HAL

lab1-HAL

new project

板子类型：STM32F103RC 第一个

语言：c或C++

RCC:

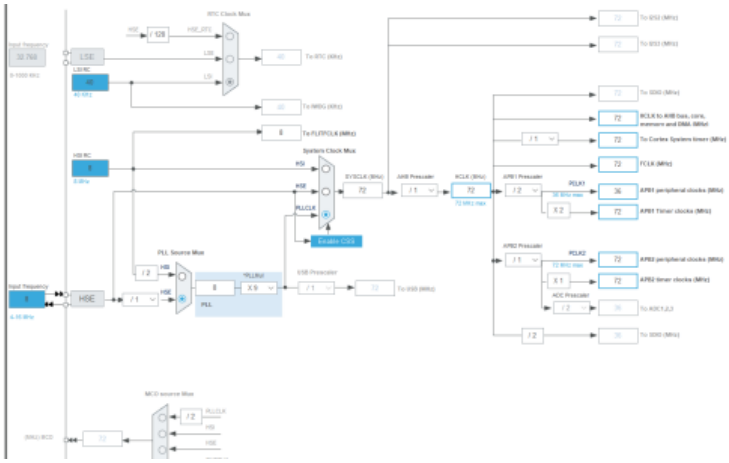
- HSE: Crystal/Ceramic Resonator
- LSE: Disable

SYS:

- Debug: Serial Wire

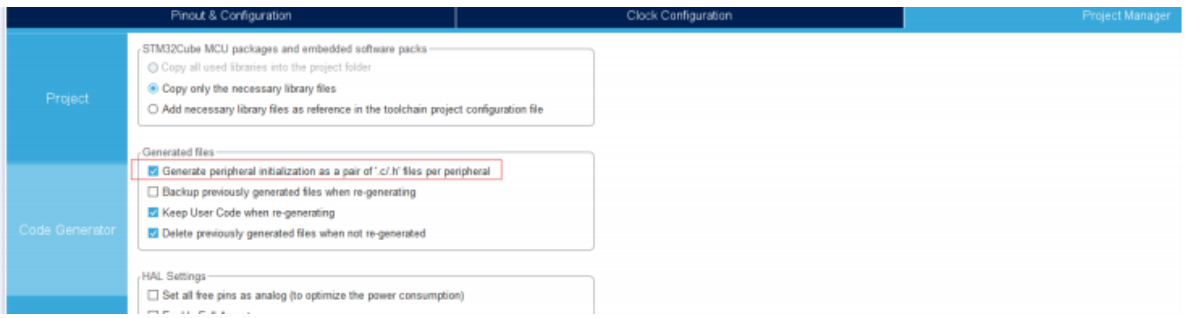
Clock Configuration:

- HCLK: 72MHz



这张图只改HCLK，其他会自动改好。

Project Manager:



USER CODE里的代码不会被重写。

Debug → Resume

Project Properties：build steps：

```
arm-none-eabi-objcopy "${ProjName}.elf" -O ihex "${ProjName}.hex"
```

GPIO

GPIO：General-purpose input/output 通用IO

每个GPIO port都有自己的寄存器，详情

每个GPIO port有8个mode

用法

Enable GPIO Port Clock

在用任何外部设备（peripheral）之前，都需要先打开对应外部设备的时钟。

```
void __HAL_RCC_GPIOx_CLK_ENABLE(); // Enable GPIOx ports clock (x can be A, B,
C, D...)
```

