

STM32

一、GPIO输出

19	PB1	I/O		PB1	ADC12_IN9/TIM3_CH4	TIM1_CH3N
20	PB2	I/O	FT	PB2/BOOT1		
21	PB10	I/O	FT	PB10	I2C2_SCL/USART3_TX	TIM2_CH3
22	PB11	I/O	FT	PB11	I2C2_SDA/USART3_RX	TIM2_CH4
23	VSS_1	S		VSS_1		
24	VDD_1	S		VDD_1		

- FT代表了可以容忍5v电压

在STTM32中,所有的GPIO口都是挂载在APB2外设总线上的

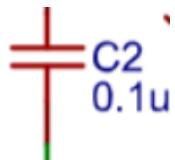
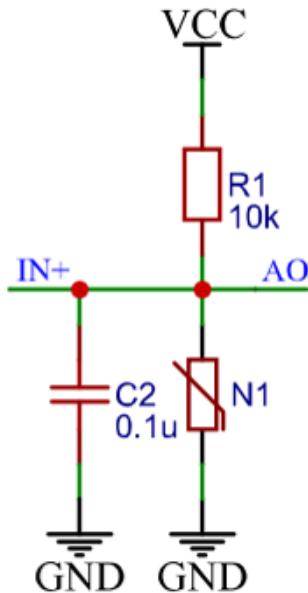
vdd是3.3v

vss是接地

- (7) 复用功能的推挽输出_AF_PP ——片内外设功能 (I2C的SCL,SDA)
- (8) 复用功能的开漏输出_AF_OD——片内外设功能 (TX1,MOSI,MISO,SCK,SS)

- 初始化所有引脚
 - | GPIO_InitStructure.GPIO_Pin = GPIO_Pin_All;
 - GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0|GPIO_Pin_1

1.



这种一端在电路中另一端接地的电容一般是滤波电容,保证电路稳定

- **输出电压**: 它是从某一点到地的电压。这是我们希望得到的分压结果。
- `typedef unsigned char uint8_t;`
- | `typedef unsigned short int uint16_t;`

二、按键控制led

1 初始化GPIO口

当按键没有按下去的时候读取对应GPIO口电平

```
GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_1) == 0 //代表读取到了低电平
```

2 初始化按键

1 按键没有按下去的时候应该是高电平(配置为上拉输入模式)

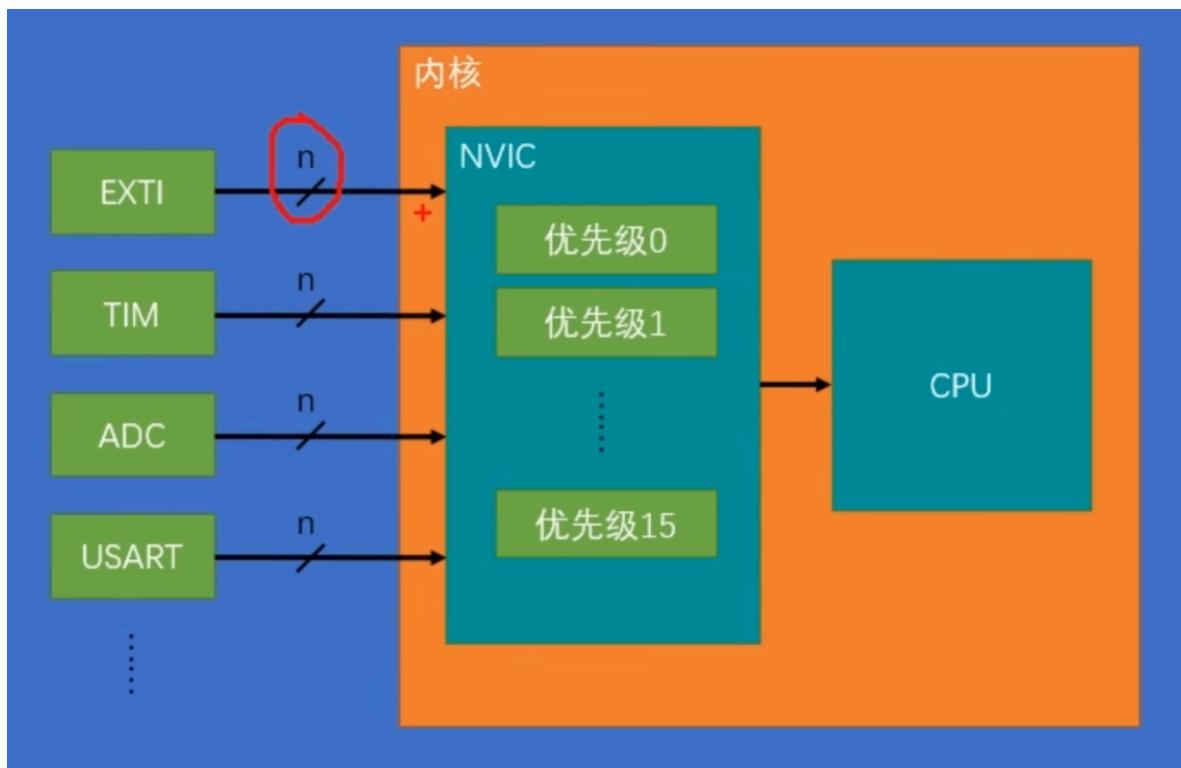
2 获取按键值

用GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_1) == 0 代表按下了按键
进行消抖

等待GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_1) == 1 的时候
进行消抖

也就是当按键值为1的时候代表了按下去,即可以对对应GPIO口的电平进行反转

三、中断



n:代表了一个外设可能会同时占用多个中断通道

NVIC优先级分组

- NVIC的中断优先级由优先级寄存器的4位（0~15）决定，这4位可以进行切分，分为高n位的抢占优先级和低4-n位的响应优先级
- 抢占优先级高的可以中断嵌套，响应优先级高的可以优先排队，抢占优先级和响应优先级均相同的按中断号排队

分组方式	抢占优先级	响应优先级
分组0	0位，取值为0	4位，取值为0~15
分组1	1位，取值为0~1	3位，取值为0~7
分组2	2位，取值为0~3	2位，取值为0~3
分组3	3位，取值为0~7	1位，取值为0~1
分组4	4位，取值为0~15	0位，取值为0

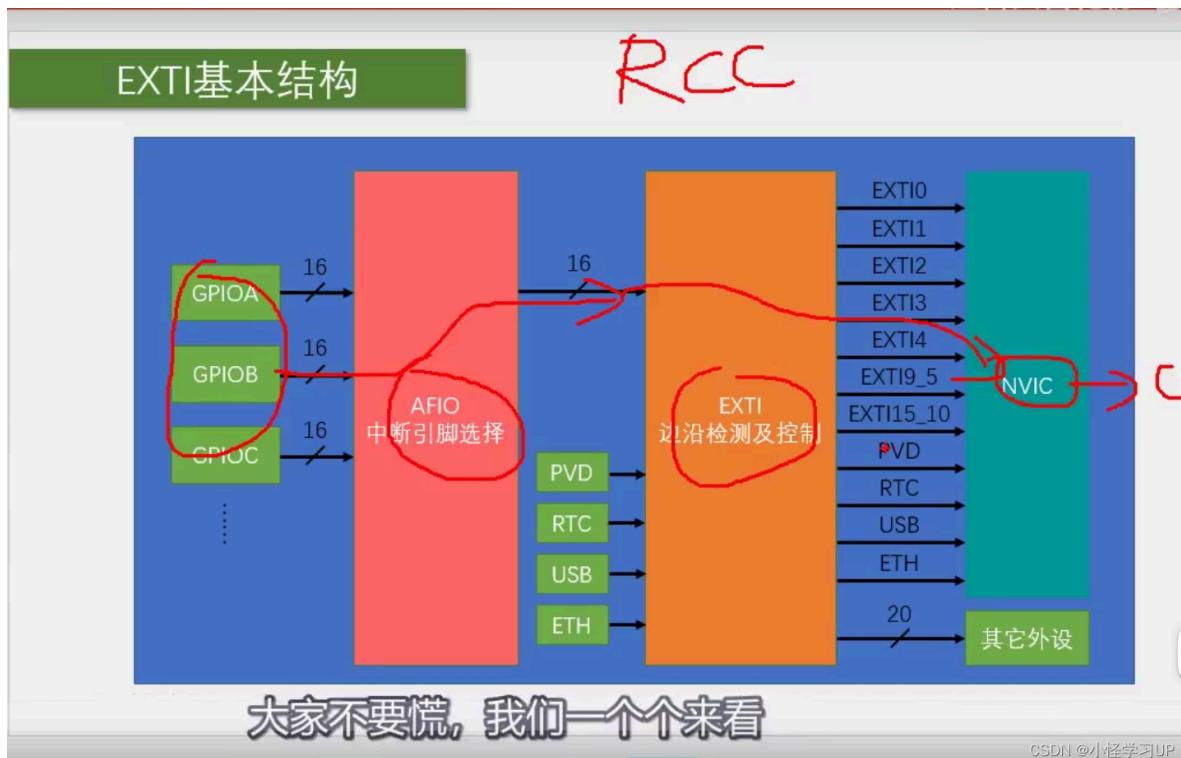
- 优先级的数是值越小优先级越高

中断/事件线	输入源
EXTI0	P <u>X</u> 0 (X 可为 A, B, C, D, E, F, G, H, I)
EXTI1	P <u>X</u> 1 (X 可为 A, B, C, D, E, F, G, H, I)
EXTI2	P <u>X</u> 2 (X 可为 A, B, C, D, E, F, G, H, I)
EXTI3	P <u>X</u> 3 (X 可为 A, B, C, D, E, F, G, H, I)
EXTI4	P <u>X</u> 4 (X 可为 A, B, C, D, E, F, G, H, I)
EXTI5	P <u>X</u> 5 (X 可为 A, B, C, D, E, F, G, H, I)
EXTI6	P <u>X</u> 6 (X 可为 A, B, C, D, E, F, G, H, I)
EXTI7	P <u>X</u> 7 (X 可为 A, B, C, D, E, F, G, H, I)
EXTI8	P <u>X</u> 8 (X 可为 A, B, C, D, E, F, G, H, I)
EXTI9	P <u>X</u> 9 (X 可为 A, B, C, D, E, F, G, H, I)
EXTI10	P <u>X</u> 10 (X 可为 A, B, C, D, E, F, G, H, I)
EXTI11	P <u>X</u> 11 (X 可为 A, B, C, D, E, F, G, H, I)
EXTI12	P <u>X</u> 12 (X 可为 A, B, C, D, E, F, G, H, I)
EXTI13	P <u>X</u> 13 (X 可为 A, B, C, D, E, F, G, H, I)
EXTI14	P <u>X</u> 14 (X 可为 A, B, C, D, E, F, G, H, I)
EXTI15	P <u>X</u> 15 (X 可为 A, B, C, D, E, F, G, H, I)
EXTI16	PVD 输出
EXTI17	RTC 闹钟事件
EXTI18	USB 唤醒事件
EXTI19	以太网唤醒事件 (只适用互联型)

EXTI简介

- EXTI (Extern Interrupt) 外部中断
- EXTI可以监测指定GPIO口的电平信号，当其指定的GPIO口产生电平变化时，EXTI将立即向NVIC发出中断申请，经过NVIC裁决后即可中断CPU主程序，使CPU执行EXTI对应的中断程序
- 支持的触发方式：上升沿/下降沿/双边沿/软件触发
- 支持的GPIO口：所有GPIO口，但相同的Pin不能同时触发中断
- 通道数：16个GPIO_Pin，外加PVD输出、RTC闹钟、USB唤醒、以太网唤醒
- 触发响应方式：中断响应/事件响应

- 中断响应就是产生中断，事件响应就是产生事件，可以触发DMA触发之类的，或者唤醒处理器



设置EXTI中断

```

1 //配置对应GPIO口,这里是GPIO_Pin_14
2 //AFIO选择中断引脚
3 GPIO_EXTILineConfig()
4 //EXTI初始化
5 //NVIC中断分组
6 NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);
7 //NVIC初始化
8

```

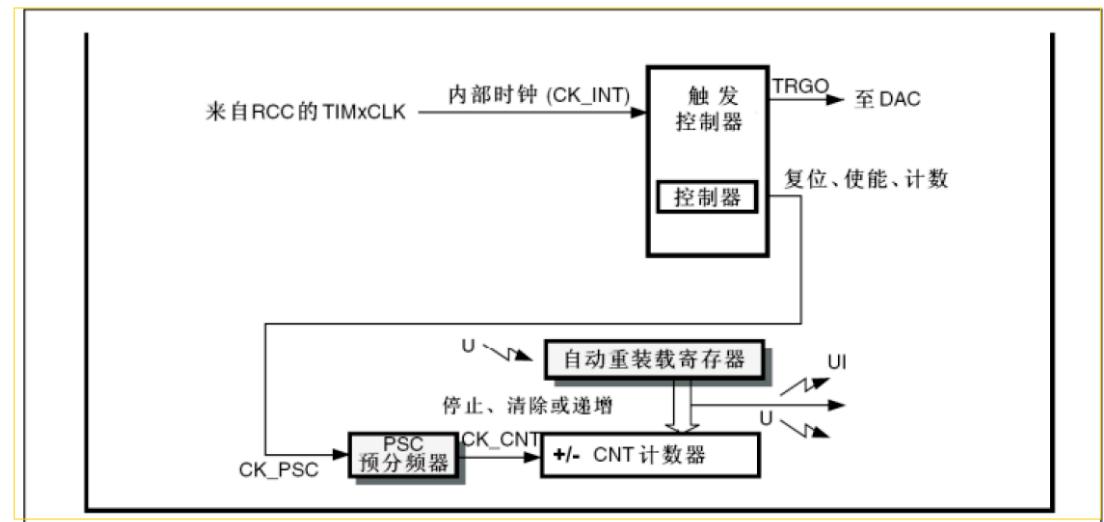
TIM定时器

中断函数:startup_stm32f10x_md.s

- 16位计数器，预分频器，自动重装器的时基单元.最大59.65s的定时
72Mhz/65536/65536
- 定时器级联：一个定时器最大的定时器是59.65s,级联一个定时器。也就是输入周期59.65 再乘以两次65536.因为主定时器的预分频器和自动重装值都可以计入
- 计数时钟每来一个上升沿，计数器的值就加1。也就是说每个周期加
1.72MHZ的时钟周期是1/72mhz

基本定时器

图144 基本定时器框图



- ui是中断
- u是更新事件，可以触发内部其他电路的工作



我们在使用DAC的时候，可能会用DAC输出一段波形，就需要每隔一段时间来触发一次DAC,让他输出下一个电压点

主模式：把定时器的更新事件映射到TRGO上面，然后TRGO直接接到DAC的触发

TRGO:

可以使用函数选择触发源 TIM_SelectOutputTrigger(TIM2, TIM_TRGOSource_Update);

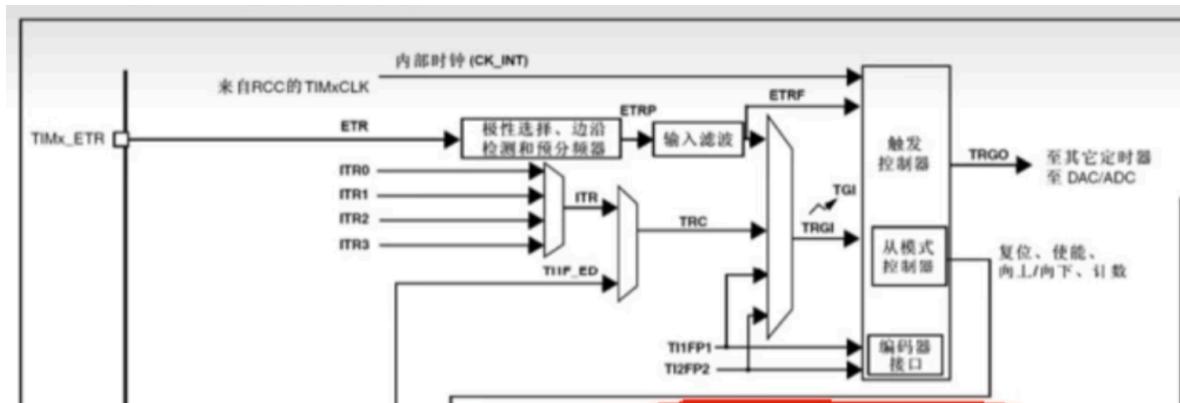
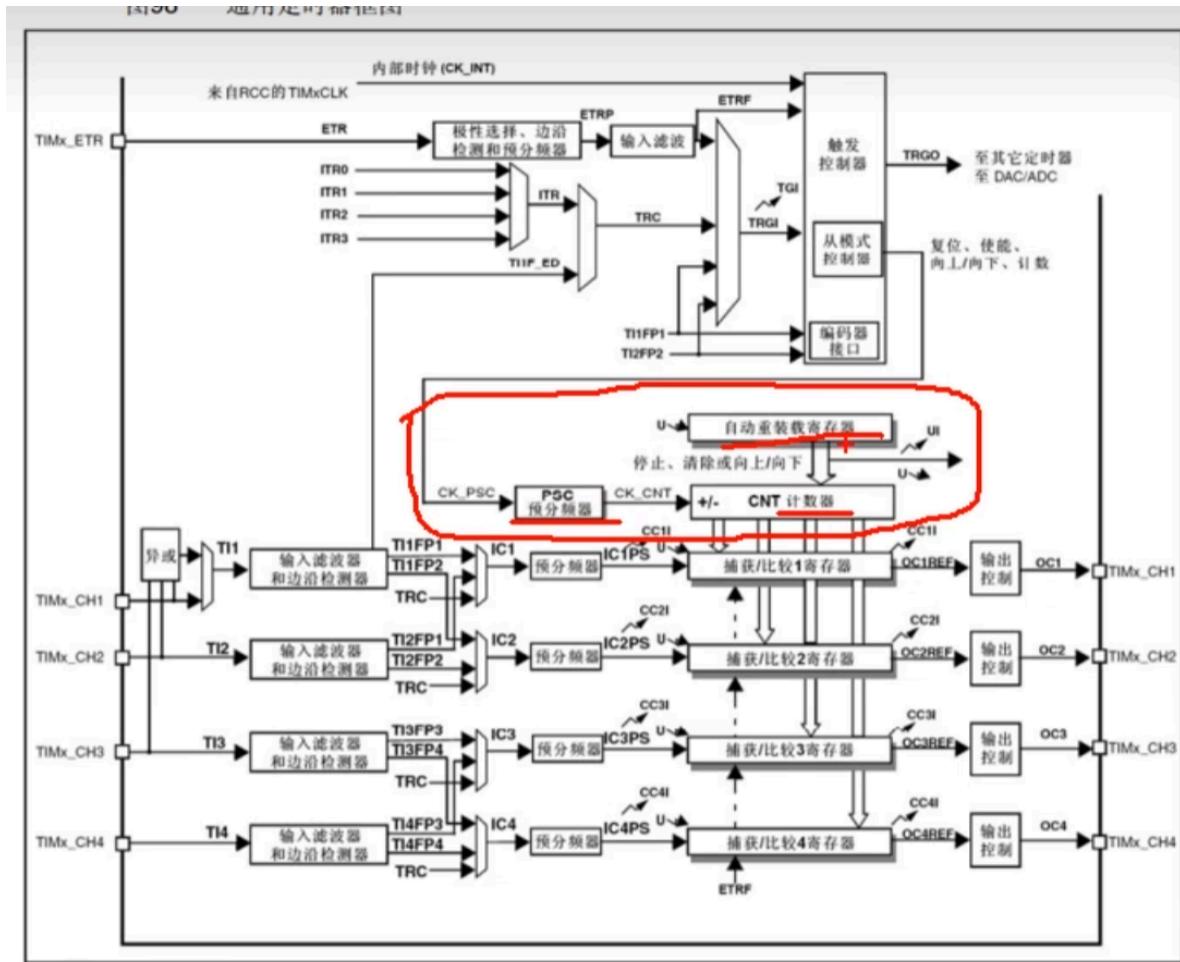
然后在DAC的输入源里面选择定时器的TRGO信号即可

```
1 void DAC_Config(void) {  
2     DAC_InitTypeDef DAC_InitStructure;  
3  
4     // 使能DAC时钟  
5     RCC_APB1PeriphClockCmd(RCC_APB1Periph_DAC, ENABLE);  
6  
7     // 初始化DAC  
8     DAC_InitStructure.DAC_Trigger = DAC_Trigger_T2_TRGO;  
9     DAC_Init(DAC_Channel_1, &DAC_InitStructure);  
10  
11    // 使能DAC  
12    DAC_Cmd(DAC_Channel_1, ENABLE);  
13 }
```

