

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
ADC 固件库使用.....	11
ADC 单通道中断读取数据.....	13
ADC 单通道使用 DMA 读取.....	15
ADC 多通道采集, DMA 读取数据.....	18
ADC1 和 ADC2 两个 ADC 控制器同时采集多通道.....	24
STM32 触摸屏操作.....	28
触摸屏校准算法.....	34
STM32 CMOS 摄像头驱动(未完成).....	42
STM32F1 电源低功耗模式.....	46
1.睡眠模式操作.....	47
2.停止模式操作.....	49
3.待机模式.....	50
STM32RTC 实时时钟操作, 及 STM32 待机唤醒.....	52
2.下面来设置 RTC 闹钟功能, 主要用来唤醒待机模式的 STM32.....	53
RTC 时间日期设置.....	57
高级定时器实现 PWM 互补输出.....	60
输入捕获比较器使用.....	67
输入捕获, 采集 PWM 信号, 测量频率, 占空比.....	74
WAV 文件解析 STM32 实现第 2 种方法.....	81
STM32F103 I2S3 输出音频数据波形实现.....	83
WAV 文件播放实现, 结合 wav 文件解析第 2 种方法文档和 STM32F103 I2S3 输出音频数据波形实现文档, 完成 wav 播放功能.....	84
用定时器解决音乐播放过慢问题.....	90
DMA 双缓冲实现音频播放(未完成).....	92

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_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)0x00A)
#define **IS_I2C_MODE(MODE)** (MODE & (I2C_Mode_I2C | I2C_Mode_SMBusDevice | I2C_Mode_SMBusHost))
通常都是选择 I2C_Mode_I2C

#define **I2C_DutyCycle_16_9** ((uint16_t)0x4000)
#define **I2C_DutyCycle_2** ((uint16_t)0xFFFF)
#define **IS_I2C_DUTY_CYCLE(CYCLE)** ((CYCLE & (I2C_DutyCycle_16_9 | I2C_DutyCycle_2)) == CYCLE)
16_9 或者 DutyCycle_2 都可以选择

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_OwnAddress1 变量支持 10 位地址， I2C_OwnAddress2 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

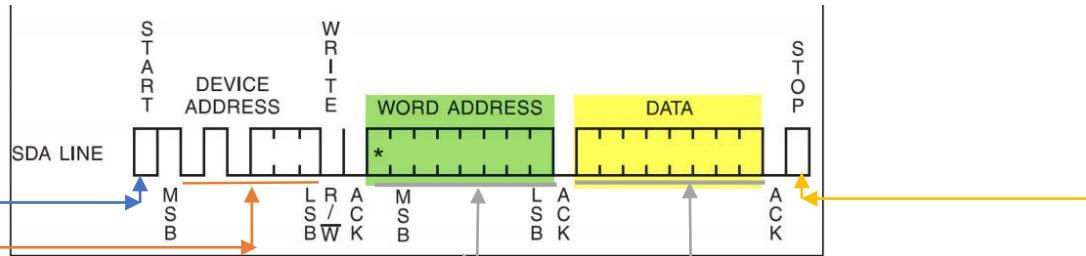


图 24-14 EEPROM 单字节写入时序(摘自《AT24C02》规格书)

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

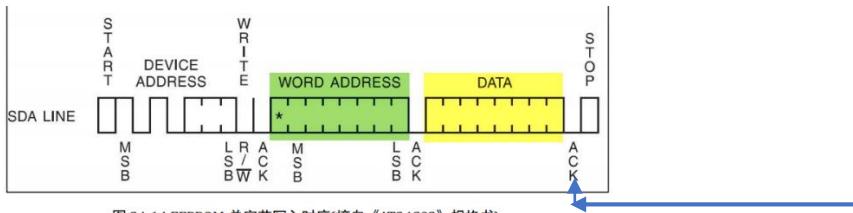


图 24-14 EEPROM 单字节写入时序(摘自《AT24C02》规格书)

```
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，我们就给从机发送一个 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
}
```

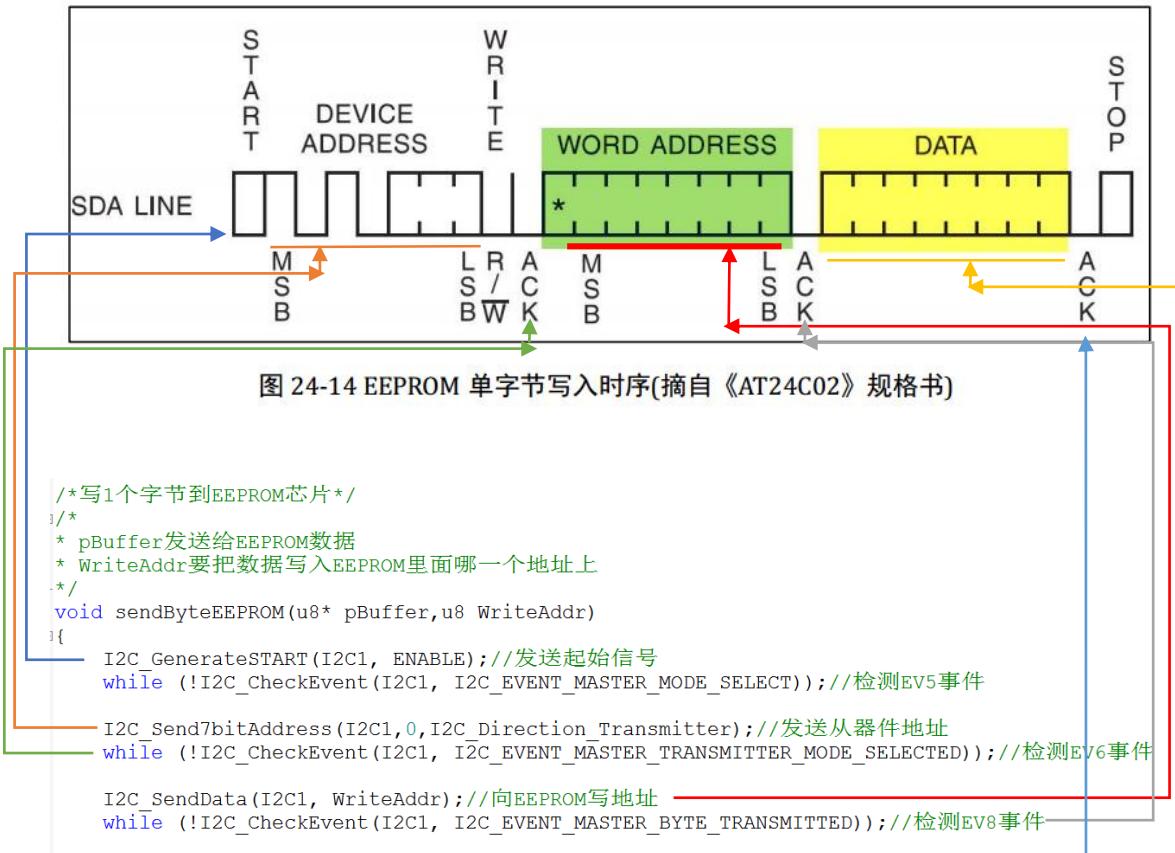


图 24-14 EEPROM 单字节写入时序(摘自《AT24C02》规格书)

```

/*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eprom数据放在pBuffer
//要读取eprom那个地址上的数据，用ReadAddr指定地址
void I2C_EEPROM_BytRead(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
}

```

```

int main(void)
{
    RCC_Configuration(); // 初始化时钟
    USART_Config(115200); // 初始化串口

    I2C_GPIO_Init(); // 初始化I2C管脚
    I2C_Mode_Config(); // 初始化I2C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("xxxxxxxx\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

```

/*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

```

/*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

```

/*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

```
71 /*CAN1接受中断配置*/
72 void CAN1_NVIC_Config(void)
73 {
74     NVIC_InitTypeDef NVIC_InitStructure; // 定义CAN中断优先级结构体
75
76     NVIC_PriorityGroupConfig(NVIC_PriorityGroup_0); // 中断分配到0组
77     NVIC_InitStructure.NVIC_IRQChannel = CAN1_RX0_IRQn; // CAN1接受邮箱0中断
78     NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1; // 抢占优先级为1
79     NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1; // 响应优先级(子优先级)为1
80     NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE; // 中断优先级开启
81     NVIC_Init(&NVIC_InitStructure); // 初始化中断优先级
82 }
83 }

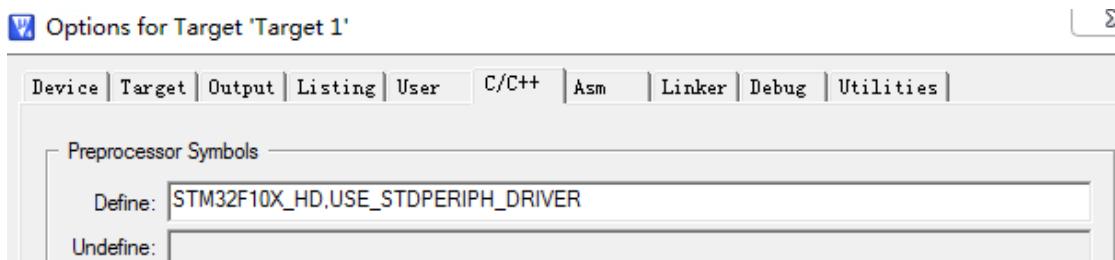
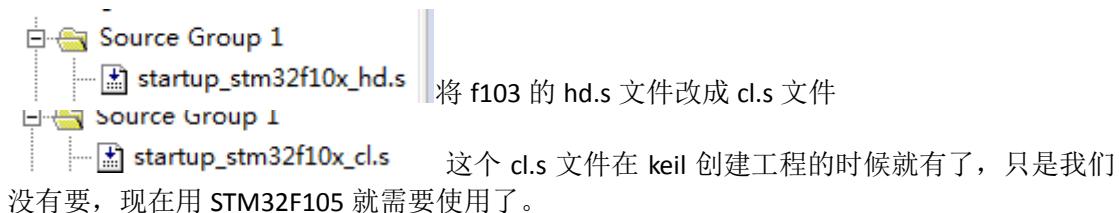
/*CAN2接受中断配置*/
void CAN2_NVIC_Config(void)
{
    NVIC_InitTypeDef NVIC_InitStructure; // 定义CAN中断优先级结构体
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_0); // 中断分配到0组
    NVIC_InitStructure.NVIC_IRQChannel = CAN2_RX1_IRQn; // CAN2接受邮箱1中断
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1; // 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
}
```

STM32F105CAN1 中断函数名是 CAN1_RX0_IRQHandler，这个 RX0 是在初始化 CAN1 的时候设置的邮箱 0 来接受数据

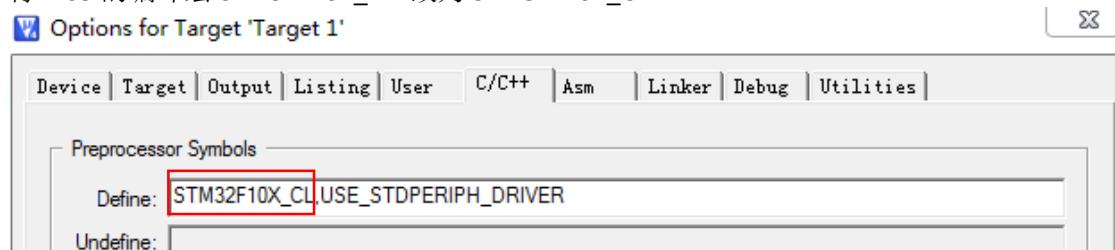
STM32F105CAN2 中断函数名是 CAN1_RX1_IRQHandler，这个 RX1 是在初始化 CAN2 的时候设置的邮箱 1 来接受数据

因为修改了中断函数名，导致编译不过！！

因为 CAN1_RX0_IRQHandler 和 CAN2_RX1_IRQHandler 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 的编译宏 STM32F10X_HD 改为 STM32F10X_CL



这样就可以成功编译 CAN1_RX0_IRQHandler 和 CAN2_RX1_IRQHandler 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

```

1 #include "stm32f10x.h"
2 #include "sysclock.h"
3 void RCC_Configuration(void)
4{
5 // ErrorStatus HSEStarUpstatus;//定义外部晶振是否正常启动的状态标志
6 // RCC_DeInit(); //复位RCC外部晶振
7 // RCC_HSEConfig(RCC_HSE_ON); //打开外部HSE高速晶振
8 // HSEStarUpstatus = RCC_WaitForHSEStartUp();
9 // if(HSEStarUpstatus == SUCCESS)
10 // {
11 //     FLASH_PrefetchBufferCmd(FLASH_PrefetchBuffer_Enable);
12 //     FLASH_SetLatency(FLASH_Latency_2);
13 //     RCC_HCLKConfig(RCC_SYSCLK_Div1);
14 //     RCC_PCLK2Config(RCC_HCLK_Div1);
15 //     RCC_PCLK1Config(RCC_HCLK_Div2);
16 //     RCC_PLLConfig(RCC_PLLSource_HSE_Div1,RCC_PLLMul_9);
17 //     RCC_PLLCmd(ENABLE);
18 //     while(RCC_GetFlagStatus(RCC_FLAG_PLLRDY)!=SET);
19 //     RCC_SYSCLKConfig(RCC_SYSCLKSource_PLLCLK);
20 //     if(RCC_GetSYSCLKSource() !=0x08);
21 // }
22
23 RCC_ClocksTypeDef RCC_ClockFreq;
24 /* RCC system reset(for debug purpose) */
25 RCC_DeInit();
26 /* Enable HSE */
27 RCC_HSEConfig(RCC_HSE_ON);
28 /* Wait till HSE is ready */
29 if(RCC_WaitForHSEStartUp() != ERROR)
30 {
31     /* Enable Prefetch Buffer */
32     FLASH_PrefetchBufferCmd(FLASH_PrefetchBuffer_Enable);
33
34 /*****
35 *      HSE=8MHz, HCLK=72MHz, PCLK2=72MHz, PCLK1=36MHz
36 *****/
37
38     /* Flash 2 wait state */
39     FLASH_SetLatency(FLASH_Latency_2);
40     /* HCLK = SYSCLK */
41     RCC_HCLKConfig(RCC_SYSCLK_Div1);
42     /* PCLK2 = HCLK */
43     RCC_PCLK2Config(RCC_HCLK_Div1);
44     /* PCLK1 = HCLK/2 */
45     RCC_PCLK1Config(RCC_HCLK_Div2);
46     RCC_ADCCLKConfig(RCC_PCLK2_Div6);
47
48     /* Configure PLLs */
49     /* PFL2 configuration: PLL2CLK = (HSE / 2) * 10 = 40 MHz */
50     RCC_PREDIV2Config(RCC_PREDIV2_Div2);
51     RCC_PLL2Config(RCC_PLL2Mul_10);
52
53     /* Enable PLL2 */
54     RCC_PLL2Cmd(ENABLE);
55
56     /* Wait till PLL2 is ready */
57     while (RCC_GetFlagStatus(RCC_FLAG_PLL2RDY) == RESET)
58     {}
59
60     /* PPL1 configuration: PLLCLK = (HSE / 1) * 9 = 72 MHz */
61     RCC_PREDIV1Config(RCC_PREDIV1_Source_HSE, RCC_PREDIV1_Div1);
62     RCC_PLLConfig(RCC_PLLSource_PREDIV1, RCC_PLLMul_9);
63
64     /* Enable PLL */
65     RCC_PLLCmd(ENABLE);
66
67     /* Wait till PLL is ready */
68     while (RCC_GetFlagStatus(RCC_FLAG_PLLRDY) == RESET)
69     {}
70
71     /* Select PLL as system clock source */
72     RCC_SYSCLKConfig(RCC_SYSCLKSource_PLLCLK);
73
74     /* Wait till PLL is used as system clock source */
75     while (RCC_GetSYSCLKSource() != 0x08)
76     {}
77
78 }
79

```

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 */

```

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) /*!< Value of HSE */
118 #else
119     #define HSE_VALUE ((uint32_t)8000000) /*!< Value of HSE */
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 CAN2 发送数据包
12 CanRxMsg CAN2_Rx_data; // 定义 CAN2 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
37 // #ifndef 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 }

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

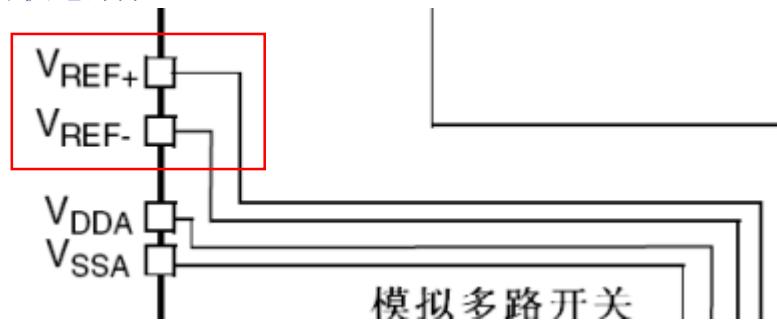
```

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

ADC 使用

ADC 电源供电部分

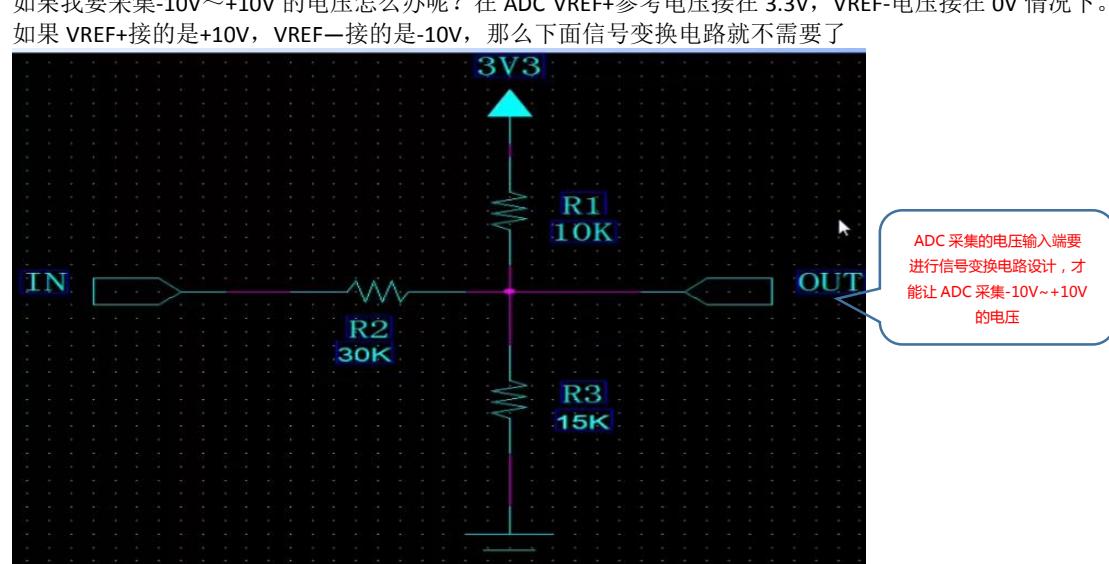


ADC输入范围: $V_{REF-} \leq V_{IN} \leq V_{REF+}$ 意思就是 VREF-是 ADC 采集的最小电压, VREF+是 ADC 采集的最大电压,所以这个 VREF 电压是自己设置的。

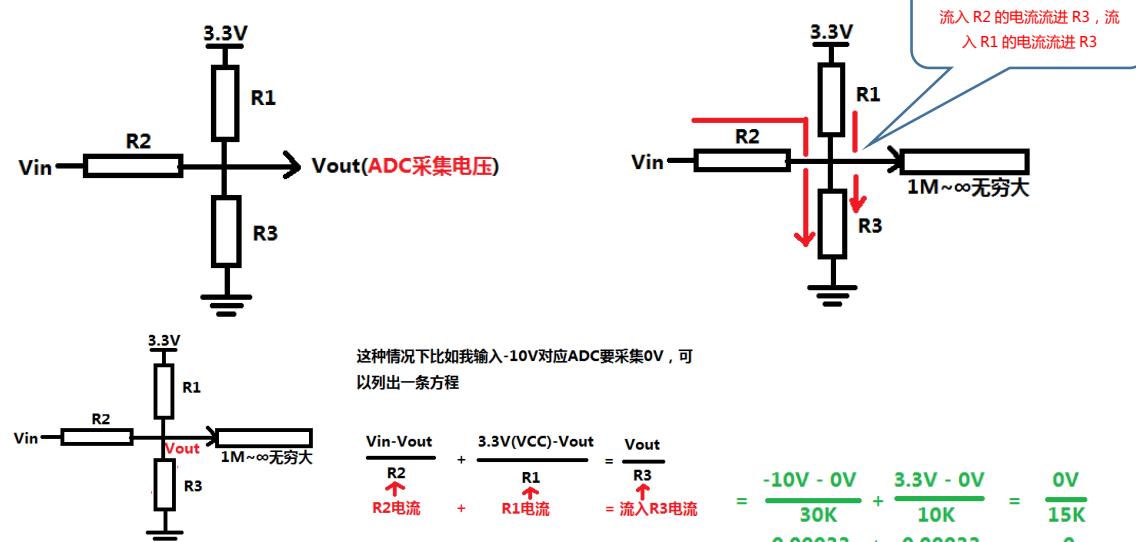
如果 VREF+接 3.3V, VREF-接 0V, 那么 ADC 电压采集范围就是 0~3.3V

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
2 {
3     uint32_t ADC_Mode;                      // ADC 工作模式选择
4     FunctionalState ADC_ScanConvMode;        /* ADC 扫描（多通道）
5                                     或者单次（单通道）模式选择 */
6     FunctionalState ADC_ContinuousConvMode;   // ADC 单次转换或者连续转换选择
7     uint32_t ADC_ExternalTrigConv;            // ADC 转换触发信号选择
8     uint32_t ADC_DataAlign;                  // ADC 数据寄存器对齐格式
9     uint8_t ADC_NbrOfChannel;                // ADC 采集通道数
0 } 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_ReqInjecSimult ((uint32_t)0x00010000)
#define	ADC_Mode_ReqSimult ((uint32_t)0x00060000)
#define	ADC_Mode_ReqSimult_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?

ADC采集的精度是12位



将ADC12位的数据放入16位寄存器

数据寄存器是16位



MCU用变量去获取数据

如果左对齐



就是将ADC 12位数据放在16位数据寄存器的左边

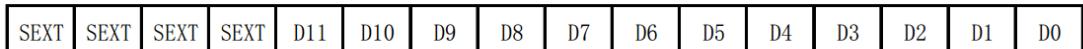
如果右对齐



就是将ADC 12位数据放在寄存器右边

图29 数据右对齐

注入组



规则组

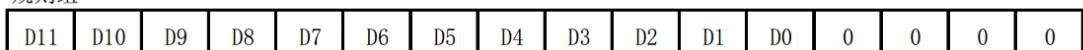


图30 数据左对齐

注入组



规则组



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

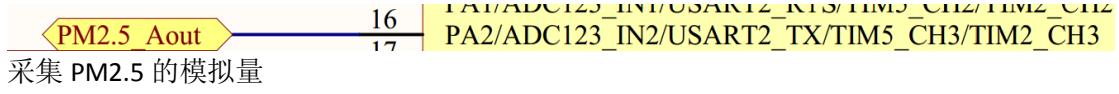
我采集的数字量必须要右移4位才是正确的数字量。才能赋值给变量

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



```

/*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)); //等待校准完成

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

/*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();
}

```

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 PA 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则

初始化 ADC 引脚, ADC 功能和 ADC 中断优先级的顺序

```
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

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

    }
    return 0;
}
```

```
ADC value = 21
xxxxzzzz
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

ADC 单通道使用 DMA 读取

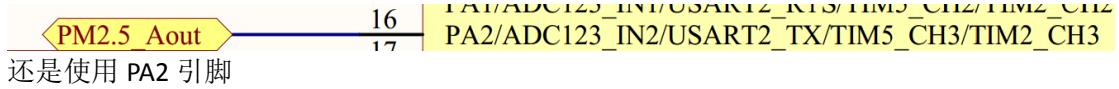


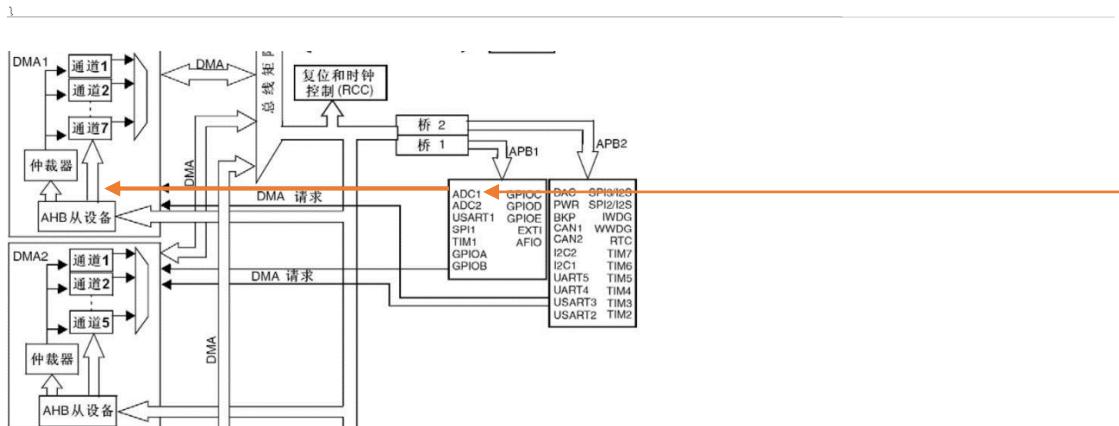
表59 各个通道的DMA1请求一览

外设	通道1	通道2	通道3	通道4	通道5	通道6	通道7
ADC1	ADC1						
SPI/I ² S		SPI1_RX	SPI1_TX	SPI/I2S2_RX	SPI/I2S2_TX		
USART		USART3_TX	USART3_RX	USART1_TX	USART1_RX	USART2_RX	USART2_TX
I ² C				I2C2_TX	I2C2_RX	I2C1_TX	I2C1_RX
TIM1		TIM1_CH1	TIM1_CH2	TIM1_TX4 TIM1_TRIG TIM1_COM	TIM1_UP	TIM1_CH3	
TIM2	TIM2_CH3	TIM2_UP			TIM2_CH1		TIM2_CH2 TIM2_CH4
TIM3		TIM3_CH3	TIM3_CH4 TIM3_UP			TIM3_CH1 TIM3_TRIG	
TIM4	TIM4_CH1			TIM4_CH2	TIM4_CH3		TIM4_UP

这是 STM32F103 DMA 支持的外设，我们使用的是 ADC1，那么要开启 DMA 通道 1

extern __IO uint16_t ADC_ConvertedValue; // 存储 DMA 得到 ADC DR 寄存器的数据

```
void ADC_DMA_Init(void)//初始化
{
    DMA_InitTypeDef DMA_InitStructure;
    RCC_AHBPeriphClockCmd(RCC_AHBPeriph_DMA1, ENABLE); // 打开 DMA 时钟
    // ADC1 对应 DMA1 通道 1, ADC3 对应 DMA2 通道 5, ADC2 没有 DMA 功能
    DMA_DeInit(DMA1_Channel1); // 复位 DMA1 通道 1 控制器
    DMA_InitStructure.DMA_PeripheralBaseAddr = (uint32_t)(&(ADC1->DR)); // 外设基址为：ADC1 数据寄存器地址
    DMA_InitStructure.DMA_MemoryBaseAddr = (uint32_t)&ADC_ConvertedValue; // 存储器地址
    DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralSRC; // 数据源来自外设
    DMA_InitStructure.DMA_BufferSize = 5; // 缓冲区大小，应该等于数据目的地的大小
    DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable; // 外设寄存器只有一个，地址不用递增
    DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Disable; // 存储器地址不用递增，因为设置了 ADC_ConvertedValue 1 个变量
    DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_HalfWord; // 外设数据大小为半字，即两个字节
    DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_HalfWord; // 内存数据大小也为半字，跟外设数据大小相同
    DMA_InitStructure.DMA_Mode = DMA_Mode_Circular; // 循环传输模式 DMA 不断的重复传输数据
    DMA_InitStructure.DMA_Priority = DMA_Priority_High; // DMA 传输通道优先级为高，当使用一个 DMA 通道时，优先级设置不影响
    DMA_InitStructure.DMA_M2M = DMA_M2M_Disable; // 禁止存储器到存储器模式，因为是从外设到存储器
    DMA_Init(DMA1_Channel1, &DMA_InitStructure); // 初始化 DMA1 通道 1
    DMA_Cmd(DMA1_Channel1, ENABLE); // 使能 DMA1 通道
}
```



```

/*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);
}

/*ADC功能初始化*/
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, 不使用, 因为我们用的DMA中断
    ADC_DMACmd(ADC1, ENABLE); //开启ADC DMA请求
    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ADC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
}

void ADC_init_Cfg(void)
{
    ADC_GPIO_init();
    ADC_Mode_Config();
    ADC_DMA_Init();
}

uint16_t ADC_ConvertedValue;

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 DMA
    printf("xxxxxxxx\r\n");
    while(1)
    {
        printf("ADC value = %d\r\n", ADC_ConvertedValue);
        delay_ms(500);
    }
    return 0;
}

```

ADC value = 22
ADC value = 84
ADC value = 20
ADC value = 18
ADC value = 19
ADC value = 22
ADC value = 20
ADC value = 23
ADC value = 25
ADC value = 204
ADC value = 110
ADC value = 19

数据输出正确，ADC 单通道 DMA 调试完成

下面贴上源代码

```
/*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);
}

/*ADC 功能初始化*/
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,不使用, 因为我们用的 DMA 中断

    ADC_DMAConfig(ADC1,ENABLE); //开启 ADC DMA 请求
    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
}

extern __IO uint16_t ADC_ConvertedValue; //存储 DMA 得到 ADC DR 寄存器的数据

void ADC_DMA_Init(void) //初始化
{
    DMA_InitTypeDef DMA_InitStructure;

    RCC_AHBPeriphClockCmd(RCC_AHBPeriph_DMA1, ENABLE); //打开 DMA 时钟

    // ADC1 对应 DMA1 通道 1, ADC3 对应 DMA2 通道 5, ADC2 没有 DMA 功能
    DMA_DeInit(DMA1_Channel1); //复位 DMA1 通道 1 控制器

    DMA_InitStructure.DMA_PeripheralBaseAddr = (uint32_t)&(ADC1->DR); //外设基址为: ADC1 数据寄存器地址

    DMA_InitStructure.DMA_MemoryBaseAddr = (uint32_t)&ADC_ConvertedValue; //存储器地址

    DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralSRC; //数据源来自外设

    DMA_InitStructure.DMA_BufferSize = 5; //缓冲区大小, 应该等于数据目的地的大小

    DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable; //外设寄存器只有一个, 地址不用递增

    DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Disable; //存储器地址不用递增, 因为设置了 ADC_ConvertedValue 1 个变量

    DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_HalfWord; //外设数据大小为半字, 即两个字节

    DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_HalfWord; //内存数据大小也为半字, 跟外设数据大小相同

    DMA_InitStructure.DMA_Mode = DMA_Mode_Circular; //循环传输模式 DMA 不断的重复传输数据

    DMA_InitStructure.DMA.Priority = DMA_Priority_High; //DMA 传输通道优先级为高, 当使用一个 DMA 通道时, 优先级设置不影响
}
```

```

DMA_InitStructure.DMA_M2M = DMA_M2M_Disable;// 禁止存储器到存储器模式，因为是从外设到存储器
DMA_Init(DMA1_Channel1, &DMA_InitStructure);// 初始化 DMA1 通道 1
DMA_Cmd(DMA1_Channel1, ENABLE);// 使能 DMA1 通道
}

void ADC_init_Cfg(void)
{
    ADC_GPIO_Init();
    ADC_Mode_Config();
    ADC_DMA_Init();
}

uint16_t ADC_ConvertedValue;

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 DMA

    printf("xxxxzzzz\r\n");
    while(1)
    {
        printf("ADC value = %d\r\n",ADC_ConvertedValue);
        delay_ms(500);
    }
    return 0;
}

```

ADC 多通道采集， DMA 读取数据

14	PA0/WKUP/ADC123_IN0/USART2_CTS/TIM2_CH1_ETR
15	PA1/ADC123_IN1/USART2_RTS/TIM5_CH2/TIM2_CH2
16	PA2/ADC123_IN2/USART2_TX/TIM5_CH3/TIM2_CH3
17	PA3/USART2_RX/TIM5_CH4/ADC123_IN3/TIM2_CH4
18	PA4/SPI1_NSS/USART2_CK/DAC_OUT1/ADC12_IN4
24	PA5/SPI1_SCK/DAC_OUT2/ADC12_IN5
25	PA6/SPI1_MISO/TIM8_BKIN/ADC12_IN6/TIM3_CH1
26	PA7/SPI1_MOSI/TIM8_CH1N/ADC12_IN7/TIM3_CH2
27	PC4/ADC12_IN14
28	PC5/ADC12_IN15
29	PB0/ADC12_IN8/TIM3_CH3/TIM8_CH2N
30	PB1/ADC12_IN9/TIM3_CH4/TIM8_CH3N
31	PC0/ADC123_IN10
32	PC1/ADC123_IN11
33	PC2/ADC123_IN12
34	PC3/ADC123_IN13

我们把 ADC 通道 0~15 全部打开

PA0,PA1,PA2,PA3,PA4,PA5,PA6,PA7.[ADC 通道 0~7]

PB0,PB1,PC0,PC1,PC2,PC3,PC4,PC5.[ADC 通道 8~15]

```

/*初始化ADC所有管脚*/
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PA0~7, 所以打开APB2时钟
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0|GPIO_Pin_1|GPIO_Pin_2|GPIO_Pin_3|GPIO_Pin_4|GPIO_Pin_5|GPIO_Pin_6|GPIO_Pin_7;
    //设置GPIOA0 ~ 7为ADC通道 0 ~ 7
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AIN; //选择引脚为模拟输入, 因为使用ADC采集
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE); //我们ADC引脚是PB0, PB1, 所以打开APB2时钟
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0|GPIO_Pin_1;
    //设置GPIOB0 ~ 1为ADC通道 8 ~ 9
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AIN; //选择引脚为模拟输入, 因为使用ADC采集
    GPIO_Init(GPIOB, &GPIO_InitStructure);

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC, ENABLE); //我们ADC引脚是PC0~5, 所以打开APB2时钟
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0|GPIO_Pin_1|GPIO_Pin_2|GPIO_Pin_3|GPIO_Pin_4|GPIO_Pin_5;
    //设置GPIOB0 ~ 5为ADC通道 10 ~ 15
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AIN; //选择引脚为模拟输入, 因为使用ADC采集
    GPIO_Init(GPIOC, &GPIO_InitStructure);
}

```

```

extern __IO uint16_t ADC_ConvertedValue[16];
//存储DMA得到ADC DR寄存器的数据, 因为是16个引脚的ADC通道都要采集, 所以DMA存储变量改成数组

```

```

void ADC_DMA_Init(void)//初始化
{
    DMA_InitTypeDef DMA_InitStructure;

    RCC_AHBPeriphClockCmd(RCC_AHBPeriph_DMA1, ENABLE); //打开DMA时钟

    // ADC1 对应 DMA1 通道 1, ADC3 对应 DMA2 通道 5, ADC2 没有 DMA 功能
    DMA_DeInit(DMA1_Channel1); // 复位 DMA1 通道1 控制器

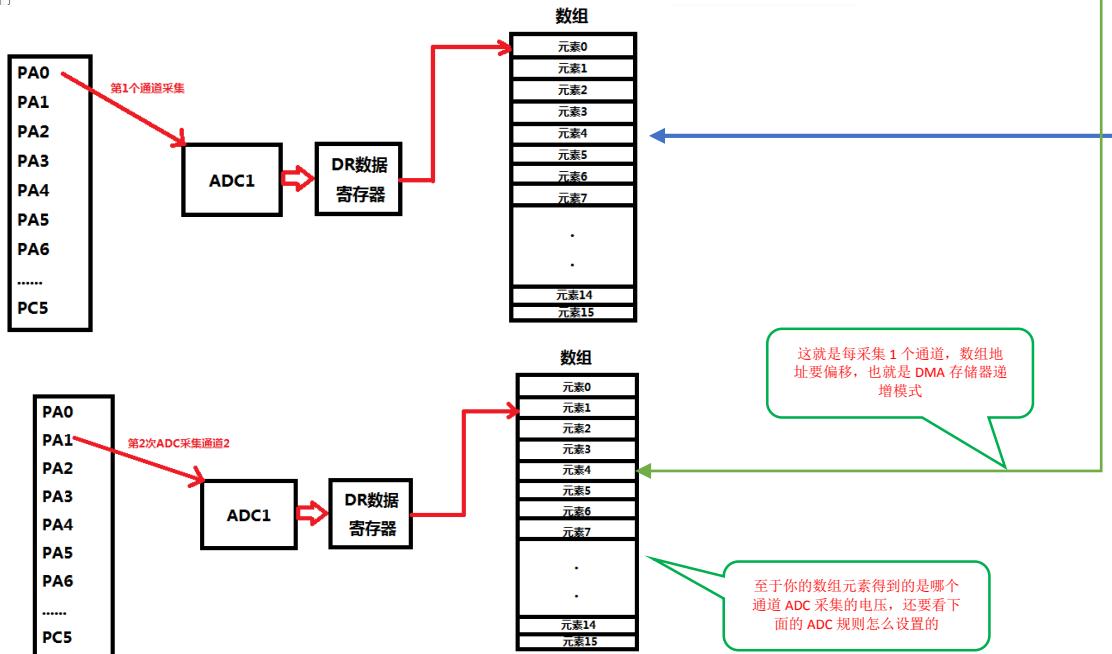
    DMA_InitStructure.DMA_PeripheralBaseAddr = (uint32_t)(&(ADC1->DR)); // 外设基址为: ADC1 数据寄存器地址
    DMA_InitStructure.DMA_MemoryBaseAddr = (uint32_t)ADC_ConvertedValue; // 存储器地址是数组
    DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralsSRC; // 数据源来自外设
    DMA_InitStructure.DMA_BufferSize = 16; // 缓冲区大小, 应该等于数据目的地的大小, 这里就是数组大小
    DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable; // 外设寄存器只有一个, 地址不用递增
    DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable; // 因为DMA_MemoryBaseAddr用的数组所以存储器地址递增, 每个元素一个ADC通道

    DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_HalfWord; // 外设数据大小为半字, 即两个字节
    DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_HalfWord; // 内存数据大小也为半字, 跟外设数据大小相同
    DMA_InitStructure.DMA_Mode = DMA_Mode_Circular; // 循环传输模式 DMA不断的重复传输数据

    DMA_InitStructure.DMA_Priority = DMA_Priority_High; // DMA 传输通道优先级为高, 当使用一个 DMA 通道时, 优先级设置不影响
    DMA_InitStructure.DMA_M2M = DMA_M2M_Disable; // 禁止存储器到存储器模式, 因为是从外设到存储器

    DMA_Init(DMA1_Channel1, &DMA_InitStructure); // 初始化 DMA1 通道1
    DMA_Cmd(DMA1_Channel1, ENABLE); // 使能 DMA1 通道
}

```



```

/*ADC功能初始化*/
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 = ENABLE; //开启扫描模式, 因为我们是多通道ADC采集
    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 = 16; //我用了16个通道ADC, 所以选择16, 这个ADC_NbrOfChannel指的是转换通道数量
    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_0, 1, ADC_SampleTime_55Cycles5); //每个通道都要指定规则, 采样时间
    ADC_RegularChannelConfig(ADC1, ADC_Channel_1, 2, ADC_SampleTime_55Cycles5); //每个通道都要指定规则, 采样时间
    ADC_RegularChannelConfig(ADC1, ADC_Channel_2, 3, ADC_SampleTime_55Cycles5); //每个通道都要指定规则, 采样时间
    ADC_RegularChannelConfig(ADC1, ADC_Channel_3, 4, ADC_SampleTime_55Cycles5); //每个通道都要指定规则, 采样时间
    ADC_RegularChannelConfig(ADC1, ADC_Channel_5, 6, ADC_SampleTime_55Cycles5); //每个通道都要指定规则, 采样时间
    ADC_RegularChannelConfig(ADC1, ADC_Channel_6, 7, ADC_SampleTime_55Cycles5); //每个通道都要指定规则, 采样时间
    ADC_RegularChannelConfig(ADC1, ADC_Channel_7, 8, ADC_SampleTime_55Cycles5); //每个通道都要指定规则, 采样时间
    ADC_RegularChannelConfig(ADC1, ADC_Channel_9, 10, ADC_SampleTime_55Cycles5); //每个通道都要指定规则, 采样时间
    ADC_RegularChannelConfig(ADC1, ADC_Channel_10, 11, ADC_SampleTime_55Cycles5); //每个通道都要指定规则, 采样时间
    ADC_RegularChannelConfig(ADC1, ADC_Channel_11, 12, ADC_SampleTime_55Cycles5); //每个通道都要指定规则, 采样时间
    ADC_RegularChannelConfig(ADC1, ADC_Channel_12, 13, ADC_SampleTime_55Cycles5); //每个通道都要指定规则, 采样时间
    ADC_RegularChannelConfig(ADC1, ADC_Channel_13, 14, ADC_SampleTime_55Cycles5); //每个通道都要指定规则, 采样时间
    ADC_RegularChannelConfig(ADC1, ADC_Channel_14, 15, ADC_SampleTime_55Cycles5); //每个通道都要指定规则, 采样时间
    ADC_RegularChannelConfig(ADC1, ADC_Channel_15, 16, ADC_SampleTime_55Cycles5); //每个通道都要指定规则, 采样时间

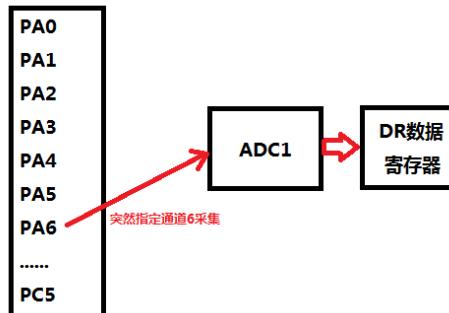
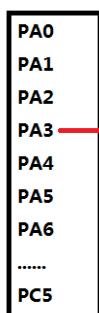
    // ADC_ITConfig(ADC1, ADC_IT_EOC, ENABLE); //ADC每次转换结束产生中断, 不使用, 因为我们用的DMA中断
    ADC_DMAConfig(ADC1, ENABLE); //开启ADC DMA请求
    ADC_Cmd(ADC1, ENABLE); //开启ADC

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

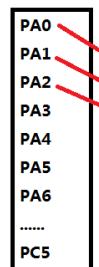
    ADC_StartCalibration(ADC1); //开始正式校准ADC
    while (ADC_GetCalibrationStatus(ADC1));

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

```



STM32 多通道 ADC 采集, 在同一时间内突然指定通道几去采集是不行的。



只能根据规则通道设置的方法来决定哪一个通道第1次采集, 哪一个通道第2次采集, 采集后, 数组的元素下标和通道不是1-2-3-4-5 对应数组下标1-2-3-4-5这种对应关系了, 而是你设置的第几个通道第1个采集, 数组下标0就是存放第几个通道的数据

```

void ADC_init_Cfg(void)
{
    ADC_GPIO_init();
    ADC_Mode_Config();
    ADC_DMA_Init();
}

uint16_t ADC_ConvertedValue[16];

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 DMA

    printf("xxxxxxxx\r\n");
    while(1)
    {
        printf("ADC channel0 PA0 value = %d ", ADC_ConvertedValue[0]);
        printf(" channel1 PA1 value = %d ", ADC_ConvertedValue[1]);
        printf(" channel2 PA2 value = %d ", ADC_ConvertedValue[2]);
        printf(" channel3 PA3 value = %d\r\n", ADC_ConvertedValue[3]);
        printf("ADC channel4 PA4 value = %d ", ADC_ConvertedValue[4]);
        printf(" channel5 PA5 value = %d ", ADC_ConvertedValue[5]);
        printf(" channel6 PA6 value = %d ", ADC_ConvertedValue[6]);
        printf(" channel7 PA7 value = %d\r\n", ADC_ConvertedValue[7]);
        printf("ADC channel8 PB0 value = %d ", ADC_ConvertedValue[8]);
        printf(" channel9 PB1 value = %d ", ADC_ConvertedValue[9]);
        printf(" channel10 PC1 value = %d ", ADC_ConvertedValue[10]);
        printf(" channel11 PC2 value = %d\r\n", ADC_ConvertedValue[11]);
        printf("ADC channel12 PC3 value = %d ", ADC_ConvertedValue[12]);
        printf(" channel13 PC4 value = %d ", ADC_ConvertedValue[13]);
        printf(" channel14 PC5 value = %d ", ADC_ConvertedValue[14]);
        printf(" channel15 PC6 value = %d ", ADC_ConvertedValue[15]);
        printf("\r\n");
        printf("===== \r\n");
        delay_ms(500);
    }
}

ADC channel0 PA0 value = 2206 channel1 PA1 value = 117 channel2 PA2 value = 15 channel3 PA3 value = 4094
ADC channel4 PA4 value = 2899 channel5 PA5 value = 2344 channel6 PA6 value = 2087 channel7 PA7 value = 1968
ADC channel8 PB0 value = 4095 channel9 PB1 value = 4093 channel10 PC1 value = 2897 channel11 PC2 value = 2347
ADC channel12 PC3 value = 2089 channel13 PC4 value = 1973 channel14 PC5 value = 2195 channel15 PC6 value = 2020
=====
ADC channel0 PA0 value = 2206 channel1 PA1 value = 118 channel2 PA2 value = 30 channel3 PA3 value = 4094
ADC channel4 PA4 value = 2899 channel5 PA5 value = 2344 channel6 PA6 value = 2087 channel7 PA7 value = 1968
ADC channel8 PB0 value = 4095 channel9 PB1 value = 4093 channel10 PC1 value = 2898 channel11 PC2 value = 2345
ADC channel12 PC3 value = 2090 channel13 PC4 value = 1972 channel14 PC5 value = 2197 channel15 PC6 value = 2021
=====
ADC channel0 PA0 value = 2206 channel1 PA1 value = 118 channel2 PA2 value = 47 channel3 PA3 value = 4093
ADC channel4 PA4 value = 2899 channel5 PA5 value = 2344 channel6 PA6 value = 2087 channel7 PA7 value = 1968
ADC channel8 PB0 value = 4095 channel9 PB1 value = 4093 channel10 PC1 value = 2898 channel11 PC2 value = 2345
ADC channel12 PC3 value = 2090 channel13 PC4 value = 1972 channel14 PC5 value = 2196 channel15 PC6 value = 2021
=====
ADC channel0 PA0 value = 2206 channel1 PA1 value = 118 channel2 PA2 value = 6 channel3 PA3 value = 4094
ADC channel4 PA4 value = 2899 channel5 PA5 value = 2344 channel6 PA6 value = 2087 channel7 PA7 value = 1968
ADC channel8 PB0 value = 4095 channel9 PB1 value = 4093 channel10 PC1 value = 2899 channel11 PC2 value = 2345
ADC channel12 PC3 value = 2090 channel13 PC4 value = 1971 channel14 PC5 value = 2196 channel15 PC6 value = 2021
=====
ADC channel0 PA0 value = 2205 channel1 PA1 value = 119 channel2 PA2 value = 6 channel3 PA3 value = 4094
ADC channel4 PA4 value = 2899 channel5 PA5 value = 2344 channel6 PA6 value = 2087 channel7 PA7 value = 1968
ADC channel8 PB0 value = 4095 channel9 PB1 value = 4093 channel10 PC1 value = 2897 channel11 PC2 value = 2346
ADC channel12 PC3 value = 2090 channel13 PC4 value = 1973 channel14 PC5 value = 2196 channel15 PC6 value = 2020

```

你看我通道 2 , PA2 数据变化还是正常的, 其余通道也采集到数据了

下面贴出源代码

```

uint16_t ADC_ConvertedValue[16];

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 DMA

    printf("xxxxxxxx\r\n");
    while(1)
    {

```

```

        printf("ADC channel0 PA0 value = %d ",ADC_ConvertedValue[0]);
        printf(" channel1 PA1 value = %d ",ADC_ConvertedValue[1]);
        printf(" channel2 PA2 value = %d ",ADC_ConvertedValue[2]);
        printf(" channel3 PA3 value = %d\r\n",ADC_ConvertedValue[3]);
        printf("ADC channel4 PA4 value = %d ",ADC_ConvertedValue[4]);
        printf(" channel5 PA5 value = %d ",ADC_ConvertedValue[5]);
        printf(" channel6 PA6 value = %d ",ADC_ConvertedValue[6]);
        printf(" channel7 PA7 value = %d\r\n",ADC_ConvertedValue[7]);
        printf("ADC channel8 PB0 value = %d ",ADC_ConvertedValue[8]);
        printf(" channel9 PB1 value = %d ",ADC_ConvertedValue[9]);
        printf(" channel10 PC1 value = %d ",ADC_ConvertedValue[10]);
        printf(" channel11 PC2 value = %d\r\n",ADC_ConvertedValue[11]);
        printf("ADC channel12 PC3 value = %d ",ADC_ConvertedValue[12]);
        printf(" channel13 PC4 value = %d ",ADC_ConvertedValue[13]);
        printf(" channel14 PC5 value = %d ",ADC_ConvertedValue[14]);
        printf(" channel15 PC6 value = %d ",ADC_ConvertedValue[15]);
        printf("\r\n");
        printf("=====\\r\\n");
        delay_ms(500);

    }
    return 0;
}

/*初始化 ADC 所有管脚*/
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 PA0~7,所以打开 APB2 时钟
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0|GPIO_Pin_1|GPIO_Pin_2|GPIO_Pin_3|GPIO_Pin_4|GPIO_Pin_5|GPIO_Pin_6|GPIO_Pin_7;
    //设置 GPIOA0 ~ 7 为 ADC 通道 0 ~ 7
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AN; //选择引脚为模拟输入,因为使用 ADC 采集
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB,ENABLE); //我们 ADC 引脚是 PB0,PB1,所以打开 APB2 时钟
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0|GPIO_Pin_1;
    //设置 GPIOB0 ~ 1 为 ADC 通道 8 ~ 9
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AN; //选择引脚为模拟输入,因为使用 ADC 采集
    GPIO_Init(GPIOB, &GPIO_InitStructure);

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC,ENABLE); //我们 ADC 引脚是 PC0~5,所以打开 APB2 时钟
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0|GPIO_Pin_1|GPIO_Pin_2|GPIO_Pin_3|GPIO_Pin_4|GPIO_Pin_5;
    //设置 GPIOB0 ~ 5 为 ADC 通道 10 ~ 15
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AN; //选择引脚为模拟输入,因为使用 ADC 采集
    GPIO_Init(GPIOC, &GPIO_InitStructure);

}

/*ADC 功能初始化*/
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 = ENABLE; //开启扫描模式, 因为我们是多通道 ADC 采集
    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 = 16;
    //我用了 16 个通道 ADC,所以选择 16,这个 ADC_NbrOfChannel 指的是转换通道数量

    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_0,1,ADC_SampleTime_55Cycles5); //每个通道都要指定规则, 采样时间
    ADC-RegularChannelConfig(ADC1,ADC_Channel_1,2,ADC_SampleTime_55Cycles5); //每个通道都要指定规则, 采样时间
    ADC-RegularChannelConfig(ADC1,ADC_Channel_2,3,ADC_SampleTime_55Cycles5); //每个通道都要指定规则, 采样时间
    ADC-RegularChannelConfig(ADC1,ADC_Channel_3,4,ADC_SampleTime_55Cycles5); //每个通道都要指定规则, 采样时间
    ADC-RegularChannelConfig(ADC1,ADC_Channel_4,5,ADC_SampleTime_55Cycles5); //每个通道都要指定规则, 采样时间
    ADC-RegularChannelConfig(ADC1,ADC_Channel_5,6,ADC_SampleTime_55Cycles5); //每个通道都要指定规则, 采样时间
    ADC-RegularChannelConfig(ADC1,ADC_Channel_6,7,ADC_SampleTime_55Cycles5); //每个通道都要指定规则, 采样时间
    ADC-RegularChannelConfig(ADC1,ADC_Channel_7,8,ADC_SampleTime_55Cycles5); //每个通道都要指定规则, 采样时间
    ADC-RegularChannelConfig(ADC1,ADC_Channel_8,9,ADC_SampleTime_55Cycles5); //每个通道都要指定规则, 采样时间
    ADC-RegularChannelConfig(ADC1,ADC_Channel_9,10,ADC_SampleTime_55Cycles5); //每个通道都要指定规则, 采样时间
    ADC-RegularChannelConfig(ADC1,ADC_Channel_10,11,ADC_SampleTime_55Cycles5); //每个通道都要指定规则, 采样时间
    ADC-RegularChannelConfig(ADC1,ADC_Channel_11,12,ADC_SampleTime_55Cycles5); //每个通道都要指定规则, 采样时间
    ADC-RegularChannelConfig(ADC1,ADC_Channel_12,13,ADC_SampleTime_55Cycles5); //每个通道都要指定规则, 采样时间
    ADC-RegularChannelConfig(ADC1,ADC_Channel_13,14,ADC_SampleTime_55Cycles5); //每个通道都要指定规则, 采样时间
}

```

```

ADC-RegularChannelConfig(ADC1,ADC_Channel_14,15,ADC_SampleTime_55Cycles5);//每个通道都要指定规则，采样时间
ADC-RegularChannelConfig(ADC1,ADC_Channel_15,16,ADC_SampleTime_55Cycles5);//每个通道都要指定规则，采样时间

// ADC_ITConfig(ADC1, ADC_IT_EOC, ENABLE);//ADC 每次转换结束产生中断,不使用，因为我们用的 DMA 中断
ADC_DMACmd(ADC1,ENABLE);//开启 ADC DMA 请求
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
}

extern __IO uint16_t ADC_ConvertedValue[16];
//存储 DMA 得到 ADC DR 寄存器的数据,因为是 16 个引脚的 ADC 通道都要采集，所以 DMA 存储变量改成数组

void ADC_DMA_Init(void)//初始化
{
    DMA_InitTypeDef DMA_InitStructure;

    RCC_AHBPeriphClockCmd(RCC_AHBPeriph_DMA1, ENABLE);//打开 DMA 时钟

    // ADC1 对应 DMA1 通道 1， ADC3 对应 DMA2 通道 5， ADC2 没有 DMA 功能
    DMA_DeInit(DMA1_Channel1);// 复位 DMA1 通道 1 控制器

    DMA_InitStructure.DMA_PeripheralBaseAddr = (uint32_t)(&(ADC1->DR));// 外设基址为： ADC1 数据寄存器地址
    DMA_InitStructure.DMA_MemoryBaseAddr = (uint32_t)ADC_ConvertedValue;// 存储器地址是数组

    DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralSRC;// 数据源来自外设

    DMA_InitStructure.DMA_BufferSize = 16;// 缓冲区大小，应该等于数据目的地的大小,这里就是数组大小

    DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable;// 外设寄存器只有一个，地址不用递增

    DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable;
    //因为 DMA_MemoryBaseAddr 用的数组所以存储器地址递增，每个元素一个 ADC 通道

    DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_HalfWord;// 外设数据大小为半字，即两个字节

    DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_HalfWord;// 内存数据大小也为半字，跟外设数据大小相同

    DMA_InitStructure.DMA_Mode = DMA_Mode_Circular;// 循环传输模式 DMA 不断的重复传输数据

    DMA_InitStructure.DMA_Priority = DMA_Priority_High;// DMA 传输通道优先级为高，当使用一个 DMA 通道时，优先级设置不影响

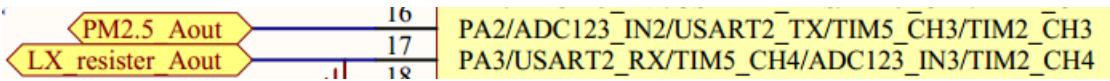
    DMA_InitStructure.DMA_M2M = DMA_M2M_Disable;// 禁止存储器到存储器模式，因为是从外设到存储器

    DMA_Init(DMA1_Channel1, &DMA_InitStructure);// 初始化 DMA1 通道 1
    DMA_Cmd(DMA1_Channel1, ENABLE);// 使能 DMA1 通道
}

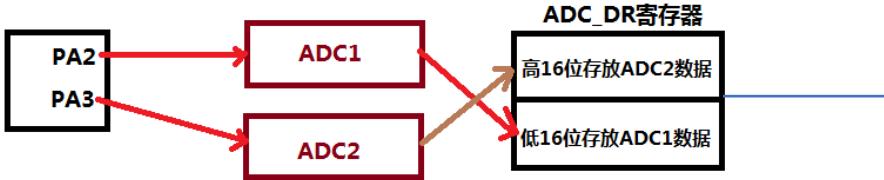
void ADC_init_Cfg(void)
{
    ADC_GPIO_init();
    ADC_Mode_Config();
    ADC_DMA_Init();
}

```

ADC1 和 ADC2 两个 ADC 控制器同时采集多通道



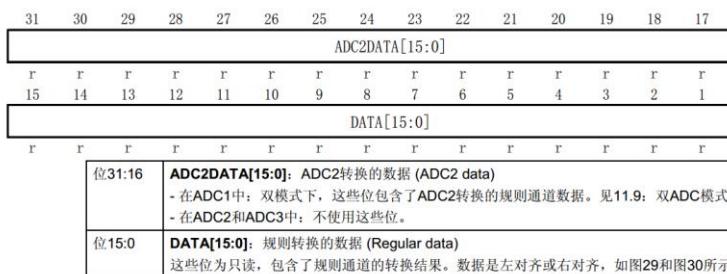
PA2 用 ADC1 模块采集， PA3 用 ADC2 模块采集



11.12.14 ADC 规则数据寄存器(ADC_DR)

地址偏移: 0x4C

复位值: 0x0000 0000



```
static void ADC_GPIO_init()
{
    GPIO_InitTypeDef GPIO_InitStruct; //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_InitStruct.GPIO_Pin = GPIO_Pin_2; //选择PA2_ADCIN2引脚
    GPIO_InitStruct.GPIO_Mode = GPIO_Mode_AN; //选择PA2引脚为模拟输入, 因为使用PA2为ADC1采集
    GPIO_Init(GPIOA, &GPIO_InitStruct);

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE); //我们ADC引脚是PA3, 所以打开APB2时钟
    GPIO_InitStruct.GPIO_Pin = GPIO_Pin_3; //选择PA3_ADCIN3引脚
    GPIO_InitStruct.GPIO_Mode = GPIO_Mode_AN; //选择PA3引脚为模拟输入, 因为使用PA3为ADC2采集
    GPIO_Init(GPIOA, &GPIO_InitStruct);
}
```

初始化 IO 口

```
extern __IO uint32_t ADC_ConvertedValue; //存储DMA到ADC DR寄存器的数据因为是双ADC, 所以要32位

void ADC_DMA_Init(void) //初始化
{
    DMA_InitTypeDef DMA_InitStruct;
    RCC_AHBPeriphClockCmd(RCC_AHBPeriph_DMA1, ENABLE); //打开DMA时钟
    // ADC1 对应 DMA1 通道 1, ADC3 对应 DMA2 通道 5, ADC2 没有 DMA 功能
    DMA_DeInit(DMA1_Channel1); // 复位 DMA1 通道1 控制器
    DMA_InitStruct.DMA_PeripheralBaseAddr = (uint32_t)((ADC1->DR)); // 外设基址为: ADC1 数据寄存器地址
    DMA_InitStruct.DMA_MemoryBaseAddr = (uint32_t)&ADC_ConvertedValue; // 存储器地址
    DMA_InitStruct.DMA_DIR = DMA_DIR_PeripheralSRC; // 数据源来自外设
    DMA_InitStruct.DMA_BufferSize = 1; // 缓冲区大小, 应该等于数据目的地的大小, 如果ADC采集2个通道这里就要改成2
    DMA_InitStruct.DMA_PeripheralInc = DMA_PeripheralInc_Disable; // 外设寄存器只有一个, 地址不用递增
    DMA_InitStruct.DMA_MemoryInc = DMA_MemoryInc_Disable; // 存储器地址不用递增, 因为设置了ADC_ConvertedValue 1个变量
    DMA_InitStruct.DMA_PeripheralDataSize = DMA_PeripheralDataSize_Word; // 外设数据为32位, 即4个字节
    DMA_InitStruct.DMA_MemoryDataSize = DMA_MemoryDataSize_Word; // 内存数据大小也为32位, 跟外设数据大小相同
    DMA_InitStruct.DMA_Mode = DMA_Mode_Circular; // 循环传输模式 DMA不断的重复传输数据
    DMA_InitStruct.DMA_Priority = DMA_Priority_High; // DMA 传输通道优先级为高, 当使用一个 DMA 通道时, 优先级设置不影响
    DMA_InitStruct.DMA_M2M = DMA_M2M_Disable; // 禁止存储器到存储器模式, 因为是从外设到存储器
    DMA_Init(DMA1_Channel1, &DMA_InitStruct); // 初始化 DMA1 通道1
    DMA_Cmd(DMA1_Channel1, ENABLE); // 使能 DMA1 通道
```

存储变量位宽记得修改, 因为高 16 位数 ADC2 采集的数据, 低 16 位数 ADC1 采集的数据

DMA 还是获取 ADC1 的数据寄存器, 但是 ADC1 的 DR 寄存器高 16 位会包含 ADC2 数据

这里要修改, 因为 DR 使用 32 位 数据了

ADC 初始化顺序一定要严格按照面的来，不然 ADC2 无法正常工作

```
/*ADC功能初始化*/
static void ADC_Mode_Config()
{
    ADC_InitTypeDef ADC_InitStructure;
    ADC_InitTypeDef ADC_InitStructure2;

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1, ENABLE); //我们使用ADC1, 打开ADC1时钟
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC2, ENABLE); //我们使用ADC2, 打开ADC2时钟

    ADC_InitStructure. ADC_Mode = ADC_Mode_RegSimult; //因为是双ADC, 设置ADC为规则同步模式
    ADC_InitStructure. ADC_ScanConvMode = ENABLE; //打开扫描模式
    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ADC数量, ADC1和ADC2都是采集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 配置1次管ADC1, ADC2

    ADC-RegularChannelConfig(ADC1, ADC_Channel_1, 1, ADC_SampleTime_55Cycles5); //指定规则通道转换顺序 PA2
    ADC_DMACmd(ADC1, ENABLE); //开启ADC1 DMA请求, 只有ADC1有DMA所以使能, ADC2没有DMA
    //*****ADC1配置完, 初始化ADC2*****
    ADC_InitStructure2. ADC_Mode = ADC_Mode_RegSimult; //和ADC1配置一样
    ADC_InitStructure2. ADC_ScanConvMode = ENABLE; //和ADC1配置一样
    ADC_InitStructure2. ADC_ContinuousConvMode = ENABLE; //和ADC1配置一样
    ADC_InitStructure2. ADC_ExternalTrigConv = ADC_ExternalTrigConv_None; //和ADC1配置一样
    ADC_InitStructure2. ADC_DataAlign = ADC_DataAlign_Right; //和ADC1配置一样
    ADC_InitStructure2. ADC_NbrOfChannel = 1; //和ADC1配置一样
    ADC_Init(ADC2, &ADC_InitStructure); //和ADC1配置一样
    RCC_ADCCLKConfig(RCC_PCLK2_Div8);

    ADC-RegularChannelConfig(ADC2, ADC_Channel_3, 1, ADC_SampleTime_55Cycles5); //和ADC1配置一样, 因为是PA3采集, 通道改成3
    ADC_ExternalTrigConvCmd(ADC2, ENABLE); //ADC2不能像ADC1那样软件触发, 所以只有设置为外部触发
    //ADC2触发源来自于ADC1的触发, 所以是外部触发
    //*****ADC2*****
```

这是初始化顺序的关键, 1. 设置 ADC2 触发方式

这是初始化顺序的关键, 2. 可以先打开 ADC1

这是初始化顺序的关键, 3. 可以再打开 ADC2

这是初始化顺序的关键, 4. ADC1 软件触发一定最后打开

```
uint32_t ADC_ConvertedValue;
int main(void)
{
    uint16_t ADC_1, ADC_2;

    RCC_Configuration(); //初始化时钟
    USART_Config(115200); //初始化串口
    ADC_init_Cfg(); //初始化启动ADC DMA

    printf("xxxxxxxx\r\n");
    while(1)
    {
        ADC_2 = ((ADC_ConvertedValue & 0xffff0000) >> 16); //这是ADC2的数据
        ADC_1 = (ADC_ConvertedValue & 0xffff); //这是ADC1的数据
        printf("ADC1 value = %d ADC2 value = %d\r\n", ADC_1, ADC_2);
        delay_ms(500);
    }
    return 0;
}
```

ADC1 和 ADC2 数据都能正常读取

下面贴出代码

ADC1 value = 18	ADC2 value = 4076
ADC1 value = 89	ADC2 value = 4076
ADC1 value = 93	ADC2 value = 4076
ADC1 value = 20	ADC2 value = 4076
ADC1 value = 20	ADC2 value = 4076
ADC1 value = 20	ADC2 value = 0
ADC1 value = 20	ADC2 value = 0
ADC1 value = 18	ADC2 value = 0
ADC1 value = 21	ADC2 value = 4076
ADC1 value = 19	ADC2 value = 4076

```

/*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 ADC1 采集
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA,ENABLE);//我们 ADC 引脚是 PA3,所以打开 APB2 时钟
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_3;//选择 PA3 ADCIN3 引脚
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AIN;//选择 PA3 引脚为模拟输入,因为使用 PA3 为 ADC2 采集
    GPIO_Init(GPIOA, &GPIO_InitStructure);
}

/*ADC 功能初始化*/
static void ADC_Mode_Config()
{
    ADC_InitTypeDef ADC_InitStructure;
    ADC_InitTypeDef ADC_InitStructure2;

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1,ENABLE);//我们使用 ADC1,打开 ADC1 时钟
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC2,ENABLE);//我们使用 ADC2,打开 ADC2 时钟

    ADC_InitStructure.ADC_Mode = ADC_Mode_RegSimult;//因为是双 ADC, 设置 ADC 为规则同步模式
    ADC_InitStructure.ADC_ScanConvMode = DISABLE;//打开扫描模式
    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 ADC 数量, ADC1 和 ADC2 都是采集 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 配置 1 次管 ADC1,ADC2

    ADC-RegularChannelConfig(ADC1,ADC_Channel_2_1,ADC_SampleTime_55Cycles5); //指定规则通道转换顺序 PA2

    ADC_DMACmd(ADC1,ENABLE); //开启 ADC1 DMA 请求, 只有 ADC1 有 DMA 所以使能, ADC2 没有 DMA

    /******ADC1 配置完, 初始化 ADC2*****/

    ADC_InitStructure2.ADC_Mode = ADC_Mode_RegSimult;//和 ADC1 配置一样
    ADC_InitStructure2.ADC_ScanConvMode = DISABLE;//和 ADC1 配置一样
    ADC_InitStructure2.ADC_ContinuousConvMode = ENABLE;//和 ADC1 配置一样

    ADC_InitStructure2.ADC_ExternalTrigConv = ADC_ExternalTrigConv_None;//和 ADC1 配置一样

    ADC_InitStructure2.ADC_DataAlign = ADC_DataAlign_Right;//和 ADC1 配置一样
    ADC_InitStructure2.ADC_NbrOfChannel = 1;//和 ADC1 配置一样

    ADC_Init(ADC2, &ADC_InitStructure); //和 ADC1 配置一样
    RCC_ADCCLKConfig(RCC_PCLK2_Div8);

    ADC-RegularChannelConfig(ADC2,ADC_Channel_3_1,ADC_SampleTime_55Cycles5); //和 ADC1 配置一样,因为是 PA3 采集, 通道改成 3

    ADC_ExernalTrigConvCmd(ADC2,ENABLE); //ADC2 不能像 ADC1 那样软件触发, 所以只有设置为外部触发
    //ADC2 触发源来自于 ADC1 的触发, 所以是外部触发

    /******ADC1*****/
    ADC_Cmd(ADC1, ENABLE); //开启 ADC1
    ADC_ResetCalibration(ADC1); //初始化校准 ADC1
    while (ADC_GetResetCalibrationStatus(ADC1)); //等待初始化 ADC1 校准完成

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

    /******ADC2*****/
    ADC_Cmd(ADC2, ENABLE); //开启 ADC2
    ADC_ResetCalibration(ADC2); //初始化校准 ADC2
    while (ADC_GetResetCalibrationStatus(ADC2)); //等待初始化 ADC2 校准完成

    ADC_StartCalibration(ADC2); //开始正式校准 ADC2
    while (ADC_GetCalibrationStatus(ADC2));

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

extern __IO uint32_t ADC_ConvertedValue; //存储 DMA 到 ADC DR 寄存器的数据因为是双 ADC, 所以要 32 位

void ADC_DMA_Init(void) //初始化
{
    DMA_InitTypeDef DMA_InitStructure;
}

```

```

RCC_AHBPeriphClockCmd(RCC_AHBPeriph_DMA1, ENABLE); // 打开 DMA 时钟

// ADC1 对应 DMA1 通道 1, ADC3 对应 DMA2 通道 5, ADC2 没有 DMA 功能
DMA_DeInit(DMA1_Channel1); // 复位 DMA1 通道 1 控制器

DMA_InitStructure.DMA_PeripheralBaseAddr = (uint32_t)&(ADC1->DR); // 外设基址为: ADC1 数据寄存器地址

DMA_InitStructure.DMA_MemoryBaseAddr = (uint32_t)&ADC_ConvertedValue; // 存储器地址

DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralSRC; // 数据源来自外设

DMA_InitStructure.DMA_BufferSize = 1; // 缓冲区大小, 应该等于数据目的地的大小, 如果 ADC 采集 2 个通道这里就要改成 2

DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable; // 外设寄存器只有一个, 地址不用递增

DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Disable; // 存储器地址不用递增, 因为设置了 ADC_ConvertedValue 1 个变量

DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_Word; // 外设数据为 32 位, 即 4 个字节

DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_Word; // 内存数据大小也为 32 位, 跟外设数据大小相同

DMA_InitStructure.DMA_Mode = DMA_Mode_Circular; // 循环传输模式 DMA 不断的重复传输数据

DMA_InitStructure.DMA_Priority = DMA_Priority_High; // DMA 传输通道优先级为高, 当使用一个 DMA 通道时, 优先级设置不影响

DMA_InitStructure.DMA_M2M = DMA_M2M_Disable; // 禁止存储器到存储器模式, 因为是从外设到存储器

DMA_Init(DMA1_Channel1, &DMA_InitStructure); // 初始化 DMA1 通道 1
DMA_Cmd(DMA1_Channel1, ENABLE); // 使能 DMA1 通道

}

void ADC_init_Cfg(void)
{
    ADC_GPIO_init();
    ADC_Mode_Config();
    ADC_DMA_Init();
}

uint32_t ADC_ConvertedValue;

int main(void)
{
    uint16_t ADC_1, ADC_2;