Config and Initialize GPIO

根据`GPIO_Init`内的参数来初始化 GPIOx 外部设备。

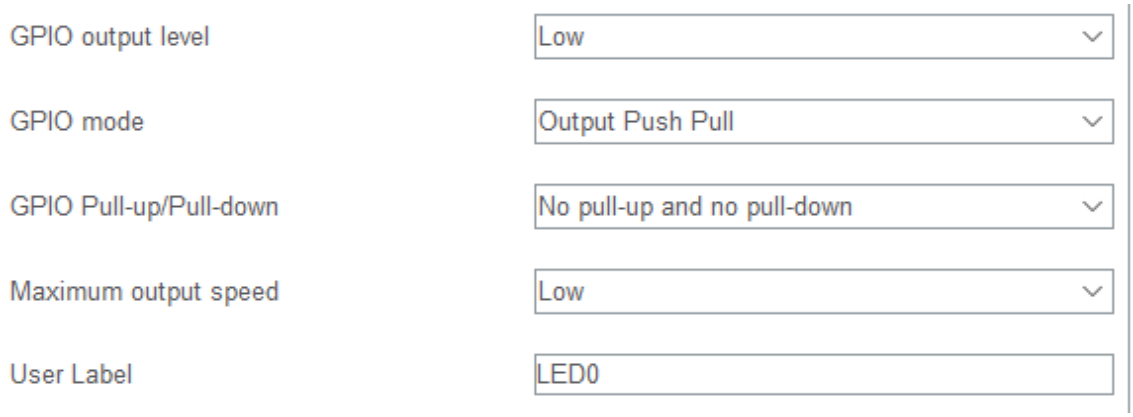
```
typedef struct
{
    uint32_t Pin; /*!< Specifies the GPIO pins to be configured.
                       This parameter can be any value of @ref
GPIO_pins_define */
    uint32_t Mode; /*!< Specifies the operating mode for the selected pins.
                      This parameter can be a value of @ref GPIO_mode_define
*/
    uint32_t Pull; /*!< Specifies the Pull-up or Pull-Down activation for the
                      selected pins.
                      This parameter can be a value of @ref GPIO_pull_define
*/
    uint32_t Speed; /*!< Specifies the speed for the selected pins.
                      This parameter can be a value of @ref
GPIO_speed_define */
} GPIO_InitTypeDef;
```

```
void HAL_GPIO_Init(GPIO_TypeDef *GPIOx, GPIO_InitTypeDef *GPIO_Init)
```

Generate Code by STM32CubeIDE

用图形化的方法初始化 GPIO：

- 打开 **Pinout view** 里的 **Pinout & Configuration**。
- 点击某个 pin，弹出的列表是这个 pin 支持的外部设备。
- **GPIO_Output** 可以用 pin 来控制LED。
- **GPIO_Input** 读取按键的输入。
- **GPIO_Output**：



*关于 pull-up 和 pull-down：

- 这个 GPIO 用于输出，选 no pull。
- 用于输入，且输入的默认值是0，选 pull-down，反之设为 pull-up。

```
while (1)
{
    if (HAL_GPIO_ReadPin(KEY0_GPIO_Port, KEY0_Pin) == GPIO_PIN_SET) {
        HAL_Delay(100);
        HAL_GPIO_TogglePin(LED0_GPIO_Port, LED0_Pin);
    }
    if (HAL_GPIO_ReadPin(KEY1_GPIO_Port, KEY1_Pin) == GPIO_PIN_SET) {
        HAL_Delay(100);
        HAL_GPIO_TogglePin(LED1_GPIO_Port, LED1_Pin);
    }
}
```

`HAL_GPIO_TogglePin` 每隔 100ms 反转一次 LED 的电平。

`GPIO_PIN_SET` ?

External Interrupts

Introduction

遇到中断时，当前的工作要暂停去执行 interrupt handler 或 interrupt service routine(ISR)，执行才返回暂停位置继续执行。

外部中断 **EXTI** 是由外部设备 peripheral 触发的，不同于 CPU 自己自动生成的内部中断。

STM32F103RCTx 有20条外部中断线，所有的 GPIO 都和外部中断线相联。

与相同 EXTI 线相连的 pin 不能同时被用作 EXIT 输入。

Configuration on STM32CubeIDE

通过设置 GPIO input 的 Mode 来触发中断。

按键的电平

KEY0 - PC5

KEY1 - PA15

WK_UP - PA0

LED0 - PA8

LED1 - PD2

Priority

STM32有两种优先级 priority：

- preemption priority
- sub priority

preemption 的值越大，优先级越低。

Programming without ST-Link/J-Link

在 IDE 里：用 debug 将二进制文件烧到 MCU，这种方法需要 ST-Link。

用FlyMCU：通过串口将 .hex 文件烧到板子里，需要配置 USART。

需要在 project setting 里找到 C/C++ build -> Settings -> Build Steps -> Command，加入下面的指令：

```
arm-none-eabi-objcopy "${ProjName}.elf" -O ihex "${ProjName}.hex"
```

