

# STM32F10x 常见应用解析



2008年9月

STM32  
全国巡回研讨会

# 系统时钟的监控和切换

## 起因:

在实际应用中，经常出现由于晶体振荡器在运行中失去作用，造成微处理器的时钟源丢失，从而出现死机的现象，导致系统出错。

**严重时，由于系统的死机造成监控失效，导致无法挽回的损失！**

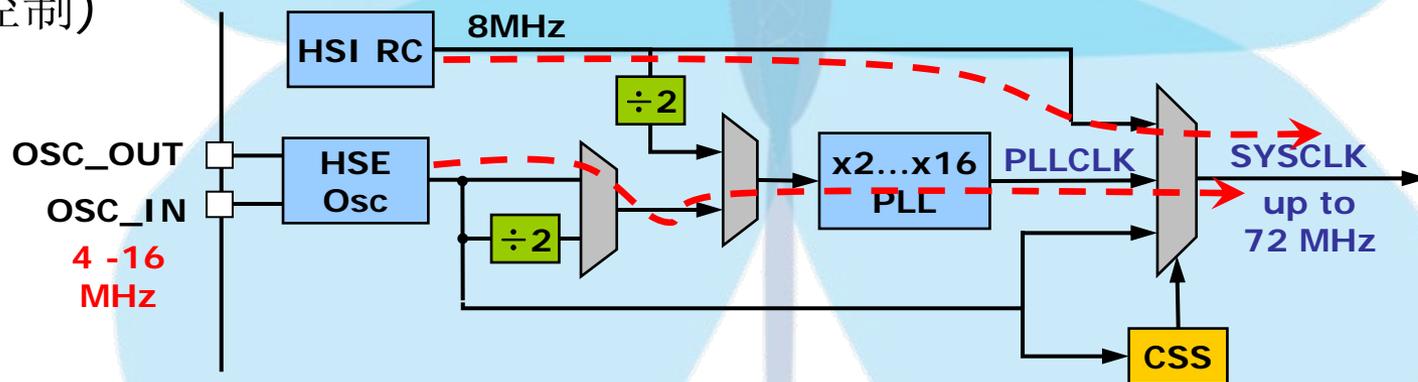
## 目的:

STM32作为一个可靠稳定的微处理器，但是不能排除由于某些外界特殊因素可能造成STM32的外部振荡器失效，所以在芯片中需要一种包含机制能够在STM32运行时，一旦外部晶体振荡器(HSE)失效，切换STM32的系统时钟源到一个稳定的时钟源，以保证STM32能够继续运行，并进行相应的保护操作。

# 系统时钟的监控和切换

## 时钟安全系统(CSS)系统原理:

时钟安全系统被激活后，时钟监控器将实时监控外部高速振荡器；如果HSE时钟发生故障，外部振荡器自动被关闭，产生时钟安全中断，此中断被连接到Cortex-M3的NMI的中断；与此同时CSS将内部RC振荡器切换为STM32的系统时钟源。(对于STM32F103，时钟失效事件还将被送到高级定时器TIM1的刹车输入端，用以实现电机保护控制)



- 注意：一旦CSS被激活，当HSE时钟出现故障时将产生CSS中断，同时自动产生NMI。NMI将被不断执行，直到CSS中断挂起位被清除。因此，在NMI的处理程序中必须通过设置时钟中断寄存器(RCC\_CIR)里的CSSC位来清除CSS中断。

# 系统时钟的监控和切换(应用举例)

## 启动时钟安全系统CSS:

RCC\_ClockSecuritySystemCmd(ENABLE); (NMI中断是不可屏蔽的!)

## 外部振荡器实现时，产生中断，对应的中断程序:

```
void NMIException(void)
```

```
{
```

```
if (RCC_GetITStatus(RCC_IT_CSS) != RESET)
```

```
{ // HSE、PLL已被禁止(但是PLL设置未变)
```

```
... .. // 客户添加相应的系统保护代码处
```

```
// 下面为HSE恢复后的预设置代码
```

```
RCC_HSEConfig(RCC_HSE_ON); // 使能HSE
```

```
RCC_ITConfig(RCC_IT_HSERDY, ENABLE); // 使能HSE就绪中断
```

```
RCC_ITConfig(RCC_IT_PLLRDY, ENABLE); // 使能PLL就绪中断
```

```
RCC_ClearITPendingBit(RCC_IT_CSS); // 清除时钟安全系统中断的挂起位
```

```
// 至此，一旦HSE时钟恢复，将发生HSERDY中断，在RCC中断处理程序里，  
系统时钟可以设置到以前的状态
```

```
}
```

```
}
```

## 在RCC的中断处理程序中，再对HSE和PLL进行相应的处理。

# 实现软件的短时间延迟

❏ 在进行开发时，程序中常常需要延时一段时间，很多人都会使用Delay(N)，N为需要延时的时间(通常为毫秒级)。

通常实现Delay(N)函数的方法为：

```
for(i = 0; i <= x; i ++);
```

*x --- 对应于N毫秒的循环值*

❏ 对于STM32系列微处理器来说，执行一条指令只有几十个ns，进行for循环时，要实现N毫秒的x值非常大，而且由于系统频率的宽广，很难计算出延时N毫秒的精确值。

❏ 针对STM32微处理器，需要重新设计一个新的方法去实现该功能，以实现在程序中使用Delay(N)。

# 应用SysTick实现短时延迟

- ❏ Cortex-M3的内核中包含一个SysTick时钟。SysTick为一个24位递减计数器，SysTick设定初值并使能后，每经过1个系统时钟周期，计数值就减1。计数到0时，SysTick计数器自动重装初值并继续计数，同时内部的COUNTFLAG标志会置位，触发中断(如果中断使能)。
- ❏ 在STM32的应用中，使用Cortex-M3内核的SysTick作为定时时钟，设定每一毫秒产生一次中断，在中断处理函数里对N减一，在Delay(N)函数中循环检测N是否为0，不为0则进行循环等待；若为0则关闭SysTick时钟，退出函数。
- ❏ 延迟时间将不随系统时钟频率改变。



# 应用 SysTick 举例

## 初始化相关模块:

```
SysTick_SetReload(9000); // 设定SysTick达到1ms计数结束  
SysTick_ITConfig(ENABLE); // 使能SysTick中断
```

## 中断处理:

```
void SysTickHandler (void) {  
    if (TimingDelay != 0x00)  
        TimingDelay--;  
}
```

全局变量**TimingDelay**  
必须定义为**volatile**

➤条件：外部晶振为8MHz，系统时钟为72MHz，SysTick的频率9MHz，SysTick产生1ms的中断。

## 延时代码:

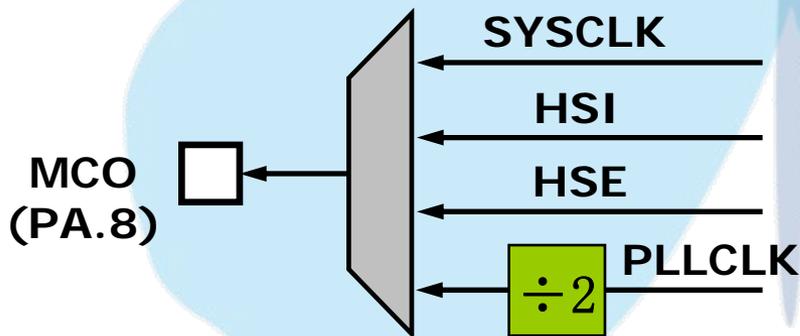
```
void Delay(u32 nTime) {  
    SysTick_CounterCmd(SysTick_Counter_Enable); // 使能SysTick计数器  
    TimingDelay = nTime; // 读取延时时间  
    while(TimingDelay != 0); // 判断延时是否结束  
    SysTick_CounterCmd(SysTick_Counter_Disable); // 关闭SysTick计数器  
    SysTick_CounterCmd(SysTick_Counter_Clear); // 清除SysTick计数器  
}
```

## 应用代码:

```
Delay(300); // 延时 300ms
```

# 输出芯片内部时钟

- 在实际应用中，一些用户常常遇到某些外设需要对其输入外部时钟或方波，针对这一需求，常用的方法是使用软件模拟，或使用有源晶振为其提供时钟或方波。
- STM32的PA.8引脚具有复用功能——时钟输出(MCO)，该功能能将STM32内部的时钟通过PA.8输出，这解决客户的问题，同时降低了硬件成本。



由于STM32 GPIO输出管脚的最大响应频率为50MHz，如果输出频率超过50MHz，则输出的波形会失真。

# 应用举例

❏ 设置PA.8为复用Push-Pull模式。

```
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8;  
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;  
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;  
GPIO_Init(GPIOA, &GPIO_InitStructure);
```

❏ 输出时钟：

❏ 时钟的选择由时钟配置寄存器(RCC\_CFGR)中的MCO[2:0]位控制。

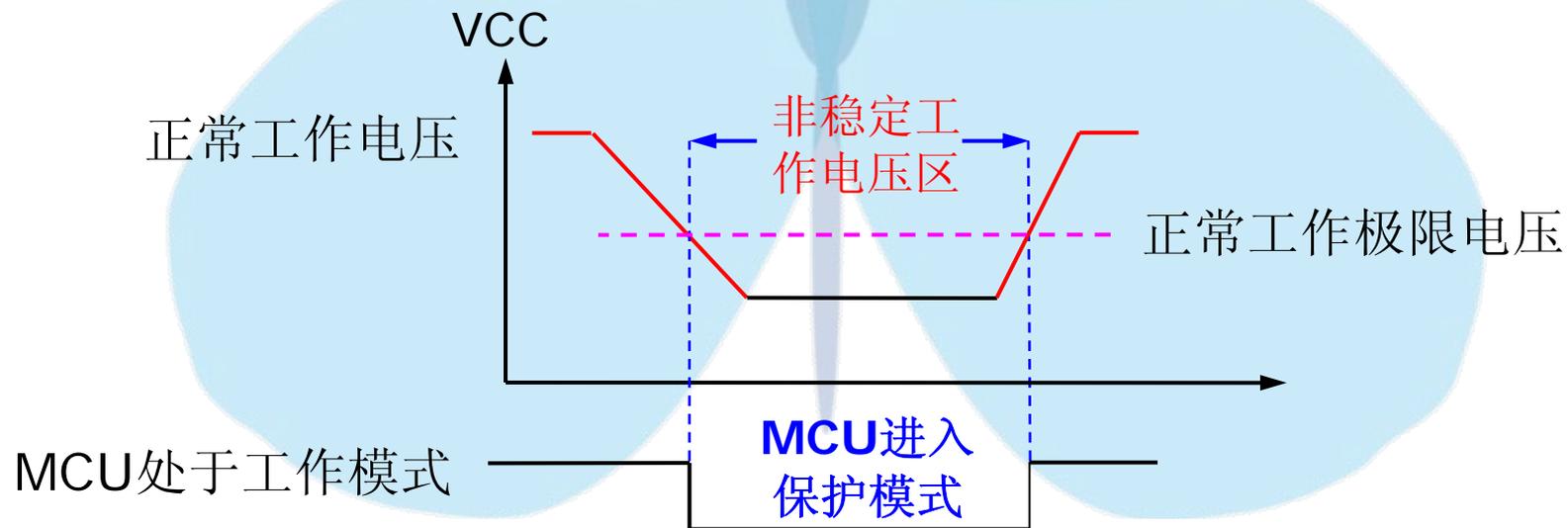
```
RCC_MCOConfig(RCC_MCO);
```

参数RCC\_MCO为要输出的内部时钟：

```
RCC_MCO_NoClock --- 无时钟输出  
RCC_MCO_SYSCLK --- 输出系统时钟 (SysCLK)  
RCC_MCO_HSI --- 输出内部高速8MHz的RC振荡器的时钟 (HSI)  
RCC_MCO_HSE --- 输出高速外部时钟信号 (HSE)  
RCC_MCO_PLLCLK_Div2 --- 输出PLL倍频后的二分频时钟 (PLLCLK/2)
```

# PVD的使用

- 在应用开发中，通常都要考虑到当系统电压下降或掉电状况，一旦出现该状况应对控制系统加以保护。
- 故在程序中需要加入对系统电压的监控。
  - 当供电电压降低到某一电压值时，需要系统进入特别保护状态，执行紧急关闭任务：对系统的一些数据保存起来，同时对外设进行相应的保护操作。



# 常用设计思路

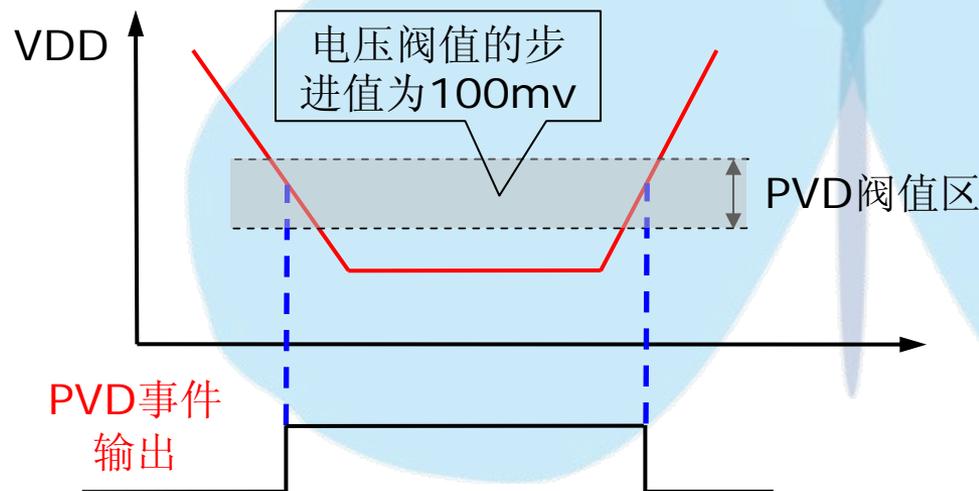
通常使用的方法是使用ADC对工作电压进行监控，MCU每隔一段时间读取ADC的转换值，并进行比较，判别电压是否下降到安全电压以下，电压正常时程序进行其他任务的运行；电压异常时程序进行保护模式，对相关寄存器、外设进行保护操作。这种方法会占用MCU的处理时间，同时使用ADC增加了系统的功耗。

STM32内部自带一个可编程电压监测器(PVD)，用于对VDD的电压进行监控。对应的电源控制寄存器中的PLS[2:0]位可用来设定监控电压的阈值，通过对外部电压进行比较来监控电源。



# 使用STM32的PVD

电源控制/状态寄存器(PWR\_CSR)中的PVDO标志用来表明VDD是高于还是低于PVD的电压阈值。当VDD下降到PVD阈值以下和/或当VDD上升到PVD阈值之上时，根据外部中断第16线的上升/下降边沿触发设置，就会产生PVD中断。在中断处理程序中执行紧急关闭任务：将MCU的寄存器、内存的数据保存起来，同时对外设进行相应的保护操作。



PLS[2:0]	电压阈值
000	2.2V
001	2.3V
010	2.4V
011	2.5V
100	2.6V
101	2.7V
110	2.8V
111	2.9V

# PVD 应用举例

☞ 系统启动后启动PVD，并开启相应的中断

```
PWR_PVDLevelConfig(PWR_PVDLevel_2V8); // 设定监控阈值
PWR_PVDCmd(ENABLE); // 使能PVD
EXTI_StructInit(&EXTI_InitStructure);
EXTI_InitStructure.EXTI_Line = EXTI_Line16; // PVD连接到中断线16上
EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt; //使用中断模式
EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Raising; //电压低于阈值时产生中断
EXTI_InitStructure.EXTI_LineCmd = ENABLE; // 使能中断线
EXTI_Init(&EXTI_InitStructure); // 初始化中断控制器
```

◆ *EXTI\_InitStructure.EXTI\_Trigger*的赋值可选项:

- **EXTI\_Trigger\_Rising** --- 表示电压从高下降到低于设定阈值时产生中断;
- **EXTI\_Trigger\_Falling** --- 表示电压从低上升到高于设定阈值时产生中断;
- **EXTI\_Trigger\_Rising\_Falling** --- 表示电压上升或下降越过设定阈值时都产生中断。

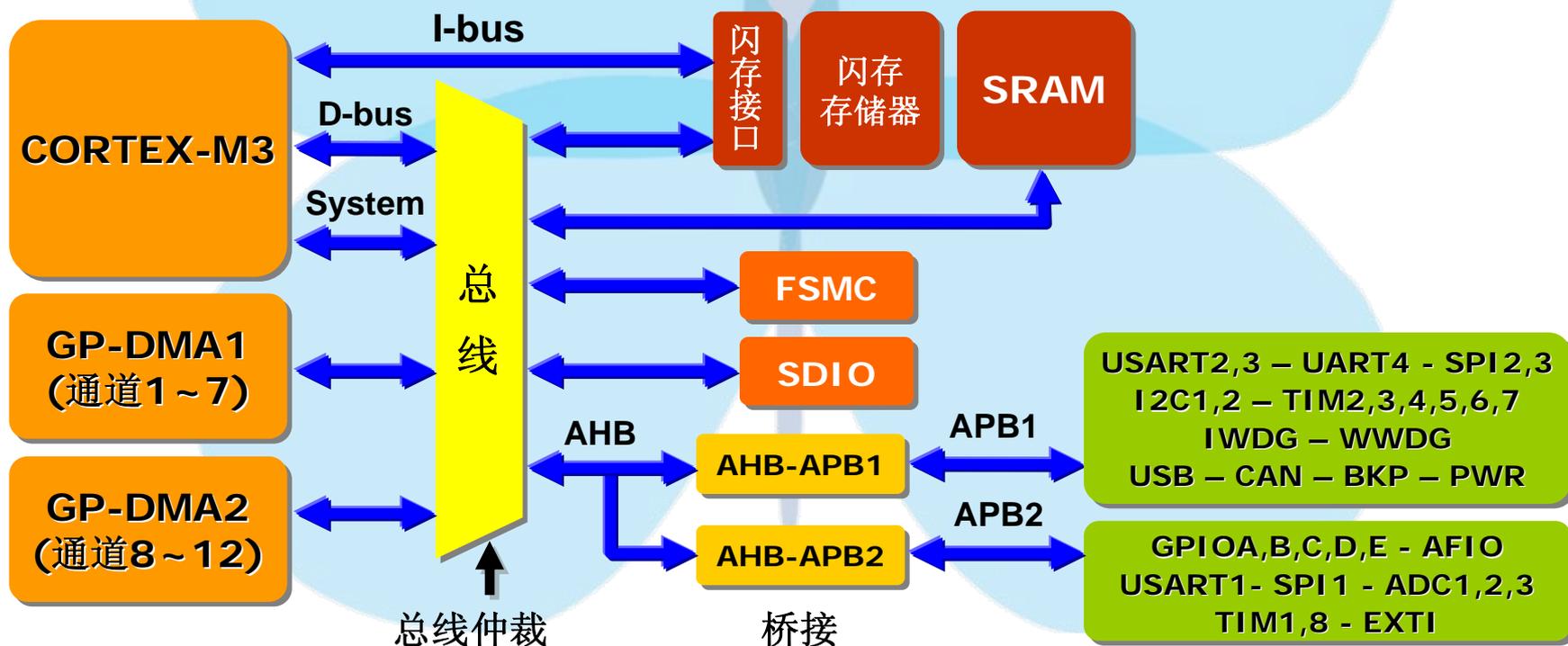
☞ 当工作电压低于设定阈值时，将产生一个中断，在中断程序中进行相应的处理:

```
void PVD_IRQHandler(void) {
    EXTI_ClearITPendingBit(EXTI_Line16);
    ... .. // 用户添加紧急处理代码处
}
```

# DMA传输模块

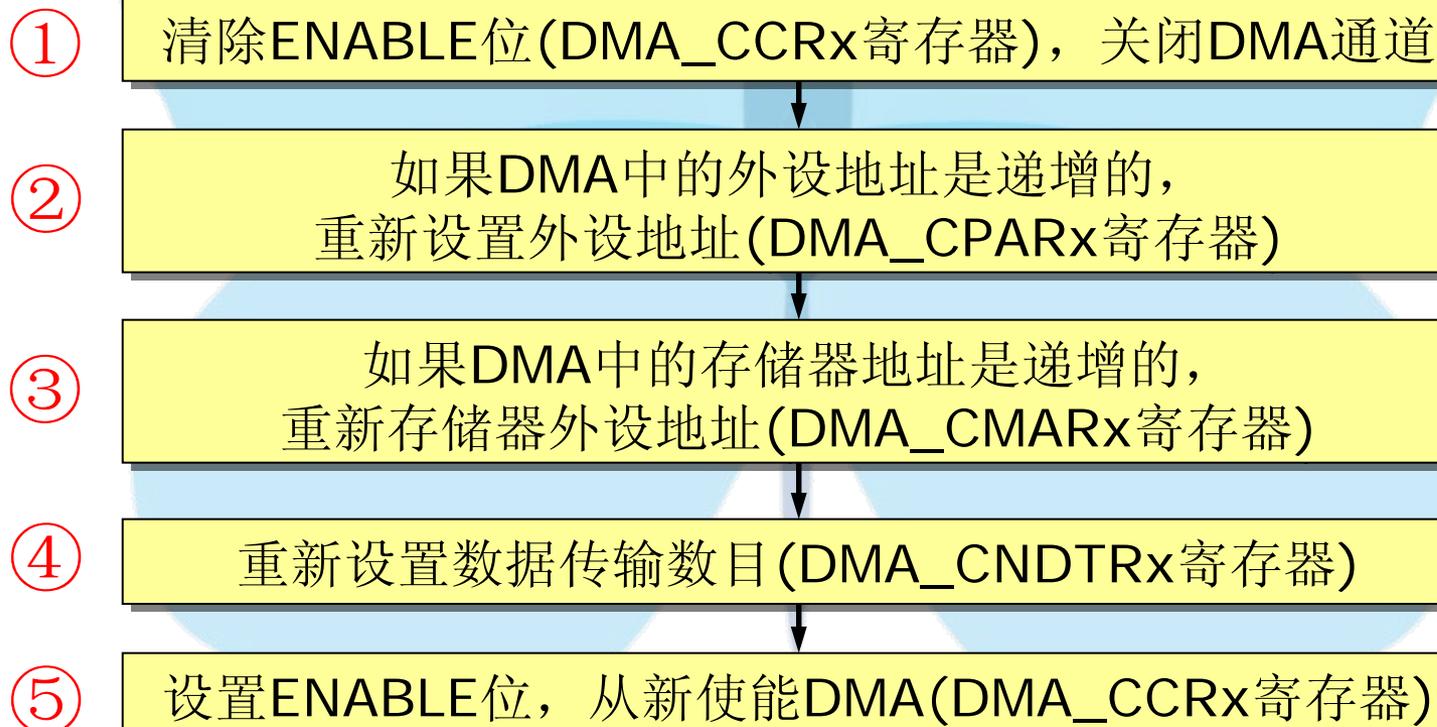
## 普通模式和循环模式的区别

- 循环模式：用于处理一个环形的缓冲区，每轮传输结束时数据传输的配置会自动地更新为初始状态，DMA传输会连续不断地进行。
- 普通模式：在DMA传输结束时，DMA通道被自动关闭，进一步的DMA请求将不被满足。



