

# 写在前面の食用指北

本文是笔者对TIM的全部了解与原理解释，TIM号称stm32最强大同时也是复杂的外设，里面有很多晦涩难懂的概念，但好在我们需要用到的只是里面的一小部分，所以在这里放一个重点指路（也就是说其他部分看不懂没关系，下列重点一定要明白）

- 时基单元计数频率计算方法（包括CNT，ARR，PSC等的关系）
- TIM的更新中断
- PWM的原理

最下方有对cube中配置的解释与常用库函数，有需自取

## 从RCC与时钟树讲起

### stm32的时钟树框图

RCC（Reset and Clock Control）模块是用于控制STM32的时钟和复位系统的。它提供了时钟源选择、时钟分频、时钟输出、复位和时钟树控制等功能，可以根据应用需求配置系统时钟。它可开启或关闭各总线的时钟，在使用各外设功能必须先开启其对应的时钟，没有这个时钟内部的各器件就不能运行（在cube配置时会自动打开，不用担心）。

**时钟是单片机运行的基础，时钟信号推动单片机内各个部分执行相应的指令。**所谓时钟树就是将时钟源经过层层分频或者倍频，得到一系列的频率用于不同的总线或外设上，使板子上的外设都在某一恰当频率下工作。

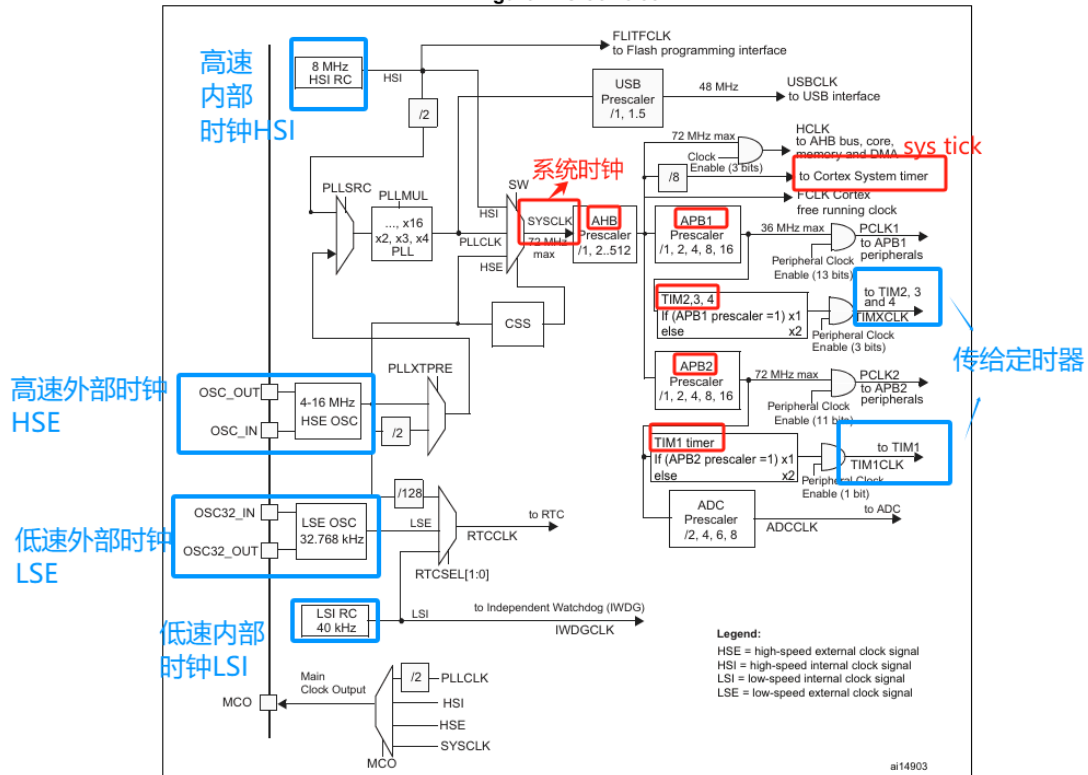
其中 `prescaler` 即为分频，如二分频就是将频率分为原来的一半；`p11mul` 是锁相环，有倍频功能

为什么 STM32 要有多个时钟源呢？

STM32本身十分复杂，外设非常多 但我们实际使用的时候只会用到有限的几个外设，使用任何外设都需要时钟才能启动，但并不是所有的外设都需要系统时钟那么高的频率，为了兼容不同速度的设备，有些高速，有些低速，如果都用高速时钟，势必造成浪费 并且，同一个电路，时钟越快功耗越快，同时抗电磁干扰能力也就越弱，所以较为复杂的MCU都是采用多时钟源的方法来解决这些问题。所以便有了STM32的时钟系统和时钟树

原文链接：<https://blog.csdn.net/as480133937/article/details/98845509>

Figure 2. Clock tree



1. When the HSI is used as a PLL clock input, the maximum system clock frequency that can be achieved is 64 MHz.
2. For the availability of the USB function both HSE and PLL must be enabled, with USBCLK running at 48 MHz.
3. To have an ADC conversion time of 1  $\mu$ s, APB2 must be at 14 MHz, 28 MHz, or 56 MHz.

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。
- 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。

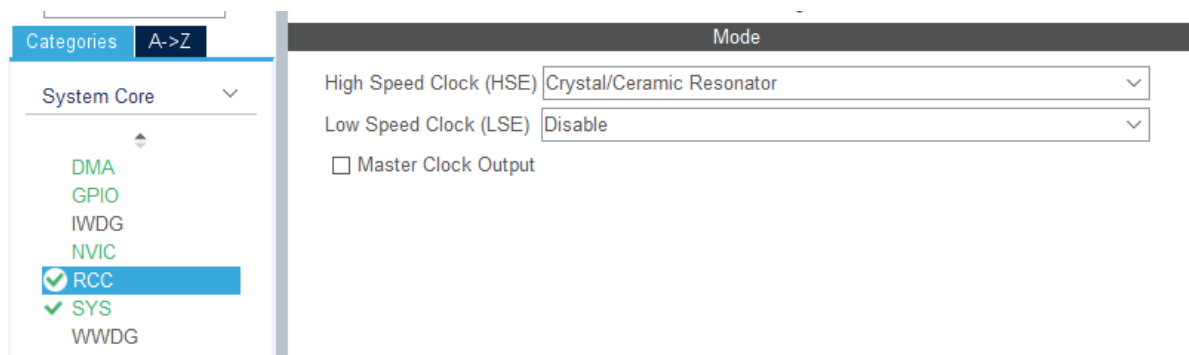
原文链接: <https://blog.csdn.net/as480133937/article/details/98845509>

还记得这里吗 这里选择的的就是时钟源:

**BYPASS Clock Source** (旁路时钟源)

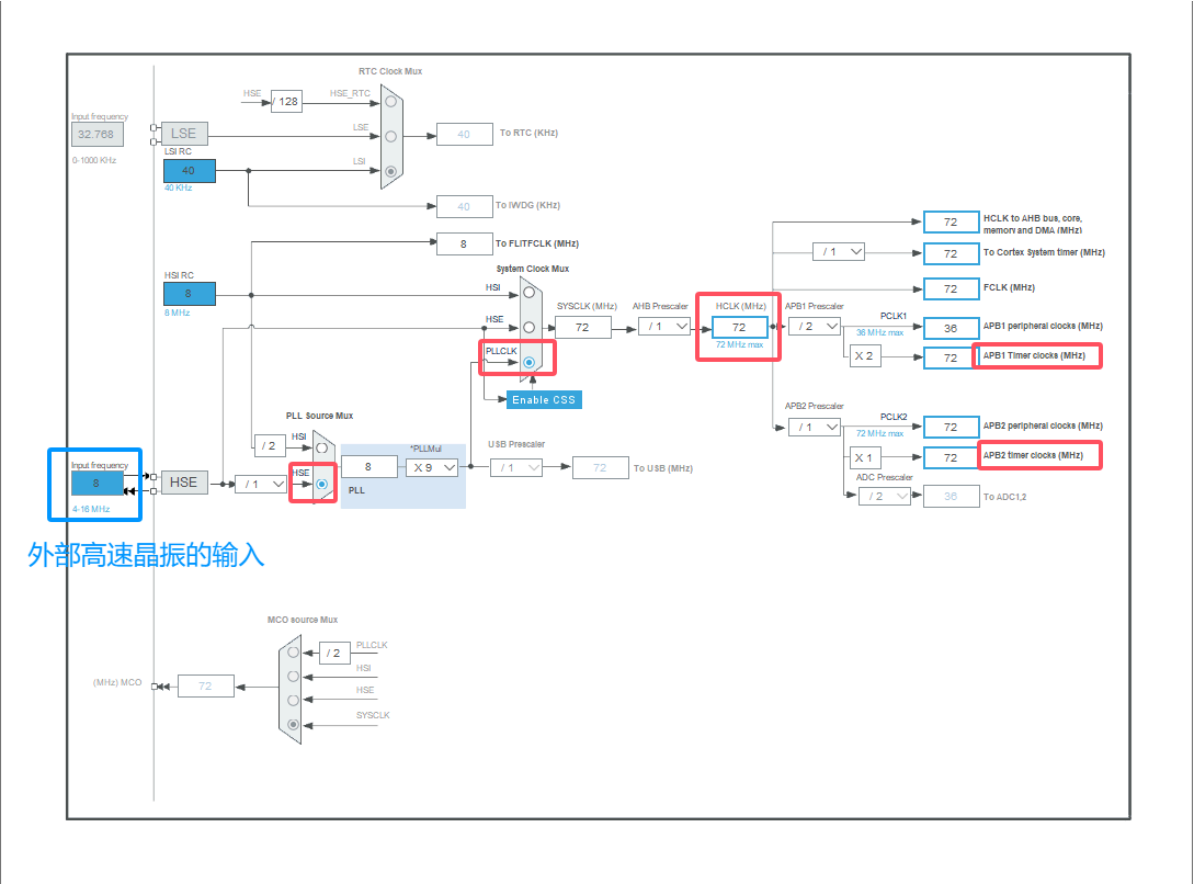
**Crystal/Ceramic Resonator** (晶体/陶瓷晶振)

虽然HSI和HSE频率可能相同，但我们通常选择外部晶振，因为外部晶振一般为石英晶体，振荡频率更为稳定



# cube中的时钟树配置图：

- 操作要点：
- 1.正确输入外部高速晶振的输入频率
  - 2.选好如图几个选项 （此步也可以直接拉中HCLK，将其数值改为max的数值，即直接拉到最大，其他设置都会相应地自动调整）

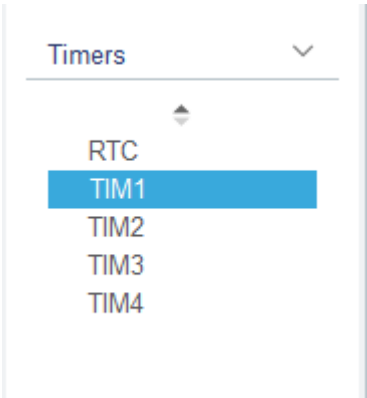


备注：其中这个HSE的 Input frequency 是需要我们去查询板子上的外部晶振频率的，我们手上这块最小系统板频率为8MHz

## TIM外设

(注意RCC与TIM并不是一个概念，RCC控制着整个系统的时钟，而TIM只是工作在对应频率下的计数器外设。也就是说TIM外设就是一个专门用来以固定频率计数的一个计数工具)

### 了解stm32的定时器资源



stm32f103c8t6 有一个 RTC(实时计数器, 可在掉电后继续计时)和四个 TIM 外设, 其中TIM1是高级定时器, TIM2, 3, 4是通用定时器 (除此之外还有系统滴嗒定时器和看门狗定时器, 感兴趣可以自行了解), 我们的主要目的是了解TIM1234。

下表比较了高级控制定时器、普通定时器和基本定时器的功能:

表4 定时器功能比较

定时器	计数器分辨率	计数器类型	预分频系数	产生DMA请求	捕获/比较通道	互补输出
TIM1	16位	向上, 向下, 向上/下	1~65536之间的任意整数	可以	4	有
TIM2 TIM3 TIM4	16位	向上, 向下, 向上/下	1~65536之间的任意整数	可以	4	没有

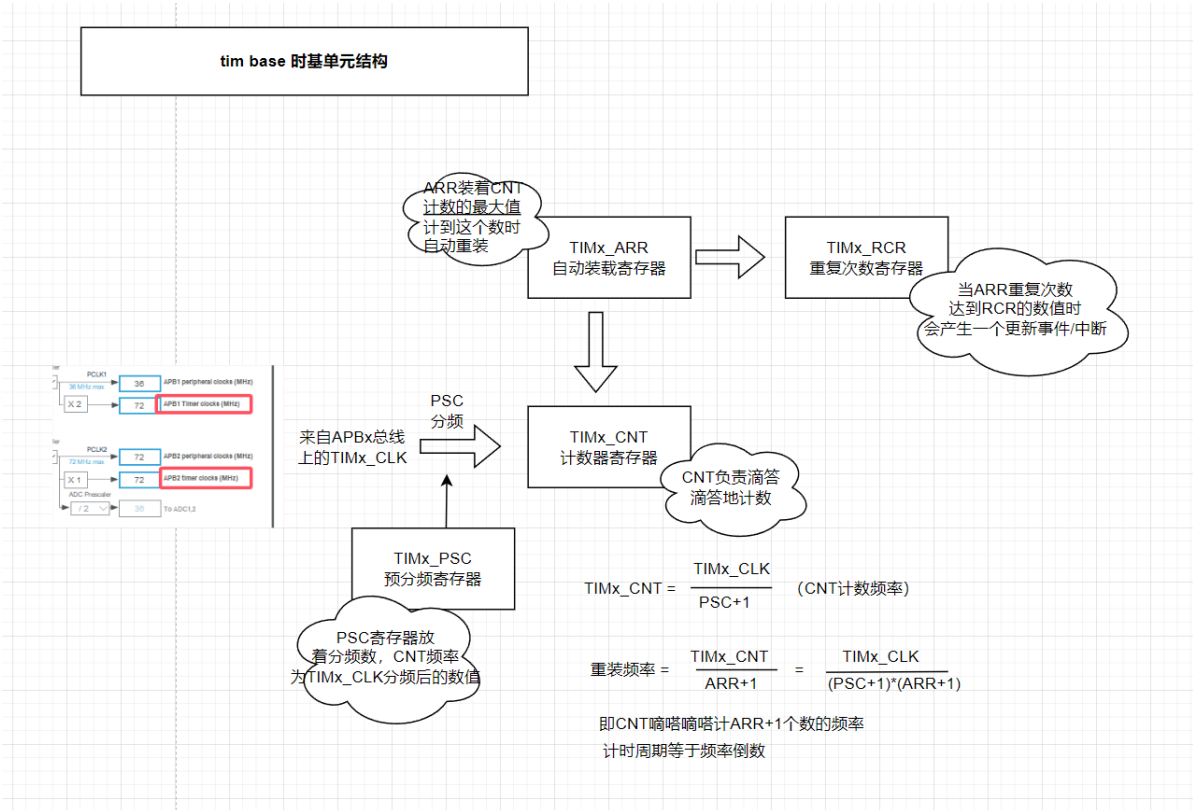
功能比较: 高级定时器>通用定时器>基本定时器, 下面我们将由简单到复杂认识这个外设

## TIM结构分析

### 基本结构: 时基单元 (重)

时基单元是定时器实现计数功能最基本的结构, 来自APB1或2上的TIMx\_CLK脉冲是TIM的输入, 这个脉冲经过PSC分频之后成为CNT的计数脉冲 (每一个脉冲进来CNT计数就加一, 由于脉冲周期是固定的, 由此就由计数衍变为了计时的效果) CNT计数有向上, 向下或向上向下三种模式, 其中向上计数是CNT从零开始加加加到ARR (ARR寄存器中存放着最大装载数值, 我们称这个数为ARR) 然后向上溢出并重装, 向下就是CNT从ARR递减到0然后向下溢出并重装, 向上向下即两者的结合。TIM的溢出可以触发更新中断 (在下面的下面会详细讲) 在已经使能的情况下重装次数达到RCR中的数值时会自动触发更新中断。我们一般将RCR配置为0 (即ARR溢出一次就会触发中断)

利用时基单元的计数, 我们可以就可以手搓定时器啦。 (大家可以先想一想怎么实现)