在 debug 目录下生成 .hex ，打开FlyMCU，选好 port，bps，.hex文件路径，run！

USART

- 用于串行。
- 全双工。
- 支持同步和异步传输 UART。

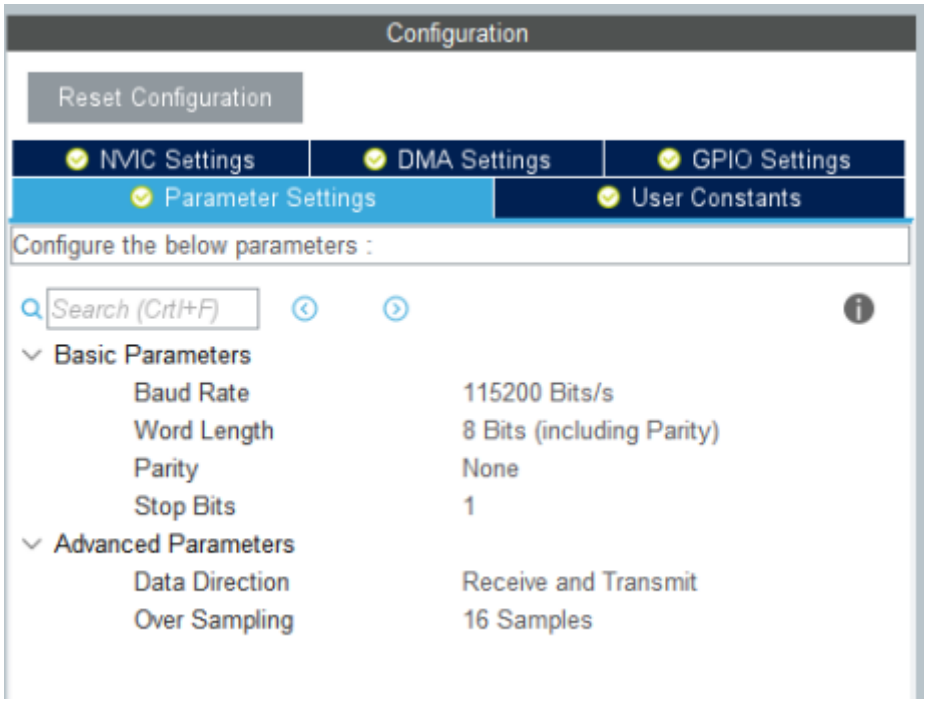
USART 双向连接需要两个 pin：

- PA10 - USART_RX 接收数据。
- PA9 - USART_TX 发送数据。

异步还需要额外的 pin 来绑定 SCLK。

Configure UART in STM32CubeIDE

- 设置任意一个 USART/UART 的模式为 `asynchronous`。
- baud rate, word length, parity，stop bits 这四项要相同。



Transmit and Receive Data in Blocking Mode

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

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

上面的两个函数会一直阻塞，直到完成 transmit/receive 或 timeout。

一般发送数据用阻塞模式，收数据用中断。

Receive Data in Non-blocking Mode

在 configuration 里 打开中断。图

在 **main.c** 中

```
/* Private define ------------------------------------------------------------*/
/* USER CODE BEGIN PD */
    HAL_StatusTypeDef HAL_UART_Receive_IT(UART_HandleTypeDef *huart, uint8_t
*pData,uint16_t Size);
/* USER CODE END PD */
```

```
/* Private variables ---------------------------------------------------------*/
UART_HandleTypeDef huart4;
HAL_StatusTypeDef HAL_UART_Receive_IT(UART_HandleTypeDef *huart, uint8_t
*pData,uint16_t Size);
```

这两个位置应该都可以。

这里的 **huart** 不要替换。

```
/* USER CODE BEGIN PV */
uint8_t rxBuffer[20];
/* USER CODE END PV */
//...
int main(void)
{
    //...
    /* USER CODE BEGIN 2 */
    HAL_UART_Receive_IT(&huart1, (uint8_t *)rxBuffer, 1);
    /* USER CODE END 2 */
    //...
}
```