更新中断:

更新事件:

通用定时器:



外部时钟

外部时钟模式2

使用TIMx_ETR引脚进行配置，引脚定义表里面有说明

10	PA0-WKUP	I/O	PA0	WKUP/USART2_CTS/ADC12 IN/TIM2_CH1 ETR
----	----------	-----	-----	---------------------------------------

在PA0引脚配置一下方波时钟，滤波之后的信号RTRF进入触发器控制器。就可以做时基单元的时钟了。使用上拉输入

表78 TIMx内部触发连接⁽¹⁾

从定时器	ITR0 (TS = 000)	ITR1 (TS = 001)	ITR2 (TS = 010)	ITR3 (TS = 011)
TIM2	TIM1	TIM8	TIM3	TIM4
TIM3	TIM1	TIM2	TIM5	TIM4
TIM4	TIM1	TIM2	TIM3	TIM8
TIM5	TIM2	TIM3	TIM4	TIM8

1. 如果某个产品中没有相应的定时器，则对应的触发信号ITRx也不存在。

```
TIM_ETRClockMode2Config(TIM2,TIM_ExtTRGPSC_OFF,TIM_ExtTRGPolarit  
y_NonInverted,0x0F);
```

外部时钟模式1

把这个TRGI当做外部时钟来使用的时候，这一路叫做“外部时钟1”.信号来源：

1.ETR引脚(一般情况通过ETR引脚即可)

1.配置定时器为外部时钟模式1

```
TIM_ETRClockMode1Config(TIM2,TIM_ExtTRGPSC_OFF,TIM_ExtTRGPolarit  
y_NonInverted,0x0F);
```

2.ETR引脚在引脚对应表上面配置，配置为输入模式即可

2. 使用ITRX(来自其他定时器的), 主模式的输出TRGO可以通向其他定时器的, 可以接到其他定时器的ITR引脚上面。这样可以实现级联的功能。

配置定时器为从模式, 选择对应的ITR0以及其他

```
/*配置定时器为从模式, 使用ITR输入*/
TIM_SelectInputTrigger(TIM2, TIM_TS_ITR0); // 选择ITR0作为输入触发源
TIM_SelectSlaveMode(TIM2, TIM_SlaveMode_External1); // 配置为从模式, 外部时钟模式1
```

3.TI1F_ED(TIMx_CH1经过输入滤波之后的信号)。通过这路输入的时钟上升沿和下降沿一样有效

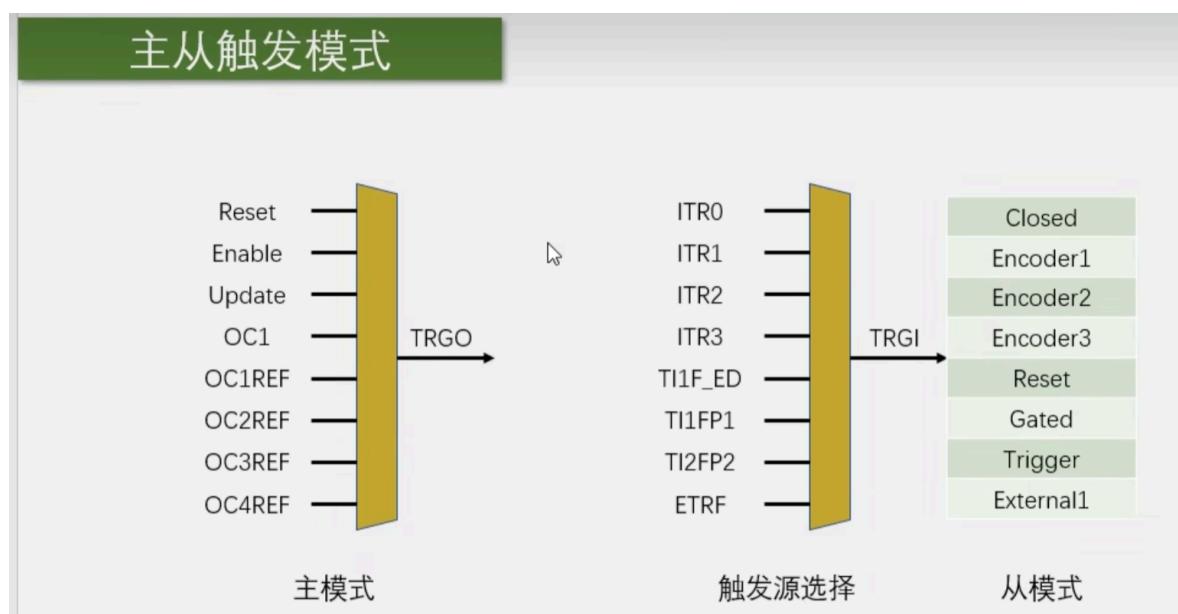
```
TIM_TIxExternalClockConfig(TIMx, TIM_TS_TI1F_ED, TIM_ICPolarity_Bot
thEdge, 0xF);
```

TIM_TS_TI1F_ED 可以由其他外设产生, 通过对称的引脚

4.TI1FP1和TI2FP2。 TI1FP1是CH1引脚的时钟, TI2FP2是CH2引脚的时钟

```
//选择TI1FP1或TI2FP2作为外部时钟源
```

```
TIM_TIxExternalClockConfig(TIMx, TIM_TS_TI1FP1, TIM_ICPolarity_Ris
ing, 0xF);
```



1. 配置为主模式可以直接配置硬件为主模式，产生TRGO信号
2. 配置从模式的时候需要配置触发源选择和触发之后的动作

开启主模式,配置输出源为TIM_TRGO_Update

两种方式：

1.

```
1 //TIM2
2 TIM_MasterConfigTypeDefTIM_MasterConfig;
3 TIM_MasterConfig.TIM_MasterOutputTrigger=TIM_TRGO_Update; //TR
4 GO输出更新事件
5 TIM_MasterConfig.TIM_MasterSlaveMode=TIM_MasterSlaveMode_Enable;
6 TIM_ConfigMaster(TIM2,&TIM_MasterConfig);
```

2.

```
1 TIM_SelectOutputTrigger(TIM3,TIM_TRGOSource_Update); // 设置
2 TRGO为更新事件
```

DAC配置触发源为TIM2

```
1 voidDAC_Config(void)
2 {DAC_InitTypeDefDAC_InitStructure;
3 //1.启用DAC时钟
4 RCC_APB2PeriphClockCmd(RCC_APB2Periph_DAC,ENABLE);
5 //2.配置DAC
6 DAC_InitStructure.DAC_Trigger=DAC_Trigger_T2_TRGO; // 触发源为
7 TIM2的TRGO
8 DAC_InitStructure.DAC_WaveGeneration=DAC_WaveGeneration_None
```

```
 ; //不使用波形生成  
8 DAC_InitStructure.DAC_OutputBuffer=DAC_OutputBuffer_Enable; //  
 /启用输出缓冲  
9 DAC_Init(DAC_Channel_1,&DAC_InitStructure);  
10 //3.启用DAC通道  
11 DAC_Cmd(DAC_Channel_1,ENABLE);  
12 //4.启用DAC软件触发 (如果需要)  
13 DAC_SoftwareTriggerCmd(DAC_Channel_1,ENABLE);}
```

从模式配置其他定时器做输入源

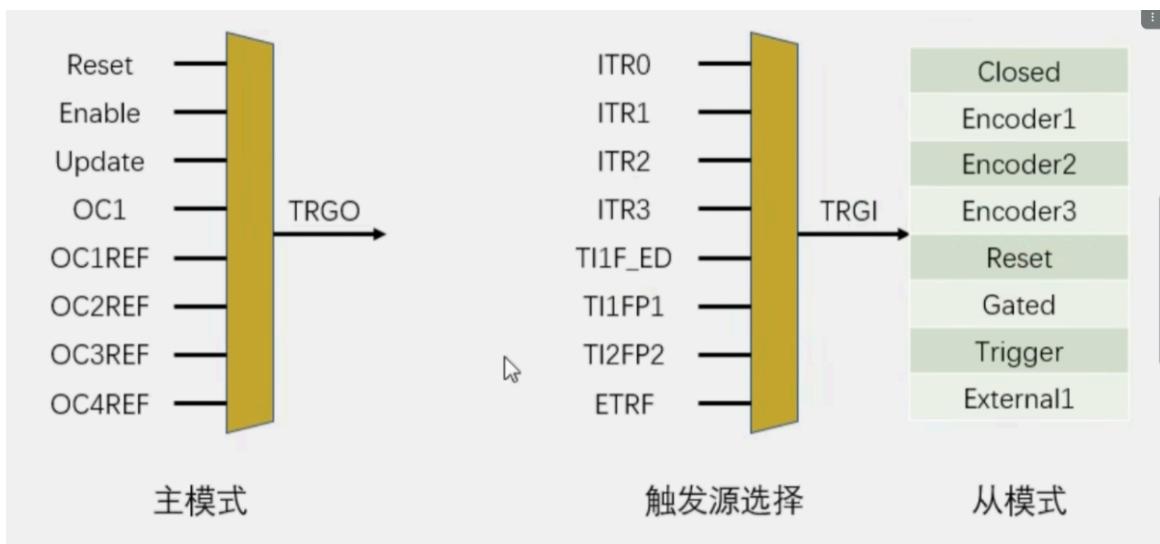
```
1 //配置TIM3为从模式, 接收TIM2的TRGO信号  
2 TIM_SlaveConfig.TIM_SlaveMode=TIM_SlaveMode_External1; //使用外  
部触发模式  
3 TIM_SlaveConfig.TIM_InputTrigger=TIM_TS_ITR1; //输入触发源为ITR1  
 (TIM2的TRGO) 这个是预先定义的  
4 TIM_SlaveConfig.TIM_TriggerPolarity=TIM_TriggerPolarity_NonIn  
 vverted; //触发极性为非反向  
5 TIM_SlaveConfig.TIM_PSCReloadMode=TIM_PSCReloadMode_Immediate  
 ; //立即更新预分频器TIM_SlaveConfig  
6 (TIM3,&TIM_SlaveConfig); //启动TIM3TIM_Cmd(TIM3,ENABLE);
```

外部事件模式2配置

TIM_ETRClockMode2Config(TIM2,
TIM_ExtTRGPSC_OFF,
TIM_ExtTRGPolarity_NonInverted, 0x0F);

其余的不用配置了

主从触发模式



主 模 式 输 出 的 触 发 源 函 数 :

```
void TIM_SelectOutputTrigger(TIM_TypeDef* TIMx, uint16_t TIM_TRGOSource);
```

触 发 源 选 择 :

```
void TIM_SelectInputTrigger(TIM_TypeDef* TIMx, uint16_t TIM_InputTriggerSource);
```

从模式：通道3和通道4只能开启捕获中断，在中断里手动清0

```
void TIM_SelectSlaveMode(TIM_TypeDef* TIMx, uint16_t TIM_SlaveMode);
```

slave mode:对应的是



这四个模式，上面的四个模式选择

有其他模式。可以选择从模式自动触发一些操作，而不是一定需要主模式的TRGO

五、pwm

1.oc(输出比较)

输出比较可以通过比较CNT与CCR寄存器值的关系，来对输出电平进行置1、置0或翻转的操作，用于输出一定频率和占空比的PWM波形

每个高级定时器和通用定时器都拥有4个输出比较通道

高级定时器的前3个通道额外拥有死区生成和互补输出的功能

TIMx_CHX:

先正常配置定时器,只需要额外配置输入捕获选择你需要的端口即可,例如TIMx_CH1对应的是PA0,只需要额外配置一下PA0即可

```
1 TIM_OCInitTypeDef TIM_OCInitStructure; // 定义结构体变量
2 TIM_OCStructInit(&TIM_OCInitStructure); // 结构体初始化, 若结构体没
   有完整赋值
3 // 则最好执行此函数, 给结构体所有成员都赋一个默认值
4 // 避免结构体初值不确定的问题
5 TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1; // 输出比较模式,
   选择PWM模式1
6 TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_High; // 输出
   极性, 选择为高, 若选择极性为低, 则输出高低电平取反
7 TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable; //
```

```

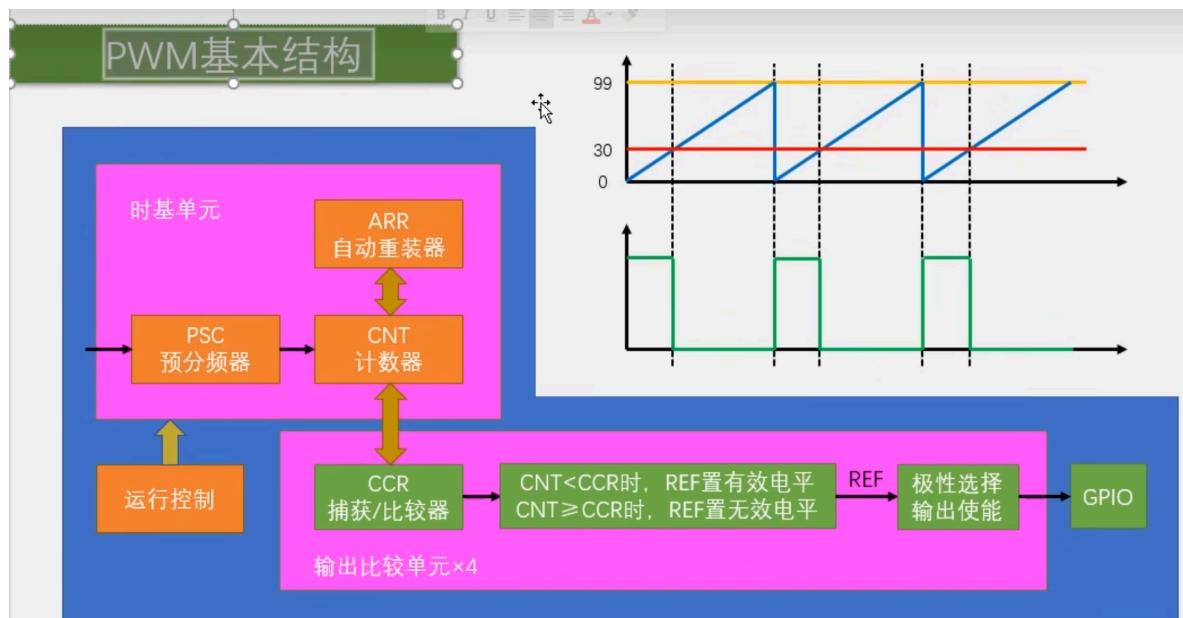
8   输出使能
9   TIM_OCInitStructure.TIM_Pulse=0; //初始的CCR值
9   TIM_OC1Init(TIM2,&TIM_OCInitStructure); // 将结构体变量交给
      TIM_OC1Init，配置TIM2的输出比较通道1

```

输出模式控制器执行逻辑：

模式	描述
冻结	CNT=CCR时, REF保持为原状态 REF是参考电平的意思
匹配时置有效电平	CNT=CCR时, REF置有效电平
匹配时置无效电平	CNT=CCR时, REF置无效电平 } 定时输出一次性信号
匹配时电平翻转	CNT=CCR时, REF电平翻转
强制为无效电平	CNT与CCR无关, REF强制为无效电平
强制为有效电平	CNT与CCR无关, REF强制为有效电平
PWM模式1	向上计数 : CNT<CCR时, REF置有效电平, CNT≥CCR时, REF置无效电平 向下计数 : CNT>CCR时, REF置无效电平, CNT≤CCR时, REF置有效电平
PWM模式2	向上计数 : CNT<CCR时, REF置无效电平, CNT≥CCR时, REF置有效电平 向下计数 : CNT>CCR时, REF置有效电平, CNT≤CCR时, REF置无效电平

2.pwm基本结构



- PWM频率 : $F_{req} = CK_{PSC} / (PSC + 1) / (ARR + 1)$
- PWM占空比 : $Duty = CCR / (ARR + 1)$
- PWM分辨率 : $Reso = 1 / (ARR + 1)$

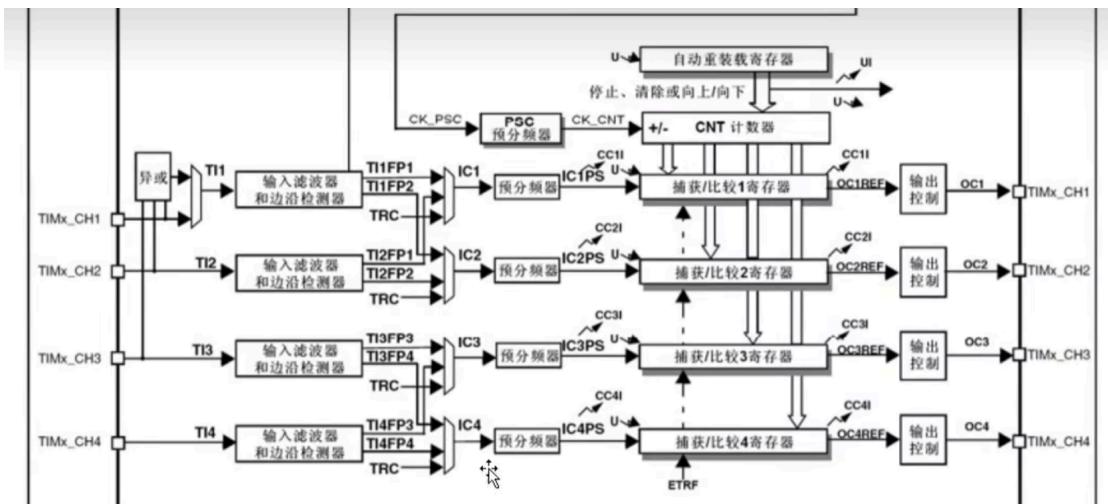
- 分辨率:占空比变化步距,占空比可以被调整的最小单位或步长
- pwm进行灯光呼吸:只需要改变高电平所占时间,从高到底从低到高

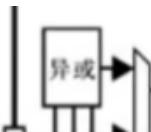
3.ic输入捕获

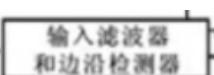
- IC (Input Capture) 输入捕获
- 输入捕获模式下, 当通道输入引脚出现指定电平跳变时, 当前CNT的值将被锁存到CCR中, 可用于测量PWM波形的频率、占空比、脉冲间隔、电平持续时间等参数
- 每个高级定时器和通用定时器都拥有4个输入捕获通道
- 可配置为PWMI模式, 同时测量频率和占空比
- 可配合主从触发模式, 实现硬件全自动测量

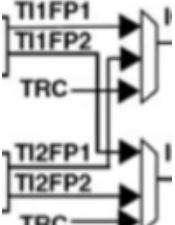
- 对于同一个定时器只能同一时间使用输入捕获或者输出比较
- 接收到输入信号, 执行CNT锁存到CCR的动作
- 测周法 : 两个上升沿内, 以标准频率 f_c 计次, 得到N, 则频率

$$f_x = f_c / N$$



- 1.  三输入的异或门，这个异或门的输入接在了通道1, 2, 3端口。当三个输入引脚的任何一个有电平翻转的时候，输出引脚就产生一次电平翻转。之后输出通过数据选择器到达输入捕获通道1。数据选择器如果选择上面一个，输入捕获通道1的输入就是三个引脚的异或值。如果选择下面一个，异或门就没有用。

- 2. 

