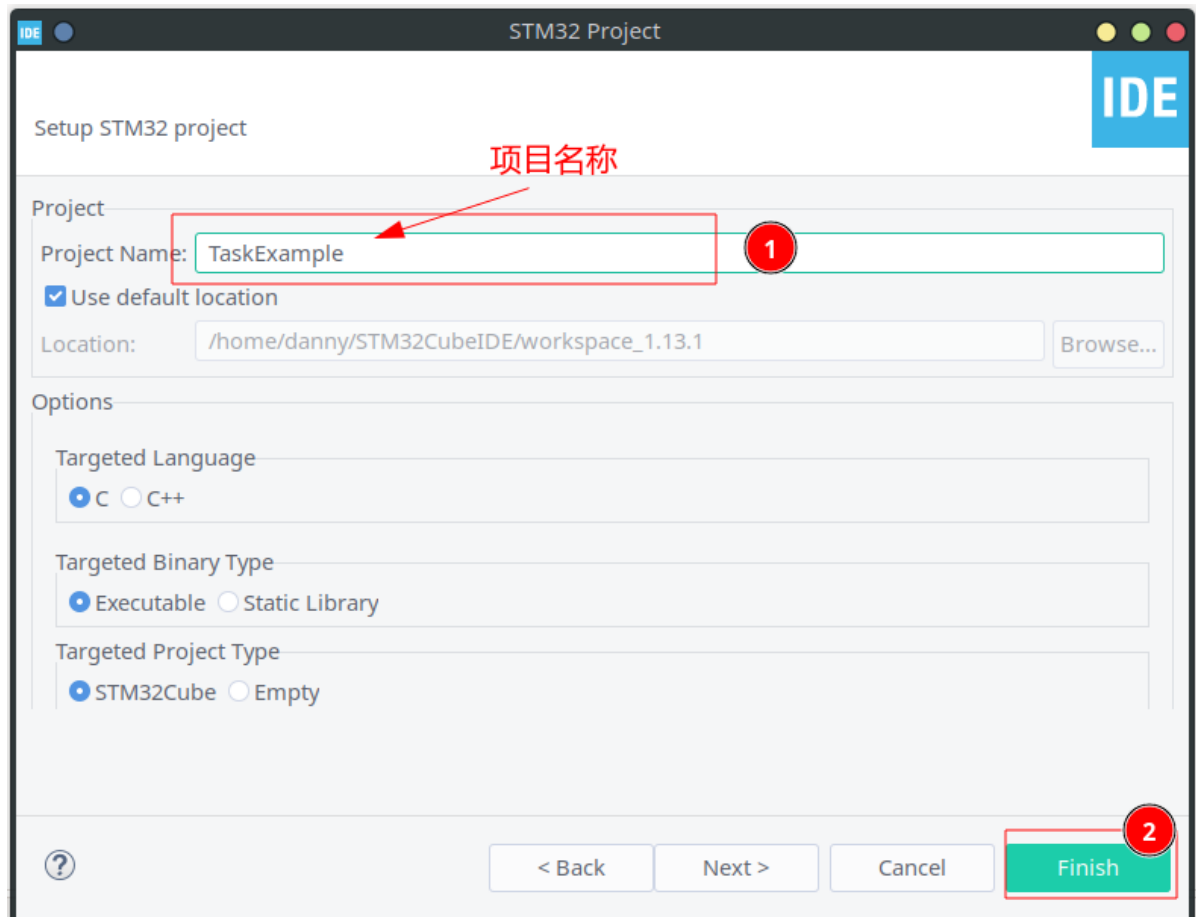
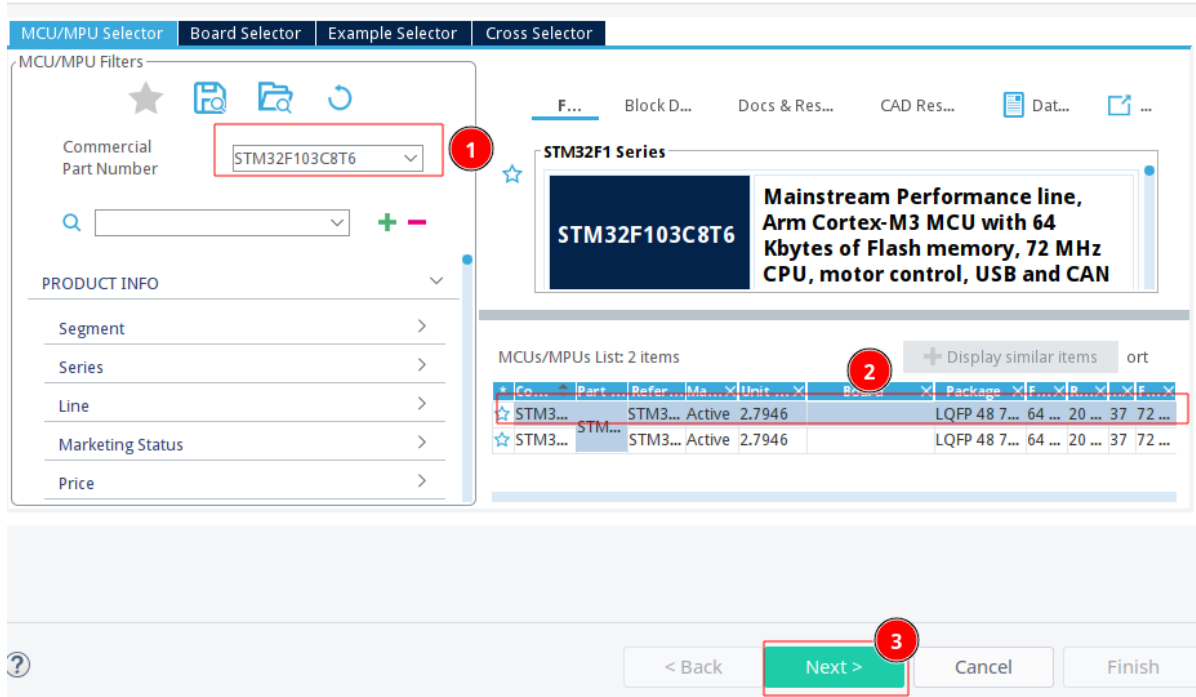


