STM32的通用定时器

通用定时器功能特点描述

STM32的通用定时器是由一个可编程预分频器（PSC）驱动的16位自动重装载计数器（CNT）构成，可用于测量输入脉冲长度（输入捕获）或者产生输出波形（输出比较和PWM）等。

- **位于低速的APB1总线上（注意：高级定时器是在高速的APB2总线上）**；
- **16位向上、向下、向上/向下（中心对齐）计数模式**，自动装载计数器（TIMx_CNT）；
- **16位**可编程（可以实时修改）预分频器（TIMx_PSC），**计数器时钟频率的分频系数 为 1～65535 之间的任意数值；**
- **4 个独立通道（TIMx_CH1~4）**，这些通道可以用来作为：
    1. 输入捕获
    2. 输出比较

3. PWM生成（边缘或中间对齐模式）
4. 单脉冲模式输出

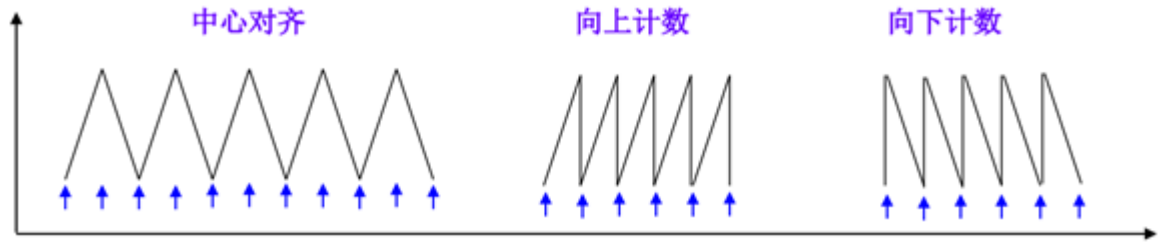
- **STM32 的每个通用定时器都是完全独立的，没有互相共享的任何资源。**

计数器模式

通用定时器可以向上计数、向下计数、向上向下双向计数模式。

向上计数模式：计数器从0计数到自动加载值（TIMx_ARR），然后重新从0开始计数并且产生一个计数器溢出事件。向下计数模式：计数器从自动装入的值（TIMx_ARR）开始向下计数到0，然后从自动装入的值重新开始，并产生一个计数器向下溢出事件。 中央对齐模式（向上/向下计数）：计数器从0开始计数到自动装入的值-1，产生一个计数器溢出事件，然后向下计数到1并且产生一个计数器溢出事件；然后再从0开始重新计数。

可以借助于这个图来理解这三种模式：



三个主要的寄存器：

- 计数器（TIMx_CNT）：存放计数器的当前值。
- 预分频器(TIMx_PSC)：对CK_PSC进行预分频。此时需要注意：**CK_CNT计算的时候，预分频系数要+1。**
- 自动重装载寄存器(TIMx_ARR)：包含将要被传送至实际的自动重装载寄存器的数值。(有点绕口，后面慢慢理解其含义)

通用定时器超时时间

Tout=（ARR+1)(PSC+1)/TIMxCLK

其中：Tout的单位为us，TIMxCLK的单位为MHz。

通用定时器相关配置库函数

- 1个初始化函数

```
HAL_StatusTypeDef HAL_TIM_Base_Init(TIM_HandleTypeDef *htim);
```

作用：用于对**预分频系数、计数方式、自动重装载计数值**、时钟分频因子等参数的设置。

- 2个使能函数

```
__HAL_TIM_ENABLE(__HANDLE__) ; //使能定时器
__HAL_TIM_ENABLE_IT(__HANDLE__, __INTERRUPT__); //使能定时器中断
```

作用：前者使能定时器，后者使能定时器中断。

- 4个状态标志位获取函数

```
__HAL_TIM_GET_FLAG(__HANDLE__, __FLAG__); //check中断标志位（返回0/1）
__HAL_TIM_CLEAR_FLAG(__HANDLE__, __FLAG__); //clear中断标志位（返回0/1）
__HAL_TIM_GET_IT_SOURCE(__HANDLE__, __INTERRUPT__); //check中断是否发生（返回SET or RESET）
__HAL_TIM_CLEAR_IT(__HANDLE__, __INTERRUPT__);//clear 中断标志位（无返回值）
```

作用：**前两者获取（或清除）状态标志位，后两者为获取（或清除）中断状态标志位。**

- ```
  __HAL_TIM_SET_COMPARE(__HANDLE__, __CHANNEL__, __COMPARE__);//设置CC寄存器的值
  __HAL_TIM_GET_COMPARE(__HANDLE__, __CHANNEL__); //获取CC寄存器的值
  __HAL_TIM_SET_COUNTER(__HANDLE__, __COUNTER__); //设置计数寄存器的值
  __HAL_TIM_GET_COUNTER(__HANDLE__); //获取计数寄存器当前值
  __HAL_TIM_SET_AUTORELOAD(__HANDLE__, __AUTORELOAD__); //设置自动重装载寄存器的值
  __HAL_TIM_GET_AUTORELOAD(__HANDLE__); //获取自动重装载寄存器的值
  ```

- ```
  __HAL_TIM_SET_CAPTUREPOLARITY(__HANDLE__, __CHANNEL__, __POLARITY__);//设置捕获模式（上升沿/下降沿/边沿）
  ```

- IT/DMA回调函数

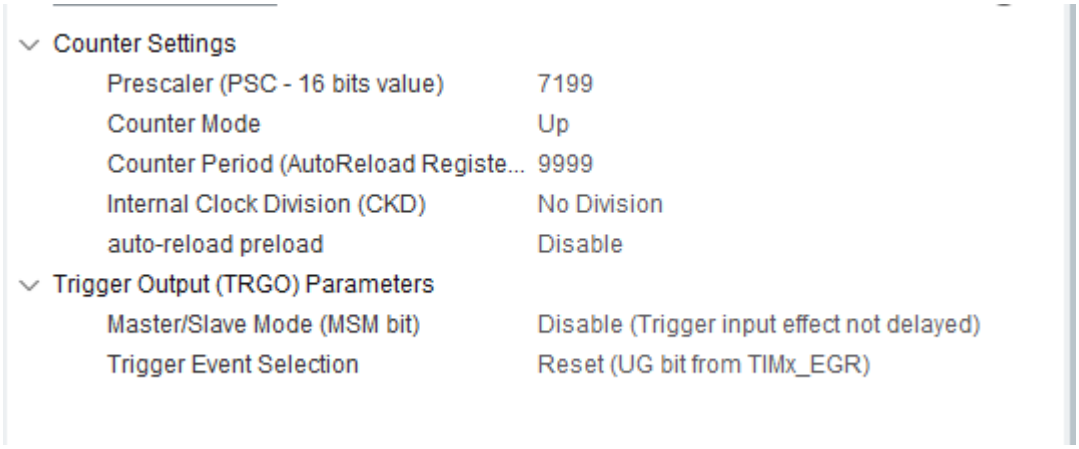
```
/* Callback in non blocking modes (Interrupt and DMA) *************************/
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim);
void HAL_TIM_OC_DelayElapsedCallback(TIM_HandleTypeDef *htim);
void HAL_TIM_IC_CaptureCallback(TIM_HandleTypeDef *htim);
void HAL_TIM_PWM_PulseFinishedCallback(TIM_HandleTypeDef *htim);
void HAL_TIM_TriggerCallback(TIM_HandleTypeDef *htim);
void HAL_TIM_ErrorCallback(TIM_HandleTypeDef *htim);
/**
```

TIM定时器中断实验

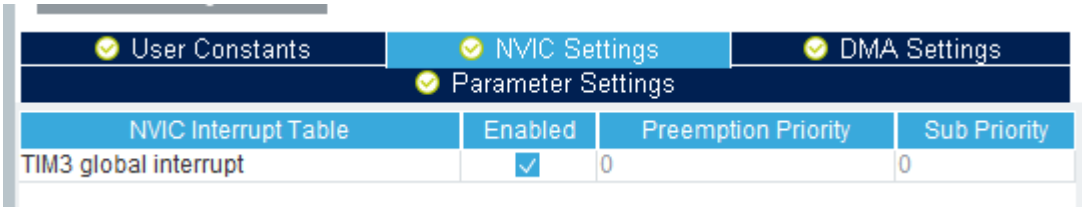
Step1、配置RCC、SYS，配置时钟。

Step2、开启TIM3定时器。

Step3、设置参数psc、arr。



Step4、开启定时器中断。



Step5、生成代码。

Step6、在主函数中加入这一行。

```
HAL_TIM_Base_Start_IT(&htim3);//开启定时器中断
```

Step7、在图示区域加上周期中断回调函数。

```
/* USER CODE BEGIN 4 */
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
  if(htim==&htim3)
  {
    HAL_GPIO_TogglePin(GPIOC,GPIO_PIN_13);
  }
}
/* USER CODE END 4 */
```