    RCC_Configuration(); // 初始化时钟
    USART_Config(115200); // 初始化串口

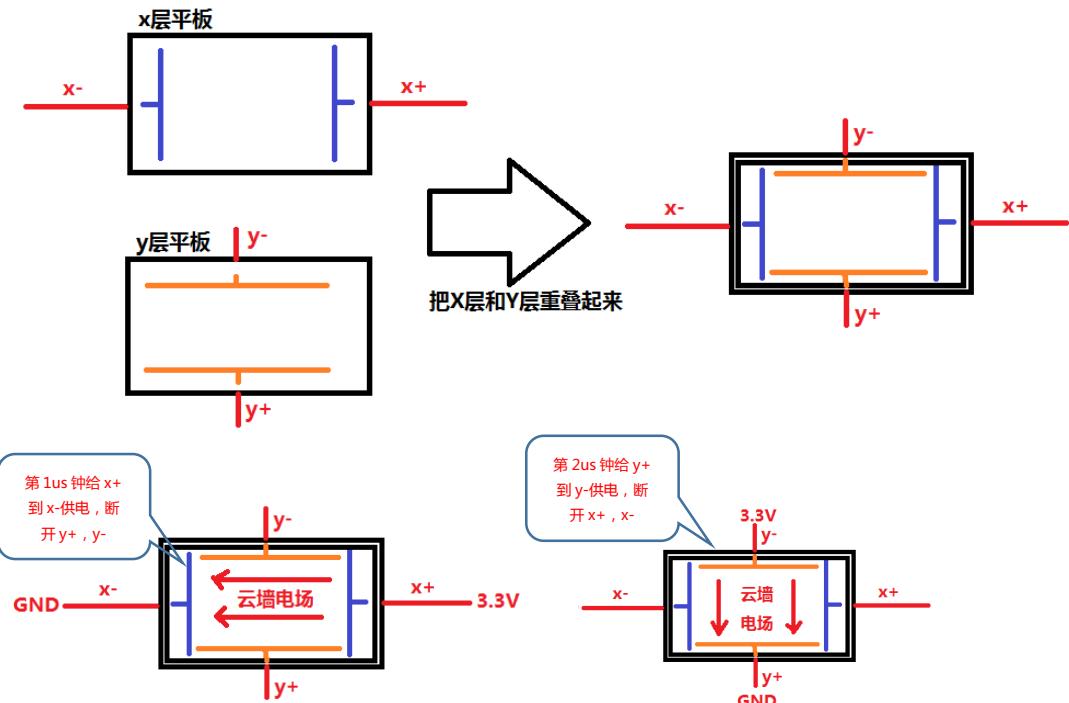
    ADC_init_Cfg(); // 初始化启动 ADC DMA

    printf("xxxxzzzz\r\n");
    while(1)
    {
        ADC_2 = ((ADC_ConvertedValue & 0xffff0000) >> 16); // 这是 ADC2 的数据
        ADC_1 = (ADC_ConvertedValue & 0xffff); // 这是 ADC1 的数据
        printf("ADC1 value = %d ADC_2 value = %d\r\n", ADC_1, ADC_2);
        delay_ms(500);
    }
    return 0;
}

```

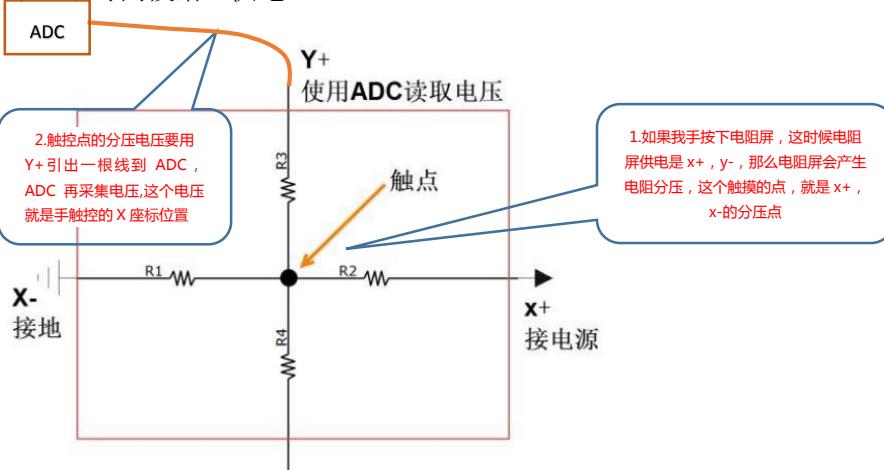
STM32 触摸屏操作

电阻触摸屏原理

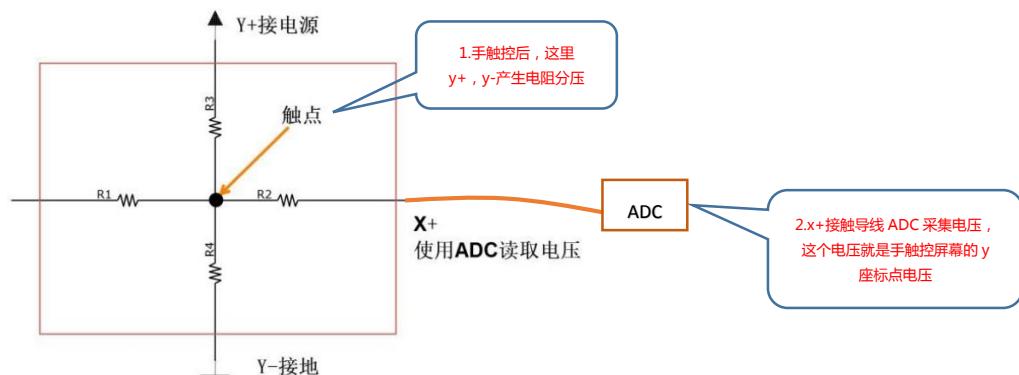


就是这种循环供电方式，不停的给 x+, x-, y+, y-交替供电。

第1个时间段给 x 供电



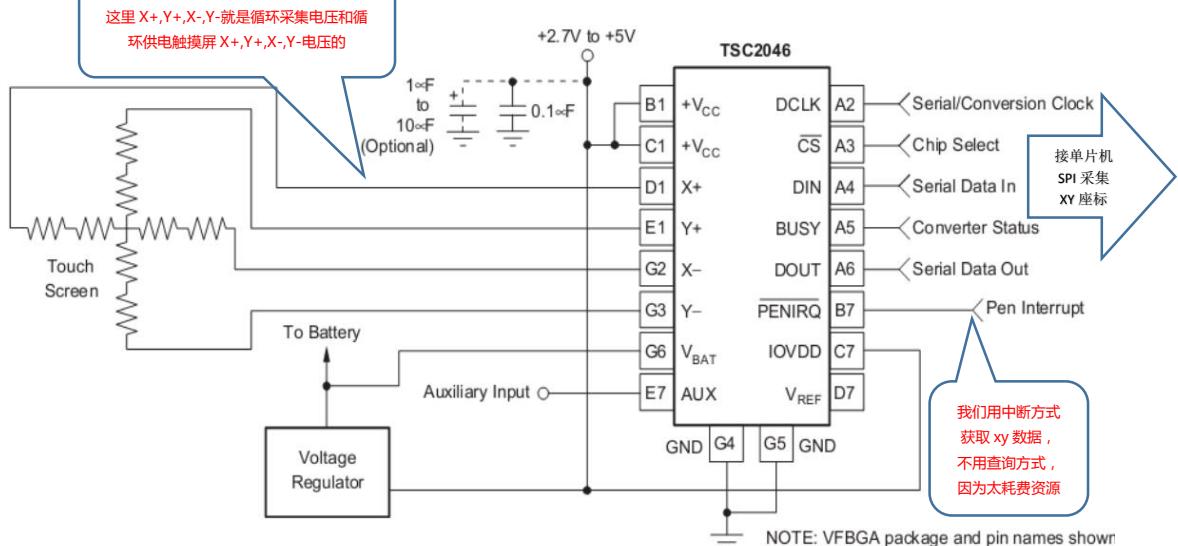
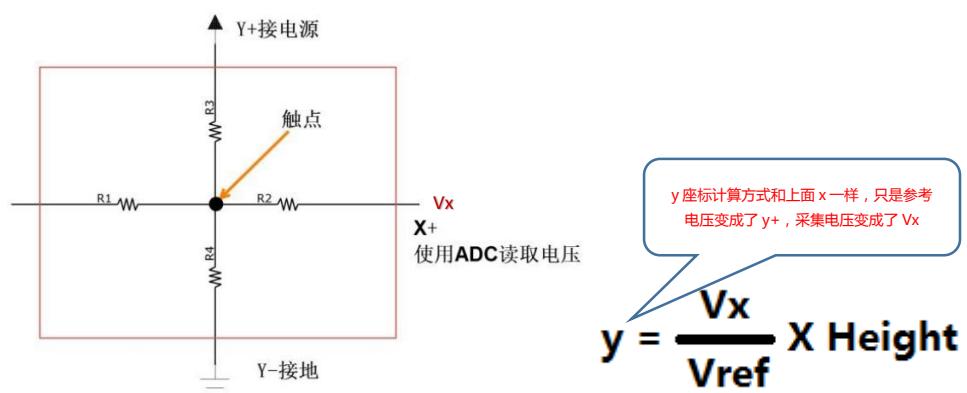
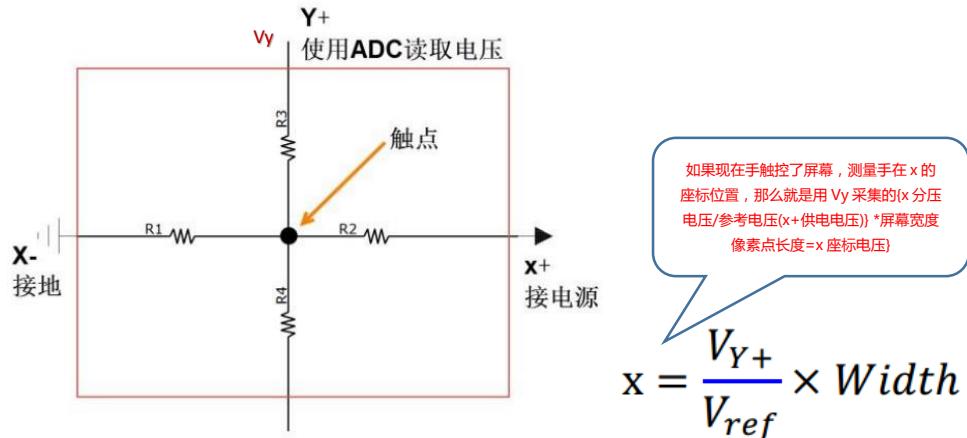
第2个时间段给 Y 供电



所以将 x 和 y 的电压循环读出来就是座标

因为 x 和 y 供电切换速度都是微妙，毫秒级别的，所以你手去触碰和放开的时间段完全可以采集出 x 和 y 的电压值。

X 和 Y 的坐标点如何计算呢？



这就是电阻屏控制芯片，我们单片机就采集 TSC2046 的 XY 座标数据就是了。

使用电阻触摸屏芯片 XPT2046 对触摸数据进行采集

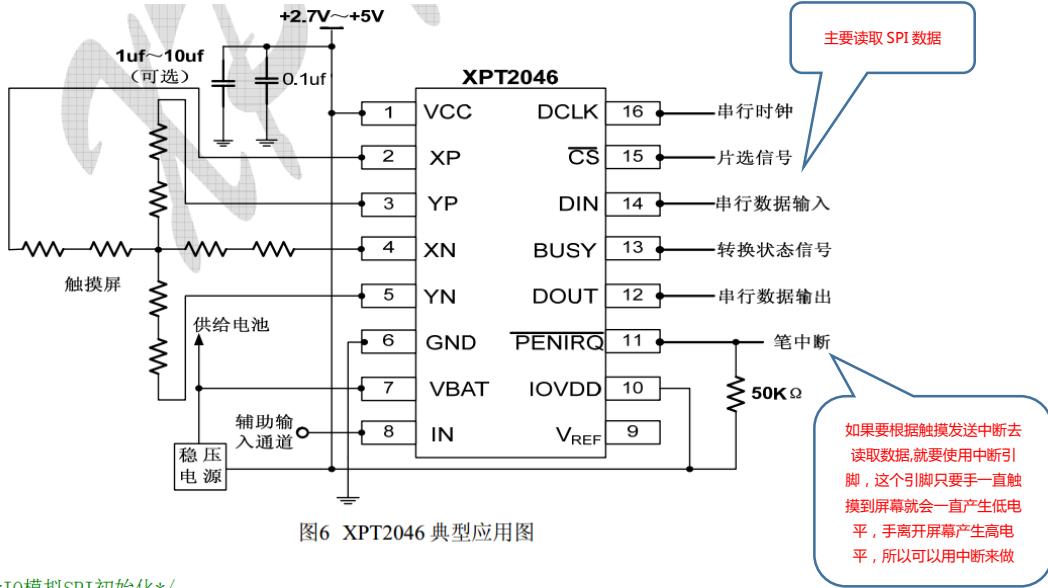


图6 XPT2046 典型应用图

```
/*IO模拟SPI初始化*/
void AnalogSpi_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC | RCC_APB2Periph_AFIO, ENABLE);

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_3|GPIO_Pin_0|GPIO_Pin_13;//初始PC0(SCLK),PC3(MOSI),PC13(CS)
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOC, &GPIO_InitStructure);

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_1|GPIO_Pin_2;//初始PC1(中断PEN),PC2(MISO)
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU ;
    GPIO_Init(GPIOC, &GPIO_InitStructure);
}

/*SPI写控制字*/
static void XPT2046_WriteCMD ( uint8_t ucCmd )
{
    uint8_t i;

    GPIO_ResetBits(GPIOC,GPIO_Pin_3); //MOSI=0
    GPIO_ResetBits (GPIOC,GPIO_Pin_0); //SCLK=0

    for ( i = 0; i < 8; i ++ ) {
        (( ucCmd >> ( 7 - i ) ) & 0x01) ? GPIO_SetBits(GPIOC,GPIO_Pin_3) : GPIO_ResetBits(GPIOC,GPIO_Pin_3);
        delay_us(5);
        GPIO_SetBits(GPIOC,GPIO_Pin_0); //SCLK=1
        delay_us(5);
        GPIO_ResetBits(GPIOC,GPIO_Pin_0); //SCLK=0
    }
}
```

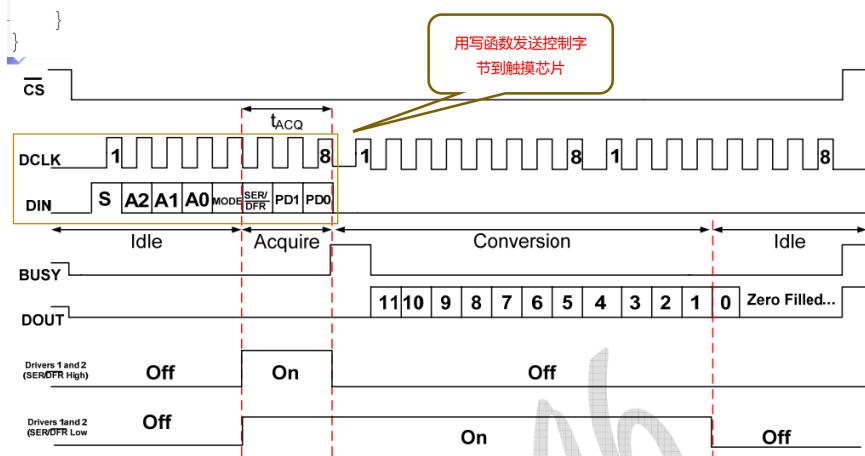


图14 8位总线接口, 无 DCLK 时钟延迟, 24时钟周期转换时序

表5 制字的控制位命令

位 7 (MSB)	位 6	位 5	位 4	位 3	位 2	位 1	位 0 (LSB)
S	A2	A1	A0	MODE	SER/DFR	PD1	PD0

S: 如果写 1 就是要求 xpt2046 开始转换触摸屏数据, 写 0 就不转换

A0~A2: 写入 001 就是 x+产生云墙电场, 检测触摸位置的 x 方向电压值。写入 101 就是 y+产生云墙电场, 采集触摸位置的 y 方向电压值, 所以我们都是轮流写 001, 101, 然后用 SPI去采集 x 值和 y 值。

MODE: 设置 xpt2046, ADC 分辨率采用 8 位还是 12 位, 我写 0 就是 12 位 ADC

SER/DFR: 就是写 1 设置触摸屏为单端输入, 写 0 触摸屏为差分输入, 我们写 0 差分输入

PD1~PD0: 写 00 就是省电模式, 不会给触摸屏产生电场

1001 0000(0x90) 检测 x 通道的电压值

0101 0000(0xd0) 检查 y 通道的电压值

```
/*SPI读数据, 读取的是12个位, 所以循环12次*/
static uint16_t XPT2046_ReadCMD(void)
{
    uint8_t i;
    uint16_t usBuf=0, usTemp;

    GPIO_ResetBits(GPIOC, GPIO_Pin_3); //MOSI=0
    GPIO_SetBits(GPIOC, GPIO_Pin_0); //SCLK=1

    for ( i=0; i<12; i++ ) {
        GPIO_ResetBits(GPIOC, GPIO_Pin_0); //SCLK=0
        usTemp = GPIO_ReadInputDataBit(GPIOC, GPIO_Pin_2); //读MISO电平
        usBuf |= usTemp << ( 11 - i );
        GPIO_SetBits(GPIOC, GPIO_Pin_0); //SCLK=1
    }

    return usBuf;
}
```

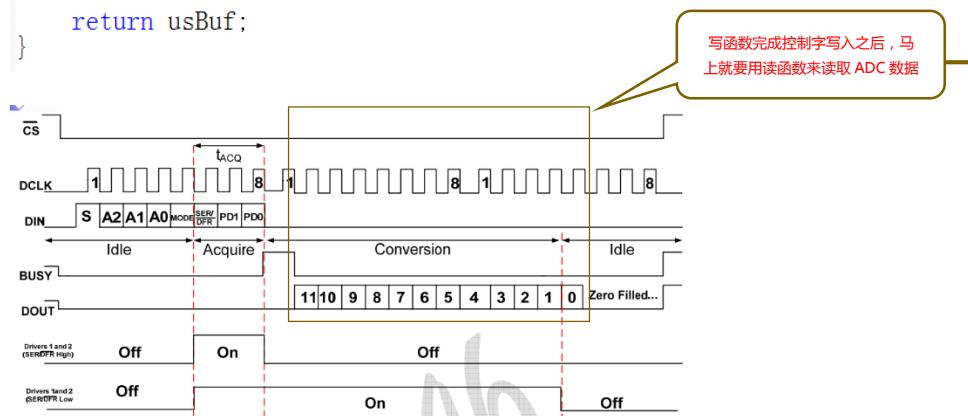


图14 8位总线接口, 无 DCLK 时钟延迟, 24 时钟周期转换时序

```
/*采集触摸屏原始数据
* ucChannel1 = 0x90 :通道 Y+的选择控制字
* ucChannel1 = 0xd0 :通道 X+的选择控制字
* return 该通道的 ADC 采样结果
*/
static uint16_t XPT2046_ReadAdc ( uint8_t ucChannel1 )
{
    XPT2046_WriteCMD ( ucChannel1 );
    return XPT2046_ReadCMD ();
}
```

所以写读一次, 就可以采集

一次触摸屏的电压值

```

/* 
 * 读取 XPT2046 的 X 通道和 Y 通道的 AD 值 ( 12 bit, 最大是 4096)
 * X_Ad : 存放 X 通道 AD 值的地址
 * Y_Ad : 存放 Y 通道 AD 值的地址
 */
static void XPT2046_ReadAdc_XY ( uint16_t * X_Ad, uint16_t * Y_Ad )
{
    uint16_t tempx = 0, tempy = 0;

    GPIO_ResetBits(GPIOC, GPIO_Pin_13); //CS=0 拉低片选
    delay_us(1);
    tempx = XPT2046_ReadAdc ( 0xd0 );//ReadAdc函数写了 0xd0 是x通道
    delay_us(1);
    tempy = XPT2046_ReadAdc ( 0x90 );//ReadAdc函数写了 0x90 是y通道
    GPIO_SetBits(GPIOC, GPIO_Pin_13); //CS=1 拉高片选, 如果你一直读触摸屏数据, 可以不用拉高

    *X_Ad = tempx;
    *Y_Ad = tempy;
}

int main(void)
{
    uint16_t Xvalue = 0, Yvalue = 0;

    RCC_Configuration(); //初始化时钟
    USART_Config(115200); //初始化串口
    AnalogSpi_Init(); //模拟SPI IO初始化
    delay_ms(3000);
    printf("xxxxxxxx\r\n");
    while(1)
    {
        XPT2046_ReadAdc_XY ( &Xvalue, &Yvalue );
        printf("X value = %d Y value = %d\r\n", Xvalue, Yvalue );
        delay_ms(1000);
    }
    return 0;
}

```

延时 1 秒让你数据看的更清楚，实际中触摸屏循环扫描是越快越好，最好无延时

然后进行触摸屏驱动基本测试

```

X value = 0 Y value = 4095
X value = 370 Y value = 444
X value = 389 Y value = 643
X value = 373 Y value = 607
X value = 357 Y value = 641
X value = 366 Y value = 608
X value = 347 Y value = 569
X value = 0 Y value = 4095
X value = 545 Y value = 3759
X value = 536 Y value = 3775
X value = 521 Y value = 3775
X value = 522 Y value = 3781
X value = 0 Y value = 4095
X value = 3279 Y value = 3781
X value = 3319 Y value = 3775
X value = 3325 Y value = 3775
X value = 3323 Y value = 3787
X value = 0 Y value = 4095
X value = 3295 Y value = 553
X value = 3307 Y value = 550
X value = 3311 Y value = 544
X value = 3319 Y value = 538
X value = 3387 Y value = 482
X value = 0 Y value = 4095

```

其实手按到触摸屏什么角我没有细看，我只是让你看到驱动运行成功了，有明显的触摸角度数据变化，但是我发现同一个角度按下出来的数据只是范围差不多，但是数据的精度还不够。

下面用滤波算法对触摸屏 x 和 y 的数字量进行重新计算

触摸屏数据读取后进行滤波运算

```
/*对获取的 X_Ad 和 Y_Ad 进行算法滤波得到真正的 X,Y 值
*x:真正滤波后的 x 值返回
*y:真正滤波后的 y 值返回
*/
void XPT2046_ReadAdc_Smooth_XY(uint16_t *X, uint16_t *Y)
{
    uint16_t sAD_X = 0, sAD_Y = 0;
    uint16_t xbuffer[10] = {0};
    uint16_t ybuffer[10] = {0};
    uint8_t count = 0;
    int i = 0;
    uint16_t Xmax = 0, Xmin = 0, Ymax = 0, Ymin = 0;

    while((GPIOC_ReadInputDataBit(GPIOC_GPIO_Pin_1) == 0) && count < 10)//如果触摸屏被按着， PEN 引脚输出低电平
    {
        XPT2046_ReadAdc_XY(&sAD_X, &sAD_Y); //采集 ADC X 和 Y 值

        xbuffer[count] = sAD_X;
        ybuffer[count] = sAD_Y;
        count++;
    }

    if(count > 8)
    {
        Xmax = Xmin = xbuffer[0];
        Ymax = Ymin = ybuffer[0];

        for(i = 1; i < 10; i++) //去除 x 的最大值和最小值然后给 xbuffer
        {
            if(xbuffer[i] < Xmin)
                Xmin = xbuffer[i];
            else
                Xmax = xbuffer[i];
        }

        for(i = 1; i < 10; i++) //去除 y 的最大值和最小值然后给 ybuffer
        {
            if(ybuffer[i] < Ymin)
                Ymin = ybuffer[i];
            else
                Ymax = ybuffer[i];
        }

        /*计算平均值滤波*/
        *X = xbuffer[0];
        *Y = ybuffer[0];
        for(i = 0; i < 10; i++)
        {

            *X = *X + xbuffer[i+1];
            *Y = *Y + ybuffer[i+1]; //这里千万别写错了,不然 4 点校准,5 点校准算法都会不准
        }

        *X = *X/10;
        *Y = *Y/10;
    }
}

int main(void)
{
    uint16_t Xvalue = 0, Yvalue = 0;

    RCC_Configuration(); //初始化时钟
    USART_Config(115200); //初始化串口
    AnalogSpi_Init(); //模拟SPI IO初始化
    delay_ms(3000);
    printf("xxxxxxxx\r\n");
    while(1)
    {
        XPT2046_ReadAdc_Smooth_XY( &Xvalue, &Yvalue );
        printf("X value = %d Y value = %d\r\n", Xvalue, Yvalue);
        delay_ms(1000);
    }
    return 0;
}
```

加入算法之后，数据输出最后两位就相对比较稳定

将 XPT2046_ReadAdc_XY
换成算法函数...Smooth_XY
就是了

X value = 3523	Y value = 3476
X value = 3531	Y value = 3483
X value = 3535	Y value = 3485
X value = 3531	Y value = 3481
X value = 3531	Y value = 3482
X value = 3531	Y value = 3482
X value = 3531	Y value = 3482
X value = 3531	Y value = 3482
X value = 3531	Y value = 3482
X value = 3531	Y value = 3482
X value = 3531	Y value = 3482

有了 X 和 Y 的数据，我们不能直接按比例运算把 XY 的数据和 LCD 屏幕的分辨率对应起来。因为 LCD 是 320x240 像素，你的触摸屏输出的原始值都是 4 位数以上的，所以我们要进行触摸屏校准。

（如触摸屏输出的原始数据范围为 0-2045，液晶屏的像素 XY 坐标为 0-239 及 0-319），那种直接转换的方式误差比较大，所以通常会采用“多点触摸校正法”来转换坐标，使用这种方式时，在应用前需要校正屏幕。

触摸屏校准算法

在校准之前，我们取消掉触摸屏 ADC 的滤波算法，直接读取 ADC 的值来看看触摸换算后坐标和 LCD 坐标是不是相对接近。

下面修改触摸屏坐标读取驱动函数

```
void XPT2046_ReadAdc_Smooth_XY(uint16_t *X , uint16_t *Y)
{
    uint16_t sAD_X = 0, sAD_Y = 0;
    uint16_t xbuffer[10] = {0};
    uint16_t ybuffer[10] = {0};
    uint8_t count = 0;
    int i = 0 ;

    uint16_t Xmax = 0, Xmin = 0, Ymax = 0, Ymin = 0;

    while((GPIO_ReadInputDataBit(GPIOC, GPIO_Pin_1) == 0) && count < 10) //如果触摸屏被按着, F
    {
        XPT2046_ReadAdc_XY(&sAD_X, &sAD_Y); //采集ADC X和Y值

        xbuffer[count] = sAD_X;
        ybuffer[count] = sAD_Y;

        *X = sAD_X;
        *Y = sAD_Y;
        return ;
        count++;
    }

    // if(count > 8)
    // {
    //     Xmax = Xmin = xbuffer[0];
    //     Ymax = Ymin = ybuffer[0];
    // }

    int main(void)
    {
        uint16_t Xvalue = 0, Yvalue = 0;
        int time = 0;

        int tx1 = 0 , ty1 = 0;
        int tx2 = 0 , ty2 = 0;
        int tx3 = 0 , ty3 = 0;
        int tx4 = 0 , ty4 = 0;
    }
}

RCC_Configuration(); //初始化时钟
USART_Config(115200); //初始化串口
LCD_Init(); //初始化LCD
LCD_DisplayOn();
AnalogSpi_Init(); //模拟SPI IO初始化
delay_ms(3000);
printf("xxxxzzzz\r\n");

LCD_Clear(WHITE);
```

主函数加入 LCD 函数

读取触摸屏原始的 ADC 值，然后返回给主程序

触摸屏校准算法取消

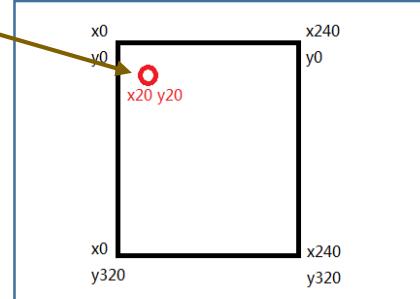
触摸屏触摸的四个角坐标值放在这些变量里面，我们现在只测试触摸屏一个点，看看触摸屏和 LCD 相对位置，所以我们只测量左上角的点

```

LCD_Fill(20, 20, 30, 30, RED);
delay_ms(2000);
for(time = 0;time<5;time++)
{
    XPT2046_ReadAdc_Smooth_XY( &Xvalue, &Yvalue );
    delay_ms(1000);
    printf("x1 = %d y1 = %d\r\n", Xvalue, Yvalue);
    if(Xvalue > 0)
    {
        tx1 = Xvalue;
        ty1 = Yvalue;
    }
}

```

在触摸屏左上角先画上一个(20 , 20)的点



```

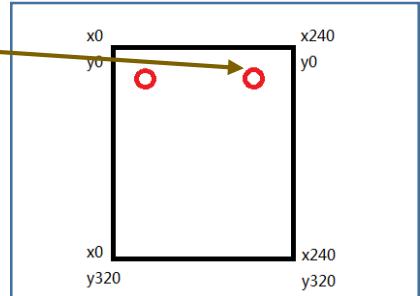
Xvalue = 0;
Yvalue = 0;

LCD_Fill(200, 20, 210, 30, RED);
delay_ms(2000);
for(time = 0;time<5;time++)
{
    XPT2046_ReadAdc_Smooth_XY( &Xvalue, &Yvalue );
    delay_ms(1000);
    printf("x2 = %d y2 = %d\r\n", Xvalue, Yvalue);

    if(Xvalue > 0)
    {
        tx2 = Xvalue;
        ty2 = Yvalue;
    }
}

```

在触摸屏右上角先画上一个(200 , 20)的点



```

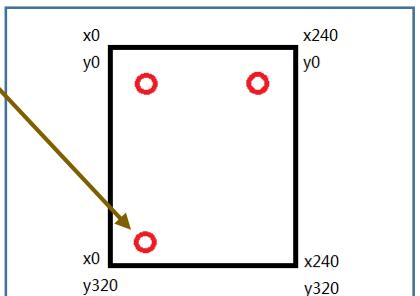
Xvalue = 0;
Yvalue = 0;

LCD_Fill(20, 300, 30, 310, RED);
delay_ms(2000);
for(time = 0;time<5;time++)
{
    XPT2046_ReadAdc_Smooth_XY( &Xvalue, &Yvalue );
    delay_ms(1000);
    printf("x3 = %d y3 = %d\r\n", Xvalue, Yvalue);

    if(Xvalue > 0)
    {
        tx3 = Xvalue;
        ty3 = Yvalue;
    }
}

```

在触摸屏左下角先画上一个(20 , 300)的点



```

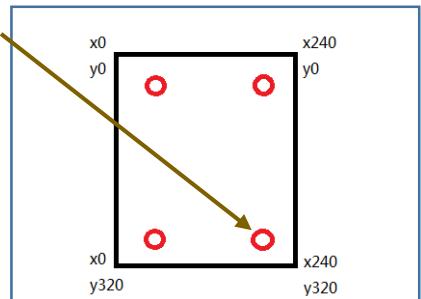
Xvalue = 0;
Yvalue = 0;

LCD_Fill(200, 300, 210, 310, RED);
delay_ms(2000);
for(time = 0;time<5;time++)
{
    XPT2046_ReadAdc_Smooth_XY( &Xvalue, &Yvalue );
    delay_ms(1000);
    printf("x4 = %d y4 = %d\r\n", Xvalue, Yvalue);

    if(Xvalue > 0)
    {
        tx4 = Xvalue;
        ty4 = Yvalue;
    }
}

```

在触摸屏右下角画上一个(200 , 300)的点



```

while(1)
{
    XPT2046_ReadAdc_Smooth_XY( &Xvalue, &Yvalue );
    printf("X value = %d Y value = %d\r\n", Xvalue, Yvalue);

    if(Xvalue>0)
    {
        X = 0;
        Y = 0;

        X = Xfac*Xvalue+Xoffset;
        Y = Yfac*Yvalue+Yoffset;
        X = (int)((Xvalue * 240)/4096);
        Y = (int)((Yvalue * 320)/4096);

        printf("X = %d Y = %d \r\n", X, Y);
        LCD_Fill(X, Y, X+10, Y+10, RED);
        Xvalue = 0;
        Yvalue = 0;
    }
}

X value = 586 Y value = 462
X = 34 Y = 36
X value = 588 Y value = 468
X = 34 Y = 36
X value = 588 Y value = 472
X = 34 Y = 36
X value = 588 Y value = 474
X = 34 Y = 37
X value = 587 Y value = 475
X = 34 Y = 37
X value = 588 Y value = 476
X = 34 Y = 37

```

在循环中直接读取没有经过滤波的触摸屏数据

触摸笔或手触碰的这个像素点的触摸数据直接/4096(触摸屏模拟分辨率)得到每个触点的电压,再乘上x方向的像素点240,得到实际LCD的x像素点坐标。我们看看这个坐标出来的数据,LCD的y也是这样算

x0 y0 x240 y0
x0 v320 x240 v320

发现触摸屏触摸的(20,20)位置转换成模拟值是(586,462),然后经过(模拟值*屏幕宽度或者高度)/4096 = (34, 36), 这个 LCD 的 x=34, y=36, 和我们设置的(20,20)坐标差十几个像素,我们可以用减法补偿。

```

while(1)
{
    XPT2046_ReadAdc_Smooth_XY( &Xvalue, &Yvalue );
    printf("X value = %d Y value = %d\r\n", Xvalue, Yvalue);

    if(Xvalue>0)
    {
        X = 0;
        Y = 0;

        X = Xfac*Xvalue+Xoffset;
        Y = Yfac*Yvalue+Yoffset;
        X = (int)((Xvalue * 240)/4096)-14;
        Y = (int)((Yvalue * 320)/4096)-16;

        printf("X = %d Y = %d \r\n", X, Y);
        LCD_Fill(X, Y, X+10, Y+10, RED);
        Xvalue = 0;
        Yvalue = 0;
    }
}

X value = 640 Y value = 512
X = 23 Y = 24
X value = 665 Y value = 514
X = 24 Y = 24
X value = 690 Y value = 526
X = 26 Y = 25
X value = 709 Y value = 538
X = 27 Y = 26

```

直接用减法补偿,简单粗暴

用减法补偿, 触摸(20,20)这个像素的位置得到改善, 但是还是有抖动没有完全到(20,20), 看来还是要对触摸屏底层的 ADC 读取 XY 坐标函数进行数字滤波算法。

XPT2046_ReadAdc_Smooth_XY(&Xvalue, &Yvalue);

对触摸屏读取函数加入滤波算法

```
void XPT2046_ReadAdc_Smooth_XY(uint16_t *X, uint16_t *Y)
{
    uint16_t sAD_X = 0, sAD_Y = 0;
    uint16_t xbuffer[10] = {0};
    uint16_t ybuffer[10] = {0};
    uint8_t count = 0;
    int i = 0;
    uint16_t Xmax = 0, Xmin = 0, Ymax = 0, Ymin = 0;

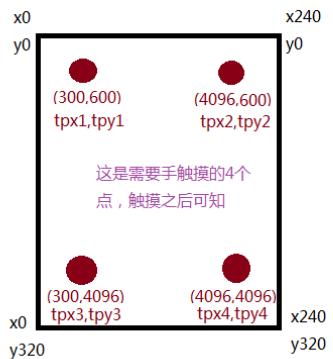
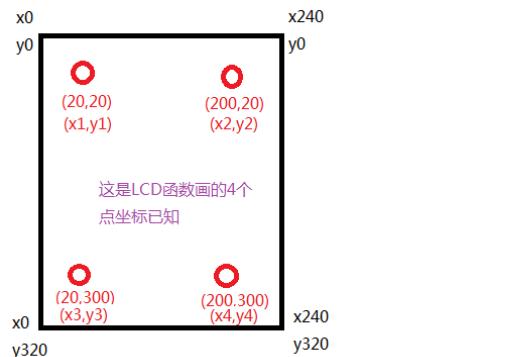
    while((GPIO_ReadInputDataBit(GPIOC, GPIO_Pin_1) == 0) && count < 10) //如果触摸屏被按着,
    {
        XPT2046_ReadAdc_XY(&sAD_X, &sAD_Y); //采集ADC X和Y值

        xbuffer[count] = sAD_X;
        ybuffer[count] = sAD_Y;
        count++;
    }
    if(count > 9)
    {
        Xmax = Xmin = xbuffer[0];
        Ymax = Ymin = ybuffer[0];

        for(i = 1; i < 10; i++) //去除x的最大值和最小值然后给xbuffer
        {
            if(xbuffer[i] < Xmin)
                Xmin = xbuffer[i];
            else
                Xmax = xbuffer[i];
        }
    }
}
```

这个取消最大值最小值,求平均数算法前面有,这里就不再全部贴出了,这种直接读取触摸屏数据在 LCD 打坐标点是为了检验触摸屏底层算法对不对。

下面进行四点校准算法



$$\text{用LCD自己设置的} x_1, x_2 \text{ 坐标 } x_{\text{facbase}} = \frac{(x_2 - x_1) + (x_2 - x_1)}{2.0}$$

$$y \text{ 方向也就是触摸屏高度同理 } y_{\text{facbase}} = \frac{(y_3 - y_1) + (y_3 - y_1)}{2.0}$$

$$\text{算出手触摸屏幕后的横坐标长度 } x_{\text{tp}} = \frac{(tpx2 - tpx1) + (tpx4 - tpx3)}{2.0}$$

$$y_{\text{tp}} = \frac{(tpy3 - tpy1) + (tpy4 - tpy2)}{2.0}$$

摸,所以触摸屏上边x

长度和下边x长度有点

出入

$$\text{计算出了} x \text{ 的缩放比例 } x_{\text{fac}} = \frac{x_{\text{facbase}}}{x_{\text{tp}}}$$

$$y_{\text{fac}} = \frac{y_{\text{facbase}}}{y_{\text{tp}}}$$

现在计算出来的 xfac 和 yfac 就是伸缩系数,有这两个系数然后去乘以读取出来的触摸屏 xy 值就可以在 LCD 上面画点了,只是点的偏差有点大

这是因为将整个 LCD 屏幕的宽度偏移计算进来,只计算了触摸点和自己设置 LCD 4 点的对比系数

$$x_{\text{offset}} = \frac{(\text{屏幕宽度} - x_{\text{fac}} \times (tpx2 + tpx1))}{2.0}$$

$$y_{\text{offset}} = \frac{(\text{屏幕高度} - y_{\text{fac}} \times (tpy3 + tpy1))}{2.0}$$

LCD 显示 $x = x_{\text{fac}} * \text{获取触摸屏的当前} x \text{ 值} + x_{\text{offset}}$

LCD 显示 $y = y_{\text{fac}} * \text{获取触摸屏的当前} y \text{ 值} + y_{\text{offset}}$

这样手触摸触摸屏某个点,LCD 就会用 x 和 y 去显示对应的点。

```

int main(void)
{
    uint16_t Xvalue = 0, Yvalue = 0;

    int time = 0;

    int tx1 = 0, ty1 = 0;
    int tx2 = 0, ty2 = 0;
    int tx3 = 0, ty3 = 0;
    int tx4 = 0, ty4 = 0;

    int centreX = 0, centreY = 0;
    float Xfac = 0, Yfac = 0, Xoffset = 0, Yoffset = 0, Xfacbase = 0, Yfacbase = 0;
    int X = 0, Y = 0;

    RCC_Configuration(); // 初始化时钟
    USART_Config(115200); // 初始化串口
    LCD_Init(); // 初始化LCD
    LCD_DisplayOn();
    AnalogSpi_Init(); // 模拟SPI IO初始化
    delay_ms(3000);
    printf("xxxxxxxx\r\n");