简单的例子

1. 新建带FreeRTOS的项目

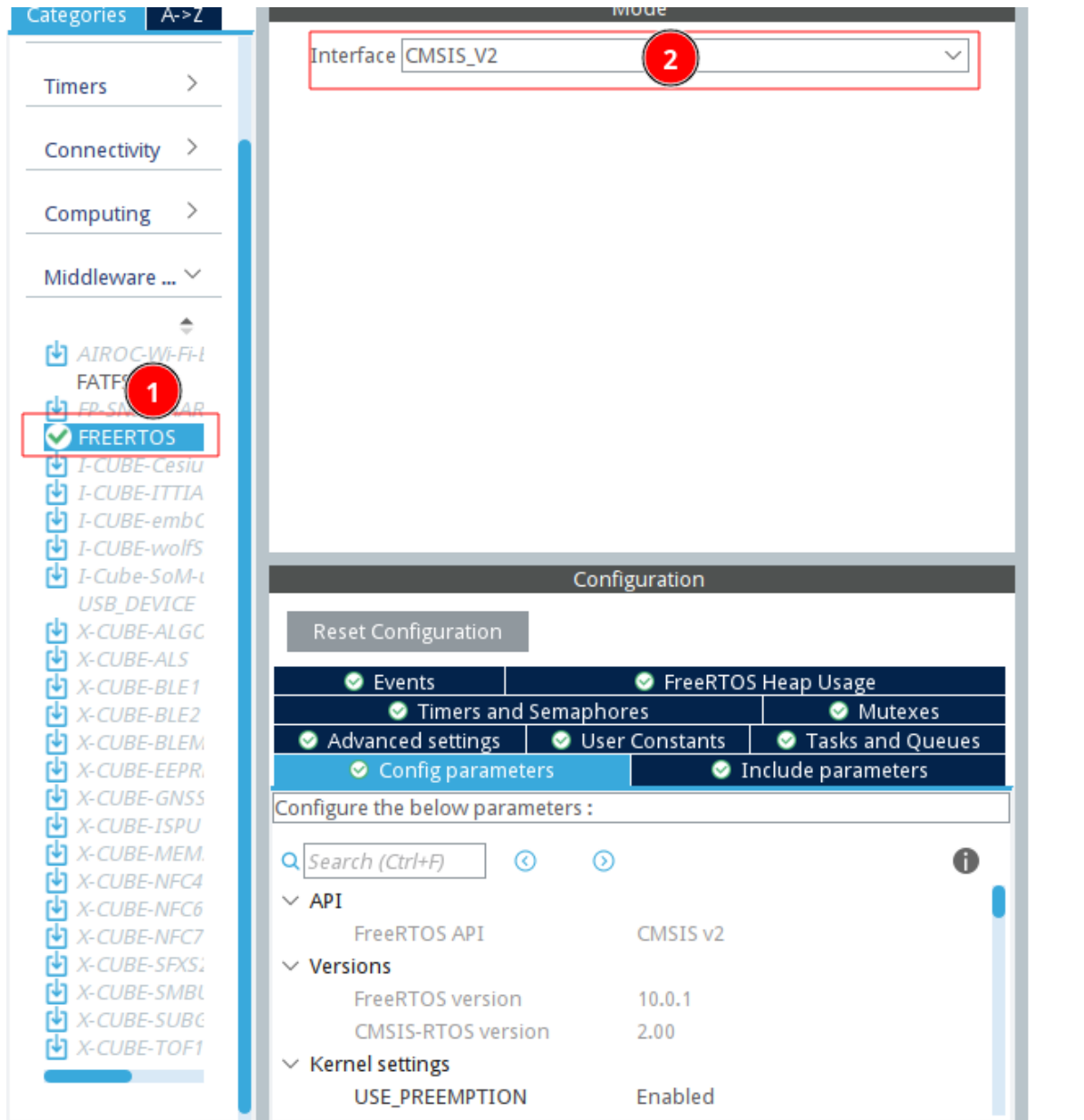
1.1. 建立项目

File->New->STM32 Project

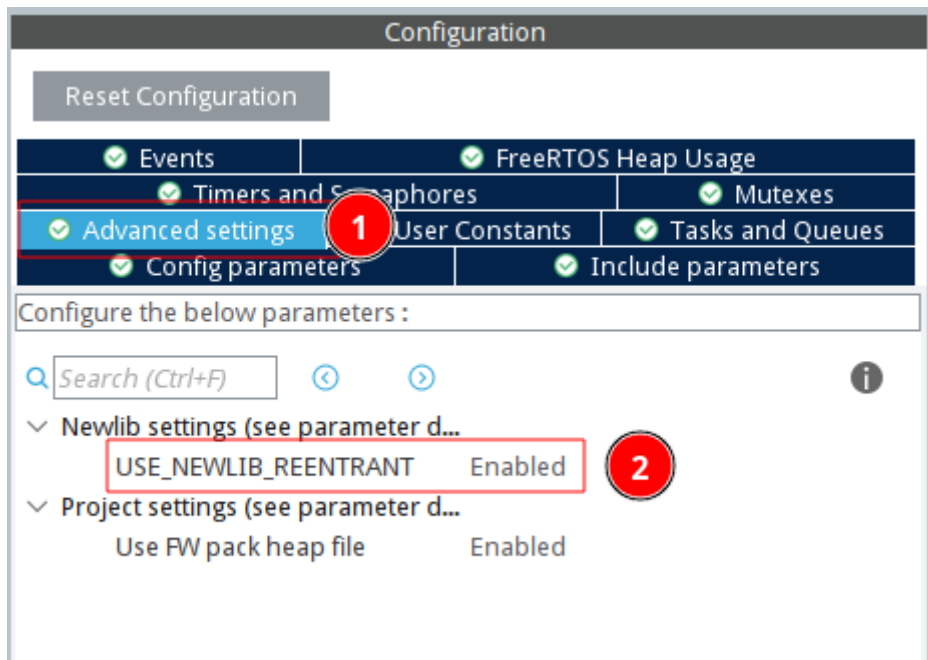


1.2. 配置项目

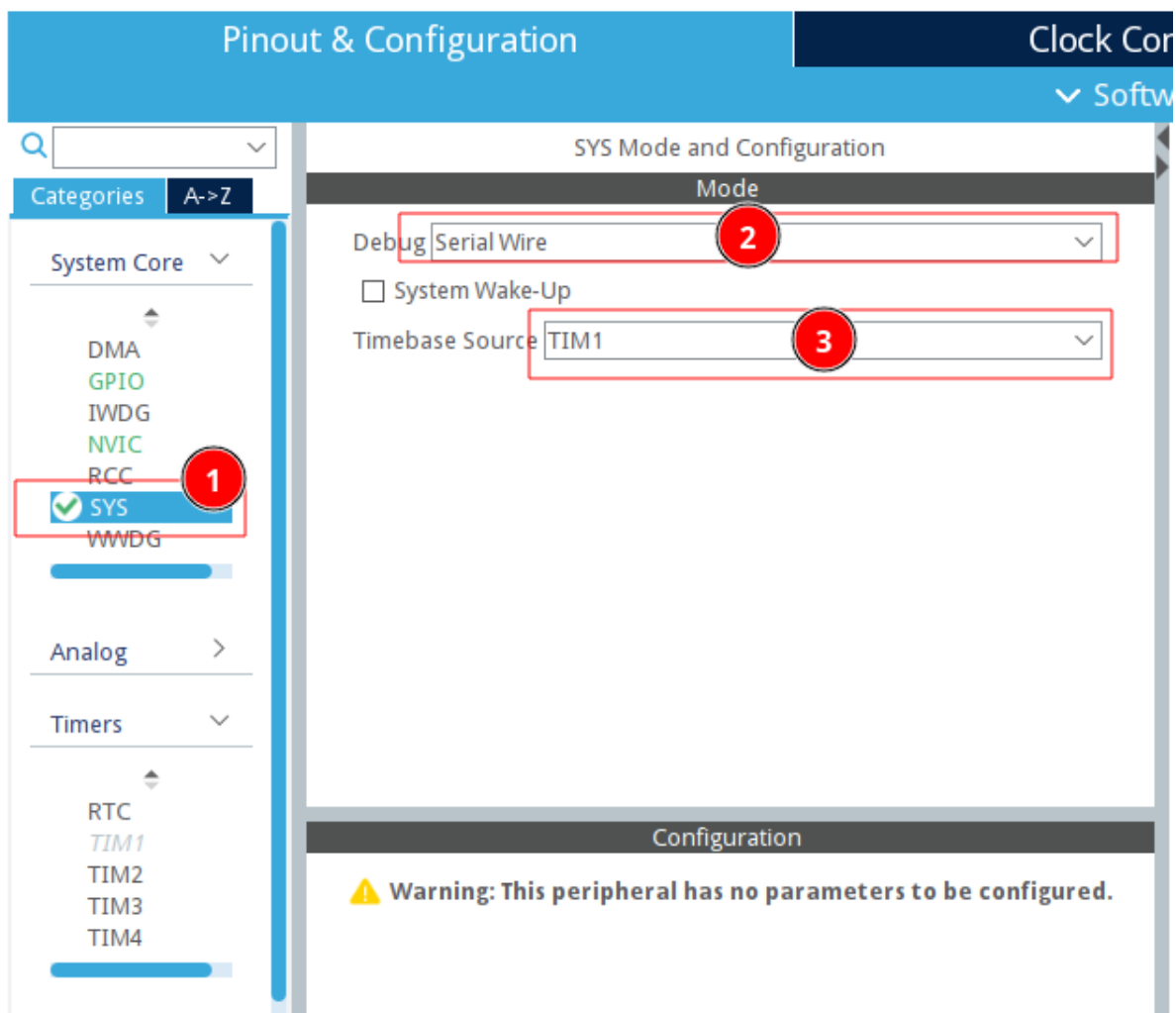