- 3.  引脚Timx_ch1的输入可以同时映射到两个捕获单元。可以用于pwmi模式同时测量频率和占空比

• 输入捕获流程

- 1.确认输入通道
- 2.输入滤波器和边沿检测器
- 3.捕获通道：选择单通道还是双通道
- 4.预分频器
- 5.捕获寄存器

六、ADC

ADC配置

- 1.选择独立模式还是双ADC
- 2.数据对齐，右对齐还是

3.是否外部触发

4.配置转换模式，连续模式和扫描模式

5.通道数

6.模拟输入_AIN——应用ADC模拟输入，或者低功耗下省电

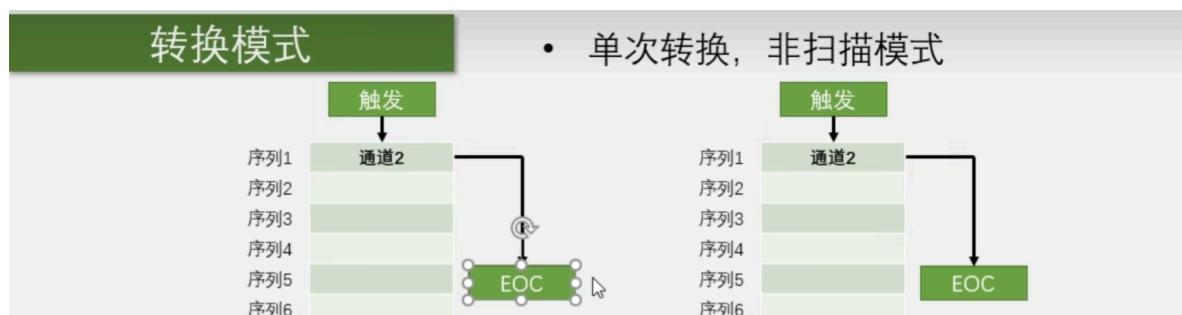
- ADC的几种工作模式是什么？数据对齐是什么？

一般使用右对齐，左对齐可以干掉精度，只取出来高八位的数据。这样12位ADC就变成了8位ADC。精度要求不高的时候可以选择使用ADC进行信号采样的步骤是什么？

单片机中ADC的输入输出是什么？怎么转换的

- 当外部电压不是标准的3.3V的时候，可以读取内部基准电压进行校准，可以得到正确电压
- 转换模式：

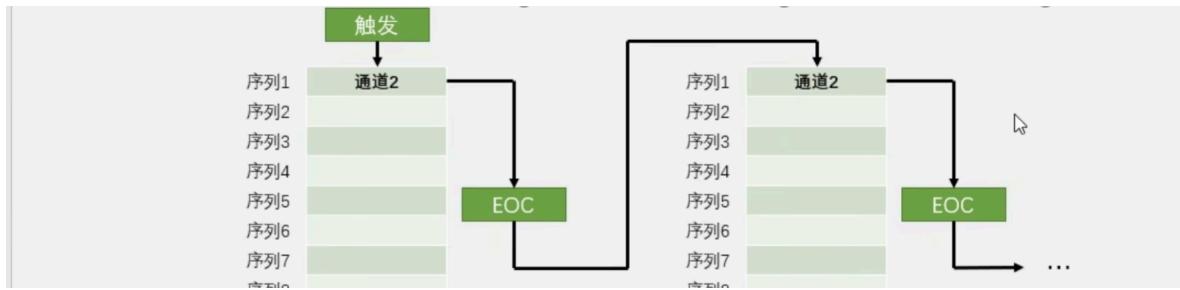
1.单次转换,非扫描模式：



判断eoc标志位置,当转换完成,可以在数据寄存器读取结果

如果要再启动一次转换,就需要再触发一次

2.连续转换非扫描模式



3. 单次转换, 扫描模式

4. 连续转换, 扫描模式

转换时间:

- AD转换的步骤 : 采样, 保持, 量化, 编码
- STM32 ADC的总转换时间为 :
 $T_{CONV} = \text{采样时间} + 12.5\text{个ADC周期}$
- 例如 : 当ADCCLK=14MHz, 采样时间为1.5个ADC周期
 $T_{CONV} = 1.5 + 12.5 = 14\text{个ADC周期} = 1\mu\text{s}$

$$T_{CONV} = \text{采样时间} + 12.5\text{个ADC周期} \quad 12.5\text{个ADC周期是因为12位ADC+0.5个周期(固定时间)}$$

量化编码就是逐次比较的过程

采样保持: AD转换, 也就是后面量化编码需要时间的, 采样保持是为了电压稳定。

在量化编码之前打开采样开关, 收集一下外部电压, 存储一下这个电压。断开采样开关, 再进行后面的AD转换, 这样量化编码期间, 电压始终不变

规则数据寄存器ADC_DR可以选择左对齐还是右对齐

数据转换结束后, 可以产生中断

溢出中断

如果发生DMA传输数据丢失，会置位**ADC**状态寄存器ADC_SR的OVR位，如果同时使能了溢出中断，那在转换结束后会产生一个溢出中断。

ADC精度和分辨率

1. 分辨率 (Resolution)

- 分辨率是指ADC能够将模拟信号量化为多少个离散的数字值。
- 在STM32F4中，ADC的分辨率是**12位**，这意味着它可以将模拟信号量化为4096个不同的数字值 ($2^{12} = 4096$)。
- 分辨率越高，ADC能够检测到的信号变化越细微。

2. 精度 (Accuracy)

- 精度是指ADC测量结果与实际模拟信号之间的误差。
- 精度受多种因素影响，包括：
 - 噪声：来自电源、PCB布局、外部环境等。
 - 参考电压：参考电压的稳定性直接影响ADC的精度。
 - 温度：温度变化会导致ADC内部电路的参数漂移。
 - 校准：ADC的校准参数是否准确。
- 在STM32F4中，ADC的典型精度为 ± 2 LSB（最低有效位），但在最佳条件下可以达到 ± 1 LSB。

1. 如何处理ADC采样中的噪声问题？硬件滤波：在ADC输入端添加RC低通滤波电路软件滤波：

2.

七、串口

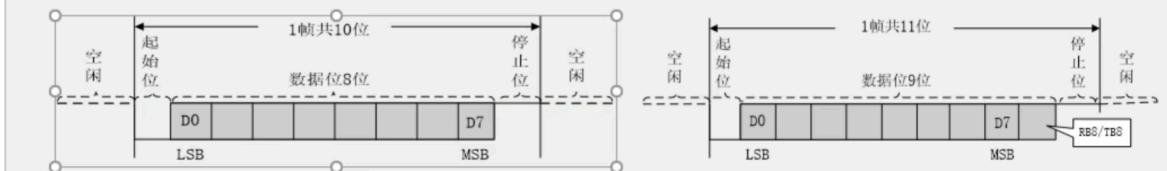
名称	引脚	双工	时钟	电平	设备
USART	TX、RX	全双工	异步	单端	点对点
I2C	SCL、SDA	半双工	同步	单端	多设备
SPI	SCLK、MOSI、MISO、CS	全双工	同步	单端	多设备
CAN	CAN_H、CAN_L	半双工	异步	差分	多设备
USB	DP、DM	半双工	异步	差分	点对点

- TTL电平：+3.3V或+5V表示1，0V表示0
- RS232电平：-3~-15V表示1，+3~+15V表示0
- RS485电平：两线压差+2~+6V表示1，-2~-6V表示0（差分信号）

- 差分信号抗干扰能力很强

串口参数及时序

- 波特率：串口通信的速率
- 起始位：标志一个数据帧的开始，固定为低电平
- 数据位：数据帧的有效载荷，1为高电平，0为低电平，低位先行
- 校验位：用于数据验证，根据数据位计算得来
- 停止位：用于数据帧间隔，固定为高电平



数据位：5678都可以

停止位：1位，1.5位，2位

校验位：无校验，偶校验，奇校验

波特率：实现数据同步

引脚需要配置成推挽输出模式

波特率计算：

$$\text{USARTDIV} = \frac{f_{\text{PCLK}}}{16 \times \text{波特率}}$$

USARTDIV 是波特率

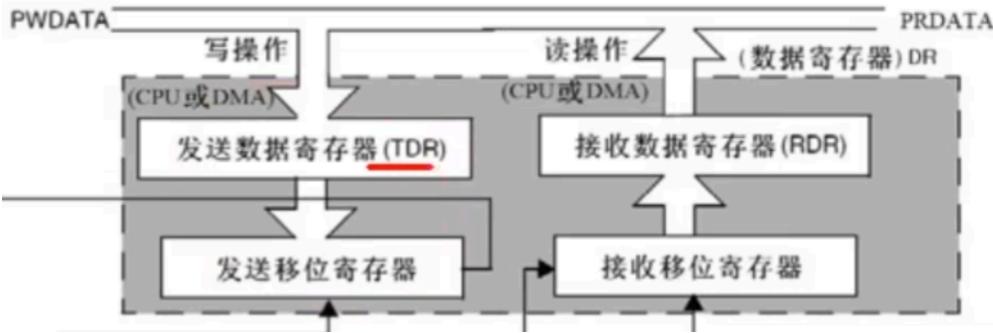
寄存器 (USART_BRR) 的值，fPCLK 是串口的时钟频率

9600 19200 38400 115200

9600代表1s之内串口可以传输9600个高低电平

- 低位先行：0xF 00001111 从1开始发送

- 低电平开始,高电平结束
- TDR和RDR占用一个地址,在程序上只表现为一个寄存器,就叫DR寄存器,实际硬件中一个用于发送TDR,一个用于接收RDR
- 发送移位寄存器的作用就是把一个字节的数据一位一位的移出去



- 写入给TDR数据之后会检测移位寄存器是不是正在移位,若没有,把数据全部发送给发送移位寄存器,准备发送
- 当数据移动到移位寄存器的时候会置一个标志位,叫**TXE**,也就是发送寄存器为空: TXE置1(此时数据没有发送出去,只是在移位寄存器里面)
- TXE为1的时候就可以发送下一个数据了
- 然后发送移位寄存器就会在下面这里的发生器控制的驱动下向右移位,然后一位一位地,把数据输出到TX引脚

在接收器控制的驱动下

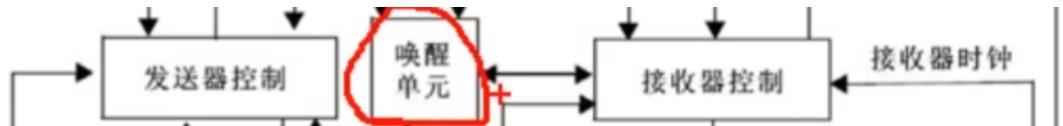
一位一位地读取RX电平,先放在最高位,然后向右移

一位一位地读取RX电平,先放在最高位,然后向右移

从高往低

- 对于接收移位寄存器,会把接收的数据从高到低,移动一个字节才会把数据发送到接收数据寄存器里面,发送过程也会置一个标志位**RXNE** 为1的时候可以从接收数据寄存器读取数据

- usart的sclk只能输出不能输入,两个usart之间不能进行同步的串口通信,可以兼容SPI



- 唤醒单元可以实现串口多设备通信

UART(USART是一个模块,UART是异步模式)

TTL电平和RS232电平

- TTL电平 : +3.3V或+5V表示1, 0V表示0
- RS232电平 : -3~-15V表示1, +3~+15V表示0
- RS485电平 : 两线压差+2~-6V表示1, -2~-6V表示0 (差分信号)

对比项 RS232 RS485		
信号类型	单端信号 (单根信号线传输)	差分信号 (两根信号线传输)
传输线数量	最少 3 条 (TX, RX, GND), 可有更多控制线	2 条 (A+ 和 B-)
通信距离	最远 15 米	最远可达 1200 米
传输速率	短距离下速率较高, 最高 115.2 kbps 或更高	长距离传输时稳定, 最高速率一般较低
抗干扰性	抗干扰性差, 容易受电磁干扰影响	抗干扰性强, 差分信号有更好的抗噪音
通信模式	全双工 (发送和接收可以同时进行)	半双工 (发送和接收需要交替进行)
设备数量	仅支持点对点通信 (1 对 1)	支持多点通信(1 对多), 最多 32 台设备
使用场景	短距离、低速、点对点通信	长距离、抗干扰、多点通信



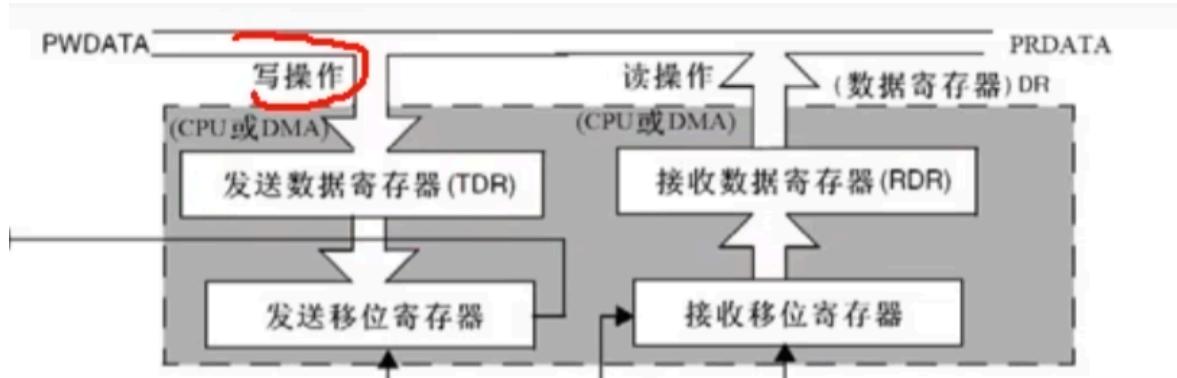
常见的波特率为4800、9600、115200

起始位0,停止位1(可以配置位0.5,1,1.5,2):这里的0.5位是指时间长短,每一位所需要的时间

USART数据寄存器(USART_DR)只有低9位有效,可以控制为8位或者9位,一般使用八位,这里的数据位是包含奇偶校验位的长度

串口通信的TX,RX,GND是必须要接的,如果有一方没有供电,则需要将stm32的供电接线给其他设备

- 发送数据低位先行,
- M 是9600,每一位的时间就是 $1/9600$,大概是104us



- 写数据流程

当往发送数据寄存器里面写值的时候,会检查发送移位寄存器有没有正在移动数据,若没有,则全部移动到发送移位寄存器里面,准备发送.

