STM32F1 外设操作

作者:向仔州

Stm32 I2C 固件库操作，有 BUG.....1
Stm32f105 双 CAN 同时使用，有 CAN1,CAN2.....5
ADC 使用.....10

STM32 I2C 固件库操作

```
typedef struct
{
   uint32_t I2C_ClockSpeed;  //设置 I2C SCL 时钟频率，此值要低于 400000 比如我要 400K 的速率我就写 400000
   uint16_t I2C_Mode;
   uint16_t I2C_DutyCycle;
   uint16_t I2C_OwnAddress1;
   uint16_t I2C_Ack;
   uint16_t I2C_AcknowledgedAddress;
}I2C_InitTypeDef;
```

指定 STM32 芯片自己的 I2C 地址，STM32 做从机可以用

```
#define I2C_Mode_I2C                ((uint16_t)0x0000)
#define I2C_Mode_SMBusDevice        ((uint16_t)0x0002)
#define I2C_Mode_SMBusHost          ((uint16_t)0x000A)
#define IS_I2C_MODE(MODE)
```

通常都是选择 I2C_Mode_I2C

```
#define I2C_DutyCycle_16_9          ((uint16_t)0x4000)
#define I2C_DutyCycle_2             ((uint16_t)0xBFFF)
#define IS_I2C_DUTY_CYCLE(CYCLE)
```

16_9 或者 DutyCycle_2 都可以选

Ack 在初始化的时候要配置 I2C 为允许应答 I2C_Ack_Enable

```
#define I2C_Ack_Disable             ((uint16_t)0x0000)
#define I2C_Ack_Enable              ((uint16_t)0x0400)
```

根据从机地址决定主机 I2C 是发送 7 位地址还是 10 位地址

```
#define I2C_AcknowledgedAddress_10bit   ((uint16_t)0xC000)
#define I2C_AcknowledgedAddress_7bit    ((uint16_t)0x4000)
```

记住如果我 STM32 做从机的话， I2C_OwmAddress1 变量支持 10 位地址，I2C_OwmAddress2 只支持 7 位模式。

配置完 I2C 模式后下面我们来看看 I2C 读写外设的函数介绍

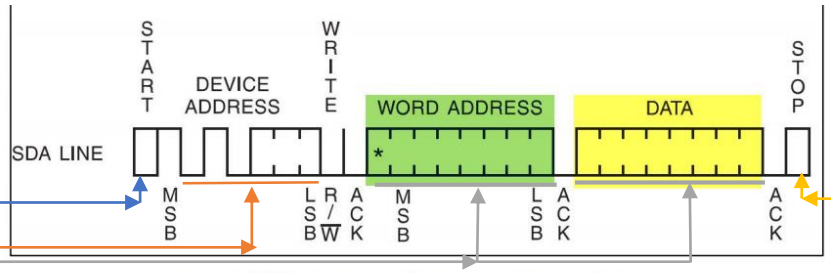


图 24-14 EEPROM 单字节写入时序(摘自《AT24C02》规格书)

I2C_GenerateSTART(I2C_TypeDef* I2Cx, FunctionalState NewState);//I2C 产生 start 开始信号
I2C_TypeDef* I2Cx： 写 I2C1 还是 I2C2(就是 I2C 控制器选择)
FunctionalState NewState: 写 ENABLE 或者 DISABLE

I2C_GenerateSTOP(I2C_TypeDef* I2Cx, FunctionalState NewState);//I2C 主机产生 stop 信号
I2C_TypeDef* I2Cx： 写 I2C1 还是 I2C2(就是 I2C 控制器选择)
FunctionalState NewState: 写 ENABLE 或者 DISABLE

I2C_Send7bitAddress(I2C_TypeDef* I2Cx, uint8_t Address, uint8_t I2C_Direction);
//I2C 主机发送或者接受器件的 I2C 地址
I2C_TypeDef* I2Cx： 写 I2C1 还是 I2C2(就是 I2C 控制器选择)
Address: 写从机 I2C 地址
I2C_Direction: I2C_Direction_Transmitter (STM32 作为发送器)/
I2C_Direction_Receiver (STM32 作为接受器)

I2C_SendData(I2C_TypeDef* I2Cx, uint8_t Data);//I2C 主机发送数据给从机
I2C_TypeDef* I2Cx： 写 I2C1 还是 I2C2(就是 I2C 控制器选择)
Data:就是 8 位数据

uint8_t I2C_ReceiveData(I2C_TypeDef* I2Cx)
I2C_TypeDef* I2Cx： 写 I2C1 还是 I2C2(就是 I2C 控制器选择)
uint8_t: 接受的数据返回值

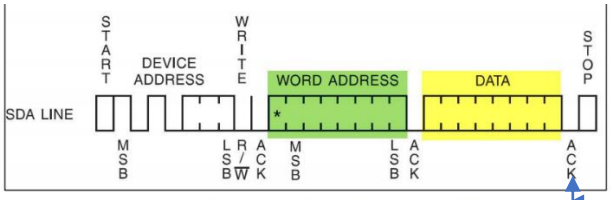


图 24-14 EEPROM 单字节写入时序(摘自《AT24C02》规格书)

I2C_AcknowledgeConfig(I2C_TypeDef* I2Cx, FunctionalState NewState);

//当我们主机觉得接受从机的数据量差不多了，不想接受从机数据了，想结束这次通信，我们就给从机发送个 NACK 信号表示接受完成

I2C_TypeDef* I2Cx： 写 I2C1 还是 I2C2(就是 I2C 控制器选择)

FunctionalState NewState: 写 ENABLE(产生 Nack)或者 DISABLE(不产生 Nack)

