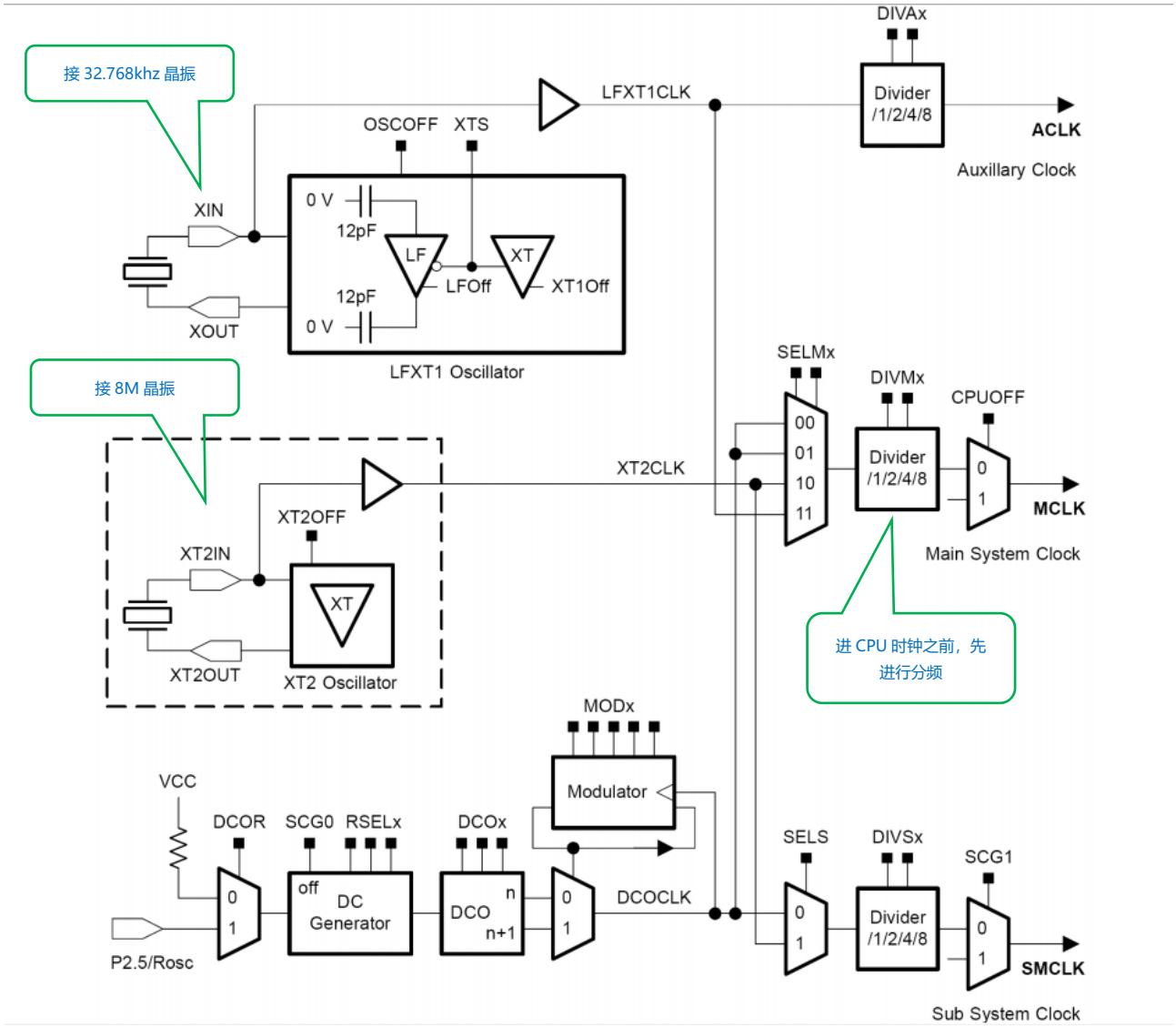
IAR 无法同时打开两个工程，第 2 个工程会覆盖第 1 个工程。如果不想第 1 个工程被第 2 个工程覆盖，就只有再次点击桌面的 IAR 软件，然后 project->Add existing project 打开第 2 个工程

时钟系统

DCOCLK 内部时钟，如果省成本，就可以用这个 DCOCLK 时钟。外部晶振故障也可以切换到该时钟。

XT2CLK 标准外部时钟外接 450kHz～8MHz 的晶振工作，一般都是 430 的主时钟来源。

LFXT1CLK 手表时钟源，外接 32.768khz，用来做时间计数。



外部 8M 晶振启动，LED 闪烁实验

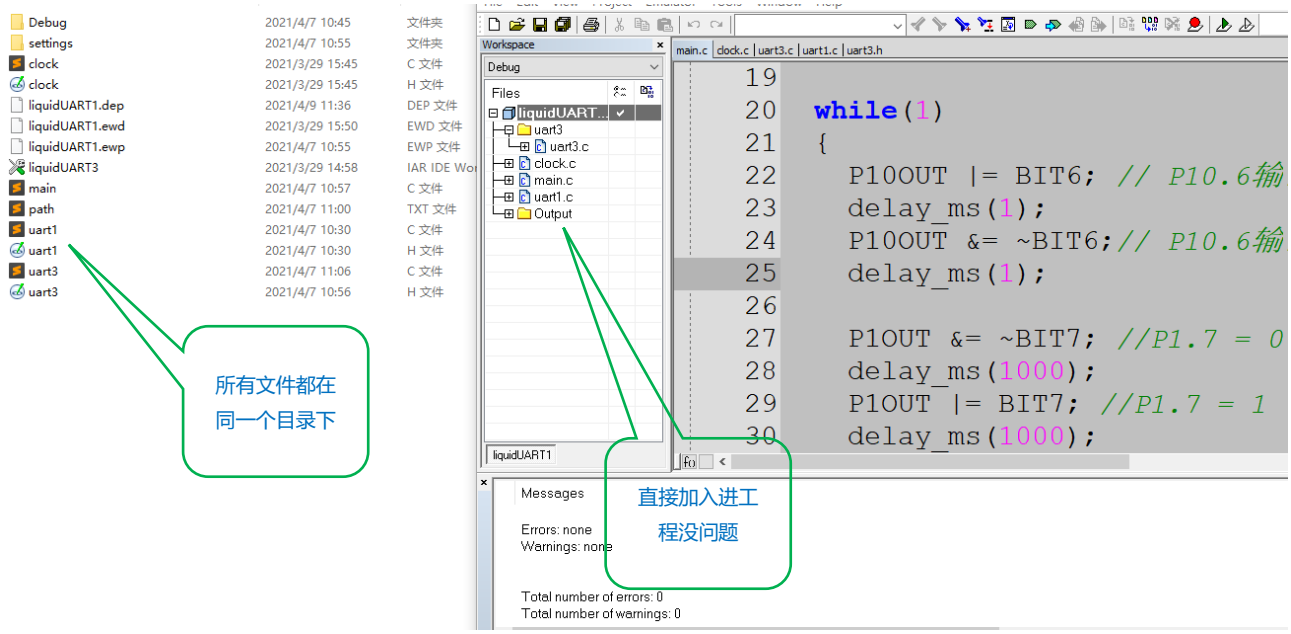
```
#include "io430.h"

/*官方库自定义延时函数*/
#define CPU_F ((double)8000000) //外部高频晶振为 8M
//#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 Clock_Init()
{
  char i;
  BCSCTL1&=~XT2OFF;                    //打开 XT2 振荡器
  BCSCTL2|=SELM1+SELS;                 //MCLK 为 8MHZ，SMCLK 为 8MHZ
  do{
    IFG1&=~OFIFG;                      //清除振荡器错误标志
    for(i=0;i<100;i++)
    {
    }
  }
  while((IFG1&OFIFG)!=0);              //如果标志位 1，则继续循环等待
  IFG1&=~OFIFG;
}

int main( void )
{
  // Stop watchdog timer to prevent time out reset
  WDTCTL = WDTPW + WDTHOLD; //关闭看门狗
  Clock_Init(); //初始化时钟

  P6DIR = 0xFF; //P6 端口 0~7 个 IO 全部为输出

  while(1)
  {
    P6OUT = 0xFF; //P6 端口 0~7 个 IO 全部输出高电平
    delay_ms(500);
    P6OUT = 0x00; //P6 端口 0~7 个 IO 全部输出低电平
    delay_ms(500);
  }

}
```

BCSCTL1 寄存器
0： XT2 高速晶振开

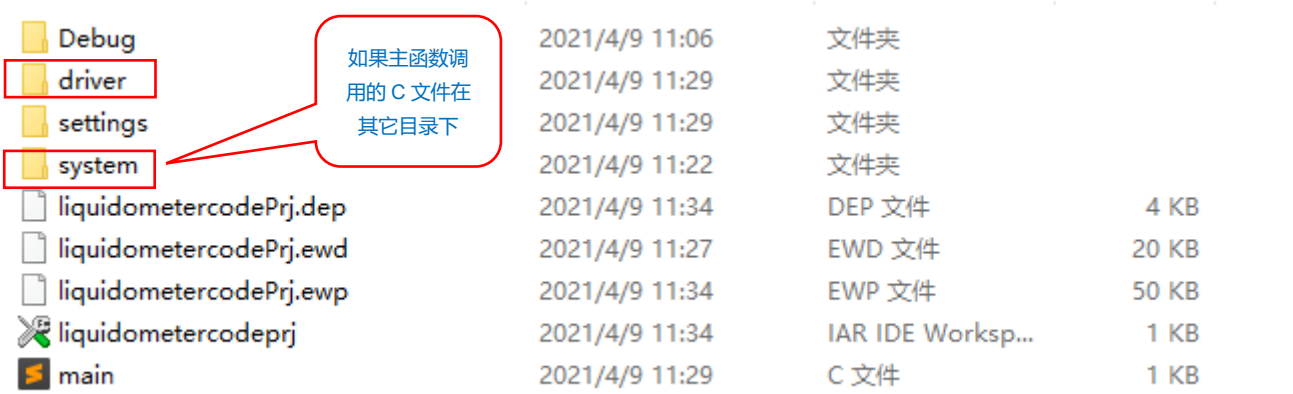
BCSCTL2 寄存器
2(10)： MCLK 时钟源为 XT2CLK
1： SMCLK 时钟源为 XT2CLK

