

尚硅谷嵌入式技术之 FreeRTOS

实时操作系统

（作者：尚硅谷研究院）

版本：V1.0.1

第 1 章 RTOS 入门

1.1 裸机与 RTOS 介绍（了解）

裸机编程是指在嵌入式系统中，直接在硬件上运行代码，没有操作系统的支持。这种方式下，开发者需要完全掌握硬件资源，包括时钟、中断、外设等。任务调度和资源管理都由开发者手动管理。这就像手动操纵一辆汽车，想开车从城市 A 到城市 B，你需要了解汽车的每个部件，掌握如何驾驶，包括油门、刹车、方向盘等。你需要手动决定何时加速、何时刹车、何时转弯。这就好比裸机编程，开发者需要亲自管理每个硬件资源，编写所有的控制逻辑。

RTOS 全称是 Real Time Operating System，中文名就是实时操作系统，提供了任务调度、内存管理、中断处理等功能。RTOS 能够让开发者更专注于应用层的开发，而不用亲自管理底层硬件资源。想从城市 A 到城市 B，你可以选择坐出租车。在出租车上，你只需要告诉司机目的地，不用亲自操纵汽车的每个部分。司机会负责加速、刹车、转弯等操作。这就好比使用 RTOS，开发者只需定义任务、调度和数据通信，RTOS 会负责底层管理。

总的来说，裸机编程就像是自己开车，而使用 RTOS 则像是坐出租车，更专注于目的地而非具体的驾驶操作。

- 任务调度：裸机编程需要手动调度任务，而 RTOS 提供自动的任务调度器。
- 硬件管理：裸机编程需要开发者手动管理硬件资源，RTOS 提供了抽象接口，简化了硬件管理。
- 复杂性：裸机编程相对较复杂，需要深入了解硬件细节。RTOS 提供了更高层次的抽象，简化了开发流程。

1.2 FreeRTOS 简介（了解）

RTOS 是指一类系统，如 FreeRTOS，uC/OS，RTX，RT-Thread 等，都是 RTOS 类操作系统。

1.2.1 FreeRTOS 优势

FreeRTOS 是一款受欢迎、广泛应用于嵌入式系统的 RTOS，其开源、轻量级、可移植的特点使其成为许多嵌入式开发者的首选，主要优势如下：

- 开源和免费：FreeRTOS 是一款开源的 RTOS，采用 MIT 许可证发布，可以免费使用、修改和分发。
- 轻量级设计：FreeRTOS 注重轻量级设计，适用于资源受限的嵌入式系统，不占用过多内存和处理器资源。
- 广泛应用：FreeRTOS 在嵌入式领域得到广泛应用，包括工业自动化、医疗设备、消费电子产品、汽车电子等。
- 多平台支持：FreeRTOS 的设计注重可移植性，可以轻松地移植到不同的硬件平台，支持多种处理器架构。
- 丰富的功能：提供了多任务调度、任务通信、同步等功能，适用于复杂的嵌入式应用场景。

1.2.2 FreeRTOS 介绍

官网：<https://freertos.org/>，并且支持中文。

- 任务调度：FreeRTOS 通过任务调度器管理多个任务，支持不同优先级的任务，实现任务的有序执行。
- 任务通信和同步：提供了队列、信号量等机制，支持任务之间的通信和同步，确保数据的安全传递。
- 内存管理：提供简单的内存管理机制，适用于嵌入式环境，有效利用有限的内存资源。
- 定时器和中断处理：支持定时器功能，能够处理中断，提供了可靠的实时性能。
- 开发社区：拥有庞大的用户社区，开发者可以在社区中获取支持、解决问题，并分享经验。
- 可移植性：设计注重可移植性，可以轻松地移植到不同的硬件平台，提高了代码的

更多 [Java](#) - [大数据](#) - [前端](#) - [python](#) 人工智能资料下载，可百度访问：[尚硅谷官网](#)

重用性。

第 2 章 FreeRTOS 基础知识

2.1 任务调度简介（熟悉）

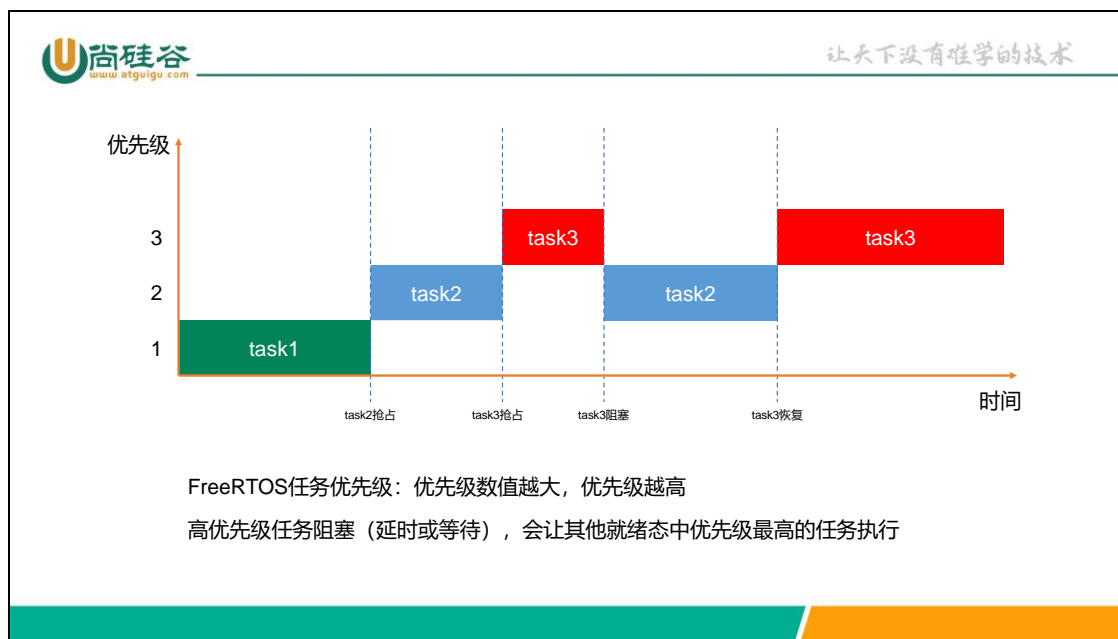
一个处理器核心在某一时刻只能运行一个任务，如果在各个任务之间迅速切换，这样看起来就像多个任务在同时运行。操作系统中任务调度器的责任就是决定在某一时刻要执行哪个任务。

FreeRTOS 使用基于优先级的抢占式任务调度策略。

➤ 抢占式调度：FreeRTOS 采用抢占式调度方式，允许更高优先级的任务在任何时刻抢占正在执行的低优先级任务。这确保了高优先级任务能够及时响应，并提高了系统的实时性。

➤ 时间片轮转：在相同优先级的任务之间，FreeRTOS 采用时间片轮转策略。每个任务执行一个时间片，如果有其他同优先级的任务等待执行，则切换到下一个任务。这有助于公平地分配 CPU 时间。

但是并不是说高优先级的任务会一直执行，导致低优先级的任务无法得到执行。如果高优先级任务等待某个资源（延时或等待信号量等）而无法执行，调度器会选择执行其他就绪的高优先级的任务。



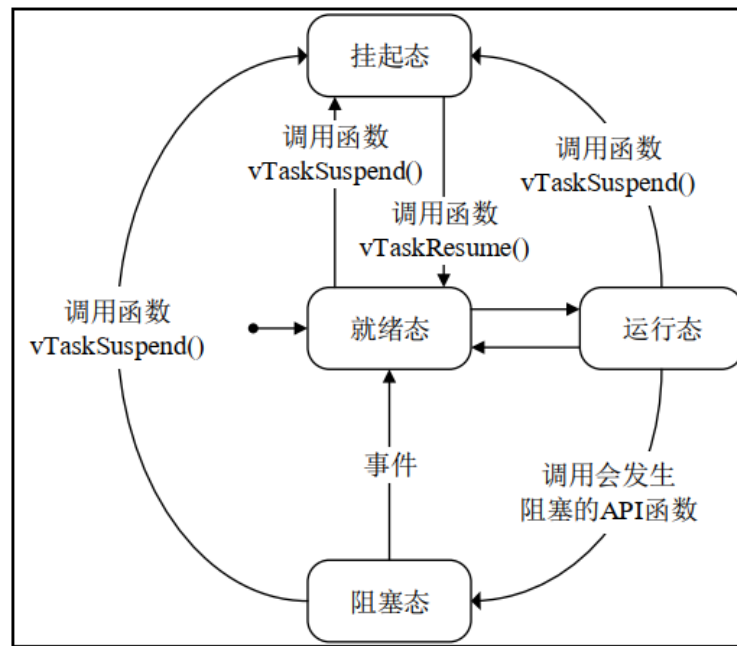


2.2 任务状态（熟悉）

FreeRTOS 中任务共存在 4 种状态：

- 运行态：当任务实际执行时，它被称为处于运行状态。如果运行 RTOS 的处理器只有一个内核，那么在任何给定时间内都只能有一个任务处于运行状态。注意在 STM32 中，同一时间仅一个任务处于运行态。
- 就绪态：准备就绪任务指那些能够执行（它们不处于阻塞或挂起状态），但目前没有执行的任务，因为同等或更高优先级的不同任务已经处于运行状态。
- 阻塞态：如果任务当前正在等待延时或外部事件，则该任务被认为处于阻塞状态。
- 挂起态：类似暂停，调用函数 `vTaskSuspend()` 进入挂起态，需要调用解挂函数 `vTaskResume()` 才可以进入就绪态。

只有就绪态可转变成运行态，其他状态的任务想运行，必须先转变成就绪态。转换关系如下：

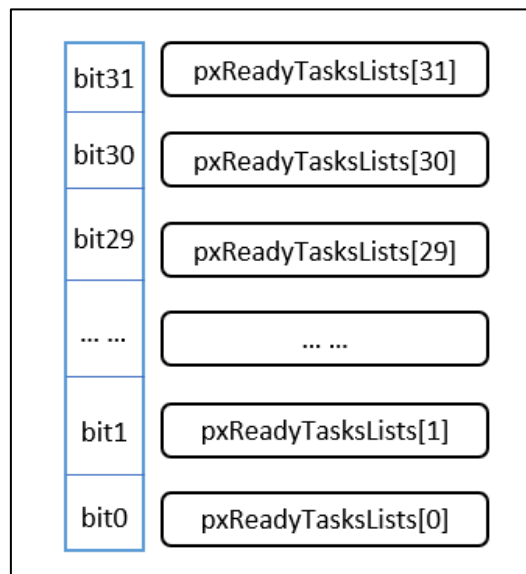


这四种状态中，除了运行态，其他三种任务状态的任务都有其对应的任务状态列表：

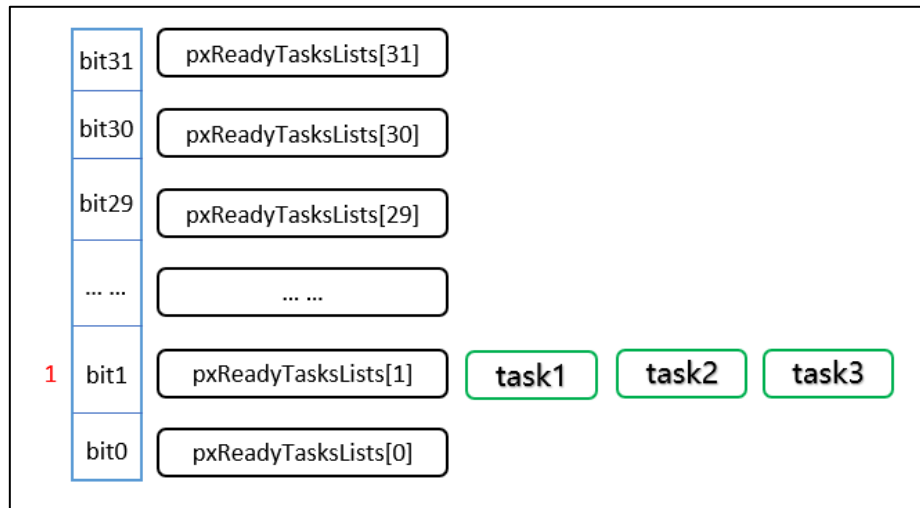
- 就绪列表：`pxReadyTasksLists[x]`，其中 `x` 代表任务优先级数值。
- 阻塞列表：`pxDelayedTaskList`。
- 挂起列表：`xsuspendedTaskList`。

列表类似于链表，后面章节会专门介绍。

以就绪列表为例。如果在 32 位的硬件中，会保存一个 32 位的变量，代表 0-31 的优先级。当某个位，置一时，代表所对应的优先级就绪列表有任务存在。



如果有多个任务优先级相同，会连接在同一个就绪列表上：



调度器总是在所有处于就绪列表的任务中，选择具有最高优先级的任务来执行。

第 3 章 FreeRTOS 移植

3.1 FreeRTOS 源码结构介绍

3.1.1 获取源码

1) 官网下载

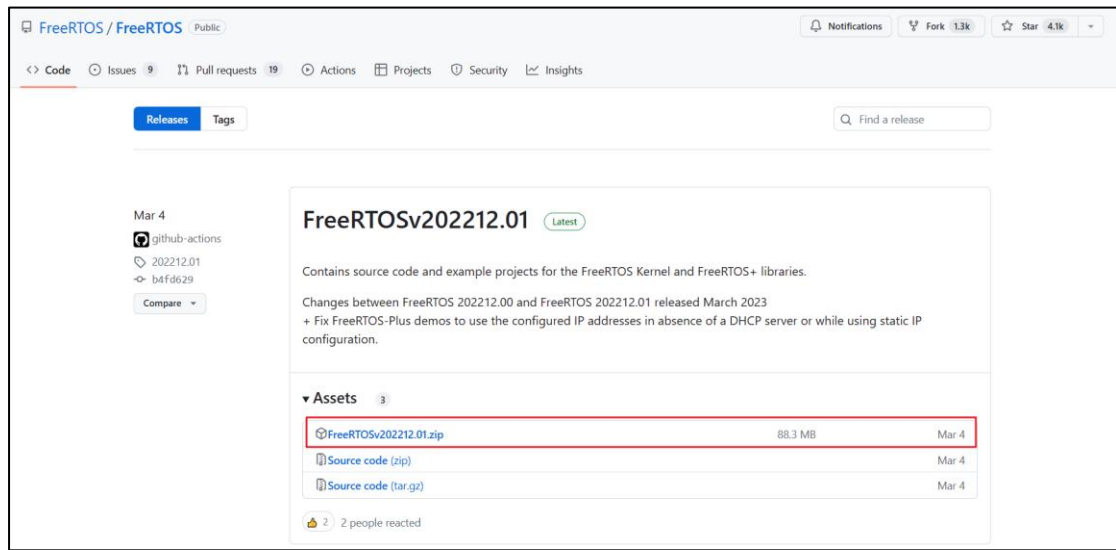
官网地址: <https://www.freertos.org/>



这里我们选择当前最新的 202212.01 版本下载。

2) Github 下载

Github 地址: <https://github.com/FreeRTOS/FreeRTOS/releases>



现在 FreeRTOS 已经将源码迁移到 Github 上，可以直接下载。

3.1.2 源码结构介绍

1) 源码整体结构

名称	描述
FreeRTOS	FreeRTOS 内核
FreeRTOS-Plus	FreeRTOS 组件，一般我们会选择使用第三方的组件
tools	工具
GitHub-FreeRTOS-Home	FreeRTOS 的 GitHub 仓库链接
Quick_Start_Guide	快速入门指南官方文档链接
Upgrading-to-FreeRTOS-xxx	升级到指定 FreeRTOS 版本官方文档链接
History.txt	FreeRTOS 历史更新记录
其他	其他

2) FreeRTOS 文件夹结构

名称	描述
Demo	FreeRTOS 演示例程，支持多种芯片架构、多种型号芯片
License	FreeRTOS 相关许可

Source	FreeRTOS 源码，最重要的文件夹
Test	公用以及移植层测试代码

3) Source 文件夹结构如下

名称	描述
include	内包含了 FreeRTOS 的头文件
portable	包含 FreeRTOS 移植文件：与编译器相关、keil 编译环境
croutine.c	协程相关文件
event_groups.c	事件相关文件
list.c	列表相关文件
queue.c	队列相关文件
stream_buffer.c	流式缓冲区相关文件
tasks.c	任务相关文件
timers.c	软件定时器相关文件

include 文件夹和.c 文件是通用的头文件和 C 文件，这两部分的文件适用于各种编译器和处理器，是通用的。标红的是移植必需的，其他.c 文件根据需要选取。

portable 文件夹里根据编译器、内核等实际环境对应选取。

4) portable 文件夹结构

FreeRTOS 操作系统归根到底是一个软件层面的东西，需要跟硬件联系在一起，portable 文件夹里面的东西就是连接桥梁。由于我们使用 MDK 开发，因此这里只重点介绍其中的部分移植文件。

名称	描述
Keil	指向 RVDS 文件夹
RVDS	不同内核芯片的移植文件
MemMang	内存管理相关文件

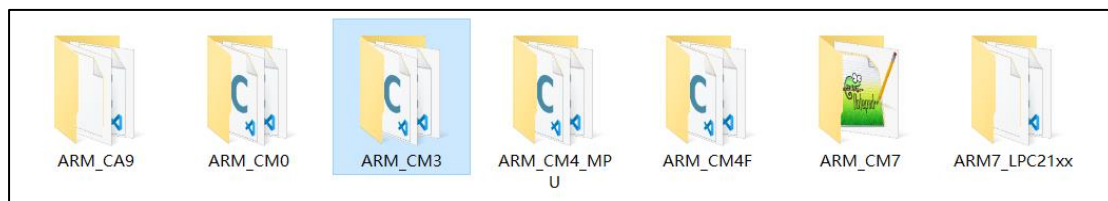
Keil 文件夹里只有一个 See-also-the-RVDS-directory.txt，意思是让我们看 RVDS 文件夹。

(1) RVDS 文件夹

更多 [Java](#) - [大数据](#) - [前端](#) - [python](#) 人工智能资料下载，可百度访问：[尚硅谷官网](#)

RVDS 文件夹包含了各种处理器相关的文件夹，FreeRTOS 是一个软件，单片机是一个硬件，FreeRTOS 要想运行在一个单片机上面，它们就必须关联在一起。

关联还是得通过写代码来关联，这部分关联的文件叫接口文件，通常由汇编和 C 联合编写。这些接口文件都是跟硬件密切相关的，不同的硬件接口文件是不一样的，但都大同小异。编写这些接口文件的过程我们就叫移植，移植的过程通常由 FreeRTOS 和 mcu 原厂的人来负责，移植好的这些接口文件就放在 RVDS 这个文件夹的目录下。



FreeRTOS 为我们提供了 cortex-m0、m3、m4 和 m7 等内核的单片机的接口文件，根据 mcu 的内核选择对应的接口文件即可。其实准确来说，不能够叫移植，应该叫使用官方的移植，因为这些跟硬件相关的接口文件，RTOS 官方都已经写好了，我们只是使用而已。

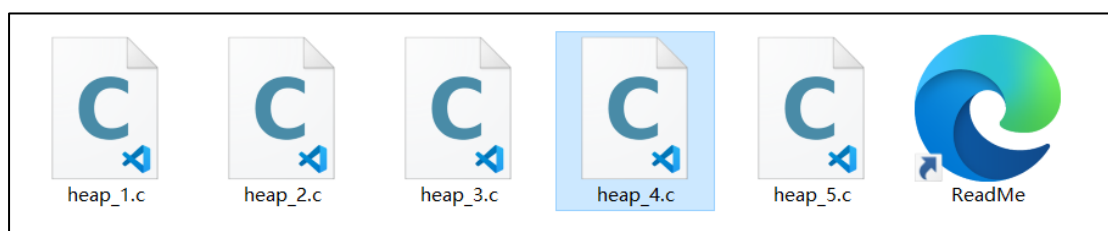
以 ARM_CM3 这个文件夹为例，里面只有“port.c”与“portmacro.h”两个文件，

➤ port.c 文件：里面的内容是由 FreeRTOS 官方的技术人员为 Cortex-M3 内核的处理器写的接口文件，里面核心的上下文切换代码是由汇编语言编写而成，对技术员的要求比较高，我们只是使用的话只需拷贝过来用即可。

➤ portmacro.h 文件：port.c 文件对应的头文件，主要是一些数据类型和宏定义。

（2）MemMang 文件夹

MemMang 文件夹下存放的是跟内存管理相关的，总共有五个 heap 文件以及一个 readme 说明文件。

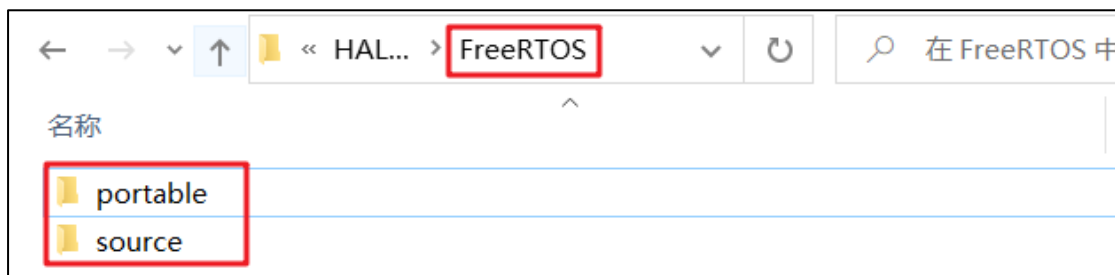


这五个 heap 文件在移植的时候必须使用一个，因为 FreeRTOS 在创建内核对象的时候使用的是动态分配内存，而这些动态内存分配的函数则在这几个文件里面实现，不同的分配算法会导致不同的效率与结果，后面在内存管理中我们会讲解每个文件的区别，由于现在是初学，所以我们选用 heap4.c 即可。

3.2 FreeRTOS 移植步骤

3.2.1 目录添加源码文件

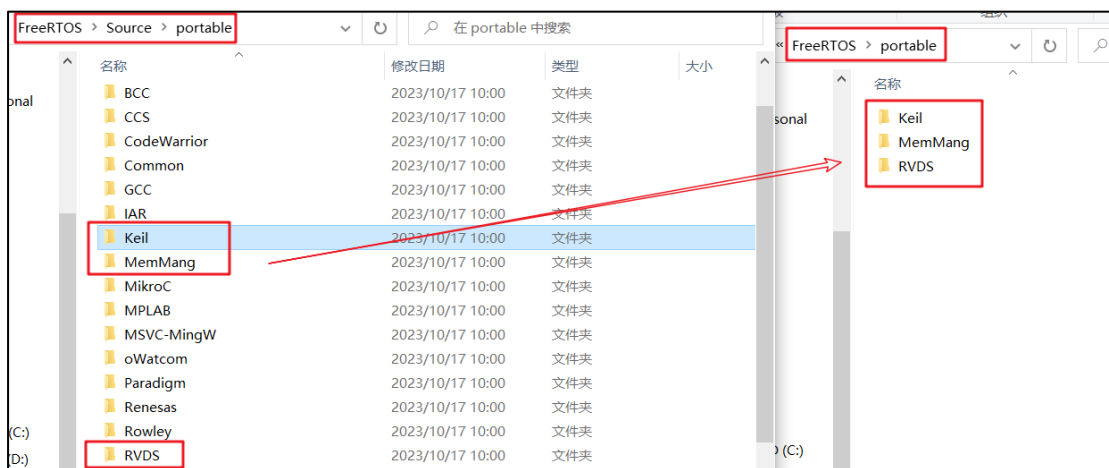
在例程的根路径下，新建“FreeRTOS”文件夹，并且在里面新建“portable”和“source”两个空文件夹。