有些 I2C 从机器件不需要主机产生 Nack

下面写个实例 STM32 I2C 读取 EEPROM 数据的程序

```
/*I2C1引脚初始化函数*/
void I2C_GPIO_INIT()
{
    GPIO_InitTypeDef GPIO_InitStructure;

    RCC_APB1PeriphClockCmd(RCC_APB1Periph_I2C1,ENABLE);//I2C1在APB1总线打开总线时钟
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB,ENABLE);//打开GPIOB的时钟

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6;/*GPIOB6是I2C1的SCL引脚*/
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_OD; //I2C引脚都是开漏输出
    GPIO_Init(GPIOB, &GPIO_InitStructure);

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_7;/*GPIOB7是I2C1的SDA引脚*/
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_OD; //I2C引脚都是开漏输出
    GPIO_Init(GPIOB, &GPIO_InitStructure);
}
/*I2C工作模式初始化*/
void I2C_mode_config()
{
    I2C_InitTypeDef I2C_InitStructure;

    I2C_InitStructure.I2C_Ack = I2C_Ack_Enable;//I2C应答ACK使能
    I2C_InitStructure.I2C_AcknowledgedAddress = I2C_AcknowledgedAddress_7bit;//I2C主机只发生7位地址
    I2C_InitStructure.I2C_ClockSpeed = 400000;//I2C通信速率400K
    I2C_InitStructure.I2C_DutyCycle = I2C_DutyCycle_2;//选择2:1没有问题
    I2C_InitStructure.I2C_Mode = I2C_Mode_I2C;//主机设置为I2C模式
    I2C_InitStructure.I2C_OwnAddress1 = 0;//STM32做从机自己I2C地址
    I2C_Init(I2C1, &I2C_InitStructure);//把上面配置的I2C参数给I2C1控制器

    I2C_Cmd(I2C1, ENABLE);//配置完后一定不要忘记打开I2C1功能
}
```