首先启用FreeRTOS支持，使用CMSIS_V2接口。



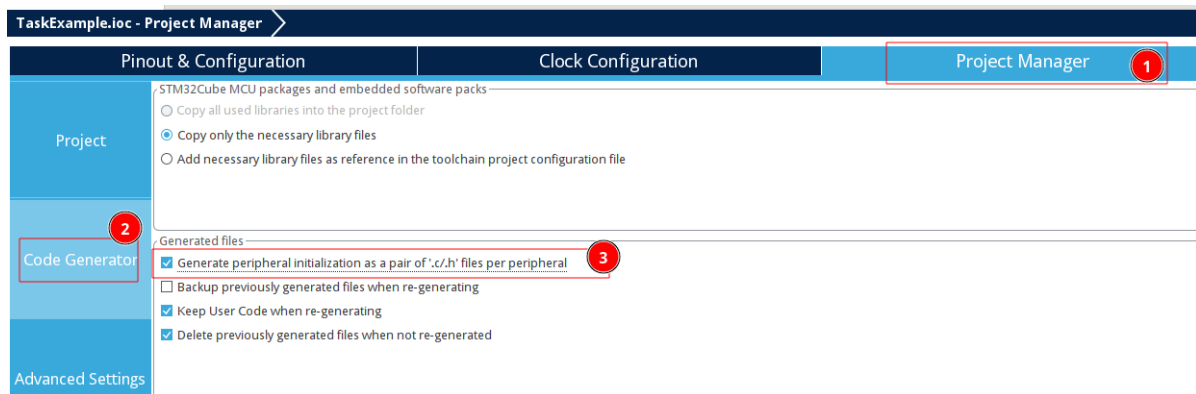
在Advanced settings 中打开 NEWLIB 的支持。



设置系统滴答的时钟源以及调试方式。



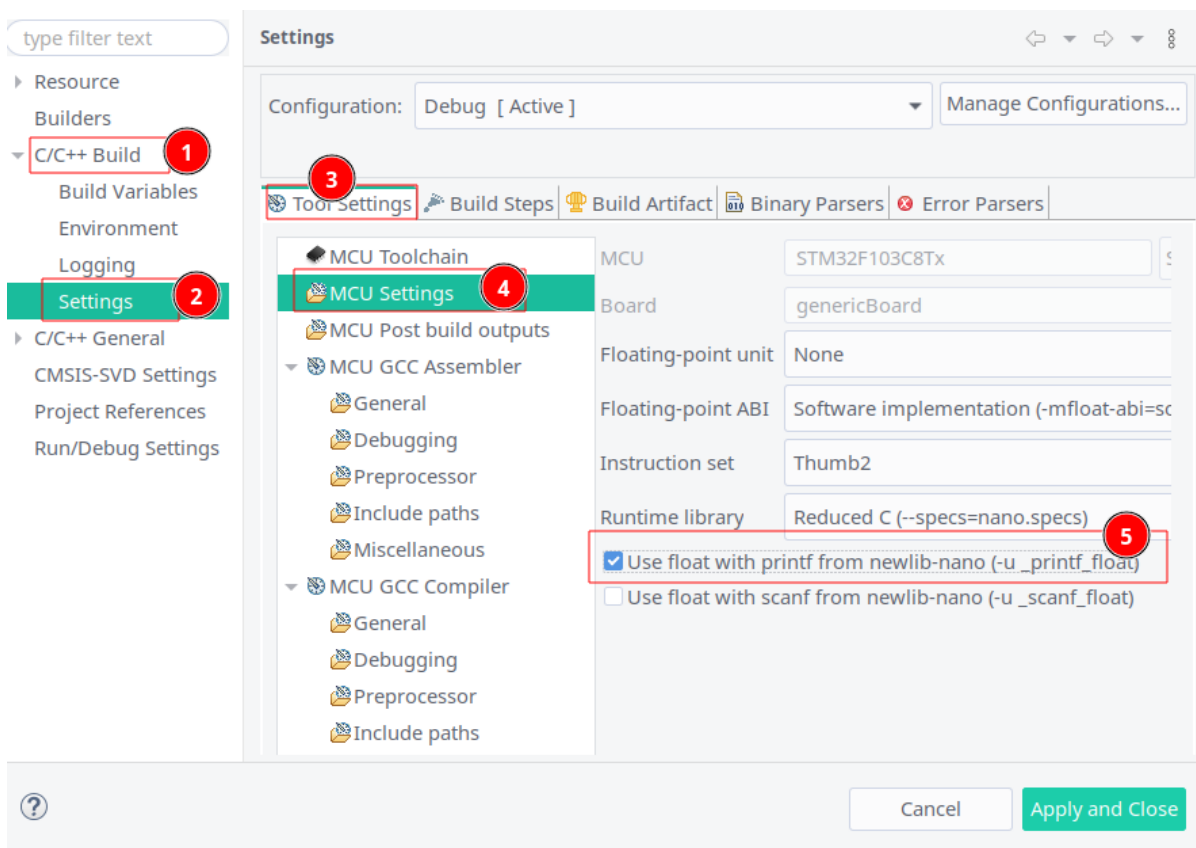
因为代码量比较多，最好为每个外设设置独立的文件。