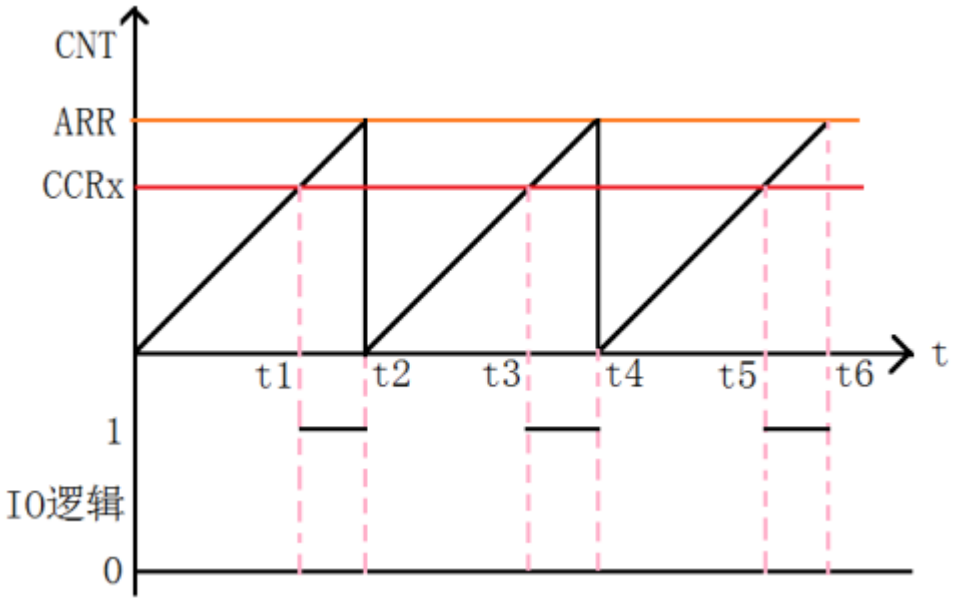
这样就可以看到PC13在以1s的频率闪烁。

TIM定时器实现PWM输出

STM32的定时器除了TIM6和TIM7（基本定时器）之外，其他的定时器都可以产生PWM输出。

TIM产生PWM波的原理

下面以向上计数为例，简单地讲述一下PWM的工作原理：



- 在PWM输出模式下，除了CNT（计数器当前值）、ARR（自动重装载值）之外，还多了一个值CCRx（捕获/比较寄存器值）。
- 当CNT小于CCRx时，TIMx_CHx通道输出低电平；
- 当CNT等于或大于CCRx时，TIMx_CHx通道输出高电平。

这个时候就可以对其下一个准确的定义了：**所谓脉冲宽度调制模式（PWM模式），就是可以产生一个由TIMx_ARR寄存器确定频率，由TIMx_CCRx寄存器确定占空比的信号。它是利用微处理器的数字输出来对模拟电路进行控制的一种非常有效的技术。**

占空比计算公式为：$q = \frac{TIMx->CCR2}{TIM->ARR}$

与PWM产生有关的寄存器

CCRx寄存器：捕获 /比较值寄存器：设置比较值； CCMRx寄存器：OCxM[2:0]位：对于PWM方式下，用于设置PWM模式1或者PWM模式2； CCER寄存器：CCxP位：输出极性。0：高电平有效，1：低电平有效。 CCER寄存器：CCxE位：输出使能。0：关闭，1：打开。

PWM输出的模式区别

通过设置寄存器TIMx_CCMR1的OC1M[2:0]位来确定PWM的输出模式：

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)。 **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。

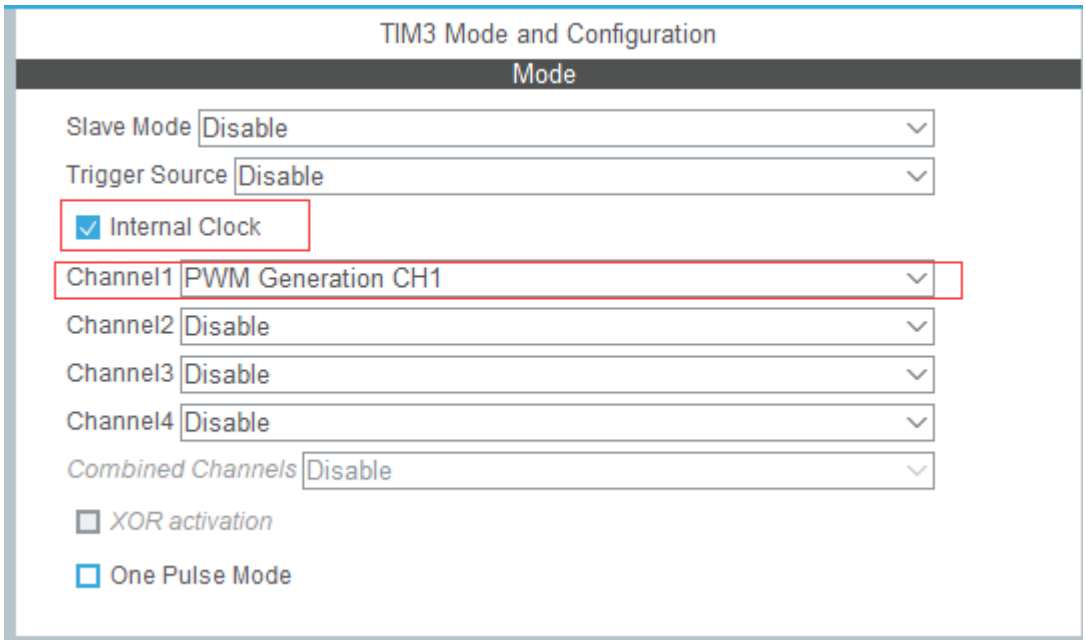
注意：**PWM的模式只是区别什么时候是有效电平，但并没有确定是高电平有效还是低电平有效。这需要结合CCER寄存器的CCxP位的值来确定。**

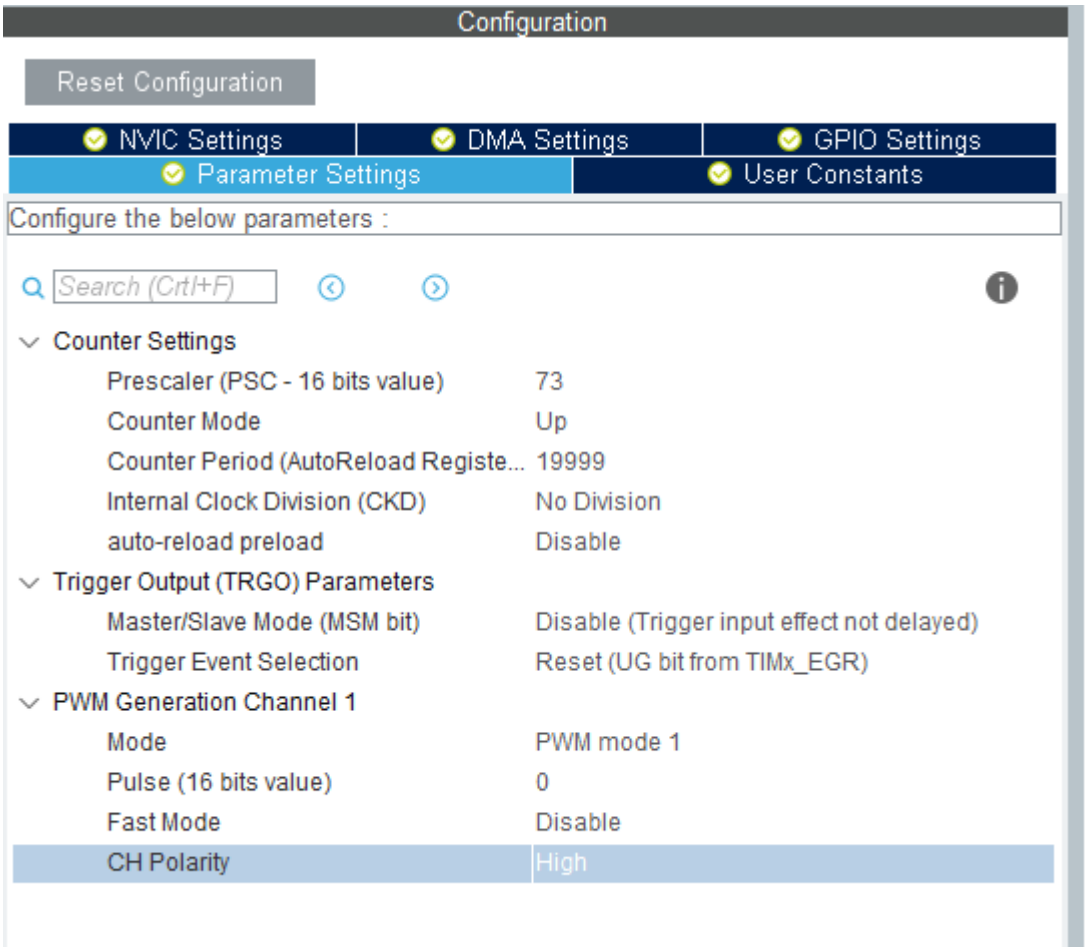
> *例如：若PWM模式1，且CCER寄存器的CCxP位为0，则当TIMx_CNT<TIMx_CCR1时，输出高电平；同样的，若PWM模式1，且CCER寄存器的CCxP位为2，则当TIMx_CNT<TIMx_CCR1时，输出低电平。*