    LCD_Clear(WHITE);

```

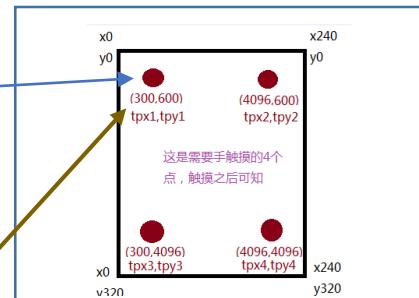
```

LCD_Fill(20, 20, 30, 30, RED);
delay_ms(2000);
for(time = 0;time<5;time++)
{
    XPT2046_ReadAdc_Smooth_XY( &Xvalue, &Yvalue );
    delay_ms(1000);
    printf("x1 = %d y1 = %d\r\n", Xvalue, Yvalue);
    if(Xvalue > 0)
    {
        tx1 = Xvalue;
        ty1 = Yvalue;
    }
}
Xvalue = 0;
Yvalue = 0;

LCD_Fill(200, 20, 210, 30, RED);
delay_ms(2000);
for(time = 0;time<5;time++)
{
    XPT2046_ReadAdc_Smooth_XY( &Xvalue, &Yvalue );
    delay_ms(1000);
    printf("x2 = %d y2 = %d\r\n", Xvalue, Yvalue);

    if(Xvalue > 0)
    {
        tx2 = Xvalue;
        ty2 = Yvalue;
    }
}

```



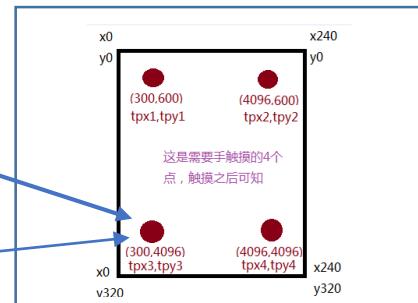
```

Xvalue = 0;
Yvalue = 0;

LCD_Fill(20, 300, 30, 310, RED);
delay_ms(2000);
for(time = 0;time<5;time++)
{
    XPT2046_ReadAdc_Smooth_XY( &Xvalue, &Yvalue );
    delay_ms(1000);
    printf("x3 = %d y3 = %d\r\n", Xvalue, Yvalue);

    if(Xvalue > 0)
    {
        tx3 = Xvalue;
        ty3 = Yvalue;
    }
}

```



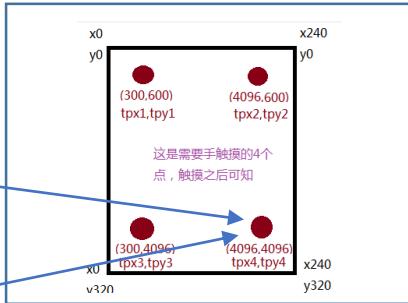
```

Xvalue = 0;
Yvalue = 0;

LCD_Fill(200, 300, 210, 310, RED);
delay_ms(2000);
for(time = 0;time<5;time++)
{
    XPT2046_ReadAdc_Smooth_XY( &Xvalue, &Yvalue );
    delay_ms(1000);
    printf("x4 = %d y4 = %d\r\n", Xvalue, Yvalue);

    if(Xvalue > 0)
    {
        tx4 = Xvalue;
        ty4 = Yvalue;
    }
}

```



现在获取到了 tpx1, tpy1, tpx2, tpy2, tpx3, tpy3, tpx4, tpy4

```

Xfacbase = ((float)((200-20)+(200-20))/2.0;
Xfac = Xfacbase/((float)((tx2-tx1)+(tx4-tx3))/2.0);
Yfacbase = ((float)((300-20)+(300-20))/2.0;
Yfac = Yfacbase/((float)((ty3-ty1)+(ty4-ty2))/2.0);
printf("Xfac = %f Yfac = %f \r\n", Xfac, Yfac);

Xoffset = ((float)((240-Xfac*(tx2+tx1))/2);
Yoffset = ((float)((320-Yfac*(ty3+ty1))/2));
printf("Xoffset = %f Yoffset = %f \r\n", Xoffset, Yoffset);

while(1)
{
    XPT2046_ReadAdc_Smooth_XY( &Xvalue, &Yvalue );
    printf("X value = %d Y value = %d\r\n", Xvalue, Yvalue);

    if(Xvalue>0)
    {
        X = 0;
        Y = 0;

        X = Xfac*Xvalue+Xoffset;
        Y = Yfac*Yvalue+Yoffset;
        printf("X = %d Y = %d \r\n", X, Y);
        LCD_Fill(X, Y, X+10, Y+10, RED);
        Xvalue = 0;
        Yvalue = 0;
    }
    delay_ms(100);
}
return 0;

```

$$X_{facbase} = \frac{(x_2 - x_1) + (x_4 - x_3)}{2.0}$$

$$x_{tp} = \frac{(tpx_2 - tpx_1) + (tpx_4 - tpx_3)}{2.0}$$

$$x_{fac} = \frac{x_{facbase}}{x_{tp}}$$

$$y_{facbase} = \frac{(y_3 - y_1) + (y_4 - y_2)}{2.0}$$

$$y_{tp} = \frac{(tpy_3 - tpy_1) + (tpy_4 - tpy_2)}{2.0}$$

$$y_{fac} = \frac{y_{facbase}}{y_{tp}}$$

$$X_{offset} = \frac{(\text{屏幕宽度} - x_{fac} \times (tpx_2 + tpx_1))}{2.0}$$

$$Y_{offset} = \frac{(\text{屏幕高度} - y_{fac} \times (tpy_3 + tpy_1))}{2.0}$$

LCD显示 x = x_{fac} * 获得触摸屏的当前x值 + Xoffset
LCD显示 y = y_{fac} * 获得触摸屏的当前y值 + Yoffset

你可以用手在屏幕上画画了

源码贴出

```
int main(void)
{
    uint16_t Xvalue = 0, Yvalue = 0;

    int time = 0;

    int tx1 = 0 , ty1 = 0;
    int tx2 = 0 , ty2 = 0;
    int tx3 = 0 , ty3 = 0;
    int tx4 = 0 , ty4 = 0;

    int centreX = 0, centreY = 0;
    float Xfac = 0,Yfac = 0,Xoffset = 0, Yoffset = 0,Xfacbase = 0, Yfacbase = 0;
    int X = 0, Y = 0;

    RCC_Configuration(); // 初始化时钟
    USART_Config(115200); // 初始化串口
    LCD_Init(); // 初始化 LCD
    LCD_DisplayOn();
    AnalogSpi_Init(); // 模拟 SPI IO 初始化
    delay_ms(3000);
    printf("xxxxxxxx\r\n");

    LCD_Clear(WHITE);

    LCD_Fill(20,20,30,RED);
    delay_ms(2000);
    for(time = 0;time<5;time++)
    {
        XPT2046_ReadAdc_Smooth_XY( &Xvalue, &Yvalue );
        delay_ms(1000);
        printf("x1 = %d y1 = %d\r\n",Xvalue,Yvalue);
        if(Xvalue > 0)
        {
            tx1 = Xvalue;
            ty1 = Yvalue;
        }
    }

    Xvalue = 0;
    Yvalue = 0;

    LCD_Fill(200,20,210,30,RED);
    delay_ms(2000);
    for(time = 0;time<5;time++)
    {
        XPT2046_ReadAdc_Smooth_XY( &Xvalue, &Yvalue );
        delay_ms(1000);
        printf("x2 = %d y2 = %d\r\n",Xvalue,Yvalue);

        if(Xvalue > 0)
        {
            tx2 = Xvalue;
            ty2 = Yvalue;
        }
    }

    Xvalue = 0;
    Yvalue = 0;

    LCD_Fill(20,300,30,310,RED);
    delay_ms(2000);
    for(time = 0;time<5;time++)
    {
        XPT2046_ReadAdc_Smooth_XY( &Xvalue, &Yvalue );
        delay_ms(1000);
        printf("x3 = %d y3 = %d\r\n",Xvalue,Yvalue);

        if(Xvalue > 0)
        {
            tx3 = Xvalue;
            ty3 = Yvalue;
        }
    }
}
```

```

        }

    }

Xvalue = 0;
Yvalue = 0;
LCD_Fill(200,300,210,310,RED);
delay_ms(2000);
for(time = 0;time<5;time++)
{
    XPT2046_ReadAdc_Smooth_XY( &Xvalue, &Yvalue );
    delay_ms(1000);
    printf("x4 = %d y4 = %d\r\n",Xvalue,Yvalue);

    if(Xvalue > 0)
    {
        tx4 = Xvalue;
        ty4 = Yvalue;
    }
}
Xvalue = 0;
Yvalue = 0;
// LCD_Fill(100,150,110,160,RED);
// delay_ms(2000);
// for(time = 0;time<5;time++)
// {
//     XPT2046_ReadAdc_Smooth_XY( &Xvalue, &Yvalue );
//     delay_ms(1000);
//     printf("centreX = %d centreY = %d\r\n",Xvalue,Yvalue);
//
//     if(Xvalue > 0)
//     {
//         centreX = Xvalue;
//         centreY = Yvalue;
//     }
// }
//
// Xvalue = 0;
// Yvalue = 0;

Xfacbase = (float)((200-20)+(200-20))/2.0;
Xfac = Xfacbase/(float)((tx2-tx1) + (tx4-tx3))/2.0;

Yfacbase = (float)((300-20)+(300-20))/2.0;
Yfac = Yfacbase/(float)((ty3-ty1)+(ty4-ty2))/2.0;

printf("Xfac = %f  Yfac = %f \r\n",Xfac,Yfac);

Xoffset = (float)((240-Xfac*(tx2+tx1))/2);
Yoffset = (float)((320-Yfac*(ty3+ty1))/2);

printf("Xoffset = %f  Yoffset = %f \r\n",Xoffset,Yoffset);

while(1)
{

    XPT2046_ReadAdc_Smooth_XY( &Xvalue, &Yvalue );
    printf("X value = %d Y value = %d\r\n",Xvalue,Yvalue);

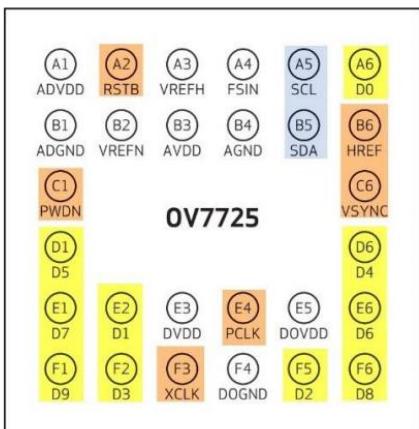
    if(Xvalue>0)
    {
        X = 0;
        Y = 0;

        X = Xfac*Xvalue+Xoffset;
        Y = Yfac*Yvalue+Yoffset;

        printf("X = %d Y = %d \r\n",X,Y);
        LCD_Fill(X,Y,X+10,Y+10,RED);
        Xvalue = 0;
        Yvalue = 0;
    }
    delay_ms(100);
}
return 0;
}

```

STM32 CMOS 摄像头驱动



这些 BGA 球就是
数据控制传输的引脚

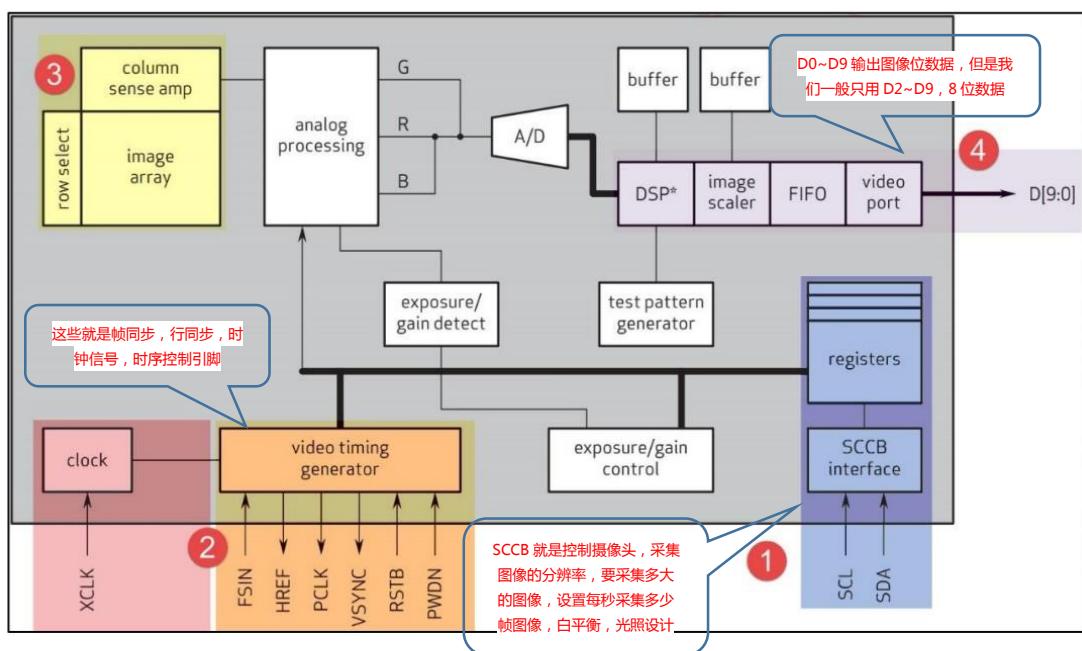
OV7725 图像传感器，我们就是操作这个摄像头

管脚名称	管脚类型	管脚描述
RSTB	输入	系统复位管脚，低电平有效
PWDN	输入	掉电/省电模式(高电平有效)
HREF	输出	行同步信号
VSYNC	输出	场同步信号
PCLK	输出	像素时钟
XCLK	输入	系统时钟输入端口
SCL	输入	SCCB 总线的时钟线
SDA	I/O	SCCB 总线的数据线
D0…D9	输出	像素数据端口

这里和 LCD 逻辑
很像

摄像头输出的颜色格式支持:YUV(422/420),YCbCr422,RGB565。这些格式的选择决定了你用什么 LCD 显示。所以摄像头和 LCD 某方面来说有些参数还需要配套。

我们用 RGB565 格式(红色 5 位, 绿色 6 位, 蓝色 5 位)。16 位数据组成一个像素点



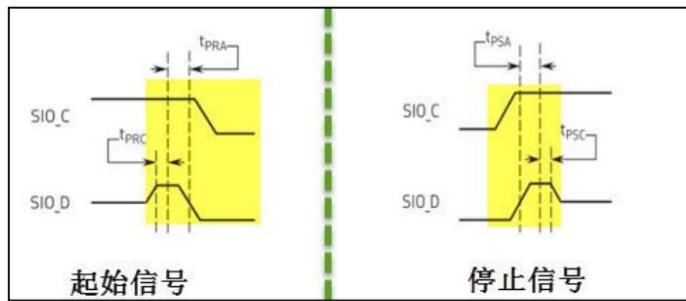


图 51-4 SCCB 停止信号

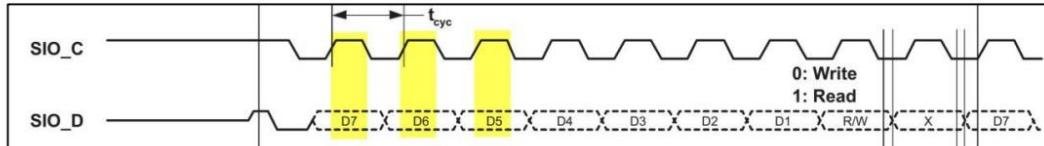


图 51-5 SCCB 的数据有效性

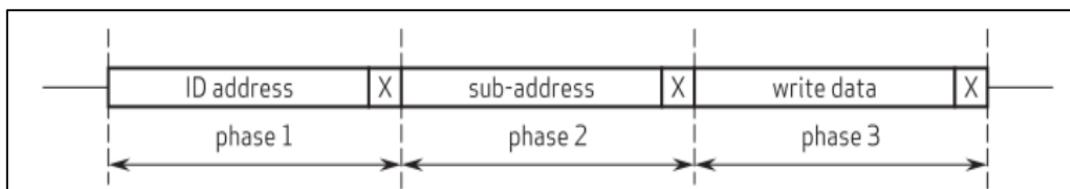


图 51-6 SCCB 的三步写操作

SCCB 写寄存器就是，发送器件地址+发送寄存器地址+向寄存器写数据，然后停止
I2C 写寄存器是，发送器件地址+发送寄存器地址+向寄存器写数据+还可以继续写数据+寄存器自动加 1+继续写数据.....，然后停止

这就是 SCCB 和 I2C 的区别，I2C 可以连续写，连续读，但是 SCCB 每次 start 到 stop 只能读或者写一次。

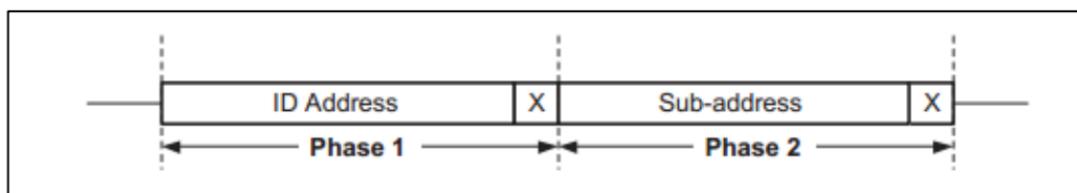


图 51-7 SCCB 的两步写操作

SCCB 读数据就是发送设备地址+发送要读取的寄存器地址+停止，然后还要从新发一次

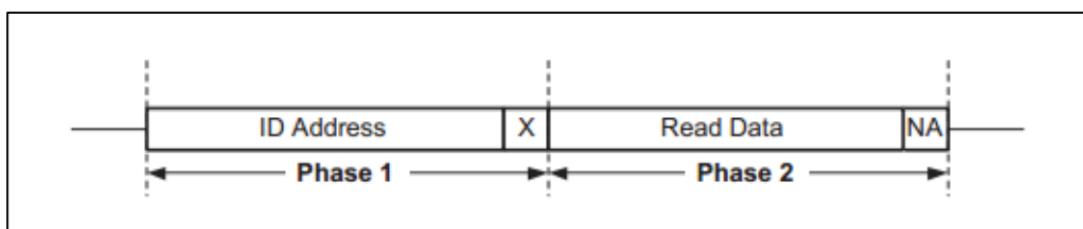


图 51-8 SCCB 的两步读操作

发送器件地址+读取 51-7 要求的寄存器里面的数据+NA 信号

所以两步读操作，它用于读取从设备目的寄存器中的数据，见图 51-8。在第一阶段中发送从设备的设备 ID+R 标志(设备地址+读方向标志)和自由位，在第二阶段中读取寄存器中的 8 位数据和写 NA 位(非应答信号)。由于两步读操作没有确定目的寄存器的地址，所以在读操作前，必需有一个两步写操作，以提供读操作中的寄存器地址。

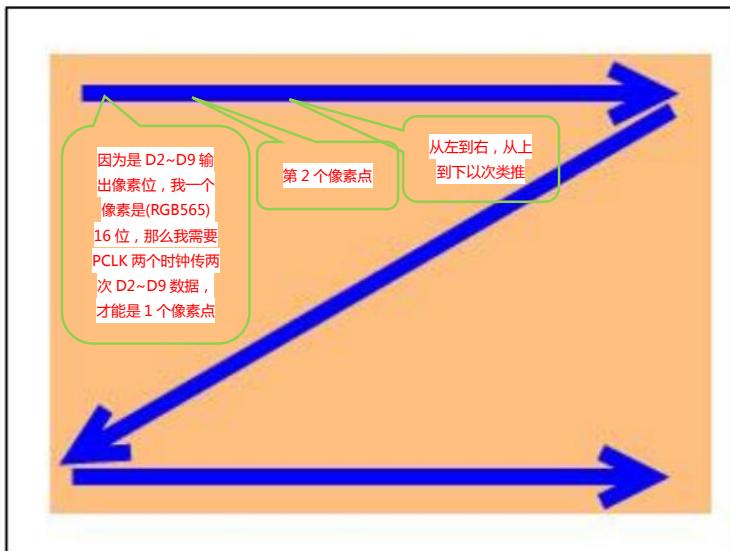


图 51-11 摄像头数据输出

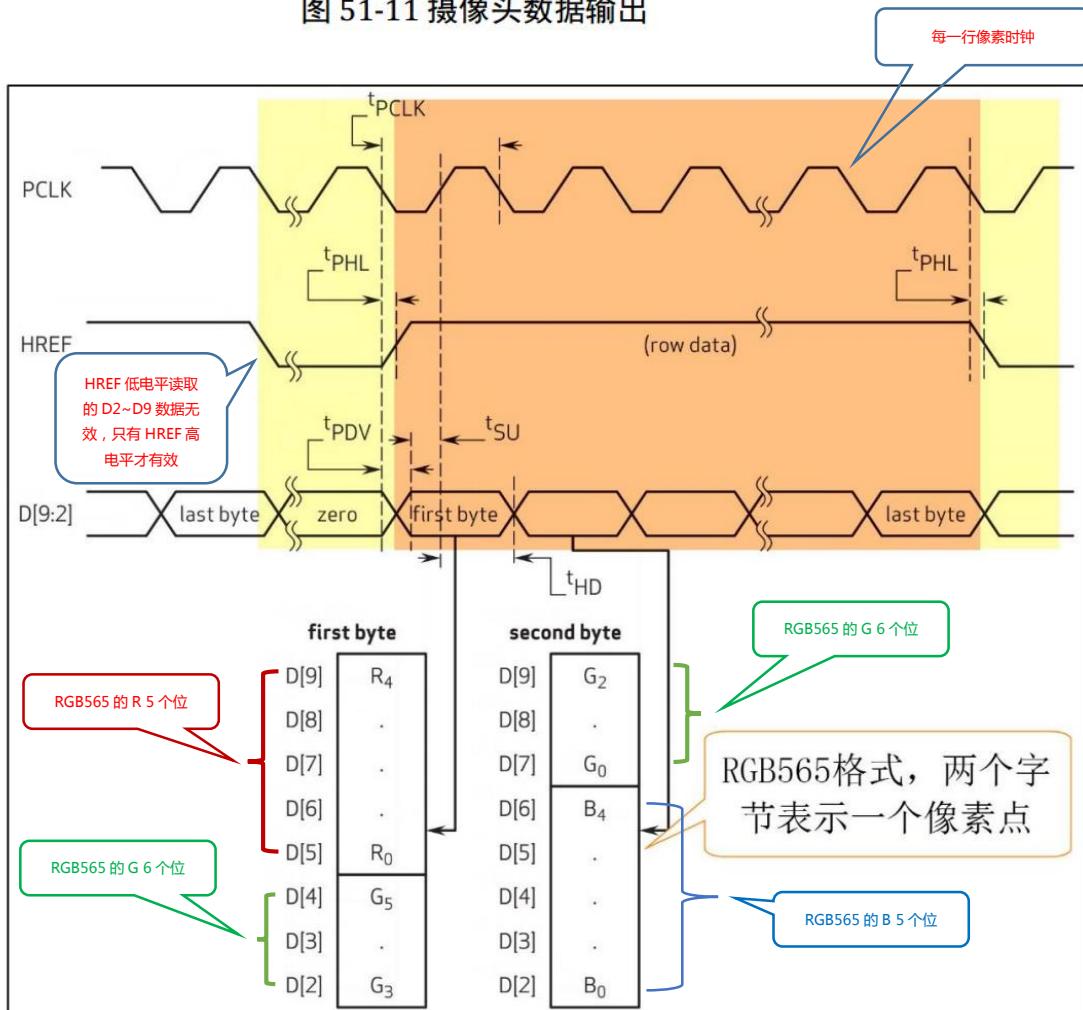


图 51-12 VGA 像素同步时序

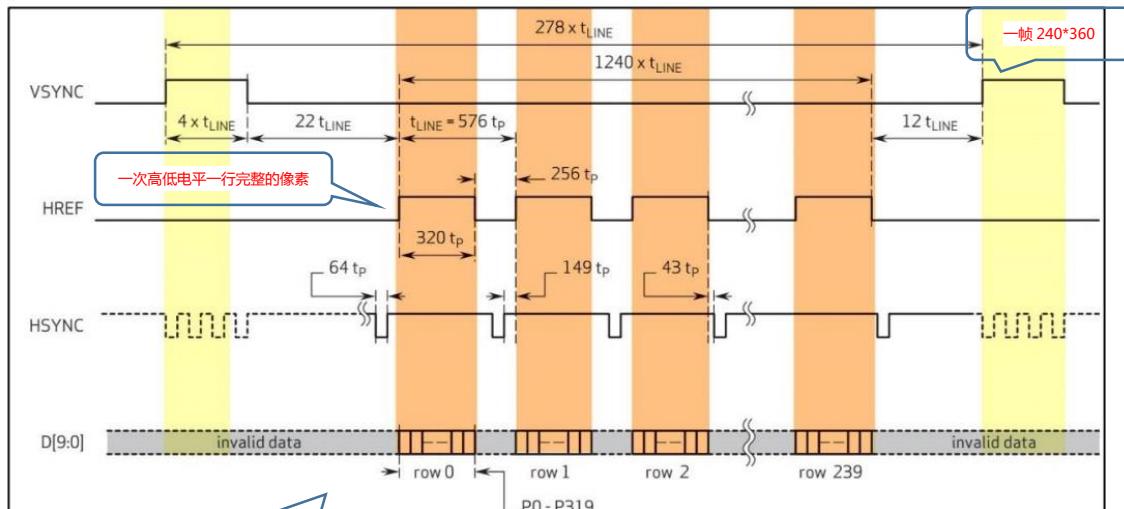


图 51-13 QVGA 帧图像同步时序

OV7725 摄像头 QVGA 模式输出的图像分辨率
为 240*320 个像素，我手上现有的 LCD 是
240*320，所以我 SCCB 设置 QVGA 输出

OV7725 摄像头 VGA 模式输出的图像分辨率
为 480*640 个像素，如果你有 480*640
的 LCD 倒是无所谓

其实摄像头像素时钟传输格式和 LCD 有点类似。

FIFO 有什么用

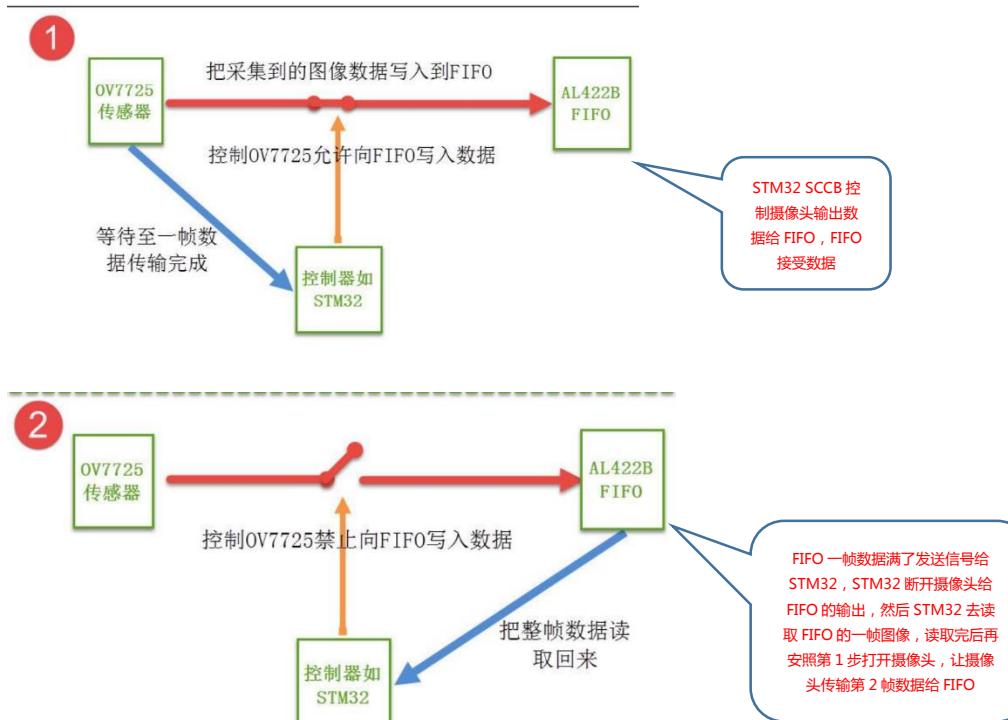
STM32F1 系列没有 DCMI 摄像头接口，所以缓存也不足，就需要加 FIFO 中转芯片

STM32F4 系列，有 DCMI 摄像头接口，而且还外扩了 SRAM 这些东西，所以不需要 FIFO

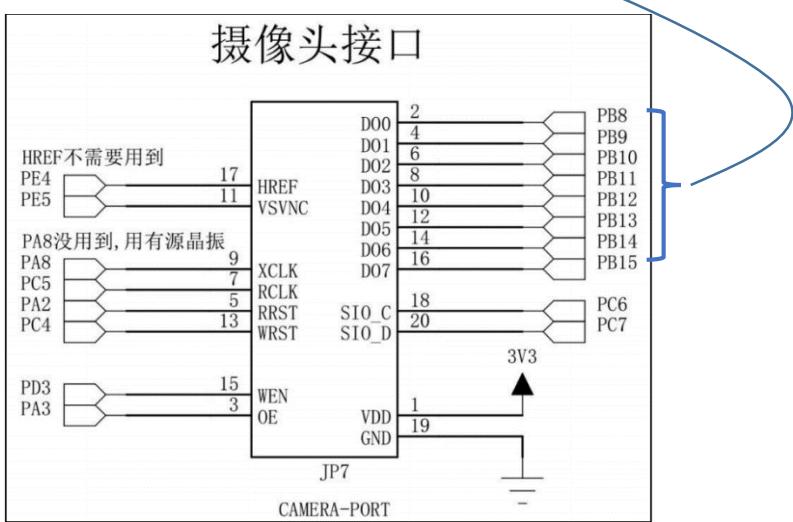
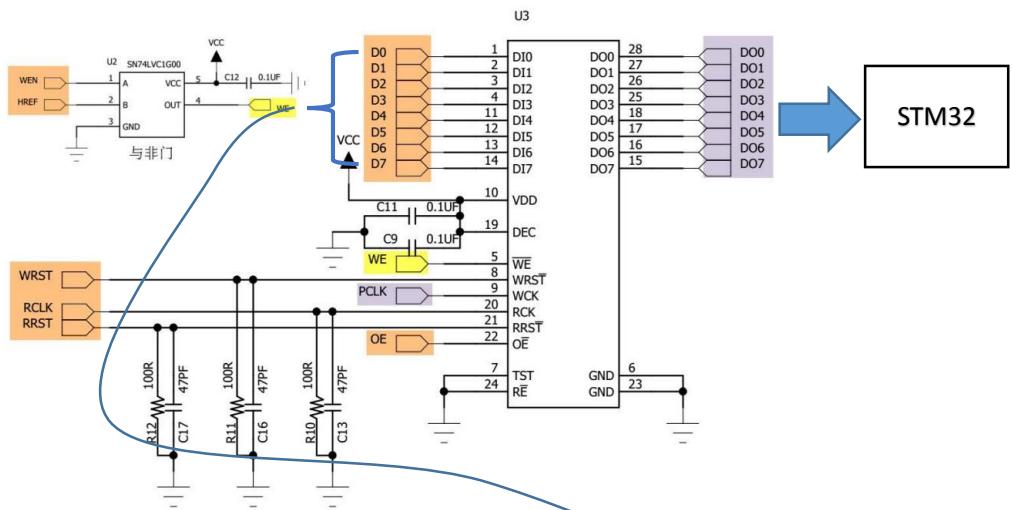
如果摄像头采集和 LCD 分辨率一样大的图像，比如 $240*320*2$ (每个像素 2 字节) $=153600$ 字节(150KB)

然而 STM32F1 系列很多都是 64KB 的 SRAM，所以连一帧的图片数据都无法存储。

所以用 AL422B 这个型号的 FIFO，容量为 393216 字节(390KB)，就可以缓存 2 帧图像。

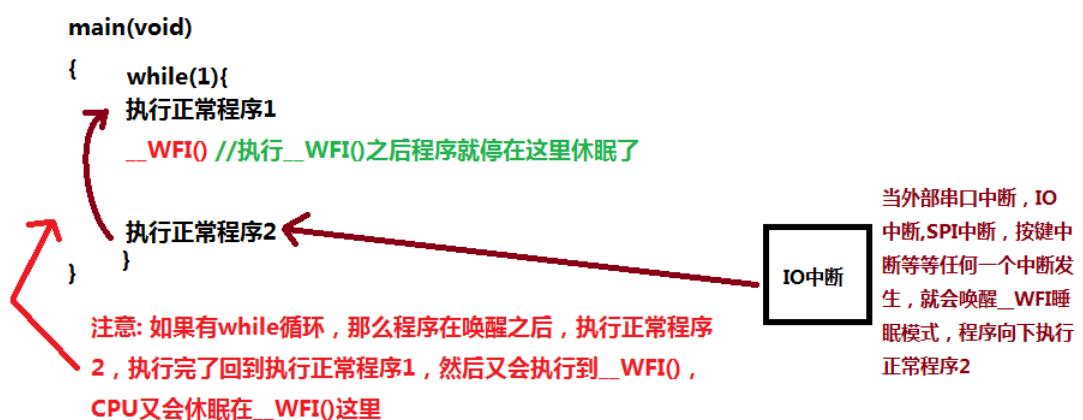
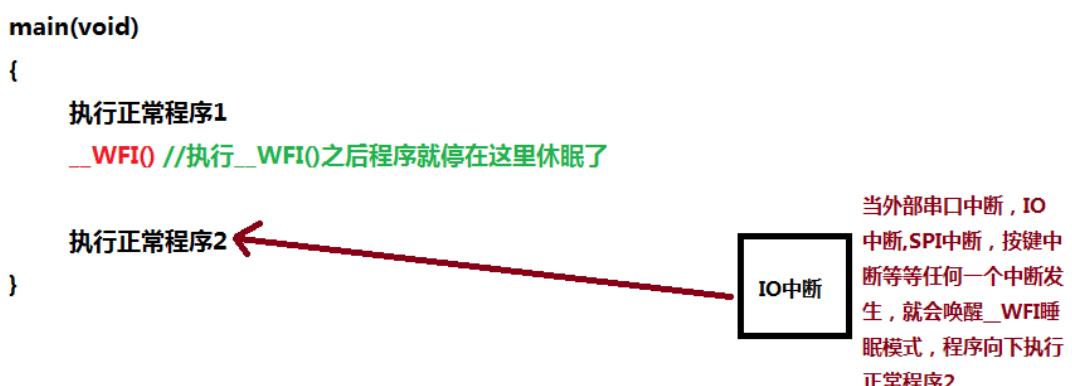
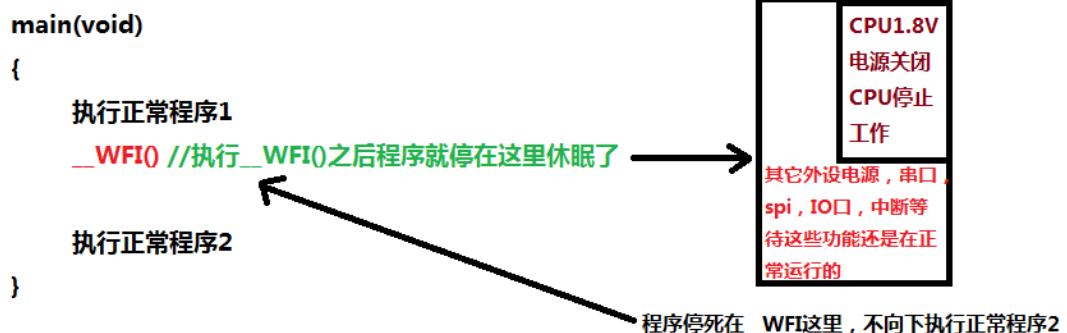


这里面就会存在 STM32 在读取 FIFO 数据的时候，摄像头后面的几帧数据会丢失。



STM32F1 电源低功耗模式

1. 睡眠模式操作



所以这时候 STM32 的复位按键就很重要了，因为上电 STM32 执行到_WFI()这里卡死休眠之后，你是无法向 STM32 下载程序的，只有借助长按复位按键的同时，点击下载新的程序(没有_WFI())才可以正常工作。



使用 PB8 按钮来唤醒休眠

```

void key_gpio_interrupt_init(void) //PB8 IO口外部输入中断, 唤醒__WFI
{
    GPIO_InitTypeDef GPIO_initStructure;
    EXTI_InitTypeDef EXTI_initStructure;
    NVIC_InitTypeDef NVIC_initStructure;

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE);

    GPIO_initStructure.GPIO_Pin = GPIO_Pin_8;
    GPIO_initStructure.GPIO_Mode = GPIO_Mode_IPU;
    GPIO_Init(GPIOB, &GPIO_initStructure);

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO, ENABLE);

    EXTI_ClearITPendingBit(EXTI_Line8);

    EXTI_initStructure.EXTI_Line = EXTI_Line8;
    EXTI_initStructure.EXTI_Mode = EXTI_Mode_Interrupt;
    EXTI_initStructure.EXTI_Trigger = EXTI_Trigger_Falling;
    EXTI_initStructure.EXTI_LineCmd = ENABLE;
    EXTI_Init(&EXTI_initStructure);