保存后生成代码框架

最后设置使用printf可以使用浮点。

Project->Properties



2. 事件标志组的使用

请参考书上333页的讲解。

这里使用两个按钮都同时按过后才能点亮灯作为例子；如果只按了一个按钮不会点亮灯；两个按钮的先后顺序没关系；也就是说这是一个AND关系。

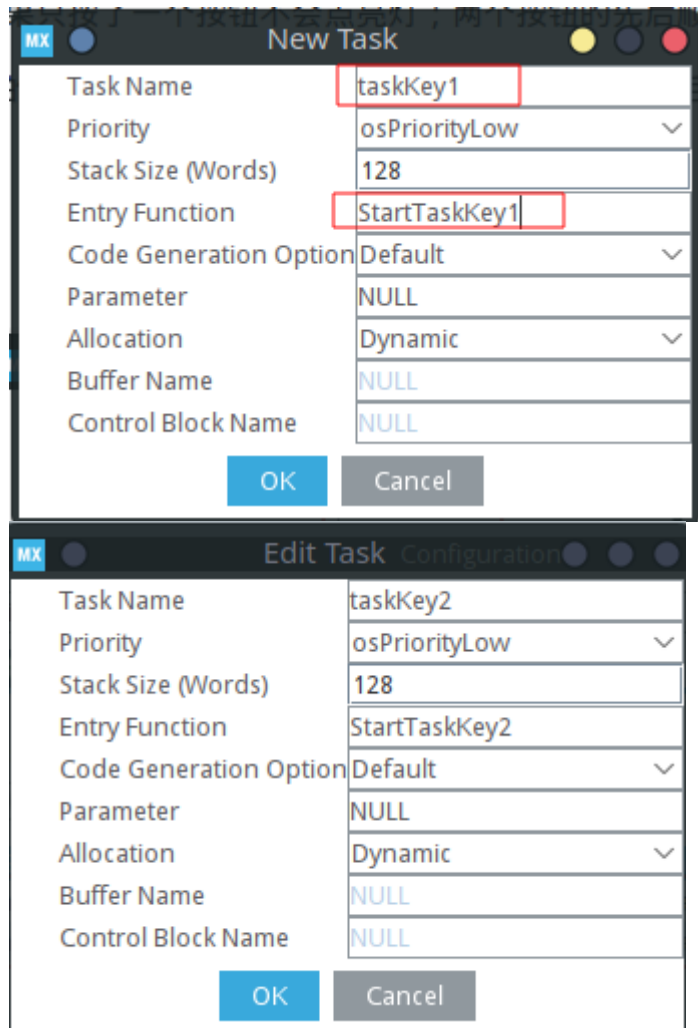
为了简单，我们并不做太多的操作，只是在一个按钮按下时候设置信号而已。当点灯的任务执行完点灯后，清除标志。

2.1. 环境配置

这里的配置都在IDE中配置完成，并不涉及代码

2.1.1. 按键线程

建立两个线程，定时检查按钮。



2.1.2. 按键配置

设置按键方式为输入，设置label为 KEY1 和 KEY2。需要在GPIO中设置两个端口的上拉（其中有个端口有外部上拉，我记不住了，就两个都设置了，也不影响）。

