信息工程系

教

案

课程名称：STM32 嵌入式开发

总 学 时 ：54

STM32 单片机基础 03——使用 GPIO 点亮一个 LED

教学目的与要求：

目的：理解 GPIO 的基本功能和操作模式，掌握如何通过编程控制 GPIO 引脚输出高低电平来点亮 LED。

要求：能够独立完成 GPIO 端口的初始化配置，编写程序控制 LED 的亮灭。

教学重难点：

重点：GPIO 端口的初始化配置，包括选择正确的 GPIO 组、引脚号、设置输出模式等。

难点：理解 GPIO 端口的工作原理，以及如何通过软件控制 GPIO 引脚的状态。

课时数：3 课时

思政元素：

强调实践与创新精神，通过动手实践加深对理论知识的理解，培养学生的动手能力和创新思维。

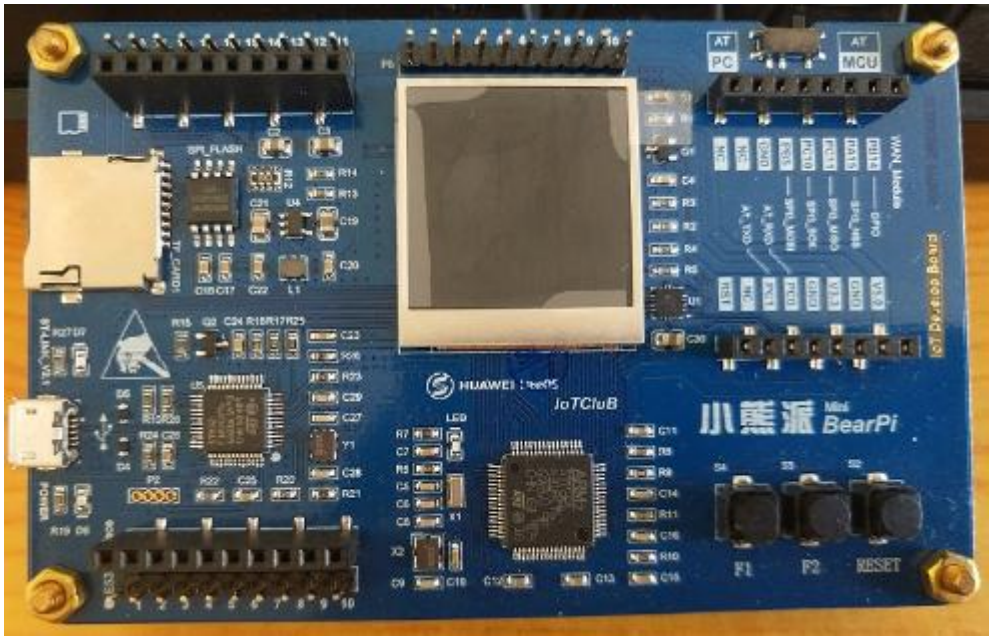
本篇文章主要介绍如何使用 STM32CubeMX 初始化 STM32L431RCT6 的 GPIO，并点亮一个 LED。

1. 准备工作

硬件准备

开发板

首先需要准备一个开发板，这里我准备的是 STM32L4 的开发板（BearPi）：



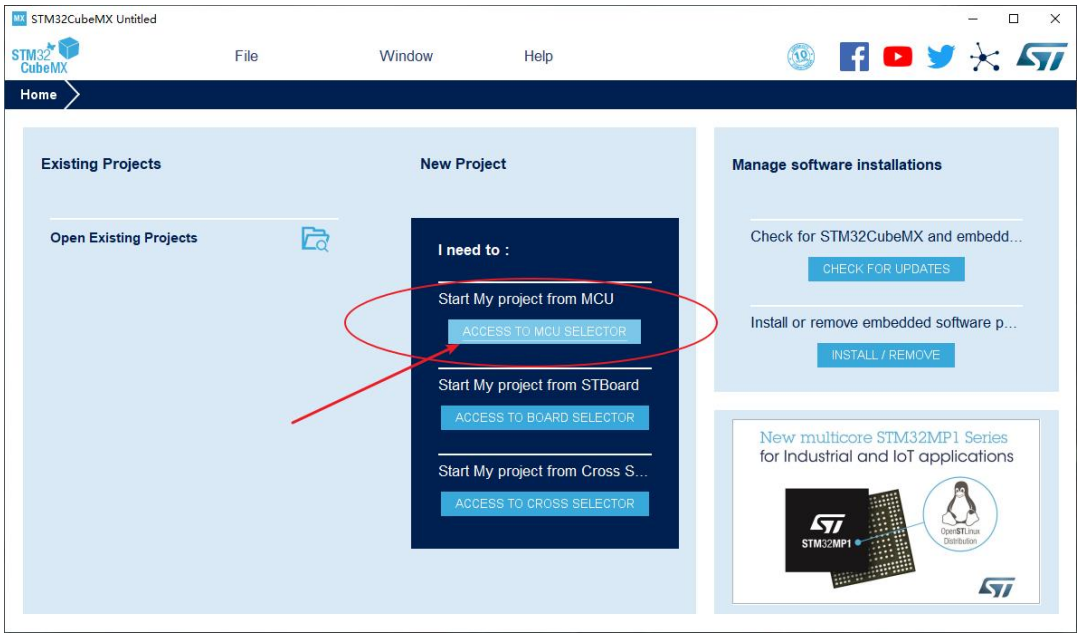
软件准备

- 需要安装好 Keil - MDK 及芯片对应的包，以便编译和下载生成的代码；
  Keil MDK 和串口助手的安装包都可以关注“**小熊派开源社区**”微信公众号，在资料教程一栏中可获取安装包。

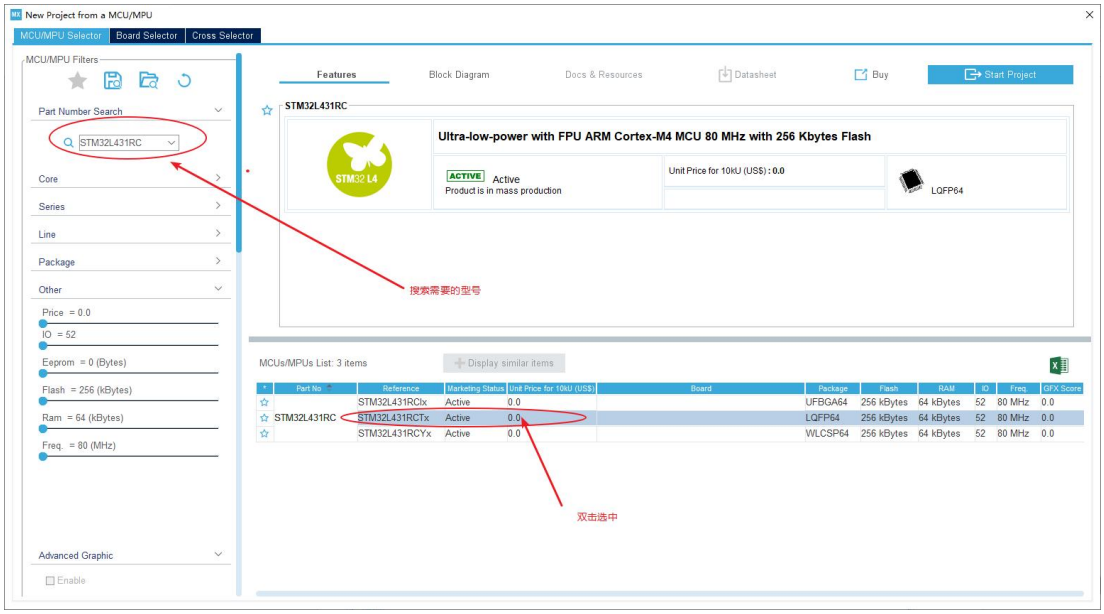
2. 生成 MDK 工程

选择芯片型号

打开 STM32CubeMX，打开 MCU 选择器：



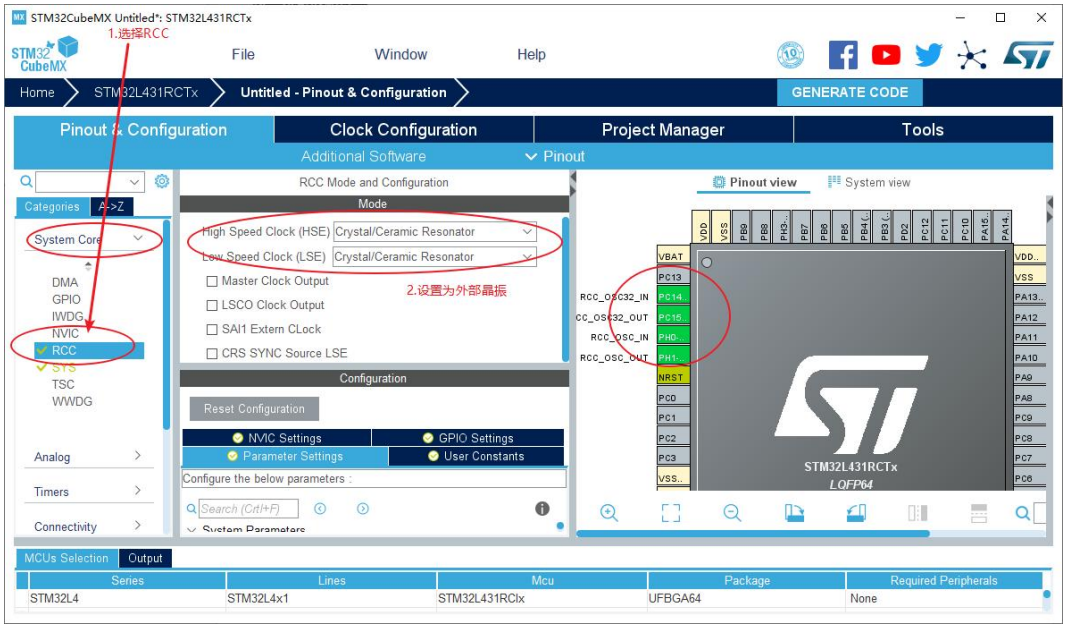
搜索并选中芯片 STM32L431RCT6:



配置时钟源

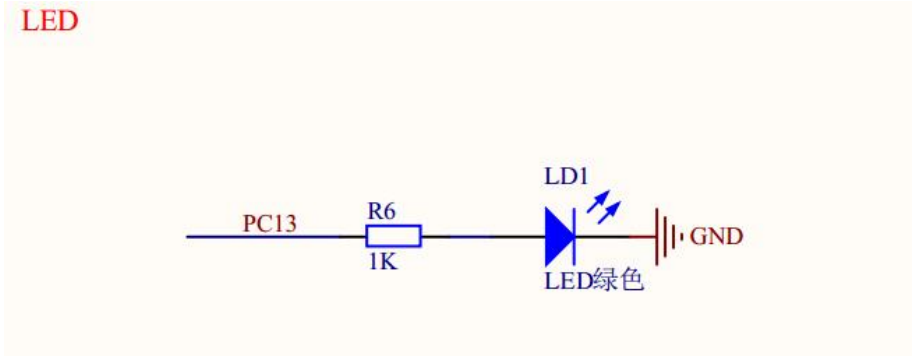
- 如果选择使用外部高速时钟（HSE），则需要在 System Core 中配置 RCC；
- 如果使用默认内部时钟（HSI），这一步可以略过；

这里我都使用外部时钟：

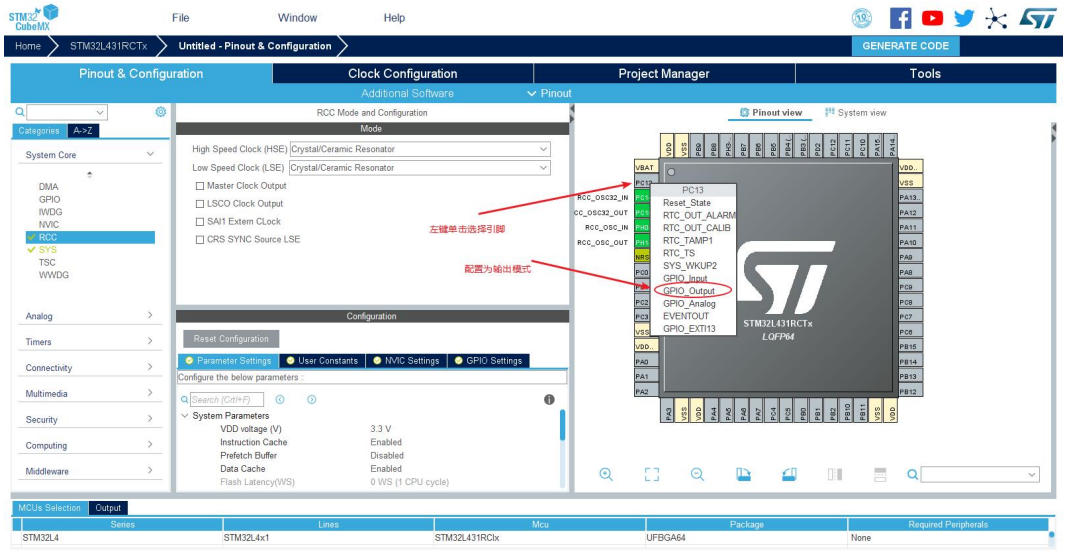


配置 GPIO 引脚

查看小熊派开发板的原理图，如下：

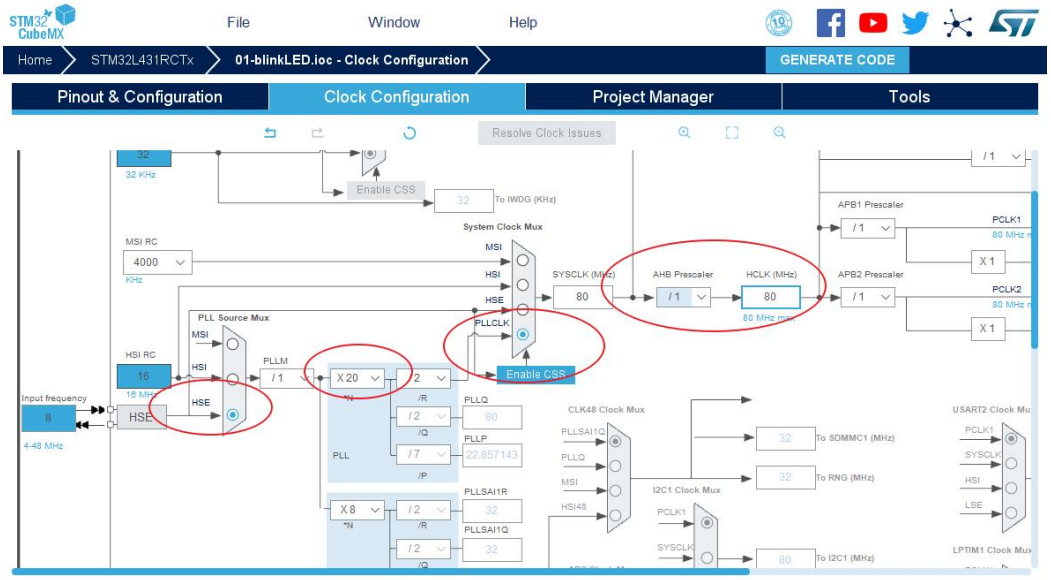


所以接下来我们选择配置 PC13 引脚：

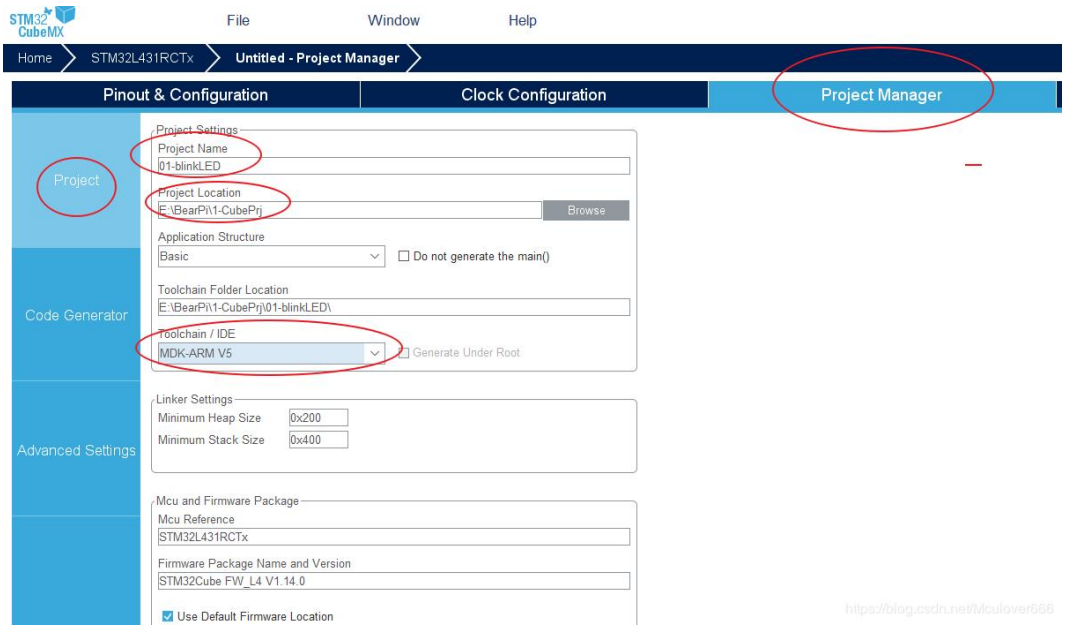


配置时钟树

STM32L4 的最高主频到 80M，所以配置 PLL，最后使 `HCLK = 80Mhz` 即可：

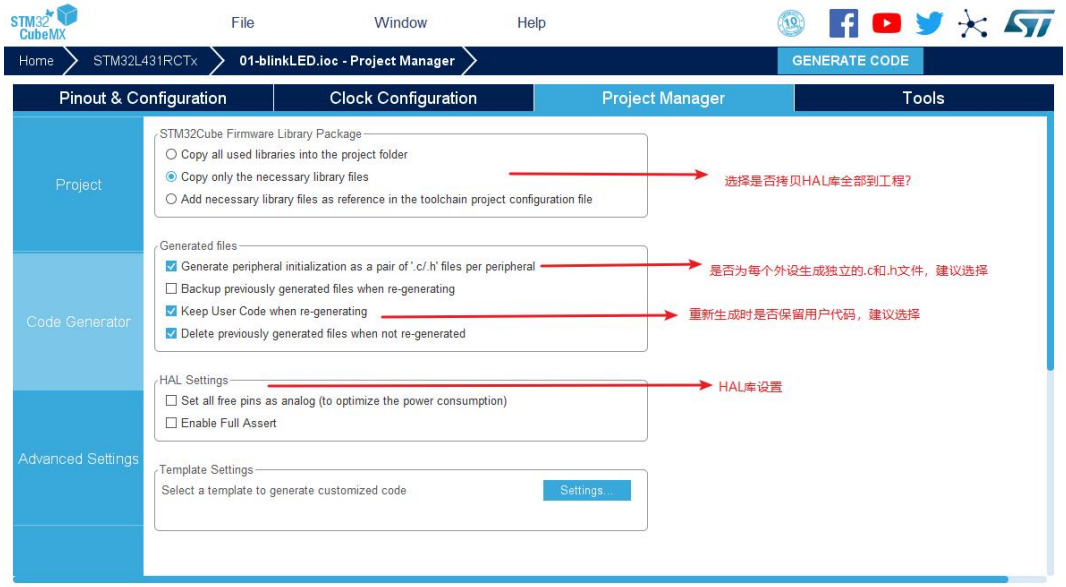


生成工程设置



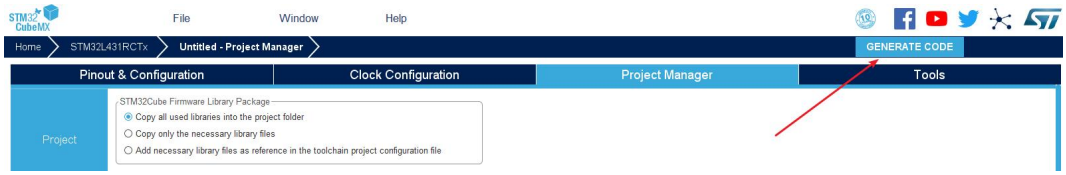
代码生成设置

最后设置生成独立的初始化文件：



生成代码

点击 GENERATE CODE 即可生成 MDK-V5 工程：



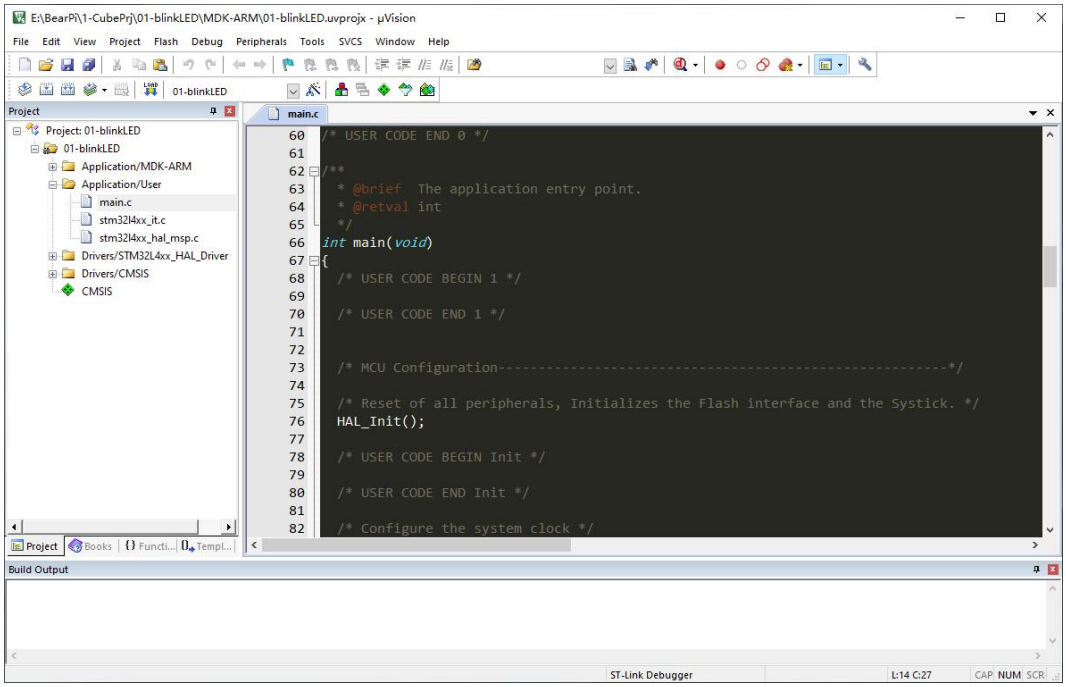
生成成功

3. 在 MDK 中编写、编译、下载用户代码

编写用户代码

STM32CubeMX 生成的代码目录如下：

进入 MDK-ARM 目录，打开工程：



在 main.c 中的 main 函数中编写简单的用户代码：

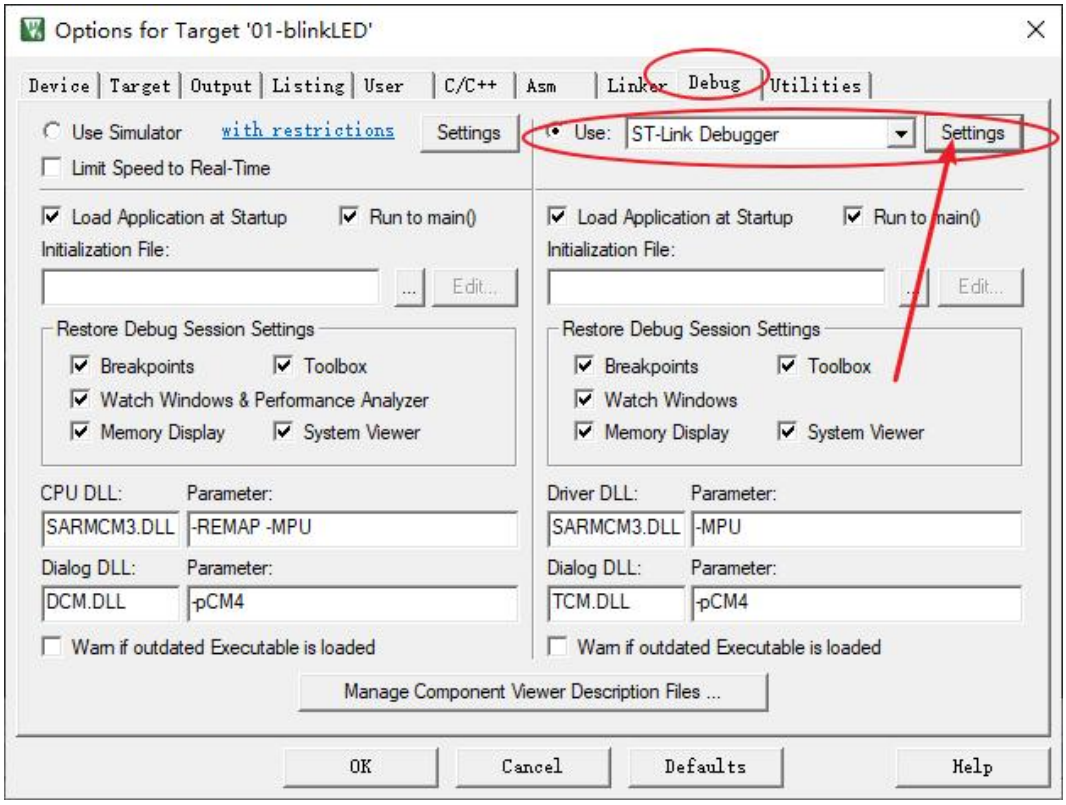
```
  while (1)

  {

    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */

    HAL_Delay(200);

          HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_13);

  }
```

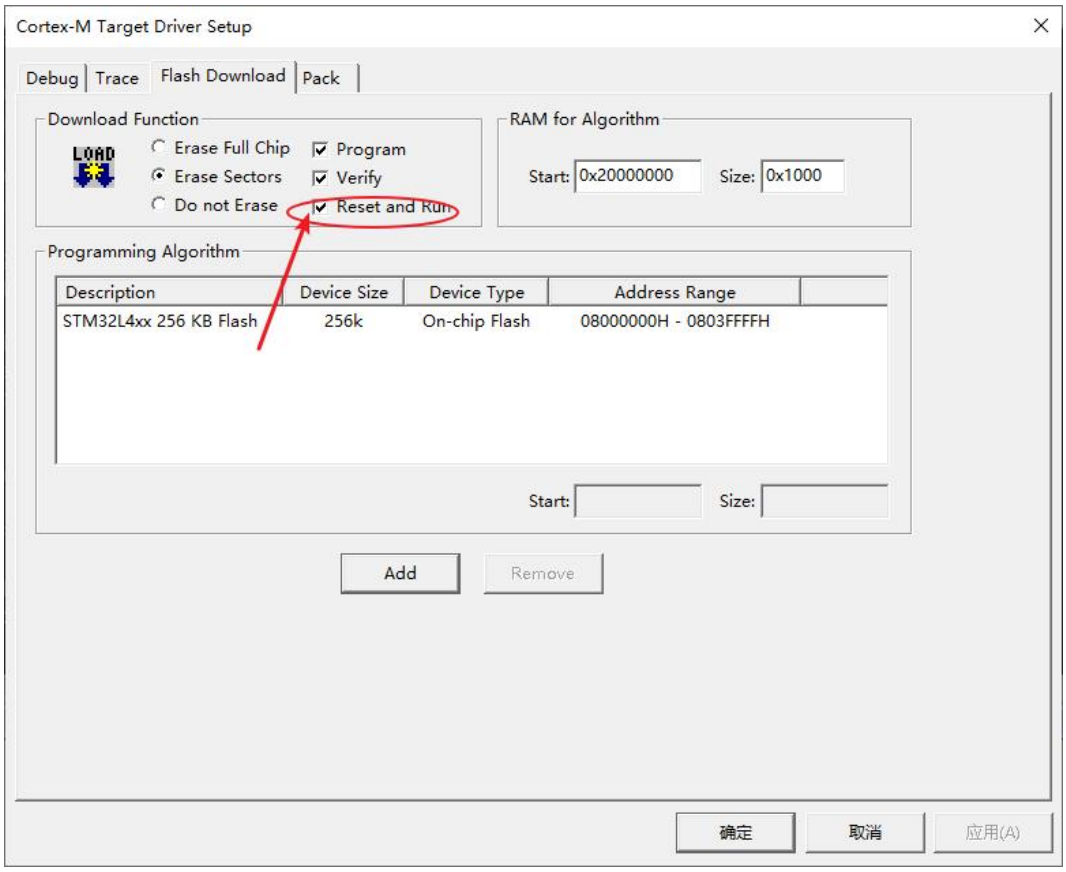
编译代码

编译整个工程：

```
Build Output
compiling stm32l4xx_hal_pwr.c...
compiling stm32l4xx_hal_pwr_ex.c...
compiling stm32l4xx_hal_cortex.c...
compiling stm32l4xx_hal_exti.c...
compiling system_stm32l4xx.c...
linking...
Program Size: Code=3312 RO-data=492 RW-data=16 ZI-data=1024
FromELF: creating hex file...
"01-blinkLED\01-blinkLED.axf" - 0 Error(s), 0 Warning(s).
Build Time Elapsed:  00:00:17
```

设置下载器

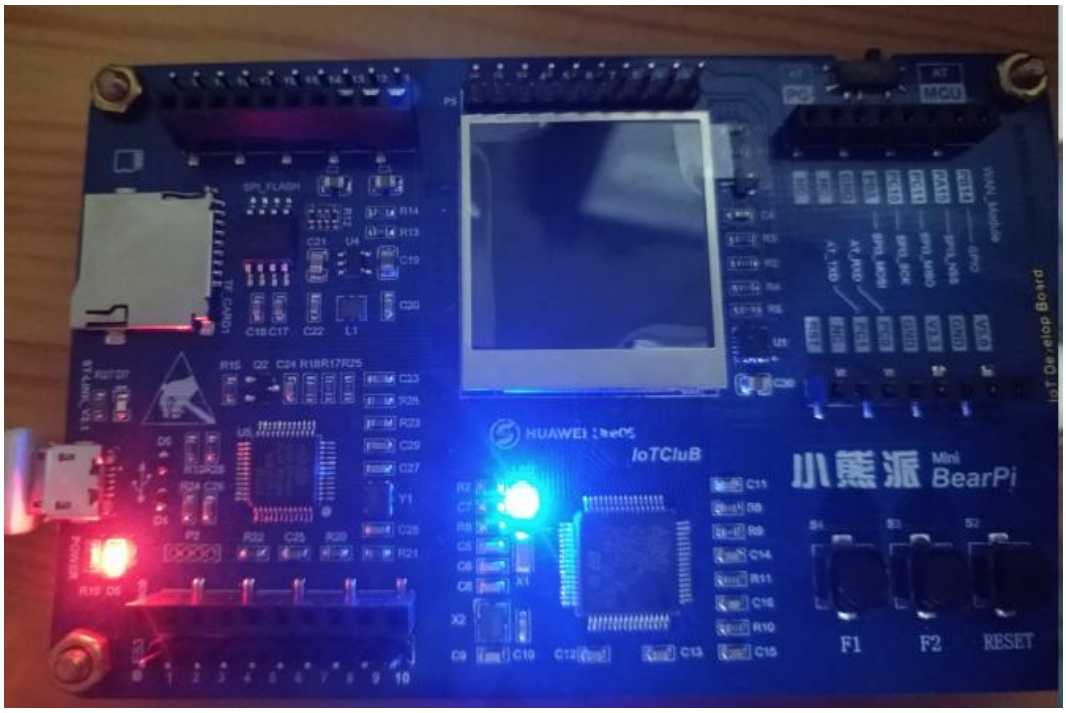




下载运行

```
Build Output
Program Size: Code=3312 RO-data=492 RW-data=16 ZI-data=1024
FromELF: creating hex file...
"01-blinkLED\01-blinkLED.axf" - 0 Error(s), 0 Warning(s).
Build Time Elapsed:  00:00:17
Load "01-blinkLED\\01-blinkLED.axf"
Erase Done.
Programming Done.
Verify OK.
Application running ...
Flash Load finished at 22:04:48
```

实验现象



至此，我们已经学会了**如何使用 STM32CubeMX 快速生成 MDK 的工程**，点亮一个 LED，接下来一节讲述如何使用 STM32CubeMX 初始化 GPIO 进行按键检测。

作业：

设计并编写程序，通过 STM32 单片机的 GPIO 控制一个 LED 灯以固定频率闪烁。

分析 GPIO 寄存器的工作原理，并解释程序中的关键代码段。

STM32 单片机基础 04——使用 GPIO 进行按键检测

教学目的与要求：

目的：掌握 GPIO 端口的输入功能，学习如何通过编程检测按键的状态。

要求：能够配置 GPIO 端口为输入模式，编写程序实现按键的按下检测，并进行去抖动处理。

教学重难点：

重点：GPIO 端口的输入配置，按键按下状态的检测逻辑。

难点：按键去抖动算法的实现，确保按键检测的准确性和稳定性。

课时数：3 课时

思政元素：

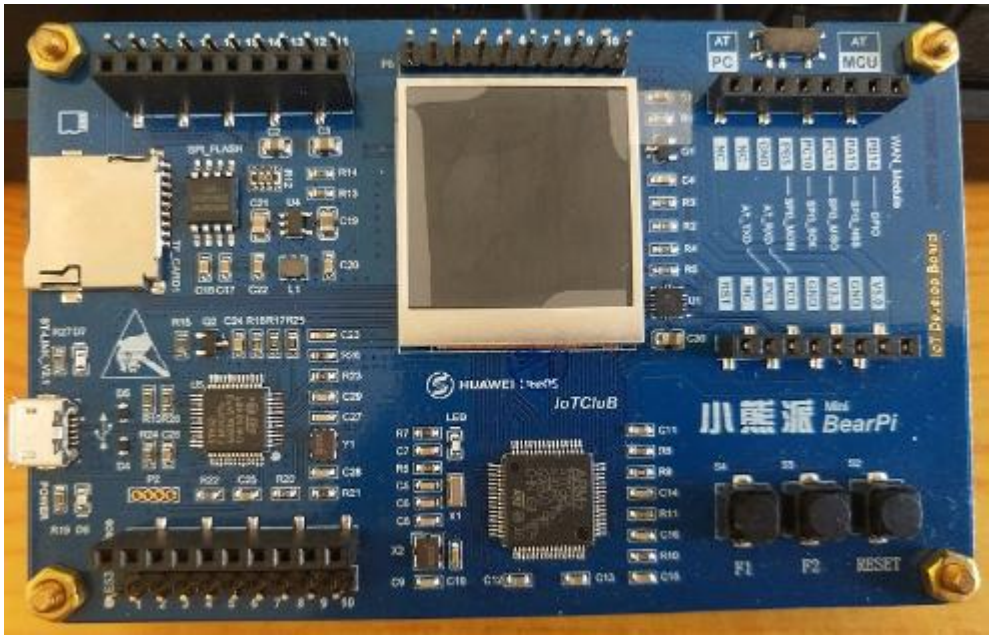
培养学生的耐心和细致的观察能力，通过按键检测的实践，理解软件设计中的细节处理对系统稳定性的影响。

1. 准备工作

硬件准备

开发板

首先需要准备一个开发板，这里我准备的是 STM32L4 的开发板（BearPi）：



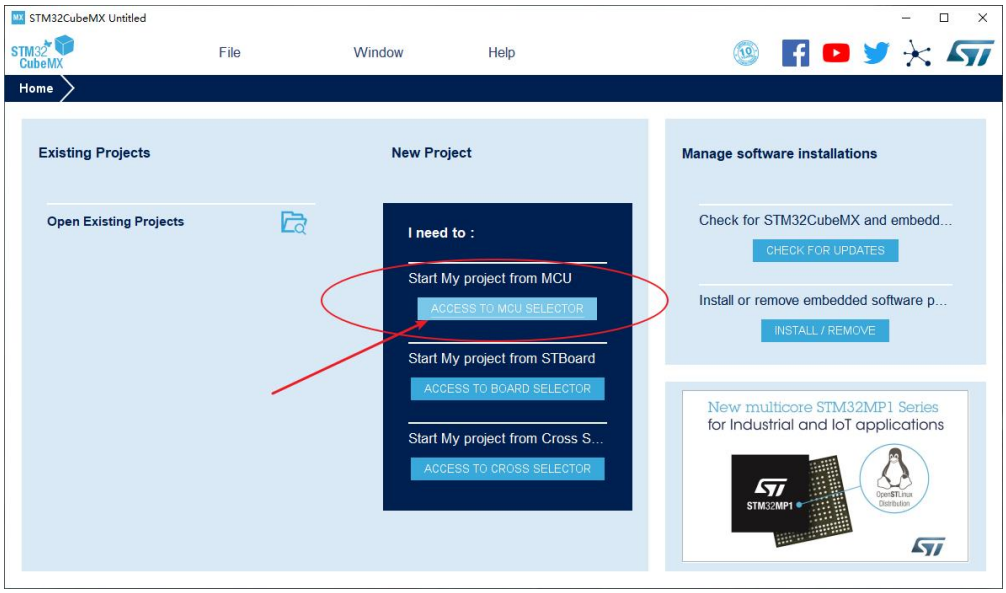
软件准备

- 需要安装好 Keil - MDK 及芯片对应的包，以便编译和下载生成的代码；
  Keil MDK 和串口助手的安装包都可以关注“**小熊派开源社区**”微信公众号，在资料教程一栏中可获取安装包。

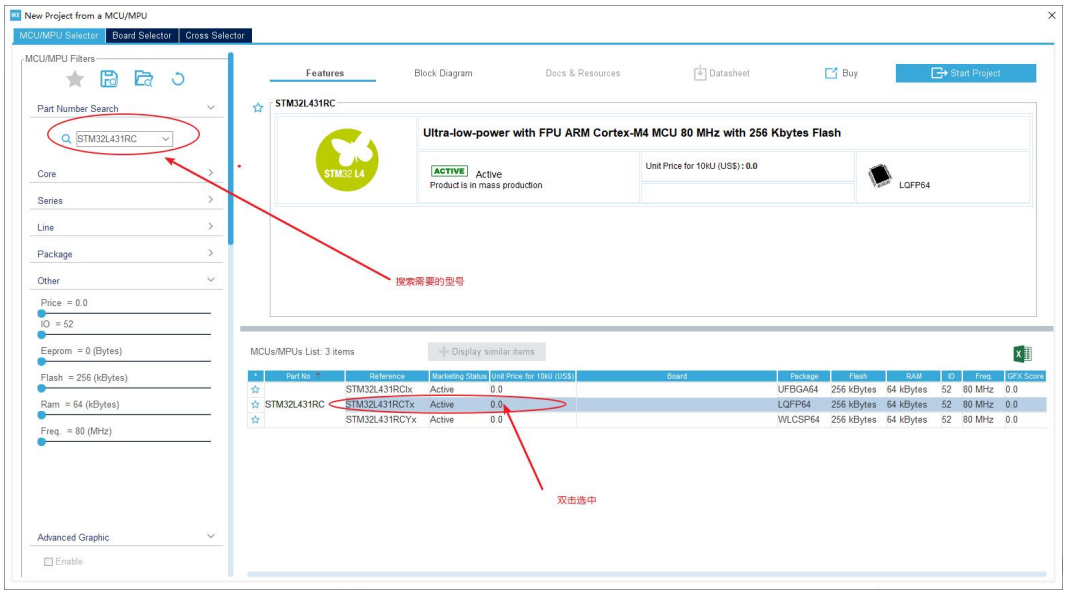
2. 生成 MDK 工程

选择芯片型号

打开 STM32CubeMX，打开 MCU 选择器：



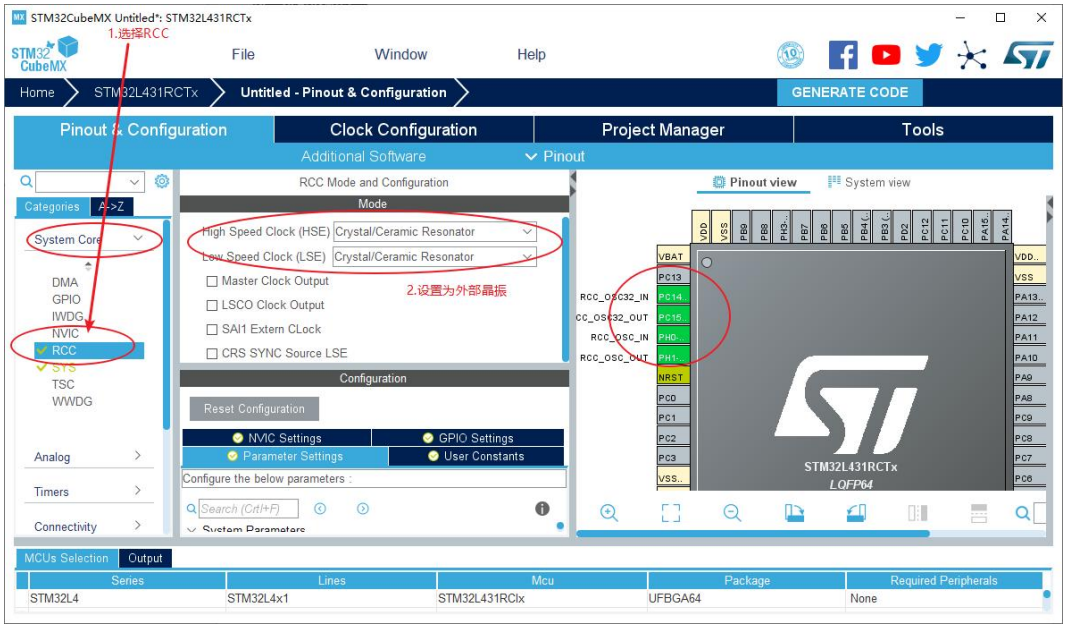
搜索并选中芯片 STM32L431RCT6：



配置时钟源

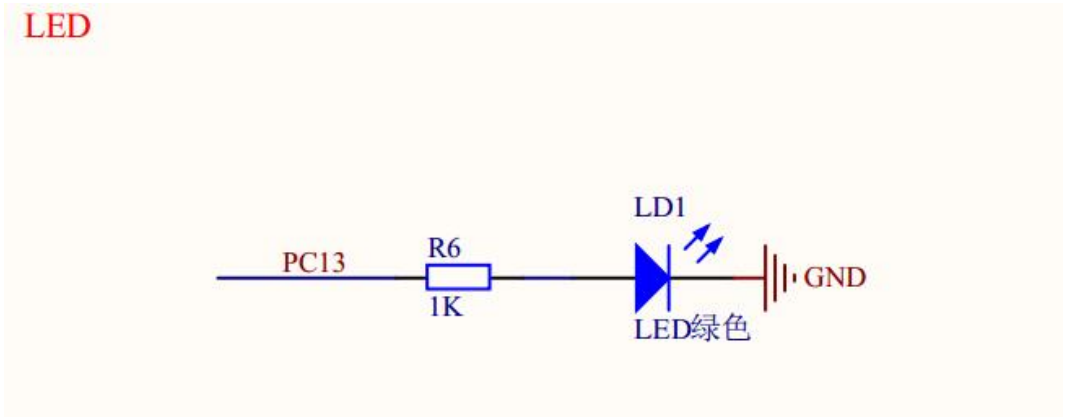
- 如果选择使用外部高速时钟（HSE），则需要在 System Core 中配置 RCC；
- 如果使用默认内部时钟（HSI），这一步可以略过；

这里我都使用外部时钟：

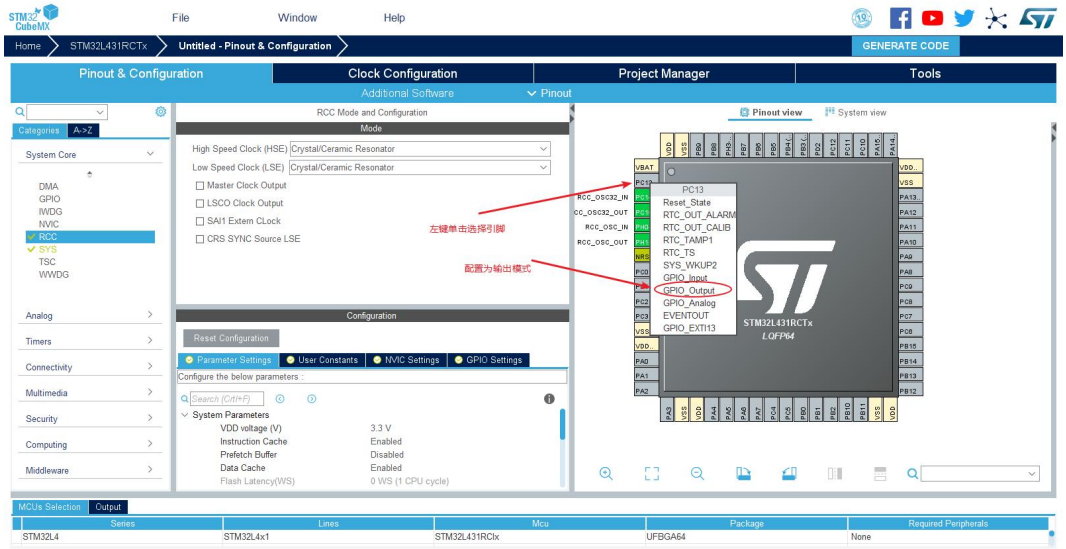


配置 LED 的 GPIO 引脚

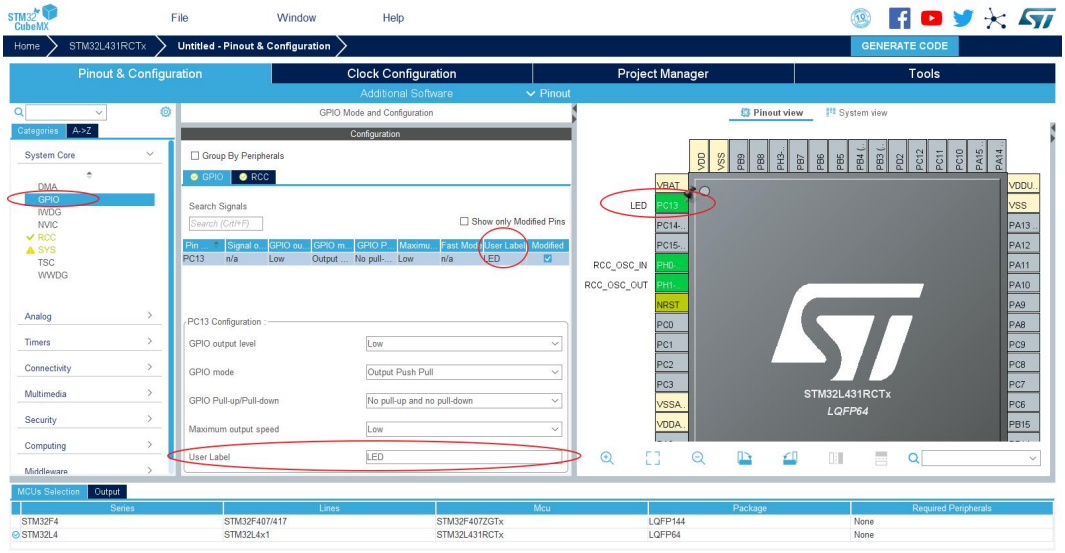
查看小熊派开发板的原理图，如下：



所以接下来我们选择配置 PC13 引脚：

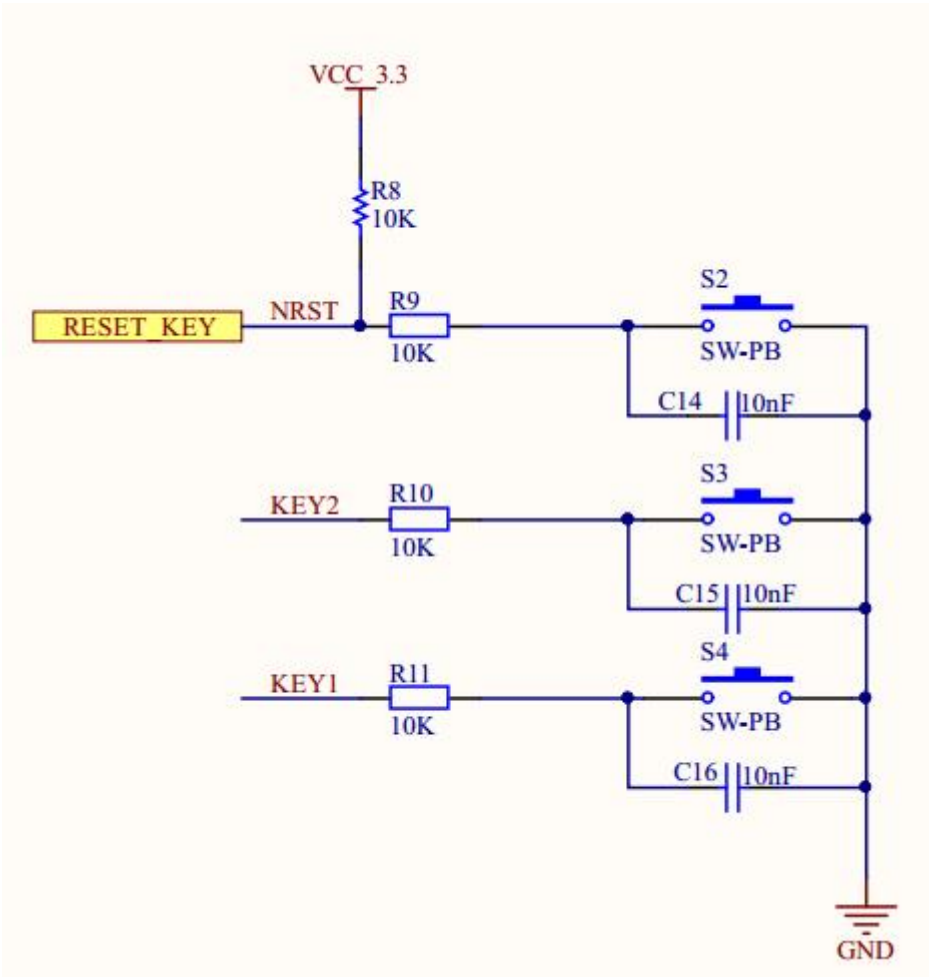


设置用户标签为 LED：

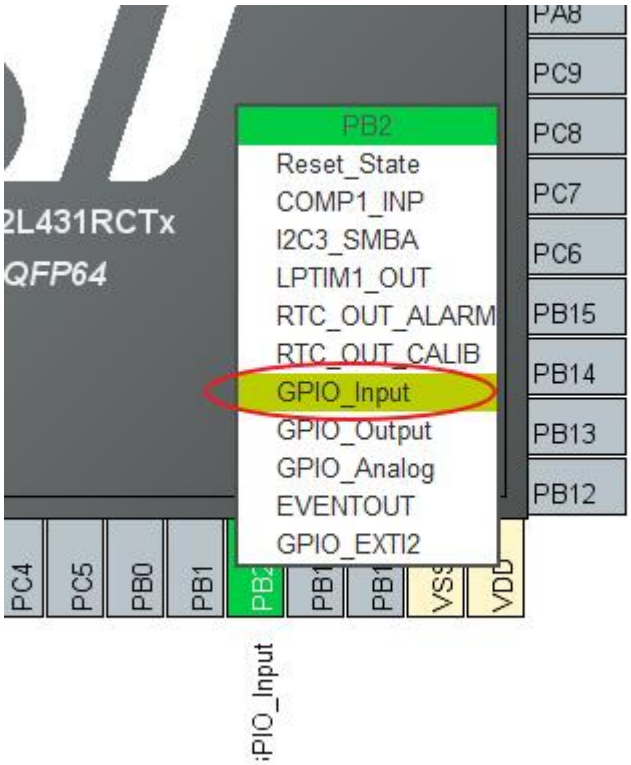


配置按键的 GPIO 引脚

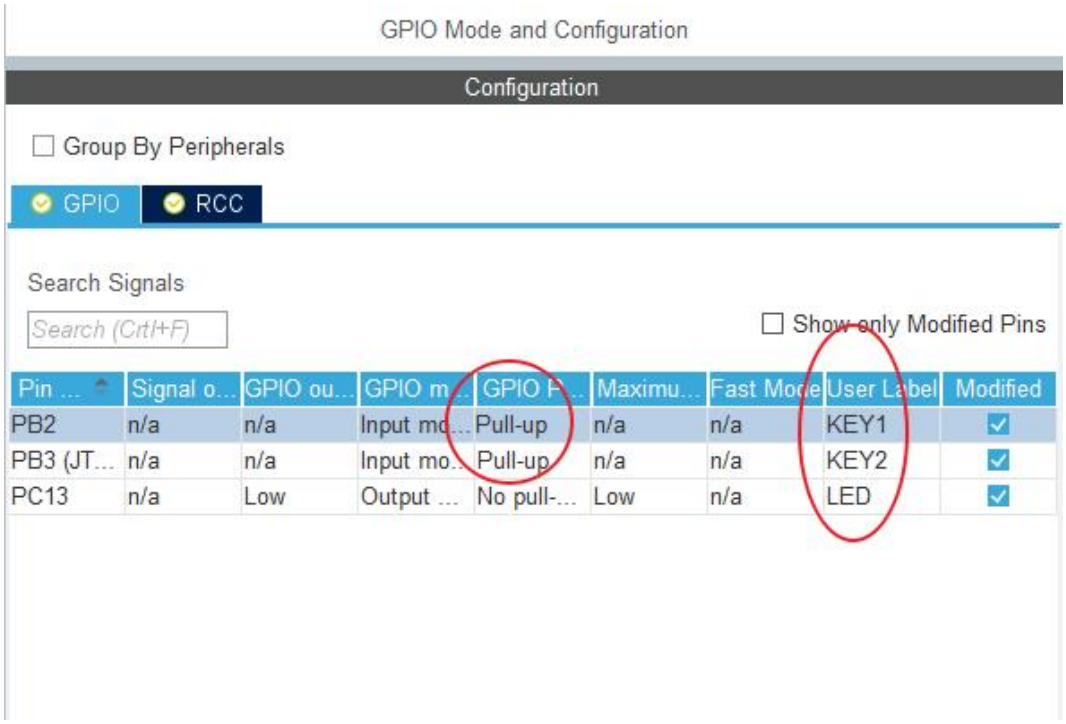
查看小熊派开发板的原理图，如下：



所以接下来我们选择配置 PB2 引脚和 PB3 引脚：

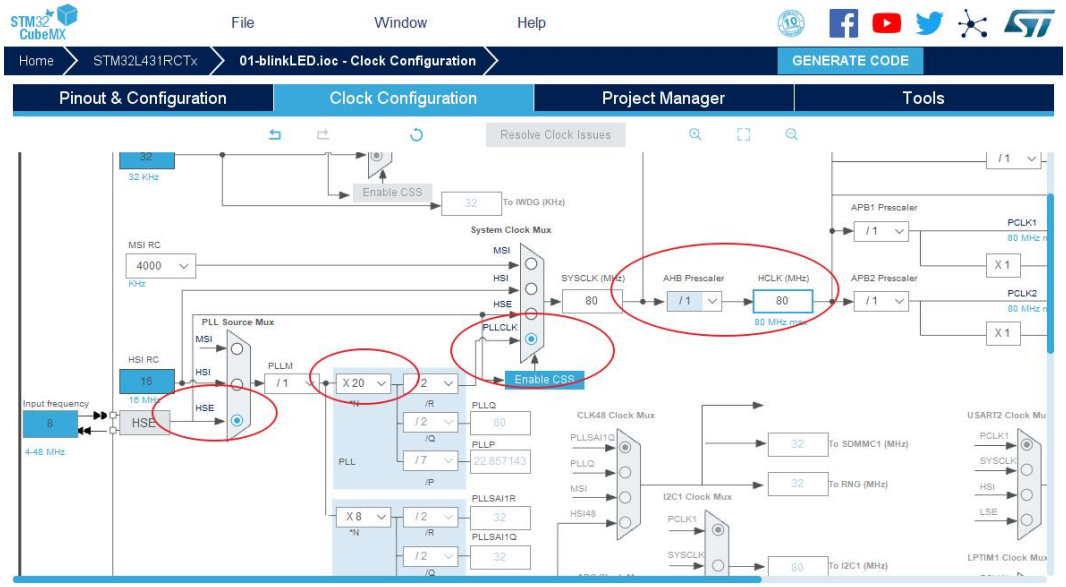


因为没有设置硬件上拉，所以我们配置开启上拉电阻，并设置用户标签为 KEY1 和 KEY2：

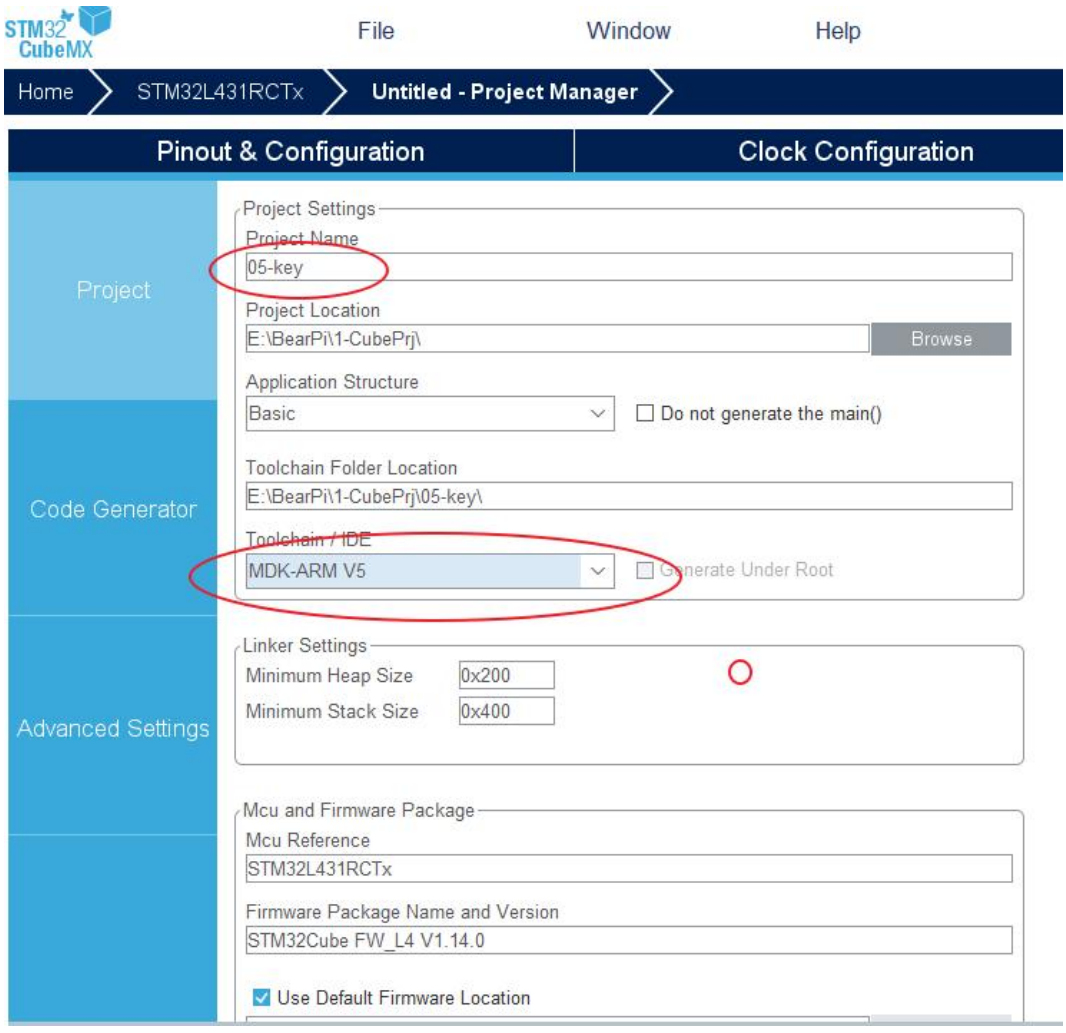


配置时钟树

STM32L4 的最高主频到 80M，所以配置 PLL，最后使 HCLK = 80Mhz 即可：

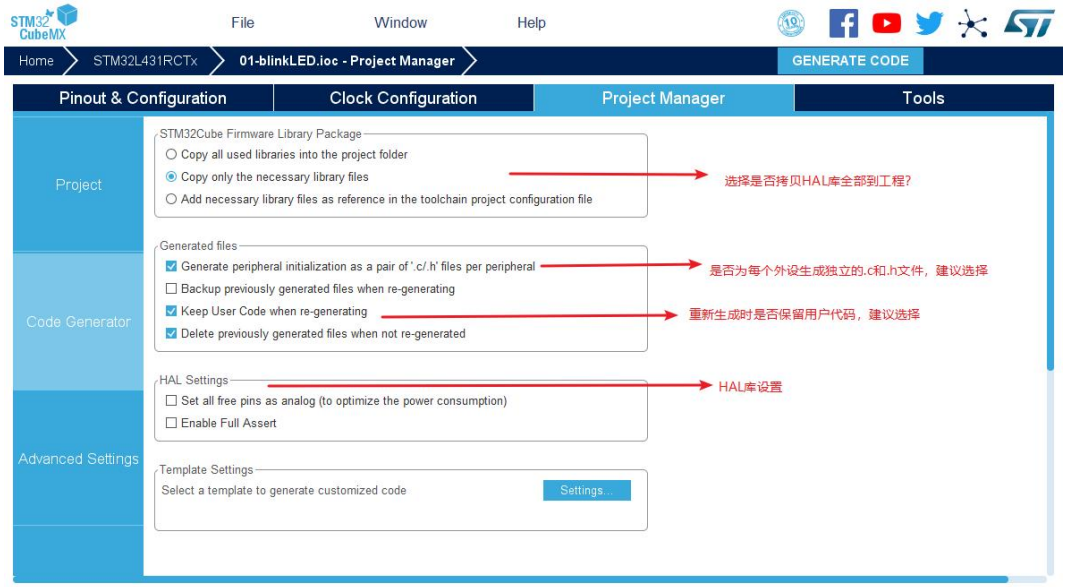


生成工程设置



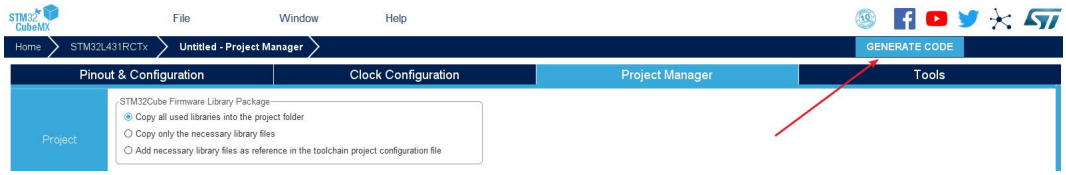
代码生成设置

最后设置生成独立的初始化文件：



生成代码

点击 GENERATE CODE 即可生成 MDK-V5 工程：



3. 在 MDK 中编写、编译、下载用户代码

编写用户代码

进入 MDK-ARM 目录，打开工程，在 main.c 中的 main 函数中编写简单的用户代码：

```
int main(void){


  HAL_Init();


  SystemClock_Config();


  MX_GPIO_Init();
```

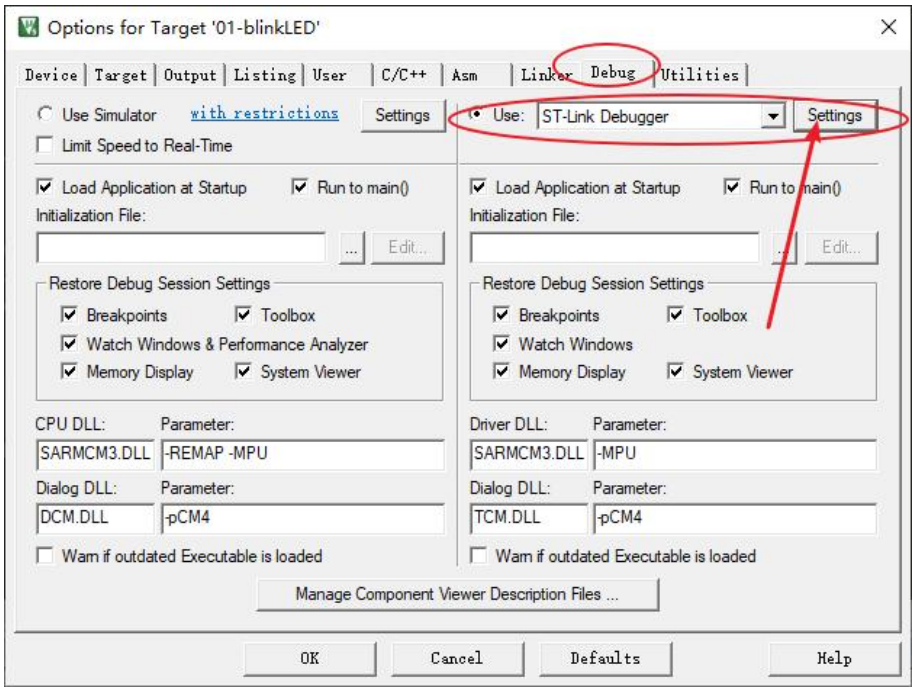
```
  while (1)
  {
    /* USER CODE BEGIN 3 */
                    if(0 == HAL_GPIO_ReadPin(KEY1_GPIO_Port, KEY1_Pin))
                    {
                        HAL_GPIO_WritePin(LED_GPIO_Port,LED_Pin,GPIO_PIN_SET);
                    }
                    if(0 == HAL_GPIO_ReadPin(KEY2_GPIO_Port, KEY2_Pin))
                    {
                        HAL_GPIO_WritePin(LED_GPIO_Port,LED_Pin,GPIO_PIN_RESET);
                    }
  }
  /* USER CODE END 3 */}
```

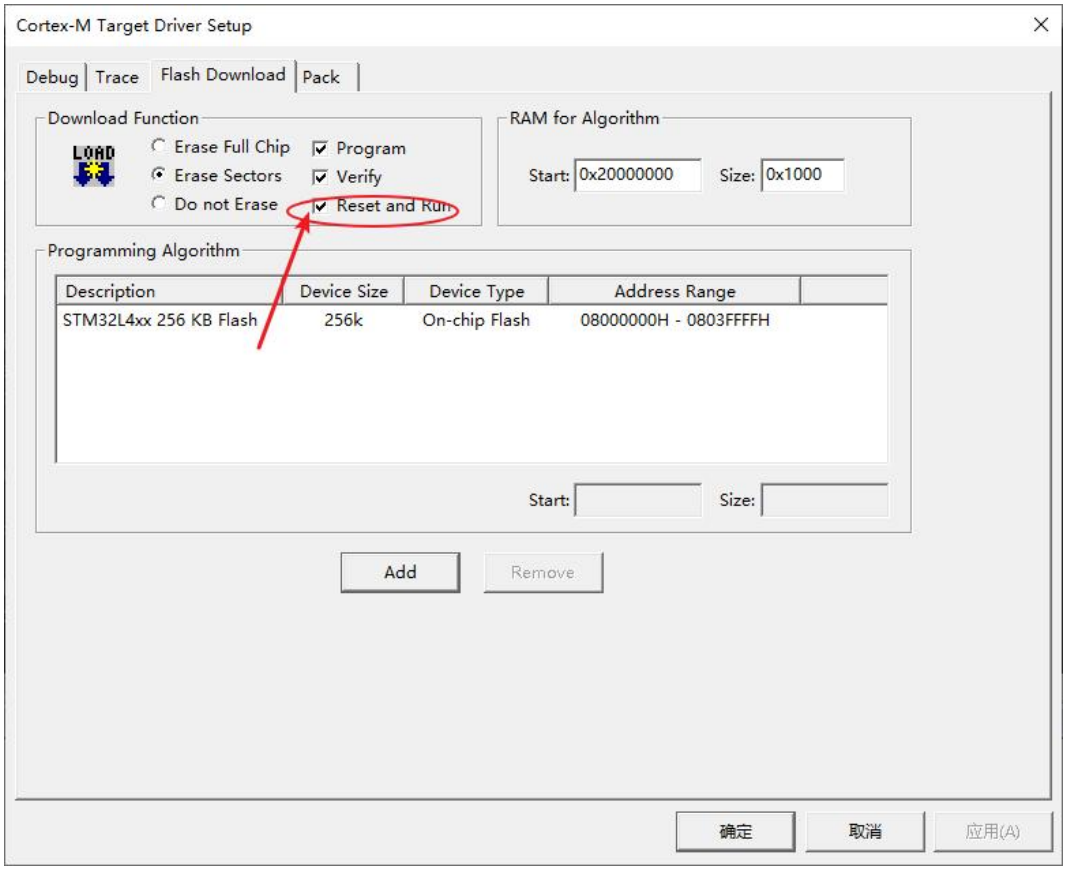
编译代码

编译整个工程：

```
Build Output
Build started: Project: 05-key
*** Using Compiler 'V5.06 update 6 (build 750)', folder: 'D:\Keil_v5\ARM\ARMCC\Bin'
Build target '05-key'
compiling led_drv.c...
linking...
Program Size: Code=3368 RO-data=492 RW-data=16 ZI-data=1024
FromELF: creating hex file...
"05-key\05-key.axf" - 0 Error(s), 0 Warning(s).
Build Time Elapsed:  00:01:17
```

设置下载器





实验现象

下载运行后，实验现象如下：

- 上电复位时 LED 处于熄灭状态；
- 按下 KEY1，LED 点亮；

- 按下 KEY2，LED 熄灭；

至此，我们已经学会了**如何使用 STM32CubeMX 快速生成 MDK 的工程**，以及**如何使用 STM32CubeMX 初始化 GPIO 进行按键检测**，下一节讲述如何配置 NVIC 使用外部中断检测按键。

作业：

设计并编写程序，使用 STM32 单片机的 GPIO 检测按键状态，并在 LED 上显示按键是否被按下。

研究并解释 GPIO 中断在按键检测中的应用（可选，根据课程深度决定）。

STM32 单片机基础 05——使用 EXIT 中断检测按键

教学目的与要求：

目的：理解 EXTI 中断的原理和应用，学习如何通过 EXTI 中断来检测按键的按下事件。

要求：能够配置 EXTI 中断，编写中断服务程序处理按键事件，提高系统的响应速度。

教学重难点：

重点：EXTI 中断的配置方法，中断服务程序的编写。

难点：理解中断的优先级和嵌套机制，确保中断处理的正确性和高效性。

课时数：3 课时

思政元素：

强调系统优化和效率提升的重要性，通过 EXTI 中断的使用，让学生理解在嵌入式系统设计中，合理利用中断可以显著提高系统的实时性和响应速度

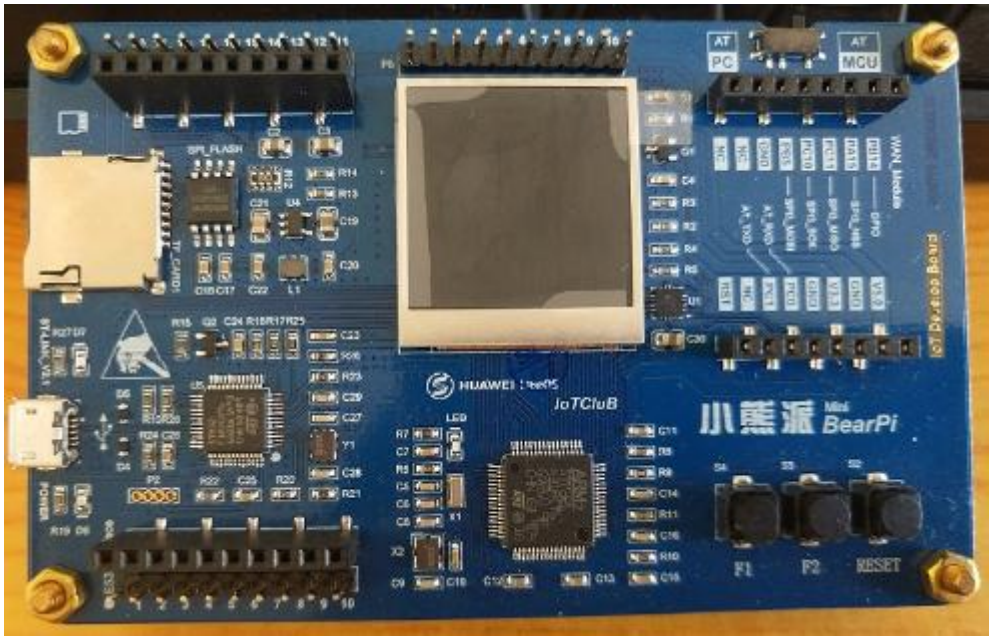
本篇文章主要介绍如何使用 STM32CubeMX 初始化 STM32L431RCT6 的 EXIT 检测按键，讲述了一些 NVIC 的小知识，并一步一步探索了 HAL 库的中断处理机制。

1. 准备工作

硬件准备

开发板

首先需要准备一个开发板，这里我准备的是 STM32L4 的开发板（BearPi）：



软件准备

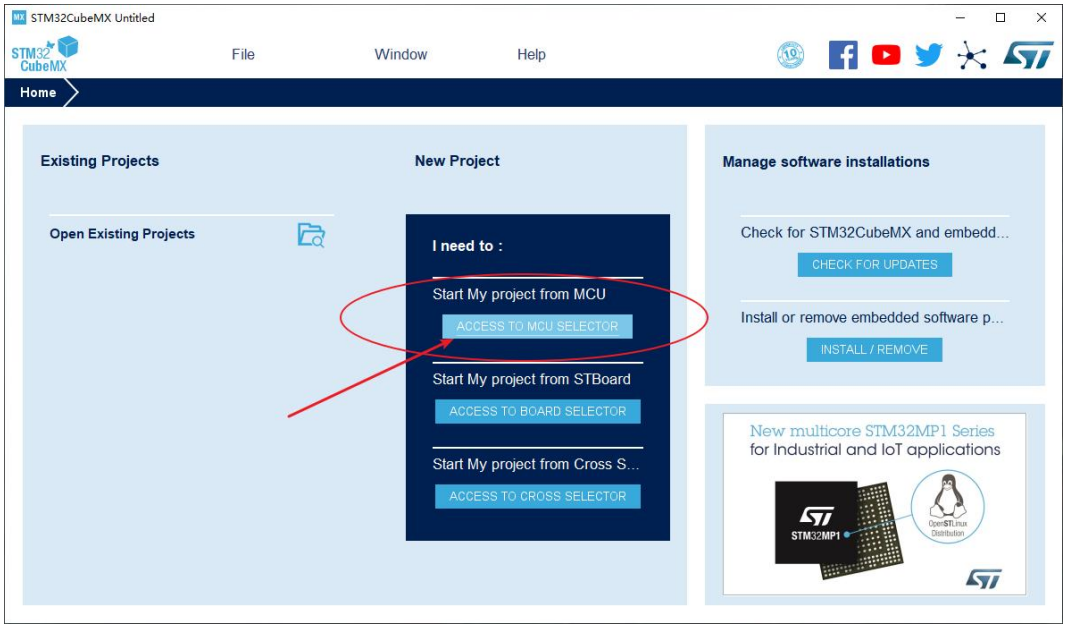
- 需要安装好 Keil - MDK 及芯片对应的包，以便编译和下载生成的代码；