拷贝 FreeRTOS 源码的 Source 文件夹的 7 个.c 文件到例程的 source 文件夹。



拷贝 FreeRTOS 源码 portable 文件夹下的 Keil、RVDS、MemMang 到例程的 portable 文件夹下。

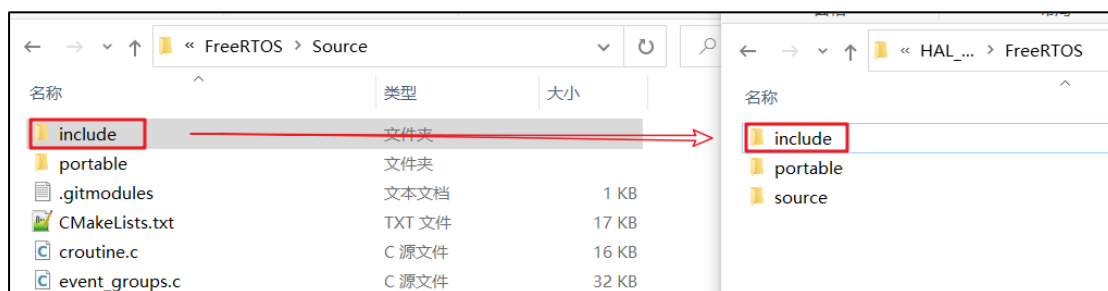


其中例程的 MemMang 可只保留 heap_4.c:



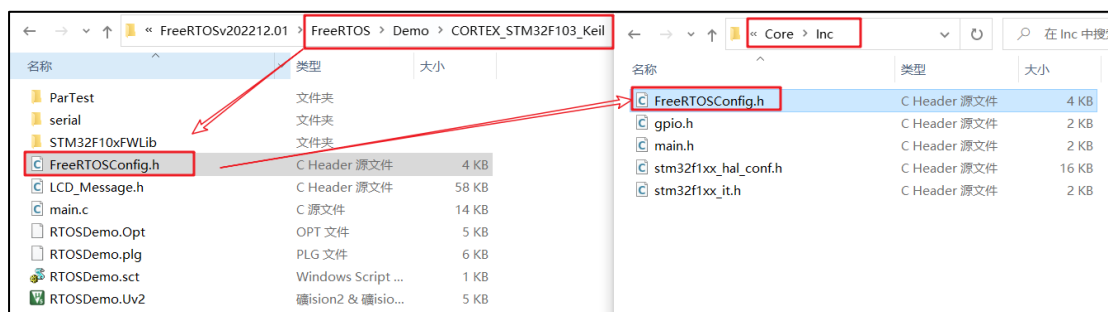
其中例程的 RVDS 可只保留 ARM_CM3（对应我们的芯片内核）。

拷贝 FreeRTOS 源码 include 文件夹到例程的 FreeRTOS 文件夹下。



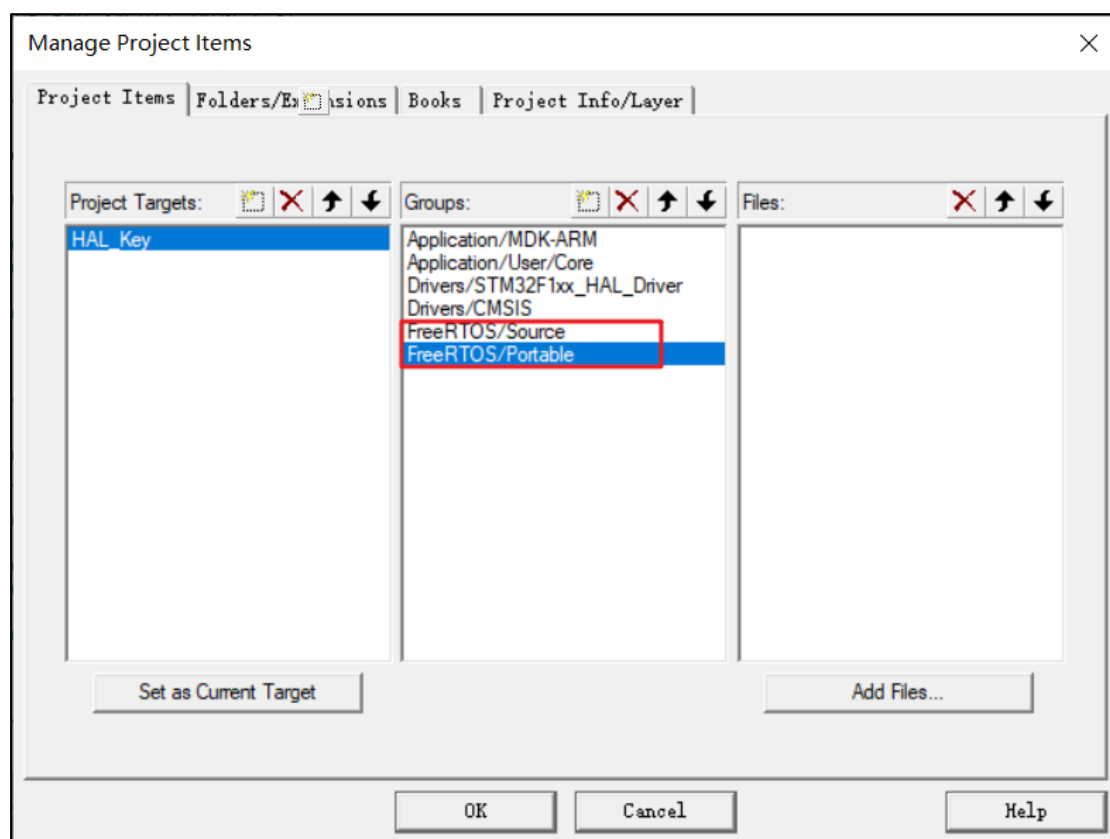
FreeRTOSConfig.h 文件是 FreeRTOS 的工程配置文件，因为 FreeRTOS 是可以裁剪的实时操作内核，应用于不同的处理器平台，用户可以通过修改这个 FreeRTOS 内核的配置头文件来裁剪 FreeRTOS 的功能，所以我们把它拷贝一份放在 user 这个文件夹下面。

在源码“..\FreeRTOS\Demo”文件夹下面找到“CORTEX_STM32F103_Keil”这个文件夹下，找到“FreeRTOSConfig.h”文件，然后拷贝到我们工程下的“Core/Inc”文件夹下即可，等下我们需要对这个文件进行修改。

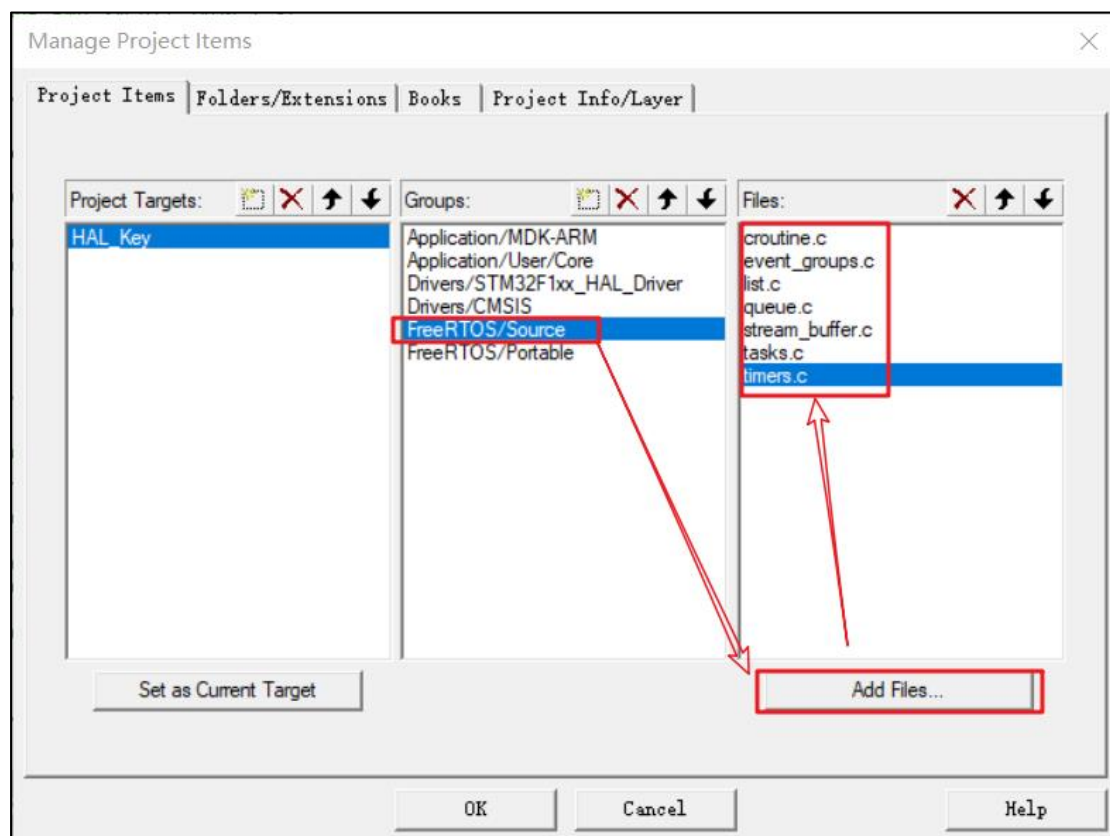


3.2.2 工程添加源码文件

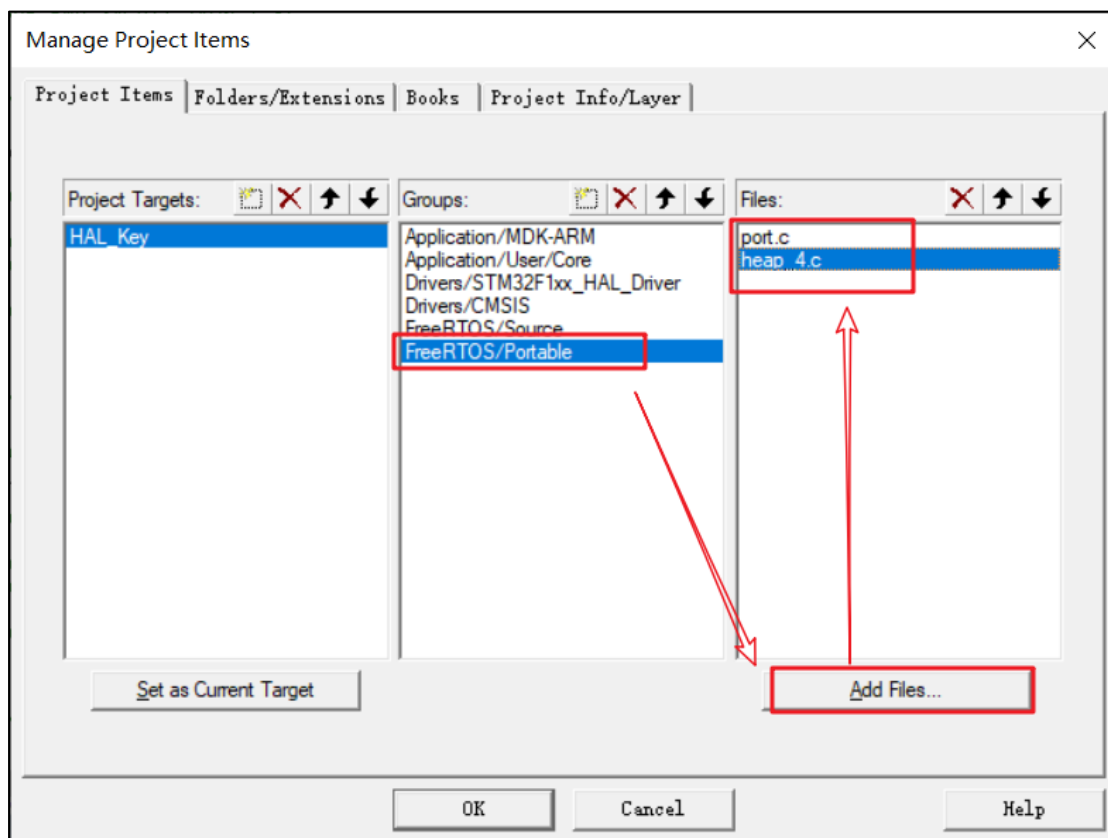
工程新建 Group “FreeRTOS/Source” 和 “FreeRTOS/Portable”。



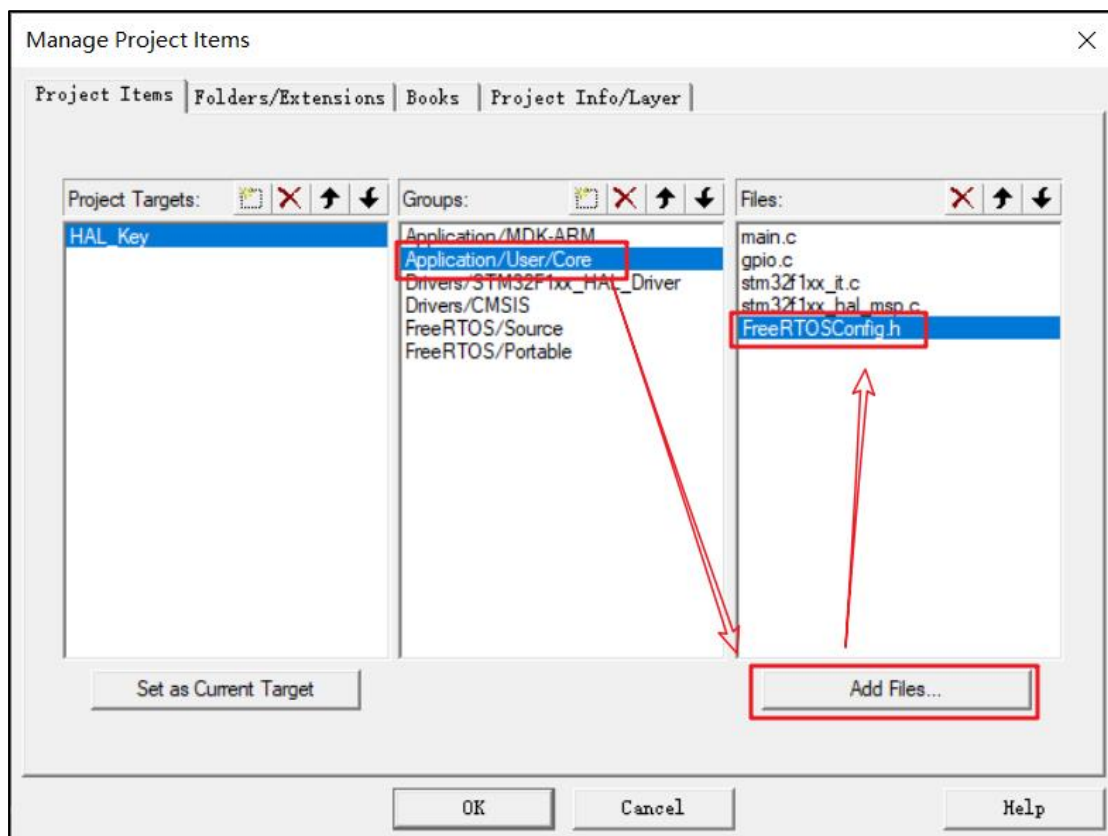
FreeRTOS/Source 添加.c 文件。



FreeRTOS/Portable 添加 port.c 和 heap_4.c 文件。

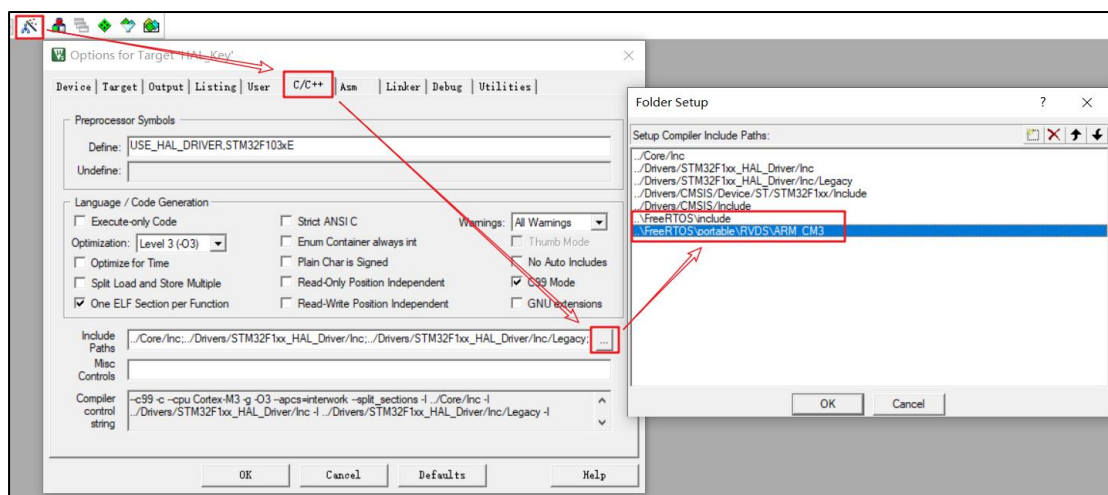


添加配置头文件。



添加头文件。

FreeRTOS 的源码已经添加到开发环境的组文件夹下面，编译的时候需要为这些源文件指定头文件的路径，不然编译会报错。FreeRTOS 的源码里面只有 include 和 RVDS\ARM_CM3 这两个文件夹下面有头文件，只需要将这两个头文件的路径在开发环境里面指定即可。



同时我们还将 FreeRTOSConfig.h 这个头文件拷贝到了工程根目录下的 Core/Inc 文件夹下，这个路径本身就在开发环境里面。

更多 [Java](#) - [大数据](#) - [前端](#) - [python](#) 人工智能资料下载，可百度访问：[尚硅谷官网](#)

3.2.3 系统配置文件修改

FreeRTOSConfig.h 中添加如下 3 个配置：

```
#define xPortPendSVHandler PendSV_Handler
#define vPortSVCHandler SVC_Handler
#define INCLUDE_xTaskGetSchedulerState 1
```

3.2.4 修改 stm32f1xx_it.c

1) 引入头文件

```
/* Private includes -----
-----*/
/* USER CODE BEGIN Includes */
#include "FreeRTOS.h"
#include "task.h"
/* USER CODE END Includes */
```

2) 注释掉 2 个函数

```
// void SVC_Handler(void)
// {
// }

// void PendSV_Handler(void)
// {
// }
```

3) 添加 SysTick 时钟中断服务函数

```
/* Private variables -----
-----*/
/* USER CODE BEGIN PV */
extern void xPortSysTickHandler(void);
/* USER CODE END PV */

void SysTick_Handler(void)
{
    /* USER CODE BEGIN SysTick_IRQn 0 */

    /* USER CODE END SysTick_IRQn 0 */
    HAL_IncTick();
    /* USER CODE BEGIN SysTick_IRQn 1 */
    if (xTaskGetSchedulerState() != taskSCHEDULER_NOT_STARTED)
```

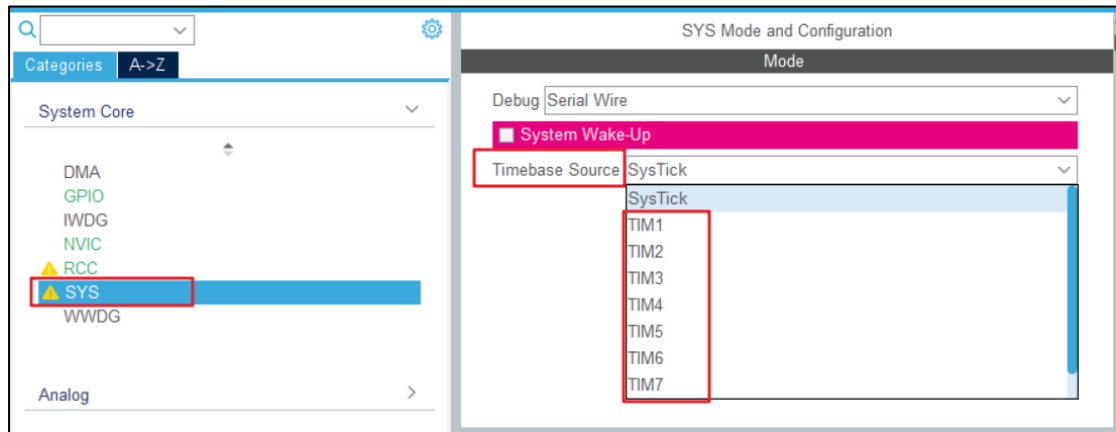
更多 Java - 大数据 - 前端 - python 人工智能资料下载，可百度访问：尚硅谷官网

```

{
    xPortSysTickHandler();
}
/* USER CODE END SysTick_IRQn 1 */
}
    
```

注意：HAL 本身和 FreeRTOS 都需要依赖 SysTick，可能出现。

SysTick 以最低的中断优先级运行，而 FreeRTOS 的临界区功能需要屏蔽中断，可能导致不可预知的问题。为了保险起见，可以考虑在 SYS 选择时钟源的时候换成其他的。



3.3 系统配置文件说明

FreeRTOSConfig.h 配置文件作用：对 FreeRTOS 的功能进行配置和裁剪，以及 API 函数的使能等。

官网中文说明：<https://www.freertos.org/zh-cn-cmn-s/a00110.html>

整体的配置项可以分为三类：

- INCLUDE 开头：一般是“INCLUDE_函数名”，函数的使能，1 表示可用，0 表示禁用。
- config 开头：FreeRTOS 的一些功能配置，比如基本配置、内存配置、钩子配置、中断配置等。
- 其他配置：PendSV 宏定义、SVC 宏定义。

根据需要进行配置，后续章节的知识点和案例，会涉及到其中一些配置，再去熟悉即可。

```

#ifndef FREERTOS_CONFIG_H
#define FREERTOS_CONFIG_H

/* 头文件 */
#include "../SYSTEM/sys/sys.h"
#include "../SYSTEM/usart/usart.h"
    
```

更多 Java - 大数据 - 前端 - python 人工智能资料下载，可百度访问：尚硅谷官网


```
#include <stdint.h>

extern uint32_t SystemCoreClock;

/* 基础配置项 */
#define configUSE_PREEMPTION 1 /*
1: 抢占式调度器, 0: 协程式调度器, 无默认需定义 */
#define configUSE_PORT_OPTIMISED_TASK_SELECTION 1 /
* 1: 使用硬件计算下一个要运行的任务, 0: 使用软件算法计算下一个要运行的任务, 默认: 0 */
#define configUSE_TICKLESS_IDLE 0 /*
1: 使能 tickless 低功耗模式, 默认: 0 */
#define configCPU_CLOCK_HZ SystemCoreClock /
* 定义 CPU 主频, 单位: Hz, 无默认需定义 */
// #define configSYSTICK_CLOCK_HZ (configCPU_CLOCK_HZ /
8) /* 定义 SysTick 时钟频率, 当 SysTick 时钟频率与内核时钟频率不同时才可以定义,
单位: Hz, 默认: 不定义 */
#define configTICK_RATE_HZ 1000 /*
定义系统时钟节拍频率, 单位: Hz, 无默认需定义 */
#define configMAX_PRIORITIES 32 /*
定义最大优先级数, 最大优先级=configMAX_PRIORITIES-1, 无默认需定义 */
#define configMINIMAL_STACK_SIZE 128 /*
定义空闲任务的栈空间大小, 单位: Word, 无默认需定义 */
#define configMAX_TASK_NAME_LEN 16 /*
定义任务名最大字符数, 默认: 16 */
#define configUSE_16_BIT_TICKS 0 /*
1: 定义系统时钟节拍计数器的数据类型为 16 位无符号数, 无默认需定义 */
#define configIDLE_SHOULD_YIELD 1 /*
1: 使能在抢占式调度下, 同优先级的任务能抢占空闲任务, 默认: 1 */
#define configUSE_TASK_NOTIFICATIONS 1 /
* 1: 使能任务间直接的消息传递, 包括信号量、事件标志组和消息邮箱, 默认: 1 */
```

```
#define
configTASK_NOTIFICATION_ARRAY_ENTRIES      1                      /
* 定义任务通知数组的大小，默认：1 */
#define
configUSE_MUTEXES                          1                      /*
1: 使能互斥信号量，默认：0 */
#define
configUSE_RECURSIVE_MUTEXES               1                      /*
1: 使能递归互斥信号量，默认：0 */
#define
configUSE_COUNTING_SEMAPHORES             1                      /
* 1: 使能计数信号量，默认：0 */
#define
configUSE_ALTERNATIVE_API                 0                      /*
已弃用!!! */
#define
configQUEUE_REGISTRY_SIZE                 8                      /*
定义可以注册的信号量和消息队列的个数，默认：0 */
#define
configUSE_QUEUE_SETS                     1                      /*
1: 使能队列集，默认：0 */
#define
configUSE_TIME_SLICING                   1                      /*
1: 使能时间片调度，默认：1 */
#define
configUSE_NEWLIB_REENTRANT                0                      /*
1: 任务创建时分配 Newlib 的重入结构体，默认：0 */
#define
configENABLE_BACKWARD_COMPATIBILITY       0                      /
* 1: 使能兼容老版本，默认：1 */
#define
configNUM_THREAD_LOCAL_STORAGE_POINTERS   0                      /
* 定义线程本地存储指针的个数，默认：0 */
#define
configSTACK_DEPTH_TYPE                    uint16_t                /
* 定义任务堆栈深度的数据类型，默认：uint16_t */
#define
configMESSAGE_BUFFER_LENGTH_TYPE          size_t                 /
* 定义消息缓冲区中消息长度的数据类型，默认：size_t */

/* 内存分配相关定义 */
#define
configSUPPORT_STATIC_ALLOCATION            0                      /
* 1: 支持静态申请内存，默认：0 */
```