    GPIO_EXTILineConfig(GPIO_PortSourceGPIOB, GPIO_PinSource8);

    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);

    NVIC_initStructure.NVIC IRQChannel = EXTI9_5_IRQn;
    NVIC_initStructure.NVIC IRQChannelPreemptionPriority = 1;
    NVIC_initStructure.NVIC IRQChannelSubPriority = 5;
    NVIC_Init(&NVIC_initStructure);
}

```

初始化 PB8 按钮

```

void EXTI9_5_IRQHandler(void)
{
    EXTI_ClearFlag(EXTI_Line8);
    EXTI_ClearITPendingBit(EXTI_Line8);
}

```

按键中断函数只做需要清除标志位就是了

```

int main(void)
{
    RCC_configuration(); // 初始化时钟
    USART_config(115200); // 初始化串口
    key_gpio_interrupt_init(); // PB8 IO口外部输入中断, 唤醒__WFI
    delay_ms(5000);

    while(1)
    {
        printf("process 1111.... \r\n");
        delay_ms(3000);
        __WFI();
    }
    printf("process 2222.... \r\n");
    delay_ms(3000);
}

return 0;
}

```

The diagram illustrates the execution flow of the main loop. It shows two terminal windows. The first window shows the output "process 1111...." followed by a cursor. A callout bubble points to the __WFI() line in the code with the text: "执行第一句之后, 遇到__WFI(), STM32 休眠了, 程序就停在这里". The second window shows the output "process 1111....", "process 2222....", and "process 1111...." again, indicating that the program has been woken up and continues to execute the loop after the __WFI() call.

所以这个__WFI()语句可以放在一个固定的地方用来做事件触发休眠

2. 停止模式操作

```
main(void)
{
    执行程序111
    PWR_EnterSTOPMode(PWR_Regulator_ON,PWR_STOPEntry_WFI); →
    执行程序222
}
```

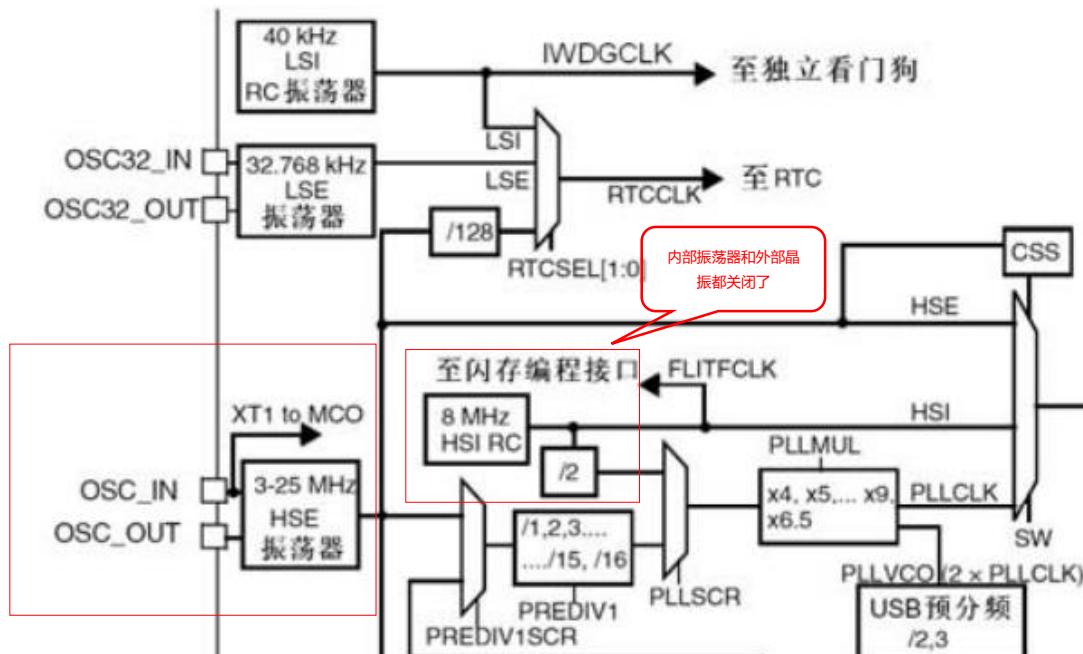
停止模式执行后程序卡在这里，和睡眠模式一样



```
main(void)
{
    执行程序111
    PWR_EnterSTOPMode(PWR_Regulator_ON,PWR_STOPEntry_WFI); →
    执行程序222 ←
}
```



但是注意，就算唤醒了，只是内部程序在跑，但是外部的 HSE,LSE 时钟是没有唤醒的。



```

int main(void)
{
    RCC_Configuration(); // 初始化时钟
    USART_Config(115200); // 初始化串口
    Key_GPIO_Init(); // PB8 IO口外部输入中断，唤醒_WFI
    delay_ms(5000);

    while(1)
    {
        printf("process 1111.... \r\n");
        delay_ms(3000);

        PWR_EnterSTOPMode(PWR_Regulator_ON, PWR_STOPEntry_WFI); // 程序停止在这里休眠
    }

    return 0;
}

```

PWR_EnterSTOPMode(PWR_Regulator_ON, PWR_STOPEntry_WFI); // 程序停止在这里休眠

```

int main(void)
{
    RCC_Configuration(); // 初始化时钟
    USART_Config(115200); // 初始化串口
    Key_GPIO_Init(); // PB8 IO口外部输入中断，唤醒_WFI
    delay_ms(5000);

    while(1)
    {
        printf("process 1111.... \r\n");
        delay_ms(3000);

        PWR_EnterSTOPMode(PWR_Regulator_ON, PWR_STOPEntry_WFI); // 程序停止在这里休眠
    }

    RCC_Configuration(); // 初始化时钟
    printf("process 2222.... \r\n");
    delay_ms(3000);
}

return 0;
}

```

这就是停止模式，程序唤醒后，接着停止位置程序段继续向下执行
所以前面的睡眠模式和这里的停止模式，都是可以断点标记后休眠的，只是停止模式会关闭外设时钟，从而更省电。

3.待机模式

```

main(void)
{
    /* 使能电源管理单元的时钟，必须要使能时钟才能进入待机模式 */
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_PWR, ENABLE);
    执行程序111
    /* 清除 WU 状态位 */
    PWR_ClearFlag(PWR_FLAG_WU);
    /* 使能 WKUP 引脚的唤醒功能，使能 PA0 */
    PWR_WakeUpPinCmd(ENABLE);
    /* 进入待机模式 */
    PWR_EnterSTANDBYMode();
    .....
    执行程序222
}

```

只有带备用电池的
RTC在工作

```

main(void)
{
    /* 使能电源管理单元的时钟,必须要使能时钟才能进入待机模式 */
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_PWR,ENABLE);

执行程序111
    /*清除 WU 状态位*/
    PWR_ClearFlag (PWR_FLAG_WU);
    /* 使能 WKUP 引脚的唤醒功能 , 使能 PA0*/
    PWR_WakeUpPinCmd(ENABLE);
    /* 进入待机模式 */
    PWR_EnterSTANDBYMode();
    .....
执行程序222      唤醒之后不会接着执行程序
                    222 , 而是复位整个STM32
                    重新执行程序
}

```

唤醒方式4种: RTC闹钟上升沿
 WKUP引脚上升沿
 NRST引脚复位
 IWDG看门狗复位

待机功耗 10uA

```

void key_gpio_interrupt_init(void); //PB8 IO口外部输入中断, 唤醒__WFI
int main(void)
{
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_PWR, ENABLE); /* 使能电源管理单元的时钟,必须要使能时钟才能进入待机模式 */
    RCC_configuration(); //初始化时钟
    USART_config(115200); //初始化串口
    key_gpio_interrupt_init(); //PB8 IO口外部输入中断, 唤醒__WFI
    delay_ms(5000);

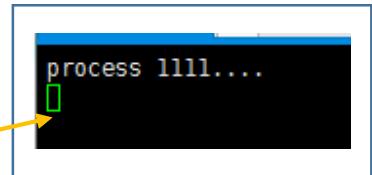
    while(1)
    {
        printf("process 1111.... \r\n");
        delay_ms(3000);

        /*清除 WU 状态位*/
        PWR_ClearFlag (PWR_FLAG_WU);
        /* 使能 WKUP 引脚的唤醒功能 , 使能 PA0*/
        PWR_WakeUpPinCmd(ENABLE);
        /* 进入待机模式 */
        PWR_EnterSTANDBYMode();

        printf("process 2222.... \r\n");
        delay_ms(3000);
    }
    return 0;
}

```

我们这里按下 PB8 按键
 唤醒待机模式下的
 STM32



看起来没有唤醒，这是因为 WKUP 引脚是固定的引脚，所以 PB8 这种通用引脚是不行的。而且待机模式的唤醒不需要设置引脚外部中断。

而且第二次给 STM32 下载程序就不行了，这是因为第二次上电 STM32 也进入了主程序的待机模式，在这种情况下只有按住板子的复位按键，然后 keil 点击下载程序，在 keil 点击 Download 的后 1 秒必须马上放开复位按键，这样才能下载程序成功。

我们后面用 RTC 的闹钟功能来测试唤醒 STM32 的待机模式

STM32RTC 实时时钟操作，及 STM32 待机唤醒

```

void RTC_Configuration(void)
{
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_PWR | RCC_APB1Periph_BKP, ENABLE); //RTC需要打开电源管理和后备电池管理
    PWR_BackupAccessCmd(ENABLE); //打开RTC后备电源供电,这样纽扣电池才能供电,这行代码一定放在RCC_APB1PeriphClockCmd之后执行
    BKP_DeInit(); //备份区域复位

    RCC_LSEConfig(RCC_LSE_ON); //开启LSE晶振
    while(RCC_GetFlagStatus(RCC_FLAG_LSERDY)==RESET); //等待LSE晶振开启

    RCC_RTCCLKConfig(RCC_RTCCLKSource_LSE); //选择LSE时钟为RTC时钟源
    RCC_RTCCLKCmd(ENABLE); //使能RTC时钟

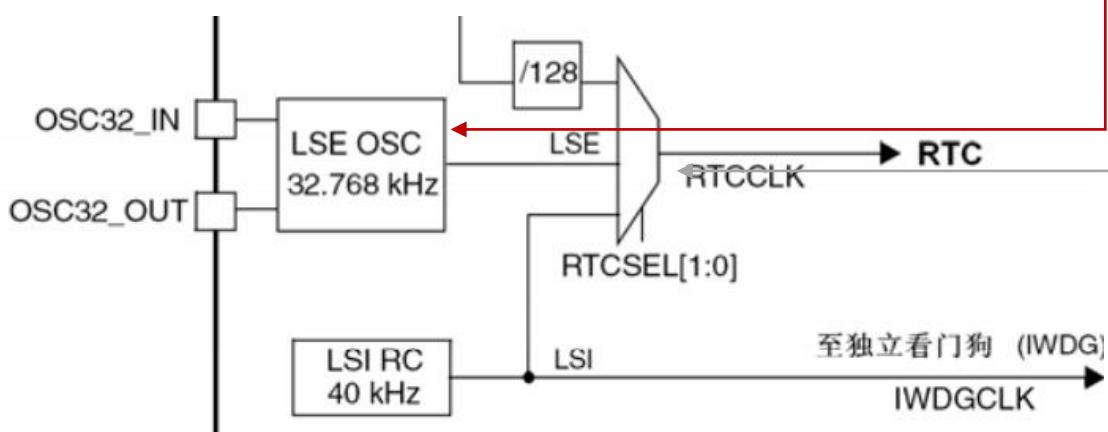
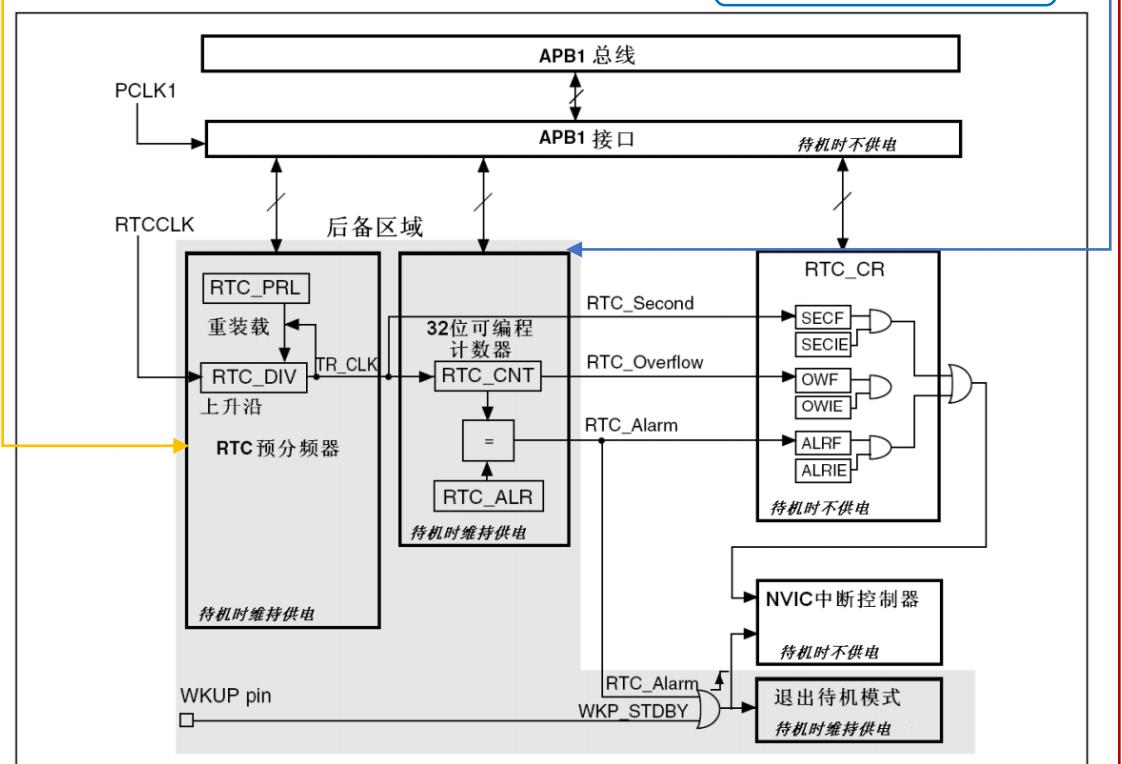
    RTC_WaitForSynchro(); //等待RTC时钟和APB1总线时钟同步
    RTC_WaitForLastTask(); //为了安全起见,等待对RTC寄存器操作完成

    RTC_ITConfig(RTC_IT_SEC, ENABLE); //开启RTC秒中断
    RTC_WaitForLastTask(); //为了安全起见,等待秒中断对RTC寄存器操作完成

    RTC_SetPrescaler(32767); //分频频率为(32.768kHz)/(32767+1)=1Hz, (32768Hz)/(32767+1)=1Hz, 所以RTC 1hz计数一次
    //为什么写32767,这是因为RTC_SetPrescaler函数内部会自动+1,然后写入分频寄存器
    //32768分频还有个原因是外部晶振是32768hz,如果你想计数快点,你可以修改外部晶振或者分频系数

    RTC_WaitForLastTask(); //为了安全起见,等待分频寄存器操作完成
}

```



```

void RTC_Configuration(void)
{
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_PWR | RCC_APB1Periph_BKP,ENABLE); //RTC 需要打开电源管理和后备电池管理
    PWR_BackupAccessCmd(ENABLE); //打开 RTC 后备电源供电, 这样纽扣电池才能供电,这行代码一定放在 RCC_APB1PeriphClockCmd 之后执行

    BKP_DeInit(); //备份区域复位

    RCC_LSEConfig(RCC_LSE_ON); //开启 LSE 晶振
    while(RCC_GetFlagStatus(RCC_FLAG_LSERDY)==RESET); //等待 LSE 晶振开启

    RCC_RTCCLKConfig(RCC_RTCCLKSource_LSE); //选择 LSE 时钟为 RTC 时钟源
    RCC_RTCCLKCmd(ENABLE); //使能 RTC 时钟

    RTC_WaitForSynchro(); //等待 RTC 时钟和 APB1 总线时钟同步
    RTC_WaitForLastTask(); //为了安全起见, 等待对 RTC 寄存器操作完成

    RTC_ITConfig(RTC_IT_SEC,ENABLE); //开启 RTC 秒中断
    RTC_WaitForLastTask(); //为了安全起见, 等待秒中断对 RTC 寄存器操作完成

    RTC_SetPrescaler(32767); //分频频率为(32.768kHz)/(32767+1)=1Hz , (32768hz)/(32767+1)=1Hz, 所以 RTC 1hz 计数一次
                           //为什么写 32767, 这是因为 RTC_SetPrescaler 函数内部会自动+1, 然后写入分频寄存器
                           //32768 分频还有个原因是外部晶振是 32768hz, 如果你想计数快点, 你可以修改外部晶振或者分频系数

    RTC_WaitForLastTask(); //为了安全起见, 等待分频寄存器操作完成
}

```

设置 RTC 中断函数

```

void RTC_IRQHandler(void) {
    if (RTC_GetITStatus(RTC_IT_SEC) != RESET) {
        /* Clear the RTC Second interrupt */
        RTC_ClearITPendingBit(RTC_IT_SEC);
    }
}

```

什么都不做, 清楚标志位即可

```

int main(void)
{
    uint32_t RTCCounterValue;

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

    RTC_Configuration(); //初始化RTC
    RTC_SetCounter(100); //设置RTC当前时间 填入uint_32的数据
    RTC_WaitForLastTask(); //等待时间设置完成
    //设置计数的开始时间

    key_gpio_interrupt_init(); //PB8 IO口外部输入中断, 唤醒_WFI
    delay_ms(1000);
    printf("xxxxxxxx\r\n");

    while(1)
    {
        RTCCounterValue = RTC_GetCounter(); //获取实时时钟时间
        printf("RTC sec = %d\r\n", RTCCounterValue);
        delay_ms(1000);
    }
    return 0;
}

xxxxxxxx
RTC sec = 101
RTC sec = 102
RTC sec = 103
RTC sec = 104
RTC sec = 105
RTC sec = 106

```

从设置的 100
开始向后+1,
获取秒时间

2.下面来设置 RTC 闹钟功能，主要用来唤醒待机模式的 STM32

```
void RTC_Configuration(void)
{
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_PWR | RCC_APB1Periph_BKP, ENABLE); //RTC需要打开电源管理和后备电池管理
    PWR_BackupAccessCmd(ENABLE); //打开RTC后备电源供电,这样纽扣电池才能供电,这行代码一定放在RCC_APB1PeriphClockCmd之后执行
    BKP_DeInit(); //备份区域复位

    RCC_LSEConfig(RCC_LSE_ON); //开启LSE晶振
    while(RCC_GetFlagStatus(RCC_FLAG_LSERDY)==RESET); //等待LSE晶振开启

    RCC_RTCCLKConfig(RCC_RTCCLKSource_LSE); //选择LSE时钟为RTC时钟源
    RCC_RTCCLKCmd(ENABLE); //使能RTC时钟

    RTC_WaitForSynchro(); //等待RTC时钟和APB1总线时钟同步
    RTC_WaitForLastTask(); //为了安全起见,等待对RTC寄存器操作完成

    RTC_ITConfig(RTC_IT_SEC, ENABLE); //开启RTC秒中断
    RTC_WaitForLastTask(); //等待秒中断开启完成
    RTC_ITConfig(RTC_IT_ALR, ENABLE); //开启闹钟中断
    RTC_WaitForLastTask(); //为了安全起见,等待秒中断对RTC寄存器操作完成

    RTC_SetPrescaler(32767); //分频频率为(32.768khz)/(32767+1)=1Hz , (32768hz)/(32767+1)=1Hz, 所以RTC 1hz计数一次
    //为什么写32767, 这是因为RTC_SetPrescaler函数内部会自动+1, 然后写入分频寄存器
    //32768分频还有个原因是外部晶振是32768hz, 如果你想计数快点, 你可以修改外部晶振或者分频系数

    RTC_WaitForLastTask(); //为了安全起见,等待分频寄存器操作完成
}
```

时钟初始化

```
void RTC_Configuration(void)
{
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_PWR | RCC_APB1Periph_BKP, ENABLE); //RTC 需要打开电源管理
    PWR_BackupAccessCmd(ENABLE); //打开 RTC 后备电源供电,这样纽扣电池才能供电,这行代码一定放在
    RCC_APB1PeriphClockCmd 之后执行
    BKP_DeInit(); //备份区域复位

    RCC_LSEConfig(RCC_LSE_ON); //开启 LSE 晶振
    while(RCC_GetFlagStatus(RCC_FLAG_LSERDY)==RESET); //等待 LSE 晶振开启

    RCC_RTCCLKConfig(RCC_RTCCLKSource_LSE); //选择 LSE 时钟为 RTC 时钟源
    RCC_RTCCLKCmd(ENABLE); //使能 RTC 时钟

    RTC_WaitForSynchro(); //等待 RTC 时钟和 APB1 总线时钟同步
    RTC_WaitForLastTask(); //为了安全起见,等待对 RTC 寄存器操作完成

    RTC_ITConfig(RTC_IT_SEC, ENABLE); //开启 RTC 秒中断
    RTC_WaitForLastTask(); //等待秒中断开启完成
    RTC_ITConfig(RTC_IT_ALR, ENABLE); //开启闹钟中断
    RTC_WaitForLastTask(); //为了安全起见,等待秒中断对 RTC 寄存器操作完成

    RTC_SetPrescaler(32767); //分频频率为(32.768khz)/(32767+1)=1Hz , (32768hz)/(32767+1)=1Hz, 所以 RTC
    1hz 计数一次
    //为什么写 32767, 这是因为 RTC_SetPrescaler 函数内部会自动+1, 然后写入分频寄存器
    //32768 分频还有个原因是外部晶振是 32768hz, 如果你想计数快点, 你可以修改外部晶振或者分频系数

    RTC_WaitForLastTask(); //为了安全起见,等待分频寄存器操作完成
}
```

```

void RTC_NVIC_init(void)
{
    NVIC_InitTypeDef NVIC_InitStructure;

    //NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1); //设置中断组
    /*秒中断*/
    NVIC_InitStructure.NVIC IRQChannel = RTC IRQn; //RTC全局中断
    NVIC_InitStructure.NVIC IRQChannelPreemptionPriority = 2;
    NVIC_InitStructure.NVIC IRQChannelSubPriority = 1;
    NVIC_InitStructure.NVIC IRQChannelCmd = ENABLE; //使能该通道中断
    NVIC_Init(&NVIC_InitStructure);

    /*加入闹钟中断*/
    NVIC_InitStructure.NVIC IRQChannel = RTCAlarm IRQn; //闹钟中断
    NVIC_InitStructure.NVIC IRQChannelPreemptionPriority = 1; //比RTC全局中断的优先级高
    NVIC_InitStructure.NVIC IRQChannelSubPriority = 2;
    NVIC_InitStructure.NVIC IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);

}

设置秒时间中断，和闹钟时间中断优先级，闹钟中断优先级一定要大于秒中断

void RTC_NVIC_init(void)
{
    NVIC_InitTypeDef NVIC_InitStructure;

    //NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1); //设置中断组
    /*秒中断*/
    NVIC_InitStructure.NVIC IRQChannel = RTC IRQn; //RTC 全局中断
    NVIC_InitStructure.NVIC IRQChannelPreemptionPriority = 2;
    NVIC_InitStructure.NVIC IRQChannelSubPriority = 1;
    NVIC_InitStructure.NVIC IRQChannelCmd = ENABLE; //使能该通道中断
    NVIC_Init(&NVIC_InitStructure);

    /*加入闹钟中断*/
    NVIC_InitStructure.NVIC IRQChannel = RTCAlarm IRQn; //闹钟中断
    NVIC_InitStructure.NVIC IRQChannelPreemptionPriority = 1; //比 RTC 全局中断的优先级高
    NVIC_InitStructure.NVIC IRQChannelSubPriority = 2;
    NVIC_InitStructure.NVIC IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);

}

void RTC_IRQHandler(void) {
    if (RTC_GetITStatus(RTC_IT_SEC) != RESET) {
        /* Clear the RTC Second interrupt */
        RTC_ClearITPendingBit(RTC_IT_SEC);
        printf("sec interrupt \r\n");
    }

    if (RTC_GetITStatus(RTC_IT_ALR) != RESET) //闹钟中断
    {
        printf("alarm interrupt \r\n");
        RTC_ClearITPendingBit(RTC_IT_ALR); //清闹钟中断
    }

    RTC_ClearITPendingBit(RTC_IT_SEC | RTC_IT_OW); //清除闹钟中断之后还有清除秒中断
    RTC_WaitForLastTask();
}

```

秒时间中断和闹钟中断都在 RTC_IRQHandler 里面执行

闹钟中断时间到了，你可以在中断里面修改
下次闹钟的时间

```

int main(void)
{
    uint32_t RTCCounterValue;

    RCC_Configuration(); // 初始化时钟
    USART_Config(115200); // 初始化串口

    RTC_Configuration(); // 初始化RTC
    RTC_NVIC_Init(); // 设置RTC中断优先级
    RTC_SetCounter(100); // 设置RTC当前时间 填入uint_32的数
    RTC_WaitForLastTask(); // 等待时间设置完

    delay_ms(1000);
    printf("xxxxzzzz\r\n");

    RTC_WaitForLastTask(); // 设置闹钟值之前必须等待RTOFF置位
    RTC_SetAlarm(110); // 设置RTC闹钟值， RTC秒计数到110就执行闹钟中断

    while(1)
    {
        RTCCounterValue = RTC_GetCounter(); // 获取实时时钟时间
        printf("RTC sec = %d\r\n", RTCCounterValue);
        delay_ms(1000);
    }
    return 0;
}

```

闹钟中断唤醒待机模式下的 STM32

```

int main(void)
{
    uint32_t RTCCounterValue;

    RCC_APB1PeriphClockCmd(RCC_APB1Periph_PWR, ENABLE); /* 使能电源管理单元的时钟，必须要使能时钟才能进入待机模式 */
    RCC_Configuration(); // 初始化时钟
    USART_Config(115200); // 初始化串口

    RTC_Configuration(); // 初始化RTC
    RTC_NVIC_Init(); // 设置RTC中断优先级
    RTC_SetCounter(100); // 设置RTC当前时间 填入uint_32的数
    RTC_WaitForLastTask(); // 等待时间设置完

    delay_ms(1000);
    printf("xxxxzzzz\r\n");

    RTC_WaitForLastTask(); // 设置闹钟值之前必须等待RTOFF置位
    RTC_SetAlarm(110); // 设置RTC闹钟值， RTC秒计数到110就执行闹钟中断

    while(1)
    {
        RTCCounterValue = RTC_GetCounter(); // 获取实时时钟时间
        printf("RTC sec = %d\r\n", RTCCounterValue);
        delay_ms(1000);

        /* 清除 WU 状态位 */
        PWR_ClearFlag(PWR_FLAG_WU);
        /* 使能 WKUP 引脚的唤醒功能， 使能 PA0 */
        PWR_WakeUpPinCmd(ENABLE);
        /* 进入待机模式 */
        PWR_EnterSTANDBYMode();
    }
    return 0;
}

```

```

xsec interrupt
xxxxzzzz
RTC sec = 101
sec interrupt
sec interrupt
alarm interrupt
xsec interrupt
xxxxzzzz
RTC sec = 101
sec interrupt

```

因为待机模式唤醒是复位重启，所以闹钟中断产生后，程序从 main 函数第一行开始，重新开始执行

RTC 时间日期设置

```
//时间结构体
typedef struct
{
    vu8 hour;
    vu8 min;
    vu8 sec;
    //公历年月日周
    vu16 w_year;
    vu8 w_month;
    vu8 w_date;
    vu8 week;
} _calendar_obj;

_calendar_obj calendar;

void RTC_Configuration(void);
void RTC_NVIC_init(void);
uint8_t RTC_Set(u16 year,u8 mon,u8 day,u8 hour,u8 min,u8 sec);
uint8_t RTC_Get(void);

int main(void)
{

    RCC_APB1PeriphClockCmd(RCC_APB1Periph_PWR,ENABLE);/* 使能电源管理单元的时钟,必须要使能时钟才能进入待机模式
*/
    RCC_configuration(); //初始化时钟
    USART_config(115200); //初始化串口

    RTC_Configuration(); //初始化 RTC
    RTC_NVIC_init(); //设置 RTC 中断优先级

    RTC_Set(2018,11,30,23,59,50);

    delay_ms(1000);
    printf("xxxxzzzz\r\n");

    RTC_WaitForLastTask(); //设置闹钟值之前必须等待 RTOFF 置位
    RTC_SetAlarm(110); //设置 RTC 闹钟值, RTC 秒计数到 110 就执行闹钟中断

    while(1)
    {
        RTC_Get(); //获取实时时钟时间
        printf("RTC      year      month      day      hour      min      sec      =      %d      %d      %d
- %d : %d : %d\r\n",calendar.w_year,calendar.w_month,calendar.w_date,calendar.hour,calendar.min,calendar.sec);
        delay_ms(1000);

    }
    return 0;
}

void RTC_Configuration(void)
{
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_PWR | RCC_APB1Periph_BKP,ENABLE); //RTC 需要打开电源管理和后备电池管理
    PWR_BackupAccessCmd(ENABLE); // 打开 RTC 后备电源供电, 这样纽扣电池才能供电, 这行代码一定放在
    RCC_APB1PeriphClockCmd 之后执行
    BKP_DeInit(); //备份区域复位

    RCC_LSEConfig(RCC_LSE_ON); //开启 LSE 晶振
    while(RCC_GetFlagStatus(RCC_FLAG_LSERDY)==RESET); //等待 LSE 晶振开启

    RCC_RTCCLKConfig(RCC_RTCCLKSource_LSE); //选择 LSE 时钟为 RTC 时钟源
    RCC_RTCCLKCmd(ENABLE); //使能 RTC 时钟

    RTC_WaitForSynchro(); //等待 RTC 时钟和 APB1 总线时钟同步
    RTC_WaitForLastTask(); //为了安全起见, 等待对 RTC 寄存器操作完成

    RTC_ITConfig(RTC_IT_SEC,ENABLE); //开启 RTC 秒中断
}
```

```

RTC_WaitForLastTask(); //等待秒中断开启完成
RTC_ITConfig(RTC_IT_ALR, ENABLE); //开启闹钟中断
RTC_WaitForLastTask(); //为了安全起见，等待秒中断对 RTC 寄存器操作完成

RTC_SetPrescaler(32767); //分频频率为(32.768khz)/(32767+1)=1Hz , (32768hz)/(32767+1)=1Hz, 所以 RTC 1hz 计数一次
//为什么写 32767, 这是因为 RTC_SetPrescaler 函数内部会自动+1, 然后写入分频寄存器,32768 分频还有个原因是外部晶振是 32768hz, 如果你想计数快点, 你可以修改外部晶振或者分频系数

RTC_WaitForLastTask(); //为了安全起见，等待分频寄存器操作完成

}

void RTC_NVIC_init(void)
{
    NVIC_InitTypeDef NVIC_InitStructure;

    //NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1); //设置中断组
    /*秒中断*/
    NVIC_InitStructure.NVIC_IRQChannel = RTC_IRQn; //RTC 全局中断
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 2;
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE; //使能该通道中断
    NVIC_Init(&NVIC_InitStructure);

    /*加入闹钟中断*/
    NVIC_InitStructure.NVIC_IRQChannel = RTCAlarm_IRQn; //闹钟中断
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1; //比 RTC 全局中断的优先级高
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 2;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);

}

void RTC_IRQHandler(void) {
    if (RTC_GetITStatus(RTC_IT_SEC) != RESET) {
        /* Clear the RTC Second interrupt */
        RTC_ClearITPendingBit(RTC_IT_SEC);
        printf("sec interrupt \r\n");
    }

    if(RTC_GetITStatus(RTC_IT_ALR)!= RESET)//闹钟中断
    {
        printf("alarm interrupt \r\n");
        RTC_ClearITPendingBit(RTC_IT_ALR); //清闹钟中断
    }

    RTC_ClearITPendingBit(RTC_IT_SEC | RTC_IT_OW); //清除闹钟中断之后还有清除秒中断
    RTC_WaitForLastTask();
}
}

uint8_t Is_Leap_Year(u16 pyear)
{
    if(pyear%4==0) //首先需能被 4 整除
    {
        if(pyear%100==0)
        {
            if(pyear%400==0) return 1; //如果以 00 结尾, 还要能被 400 整除
            else return 0;
        }
        else
            return 1;
    }
    else
        return 0;
}

```

```

const uint8_t mon_table[12]={31,28,31,30,31,30,31,31,30,31,30,31};
/*
得到当前的时间
成功返回 0, 错误返回其它
*/
uint8_t RTC_Get(void)
{
    static u16 dayCount=0;
    u32 secCount=0;
    u32 tmp=0;
    u16 tmp1=0;
    secCount=RTC_GetCounter();
    tmp=secCount/86400;//得到天数
    if(dayCount!=tmp)//超过一天
    {
        dayCount=tmp;
        tmp1=1970;//从 1970 年开始
        while(tmp1>=365)
        {
            if(Is_Leap_Year(tmp1))//是闰年
            {
                if(tmp1>=366)
                    tmp1-=366;//减掉闰年的天数
                else
                {
                    //    tmp1++;
                    break;
                }
            }
            else
                tmp1-=365;//平年
            tmp1++;
        }
        calendar.w_year=tmp1;//得到年份
        tmp1=0;
        while(tmp1>=28)//超过一个月
        {
            if(Is_Leap_Year(calendar.w_year)&&tmp1==1)//当年是闰年且轮循到 2 月
            {
                if(tmp1>=29)
                    tmp1=29;
                else
                    break;
            }
            else
            {
                if(tmp1>=mon_table[tmp1])//平年
                    tmp1-=mon_table[tmp1];
                else
                    break;
            }
            tmp1++;
        }
        calendar.w_month=tmp1+1;//得到月份, tmp1=0 表示 1 月, 所以要加 1
        calendar.w_date=tmp1+1; //得到日期, 因为这一天还没过完, 所以 tmp 只到其前一天, 但是显示的时候要显示
正常日期
    }
    tmp=secCount%86400;//得到秒钟数
    calendar.hour=tmp/3600;//小时
    calendar.min=(tmp%3600)/60;//分钟
    calendar.sec=(tmp%3600)%60;//秒
    return 0;
}

```

```

/*
*设置时钟
*把输入的时钟转换为秒钟
*以 1970 年 1 月 1 日为基准
*1970~2099 年为合法年份
*返回值: 0, 成功; 其它: 错误
*/
uint8_t RTC_Set(u16 year,u8 mon,u8 day,u8 hour,u8 min,u8 sec)
{
    u16 t;
    u32 secCount=0;
    if(year<1970||year>2099)
        return 1;//3? 1
    for(t=1970;t<year;t++) //把所有年份的秒钟相加
    {
        if(Is_Leap_Year(t))//闰年
            secCount+=31622400;//闰年的秒钟数
        else
            secCount+=31536000;
    }
    mon-=1;//先减掉一个月再算秒数 (如现在是 5 月 10 日, 则只需要算前 4 个月的天数, 再加上 10 天, 然后计算秒数)
    for(t=0;t<mon;t++)
    {
        secCount+=(u32)mon_table[t]*86400;//月份秒钟数相加
        if(Is_Leap_Year(year)&&t==1)
            secCount+=86400;//闰年, 2 月份增加一天的秒钟数
    }

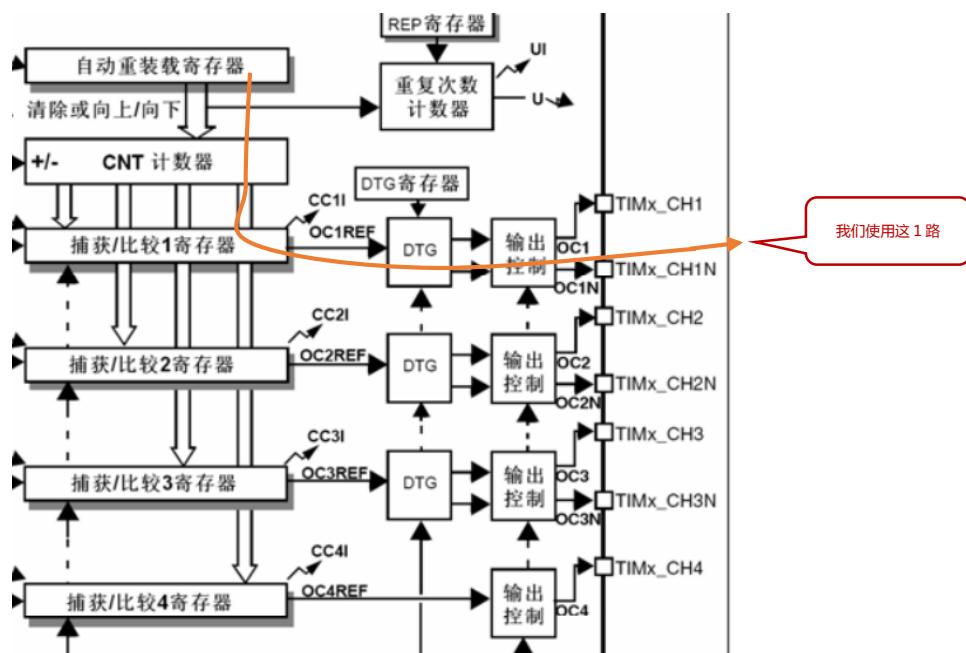
    secCount+=(u32)(day-1)*86400;//把前面日期的秒钟数相加 (这一天还没过完, 所以-1)
    secCount+=(u32)hour*3600;//小时秒钟数
    secCount+=(u32)min*60;//分钟秒钟数
    secCount+=sec;