  Keil MDK 和串口助手的安装包都可以关注“**小熊派开源社区**”微信公众号，在资料教程一栏中可获取安装包。

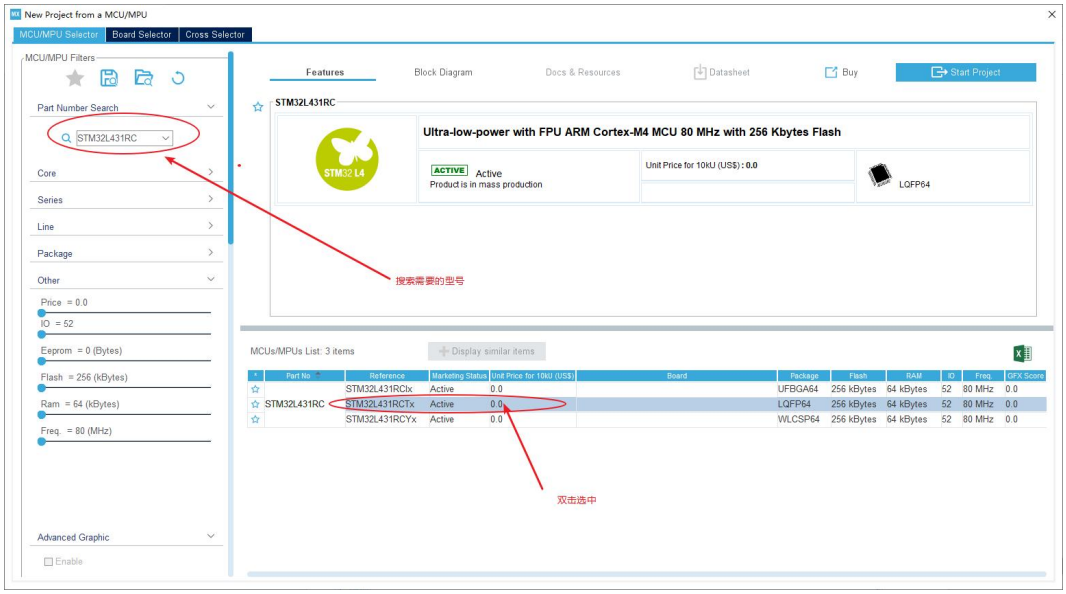
2.生成 MDK 工程

选择芯片型号

打开 STM32CubeMX，打开 MCU 选择器：



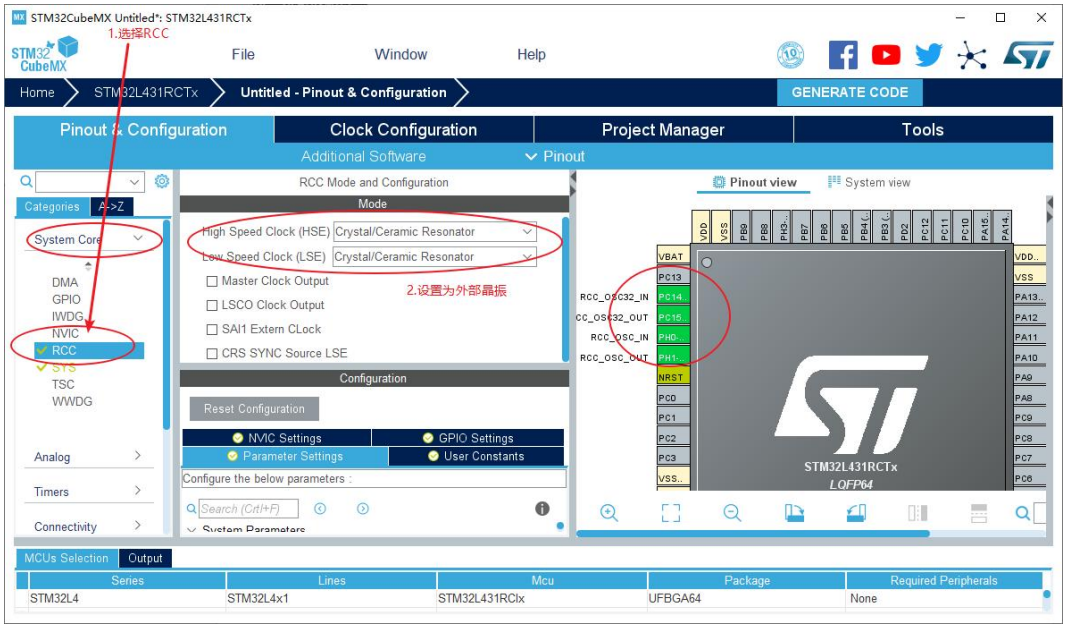
搜索并选中芯片 STM32L431RCT6:



配置时钟源

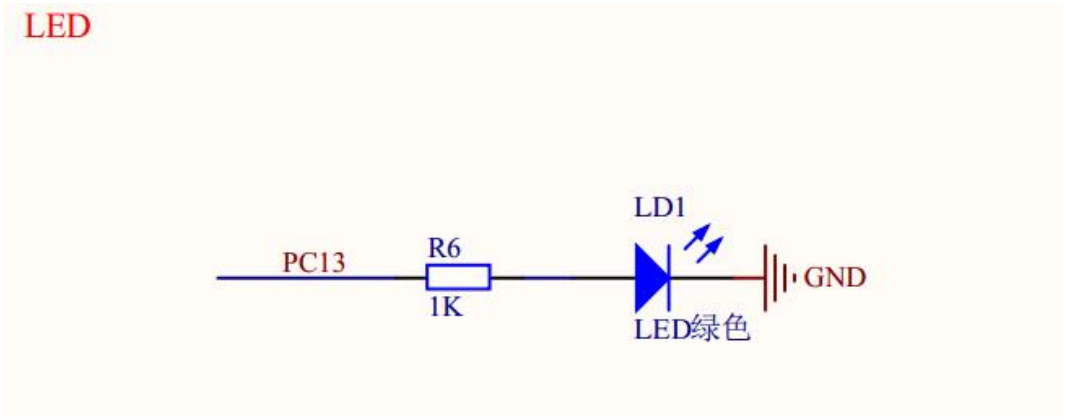
- 如果选择使用外部高速时钟（HSE），则需要在 System Core 中配置 RCC；
- 如果使用默认内部时钟（HSI），这一步可以略过；

这里我都使用外部时钟：

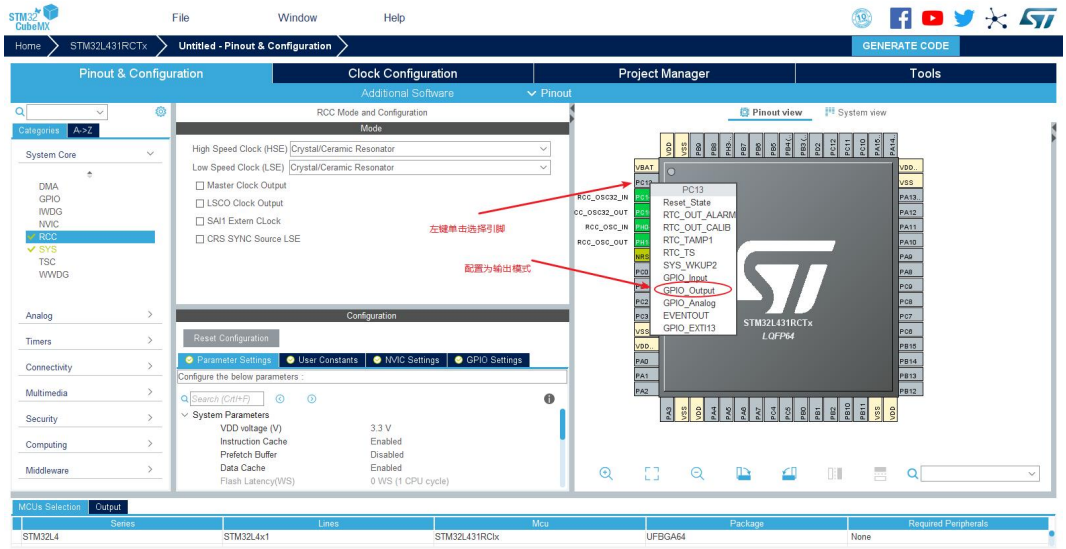


配置 LED 的 GPIO 引脚

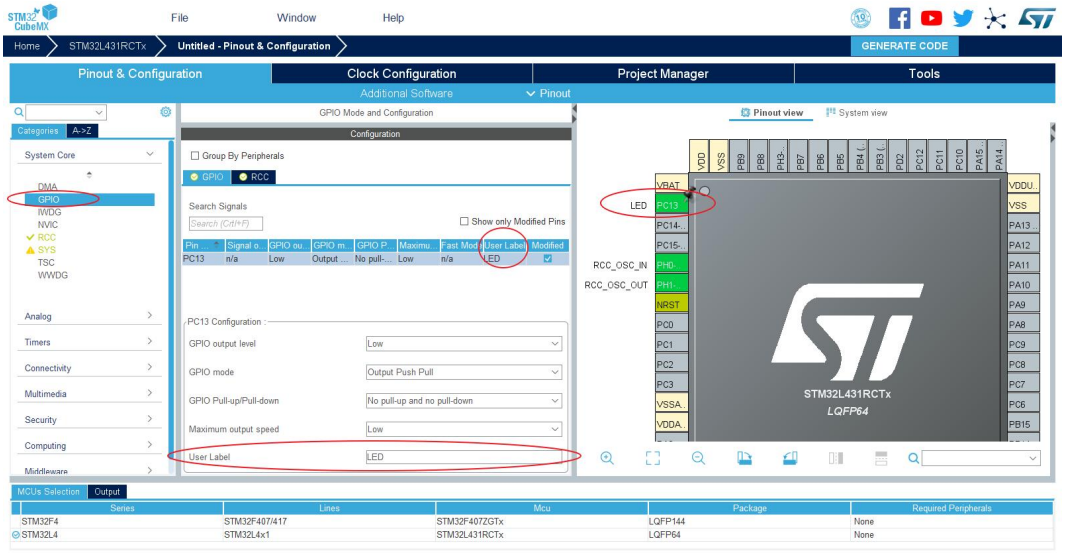
查看小熊派开发板的原理图，如下：



所以接下来我们选择配置 PC13 引脚：

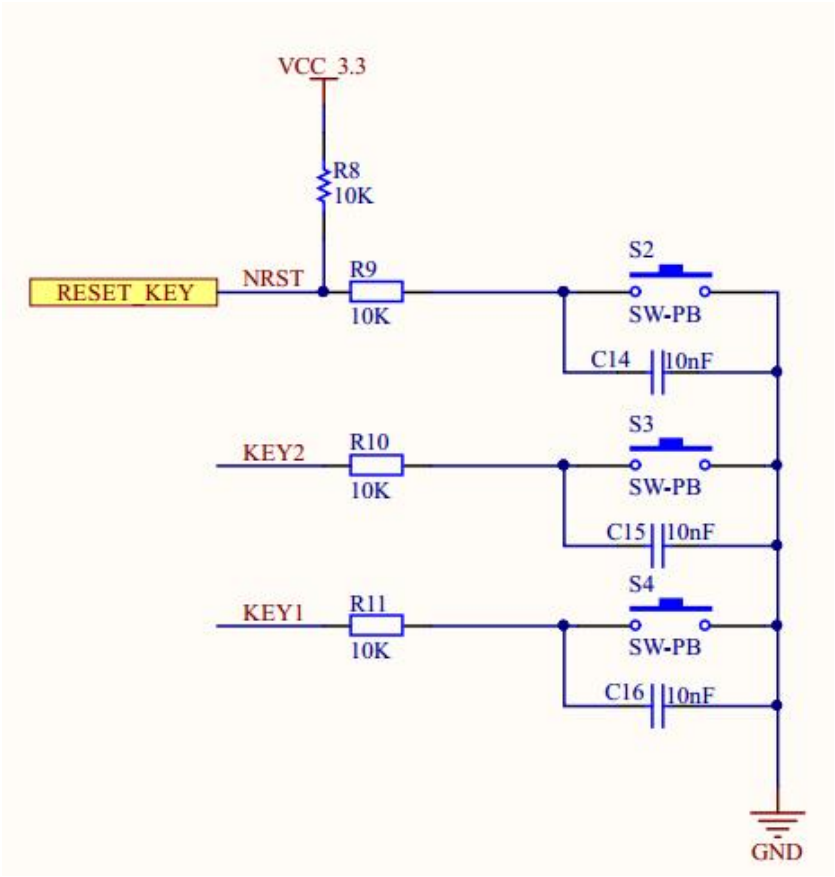


设置用户标签为 LED：

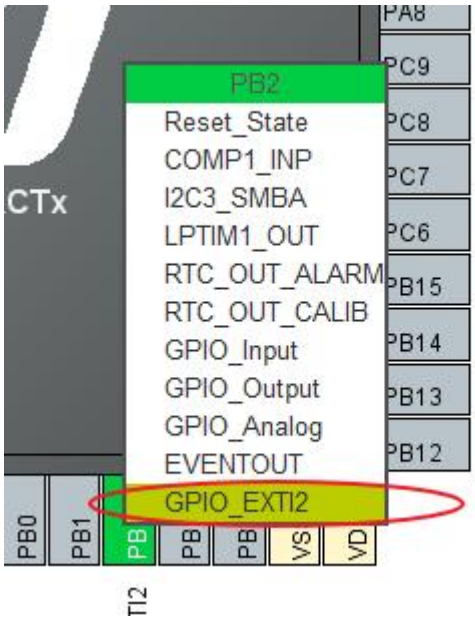


配置 GPIO 引脚为外部中断引脚

查看小熊派开发板的原理图，如下：



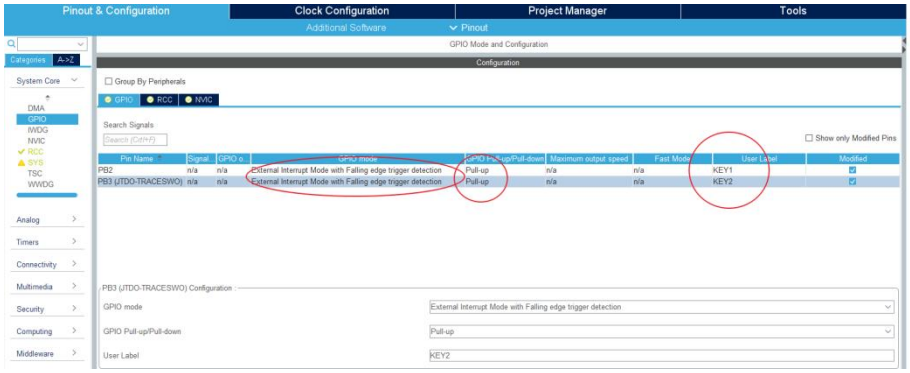
所以接下来我们选择配置 PB2 引脚和 PB3 引脚为外部中断引脚：



因为没有设置硬件上拉，所以我们配置开启上拉电阻，并设置用户标签为 KEY1 和 KEY2，接下来是最重要的一步：

- 开启下降沿触发中断：即在**按下按键时**电平由高变为低时触发
- 开启上升沿触发中断：即在**按下按键后松开时**电平由低变为高时触发
- 开启下降沿上升沿都触发中断：即在**按下时触发，松开时再次触发**

这里我选择开启下降沿触发中断：



配置 NVIC 设置中断优先级

知识小卡片 —— NVIC

NVIC 全称 Nested vectored interrupt controller，即嵌套向量中断控制器，用来决定**中断的优先级**。

NVIC 在 ARM Conrtex-M 内核中，用一个 8 位的寄存器来配置，总共可以配置 2 8 = 256 2^8=25628=256 级中断，但是 ST 公司在生产 STM32 的时候，发现一个小小的单片机根本用不了这么多，纯属浪费，所以将该寄存器的低 4 位 全部置 0，只使用高 4 位来配置，这样一来 STM32 就只有 2 4 = 16 2^4=1624=16 级中断啦。

简化为 16 级中断后，ST 发现 STM32 内部这么丰富的外设，还是不方便配置，干脆**人工给这 4 位来个分组**，划分出了 5 个分组：

优先级分组	抢占优先级占的位数	子优先级占的位数
NVIC_PriorityGroup_0	0 bit	4 bit
NVIC_PriorityGroup_1	1 bit	3 bit
NVIC_PriorityGroup_2	2 bit	2 bit
NVIC_PriorityGroup_3	3 bit	1 bit
NVIC_PriorityGroup_4	4 bit	0 bit

再次强调一下，这 5 种中断分组规则是人为的，用哪种规则，之后设置具体的优先级时对应就行，STM32 默认使用的规则是 NVIC_PriorityGroup_0 。

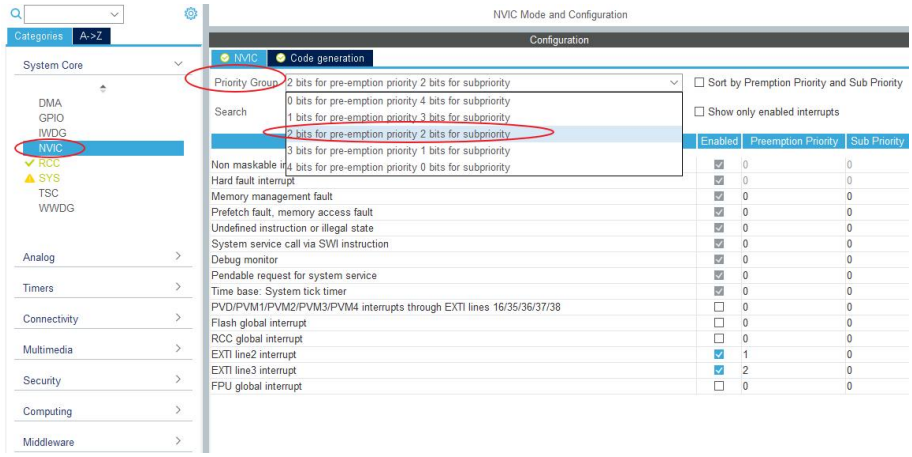
STM32 的 CPU 判断优先级的方法如下：

- 先判断抢占优先级，数字越小，优先级越高；
- 若抢占优先级相同，判断子优先级，同样，数字越小，优先级越高；
  知识小卡片结束啦~ 对 NVIC 有没有了解呢？

接下来在 STM32CubeMX 中配置中断优先级：

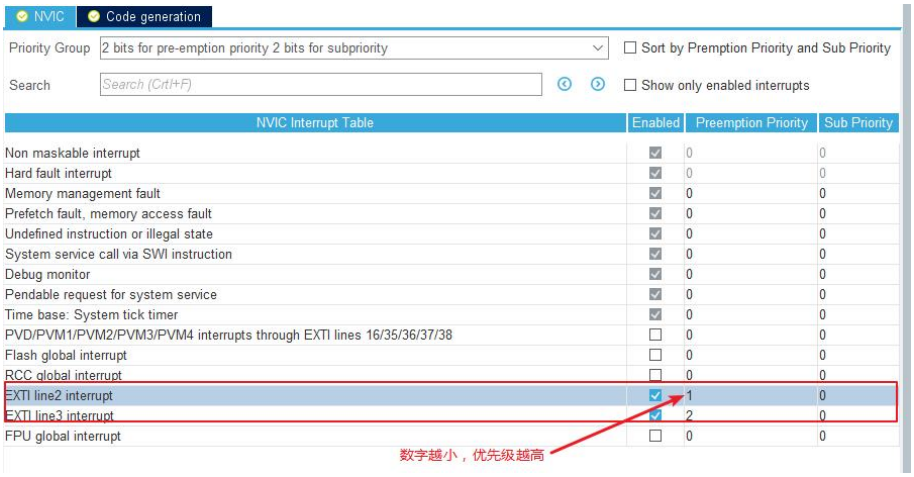
配置优先级分组

这里我配置使用中断优先级分组规则 NVIC_PriorityGroup_2：



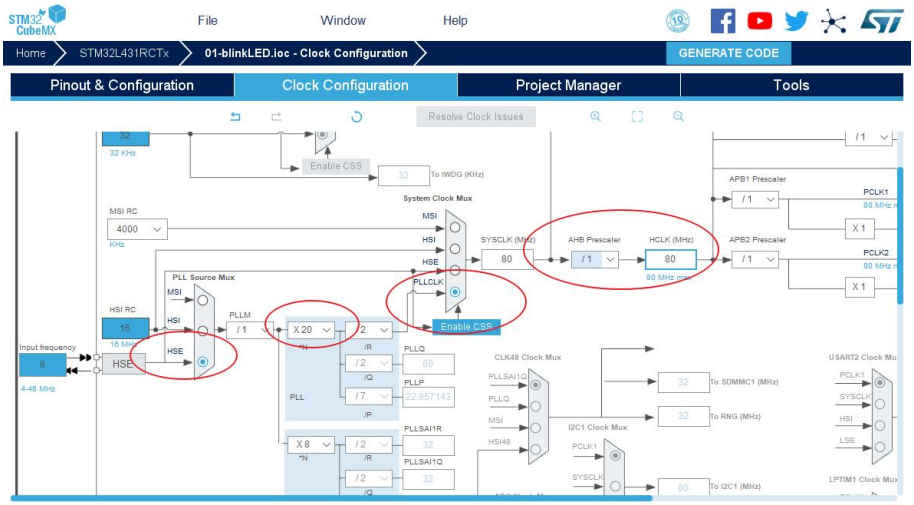
配置具体的优先级大小

根据中断优先级分组规则 NVIC_PriorityGroup_2 来设置具体的优先级大小：

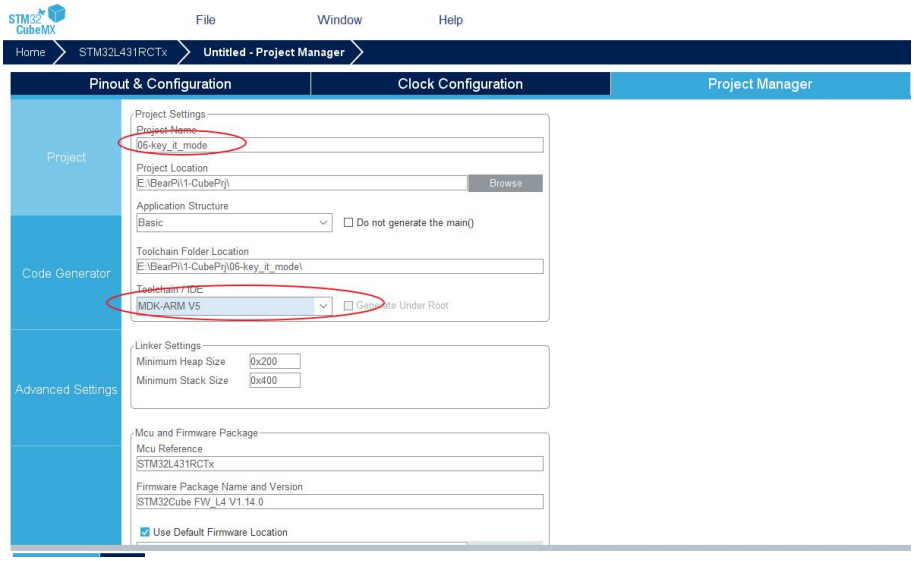


配置时钟树

STM32L4 的最高主频到 80M，所以配置 PLL，最后使 `HCLK = 80Mhz` 即可：

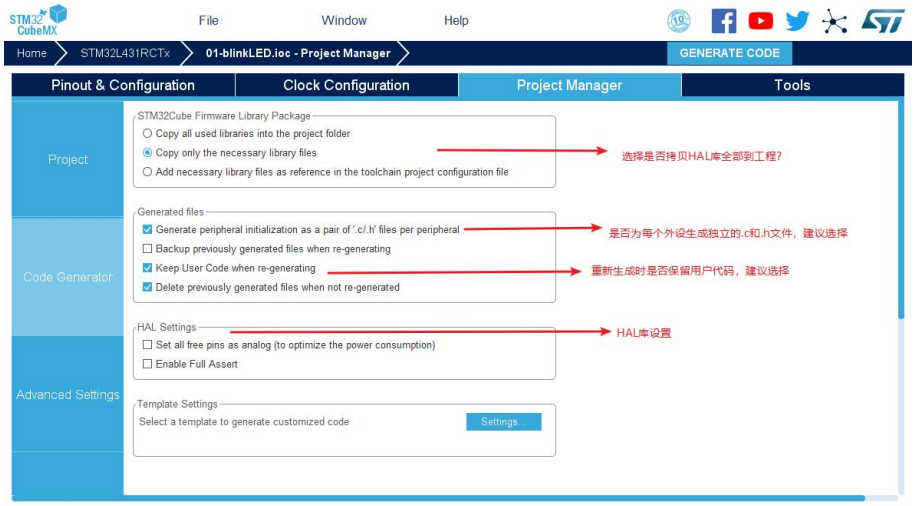


生成工程设置



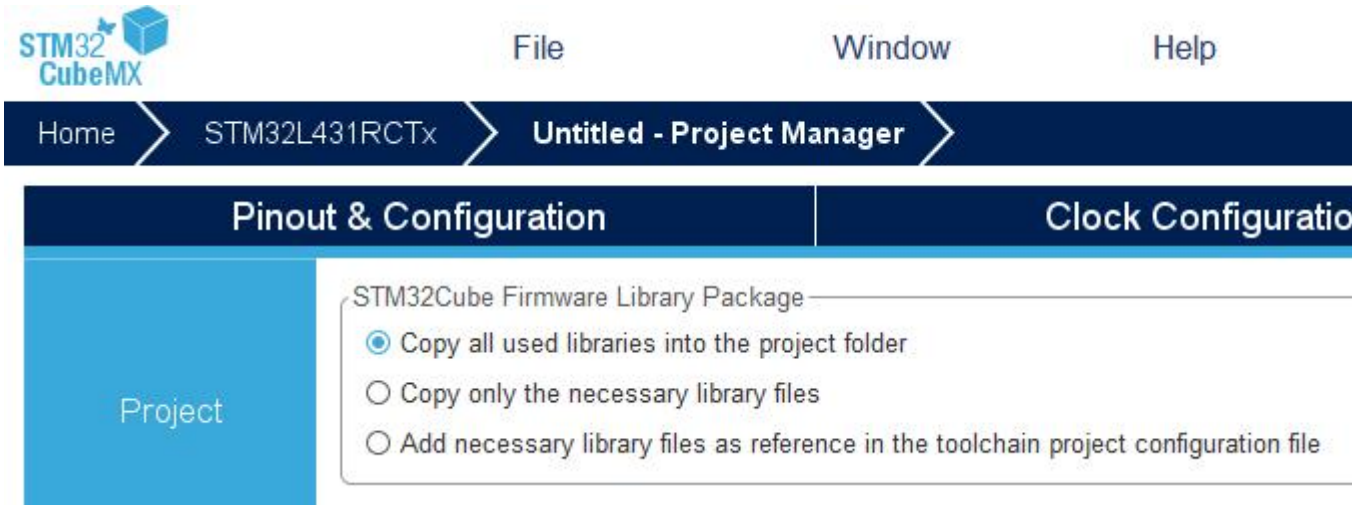
代码生成设置

最后设置生成独立的初始化文件：



生成代码

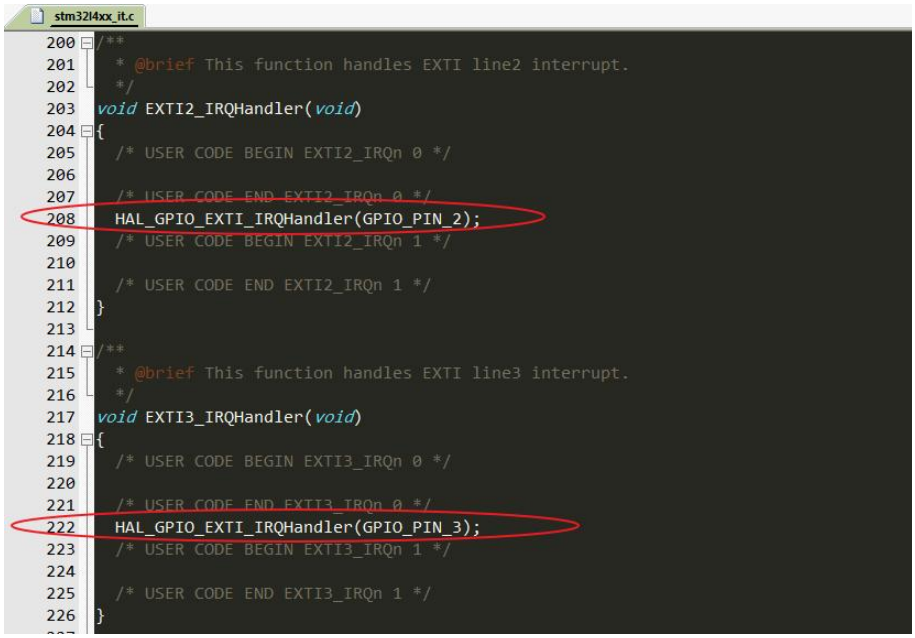
点击 `GENERATE CODE` 即可生成 MDK-V5 工程：



3. 在 MDK 中编写、编译、下载用户代码

STM32 HAL 库中断处理机制

先打开 `stm32l4xx_it.c` 文件：



可以看到其中处理 EXIT2 和 EXIT3 中断都调用了同一个函数，但是 EXIT2 和 EXIT3 向该函数传入的参数不同：

```
HAL_GPIO_EXTI_IRQHandler();
```

那么，HAL 库对于中断是如何处理的呢？我们打开 `stm32l4xx_hal_gpio.c` 文件，看一下该函数的原型，一探究竟：

```
/**
  * @brief  Handle EXTI interrupt request.
  * @param  GPIO_Pin Specifies the port pin connected to corresponding EXTI line.
  * @retval
  */void HAL_GPIO_EXTI_IRQHandler(uint16_t GPIO_Pin){
  /* EXTI line interrupt detected */
  if(__HAL_GPIO_EXTI_GET_IT(GPIO_Pin) != 0x00u)
  {
    __HAL_GPIO_EXTI_CLEAR_IT(GPIO_Pin);
    HAL_GPIO_EXTI_Callback(GPIO_Pin);
  }}
```

可以看到，在该函数中首先读取了一下中断寄存器，确认该中断是否发生，确认之后又调用了一个函数，并将接收到的参数 GPIO_Pin 继续传给该函数：

```
HAL_GPIO_EXTI_Callback(GPIO_Pin);
```

该函数称为 EXIT 中断的回调函数，用来处理所有发生的 EXIT 中断事件。

那么，这个函数又干了什么呢？接着探索哈哈哈~

同样在 stm32l4xx_hal_gpio.c 文件中找到该函数的原型：

```
/**
  * @brief  EXTI line detection callback.
  * @param  GPIO_Pin: Specifies the port pin connected to corresponding EXTI line.
  * @retval
  */
__weak void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin){
  /* Prevent unused argument(s) compilation warning */
  UNUSED(GPIO_Pin);

  /* NOTE: This function should not be modified, when the callback is needed,
           the HAL_GPIO_EXTI_Callback could be implemented in the user file
   */}
```

哈哈哈，这下是不是非常清楚了~

该回调函数使用__weak 进行了弱定义，所以**用户可以再次定义该函数**，并且这个 note 写的非常清楚：

> 这个函数不应该被改变，如果需要使用回调函数，请重新在用户文件中实现该函数。

自己实现 EXIT 中断处理回调函数

这个函数放在哪都行，为了方便，我们放在 gpio.c 的最后。

实现的基本思想是：

- 因为所有的 EXIT 中断都会调用该函数，所以首先判断具体的中断事件；
- 对该中断事件进行处理

实现代码如下：

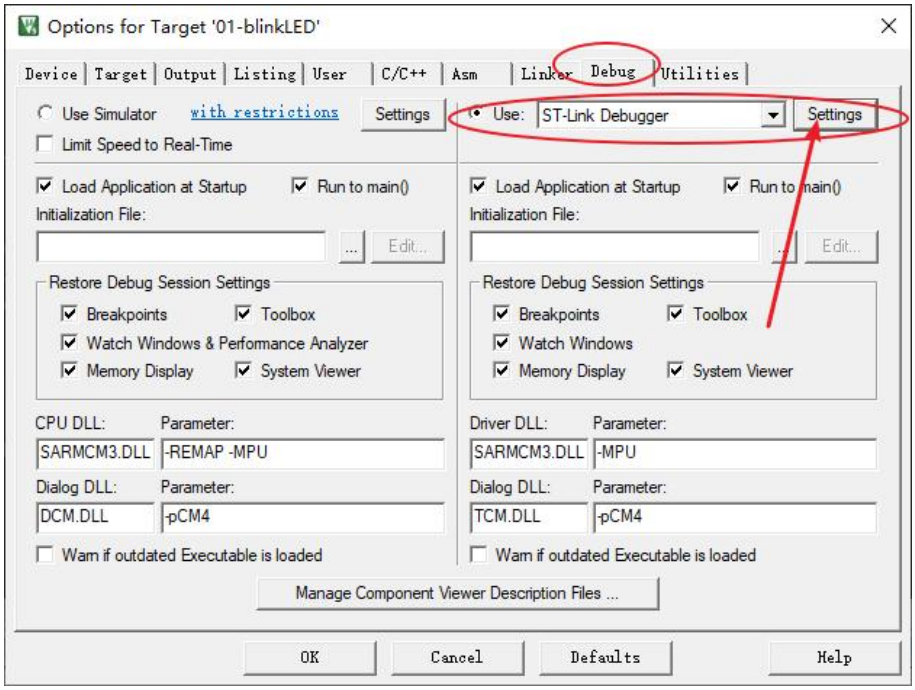
```
/* USER CODE BEGIN 2 *//**
 * @brief EXIT中断回调函数
 * @param GPIO_Pin —— 触发中断的引脚
 * @retval none
*/void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin){
        /* 判断哪个引脚触发了中断 */
        switch(GPIO_Pin)
        {
                case GPIO_PIN_2:
                        /* 处理GPIO2发生的中断 */
                        //点亮LED
                        HAL_GPIO_WritePin(LED_GPIO_Port,LED_Pin,GPIO_PIN_SET);
                        break;
                case GPIO_PIN_3:
                        /* 处理GPIO3发生的中断 */
                        //熄灭LED
                        HAL_GPIO_WritePin(LED_GPIO_Port,LED_Pin,GPIO_PIN_RESET);
                        break;
                default:
                        break;
        }}/* USER CODE END 2 */
```

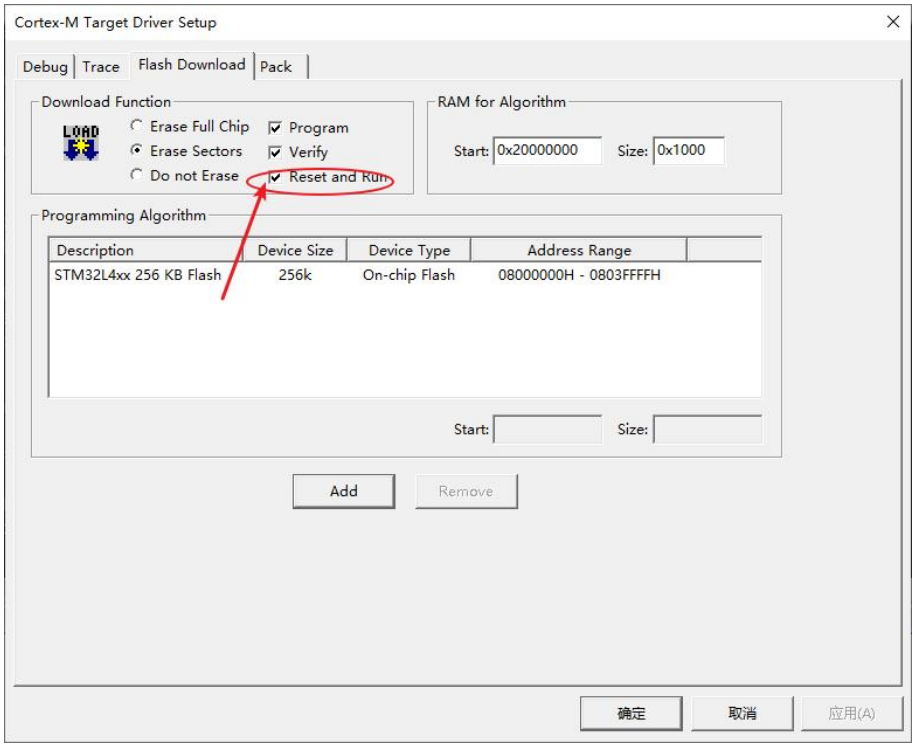
编译代码

编译整个工程：

```
Build Output
compiling stm32l4xx_hal_pwr_ex.c...
compiling stm32l4xx_hal_cortex.c...
compiling stm32l4xx_hal_exti.c...
compiling system_stm32l4xx.c...
linking...
Program Size: Code=3388 RO-data=492 RW-data=16 ZI-data=1024
FromELF: creating hex file...
"06-key_it_mode\06-key_it_mode.axf" - 0 Error(s), 0 Warning(s).
Build Time Elapsed:  00:00:16
```

设置下载器

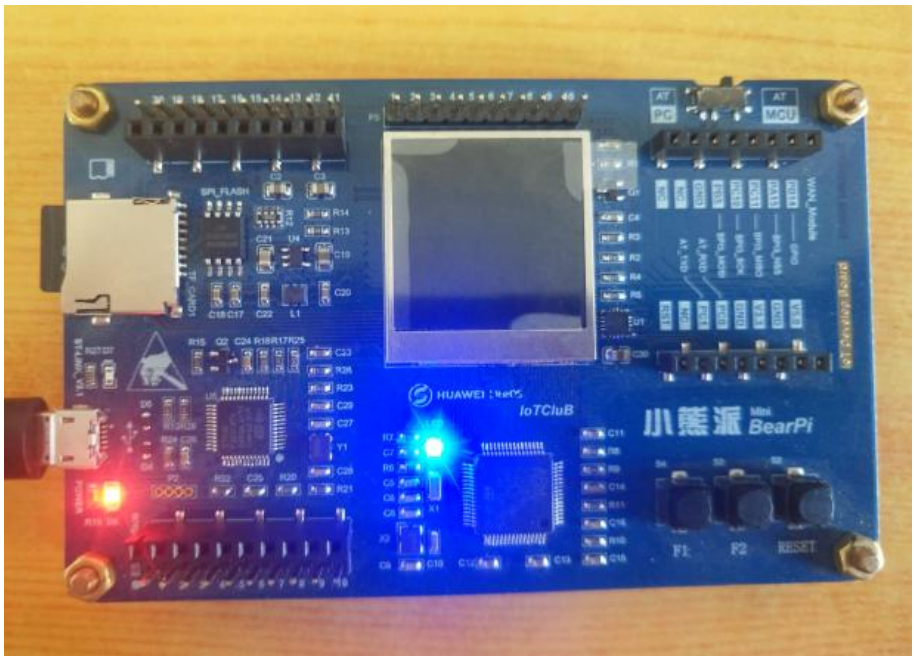




实验现象

下载运行后，实验现象如下：

- 上电复位时 LED 处于熄灭状态；
- 按下 KEY1，LED 点亮；
- 按下 KEY2，LED 熄灭；



至此，我们已经学会了**如何配置 NVIC 使用外部中断检测按键**，并了解了 NVIC 和 HAL 库中断处理机制的一些基本知识，下一节讲述如何配置 USART 以及实现 `printf` 函数。

作业：

使用 STM32CubeMX 配置 EXTI 中断，编写程序通过中断方式检测按键状态，并在 LED 上显示。

分析并比较轮询检测与中断检测在按键检测中的效率和优缺点。

STM32 单片机基础 06——使用 USART 发送和接收数据(查询模式)

教学目的与要求：

目的：理解 USART 通信原理，掌握 USART 的查询模式，实现数据的发送和接收。

要求：能够配置 USART 参数（波特率、数据位、停止位等），编写程序实现数据的发送和接收。

教学重难点：

重点：USART 的配置和数据的发送接收流程。

难点：理解 USART 通信协议，确保数据传输的正确性和可靠性。

课时数：3 课时

思政元素：

培养学生的通信协议意识，通过 USART 通信的实践，理解在嵌入式系统设计中，正确配置通信参数和遵循通信协议的重要性，培养学生的严谨性和规范性。

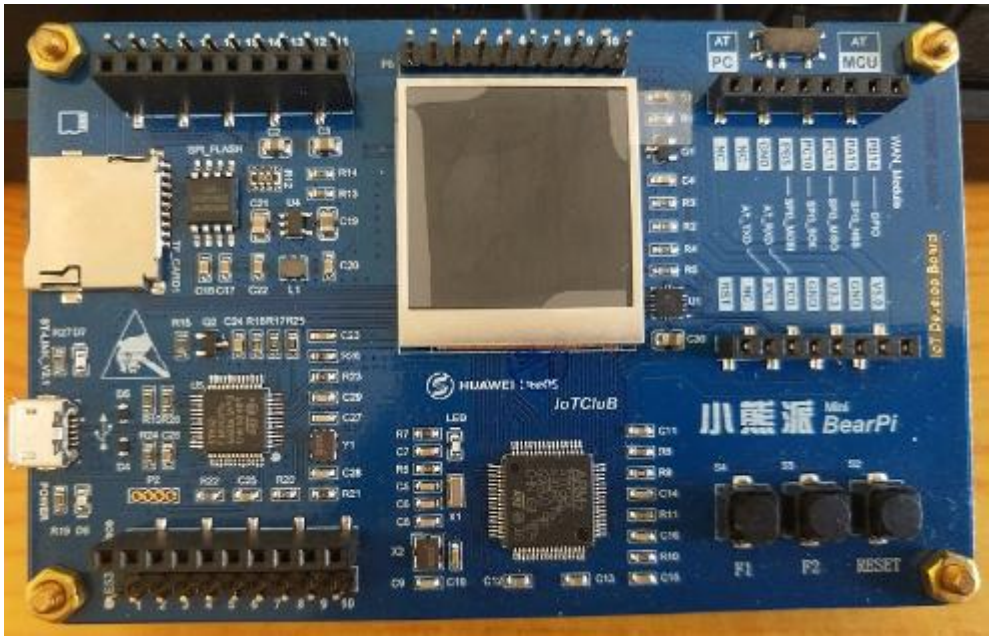
本篇文章主要介绍如何使用 STM32CubeMX 初始化 STM32L431RCT6 的 USART，并使用查询模式发送数据，使用查询模式接收数据。

1. 准备工作

硬件准备

开发板

首先需要准备一个开发板，这里我准备的是 STM32L4 的开发板（BearPi）：



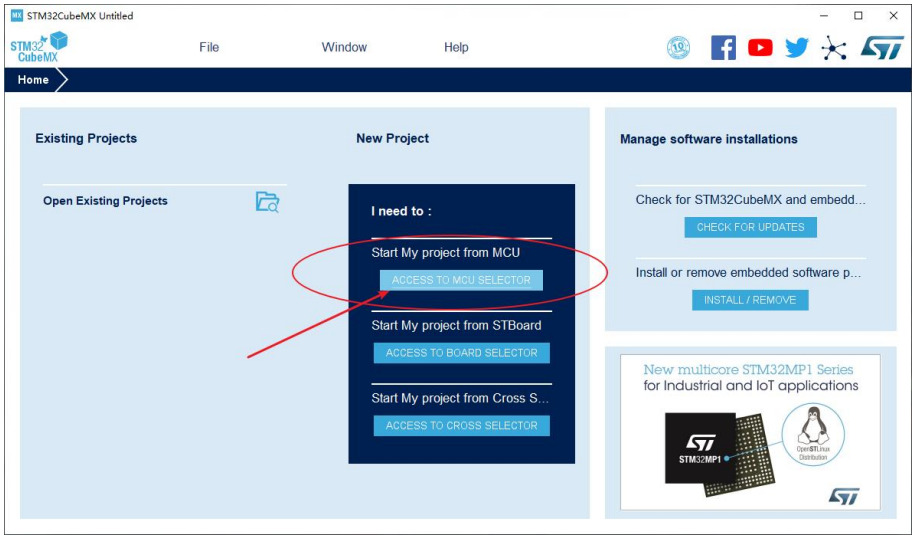
软件准备

- 需要安装好 Keil - MDK 及芯片对应的包，以便编译和下载生成的代码；
  Keil MDK 和串口助手的安装包都可以关注“**小熊派开源社区**”微信公众号，在资料教程一栏中可获取安装包。

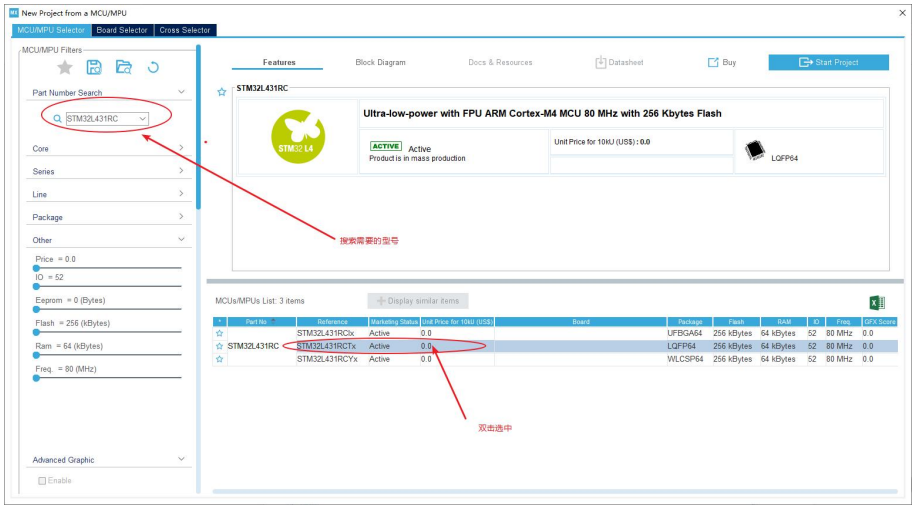
2.生成 MDK 工程

选择芯片型号

打开 STM32CubeMX，打开 MCU 选择器：



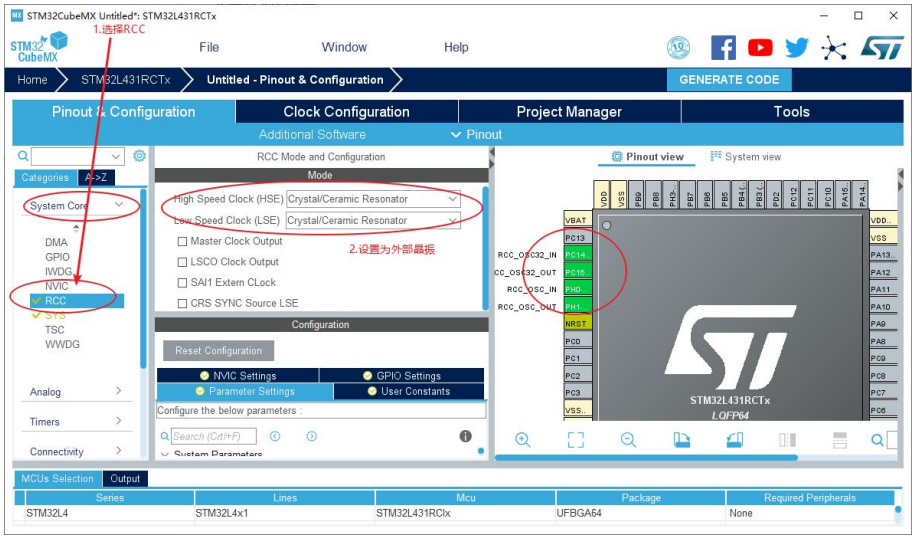
搜索并选中芯片 STM32L431RCT6:



配置时钟源

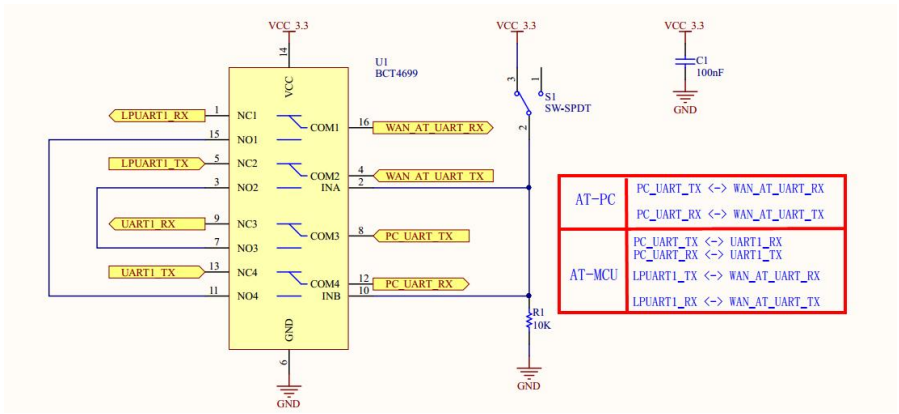
- 如果选择使用外部高速时钟（HSE），则需要在 System Core 中配置 RCC；
- 如果使用默认内部时钟（HSI），这一步可以略过；

这里我都使用外部时钟：



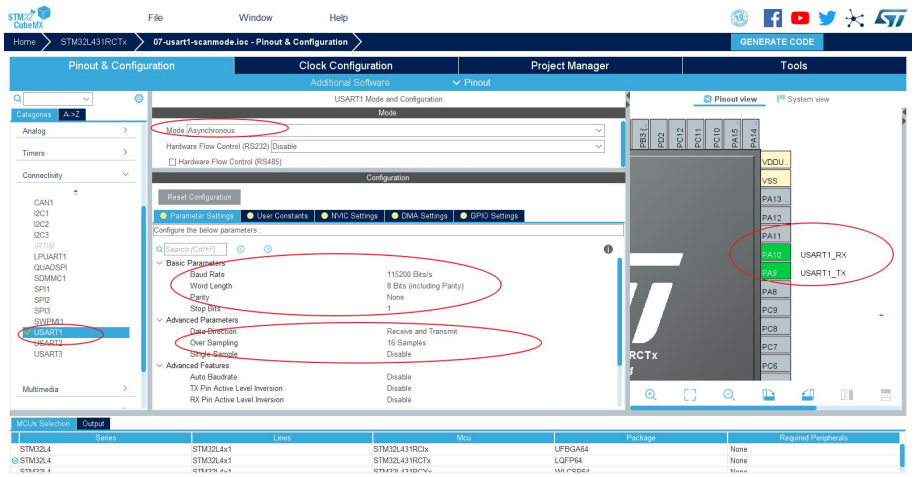
配置串口

小熊派开发板板载 ST-Link 并且虚拟了一个串口，原理图如下：



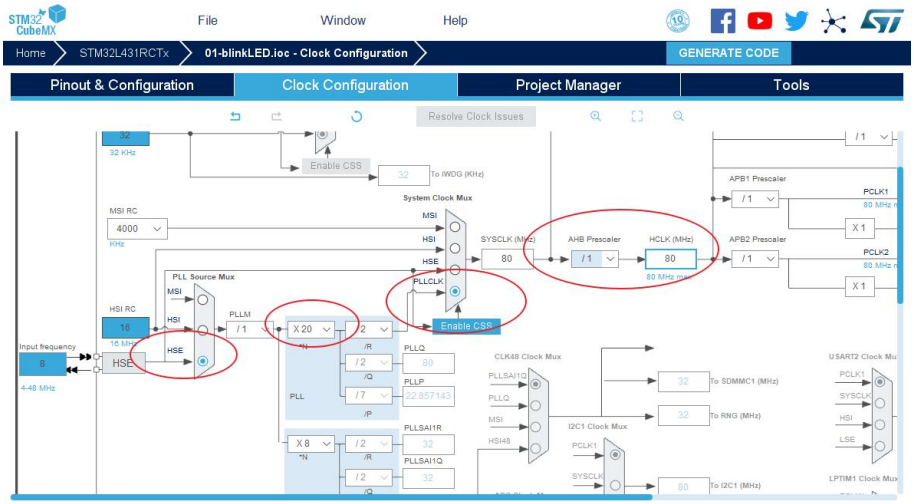
这里我将开关拨到 AT-MCU 模式，使 PC 的串口与 USART1 之间连接。

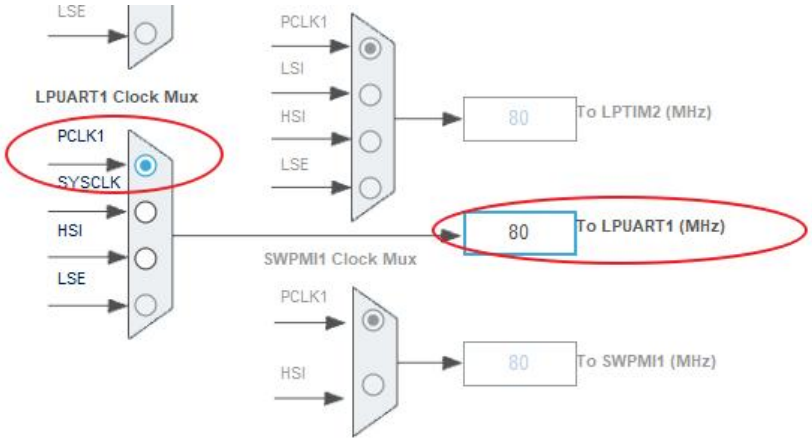
接下来开始配置 USART1：



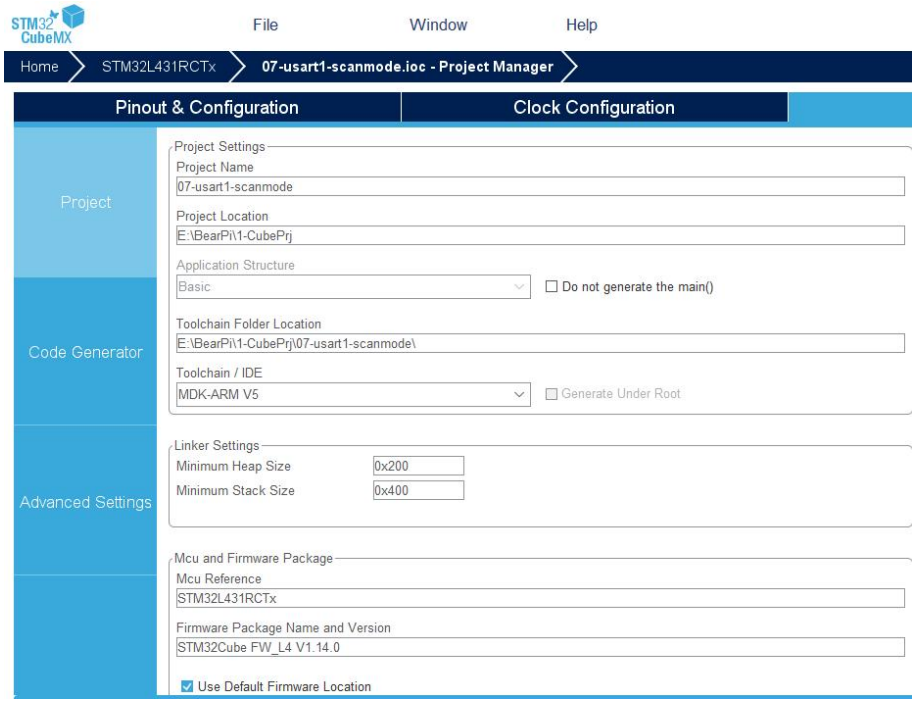
配置时钟树

STM32L4 的最高主频到 80M，所以配置 PLL，最后使 HCLK = 80Mhz 即可：



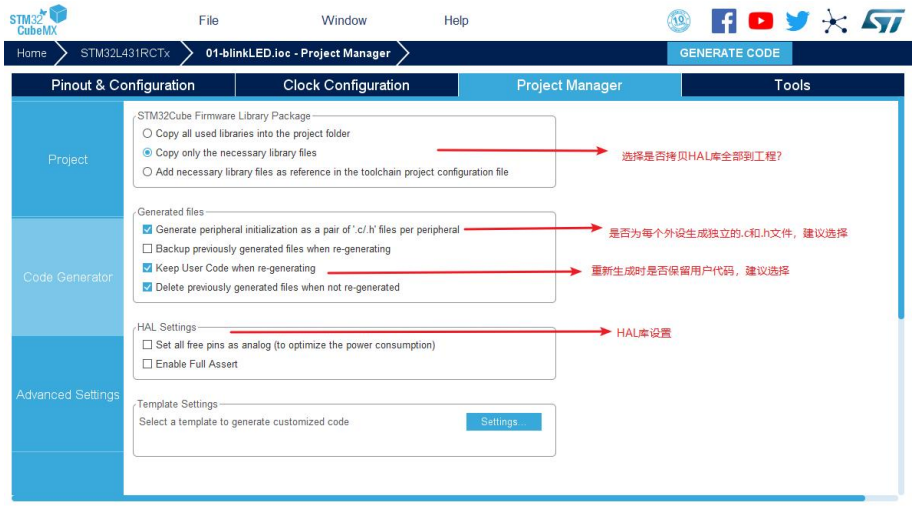


生成工程设置



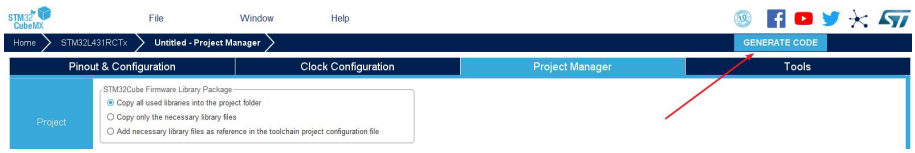
代码生成设置

最后设置生成独立的初始化文件：



生成代码

点击 GENERATE CODE 即可生成 MDK-V5 工程：



3. 在 MDK 中编写、编译、下载用户代码

编写查询模式发送和接收代码

编写 `main` 函数如下：

```
int main(void){
  /* USER CODE BEGIN 1 */
          char str[12] = "Hello World\n";
          char recv_buf[12] = {0};
  /* USER CODE END 1 */

  HAL_Init();

  SystemClock_Config();

  MX_GPIO_Init();
  MX_USART1_UART_Init();

  /* USER CODE BEGIN 2 */
          HAL_UART_Transmit(&huart1, (uint8_t*)str, 12, 0xFFFF);
  /* USER CODE END 2 */

  while (1)
  {
    /* USER CODE END WHILE */
    /* USER CODE BEGIN 3 */
    //接收 12 个字节的数据，不超时
    if(HAL_OK == HAL_UART_Receive(&huart1, (uint8_t*)recv_buf, 12, 0xFFFF))
    {
```

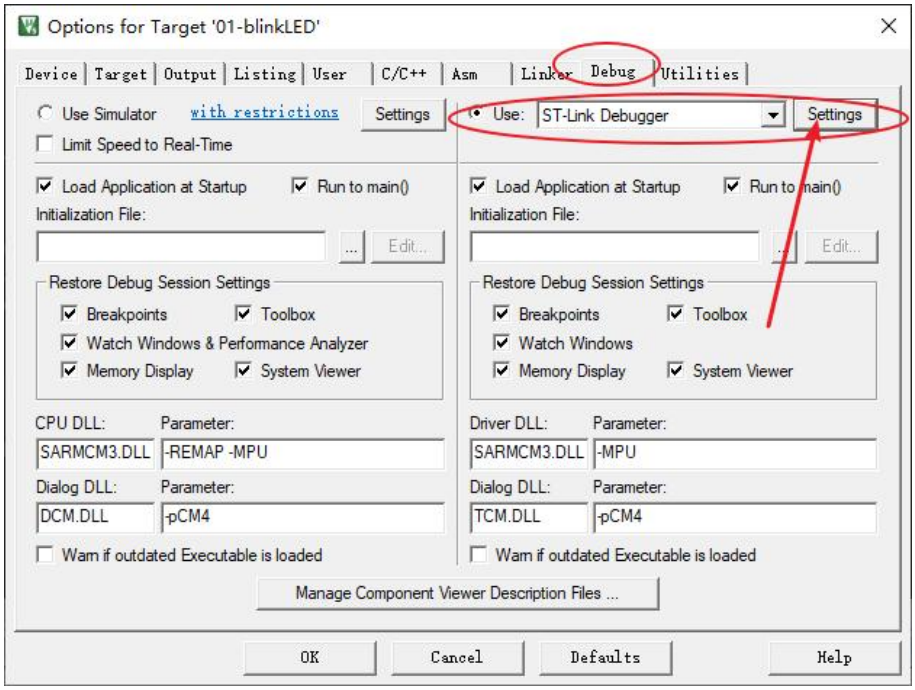
```
    //将接收到的数据发送
    HAL_UART_Transmit(&huart1, (uint8_t*)recv_buf, 12, 0xFFFF);
  }
}
/* USER CODE END 3 */
```

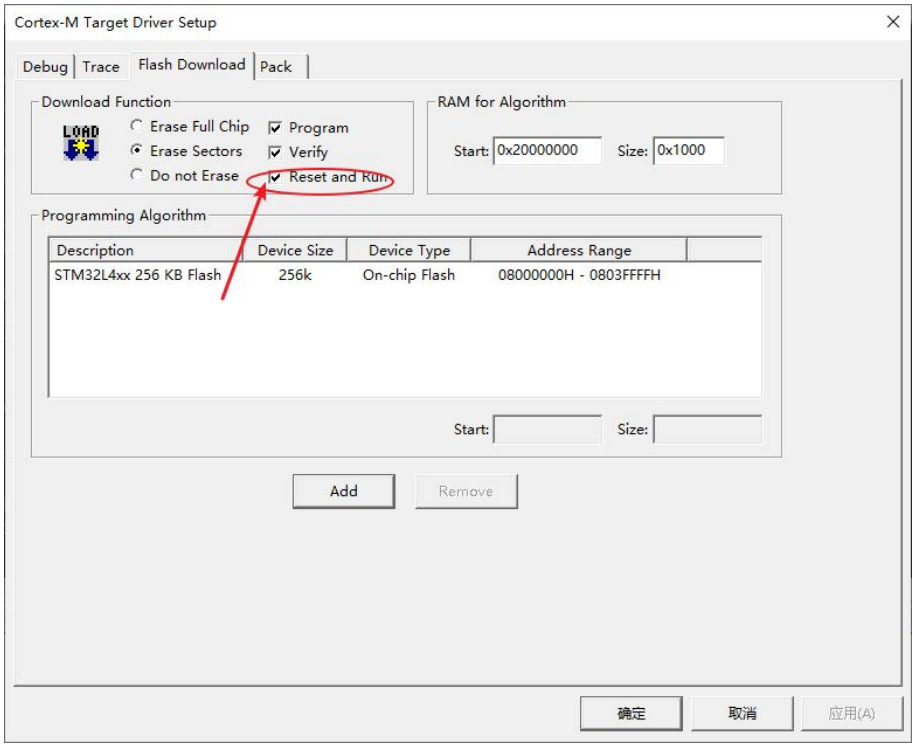
编译代码

编译整个工程：

```
Build Output
compiling stm32l4xx_hal_pwr_ex.c...
compiling stm32l4xx_hal_cortex.c...
compiling stm32l4xx_hal_exti.c...
compiling system_stm32l4xx.c...
linking...
Program Size: Code=3388 RO-data=492 RW-data=16 ZI-data=1024
FromELF: creating hex file...
"06-key_it_mode\06-key_it_mode.axf" - 0 Error(s), 0 Warning(s).
Build Time Elapsed:  00:00:16
```

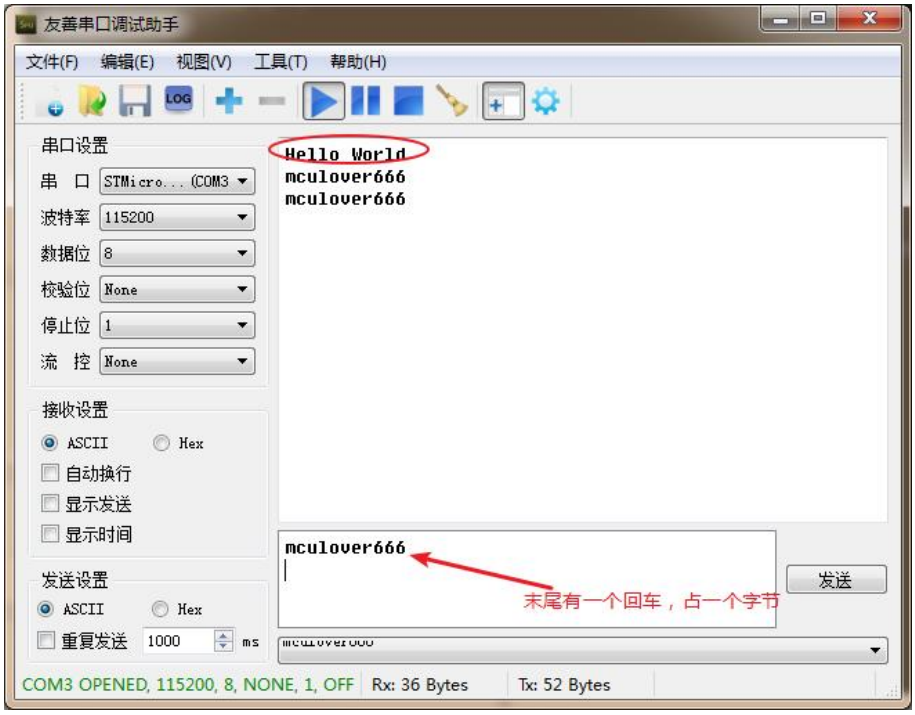
设置下载器





实验现象

下载运行后，实验现象如下：



至此，我们已经学会了**如何配置 USART 使用查询模式发送和接收数据**，下一节将讲述如何配置 USART 使用中断模式接收数据。

作业：编写程序，使用 USART 在 STM32 单片机和 PC 之间通过查询模式发送和接收数据。
调试并优化 USART 通信的稳定性和效率。

STM32 单片机基础 07——使用 USART 发送和接收数据(中断模式)

教学目的与要求：

目的：掌握 USART 的中断模式，实现数据的异步发送和接收，进一步提高系统的响应速度和效率。

要求：能够配置 USART 中断，编写中断服务程序处理数据发送和接收事件，理解中断的优先级和嵌套。

教学重难点：

重点：USART 中断的配置和中断服务程序的编写。

难点：确保中断处理的正确性和高效性，避免中断冲突和丢失数据。

课时数：3 课时

思政元素：

强调系统设计的灵活性和高效性，通过 USART 中断模式的学习，让学生理解在嵌入式系统设计中，根据具体需求选择合适的数据传输模式，提高系统的整体性能和用户体验。

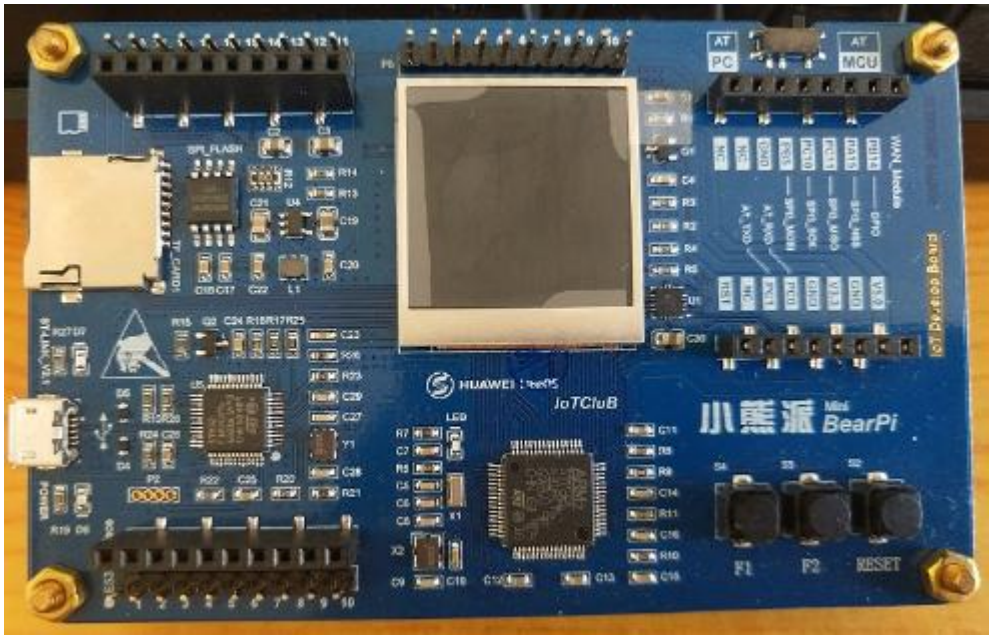
本篇文章主要介绍如何使用 STM32CubeMX 初始化 STM32L431RCT6 的 USART，并使用**中断模式**发送和接收数据。

1. 准备工作

硬件准备

开发板

首先需要准备一个开发板，这里我准备的是 STM32L4 的开发板（BearPi）：



软件准备

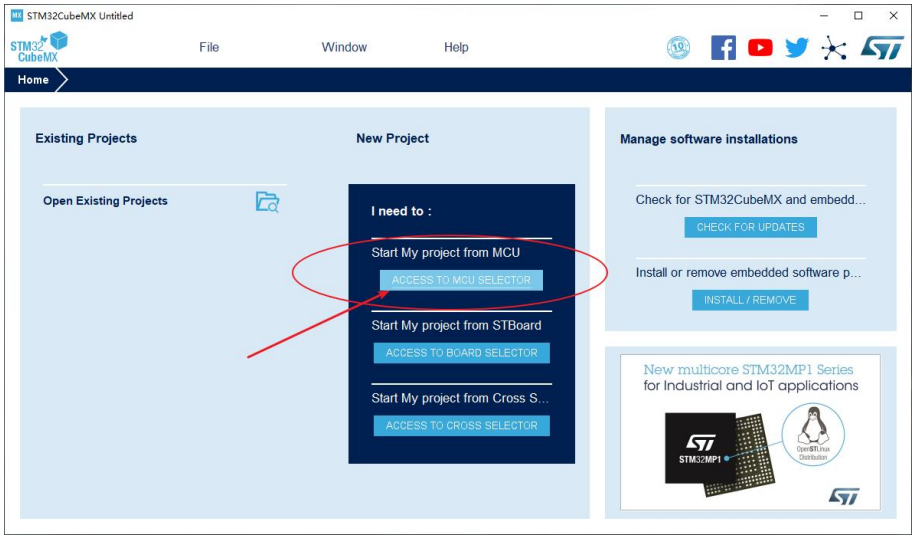
- 需要安装好 Keil - MDK 及芯片对应的包，以便编译和下载生成的代码；

Keil MDK 和串口助手的安装包都可以关注“**小熊派开源社区**”微信公众号，在资料教程一栏中可获取安装包。

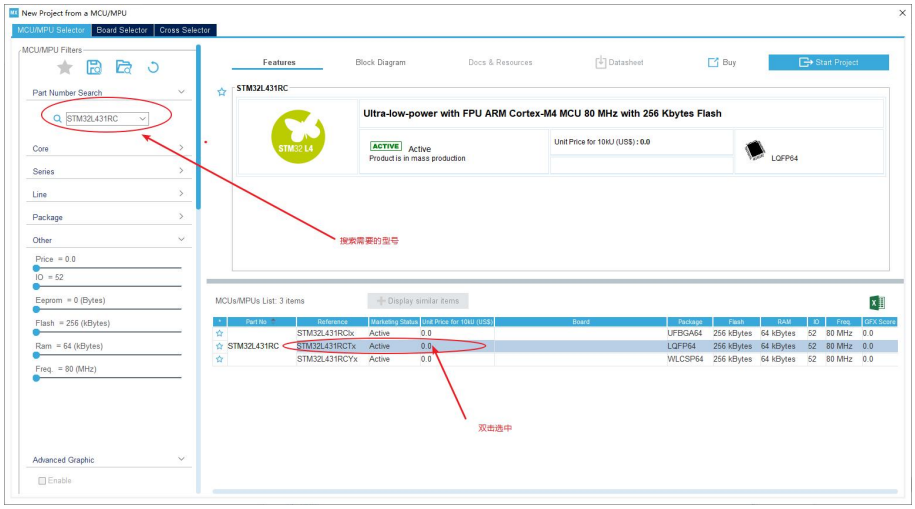
2. 生成 MDK 工程

选择芯片型号

打开 STM32CubeMX，打开 MCU 选择器：



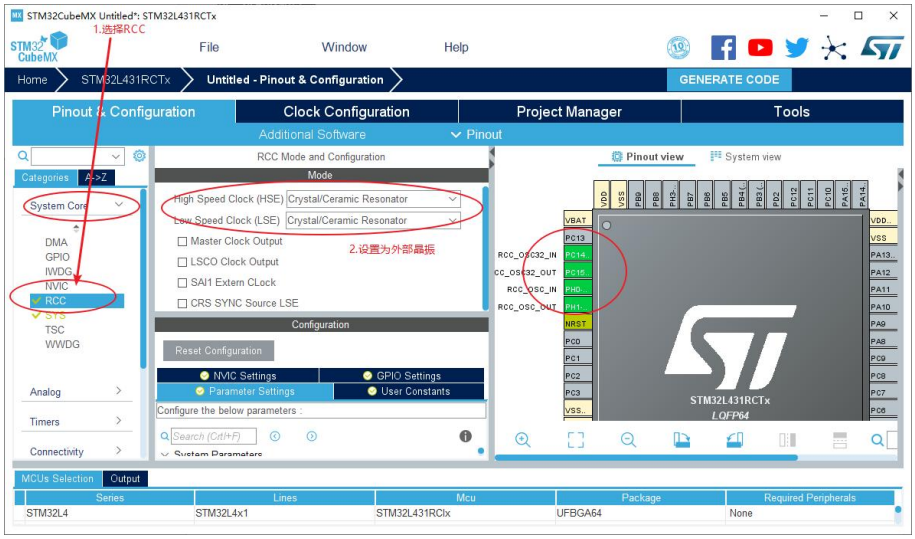
搜索并选中芯片 STM32L431RCT6:



配置时钟源

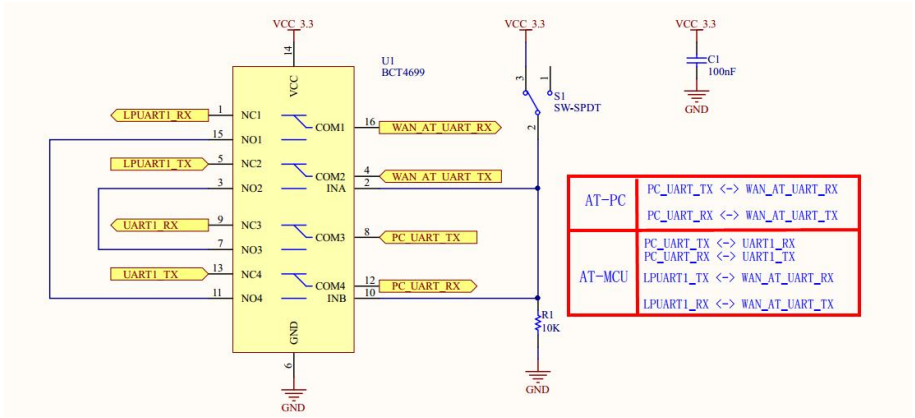
- 如果选择使用外部高速时钟（HSE），则需要在 System Core 中配置 RCC；
- 如果使用默认内部时钟（HSI），这一步可以略过；