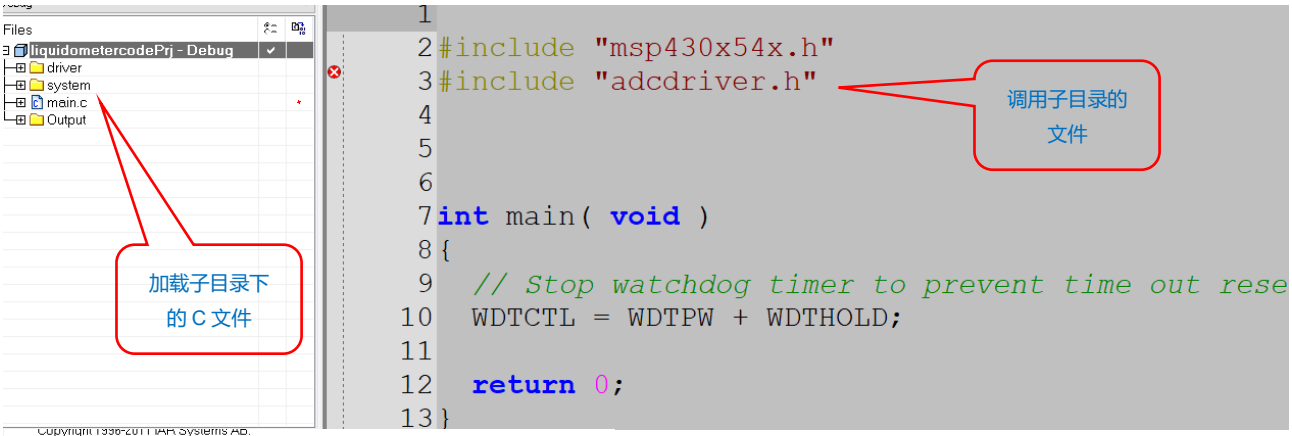
IAR 自定义多目录，多 C/H 文件路径加载方法

所有 c/h 文件在同一个目录下，直接加载进工程就可以



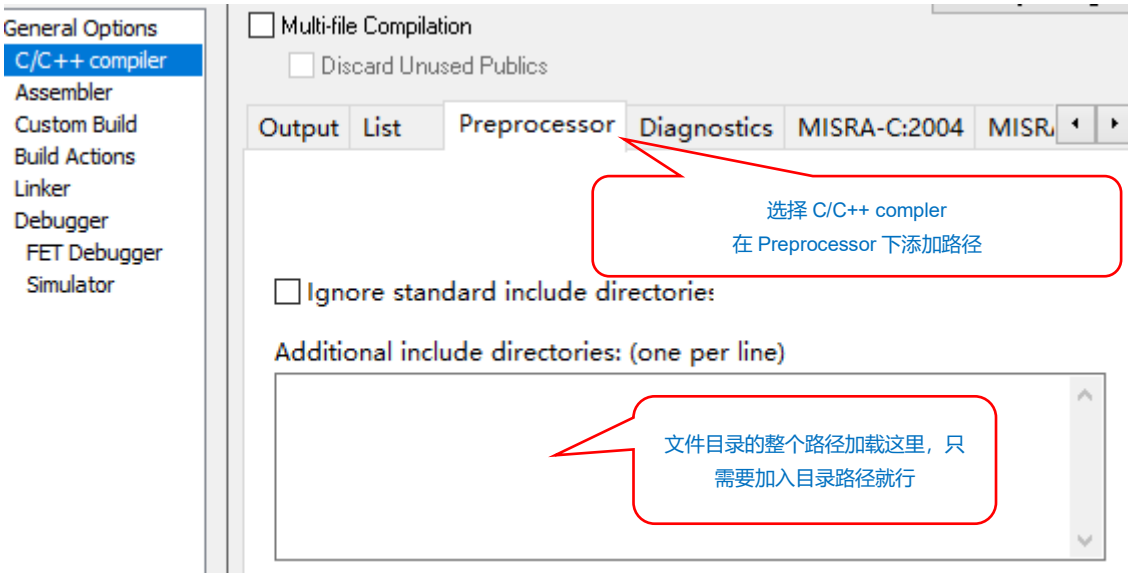
编译通过

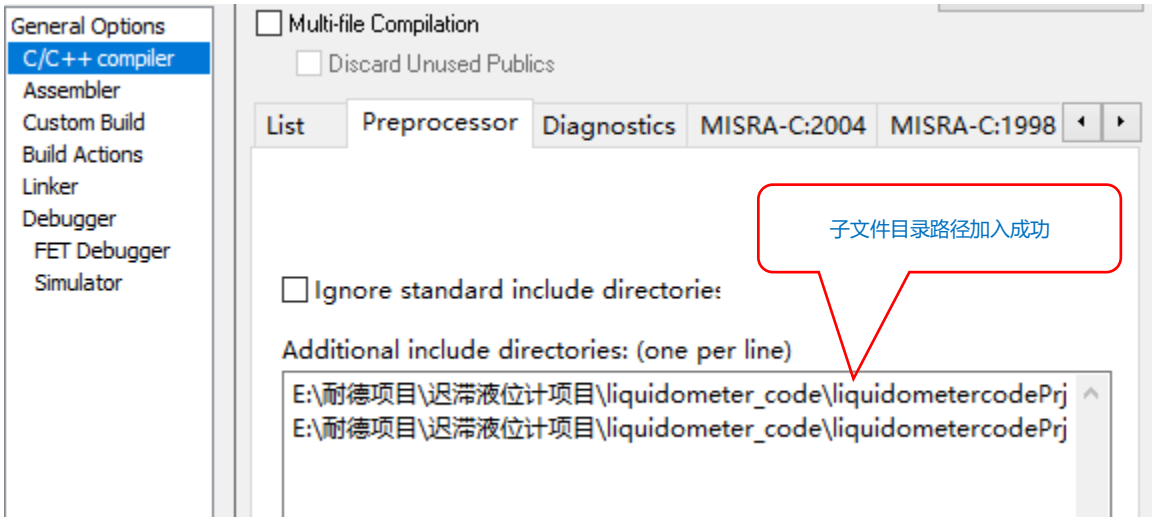


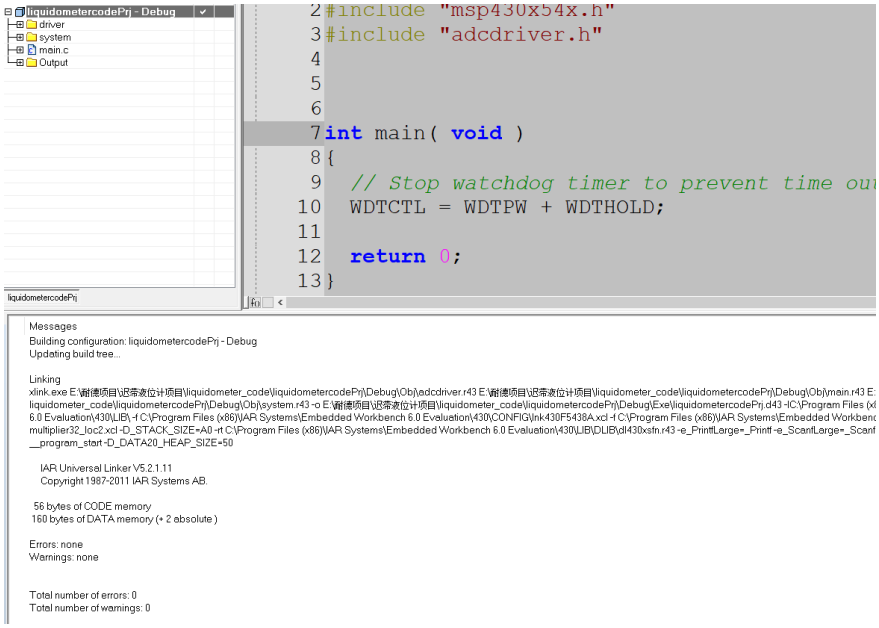


发现编译报错

这是因为 IAR 如果要加入子目录下的 C/H 文件，需要在 IAR IDE 编译器下设置文件路径

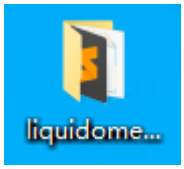




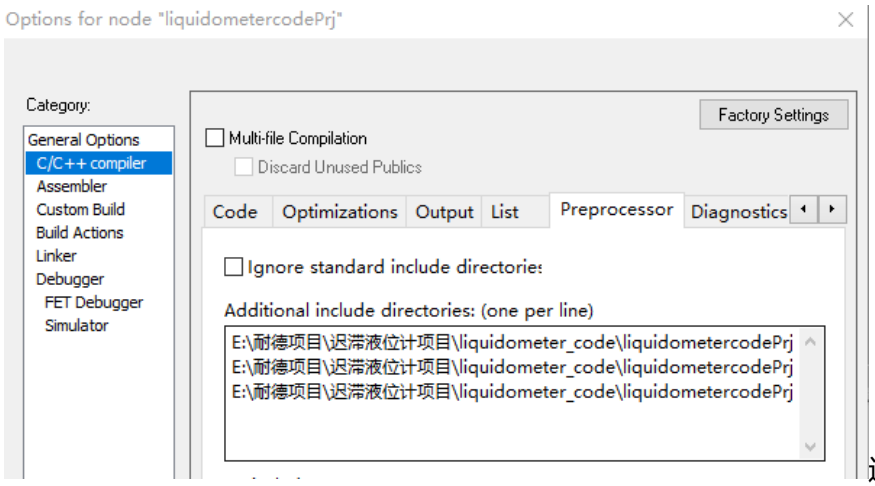


编译通过

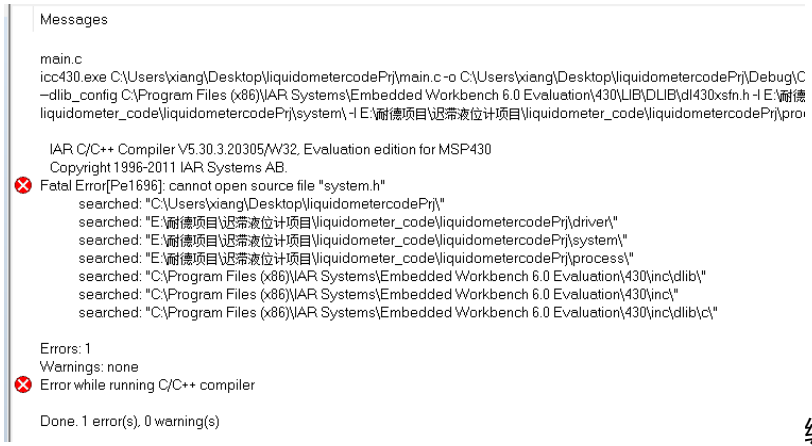
为了方便，可以使用$PROJ_DIR$相对路径的方式来加载 h 文件，这样换个电脑也可以编译，不用重新修改路径