# 如何重新启动DMA传输

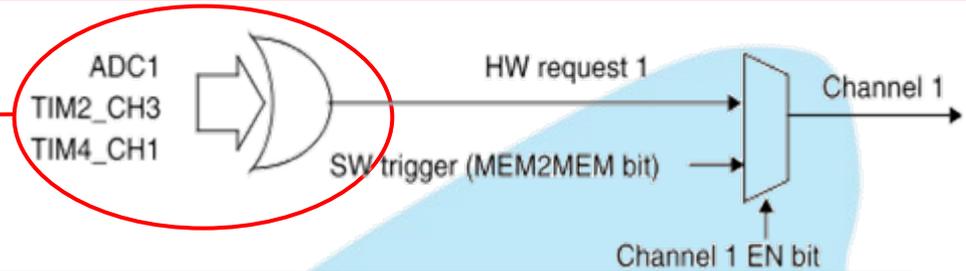
- 方案一：使用DMA的循环模式。
- 方案二：重新配置部分DMA的寄存器，方法如下
  - 第一次DMA传输配置为普通模式，然后从第二次开始：



# DMA的概念

- ❏ DMA是Direct Memory Access的首字母缩写。它的作用是不需要经过CPU而进行数据传输。
  - ❏ DMA控制器可以代替CPU驱动产生数据访问的地址并执行数据的读出和写入操作。
- ❏ DMA传输有三大要素：
  - ❏ 传输源：DMA控制器从传输源读出数据；
  - ❏ 传输目标：DMA控制器将数据传输的目标；
  - ❏ 触发信号：用于触发一次数据传输的动作，执行一个单位的传输源至传输目标的数据传输；可以用来控制传输的时机。
- ❏ STM32的DMA特征
  - ❏ 传输源和传输目标可以分别是存储器和/或片上外设，源和目标可以同为存储器或同为片上外设；
  - ❏ 一次数据传输的单位可以为：8位、16位或32位
  - ❏ 数据传输的触发信号由硬件确定，不能更改

# DMA触发源解读



设备	通道1	通道2	通道3	通道4	通道5	通道6	通道7
ADC1	ADC1						
SPI/I2S		SPI1_RX	SPI1_TX	SPI/I2S2_RX	SPI/I2S2_TX		
USART		USART3_TX	USART3_RX	USART1_TX	USART1_RX	USART2_RX	USART2_TX
I2C				I2C2_TX	I2C2_RX	I2C1_TX	I2C1_RX
TIM1		TIM1_CH1	TIM1_CH2	TIM1_CH4 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

- ❏ ADC1、TIM2\_CH3、TIM4\_CH1的DMA请求复用同一个通道，他们不能同时使用
- ❏ 定时器的各个通道不是使用的同一个DMA请求通道，而且不是所有通道都能触发DMA请求，例如：
  - ❏ TIM4的通道1、2、3和更新事件才有DMA请求
  - ❏ TIM3的通道1、2、3、触发和更新事件才有DMA请求

# DMA配置实例

在这个例子中使用了DMA通道6:

DMA请求(触发)源是定时器3

DMA数据源是GPIO输入寄存器/DMA目标是RAM中的缓冲区

设置外设地址:  $\text{DMA\_CPAR6} = (\text{u32})\&\text{GPIOID}\rightarrow\text{IDR}$

设置存储器地址:  $\text{DMA\_CMAR6} = (\text{u32})\ \text{Parallel\_Data\_Buffer}$

设置数据传输数目:  $\text{DMA\_CNDTR6} = 512$

配置 DMA\_CCR6 寄存器:

存储器数据宽度 = 半字 (16位)

外设数据宽度 = 半字 (16位)

外设地址递增 = 无递增

存储器地址递增 = 自动递增

传输方向 = 外设地址为传输的源地址

DMA 模式 = 循环

通道优先级 = 非常高

中断 = 禁止

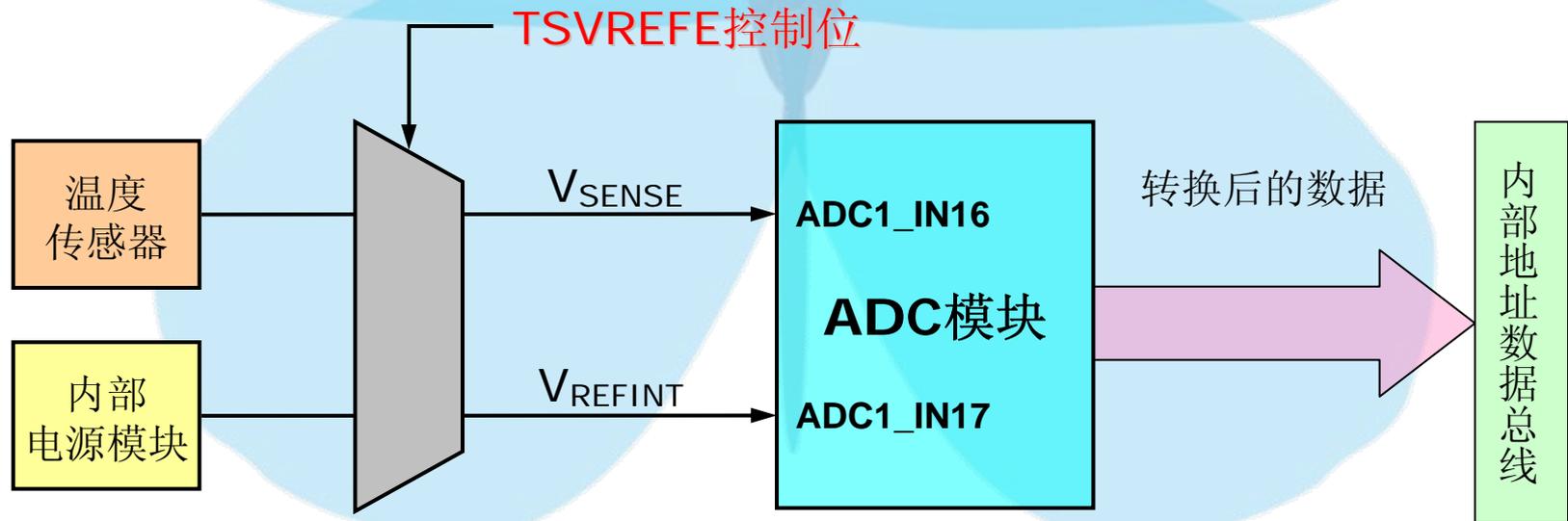
设置DMA\_CCR6寄存器中的ENABLE位, 启动传输通道

开启DMA请求: 设置TIM3\_DIER寄存器中的TDE位



# STM32的内部温度传感器

- STM32集成了片上的温度传感器，可以用来测量芯片内部的温度；
- STM32内部温度传感器与ADC的通道16相连，与ADC配合使用实现温度测量；
- 测量范围 $-40\sim 125^{\circ}\text{C}$ ，精度  $\pm 1.5^{\circ}\text{C}$ 。



# 使用STM32的内部温度传感器

## 配置步骤:

### 1. 设置ADC相关参数

```
// ADC1 configuration -----
ADC_InitStructure.ADC_Mode = ADC_Mode_Independent;
ADC_InitStructure.ADC_ScanConvMode = ENABLE;
ADC_InitStructure.ADC_ContinuousConvMode = ENABLE;
ADC_InitStructure.ADC_ExternalTrigConv = ADC_ExternalTrigConv_None;
ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;
ADC_InitStructure.ADC_NbrOfChannel = 1;
ADC_Init(ADC1, &ADC_InitStructure);
```

### 2. 选中ADC1的通道16作为输入

### 3. 设置采样时间17.1 us

### 4. 设置寄存器ADC\_CR2中的TSVREFE位激活温度传感器

```
// ADC1 regular channel16 Temp Sensor configuration
ADC_RegularChannelConfig(ADC1, ADC_Channel_16, 1, ADC_SampleTime_55Cycles5);
// Enable the temperature sensor and vref internal channel
ADC_TempSensorVrefintCmd(ENABLE);
```



$$N_{\text{cycle}} \times t_{\text{ADC}} = 17.1 \mu\text{s}$$

# 温度数值计算

ADC转换结束以后，读取ADC\_DR寄存器中的结果，通过下面的公式计算

$V_{25}$ : 温度传感器在25°C时的输出电压，典型值1.43 V

$V_{SENSE}$ : 温度传感器的当前输出电压，与ADC\_DR寄存器中的结果ADC\_ConvertedValue之间的转换关系为:

$$V_{SENSE} = \frac{ADC\_ConvertedValue * Vdd}{Vdd\_convert\_value(0xFFF)}$$

$$T(^{\circ}C) = \frac{V_{25} - V_{SENSE}}{Avg\_Slope} + 25$$

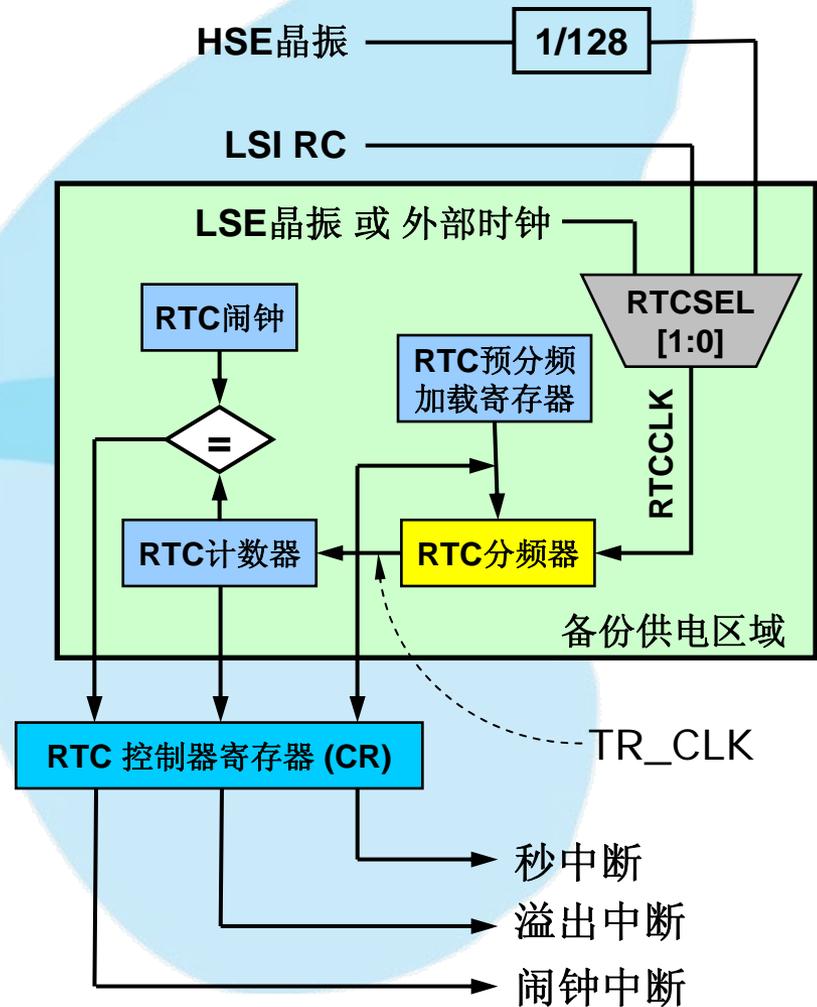
Avg\_Slope: 温度传感器输出电压和温度的关联参数，典型值4.3 mV/°C

转换程序

```
Vtemp_sensor = ADC_ConvertedValue * vdd / Vdd_convert_value;  
Current_Temp = (V25 - Vtemp_sensor)/Avg_Slope + 25;
```

# RTC系统框图

- ❏ 配置RTCCLK以及RTC\_DIV，使得预分频器产生频率为1秒的秒脉冲 (TR\_CLK)，作为RTC的时钟基准。
- ❏ 32位的RTC计数器，以1秒的频率进行计数，可以持续长达  
 $2^{32} = 4 * 1024 * 1024 * 1024$  (秒)  
 $= 49710$  (天)  
 $= 136$  (年)
- ❏ RTC系统的供电
  - ❏ 处于备份区域：即系统掉电后，只要有外部电池供电，该区域的功能可以继续工作；
  - ❏ 只要一直有电池供电，RTC的计数器持续计数，由此可以计算出实时的时间。



# 使用RTC制作万年历

## 上电后初始化系统时钟

## 配置RTC

- 使能PWR和BKP模块的时钟

- 使能对备份区域的访问

## 检查预定义的标记判断RTC是否曾经设置过

- 预定义的标记设置在备份区域的备份寄存器，不受系统掉电的影响

- 如果RTC未曾设置过，则初始化RTC模块：

- 时钟源LSE

- 秒中断的产生

## 设置RTC计数器

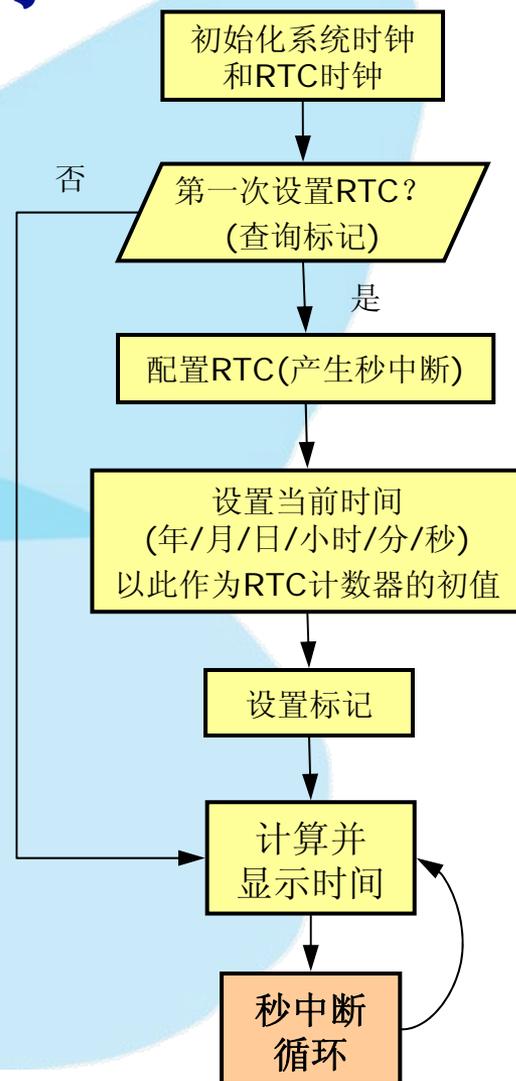
- 以一个基准作为时间原点

例如RTC计数值为0时表示：2008年1月1日 0:0:0

输入当前年/月/日/时/分/秒 → 计算离时间原点的秒数

## 显示时间

- 根据当前RTC计数器的值，得到现在距时间原点的秒数，反算出当前的年/月/日/时/分/秒。



# 校准实时时钟RTC

## 需求

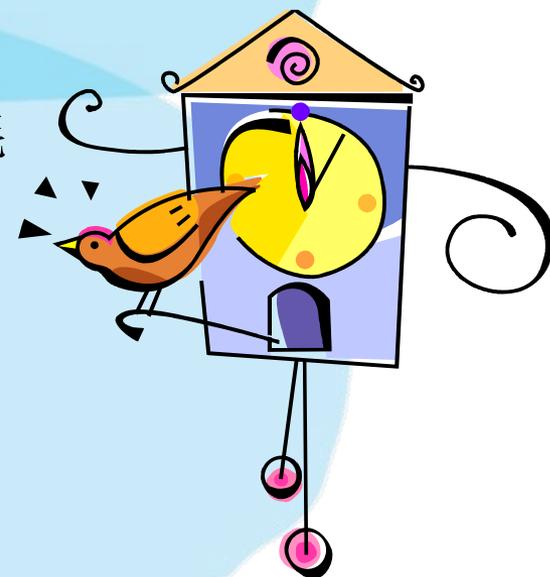
- ❏ 嵌入式系统使用的实时时钟精度要求高
- ❏ RTC晶振精度远远高于传统振荡器，精度可达到ppm级
- ❏ 受到温度等因素影响，晶振的精度会出现波动
- ❏ 对MCU提出了RTC时钟校准的要求

## 传统校准手段：调节晶振的负载电容

- ❏ 缺点：需要外部器件，可能增加晶振的功耗

## STM32中设置了BKP\_RTCCR寄存器，实现软件校准

- ❏ 优点：软件校准，灵活性高，易于使用



# 软件校准RTC的原理

- 设置BKP\_RTCCR寄存器，每 $2^{20}$ (1048576)个时钟周期中，减去相应周期数
- 每个单位能实现 $0.954(1000000/2^{20})$  ppm的精度校准
- BKP\_RTCCR寄存器取值范围0-127，时钟可以调慢0 -121 ppm
- 对于32,768Hz晶振，可补偿频偏范围为：

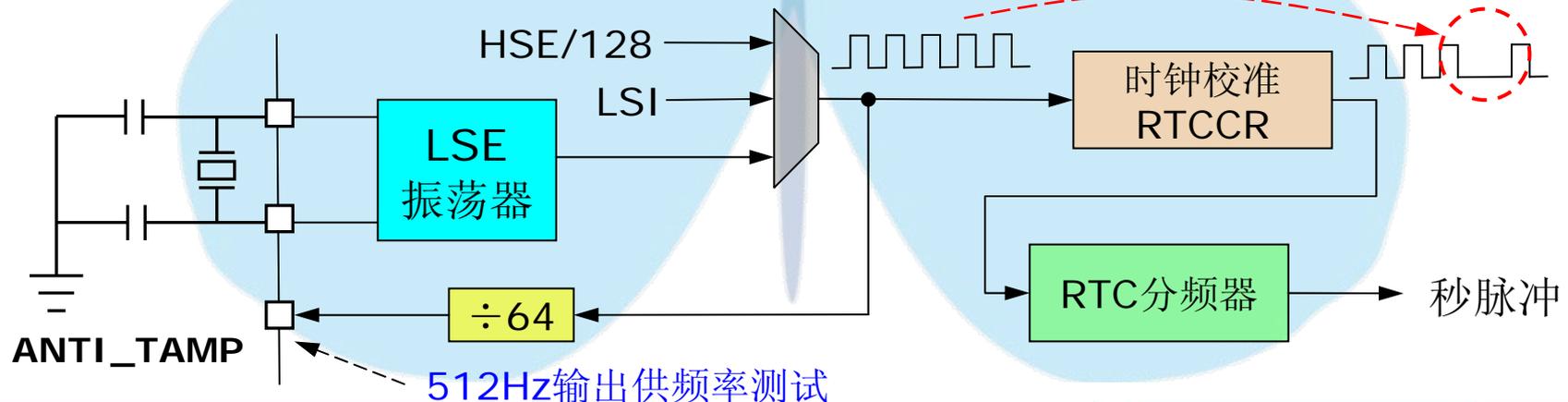
$$32,768\text{Hz} < f_{\text{LSE}} < 32,772\text{Hz} \quad \leftarrow \text{调慢}$$

- 设置RTC预分频寄存器RTC\_PRLH / RTC\_PRL

例如：由预设值32768调整为32766

- 再设置BKP\_RTCCR寄存器，此时，对于32,768Hz晶振，可补偿频偏范围：

$$32,766\text{Hz} < f_{\text{LSE}} < 32,770\text{Hz} \quad \leftarrow \text{调快}$$



# RTC校准实例

- 打开LSE，选择LSE为RTC时钟
- 设RTC预分频为**32766**(RTC\_PRLH/RTC\_PRLl写入**32765**)
- 将 $f_{\text{RTC}}/64$ 由MCO管脚输出(置BKP\_RTCCR寄存器CCO位为1)
- 通过测量，计算频率偏移，单位为ppm
  - 注意：**由于RTC预分频为32766，计算公式为  $\Delta f(\text{Hz})/511.968(\text{Hz})$
- 对照**校准表**查表得到BKP\_RTCCR寄存器应设的值
  - 校准表：AN2604，STM32F101xx and STM32F103xx RTC calibration application note, page 7 - Table 1 (见下页)  
<http://www.st.com/stonline/products/literature/an/13789.pdf>
- 要点
  - RTC各寄存器位于电池维持的后备域，校准信息**不会**因为复位、掉电而丢失；
  - 本RTC校准方法针对**长期**进行精度补偿，有限时间内补偿未必发生作用；
  - 可配合温度传感器进行适当的温度补偿。



**Table 1. Calibration table: compensation values in ppm and seconds per month (30 days)**

Calibration value	Value in ppm rounded to the nearest ppm	Value in seconds per month (30 days) rounded to the nearest second	Calibration value	Value in ppm rounded to the nearest ppm	Value in seconds per month (30 days) rounded to the nearest second
0	0	0	64	61	158
1	1	2	65	62	161
2	2	5	66	63	163
3	3	7	67	64	166
4	4	10	68	65	168
5	5	12	69	66	171
6	6	15	70	67	173
7	7	17	71	68	176
8	8	20	72	69	178
9	9	22	73	70	180
10	10	25	74	71	183
11	10	27	75	72	185