这里我都使用外部时钟：



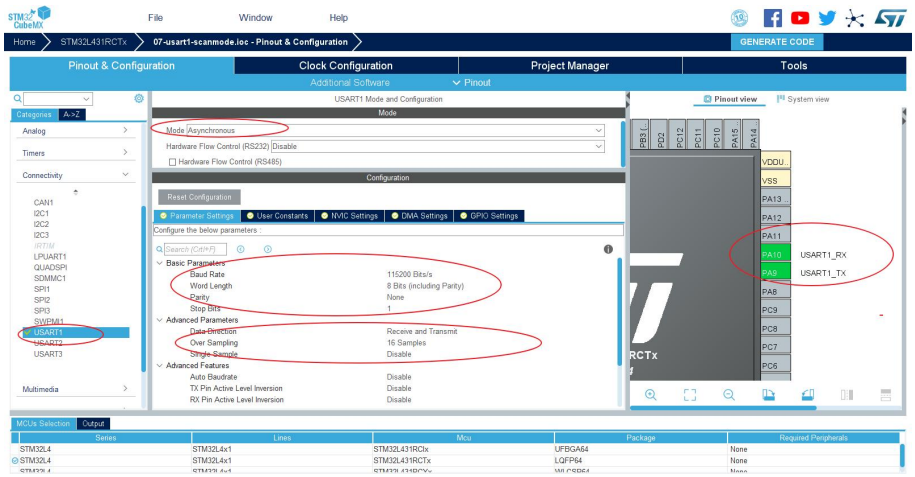
配置串口

小熊派开发板板载 ST-Link 并且虚拟了一个串口，原理图如下：



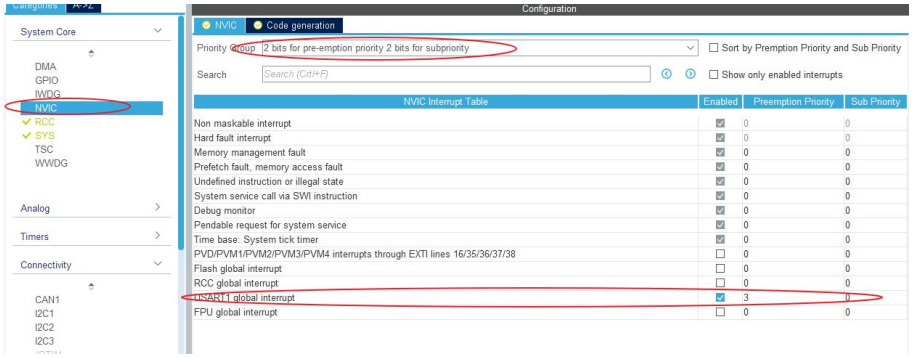
这里我将开关拨到 AT-MCU 模式，使 PC 的串口与 USART1 之间连接。

接下来开始配置 USART1：



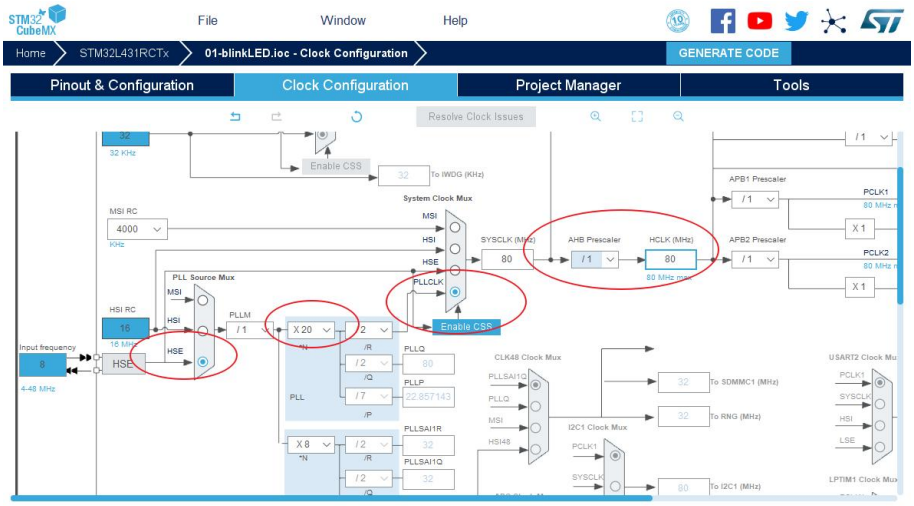
NVIC 配置

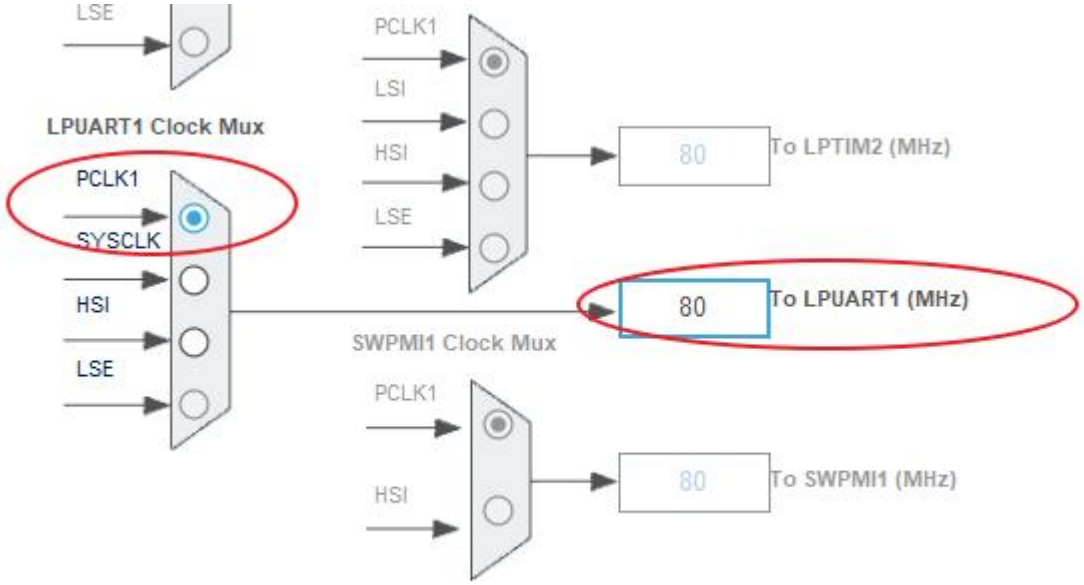
在 NVIC 中配置 USART 中断优先级：



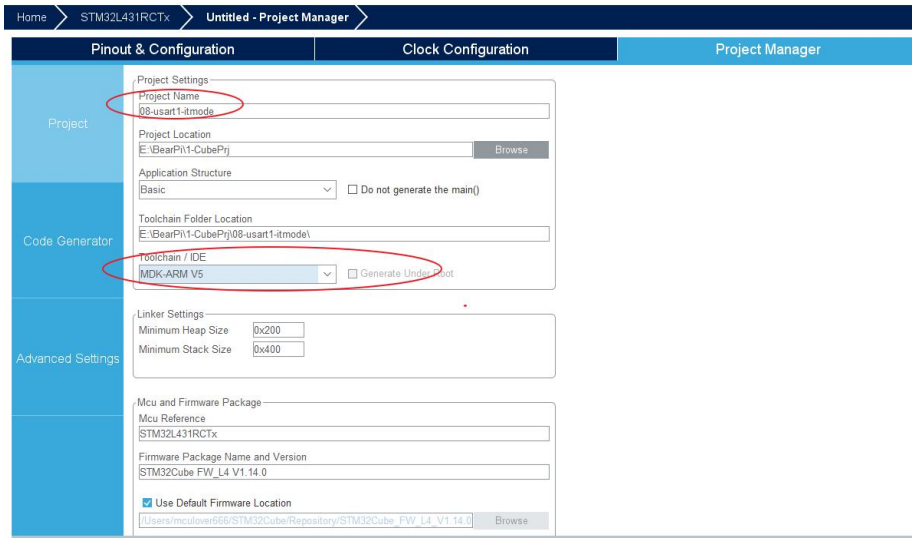
配置时钟树

STM32L4 的最高主频到 80M，所以配置 PLL，最后使 `HCLK = 80Mhz` 即可：



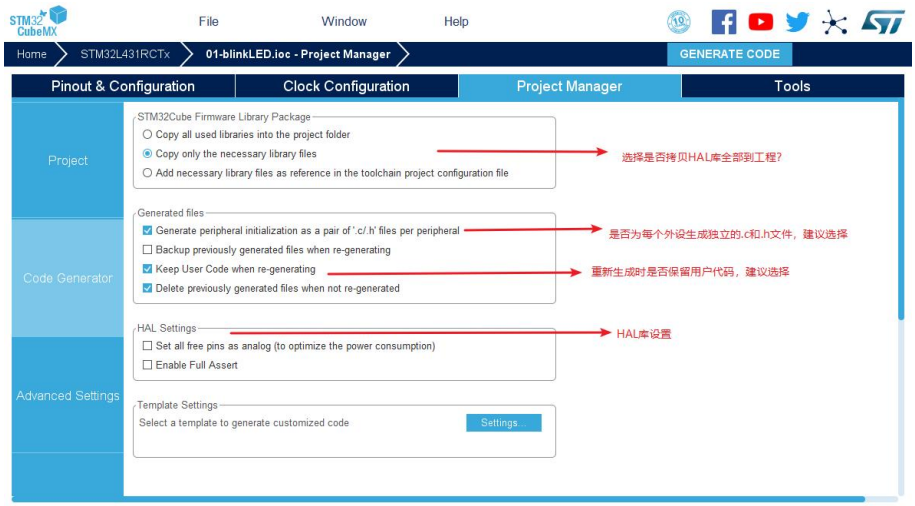


生成工程设置



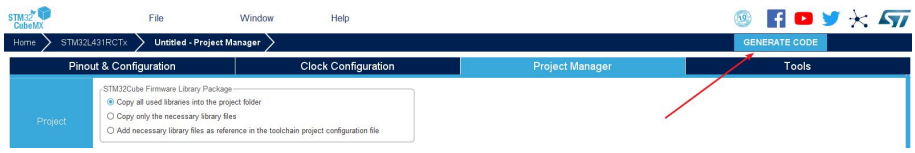
代码生成设置

最后设置生成独立的初始化文件：



生成代码

点击 GENERATE CODE 即可生成 MDK-V5 工程：



3. 在 MDK 中编写、编译、下载用户代码

定义发送和接收缓冲区

```
/* Private user code ---------------------------------------------------------*//* USER CODE BEGIN 0 */
```

```
uint8_t hello[] = "USART1 is ready...\n";

uint8_t recv_buf[13] = {0};/* USER CODE END 0 */
```

重新实现中断回调函数

在 NVIC 一讲中我们探索了 HAL 库的中断处理机制，HAL 中弱定义了一个中断回调函数 `HAL_UART_RxCpltCallback`，我们需要在用户文件中重新定义该函数，放在哪都可以，这里我放在 `main.c` 中：

```
/* USER CODE BEGIN 4 *//* 中断回调函数 */void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart){

        /* 判断是哪个串口触发的中断 */

        if(huart ->Instance == USART1)

        {

                //将接收到的数据发送

                HAL_UART_Transmit_IT(huart, (uint8_t*)recv_buf, 13);

                //重新使能串口接收中断

                HAL_UART_Receive_IT(huart, (uint8_t*)recv_buf, 13);

        }}/* USER CODE END 4 */
```

修改 main 函数

在 main 函数中首先开启串口中断接收，然后发送提示信息：

```
int main(void){

  HAL_Init();


  SystemClock_Config();


  MX_GPIO_Init();

  MX_USART1_UART_Init();


  /* USER CODE BEGIN 2 */
```

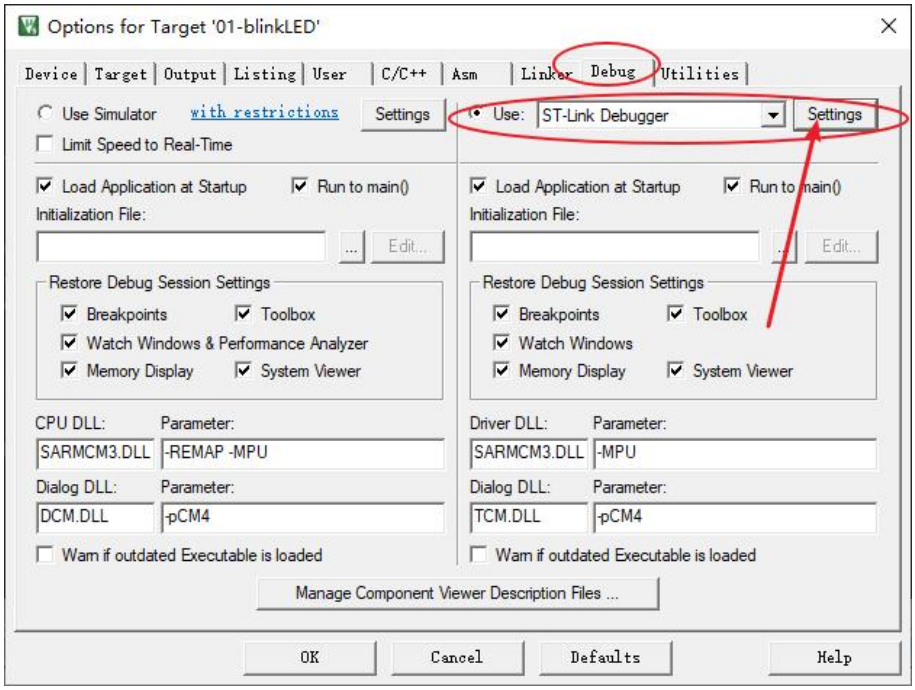
```
  //使能串口中断接收
  HAL_UART_Receive_IT(&huart1, (uint8_t*)recv_buf, 13);
  //发送提示信息
  HAL_UART_Transmit_IT(&huart1, (uint8_t*)hello, sizeof(hello));
  /* USER CODE END 2 */


  while (1)
  {
  }}
```

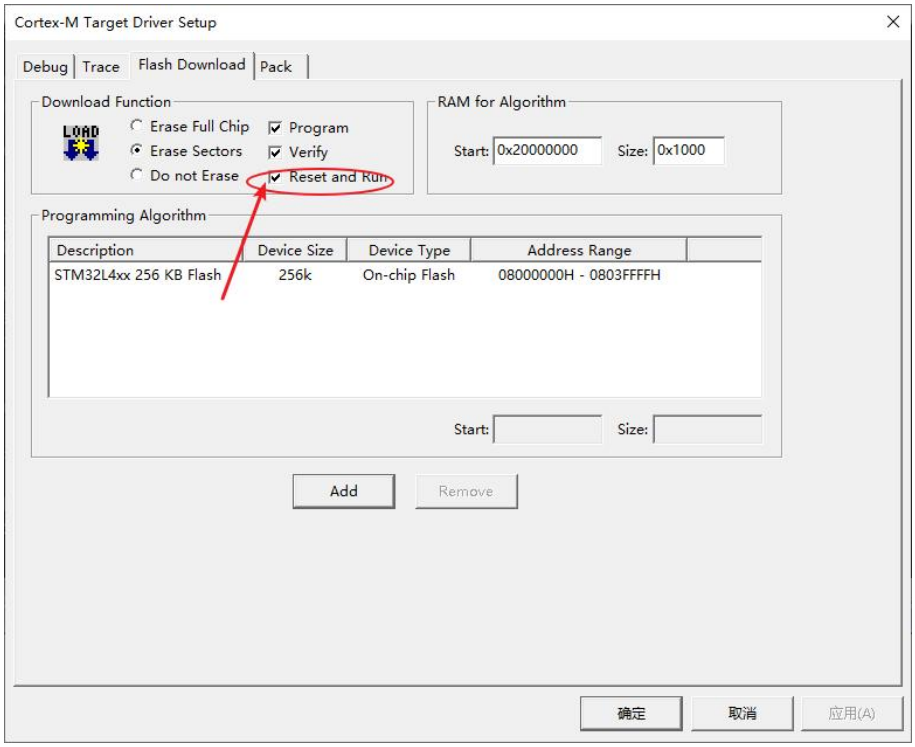
编译代码

编译整个工程：

```
Build Output
Build target '08-usart1-itmode'
compiling main.c...
linking...
Program Size: Code=6788 RO-data=500 RW-data=36 ZI-data=1172
FromELF: creating hex file...
"08-usart1-itmode\08-usart1-itmode.axf" - 0 Error(s), 0 Warning(s).
Build Time Elapsed:  00:00:01
```

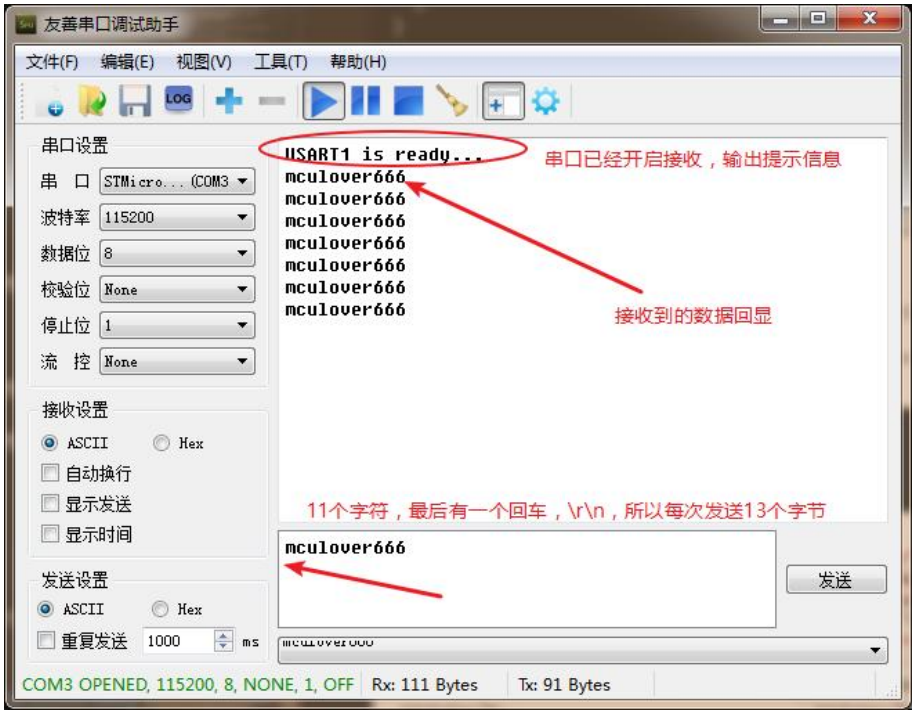
设置下载器





实验现象

下载运行后，实验现象如下：



至此，我们已经学会了**如何配置 USART 使用中断模式发送和接收数据**，下一节将讨论实现 printf()函数的多种方法。

作业：配置 USART 中断，编写程序在中断模式下实现数据的发送和接收。
比较查询模式与中断模式在 USART 通信中的差异和适用场景。

STM32 单片机基础 08——使用 USART 发送和接收数据(DMA 模式)

教学目的与要求：

目的：掌握 USART 的 DMA 模式，实现高速、高效的数据传输，减少 CPU 的干预。

要求：能够配置 DMA 和 USART 以支持 DMA 模式，理解 DMA 的工作原理，编写程序实现数据的自动传输。

教学重难点：

重点：DMA 的配置和 USART 与 DMA 的联合使用。

难点：理解 DMA 传输的完整流程，确保数据传输的正确性和稳定性。

课时书：3 课时

思政元素：

培养学生的系统整合能力，通过 USART DMA 模式的学习，让学生理解在复杂系统中，各个模块之间的协同工作对于实现高效数据传输的重要性，培养学生的全局观念和协作精神。

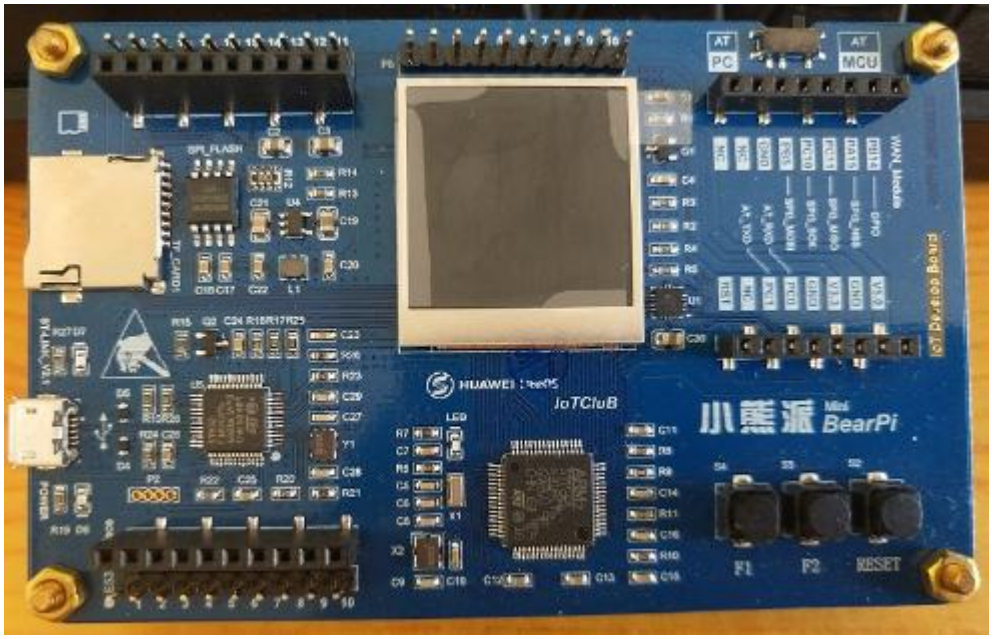
本篇文章主要介绍如何使用 STM32CubeMX 初始化 STM32L431RCT6 的 USART，并使用 **DMA 模式**发送数据和接收数据。

1. 准备工作

硬件准备

开发板

首先需要准备一个开发板，这里我准备的是 STM32L4 的开发板（BearPi）：



软件准备

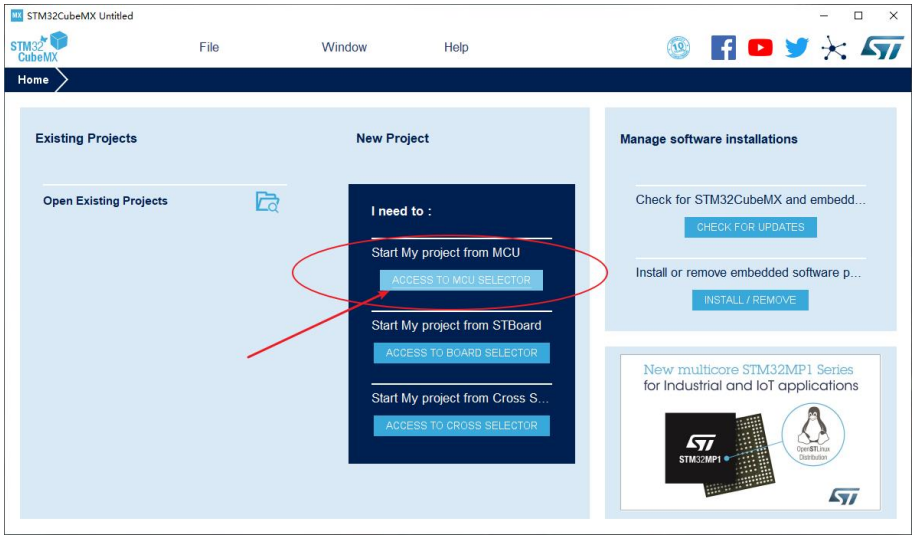
- 需要安装好 Keil - MDK 及芯片对应的包，以便编译和下载生成的代码；

Keil MDK 和串口助手的安装包都可以关注“**小熊派开源社区**”微信公众号，在资料教程一栏中可获取安装包。

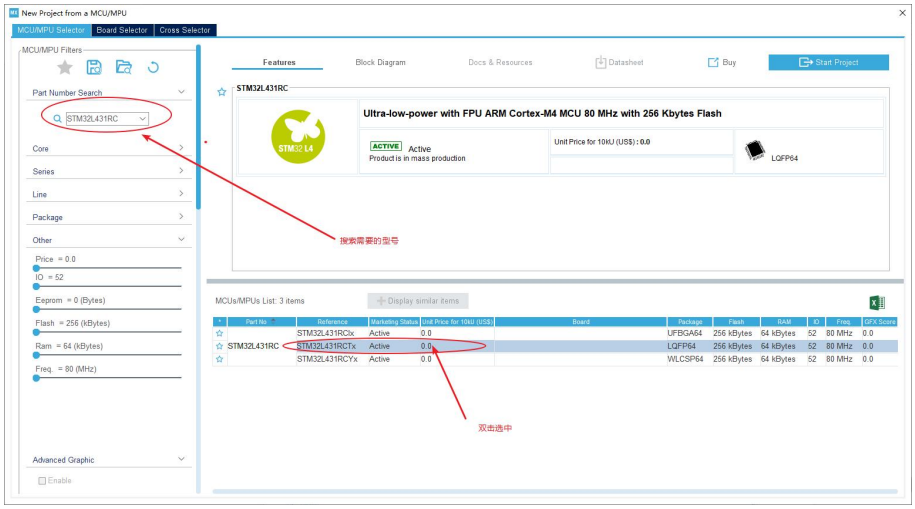
2. 生成 MDK 工程

选择芯片型号

打开 STM32CubeMX，打开 MCU 选择器：



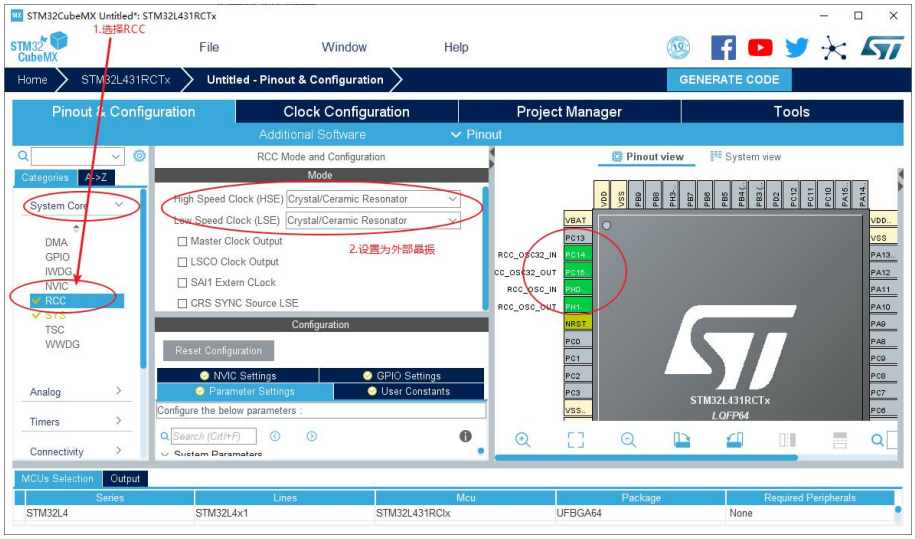
搜索并选中芯片 STM32L431RCT6:



配置时钟源

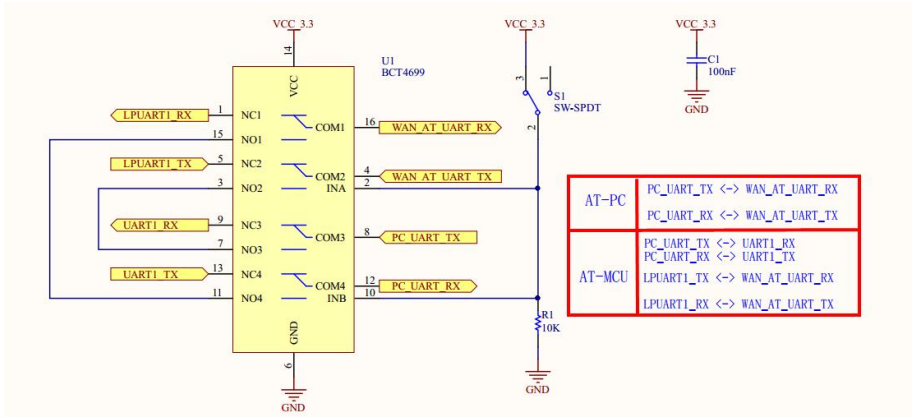
- 如果选择使用外部高速时钟（HSE），则需要在 System Core 中配置 RCC；
- 如果使用默认内部时钟（HSI），这一步可以略过；

这里我都使用外部时钟：



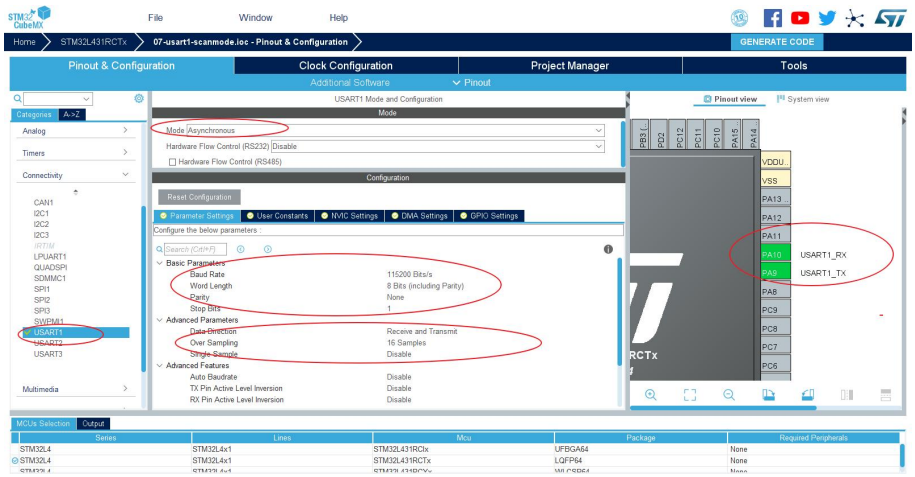
配置串口

小熊派开发板板载 ST-Link 并且虚拟了一个串口，原理图如下：



这里我将开关拨到 AT-MCU 模式，使 PC 的串口与 USART1 之间连接。

接下来开始配置 USART1：



USART DMA 配置

知识小卡片 —— DMA

DMA 全称 Direct Memory Access(直接存储器访问)， 是 STM32 的一个外设，它的特点在于：

在**不占用 CPU 的情况下**将数据从存储器直接搬运到外设，或者从外设直接搬运到存储器，当然也可以从存储器直接搬运到存储器。

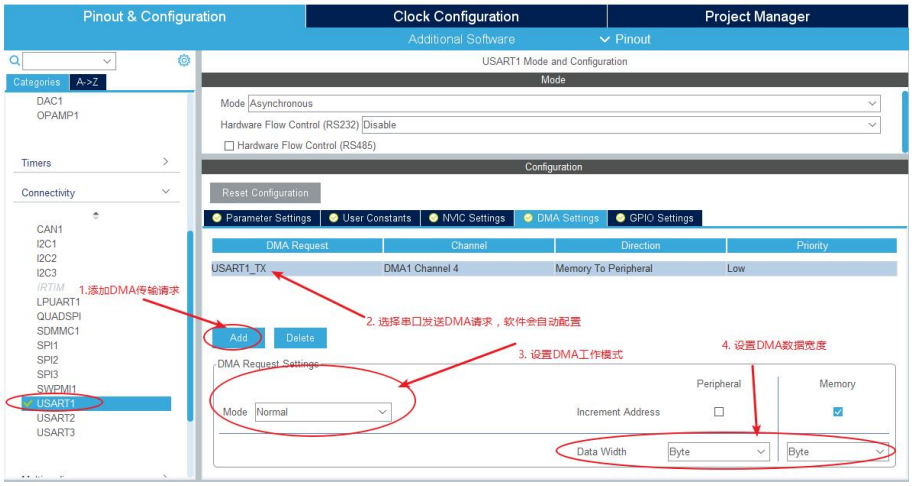
比如在需要串口发送大量数据的时候，CPU 只需要**发起 DMA 传输请求**，然后就可以去做别的事情了，DMA 会将数据传输到串口发送，**DMA 传输完之后会触发中断**，CPU 如果有需要，可以对该中断进行处理，这样一来 CPU 的效率是不是大大提高了？

在 STM32L431RCT6 中有 2 个 DMA 外设：DMA1 和 DMA2，每个 DMA 外设有 7 个通道，每个通道都是独立的，配置 DMA 的时候有几个关键点：

- 数据从哪里来？
- 数据到哪里去？
- 有多少数据？

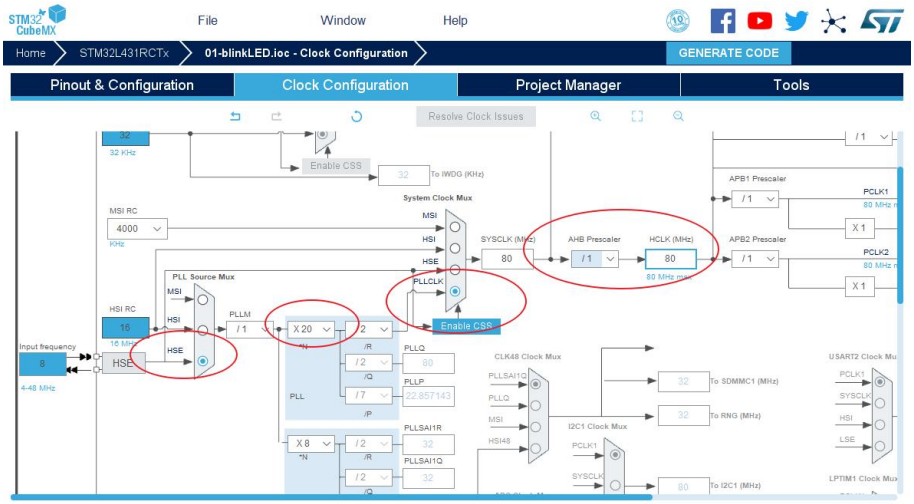
知识小卡片结束啦~对 STM32 的 DMA 外设有没有了解呢？

接下来我们配置 DMA，将存储器（SRAM）中的数据直接搬运到串口外设去发送：

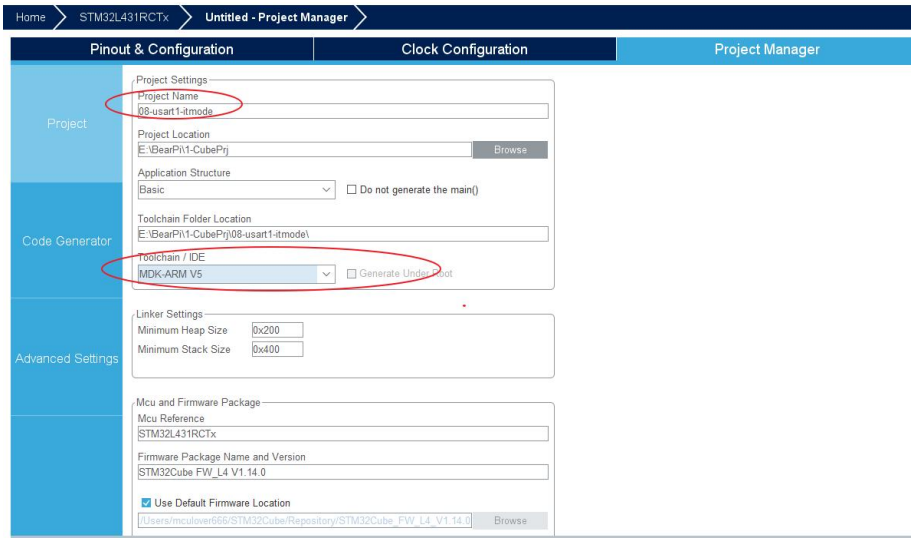


配置时钟树

STM32L4 的最高主频到 80M，所以配置 PLL，最后使 HCLK = 80Mhz 即可：

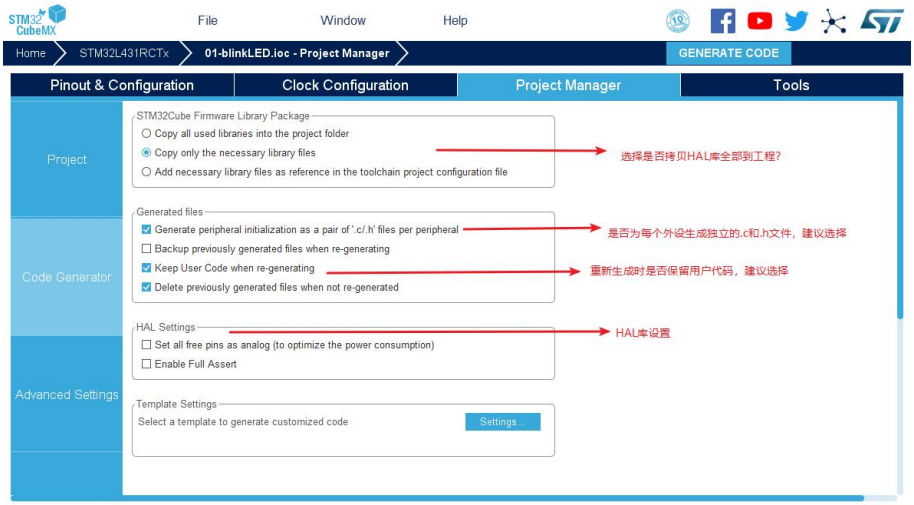


生成工程设置



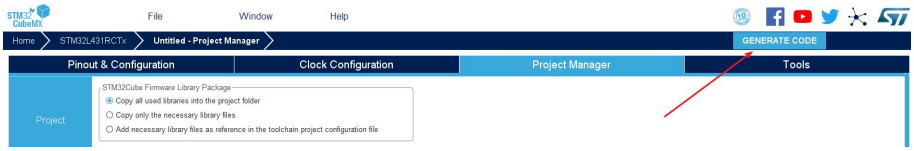
代码生成设置

最后设置生成独立的初始化文件：



生成代码

点击 GENERATE CODE 即可生成 MDK-V5 工程：



3. 在 MDK 中编写、编译、下载用户代码

定义发送数据区域

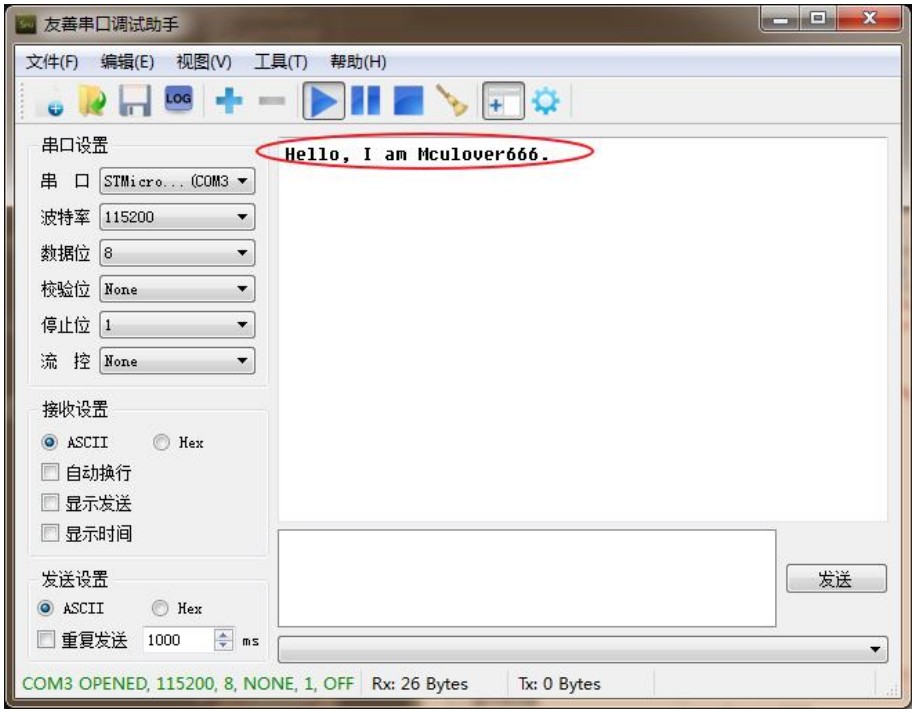
```
/* Private user code ---------------------------------------------------------*//* USER CODE BEGIN 0 */

uint8_t dat[] = "Hello, I am Mculover666.\n";/* USER CODE END 0 */
```

在 main 函数中发起 DMA 传输

```
int main(void){

  HAL_Init();


  SystemClock_Config();


  MX_GPIO_Init();
  MX_USART1_UART_Init();


  /* USER CODE BEGIN 2 */
  HAL_UART_Transmit_DMA(&huart1, (uint8_t*)dat, sizeof(dat));
  /* USER CODE END 2 */


  while (1)
  {
  }}
```

实验现象

编译下载运行后，实验现象如下：

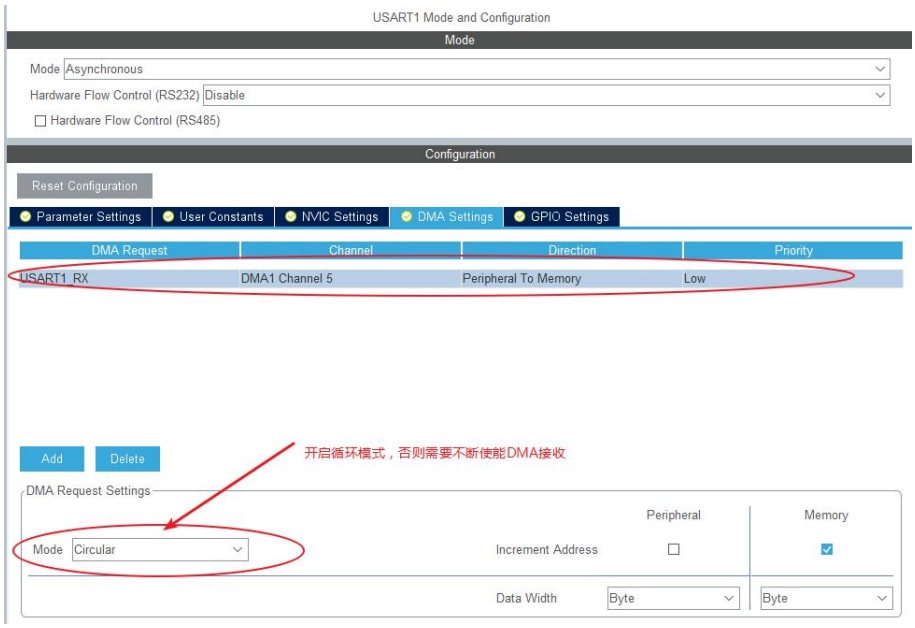


4. 使用 DMA 接收串口数据

说明

- 使用 HAL 库的时候不能同时使用 DMA 发送和接收数据，会出错。
- 所有的步骤和发送时一样，这里我只给出需要修改的部分。

修改串口 DMA 配置



添加串口接收缓冲区

```
/* Private user code ---------------------------------------------------------*//* USER CODE BEGIN 0 */

uint8_t dat[] = "Hello, I am Mculover666.\n";

uint8_t recv_buf[13] = {0};             //串口接收缓冲区/* USER CODE END 0 */
```

修改 main 函数

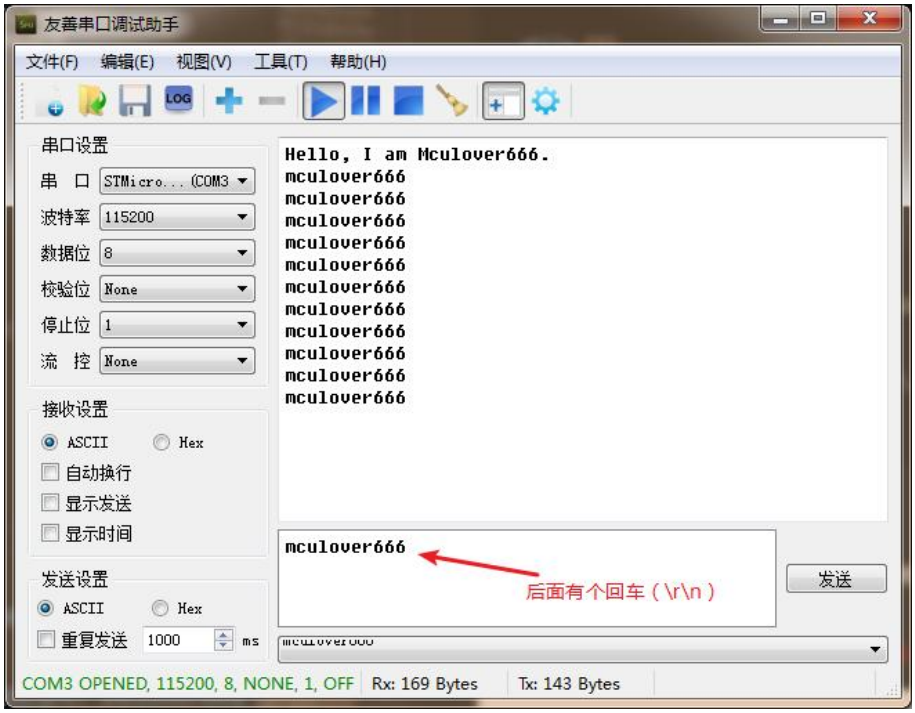
```
int main(void){

  HAL_Init();



  SystemClock_Config();



  MX_GPIO_Init();

  MX_DMA_Init();

  MX_USART1_UART_Init();



  /* USER CODE BEGIN 2 */

  HAL_UART_Transmit(&huart1, (uint8_t*)dat, sizeof(dat), 0xFFFF);

  HAL_UART_Receive_DMA(&huart1, recv_buf, 13);  //使能 DMA 接收

  /* USER CODE END 2 */



  while (1)

  {

  }}
```

添加串口接收中断回调函数

```
/* USER CODE BEGIN 4 */void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart) {

          //将接收到的数据再发送
```

```
        HAL_UART_Transmit(&huart1,recv_buf,13, 0xFFFF);}/* USER CODE END 4 */
```

实验现象



至此，我们已经学会了**如何配置 USART 使用 DMA 模式发送数据和接收数据**，下一节将讨论实现 printf()函数的多种方法。

作业：使用 DMA（直接存储器访问）方式优化 USART 的数据传输效率。
编写程序实现大量数据的快速发送和接收，并验证 DMA 的效果。

STM32 单片机基础 09——重定向 printf 函数到串口输出的多种方法

教学目的与要求：

目的：掌握将 C 标准库中的 printf 函数输出重定向到 USART 的方法，方便调试和日志记录。

要求：能够实现 printf 函数输出的重定向，并理解其背后的原理。

教学重难点：

重点：printf 函数重定向的实现方法。

难点：理解标准 I/O 库的工作原理，以及如何在嵌入式系统中进行重定向。

课时数：3 课时

思政元素：

培养学生的创新精神和问题解决能力，通过 printf 函数重定向的实践，让学生理解在资源受限的嵌入式系统中，如何通过创新方法解决调试和日志记录的问题，培养学生的灵活性和创新思维。

本文详细的介绍了如何重定向 printf 输出到串口输出的多种方法，包括调用 MDK 微库（MicroLib）的方法，调用标准库的方法，以及适用于 `GNUC` 系列编译器的方法。

1.printf 与 fputc

对于 printf 函数相信大家都不陌生，第一个 C 语言程序就是使用 printf 函数在屏幕上的控制台打印出 `Hello World`，之后使用 printf 函数输出各种类型的数据，使用格式控制输出各种长度的字符，甚至输出各种各样的图案。

除此之外，在程序出错的时候，懒得调试，直接简单粗暴的加个 printf 找 bug，有时候也不失为一种有效的方法。

对于已经习惯的 printf 函数，你了解多少呢？

printf 定义在 `<stdio.h>` 头文件中，如下：

```
int printf(const char *format, ...);
```

printf 函数根据 `format` 字符串给出的格式打印输出到 `stdout`（标准输出）中，当然，**printf 函数是不会一个字符一个字符去输出，它会调用更底层的 I/O 函数：`fputc` 去逐个字符打印**。

fputc 也定义于头文件 `<stdio.h>`中，如下：

```
int fputc(int ch, FILE *stream);
```

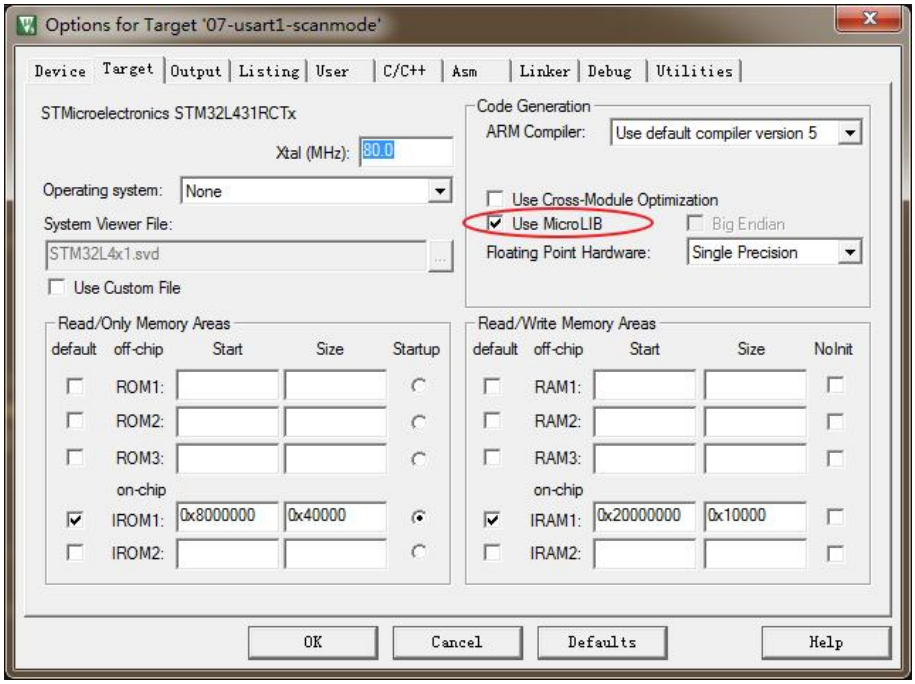
fputc 函数写入字符 ch 到给定输出流 stream,printf 函数在调用该函数时,会向 stream 参数传入 `stdout` 从而打印数据到标准输出。

那么，要实现 printf 打印到串口就变得非常简单了，**只需要重新定义 fputc 函数，在 fputc 的函数中将数据通过串口发送，称之为：fputc 重定向或者 printf 重定向。**

2.在 MDK 中使用 MicroLib 重定向 printf

勾选 Use MicroLib

MicroLib 是对标准 C 库进行了高度优化之后的库，供 MDK 默认使用，相比之下，MicroLIB 的代码更少，资源占用更少：



重定义 fputc 到串口

重新实现 fputc 函数，编写代码将这个字符通过串口发送，因为发送每个字符时都会调用该函数，所以**为了效率**，不再调用库函数 `HAL_UART_Transmit` 发送，而是直接操作寄存器发送。

检测串口当前状态

STM32L431 的 USART 串口外设有一个 `ISR` 寄存器，全名 `Interrupt and status register`，用来指示当前串口的状态，如图：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	TCBGT	Res.	Res.	REACK	TEACK	WUF	RWU	SBKF	CMF	BUSY
						r			r	r	r	r	r	r	r
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ABRF	ABRE	Res.	EOBF	RTOF	CTS	CTSIF	LBDF	TXE	TC	RXNE	IDLE	ORE	NF	FE	PE
r	r		r	r	r	r	r	r	r	r	r	r	r	r	r

其中 `BIT6 TC` 用来指示当前串口是否发送完成，如图：

Bit 6 **TC**: Transmission complete

This bit is set by hardware if the transmission of a frame containing data is complete and if TXE is set. An interrupt is generated if TCIE=1 in the USART_CR1 register. It is cleared by software, writing 1 to the TCCF in the USART_ICR register or by a write to the USART_TDR register.

An interrupt is generated if TCIE=1 in the USART_CR1 register.

0: Transmission is not complete

1: Transmission is complete

Note: If TE bit is reset and no transmission is on going, the TC bit will be set immediately.

可以通过判断该位来判断串口当前是否处于发送状态，代码如下：

```
while((USART1->ISR & 0X40) == 0);
```

- 串口发送字符 ch

同样，为了提高发送效率，直接使用寄存器来操作：

```
USART1->TDR = (uint8_t) ch;
```

最后实现 fputc 函数就变的非常简单了，这里我放在 usart.c 文件的末尾：

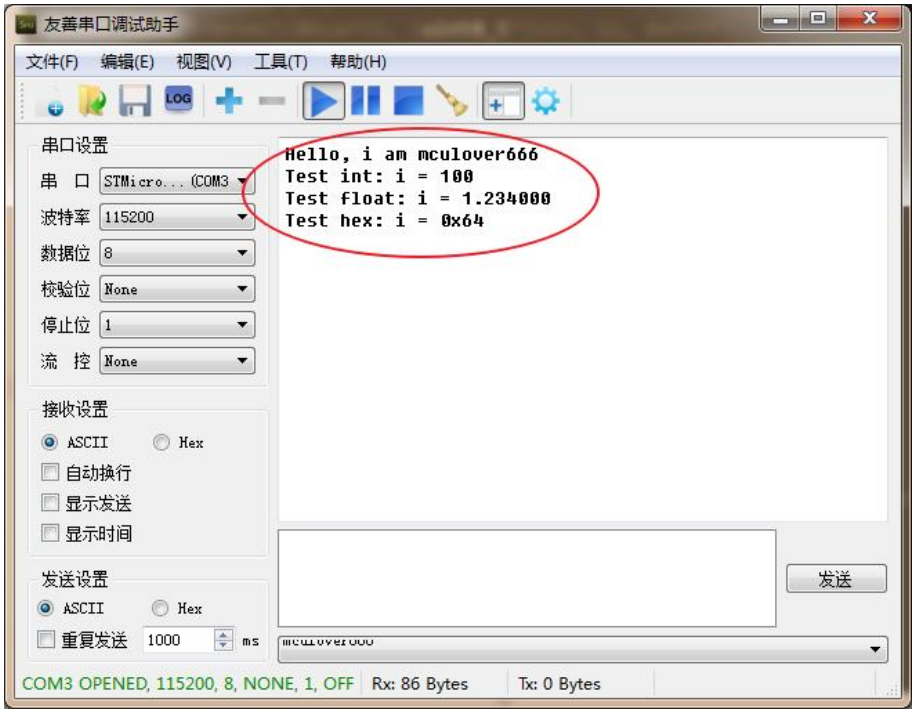
```
/* USER CODE BEGIN 1 */#if 1#include <stdio.h>

int fputc(int ch, FILE *stream){

    /* 堵塞判断串口是否发送完成 */

    while((USART1->ISR & 0X40) == 0);



    /* 串口发送完成，将该字符发送 */

    USART1->TDR = (uint8_t) ch;



    return ch;}#endif

/* USER CODE END 1 */
```

测试 printf

在 main 函数中测试一下 printf 函数是否可以正常使用：

```
    /* USER CODE BEGIN 2 */

    printf("Hello, i am %s\n", "mculover666");

    printf("Test int: i = %d", 100);

    printf("Test float: i = %f", 1.234);

    printf("Test hex: i = 0x%2x",100);

    /* USER CODE END 2 */
```

结果如下：



3. 在 MDK 中使用标准库重定向 printf

printf 函数使用了半主机模式，所以直接使用标准库会导致程序无法运行，因此必须提前告知编译器不使用半主机模式：

- 不使用半主机模式

```
/* 告知连接器不从 C 库链接使用半主机的函数 */#pragma import(__use_no_semihosting)

/* 定义 _sys_exit() 以避免使用半主机模式 */void _sys_exit(int x){

    x = x;}
```

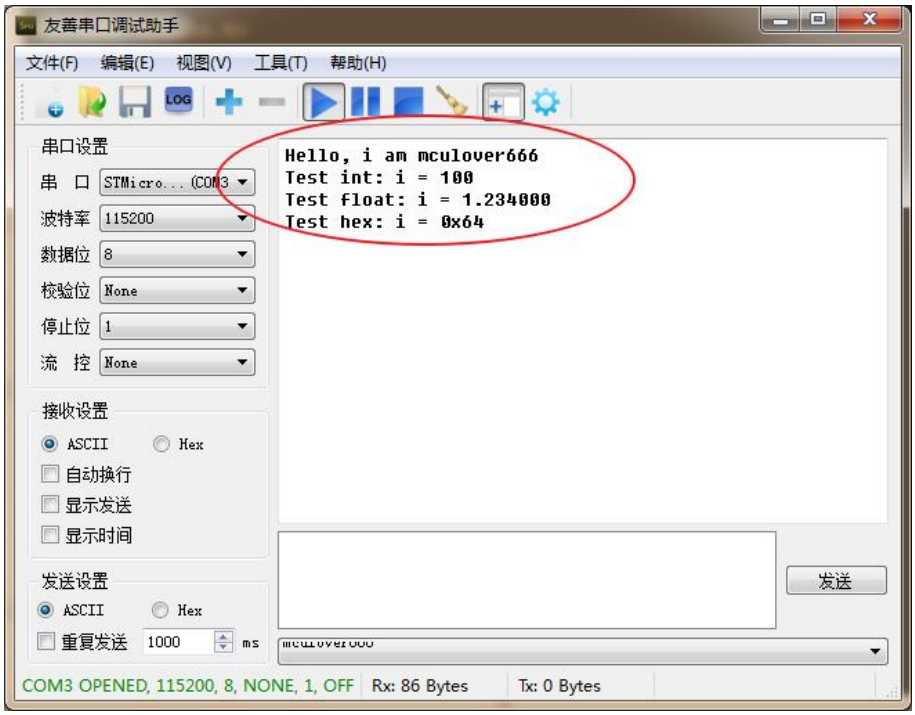
所以，重定向 fputc()函数完整的代码如下：

```
#if 1#include <stdio.h>

/* 告知连接器不从 C 库链接使用半主机的函数 */#pragma import(__use_no_semihosting)

/* 定义 _sys_exit() 以避免使用半主机模式 */void _sys_exit(int x){

    x = x;}

/* 标准库需要的支持类型 */struct __FILE{

    int handle;};



FILE __stdout;
```

```
/*  */int fputc(int ch, FILE *stream){

    /* 堵塞判断串口是否发送完成 */

    while((USART1->ISR & 0X40) == 0);


    /* 串口发送完成，将该字符发送 */

    USART1->TDR = (uint8_t) ch;


    return ch;}

#endif
```

测试 printf

测试 printf 函数的代码不变，在 MDK 设置中取消勾选 USE MICROLIB，然后重新编译，下载代码后试验现象如下：



4.在 GCC 中使用标准库重定向 printf

不同的编译器对于 C 库的底层实现机制是不同的，所以上面两种在 MDK 中的实现方法，在使用 Gcc 编译器的时候是不可行的。

在 Gcc 中重定向 printf 函数时注意两个关键点：

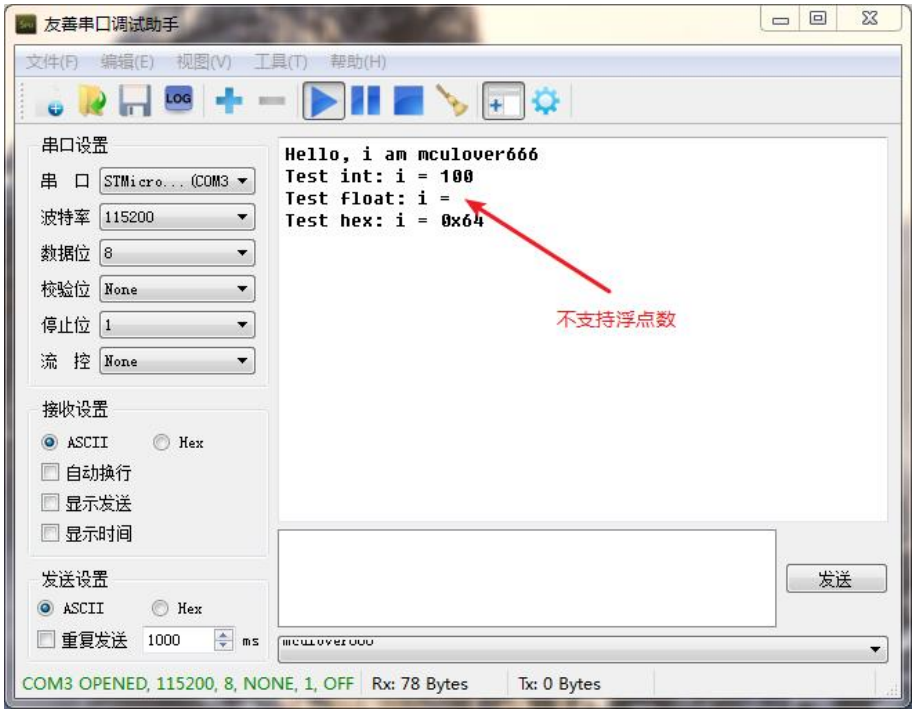
- 与重定义 fputs()函数一样，在使用 Gcc 编译器的时候，需要重新定义_write 函数；

- Gcc 中没有 MicroLib，只能使用标准库；

所以重定向 printf 函数的代码如下：

```
/* USER CODE BEGIN 1 */#if 1#include <stdio.h>

int _write(int fd, char *ptr, int len)  {

  HAL_UART_Transmit(&huart1, (uint8_t*)ptr, len, 0xFFFF);

  return len;}#endif/* USER CODE END 1 */
```

使用 STM32CubeMX 生成 makefile，然后使用 arm-none-eabi-gcc 编译没有问题，再使用 STM32 ST-LINK utility 下载后实验现象如下：



至此，我们已经学会**实现 printf()函数的多种方法**，下一节将讲述如何使用 ADC 读取 MQ-2 气体传感器的值。

加粗样式

作业：实现 printf 函数的串口重定向，并通过串口输出调试信息。
比较不同重定向方法的优缺点，并选择合适的实现方式。

STM32 单片机基础 10——使用 ADC 读取气体传感器数据（MQ-2）

教学目的与要求：

目的：掌握 ADC 的配置和使用方法，学会读取模拟传感器的数据，并转换为有用的数字信息。

要求：能够配置 ADC 以读取 MQ-2 气体传感器的输出，编写程序处理 ADC 数据，获取气体浓度信息。

教学重难点：

重点：ADC 的配置和数据的读取处理。

难点：理解模拟信号到数字信号的转换过程，以及如何进行数据的校准和滤波。

课时数：3 课时

思政元素：

培养学生的实践能力和科学精神，通过 ADC 读取气体传感器数据的实践，让学生理解在环境监测等应用中，准确获取和处理数据的重要性，培养学生的严谨性和科学素养。

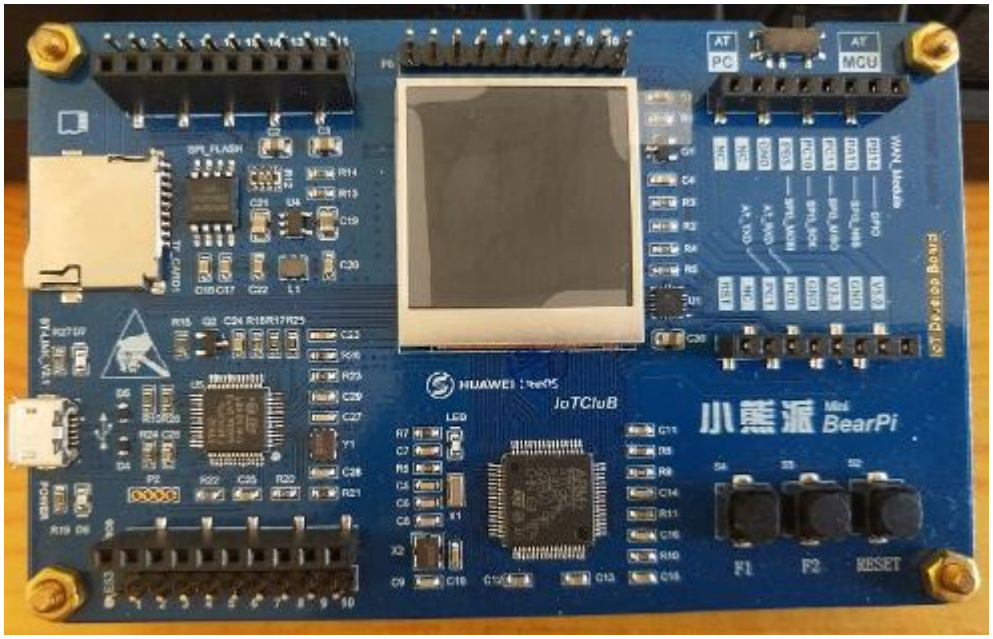
本篇详细的记录了如何使用 STM32CubeMX 配置 STM32L431RCT6 的 ADC 外设，读取 MQ-2 气体传感器的数据并通过串口发送。

1. 准备工作

硬件准备

开发板

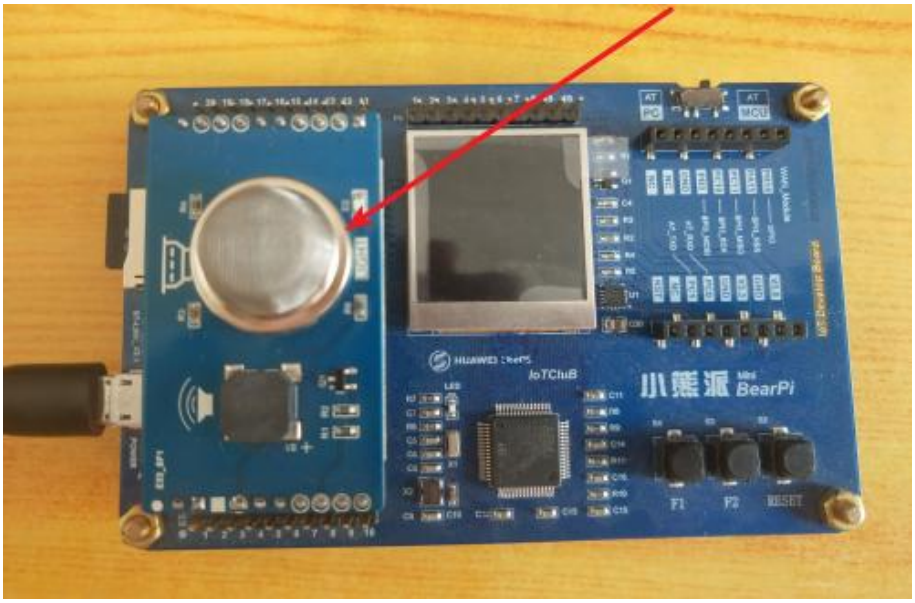
首先需要准备一个开发板，这里我准备的是 STM32L4 的开发板（BearPi）：



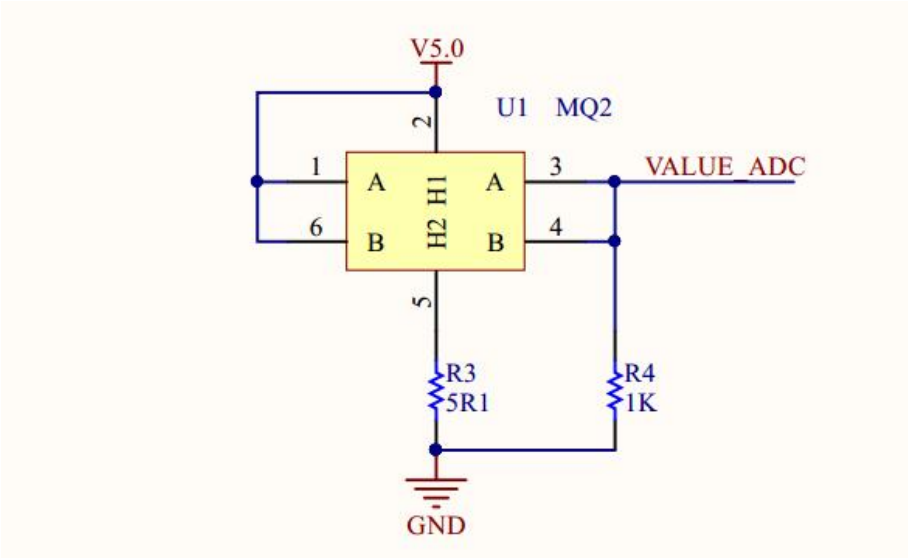
- MQ-2 模块

MQ-2 气体传感器一般用于家庭和工厂的气体泄漏监测装置，适用于液化气、丁烷、丙烷、甲烷、酒精、

氢气、烟雾等的探测，如图：



MQ-2 的原理图如下：



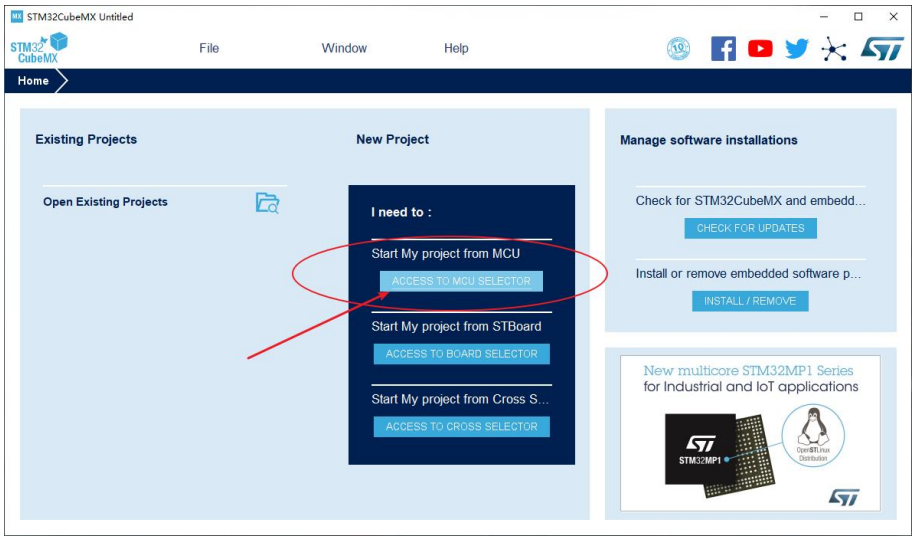
软件准备

- 需要安装好 Keil - MDK 及芯片对应的包，以便编译和下载生成的代码；
  Keil MDK 和串口助手的安装包都可以关注“**小熊派开源社区**”微信公众号，在资料教程一栏中可获取安装包。

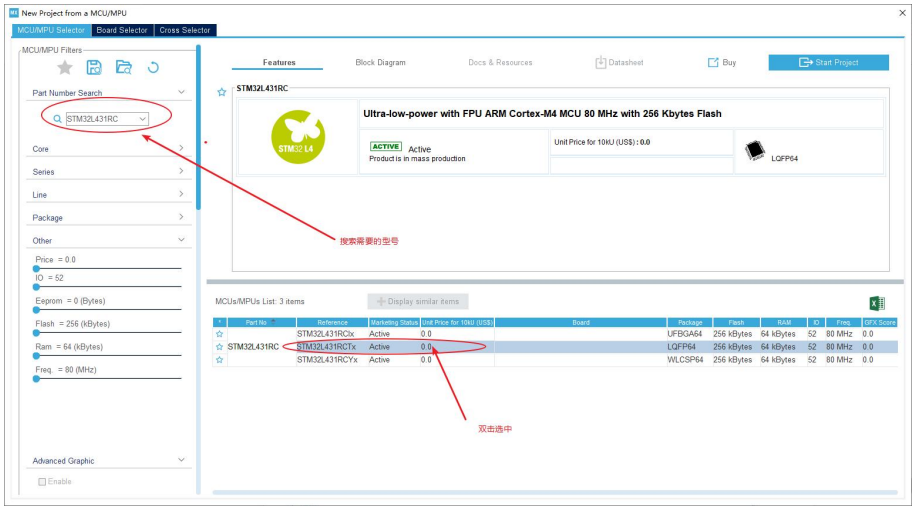
2.生成 MDK 工程

选择芯片型号

打开 STM32CubeMX，打开 MCU 选择器：



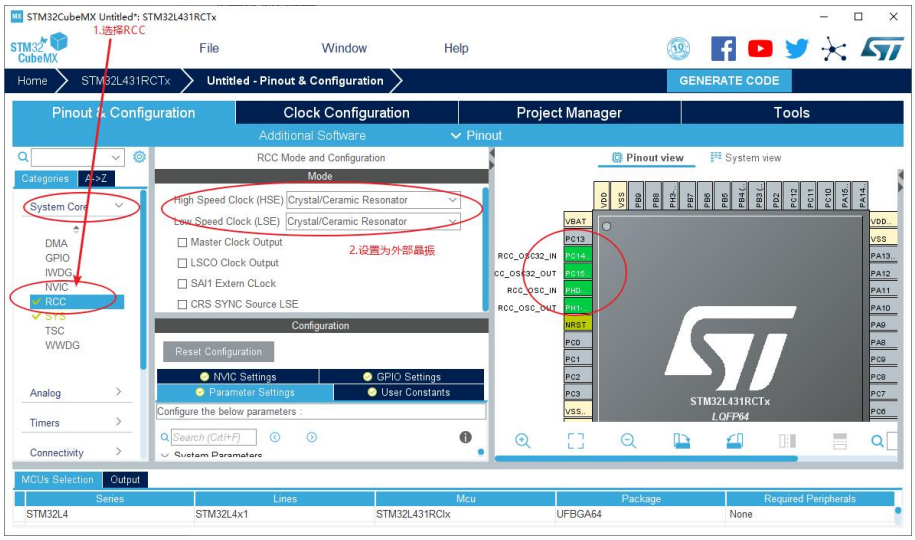
搜索并选中芯片 STM32L431RCT6:



配置时钟源

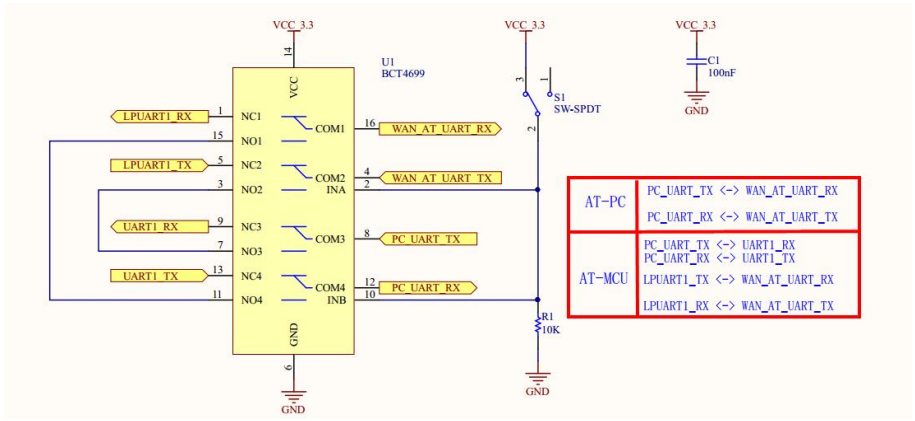
- 如果选择使用外部高速时钟（HSE），则需要在 System Core 中配置 RCC；
- 如果使用默认内部时钟（HSI），这一步可以略过；