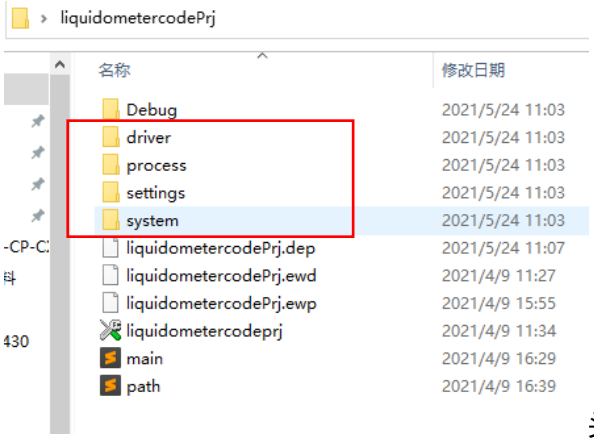
比如该项目拖入到桌面了



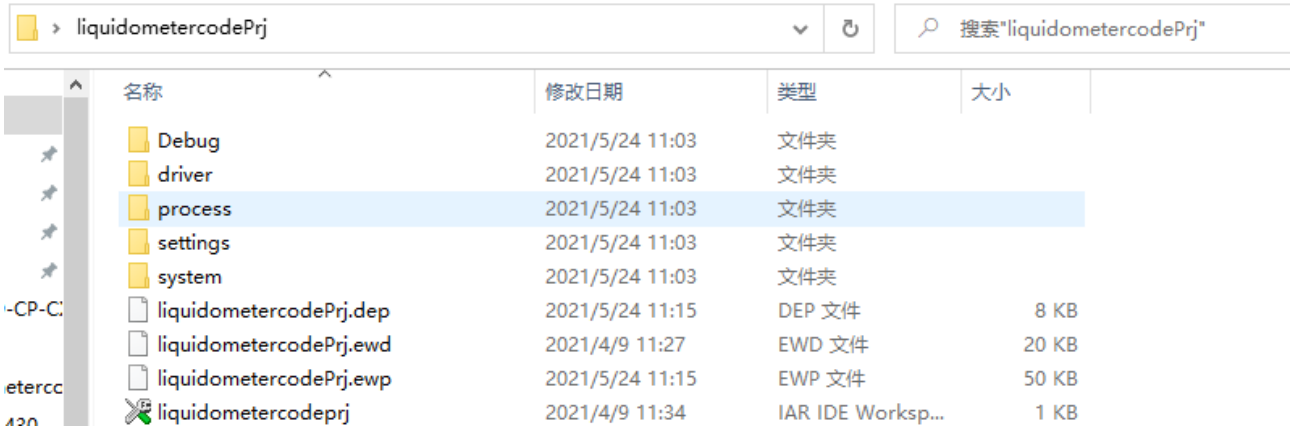
这是 liquidome 以前头文件路径

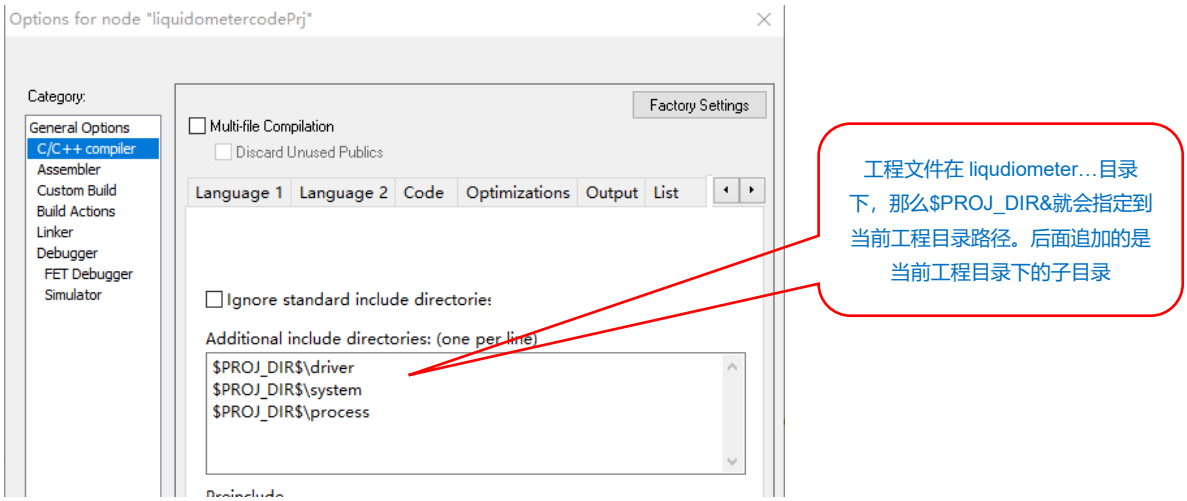


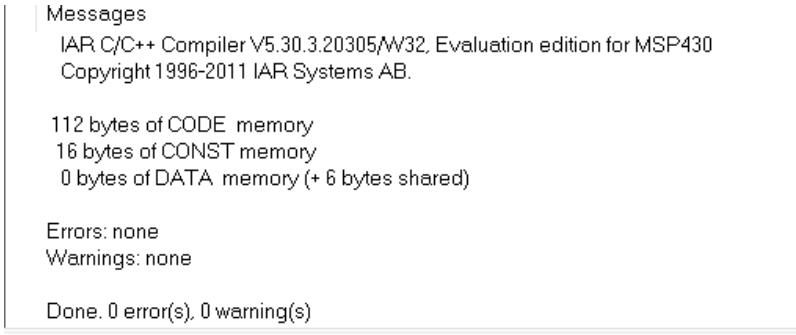
编译错误现在把绝对路径改成相对路径



头文件在第 1 级目录的子目录下



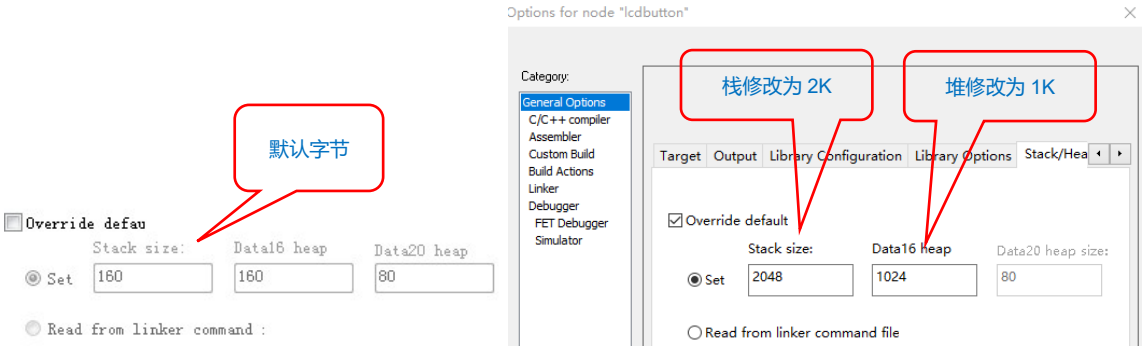




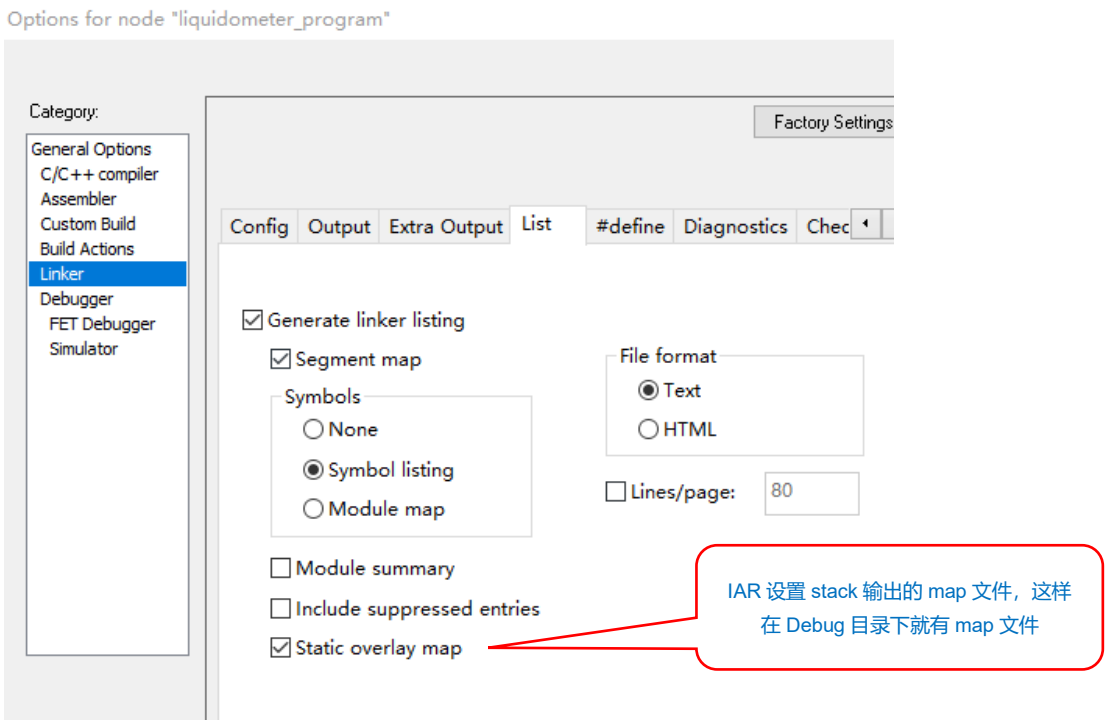
编译成功

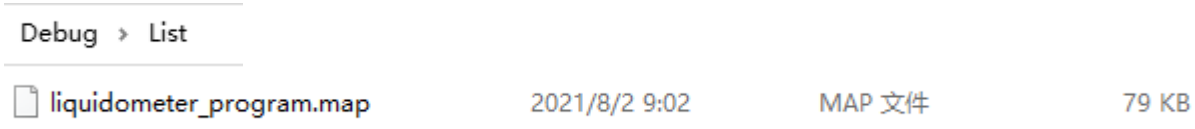
IAR 设置 MSP430 单片机堆栈大小

工程新建立的时候，默认堆栈是 160 字节



查看程序每个函数的栈使用大小





用 IAR 打开 map 文件

```
01  main
      | Stack used (prev) :  0000015E
      | + function block  :  00000008
<-Sub-tree of type: Function tree
  | Stack used           :  00000166
```

主函数栈使用情况

```
03    button_init
        | Stack used (prev) :  00000068
        | + function block  :  0000000C
03    key_PinInit
        | Stack used (prev) :  00000068
        | + function block  :  00000004
02  key_driver_Init
      | Stack used (prev) :  00000074
      | + function block  :  00000004
03    ReadFlash
        | Stack used (prev) :  00000068
        | + function block  :  0000000C
```

各个子函数栈使用情况

GPIO 操作

端口	功能
P1 组、P2 组	I/O、中断能力、其它片内外设功能
P2、P3、P4、P6 组	I/O、其它片内外设功能

P1 和 P2 有 7 个寄存器
P3～P6 各有 4 个寄存器
P1 和 P2 所有 8 位全都可做外部中断处理

返回值 = PxIN_bit.Pn 位操作变量

x: 填入第几组 GPIO

n: 填入该组 GPIO 的第几个管脚

IO 口输入输出程序实例

```
int main( void )
{
  // Stop watchdog timer to prevent time out reset
  WDTCTL = WDTPW + WDTHOLD; //关闭看门狗
  Clock_Init(); //初始化时钟
  P1DIR = 0xF0;  //P1.0~P1.3 输入模式，外部电路已接上拉电阻
  P6DIR = 0xFF; //P6 端口 0~7 个 IO 全部为输出
  while(1)
  {
    if(P1IN_bit.P0 == 0) //位操作(P1IN_bit.P0) P1.0 端口输入低电平，为真
    {
      P6OUT = 0xfe; //(1111 1110)P6.1 端口 = 0 点亮 LED
    }
    if(P1IN_bit.P1 == 0) //位操作(P1IN_bit.P1) P1.1 端口输入低电平，为真
    {
      P6OUT = 0xfd; //(1111 1101)P6.2 端口 = 0 点亮 LED
    }
    if(P1IN_bit.P2 == 0) //位操作(P1IN_bit.P2) P1.2 端口输入低电平，为真
    {
      P6OUT = 0xfb; //(1111 1011)P6.3 端口 = 0 点亮 LED
    }
    if(P1IN_bit.P3 == 0) //位操作(P1IN_bit.P3) P1.3 端口输入低电平，为真
    {
      P6OUT = 0xf7; //(1111 0111)P6.4 端口 = 0 点亮 LED
    }
    P6OUT = 0xFF; //P6 端口 0~7 个 IO 全部输出高电平,关闭 LED

  }
}
```

单个 LED 操作

```
int main( void )
{
  // Stop watchdog timer to prevent time out reset
  WDTCTL = WDTPW + WDTHOLD;

  Clock_Init(); //初始化时钟
//  P6DIR = 0xFF; //P6端口0~7个IO全部为输出
  P6DIR |= BIT0 | BIT2 | BIT4;  //选择P6寄存器的第0位，2位，4位为1，P6.0 P6.2,P6.4输出模式
  while(1)
  {
    P6OUT = 0xFF; //P6端口0~7个IO全部输出高电平，发现只有P6.0,6.2,6.4输出
    delay_ms(500);
    P6OUT = 0x00; //P6端口0~7个IO全部输出低电平
    delay_ms(500);
  }
```

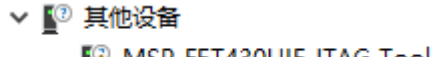
单个 LED 输出操作

```
P6DIR |= BIT2; //P6.2为输出
while(1)
{
  P6OUT |= BIT2; // P6.2输出高电平
  delay_ms(1000);
  P6OUT &= ~BIT2; // P6.2输出高电平
  delay_ms(1000);

}
```

MSP430 JTAG 仿真器使用

JTAG 仿真器，TI 官方自带 USB 供电，可以给板子供电。

市面杂牌 JTAG 仿真器 有些不带给板子供电的功能

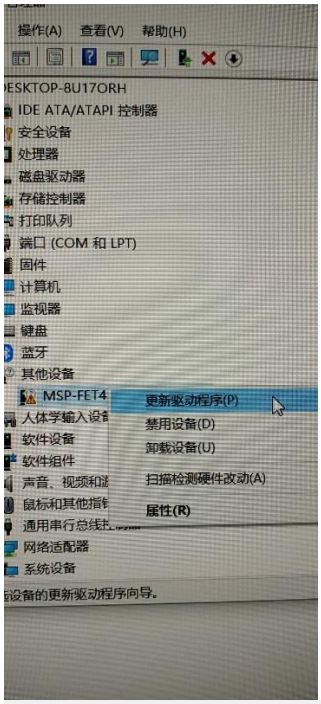


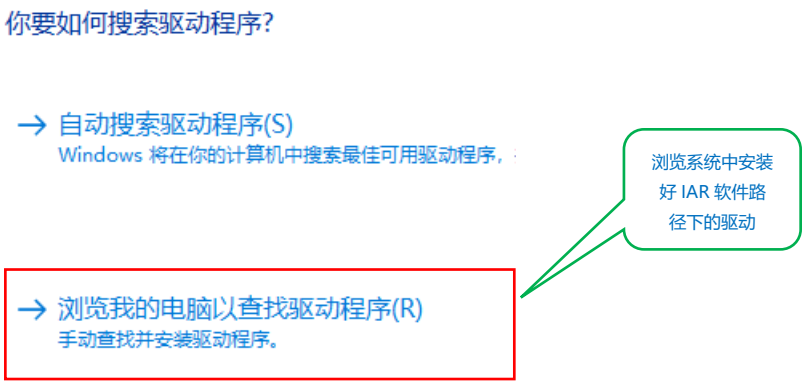
插上仿真器后，有感叹号，表示仿真器驱动没有安装。

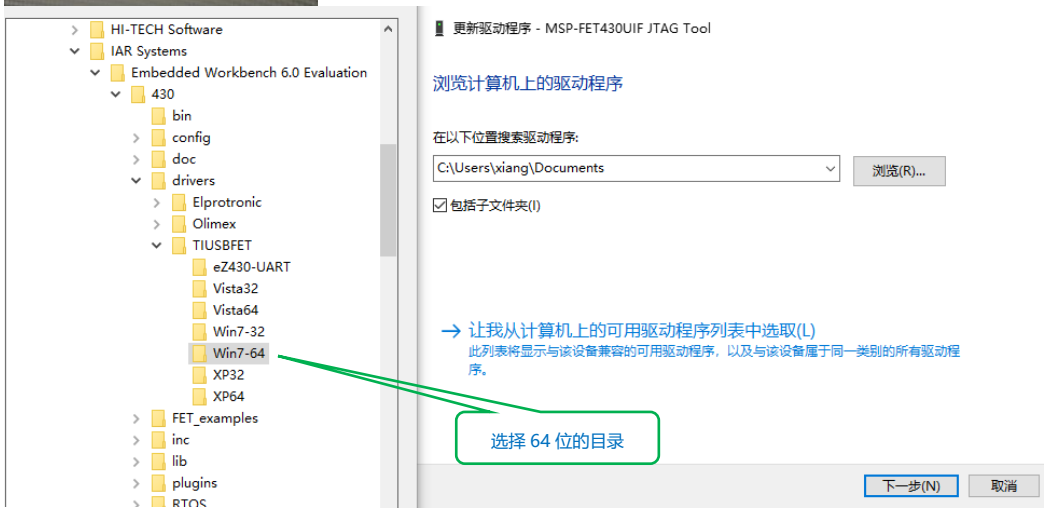
如果安装了 IAR 软件，那么在 IAR 软件安装目录下就有自带的驱动。我们导入即可。

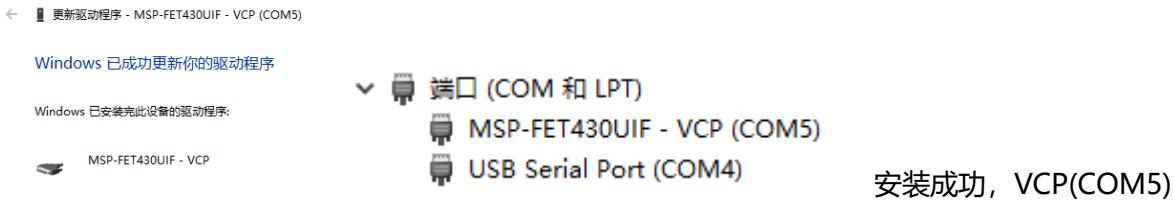
C:\Program Files (x86)\IAR Systems\Embedded Workbench 6.0 Evaluation\430\drivers\TIUSBFET\Win7-64

我们 windows10 就使用 win7-64 目录来安装

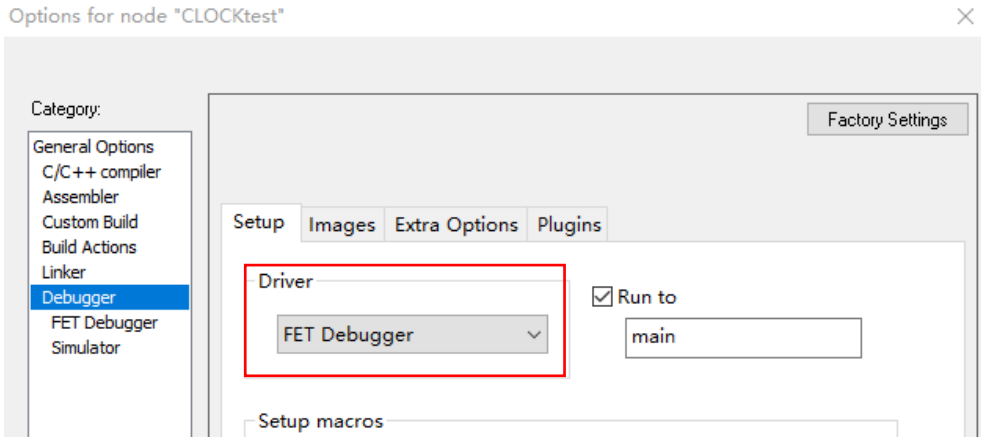




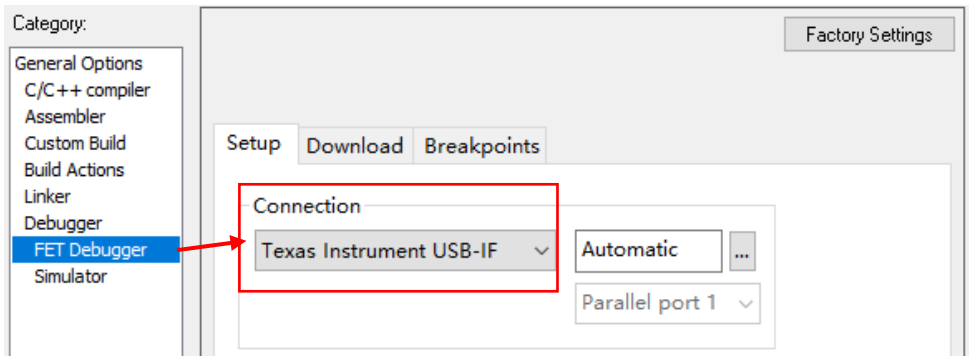




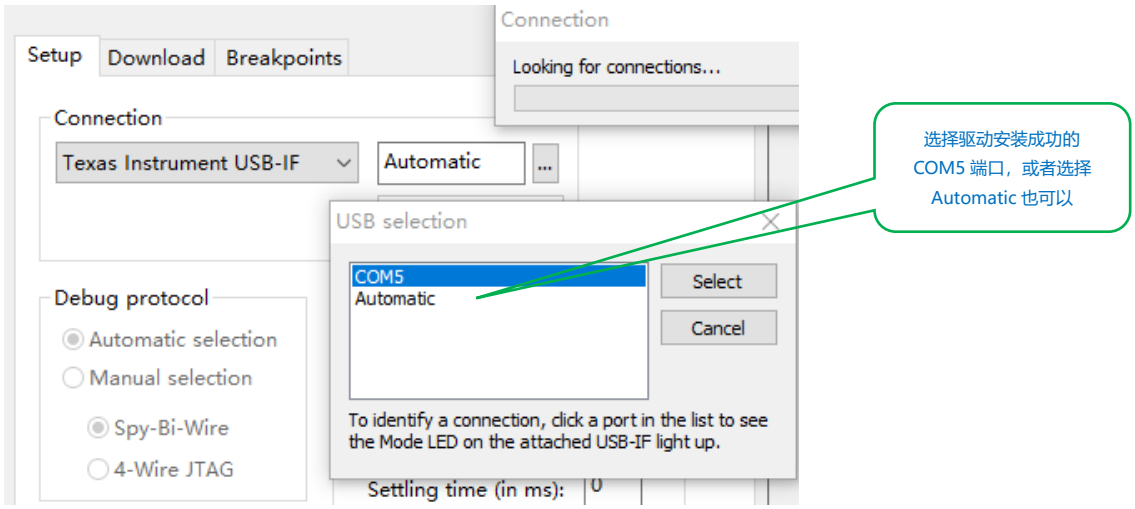
安装成功，VCP(COM5)