`HAL_UART_Receive_IT`以非阻塞方式接受数据并触发中断。当 UART 遇到中断时，执行`USART1_IRQHandler`。

*注意这里的 **&huart1** 要替换成自己选择的 **uart**。

```
void USART1_IRQHandler(void)
{
    /* USER CODE BEGIN USART1_IRQn 0 */
    /* USER CODE END USART1_IRQn 0 */
    HAL_UART_IRQHandler(&huart1);
    /* USER CODE BEGIN USART1_IRQn 1 */
    HAL_UART_Receive_IT(&huart1, (uint8_t *)rxBuffer, 1);
    /* USER CODE END USART1_IRQn 1 */
}
```

调用 `HAL_UART_IRQHandler` 函数处理中断请求。

之后再次调用 `HAL_UART_Receive_IT` 保持数据接收。

注意这里的 **&huart1** 要替换成自己选择的 **uart**。

```
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
    if(huart->Instance==USART1) // 注意这里 USART1 也要替换
    {
        static unsigned char uRx_Data[1024] = {0};
        static unsigned int uLength = 0;
        if(rxBuffer[0] == '\n')
        {
            HAL_UART_Transmit(&huart1, uRx_Data, uLength, 0xffff);
            uLength = 0;
        }
        else
        {
            uRx_Data[uLength] = rxBuffer[0];
            uLength++;
        }
    }
}
```

要以非阻塞模式接收数据需要重写这个函数，用一个数组来存数据，遇到换行就发送这些数据。

如果 debug 有 error：first define here 直接注释 error 部分代码。

发送数据：

```
static unsigned char uRx_Data[1024] = {0}; // 'xxx', 'xx' ...
HAL_UART_Transmit(&huart1, uRx_Data, uLength, 0xffff);
```

Timer Interrupt

Introduction

STM32 有三种 timer：

- advanced
- general purpose
- basic

不同 timer 的比较：lab 5

error: 重复定义函数：找到库里的函数注释掉

Pulse Width Modulation

Watchdog

watchdog timer 用来检测设备故障并在故障时重启恢复。

在watchdog timer 里有一个 down counter，当计数器到 reload value 系统就会重启。通常，系统会周期性重启计数器防止系统被重启。

STM32 的两个 watchdog timer：

- IWDG：independent watchdog
- WWDG： window watchdog

IWDG 有自己专属的 LSI（低速clock），所以当系统时钟失效时仍能工作。IWDG 的计数器从 0xFFF 开始计数，到 0x000 就重启 MCU。

WWDG 的计数器是从 APB1 预置的，它有一个可配置的 time-window （可以用来检测应用的异常行为）。如果要刷新计数器，则必须在 time-window 的范围内，否则会导致 MCU 重启。window 的下界为 0x4F，上界由用户决定。当计数器到 0x40时，WWDG还可以触发 wakeup 中断。

