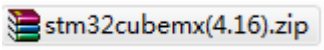
STM32F072-M0 内核操作手册

作者:向仔州

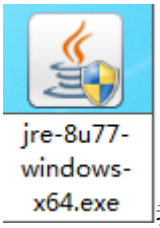
安装 STM32CUBEMX 软件



这个软件就是用图形来生成代码，很方便

安装 cubemx 之前要安装 JAVA 运行环境

去 orcal 官网下载 JAVA 的 jre 包运行环境

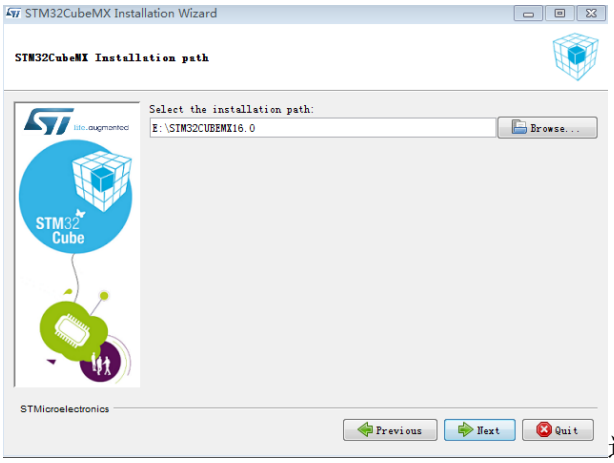


我电脑上 64 位的，所以就安装这个，直接打开软件点击安装就是了

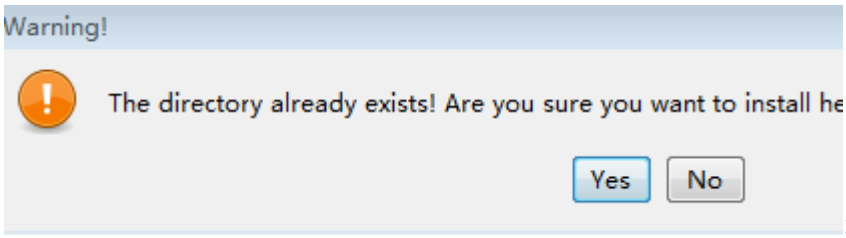
JAVA 运行环境安装完成

安装 cubemx 软件

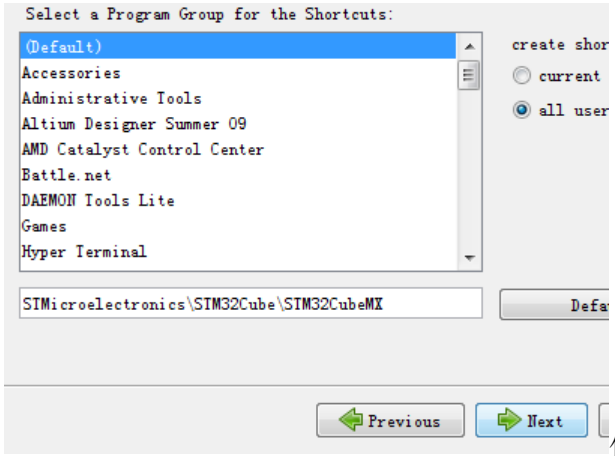
我这里的版本是 16.0，点击我允许协议，然后点击下一步



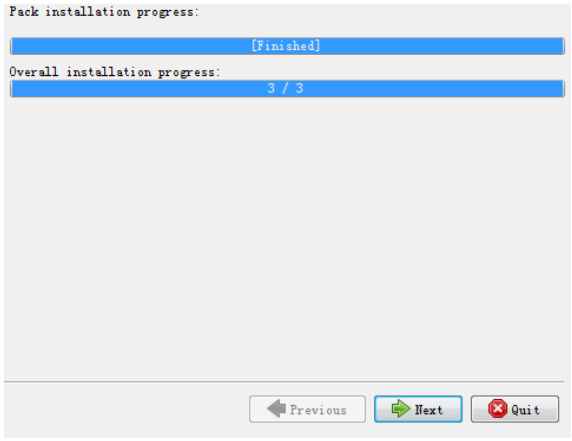
选择软件安装的路径



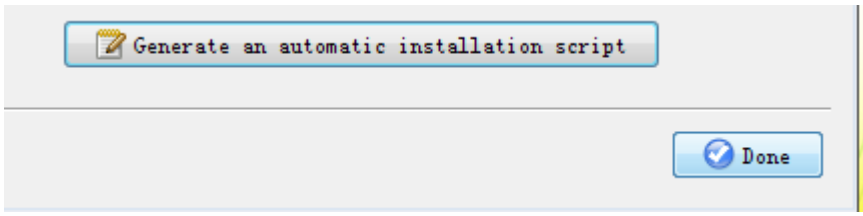
这里选择 yes



什么都不要管直接点击 next



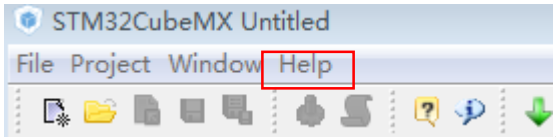
这就是安装完成，点击下一步



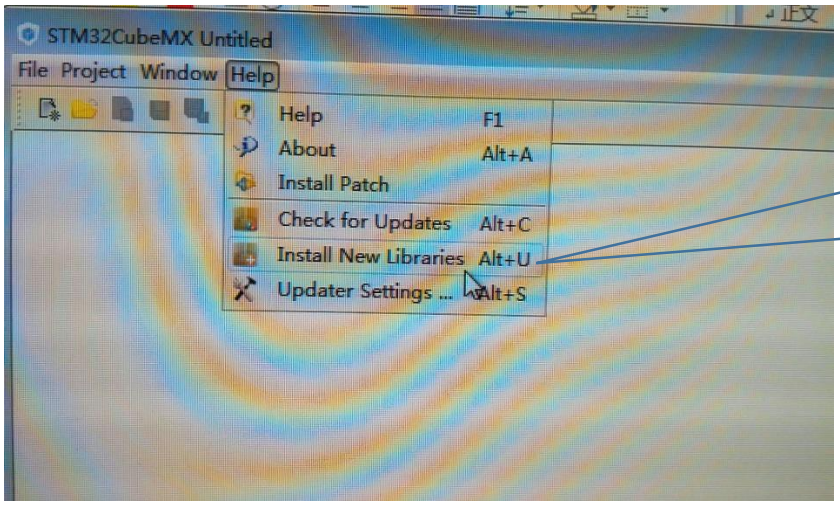
点击 done

这个图标就是 STM32CubeMX 软件

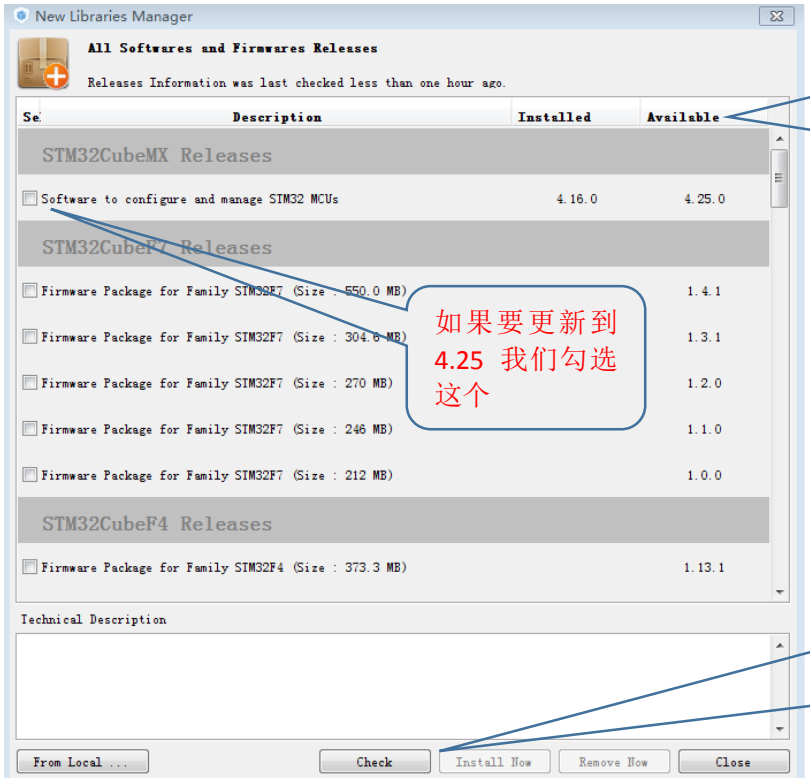
打开 CubeMX 软件



点击 help



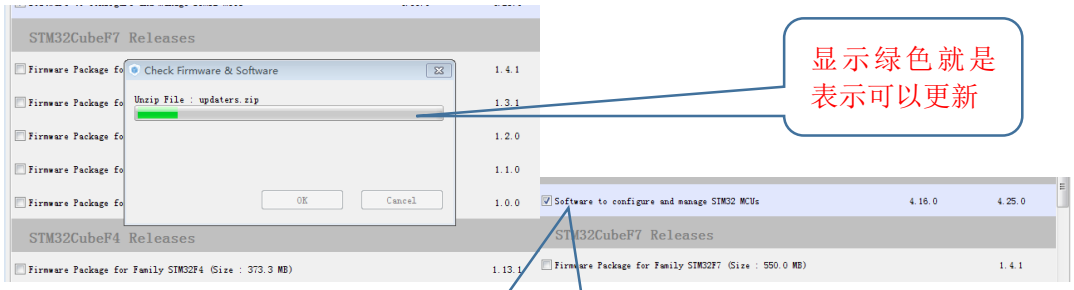
如果你软件不是最新版本可以在这里选择更新



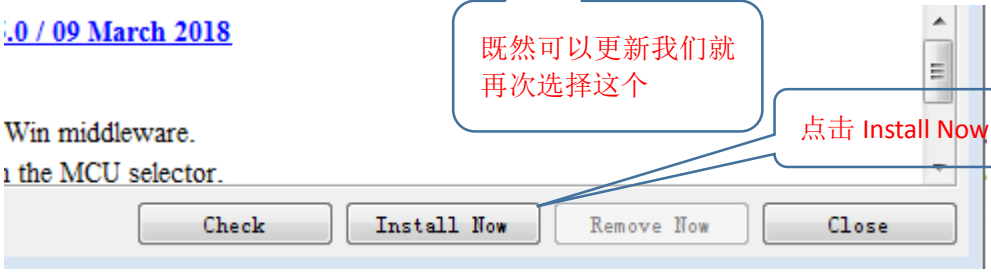
这里提示官方有 4.25 的版本了，但是我们现在安装的是 4.16 的版本

如果要更新到 4.25 我们勾选这个

然后点击 check 检查是否能更新，记住电脑一定要上网

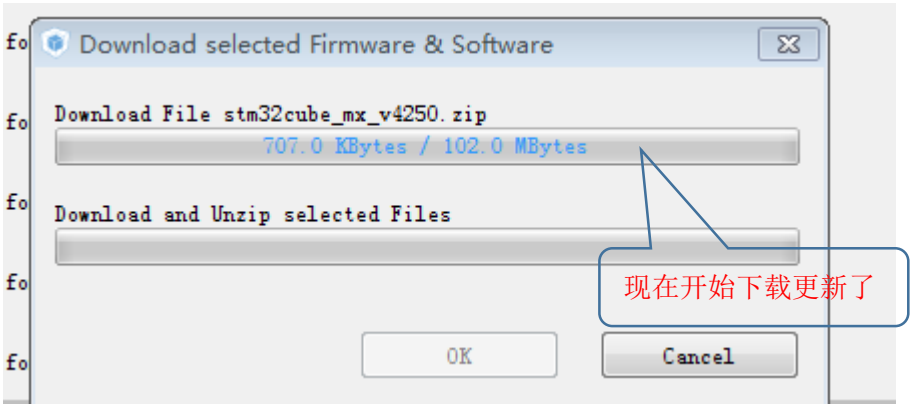


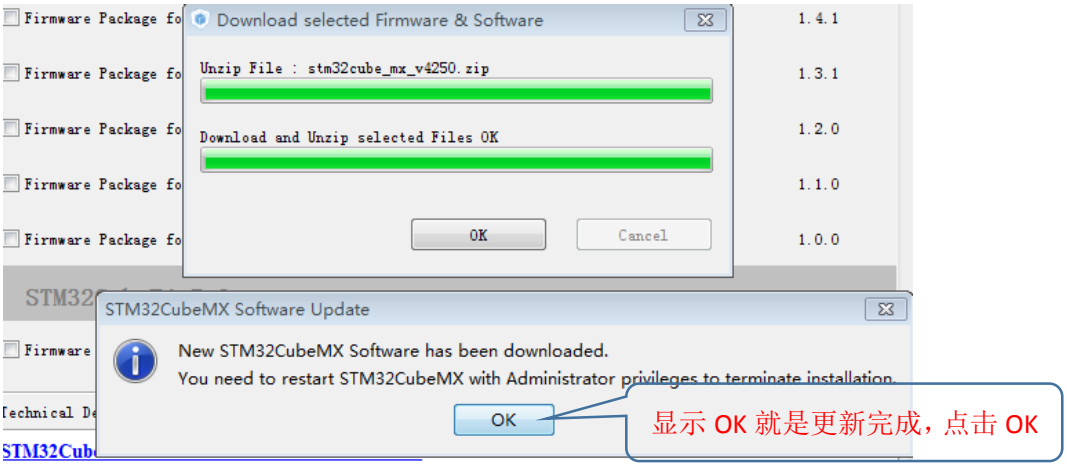
显示绿色就是表示可以更新

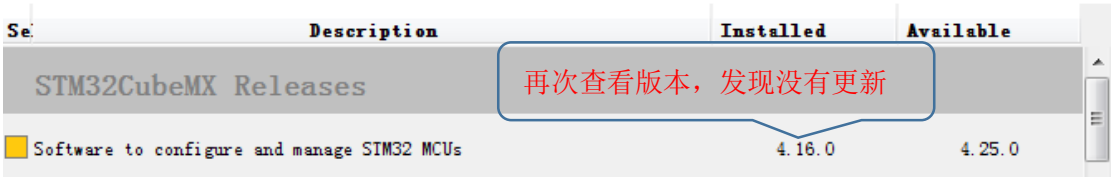


既然可以更新我们就再次选择这个

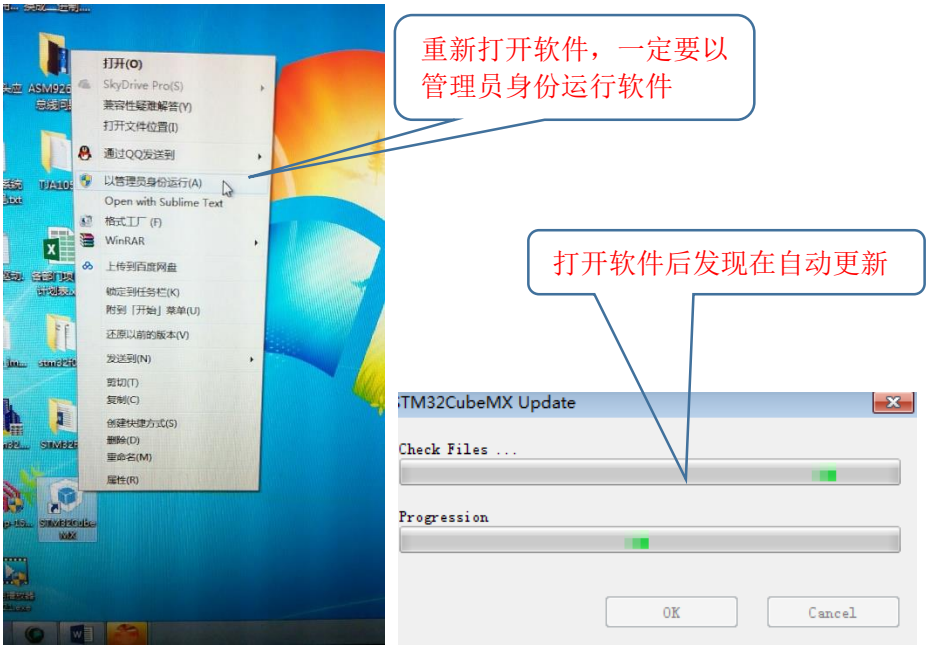
点击 Install Now 更新



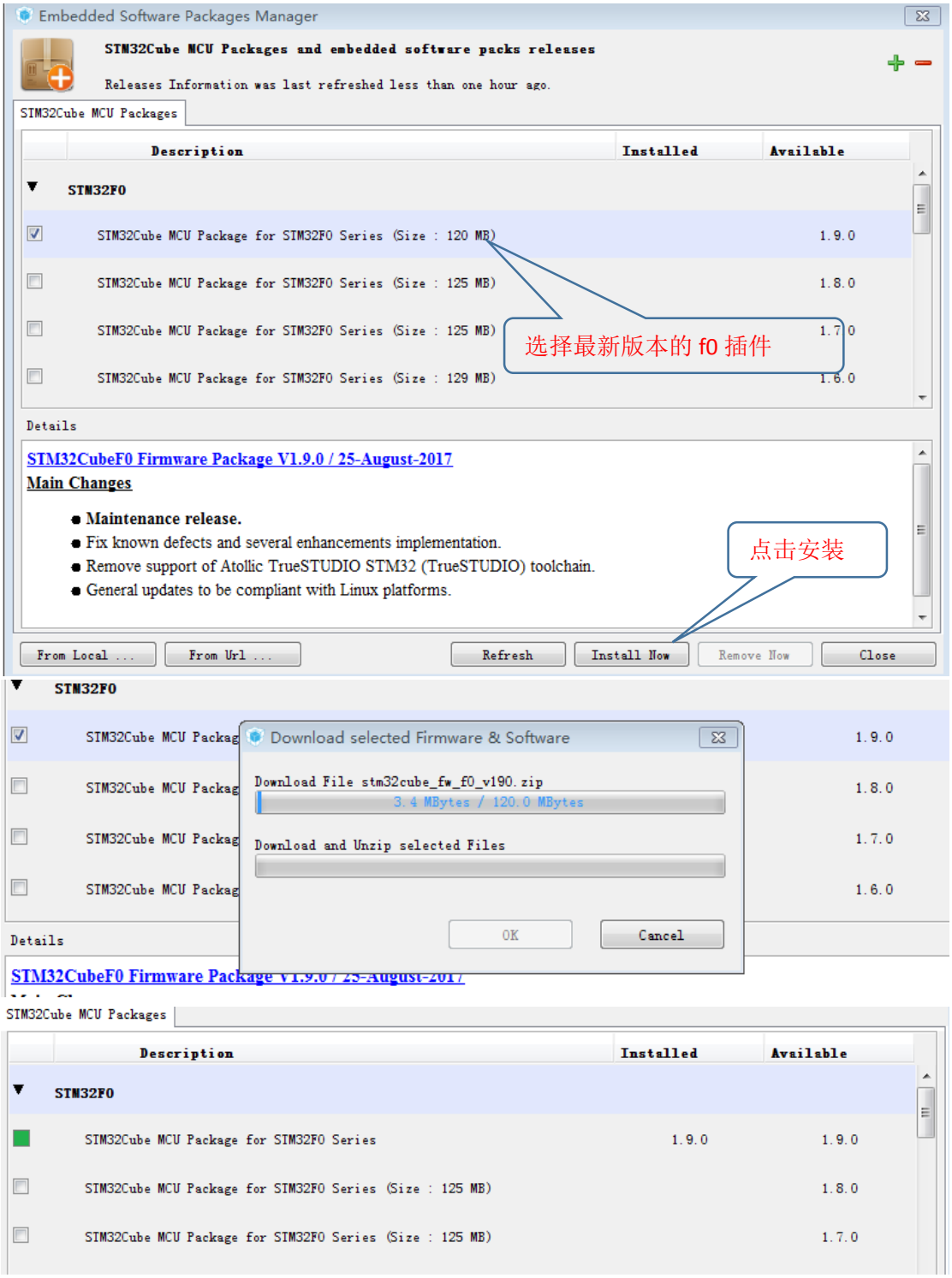




这个时候我们要关闭软件，其实更新脚本已经下载进来了只是我们要重启软件才能完全更新



然后安装插件，我们要选择和我们使用的芯片型号配对的插件版本我这里使用的是 STM32F0 所以我们最新的 f0 版本插件

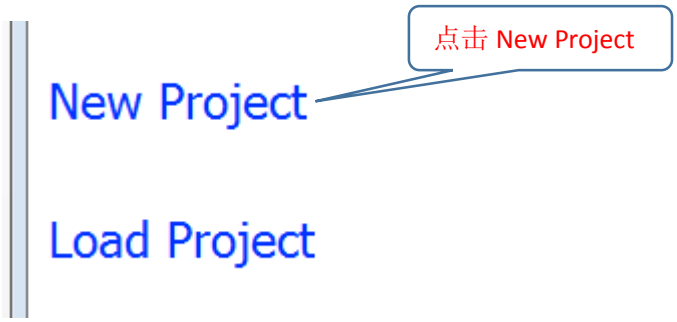


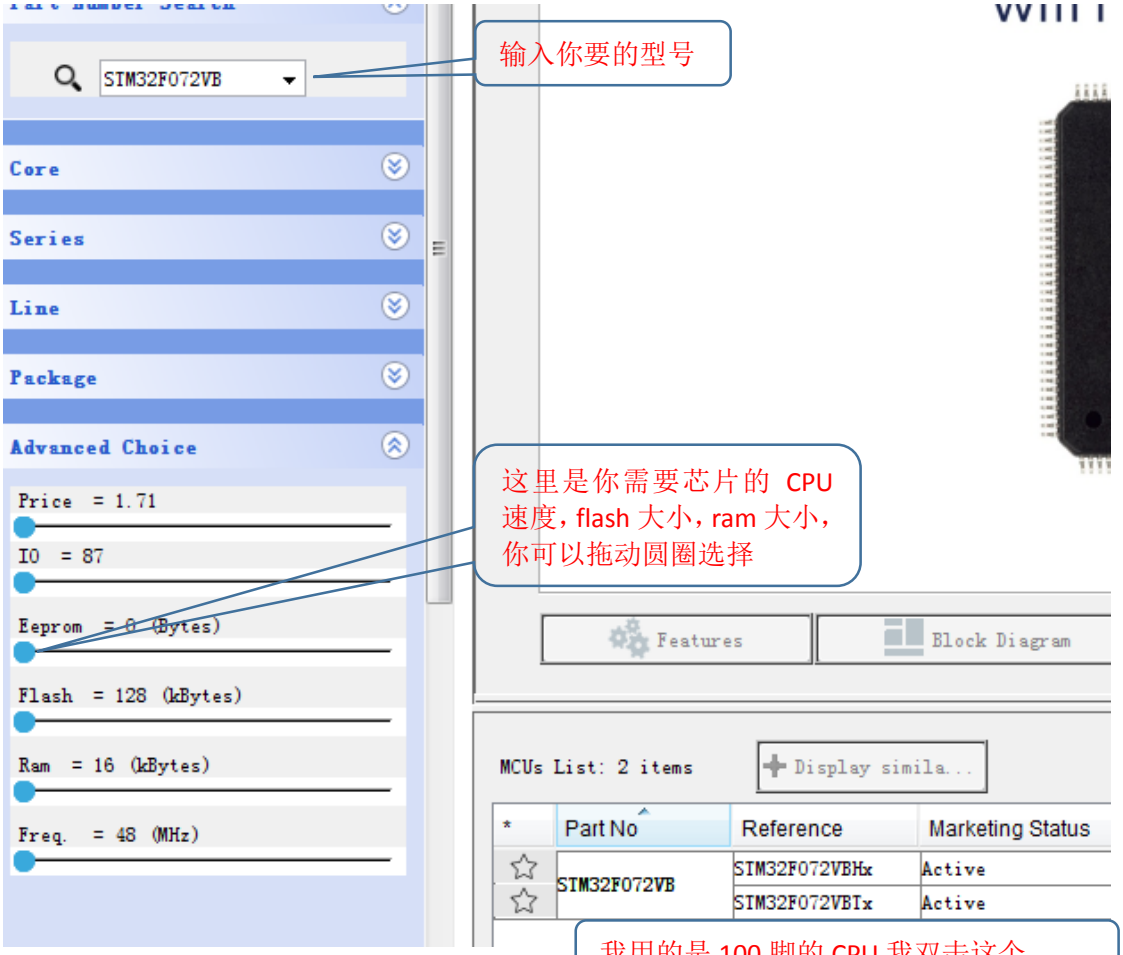
插件安装完成

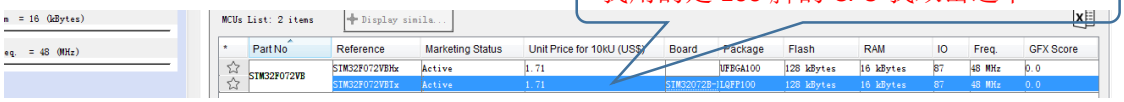
然后关闭整个软件，一定要关闭整个软件，然后再次打开软件就可以正常使用了。

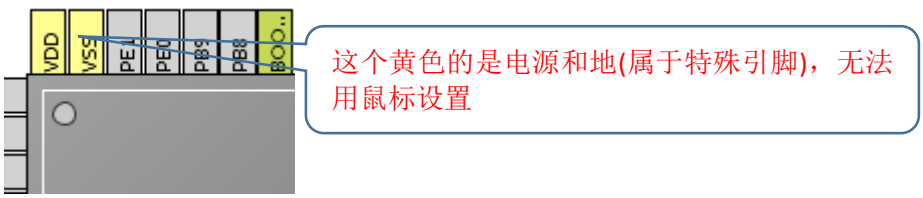
用 STM32CubeMX 生成代码

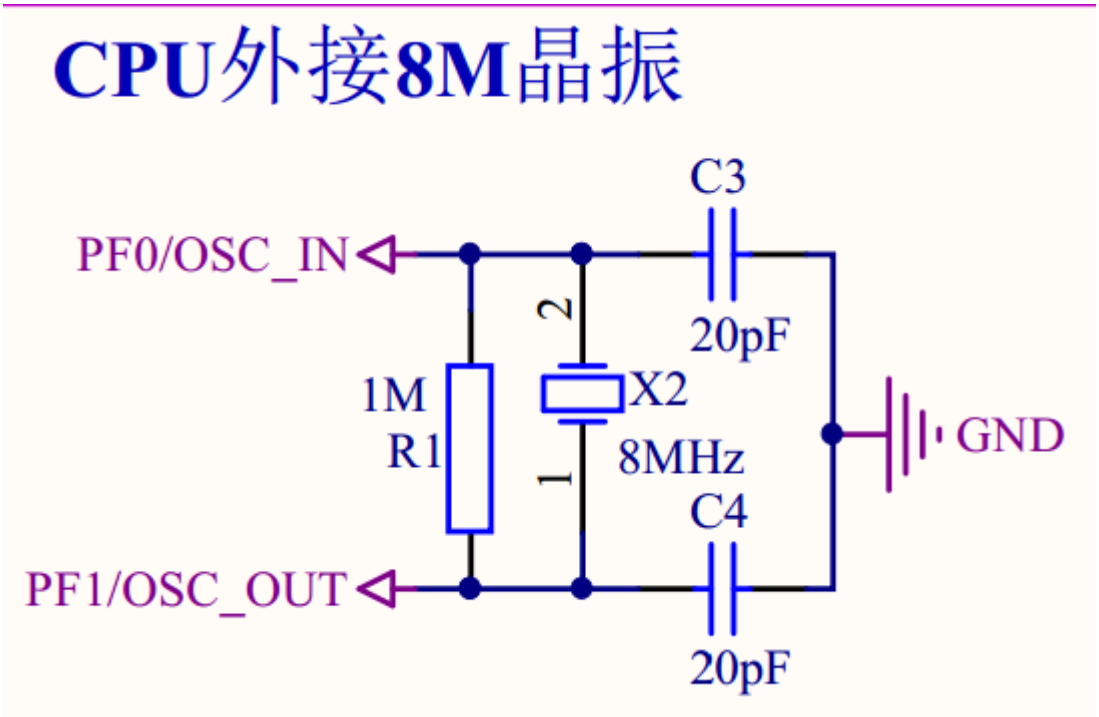
打开 CubeMX



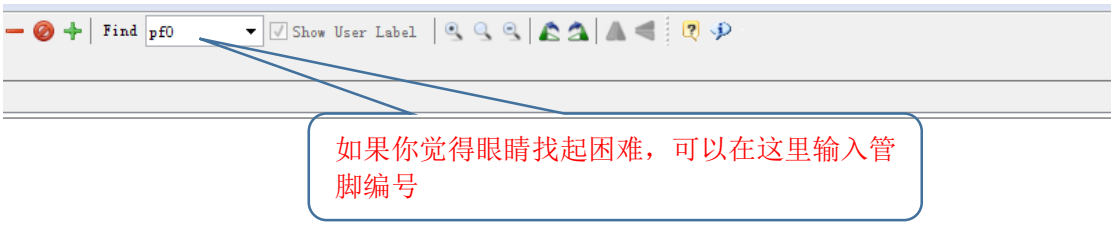


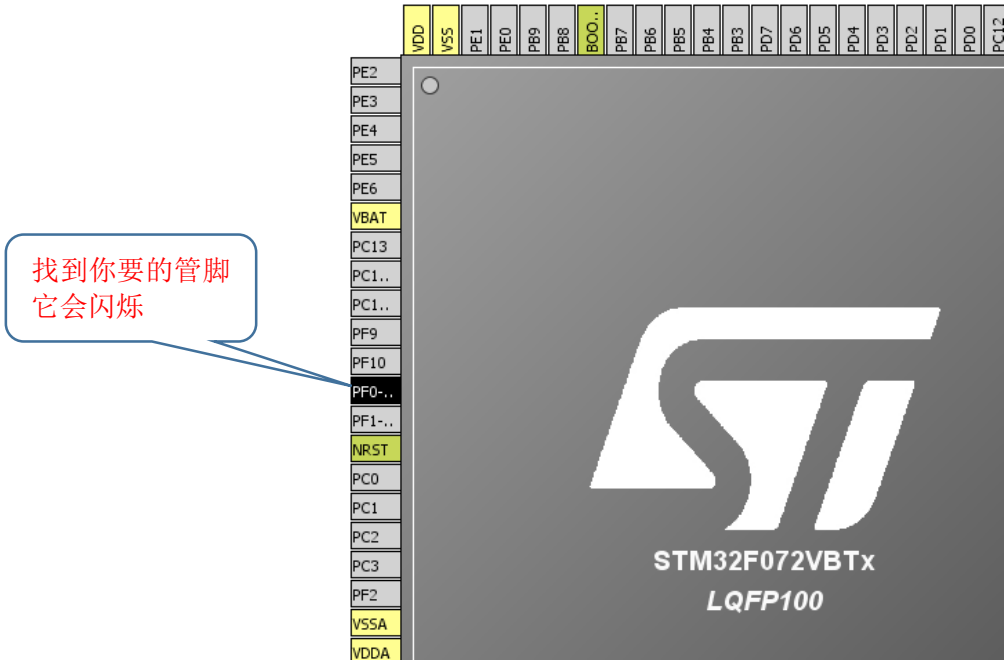


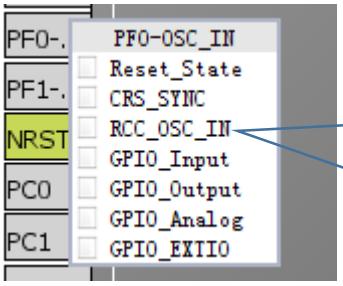




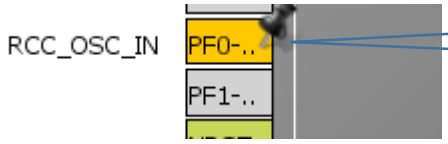
我们 MCU 运行需要时钟，所以要设置 PF0,PF1 管脚



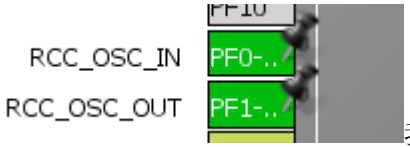




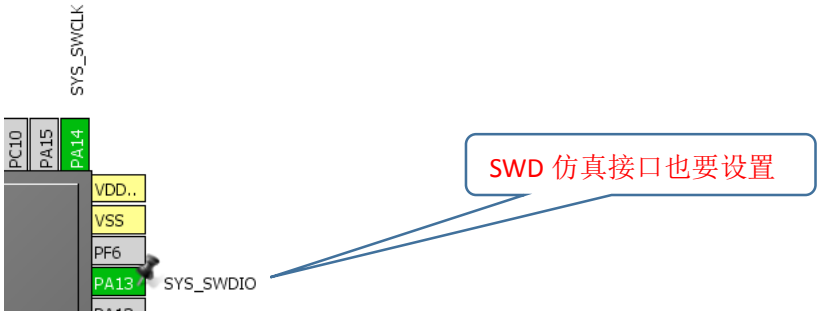
我们双击 PF0 引脚，会出现一个勾选列表，你需要管脚设置为什么功能，你勾选就是了，我这里设置管脚为时钟接口