当数据从TDR全部到了发送移位寄存器的时候会置一个标志位TXE(TDR为空)

- 接收数据流程

接收移位寄存器会把数据一位一位的高位先行移动过来,当接收了一个字节,就直接整体转移到接收数据寄存器中.在转移过程中会置一个标志位RXNE(接收数据寄存器非空)

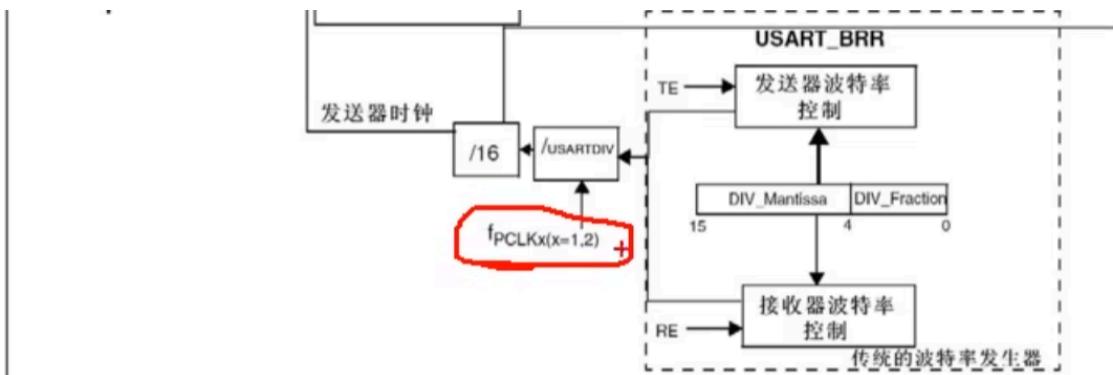
USART的SCLK

- 由于uart的SCLK时钟只能输出不能输入,两个USART之间不能实现同步的串口通信,这个时钟可以兼容SPI.
- 当不知道发送方的波特率的时候,可以测量一下时钟周期得到波特率

USART多设备

可以通过唤醒设备来实现多设备的功能,发送地址开始唤醒工作

USART时钟



USART1挂载在APB2,所以就是pclk2的时钟,72M.其他的USART都挂载在APB1,所以PCLK的时钟是36M

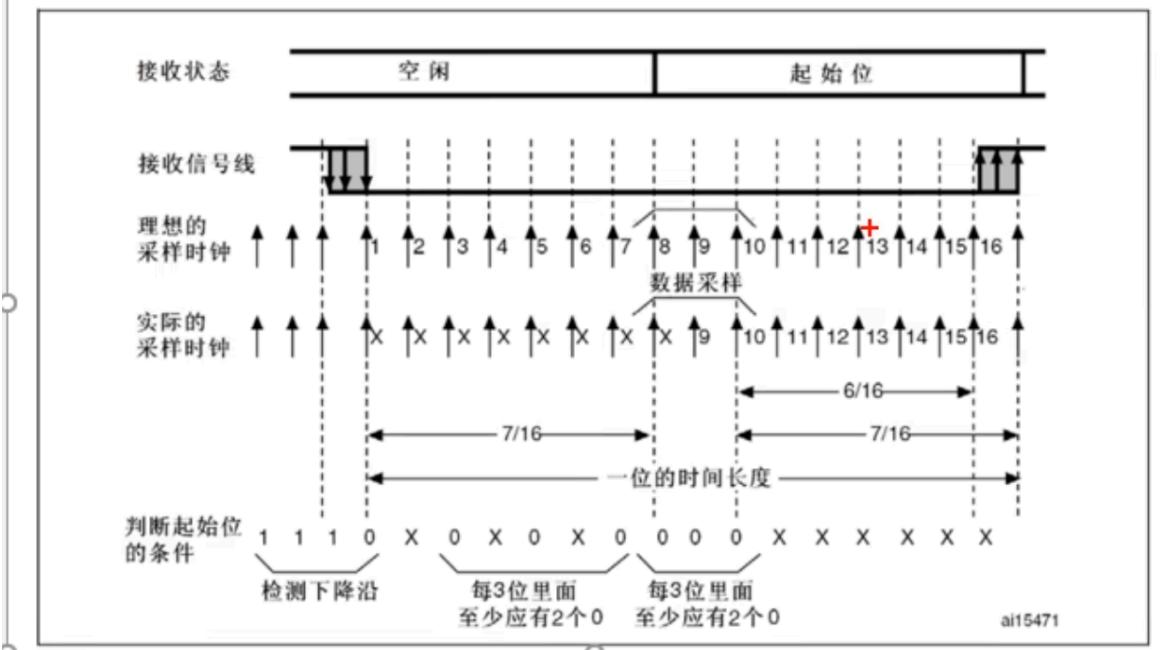
TE和RE分别是发送器波特率和接收器波特率

波特率发生器

- 波特率发生器起始就是分频器，APB时钟进行分频，得到发送和接收移位的时钟。图上时钟输入是 f_{PCLKx} ($x=1$ 或 2)，USART1挂载在APB2，所以就是PCLK2的时钟，一般是72MHz，其他的USART都挂载在APB1，所以是PCLK1的市长，一般是36MHz。

起始位侦测

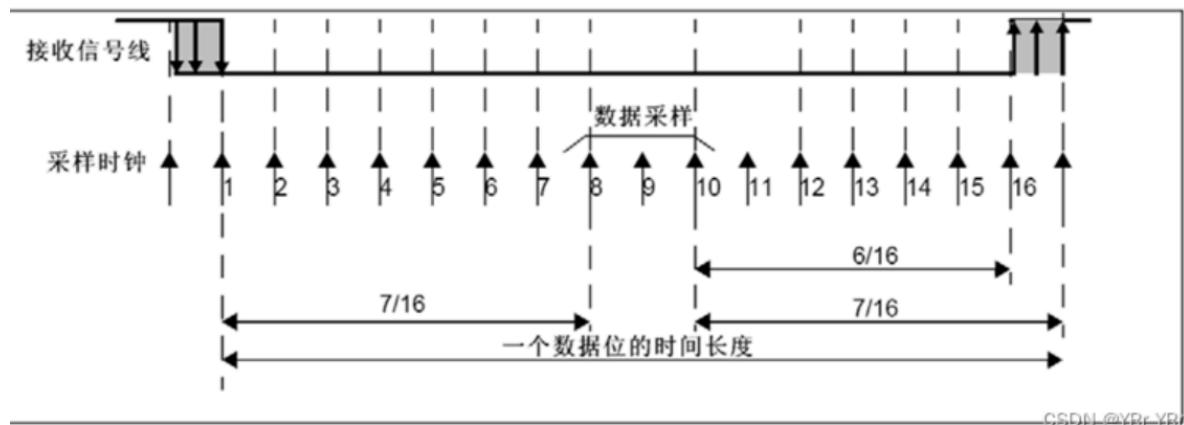
图252 起始位侦测



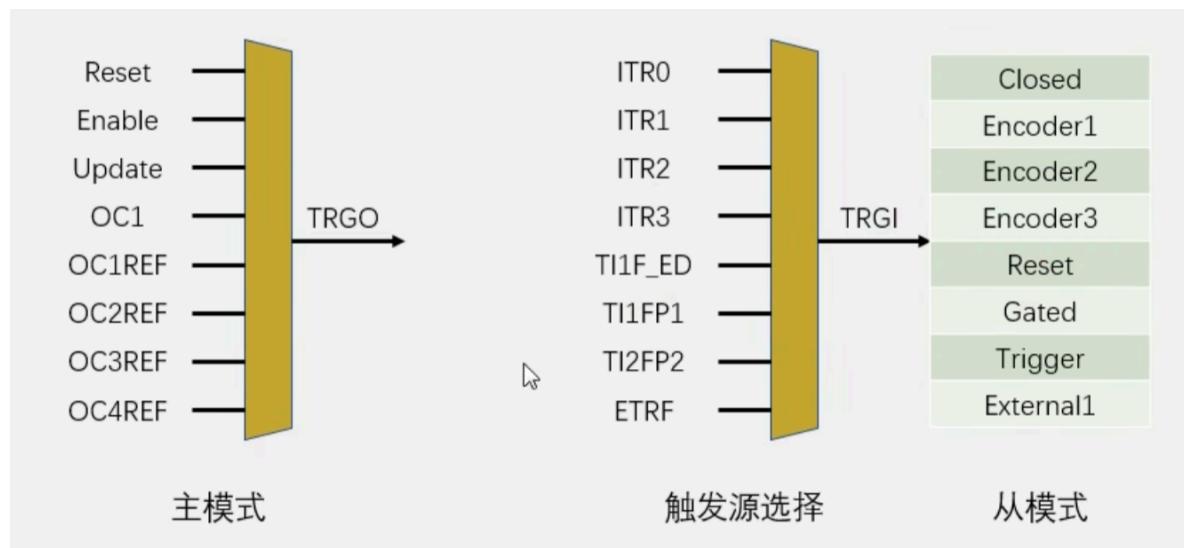
- 从起始位置开始采样位置就要对齐到位的正中间.检测到起始位置,以波特率的16倍进行采样,若没有噪声则连续16个都是0
 - 首先, 输入的这部分电路对采样时钟进行了细分, 它会以波特率的16倍频率进行采样, 也就是在一位的时间里, 可以进行16次采样。
 - 最开始, 空闲状态高电平, 采样就一直是1, 在某个位置, 突然采到一个0, 那么就说明, 在这两次采样之间, 出现了下降沿, 如果没有任何噪声, 那之后就应该是起始位了, 在起始位, 会连续进行16次采样, 没有噪声的话, 这16次采样, 肯定都是0, 就没有问题, 但是实际电路还是会存在一些噪声, 所以即使出现下降沿, 后续也要再采样几次, 以防万一。根据手册描述, 这个接收电路, 还会再下降沿之后的第3次、5次、7次, 进行一批采样, 在第8次、9次、10次, 再进行一批采样, 且这两批采样, 都要要求每3位里面至少应有2个0, 如果没有噪声, 那肯定全是0, 满足情况, 如果有一些轻微的噪声, 导致这里3位里面, 只有2个0, 另一个是1, 那也算是检测到了起始位, 但是在状态寄存器里会置一个NE (Noise Error), 噪声标志位, 就是提醒一下, 数据收到了, 但是有噪声。如果3位里面, 只有1个0, 就不算检测到了起始位, 可能前面那个下降沿是噪声导致的, 这时电路就忽略前面的数据重新开始捕捉下降沿, 这就是STM32的串口, 在接收过程中, 对噪声的处理。
 - 如果通过了起始位侦测, 那接收状态就由空闲, 变为接收起始位, 同时, 第8、9、10次采样的位置, 就正好是起始位的正中间, 之后, 就收数据位时, 就都在第8、9、10次进行采样, 这样就能保证采样位置在位的正中间了, 这就是起始位侦测和采样位置对齐的策略。

数据采样

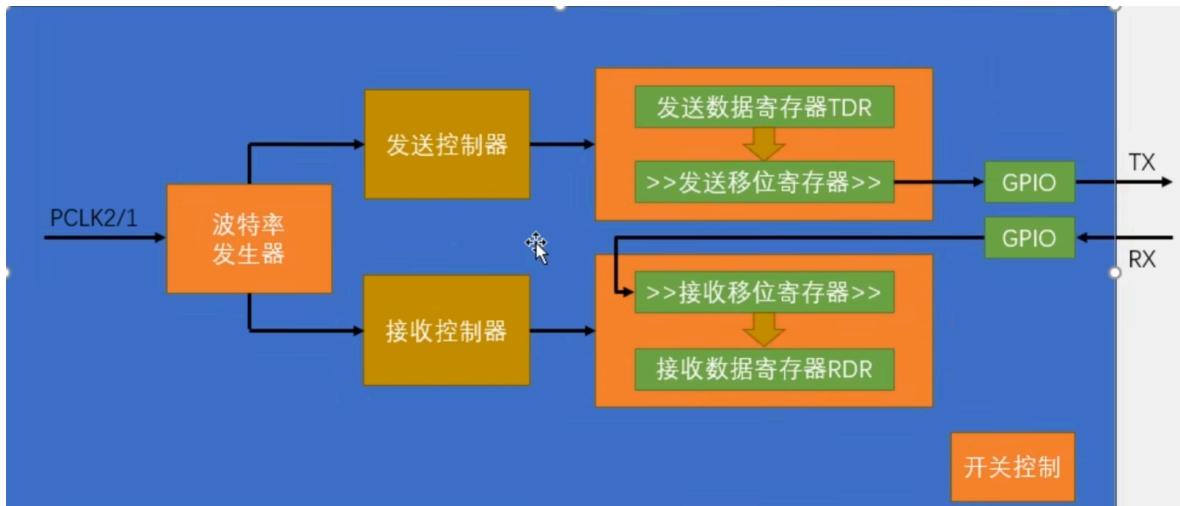
图253 检测噪声的数据采样



从1到16，是一个数据位的时间长度，在一个数据位，有16个采样时钟，由于起始位侦测已经对齐了采样时钟，所以这里直接在第8、9、10次采样数据位，为了保证数据的可靠性，这里是连续采样3次，没有噪声的理想情况下，这3次肯定全为1或者全为0，全为1，就认为收到了1，全为0，就认为收到了0。如果有噪声，导致3次采样不全为1或者全为0，那它就按照2：1的规则来，2次为1就认为收到了1，2次为0，就认为收到了0，在这种情况下，噪声标志位NE也会置1，这就是检测噪声的数据采样。



基本流程

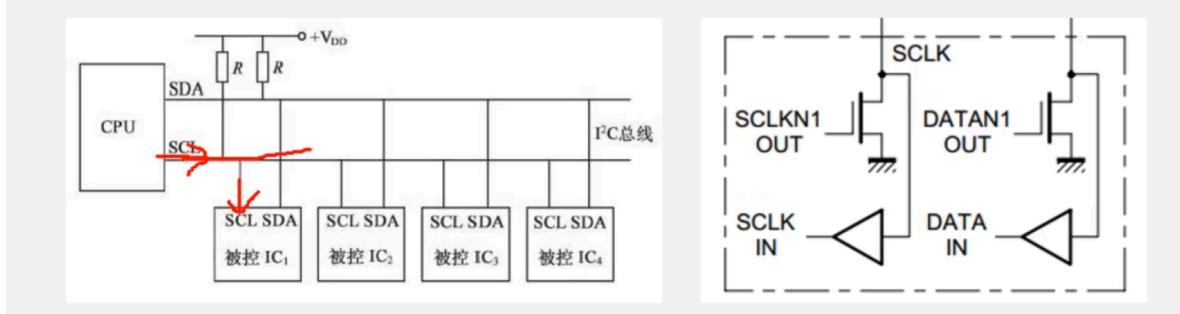


八、 i2c

- I2C总线 (Inter IC BUS) 是由Philips公司开发的一种通用数据总线
 - 两根通信线：SCL (Serial Clock) 、SDA (Serial Data)
 - 同步，半双工
 - 带数据应答
 - 支持总线挂载多设备 (一主多从、多主多从)
- 时钟速度一般100k, 快速是400khz, 高速模式3.4MHz(使用很少)
 - 高位先行

硬件电路：

- 所有I2C设备的SCL连在一起， SDA连在一起
- 设备的SCL和SDA均要配置成开漏输出模式
- SCL和SDA各添加一个上拉电阻，阻值一般为4.7KΩ左右



- SCL可以使用推挽输出,但是为了多主机模式还是选择上拉电阻+开漏输出
IIC总线设备需要上拉电阻(弱上拉):
 - 为了在设备不拉低信号时, 总线能够回到高电平状态, 必须通过外部上拉电阻将SCL和SDA拉高。
 - 为了避免总线没有协调好导致电源短路, 禁止所有设备输出强上拉
 - 避免频繁切换模式, 开漏 + 弱上拉的模式, 同时兼具了输入和输出的功能
 - 只要一个设备是低电平总线就是低电平, 只有所有设备都输出高电平总线才处于高电平, 实现多主机模式下的时钟同步和总线仲裁
 - 开漏输出加上拉电阻, 所以I2C信号的抗干扰能力是比较弱的, 它只适合于同一块电路板上的芯片之间进行通信, 并不适合超过30厘米电路板之间的通信
 - 避免高电平造成的引脚浮空
 - 数据帧大小只能八位

时序基本单元

- 发送数据的时候是高位先行, 有时间同步的情况下可以间断发送
默认的时候总线被弱上拉到高电平, 高位先行
SCL和SDA始终处于输入模式, 当主机需要接收的时候, 需要先释放SDA
 - 开始:SCL高电平, SDA高到低, 所以开始的时候SCL和SDA都必须是高电平
 - 结束:SCL高电平, SDA低到高
 - 主机发送:SCL低电平期间
 - SCL高电平期间读数据, 在SCL上升沿的时候从机就要读取SDA的数据了

主机接收数据:

1. 主机先释放SDA, 切换为输入模式, 因为如果主机此时动了SDA, 那么SDA将一直处于低电平
2. SCL低电平从机写数据, 高电平主机读数据

- 发送应答：主机在接收完一个字节之后，在下一个时钟发送一位数据，数据0表示应答，数据1表示非应答
- 接收应答：主机在发送完一个字节之后，在下一个时钟接收一位数据，判断从机是否应答，数据0表示应答，数据1表示非应答（主机在接收之前，需要释放SDA）

- 由于主机在等待应答之前要释放SDA给从机，所以释放之后SDA应该是高电平，此时应该等待从机拉低电平，所以应答位应该为0

- 指定地址写
 - 对于指定设备（Slave Address），在指定地址（Reg Address）下，写入指定数据（Data）



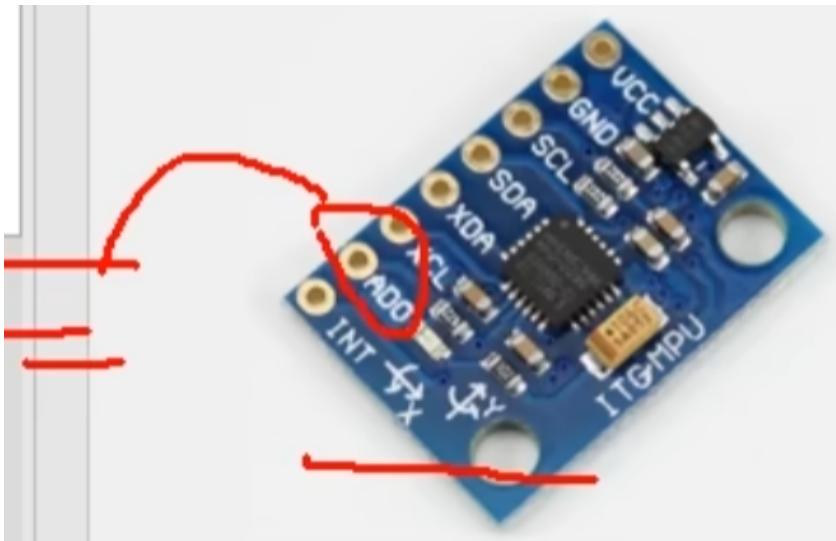
- 写入多个字节，最后不结束。
- 指定地址后地址指针会自动+1，变成0x1A



从机设备地址

- 主机在起始条件之后,要先发送一个字节叫一下从机名字.所有从机都会收到这个字节,主机地址分为7位地址和10位地址

若有相同的设备链接在电路上面则可以通过电路改变设备的地址号(AD0,接低电平就是0,高电平就是1),地址的最后几位是可以在电路中改变的



- 将AA0接入到低电平则会改变设备号的最后一位
- RA:0后面的电平变化是从机交出SDA控制的变化

开始位 +从机地址+读写位(7位+1位)

SCL低电平之间进行数据变换,高电平之间读取SDA

数据在SCL下降沿的时候进行改变, 上升沿的时候进行读取, 高电平期间保持稳定

上拉电阻越小速度越快;

上拉电阻在1K到10K之间吧, 太小容易击穿io口, 端口输出的低电平增大。太大的时候上升沿太慢。

当前地址读

对于指定设备 (Slave Address)，在当前地址指针指示的地址下，读取从机数据 (Data)



主机读取的时候先发送一个字节,来进行从机的寻址和指定读写标志位(最后一
位):1是读,0是写

- 当前地址指针:所有的寄存器被分配到了一个线性区域中(类似于数组),上电的时候默认指针指向0的位置,每次写入和读出一个字节后,指针自动自增.每次写入字节,会把指针移动到对应位置
- 当前地址指针是存储在从设备内部的一个计数器, 用于记录当前正在访问的寄存器或内存地址。它的主要作用是简化I2C通信过程, 允许主机设备在连续读写操作中高效地访问从设备的多个寄存器, 而无需在每次操作时都重新发送完整的寄存器地址

I2C时序

- 指定地址读
- 对于指定设备 (Slave Address) 下, 读取从机数据 (Data)



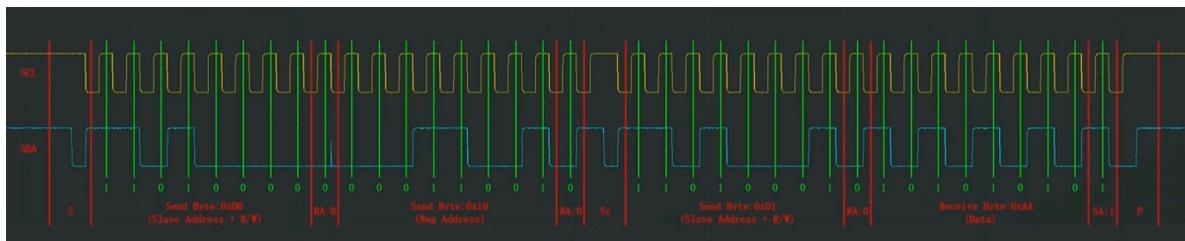
就是指定地址写但是不进行写操作+当前地址读

sr:重复起始条件

- 先正常程序,起始位+设备地址+读写位置+回应位+读的地址(这个会改变当前地址的指针位置)
- 之后再次进行当前地址读程序
- sr:重复起始条件 因为指定标志位只能是跟着起始条件后的第一个字节，所以这里必须再次开始改变读写状态
- 只想读一个字节的话,在读取了第一个字节后必须发送一个非应答回复

软件I2C

- 波形规整



SCL下降沿的时候就开始写入数据了,上升沿的时候进行读取

iic通信速率和空闲时间

I^C传输位速率在标准模式下可达100Kbit/s，快速模式下可达400Kbit/s，高速模式下可达3.4Mbit/s；也可以理解为时钟频率在标准模式下可达100kHz，快速模式下可达400kHz，高速模式下可达3.4MHz。

I^C一个周期只能传输一位数据，因为只能低电平传输，高电平读取

I^C总线有两种空闲状态：

①第一种是设备第一次启动前的空闲状态，其满足条件为：SDA和SCL同为高电平，且保持时间较长（此处官方没给出具体参数，但笔者认为不应小于第二种空闲状态高电平保持时间tBUF）；

②第二种是设备停止总线后，再启动前的空闲状态，其满足条件为：SDA和SCL同为高电平，且保持时间 $\geq t_{BUF}$ （ t_{BUF} ：标准模式 $\geq 4.7\mu s$ ，快速模式 $\geq 1.3\mu s$ ，快速增强模式 $\geq 0.5\mu s$ ，超快模式 $\geq 80ns$ ）。

值得注意的是，I²C总线在工作中的热启动之前（SDA和SCL同为高电平）的状态，协议判定为忙碌状态而非空闲状态。

最多可以接多少从机

I²C总线支持多个从机设备连接到同一总线上，理论上最多可以连接的从机数量取决于地址位的长度：

- **7位地址模式**：最多支持127个从机（地址范围为0x00到0x7F）。
- **10位地址模式**：最多支持1023个从机。

软件IIC和硬件IIC

硬件IIC效率高，实现简单，可以用DMA.