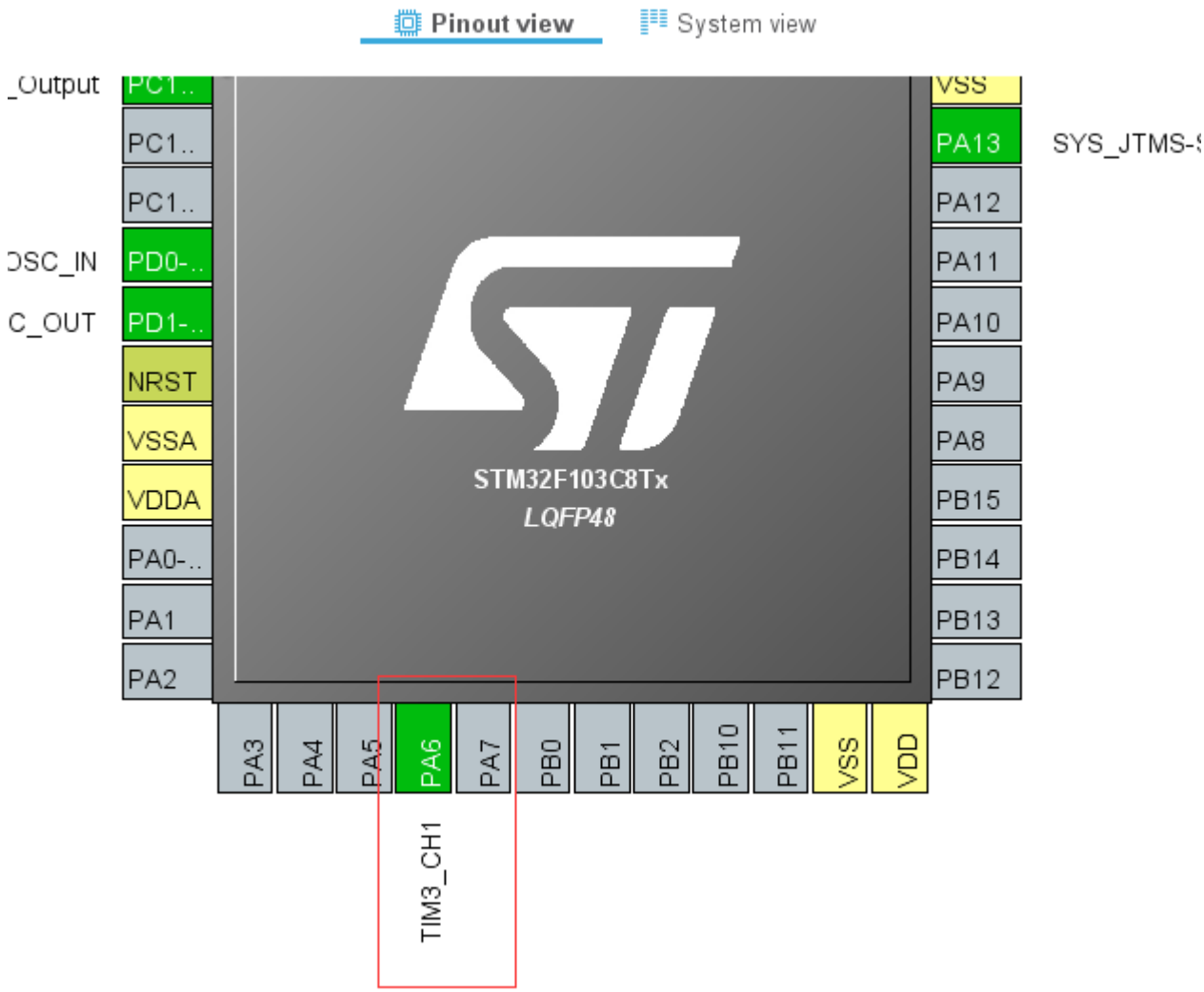
TIM定时器产生PWM波实验

操作步骤：


TIM2
TIM3
TIM4


TIM3 Mode and Configuration
Mode
Slave Mode Disable
Trigger Source Disable
Internal Clock
Channel1 PWM Generation CH1
Channel2 Disable
Channel3 Disable
Channel4 Disable
Combined Channels Disable
XOR activation
One Pulse Mode


Configuration
Reset Configuration
NVIC Settings
DMA Settings
GPIO Settings
Parameter Settings
User Constants
Configure the below parameters :
Search (Crtl+F)
Counter Settings
Prescaler (PSC - 16 bits value) 73
Counter Mode Up
Counter Period (AutoReload Registe... 19999
Internal Clock Division (CKD) No Division
auto-reload preload Disable
Trigger Output (TRGO) Parameters
Master/Slave Mode (MSM bit) Disable (Trigger input effect not delayed)
Trigger Event Selection Reset (UG bit from TIMx_EGR)
PWM Generation Channel 1
Mode PWM mode 1
Pulse (16 bits value) 0
Fast Mode Disable
CH Polarity High



生成工程模板，编写代码如下，可以实现呼吸灯：

```c
int main(void)
{
  /* USER CODE BEGIN 1 */
  int pwm_val=0;
  /* USER CODE END 1 */

  /* MCU Configuration--------------------------------------------------------

  /* Reset of all peripherals, Initializes the Flash interface and the Syst
  HAL_Init();

  /* USER CODE BEGIN Init */

  /* USER CODE END Init */

  /* Configure the system clock */
  SystemClock_Config();

  /* USER CODE BEGIN SysInit */

  /* USER CODE END SysInit */

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_TIM3_Init();
  /* USER CODE BEGIN 2 */
  HAL_TIM_PWM_Start(&htim3,TIM_CHANNEL_1);//开启pwm通道
  TIM3->ARR=19999;//脉冲频率为20ms
  /* USER CODE END 2 */

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
    while(pwm_val<19999)
    {pwm_val+=1000;
      TIM3->CCR1=pwm_val;//改变脉宽
      HAL_Delay(1000);
    }
    while(pwm_val)
    {pwm_val-=1000;
      TIM3->CCR1=pwm_val;
      HAL_Delay(1000);
    }
```

TIM定时器实现输入捕获

输入捕获模式可以用来测量脉冲宽度或者测量频率。STM32的定时器，除了TIM6、TIM7，其他的定时器都有输入捕获的功能。下面以一个简单的脉冲输入为例，简单地讲述一下输入捕获用于测量脉冲宽度的工作原理：