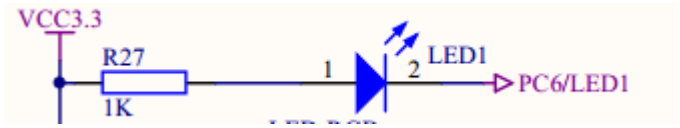
这就是设置成功后的样子

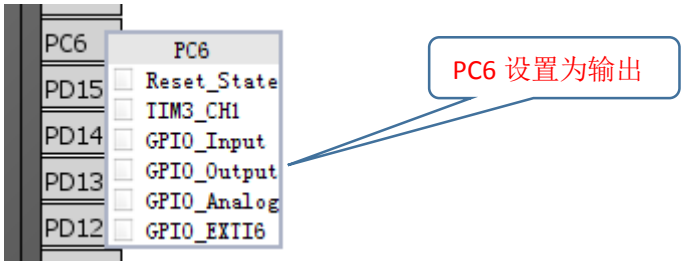


我们把 PF0 和 PF1 都设置上

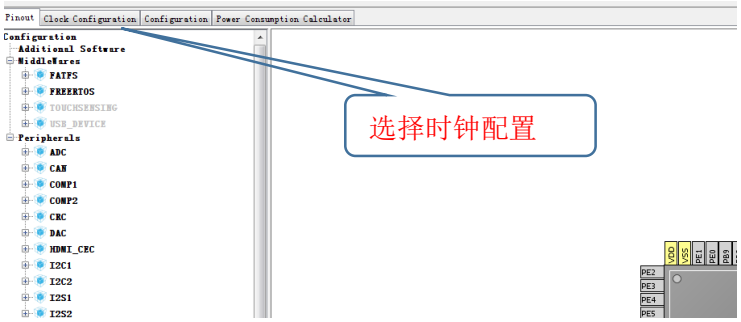


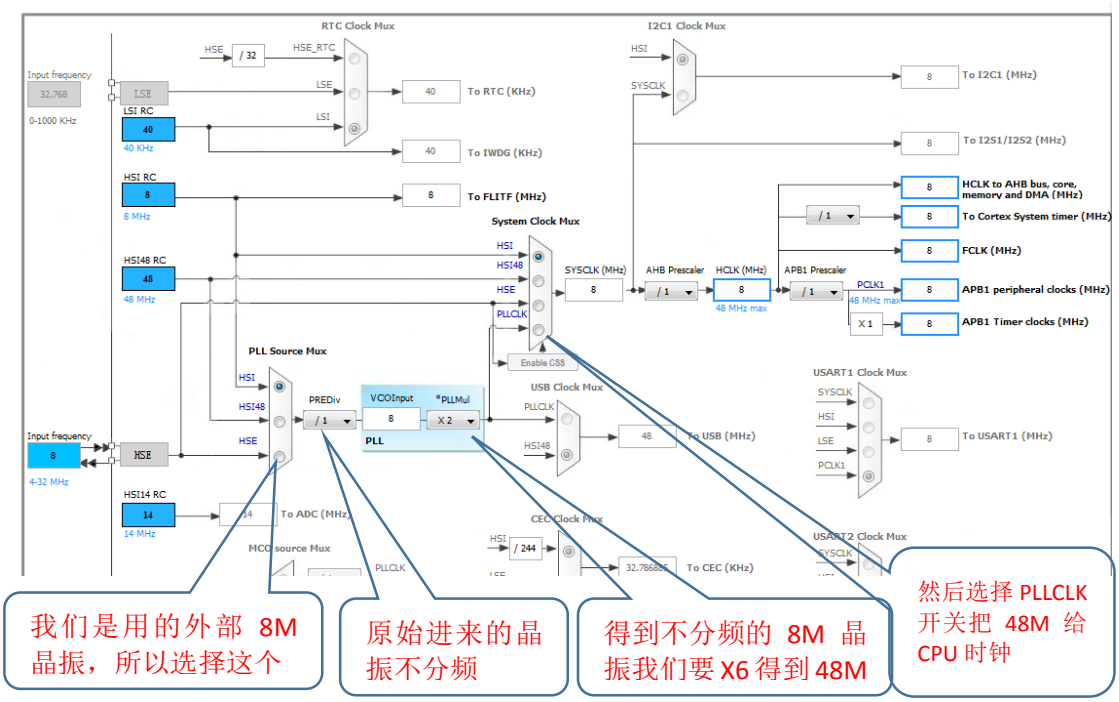
因为我们是控制 LED，所以我们还有选择 PC6 管脚

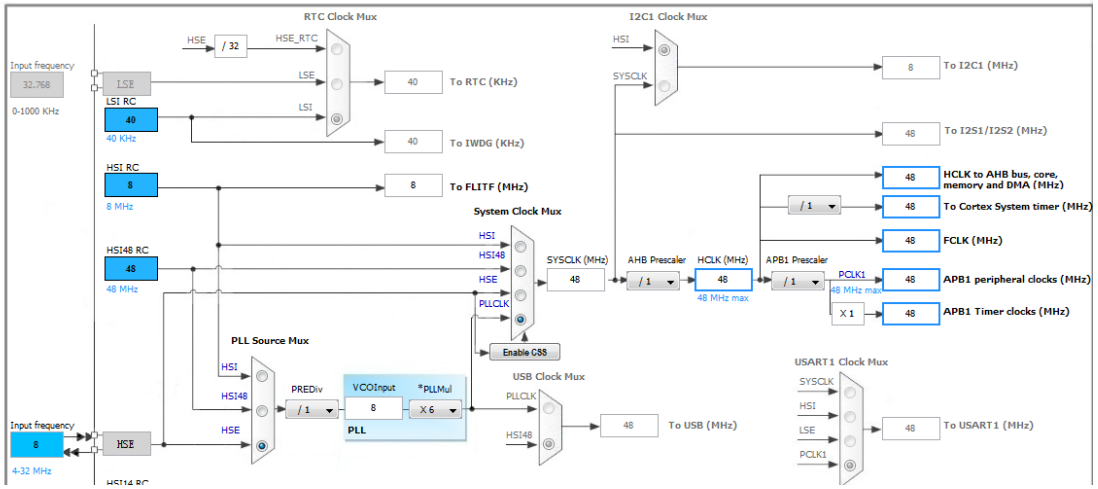




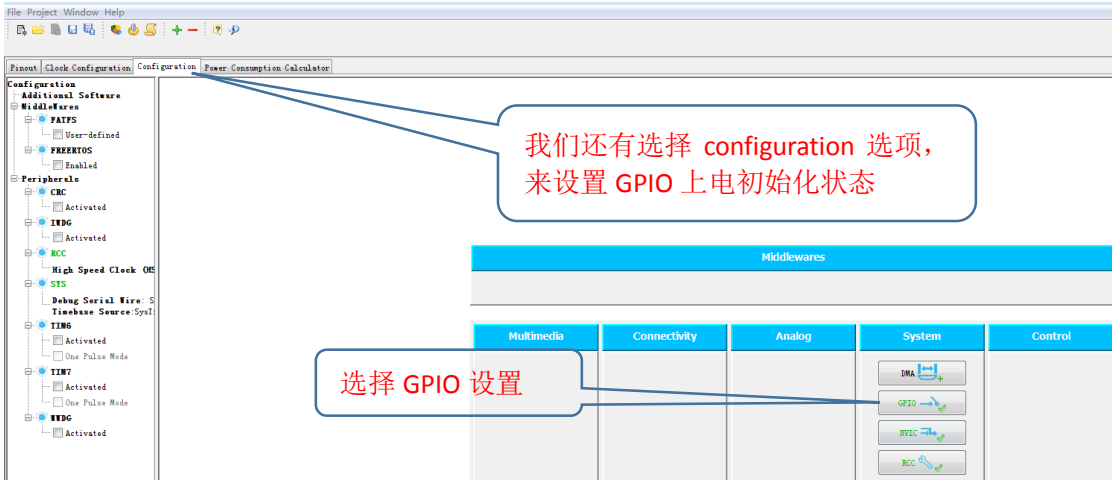
现在 MCU 要使用的引脚我们已经配置完成，下一步要设置时钟树

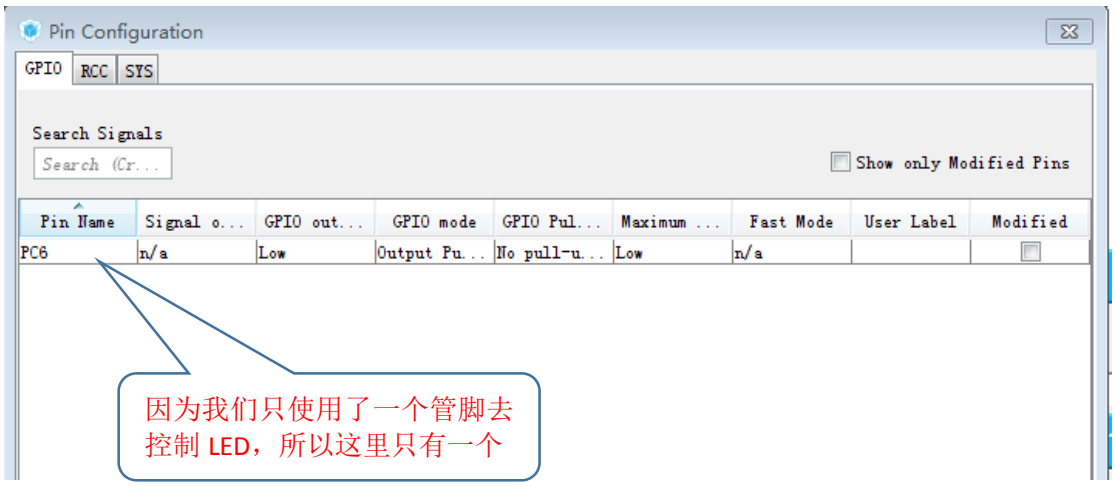


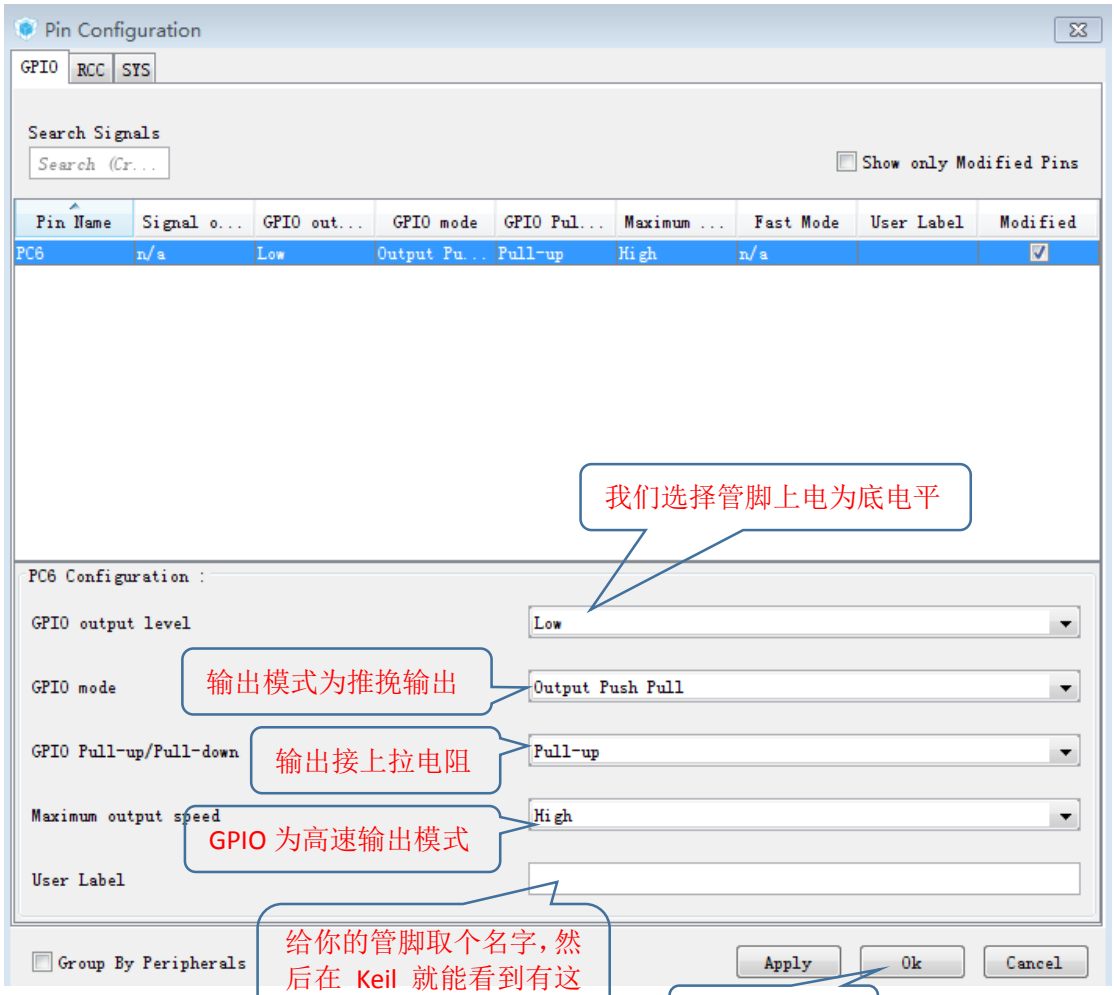


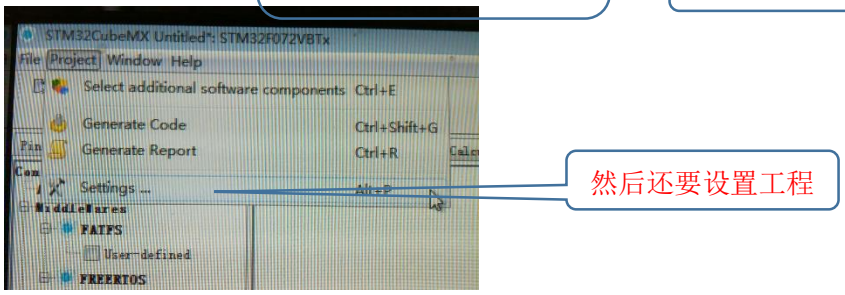


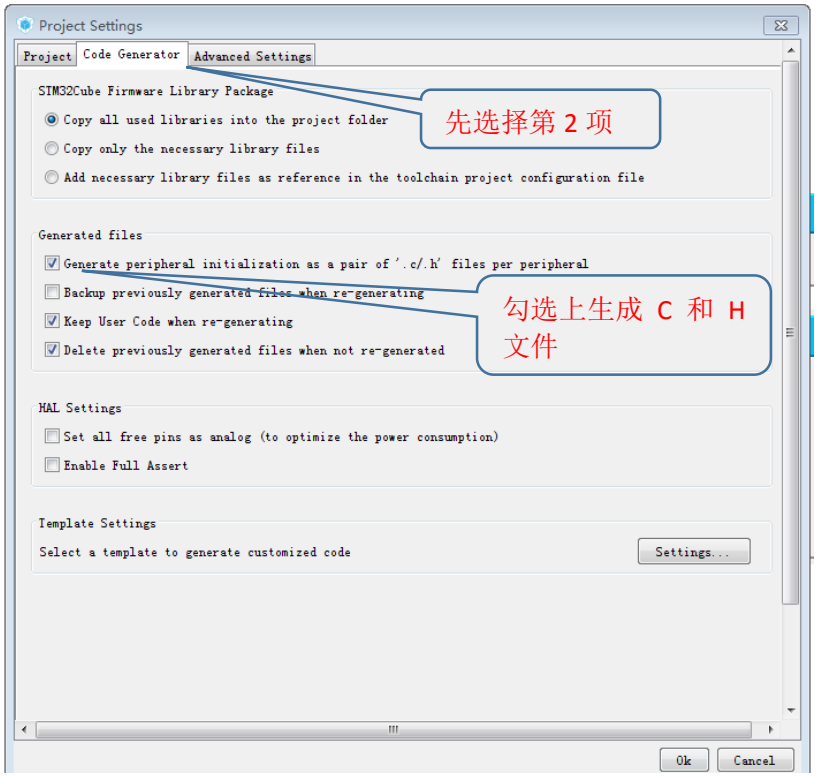
这就是设置后的结果。时钟配置完成

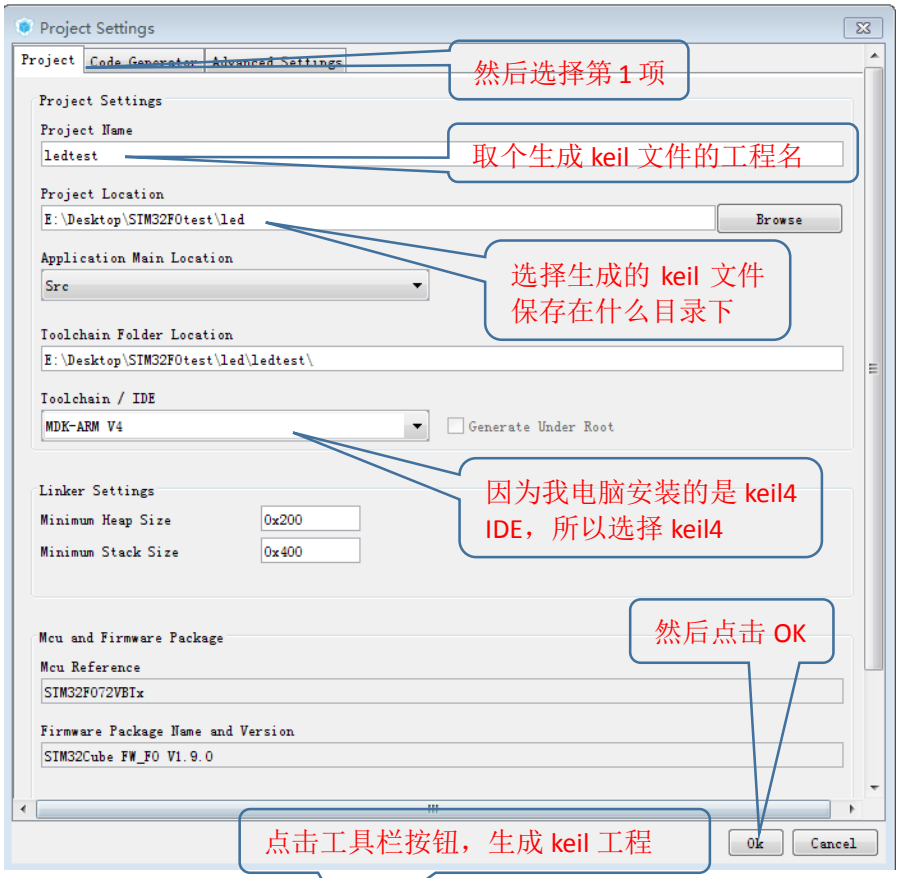



Pin Configuration
GPIO
RCC
SYS
Search Signals
Search (Cr...
Show only Modified Pins
Pin Name
Signal o...
GPIO out...
GPIO mode
GPIO Pul...
Maximum ...
Fast Mode
User Label
Modified
PC6
n/a
Low
Output Pu...
No pull-u...
Low
n/a
因为我们只使用了一个管脚去控制 LED，所以这里只有一个

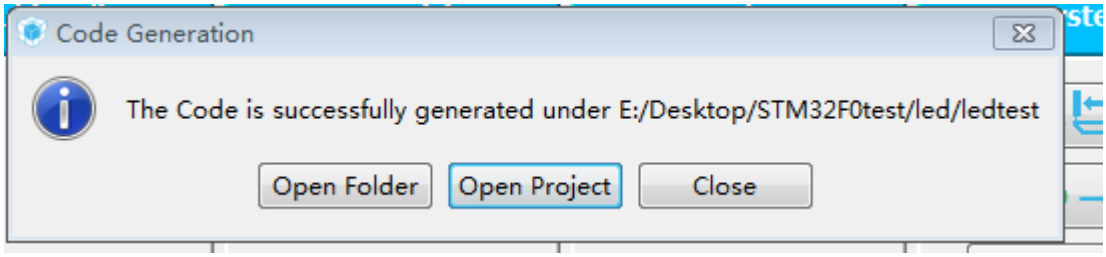

Pin Configuration
GPIO
RCC
SYS
Search Signals
Search (Cr...
Show only Modified Pins
Pin Name
Signal o...
GPIO out...
GPIO mode
GPIO Pul...
Maximum ...
Fast Mode
User Label
Modified
PC6
n/a
Low
Output Pu...
Pull-up
High
n/a
我们选择管脚上电为底电平
PC6 Configuration :
GPIO output level
Low
GPIO mode
输出模式为推挽输出
Output Push Pull
GPIO Pull-up/Pull-down
输出接上拉电阻
Pull-up
Maximum output speed
High
GPIO 为高速输出模式
User Label
Group By Peripherals
给你的管脚取个名字，然后在 Keil 就能看到有这个名字的代码
Apply
Ok
Cancel
最后点击 OK


STM32CubeMX Untitled*: STM32F072VBTx
File
Project
Window
Help
Select additional software components Ctrl+E
Generate Code Ctrl+Shift+G
Generate Report Ctrl+R
Settings ...
FATFS
User-defined
FREERTOS
然后还要设置工程

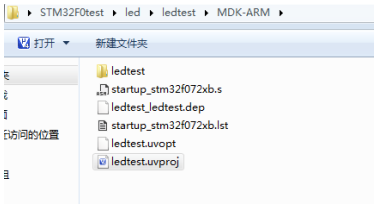

Project Settings
Project
Code Generator
Advanced Settings
STM32Cube Firmware Library Package
Copy all used libraries into the project folder
Copy only the necessary library files
Add necessary library files as reference in the toolchain project configuration file
先选择第 2 项
Generated files
Generate peripheral initialization as a pair of '.c/.h' files per peripheral
Backup previously generated files when re-generating
Keep User Code when re-generating
Delete previously generated files when not re-generated
勾选上生成 C 和 H 文件
HAL Settings
Set all free pins as analog (to optimize the power consumption)
Enable Full Assert
Template Settings
Select a template to generate customized code
Settings...
Ok
Cancel


Project Settings
Project
然后选择第 1 项
Project Settings
Project Name
ledtest
取个生成 keil 文件的工程名
Project Location
E:\Desktop\STM32F0test\led
Browse
Application Main Location
Src
选择生成的 keil 文件保存在什么目录下
Toolchain Folder Location
E:\Desktop\STM32F0test\led\ledtest\
Toolchain / IDE
MDK-ARM V4
Generate Under Root
因为我电脑安装的是 keil4 IDE，所以选择 keil4
Linker Settings
Minimum Heap Size
0x200
Minimum Stack Size
0x400
然后点击 OK
Mcu and Firmware Package
Mcu Reference
STM32F072VBIx
Firmware Package Name and Version
STM32Cube FW_F0 V1.9.0
Ok
Cancel
点击工具栏按钮，生成 keil 工程