    // RCC_APB1PeriphClockCmd(RCC_APB1Periph_PWR      | RCC_APB1Periph_BKP,ENABLE);
    // PWR_BackupAccessCmd(ENABLE);
    RTC_SetCounter(secCount);//设置 RTC 计数器的值
    RTC_WaitForLastTask(); //等待最近一次对 RTC 寄存器的写操作完成
    RTC_Get(); //更新时间
    return 0;
}

```

高级定时器实现 PWM 互补输出

需要两路 PWM 输出的需求时, 就用这种方法



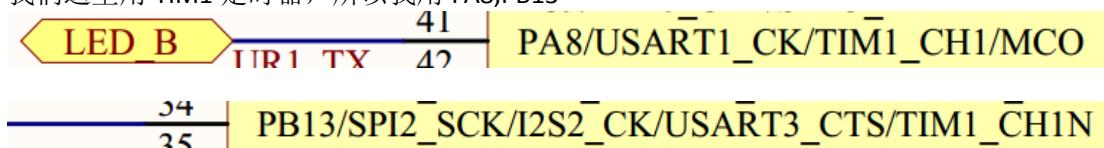
不同的定时器只能设置它对应的
GPIO 管脚输出 PWM，比如
TIM1 就只能设置 PA8/PE9 输出一
路 PWM, PB13/PA7/PE8 输出对
应的第 2 路 PWM

如果是 TIM8 定时器，就只能设置
PC6, PC7 输出 PWM

表 33-1 高级控制和通用定时器通道引脚分布

引脚	高级定时器		通用定时器		
	TIM1	TIM8	TIM2	TIM5	TIM3
CH1	PA8/PE9	PC6	PA0/PA15	PA0	PA6/PC6/PB4
CH1N	PB13/PA7/PE8	PA7			
CH2	PA9/PE11	PC7	PA1/PB3	PA1	PA7/PC7/PB5
CH2N	PB14/PB0/PE10	PB0			
CH3	PA10/PE13	PC8	PA2/PB10	PA2	PB0/PC8
CH3N	PB15/PB1/PE12	PB1			
CH4	PA11/PE14	PC9	PA3/PB11	PA3	PB1/PC9
ETR	PA12/PE7	PA0	PA0/PA15		PD2
BKIN	PB12/PA6/PE15	PA6			

我们这里用 TIM1 定时器，所以我用 PA8, PB13



```

static void PWM_IO_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;

    // 输出比较通道 GPIO 初始化
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE); // 使用PA8输出PWM
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    // 输出比较通道互补通道 GPIO 初始化
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE); // 使用PB13输出互补的PWM
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_13;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOB, &GPIO_InitStructure);

    // 输出比较通道刹车通道 GPIO 初始化
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE);
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_12;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOB, &GPIO_InitStructure);
    // BKIN 引脚默认先输出低电平
    GPIO_ResetBits(GPIOB, GPIO_Pin_12);
}

```

设置 GPIO 功能

```

static void TIM1_PWM_Mode(void)
{
    /*-----时基结构体初始化-----*/
    TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;

    /*-----输出比较结构体初始化-----*/
    TIM_OCInitTypeDef TIM_OCInitStructure;

    /*-----刹车和死区结构体初始化-----*/
    TIM_BDTRInitTypeDef TIM_BDTRInitStructure;

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_TIM1, ENABLE); // 开启定时器时钟, 即内部时钟 CK_INT=72M

    TIM_TimeBaseStructure.TIM_Period=(8-1); // 自动重装载寄存器的值, 累计 TIM_Period+1 个频率后产生一个更新或者中断

    TIM_TimeBaseStructure.TIM_Prescaler=(9-1); // 驱动 CNT 计数器的时钟 = Fck_int/(psc+1)

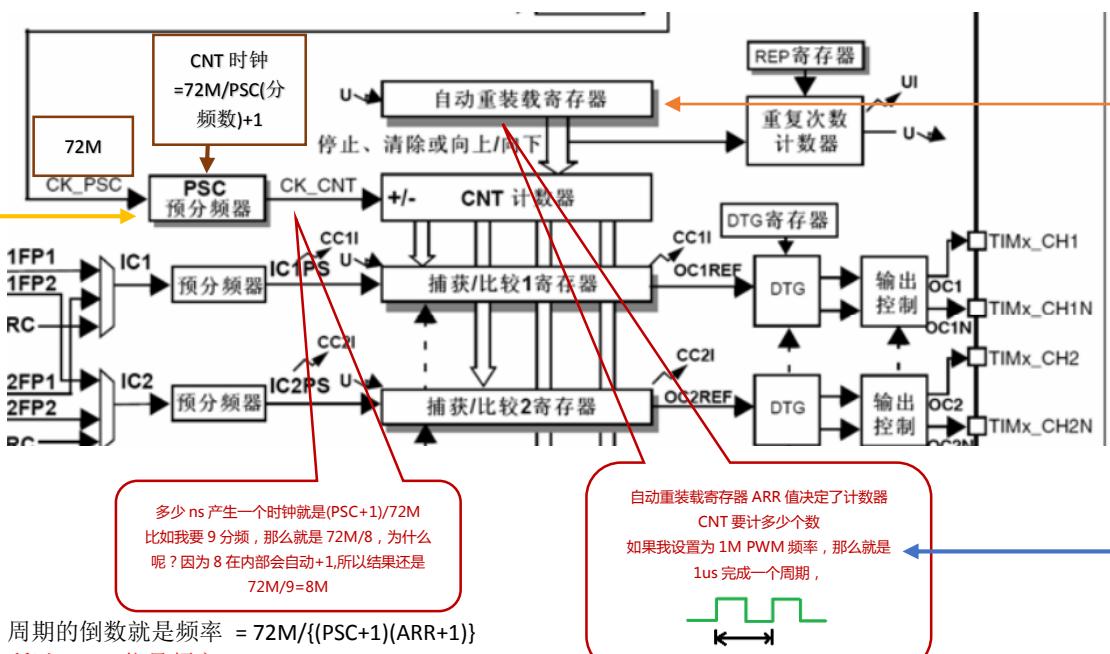
    TIM_TimeBaseStructure.TIM_ClockDivision=TIM_CKD_DIV1; // 时钟分频因子, 配置死区时间时需要用到, 不使用死区时间也要保留代码

    TIM_TimeBaseStructure.TIM_CounterMode=TIM_CounterMode_Up; // 计数器计数模式, 设置为向上计数

    TIM_TimeBaseStructure.TIM_RepetitionCounter=0; // 重复计数器的值, 没用到不用管

    TIM_TimeBaseInit(TIM1, &TIM_TimeBaseStructure); // 初始化定时器
}

```



周期的倒数就是频率 = $72M/\{(PSC+1)(ARR+1)\}$

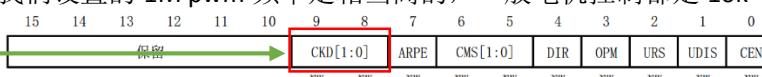
所以 PWM 信号频率 = $72M/\{(PSC+1)(ARR+1)\}$

设置 pwm 为 1M = $72M/\{(8+1)(7+1)\}$ // $72M/\{(PSC+1)(ARR+1)\}$

简化就是 $72M/(9)(8) = 72M/72 = 1M$ pwm 频率

占空比就是 TIM_{Pulse}/TIM_{Period} : 比如 $TIM_{Pulse}=4$ $TIM_{Period} = 8$ 那么 $4/8 = 50\%$ 占空比

我们设置的 1M pwm 频率是相当高的, 一般电机控制都是 10k 左右的 pwm 频率。



位15:10	保留, 始终读为0。
位9:8	CKD[1:0]: 时钟分频因子 (Clock division) 这2位定义在定时器时钟(CK_INT)频率、死区时间和由死区发生器与数字滤波器(ETR,TIx)所用的采样时钟之间的分频比例。 00: tDTS = tCK_INT 01: tDTS = 2 x tCK_INT 10: tDTS = 4 x tCK_INT 11: 保留, 不要使用这个配置

设置内部 72M 时钟是否分频, 如果分频, 就填入参数, 然后用 tDTS 的时钟去计算波形延时输出时间



图 33-8 带死区插入的互补输出

```

TIM_OCIInitStructure.TIM_OCMode = TIM_OCMode_PWM1; // 配置为 PWM 模式 1
TIM_OCIInitStructure.TIM_OutputState = TIM_OutputState_Enable; // PA8输出使能
TIM_OCIInitStructure.TIM_OutputNState = TIM_OutputNState_Enable; // PB13互补输出使能
TIM_OCIInitStructure.TIM_Pulse = 4; // 设置占空比大小
TIM_OCIInitStructure.TIM_OCPolarity = TIM_OCPolarity_High; // 输出通道电平极性配置
TIM_OCIInitStructure.TIM_OCNPolarity = TIM_OCNPolarity_High; // 互补输出通道电平极性配置
TIM_OCIInitStructure.TIM_OCIDleState = TIM_OCIDleState_Set; // 输出通道空闲电平极性配置
TIM_OCIInitStructure.TIM_OCNIdleState = TIM_OCNIdleState_Reset; // 互补输出通道空闲电平极性配置
TIM_OC1Init(TIM1, &TIM_OCIInitStructure);
TIM_OC1PreloadConfig(TIM1, TIM_OCPreload_Enable);

```

当计数器向上计数到 ARR 设置值之后，自动清 0，重新计数到 ARR 值，如此循环

当两路 PWM 信号，PA8,PB13 关闭的时候，IO 口默认持续输出什么电平

一旦 CNT 计数的值超过 CCR 寄存器写入的值，那么就输出低电平



模式	计数器 CNT 计算方式	说明
PWM1	递增	CNT<CCR，通道 CH 为有效，否则为无效
	递减	CNT>CCR，通道 CH 为无效，否则为有效
PWM2	递增	CNT<CCR，通道 CH 为无效，否则为有效
	递减	CNT>CCR，通道 CH 为有效，否则为无效

这是 PWM1 模式，递增方式的 CNT 和 CCR 计算方式
PWM2 和其余递增递减方式也是这个思路。

计数器计数的值 CNT<CCR 寄存器写入的值，有效，配置波形输出高电平



PA8



PA8



PB13跟着PA8互补输出同样的PWM信号

就是设置 CH 通道有效时候输出什么电平，我们设置的是高电平。
你也可以设置成低电平，看需求

模式	计数器 CNT 计算方式	说明
PWM1	递增	CNT<CCR，通道 CH 为有效，否则为无效
	递减	CNT>CCR，通道 CH 为无效，否则为有效
PWM2	递增	CNT<CCR，通道 CH 为无效，否则为有效
	递减	CNT>CCR，通道 CH 为有效，否则为无效

```

// 有关刹车和死区结构体的成员具体可参考 BDTR 寄存器的描述

TIM_BDTRInitStructure.TIM_OSSRState = TIM_OSSRState_Enable;
TIM_BDTRInitStructure.TIM_OSSIState = TIM_OSSIState_Enable;
TIM_BDTRInitStructure.TIM_LOCKLevel = TIM_LOCKLevel_1;
// 输出比较信号死区时间配置，具体如何计算可参考 BDTR:UTG[7:0]的描述
// 这里配置的死区时间为 152ns
TIM_BDTRInitStructure.TIM_DeadTime = 11; 这三个参数看  
数据手册
TIM_BDTRInitStructure.TIM_Break = TIM_Break_Enable;
// 当 BKIN 引脚检测到高电平的时候，输出比较信号被禁止，就好像是刹车一样
TIM_BDTRInitStructure.TIM_BreakPolarity = TIM_BreakPolarity_High;
TIM_BDTRInitStructure.TIM_AutomaticOutput = TIM_AutomaticOutput_Enable;
TIM_BDTRConfig(TIM1, &TIM_BDTRInitStructure);

// 使能计数器
TIM_Cmd(TIM1, ENABLE); 打开 TIM1 开始计数
// 主输出使能，当使用的是通用定时器时，这句不需要
TIM_CtrlPWMOutputs(TIM1, ENABLE);
}


当 TIM1 通道 PB12 被外部拉高电平时，  

PWM 输出就会停止，所以为什么我还初始  

了一个 PB12 引脚，PB12 复用成 BKIN 信  

号


BKIN
PB12/PA6/PE15


```

13.4.18 TIM1和TIM8刹车和死区寄存器(TIMx_BDTR)

位7:0	UTG[7:0]: 死区发生器设置 (Dead-time generator setup) 这些位定义了插入互补输出之间的死区持续时间。假设DT表示其持续时间： $DT = DTG[7:0] \times T_{dtg}$, $T_{dtg} = T_{DTS}$; $DTG[7:5]=0xx \Rightarrow DT = DTG[7:0] \times T_{dtg}, T_{dtg} = 2 \times T_{DTS};$ $DTG[7:5]=110 \Rightarrow DT = (32+DTG[4:0]) \times T_{dtg}, T_{dtg} = 8 \times T_{DTS};$ $DTG[7:5]=111 \Rightarrow DT = (32+DTG[4:0]) \times T_{dtg}, T_{dtg} = 16 \times T_{DTS};$ 例：若 $T_{DTS} = 125\text{ns}$ (8MHz)，可能的死区时间为： 0到15875ns，若步长时间为125ns； 16us到31750ns，若步长时间为250ns； 32us到63us，若步长时间为1us； 64us到126us，若步长时间为2us； 注：一旦LOCK级别(TIMx_BDTR寄存器中的LOCK位)设为1、2或3，则不能修改这些位
------	---

`TIM_BDTRInitStructure.TIM_DeadTime = 11; //那么就是 0000 1011 第 7:5 位为 0，选择第 1 个 DTG 计算公式`

$$DTG[7:5]=0xx \Rightarrow DT=DTG[7:0] \times T_{dtg}, T_{dtg}=T_{DTS};$$

DT:是死区时间

$DT = 0000\ 1011 \times T_{dtg}$ ， T_{dtg} 也就是 T_{DTS} ，那么 T_{DTS} 等于 CK8 分频的时钟大小

`TIM_TimeBaseStructure.TIM_ClockDivision=TIM_CKD_DIV1; // 时钟分频因子`，
因为 `TIM_CKD_DIV1` 对内部 72M 时钟是 1 分频，所以 $T_{DTS} = 72\text{MHz}$

$$DTG[7:0] = 0000\ 1011 = 11$$

$$DTG \times \frac{1}{T_{dtg}} = 11 \times \frac{1}{72000000\ (72M)} = 152\text{ns}$$

死区时间 152ns

```

static void TIM1_PWM_Mode(void)
{
    /*-----时基结构体初始化-----*/
    TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;

    /*-----输出比较结构体初始化-----*/
    TIM_OCInitTypeDef TIM_OCInitStructure;

    /*-----刹车和死区结构体初始化-----*/
    TIM_BDTRInitTypeDef TIM_BDTRInitStructure;

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_TIM1,ENABLE); // 开启定时器时钟,即内部时钟 CK_INT=72M

    TIM_TimeBaseStructure.TIM_Period=(8-1); // 自动重装载寄存器的值, 累计 TIM_Period+1 个频率后产生一个更新或者中断

    TIM_TimeBaseStructure.TIM_Prescaler= (9-1); // 驱动 CNT 计数器的时钟 = Fck_int/(psc+1)

    TIM_TimeBaseStructure.TIM_ClockDivision=TIM_CKD_DIV1; // 时钟分频因子 , 配置死区时间时需要用到,不使用死区时间也要保留代码

    TIM_TimeBaseStructure.TIM_CounterMode=TIM_CounterMode_Up; // 计数器计数模式, 设置为向上计数

    TIM_TimeBaseStructure.TIM_RepetitionCounter=0; // 重复计数器的值, 没用到不用管

    TIM_TimeBaseInit(TIM1, &TIM_TimeBaseStructure); // 初始化定时器

    TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1; // 配置为 PWM 模式 1

    TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable; // PA8 输出使能

    TIM_OCInitStructure.TIM_OutputNState = TIM_OutputNState_Enable; // PB13 互补输出使能

    TIM_OCInitStructure.TIM_Pulse = 4; // 设置占空比大小

    TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_High; // 输出通道电平极性配置

    TIM_OCInitStructure.TIM_OCNPolarity = TIM_OCNPolarity_High; // 互补输出通道电平极性配置

    TIM_OCInitStructure.TIM_OCIdleState = TIM_OCIdleState_Set; // 输出通道空闲电平极性配置

    TIM_OCInitStructure.TIM_OCNIdleState = TIM_OCNIdleState_Reset; // 互补输出通道空闲电平极性配置

    TIM_OC1Init(TIM1, &TIM_OCInitStructure);
    TIM_OC1PreloadConfig(TIM1, TIM_OCPreload_Enable);

    // 有关刹车和死区结构体的成员具体可参考 BDTR 寄存器的描述

    TIM_BDTRInitStructure.TIM_OSSRState = TIM_OSSRState_Enable;
    TIM_BDTRInitStructure.TIM_OSSIState = TIM_OSSIState_Enable;
    TIM_BDTRInitStructure.TIM_LOCKLevel = TIM_LOCKLevel_1;
    // 输出比较信号死区时间配置, 具体如何计算可参考 BDTR:UTG[7:0]的描述
    // 这里配置的死区时间为 152ns
    TIM_BDTRInitStructure.TIM_DeadTime = 11;
    TIM_BDTRInitStructure.TIM_Break = TIM_Break_Enable;
    // 当 BKIN 引脚检测到高电平的时候, 输出比较信号被禁止, 就好像是刹车一样
    TIM_BDTRInitStructure.TIM_BreakPolarity = TIM_BreakPolarity_High;
    TIM_BDTRInitStructure.TIM_AutomaticOutput = TIM_AutomaticOutput_Enable;
    TIM_BDTRConfig(TIM1, &TIM_BDTRInitStructure);

    // 使能计数器
    TIM_Cmd(TIM1, ENABLE);
    // 主输出使能, 当使用的是通用定时器时, 这句不需要
    TIM_CtrlPWMOutputs(TIM1, ENABLE);
}

int main(void)
{
    RCC_Configuration(); // 初始化时钟
    USART_Config(115200); // 初始化串口

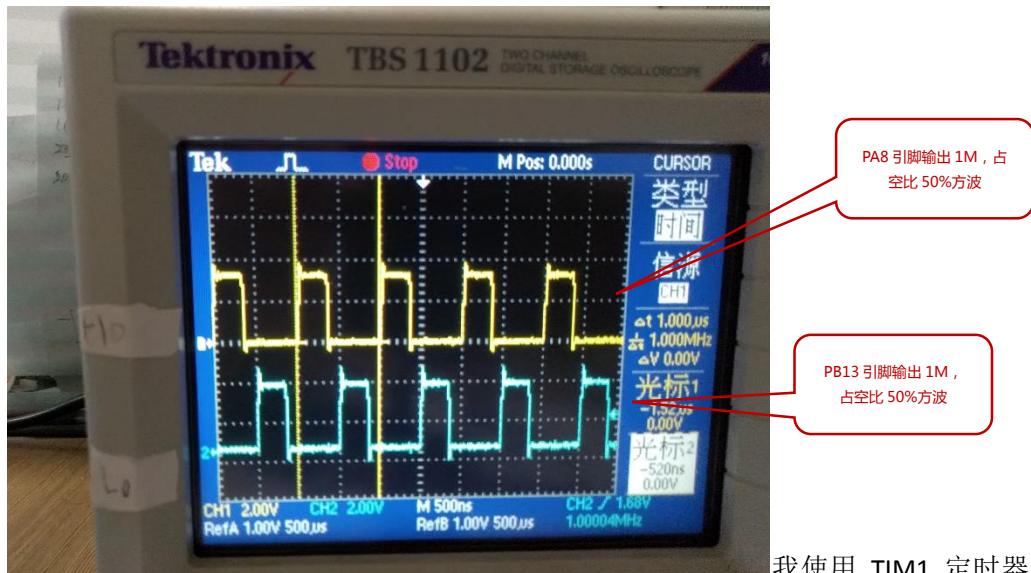
    PWM_IO_Init(); // PB8, PB13, PB12 PWM功能引脚初始化
    TIM1_PWM_Mode(); // 启动 PWM PB8 PB13输出PWM信号

    delay_ms(1000);
    printf("xxxxzzzz\r\n");

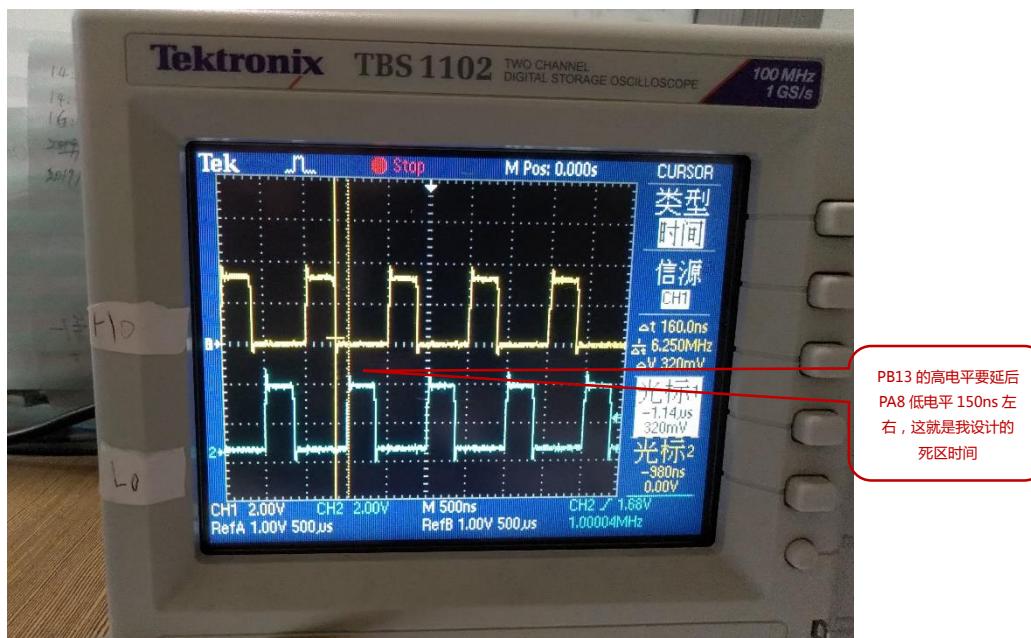
    while(1)
    {
    }

    return 0;
}

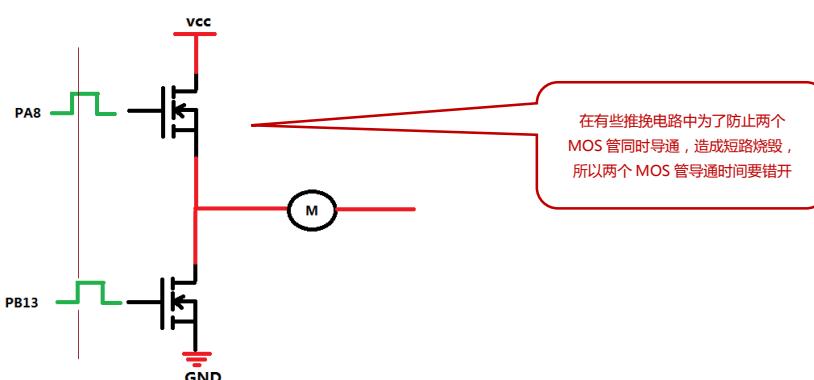
```



我使用 TIM1 定时器设置的 1M 输出，占空比 50%

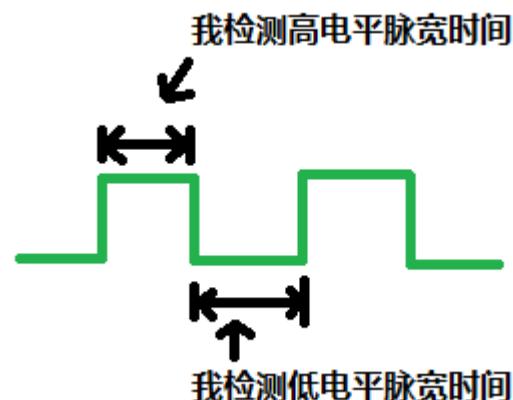


这种死区时间有什么用？



我们就是让两路 PWM 信号的高电平错开，不要同时导通，来保护推挽电路。

输入捕获比较器使用



像这种脉宽检测，如果用单片机 GPIO 加定时器采集就很慢，而且不是很准。

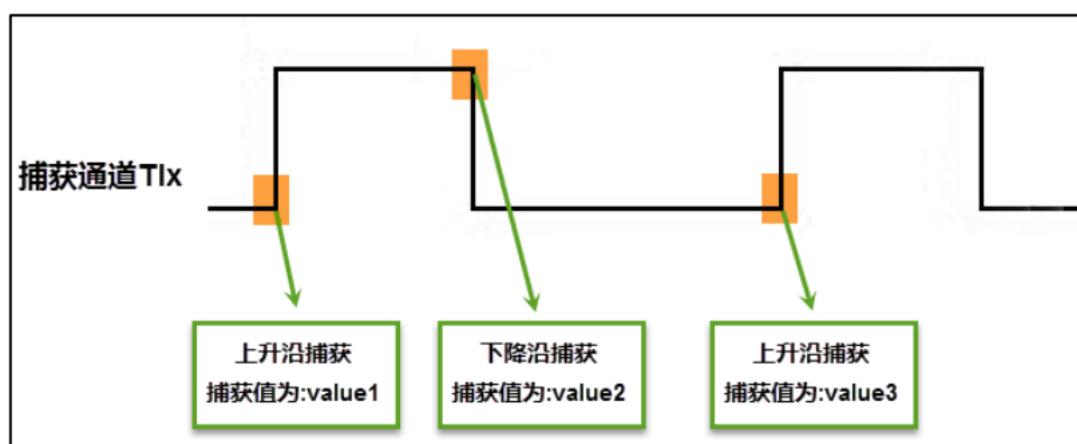
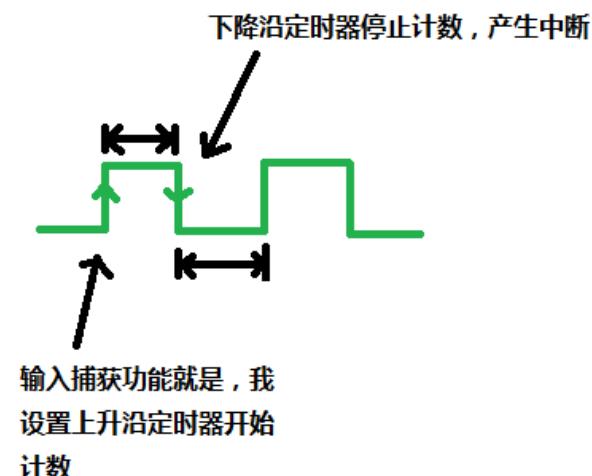
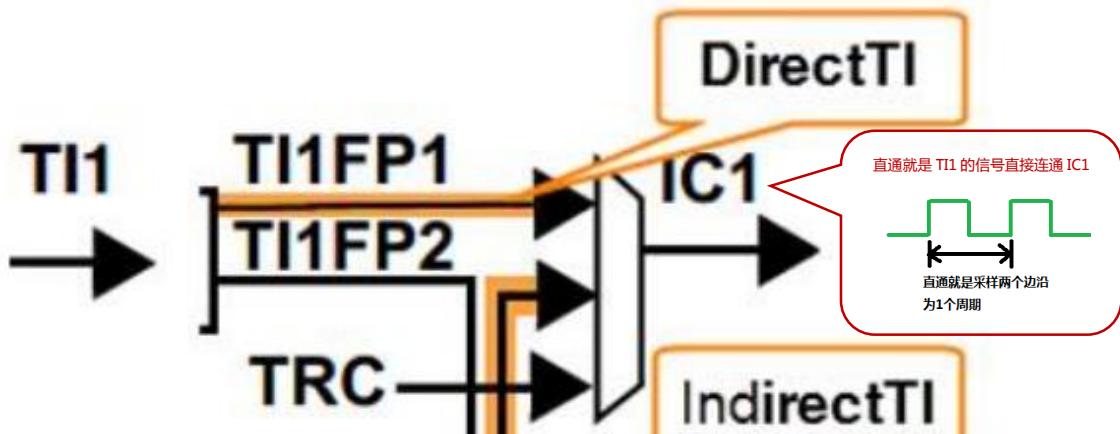


图 33-10 脉宽/频率测量示意图

测量脉宽和频率还有一个更简便的方法就是使用 PWM 输入模式，该模式是输入捕获的特例，只能使用通道 1 和通道 2，通道 3 和通道 4 使用不了



这种只能采用频率，无法采样占空比

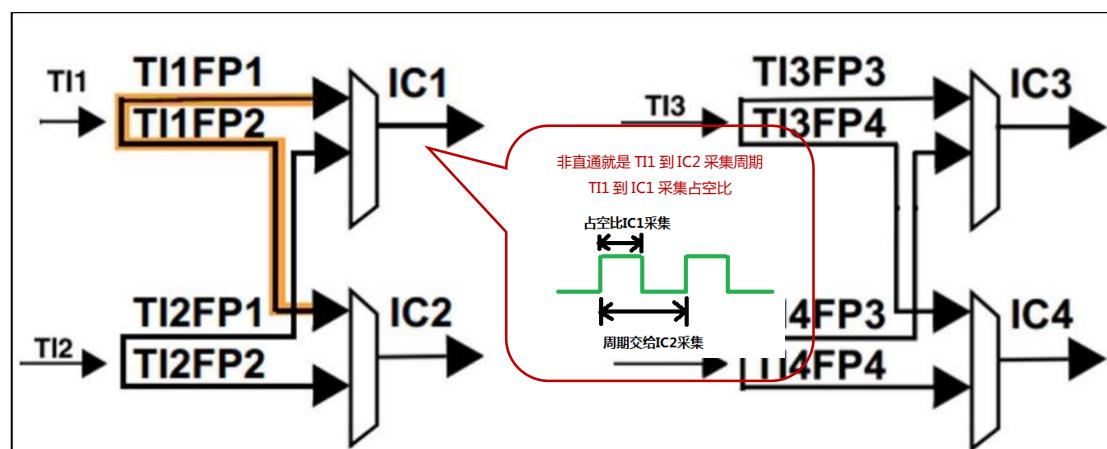


表 33-1 高级控制和通用定时器通道引脚分布

引脚	高级定时器		通用定时器		
	TIM1	TIM8	TIM2	TIM5	TIM3
CH1	PA8/PE9	PC6	PA0/PA15	PA0	PA6/PC6/PB4
CH1N	PB13/PA7/PE8	PA7	-	-	-
CH2	PA9/PE11	PC7	PA1/PB3	PA1	PA7/PC7/PB5
CH2N	PB14/PB0/PE10	PB0	-	-	-
CH3	PA10/PE13	PC8	PA2/PB10	PA2	PB0/PC8
CH3N	PB15/PB1/PE12	PB1	-	-	-
CH4	PA11/PE14	PC9	PA3/PB11	PA3	PB1/PC9
ETR	PA12/PE7	PA0	PA0/PA15	-	PD2
BKIN	PB12/PA6/PE15	PA6	-	-	-

用 PA6 做输入信号脉宽测量引脚

```

void Input_GPIO_init(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;

    // 输入捕获通道 GPIO 初始化
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
    GPIO_Init(GPIOA, &GPIO_InitStructure);
}

```

初始化 PA6 为浮空输入，来检测电平

```

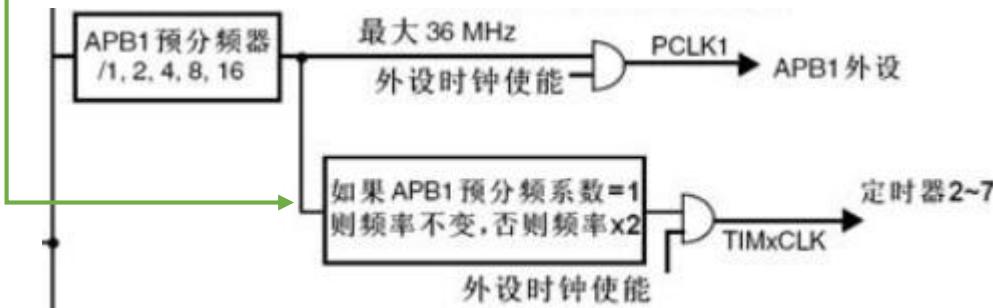
void TIM3_Mode_Init(void)
{
    TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;
    /*-----输入捕获结构体初始化-----*/
    TIM_ICInitTypeDef TIM_ICInitStructure;

    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3, ENABLE); // 2~7定时器都挂在APB1总线，开启定时器时钟，即内部时钟 CK_INT=72M
    TIM_TimeBaseStructure.TIM_Period=0xffff; // 自动重装载寄存器的值，累计 TIM_Period+1 个频率后产生一个更新或者中断
    TIM_TimeBaseStructure.TIM_Prescaler= (72-1); // 驱动 CNT 计数器的时钟 = Fck_int/(psc+1)
    TIM_TimeBaseStructure.TIM_ClockDivision=TIM_CKD_DIV1; // 时钟分频因子，配置死区时间时需要用到，不使用死区时间也要保留代码
    TIM_TimeBaseStructure.TIM_CounterMode=TIM_CounterMode_Up; // 计数器计数模式，设置为向上计数
    TIM_TimeBaseStructure.TIM_RepetitionCounter=0; // 重复计数器的值，没用到不用管
    TIM_TimeBaseInit(TIM3, &TIM_TimeBaseStructure); // 初始化定时器3

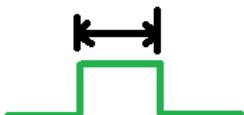
    TIM_ICInitStructure.TIM_Channel = TIM_Channel_1; // 配置输入捕获的通道，需要根据具体的 GPIO 来配置
    TIM_ICInitStructure.TIM_ICPolarity = TIM_ICPolarity_Rising; // 输入捕获信号的极性配置
    TIM_ICInitStructure.TIM_ICSelection = TIM_ICSelection_DirectTI; // 输入通道和捕获通道的映射关系，有直连和非直连两种
    TIM_ICInitStructure.TIM_ICPrescaler = TIM_ICPSC_DIV1; // 输入的需要被捕获的信号的分频系数
    TIM_ICInitStructure.TIM_ICFilter = 0; // 输入的需要被捕获的信号的滤波系数
    TIM_ICInit(TIM3, &TIM_ICInitStructure); // 定时器输入捕获初始化

    TIM_ClearFlag(TIM3, TIM_FLAG_Update|TIM_IT_CC1); // 清除更新和捕获中断标志位
    TIM_ITConfig (TIM3, TIM_IT_Update | TIM_IT_CC1, ENABLE ); // 开启更新和捕获中断
    TIM_Cmd(TIM3, ENABLE); // 使能计数器
}

```



最小采集脉宽为1us，因为你的计数器最小是1us，所以你计数脉冲宽度是按照1us，每次增加1us计算的



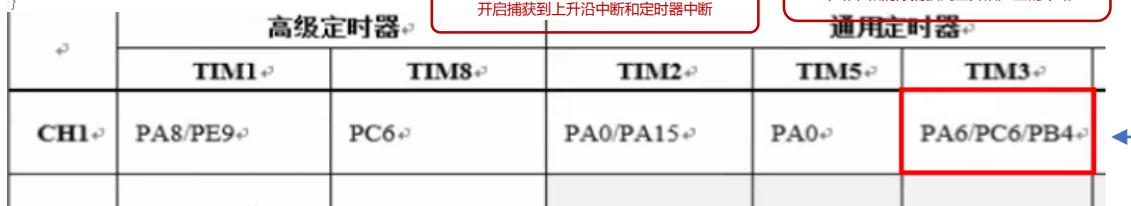
所以你想采集 500ns 的脉宽是不行的，你的脉宽必须是 1us 以上的。

```

TIM_ICInitStructure.TIM_Channel = TIM_Channel_1; // 配置输入捕获的通道，需要根据具体的 GPIO 来配置
TIM_ICInitStructure.TIM_ICPolarity = TIM_ICPolarity_Rising; // 输入捕获信号的极性配置
TIM_ICInitStructure.TIM_ICSelection = TIM_ICSelection_DirectTI; // 输入通道和捕获通道的映射关系，有直连和非直连两种
TIM_ICInitStructure.TIM_ICPrescaler = TIM_ICPSC_DIV1; // 输入的需要被捕获的信号的分频系数
TIM_ICInitStructure.TIM_ICFilter = 0; // 输入的需要被捕获的信号的滤波系数
TIM_ICInit(TIM3, &TIM_ICInitStructure); // 定时器输入捕获初始化