软件设置选择 FET

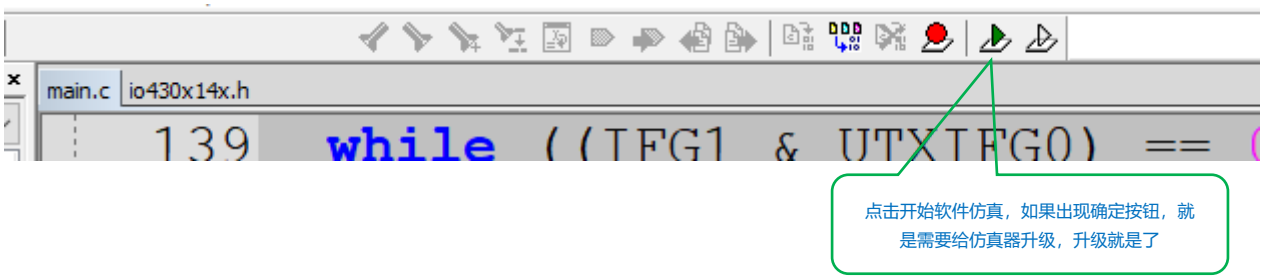


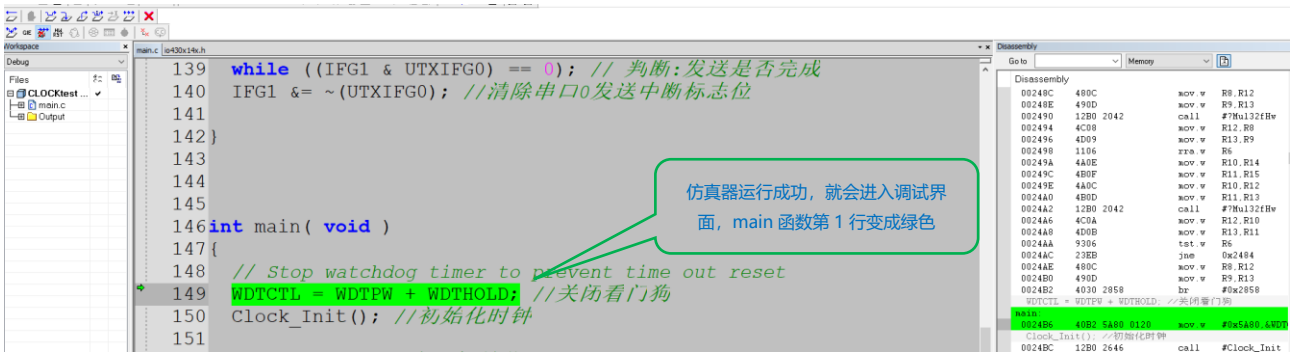
选择 TI USB 仿真器

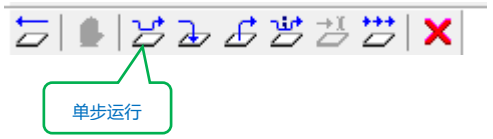


点击 OK，然后保存整个工程 Save All，硬件仿真最好使用 IAR5.3 版本。更高版本会有些 BUG。

开始仿真








```
int main( void )
{
  // Stop watchdog timer to prevent time out reset
  WDTCTL = WDTPW + WDTHOLD; //关闭看门狗
  Clock_Init(); //初始化时钟

  InitUSART(); //串口初始化
  Init cap(); //初始化捕获中断
```

单步运行了两步

如果用 JTAG 仿真器下载整个程序，用 Download active application

选择工程 Project->Download->Download active application 进行下载

```
Fri Mar 19, 2021 10:24:35: Interface dll version 2.4.9.1
Fri Mar 19, 2021 10:24:35: Device : MSP430F149
Fri Mar 19, 2021 10:24:35: External voltage : 0.0 V
Fri Mar 19, 2021 10:24:35: VCC voltage : 3.3 V
Fri Mar 19, 2021 10:24:37: Download complete.
Fri Mar 19, 2021 10:24:37: Loaded debugee: F:\MSP430TEST\TIMRAinputcap\Debug\Exe\CLOCKtest.d43
Fri Mar 19, 2021 10:24:37: Target reset
```

下载之后显示 Target reset 表示下载成功。然后要点击复位键才能开始运行。

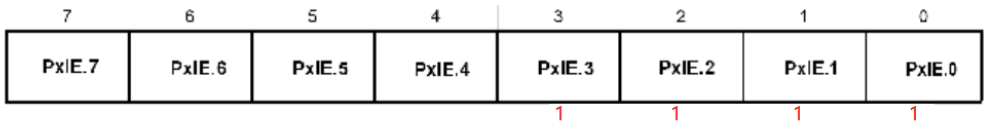
编译警告 pointless integer comparison, the result is always false

如果定义的是 unsigned int 变量

然后 if(变量 == 115200) 那么就会报错，因为 MSP430 中 unsigned int 是 16 位宽的最大 65535 比 115200 少了个数量级。

GPIO 外部中断实验

[6] PxIE　中断允许寄存器



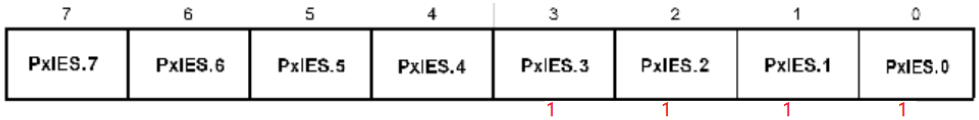
该寄存器只有 P1 和 P2 口才有，该寄存器有 8 个标志位，标志相应引脚是否能响应中断请求。

PxIFG.x：中断允许标志

0：　该引脚中断禁止

1：　该引脚中断允许

[7] PxIES　中断触发沿控制寄存器



该寄存器只有 P1 和 P2 口才有，该寄存器有 8 个标志位，标志相应引脚的中断触发沿。

PxIFG.x：中断触发沿选择

0：　上升沿产生中断

1：　下降沿产生中断

外部中断示例程序

```
#include "io430.h"
#include "in430.h" //外部中断_EINT();需要用到的输入头文件


int main( void )
{
  // Stop watchdog timer to prevent time out reset
  WDTCTL = WDTPW + WDTHOLD; //关闭看门狗
  Clock_Init(); //初始化时钟

  P1DIR = 0xF0;   //P1.0~P1.3 输入模式，方便做中断输入
  P1IE  = 0x0F;                    //开启 P1.0~P1.3 中断
  P1IES = 0x0F;                    //P1.0~P1.3 下降沿触发中断
  P1IFG = 0x00;                    //软件清零中断标志寄存器

  P6DIR = 0xFF; //P6 端口 0~7 个 IO 全部为输出

  _EINT(); //IO 口寄存器设置完毕，开启总中断



  while(1)
  {
    P6OUT = 0xff;
    delay_ms(1000);
  }
}
```

```
/*
*P1 口中断服务程序
* P1 组任何一个 IO 口中断都会进入 P1 中断服务程序
*/

#pragma vector = PORT1_VECTOR
__interrupt void P1_IRQ(void)
{
  P6OUT = 0xfe; //(1111 1110)P6.1 端口 = 0 点亮 LED
  P6OUT = 0xfd; //(1111 1101)P6.2 端口 = 0 点亮 LED
  P6OUT = 0xfb; //(1111 1011)P6.3 端口 = 0 点亮 LED
  P6OUT = 0xf7; //(1111 0111)P6.4 端口 = 0 点亮 LED
  P1IFG = 0x00; //软件清零中断标志寄存器 一定要在中断中清除
  delay_ms(2000); //中断里面不要执行消耗时间的函数，这里只是为了演示 LED
}
```

[5] PxIFG 中断标志寄存器

7	6	5	4	3	2	1	0
PxIFG.7	PxIFG.6	PxIFG.5	PxIFG.4	PxIFG.3	PxIFG.2	PxIFG.1	PxIFG.0
				1/0	1/0	1/0	1/0

该寄存器只有 P1 和 P2 口才有，该寄存器有 8 个标志位，标志相应引脚是否有中断请求。

PxIFG.x：中断标志

0：该引脚无中断请求

1：该引脚有中断请求

如果有任意1位外部中断触发，就会导致中断标志寄存器置1.必须手动清除中断标志位

外部输入脉冲触发中断信号必须保持 1.5 个 MCLK 才能触发中断(也就是脉冲宽度必须保持 1.5 个 MCLK)

串口通信

串口 0 程序实例

```
#define CPU_F ((double)8000000) //外部高频晶振为 8M

//串口波特率计算，当 BRCLK=CPU_F 时用下面的公式可以计算，否则要根据设置加入分频系数
#define baud          9600                                    //设置波特率的大小
#define baud_setting  (unsigned int)((unsigned long)CPU_F/((unsigned long)baud))  //波特率计算公式
#define baud_h        (unsigned char)(baud_setting>>8)        //提取高位
#define baud_l        (unsigned char)(baud_setting)
void InitUSART(void)
{
  U0CTL|=SWRST;                 //复位 SWRST
  U0CTL|=CHAR;                  //8 位数据模式
  U0TCTL|=SSEL1;                //SMCLK 为串口时钟
  U0BR1=baud_h;                 //SMCLK 频率/9600 = 16 进制值，值高 8 位写 U0BR1
  U0BR0=baud_l;                 //16 进制值低 8 位写 U0BR0    注意:9600 波特率 8M/9600
  U0MCTL=0x00;                  //微调寄存器为 0，波特率 9600bps
  ME1|=UTXE0;                   //UART0 发送使能
  ME1|=URXE0;                   //UART0 接收使能
  U0CTL&=~SWRST;                //SWRST = 0,串口初始化完成
  IE1|=URXIE0;                  //接收中断使能位
  P3SEL|= BIT4;                 //设置 IO 口为普通 I/O 模式
  P3DIR|= BIT4;                 //设置 IO 口方向为输出
  P3SEL|= BIT5;
  _EINT();                      // 不要忘了开中断
}
unsigned char TBuff[8]; // 发送缓冲区
// 串口发送函数(不需要开发送中断)发送一个数组(共 8 个字节)
void USART_Send(unsigned char *pData)
{
  unsigned char i;
  for(i=0; i<8; i++)
  {
  TXBUF0 = pData[i]; // 装入发送寄存器
  while ((IFG1 & UTXIFG0) == 0); // 判断:发送是否完成
  IFG1 &= ~(UTXIFG0);
  }
```

U0CTL 寄存器
SWRST 位，在初始化串口的时候一定要先将 SWRST = 1，然后开始初始化串口其它参数，初始化完串口其它参数后，必须将 SWRST = 0.这样才能使串口正常收发

U0CTL 寄存器
(10) CHAR 寄存器置 1，8 位数据位
CHAR 寄存器置 0，7 位数据位

ME1 串口 0 发送接受允许/禁止寄存器

```
}
int main( void )
{
  // Stop watchdog timer to prevent time out reset
  WDTCTL = WDTPW + WDTHOLD; //关闭看门狗
  Clock_Init(); //初始化时钟
  InitUSART(); //串口初始化

  unsigned char buffer[8] = {0x01,0x02,0x03,0x04,0x05,0x06,0x07};
  P6DIR = 0xFF; //P6 端口 0~7 个 IO 全部为输出
  P6OUT = 0x00; //点亮全部 LED
  while(1)
  {
    USART_Send(buffer);
    delay_ms(1000);
  }
}
```

串口 printf 实现

只需要重新编写 putchar 函数即可

```
#include <stdio.h>
int main( void )
{
  // Stop watchdog timer to prevent time out reset
  WDTCTL = WDTPW + WDTHOLD; //关闭看门狗
  Clock_Init(); //初始化时钟

  InitUSART(); //串口初始化 9600 波特率

  P6DIR = 0xFF; //P6 端口 0~7 个 IO 全部为输出
  P6OUT = 0x00; //点亮全部 LED

  int data = 20;
  while(1)
  {
    printf("xxxzzzzz = %d\n",data); //IAR 430 编辑器换行只需要‘\n 即可’不需要\r

    delay_ms(1000);
  }
}


/*
* 主要实现 printf 底层的 putchar 函数重写
*/
int putchar(int ch)
{
  if(ch == '\n')
  {
    while ((IFG1 & UTXIFG0) == 0); // 判断:发送是否完成
    TXBUF0 = 0x0d; // 装入发送寄存器
  }
  while ((IFG1 & UTXIFG0) == 0); // 判断:发送是否完成
  TXBUF0 = ch; // 装入发送寄存器
   return (ch);
}
```

串口 0 接受中断实现

```
/*
* 串口单字节发送
*/
void USART_Send(unsigned char pData)
{
  TXBUF0 = pData; // 装入发送寄存器
  while ((IFG1 & UTXIFG0) == 0); // 判断:发送是否完成
  IFG1 &= ~(UTXIFG0); //清除串口 0 发送中断标志位

}


int main( void )
{
  // Stop watchdog timer to prevent time out reset
  WDTCTL = WDTPW + WDTHOLD; //关闭看门狗
  Clock_Init(); //初始化时钟

  InitUSART(); //串口初始化


  while(1)
  {

    delay_ms(1000);
  }
}

// 串口接受函数(需要开接受中断)
#pragma vector=UART0RX_VECTOR
__interrupt void USART0_RXIRQ (void)
{
  unsigned char Temp;
  // 暂存接受数据
  Temp = RXBUF0;
  IFG1 &= ~(URXIFG0); //清除串口 0 接收中断标志位
  USART_Send(Temp);//接受数据转发回 PC

}
```