Configure UART in STM32CubeIDE

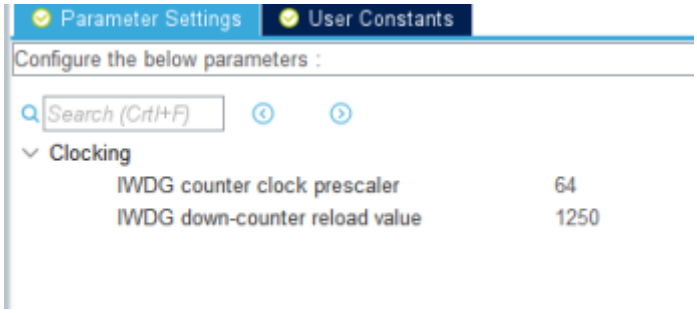
激活并配置参数。

IWDG

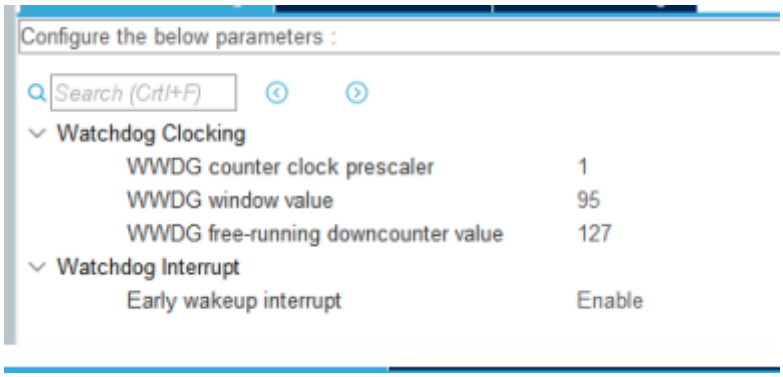
T_out = presaler * reload / freq

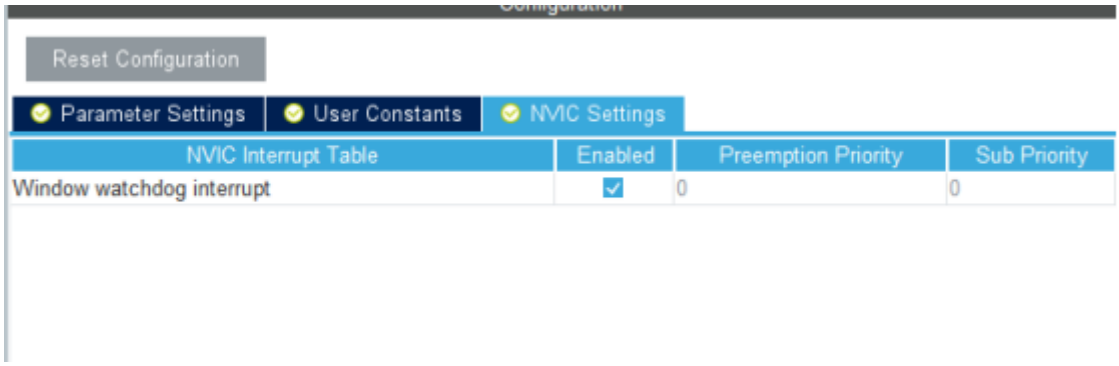
The freq of STM32RCT6 LSI is 40Hz

算出的 T_out 就是启动多少 ms 后重启?



WWDG





按钮用作 GPIO input 和 EXIT 有什么区别?

IWDG

下面代码加到 main.c

```
char msg [10];
int i = 0;
HAL_UART_Transmit(&huart1, "Restart\n", 8, HAL_MAX_DELAY);
while (1)
{
    /* USER CODE END WHILE */
    /* USER CODE BEGIN 3 */
    HAL_GPIO_EXTI_Callback（KEY_1_Pin);
    i++;
    sprintf(msg, "%d\r\n", i);  //  格式化字符串
    HAL_UART_Transmit(&huart1, (uint8_t*)msg, strlen(msg), HAL_MAX_DELAY);
    HAL_Delay(1000);
}
```

```c
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    HAL_Delay(100);
    switch (GPIO_Pin) {
        case KEY1_Pin:
            if (HAL_GPIO_ReadPin(KEY1_GPIO_Port, KEY1_Pin) == GPIO_PIN_RESET) {
                HAL_IWDG_Refresh(&hiwdg);
            }
            break;
        default:
            break;
    }
}
```

效果： When press the KEY1 continuously, the output will increase, otherwise, it will back to 1.

WWDG

在 main 重写 `HAL_WWDG_EarlyWakeupCallback`

```c
void HAL_WWDG_EarlyWakeupCallback(WWDG_HandleTypeDef *hwwdg)
{
    HAL_WWDG_Refresh(hwwdg);
}
```

```c
char msg [10];
int i = 0;
HAL_UART_Transmit(&huart1, "Restart\n", 8, HAL_MAX_DELAY);
while (1)
{
    /* USER CODE END WHILE */
    /* USER CODE BEGIN 3 */
    HAL_GPIO_EXTI_Callback（KEY_1_Pin);
    i++;
    sprintf(msg, "%d\r\n", i);  //  格式化字符串
    HAL_UART_Transmit(&huart1, (uint8_t*)msg, strlen(msg), HAL_MAX_DELAY);
    HAL_Delay(1000);
}
```