更多 [Java](#) - [大数据](#) - [前端](#) - [python](#) 人工智能资料下载，可百度访问：[尚硅谷官网](#)

```
#define
configSUPPORT_DYNAMIC_ALLOCATION          1                      /
* 1: 支持动态申请内存, 默认: 1 */
#define configTOTAL_HEAP_SIZE              ((size_t)(10 *
1024)) /* FreeRTOS 堆中可用的 RAM 总量, 单位: Byte, 无默认需定义 */
#define
configAPPLICATION_ALLOCATED_HEAP          0                      /
* 1: 用户手动分配 FreeRTOS 内存堆(ucHeap), 默认: 0 */
#define
configSTACK_ALLOCATION_FROM_SEPARATE_HEAP  0                      /
* 1: 用户自行实现任务创建时使用的内存申请与释放函数, 默认: 0 */

/* 钩子函数相关定义 */
#define
configUSE_IDLE_HOOK                       0                      /*
1: 使能空闲任务钩子函数, 无默认需定义 */
#define
configUSE_TICK_HOOK                       0                      /*
1: 使能系统时钟节拍中断钩子函数, 无默认需定义 */
#define
configCHECK_FOR_STACK_OVERFLOW            0                      /
* 1: 使能栈溢出检测方法 1, 2: 使能栈溢出检测方法 2, 默认: 0 */
#define
configUSE_MALLOC_FAILED_HOOK              0                      /
* 1: 使能动态内存申请失败钩子函数, 默认: 0 */
#define
configUSE_DAEMON_TASK_STARTUP_HOOK        0                      /
* 1: 使能定时器服务任务首次执行前的钩子函数, 默认: 0 */

/* 运行时间和任务状态统计相关定义 */
#define
configGENERATE_RUN_TIME_STATS             0                      /
* 1: 使能任务运行时间统计功能, 默认: 0 */
#if configGENERATE_RUN_TIME_STATS
#include "../BSP/TIMER/btim.h"
#define
portCONFIGURE_TIMER_FOR_RUN_TIME_STATS()  ConfigureTimeForRunTimeS
tats()
extern uint32_t FreeRTOSRunTimeTicks;
#define
portGET_RUN_TIME_COUNTER_VALUE()          FreeRTOSRunTimeTicks
#endif
```

```
#define
configUSE_TRACE_FACILITY 1 /*
1: 使能可视化跟踪调试, 默认: 0 */
#define
configUSE_STATS_FORMATTING_FUNCTIONS 1 /
* 1: configUSE_TRACE_FACILITY 为 1 时, 会编译 vTaskList()和
vTaskGetRunTimeStats()函数, 默认: 0 */

/* 协程相关定义 */
#define
configUSE_CO_ROUTINES 0 /*
1: 启用协程, 默认: 0 */
#define
configMAX_CO_ROUTINE_PRIORITIES 2 /
* 定义协程的最大优先级, 最大优先级=configMAX_CO_ROUTINE_PRIORITIES-1, 无默
认 configUSE_CO_ROUTINES 为 1 时需定义 */

/* 软件定时器相关定义 */
#define
configUSE_TIMERS 1
/* 1: 使能软件定时器, 默认: 0 */
#define
configTIMER_TASK_PRIORITY ( configMAX_PRIORITIES -
1 ) /* 定义软件定时器任务的优先级, 无默认 configUSE_TIMERS 为 1 时需定义 */
#define
configTIMER_QUEUE_LENGTH 5
/* 定义软件定时器命令队列的长度, 无默认 configUSE_TIMERS 为 1 时需定义
*/
#define
configTIMER_TASK_STACK_DEPTH ( configMINIMAL_STACK_SI
ZE * 2 ) /* 定义软件定时器任务的栈空间大小, 无默认 configUSE_TIMERS 为 1 时需定
义 */

/* 可选函数, 1: 使能 */
#define
INCLUDE_vTaskPrioritySet 1 /*
设置任务优先级 */
#define
INCLUDE_uxTaskPriorityGet 1 /*
获取任务优先级 */
#define
INCLUDE_vTaskDelete 1 /*
删除任务 */
```

```
#define
INCLUDE_vTaskSuspend 1 /*
挂起任务 */
#define
INCLUDE_xResumeFromISR 1 /*
恢复在中断中挂起的任务 */
#define
INCLUDE_vTaskDelayUntil 1 /*
任务绝对延时 */
#define
INCLUDE_vTaskDelay 1 /*
任务延时 */
#define
INCLUDE_xTaskGetSchedulerState 1 /
* 获取任务调度器状态 */
#define
INCLUDE_xTaskGetCurrentTaskHandle 1 /
* 获取当前任务的任务句柄 */
#define
INCLUDE_uxTaskGetStackHighWaterMark 1 /
* 获取任务堆栈历史剩余最小值 */
#define
INCLUDE_xTaskGetIdleTaskHandle 1 /
* 获取空闲任务的任务句柄 */
#define
INCLUDE_eTaskGetState 1 /*
获取任务状态 */
#define
INCLUDE_xEventGroupSetBitFromISR 1 /
* 在中断中设置事件标志位 */
#define
INCLUDE_xTimerPendFunctionCall 1 /
* 将函数的执行挂到定时器服务任务 */
#define
INCLUDE_xTaskAbortDelay 1 /*
中断任务延时 */
#define
INCLUDE_xTaskGetHandle 1 /*
通过任务名获取任务句柄 */
#define
INCLUDE_xTaskResumeFromISR 1 /*
恢复在中断中挂起的任务 */

/* 中断嵌套行为配置 */
```

更多 [Java](#) - [大数据](#) - [前端](#) - [python](#) 人工智能资料下载, 可百度访问: [尚硅谷官网](#)

```
#ifndef __NVIC_PRIO_BITS
    #define configPRIO_BITS __NVIC_PRIO_BITS
#else
    #define configPRIO_BITS 4
#endif

#define
configLIBRARY_LOWEST_INTERRUPT_PRIORITY      15          /* 中
断最低优先级 */
#define
configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY  5          /*
FreeRTOS 可管理的最高中断优先级 */
#define
configKERNEL_INTERRUPT_PRIORITY              ( configLIBRARY_LOWEST_I
NTERRUPT_PRIORITY << (8 - configPRIO_BITS) )
#define
configMAX_SYSCALL_INTERRUPT_PRIORITY          ( configLIBRARY_MAX_SYSC
ALL_INTERRUPT_PRIORITY << (8 - configPRIO_BITS) )
#define
configMAX_API_CALL_INTERRUPT_PRIORITY        configMAX_SYSCALL_INTERR
UPT_PRIORITY

/* FreeRTOS 中断服务函数相关定义 */
#define xPortPendSVHandler                    PendSV_Handler
#define vPortSVCHandler                      SVC_Handler

/* 断言 */
#define vAssertCalled(char, int) printf("Error: %s, %d\r\n", char, int)
#define configASSERT( x ) if( ( x ) == 0 ) vAssertCalled( __FILE__,
__LINE__ )

/* FreeRTOS MPU 特殊定义 */
// #define configINCLUDE_APPLICATION_DEFINED_PRIVILEGED_FUNCTIONS 0
// #define configTOTAL_MPU_REGIONS                        8
// #define configTEX_S_C_B_FLASH                          0x07UL
// #define configTEX_S_C_B_SRAM                          0x07UL
// #define configENFORCE_SYSTEM_CALLS_FROM_KERNEL_ONLY    1
// #define configALLOW_UNPRIVILEGED_CRITICAL_SECTIONS    1

/* ARMv8-M 安全侧端口相关定义。 */
// #define secureconfigMAX_SECURE_CONTEXTS                5

#endif /* FREERTOS_CONFIG_H */
```

第 4 章 FreeRTOS 的任务创建和删除

4.1 任务创建和删除的 API 函数（熟悉）

任务的创建和删除本质就是调用 FreeRTOS 的 API 函数，主要如下：

API 函数	描述
xTaskCreate()	动态方式创建任务
xTaskCreateStatic()	静态方式创建任务
vTaskDelete()	删除任务

➤ 动态创建任务：任务的任务控制块以及任务的栈空间所需的内存，均由 FreeRTOS 从 FreeRTOS 管理的堆中分配。

➤ 静态创建任务：任务的任务控制块以及任务的栈空间所需的内存，需用户分配提供。

4.1.1 动态创建任务函数

1) 函数说明

```
BaseType_t xTaskCreate
(
    TaskFunction_t pxTaskCode,           /* 指向任务函数的指针 */
    const char * const pcName,           /* 任务名字，最大长度
configMAX_TASK_NAME_LEN */
    const configSTACK_DEPTH_TYPE usStackDepth, /* 任务堆栈大小，默认单位
2 字节 */
    void * const pvParameters,           /* 传递给任务函数的参数 */
    UBaseType_t uxPriority,               /* 任务优先级，范围：0 ~
configMAX_PRIORITIES - 1 */
    TaskHandle_t * const pxCreatedTask    /* 任务句柄，就是任务的任
务控制块 */
)
```

返回值说明如下：

- pdPASS：任务创建成功。
- errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY：任务创建失败。

2) 动态创建任务步骤

- （1）将宏 configSUPPORT_DYNAMIC_ALLOCATION 配置为 1。
- （2）定义函数入口参数。

(3) 编写任务函数。

此函数创建的任务会立刻进入就绪态，由任务调度器调度运行。

3) 动态创建任务函数内部实现

(1) 申请堆栈内存&任务控制块内存。

(2) TCB 结构体成员赋值。

(3) 添加新任务到就绪列表中。

任务控制块结构体成员介绍。

```
typedef struct tskTaskControlBlock
{
    volatile StackType_t * pxTopOfStack; /* 任务栈栈顶，必须为 TCB 的第一个成员 */
    ListItem_t xStateListItem;           /* 任务状态列表项 */
    ListItem_t xEventListItem;           /* 任务事件列表项 */
    UBaseType_t uxPriority;               /* 任务优先级，数值越大，优先级越大 */
    StackType_t * pxStack;                /* 任务栈起始地址 */
    char pcTaskName[ configMAX_TASK_NAME_LEN ]; /* 任务名字 */
    ...
    省略很多条件编译的成员
} tskTCB;
```

任务栈栈顶，在任务切换时的任务上下文保存、任务恢复息息相关。每个任务都有属于自己的任务控制块，类似身份证。

4.1.2 静态创建任务函数

1) 函数说明

```
TaskHandle_t xTaskCreateStatic
(
    TaskFunction_t pxTaskCode,           /* 指向任务函数的指针 */
    const char * const pcName,           /* 任务函数名 */
    const uint32_t ulStackDepth,         /* 任务堆栈大小,单位是 4 字节 */
    void * const pvParameters,           /* 传递的任务函数参数 */
    UBaseType_t uxPriority,               /* 任务优先级 */
    StackType_t * const puxStackBuffer, /* 任务堆栈，一般为数组，由用户分配 */
    StaticTask_t * const pxTaskBuffer    /* 任务控制块指针，由用户分配 */
)
```

返回值如下：

➤ NULL：用户没有提供相应的内存，任务创建失败。

更多 [Java](#) - [大数据](#) - [前端](#) - [python](#) 人工智能资料下载，可百度访问：[尚硅谷官网](#)

- 其他值：任务句柄，任务创建成功。

2) 静态创建任务步骤

(1) 将宏 `configSUPPORT_STATIC_ALLOCATION` 配置为 1。

(2) 定义空闲任务&定时器任务的任务堆栈及 TCB。

(3) 实现接口函数：

- `vApplicationGetIdleTaskMemory()`
- `vApplicationGetTimerTaskMemory()`（如果开启软件定时器）

(4) 定义函数入口参数。

(5) 编写任务函数。

此函数创建的任务会立刻进入就绪态，由任务调度器调度运行。

3) 静态创建内部实现

- (1) TCB 结构体成员赋值
- (2) 添加新任务到就绪列表中

4.1.3 任务删除函数

1) 函数说明

```
void vTaskDelete( TaskHandle_t xTaskToDelete )
```

参数说明：xTaskToDelete 待删除任务的任务句柄。当传入的参数为 NULL，则代表删除任务自身（当前正在运行的任务）。

该函数用于删除已被创建的任务，被删除的任务将从就绪态任务列表、阻塞态任务列表、挂起态任务列表和事件列表中移除。

需要注意的是，空闲任务会负责释放被删除任务中由系统分配的内存，但是由用户在任务删除前申请的内存，则需要由用户在任务被删除前提前释放，否则将导致内存泄露。

2) 删除任务流程

- (1) 使用删除任务函数，需将宏 `INCLUDE_vTaskDelete` 配置为 1
- (2) 入口参数输入需要删除的任务句柄（NULL 代表删除本身）

3) 内部实现过程

- (1) 获取所要删除任务的控制块

通过传入的任务句柄，判断所需要删除哪个任务，NULL 代表删除自身。

- (2) 将被删除任务，移除所在列表

将该任务在所在列表中移除，包括：就绪、阻塞、挂起、事件等列表。

（3）判断所需要删除的任务

如果删除任务自身，需先添加到等待删除列表，内存释放将在空闲任务执行；如果删除其他任务，释放内存，任务数量--。

（4）更新下个任务的阻塞时间

更新下一个任务的阻塞超时时间，以防被删除的任务就是下一个阻塞超时的任务。

4.2 任务创建和删除（动态方法）（掌握）

动态创建，堆栈是在 FreeRTOS 管理的堆内存里，注意任务不要重复创建。

xxxxx_STACK_SIZE 128

uxTaskGetStackHighWaterMark() 获取指定任务的任务栈的历史剩余最小值，根据这个结果适当调整启动任务的大小。

4.2.1 实验目标

学会 xTaskCreate() 和 vTaskDelete() 的使用：

- start_task：用来创建其他的三个任务。
- task1：实现 LED1 每 500ms 闪烁一次。
- task2：实现 LED2 每 500ms 闪烁一次。
- task3：判断按键 KEY1 是否按下，按下则删掉 task1。

4.2.2 FreeRTOSConfig.h 代码清单

```
#define configSUPPORT_DYNAMIC_ALLOCATION
```

1

4.2.3 freertos_demo.c 代码清单

1) 任务设置

```
/* 启动任务函数 */
#define START_TASK_PRIORITY 1
#define START_TASK_STACK_DEPTH 128
TaskHandle_t start_task_handler;
void Start_Task(void *pvParameters);

/* Task1 任务 配置 */
#define TASK1_PRIORITY 2
#define TASK1_STACK_DEPTH 128
TaskHandle_t task1_handler;
```

更多 [Java](#) - [大数据](#) - [前端](#) - [python](#) 人工智能资料下载，可百度访问：[尚硅谷官网](#)

```
void Task1(void *pvParameters);

/* Task2 任务 配置 */
#define TASK2_PRIORITY 3
#define TASK2_STACK_DEPTH 128
TaskHandle_t task2_handler;
void Task2(void *pvParameters);

/* Task3 任务 配置 */
#define TASK3_PRIORITY 4
#define TASK3_STACK_DEPTH 128
TaskHandle_t task3_handler;
void Task3(void *pvParameters);
```

2) 入口函数

```
/**
 * @description: FreeRTOS 入口函数：创建任务函数并开始调度
 * @return {*}
 */
void FreeRTOS_Start(void)
{
    xTaskCreate((TaskFunction_t)Start_Task,
                (char *)"Start_Task",
                (configSTACK_DEPTH_TYPE)START_TASK_STACK_DEPTH,
                (void *)NULL,
                (UBaseType_t)START_TASK_PRIORITY,
                (TaskHandle_t *)&start_task_handler);
    vTaskStartScheduler();
}
```

3) 启动任务函数

```
void Start_Task( void * pvParameters )
{
    taskENTER_CRITICAL();          /* 进入临界区 */
    xTaskCreate((TaskFunction_t)   ) Task1,
                (char *)           ) "Task1",
                (configSTACK_DEPTH_TYPE) TASK1_STACK_DEPTH,
                (void *)           ) NULL,
                (UBaseType_t)       ) TASK1_PRIORITY,
                (TaskHandle_t *     ) &task1_handler );

    xTaskCreate((TaskFunction_t)   ) Task2,
```

```
        (char *                )    "Task2",
        (configSTACK_DEPTH_TYPE)    TASK2_STACK_DEPTH,
        (void *                )    NULL,
        (UBaseType_t           )    TASK2_PRIORITY,
        (TaskHandle_t *        )    &task2_handler );

    xTaskCreate((TaskFunction_t    )    Task3,
        (char *                )    "Task2",
        (configSTACK_DEPTH_TYPE)    TASK3_STACK_DEPTH,
        (void *                )    NULL,
        (UBaseType_t           )    TASK3_PRIORITY,
        (TaskHandle_t *        )    &task3_handler );

    vTaskDelete(NULL);
    taskEXIT_CRITICAL();           /* 退出临界区 */
}
```

4) task1 函数

```
/**
 * @description: LED1 每 500ms 翻转一次
 * @param {void *} pvParameters
 * @return {*}
 */
void Task1(void * pvParameters)
{
    while(1)
    {
        printf("task1 运行....\r\n");
        LED_Toggle(LED1_Pin);
        vTaskDelay(500);
    }
}
```

5) task2 函数

```
/**
 * @description: LED2 每 500ms 翻转一次
 * @param {void *} pvParameters
 * @return {*}
 */
void Task2(void * pvParameters)
{
    while(1)
    {
```

```
    printf("task2 运行....\r\n");
    LED_Toggle(LED2_Pin);
    vTaskDelay(500);
}
}
```

6) task3 函数

```
/**
 * @description: 按下 KEY1 删除 task1
 * @param {void *} pvParameters
 * @return {*}
 */
void Task3(void * pvParameters)
{
    uint8_t key = 0;
    while(1)
    {
        printf("task3 正在运行...\r\n");
        key = Key_Detect();
        if(key == KEY1_PRESS)
        {
            if(task1_handler != NULL)
            {
                printf("删除 task1 任务...\r\n");
                vTaskDelete(task1_handler);
                task1_handler = NULL;
            }
        }
        vTaskDelay(10);
    }
}
```

4.3 任务创建和删除（静态方法）（掌握）

4.3.1 实验目标

学会 `xTaskCreateStatic()` 和 `vTaskDelete()` 的使用：

- `start_task`：用来创建其他的三个任务。
- `task1`：实现 LED1 每 500ms 闪烁一次。
- `task2`：实现 LED2 每 500ms 闪烁一次。

- task3: 判断按键 KEY1 是否按下, 按下则删掉 task1。

4.3.2 FreeRTOSConfig.h 代码清单

```
#define configSUPPORT_STATIC_ALLOCATION 1

/* 软件定时器相关定义 */
#define
configUSE_TIMERS 1
/* 使能软件定时器, 默认: 0。为 1 时需要定义下面 3 个 */
#define
configTIMER_TASK_PRIORITY ( configMAX_PRIORITIES -
1 ) /* 软件定时器任务的优先级 */
#define
configTIMER_QUEUE_LENGTH 5
/* 软件定时器命令队列的长度 */
#define
configTIMER_TASK_STACK_DEPTH ( configMINIMAL_STACK_SI
ZE * 2 ) /* 软件定时器任务的栈空间大小 */
```

4.3.3 freertos_demo.c 代码清单

1) 任务设置

```
/* START_TASK 任务 配置
 * 包括: 任务句柄 任务优先级 堆栈大小 创建任务
 */
#define START_TASK_PRI0 1
#define START_TASK_STACK_SIZE 128
TaskHandle_t start_task_handler;
StackType_t start_task_stack[START_TASK_STACK_SIZE];
StaticTask_t start_task_tcb;
void start_task( void * pvParameters );

/* TASK1 任务 配置
 * 包括: 任务句柄 任务优先级 堆栈大小 创建任务
 */
#define TASK1_PRI0 2
#define TASK1_STACK_SIZE 128
TaskHandle_t task1_handler;
StackType_t task1_stack[TASK1_STACK_SIZE];
StaticTask_t task1_tcb;
void task1( void * pvParameters );

/* TASK2 任务 配置
```

更多 [Java](#) - [大数据](#) - [前端](#) - [python](#) 人工智能资料下载, 可百度访问: [尚硅谷官网](#)

```
/* 包括：任务句柄 任务优先级 堆栈大小 创建任务
*/
#define TASK2_PRIIO      3
#define TASK2_STACK_SIZE 128
TaskHandle_t task2_handler;
StackType_t task2_stack[TASK2_STACK_SIZE];
StaticTask_t task2_tcb;
void task2( void * pvParameters );

/* TASK3 任务 配置
* 包括：任务句柄 任务优先级 堆栈大小 创建任务
*/
#define TASK3_PRIIO      4
#define TASK3_STACK_SIZE 128
TaskHandle_t task3_handler;
StackType_t task3_stack[TASK3_STACK_SIZE];
StaticTask_t task3_tcb;
void task3( void * pvParameters );
```

2) 空闲任务和定时器配置及接口函数

```
/* 空闲任务配置 */
StaticTask_t idle_task_tcb;
StackType_t idle_task_stack[configMINIMAL_STACK_SIZE];

/* 软件定时器任务配置 */
StaticTask_t timer_task_tcb;
StackType_t timer_task_stack[configTIMER_TASK_STACK_DEPTH];

/* 空闲任务内存分配 */
void vApplicationGetIdleTaskMemory( StaticTask_t **
ppxIdleTaskTCBBuffer,
                                   StackType_t **
ppxIdleTaskStackBuffer,
                                   uint32_t * pulIdleTaskStackSize )
{
    * ppxIdleTaskTCBBuffer = &idle_task_tcb;
    * ppxIdleTaskStackBuffer = idle_task_stack;
    * pulIdleTaskStackSize = configMINIMAL_STACK_SIZE;
}

/* 软件定时器内存分配 */
void vApplicationGetTimerTaskMemory( StaticTask_t **
ppxTimerTaskTCBBuffer,
```

```
StackType_t **
ppxTimerTaskStackBuffer,
uint32_t * pulTimerTaskStackSize )
{
    * ppxTimerTaskTCBBuffer = &timer_task_tcb;
    * ppxTimerTaskStackBuffer = timer_task_stack;
    * pulTimerTaskStackSize = configTIMER_TASK_STACK_DEPTH;
}
```

3) 入口函数

```
/**
 * @description: FreeRTOS 入口函数：创建任务函数并开始调度
 * @return {*}
 */
void FreeRTOS_Start(void)
{
    start_task_handler = xTaskCreateStatic((TaskFunction_t)Start_Task,
                                           (char *)"Start_Task",
                                           (uint32_t)START_TASK_STACK_DEP
TH,
                                           (void *)NULL,
                                           (UBaseType_t)START_TASK_PRIORI
TY,
                                           (StackType_t
*)start_task_stack,
                                           (StaticTask_t
*)&start_task_tcb);
    vTaskStartScheduler();
}
```