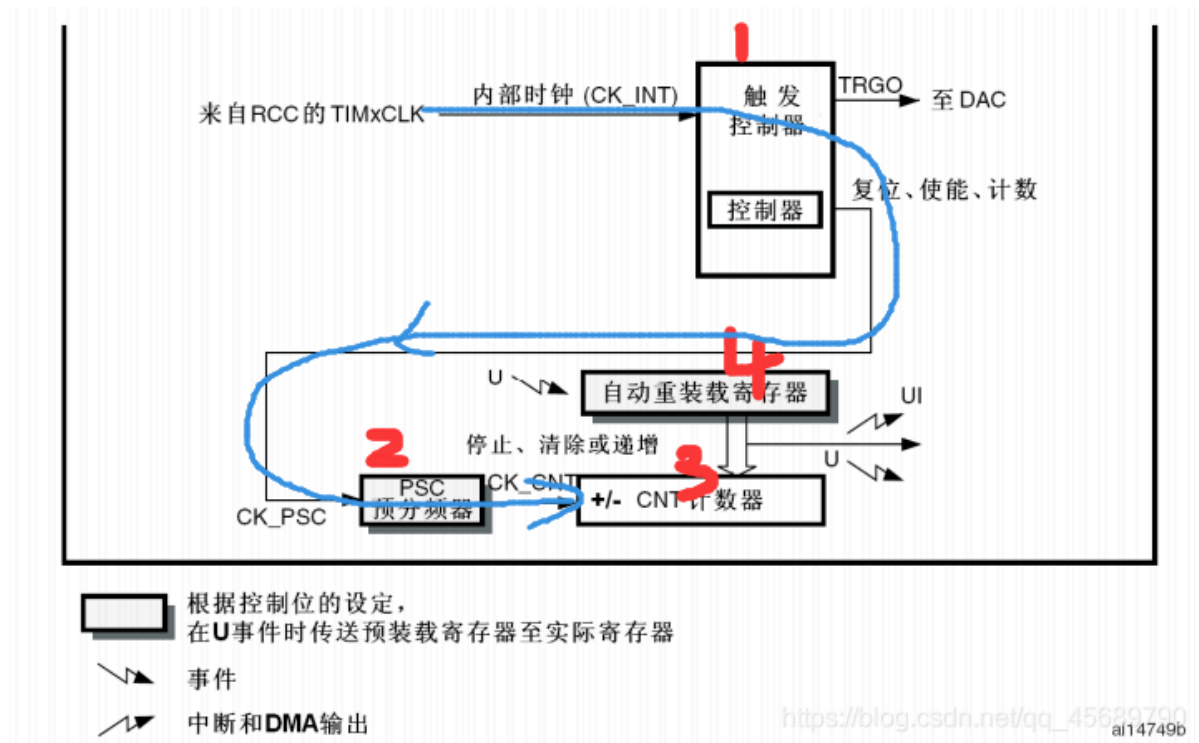


各频率计算如上图 (这些公式很重要!!!)

P.S. PSC, ARR, RCR 的数值都加一是因为寄存器的数值都是从零开始计数的, 因此实际值比寄存器中的值大1.

由于这几个寄存器都是16位的，所以它们的取值范围都是0~65535 ( $2^{16}=65536$ )，由此也可以知道stm32的计时是有上限的，单个计时器在72MHz下最多计时  $\frac{1}{72000000} \times 65536 \times 65535$  s ARR就会溢出 (大约59.65s)，若还嫌不够 可以使用stm32的定时器级联功能 (一般不会不够啦)

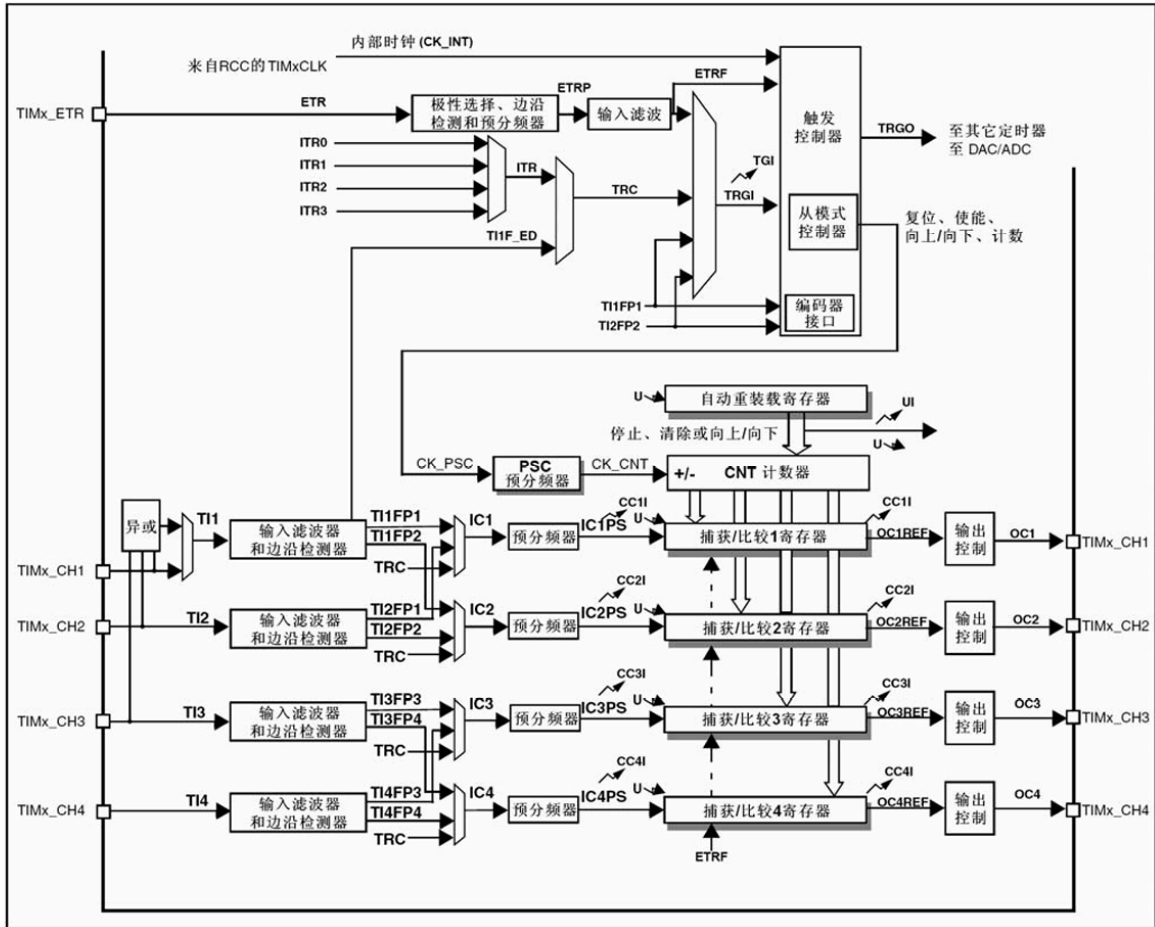
**基本定时器**只有一个时基单元 (只有计数功能) (下图为网上扒的基本定时器框图，原理同上) (c8t6没有基本定时器)



## 时基外围的复杂结构 (略)

谈到通用甚至高级定时器，则在时基单元的基础上外加了许多复杂的结构----->((注意框图看不懂没有关系 (因为我也看不懂) 只是想让大家知道有这么一个东西存在))