16 位定时器 A 使用

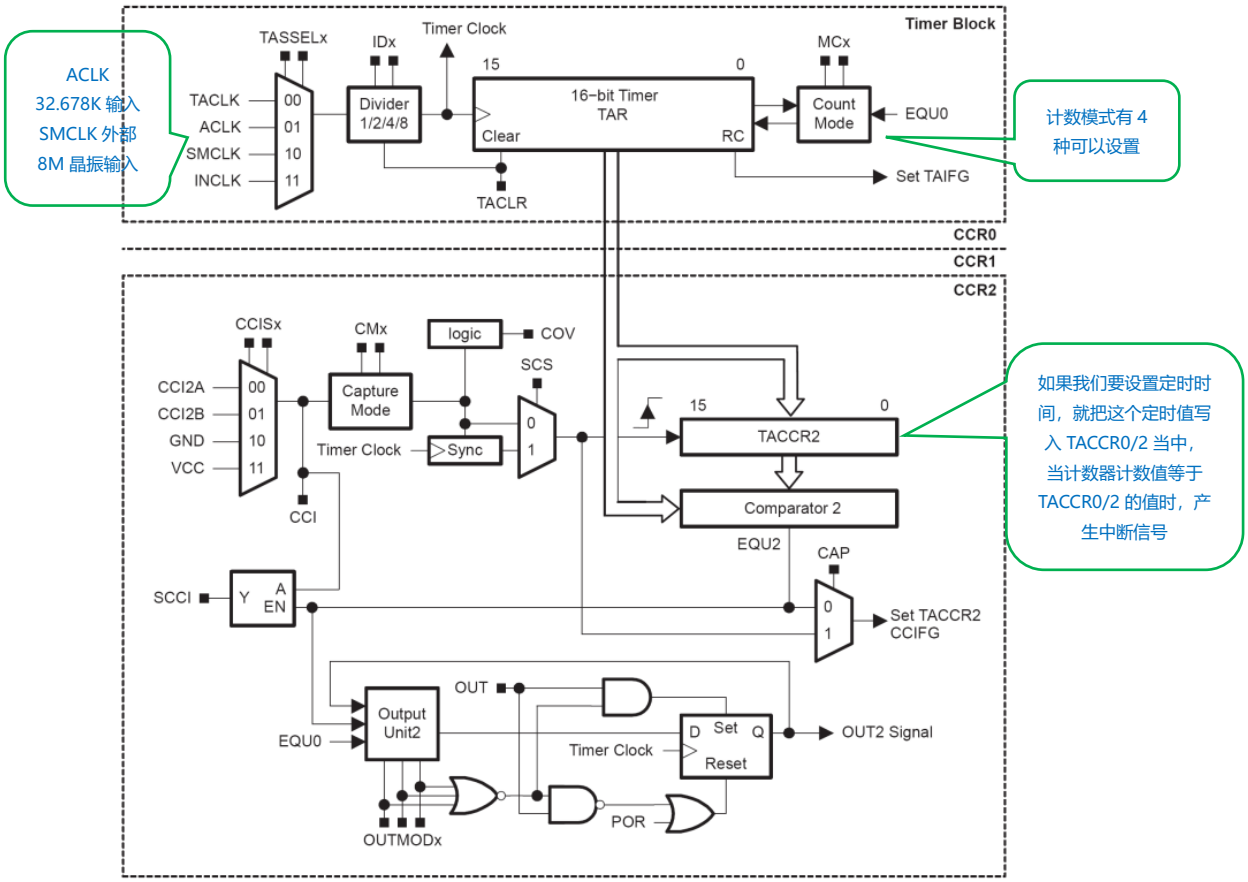
定时器 A 有 3 个捕获比较寄存器，可配置位 PWM 输出

支持 16 位异步定时计数

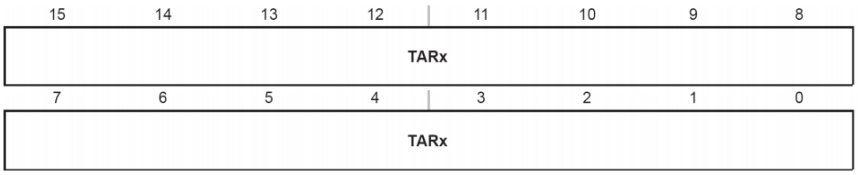
PWM 输出配合滤波器输出 DAC

对外部输入中断进行计数

TIMER_A 的结构原理图。



[2] TAR TIMER_A 计数器

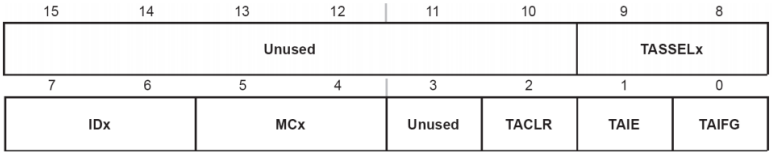


该单元就是执行计数的单元，时计数器的主体，其内容可读可写。

TAR 计数器，如果不和 TACCR0 对比，那么 TAR 16 位计数器计数满 0xffff 就会产生溢出中断。

TAR 计数器和 TACCR0 对比，那么 TAR 计数计数值增加到与 TACCR0 值一样时，会产生捕获比较中断。

[1] TACTL TIMER_A 控制寄存器



TACTL 寄存器，要注意 TACLR 位清 0 功能。该位不仅会清除计数器的值，还会清除定时器时钟的设置。还会清除计数器递增，递减方向设置。

定时器启动方式

MCx	Mode	Description
00	Stop	The timer is halted.
01	Up	The timer repeatedly counts from zero to the value of TACCR0
10	Continuous	The timer repeatedly counts from zero to 0FFFFh.
11	Up/down	The timer repeatedly counts from zero up to the value of TACCR0 and back down to zero.

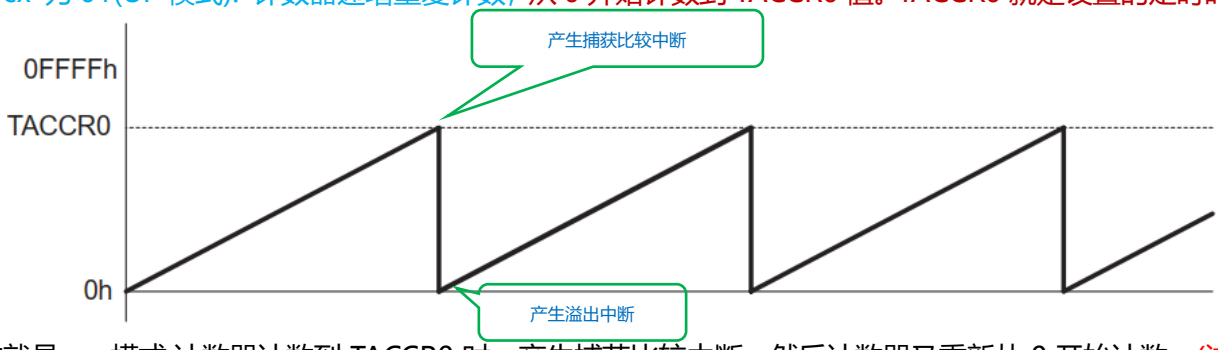
MCx 位 >0 时，并且定时器外部输入的时钟源有效。那么定时器就开始启动。所以这一点和 STM32 和 51 还不一样。MSP430 没有专门的定时器启动函数。

如果 MCx 是计数器递增或递减模式。通过给 TACCR0 写 0 就可以终止计数。如果给 TACCR0 写非 0 的值，计数器就会从 0 开始重新启动。一直计数到 TACCR0 产生中断。

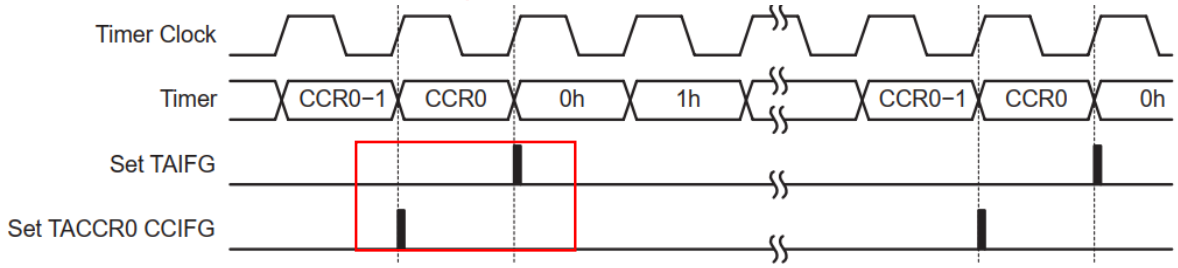
Mcx 为 00 (停止模式)，用于定时器暂停，但是不会给计数器清 0，当定时器从暂停之后开始启动。那么计数器会接着计数。类似播放音乐这种暂停方式。而且最先设置的计数器递增或者递减模式不会被改变。如果想定时器暂停后改变计数器的模式。就必须给 TACCR0 清 0

或者给 TACLR 位清 0，来清除设置。然后重新设置定时器。

Mcx 为 01(UP 模式): 计数器递增重复计数，从 0 开始计数到 TACCR0 值。TACCR0 就是设置的定时时间

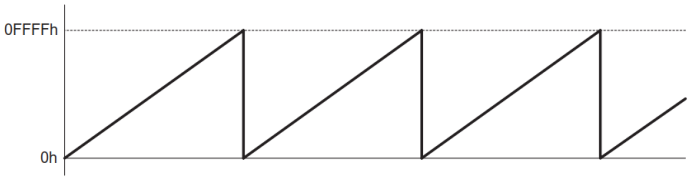


这就是 up 模式,计数器计数到 TACCR0 时，产生捕获比较中断，然后计数器又重新从 0 开始计数。(注意计数器变为 0 时会产生溢出中断)。所以 up 模式是两个中断。

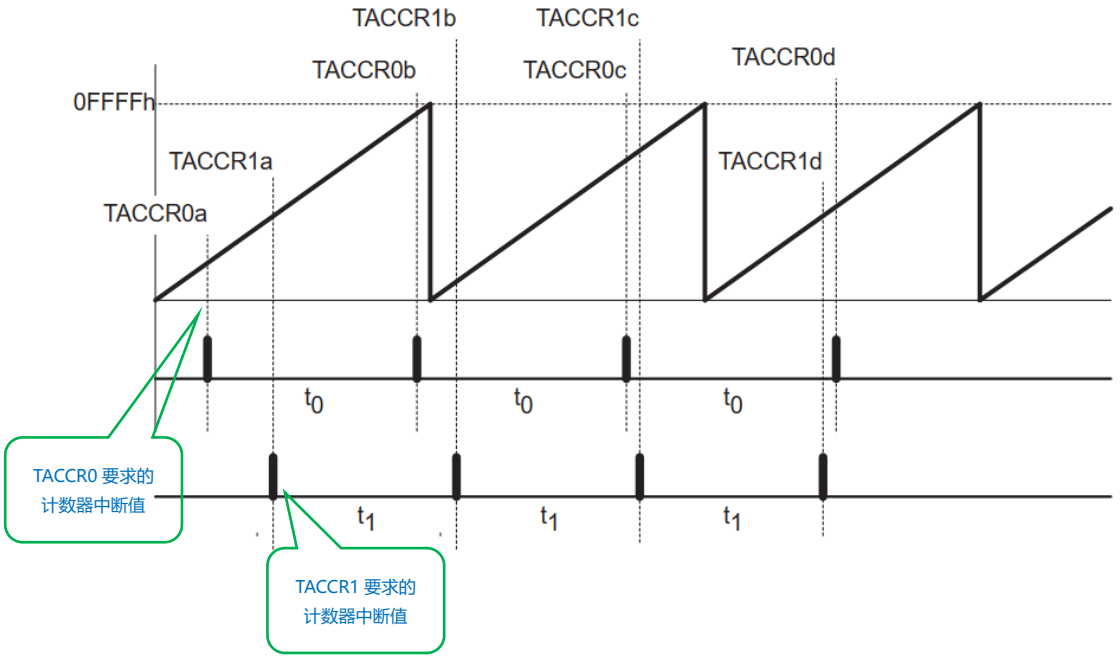


这就是 up 模式的两个中断。

Mcx 为 10(连续计数模式): 就是计数器最大值为 0xffff，不能修改，计数器不停的计数，计数到 0xffff 产生中断，然后计数器清 0，从新开始计数。