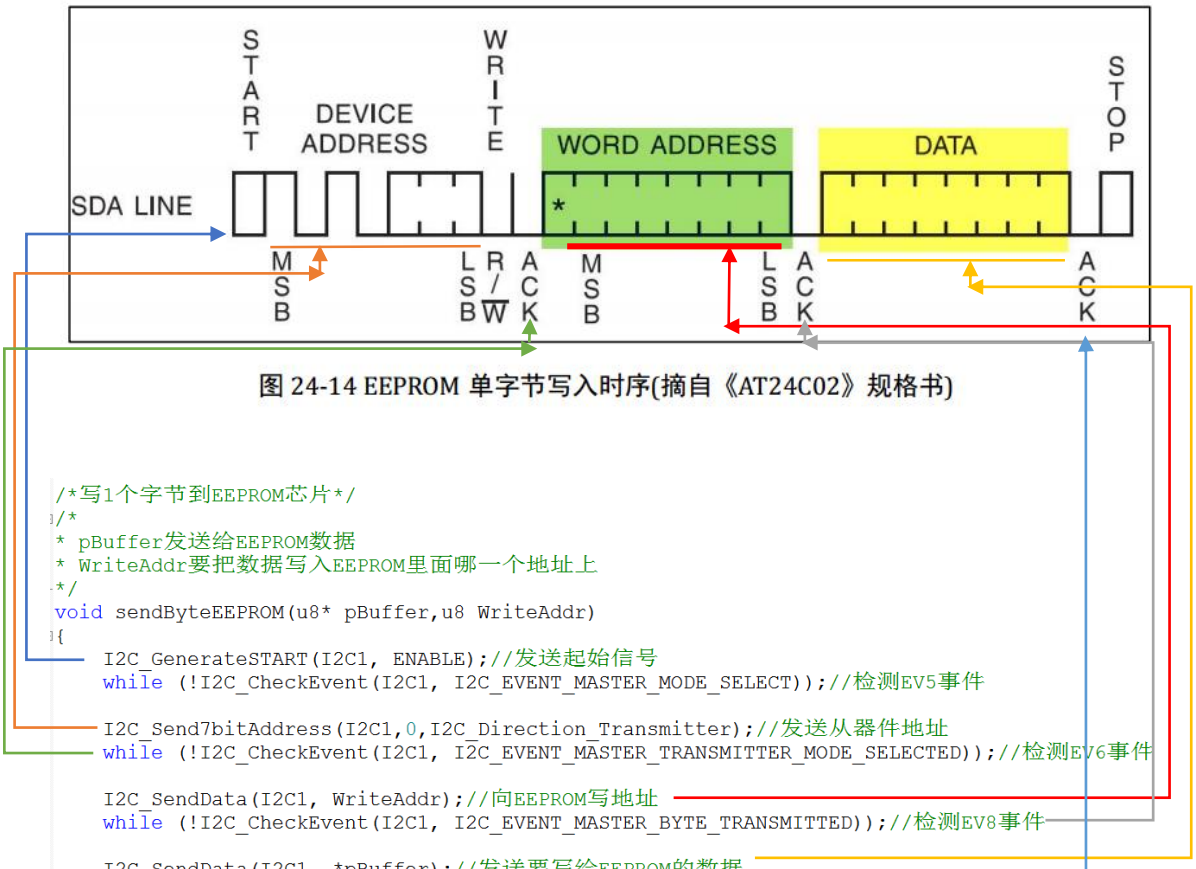


图 24-14 EEPROM 单字节写入时序(摘自《AT24C02》规格书)

```
/*写1个字节到EEPROM芯片*/
/*
* pBuffer发送给EEPROM数据
* WriteAddr要把数据写入EEPROM里面哪一个地址上
*/
void sendByteEEPROM(u8* pBuffer,u8 WriteAddr)
{
    I2C_GenerateSTART(I2C1, ENABLE);//发送起始信号
    while (!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_MODE_SELECT));//检测EV5事件

    I2C_Send7bitAddress(I2C1,0,I2C_Direction_Transmitter);//发送从器件地址
    while (!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_TRANSMITTER_MODE_SELECTED));//检测EV6事件

    I2C_SendData(I2C1, WriteAddr);//向EEPROM写地址
    while (!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_BYTE_TRANSMITTED));//检测EV8事件

    I2C_SendData(I2C1, *pBuffer);//发送要写给EEPROM的数据
    while (!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_BYTE_TRANSMITTED));//检测EV8事件

    I2C_GenerateSTOP(I2C1, ENABLE);//发送停止信号

}
/*读取EEPROM数据*/
//读取的eeprom数据放在pBuffer
//要读取eeprom那个地址上的数据，用ReadAddr指定地址
void I2C_EEPROM_ByteRead(uint8_t *pBuffer, uint8_t ReadAddr)
{
    //I2C_EE_WaitEepromStandbyState();//操作EEPROM的话这里还要执行等待EEPROM准备好的函数
    //操作其它IIC器件应该不需要等待函数

    I2C_GenerateSTART(I2C1, ENABLE);//发送起始信号
    while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_MODE_SELECT));//检测EV5

    I2C_Send7bitAddress(I2C1, 0, I2C_Direction_Transmitter);//器件地址是0x00
    while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_TRANSMITTER_MODE_SELECTED));//检测EV6

    I2C_SendData(I2C1, ReadAddr);//要读取那个地址上的数据，发送读取的地址
    while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_BYTE_TRANSMITTED));//检测EV8

    I2C_GenerateSTART(I2C1, ENABLE);//从新发送start信号
    while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_MODE_SELECT));//检测EV5

    I2C_Send7bitAddress(I2C1, 0, I2C_Direction_Receiver);//器件地址是0x00
    while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_RECEIVER_MODE_SELECTED));//检测EV6

    while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_BYTE_RECEIVED)); //检测EV7

    *pBuffer = I2C_ReceiveData(I2C1);//读取从机地址上的数据

    I2C_AcknowledgeConfig(I2C1, DISABLE);//STM32主机发送非应答(不需要从机应答)信号
    I2C_GenerateSTOP(I2C1, ENABLE);//产生停止信号
}
```

```c
int main(void)
{

    RCC_configuration();//初始化时钟
    USART_config(115200);//初始化串口

    I2C_GPIO_INIT();//初始化IIC管脚
    I2C_mode_config();//初始IIC模式

    //sendByteEEPROM(u8* pBuffer,u8 WriteAddr);//发数据到EEPROM
    // I2C_EEPROM_ByteRead(uint8_t *pBuffer, uint8_t ReadAddr)读取EEPROM数据
    while(1)
    {
        printf("xxxzzzzzz\r\n");

    }
    return 0;

}
```

这就是 STM32 主机操作 EEPROM 的测试程序，但是没有上板子试过，库函数使用方法就是这样。

这里要注意，不只是 I2C1 控制器有问题，I2C2 控制器在有其它程序中断的情况下也可能出问题。所以还是建议用模拟的 I2C 接口

STM32 双 CAN 同时使用，CAN1,CAN2

STM32F103 是没有两个 CAN 的，F103 只有 CAN1。STM32F105 才有 CAN1 和 CAN2。

在移植 STM32F103 的 CAN 程序到 STM32F105 芯片上时要做一些变动。

STM32F103 和 STM32F105 的 CAN 配置有 80%是一模一样，但是还是有 20%不一样。

1 过滤器差异

```c
/*CAN1过滤器配置*/
void CAN1_Filter(uint32_t ID)
{
    CAN_FilterInitTypeDef CAN_FilterTypeStruct;

    CAN_FilterTypeStruct.CAN_FilterActivation = ENABLE;//是否启动过滤器,我这里关闭CAN过滤器
    CAN_FilterTypeStruct.CAN_FilterFIFOAssignment = CAN_Filter_FIFO0;//过滤出需要ID的数据就放在fifo0
    CAN_FilterTypeStruct.CAN_FilterNumber = 0;
    CAN_FilterTypeStruct.CAN_FilterScale = CAN_FilterScale_32bit;//用32位过滤器
    CAN_FilterTypeStruct.CAN_FilterMode = CAN_FilterMode_IdMask;
```

STM32F105 芯片复位后默认使用 0～13 号过滤寄存器，过滤数据到 CAN1 的 FIFO1 或者 FIFO0

但是 CAN2 不能使用 0～13 过滤器

CAN2 只能使用 14～17 号过滤器，过滤数据到 FIFO0 或者 FIFO1

```c
/*CAN2过滤器配置*/
void CAN2_Filter(uint32_t ID)
{
    CAN_FilterInitTypeDef CAN_FilterTypeStruct;

    CAN_FilterTypeStruct.CAN_FilterActivation = ENABLE;//是否启动过滤器,我这里关闭CAN过滤器
    CAN_FilterTypeStruct.CAN_FilterFIFOAssignment = CAN_Filter_FIFO1;//过滤出需要ID的数据就放在fifo1
    CAN_FilterTypeStruct.CAN_FilterNumber = 14;//CAN过滤器通道必须从14通道开始 14~27
    CAN_FilterTypeStruct.CAN_FilterScale = CAN_FilterScale_32bit;//用32位过滤器
    CAN_FilterTypeStruct.CAN_FilterMode = CAN_FilterMode_IdMask;
```

过滤器修改完成之后还要修改 CAN 的中断优先级的入口函数名

```c
/*CAN接受中断配置*/
void CAN_NVIC_Config(void)
{
    NVIC_InitTypeDef NVIC_InitStructure;//定义CAN中断优先级结构体

    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_0);//中断分配到0组
    NVIC_InitStructure.NVIC_IRQChannel = USB_LP_CAN1_RX0_IRQn;//CAN1接受邮箱0中断
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;//抢占优先级为1
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1;//响应优先级(子优先级)为1
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;//中断优先级开启
    NVIC_Init(&NVIC_InitStructure);//初始化中断优先级
```

因为 STM32F103 只有 1 个 CAN1,而且 CAN1 和 USB 是复用的，所以接受数据的中断函数入口名是 USB_LP_CAN1_RX0_IRQn

但是 STM32F105 是双 CAN，所以中断函数名发送了变化

```
/*CAN1接受中断配置*/
void CAN1_NVIC_Config(void)
{
    NVIC_InitTypeDef NVIC_InitStructure;//定义CAN中断优先级结构体

    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_0);//中断分配到0组
    NVIC_InitStructure.NVIC_IRQChannel = CAN1_RX0_IRQn;//CAN1接受邮箱0中断
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;//抢占优先级为1
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1;//响应优先级(子优先级)为1
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;//中断优先级开启
    NVIC_Init(&NVIC_InitStructure);//初始化中断优先级

}
```