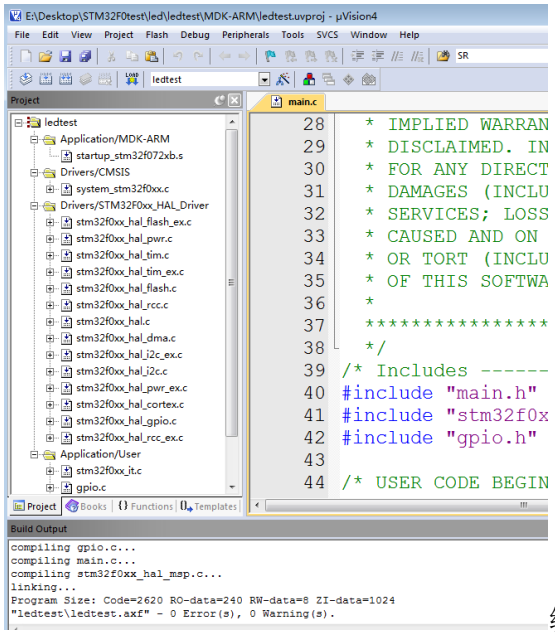

File
Project
Window
Help



工程生成完成，然后你点击 close，再关闭 CubeMX 软件，用 keil 打开工程



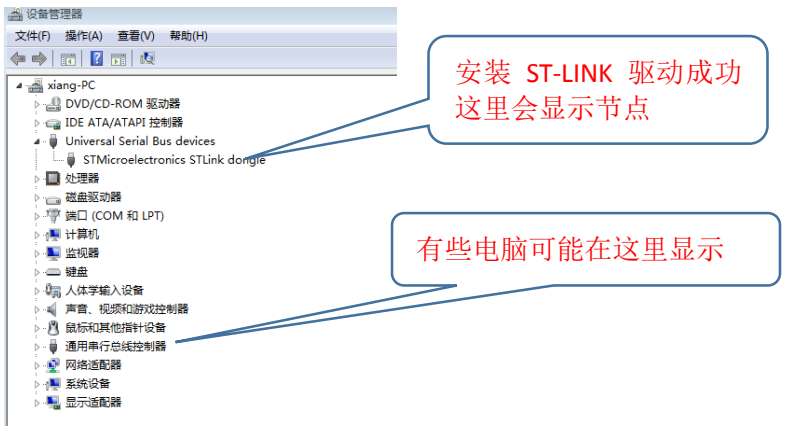
工程文件就在你 CubeMX 指定生成 keil 工程的目录下

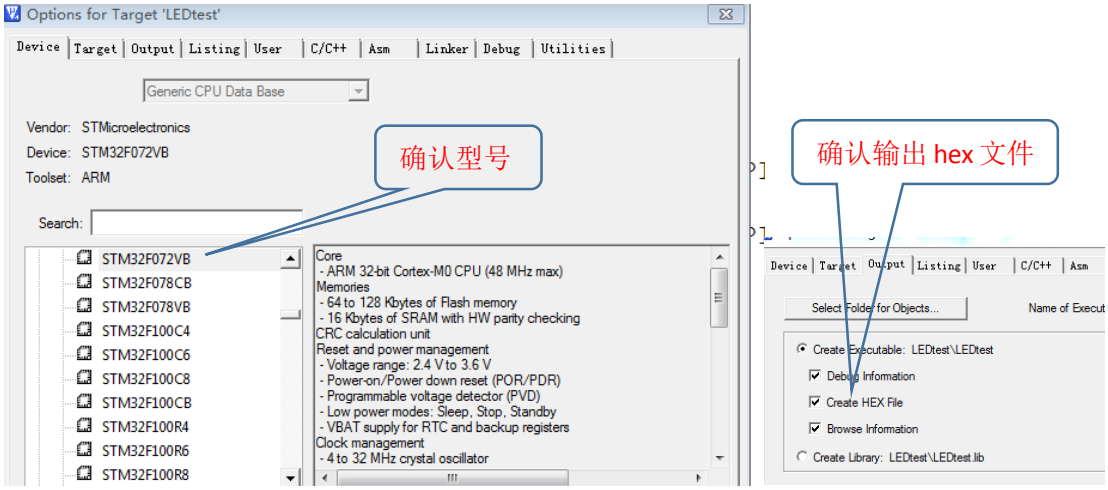


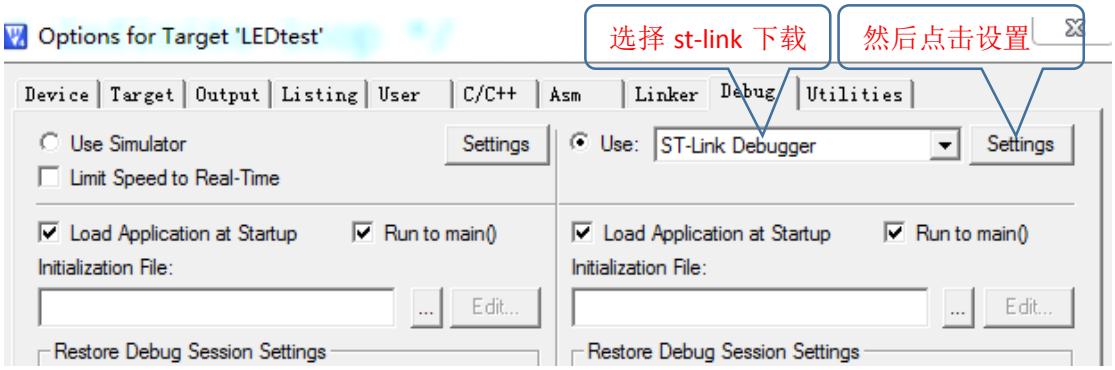
编译通过，证明工程建立没有问题，后面按照常规的操作 keil 方法操作就是了。

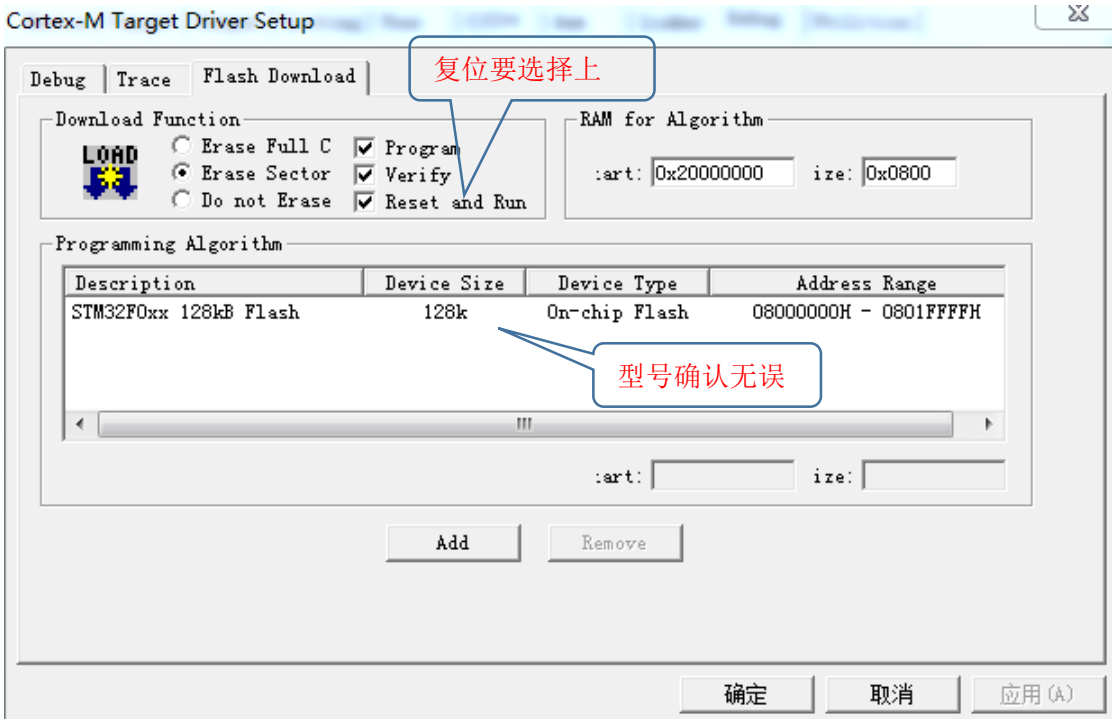
设置 keil 的下载功能

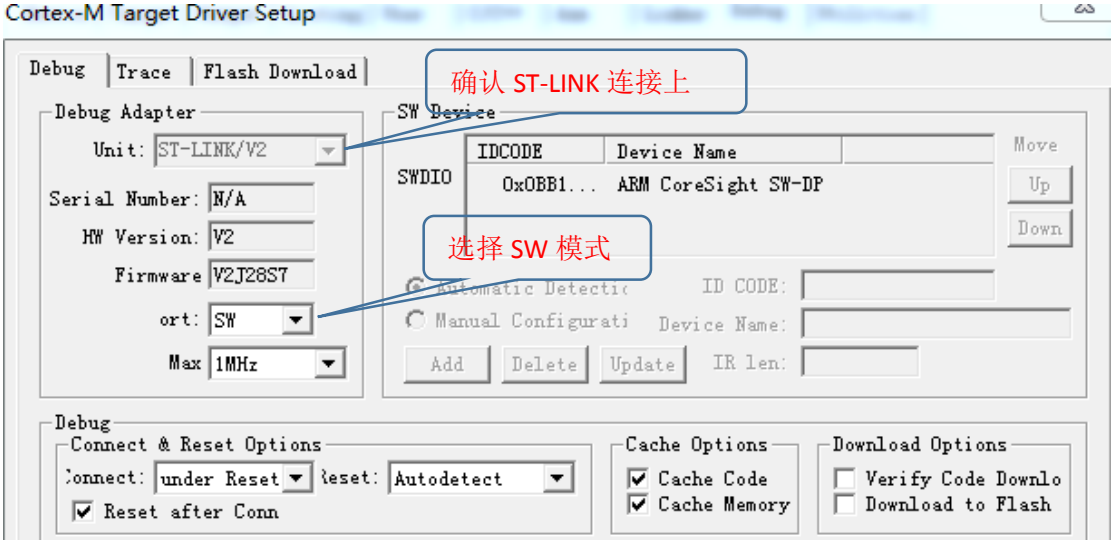
我们这里用的是 st-link v2 下载器

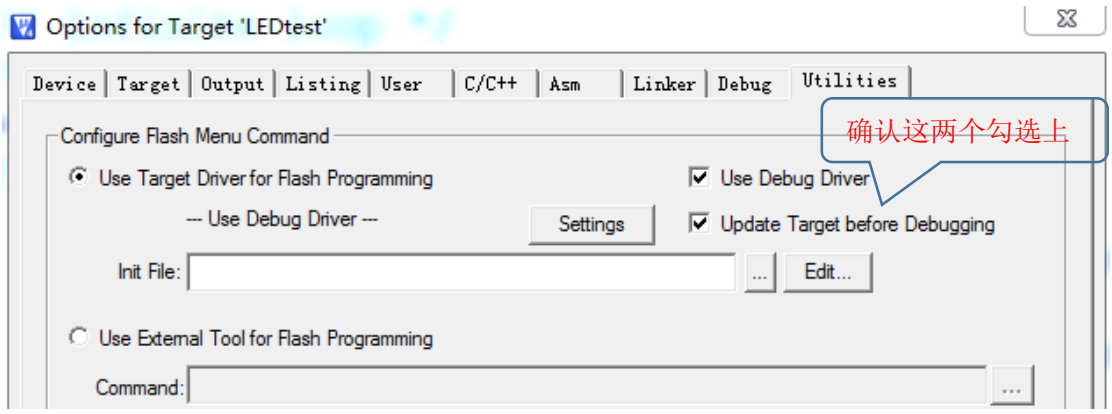


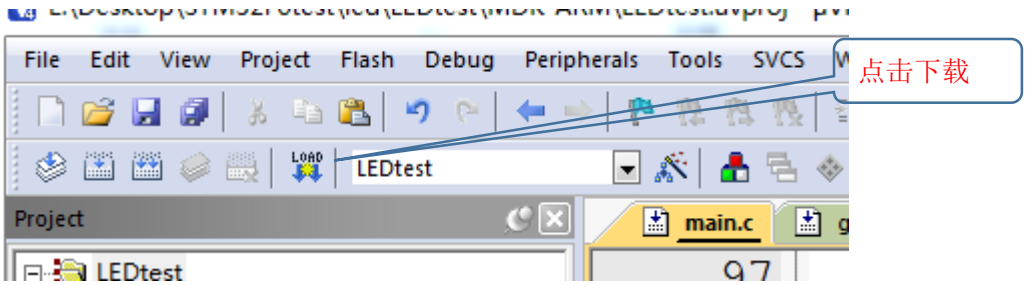

Options for Target 'LEDtest'
Device
Target
Output
Listing
User
C/C++
Asm
Linker
Debug
Utilities
Generic CPU Data Base
Vendor: STMicroelectronics
Device: STM32F072VB
Toolset: ARM
Search:
STM32F072VB
STM32F078CB
STM32F078VB
STM32F100C4
STM32F100C6
STM32F100C8
STM32F100CB
STM32F100R4
STM32F100R6
STM32F100R8
Core
- ARM 32-bit Cortex-M0 CPU (48 MHz max)
Memories
- 64 to 128 Kbytes of Flash memory
- 16 Kbytes of SRAM with HW parity checking
CRC calculation unit
Reset and power management
- Voltage range: 2.4 V to 3.6 V
- Power-on/Power down reset (POR/PDR)
- Programmable voltage detector (PVD)
- Low power modes: Sleep, Stop, Standby
- VBAT supply for RTC and backup registers
Clock management
- 4 to 32 MHz crystal oscillator
确认型号
确认输出 hex 文件
Select Folder for Objects...
Name of Execut
Create Executable: LEDtest\LEDtest
Debug Information
Create HEX File
Browse Information
Create Library: LEDtest\LEDtest.lib

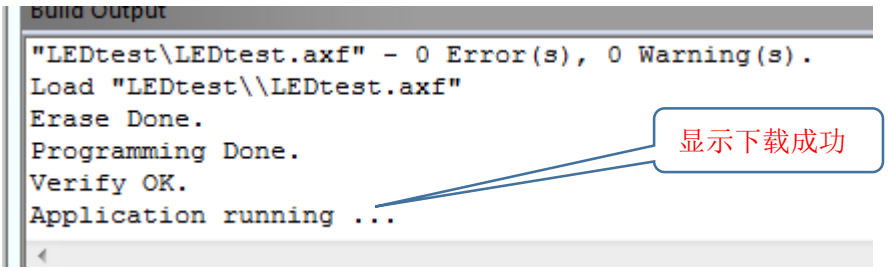

Options for Target 'LEDtest'
选择 st-link 下载
然后点击设置
Device
Target
Output
Listing
User
C/C++
Asm
Linker
Debug
Utilities
Use Simulator
Settings
Limit Speed to Real-Time
Use:
ST-Link Debugger
Settings
Load Application at Startup
Run to main()
Initialization File:
Edit...
Restore Debug Session Settings
Load Application at Startup
Run to main()
Initialization File:
Edit...
Restore Debug Session Settings


Cortex-M Target Driver Setup
Debug
Trace
Flash Download
复位要选择上
Download Function
LOAD
Erase Full C
Erase Sector
Do not Erase
Program
Verify
Reset and Run
RAM for Algorithm
:art: 0x20000000
ize: 0x0800
Programming Algorithm
Description
Device Size
Device Type
Address Range
STM32F0xx 128kB Flash
128k
On-chip Flash
08000000H - 0801FFFFH
型号确认无误
:art:
ize:
Add
Remove
确定
取消
应用(A)

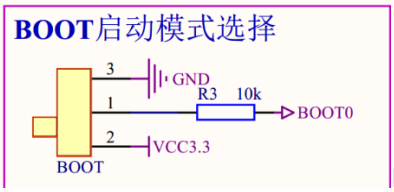






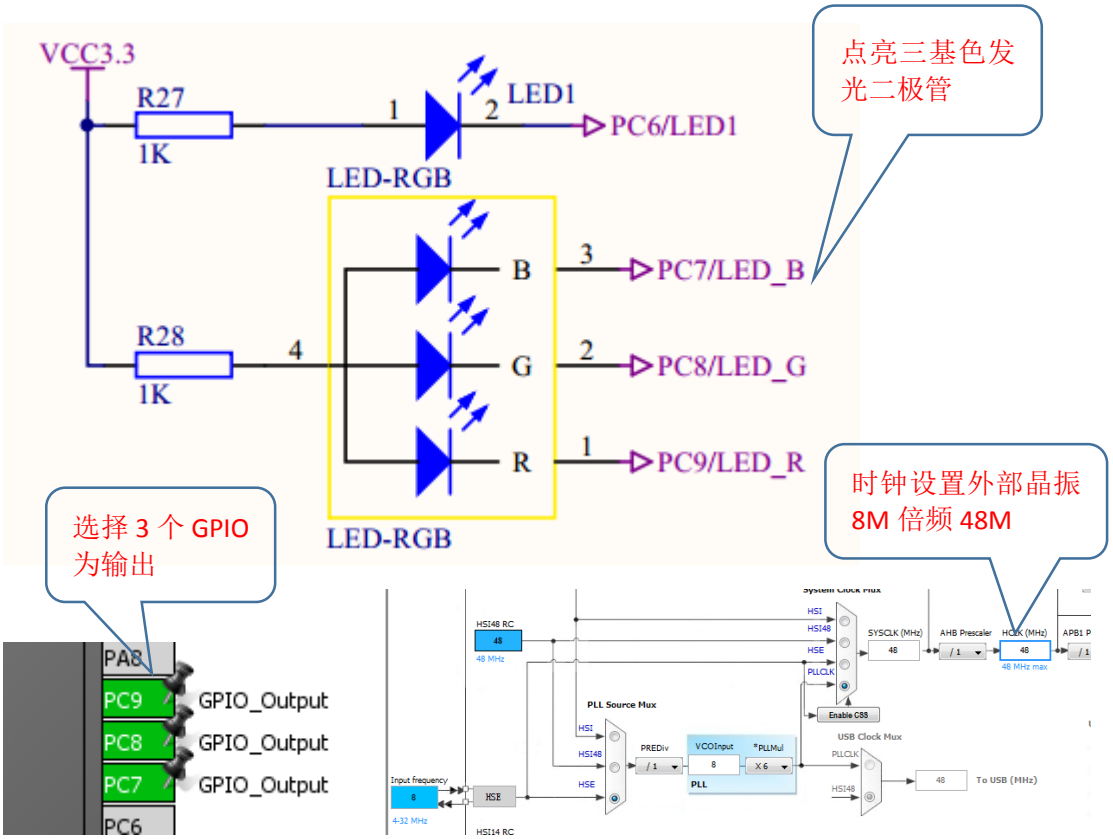


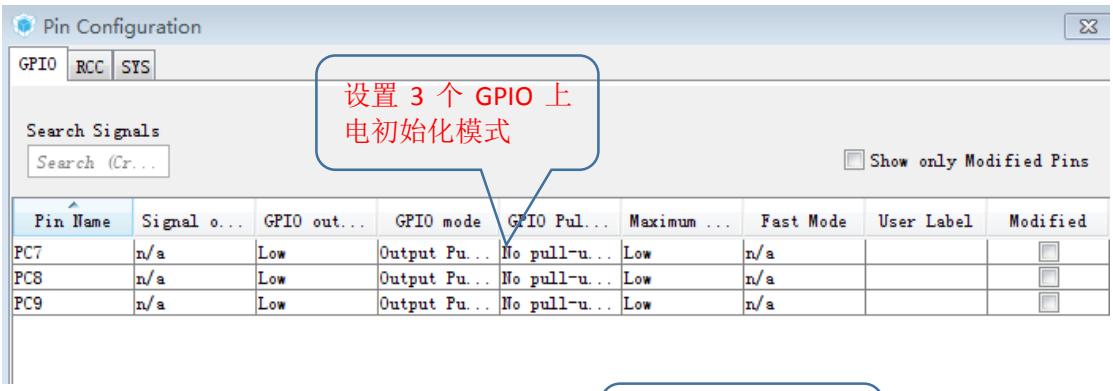
如果程序正常下载后 STM32F072 没有运行起来，那么确认下 boot 引脚的启动模式

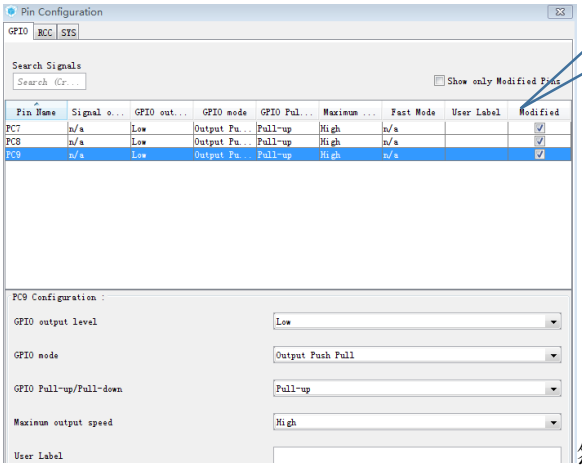


boot0 要接地，才能从 flash 启动。所以这里要注意

GPIO 管脚输出操作







然后按照上面 CubeMX 的方法生成 keil 工程

```c
void MX_GPIO_Init(void)
{

  GPIO_InitTypeDef GPIO_InitStruct;

  /* GPIO Ports Clock Enable */
  __HAL_RCC_GPIOF_CLK_ENABLE();
  __HAL_RCC_GPIOC_CLK_ENABLE();
  __HAL_RCC_GPIOA_CLK_ENABLE();

  /*Configure GPIO pin Output Level */
  HAL_GPIO_WritePin(GPIOC, GPIO_PIN_7|GPIO_PIN_8|GPIO_PIN_9, GPIO_PIN_RESET);

  /*Configure GPIO pins : PC7 PC8 PC9 */
  GPIO_InitStruct.Pin = GPIO_PIN_7|GPIO_PIN_8|GPIO_PIN_9;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_PULLUP;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_HIGH;
  HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);

}
```

各组 GPIO 管脚时钟初始化函数

向某个 GPIO 管脚写 0 或者写 1 函数

设置某个 GPIO 管脚模式

GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP; //GPIO 推挽输出模式

GPIO_InitStruct.Pull = GPIO_PULLUP; //GPIO 带上拉电阻

GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_HIGH;//GPIO 电平输出速度为高速

HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);//将设置的 GPIO 参数初始化

```c
#include "main.h"
#include "stm32f0xx_hal.h"
#include "gpio.h"


void SystemClock_Config(void);

int main(void)
{

  HAL_Init();
  SystemClock_Config();

  MX_GPIO_Init();
  HAL_GPIO_WritePin(GPIOC, GPIO_PIN_7|GPIO_PIN_8|GPIO_PIN_9, GPIO_PIN_SET);
  while (1)
  {
    HAL_GPIO_WritePin(GPIOC, GPIO_PIN_7|GPIO_PIN_8, GPIO_PIN_RESET);
    HAL_Delay(500);
    HAL_GPIO_WritePin(GPIOC, GPIO_PIN_7|GPIO_PIN_8, GPIO_PIN_SET);
    HAL_Delay(500);


  }
```

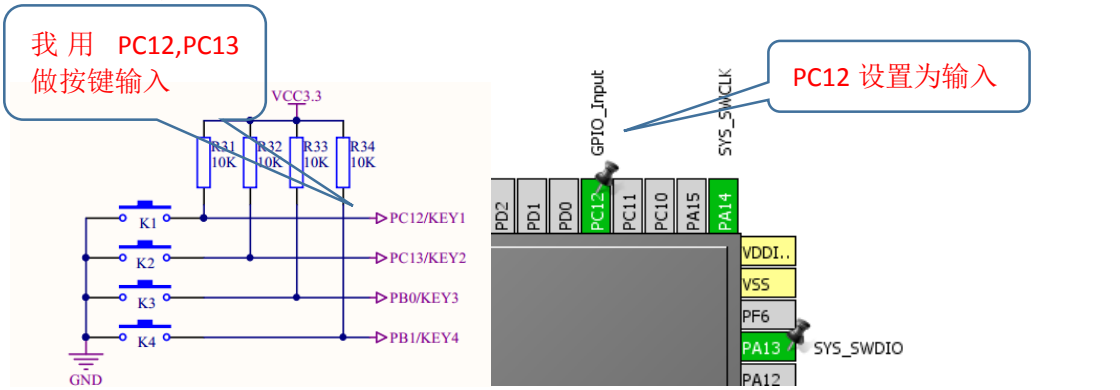
设置完成 LED 每延迟 500ms 闪烁

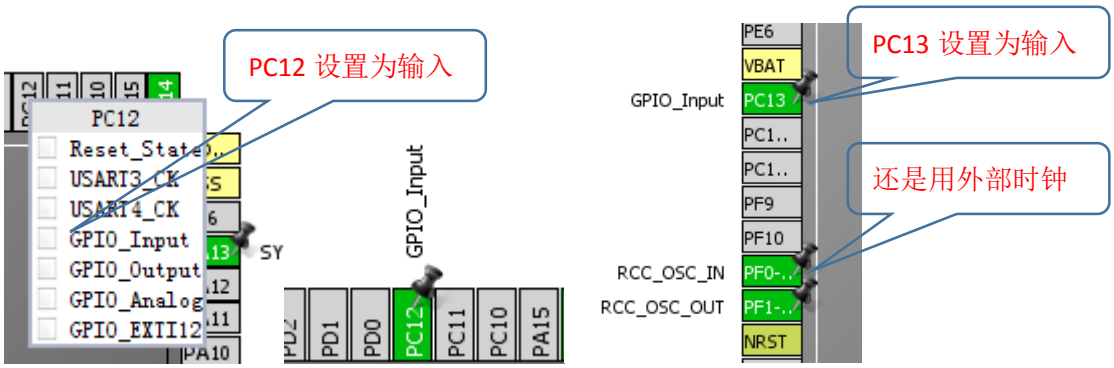
HAL_GPIO_WritePin (GPIO 组, GPIO 单个引脚, 电平状态);

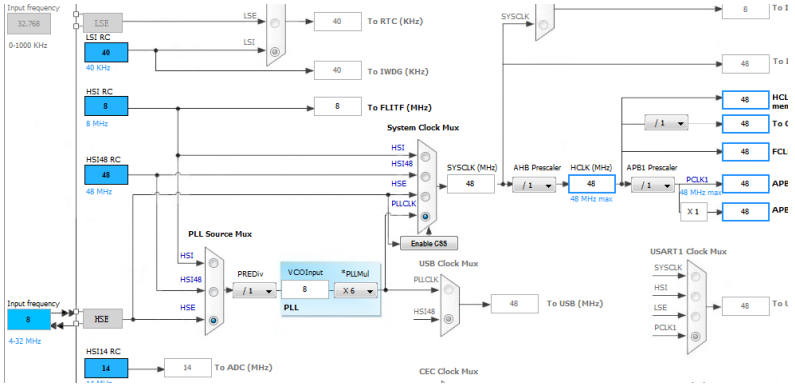
GPIO_PIN_RESET = 0

GPIO_PIN_SET = 1

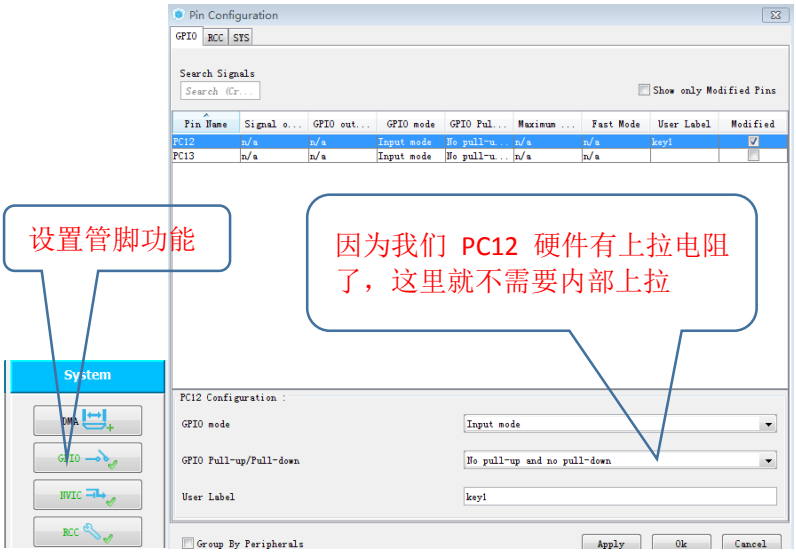
GPIO 管脚输入操作







时钟设置还是老样子，外部晶振 8M，倍频 48M



PC13 设置方式一样的

然后生成 keil 工程

```
void MX_GPIO_Init(void)
{

  GPIO_InitTypeDef GPIO_InitStruct;

  /* GPIO Ports Clock Enable */
  __HAL_RCC_GPIOC_CLK_ENABLE();
  __HAL_RCC_GPIOF_CLK_ENABLE();
  __HAL_RCC_GPIOA_CLK_ENABLE();

  /*Configure GPIO pins : PCPin PCPin */
  GPIO_InitStruct.Pin = key2_Pin|key1_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
    /*Configure GPIO pins : PC7 OUT */
  GPIO_InitStruct.Pin = GPIO_PIN_7;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_PULLUP;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_HIGH;
  HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);

}
```

打开 GPIO 时钟

设置 GPIO 管脚输入模式

设置另外 GPIO 管脚输出模式

```
int main(void)
{

  /* MCU Configuration--------------------------------------------
  /* Reset of all peripherals, Initializes the Flash interface a
  HAL_Init();

  /* Configure the system clock */
  SystemClock_Config();
  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  while (1)
  {
    if(HAL_GPIO_ReadPin(GPIOC,GPIO_PIN_13)==GPIO_PIN_RESET)
    {
        HAL_Delay(20);
        if(HAL_GPIO_ReadPin(GPIOC,GPIO_PIN_13)==GPIO_PIN_RESET)
        {
            HAL_GPIO_WritePin(GPIOC,GPIO_PIN_7,GPIO_PIN_RESET);
        }
    }
    else
    {
        HAL_GPIO_WritePin(GPIOC,GPIO_PIN_7,GPIO_PIN_SET);
    }
  }
```

GPIO_PinState HAL_GPIO_ReadPin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin) 读单个 GPIO 电平

HAL_GPIO_ReadPin(GPIOC,GPIO_PIN_13)返回 GPIO_PIN_RESET 就是 0 返回 GPIO_PIN_SET 是 1

✓ 可以选择 4 种中断优先级（0~3）；

软件配置中断优先级可以选择 4 种，设置 0 优先级最高，3 优先级最低

如果两个 IO 管脚软件设置的中断优先级一样，比如都设置为优先级 1，那么先触发哪个 IO 口的中断函数呢？，这就要根据中断向量表来决定了。

表 15.1　中断向量表

位置	优先级	优先级类型	名称	说明
	-	-	-	保留(Reserved)
	-3	固定	Reset	复位(Reset)
	-2	固定	NMI	不可屏蔽中断。RCC 时钟安全系统(CSS)连接到 NMI 向量
	-1	固定	HardFault	所有类型的错误(fault)
	3	可设置	SVCall	通用 SWI 指令调用的系统服务
	5	可设置	PendSV	可挂起的系统服务
	6	可设置	SysTick	系统嘀嗒定时器
0		可设置	WWDG	窗口看门狗中断
1		可设置	PVD	连接到 EXTI 线的可编程电压检测(PVD)中断
2		可设置	RTC	RTC 全局中断
3		可设置	FLASH	FLASH 全局中断
4		可设置	RCC	RCC 全局中断
5		可设置	EXTI0_1	EXTI 线[1:0]中断
6		可设置	EXTI2_3	EXTI 线[3:2]中断
7		可设置	EXTI4_15	EXTI 线 15 和 EXTI 线 4 中断
8		可设置	TSC	触摸传感中断
9		可设置	DMA_CH1	DMA 通道 1 中断
10		可设置	DMA_CH2_3	DMA 通道 2 和 3 中断
11		可设置	DMA_CH4_5	DMA 通道 4 和 5 中断
12		可设置	ADC_COMP	ADC 和比较器 1 和 2 中断
13		可设置	TIM1_BRK_UP_TRG_COM	TIM1 刹车、更新、触发和通信中断