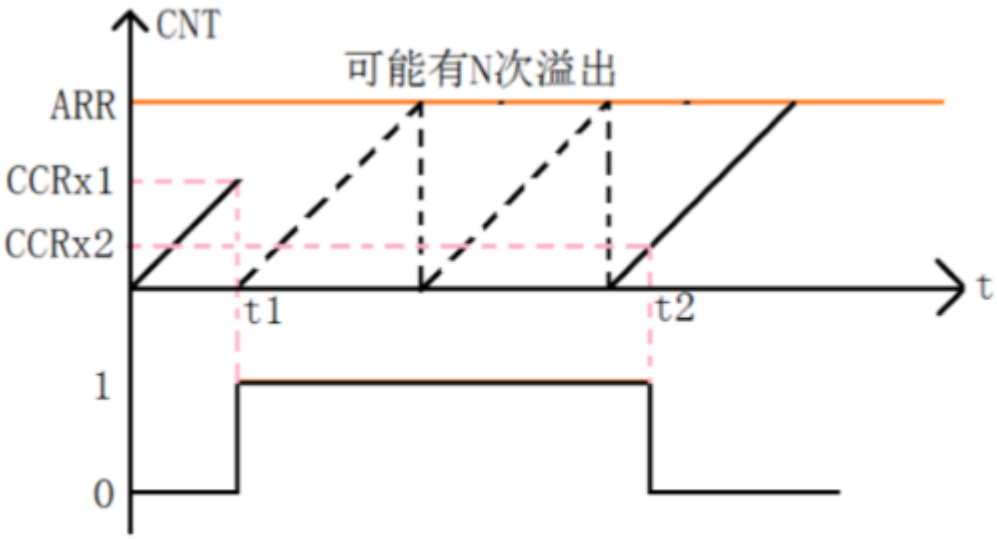
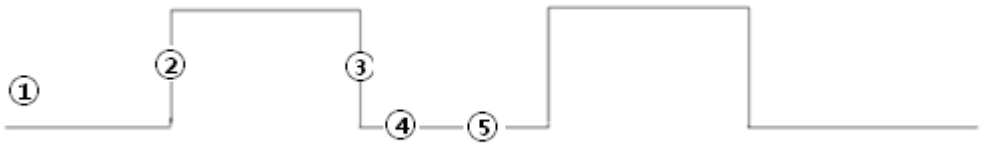


图 14.1.1 输入捕获脉宽测量原理



如图 14.1.1 所示，就是输入捕获测量高电平脉宽的原理，假定定时器工作在向上计数模式， 图中 t1~t2 时间，就是我们需要测量的高电平时间。

测量方法如下：首先设置定时器通道 x 为 上升沿捕获，这样，t1 时刻，就会捕获到当前的 CNT 值，然后立即清零 CNT，并设置通道 x 为下降沿捕获，这样到 t2 时刻，又会发生捕获事件，得到此时的 CNT 值，记为 CCRx2。这样，根据定时器的计数频率，我们就可以算出 t1~t2 的时间，从而得到高电平脉宽。 在 t1~t2 之间，可能产生 N 次定时器溢出，这就要求我们对定时器溢出做处理，防止高 电平太长，导致数据不准确。如图14.1.1所示，t1~t2之间，CNT计数的次数等于：**N*ARR+CCRx2**， 有了这个计数次数，再乘以 CNT 的计数周期，即可得到 t2-t1 的时间长度，即高电平持续时间。 输入捕获的原理，我们就介绍到这。

同时还可以配置捕获时是否触发中断/DMA 等。

输入捕获相关寄存器

捕获/比较模式寄存器1（TIMx_CCMR1）

作用：在输入捕获模式下，确定**数字滤波器、通道映射、预分频系数。**

捕获/比较使能寄存器·（TIMx_CCER）

作用：在输入捕获模式下，确定**捕捉极性和捕捉使能。**

捕获/比较寄存器1（TIMx_CCR1）

作用：在输入捕获模式下，确定上一次输入捕捉事件传输的计数值。

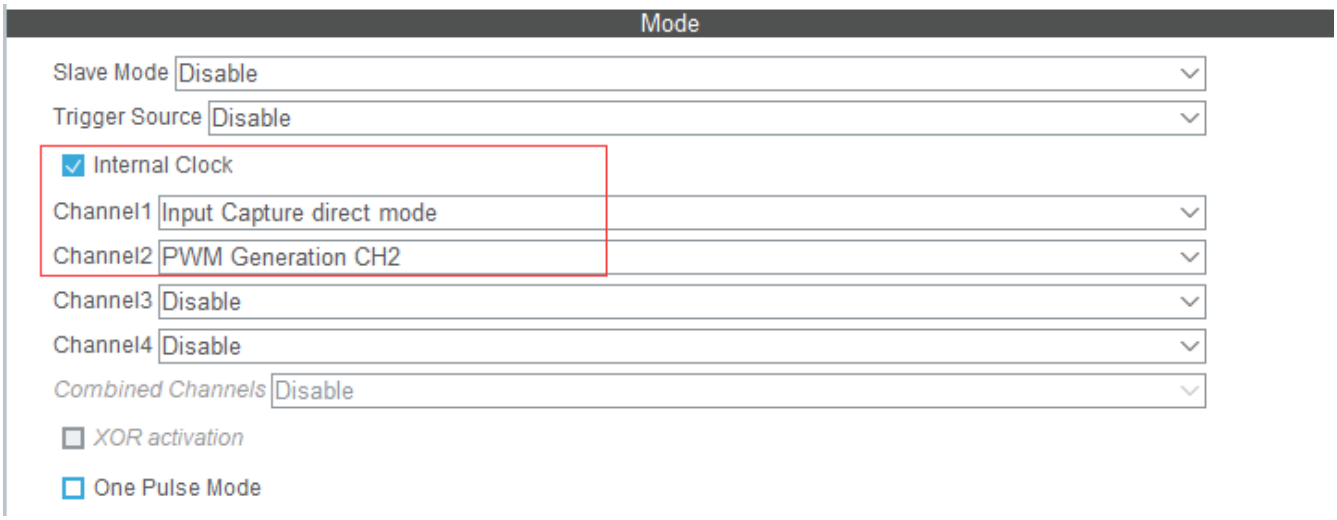
输入捕获相关配置库函数

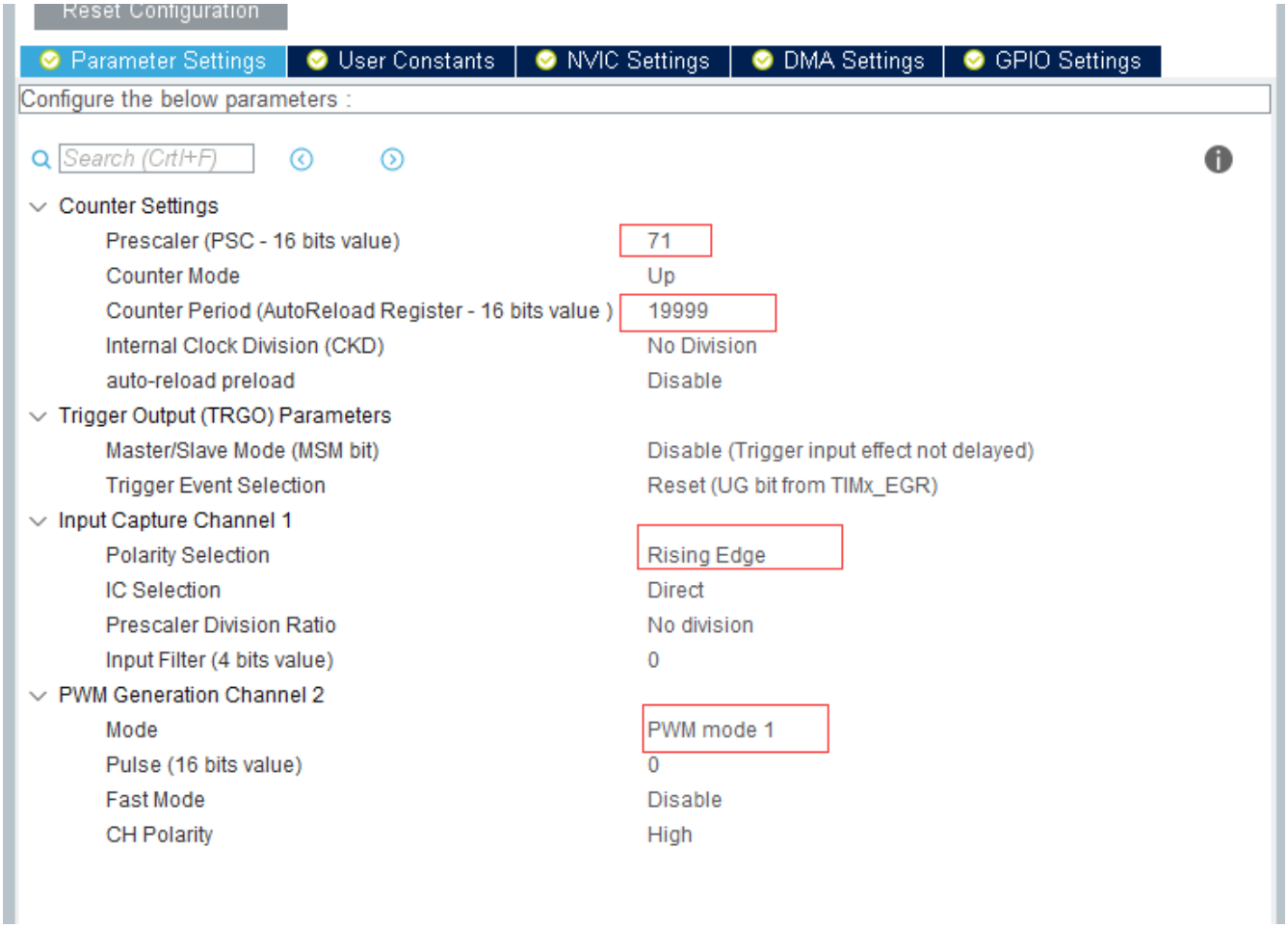
```
/* Timer Input Capture functions **********************************************/
HAL_StatusTypeDef HAL_TIM_IC_Init(TIM_HandleTypeDef *htim);
HAL_StatusTypeDef HAL_TIM_IC_DeInit(TIM_HandleTypeDef *htim);
void HAL_TIM_IC_MspInit(TIM_HandleTypeDef *htim);
void HAL_TIM_IC_MspDeInit(TIM_HandleTypeDef *htim);
/* Blocking mode: Polling */
HAL_StatusTypeDef HAL_TIM_IC_Start(TIM_HandleTypeDef *htim, uint32_t Channel);
HAL_StatusTypeDef HAL_TIM_IC_Stop(TIM_HandleTypeDef *htim, uint32_t Channel);
/* Non-Blocking mode: Interrupt */
HAL_StatusTypeDef HAL_TIM_IC_Start_IT(TIM_HandleTypeDef *htim, uint32_t Channel);
HAL_StatusTypeDef HAL_TIM_IC_Stop_IT(TIM_HandleTypeDef *htim, uint32_t Channel);
/* Non-Blocking mode: DMA */
HAL_StatusTypeDef HAL_TIM_IC_Start_DMA(TIM_HandleTypeDef *htim, uint32_t Channel, uint32_t *pData,
uint16_t Length);
HAL_StatusTypeDef HAL_TIM_IC_Stop_DMA(TIM_HandleTypeDef *htim, uint32_t Channel);
```

```
void HAL_TIM_IC_CaptureCallback(TIM_HandleTypeDef *htim);//输入捕获中断回调函数
```