软件IIC容易移植且更稳定。

iic 总线仲裁与实现

仲裁机制

基于线与逻辑：I²C总线的仲裁机制基于总线的“线与”逻辑功能，即当多个设备同时向总线发送信号时，总线上呈现的信号是所有发送信号的逻辑与结果。

低电平优先原则：在仲裁过程中，遵循“低电平优先”的原则。如果多个主设备在发送同一数据位时，有设备发送低电平而其他设备发送高电平，则发送低电平的设备将赢得仲裁，继续控制总线。

逐位仲裁过程：

启动条件：当SCL线为高电平时，SDA线由高电平跳变到低电平表示启动通信。

发送地址：主机发送一个设备地址（7位）加上一个读/写位（1位），以指定要通信的从设备和操作类型。每个试图获得总线控制权的主机都会开始发送地址字节。

仲裁：在发送地址字节时，所有尝试访问总线的主机将发送地址和控制位，并进行逐位比较。如果一个主机发送的是逻辑“1”而检测到的是逻辑“0”，则表明有更高优先级的主机正在使用总线。拥有最低地址的主机将获得总线控制权。

结果：最后一个发送成功地址字节的主机会赢得仲裁，获得总线的控制权。

败方停止：检测到仲裁失败的主机将停止发送数据，并释放总线，等待总线空闲后才能再次尝试通信。

仲裁实现

硬件实现：

开漏输出与上拉电阻： I^2C 总线的SDA和SCL线通常采用开漏输出，并通过外部上拉电阻来实现高电平状态。这种结构使得当多个设备同时向总线发送信号时，总线能够表现出“线与”的逻辑功能。

时钟同步： I^2C 总线还通过时钟同步机制来协调不同设备之间的操作。所有主设备在SCL线上产生它们自己的时钟信号，但这些时钟信号通过线与逻辑在总线上表现为统一的时钟信号。

软件实现：在软件层面，主设备需要在发送数据的同时，实时检测SDA线上的电平状态。如果检测到的电平与自己发送的电平不一致，则表明仲裁失败，需要停止发送数据并释放总线

从机如果没有发送应答位可能有几种情况

从机地址完全一样怎么进行区分呢

SPI

- 四根通信线:**SCK** **MOSI**(主机输出从机输入) **MISO**(主机输入从机输出) **SS**(从机选择)
- 全双工,同步
- 不支持多主机模式**

和I²C对比

- 1.速度更快,因为推挽输出高低电平变换时间更低
- 2.设计简单
- 3.硬件开销大
- 4.没有起始位和停止位, 可以连续发送任意数量的位

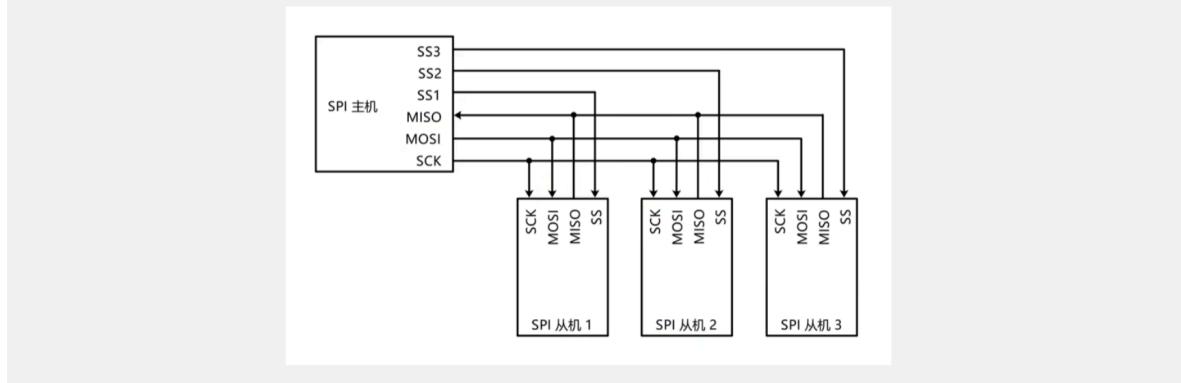
上拉电阻对速率的影响

1. 电阻越小, 速率越高

提高时钟速度可以提高速率

硬件电路

- 所有SPI设备的SCK、MOSI、MISO分别连在一起
- 主机另外引出多条SS控制线, 分别接到各从机的SS引脚
- 输出引脚配置为推挽输出, 输入引脚配置为浮空或上拉输入



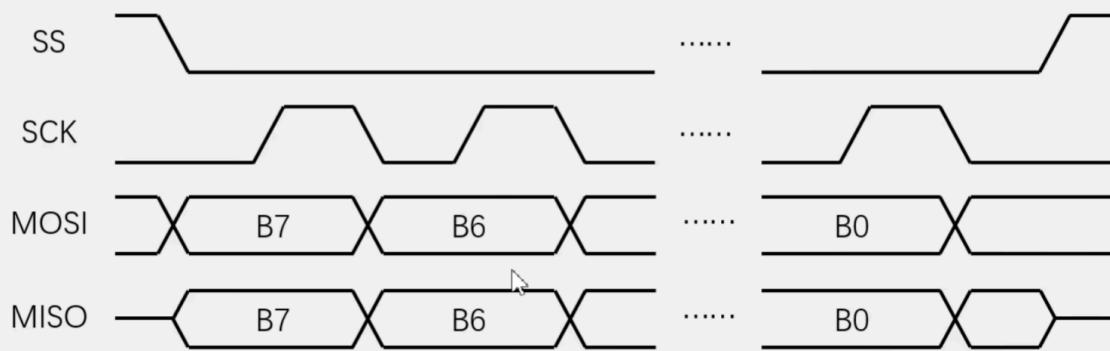
- SS:从机选择,选择那个从机就把主机对应的SS口置为低电平,当通信结束置高电平,同一时间只能选择一个从机
- 传播速度快,因为推挽输出的驱动能力强,低电平到高电平的时间快,同理高到低也是
- 为什么I2C选择开漏输出呢,1:要频繁切换为输入输出 2.避免电流短路.3实现多主机的时钟同步和总线仲裁

- 对于MISO,当SS为高电平的时候,MISO引脚必须为高阻态(关闭输出),防止一条线有多个输出而导致的电平冲突问题。结束之后MISO,从机必须得置回高阻态
- 上图用箭头标出哪个是输出那个是输入

SPI时序基本单元

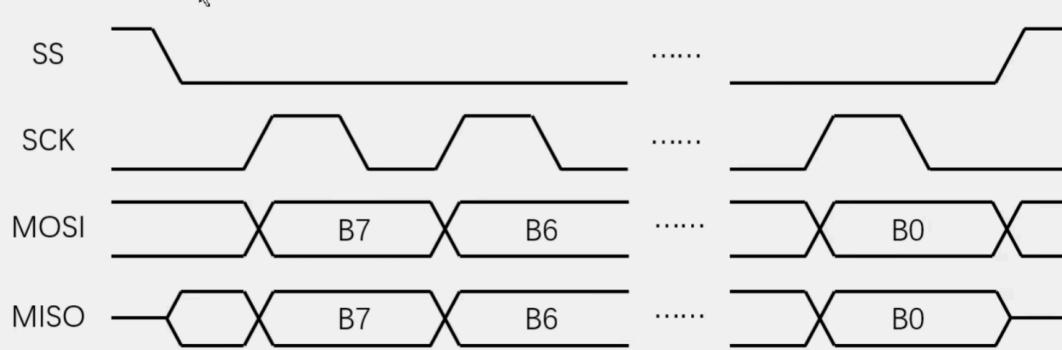
1

- 交换一个字节 (模式0)
- CPOL=0: 空闲状态时, SCK为低电平
- CPHA=0: SCK第一个边沿移入数据, 第二个边沿移出数据



第一个边沿之前就要移出数据

- 交换一个字节 (模式1)
- CPOL=0: 空闲状态时, SCK为低电平
- CPHA=1: SCK第一个边沿移出数据, 第二个边沿移入数据



- 对比模式1和模式0,模式0比模式1的交换数据早半个周期,不然等到第一个上升沿的时候没有数据移入,模式0的应用是最多的

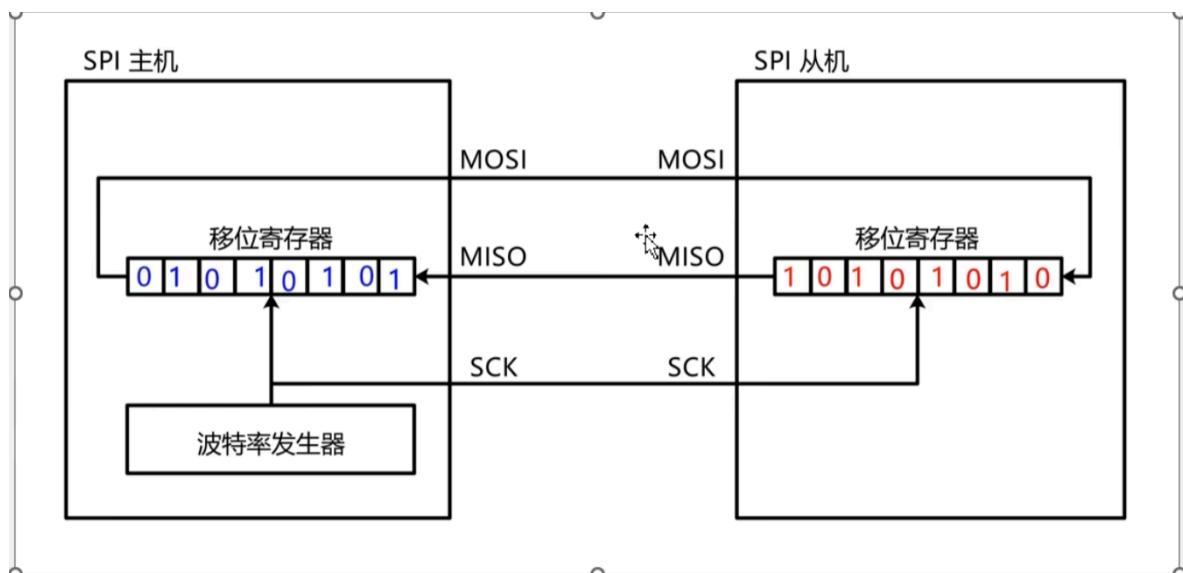
- 表示高阻态

- 交换一个字节（模式2）
 - CPOL=1：空闲状态时，SCK为高电平
 - CPHA=0：SCK第一个边沿移入数据，第二个边沿移出数据

- 交换一个字节（模式3）
 - CPOL=1：空闲状态时，SCK为高电平
 - CPHA=1：SCK第一个边沿移出数据，第二个边沿移入数据

CPHA:规定第一个边沿移入还是第二个边沿移入

工作流程图：



spi是高位先行。每来一个时钟，移位寄存器就会向左进行移位。主机移位寄存器左边移出去的数据，通过miso引脚输入到从机移位寄存器的右边。

波特率上升沿，所有移位寄存器就会向左进行移位，放在引脚上面。

- SCK的上升沿进行数据发送,下降沿进行接收

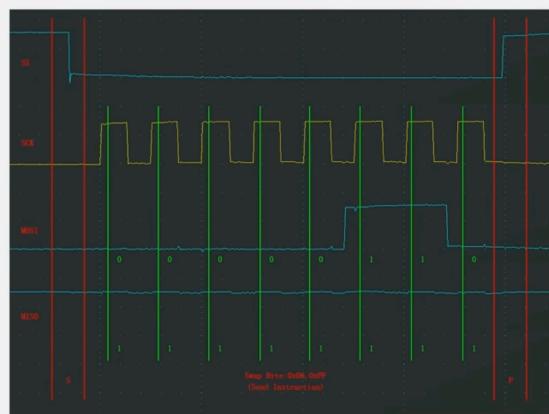
起始条件:SS从高电平到低电平

终止条件:SS从低电平到高电平

空闲状态的时候SCK为低电平

SPI时序

- 发送指令
- 向SS指定的设备，发送指令（0x06）



W25Q64里面0x06是写使能

跟I2C不同的是,SPI是在开始后发送指令集+读写数据

指令集在从机中会进行定义,我们可以在第一个字节发送我们想要发送的命令

- 在指定地址写入数据
 - 指定地址写
 - 向SS指定的设备，发送写指令（0x02），随后在指定地址（Address[23:0]）下，写入指定数据（Data）

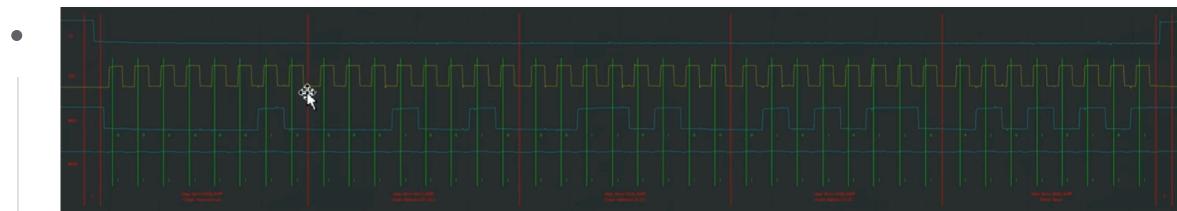


模式0:在ss下降沿，SCK第一个上升沿之前进行交换数据,然后在后面的上升沿进行采样数据,下降沿发送数据

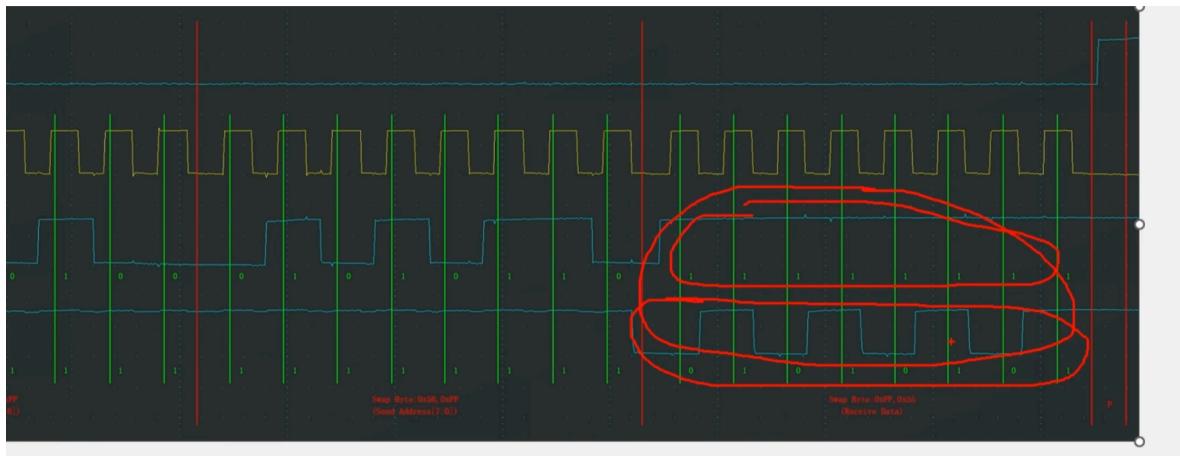
如果在指令集后面还有地址的话,在最后一次采样数据后的下降沿继续发送数据即可

上面这个图就是用0x06换回0xFF,也就是发送了写使能,0xFF不需要看

- 在指定地址写



- 1.先发送0x02换回0xFF ,0x22是写数据的指令
- 2.在最后一个下降沿把下一个字节的最高位放在MOSI上面,用0x12换来了0xff。一共三个字节：0x123456
- 3.最后表示写入数据
- 在指定地址读取数据



前面的和写数据是一样的,先写指令+指定地址 后面用0XFF跟从机的指定地址交换数据即可

- 若在发送数据之后继续发送0XFF,则从机会把指定地址之后的数据发送给主机
- 读数据和写数据一样流程，都是交换数据

SPI硬件

- APB2的时钟是72MHZ,APB1的时钟是36MHZ

时钟系统

STM32 有4个独立时钟源:HSI、HSE、LSI、LSE。

- ①、HSI是高速内部时钟，RC振荡器，频率为8MHz，精度不高。
- ②、HSE是高速外部时钟，可接石英/陶瓷谐振器，或者接外部时钟源，频率范围为4MHz~16MHz。比内部更加稳定
- ③、LSI是低速内部时钟，RC振荡器，频率为40kHz，提供低功耗时钟。
- ④、LSE是低速外部时钟，接频率为32.768kHz的石英晶体。