图98 通用定时器框图

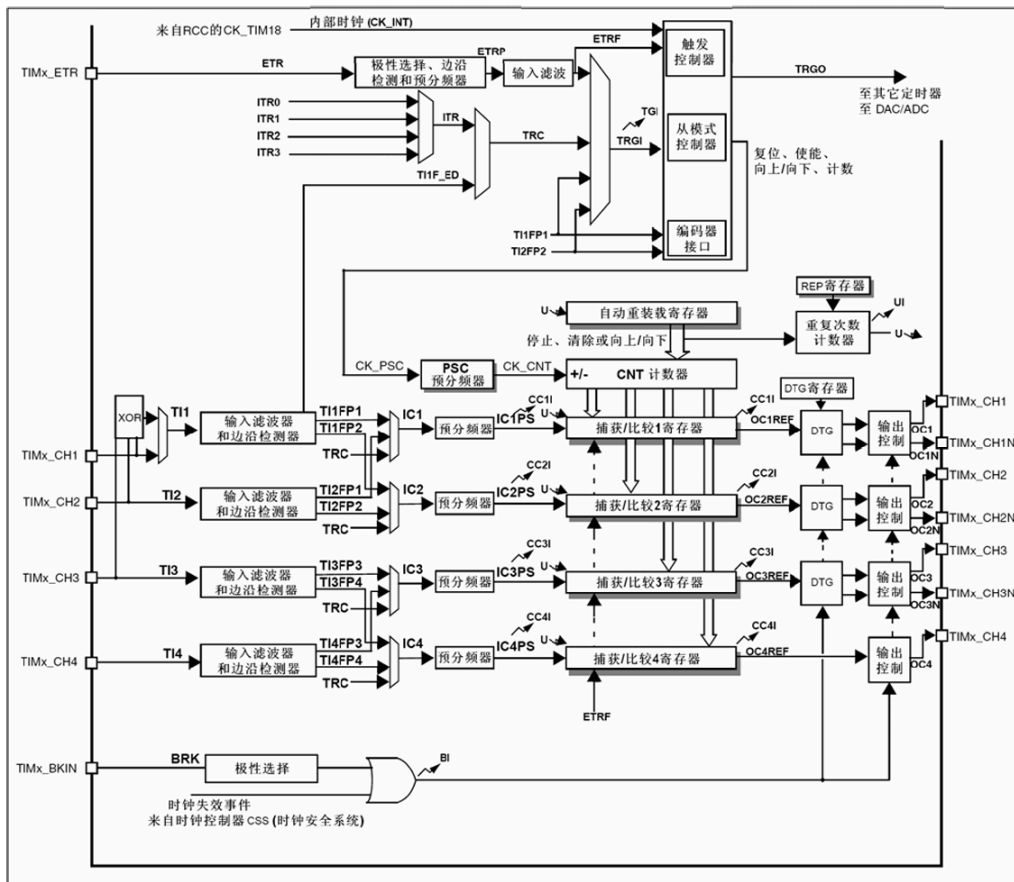





注:  根据控制位的设定, 在U事件时传送预加载寄存器的内容至工作寄存器

 事件

 中断和DMA输出

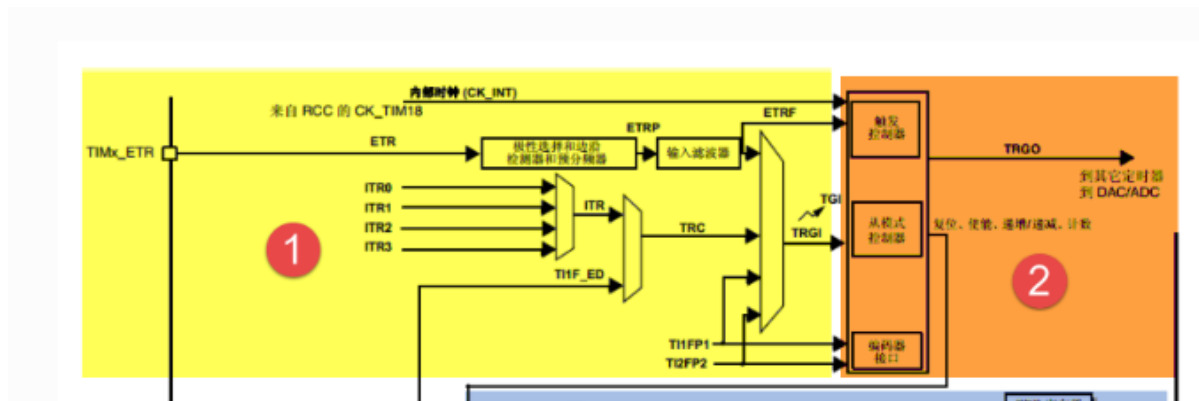
图50 高级控制定时器框图



注：  
 根据控制位的设定，在U(更新)事件时传送预加载寄存器的内容至工作寄存器  
 事件  
 中断和DMA输出

可以发现高级定时器其实只比通用定时器多了最下方的一个BKIN（互补输出 刹车输入功能）（目前基本用不到，不重要）所以就把它们的结构放在一起介绍啦

### 高级控制定时器的四个时钟源



参考[32. TIM—高级定时器](http://www.embedfire.com) — 野火STM32库开发实战指南——基于野火MINI开发板 文档  
[www.embedfire.com](http://www.embedfire.com)

高级控制定时器有四个时钟源可选：

- **内部时钟源CK\_INT**（我们一般用这个，其他有兴趣了解没兴趣跳过）

CK\_INT来自芯片内部的总线，是输入TIM的频率(TIMx\_clk)

- **外部时钟模式1：外部输入引脚Tlx (x=1,2,3,4)**

。计数器可以在选定输入端的每个上升沿 或下降沿计数。选择这个模式时输入时钟信号来自于定时器的输入通道，总共有4个，分别为TI1/2/3/4，即TIMx\_CH1/2/3/4

- **外部时钟模式2：外部触发输入ETR**

计数器能够在外部触发ETR的每一个上升沿或下降沿计数。选择这个模式时触发信号来自对应的ETR引脚（只有一个）

- **内部触发输入(ITRx)**

内部触发输入是使用一个定时器作为另一个定时器的预分频器。硬件上高级控制定时器和通用定时器在内部连接在一起，可以实现定时器同步或级联。主模式的定时器可以对从模式定时器执行复位、启动、停止或提供时钟。