# 使用Timer进行周期定时

## 常见的应用场合

- 在STM32的某些应用中，用户有周期性执行某些程序的要求，使用定时器可以产生固定的时间周期，满足这样的需求。

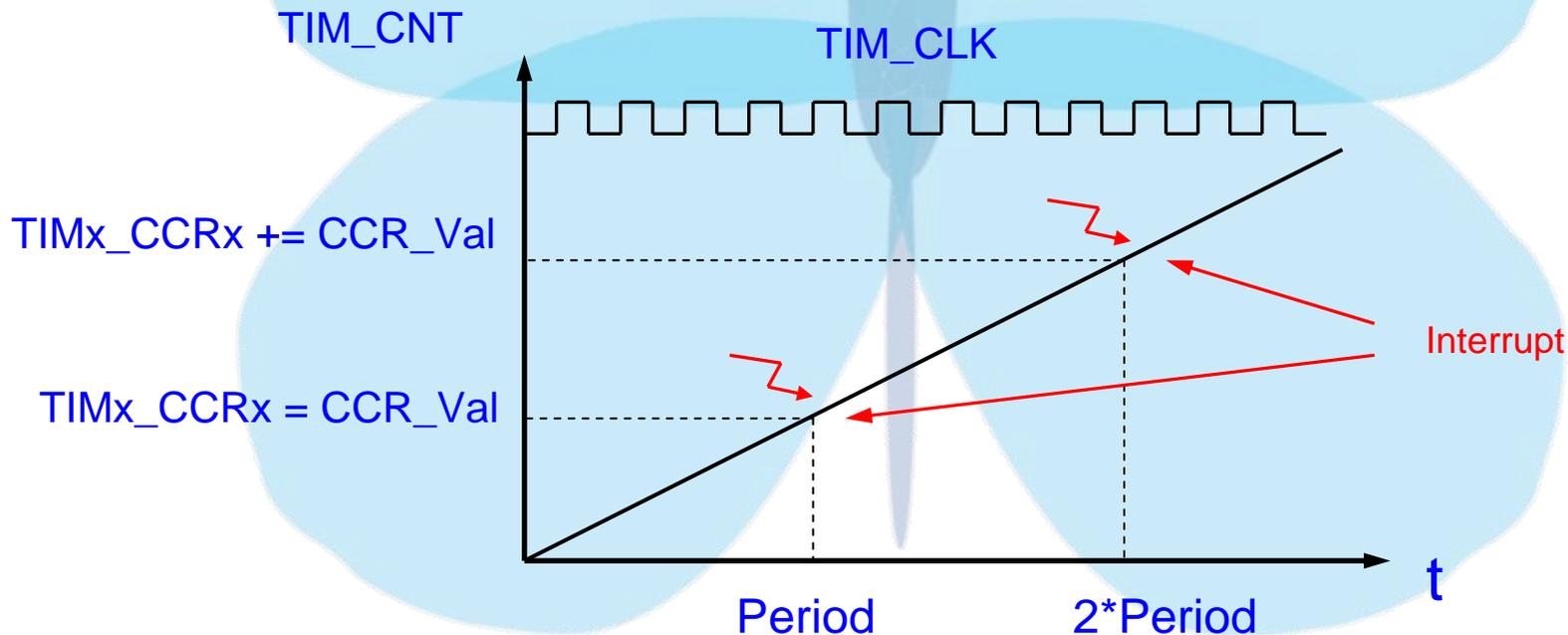
## STM32相关特征

- STM32高级定时器TIM1、TIM8，通用定时器TIM2、TIM3、TIM4、TIM5；
- 定时器最大时钟72MHz，配合预分频，提供灵活的时钟周期；
- 每个TIM有4个独立捕获/比较通道，DMA/中断功能；
- 通道工作在输出比较定时模式，一个TIM至多可以提供4个不同的定时周期。

# 使用Timer进行周期定时

## 原理

- TIM某输出/捕获通道工作在**输出比较定时模式**
- 计数器计数至比较值时产生中断，在中断中**刷新捕获比较寄存器**，这样在相同时间间隔后可产生下一次中断



# 使用Timer进行周期定时

## ❏ 示例

### ❏ 进行恰当的TIM基本设置

设置自动重载寄存器值为最大值0xFFFF。

```
/* Time base configuration */
TIM_TimeBaseStructure.TIM_Period = 0xFFFF; //设置自动重载寄存器值为最大值
TIM_TimeBaseStructure.TIM_Prescaler = UserdefinedPrescaler; // 自定义预分频
TIM_TimeBaseStructure.TIM_ClockDivision = 0x0;
TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;
TIM_TimeBaseInit(TIM4, &TIM_TimeBaseStructure);
```

### ❏ 设置捕获/比较通道，工作在输出比较定时模式

```
/* Output Compare Timing Mode configuration: Channel1 */
TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_Timing; // 输出比较定时模式
TIM_OCInitStructure.TIM_Channel = TIM_Channel_1;
TIM_OCInitStructure.TIM_Pulse = Userdefined_Period; // 用户定义定时的周期
TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_High;
TIM_OCInit(TIM4, &TIM_OCInitStructure);
```

# 使用Timer进行周期定时

- 通过设TIMx\_CCMRx的OCxPE位为0关闭预载入寄存器，这样对TIMx\_CCRx的修改会即时生效，打开相应通道的捕获/比较中断

```
TIM_OC1PreloadConfig(TIM4, TIM_OCPreload_Disable); //关闭预载入寄存器  
TIM_ITConfig(TIM4, TIM_IT_CC1, ENABLE); // 打开捕获比较中断
```

- 根据需要设置定时器的捕获比较寄存器值（TIMx\_CCRx），并在输出比较中断发生时，刷新该寄存器的值

- 在每次中断中，TIMx\_CCRx += Userdefined\_Period，修改Userdefined\_Period可即时修改定时的周期

```
void TIM4_IRQHandler(void)  
{  
    u16 capture;  
    if (TIM_GetITStatus(TIM4, TIM_IT_CC1) != RESET)  
    {  
        TIM_ClearITPendingBit(TIM4, TIM_IT_CC1);  
        capture = TIM_GetCapture1(TIM4);  
        // 设置新的CCRx值  
        TIM_SetCompare1(TIM4, capture + Userdefined_Period);  
        User_Application(); // 用户程序  
    }  
}
```

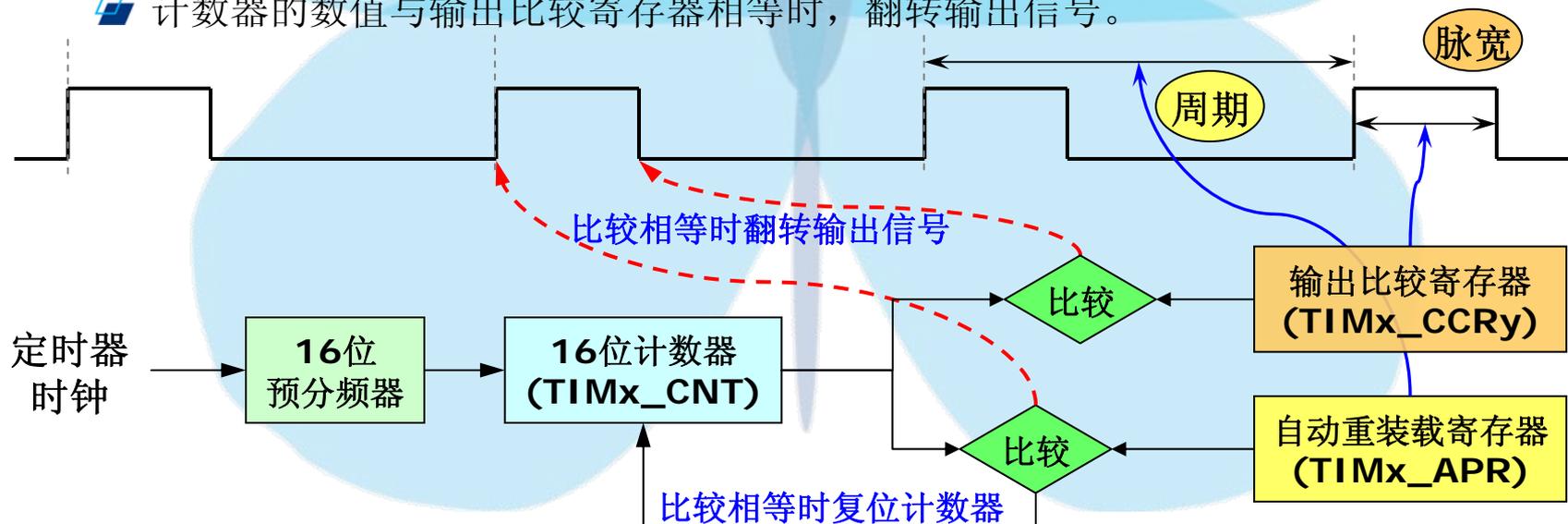
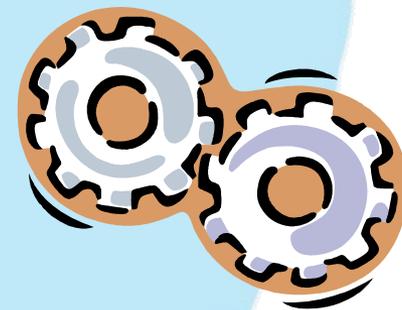
# 使用定时器产生PWM输出

STM32最多有6个定时器可以产生准确的PWM波形

- 高级定时器TIM1和TIM8，普通定时器TIM2~TIM5；
- 每个定时器有至少4个通道，可以产生4路PWM输出；

STM32定时器中的PWM模式

- 定时器时钟经预分频器分频后为计数器提供时钟；
- 重载寄存器和输出比较寄存器的数值不断与计数器比较；
- 计数器的数值与重载寄存器相等时，复位计数器并翻转输出信号；
- 计数器的数值与输出比较寄存器相等时，翻转输出信号。



## 使用定时器PWM模式举例

- 使用定时器的PWM模式，可以在一个定时器的4个不同的通道，产生4路频率相同但占空比不同的输出
- 以Timer3为例的配置过程
  - CK\_PSC(TIMxCLK)经预分频器得到计数器的时间基准CK\_CNT
  - Timer工作模式设置为PWM mode  $\rightarrow$  OCxM=110/111
  - 根据PWM的频率设置重载寄存器ARR:  $f = \text{CK\_CNT} / (\text{ARR} + 1)$
  - 根据各自需要的占空比设置CCRy: 占空比 =  $\text{CCRy} / (\text{ARR} + 1)$
  - 配置Timer3的4个通道所占用的I/O口  $\rightarrow$  AF-PP



# 定时器输出比较模式产生PWM输出

- 使用定时器的PWM模式只能在4个通道产生频率相同但占空比不同的输出信号
- 使用定时器的输出比较模式可以在4个通道上产生频率不同，占空比也不同的输出信号
- 以TIM2为例说明配置步骤
  - 置TIM2的4个通道所占用的I/O口(PA.0/1/2/3) → AF-PP
  - CK\_PSC(TIM2CLK)经预分频器得到计数器的时间基准CK\_CNT
  - 置TIM2为递增计数模式，并置重载数值为0xFFFF
  - 使用输出比较翻转(Output compare-toggle)模式 → OCxM=011
  - 根据需要的频率和占空比计算出输出高电平的时间和低电平的时间：
    - $OCH_y$  = 通道y的高电平时间； $OCL_y$  = 通道y的低电平时间
  - 把 $OCH_y$ 写入对应的输出比较寄存器，并启动计数器开始计数
  - 比较匹配后输出信号被翻转并产生中断
  - 每次中断中轮流把输出比较寄存器的数值增加 $OCH_y$ 或 $OCL_y$ ，即可产生希望的PWM输出

# 输出比较模式产生PWM计算实例

要求产生4个PWM输出：

输出1：频率=10kHz 占空比=40:60

输出2：频率=15kHz 占空比=30:70

输出3：频率=18kHz 占空比=10:90

输出4：频率=24kHz 占空比=50:50

通道	高电平	低电平	OCH	OCL
输出1	40us	60us	720	1080
输出2	20us	46.7us	360	840
输出3	5.56us	50us	100	900
输出4	20.8us	20.8us	375	375

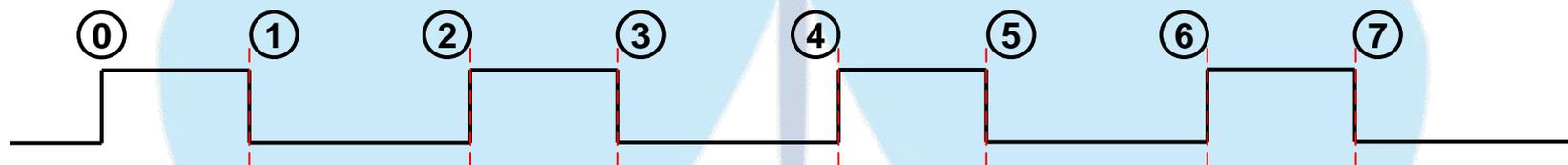
设置定时器输入时钟频率为72MHz，预分频器TIMx\_PSC=4

计算得到：计数器的时间基准CK\_CNT=4/72MHz

各通道的高低电平时间按计数器基准换算成OCH和OCL

输出比较计数器的数值变化如下图(以通道1为例)

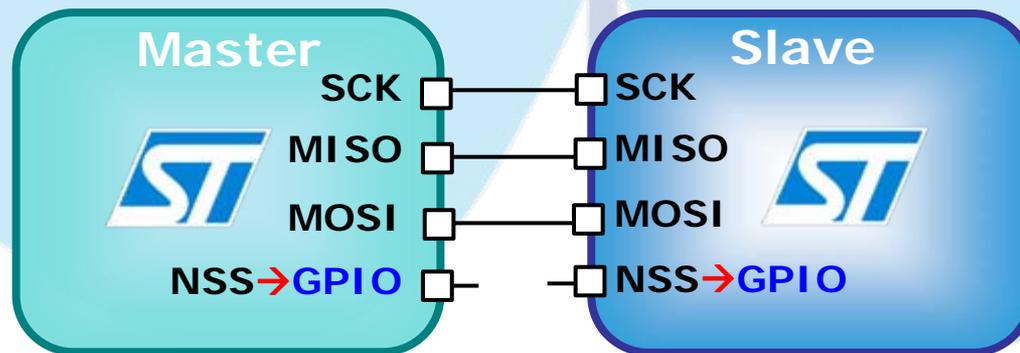
调整OCH和OCL即可调整频率和占空比



	第①次	第②次	第③次	第④次	第⑤次	第⑥次	.....
TIMx_CCR1	720	1800 (720+1080)	2520 (1800+720)	3600 (2520+1080)	4320 (3600+720)	7200 (4320+1080)	.....

# 使用SPI外设时如何设定NSS为通用IO口

- 主模式和从模式下均可以由软件或硬件进行NSS管理；
- 将SPI\_CR1寄存器的SSM位置为1时，NSS引脚将被释放出来用作GPIO口；
- 使用STM32软件库时，初始化SPI外设时，使用如下代码：
  - ▶ `SPI_InitStructure.SPI_NSS = SPI_NSS_Soft;`
- 如果NSS引脚用于其他外设时，需要使能NSS输出：
  - ▶ `SPI_SSOutputCmd(SPIx, ENABLE);`



# 使用USART操作SPI设备

## 为什么需要USART产生SPI信号

- 片上SPI接口不够用
- 需要特殊的数据格式：9位/字节
- 或需要带奇偶检验的数据字节
  - (7位数据+奇偶位)/字节
  - (8位数据+奇偶位)/字节

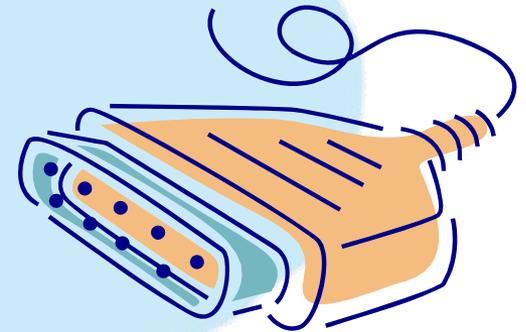
## SPI接口有以下属性

- 四条信号线：MISO & MOSI & SCK (&NSS)
- 时钟相位和极性 CPOL & CPHA
- 数据帧格式：MSB/LSB

## USART接口以及时序要求

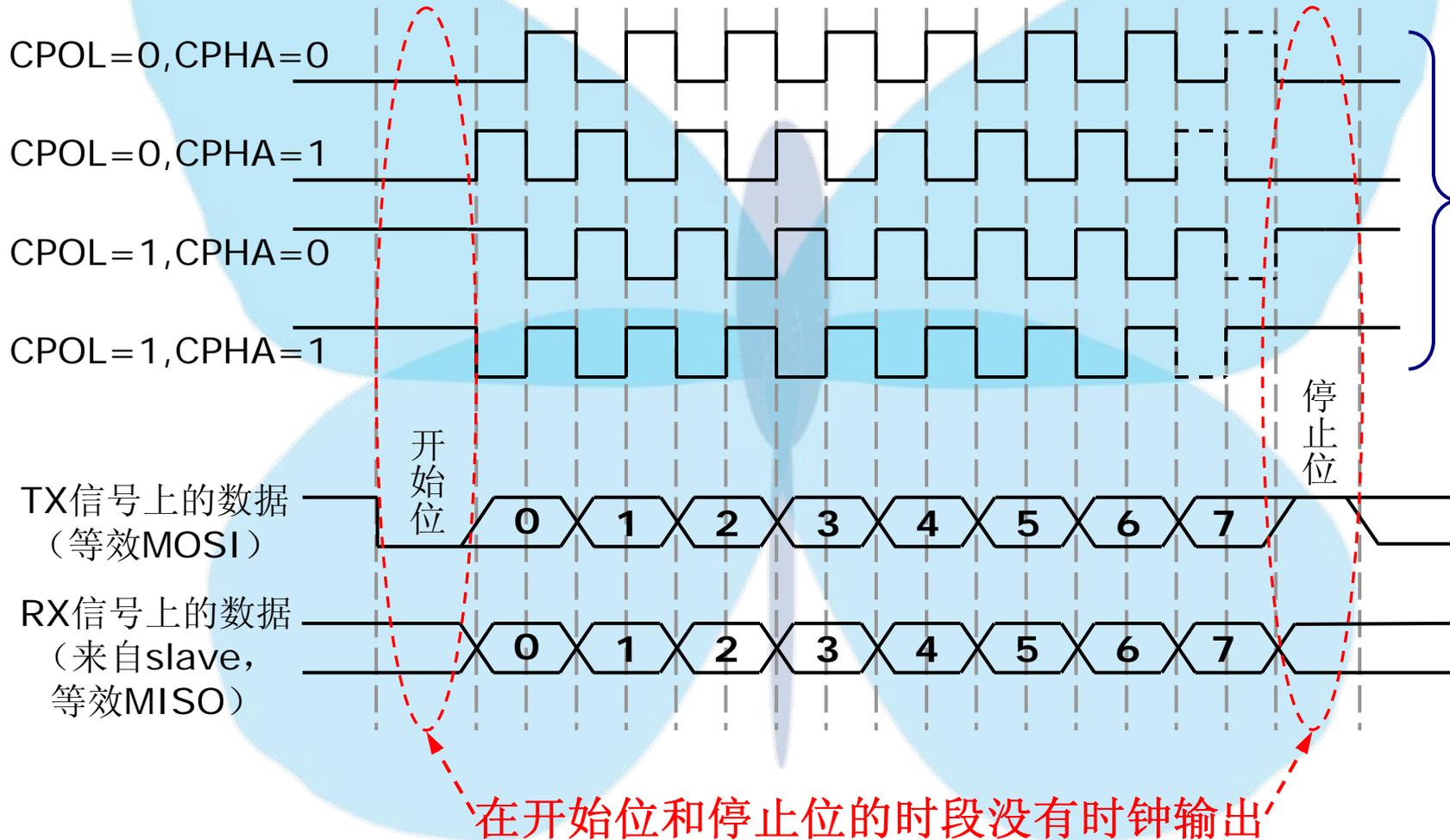
- RX & TX & SCLK
- 时钟相位和极性 CPOL & CPHA
- 数据帧格式：8/9位数据 (start和stop位没有对应的时钟脉冲)
- 最大时钟频率：4.5MHz

**SCLK只作为输出！**  
使用USART充当  
SPI只能做master  
用，不能当slave用



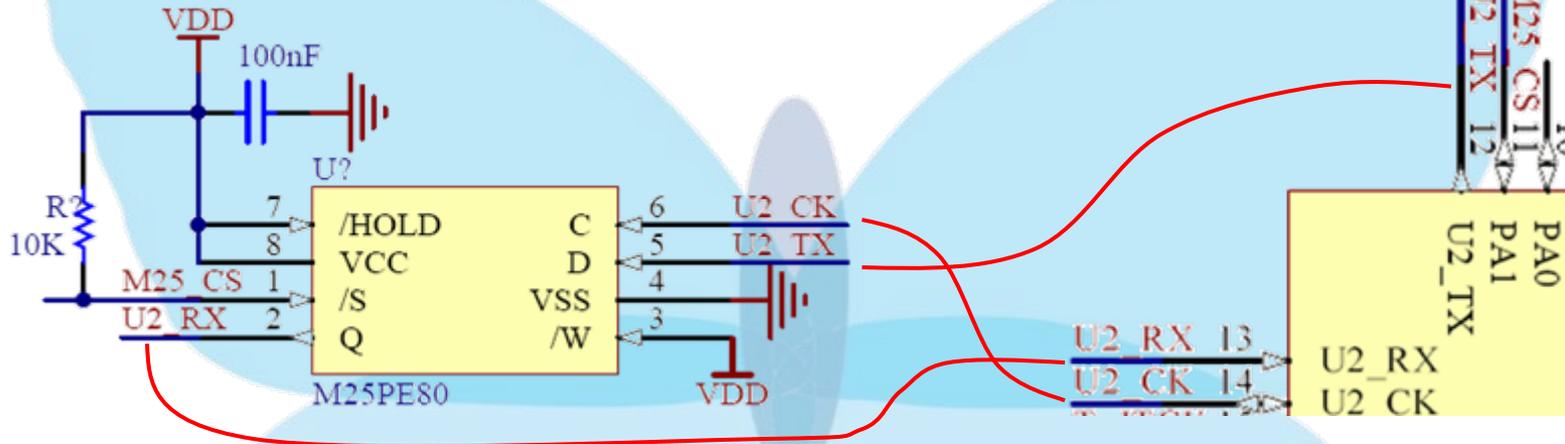
# USART同步方式时序

四种时钟输出方式



# USART作为SPI应用举例

## 使用USART接口和SPI Flash (M25PE80)通信



## 初始化USART

```
USART_InitStructure.USART_WordLength = USART_WordLength_8b;
USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None;
USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;
USART_InitStructure.USART_Clock = USART_Clock_Enable;
```