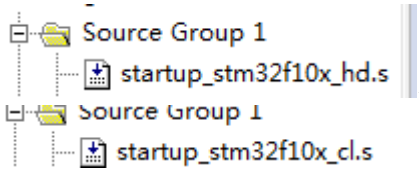
STM32F105CAN1 中断函数名是 CAN1_RX0_IRQn，这个 RX0 是在初始化 CAN1 的时候设置的邮箱 0 来接受数据

```
/*CAN2接受中断配置*/
void CAN2_NVIC_Config(void)
{
    NVIC_InitTypeDef NVIC_InitStructure;//定义CAN中断优先级结构体

    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_0);//中断分配到0组
    NVIC_InitStructure.NVIC_IRQChannel = CAN2_RX1_IRQn;//CAN2接受邮箱1中断
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;//抢占优先级为1
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1;//响应优先级(子优先级)为1
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;//中断优先级开启
    NVIC_Init(&NVIC_InitStructure);//初始化中断优先级

}
```

STM32F105CAN2 中断函数名是 CAN1_RX1_IRQn，这个 RX1 是在初始化 CAN2 的时候设置的邮箱 1 来接受数据

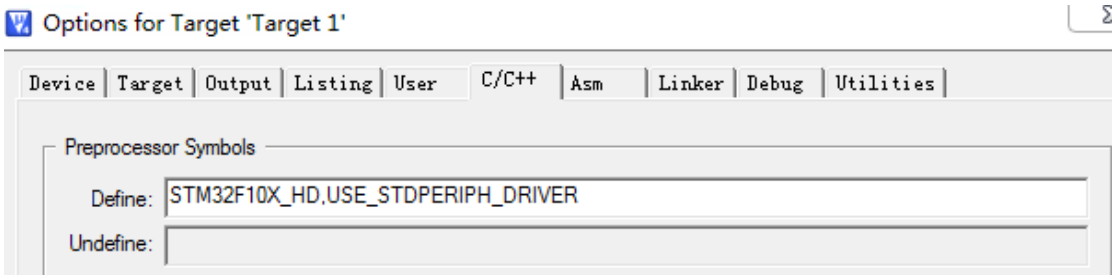
因为修改了中断函数名，导致编译不过!!

因为 CAN1_RX0_IRQn 和 CAN2_RX1_IRQn 是在 stm32f10x.h 文件里面的 STM32F10X_CL 宏下面定义的，但是添加这个宏就要修改启动文件。

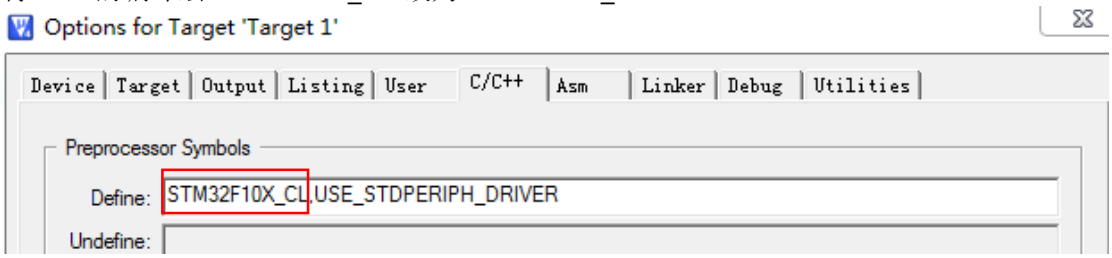


将 f103 的 hd.s 文件改成 cl.s 文件

这个 cl.s 文件在 keil 创建工程的时候就有了，只是我们没有要，现在用 STM32F105 就需要使用了。



将 f103 的编译宏 STM32F10X_HD 改为 STM32F10X_CL



这样就可以成功编译 CAN1_RX0_IRQn 和 CAN2_RX1_IRQn 了

但是编译过程中发现 RCC 初始化报错

这是因为 STM32F105 的 RCC 初始化方式和 F103 不一样

```c
#include "stm32f10x.h"
#include "sysclock.h"
void RCC_configuration(void)
{
//  ErrorStatus HSEStarUpstatus;//定义外部晶振是否正常启动的状态标志
//  RCC_DeInit(); //复位RCC外部晶振
//  RCC_HSEConfig(RCC_HSE_ON);//打开外部HSE高速晶振
//  HSEStarUpstatus = RCC_WaitForHSEStartUp();
//  if(HSEStarUpstatus == SUCCESS)
//  {
//      FLASH_PrefetchBufferCmd(FLASH_PrefetchBuffer_Enable);
//      FLASH_SetLatency(FLASH_Latency_2);
//      RCC_HCLKConfig(RCC_SYSCLK_Div1);
//      RCC_PCLK2Config(RCC_HCLK_Div1);
//      RCC_PCLK1Config(RCC_HCLK_Div2);
//      RCC_PLLConfig(RCC_PLLSource_HSE_Div1,RCC_PLLMul_9);
//      RCC_PLLCmd(ENABLE);
//      while(RCC_GetFlagStatus(RCC_FLAG_PLLRDY)!=SET);
//      RCC_SYSCLKConfig(RCC_SYSCLKSource_PLLCLK);
//      if(RCC_GetSYSCLKSource()!=0x08);
//  }

    RCC_ClocksTypeDef RCC_ClockFreq;
    /* RCC system reset(for debug purpose) */
    RCC_DeInit();
    /* Enable HSE */
    RCC_HSEConfig(RCC_HSE_ON);
    /* Wait till HSE is ready */
    if(RCC_WaitForHSEStartUp() != ERROR)
    {
        /* Enable Prefetch Buffer */
        FLASH_PrefetchBufferCmd(FLASH_PrefetchBuffer_Enable);

  /*********************************************************************/
  /*      HSE=8MHz, HCLK=72MHz, PCLK2=72MHz, PCLK1=36MHz               */
  /*********************************************************************/
        /* Flash 2 wait state */
        FLASH_SetLatency(FLASH_Latency_2);
        /* HCLK = SYSCLK */
        RCC_HCLKConfig(RCC_SYSCLK_Div1);
        /* PCLK2 = HCLK */
        RCC_PCLK2Config(RCC_HCLK_Div1);
        /* PCLK1 = HCLK/2 */
        RCC_PCLK1Config(RCC_HCLK_Div2);
       RCC_ADCCLKConfig(RCC_PCLK2_Div6);

       /* Configure PLLs ********************************************
       /* PPL2 configuration: PLL2CLK = (HSE / 2) * 10 = 40 MHz */
       RCC_PREDIV2Config(RCC_PREDIV2_Div2);
       RCC_PLL2Config(RCC_PLL2Mul_10);

       /* Enable PLL2 */
       RCC_PLL2Cmd(ENABLE);

       /* Wait till PLL2 is ready */
       while (RCC_GetFlagStatus(RCC_FLAG_PLL2RDY) == RESET)
       {}

       /* PPL1 configuration: PLLCLK = (HSE / 1) * 9 = 72 MHz */
       RCC_PREDIV1Config(RCC_PREDIV1_Source_HSE, RCC_PREDIV1_Div1);
       RCC_PLLConfig(RCC_PLLSource_PREDIV1, RCC_PLLMul_9);

       /* Enable PLL */
       RCC_PLLCmd(ENABLE);

       /* Wait till PLL is ready */
       while (RCC_GetFlagStatus(RCC_FLAG_PLLRDY) == RESET)
       {}

       /* Select PLL as system clock source */
       RCC_SYSCLKConfig(RCC_SYSCLKSource_PLLCLK);

         /* Wait till PLL is used as system clock source */
         while (RCC_GetSYSCLKSource() != 0x08)
         {}
    }
}
```