EXTI0_1 中断线软件设置优先级为 1

EXTI2_3 中断线软件设置优先级也为 1

根据中断向量表，程序会先执行最小的那个向量中断函数，这里是 5，EXTI0_1

我现在来设置 IO 管脚的输入中断

STM32F0 这款芯片 GPIO 分为 A,B,C,D,E,F 组，每一组有 0～15 个引脚，但是 F 组引脚不足 15

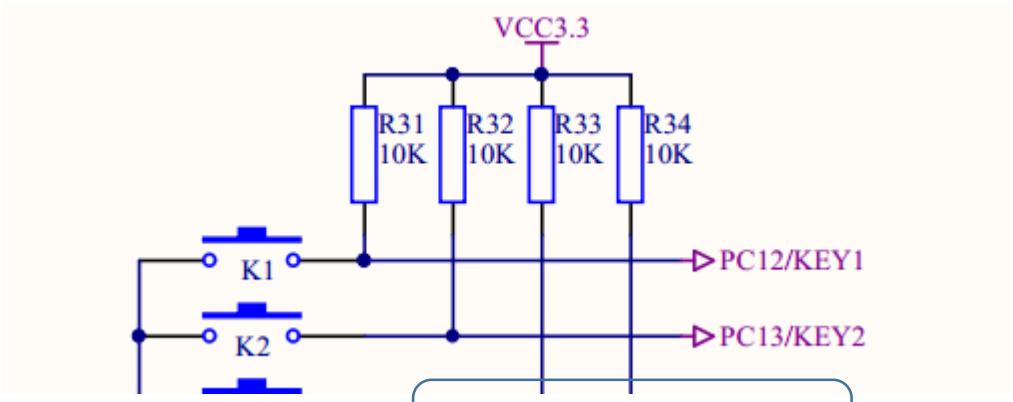
5		可设置	EXTI0_1	EXTI 线[1:0]中断
6		可设置	EXTI2_3	EXTI 线[3:2]中断
7		可设置	EXTI4_15	EXTI 线 15 和 EXTI 线 4 中断

EXTI0_1 对应 GPIOA0,GPIOA1，或者是 GPIOB0,GPIOB1 其余的 GPIO 组也是一样

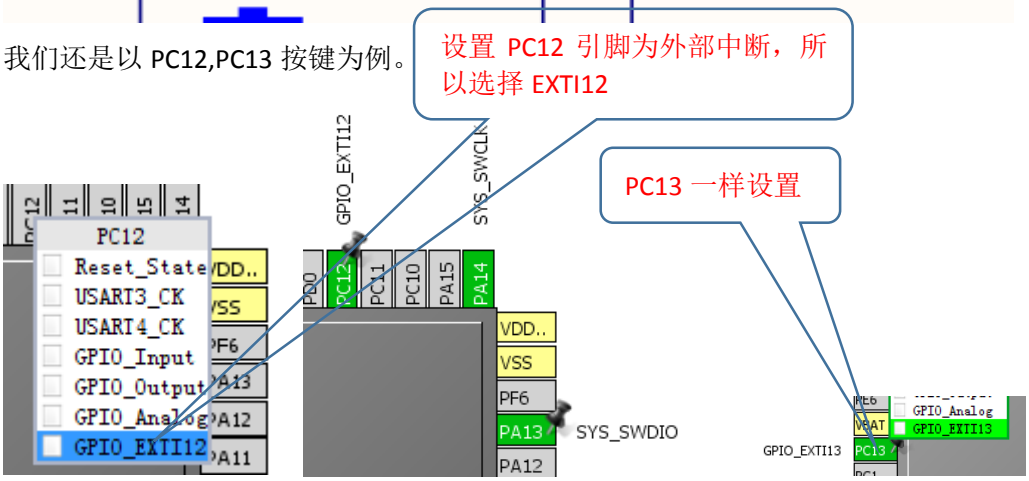
EXTI2_3 对应 GPIOA2,GPIOA3，或者是 GPIOB2,GPIOB3 其余的 GPIO 组也是一样

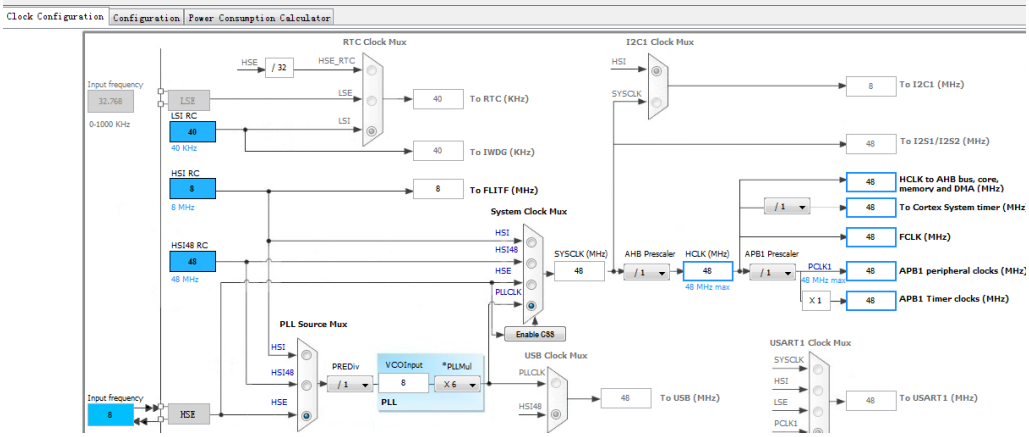
EXTI4_15 对应 GPIOA4 ～ GPIOA15,或者是 GPIOB4～GPIOB15 其余的 GPIO 组也是一样

GPIO 组里面每个管脚的 EXTI 中断线不一样，但是 GPIO 组的中断线一样，比如 GPIOA0,GPIOB0 用的都是同一个中断线，所以这两个 IO 口共享一个中断函数。

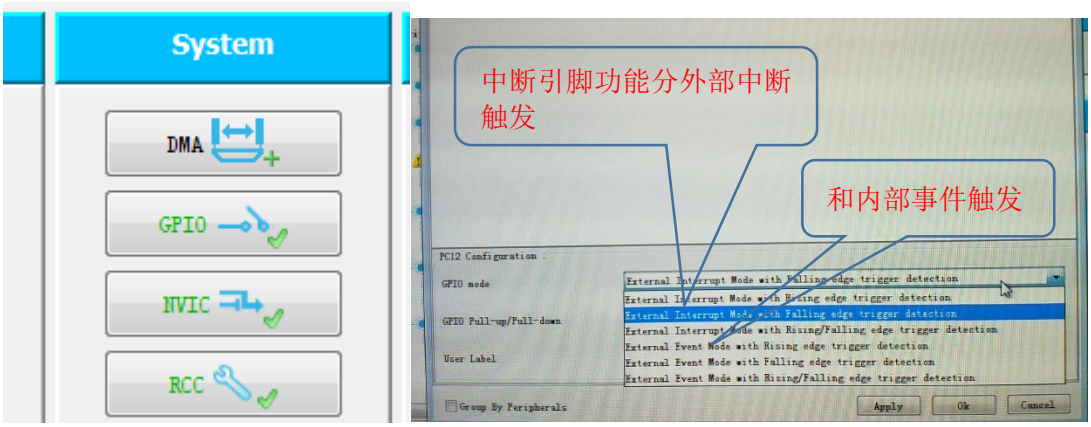


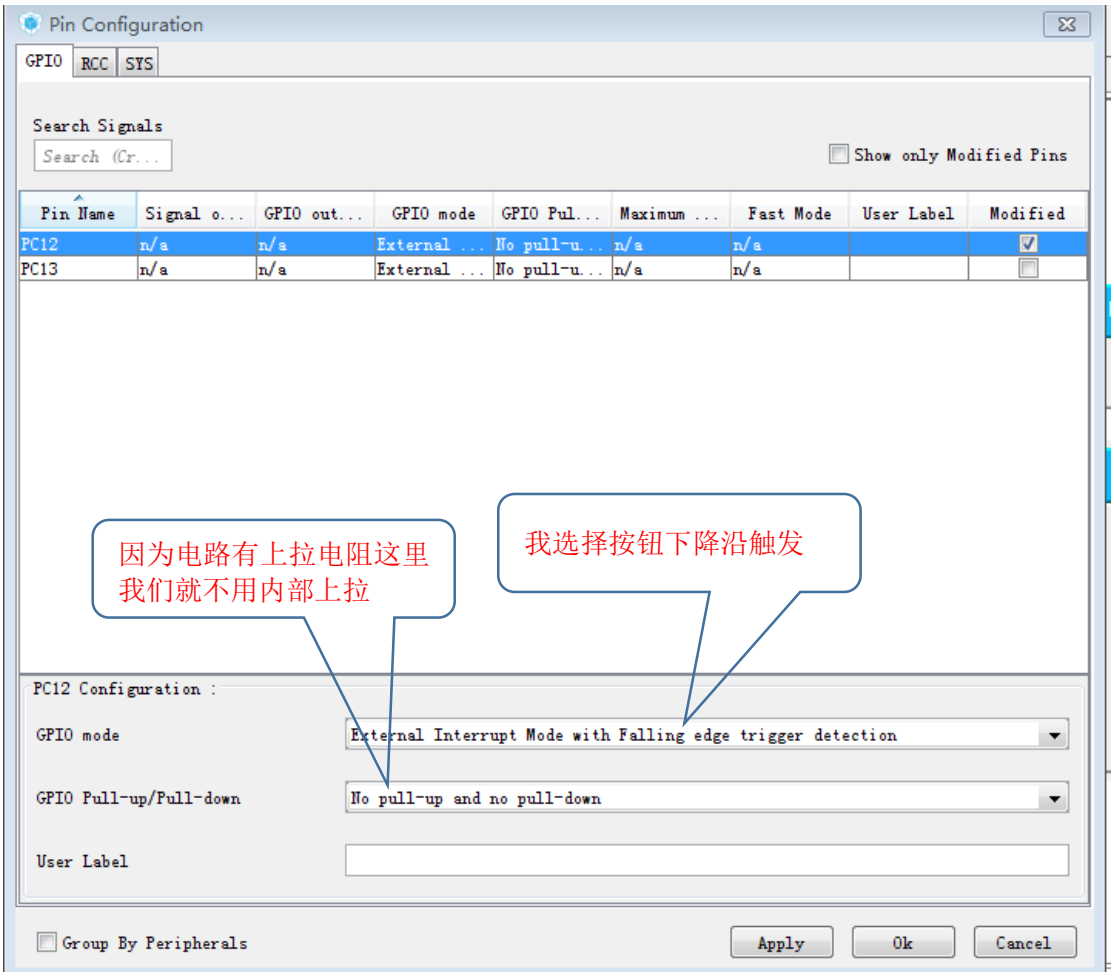
我们还是以 PC12,PC13 按键为例。





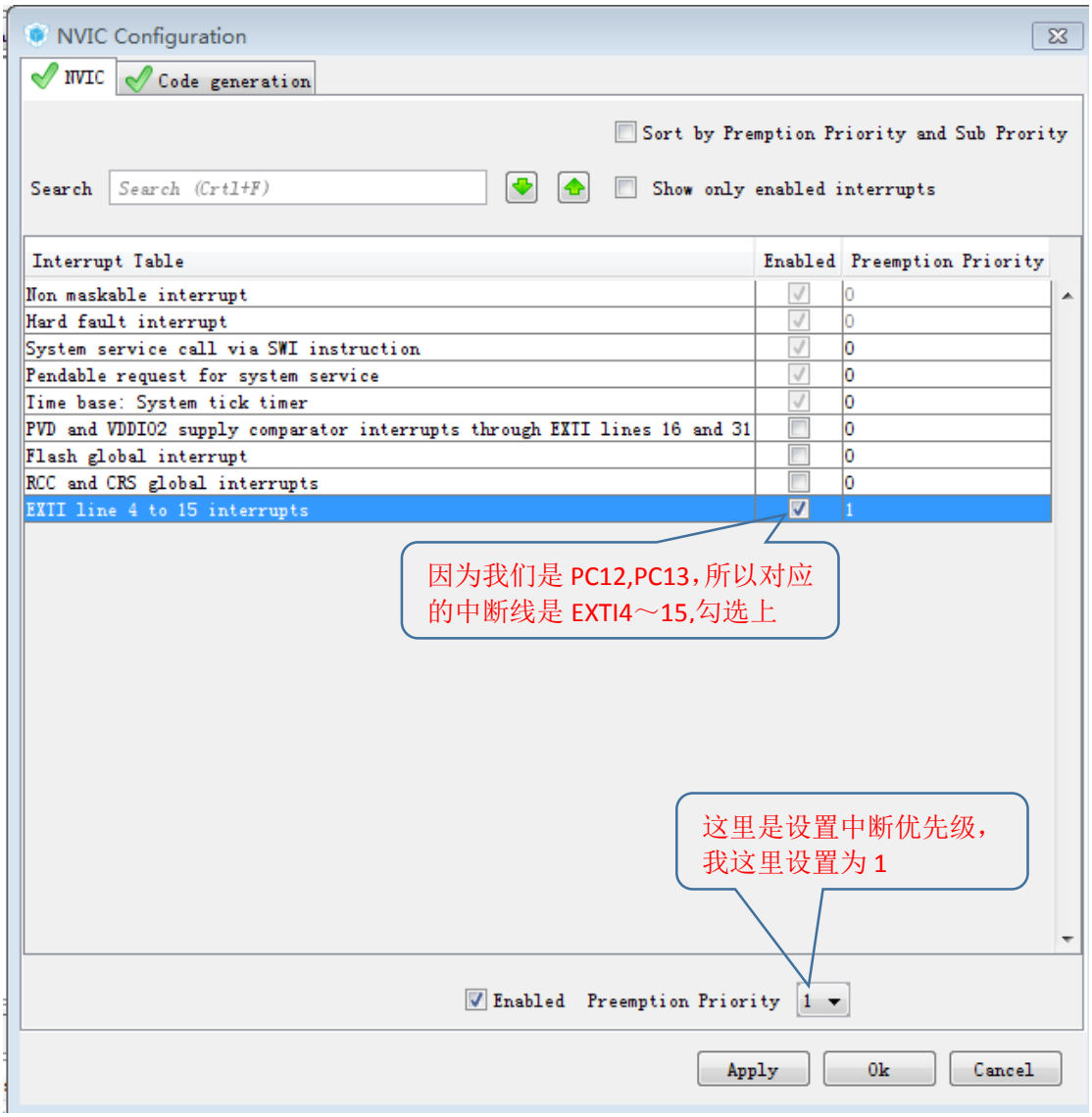
时钟设置还是和标准的外部 8M 晶振一样。倍频 48M





然后取名字确定。引脚外部中断功能只分为上升沿触发，下降沿触发，双边沿触发，没有高电平触发和低电平触发。

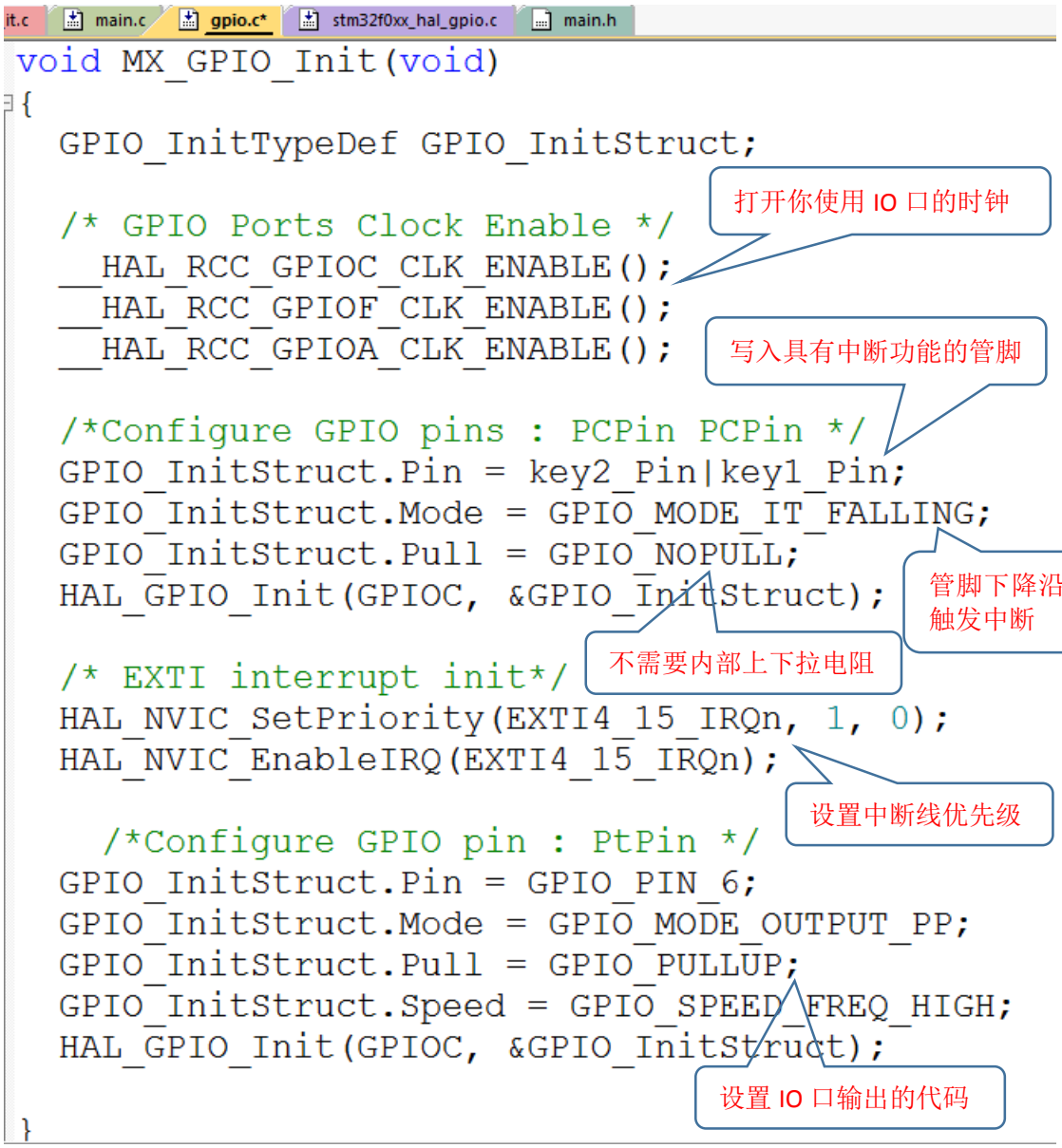
设置中断优先级

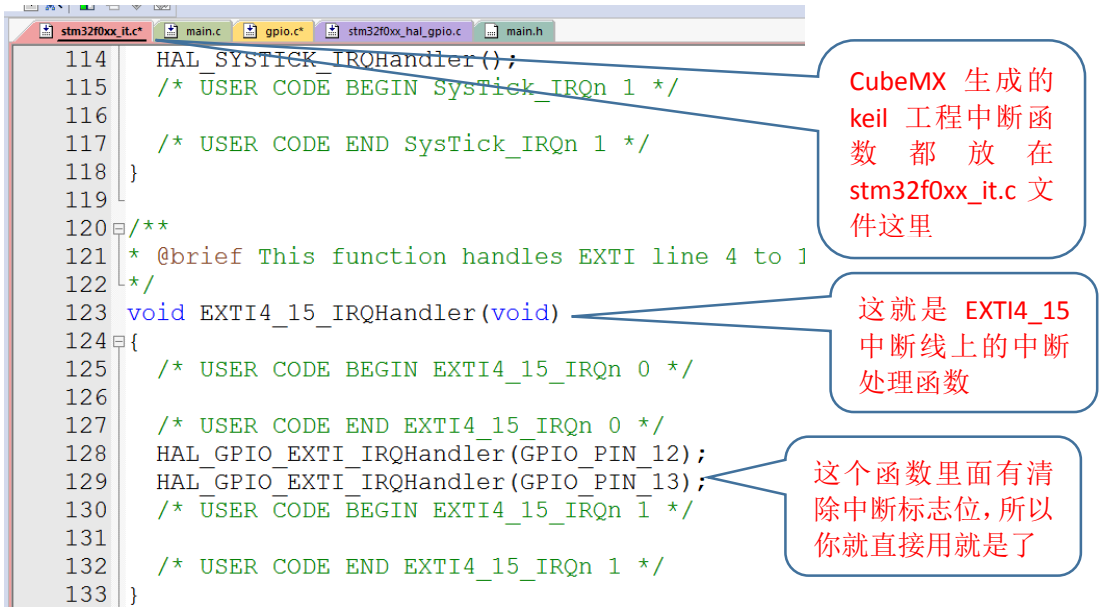


然后点击 OK

生成 keil 工程

```
int main(void)
{

  /* Reset of all peripherals, Initializes the Flash
  HAL_Init();
  /* Configure the system clock */
  SystemClock_Config();
  /* Initialize all configured peripherals */
  MX_GPIO_Init();

  while (1)
  {
```


it.c
main.c
gpio.c*
stm32f0xx_hal_gpio.c
main.h
void MX_GPIO_Init(void)
{
GPIO_InitTypeDef GPIO_InitStruct;
/* GPIO Ports Clock Enable */
__HAL_RCC_GPIOC_CLK_ENABLE();
__HAL_RCC_GPIOF_CLK_ENABLE();
__HAL_RCC_GPIOA_CLK_ENABLE();
/*Configure GPIO pins : PCPin PCPin */
GPIO_InitStruct.Pin = key2_Pin|key1_Pin;
GPIO_InitStruct.Mode = GPIO_MODE_IT_FALLING;
GPIO_InitStruct.Pull = GPIO_NOPULL;
HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
/* EXTI interrupt init*/
HAL_NVIC_SetPriority(EXTI4_15_IRQn, 1, 0);
HAL_NVIC_EnableIRQ(EXTI4_15_IRQn);
/*Configure GPIO pin : PtPin */
GPIO_InitStruct.Pin = GPIO_PIN_6;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Pull = GPIO_PULLUP;
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_HIGH;
HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
}
打开你使用 IO 口的时钟
写入具有中断功能的管脚
管脚下降沿触发中断
不需要内部上下拉电阻
设置中断线优先级
设置 IO 口输出的代码


stm32f0xx_it.c*
main.c
gpio.c*
stm32f0xx_hal_gpio.c
main.h
114 HAL_SYSTICK_IRQHandler();
115 /* USER CODE BEGIN SysTick_IRQn 1 */
116
117 /* USER CODE END SysTick_IRQn 1 */
118 }
119
120 /**
121 * @brief This function handles EXTI line 4 to 1
122 */
123 void EXTI4_15_IRQHandler(void)
124 {
125 /* USER CODE BEGIN EXTI4_15_IRQn 0 */
126
127 /* USER CODE END EXTI4_15_IRQn 0 */
128 HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_12);
129 HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_13);
130 /* USER CODE BEGIN EXTI4_15_IRQn 1 */
131
132 /* USER CODE END EXTI4_15_IRQn 1 */
133 }
CubeMX 生成的 keil 工程中断函数都放在 stm32f0xx_it.c 文件这里
这就是 EXTI4_15 中断线上的中断处理函数
这个函数里面有清除中断标志位，所以你就直接用就是了

我觉得把中断函数放在单独的一个文件里面我感觉不爽

```
//void EXTI4_15_IRQHandler(void)
//{
  /* USER CODE BEGIN EXTI4_15_IRQn 0 */

  /* USER CODE END EXTI4_15_IRQn 0 */
//  HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_12);
//  HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_13);
  /* USER CODE BEGIN EXTI4_15_IRQn 1 */

  /* USER CODE END EXTI4_15_IRQn 1 */
//}
```

把中断函数复制到你想要的文件下面，删除 it.c 里面的中断函数

我直接把中断函数复制到 main.c 主函数文件下面

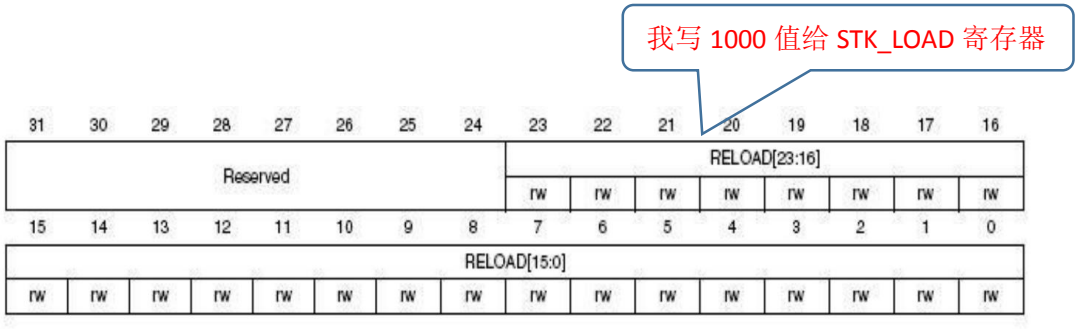
```
void EXTI4_15_IRQHandler(void)
{
  if(HAL_GPIO_ReadPin(GPIOC,GPIO_PIN_13)==GPIO_PIN_RESET)
  {
    HAL_GPIO_WritePin(GPIOC, GPIO_PIN_6, GPIO_PIN_RESET);

  }

  if(HAL_GPIO_ReadPin(GPIOC,GPIO_PIN_12)==GPIO_PIN_RESET)
  {

    HAL_GPIO_WritePin(GPIOC, GPIO_PIN_6, GPIO_PIN_SET);

  }

  HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_12);
  HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_13);
}
```

因为我的两个 IO 口都是用的同一个中断线，所以我就只有用判断 IO 口的方法来确定是哪个 IO 口发生了中断

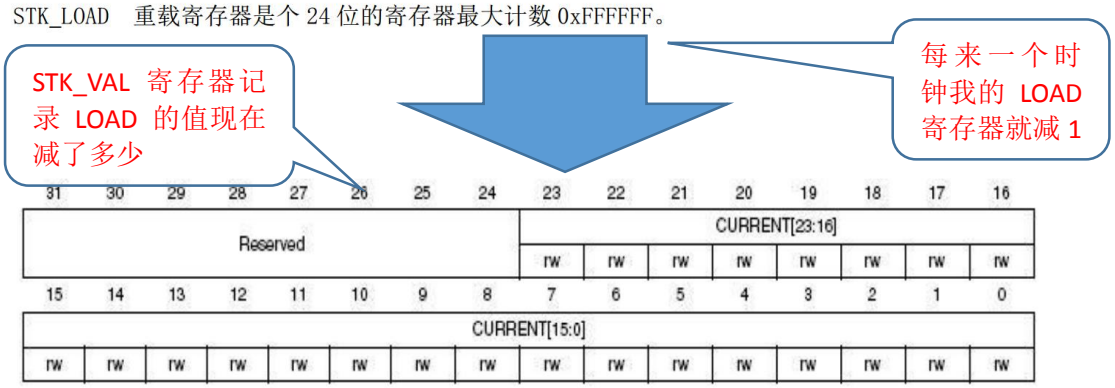
清除中断标志位

外部中断就讲解完毕了

Systick 内核定时器做延时函数



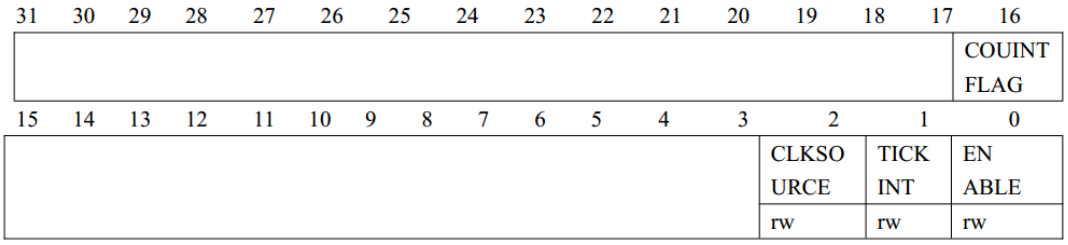
Systick 是一个递减的定时器，当定时器递减至 0 时，重载寄存器中的值就会被重装载，继续开始递减。STK_LOAD 重载寄存器是个 24 位的寄存器最大计数 0xFFFFFF。

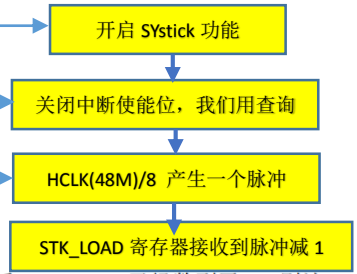


也是个 24 位的寄存器，读取时返回当前倒计数的值，写它则使之清零，同时还会清除在 SysTick 控制及状态寄存器中的 COUNTFLAG 标志。

Systick 时钟开启流程

17.1.2 STK_CTRL 控制寄存器





第 0 位：ENABLE，Systick 使能位
（0：关闭 Systick 功能；1：开启 Systick 功能）
第 1 位：TICKINT，Systick 中断使能位
（0：关闭 Systick 中断；1：开启 Systick 中断）
第 2 位：CLKSOURCE，Systick 时钟源选择
（0：使用 HCLK/8 作为 Systick 时钟；1：使用 HCLK 作为 Systick 时钟）
第 16 位：COUNTFLAG，Systick 计数比较标志，如果在上次读取本寄存器后，SysTick 已经数到了 0，则该位为 1。如果读取该位，该位将自动清零。

```
HAL_Delay(500);
```

这个 HAL 库里面的延时函数采用了中断功能，所以这个延时函数精确度不高。我们还是用查询法来完成延时函数。

17.1.5 精准微秒延时函数

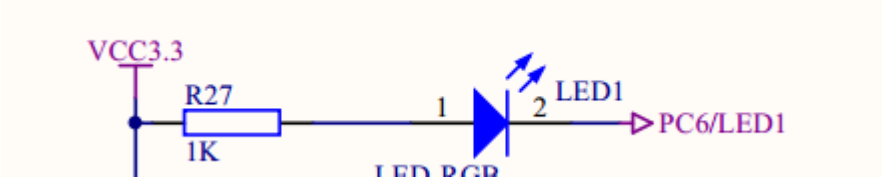
STM32F1 系列是 72M，所以这里应该是 72/8 = 9M，那么 9M=9000000 给 systick，也就是每秒钟可以记录 9000000 个数。
因为 1 秒记录 9000000 个数，那么 1ms 就要记录 9000 个数，那么 1us 就要记录 9 个数。所以这里是 9

```
void delay_us(u32 nus)
{
    u32 temp;
    SysTick->LOAD = 9*nus;//这里是 F1 系列参数，其他系列需要更改系数。
    SysTick->VAL=0X00;//清空计数器
    SysTick->CTRL=0X01;//使能，减到零是无动作，采用外部时钟源
    do
    {
       temp=SysTick->CTRL;//读取当前倒计数值
    }while((temp&0x01)&&(!(temp&(1<<16))));//等待时间到达
    SysTick->CTRL=0x00; //关闭计数器
    SysTick->VAL =0X00; //清空计数器
}
```

