

STM8 操作指南

作者：向仔州

下载 IAR 开发环境.....	2
编写一个 RGB led 程序.....	8
IAR 硬件仿真 STM8.....	9
STM8 时钟初始化.....	11
STM8 中断处理.....	13
UART 串口编程.....	17
IAR for STM8 软件多个 C 文件编译.....	24
单片机 printf 实现.....	27
STM8 延时函数.....	30
STM8 CAN 总线使用.....	31
STM8 库函数工程建立.....	50
STM8 寄存器操作 CAN 在波特率上有些问题，所以要用到 STM8 库函数方案， 实现库函数寄存器双用的工程建立.....	55
CAN 总线寄存器和库函数中断接收混合编程.....	58
CAN 总线固件库函数屏蔽滤波功能.....	59
STM8CAN 总线固件库函数波特率计算.....	60

下载 IAR 开发环境

进入 ST 官网, 下载的是 IAR2.20 版本

The screenshot shows a product listing for 'IAR-EWSTM8'. The 'GO TO SITE' button is highlighted with a callout: '点击 GO TO SITE 到 IAR 网站'. Below it, the 'IAR EMBEDDED WORKBENCH' logo is shown with another callout: '点击 IAR EMBEDDED WORKBENCH'. At the bottom right, there's a 'Free trials' button with a callout: '点击 Free trials'.

Part Number	Marketing Status	Supplier	Software Version	Third Party Link
IAR-EWSTM8	Active	IAR		GO TO SITE

[Overview](#) [Release notes](#) [What's included?](#) [Free trials](#)

然后不要被第一个标题蒙住了，网页下面还有很多

The screenshot shows the 'IAR Embedded Workbench for STM8' download page. A callout points to the 'Download Software' button: '点击 STM8 + 号'. Another callout points to the 'Download Software' button: '下载'.

[H8](#) IAR Embedded Workbench for H8

[STM8](#) IAR Embedded Workbench for STM8

[COLD FIRE](#) IAR Embedded Workbench for Coldfire

[Download Software](#)

The evaluation license is completely free of charge and allows you to try the integrated development environment and evaluate its efficiency and ease of use. When you first run the product, you will be asked to register to get your evaluation license. After download and installation, you can use the software for up to 30 days. During this time, you will have:

- a 30-day time-limited but fully functional license
- a size-limited Kickstart license without any time limit

Restrictions to the 30-day time-limited evaluation

- Source code for runtime libraries is not included.
- No support for MISRA C.
- Limited technical support.
- This license may not be used for product development or any other kind of commercial use.

IAR 环境安装

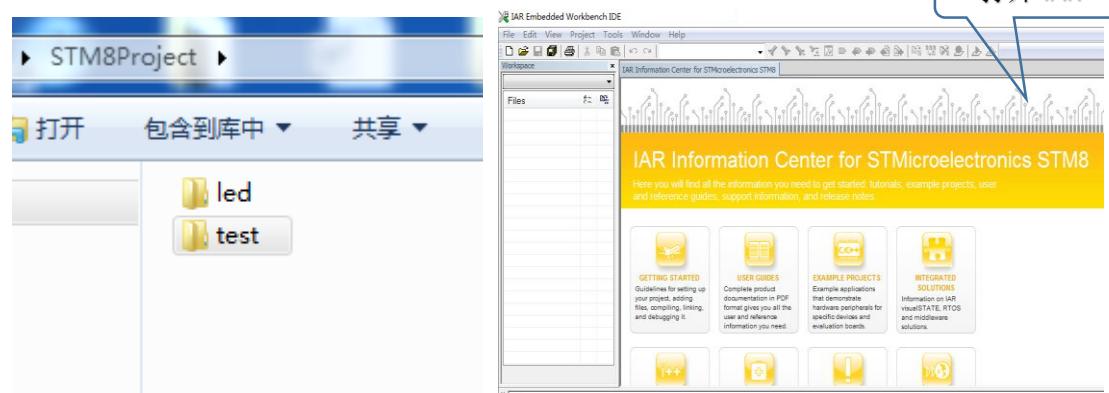
安装方法请在这个网站上去看

<http://blog.csdn.net/ybhuangfugui/article/details/52936636>

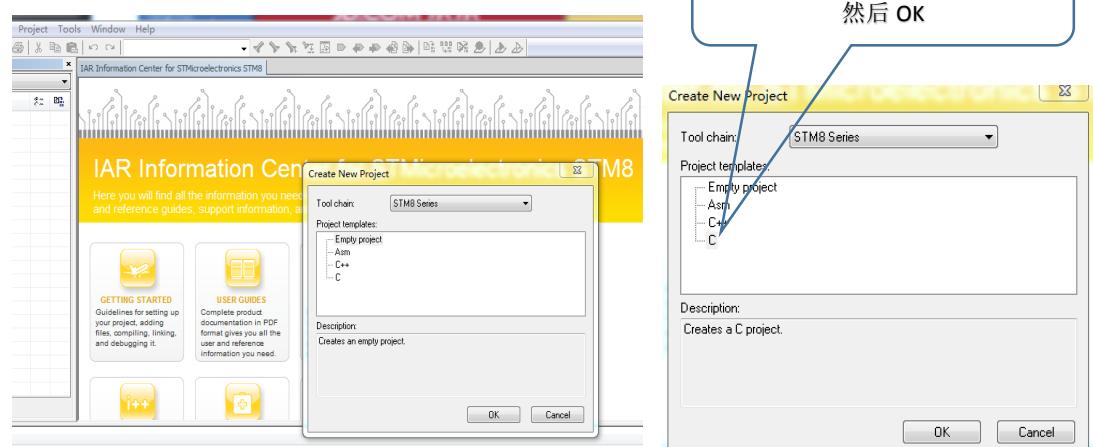
注册机在 github 的 STM8 目录下 IAR for STM8 2.10 破解.rar 压缩包

创建 STM8 工程

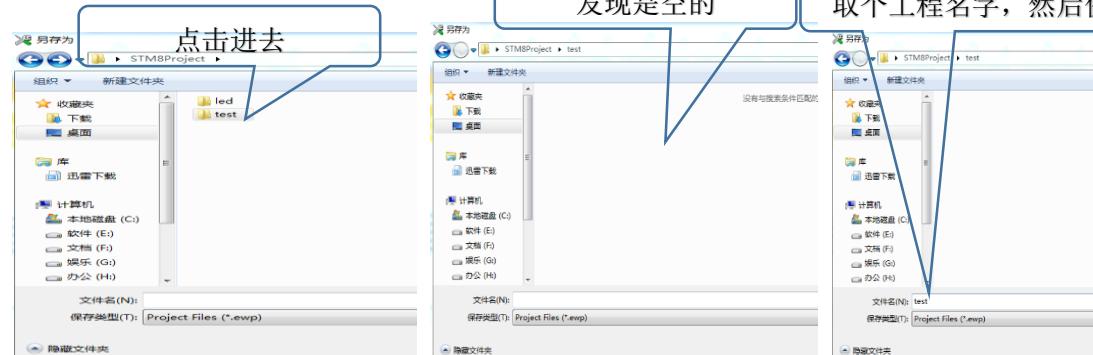
在桌面或者什么地方创建你个目录，比如我要测试 STM8 基本功能，我可以给这个目录取个名字叫 test



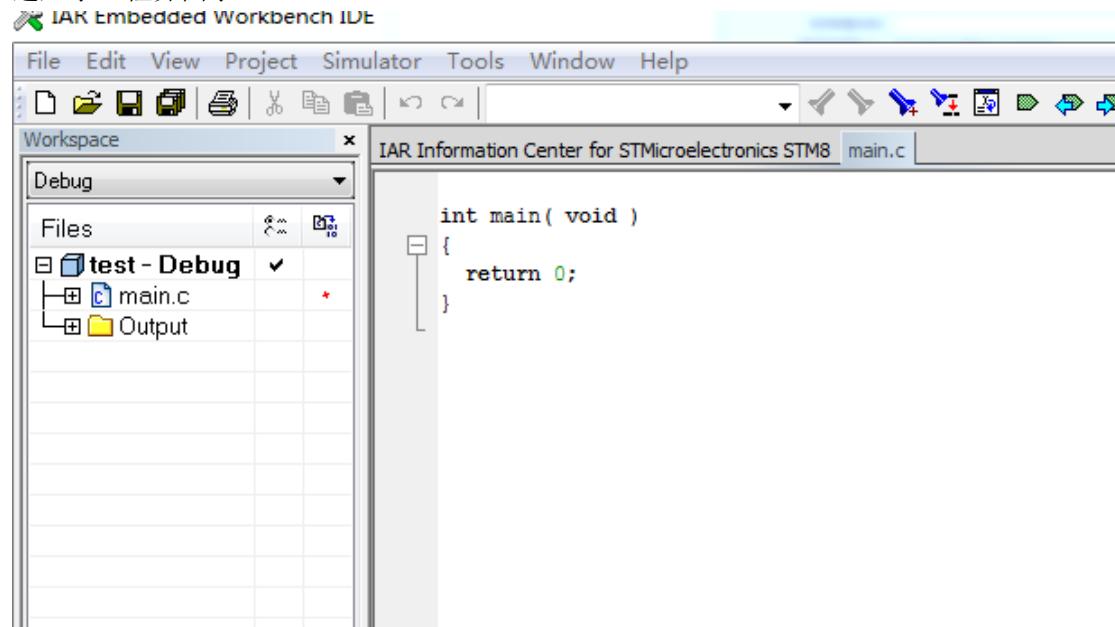
点击 project->Create New Project



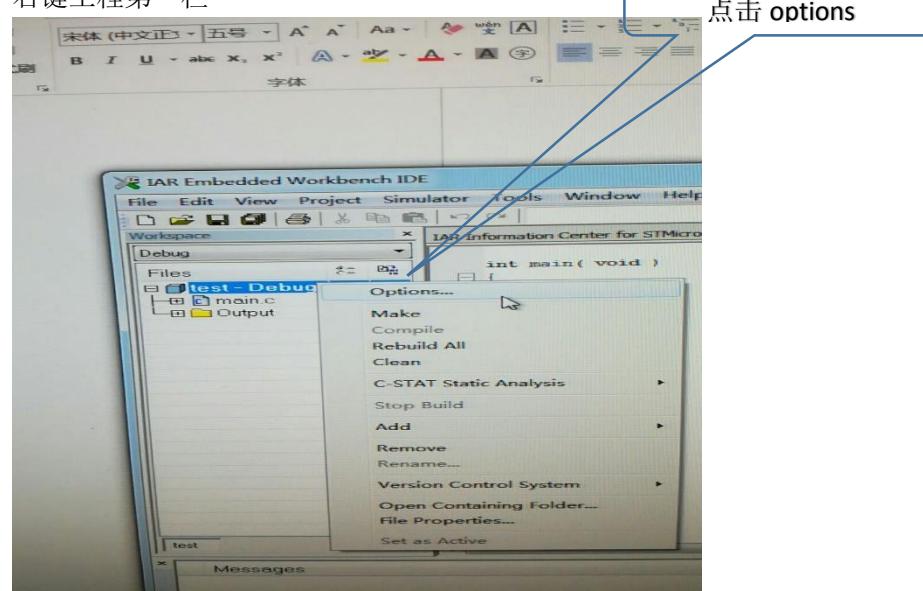
选择你上面建立的目录



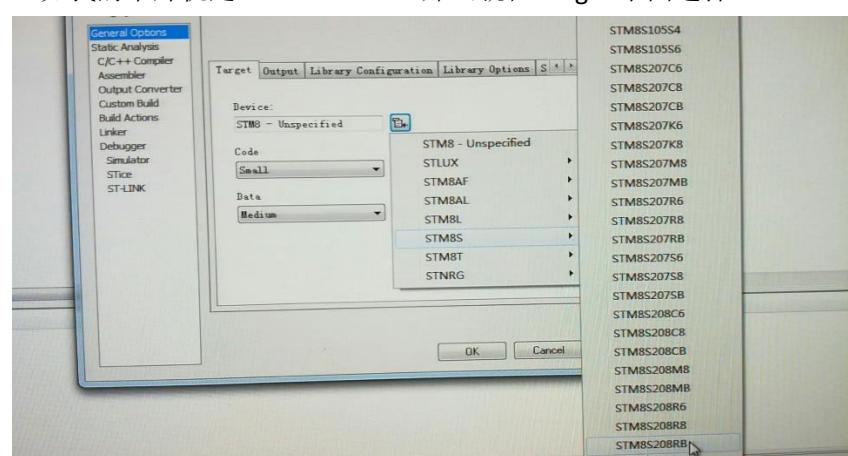
进入了工程界面了



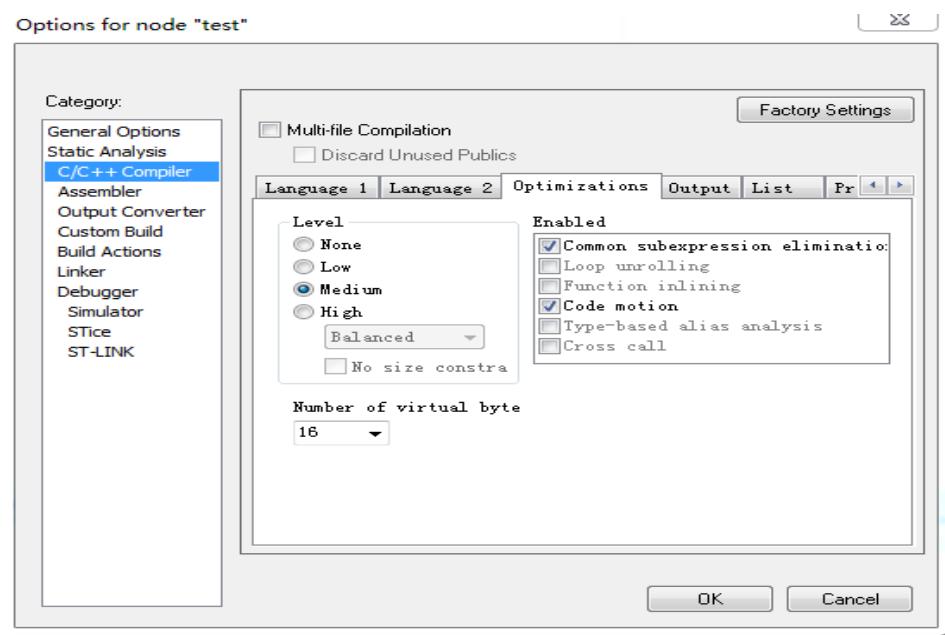
我们单片机型号参数这些都没有选择，怎么操作这个工程呢？所以我们要点击选项右键工程第一栏



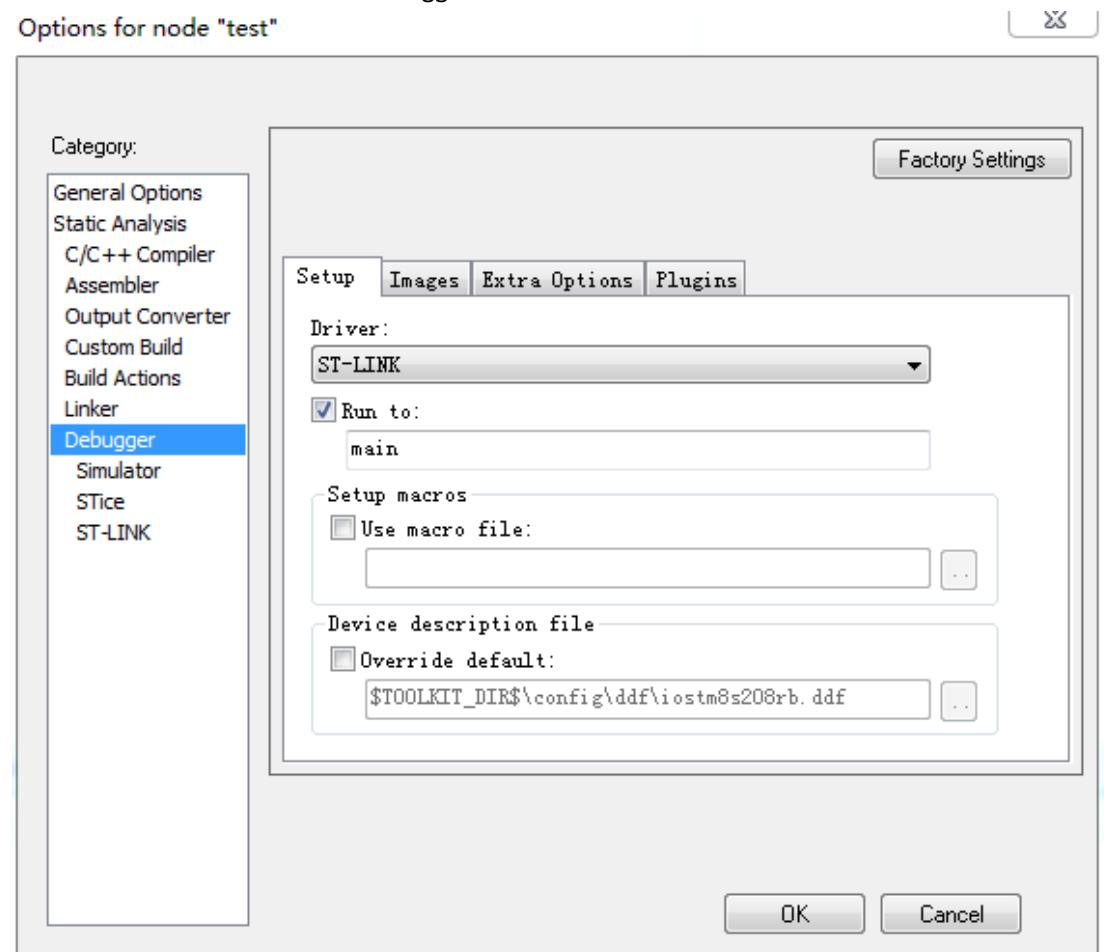
比如我的单片机是 STM8S208RB，那么就在 Target 下面选择



然后再 C/C++ 编译器优化等级里面选择 medium



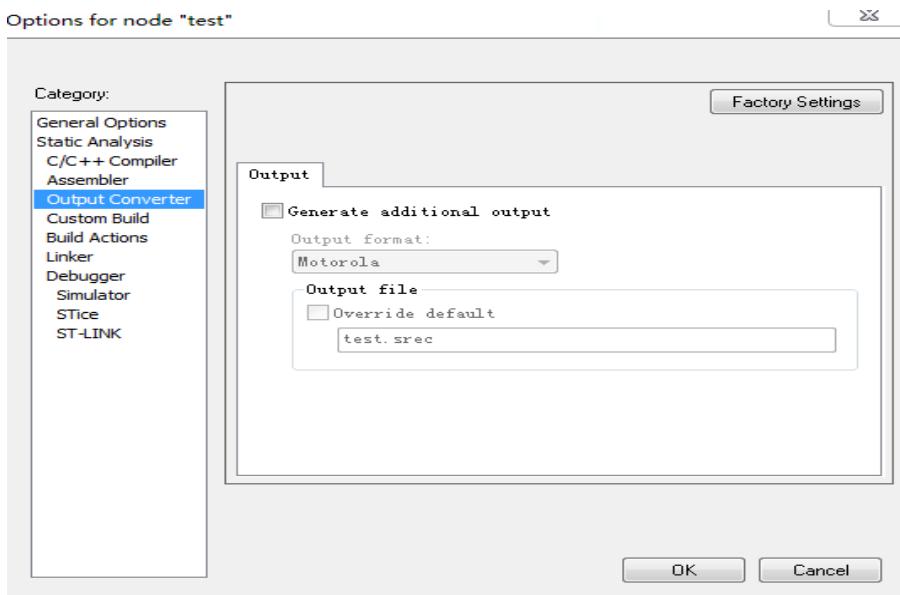
如果你有 ST-link 工具你可以 Debugger 里面 Driver 里选择 ST-LINK



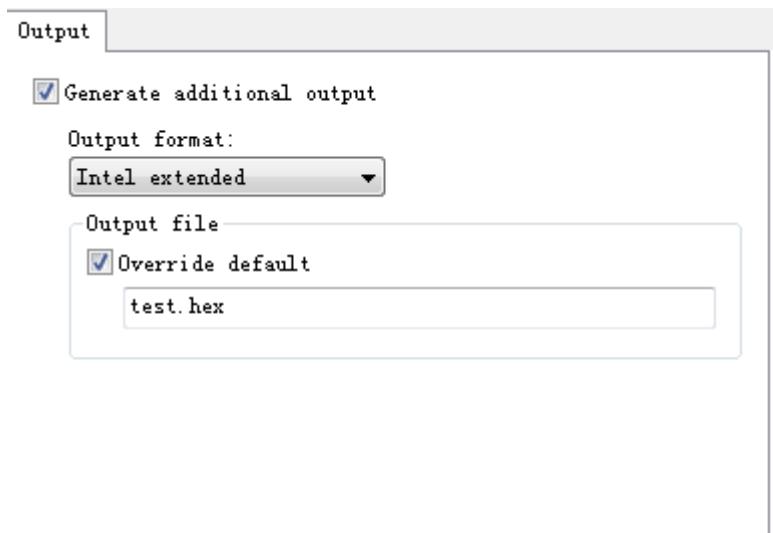
如果没有 ST-LINK 工具你可以不选择，然后点击 OK

然后点击 IAR 的 指定位置保存整个工程

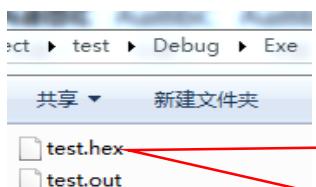
如何输出 hex 文件呢？在 Options->Output Converter



选择生成 hex 文件



然后再进行工程文件编译，在工程文件下的 Debug 里的 Exe 文件夹下就生成了 hex 文件



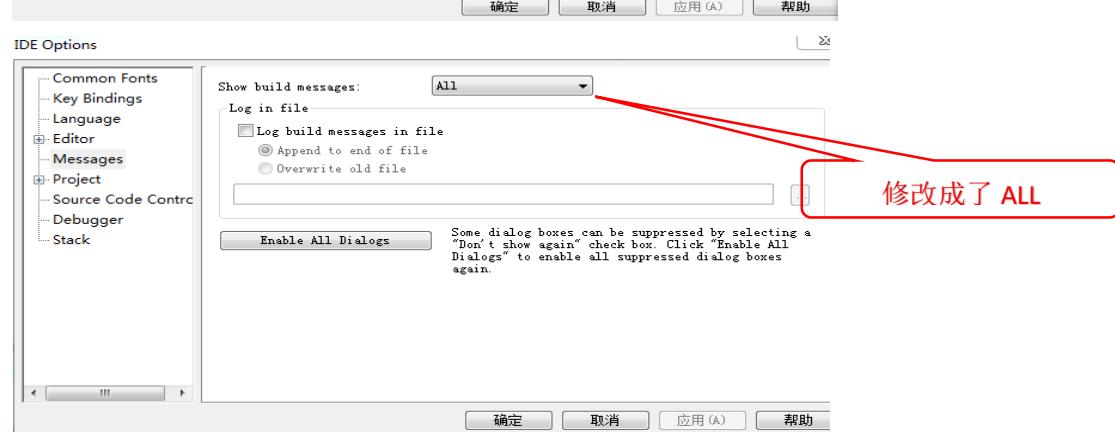
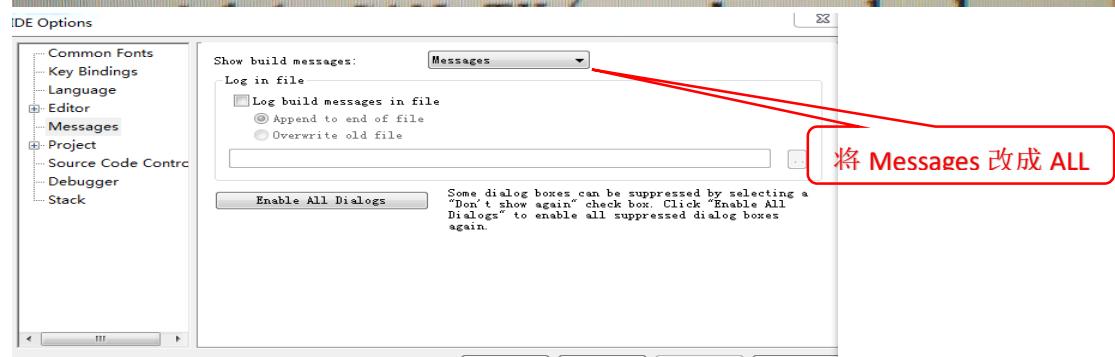
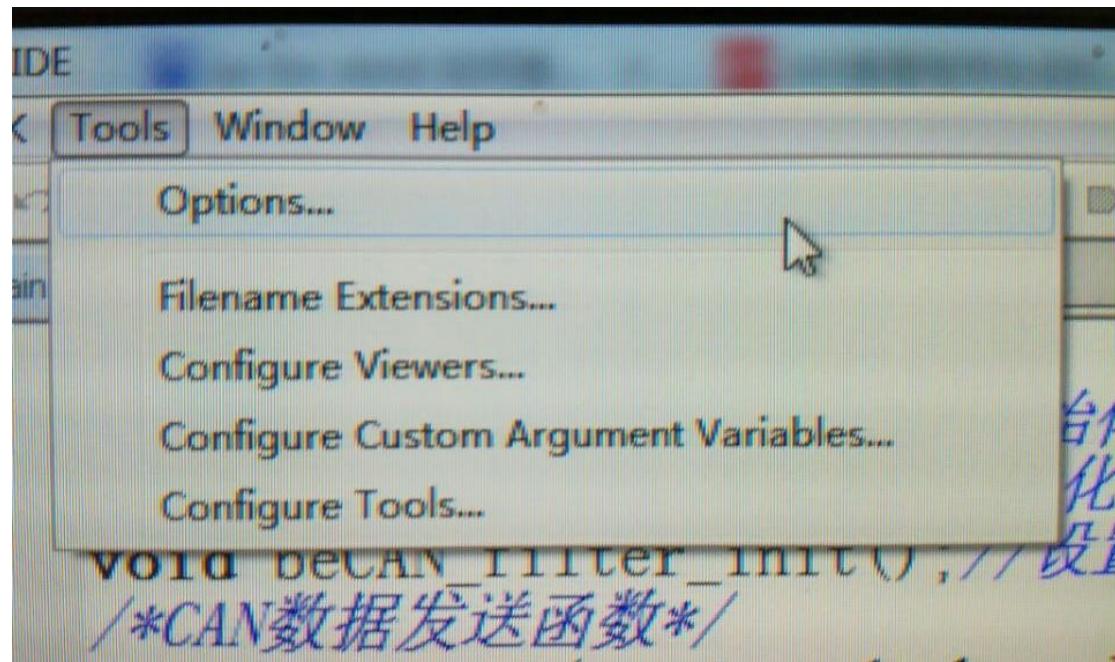
这里一定要注意，你的 IAR 如果只是编译代码通过，那么你编译的代码是不会覆盖 hex 文件的，你 hex 下载进单片机的还是先前的程序，一定要点击 make，才会形成新的 hex 程序

在 IAR 中 STM8 头文件和 STVD 里面的头文件名字不一样，STVD 头文件是 #include<STM8S208RB.h> 但是在 IAR 中是#include<iostm8s208rb.h> 头文件在安装 IAR 软件目录下



STM8 IAR-IDE 设置查看编写代码占用内存大小

IAR 编译器默认在编译的时候是不显示代码大小的



Copyright 2010-2015 IAR Systems AB.