Configuration

Group By Peripherals

GPIO SYS

Search Signals

Search (Ctrl+F) Show only Modified Pins

| Pin Name | Signal on Pin | GPIO output... | GPIO mode | GPIO Pull-u... | Maximu... | User Label | Modified |
|----------|---------------|----------------|------------|----------------|-----------|------------|-------------------------------------|
| PB12 | n/a | n/a | Input mode | Pull-up | n/a | KEY1 | <input checked="" type="checkbox"/> |
| PB13 | n/a | n/a | Input mode | Pull-up | n/a | KEY2 | <input checked="" type="checkbox"/> |

PB13 Configuration :

GPIO mode: Input mode

GPIO Pull-up/Pull-down: Pull-up

User Label: KEY2

2.1.3. LED灯配置

使用红色灯PB0端口设置成输出，并设置label为LED。

GPIO SYS

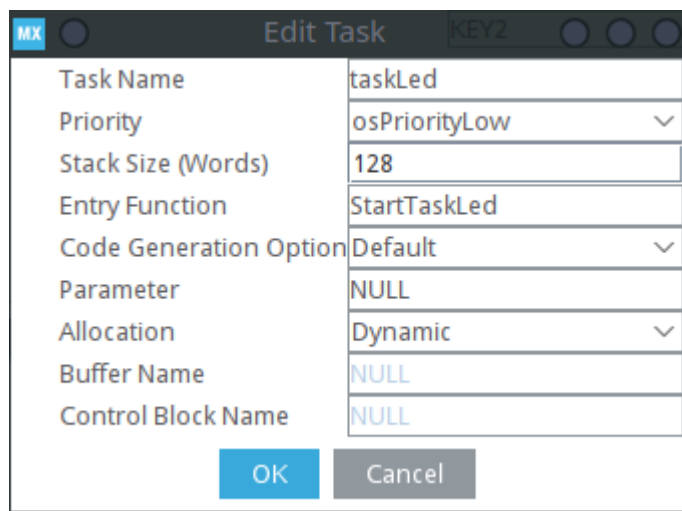
Search Signals

Search (Ctrl+F) Show only Modified Pins

| Pin Name | Signal on Pin | GPIO outp... | GPIO mode | GPIO Pull-... | Maximum ... | User Label | Modified |
|----------|---------------|--------------|--------------|----------------|-------------|------------|-------------------------------------|
| PB0 | n/a | Low | Output Pu... | No pull-up ... | Low | LED | <input checked="" type="checkbox"/> |
| PB12 | n/a | n/a | Input mode | Pull-up | n/a | KEY1 | <input checked="" type="checkbox"/> |
| PB13 | n/a | n/a | Input mode | Pull-up | n/a | KEY2 | <input checked="" type="checkbox"/> |

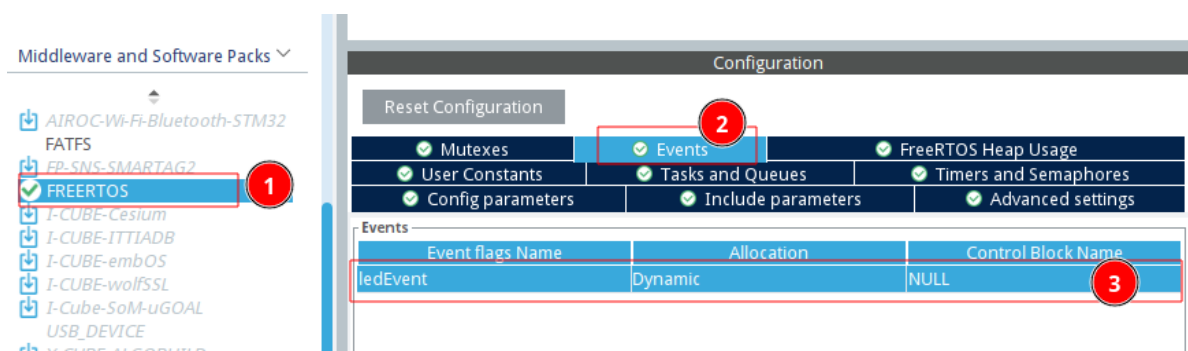
2.1.4. LED控制线程

建立一个LED控制线程



2.1.5. 事件标志组

书上是通过代码的方式建立事件标志组的，代码有些问题。这里直接使用配置的方式建立事件标志组。



2.2. 代码解析

IDE已经为我们生成了代码框架，接下来看看主要的代码块在什么位置。

2.2.1. 代码框架

GPIO的端口常量定义在main.h 的大约 60 行：

```
#define LED_Pin GPIO_PIN_0
#define LED_GPIO_Port GPIOB
#define KEY1_Pin GPIO_PIN_12
#define KEY1_GPIO_Port GPIOB
#define KEY2_Pin GPIO_PIN_13
#define KEY2_GPIO_Port GPIOB
```

这里定义了我们的三个IO的port和pin，到时候需要使用。

在 freertos.c 的大约67行定义了事件标志组的变量：

```
/* Definitions for ledEvent */
osEventFlagsId_t ledEventHandle;
const osEventFlagsAttr_t ledEvent_attributes = { .name = "ledEvent" };
```