这里我都使用外部时钟：



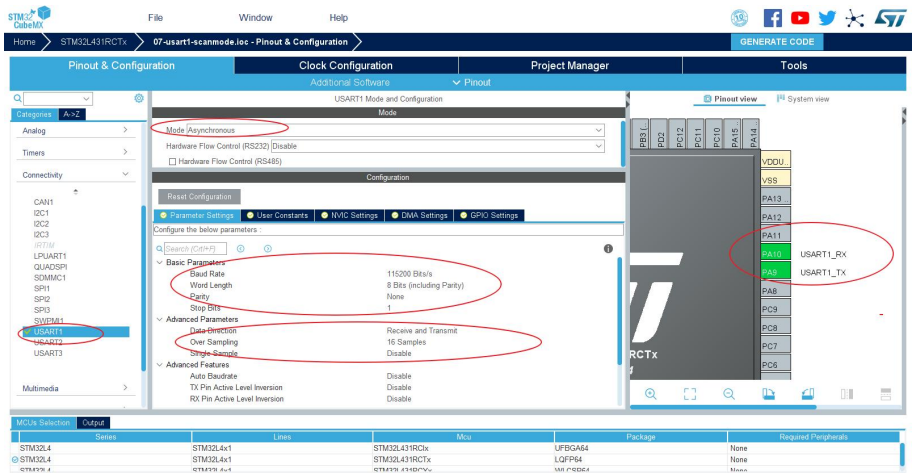
配置串口

小熊派开发板板载 ST-Link 并且虚拟了一个串口，原理图如下：



这里我将开关拨到 AT-MCU 模式，使 PC 的串口与 USART1 之间连接。

接下来开始配置 USART1：



配置 ADC

知识小卡片 —— ADC

ADC 全称 Analog-to-Digital Converter，即模拟-数字转换器，可以将连续变化的模拟信号转换为离散的数字信号，进而使用数字电路进行处理，称之为数字信号处理。

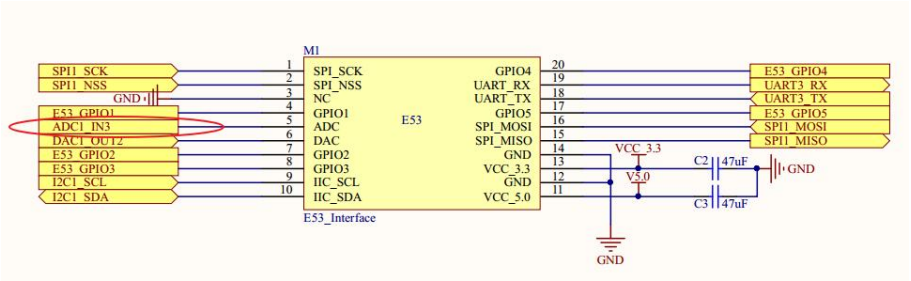
STM32L431xx 系列有 1 个 ADC，ADC 分辨率高达 12 位，每个 ADC 具有多达 20 个的采集通道，这些通道的 A/D 转换可以单次、连续、扫描或间断模式执行。 ADC 的结果可以左对齐或右对齐方式存储在 16 位数据寄存器中。

STM32L431 的 ADC 最大的转换速率为 5.33Mhz，也就是转换时间为 0.188us（12 位分辨率时），ADC 的转换时间与 AHB 总线时钟频率无关。

知识小卡片结束啦~对 ADC 有没有了解呢？

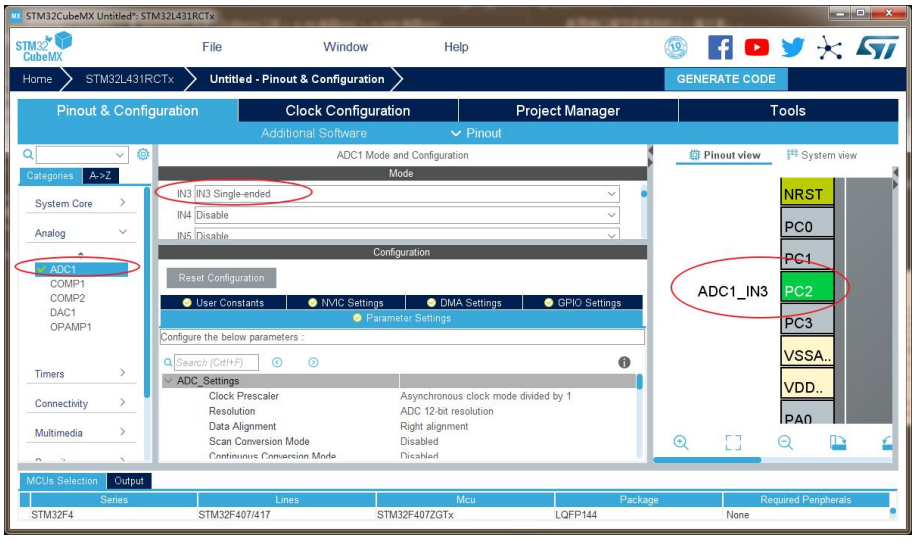
确定 ADC 通道

查看小熊派 E53 接口的原理图：



配置 ADC（单次转换模式）

首先选择 ADC1，开启通道 3：



接下来是对 ADC 的设置，这里我们保持默认即可：

ADC_Settings	
Clock Prescaler	Asynchronous clock mode divided by 1
Resolution	ADC 12-bit resolution
Data Alignment	Right alignment
Scan Conversion Mode	Disabled
Continuous Conversion Mode	Disabled
Discontinuous Conversion Mode	Disabled
DMA Continuous Requests	Disabled
End Of Conversion Selection	End of single conversion
Overrun behaviour	Overrun data preserved
Low Power Auto Wait	Disabled

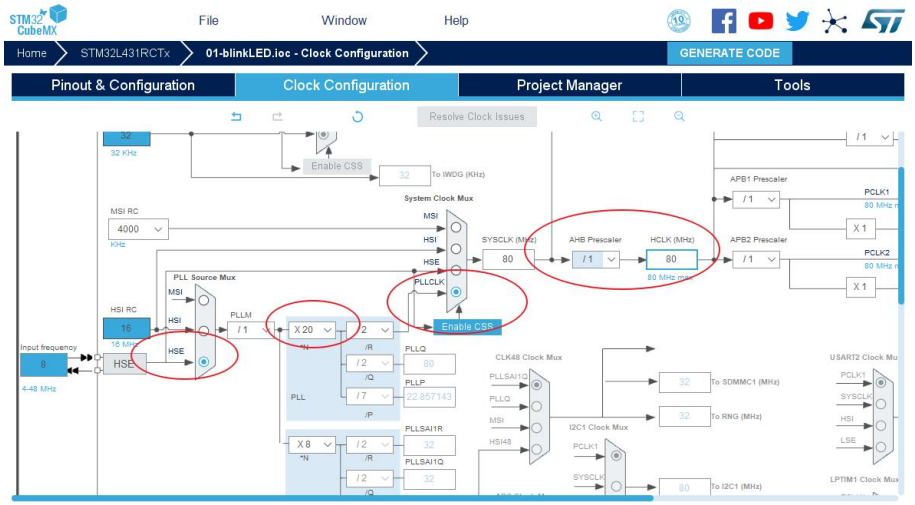
最后设置 ADC 的转换规则：

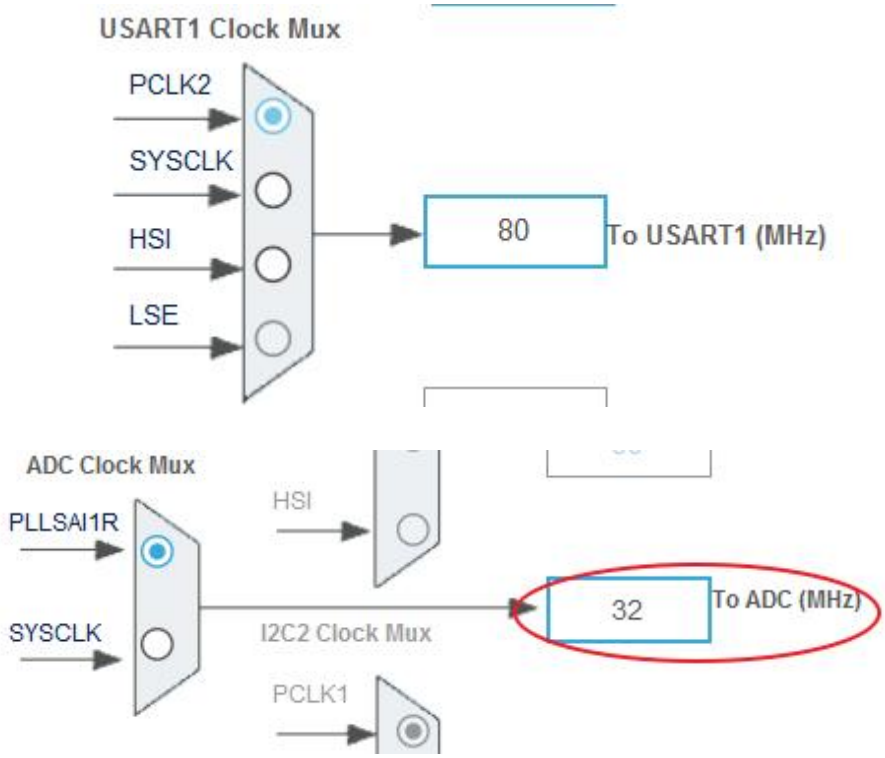
ADC_Regular_ConversionMode	
Enable Regular Conversions	Enable
Enable Regular Oversampling	Disable
Number Of Conversion	1
External Trigger Conversion Source	Regular Conversion launched by software
External Trigger Conversion Edge	None
Rank	1
Channel	Channel 3
Sampling Time	2.5 Cycles
Offset Number	No offset
ADC_Injected_ConversionMode	
Enable Injected Conversions	Disable

其余的一些设置保持默认即可。

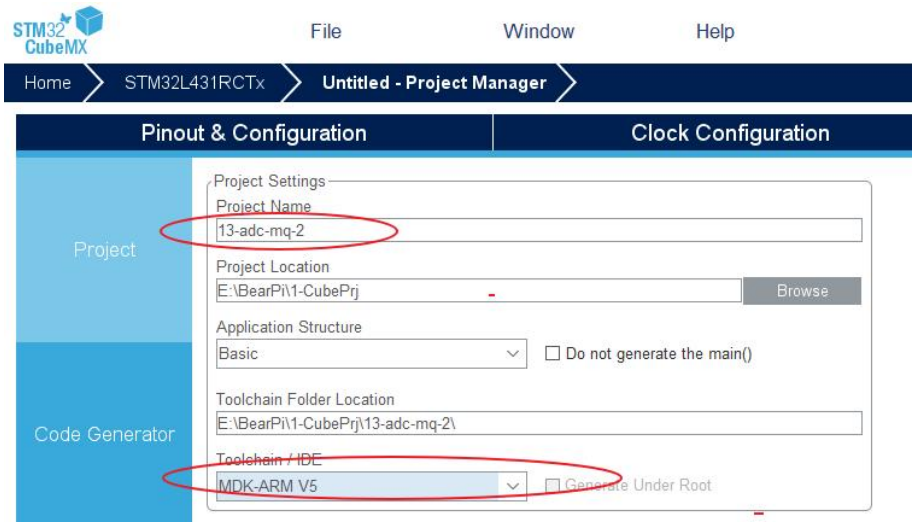
配置时钟树

STM32L4 的最高主频到 80M，所以配置 PLL，最后使 `HCLK = 80Mhz` 即可：



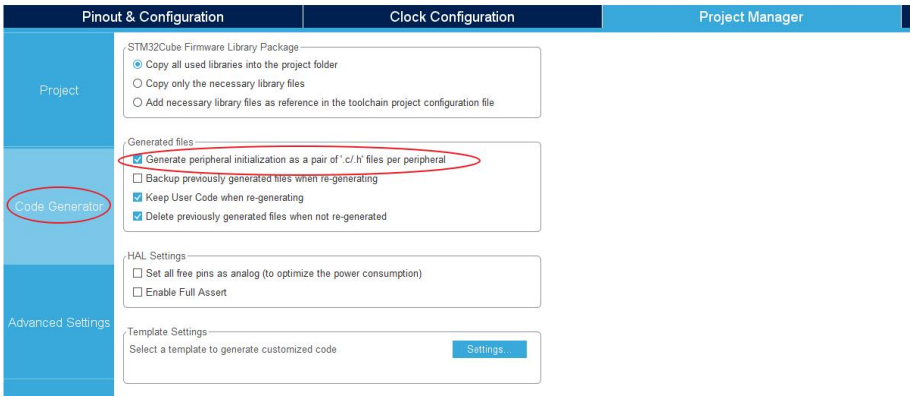


生成工程设置



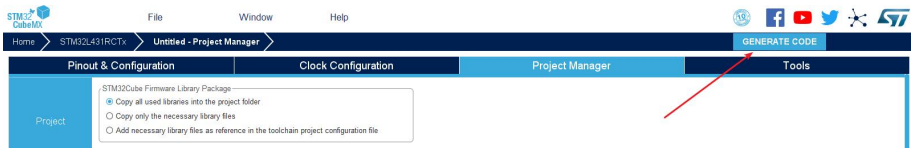
代码生成设置

最后设置生成独立的初始化文件：



生成代码

点击 GENERATE CODE 即可生成 MDK-V5 工程：



3. 在 MDK 中编写、编译、下载用户代码

重定向 printf( )函数

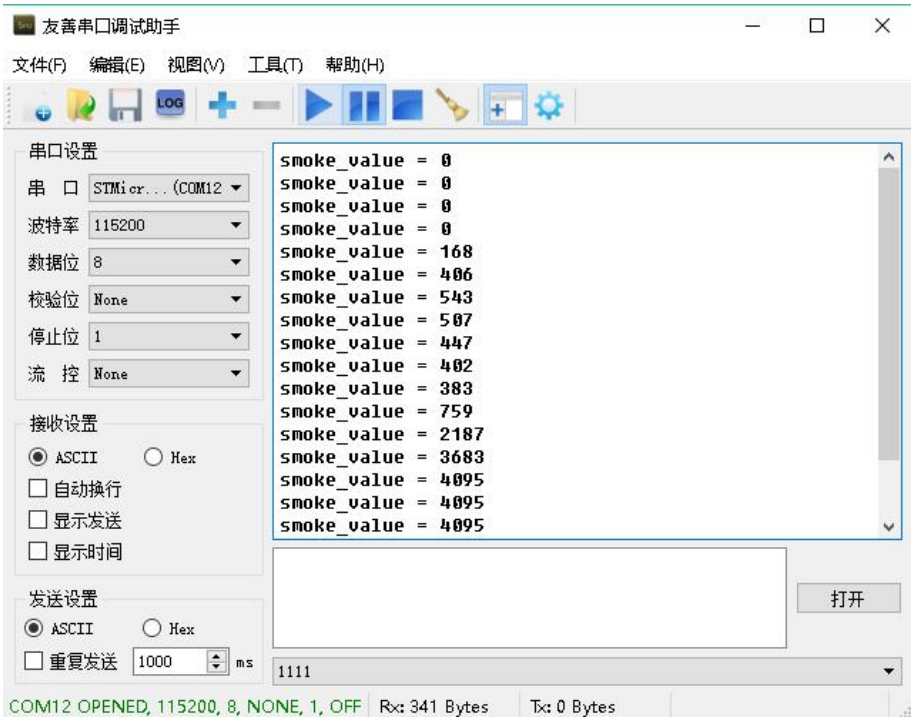
参考：基于串口发送函数实现 printf()。

编写读取数据的测试代码

修改 main 函数如下：

```
int main(void){

        uint16_t smoke_value = 0;


    HAL_Init();


    SystemClock_Config();


    MX_GPIO_Init();
    MX_ADC1_Init();
    MX_USART1_UART_Init();
```

```
    while (1)
    {
        HAL_ADC_Start(&hadc1);                    //启动 ADC 单次转换
        HAL_ADC_PollForConversion(&hadc1, 50);      //等待 ADC 转换完成
        smoke_value = HAL_ADC_GetValue(&hadc1); //读取 ADC 转换数据
        printf("smoke_value = %d\n", smoke_value);
        HAL_Delay(500);
    }}
```



至此，我们已经学会**如何使用 ADC 读取 MQ-2 传感器的值**，下一节将讲述如何使用通用定时器闪烁 LED。

作业：

编写程序，通过 ADC 读取 MQ-2 气体传感器的输出，并根据输出值判断环境中的气体浓度。

分析 ADC 的精度和采样率对测量结果的影响。

STM32 单片机基础 11——使用通用定时器闪烁 LED

教学目的与要求：

目的：理解 STM32 通用定时器的工作原理，掌握如何通过定时器控制 LED 的闪烁频率。

要求：能够配置通用定时器，编写程序实现 LED 的定时闪烁，理解定时器溢出中断或更新事件的应用。

教学重难点：

重点：通用定时器的配置和 LED 闪烁逻辑的实现。

难点：定时器中断或更新事件的理解与应用，确保 LED 闪烁频率的准确性。

课时数：3 课时

思政元素：

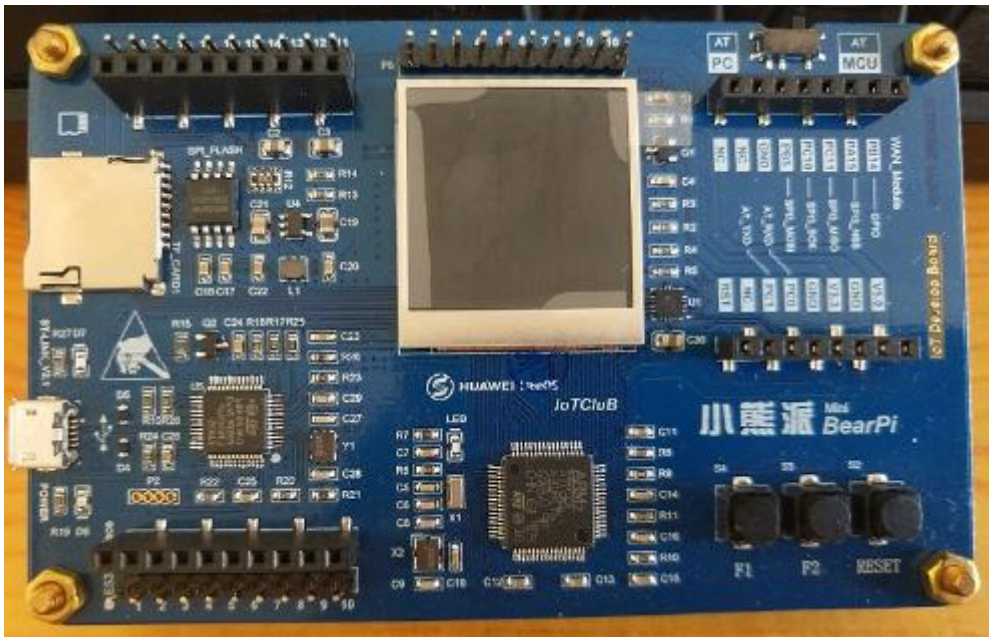
培养学生的时间观念和节奏感，通过定时器控制 LED 闪烁的实践，让学生理解在嵌入式系统设计中，精确控制时间对于实现各种定时任务的重要性，培养学生的时间管理能力和对精确性的追求

本篇详细的记录了如何使用 STM32CubeMX 配置 STM32L431RCT6 的通用定时器外设，以中断的方式使 LED 闪烁。

1. 准备工作

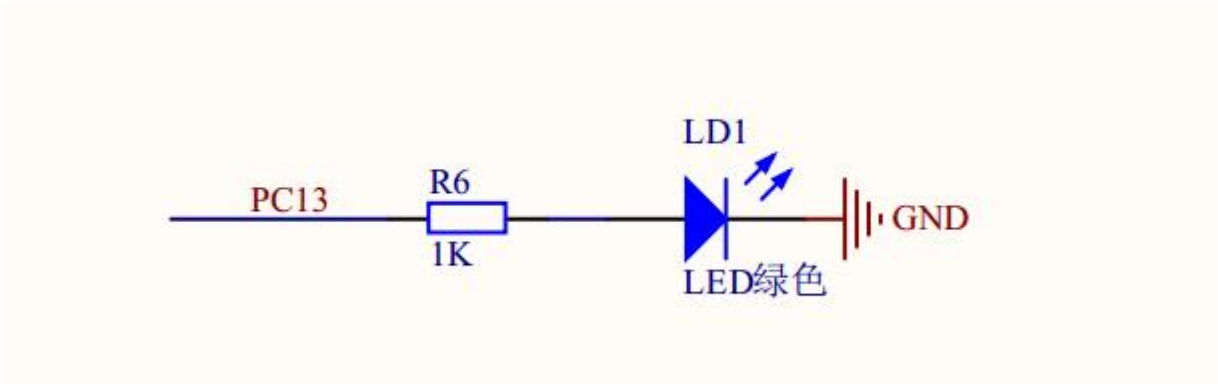
硬件准备

开发板
首先需要准备一个开发板，这里我准备的是 STM32L4 的开发板（BearPi）：



测试 LED

这里我直接使用板载 LED，原理图如下：



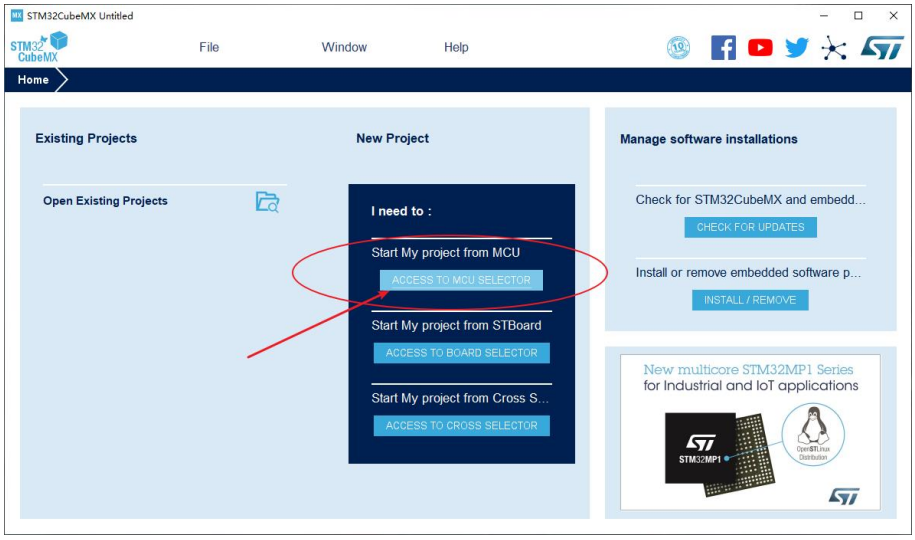
软件准备

- 需要安装好 Keil - MDK 及芯片对应的包，以便编译和下载生成的代码；
  Keil MDK 和串口助手的安装包都可以关注“**小熊派开源社区**”微信公众号，在资料教程一栏中可获取安装包。

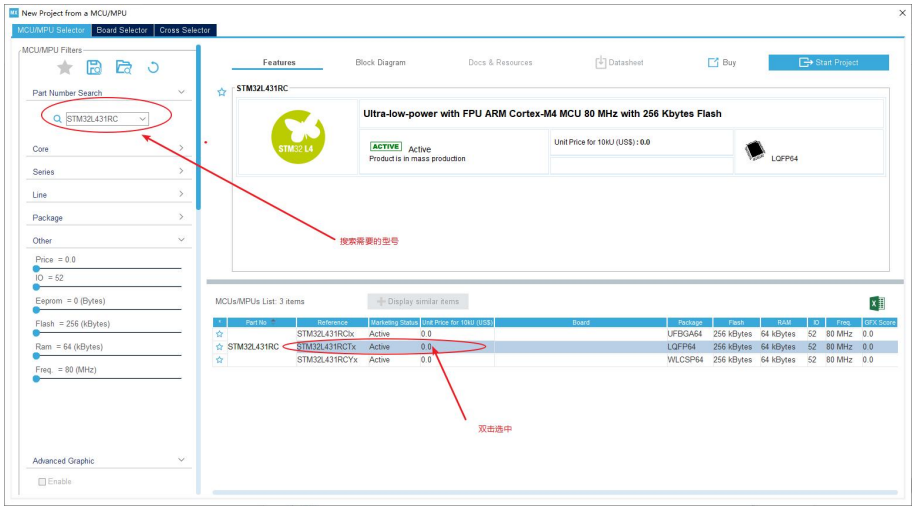
2.生成 MDK 工程

选择芯片型号

打开 STM32CubeMX，打开 MCU 选择器：



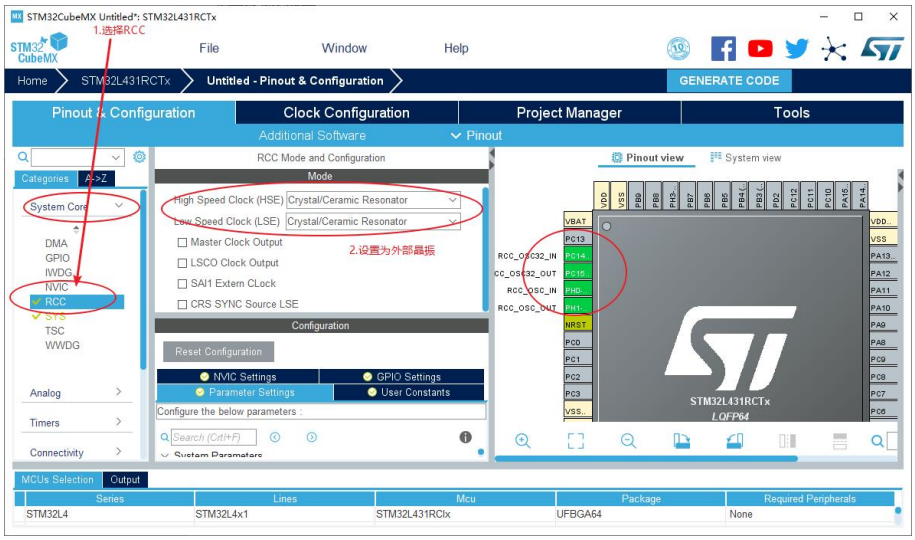
搜索并选中芯片 STM32L431RCT6:



配置时钟源

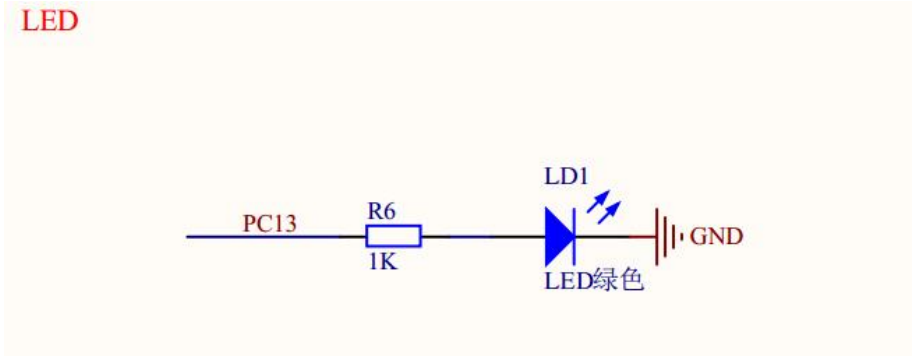
- 如果选择使用外部高速时钟（HSE），则需要在 System Core 中配置 RCC；
- 如果使用默认内部时钟（HSI），这一步可以略过；

这里我都使用外部时钟：

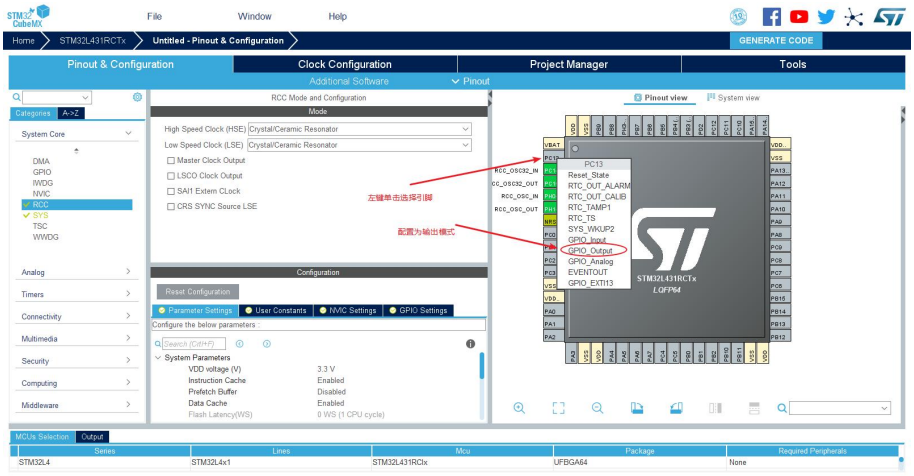


配置 LED 的 GPIO 引脚

查看小熊派开发板的原理图，如下：



所以接下来我们选择配置 PC13 引脚：



配置通用定时器 TIM2

知识小卡片——STM32L431 的定时器

STM32L431xx 系列有 1 个高级定时器（TIM1），3 个通用定时器（TIM2、TIM15、TIM16），两个基本定时器（TIM6、TIM7），还有两个低功耗定时器（LPTIM1、LPTIM2）。

STM32L431 的通用 TIMx (TIM2、TIM15、TIM16)定时器功能包括：

- 16 位(TIM15,TIM16)/32 位(TIM2)向上、向下、向上/向下自动装载计数器，注意：TIM15、TIM16 只支持向上（递增）计数方式；
- 16 位可编程(可以实时修改)预分频器，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， 其中 TIM15 最多 2 个通道， TIM16 最多 1 个通道），这些通道可以用来作为：
  - 输入捕获
  - 输出比较
  - PWM 生成(边缘或中间对齐模式)
  - 单脉冲模式输出
- 可使用外部信号控制定时器和定时器互连的同步电路。

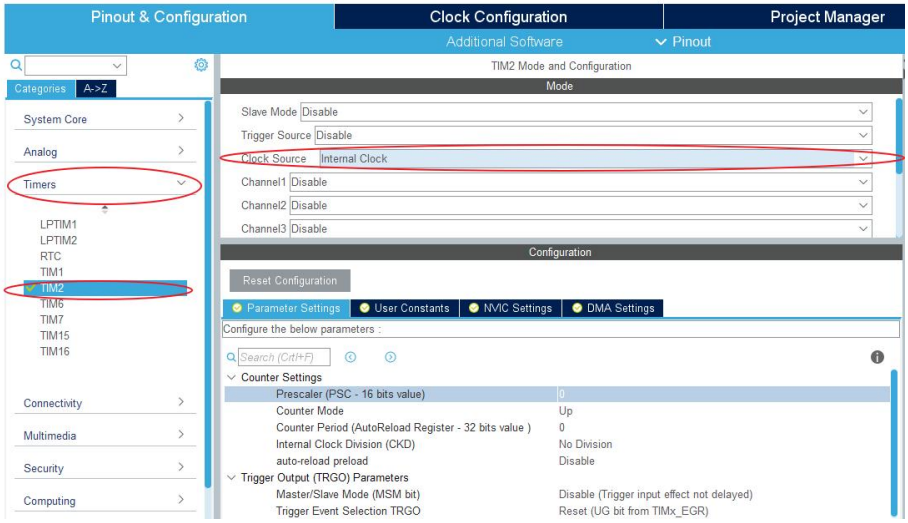
如下事件发生时产生中断/DMA：

-
  - 更新：计数器向上溢出/向下溢出，计数器初始化(通过软件或者内部/外部触发)
  - 触发事件(计数器启动、停止、初始化或者由内部/外部触发计数)
  - 输入捕获
  - 输出比较

知识小卡片结束啦~

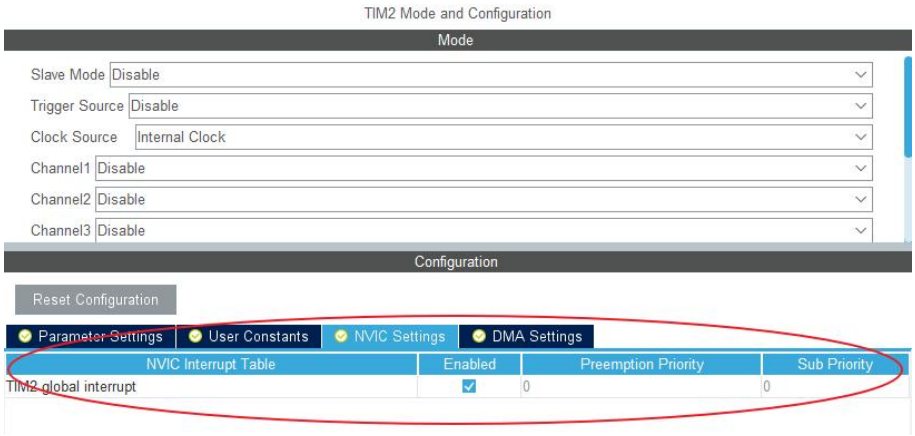
配置定时器 TIM2

首先选择 TIM2，时钟源选择内部时钟：



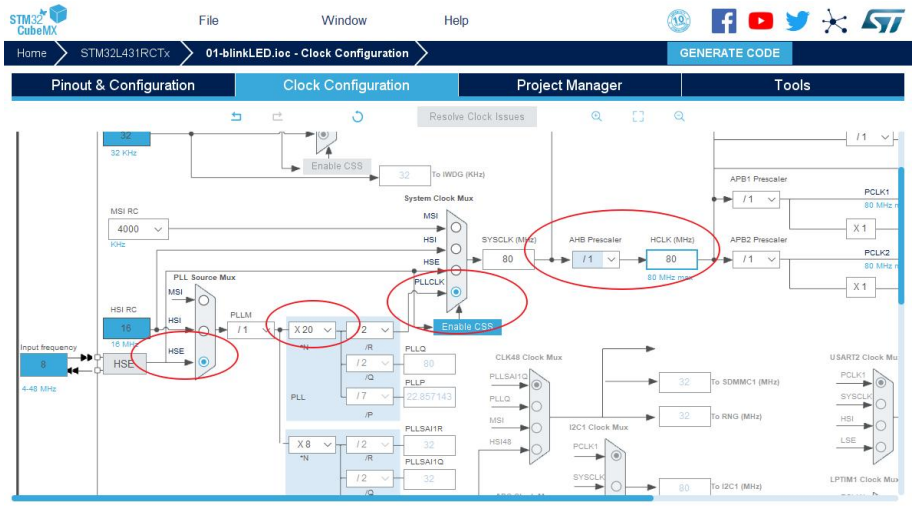
接下来是对 TIM2 的参数设置，参照数据手册中的 RCC 时钟树，TIM2 内部时钟来源是 PCLK1 = 80Mhz，我们的目的是每秒钟产生 2 次中断，所以预分频系数设置为 40000-1，自动重载值为 1000-1，得到的计时器更新中断频率即为 80000000/40000/1000=2Hz：

其余的一些设置保持默认即可，最后开启 TIM2 中断：

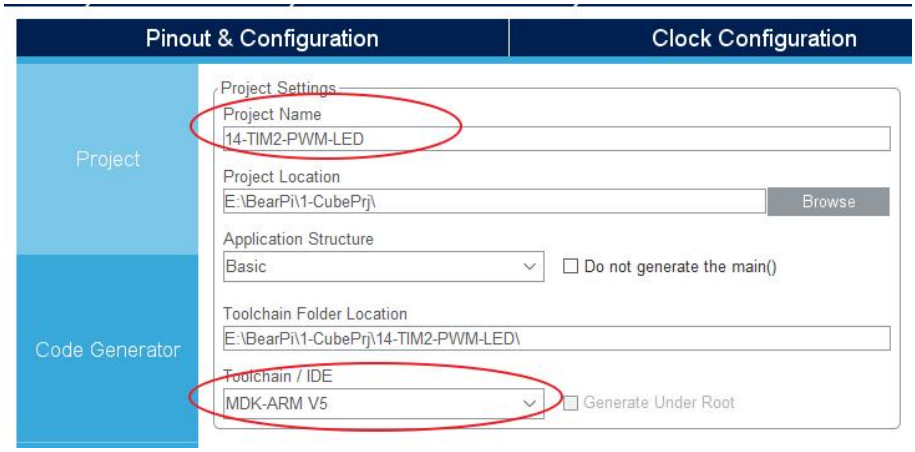


配置时钟树

STM32L4 的最高主频到 80M，所以配置 PLL，最后使 `HCLK = 80Mhz` 即可：

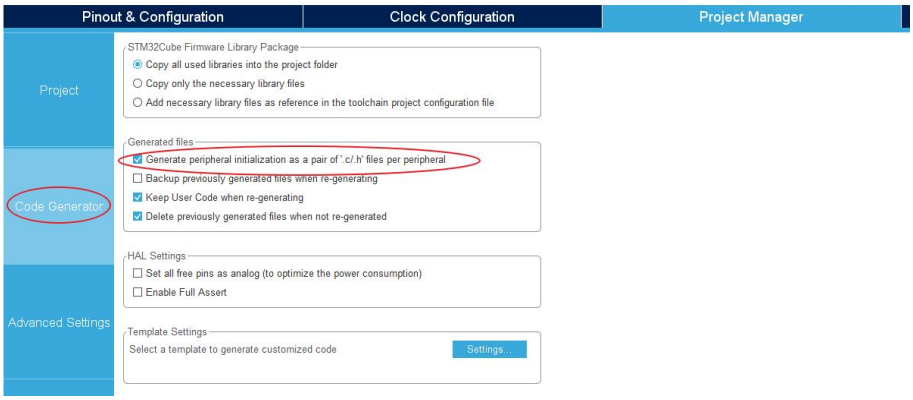


生成工程设置



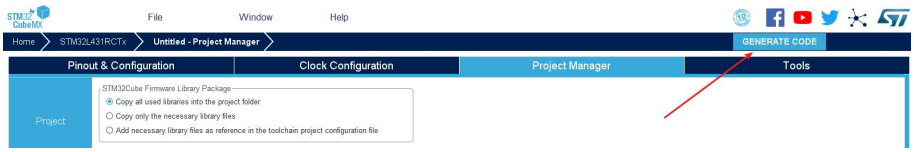
代码生成设置

最后设置生成独立的初始化文件：



生成代码

点击 GENERATE CODE 即可生成 MDK-V5 工程：



3. 在 MDK 中编写、编译、下载用户代码

编写中断回调函数

在 stm32l4xx_it.c 中生成的中断处理函数如下，定时器 TIM2 所有的中断都会调用该中断服务函数 TIM2_IRQHandler：

```
200 /**
201   * @brief This function handles TIM2 global interrupt.
202   */
203 void TIM2_IRQHandler(void)
204 {
205   /* USER CODE BEGIN TIM2_IRQn 0 */
206
207   /* USER CODE END TIM2_IRQn 0 */
208   HAL_TIM_IRQHandler(&htim2);
209   /* USER CODE BEGIN TIM2_IRQn 1 */
210
211   /* USER CODE END TIM2_IRQn 1 */
212 }
```

在中断处理函数中自动生成了 HAL_TIM_IRQHandler(&htim2) 代码，该代码会自动根据中断事件回调相应的函数，这里我们需要处理**更新中断的事件**，回调函数默认是 __weak 定义的，所以在 tim.c 中重新定义该回调函数，并且在该函数中添加功能的时候，因为该回调函数会被所有的定时器共用，所以需要先判断是哪个定时器在调用：

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef* tim_baseHandle){

        if(tim_baseHandle->Instance == htim2.Instance)

                HAL_GPIO_TogglePin(LED_GPIO_Port, LED_Pin);}
```

启动定时器并使能中断

最后在 `main` 函数中开启 TIM2 并使能其中断（TIM2 初始化代码之后，while 之前）：

```
HAL_TIM_Base_Start_IT(&htim2);
```

测试结果

编译下载后即可看到 LED 以 2 Hz 的频率闪烁。

至此，我们已经学会**如何使用通用定时器闪烁 LED**，下一节将讲述如何使用通用定时器产生 PWM 驱动蜂鸣器。

作业：
使用 STM32 的通用定时器控制 LED 以不同频率闪烁。
分析定时器的工作原理和配置方法。

STM32 单片机基础 12——使用通用定时器产生 PWM 驱动蜂鸣器

教学目的与要求：

目的：掌握 PWM（脉冲宽度调制）信号的产生原理，学会使用 STM32 通用定时器产生 PWM 信号来控制蜂鸣器的声音频率。

要求：能够配置通用定时器以产生 PWM 信号，编写程序调整 PWM 占空比以改变蜂鸣器的发声频率。

教学重难点：

重点：PWM 信号的产生原理和通用定时器的配置。

难点：理解 PWM 占空比与蜂鸣器发声频率之间的关系，确保 PWM 信号输出的准确性和稳定性。

课时数：3 课时

思政元素：

培养学生的创新能力和实践能力，通过 PWM 信号控制蜂鸣器的实践，让学生理解在音频控制等应用中，PWM 信号的灵活性和重要性，鼓励学生尝试不同的 PWM 参数组合，探索不同的声音效果，培养学生的探索精神和创造力。

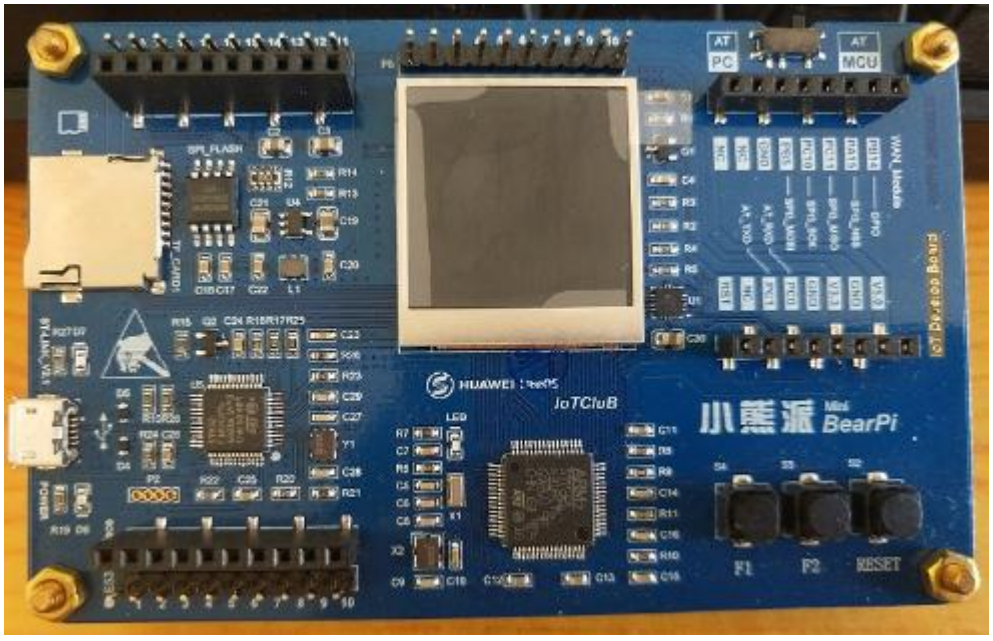
本篇详细的记录了如何使用 STM32CubeMX 配置 STM32L431RCT6 的通用定时器外设，产生 PWM 驱动无源蜂鸣器。

1. 准备工作

硬件准备

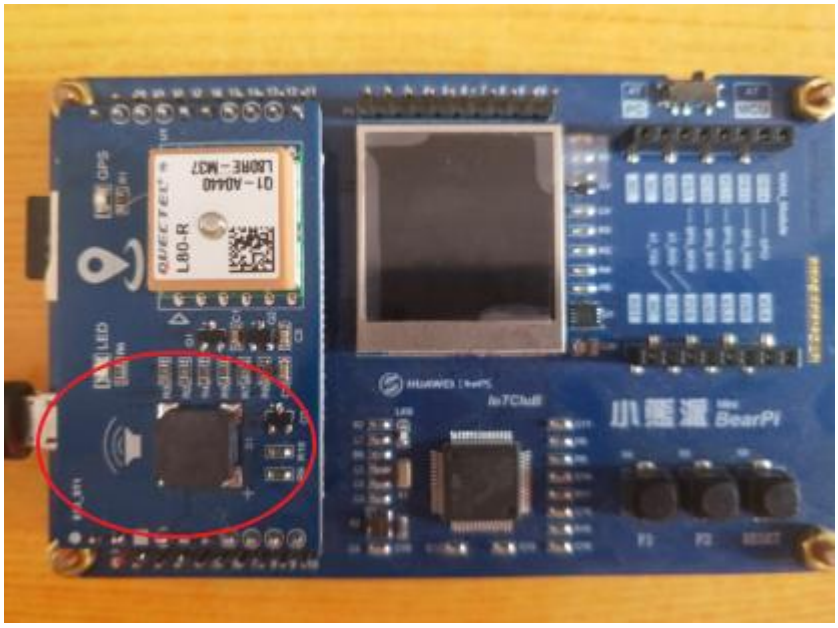
开发板

首先需要准备一个开发板，这里我准备的是 STM32L4 的开发板（BearPi）：

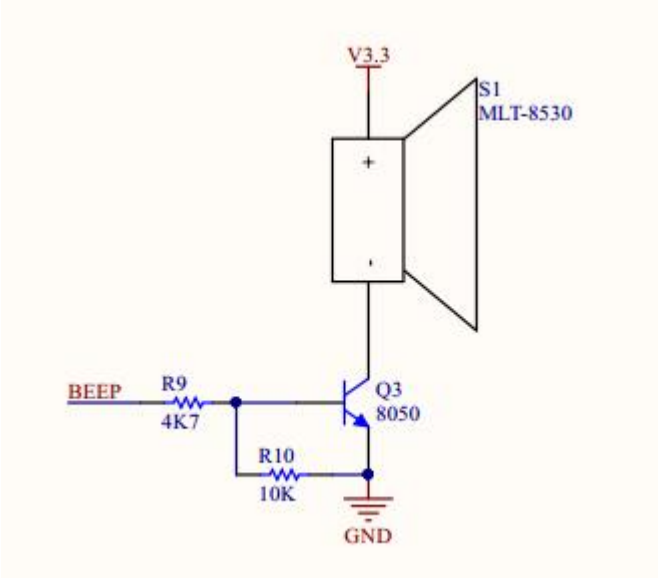


- 蜂鸣器

这里我直接使用扩展板上的蜂鸣器，如图：



蜂鸣器的原理图如下：



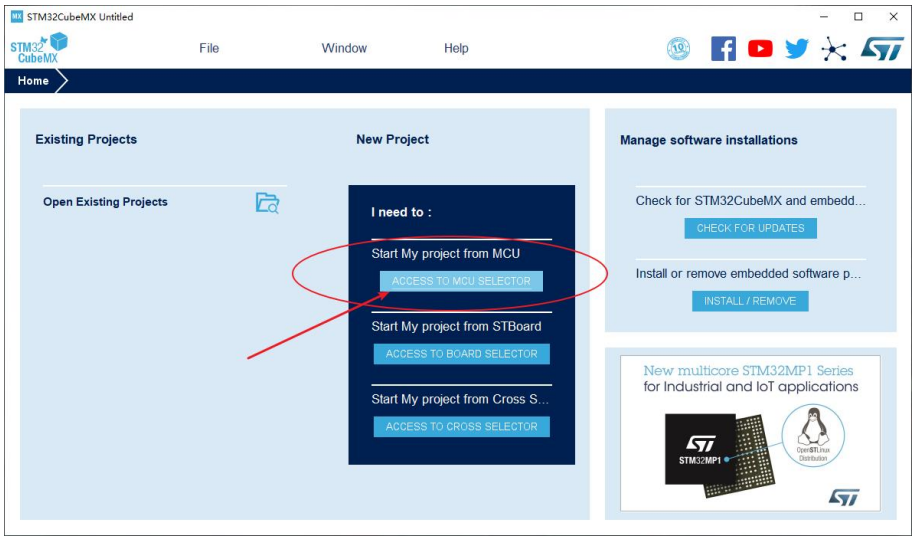
软件准备

- 需要安装好 Keil - MDK 及芯片对应的包，以便编译和下载生成的代码；
  Keil MDK 和串口助手的安装包都可以关注“**小熊派开源社区**”微信公众号，在资料教程一栏中可获取安装包。

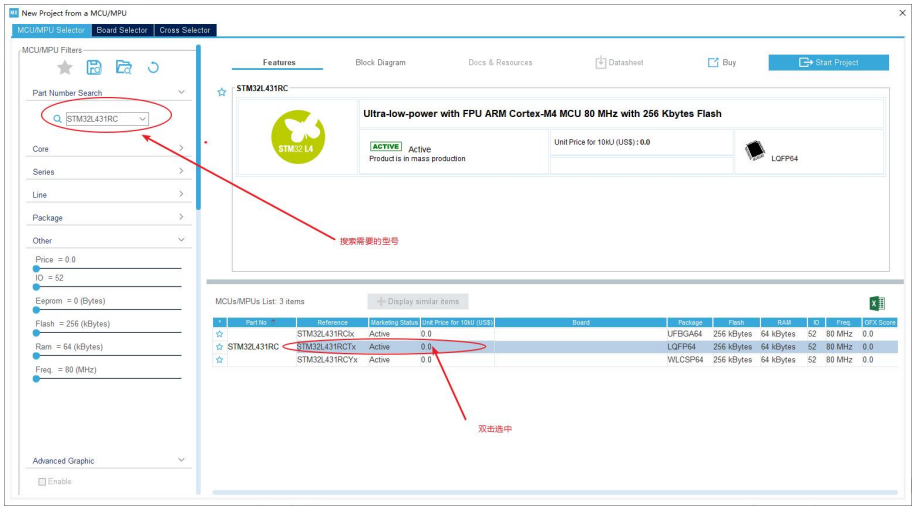
2.生成 MDK 工程

选择芯片型号

打开 STM32CubeMX，打开 MCU 选择器：



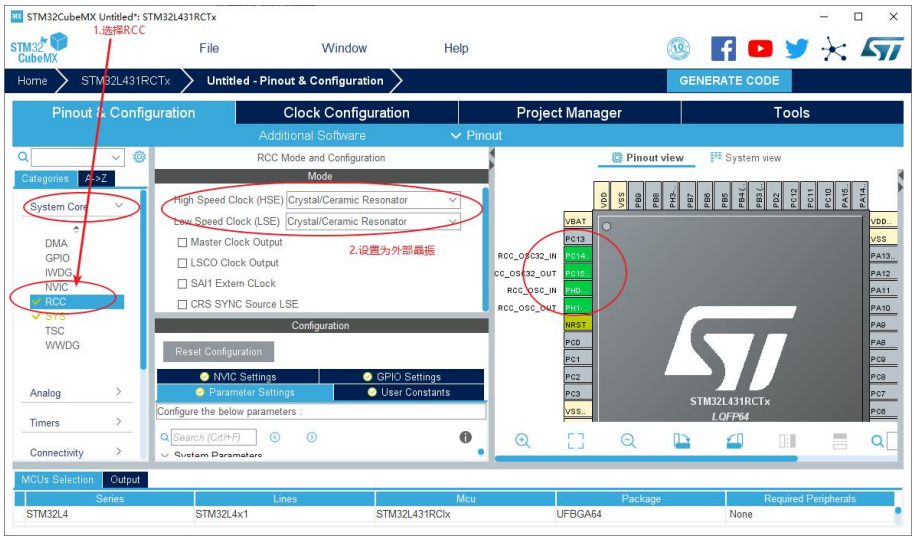
搜索并选中芯片 STM32L431RCT6:



配置时钟源

- 如果选择使用外部高速时钟（HSE），则需要在 System Core 中配置 RCC；
- 如果使用默认内部时钟（HSI），这一步可以略过；

这里我都使用外部时钟：



配置通用定时器 TIM16

知识小卡片——STM32L431 的定时器

STM32L431xx 系列有 1 个高级定时器（TIM1），3 个通用定时器（TIM2、TIM15、TIM16），两个基本定时器（TIM6、TIM7），还有两个低功耗定时器（LPTIM1、LPTIM2）。

STM32L431 的通用 TIMx (TIM2、TIM15、TIM16)定时器功能包括：

- 16 位(TIM15,TIM16)/32 位(TIM2)向上、向下、向上/向下自动装载计数器，注意：TIM15、TIM16 只支持向上（递增）计数方式；
- 16 位可编程(可以实时修改)预分频器，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， 其中 TIM15 最多 2 个通道， TIM16 最多 1 个通道），这些通道可以用来作为：
  - 输入捕获
  - 输出比较
  - PWM 生成(边缘或中间对齐模式)
  - 单脉冲模式输出
- 可使用外部信号控制定时器和定时器互连的同步电路。

-

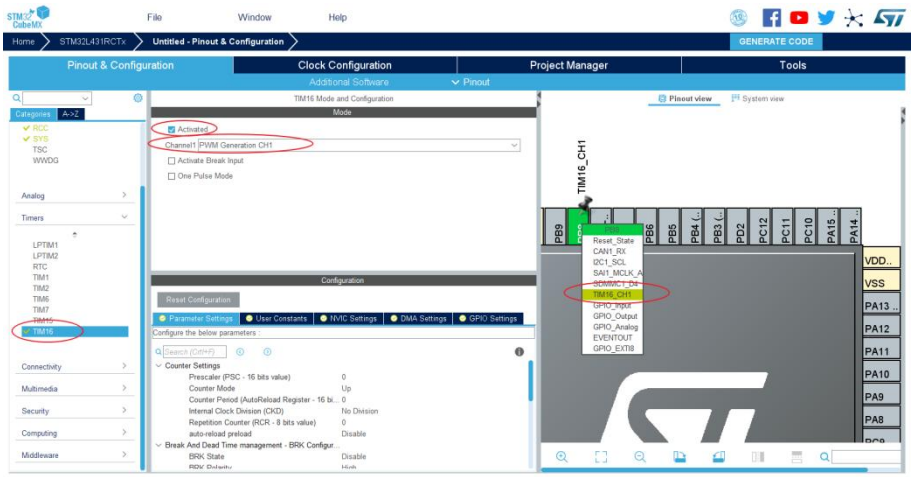
如下事件发生时产生中断/DMA：

-
  - 更新：计数器向上溢出/向下溢出，计数器初始化(通过软件或者内部/外部触发)
  - 触发事件(计数器启动、停止、初始化或者由内部/外部触发计数)
  - 输入捕获
  - 输出比较

知识小卡片结束啦~

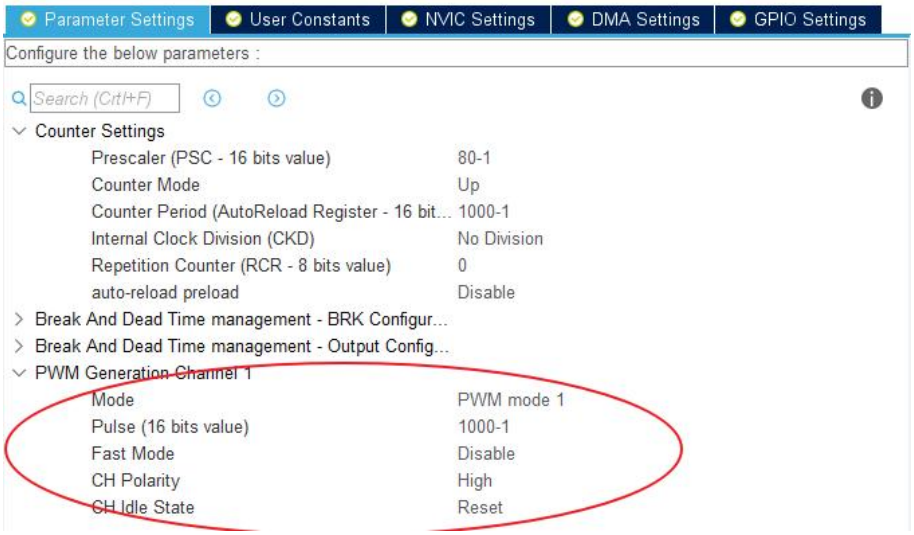
接下来开始配置 TIM16 定时器的 PWM 功能：

首先选择 TIM，选择通道 1 的功能，默认的 CH1 是 PA6 引脚，但是开发板上是与 PB8 连接的，所以在右边将 PB8 配置为 TIM16_CH1：



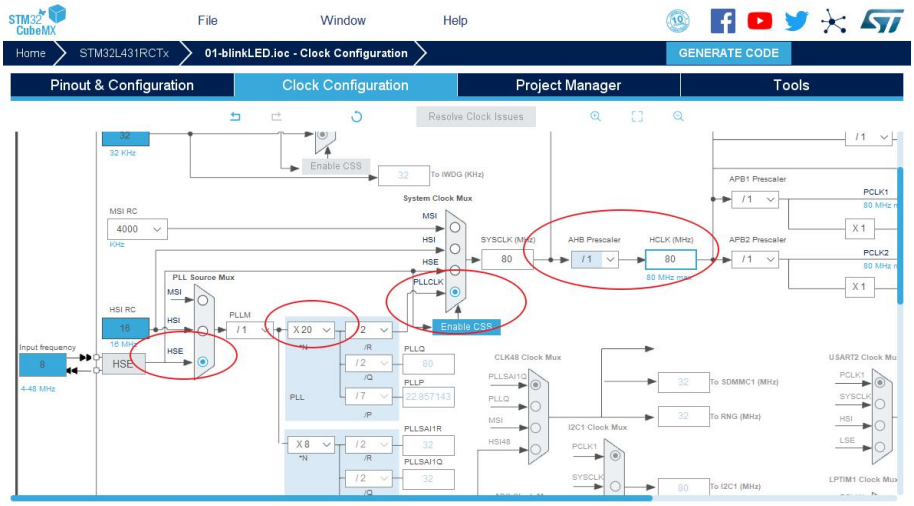
接下来是对 TIM16 的参数设置，参照数据手册中的 RCC 时钟树，TIM16 内部时钟来源是 PCLK2 = 80Mhz，我们的目的是产生 1khz 的 PWM，所以预分频系数设置为 80-1，自动重载值为 1000-1，得到的计时器更新中断频率即为 80000000/80/1000 = 1000 Hz = 1K Hz：

其余的一些设置保持默认即可，最后配置 PWM 占空比：

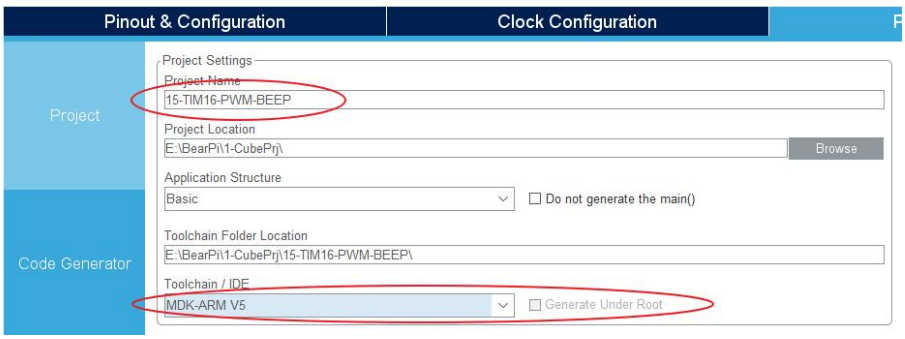


配置时钟树

STM32L4 的最高主频到 80M，所以配置 PLL，最后使 `HCLK = 80Mhz` 即可：

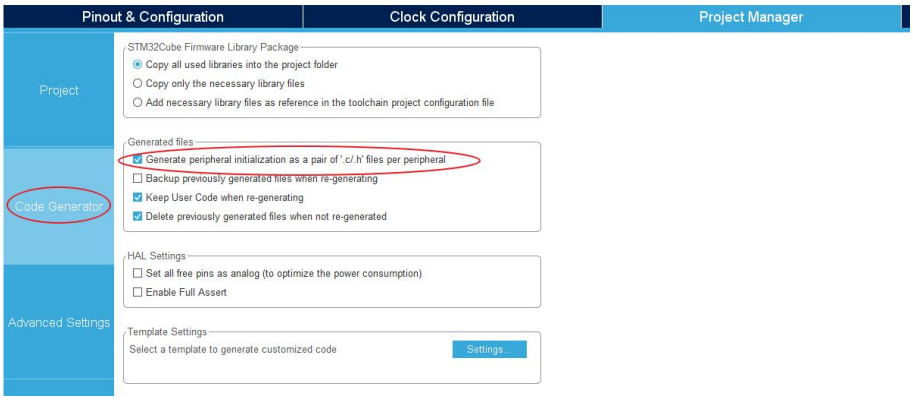


生成工程设置



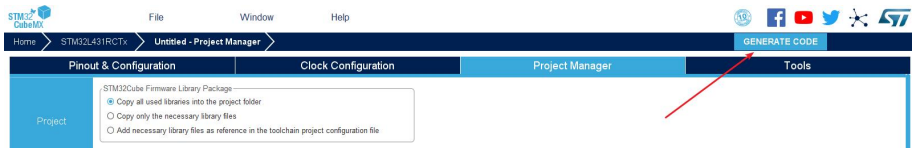
代码生成设置

最后设置生成独立的初始化文件：



生成代码

点击 GENERATE CODE 即可生成 MDK-V5 工程：



3. 在 MDK 中编写、编译、下载用户代码

启动定时器并产生 PWM

最后在 main 函数中开启 TIM2 并使能其中断（TIM2 初始化代码之后）：

```
while (1){
  HAL_TIM_PWM_Start(&htim16,TIM_CHANNEL_1);
  HAL_Delay(1000);
  HAL_TIM_PWM_Stop(&htim16,TIM_CHANNEL_1);
  HAL_Delay(1000);}
```

测试结果

编译下载后即可听到无源蜂鸣器开始工作。

至此，我们已经学会**如何使用通用定时器产生 PWM 驱动蜂鸣器**，下一节将讲述如何使用硬件 IIC 接口读写 EEPROM。

作业

编写程序，使用 PWM 信号控制蜂鸣器发出不同频率的声音。

分析 PWM 信号的生成原理及其在音频控制中的应用。

STM32 单片机基础 13——使用硬件 I2C 读写 EEPROM（AT24C02）

教学目的与要求：

目的：掌握 I2C 通信协议的基本原理，学会使用 STM32 的硬件 I2C 接口与 EEPROM 进行通信，实现数据的存储和读取。

要求：能够配置 STM32 的硬件 I2C 接口，编写程序对 EEPROM 进行读写操作，理解 I2C 通信的时序和协议细节。

教学重难点：

重点：I2C 通信协议的理解和硬件 I2C 接口的配置。

难点：确保 I2C 通信的稳定性和可靠性，处理可能的通信错误和异常情况。

课时数：3 课时

思政元素：

培养学生的团队合作精神和沟通能力，在 I2C 通信的学习过程中，由于涉及到多个设备的协同工作，需要理解并遵守通信协议，这有助于培养学生的团队意识和协作能力，同时，也让学生理解在复杂系统中，遵守规则和协议的重要性。

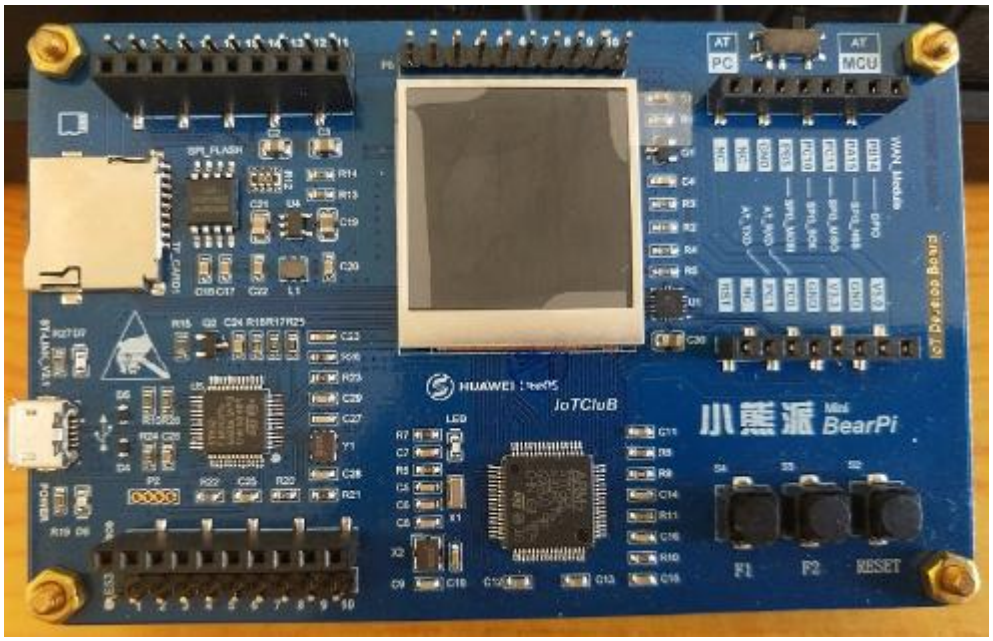
本篇详细的记录了如何使用 STM32CubeMX 配置 STM32L431RCT6 的硬件 I2C 外设读取 EEPROM 数据（以 AT24C02 为例）。

1. 准备工作

硬件准备

开发板

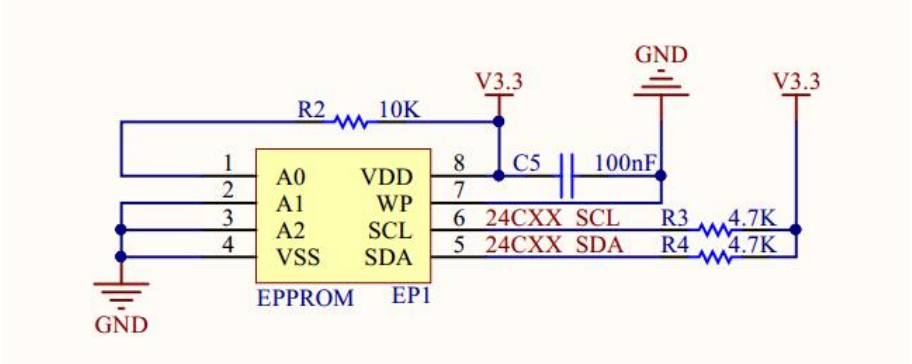
首先需要准备一个开发板，这里我准备的是 STM32L4 的开发板（BearPi）：



- EEPROM

小熊派开发板左边的接口是 E53 接口，用来连接 E53 接口的扩展板，每个扩展板都板载了一块 EEPROM 用来保存信息，如图：

AT24C02 的原理图如下（**该原理图中有 bug，A0 的上拉电阻无效，实际 A0 为低电平**）：



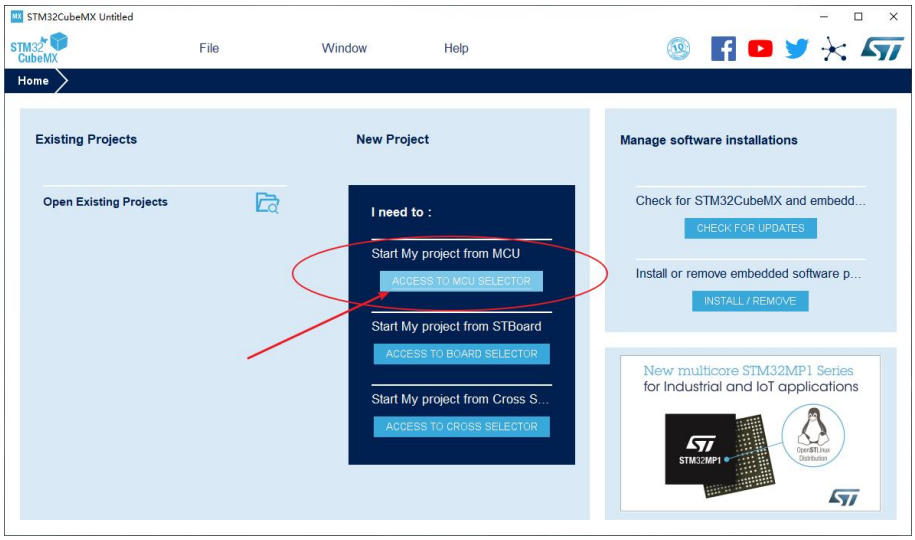
软件准备

- 需要安装好 Keil - MDK 及芯片对应的包，以便编译和下载生成的代码；
  Keil MDK 和串口助手的安装包都可以关注“**小熊派开源社区**”微信公众号，在资料教程一栏中可获取安装包。

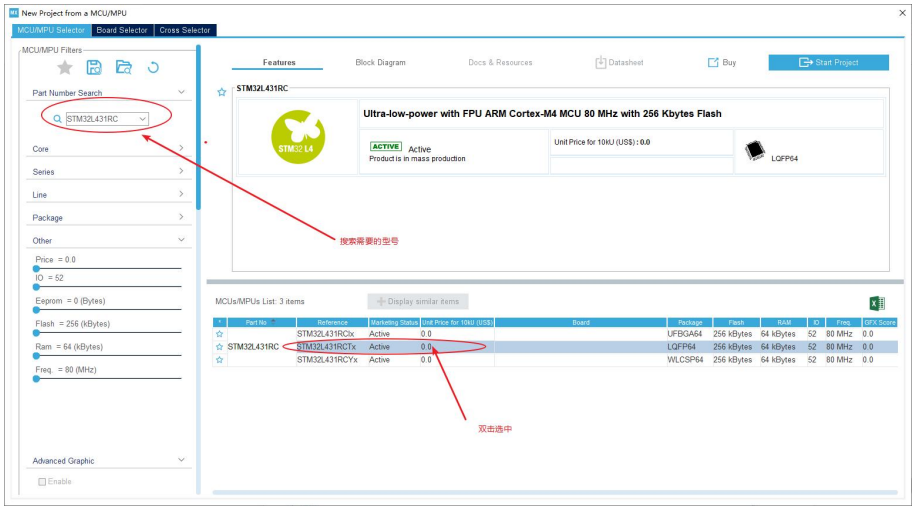
2.生成 MDK 工程

选择芯片型号

打开 STM32CubeMX，打开 MCU 选择器：



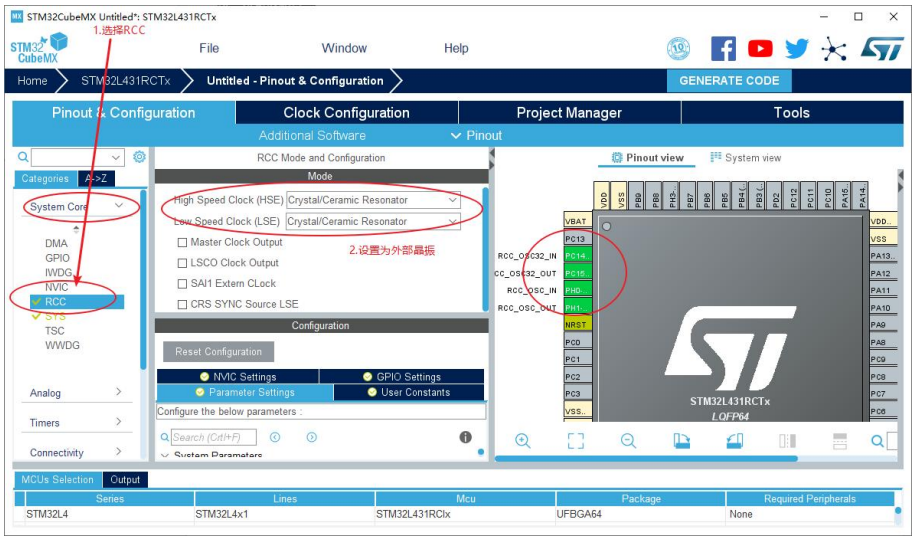
搜索并选中芯片 STM32L431RCT6:



配置时钟源

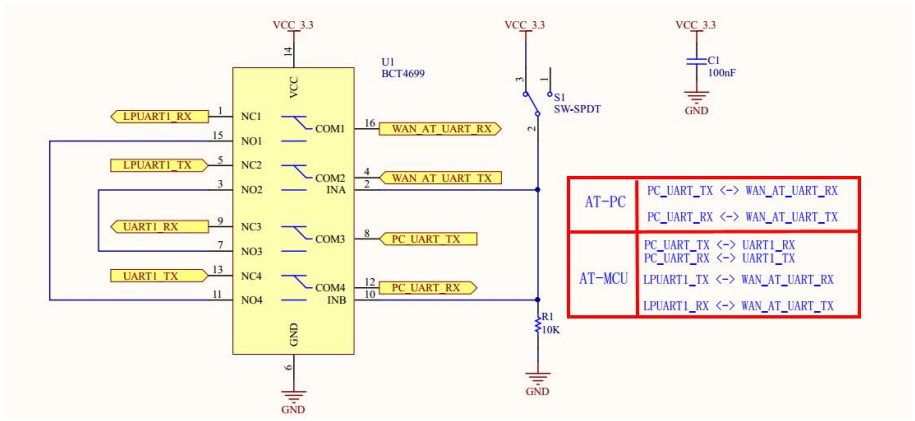
- 如果选择使用外部高速时钟（HSE），则需要在 System Core 中配置 RCC；
- 如果使用默认内部时钟（HSI），这一步可以略过；

这里我都使用外部时钟：



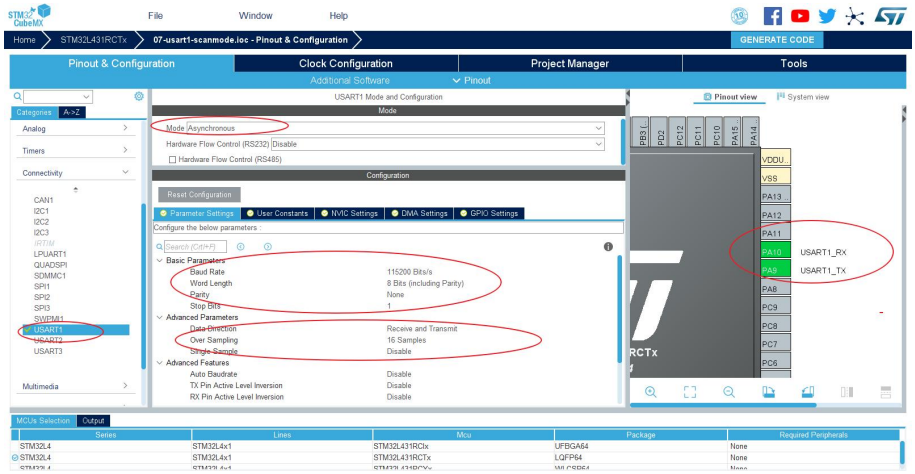
配置串口

小熊派开发板板载 ST-Link 并且虚拟了一个串口，原理图如下：



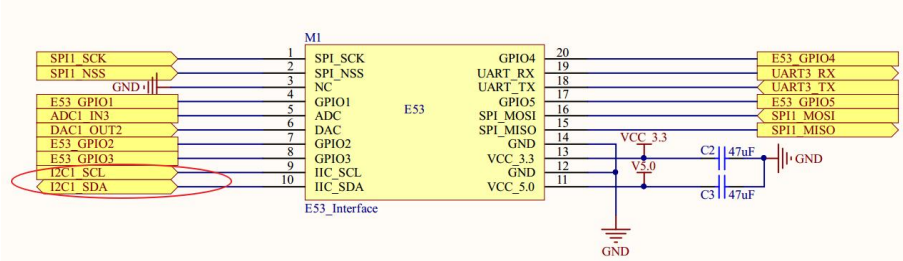
这里我将开关拨到 AT-MCU 模式，使 PC 的串口与 USART1 之间连接。

接下来开始配置 USART1：

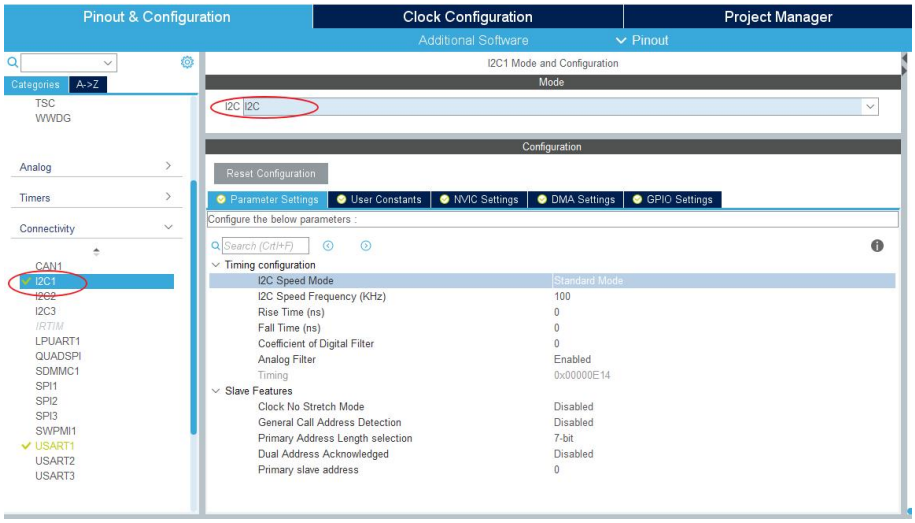


配置硬件 I2C

首先查看小熊派开发板的原理图，确定 EEPROM 接在哪个 I2C 接口上，如图：

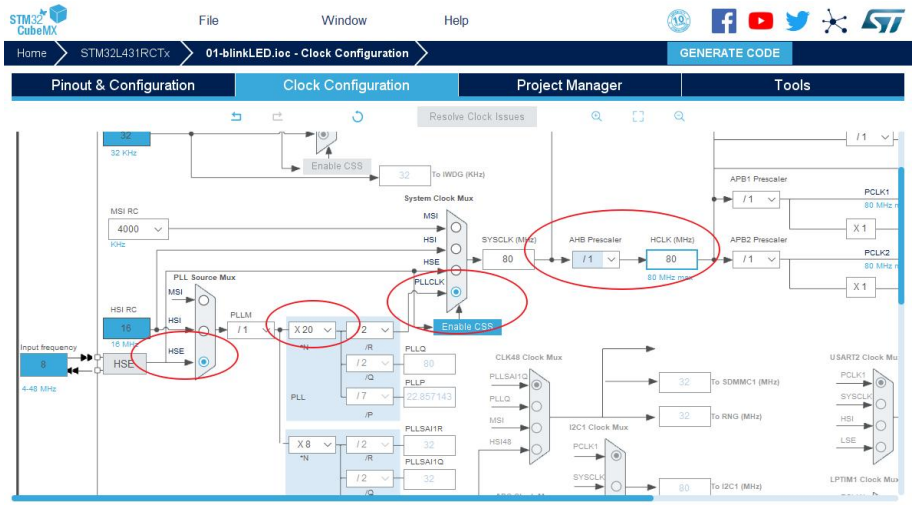


接下来开始配置 I2C 接口 1：

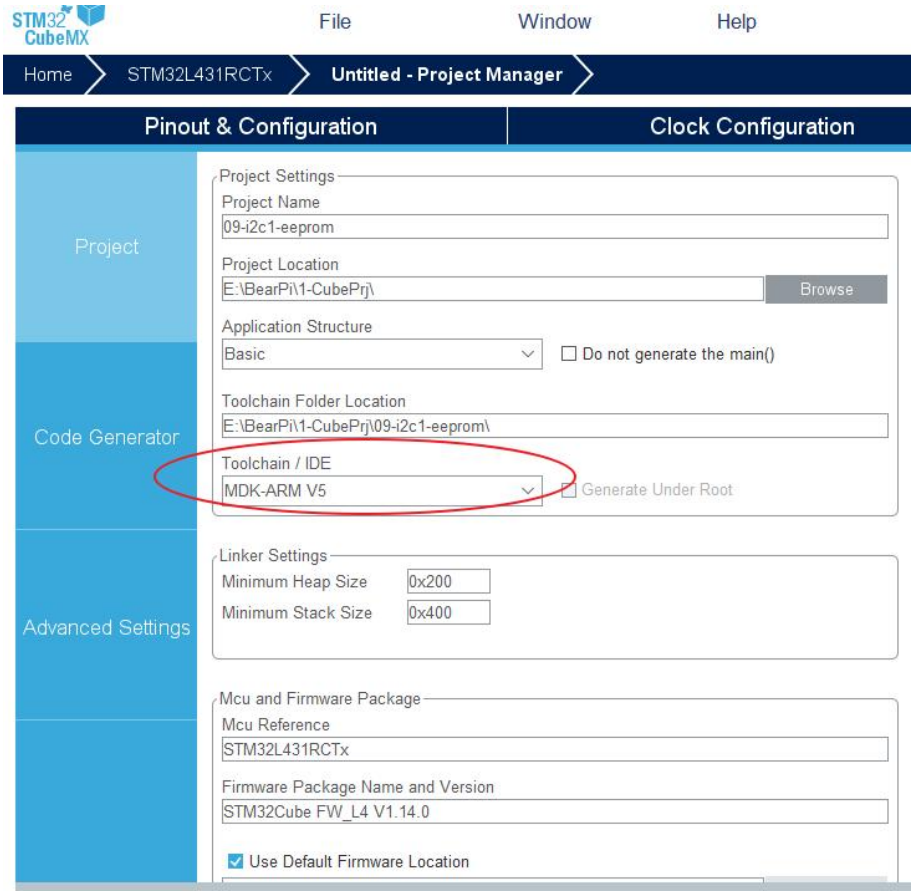


配置时钟树

STM32L4 的最高主频到 80M，所以配置 PLL，最后使 `HCLK = 80Mhz` 即可：

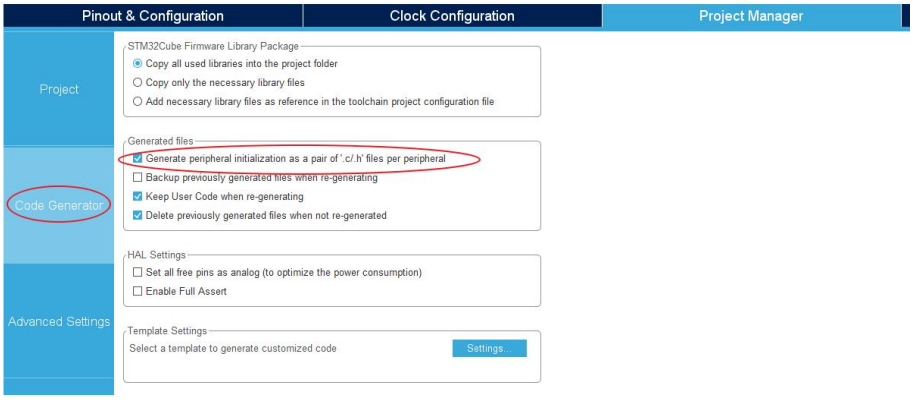


生成工程设置



代码生成设置

最后设置生成独立的初始化文件：



生成代码

点击 GENERATE CODE 即可生成 MDK-V5 工程：

3. 在 MDK 中编写、编译、下载用户代码

修改 I2C 初始化代码的小 BUG