4) 启动函数

```
void Start_Task(void *pvParameters)
{
    taskENTER_CRITICAL(); /* 进入临界区 */

    task1_handler = xTaskCreateStatic((TaskFunction_t)Task1,
                                      (char *)"Task1",
                                      (uint32_t)TASK1_STACK_DEPTH,
                                      (void *)NULL,
                                      (UBaseType_t)TASK1_PRIORITY,
                                      (StackType_t *)task1_stack,
                                      (StaticTask_t *)&task1_tcb);
}
```



```
task2_handler = xTaskCreateStatic((TaskFunction_t)Task2,
                                   (char *)"Task2",
                                   (uint32_t)TASK2_STACK_DEPTH,
                                   (void *)NULL,
                                   (UBaseType_t)TASK2_PRIORITY,
                                   (StackType_t *)task2_stack,
                                   (StaticTask_t *)&task2_tcb);

task3_handler = xTaskCreateStatic((TaskFunction_t)Task3,
                                   (char *)"Task3",
                                   (uint32_t)TASK3_STACK_DEPTH,
                                   (void *)NULL,
                                   (UBaseType_t)TASK3_PRIORITY,
                                   (StackType_t *)task3_stack,
                                   (StaticTask_t *)&task3_tcb);

vTaskDelete(start_task_handler);

taskEXIT_CRITICAL(); /* 退出临界区 */
}
```

5) task1 函数

```
/**
 * @description: LED1 每 500ms 翻转一次
 * @param {void *} pvParameters
 * @return {*}
 */
void Task1(void * pvParameters)
{
    while(1)
    {
        printf("task1 运行...\r\n");
        LED_Toggle(LED1_Pin);
        vTaskDelay(500);
    }
}
```

6) task2 函数

```
/**
 * @description: LED2 每 500ms 翻转一次
 * @param {void *} pvParameters
```

```
* @return {*}
*/
void Task2(void * pvParameters)
{
    while(1)
    {
        printf("task2 运行....\r\n");
        LED_Toggle(LED2_Pin);
        vTaskDelay(500);
    }
}
```

7) task3 函数

```
/**
 * @description: 按下 KEY1 删除 task1
 * @param {void *} pvParameters
 * @return {*}
 */
void Task3(void * pvParameters)
{
    uint8_t key = 0;
    while(1)
    {
        printf("task3 正在运行...\r\n");
        key = Key_Detect();
        if(key == KEY1_PRESS)
        {
            if(task1_handler != NULL)
            {
                printf("删除 task1 任务...\r\n");
                vTaskDelete(task1_handler);
                task1_handler = NULL;
            }
        }
        vTaskDelay(10);
    }
}
```

第 5 章 FreeRTOS 的任务挂起与恢复

5.1 任务的挂起与恢复的 API 函数（熟悉）

- `vTaskSuspend()`: 挂起任务, 类似暂停, 可恢复
- `vTaskResume()`: 恢复被挂起的任务
- `xTaskResumeFromISR()`: 在中断中恢复被挂起的任务

5.1.1 任务挂起函数

```
void vTaskSuspend( TaskHandle_t xTaskToSuspend )
```

- `xTaskToSuspend`: 待挂起任务的任务句柄, 为 NULL 表示挂起任务自身。
- 需将宏 `INCLUDE_vTaskSuspend` 配置为 1。

5.1.2 任务恢复函数

```
void vTaskResume( TaskHandle_t xTaskToResume )
```

- `INCLUDE_vTaskSuspend` 必须定义为 1。
- 不论任务被使用 `vTaskSuspend()` 挂起多少次, 只需调用 `vTaskResume()` 一次, 即可使其继续执行。被恢复的任务会重新进入就绪状态。

5.1.3 任务恢复函数（中断中恢复）

1) 函数说明

```
BaseType_t xTaskResumeFromISR( TaskHandle_t xTaskToResume )
```

返回值如下:

- `pdTRUE`: 任务恢复后需要进行任务切换。
- `pdFALSE`: 任务恢复后不需要进行任务切换。

2) 注意事项

- `INCLUDE_vTaskSuspend` 和 `INCLUDE_xTaskResumeFromISR` 必须定义为 1。
- 在中断服务程序中调用 FreeRTOS 的 API 函数时, 中断的优先级不能高于 FreeRTOS 所管理的最高任务优先级。

5.2 任务挂起与恢复实验（掌握）

5.2.1 实验目标

学会 `vTaskSuspend()`、`vTaskResume()` 任务挂起与恢复相关 API 函数使用:

- start_task: 用来创建其他的三个任务。
- task1: 实现 LED1 每 500ms 闪烁一次。
- task2: 实现 LED2 每 500ms 闪烁一次。
- task3: 判断按键按下逻辑, KEY1 按下, 挂起 task1, 按下 KEY2 在任务中恢复 task1。

5.2.2 FreeRTOSConfig.h 代码清单

```
#define INCLUDE_vTaskSuspend 1
#define INCLUDE_xResumeFromISR 1
```

5.2.3 freertos_demo.c 代码清单

1) 任务设置

```
/* 启动任务函数 */
#define START_TASK_PRIORITY 1
#define START_TASK_STACK_DEPTH 128
TaskHandle_t start_task_handler;
void Start_Task(void *pvParameters);

/* Task1 任务 配置 */
#define TASK1_PRIORITY 2
#define TASK1_STACK_DEPTH 128
TaskHandle_t task1_handler;
void Task1(void *pvParameters);

/* Task2 任务 配置 */
#define TASK2_PRIORITY 3
#define TASK2_STACK_DEPTH 128
TaskHandle_t task2_handler;
void Task2(void *pvParameters);

/* Task3 任务 配置 */
#define TASK3_PRIORITY 4
#define TASK3_STACK_DEPTH 128
TaskHandle_t task3_handler;
void Task3(void *pvParameters);
```

2) 入口函数

```
/**
 * @description: FreeRTOS 入口函数: 创建任务函数并开始调度
 * @return {*}
 */
```

```
void FreeRTOS_Start(void)
{
    xTaskCreate((TaskFunction_t)Start_Task,
                (char *)"Start_Task",
                (configSTACK_DEPTH_TYPE)START_TASK_STACK_DEPTH,
                (void *)NULL,
                (UBaseType_t)START_TASK_PRIORITY,
                (TaskHandle_t *)&start_task_handler);
    vTaskStartScheduler();
}
```

3) 启动任务函数

```
void Start_Task( void * pvParameters )
{
    taskENTER_CRITICAL();          /* 进入临界区 */
    xTaskCreate((TaskFunction_t)   ) Task1,
                (char *)           ) "Task1",
                (configSTACK_DEPTH_TYPE) TASK1_STACK_DEPTH,
                (void *)           ) NULL,
                (UBaseType_t)       ) TASK1_PRIORITY,
                (TaskHandle_t *)    ) &task1_handler );

    xTaskCreate((TaskFunction_t)   ) Task2,
                (char *)           ) "Task2",
                (configSTACK_DEPTH_TYPE) TASK2_STACK_DEPTH,
                (void *)           ) NULL,
                (UBaseType_t)       ) TASK2_PRIORITY,
                (TaskHandle_t *)    ) &task2_handler );

    xTaskCreate((TaskFunction_t)   ) Task3,
                (char *)           ) "Task2",
                (configSTACK_DEPTH_TYPE) TASK3_STACK_DEPTH,
                (void *)           ) NULL,
                (UBaseType_t)       ) TASK3_PRIORITY,
                (TaskHandle_t *)    ) &task3_handler );
    vTaskDelete(NULL);
    taskEXIT_CRITICAL();           /* 退出临界区 */
}
```

4) task1 函数

```
/**
 * @description: LED1 每 500ms 翻转一次
```

```
* @param {void *} pvParameters
* @return {*}
*/
void Task1(void * pvParameters)
{
    while(1)
    {
        printf("task1 运行....\r\n");
        LED_Toggle(LED1_Pin);
        vTaskDelay(500);
    }
}
```

5) task2 函数

```
/**
 * @description: LED2 每 500ms 翻转一次
 * @param {void *} pvParameters
 * @return {*}
 */
void Task2(void * pvParameters)
{
    while(1)
    {
        printf("task2 运行....\r\n");
        LED_Toggle(LED2_Pin);
        vTaskDelay(500);
    }
}
```

6) task3 函数

```
/**
 * @description: 按下 KEY1 挂起 task1, 按下 KEY2 恢复 task1
 * @param {void *} pvParameters
 * @return {*}
 */
void Task3(void *pvParameters)
{
    uint8_t key = 0;
    while (1)
    {
        key = Key_Detect();
        if (key == KEY1_PRESS)
```

```
{
    printf("挂起 task1...\r\n");
    vTaskSuspend(task1_handler);
}
else if (key == KEY2_PRESS)
{
    printf("恢复 task1...\r\n");
    vTaskResume(task1_handler);
}

vTaskDelay(10);
}
```

第 6 章 FreeRTOS 中断管理

6.1 FreeRTOS 中断管理（熟悉）

6.1.1 FreeRTOS 的中断管理


在 STM32 中，中断优先级是通过中断优先级配置寄存器的高 4 位 [7:4] 来配置的。因此 STM32 支持最多 16 级中断优先级，其中数值越小表示优先级越高，即更紧急的中断。

（任务调度的任务优先级相反，是数值越大越优先）

FreeRTOS 可以与 STM32 原生的中断机制结合使用，但它提供了自己的中断管理机制，主要是为了提供更强大和灵活的任务调度和管理功能。

FreeRTOS 中，将 PendSV 和 SysTick 设置最低中断优先级（数值最大，15），保证系统任务切换不会阻塞系统其他中断的响应。

FreeRTOS 利用 **BASEPRI** 寄存器实现中断管理，屏蔽优先级低于某一个阈值的中断。比如：**BASEPRI** 设置为 0x50（只看高四位，也就是 5），代表中断优先级在 5~15 内的均被屏蔽，0~4 的中断优先级正常执行。


让天下没有难学的技术

参数 configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY 设为5（也就是BASEPRI设置为0x50）

中断优先级0
.....
中断优先级4
中断优先级5
.....
中断优先级15

不被FreeRTOS使用，不能调用xxxFromISR函数

中断优先级在5~15内的均被屏蔽，由FreeRTOS管理
可以调用FreeRTOS的xxxFromISR函数
中断可以嵌套

在中断服务函数中调用 FreeRTOS 的 API 函数需注意：

- 中断服务函数的优先级需在 FreeRTOS 所管理的范围内，阈值由 configMAX_SYSCALL_INTERRUPT_PRIORITY 指定。
- 建议将所有优先级位指定为抢占优先级位，方便 FreeRTOS 管理。
- 在中断服务函数里边需调用 FreeRTOS 的 API 函数，必须使用带“FromISR”后缀的函数。

6.1.2 FreeRTOS 的开关中断

FreeRTOS 开关中断函数其实是宏定义，在 portmacro.h 中有定义，如下：

```
#define portDISABLE_INTERRUPTS()      vPortRaiseBASEPRI()
#define portENABLE_INTERRUPTS()       vPortSetBASEPRI( 0 )
```

6.1.3 FreeRTOS 的临界段代码

临界段代码，又称为临界区，指的是那些必须在不被打断的情况下完整运行的代码段。例如，某些外设的初始化可能要求严格的时序，因此在初始化过程中不允许被中断打断。在 FreeRTOS 中，进入临界段代码时需要关闭中断，在处理完临界段代码后再重新开启中断。FreeRTOS 系统本身包含许多临界段代码，并对其进行了保护。在编写用户程序时，有些情况下也需要添加临界段代码以确保代码的完整性。

与临界段代码保护有关的函数有 4 个：

- taskENTER_CRITICAL()：进入临界段。

- `taskEXIT_CRITICAL()` : 退出临界段。
- `taskENTER_CRITICAL_FROM_ISR()` : 进入临界段（中断级）。
- `taskEXIT_CRITICAL_FROM_ISR()`: 退出临界段（中断级）。

进入和退出临界段是成对使用的。每进入一次临界段，全局变量 `uxCriticalNesting` 都会加一，每调用一次退出临界段，`uxCriticalNesting` 减一，只有当 `uxCriticalNesting` 为 0 的时候才会调用函数 `portENABLE_INTERRUPTS()` 使能中断。这确保了在存在多个临界段代码的情况下，不会因为某个临界段代码的退出而破坏其他临界段的保护。只有当所有的临界段代码都退出时，中断才会被重新使能。

6.1.4 挂起和恢复任务调度器

挂起和恢复任务调度器，调用此函数不需要关闭中断：

- `vTaskSuspendAll()`: 挂起任务调度器。
- `xTaskResumeAll()`: 恢复任务调度器。

与临界区不同的是，挂起任务调度器时未关闭中断；这种方式仅仅防止了任务之间的资源争夺，中断仍然可以直接响应；挂起调度器的方法适用于临界区位于任务与任务之间的情况；这样既不需要延迟中断，同时又能确保临界区的安全性。

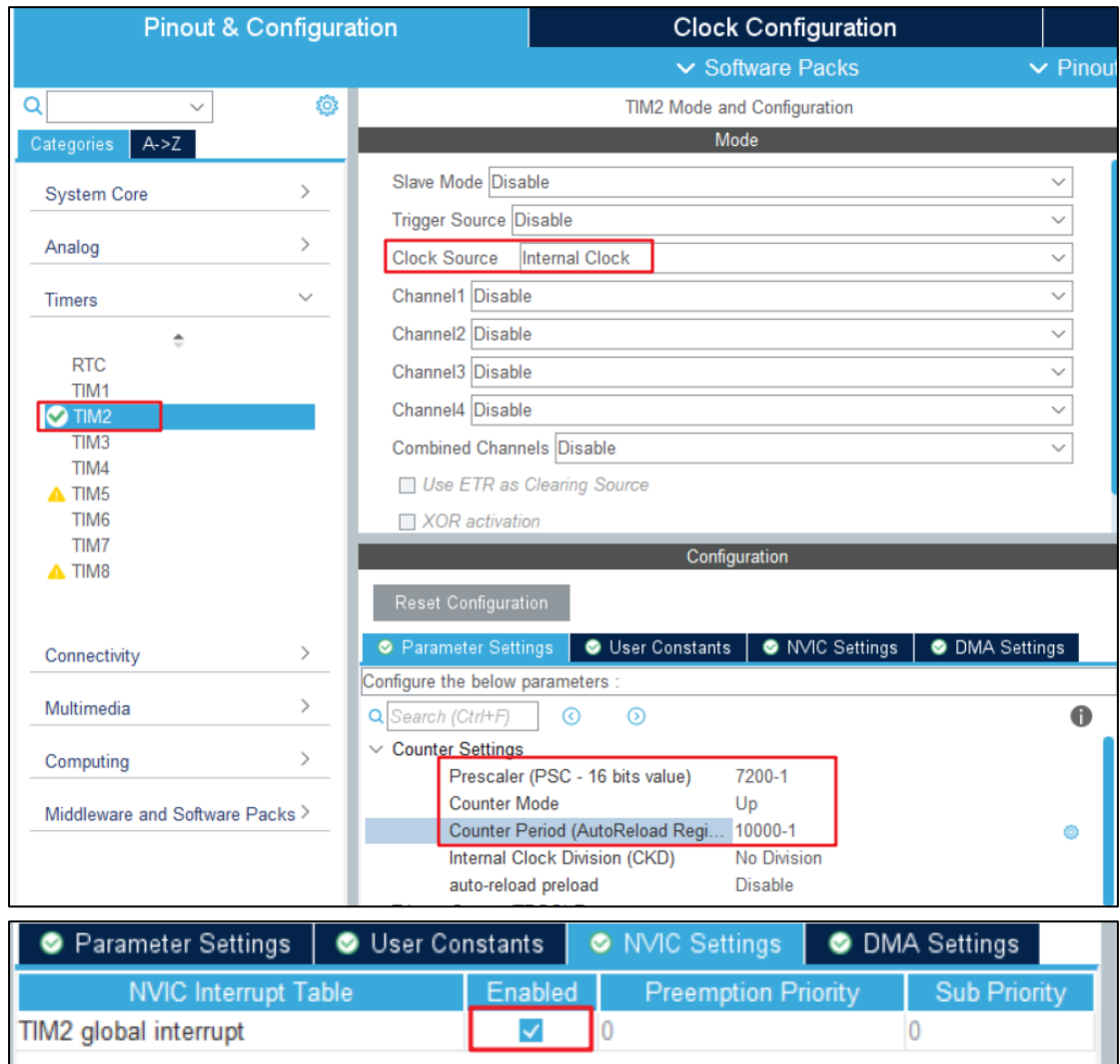
6.2 FreeRTOS 中断管理实验（掌握）

6.2.1 实验目标

学会 FreeRTOS 中断管理：

- 设置管理的优先级范围：5~15。
- 使用两个定时器，一个优先级为 4，一个优先级为 6。
- 两个定时器每 1s，打印一段字符串，当关中断时，停止打印，开中断时持续打印。

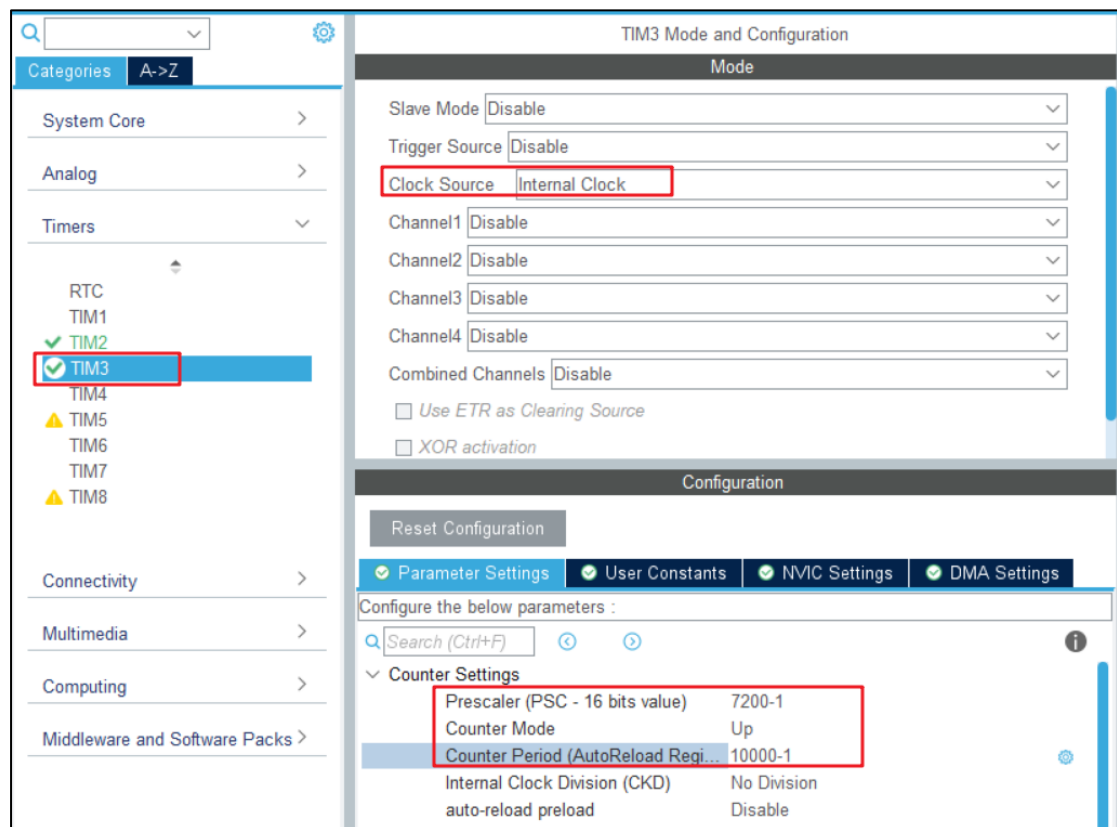
6.2.2 添加定时器



The screenshot shows the STM32CubeMX Pinout & Configuration window. The left sidebar shows the 'Timers' category with TIM2 selected. The main area shows the 'TIM2 Mode and Configuration' settings. The 'Clock Source' is set to 'Internal Clock'. The 'Counter Settings' section shows the 'Prescaler (PSC - 16 bits value)' set to 7200-1, 'Counter Mode' set to 'Up', and 'Counter Period (AutoReload Regi...' set to 10000-1. The 'NVIC Interrupt Table' at the bottom shows the 'TIM2 global interrupt' is enabled.