取消掉 F103 的 RCC 初始化程序

修改为 F105 的 RCC 初始化程序

下面是 RCC 初始化代码可以复制片段

```
RCC_ClocksTypeDef RCC_ClockFreq;
/* RCC system reset(for debug purpose) */
RCC_DeInit();
/* Enable HSE */
RCC_HSEConfig(RCC_HSE_ON);
/* Wait till HSE is ready */
if(RCC_WaitForHSEStartUp() != ERROR)
{
      /* Enable Prefetch Buffer */
      FLASH_PrefetchBufferCmd(FLASH_PrefetchBuffer_Enable);
```

```
/*******************************************************************/
/*          HSE=8MHz, HCLK=72MHz, PCLK2=72MHz, PCLK1=36MHz          */
/*******************************************************************/
        /* Flash 2 wait state */
        FLASH_SetLatency(FLASH_Latency_2);
        /* HCLK = SYSCLK */
        RCC_HCLKConfig(RCC_SYSCLK_Div1);
        /* PCLK2 = HCLK */
        RCC_PCLK2Config(RCC_HCLK_Div1);
        /* PCLK1 = HCLK/2 */
        RCC_PCLK1Config(RCC_HCLK_Div2);
        /*    ADCCLK = PCLK2/4 */
        RCC_ADCCLKConfig(RCC_PCLK2_Div6);

        /*****ConfigurePLLs ******/
        /* PPL2 configuration: PLL2CLK = (HSE / 2) * 10 = 40 MHz */
        RCC_PREDIV2Config(RCC_PREDIV2_Div2);
        RCC_PLL2Config(RCC_PLL2Mul_10);

        /* Enable PLL2 */
        RCC_PLL2Cmd(ENABLE);

        /* Wait till PLL2 is ready */
        while (RCC_GetFlagStatus(RCC_FLAG_PLL2RDY) == RESET)
        {}

        /* PPL1 configuration: PLLCLK = (HSE / 1) * 9 = 72 MHz */
        RCC_PREDIV1Config(RCC_PREDIV1_Source_HSE, RCC_PREDIV1_Div1);
        RCC_PLLConfig(RCC_PLLSource_PREDIV1, RCC_PLLMul_9);

        /* Enable PLL */
        RCC_PLLCmd(ENABLE);

        /* Wait till PLL is ready */
        while (RCC_GetFlagStatus(RCC_FLAG_PLLRDY) == RESET)
        {}

        /* Select PLL as system clock source */
        RCC_SYSCLKConfig(RCC_SYSCLKSource_PLLCLK);

        /* Wait till PLL is used as system clock source */
        while (RCC_GetSYSCLKSource() != 0x08)
        {}
    }
```

编译完全通过，但是串口输出数据不正常，CAN 时钟也没有反应，这是因为外部时钟没有设置对。

打开 STM32F10x.h 头文件去修改 HSE_VALUE 宏

```
80  */
81 #if !defined  HSE_Value
82  #ifdef STM32F10X_CL
83   #define HSE_Value    ((uint32_t)25000000) /*!< Value of the External oscillator in Hz */
84  #else
85   #define HSE_Value    ((uint32_t)8000000) /*!< Value of the External oscillator in Hz */
86  #endif /* STM32F10X_CL */
87 #endif /* HSE_Value */
```

改成8000000

官方为了兼容 STM32F105 USBOTG 和 STM32F107 互联网型的网络功能，要求外部晶振必须配 25M。

```
115 #if !defined  HSE_VALUE
116  #ifdef STM32F10X_CL
117   #define HSE_VALUE    ((uint32_t)8000000) /*!< Va
118  #else
119   #define HSE_VALUE    ((uint32_t)8000000) /*!< Va
120  #endif /* STM32F10X_CL */
121 #endif /* HSE_VALUE */
```

我们用的是外部 8M 晶振，所以这里改成 8 就是了，这个宏会自动拿去给 RCC 初始化运算。

```
 8 CanTxMsg CAN1_Tx_data;//定义CAN1发送数据包
 9 CanRxMsg CAN1_Rx_data;//定义CAN1接受数据包
10
11 CanTxMsg CAN2_Tx_data;//定义CAN1发送数据包
12 CanRxMsg CAN2_Rx_data;//定义CAN1接受数据包
13
```

定义 CAN1 和 CAN2 的发送数据变量和接受数据变量

```
32 /*CAN1 接受邮箱0的中断服务函数*/
33 //void USB_LP_CAN1_RX0_IRQHandler(void)
34 void CAN1_RX0_IRQHandler(void)
35 {
36     CAN_Receive(CAN1,CAN_FIFO0,&CAN1_Rx_data);//这
37 //#ifdef CAN1DEBUG
38     printf("Receive CAN ID = %x data0 = %x data1 =
39  data5 = %x data6 = %x data7 = %x \r\n",CAN1_Rx_da
40     CAN1_Rx_data.Data[2],CAN1_Rx_data.Data[3],CAN1
41     CAN1_Rx_data.Data[7]);
42 //#endif
43 }
44
45 void CAN2_RX1_IRQHandler(void)
46 {
47     printf("CAN2.....\r\n");
48 }
```