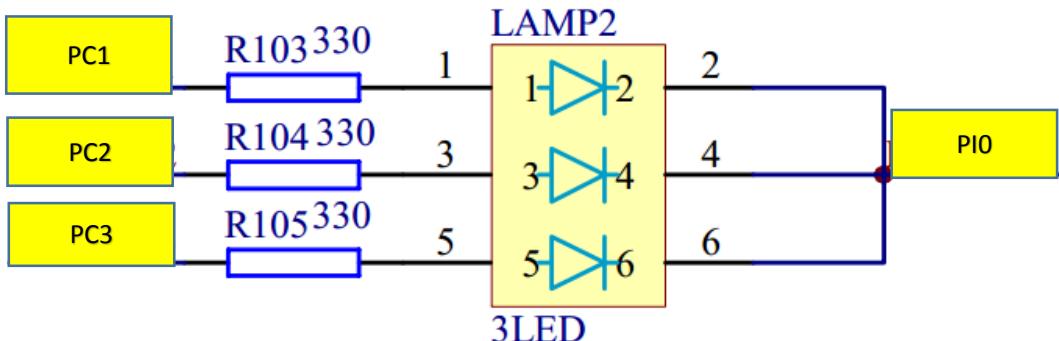
8 803 bytes of readonly code memory
488 bytes of readonly data memory
714 bytes of readwrite data memory (+ 31 absolute)

再次编译就看到代码容量了

Errors: none
Warnings: none

C:\Users\zhang\Documents\IAR\STM8\STM8\main.c 0.00 /CPU:P 0.02 /T=16MHz

编写一个 RGB led 程序



```
#include <iostm8s208rb.h>
int main( void )
{
    PC_DDR = 0x0E;      //PC3 ~ PC1为输出0000 1110
    PC_CR1 = 0x0E;      //0000 1110
    PC_CR2 = 0x00;

    PI_DDR = 0x01;
    PI_CR1 = 0x01;
    PI_CR2 = 0x00;

    PI_ODR = 0x00; //PIO输出低电平
    PC_ODR = 0x08; //PC1输出高电平
    while(1);
}
```

代码分析:

`PC_DDR = 0x0E; //PC3 ~ PC1 为输出 0000 1110 设置 IO 口方向`

11.9.3 端口 x 数据方向 (Px_DDR)

地址偏移值: 0x02

复位值: 0x00

7	6	5	4	3	2	1	0
rw 0	rw 0	rw 0	rw 0	rw 1	rw 1	rw 1	rw 0
位7:0 DDR[7:0]: 数据方向寄存器位 这些位可通过软件置1或置0，选择引脚输入或输出 0: 输入模式 1: 输出模式							

`PC_CR1 = 0x0E; //0000 1110 设置输出用哪种模式`

11.9.4 端口 x 控制寄存器 1 (Px_CR1)

地址偏移值: 0x03

复位值: 0x00

7	6	5	4	3	2	1	0
C17	C16	C15	C14	C13	C12	C11	C10

位7:0	C1[7:0] 控制寄存器位 这些位可通过软件置1或置0，用来在输入或输出模式下选择不同的功能。请参考表18 在输入模式时(DDR=0): 0: 浮空输入 1: 带上拉电阻输入 在输出模式时(DDR=1): 0: 模拟开漏输出(不是真正的开漏输出) 1: 推挽输出,由CR2相应的位做输出摆率控制
------	--

PC_CR2 = 0x00; 设置输出速度 2M

11.9.5 端口 x 控制寄存器 2 (Px_CR2)

地址偏移值: 0x04

复位值: 0x00

7	6	5	4	3	2	1	0
C27	C26	C25	C24	C23	C22	C21	C20

位7:0	C2[7:0] 控制寄存器位 相应的位通过软件置1或置0，用来在输入或输出模式下选择不同的功能。在输入模式下，由CR2相应的位使能中断。如果该引脚无中断功能，则对该引脚无影响。 在输出模式下，置位将提高IO速度。此功能适用O3和O4输出类型。(参见引脚描述表) 在 输入模式时(DDR=0): 0: 禁止外部中断 1: 使能外部中断 在 输出模式时(DDR=1): 0: 输出速度最大为2MHZ. 1: 输出速度最大为10MHZ
------	---

PC_ODR = 0x02; //PC1 输出高电平

11.9.1 端口 x 输出数据寄存器 (Px_ODR)

地址偏移值: 0x00

复位值: 0x00

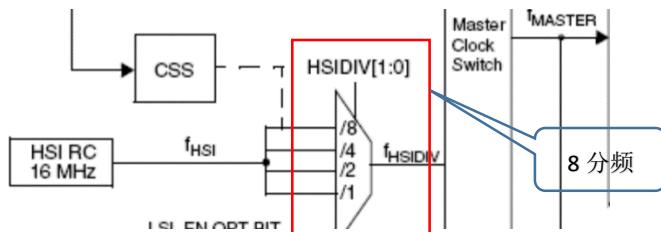
7	6	5	4	3	2	1	0
ODR7	ODR6	ODR5	ODR4	ODR3	ODR2	ODR1	ODR0

位7:0	ODR[7:0] : 端口输出数据寄存器位 在输出模式下，写入寄存器的数值通过锁存器加到相应的引脚上。读ODR寄存器，返回之前锁存的寄存器值。 在输入模式下，写入ODR的值将被锁存到寄存器中，但不会改变引脚状态。ODR寄存器在复位后总是为0。位操作指令(BSET, BRST) 可以用来设置DR寄存器来驱动相应的引脚，但不会影响到其他引脚。
------	---

IAR 硬件仿真 STM8

```
void delay_ms(unsigned int ms)
{
    unsigned int x,y;
    for(x=ms;x>0;x--)
    {
        for(y=300;y>0;y--)
        {
        }
    }
}
```

因为我们没有在主函数初始化 STM8 的外部时钟，
所以 STM8 CPU 自动打开了内部 RC 时钟，然后经过 8 分频，给 CPU 供时钟。所以这个延时函数是
STM8 内部时钟为参考设计的，内部时钟为 2M



```

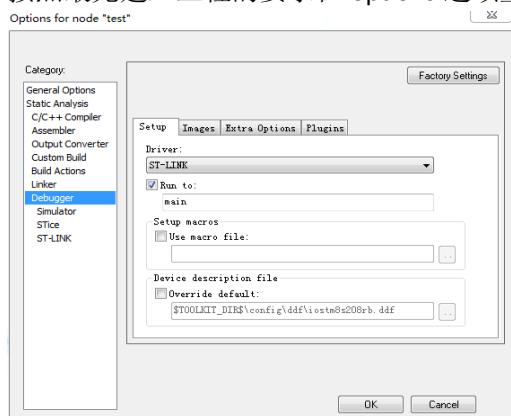
PI_DDR = 0x01;
PI_CR1 = 0x01;
PI_CR2 = 0x00;

PI_ODR = 0x00; //PIO输出低电平
PC_ODR = 0x02; //PC1输出高电平
while(1)
{
    PC_ODR = 0x02; //PC1输出高电平
    delay_ms(300); //因为没有初始化时钟，所以我们使用的是内部2M时钟
    PC_ODR = 0x04; //PC2输出高电平
    delay_ms(300);
    PC_ODR = 0x08; //PC3输出高电平
    delay_ms(300);
}

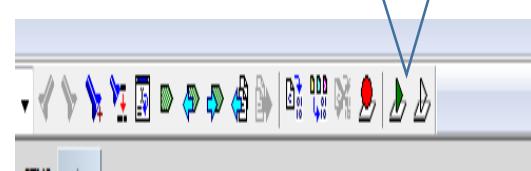
```

这是要仿真的程序

按照最先建立工程的要求在 options 选项里面选择 ST-LINK



在代码界面点击进入硬件调试



进来之后就可以看见调试按钮和右边的汇编界面



还有一个红手掌 是在全速运行的时候可以执行马上运行停止

STM8 时钟初始化

STM8 启动是用的内部 16M HIS 时钟，如果要切换到外部 24M 时钟，有两种方式

1. 手动切换

```
PI_ODR = 0x00; //PIO输出低电平  
PC_ODR = 0x02; //PC1输出高电平  
CLK_SWR= 0xB4; //时钟源为HSE 0xB4  
//时钟源为HSI 0xE1  
//时钟源为LSI 0x02  
  
while((CLK_SWCR&0x08)==0); //等待目标时钟准备就绪  
while(1)  
{  
    temp++;  
    if(temp>5)  
    {  
        CLK_SWCR |=0x02; //将时钟切换到HSE 上  
    }  
    PC_ODR = 0x02; //PC1输出高电平  
    delay_ms(1000); //因为没有初始化时钟，所以我们使用的是内部2M时钟  
    PC_ODR = 0x04; //PC2输出高电平  
    delay_ms(1000);  
    PC_ODR = 0x08; //PC3输出高电平  
    delay_ms(1000);  
}
```

CLK_SWR= 0xB4; //时钟源选择

8.9.4 主时钟切换寄存器 (CLK_SWR)

地址偏移值: 0x04

复位值: 0xE1

7	6	5	4	3	2	1	0
SWI[7:0]							
rw	rw	rw	rw	rw	rw	rw	rw
位 7:0 SWI[7:0]: 主时钟选择位 由软件写入。用以选择主时钟源。当时钟切换正在进行(SWBSY=1)时，该寄存器的内容将被写保护。如果寄存器CLK_CSSR的位AUX=1，则该寄存器将被置位复位值(HSI)。如果选择了快速Halt唤醒模式(寄存器CLK_ICKR的位FHW=1)，从停机(Halt)/ 活跃停机(Active Halt)唤醒时，该寄存器将被硬件设置为E1h(选择HSI) 0xE1: HSI为主时钟源(复位值) 0xD2: LSI为主时钟源(仅当LSI_EN选项位为1时) 0xB4: HSE为主时钟源							

while((CLK_SWCR&0x08)==0); //等待目标时钟准备就绪，这是手动切换时钟的重点要等待

8.9.5 切换控制寄存器 (CLK_SWCR)

地址偏移值: 0x05

复位值: 未定义

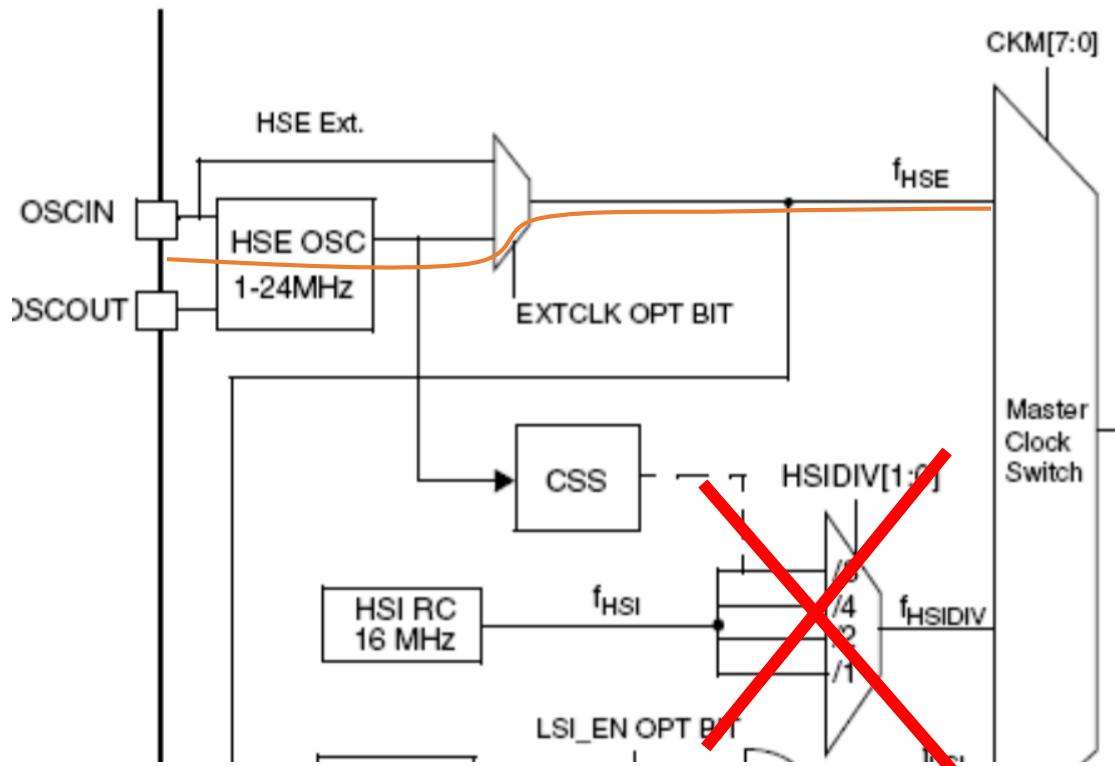
7	6	5	4	3	2	1	0
保留			SWIF	SWIEN	SWEN	SWBSY	
rc_w0			rw	rw	rw	rw	

位 3 SWIF: 时钟切换中断标志位
由硬件置位或软件写0清除。该位的含义取决于SWEN位的状态。参见图15和图16。
手动切换模式下(SWEN=0):
0: 目标时钟源未准备就绪
1: 目标时钟源准备就绪
自动切换模式下(SWEN=0):
0: 无时钟切换事件发生
1: 有时钟切换事件发生

CLK_SWCR |=0x02; //将时钟切换到 HSE 上

位 1	SWEN: 切换启动/停止 由软件置位或清除。向该位写1将切换主时钟至寄存器CLK_SWR指定的时钟源。 0: 禁止时钟切换的执行 1: 使能时钟切换的执行
-----	---

这样手动切换时钟就成功了，切换到了外部时钟 24M



2. 时钟自动切换

```

if (temp>5)
{
    CLK_SWCR |=0x02; //将时钟切换到HSE上
    CLK_SWR= 0xB4; //时钟源为HSE 0xB4
}

PC_ODR = 0x02; //PC1输出高电平
delay_ms(1000);
PC_ODR = 0x04; //PC2输出高电平
delay_ms(1000);
PC_ODR = 0x08; //PC3输出高电平
delay_ms(1000);

```

时钟自动切换就是少了
while((CLK_SWCR&0x08)==0);时
钟就绪函数，但是时钟自动切
换顺序是先切换时钟到 HSE
上，然后再设置时钟源，不要
搞反了，否则时钟切换失败

下面这样自动切换就是错误的

```

if (temp>5)
{
    CLK_SWR= 0xB4; //时钟源为HSE 0xB4
    CLK_SWCR |=0x02; //将时钟切换到HSE上

}

PC_ODR = 0x02; //PC1输出高电平
delay_ms(1000);
PC_ODR = 0x04; //PC2输出高电平
delay_ms(1000);
PC_ODR = 0x08; //PC3输出高电平
delay_ms(1000);

```

这两段代码顺序很重要

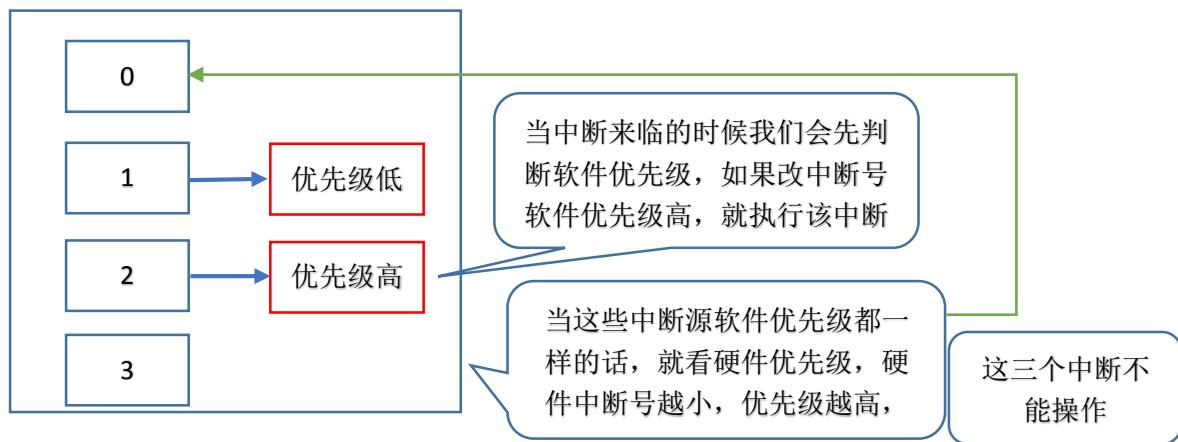
时钟手动切换和自动切换就是这样了。

STM8 中断处理

最高软件优先级中断是 **RESET , TLI , TRAP** 这三个中断不可屏蔽

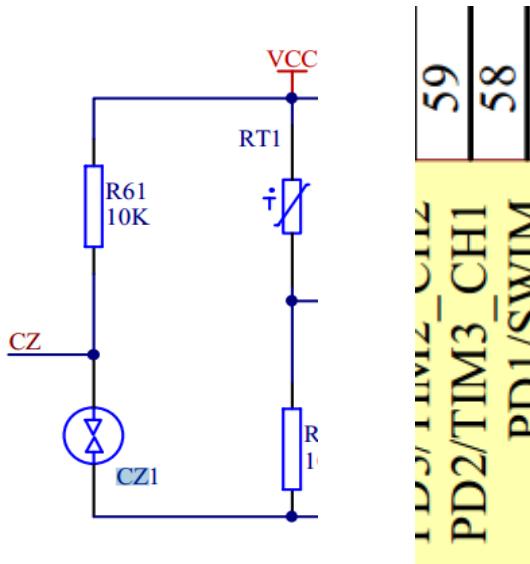
有唯一的中断向量号，就有对应的唯一中断地址

比如说：我这里中断向量号有 0 1 2 3



中断向量号	中断源	描述	从停机(Halt)模式唤醒功能	从活跃停机(Active Halt)模式唤醒功能	向量地址
	RESET	复位	是	是	8000h
	TRAP	软件中断			8004h
0	TLI	外部最高级中断			8008h
1	AWU	自动唤醒HALT模式中断		是	800Ch
2	CLK	时钟控制器			8010h
3	EXTI0	端口A外部中断	是	是	8014h
4	EXTI1	端口B外部中断	是	是	8018h
5	EXTI2	端口C外部中断	是	是	801Ch
6	EXTI3	端口D外部中断	是	是	8020h
7	EXTI4	端口E外部中断	是	是	8024h

Stm8 外部中断代码操作



我们操作 PD2 io 口来识别振动传感器中断

```
/////////中断初始化函数/////////
void EXTI_init(void)
{
    PD_DDR=0x00; //PD2为输入模式
    PD_CR1=0x04; //0000 0100 带上拉输入
    PD_CR2=0x04; //0000 0100 使能PD2口外部中断

    EXTI_CR1=0x80; //1000 0000 PD口下降沿触发

    asm("rim"); //开全局中断 IAR的汇编语言asm前面不需要斜杠,
}
}
```

11.9.3 端口 x 数据方向 (Px_DDR)

地址偏移值: 0x02

复位值: 0x00

7	6	5	4	3	2	1	0
DDR7	DDR6	DDR5	DDR4	DDR3	DDR2	DDR1	DDRO

位7:0 DDR[7:0]: 数据方向寄存器位
这些位可通过软件置1或置0，选择引脚输入或输出
0: 输入模式
1: 输出模式

11.9.4 端口 x 控制寄存器 1 (Px_CR1)

地址偏移值: 0x03

复位值: 0x00

7	6	5	4	3	2	1	0
C17	C16	C15	C14	C13	C12	C11	C10

位7:0 C1[7:0]控制寄存器位
这些位可通过软件置1或置0，用来在输入或输出模式下选择不同的功能。请参考表18
在 输入模式时(**DDR=0**):
0: 浮空输入
1: 带上拉电阻输入
在 输出模式时(**DDR=1**):
0: 模拟开漏输出(不是真正的开漏输出)
1: 推挽输出，由CR2相应的位做输出摆率控制

```

//////////中断初始化函数/////////
void EXTI_init(void)
{
    PD_DDR=0x00; //PD2为输入模式
    PD_CR1=0x04; //0000 0100 带上拉输入
    PD_CR2=0x04; //0000 0100 使能PD2口外部中断
    EXTI_CR1=0x80; //1000 0000 PD口下降沿触发
    asm("rim"); //开全局中断 IAR的汇编语言asm前面不需要斜杠,
}

```

设置 IO 口工作在中断模式

设置该组 IO 口的
中断触发方式

11.9.5 端口 x 控制寄存器 2 (Fx_CR2)

地址偏移值: 0x04

复位值: 0x00

7	6	5	4	3	2	1	0
C27	C26	C25	C24	C23	C22	C21	C20

位7:0	C2[7:0] 控制寄存器位 相应的位通过软件置1或置0，用来在输入或输出模式下选择不同的功能。在输入模式下，由 CR2相应的位使能中断。如果该引脚无中断功能，则对该引脚无影响。 在输出模式下，置位将提高IO速度。此功能适用O3和O4输出类型。(参见引脚描述表) 在 输入模式时(DDR=0): 0: 禁止外部中断 1: 使能外部中断 在 输出模式时(DDR=1): 0: 输出速度最大为2MHZ. 1: 输出速度最大为10MHZ
------	--

10.9.3 外部中断控制寄存器 1 (EXTI_CR1)

地址偏移值: 0x01

复位值: 0x00

因为是 PD2, 所以中断所在的位置是 PDIS

7	6	5	4	3	2	1	0
PDIS[1:0]		PCIS[1:0]	PBIS[1:0]	PAIS[1:0]			

位7:0	PDIS[1:0] : PORT D 的中断触发位 这些位仅在CC寄存器的I1和I0位都为1(级别3)时才可以写入。这些位定义端口D 的 中断触发位 00: 下降沿和低电平触发 01: 仅上升沿触发 10: 仅下降沿触发 11: 上升沿和下降沿触发
------	---

_asm("rim"); //STVD里面的汇编_asm("");

asm("rim"); //开全局中断 IAR的汇编语言asm前面不需要斜杠,

所以注意 **asm** 的变化，在这里中断初始化函数就写完了，这个中断没有设置优先级

```

/* 中断服务函数 */
#pragma vector=EXTI3_vector
_interrupt void EXTI3(void)
{
  NUM++;
}

```

这是中断服务函数

中断服务函数在 IAR 和 STVD 里面是有格式区别的

```
/*中断服务函数*/  
#pragma vector=EXTI3_vector  
_interrupt void EXTI3(void)  
{  
    NUM++;  
}
```

IAR 版本中断服务函数

```
/*外部中断服务函数***/  
@far @interrupt void PORTD_EXTI_Interrupt(void)  
{  
    NUM++;  
    if(NUM>=10000)  
    {  
        NUM=0;  
    }  
}/*****结束*****/
```

STVD 版本中断服务函数

#pragma vector=EXTI3_vector **vector= 填写你要使用的中断号**

EXTI3_vector 在这个头文件里面

: iostm8s208rb.h

为什么要选择 EXTI3?