```
i2c.c
61  void HAL_I2C_MspInit(I2C_HandleTypeDef* i2cHandle)
62  {
63
64    GPIO_InitTypeDef GPIO_InitStruct = {0};
65    if(i2cHandle->Instance==I2C1)
66    {
67    /* USER CODE BEGIN I2C1_MspInit 0 */
68
69    /* USER CODE END I2C1_MspInit 0 */
70
71      __HAL_RCC_GPIOB_CLK_ENABLE();
72      /**I2C1 GPIO Configuration
73      PB6      ------> I2C1_SCL
74      PB7      ------> I2C1_SDA
75      */
76      GPIO_InitStruct.Pin = GPIO_PIN_6|GPIO_PIN_7;
77      GPIO_InitStruct.Mode = GPIO_MODE_AF_OD;
78      GPIO_InitStruct.Pull = GPIO_PULLUP;
79      GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
80      GPIO_InitStruct.Alternate = GPIO_AF4_I2C1;
81      HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);
82
83      /* I2C1 clock enable */
84      __HAL_RCC_I2C1_CLK_ENABLE();
85    /* USER CODE BEGIN I2C1_MspInit 1 */
86
87    /* USER CODE END I2C1_MspInit 1 */
88    }
89  }
```

将I2C时钟使能代码移至GPIO初始化代码之前

重定向 printf( )函数

参考：重定向 printf 函数到串口输出的多种方法。

编写 EEPROM 驱动程序

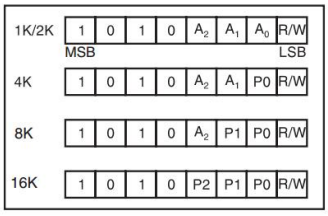
EEPROM 的驱动编写篇幅过多，单独分出来一节讲述。

4. AT24C02 驱动的编写

确定 IIC 器件地址

根据 AT24C02 的 Datasheet 可知 AT24C02 有 **2K bit，即 256B，分为 32 页，每页 8 个字节**，结合数据手册和原理图可以得知，板载 AT24C02 的读地址为 `0xA2`，写地址为 `0xA3`：

Figure 7. Device Address



首先在 `at24c02_i2c_drv.h` 中编写 AT24C02 相关的宏定义：

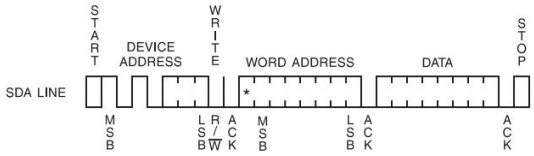
```
#define    AT24C02_ADDR_WRITE   0xA0#define          AT24C02_ADDR_READ    0xA1
```

然后在 at24c02_i2c_drv.c 中引入 i2c.h，基于 HAL 提供的硬件 IIC 操作函数，编写 AT24C02 的一些底层函数，如下。

任意地址写一个字节

根据 AT24C02 的数据手册可知，AT24C02 写一个字节的格式如下：

Figure 8. Byte Write



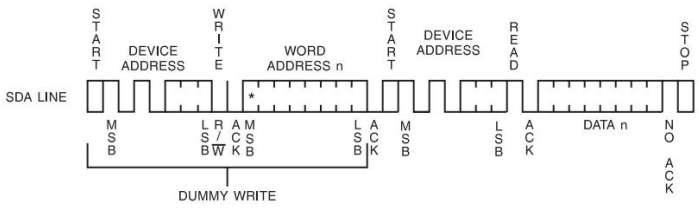
编写的函数如下：

```
/**
 * @brief          AT24C02 任意地址写一个字节数据
 * @param          addr —— 写数据的地址（0-255）
 * @param          dat  —— 存放写入数据的地址
 * @retval         成功 —— HAL_OK
*/
uint8_t At24c02_Write_Byte(uint16_t addr, uint8_t* dat){
          return HAL_I2C_Mem_Write(&hi2c1, AT24C02_ADDR_WRITE, addr, I2C_MEMADD_SIZE_8BIT, dat, 1, 0xFFFF);}
```

任意地址读一个字节

根据 AT24C02 的数据手册可知，AT24C02 读一个字节的格式如下：

Figure 11. Random Read



编写的函数如下：

```
/**
 * @brief          AT24C02 任意地址读一个字节数据
```

```
 * @param           addr —— 读数据的地址（0-255）
 * @param           read_buf —— 存放读取数据的地址
 * @retval          成功 —— HAL_OK
*/
uint8_t At24c02_Read_Byte(uint16_t addr, uint8_t* read_buf){
        return HAL_I2C_Mem_Read(&hi2c1, AT24C02_ADDR_READ, addr, I2C_MEMADD_SIZE_8BIT, read_buf, 1, 0xFFFF);}
```

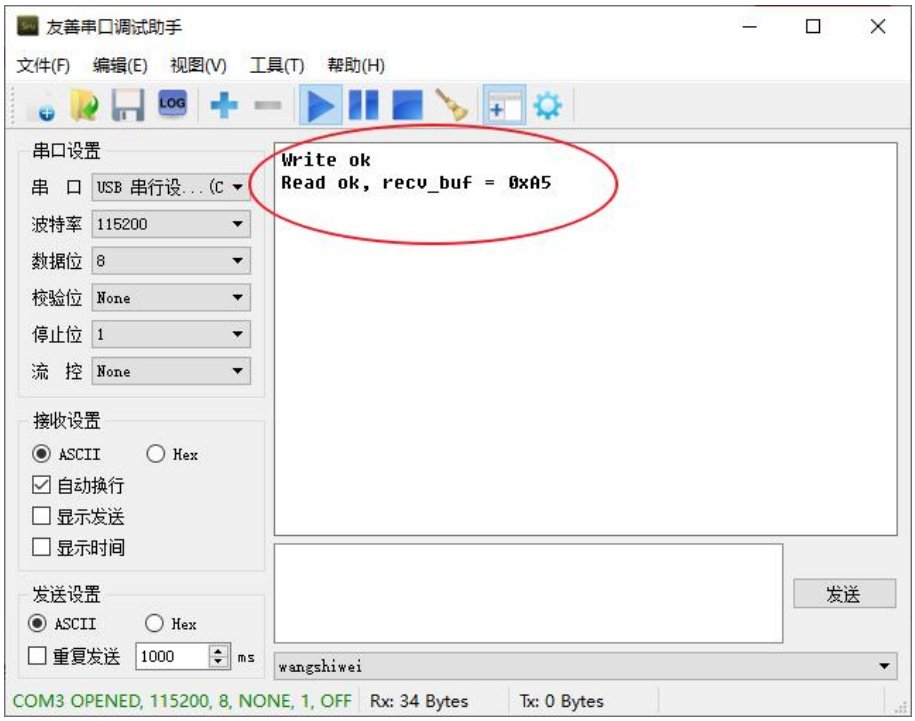
测试字节读写函数

在 `main.c` 中测试：

```
int main(void){
        uint8_t write_dat = 0xa5;
        uint8_t recv_buf = 0;

        HAL_Init();
        SystemClock_Config();
        MX_GPIO_Init();
        MX_I2C1_Init();
        MX_USART1_UART_Init();

        if(HAL_OK == At24c02_Write_Byte(10,&write_dat))
        {
                printf("Write ok\n");
        }
        else
        {
                printf("Write fail\n");
```

```
        }

        HAL_Delay(50);                  //写一次和读一次之间需要短暂的延时

        if(HAL_OK == At24c02_Read_Byte(10,&recv_buf))
        {
                printf("Read ok, recv_buf = 0x%02X\n", recv_buf);
        }
        else
        {
                printf("Read fail\n");
        }

        while(1);
```

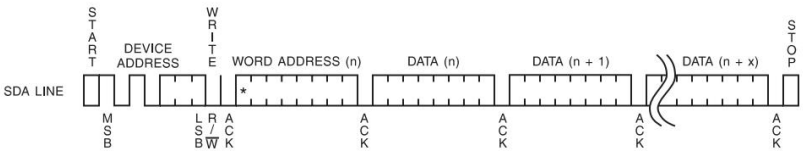
测试结果如下：



任意地址连续写多个字节

AT24C02 连续写字节的时候需要注意，**不能使用写单个字节函数连续的写入**，因为 AT24C02 分为了 32 页，每页是 8 个字节，如果连续的单字节写入 8 个字节后，会重复的继续往该页写数据，所以要使用如下的写一页的格式：

Figure 9. Page Write



(* = DON'T CARE bit for 1K)

```
/**
 * @brief           AT24C02 任意地址连续写多个字节数据
 * @param           addr —— 写数据的地址（0-255）
 * @param           dat  —— 存放写入数据的地址
 * @retval          成功 —— HAL_OK
 */
uint8_t At24c02_Write_Amount_Byte(uint16_t addr, uint8_t* dat, uint16_t size){
    uint8_t i = 0;
    uint16_t cnt = 0;                   //写入字节计数

    /* 对于起始地址，有两种情况，分别判断 */
    if(0 == addr % 8 )
    {
        /* 起始地址刚好是页开始地址 */

        /* 对于写入的字节数，有两种情况，分别判断 */
        if(size <= 8)
        {
```

```
            //写入的字节数不大于一页，直接写入
            return HAL_I2C_Mem_Write(&hi2c1, AT24C02_ADDR_WRITE, addr, I2C_MEMADD_SIZE_8BIT, dat, size, 0xFFFF);
        }
        else
        {
            //写入的字节数大于一页，先将整页循环写入
            for(i = 0;i < size/8; i++)
            {
                HAL_I2C_Mem_Write(&hi2c1, AT24C02_ADDR_WRITE, addr, I2C_MEMADD_SIZE_8BIT, &dat[cnt], 8, 0xFFFF);
                addr += 8;
                cnt += 8;
            }
            //将剩余的字节写入
            return HAL_I2C_Mem_Write(&hi2c1, AT24C02_ADDR_WRITE, addr, I2C_MEMADD_SIZE_8BIT, &dat[cnt], size - cnt, 0xFFFF);
        }
    }
    else
    {
        /* 起始地址偏离页开始地址 */
        /* 对于写入的字节数，有两种情况，分别判断 */
        if(size <= (8 - addr%8))
        {
            /* 在该页可以写完 */
```

```
            return HAL_I2C_Mem_Write(&hi2c1, AT24C02_ADDR_WRITE, addr, I2C_MEMADD_SIZE_8BIT,
dat, size, 0xFFFF);

        }

        else

        {

            /* 该页写不完 */

            //先将该页写完

            cnt += 8 - addr%8;

            HAL_I2C_Mem_Write(&hi2c1, AT24C02_ADDR_WRITE, addr, I2C_MEMADD_SIZE_8BIT, dat, cnt,
0xFFFF);

            addr += cnt;



            //循环写整页数据

            for(i = 0;i < (size - cnt)/8; i++)

            {

                HAL_I2C_Mem_Write(&hi2c1, AT24C02_ADDR_WRITE, addr, I2C_MEMADD_SIZE_8BIT,
&dat[cnt], 8, 0xFFFF);

                addr += 8;

                cnt += 8;

            }



            //将剩下的字节写入

            return HAL_I2C_Mem_Write(&hi2c1, AT24C02_ADDR_WRITE, addr, I2C_MEMADD_SIZE_8BIT,
&dat[cnt], size - cnt, 0xFFFF);

        }

    }}
```

任意地址连续读多个字节

AT24C02 连续读多个字节没有限制，直接读取即可，代码如下：

```
/**
 * @brief          AT24C02 任意地址连续读多个字节数据
 * @param          addr —— 读数据的地址（0-255）
 * @param          dat  —— 存放读出数据的地址
 * @retval         成功 —— HAL_OK
*/
uint8_t At24c02_Read_Amount_Byte(uint16_t addr, uint8_t* recv_buf, uint16_t size){
        return HAL_I2C_Mem_Read(&hi2c1, AT24C02_ADDR_READ, addr, I2C_MEMADD_SIZE_8BIT, recv_buf, size, 0xFFFF);}
```

测试任意地址连续读写多个字节

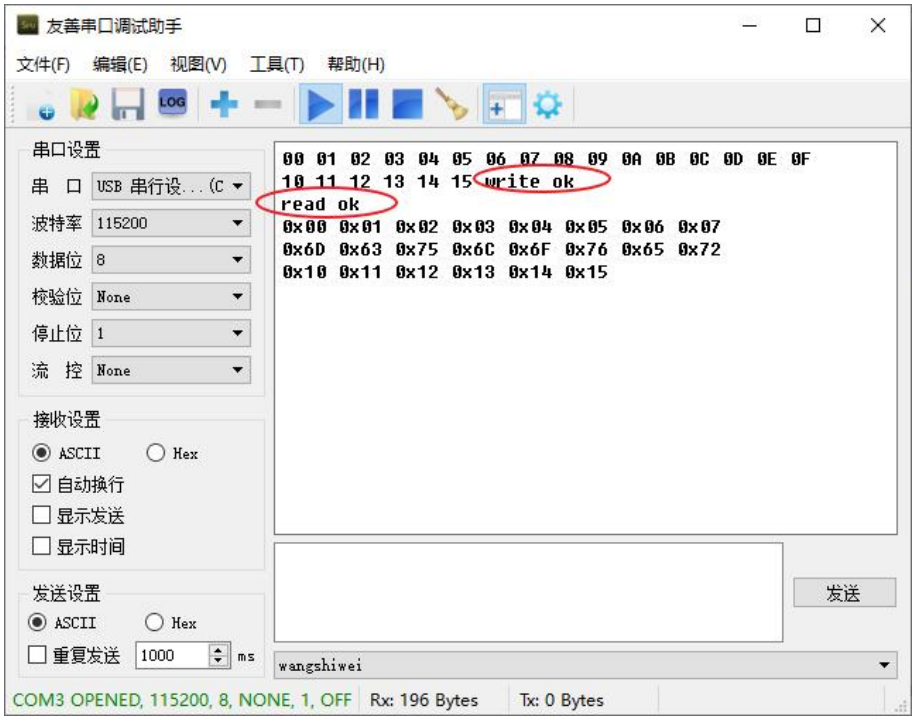
在 main.c 中测试：

```
int main(void){
        uint8_t write_dat[22] = {0};
        uint8_t recv_buf[22] = {0};

        HAL_Init();
        SystemClock_Config();
        MX_GPIO_Init();
        MX_I2C1_Init();
        MX_USART1_UART_Init();

        for(i = 0;i < 22; i++)
        {
                write_dat[i] = i;
                printf("%02X ", write_dat[i]);
```

```
                    if((i+1) % 16 == 0)
                    {
                        printf("\n");
                    }
            }
            if(HAL_OK == At24c02_Write_Amount_Byte(0, write_dat, 22))
            {
                    printf("write ok\n");
            }
            else
            {
                    printf("write fail\n");
            }
            HAL_Delay(50);
            if(HAL_OK == HAL_I2C_Mem_Read(&hi2c1, AT24C02_ADDR_READ, 0, I2C_MEMADD_SIZE_8BIT,
recv_buf, 22, 0xFFFF))
            {
                    printf("read ok\n");
                    for(i = 0; i < 22; i++)
                    {
                        printf("0x%02X ", recv_buf[i]);
                        if((i+1) % 8 == 0)
                        {
                            printf("\n");
                        }
```

```
                    }
                }
                else
                {
                    printf("read fail\n");
                }

                while(1);
```

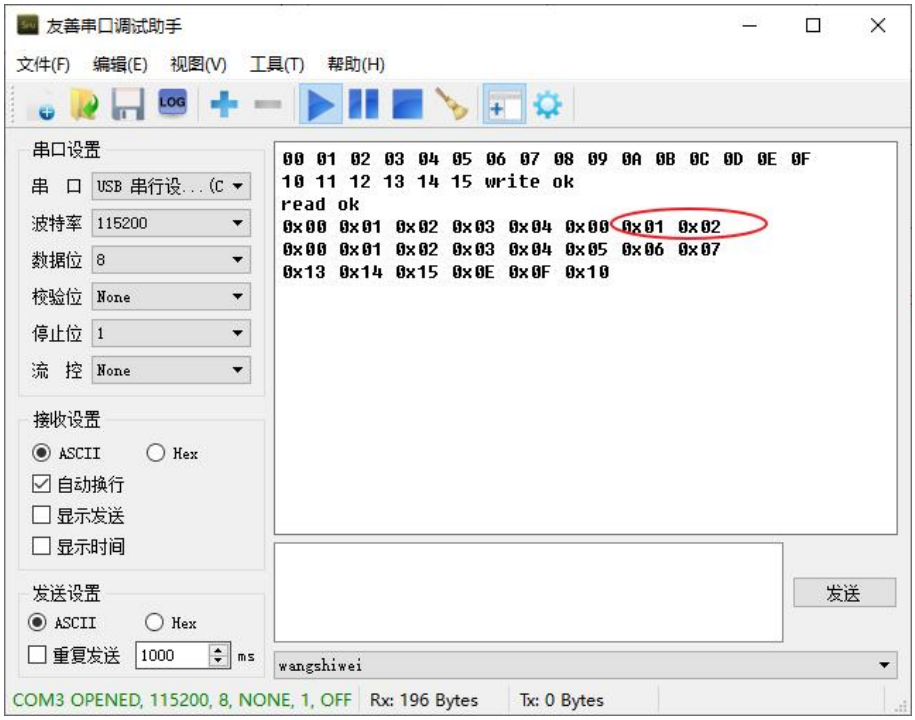
测试结果：



将上面的读写地址由 0 改为 5，再次测试：

```
if(HAL_OK == At24c02_Write_Amount_Byte(5, write_dat, 22))
```

测试结果：



至此，我们已经学会**如何使用硬件 IIC 接口读写 EEPROM**，下一节将讲述如何使用硬件 IIC 接口读取环境光强度传感器数据（BH1750）。

作业：

编写程序，通过 I2C 接口读写 AT24C02 EEPROM 中的数据。

分析 I2C 通信协议的工作原理和配置方法

STM32 单片机基础 14——使用硬件 I2C 读取环境光强度传感器数据（BH1750）

教学目的与要求：

目的：理解环境光强度传感器的应用原理，掌握通过 I2C 接口读取传感器数据的方法。

要求：能够配置 STM32 的硬件 I2C 接口，编写程序读取环境光强度传感器的数据，并将其转换为实际的光强度值。

教学重难点：

重点：I2C 接口的配置和环境光强度传感器的数据读取。

难点：理解环境光强度传感器数据的转换公式，确保测量的准确性。

课时数：3 课时

思政元素：

培养学生的环保意识，通过环境光强度传感器的应用，让学生意识到光环境对人类生活的重要性，以及如何通过技术手段监测和改善光环境

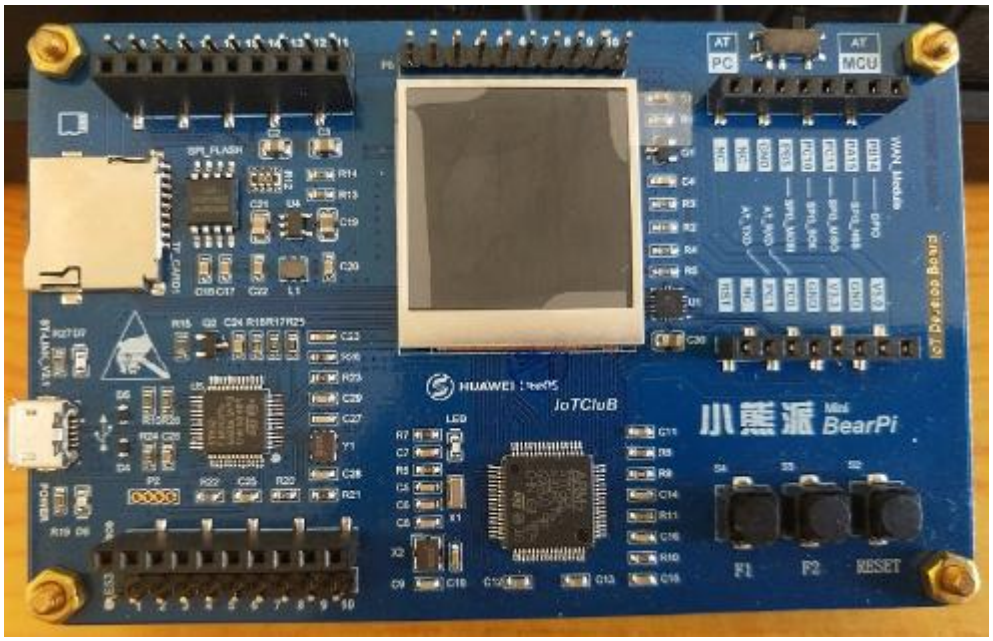
本篇详细的记录了如何使用 STM32CubeMX 配置 STM32L431RCT6 的硬件 I2C 外设读取环境光强度传感器数据（BH1750）。

1. 准备工作

硬件准备

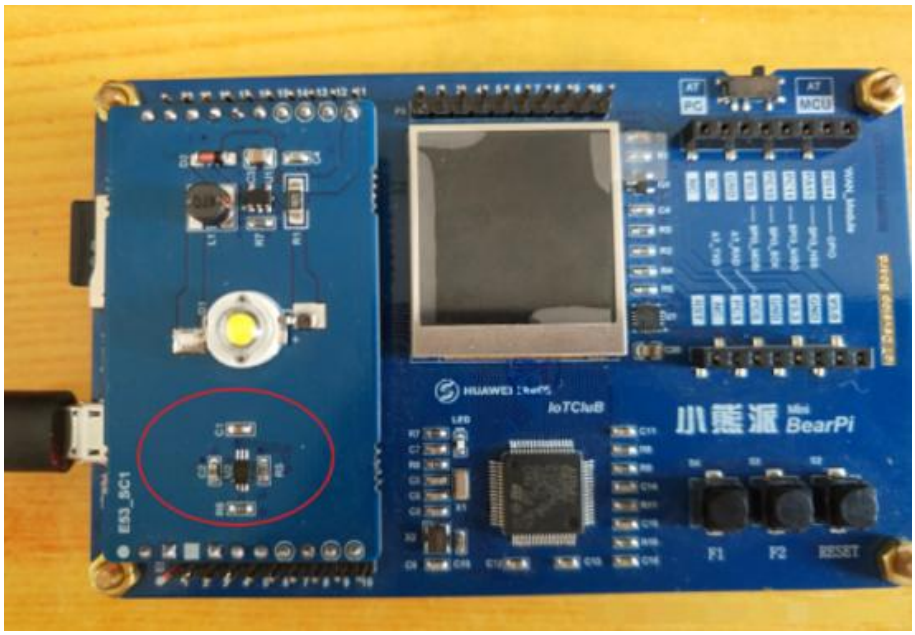
开发板

首先需要准备一个开发板，这里我准备的是 STM32L4 的开发板（BearPi）：

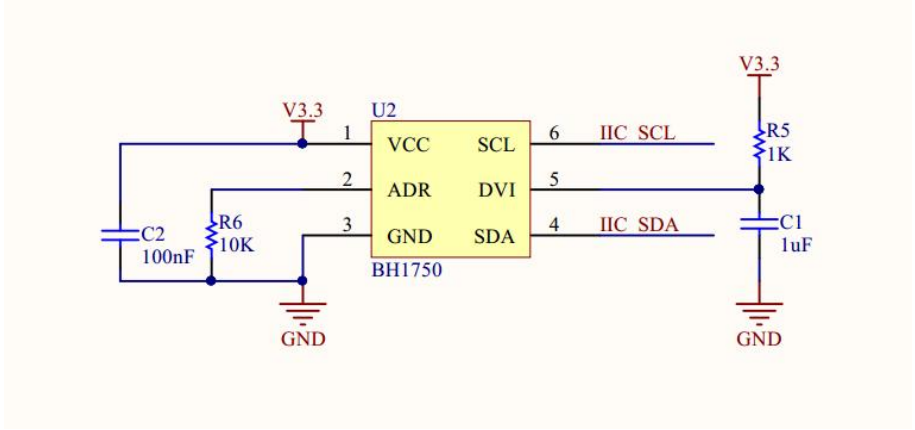


- BH1750 模块

BH1750FVI 是两线式串行总线接口（IIC）的 16 位数字输出型环境光强度传感器，利用它的高分辨率可以探测较大范围内的光照强度变化（1lx - 65535lx）。



BH1750 的原理图如下：



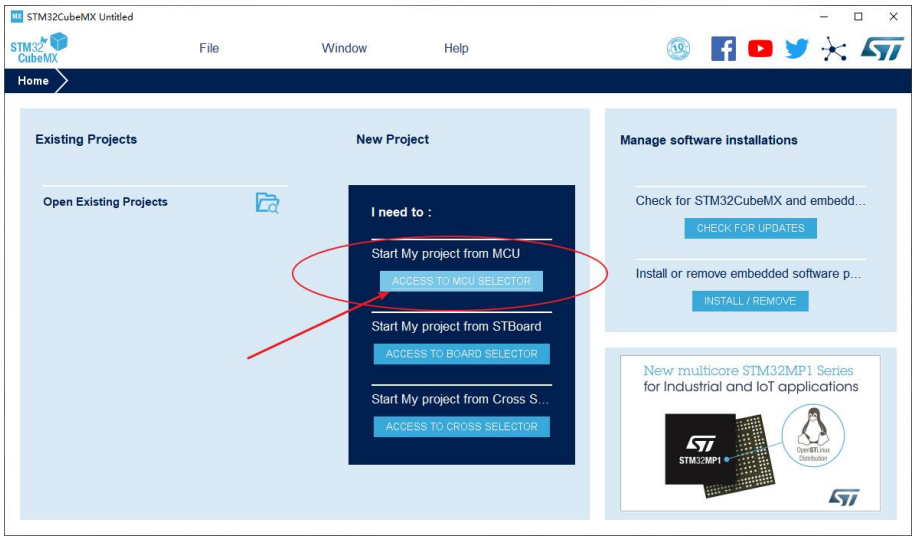
软件准备

- 需要安装好 Keil - MDK 及芯片对应的包，以便编译和下载生成的代码；
  Keil MDK 和串口助手的安装包都可以关注“**小熊派开源社区**”微信公众号，在资料教程一栏中可获取安装包。

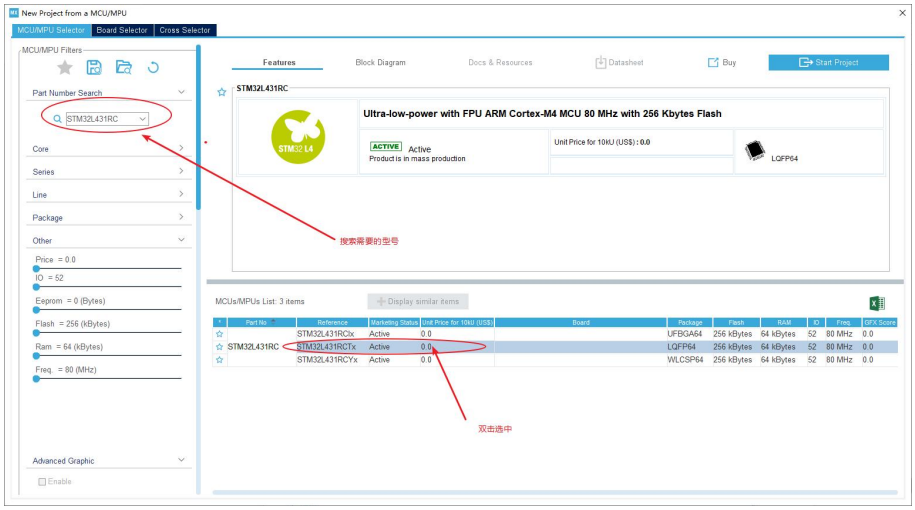
2.生成 MDK 工程

选择芯片型号

打开 STM32CubeMX，打开 MCU 选择器：



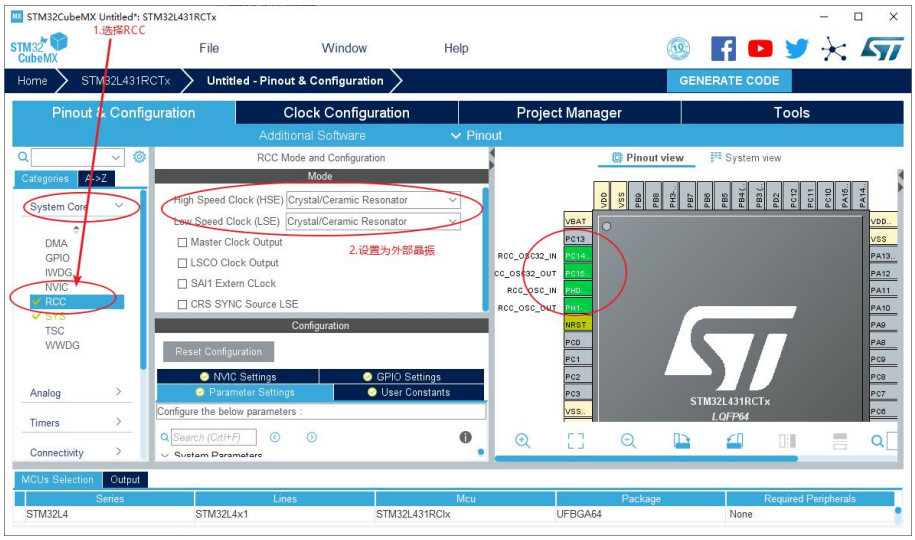
搜索并选中芯片 STM32L431RCT6:



配置时钟源

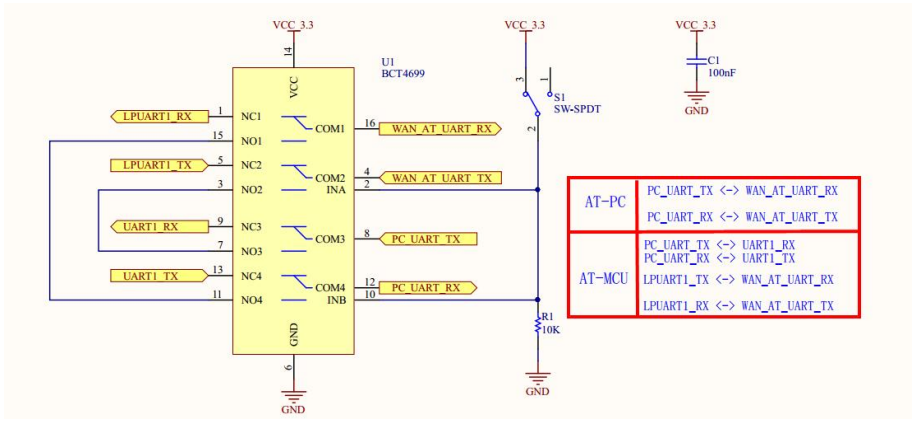
- 如果选择使用外部高速时钟（HSE），则需要在 System Core 中配置 RCC；
- 如果使用默认内部时钟（HSI），这一步可以略过；

这里我都使用外部时钟：



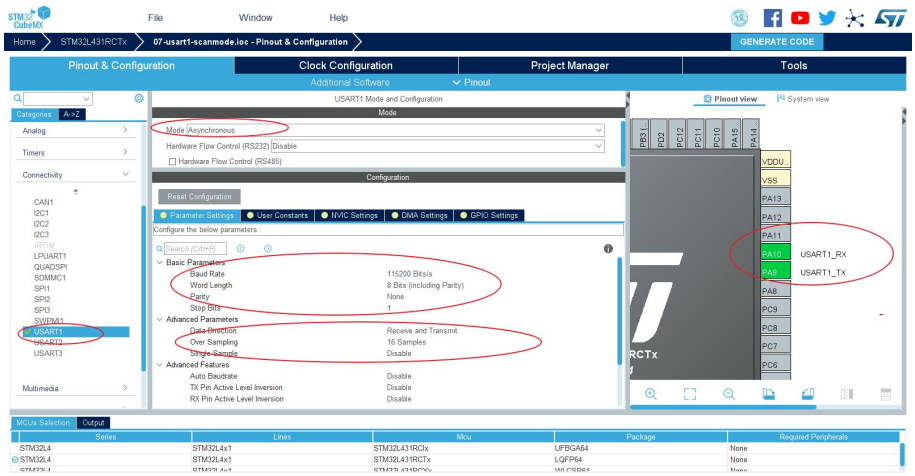
配置串口

小熊派开发板板载 ST-Link 并且虚拟了一个串口，原理图如下：



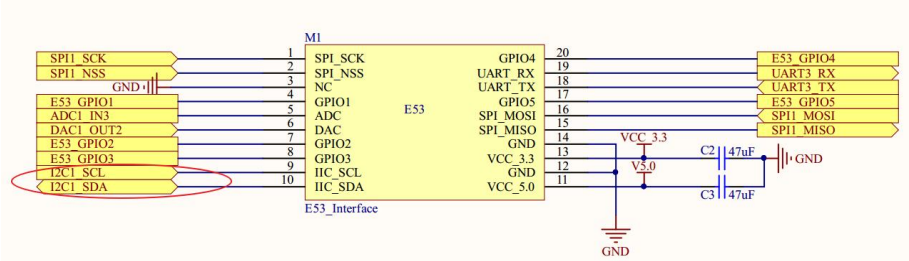
这里我将开关拨到 AT-MCU 模式，使 PC 的串口与 USART1 之间连接。

接下来开始配置 USART1：

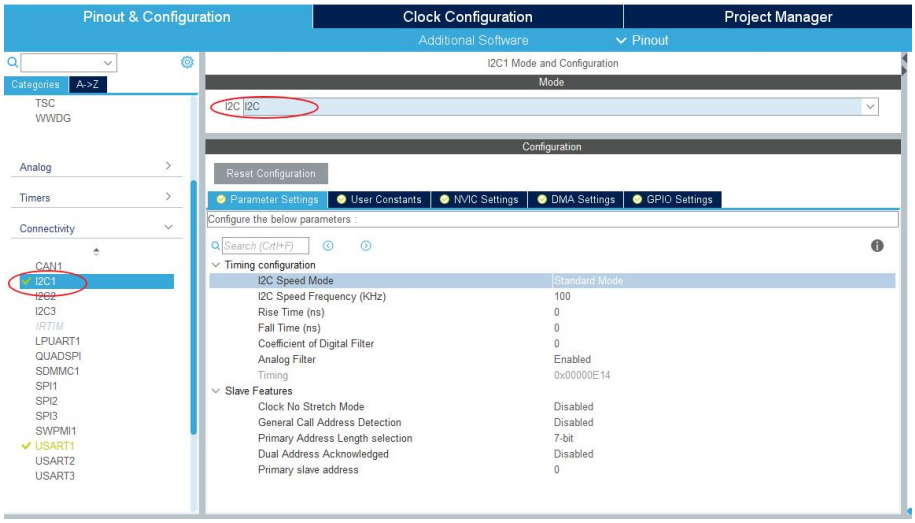


配置硬件 I2C

首先查看小熊派开发板的原理图，确定 EEPROM 接在哪个 I2C 接口上，如图：

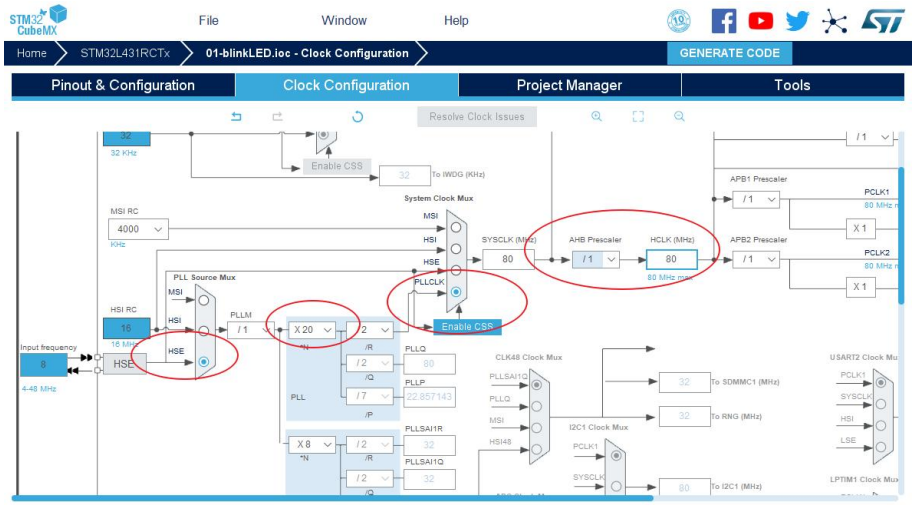


接下来开始配置 I2C 接口 1：

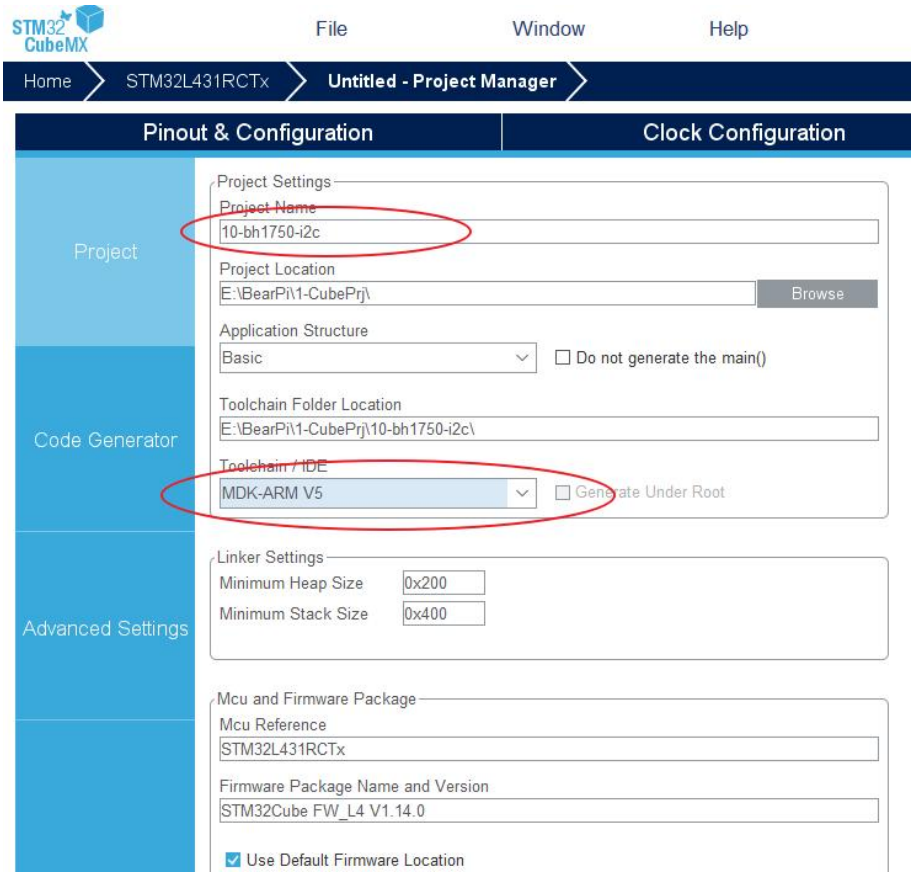


配置时钟树

STM32L4 的最高主频到 80M，所以配置 PLL，最后使 `HCLK = 80Mhz` 即可：

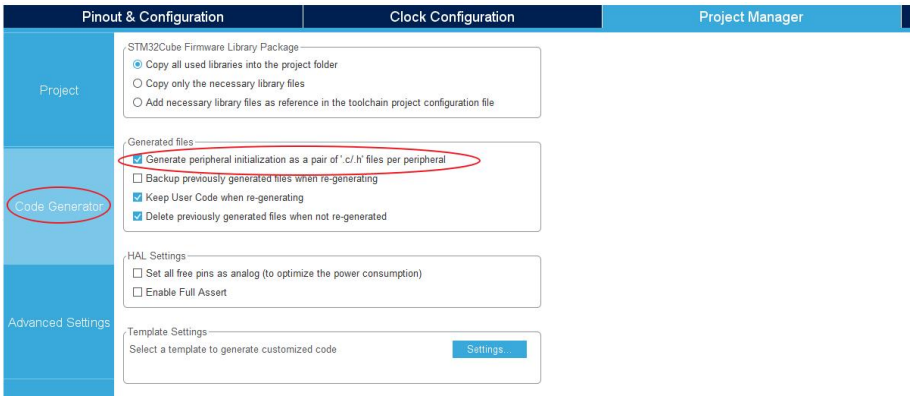


生成工程设置



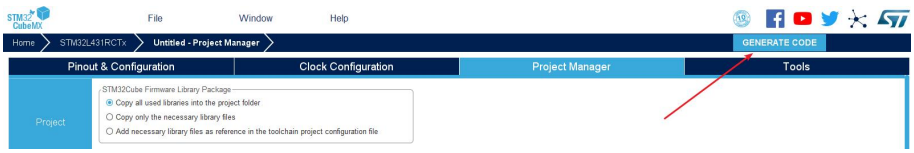
代码生成设置

最后设置生成独立的初始化文件：



生成代码

点击 GENERATE CODE 即可生成 MDK-V5 工程：



3. 在 MDK 中编写、编译、下载用户代码

修改 I2C 初始化代码的小 BUG

i2c.c

```
61  void HAL_I2C_MspInit(I2C_HandleTypeDef* i2cHandle)
62  {
63
64    GPIO_InitTypeDef GPIO_InitStruct = {0};
65    if(i2cHandle->Instance==I2C1)
66    {
67    /* USER CODE BEGIN I2C1_MspInit 0 */
68
69    /* USER CODE END I2C1_MspInit 0 */
70
71      __HAL_RCC_GPIOB_CLK_ENABLE();
72      /**I2C1 GPIO Configuration
73      PB6      ------> I2C1_SCL
74      PB7      ------> I2C1_SDA
75      */
76      GPIO_InitStruct.Pin = GPIO_PIN_6|GPIO_PIN_7;
77      GPIO_InitStruct.Mode = GPIO_MODE_AF_OD;
78      GPIO_InitStruct.Pull = GPIO_PULLUP;
79      GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
80      GPIO_InitStruct.Alternate = GPIO_AF4_I2C1;
81      HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);
82
83      /* I2C1 clock enable */
84      __HAL_RCC_I2C1_CLK_ENABLE();
85    /* USER CODE BEGIN I2C1_MspInit 1 */
86
87    /* USER CODE END I2C1_MspInit 1 */
88    }
89  }
```

将I2C时钟使能代码移至GPIO初始化代码之前

重定向 printf( )函数

参考：重定向 printf 函数到串口输出的多种方法。

4. 编写 BH1750 驱动程序

参考 bh1750FVI 中文数据手册.pdf 进行编程。

宏定义 BH1750 器件地址

BH1750 的器件地址由 ADDR 端口的高低电平决定：

从属地址有 2 中形式，由 ADDR 端口决定。

ADDR= “H” (ADDR ≧ 0.7VCC) → “1011100”

ADDR= “L” (ADDR ≦ 0.3VCC) → “0100011”

结合原理图，在 bh1750_i2c_drv.h 头文件中可以定义如下：

```
#define    BH1750_ADDR_WRITE    0x46      //01000110#define    BH1750_ADDR_READ    0x47
          //01000111
```

枚举 BH1750 工作模式

参考数据手册在 bh1750_i2c_drv.h 头文件中进行如下枚举定义：

```
typedef enum{

        POWER_OFF_CMD       =       0x00,       //断电：无激活状态

        POWER_ON_CMD        =       0x01,       //通电：等待测量指令

        RESET_REGISTER      =       0x07,       //重置数字寄存器（在断电状态下不起作用）

        CONT_H_MODE                 =       0x10,       //连续 H 分辨率模式：在 1lx 分辨率下开始测量，测量时间 120ms

        CONT_H_MODE2        =       0x11,       //连续 H 分辨率模式 2：在 0.5lx 分辨率下开始测量，测量时间 120ms

        CONT_L_MODE                 =       0x13,       //连续 L 分辨率模式：在 4lx 分辨率下开始测量，测量时间 16ms

        ONCE_H_MODE                 =       0x20,       //一次高分辨率模式：在 1lx 分辨率下开始测量，测量时间 120ms，测量后自动设置为断电模式

        ONCE_H_MODE2        =       0x21,       //一次高分辨率模式 2：在 0.5lx 分辨率下开始测量，测量时间 120ms，测量后自动设置为断电模式

        ONCE_L_MODE                 =       0x23        //一次低分辨率模式：在 4lx 分辨率下开始测量，测量时间 16ms，测量后自动设置为断电模式} BH1750_MODE;
```

发送命令和读取数据

接下来编写 bh1750_i2c_drv.c 驱动文件，参考数据手册中的这部分：

本驱动程序底层使用 HAL 库的 IIC 初始化文件，所以包含如下头文件：

```
#include "bh1750_i2c_drv.h"#include "i2c.h"
```

根据上图，发送命令的函数如下：

```
/**
```

```
 * @brief 向BH1750发送一条指令
 * @param cmd —— BH1750工作模式指令（在BH1750_MODE中枚举定义）
 * @retval成功返回HAL_OK
*/
uint8_t    BH1750_Send_Cmd(BH1750_MODE cmd){
          return HAL_I2C_Master_Transmit(&hi2c1, BH1750_ADDR_WRITE, (uint8_t*)&cmd, 1, 0xFFFF);}
```

接收光照强度数据的函数如下：

```
/**
 * @brief 从BH1750接收一次光强数据
 * @param dat —— 存储光照强度的地址（两个字节数组）
 * @retval成功 —— 返回HAL_OK
*/
uint8_t BH1750_Read_Dat(uint8_t* dat){
          return HAL_I2C_Master_Receive(&hi2c1, BH1750_ADDR_READ, dat, 2, 0xFFFF);}
```

数据转换函数

根据数据手册中给出的公式，编写将从BH1750读出的两个字节数据转换为对应强度值的函数：

```
/**
 * @brief 将BH1750的两个字节数据转换为光照强度值（0-65535）
 * @param dat  —— 存储光照强度的地址（两个字节数组）
 * @retval成功 —— 返回光照强度值
*/
uint16_t BH1750_Dat_To_Lux(uint8_t* dat){
          uint16_t lux = 0;
          lux = dat[0];
          lux <<= 8;
```

```
        lux += dat[1];
        lux = (int)(lux / 1.2);

        return lux;}
```

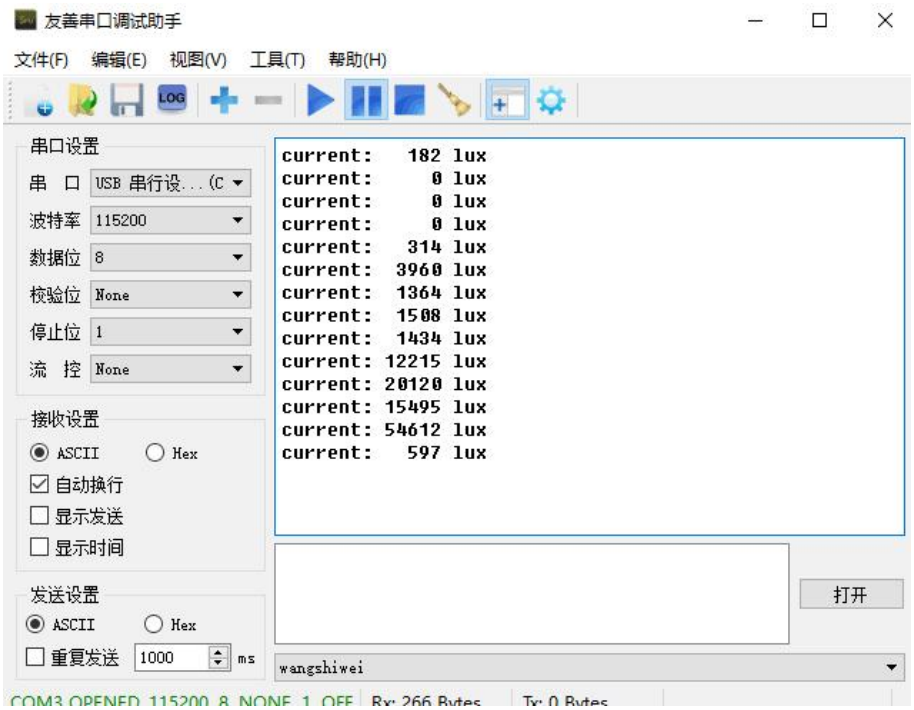
5. 测试驱动程序

在 main.c 中测试驱动程序是否正常：

```
int main(void){
    uint8_t dat[2] = {0};                //dat[0]是高字节，dat[1]是低字节

    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    MX_I2C1_Init();
    MX_USART1_UART_Init();

    while (1)
    {
        if(HAL_OK == BH1750_Send_Cmd(ONCE_H_MODE))
        {
            //printf("send ok\n");
        }
        else
        {
            //printf("send fail\n");
        }
```

```
        HAL_Delay(200);
        if(HAL_OK == BH1750_Read_Dat(dat))
        {
            //printf("recv ok\n");
            printf("current: %5d lux\n", BH1750_Dat_To_Lux(dat));

        }
        else
        {
            //printf("recv fail");
        }

        HAL_Delay(1000);
    }}
```

编译下载运行，测试结果如下：



至此，我们已经学会**如何使用硬件 IIC 接口读取环境光强度传感器数据（BH1750）**，下一节将讲述如何使用硬件 IIC 接口读取温湿度传感器数据并使用软件 CRC 校验（SHT30）。

作业：

编写程序，通过 I2C 接口读取 BH1750 环境光强度传感器的数据，并处理显示。

分析环境光强度传感器的测量原理和应用场景。

STM32 单片机基础 15——使用硬件 I2C 读取温湿度传感器数据（SHT30）

教学目的与要求：

目的：掌握温湿度传感器的数据读取方法，理解温湿度数据的获取和处理过程。

要求：能够配置 STM32 的硬件 I2C 接口，编写程序读取 SHT30 温湿度传感器的数据，并进行适当的处理。

教学重难点：

重点：I2C 接口的配置和 SHT30 温湿度传感器的数据读取。

难点：理解温湿度数据的校正方法，确保测量结果的准确性。

课时数：3 课时

思政元素：

培养学生的科学探索精神，通过温湿度传感器的应用，让学生意识到环境监测在日常生活和科学研究中的重要性，鼓励学生关注环境变化，培养环保意识。

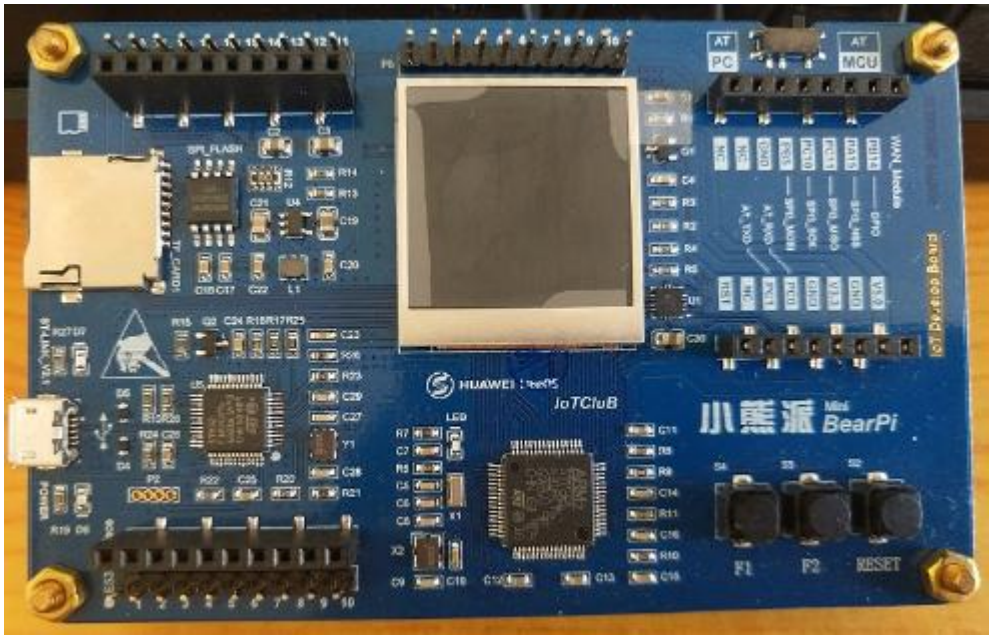
本篇详细的记录了如何使用 STM32CubeMX 配置 STM32L431RCT6 的硬件 I2C 外设，读取 SHT30 温湿度传感器的数据并通过串口发送。

1. 准备工作

硬件准备

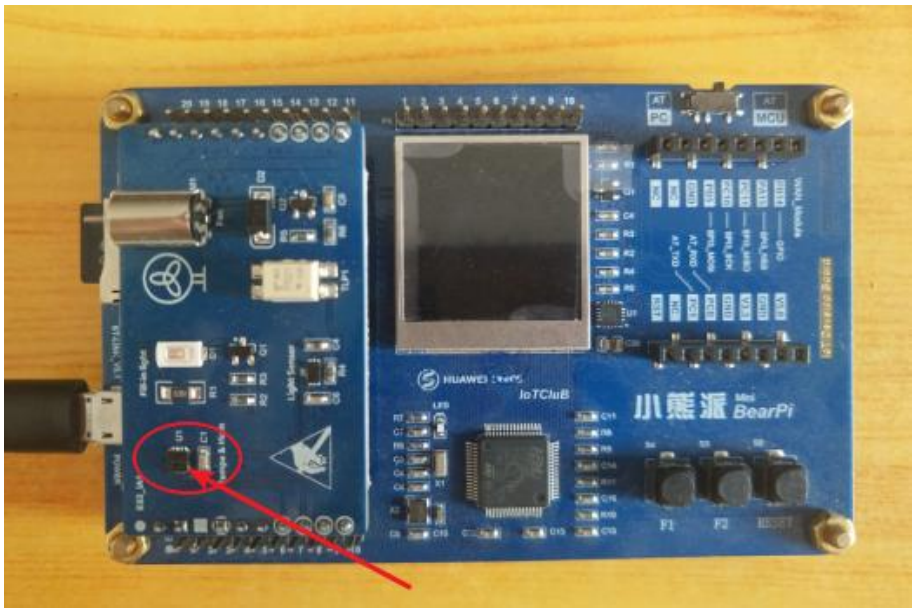
开发板

首先需要准备一个开发板，这里我准备的是 STM32L4 的开发板（BearPi）：

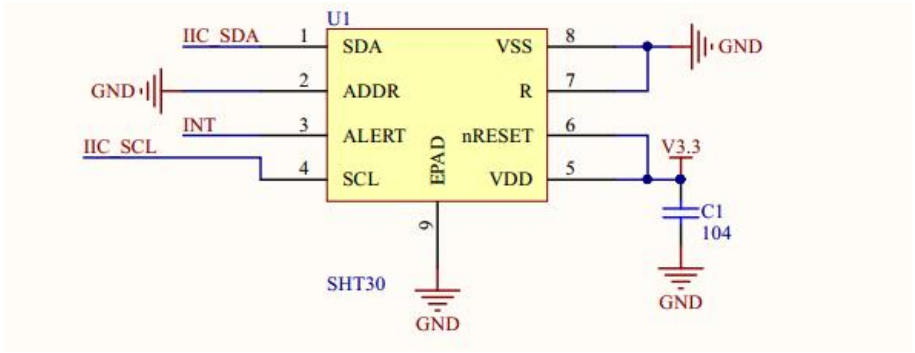


- SHT30 温湿度传感器

SHT30 温湿度传感器是一个完全校准的、现行的、带有温度补偿的**数字输出型**传感器，具有 2.4V-5.5V 的宽电压支持，使用 IIC 接口进行通信，最高速率可达 1M 并且有两个用户可选地址，除此之外，它还具有 8 个引脚的 DFN 超小封装，如图：



SHT30 的原理图如下：



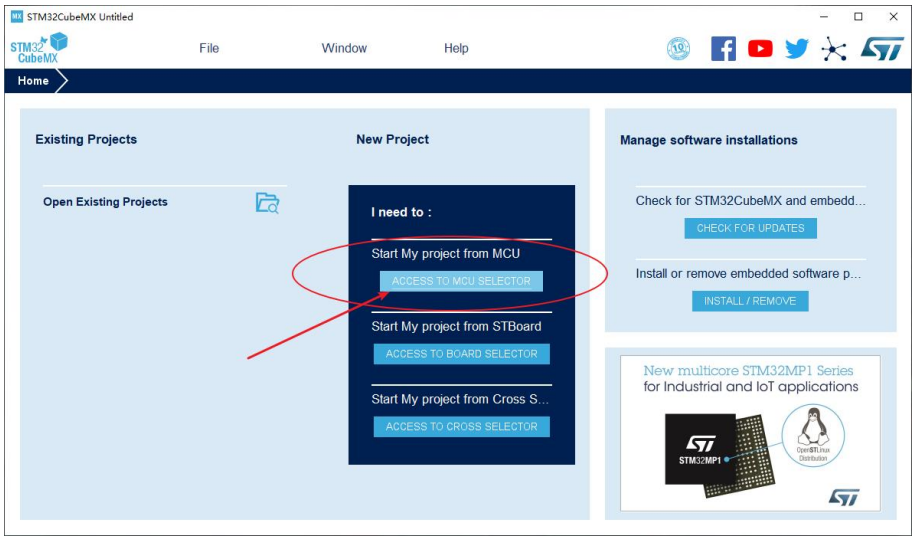
软件准备

- 需要安装好 Keil - MDK 及芯片对应的包，以便编译和下载生成的代码；
  Keil MDK 和串口助手的安装包都可以关注“**小熊派开源社区**”微信公众号，在资料教程一栏中可获取安装包。

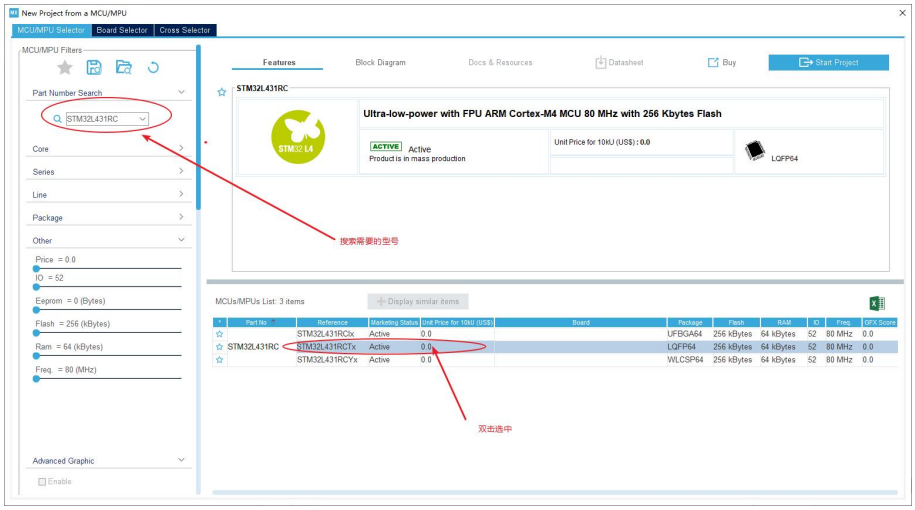
2.生成 MDK 工程

选择芯片型号

打开 STM32CubeMX，打开 MCU 选择器：



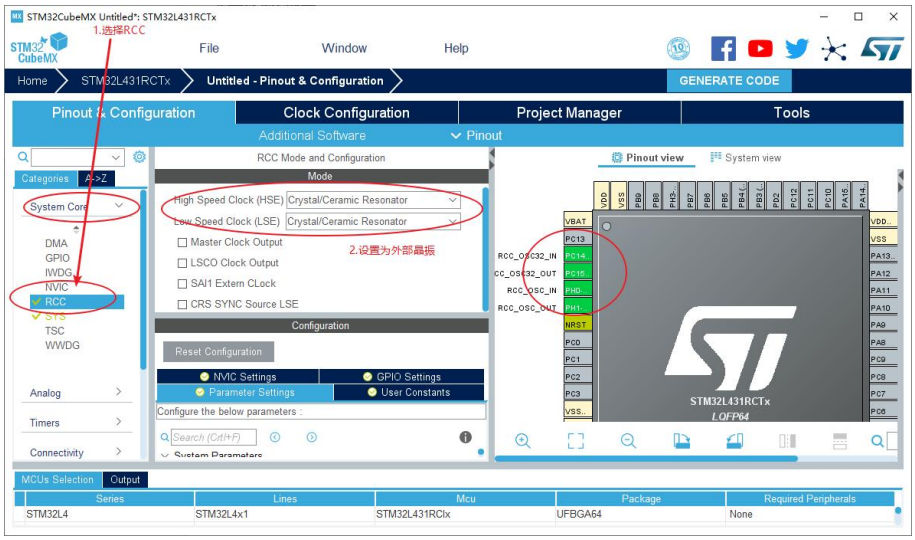
搜索并选中芯片 STM32L431RCT6:



配置时钟源

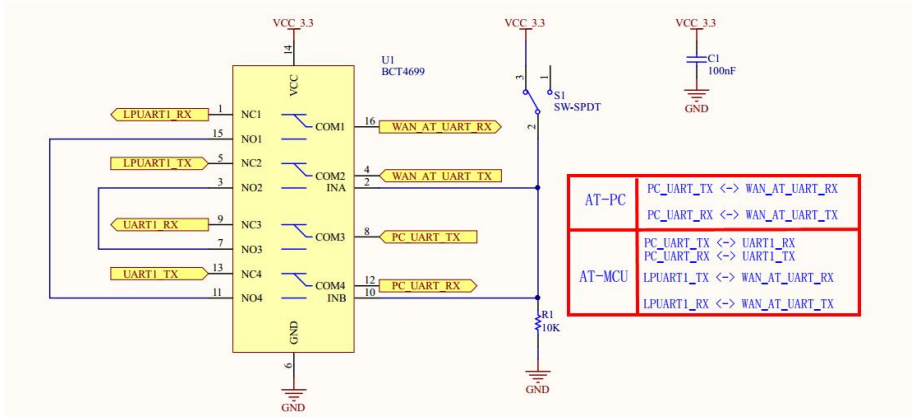
- 如果选择使用外部高速时钟（HSE），则需要在 System Core 中配置 RCC；
- 如果使用默认内部时钟（HSI），这一步可以略过；

这里我都使用外部时钟：



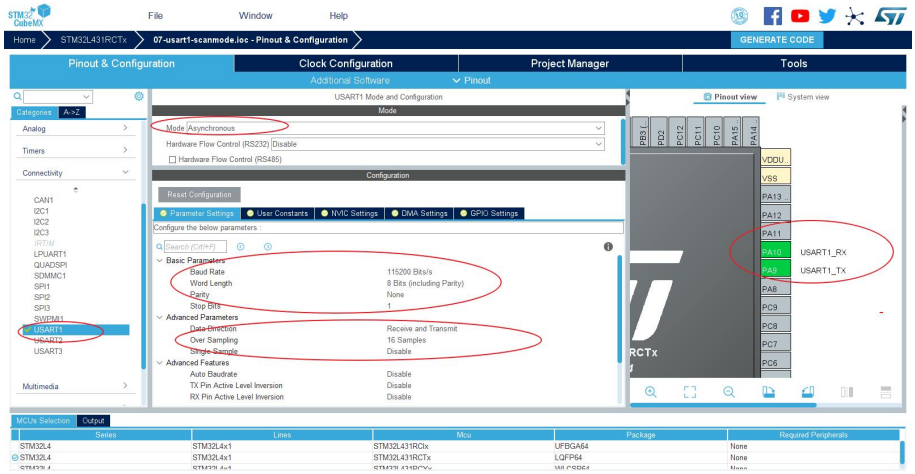
配置串口

小熊派开发板板载 ST-Link 并且虚拟了一个串口，原理图如下：



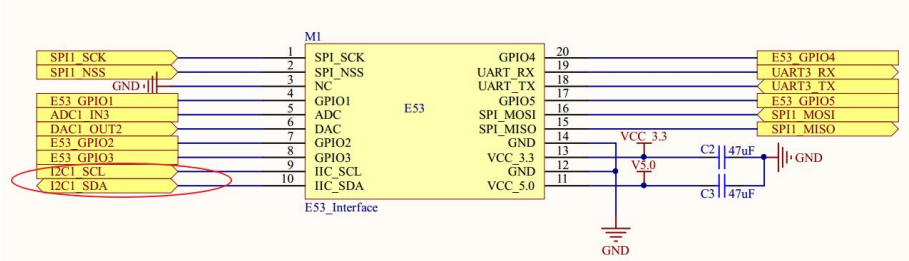
这里我将开关拨到 AT-MCU 模式，使 PC 的串口与 USART1 之间连接。

接下来开始配置 USART1：

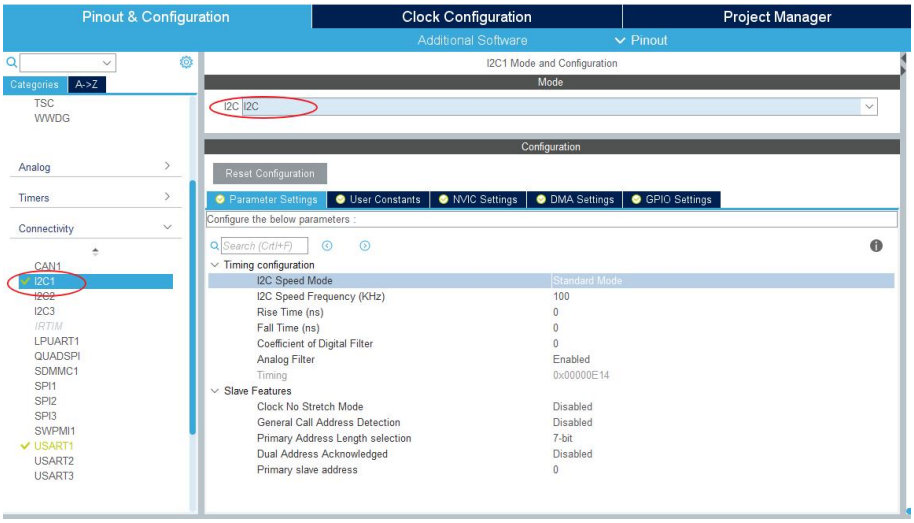


配置 I2C 接口

查看小熊派 E53 接口的原理图：

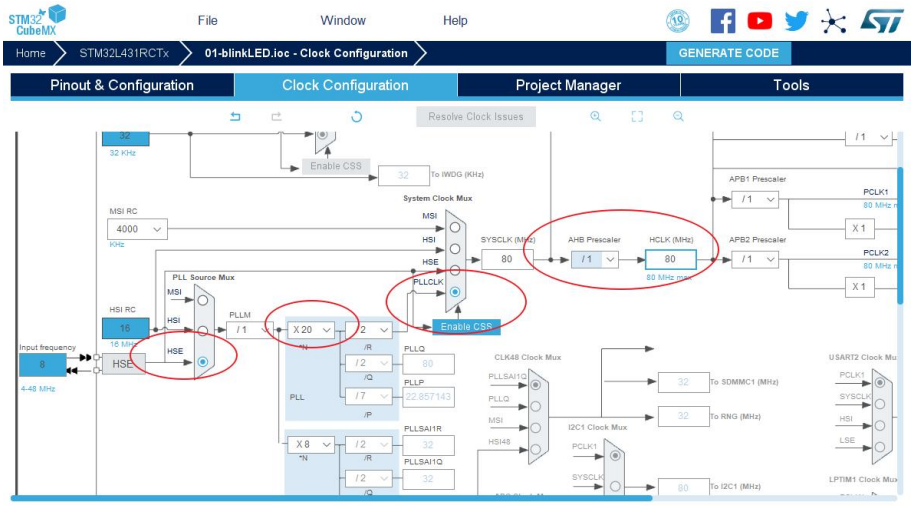


接下来开始配置 I2C 接口 1：

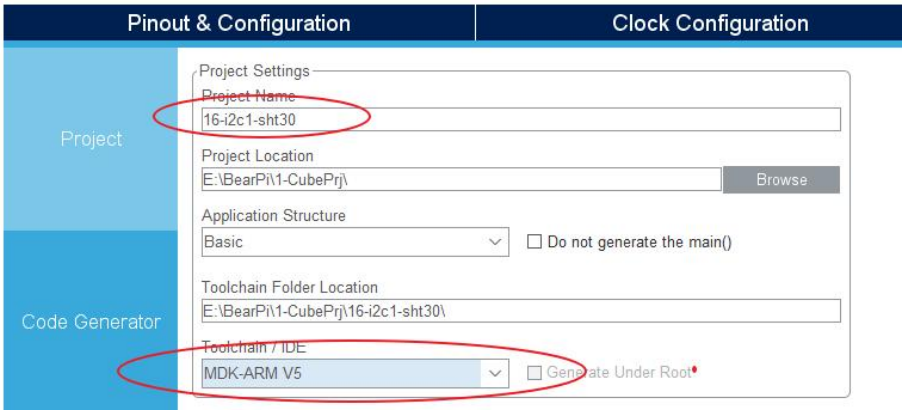


配置时钟树

STM32L4 的最高主频到 80M，所以配置 PLL，最后使 `HCLK = 80Mhz` 即可：

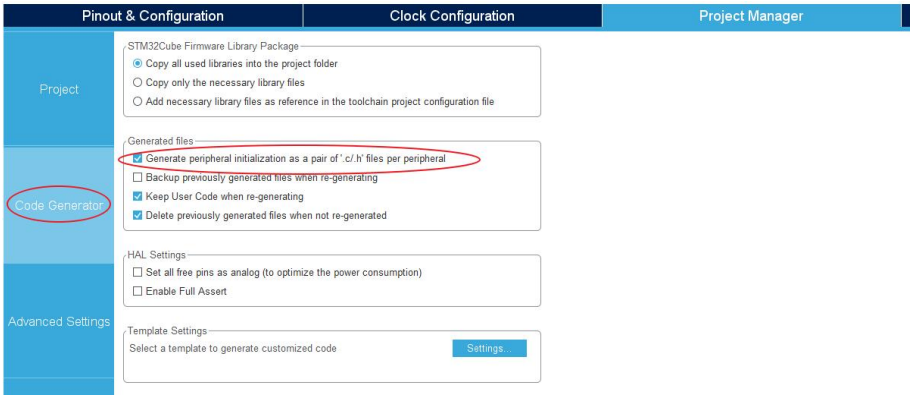


生成工程设置



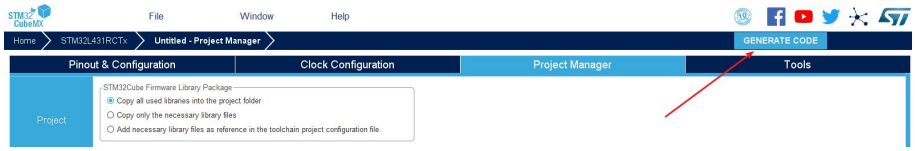
代码生成设置

最后设置生成独立的初始化文件：



生成代码

点击 GENERATE CODE 即可生成 MDK-V5 工程：



3. 在 MDK 中编写、编译、下载用户代码

重定向 printf( )函数

参考：重定向 printf 函数到串口输出的多种方法。

修改 I2C 初始化代码的小 BUG

i2c.c

