STM32学习笔记

GPIO配置步骤

步骤：

1. 第一步，使用RCC开启GPIO的时钟
2. 第二步，使用GPIO_Init()函数初始化GPIO
3. 第三步，使用输出或者输入的函数控制GPIO口

常用的RCC开启始终函数

```
void RCC_AHBPeriphClockCmd(uint32_t RCC_AHBPeriph,FunctionalState
NewState);
void RCC_APB2PeriphClockCmd(uint32_t RCC_APB2Periph,FunctionalState
NewState);
void RCC_APB1PeriphClockCmd(uint32_t RCC_APB1Periph,FunctionalState
NewState);
```

参数1：选择外设，参数2：使能或者失能

常用的GPIO函数

复位GPIO外设

```
void GPIO_DeInit(GPIO_TypeDef* GPIOx);
```

复位AFIO外设

```
void GPIO_AFIODeInit(void);
```

初始化GPIO口

用结构体的参数来初始化GPIO口，先定义一个结构体变量，然后把再给结构体赋值，最后调用此函数，函数内部会自动读取结构体的值，然后自动把外设的各个参数配置好

```
void GPIO_Init(GPIO_TypeDef* GPIOx,GPIO_InitTypedef*
GPIO_InitStruct);
```

给GPIO结构体变量赋一个默认值

```
void GPIO_StructInit(GPIO_InitTypedef* GPIO_InitTypedef);
```

GPIO的输出函数

把制定的端口设置为高电平:

```
void GPIO_SetBits(GPIO_InitTypedef* GPIOx,uint16_t GPIO_Pin);
```

把指定的端口设置为低电平

```
void GPIO_ResetBits(GPIO_InitTypedef* GPIOx,uint16_t GPIO_Pin);
```

对根据第三个参数的值来设置电平

```
void GPIO_WriteBit(GPIO_InitTypedef* GPIOx,uint16_t
GPIO_Pin,BitAction BitVal);
```

对GPIOx 16个端口同时进行写入操作:

```
void GPIO_Write(GPIO_InitTypedef* GPIOx,uint16_t PortVal);
```

在推挽输出模式下，高低电平都具有驱动能力，开漏输出模式的高电平是没有驱动能力的，开漏输出模式的低电平具有驱动能力

#define的新名字在左边，并且可以给任何变量换名字，而typedef只能给变量换名字，新名字在右边

GPIO的输入函数

读取输入数据寄存器某个端口的输入值，返回值是高低电平

```
uint8_t GPIO_ReadInputDataBit(GPIO_InitTypedef* GPIOx,uint16_t
GPIO_Pin);
```

读取GPIO的每一位的值，返回值是16位的数据,每一位代表一个端口值

```
uint16_t GPIO_ReadInputData(GPIO_InitTypedef* GPIOx);
```

读取输出数据寄存器的某一位

```
uint8_t GPIO_ReadOutputDataBit(GPIO_InitTypedef* GPIOx,uint16_t
GPIO_Pin);
```

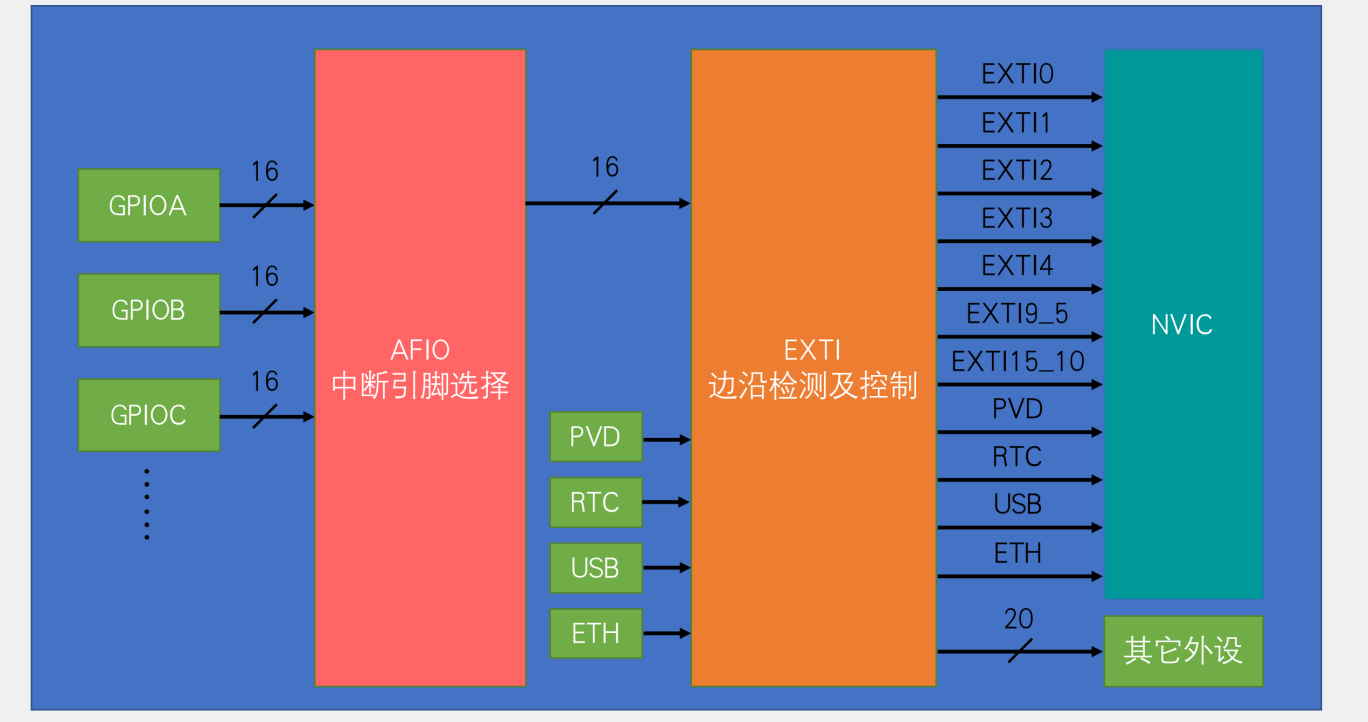
读取整个输出寄存器

```
uint16_t GPIO_ReadOutputData(GPIO_InitTypedef* GPIOx);
```

中断函数

- 中断：在主程序运行过程中，出现了特定的中断触发条件（中断源），使得CPU暂停当前正在运行的程序，转而去处理中断程序，处理完成后又返回原来被暂停的位置继续运行

- 中断优先级：当有多个中断源同时申请中断时，CPU会根据中断源的轻重缓急进行裁决，优先响应更加紧急的中断源
- 中断嵌套：当一个中断程序正在运行时，又有新的更高优先级的中断源申请中断，CPU再次暂停当前的中断程序，转而去处理新的中断程序，处理完后依次进行返回
- NVIC：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进行中断嵌套，响应优先级高的可以进行优先排队，抢占优先级和响应优先级均相同的按中断号排队
- EXTI：（Extern Interrupt）外部中断
- EXTI可以检测指定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

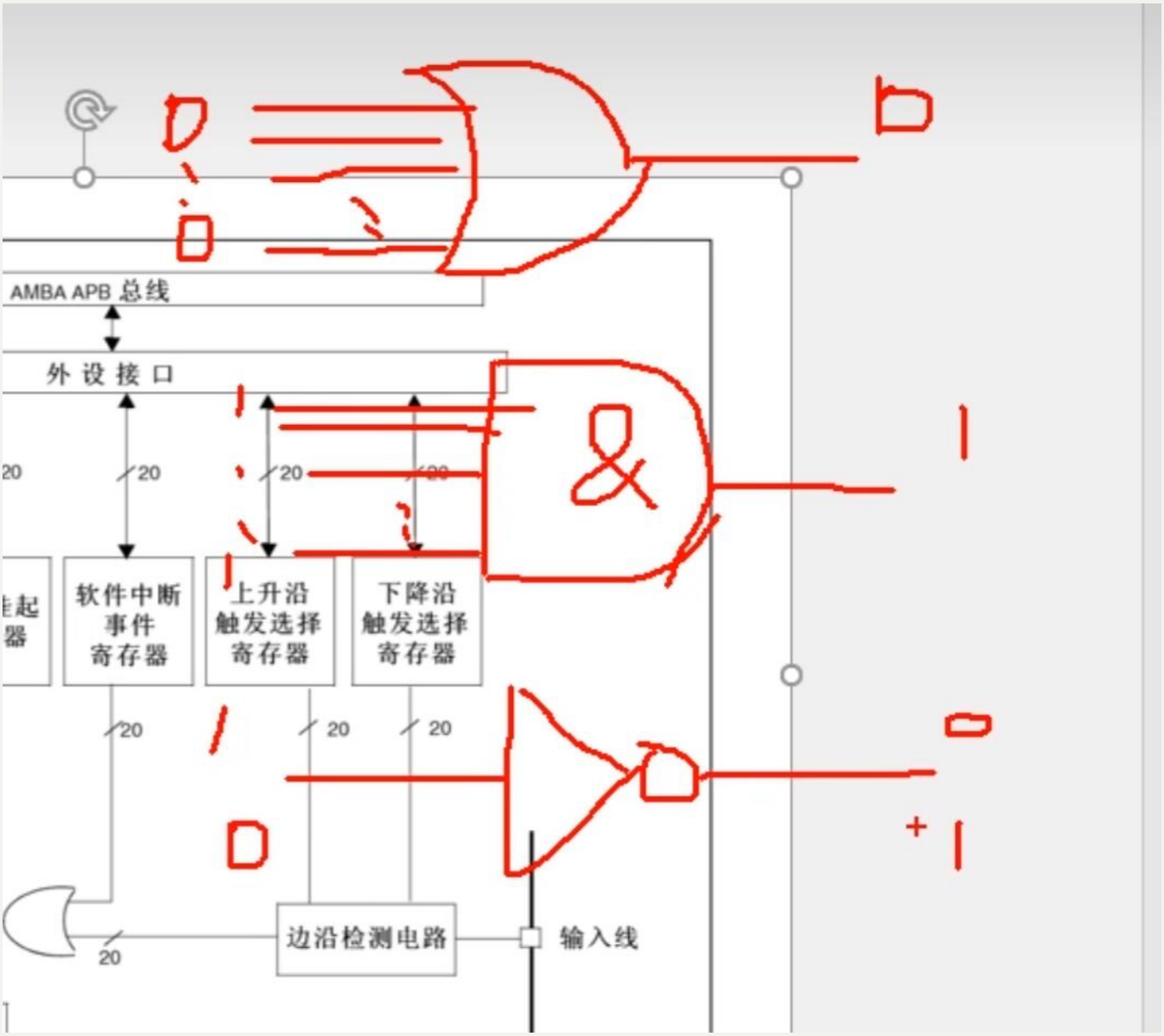


AFIO选择中断引脚，外部中断的9-5,15-10会触发同一个中断函数，再根据标志位来区分到底是哪个中断进来的

配置数据选择器，只有一个Pin接到EXTI

在STM32中AFIO主要完成两个任务：复用功能引脚重映射、中断引脚选择

或、与、非门



EXTI配置步骤

1. 第一步，配置RCC，把设计到的外设时钟都打开
2. 第二步，配置GPIO，选择端口为输入模式
3. 第三步，配置AFIO，选择使用的一路GPIO，连接到后面的EXTI
4. 第四步，配置EXTI，选择边沿触发方式，选择触发响应方式
5. 第五步，配置NVIC，给中断选择一个合适的优先级