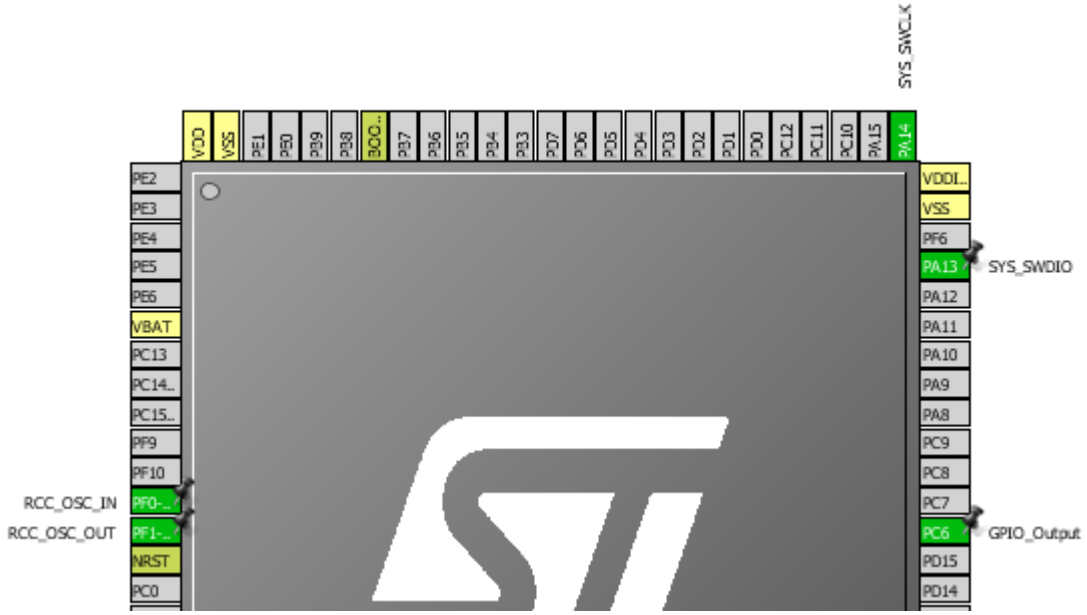
如果 STM32f1 精确延时 1ms,就将 LOAD 的 9 改成 9000 即可

```
SysTick->LOAD = 9000*nms;//这里是 F1 系列参数,其他系列需要更改系数。
```

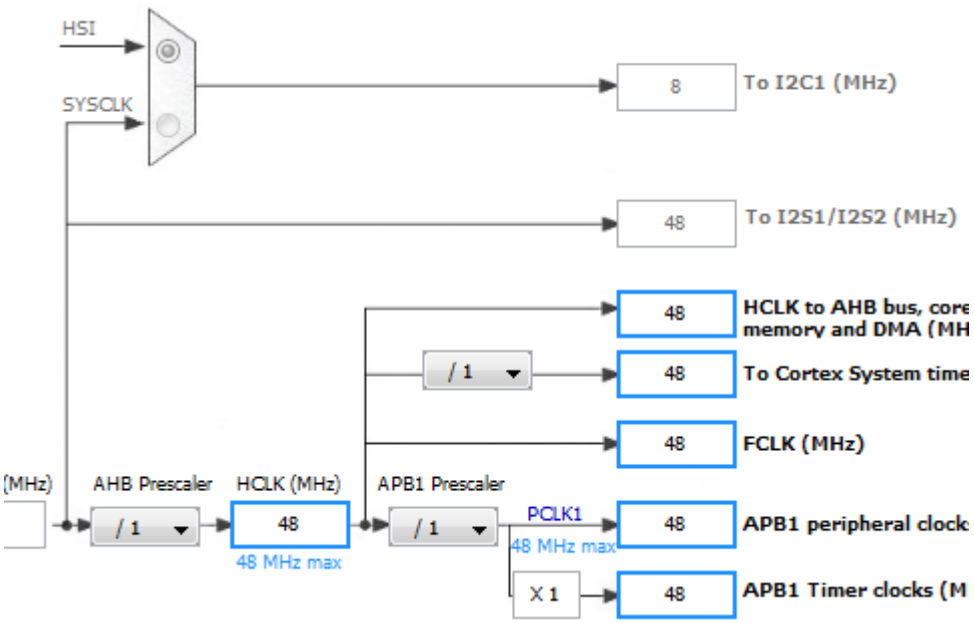
下面我们再来看看 STM32F0 的延时函数



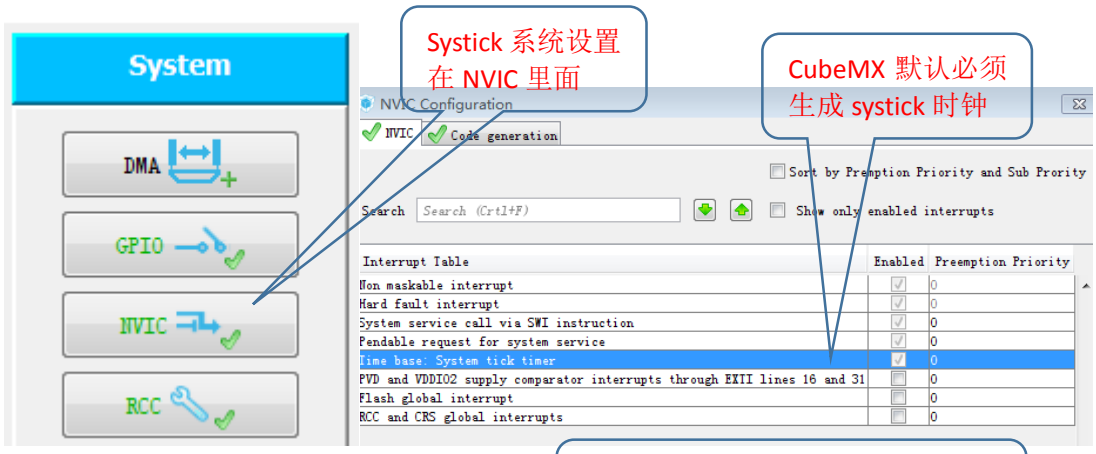
我们还是用 PC6 引脚驱动 LED 延时为例

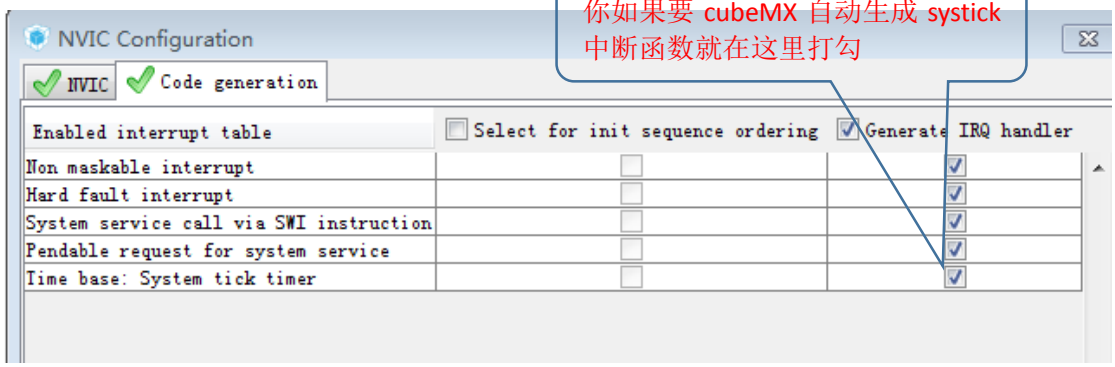


引脚定义老样子



时钟还是外部晶振 48M





然后生成 keil 工程

因为 cubeMX 生成的 systick 时钟是不准的，但是为了芯片上电初始化 systick 时钟

将 keil 工程主函数下的 SystemClock_Config 里面的 systick 部分初始化函数去掉

```
  */
HAL_SYSTICK_Config(HAL_RCC_GetHCLKFreq()/1000);

  /**Configure the Systick
  */
HAL_SYSTICK_CLKSourceConfig(SYSTICK_CLKSOURCE_HCLK);

/* SysTick_IRQn interrupt configuration */
HAL_NVIC_SetPriority(SysTick_IRQn, 0, 0);
```

屏蔽掉这三个函数

```
   */
  //HAL_SYSTICK_Config(HAL_RCC_GetHCLKFreq()/1000);

    /**Configure the Systick
    */
  //HAL_SYSTICK_CLKSourceConfig(SYSTICK_CLKSOURCE_HCLK);

  /* SysTick_IRQn interrupt configuration */
  //HAL_NVIC_SetPriority(SysTick_IRQn, 0, 0);
```

屏蔽成功。

然后在主函数上自己实现 systick 查询方式的微妙和毫秒延时

```
#include "main.h"
#include "stm32f0xx_hal.h"
void delay_us(unsigned int nus)//微妙1us是2.2us
{
    unsigned int temp;
    SysTick->LOAD = 6*nus;
    SysTick->VAL = 0x00;
    SysTick->CTRL = 0x01;
    do
    {
            temp = SysTick->CTRL;
    }while((temp&0x01)&&(!(temp&(1<<16))));
    SysTick->CTRL = 0x00;
    SysTick->VAL = 0x00;
}
```

和 STM32F1 一 样 ，HCLK(48M)/8=6000000，1 微妙=6

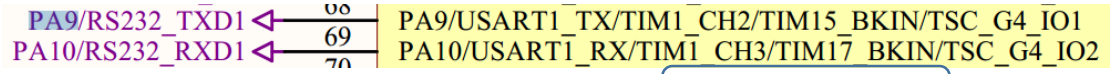
```
void delay_ms(unsigned int nus)//毫秒1ms就是1ms
{
    unsigned int temp;
    SysTick->LOAD = 6000*nus;
    SysTick->VAL = 0x00;
    SysTick->CTRL = 0x01;
    do
    {
            temp = SysTick->CTRL;
    }while((temp&0x01)&&(!(temp&(1<<16))));
    SysTick->CTRL = 0x00;
    SysTick->VAL = 0x00;
}
```

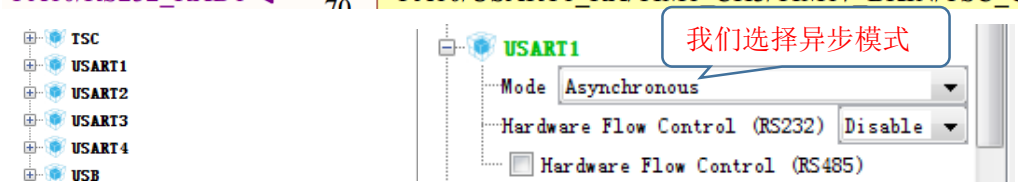
这里自然是 6000

```
while (1)
{
  HAL_GPIO_WritePin(GPIOC, GPIO_PIN_6, GPIO_PIN_RESET);
  delay_us(1);
  HAL_GPIO_WritePin(GPIOC, GPIO_PIN_6, GPIO_PIN_SET);
  delay_us(1);
}
```

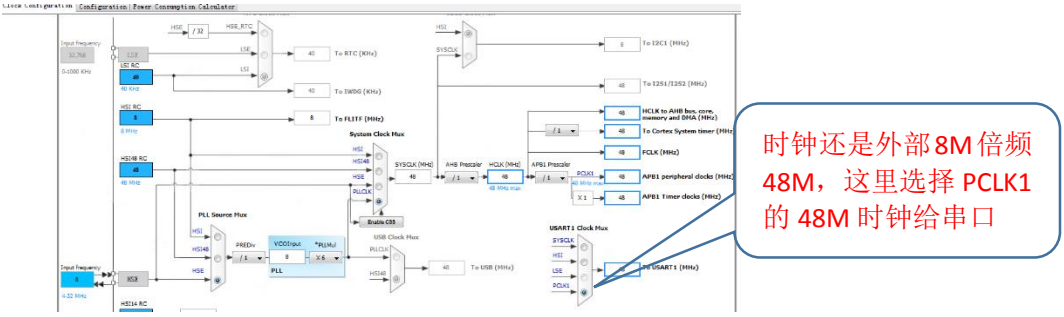
程序执行正常，systick 延时讲解完毕

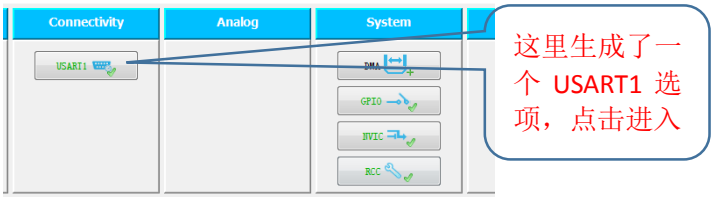
USART 串口通讯

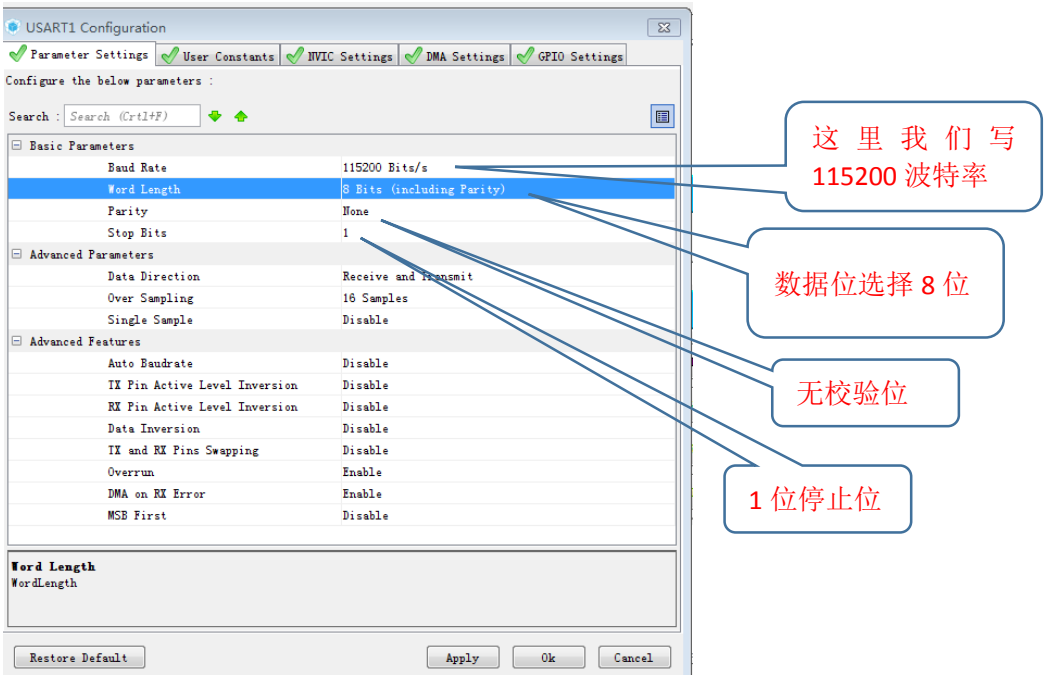


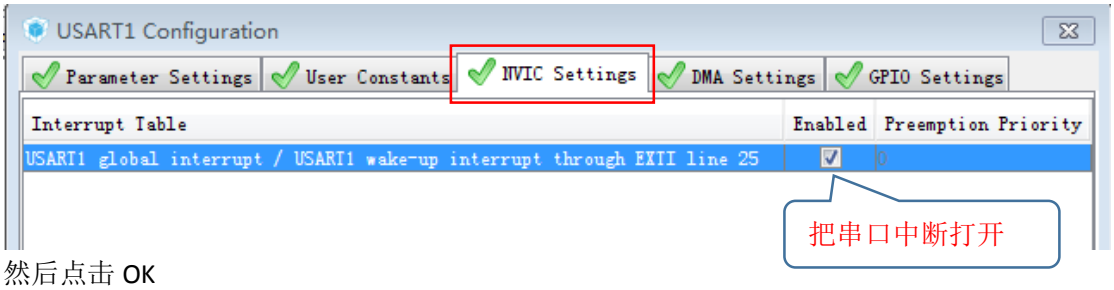


其余的晶振啊，SWD 接口都要选择上。

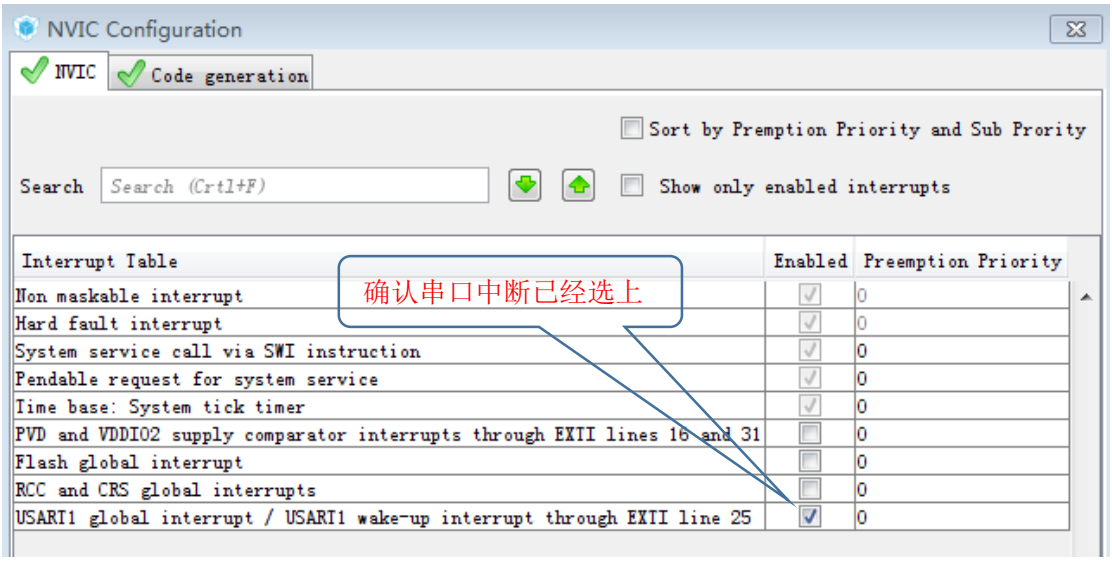


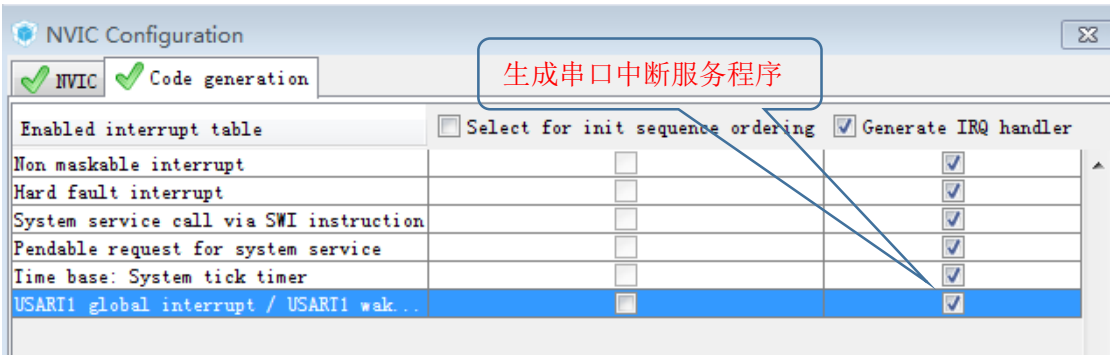






然后点击 OK



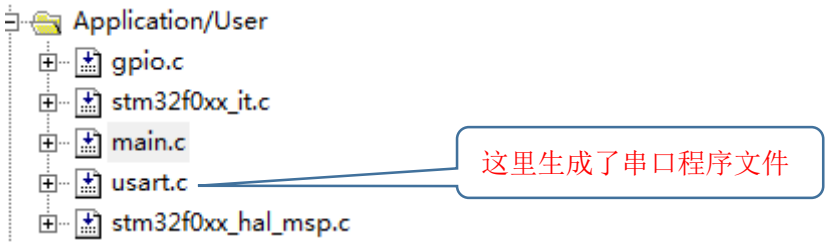


然后点击 OK，设置一下工具栏，生成 keil 工程

```
int main(void)
{

  /* MCU Configuration--------------------------
  /* Reset of all peripherals, Initializes the F
  HAL_Init();
  /* Configure the system clock */
  SystemClock_Config();
  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_USART1_UART_Init();
  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
```

这里只初始化 IO 时钟

这里初始化串口




```c
void MX_USART1_UART_Init(void)
{

  huart1.Instance = USART1;
  huart1.Init.BaudRate = 115200;
  huart1.Init.WordLength = UART_WORDLENGTH_8B;
  huart1.Init.StopBits = UART_STOPBITS_1;
  huart1.Init.Parity = UART_PARITY_NONE;
  huart1.Init.Mode = UART_MODE_TX_RX;
  huart1.Init.HwFlowCtl = UART_HWCONTROL_NONE;
  huart1.Init.OverSampling = UART_OVERSAMPLING_16;
  huart1.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE;
  huart1.AdvancedInit.AdvFeatureInit = UART_ADVFEATURE_NO_INIT;
  if (HAL_UART_Init(&huart1) != HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

}
```

使用串口 USART1

波特率 115200

8 位数据位

1 位停止位

无校验位

串口配置完毕，发送一个字节数据

```c
static unsigned char rxdata = 0xfa;
int main(void)
{

  /* MCU Configuration--------------------------
  /* Reset of all peripherals, Initializes the
  HAL_Init();
  /* Configure the system clock */
  SystemClock_Config();
  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_USART1_UART_Init();
  //HAL_UART_Receive_IT(&huart1,&rxdata,1);
  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
    HAL_UART_Transmit(&huart1,&rxdata,1,1);
  }
  /* USER CODE END 3 */

}
```

定义一个要发送到数据

用 HAL_UART_Transmit 发送

HAL_StatusTypeDef HAL_UART_Transmit(UART_HandleTypeDef *huart, uint8_t *pData, uint16_t Size, uint32_t Timeout)

UART_HandleTypeDef *huart ： 串口号

uint8_t *pData: 1 个字节的数据，8 位，也可以传入一个数组的数据

uint16_t Size：发送多少个字节，我这里是一个字节，所以天

uint32_t Timeout：这个 timeou 是决定发送一个字节要多少时间,单位为毫秒

当你波特率是 9600bit 的时候，9600/8 = 1200 个字节，1 秒钟发送 1200 个字节(1000ms/1200) = 0.83ms，那么我的 timeout 就要设置为 1ms

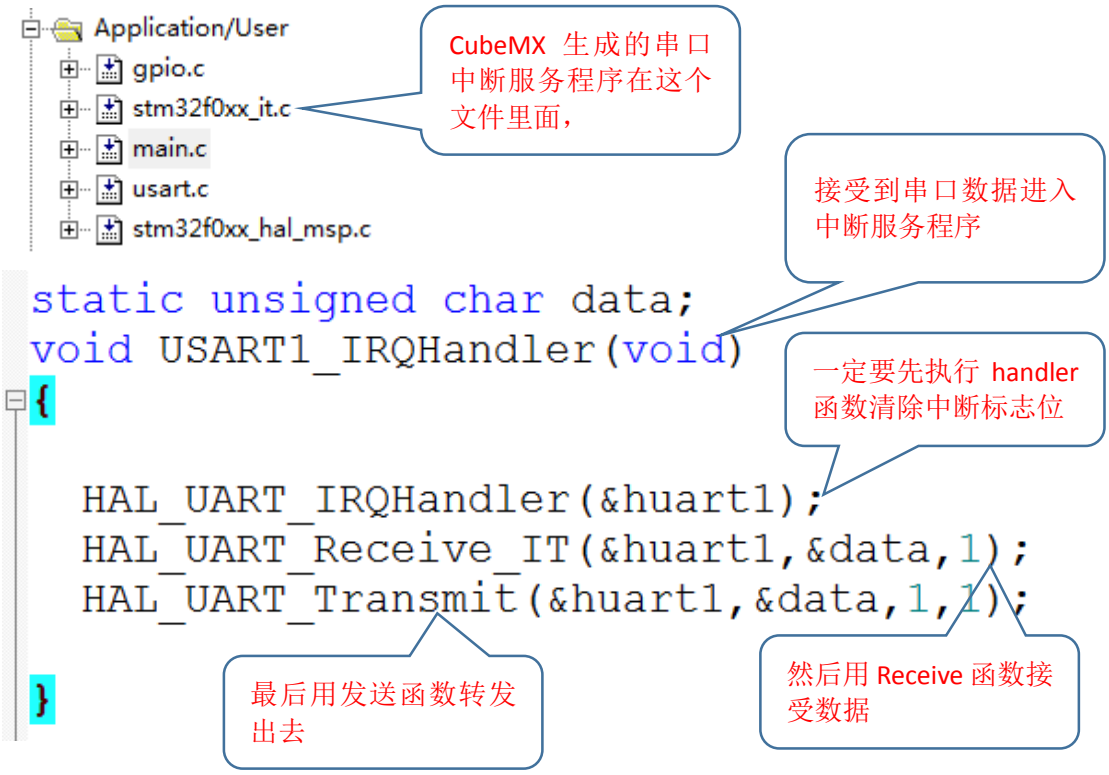
当你的波特率为 115200 的时候，1 秒钟发送 14400 字节，一个字节需要 0.0695ms，所以 timeout 设置 1ms 也是可以的

串口中断接受一个字节的数据

```c
#include "main.h"
#include "stm32f0xx_hal.h"
#include "usart.h"
#include "gpio.h"
```

```c
int main(void)
{
    unsigned char data;
  /* MCU Configuration--------------------------
  /* Reset of all peripherals, Initializes the
  HAL_Init();

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

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_USART1_UART_Init();
  HAL_UART_Receive_IT(&huart1,&data,1);
  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {

  }
```

因为串口接受程序和串口中断使能程序都在这个函数里面，所以主函数先执行这个函数是为了打开中断使能



HAL_StatusTypeDef HAL_UART_Receive_IT(UART_HandleTypeDef *huart, uint8_t *pData, uint16_t Size)

UART_HandleTypeDef *huart：串口号

uint8_t *pData：要接受的数据类型

uint16_t Size：接受多少个字节数据

这就是电脑发送给 F0 的数据又转回电脑。

串口映射成 printf 实现

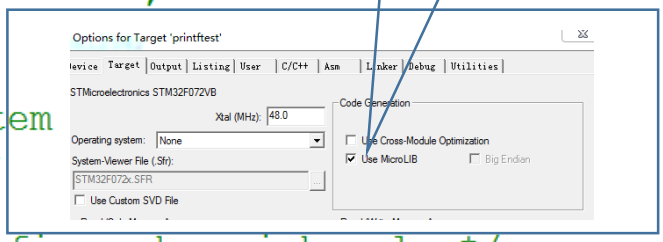
串口程序生成方法和前面的 USART 串口通讯方式一样。

```
#include "main.h"
#include "stm32f0xx_hal.h"
#include "usart.h"
#include "gpio.h"



/* Private function prototypes -----------------
void SystemClock_Config(void);

/* USER CODE BEGIN PFP */
/* Private function prototypes -----------------
int fputc(int ch,FILE *f)
{
    HAL_UART_Transmit(&huart1,(uint8_t *)&ch,1,1);
    return ch;
}

  */
int main(void)
{
  unsigned int d = 20;
  float f = 3.14;
  unsigned char c = 'c';
  unsigned int x= 0x1234;
  unsigned char s[]="abcdefg";
  /* Reset of all peripherals, Initializes the Fl
  HAL_Init();

  /* Configure the system
  SystemClock_Config();

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_USART1_UART_Init();

  while (1)
  {
    printf("xxxxx\r\n");
    printf("d = %d\r\n",d);
    printf("f = %f\r\n",f);
    printf("%c\r\n",c);
    printf("0x = %x\r\n",x);
    printf("%s\r\n",s);
    HAL_Delay(500);
```