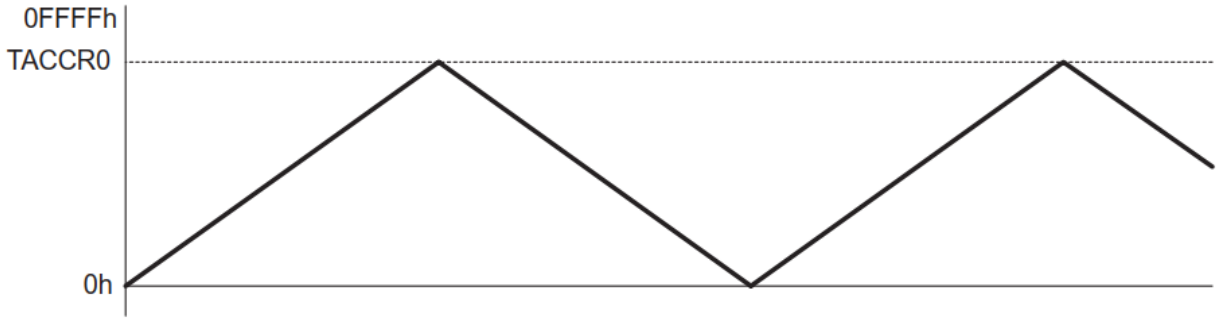


我们也可以利用 TACCR0,TACCR1 来对连续模式的定时器进行采样。这样可以产生多个不同计数值得中断

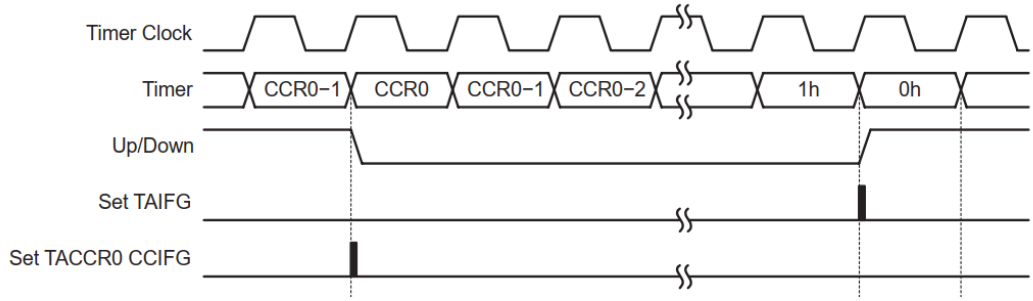


这种方式主要是使用外部引进捕获功能。来计算外部信号的时间宽度。

Mcx 为 11(连续递增递减模式): 计数器先从 0 递增到 TACCR0，然后 TACCR0 再递减到 0，重复执行



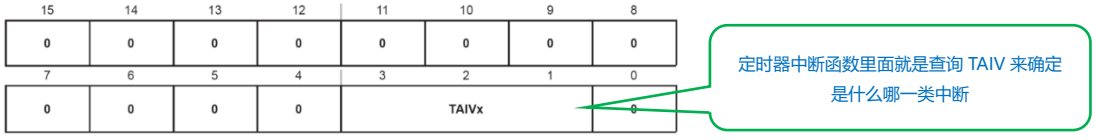
适合做三角波发生器。



递增递减模式，一个三角波两个中断。

在定时器中断函数里面要判断是哪一个中断向量触发了

[5] TAIV TIMER_A 中断向量寄存器



TAIV Contents	Interrupt Source	Interrupt Flag	Interrupt Priority
00h	No interrupt pending	–	
02h	Capture/compare 1	TACCR1 CCIFG	Highest
04h	Capture/compare 2	TACCR2 CCIFG	
06h	Reserved	–	
08h	Reserved	–	
0Ah	Timer overflow	TAIFG	
0Ch	Reserved	–	
0Eh	Reserved	–	Lowest

02,04 是比较捕获中断，0A 是定时器溢出中断

定时器连续模式闪烁 LED 实例

```
/*****************************
* 定时器 A 初始化
* 定时器连续模式
*****************************/
void TIMRA(void)
{
  TACTL |= TASSEL1 + TACLR + ID0 + ID1 + MC1 + TAIE;
  //TASSEL1  用 SMCLK 做时钟源，
  //TACLR  清除定时器默认配置
  //ID0 + ID1 (11) 8 分频(晶振 8M/8=1M)计数器加 1 个数是 1us
  //MC1 连续计数模式，计数到 0XFFFF
  //TAIE (10)  允许定时器溢出中断,清除 TA 溢出标志位
}
int main( void )
{
  // Stop watchdog timer to prevent time out reset
  WDTCTL = WDTPW + WDTHOLD; //关闭看门狗
  Clock_Init(); //初始化时钟

  TIMRA();      //定时器初始化
  _EINT(); //开启总中断

  P6DIR = 0xFF; //P6 端口 0~7 个 IO 全部为输出
  P6OUT = 0x00; //点亮全部 LED

  while(1)
  {
  }
}
/*定时器 A 中断触发程序*/
#pragma vector = TIMERA1_VECTOR
__interrupt void Timer_A(void)
{
        //定时器的中断，必须读取 TAIV 变量来清除中断标志位，
  //不然第一次进入中断后就无法退出中断，一定要注意，困扰了我一下午
  switch(TAIV)
  {
    case 10: P6OUT = ~P6OUT;break; //10(0A)  定时器溢出中断,65ms 中断一次，连续模式是增加到 65535(0xffff)才会中断，LED 快闪。
  }
}
```

定时器指定，定时时间实例

```
/*******************************
* 定时器 A 初始化
* 定时器增量模式
*******************************/
void TIMRA(void)
{
  TACTL |= TASSEL1 + TACLR + ID0 + ID1 + MC0 + TAIE;
  //TASSEL1 用 SMCLK 做时钟源，
  //TACLR 清除 TAR，时钟分频，计数模式的设置。清除设置后自动清零
  //MC0(01) 计数器增量模式
  //ID0 + ID1 (11) 8 分频(晶振 8M/8=1M)计数器加 1 个数是 1us
  //TAIE 允许定时器溢出中断

  TACCR0 = 9999; //定时时间初值，计数器 1us 加一次，10ms 产生一次中断
  //等待计数器累加到 TACCR0 的值，就产生中断
}

int main( void )
{
  // Stop watchdog timer to prevent time out reset
  WDTCTL = WDTPW + WDTHOLD; //关闭看门狗
  Clock_Init(); //初始化时钟
  TIMRA();      //定时器初始化

  _EINT(); //开启总中断

  P6DIR = 0xFF; //P6 端口 0~7 个 IO 全部为输出
  P6OUT = 0x00; //点亮全部 LED

  while(1)
  {
    if(Count > 20) //这就是定时时间，定时器 10ms 中断一次，200ms 间隔灯翻转
    {
      Count = 0;
      P6OUT = ~P6OUT;
    }
   // delay_ms(500);
  }
}
/*定时器 A 中断触发程序*/
#pragma vector = TIMERA1_VECTOR
__interrupt void Timer_A(void)
{
  //定时器的中断，必须读取 TAIV 变量来清除中断标志位，
  //不然第一次进入中断后就无法退出中断，一定要注意，困扰了我一下午
  switch(TAIV)
  {
    case 10:Count++ ;break; //10(0A)定时器溢出中断
  }
}
```

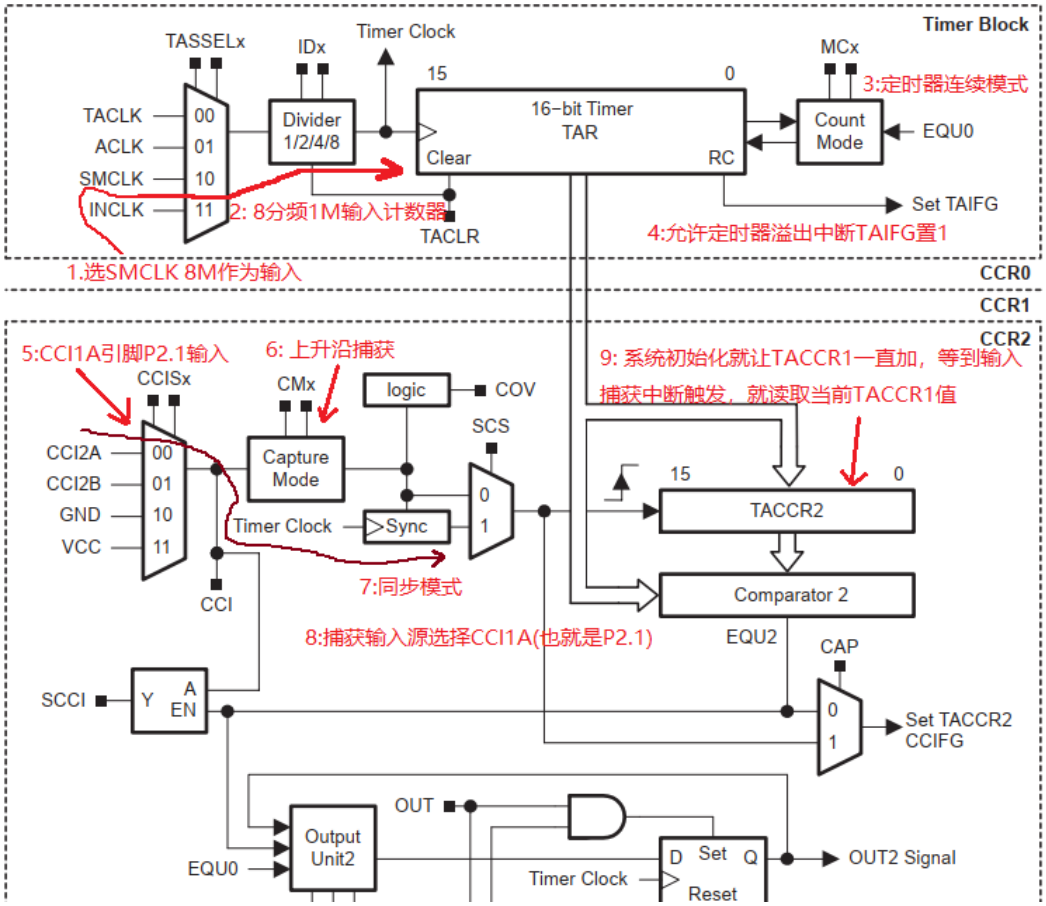
引脚输入信号电平捕获(定时器输入捕获)

输入捕获引脚有哪些?

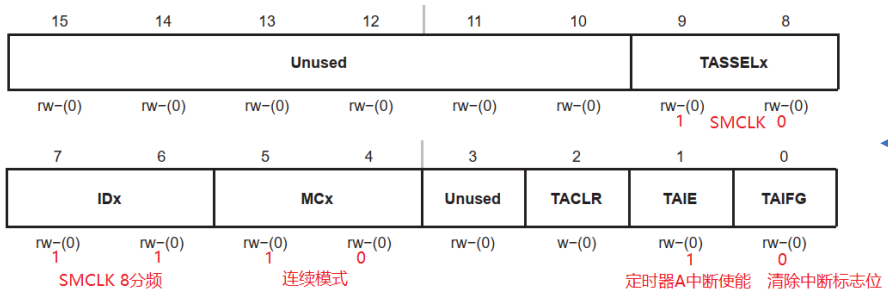
TIMER_A3 SIGNAL CONNECTIONS					
INPUT PIN NUMBER	DEVICE INPUT SIGNAL	MODULE INPUT NAME	MODULE BLOCK	MODULE OUTPUT SIGNAL	OUTPUT PIN NUMBER
12 - P1.0	TACLK	TACLK	Timer	NA	
	ACLK	ACLK			
	SMCLK	SMCLK			
21 - P2.1	TAINCLK	INCLK			
13 - P1.1	TA0	CCI0A	CCR0	TA0	13 - P1.1
22 - P2.2	TA0	CCI0B			17 - P1.5
	DV_{SS}	GND			27 - P2.7
	DV_{CC}	V_{CC}			
14 - P1.2	TA1	CCI1A	CCR1	TA1	14 - P1.2
	CAOUT (internal)	CCI1B			18 - P1.6
	DV_{SS}	GND			23 - P2.3
	DV_{CC}	V_{CC}			ADC12 (internal)
15 - P1.3	TA2	CCI2A	CCR2	TA2	15 - P1.3
	ACLK (internal)	CCI2B			19 - P1.7
	DV_{SS}	GND			24 - P2.4
	DV_{CC}	V_{CC}			

我们选用 P1.2 端口输入捕获，那么就是 TIMRA 定时器，获取 TACCR1 寄存器数据计算方波周期

方波周期采集

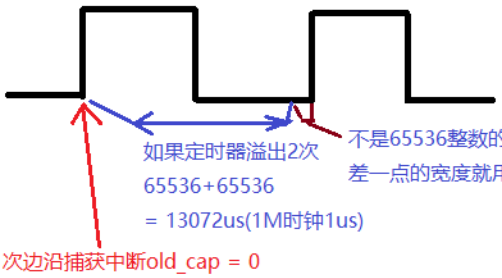


TACTL, Timer_A Control Register




```
/*
* 定时器 A 初始化
*/
void Init_cap(void)
{
  P1DIR = 0x00; //P1.2 设置 IO 口为输入模式
  P1SEL  |=0x04;        //选择 P1.2 作为输入捕获功能
  TACTL   =TASSEL_2+MC_2+ID_3+TAIE;    //选择 8M - SMCLK 时钟
  //TASSEL_2 选择 SMLCK 时钟
  //MC_2 定时器连续模式
  //ID_3 输入时钟 8 分频，SMLCK/8=1M(1us) TIMRA 定时器的计数器 1us 累加 1 次
  // TAIE 允许定时器溢出中断

  TACCTL1 = CM_1+SCS+CCIS_0+CAP+CCIE; //上升沿触发，同步模式，使能中断
  //CM_1 输入捕获上升沿采样，捕获到输入方波上升沿 进入 0x02 捕获比较 1 中断
  //SCS   同步捕获
  //CCIS_0 输入捕获引脚选择 CCI1A
  //CAP   输入捕获模式
  //CCIE   允许 TACCR1 捕获中断
  TACCR1 = 0; //清 0 TACCR1 寄存器
}
unsigned int old_cap = 0; //上一次的捕获计数值
unsigned int period = 0;   //得到信号周期
unsigned int TA_ov_num = 0; //定时器计数溢出次数
/*
* 定时器 A 连续模式，计数器溢出中断
* CCI1A 输入捕获中断
*/
#pragma vector=TIMERA1_VECTOR
__interrupt void Timer_A(void)
{
  switch(TAIV)
  {
    case 2: //0x02 捕获比较 1 中断

      period = TA_ov_num*65536+TACCR1-old_cap;   //计算周期宽度，单位 us，这是 1M SMCLK 时钟下，如果是 2M 时钟那就用 0.5us 计算
       old_cap = TACCR1;
       TA_ov_num = 0;
       break;
    case 4:break;
    case 10:TA_ov_num++; //0x0A(10),定时器溢出中断
    break;
   }
}
```

TACCR1 寄存器什么时候启动?
当外部引脚有输入边沿信号触发捕获的时候，将定时器 TAR 这时候累加的值复制给 TACCR1 寄存器。所以 TACCR1 寄存器并没有去根据时钟累加值，而是用的 TAR 的值



如果定时器溢出2次
65536+65536
= 13072us(1M时钟1us)

不是65536整数的宽度，
差一点的宽度就用TACCR1来获取

那么第2个边沿进入中断
因为定时器溢出一次TA_ov_num++
所以TA_ov_num = 2，那么两次溢出就是2*65535 = 13072
13072 再加上第2次边沿之前得到的 差一点点宽度
定时器值TACCR1，然后减去第一次脉冲不要的宽度500us 得到周期宽度us