LSI是作为IWDGCLK(独立看门狗)时钟源和RTC时钟源 而独立使用

HSI高速内部时钟 HSE高速外部时钟 PLL锁相环时钟 这三个经过分频或者倍频 作为系统时钟来使用

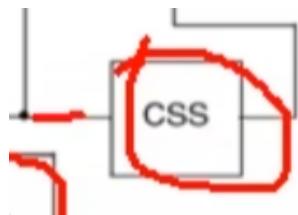
PLL为锁相环倍频输出，其时钟输入源可选择为HSI/2、HSE或者HSE/2。倍频可选择为2~16倍，但是其输出频率最大不得超过72MHz。通过倍频之后作

为系统时钟的时钟源

SystemInit函数(配置时钟)

1.启动内部时钟，选择内部8MHZ为系统时钟，暂时以内部8MHZ的时钟运行。

2.启动外部时钟，进行锁相环倍频输出得到72MHZ,等到稳定之后再选择这个72MHZ输出为系统时钟



时钟安全系统负责切换时钟的，当外部时钟失效之后自动切换为内部时钟

DMA

- DMA可以提供外设和存储器或者存储器和存储器之间的高速数据传输，无须CPU干预，节省了CPU的资源
- 12个独立可配置的通道： DMA1 (7个通道) , DMA2 (5个通道)

代码设置通道

```
DMA_Cmd(DMA1_Channel1, ENABLE); // DMA1的通道1使能  
ADC_DMACmd(ADC1, ENABLE); // ADC1触发DMA1的信号使能
```

类型	起始地址	存储器	用途
ROM	0x0800 0000	程序存储器Flash	存储C语言编译后的程序代码
	0x1FFF F000	系统存储器	存储BootLoader, 用于串口下载
	0x1FFF F800	选项字节	存储一些独立于程序代码的配置参数
RAM	0x2000 0000	运行内存SRAM	存储运行过程中的临时变量
	0x4000 0000	外设寄存器	存储各个外设的配置参数
	0xE000 0000	内核外设寄存器	存储内核各个外设的配置参数

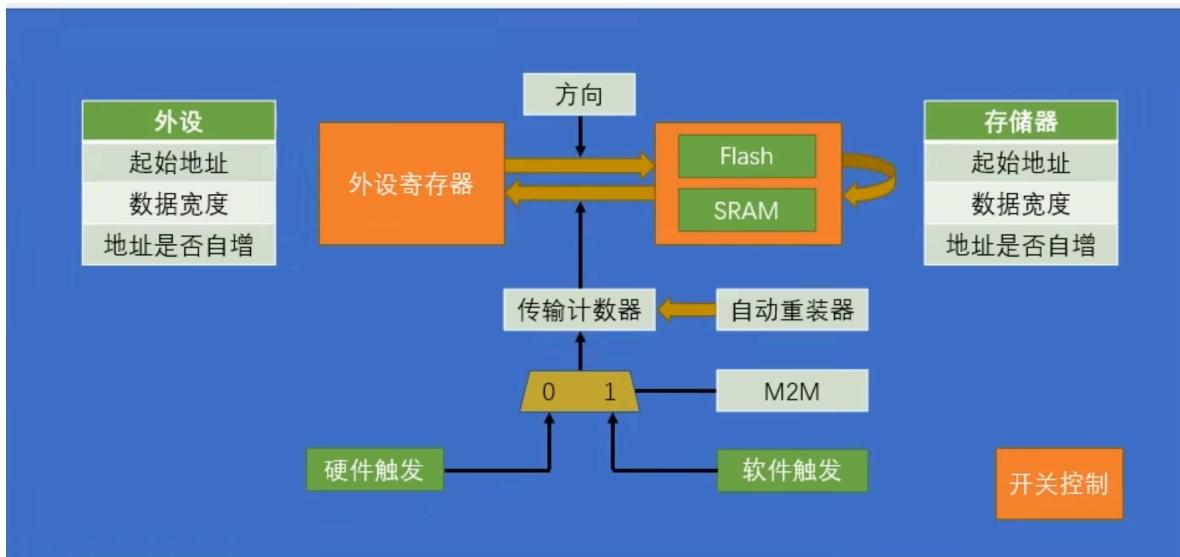
仲裁器

仲裁器管理DMA通道请求分为两个阶段。 第一阶段属于软件阶段，可以在 DMA_CCRx 寄存器中设置

第二阶段属于硬件阶段， 如果两个或以上的DMA通道请求设置的优先级一样，则他们优先级取决于通道编号， 编号越低优先权越高

DMA1控制器拥有高于DMA2控制器的优先级。

- CPU或者DMA直接访问Flash的话,是只可以读而不可以写的



- 对于数据宽度:字节(8位),半字(16位),字(32位)
- 软件触发和自动重装器不能同时使用,软件触发是以最快的速度连续不断的触发DMA,一般只应用在存储器到存储器的情况
- 当数据宽度不一样的时候,小的大的高位补0,大的小的,舍弃高位.

串口dma，什么时候dma去进行搬运数据？

- 1.当发送数据寄存器为空的时候
- 2.当接收数据寄存器

基地址，目标地址。传输数据单位是什么，传多少数据，是一次传输还是循环传输

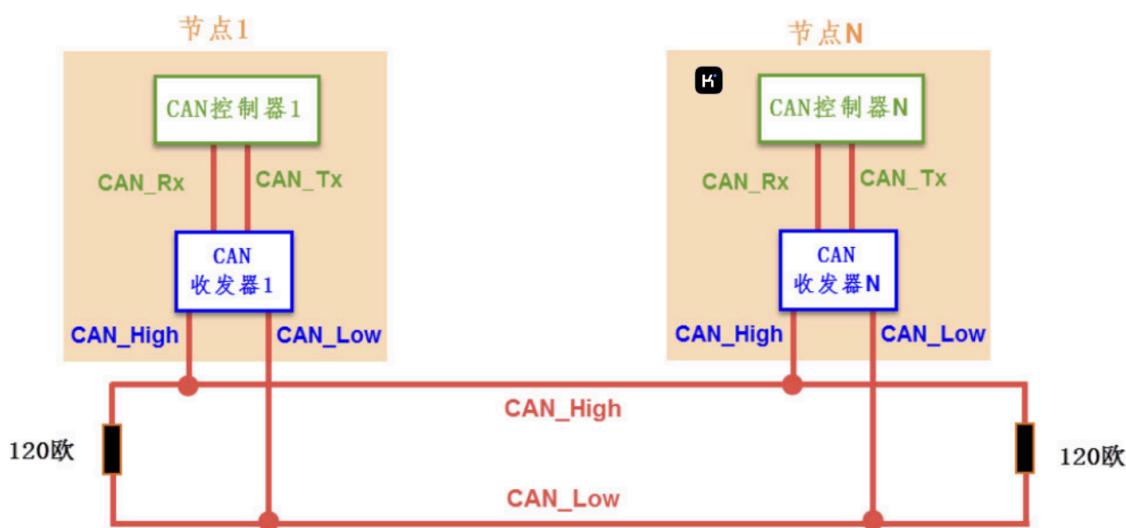
什么时候传输完成

数据什么时候传输完成，我们可以通过查询标志位或者通过中断的方式来鉴别。每个DMA通道在DMA传输过半、传输完成和传输错误时都会有相应的标志位，如果使能了该类型的中断后，则会产生中断。

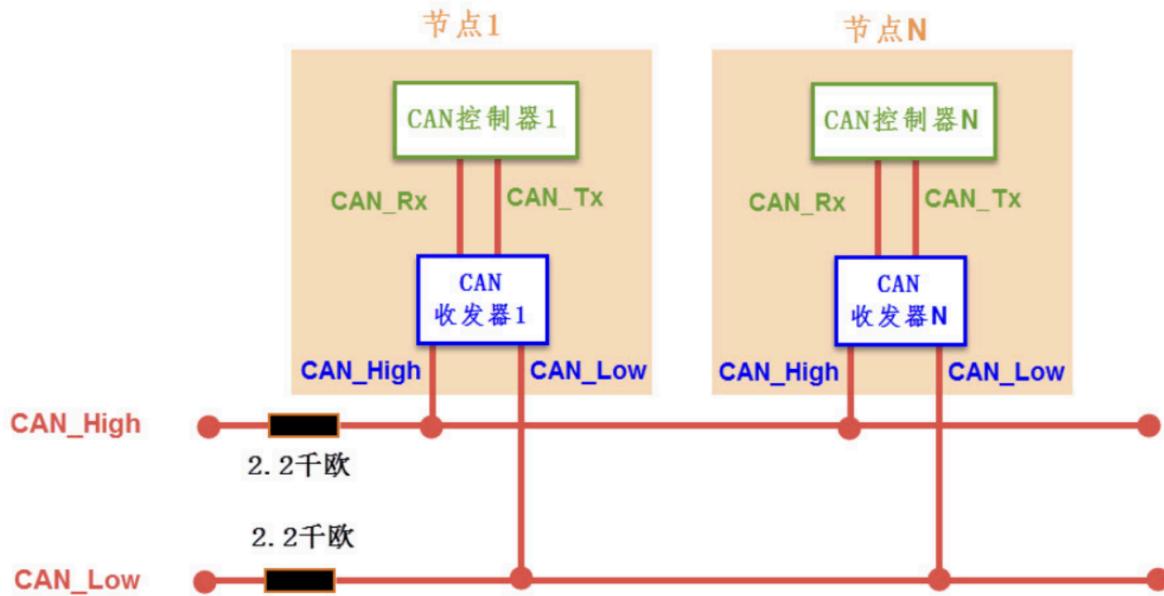
十、CAN总线

简介

- 异步通讯,只具有 CAN_High 和 CAN_Low 两条信号线，共同构成一组差分信号线，以差分信号的形式进行通讯
- CAN 物理层的形式主要有两种，图中的 CAN 通讯网络是一种遵循 ISO11898 标准的高速、短距离“闭环网络”，它的总线最大长度为 40m，通信速度最高为 1Mbps，总线的两端各要求有一个“120 欧”的电阻。



- 图中的是遵循 ISO11519-2 标准的低速、远距离“开环网络”，它的最大传输距离为 1km，最高通讯速率为 125kbps，两根总线是独立的、不形成闭环，要求每根总线上各串联有一个“2.2千欧”的电阻。



- CAN 通讯节点由一个 CAN 控制器及 CAN 收发器组成，控制器与收发器之间通过 CAN_Tx 及 CAN_Rx 信号线相连，收发器与 CAN 总线之间使用 CAN_High 及 CAN_Low 信号线相连。其中 CAN_Tx 及 CAN_Rx 使用普通的类似 TTL 逻辑信号，而 CAN_High 及 CAN_Low 是一对差分信号线，使用比较特别的差分信号

电阻作用：

- 如果不加电阻,信号波形会在线路终端反射,进而干扰原信号,最终波形可能都会有毛刺
- 在闭环电路中,在没有操作的时候将两根线的电压拉到同一水平,变成默认状态1
- 高速can回归隐形电平速度更快所以速度更快

11.看门狗

单片机STM32的看门狗（Watchdog）是一种硬件定时器，用于监控系统的运行状态并在出现故障或死锁时采取措施以恢复正常操作。看门狗的主要功能是

定期检查系统是否正常运行，并在系统出现问题时触发复位操作。

独立看门狗，也叫宠物狗，IWDG.

它是一个独立的硬件模块，可以在系统内部独立运行。通过配置IWDG定时器的计数器和预分频器，可以设置看门狗的定时时间。当看门狗定时器计数器达到预设的值时，会产生看门狗超时事件，触发系统复位。

独立看门狗用通俗一点的话来解释就是一个12位的递减计数器，当计数器的值从某个值一直减到0的时候，系统就会产生一个复位信号，即IWDG_RESET。如果在计数没减到0之前，刷新了计数器的值的话，那么就不会产生复位信号，这个动作就是我们经常说的喂狗。看门狗功能由VDD电压域供电，在停止模式和待机模式下仍能工作。

适合时间精度较低，时钟一般取40KHZ

重装载寄存器：12位 RLR

预分频寄存器PR

34.2.5. 键寄存器

键寄存器IWDG_KR可以说是独立看门狗的一个控制寄存器，主要有三种控制方式，往这个寄存器写入下面三个不同的值有不同的效果。

键值	键值作用
0XAAAA	把 RLR 的值重装载到 CNT
0X5555	PR 和 RLR 这两个寄存器可写
0XCCCC	启动 IWDG

通过写往键寄存器写0XCCC来启动看门狗是属于软件启动的方式，一旦独立看门狗启动，它就关不掉，只有复位才能关掉。

窗口看门狗定时器 (WWDG) , 警犬

它允许在特定的时间窗口内更新计数器值，以避免触发复位。WWDG可以通过设定窗口和计数器的值来进行配置，并在每次更新计数器时，确保计数

器值位于设定的窗口范围内。如果计数器超出窗口范围，将触发看门狗复位。

窗口看门狗跟独立看门狗一样，也是一个递减计数器不断的往下递减计数，当减到一个固定值0X40时还不喂狗的话，产生复位，这个值叫窗口的下限，是固定的值，不能改变。不同的地方是窗口看门狗的计数器的值在减到某一个数之前喂狗的话也会产生复位，这个值叫窗口的上限，上限值由用户独立设置。窗口看门狗计数器的值必须在上窗口和下窗口之间才可以喂狗，这就是窗口看门狗中窗口两个字的含义

上窗口的值可以改变

如果我们要监控的程序段A运行的时间为Ta，当执行完这段程序之后就要进行喂狗，如果在窗口时间内没有喂狗的话，那程序就肯定是出问题了。一般计数器的值TR设置成最大0X7F，窗口值为WR，计数器减一个数的时间为T，那么时间：(TR-WR)*T应该稍微小于Ta即可，这样就能做到刚执行完程序段A之后喂狗，起到监控的作用，这样也就可以算出WR的值是多少。

- CAN总线采用差分信号，即两线电压差 ($V_{CAN_H} - V_{CAN_L}$) 传输数据位
- 高速CAN规定：

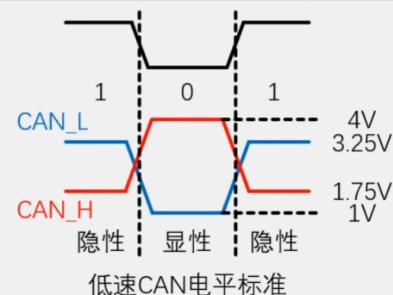
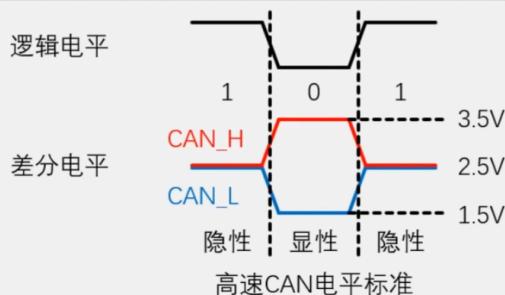
电压差为0V时表示逻辑1（隐性电平）

电压差为2V时表示逻辑0（显性电平）

- 低速CAN规定：

电压差为-1.5V时表示逻辑1（隐性电平）

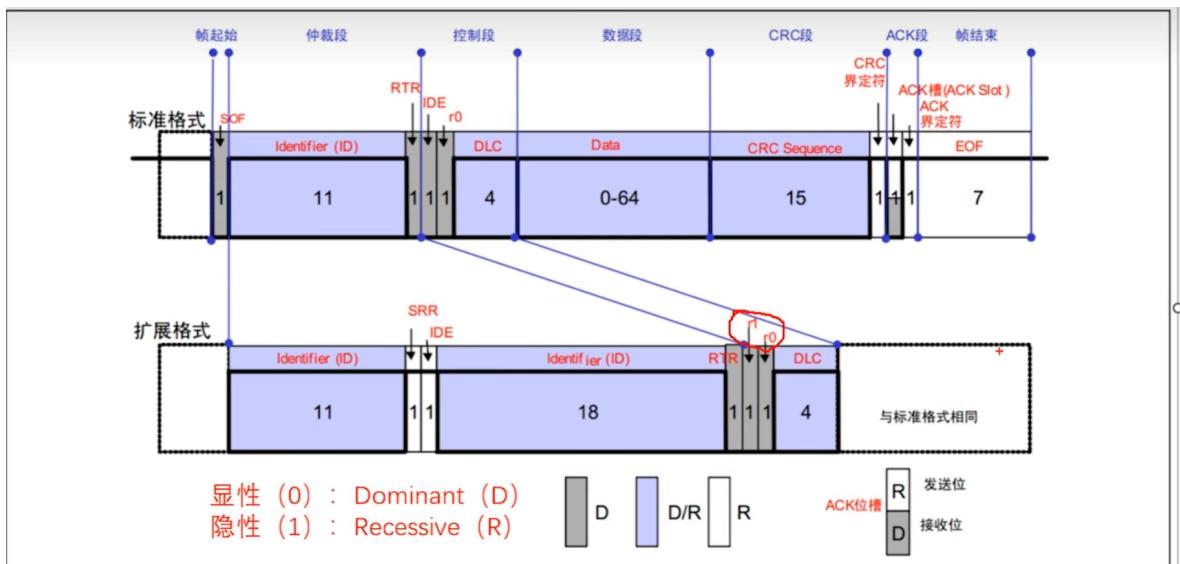
电压差为3V时表示逻辑0（显性电平）



- 因为电路中0强于1的规定，所以0是显性电平，1是隐形电平

- 高速CAN (ISO11898) : 125k~1Mbps, <40m
- 低速CAN (ISO11519) : 10k~125kbps, <1km
- 异步, 无需时钟线, 通信速率由设备各自约定
- 半双工, 可挂载多设备, 多设备同时发送数据时通过仲裁判断先后顺序
- 11位/29位报文ID, 用于区分消息功能, 同时决定优先级
- 可配置1~8字节的有效载荷
- 可实现广播式和请求式两种传输方式
- 应答、CRC校验、位填充、位同步、错误处理等特性