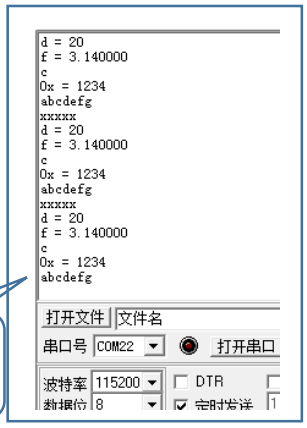
找个放全局函数的文件添加 fputc 函数

定义整形，浮点型，字符型，16 进制。
注意字符串初始化必须用数组承载，用指针不行

确保 keil 的 Use 配置上了的



打印各种类型



输出结果正常

这就是 STM32F0 芯片 printf 函数实现

CAN 总线实现

STM32f0CAN 总线固件库函数波特率计算

CAN 波特率=CAN 总线频率/(perscaler 分频器系数*(tq+tBS1+tBS2))

tq：值固定为 1

BRP： 设置范围 1～4，一般设置为 3，尽量别动。

CAN_BS1：设置范围 1～16

CAN_BS2：设置范围 1～8

设置注意：BS1 ≥ BS2(BS1 值必须大于等于 BS2)，BS2 ≥1 个 CAN 时钟周期，BS2 ≥ 2perscaler

公式分解:

$$\frac{HCLK}{(tBS2+tBS1+tq)\ X\ Perscaler} = baudrate$$

计算 CAN 波特率公式

$$\frac{CAN时钟(48M)}{(7+8+1)\ X\ 3} = 1M波特率$$

计算 1M 波特率

我们也可以设定波特率然后计算 BS1,BS2

$$\frac{48000000(48M)}{1000000(1M)} = 48$$

$$(tBS2+tBS1+1)\ X\ 3 = 48$$

$$(7+8+1)\ X\ 3 = 48$$

tBS2 = 7

tBS1 = 8

tq = 1

perscaler = 3

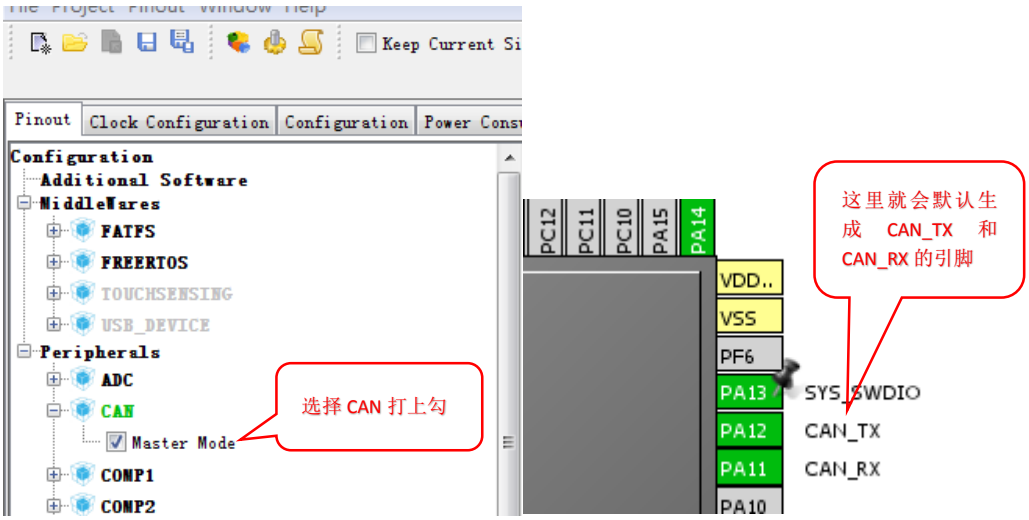
满足 BS1 大于 BS2 的要求

tq 不要变

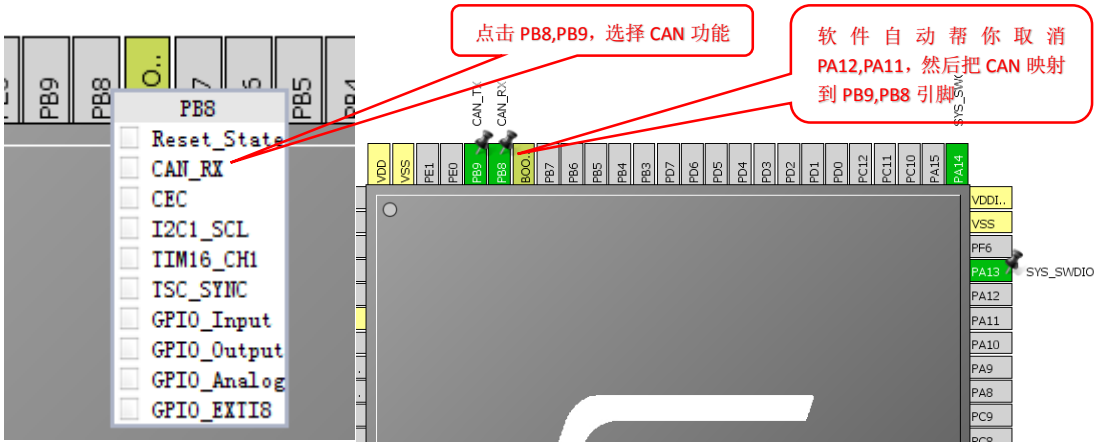
满足 BS2 大于 2 倍 perscaler 的要求

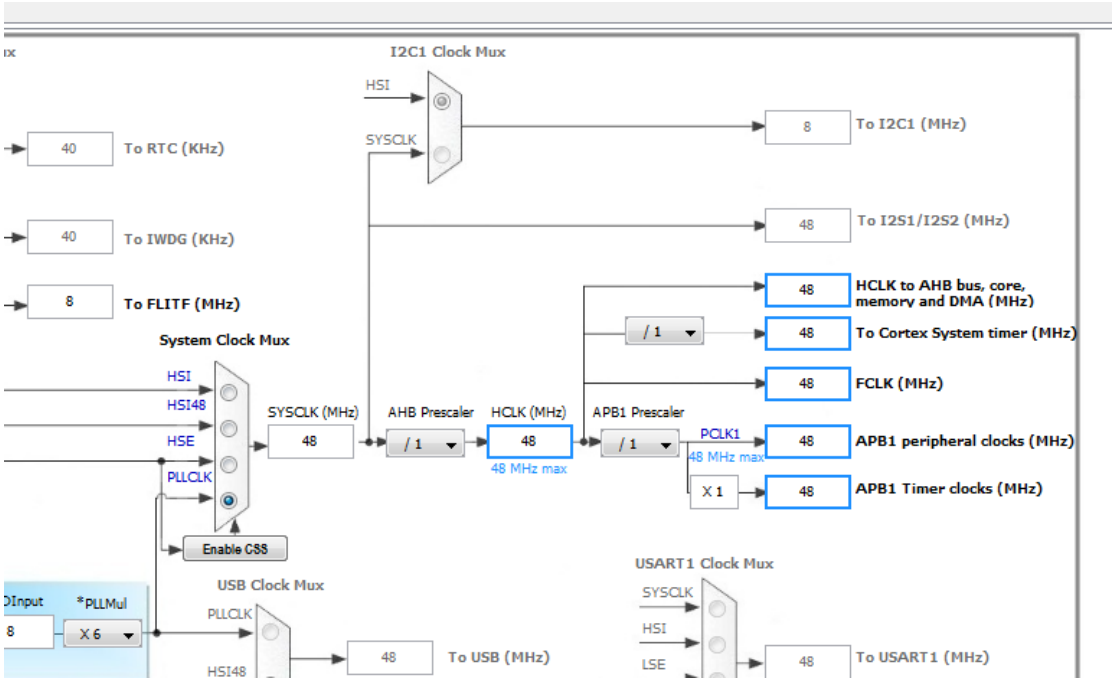
现在得到 tBS2 = 7, tBS1 = 8, tq = 1

打开 CubeMX 软件，生成 CAN 代码

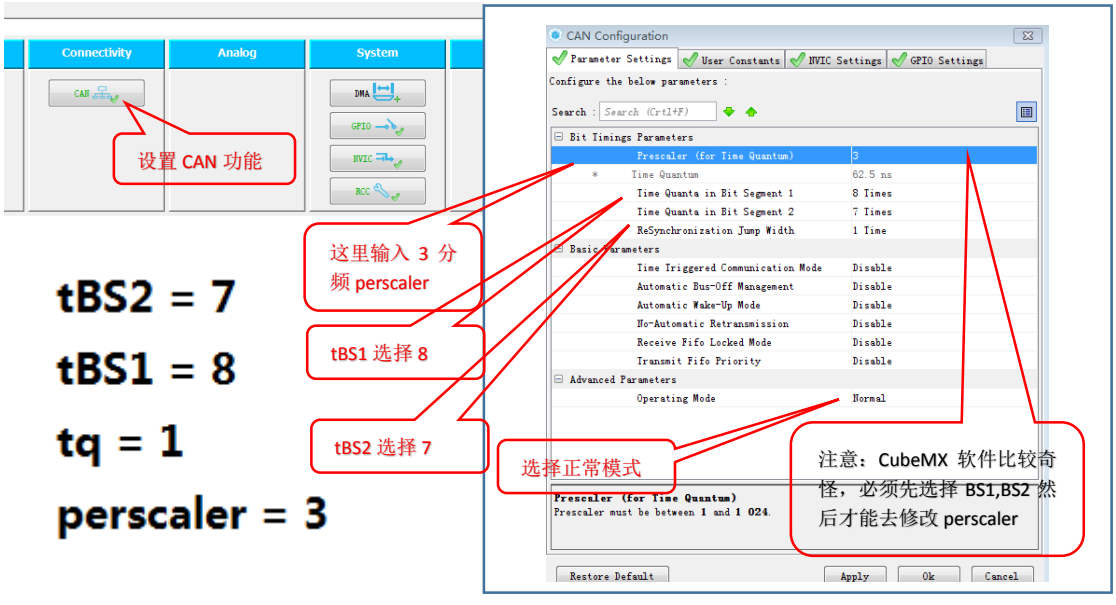


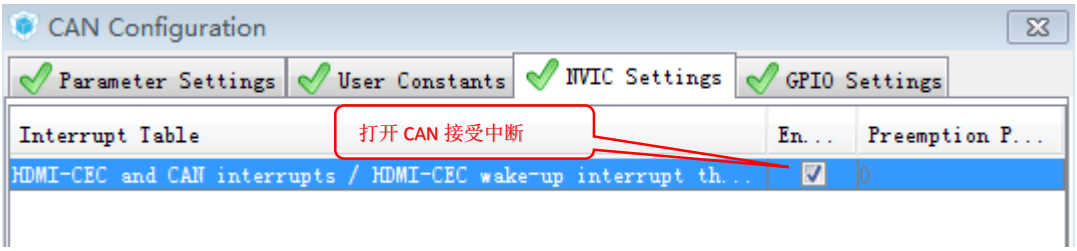
如果你的 PA12,PA11 引脚被其它功能占用了，那么你可以改成 PB8,PB9

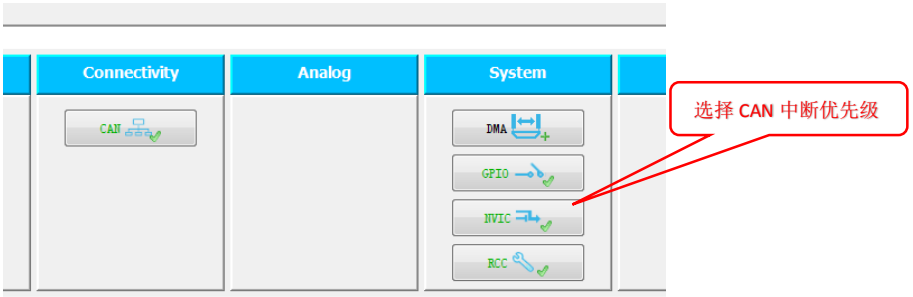


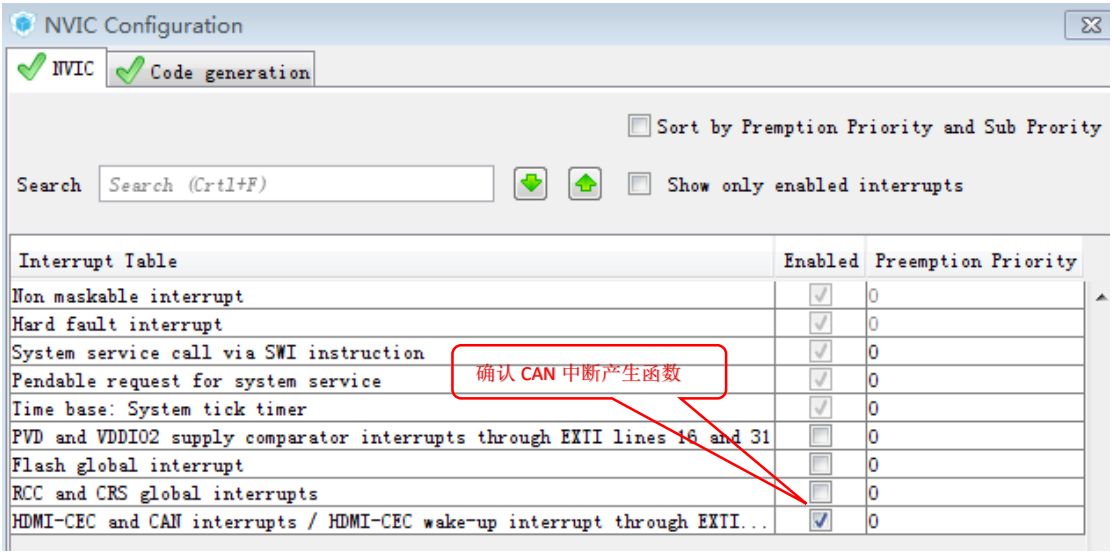


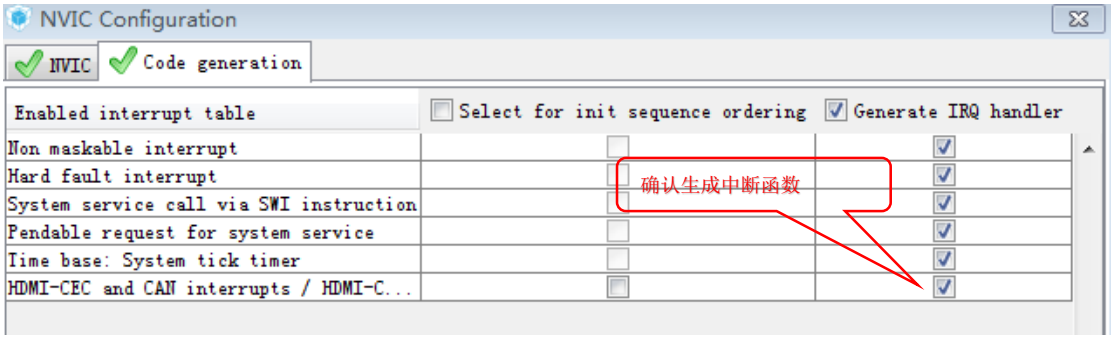
老样子选择外部晶振 8M，倍频 48M 系统时钟，CAN 时钟和系统时钟 HCLK 一样 48M











CubeMX 设置完毕，按照上面方法设置选项栏，然后生成 keil 文件。

```
int main(void)
{
  hcan.pTxMsg = &TxMsg;
  hcan.pRxMsg = &RxMsg;
  /* Reset of all peripherals, Initializes the
  HAL_Init();

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

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_USART1_UART_Init();
  MX_CAN_Init();
  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
```

确认初始化函数

在 can.c 文件下

```
/* CAN init function */
void MX_CAN_Init(void)
{

  hcan.Instance = CAN;
  hcan.Init.Prescaler = 3;
  hcan.Init.Mode = CAN_MODE_NORMAL;
  hcan.Init.SJW = CAN_SJW_1TQ;
  hcan.Init.BS1 = CAN_BS1_8TQ;
  hcan.Init.BS2 = CAN_BS2_7TQ;
  hcan.Init.TTCM = DISABLE;
  hcan.Init.ABOM = DISABLE;
  hcan.Init.AWUM = DISABLE;
  hcan.Init.NART = DISABLE;
  hcan.Init.RFLM = DISABLE;
  hcan.Init.TXFP = DISABLE;
  if (HAL_CAN_Init(&hcan) != HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

}
```

设备节点为 CAN

3 分频

正常模式

tq = 1

tBS1 = 8

tBS2 = 7

CAN 总线初始化完成，波特率为 1M

然后我们 STM32F0 的 CAN 控制器发送数据给电脑，我们发送标准数据帧(ID 是 16 位)

```
/* Includes ------------------
#include "main.h"
#include "stm32f0xx_hal.h"
#include "can.h"
#include "usart.h"
#include "gpio.h"

static CanTxMsgTypeDef TxMsg;
static CanRxMsgTypeDef RxMsg;
```

我们要在内存开辟一个 CAN 总线的结构体,这个一定要做

```c
int main(void)
{
  hcan.pTxMsg = &TxMsg;
  hcan.pRxMsg = &RxMsg;
  /* Reset of all peripherals, Initializes the
  HAL_Init();

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

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_USART1_UART_Init();
  MX_CAN_Init();
  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
   while (1)
   {

     hcan.pTxMsg->StdId = 0x12;
//   hcan.pTxMsg->ExtId = 0x01;
     hcan.pTxMsg->IDE = CAN_ID_STD;
     hcan.pTxMsg->RTR =CAN_RTR_DATA;
     hcan.pTxMsg->DLC = 8;

     hcan.pTxMsg->Data[0] = 0x01;
     hcan.pTxMsg->Data[1] = 0x02;
     hcan.pTxMsg->Data[2] = 0x03;
     hcan.pTxMsg->Data[3] = 0x04;
     hcan.pTxMsg->Data[4] = 0x05;
     hcan.pTxMsg->Data[5] = 0x06;
     hcan.pTxMsg->Data[6] = 0x07;
     hcan.pTxMsg->Data[7] = 0x08;
     HAL_CAN_Transmit(&hcan,10);
     //HAL_Delay(500);
```

CAN 总线发送的数据全部要向 hcan.pTxMsg 写入。所以我们得让 hcan.pTxMsg 指向 TxMsg 这个开辟的内存空间
这样 hcan.pTxMsg 指针就有了确切的地址

写入标准 ID,这个值最大范围是 0～0x7FF

我们发送标准帧，所以 ExtId 不写

数据帧

发送数据个数

CAN_ID_STD 是标准 ID，因为我们 Extid 没有写，所以发送的是标准 ID

发送数据

将设置好的 CAN 参数写入 CAN 寄存器

文本显示

ID	Da0	Da1	Da2	Da3	Da4	Da5	Da6	Da7	Len	Fmt	Typ
0x012	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x08	Standard	Data
0x012	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x08	Standard	Data

我们再来发送扩展数据帧(ID 是 29 位)

扩展 ID 发送，can 波特率和上面一样，can 缓冲区 pTxMsg 和上面一样，就修改 ID 就行

```
while (1)
{

    hcan.pTxMsg->ExtId = 0x1234;
    hcan.pTxMsg->IDE = CAN_ID_EXT;
    hcan.pTxMsg->RTR =CAN_RTR_DATA;
    hcan.pTxMsg->DLC = 8;

    hcan.pTxMsg->Data[0] = 0x10;
    hcan.pTxMsg->Data[1] = 0x20;
    hcan.pTxMsg->Data[2] = 0x30;
    hcan.pTxMsg->Data[3] = 0x40;
    hcan.pTxMsg->Data[4] = 0x50;
    hcan.pTxMsg->Data[5] = 0x60;
    hcan.pTxMsg->Data[6] = 0x70;
    hcan.pTxMsg->Data[7] = 0x80;
    HAL_CAN_Transmit(&hcan,10);
    //HAL_Delay(500);
}
```

将 StdId 去掉，用 ExtId 来承载扩展数据 ID，范围 0～0x1FFFFFFF

将 CAN_ID_STD 标准 ID 模式改成扩展 ID 模式 CAN_ID_EXT

其余不变，编译下载运行。

文本显示

ID	Da0	Da1	Da2	Da3	Da4	Da5	Da6	Da7	Len	Fmt	Typ
0x00001234	0x10	0x20	0x30	0x40	0x50	0x60	0x70	0x80	0x08	Extended	Data
0x00001234	0x10	0x20	0x30	0x40	0x50	0x60	0x70	0x80	0x08	Extended	Data

CAN 总线实现接受

CAN 总线是在中断程序里面接受主机发过来的数据,然后我用串口打印出来

我们要先在主函数上加屏蔽滤波功能，接受程序必须先这样,因为内核规定要加屏蔽滤波函数 CAN 才能正常工作。

```
void CAN_filter_fig()
{
    CAN_FilterConfTypeDef filter;
    filter.FilterNumber = 0;
    filter.FilterMode = CAN_FILTERMODE_IDMASK;
    filter.FilterScale = CAN_FILTERSCALE_32BIT;

    filter.FilterIdHigh = 0;
    filter.FilterIdLow = 0;
    filter.FilterMaskIdHigh = 0;
    filter.FilterMaskIdLow = 0;
    filter.FilterFIFOAssignment = 0;
    filter.FilterActivation = ENABLE;

    HAL_CAN_ConfigFilter(&hcan,&filter);
}
```

现在我们接受所有的数据，所以我们这里不做屏蔽处理

在屏蔽滤波里面一定要选择激活该功能

然后将 CAN 总线设置的屏蔽滤波参数写入寄存器

```
int main(void)
{
  CanRxMsgTypeDef RxMsg;
  hcan.pRxMsg = &RxMsg;
  /* Reset of all peripherals, Initializes the
  HAL_Init();

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

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_CAN_Init();
  MX_USART1_UART_Init();

  CAN_filter_fig();
  HAL_CAN_Receive_IT(&hcan,CAN_FIFO0);
```

在主函数里面要先给接受 CAN 总线的指针赋地址

一定要初始化屏蔽滤波

一定要在初始化时提前打开一次 CAN 接受中断，接受的数据放在 FIFO0

```
void CEC_CAN_IRQHandler(void)
{

  HAL_CAN_IRQHandler(&hcan);

}
```

本来 CAN 中断服务函数是在 it.c 文件里面的，但是我为了方便就剪贴到主函数去

```
#include "main.h"
#include "stm32f0xx_hal.h"
#include "stm32f0xx_it.h"
#include "can.h"
#include "usart.h"
#include "gpio.h"
```

记着主函数要加 it.h 和 can.h 头文件

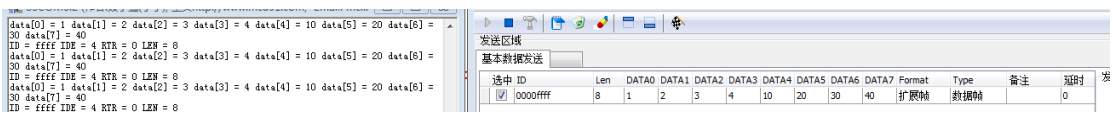
```
void CEC_CAN_IRQHandler(void)
{

  HAL_CAN_IRQHandler(&hcan);
  rID = hcan.pRxMsg->ExtId;
  rIDE = hcan.pRxMsg->IDE;//标准ID还是扩展ID
  rRTR = hcan.pRxMsg->RTR;//数据帧还是远程帧
  rlenght = hcan.pRxMsg->DLC;//数据长度

  data[0] = hcan.pRxMsg->Data[0];
    data[1] = hcan.pRxMsg->Data[1];
    data[2] = hcan.pRxMsg->Data[2];
    data[3] = hcan.pRxMsg->Data[3];
    data[4] = hcan.pRxMsg->Data[4];
    data[5] = hcan.pRxMsg->Data[5];
    data[6] = hcan.pRxMsg->Data[6];
    data[7] = hcan.pRxMsg->Data[7];

    printf("ID = %x IDE = %d RTR = %d LEN = %d\r\n",rID,rIDE,rRTR,rlenght);
    printf("data[0] = %x data[1] = %x data[2] = %x data[3] = %x\
    data[4] = %x data[5] = %x data[6] = %x data[7] = %x\r\n",
    data[0],data[1],data[2],data[3],data[4],data[5],data[6],data[7]);
    HAL_CAN_Receive_IT(&hcan,CAN_FIFO0);

}
```

在中断函数里面先清除中断标志位

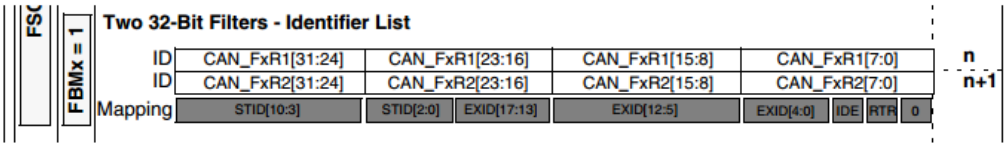
接受数据

CAN 中断服务程序接受完数据会自动关闭 CAN 中断，所以这里还要再打开

主函数死循环什么都不用做。



4.2. 32位宽的列表模式



修改 CAN_filter_fig 函数里面的屏蔽参数

```
void CAN_filter_fig()
{
    CAN_FilterConfTypeDef filter;
    uint32_t StdId =0x321; //接受发送的标准ID号
    uint32_t ExtId = 0x1234;//接受发送的扩展ID号


    filter.FilterNumber = 0;
    filter.FilterMode = CAN_FILTERMODE_IDLIST;
    filter.FilterScale = CAN_FILTERSCALE_32BIT;

    filter.FilterIdHigh = StdId<<5;
    filter.FilterIdLow = 0| CAN_ID_STD;


    filter.FilterMaskIdHigh = ((ExtId<<3)>>16)&0xffff;
    filter.FilterMaskIdLow = ((ExtId<<3)&0xffff)|CAN_ID_EXT;
    filter.FilterFIFOAssignment = CAN_FILTER_FIFO0;
    filter.FilterActivation = ENABLE;

    HAL_CAN_ConfigFilter(&hcan,&filter);
}
```

选择为标识符模式，就是电脑发送的 ID 和我 CAN 接受到 ID 一一对应的 ID 数据才允许接收进 STM32 芯片

设置 STM32 允许接受的标识符 ID

接受的数据放入 fifo0

设置 STM32 允许接受的扩展 ID

```
int main(void)
{

  CanRxMsgTypeDef RxMsg;
  hcan.pRxMsg = &RxMsg;
  /* Reset of all peripherals, Initializes the
  HAL_Init();

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

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_CAN_Init();
  MX_USART1_UART_Init();

  CAN_filter_fig();
  HAL_CAN_Receive_IT(&hcan,CAN_FIFO0);
```

设置 CAN 接受数据的地址，和上面一样

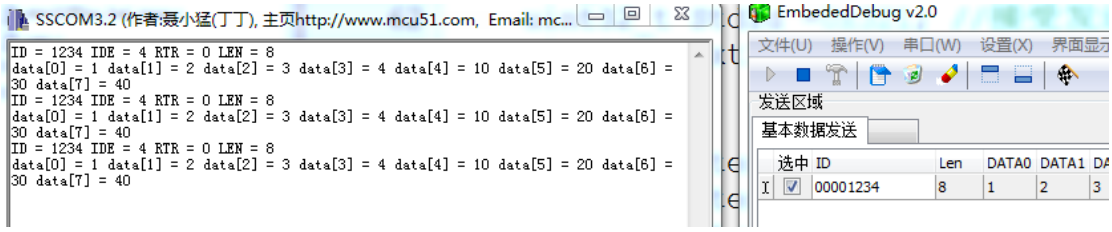
设置我要接受的 ID

前面说了要用这个函数先打开 CAN 接受中断

中断函数和 CAN 总线接受实现章节将的一样。

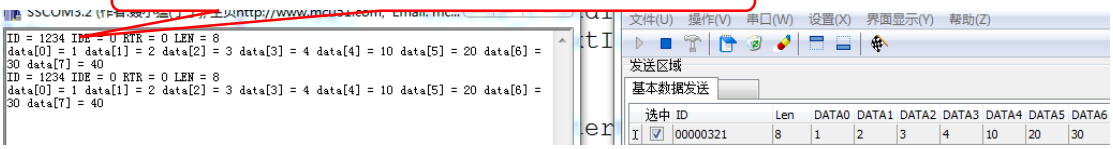
```
void CAN_filter_fig()
{
    CAN_FilterConfTypeDef filter;
    uint32_t StdId =0x321; //接受发送的标准ID号
    uint32_t ExtId = 0x1234;//接受发送的扩展ID号
```

我写的扩展接受 ID 是 1234



这里接受的也是 1234，没有问题

这里是 1234 是因为我中断里面没有写读取标识符 ID 的函数



表示符 ID 接受的是 321 也是没有问题的。

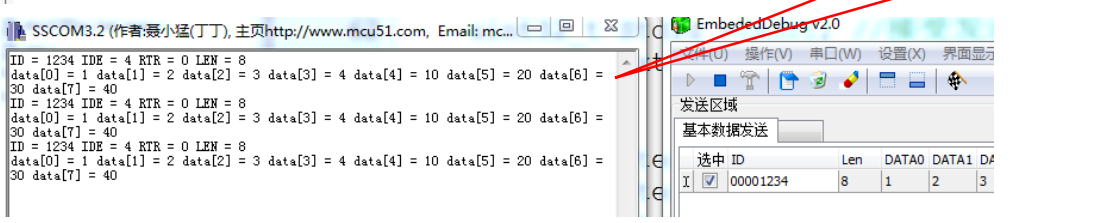
但是现在我不想我的 STM32 芯片除了接受扩展 ID 还要接受标识符 ID。我只想接受扩展 ID

```
void CAN_filter_fig()
{
    CAN_FilterConfTypeDef filter;
    //uint32_t StdId =0x321; //接受发送的标准ID号
    uint32_t ExtId = 0x1234;//接受发送的扩展ID号


    filter.FilterNumber = 0;
    filter.FilterMode = CAN_FILTERMODE_IDLIST;
    filter.FilterScale = CAN_FILTERSCALE_32BIT;

    //filter.FilterIdHigh = StdId<<5;
    //filter.FilterIdLow = 0| CAN_ID_STD;


    filter.FilterMaskIdHigh = ((ExtId<<3)>>16)&0xffff;
    filter.FilterMaskIdLow = ((ExtId<<3)&0xffff)|CAN_ID_EXT;
    filter.FilterFIFOAssignment = CAN_FILTER_FIFO0;
    filter.FilterActivation = ENABLE;

    HAL_CAN_ConfigFilter(&hcan,&filter);
}
```