EXTI和NVIC时钟默认是打开的，NVIC是内核的外设，内核的外设都不需要开启时钟，RCC管的都是内核外的外设

复位AFIO外设

```
void GPIO_AFIODeInit(void);
```

锁定GPIO配置

锁定引脚的配置，防止意外更改

```
void GPIO_PinLockConfig(GPIO_TypeDef* GPIOx,uint16_t GPIO_Pin);
```

配置AFIO的事件输出功能

```
void GPIO_EventOutputConfig(uint8_t GPIO_PortSource,uint8_t
GPIO_PinSource);
void GPIO_EventOutputCmd(FunctionalState NewState);
```

配置引脚重映射

```
void GPIO_PinRemapConfig(uint32_t GPIO_Remap,FunctionalState
NewState);
```

配置AFIO的数据选择器

选择想使用的中断引脚

```
void GPIO_EXTILineConfig(uint8_t GPIO_PortSource,uint8_t
GPIO_PinSource);
```

清除EXTI的配置

恢复上电默认的状态

```
void EXTI_DeInit(void);
```

根据结构体配置EXTI外设

```
void EXTI_Init(EXTI_InitTypedef* EXTI_InitStruct);
```

给传入的结构体参数赋一个默认值

```
void EXTI_StructInit(EXTI_InitTypedef* EXTI_InitStruct);
```

软件触发外部中断

参数给一个中断线，就能软件触发一次这个外部中断

```
void EXTI_GenerateSWInterrupt(uint32_t EXTI_Line);
```

在外设运行的时候会产生一些状态标志位，例如：外部中断来了，挂起寄存器会置一个标志位，标志位放在状态寄存器，

当程序想看这些标志位

获取指定的标志位

```
FlagStatus EXTI_GetFlagStatus(uint32_t EXTI_Line);
```

对置1的标志位进行清除

```
void EXTI_ClearFlag(uint32_t EXTI_Line);
```

在中断函数中获取标志位

```
ITStatus EXTI_GetITStatus(uint32_t EXTI_Line);
```

清除中断挂起标志位

```
void EXTI_ClearITPendingBit(uint32_t EXTI_Line);
```

中断分组

```
void NVIC_PriorityGroupConfig(uint32_t NVIC_PriorityGroup);
```

根据结构体里面的参数初始化NVIC

```
void NVIC_Init(NVIC_InitTypedef* NVIC_InitStruct);
```

设置中断向量表

```
NVIC_SetVectorTable(uint8_t NVIC_VectTab,uint32_t Offset);
```

系统低功耗配置

```
void NVIC_SystemLPConfig(uint8_t LowPowerMode,FunctionalState
NewState)
```

中断函数要简短快速，不要在中断中执行Delay

定时器

- TIM（Timer）定时器
- 定时器可以对输入的时钟进行计数，并在计数值达到设定值时触发中断
- 16位计数器、预分频、自动重装寄存器的时基单元，在72M计数时钟下可以实现最大59.65s的定时
- 不仅具备基本的定时器中断功能，而且还包含内外时钟源选择、输入捕获、输出比较、编码器接口、主从触发模式等多种功能
- 根据复杂度和应用场景分为了高级定时器、通用定时器、基本定时器三种类型
- 对72MHz计72个数就是1MHz，也就是1us的时间，计72000个数，那就是1KHz也就是1ms的时间
- 59.65s = 1/72M/65536/65536(中断频率取倒数)，
- STM32的定时器支持级联的模式：一个定时器的输出当做另一个定时器的输入最大定时时间就是59.65s X 65536 X 65536

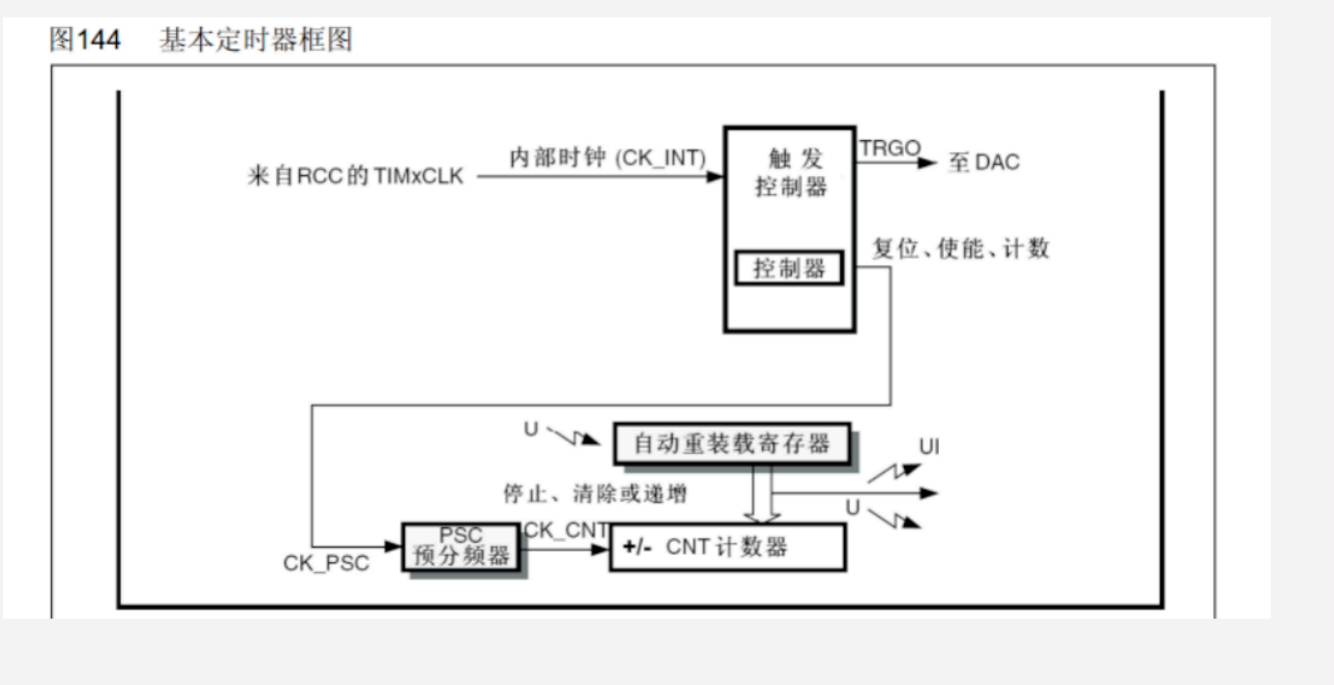
定时器类型

类型	编号	总线	功能
高级定时器	TIM1、TIM8	APB2	拥有通用定时器全部功能，并额外具有重复计数器、死区生成、互补输出、刹车输入等功能
通用定时器	TIM2、TIM3、TIM4、TIM5	APB1	拥有基本定时器全部功能，并额外具有内外时钟源选择、输入捕获、输出比较、编码器接口、主从触发模式等功能
基本定时器	TIM6、TIM7	APB1	拥有定时中断、主模式触发DAC的功能

- STM32F103C8T6定时器资源：TIM1、TIM2、TIM3、TIM4

- 预分频器（PSC）：对输入的基准频率提前进行一个分频的操作
- 实际分频系数 = 预分频器的值 + 1，最大可以写65535即65536分频
- 计数器（CNT）：也是16位，值可以从0~65535，当计数器的值自增（自减）到目标值时，产生中断，完成定时
- 自动重装寄存器（）：也是16位当计数值等于自动重装值时，就是计时的时间到了，就会产生一个中断信号，并且清零计数器，计数器自动开始下一次的计数计时，计数值等于自动重装值的中断一般叫做“更新中断”，此更新中断就会通往NVIC，再配置好NVIC的定时器通道，定时器上的更新中断就会得到CPU的响应

了，对应的事件叫做“更新事件”，更新事件不会触发中断，但可以触发内部其他电路的工作

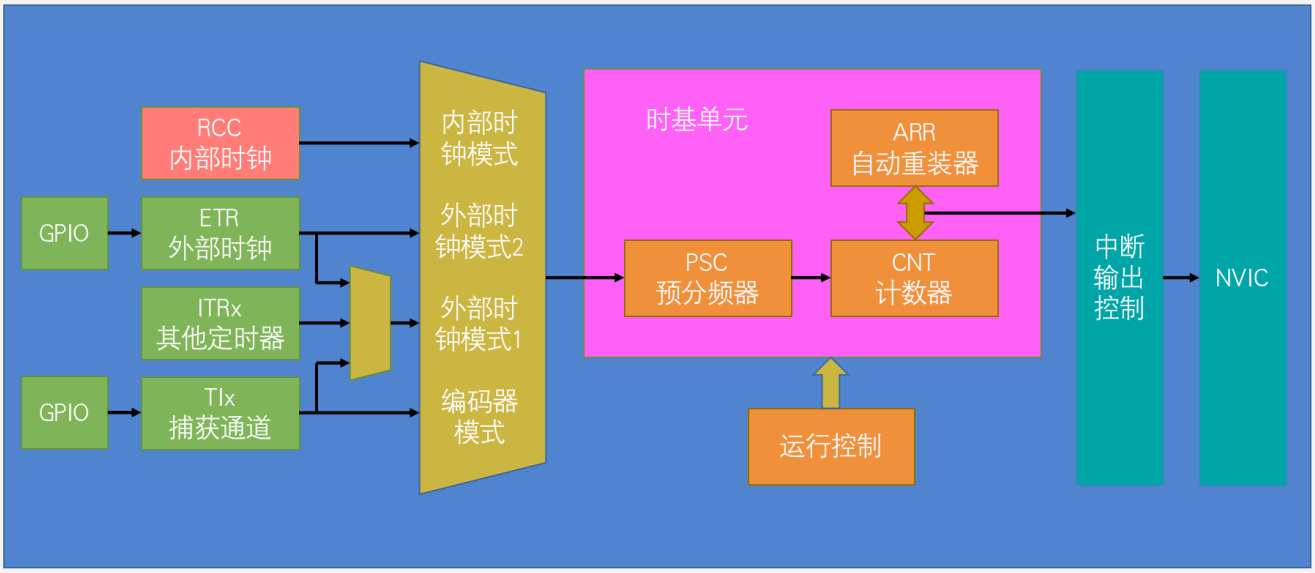


- 从基准时钟，到预分频器，再到计数器，计数器自增，同时不断地与自动重装寄存器进行比较，计数器和自动重装寄存器的值相等时，即计时时间到，这时会产生一个更新中断和更新事件，CPU响应更新中断，就完成了定时中断的任务了。

主从触发模式

使用定时器的主模式，可以把定时器的更新事件映射到触发输出TRGO（Trigger Out）的位置，TRGO直接接到DAC的触发转换引脚上，这样定时器的更新就不需要再通过中断来触发DAC转换了

定时中断基本结构



缓冲寄存器：某个时刻把预分频器由0改成了1，当技术计到一半的时候改变了分频值，这个变化不会立即生效，而是会等到本次计数周期结束时，产生了了更新事件，预分频器的值才会被传递到缓冲寄存器里面去，才会生效。

计数器计数频率：CK_CNT = CK_PSC / (PSC + 1)

计数器溢出频率：CK_CNT_OV = CK_CNT / (ARR + 1) = CK_PSC / (PSC + 1) / (ARR + 1)