TIM_ClearFlag(TIM3, TIM_FLAG_Update | TIM_IT_CC1); // 清除更新和捕获中断标志位
TIM_ITConfig (TIM3, TIM_IT_Update | TIM_IT_CC1, ENABLE ); // 开启更新和捕获中断
TIM_Cmd(TIM3, ENABLE); // 使能计数器
}

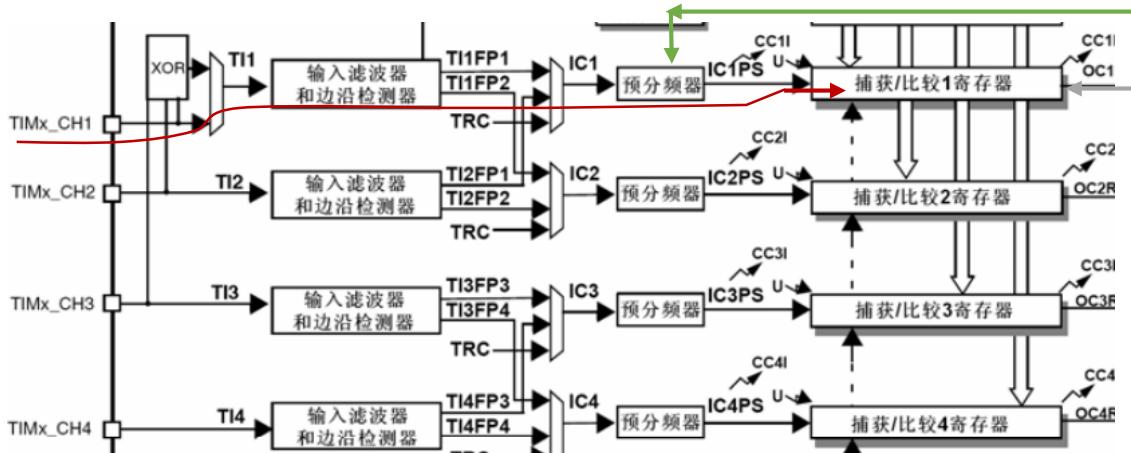
```



PA6 用的是通道 1



第 1 次捕获就从信号的第一次上升沿开始捕获，捕获到第 1 个上升沿就开始进入中断。



我们 PA6 捕获的信号直接连接到捕获/比较器 1

```

void TIM3_NVIC_Init()
{
    NVIC_InitTypeDef NVIC_InitStructure;

    // 设置中断组为 0
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_0);
    // 设置中断来源
    NVIC_InitStructure.NVIC_IRQChannel = TIM3_IRQn;
    // 设置主优先级为 0
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
    // 设置抢占优先级为 3
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 3;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);
}

```

为什么只设置 TIM3 的中断优先级，不设置输入捕获的中断优先级呢？这是因为输入捕获中断优先级也在定时器里面，然后在中断函数里面用 if 去判断哪种中断发生了

```

void TIM3_Mode_Init(void)
{
    TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;

    /*-----输入捕获结构体初始化-----*/
    TIM_ICInitTypeDef TIM_ICInitStructure;

    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3,ENABLE); // 2~7 定时器都挂在 APB1 总线,开启定时器时钟,即内部时钟 CK_INT=72M

    TIM_TimeBaseStructure.TIM_Period=0xffff; // 自动重装载寄存器的值, 累计 TIM_Period+1 个频率后产生一个更新或者中断

    TIM_TimeBaseStructure.TIM_Prescaler=(72-1); // 驱动 CNT 计数器的时钟 = Fck_int/(psc+1)

    TIM_TimeBaseStructure.TIM_ClockDivision=TIM_CKD_DIV1; // 时钟分频因子 , 配置死区时间时需要用到,不使用死区时间也要保留代码

    TIM_TimeBaseStructure.TIM_CounterMode=TIM_CounterMode_Up; // 计数器计数模式, 设置为向上计数

    TIM_TimeBaseStructure.TIM_RepetitionCounter=0; // 重复计数器的值, 没用到不用管

    TIM_TimeBaseInit(TIM3, &TIM_TimeBaseStructure); // 初始化定时器 3

    TIM_ICInitStructure.TIM_Channel = TIM_Channel_1; // 配置输入捕获的通道, 需要根据具体的 GPIO 来配置

    TIM_ICInitStructure.TIM_ICPolarity = TIM_ICPolarity_Rising; // 输入捕获信号的极性配置

    TIM_ICInitStructure.TIM_ICSelection = TIM_ICSelection_DirectTI; // 输入通道和捕获通道的映射关系, 有直连和非直连两种

    TIM_ICInitStructure.TIM_ICPrescaler = TIM_ICPSC_DIV1; // 输入的需要被捕获的信号的分频系数

    TIM_ICInitStructure.TIM_ICFilter = 0; // 输入的需要被捕获的信号的滤波系数

    TIM_ICInit(TIM3, &TIM_ICInitStructure); // 定时器输入捕获初始化

    TIM_ClearFlag(TIM3, TIM_FLAG_Update | TIM_IT_CC1); // 清除更新和捕获中断标志位

    TIM_ITConfig (TIM3, TIM_IT_Update | TIM_IT_CC1, ENABLE); // 开启更新和捕获中断

    TIM_Cmd(TIM3, ENABLE); // 使能计数器
}

void TIM3_NVIC_Init()
{
    NVIC_InitTypeDef NVIC_InitStructure;

    // 设置中断组为 0
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_0);
    // 设置中断来源
    NVIC_InitStructure.NVIC IRQChannel = TIM3_IRQn;
    // 设置主优先级为 0
    NVIC_InitStructure.NVIC IRQChannelPreemptionPriority = 0;
    // 设置抢占优先级为 3
    NVIC_InitStructure.NVIC IRQChannelSubPriority = 3;
    NVIC_InitStructure.NVIC IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);
}

```

```

uint16_t Capture_Period = 0; //自动重装载寄存器标准位
uint16_t Capture_CcrValue = 0; //捕获寄存器值
uint8_t Capture_StartFlag = 0; //捕获开始标志位
uint8_t Capture_FinishFlag = 0; //捕获结束标志位

```

```

void TIM3_IRQHandler(void)
{
    // 当要被捕获的信号的周期大于定时器的最长定时，定时器就会溢出，产生更新中断
    // 这个时候我们需要把这个最长的定时周期加到捕获信号的时间里面去
    if (TIM_GetITStatus(TIM3, TIM_IT_Update) != RESET) {
        Capture_Period++;
        TIM_ClearITPendingBit(TIM3, TIM_FLAG_Update);
    }

    // 上升沿捕获中断
    if (TIM_GetITStatus(TIM3, TIM_IT_CC1) != RESET) {
        // 第一次捕获
        if (Capture_StartFlag == 0) {
            // 计数器清 0
            TIM_SetCounter(TIM3, 0);
            // 自动重装载寄存器更新标志清 0
            Capture_Period = 0;
            // 存捕获比较寄存器的值的变量的值清 0
            Capture_CcrValue = 0;
            // 当第一次捕获到上升沿之后，就把捕获边沿配置为下降沿
            TIM_OC1PolarityConfig(TIM3, TIM_ICPolarity_Falling);
            // 开始捕获标准置 1
            Capture_StartFlag = 1;
        }
        // 下降沿捕获中断
        else { // 第二次捕获
            // 获取捕获比较寄存器的值，这个值就是捕获到的高电平的时间的值
            Capture_CcrValue = TIM_GetCapture1(TIM3);
        }
    }
}

```

如果是第1个脉宽上升沿，马上将定时器清0，开始从新计数，这里的重新计数就是计算脉宽时间了

第2次捕获中断产生，就会去执行这句else

如果PA6引脚发送上升沿，那就判断PA6引脚CH1通道1是否发送上升沿

在脉宽高电平的时候，改变捕获中断为下降沿触发

如果定时器时间到，脉宽太长还没有出现下降沿，就把这次定时时间写入变量

定时器最大65535时间线

有些脉宽太长超出了定时器计数范围

清除捕获中断标志位



```

int main(void)
{
    uint32_t time;
    uint32_t TIM_PscCLK = 72000000 / (72); // 计数器时钟是1M，也就是1us计数一次
    RCC_Configuration(); // 初始化时钟
    USART_Config(115200); // 初始化串口

    Input_GPIO_Init(); // PA6输入捕获IO口初始化
    TIM3_Mode_Init(); // 定时器和输入捕获通道CH1初始化
    TIM3_NVIC_Init(); // TIM3中断优先级

    delay_ms(1000);
    printf("xxxxxxxx\r\n");

    while(1)
    {
        if (Capture_FinishFlag == 1) {
            // 计算高电平时间的计数器的值
            time = Capture_Period * (0xffff + 1) + (Capture_CcrValue + 1);
            printf(" high level time length: %.d.%d s\r\n", time/TIM_PscCLK, time%TIM_PscCLK);
            Capture_FinishFlag = 0;
        }
    }
    return 0;
}

```

第二次下降沿，FinishFlag 置 1

那么脉宽长度就是 CcrValue 的值

如果脉宽是在 65535 范围内 Period 是不会被+1 的，那么就是 0

计数器值/1M 就是高电平时间

```

void TIM3_IRQHandler(void)
{
    // 当要被捕获的信号的周期大于定时器的最长定时，定时器就会溢出，产生更新中断
    // 这个时候我们需要把这个最长的定时周期加到捕获信号的时间里面去
    if( TIM_GetITStatus ( TIM3, TIM_IT_Update ) != RESET ){
        Capture_Period++;
        TIM_ClearITPendingBit ( TIM3, TIM_FLAG_Update );
    }

    // 上升沿捕获中断
    if( TIM_GetITStatus ( TIM3, TIM_IT_CC1 ) != RESET ){
        // 第一次捕获
        if ( Capture_StartFlag == 0 ){
            // 计数器清 0
            TIM_SetCounter ( TIM3, 0 );
            // 自动重装载寄存器更新标志清 0
            Capture_Period = 0;
            // 存捕获比较寄存器的值的变量的值清 0
            Capture_CcrValue = 0;

            // 当第一次捕获到上升沿之后，就把捕获边沿配置为下降沿
            TIM_OC1PolarityConfig(TIM3, TIM_ICPolarity_Falling);
            // 开始捕获标准置 1
            Capture_StartFlag = 1;
        }
        // 下降沿捕获中断
        else { // 第二次捕获
            // 获取捕获比较寄存器的值，这个值就是捕获到的高电平的时间的值
            Capture_CcrValue = TIM_GetCapture1(TIM3);

            // 当第二次捕获到下降沿之后，就把捕获边沿配置为上升沿，好开启新一轮捕获
            TIM_OC1PolarityConfig(TIM3, TIM_ICPolarity_Rising);
            // 开始捕获标志清 0
            Capture_StartFlag = 0;
            // 捕获完成标志置 1
            Capture_FinishFlag = 1;
        }
        TIM_ClearITPendingBit (TIM3,TIM_IT_CC1);
    }
}

int main(void)
{
    uint32_t time;

    uint32_t TIM_PscCLK = 72000000 / (72); // 计数器时钟是 1M，也就是 1us 计数一次

    RCC_Configuration(); // 初始化时钟
    USART_Config(115200); // 初始化串口

    Input_GPIO_Init(); // PA6 输入捕获 IO 口初始化
    TIM3_Mode_Init(); // 定时器和输入捕获通道 CH1 初始化
    TIM3_NVIC_Init(); // TIM3 中断优先级

    delay_ms(1000);
    printf("xxxzzzz\r\n");

    while(1)
    {
        if (Capture_FinishFlag == 1) {
            // 计算高电平时间的计数器的值
            time = Capture_Period * (0xffff+1) +(Capture_CcrValue+1);
            Capture_FinishFlag = 0;

            printf (" hight level time length: %d.%d s\r\n", time/TIM_PscCLK, time%TIM_PscCLK );
        }
    }
    return 0;
}

}

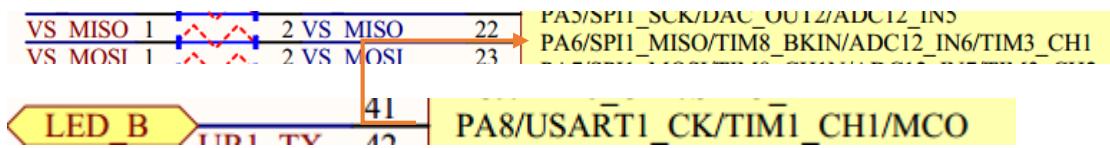
```

hight level time length: 0.3412 s
hight level time length: 0.4 s
hight level time length: 0.1 s
hight level time length: 0.4114 s
hight level time length: 0.3412 s
hight level time length: 0.7 s
hight level time length: 0.12 s
hight level time length: 0.4208 s
hight level time length: 0.11 s
hight level time length: 0.12 s
hight level time length: 0.3918 s
hight level time length: 0.4158 s
hight level time length: 0.3393 s

这就是每次 PA6 接受脉冲宽度的时间，

我发现直接给 PA6 3.3V 电压是不行的，貌似给 2.5V 是可以的，可能是 PA6 引脚悬空的问题

输入捕获，采集 PWM 信号，测量频率，占空比



我们用 TIM3 定时器来实验，看看 TIM3 定时器缺陷在哪里

```
void Input_GPIO_Init(void) //PA6作为输入捕获引脚
{
    GPIO_InitTypeDef GPIO_InitStructure;
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
    GPIO_Init(GPIOA, &GPIO_InitStructure);
}

static void PWM_IO_Init(void) //PA8 PWM信号输出引脚
{
    GPIO_InitTypeDef GPIO_InitStructure;

    // 输出比较通道 GPIO 初始化
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE); // 使用PA8输出PWM
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOA, &GPIO_InitStructure);
}
```

引脚初始化，PA6 捕获引脚设置为浮空输入，PA8 PWM 输出引脚设置为复用输出

```
static void TIM1_PWM_Mode(void) //TIM1设置 PWM输出100Khz方波，占空比50%
{
    /*-----时基结构体初始化-----*/
    TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;
    /*-----输出比较结构体初始化-----*/
    TIM_OCInitTypeDef TIM_OCInitStructure;
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_TIM1, ENABLE); // 开启定时器时钟，即内部时钟 CK_INT=72MHz
    TIM_TimeBaseStructure.TIM_Period=(10-1); // 自动重装载寄存器的值，改成100Khz PWM输出
    TIM_TimeBaseStructure.TIM_Prescaler= (72-1); // 驱动 CNT 计数器的时钟 = Fck_int/(psc+1) 改成100Khz PWM输出
    TIM_TimeBaseStructure.TIM_ClockDivision=TIM_CKD_DIV1; // 时钟分频因子，配置死区时间时需要用到，不使用死区时间也要保留代码
    TIM_TimeBaseStructure.TIM_CounterMode=TIM_CounterMode_Up; // 计数器计数模式，设置为向上计数
    TIM_TimeBaseStructure.TIM_RepetitionCounter=0; // 重复计数器的值，没用到不用管
    TIM_TimeBaseInit(TIM1, &TIM_TimeBaseStructure); // 初始化定时器
    TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1; // 配置为 PWM 模式 1
    TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable; // PA8输出使能
    TIM_OCInitStructure.TIM_OutputNState = TIM_OutputNState_Enable; // PB13互补输出使能
    TIM_OCInitStructure.TIM_Pulse = 5; // 设置占空比大小
    TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_High; // 输出通道电平极性配置
    TIM_OC1Init(TIM1, &TIM_OCInitStructure);

    // 使能计数器
    TIM_Cmd(TIM1, ENABLE);
    // 主输出使能，当使用的是通用定时器时，这句不需要
    TIM_CtrlPWMOutputs(TIM1, ENABLE);
}
```

设置之后 PWM 正常输出 100Khz 方波，50%占空比

```

void TIM3_Mode_Init(void) //TIM3定时器设置输入捕获功能
{
    TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;
    /*-----输入捕获结构体初始化-----*/
    TIM_ICInitTypeDef TIM_ICInitStructure;

    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3, ENABLE); // 2~7定时器都挂在APB1总线, 开启定时器时钟, 即内部时钟 CK_INT=72M

    TIM_TimeBaseStructure.TIM_Period=(1000-1); // 自动重装载寄存器是计数1000us才溢出, 也就是1ms溢出
    //这样周期小于1ms的PWM信号都能捕获比如100khz方波

    TIM_TimeBaseStructure.TIM_Prescaler= (72-1); // 计数器是72M/72=1M的时钟, 也就是1us计数一次
    TIM_TimeBaseStructure.TIM_ClockDivision=TIM_CKD_DIV1; // 时钟分频因子 , 配置死区时间时需要用到, 不使用死区时间也要保留代码
    TIM_TimeBaseStructure.TIM_CounterMode=TIM_CounterMode_Up; // 计数器计数模式, 设置为向上计数
    TIM_TimeBaseStructure.TIM_RepetitionCounter=0; // 重复计数器的值, 没用到不用管

    TIM_TimeBaseInit(TIM3, &TIM_TimeBaseStructure); // 初始化定时器3

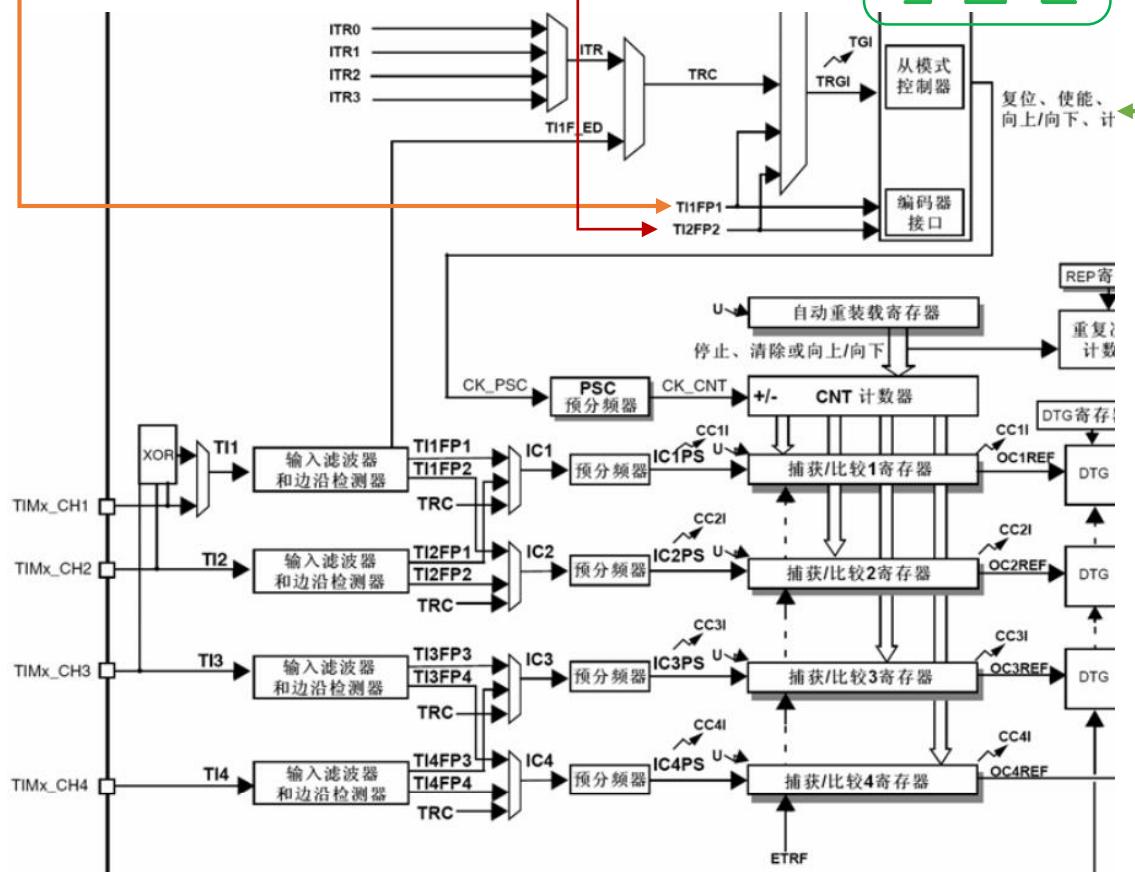
    TIM_ICInitStructure.TIM_Channel = TIM_Channel_1;// 配置输入捕获的通道, 需要根据具体的 GPIO 来配置
    TIM_ICInitStructure.TIM_ICPolarity = TIM_ICPolarity_Rising; // 输入捕获信号的极性配置, 上升沿开始捕获
    TIM_ICInitStructure.TIM_ICSelection = TIM_ICSelection_DirectTI; // 输入通道和捕获通道的映射关系, 有直连和非直连两种
    TIM_ICInitStructure.TIM_ICPrescaler = TIM_ICPSC_DIV1; // 输入的需要被捕获的信号的分频系数
    TIM_ICInitStructure.TIM_ICFilter = 0; // 输入的需要被捕获的信号的滤波系数

    TIM_PWMConfig(TIM3, &TIM_ICInitStructure); // 定时器输入捕获初始化改为PWM输入捕获初始化模式
}

// PWM 输入模式时, 从模式必须工作在复位模式, 当捕获开始时, 计数器 CNT 会被复位
TIM_SelectInputTrigger(TIM3, TIM_TS_TI1FP1); //输入捕获触发信号通道选择, 就是上升沿来了开始触发, 计算为周期
TIM_SelectSlaveMode(TIM3, TIM_SlaveMode_Reset); //选择从模式: 复位模式
// PWM 输入模式时, 从模式必须工作在复位模式, 当捕获开始时, 计数器 CNT 会被复位
TIM_SelectMasterSlaveMode(TIM3, TIM_MasterSlaveMode_Enable);

TIM_ClearITPendingBit(TIM3, TIM_IT_CC1); // 清除 CH1 中断标志位
TIM_ITConfig (TIM3, TIM_IT_Update | TIM_IT_CC1, ENABLE ); // 开启更新和捕获中断
TIM_Cmd(TIM3, ENABLE); // 使能计数器
}

```



```

void TIM3_IRQHandler(void)
{
    /* 清除中断标志位 */
    TIM_ClearITPendingBit(TIM3, TIM_IT_CC1);

    /* 获取输入捕获值 */
    IC1Value = TIM_GetCapture1(TIM3);
    IC2Value = TIM_GetCapture2(TIM3);

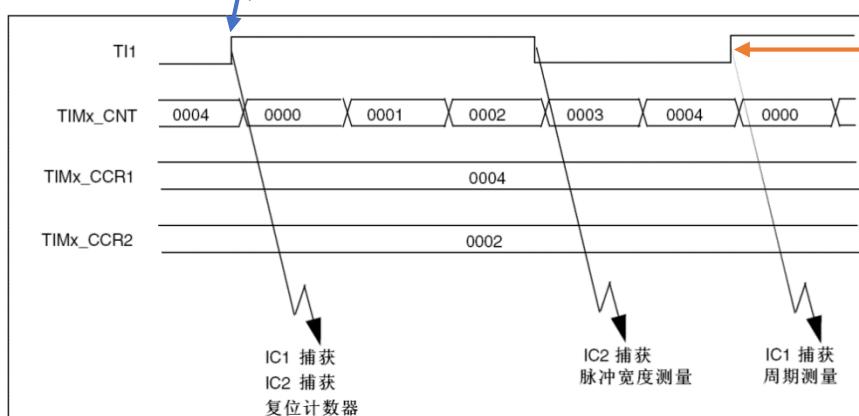
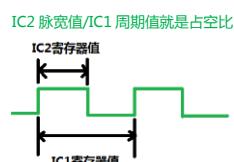
    // 注意：捕获寄存器 CCR1 和 CCR2 的值在计算占空比和频率的时候必须加 1
    if (IC1Value != 0) {
        /* 占空比计算 */
        DutyCycle = (float)((IC2Value+1) * 100) / (IC1Value+1);

        /* 频率计算 */
        Frequency = (72000000/9)/(float)(IC1Value+1);
        printf("DutyCycle: %0.2f%% Frequency: %0.2fHz\r\n", DutyCycle, Frequency);
    }
    else {
        DutyCycle = 0;
        Frequency = 0;
    }
}

```

第 1 个上升沿 IC1,IC2 寄存器清 0，所以第 1 个中断，IC1,IC2 读取的值为 0

第 2 个上升沿，产生中断，这时候 IC1 寄存器里面有值，IC2 寄存器就有高电平脉宽值，读取出来



```

int main(void)
{
    RCC_Configuration(); // 初始化时钟
    USART_Config(115200); // 初始化串口

    PWM_IO_Init(); // PWM IO 口初始化
    TIM1_PWM_Mode(); // PWM 定时器初始化

    Input_GPIO_Init(); // PA6 输入捕获 IO 口初始化
    TIM3_Mode_Init(); // 定时器和输入捕获通道 CH1 初始化
    TIM3_NVIC_Init(); // TIM3 中断优先级
}

```

使用的时候一定是 PWM 先初始化，然后再初始化捕获，否则先初始化捕获就有可能一直进入中断，后面程序就不执行了，这里只是猜测

```

delay_ms(1000);
printf("xxxxzzzz\r\n");

while(1)
{
}

return 0;
}

```

根据输出结果，占空比是没有问题的，但是频率是错误的，我是 100kHz 的方波，怎么采集出来是 800kHz

```

DutyCycle: 50.00% Frequency: 800000.00Hz

```

难道 TIM3 不适合做频率采集，必须 TIM1 高级定时器才可以吗

下面贴出全代码

```
void Input_GPIO_Init(void)//PA6 作为输入捕获引脚
{
    GPIO_InitTypeDef GPIO_InitStructure;
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
    GPIO_Init(GPIOA, &GPIO_InitStructure);
}

void TIM3_Mode_Init(void)//TIM3 定时器设置输入捕获功能
{
    TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;

    /*-----输入捕获结构体初始化-----*/
    TIM_ICInitTypeDef TIM_ICInitStructure;

    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3,ENABLE);//2~7 定时器都挂在 APB1 总线,开启定时器时钟,即内部时钟 CK_INT=72M

    TIM_TimeBaseStructure.TIM_Period=(1000-1);// 自动重装载寄存器是计数 1000us 才溢出, 也就是 1ms 溢出
    //这样周期小于 1ms 的 PWM 信号都能捕获比如 100khz 方波

    TIM_TimeBaseStructure.TIM_Prescaler=(72-1);// 计数器是 72M/72=1M 的时钟, 也就是 1us 计数一次

    TIM_TimeBaseStructure.TIM_ClockDivision=TIM_CKD_DIV1;// 时钟分频因子 , 配置死区时间时需要用到,不使用死区时间也要保留代码

    TIM_TimeBaseStructure.TIM_CounterMode=TIM_CounterMode_Up;// 计数器计数模式, 设置为向上计数

    TIM_TimeBaseStructure.TIM_RepetitionCounter=0;// 重复计数器的值, 没用到不用管

    TIM_TimeBaseInit(TIM3, &TIM_TimeBaseStructure);// 初始化定时器 3

    TIM_ICInitStructure.TIM_Channel = TIM_Channel_1;// 配置输入捕获的通道, 需要根据具体的 GPIO 来配置

    TIM_ICInitStructure.TIM_ICPolarity = TIM_ICPolarity_Rising;// 输入捕获信号的极性配置, 上升沿开始捕获

    TIM_ICInitStructure.TIM_ICSelection = TIM_ICSelection_DirectTl;// 输入通道和捕获通道的映射关系, 有直连和非直连两种

    TIM_ICInitStructure.TIM_ICPrescaler = TIM_ICPSC_DIV1;// 输入的需要被捕获的信号的分频系数

    TIM_ICInitStructure.TIM_ICFilter = 0;// 输入的需要被捕获的信号的滤波系数

    TIM_PWMConfig(TIM3, &TIM_ICInitStructure);// 定时器输入捕获初始化改为 PWM 输入捕获初始化模式

    TIM_SelectInputTrigger(TIM3, TIM_TS_TI1FP1);//输入捕获触发信号通道选择, 就是上升沿来了开始触发, 计算为周期
    TIM_SelectSlaveMode(TIM3, TIM_SlaveMode_Reset);//选择从模式: 复位模式

    // PWM 输入模式时,从模式必须工作在复位模式, 当捕获开始时,计数器 CNT 会被复位
    TIM_SelectMasterSlaveMode(TIM3,TIM_MasterSlaveMode_Enable);

    TIM_ClearITPendingBit(TIM3, TIM_IT_CC1);

    TIM_ITConfig (TIM3, TIM_IT_Update | TIM_IT_CC1, ENABLE );// 开启更新和捕获中断

    TIM_Cmd(TIM3, ENABLE);// 使能计数器
}

void TIM3_NVIC_Init()//TIM3 中断优先级
{
    NVIC_InitTypeDef NVIC_InitStructure;

    // 设置中断组为 0
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_0);
    // 设置中断来源
    NVIC_InitStructure.NVIC IRQChannel = TIM3_IRQn ;
    // 设置主优先级为 0
    NVIC_InitStructure.NVIC IRQChannelPreemptionPriority = 0;
    // 设置抢占优先级为 3
    NVIC_InitStructure.NVIC IRQChannelSubPriority = 3;
    NVIC_InitStructure.NVIC IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);
}

uint16_t IC1Value = 0 ,IC2Value = 0;
float DutyCycle = 0;
float Frequency = 0;

void TIM3_IRQHandler(void)
{
    /* 清除中断标志位 */
    TIM_ClearITPendingBit(TIM3, TIM_IT_CC1);

    /* 获取输入捕获值 */
    IC1Value = TIM_GetCapture1(TIM3);
    IC2Value = TIM_GetCapture2(TIM3);

    // 注意: 捕获寄存器 CCR1 和 CCR2 的值在计算占空比和频率的时候必须加 1

    if (IC1Value != 0) {
        /* 占空比计算 */
        DutyCycle = (float)((IC2Value+1) * 100) / (IC1Value+1);

        /* 频率计算 */
        Frequency = (72000000/9)/(float)(IC1Value+1);
        printf("DutyCycle: %0.2f%% Frequency: %0.2fHz\r\n",DutyCycle,Frequency);
    }
}
```

```

        else {
            DutyCycle = 0;
            Frequency = 0;
        }
    }

static void PWM_IO_Init(void)//PA8 PWM 信号输出引脚
{
    GPIO_InitTypeDef GPIO_InitStructure;

    // 输出比较通道 GPIO 初始化
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE); // 使用 PA8 输出 PWM
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOA, &GPIO_InitStructure);
}

static void TIM1_PWM_Mode(void) //TIM1 设置 PWM 输出 100Khz 方波，占空比 50%
{
    /*-----时基结构体初始化-----*/
    TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;

    /*-----输出比较结构体初始化-----*/
    TIM_OCInitTypeDef TIM_OCInitStructure;

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_TIM1,ENABLE); // 开启定时器时钟,即内部时钟 CK_INT=72M

    TIM_TimeBaseStructure.TIM_Period=(10-1); // 自动重装载寄存器的值, 改成 100Khz PWM 输出
    TIM_TimeBaseStructure.TIM_Prescaler=(72-1); // 驱动 CNT 计数器的时钟 = Fck_int/(psc+1) 改成 100Khz PWM 输出
    TIM_TimeBaseStructure.TIM_ClockDivision=TIM_CKD_DIV1; // 时钟分频因子 , 配置死区时间时需要用到,不使用死区时间也要保留代码
    TIM_TimeBaseStructure.TIM_CounterMode=TIM_CounterMode_Up; // 计数器计数模式, 设置为向上计数
    TIM_TimeBaseStructure.TIM_RepetitionCounter=0; // 重复计数器的值, 没用到不用管

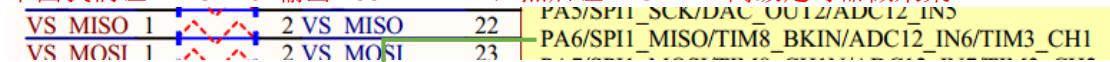
    TIM_TimeBaseInit(TIM1, &TIM_TimeBaseStructure); // 初始化定时器

    TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1; // 配置为 PWM 模式 1
    TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable; // PA8 输出使能
    TIM_OCInitStructure.TIM_OutputNState = TIM_OutputNState_Enable; // PB13 互补输出使能
    TIM_OCInitStructure.TIM_Pulse = 5; // 设置占空比大小
    TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_High; // 输出通道电平极性配置
    TIM_OC1Init(TIM1, &TIM_OCInitStructure);

    // 使能计数器
    TIM_Cmd(TIM1, ENABLE);
    // 主输出使能, 当使用的是通用定时器时, 这句不需要
    TIM_CtrlPWMOutputs(TIM1, ENABLE);
}

```

下面我们让 TIM3 PA6 输出 100Khz PWM, 然后让 PA8 TIM1 高级定时器做采集



LED_B ————— PA8/USART1_CK/TIM1_CH1/MCO

```

void Input_GPIO_Init(void) //PA8作为输入捕获引脚
{
    GPIO_InitTypeDef GPIO_InitStructure;
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
    GPIO_Init(GPIOA, &GPIO_InitStructure);
}

static void PWM_IO_Init(void) //PA6 PWM信号输出引脚
{
    GPIO_InitTypeDef GPIO_InitStructure;
    // 输出比较通道 GPIO 初始化
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA | RCC_APB2Periph_AFIO, ENABLE); // 使用PA6输出PWM
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOA, &GPIO_InitStructure);
}

```

```

static void TIM3_PWM_Mode(void) //TIM3设置 PWM输出100Khz方波，占空比50%
{
    /*-----时基结构体初始化-----*/
    TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;

    /*-----输出比较结构体初始化-----*/
    TIM_OCInitTypeDef TIM_OCInitStructure;

    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3, ENABLE); // 开启定时器时钟, 记住TIM3是在APB1总线, 即内部时钟 CK_INT=72M
    TIM_TimeBaseStructure.TIM_Period= (10-1); // 自动重装载寄存器的值, 改成100Khz PWM输出
    TIM_TimeBaseStructure.TIM_Prescaler= (72-1); // 驱动 CNT 计数器的时钟 = Fck_int/(psc+1) 改成100Khz PWM输出
    TIM_TimeBaseStructure.TIM_ClockDivision=TIM_CKD_DIV1; // 时钟分频因子 , 配置死区时间时需要用到, 不使用死区时间也要保留代码
    TIM_TimeBaseStructure.TIM_CounterMode=TIM_CounterMode_Up; // 计数器计数模式, 设置为向上计数
    //TIM_TimeBaseStructure.TIM_RepetitionCounter=0; // 重复计数器的值, 没用到不用管

    TIM_TimeBaseInit(TIM3, &TIM_TimeBaseStructure); // 初始化定时器

    // TIM_ClearITPendingBit(TIM3, TIM_IT_Update);
    // TIM_ITConfig(TIM3, TIM_IT_Update, ENABLE);

    TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1; // 配置为 PWM 模式 1
    TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable; // PA6输出使能

    TIM_OCInitStructure.TIM_Pulse = 5; // 设置占空比大小
    TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_High; // 输出通道电平极性配置

    TIM_OC1Init(TIM3, &TIM_OCInitStructure);

    // TIM_OC1PreloadConfig(TIM3, TIM_OCPreload_Enable); //使能预装载寄存器
    //
    // TIM_CtrlPWMOutputs(TIM3, ENABLE);
    // TIM_ARRPreloadConfig(TIM3, ENABLE);
    // 使能计数器
    TIM_Cmd(TIM3, ENABLE);
    // 主输出使能, 当使用的是通用定时器时, 这句不需要
    //TIM_CtrlPWMOutputs(TIM3, ENABLE);
}

void TIM1_Mode_Init(void) //TIM1高级定时器设置输入捕获功能
{
    TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;
    /*-----输入捕获结构体初始化-----*/
    TIM_ICInitTypeDef TIM_ICInitStructure;

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_TIM1, ENABLE); // 定时器1挂在APB2总线, 开启定时器时钟, 即内部时钟 CK_INT=72M
    TIM_TimeBaseStructure.TIM_Period=(1000-1); // 自动重装载寄存器是计数1000us才溢出, 也就是1ms溢出
    //这样周期小于1ms的PWM信号都能捕获比如100khz方波

    TIM_TimeBaseStructure.TIM_Prescaler= (72-1); // 计数器是72M/72=1M的时钟, 也就是1us计数一次
    TIM_TimeBaseStructure.TIM_ClockDivision=TIM_CKD_DIV1; // 时钟分频因子 , 配置死区时间时需要用到, 不使用死区时间也要保留代码
    TIM_TimeBaseStructure.TIM_CounterMode=TIM_CounterMode_Up; // 计数器计数模式, 设置为向上计数