10.8 中断映射

表16 中断映射表

中断向量号	中断源	描述	从停机(Halt)模式唤醒功能	从活跃停机(Active Halt)模式唤醒功能	向量地址
	RESET	复位	是	是	8000h
	TRAP	软件中断			8004h
0	TLI	外部最高级中断			8008h
1	AWU	自动唤醒HALT模式中断		是	800Ch
2	CLK	时钟控制器			8010h
3	EXTI0	端口A外部中断	是	是	8014h
4	EXTI1	端口B外部中断	是	是	8018h
5	EXTI2	端口C外部中断	是	是	801Ch
6	EXTI3	端口D外部中断	是	是	8020h

这是
PD 端
口的
中断
位置

将 EXTI3_vector 里面的 vector 去掉就是中断入口函数

_interrupt void EXTI3(void)

这个是两个下划线记住

填写中断服务函数内容.....

}

这样整个中断程序就写完了，只是中断优先级没有写这种情况下就看中断号来确定优先级

```
int main( void )  
{  
    CLK_SWCR |=0x02;//将时钟切换到HSE上  
    CLK_SWR= 0xB4; //时钟源为HSE 0xB4  
    EXTI_init(); //中断初始化  
    Seg_Init();  
  
    while(1)  
    {  
        Display(NUM);  
    }  
}
```

```
/*中断服务函数*/  
#pragma vector=EXTI3_vector  
_interrupt void EXTI3(void)  
{  
    NUM++;  
}
```

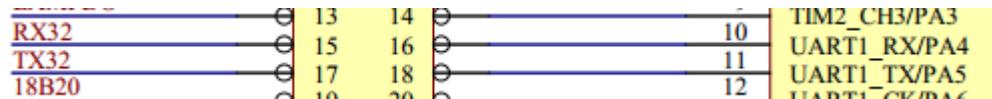
UART 串口编程

STM8S208R8T6 有 UART1 和 UART3 串口

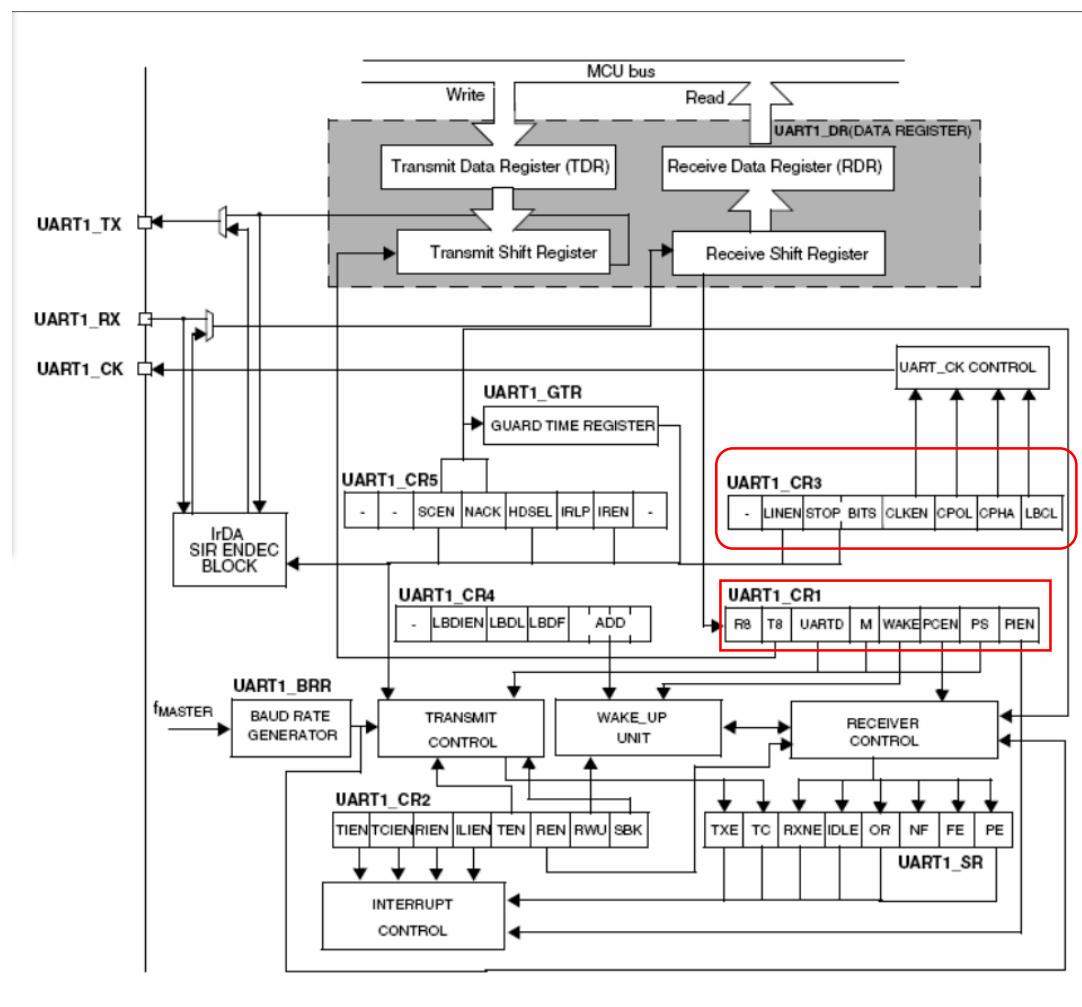
我们来设置 UART1 串口

UART1_RX/ (HS) PA4 10
UART1_TX/ (HS) PA5 11
UART1_CK/ (HS) PA6 12

这是数据手册上的



这是原理图上的，我们就是要串口普通功能，所以不需要 UART1_CK 时钟



22.7.5 控制寄存器 1(UART_CR1)

地址偏移值: 0x04

复位值: 0x00

7	6	5	4	3	2	1	0
R8	T8	UARTD	M 0	WAKE	PCEN	PS	PIEN
RW	RW	RW	RW	RW	RW	RW	RW

位4	M: 字长 该位定义了数据字的长度, 由软件对其进行置位和清零操作 0: 一个起始位, 8个数据位, n个停止位(n取决于UART_CR3中的STOP[1:0]位) 1: 一个起始位, 9个数据位, 一个停止位。 注意: 在数据传输过程中(发送或者接收时), 不能修改这个位。 在LIN从模式, M位和UART_CR3寄存器的STOP[1:0]应当保持为0
----	--

M 写, 其余为默认为 0

```
void UART1_init()
{
  UART1_CR1=0x00; //1个停止位, 8个数据位
  UART1_CR3=0x00;
  UART1_BRR2=0x01; //波特率9600bps DIV=833d=0341h
  UART1_BRR1=0x34;
  UART1_CR2=0x0c; //TEN=1 允许发送 REN=1允许接收
}
```

初始化 UART1

22.7.7 控制寄存器 3(UART_CR3)

地址偏移值: 0x06

复位值: 0x00

7	6	5	4	3	2	1	0
保留	LINEN	STOP[1:0]	0	CLKEN	CPOL	CPHA	LBCL
RW	RW	RW	RW	RW	RW	RW	RW

位5:4	STOP: 停止位 用来设置停止位的位数。 00: 1个停止位; 01: 保留 10: 2个停止位; 11: 1.5个停止位; 注意: 对于LIN从模式, 这两位应该都是0。
------	--

STOP 选择 1 个停止位, 其余默认写 0

22.3.4 高精度波特率发生器

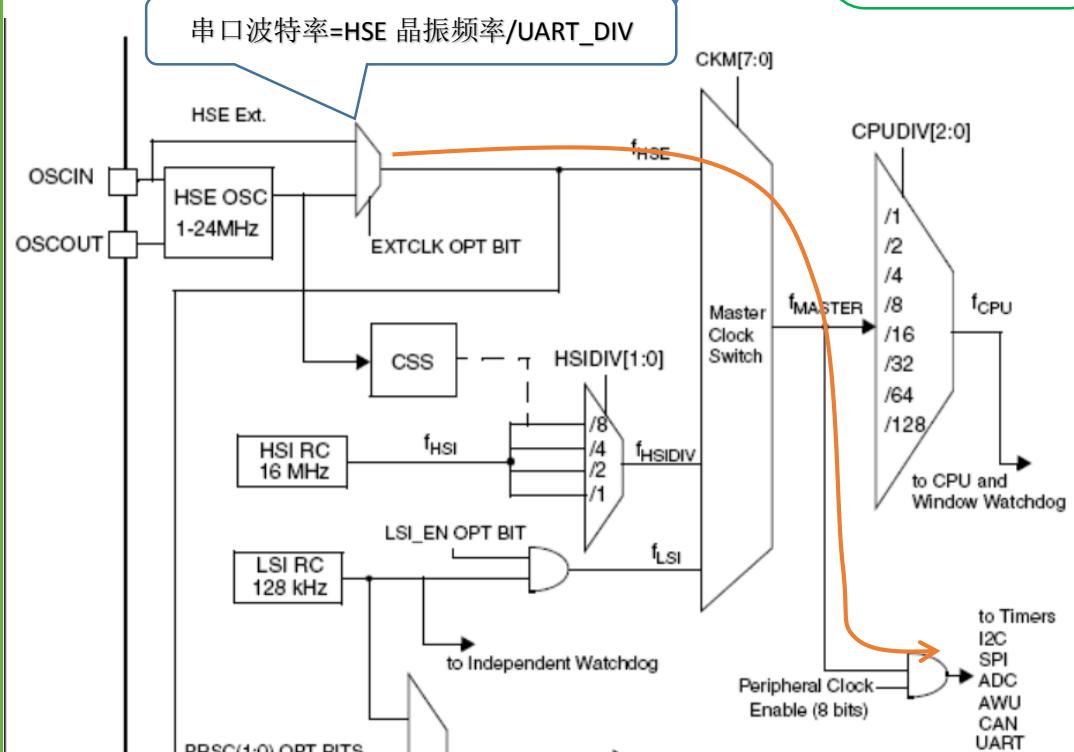
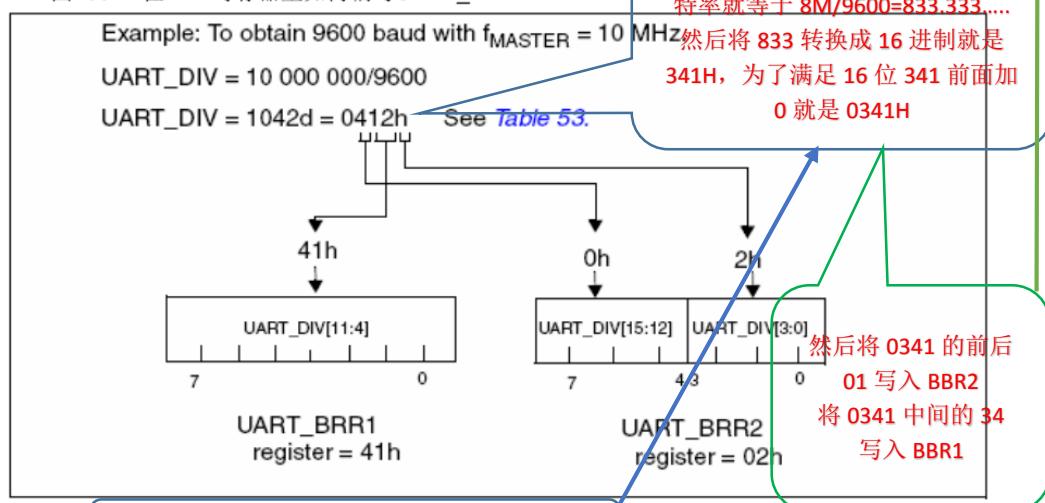
接收器和发送器的波特率可按照下面的公式通过配置16位除法器UART_DIV来设置:

$$\text{Tx/Rx baud rate} = \frac{f_{\text{MASTER}}}{\text{UART_DIV}}$$

UART_DIV是一个无符号的整数。存储在寄存器BRR1和BRR2中。如图105。

请参考表49来了解典型的波特率设置。

图105 在BRR寄存器里如何编写UART_DIV



因为我们是用 while 轮询接受串口数据，和 while 轮询发送串口数据所以这里不需要中断

22.7.6 控制寄存器 2(UART_CR2)

地址偏移值: 0x05

复位值: 0x00

打开串口接收和发送功能，不设置串口
其余的中断功能

TIEN	TCIEN	RIEN	ILIEN	TEN	REN	RWU	SBK
rw	rw	rw	rw	rw	rw	rw	rw

串口接收程序

```
unsigned char Receive()
{
    unsigned char temp;
    temp=0;

    if((UART1_SR&0x20)==0x20) //RXNE位置1表示接受完成
    {
        temp=UART1_DR;
    }

    return temp;
}
```

22.7 UART寄存器描述

22.7.1 状态寄存器(UART_SR)

地址偏移值: 0x00

复位值: 0xC0

7	6	5	4	3	2	1	0
TXE	TC	RXNE	IDLE	OR/LHE	NF	FE	PE
r	rc_w0	rc_w0	r	r	r	r	r

位5 **RXNE:** 读数据寄存器非空
当RDR移位寄存器中的数据被转移到UART_DR寄存器中，该位被硬件置位。如果UART_CR1寄存器中的RXNEIE为1，则产生中断。对UART_DR的读操作可以将该位清零。RXNE位也可以通过写入0来清除，对于UART2和UART3，该位也可以通过写入0来清除。
0: 数据没有收到；
1: 收到数据，可以读出。

22.7.2 数据寄存器(UART_DR)

地址偏移值: 0x01

复位值: 未定义

7	6	5	4	3	2	1	0
DR[7:0]							
rW	rW	rW	rW	rW	rW	rW	rW

temp=UART1_DR ; temp 读取完 UART_DR 寄存器后，RXNE 位会自动清 0 不需要我去操作
串口发送程序

```
void Transmit(unsigned char data)
{
    UART1_DR=data;
    while((UART1_SR&0x40)==0); //等待发送完成
}
```

位6	TC: 发送完成 当包含有数据的一帧发送完成后，由硬件将该位置位。如果UART_CR2中的TCIEN为1，则产生中断。可用用户程序清除该位(先读UART_SR，然后写入UART_DR)。对于UART2和UART3，该位也可以通过写入0来清除。 0: 发送还未完成； 1: 发送完成。
----	---

我们现在使用串口来给 PC 发数据

```
#include<iostm8s208rb.h>

void UART1_init()
{
    UART1_CR1=0x00; //1个停止位，8个数据位
    UART1_CR3=0x00;
    UART1_BRR2=0x01; //波特率9600bps DIV=833d=0341h
    UART1_BRR1=0x34;
    UART1_CR2=0x0c; //TEN=1 允许发送 REN=1允许接收
}

unsigned char Receive()
{
    unsigned char temp;
    temp=0;

    if((UART1_SR&0x20)==0x20) //RXNE位置1表示接受完成
    {
        temp=UART1_DR;
    }

    return temp;
}

void Transmit(unsigned char data)
{
    UART1_DR=data;
    while((UART1_SR&0x40)==0); //等待发送完成
}

int main( void )
{
    CLK_SWCR = 0x02; //使能时钟切换
    CLK_SWR = 0xb4; //时钟源为HSE
    UART1_init();

    while(1)
    {
        Transmit(0xfe);
    }
}
```

这是无线循环发送



你看我 PC 机正常接收数据了

下面我们用 PC 机给串口发数据

```
unsigned char Receive()
{
    unsigned char temp;
    temp=0;
    while((UART1_SR&0x20)!=0x20); //等到电脑数据发到单片机
    if((UART1_SR&0x20)==0x20) //RXNE位置1表示接受完成
    {
        temp=UART1_DR;
    }

    return temp;
}
```

修改了接受程序，如果用上面那个接受程序，串口终端就被刷屏了

```
int main( void )
{
    unsigned char data;
    CLK_SWCR = 0x02; //使能时钟切换
    CLK_SWR = 0xb4; //时钟源为HSE
    UART1_init();

    while(1)
    {
        data=Receive();
        Transmit(data);
    }
}
```



单片机串口接受 PC 数据成功

串口用中断方式接受数据

22.7.6 控制寄存器 2(UART_CR2)

地址偏移值: 0x05

复位值: 0x00

	7	6	5	4	3	2	1	0
	TIEN	TCIEN	RIEN	ILIEN	TEN	REN	RWU	SBK
	RW	RW	RW	RW	RW	RW	RW	RW
位5	RIEN: 接收中断使能 软件对该位置位或者清零 0: 中断被禁止; 1: 当USART_SR中的OR或者RXNE为1时, 产生USART中断。							

```
void UART1_init()
{
    UART1_CR1=0x00; //1个停止位, 8个数据位
    UART1_CR3=0x00;
    UART1_BRR2=0x01; //波特率9600bps DIV=833d=0341h
    UART1_BRR1=0x34;

    asm("rim"); //IAR编译器汇编语言开总中断, rim前面不需要杠

    UART1_CR2=0x2c; //TEN=1 允许发送 REN=1允许接收 增加接受中断使能
}
```

初始化函数增加汇编

增加一位接受中断

```

#include<iostm8s208rb.h>

void UART1_init()
{
    UART1_CR1=0x00; //1个停止位，8个数据位
    UART1_CR3=0x00;
    UART1_BRR2=0x01; //波特率9600bps DIV=833d=0341h
    UART1_BRR1=0x34;

    asm("rim"); //IAR编译器汇编语言开总中断，rim前面不需要杠

    UART1_CR2=0x2c; //TEN=1 允许发送 REN=1允许接收 增加接受中断使能
}

void Transmit(unsigned char data)
{
    UART1_DR=data;
    while((UART1_SR&0x40)==0); //等待发送完成
}

#pragma vector=UART1_R_RXNE_vector //串口中断服务程序
__interrupt void UART1() //轮询接受改为中断接受
{
    unsigned char temp;
    temp=UART1_DR;
    Transmit(temp);
}

int main( void )
{
    CLK_SWCR = 0x02; //使能时钟切换
    CLK_SWR = 0xb4; //时钟源为HSE
    UART1_init();

    while(1)
    {
    }
}

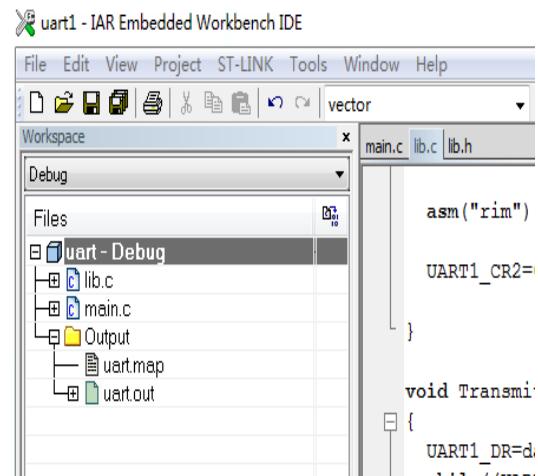
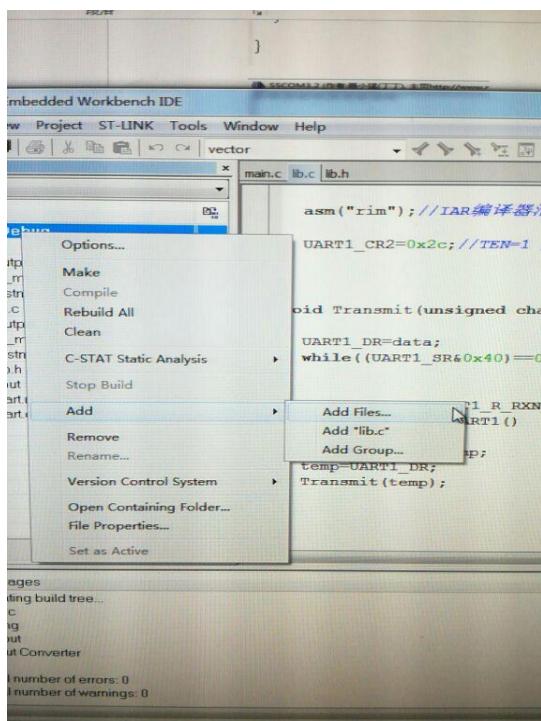
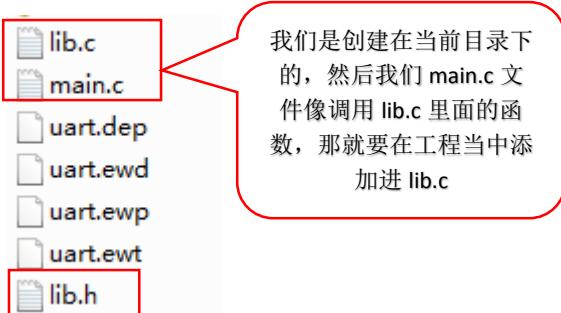
```