开启定时器步骤

1. 第一步，RCC开启时钟
2. 第二步，选择时基单元的时钟源
3. 第三步，配置时基单元
4. 第四步，配置输出中断控制，允许更新中断输出到NVIC
5. 第五步，配置NVIC，在NVIC中打开定时器中断的通道，并分配一个优先级
6. 第六步，运行控制
7. 第七步，使能计数器

定时器常用的库函数

恢复缺省配置

```
void TIM_DeInit(TIM_TypeDef* TIMx);
```

时基单元初始化

```
void TIM_TimeBaseInit(TIM_TypeDef* TIMx, TIM_TimeBaseInitTypeDef*
TIM_TimeBaseInitStruct);
```

把结构体变量赋一个默认值

```
void TIM_TimeBaseStructInit(TIM_TimeBaseInitTypeDef*
TIM_TimeBaseInitStruct);
```

使能计数器

```
void TIM_Cmd(TIM_TypeDef* TIMx, FunctionalState NewState);
```

使能中断输出信号

```
void TIM_ITConfig(TIM_TypeDef* TIMx, uint16_t TIM_IT, FunctionalState
NewState);
```

选择内部时钟

```
void TIM_InternalClockConfig(TIM_TypeDef* TIMx);
```

选择ITRx其他定时器的时钟

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

选择TIx捕获通道的时钟

```
void TIM_TIxExternalClockConfig(TIM_TypeDef* TIMx, uint16_t
TIM_TIxExternalCLKSource,
                                uint16_t TIM_ICPolarity, uint16_t
ICFilter);
```

参数3：输入的极性 参数4：滤波器

选择ETR通过外部时钟模式1输入的时钟

```
void TIM_ETRClockMode1Config(TIM_TypeDef* TIMx, uint16_t
TIM_ExtTRGPrescaler, uint16_t TIM_ExtTRGPolarity, uint16_t
ExtTRGFilter);
```

参数2：预分频器 参数3：输入的极性 参数4：滤波器

选择ETR通过外部时钟模式2输入的时钟

```
void TIM_ETRClockMode2Config(TIM_TypeDef* TIMx, uint16_t
TIM_ExtTRGPrescaler, uint16_t TIM_ExtTRGPolarity, uint16_t
ExtTRGFilter);
```

单独配置ETR引脚的预分频器、极性、滤波器这些参数的

```
void TIM_ETRConfig(TIM_TypeDef* TIMx, uint16_t TIM_ExtTRGPrescaler,
uint16_t TIM_ExtTRGPolarity, uint16_t ExtTRGFilter);
```

单独写预分频值

```
void TIM_PrescalerConfig(TIM_TypeDef* TIMx, uint16_t Prescaler,
uint16_t TIM_PSCReloadMode);
```

参数3：写入的模式，在更新事件生效，或者在写入后，手动产生一个更新事件，让这个值立刻生效

改变计数器的计数模式

```
void TIM_CounterModeConfig(TIM_TypeDef* TIMx, uint16_t
TIM_CounterMode);
```

自动重装器预装功能配置

```
void TIM_ARRPreloadConfig(TIM_TypeDef* TIMx, FunctionalState
NewState);
```

给计数器写入一个值

```
void TIM_SetCounter(TIM_TypeDef* TIMx, uint16_t Counter);
```

给自动重装器写入一个值

```
void TIM_SetAutoreload(TIM_TypeDef* TIMx, uint16_t Autoreload);
```

获取当前计数器的值

```
uint16_t TIM_GetCounter(TIM_TypeDef* TIMx);
```

获取当前预分频器的值

```
uint16_t TIM_GetPrescaler(TIM_TypeDef* TIMx);
```

使用跨文件的变量： extern声明变量，告诉编译器，有Num这个变量在别的文件中定义了，在此文件中也可以使用

输出比较

- OC（Output Compare）输出比较
- 输出比较可以通过比较CNT和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个通道额外拥有死去生成和互补输出的功能

输出比较常用的函数

配置输出比较

```
void TIM_OC1Init(TIM_TypeDef* TIMx, TIM_OCInitTypeDef*
TIM_OCInitStruct);
void TIM_OC2Init(TIM_TypeDef* TIMx, TIM_OCInitTypeDef*
TIM_OCInitStruct);
void TIM_OC3Init(TIM_TypeDef* TIMx, TIM_OCInitTypeDef*
TIM_OCInitStruct);
void TIM_OC4Init(TIM_TypeDef* TIMx, TIM_OCInitTypeDef*
TIM_OCInitStruct);
```

给输出比较结构体赋一个默认值

```
void TIM_OCStructInit(TIM_OCInitTypeDef* TIM_OCInitStruct);
```

配置强制输出模式

在运行中想要暂停输出波形并且强制输出高或者低电平，强制输出高电平和设置百分百占空比一样，强制输出低电平和设置百分百低电平是一样的。

```
void TIM_ForcedOC1Config(TIM_TypeDef* TIMx, uint16_t
TIM_ForcedAction);
void TIM_ForcedOC2Config(TIM_TypeDef* TIMx, uint16_t
TIM_ForcedAction);
void TIM_ForcedOC3Config(TIM_TypeDef* TIMx, uint16_t
TIM_ForcedAction);
void TIM_ForcedOC4Config(TIM_TypeDef* TIMx, uint16_t
TIM_ForcedAction);
```

配置CCR寄存器的预装功能

预装功能就是影子寄存器：写入的值不会立即生效，而是在更新事件才会生效

```
void TIM_OC1PreloadConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCPreload);
void TIM_OC2PreloadConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCPreload);
void TIM_OC3PreloadConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCPreload);
void TIM_OC4PreloadConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCPreload);
```

配置快速使能

```
void TIM_OC1FastConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCFast);
void TIM_OC2FastConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCFast);
void TIM_OC3FastConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCFast);
void TIM_OC4FastConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCFast);
```

外部事件时清除REF信号

```
void TIM_ClearOC1Ref(TIM_TypeDef* TIMx, uint16_t TIM_OCClear);
void TIM_ClearOC2Ref(TIM_TypeDef* TIMx, uint16_t TIM_OCClear);
void TIM_ClearOC3Ref(TIM_TypeDef* TIMx, uint16_t TIM_OCClear);
void TIM_ClearOC4Ref(TIM_TypeDef* TIMx, uint16_t TIM_OCClear);
```

单独设置输出比较的极性

带N的是高级定时器里互补通道的配置

```
void TIM_OC1PolarityConfig(TIM_TypeDef* TIMx, uint16_t
TIM_OCPolarity);
void TIM_OC1NPolarityConfig(TIM_TypeDef* TIMx, uint16_t
TIM_OCNPolarity);
void TIM_OC2PolarityConfig(TIM_TypeDef* TIMx, uint16_t
TIM_OCPolarity);
void TIM_OC2NPolarityConfig(TIM_TypeDef* TIMx, uint16_t
TIM_OCNPolarity);
void TIM_OC3PolarityConfig(TIM_TypeDef* TIMx, uint16_t
TIM_OCPolarity);
void TIM_OC3NPolarityConfig(TIM_TypeDef* TIMx, uint16_t
TIM_OCNPolarity);
void TIM_OC4PolarityConfig(TIM_TypeDef* TIMx, uint16_t
TIM_OCPolarity);
```

单独修改输出使能参数

```
void TIM_CCxCmd(TIM_TypeDef* TIMx, uint16_t TIM_Channel, uint16_t
TIM_CCx);
void TIM_CCxNCmd(TIM_TypeDef* TIMx, uint16_t TIM_Channel, uint16_t
TIM_CCxN);
```

选择输出比较模式

```
void TIM_SelectOCxM(TIM_TypeDef* TIMx, uint16_t TIM_Channel, uint16_t
TIM_OCMode);
```

单独更改CCR寄存器的值的函数

```
void TIM_SetCompare1(TIM_TypeDef* TIMx, uint16_t Compare1);
void TIM_SetCompare2(TIM_TypeDef* TIMx, uint16_t Compare2);
void TIM_SetCompare3(TIM_TypeDef* TIMx, uint16_t Compare3);
void TIM_SetCompare4(TIM_TypeDef* TIMx, uint16_t Compare4);
```

使用高级定时器输出PWM时调用使能主输出函数

```
void TIM_CtrlPWMOutputs(TIM_TypeDef* TIMx, FunctionalState NewState);
```

定时器输出需要使用复用推挽输出，开启复用推挽输出引脚的控制权才能交给片上外设，PWM波形才能通过引脚输出

引脚重映射

开启AFIO时钟

```
void GPIO_PinRemapConfig(uint32_t GPIO_Remap, FunctionalState
NewState);
```