    TIM_TimeBaseStructure.TIM_RepetitionCounter=0; // 重复计数器的值, 没用到不用管

    TIM_TimeBaseInit(TIM1, &TIM_TimeBaseStructure); // 初始化定时器1

    TIM_ICInitStructure.TIM_Channel = TIM_Channel_1; // 配置输入捕获的通道, 需要根据具体的 GPIO 来配置
    TIM_ICInitStructure.TIM_ICPolarity = TIM_ICPolarity_Rising; // 输入捕获信号的极性配置, 上升沿开始捕获
    TIM_ICInitStructure.TIM_ICSelection = TIM_ICSelection_DirectTI; // 输入通道和捕获通道的映射关系, 有直连和非直连两种
    TIM_ICInitStructure.TIM_ICPrescaler = TIM_ICPSC_DIV1; // 输入的需要被捕获的信号的分频系数
    TIM_ICInitStructure.TIM_ICFilter = 0; // 输入的需要被捕获的信号的滤波系数

    TIM_PWMConfig(TIM1, &TIM_ICInitStructure); // 定时器输入捕获初始化改为PWM输入捕获初始化模式
    TIM_SelectInputTrigger(TIM1, TIM_TS_TI1FP1); //输入捕获触发信号通道选择, 就是上升沿来了开始触发, 计算为周期
    TIM_SelectSlaveMode(TIM1, TIM_SlaveMode_Reset); //选择从模式: 复位模式

    // PWM 输入模式时, 从模式必须工作在复位模式, 当捕获开始时, 计数器 CNT 会被复位
    TIM_SelectMasterSlaveMode(TIM1, TIM_MasterSlaveMode_Enable);

    TIM_ClearITPendingBit(TIM1, TIM_IT_CC1);

    TIM_ITConfig (TIM1, TIM_IT_Update | TIM_IT_CC1, ENABLE ); // 开启更新和捕获中断

    TIM_Cmd(TIM1, ENABLE); // 使能计数器
}

```

```

void TIM1_NVIC_Init()//TIM3中断优先级
{
    NVIC_InitTypeDef NVIC_InitStructure;

    // 设置中断组为 0
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_0);
    // 设置中断来源
    NVIC_InitStructure.NVIC_IRQChannel = TIM1_CC_IRQn;
    // 设置主优先级为 0
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
    // 设置抢占优先级为 3
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 3;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);
}

uint16_t IC1Value = 0 , IC2Value = 0;
float DutyCycle = 0;
float Frequency = 0;

void TIM1_CC_IRQHandler(void)
{
    /* 清除中断标志位 */
    TIM_ClearITPendingBit(TIM1, TIM_IT_CC1);

    /* 获取输入捕获值 */
    IC1Value = TIM_GetCapture1(TIM1);
    IC2Value = TIM_GetCapture2(TIM1);

    // 注意：捕获寄存器 CCR1 和 CCR2 的值在计算占空比和频率的时候必须加 1
    if (IC1Value != 0) {

        /* 占空比计算 */
        DutyCycle = (float)((IC2Value+1) * 100) / (IC1Value+1);

        /* 频率计算 */
        Frequency = (72000000/9)/(float)(IC1Value+1);
        printf("DutyCycle: %0.2f% Frequency: %0.2fHz\r\n", DutyCycle, Frequency);

    }
    else {
        DutyCycle = 0;
        Frequency = 0;
    }
}

int main(void)
{
    RCC_Configuration(); // 初始化时钟
    USART_Config(115200); // 初始化串口

    PWM_IO_Init(); // PWM IO口初始化
    TIM3_PWM_Mode(); // PWM定时器初始化

    Input_GPIO_Init(); // PA8输入捕获IO口初始化
    TIM1_Mode_Init(); // 定时器1和输入捕获通道CH1初始化
    TIM1_NVIC_Init(); // TIM1中断优先级

    delay_ms(1000);
    printf("xxxxxxxx\r\n");

    while(1)
    {
    }
    return 0;
}

```

DutyCycle: 50.00% Frequency: 800000.00Hz
 DutyCycle: 50.00% Frequency: 800000.00Hz

输出结果还是频率没对，占空比正确


```

result = f_mount(&fsp, "0:", 1);
if (result == FR_OK) {
    printf("SD card mount success!!!->%d\r\n", result);
}
else
{
    printf("SD card mount failed!!!->%d\r\n", result);
}

result=f_open(&fp, "0:DBFS.wav", FA_WRITE|FA_READ);
if(result == FR_OK)
{
    printf("file open success\r\n");
    result = f_stat("0:DBFS.wav",&info);
    printf("file size = %ld\r\n", info.fsize);

    f_read(&fp, TempBuf, 1024, &br);
    riff=(ChunkRIFF *)TempBuf;
    printf("br = %d\r\n", br);
    printf("riff Format = %x\r\n", riff->Format); //wav文件0x45564157
    if(riff->Format==0X45564157)
    {
        fmt=(ChunkFMT *) (TempBuf+48);
        printf("fmt chunkID = %x\r\n", fmt->ChunkID);
        printf("fmt ChunkSize = %x\r\n", fmt->ChunkSize);
        printf("fmt AudioFormat = %x\r\n", fmt->AudioFormat);
        printf("fmt NumOfChannels = %x\r\n", fmt->NumOfChannels);
        printf("fmt SampleRate = %x\r\n", fmt->SampleRate);
        printf("fmt ByteRate = %x\r\n", fmt->ByteRate);
        printf("fmt BlockAlign = %x\r\n", fmt->BlockAlign);
        printf("fmt BitsPerSample = %x\r\n", fmt->BitsPerSample);
    }
}

```

52	49	46	46	12	CC	AF	00	57	41	56	45	4A	55	4E	4B
1C	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
66	6D	74	20	10	00	00	00	01	00	02	00	80	BB	00	00
00	EE	02	00	04	00	10	00	64	61	74	61	00	C8	AF	00
00	00	00	00	AC	01	AC	01	50	03	50	03	E6	04	E7	04

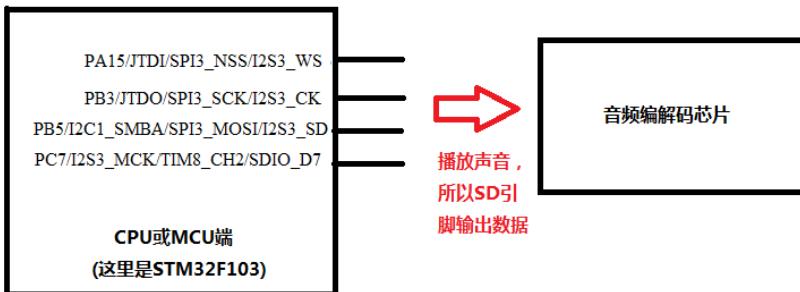
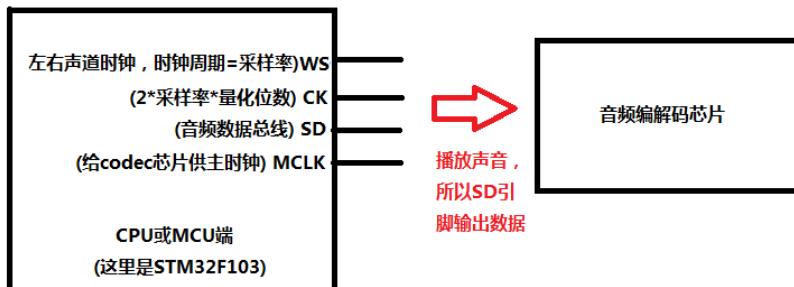
你看绕过了 52 开始到最后 00 才能解析到 66 6D 74 20

main 函数调用其它 C 文件程序的空函数，出现内存溢出问题

<pre> main() { /*这里已经写了很多占用内存的代码了*/ /////////////////////////////// subfuc() //我这里调用另外文件的subfunc程序 } </pre> <p>文件1</p>	<pre> static uint16_t TABLE[58]= //我静态申请了一个比较大的数组 { 0X0000,0X0000,0X0000,0X0000,0X0050,0X0000,0X0140,0X0000, 0X0000,0X0000,0X0000,0X00FF,0X00FF,0X0000,0X0100,0X00FF, 0X00FF,0X0000,0X012C,0X002C,0X002C,0X002C,0X0000, 0X0032,0X0000,0X0000,0X0000,0X0000,0X0000,0X0000,0X0000, 0X0038,0X000B,0X0032,0X0000,0X0008,0X000C,0X0093,0X00E9, 0X0000,0X0000,0X0000,0X0000,0X0003,0X0010,0X0010,0X0100, 0X0100,0X0002,0X0001,0X0001,0X0039,0X0039,0X0039,0X0039, 0X0001,0X0001 }; subfuc() //我的subfunc只是一个空函数 { } </pre> <p>文件2</p>
--	--

这时候就算主程序调用文件2的subfuc()空函数，
也会出现内存泄漏，因为
文件2的TABLE被初始化
在内存了，因为是static
的数组

STM32F103 I2S3 输出音频数据波形实现



所以我要初始化
PA15,PB3,PB5,PC7引脚

```
//I2S2 初始化
//参数 I2S_Standard: @ref SPI_I2S_Standard I2S 标准,
//I2S_Standard_Phillips,飞利浦标准;
//I2S_Standard_MSB,MSB 对齐标准(右对齐);
//I2S_Standard_LSB,LSB 对齐标准(左对齐);
//I2S_Standard_PCMShort,I2S_Standard_PCMLong:PCM 标准
//参数 I2S_Mode: @ref SPI_I2S_Mode I2S_Mode_SlaveTx:从机发送;I2S_Mode_SlaveRx:从机接收;I2S_Mode_MasterTx:主机发送;I2S_Mode_MasterRx:主机接收;
//参数 I2S_Clock_Polarity @ref SPI_I2S_Clock_Polarity: I2S_CPOL_Low,时钟低电平有效;I2S_CPOL_High,时钟高电平有效
//参数 I2S_DataFormat: @ref SPI_I2S_Data_Format :数据长度,I2S_DataFormat_16b,16 位标准;I2S_DataFormat_16bextended,16 位扩展(frame=32bit);I2S_DataFormat_24b,24 位;I2S_DataFormat_32b,32 位.
void I2S3_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    I2S_InitTypeDef I2S_InitStructure;

    /* Enable GPIOB, GPIOC and AFIO clock */
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA | RCC_APB2Periph_GPIOB | RCC_APB2Periph_GPIOC |
                           RCC_APB2Periph_AFIO , ENABLE);

    /* I2S3 SD, CK pins configuration */
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_3 | GPIO_Pin_5;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
    GPIO_Init(GPIOB, &GPIO_InitStructure);

    /* I2S3 WS pins configuration */
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_15;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    /* I2S3 MCLK pins configuration */
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_7;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
    GPIO_Init(GPIOC, &GPIO_InitStructure);

    RCC_APB1PeriphResetCmd(RCC_APB1Periph_SPI3, ENABLE); //复位 SPI3
    RCC_APB1PeriphResetCmd(RCC_APB1Periph_SPI3, DISABLE); //结束复位
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_SPI3, ENABLE); //使能 SPI3 时钟,I2S3 时钟在 SPI3
    GPIO.PinRemapConfig(GPIO_Remap_SWJ_JTAGDisable,ENABLE); //关闭 JTAG 调试功能
}
```

初始化 SD 和 CK 引脚为复用输出模式

初始化 WS 和 MCLK 引脚为复用输出模式

```

    SPI_I2S_DeInit(SPI3);//先恢复 SPI3 寄存器默认值

    I2S_InitStructure.I2S_Mode=I2S_Mode_MasterTx;//IIS 主机发送，从机接受
    I2S_InitStructure.I2S_Standard=I2S_Standard_Phillips;//IIS 标准
    I2S_InitStructure.I2S_DataFormat=I2S_DataFormat_16b;//IIS 数据长度
    I2S_InitStructure.I2S_MCLKOutput=I2S_MCLKOutput_Enable;//主时钟输出打开
    I2S_InitStructure.I2S_AudioFreq=I2S_AudioFreq_48k;//IIS 频率设置(采样率 48K)
    I2S_InitStructure.I2S_CPOL=I2S_CPOL_Low;//空闲状态时钟电平
    I2S_Init(SPI3,&I2S_InitStructure);//初始化 IIS

    I2S_Cmd(SPI3,ENABLE);//SPI3 I2S3 EN 使能.

}

main()
{
    I2S3_Init();//在主程序初始化 I2S3 的模式

    while(1)
    {
        SPI_I2S_SendData(SPI3, 0x00ff);//一定要用 SPI_I2S_SendData 向 SD 引脚不停的发送数据，那么 WS,CK,MLCK 时钟才会输出，如果停止使用 SPI_I2S_SendData，那么 WS,CK,MLCK 时钟就不会输出
    }
}

```

WAV 文件播放实现，结合 wav 文件解析第 2 种方法文档和 STM32F103 I2S3 输出音频数据波形实现文档，完成 wav 播放功能

我现在是明确知道 wav 文件是 48000hz 采样率和 16 位数据宽度(采样位宽)，进行 wav 播放设计

```

#include "uartprintf.h"
#include "stdio.h"
#include "ff.h"
#include "sdecode.h"
#include "wavfmt.h"
#include "wm8978.h"
#include "i2s.h"

FATFS fsp;//文件系统使用要先有这个句柄，这个句柄很大所以定义在全局变量
//extern SD_CardInfo SDCardInfo;

#define WAVEFILEBUFSIZE      2048
FIL fp;
uint8_t WaveFileBuf[WAVEFILEBUFSIZE];//2048内存

int main(void)
{
    uint8_t TempBuf[1024];
    int result = 0, i = 0, x = 0;
    uint32_t br=0;
    uint8_t res=0;
    ChunkRIFF *riff;
    ChunkFMT *fmt;
    ChunkFACT *fact;
    ChunkDATA *data;
    uint8_t Hdata = 0;
    uint8_t Ldata = 0;
    uint8_t HLdata = 0;
    FILINFO info;
    RCC_configuration();//初始化时钟
    USART_config(I15200);//初始化串口
    I2S3_Init();//MCU端I2S初始化

    res = WM8978_Init();//播放前确保WM8978音频芯片I2S初始化成功
    printf("wm8978 reset = %d\r\n",res);

    delay_ms(3000);

    result = f_mount(&fsp, "0:", 1);
    if (result == FR_OK) {
        printf("SD card mount success!!!->%d\r\n",result);
    }
    else {
        printf("SD card mount failed!!!->%d\r\n",result);
    }
}

```

定义需要的数据结构

初始化 I2S，挂载文件系统

我们就是使用 **00 C8 AF 00** 音频数据大小字段，来决定我要循环读取多少数据来播放

如果这里面没有低，可以从头偏移 80 个字节，然后读取 2048 个数据，看看前面的数据是不是音频数据

```
data=(ChunkDATA *) (TempBuf+72); //从riff首字节偏移72字节就是data->ChunkID
printf("data ChunkID = %x\r\n", data->ChunkID);
printf("sound num = %x\r\n", data->ChunkSize); //data里面的ChunkSize是后面的音频数据大小
```

```
f_lseek(&fp, 80); //从头开始偏移80字节，后面就是音频数据字节了
f_read(&fp, WaveFileBuf, 2048, &br); //每次循环读取2048字节音频数据进行播放
printf("read WAVE data = %d\r\n", br);
for(i=0;i<40;i++)
{
    printf("data = %x ", WaveFileBuf[i]);
}
```

```
data = 0 data = 0 data = 0 data = ac data = 1 data = ac data = 1 data = 50 data = 3 data = 50 data = 3 data = e6 data = 6 data = 67 data = 6 data = cb data = 7 data = ca data = 7 data = d data = 9 data = d data = 9 data = 28 data = a data =
```

Offset	0 1 2 3 4 5 6 7 8 9 A B C D E F	RIFF i— WAVEJUNK
00000000	52 49 46 46 12 CC AF 00 57 41 56 45 4A 55 4E 4B	R I F F i — W A V E J U N K
00000010	1C 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	偏移 80 个数据
00000020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00000030	66 6D 74 20 10 00 00 00 01 00 02 00 80 BB 00 00	
00000040	00 EE 02 00 04 00 10 00 64 61 74 61 00 C8 AF 00	
00000050	00 00 00 00 AC 01 AC 01 50 03 50 03 E6 04 E7 04	i —————— data E
00000060	66 06 67 06 CB 07 CA 07 0D 09 0D 09 28 0A 28 0A	P P æ S
00000070	16 0B 16 0B D4 0B D3 0B 5D 0C 5D 0C B1 0C B0 0C	f g F F (
00000080	CD 0C CD 0C B1 0C B1 0C 5D 0C 5E 0C D4 0B D3 0B	Ö Ö]] ± °
00000090	16 0B 15 0B 28 0A 28 0A 0D 09 0D 09 CB 07 CA 07	í í ± ±] ^ Ö Ö
000000A0	66 06 66 06 E6 04 E6 04 50 03 50 03 AC 01 AB 01	((È È
000000B0	00 00 00 00 55 FE 54 FE B0 FC B0 FC 1A FB 1A FB	f f æ æ P P ↗ «
000000C0	9A F9 9A F9 35 F8 35 F8 F3 F6 F3 F6 D9 F5 D9 F5	UpTp ° ü ° ü û û
nnnnnnnn	FA F4 FA F4 20 F4 20 F4 A3 F3 A3 F3 4F F3 50 F3	Iù Iù 50 50 50 50 50 50 50 50

下面按照这种偏移之后提取数据的方式播放

```
f_lseek(&fp, 80); //从头开始偏移80字节，后面就是音频数据字节了
printf("read WAVE data = %d\r\n", br);

x = 0;
while(x<data->ChunkSize)//用音频数据大小来确定能播放多长时间，时间到退出while循环
{
    i = 0;
    f_read(&fp, WaveFileBuf, 2048, &br); //每次循环读取2048字节音频数据进行播放
    while(i < sizeof(WaveFileBuf))
    {
        HLdata = 0;

        Ldata = WaveFileBuf[i];//音频低字节
        Hdata = WaveFileBuf[i++];//音频高字节
        i++;
        HLdata = (uint16_t) (Hdata<<8|Ldata); //高位在前，低位在后
        //HLdata = (uint16_t) (Ldata<<8|Hdata);
        SPI_I2S_SendData(SPI3, HLdata); //将高8位低位音频数据放入I2S总线

    }
    x=x+2048;
    //printf("x = %d\r\n", x); //可以打印数据看看播放完成后的大小是不是接近wav文件大小
}
```

这就是完整的播放 wav 文件 PCM 数据的方法，有些 WAV 文件的 WAV 头格式可能有所区别。所以最好是将 WAV 文件用 hex 软件打开，查找 wav 头占用的数据大小。

我们只要找到 **52 49 46 46** 为 riff，**64 61 74 61** 为 data，然后 data 后面 4 字节为数据长度然后 4 个 **00 00 00 00** 开始就是音频数据

下面贴出源代码，但是声音播放有问题，后面会修改，这里只是贴出部分测试内容

```
#include "stm32f10x.h"
#include "sysclock.h"
#include "uartprintf.h"
#include "stdio.h"
#include "ff.h"
#include "sdcode.h"
#include "wavfmt.h"
#include "wm8978.h"
#include "i2s.h"

FATFS fsp;//文件系统使用要先有这个句柄，这个句柄很大所以定义在全局变量
//extern SD_CardInfo SDCardInfo;

#define WAVEFILEBUFSIZE          2048
FIL fp;
uint8_t WaveFileBuf[WAVEFILEBUFSIZE];//2048 内存

int main(void)
{
    uint8_t TempBuf[1024];
    int result = 0,j = 0,x = 0;
    uint32_t br=0;
    uint8_t res=0;
    ChunkRIFF *riff;
    ChunkFMT *fmt;
    ChunkFACT *fact;
    ChunkDATA *data;
    uint8_t Hdata = 0;
    uint8_t Ldata = 0;
    uint16_t HLdata = 0; //这里是 uint16_t 前面标注有错误
    FILINFO info;

    RCC_Configuration(); //初始化时钟
    USART_Config(115200); //初始化串口
    I2S3_Init(); //MCU 端 I2S 初始化

    res = WM8978_Init(); //播放前确保 WM8978 音频芯片 I2S 初始化成功
    printf("wm8978 reset = %d\r\n",res);

    delay_ms(3000);

    result = f_mount(&fsp,"0:",1);
    if (result == FR_OK) {
        printf("SD card mount success!!!->%d\r\n",result);
    }
    else {
        printf("SD card mount failed!!!->%d\r\n",result);
    }

    result=f_open(&fp,"0:DBFS.wav", FA_WRITE|FA_READ); //把 wav 文件打开
    printf("f_open = %d\r\n",result);
    if(result == FR_OK)
    {
        printf("file open success\r\n");

        result = f_stat("0:DBFS.wav",&info); //看看 WAV 文件大小和电脑 WAV 文件大小是否一致
        printf("file size = %ld\r\n",info.fsize);

        f_read(&fp,TempBuf,1024,&br); //读取 WAV 头 1024 字节
        riff=(ChunkRIFF *)TempBuf; //将头 1024 字节转成 riff，这样可以查看 riff 信息是否正确

        printf("riff Format = %x\r\n",riff->Format); //wav 文件 0x45564157
        if(riff->Format==0X45564157) //是 WAV 文件
        {
            fmt=(ChunkFMT *)(TempBuf+48); //获取 FMT 块，这里从头开始偏移 48 字节
            printf("fmt chunkID = %x\r\n",fmt->ChunkID);
            printf("fmt ChunkSize = %x\r\n",fmt->ChunkSize);
            printf("fmt AudioFormat = %x\r\n",fmt->AudioFormat);
            printf("fmt NumOfChannels = %x\r\n",fmt->NumOfChannels);
            printf("fmt SampleRate = %x\r\n",fmt->SampleRate);
            printf("fmt ByteRate = %x\r\n",fmt->ByteRate);
            printf("fmt BlockAlign = %x\r\n",fmt->BlockAlign);
            printf("fmt BitsPerSample = %x\r\n",fmt->BitsPerSample);
        }

        data=(ChunkDATA *)(TempBuf+128); //从 riff 首字节偏移 72 字节就是 data->ChunkID
        printf("data ChunkID = %x\r\n",data->ChunkID);
        printf("sound num = %x\r\n",data->ChunkSize); //data 里面的 ChunkSize 是后面的音频数据大小

        f_lseek(&fp, 136); //从头开始偏移 80 字节，后面就是音频数据字节了
    //    f_read(&fp,WaveFileBuf,2048,&br); //每次循环读取 2048 字节音频数据进行播放
    //    printf("read WAVE data = %d\r\n",br);
}
```

```

//           for(i=0;i<40;i++)
//           {
//               printf("data = %x ",WaveFileBuff[i]);
//           }

x = 0;
while(x<data->ChunkSize)//用音频数据大小来确定能播放多长时间，时间到退出 while 循环
{
    i = 0;
    f_read(&fp,WaveFileBuf,2048,&br);//每次循环读取 2048 字节音频数据进行播放
    while(i < sizeof(WaveFileBuf))
    {
        HLdata = 0;

Ldata = WaveFileBuf[i];//音频低字节
Hdata = WaveFileBuf[i+];//音频高字节//这里获取数组的方式有问题，下面讲解问题解决方案
i++;
HLdata = (uint16_t)(Hdata<<8|Ldata); //高位在前，低位在后
//HLdata = (uint16_t)(Ldata<<8|Hdata);//
SPI_I2S_SendData(SPI3, HLdata); //将高 8 位低位音频数据放入 I2S 总线

    }
    x=x+2048;
    //printf("x = %d\r\n",x); //可以打印数据看看播放完成后的大小是不是接近 wav 文件大小
}

}

else
{
    printf("file open failed\r\n");
    result = f_close (&fp);
}

}

f_close(&fp);

printf("exec end ... \r\n");
result = f_read(&fp,&xzzwavhead,sizeof(xzzwavhead),&bw);

while(1)
{
    SPI_I2S_SendData(SPI3, 0x00ff);
}

return 0;
}

```

声音是出来了，但是感觉声音有点没对

我们还是看看正弦波 wav 文件

Offset	0 1 2 3 4 5 6 7 8 9 A B C D E F
00000000	52 49 46 46 12 CC AF 00 57 41 56 45 4A 55 4E 4B RIFF
00000010	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000030	66 6D 74 20 10 00 00 00 01 00 02 00 80 BB 00 00 fmt
00000040	00 EE 02 00 04 00 10 00 64 61 74 61 00 C8 AF 00 i
00000050	00 00 00 00 AC 01 AC 01 50 03 50 03 E6 04 E7 04
00000060	00 05 00 00 B1 0C 01 50 03 50 03 E6 04 E7 04 g E
00000070	66 66 67 06 CB 07 CA 07 0D 09 0D 09 28 OA 28 OA f g E
00000080	00 0B 16 0B D4 0B D3 0B SD 0C 5D 0C B1 0C B0 0C
00000090	CD 0C CD 0C B1 0C B1 0C SD 0C 5E 0C D4 0B D3 0B i i ±
000000A0	16 0B 15 0B 28 OA 28 OA 0D 09 0D 09 CB 07 CA 07 (
000000AO	66 66 66 E6 04 E6 04 50 03 50 03 AC 01 AB 01 f f ±
000000B0	00 00 00 00 55 FE 54 FE B0 FC B0 FC IA FB IA FB U
000000C0	00 F9 9A F9 35 F6 35 F8 F3 FC F3 F6 D9 F5 D9 F5 3145
000000D0	9A F9 9A F9 35 F6 35 F8 F3 FC F3 F6 D9 F5 D9 F5
000000E0	FA F4 FA F4 2C F4 2D F4 A3 F3 A3 F3 4F F3 50 F3 5A 5A

```

f_lseek(&fp, 80); //从头开始偏移80字节，后面就是音频数据字节了

f_read(&fp, WaveFileBuf, 2048, &br); //每次循环读取2048字节音频数据进行播
printf(" read WAVE data = %d\r\n", br);

for(i=0;i<40;i++)
{
    printf("data = %x ",WaveFileBuf[i]); ← 我们打印我 wav 文件的前 40 个音频数据，没有问题
}

x = 0;
while(x<data->ChunkSize)//用音频数据大小来确定能播放多长时间，时间到进
{
    i = 0;
    f_read(&fp, WaveFileBuf, 2048, &br); //每次循环读取2048字节音频数据进

    while(i < sizeof(WaveFileBuf))
    {
        HLdata = 0;
        Ldata = WaveFileBuf[i];//音频低字节
        Hdata = WaveFileBuf[i+];//音频高字节, 必须先++i, 如果你直接i+
        i++;
        printf("%x %x ", Ldata, Hdata); ← 但是我提取数据打印出来没有符合高字节在前，低字节在后的规则，而且数据重叠了
    }

    HLdata = (uint16_t) (Hdata<<8|Ldata); //高位在前，低位在后
    SPI_I2S_SendData(SPI3, HLdata); //将高8位低位音频数据放入I2S总
    delay_us(10);
}

x=x+2048;
//printf("x = %d\r\n", x); //可以打印数据看看播放完成后的大小是不是

```

```

data = 0 data = 0 data = 0 data = ac data = 1 data = ac data = 1 data = 50 data = 3 data = 50 data = 3 data = e6 data = 4 data = e7 data = 4
ata = 6 data = 6 data = cb data = 7 data = ca data = 7 data = d data = 9 data = d data = 9 data = 28 data = a data = 28 data = a data = 16 d
a = 16 data = b data = d4 data = b data = d3 data = b =====
0 0 0 0 ac ac ac 50 50 50 e6 e7 e7 66 66 67 67 cb cb ca ca d d d 28 28 28 16 16 16 16 d4 d4 d3 d3 5d 5d b1 b1 b0 c0 cd cd c0 b1
d 5d 5e d4 d3 d3 16 16 15 15 28 28 28 d d d cb cb ca ca 66 66 66 e6 e6 e6 e6 50 50 50 ac ab ab 0 0 0 0 55 55 54 54 b0 b0 b0 b0 la
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

```

```

f_lseek(&fp, 80); //从头开始偏移80字节, 后面就是音频数据字节了
printf("read WAVE data = %d\r\n", br);

x = 0;
while(x<data->ChunkSize) //用音频数据大小来确定能播放多长时间, 时间到退出while循环
{
    i = 0;
    f_read(&fp, WaveFileBuf, 2048, &br); //每次循环读取2048字节音频数据进行播放
    while(i < sizeof(WaveFileBuf))
    {
        HLdata = 0;

        Ldata = WaveFileBuf[i]; //音频低字节
        Hdata = WaveFileBuf[i++]; //音频高字节
        i++;
        HLdata = (uint16_t)(Hdata<<8|Ldata); //高位在前, 低位在后
        //HLdata = (uint16_t)(Ldata<<8|Hdata);
        SPI_I2S_SendData(SPI3, HLdata); //将高8位低位音频数据放入I2S总线

    }
    x=x+2048;
    //printf("x = %d\r\n", x); //可以打印数据看看播放完成后的大小是不是接近wav文件大小
}

x = 0;
while(x<data->ChunkSize) //用音频数据大小来确定能播放多长时间, 时间到退出while循环
{
    i = 0;
    f_read(&fp, WaveFileBuf, 2048, &br); //每次循环读取2048字节音频数据进行播放
    while(i < sizeof(WaveFileBuf))
    {
        HLdata = 0;
        Ldata = WaveFileBuf[i]; //音频低字节
        Hdata = WaveFileBuf[++i]; //音频高字节, 必须先++i, 如果你直接i++, 就形成了先去i数据, 然后再i++有问题
        i++;
        //printf("%x %x ", Ldata, Hdata);

        HLdata = (uint16_t)(Hdata<<8|Ldata); //高位在前, 低位在后
        SPI_I2S_SendData(SPI3, HLdata); //将高8位低位音频数据放入I2S总线
        delay_us(10);
    }
    x=x+2048;
    //printf("x = %d\r\n", x); //可以打印数据看看播放完成后的大小是不是接近wav文件大小
}

```

你看第1个字节取得低字节

第2个字节并没有加了i后取, 而是i值取了之后再i++

这样就对了

这里的延时就是根据 bps(位速率来的), 比如 $1/48000=0.00002$ 秒(20us), 也就是 20us 放一个数据进入i2s, 这个问题最好用定时器来精确延时, 这里只是测试

在 codec 芯片中别忘了打开 DAC 和耳机音量设置

```

uint8_t WM8978_Init(void)
{
    u8 res;

    IIC_Init(); //初始化IIC接口
    res=WM8978_Write_Reg(0, 0); //软复位WM8978
    if(res) return 1; //发送指令失败, WM8978异常
    //以下为通用设置
    WM8978_Write_Reg(1, 0X1B); //R1, MIC设置为1(MIC使能), BIASEN设置为1(模拟器工作), VM
    WM8978_Write_Reg(2, 0X1B0); //R2, ROUT1, LOUT1输出使能(耳机可以工作), BOOSTENR, BOOSTEN
    WM8978_Write_Reg(3, 0X6C); //R3, LOUT2, ROUT2输出使能(喇叭工作), RMIX, LMIX使能
    WM8978_Write_Reg(6, 0); //R6, MCLK由外部提供
    //WM8978_Write_Reg(43, 1<<4); //R43, INVRROUT2反向, 驱动喇叭
    WM8978_Write_Reg(47, 1<<8); //R47设置, PGABOOSTL, 左通道MIC获得20倍增益
    WM8978_Write_Reg(48, 1<<8); //R48设置, PGABOOSTR, 右通道MIC获得20倍增益
    WM8978_Write_Reg(49, 1<<1); //R49, TSDEN, 开启过热保护
    WM8978_Write_Reg(10, 1<<3); //R10, SOFTMUTE关闭, 128x采样, 最佳SNR
    WM8978_Write_Reg(14, 1<<3); //R14, ADC 128x采样率

    // WM8978_Output_Cfg(1, 0); //开启BYPASS输出
    // WM8978_Input_Cfg(1, 1, 0); //开启输入通道(MIC&LINE IN)
    WM8978_I2S_Cfg(2, 0); //设置I2S接口模式, 数据位数不需要设置, 播放从设备不使用
    WM8978_ADDA_Cfg(1, 0); //打开DAC

    WM8978_HPvol_Set(30, 30); //设置耳机输出音量
    return 0;
}

```

这样你的正弦波音频输出正常

用定时器解决音乐播放过慢问题

如果是分辨率16位，立体声(左右声道，双声道)，44.1K的音频系统，那么一个音阶如下表述

左声道低8位	左声道高8位	右声道低8位	右声道高8位
--------	--------	--------	--------

所以一个音阶是32位

如果音频文件是44.1Khz的采样率

$1/44100 = 0.00002(20\mu s)$ 播放一个音阶

那么我们定时器中断要设置成20us中断一次

```
TIME_interrupt ( ) //中断函数
{
    SPI_I2S_SendData(SPI3, RightHLDdata); //将高16位右声道音频数据放入I2S总线
    while (SPI_I2S_GetFlagStatus(SPI3, SPI_I2S_FLAG_TXE) == RESET); //等待I2S总线音频数据发送完成，才能发生第2次
    SPI_I2S_SendData(SPI3, LeftHLDdata); //将高16位左声道音频数据放入I2S总线
    while (SPI_I2S_GetFlagStatus(SPI3, SPI_I2S_FLAG_TXE) == RESET); //等待第2次I2S音频数据发送完成
}
```

这就是20us中断一次，在中断里面发送1次左右声道数据，也就是1个音阶

然后中断退出，解析新的wav数据又等20us在这里发

还有wav文件的kbps = 采样率 x 位宽 x 通道数

如44.1k 双通道 16位 ,kbps=44100 x 2 x 16=1411200(1411kbps)

```
void TIM3_init(void)
{
    TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;
    NVIC_InitTypeDef NVIC_InitStructure;

    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3, ENABLE); //开启TIM3时钟

    TIM_DeInit(TIM3);
    TIM_InternalClockConfig(TIM3);

    TIM_TimeBaseStructure.TIM_Period = 20; //20us计数满产生一次中断
    TIM_TimeBaseStructure.TIM_Prescaler = 71; //计数器1us加一次1, 72M/72=1M
    TIM_TimeBaseStructure.TIM_ClockDivision = 0x0;
    TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;

    TIM_TimeBaseInit(TIM3, &TIM_TimeBaseStructure);

    NVIC_InitStructure.NVIC_IRQChannel = TIM3_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 3;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);

    TIM_Cmd(TIM3, ENABLE);

    TIM_PrescalerConfig(TIM3, 71, TIM_PSCReloadMode_Immediate);
    TIM_ClearFlag(TIM3, TIM_FLAG_Update);
    TIM_ITConfig(TIM3, TIM_IT_Update, ENABLE);
}
```

定时器设置成 20us 中断一次

```

void TIM3_IRQHandler(void)
{
    if (TIM_GetITStatus(TIM3, TIM_IT_Update) != RESET) // 检查TIM3中断是否发生
    {
        TIM_ClearITPendingBit(TIM3, TIM_IT_Update); // 清楚TIM3中断标志位
        if (GlobaleInc == 1) // 定时器中断判断变量
        {
            SPI_I2S_SendData(SPI3, RightHldata); // 将高16位右声道音频数据放入I2S总线
            while (SPI_I2S_GetFlagStatus(SPI3, SPI_I2S_FLAG_TXE) == RESET); // 等待I2S总线音频数据发送完成
            SPI_I2S_SendData(SPI3, LeftHldata); // 将高16位左声道音频数据放入I2S总线
            while (SPI_I2S_GetFlagStatus(SPI3, SPI_I2S_FLAG_TXE) == RESET); // 等待第2次I2S音频数据发送完成

            GlobaleInc = 0;
            Globalefunc = 1;
        }
    }

    f_lseek(&fp, 144); // 从头开始偏移136字节，后面就是音频数据字节了
    TIM3_init(); // 初始化TIM3
    x = 0;
    while (x < data->ChunkSize) // 用音频数据大小来确定能播放多长时间，时间到退出while
    {
        i = 0;
        f_read(&fp, WaveFileBuf, 2048, &br); // 每次循环读取2048字节音频数据进行播放

        while (i < sizeof(WaveFileBuf))
        {
            if (Globalefunc == 1)
            {
                Globalefunc = 0;

                Ldata = WaveFileBuf[i]; // 音频低字节
                Hdata = WaveFileBuf[++i]; // 音频高字节，必须先++i，如果你直接i++，就形成了先去i数据，然后再i+
                LeftHldata = (uint16_t)(Hdata << 8 | Ldata); // 高位在前，低位在后

                i++;
                Ldata = WaveFileBuf[i]; // 音频低字节
                Hdata = WaveFileBuf[++i]; // 音频高字节，必须先++i，如果你直接i++，就形成了先去i数据，然后再i+
                RightHldata = (uint16_t)(Hdata << 8 | Ldata); // 高位在前，低位在后

                i++;
            }
            GlobaleInc = 1; // 解析完毕置位定时器中断变量
        }
        x = x + 2048;
    }
    // printf("%d\r\n", x); // 可以打印数据看看播放完成后的大小是不是接近wav文件大小
}

```

这就是定时器精确中断的播放过程，声音听起来是同步了，但是还是有点点问题，有可能是没有 DMA 进行双缓冲造成的，下面用 DMA 双缓冲尝试一下。

DMA 双缓冲实现音频播放

DMA 双缓冲需要用到上面章节的 I2S 调试成功，定时器播放调试成功的程序。