Analogue-to-Digital Converter

模拟-数字转换器 ADC 能把模拟信号转化为数字信号。

板子有三个 ADC，彼此独立。

12 bit-ADC 有 18 个多路复用信道，可以测量 16 个外部信号源和 2 个内部信号源。

A/D 转换有多种模式，结果储存在16-bit 左对齐或右对齐数据寄存器里。

channel 16 — Temperature Sensor Channel

其他信道

Configuration on STM32CubeIDE

Analog - **ADC1** - **IN1**，in 后面的数字代表信道对应的 ADC，对应表。

Parameter Settings - ADC_Settings - Data Alignment 右对齐

Single Channel, Single Conversion Mode

Single Conversion Mode 只转换一次，如果需要更新测量需要重启转换。

```
// in main.c file
int main(void)
{
    // ...
    /* USER CODE BEGIN 1 */
    uint16_t raw;
    char msg[20];
    // ...
    while (1)
    {
        HAL_ADC_Start(&hadc1);
        // Wait for regular group conversion to be completed
        HAL_ADC_PollForConversion(&hadc1, HAL_MAX_DELAY);
        // Get ADC value
        raw = HAL_ADC_GetValue(&hadc1); // the voltage should be raw *
(3.3/4096)(12bits)
        // Convert to string and print
        sprintf(msg, "%u\r\n", raw);
        HAL_UART_Transmit(&huart1, (uint8_t*)msg, strlen(msg), HAL_MAX_DELAY);
    }
}
```

Single Channel, Continuous Conversion Mode

连续模式，自动重启转换

```
// in main.c file
int main(void)
{
    // ...
    /* USER CODE BEGIN 1 */
    uint16_t raw;
    char msg[20];
    // ...
    HAL_ADC_Start(&hadc1);
    while (1)
    {
        // Wait for regular group conversion to be completed
        HAL_ADC_PollForConversion(&hadc1, HAL_MAX_DELAY);
        // Get ADC value
        raw = HAL_ADC_GetValue(&hadc1); // the voltage should be raw *
(3.3/4096)(12bits)
        // Convert to string and print
        sprintf(msg, "%u\r\n", raw);
        HAL_UART_Transmit(&huart2, (uint8_t*)msg, strlen(msg), HAL_MAX_DELAY);
    }
}
```

Multiple Channel, Single Conversion Mode

一次测量多个信道

```
while (1)
{
    /* USER CODE END WHILE */
    /* USER CODE BEGIN 3 */
    for (int i = 0; i < 3; i++)
    {
        HAL_ADC_Start(&hadc1);
        HAL_ADC_PollForConversion(&hadc1, HAL_MAX_DELAY);
        adcBuf[i]=HAL_ADC_GetValue(&hadc1);
        sprintf(msg, "ch:%d %d\r\n", i, adcBuf[i]);
        HAL_UART_Transmit(&huart1, (uint8_t*)msg, strlen(msg), HAL_MAX_DELAY);
    }
    HAL_Delay(500);
}
```

lab10 FreeRTOS

Introduction

实时操作系统 RTOS 服务实时应用

没有 buffer 的延迟

有时间限制，RTOS中的进程必须在约束内完成，否则系统异常

允许 multi-tasking，多个程序可以并发

通过 time-sharing（可用的处理器时间被分给多个进程）来实现并发。这些进程在时间片中被操作系统的任务调度子系统反复中断。

一个 task 一般有三个状态：

1. running(executing on the CPU)
2. Ready(ready to be executed)
3. Blocked(waiting for an event, I/O for example)

   大部分时间里，大部分的 task都是 blocked 或 ready，因为一般同时只能有一个 task 运行在 CPU 上。由于任务是在一个小的时间片中轮流调度的，因此似乎有多个任务同时运行。