Parameter	Value
Slave Mode	Disable
Trigger Source	Disable
Clock Source	Internal Clock
Channel1	Disable
Channel2	Disable
Channel3	Disable
Channel4	Disable
Combined Channels	Disable
Use ETR as Clearing Source	<input type="checkbox"/>
XOR activation	<input type="checkbox"/>
Prescaler (PSC - 16 bits value)	7200-1
Counter Mode	Up
Counter Period (AutoReload Regi...)	10000-1
Internal Clock Division (CKD)	No Division
auto-reload preload	Disable

NVIC Interrupt Table	Enabled	Preemption Priority	Sub Priority
TIM2 global interrupt	<input checked="" type="checkbox"/>	0	0



TIM3 Mode and Configuration

Mode

- Slave Mode: Disable
- Trigger Source: Disable
- Clock Source: Internal Clock**
- Channel1: Disable
- Channel2: Disable
- Channel3: Disable
- Channel4: Disable
- Combined Channels: Disable
- ☐ Use ETR as Clearing Source
- ☐ XOR activation

Configuration

Reset Configuration

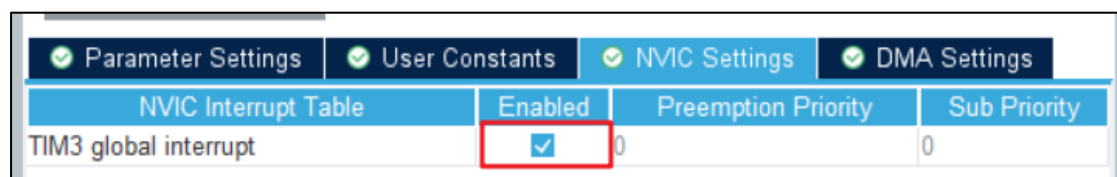
☒ Parameter Settings ☒ User Constants ☒ NVIC Settings ☒ DMA Settings

Configure the below parameters :

Search (Ctrl+F)

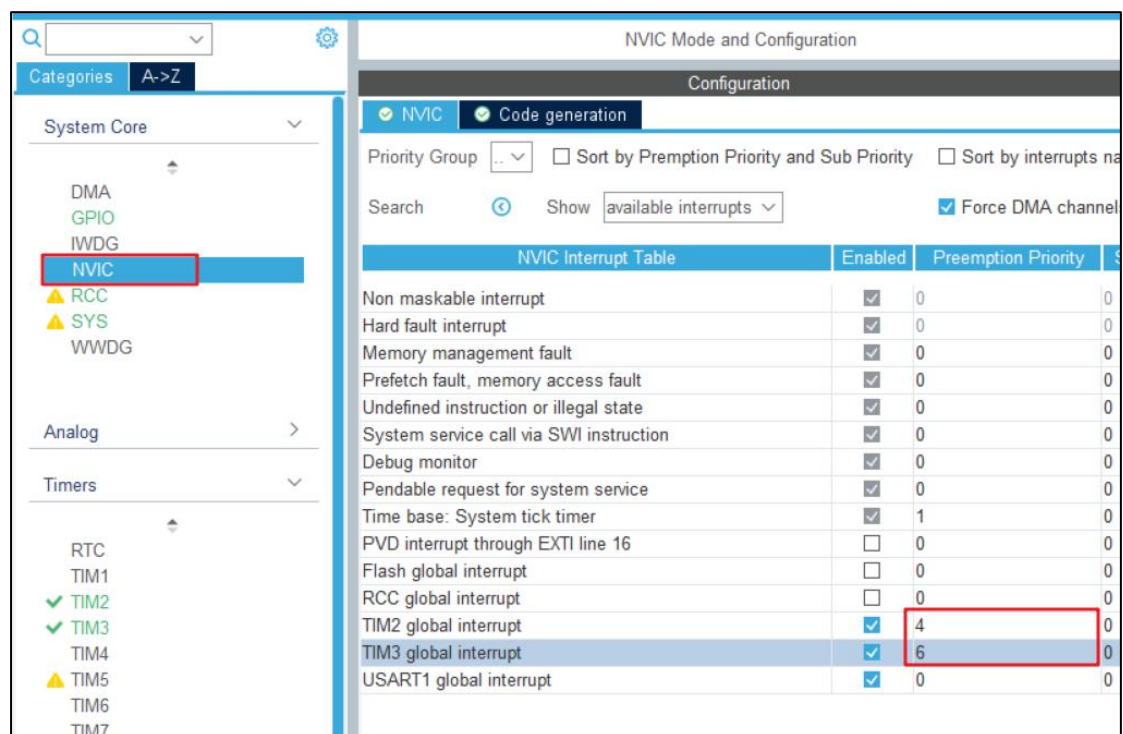
Counter Settings

- Prescaler (PSC - 16 bits value): 7200-1
- Counter Mode: Up
- Counter Period (AutoReload Regi...): 10000-1**
- Internal Clock Division (CKD): No Division
- auto-reload preload: Disable



☒ Parameter Settings ☒ User Constants ☒ NVIC Settings ☒ DMA Settings

NVIC Interrupt Table	Enabled	Preemption Priority	Sub Priority
TIM3 global interrupt	<input checked="" type="checkbox"/>	0	0



NVIC Mode and Configuration

Configuration

☒ NVIC ☐ Code generation

Priority Group: ... ☐ Sort by Preemption Priority and Sub Priority ☐ Sort by interrupts na

Search: ... Show: available interrupts ☒ Force DMA channel

NVIC Interrupt Table	Enabled	Preemption Priority	Sub Priority
Non maskable interrupt	<input checked="" type="checkbox"/>	0	0
Hard fault interrupt	<input checked="" type="checkbox"/>	0	0
Memory management fault	<input checked="" type="checkbox"/>	0	0
Prefetch fault, memory access fault	<input checked="" type="checkbox"/>	0	0
Undefined instruction or illegal state	<input checked="" type="checkbox"/>	0	0
System service call via SWI instruction	<input checked="" type="checkbox"/>	0	0
Debug monitor	<input checked="" type="checkbox"/>	0	0
Pendable request for system service	<input checked="" type="checkbox"/>	0	0
Time base: System tick timer	<input checked="" type="checkbox"/>	1	0
PVD interrupt through EXTI line 16	<input type="checkbox"/>	0	0
Flash global interrupt	<input type="checkbox"/>	0	0
RCC global interrupt	<input type="checkbox"/>	0	0
TIM2 global interrupt	<input checked="" type="checkbox"/>	4	0
TIM3 global interrupt	<input checked="" type="checkbox"/>	6	0
USART1 global interrupt	<input checked="" type="checkbox"/>	0	0

添加完定时器，重新注释掉 stm32f1xx_it.c 的 SVC_Handler 和 PendSV_Handler 函数。

6.2.3 main.c 代码清单

```
/* USER CODE BEGIN 0 */
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    if (htim->Instance == TIM2)
    {
        printf("TIM2 优先级为 4,运行中...\r\n");
    }
    else if(htim->Instance == TIM3)
    {
        printf("TIM3 优先级为 6,运行中...\r\n");
    }
}
/* USER CODE END 0 */

/* USER CODE BEGIN 2 */
HAL_TIM_Base_Start_IT(&htim2);
HAL_TIM_Base_Start_IT(&htim3);
/* USER CODE END 2 */
```

6.2.4 FreeRTOSConfig.h 代码清单

```
/* 中断嵌套行为配置 */
#ifdef __NVIC_PRIO_BITS
    #define configPRIO_BITS __NVIC_PRIO_BITS
#else
    #define configPRIO_BITS 4
#endif

#define
configLIBRARY_LOWEST_INTERRUPT_PRIORITY          15          /* 中
断最低优先级 */
#define
configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY      5          /*
FreeRTOS 可管理的最高中断优先级 */
#define
configKERNEL_INTERRUPT_PRIORITY                  ( configLIBRARY_LOWEST_I
NTERRUPT_PRIORITY << (8 - configPRIO_BITS) )
#define
configMAX_SYSCALL_INTERRUPT_PRIORITY              ( configLIBRARY_MAX_SYSC
ALL_INTERRUPT_PRIORITY << (8 - configPRIO_BITS) )
```

更多 [Java](#) - [大数据](#) - [前端](#) - [python](#) 人工智能资料下载，可百度访问：尚硅谷官网

```
#define
configMAX_API_CALL_INTERRUPT_PRIORITY          configMAX_SYSCALL_INTERRUPT_PRIORITY
```

6.2.5 freertos_demo.c 代码清单

1) 任务配置

```
/* 启动任务函数 */
#define START_TASK_PRIORITY 1
#define START_TASK_STACK_DEPTH 128
TaskHandle_t start_task_handler;
void Start_Task(void *pvParameters);

/* Task1 任务 配置 */
#define TASK1_PRIORITY 2
#define TASK1_STACK_DEPTH 128
TaskHandle_t task1_handler;
void Task1(void *pvParameters);
```

2) 入口函数

```
/**
 * @description: FreeRTOS 入口函数：创建任务函数并开始调度
 * @return {*}
 */
void FreeRTOS_Start(void)
{
    xTaskCreate((TaskFunction_t)Start_Task,
                (char *)"Start_Task",
                (configSTACK_DEPTH_TYPE)START_TASK_STACK_DEPTH,
                (void *)NULL,
                (UBaseType_t)START_TASK_PRIORITY,
                (TaskHandle_t *)&start_task_handler);
    vTaskStartScheduler();
}
```

3) 初始任务函数

```
void Start_Task(void *pvParameters)
{
    taskENTER_CRITICAL(); /* 进入临界区 */
    xTaskCreate((TaskFunction_t)Task1,
                (char *)"Task1",
                (configSTACK_DEPTH_TYPE)TASK1_STACK_DEPTH,
                (void *)NULL,
```

```
        (UBaseType_t)TASK1_PRIORITY,  
        (TaskHandle_t *)&task1_handler);  
  
    vTaskDelete(NULL);  
    taskEXIT_CRITICAL(); /* 退出临界区 */  
}
```

4) task1 任务函数

```
/**  
 * @description: 开关中断  
 * @param {void *} pvParameters  
 * @return {*}  
 */  
void Task1(void *pvParameters)  
{  
    uint8_t count = 0;  
    uint32_t delay = 0;  
    while (1)  
    {  
        delay=500000;  
        if(++count == 5)  
        {  
            count = 0;  
            printf("执行关中断...\r\n");  
            portDISABLE_INTERRUPTS();  
            while(delay--);  
            printf("执行开中断...\r\n");  
            portENABLE_INTERRUPTS();  
        }  
        vTaskDelay(1000);  
    }  
}
```

第 7 章 FreeRTOS 时间片调度

7.1 时间片调度简介（熟悉）

在 FreeRTOS 中，同等优先级的任务会轮流分享相同的 CPU 时间，这个时间被称为时间片。在这里，一个时间片的长度等同于 SysTick 中断的周期。

7.2 时间片调度实验演示（掌握）

7.2.1 实验目标

理解 FreeRTOS 的时间片调度：

- start_task：用来创建其他的 2 个任务。
- task1：通过串口打印 task1 的运行次数，设置任务优先级为 2。
- task2：通过串口打印 task2 的运行次数，设置任务优先级为 2。

为了更好地观察现象，滴答定时器的中断频率设置为 50ms 中断一次（一个时间片）。

7.2.2 FreeRTOSConfig.h 代码清单

```
#define configUSE_TIME_SLICING          1
#define configUSE_PREEMPTION            1
#define configTICK_RATE_HZ              ( ( TickType_t ) 20 )
```

7.2.3 freertos_demo.c 代码清单

1) 任务配置

```
/* 启动任务函数 */
#define START_TASK_PRIORITY 1
#define START_TASK_STACK_DEPTH 128
TaskHandle_t start_task_handler;
void Start_Task(void *pvParameters);

/* Task1 任务 配置 */
#define TASK1_PRIORITY 2
#define TASK1_STACK_DEPTH 128
TaskHandle_t task1_handler;
void Task1(void *pvParameters);

/* Task2 任务 配置 */
#define TASK2_PRIORITY 2
#define TASK2_STACK_DEPTH 128
TaskHandle_t task2_handler;
void Task2(void *pvParameters);
```

2) 添加延时函数

```
/**
 * @description: for 循环实现延时函数
 * @param {uint32_t} ms
 * @return {*}
 */
```

更多 [Java](#) - [大数据](#) - [前端](#) - [python](#) 人工智能资料下载，可百度访问：[尚硅谷官网](#)

```
*/  
void for_delay_ms(uint32_t ms)  
{  
    uint32_t Delay = ms * 72000/9; /* 72M 时钟频率, 9 是 PLL 倍频 */  
    do  
    {  
        __NOP(); /* 空指令 (NOP) 来占用 CPU 时间 */  
    }  
    while (Delay --);  
}
```

3) 入口函数

```
/**  
 * @description: FreeRTOS 入口函数: 创建任务函数并开始调度  
 * @return {*}  
 */  
void FreeRTOS_Start(void)  
{  
    xTaskCreate((TaskFunction_t)Start_Task,  
                (char *) "Start_Task",  
                (configSTACK_DEPTH_TYPE)START_TASK_STACK_DEPTH,  
                (void *)NULL,  
                (UBaseType_t)START_TASK_PRIORITY,  
                (TaskHandle_t *)&start_task_handler);  
    vTaskStartScheduler();  
}
```

4) 初始任务函数

```
void Start_Task( void * pvParameters )  
{  
    taskENTER_CRITICAL(); /* 进入临界区 */  
    xTaskCreate((TaskFunction_t ) Task1,  
                (char * ) "Task1",  
                (configSTACK_DEPTH_TYPE ) TASK1_STACK_DEPTH,  
                (void * ) NULL,  
                (UBaseType_t ) TASK1_PRIORITY,  
                (TaskHandle_t * ) &task1_handler );  
  
    xTaskCreate((TaskFunction_t ) Task2,  
                (char * ) "Task2",  
                (configSTACK_DEPTH_TYPE ) TASK2_STACK_DEPTH,  
                (void * ) NULL,
```



```
        (UBaseType_t          )    TASK2_PRIORITY,  
        (TaskHandle_t *       )    &task2_handler );  
  
vTaskDelete(NULL);  
taskEXIT_CRITICAL();           /* 退出临界区 */  
}
```

5) task1 任务函数

```
/**  
 * @description: 打印任务 1 执行次数  
 * @param {void *} pvParameters  
 * @return {*}  
 */  
void Task1(void * pvParameters)  
{  
    uint16_t task1_count=0;  
    while(1)  
    {  
        /* 临界区避免 printf 执行一半被打断 */  
        taskENTER_CRITICAL();  
        printf("task1 运行次数=%d...\r\n",++task1_count);  
        for_delay_ms(10);  
        taskEXIT_CRITICAL();  
    }  
}
```

6) task2 任务函数

```
/**  
 * @description: 打印任务 2 执行次数  
 * @param {void *} pvParameters  
 * @return {*}  
 */  
void Task2(void * pvParameters)  
{  
    uint16_t task2_count=0;  
    while(1)  
    {  
        /* 临界区避免 printf 执行一半被打断 */  
        taskENTER_CRITICAL();  
        printf("task2 运行次数=%d...\r\n",++task2_count);  
        for_delay_ms(10);  
    }  
}
```

```
        taskEXIT_CRITICAL();  
    }  
}
```

第 8 章 FreeRTOS 任务相关 API 函数

8.1 FreeRTOS 任务相关 API 函数介绍（熟悉）

任务相关的 API 主要如下：

函数	描述
uxTaskPriorityGet()	获取任务优先级
vTaskPrioritySet()	设置任务优先级
uxTaskGetNumberOfTasks()	获取系统中任务的数量
uxTaskGetSystemState()	获取所有任务状态信息
vTaskGetInfo()	获取指定单个的任务信息
xTaskGetCurrentTaskHandle()	获取当前任务的任务句柄
xTaskGetHandle()	根据任务名获取该任务的任务句柄
uxTaskGetStackHighWaterMark()	获取任务的任务栈历史剩余最小值
eTaskGetState()	获取任务状态
vTaskList()	以“表格”形式获取所有任务的信息
vTaskGetRunTimeStats()	获取任务的运行时间

官网：<https://freertos.org/zh-cn-cmn-s/a00106.html>

8.2 任务状态查询 API 函数实验（掌握）

8.2.1 实验目标

学会使用 FreeRTOS 任务状态查询相关 API 函数：

- start_task：用来创建其他的 2 个任务。
- task1：LED1 每 500ms 闪烁一次，提示程序正在运行。
- task2：用于展示任务状态查询相关 API 函数的使用。

8.2.2 FreeRTOSConfig.h 代码清单

```
#define INCLUDE_xTaskGetSchedulerState 1  
#define configUSE_TRACE_FACILITY 1
```

```
#define configUSE_STATS_FORMATTING_FUNCTIONS 1
#define INCLUDE_xTaskGetHandle 1
#define INCLUDE_uxTaskGetStackHighWaterMark 1
#define INCLUDE_eTaskGetState 1
```

8.2.3 freertos_demo.c 代码清单

1) 任务配置

```
/* 启动任务函数 */
#define START_TASK_PRIORITY 1
#define START_TASK_STACK_DEPTH 128
TaskHandle_t start_task_handler;
void Start_Task(void *pvParameters);

/* Task1 任务 配置 */
#define TASK1_PRIORITY 2
#define TASK1_STACK_DEPTH 128
TaskHandle_t task1_handler;
void Task1(void *pvParameters);

/* Task2 任务 配置 */
#define TASK2_PRIORITY 3
#define TASK2_STACK_DEPTH 128
// #define TASK2_STACK_DEPTH 105
TaskHandle_t task2_handler;
void Task2(void *pvParameters);
```

2) 入口函数

```
/**
 * @description: FreeRTOS 入口函数：创建任务函数并开始调度
 * @return {*}
 */
void FreeRTOS_Start(void)
{
    xTaskCreate((TaskFunction_t)Start_Task,
                (char *)"Start_Task",
                (configSTACK_DEPTH_TYPE)START_TASK_STACK_DEPTH,
                (void *)NULL,
                (UBaseType_t)START_TASK_PRIORITY,
                (TaskHandle_t *)&start_task_handler);
    vTaskStartScheduler();
}
```

3) 初始函数

```
void Start_Task(void *pvParameters)
{
    taskENTER_CRITICAL(); /* 进入临界区 */
    xTaskCreate((TaskFunction_t)Task1,
                (char *)"Task1",
                (configSTACK_DEPTH_TYPE)TASK1_STACK_DEPTH,
                (void *)NULL,
                (UBaseType_t)TASK1_PRIORITY,
                (TaskHandle_t *)&task1_handler);

    xTaskCreate((TaskFunction_t)Task2,
                (char *)"Task2",
                (configSTACK_DEPTH_TYPE)TASK2_STACK_DEPTH,
                (void *)NULL,
                (UBaseType_t)TASK2_PRIORITY,
                (TaskHandle_t *)&task2_handler);

    vTaskDelete(NULL);
    taskEXIT_CRITICAL(); /* 退出临界区 */
}
```

4) task1 任务函数

```
/**
 * @description: LED1 每 500ms 翻转一次
 * @param {void *} pvParameters
 * @return {*}
 */
void Task1(void *pvParameters)
{
    while (1)
    {
        LED_Toggle(LED1_Pin);
        vTaskDelay(500);
    }
}
```

5) task2 任务函数

```
/**
 * @description: 查询任务信息
 * @param {void *} pvParameters
 * @return {*}
 */
```

```
*/
char task_info[500];
void Task2(void *pvParameters)
{
    UBaseType_t task_priority = 0;
    UBaseType_t task_num = 0;
    UBaseType_t task_num2 = 0;
    TaskStatus_t task_status[4] = 0;
    TaskStatus_t task_status2[1] = 0;
    TaskHandle_t task_handle = 0;
    UBaseType_t task_stack_remain_min = 0;
    eTaskState task_state = 0;

    /* 查询任务优先级 */
    task_priority = uxTaskPriorityGet(task1_handler);
    printf("task1 任务优先级=%d....\r\n", task_priority);
    task_priority = uxTaskPriorityGet(task2_handler);
    printf("task2 任务优先级=%d....\r\n", task_priority);

    /* 设置任务优先级 */
    vTaskPrioritySet(task1_handler, 4);
    task_priority = uxTaskPriorityGet(task1_handler);
    printf("task1 任务优先级=%d....\r\n", task_priority);

    /* 查询任务数量：包含启动调度器时底层启动的任务 */
    task_num = uxTaskGetNumberOfTasks();
    printf("任务数量=%d....\r\n", task_num);

    /* 获取系统状态 */
    task_num2 = uxTaskGetSystemState(task_status, task_num, NULL);
    printf("任务名\t任务编号\t任务优先级\r\n");
    for (uint8_t i = 0; i < task_num2; i++)
    {
        printf("%s\t%d\t%d\r\n",
            task_status[i].pcTaskName,
            task_status[i].xTaskNumber,
            task_status[i].uxCurrentPriority);
    }

    /* 获取单个任务信息 */
    vTaskGetInfo(task1_handler,
```

```
        task_status2,  
        pdTRUE,  
        eInvalid);  
printf("任务名: %s\r\n", task_status2->pcTaskName);  
printf("任务编号: %d\r\n", task_status2->xTaskNumber);  
printf("任务优先级: %d\r\n", task_status2->uxCurrentPriority);  
printf("任务状态: %d\r\n", task_status2->eCurrentState);  
  
/* 根据任务名获取任务句柄 */  
task_handle = xTaskGetHandle("Task1");  
printf("获取 Task1 任务句柄: %#x\r\n", task_handle);  
printf("Task1 任务句柄: %#x\r\n", task1_handler);  
  
/* 获取指定任务的任务栈历史最小剩余值 */  
task_stack_remain_min =  
uxTaskGetStackHighWaterMark( task2_handler );  
printf("task2 任务栈历史最小值=%d\r\n", task_stack_remain_min);  
  
/* 获取指定任务的状态 */  
task_state = eTaskGetState( task2_handler );  
printf("task2 当前任务状态=%d\r\n", task_state);  
  
/* 以表格形式获取系统中任务的信息 */  
vTaskList(task_info);  
printf("%s\r\n", task_info);  
while (1)  
{  
    vTaskDelay(100);  
}
```

8.3 任务时间统计 API 函数实验（掌握）

8.3.1 实验目标

学会使用 FreeRTOS 任务运行时间统计相关 API 函数：

- start_task：用来创建其他的 2 个任务。
- task1：LED0 每 500ms 闪烁一次，提示程序正在运行。
- task2：用于展示任务运行时间统计相关 API 函数的使用。

8.3.2 tim.c 代码清单

```
void MX_TIM2_Init(void)
{
    .....
    htim2.Instance = TIM2;
    htim2.Init.Prescaler = 72-1;
    htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
    htim2.Init.Period = 10-1;
    htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
    htim2.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
    .....
}
```

8.3.3 main.c 代码清单

```
/* USER CODE BEGIN 0 */
volatile unsigned long ulHighFrequencyTimerTicks;
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    if (htim->Instance == TIM2)
    {
        ulHighFrequencyTimerTicks++;
    }
}
/* USER CODE END 0 */
```

8.3.4 FreeRTOSConfig.h 代码清单

```
/* 运行时间和任务状态统计相关定义 */
#define configGENERATE_RUN_TIME_STATS 1 /* 1: 使能任务运行时间统计功能, 默认: 0 */
#if configGENERATE_RUN_TIME_STATS
extern volatile unsigned long ulHighFrequencyTimerTicks;
#define portCONFIGURE_TIMER_FOR_RUN_TIME_STATS()
( ulHighFrequencyTimerTicks = 0UL )
#define portGET_RUN_TIME_COUNTER_VALUE() ulHighFrequencyTimerTicks
#endif
#define configUSE_TRACE_FACILITY 1
#define configUSE_STATS_FORMATTING_FUNCTIONS 1
```

➤ portCONFIGURE_TIMER_FOR_RUNTIME_STATE(): 用于初始化用于配置任务运行时间统计的时基定时器。它的时间精度需要比 tick 中断具有更高的精度, 建议 10 到 100

倍。

➤ portGET_RUN_TIME_COUNTER_VALUE(): 返回该定时器的计数值，即当前已运行的时间。

参数说明: <https://freertos.org/zh-cn-cmn-s/rto-s-run-time-stats.html>

8.3.5 freertos_demo.c 代码清单

1) 任务配置

```
/* 启动任务函数 */
#define START_TASK_PRIORITY 1
#define START_TASK_STACK_DEPTH 128
TaskHandle_t start_task_handler;
void Start_Task(void *pvParameters);

/* Task1 任务 配置 */
#define TASK1_PRIORITY 2
#define TASK1_STACK_DEPTH 128
TaskHandle_t task1_handler;
void Task1(void *pvParameters);

/* Task2 任务 配置 */
#define TASK2_PRIORITY 3
#define TASK2_STACK_DEPTH 128
TaskHandle_t task2_handler;
void Task2(void *pvParameters);
```

2) 入口函数

```
/**
 * @description: FreeRTOS 入口函数：创建任务函数并开始调度
 * @return {*}
 */
void FreeRTOS_Start(void)
{
    xTaskCreate((TaskFunction_t)Start_Task,
                (char *)"Start_Task",
                (configSTACK_DEPTH_TYPE)START_TASK_STACK_DEPTH,
                (void *)NULL,
                (UBaseType_t)START_TASK_PRIORITY,
                (TaskHandle_t *)&start_task_handler);
    vTaskStartScheduler();
}
```

更多 Java - 大数据 - 前端 - python 人工智能资料下载，可百度访问：尚硅谷官网

3) 初始任务函数

```
void Start_Task(void *pvParameters)
{
    taskENTER_CRITICAL(); /* 进入临界区 */
    xTaskCreate((TaskFunction_t)Task1,
                (char *)"Task1",
                (configSTACK_DEPTH_TYPE)TASK1_STACK_DEPTH,
                (void *)NULL,
                (UBaseType_t)TASK1_PRIORITY,
                (TaskHandle_t *)&task1_handler);
    xTaskCreate((TaskFunction_t)Task2,
                (char *)"Task2",
                (configSTACK_DEPTH_TYPE)TASK2_STACK_DEPTH,
                (void *)NULL,
                (UBaseType_t)TASK2_PRIORITY,
                (TaskHandle_t *)&task2_handler);

    vTaskDelete(NULL);
    taskEXIT_CRITICAL(); /* 退出临界区 */
}
```

4) task1 任务函数

```
/**
 * @description: LED1 每 500ms 翻转一次
 * @param {void *} pvParameters
 * @return {*}
 */
void Task1(void *pvParameters)
{
    while(1)
    {
        LED_Toggle(LED1_Pin);
        vTaskDelay(500);
    }
}
```

5) task2 任务函数

```
/**
 * @description: 按下 KEY1, 打印任务运行时间
 * @param {void *} pvParameters
 * @return {*}
 */
```