单片机成功接受 PC 数据，然后发回给 PC

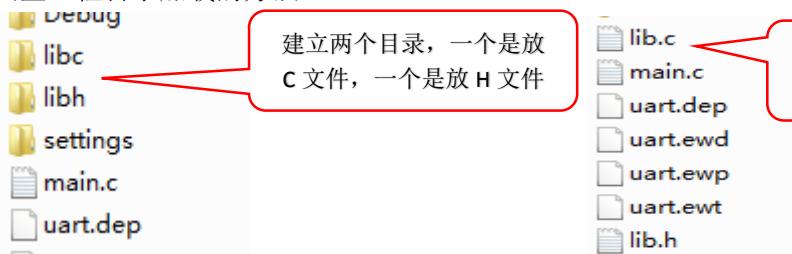
IAR for STM8 软件多个 C 文件编译

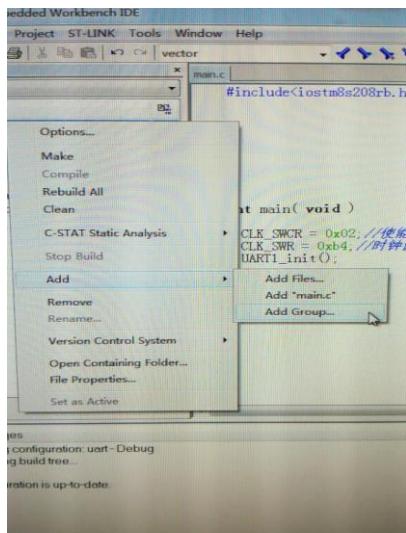
先用编译器创建其它 C 文件



加载之后就是这样了，当然这是在几个小文件之间使用，如果项目工程大了，我们就要用目录的方法来管理。

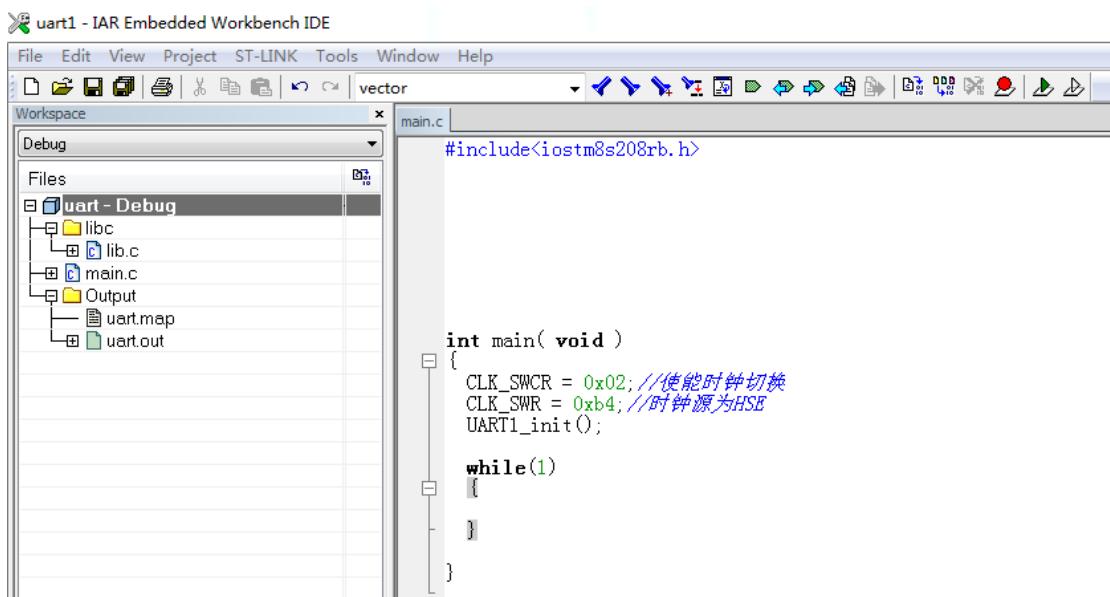
大型工程目录加载的方法



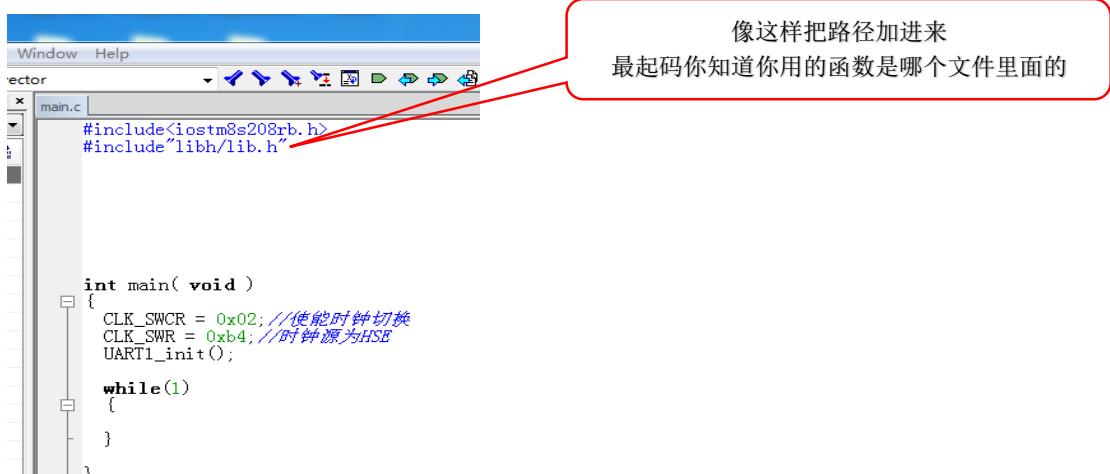


选择 ADD Group 加载目录

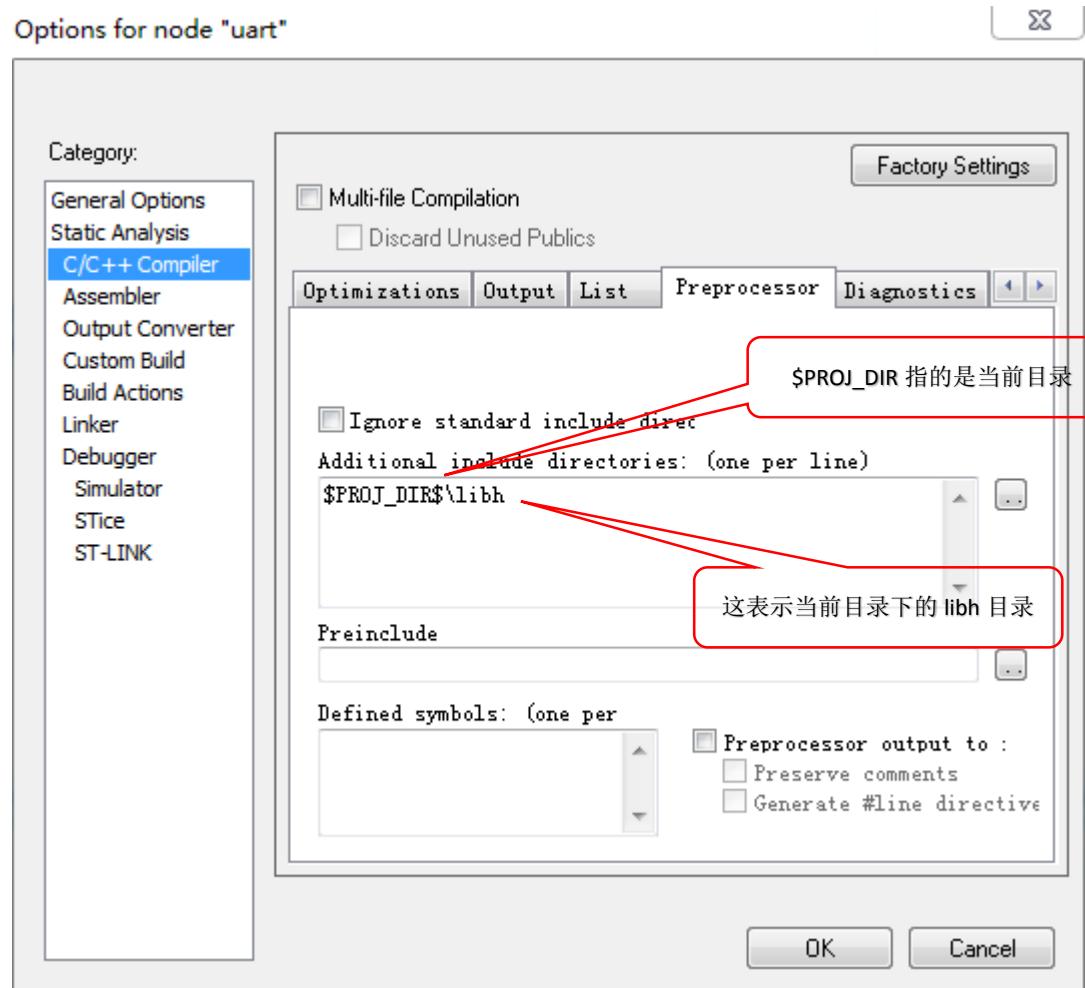
这样就成功了



你会发现主函数连调用子函数的头文件都不用加就可以调用，这是因为编译器对软件进行了链接优化，但是我觉得还是注意使用!!!



如果你觉得在 C 文件里面给头文件加路径斜杠一大堆很不爽，那么我们在编译器的预处理设置里面加路径。



The screenshot shows the 'main.c' code editor window. The code is as follows:

```
#include<iostm8s208rb.h>
#include"lib.h"

int main( void )
{
    CLK_SWCR = 0x02; //使能时钟切换
    CLK_SWR = 0xb4; //时钟源为HSE
    UART1_init();
}

while(1)
{
}
```

Annotations with red boxes and arrows point to specific parts:

- A box points to the '#include"lib.h"' line with the text '就直接写你要调用的 h 文件'.

你主函数直接写 include 的 h 文件，不用像前面那样加路径了，因为预处理设置帮你加好了。

单片机 printf 实现

The screenshot shows the IAR Embedded Workbench IDE interface. The workspace contains a project named "uart - Debug" with files "libc", "lib.c", "main.c", "Output", "uart.map", and "uart.out". The main editor window displays the "main.c" file. A red box highlights the line "#include<stdio.h>" with the annotation "Printf 需要头文件 stdio.h". Another red box highlights the line "FILE *f)" with the annotation "FILE 文件报错". The messages window shows two errors: "Error[Pe020]: identifier \"FILE\" is undefined" and "Error while running C/C++ Compiler".

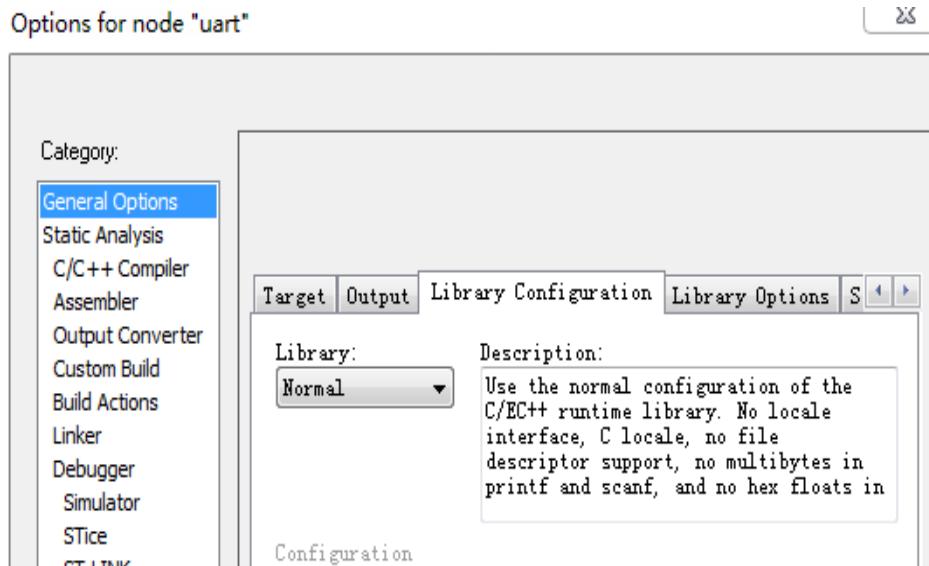
```
#include<iostm8s208rb.h>
#include<stdio.h>
#include"lib.h"

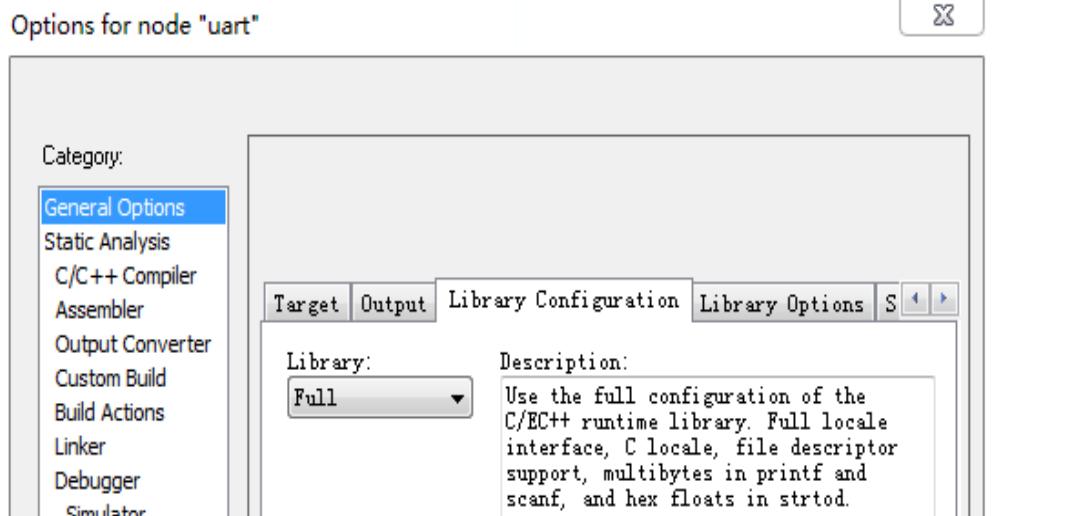
int fputc(int ch, FILE *f)
{
    UART1_DR=((unsigned char)ch);
    while((UART1_SR&0x40)==0); //等待发送完成
    return ch;
}

int main( void )
{
    CLK_SWCR = 0x02; //使能时钟切换
    CLK_SWR = 0xb4; //时钟源为HSB
    UART1_init();

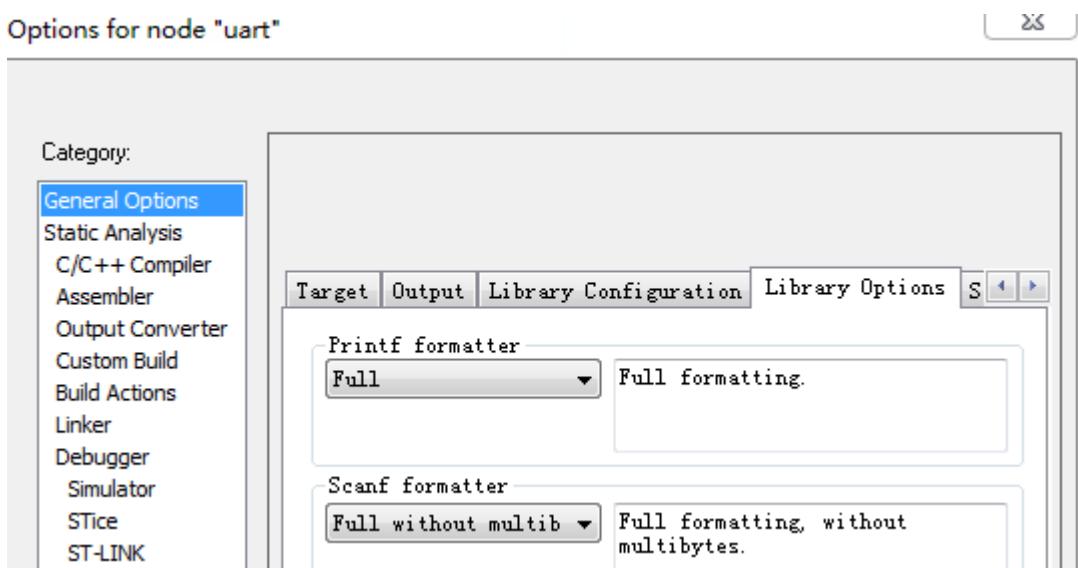
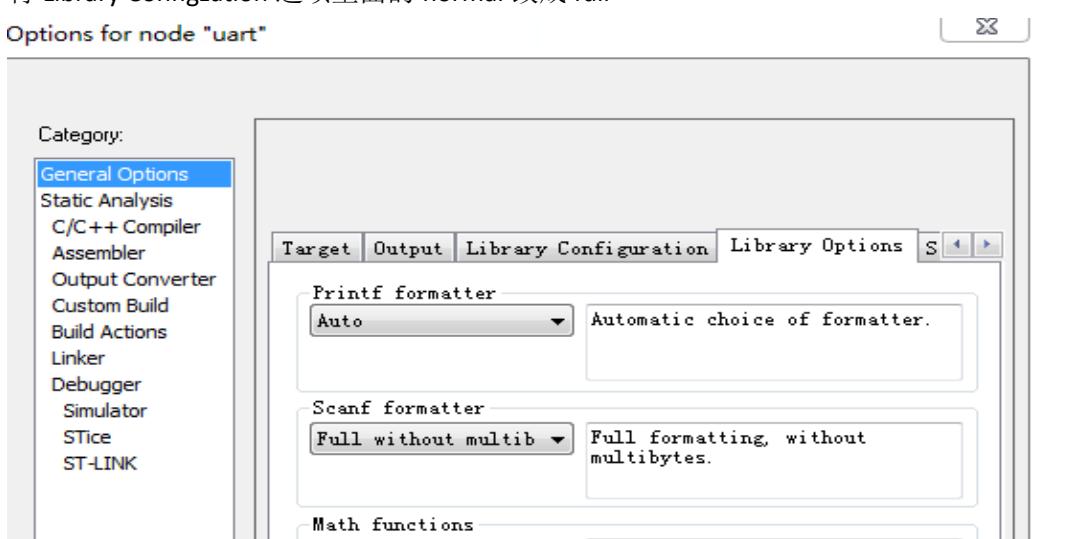
    while(1)
    {
        printf("STM8单片机输出\r\n");
    }
}
```

用串口实现 printf 出现报错，`fputc(int ch, FILE *f)`这个是标准输出流函数，看来编译器默认是支持正常输出方式，不是单片机方式





将 Library Configuration 选项里面的 normal 改成 full



将 Library Options 里面的 printf formatter auto 改成 full

然后确定，这样你就能正常编译通过了

uart1 - IAR Embedded Workbench IDE

File Edit View Project ST-LINK Tools Window Help

Workspace

Debug

Files

uart - Debug

- libc
- lib.c
- main.c
- Output
- uart.map
- uart.out

uart

Messages

- lib.c
- main.c
- Linking
- uart.out
- Output Converter

Total number of errors: 0
Total number of warnings: 0

main.c | stdio.h | lib.c

```
#include<iostm8s208rb.h>
#include<stdio.h>
#include"lib.h"

int fputc(int ch,FILE *f)
{
    UART1_DR=((unsigned char)ch);
    while((UART1_SR&0x40)==0); //等待发送完成
    return ch;
}

int main( void )
{
    CLK_SWCR = 0x02; //使能时钟切换
    CLK_SWR = 0xb4; //时钟源为HSE
    UART1_init();

    while(1)
    {
        printf("STM8单片机输出\r\n");
    }
}
```



STM8 延时函数

外部晶振 8M

```
/*************延时MS函数*****/
void Delay_Ms(unsigned int ms)
{
    unsigned int x, i;
    for(x=ms; x>0; x--)
    {
        for(i=1550; i>0; i--) ;
    }
}
```

误差千分之 3

```
/*************延时3us函数*****/
void Delay_us(unsigned int us)
{
    while(us--);
}
```

8M 外部晶振这里只能做到 2.8us 多点，差不多 3us。100us 实测 88us

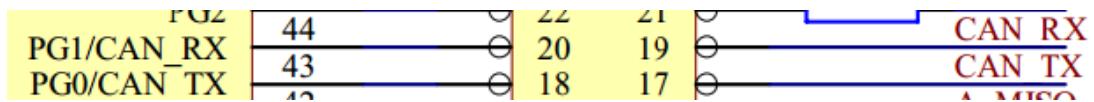
STM8 CAN 总线使用

CAN 总线硬件设计和数据帧原理请看我的 STM32 使用手册

这里讲 STM8 can 总线软件使用方法

STM8 CAN 总线要操作两种寄存器，一种是直接写寄存器地址比如操作 CFR 寄存器，
CFR=0x0F；还有一种是不能直接写寄存器地址，要先转换页，比如 CFR 寄存器在 FFF 寄存器页里面，那么就要这样写 FFF=CFR，然后 FFF=0x0F，其实就是在写 CFR=0x0F 的意思

1. 初始化 CAN 总线 GPIO



```
void beCAN_gpio_init() //初始化CAN总线IO口
{
    PG_DDR = 0x01; //PG0是CAN_TX设置为输出
    PG_CR1 = 0x01; //PG0输出为推挽
    PG_CR2 = 0x00; //PG0输出速度2M
    /*PG1是CAN_RX 默认是输入模式*/
}
```

2. 初始化 CAN 总线寄存器

```
void beCAN_bus_init() //初始化CAN总线
{
    CAN_MCR = 0x01; //CAN进入初始化模式
    while((CAN_MSR&0X01)==0X00); //等待CAN进入初始化模式
    Delay_us(100);
    CAN_MCR |= 0x50; //初始化模式设置
    CAN_DGR = 0x01; //设置CAN为环回模式

    /*设置CAN总线的波特率*/
    /*设置页面里面的寄存器就不能直接向寄存器里面写值，要先换页再写值*/
    CAN_FPSR = 0x06; //BTR1和BTR2寄存器是在PAGE 6, 所以我用FPSR来切换页面到0x06
    CAN_P4 = 0x05; //同步跳跃 CAN_BTR1 BRP=5
    CAN_P5 = 0x75; //CAN_BTR2 BS1=6 BS2=8

    /*设置过滤器*/
    CAN_P8 = 0x00; //CAN_FMR1 过滤器组0工作在屏蔽位
    CAN_P9 = 0x00;
    CAN_PA = 0x06; //过滤器组0 32位 过滤器组1 8位
    CAN_PB = 0x00; //过滤器组2 8位 过滤器组3 8位
    CAN_PC = 0x00; //过滤器组4 8位 过滤器组5 8位
    Delay_us(100);

    CAN_MCR &= 0xFE; //把MCR最低位INRQ=0 CAN进入正常模式
}
```

CAN_MCR = 0x01

23.11.1 CAN主控制寄存器 (CAN_MCR)

地址偏移值: 0x00

复位值: 0x02

7	6	5	4	3	2	1	0
TTCM 0	ABOM 0	AWUM 0	NART 0	RFLM 0	TXFP 0	SLEEP 0	INRQ 1

位0	INRQ: 初始化请求 软件对该位清0可使CAN从初始化模式进入正常工作模式：当CAN在接收引脚检测到连续的11个隐性位后，CAN就达到同步，并准备好接收和发送数据。同时硬件相应地将CAN_MSR寄存器的INAK位清0。 软件对该位置1可使CAN进入初始化模式：一旦软件将该位置位，CAN硬件等待当前任务(发送或接收)结束，再进入初始化模式。相应地，硬件将CAN_MSR寄存器的INAK位置1。
----	--

while((CAN_MSR&0X01)==0X00); //等待 CAN 进入初始化模式

23.11.2 CAN主状态寄存器 (CAN_MSR)

地址偏移值: 0x01

复位值: 0x02

我们用这个寄存器来确认我们前面设置的初始化模式是否成功

7	6	5	4	3	2	1	0
保留	RX	TX	WKUI	ERRI	SLEEP	INAK	
r	r	rc_w1	rc_w1	r	r	r	

位0	INAK: 初始化确认标志 该位由硬件置1，表示CAN当前处于初始化模式，供软件进行状态查询。该位是对软件请求进入初始化模式的确认(对CAN_MCR寄存器的INRQ位置1)。 当CAN退出初始化模式时硬件对该位清0(需要跟CAN总线同步)。这里跟CAN总线同步是指，硬件需要在CAN的RX引脚上检测到连续的11位隐性位。
----	---

CAN_MCR |= 0x50; //初始化模式设置

23.11 CAN 寄存器描述

23.11.1 CAN主控制寄存器 (CAN_MCR)

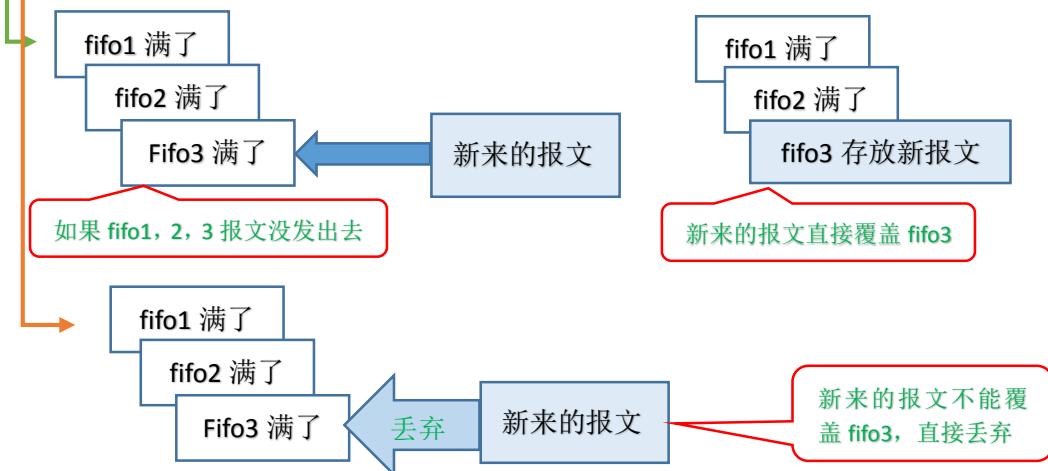
地址偏移值: 0x00

复位值: 0x02

7	6	5	4	3	2	1	0
TTCM 0	ABOM 1	AWUM 0	NART 1	RFLM 0	TXFP 0	SLEEP 0	INRQ 0
r w	r w	r w	r w	r w	r w	r w	r w