Configuration on STM32CubeIDE

图片

不要在 while 里加代码，不会执行。

在 main.c 实现 `Func_LED0` 和 `Func_LED1`

```
/* USER CODE BEGIN Header_Func_LED0 */
/**
    * @brief Function implementing the Task_LED0 thread.
    * @param argument: Not used
    * @retval None
    */
/* USER CODE END Header_Func_LED0 */
void Func_LED0(void *argument)
```

```
{
    /* USER CODE BEGIN Func_LED0 */
    /* Infinite loop */
    for(;;)
    {
        HAL_GPIO_TogglePin(LED0_GPIO_Port, LED0_Pin);
        osDelay(500);
    }
    /* USER CODE END Func_LED0 */
}
/* USER CODE BEGIN Header_Func_LED1 */
/**
* @brief Function implementing the Task_LED1 thread.
* @param argument: Not used
* @retval None
*/
/* USER CODE END Header_Func_LED1 */
void Func_LED1(void *argument)
{
    /* USER CODE BEGIN Func_LED1 */
    /* Infinite loop */
    for(;;)
    {
        HAL_GPIO_TogglePin(LED1_GPIO_Port, LED1_Pin);
        osDelay(500);
    }
    /* USER CODE END Func_LED1 */
}
```

`osDelay` 延迟当前 task，可以执行其他 task。

`HAL_Delay` CPU什么都不做。

lab11 FreeRTOS-Semaphore

Introduction

在多任务操作系统中，信号量 Semaphore 是用来控制多个进程访问公共资源的变量。信号量的值代表多进程可获得的资源数量。

信号量的两个操作：V是增加，P是减少。

当信号量值为0时执行P操作，执行这个操作的进程会被阻塞，直到信号量值大于0.

允许任意资源计数的信号被称为计数信号，而限制在0和1的信号(锁定/解锁，不可用/可用)被称为二进制信号，用于实现互斥锁。

Configuration on STM32CubeIDE

图片

lab12 FreeRTOS Inter-task Communication

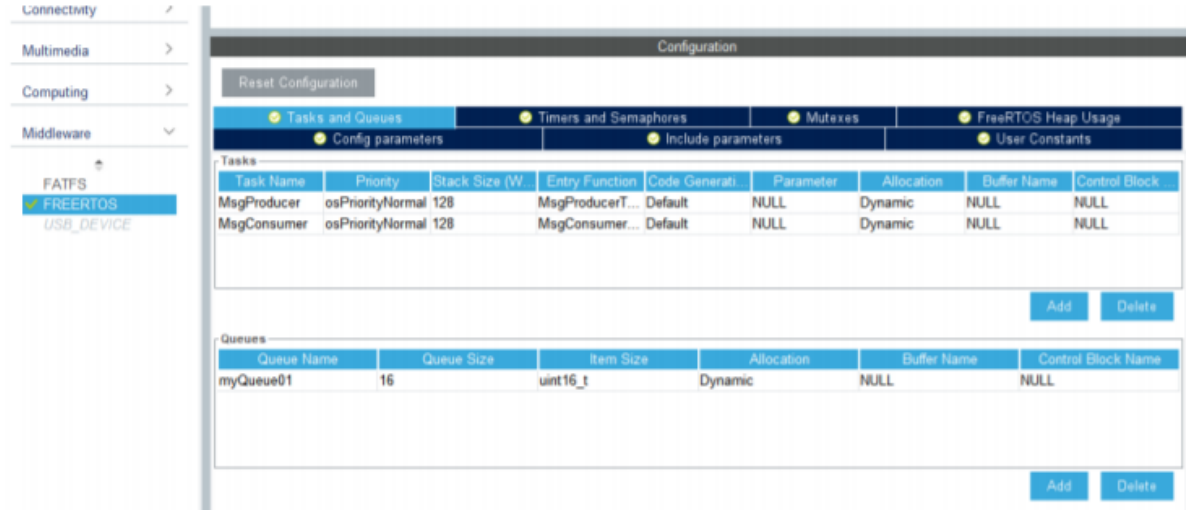
Introduction

RTOS一次只能执行一个任务，需要任务间通信来在task之间交换信息。

Message queues

FIFO queue

Configuration



Mail Queues

用内存池来分配和释放内存。

内存池满了就不能put了。