## 发送同时接收数据

```
while(USART_GetFlagStatus(USART2, USART_FLAG_TXE) == RESET);
USART_SendData(USART2, byte);
while(USART_GetFlagStatus(USART2, USART_FLAG_RXNE) == RESET);
return USART_ReceiveData(USART2);
```

# SPI 单线传输方式 (1/3)

## ❏ SPI单线传输的应用:

- ❏ 在与外部设备的通信中，如果只需要1个时钟线和1个数据线，可使用SPI的单线传输方式，节省出来的管脚可以用于其他用途。
- ❏ 限制：只能用作输入或者输出，或者工作在半双工模式下
- ❏ 应用范围：传感器、受控终端、控制端(控制数字电位器等)、通信、RAM/ROM等

## ❏ 原理（两种单线传输连接方式）

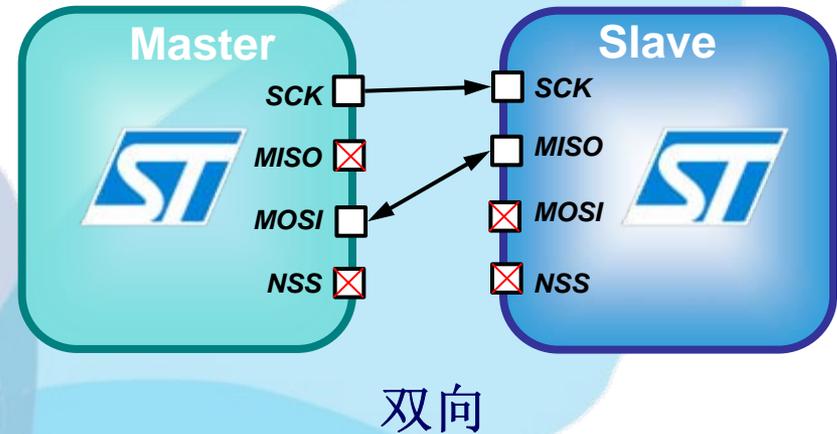
- ❏ 在SPI的通信设置中，当寄存器SPI\_CR1中的BIDIMODE = 1时，单线双向数据传输模式被选中，可通过SPI\_CR1中的BIDIOE位来选择数据的传输方向。将不需要的管脚设为GPIO
- ❏ 当只作为输入或输出设备时，将不需要的管脚设为GPIO



# SPI 单线传输方式 (2/3)

## 硬件连接 (双向数据传输)

- 当工作在Master模式下，使用MOSI管脚，MISO可以作为GPIO使用
- 当工作在Slave模式下，使用MISO管脚，MOSI可以作为GPIO使用



## 硬件连接 (单向数据传输)

- 如果传输设备一方只发送数据而另一方只接收数据时，可以采用右面这种连接方式，另一个数据管脚MISO可以作为GPIO使用



# SPI 单线传输方式 (3/3)

## 软件设置

当采用双向数据传输时按照下面设置：

```
SPI_InitStructure.SPI_Mode = SPI_Mode_Master;
SPI_InitStructure.SPI_Direction = SPI_Direction_1Line_Tx; /* Master, Transmit */
SPI_InitStructure.SPI_Mode = SPI_Mode_Master;
SPI_InitStructure.SPI_Direction = SPI_Direction_1Line_Rx; /* Master,
Receive */
SPI_InitStructure.SPI_Mode = SPI_Mode_Slave;
SPI_InitStructure.SPI_Direction = SPI_Direction_1Line_Tx; /* Slave,
Transmit */
SPI_InitStructure.SPI_Mode = SPI_Mode_Slave;
SPI_InitStructure.SPI_Direction = SPI_Direction_1Line_Tx; /* Slave,
Receive */
```

采用单向数据传输时：

将SPI要用到的管脚设为AF\_PP，将不需要的管脚设为GPIO

```
SPI_InitStructure.SPI_Mode = SPI_Mode_Master;
SPI_InitStructure.SPI_Direction = SPI_Direction_2Lines_FullDuplex; /* Master, Transmit */
注：主控端只能用作输出
SPI_InitStructure.SPI_Mode = SPI_Mode_Slave;
SPI_InitStructure.SPI_Direction = SPI_Direction_2Lines_RxOnly; /* Slave, Receive */
```

注：从端只能用作输入

将SPI要用到的管脚设为AF\_PP，将不需要的管脚设为GPIO

注意：当设置为从模式(Slave)时，需要注意Slave Select的设置是由软件控制还是硬件控制

# STM32 NVIC的优先级概念

❏ 为什么中断要有优先级

❏ 何为占先式优先级 (pre-emption priority)

❏ 高占先式优先级的中断事件会打断当前的主程序/中断程序运行——抢断式优先响应，俗称中断嵌套。

❏ 何为副优先级 (subpriority)

❏ 在占先式优先级相同的情况下，高副优先级的中断优先被响应；

❏ 在占先式优先级相同的情况下，如果有低副优先级中断正在执行，高副优先级的中断要等待已被响应的低副优先级中断执行结束后才能得到响应——非抢断式响应(不能嵌套)。

❏ 判断中断是否会被响应的依据

❏ 首先是占先式优先级，其次是副优先级；

❏ 占先式优先级决定是否会有中断嵌套；

❏ Reset、NMI、Hard Fault 优先级为负(高于普通中断优先级)且不可调整。

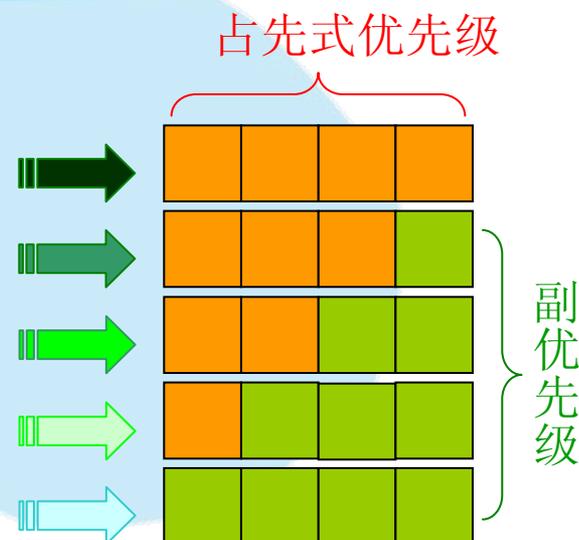


# 如何配置STM32中断的优先级

- ❏ 每一个中断都有一个专门的寄存器(Interrupt Priority Registers)来描述该中断的占先式优先级及副优先级
  - ❏ 在这个寄存器中STM32使用4个二进制位描述优先级（Cortex-M3定义了8位，但STM32只使用了4位）
- ❏ 占先式优先级与副优先级的分配
  - ❏ 4个描述优先级位有下列5种组合使用方式
  - ❏ “优先级组别”决定如何解释这4位。

4个优先级描述位可以有5种组合方式。

优先级组别	占先式优先级	副优先级
4	4位/16级	0位/0级
3	3位/8级	1位/2级
2	2位/4级	2位/4级
1	1位/2级	3位/8级
0	0位/0级	4位/16级



# STM32设置优先级举例

选择两位占先式优先级和两位副优先级，即第2组优先级配置，（0~3级占先式优先级/0~3级副优先级） ①

设置EXTI9\_5的中断优先级为：

占先式优先级 = 2 ②

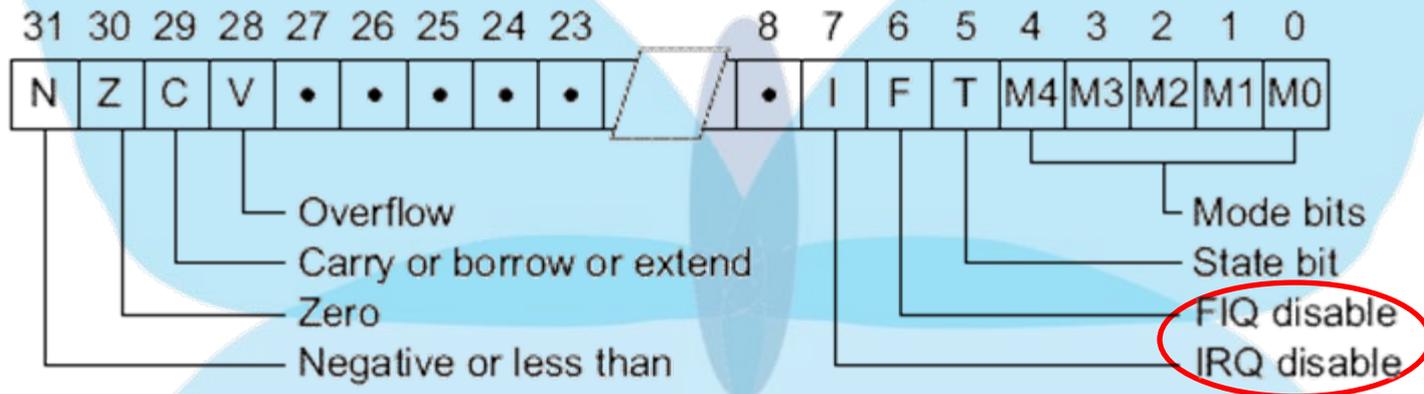
副优先级 = 1 ③

```
/* Configure the Priority Group to 2 bits */
NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2); ①

/* Enable the EXTI9_5 Interrupt */
NVIC_InitStructure.NVIC_IRQChannel = EXTI9_5_IRQChannel;
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 2; ②
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1; ③
NVIC_Init(&NVIC_InitStructure);
```

# STM32的总中断控制

- 在STR7/STR9使用的ARM7/9核心中，在CPSR(当前程序状态寄存器)中有两位分别控制IRQ和FIQ



- 在STM32/Cortex-M3中是通过改变CPU的当前优先级来允许或禁止中断
  - PRIMASK位：只允许NMI和hard fault异常，其他中断/异常都被屏蔽(当前CPU优先级=0)。
  - FAULTMASK位：只允许NMI，其他所有中断/异常都被屏蔽(当前CPU优先级=-1)。

# 总中断控制函数

在STM32固件库中 (*stm32f10x\_nvic.c*和*stm32f10x\_nvic.h*) 定义了四个函数操作PRIMASK位和FAULTMASK位，改变CPU的当前优先级，从而达到控制所有中断的目的。

下面两个函数等效于关闭总中断：

```
void NVIC_SETPRIMASK(void);  
void NVIC_SETFAULTMASK(void);
```

下面两个函数等效于开放总中断：

```
void NVIC_RESETPRIMASK(void);  
void NVIC_RESETFAULTMASK(void);
```

注意事项

上面两组函数要成对使用，不能交叉使用

STM32中的中断源很多，最好能够安排好分头控制，不建议这样控制。



BACK IN AN  
HOUR

# I/O口常见工作模式需求

## ❏ 单向

### ❏ 输入(悬浮、上拉、下拉)

❏ 上拉和下拉保证对方芯片I/O口高阻态时能读到稳定的电平

### ❏ 输出(推挽、开漏)

❏ 开漏输出只能输出低和释放，不能输出高

## ❏ 双向

### ❏ 总线握手

### ❏ 模拟信号输入输出

### ❏ 复用功能



# 使用I/O口需要考虑的主要问题

- ❏ 信号电平能否被双方器件正确识别
  - ❏ 理解 $V_{IH}/V_{IL}$ ,  $V_{OH}/V_{OL}$  (输入高/输入低, 输出高/输出低)
    - ❏ TTL/CMOS
  - ❏ 负载对电平的影响
    - ❏ I/O口的PMOS/NMOS的 $R_{ds(on)}$
    - ❏ 电平移位所使用的PMOS/NMOS的 $R_{ds(on)}$ 及S-D间寄生二极管压降
    - ❏ 过重负载会引起 $V_{OH}$ 下降及 $V_{OL}$ 上升
- ❏ 信号是否会使器件造成损坏
  - ❏ 施加到I/O上的电压是否大于其承受能力
  - ❏ 通过I/O口的电流或注入电流(injected current)是否大于其承受能力
- ❏ 连接方式对速率的影响
  - ❏ 上拉电阻及I/O口对地电容
- ❏ 连接方式对电平逻辑的影响
  - ❏ 某些连接方式可能伴随逻辑取反



# STM32如何连接5V器件

## ❏ 单向传输——输入

- ❏ 使用专用芯片(ST1G3234, 74LVC2t45等), 或使用分立器件
- ❏ 对于可承受5V的STM32 I/O, 可直接连接 👍

## ❏ 单向传输——输出

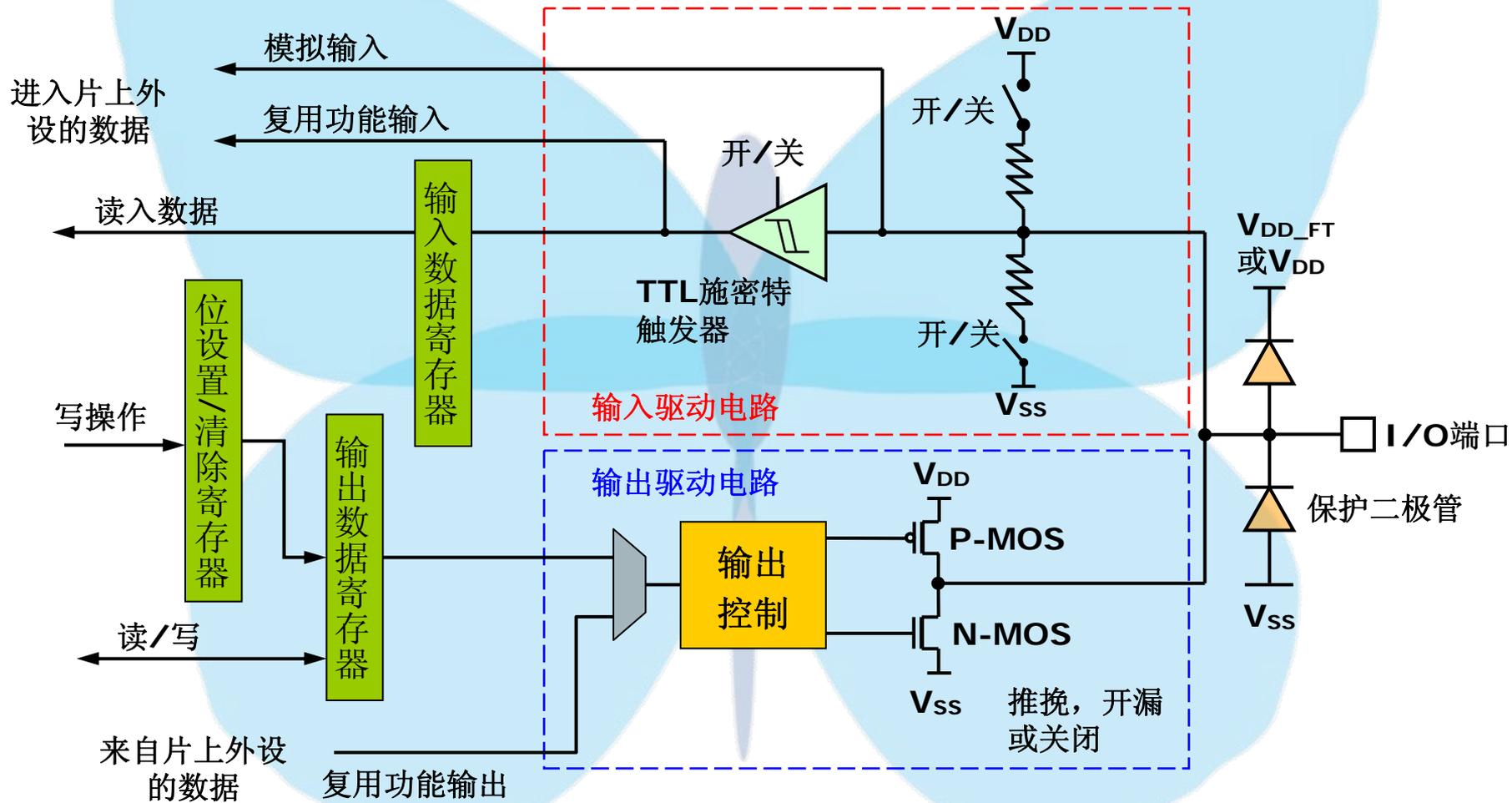
- ❏ 使用专用芯片(ST1G3234, 74LVC2t45等), 或使用分立器件
- ❏ 如对方可兼容TTL, 可考虑直接连接 👍
- ❏ 对于可承受5V的STM32 I/O, 可使用开漏输出上拉电阻连接 👍

## ❏ 双向握手传输

- ❏ 使用专用芯片 (MAX3378等), 或使用分立器件
- ❏ 对于可承受5V的STM32 I/O端口, 可使用开漏输出加上拉电阻连接。👍

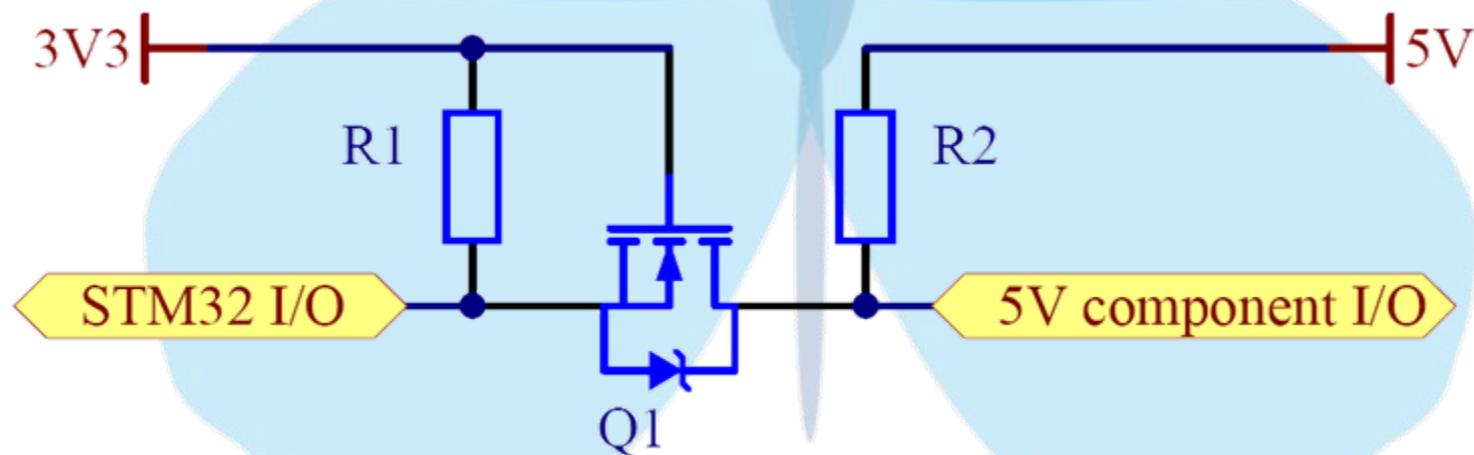


# STM32 GPIO端口框图



## 典型分立器件解决方案

- 注意电阻值对速率及功耗的影响
- 注意MOS的选择， $V_{GS(th)}$  需小于3.3V； $R_{ds(on)}$  需远小于 $R_2$ ，寄生二极管压降要足够小，保证5V I/O为低时，STM32可正确接受到逻辑‘0’