在大约87行的MX_FREERTOS_Init 初始化函数中对线程以及事件标志进行初始化：

```
void MX_FREERTOS_Init(void) {
    /* USER CODE BEGIN Init */
```

```

/* USER CODE END Init */

/* USER CODE BEGIN RTOS_MUTEX */
/* add mutexes, ... */
/* USER CODE END RTOS_MUTEX */

/* USER CODE BEGIN RTOS_SEMAPHORES */
/* add semaphores, ... */
/* USER CODE END RTOS_SEMAPHORES */

/* USER CODE BEGIN RTOS_TIMERS */
/* start timers, add new ones, ... */
/* USER CODE END RTOS_TIMERS */

/* USER CODE BEGIN RTOS_QUEUES */
/* add queues, ... */
/* USER CODE END RTOS_QUEUES */

/* Create the thread(s) */
/* creation of defaultTask */
defaultTaskHandle = osThreadNew(StartDefaultTask, NULL,
                                &defaultTask_attributes);

/* creation of taskKey1 */
taskKey1Handle = osThreadNew(StartTaskKey1, NULL, &taskKey1_attributes);

/* creation of taskKey2 */
taskKey2Handle = osThreadNew(StartTaskKey2, NULL, &taskKey2_attributes);

/* creation of taskLed */
taskLedHandle = osThreadNew(StartTaskLed, NULL, &taskLed_attributes);

/* USER CODE BEGIN RTOS_THREADS */
/* add threads, ... */
/* USER CODE END RTOS_THREADS */

/* Create the event(s) */
/* creation of ledEvent */
ledEventHandle = osEventFlagsNew(&ledEvent_attributes);

/* USER CODE BEGIN RTOS_EVENTS */
/* add events, ... */
/* USER CODE END RTOS_EVENTS */

}

```

在freertos.c中有三个自定义的线程，可以在IDE的Outline中比较方便的看到：

- StartTaskKey1(void*): void
- StartTaskKey2(void*): void
- StartTaskLed(void*): void

这三个函数就是具体的线程任务，我们看看其中的一个，其他三个都是类似的。

```
/* USER CODE BEGIN Header_StartTaskKey1 */
/**
 * @brief Function implementing the taskKey1 thread.
 * @param argument: Not used
 * @retval None
 */
/* USER CODE END Header_StartTaskKey1 */
void StartTaskKey1(void *argument) {
    /* USER CODE BEGIN StartTaskKey1 */
    /* Infinite loop */
    for (;;) {
        osDelay(1);
    }
    /* USER CODE END StartTaskKey1 */
}
```

2.2.2. 添加代码

我们需要在两个按键的线程中读取按键的状态，当按钮被按下后，设置相应的事件标志位；在后在LED线程中等待相应的标志位，等待选项设置成 `osFlagsWaitAll`，表示当所有的事件被设置后才相应。

实验中采用第0位和第1位作为事件标志。

两个按键线程的代码分别是：

```
/* USER CODE BEGIN Header_StartTaskKey1 */
/**
 * @brief Function implementing the taskKey1 thread.
 * @param argument: Not used
 * @retval None
 */
/* USER CODE END Header_StartTaskKey1 */
void StartTaskKey1(void *argument) {
    /* USER CODE BEGIN StartTaskKey1 */
    /* Infinite loop */
    for (;;) {
        osDelay(10);
        if (!HAL_GPIO_ReadPin(KEY1_GPIO_Port, KEY1_Pin))
            osEventFlagsSet(ledEventHandle, 0x01);
    }
    /* USER CODE END StartTaskKey1 */
}
```

```
/* USER CODE BEGIN Header_StartTaskKey2 */
/**
 * @brief Function implementing the taskKey2 thread.
 * @param argument: Not used
 * @retval None
 */
/* USER CODE END Header_StartTaskKey2 */
void StartTaskKey2(void *argument) {
```

```

/* USER CODE BEGIN StartTaskKey2 */
/* Infinite loop */
for (;;) {
    osDelay(10);
    if (!HAL_GPIO_ReadPin(KEY2_GPIO_Port, KEY2_Pin))
        osEventFlagsSet(ledEventHandle, 0x02);
}
/* USER CODE END StartTaskKey2 */
}