- 完全重映射：四个引脚全换
- 部分重映射：前面两个引脚变了或者后面两个引脚变了
- 调试端口不能做普通的GPIO口使用，需要解除复用

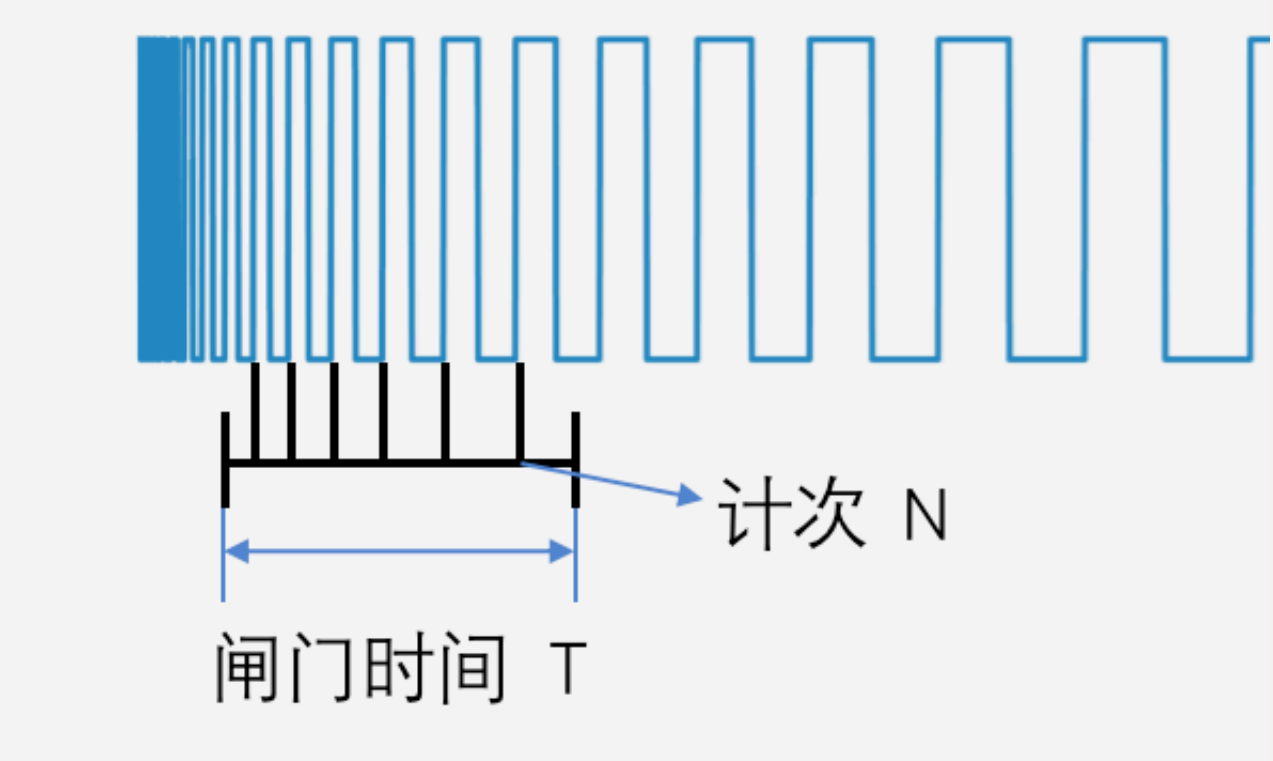
输入捕获

- 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

频率测量：

测频法：在闸门时间T内，对上升沿计次，得到N，则频率

$$f_x = N / T$$

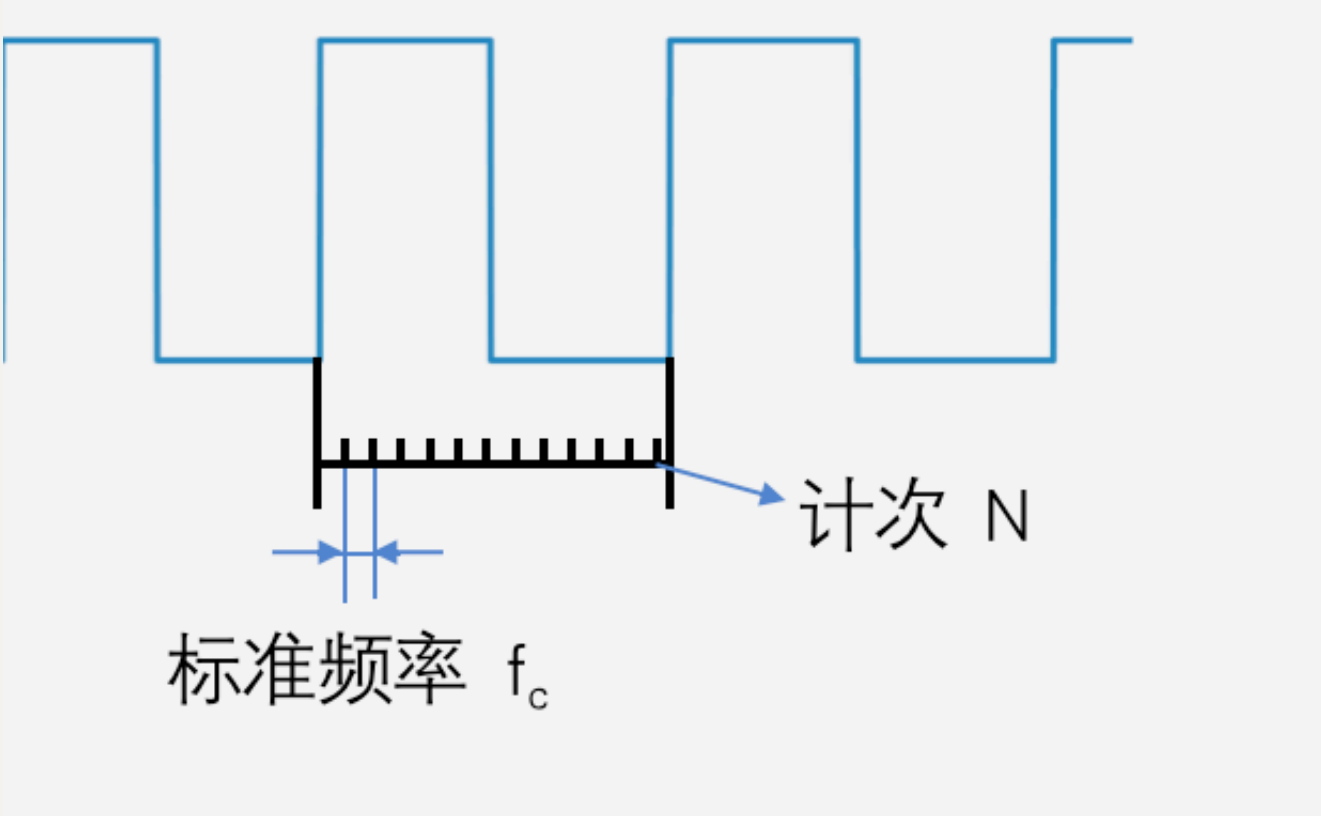


测频法：自定一个闸门时间T，通常设置为1s，在1s时间内，对信号上升沿计次，从0开始计，每来一个上升沿，计次+1，每来一个上升沿，其实就是来了一个周期的信号，在1s时间内，来个几个周期，频率就是多少Hz，（频率的定义：1s内出现了多少个重复的周期），这是一种直接按频率定义来测量的方法，闸门时间也可以是2s，计次值除2，就是频率

测频法测量的是一个闸门时间的多个周期自带一个均值滤波，如果在闸门时间内波形频率有变化，得到的其实是这一段时间的平均频率，测频法测量时间慢，测量结果是一段时间的平均值，值比较平滑

测周法：两个上升沿内，以标准频率计次，得到N，则频率

$$f_x = f_c / N$$



测周法：捕获信号的两个上升沿，测量之间持续的时间，使用一个已知的标准频率的计次时钟，来驱动计数器，从一个上升沿开始计，计数器从0开始，一直计到下一个上升沿，停止，计一个数的时间是1/fc，计N个数时间就是N/fc，N/fc就是周期，再取个倒数，就得到频率的公式，fx = fc/N

测周法只测量一个周期，就能出一次结果，出结果的速度取决于待测信号的频率，一般来说测周法结果更新更快，但是由于他只测量一个周期，所以结果值会受噪声的影响，波动比较大。

测频法适合测高频信号，测周法适合测量低频信号

例如：定了1s为闸门周期，结果1s内一个上升沿都没有，但不能认为频率是0，计次N很少时，误差会非常大，所以测频法适合测量高频率的信号，测周法适合低频信号，低频信号，周期比较长，计次就会比较多，有助于减少误差。如果待测频率太高，那么一个周期内只能计一两个数，如果待测信号再高一些，甚至一个数也计不到，不能认为频率无穷大

中界频率：测频法与测周法误差相等时的频率点（测频法和测周法的N相同）

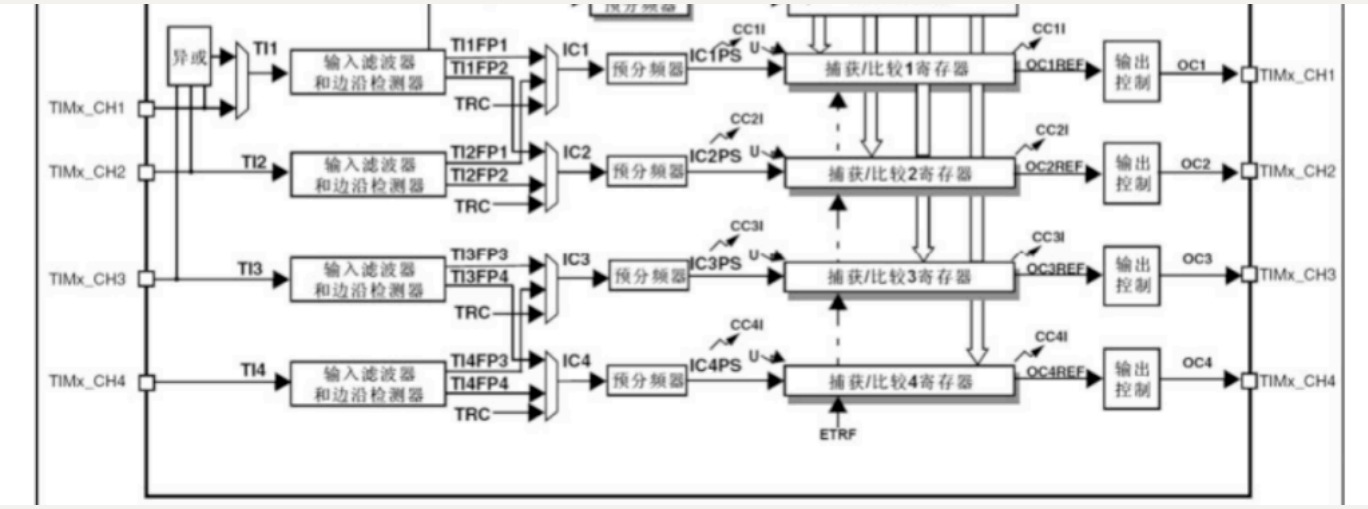
$$f_m = \sqrt{f_c / T}$$

计数次数越多，+-1误差对结果的影响越小

待测频率<中界频率，测周法合适

待测频率>中界频率，测频法合适

异或门：当输入引脚的任何一个引脚有电平翻转时，输出引脚就产生一次电平翻转



输入信号来到输入滤波器（对信号进行滤波，避免高频的毛刺信号误触发）和边沿检测器（可以选择高电平触发，或者低电平触发）

有两套滤波和边沿检测电路，第一套电路：经过滤波和极性选择得到TI1FP1，输入给通道1的后续电路，第二套电路：经过另一个滤波和极性选择得到TI1FP2，输入给下面通道2的后续电路，同理下面TI2的信号进来，也经过两套滤波和极性选择，得到TI2FP1和TI2FP2，其中TI2FP1输入给上面，TI2FP2输入给下面，两个输入信号进来可以选择各走各的，也可以选择进行交叉，让CH2引脚输入给通道1，或者CH1引脚输入给通道2，这样做的目的可以灵

活切换后续捕获电路的输入，通过数据选择器进行灵活选择，可以把一个引脚的输入，同时映射到两个捕获单元，这是不PWMI的经典结构，

例如，第一个捕获通道，使用上升沿触发，用来捕获周期，第二个通道，使用下降沿触发，用来捕获占空比，两个通道同时对一个引脚进行捕获，就可以同时测量频率和占空比，这就是PWMI模式。