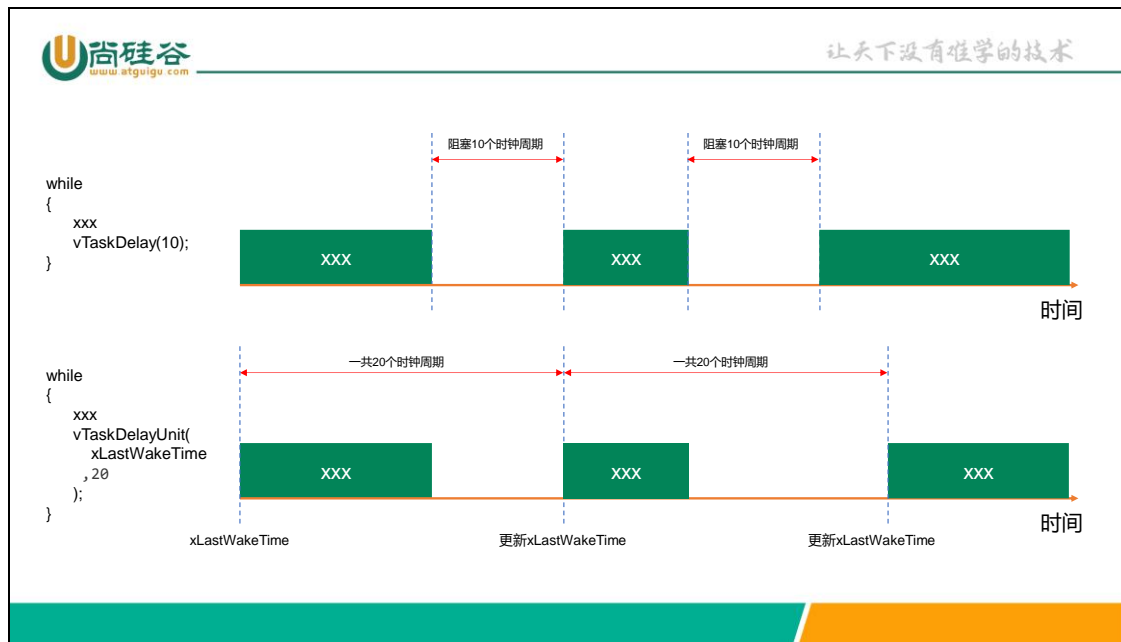
```
*/  
char task_stat[500];  
void Task2(void *pvParameters)  
{  
    uint8_t key = 0;  
    while(1)  
    {  
        key = Key_Detect();  
        if(key == KEY1_PRESS)  
        {  
            vTaskGetRunTimeStats(task_stat);  
            printf("%s\r\n", task_stat);  
        }  
        vTaskDelay(10);  
    }  
}
```

第 9 章 FreeRTOS 时间管理

9.1 延时函数介绍（了解）

- `vTaskDelay()`: 相对延时。从执行 `vTaskDelay()` 函数开始, 直到指定延时的时间结束。
- `xTaskDelayUntil()`: 绝对延时。将整个任务的运行周期视为一个整体, 适用于需要以固定频率定期执行的任务。

假设有一个定时器, 每隔 1 秒触发一次, 希望在每次触发时执行某个任务。如果使用 `vTaskDelay` 来实现, 那么你能实现任务每秒执行一次, 而不能确保任务在每秒的开始时刻执行。但如果你使用 `xTaskDelayUntil`, 你可以指定任务在每秒的开始时刻执行, 即使任务执行的时间不同。



9.2 延时函数演示实验（掌握）

9.2.1 实验目标

学习 FreeRTOS 延时函数的使用，了解相对延时和绝对延时的区别：

- start_task：用来创建其他的 2 个任务。
- task1：用于展示相对延时函数 vTaskDelay () 的使用。
- task2：用于展示绝对延时函数 vTaskDelayUntil() 的使用。

为了观察两个延时函数的区别，将使用 LED1 和 LED2 的翻转波形来表示。

9.2.2 freertos_demo.c 代码清单

1) 任务配置

```
/* 启动任务函数 */
#define START_TASK_PRIORITY 1
#define START_TASK_STACK_DEPTH 128
TaskHandle_t start_task_handler;
void Start_Task(void *pvParameters);

/* Task1 任务 配置 */
#define TASK1_PRIORITY 2
#define TASK1_STACK_DEPTH 128
TaskHandle_t task1_handler;
void Task1(void *pvParameters);
```

```
/* Task2 任务 配置 */
#define TASK2_PRIORITY 3
#define TASK2_STACK_DEPTH 128
TaskHandle_t task2_handler;
void Task2(void *pvParameters);
```

2) 延时函数

```
/**
 * @description: for 循环实现延时函数
 * @param {uint32_t} ms
 * @return {*}
 */
void for_delay_ms(uint32_t ms)
{
    uint32_t Delay = ms * 72000 / 9; /* 72M 时钟频率, 9 是 PLL 倍频 */
    do
    {
        __NOP(); /* 空指令 (NOP) 来占用 CPU 时间 */
    } while (Delay--);
}
```

3) 入口函数

```
/**
 * @description: FreeRTOS 入口函数: 创建任务函数并开始调度
 * @return {*}
 */
void FreeRTOS_Start(void)
{
    xTaskCreate((TaskFunction_t)Start_Task,
                (char *)"Start_Task",
                (configSTACK_DEPTH_TYPE)START_TASK_STACK_DEPTH,
                (void *)NULL,
                (UBaseType_t)START_TASK_PRIORITY,
                (TaskHandle_t *)&start_task_handler);
    vTaskStartScheduler();
}
```

4) 初始任务函数

```
void Start_Task(void *pvParameters)
{
    taskENTER_CRITICAL(); /* 进入临界区 */
    xTaskCreate((TaskFunction_t)Task1,
                (char *)"Task1",
```

```
        (configSTACK_DEPTH_TYPE)TASK1_STACK_DEPTH,  
        (void *)NULL,  
        (UBaseType_t)TASK1_PRIORITY,  
        (TaskHandle_t *)&task1_handler);  
  
    xTaskCreate((TaskFunction_t)Task2,  
                (char *)"Task2",  
                (configSTACK_DEPTH_TYPE)TASK2_STACK_DEPTH,  
                (void *)NULL,  
                (UBaseType_t)TASK2_PRIORITY,  
                (TaskHandle_t *)&task2_handler);  
    vTaskDelete(NULL);  
    taskEXIT_CRITICAL(); /* 退出临界区 */  
}
```

5) task1 任务函数

```
/**  
 * @description: LED1 每 500ms 翻转一次  
 * @param {void *} pvParameters  
 * @return {*}  
 */  
void Task1(void *pvParameters)  
{  
    while (1)  
    {  
        LED_Toggle(LED1_Pin);  
        for_delay_ms(20);  
        vTaskDelay(500);  
    }  
}
```

6) task2 任务函数

```
/**  
 * @description: LED2 每 500ms 翻转一次  
 * @param {void *} pvParameters  
 * @return {*}  
 */  
void Task2(void *pvParameters)  
{  
    TickType_t xLastWakeTime;  
    xLastWakeTime = xTaskGetTickCount();
```

```
while (1)
{
    LED_Toggle(LED2_Pin);
    for_delay_ms(20);
    vTaskDelayUntil(&xLastWakeTime,500);
}
```

第 10 章 FreeRTOS 消息队列

10.1 队列简介（了解）

FreeRTOS 中的队列是一种用于任务之间通信的机制，它允许一个任务发送消息给另一个任务。队列是线程安全的数据结构，任务可以通过队列在彼此之间传递数据。有以下关键特点：

- FIFO 顺序：队列采用先进先出 (FIFO) 的顺序，即先发送的消息会被先接收。
- 线程安全：队列操作是原子的，确保在多任务环境中的数据完整性。
- 阻塞和非阻塞操作：任务可以通过阻塞或非阻塞的方式发送和接收消息。如果队列满了或者为空，任务可以选择等待直到有空间或者数据可用，或者立即返回。
- 优先级继承：FreeRTOS 支持基于优先级的消息传递，确保高优先级任务在队列操作期间不会被低优先级任务阻塞。
- 可变长度项：队列中的项可以是不同长度的数据块，而不是固定大小。

使用队列，任务可以通过发送消息来共享信息，从而更好地协调和同步系统中的不同部分。

10.2 队列相关 API 函数介绍（熟悉）

10.2.1 创建队列

函数	描述
<code>xQueueCreate()</code>	动态方式创建队列
<code>xQueueCreateStatic()</code>	静态方式创建队列

动态创建队列时，FreeRTOS 会在运行时从其内置的堆中为队列分配所需的内存空间。这种方式更加灵活，允许系统根据需要动态调整内存。

相反，静态创建队列要求用户在编译时手动为队列分配内存，而不依赖于 FreeRTOS 的

堆管理。这使得内存的分配在编写代码时就能确定，因此在资源受限或对内存使用有严格要求的嵌入式系统中可能更为合适。

总体而言，动态创建提供了更大的灵活性，但可能会增加堆管理的复杂性。静态创建则更为直观，适用于在编译时就能确定内存分配的情况。选择使用哪种方式通常取决于系统的需求和设计考虑。

10.2.2 往队列写入消息

函数	描述
<code>xQueueSend()</code>	往队列的尾部写入消息
<code>xQueueSendToBack()</code>	同 <code>xQueueSend()</code>
<code>xQueueSendToFront()</code>	往队列的头部写入消息
<code>xQueueOverwrite()</code>	覆写队列消息（只用于队列长度为 1 的情况）
<code>xQueueSendFromISR()</code>	在中断中往队列的尾部写入消息
<code>xQueueSendToBackFromISR()</code>	同 <code>xQueueSendFromISR()</code>
<code>xQueueSendToFrontFromISR()</code>	在中断中往队列的头部写入消息
<code>xQueueOverwriteFromISR()</code>	在中断中覆写队列消息（只用于队列长度为 1 的情况）

10.2.3 从队列读取消息

函数	描述
<code>xQueueReceive()</code>	从队列头部读取消息，并删除消息
<code>xQueuePeek()</code>	从队列头部读取消息
<code>xQueueReceiveFromISR()</code>	在中断中从队列头部读取消息，并删除消息
<code>xQueuePeekFromISR()</code>	在中断中从队列头部读取消息

10.3 队列操作实验（掌握）

10.3.1 实验目标

学习使用 FreeRTOS 的队列相关函数，包括创建队列、入队和出队操作：

➤ `start_task`：用来创建其他的 3 个任务。

➤ `task1`：当按键 `key1` 或 `key2` 按下，将键值拷贝到队列 `queue1`（入队）；当按键 `key3` 按下，将传输大数据，这里拷贝大数据的地址到队列 `big_queue` 中。

- task2: 读取队列 queue1 中的消息（出队），打印出接收到的键值。
- task3: 从队列 big_queue 读取大数据地址，通过地址访问大数据。

10.3.2 freertos_demo.c 代码清单

1) 引入队列头文件

```
#include "queue.h"
```

2) 任务配置

```
/* 启动任务函数 */
#define START_TASK_PRIORITY 1
#define START_TASK_STACK_DEPTH 128
TaskHandle_t start_task_handler;
void Start_Task(void *pvParameters);

/* Task1 任务 配置 */
#define TASK1_PRIORITY 2
#define TASK1_STACK_DEPTH 128
TaskHandle_t task1_handler;
void Task1(void *pvParameters);

/* Task2 任务 配置 */
#define TASK2_PRIORITY 3
#define TASK2_STACK_DEPTH 128
TaskHandle_t task2_handler;
void Task2(void *pvParameters);

/* Task3 任务 配置 */
#define TASK3_PRIORITY 4
#define TASK3_STACK_DEPTH 128
TaskHandle_t task3_handler;
void Task3(void *pvParameters);
```

3) 入口函数

```
QueueHandle_t queue1;    /* 小数据句柄 */
QueueHandle_t big_queue; /* 大数据句柄 */
char buff[100] = {"大大大 fdahjk324hjkfhjksdahjk#$@!@#jfaskdfhjka"};
/**
 * @description: FreeRTOS 入口函数：创建任务函数并开始调度
 * @return {*}
 */
void FreeRTOS_Start(void)
```



```
{
    /* 创建 queue1 队列 */
    queue1 = xQueueCreate(2, sizeof(uint8_t));
    if (queue1 != NULL)
    {
        printf("queue1 队列创建成功\r\n");
    }
    else
    {
        printf("queue1 队列创建失败\r\n");
    }
    /* 创建 big_queue 队列 */
    big_queue = xQueueCreate(1, sizeof(char *));
    if (big_queue != NULL)
    {
        printf("big_queue 队列创建成功\r\n");
    }
    else
    {
        printf("big_queue 队列创建失败\r\n");
    }

    xTaskCreate((TaskFunction_t)Start_Task,
                (char *)"Start_Task",
                (configSTACK_DEPTH_TYPE)START_TASK_STACK_DEPTH,
                (void *)NULL,
                (UBaseType_t)START_TASK_PRIORITY,
                (TaskHandle_t *)&start_task_handler);
    vTaskStartScheduler();
}
```

4) 初始任务函数

```
void Start_Task(void *pvParameters)
{
    taskENTER_CRITICAL(); /* 进入临界区 */
    xTaskCreate((TaskFunction_t)Task1,
                (char *)"Task1",
                (configSTACK_DEPTH_TYPE)TASK1_STACK_DEPTH,
                (void *)NULL,
                (UBaseType_t)TASK1_PRIORITY,
                (TaskHandle_t *)&task1_handler);
}
```

```
xTaskCreate((TaskFunction_t)Task2,
            (char *)"Task2",
            (configSTACK_DEPTH_TYPE)TASK2_STACK_DEPTH,
            (void *)NULL,
            (UBaseType_t)TASK2_PRIORITY,
            (TaskHandle_t *)&task2_handler);

xTaskCreate((TaskFunction_t)Task3,
            (char *)"Task2",
            (configSTACK_DEPTH_TYPE)TASK3_STACK_DEPTH,
            (void *)NULL,
            (UBaseType_t)TASK3_PRIORITY,
            (TaskHandle_t *)&task3_handler);

vTaskDelete(NULL);
taskEXIT_CRITICAL(); /* 退出临界区 */
}
```

5) task1 任务函数

```
/**
 * @description: 入队
 * @param {void *} pvParameters
 * @return {*}
 */
void Task1(void *pvParameters)
{
    uint8_t key = 0;
    char *buf;
    BaseType_t err = 0;
    buf = &buff[0];
    while (1)
    {
        key = Key_Detect();
        if (key == KEY1_PRESS || key == KEY2_PRESS)
        {
            err = xQueueSend(queue1, &key, portMAX_DELAY);
            if (err != pdTRUE)
            {
                printf("queue1 队列发送失败\r\n");
            }
        }
        else if (key == KEY3_PRESS)
        {

```

```
        err = xQueueSend(big_queue, &buf, portMAX_DELAY);
        if (err != pdTRUE)
        {
            printf("big_queue 队列发送失败\r\n");
        }
    }
    vTaskDelay(10);
}
```

6) task2 任务函数

```
/**
 * @description: 小数据出队
 * @param {void *} pvParameters
 * @return {*}
 */
void Task2(void *pvParameters)
{
    uint8_t key = 0;
    BaseType_t err = 0;
    while (1)
    {
        err = xQueueReceive(queue1, &key, portMAX_DELAY);
        if (err != pdTRUE)
        {
            printf("queue1 队列读取失败\r\n");
        }
        else
        {
            printf("queue1 读取队列成功, 数据: %d\r\n", key);
        }
    }
}
```

7) task3 任务函数

```
/**
 * @description: 大数据出队
 * @param {void *} pvParameters
 * @return {*}
 */
void Task3(void *pvParameters)
{
```

```
char *buf;
BaseType_t err = 0;
while (1)
{
    err = xQueueReceive(big_queue, &buf, portMAX_DELAY);
    if (err != pdTRUE)
    {
        printf("big_queue 队列读取失败\r\n");
    }
    else
    {
        printf("数据: %s\r\n", buf);
    }
}
```

第 11 章 信号量

11.1 信号量的简介（了解）

FreeRTOS 中的信号量是一种用于任务间同步和资源管理的机制。信号量可以是二进制的（只能取 0 或 1）也可以是计数型的（可以是任意正整数）。信号量的基本操作包括“获取”和“释放”。

比如动车上的卫生间，一个卫生间同时只能容纳一个人，由指示灯来表示是否有人在使用。当我们想使用卫生间的时候，有如下过程：

- （1）判断卫生间是否有人使用（判断信号量是否有资源）
- （2）卫生间空闲（信号量有资源），那么就可以直接进入卫生间（获取信号量成功）
- （3）卫生间使用中（信号量没有资源），那么这个人可以选择不上卫生间（获取信号量失败），也可以在门口等待（任务阻塞）

信号量与队列的区别如下：

信号量	队列
主要用于管理对共享资源的访问，确保在同一时刻只有一个任务可以访问共享资源	用于任务之间的数据通信，通过在任务之间传递消息，实现信息的传递和同步。
可以是二进制信号量（Binary Semaphore）或计数信号量（Counting Semaphore）	存储和传递消息的数据结构，任务可以发送消息到队列，也可以从队列接收消息。

适用于对资源的互斥访问,控制任务的执行顺序,或者限制同时访问某一资源的任务数量。

适用于在任务之间传递数据,实现解耦和通信。

11.2 二值信号量（熟悉）

二值信号量（Binary Semaphore）是一种特殊类型的信号量，它只有两个可能的值：0 和 1。这种信号量主要用于实现对共享资源的互斥访问或者任务之间的同步。

➤ 两个状态： 二值信号量只能处于两个状态之一，通常用 0 和 1 表示。当信号量的值为 0 时，表示资源不可用；当值为 1 时，表示资源可用。

➤ 互斥访问： 常用于控制对共享资源的互斥访问，确保在同一时刻只有一个任务可以访问共享资源。任务在访问资源之前会尝试获取信号量，成功则继续执行，失败则等待。

➤ 任务同步： 也可以用于任务之间的同步，例如一个任务等待另一个任务完成某个操作。

二值信号量相关函数：

函数	描述
xSemaphoreCreateBinary()	使用动态方式创建二值信号量
xSemaphoreCreateBinaryStatic()	使用静态方式创建二值信号量
xSemaphoreGive()	释放信号量
xSemaphoreGiveFromISR()	在中断中释放信号量
xSemaphoreTake()	获取信号量
xSemaphoreTakeFromISR()	在中断中获取信号量

11.3 二值信号量实验（掌握）

11.3.1 实验目标

学习使用 FreeRTOS 的二值信号量相关函数：

- start_task：用来创建其他的 2 个任务。
- task1：用于按键扫描，当检测到按键 KEY1 被按下时，释放二值信号量。
- task2：获取二值信号量，当成功获取后打印提示信息。

11.3.2 freertos_demo.c 代码清单

1) 引入信号量头文件

```
#include "semphr.h"
```

2) 任务配置

```
/* 启动任务函数 */
#define START_TASK_PRIORITY 1
#define START_TASK_STACK_DEPTH 128
TaskHandle_t start_task_handler;
void Start_Task(void *pvParameters);

/* Task1 任务 配置 */
#define TASK1_PRIORITY 2
#define TASK1_STACK_DEPTH 128
TaskHandle_t task1_handler;
void Task1(void *pvParameters);

/* Task2 任务 配置 */
#define TASK2_PRIORITY 3
#define TASK2_STACK_DEPTH 128
TaskHandle_t task2_handler;
void Task2(void *pvParameters);
```

3) 入口函数

```
QueueHandle_t semaphore_handle;
/**
 * @description: FreeRTOS 入口函数：创建任务函数并开始调度
 * @return {*}
 */
void FreeRTOS_Start(void)
{
    semaphore_handle = xSemaphoreCreateBinary();
    if (semaphore_handle != NULL)
    {
        printf("二值信号量创建成功\r\n");
    }
    xTaskCreate((TaskFunction_t)Start_Task,
                (char *)"Start_Task",
                (configSTACK_DEPTH_TYPE)START_TASK_STACK_DEPTH,
                (void *)NULL,
                (UBaseType_t)START_TASK_PRIORITY,
```

```
        (TaskHandle_t *)&start_task_handler);  
    vTaskStartScheduler();  
}
```

4) 初始任务函数

```
void Start_Task(void *pvParameters)  
{  
    taskENTER_CRITICAL(); /* 进入临界区 */  
    xTaskCreate((TaskFunction_t)Task1,  
                (char *)"Task1",  
                (configSTACK_DEPTH_TYPE)TASK1_STACK_DEPTH,  
                (void *)NULL,  
                (UBaseType_t)TASK1_PRIORITY,  
                (TaskHandle_t *)&task1_handler);  
  
    xTaskCreate((TaskFunction_t)Task2,  
                (char *)"Task2",  
                (configSTACK_DEPTH_TYPE)TASK2_STACK_DEPTH,  
                (void *)NULL,  
                (UBaseType_t)TASK2_PRIORITY,  
                (TaskHandle_t *)&task2_handler);  
    vTaskDelete(NULL);  
    taskEXIT_CRITICAL(); /* 退出临界区 */  
}
```

5) task1 任务函数

```
/**  
 * @description: 释放二值信号量  
 * @param {void *} pvParameters  
 * @return {*}  
 */  
void Task1(void *pvParameters)  
{  
    uint8_t key = 0;  
    BaseType_t err;  
    while (1)  
    {  
        key = Key_Detect();  
        if (key == KEY1_PRESS)  
        {  
            if (semaphore_handle != NULL)  
            {  

```

```
        err = xSemaphoreGive(semaphore_handle);
        if (err == pdPASS)
        {
            printf("信号量释放成功\r\n");
        }
        else
            printf("信号量释放失败\r\n");
    }
}
vTaskDelay(10);
}
```

6) task2 任务函数

```
/**
 * @description: 获取二值信号量
 * @param {void *} pvParameters
 * @return {*}
 */
void Task2(void *pvParameters)
{
    uint32_t i = 0;
    BaseType_t err;
    while (1)
    {
        /* 一直等待获取信号量 */
        err = xSemaphoreTake(semaphore_handle, portMAX_DELAY);
        if (err == pdTRUE)
        {
            printf("获取信号量成功\r\n");
        }
        else
        {
            printf("已超时%d\r\n", ++i);
        }
    }
}
```

11.4 计数型信号量（熟悉）

正如二进制信号量可以被认为是长度为 1 的队列那样，计数信号量也可以被认为是长度大于 1 的队列。信号量的用户对存储在队列中的数据不感兴趣，他们只关心队列是否为

空。

计数信号量通常用于两种情况：

（1）事件计数：在此使用方案中，每次事件发生时，事件处理程序将“给出”一个信号量（信号量计数值递增），并且 处理程序任务每次处理事件（信号量计数值递减）时“获取”一个信号量。因此，计数值是 已发生的事件数与已处理的事件数之间的差值。在这种情况下， 创建信号量时计数值可以为零。

（2）资源管理：在此使用情景中，计数值表示可用资源的数量。要获得对资源的控制权，任务必须首先获取 一个信号量——同时递减信号量计数值。当计数值达到零时，表示没有空闲资源可用。当任务使用完资源时，“返还”一个信号量——同时递增信号量计数值。在这种情况下， 创建信号量时计数值可以等于最大计数值。

计数型信号量相关函数：

函数	描述
xSemaphoreCreateCounting()	使用动态方法创建计数型信号量。
xSemaphoreCreateCountingStatic()	使用静态方法创建计数型信号量
uxSemaphoreGetCount()	获取信号量的计数值

11.5 计数型信号量实验（掌握）

11.5.1 实验目标

学习使用 FreeRTOS 的计数型信号量相关函数：

- start_task：用来创建其他的 2 个任务。
- task1：用于按键扫描，当检测到按键 KEY1 被按下时，释放计数型信号量。
- task2：每过一秒获取一次计数型信号量，当成功获取后打印信号量计数值。

11.5.2 FreeRTOSConfig.h 代码清单

```
#define configUSE_COUNTING_SEMAPHORES 1
```

11.5.3 freertos_demo.c 代码清单

1) 引入信号量头文件

```
#include "semphr.h"
```

2) 任务配置

```
/* 启动任务函数 */
```

```
#define START_TASK_PRIORITY 1
```

更多 [Java](#) - [大数据](#) - [前端](#) - [python](#) 人工智能资料下载，可百度访问：[尚硅谷官网](#)

```
#define START_TASK_STACK_DEPTH 128
TaskHandle_t start_task_handler;
void Start_Task(void *pvParameters);
```

```
/* Task1 任务 配置 */
```

```
#define TASK1_PRIORITY 2
#define TASK1_STACK_DEPTH 128
TaskHandle_t task1_handler;
void Task1(void *pvParameters);
```

```
/* Task2 任务 配置 */
```

```
#define TASK2_PRIORITY 3
#define TASK2_STACK_DEPTH 128
TaskHandle_t task2_handler;
void Task2(void *pvParameters);
```