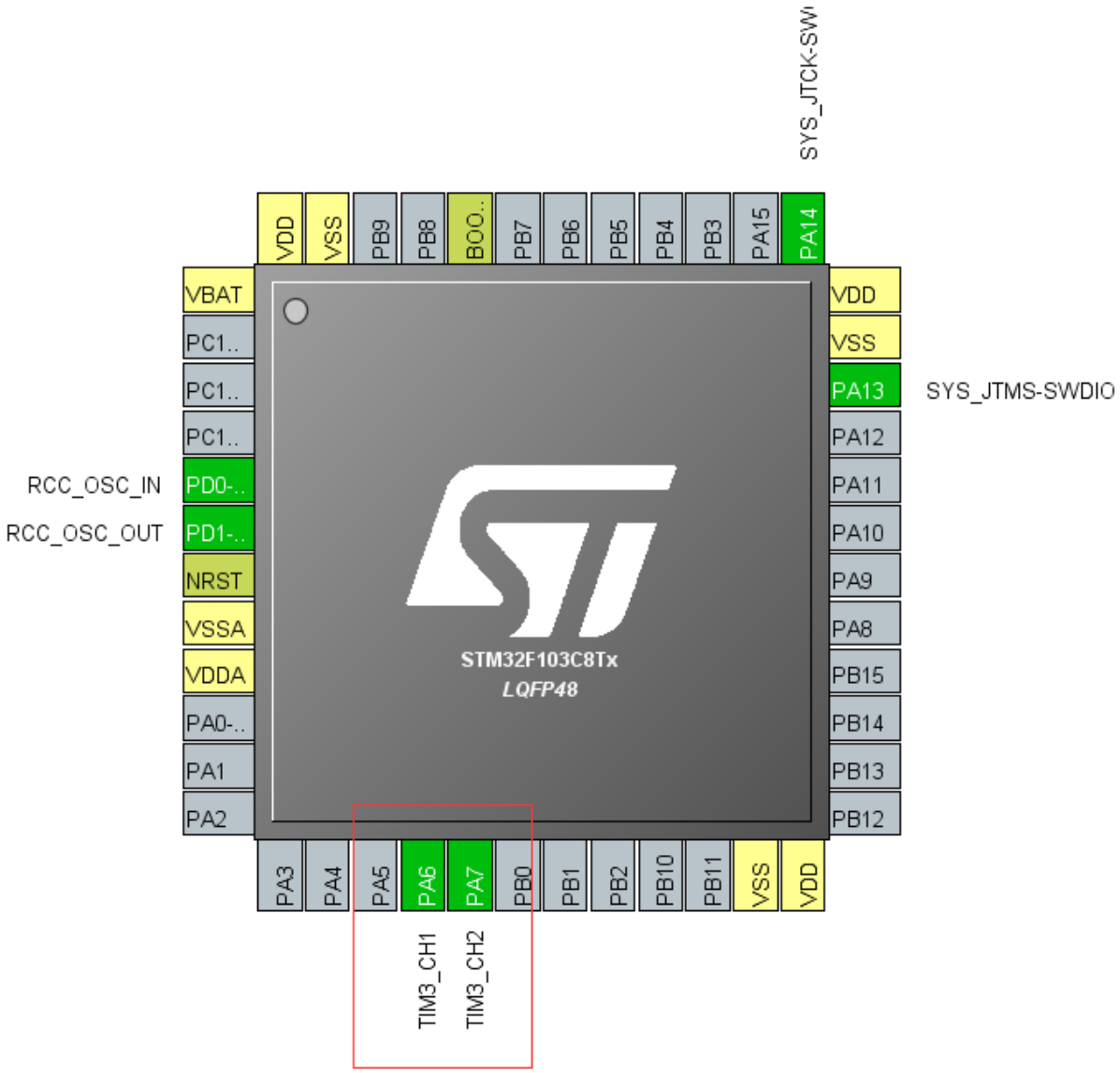
用Cube生成工程

本实验我们利用TIM3的CH1和CH2，CHI设置为IC模式，CH2设置为PWM输出模式。并开启中断。





我们开启TIM3的CH1和CH2通道的同时，Cube帮我们映射到了管脚PA6\PA7上，如图。



Cube主要设置就这么多。

在主函数前加入代码，

```
/* USER CODE BEGIN Includes */
int pwm_val=0;
int high_time,IC_flag,IC_buf[2];
/* USER CODE END Includes */
```

在初始化函数之后加入，

```
/* USER CODE BEGIN 2 */
HAL_TIM_IC_Start_IT(&htim3,TIM_CHANNEL_1);
HAL_TIM_PWM_Start(&htim3,TIM_CHANNEL_2);
/* USER CODE END 2 */

/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{
  while(pwm_val<19999)
    {
      pwm_val+=1000;
      TIM3->CCR2=pwm_val;
      HAL_Delay(1000);
    }
   while(pwm_val)
    {
      pwm_val-=1000;
      TIM3->CCR2=pwm_val;
      HAL_Delay(1000);
    }
  /* USER CODE END WHILE */
```

在图示区域加上，

```
/* USER CODE BEGIN 4 */
void HAL_TIM_IC_CaptureCallback(TIM_HandleTypeDef *htim)//输入捕获中断回调函数
{
  if(IC_flag==0)//我用IC_flag==0表示当前是上升沿触发
  {
    IC_buf[0]=HAL_TIM_ReadCapturedValue(&htim3,TIM_CHANNEL_1);//记录CCR1的值
    TIM_SET_CAPTUREPOLARITY(&htim3,TIM_CHANNEL_1,TIM_INPUTCHANNELPOLARITY_FALLING);//改变触发极性
    IC_flag=1;//我用IC_lag==1表示当前是下降沿触发
  }
  else
  {
    IC_buf[1]=HAL_TIM_ReadCapturedValue(&htim3,TIM_CHANNEL_1);//记录CCR1的值
    if(IC_buf[1]>IC_buf[0])
    {
      high_time=IC_buf[1]-IC_buf[0];//计算高电平时间
    }
    else
    {
      high_time=IC_buf[1]-IC_buf[0]+20000;
    }
  }
}
/* USER CODE END 4 */
```