TRC是为了无刷电机的驱动

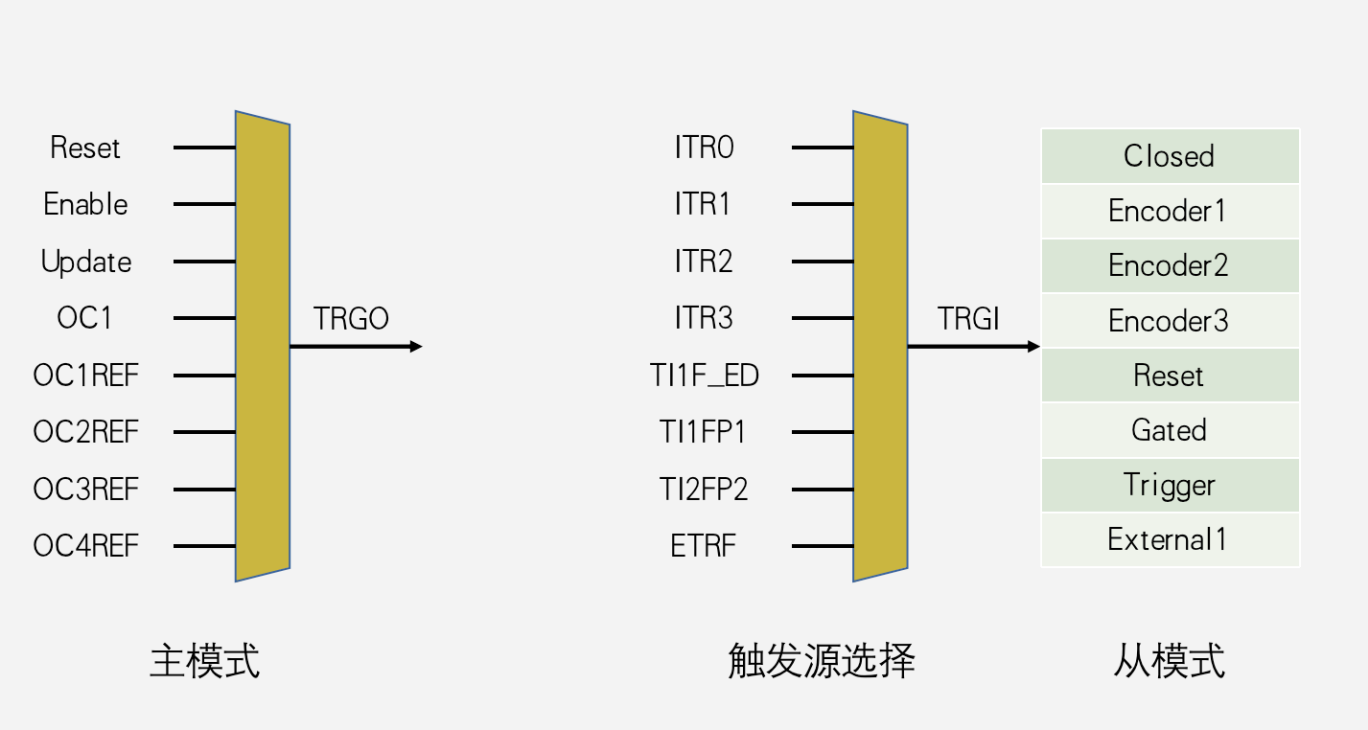
输入信号进行滤波和极性选择后，来到预分频器，预分频器，每个通道各有一个，可以选择对前面的信号进行分频，分频之后的触发信号就可以触发捕获电路进行工作了，每来一个触发信号，CNT的值就会向CCR转运一次，转运的同时，会发送一个捕获事件，这个事件会在状态寄存器置标志位，同时也可以产生中断，如果需要再捕获期间处理事情就可以开启这个捕获中断

例如：配置上升沿触发捕获，每来一个上升沿，CNT转运到CCR一次，因为CNT计数器是由内部的标准时钟驱动的，所以CNT的数值，可以用来记录两个上升沿之间的时间间隔，这个时间间隔就是周期，再取个倒数就是测周法测量的频率了，

每次捕获后要把CNT清0，下次再上升沿再捕获的时候取出的CNT才是两个上升沿的时间间隔，可以用主从触发模式，自动来完成。

数字滤波器：由一个事件计数器组成，记录到N个事件后会产生一个输出的跳变，简单来说滤波器的工作原理就是，以采样频率对输入信号进行采样，当连续N个值都为高电平，输出才为高电平，连续N个值都为低电平输出才为低电平，如果信号出现高频抖动，导致连续采样N个值不全都一样，那输出就不会变化，这样就可以达到滤波的效果，采样频率越低，采样个数N越大，滤波效果就越好。

主从触发模式：（主模式、从模式和触发源选择三个功能的简称）



主模式：将定时器内部的信号映射到TRGO引脚，用于触发别的外设。

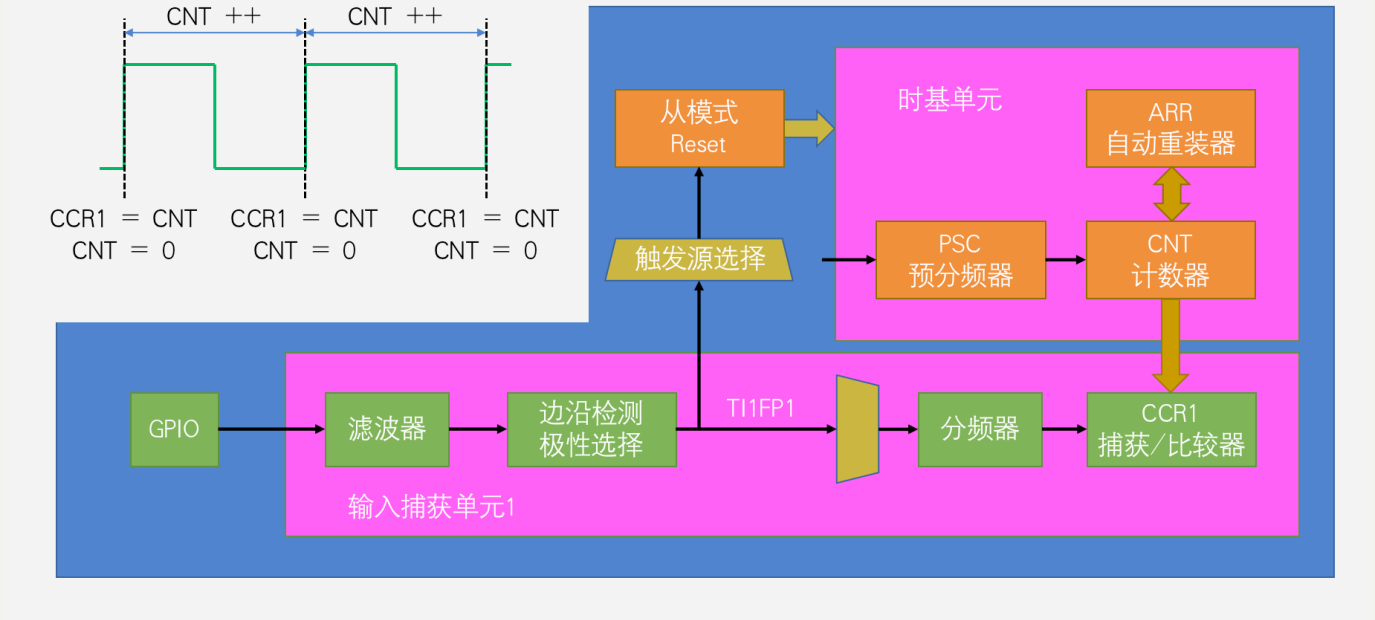
从模式：接收其他外设或者自身外设的一些信号，用于控制自身定时器的运行，也就是被别的信号控制。

触发源选择：选择从模式的触发信号源，也可以认为是从模式的一部分，触发源选择，选择一个指定的信号得到TRGI，TRGI去触发从模式，从模式可以在上述列表里，选择一项操作来自动执行。

例如：让TI1FP11信号自动触发CNT清零，触发源选择可以选择TI1FP1，从模式执行的操作，就可以选择执行Reset的操作，这样TI1FP1的信号就可以自动触发从模式，从模式自动清零CNT，实现硬件全自动测量

输入捕获基本结构：

输入捕获基本结构



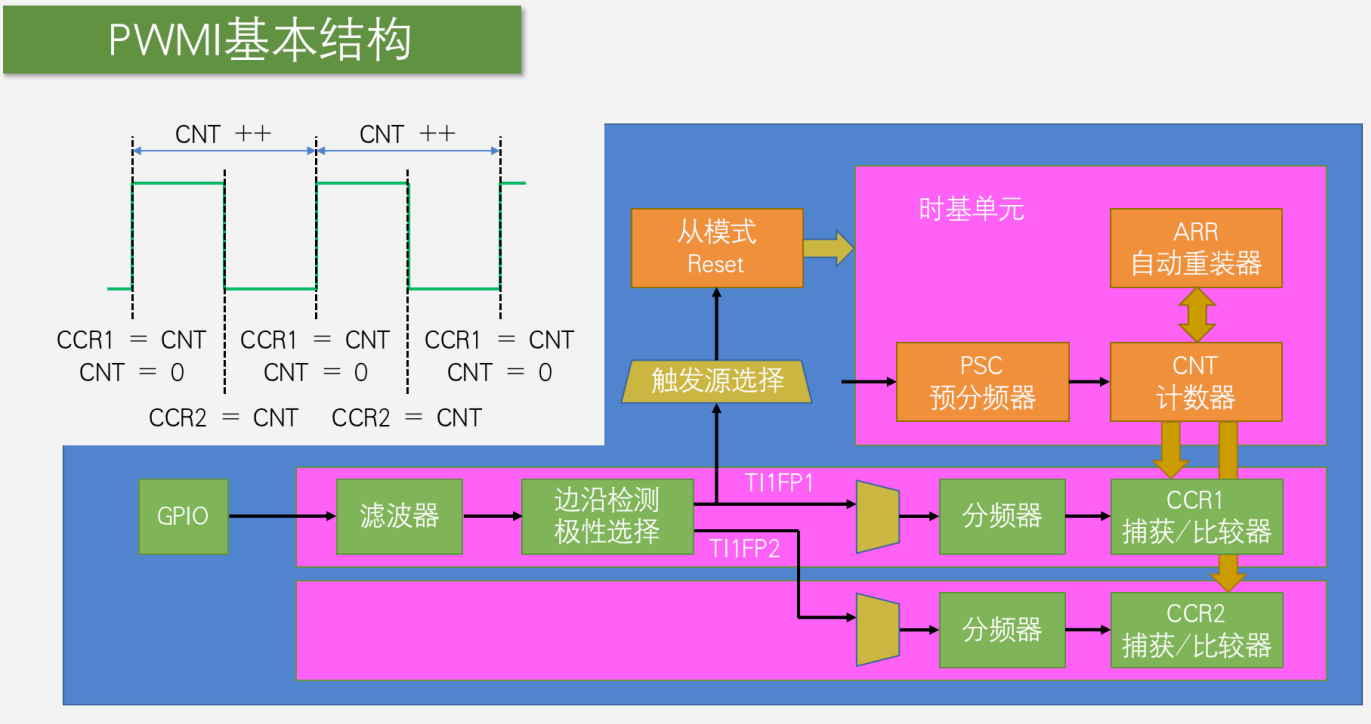
只使用了一个通道，目前只能测量频率，配置好时基单元，启动定时器，CNT就会在预分频之后的时钟驱动下，不断自增，这个CNT就是测周法用来计数计时的，经过预分频之后的时钟频率就是，驱动CNT的标准频率fc，(标准频率 = 72M/预分频系数)，下面输入捕获通道1的GPIO口，输入一个上面的方波信号，经过滤波器和边沿检测，选择TI1FP1为上升沿触发，之后输入选择直连的通道分频器选择不分频，当TI1FP1出现上升沿之后，CNT的当前计数值转运到CCR1里，同时触发源选择，选择TI1FP1选择为触发信号，选中TI1FP1为触发信号，从模式选择复位操作，TI1FP1的上升沿也同样会通过上面的触发源选择那一路，取触发CNT清零，注意是先转运CNT的值到CCR里去，再出发从模式给CNT清零或者是非阻塞的同时转移，CNT的值转移到CCR，同时0转移到CNT里面去，不能是先清零CNT，再捕获，否则捕获值都是0了。

例如：左上角图，信号产生一个上升沿，CCR1 = CNT，就是把CNT的值转运到CCR1里面去，这是输入捕获自动执行的让CNT = 0，清零计数器（从模式自动执行的），在一个周期之内，CNT在标准时钟的驱动下，不断自增，并且由于之前清零过了，所以CNT就是从上升沿开始，从0开始计数一直++，指导，下一次上升沿来临，然后执行相同的操作，CCR1 = CNT，CNT = 0，第二次捕获时CNT，继续执行操作

如果信号频率太低，CNT的计数值可能会溢出

想使用从模式自动清除CNT，只能用通道1和通道2，对于通道3和通道4，就只能开启捕获中断，在中断里手动清零了。（这样做程序会处于频繁中断的状态，比较消耗软件资源）