## TIM的四个独立通道(TIM\_CH1~4)

TIMx的每个CHANNEL都对应了一个到两个引脚，每个通道可选择的功能如下

### — 输入捕获 (Input Capture) (略)

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

### — 输出比较 (Output Compare) (重)（输出比较和PWM会在下面详述）

此项功能是用来控制一个输出波形，或者指示一段给定的时间已经到时。

### — PWM生成(PWM Generation) (重)

脉冲宽度调制模式可以产生一个由TIMx\_ARR寄存器确定频率、由TIMx\_CCRx寄存器确定占空比的信号。

### — 单脉冲模式输出 (略)

单脉冲模式(OPM)是前述众多模式的一个特例。这种模式允许计数器响应一个激励，并在一个程序可控的延时之后，产生一个脉宽可程序控制的脉冲。

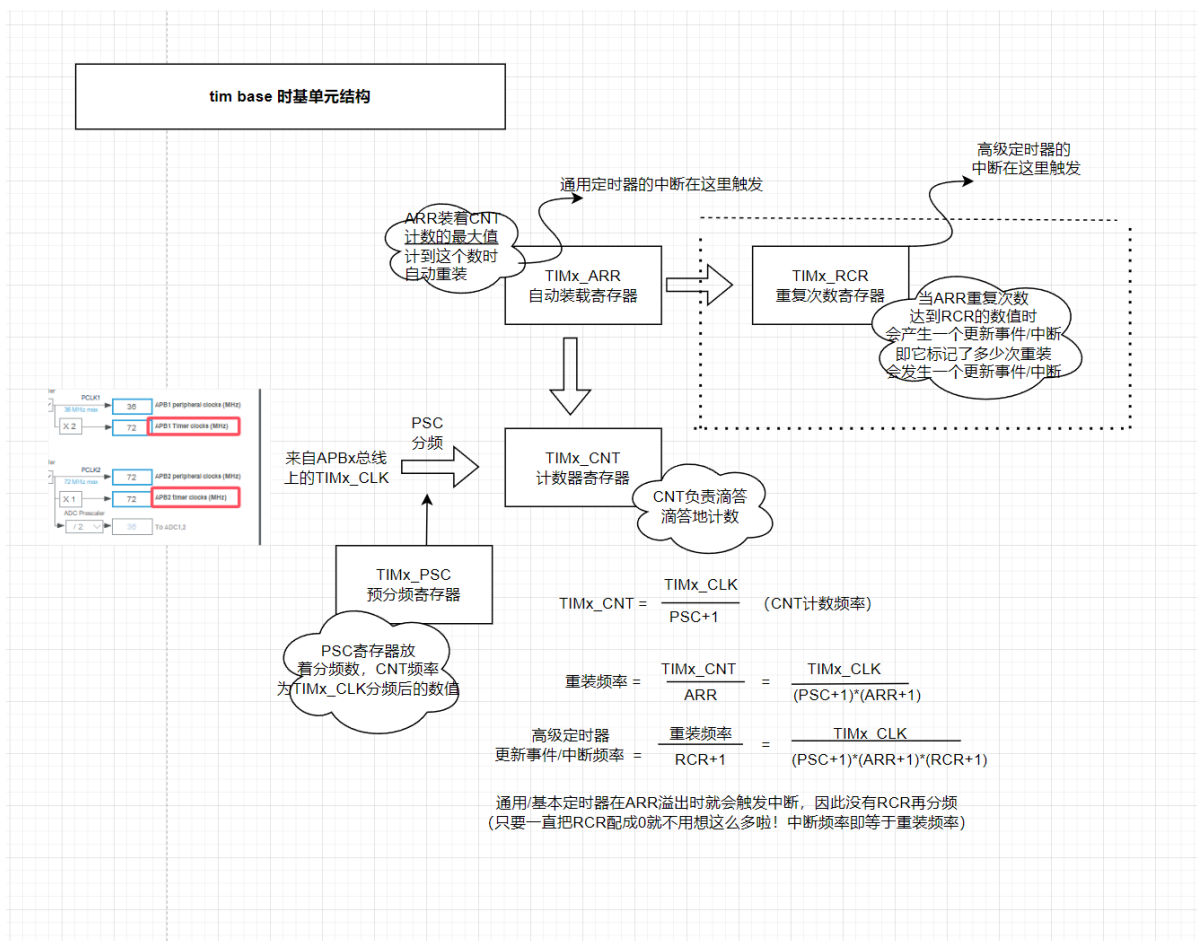
#####

## TIM定时器更新中断（重）（这里是原理和计算方法，配置方法放在后面）

定时器的中断既然属于中断，它和GPIO外部中断的原理其实是一样的，都是在**满足特定中断条件**时调动对应的中断服务函数以至中断回调函数，只是这个**中断条件**由外部引脚的触发变成了**定时器的溢出**，这种中断我们将其称为**更新中断**。更新中断其实只用到了时基单元的部分，与那些复杂的通道都没什么关系，这是TIM的基本功能。

下面是计算**定时时长**的计算方法。





时基的计时原理还是一样的，由图可知TIMx\_CLK频率为72MHz（在时钟树可以看到，c8t6是这个数，换其他板子的时候就不一定了）。配置PSC的数值和ARR的数值，CNT的周期为  $T(TIMx\_CLK) * (PSC+1)$ ，（以向上计数为例）只要CNT吭哧吭哧跑到ARR+1，定时器就会溢出，（在已经使能的情况下）就会触发中断啦

所以最最最重要的就是这个公式：

中断频率：

$$\frac{TIMx\_CLK}{(PSC+1)*(ARR+1)}$$

每隔时长

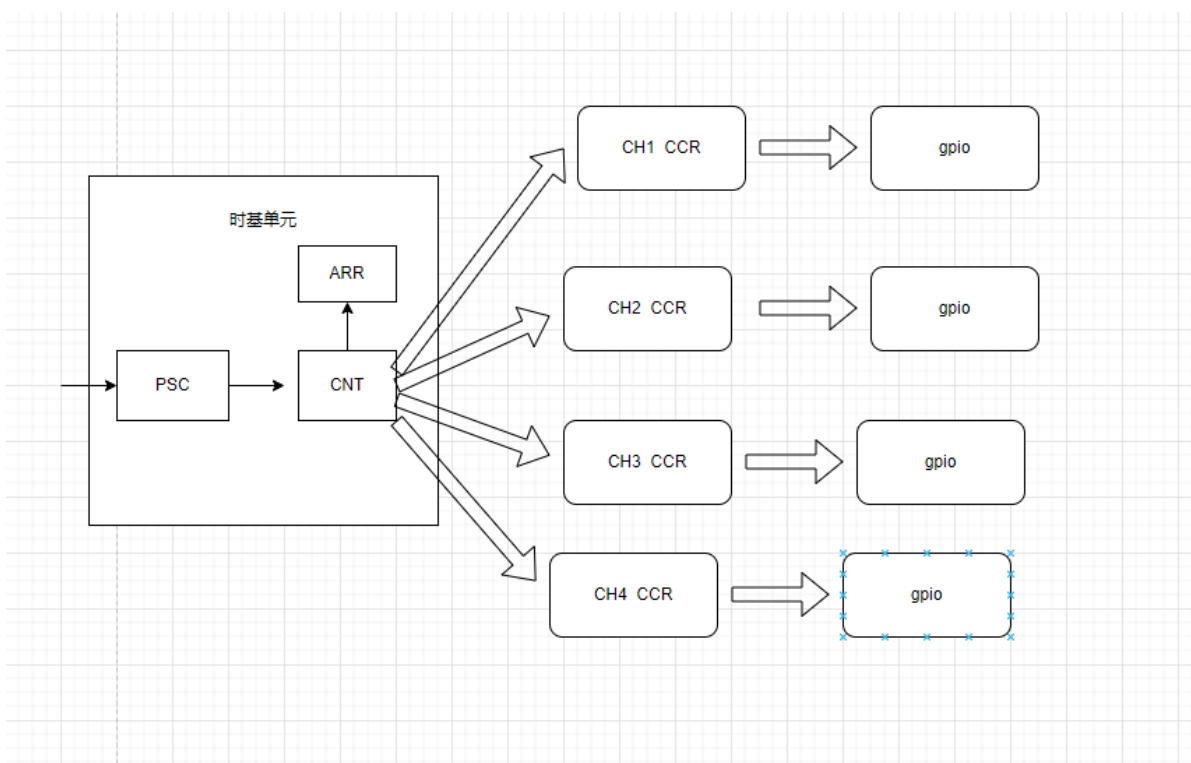
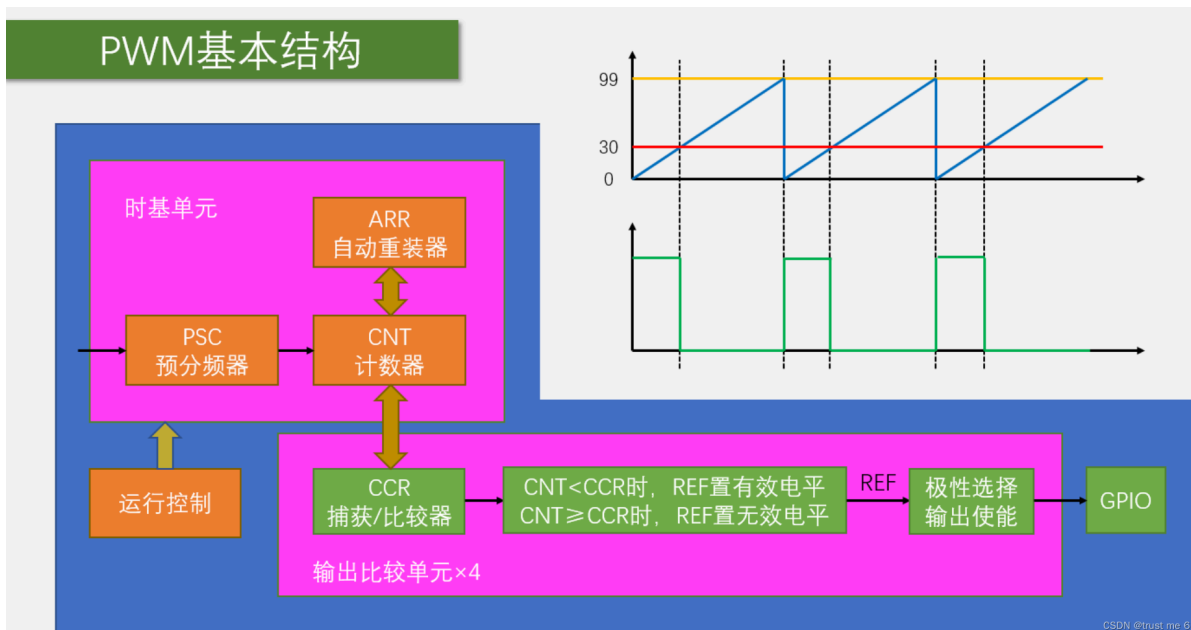
$$\frac{(PSC+1)*(ARR+1)}{TIMx\_CLK}$$