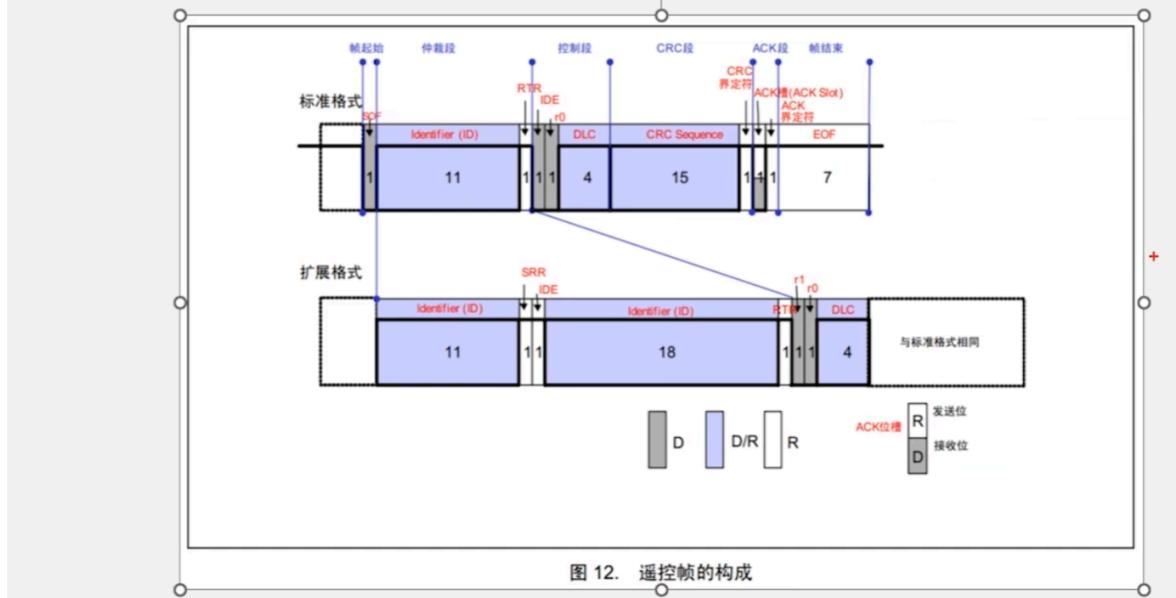
帧类型	用途
数据帧	发送设备主动发送数据 (广播式)
遥控帧	接收设备主动请求数据 (请求式)
错误帧	某个设备检测出错误时向其他设备通知错误
过载帧	接收设备通知其尚未做好接收准备
帧间隔	用于将数据帧及遥控帧与前面的帧分离开



- SOF (Start of Frame)：帧起始，表示后面一段波形为传输的数据位
- ID (Identify)：标识符，区分功能，同时决定优先级
- RTR (Remote Transmission Request)：远程请求位，区分数数据帧和遥控帧
- IDE (Identifier Extension)：扩展标志位，区分标准格式和扩展格式
- SRR (Substitute Remote Request)：替代RTR，协议升级时留下的无意义位
- r0/r1 (Reserve)：保留位，为后续协议升级留下空间
- DLC (Data Length Code)：数据长度，指示数据段有几个字节
- Data：数据段的1~8个字节有效数据
- CRC (Cyclic Redundancy Check)：循环冗余校验，校验数据是否正确
- ACK (Acknowledgement)：应答位，判断数据有没有被接收方接收
- CRC/ACK界定符：为应答位前后发送方和接收方释放总线留下时间
- EOF (End of Frame)：帧结束，表示数据位已经传输完毕

- IDE:显性0是标准数据帧,1是扩展帧数
- 保留位必须是0:仲裁时候0的优先级更高,之后如果用到了这个保留位.可以保证定义的优先级高于以后扩展定义的优先级.比如现在IDE默认是0,所以标准帧优先级高于扩展优先级这个特性
- can总线也是高位先行
- ID号+RTR是仲裁的依据 数据帧优先级比遥控帧高
- crc界定符时候，发送方必须发隐性。除了做一个分割，发送方必须释放总线
- EOF:7个1
- 仲裁是先比较id，再比较RTR.所以RTR必须在所有id位的后面。收到IDE之前标准帧和扩展帧格式要保证没有区别，所以存在SRR位置

- 遥控帧无数据段, RTR为隐性电平1, 其他部分与数据帧相同



- 即使遥控帧没有数据段,还是可以指定DLC进行后续data字节

位填充

2 位填充

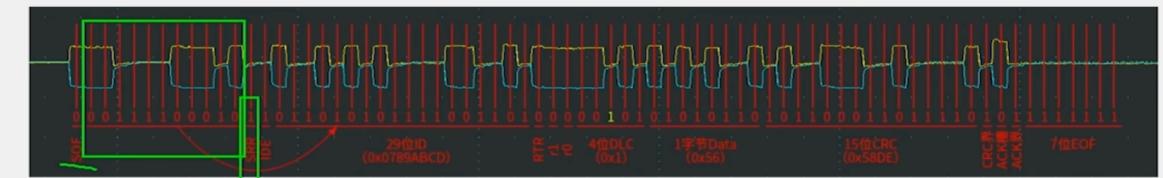
江协科技
jiangxiekeji.com

- 位填充规则: 发送方每发送5个相同电平后, 自动追加一个相反电平的填充位, 接收方检测到填充位时, 会自动移除填充位, 恢复原始数据
- 例如:

即将发送:	100000110	10000011110	011111111110
实际发送:	1000001110	1000001111100	011111011111010
实际接收:	1000001110	1000001111100	011111011111010
移除填充后:	100000110	10000011110	0111111111110
- 位填充作用:
 - 增加波形的定时信息, 利于接收方执行“再同步”, 防止波形长时间无变化, 导致接收方不能精确掌握数据采样时机
 - 将正常数据流与“错误帧”和“过载帧”区分开, 标志“错误帧”和“过载帧”的特异性
 - 保持CAN总线在发送正常数据流时的活跃状态, 防止被误认为总线空闲

扩展数据帧

- 扩展数据帧，报文ID为0x0789ABCD，数据长度1字节，数据内容为0x56

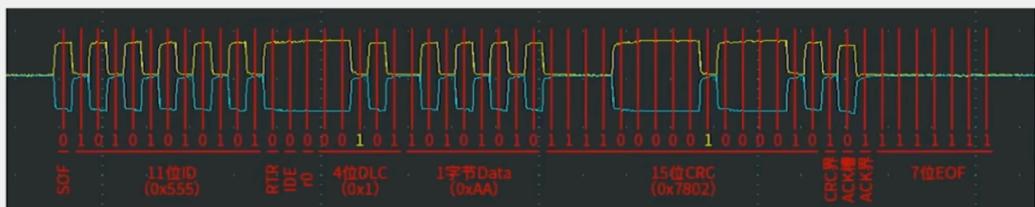


标准帧

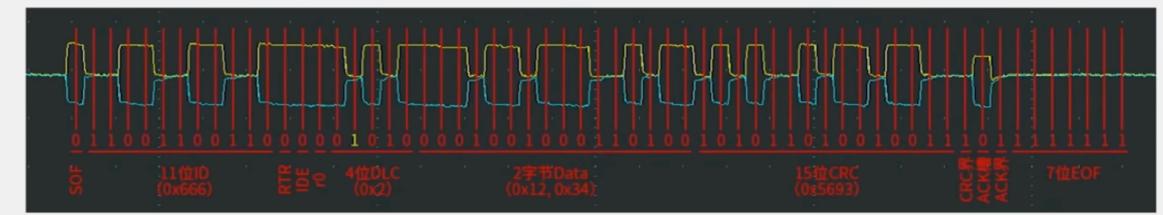
2 波形实例



- 标准数据帧，报文ID为0x555，数据长度1字节，数据内容为0xAA



- 标准数据帧，报文ID为0x666，数据长度2字节，数据内容为0x12, 0x34



过载帧

2 过载帧



- 当接收方收到大量数据而无法处理时，其可以发出过载帧，延缓发送方的数据发送，以平衡总线负载，避免数据丢失

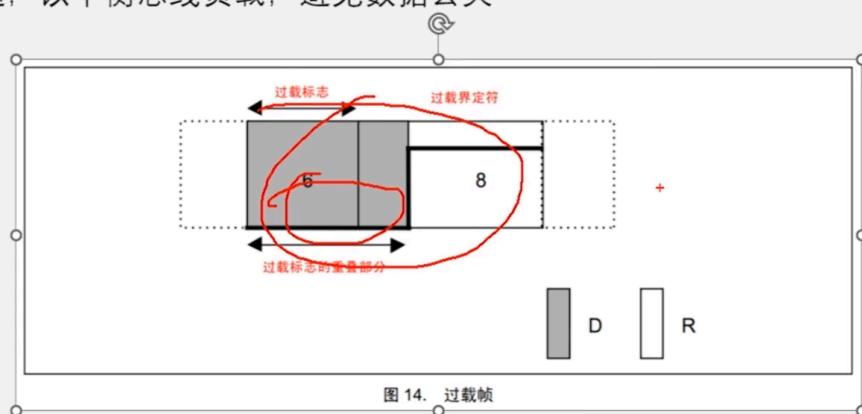
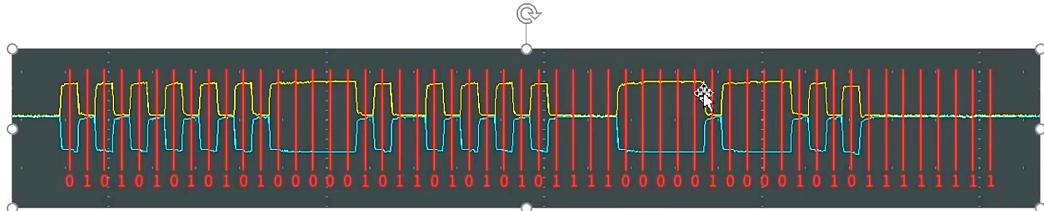


图 14. 过载帧

3 接收方数据采样

- CAN总线没有时钟线，总线上的所有设备通过约定波特率的方式确定每一个数据位的时长
- 发送方以约定的位时长每隔固定时间输出一个数据位
- 接收方以约定的位时长每隔固定时间采样总线的电平，输入一个数据位
- 理想状态下，接收方能依次采样到发送方发出的每个数据位，且采样点位于数据位中心附近

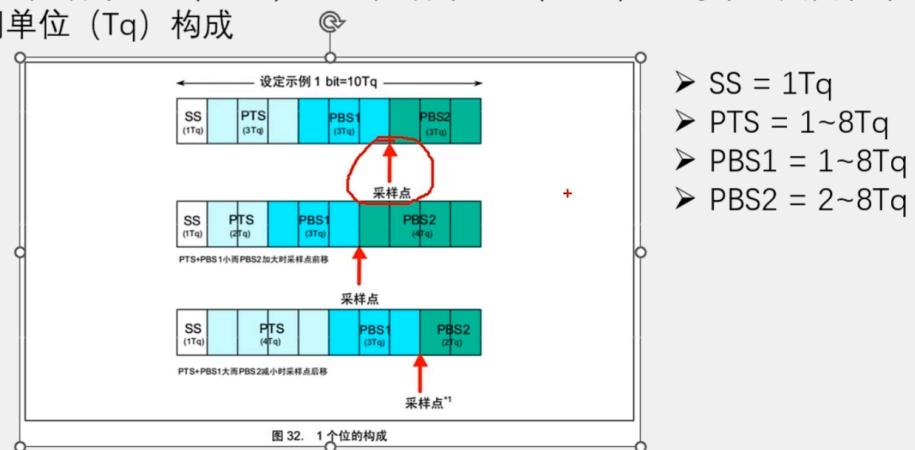


不能更改发送方的时钟,只能更改接收方的时钟

位时序

3 位时序

- 为了灵活调整每个采样点的位置，使采样点对齐数据位中心附近，CAN总线对每一个数据位的时长进行了更细的划分，分为同步段（SS）、传播时间段（PTS）、相位缓冲段1（PBS1）和相位缓冲段2（PBS2），每个段又由若干个最小时间单位（Tq）构成



PST:吸收网络上的物理延迟(发送单元的输出延迟+总线上信号的传播延迟+接收单元的输入延迟) $\times 2$

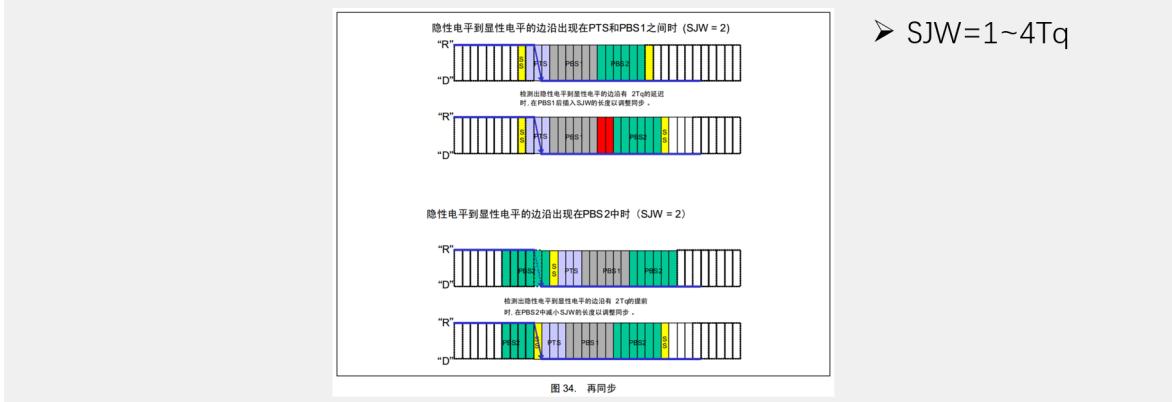
3 硬同步

- 每个设备都有一个位时序计时周期，当某个设备（发送方）率先发送报文，其他所有设备（接收方）收到SOF的下降沿时，接收方会将自己的位时序计时周期拨到SS段的位置，与发送方的位时序计时周期保持同步
- 硬同步只在帧的第一个下降沿（SOF下降沿）有效
- 经过硬同步后，若发送方和接收方的时钟没有误差，则后续所有数据位的采样点必然都会对齐数据位中心附近



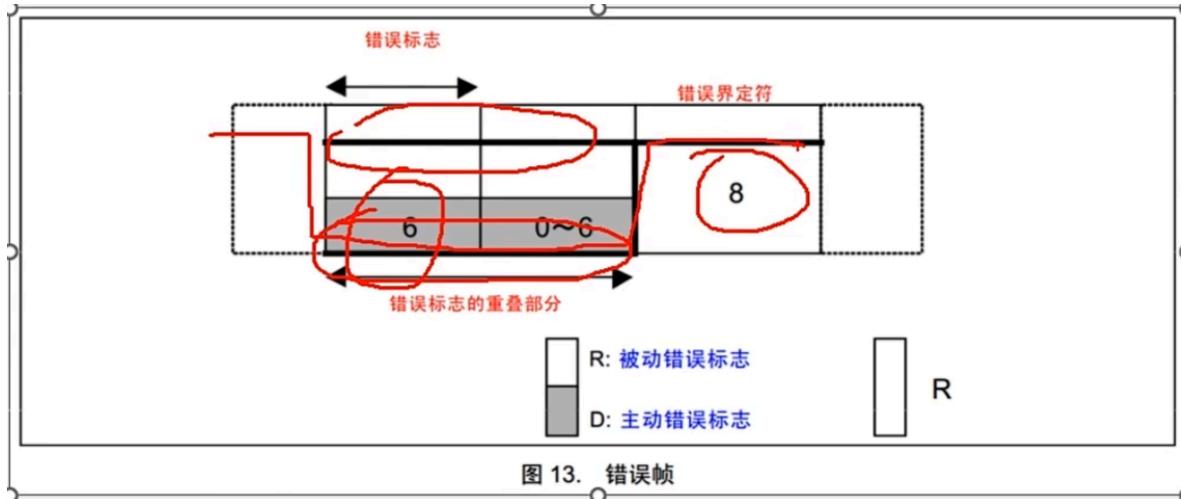
3 再同步

- 若发送方或接收方的时钟有误差，随着误差积累，数据位边沿逐渐偏离SS段，则此时接收方根据再同步补偿宽度值（SJW）通过加长PBS1段，或缩短PBS2段，以调整同步
- 再同步可以发生在第一个下降沿之后的每个数据位跳变边沿



- SJW是补偿的最大值,出现误差,补偿值是由误差值和SJW共同决定的
误差<=SJW,只补偿误差值,误差>SJW则补偿SJW
- 如果我发出1,自己读到了0,那么肯定有其他人在控制设备,那我就退出

错误帧

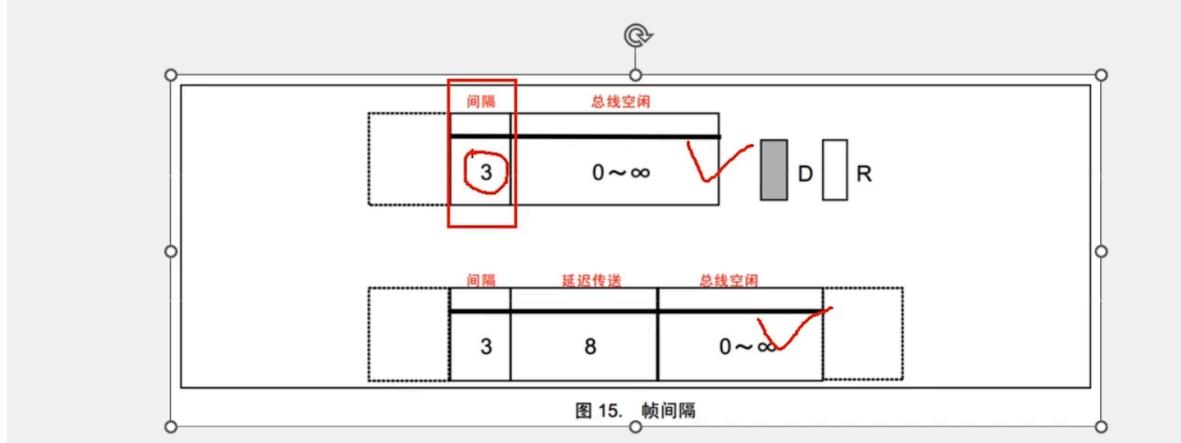


主动错误:6个显性0

被动错误:6个隐性1

帧间隔

- 将数据帧和遥控帧与前面的帧分离开

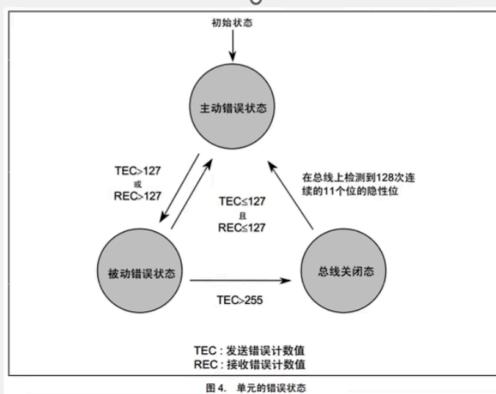


上面是主动错误状态,下面是被动错误状态

错误状态

5 错误状态

- 主动错误状态的设备正常参与通信并在检测到错误时发出主动错误帧
- 被动错误状态的设备正常参与通信但检测到错误时只能发出被动错误帧
- 总线关闭状态的设备不能参与通信
- 每个设备内部管理一个TEC和REC，根据TEC和REC的值确定自己的状态