# GPIO重映射

STM32F103XX增强型  
LQFP100管脚图

TIM4\_CH1  
TIM4\_CH2  
TIM4\_CH3  
TIM4\_CH4

SPI1\_NSS  
SPI1\_SCK  
SPI1\_MISO  
SPI1\_MOSI

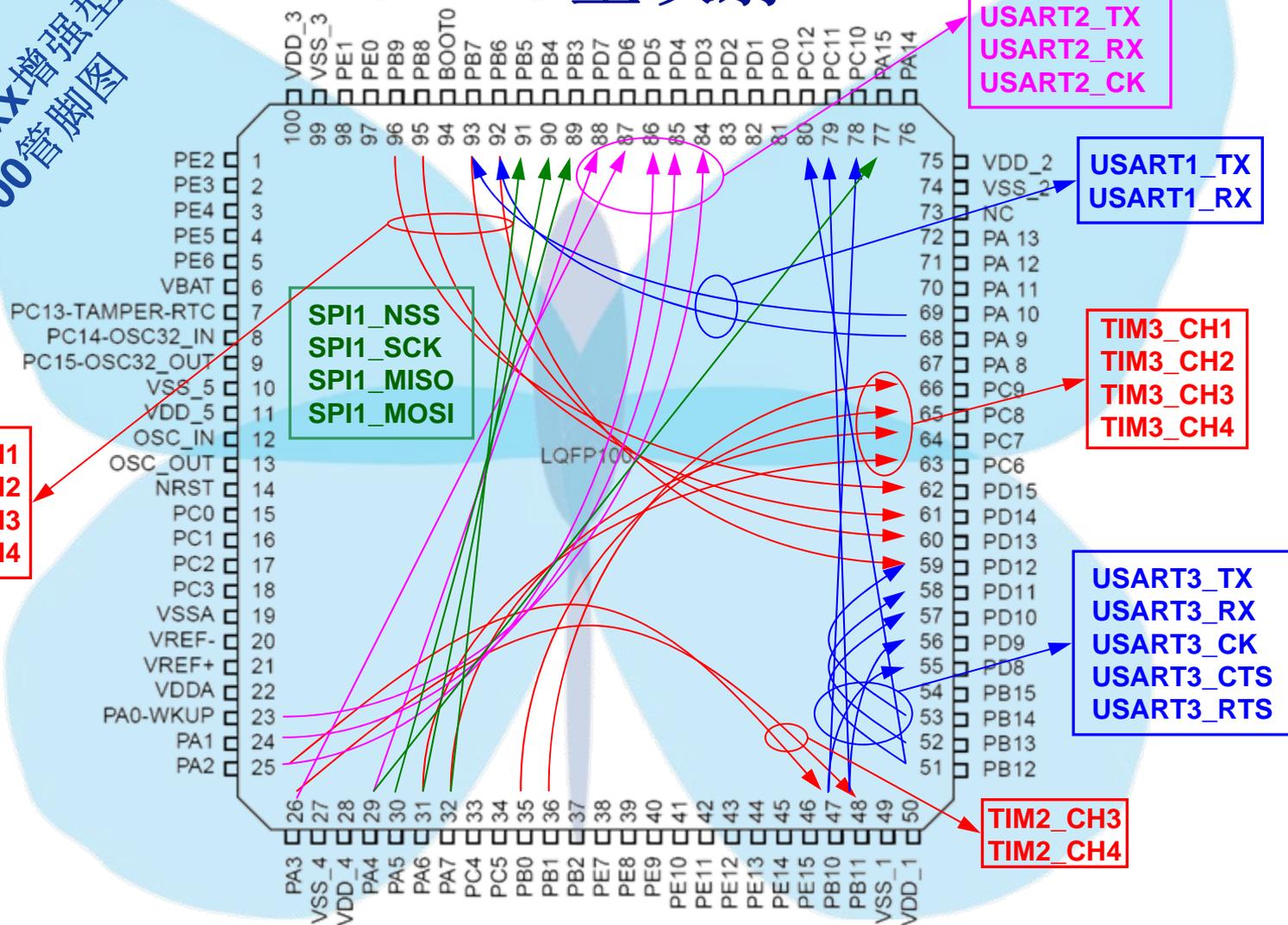
USART2\_CTS  
USART2\_RTS  
USART2\_TX  
USART2\_RX  
USART2\_CK

USART1\_TX  
USART1\_RX

TIM3\_CH1  
TIM3\_CH2  
TIM3\_CH3  
TIM3\_CH4

USART3\_TX  
USART3\_RX  
USART3\_CK  
USART3\_CTS  
USART3\_RTS

TIM2\_CH3  
TIM2\_CH4



# I/O端口的重映射

## 重映射技术的需求背景

- I/O的复用：GPIO和内置外设共用引出管脚
- I/O的重映射：复用功能(AFIO)从不同的GPIO管脚引出
- 方便了PCB的设计，潜在地减少了信号的交叉干扰
- 分时复用某些外设，虚拟地增加了端口数目

## AFIO重映射的操作步骤

1. 使能被重新映射到的I/O端口时钟
2. 使能被重新映射的外设时钟
3. 使能AFIO功能的时钟 (勿忘!)
4. 进行重映射



# 举例：重映射USART2

- ❏ USART2的TX/RX在PA.2/3
- ❏ PA.2已经被Timer2的channel3使用
- ❏ 需要把USART2的TX/RX重映射到PD.5/6
- ❏ 库函数的调用

- ❏ (1)使能被重新映射到的I/O端口时钟

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOD, ENABLE);
```

- ❏ (2)使能被重新映射的外设时钟

```
RCC_APB1PeriphClockCmd(RCC_APB1Periph_USART2, ENABLE);
```

- ❏ (3)使能AFIO功能的时钟 (勿忘!)

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO, ENABLE);
```

- ❏ (4)进行重映射

```
GPIO_PinRemapConfig(GPIO_Remap_USART2, ENABLE);
```

B9	B6	-	86	119	PD5	I/O	FT	PD5	FSMC_NWE	USART2_TX
E7	-	-	-	120	V <sub>SS_10</sub>	S				
F7	-	-	-	121	V <sub>DD_10</sub>	S				
A8	C6	-	87	122	PD6	I/O	FT	PD6	FSMC_NWAIT	USART2_RX

# JTAG管脚作为GPIO端口

- JTAG或SWD调试使用5个或2个管脚
- 在正式产品中不需要使用调试功能，因此这些管脚可以当作普通I/O端口使用

LQFP144管脚位置	105脚	109脚	110脚	133脚	134脚
调试功能	JTMS/SWDIO	JTCK/SWCLK	JTDI	JTDO	JNTRST
完全调试模式 (JTAG+SWD)	调试口	调试口	调试口	调试口	调试口
无JNTRST调试模式 (JTAG+SWD)	调试口	调试口	调试口	调试口	PB4
SWD模式	调试口	调试口	PA15	PB3	PB4
GPIO模式	PA13	PA14	PA15	PB3	PB4

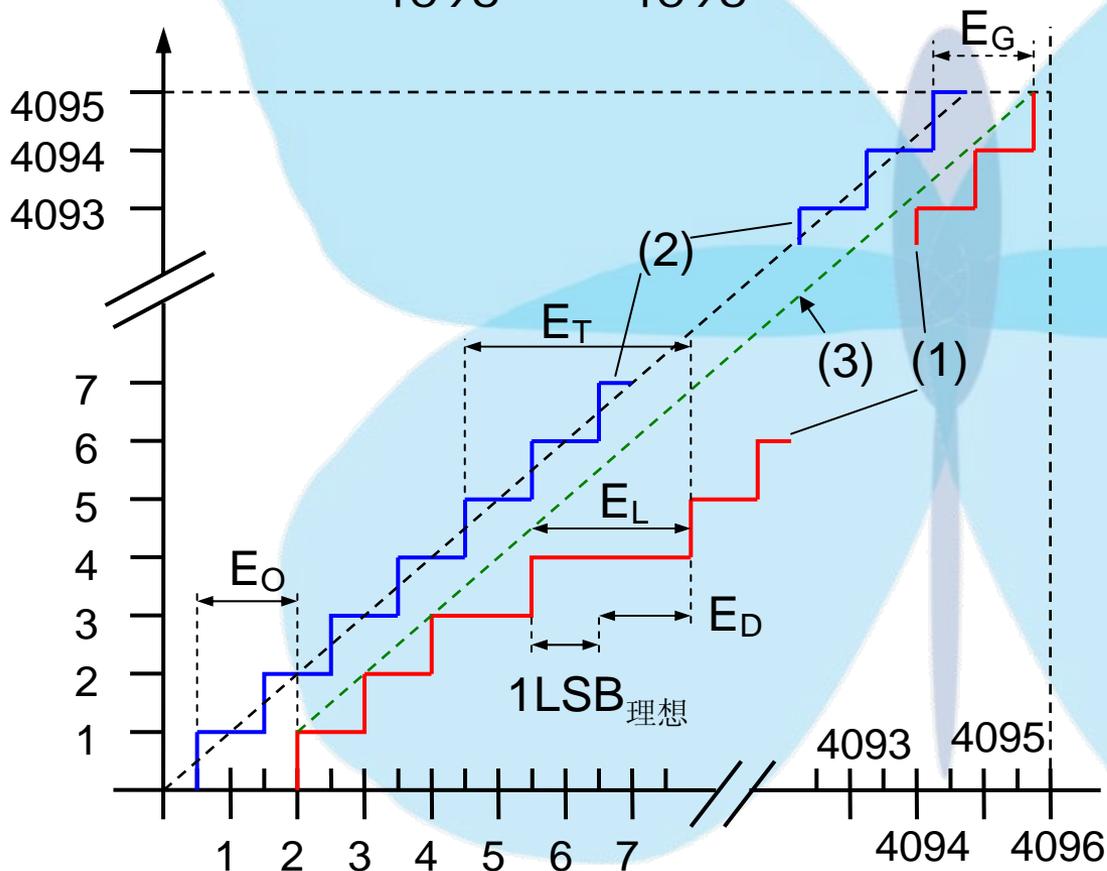
复位后的默认模式

- STM32固件库中有应用例程

STM32F10xFWLib\_V2\FWLib\examples\GPIO\JTAG\_Remap

# ADC精度参数解读

$$1\text{LSB}_{\text{理想}} = \frac{V_{\text{REF}+}}{4096} \text{ (或 } \frac{V_{\text{DDA}}}{4096} \text{ 视封装而定)}$$



- (1) 实际ADC转换曲线
- (2) 理想ADC转换曲线
- (3) 实际ADC两终点连线

$E_T$  总误差：实际ADC转换曲线与理想曲线间的最大偏离。

$E_O$  偏移误差：实际转换曲线上第一次跃迁与理想曲线中第一次跃迁之差。

$E_G$  增益误差：实际转换曲线上最后一次跃迁与理想曲线中最后一次跃迁之差。

$E_D$  微分线性误差：实际转换曲线上步距与理想步距 ( $1\text{LSB}$ ) 之差。

$E_L$  积分线性误差：实际转换曲线与终点曲线间最大偏离。

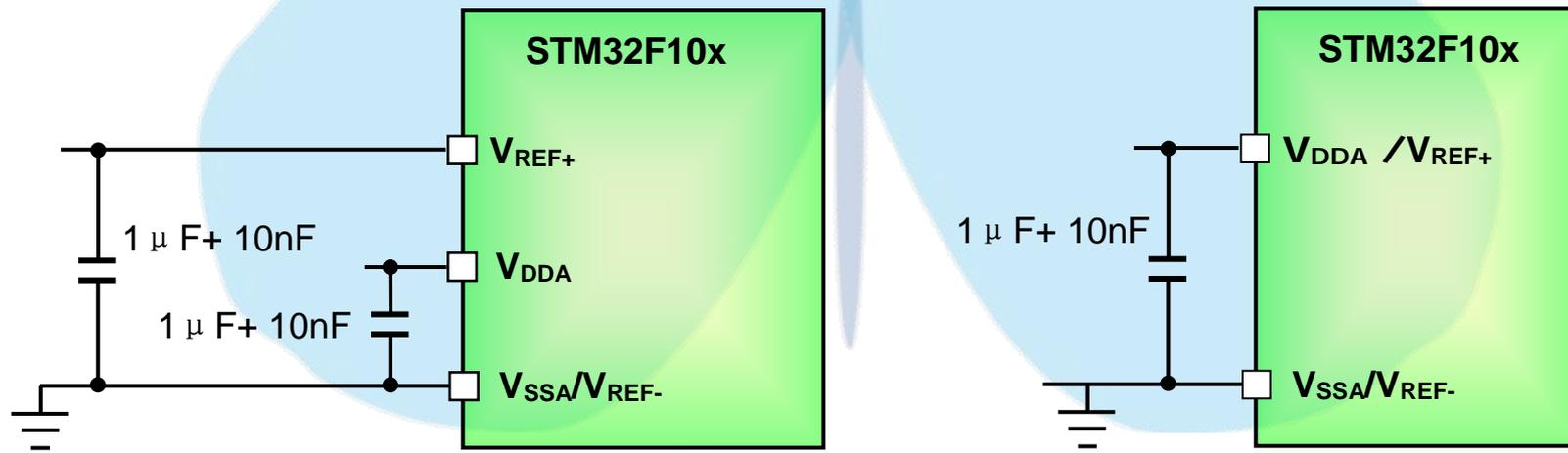
# ADC精度参数表

符号	参数	测试条件	典型值	最大值	单位
ET	总误差	$f_{PCLK2} = 56\text{MHz}$ $f_{ADC} = 14\text{MHz}, R_{AIN} < 10\text{K}\Omega$ $V_{DDA} = 3\text{V} \sim 3.6\text{V}$ $T_A = 25^\circ\text{C}$ $V_{REF+} = V_{DDA}$ 校准之后的测试结果	$\pm 1.3$	$\pm 2$	LSB
EO	偏移误差		$\pm 1$	$\pm 1.5$	
EG	增益误差		$\pm 0.5$	$\pm 1.5$	
ED	微分线性误差		$\pm 0.7$	$\pm 1$	
EL	积分线性误差		$\pm 0.8$	$\pm 1.5$	

符号	参数	测试条件	典型值	最大值	单位
ET	总误差	$f_{PCLK2} = 56\text{MHz}$ $f_{ADC} = 14\text{MHz}, R_{AIN} < 10\text{K}\Omega$ $V_{DDA} = 2.4\text{V} \sim 3.6\text{V}$ 校准之后的测试结果	$\pm 2$	$\pm 5$	LSB
EO	偏移误差		$\pm 1.5$	$\pm 2.5$	
EG	增益误差		$\pm 1.5$	$\pm 3$	
ED	微分线性误差		$\pm 1$	$\pm 2$	
EL	积分线性误差		$\pm 1.5$	$\pm 3$	

# 使用ADC的注意事项

- 外部电路的等效输入阻抗要匹配，采样速度越快，输入阻抗越小。STM32数据手册上有对照表。
- 只有在100脚和144脚的产品上才有 $V_{REF+}$ 和 $V_{REF-}$ 引出管脚。在其它产品封装中， $V_{REF+}$ 在芯片内部与 $V_{DDA}$ 相连， $V_{REF-}$ 在芯片内部与 $V_{SSA}$ 相连。
- 电路设计时，在 $V_{REF+}$ 和 $V_{DDA}$ 上一定要有良好的滤波，使用高质量的滤波电容且一定要尽量靠近芯片的管脚。



# ADC 注入模式的用法与常见应用场合（1）

## ADC转换通道编组

- 常规转换组：最大16个通道

- 注入转换组：最大4个通道

- 每个注入通道的转换值，存储在各自相应的寄存器中，即：注入转换组有4个转换值寄存器；而常规转换组只有1个转换值寄存器。

## 多触发源

- 每个组可被来自定时器的6个事件触发

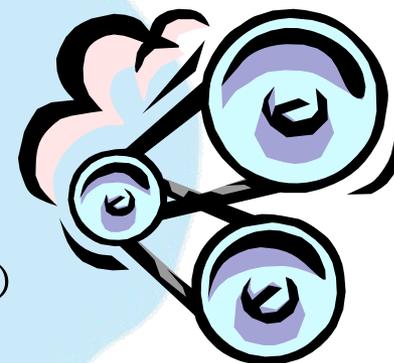
- 可由外部事件和软件触发

## 基于定时器的扫描模式：

- 任意通道，任意次序

- 最大4个通道的注入转换（结果由相应寄存器存储）

## 中断和DMA



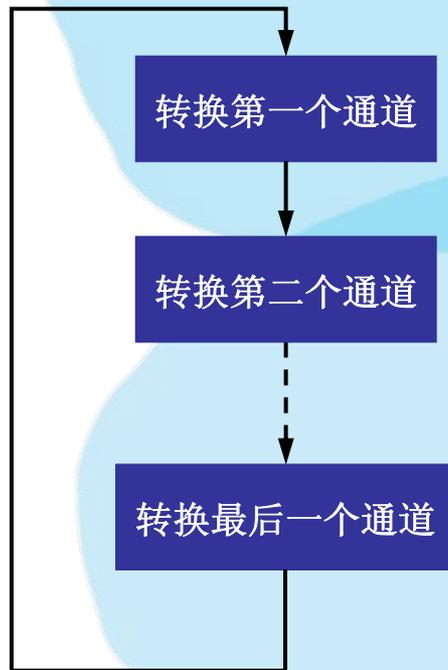
## ADC 注入模式的用法与常见应用场合（2）

- 转换数据可向左或向右对齐
- 4个偏移补偿寄存器
  - 补偿外部电路的偏移，如运放。如需要可提供带符号值
- 每个通道可单独编程采样时间，可以采样不同输入阻抗的信号
  - 从 $1.5cy$  ( $R_{in} < 1.2K$ ) 到  $239.5cy$  ( $R_{in} < 350K$ )，共8个值
  - 当采样率为 $1MSps$ 时，可不用电压跟随器
- ADC双模式
  - 只能在拥有2个ADC的MCU中实现

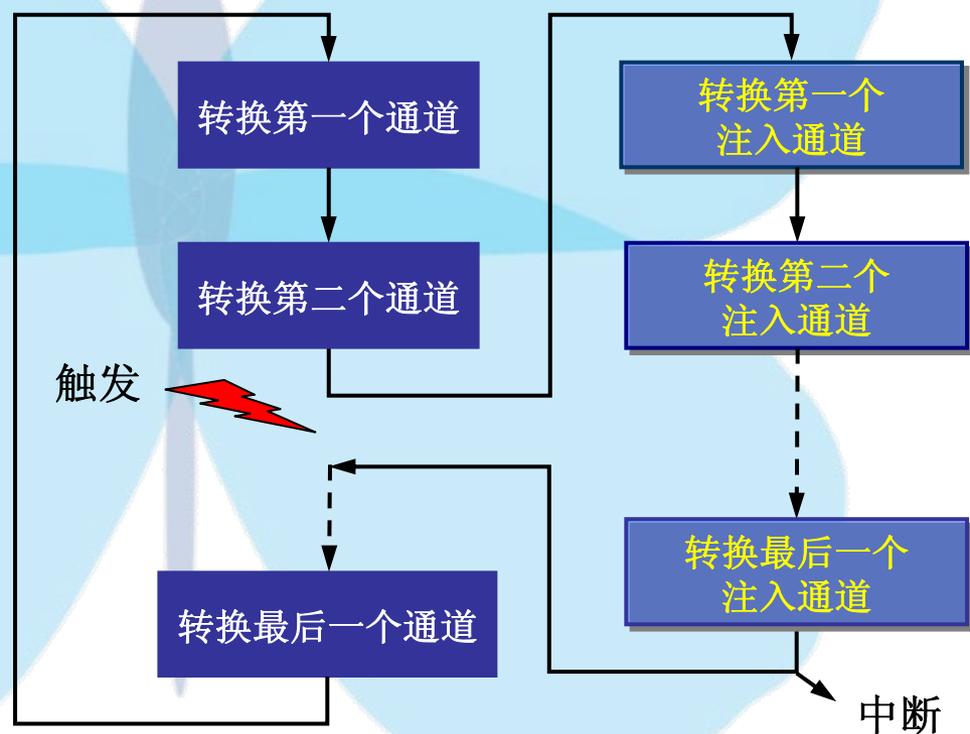


# ADC 注入模式的用法与常见应用场合 (3)

## 常规转换扫描模式



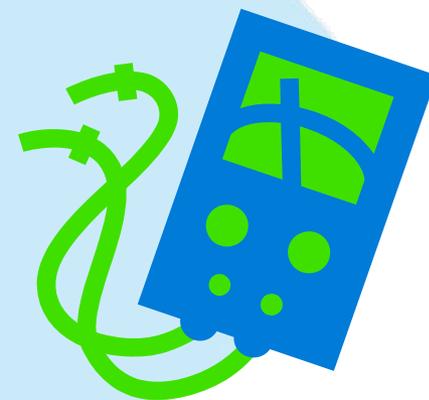
## 注入转换扫描模式



## ADC 注入模式的用法与常见应用场合（4）

- 应用场合举例 ---- 马达矢量控制中实现相电流同时采样
- PWM定时器中的同步单元可实现ADC同步
- 可有2个选择：
  - 直接由PWM定时器计数器的峰顶、谷底或两者中的任一个同步
  - 由PWM定时器的第4个输出比较所产生的延时同步
- ADC的结果可由“转换完成”中断处理或由DMA存储
- PWM定时器的事件触发两个ADC，使其工作在“同时注入”双模式下
  - 由于同时采样2相电流，不会产生误差