第1次边沿捕获中断old_cap = 0

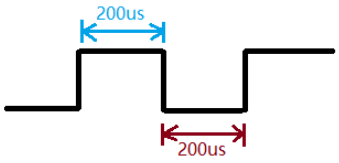
TACCR1 = 某个数(如当前TAR累加到500，那么TACCR1 = 500)

第1次如果在边沿触发前定时器没有溢出，TA_ov_num = 0

那么period周期显示500us

```
int main( void )
{
  // Stop watchdog timer to prevent time out reset
  WDTCTL = WDTPW + WDTHOLD; //关闭看门狗
  Clock_Init(); //初始化时钟

  InitUSART(); //串口初始化
  Init_cap(); //初始化捕获中断

  _EINT(); //开启总中断

  while(1)
  {
      printf("period = %d\n", period);

  }
}
```

方波高电平时间采集



高电平200us宽度采集
可以用双边沿触发
但是也会把低电平宽度采集算进来

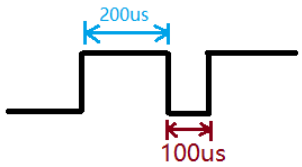
```
unsigned int old_cap = 0; //上一次的捕获计数值
unsigned int HightLevelWidht = 0;//得到信号高电平宽度
unsigned int TA_ov_num = 0; //定时器计数溢出次数

/*
* 定时器 A 初始化
*/
void Init_cap(void)
{
  P1DIR = 0x00; //P1.2 设置 IO 口为输入模式
  P1SEL  |=0x04;        //选择 P1.2 作为输入捕获功能
  TACTL  =TASSEL_2+MC_2+ID_3+TAIE;    //选择 8M - SMCLK 时钟
  //TASSEL_2 选择 SMLCK 时钟
```

```
  //MC_2 定时器连续模式
  //ID_3 输入时钟 8 分频，SMLCK/8=1M(1us) TIMRA 定时器的计数器 1us 累加 1 次
  // TAIE 允许定时器溢出中断

  TACCTL1 = CM_3+SCS+CCIS_0+CAP+CCIE;//上升沿触发，同步模式，使能中断
  //CM_3 输入捕获双边沿采样，捕获到输入方波上升沿或者下降沿 进入 0x02 捕获比较 1 中断，主要修改这句
  //SCS  同步捕获
  //CCIS_0 输入捕获引脚选择 CCI1A
  //CAP  输入捕获模式
  //CCIE  允许 TACCR1 捕获中断
  TACCR1 = 0; //清 0 TACCR1 寄存器
}
/*
* 定时器 B 初始化
*/
void Init_timer_b(void) //为了不影响主循环模拟方波输出，打印交给定时器 B
{
  TBCCTL0 = CCIE;                          // CCR0 interrupt enabled
  TBCCR0 = 32768;                          //定时 1 秒
  TBCTL = TBSSEL_1 + MC_1;                  // ACLK, upmode
}

/*
* 定时器 A 连续模式，计数器溢出中断
* CCI1A 输入捕获中断
*/
#pragma vector=TIMERA1_VECTOR
__interrupt void Timer_A(void)
{
  switch(TAIV)
  {
    case 2: //0x02 捕获比较 1 中断

      HightLevelWidht = TA_ov_num*65536+TACCR1-old_cap;
      old_cap = TACCR1;
      TA_ov_num = 0;
      break;

    case 4:break;
    case 10:TA_ov_num++; //0x0A(10),定时器溢出中断
    break;
   }
}
#pragma vector=TIMERB0_VECTOR
__interrupt void Timer_B (void)
{

     printf("Widht = %d\n",HightLevelWidht); //用来做 1 秒的方波电平宽度打印

}
int main( void )
{
  // Stop watchdog timer to prevent time out reset
  WDTCTL = WDTPW + WDTHOLD; //关闭看门狗
  Clock_Init(); //初始化时钟
```

```
  InitUSART(); //串口初始化
  Init_cap(); //初始化捕获中断
  Init_timer_b(); //初始化定时器 B CCR0 中断

  _EINT(); //开启总中断
  P6DIR = 0xFF; //P6 端口 0~7 个 IO 全部为输出
  P6OUT = 0x00; //点亮全部 LED

  while(1)
  {
    /*IO 口模拟输出 200us 的高电平，200us 低电平方波*/
    P6DIR = 0xFF; //P6 组全输出高电平
    delay_us(200);
    P6DIR = 0x00; //P6 组全输出低电平
    delay_us(200);

  }
}
```

```
Widht = 207
Widht = 208
Widht = 207
Widht = 207
Widht = 207
Widht = 208
Widht = 207
```

207us 高电平宽度和低电平宽度。主要是驱动了 LED 所以有几个 us 误差。



如果高电平是200us

低电平是100us

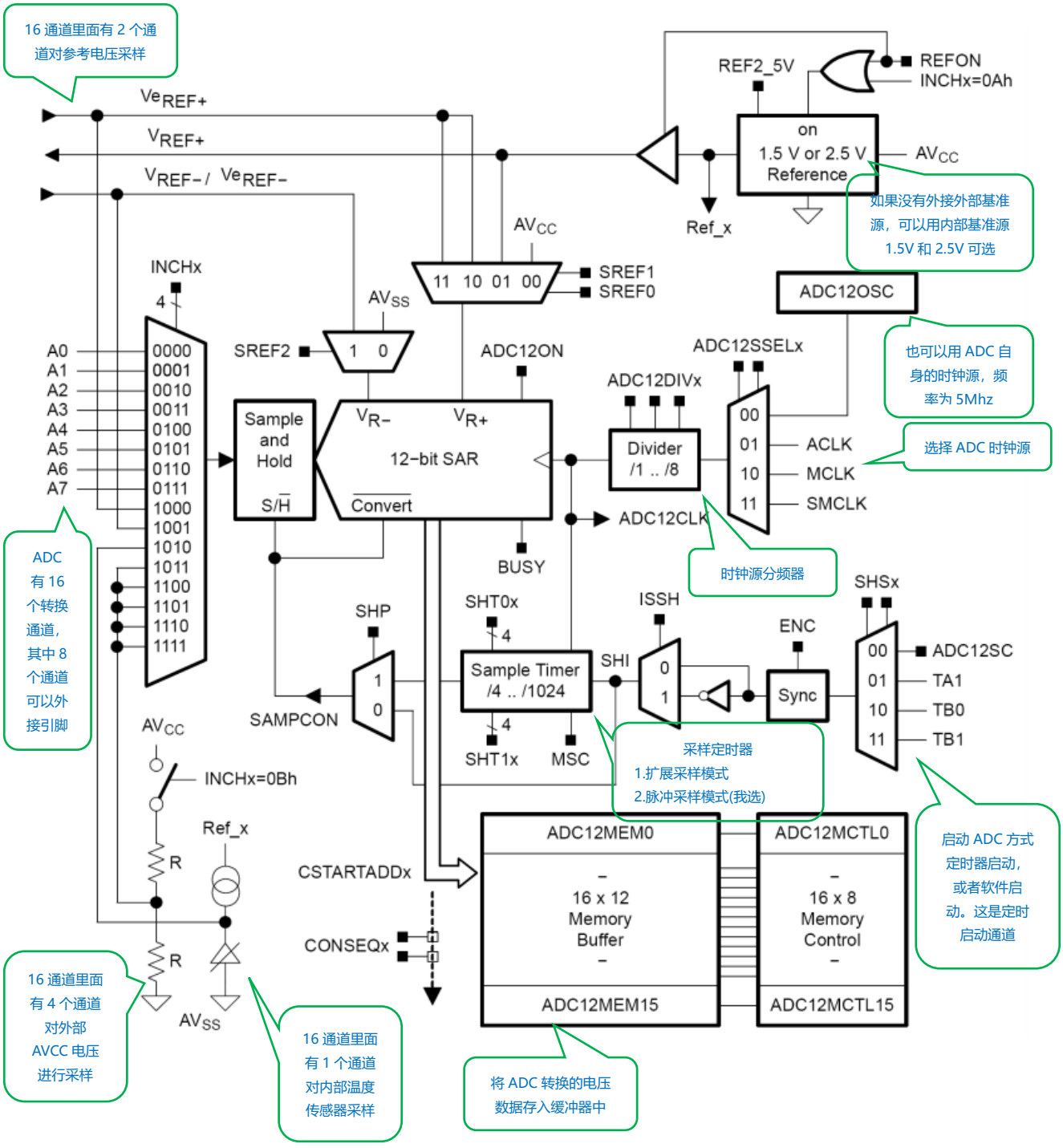
双边沿采集会是什么结果?

```
while(1)
  {
    /*IO 口模拟输出 200us 的高电平，100us 低电平方波*/
    P6DIR = 0xFF; //P6 组全输出高电平 200us
    delay_us(200);
    P6DIR = 0x00; //P6 组全输出低电平 100us
    delay_us(100);
  }
```

```
Widht = 207
Widht = 108
Widht = 108
Widht = 207
Widht = 107
Widht = 107
Widht = 208
Widht = 107
Widht = 207
Widht = 108
```

发现采集的电平宽度有长，有短，107us 是低电平宽度，207 是高电平宽度

ADC 使用



$$N_{ADC} = 4095 \times \frac{Vin - V_{R-}}{V_{R+} - V_{R-}}$$

VR+:是基准源正极, VR-:是基准源负极，Vin 是模拟通道输入电压。

如果基准源是正 1.5V，那么 VR- = 0V, VR+ = 1.5V, 如果 Vin 接地(0V),那么 Nadc 数字量为 0

如果基准源是正 1.5V，那么 VR- = 0V, VR+ = 1.5V, 如果 Vin=1.5V,那么 Nadc 数字量为 4095(FFF)

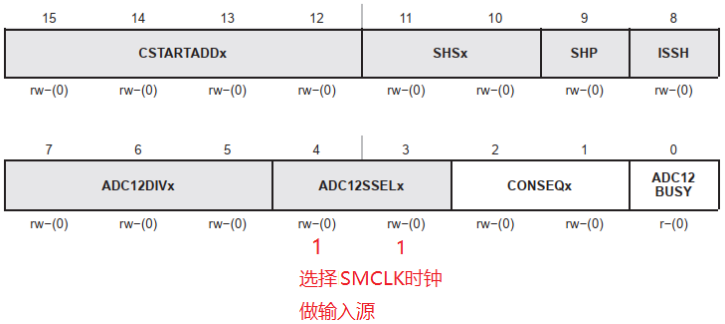
P6.0/A0	59	I/O	General-purpose digital I/O pin Analog input A0 for ADC
P6.1/A1	60	I/O	General-purpose digital I/O pin Analog input A1 for ADC
P6.2/A2	61	I/O	General-purpose digital I/O pin Analog input A2 for ADC
P6.3/A3	2	I/O	General-purpose digital I/O pin Analog input A3 for ADC
P6.4/A4	3	I/O	General-purpose digital I/O pin Analog input A4 for ADC
P6.5/A5	4	I/O	General-purpose digital I/O pin Analog input A5 for ADC
P6.6/A6	5	I/O	General-purpose digital I/O pin Analog input A6 for ADC
P6.7/A7	6	I/O	General-purpose digital I/O pin Analog input A7 for ADC

P6 引脚是 ADC 8 个通道模拟输入引脚

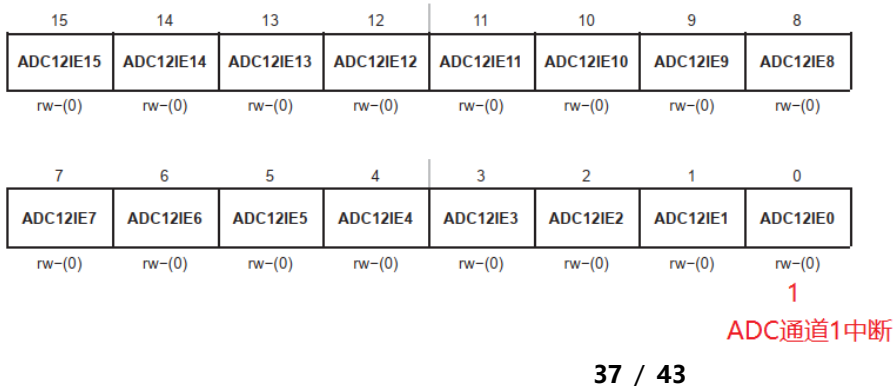
ADC12CTL0, ADC12 Control Register 0

采样保持时间越长，数据波动越小，数据值越稳定。采样保持时间越短，数据波动大，数据看起变动很大。

ADC12CTL1, ADC12 Control Register 1

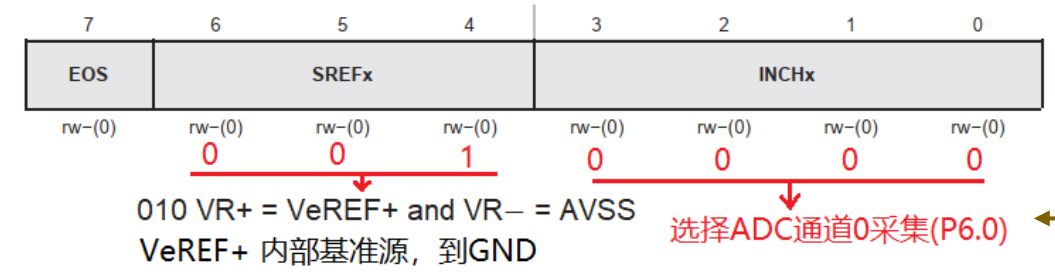


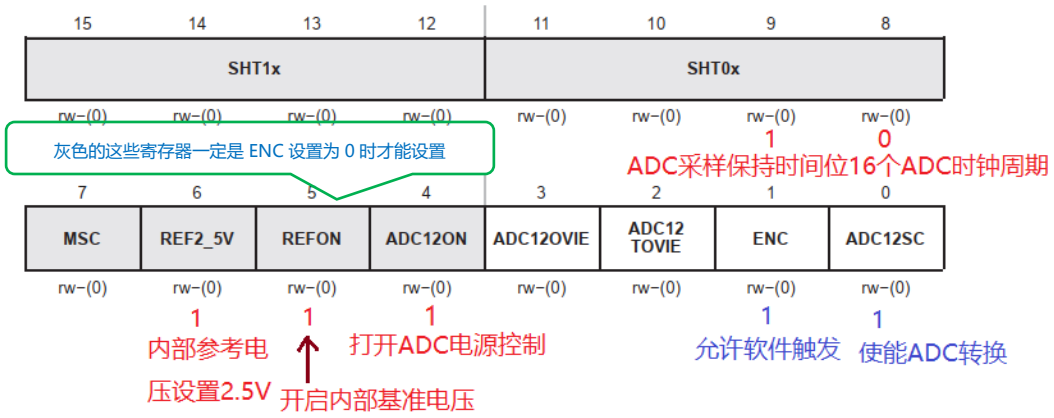
ADC12IE, ADC12 Interrupt Enable Register



```
void ADC_Init()
{
  P6SEL|=0x01;                                  //选择 ADC 通道 0，P6.0 /A0 功能模式
  ADC12CTL0|= ADC12ON + SHT0_2 + REF2_5V + REFON; //ADC 电源控制开，16 个 CLK，内部基准 2.5V
  ADC12CTL1|= ADC12SSEL1 + ADC12SSEL0;          //SMCLK 做时钟源
  ADC12MCTL0= SREF0 + INCH_0;                   //参考控制位及通道选择，这里选择通道 0
  ADC12IE|= 0x01;                               //中断允许  ADC 通道 1 中断
  ADC12CTL0|= ENC + ADC12SC;                    //使能转换器
}
```