位7	TTCM: 时间触发通信模式 0: 禁止时间触发通信模式； 1: 允许时间触发通信模式。 注: 要想了解关于时间触发通信模式的更多信息，请参考 23.4.4时间触发通讯模式 。
位6	ABOM: 自动离线(Bus-Off)管理 该位决定CAN硬件在什么条件下可以退出离线状态。 0: 通过软件请求使CAN退出离线状态。请参考 23.6.5出错管理离线恢复 。 1: 一旦硬件检测到128次11位连续的隐性位，自动退出离线状态。 注: 要想了解关于离线状态的更多信息，请参考 23.6.5出错管理 。
位5	AWUM: 自动唤醒模式 该位决定CAN处在睡眠模式时由硬件还是软件唤醒CAN: 0: 睡眠模式通过清除CAN_MCR寄存器的SLEEP位，由软件唤醒； 1: 睡眠模式通过检测CAN报文，由硬件自动唤醒。唤醒的同时，硬件自动对CAN_MSR寄存器的SLEEP和SLAK位清0。

位4	NART: 禁止报文自动重传 0: 按照CAN标准, CAN硬件在发送报文失败时会一直自动重发, 直到发送成功; 1: 不管发送的结果如何(成功、出错或仲裁丢失), CAN报文只被发送1次。	我们选择每次都发一次
位3	RFLM: 接收FIFO锁定模式 0: 在接收溢出时FIFO未被锁定, 当接收FIFO处于满状态, 下一个收到的报文会覆盖先前的报文; 1: 在接收溢出时FIFO被锁定, 当接收FIFO处于满状态, 下一个收到的报文会被丢弃。	
位2	TXFP: 发送FIFO优先级 当有多个报文同时在等待发送时, 该位决定这些报文的发送顺序 0: 优先级由报文的标识符来决定; 1: 优先级由发送请求的顺序来决定(按时间发生顺序)。	不管谁先把数据放入邮箱, 我都按照标识符顺序来发送, 哪怕是最最后放入邮箱的只要标识符高就先发送 谁先将数据放入邮箱, 谁先发送
位1	SLEEP: 睡眠模式请求 软件对该位置1可以请求CAN进入睡眠模式, 一旦当前CAN的任务(发送或接收报文)结束, CAN就立即进入睡眠。 如果软件对该位清0, 将使CAN退出睡眠模式。	CAN总线复位是睡眠模式, 我们要设置CAN寄存器就要让can总线醒着, 设置0 当AWUM位被置位且在CAN Rx信号中检测出SOF位时, 硬件对该位清0。
位0	INRQ: 初始化请求 软件对该位清0可使CAN从初始化模式进入正常工作模式: 当CAN在接收引脚检测到连续的11个隐性位后, CAN就达到同步, 并准备好接收和发送数据。同时硬件相应地将CAN_MSR寄存器的INAK位清0。 软件对该位置1可使CAN进入初始化模式: 一旦软件将该位置位, CAN硬件等待当前任务(发送或接收)结束, 再进入初始化模式。相应地, 硬件将CAN_MSR寄存器的INAK位置1。	



`CAN_DGR = 0x01; //设置 CAN 为环回模式`

23.11.7 CAN诊断寄存器 (CAN_DGR)

地址偏移值: 0x06

复位值: 0x0C

7	6	5	4	3	2	1	0
保留		TXM2E	RX	SAMP	SILM	LBKM	

位7:5	保留位, 读出值为0。
位4	TXM2E: 发送邮箱2使能位 0: 强制beCAN与ST7的beCAN(2个发送邮箱)兼容 – 复位状态。 1: 使能第3个发送邮箱(邮箱号为2)
位3	RX: CAN接收电平 该位反映CAN接收引脚(CAN_RX)的实际电平。
位2	SAMP: 上次采样值 CAN接收引脚的上次采样值。
位1	SILM: 静默模式 0: 正常状态 1: 静默模式。
位0	LBKM: 环回模式 0: 禁止环回模式; 1: 允许环回模式。

设置 CAN 总线波特率

CAN_P4 = 0x05;

CAN_P5 = 0x75;

CAN_P4, CAN_P5 是设置波特率的寄存器但是我们不能直接向 CAN_P4 和 CAN_P5 里面写值
CAN_FPSR = 0x06;

我们要用 CAN_FPSR 来换页，因为 CAN_P4 和 CAN_P5 寄存器在页里面。

所以设置顺序得颠倒过来。先设置 FPSR 切换页

CAN_FPSR = 0x06;//BTR1 和 BTR2 寄存器是在 PAGE 6,所以我用 FPSR 来切换页面到 0x06

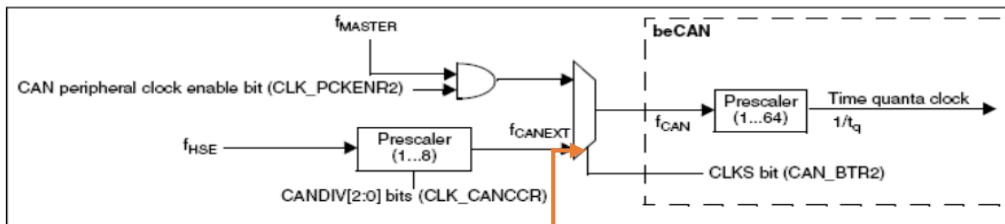
	PAGE 0	PAGE 1	PAGE 2	PAGE 3	PAGE 4
0x00	CAN_MCSR	CAN_MCSR	CAN_F0R1	CAN_F2R1	CAN_F4R1
0x01	CAN_MDLCR	CAN_MDLCR	CAN_F0R2	CAN_F2R2	CAN_F4R2
0x02	CAN_MIDR1	CAN_MIDR1	CAN_F0R3	CAN_F2R3	CAN_F4R3
0x03	CAN_MIDR2	CAN_MIDR2	CAN_F0R4	CAN_F2R4	CAN_F4R4
0x04	CAN_MIDR3	CAN_MIDR3	CAN_F0R5	CAN_F2R5	CAN_F4R5
0x05	CAN_MIDR4	CAN_MIDR4	CAN_F0R6	CAN_F2R6	CAN_F4R6
0x06	CAN_MDAR1	CAN_MDAR1	CAN_F0R7	CAN_F2R7	CAN_F4R7
0x07	CAN_MDAR2	CAN_MDAR2	CAN_F0R8	CAN_F2R8	CAN_F4R8
0x08	CAN_MDAR3	CAN_MDAR6	CAN_F1R1	CAN_F3R1	CAN_F5R1
0x09	CAN_MDAR4	CAN_MDAR4	CAN_F1R2	CAN_F3R2	CAN_F5R2
0x0A	CAN_MDAR5	CAN_MDAR5	CAN_F1R3	CAN_F3R3	CAN_F5R3
0x0B	CAN_MDAR6	CAN_MDAR6	CAN_F1R4	CAN_F3R4	CAN_F5R4
0x0C	CAN_MDAR7	CAN_MDAR7	CAN_F1R5	CAN_F3R5	CAN_F5R5
0x0D	CAN_MDAR8	CAN_MDAR8	CAN_F1R6	CAN_F3R6	CAN_F5R6
0x0E	CAN_MTSRL	CAN_MTSRL	CAN_F1R7	CAN_F3R7	CAN_F5R7
0x0F	CAN_MTSRH	CAN_MTSRH	CAN_F1R8	CAN_F3R8	CAN_F5R8
	Tx Mailbox 0	Tx Mailbox 1	Acceptance Filter 0:1	Acceptance Filter 2:3	Acceptance Filter 4:5
	PAGE 5	PAGE 6	PAGE 7		
	PAGE 5	PAGE 6	PAGE 7		
0x00	CAN_MCSR	CAN_ESR	CAN_MFMIR		
0x01	CAN_MDLCR	CAN_EIER	CAN_MDLCR		
0x02	CAN_MIDR1	CAN_TECR	CAN_MIDR1		
0x03	CAN_MIDR2	CAN_RECRR	CAN_MIDR2		
0x04	CAN_MIDR3	CAN_BTR1	CAN_MIDR3		
0x05	CAN_MIDR4	CAN_BTR2	CAN_MIDR4		
0x06	CAN_MDAR1	Reserved	CAN_MDAR1		
0x07	CAN_MDAR2	Reserved	CAN_MDAR2		
0x08	CAN_MDAR3	CAN_FMR1	CAN_MDAR3		
0x09	CAN_MDAR4	CAN_FMR2	CAN_MDAR4		
0x0A	CAN_MDAR5	CAN_FCR1	CAN_MDAR5		
0x0B	CAN_MDAR6	CAN_FCR2	CAN_MDAR6		
0x0C	CAN_MDAR7	CAN_FCR3	CAN_MDAR7		
0x0D	CAN_MDAR8	Reserved	CAN_MDAR8		
0x0E	CAN_MTSRL	Reserved	CAN_MTSRL		
0x0F	CAN_MTSRH	Reserved	CAN_MTSRH		

每一页地址都是一样只是功能不一样，所以我们设置 BTR1 寄存器就写成 CAN_P4 = 0x04
设置 BTR2 寄存器 CAN_P5 = 0x05

下面我们来计算下波特率

时钟我们采用外部的 8M 晶体振荡器

图143 时钟接口



外部时钟 f_{CANEXT} 的频率必须小于CPU的时钟频率(f_{CPU}).

有两种方法可以配置beCAN的时钟：

- 选择 f_{MASTER} 作为 CAN 的时钟。在这种情况下，在低功耗模式期间时钟可以在外设级别（外设时钟门控寄存器 CLK_PCKENR2）被停止。
显然，对于高速 CAN 应用， f_{MASTER} 必须由外部晶振提供。
- 或选择 f_{CANEXT} 的(设置 CLKS 位)作为 CAN 的时钟。在这种情况下，通过外设时钟门控寄存器不能将时钟停止。

23.11.14 CAN位时间特性寄存器 (CAN_BTR2)

地址偏移值：见表63

复位值：0x23

7	6	5	4	3	2	1	0
CLKS		BS[2:0]			BS1[3:0]		
r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w

本寄存器只能在CAN硬件为初始化模式下由软件访问。

位7	CLKS: 时钟输入选择位 0: 选择CPU时钟($f_{CAN} = f_{MASTER}$); 1: 选择外部时钟($f_{CAN} = f_{CANEXT}$).	Fcan 我们选择外部晶振
位6:4	BS2[2:0]: 时间段2 该位域定义了时间段2占用了多少个时间单元 时间段2 = BS2[2:0] + 1	为了满足 t_q 的公式这里设置为 8 BS2=7
位3:0	BS1[3:0]: 时间段1 该位域定义了时间段1占用了多少个时间单元 时间段1 = BS1[3:0] + 1	为了满足 t_q 的公式这里设置为 6 BS1=5 关于位时间特性的详细信息，请参考23.6.6节位。

波特率 = 100Kbps

$$\text{位时间} = \frac{1}{\text{波特率}} = \frac{1}{100000} = 0.00001$$

$$\text{时间单元}(t_q) = \frac{1}{(1+8+6)} \times \text{位时间} = \frac{1}{1+8+6} \times 0.00001 = 0.00000066$$

$$t_q = (\text{BRP}[5:0] + 1) \times \frac{1}{F_{can}}$$

$$(\text{BRP}[5:0] + 1) = \frac{t_q}{F_{can}} = t_q / (1/F_{can})$$

$$F_{can} = \frac{1}{\text{外部晶振}8M} = 0.000000125$$

$$(\text{BRP}[5:0] + 1) = t_q / F_{can}$$

$$(\text{BRP}[5:0] + 1) = 0.00000066 / 0.000000125 = 5.28$$

$$\text{BRP}[5:0] = 5.28 - 1 = 4.28$$

$$\text{BRP}[5:0] \approx 4$$

BTR2 寄存器设置后，我们根据 BRP 的值设置 BTR1 寄存器

CAN_P5 = 0x75; // CAN_BTR2_BS1=6 BS2=8

23.11.13 CAN位时间特性寄存器 (CAN_BTR1)

地址偏移: 值: 见表63

复位值: 0x40

7	6	5	4	3	2	1	0
SJW[1:0]		BRP[5:0]					
RW	RW	RW	RW	RW	RW	RW	RW

本寄存器只能在CAN硬件为初始化模式下由软件访问。

位7:6	SJW[1:0]: 重新同步跳跃宽度 为了重新同步, 该位域定义了CAN硬件在每位中可以延长或缩短多少个时间单元的上限。 跳跃宽度 = (SJW[1:0] + 1)。
位5:0	BRP[5:0]: 波特率分频器 该位域定义了时间单元(t_q)的时间长度 $t_q = (BRP[5:0]+1) / f_{CAN}$ 不管 $f_{CAN}=f_{CANEXT}$ 或 f_{MASTER} (参考CAN_BTR2寄存器中的CLK位) 更多关于位时间的信息, 请参考23.6.6节位

CAN_P4 = 0x05; // 同步跳跃 CAN_BTR1 BRP=5

这样波特率就设置好了。

$$\text{波特率} = 100\text{Kbps}$$

$$\text{位时间} = \frac{1}{\text{波特率}} = \frac{1}{100000} = 0.00001$$

$$\text{时间单元}(t_q) = \frac{1}{(1+8+6)} \times \text{位时间} = \frac{1}{1+8+6} \times 0.00001 = 0.00000066$$

这个 BTR2
寄存器不
变

$$t_q = (BRP[5:0]+1) \times \frac{1}{f_{can}}$$

如果我们设置其他的波
特率只需要改这个位时间

$$(BRP[5:0]+1) = \frac{t_q}{\frac{1}{f_{can}}} = t_q/(1/f_{can})$$

$$f_{can} = \frac{1}{\text{外部晶振8M}} = 0.000000125$$

$$(BRP[5:0]+1) = t_q / 0.000000125$$

$$(BRP[5:0]+1) = 0.00000066 / 0.000000125 = 5.28$$

$$BRP[5:0] = 5.28 - 1 = 4.28$$

$$BRP[5:0] \approx 4$$

得出结果修改 BTR1 寄存器
的 BRP 就可以了

100kbps 波特率就是下面这三条语句

CAN_FPSR = 0x06; // BTR1 和 BTR2 寄存器是在 PAGE 6, 所以我用 FPSR 来切换页面到 0x06

CAN_P4 = 0x05; // 同步跳跃 CAN_BTR1 BRP=5

CAN_P5 = 0x75; // CAN_BTR2 BS1=6 BS2=8

我们现在还是在第 6 页设置
过滤器的工作方式是在第 6 页设置

我们来设置过滤器

我们要将 CAN 接受数据的过滤器设置成屏蔽模式

CAN_P8 = 0x00; //CAN_FMR1 过滤器组 0 工作在屏蔽位

CAN_P8 就是 0x08

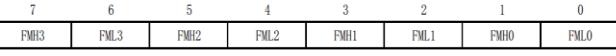
	PAGE 5	PAGE 6
0x00	CAN_MCSR	CAN_ESR
0x01	CAN_MDLCR	CAN_EIER
0x02	CAN_MIDR1	CAN_TECR
0x03	CAN_MIDR2	CAN_RECVR
0x04	CAN_MIDR3	CAN_BTR1
0x05	CAN_MIDR4	CAN_BTR2
0x06	CAN_MDAR1	Reserved
0x07	CAN_MDAR2	Reserved
0x08	CAN_FMR1	CAN_FMR1
0x09	CAN_MDAR4	CAN_FMR2
0x0A	CAN_MDAR5	CAN_FCR1
0x0B	CAN_MDAR6	CAN_FCR2
0x0C	CAN_MDAR7	CAN_FCR3
0x0D	CAN_MDAR8	Reserved
0x0E	CAN_MTSLR	Reserved
0x0F	CAN_MTSLRH	Reserved
Tx Mailbox 2 (if TXM2E=1 in CAN_DGR register)		
Configuration/Diagnostic		

23.11.16 CAN过滤器寄存器

CAN 过滤器主控寄存器 (CAN_FMR1)

地址偏移值: 见表63

复位值: 0x00



位1	FMOH: 过滤器0模式的高位 过滤器0的标识符/屏蔽位的高位寄存器的模式: 0: 高位寄存器工作在屏蔽位模式; 1: 高位寄存器工作在标识符列表模式。
位0	FML0: 过滤器0模式的低位 过滤器0的标识符/屏蔽位的低位寄存器的模式: 0: 低位寄存器工作在屏蔽位模式; 1: 低位寄存器工作在标识符列表模式。

我们发现过滤器 0, 低位 FML0 设置成了屏蔽位模式, 那么高位 FMHO 也必须为屏蔽位模式
不能 FML0 设置为标识符模式, 然后后 FMHO 设置为屏蔽位模式, 必须配对

图134 32位过滤器组设置(CAN_FCRx寄存器的FSCx位为11b)

Mapping	Filter registers						Filter mode ¹			
	STID[10:3] / EXID[28:21]	STID [2:0] / EXID[20:18]	EI RI	IDE	EXID [17:15]	EXID [14:7]	EXID[6:0]	0	FMHX = 0 FMLX = 0	FMHX = 1 FMLX = 1
Identifier	CAN_FxR1	CAN_FxR2		CAN_FxR3	CAN_FxR4		CAN_FxR5		ID	n
Identifier/Mask	CAN_FxR5	CAN_FxR6		CAN_FxR7	CAN_FxR8		CAN_FxR6		M	n+1
ID= Identifier n = Filter number M = Mask x = Filter bank number									这个过滤器一共有 6 组	

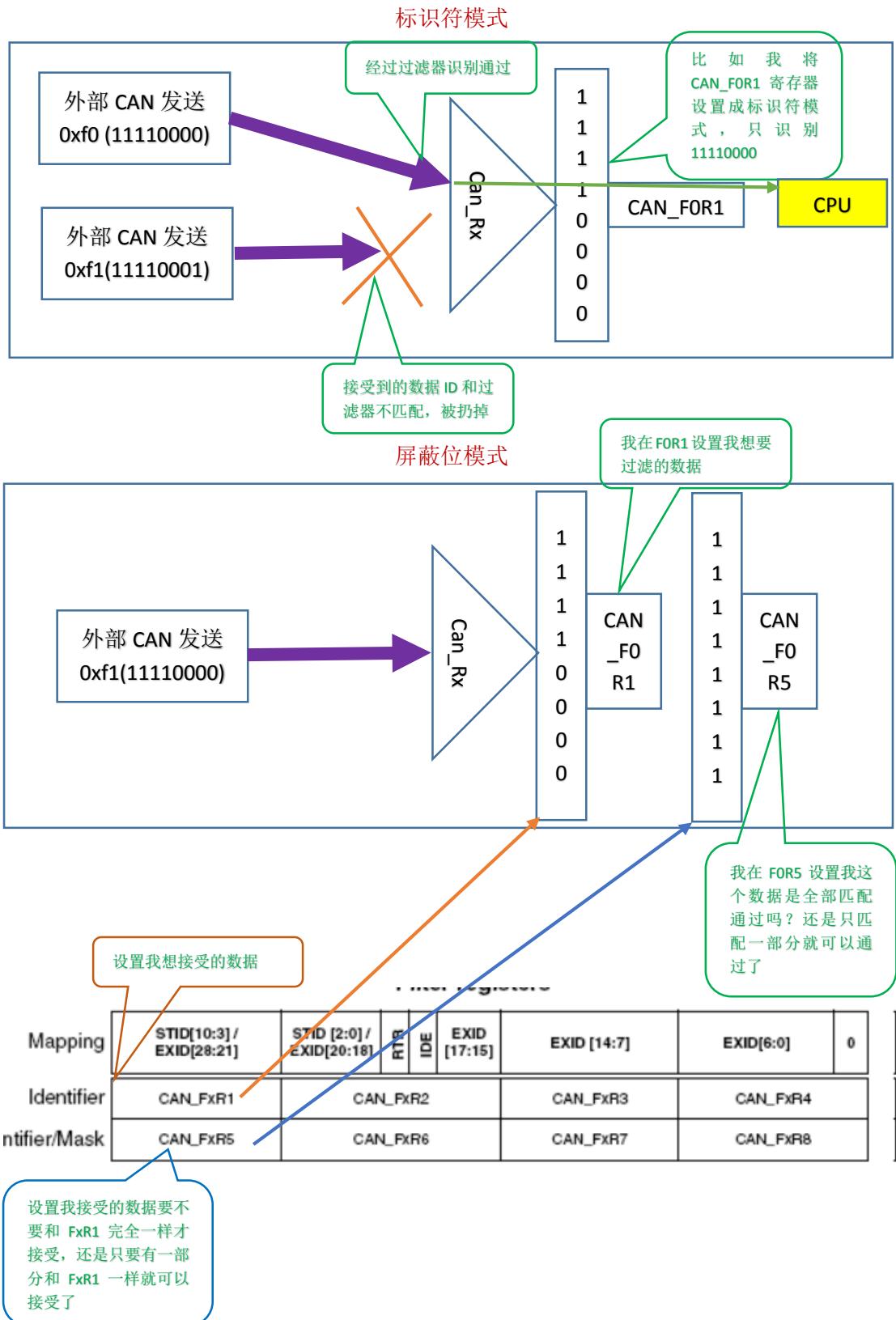
¹ The FMHX and FMLX bits are located in the CAN_FMR1 and CAN_FMR2 registers.