TEC:发送错误计数器,设备发送时候每发现一个错误就增加一次.每进行一次正常的发送后,TEC也会减小一次

REC:接收错误计数器

在总线上检测128次连续的11个位的隐形位后TEC和REC都会清零

处于被动错误状态的接收方(因为REC>127),只要进行一次正确接收就可以立马回到主动错误状态

错误计数器

表 2. 错误计数值的变动条件

	接受和发送错误计数值的变动条件	发送错误计数值 (TEC)	接收错误计数值 (REC)
1	接收单元检测出错误时。 例外：接收单元在发送错误标志或过载标志中检测出“位错误”时，接收错误计数值不增加。	—	+1
2	接收单元在发送完错误标志后检测到的第一个位为显性电平时。	—	+8
3	发送单元在输出错误标志时。	+8	—
4	发送单元在发送主动错误标志或过载标志时，检测出位错误。	+8	—
5	接收单元在发送主动错误标志或过载标志时，检测出位错误。	—	+8
6	各单元从主动错误标志、过载标志的最开始检测出连续 14 个位的显性位时。 之后，每检测出连续的 8 个位的显性位时。	发送时 +8	接收时 +8
7	检测出在被动错误标志后追加的连续 8 个位的显性位时。	发送时 +8	接收时 +8
8	发送单元正常发送数据结束时(返回 ACK 且到帧结束也未检测出错误时)。	-1 TEC=0 时±0	—
9	接收单元正常接收数据结束时(到 CRC 未检测出错误且正常返回 ACK 时)。	—	1≤REC≤127 时-1 REC=0 时±0 REC>127 时 设 REC=127
10	处于总线关闭态的单元，检测到 128 次连续 11 个位的隐性位。	TEC=0	REC=0

错误类型

5 错误类型

- 错误共有5种：位错误、填充错误、CRC错误、格式错误、应答错误

表9. 错误的种类

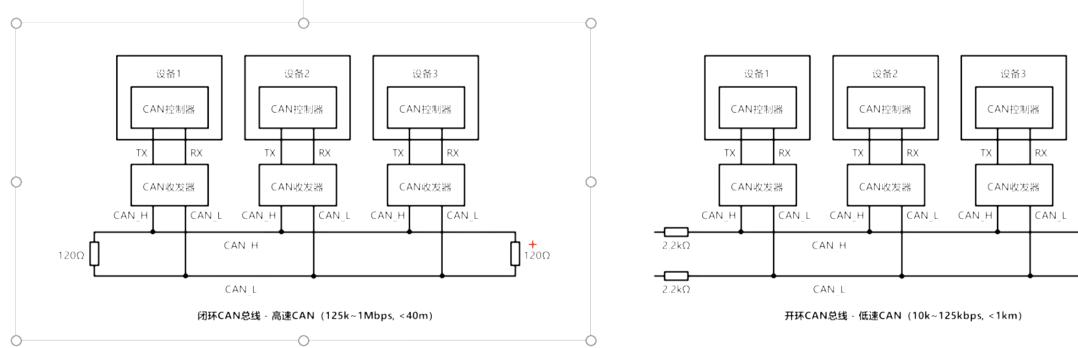
错误的种类	错误的内容	错误的检测帧（段）	检测单元
位错误	比较输出电平和总线电平（不含填充位），当两电平不一样时所检测到的错误。	• 数据帧（SOF~EOF） • 遥控帧（SOF~EOF） • 错误帧 • 过载帧	发送单元 接收单元
填充错误	在需要位填充的段内，连续检测到6位相同的电平时所检测到的错误。	• 数据帧（SOF~CRC顺序） • 遥控帧（SOF~CRC顺序）	发送单元 接收单元
CRC错误	从接收到的数据计算出的CRC结果与接收到的CRC顺序不同时所检测到的错误。	• 数据帧（CRC顺序） • 遥控帧（CRC顺序）	接收单元
格式错误	检测出与固定格式的位段相反的格式时所检测到的错误。	• 数据帧（ (CRC界定符、ACK界定符、 EOF) • 遥控帧 (CRC界定符、ACK界定符、 EOF) • 错误界定符 • 过载界定符	接收单元
ACK错误	发送单元在ACK槽(ACK Slot)中检测出隐性电平时所检测到的错误（ACK没被传送过来时所检测到的错误）。	• 数据帧（ACK槽） • 遥控帧（ACK槽）	发送单元

一个主动错误状态的设备和被动错误状态的设备都想发数据,检测到了11个隐形位,主动错误直接取得控制权

can硬件电路

1 CAN硬件电路

- 每个设备通过CAN收发器挂载在CAN总线上
- CAN控制器引出的TX和RX与CAN收发器相连, CAN收发器引出的CAN_H和CAN_L分别与总线的CAN_H和CAN_L相连
- 高速CAN使用闭环网络, CAN_H和CAN_L两端添加120Ω的终端电阻
- 低速CAN使用开环网络, CAN_H和CAN_L其中一端添加2.2kΩ的终端电阻



第一个, 是防止回波反射 第二个就是收紧电压, 让其电压一致, 默认状态变成1, 需要变成0的时候再去拉开

58 / 69

波特率计算

- 波特率 = $1 / (\text{一个数据位的时长} = 1 / (\text{T}_{\text{SS}} + \text{T}_{\text{PTS}} + \text{T}_{\text{PBS1}} + \text{T}_{\text{PBS2}}))$

多设备选择

1. 先到先得

- 若当前已经有设备正在操作总线发送数据帧/遥控帧，则其他任何设备不能再同时发送数据帧/遥控帧（可以发送错误帧/过载帧破坏当前数据）
- 任何设备检测到连续11个隐性电平，即认为总线空闲，只有在总线空闲时，设备才能发送数据帧/遥控帧
- 一旦有设备正在发送数据帧/遥控帧，总线就会变为活跃状态，必然不会出现连续11个隐性电平，其他设备自然也不会破坏当前发送
- 若总线活跃状态其他设备有发送需求，则需要等待总线变为空闲，才能执行发送需求

2. 非破坏性仲裁

- 若多个设备的发送需求同时到来或因等待而同时到来，则CAN总线协议会根据ID号（仲裁段）进行非破坏性仲裁，ID号小的（优先级高）取到总线控制权，ID号大的（优先级低）仲裁失利后将转入接收状态，等待下一次总线空闲时再尝试发送
- 实现非破坏性仲裁需要两个要求：
 - 线与特性：总线上任何一个设备发送显性电平0时，总线就会呈现显性电平0状态，只有当所有设备都发送隐性电平1时，总线才呈现隐性电平1状态，即： $0 \& X \& X = 0, 1 \& 1 \& 1 = 1$
 - 回读机制：每个设备发出一个数据位后，都会读回总线当前的电平状态，以确认自己发出的电平是否被真实地发送出去了，根据线与特性，发出0读回必然是0，发出1读回不一定是1

stm32启动流程

1. 选择启动模式：

自举模式选择引脚		自举模式	自举空间
BOOT1	BOOT0		
x	0	主 Flash	选择主 Flash 作为自举空间
0	1	系统存储器	选择系统存储器作为自举空间
1	1	嵌入式 SRAM	选择嵌入式 SRAM 作为自举空间

BooT0是专用引脚， BOOT1是和GPIO引脚共用的，一旦完成对BOOT1的采样响应的GPIO引脚立马进入空闲状态。

1.从主 Flash 启动， 主 Flash 一般是内置 Flash， Flash 地址被重映射到 0x00000000;JAG或者SWD模式下载调试程序选择这个

2.从系统存储器启动， 系统存储器一般指的是内置ROM， 跟上面一样映射。ROM里面有一段出厂预置的代码， 这段代码起到一个桥的作用， 允许外部通过UART/CAN或USB等将代码写入STM32的内置Flash中。这段代码也叫ISP代码

3.从SRAM里面启动， 内置SRAM启动。速度快，适合调试。但是掉电丢失

2.启动后bootloader做了什么？

从0x00000000开始执行代码，该处存放的代码就是bootloader.

1.硬件设置 SP、 PC，进入复位中断函数 Reset_Handler():

- 从 0x0800 0000 读取数据赋值给栈指针 SP(MSP),这个地址保存的是 __initial_sp的地址，并设置为栈顶指针 0x2000 0000 + RAM_Size。这一步确定了栈的起始位置，为函数调用等操作提供了空间。
- 从 0x0800 0004 读取数据赋值给 PC (指向 Reset_Handler 中断服务函数)

3.stm32是小端模式

4.Reset_Handler函数

Reset_Handler函数里主要执行了 SystemInit 和 __main 这两个函数

SystemInit:库函数，主要是初始化系统时钟

__main：主要作用是初始化RW段和ZI段（通过调用__scatterload）， 初始化堆栈以及跳转到主程序的main函数（通过调用__rt_entry）。

5. 进入main函数

执行main函数中的while死循环，当中断到来时从中断向量表中找到对应的中断服务函数并执行。

介绍F411CEU6

STM32F411CEU6 是带 DSP 和 FPU 的高性能的 ARM® Cortex® -M4 32 位 MCU+FPU

程序存储器大小: 512 kB

ADC分辨率: 12 bit

最大时钟频率: 100 MHz

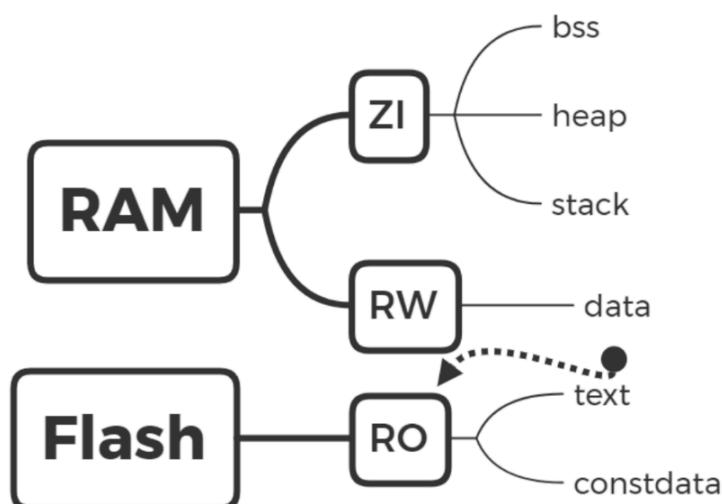
RAM: 128KB

一页: 2K

STM32 的内存管理

STM32 的内存管理起始就是对0X0800 0000 开始的 Flash 部分 和0x2000 0000 开始的 SRAM 部分使用管理。

这是编译之前的内存分布，只有代码段和不变量在Flash里面



编译之后：

Flash: text, 只读数据,

ZI-data(ZeroInitialize)

未初始化的全局变量和静态变量，以及初始化为0的变量；.BSS段ZI的数据全部是0，没必要开始就包含，只要程序运行之前将ZI数据所在的区域（RAM里面）一律清0，不占用Flash，运行时候占用RAM.heap和stack其实也属于ZI，只不过他不是程序编译就能确定大小的，必须在运行中才会有大小，而是是变化的

因为RAM掉电丢失，所以 RW-data 数据也得下载到ROM (flash) 中，在运行的时候复制到 RAM中运行

STM32FLASH

STM32F1系列的FLASH包含程序存储器、系统存储器和选项字节.

类型	起始地址	存储器	用途
ROM	0x0800 0000	程序存储器Flash	存储C语言编译后的程序代码
	0x1FFF F000	系统存储器	存储BootLoader, 用于串口下载
	0x1FFF F800	选项字节	存储一些独立于程序代码的配置参数
RAM	0x2000 0000	运行内存SRAM	存储运行过程中的临时变量
	0x4000 0000	外设寄存器	存储各个外设的配置参数
	0xE000 0000	内核外设寄存器	存储内核各个外设的配置参数

- 通过闪存存储器接口(外设)可以对程序存储器和选项字节进行擦除和编程

表2 闪存模块组织(中容量产品)

块	名称	地址范围	长度(字节)
主存储器	页0	0x0800 0000 – 0x0800 03FF	1K
	页1	0x0800 0400 – 0x0800 07FF	1K
	页2	0x0800 0800 – 0x0800 0BFF	1K
	页3	0x0800 0C00 – 0x0800 0FFF	1K
	页4	0x0800 1000 – 0x0800 13FF	1K
	.	.	.
	.	.	.
	页127	0x0801 FC00 – 0x0801 FFFF	1K
	启动程序代码	0x1FFF F000 – 0x1FFF F7FF	2K
	用户选择字节	0x1FFF F800 – 0x1FFF F80F	16
闪存存储器接口寄存器	FLASH_ACR	0x4002 2000 – 0x4002 2003	4
	FLASH_KEYR	0x4002 2004 – 0x4002 2007	4
	FLASH_OPTKEYR	0x4002 2008 – 0x4002 200B	4
	FLASH_SR	0x4002 200C – 0x4002 200F	4
	FLASH_CR	0x4002 2010 – 0x4002 2013	4
	FLASH_AR	0x4002 2014 – 0x4002 2017	4
	保留	0x4002 2018 – 0x4002 201B	4
	FLASH_OBR	0x4002 201C – 0x4002 201F	4
	FLASH_WRPR	0x4002 2020 – 0x4002 2023	4

闪存容量只包括主存储器



FLASH解锁

- FPEC共有三个键值：
RDPRT键 = 0x000000A5
KEY1 = 0x45670123
KEY2 = 0xCDEF89AB
- 解锁：
复位后，FPEC被保护，不能写入FLASH_CR
在FLASH_KEYR先写入KEY1，再写入KEY2，解锁
错误的操作序列会在下次复位前锁死FPEC和FLASH_CR
- 加锁：
设置FLASH_CR中的LOCK位锁住FPEC和FLASH_CR

闪存的读取不需要解锁

闪存全擦除过程：

1. 读取lock位置，直接进行解锁

2. 置FLASH_CR的MER = 1;
置FLASH_CR的STRT = 1

3. 循环判断 ~~FLASH_SR的BSY位=1?~~，等于0的适合就代表擦除完成

闪存页擦除过程1

1. 读取lock，进行解锁
2. 置FLASH_CR的PER = 1;
在FLASH_AR中选择要擦除的页
置FLASH_CR的STRT = 1

3. 读取bsy

闪存写入流程

写入之前会检测有没有擦除，除非写入的全是0

1. 读取lock, 解锁
2. 置FLASH_CR的PG位=1

3. 在指定的地址写入半字(16位)

任意读写：把闪存的一页读到SRAM里面，读写完成后再擦除一页

新知识：

校验和

1. 简单累加和
2. 取反累加和：
3. 异或校验：

循环冗余校验 (CRC)

cortexM3和cortexM4处理任务的区别

arm架构函数调用参数的传递（寄存器）：R0-R3，多于4个压栈。返回值是R0

Linux和FreeRTOS区别

flash 和 eeprom 区别和关系、nor flash和nand flash区别

EEPROM容量小、速度慢、K字节级别，按bit操作，可读可写，一般保存当前工作状态，多用数据存储。速度慢

FLASH容量大、速度快、M字节级别，按块擦除，页，字节读取，可读可写（多用于读）程序都存在这里。

利用IIC读取结构体

```
1 void M24LCxx_ReadBytes(uint16_t addr, uint8_t *readBuf, uint16_t readNum)
2 {
3     uint16_t i;
4     SoftI2C_Start();
5     SoftI2C_SendByte(M24LCxx_WAddr);
6     while(SoftI2C_ReceiveAck());
7     SoftI2C_SendByte(addr >> 8);
8     while(SoftI2C_ReceiveAck());
9     SoftI2C_SendByte(addr);
10    while(SoftI2C_ReceiveAck());
11    SoftI2C_Start();
12    SoftI2C_SendByte(M24LCxx_RAddr);
13    while(SoftI2C_ReceiveAck());
14    for(i = 0;i < readNum - 1;i++)
15    {
16        readBuf[i] = SoftI2C_ReceiveByte();
17        SoftI2C_SendAck(ACK);
18    }
19    readBuf[readNum - 1] = SoftI2C_ReceiveByte();
20    SoftI2C_SendAck(NAK);
21    SoftI2C_Stop();
22 }
23 typedef struct{
24     uint32_t OTA_flag;
25     uint32_t OTA_app_size;
26     uint32_t OTA_app_ver;
27
28     uint32_t reserved;
29
30     uint32_t App_number;
31     uint32_t App_index;
32     uint32_t App_versionLen[13];
33     uint32_t App_version[13];
```

```
34 }OTA_infoCtrlBlock;
35 OTA_infoCtrlBlockOTA_Info;
36 M24LCxx_ReadBytes(0,
    (uint8_t*)&OTA_Info,OTA_infoCtrlBlockSize);
```

CRC校验

1. 初始值，预算用的多项式

模2除法，对应C语言就是异或预算

按位运算，判断最高位是1还是0

1：异或多项式

0：不异或多项式

2.crc16初始值和多项式都是2字节，crc32是4个字节

3.服务端先发‘C’,500ms一次，收到第一个包的时候就不发c了，回一个ACK

DMA缓存一致性问题：

如何解决缓存一致性问题？

1.手动维护缓存一致性，确保缓存和物理内存的数据同步。常用方法如下：

- 在启动 DMA 传输前，强制将 CPU 缓存中的数据刷新到物理内存，确保 DMA 读取的是最新数据。
- 在 DMA 传输完成后，标记 CPU 缓存中的旧数据为无效，强制 CPU 下次读取时从物理内存获取最新数据。
- 通过 MPU（内存保护单元）或链接脚本，将 DMA 缓冲区配置为非缓存内存，避免缓存一致性问题。

实现按键长按

CPU利用率

一段时间内多少次回到空闲任务

按键抖动

Main函数和中断中如果都用到了同一个函数，有什么值得注意的，如果都用到一个全局变量

在嵌入式系统编程中，Main 函数和中断服务例程（ISR）中调用同一个函数或使用同一个全局变量时，需要注意以下问题：

调用同一个函数时的注意事项

函数的可重入性：

如果 Main 函数和中断服务例程都调用同一个函数，该函数必须是可重入的。

可重入函数是指可以被中断的函数，即在函数执行过程中被中断，然后在中断服务例程中再次调用该函数，不会导致数据错误。

为了保证函数的可重入性，函数内部不应使用静态局部变量或全局变量，或者必须对这些变量进行互斥保护。

函数调用的上下文：

中断服务例程的执行上下文与 Main 函数不同。中断服务例程有自己独立的栈空间，而 Main 函数使用的是主程序的栈。

如果函数内部使用了局部变量，这些局部变量在中断服务例程和 Main 函数中是独立的，不会相互影响。

使用同一个全局变量时的注意事项

数据一致性问题：

中断服务例程可以在任何时候中断 Main 函数的执行，因此全局变量的访问需要确保数据一致性。如果 Main 函数和中断服务例程都对同一个全局变量进行读写操作，可能会导致数据错误。

为了避免数据一致性问题，可以在访问全局变量时进入临界区，即关闭中断，完成操作后再开启中断。

使用 volatile 关键字：

如果全局变量在中断服务例程中被修改，需要在变量定义时加上 volatile 关键字，以防止编译器对变量的访问进行优化，确保每次访问都是最新的值。

减少全局变量的使用：

尽量减少全局变量的使用，可以通过函数参数传递数据，或者使用局部静态变量来替代全局变量。

模块化设计：

如果全局变量必须被多个模块共享，可以通过模块接口函数来访问和修改全局变量，而不是直接操作全局变量。

总之，在 Main 函数和中断服务例程中调用同一个函数或使用同一个全局变量时，需要特别注意函数的可重入性、数据一致性以及全局变量的访问保护，以确保程序的稳定性和可靠性。