ADC12MCTLx, ADC12 Conversion Memory Control Registers





单次 ADC 转换，使用 P6.0/A0 通道进行测试，

单次 ADC 转换适用于多路模拟开关采集多个模拟量，用模拟开关切换开关的方式送入同一个 AD 采集通道。

```
/** ADC 采集通道 0 数据
*/
void ADC_Init()
{
  P6SEL|=0x01;                                  //选择 ADC 通道 0,P6.0/A0 功能模式
  ADC12CTL0|= ADC12ON + SHT0_2 + REF2_5V + REFON; //ADC 电源控制开，16 个 CLK，内部基准 2.5V
  ADC12CTL1|= ADC12SSEL1 + ADC12SSEL0;          //SMCLK 做时钟源
  ADC12MCTL0= SREF0 + INCH_0;                   //参考控制位及通道选择，这里选择通道 0
  ADC12IE|= 0x01;                               //中断允许
  ADC12CTL0|= ENC + ADC12SC;                    //使能转换器
}
unsigned int TEMP = 0;
//*****************************************************************************
//      ADC 中断服务程序
//*****************************************************************************
#pragma vector=ADC_VECTOR
__interrupt void ADC12ISR(void)
{
  while((ADC12CTL1&0x01)==1);                   //如果 ADC 忙，则等待，否则读取 ADC 转换数值
  TEMP = ADC12MEM0;                             //读取 ADC 转换值
}
```

```
int main( void )
{
  // Stop watchdog timer to prevent time out reset
  WDTCTL = WDTPW + WDTHOLD; //关闭看门狗
  Clock_Init(); //初始化时钟
  InitUSART(); //串口初始化
  ADC_Init();
  _EINT(); //开启总中断

  printf("xxxzzzzz\n");

          //开启转换
  while(1)
  {
    ADC12CTL0 |= ADC12SC; //因为是单次 ADC 转换，所以每次转换要启动一次 ADC，然后在中断中获取 AD 值
    printf("TEMP = %d\n",TEMP);
    delay_ms(1000);
    ADC12CTL0 &= ~ADC12SC;            //因为是单次转换，必须转换之后要清 0
  }
}
```

```
xxxzzzzz
TEMP = 0
TEMP = 0
TEMP = 0
TEMP = 0
TEMP = 775
TEMP = 775
TEMP = 4095
TEMP = 4095
TEMP = 4095
TEMP = 4095
TEMP = 4095
TEMP = 4095
TEMP = 4095
TEMP = 4095
TEMP = 2725
TEMP = 2725
TEMP = 580
TEMP = 580
TEMP = 0
TEMP = 0
TEMP = 0
TEMP = 0
TEMP = 0
TEMP = 0
```

开机发现 ADC 没采集到数据，这是因为用的延时，并不清楚 ADC 什么时候转换完成。所以最好在中断加标志位来处理

过几秒才发现 AD 数据采集进来，这是因为中断触发了

下面给 ADC 加入中断标志位

```
unsigned int TEMP = 0;
unsigned char ADCflag = 0; //给 ADC 加入中断标志位
//*****************************************************************************
//      ADC 中断服务程序
//*****************************************************************************
#pragma vector=ADC_VECTOR
__interrupt void ADC12ISR(void)
{

  while((ADC12CTL1&0x01)==1);          //如果 ADC 忙，则等待，否则读取 ADC 转换数值
  TEMP = ADC12MEM0;                    //读取 ADC 转换值
  ADCflag = 1;
}


int main( void )
{
  // Stop watchdog timer to prevent time out reset
  WDTCTL = WDTPW + WDTHOLD; //关闭看门狗
  Clock_Init(); //初始化时钟
```

```
  InitUSART(); //串口初始化
  ADC_Init();
  _EINT(); //开启总中断
  printf("xxxzzzzz\n");
  ADCflag = 1;//开启第 1 次 ADC 转换

  while(1)
  {
    while(ADCflag == 1)
    {
      ADC12CTL0 |= ADC12SC; //因为是单次 ADC 转换，所以每次转换要启动一次 ADC，然后在中断中获取 AD 值
      ADC12CTL0 &= ~ADC12SC;              //因为是单次转换，必须转换之后要清 0
      ADCflag = 0;
      printf("TEMP = %d\n",TEMP);
       delay_ms(500);
      //ADC12CTL0 |= ADC12SC; //因为是单次 ADC 转换，所以每次转换要启动一次 ADC，然后在中断中获取 AD 值
    }
  }
}
```

使能 SHP，使 ADC 转换结束后自动清 0

```
unsigned int TEMP = 0;
unsigned char ADCflag = 0; //给 ADC 加入中断标志位
//*****************************************************************************
//      ADC 中断服务程序
//*****************************************************************************
#pragma vector=ADC_VECTOR
__interrupt void ADC12ISR(void)
{

  while((ADC12CTL1&0x01)==1);              //如果 ADC 忙，则等待，否则读取 ADC 转换数值
  TEMP = ADC12MEM0;                          //读取 ADC 转换值
  ADCflag = 1;
}

/*
* ADC 采集通道 0 数据
*/
void ADC_Init()
{
  P6SEL|=0x01;                                      //选择 ADC 通道 0,P6.0/A0 功能模式
  ADC12CTL0|= ADC12ON + SHT0_2 + REF2_5V + REFON; //ADC 电源控制开，16 个 CLK，内部基准 2.5V
  ADC12CTL1|= ADC12SSEL1 + ADC12SSEL0 + SHP;              //SMCLK 做时钟源，SHP AD 转换结束后自动清 0 ADC12SC,
  ADC12MCTL0= SREF0 + INCH_0;                       //参考控制位及通道选择，这里选择通道 0
  ADC12IE|= 0x01;                                   //中断允许
  ADC12CTL0|= ENC + ADC12SC;                        //使能转换器
}

int main( void )
{
  WDTCTL = WDTPW + WDTHOLD; //关闭看门狗
  Clock_Init(); //初始化时钟
  InitUSART(); //串口初始化
  ADC_Init();
  _EINT(); //开启总中断
```

```
    printf("xxxzzzzz\n");
    ADCflag = 1;//开启第 1 次 ADC 转换
    while(1)
    {
      ADC12CTL0 |= ADC12SC; //因为是单次 ADC 转换，所以每次转换要启动一次 ADC，然后在中断中获取 AD 值
      while(ADCflag == 1)
      {
//      ADC12CTL0 &= ~ADC12SC;              //加入了 SHP 不需要手动清 0
        ADCflag = 0;
        printf("TEMP = %d\n",TEMP);
         delay_ms(500);
      }
    }
}
```

```
xxxzzzzz
TEMP = 1695
TEMP = 1774
TEMP = 2039
TEMP = 2039
TEMP = 1927
TEMP = 1971
TEMP = 2335
TEMP = 2140
TEMP = 4095
TEMP = 4095
TEMP = 4095
TEMP = 4095
TEMP = 2525
TEMP = 2013
TEMP = 2303
TEMP = 40
TEMP = 34
TEMP = 34
TEMP = 30
TEMP = 34
```

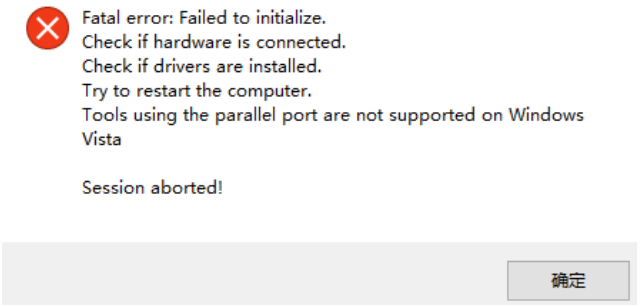
空气电压采集正常

高电平采集正常

低电平

ADC 死循环查询法采集数据

IAR 经过 TI JTAG 仿真器下载，如果某一天出现 Fatal error: Failed to initialize. Check if hardware is connected. Check if drivers are installed. Try to restart the computer. Tools using the parallel port are not supported on Windows Vista Session aborted!



这不是操作问题，请把电脑的杀毒软件，电脑管家类的东西退出。就能正常下载了。

MSP430 因为是 16 位单片机，函数返回值必须满足 16 位。如果返回值是 32 位数据类型，就会有问题（重点）

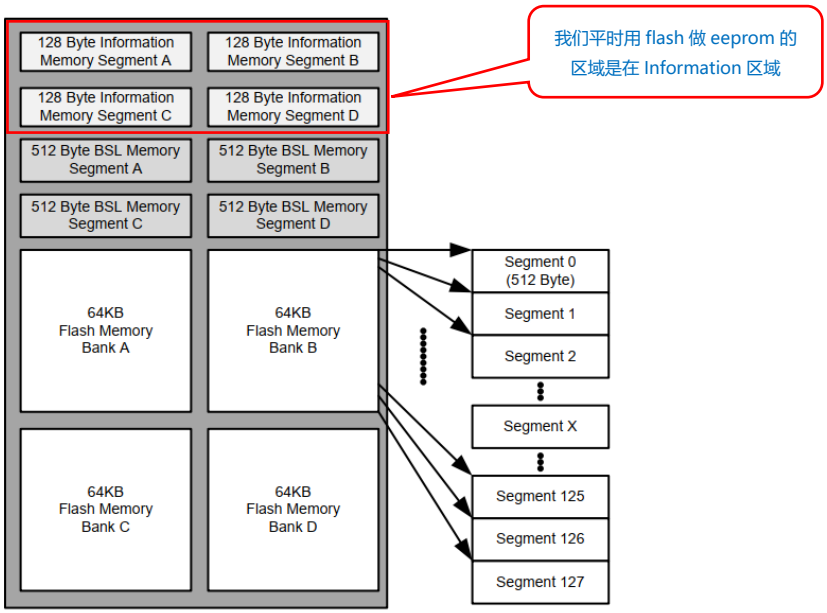
```
float Get()
{
        float data = 100;
        return data; //正常 32 位的单片机应该返回 100，正常
}

int main()
{
        float ret = 0;
        ret = Get();
        printf( “ret = %f\n” , ret); //我们发现 ret 得到的数据位 0
}
```

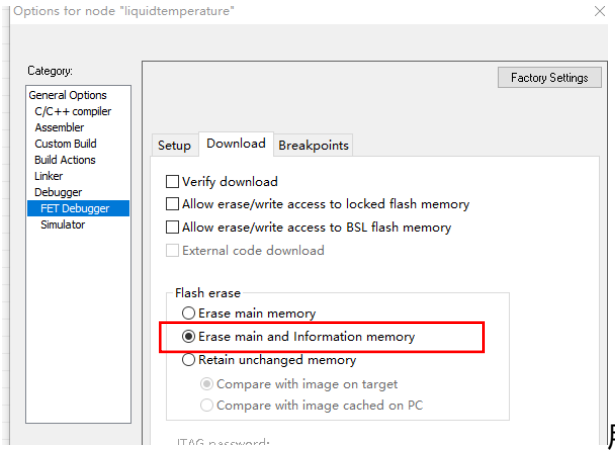
这就是 16 位单片机，函数返回值不能超过 16 位，所以只有返回 int，或者 char。

所以可以用指针做形参的方式，取地址，修改传入的变量。

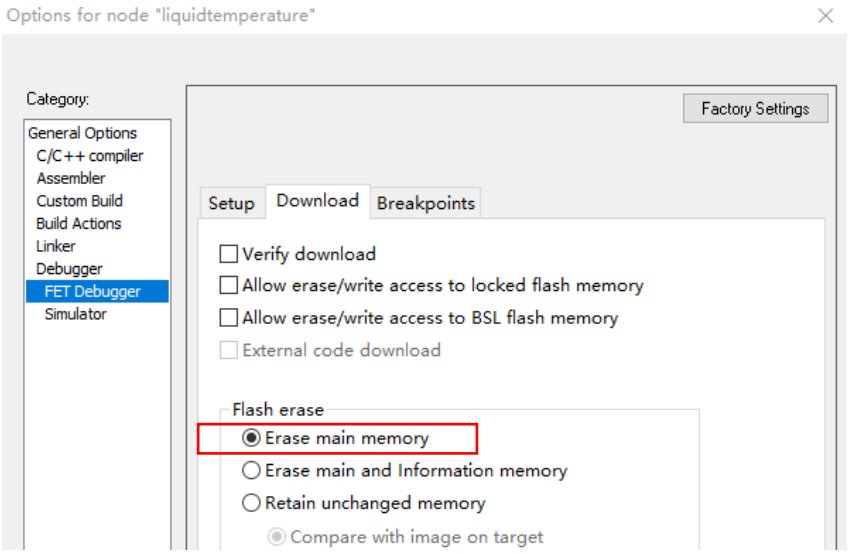
经过测试发现数据类型 long 做返回值没有问题。为什么 double 和 float 返回有问题呢?

MSP430 内部 flash 模拟 EEPROM，如何防止已经存入的数据，在第二次用 JTAG 下载的时候不会被清空





用 IAR 创建的工程，默认是下载程序的时候将 flash 中的 main 主程序区和 information 区域全部擦除掉



现在改成下载程序的时候只擦除 main 主程序区，这样你上次在 information 区域保存的设备数据就不会被擦除。做到 eeprom 永久保存功能。