```c
void HAL_I2C_MspInit(I2C_HandleTypeDef* i2cHandle)
{

  GPIO_InitTypeDef GPIO_InitStruct = {0};
  if(i2cHandle->Instance==I2C1)
  {
  /* USER CODE BEGIN I2C1_MspInit 0 */

  /* USER CODE END I2C1_MspInit 0 */

    __HAL_RCC_GPIOB_CLK_ENABLE();
    /**I2C1 GPIO Configuration
    PB6      ------> I2C1_SCL
    PB7      ------> I2C1_SDA
    */
    GPIO_InitStruct.Pin = GPIO_PIN_6|GPIO_PIN_7;
    GPIO_InitStruct.Mode = GPIO_MODE_AF_OD;
    GPIO_InitStruct.Pull = GPIO_PULLUP;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
    GPIO_InitStruct.Alternate = GPIO_AF4_I2C1;
    HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);

    /* I2C1 clock enable */
    __HAL_RCC_I2C1_CLK_ENABLE();
  /* USER CODE BEGIN I2C1_MspInit 1 */

  /* USER CODE END I2C1_MspInit 1 */
  }
}
```

将I2C时钟使能代码移至GPIO初始化代码之前

4. 编写 SHT30 驱动程序

参考 SHT30 数据手册.pdf 进行编程。

宏定义 SHT30 器件地址

先来编写 sht30_i2c_drv.h 头文件，SHT30 的器件地址由 ADDR 端口的高低电平决定：

SHT3x-DIS	I2C Address in Hex. representation	Condition
I2C address A	0x44 (default)	ADDR (pin 2) connected to VSS
I2C address B	0x45	ADDR (pin 2) connected to VDD

注意数据手册中给出了 8 位数据，只有低 7 位用作地址，结合原理图，可以定义如下：

```c
/* ADDR Pin Conect to VSS */

#define    SHT30_ADDR_WRITE     0x44<<1          //10001000#define        SHT30_ADDR_READ          (0x44<<1)+1
```

枚举 SHT30 命令列表

参考数据手册，在 sht30_i2c_drv.h 头文件中给出如下枚举定义：

```c
typedef enum{
```

```
    /* 软件复位命令 */

    SOFT_RESET_CMD = 0x30A2,
    /*
    单次测量模式
    命名格式：Repeatability_CS_CMD
    CS:  Clock stretching
    */
    HIGH_ENABLED_CMD     = 0x2C06,
    MEDIUM_ENABLED_CMD  = 0x2C0D,
    LOW_ENABLED_CMD      = 0x2C10,
    HIGH_DISABLED_CMD    = 0x2400,
    MEDIUM_DISABLED_CMD = 0x240B,
    LOW_DISABLED_CMD     = 0x2416,

    /*
    周期测量模式
    命名格式：Repeatability_MPS_CMD
    MPS：measurement per second
    */
    HIGH_0_5_CMD    = 0x2032,
    MEDIUM_0_5_CMD = 0x2024,
    LOW_0_5_CMD     = 0x202F,
    HIGH_1_CMD      = 0x2130,
    MEDIUM_1_CMD    = 0x2126,
```

```
    LOW_1_CMD       = 0x212D,
    HIGH_2_CMD      = 0x2236,
    MEDIUM_2_CMD    = 0x2220,
    LOW_2_CMD       = 0x222B,
    HIGH_4_CMD      = 0x2334,
    MEDIUM_4_CMD    = 0x2322,
    LOW_4_CMD       = 0x2329,
    HIGH_10_CMD     = 0x2737,
    MEDIUM_10_CMD   = 0x2721,
    LOW_10_CMD      = 0x272A,
          /* 周期测量模式读取数据命令 */
          READOUT_FOR_PERIODIC_MODE = 0xE000,} SHT30_CMD;
```

发送命令函数

```
/**
 * @brief 向SHT30发送一条指令(16bit)
 * @param cmd —— SHT30指令（在SHT30_MODE中枚举定义）
 * @retval成功返回HAL_OK
*/static uint8_t     SHT30_Send_Cmd(SHT30_CMD cmd){
    uint8_t cmd_buffer[2];
    cmd_buffer[0] = cmd >> 8;
    cmd_buffer[1] = cmd;
    return HAL_I2C_Master_Transmit(&hi2c1, SHT30_ADDR_WRITE, (uint8_t* cmd_buffer, 2, 0xFFFF);}
```

复位函数

```
/**
 * @brief 复位SHT30
```

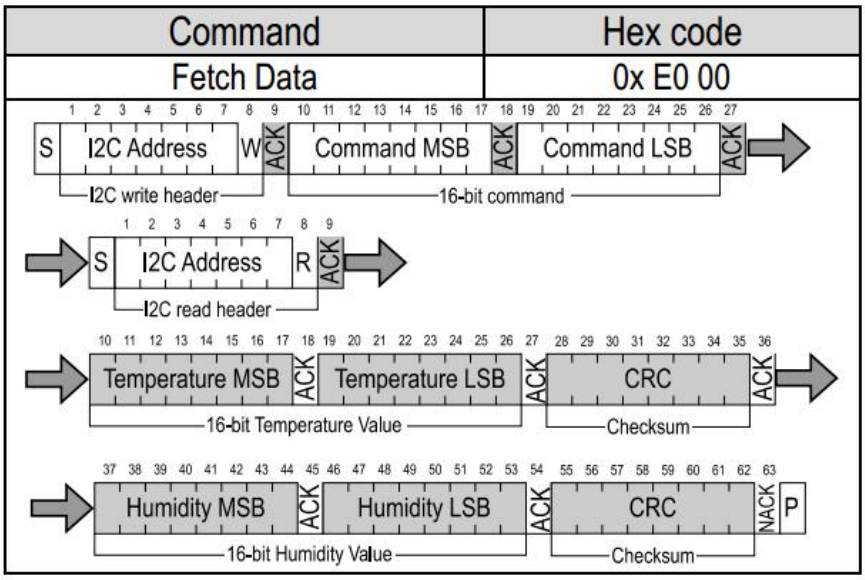
```
 * @param none
 * @retval none
*/void SHT30_reset(void){
    SHT30_Send_Cmd(SOFT_RESET_CMD);
    HAL_Delay(20);}
```

SHT30 工作模式初始化函数（周期测量模式）

```
/**
 * @brief 初始化 SHT30
 * @param none
 * @retval 成功返回 HAL_OK
 * @note  周期测量模式
*/
uint8_t SHT30_Init(void){
    return SHT30_Send_Cmd(MEDIUM_2_CMD);}
```

从 SHTY30 读取一次数据（周期测量模式下）

从 SHT30 数据手册中可以得到在周期测量模式下读取一次数据的时序，如图：



根据该时序可以看出，首先要发送读数据的命令，然后接收 6 个字节的数据，编写程序如下：

```
/**
 * @brief 从 SHT30 读取一次数据
 * @param dat —— 存储读取数据的地址（6 个字节数组）
 * @retval成功 —— 返回 HAL_OK
*/
uint8_t SHT30_Read_Dat(uint8_t* dat){
        SHT30_Send_Cmd(READOUT_FOR_PERIODIC_MODE);
        return HAL_I2C_Master_Receive(&hi2c1, SHT30_ADDR_READ, dat, 6, 0xFFFF);}
```

从接收数据中校验并解析温度值和湿度值

在数据手册中可知，SHT30 分别在温度数据和湿度数据之后发送了 8-CRC 校验码，确保了数据可靠性。

CRC-8 校验程序如下：

```
#define CRC8_POLYNOMIAL 0x31

uint8_t CheckCrc8(uint8_t* const message, uint8_t initial_value){
    uint8_t  remainder;            //余数
    uint8_t  i = 0, j = 0;  //循环变量

    /* 初始化 */
    remainder = initial_value;

    for(j = 0; j < 2;j++)
    {
        remainder ^= message[j];

        /* 从最高位开始依次计算  */
```

```
        for (i = 0; i < 8; i++)
        {
            if (remainder & 0x80)
            {
                remainder = (remainder << 1)^CRC8_POLYNOMIAL;
            }
            else
            {
                remainder = (remainder << 1);
            }
        }
    }

    /* 返回计算的 CRC 码 */
    return remainder;}
```

计算温度值和湿度值的公式在数据手册中已给出，如图：

Relative humidity conversion formula (result in %RH):

$$RH = 100 \cdot \frac{S_{RH}}{2^{16} - 1}$$

Temperature conversion formula (result in °C & °F):

$$T[°C] = -45 + 175 \cdot \frac{S_T}{2^{16} - 1}$$

$$T[°F] = -49 + 315 \cdot \frac{S_T}{2^{16} - 1}$$

接下来编写解析数据的函数：

```
/**
 * @brief 将SHT30接收的6个字节数据进行CRC校验，并转换为温度值和湿度值
 * @param dat  —— 存储接收数据的地址（6个字节数组）
 * @retval校验成功  —— 返回0
 *                  校验失败  —— 返回1，并设置温度值和湿度值为0
*/
uint8_t SHT30_Dat_To_Float(uint8_t* const dat, float* temperature, float* humidity){
        uint16_t recv_temperature = 0;
        uint16_t recv_humidity = 0;

        /* 校验温度数据和湿度数据是否接收正确 */
        if(CheckCrc8(dat, 0xFF) != dat[2] || CheckCrc8(&dat[3], 0xFF) != dat[5])
                return 1;

        /* 转换温度数据 */
        recv_temperature = ((uint16_t)dat[0]<<8)|dat[1];
        *temperature = -45 + 175*((float)recv_temperature/65535);

        /* 转换湿度数据 */
        recv_humidity = ((uint16_t)dat[3]<<8)|dat[4];
        *humidity = 100 * ((float)recv_humidity / 65535);

        return 0;}
```

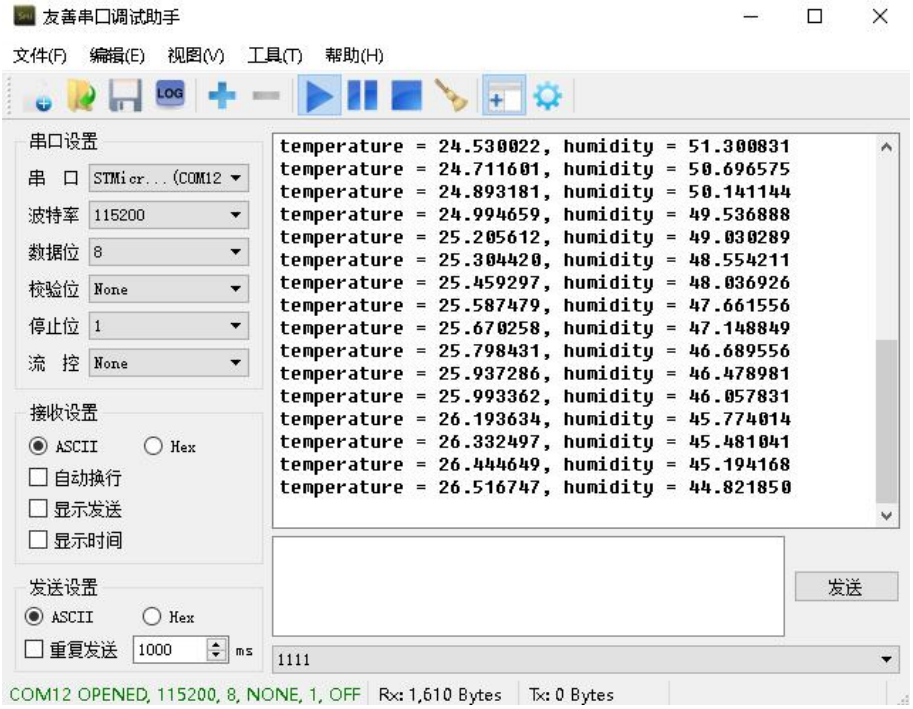
5. 测试SHT30驱动程序

在main函数中对该驱动进行测试，在`main.c`中添加如下代码：

```c
#include <stdio.h>#include "sht30_i2c_drv.h"
int main(void){
    /* USER CODE BEGIN 1 */
    uint8_t recv_dat[6] = {0};
    float temperature = 0.0;
    float humidity = 0.0;
    /* USER CODE END 1 */

    HAL_Init();

    SystemClock_Config();

    MX_GPIO_Init();
    MX_I2C1_Init();
    MX_USART1_UART_Init();

    /* USER CODE BEGIN 2 */
    SHT30_Reset();
    if(SHT30_Init() == HAL_OK)
        printf("sht30 init ok.\n");
    else
        printf("sht30 init fail.\n");
  /* USER CODE END 2 */

  /* Infinite loop */
```

```
  /* USER CODE BEGIN WHILE */
  while (1)
  {
    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
                    HAL_Delay(1000);
                    if(SHT30_Read_Dat(recv_dat) == HAL_OK)
                    {
                            if(SHT30_Dat_To_Float(recv_dat, &temperature, &humidity)==0)
                            {
                                    printf("temperature = %f, humidity = %f\n",
temperature, humidity);
                            }
                            else
                            {
                                    printf("crc check fail.\n");
                            }
                    }
                    else
                    {
                            printf("read data from sht30 fail.\n");
                    }
          }
  /* USER CODE END 3 */}
```

测试结果如图：



至此，我们已经学会**如何使用硬件 IIC 接口读取温湿度传感器数据并使用软件 CRC 校验（SHT30）**，下一节将讲述如何使用硬件 CRC 校验 SHT30 的数据。

作业：

编写程序，通过 I2C 接口读取 SHT30 温湿度传感器的数据，并处理显示。

分析温湿度传感器的测量精度和响应时间。

STM32 单片机基础 16——使用硬件 CRC 校验数据（以 SHT30 为例）

教学目的与要求：

目的：学习 CRC（循环冗余校验）算法的原理，并应用于数据通信中以提高数据传输的可靠性。

要求：能够编写程序实现 CRC 校验算法，并将其应用于 SHT30 传感器数据的校验过程中。

教学重难点：

重点：CRC 算法的原理和实现。

难点：理解 CRC 校验在数据传输中的作用，确保校验结果的正确性。

课时数：3 课时

思政元素：

培养学生的严谨性和责任心，通过 CRC 校验的学习，让学生理解在数据传输中，确保数据完整性和准确性的重要性，培养学生的细致入微和负责任的工作态度。

本篇详细的记录了如何使用 STM32CubeMX 配置 STM32L431RCT6 的硬件 CRC 外设校验数据，并用 SHT30 温湿度传感器为例检查是否可以正确校验。

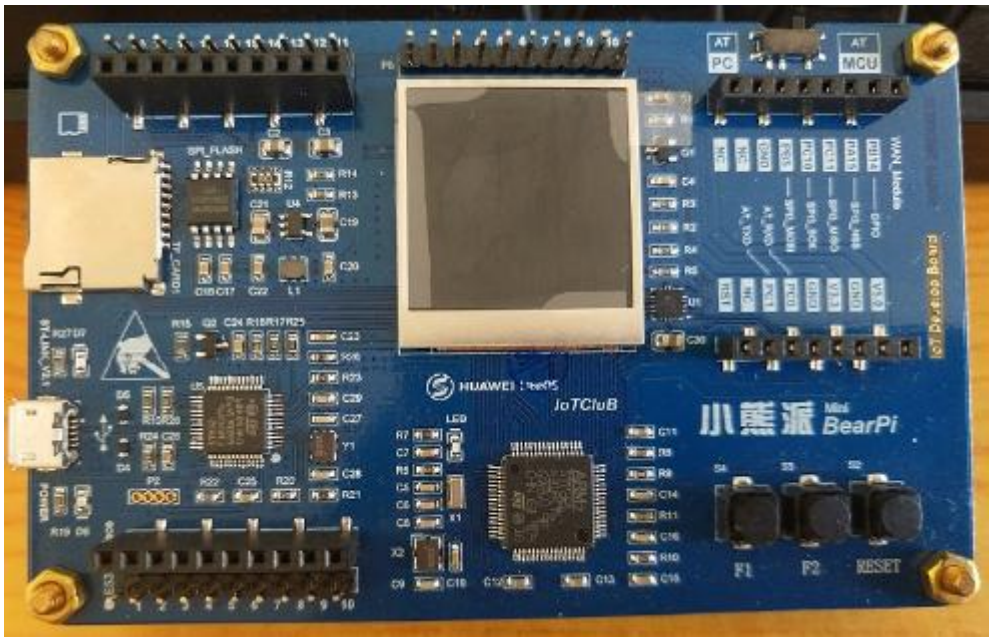
1. 准备工作

硬件准备

硬件准备

开发板

首先需要准备一个开发板，这里我准备的是 STM32L4 的开发板（BearPi）：



软件准备

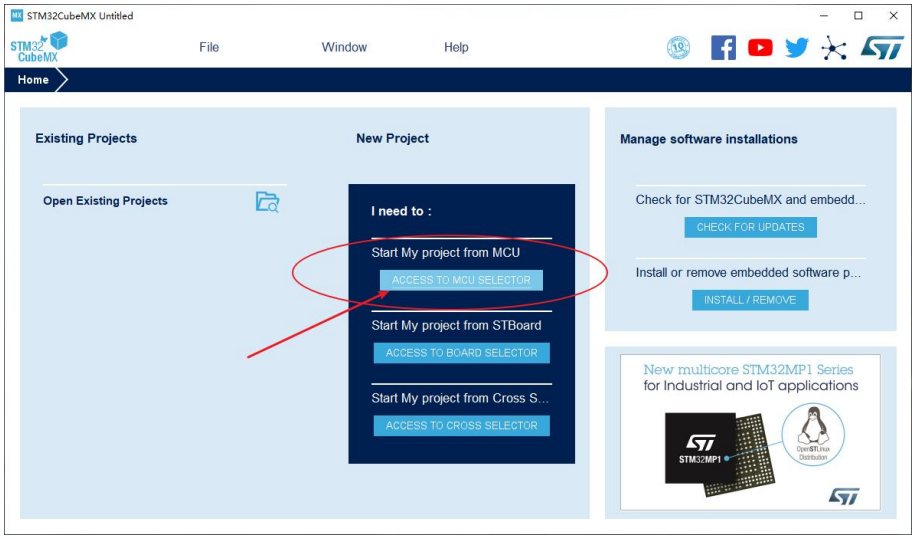
- 需要安装好 Keil - MDK 及芯片对应的包，以便编译和下载生成的代码；

Keil MDK 和串口助手的安装包都可以关注“**小熊派开源社区**”微信公众号，在资料教程一栏中可获取安装包。

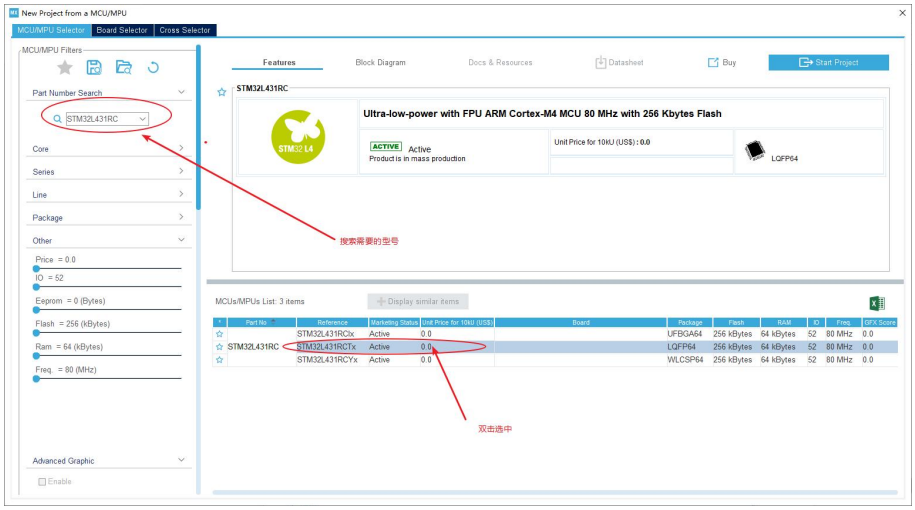
2. 生成 MDK 工程

选择芯片型号

打开 STM32CubeMX，打开 MCU 选择器：



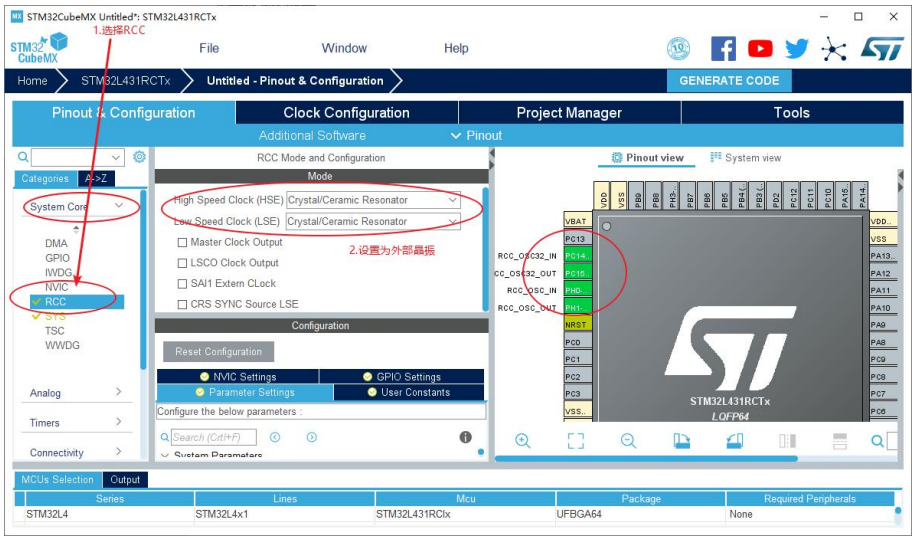
搜索并选中芯片 STM32L431RCT6:



配置时钟源

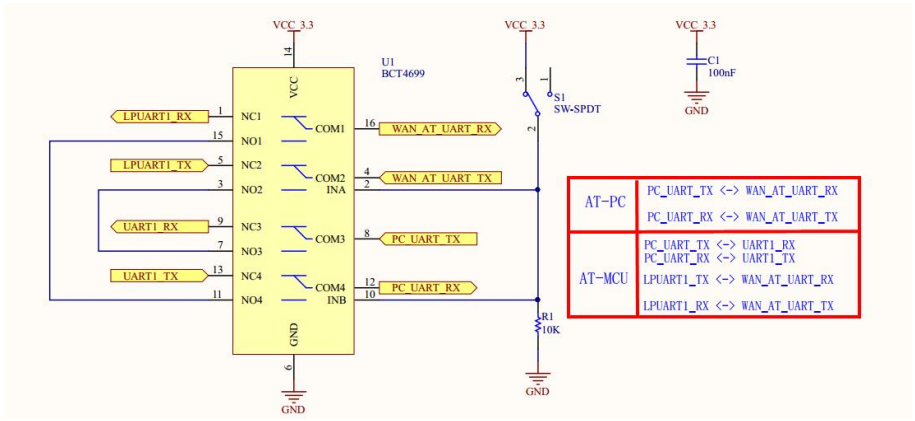
- 如果选择使用外部高速时钟（HSE），则需要在 System Core 中配置 RCC；
- 如果使用默认内部时钟（HSI），这一步可以略过；

这里我都使用外部时钟：



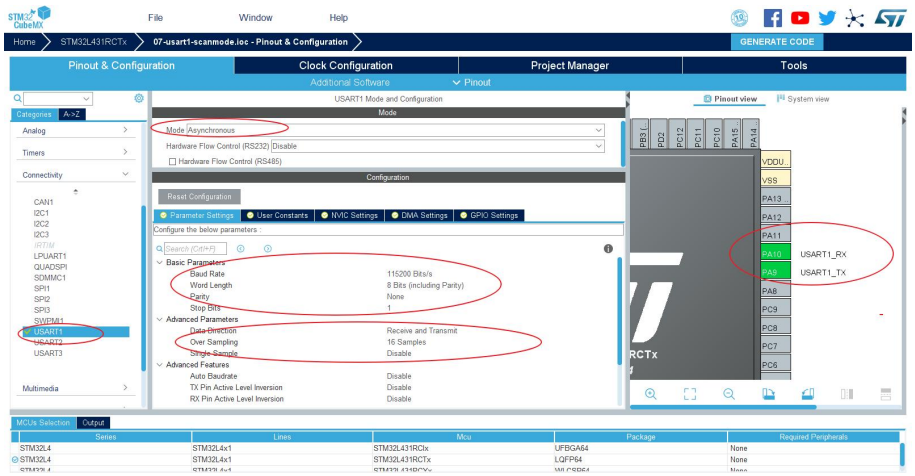
配置串口

小熊派开发板板载 ST-Link 并且虚拟了一个串口，原理图如下：



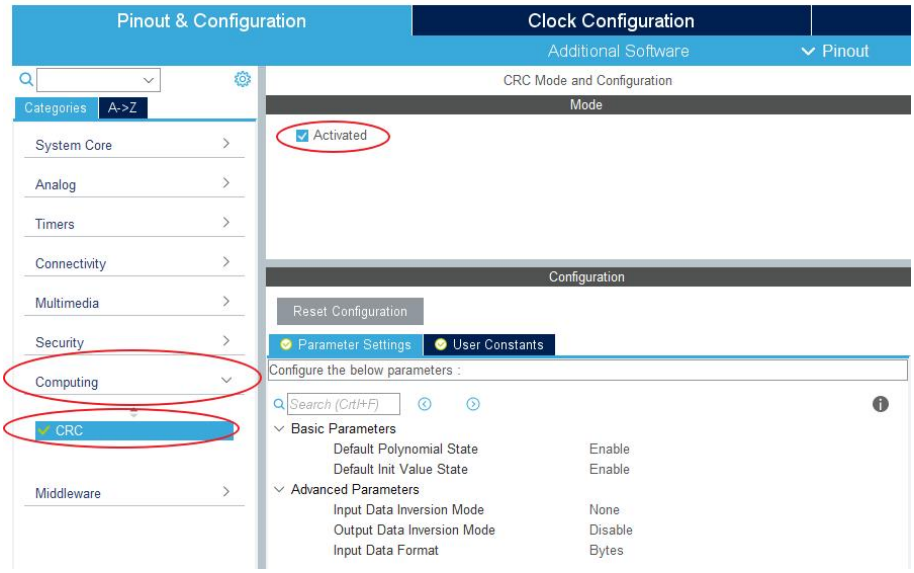
这里我将开关拨到 AT-MCU 模式，使 PC 的串口与 USART1 之间连接。

接下来开始配置 USART1：



配置 CRC 外设

首先激活 CRC：

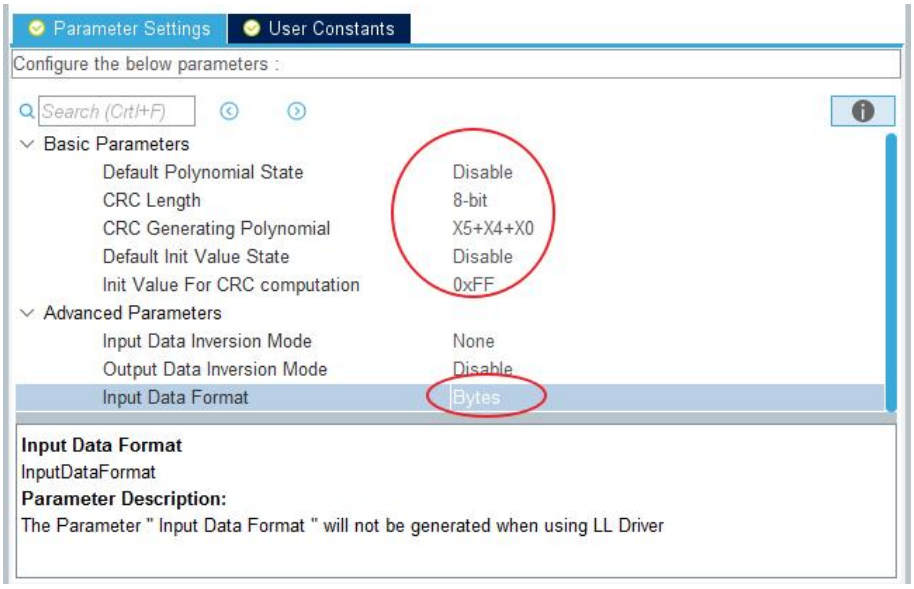


然后配置 CRC 校验的初始值：

这里我们以 SHT30 为例，其数据手册中已给出，如图：

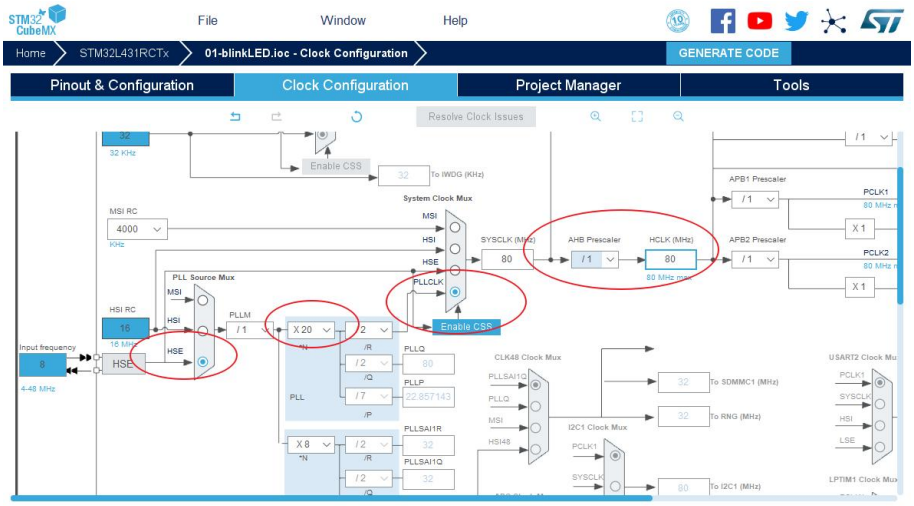
Property	Value
Name	CRC-8
Width	8 bit
Protected data	read and/or write data
Polynomial	0x31 ($x^8 + x^5 + x^4 + 1$) CRC生成多项式
Initialization	0xFF 初始值
Reflect input	False
Reflect output	False
Final XOR	0x00
Examples	CRC (0xBEEF) = 0x92 示例

据此，CRC 外设的配置如下：

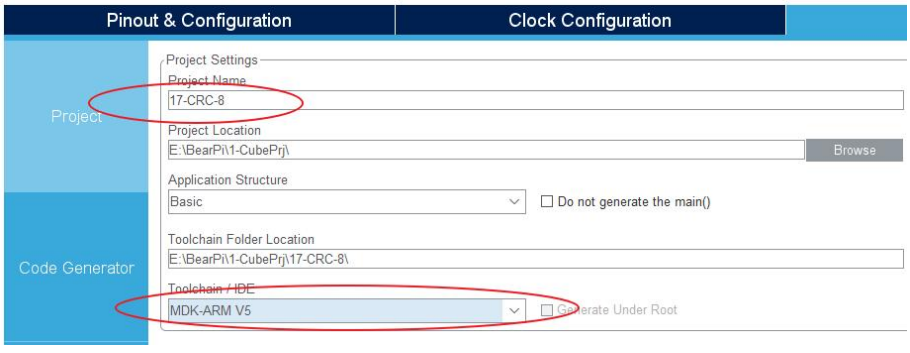


配置时钟树

STM32L4 的最高主频到 80M，所以配置 PLL，最后使 `HCLK = 80Mhz` 即可：

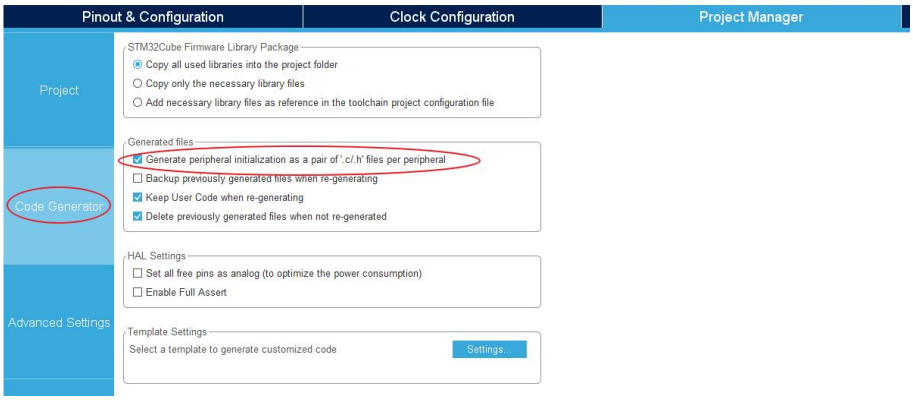


生成工程设置



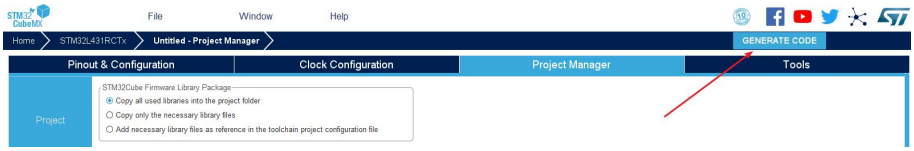
代码生成设置

最后设置生成独立的初始化文件：



生成代码

点击 `GENERATE CODE` 即可生成 MDK-V5 工程：



3. 在 MDK 中编写、编译、下载用户代码

重定向 printf( )函数

参考：重定向 printf 函数到串口输出的多种方法。

测试 CRC 校验

在 `main.c` 文件中添加如下代码：

```
/* USER CODE BEGIN Includes */#include <stdio.h>/* USER CODE END Includes */
```

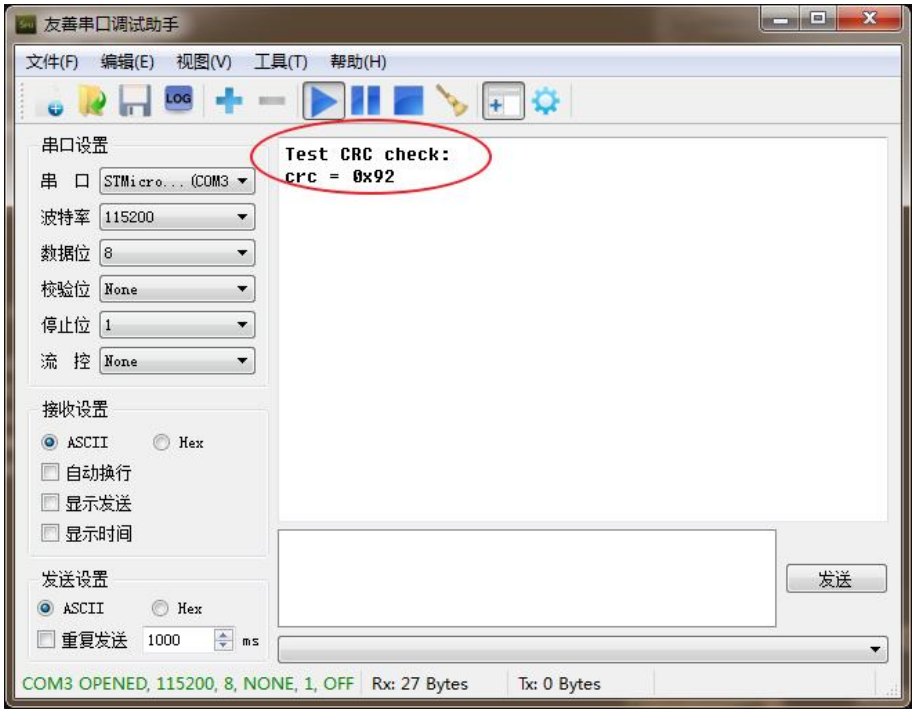
然后修改 `main` 函数：

```
int main(void){

  /* USER CODE BEGIN 1 */

        uint8_t dat[2] = {0xBE, 0xEF};

        uint8_t crc = 0;



  /* USER CODE END 1 */

  HAL_Init();
```

```
    SystemClock_Config();


    MX_GPIO_Init();
    MX_CRC_Init();
    MX_USART1_UART_Init();

    /* USER CODE BEGIN 2 */
          printf("Test CRC check:\n");
          crc = HAL_CRC_Accumulate(&hcrc, (uint32_t*)dat, 2);
          printf("crc = %#x\n", crc);
    /* USER CODE END 2 */


    while (1)
    {
    }}
```

测试结果

测试结果如下：



至此，我们已经学会**如何使用硬件 CRC 校验 SHT30 的数据**，下一节将讲述如何使用硬件 SPI 驱动 LCD 屏幕（ST7789）。

作业：

实现 CRC 校验算法，对 SHT30 传感器数据进行校验，确保数据完整性。

分析 CRC 校验在数据传输中的重要性。

STM32 单片机基础 17——使用硬件 SPI 驱动 TFT-LCD（ST7789）

教学目的与要求：

目的：掌握 SPI 通信协议，了解 TFT-LCD 显示屏的工作原理，实现图形和文字的显示。

要求：能够配置 STM32 的 SPI 接口，编写程序控制 ST7789 TFT-LCD 显示屏，显示静态和动态图形以及文本信息。

教学重难点：

重点：SPI 接口的配置和 TFT-LCD 显示屏的初始化与显示控制。

难点：理解 TFT-LCD 显示屏的像素寻址和颜色控制，实现高质量的图形和文本显示。

课时数：3 课时

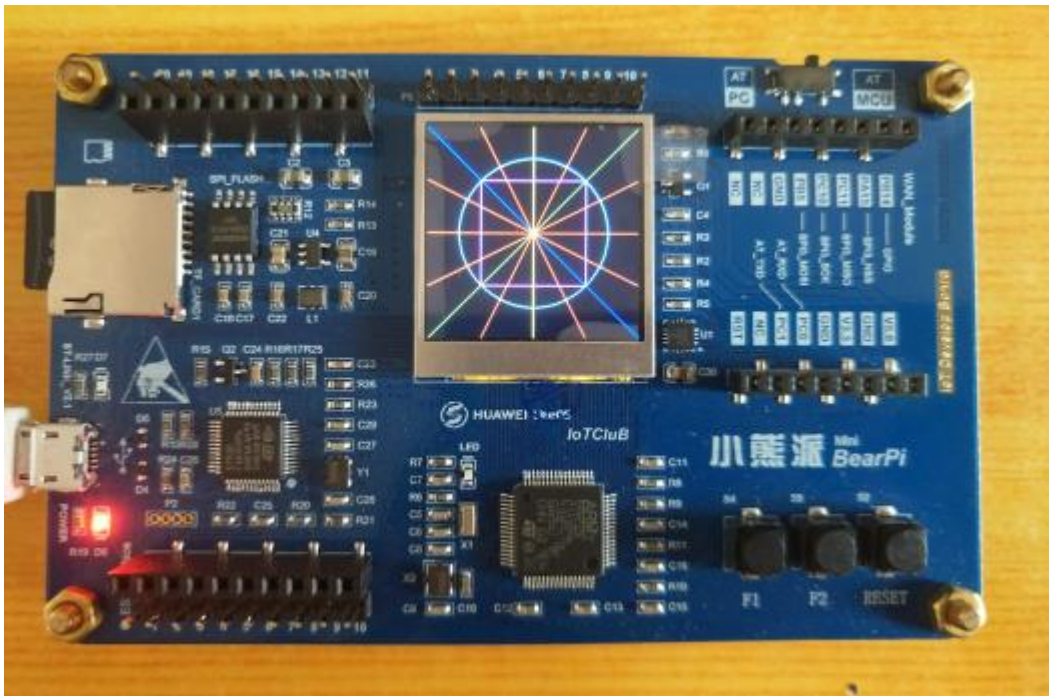
思政元素：

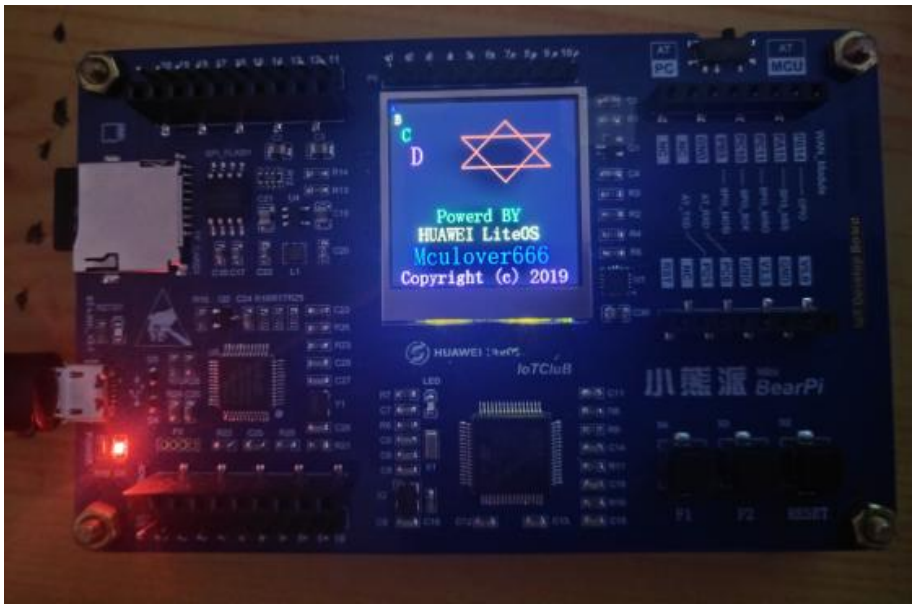
培养学生的创新思维和审美能力，通过 TFT-LCD 显示屏的应用，让学生认识到图形用户界面在人机交互中的重要性，鼓励学生探索不同的显示效果和界面设计，培养创新精神和审美素养。

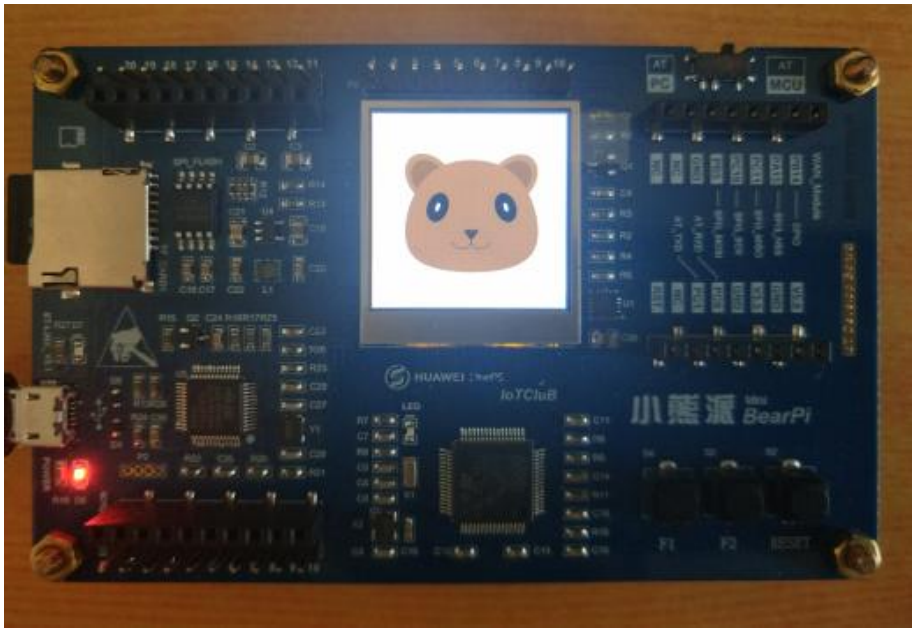
本篇详细的记录了如何使用 STM32CubeMX 配置 STM32L431RCT6 的硬件 SPI 外设与 ST7789 通信，驱动 16bit TFT-LCD 屏幕。

0. 前言

> 学习 SPI 外设驱动 LCD 屏幕没有必要手写驱动，学习这部分代码的目的是为了了解 TFT-LCD 的工作原理，每个像素点是如何显示的，**不要花过多的精力在弄明白每个命令的意思**，建议基于本驱动，学习一下打点，画线算法，画圆算法，画多边形算法等等，还可以学习显示英文字符，中文字符，最后还可以移植 STemwin 显示界面等等好玩的东西~





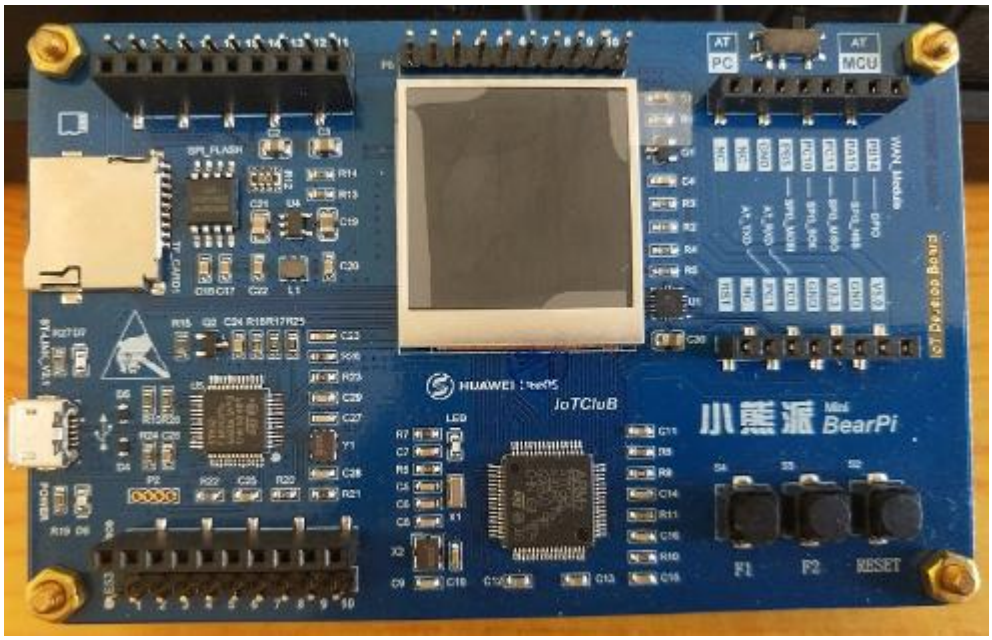


1. 准备工作

硬件准备

开发板

首先需要准备一个开发板，这里我准备的是 STM32L4 的开发板（BearPi）：



- LCD 屏幕

  小熊派开发板板载 LCD 屏幕**大小 1.3 寸，分辨率 240*240，色彩深度 16bit，使用 ST7789V2 液晶控制器**。

软件准备

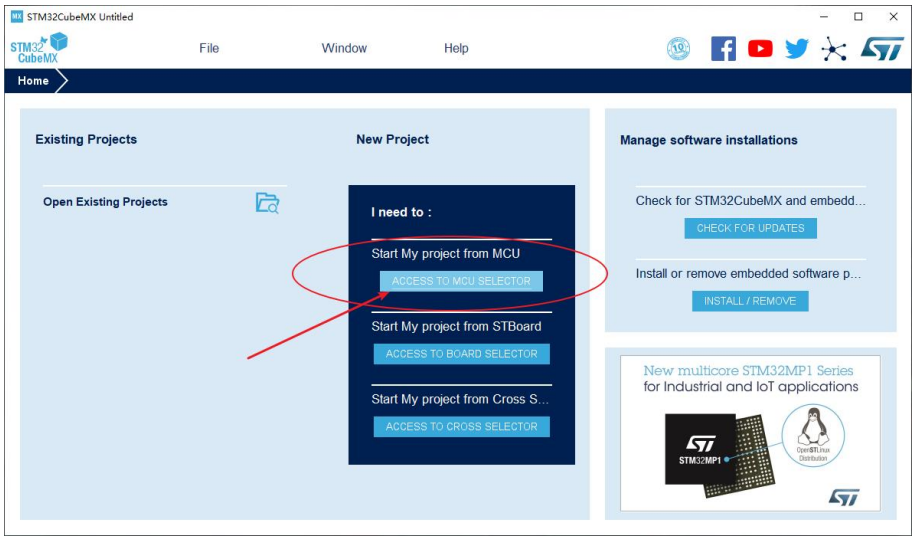
- 需要安装好 Keil - MDK 及芯片对应的包，以便编译和下载生成的代码；

  Keil MDK 和串口助手的安装包都可以关注“**小熊派开源社区**”微信公众号，在资料教程一栏中可获取安装包。

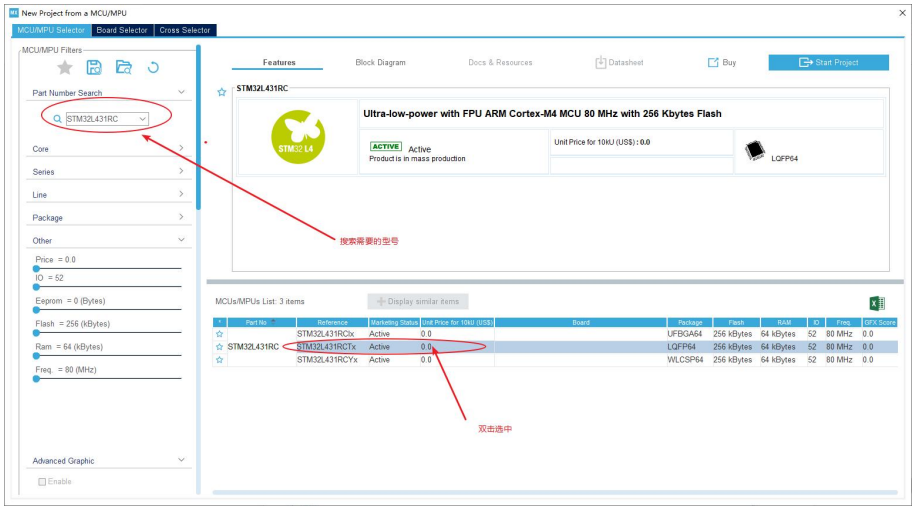
2.生成 MDK 工程

选择芯片型号

打开 STM32CubeMX，打开 MCU 选择器：



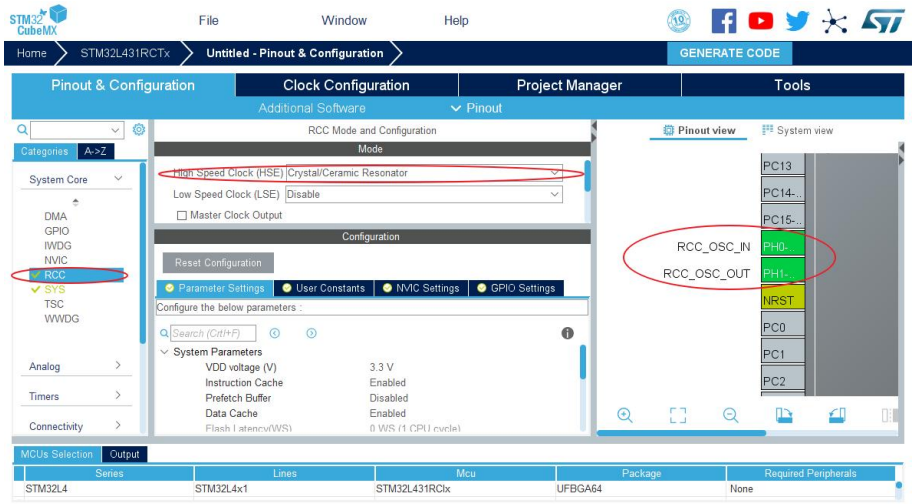
搜索并选中芯片 STM32L431RCT6:



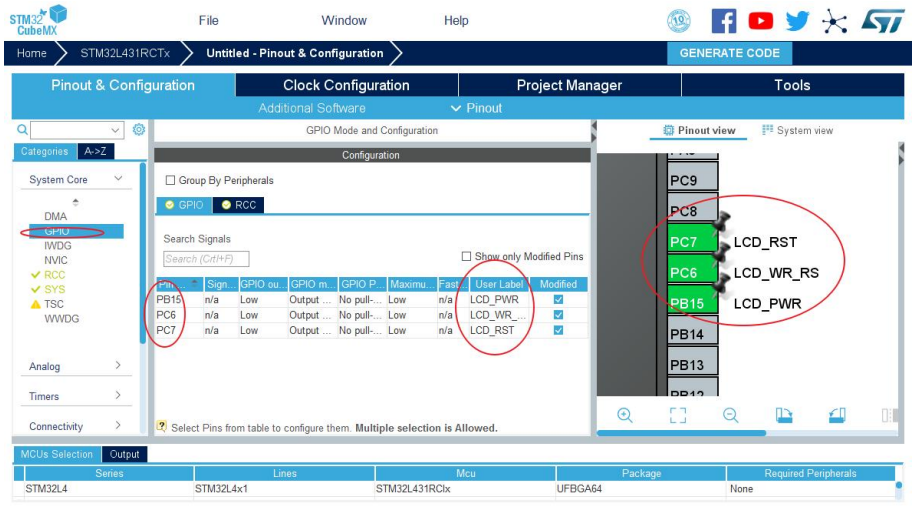
配置时钟源

- 如果选择使用外部高速时钟（HSE），则需要在 System Core 中配置 RCC；
- 如果使用默认内部时钟（HSI），这一步可以略过；

这里我都使用外部时钟：

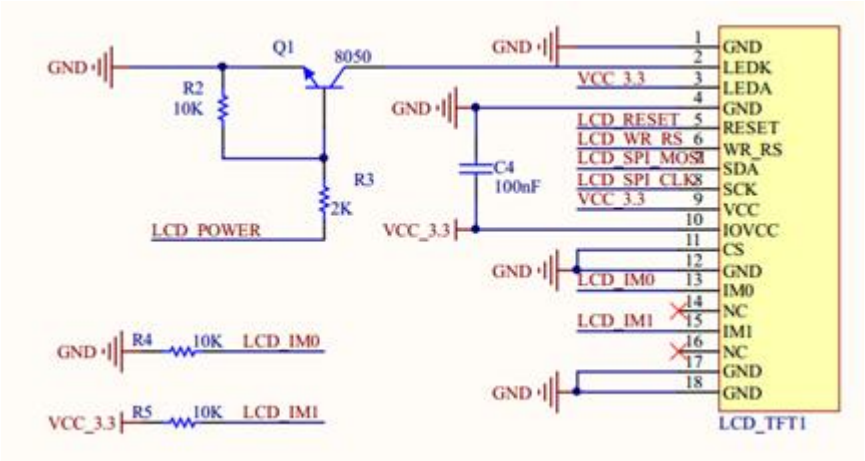


配置 LCD 控制 GPIO



配置 SPI2 接口

查看小熊派 LCD 接口的原理图：



引脚对应表如下：

LCD 引脚	MCU 引脚

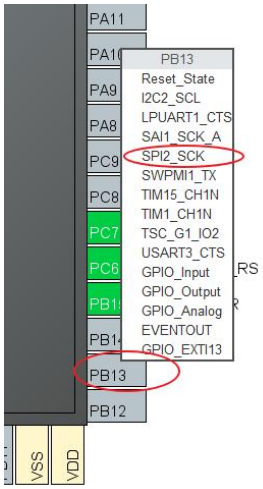
LCD 引脚	MCU 引脚
SPI2_MOSI	PC3
SPI2_CLK	PB13
LCD_WR_RS	PC6
LCD_RESET	PC7
LCD_POWER	PB15

MCU 只需要通过 SPI 向 LCD 控制器发送命令/数据即可，所以硬件上接 SPI2 的 SCK 和 MOSI 引脚，软件上将 SPI2 配置为发送主机模式，接下来开始配置 SPI2 接口：

参数设置如下：

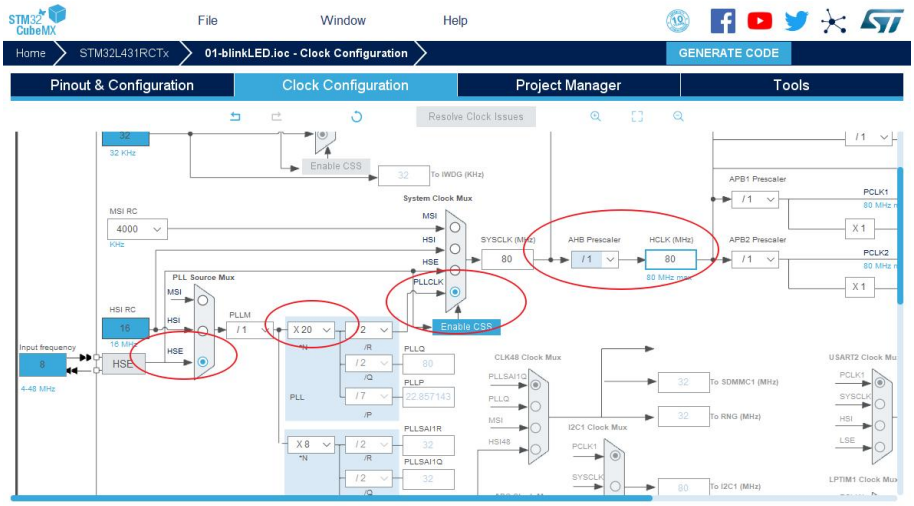
[外链图片转存失败,源站可能有防盗链机制,建议将图片保存下来直接上传(img-DUX2uXV4-1581661413301)(http://mculover666.cn/image/20190829/1DHh2ytSwgcp.png?imageslim)]

SPI2 默认 SCK 引脚是 PB10，和开发板不对应，所以重新修改引脚为 PB13：

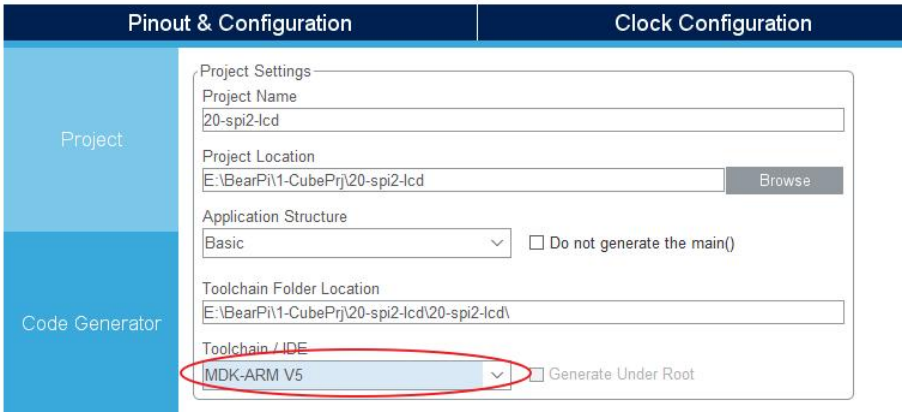


配置时钟树

STM32L4 的最高主频到 80M，所以配置 PLL，最后使 HCLK = 80Mhz 即可：

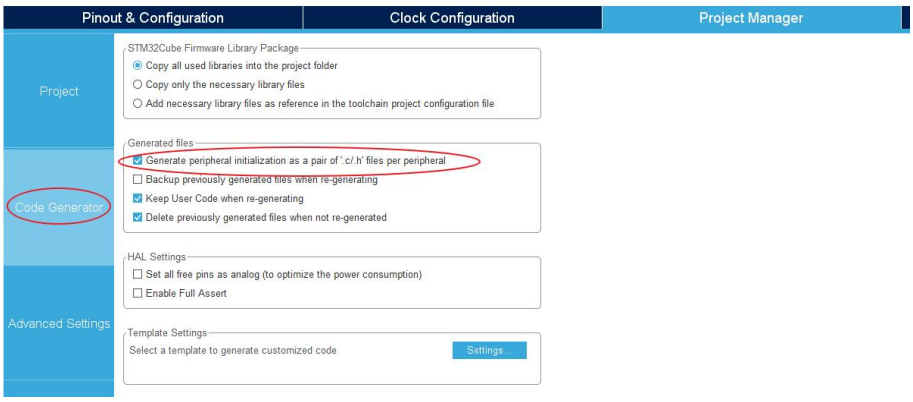


生成工程设置



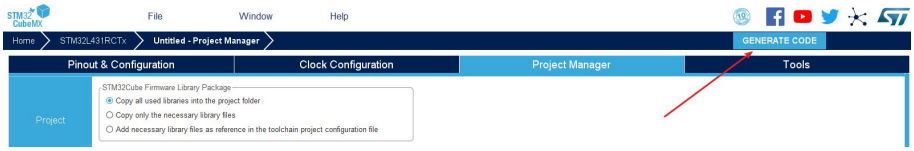
代码生成设置

最后设置生成独立的初始化文件：



生成代码

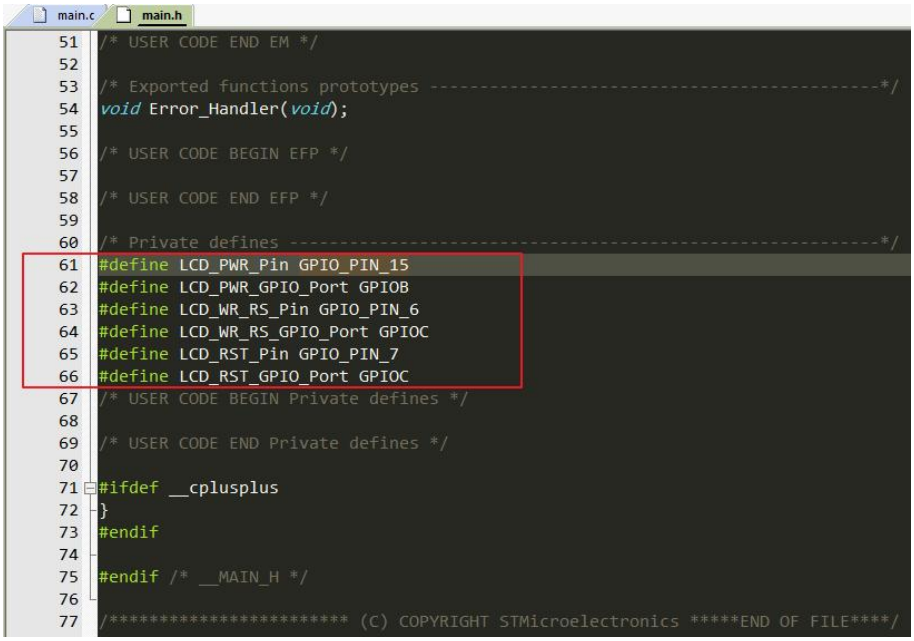
点击 GENERATE CODE 即可生成 MDK-V5 工程：



3. 编写 LCD 驱动（ST7789）—— 封装宏和底层函数

3.1. 封装控制 LCD 控制引脚高低电平的宏

控制引脚宏定义已经包含在 main.h 中，如图：



在编写驱动的过程中需要不断的控制这些控制引脚的电平，所以首先在 lcd_spi2_drv.h 头文件中编写控制这些引脚的宏：

```
#include "main.h"

#define    LCD_PWR(n)            (n?\

           HAL_GPIO_WritePin(LCD_PWR_GPIO_Port,LCD_PWR_Pin,GPIO_PIN_SET):\

           HAL_GPIO_WritePin(LCD_PWR_GPIO_Port,LCD_PWR_Pin,GPIO_PIN_RESET))#define
           LCD_WR_RS(n)          (n?\

           HAL_GPIO_WritePin(LCD_WR_RS_GPIO_Port,LCD_WR_RS_Pin,GPIO_PIN_SET):\
```

```
        HAL_GPIO_WritePin(LCD_WR_RS_GPIO_Port,LCD_WR_RS_Pin,GPIO_PIN_RESET))#define
        LCD_RST(n)          (n?\

        HAL_GPIO_WritePin(LCD_RST_GPIO_Port,LCD_RST_Pin,GPIO_PIN_SET):\

        HAL_GPIO_WritePin(LCD_RST_GPIO_Port,LCD_RST_Pin,GPIO_PIN_RESET))
```

3.2. 宏定义屏幕分辨率和颜色值

```
//LCD屏幕分辨率定义#define LCD_Width   240#define LCD_Height  240//颜色定义#define WHITE
0xFFFF    //白色#define YELLOW  0xFFE0   //黄色#define BRRED   0XFC07  //棕红色#define PINK
0XF81F    //粉色#define RED     0xF800   //红色#define BROWN   0XBC40  //棕色#define GRAY
0X8430  //灰色#define GBLUE   0X07FF    //兰色#define GREEN   0x07E0   //绿色#define BLUE
0x001F  //蓝色#define BLACK   0x0000    //黑色
```

接下来开始在 `lcd_spi2_drv.c` 编写驱动程序~

3.3. 封装 LCD 控制引脚初始化函数

首先包含必要的头文件：

```
#include "lcd_spi2_drv.h"#include "gpio.h"#include "spi.h"
```

这个函数只能在本文件内由 LCD 初始化函数调用，所以使用 static 修饰为静态的：

```
/**
 *@brief    LCD控制引脚和通信接口初始化
 *@param    none
 *@retval   none
*/static void LCD_GPIO_Init(void){
    /* 初始化引脚 */
        MX_GPIO_Init();

        /* 复位LCD */
    LCD_PWR(0);
    LCD_RST(0);
```

```
    HAL_Delay(100);
    LCD_RST(1);


          /* 初始化 SPI2 接口 */
    MX_SPI2_Init();}
```

3.4. 封装 LCD 发送数据和发送命令函数

数据都是由 SPI2 的 MOSI 发送，由 LCD_WR_RS 引脚指明该数据是命令还是数据。

首先在 `spi.c` 的最后调用 HAL 库封装一个函数，供驱动程序调用：

```
/* USER CODE BEGIN 1 *//**
 * @brief    SPI 发送字节函数
 * @param    TxData 要发送的数据
 * @param    size   发送数据的字节大小
 * @return   0:写入成功,其他:写入失败
 */
uint8_t SPI_WriteByte(uint8_t *TxData,uint16_t size){
          return HAL_SPI_Transmit(&hspi2,TxData,size,1000);}/* USER CODE END 1 */
```

不要忘了在 spi.h 中声明该函数！

然后基于 spi 发送字节函数，在**驱动文件中**继续封装一个向 LCD 发送数据的函数，一个向 LCD 发送命令的函数：

```
/**
 * @brief    写命令到 LCD
 * @param    cmd —— 需要发送的命令
 * @return   none
 */static void LCD_Write_Cmd(uint8_t cmd){
    LCD_WR_RS(0);
    SPI_WriteByte(&cmd, 1);}
```

```
/**
 * @brief   写数据到LCD
 * @param   dat —— 需要发送的数据
 * @return  none
 */static void LCD_Write_Data(uint8_t dat){
    LCD_WR_RS(1);
    SPI_WriteByte(&dat, 1);}
```

4. 编写 LCD 驱动（ST7789）—— 对照 datasheet 编程

4.1. 打开/关闭背光函数

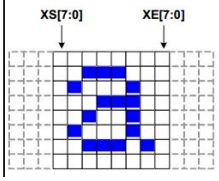
这两个函数比较简单，直接调用控制 LCD 背光的引脚控制宏即可：

```
/**
 * @breif   打开LCD显示背光
 * @param   none
 * @return  none
 */void LCD_DisplayOn(void){
    LCD_PWR(1);}/**
 * @brief   关闭LCD显示背光
 * @param   none
 * @return  none
 */void LCD_DisplayOff(void){
    LCD_PWR(0);}
```

4.2. 指定显示 RAM 操作地址

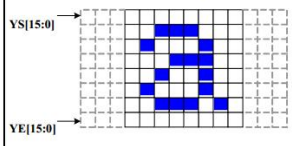
根据数据手册，当要改变某个区域像素点的颜色时，首先应该确定 X 方向起始地址和 X 方向结束地址：

9.1.20 CASET (2Ah): Column Address Set

2AH	CASET (Column Address Set)												
Inst / Para	D/CX	WRX	RDX	D17-8	D7	D6	D5	D4	D3	D2	D1	D0	HEX
CASET	0	↑	1	-	0	0	1	0	1	0	1	0	(2Ah)
1st parameter	1	↑	1	-	XS15	XS14	XS13	XS12	XS11	XS10	XS9	XS8	
2nd parameter	1	↑	1	-	XS7	XS6	XS5	XS4	XS3	XS2	XS1	XS0	
3rd parameter	1	↑	1	-	XE15	XE14	XE13	XE12	XE11	XE10	XE9	XE8	
4th parameter	1	↑	1	-	XE7	XE6	XE5	XE4	XE3	XE2	XE1	XE0	
2. Description	-The value of XS [7:0] and XE [7:0] are referred when RAMWR command comes. -Each value represents one column line in the Frame Memory. 												

然后确定 Y 方向起始地址和 Y 方向结束地址：

9.1.21 RASET (2Bh): Row Address Set

2BH	RASET (Row Address Set)												
Inst / Para	D/CX	WRX	RDX	D17-8	D7	D6	D5	D4	D3	D2	D1	D0	HEX
RASET	0	↑	1	-	0	0	1	0	1	0	1	1	(2Bh)
1st parameter	1	↑	1	-	YS15	YS14	YS13	YS12	YS11	YS10	YS9	YS8	
2nd parameter	1	↑	1	-	YS7	YS6	YS5	YS4	YS3	YS2	YS1	YS0	
3rd parameter	1	↑	1	-	YE15	YE14	YE13	YE12	YE11	YE10	YE9	YE8	
4th parameter	1	↑	1	-	YE7	YE6	YE5	YE4	YE3	YE2	YE1	YE0	
3. Description	-This command is used to defined area of frame memory where MCU can access. -The value of YS [15:0] and YE [15:0] are referred when RAMWR command comes. -Each value represents one page line in the Frame Memory. 												

最后再确定该区域内每个像素点的值(16bit)：

9.1.22 RAMWR (2Ch): Memory Write

2CH	RAMWR (Memory Write)												
Inst / Para	D/CX	WRX	RDX	D17-8	D7	D6	D5	D4	D3	D2	D1	D0	HEX
RAMWR	0	↑	1	-	0	0	1	0	1	1	0	0	(2Ch)
1st parameter	1	↑	1	D1[17]-1[8]	D1[7]	D1[6]	D1[5]	D1[4]	D1[3]	D1[2]	D1[1]	D1[0]	
…	1	↑	1	Dx[17]-x[8]	Dx[7]	Dx[6]	Dx[5]	Dx[4]	Dx[3]	Dx[2]	Dx[1]	Dx[0]	
N parameter	1	↑	1	Dn[17]-n[8]	Dn[7]	Dn[6]	Dn[5]	Dn[4]	Dn[3]	Dn[2]	Dn[1]	Dn[0]	
Description	-This command is used to transfer data from MCU to frame memory. -When this command is accepted, the column register and the page register are reset to the start column/start page positions. -The start column/start page positions are different in accordance with MADCTL setting. -Sending any other command can stop frame write.												

综上，我们每次操作的时候都需要指定操作区域，所以编写该函数：

```
/**
 * @brief   设置数据写入 LCD 显存区域
```

```
 * @param   x1,y1   —— 起点坐标
 * @param   x2,y2   —— 终点坐标
 * @return  none
 */void LCD_Address_Set(uint16_t x1, uint16_t y1, uint16_t x2, uint16_t y2){
    /* 指定X方向操作区域 */
    LCD_Write_Cmd(0x2a);
    LCD_Write_Data(x1 >> 8);
    LCD_Write_Data(x1);
    LCD_Write_Data(x2 >> 8);
    LCD_Write_Data(x2);

    /* 指定Y方向操作区域 */
    LCD_Write_Cmd(0x2b);
    LCD_Write_Data(y1 >> 8);
    LCD_Write_Data(y1);
    LCD_Write_Data(y2 >> 8);
    LCD_Write_Data(y2);

    /* 发送该命令，LCD开始等待接收显存数据 */
    LCD_Write_Cmd(0x2C);}
```

4.3. 清屏函数

编写完指定显存操作区域后，趁热打铁，编写清屏函数就很简单啦，直接调用上面编写的函数，指定操作地址为全屏幕，然后循环发送颜色值即可：

```
#define LCD_TOTAL_BUF_SIZE     (240*240*2)#define LCD_Buf_Size 1152static uint8_t
lcd_buf[LCD_Buf_Size];/**
 * @brief   以一种颜色清空LCD屏
```

```
* @param   color —— 清屏颜色(16bit)
* @return  none
*/void LCD_Clear(uint16_t color){
    uint16_t i, j;
    uint8_t data[2] = {0};  //color是16bit的，每个像素点需要两个字节的显存

    /* 将16bit的color值分开为两个单独的字节 */
    data[0] = color >> 8;
    data[1] = color;