PWMI基本结构：



PWMI模式，使用了两个通道捕获一个引脚可以同时测量周期和占空比，TI1FP1配置上升沿触发，触发捕获和清零CNT，TI1FP2，配置为下降沿触发，通过交叉通道，去触发通道2的捕获单元，去触发通道2的捕获单元

例如：左上角图，最开始上升沿，CCR1捕获，同时清零CNT，之后CNT一直++，在下降沿这个时刻，触发CCR2捕获，这时CCR的值就是高电平期间的计数值，CCR2捕获不会触发CNT清零，CNT++，直到下一次上升沿，CCR1捕获周期，CNT清零，这样执行，CCR1就一整个周期的计数值，CCR2就是高电平期间的计数值，用CCR2/CCR1，就是占空比。

单独写入PSC的函数

```
void TIM_PrescalerConfig(TIM_TypeDef* TIMx, uint16_t Prescaler,
uint16_t TIM_PSCReloadMode);
```

输入捕获步骤

第一步，RCC开启时钟，把GPIO的TIM的时钟打开

第二步，GPIO初始化，把GPIO配置成输入模式，一般选择上拉输入或者浮空输入模式

第三步，配置时基单元，让CNT计数器在内部时钟的驱动下自增运行

第四步，配置输入捕获单元，包括滤波器、极性、直连通道还是交叉通道、分频器这些参数

第五步，选择从模式的触发源，触发源选择TI1FP1，调用一个库函数即可

第六步，选择触发之后执行的操作，执行Reset操作，调用一个库函数即可

第七步，调用TIM_Cmd函数，开启定时器

输入捕获常用函数

结构体配置输入捕获单元的函数

输出比较每个通道占用一个函数，输入捕获4个通道是共用一个函数的,在结构体中有额外的参数来选择通道

```
void TIM_ICInit(TIM_TypeDef* TIMx, TIM_ICInitTypeDef*
TIM_ICInitStruct);
```

另一个输入捕获的初始化函数

与上一个函数类似都是用于初始化输入捕获单元的，上一个函数只是单一的配置一个通道，而这个函数可以快速配置两个通道，把外设电路配置成PWMI的电路

```
void TIM_PWMIConfig(TIM_TypeDef* TIMx, TIM_ICInitTypeDef*
TIM_ICInitStruct);
```

给输入捕获结构体赋一个初始值

```
void TIM_ICStructInit(TIM_ICInitTypeDef* TIM_ICInitStruct);
```

选择输入触发源TRGI

调用此函数可以选择从模式的触发源

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

选择输出触发源**TRGO**

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

选择从模式

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

单独配置通道**1**、**2**、**3**、**4**的分频器

在参数结构体里也可以配置

```
void TIM_SetIC1Prescaler(TIM_TypeDef* TIMx, uint16_t TIM_ICPSC);
void TIM_SetIC2Prescaler(TIM_TypeDef* TIMx, uint16_t TIM_ICPSC);
void TIM_SetIC3Prescaler(TIM_TypeDef* TIMx, uint16_t TIM_ICPSC);
void TIM_SetIC4Prescaler(TIM_TypeDef* TIMx, uint16_t TIM_ICPSC);
```

读取四个通道的**CCR**

输出比较模式下，CCR是只写的，要用SetCompare写入，输入捕获模式下，CCR是只读的，要用GetCapture读出

```
uint16_t TIM_GetCapture1(TIM_TypeDef* TIMx);
uint16_t TIM_GetCapture2(TIM_TypeDef* TIMx);
uint16_t TIM_GetCapture3(TIM_TypeDef* TIMx);
uint16_t TIM_GetCapture4(TIM_TypeDef* TIMx);
```

编码器接口

Encoder Interface编码器接口

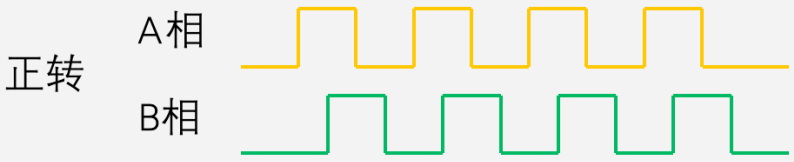
编码器接口可接收增量（正交）编码器的信号，根据编码器旋转产生的正交信号脉冲，自动控制CNT自增或自减，从而指示编码器的位置、旋转方向和旋转速度

每个高级定时器和通用定时器都拥有1个编码器接口

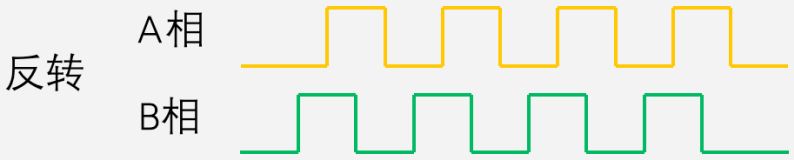
两个输入引脚借用了输入捕获的通道1和通道2

对于需要频繁执行，操作简单的任务，一般会设计一个硬件模块来自动完成

把两个编码器的A相和B相，接入STM32，定时器的编码器接口，编码器接口自动控制时基单元中的CNT计数器，进行自增或者自减，例如CNT初始值为0，编码器右转CNT++，右转产生一个脉冲，CNT++,左转CNT--，编码器接口（相当于带有方向控制的外部时钟）同时控制CNT的计数时钟和计数方向，CNT的值就表示了编码器的位置，每隔一段时间取一次CNT的值再把CNT清零，每次取出来的值就带标 了编码器的速度，编码器的测速实际上就是测频法测正交脉冲的频率，CNT计次，每隔一段时间取一次计次，也可以用外部中断来接编码器（用软件资源来弥补硬件资源）



边沿	另一相状态
A相↑	B相低电平
A相↓	B相高电平
B相↑	A相高电平
B相↓	A相低电平



边沿	另一相状态
A相↑	B相高电平
A相↓	B相低电平
B相↑	A相低电平
B相↓	A相高电平

当编码器的旋转轴转起来时，A相和B相就会输出方波信号，转的越快，方波的频率越高，方波的频率就代表了速度，取出任意一相的信号来测量频率，就能知道旋转速度，只有一相的信号无确定旋转方向。

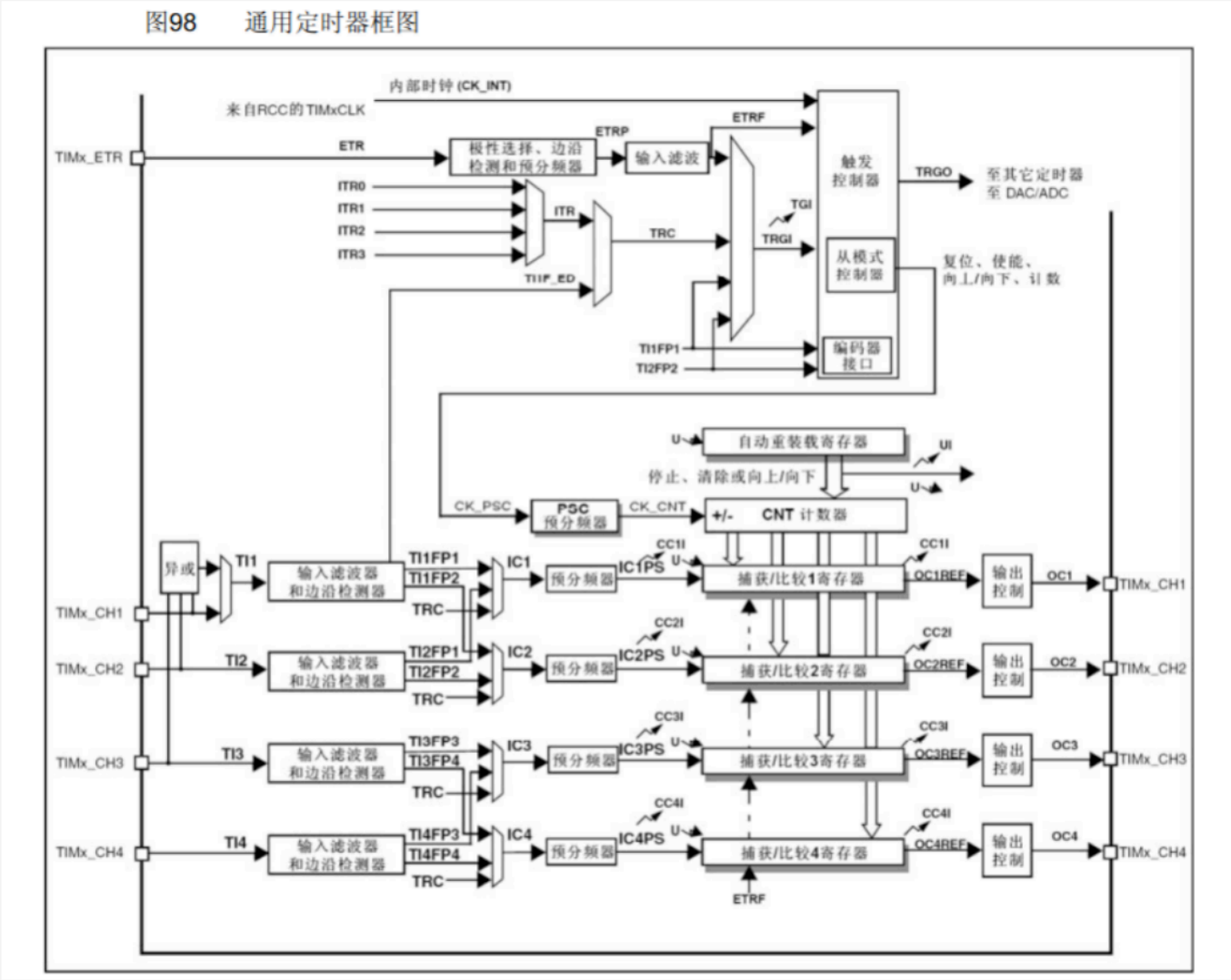
正交信号：当正转时，A相超前B相90度，翻转时，A相滞后B相90度。

正转时，第一个时刻，A相上升沿，对应B此时是低电平，第二个时刻，B相上升沿，对应A相高电平，第三个时刻，A相下降沿，对应B相高电平，B相下降沿，对应A相低电平。

反转时，第一个时刻，B相上升沿，对应A相低电平，第二个时刻A相上升沿，对应B相高电平，第三个时刻，B相下降沿，对应A相高电平，第四个时刻，A相下降沿，对应B相低电平。