我们只需要把标识符 ID 的代码屏蔽了

我们只需要把标识符 ID 的代码屏蔽了

STM32CAN 就只判断扩展 ID



CAN 波特率档位统计，1M，500K，125K

修改 CAM 初始化函数里面的内容 MX_CAN_Init

500k 波特率

```
void MX_CAN_Init(void)
{

  hcan.Instance = CAN;

  hcan.Init.Prescaler = 6;//500K
  hcan.Init.Mode = CAN_MODE_NORMAL;
  hcan.Init.SJW = CAN_SJW_1TQ;

  hcan.Init.BS1 = CAN_BS1_8TQ;//500k
  hcan.Init.BS2 = CAN_BS2_7TQ;//500K
  hcan.Init.TTCM = DISABLE;
```

只需修改分频系数

只需修改 BS1,BS2

1M 波特率

```
void MX_CAN_Init(void)
{

  hcan.Instance = CAN;
  hcan.Init.Prescaler = 3;//1M

  hcan.Init.Mode = CAN_MODE_NORMAL;
  hcan.Init.SJW = CAN_SJW_1TQ;

  hcan.Init.BS1 = CAN_BS1_8TQ;//1M
  hcan.Init.BS2 = CAN_BS2_7TQ;//1M
```

125K 波特率

```
void MX_CAN_Init(void)
{

  hcan.Instance = CAN;

  hcan.Init.Prescaler = 24;//125K
  hcan.Init.Mode = CAN_MODE_NORMAL;
  hcan.Init.SJW = CAN_SJW_1TQ;

  hcan.Init.BS1 = CAN_BS1_8TQ;//125k
  hcan.Init.BS2 = CAN_BS2_7TQ;//125k
```

TIM1 定时器使用

Table 7. Timer feature comparison

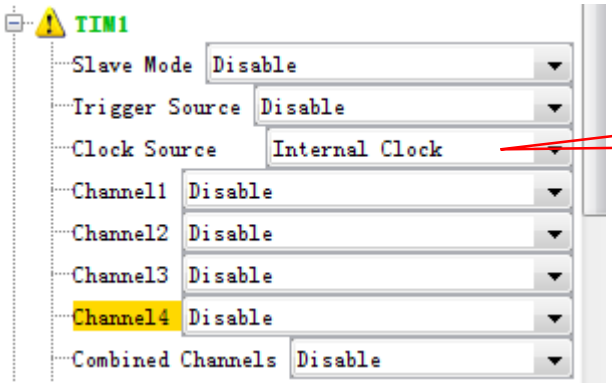
Timer type	Timer	Counter resolution	Counter type	Prescaler factor	DMA request generation	Capture/compare channels
Advanced control	TIM1	16-bit	Up, down, up/down	integer from 1 to 65536	Yes	4

定时器 1 是 16 位可以计数从 0～65535 溢出

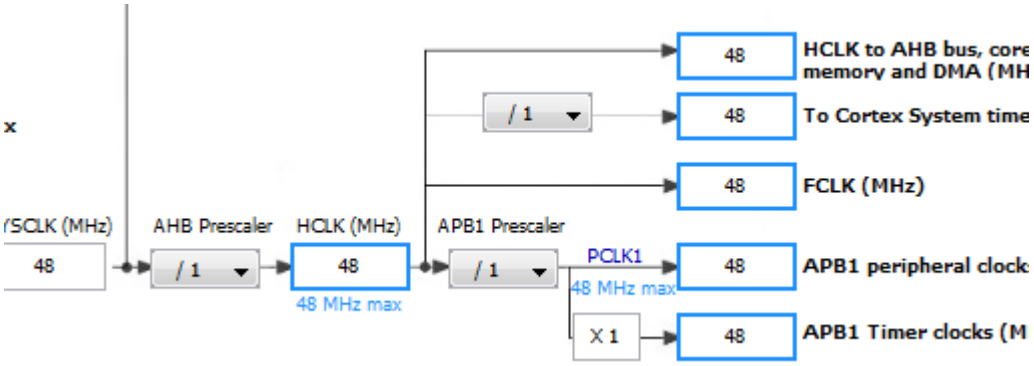
Up 向上计数从 0 加到 65535 溢出
Down 向下，从 65535 减到 0 溢出

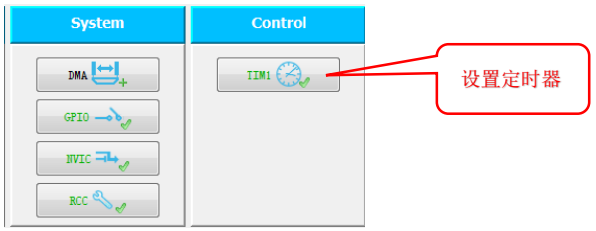
计数时间可以用主时钟分频，分频系数为 1～65536

定时器产生 4 路 PWM

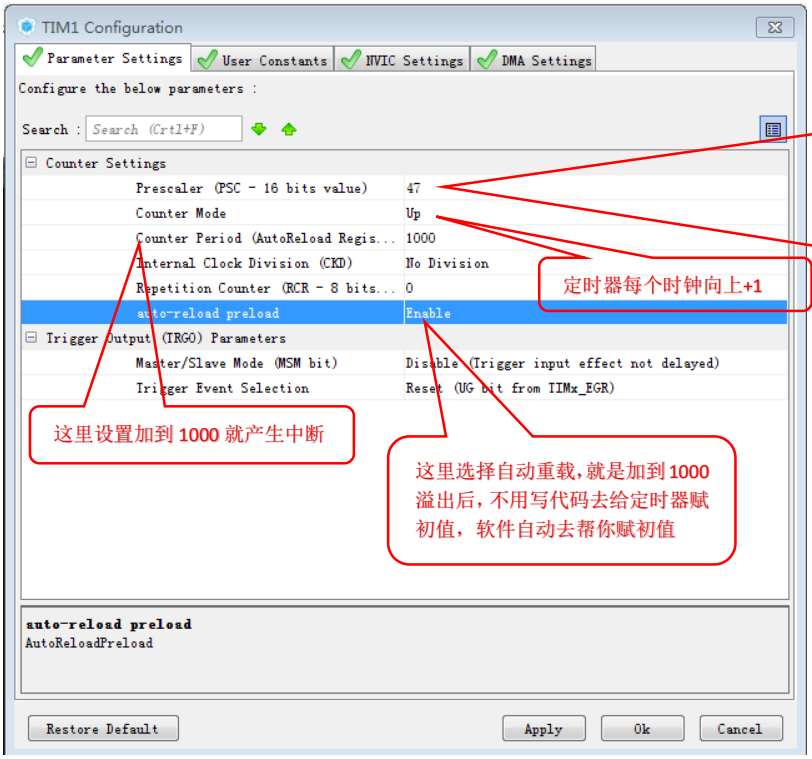


选择内部时钟 48M





设置定时器

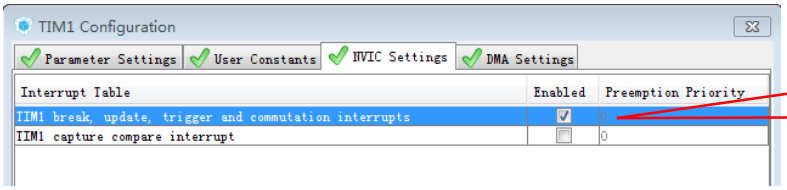


我们要设置 tim1 定时器 1ms(毫秒)产生一次中断
因为用的是内部时钟 48M，那么就是 20ns 产生一次中断，太快了，所以要分频。
48000000/48=1M，那么就是 1us 产生一次中断
这里本应该写 48 分频的，因为软件内部要自动+1，所以写 47，最后软件自动分频是 48

定时器每个时钟向上+1

这里设置加到 1000 就产生中断

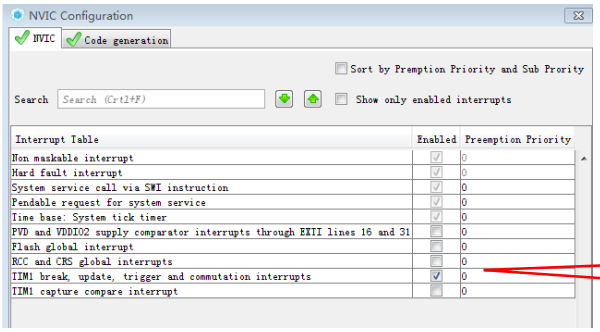
这里选择自动重载，就是加到 1000 溢出后，不用写代码去给定时器赋初值，软件自动去帮你赋初值



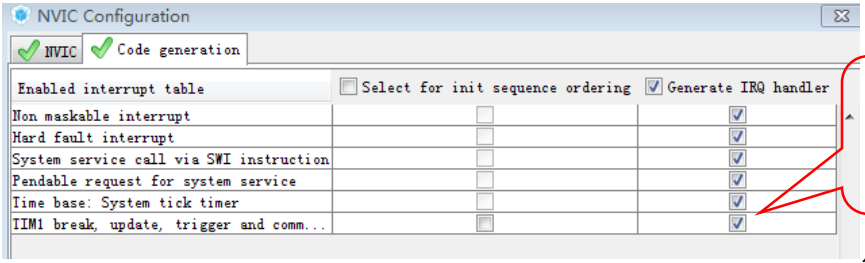
定时器溢出产生中断，进入中断子程序

点击确定

点击中断管理



确认中断使能



确认产生 tim1 中断服务函数，然后配置生成代码的选项

点击 OK 生成代码

```c
int main(void)
{
  HAL_Init();

  SystemClock_Config();

  MX_GPIO_Init();
  MX_TIM1_Init();
```

tim.c

```c
/* TIM1 init function */
void MX_TIM1_Init(void)
{
  TIM_ClockConfigTypeDef sClockSourceConfig;
  TIM_MasterConfigTypeDef sMasterConfig;

  htim1.Instance = TIM1;
  htim1.Init.Prescaler = 47;
  htim1.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim1.Init.Period = 1000;
  htim1.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  htim1.Init.RepetitionCounter = 0;
  htim1.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_ENABLE;
  if (HAL_TIM_Base_Init(&htim1) != HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

  sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
  if (HAL_TIM_ConfigClockSource(&htim1, &sClockSourceConfig) != HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

  sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
```

使用 tim1 定时器

+1 后分频 48

向上计数

计数加到 1000 溢出

溢出后自动重装初值

stm32f0xx_it.c
gpio.c
stm32f0xx_hal_msp.c

```c
*/
void TIM1_BRK_UP_TRG_COM_IRQHandler(void)
{

  HAL_TIM_IRQHandler(&htim1);

}
```

TIM1 溢出后产生中断的入口函数

清除中断标志位

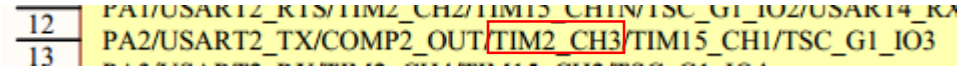
```c
SystemClock_Config();

MX_GPIO_Init();
MX_TIM1_Init();
HAL_TIM_Base_Start_IT(&htim1);
```

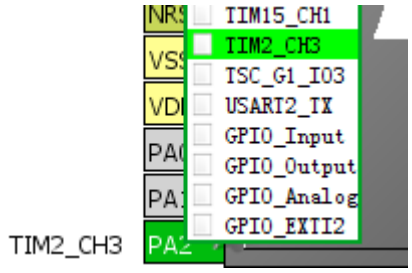
在主函数中一定要打开这个，tim1 才开始工作

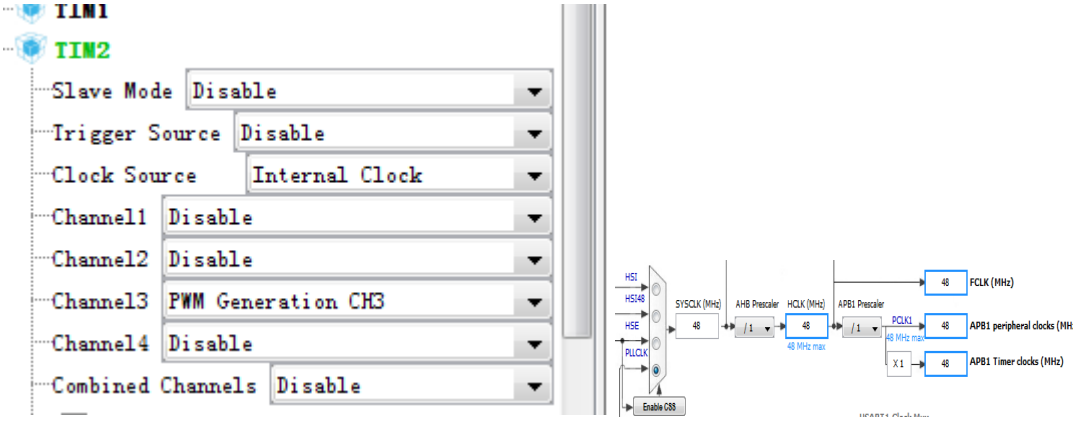
Tim1 使用完毕，申请变量在中断函数里面自动+1，加到 1000 就是 1 秒

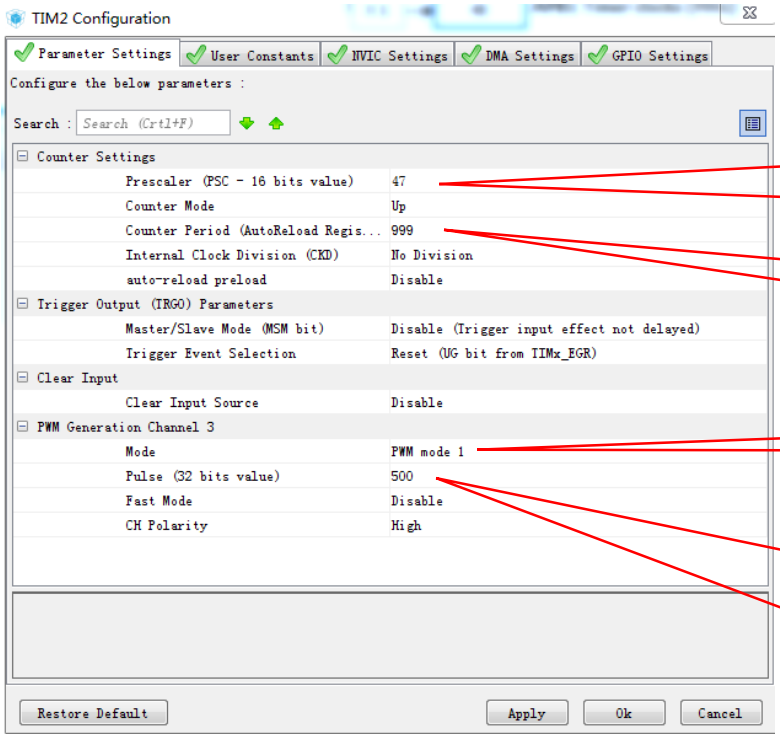
用定时器 2 做 PWM 输出

我们用 TIM2 定时器，CH3 通道做 PWM







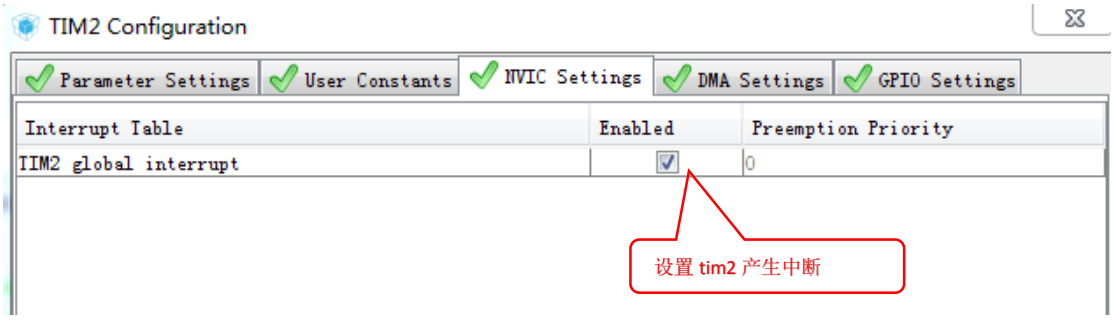


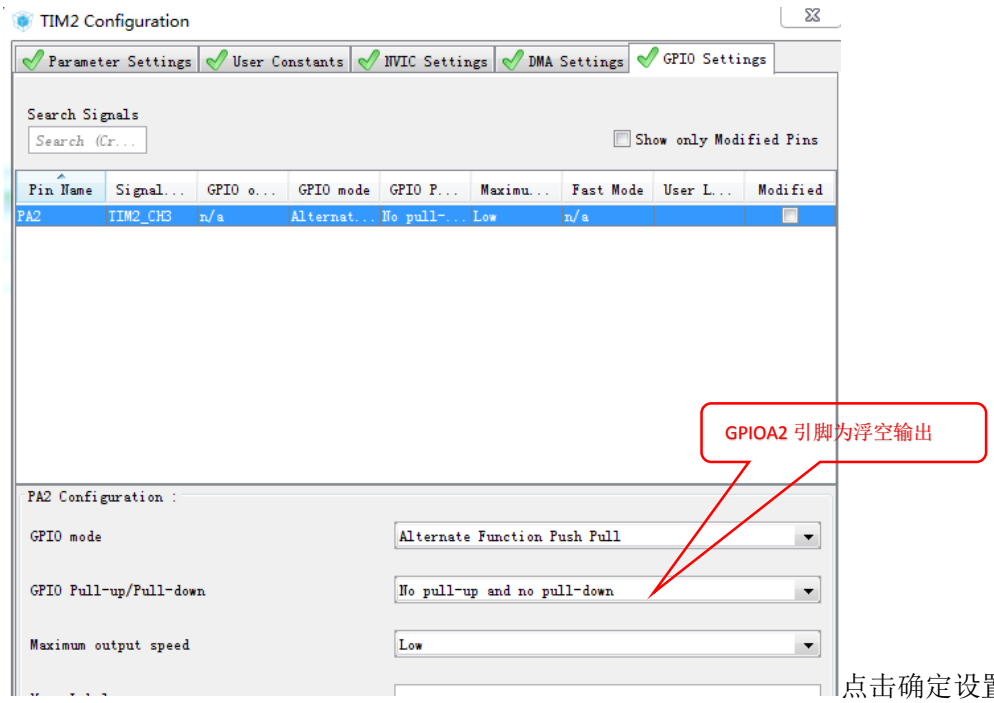
48M 分频为 1M，为什么不是 48 而是 47，请看 tim1 定时器使用

这里是 999+1，所以 PWM 周期是 1ms

设置 PWM 模式 1

这里是 PWM 占空比为 500，意思就是你上面设置的 pwm 周期是 1ms，这里 500 就是 0.5 占空比，这个占空比是可以随时调节的





点击确定设置完毕。

然后配置代码生成选项，生成代码。

```
int main(void)
{


  /* MCU Configuration--------------------------
  /* Reset of all peripherals, Initializes the
  HAL_Init();
  /* Configure the system clock */
  SystemClock_Config();
  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_TIM2_Init();
  HAL_TIM_PWM_Start(&htim2,TIM_CHANNEL_3);
  /* USER CODE BEGIN WHILE */
  while (1)
  {

void MX_GPIO_Init(void)
{

  /* GPIO Ports Clock Enable */
  __HAL_RCC_GPIOF_CLK_ENABLE();
  __HAL_RCC_GPIOA_CLK_ENABLE();

}
```

这是开启 TIM2,CH3 通道 PWM 功能，这个函数不能少

这里只设置 GPIO 的时钟，GPIO 的 PWM 引脚功能交由其他函数设置

```
47 TIM_HandleTypeDef htim2;
48
49 /* TIM2 init function */
50 void MX_TIM2_Init(void)
51 {
52   TIM_ClockConfigTypeDef sClockSourceConfig;
53   TIM_MasterConfigTypeDef sMasterConfig;
54   TIM_OC_InitTypeDef sConfigOC;
55
56   htim2.Instance = TIM2;
57   htim2.Init.Prescaler = 47;
58   htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
59   htim2.Init.Period = 999;
60   htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
61   htim2.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
62   if (HAL_TIM_Base_Init(&htim2) != HAL_OK)
63   {
```

PWM 使用的定时器

设置分频系数

向上计数

设置 PWM 频率为 1K

```
  sConfigOC.OCMode = TIM_OCMODE_PWM1;
  sConfigOC.Pulse = 500;
  sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
  sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
  if (HAL_TIM_PWM_ConfigChannel(&htim2, &sConfigOC, TIM_CHANNEL_3) != HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

  HAL_TIM_MspPostInit(&htim2);
```

设置占空比

设置 GPIO 管脚复用功能

然后你的单片机就输出 0.5 占空比的 PWM 波形了，浮空输出也是输出的 0～3.3V

如果你要随时调节 PWM 的占空比

```
void pwm_dutyratio(uint32_t pulse){
    TIM_OC_InitTypeDef sConfigOC;

    sConfigOC.OCMode = TIM_OCMODE_PWM1;
    sConfigOC.Pulse = pulse;
    sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
    sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
    if (HAL_TIM_PWM_ConfigChannel(&htim2, &sConfigOC, TIM_CHANNEL_3) != HAL_OK)
    {
        _Error_Handler(__FILE__, __LINE__);
    }
    HAL_TIM_PWM_Start(&htim2,TIM_CHANNEL_3);//重新启动TIM2定时器，CH3通道PWM
}
```

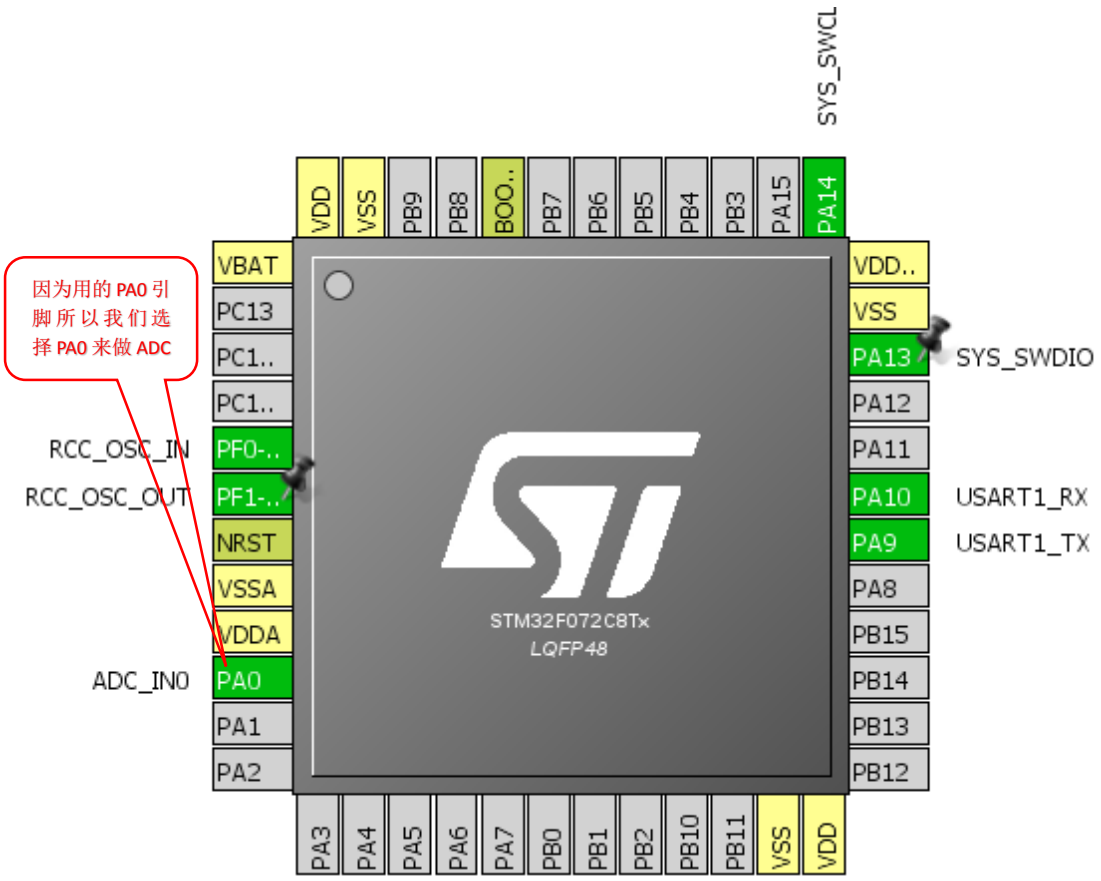
```
for(i=0;i<1000;i++)
{
    pwm_dutyratio(i);
    HAL_Delay(1);
}
```

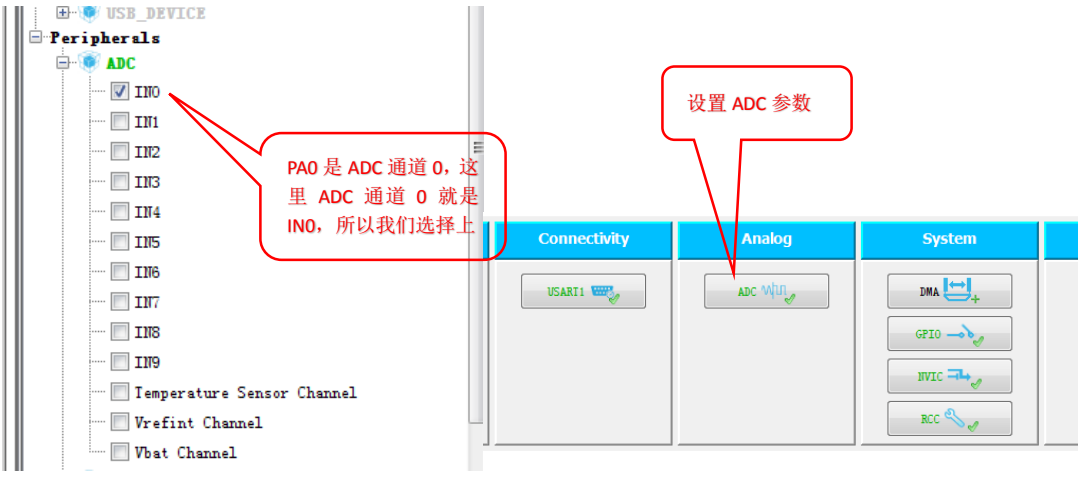
这样就可以了

ADC 单通道轮询问电压采集

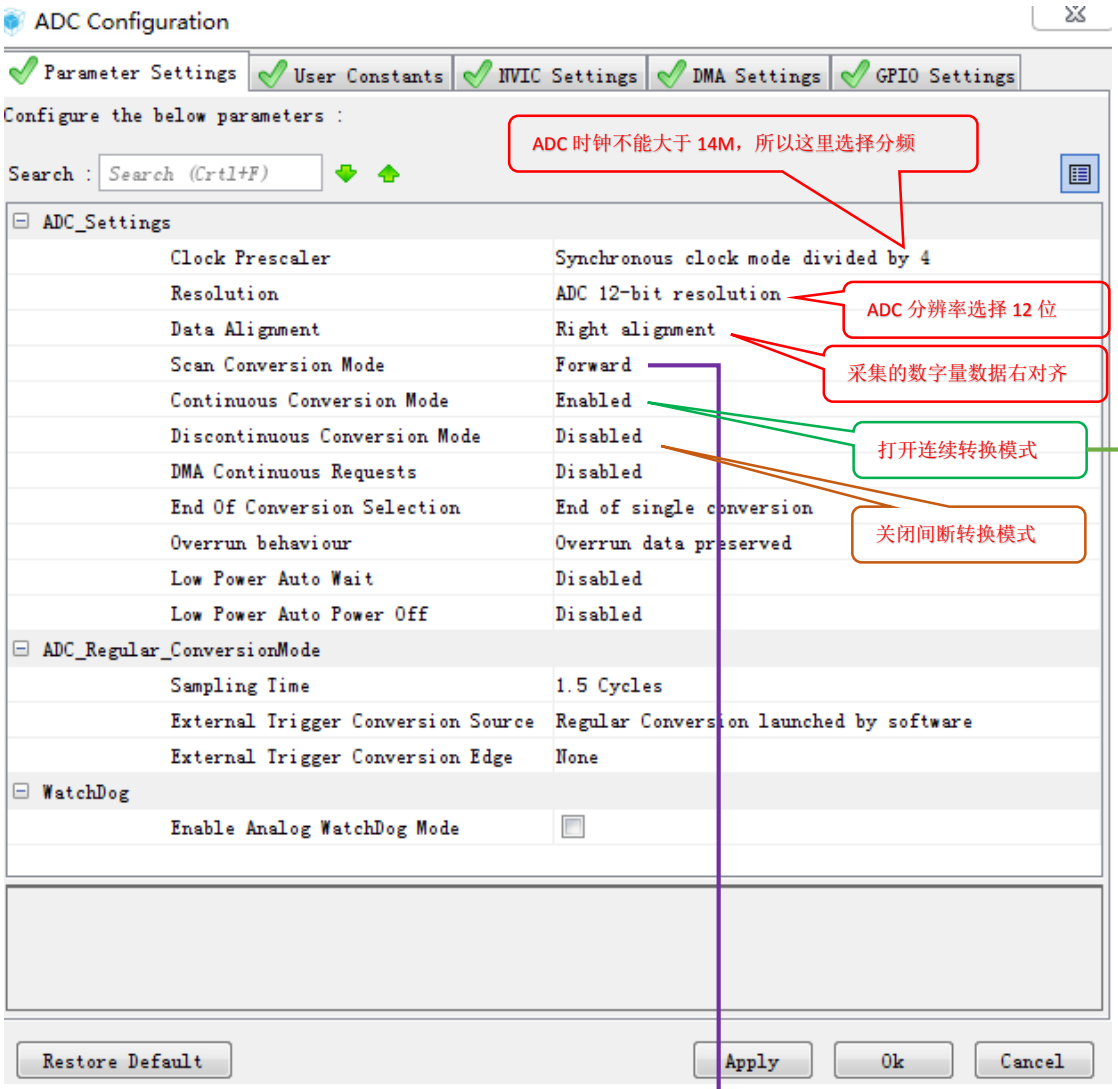
PA0/WKUP 23 24 PA0/USART2_CTS/TIM2_CH1_ETR/TSC_G1_IO1/USART4_TX/RTC_ TAMP2/WKUP1/COMP1_OUT/ADC_IN0/COMP1_INM6