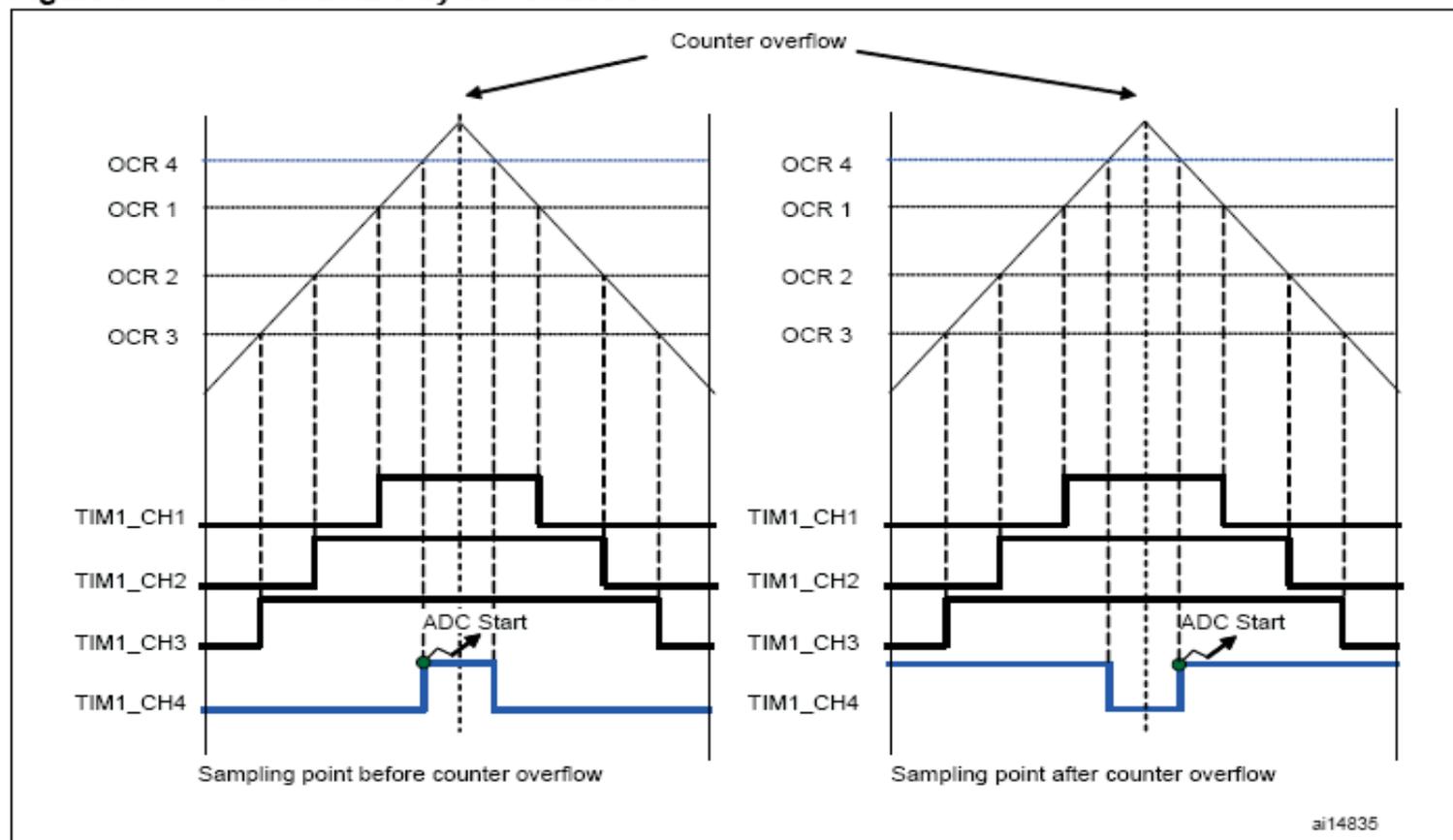
新型号又增加了一个ADC模块



# ADC 注入模式的用法与常见应用场合 (5)

由TIM1的CH4输出进行同时采样

Figure 34. PWM and ADC synchronization



# 窗口型看门狗



一个可编程的递减计数器

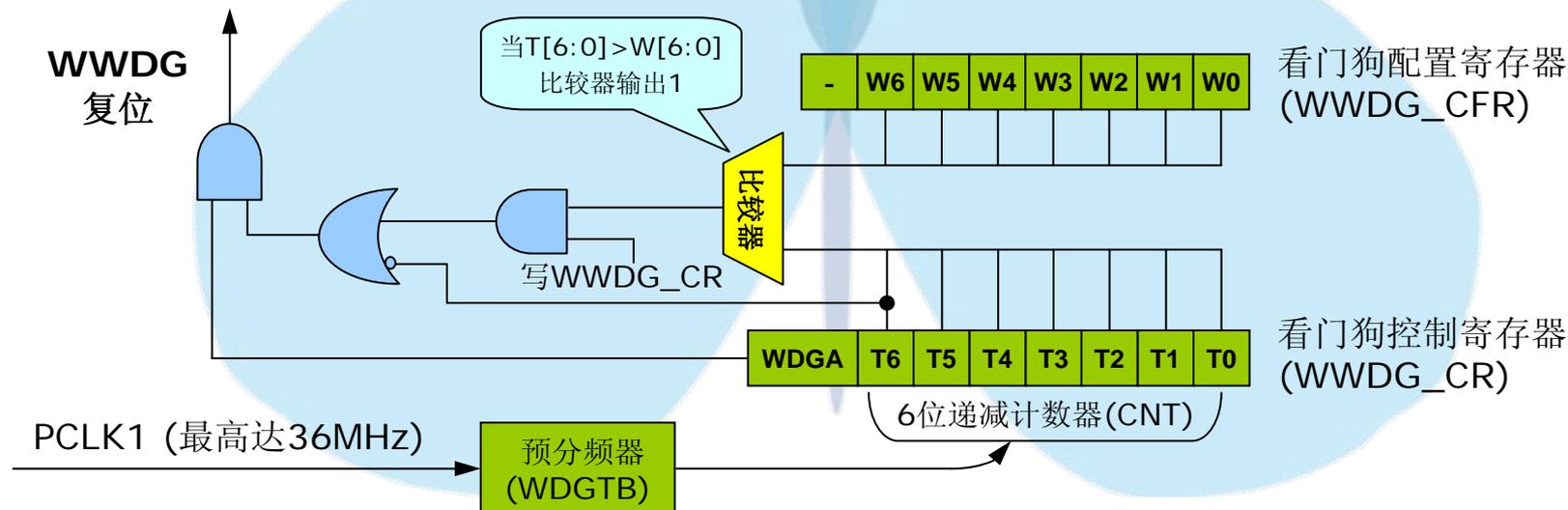
触发芯片复位的条件：

超过一定时间未对递减计数器更新——俗称喂狗

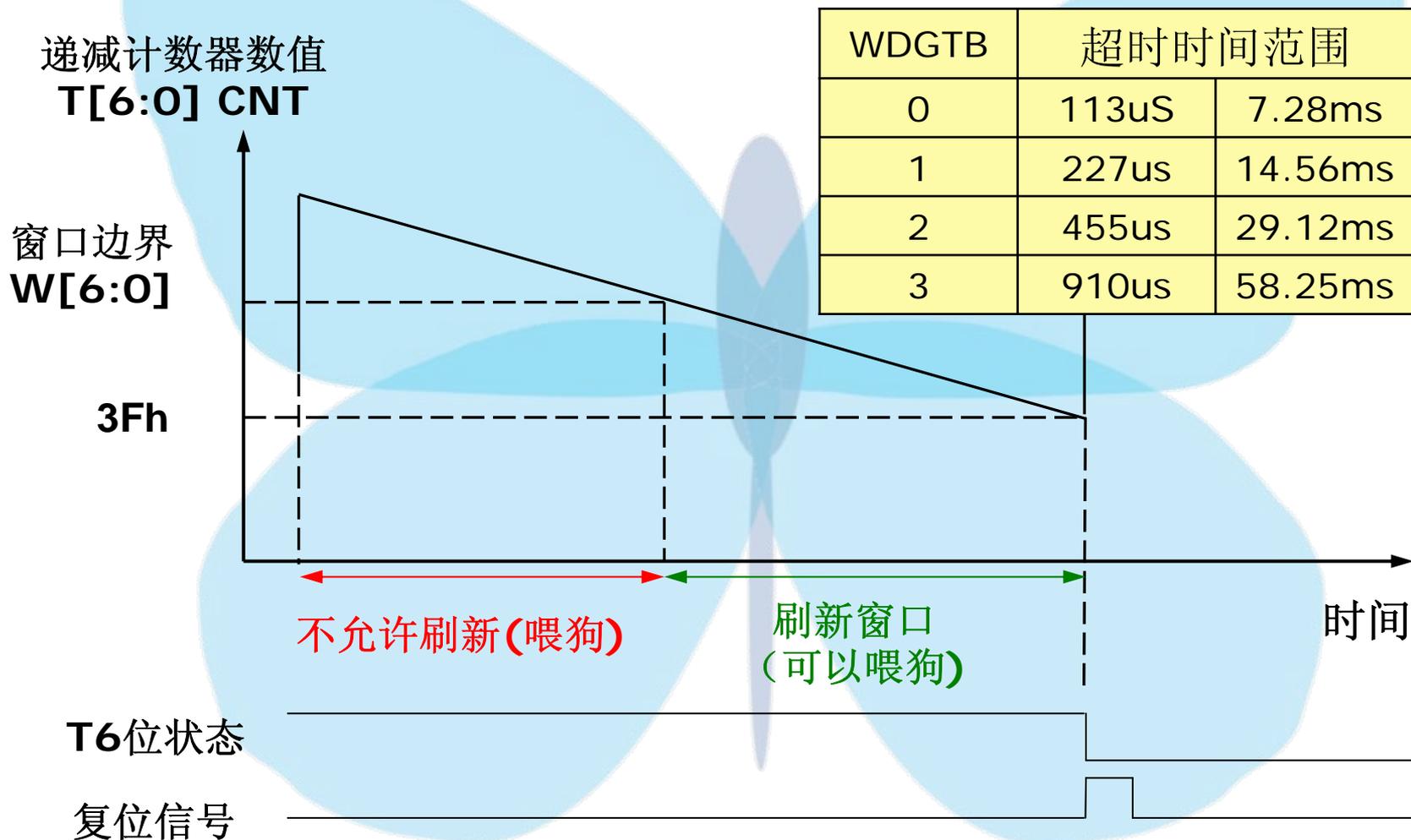
未在规定的时间内更新——喂狗太早，狗还没饿

看门狗复位预警中断

让应用程序在芯片复位前有机会更新递减计数器(喂狗)

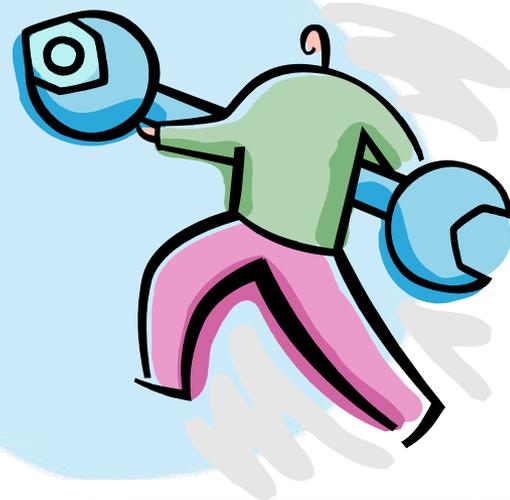


# 窗口型看门狗工作原理



# 调试状态下的操作模式

- ❏ 在调试实际应用程序时，经常会要求程序遇到断点暂停时，某些外设不能停止，如控制电机的定时器、CAN通信等；有时又需要某些外设能够停下来，如看门狗的计数器。
- ❏ 根据这些要求，STM32设置了调试配置寄存器，可以由用户通过程序指定是否需要在调试暂停时停止某些功能模块。
- ❏ 下列模块在调试暂停时的行为可以由调试配置寄存器控制
  - ❏ 低功耗模式(STANDBY、STOP)
    - ❏ 选项为：使用HSI或停止时钟
  - ❏ 定时器和看门狗的计数器
    - ❏ 选项为：停止计数器或正常计数
  - ❏ bxCAN通讯
    - ❏ 选项为：停止接收器或正常接收
  - ❏ SMBUS超时模式
    - ❏ 选项为：停止超时计时或继续超时计时



# 调试时经常碰到的问题

❏ 在调试那些可以在CPU不干预的时候自动运行的模块时，或在调试低功耗模式的程序时，往往碰到重新下载程序失败、调试器不能停止CPU运行等问题。

❏ 例如：定时器，DMA模块，ADC的连续转换模式等

❏ 这个问题的根源是：

❏ 调试器需要在RAM执行一段程序，对Flash进行擦写操作，如果不停止这些自动运行的模块，它们会干扰程序在RAM中的执行，致使下载失败。

❏ 低功耗是通过停止CPU的时钟而实现，JTAG调试是通过与CPU的通信实现，停止了CPU的时钟致使调试器会失去与CPU的通信。

❏ 解决办法：

1. 退出调试状态时或main()开始时执行DeInit()，或
2. 下载程序或进入调试前，手工做硬件复位，或
3. 进入main()后，推迟启动自动运行的模块，或
4. 进入main()后，通过外设复位寄存器执行外设复位并推迟启动相应外设。



# 程序从Flash启动改到从SRAM启动

调试代码时，每次修改都需要重新烧写FLASH。  
需要花费多少时间？FLASH可以满足多少次的烧写？



STM32F10xxx系列  
提供6K至64K不等的SRAM

通过改变BOOT引脚可以  
选择芯片从Flash启动，  
从SRAM启动，或从  
System Memory启动。

选择芯片  
从SRAM启动

# 程序从Flash中启动改到从SRAM中启动 步骤

1、将目标板上的BOOT引脚改成从SRAM启动，即  
**BOOT0=1， BOOT1=1。**

2、如果使用ST提供的库函数，可以通过调用以下两个函数切换中断向量表的指向。

```
NVIC_SetVectorTable(NVIC_VectTab_FLASH, 0x00);  
NVIC_SetVectorTable(NVIC_VectTab_RAM, 0x0);
```

3、如果不使用ST提供的库函数，可以通过直接改写Cortex-M3内部的Vector Table Offset (向量表偏移)寄存器来修改中断向量表的地址。向量表偏移寄存器的地址为0xE000ED08。

(详细信息请参考Cortex-M3 Technical Reference Manual)

# 程序从Flash中启动改到从SRAM中启动 步骤

4、根据使用的编译器的不同，修改Link文件。

如使用IAR，可对Inkarm\_flash.xcl文件进行如下修改：

-DROMSTART=0x8000000  
-DROMEND=0x801FFFF



-DROMSTART=0x20000000  
-DROMEND=0x20005000

5、修改完毕后，启动调试工具，烧录代码至SRAM中，芯片即可从SRAM中启动运行。



# Unique Device ID介绍

- ❑ STM32F10xxx系列为每块芯片都提供一个96位的独一无二的ID供用户控制使用。
- ❑ 独特ID可以做为用户最终产品的序列号，帮助用户进行产品的管理。
- ❑ 在某些需要保证安全性的功能代码运行前，通过校验此独特ID，保证最终产品的某些功能的安全性。
- ❑ 独特ID配合加解密算法，对芯片内部的代码进行加解密，以保证用户产品的安全性和不可复制性。



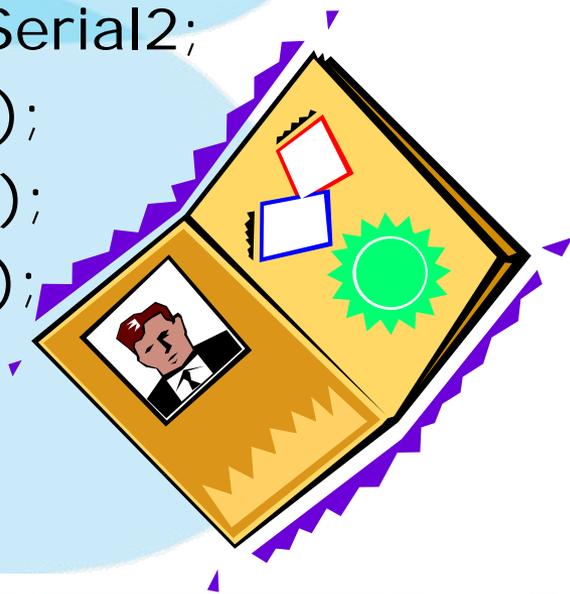
# Unique Device ID访问

96位的独特ID位于地址0x1FFFF7E8 ~ 0x1FFFF7F4的系统存储区，由ST在工厂中写入(用户不能修改)，用户可以以字节、半字、或字的方式单独读取其间的任一地址。

(具体请参考Reference Manual V4.0版本的25.2章)

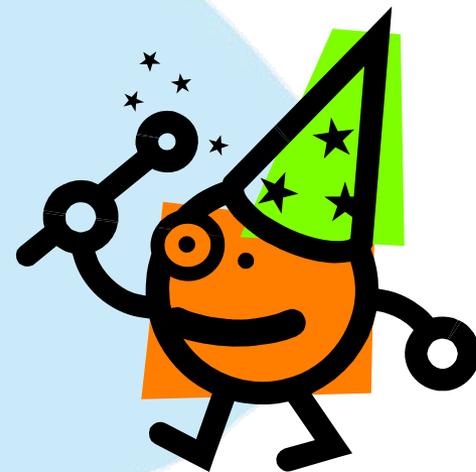
例程：

```
u32 Dev_Serial0, Dev_Serial1, Dev_Serial2;  
Dev_Serial0 = *(vu32*)(0x1FFFF7E8);  
Dev_Serial1 = *(vu32*)(0x1FFFF7EC);  
Dev_Serial2 = *(vu32*)(0x1FFFF7F0);
```



# BIT-BAND介绍

- ❏ Bit-band操作作为Cortex-M3提供的特殊操作，Bit-band区域的每个字都会有普通区域的某个字节的某个比特位与之对应。
- ❏ 对Bit-band区域某个字节的写操作，Cortex-M3都将自动转换成对相对应比特位的读—修改—写操作。对Bit-band区域某个字节的读操作则将转换成相对应比特位的读操作。
- ❏ 与原有的读字节—修改相应比特位—写字节的操作方式相比，Bit-band虽然不能减少操作时间，却能简化操作减小代码量，并防止错误的写入。
- ❏ Bit-band区域的存储器以32位的方式进行访问，但其中有效的仅仅是BIT0位，只有BIT0位的值才对应到相应的普通区域的比特位上，其他位无效。
- ❏ STM32F10xxx系列芯片为所有的外设寄存器和SRAM提供相对应的Bit-band区域，以简化对外设寄存器和SRAM的操作。



# BIT-BAND: 寻址

Bit-band操作中最重要的一环就是寻址，即为需要操作的外设寄存器或者SRAM中的变量找到相对应的Bit-band区域地址。

Bit-band地址的计算公式为：

$$\text{Bit-Band地址} = \text{Bit-Band域首地址} + (\text{操作字节的偏移量} \times 32) + (\text{操作位的偏移量} \times 4)$$

内置SRAM区的Bit-Band域首地址为0x22000000  
外设寄存器区的Bit-Band域首地址为0x42000000

例如：

1，在SRAM的0x20004000地址定义个长度为512字节的数组。

```
#pragma location=0x20004000
__root __no_init u8 Buffer[512];
```

该数组首字节的BIT0对应的Bit-band地址为：

$$0x22000000 + (0x4000 \times 32) + (0 \times 4) = 0x22080000$$

该数组第二个字节的BIT3对应的Bit-band地址为：

$$0x22000000 + (0x4001 \times 32) + (3 \times 4) = 0x2208002C$$

2，GPIOA的端口输出数据寄存器位于地址0x4001080C，对于GPIOA的PIN0来说，控制其输出电平的比特位的Bit-band地址为：

$$0x42000000 + (0x1080C \times 32) + (0 \times 4) = 0x42021018$$

# BIT-BAND: 使用

合理的使用Bit-band功能，将大大简化操作。

例1：将前页数组中的数据通过GPIOA的PIN0口输出：

❏ 不使用Bit-band功能：

```
for(u16 cnt=0; cnt<512; cnt++)
    for(u8 num=0; num<8; num++)
        if ((Buffer[cnt] >> num) & 0x01)
            GPIOA->BSRR = 1;
        else
            GPIOA->BRR = 1;
}
```

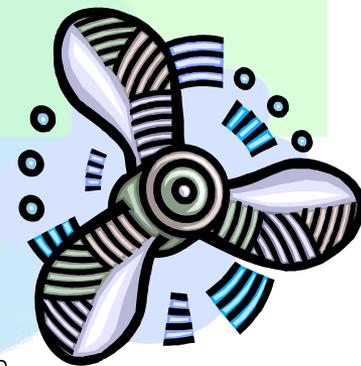
❏ 使用Bit-band功能：

```
u32 *pBuffer = ((u32*)0x22080000);
u16 cnt = 512 * 8;

While (cnt--){
    (*(u32*) 0x42021018) =
        *pBuffer++;
}
```

# Have fun!

可见，使用了Bit-band功能，运算量和代码量都大大减少。



# BIT-BAND: 使用

合理的使用Bit-band功能，将大大简化操作。

例2：修改SPI控制寄存器的BIT6，使能SPI功能：

 不使用Bit-band功能：

```
u32 spi_ctrl = SPI1->CR1;  
spi_ctrl = spi_ctrl |  
    0x00000040;  
SPI1->CR1 = spi_ctrl;
```

 使用Bit-band功能：

SPI1的CR1寄存器地址为0x40013000，因此，CR1寄存器BIT6位的Bit-band地址为：  
 $0x42000000 + 0x13000 \times 32 + 6 \times 4$   
 $= 0x42260018$

对BIT6的置位操作为：

```
(*((u32*) 0x42260018)) = 1;
```

对Bit-band区域的写操作Cortex-M3内核最终仍然执行读—修改—写的过程，但代码量得到简化，并能防止错误的写入。

# STM32进入ISR的CPU周期(1/3)

❏ 最短6个周期，最长视情况而定

❏ 影响因素

❏ 将寄存器{PC,R0-R3,R12,R14, xPSR}内容入栈需要12个周期（如果需要），这个过程由内核自动完成

❏ 该中断产生时是否有相同或更高优先级的中断服务程序在执行

❏ 正在执行的指令

内核需要执行完当前运行的指令然后转向中断处理，大部分指令都不是单周期指令，所以在进入中断处理时需要增加额外的周期；某些指令可以被中断

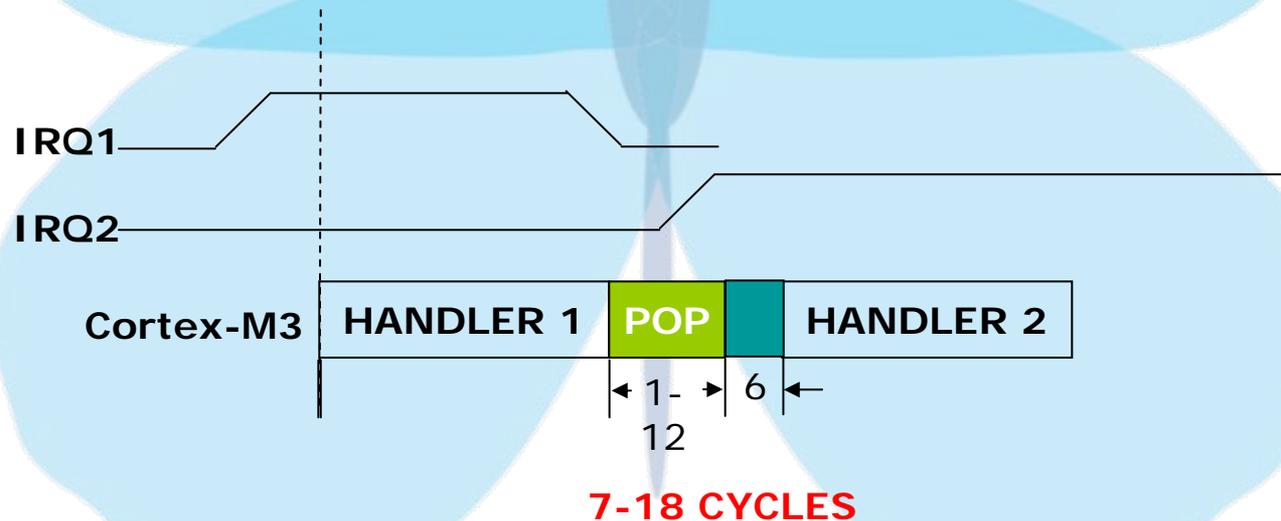
❏ 程序的位置以及是否有DMA传输在进行

DMA传输如果与内核同时访问FLASH、memory或者外设时，内核可能会被阻断若干个周期（但总线管理至少会保证CPU占有一半的带宽，DMA在每个数据传输之后会空出一个周期来保证CPU对总线的访问）