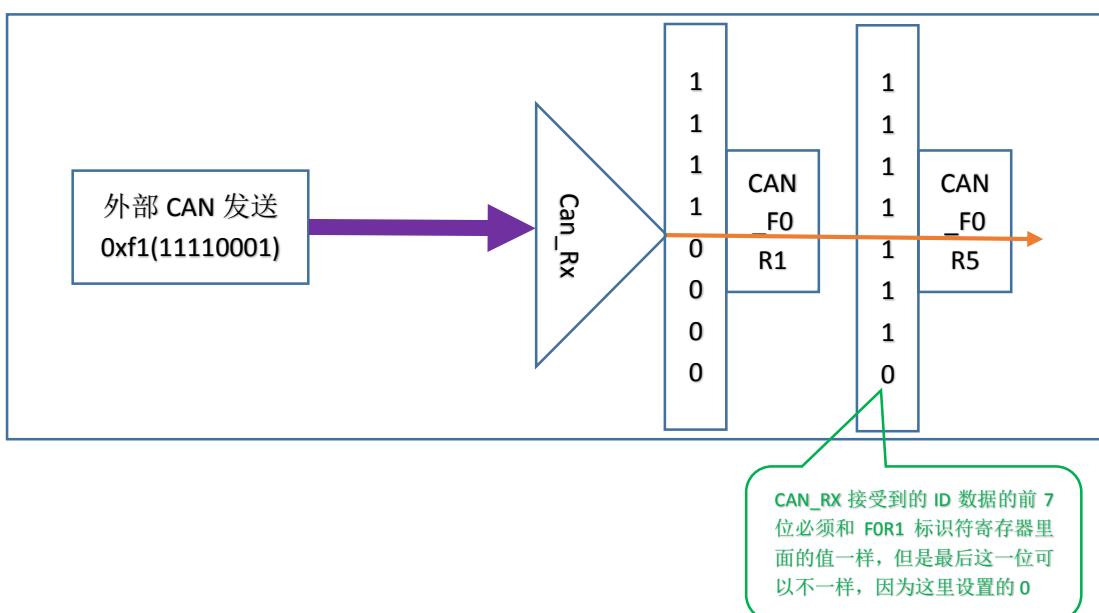
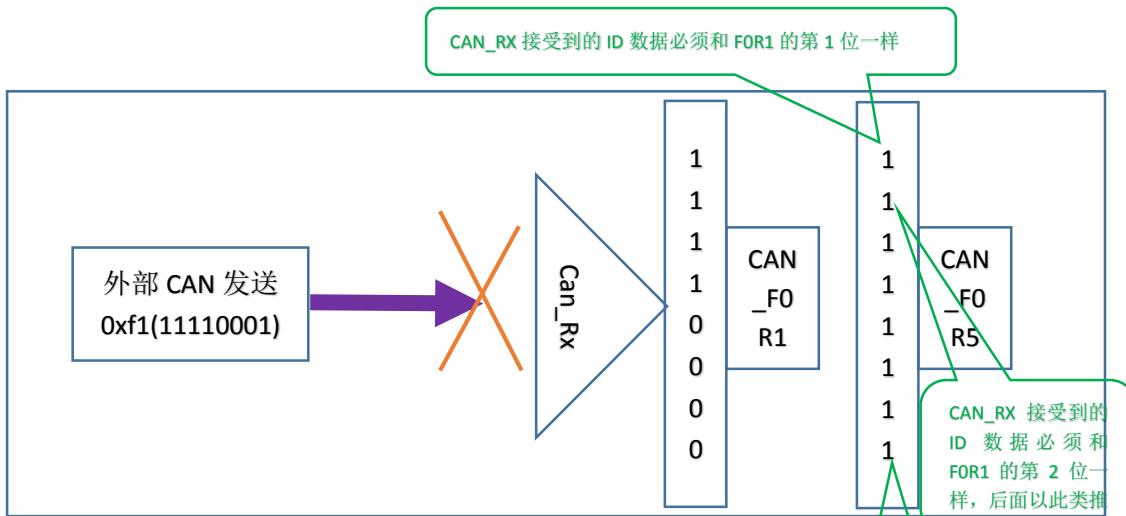
我们用的第 0 组

PAGE 2	PAGE 3	PAGE 4
CAN_F0R1	CAN_F2R1	CAN_F4R1
CAN_F0R2	CAN_F2R2	CAN_F4R2
CAN_F0R3	CAN_F2R3	CAN_F4R3
CAN_F0R4	CAN_F2R4	CAN_F4R4
CAN_F0R5	CAN_F2R5	CAN_F4R5
CAN_F0R6	CAN_F2R6	CAN_F4R6
CAN_F0R7	CAN_F2R7	CAN_F4R7
CAN_F0R8	CAN_F2R8	CAN_F4R8
CAN_F1R1	CAN_F3R1	CAN_F5R1
CAN_F1R2	CAN_F3R2	CAN_F5R2
CAN_F1R3	CAN_F3R3	CAN_F5R3
CAN_F1R4	CAN_F3R4	CAN_F5R4
CAN_F1R5	CAN_F3R5	CAN_F5R5
CAN_F1R6	CAN_F3R6	CAN_F5R6
CAN_F1R7	CAN_F3R7	CAN_F5R7
CAN_F1R8	CAN_F3R8	CAN_F5R8
Acceptance Filter 0:1		
Acceptance Filter 2:3		
Acceptance Filter 4:5		

CAN_P9 = 0x00; //P9 和 P8 差不多, 也是设置过滤器到底是标识符模式还是屏蔽位模式

什么是屏蔽位模式？什么是标识符模式？





这就是标识符和屏蔽位功能的区别，
标识符要求接收到 ID 的数据必须和我设置的过滤器 ID 值一模一样。
屏蔽位功能就不这么要求，只要求接收到的 ID 值和我过滤器 ID 值一部分一样就可以了，
也可以全部一样，就看你在过滤里面写 0 还是写 1.

CAN_PA = 0x06; //过滤器组 0 32 位 过滤器组 1 8 位

CAN 过滤器设置寄存器 (CAN_FCR1)

地址偏移量：见表63

复位值：0x00

7	6	5	4	3	2	1	0
保留 0	FSC11 0	FSC10 0	FACT1 0	保留 0	FSC01 1	FSC00 1	FACT0 0

RW RW RW RW RW RW RW

STM8S参考手册

位2:1	FSC1[1:0]: 过滤器0位宽设置 这两位定义了过滤器0的位宽。 设置为 11 就是让过滤器位 32 位宽
位0	FACT0: 过滤器0激活 软件将该位置位来激活过滤器0。修改过滤器0的寄存器(CAN_F0Rx)时，必须将该位清0。 0: 过滤器0被禁用； 1: 过滤器0被激活。 设置 0 激活过滤器

PAGE 2

CAN_F0R1
CAN_F0R2
CAN_F0R3
CAN_F0R4
CAN_F0R5
CAN_F0R6
CAN_F0R7
CAN_F0R8

我们前面讲的标识符和屏蔽位原理是用的 8 位寄存器 FOR1 来讲的，但是我们 CAN 总线 ID 最大可以做到 29 位，所以我们直接设置屏蔽过滤器位 32 位

Filter registers						
Mapping	STID[10:3] / EXID[28:21]	STID [2:0] / EXID[20:16]	RTR	ID[17:16]	EXID [14:7]	EXID[6:0]
Identifier	CAN_F0R1	CAN_F0R2		CAN_F0R3	CAN_F0R4	
Identifier/Mask	CAN_F0R5	CAN_F0R6		CAN_F0R7	CAN_F0R8	

ID= Identifier n = Filter number
M = Mask x = Filter bank number

¹ The FMRx and FMLx bits are located in the CAN_FMR1 and CAN_FMR2 registers

这样 FxR1, FxR2, FxR3, FxR4 就是 4 个 8 位寄存器相加，就是 32 位寄存器，这 4 个可以设置成我们想要通过的数据

然后 FxR5, FxR6, FxR7, FxR8 就是 4 个 8 位寄存器相加，就是 32 位寄存器，这 4 个可以设置成哪些位必须和标识符寄存器一样才能通过，哪些位和标识符寄存器不一样才能通过

CAN_PB = 0x00; //过滤器组 2 8 位 过滤器组 3 8 位

CAN_PC = 0x00; //过滤器组 4 8 位 过滤器组 5 8 位

过滤器 1~5 我们不用设置成 0

CAN_MCR &= 0xFE; //把 MCR 最低位 INRQ=0 CAN 进入正常模式

CAN 总线初始化设置完成了，我们在 CAN 初始化里面设置了 CAN 总线工作在回环模式，波特率，和用几个过滤器。

我们来设置过滤器到底要接受哪些数据

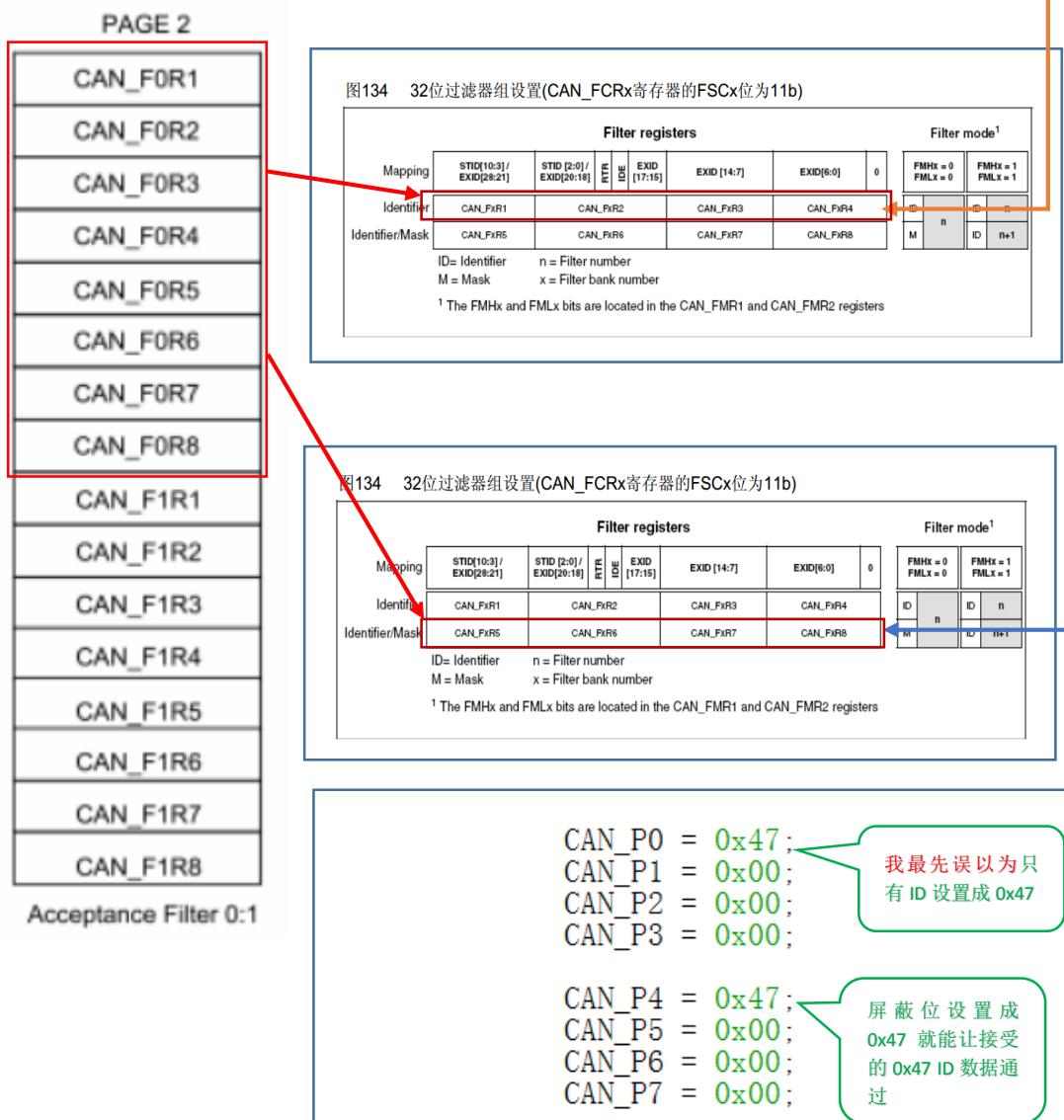
```

void beCAN_filter_init() //设置接收过滤器值
{
    /*因为滤波器寄存器在PAGE 2上，所以要先换页*/
    CAN_FPSR = 0x02; //PAGE 2上有过滤器0 过滤器1
    CAN_P0 = 0x3f;
    CAN_P1 = 0x00;
    CAN_P2 = 0x01;
    CAN_P3 = 0xfe;

    CAN_P4 = 0x3f;
    CAN_P5 = 0x00;
    CAN_P6 = 0x01;
    CAN_P7 = 0xfe;

    CAN_FPSR = 0x06; //设置完过滤器，切换到PAGE 6页去激活过滤器
    CAN_PA |= 0x01; //激活过滤器1
}

```



理论上上面的做法是没有问题的，标识符 0x47 和屏蔽位 0x47 是应该能让数据通过，但是在 STM8 是不行的。

图134 32位过滤器组设置(CAN_FCRx寄存器的FSCx位为11b)

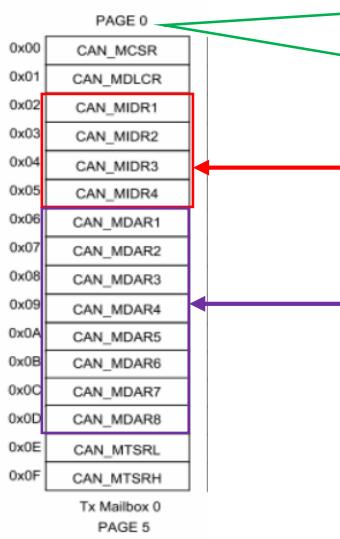
像 CAN_FXR1 只能写 EXID 的 28 位和 21 位，而 CAN_FxR2 在寄存器里面还加了不是存放 ID 的两位，所以不能简单的认为每个寄存器都是放 ID 位的，挨着顺序将 29 位 ID 写进去就 OK。

现在我们来测试发送数据和接受滤波功能

```
/*CAN数据发送函数*/
void beCAN_TX(unsigned char data1,unsigned char data2,unsigned char data3,
              unsigned char data4,unsigned char data5,unsigned char data6,
              unsigned char data7,unsigned char data8)
{
    /*我们选择发送邮箱0*/
    CAN_FPSR = 0x00; //选择PAGE 0页 发送邮箱0
    /*CAN总线ID设置*/
    CAN_P2 = 0x47; //CAN_MIDR1 EXID[28:24]= 0 0111 //CAN_P2不能乱写ID, 因为ID里面还包含了控制位, 乱写会出现接受乱码
    CAN_P3 = 0xec; //CAN_MIDR2 EXID[23:16]=1110 1100
    CAN_P4 = 0x00; //CAN_MIDR3 EXID[15:8]=1010 1100
    CAN_P5 = 0x00; //CAN_MIDR3 EXID[7:0]=0000 0000
    CAN_P1 = 0x08; //CAN_MDLCR CAN发送8个数据

    /*把数据放入8个寄存器*/
    CAN_P6 = data1;
    CAN_P7 = data2;
    CAN_P8 = data3;
    CAN_P9 = data4;
    CAN_PA = data5;
    CAN_PB = data6;
    CAN_PC = data7;
    CAN_PD = data8;

    CAN_P0 |= 0x01; //把邮箱0数据发送出去 TXRQ=1
    Delay_us(100);
}
```



邮箱有 3 个，我们选择 0 号邮箱来发送，0 号邮箱在 PAGE0

这是 4 个 CAN 设备 ID 寄存器，写代码用 P2~P5 来寻址

这是 8 个数据寄存器，写代码用 P6~PA 来寻址

CAN邮箱标识符寄存器 1(CAN_MIDR1)

地址偏移值: 见表58和表59

复位值: 未定义

7	6	5	4	3	2	1	0
保留	IDE	RTR	STID[10:6] / EXID[28:24]				
RW	RW	RW	RW	RW	RW	RW	RW

写发送 ID 数据特别 要注意这 3位	位7	保留位, 读出值为0。
	位6	IDE: 扩展标识符 该位决定发送邮箱中报文使用 *CAN数据发送函数*/ 0: 使用标准标识符; 1: 使用扩展标识符。 <code>void beCAN_TX(unsigned char data1, unsigned char data2, unsigned char data3, unsigned char data4, unsigned char data5, unsigned char data6, unsigned char data7, unsigned char data8)</code>
	位5	RTR: 远程发送请求 /*我们选择发送邮箱0*/ CAN_FPSR = 0x00; //选择PAGE 0页 发送邮箱0 /*CAN总线ID设置*/ CAN_P2 = 0x47; //CAN_MIDR1 EXID[28:24]= 0 0111 1110 110 0000 0000 0000 0000 CAN_P3 = 0xee; //CAN_MIDR2 EXID[23:16]=1110 1100 0000 0000 CAN_P4 = 0x00; //CAN_MIDR3 EXID[15:8]=0000 0000 0000 0000 CAN_P5 = 0x00; //CAN_MIDR3 EXID[7:0]=0000 0000
	位4:0	STID[10:6]: 标准标识符 标识符的标准部分的高5位。 或 EXID[28:24]: 扩展标识符 扩展标识符的“基本”部分的高5位。

位 7, 位 6, 位 5 是占用了发送寄存器高位的三位, 导致了不能认为 MIDR1 寄存器完全是用来写 ID 数据的。前面三位是 CAN ID 的功能设置, 后面 5 位才是写 ID 数据的位置。

所以在计算 ID 值的时候必须从第 4 位开始向后面计算。

我这一帧 CAN 的 ID 值是 0x7ec0000(0 0111 1110 110 0000 0000 0000 0000),

而不是 0x47ec0000

CAN邮箱标识符寄存器 2(CAN_MIDR2)

地址偏移值: 见表58和表59

复位值: 未定义

7	6	5	4	3	2	1	0
STID[5:0] / EXID[23:18]						EXID[17:16]	
RW	RW	WT	RW	RW	RW	RW	RW
STID[5:0]: 标准标识符 标识符的标准部分的低6字节。 或 EXID[23:18]: 扩展标识符 扩展标识符的“基本”部分的低6位。							
EXID[17:16]: 扩展标识符 扩展标识符的“扩展”部分的高2位。							

复位值: 未定义

7	6	5	4	3	2	1	0
EXID[15:8]							
RW	RW	RW	RW	RW	RW	RW	RW
EXID[7:0]							
RW	RW	RW	RW	RW	RW	RW	RW

因为发送 ID 是 29 位, 所以在接受过滤器这里也必须要注意过滤方法

```

void beCAN_filter_init() //设置接收过滤器值
{
    /*因为滤波器寄存器在PAGE 2上，所以要先换页*/
    CAN_FPSR = 0x02; //PAGE 2上有过滤器0 过滤器1
    CAN_P0 = 0x3f;
    CAN_P1 = 0x00;
    CAN_P2 = 0x00;
    CAN_P3 = 0xfe;

    CAN_P4 = 0x3f;
    CAN_P5 = 0x00;
    CAN_P6 = 0x00;
    CAN_P7 = 0x00;

    CAN_FPSR = 0x06; //设置完过滤器，切换到PAGE 6页去激活过滤器
    CAN_PA |= 0x01; //激活过滤器1
}

```

前面说了 接受过滤器要接受 0x7ec00000 的数据，这里不能写 0x47，而是写 0x3f，为什么呢？

CAN_P2 = 0x47; //CAN_MIDR1 EXID[28:24]= 0 0111
 CAN_P3 = 0xec; //CAN_MIDR2 EXID[23:16]=1110 1100
 我们要接受匹配的是 CAN_P2,P3 的 ID
 我们只取 29 位的高 8 位做过滤实验，所以就是 0011 1111

```

/*CAN数据发送函数*/
void beCAN_TX(unsigned char data1, unsigned char data2, unsigned
               char data4, unsigned char data5, unsigned
               char data7, unsigned char data8)

/*我们选择发送邮箱*/  

CAN_FPSR = 0x00; //选择PAGE 0页 发送邮箱0
/*CAN总线ID设置*/
CAN_P2 = 0x47; //CAN_MIDR1 EXID[28:24]= 0 0111 //CAN_P2不
CAN_P3 = 0xec; //CAN_MIDR2 EXID[23:16]=1110 1100
CAN_P4 = 0x00; //CAN_MIDR3 EXID[15:8]=0000 0000
CAN_P5 = 0x00; //CAN_MIDR3 EXID[7:0]=0000 0000

```

所以为什么发送端 P2 和 P3 都要写上 ID 数据，那是为了和过滤器的高 8 位 ID 匹配

所以 STM8 的 CAN 发送寄存器和接受寄存器都不是直观的，都是在寄存器里面加了一些不是 ID 的位，让你不是这么好写寄存器。

设置完发送功能，和接受过滤器后我们看 CAN 总线接受的数据放在哪里？

```

/*CAN数据接受函数*/
void beCAN_RX(struct CAN_RX_buf *rxbuf)
{
    if(CAN_RFR==0x01) //表示CAN总线收到数据了
    {
        /*FIFO1 在PAGE7 页面*/
        CAN_FPSR = 0x07; //切换到页面7
        Delay_us(100);
        rxbuf->ID[0]=CAN_P2;
        rxbuf->ID[1]=CAN_P3;
        rxbuf->ID[2]=CAN_P4;
        rxbuf->ID[3]=CAN_P5;
        rxbuf->data[0]=CAN_P6;
        rxbuf->data[1]=CAN_P7;
        rxbuf->data[2]=CAN_P8;
        rxbuf->data[3]=CAN_P9;
        rxbuf->data[4]=CAN_PA;
        rxbuf->data[5]=CAN_PB;
        rxbuf->data[6]=CAN_PC;
        rxbuf->data[7]=CAN_PD;
        Delay_us(100);

        CAN_RFR |=0x20; //释放邮箱
    }
}

```

PAGE 7
CAN_MFMR
CAN_MDLR
CAN_MIDR1
CAN_MIDR2
CAN_MIDR3
CAN_MIDR4
CAN_MDAR1
CAN_MDAR2
CAN_MDAR3
CAN_MDAR4
CAN_MDAR5
CAN_MDAR6
CAN_MDAR7
CAN_MDAR8
CAN_MTSRL
CAN_MTSRH

Receive FIFO

PAGE7 页看起来好像和发送邮箱一样，其实还是有不一样的地方，这个 7 页里面的寄存器就是用来做接受的

以上就是 CAN 总线的发送与接受功能说明。

对 CAN 总线的接受函数和发送函数进行封装。

我们看到，STM8CAN 寄存器的配置方法过于麻烦，所以对过滤器和发送功能，接受功能进行函数封装，方便调用。

```
void beCAN_filter_init_package(unsigned long ID) //设置接收过滤器值接口封装
{
    unsigned char FxR1, FxR2, FxR3, FxR4;
    CAN_FPSR = 0x06; //设置完过滤器，切换到PAGE 6页去激活过滤器
    CAN_PA |= 0x00; //关闭过滤器1
    /*因为滤波器寄存器在PAGE 2上，所以要先换页*/
    CAN_FPSR = 0x02; //PAGE 2上有过滤器0 过滤器1

    FxR1 = (unsigned char) (((ID>>24)&0x0f)<<3);
    FxR1 = (unsigned char) (((((ID>>16)>>5)&0x07) | FxR1));
    FxR2 = (unsigned char) (((ID>>16)<<1)&0xe7);
    FxR2 = (unsigned char) (((((ID>>8)>>7)&0x01) | FxR2));
    FxR3 = (unsigned char) ((ID>>8)<<1);
    FxR3 = (unsigned char) (((ID>>7)&(0x01)) | FxR3);
    FxR4 = (unsigned char) ((ID<<1)&0xfe);

    CAN_P0 = FxR1;
    CAN_P1 = FxR2;
    CAN_P2 = FxR3;
    CAN_P3 = FxR4;

    CAN_P4 = 0x0f; //0000 1111
    CAN_P5 = 0x00; //0000 0000
    CAN_P6 = 0xff; //1111 1111
    CAN_P7 = 0xff; //1111 1111

    CAN_FPSR = 0x06; //设置完过滤器，切换到PAGE 6页去激活过滤器
    CAN_PA |= 0x01; //激活过滤器1
}

/*CAN数据发送函数接口封装*/
void beCAN_TX_package(unsigned long ID,
                      unsigned char data1, unsigned char data2, unsigned char data3,
                      unsigned char data4, unsigned char data5, unsigned char data6,
                      unsigned char data7, unsigned char data8)
{
    unsigned char ID1=0;
    unsigned char ID2=0;
    unsigned char ID3=0;
    unsigned char ID4=0;

    ID1 = (unsigned char) (((ID>>24)&0x4f) | 0x40);
    ID2 = (unsigned char) ((ID>>16));
    ID3 = (unsigned char) ((ID>>8));
    ID4 = (unsigned char) (ID);

    /*我们选择发送邮箱0*/
    CAN_FPSR = 0x00; //选择PAGE 0页 发送邮箱0
    /*CAN总线ID设置*/
    CAN_P2 = ID1; //CAN_MIDR1 EXID[28:24] //CAN_P2不能乱写ID，因为ID里面还包含了控制位，乱写会出现接受乱码
    CAN_P3 = ID2; //CAN_MIDR2 EXID[23:16]
    CAN_P4 = ID3; //CAN_MIDR3 EXID[15:8]
    CAN_P5 = ID4; //CAN_MIDR3 EXID[7:0]

    CAN_P1 = 0x08; //CAN_MDLCR CAN发送8个数据

    /*把数据放入8个寄存器*/
    CAN_P6 = data1;
    CAN_P7 = data2;
    CAN_P8 = data3;
    CAN_P9 = data4;
    CAN_PA = data5;
    CAN_PB = data6;
    CAN_PC = data7;
    CAN_PD = data8;

    CAN_P0 |= 0x01; //把邮箱0数据发送出去 TXRQ=1
    Delay_us(100);
}
```

我们用的底 16 位做过滤器滤波,因为
高 13 位寄存器还没有实验出来,但
是底 16 位是可以的

封装后的过滤器和发送接口进行测试

```
int main( void )
{
    struct CAN_RX_buf rx_buf;

    CLK_start();
    UART1_init();
    beCAN_gpio_init();
    beCAN_bus_init();
    //beCAN_filter_init();
    beCAN_filter_init_package(5);
    rx_buf.ID[0]=0x00;
    rx_buf.ID[1]=0x00;
    rx_buf.ID[2]=0x00;
    rx_buf.ID[3]=0x00;
    rx_buf.data[0]=0x00;
    rx_buf.data[1]=0x00;
    rx_buf.data[2]=0x00;
    rx_buf.data[3]=0x00;
    rx_buf.data[4]=0x00;
    rx_buf.data[5]=0x00;
    rx_buf.data[6]=0x00;
    rx_buf.data[7]=0x00;

    while(1)
    {
        printf("can总线发送数据 0x11 0x22 0x33 0x44 0x55 0x66 0x77 0x88 \r\n");
        //beCAN_TX(0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88);
        beCAN_TX_package(5, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88);
        printf("\r\n");

        beCAN_RX(&rx_buf);
        printf("can总线接受数据 ID = %x, %x, %x, %x, DATA = %x, %x, %x, %x, %x, %x, %x\r\n",
               rx_buf.ID[0], rx_buf.ID[1], rx_buf.ID[2], rx_buf.ID[3], rx_buf.data[0],
               rx_buf.data[1], rx_buf.data[2], rx_buf.data[3], rx_buf.data[4]);
    }
}
```

can总线发送数据 0x11 0x22 0x33 0x44 0x55 0x66 0x77 0x88
can总线接受数据 ID = 40, 0, 0, 5, DATA = 11, 22, 33, 44, 55, 66, 77, 88

这就是测试结果

既然 ID 是 5 为什么高位不是 00 而是 40 呢?