就会触发一次定时中断。

## PWM--脉冲调制电路（重）（原理）

务必点开看：[STM32——PWM原理及应用（附代码）](#) [stm32\\_pwm-CSDN博客](#) 主要了解其原理（主要概念解释都在文档里）

## PWM基本结构



每一个通用/高级计数器都有四个独立通道，可用于输出PWM波，且同一个定时器的四个通道PSC,ARR和CNT值是一样的(同一个时基单元)，但是捕获比较寄存器CCR有四个，可以输出四路PWM波。

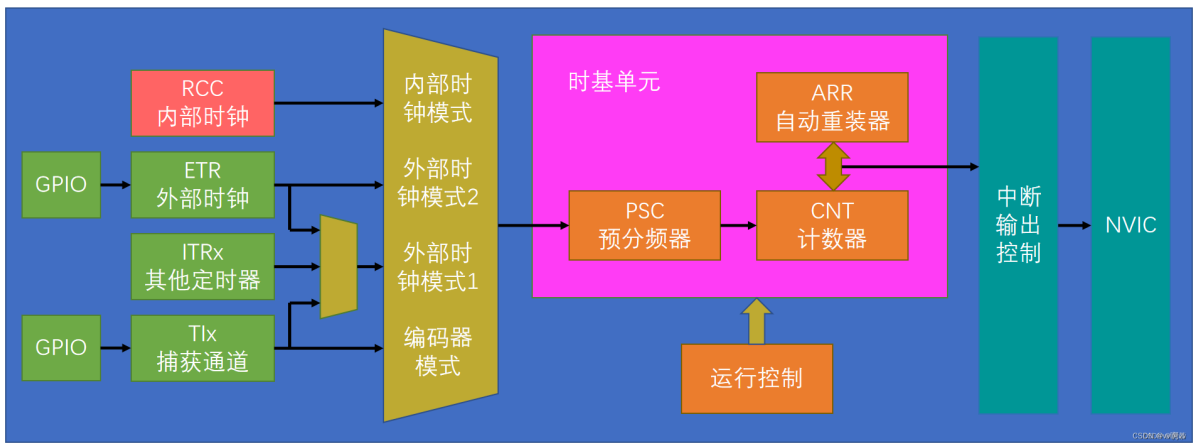
产生过程：CNT从0开始递增（或者从ARR开始递减），然后与CCR中的数进行比较，若大于CCR置有效电平，小于CCR置无效电平（有效无效哪个是高电平与所选的配置有关，但总之它们相反）然后从GPIO输出。所以PWM的周期就是CNT走（ARR+1）个数的时间，频率的计算方法与溢出频率相同。得知PWM的脉冲与占空比，即可算出有效电平脉冲宽度。

**PWM频率：**  $\text{Freq} = \text{CK\_PSC} / (\text{PSC} + 1) / (\text{ARR} + 1)$

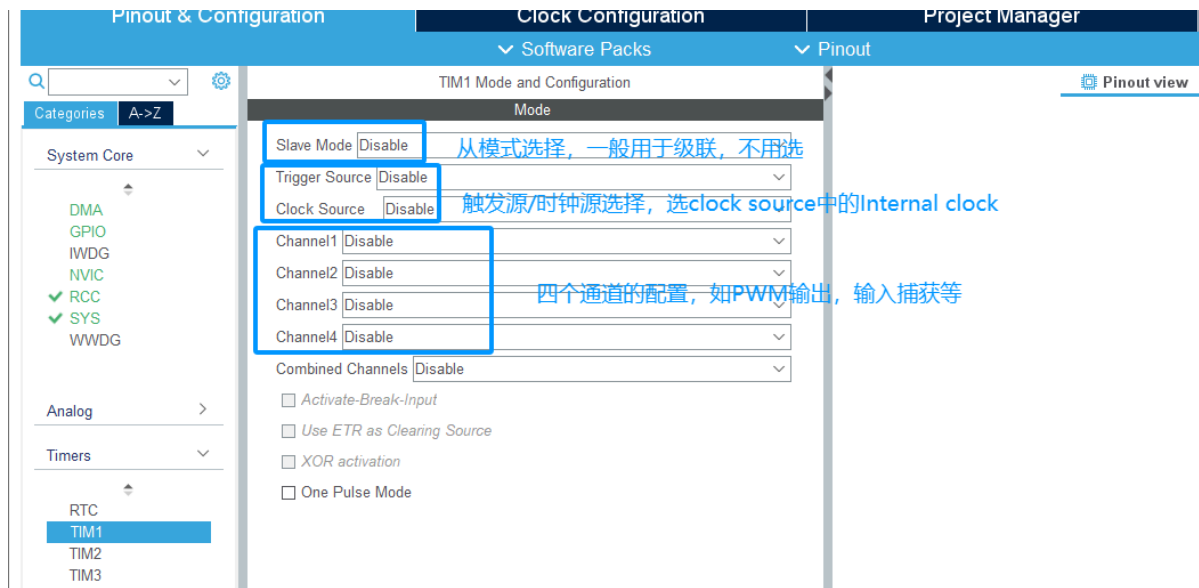
**PWM占空比：**  $\text{Duty} = \text{CCR} / (\text{ARR} + 1)$