## STM32进入ISR的CPU周期(2/3)

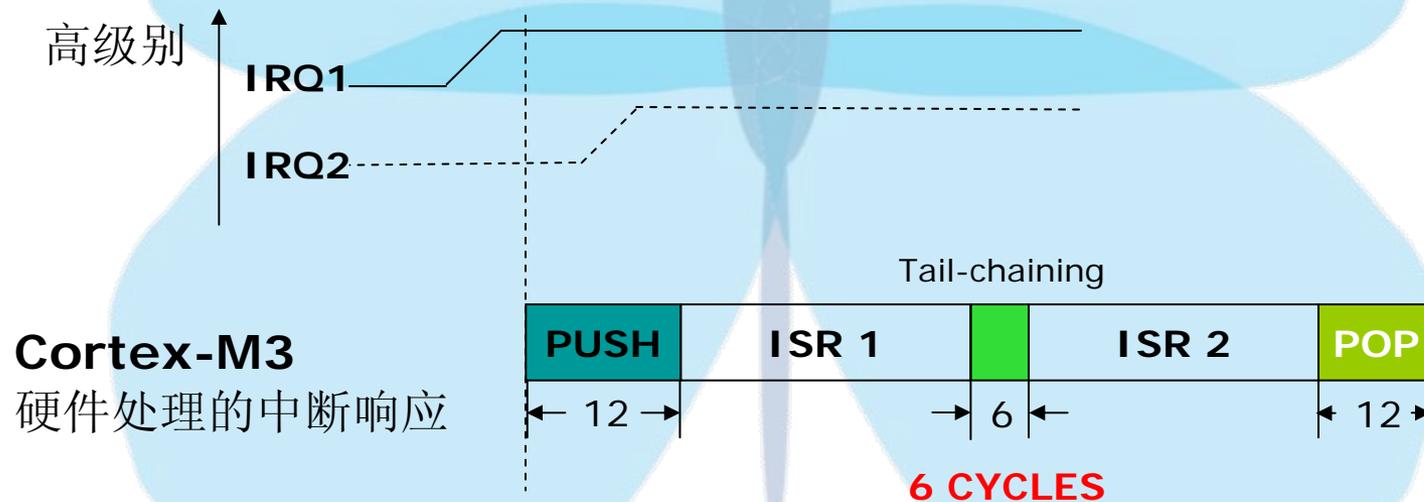
进入中断处理最小的情况（6个CPU周期）

如果前一个ISR正在状态恢复（出栈）的过程中产生了另一个中断，内核会放弃已取出的内容，且不修改栈指针，直接转向中断处理程序，这个时候只需要完成中断服务程序的预取指，仅需要6个CPU周期即可进入中断处理程序。



# STM32进入ISR的CPU周期(3/3)

- 如果中断(ISR2)产生时有相同或更高优先级的中断(ISR1)正在运行，那么内核将会把ISR2设为等待，直到ISR1执行结束，但不再进行出栈入栈操作，直接转向中断处理程序进行预取指。



# STM32的三种低功耗模式

模式	功能	适用性
睡眠模式 <b>Sleep</b>	电压调节器开启，Cortex-M3内核停止运行，外设保持运行态。	适用于等待外设的中断或事件时，降低系统的功耗。
停止模式 <b>Stop</b>	电压调节器可选择性开启，所有外设时钟、PLL、HSI和HSE被关闭，Cortex-M3内核和所有外设停止运行，保留SRAM和寄存器的内容。	适用于等待只有外部中断(EXIT)时，降低系统的功耗。
待机模式 <b>Standby</b>	电压调节器关闭、整个1.8v区域断电。除了备份区域和待机电路的寄存器以外，SRAM和寄存器的内容全部丢失。	

## 降低功耗的基本原则：

- 降低系统时钟；
- 关闭APB和AHB总线上未使用外设的时钟。

# 睡眠模式

- 通过执行指令(WFI/WFE)进入该模式；
- Cortex-M3控制寄存器中的SLEEPONEXIT位决定进入该模式的机制：
  - Sleep-Now**(SLEEPONEXIT=0)：当执行WFI/WFE指令时，MCU立即进入睡眠模式。
  - Sleep-on-Exit**(SLEEPONEXIT=1)：当从优先级最低的中断处理中退出时，MCU进入睡眠模式。
- 退出睡眠模式：
  - WFI**(等待中断)：
    - 任意可被矢量嵌套中断控制器(NVIC)识别的外设中断。
  - WFE**(等待事件)
    - 在事件模式下，一个在外设控制寄存器中可设置为中断使能的事件，或设为事件模式的外部中断线事件。
    - 进入/退出无时间损失。

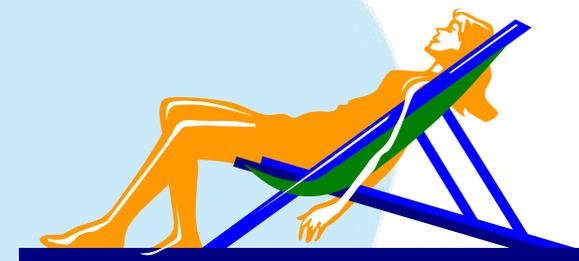
为了更多地降低功耗，用户可以在进入睡眠模式前关闭不工作的外设时钟，以此节省功耗。

# 睡眠模式应用举例

进入睡眠模式的代码：

```
void PWR_EnterSLEEPMode(u32 SysCtrl_Set, u8 PWR_SLEEPEntry)
{
    if (SysCtrl_Set) // 对Cortex-M3中系统控制寄存器的SLEEPONEXIT位置位
        *(vu32 *) SCB_SysCtrl |= SysCtrl_SLEEPONEXIT_Set;
    else // 对Cortex-M3中系统控制寄存器的SLEEPONEXIT位复位
        *(vu32 *) SCB_SysCtrl &= ~SysCtrl_SLEEPONEXIT_Set;

    // 对Cortex-M3中系统控制寄存器的SLEEPDEEP位复位
    *(vu32 *) SCB_SysCtrl &= ~SysCtrl_SLEEPDEEP_Set;
    if (PWR_SLEEPEntry == PWR_SLEEPEntry_WFI)
        __WFI(); // 等待中断请求
    else
        __WFE(); // 等待事件请求
}
```



注：#define SysCtrl\_SLEEPONEXIT\_Set ((u32)0x00000002)

# 停止模式

❏ 在以下条件下执行WFI或WFE指令进入停止模式：



1. 设置Cortex-M3系统控制寄存器中的SLEEPDEEP位；
2. 清除电源控制寄存器(PWR\_CR)中的PDDS位(可选项)；
3. 通过设置PWR\_CR中LPDS位选择电压调节器的模式。

❏ 如果RTC和IWDG正在运行，进入停止模式时该外设不会停止(包括时钟)。

❏ 为了降低更多的功耗，可以清除电源控制寄存器(PWR\_CR)中的PDDS位将电压调节器置为低功耗模式；

❏ 退出停止模式：

❏ 以WFI指令进入时：任意设置为中断模式的外部中断线；

❏ 以WFE指令进入时：任意被设置为事件模式的外部中断线；

➔ 从停止模式恢复后，时钟的配置返回到复位时的状态(即系统时钟为HSI)。

# 停止模式应用举例

## 进入停止模式的代码：

```
void PWR_EnterSTOPMode(u32 PWR_Regulator, u8 PWR_STOPEntry)
{
    PWR->CR &= CR_DS_Mask; // 清除 PDDS 和 LPDS 位
    PWR->CR |= PWR_Regulator; // 根据PWR_Regulator的值设置LPDS位
    *(vu32 *) SCB_SysCtrl |= SysCtrl_SLEEPDEEP_Set; // 置位 SLEEPDEEP 位
    // 选择进入STOP模式的入口
    if (PWR_STOPEntry == PWR_STOPEntry_WFI)
        __WFI(); // 请求等待中断
    else
        __WFE(); //请求等待事件
}
```

## 在实际应用中：

```
// 进入STOP模式
PWR_EnterSTOPMode(PWR_Regulator_ON, PWR_STOPEntry_WFI);
RCC_Configuration(); // 唤醒后重新进行时钟配置
```

注：#define SysCtrl\_SLEEPDEEP\_Set ((u32)0x00000004)

**特别注意：PWR的时钟要使能！**



# 待机模式

❏ 在以下条件下执行WFI或WFE指令：

1. 设置Cortex-M3系统控制寄存器中的SLEEPDEEP位；
2. 设置电源控制寄存器(PWR\_CR)中的PDDS位；
3. 清除电源控制/状态寄存器(PWR\_CSR)中的WUF位。

❏ 在待机模式下，RTC和IWDG（如果开启）继续运行；

❏ 在待机模式下，所有IO口引脚处于高阻状态，除了以下一些情况：

- ❏ 复位引脚(始终有效)；
- ❏ 防侵入引脚，如果被设置为检测侵入或校准；
- ❏ 唤醒引脚，如果使能。

❏ 唤醒源：

- ❏ 唤醒引脚的上升沿
- ❏ RTC警告
- ❏ 复位引脚外部复位信号
- ❏ IWDG复位



# 待机模式应用举例

进入停止模式的代码：

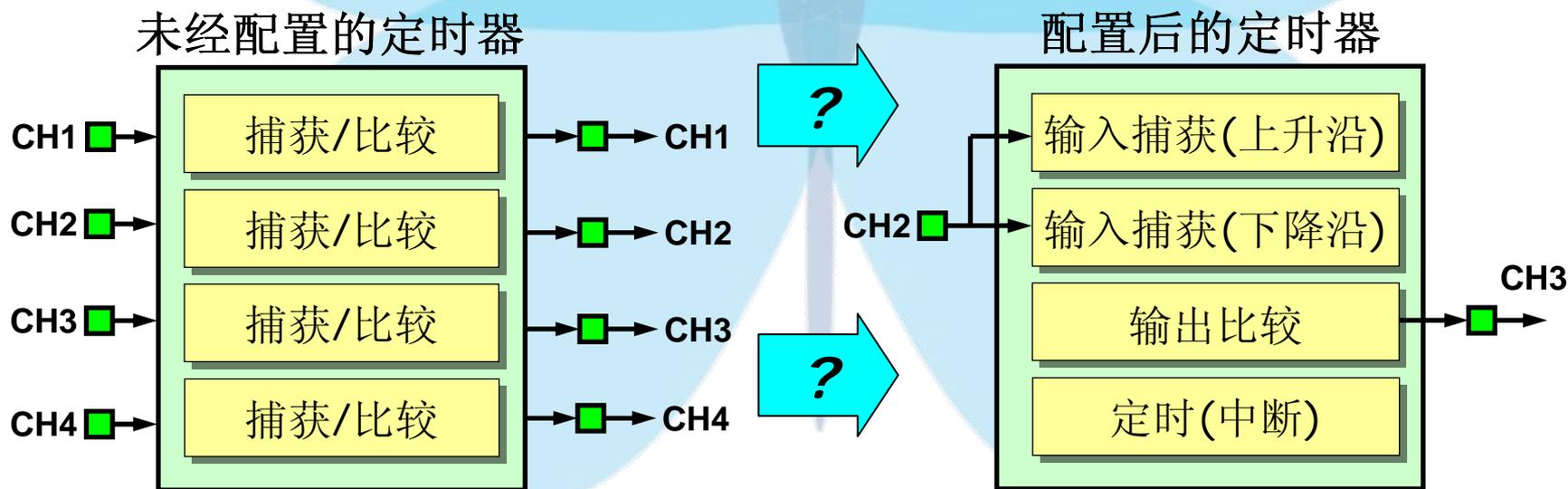
```
void PWR_EnterSTANDBYMode(void)
{
    PWR->CR |= CR_CWUF_Set; // 清除 Wake-up 标志
    PWR->CR |= CR_PDDS_Set; // 选择进入 STANDBY 模式
    // 置位 SLEEPDEEP 位
    *(vu32 *) SCB_SysCtrl |= SysCtrl_SLEEPDEEP_Set;
    __WFI(); // 等待中断请求
}
```

用户只需在程序中调用该函数即可。同时还有设定唤醒源。

用户使用 **IWDG**，在进入待机模式时要考虑 **IWDG** 的计数长度，同时在程序启动时需要判断复位源。

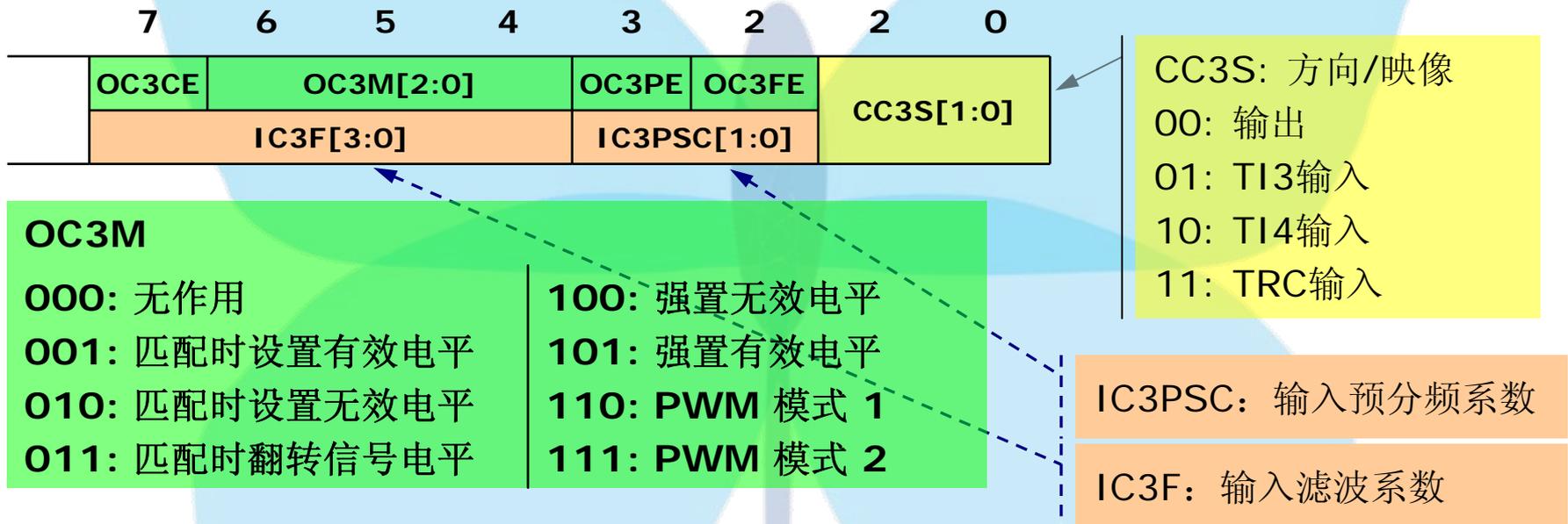
# 如何在一个定时器上同时使用输入/输出

- STM32的普通定时器有多达4个通道，每个通道都可以配置为输出比较、输入捕获等多种模式。
- 下面以一个例子说明如何在一个定时器上同时使用输入/输出功能：
  - 通道1、2配置为输入捕获模式
  - 通道3配置为输出比较模式
  - 通道4用于产生定时的中断，无输入/输出管脚配置



# 配置定时器同时使用输入/输出

使用TIMx\_CCMR1和TIMx\_CCMR2配置输入输出方向和模式



TIMx\_CCER用于使能通道(CCxE)并选择极性(CCxP)

CC3E选择输入捕获或输出比较

CC3P选择极性

# 定时器的参数选择

例：

- ▣ 通道1、2作为输入捕获
- ▣ 通道3使用输出翻转模式
- ▣ 通道4作为中断定时

▣ 配置代码(假定使用TIM4)

```
TIM4_CCMR1 = 0x0102;
TIM4_CCMR2 = 0x0030;
TIM4_CCER = 0x0131;
```



通道 1 = 0x02

- 输入捕获
- 与TI2相连 (CH2输入)

通道 2 = 0x01

- 输入捕获
- 与TI2相连 (CH2输入)

通道 3 = 0x30

- 输出比较
- 翻转模式

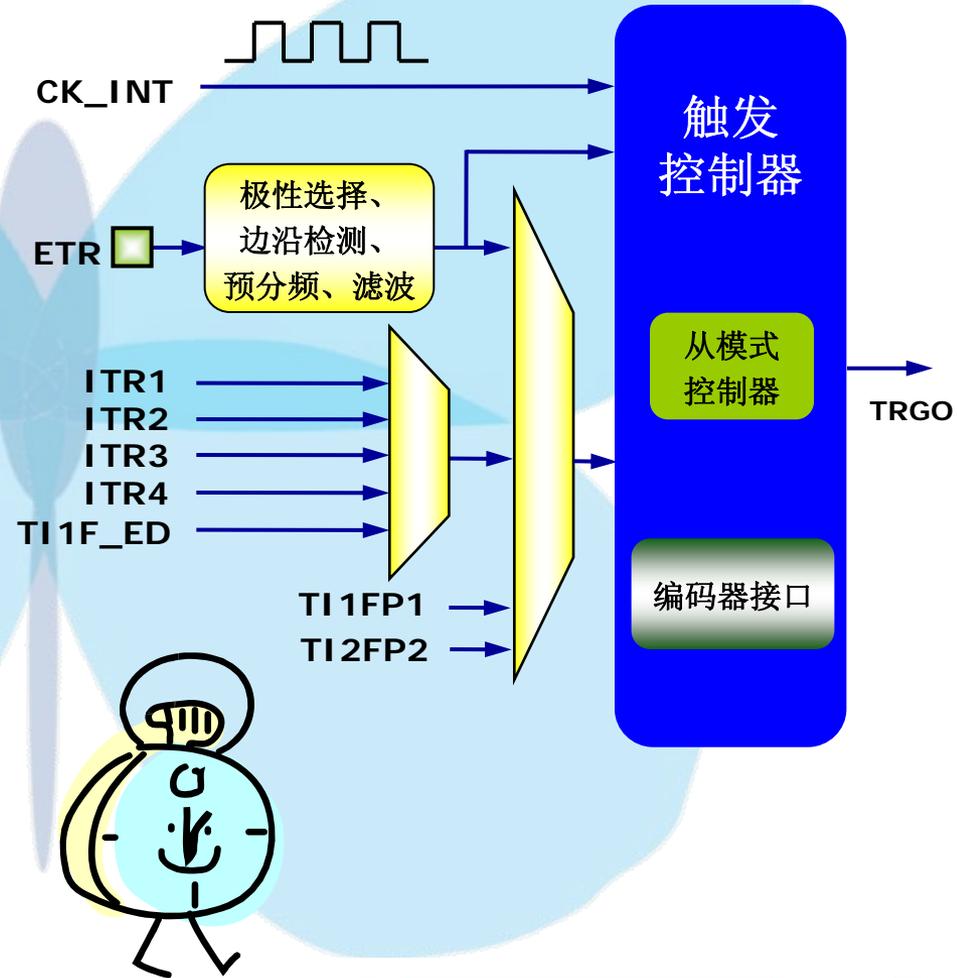
通道 4 = 0x00

- 输出比较
- 无输出

通道 1 = 0x1	使能	上升沿
通道 2 = 0x3	使能	下降沿
通道 3 = 0x1	使能	有效电平为高
通道 4 = 0x0	禁止	有效电平为高

# TimerX 的时钟源——基本结构

- 计数器时钟可由下列时钟源提供
  - 内部时钟(CK\_INT)
  - 外部时钟源模式1：外部比较捕获引脚的上升沿、下降沿或者边沿信号
    - TI1FP1 或 TI1F\_ED
    - TI2FP2
  - 外部时钟源模式2：外部引脚触发输入ETR
  - 内部触发输入 (ITRx)：使用一个定时器作为另一个定时器的预分频器，例如，你可以配置一个定时器Timer1 而作为另一个定时器Timer2 的预分频器。



# TimerX 的时钟源—寄存器配置

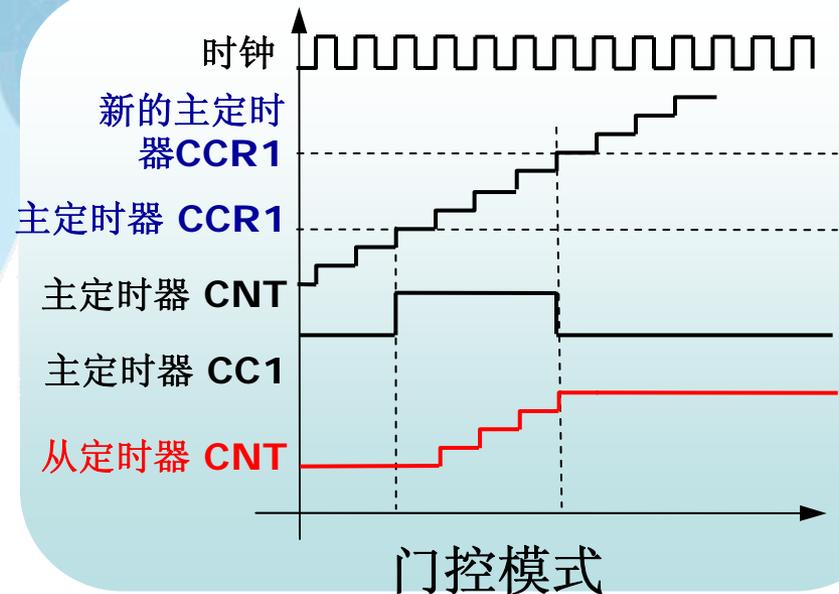
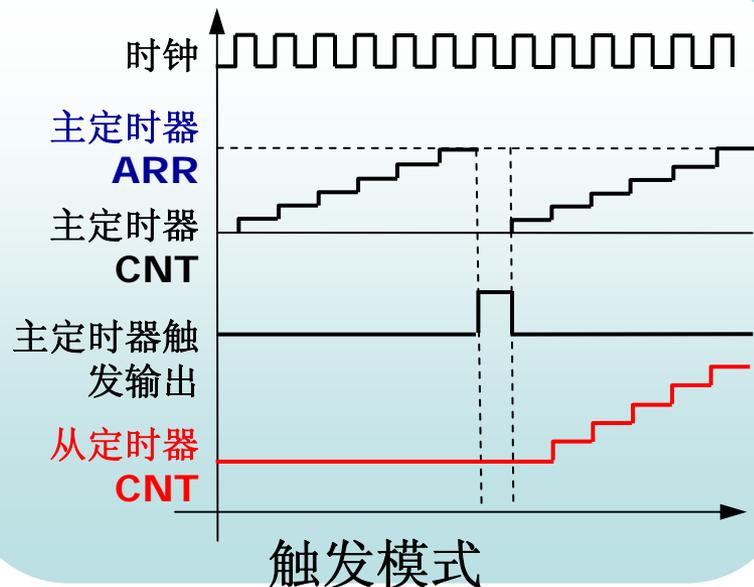
 配置相应的寄存器为TIMx选择所需的时钟源

时钟源	SMS	CEN	ECE	描述
内部时钟	000	1	--	
外部时钟源模式1	111	1	--	计数器在选定引脚的上升沿或下降沿计数
外部时钟源模式2	--	1	1	计数器在每个外部触发输入(ETR)的上升沿或下降沿计数
内部触发输入 (ITRx)	100 101 110	1	--	将一个计数器的输出作为另一个的输入。工作在Slave mode.