```

使用一个死循环去循环检测按钮，每次检测的间隔是10个系统滴答（10ms，因为系统滴答的频率是1000）。注意，这里必须使用osDelay函数！

如果检测到按键被按下，设置事件标志组的标志。按键1设置第0位，因此是0x01；按键2设置第1位，因此是0x02。

LED线程等待标志位：

```

/* USER CODE BEGIN Header_StartTaskLed */
/**
 * @brief Function implementing the taskLed thread.
 * @param argument: Not used
 * @retval None
 */
/* USER CODE END Header_StartTaskLed */
void StartTaskLed(void *argument) {
    /* USER CODE BEGIN StartTaskLed */
    /* Infinite loop */
    for (;;) {
        osEventFlagsWait(ledEventHandle, 0x03, osFlagsWaitAll, osWaitForever);
        HAL_GPIO_TogglePin(LED_GPIO_Port, LED_Pin);
    }
    /* USER CODE END StartTaskLed */
}

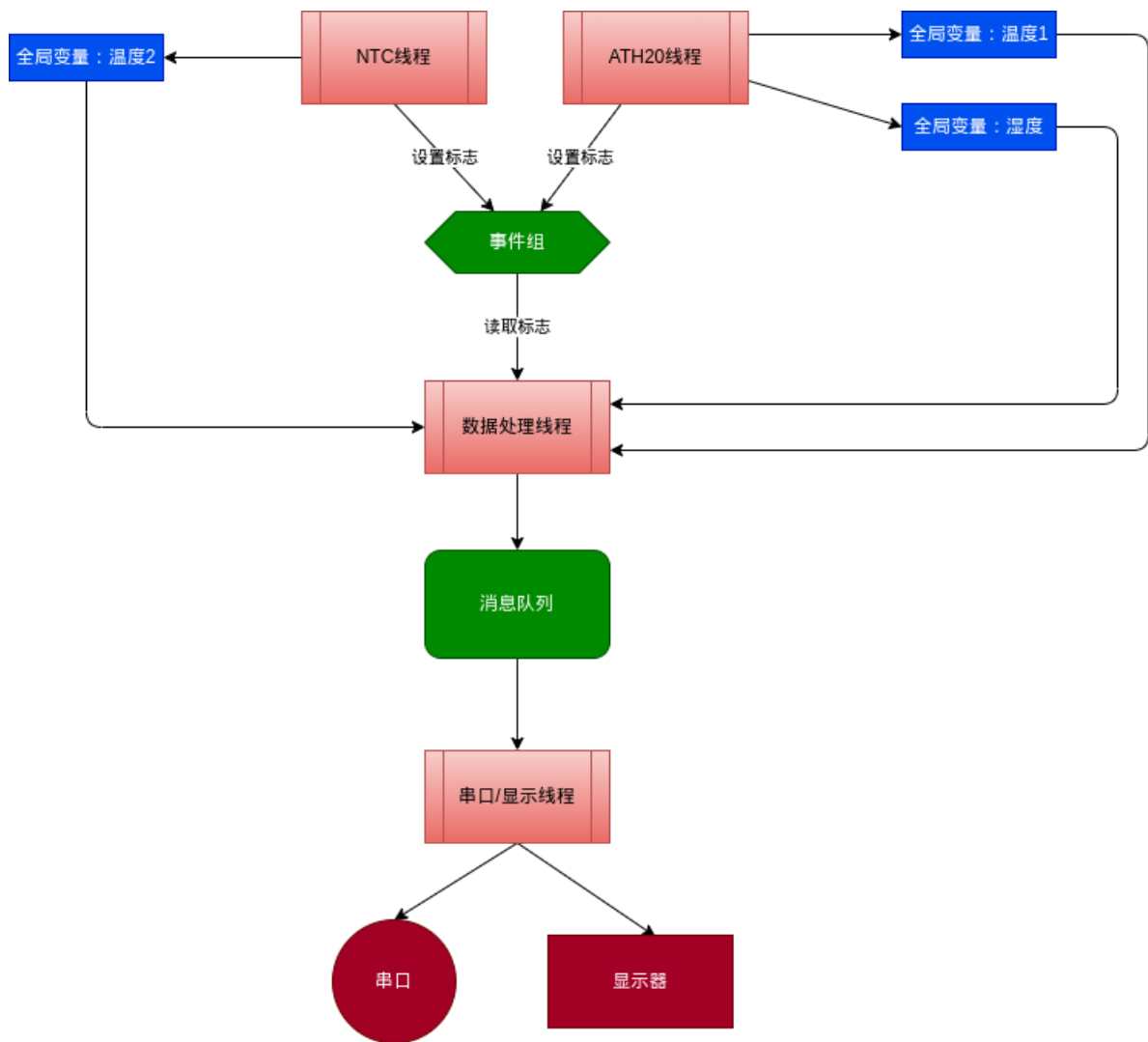
```

因为第0位和第1位都设置后的位掩码对应的数字是0x03，这样当检测到两个标志都设置后才对LED进行翻转。如果osEventFlagsWait等待到事件满足条件后，会清空相应的事件标志。

2.2.3. 实验效果

单按一个按钮LED并不会进行状态转换，按一个后（可以释放），再按第二个后，LED的状态翻转。

实验3设计



注意以下几点：

1. 从最小的代码量一步步完善。如果一开始就是很多代码，那么一旦错误很难定位是哪里出现问题；从最少的代码逐步完善，那么可以基本保证前面代码的正确性。
2. 当遇到问题的时候，需要仔细看错误提示；善用跟踪调试工具，并在过程中结合自己的知识，网上的资料和合理的猜想。
3. 一条路走不通，换条路走，代码的实现方式有很多种。