当A、B相出现这些边沿时，对应另一相的状态，正转和反转正好是相反的

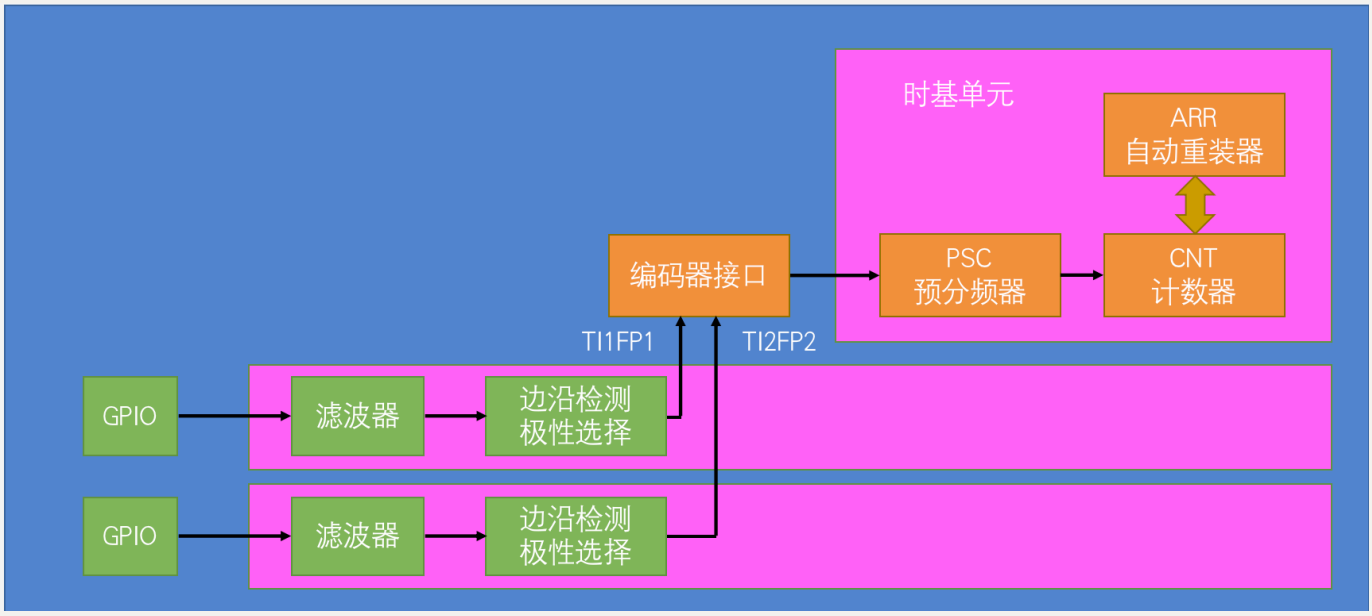
编码器接口的设计逻辑是：首先把A相和B相的所有边沿作为计数器的计数时钟，出现边沿信号时，就计数自增或者自减，



编码器接口的两个引脚，借用了输入捕获单元的前两个通道，编码器的输入引脚就是定时器的CH1和CH2两个引脚，信号的通路就是，CH1通过这里，通向编码器接口，CH3和CH4和编码器接口无关，其中CH1和CH2的输入捕获滤波器和边沿检测，编码器接口也有使用，但是后面的是否交叉，预分频器和CCR寄存器，与编码器接口无关，这就是编码器接口的输入部分，编码器接口的输出部分，相当于从模式控制器，控制CNT的计数时钟和计数方向，输出过程就是如果产生边沿信号，并且对应另一相的状态为正转，则控制CNT自增否则控制CNT自减，此时计数时钟和计数方向都处于编码器接口托管的状态，计数器的自增和自减，

受编码器的控制。

编码器接口的基本结构：



输入捕获的前两个通道，通过GPIO口接入编码器的A、B相然后通过滤波器和边沿检测极性选择，产生TI1TP1和TI2FP2，通向编码器接口，编码器接口通过控制预分频器控制CNT计数器的时钟，同时，编码器接口还根据编码器的旋转方向，控制CNT的计数方向，编码器正转时，CNT自增，编码器反转时，CNT自减，一般设置ARR为65535，最大量程

工作模式：

表77　计数方向与编码器信号的关系

有效边沿	相对信号的电平 (TI1FP1对应TI2, TI2FP2对应TI1)	TI1FP1信号		TI2FP2信号	
		上升	下降	上升	下降
仅在TI1计数	高	向下计数	向上计数	不计数	不计数
	低	向上计数	向下计数	不计数	不计数
仅在TI2计数	高	不计数	不计数	向上计数	向下计数
	低	不计数	不计数	向下计数	向上计数
在TI1和TI2上计数	高	向下计数	向上计数	向上计数	向下计数
	低	向上计数	向下计数	向下计数	向上计数

编码器接口的工作逻辑：TI1FP1和TI2FP2接的就是编码器的A、B相，在A相和B相的上升沿或者下降沿触发计数，向上计数还是向下计数取决于边沿信号发生时，另一相的电平状态（相对信号的电平）

配置流程：

第一步，RCC开启时钟，开启GPIO和定时器的时钟

第二步，配置GPIO，配置为输入模式

第三步，配置时基单元，预分频器选择不分频，自动重装，一般给最大65535

第四步，配置输入捕获单元，只需要配置滤波器和极性两个参数

第五步，配置编码器接口模式，调用一个库函数即可

第六步，调用TIM_cmd启动定时器

如果需要测量编码器的速度：每隔一段固定的闸门时间，取出一次CNT，然后把CNT清零

定时器编码器接口配置

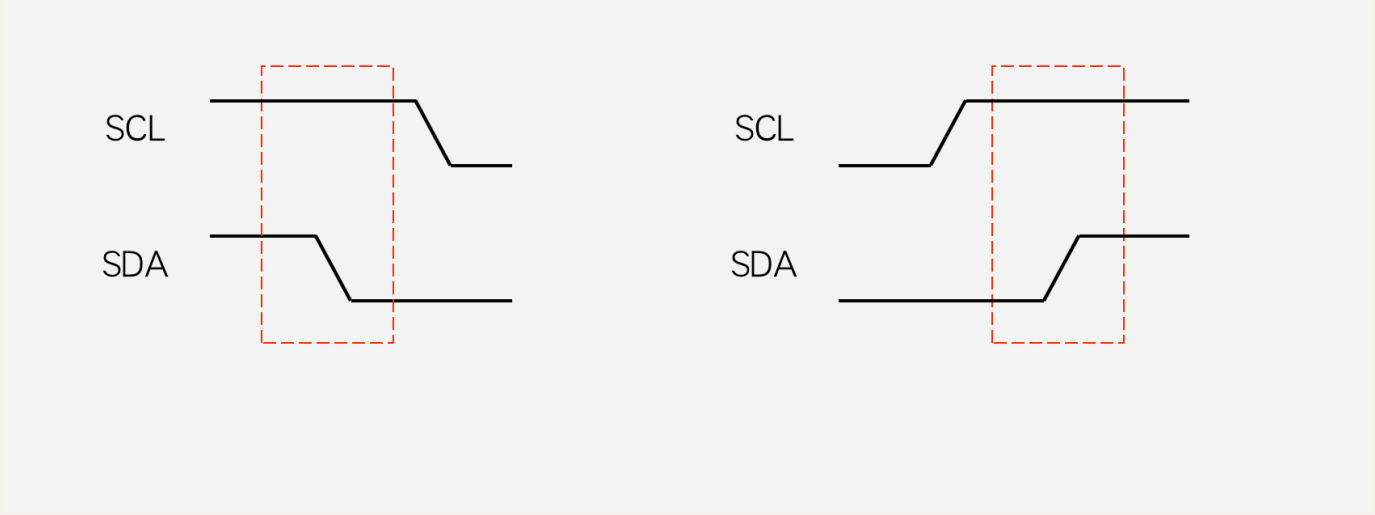
```
void TIM_EncoderInterfaceConfig(TIM_TypeDef* TIMx, uint16_t
TIM_EncoderMode,
                                uint16_t TIM_IC1Polarity, uint16_t
TIM_IC2Polarity);
```

配置上拉输入还是下拉输入：看外部模块输出的默认电平，与外部模块输出的默认电平相同，防止默认电平打架，如果不确定外部模块输出的默认状态，或者外部信号输出功率非常小，尽量选择浮空输入

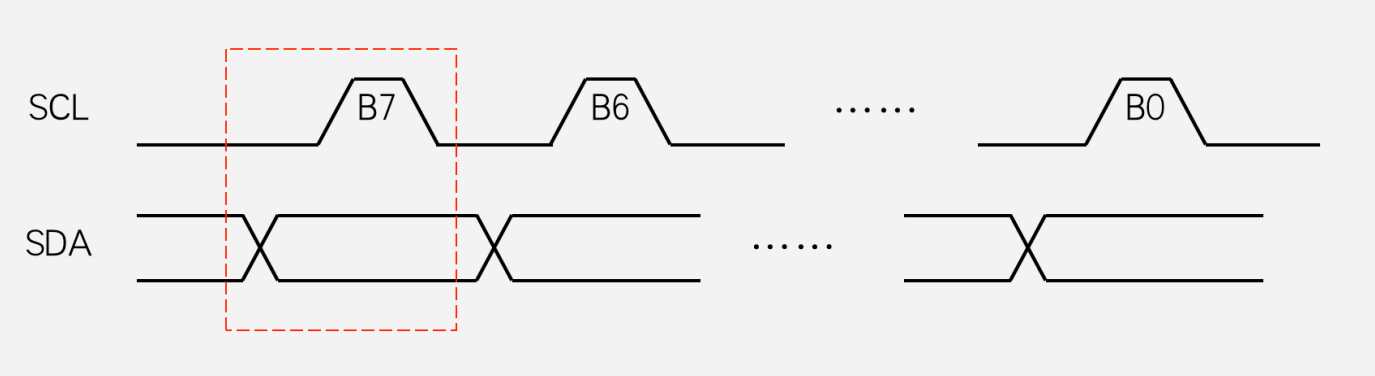
I2C

- I2C总线（Inter IC Bus）
- 两根通信线：SCL（Serial Clock）、SDA（Serial Data）
- 同步，半双工
- 带数据应答
- 支持总线挂在多设备（一主多从、多主多从）
- 起始条件：SCL高电平期间，SDA从高电平切换到低电平
- 终止条件：SCL高电平期间，SDA从低电平切换到高电平
- 每个时序单元的SCL都是以低电平开始，低电平结束

- 从机不允许产生起始和终止

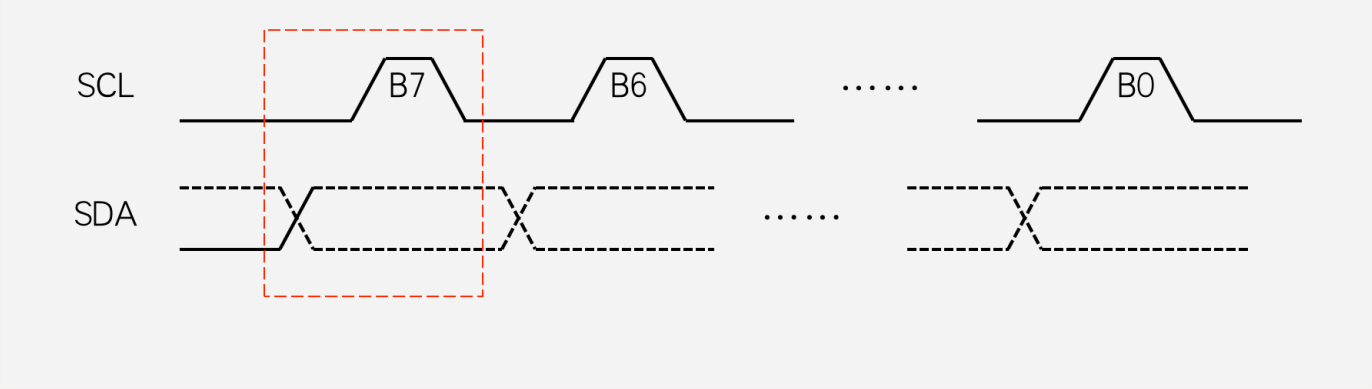


- 发送一个字节：SCL低电平期间，主机将数据位依次放到SDA线上（高位先行），然后释放SCL，从机将在SCL高电平期间读取数据位，所以SCL高电平期间SDA不允许有数据变化，依次循环上述过程8次即可发生一个字节