实现 CAN1 和 CAN2 接受数据中断

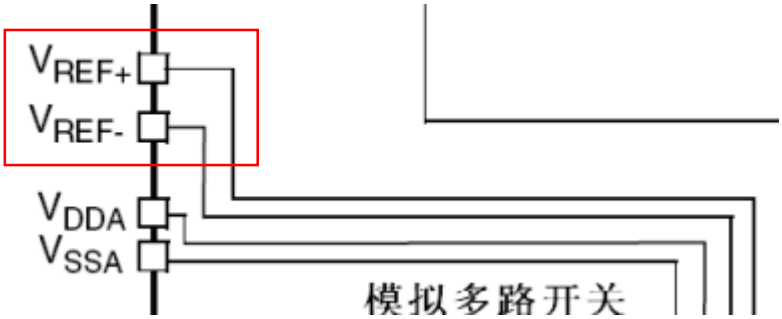
这里会无限打印 CAN2... 因为没有执行 CAN_Receive 清楚 CAN 总线中断标志位

```
44
45 void CAN2_RX1_IRQHandler(void)
46 {
47     CAN_Receive(CAN2,CAN_FIFO1,&CAN2_Rx_data);//这个CAN_Receive函数会
48     printf("CAN2.....\r\n");
49 }
50
```

这些 CAN1 和 CAN2 就驱动起来了

ADC 使用

ADC 电源供电部分

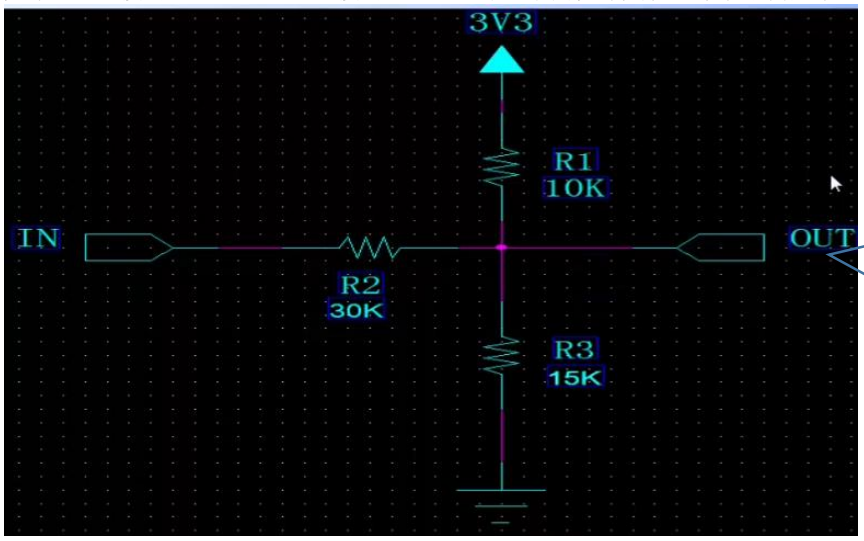


ADC输入范围：$V_{REF-} \leqslant V_{IN} \leqslant V_{REF+}$ 意思就是 VREF-是 ADC 采集的最小电压，VREF+是 ADC 采集的最大电压,所以这个 VREF 电压是自己设置的。

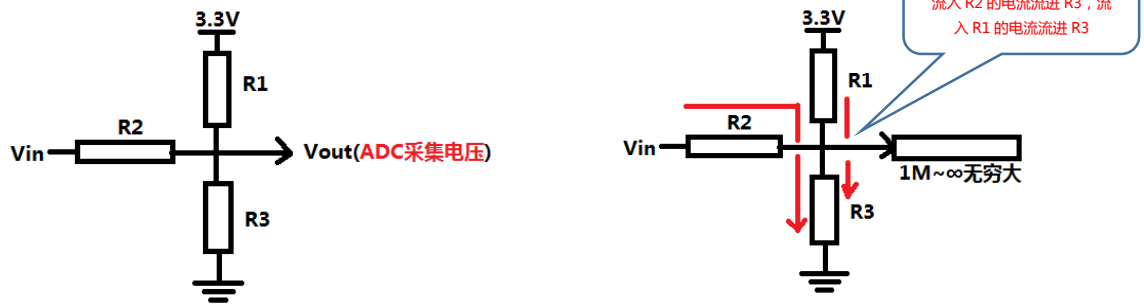
如果 VREF+接 3.3V，VREF-接 0V，那么 ADC 电压采集范围就是 0～3.3V

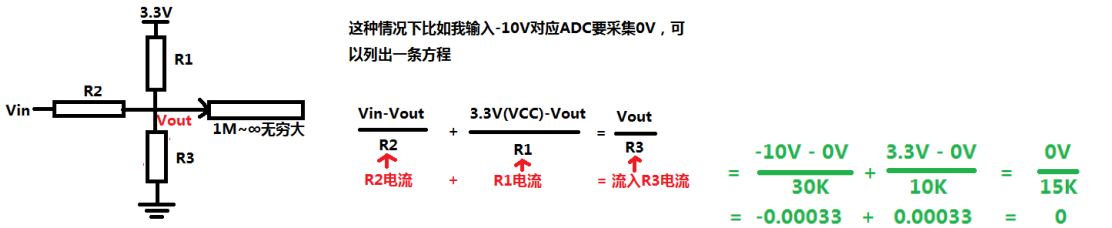
如果我要采集-10V～+10V 的电压怎么办呢？在 ADC VREF+参考电压接在 3.3V，VREF-电压接在 0V 情况下。

如果 VREF+接的是+10V，VREF—接的是-10V，那么下面信号变换电路就不需要了



这种电路该怎么计算?