    /* 显存的值需要逐字节写入 */
    for(j = 0; j < LCD_Buf_Size / 2; j++)
    {
        lcd_buf[j * 2] =  data[0];
        lcd_buf[j * 2 + 1] =  data[1];
    }
    /* 指定显存操作地址为全屏幕 */
    LCD_Address_Set(0, 0, LCD_Width - 1, LCD_Height - 1);
    /* 指定接下来的数据为数据 */
    LCD_WR_RS(1);
    /* 将显存缓冲区的数据全部写入缓冲区 */
    for(i = 0; i < (LCD_TOTAL_BUF_SIZE / LCD_Buf_Size); i++)
    {
        SPI_WriteByte(lcd_buf, (uint16_t)LCD_Buf_Size);
    }}
```

4.4. LCD 初始化函数

至此，LCD 的一些操作函数全部编写完成，最后编写初始化 LCD 模式的函数：

```
/**
 * @brief   LCD 初始化
 * @param   none
 * @return  none
 */void LCD_Init(void){
    /* 初始化和 LCD 通信的引脚 */
    LCD_GPIO_Init();
    HAL_Delay(120);

    /* 关闭睡眠模式 */
    LCD_Write_Cmd(0x11);
    HAL_Delay(120);

    /* 开始设置显存扫描模式，数据格式等 */
    LCD_Write_Cmd(0x36);
    LCD_Write_Data(0x00);
    /* RGB 5-6-5-bit 格式  */
    LCD_Write_Cmd(0x3A);
    LCD_Write_Data(0x65);
    /* porch 设置 */
    LCD_Write_Cmd(0xB2);
    LCD_Write_Data(0x0C);
    LCD_Write_Data(0x0C);
```

```
    LCD_Write_Data(0x00);
    LCD_Write_Data(0x33);
    LCD_Write_Data(0x33);
    /* VGH 设置 */
    LCD_Write_Cmd(0xB7);
    LCD_Write_Data(0x72);
    /* VCOM 设置 */
    LCD_Write_Cmd(0xBB);
    LCD_Write_Data(0x3D);
    /* LCM 设置 */
    LCD_Write_Cmd(0xC0);
    LCD_Write_Data(0x2C);
    /* VDV and VRH 设置 */
    LCD_Write_Cmd(0xC2);
    LCD_Write_Data(0x01);
    /* VRH 设置 */
    LCD_Write_Cmd(0xC3);
    LCD_Write_Data(0x19);
    /* VDV 设置 */
    LCD_Write_Cmd(0xC4);
    LCD_Write_Data(0x20);
    /* 普通模式下显存速率设置 60Mhz */
    LCD_Write_Cmd(0xC6);
    LCD_Write_Data(0x0F);
    /* 电源控制 */
```

```
    LCD_Write_Cmd(0xD0);
    LCD_Write_Data(0xA4);
    LCD_Write_Data(0xA1);
    /* 电压设置 */
    LCD_Write_Cmd(0xE0);
    LCD_Write_Data(0xD0);
    LCD_Write_Data(0x04);
    LCD_Write_Data(0x0D);
    LCD_Write_Data(0x11);
    LCD_Write_Data(0x13);
    LCD_Write_Data(0x2B);
    LCD_Write_Data(0x3F);
    LCD_Write_Data(0x54);
    LCD_Write_Data(0x4C);
    LCD_Write_Data(0x18);
    LCD_Write_Data(0x0D);
    LCD_Write_Data(0x0B);
    LCD_Write_Data(0x1F);
    LCD_Write_Data(0x23);
    /* 电压设置 */
    LCD_Write_Cmd(0xE1);
    LCD_Write_Data(0xD0);
    LCD_Write_Data(0x04);
    LCD_Write_Data(0x0C);
    LCD_Write_Data(0x11);
```

```
    LCD_Write_Data(0x13);
    LCD_Write_Data(0x2C);
    LCD_Write_Data(0x3F);
    LCD_Write_Data(0x44);
    LCD_Write_Data(0x51);
    LCD_Write_Data(0x2F);
    LCD_Write_Data(0x1F);
    LCD_Write_Data(0x1F);
    LCD_Write_Data(0x20);
    LCD_Write_Data(0x23);
    /* 显示开 */
    LCD_Write_Cmd(0x21);
    LCD_Write_Cmd(0x29);

    /* 清屏为白色 */
    LCD_Clear(WHITE);


    /*打开显示*/
    LCD_PWR(1);}
```

至此，驱动编写完成。

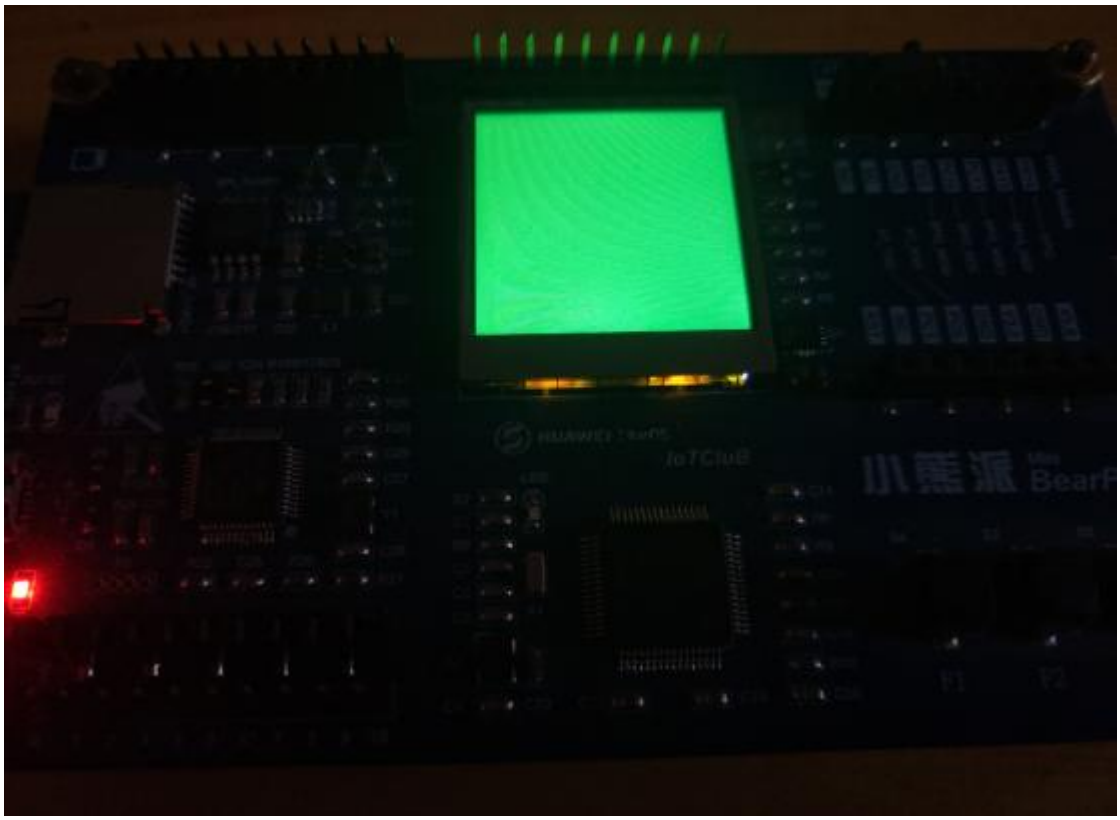
5. 测试驱动程序

在 `main函数` 中编写驱动测试代码，在 `while(1)` 之前添加如下代码：

```
  /* USER CODE BEGIN 2 */
  LCD_Init();
  LCD_Clear(GREEN);
```

```
/* USER CODE END 2 */
```

测试结果如图：



绿绿的，是不是很好看哈哈(斜眼笑.jpg)~

至此，我们已经学会**如何使用硬件 SPI 驱动 LCD 屏幕（ST7789）**，下一节将讲述如何使用硬件 QSPI 接口读写 SPI Flash 的数据。

作业：
编写程序，通过 SPI 接口控制 ST7789 TFT-LCD 显示屏显示图像和文字。
分析 SPI 通信协议在 TFT-LCD 驱动中的应用。

STM32 单片机基础 18——使用硬件 QSPI 读写 SPI Flash（W25Q64）

教学目的与要求：

目的：学习 QSPI 通信协议，了解 SPI Flash 的工作原理，实现高速的外部存储访问。

要求：能够配置 STM32 的 QSPI 接口，编写程序对 W25Q64 SPI Flash 进行读写操作，理解 QSPI 的传输效率和优势。

教学重难点：

重点：QSPI 接口的配置和 SPI Flash 的读写操作。

难点：理解 QSPI 与 SPI 通信协议的区别，确保 QSPI 传输的稳定性和效率。

课时数：3 课时

思政元素：

培养学生的效率意识和时间管理能力，通过 QSPI 高速读写的学习，让学生认识到在嵌入式系统设计中，优化数据传输效率的重要性，培养学生的效率观念和快速响应的能力。

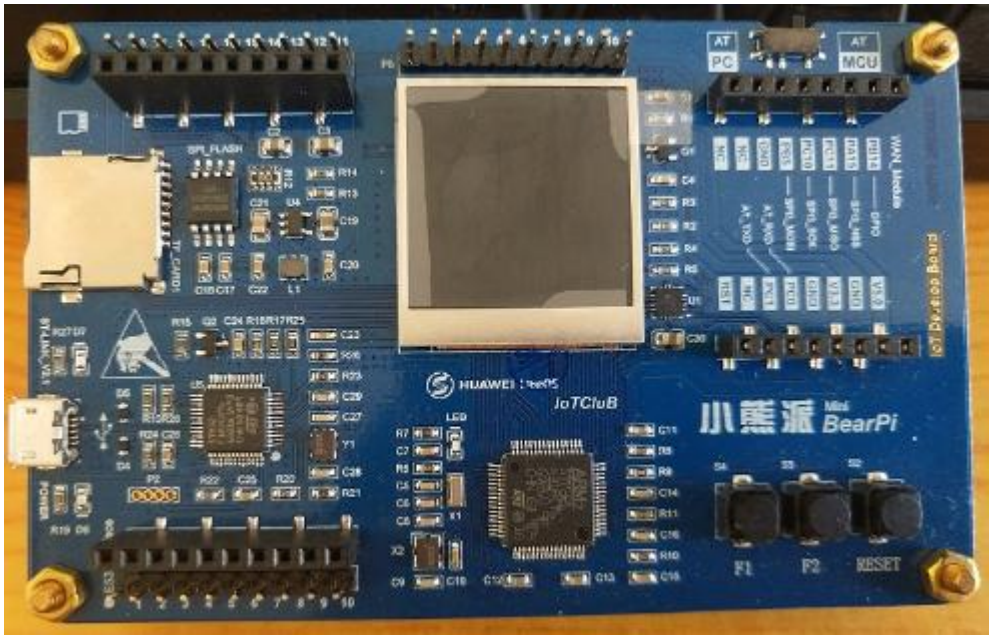
本篇详细的记录了如何使用 STM32CubeMX 配置 STM32L431RCT6 的硬件 QSPI 外设与 SPI Flash 通信（W25Q64）。

1. 准备工作

硬件准备

开发板

首先需要准备一个开发板，这里我准备的是 STM32L4 的开发板（BearPi）：



- SPI Flash

小熊派开发板板载一片 SPI Flash，型号为 W25Q64，大小为 8 MB，最大支持 80 Mhz 的操作频率。

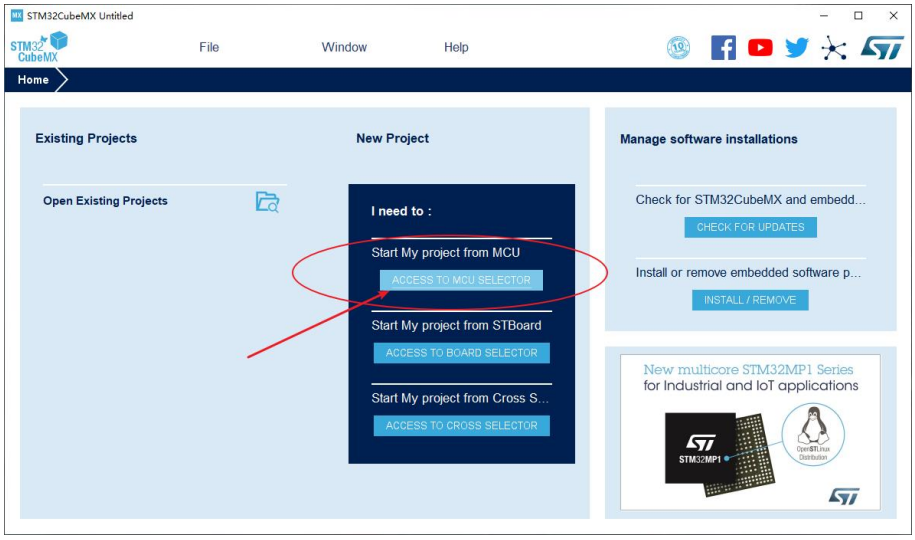
软件准备

- 需要安装好 Keil - MDK 及芯片对应的包，以便编译和下载生成的代码；
  Keil MDK 和串口助手的安装包都可以关注“**小熊派开源社区**”微信公众号，在资料教程一栏中可获取安装包。

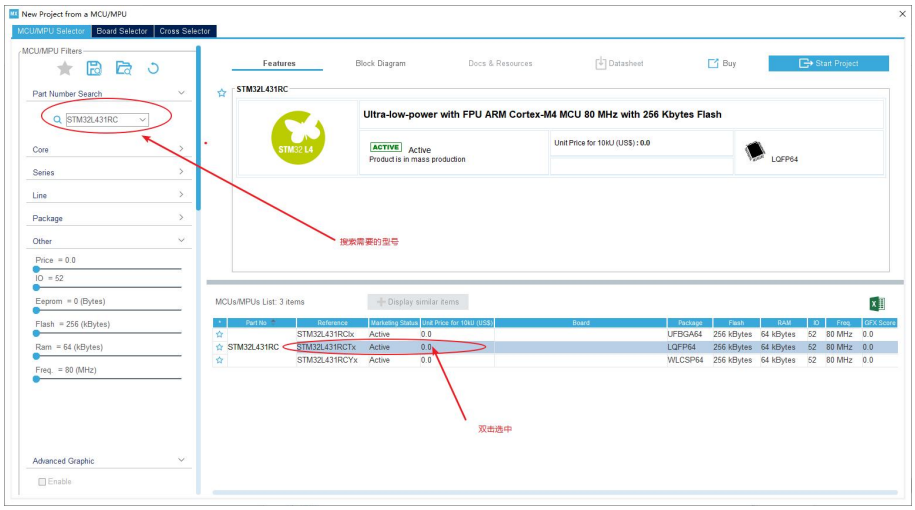
2. 生成 MDK 工程

选择芯片型号

打开 STM32CubeMX，打开 MCU 选择器：



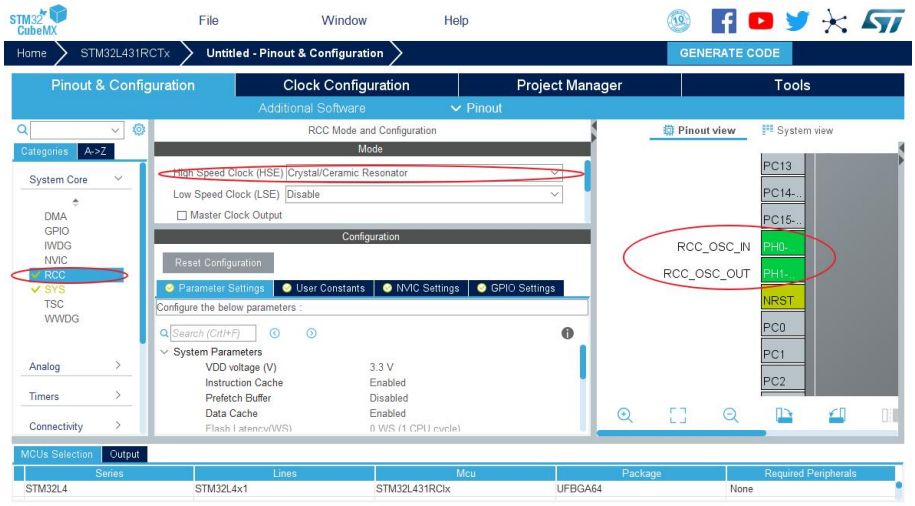
搜索并选中芯片 STM32L431RCT6:



配置时钟源

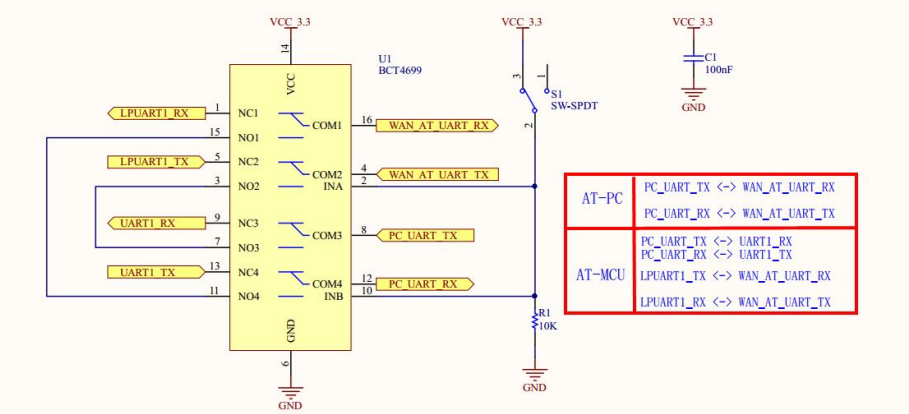
- 如果选择使用外部高速时钟（HSE），则需要在 System Core 中配置 RCC；
- 如果使用默认内部时钟（HSI），这一步可以略过；

这里我都使用外部时钟：



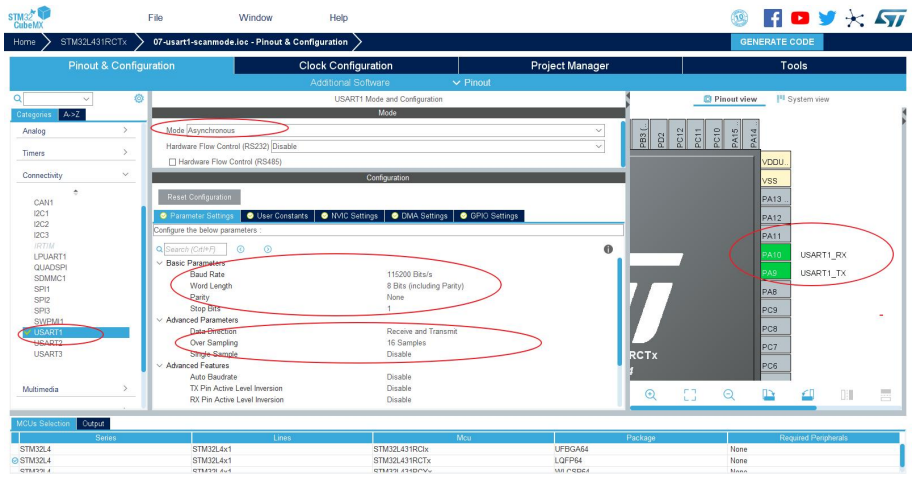
配置串口

小熊派开发板板载 ST-Link 并且虚拟了一个串口，原理图如下：



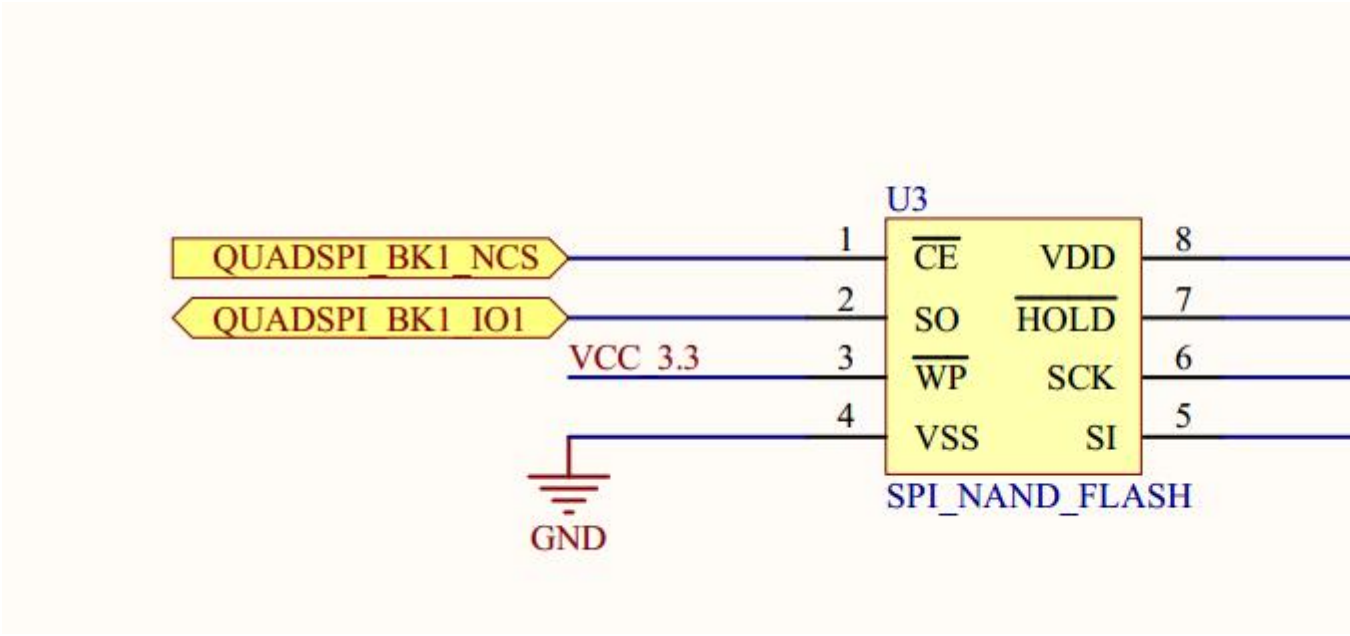
这里我将开关拨到 AT-MCU 模式，使 PC 的串口与 USART1 之间连接。

接下来开始配置 USART1：



配置 QSPI 接口

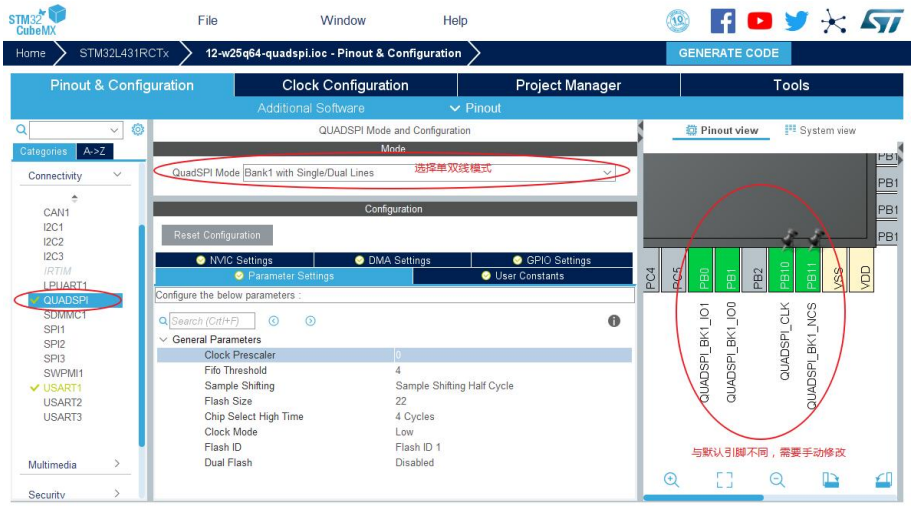
首先查看小熊派开发板上 SPI Flash 的原理图：



其引脚连接情况如下：

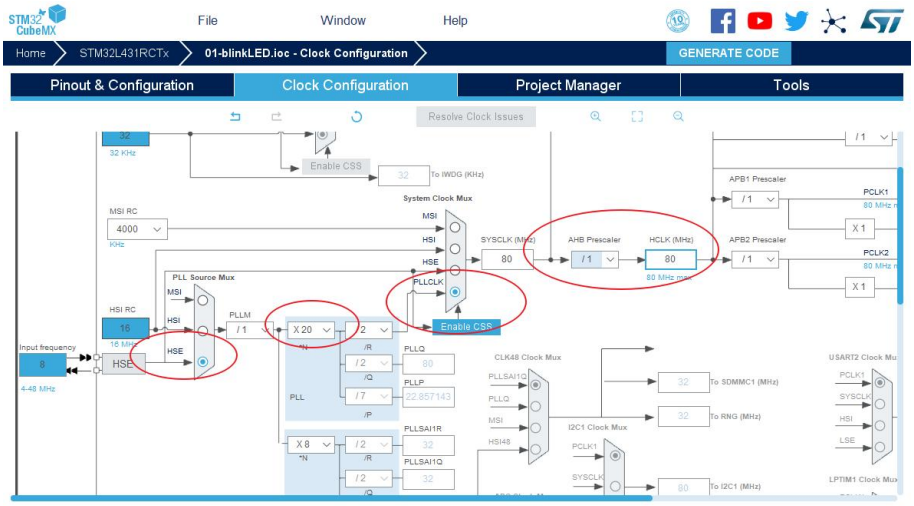
SPI Flash 连接引脚	对应引脚
QUADSPI_BK1_NCS	PB11
QUADSPI_BK1_CLK	PB10
QUADSPI_BK1_IO0	PB1
QUADSPI_BK1_IO1	PB0

接下来配置 QSPI 接口：

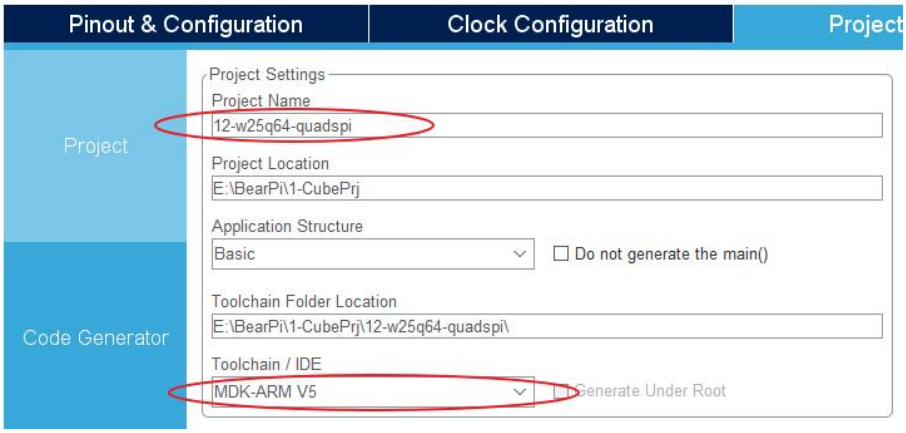


配置时钟树

STM32L4 的最高主频到 80M，所以配置 PLL，最后使 HCLK = 80Mhz 即可：

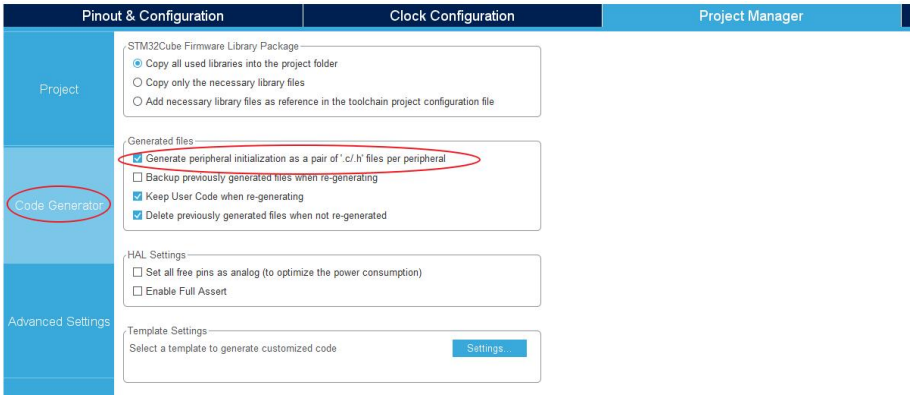


生成工程设置



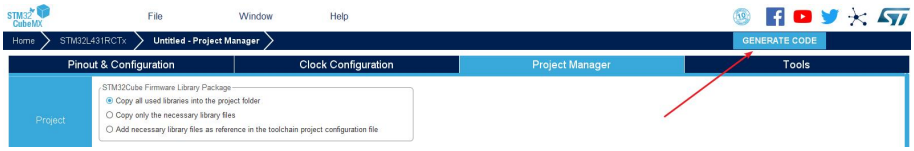
代码生成设置

最后设置生成独立的初始化文件：



生成代码

点击 GENERATE CODE 即可生成 MDK-V5 工程：



3. 在 MDK 中编写、编译、下载用户代码

重定向 printf( )函数

参考：重定向 printf 函数到串口输出的多种方法。

4. 封装 SPI Flash（W25Q64）的命令和底层函数

MCU 通过向 SPI Flash **发送各种命令** 来读写 SPI Flash 内部的寄存器，所以这种裸机驱动，首先要先宏定义出需要使用的命令，然后利用 HAL 库提供的库函数，封装出三个底层函数，**便于移植**：

- 向 SPI Flash 发送命令的函数
- 向 SPI Flash 发送数据的函数
- 从 SPI Flash 接收数据的函数

接下来开始编写代码~

宏定义操作命令

```
#define ManufactDeviceID_CMD   0x90#define READ_STATU_REGISTER_1   0x05#define
READ_STATU_REGISTER_2   0x35#define READ_DATA_CMD          0x03#define WRITE_ENABLE_CMD
0x06#define WRITE_DISABLE_CMD      0x04#define SECTOR_ERASE_CMD            0x20#define
CHIP_ERASE_CMD              0xc7#define PAGE_PROGRAM_CMD        0x02
```

封装发送命令的函数（重点）

```
/**
  * @brief          向 SPI Flash 发送指令
  * @param          instruction —— 要发送的指令
  * @param          address     —— 要发送的地址
  * @param          dummyCycles         —— 空指令周期数
  * @param          instructionMode —— 指令发送模式
  * @param          addressMode —— 地址发送模式
  * @param          addressSize         —— 地址大小
  * @param          dataMode    —— 数据发送模式
  * @retval     成功返回 HAL_OK
```

```
*/
HAL_StatusTypeDef QSPI_Send_Command(uint32_t instruction,
                                    uint32_t address,
                                    uint32_t dummyCycles,
                                    uint32_t instructionMode,
                                    uint32_t addressMode,
                                    uint32_t addressSize,
                                    uint32_t dataMode){
    QSPI_CommandTypeDef cmd;

    cmd.Instruction = instruction;                          //指令
    cmd.Address = address;                                  //地址
    cmd.DummyCycles = dummyCycles;                          //设置空指令周期数
    cmd.InstructionMode = instructionMode;                  //指令模式
    cmd.AddressMode = addressMode;                          //地址模式
    cmd.AddressSize = addressSize;                          //地址长度
    cmd.DataMode = dataMode;                                //数据模式
    cmd.SIOOMode = QSPI_SIOO_INST_EVERY_CMD;                //每次都发送指令
    cmd.AlternateByteMode = QSPI_ALTERNATE_BYTES_NONE; //无交替字节
    cmd.DdrMode = QSPI_DDR_MODE_DISABLE;                    //关闭 DDR 模式
    cmd.DdrHoldHalfCycle = QSPI_DDR_HHC_ANALOG_DELAY;
```

```
    return HAL_QSPI_Command(&hqspi, &cmd, 5000);}
```

封装发送数据的函数

```
/**
* @brief    QSPI 发送指定长度的数据
* @param    buf  —— 发送数据缓冲区首地址
* @param    size —— 要发送数据的字节数
 * @retval成功返回 HAL_OK
 */
HAL_StatusTypeDef QSPI_Transmit(uint8_t* send_buf, uint32_t size){
    hqspi.Instance->DLR = size - 1;                            //配置数据长度
    return HAL_QSPI_Transmit(&hqspi, send_buf, 5000);          //接收数据}
```

封装接收数据的函数

```
/**
 * @brief   QSPI 接收指定长度的数据
 * @param   buf  —— 接收数据缓冲区首地址
 * @param   size —— 要接收数据的字节数
 * @retval成功返回 HAL_OK
 */
HAL_StatusTypeDef QSPI_Receive(uint8_t* recv_buf, uint32_t size){
    hqspi.Instance->DLR = size - 1;                          //配置数据长度
    return HAL_QSPI_Receive(&hqspi, recv_buf, 5000);                              //接收数
据}
```

5. 编写 W25Q64 的驱动程序

接下来开始利用上一节封装的宏定义和底层函数，编写 W25Q64 的驱动程序：

读取 Manufacture ID 和 Device ID

读取 Flash 内部这两个 ID 有两个作用：

- 检测 SPI Flash 是否存在
- 可以根据 ID 判断 Flash 具体型号

数据手册上给出的操作时序如图：

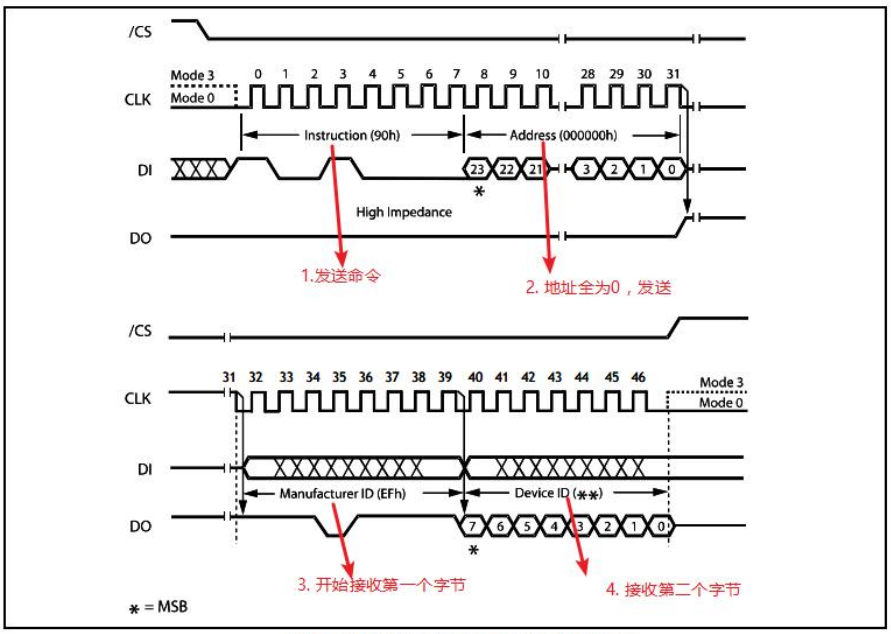


Figure 27. Read Manufacturer / Device ID Diagram

根据该时序，编写代码如下：

```
/**
 * @brief   读取 Flash 内部的 ID
 * @param   none
 * @retval 成功返回 device_id
 */
uint16_t W25QXX_ReadID(void){
        uint8_t recv_buf[2] = {0};     //recv_buf[0]存放 Manufacture ID, recv_buf[1]存放 Device ID
        uint16_t device_id = 0;
        if(HAL_OK == QSPI_Send_Command(ManufactDeviceID_CMD, 0, 0, QSPI_INSTRUCTION_1_LINE, QSPI_ADDRESS_1_LINE, QSPI_ADDRESS_24_BITS, QSPI_DATA_1_LINE))
        {
      //读取 ID
```

```
        if(HAL_OK == QSPI_Receive(recv_buf, 2))
        {
            device_id = (recv_buf[0] << 8) | recv_buf[1];
            return device_id;
        }
        else
        {
            return 0;
        }
          }
          else
          {
                    return 0;
          }}
```

读取数据

SPI Flash 读取数据可以任意地址（地址长度 32bit）读任意长度数据（最大 65535 Byte），没有任何限制，数据手册给出的时序如下：

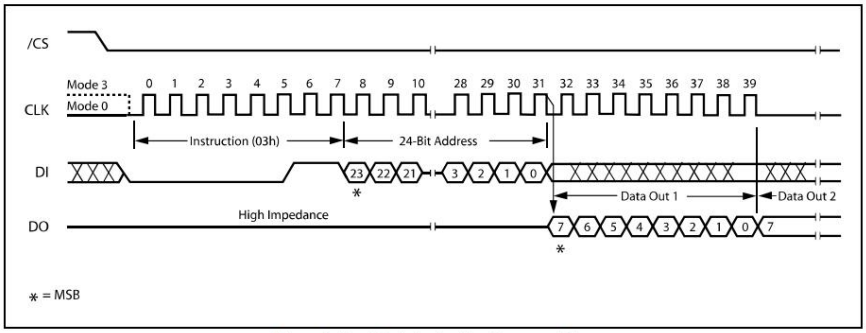


Figure 8. Read Data Instruction Sequence Diagram

根据该时序图编写代码如下：

```
/**
 * @brief 读取 SPI FLASH 数据
```

```
 * @param   dat_buffer —— 数据存储区

 * @param   start_read_addr —— 开始读取的地址(最大 32bit)

 * @param   byte_to_read —— 要读取的字节数(最大 65535)

 * @retval  none

 */void W25QXX_Read(uint8_t* dat_buffer, uint32_t start_read_addr, uint16_t byte_to_read){

        QSPI_Send_Command(READ_DATA_CMD, start_read_addr, 0, QSPI_INSTRUCTION_1_LINE,
QSPI_ADDRESS_1_LINE, QSPI_ADDRESS_24_BITS, QSPI_DATA_1_LINE);

   QSPI_Receive(dat_buffer, byte_to_read);}
```

读取状态寄存器数据并判断 Flash 是否忙碌

上文中提到，SPI Flash 的所有操作都是靠发送命令完成的，但是 Flash 接收到命令后，需要一段时间去执行该操作，这段时间内 Flash 处于“忙”状态，MCU 发送的命令无效，不能执行，在 Flash 内部有 2-3 个状态寄存器，指示出 Flash 当前的状态，有趣的一点是：

当 Flash 内部在执行命令时，不能再执行 MCU 发来的命令，但是 MCU 可以一直读取状态寄存器，这下就很好办了，**MCU 可以一直读取，然后判断 Flash 是否忙完：**

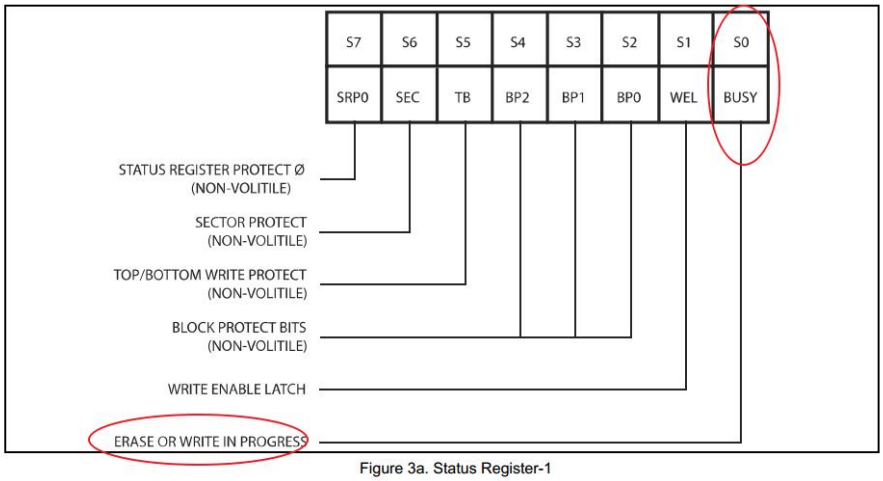


Figure 3a. Status Register-1

首先读取状态寄存器的代码如下：

```
/**

 * @brief 读取 W25QXX 的状态寄存器，W25Q64 一共有 2 个状态寄存器

 * @param reg  —— 状态寄存器编号(1~2)

 * @retval状态寄存器的值

 */
```

```
uint8_t W25QXX_ReadSR(uint8_t reg){
        uint8_t cmd = 0, result = 0;
        switch(reg)
        {
                case 1:
                        /* 读取状态寄存器 1 的值 */
                        cmd = READ_STATU_REGISTER_1;
                case 2:
                        cmd = READ_STATU_REGISTER_2;
                case 0:
                default:
                        cmd = READ_STATU_REGISTER_1;
        }
        QSPI_Send_Command(cmd, 0, 0, QSPI_INSTRUCTION_1_LINE, QSPI_ADDRESS_NONE,
QSPI_ADDRESS_24_BITS, QSPI_DATA_1_LINE);
        QSPI_Receive(&result, 1);

        return result;}
```

然后编写**阻塞判断** Flash 是否忙碌的函数：

```
/**
 * @brief 阻塞等待 Flash 处于空闲状态
 * @param   none
 * @retval  none
 */void W25QXX_Wait_Busy(void){
    while((W25QXX_ReadSR(1) & 0x01) == 0x01); // 等待 BUSY 位清空}
```

写使能/禁止

Flash 芯片默认禁止写数据，所以**在向 Flash 写数据之前，必须发送命令开启写使能**，数据手册中给出的时序如下：

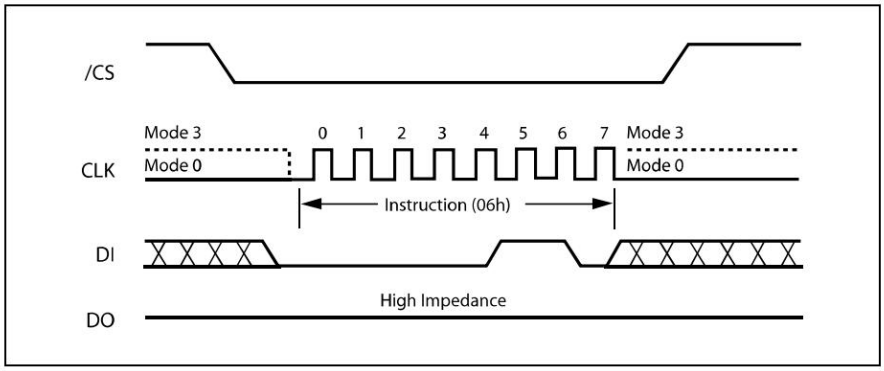


Figure 4. Write Enable Instruction Sequence Diagram

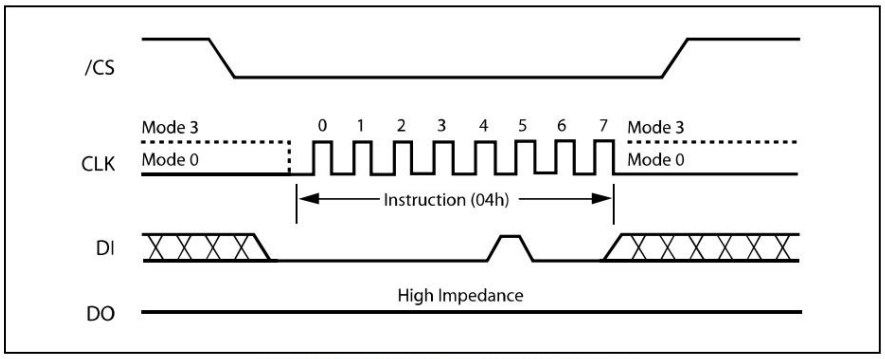


Figure 5. Write Disable Instruction Sequence Diagram

编写函数如下：

```
/**
 * @brief W25QXX 写使能,将 S1 寄存器的 WEL 置位
 * @param none
 * @retval
 */void W25QXX_Write_Enable(void){
    QSPI_Send_Command(WRITE_ENABLE_CMD, 0, 0, QSPI_INSTRUCTION_1_LINE, QSPI_ADDRESS_NONE,
QSPI_ADDRESS_8_BITS, QSPI_DATA_NONE);
                    W25QXX_Wait_Busy();}
/**
 * @brief W25QXX 写禁止,将 WEL 清零
```

```
 * @param none

 * @retval none

 */void W25QXX_Write_Disable(void){

    QSPI_Send_Command(WRITE_DISABLE_CMD, 0, 0, QSPI_INSTRUCTION_1_LINE, QSPI_ADDRESS_NONE,
QSPI_ADDRESS_8_BITS, QSPI_DATA_NONE);

                    W25QXX_Wait_Busy();}
```

擦除扇区

SPI Flash 有个特性：

数据位可以由 1 变为 0，但是不能由 0 变为 1。

所以在向 Flash 写数据之前，必须要先进行擦除操作，并且 Flash **最小只能擦除一个扇区**，擦除之后该扇区所有的数据变为 0xFF（即全为 1），数据手册中给出的时序如下：

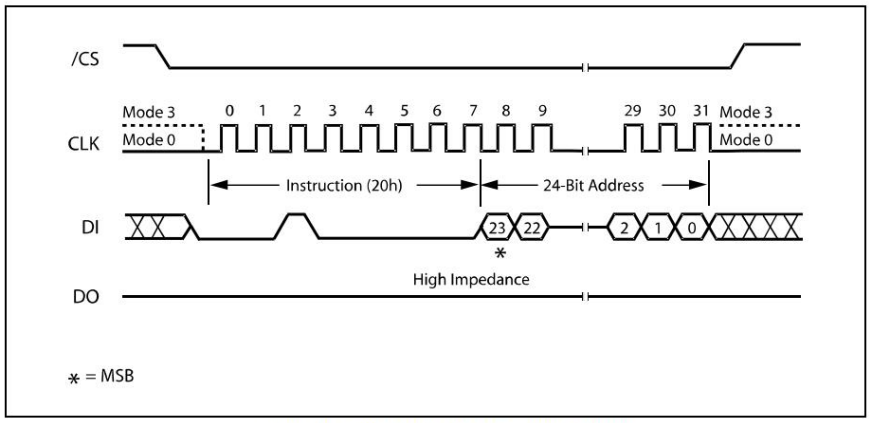


Figure 17. Sector Erase Instruction Sequence Diagram

根据此时序编写函数如下：

```
/**

 * @brief W25QXX 擦除一个扇区

 * @param   sector_addr        —— 扇区地址 根据实际容量设置

 * @retval  none

 * @note  阻塞操作

 */void W25QXX_Erase_Sector(uint32_t sector_addr){

    sector_addr *= 4096;        //每个块有 16 个扇区，每个扇区的大小是 4KB，需要换算为实际地址

    W25QXX_Write_Enable();  //擦除操作即写入 0xFF，需要开启写使能
```

```
    W25QXX_Wait_Busy();                    //等待写使能完成

    QSPI_Send_Command(SECTOR_ERASE_CMD, sector_addr, 0, QSPI_INSTRUCTION_1_LINE,
QSPI_ADDRESS_1_LINE, QSPI_ADDRESS_24_BITS, QSPI_DATA_NONE);

    W25QXX_Wait_Busy();          //等待扇区擦除完成}
```

页写入操作

向 Flash 芯片写数据的时候，因为 Flash 内部的构造，可以按页写入：

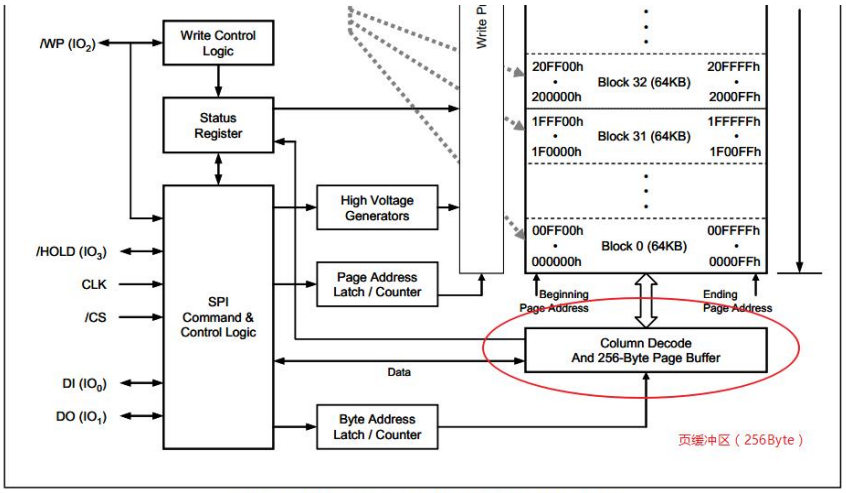


Figure 2. W25Q64BV Serial Flash Memory Block Diagram

页写入的时序如图：

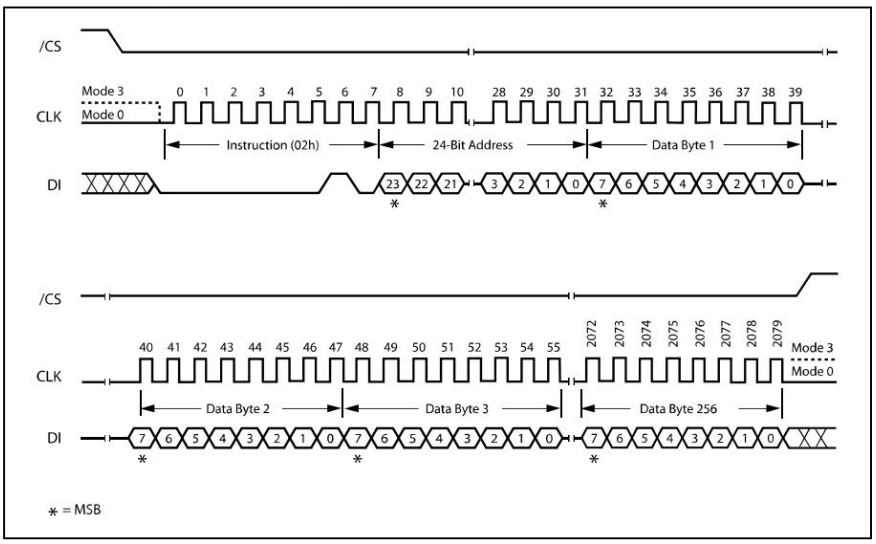


Figure 15. Page Program Instruction Sequence Diagram

编写代码如下：

```
/**

 * @brief 页写入操作
```

```
 * @param dat —— 要写入的数据缓冲区首地址

 * @param WriteAddr —— 要写入的地址

 * @param   byte_to_write —— 要写入的字节数（0-256）

 * @retval none

 */void W25QXX_Page_Program(uint8_t* dat, uint32_t WriteAddr, uint16_t byte_to_write){

          W25QXX_Write_Enable();

          QSPI_Send_Command(PAGE_PROGRAM_CMD, WriteAddr, 0, QSPI_INSTRUCTION_1_LINE,
QSPI_ADDRESS_1_LINE, QSPI_ADDRESS_24_BITS, QSPI_DATA_1_LINE);

          QSPI_Transmit(dat, byte_to_write);

    W25QXX_Wait_Busy();}
```

6. 测试驱动

在 `main.c` 函数中编写代码，测试驱动：

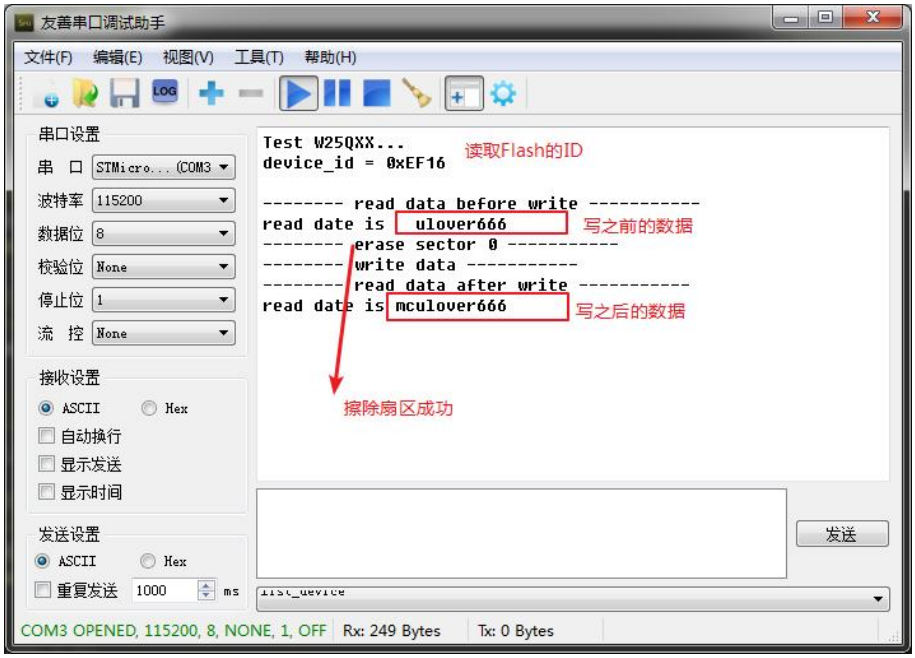
首先定义两个缓存：

```
/* Private user code ---------------------------------------------------------*//* USER CODE
BEGIN 0 */

uint8_t dat[11] = "mculover666";

uint8_t read_buf[11] = {0};/* USER CODE END 0 */
```

然后在 main 函数中编写代码：

```
/* USER CODE BEGIN 2 */printf("Test W25QXX...\r\n");

device_id = W25QXX_ReadID();printf("device_id = 0x%04X\r\n\r\n", device_id);

/* 为了验证，首先读取要写入地址处的数据 */printf("-------- read data before write
----------\r\n");W25QXX_Read(read_buf, 5, 11);printf("read date is %s\r\n", (char*)read_buf);

/* 擦除该扇区 */printf("-------- erase sector 0 ----------\r\n");W25QXX_Erase_Sector(0);

/* 写数据 */printf("-------- write data ----------\r\n");W25QXX_Page_Program(dat, 5, 11);

/* 再次读数据 */printf("-------- read data after write ----------\r\n");W25QXX_Read(read_buf,
5, 11);printf("read date is %s\r\n", (char*)read_buf);/* USER CODE END 2 */
```

测试结果如下：



至此，我们已经学会**如何使用硬件 QSPI 接口读写 SPI Flash 的数据**，下一节将讲述如何使用硬件 SDMMC 接口读取 SD 卡数据。

作业

编写程序，通过 QSPI 接口读写 W25Q64 SPI Flash 中的数据。

分析 QSPI 与 SPI 在高速存储访问中的性能差异

STM32 单片机基础 19——使用 SDMMC 接口读写 SD 卡数据

教学目的与要求：

目的：掌握 SD 卡的数据存储和访问方法，了解 SDMMC 接口的工作原理。

要求：能够配置 STM32 的 SDMMC 接口，编写程序实现 SD 卡的初始化、文件系统的挂载、文件的读写等操作。

教学重难点：

重点：SDMMC 接口的配置和 SD 卡文件系统的操作。

难点：理解 SD 卡文件系统的工作原理，确保文件操作的稳定性和数据的安全性。

课时数：3 课时

思政元素：

培养学生的信息存储和安全管理意识，通过 SD 卡读写的学习，让学生认识到数据存储在现代社会中的重要性，以及数据安全和隐私保护的必要性，培养学生的信息安全意识和社会责任感。

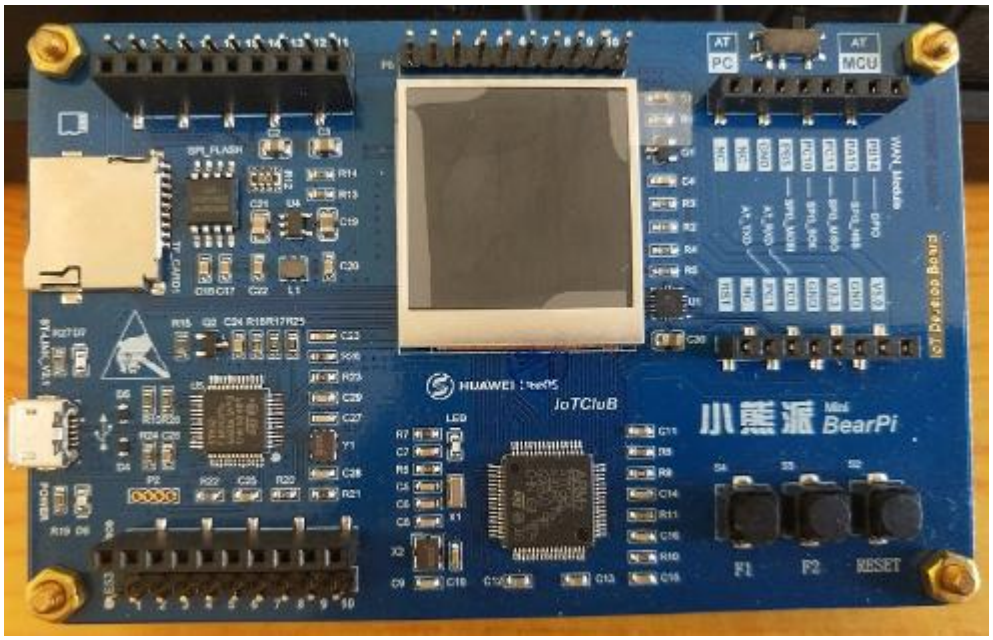
本篇详细的记录了如何使用 STM32CubeMX 配置 STM32L431RCT6 的硬件 SDMMC 外设读取 SD 卡数据。

1. 准备工作

硬件准备

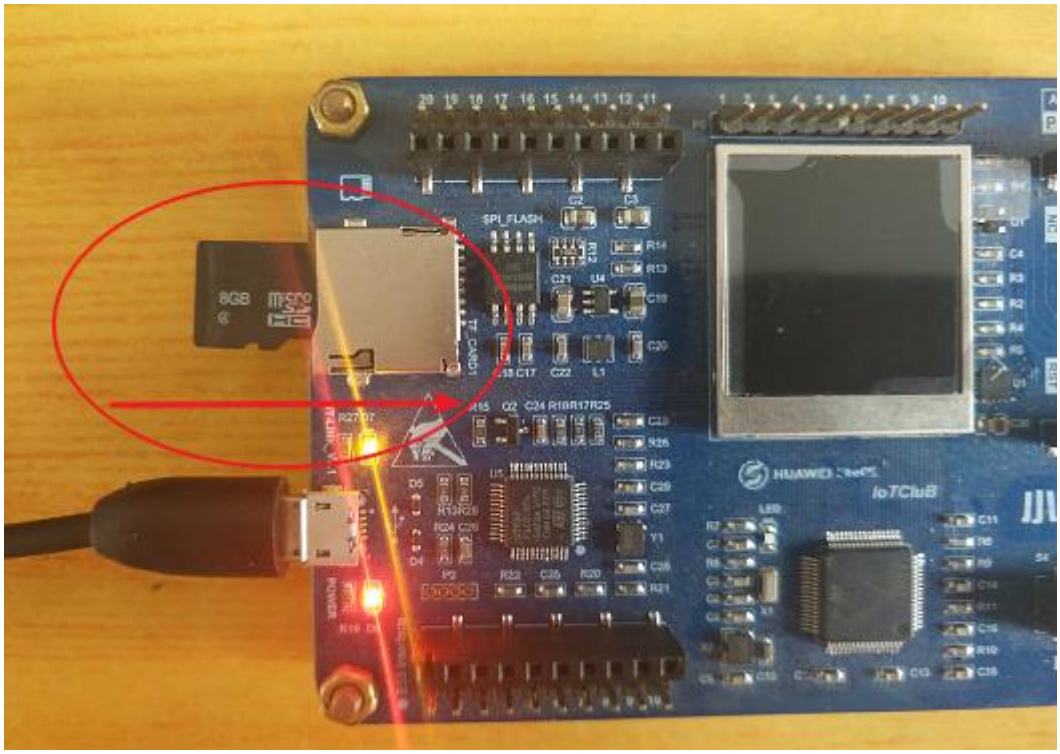
开发板

首先需要准备一个开发板，这里我准备的是 STM32L4 的开发板（BearPi）：



- Micro SD 卡

小熊派开发板板载 Micro SD 卡槽，最大支持 32 GB，需要提前自行准备一张 Micro SD 卡，如图：



软件准备

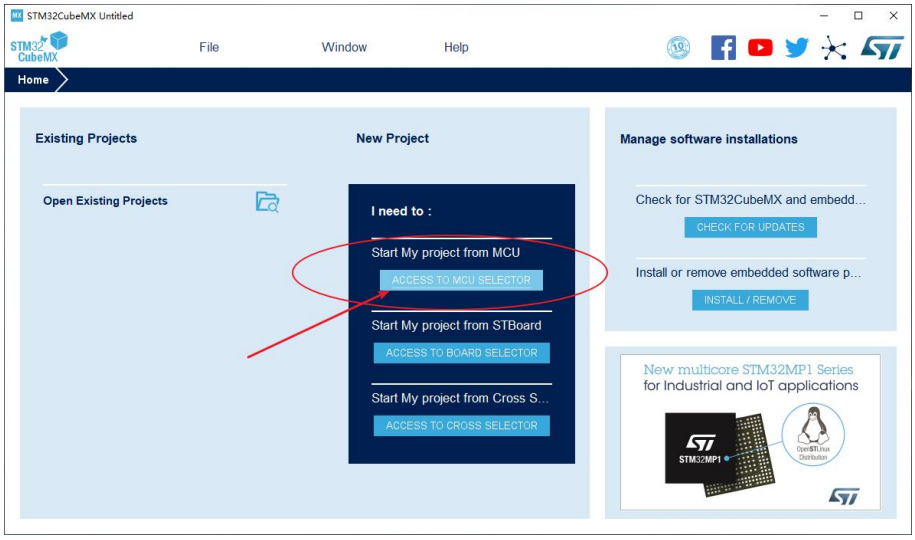
- 需要安装好 Keil - MDK 及芯片对应的包，以便编译和下载生成的代码；

Keil MDK 和串口助手的安装包都可以关注“**小熊派开源社区**”微信公众号，在资料教程一栏中可获取安装包。

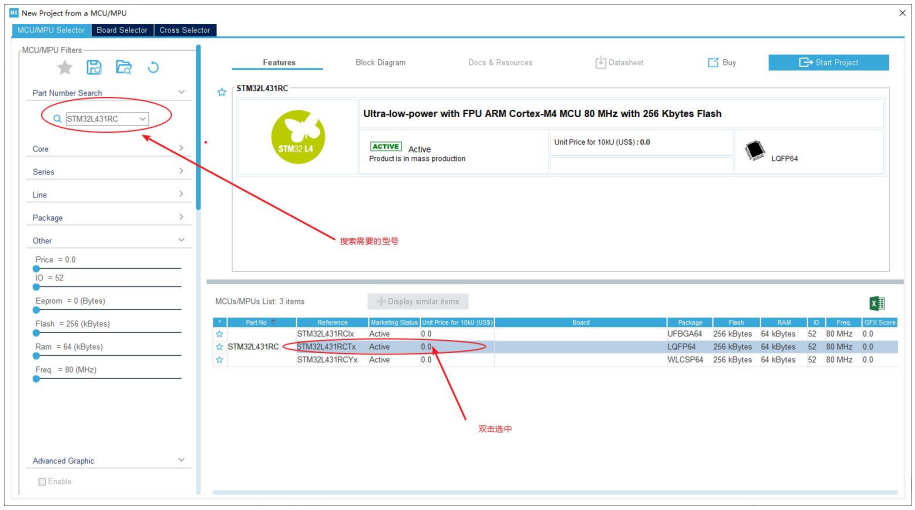
2.生成 MDK 工程

选择芯片型号

打开 STM32CubeMX，打开 MCU 选择器：



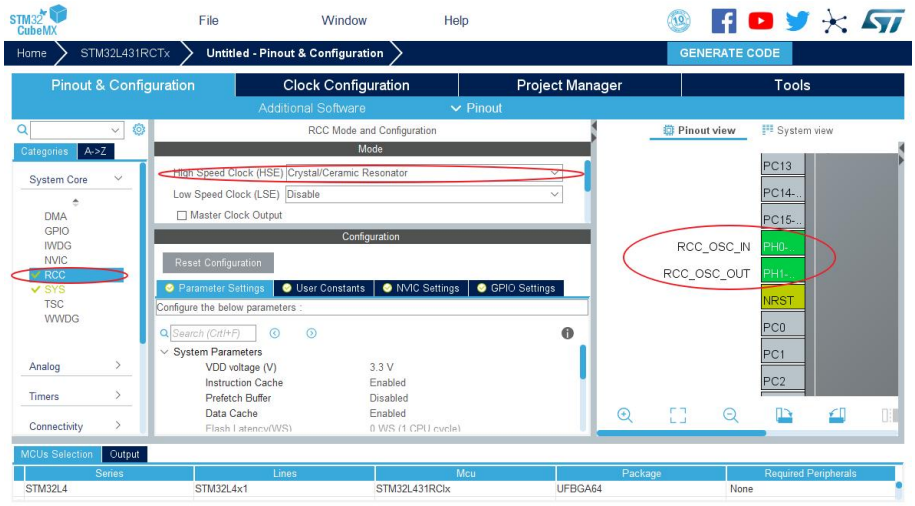
搜索并选中芯片 STM32L431RCT6：



配置时钟源

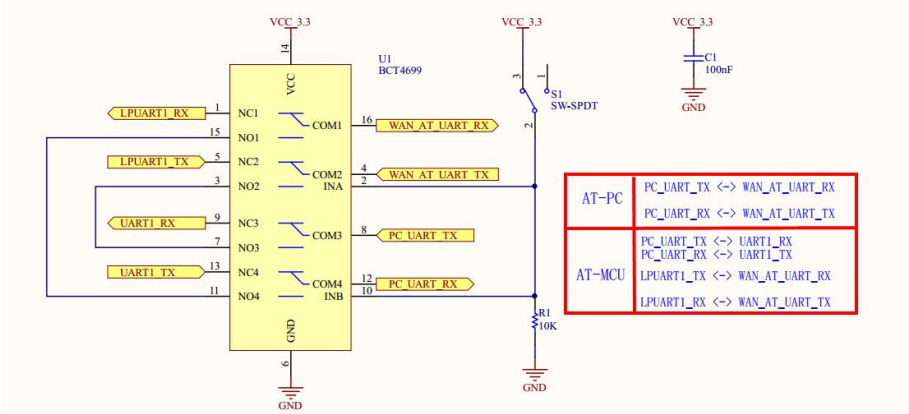
- 如果选择使用外部高速时钟（HSE），则需要在 System Core 中配置 RCC；
- 如果使用默认内部时钟（HSI），这一步可以略过；

这里我都使用外部时钟：



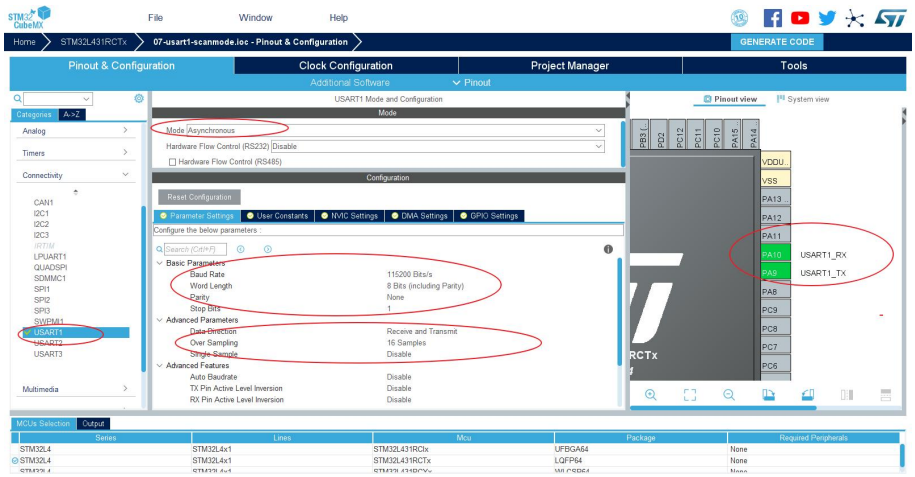
配置串口

小熊派开发板板载 ST-Link 并且虚拟了一个串口，原理图如下：



这里我将开关拨到 AT-MCU 模式，使 PC 的串口与 USART1 之间连接。

接下来开始配置 USART1：



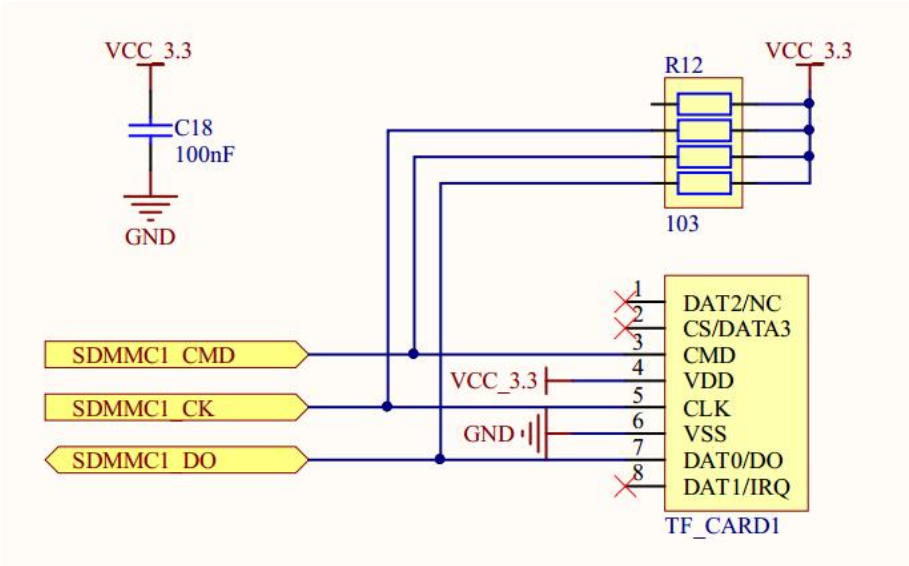
配置 SDMMC 接口

知识小卡片 —— SDMMC 接口

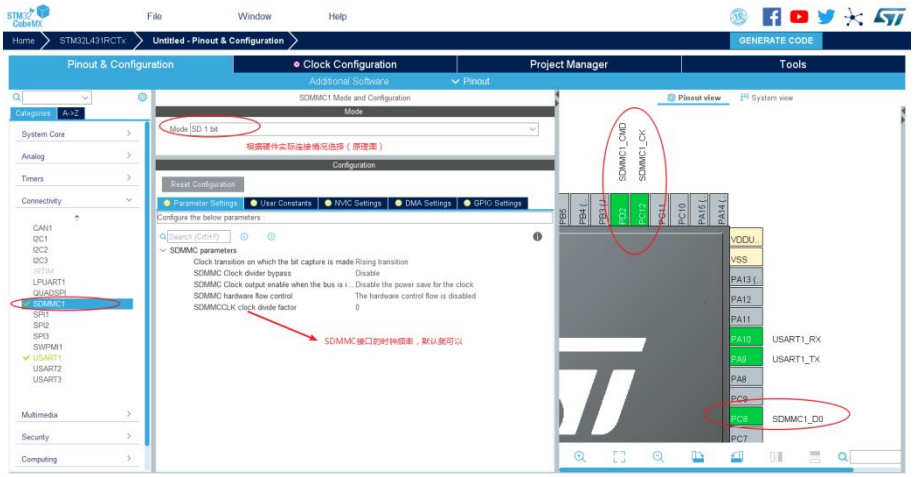
SDMMC 接口的全称叫 SD/SDIO MMC card host interface，SD/SDIO MMC 卡 主机接口，通俗的来说，就是这个接口支持 SD 卡，支持 SDIO 设备，支持 MMC 卡。

知识小卡片结束啦~

首先查看小熊派开发板的原理图：

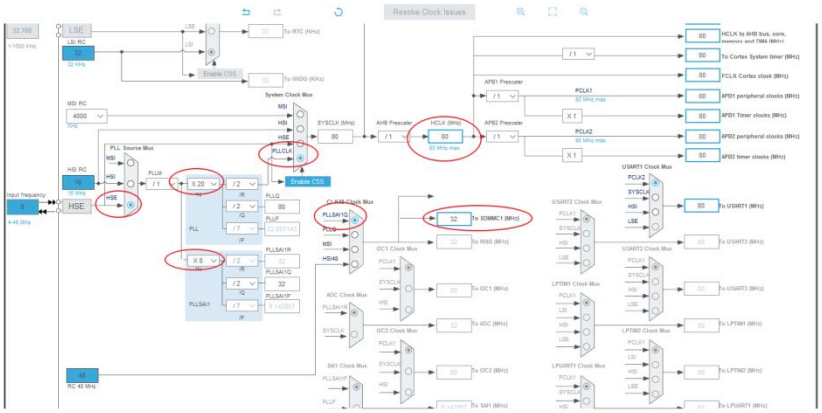


然后根据原理图配置 SDMMC 接口：

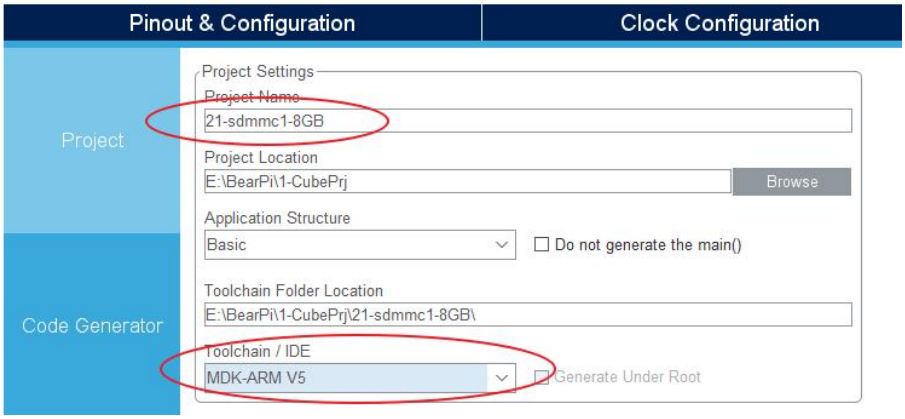


配置时钟树

STM32L4 的最高主频到 80M，所以配置 PLL，最后使 HCLK = 80Mhz 即可：

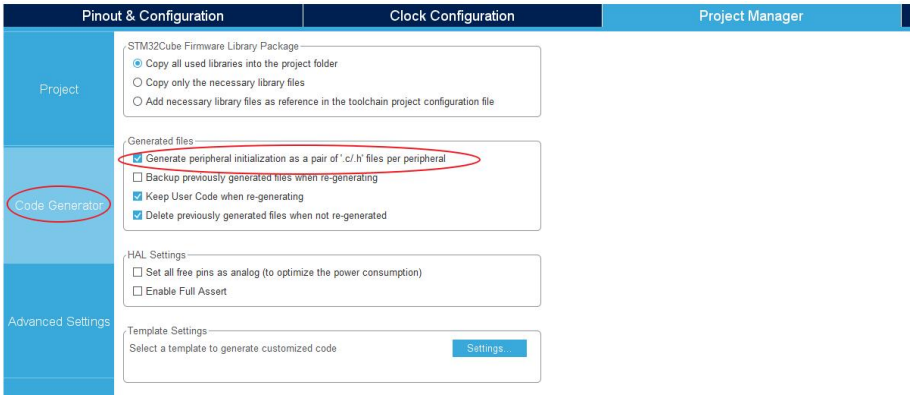


生成工程设置



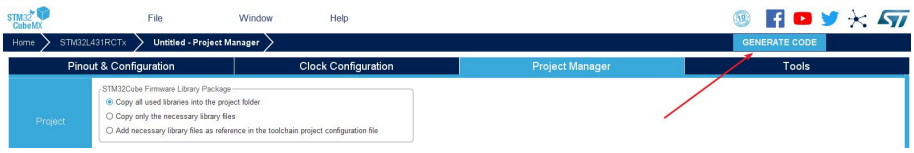
代码生成设置

最后设置生成独立的初始化文件：



生成代码

点击 GENERATE CODE 即可生成 MDK-V5 工程：



3. 在 MDK 中编写、编译、下载用户代码

重定向 printf( )函数

参考：重定向 printf 函数到串口输出的多种方法。

读取 SD 卡信息并打印

SD 卡系统(包括主机和 SD 卡)定义了两种操作模式：

- 卡识别模式
- 数据传输模式

在**系统复位后，主机处于卡识别模式**，寻找总线上可用的 SD 卡设备；同时，SD 卡也处于卡识别模式，直到被主机识别到。

使用 STM32CubeMX 初始化的工程中会自动生成 SDMMC 初始化函数，向 SD 卡发送命令，**当 SD 卡接收到命令后， SD 卡就会进入数据传输模式，而主机在总线上所有卡被识别后也进入数据传输模式**。

所以在操作之前，需要先检查 SD 卡是否处于数据传输模式并且处于数据传输状态：

在 main 函数中首先定义一个变量用于存储 SD 卡状态：

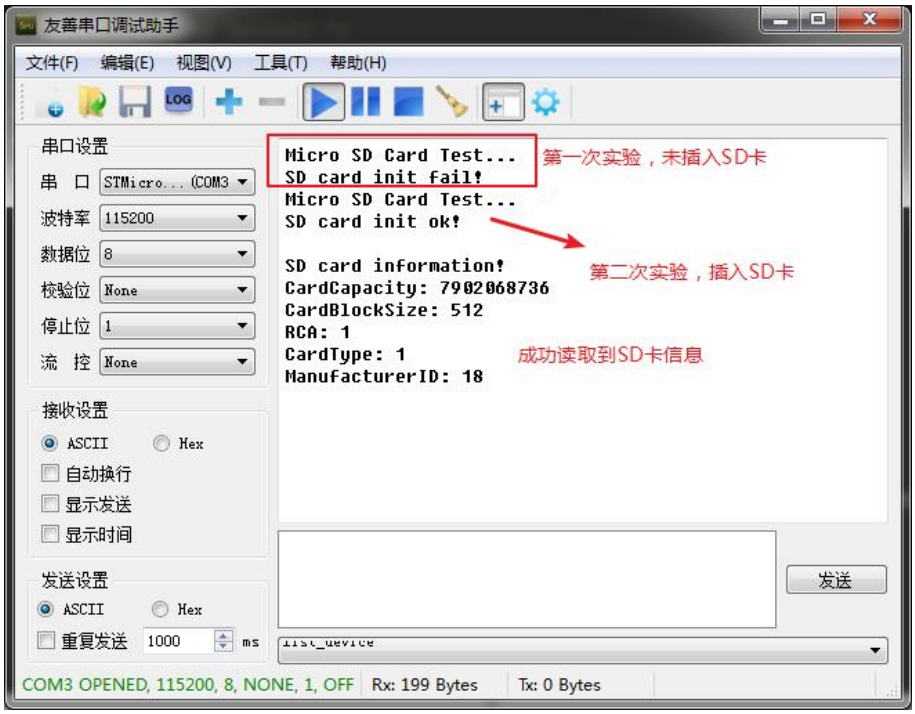
```
int sdcard_status = 0;