**PWM分辨率：**  $\text{Reso} = 1 / (\text{ARR} + 1)$

至此原理部分讲解就差不多啦 再附上一张图 往下就是cube的配置和代码部分啦



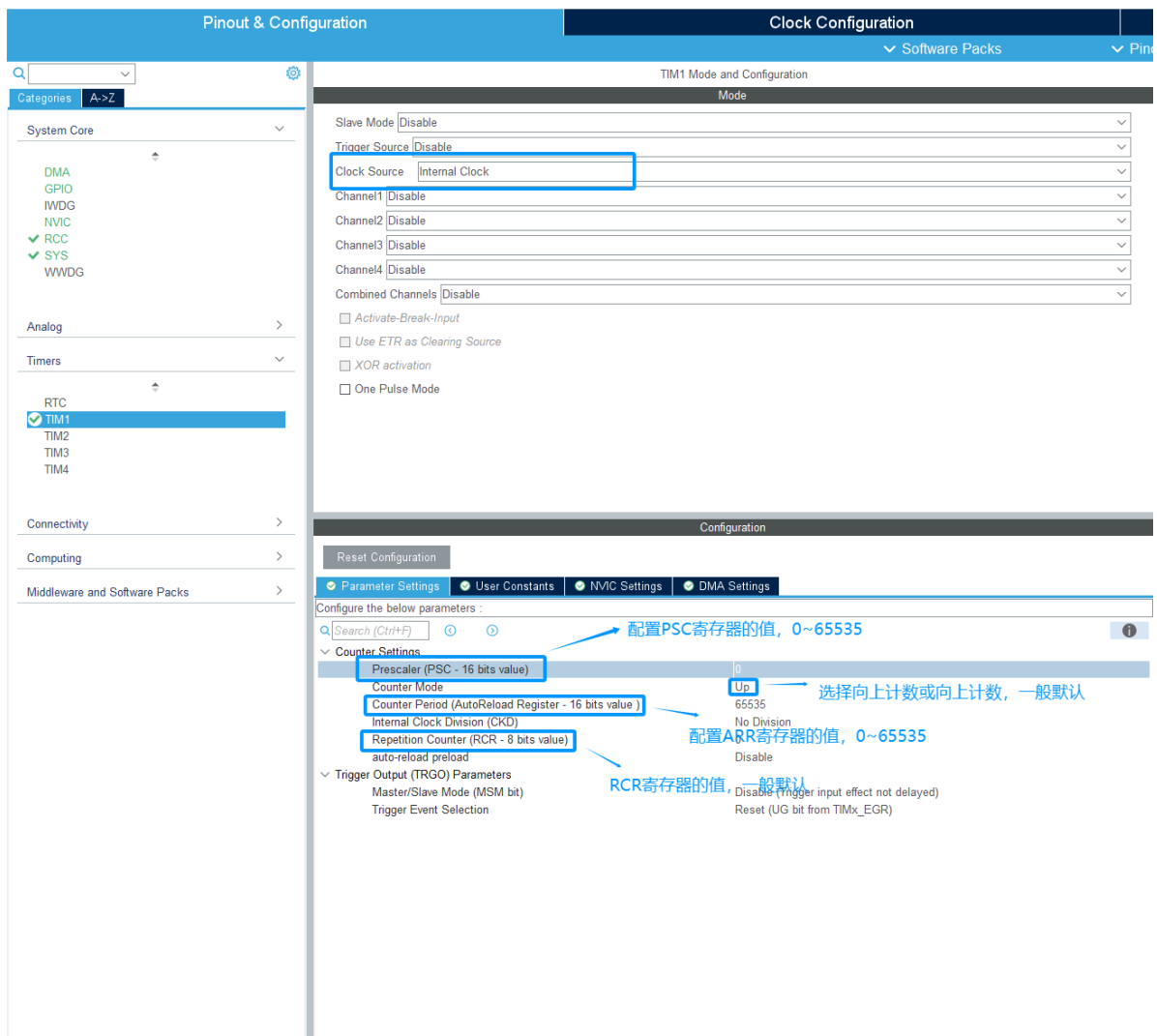
## 从cube配置以及库函数再认识TIM外设（实用）



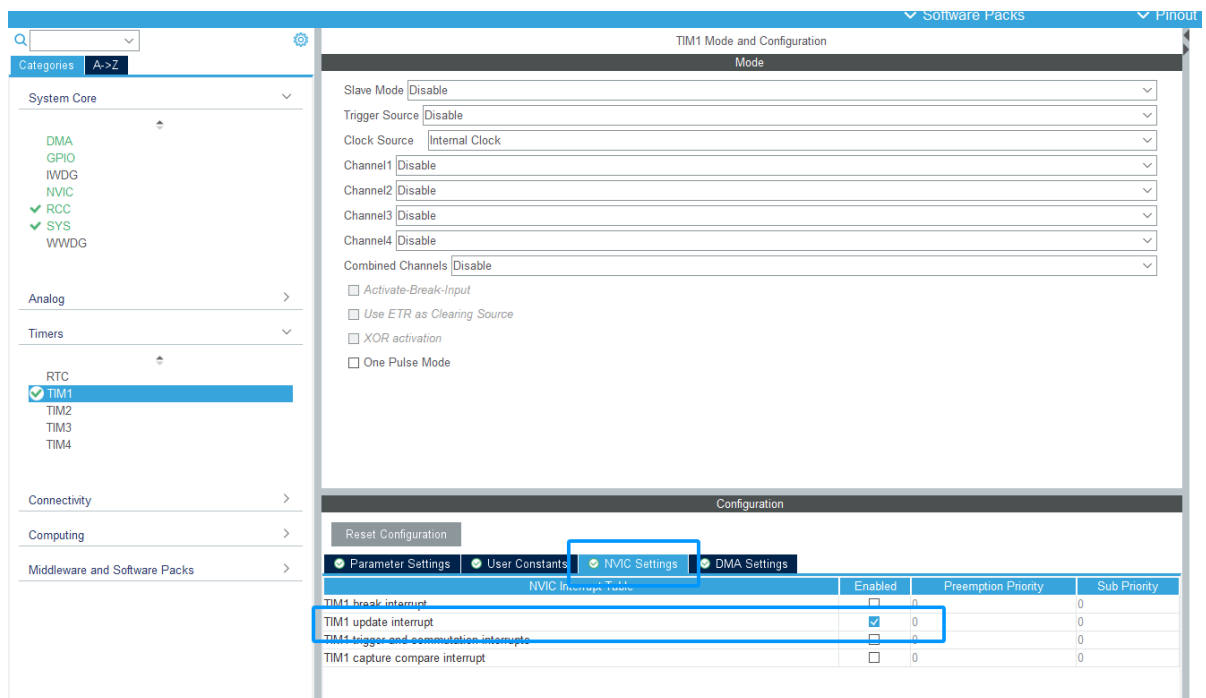
### Pro1 ---- 使用定时器及其中断

根据所需要的频率配置好PSC与ARR的值，注意所填的值应该在所需要的分频数或者重装数的基础上减一，比如原来主频为72MHz，我希望它经过PSC后为1MHz，那么我的PSC应为 $72-1=71$ 。可以直接往里面输入72-1，它会帮你算好的~

其他的默认就好，不用配置



如果要使用更新中断, 务必在NVIC这里把Update interrupt的使能勾上!



其他的必要配置同GPIO所写教程 (包括RCC, SYS, 时钟树配置)

常用库函数:

/\*\*

\* 功能: 启动时基单元 (这一句之后时基会开始嘎嘎计数)

```

* 参数1: htim 定时器句柄, 如 &htim1
*/
HAL_StatusTypeDef HAL_TIM_Base_Start(TIM_HandleTypeDef *htim);

/**
* 功能: 关闭时基, 停止计数
* 参数1: htim 定时器句柄
*/
HAL_StatusTypeDef HAL_TIM_Base_Stop(TIM_HandleTypeDef *htim);

/** (这其实是一个宏定义而非函数)
* 功能: 设置ARR的值 (是的, ARR能在cube中配置, 当然也可以用代码修改
* 参数1: htim 定时器句柄, 如&htim1
* 参数2: 所设ARR值 (记得减一)
*/
__HAL_TIM_SetAutoreload(__HANDLE__, __AUTORELOAD__);

/** (这其实是一个宏定义而非函数)
* 功能: 手动设置CNT值
* 参数1: htim 定时器句柄, 如&htim1
* 参数2: 所设cnt值
*/
__HAL_TIM_SetCounter(__HANDLE__, __COUNTER__)

/** (这其实是一个宏定义而非函数)
* 功能: 读取CNT寄存器的值
* 参数1: htim 定时器句柄, 如&htim1
* 返回值: 读到的cnt值
*/
__HAL_TIM_GetCounter(__HANDLE__);

/**
* 功能: 在中断模式下启动定时器 (如果要使用定时中断记得加这一句
* 参数1: htim 定时器句柄
*/
HAL_StatusTypeDef HAL_TIM_Base_Start_IT(TIM_HandleTypeDef *htim);

/**
* 功能: 中断模式下停止定时器
* 参数1: htim 定时器句柄
*/
HAL_StatusTypeDef HAL_TIM_Base_Stop_IT(TIM_HandleTypeDef *htim);

```

使用示例:

```

MX_USART1_UART_Init();
MX_TIM1_Init();
/* USER CODE BEGIN 2 */
HAL_TIM_Base_Start(&htim1);
/* USER CODE END 2 */

/* Infinite loop */

```

如图, 开始计时

已知HAL\_Delay()最低为ms级别, 那么如果我需要一个us级别的定时器----->

只需配置PSC=72-1, 那么易得每一微秒CNT++, 那么是否只需要让CNT反复加叻叻加 t 下, 我就可以成功计数 t 微秒了? ----->

```

void delay_us(uint16_t us)
{
    uint16_t counter = 0;
    __HAL_TIM_SetAutoreload(&htim2, us);           //设置ARR值
    __HAL_TIM_SetCounter(&htim2, counter);          //设置CNT初始值
    HAL_TIM_Base_Start(&htim2);                     //启动时基，开始计时

    while(counter != us) //counter是读取CNT所得到的值，即若CNT还没有到所设定的us，则一直
    计数，直到时间到了再跳出循环
    {
        counter = __HAL_TIM_GetCounter(&htim2); // 获取定时器当前计数
    }
    HAL_TIM_Base_Stop(&htim2); // 停止定时器
}

```

注意这样的计数器最长计数时间为65535us。

如果需要使用更新中断：

```

//更新中断回调函数：
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    if(htim->Instance == TIM1) //判断是哪个定时器触发的中断
    {

    }
}

```

开启时基需要使用这一句，这样才能使能中断

```

MX_TIM1_Init();
/* USER CODE BEGIN 2 */
HAL_TIM_Base_Start_IT(&htim1);
/* USER CODE END 2 */

```

中断回调函数的复写仍然写在4中，编写自己所需要的回调函数即可

```

L52 |
L53 | /* USER CODE BEGIN 4 */
L54 | void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
L55 | {
L56 |     if(htim->Instance == TIM1)
L57 |     {
L58 |
L59 |     }
L60 | }
L61 | /* USER CODE END 4 */
L62 |