3) 入口函数

```
QueueHandle_t count_semaphore_handle;
/**
 * @description: FreeRTOS 入口函数：创建任务函数并开始调度
 * @return {*}
 */
void FreeRTOS_Start(void)
{
    /* 创建计数型信号量 */
    count_semaphore_handle = xSemaphoreCreateCounting(100 , 0);
    if(count_semaphore_handle != NULL)
    {
        printf("计数型信号量创建成功\r\n");
    }
    xTaskCreate((TaskFunction_t)Start_Task,
                (char *)"Start_Task",
                (configSTACK_DEPTH_TYPE)START_TASK_STACK_DEPTH,
                (void *)NULL,
                (UBaseType_t)START_TASK_PRIORITY,
                (TaskHandle_t *)&start_task_handler);
    vTaskStartScheduler();
}
```

4) 初始任务函数

```
void Start_Task(void *pvParameters)
{
```

```
taskENTER_CRITICAL(); /* 进入临界区 */
xTaskCreate((TaskFunction_t)Task1,
            (char *)"Task1",
            (configSTACK_DEPTH_TYPE)TASK1_STACK_DEPTH,
            (void *)NULL,
            (UBaseType_t)TASK1_PRIORITY,
            (TaskHandle_t *)&task1_handler);

xTaskCreate((TaskFunction_t)Task2,
            (char *)"Task2",
            (configSTACK_DEPTH_TYPE)TASK2_STACK_DEPTH,
            (void *)NULL,
            (UBaseType_t)TASK2_PRIORITY,
            (TaskHandle_t *)&task2_handler);
vTaskDelete(NULL);
taskEXIT_CRITICAL(); /* 退出临界区 */
}
```

5) task1 任务函数

```
/**
 * @description: 释放计数型信号量
 * @param {void *} pvParameters
 * @return {*}
 */
void Task1(void *pvParameters)
{
    uint8_t key = 0;
    while(1)
    {
        key = Key_Detect();
        if(key == KEY1_PRESS)
        {
            if(count_semaphore_handle != NULL)
            {
                /* 释放信号量 */
                xSemaphoreGive(count_semaphore_handle);
            }
        }
        vTaskDelay(10);
    }
}
```

6) task2 任务函数

更多 [Java](#) - [大数据](#) - [前端](#) - [python](#) 人工智能资料下载，可百度访问：[尚硅谷官网](#)

```
/**
 * @description: 获取计数型信号量
 * @param {void *} pvParameters
 * @return {*}
 */
void Task2(void *pvParameters)
{
    BaseType_t err = 0;
    while(1)
    {
        /* 一直等待获取信号量 */
        err = xSemaphoreTake(count_semaphore_handle,portMAX_DELAY);
        if(err == pdTRUE)
        {
            printf("信号量的计数值\n", (int)uxSemaphoreGetCount(count_semaphore_handle));
        }
        vTaskDelay(1000);
    }
}
```

11.6 优先级翻转简介（熟悉）

优先级翻转是一个在实时系统中可能出现的问题，特别是在多任务环境中。该问题指的是一个较低优先级的任务阻塞了一个较高优先级任务的执行，从而导致高优先级任务无法及时完成。

典型的优先级翻转场景如下：

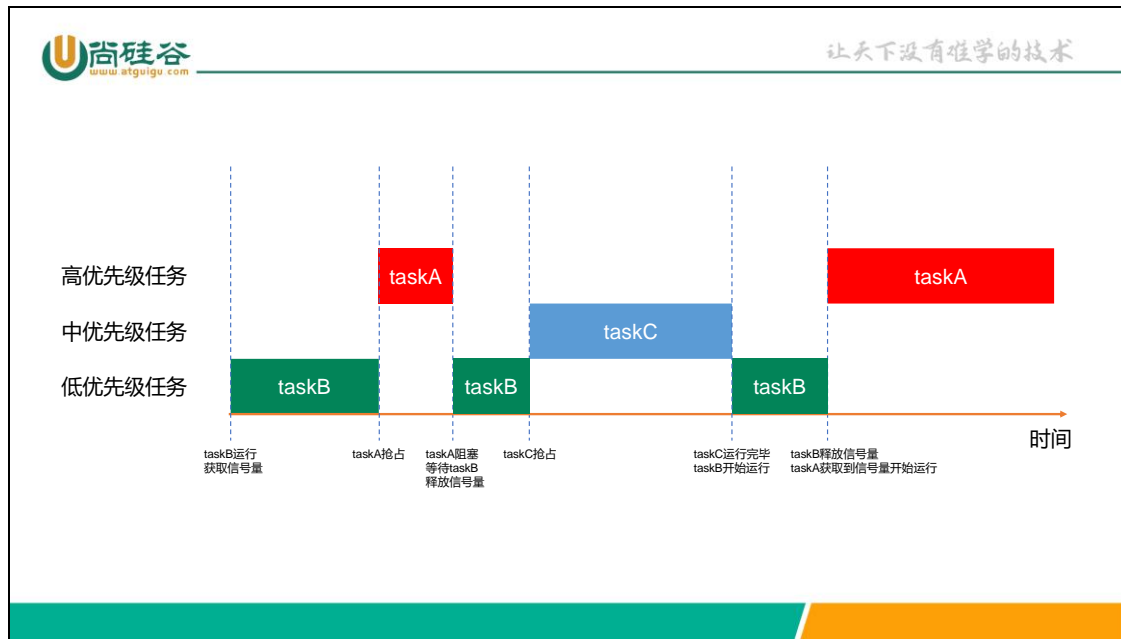
- 任务 A（高优先级）：拥有高优先级，需要访问共享资源，比如一个关键数据结构。
- 任务 B（低优先级）：拥有低优先级，目前正在访问该共享资源。
- 任务 C（中优先级）：位于任务 A 和任务 B 之间，具有介于两者之间的优先级。

具体流程如下：

- （1）任务 A 开始执行，但由于任务 B 正在访问共享资源，任务 A 被阻塞等待。
- （2）任务 C 获得执行权，由于优先级高于任务 B，它可以抢占任务 B。
- （3）任务 C 执行完成后，任务 B 被解除阻塞，开始执行，完成后释放了共享资源。
- （4）任务 A 重新获取执行权，继续执行。

这个过程中，任务 A 因为资源被占用而被阻塞，而任务 B 却被中优先级的任务 C 抢占，导致任务 B 无法及时完成。这种情况称为优先级翻转，因为任务 C 的介入翻转了高优先级更多 [Java - 大数据 - 前端 - python 人工智能资料下载](#)，可百度访问：[尚硅谷官网](#)

任务 A 的执行顺序。



11.7 优先级翻转实验（掌握）

11.7.1 实验目标

模拟优先级翻转，观察对抢占式内核的影响：

- start_task：用来创建其他的 3 个任务。
- task1：低优先级任务，同高优先级一样的操作，不同的是低优先级任务占用信号量的时间久一点。
- task2：中等优先级任务，简单的应用任务。
- task3：高优先级任务，会获取二值信号量，获取成功以后打印提示信息，处理完后释放信号量。

11.7.2 freertos_demo.c 代码清单

1) 引入信号量头文件

```
#include "semphr.h"
```

2) 任务配置

```
/* 启动任务函数 */
#define START_TASK_PRIORITY 1
#define START_TASK_STACK_DEPTH 128
TaskHandle_t start_task_handler;
void Start_Task(void *pvParameters);
```

更多 [Java](#) - [大数据](#) - [前端](#) - [python](#) 人工智能资料下载，可百度访问：尚硅谷官网

```
/* Task1 任务 配置 */
#define TASK1_PRIORITY 2
#define TASK1_STACK_DEPTH 128
TaskHandle_t task1_handler;
void Task1(void *pvParameters);

/* Task2 任务 配置 */
#define TASK2_PRIORITY 3
#define TASK2_STACK_DEPTH 128
TaskHandle_t task2_handler;
void Task2(void *pvParameters);
```

3) 添加延时函数

```
/**
 * @description: for 循环实现延时函数
 * @param {uint32_t} ms
 * @return {*}
 */
void for_delay_ms(uint32_t ms)
{
    uint32_t Delay = ms * 72000/9; /* 72M 时钟频率, 9 是 PLL 倍频 */
    do
    {
        __NOP(); /* 空指令 (NOP) 来占用 CPU 时间 */
    }
    while (Delay --);
}
```

4) 入口函数

```
QueueHandle_t semaphore_handle;
/**
 * @description: FreeRTOS 入口函数: 创建任务函数并开始调度
 * @return {*}
 */
void FreeRTOS_Start(void)
{
    /* 创建二值信号量, 并释放一次信号量 */
    semaphore_handle = xSemaphoreCreateBinary();
    if(semaphore_handle != NULL)
    {
        printf("二值信号量创建成功\r\n");
    }
}
```

```
}
xSemaphoreGive(semaphore_handle);

xTaskCreate((TaskFunction_t)Start_Task,
            (char *)"Start_Task",
            (configSTACK_DEPTH_TYPE)START_TASK_STACK_DEPTH,
            (void *)NULL,
            (UBaseType_t)START_TASK_PRIORITY,
            (TaskHandle_t *)&start_task_handler);
vTaskStartScheduler();
}
```

5) 初始任务函数

```
void Start_Task( void * pvParameters )
{
    taskENTER_CRITICAL();           /* 进入临界区 */
    xTaskCreate((TaskFunction_t      ) Task1,
                (char *              ) "Task1",
                (configSTACK_DEPTH_TYPE ) TASK1_STACK_DEPTH,
                (void *              ) NULL,
                (UBaseType_t         ) TASK1_PRIORITY,
                (TaskHandle_t *      ) &task1_handler );

    xTaskCreate((TaskFunction_t      ) Task2,
                (char *              ) "Task2",
                (configSTACK_DEPTH_TYPE ) TASK2_STACK_DEPTH,
                (void *              ) NULL,
                (UBaseType_t         ) TASK2_PRIORITY,
                (TaskHandle_t *      ) &task2_handler );

    xTaskCreate((TaskFunction_t      ) Task3,
                (char *              ) "Task3",
                (configSTACK_DEPTH_TYPE ) TASK3_STACK_DEPTH,
                (void *              ) NULL,
                (UBaseType_t         ) TASK3_PRIORITY,
                (TaskHandle_t *      ) &task3_handler );

    vTaskDelete(NULL);
    taskEXIT_CRITICAL();           /* 退出临界区 */
}
```

6) 低优先级任务函数

```
/**
```

```

* @description: 低优先级任务
* @param {void *} pvParameters
* @return {*}
*/
void Task1(void * pvParameters)
{
    while(1)
    {
        printf("低优先级 Task1 获取信号量\r\n");
        xSemaphoreTake(semaphore_handle, portMAX_DELAY);
        printf("低优先级 Task1 正在运行\r\n");
        for_delay_ms(3000);
        printf("低优先级 Task1 释放信号量\r\n");
        xSemaphoreGive(semaphore_handle);
        vTaskDelay(1000);
    }
}
```

7) 中优先级任务函数

```

/**
* @description: 任务 2: 获取二值信号量, 打印相关信息
* @return {*}
*/
void Task2(void *pvParameters)
{
    while (1)
    {
        printf("中优先级的 Task2 正在执行\r\n");
        for_delay_ms(1500);
        printf("Task2 for 循环完毕.....\r\n");
        vTaskDelay(1000);
    }
}
```

8) 高优先级任务函数

```

/**
* @description: 高优先级任务
* @param {void *} pvParameters
* @return {*}
*/
void Task3(void * pvParameters)
{

```

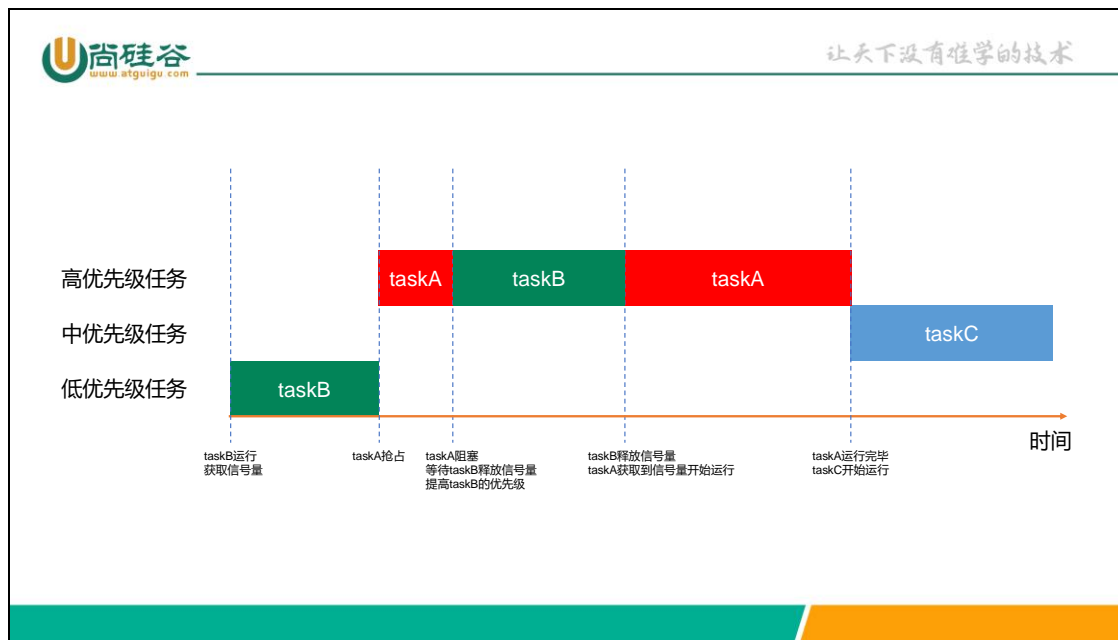


```
while(1)
{
    printf("高优先级 Task3 获取信号量\r\n");
    xSemaphoreTake(semaphore_handle,portMAX_DELAY);
    printf("高优先级 Task3 正在运行\r\n");
    for_delay_ms(1000);
    printf("高优先级 Task3 释放信号量\r\n");
    xSemaphoreGive(semaphore_handle);
    vTaskDelay(1000);
}
}
```

11.8 互斥信号量（熟悉）

互斥信号量是包含优先级继承机制的二进制信号量。二进制信号量能更好实现实现同步（任务间或任务与中断之间），而互斥信号量有助于更好实现简单互斥（即相互排斥）。

优先级继承是一种解决实时系统中任务调度引起的优先级翻转问题的机制。在具体的任务调度中，当一个高优先级任务等待一个低优先级任务所持有的资源时，系统会提升低优先级任务的优先级，以避免高优先级任务长时间等待的情况。



优先级继承无法完全解决优先级翻转，只是在某些情况下将影响降至最低。

不能在中断中使用互斥信号量，原因如下：

➤ 互斥信号量使用的优先级继承机制要求从任务中（而不是从中断中）获取和释放互斥信号量。

更多 [Java](#) - [大数据](#) - [前端](#) - [python](#) 人工智能资料下载，可百度访问：[尚硅谷官网](#)

- 中断无法保持阻塞来等待一个被互斥信号量保护的资源。

互斥信号量相关函数：

函数	描述
xSemaphoreCreateMutex()	使用动态方法创建互斥信号量。
xSemaphoreCreateMutexStatic()	使用静态方法创建互斥信号量。

互斥信号量的获取和释放函数与二值信号量的相应函数相似，但有一个重要的区别：互斥信号量不支持在中断服务程序中直接调用。注意，当创建互斥信号量时，系统会自动进行一次信号量的释放操作。

11.9 互斥信号量实验（掌握）

11.9.1 实验目标

在前面优先级翻转实验的案例中，通过互斥信号量来解决优先级翻转问题：

信号量函数改成互斥信号量，通过串口打印提示信息。

11.9.2 FreeRTOSConfig.h 代码清单

```
#define configUSE_MUTEXES 1
```

11.9.3 freertos_demo.c 代码清单

1) 引入信号量头文件

```
#include "semphr.h"
```

2) 任务配置

```
/* 启动任务函数 */
#define START_TASK_PRIORITY 1
#define START_TASK_STACK_DEPTH 128
TaskHandle_t start_task_handler;
void Start_Task(void *pvParameters);

/* Task1 任务 配置 */
#define TASK1_PRIORITY 2
#define TASK1_STACK_DEPTH 128
TaskHandle_t task1_handler;
void Task1(void *pvParameters);

/* Task2 任务 配置 */
#define TASK2_PRIORITY 3
```

```
#define TASK2_STACK_DEPTH 128
TaskHandle_t task2_handler;
void Task2(void *pvParameters);
```

3) 添加延时函数

```
/**
 * @description: for 循环实现延时函数
 * @param {uint32_t} ms
 * @return {*}
 */
void for_delay_ms(uint32_t ms)
{
    uint32_t Delay = ms * 72000/9; /* 72M 时钟频率, 9 是 PLL 倍频 */
    do
    {
        __NOP(); /* 空指令 (NOP) 来占用 CPU 时间 */
    }
    while (Delay --);
}
```

4) 入口函数

```
QueueHandle_t mutex_semaphore_handle;
/**
 * @description: FreeRTOS 入口函数: 创建任务函数并开始调度
 * @return {*}
 */
void FreeRTOS_Start(void)
{
    /* 创建互斥信号量, 并且主动释放一次信号量 */
    mutex_semaphore_handle = xSemaphoreCreateMutex();
    if (mutex_semaphore_handle != NULL)
    {
        printf("互斥信号量创建成功\r\n");
    }

    xTaskCreate((TaskFunction_t)Start_Task,
                (char *)"Start_Task",
                (configSTACK_DEPTH_TYPE)START_TASK_STACK_DEPTH,
                (void *)NULL,
                (UBaseType_t)START_TASK_PRIORITY,
                (TaskHandle_t *)&start_task_handler);
    vTaskStartScheduler();
}
```

```
}
```

5) 初始任务函数

```
void Start_Task( void * pvParameters )
{
    taskENTER_CRITICAL();           /* 进入临界区 */
    xTaskCreate((TaskFunction_t      ) Task1,
                (char *               ) "Task1",
                (configSTACK_DEPTH_TYPE) TASK1_STACK_DEPTH,
                (void *               ) NULL,
                (UBaseType_t          ) TASK1_PRIORITY,
                (TaskHandle_t *       ) &task1_handler );

    xTaskCreate((TaskFunction_t      ) Task2,
                (char *               ) "Task2",
                (configSTACK_DEPTH_TYPE) TASK2_STACK_DEPTH,
                (void *               ) NULL,
                (UBaseType_t          ) TASK2_PRIORITY,
                (TaskHandle_t *       ) &task2_handler );

    xTaskCreate((TaskFunction_t      ) Task3,
                (char *               ) "Task2",
                (configSTACK_DEPTH_TYPE) TASK3_STACK_DEPTH,
                (void *               ) NULL,
                (UBaseType_t          ) TASK3_PRIORITY,
                (TaskHandle_t *       ) &task3_handler );

    vTaskDelete(NULL);
    taskEXIT_CRITICAL();           /* 退出临界区 */
}
```

6) 低优先级任务函数

```
/**
 * @description: 低优先级任务
 * @param {void *} pvParameters
 * @return {*}
 */
void Task1(void * pvParameters)
{
    while(1)
    {
        printf("低优先级 Task1 获取信号量\r\n");
        xSemaphoreTake(mutex_semaphore_handle, portMAX_DELAY);
    }
}
```

```
    printf("低优先级 Task1 正在运行\r\n");
    for_delay_ms(3000);
    printf("低优先级 Task1 释放信号量\r\n");
    xSemaphoreGive(mutex_semaphore_handle);
    vTaskDelay(1000);
}
}
```

7) 中优先级任务函数

```
/**
 * @description: 任务 2: 获取二值信号量, 打印相关信息
 * @return {*}
 */
void Task2(void *pvParameters)
{
    while (1)
    {
        printf("中优先级的 Task2 正在执行\r\n");
        for_delay_ms(1500);
        printf("Task2 for 循环完毕.....\r\n");
        vTaskDelay(1000);
    }
}
```

8) 高优先级任务函数

```
/**
 * @description: 高优先级任务
 * @param {void *} pvParameters
 * @return {*}
 */
void Task3(void * pvParameters)
{
    while(1)
    {
        printf("高优先级 Task3 获取信号量\r\n");
        xSemaphoreTake(mutex_semaphore_handle,portMAX_DELAY);
        printf("高优先级 Task3 正在运行\r\n");
        for_delay_ms(1000);
        printf("高优先级 Task3 释放信号量\r\n");
        xSemaphoreGive(mutex_semaphore_handle);
        vTaskDelay(1000);
    }
}
```

```
}
```

第 12 章 队列集

12.1 队列集简介（了解）

队列集（Queue Set）是 FreeRTOS 中的一种数据结构，用于管理多个队列。它提供了一种有效的方式，通过单个 API 调用来操作和访问一组相关的队列。

在多任务系统中，任务之间可能需要共享数据，而这些数据可能存储在不同的队列中。队列集的作用就是为了更方便地管理这些相关队列，使得任务能够轻松地访问和处理多个队列的数据。

队列集的特点和用法：

- 集中管理多个队列：队列集允许你将多个相关联的队列组织在一起，方便集中管理。
- 单一 API 调用：通过单一的 API 调用，任务可以同时操作多个队列，而无需分别处理每个队列。
- 简化任务代码：对于需要处理多个相关队列的任务，使用队列集可以简化代码，提高可读性和维护性。
- 提高系统效率：在需要协同工作的任务之间共享和传递数据时，队列集可以提高系统的效率。
- 协同工作：任务可以更方便地协同工作，共享数据，实现更复杂的任务间通信和同步。

使用队列集时，你需要了解如何创建、添加和访问队列集，以及如何使用队列集 API 进行数据的发送和接收。队列集是 FreeRTOS 提供的一个强大工具，用于更灵活地组织和处理任务之间的数据流。

想象一下你有一个智能家居系统，有一个任务负责处理温度信息，另一个任务负责光照信息。你可能有两个队列，一个用于温度，一个用于光照。现在，通过队列集，你可以方便地管理这两个队列，让控制任务能够在需要时从这两个队列中获取信息，从而更智能地控制环境。

12.2 队列集相关 API 函数介绍（熟悉）

队列集相关函数：

函数	描述
xQueueCreateSet()	创建队列集
xQueueAddToSet()	队列添加到队列集中
xQueueRemoveFromSet()	从队列集中移除队列
xQueueSelectFromSet()	获取队列集中有有效消息的队列
xQueueSelectFromSetFromISR()	在中断中获取队列集中有有效消息的队列

12.3 队列集操作实验（掌握）

12.3.1 实验目标

学习使用 FreeRTOS 的队列集相关函数：

➤ start_task：用来创建其他 2 个任务，并创建队列集、队列/信号量，将队列/信号量添加到队列集中。

➤ task1：用于扫描按键，当 KEY1 按下，往队列写入数据，当 KEY2 按下，释放二值信号量。

➤ task2：读取队列集中的消息，并打印。

12.3.2 FreeRTOSConfig.h 代码清单

```
#define configUSE_QUEUE_SETS 1
```

12.3.3 freertos_demo.c 代码清单

1) 引入头文件

```
#include "queue.h"
#include "semphr.h"
```

2) 任务配置

```
/* 启动任务函数 */
#define START_TASK_PRIORITY 1
#define START_TASK_STACK_DEPTH 128
TaskHandle_t start_task_handler;
void Start_Task(void *pvParameters);

/* Task1 任务 配置 */
#define TASK1_PRIORITY 2
#define TASK1_STACK_DEPTH 128
TaskHandle_t task1_handler;
void Task1(void *pvParameters);
```

更多 [Java](#) - [大数据](#) - [前端](#) - [python](#) 人工智能资料下载，可百度访问：尚硅谷官网

```
/* Task2 任务 配置 */
#define TASK2_PRIORITY 3
#define TASK2_STACK_DEPTH 128
TaskHandle_t task2_handler;
void Task2(void *pvParameters);
```