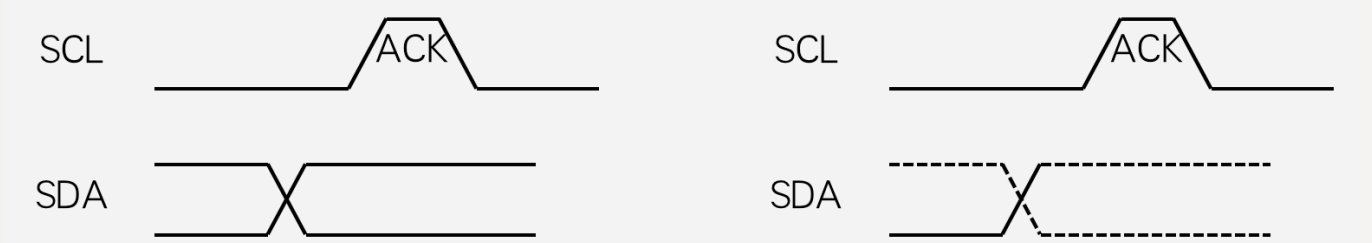


- 主机拉低SCL，把数据放在SDA上，主机松开SCL，从机读取SDA的数据
- （高位先行）在SCL低电平期间，主机如果想要发送0，就拉低SDA到低电平，如果想要发送1，就放手，SDA回弹到高电平，在SCL低电平期间允许改变SDA的电平，当这一位放好后，主机就松手时钟线，SCL回弹到高电平，在高电平期间是从机读取SDA的时候，SCL高电平期间，SDA不允许变化，SDA处于高电平时从机需要尽快读取SDA，一般是在上升沿的时刻，从机已经读取完成了，主机在放手SCL一段时间后，就可以继续拉低SCL传输下一位，主机需要在SCL下降沿之后尽快把数据放在SDA上，主机有时钟的主导权，不需要着急，只需要在低电平的任意时刻把数据放在SDA上就行了，数据放完之后，主机再松手SCL，SCL高电平从机读取这一位，在SCL的同步下，依次进行主机发送和从机接收，循环8次就发送了8位数据，也就是一个字节。
- 接收一个字节：SCL低电平期间，从机将数据位依次放到SDA线上（高位先行），然后释放SCL，主机将在SCL高电平期间读取数据位，所以SCL高电平期间期间

SDA不允许有数据变化，依次循环上述过程8次，即可接收一个字节（主机再接收之前，需要释放SDA，释放SDA就相当于切换为输入模式）。



- 也可以理解为：所有设备包括主机始终都属于输入模式，当主机需要发送的时候，就可以去主动拉低SDA，而主机再被动接收的时候，就必须先释放SDA，总线是线与的特征，任何一个设备拉低了，总线就是低电平，如果接收的时候还拽着SDA不放手，无论别人发什么数据，总线都始终属于是低电平。
- 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）

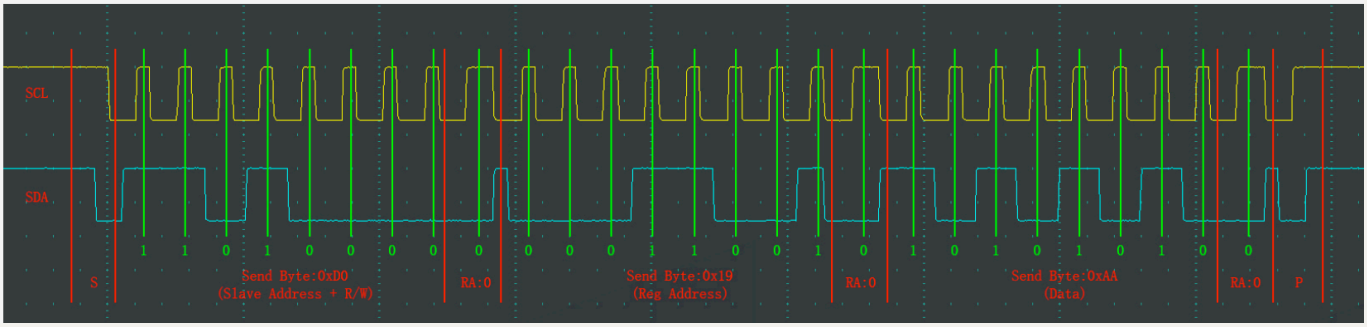


- 也可以理解为发送1位和接收1位，这一位用来作为应答，在发送完一个数据之后，就要立即进行接收应答，来判断从机是否接收到主机发送的数据
- 主机在起始条件之后，要先发送一个字节叫一下从机名字，所有从机都会收到第一个字节，与自己的名字（地址）比较，如果一样，相对应的从机就会响应主机的读写操作，在同一条I2C总线里，挂在的每个设别地址必须不一样，防止主机叫一个地址有多个设备都响应。
- 从机设备地址，在I2C协议标准里分为7位地址和10位地址
- 每个I2C设厂时，厂商都会为它分配一个7位的地址、
- MPU6050的地址是：1101 000
- 一般不同型号的设备地址都是不同的，相同型号的设备地址都是相同的
- 如果相同型号的设备挂在在同一条总线上，可以利用设备的地址的可变部分，一般

器件地址的最后几位是可以在电路中改变的，例如MPU6050地址的最后一位，由板子上的AD0引脚确定，AD0引脚接低电平，那它的地址就是1101 000，AD0引脚接高电平那它的地址就是1101 001，AT24C02地址的最后三位都可以分别由这个板子上的A0、A1、A2引脚确定

指定地址写

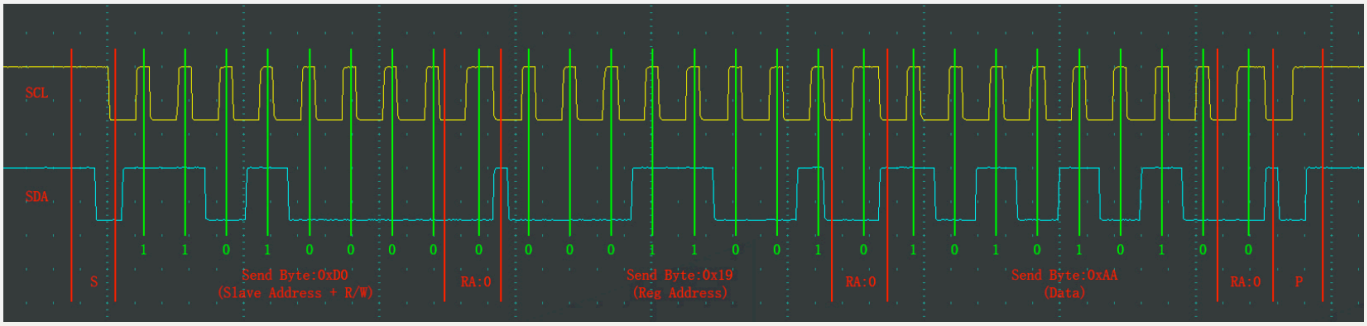
- 对于指定设备（Slave Address），在指定地址（Reg Address）下写入数据（Data）
- 空闲状态下两个总线都是高电平，主机需要给从机写入数据的时候，在SCL高电平期间，拉低SDA，产生起始条件，在起始条件之后紧跟的时序，必须是发送一个字节的时序，字节的内容必须是从机地址+读写位，（从机地址是7位，读写位是1位加起来就是1个字节8位）（发送从机地址：确定通信的对象），（发送读写位：确认接下来是要写入还是读出，0：写入，1：读出），紧跟着的单元是接收从机的应答位(Receive Ack,RA),这个时刻主机需要释放SDA，如果从机应答，从机会立即拉低SDA，应答位产生后，从机释放SDA，从机交出SDA的控制权，同样的时序再来一遍，第二个字节数据就会送入指定数据的内部，一般第二个字节是寄存器地址或者是指令控制字，第三个字节是想要往寄存器地址中写入的值，如果主机不想发送数据了，要产生停止条件，在产生停止条件之前，先拉低SDA，会后续的上升沿做准备，然后释放SCL，再释放SDA，产生SCL高电平期间SDA的上升沿



- 此数据帧的作用是：对于从机地址为1101000的设备在其内部0x19地址的寄存器中，写入0xAA这个数据

当前地址读

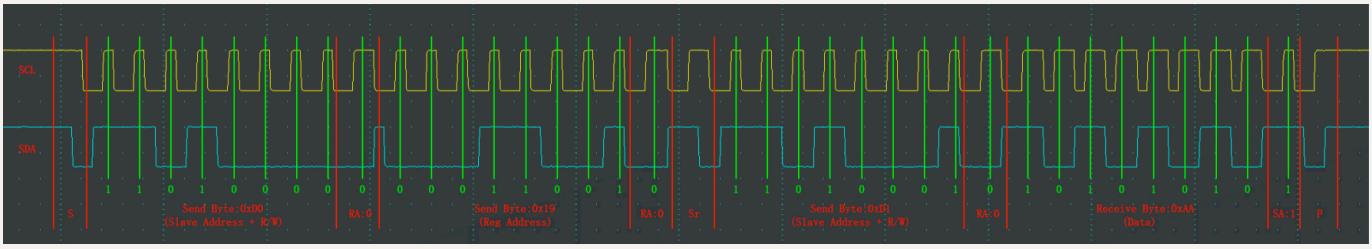
- 对于不指定设别（Slave Address），在当前地址指针指示的地址下，读取从机数据（Data）



- 在SCL高电平期间，拉低SDA，产生起始条件，主机首先发送一个字节，来进行从机的寻址和读写标志位，图示波形代表，本次寻址的目标是1101000的设备，读写标志为1，表示主机接下来想要读取设备，发送一个字节后，接收从机应答位，代表从机收到了第一个字节，把SDA的控制权交给从机，主机调用接收一个字节的时序，进行接收操作，从机接收到了主机的允许，可以在SCL低电平期间写入SDA，主机在哪SCL高电平期间读取SDA，主机再SCL高电平期间依次读取8位，就接收到了从机发送的一个字节数据0000 1111也就是（0x0F），没有指定地址这个环节，0x0F，（在从机中所有寄存器被分配到了一个线性区域中，会有个单独的指针变量，指示着其中一个寄存器，这个指针上电一般默认0地址，每写入一个字节或者读出一个字节后，这个指针就是自动自增一次，移动到下一个位置），从机返回的是当前指针指向的寄存器的值

指定地址读

- 对于指定设备（Slave Address），在指定地址（Reg Address）下读取从机数据（Data）



- 指定从机地址是1101000 读写标志位是0，代表要进行写的操作，经过从机应答后，在发送一个字节第二个字节0001 1001，用来指定地址，这个数据就写入到从机的地址指针里了，从机接收到这个地址后，它的寄存器指针就指向了0x19这个位置，不给从机发要写入的数据，而是再来个起始条件，起始条件后，重新寻址并且指定读写标志位，此时读写标志位为1代表开始读，继续主机接收一个字节，这个字节数据就是0x19地址下的数据。

写多个字节：重复三遍，发送一个字节和接收应答，第一个数据就写入0x19的位置（写入一个地址后地址指针会自动+1，编程吧0x1A）第二个数据就会写到0x1A的位置，第三个数据写入的是0x1B的位置

欧拉角：飞机与XYZ轴的夹角，反应了飞机的姿态，侧仰，上倾，下倾；

获得欧拉角需要多个数据，常用的数据融合算法：互补滤波、卡尔曼滤波等

MPU6050 XCL和SDA是扩展使用，通常是外接磁力计或者气压计