```

## Pro2 ----pwm

要输出pwm波，只需要将所用的通道配置为PWM模式

tips：大家在选择通道的时候可能会注意到两个有点像的选项：Output Compare 和PWM Generation两种模式，下面是它们的区别

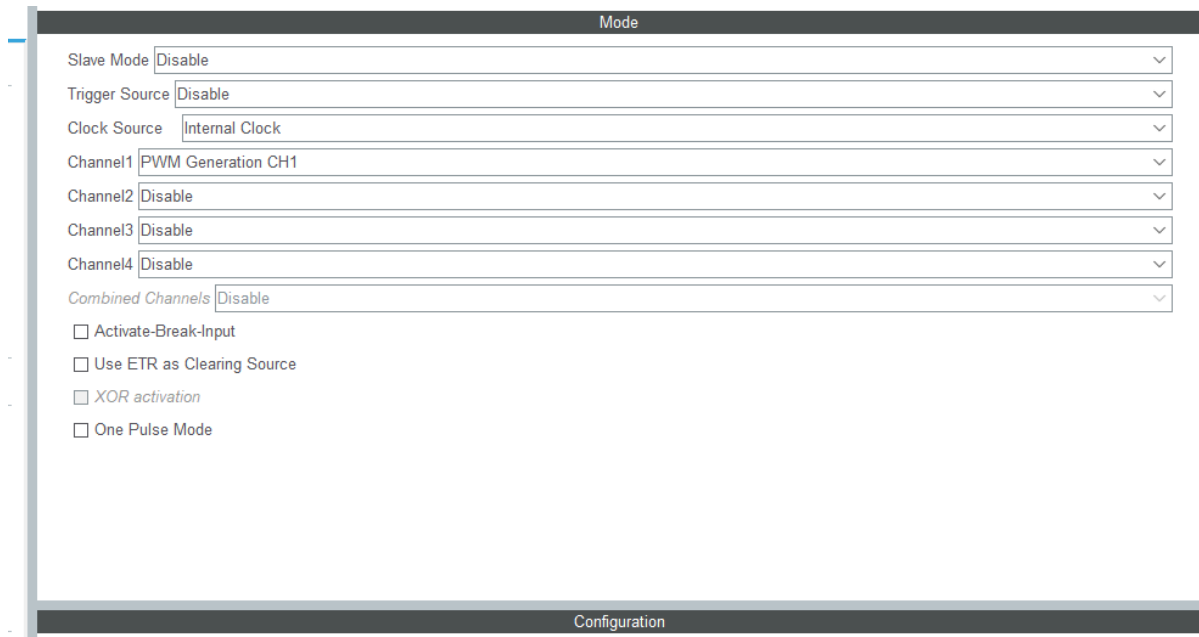
输出比较模式和PWM模式都可以用来输出PWM波，在功能上两者有相同之处，对于一个定时器这两种方式都可以做到四路输出PWM，每一路PWM占空比都可调，也有不同之处，输出比较模式可以方便的调节每一路PWM波的频率，可以输出四路频率不同，占空比不同的PWM。但是PWM模式如果想要调节PWM波的频率，那么就只能重新设置预分频系数或者自动重装载寄存器ARR，并

且输出的四路PWM频率必定一致。

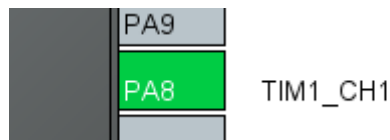
输出比较模式和PWM模式的原理很相似，在输出比较模式输出PWM的实验中，有一点不同的是PWM1递增模式下CNT与CCR作比较，若CNT小于CCR则输出为高电平，若CNT大于CCR则输出为低电平，并且在CNT计数至ARR时，CNT会更新至0并产生上溢事件。但是输出比较模式在CNT与CCR不断做比较的过程中，若CNT等于CCR，产生的则是电平翻转，并且会产生中断，通过对中断回调函数的编写，就能够实现多路不同频率信号的输出。

我们所学的pwm原理对应的是PWM Generation模式

原文链接：[https://blog.csdn.net/weixin\\_47042449/article/details/122619370](https://blog.csdn.net/weixin_47042449/article/details/122619370)



可以看到对应的引脚已经自动配置好了



ARR与PSC的配置与计算也是一个道理，如图，pwm的频率即为1000Hz，此时当CCR==500时占空比为50%（ccr不在此处配，只能用代码改）



**110: PWM模式1**— 在向上计数时，一旦TIMx\_CNT<TIMx\_CCR1时通道1为有效电平，否则为无效电平；在向下计数时，一旦TIMx\_CNT>TIMx\_CCR1时通道1为无效电平(OC1REF=0)，否则为有效电平(OC1REF=1)。

**111: PWM模式2**— 在向上计数时，一旦TIMx\_CNT<TIMx\_CCR1时通道1为无效电平，否则为有效电平；在向下计数时，一旦TIMx\_CNT>TIMx\_CCR1时通道1为有效电平，否则为无效电平。

常用函数：

```
/**
 * 功能：启动pwm （输出pwm一定得写上这一句）
 * 参数1： htim 定时器句柄，如 &htim1
 * 参数2： 通道数 如TIM_CHANNEL_1
 */
HAL_StatusTypeDef HAL_TIM_PWM_Start(TIM_HandleTypeDef *htim, uint32_t Channel) ;

/**
 * 功能： 停止pwm
 * 参数1： htim 定时器句柄，如 &htim1
 * 参数2： 通道数 如TIM_CHANNEL_1
 */
HAL_StatusTypeDef HAL_TIM_PWM_Stop(TIM_HandleTypeDef *htim, uint32_t Channel);

/**(宏定义)
 * 功能：修改CCR的值
 * 参数1： htim 定时器句柄，如 &htim1
 * 参数2： 通道数： TIM_CHANNEL_1
 * 参数3： 所要设置的ccr值
 */
__HAL_TIM_SET_COMPARE(__HANDLE__, __CHANNEL__, __COMPARE__);

//ps.修改ARR的语句与上一样噢
```

使用示例

\*注意 HAL\_TIM\_Base\_Start 仍然要写噢，因为pwm还是需要时基的

```
MX_ADC1_Init();
/* USER CODE BEGIN 2 */
HAL_TIM_Base_Start(&htim1);
HAL_TIM_PWM_Start(&htim1,TIM_CHANNEL_1);
/* USER CODE END 2 */
```

设置一下ccr就可以获得任意占空比的方波信号啦~~



```

97      /* Infinite loop */
98      /* USER CODE BEGIN WHILE */
99      while (1)
100     {
101
102         /* USER CODE END WHILE */
103
104         /* USER CODE BEGIN 3 */
105
106         __HAL_TIM_SetCompare(&htim1,TIM_CHANNEL_1,125);
107     }
108     /* USER CODE END 3 */
109 }
110

```

那么请大家思考一下呼吸灯如何用pwm波实现叭

----->

```

04      /* USER CODE BEGIN WHILE */
05      while (1)
06      {
07          /* USER CODE END WHILE */
08
09          /* USER CODE BEGIN 3 */
10
11          for(cnt=0;cnt<100;cnt++)
12          {
13              __HAL_TIM_SET_COMPARE(&htim3,TIM_CHANNEL_1,cnt);
14              HAL_Delay(10);
15          }
16          for(cnt=99;cnt>0;cnt--)
17          {
18              __HAL_TIM_SET_COMPARE(&htim3,TIM_CHANNEL_1,cnt);
19              HAL_Delay(10);
20          }
21      }

```

- 易错点提醒
- 如果发现led长亮，可能是延迟的问题噢，因为就算是cnt从0自增到100，中间的时间间隔也非常非常短，会让人眼产生这是长亮的错觉
- cnt的变量类型是需要留心的，如果cnt是uint类型，那么cnt自减的时候边界再写cnt>=0就不合适了。为什么嘞？

以及上述两个示例只是TIM应用的其中两种，除此之外TIM还有许许多多多的功能，如编码器模式（下节课就会用到啦）与输入捕获等等，只是时间有限我们没办法(可能也没必要)拓展太多，希望大家在以后遇到相应的需求的时候可以想起来还有一个功能如此强大的外设丫

(祝大家玩得开心`)

附参考资料：

[STM32的5个时钟源知识 - 知乎 (zhihu.com)]([https://zhuanlan.zhihu.com/p/138661512#:~:text=高速外部时钟\(HS](https://zhuanlan.zhihu.com/p/138661512#:~:text=高速外部时钟(HS))

[csdn Z小璇]<https://blog.csdn.net/as480133937/article/details/98845509>