CAN邮箱标识符寄存器 1(CAN_MIDR1)							
地址偏移值: 见 表58 和 表59							
复位值: 未定义							
7	6	5	4	3	2	1	0
保留	IDE	RTR		STID[10:6] / EXID[28:24]			
rw	rw	rw		rw	rw	rw	rw
位7	保留位。读出值为0。						
位6	IDE: 扩展标识符 该位决定发送邮箱中报文使用的标识符类型 0: 使用标准标识符; 1: 使用扩展标识符。						
位5	RTR: 远程发送请求 0: 数据帧; 1: 远程帧。						
位4:0	STID[10:6]: 标准标识符 标识符的“基本”部分的高5位。 或 EXID[28:24]: 扩展标识符 扩展标识符的“基本”部分的高5位。						

因为我们的发送寄存器高位的 IDE 必须写 1 才能是扩展标识符发送

using can::can_tut;

```
ID1 = (unsigned char )(((ID>>24)&0x4f) | 0x40);
ID2 = (unsigned char )(ID>>16);
ID3 = (unsigned char )(ID>>8);
TN4 = (unsigned char )(TN);
```

所以我在发送封装函数里面加了扩展标识符

```

int main( void )
{
    struct CAN_RX_buf rx_buf;

    CLK_start();
    UART1_init();
    beCAN_gpio_init();
    beCAN_bus_init();
    //beCAN_filter_init();
    beCAN_filter_init_package(4);
    rx_buf.ID[0]=0x00;
    rx_buf.ID[1]=0x00;
    rx_buf.ID[2]=0x00;
    rx_buf.ID[3]=0x00;
    rx_buf.data[0]=0x00;
    rx_buf.data[1]=0x00;
    rx_buf.data[2]=0x00;
    rx_buf.data[3]=0x00;
    rx_buf.data[4]=0x00;
    rx_buf.data[5]=0x00;
    rx_buf.data[6]=0x00;
    rx_buf.data[7]=0x00;

    while(1)
    {
        printf("can总线发送数据 0x11 0x22 0x33 0x44 0x55 0x66 0x77 0x88 \r\n");
        //beCAN_TX(0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88);
        beCAN_TX_package(5, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88);
        printf("\r\n");
    }
}

```

如果我想接受的 ID 是 4，我在过滤器里面写 4

但是我发送的是 5

```

can总线发送数据 0x11 0x22 0x33 0x44 0x55 0x66 0x77 0x88
can总线接受数据 ID = 0,0,0,0,DATA = 0,0,0,0,0,0,0,0

```

你看接受的 ID 不是 4 就会把接受的数据包扔掉，所以我们可以判断 ID 是不是 0 0 0 0 来确定数据包收到没得。

CAN 总线接受中断实验

```

void beCAN_bus_init() //初始化CAN总线
{
    CAN_MCR = 0x01; //CAN进入初始化模式
    while((CAN_MSR&0X01)==0X00); //等待CAN进入初始化模式
    Delay_us(100);
    CAN_MCR |= 0x50; //初始化模式设置
    CAN_DGR = 0x01; //设置CAN为环回模式

    /*设置CAN总线的波特率*/
    /*设置页面里面的寄存器就不能直接向寄存器里面写值，要先换页再写值*/
    CAN_FPSR = 0x06; //BTR1和BTR2寄存器是在PAGE 6,所以我用FPSR来切换页面到0x06
    CAN_P4 = 0x05; //同步跳跃 CAN_BTR1 BRP=5
    CAN_P5 = 0x75; //CAN_BTR2 BS1=6 BS2=8

    /*设置过滤器*/
    CAN_P8 = 0x00; //CAN_FMR1 过滤器组0工作在屏蔽位
    CAN_P9 = 0x00;
    CAN_PA = 0x06; //过滤器组0 32位 过滤器组1 8位
    CAN_PB = 0x00; //过滤器组2 8位 过滤器组3 8位
    CAN_PC = 0x00; //过滤器组4 8位 过滤器组5 8位
    Delay_us(100);

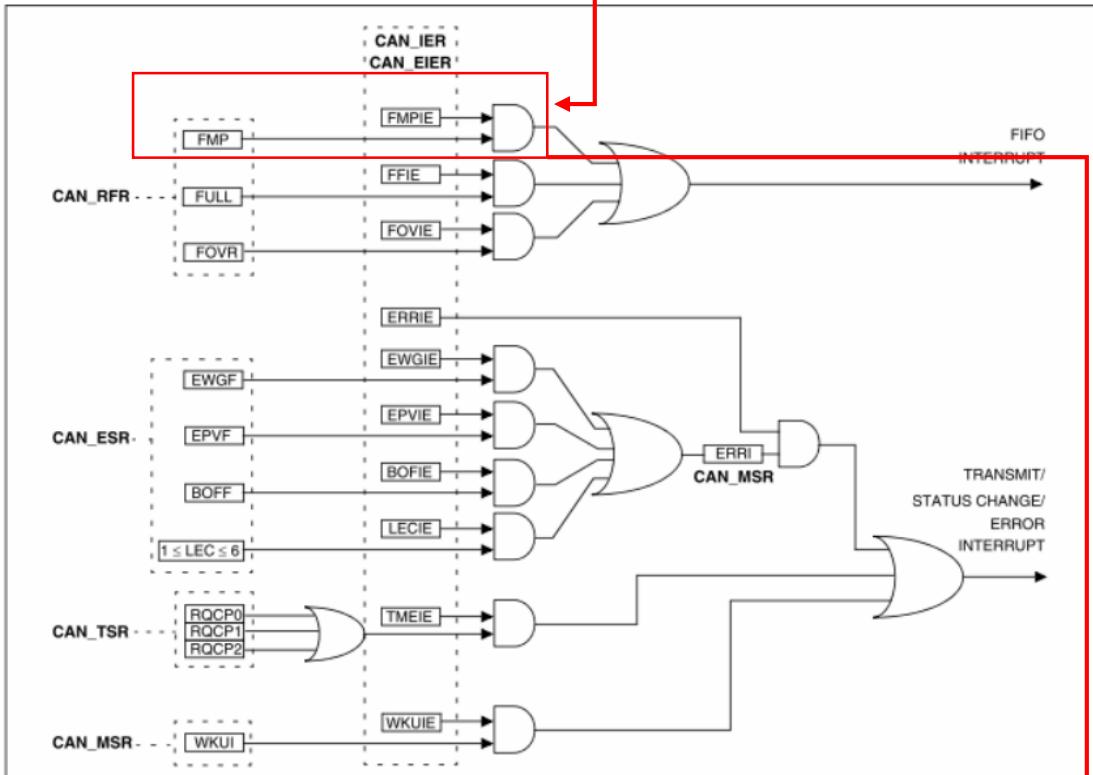
    CAN_MCR &= 0xFE; //把MCR最低位INRQ=0 CAN进入正常模式
    CAN_IER = 0x02; //打开CAN总线接受中断
}

```

我们在先前的
CAN 总线初始化
函数里面打开
CAN 中断功能

CAN_IER = 0x02;

图142 事件标志和中断产生



23.11.6 CAN中断允许寄存器 (CAN_IER)

地址偏移值: 0x05

复位值: 0x00

	7	6	5	4	3	2	1	0
	WKUIE		保留		FOVIE	FFIE	FMPIE	TMEIE
r				r	r	r	r	r

位1

FMPIE: FIFO消息挂号中断允许
0: 当FIFO的FMP[1:0]位由00b转变为01b时, 没有中断产生;
1: 当FIFO的FMP[1:0]位由00b转变为01b时, 产生中断。

```
#pragma vector=beCAN_FMP_vector
interrupt void CAN()
{
    printf("进入CAN接受中断");
    beCAN_RX(&rx_buf);
    CAN_RFR=(1<<3);
}
```

如果不写这位 CAN 总线在接受到第一位数据后就会不断的进入中断

FULL1: FIFO满
当有3个报文被存入FIFO 1时, 硬件对该位置1。
软件对该位写'1'、或通过RFOM释放FIFO, 可以将其清0。

23.11.5 CAN接收FIFO 1 寄存器(CAN_RFR)

地址偏移值: 0x04

复位值: 0x00

	7	6	5	4	3	2	1	0
	保留		RFOM	FOVR	FULL	保留		FMP[1:0]
rs			rc_w1	rc_w1	r			r

CAN_RFR=(1<<3); 写了这位接受 FIFO 里面的数据会被清空，所以一定要在中断里面执行 BeCAN_RX 函数把数据从接受 fifo 读出来。

```
beCAN_filter_init_package(5);
rx_buf.ID[0]=0x00;
rx_buf.ID[1]=0x00;
rx_buf.ID[2]=0x00;
rx_buf.ID[3]=0x00;
rx_buf.data[0]=0x00;
rx_buf.data[1]=0x00;
rx_buf.data[2]=0x00;
rx_buf.data[3]=0x00;
rx_buf.data[4]=0x00;
rx_buf.data[5]=0x00;
rx_buf.data[6]=0x00;
rx_buf.data[7]=0x00;

while(1)
{
    printf("can总线发送数据 0x11 0x22 0x33 0x44 0x55 0x66 0x77 0x88 \r\n");
    beCAN_TX_package(5, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88);
```

```
can总线发送数据 0x11 0x22 0x33 0x44 0x55 0x66 0x77 0x88
进入CAN接受中断can总线接受数据 ID = 40, 0, 0, 5, DATA = 11, 22, 33, 44, 55, 66, 77, 88
```

你看只要过滤器和发送的 ID 对应上，就能正常接收数据

```
int main( void )
{

    CLK_start();
    UART1_init();
    beCAN_gpio_init();
    beCAN_bus_init();
    //beCAN_filter_init();
    beCAN_filter_init_package(4);
    rx_buf.ID[0]=0x00;
    rx_buf.ID[1]=0x00;
    rx_buf.ID[2]=0x00;
    rx_buf.ID[3]=0x00;
    rx_buf.data[0]=0x00;
    rx_buf.data[1]=0x00;
    rx_buf.data[2]=0x00;
    rx_buf.data[3]=0x00;
    rx_buf.data[4]=0x00;
    rx_buf.data[5]=0x00;
    rx_buf.data[6]=0x00;
    rx_buf.data[7]=0x00;

    while(1)
    {
        printf("can总线发送数据 0x11 0x22 0x33 0x44 0x55 0x66 0x77 0x88 \r\n");
        beCAN_TX_package(5, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88);
```

```
can总线发送数据 0x11 0x22 0x33 0x44 0x55 0x66 0x77 0x88
can总线接受数据 ID = 0, 0, 0, 0, DATA = 0, 0, 0, 0, 0, 0, 0, 0
```

只要过滤器和发送端 ID 对不上，那么 CAN 接受中断程序就不会执行。

STM8 库函数工程建立

去官方网站 www.stm8mcu.org



- 所有资料 更多
- ▶ 官网资料.STM32F0 (64 NEW)
 - ▶ 官网资料.STM32F1 (49 NEW)
 - ▶ 官网资料.STM32F2 (45 NEW)
 - ▶ 官网资料.STM32F3 (53 NEW)
 - ▶ 官网资料.STM32F4 (60 NEW)
 - ▶ 官网资料.STM32F7 (50 NEW)
 - ▶ 官网资料.STM32L0 (51 NEW)
 - ▶ 官网资料.STM32L1 (47 NEW)
 - ▶ 官网资料.STM32L4 (46 NEW)
 - ▶ 官网资料.STM32L4+ (49 NEW)
 - ▶ 官网资料.STM32H7 (51 NEW)
 - ▶ **官网资料.STM8S (3 NEW)**

官网资料.STM8S

I. 芯片文档

II. 固件、软件、工具资源

III. 硬件相关资料

A dropdown menu on the left side of the page. The second item, "II. 固件、软件、工具资源", is highlighted with a red box.

STSW-STM8069 无 无

文档说明 : STM8S /标准外设库 (STM8S/A Standard peripheral library)

[点击进去](#)

« < 1 2 3 4 > »

STSW-STM8069

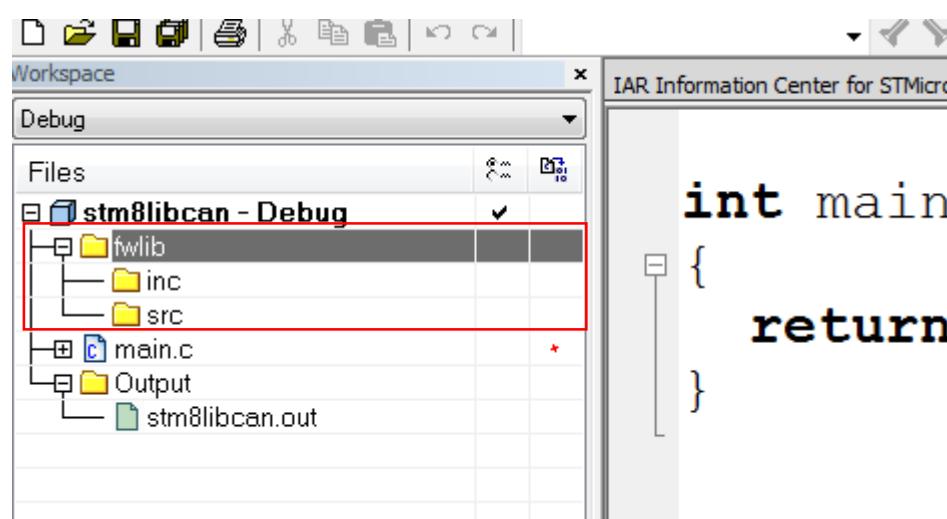
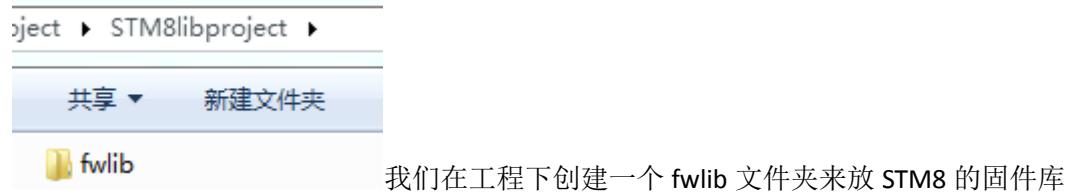
更新时间 2015-01-15

STM8S /标准外设库
(STM8S/A Standard peripheral library)

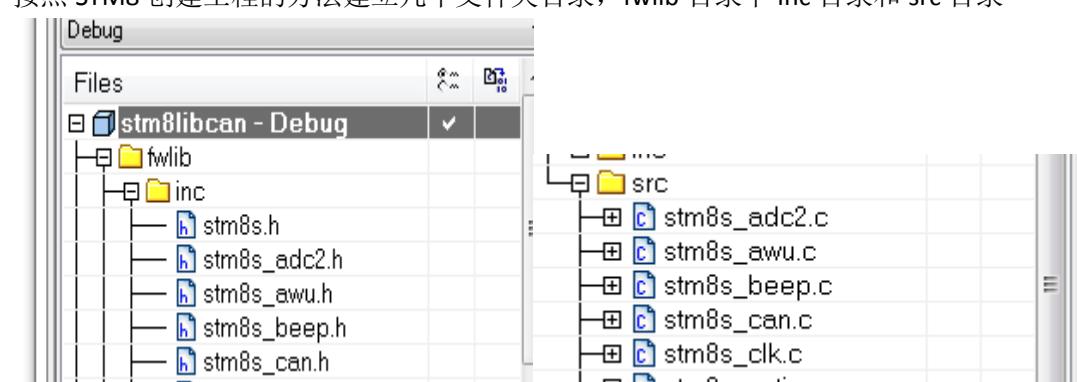
[点击下载](#)

类型	文档标题	格式	版本	文件大小	下载次数
附件	stsw-stm8069	zip	2.2.0	10.65MB	10953

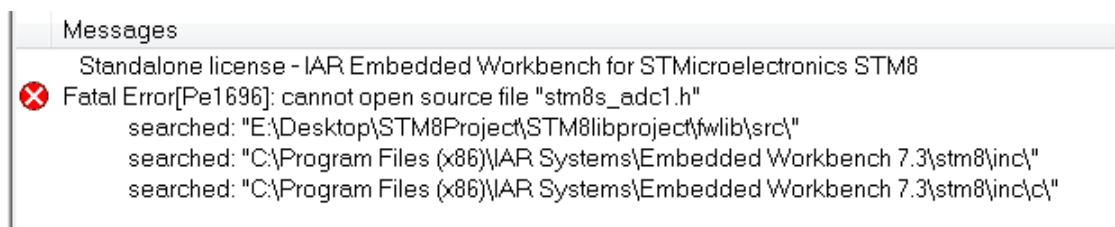




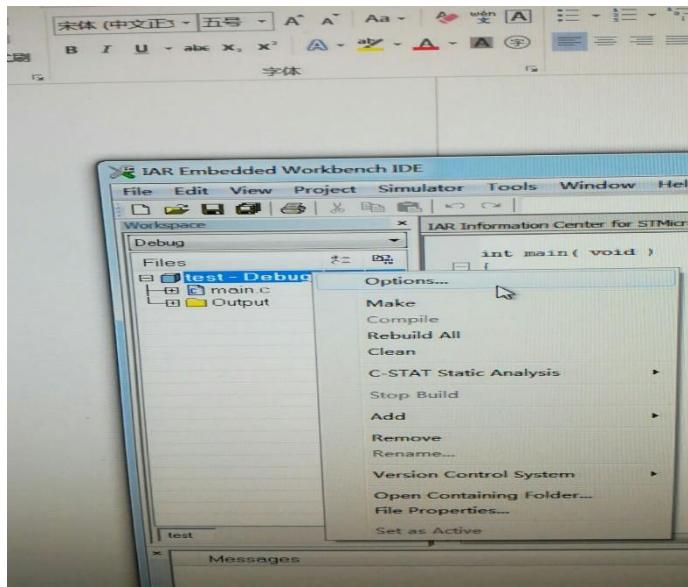
按照 STM8 创建工程的方法建立几个文件夹目录， fwlib 目录下 inc 目录和 src 目录



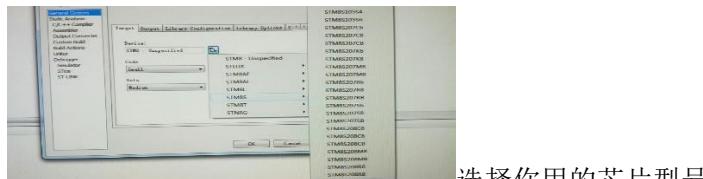
在工程里面添加 inc 固件库的 h 文件和 src 的固件库 c 文件



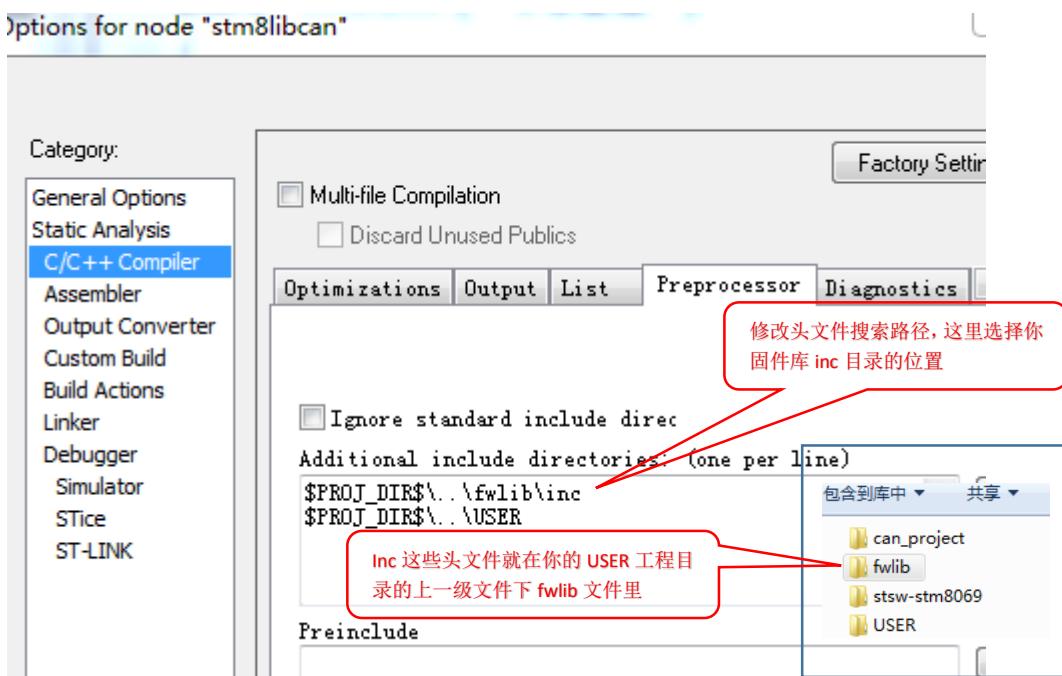
用 make 编译的时候发现很多 h 文件找不到，这是因为没有指定这些头文件的路径
这是因为你工程里面加了很多 h 文件，但是你没有告诉编译器 h 文件的存放路径

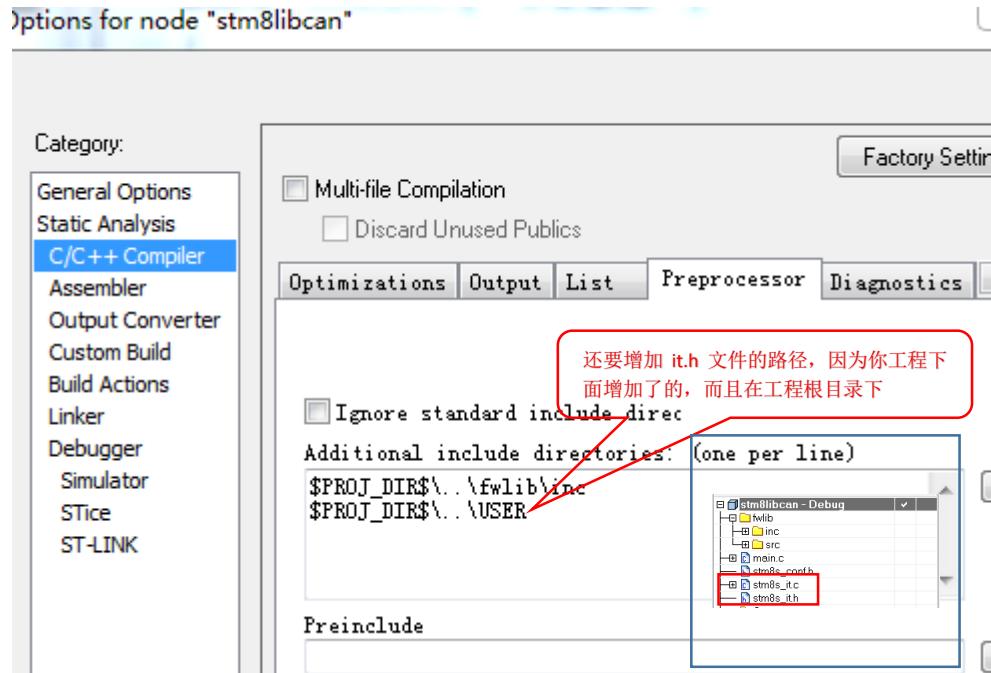


和上面一样在 options 里面设置



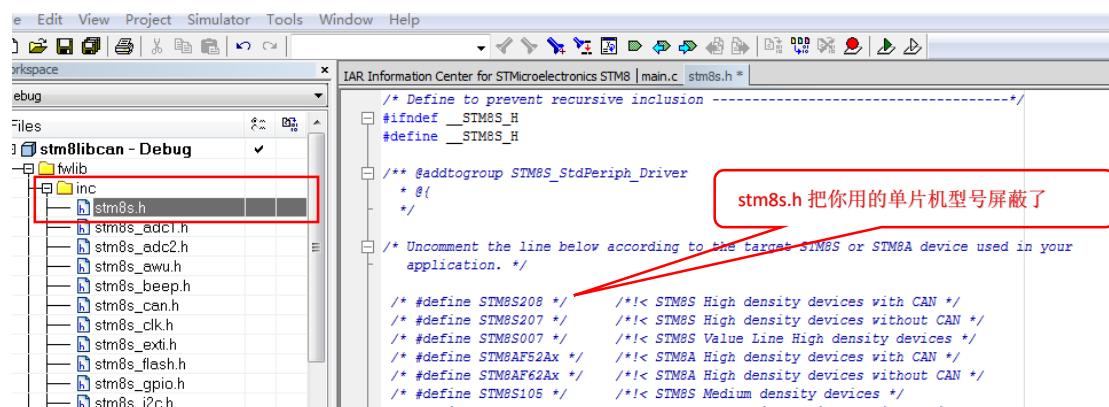
选择你用的芯片型号





IAR C/C++ Compiler V2.20.3.189 for STM8
Copyright 2010-2017 IAR Systems AB.
Standalone license - IAR Embedded Workbench for STMicroelectronics STM8
Fatal Error[Pe035]: #error directive: "Please select first the target STM8S/A device used in your application (in stm8s.h file)"