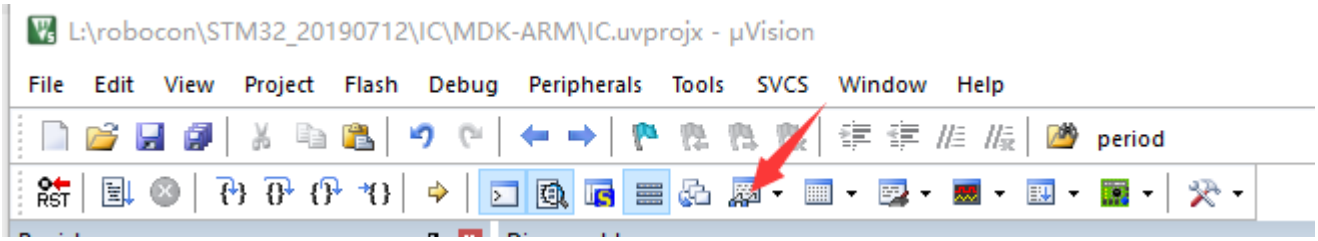
代码撰写完毕。

然后将PA6和PA7连接在一起。

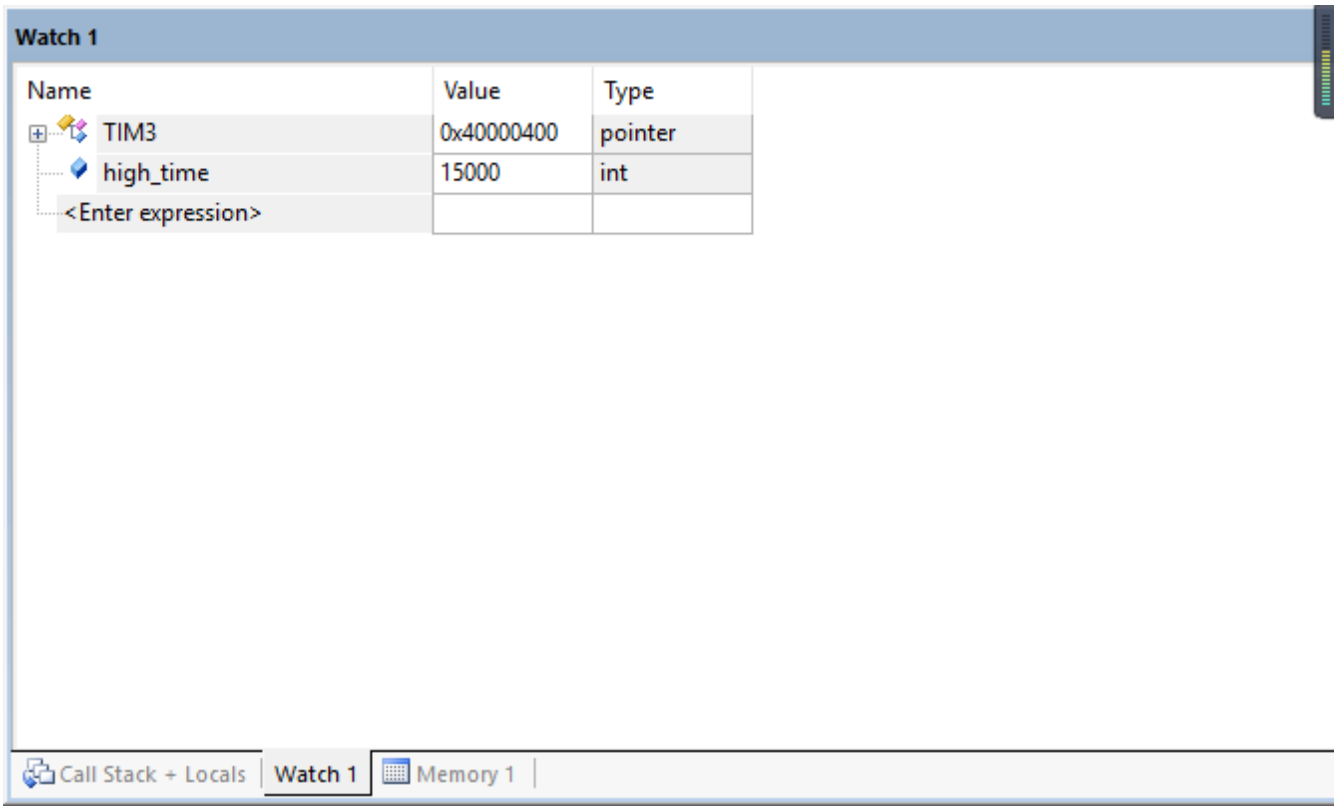
下载代码，打开debugging窗口，如图。

Step1：点击箭头指向图标，进入调试状态。

Step2：点击箭头指向图标，调出窗口监视器。



Step3、向watch window添加想要看的变量。



Step4、点击运行按钮，图标灰色表示代码开始运行。

这时可以看到监视器中变量会随着代码运行动态变化。思考是不是变化是否和代码契合?

ADC

Analog-to-Digital Converter的缩写。指模/数转换器或者模拟/数字转换器。是指将**连续变量的模拟信号转换为离散的数字信号的器件。**

STM32的ADC， 是 **12 位逐次逼近型**的模数转换器。ADC 的结果可以左对齐或右对齐方式存储在 16 位数据寄存器中。

STM32将 ADC 的转换分为 2 个通道组：**规则通道组**和**注入通道组**。规则通道相当于 你正常运行的程序，而注入通道呢，就相当于中断。

STM32的ADC的各通道可以组成规则通道组或注入通道组，但是在转换方式还可以有**单次转换、连续转换、扫描转换模式。**至于他们的不同，感兴趣的话上网搜。我们一般都是采用连续转换模式。

有关ADC主要的hal库函数：

```
HAL_StatusTypeDef HAL_ADC_Start(ADC_HandleTypeDef* hadc);//使能ADC
HAL_StatusTypeDef HAL_ADC_PollForConversion(ADC_HandleTypeDef* hadc, uint32_t Timeout); //等待ADC转换完成
uint32_t HAL_ADC_GetValue(ADC_HandleTypeDef* hadc); //获取ADC转换的结果
```

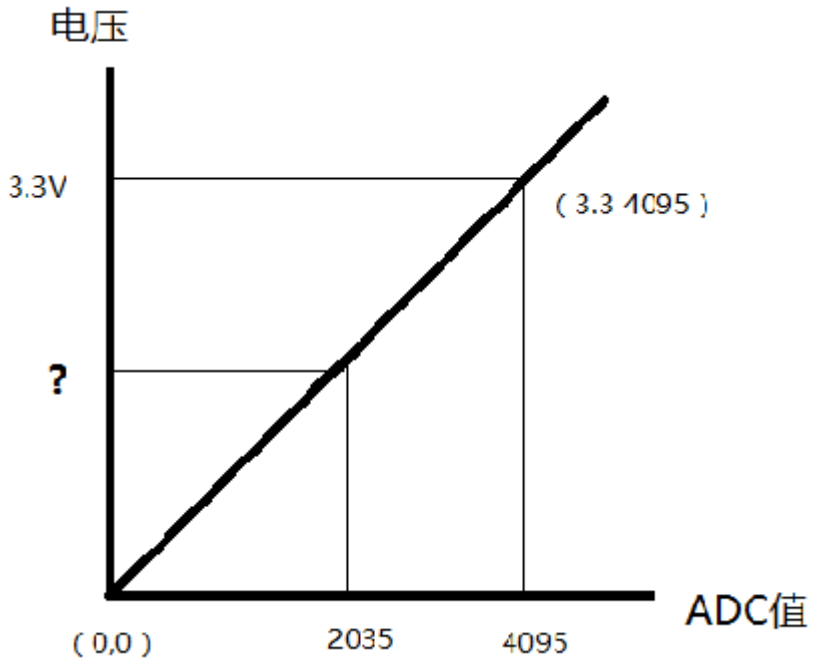
ADC采样会得到什么值

由于STM32的ADC是12位逐次逼近型的模拟数字转换器，也就是说ADC模块读到的数据是12位的数据。

因此：**STM32读到的ADC值，是从0到4095（111111111111）。当把ADC引脚接了GND，读到的就是0；当把ADC引脚接了VDD，读到的就是4095。**

读到的值怎么换算成实际的电压值

前面提到了，我们输入GND，读到的值是0，输入VDD，得到的值是4095，那么，当读到2035的时候，怎么求输入电压多少V吗？这个问题，归根接地，就到了数学XY坐标，已知两点坐标值（0,0）（3.3,4095），给出任意X坐标值，求Y值的问题了吧？简单不简单？