我们采用 STM32F072 的 PA0 引脚来采集电源电压，记得在该引脚并联一个 1uf 的电容，这样采集的电压数据更稳定，跳变在个位数。





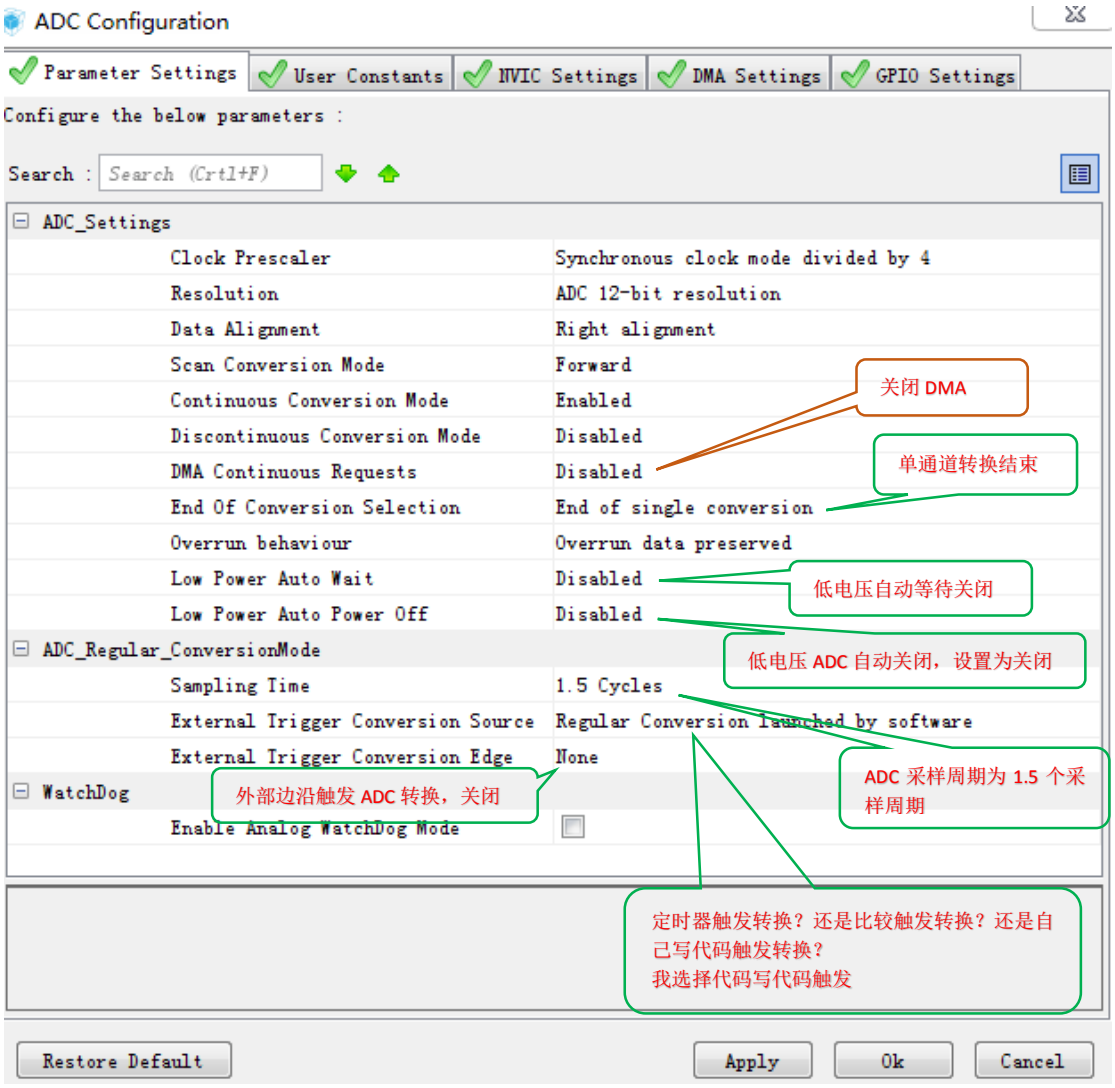
ADC 时钟最大 14M

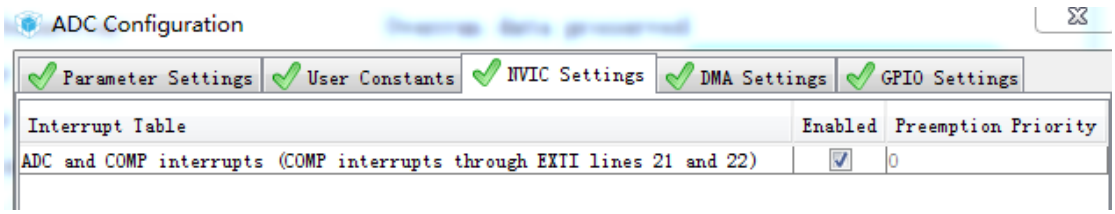


Forward 扫描通道从第一个通道到最后一个通道
Backward 扫描通道从最后一个通道到第一个通道

要求进行一次 ADC 转换：配置为单次模式使能，扫描模式失能。这样 ADC 的这个通道，转换一次后，就停止转换。

要求进行连续 ADC 转换：配置为连续模式使能，扫描模式失能。这样 ADC 的这个通道，转换一次后，接着进行下一次转换，不断连续





打开 ADC 中断，其实我软件触发不知道打开 ADC 中断有什么用？

```c
int main(void)
{
    uint32_t data = 0;

  /* Reset of all peripherals, Initializes the F
  HAL_Init();
  /* Configure the system clock */
  SystemClock_Config();
  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_USART1_UART_Init();
  MX_ADC_Init();
```

ADC 初始代码

```c
ADC_HandleTypeDef hadc;//定义个ADC结构体

/* ADC init function */
void MX_ADC_Init(void)
{
  ADC_ChannelConfTypeDef sConfig;

    /**Configure the global features of the ADC (Clock, Resolution, Data Alignment and
    */
  hadc.Instance = ADC1;//选择ADC1
  hadc.Init.ClockPrescaler = ADC_CLOCK_SYNC_PCLK_DIV4;//HSI时钟分频为14M给ADC
  hadc.Init.Resolution = ADC_RESOLUTION_12B;//ADC采样分辨率为12位
  hadc.Init.DataAlign = ADC_DATAALIGN_RIGHT;//采集数据右对齐
  hadc.Init.ScanConvMode = ADC_SCAN_DIRECTION_FORWARD;//扫描通道从第一个到最后一个通道
  hadc.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
  hadc.Init.LowPowerAutoWait = DISABLE;//关闭低压自动等待
  hadc.Init.LowPowerAutoPowerOff = DISABLE;//关闭ADC低电压自动关闭功能
  hadc.Init.ContinuousConvMode = ENABLE;//打开连续转换模式
  hadc.Init.DiscontinuousConvMode = DISABLE;//间断转换模式关闭
  hadc.Init.ExternalTrigConv = ADC_SOFTWARE_START;//ADC由我写的代码启动
  hadc.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;//编译触发启动ADC关闭
  hadc.Init.DMAContinuousRequests = DISABLE;//DMA功能关闭
  hadc.Init.Overrun = ADC_OVR_DATA_PRESERVED;
  if (HAL_ADC_Init(&hadc) != HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

    /**Configure for the selected ADC regular channel to
    */
  sConfig.Channel = ADC_CHANNEL_0;
  sConfig.Rank = ADC_RANK_CHANNEL_NUMBER;
  sConfig.SamplingTime = ADC_SAMPLETIME_1CYCLE_5;
  if (HAL_ADC_ConfigChannel(&hadc, &sConfig) != HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

}
```

ADC 通道 0，也就是 PA0

ADC 采样周期为 1.5 个采样周期

下面是主函数死循环启动 ADC

```
while (1)
{
    HAL_ADC_Start(&hadc);
    HAL_ADC_PollForConversion(&hadc,10);
    data = HAL_ADC_GetValue(&hadc);
    printf("data = %d\r\n",data);
    HAL_ADC_Stop(&hadc);
    HAL_Delay(100);

}
```

启动 ADC 转换

不知道 ADC 转换时间是多久，我这里等待 10 毫秒

获取 ADC 转换后的数据

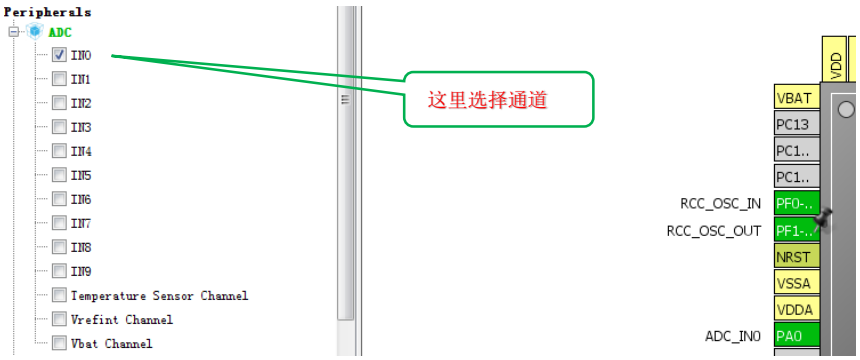
关闭 ADC 转换

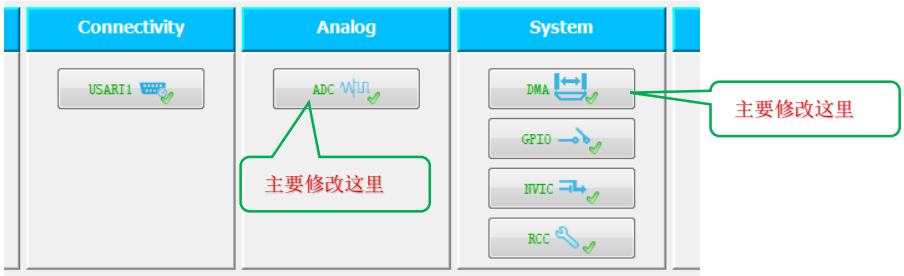
```
 HAL_ADC_Start(&hadc);
while (1)
{

    HAL_ADC_PollForConversion(&hadc,10);
    data = HAL_ADC_GetValue(&hadc);
    printf("data = %d\r\n",data);

    HAL_Delay(100);

}
HAL_ADC_Stop(&hadc);
```

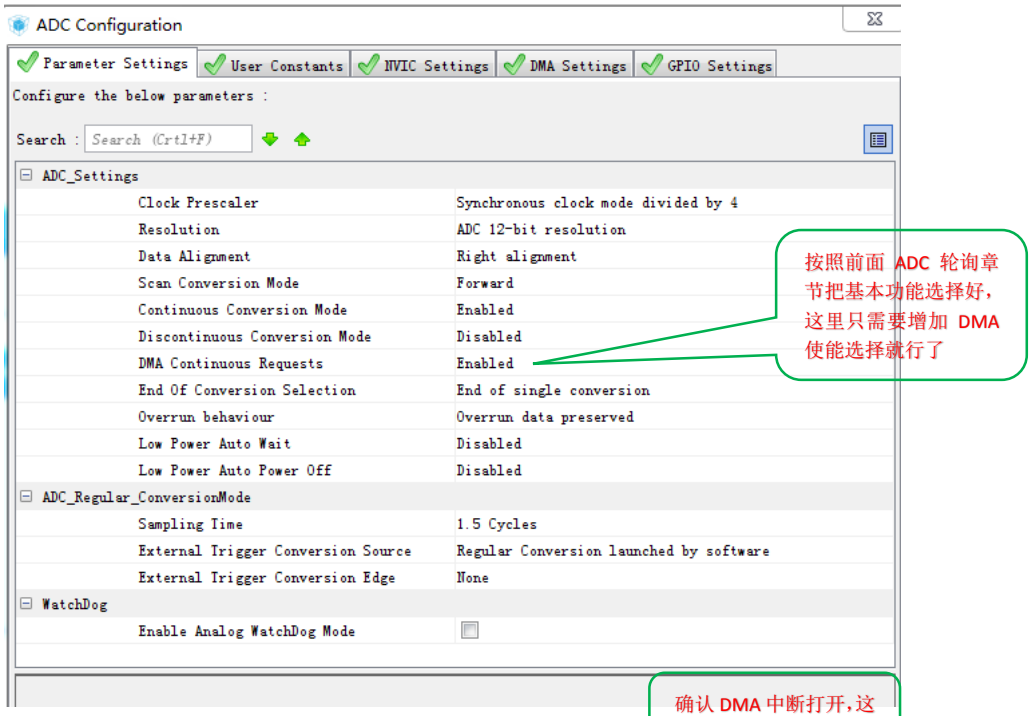
因为是连续转换，所以启动 ADC 之后可以不关闭

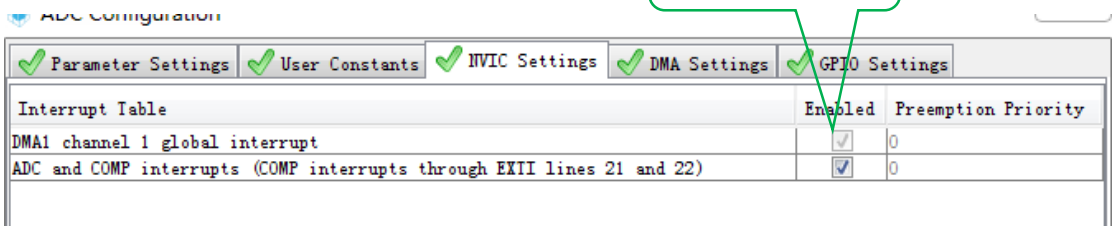
每次在这里等待 AD 采样结束就是

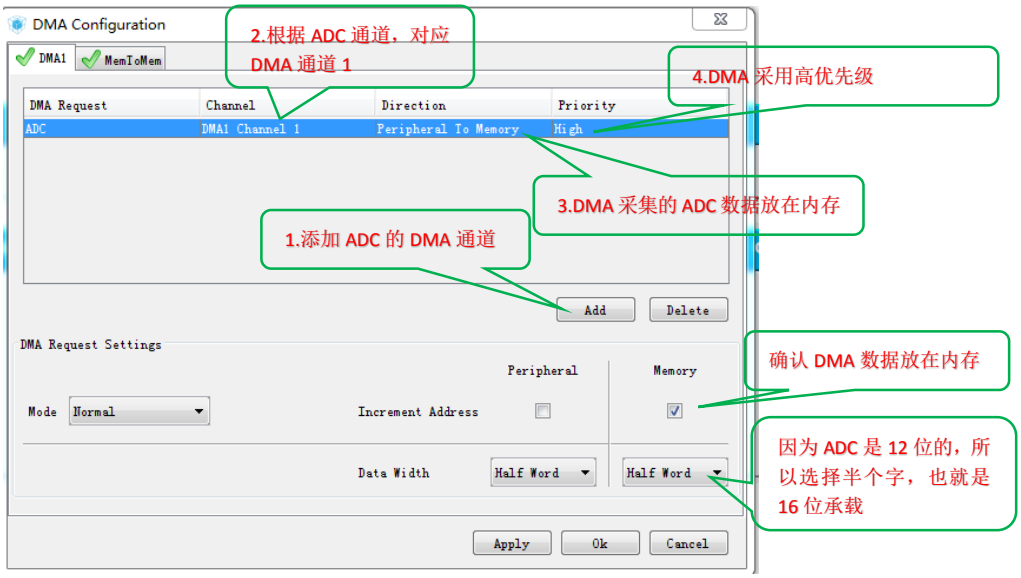
ADC 单通道 DMA 循环采集











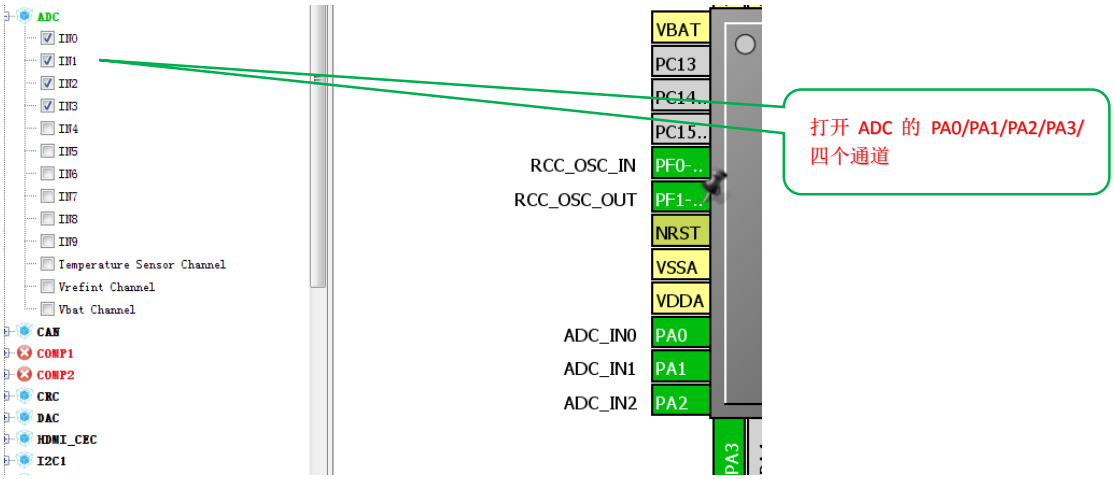
设置选项生成代码

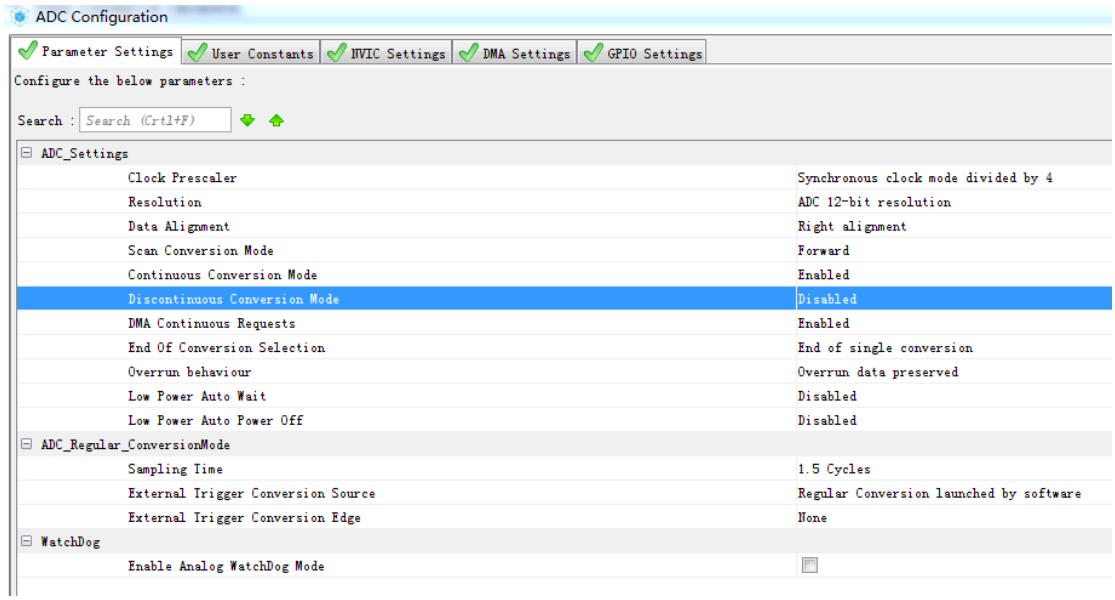
```c
int main(void)
{
  uint32_t ADCbuff[100];
  uint64_t adc1=0;
  uint32_t i=0;
  HAL_Init();

  SystemClock_Config();
  MX_GPIO_Init();
  MX_DMA_Init();//初始化DMA
  MX_USART1_UART_Init();
  MX_ADC_Init();//初始化ADC通道1

  while (1)
  {

    if(HAL_ADC_Start_DMA(&hadc,(uint32_t* )&ADCbuff,100)!=HAL_OK)//启动一次DMA
    {
        printf("ADC DMA conversion error\r\n");//启动DMA失败
    }
    else
    {
        printf("ADC DMA conversion success...\r\n");//启动DMA成功
    }
    HAL_ADC_Stop_DMA(&hadc);//这里一定要关闭，否则下次启动DMA会失败

    for(i=0;i<100;i++)
    {
        adc1 += ADCbuff[i];
        //因为每次采集的数据都是32位的，所以100次加起来比32位大，用64位承载

    }
    adc1=adc1/100;//平均值
    printf("adc1 = %lld\r\n",adc1);
    HAL_Delay(500);

  }
```

单通道 DMA 采集这里一定要写采集 100 次

代码使用例程

ADC 多通道轮询采集





配置保持 ADC 单通道轮询的设置，这里加入了 DMA，但是没有用到 DMA

然后生成代码

这里如果不设置 16 位变量接受数据，而是用 32 位，或者 8 位，那么数据就会乱，出现最大最小值

```
int main(void)
{
  __IO uint16_t ADCbuff[3];
    //因为ADC是按照16位格式的数据返回的，所以这里要用16位变量接受
  int i=0;
  /* Reset of all peripherals, Initializes the Flash interface and the Systick.
  HAL_Init();

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

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_DMA_Init();
  MX_USART1_UART_Init();
  MX_ADC_Init();
```

```
while (1)
{
  for(i=0;i<4;i++)
  {
      HAL_ADC_Start(&hadc);
      HAL_ADC_PollForConversion(&hadc,10);//等待10ms采集结束
      ADCbuff[i] = HAL_ADC_GetValue(&hadc);
      printf("ADCbuff %d = %d\r\n",i,ADCbuff[i]);
  }
  HAL_ADC_Stop(&hadc);
  HAL_Delay(500);
}
```

这里轮询记住，每次 start 一次 ADC 通道就会向前增加，所以注意

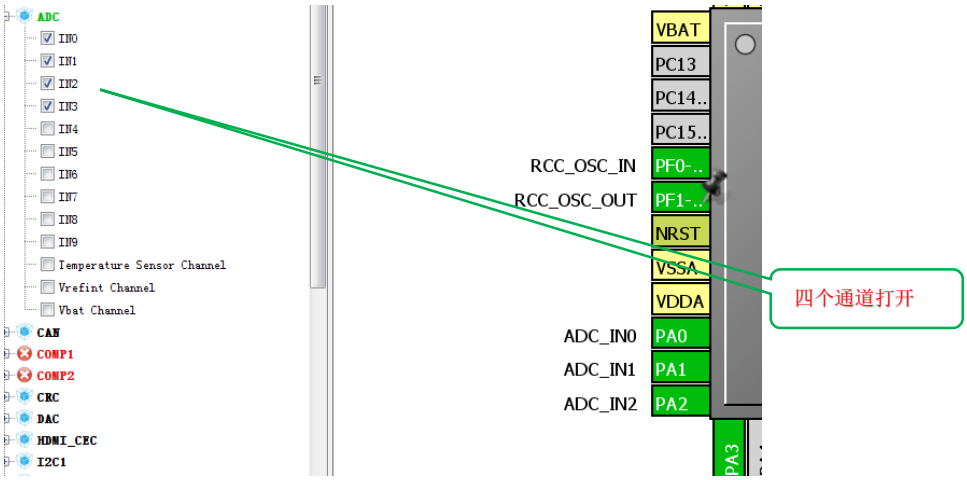
因为 ADC 是 start 一次通道向上增加，所以我这里是要采集四个 ADC 通道的数据，所以不能 start 一次就用 stop 关闭 ADC

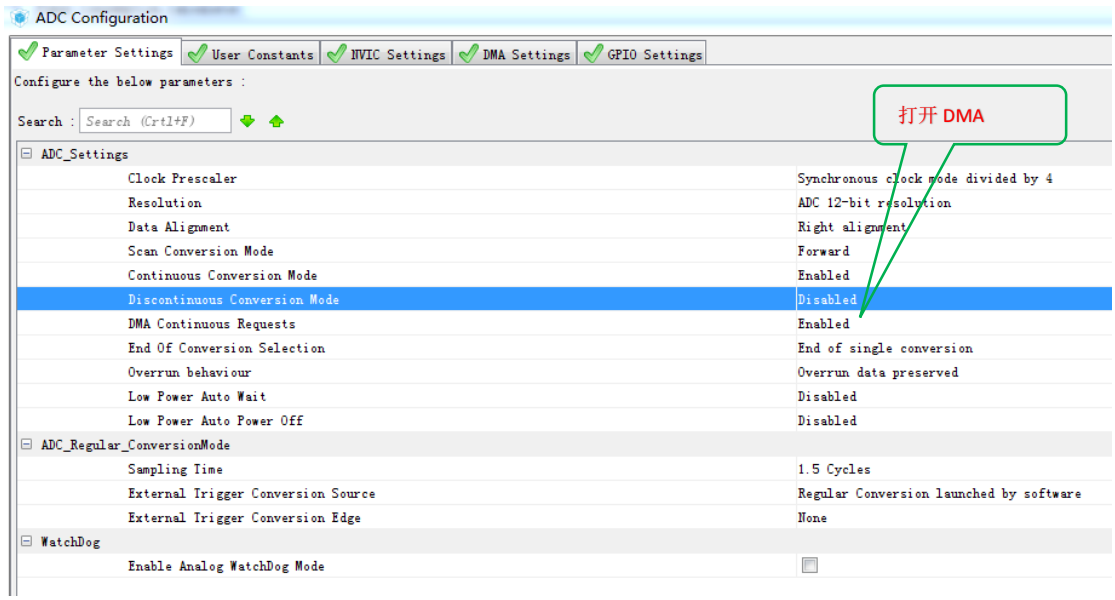
必须等待 4 个 ADC 通道采集完，也就是 start 四次，循环 4 次之后才能执行 stop

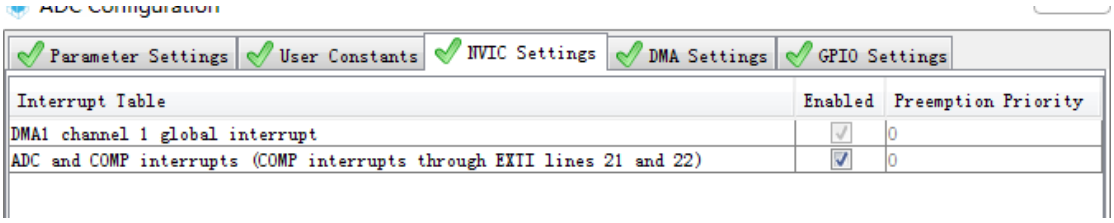
所以数组里面存放的是 4 个通道 ADC 采集的值

所以轮询模式下你要获取 ADC 某个通道的值只有，先四个通道轮询完，然后获取数组对应 ADC 通道的值。

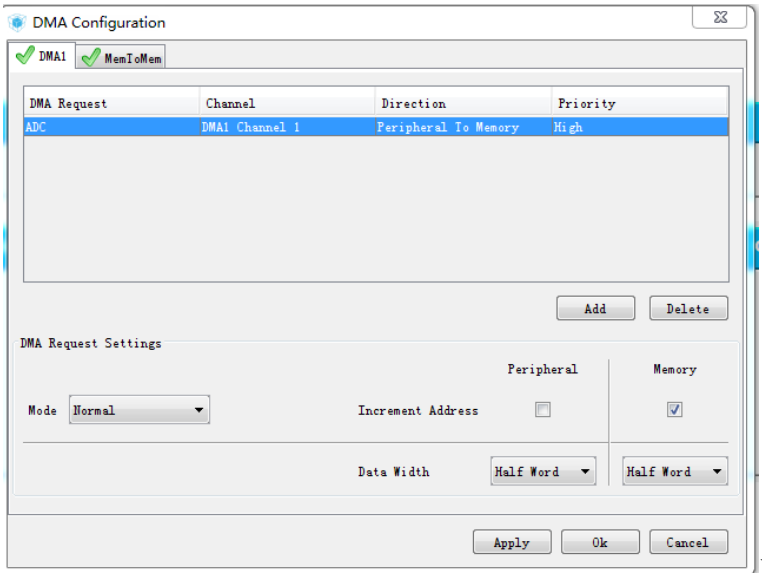
ADC 多通道 DMA 采集







确认勾选上



设置 ADC 对应的 DMA 通道

```c
int main(void)
{
  __IO uint16_t ADCbuff[3];
    //因为ADC是按照16位格式的数据返回的，所以这里要用16位变量接受
  int i=0;
  /* Reset of all peripherals, Initializes the Flash interface and the Systick.
  HAL_Init();

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

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_DMA_Init();
  MX_USART1_UART_Init();
  MX_ADC_Init();
```

```c
while (1)
{
  if(HAL_ADC_Start_DMA(&hadc,(uint32_t*) &ADCbuff,4)!=HAL_OK)
  {
      printf("ADCbuff 1 2 3 4 channel error \r\n");
  }
  else
  {
      printf("ADCbuff 1 2 3 4 channel success \r\n");
  }
  HAL_ADC_Stop_DMA(&hadc);//DMA执行一次获取4个通道的ADC值
  for(i=0;i<4;i++)
  {

      printf("ADCbuff %d = %d\r\n",i,ADCbuff[i]);//将4个通道的ADC值提取出来
  }

  HAL_Delay(500);
```

这就是 DMA 模式获取 ADC 多通道值的方法