# TimerX的slave modes用法及常见应用场合

## TimerX 的几种Slave mode

- 复位模式：计数器、预分频器被初始化，若 (URS=0)，产生更新事件。
- 门控模式：计数器的启停均由触发信号控制。
- 触发模式：计数器的启动由触发信号控制。



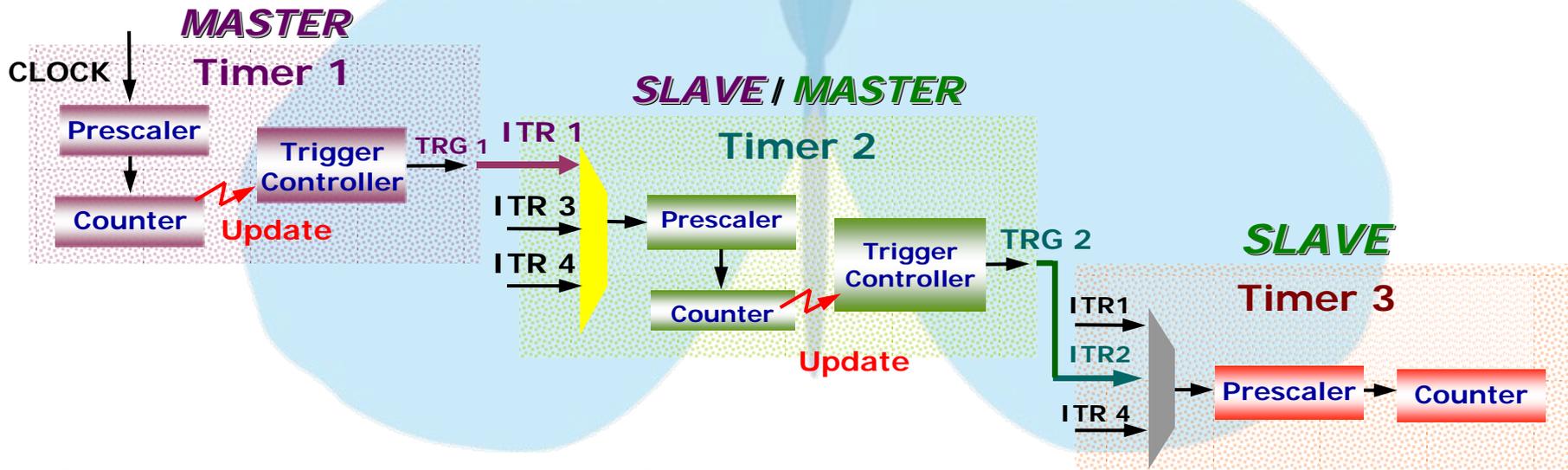
# TimerX的slave modes用法及常见应用场合

利用以上各种Slave模式，可以有以下一些常用的方式来对Timer进行同步

- ▣ 使用一个定时器作为另一个的预分频器
- ▣ 使用一个定时器去使能另一个定时器
- ▣ 使用一个定时器去启动另一个定时器
- ▣ 使用一个外部触发同步地启动多个定时器

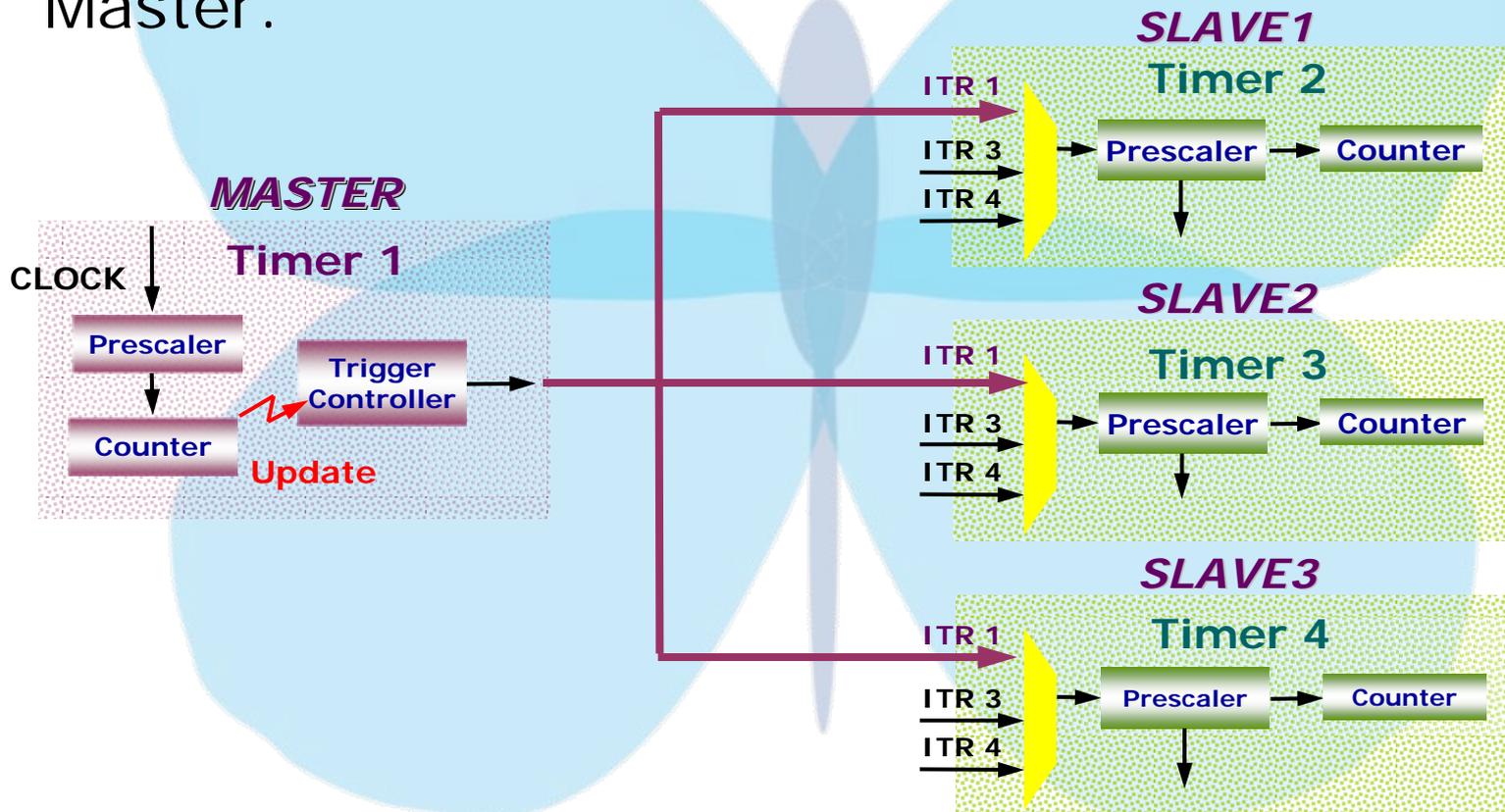
连接方式

- ▣ 串联模式：TIM1是TIM2的Master, TIM2 是TIM1的Slave和TIM3的Master



# TimerX的slave modes用法及常见应用场合

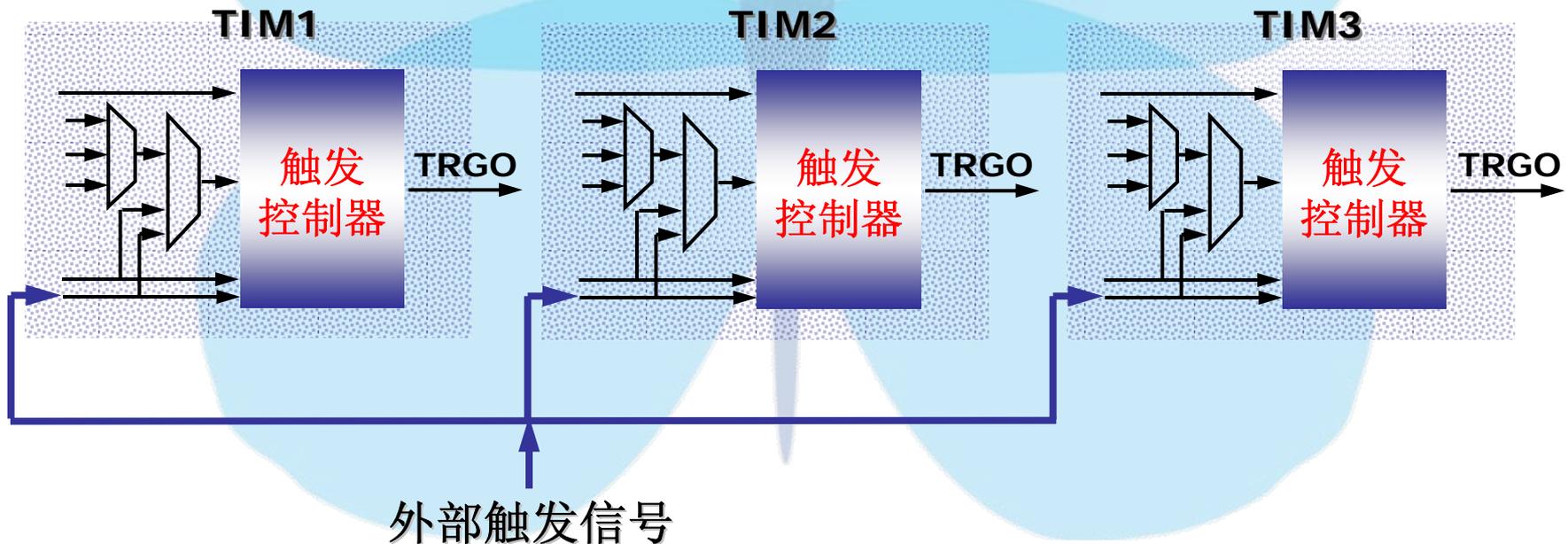
一主多从连接方式：TIM1 作为TIM2, TIM3 和 TIM4的 Master.



# TimerX的slave modes用法及常见应用场合

## 多个Timer与外部触发信号同步

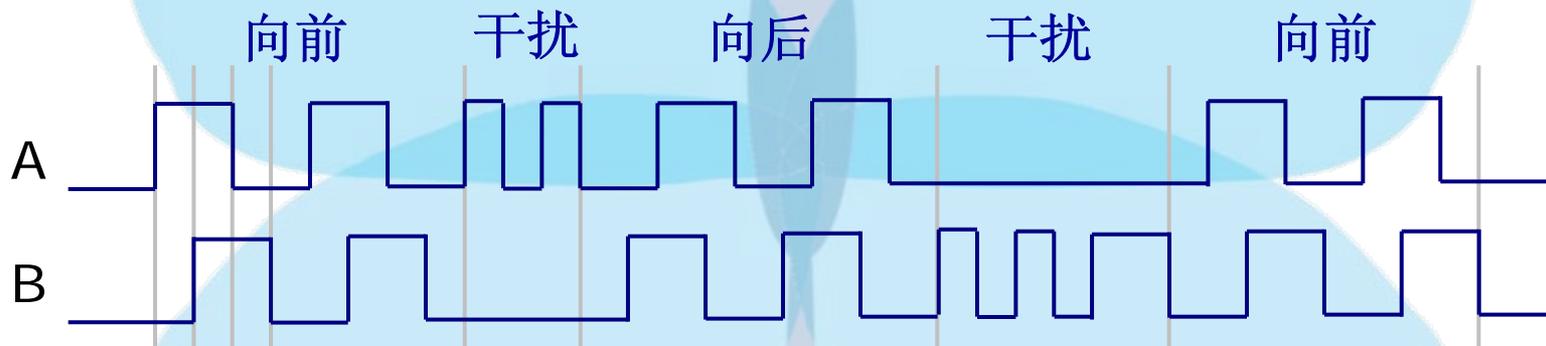
- 外部信号分别连接到TIM1, TIM2和TIM3的输入端, TIM1、TIM2和TIM3工作在Slave mode



# 如何使用 Timer 连接正交编码器（1）

## 正交编码器原理

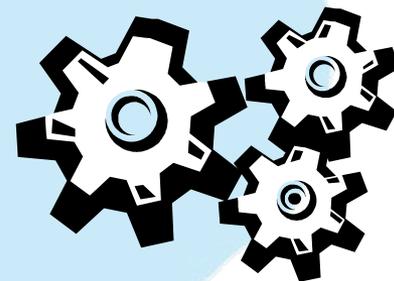
- 光电编码器，分为增量式和绝对式，可反馈马达的转子位置及转速信号；
- STM32F10x可与增量式正交编码器直接接口；
- 增量式正交编码器输出信号波形：



## Timer与正交编码器的接口

### 可编程的计数率

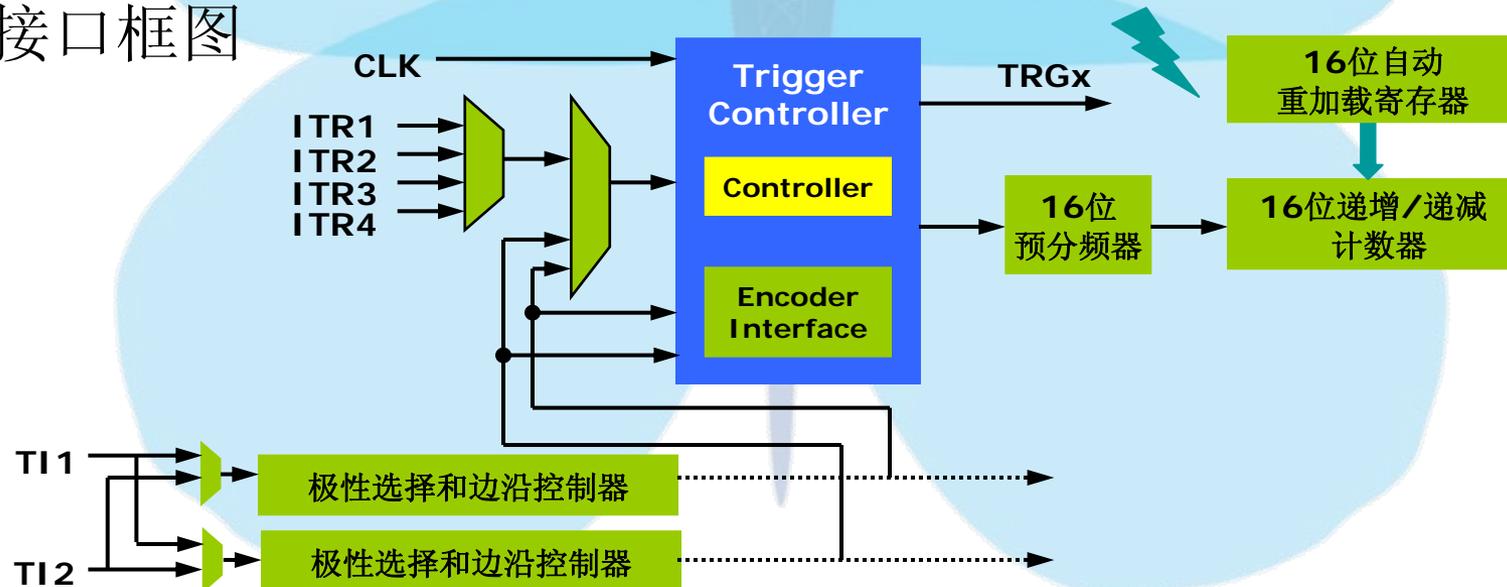
- x4: 标准模式，所有边沿有效
  - 1000线的正交编码器每转产生4000个计数脉冲
- x2: 只对A (或B)计数，但仍可确定方向
- "转速模式": 正交编码器输入可分频



## 如何使用 Timer 连接正交编码器（2）

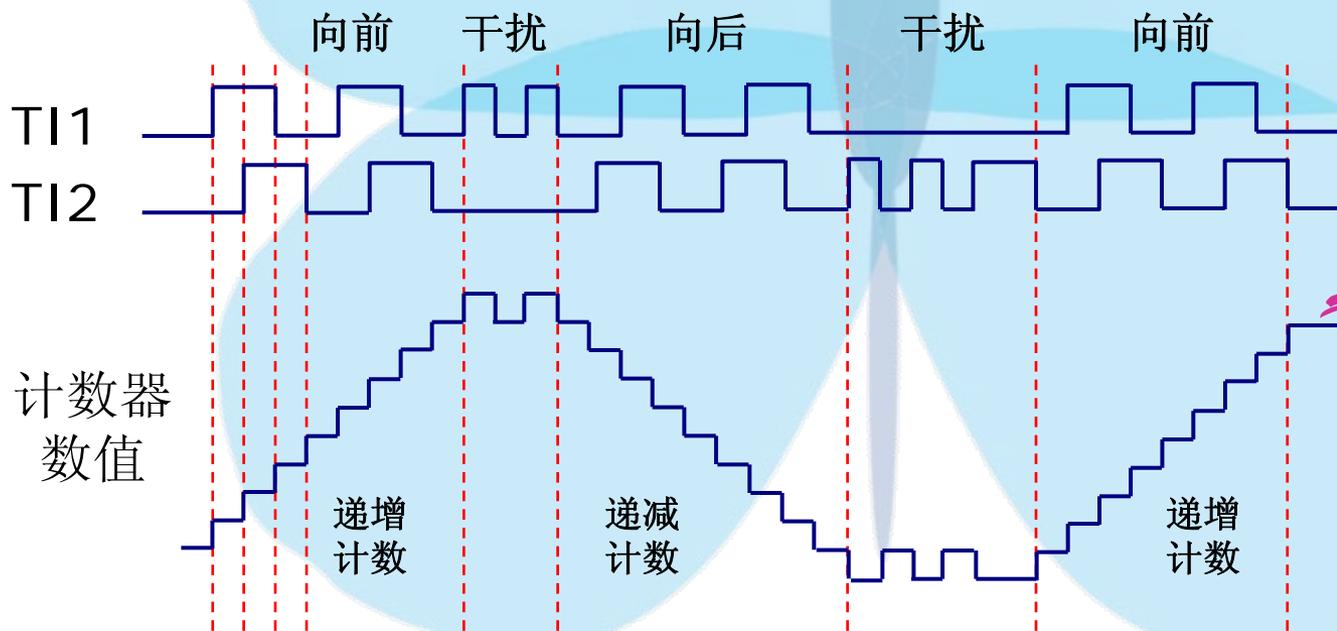
- 当自动重载寄存器的值配置为正交编码器每转产生的计数脉冲时，则计数器的值直接为马达转子的角度/位置
- 编码器每转一周可发出一个或多个中断
  - 一个，每 $360^\circ$ ；
  - 多个，每 $60^\circ$ 、 $90^\circ$ 、... (依赖于自动重载寄存器的配置)

### 接口框图



## 如何使用 Timer 连接正交编码器（3）

- 两个输入TI1和TI2被用来作为增量编码器的接口
- 依据两个输入信号的跳变顺序，计数器向上或向下计数，因此TIM1\_CR1寄存器的DIR位由硬件进行相应的设置。
- 编码器模式下的计数器操作实例



<http://blog.ednchina.com/STM32/>

## STM32博客空间

- 在STM32中如何配置片内外设使用的IO端口
- STM32 GPIO端口的输入电平说明
- 如何使用STM32的USB非控制端点发送多个数据包
- 【分析】STM32的代码，跑在RAM里快？还是跑在Flash里快？
- STM32中如何使用PC14和PC15
- 如何使用STM32的USB库支持控制端点0
- STM32(Cortex-M3)中的优先级概念
- 什么是IAP？如何实现IAP？
- STM32中用到的Cortex-M3寄存器说明
- STM32质量报告和可靠性报告
- 关于使用STM32的USART模块实现Modbus协议的讨论
- 使用STM32的单个普通定时器产生4路不同频率的方波
- 关于IO用作复用功能时的时钟设置注意要点



谢谢各位！