HAL_SD_CardCIDTypeDef sdcard_cid;
```

然后在 while(1)之前编写如下读取信息代码：

```
/* USER CODE BEGIN 2 */printf("Micro SD Card Test...\r\n");

/* 检测 SD 卡是否正常（处于数据传输模式的传输状态） */

sdcard_status = HAL_SD_GetCardState(&hsd1);if(sdcard_status == HAL_SD_CARD_TRANSFER){

    printf("SD card init ok!\r\n\r\n");

    

    //打印 SD 卡基本信息

    printf("SD card information!\r\n");

    printf("CardCapacity: %llu\r\n",((unsigned long
long)hsd1.SdCard.BlockSize*hsd1.SdCard.BlockNbr));

    printf("CardBlockSize: %d \r\n",hsd1.SdCard.BlockSize);

    printf("RCA: %d \r\n",hsd1.SdCard.RelCardAdd);

    printf("CardType: %d \r\n",hsd1.SdCard.CardType);
```

```
    //读取并打印 SD 卡的 CID 信息
    HAL_SD_GetCardCID(&hsd1,&sdcard_cid);
    printf("ManufacturerID: %d \r\n",sdcard_cid.ManufacturerID);}else{
    printf("SD card init fail!\r\n" );
    return 0;}/* USER CODE END 2 */
```

编译下载后串口助手输出结果如下：



擦除 SD 卡块数据

为了验证实验的正确性或，先擦除数据：

```
/* 擦除 SD 卡块 */printf("------------------- Block Erase
--------------------------------\r\n");

sdcard_status = HAL_SD_Erase(&hsd1, 0, 512);if (sdcard_status == 0){
    printf("Erase block ok\r\n");}else{
    printf("Erase block fail\r\n");}
```

读取 SD 卡块数据

首先开辟一个**全局**缓冲区，用于存放从 SD 卡读出的数据：

```
/* Private user code ---------------------------------------------------------*//* USER CODE BEGIN 0 */

uint8_t read_buf[512];/* USER CODE END 0 */
```

然后在之前读取信息的代码之后添加读取数据的代码：

```
/* 读取未操作之前的数据 */printf("------------------- Read SD card block data Test -------------------\r\n");

sdcard_status = HAL_SD_ReadBlocks(&hsd1,(uint8_t *)read_buf,0,1,0xffff);if(sdcard_status == 0){

    printf("Read block data ok \r\n" );

    for(i = 0; i < 512; i++)

    {

        printf("0x%02x ", read_buf[i]);

        if((i+1)%16 == 0)

        {

            printf("\r\n");

        }

    }}else{

    printf("Read block data fail!\r\n " );}
```

向 SD 卡块写入数据

同样的，开辟一个**全局**缓冲区，用于存放即将要写入 SD 卡的数据：

```
uint8_t write_buf[512];
```

然后在之前读取数据的代码之后添加的代码，将缓冲区的数据赋初值：

```
/* 填充缓冲区数据 */for(i = 0; i < 512; i++){

    write_buf[i] = i % 256;}
```

然后继续添加代码，将该缓冲区数据写入 SD 卡：

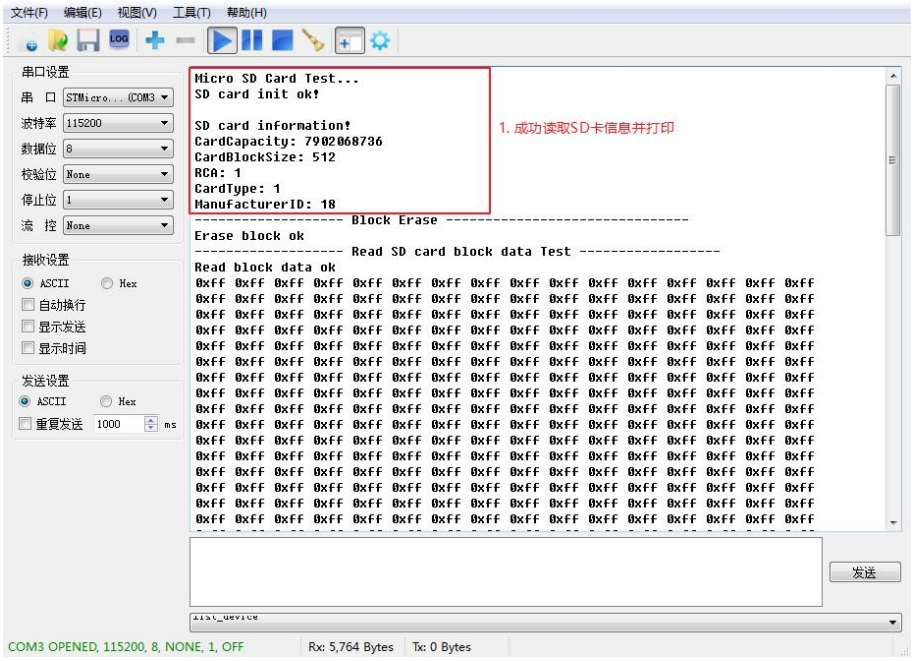
```
/* 向 SD 卡块写入数据 */printf("------------------- Write SD card block data Test -------------------\r\n");
```

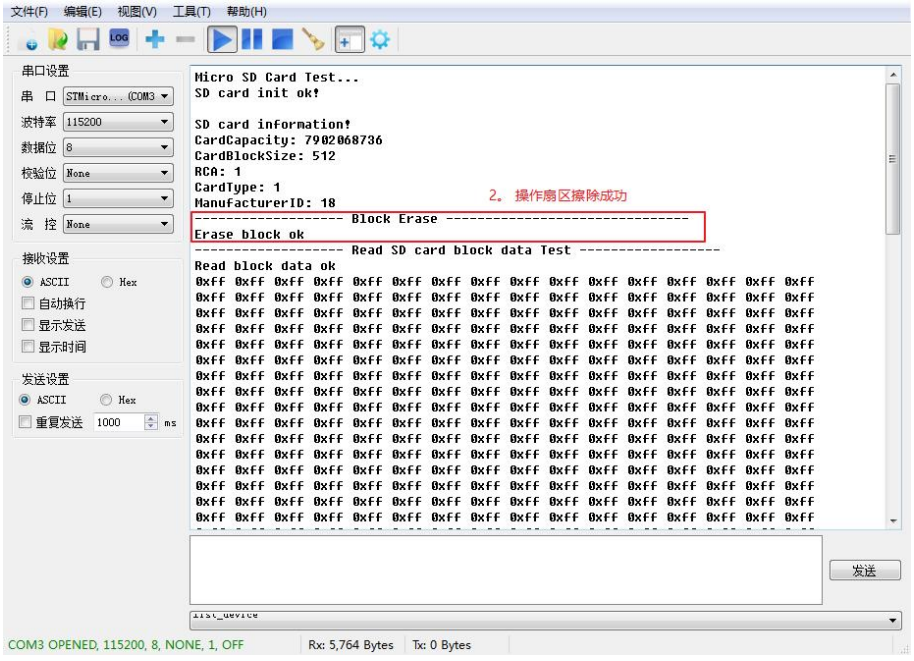
```
sdcard_status = HAL_SD_WriteBlocks(&hsd1,(uint8_t *)write_buf,0,1,0xffff);if(sdcard_status == 0){
    printf("Write block data ok \r\n" );}else{
    printf("Write block data fail!\r\n " );}
```

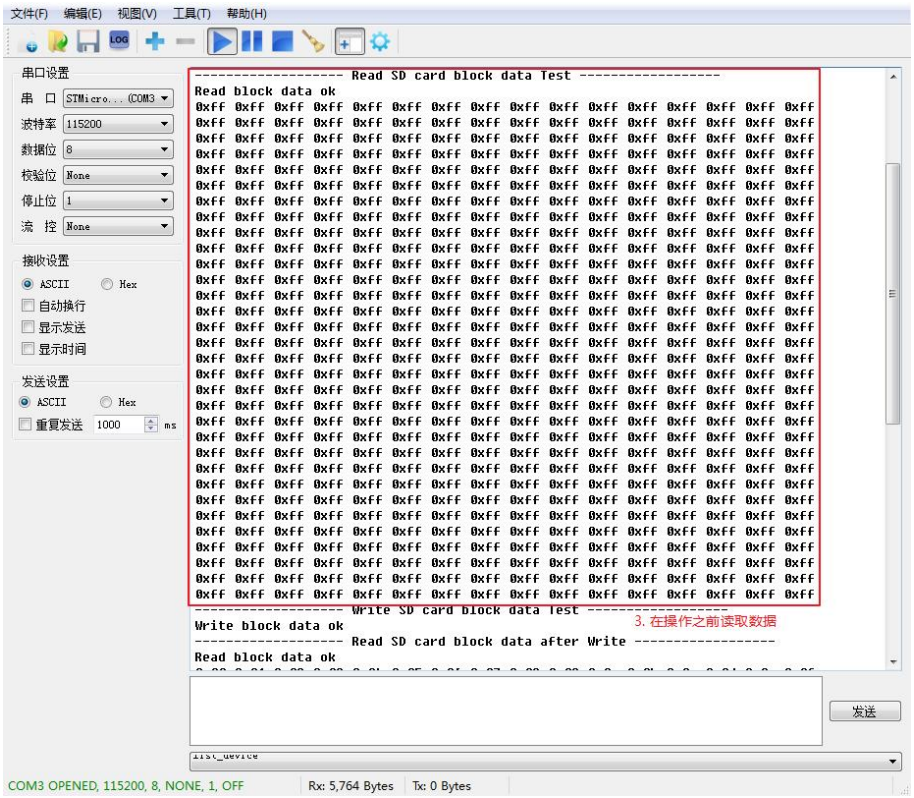
添加完之后，为了检查数据是否正常写入，再将数据读出：

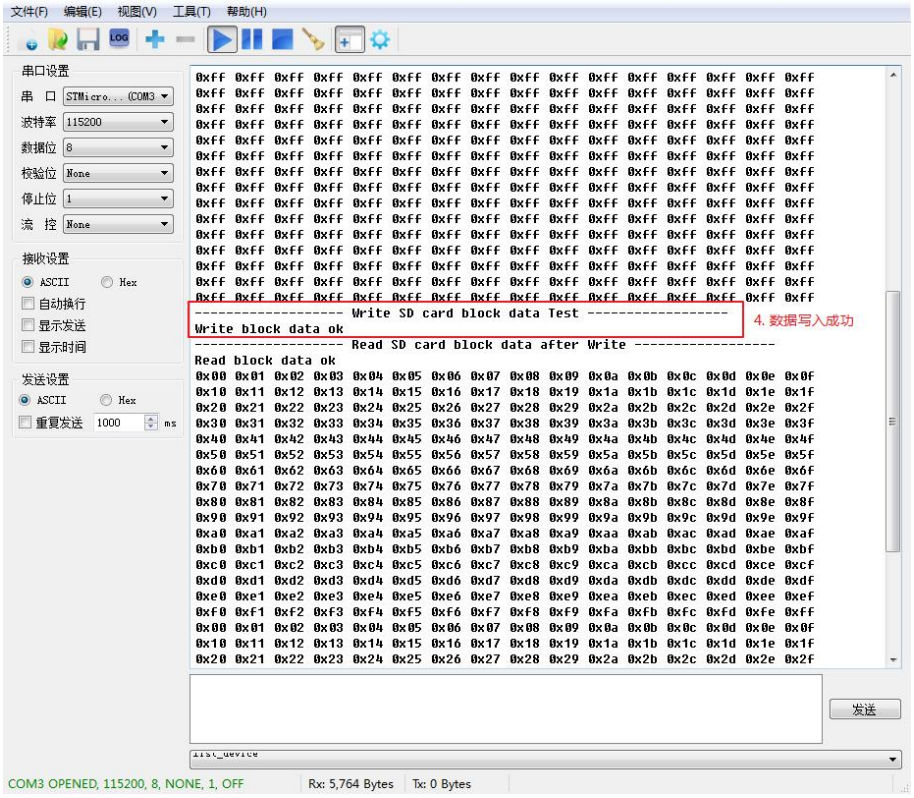
```
/* 读取操作之后的数据 */
printf("-------------------- Read SD card block data after Write ------------------\r\n");
sdcard_status = HAL_SD_ReadBlocks(&hsd1,(uint8_t *)read_buf,0,1,0xffff);
if(sdcard_status == 0)
{
    printf("Read block data ok \r\n" );
    for(i = 0; i < 512; i++)
    {
        printf("0x%02x ", read_buf[i]);
        if((i+1)%16 == 0)
        {
            printf("\r\n");
        }
    }
}
```

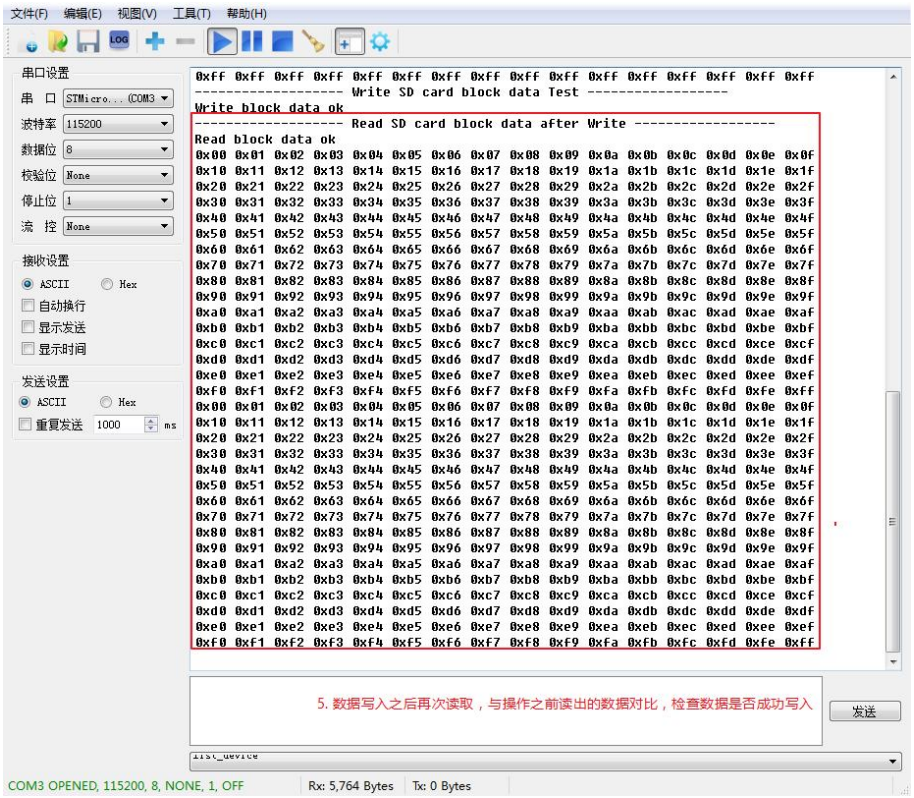
将程序编译下载，最终的实验结果如下：


文件(F) 编辑(E) 视图(V) 工具(T) 帮助(H)
串口设置
串 口 STMicro... (COM3
波特率 115200
数据位 8
校验位 None
停止位 1
流 控 None
接收设置
ASCII Hex
自动换行
显示发送
显示时间
发送设置
ASCII Hex
重复发送 1000 ms
Micro SD Card Test...
SD card init ok!
SD card information!
CardCapacity: 7902068736
CardBlockSize: 512
RCA: 1
CardType: 1
ManufacturerID: 18
1. 成功读取SD卡信息并打印
------------------- Block Erase --------------------------------
Erase block ok
------------------- Read SD card block data Test -------------------
Read block data ok
0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff
发送
COM3 OPENED, 115200, 8, NONE, 1, OFF
Rx: 5,764 Bytes
Tx: 0 Bytes


文件(F) 编辑(E) 视图(V) 工具(T) 帮助(H)
串口设置
串 口 STMicro... (COM3
波特率 115200
数据位 8
校验位 None
停止位 1
流 控 None
接收设置
ASCII Hex
自动换行
显示发送
显示时间
发送设置
ASCII Hex
重复发送 1000 ms
Micro SD Card Test...
SD card init ok!
SD card information!
CardCapacity: 7902068736
CardBlockSize: 512
RCA: 1
CardType: 1
ManufacturerID: 18
2. 操作扇区擦除成功
------------------- Block Erase --------------------------------
Erase block ok
------------------- Read SD card block data Test -------------------
Read block data ok
0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff
发送
COM3 OPENED, 115200, 8, NONE, 1, OFF
Rx: 5,764 Bytes
Tx: 0 Bytes


文件(F) 编辑(E) 视图(V) 工具(T) 帮助(H)
串口设置
串 口 STMicro... (COM3
波特率 115200
数据位 8
校验位 None
停止位 1
流 控 None
接收设置
ASCII Hex
自动换行
显示发送
显示时间
发送设置
ASCII Hex
重复发送 1000 ms
------------------- Read SD card block data Test -------------------
Read block data ok
0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff
------------------- Write SD card block data Test -------------------
Write block data ok
3. 在操作之前读取数据
------------------- Read SD card block data after Write -------------------
Read block data ok
发送
COM3 OPENED, 115200, 8, NONE, 1, OFF
Rx: 5,764 Bytes
Tx: 0 Bytes


文件(F) 编辑(E) 视图(V) 工具(T) 帮助(H)
串口设置
串 口 STMicro... (COM3
波特率 115200
数据位 8
校验位 None
停止位 1
流 控 None
接收设置
ASCII Hex
自动换行
显示发送
显示时间
发送设置
ASCII Hex
重复发送 1000 ms
0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff
------------------- Write SD card block data Test -------------------
Write block data ok
4. 数据写入成功
------------------- Read SD card block data after Write -------------------
Read block data ok
0x00 0x01 0x02 0x03 0x04 0x05 0x06 0x07 0x08 0x09 0x0a 0x0b 0x0c 0x0d 0x0e 0x0f
0x10 0x11 0x12 0x13 0x14 0x15 0x16 0x17 0x18 0x19 0x1a 0x1b 0x1c 0x1d 0x1e 0x1f
0x20 0x21 0x22 0x23 0x24 0x25 0x26 0x27 0x28 0x29 0x2a 0x2b 0x2c 0x2d 0x2e 0x2f
0x30 0x31 0x32 0x33 0x34 0x35 0x36 0x37 0x38 0x39 0x3a 0x3b 0x3c 0x3d 0x3e 0x3f
0x40 0x41 0x42 0x43 0x44 0x45 0x46 0x47 0x48 0x49 0x4a 0x4b 0x4c 0x4d 0x4e 0x4f
0x50 0x51 0x52 0x53 0x54 0x55 0x56 0x57 0x58 0x59 0x5a 0x5b 0x5c 0x5d 0x5e 0x5f
0x60 0x61 0x62 0x63 0x64 0x65 0x66 0x67 0x68 0x69 0x6a 0x6b 0x6c 0x6d 0x6e 0x6f
0x70 0x71 0x72 0x73 0x74 0x75 0x76 0x77 0x78 0x79 0x7a 0x7b 0x7c 0x7d 0x7e 0x7f
0x80 0x81 0x82 0x83 0x84 0x85 0x86 0x87 0x88 0x89 0x8a 0x8b 0x8c 0x8d 0x8e 0x8f
0x90 0x91 0x92 0x93 0x94 0x95 0x96 0x97 0x98 0x99 0x9a 0x9b 0x9c 0x9d 0x9e 0x9f
0xa0 0xa1 0xa2 0xa3 0xa4 0xa5 0xa6 0xa7 0xa8 0xa9 0xaa 0xab 0xac 0xad 0xae 0xaf
0xb0 0xb1 0xb2 0xb3 0xb4 0xb5 0xb6 0xb7 0xb8 0xb9 0xba 0xbb 0xbc 0xbd 0xbe 0xbf
0xc0 0xc1 0xc2 0xc3 0xc4 0xc5 0xc6 0xc7 0xc8 0xc9 0xca 0xcb 0xcc 0xcd 0xce 0xcf
0xd0 0xd1 0xd2 0xd3 0xd4 0xd5 0xd6 0xd7 0xd8 0xd9 0xda 0xdb 0xdc 0xdd 0xde 0xdf
0xe0 0xe1 0xe2 0xe3 0xe4 0xe5 0xe6 0xe7 0xe8 0xe9 0xea 0xeb 0xec 0xed 0xee 0xef
0xf0 0xf1 0xf2 0xf3 0xf4 0xf5 0xf6 0xf7 0xf8 0xf9 0xfa 0xfb 0xfc 0xfd 0xfe 0xff
0x00 0x01 0x02 0x03 0x04 0x05 0x06 0x07 0x08 0x09 0x0a 0x0b 0x0c 0x0d 0x0e 0x0f
0x10 0x11 0x12 0x13 0x14 0x15 0x16 0x17 0x18 0x19 0x1a 0x1b 0x1c 0x1d 0x1e 0x1f
0x20 0x21 0x22 0x23 0x24 0x25 0x26 0x27 0x28 0x29 0x2a 0x2b 0x2c 0x2d 0x2e 0x2f
发送
COM3 OPENED, 115200, 8, NONE, 1, OFF
Rx: 5,764 Bytes
Tx: 0 Bytes



至此，我们已经学会**如何使用硬件 SDMMC 接口读取 SD 数据**。

作业：编写程序，通过 SDMMC 接口读写 SD 卡中的数据。
分析 SDMMC 接口的工作原理和配置方法。

STM32 单片机基础 20——使用 DAC 输出任意指定电压

教学目的与要求：

目的：理解 DAC（数模转换器）的工作原理，掌握 DAC 的配置和使用方法，实现模拟信号的精确控制。

要求：能够配置 STM32 的 DAC 模块，编写程序输出指定电压值，并通过 ADC 进行验证，确保输出的准确性。

教学重难点：

重点：DAC 模块的配置和电压输出控制。

难点：理解 DAC 的分辨率和精度，确保输出电压的精确性。

课时数：3 课时

思政元素：

培养学生的精确控制能力和科学素养，通过 DAC 输出电压的学习，让学生认识到精确控制在科学研究和技术应用中的重要性，培养学生的严谨态度和科学精神。

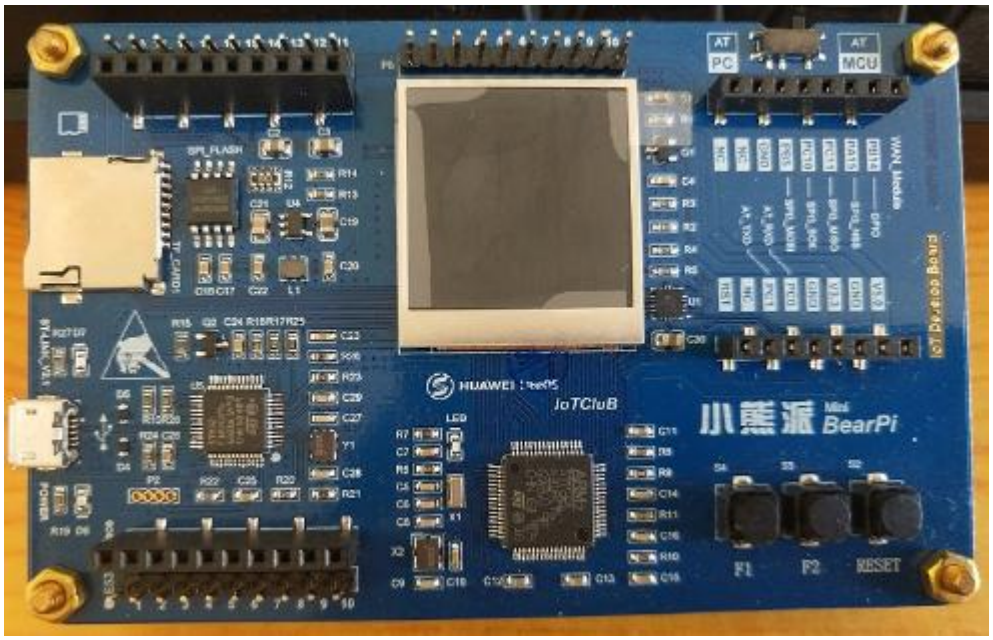
本篇详细的记录了如何使用 STM32CubeMX 配置 STM32L431RCT6 的 DAC 外设，输出任意指定电压值。

1. 准备工作

硬件准备

开发板

首先需要准备一个开发板，这里我准备的是 STM32L4 的开发板（BearPi）：



- 万用表

软件准备

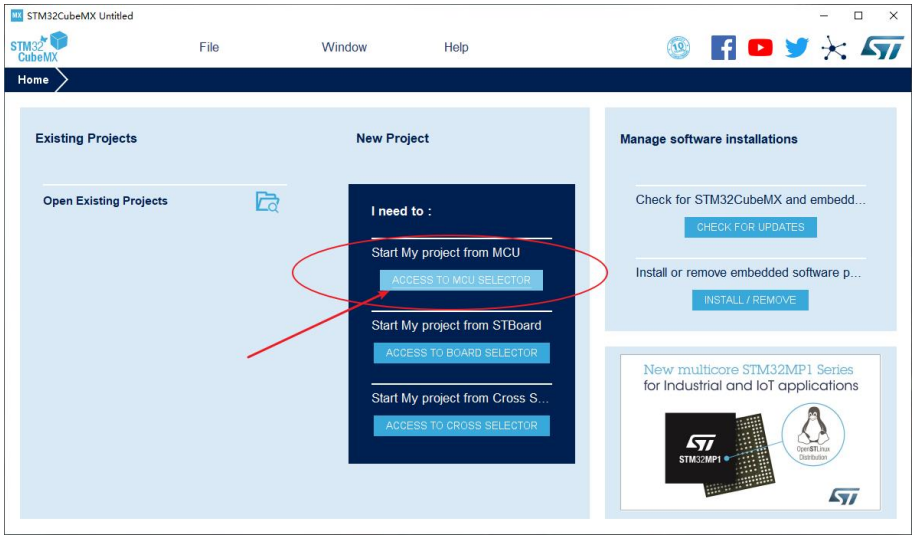
- 需要安装好 Keil - MDK 及芯片对应的包，以便编译和下载生成的代码；

Keil MDK 和串口助手的安装包都可以关注“**小熊派开源社区**”微信公众号，在资料教程一栏中可获取安装包。

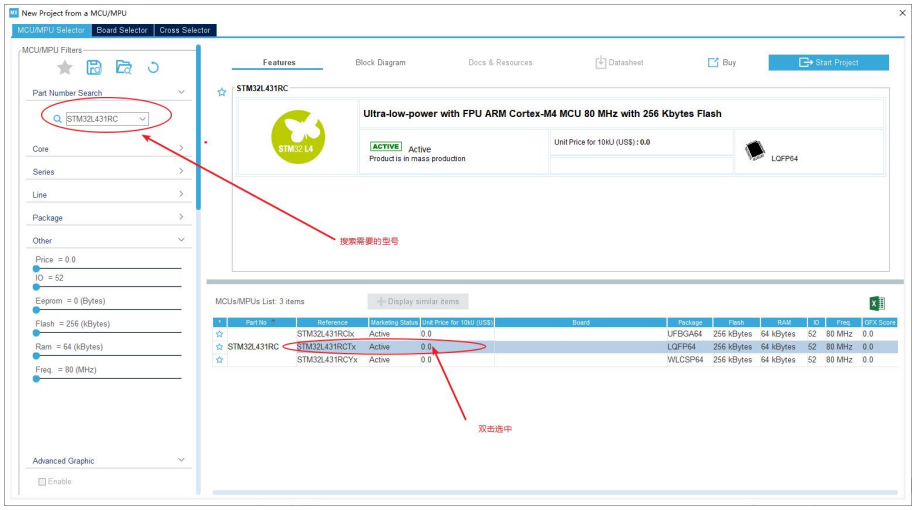
2. 生成 MDK 工程

选择芯片型号

打开 STM32CubeMX，打开 MCU 选择器：



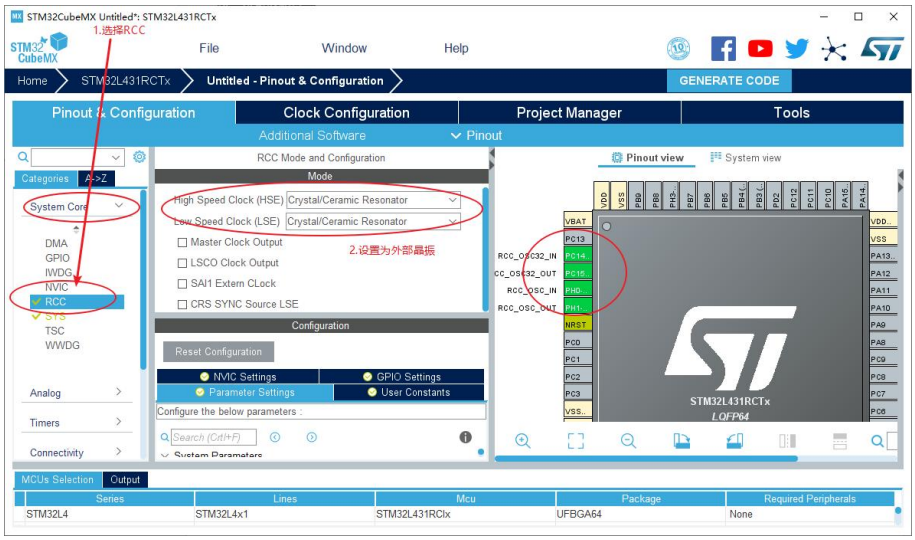
搜索并选中芯片 STM32L431RCT6:



配置时钟源

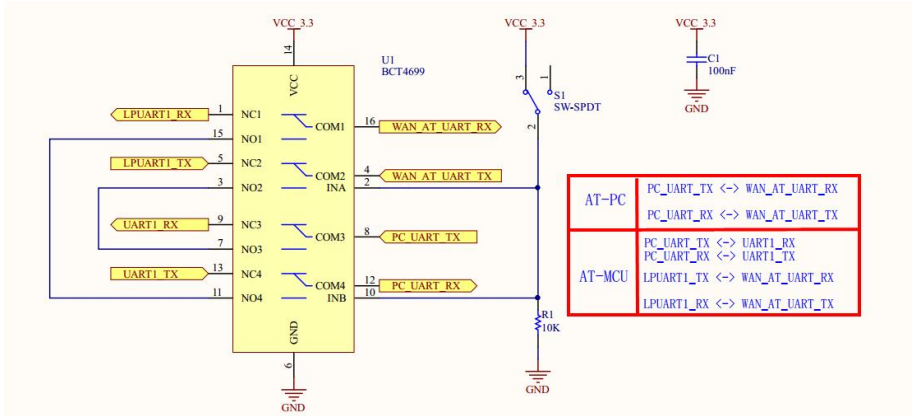
- 如果选择使用外部高速时钟（HSE），则需要在 System Core 中配置 RCC；
- 如果使用默认内部时钟（HSI），这一步可以略过；

这里我都使用外部时钟：



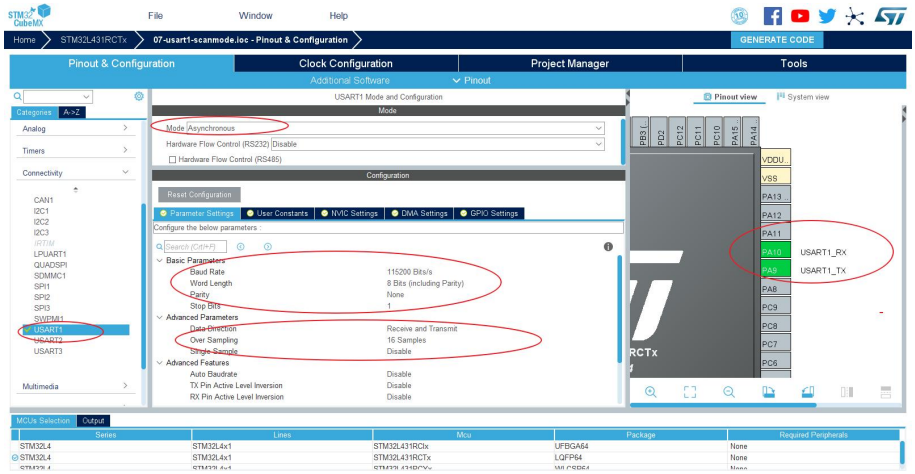
配置串口

小熊派开发板板载 ST-Link 并且虚拟了一个串口，原理图如下：



这里我将开关拨到 AT-MCU 模式，使 PC 的串口与 USART1 之间连接。

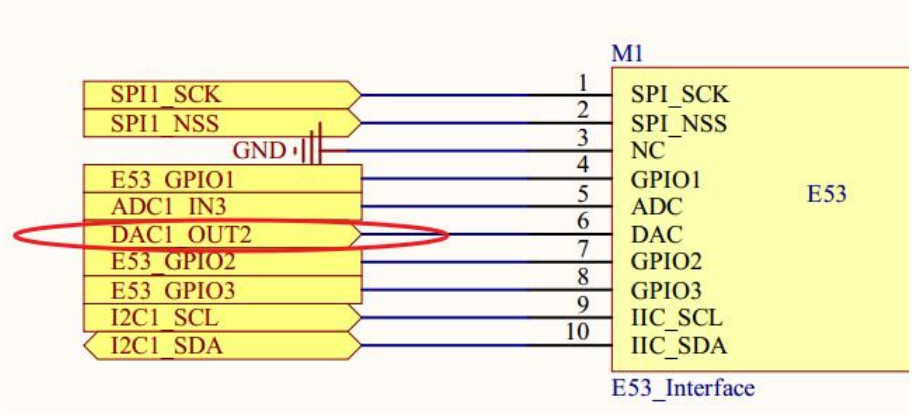
接下来开始配置 USART1：



配置 DAC

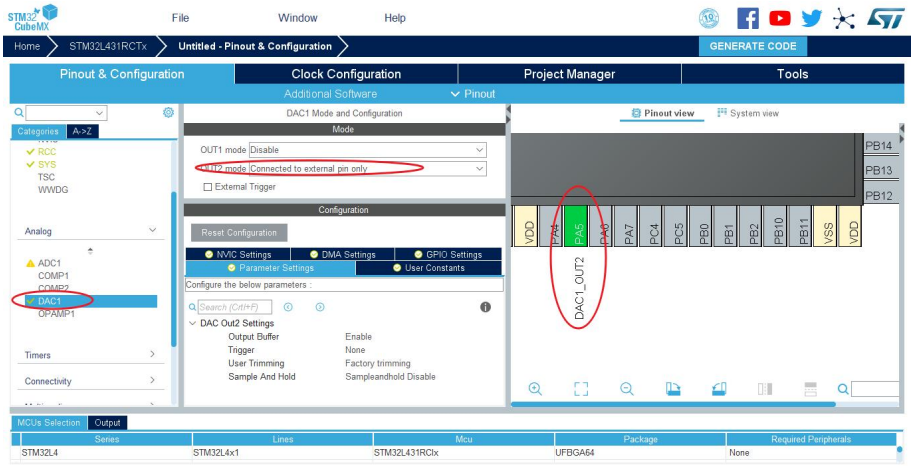
确定 DAC 输出通道

查看小熊派 E53 接口的原理图：



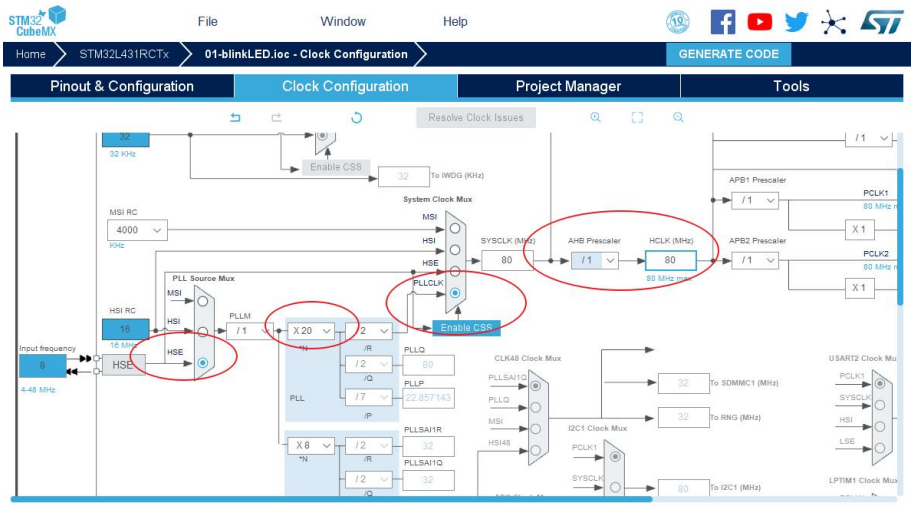
配置 DAC

选择 DAC1，开启输出通道 2，配置保持默认即可：

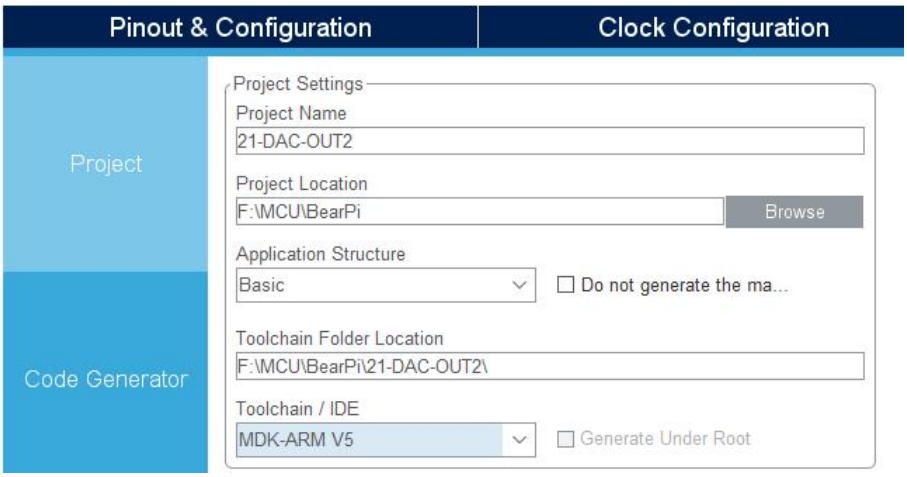


配置时钟树

STM32L4 的最高主频到 80M，所以配置 PLL，最后使 HCLK = 80Mhz 即可：

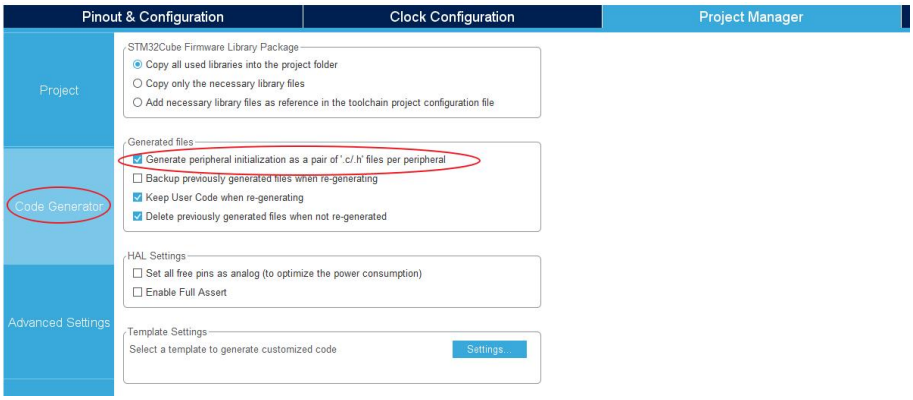


生成工程设置



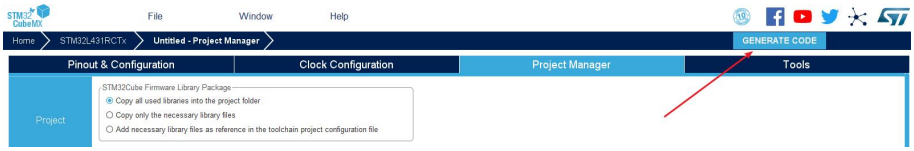
代码生成设置

最后设置生成独立的初始化文件：



生成代码

点击 `GENERATE CODE` 即可生成 MDK-V5 工程：



3. 在 MDK 中编写、编译、下载用户代码

重定向 printf( )函数

参考：重定向 printf 函数到串口输出的多种方法。

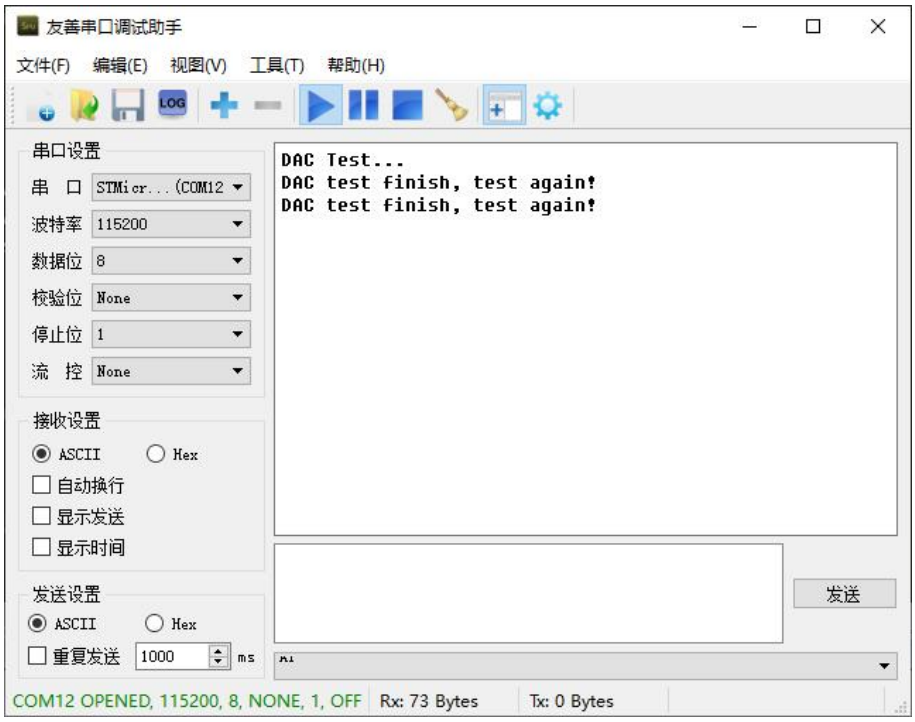
编写测试代码

首先设置 DAC 输出的数据为 **12 位右对齐**，然后指定输出的值 0-4096，实际输出的电压为 `value/4096x3.3V`，最后使能 DAC 转换，代码如下：

```
int main(void){
```

```
    /* USER CODE BEGIN 1 */
    uint16_t i = 0;
    /* USER CODE END 1 */

    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    MX_DAC1_Init();
    MX_USART1_UART_Init();

    /* USER CODE BEGIN 2 */
    printf("DAC Test...\r\n");
    HAL_DAC_Start(&hdac1, DAC_CHANNEL_2);

    /* USER CODE END 2 */

    /* Infinite loop */
    /* USER CODE BEGIN WHILE */
    while (1)
    {
    /* USER CODE END WHILE */
        for(i = 0; i < 4096; i++)
        {
            HAL_DAC_SetValue(&hdac1, DAC_CHANNEL_2, DAC_ALIGN_12B_R, i);
            HAL_Delay(2);
```

```
        }
        printf("DAC test finish, test again!\r\n");

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



至此，我们已经学会**如何使用 DAC 输出任意指定电压值**。

作业：

编写程序，通过 DAC 输出指定电压值，并通过 ADC 进行验证。

分析 DAC 的分辨率和精度对输出电压的影响。

STM32 单片机基础 21——使用 ADC 读取电压值

教学目的与要求：

目的：巩固 ADC 的相关知识，深入理解 ADC 的工作原理，实现模拟信号的精确测量。

要求：能够配置 STM32 的 ADC 模块，编写程序读取外部模拟电压值，并进行适当的处理和分析。

教学重难点：

重点：ADC 模块的配置和电压值的读取处理。

难点：理解 ADC 的采样率和分辨率对测量结果的影响，确保测量的准确性和稳定性。

课时数：3 课时

思政元素：

培养学生的实践能力和数据分析能力，通过 ADC 读取电压值的学习，让学生认识到数据采集和处理在科学实验和工程技术中的重要性，培养学生的实践精神和数据分析能力。

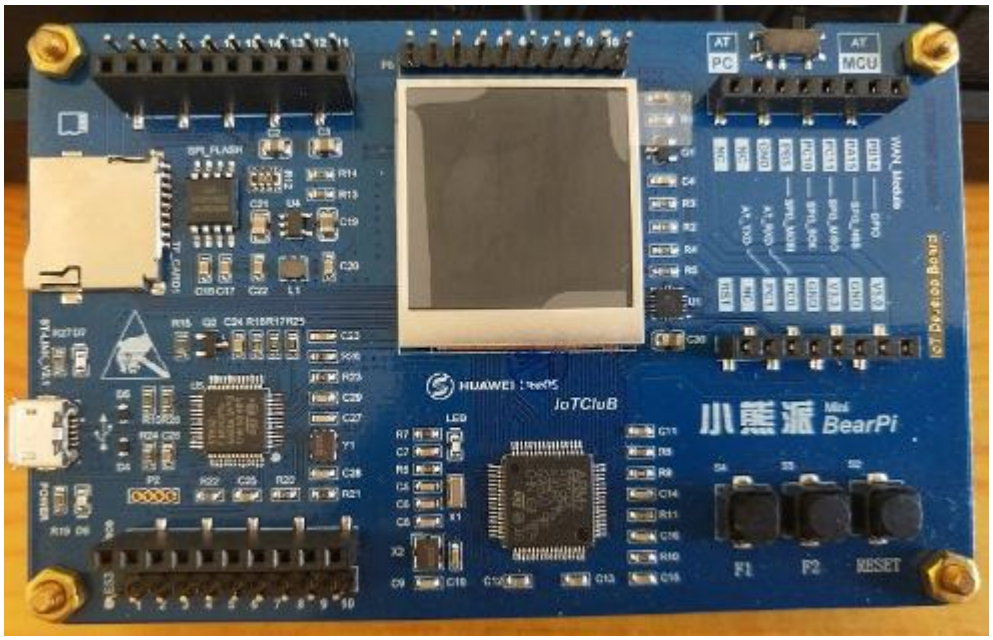
本篇详细的记录了如何使用 STM32CubeMX 配置 STM32L431RCT6 的 ADC 外设，读取 DAC 输出引脚的电压值。

1. 准备工作

硬件准备

开发板

首先需要准备一个开发板，这里我准备的是 STM32L4 的开发板（BearPi）：



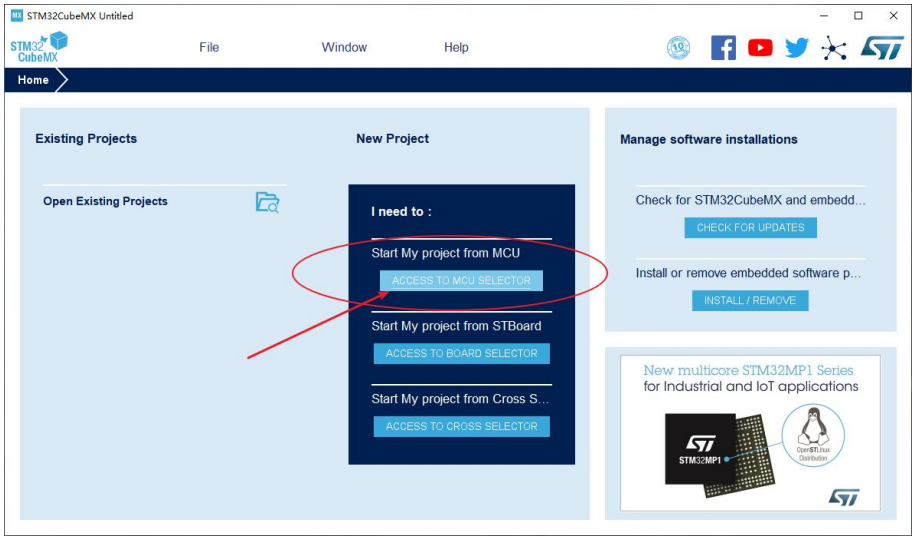
软件准备

- 需要安装好 Keil - MDK 及芯片对应的包，以便编译和下载生成的代码；
  Keil MDK 和串口助手的安装包都可以关注“**小熊派开源社区**”微信公众号，在资料教程一栏中可获取安装包。

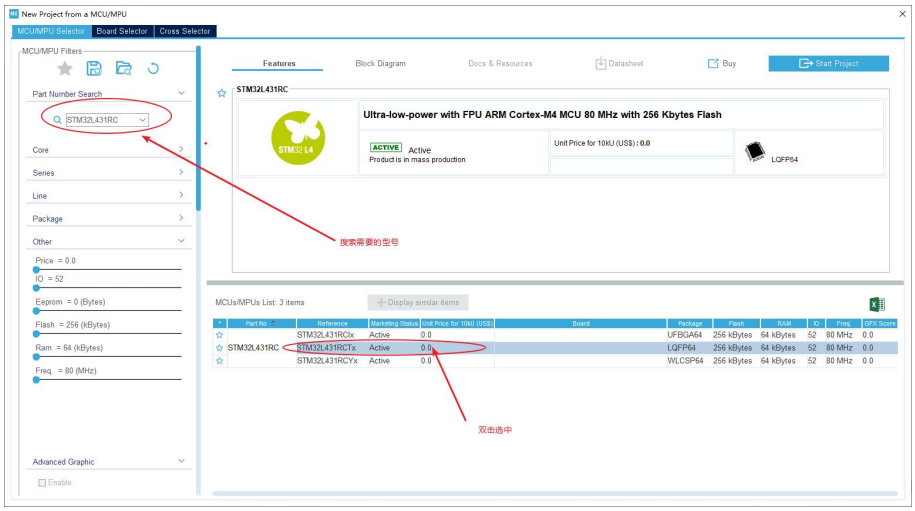
2. 生成 MDK 工程

选择芯片型号

打开 STM32CubeMX，打开 MCU 选择器：



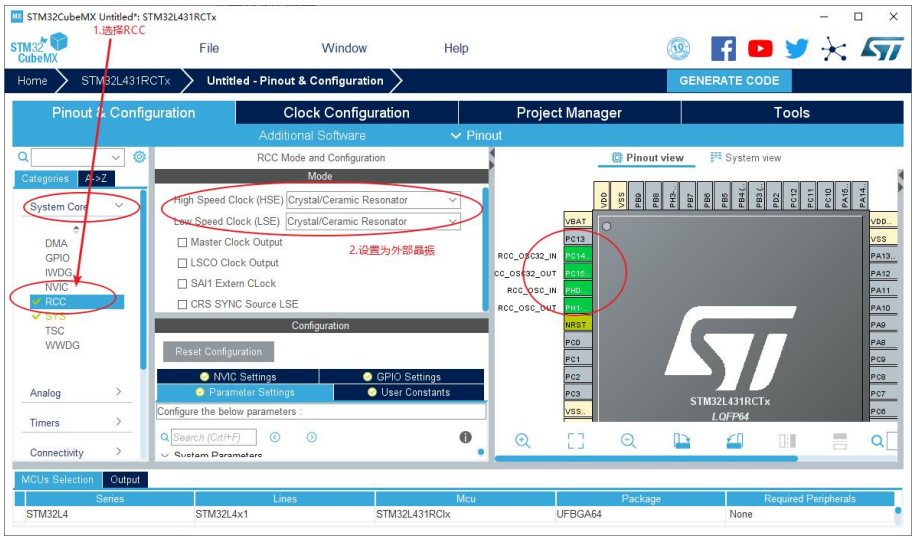
搜索并选中芯片 STM32L431RCT6:



配置时钟源

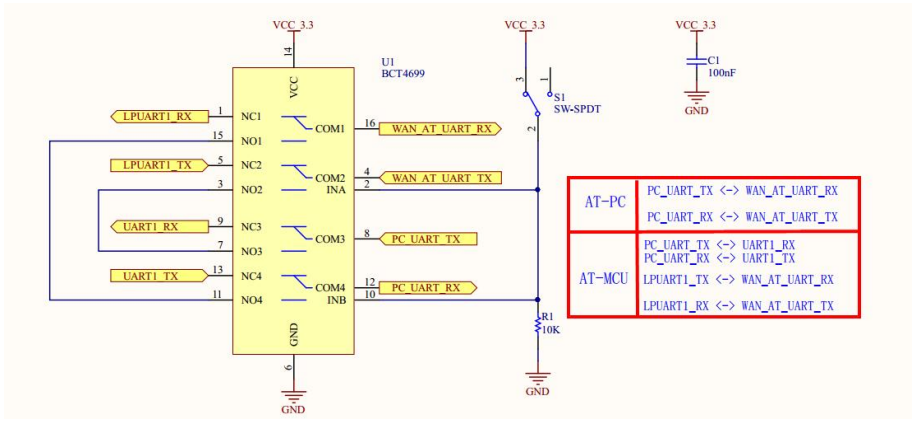
- 如果选择使用外部高速时钟（HSE），则需要在 System Core 中配置 RCC；
- 如果使用默认内部时钟（HSI），这一步可以略过；

这里我都使用外部时钟：



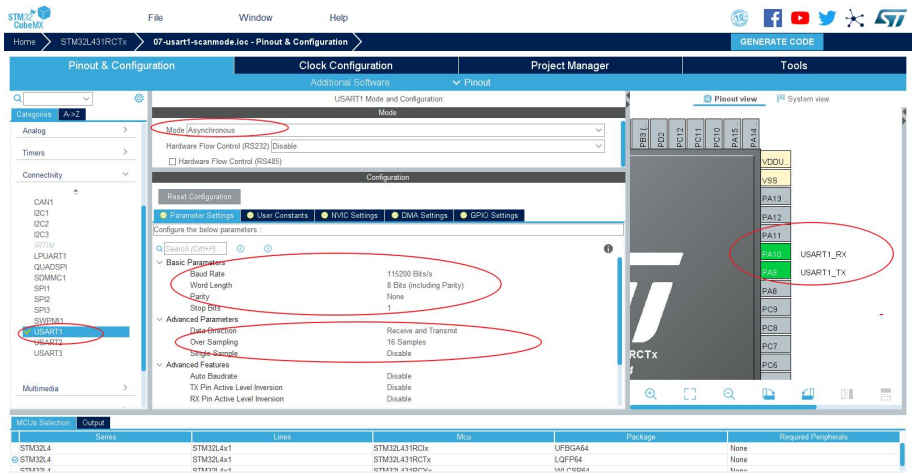
配置串口

小熊派开发板板载 ST-Link 并且虚拟了一个串口，原理图如下：



这里我将开关拨到 AT-MCU 模式，使 PC 的串口与 USART1 之间连接。

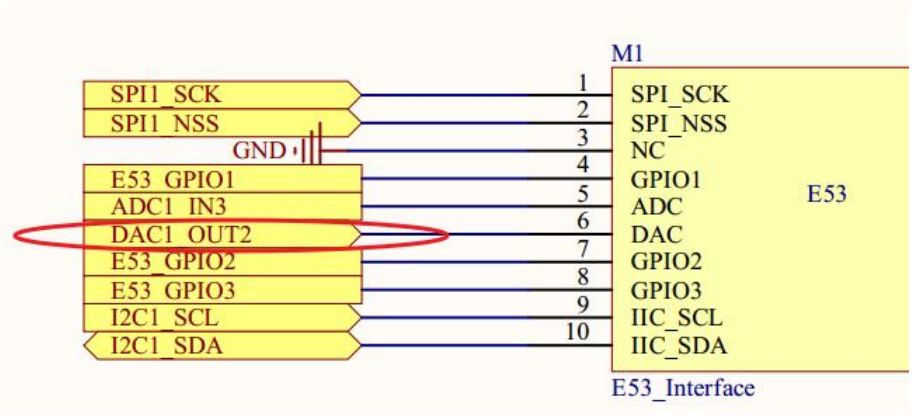
接下来开始配置 USART1：



配置 DAC

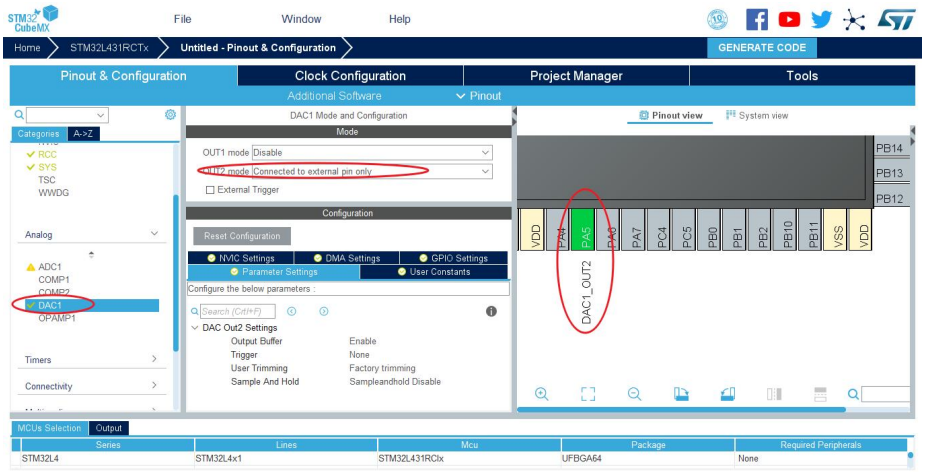
确定 DAC 输出通道

查看小熊派 E53 接口的原理图：



配置 DAC

选择 DAC1，开启输出通道 2，配置保持默认即可：



配置 ADC

知识小卡片 —— ADC

ADC 全称 Analog-to-Digital Converter，即模拟-数字转换器，可以将连续变化的模拟信号转换为离散的数字信号，进而使用数字电路进行处理，称之为数字信号处理。

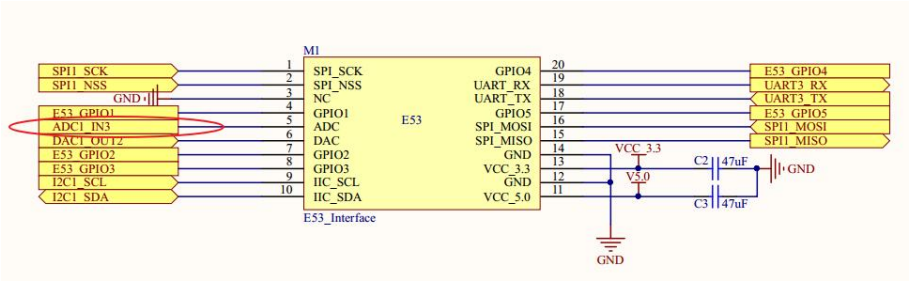
STM32L431xx 系列有 1 个 ADC，ADC 分辨率高达 12 位，每个 ADC 具有多达 20 个的采集通道，这些通道的 A/D 转换可以单次、连续、扫描或间断模式执行。 ADC 的结果可以左对齐或右对齐方式存储在 16 位数据寄存器中。

STM32L431 的 ADC 最大的转换速率为 5.33Mhz，也就是转换时间为 0.188us（12 位分辨率时），ADC 的转换时间与 AHB 总线时钟频率无关。

知识小卡片结束啦~对 ADC 有没有了解呢？

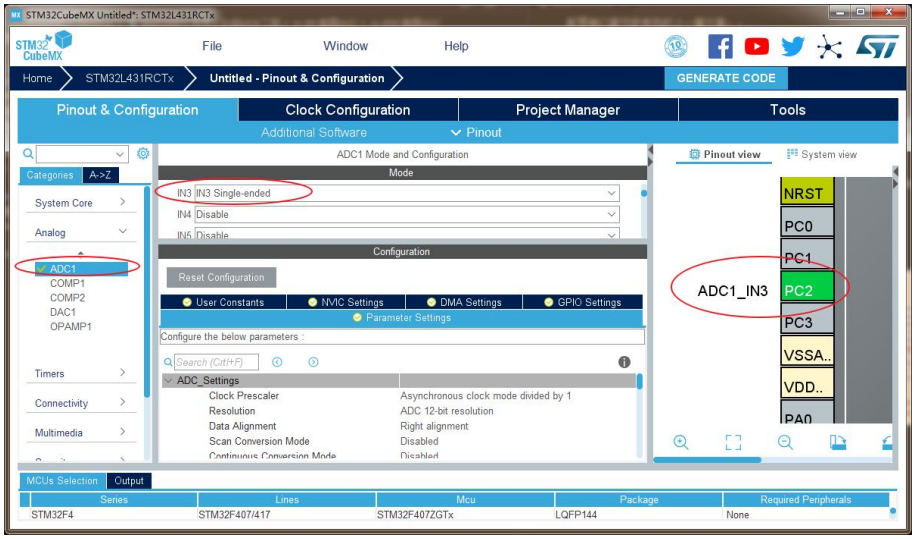
确定 ADC 通道

查看小熊派 E53 接口的原理图：



配置 ADC（单次转换模式）

首先选择 ADC1，开启通道 3：



接下来是对 ADC 的设置，这里我们保持默认即可：

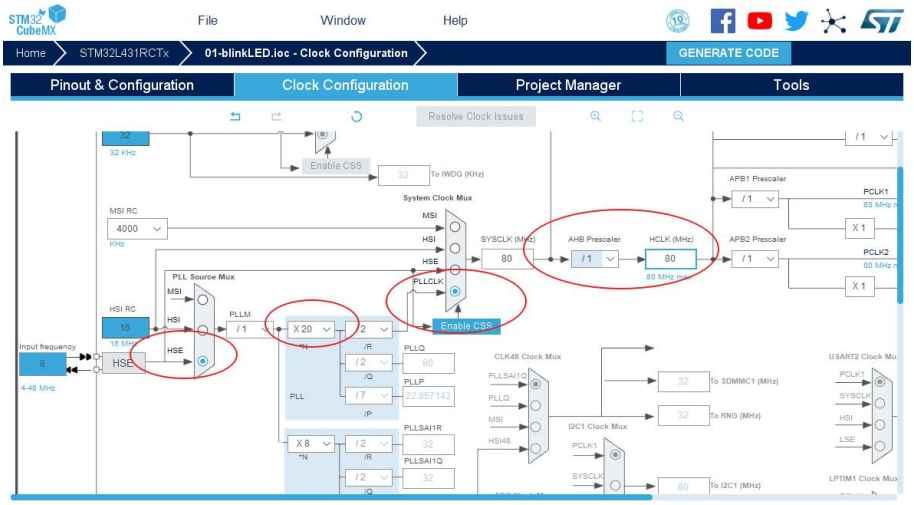
最后设置 ADC 的转换规则：

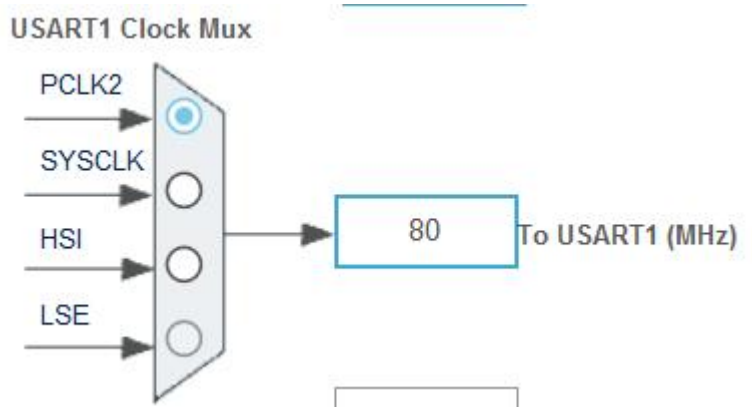
ADC_Regular_ConversionMode	
Enable Regular Conversions	Enable
Enable Regular Oversampling	Disable
Number Of Conversion	1
External Trigger Conversion Source	Regular Conversion launched by software
External Trigger Conversion Edge	None
Rank	1
Channel	Channel 3
Sampling Time	2.5 Cycles
Offset Number	No offset
ADC_Injected_ConversionMode	
Enable Injected Conversions	Disable

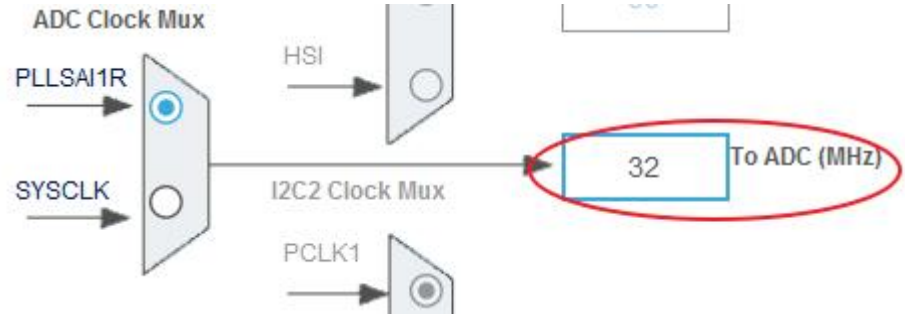
其余的一些设置保持默认即可。

配置时钟树

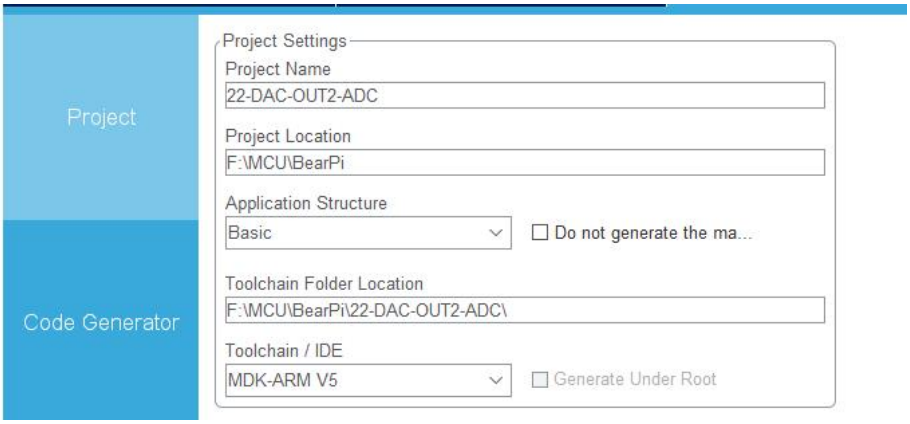
STM32L4 的最高主频到 80M，所以配置 PLL，最后使 `HCLK = 80Mhz` 即可：





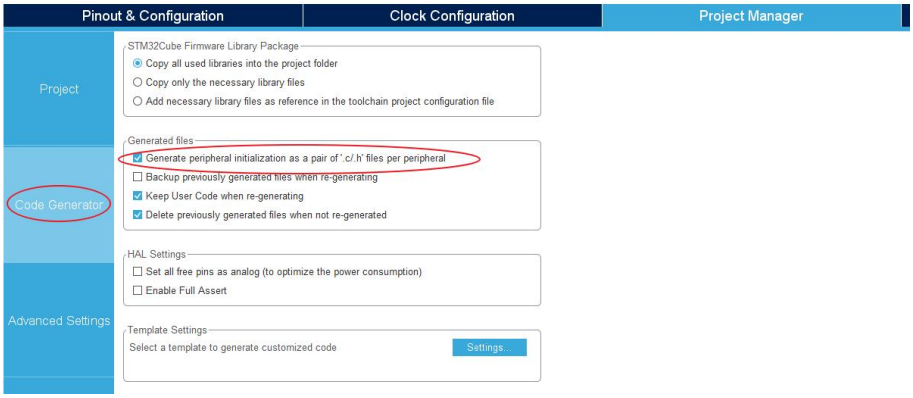


生成工程设置



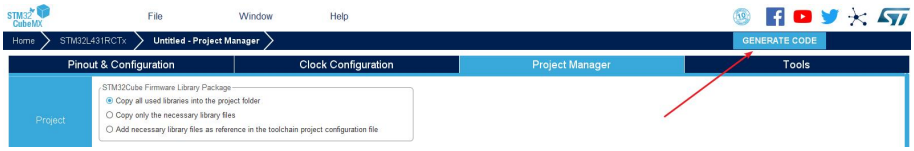
代码生成设置

最后设置生成独立的初始化文件：



生成代码

点击 GENERATE CODE 即可生成 MDK-V5 工程：



3. 在 MDK 中编写、编译、下载用户代码

重定向 printf( )函数

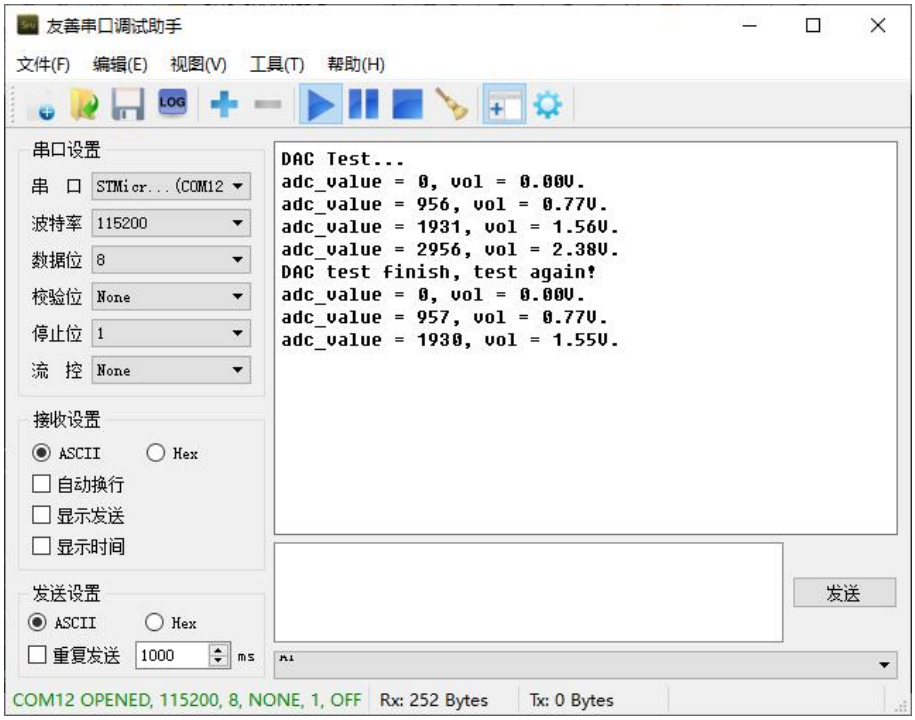
参考：重定向 printf 函数到串口输出的多种方法。

编写读取数据的测试代码

修改 `main` 函数如下：

```
int main(void){
    /* USER CODE BEGIN 1 */
    uint16_t i = 0;
    uint16_t adc_value = 0;
    float vol = 0.0;
    /* USER CODE END 1 */

    HAL_Init();
    SystemClock_Config();

    MX_GPIO_Init();
    MX_DAC1_Init();
    MX_USART1_UART_Init();
    MX_ADC1_Init();

    /* USER CODE BEGIN 2 */
    printf("DAC Test...\r\n");
    HAL_DAC_Start(&hdac1, DAC_CHANNEL_2);
    /* USER CODE END 2 */

    /* Infinite loop */
    /* USER CODE BEGIN WHILE */
```

```
    while (1)
    {
    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
    for(i = 0; i < 4096; i++)
    {
        HAL_DAC_SetValue(&hdac1, DAC_CHANNEL_2, DAC_ALIGN_12B_R, i);
        HAL_Delay(2);
        if(i%1024 == 0)
        {
            /* 使用 ADC 采样 */
            HAL_ADC_Start(&hadc1);                          //启动 ADC 单次转换
            HAL_ADC_PollForConversion(&hadc1, 50); //等待 ADC 转换完成
            adc_value = HAL_ADC_GetValue(&hadc1);  //读取 ADC 转换数据
            vol = ((double)adc_value/4096)*3.3;
            printf("adc_value = %d, vol = %.2fV.\n", adc_value, vol);
        }
    }

    printf("DAC test finish, test again!\r\n");
    }
    /* USER CODE END 3 */
    }
```



至此，我们已经学会**如何使用 ADC 读取 DAC 输出引脚的电压值**。

作业

编写程序，通过 ADC 读取外部模拟电压值，并进行处理和分析。

分析 ADC 的采样率和精度对测量结果的影响