再次 make 发现没有在头文件里面声明你用的单片机型号



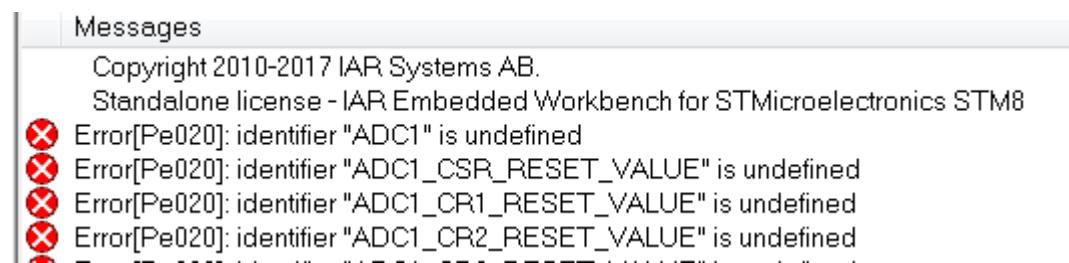
/* Uncomment the line below according to
application. */

#define STM8S208 /*!< STM8S High
/* #define STM8S207 */ /*!< STM8S
/* #define STM8S007 */ /*!< STM8S
/* #define STM8AF52AX */ /*!< STM8A High density devices with CAN
/* #define STM8AF62AX */ /*!< STM8A High density devices without CAN
/* #define STM8S105 */ /*!< STM8S Medium density devices
/* #define STM8C005 */ /*!< STM8C Value Line Medium density devices */

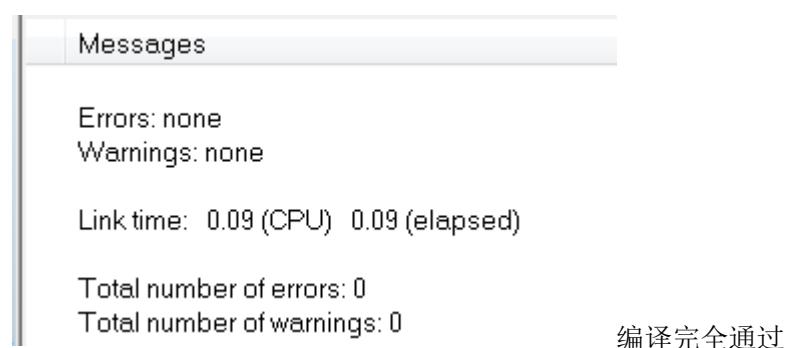
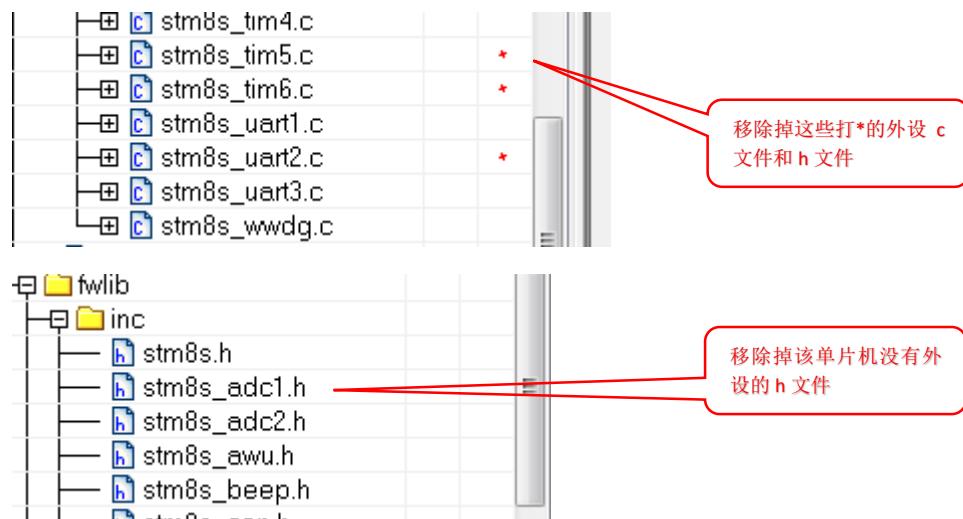
Standalone license - IAR Embedded Workbench for STMicroelectronics STM8

Error[Pe020]: identifier "UART4" is undefined
Error[Pe020]: identifier "UART4_BRR2_RESET_VALUE" is undefined
Error[Pe020]: identifier "UART4_BRR1_RESET_VALUE" is undefined
Error[Pe020]: identifier "UART4_CR1_RESET_VALUE" is undefined

然后 make 又报错，这个错误是你选择的是 stm8s208 单片机，但是该单片机没有固件库里面的 UART4 外设



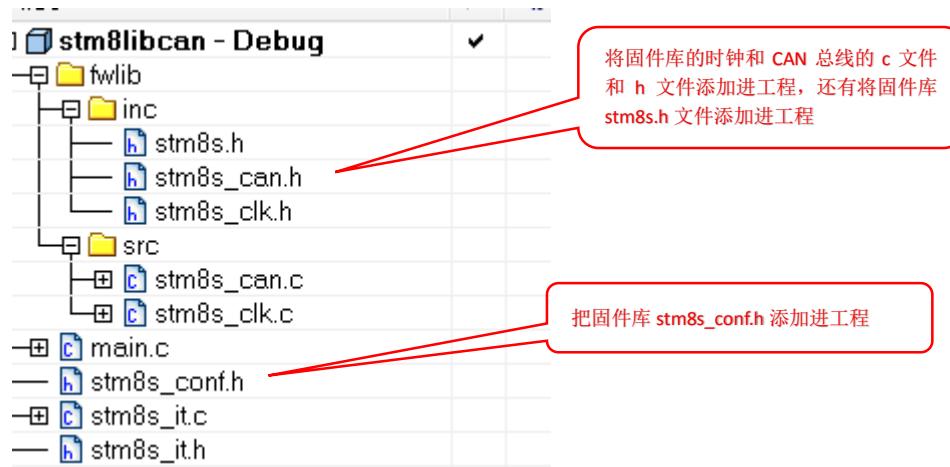
该单片机没有 ADC1 外设，固件库是把 STM8 单片机系列所有型号的外设都加进去了，但是我们用的 STM8S208 单片机没有这些多余的外设，所以我们要在工程中移除。



然后按照文档第最前面创建工程的要求，将 hex 文件输出路径，ST-LINK 调试设置，C/C++ 编译优化都设置一遍，然后再 make 就可以了。

STM8 寄存器操作 CAN 在波特率上有些问题，所以要用到 STM8 库

函数方案，实现库函数寄存器双用的工程建立



我们先编译空文件

Error[Li005] : no definition for "assert_failed" 报错

```
IAR ELF Linker V2.20.1.176 for STM8
Copyright 2010-2015 IAR Systems AB.
✖ Error[Li005]: no definition for "assert_failed" [referenced from E:\Desktop\STM8Project\STM8libproject\USER\Debug\Obj\stm8s_can.o]
```

该错误是在工程加入了 `stm8s_conf.h` 造成的。

在 `main.c` 中包含 `stm8s_conf.h` 可能是官方的 BUG

我们需要去 `stm8s_conf.h` 中注释 `#define USE_FULL_ASSERT`

```
Standard peripheral Library drivers code
//#define USE_FULL_ASSERT      (1)
```

```
Copyright 2007-2015 IAR Systems AB.
Loading E:\Desktop\STM8Project\STM8libproject\US
Saving ihex file to E:\Desktop\STM8Project\STM8libp
```

Total number of errors: 0

Total number of warnings: 0

编译通过

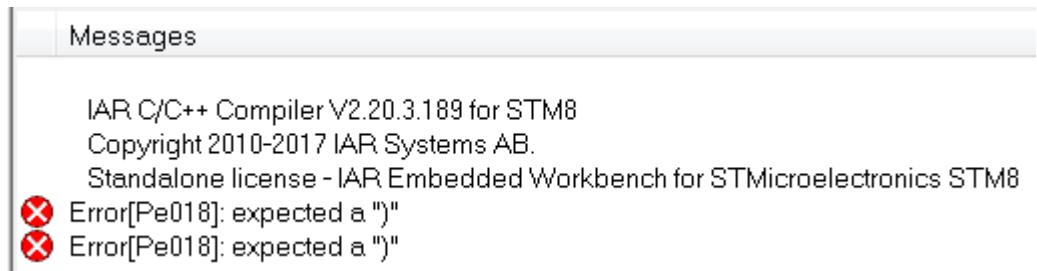
我们现在给 `main` 函数添加头文件

```
#include "stm8s.h" //引入STM8的头文件
#include "stm8s_can.h"
#include "stm8s_clk.h"
#include<iostm8s208rb.h>

/* *****
* 函数名: CAN_Configuration
```

然后 make

报错 Error[Pe018]: expected a ")" "



这种错误有两种情况，一种是 define 定义的后面加了分号，但是我们这里是系统头文件和本地头文件调用顺序错误

The screenshot shows two code snippets in the IAR Information Center for STM8. The top snippet is from 'stm8s_conf.h' and the bottom one is from 'main.c'. A red callout box highlights the line '#include<iostm8s208rb.h>' in 'main.c' with the text: 'C 语言规定打<>括号的头文件必须在""头文件之前' (C language规定打<>括号的头文件必须在""头文件之前).

```
#include "stm8s.h" // 引入STM8的头文件
#include "stm8s_can.h"
#include "stm8s_clk.h"
#include<iostm8s208rb.h>

/****************** 函数名: CAN_Configuration ********************/
+   +-----+-----+-----+-----+-----+-----+-----+
```



```
#include<iostm8s208rb.h>
#include "stm8s.h" // 引入STM8的头文件
#include "stm8s_can.h"
#include "stm8s_clk.h"
```

这样就编译通过了

接下来我们给 STM8 工程增加 CAN 库函数操作代码和 CAN 寄存器操作代码，混合编程

```
/***************** 延时ms函数 *****/
void Delay_ms(unsigned int ms)
{
    unsigned int x,i;
    for(x=ms;x>0;x--)
    {
        for(i=1550;i>0;i--);
    }
}

/***************** 延时3us函数 *****/
void Delay_us(unsigned int us)
{
    while(us--);
}
```

寄存器延时函数

```

/*
*   函数名: CAN_Configuration
*   功能说明: CAN初始化
*   形参: 无
*   返回值: 无
*/
void CAN_Configuration(void)
{
    //初始化CAN波特率为1M
    CAN_Init(CAN_MasterCtrl_AllDisabled,CAN_Mode_Normal,CAN_SynJumpWidth_1TimeQuantum,CAN_BitSeg1_11TimeQuantum,
              CAN_BitSeg2_4TimeQuantum,1);
    //配置CAN第0组过滤器为屏蔽模式。但是屏蔽位都设置为0，所以相当于没启动屏蔽功能。
    CAN_FilterInit(CAN_FilterNumber_0,ENABLE,CAN_FilterMode_IdMask,CAN_FilterScale_32Bit,0,0,0,0,0,0,0,0);
    //使能CAN接收中断功能
    CAN_ITConfig(CAN_IT_FMP,ENABLE);
}

```

库函数初始化 CAN 波特率，这个波特率是关键，我前面用寄存器操作 CAN 总线就是波特率设置没计算对。

```

void BSP_Configuration(void)
{
    CLK_HSI_PrescalerConfig(CLK_PRESCALER_HSIDIV1); //时钟速度为内部16M, 1分频,
    CAN_Configuration(); //调用CAN初始化函数
    rim(); //打开总中断
}

```

初始化 STM8 系统时钟，我们用的 STM8 内部 16M 时钟，然后初始化 CAN，然后打开中断

```

void beCAN_gpio_init() //初始化CAN总线IO口
{
    PG_DDR = 0x01; //PG0是CAN TX设置为输出
    PG_CR1 = 0x01; //PG0输出为推挽
    PG_CR2 = 0x00; //PG0输出速度2M
    /*PG1是CAN_RX 默认是输入模式*/
}

```

寄存器方式初始化 STM8 上的 CAN 引脚

```

int main( void )
{
    beCAN_gpio_init();
    BSP_Configuration(); //硬件驱动初始化函数
    while(1)
    {
        beCAN_TX(0x11,0x22,0x33,0x44,0x55,0x66,0x77,0x88);
        Delay_ms(500);
    }
    return 0;
}

```

CAN 数据发送函数还是用寄存器方式的发送函数，没有问题。

CAN 总线寄存器和库函数中断接收混合编程

```
void BSP_Configuration(void)
{
    CLK_HSI_PrescalerConfig(CLK_PRESCALER_HSIDIV1); //时钟速度为内部16M, 1分频,
    CAN_Configuration(); //调用CAN初始化函数
    rim(); //打开总中断
    CAN_IER = 0x02; //打开CAN总线接受中断
}

int main( void )
{
    beCAN_gpio_init();
    BSP_Configuration(); //硬件驱动初始化函数
    beCAN_filter_init_package(0); //选择寄存器版本的 CAN 接受滤波函数

    while(1)
    {
        //beCAN_TX(0x11,0x22,0x33,0x44,0x55,0x66,0x77,0x88);

        Delay_ms(500);
    }
    return 0;
}
```

在初始化函数里面打开总中断，然后再打开 CAN 接收中断

选择寄存器版本的 CAN 接受滤波函数

因为用了固件库框架，所以寄存器中断就编译不过，在 it.c 里面找到 CAN 接收中断函数

```
- stm8s_it.c
- stm8s_it.h

//INTERRUPT_HANDLER(CAN_RX_IRQHandler, 8)
//{
/* In order to detect unexpected events during development,
   it is recommended to set a breakpoint on the following instruction.
*/
//}
```

因为 it.c 里面中断函数接口太多了不方便观看，我们把 CAN_RX 中断接受函数拷贝到主函数里面，然后把 it.c 里面的 CAN_RX 中断接受函数屏蔽了以免发生编译冲突。

```
INTERRUPT_HANDLER(CAN_RX_IRQHandler, 8)
{
/* In order to detect unexpected events during development,
   it is recommended to set a breakpoint on the following instruction.
*/
int ID;
beCAN_RX(&rx_buf);
ID = (rx_buf.ID[0] | rx_buf.ID[1] | rx_buf.ID[2] | rx_buf.ID[3]);
beCAN_TX_package(ID, rx_buf.data[0], rx_buf.data[1], rx_buf.data[2], rx_buf.data[3],
                  rx_buf.data[4], rx_buf.data[5], rx_buf.data[6], rx_buf.data[7]);
}
```

将中断接受的 CAN 数据转发回电脑

```
INTERRUPT_HANDLER(CAN_RX_IRQHandler, 8)
{
    /* In order to detect unexpected events during development,
       it is recommended to set a breakpoint on the following instruction.
    */
    unsigned long ID, ID1;
    beCAN_RX(&rx_buf);

    ID1 = (unsigned long)rx_buf.ID[0];
    ID = ID1<<24;
    ID1 = (unsigned long)rx_buf.ID[1];
    ID = ID|(ID1<<16);
    ID1 = (unsigned long)rx_buf.ID[2];
    ID = ID|(ID1<<8);
    ID = ID|((unsigned long)rx_buf.ID[3]);

    beCAN_TX_package(ID, rx_buf.data[0], rx_buf.data[1], rx_buf.data[2], rx_buf.data[3],
                     rx_buf.data[4], rx_buf.data[5], rx_buf.data[6], rx_buf.data[7]);
}
```

中断发生之后用 beCAN_RX 寄存器操作函数来接受 CAN 总线的数据，然后解析再次用 beCAN_TX_package 转发出去。

CAN 总线固件库函数屏蔽滤波功能

```
void CAN_Configuration(void)
{
    //初始化CAN波特率为1M
    CAN_Init(CAN_MasterCtrl_AllDisabled, CAN_Mode_Normal, CAN_SynchroWidth_1TimeQuantum, CAN_BitSeg1_11TimeQuantum,
              CAN_BitSeg2_4TimeQuantum, 1);
    //配置CAN第0组过滤器为屏蔽模式。但是屏蔽位都设置为0，所以相当于没启动屏蔽功能。
    //CAN_FilterInit(CAN_FilterNumber_0, ENABLE, CAN_FilterMode_IdMask, CAN_FilterScale_32Bit, 0, 0, 0x7f, 0xff, 0, 0, 0xff, 0xff);
    CAN_FilterInit_socket(2);

    //使能CAN接收中断功能
    //CAN_ITConfig(CAN_IT_FMP, ENABLE);
}
```

这个数值就是你运行通过过滤器的数值，其他数值无法通过

在 CAN 初始化配置函数里面添加 CAN_FilterInit_socket 函数

接口函数的实现如下

```
void CAN_FilterInit_socket(unsigned long ID)//CAN固件库接口封装
{
    unsigned char FxR1, FxR2, FxR3, FxR4;
    FxR1 = (unsigned char)((ID>>24)&0x0f)<<3;
    FxR1 = (unsigned char)((((ID>>16)>>5)&0x07)|FxR1);
    FxR2 = (unsigned char)((ID>>16)<<1)&0xe7;
    FxR2 = (unsigned char)((((ID>>8)>>7)&0x01)|FxR2);
    FxR3 = (unsigned char)((ID>>8)<<1);
    FxR3 = (unsigned char)((ID>>7)&(0x01))|FxR3;
    FxR4 = (unsigned char)((ID<<1)&0xfe);

    CAN_FilterInit(CAN_FilterNumber_0, ENABLE, CAN_FilterMode_IdMask, CAN_FilterScale_32Bit, FxR1, FxR2, FxR3, FxR4, 0x0f, 0, 0xff, 0xff);
}
```

因为固件库函数也是和寄存器函数一样去操作过滤器寄存器组，我们知道过滤器寄存器组有点奇葩，所以要先进行数值运算，然后才能放进过滤器

CAN_P0 = FxR1;
CAN_P1 = FxR2;
CAN_P2 = FxR3;
CAN_P3 = FxR4;

类似用寄存器方式操作过滤器的 P0~P3

CAN_P4 = 0x0f; //0000 1111
CAN_P5 = 0x00; //0000 0000
CAN_P6 = 0xff; //1111 1111
CAN_P7 = 0xff; //1111 1111

类似用寄存器方式操作过滤器的 P4~P7

STM8CAN 总线固件库函数波特率计算

CAN 波特率=APB1 总线频率/(BRP 分频器系数*(1+tBS1+tBS2))



CAN 总线的波特率=PCLK1/((CAN_SJW +CAN_BS1 + CAN_BS2)*CAN_Prescaler)

PCLK1: STM8 系统主时钟, 可以是 STM8 内部 16M 时钟, 也可以是外部 8M 晶振

Sync: 值固定为 1

CAN_SJW: 设置范围 1~4, 一般设置为 1, 尽量别动。

CAN_BS1: 设置范围 1~16

CAN_BS2: 设置范围 1~8

设置注意: BS1 ≥ BS2(BS1 值必须大于等于 BS2), BS2 ≥1 个 CAN 时钟周期, BS2 ≥ 2SJW

公式分解:

$$\frac{\text{PCLK1}}{(\text{CAN}_\text{SJW} + \text{CAN}_\text{BS1} + \text{CAN}_\text{BS2}) \times \text{CAN}_\text{Prescaler}} = \text{kbps(波特率)}$$

这个我一般设置为 1
需要修改
需要修改
需要修改

STM8 内部时钟(或者晶振)

例如: 我 STM8 使用的是内部 16M 时钟, 求 1Mbps 波特率

$$\frac{\text{PCLK1}}{(\text{CAN}_\text{SJW} + \text{CAN}_\text{BS1} + \text{CAN}_\text{BS2}) \times \text{CAN}_\text{Prescaler}} = \text{kbps(波特率)}$$

$$\frac{16000(16\text{M})}{1000 (1\text{M 波特率})} = 16$$

$$(\text{CAN}_\text{SJW} + \text{CAN}_\text{BS1} + \text{CAN}_\text{BS2}) \times \text{CAN}_\text{Prescaler} = 16$$
$$(1 + 11 + 4) \times 1 = 16$$

得到结果就是: SJW = 1

$$\text{BS1} = 11$$

满足 BS1 大于 BS2 的要求

$$\text{BS2} = 4$$

满足 BS2 大于 2 倍 prescaler 的要求

$$\text{Prescaler} = 1$$

```
void CAN_Configuration(void)
{
    //初始化CAN波特率为1M
    CAN_Init(CAN_MasterCtrl_AllDisabled, CAN_Mode_Normal, CAN_SynchroJumpWidth_1TimeQuantum, CAN_BitSeg1_11TimeQuantum,
              CAN_BitSeg2_4TimeQuantum, 1);
```

$$\text{BS2} = 4$$

$$\text{Prescaler} = 1$$

Syn 固定值为 1

$$\text{BS1} = 11$$

这就是固件库函数设置 STM8 CAN 总线的波特率

```
void CAN_Configuration(void)
{
    //初始化CAN波特率为1M
    CAN_Init(CAN_MasterCtrl_AllDisabled, CAN_Mode_Normal, CAN_SynJumpWidth_1TimeQuantum, CAN_BitSeg1_11TimeQuantum,
              CAN_BitSeg2_4TimeQuantum, 1);
    //初始化CAN波特率为800K
    /* CAN_Init(CAN_MasterCtrl_AllDisabled, CAN_Mode_Normal, CAN_SynJumpWidth_1TimeQuantum, CAN_BitSeg1_5TimeQuantum,
               CAN_BitSeg2_4TimeQuantum, 2); */
    //初始化CAN波特率为500K
    /* CAN_Init(CAN_MasterCtrl_AllDisabled, CAN_Mode_Normal, CAN_SynJumpWidth_1TimeQuantum, CAN_BitSeg1_4TimeQuantum,
               CAN_BitSeg2_3TimeQuantum, 4); */
    //初始化CAN波特率为250K
    /* CAN_Init(CAN_MasterCtrl_AllDisabled, CAN_Mode_Normal, CAN_SynJumpWidth_1TimeQuantum, CAN_BitSeg1_4TimeQuantum,
               CAN_BitSeg2_3TimeQuantum, 8); */
    //初始化CAN波特率为125K
    /* CAN_Init(CAN_MasterCtrl_AllDisabled, CAN_Mode_Normal, CAN_SynJumpWidth_1TimeQuantum, CAN_BitSeg1_8TimeQuantum,
               CAN_BitSeg2_7TimeQuantum, 8); */
}
```

我计算了 125K, 250, 500K , 800K ,1M 这几个波特率。