如果是+10V 输入，也就是 Vin 设置为+10V, Vout 设置为 3.3V，代入上面公式去算

(Vin-Vout)/R2 + (VCC(3.3V)-Vout)/R1=Vout/15k = (10-3.3)/30K+(3.3-3.3)/10K=3.3/15K 正好两边电流相等。

ADC 固件库使用

```
typedef struct
{
    uint32_t ADC_Mode;                      // ADC 工作模式选择
    FunctionalState ADC_ScanConvMode;       /* ADC 扫描（多通道）
                                               或者单次（单通道）模式选择 */
    FunctionalState ADC_ContinuousConvMode; // ADC 单次转换或者连续转换选择
    uint32_t ADC_ExternalTrigConv;          // ADC 转换触发信号选择
    uint32_t ADC_DataAlign;                 // ADC 数据寄存器对齐格式
    uint8_t ADC_NbrOfChannel;               // ADC 采集通道数
} ADC_InitTypeDef;
```

ADC_Mode: 一般选择 ADC_Mode_Independent 独立模式，这个模式只针对一个 ADC 外设，其余的参数是双 ADC 下选择的。

```
#define ADC_Mode_AlterTrig ((uint32_t)0x00090000)
#define ADC_Mode_FastInterl ((uint32_t)0x00070000)
#define ADC_Mode_Independent ((uint32_t)0x00000000)
#define ADC_Mode_InjecSimult ((uint32_t)0x00050000)
#define ADC_Mode_InjecSimult_FastInterl ((uint32_t)0x00030000)
#define ADC_Mode_InjecSimult_SlowInterl ((uint32_t)0x00040000)
#define ADC_Mode_RegInjecSimult ((uint32_t)0x00010000)
#define ADC_Mode_RegSimult ((uint32_t)0x00060000)
#define ADC_Mode_RegSimult_AlterTrig ((uint32_t)0x00020000)
#define ADC_Mode_SlowInterl ((uint32_t)0x00080000)
#define IS_ADC_MODE(MODE)
```

ADC_ScanConvMode:选择 ENABLE，那么 ADC1 采集多个引脚输入的模拟量数据，比如 ADC1 转换完 1 脚的模拟量会自动去转换 2 脚模拟量，以此类推….。如果选择 DISABLE 那么 ADC1 只采集一个指定引脚的模拟量。

ADC_ContinuousConvMode:选择 DISABLE 那么该 ADC 采集一次通道中的数据就不再采集了，如果选择 ENABLE，那么 ADC 会不停的重复采集通道中的数据。我选择 ENABLE。

ADC_ExternalTrigConv:ADC 通道触发源选择

```
#define ADC_ExternalTrigConv_Ext_IT11_TIM8_TRGO ((uint32_t)0x000C0000)
#define ADC_ExternalTrigConv_None ((uint32_t)0x000E0000)
#define ADC_ExternalTrigConv_T1_CC1 ((uint32_t)0x00000000)
#define ADC_ExternalTrigConv_T1_CC2 ((uint32_t)0x00020000)
#define ADC_ExternalTrigConv_T1_CC3 ((uint32_t)0x00040000)
#define ADC_ExternalTrigConv_T2_CC2 ((uint32_t)0x00060000)
#define ADC_ExternalTrigConv_T2_CC3 ((uint32_t)0x00020000)
#define ADC_ExternalTrigConv_T3_CC1 ((uint32_t)0x00000000)
#define ADC_ExternalTrigConv_T3_TRGO ((uint32_t)0x00080000)
#define ADC_ExternalTrigConv_T4_CC4 ((uint32_t)0x000A0000)
#define ADC_ExternalTrigConv_T5_CC1 ((uint32_t)0x000A0000)
#define ADC_ExternalTrigConv_T5_CC3 ((uint32_t)0x000C0000)
#define ADC_ExternalTrigConv_T8_CC1 ((uint32_t)0x00060000)
#define ADC_ExternalTrigConv_T8_TRGO ((uint32_t)0x00080000)
#define IS_ADC_EXT_TRIG(REGTRIG)
```

ADC_DataAlign:数据对齐选择，我们一般选择 ADC_DataAlign_Right 右对齐

```
#define ADC_DataAlign_Left ((uint32_t)0x00000800)
#define ADC_DataAlign_Right ((uint32_t)0x00000000)
#define IS_ADC_DATA_ALIGN(ALIGN)
```

左对齐,右对齐什么意思？

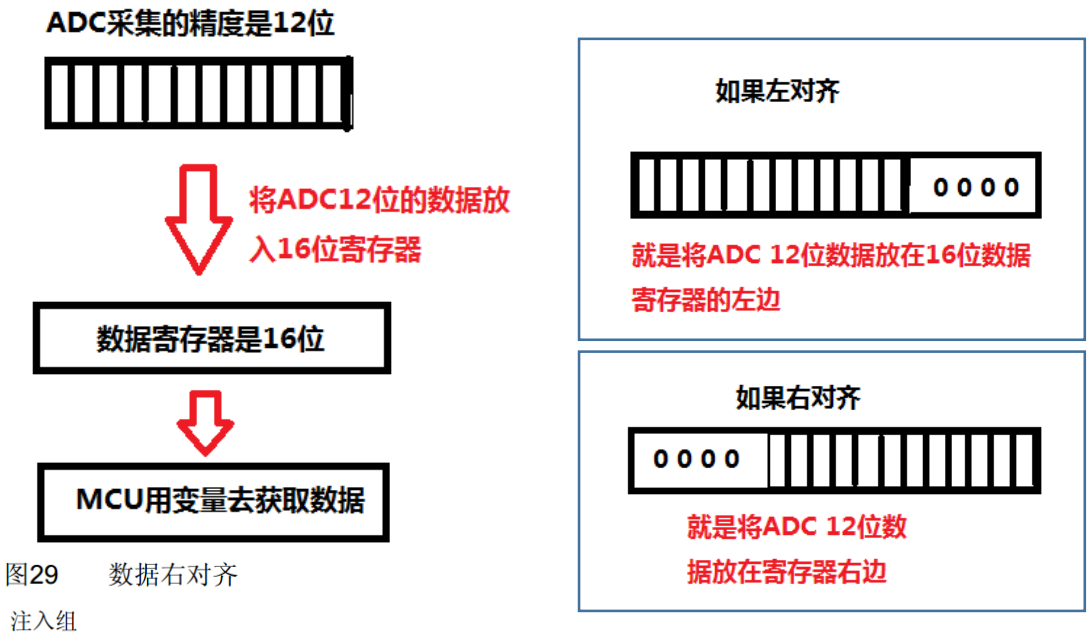


图29　　数据右对齐

注入组

SEXT	SEXT	SEXT	SEXT	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0

规则组

0	0	0	0	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0

图30　　数据左对齐

注入组

SEXT	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0	0	0	0

规则组

D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0	0	0	0	0

这个左对齐或者右对齐决定了你 MCU 变量读取出来的数据是否和 ADC 采集的数据一致。右对齐的数据形式，比如我采集到了 0x00ff 数字量。放在数据寄存器的形式如下:

右对齐

我采集到的数字量就是一一对应的赋值给我变量

0x00ff　0000 0000 1111 1111

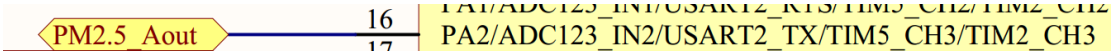
左对齐

我采集的数字量必须要右移4位才是正确的数字量。才能赋值给变量

0x00ff　0000 1111 1111 0000

ADC_NbrOfChannel:这是选择你要 ADC 转换几个通道，我 MCU 外部有 16 个引脚支持 ADC 转换，所以最大支持 16 个通道。

ADC 单通道中断读取数据



采集 PM2.5 的模拟量

```
/*初始化ADC管脚PA2*/
static void ADC_GPIO_init()
{
    GPIO_InitTypeDef GPIO_InitStructure; //设置GPIO复用ADC功能

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA,ENABLE);//我们ADC引脚是PA2,所以打开APB2时钟

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_2;//选择PA2  ADCIN2引脚
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AIN;//选择PA2引脚为模拟输入,因为使用PA2为ADC采集

    GPIO_Init(GPIOA, &GPIO_InitStructure);
}
static void ADC_Mode_Config()
{
    ADC_InitTypeDef ADC_InitStructure;
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1,ENABLE);//我们使用ADC1,打开ADC1时钟

    ADC_InitStructure.ADC_Mode = ADC_Mode_Independent;//设置ADC为独立模式
    ADC_InitStructure.ADC_ScanConvMode = DISABLE ;//关闭扫描模式，因为我们是单通道ADC采集,不是多通道ADC
    ADC_InitStructure.ADC_ContinuousConvMode = ENABLE;//连续不停的转换

    ADC_InitStructure.ADC_ExternalTrigConv = ADC_ExternalTrigConv_None;
    //取消外部触发,(外部触发是定时器或者GPIO),我们用代码去触发ADC转换

    ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;//右对齐

    ADC_InitStructure.ADC_NbrOfChannel = 1;
    //我们只用了PA2通道ADC,所以选择1,这个ADC_NbrOfChannel指的是转换通道数量,而不是1个引脚1个通道

    ADC_Init(ADC1, &ADC_InitStructure);//使用ADC1外设

    RCC_ADCCLKConfig(RCC_PCLK2_Div8);//配置ADC时钟72M/8=9M

    ADC_RegularChannelConfig(ADC1,ADC_Channel_2,1,ADC_SampleTime_55Cycles5);//指定规则通道

    ADC_ITConfig(ADC1, ADC_IT_EOC, ENABLE);//ADC每次转换结束产生中断
    ADC_Cmd(ADC1, ENABLE);//开启ADC

    ADC_ResetCalibration(ADC1);//初始化校准ADC
    while (ADC_GetResetCalibrationStatus(ADC1));//等待初始化ADC1校准完成

    ADC_StartCalibration(ADC1);//开始正式校准ADC
    while (ADC_GetCalibrationStatus(ADC1));

    ADC_SoftwareStartConvCmd(ADC1, ENABLE);//打开软件触发ADC
}
```

STM32 有 ADC1 和 ADC2 两个外设，但是都是采集同样的模拟引脚，我们选择 ADC1

设置 PA2 GPIO 时钟和 IO 功能为模拟输入

因为我们是 PA2 采集数据就一个 ADC 通道，所以设置为独立模式

配置 ADC 时钟为 9M,STM32 的 ADC 时钟最大不能超过 14M

ADC 时钟我设置的 9M，Tconv 转换最快时间 = 1.5 周期 + 12.5 周期 = 14 周期 = 1us。我这里设置 55 个周期，那么就是 55+12.5=67.5=7.5us

我们用的 PA2 引脚，就是 ADC_IN2，所以这里必须要写 Channel1_2

这里默认写 1 因为是单通道模式，如果是多通道，那么就要在这里设置规

```
/*ADC中断优先级初始化*/
static void ADC_NVIC_Config(void)
{
    NVIC_InitTypeDef NVIC_InitStructure;

    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);
    NVIC_InitStructure.NVIC_IRQChannel = ADC1_2_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);
}
```

```
void ADC_init_Cfg(void)
{
    ADC_NVIC_Config();
    ADC_GPIO_init();
    ADC_Mode_Config();
}
```

初始化 ADC 引脚，ADC 功能和 ADC 中断优先级的顺序

```c
void ADC1_2_IRQHandler(void)
{
    if (ADC_GetITStatus(ADC1,ADC_IT_EOC)==SET)
    {
        ADC_ConvertedValue = ADC_GetConversionValue(ADC1);//读取ADC转换值
    }
    ADC_ClearITPendingBit(ADC1,ADC_IT_EOC);//清楚ADC1中断标志位

}
```

ADC 每次转换结束的中断函数，记住中断函数名不要写错了，否则会开机卡死

```c
uint16_t ADC_ConvertedValue=0;

int main(void)
{
    RCC_configuration();//初始化时钟
    USART_config(115200);//初始化串口

    ADC_init_Cfg();//初始化启动ADC

    printf("xxxzzzzzz\r\n");
    while(1)
    {
        printf("ADC value = %d\r\n",ADC_ConvertedValue);
        delay_ms(500);

    }
    return 0;
}
```

```
ADC value = 21
xxxzzzzzz
ADC value = 1136
ADC value = 21
ADC value = 21
ADC value = 19
ADC value = 21
ADC value = 20
ADC value = 19
ADC value = 21
ADC value = 19
ADC value = 19
ADC value = 20
ADC value = 22
ADC value = 148
ADC value = 182
ADC value = 23
ADC value = 22
```

运行一切正常