参考电压

我们板子默认参考电压是3.3V。

ADC引脚的输入电压范围是多大

0~Vref。

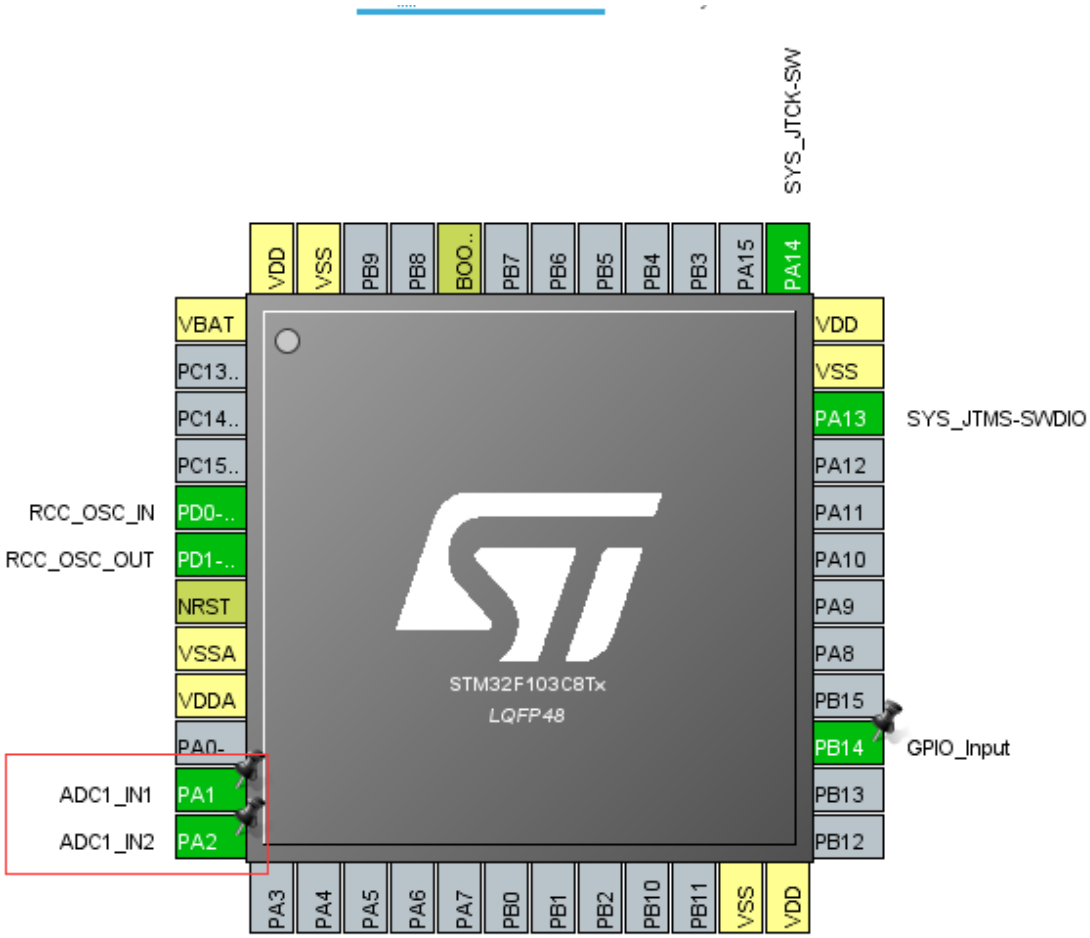
如果待检测电压大于ADC的最大输入电压，可以通过电阻分压减小输入电压。

常用的ADC库函数：

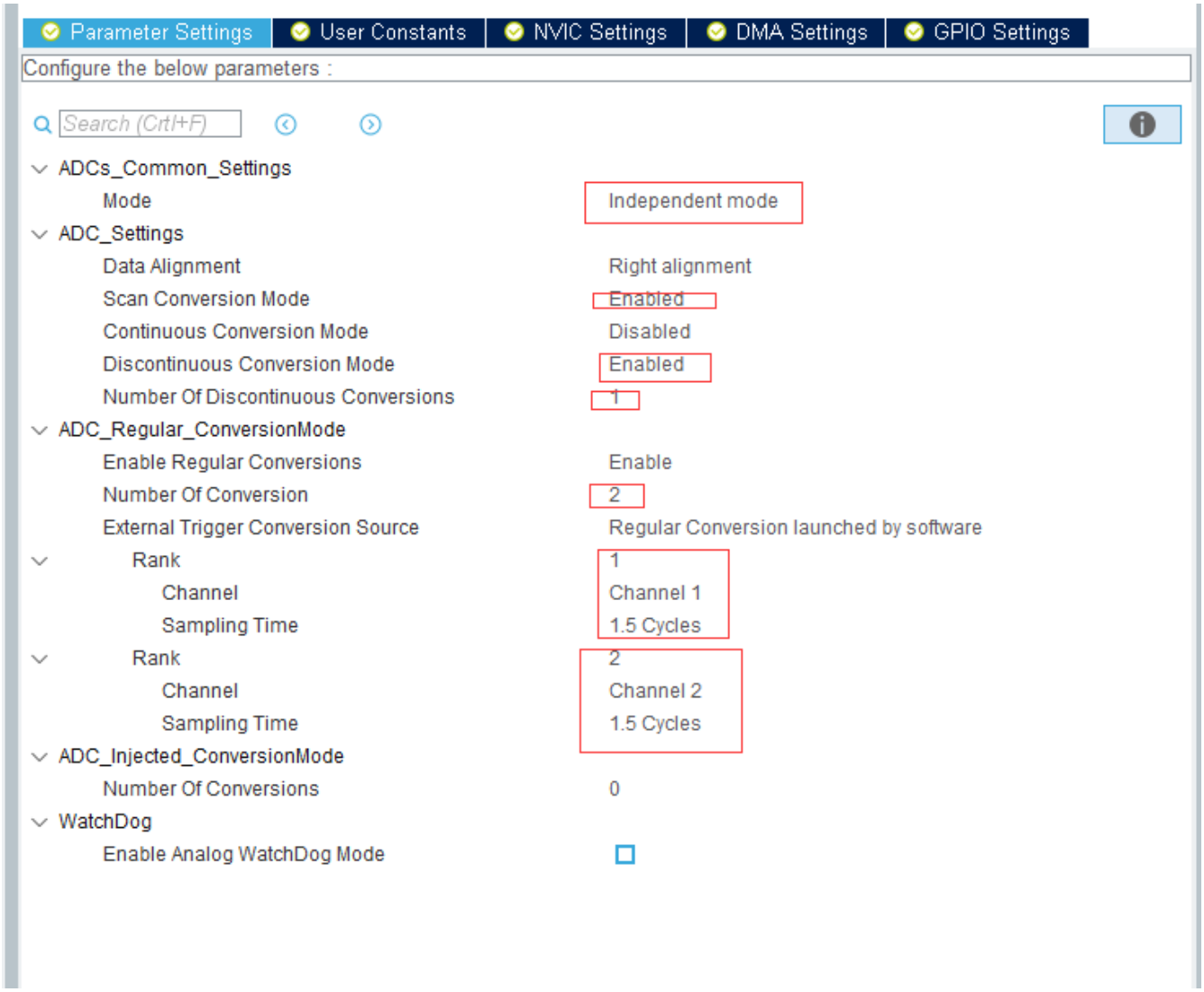
```
HAL_StatusTypeDef        HAL_ADC_Start(ADC_HandleTypeDef* hadc);
HAL_StatusTypeDef        HAL_ADC_Stop(ADC_HandleTypeDef* hadc);
uint32_t                 HAL_ADC_GetValue(ADC_HandleTypeDef* hadc);//获取当前ad转换的值
```

PS2 ADC实验

Step1、开启PA1和PA2为ADC采集通道。



Step2、ADC配置如图。



多路ADC采集时，必须要选用扫描模式和间断模式。

连续模式适用于单路采集。

可以设置优先级Rank，即采集顺序。

设置每次采集一个，分两次采集。

Step3、定义两个全局变量。

```
uint32_t value_x,value_y;
/* USER CODE END Includes */
```

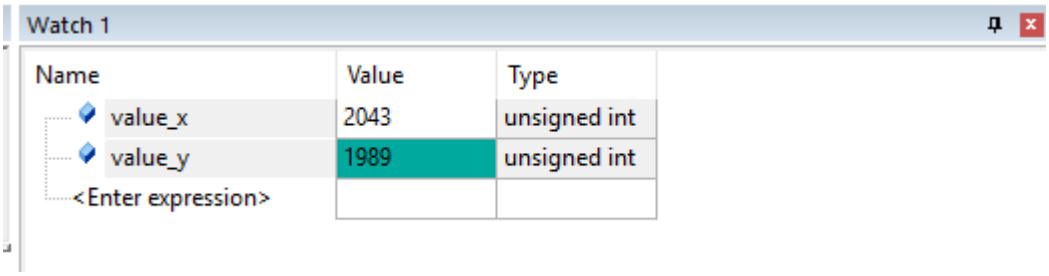
Step4、在主函数while循环中加入如下代码，有注释自己看。

```
/* USER CODE BEGIN WHILE */
while (1)
{
  HAL_ADC_Start(&hadc1);//只能转换一次
  HAL_ADC_PollForConversion(&hadc1,10);//等待ad转换完成
  value_x=HAL_ADC_GetValue(&hadc1);//获取in1ad转换结果
  HAL_ADC_Start(&hadc1);//继续开启转换
  HAL_ADC_PollForConversion(&hadc1,10);//等待ad转换完成
  value_y=HAL_ADC_GetValue(&hadc1);//获取in2ad转换结果
  /* USER CODE END WHILE */

  /* USER CODE BEGIN 3 */
}
/* USER CODE END 3 */
```

每一个通道adc采集需要三行语句。

然后在调试器里面便可以看到value_x和value_y的结果。



DAC

Digital-to-Analog Converter的缩写。指数/模转换器。是指将离散的数字信号转换为连续变量的模拟信号的器件。

STM32的DAC模块是**12位数字输入**，电压输出型的DAC。

STM32F4xx有2个DAC，每个DAC对应一个输出通道。

表29　ADC/DAC

ADC/DAC引脚	GPIO配置
ADC/DAC	模拟输入

为什么要使用模拟输入模式呢？因为一旦使能DACx通道后，相应的GPIO引脚会自动与DAC的模拟输出相连，设置为输入，是为了防止额外的干扰。

DAC数据格式，一般都采用12位数据右对齐，这样便于数据处理。

DAC输出电压

当DAC的参考电压位VREF+的时候，数字输入经过DAC被线性地转换为模拟电压输出，其范围为0到VREF+。

DAC输出 = VREF x (DOR / 4095)。

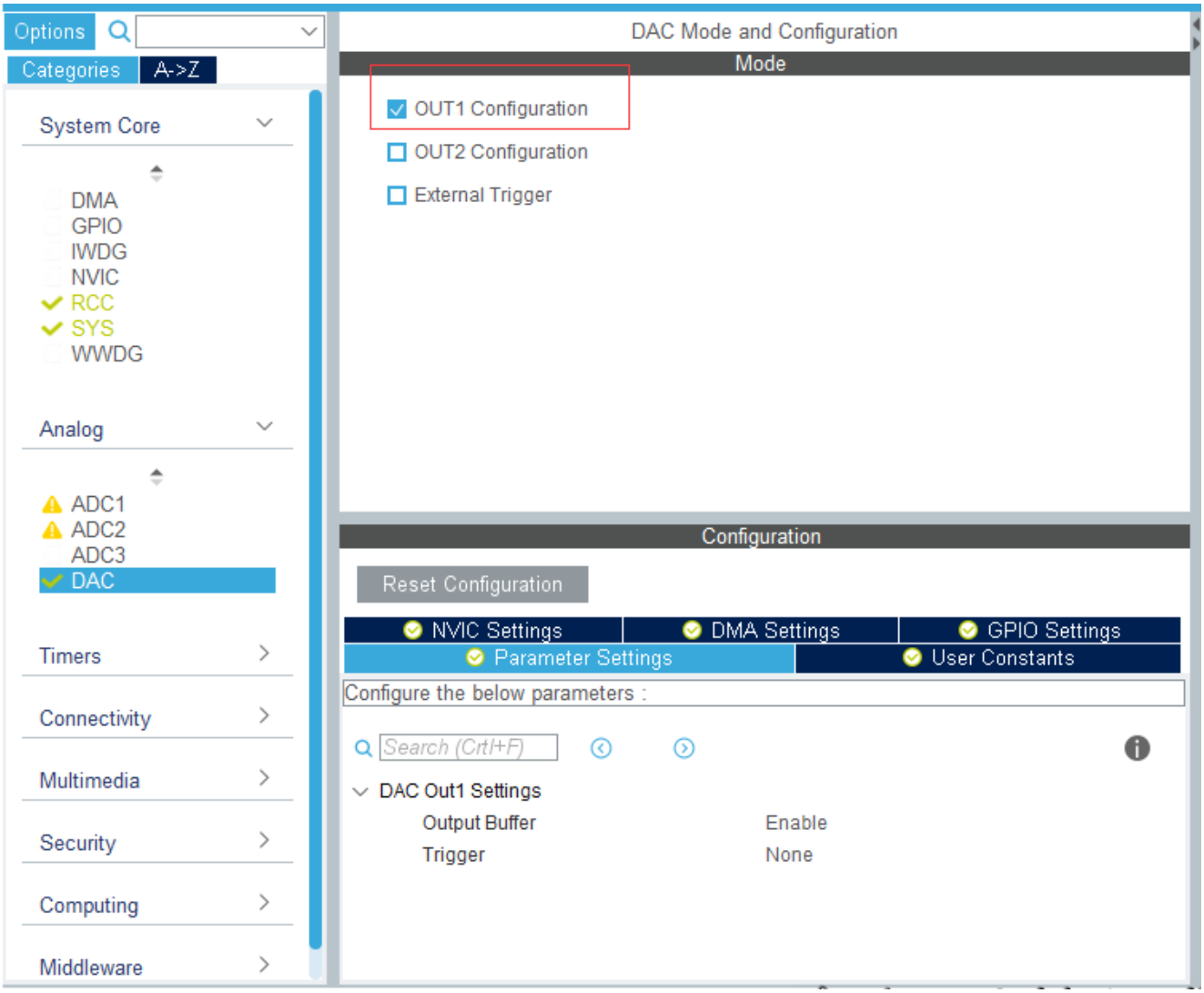
注意：数据格式应该选择12位数据右对齐。

主要用到的函数：

```c
/**
  * @brief  Set the specified data holding register value for DAC channel.
  * @param  hdac pointer to a DAC_HandleTypeDef structure that contains
  *         the configuration information for the specified DAC.
  * @param  Channel The selected DAC channel.
  *          This parameter can be one of the following values:
  *            @arg DAC_CHANNEL_1: DAC Channel1 selected
  *            @arg DAC_CHANNEL_2: DAC Channel2 selected
  * @param  Alignment Specifies the data alignment.
  *          This parameter can be one of the following values:
  *            @arg DAC_ALIGN_8B_R: 8bit right data alignment selected
  *            @arg DAC_ALIGN_12B_L: 12bit left data alignment selected
  *            @arg DAC_ALIGN_12B_R: 12bit right data alignment selected
  * @param  Data Data to be loaded in the selected data holding register.
  * @retval HAL status
  */
HAL_StatusTypeDef HAL_DAC_SetValue(DAC_HandleTypeDef* hdac, uint32_t Channel, uint32_t Alignment, uint32_t Data);
```

DAC输出pwm实验（基于F427IIHx)

Step1、开启DAC，F4系列只有一个DAC，开启DAC的CH1，配置默认。



同时，会发现Cube为我们开好了一个PA4作为该DAC输出引脚。

Step2、进入keil。定义一个变量，作为输入数字量。通过修改数字量的大小，可以得到不同的输出电压。

```
/* USER CODE BEGIN 1 */
  uint32_t value=0;
/* USER CODE END 1 */
```

Step3、开启DAC通道。

```
/* USER CODE BEGIN 2 */
HAL_DAC_Start(&hdac,DAC_CHANNEL_1);//开启dac的1通道
/* USER CODE END 2 */
```

Step4、进行AD转换。

```
/* USER CODE BEGIN WHILE */
while (1)
{
  if(value>4025)//防止超过2^12-1
  {
    value=0;
  }
  HAL_DAC_SetValue(&hdac,DAC_CHANNEL_1,DAC_ALIGN_12B_R,value);//DA转换函数
  value+=100;
  HAL_GPIO_TogglePin(GPIOF,GPIO_PIN_14);
  HAL_Delay(1000);
  /* USER CODE END WHILE */

  /* USER CODE BEGIN 3 */
}
```

至此，代码编写完毕。用万用表测量PA4的引脚可以看到输出电压在0~3.3V之间线性变化，符合预期效果。

DAC的两个通道的赋值是互不干扰的。

独立看门狗IWDG（基于F427）