3) 入口函数

```
QueueSetHandle_t queueset_handle;
QueueHandle_t queue_handle;
QueueHandle_t semphr_handle;
/**
 * @description: FreeRTOS 入口函数：创建任务函数并开始调度
 * @return {*}
 */
void FreeRTOS_Start(void)
{
    xTaskCreate((TaskFunction_t)Start_Task,
                (char *)"Start_Task",
                (configSTACK_DEPTH_TYPE)START_TASK_STACK_DEPTH,
                (void *)NULL,
                (UBaseType_t)START_TASK_PRIORITY,
                (TaskHandle_t *)&start_task_handler);
    vTaskStartScheduler();
}
```

4) 初始任务函数

```
void Start_Task(void *pvParameters)
{
    taskENTER_CRITICAL(); /* 进入临界区 */
    /* 创建队列集，可以存放 2 个队列 */
    queueset_handle = xQueueCreateSet(2);
    if (queueset_handle != NULL)
    {
        printf("队列集创建成功\r\n");
    }
    /* 创建队列 */
    queue_handle = xQueueCreate(1, sizeof(uint8_t));
    /* 创建二值信号量 */
    semphr_handle = xSemaphoreCreateBinary();
    /* 添加到队列集 */
    xQueueAddToSet(queue_handle, queueset_handle);
```



```
xQueueAddToSet(semphr_handle, queueset_handle);

xTaskCreate((TaskFunction_t)Task1,
            (char *)"Task1",
            (configSTACK_DEPTH_TYPE)TASK1_STACK_DEPTH,
            (void *)NULL,
            (UBaseType_t)TASK1_PRIORITY,
            (TaskHandle_t *)&task1_handler);

xTaskCreate((TaskFunction_t)Task2,
            (char *)"Task2",
            (configSTACK_DEPTH_TYPE)TASK2_STACK_DEPTH,
            (void *)NULL,
            (UBaseType_t)TASK2_PRIORITY,
            (TaskHandle_t *)&task2_handler);

vTaskDelete(NULL);
taskEXIT_CRITICAL(); /* 退出临界区 */
}
```

5) task1 任务函数

```
/**
 * @description: 实现队列发送以及信号量释放
 * @param {void *} pvParameters
 * @return {*}
 */
void Task1(void *pvParameters)
{
    uint8_t key = 0;
    BaseType_t err = 0;
    while (1)
    {
        key = Key_Detect();
        if (key == KEY1_PRESS)
        {
            err = xQueueSend(queue_handle, &key, portMAX_DELAY);
            if (err == pdPASS)
            {
                printf("往队列 queue_handle 写入数据成功\r\n");
            }
        }
        else if (key == KEY2_PRESS)
        {
            // ...
        }
    }
}
```

```
    {
        err = xSemaphoreGive(semphr_handle);
        if (err == pdPASS)
        {
            printf("释放信号量成功\r\n");
        }
    }
    vTaskDelay(10);
}
```

6) task2 任务函数

```
/**
 * @description: 获取队列集的消息
 * @param {void *} pvParameters
 * @return {*}
 */
void Task2(void *pvParameters)
{
    QueueSetMemberHandle_t member_handle;
    uint8_t key;
    while (1)
    {
        member_handle = xQueueSelectFromSet(queueaset_handle,
portMAX_DELAY);
        if (member_handle == queue_handle)
        {
            xQueueReceive(member_handle, &key, portMAX_DELAY);
            printf("获取到的队列数据=%d\r\n", key);
        }
        else if (member_handle == semphr_handle)
        {
            xSemaphoreTake(member_handle, portMAX_DELAY);
            printf("获取信号量成功\r\n");
        }
    }
}
```

第 13 章 事件标志组

13.1 事件标志组简介（了解）

13.1.1 基本概念

当在嵌入式系统中运行多个任务时，这些任务可能需要相互通信，协调其操作。FreeRTOS 中的事件标志组（Event Flags Group）提供了一种轻量级的机制，用于在任务之间传递信息和同步操作。

事件标志组就像是一个共享的标志牌集合，每个标志位都代表一种特定的状态或事件。任务可以等待或设置这些标志位，从而实现任务之间的协同工作。

1) 事件位（事件标志）

事件位用于指示事件是否发生。事件位通常称为事件标志。例如，应用程序可以：

- 定义一个位（或标志），设置为 1 时表示“已收到消息并准备好处理”，设置为 0 时表示“没有消息等待处理”。
- 定义一个位（或标志），设置为 1 时表示“应用程序已将准备发送到网络的消息排队”，设置为 0 时表示“没有消息需要排队准备发送到网络”。
- 定义一个位（或标志），设置为 1 时表示“需要向网络发送心跳消息”，设置为 0 时表示“不需要向网络发送心跳消息”。

2) 事件组

事件组就是一组事件位。事件组中的事件位通过位编号来引用。同样，以上面列出的三个例子为例：

- 事件标志组位编号为 0 表示“已收到消息并准备好处理”。
- 事件标志组位编号为 1 表示“应用程序已将准备发送到网络的消息排队”。
- 事件标志组位编号为 2 表示“需要向网络发送心跳消息”。

13.1.2 事件组和事件位数据类型

事件组由 `EventGroupHandle_t` 类型的变量引用。

在事件组中实现的位数（或标志数）取决于是使用 `configUSE_16_BIT_TICKS` 还是 `configTICK_TYPE_WIDTH_IN_BITS` 来控制 `TickType_t` 的类型：

- 如果 `configUSE_16_BIT_TICKS` 设置为 1，则事件组内实现的位数（或标志数）

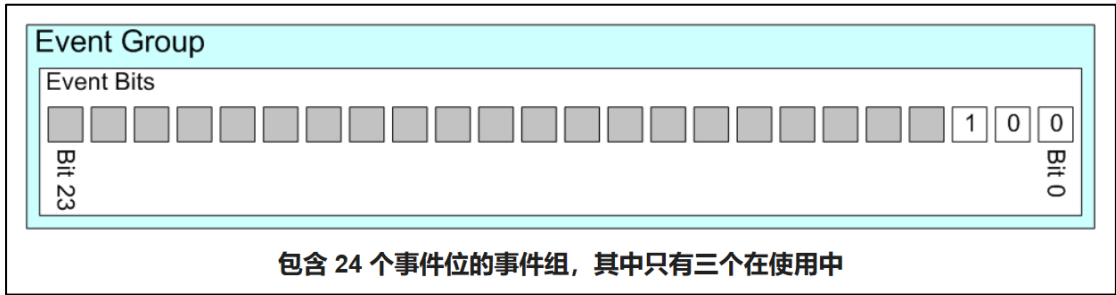
为 8；如果 configUSE_16_BIT_TICKS 设置为 0，则为 24。

- 如果 configTICK_TYPE_WIDTH_IN_BITS 设为 TICK_TYPE_WIDTH_16_BITS，则事件组内实现的位数（或标志数）为 8。
- 如果 configTICK_TYPE_WIDTH_IN_BITS 设为 TICK_TYPE_WIDTH_32_BITS，则为 24。
- 如果 configTICK_TYPE_WIDTH_IN_BITS 设为 TICK_TYPE_WIDTH_64_BITS，则为 56。

对 configUSE_16_BIT_TICKS 或 configTICK_TYPE_WIDTH_IN_BITS 的依赖于 RTOS 任务内部实现中用于线程本地存储的数据类型。我们当前的版本不支持 configTICK_TYPE_WIDTH_IN_BITS 配置，只有 configUSE_16_BIT_TICKS 配置。

事件组中的所有事件位都存储在 EventBits_t 类型的单个无符号整数变量中。事件位 0 存储在位 0 中，事件位 1 存储在位 1 中，依此类推。

下图表示一个 24 位事件组，使用 3 个位来保存前面描述的 3 个示例事件。在图片中，仅设置了事件位 2。



13.1.3 事件标志组和信号量的区别

事件标志组（Event Flags Group）和信号量（Semaphore）都是 FreeRTOS 中用于任务同步和通信的机制，但它们在用途和行为上有一些关键的区别。

事件标志组	信号量
主要用于任务之间的事件通知和同步。每个标志位通常代表一个特定的状态或事件，任务可以等待某些标志的发生或者设置标志来通知其他任务。	用于任务之间的资源控制和同步。信号量通常用来保护共享资源，控制对共享资源的访问，以及在任务之间提供同步。
每个标志位通常代表一个不同的事件，每个	信号量是一个计数器，可以具有大于 1 的

标志位只有两个状态，即已设置或未设置。	值，表示可用的资源数量。信号量的计数可以动态增减，而且可以用于实现互斥、同步等场景。
适用于需要向其他任务通知事件发生或等待特定事件的场景，例如数据准备就绪、某个条件满足等。	适用于需要对共享资源进行控制，限制同时访问某个资源的任务数量，以及确保任务按顺序访问共享资源的场景。
任务可以等待多个特定的标志位同时发生，或者等待任意一个标志位发生。	任务等待信号量的发放，当信号量的计数大于零时，任务可以继续执行。

总体来说，事件标志组更侧重于任务间的事件通知和同步，而信号量更侧重于资源的控制和同步。在设计中，根据具体需求选择合适的机制会更有利于系统的设计和性能。

13.2 事件标志组相关 API 函数介绍（熟悉）

事件标志组相关函数：

函数	描述
xEventGroupCreate()	使用动态方式创建事件标志组
xEventGroupCreateStatic()	使用静态方式创建事件标志组
xEventGroupClearBits()	清零事件标志位
xEventGroupClearBitsFromISR()	在中断中清零事件标志位
xEventGroupSetBits()	设置事件标志位
xEventGroupSetBitsFromISR()	在中断中设置事件标志位
xEventGroupWaitBits()	等待事件标志位
xEventGroupSync()	设置事件标志位，并等待事件标志位

13.3 事件标志组实验（掌握）

13.3.1 实验目标

学习使用 FreeRTOS 的事件标志组相关函数：

- start_task：用来创建其他 2 个任务，并创建事件标志组。
 - task1：读取按键按下键值，根据不同键值将事件标志组相应事件位置一，模拟事件发生。
 - task2：同时等待事件标志组中的多个事件位，当这些事件位都置 1 的话就执行相
- 更多 [Java](#) - [大数据](#) - [前端](#) - [python](#) 人工智能资料下载，可百度访问：[尚硅谷官网](#)

应的处理。

13.3.2 freertos_demo.c 代码清单

1) 引入头文件

```
#include "event_groups.h"
```

2) 任务配置

```
/* 启动任务函数 */
#define START_TASK_PRIORITY 1
#define START_TASK_STACK_DEPTH 128
TaskHandle_t start_task_handler;
void Start_Task(void *pvParameters);

/* Task1 任务 配置 */
#define TASK1_PRIORITY 2
#define TASK1_STACK_DEPTH 128
TaskHandle_t task1_handler;
void Task1(void *pvParameters);

/* Task2 任务 配置 */
#define TASK2_PRIORITY 3
#define TASK2_STACK_DEPTH 128
TaskHandle_t task2_handler;
void Task2(void *pvParameters);
```

3) 入口函数

```
EventGroupHandle_t eventgroup_handle;
#define EVENTBIT_0 (1 << 0)
#define EVENTBIT_1 (1 << 1)
/**
 * @description: FreeRTOS 入口函数：创建任务函数并开始调度
 * @return {*}
 */
void FreeRTOS_Start(void)
{
    xTaskCreate((TaskFunction_t)Start_Task,
                (char *)"Start_Task",
                (configSTACK_DEPTH_TYPE)START_TASK_STACK_DEPTH,
                (void *)NULL,
                (UBaseType_t)START_TASK_PRIORITY,
                (TaskHandle_t *)&start_task_handler);
    vTaskStartScheduler();
}
```

更多 [Java](#) - [大数据](#) - [前端](#) - [python](#) 人工智能资料下载，可百度访问：尚硅谷官网

```
}
```

4) 初始任务函数

```
void Start_Task( void * pvParameters )
{
    taskENTER_CRITICAL();           /* 进入临界区 */
    /* 创建事件标志组 */
    eventgroup_handle = xEventGroupCreate();
    if(eventgroup_handle != NULL)
    {
        printf("事件标志组创建成功\r\n");
    }
    xTaskCreate((TaskFunction_t) Task1,
                (char *) "Task1",
                (configSTACK_DEPTH_TYPE) TASK1_STACK_DEPTH,
                (void *) NULL,
                (UBaseType_t) TASK1_PRIORITY,
                (TaskHandle_t *) &task1_handler );

    xTaskCreate((TaskFunction_t) Task2,
                (char *) "Task2",
                (configSTACK_DEPTH_TYPE) TASK2_STACK_DEPTH,
                (void *) NULL,
                (UBaseType_t) TASK2_PRIORITY,
                (TaskHandle_t *) &task2_handler );

    vTaskDelete(NULL);
    taskEXIT_CRITICAL();           /* 退出临界区 */
}
```

5) task1 任务函数

```
/**
 * @description: 根据按键，事件标志组相应为置一
 * @param {void *} pvParameters
 * @return {*}
 */
void Task1(void * pvParameters)
{
    uint8_t key = 0;
    while(1)
    {
        key = Key_Detect();
        if(key == KEY1_PRESS)
```

```
{
    /* 将事件标志组的 bit0 位置 1 */
    xEventGroupSetBits( eventgroup_handle, EVENTBIT_0);
}
else if(key == KEY2_PRESS)
{
    /* 将事件标志组的 bit1 位置 1 */
    xEventGroupSetBits( eventgroup_handle, EVENTBIT_1);
}
vTaskDelay(10);
}
}
```

6) task2 任务函数

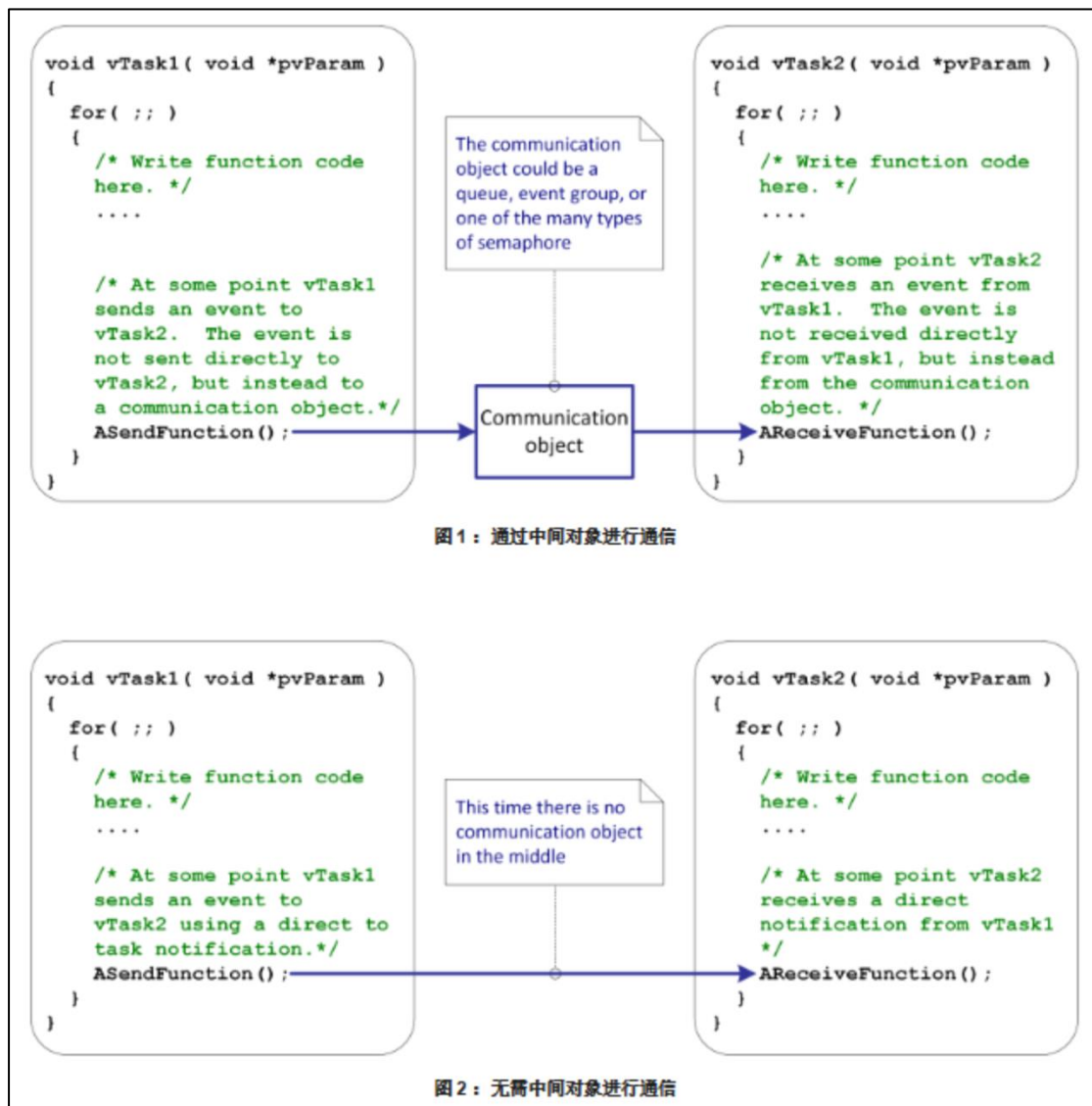
```
/**
 * @description: 同时等待事件标志组中的多个事件位
 * @param {void *} pvParameters
 * @return {*}
 */
void Task2(void * pvParameters)
{
    EventBits_t event_bit = 0;
    while(1)
    {
        event_bit = xEventGroupWaitBits( eventgroup_handle,          /* 事件标志组句柄 */
                                         EVENTBIT_0 | EVENTBIT_1,    /* 等待事件标志组的 bit0 和 bit1 位 */
                                         pdTRUE,                      /* 成功等待到事件标志位后，清除事件标志组中的 bit0 和 bit1 位 */
                                         pdTRUE,                      /* 等待事件标志组的 bit0 和 bit1 位都置 1,就成立 */
                                         portMAX_DELAY );            /* 一直等 */
        printf("等待到的事件标志位值=%#x\r\n",event_bit);
    }
}
```


第 14 章 FreeRTOS 任务通知

14.1 任务通知的简介（了解）

任务通知是 FreeRTOS 中一种用于任务间通信的机制，它允许一个任务向其他任务发送简单的通知或信号，以实现任务间的同步和协作。任务通知通常用于替代二值信号量或事件标志组，提供了更轻量级的任务间通信方式。

大多数任务间通信方法通过中间对象，如队列、信号量或事件组。发送任务写入通信对象，接收任务从通信对象读取。当使用直接任务通知时，顾名思义，发送任务直接向接收任务发送通知，而无需中间对象。



每个 RTOS 任务都有一个任务通知组，每条通知均独立运行，都有“挂起”或“非挂起”

的通知状态, 以及一个 32 位通知值。常量 `configTASK_NOTIFICATION_ARRAY_ENTRIES` 可设置任务通知组中的索引数量。在 FreeRTOS V10.4.0 版本前, 任务只有单条任务通知 (即只能一对一), 没有任务通知组。

向任务发送“任务通知”会将目标任务通知设为“挂起”状态。正如任务可以阻塞中间对象 (如等待信号量可用的信号量), 任务也可以阻塞任务通知, 以等待通知状态变为“挂起”。向任务发送“任务通知”也可以更新目标通知的值 (可选), 可使用下列任一方法:

- 覆盖原值, 无论接收任务是否已读取被覆盖的值。
- 覆盖原值 (仅当接收任务已读取被覆盖的值时)。
- 在值中设置一个或多个位。
- 对值进行增量 (添加 1)。

RTOS 任务通知功能默认为启用状态, 将 `configUSE_TASK_NOTIFICATIONS` 设为 0 可以禁用。

14.2 任务通知相关 API 函数介绍 (熟悉)

任务通知相关函数如下:

函数	描述
<code>xTaskNotify()</code>	发送通知, 带有通知值
<code>xTaskNotifyAndQuery()</code>	发送通知, 带有通知值并且保留接收任务的原通知值
<code>xTaskNotifyGive()</code>	发送通知, 不带通知值
<code>xTaskNotifyFromISR()</code>	在中断中发送任务通知
<code>xTaskNotifyAndQueryFromISR()</code>	
<code>vTaskNotifyGiveFromISR()</code>	
<code>ulTaskNotifyTake()</code>	获取任务通知, 可选退出函数时对通知置清零或减 1
<code>xTaskNotifyWait()</code>	获取任务通知, 可获取通知值和清除通知值的指定位

注意: 发送通知有相关 ISR 函数, 接收通知没有 ISR 函数, 不能在 ISR 中接收任务通知。

14.3 任务通知模拟信号量实验（掌握）

14.3.1 实验目标

学习将任务通知用作轻量级二进制信号量：

- start_task：用来创建其他 2 个任务。
- task1：用于按键扫描，当检测到按键 KEY1 被按下时，将发送任务通知。
- task2：用于接收任务通知，并打印相关提示信息。

14.3.2 freertos_demo.c 代码清单

1) 任务配置

```
/* 启动任务函数 */
#define START_TASK_PRIORITY 1
#define START_TASK_STACK_DEPTH 128
TaskHandle_t start_task_handler;
void Start_Task(void *pvParameters);

/* Task1 任务 配置 */
#define TASK1_PRIORITY 2
#define TASK1_STACK_DEPTH 128
TaskHandle_t task1_handler;
void Task1(void *pvParameters);

/* Task2 任务 配置 */
#define TASK2_PRIORITY 3
#define TASK2_STACK_DEPTH 128
TaskHandle_t task2_handler;
void Task2(void *pvParameters);
```

2) 入口函数

```
/**
 * @description: FreeRTOS 入口函数：创建任务函数并开始调度
 * @return {*}
 */
void FreeRTOS_Start(void)
{
    xTaskCreate((TaskFunction_t)Start_Task,
                (char *)"Start_Task",
                (configSTACK_DEPTH_TYPE)START_TASK_STACK_DEPTH,
                (void *)NULL,
```

```
        (UBaseType_t)START_TASK_PRIORITY,  
        (TaskHandle_t *)&start_task_handler);  
    vTaskStartScheduler();  
}
```

3) 初始任务函数

```
void Start_Task( void * pvParameters )  
{  
    taskENTER_CRITICAL();           /* 进入临界区 */  
    xTaskCreate((TaskFunction_t    ) Task1,  
                (char *            ) "Task1",  
                (configSTACK_DEPTH_TYPE) TASK1_STACK_DEPTH,  
                (void *            ) NULL,  
                (UBaseType_t       ) TASK1_PRIORITY,  
                (TaskHandle_t *    ) &task1_handler );  
  
    xTaskCreate((TaskFunction_t    ) Task2,  
                (char *            ) "Task2",  
                (configSTACK_DEPTH_TYPE) TASK2_STACK_DEPTH,  
                (void *            ) NULL,  
                (UBaseType_t       ) TASK2_PRIORITY,  
                (TaskHandle_t *    ) &task2_handler );  
    vTaskDelete(NULL);  
    taskEXIT_CRITICAL();           /* 退出临界区 */  
}
```

4) task1 任务函数

```
/**  
 * @description: 发送任务通知值  
 * @param {void *} pvParameters  
 * @return {*}  
 */  
void Task1(void * pvParameters)  
{  
    uint8_t key = 0;  
  
    while(1)  
    {  
        key = Key_Detect();  
        if(key == KEY1_PRESS)  
        {  
            printf("任务通知模拟二值信号量释放\r\n");  
        }  
    }  
}
```

```
        xTaskNotifyGive(task2_handler);
    }
    vTaskDelay(10);
}
}
```

5) task2 任务函数

```
/**
 * @description: 接收任务通知值
 * @param {void *} pvParameters
 * @return {*}
 */
void Task2(void * pvParameters)
{
    uint32_t rev = 0;
    while(1)
    {
        rev = ulTaskNotifyTake(pdTRUE , portMAX_DELAY);
        if(rev != 0)
        {
            printf("接收任务通知成功，模拟获取二值信号量\r\n");
        }
    }
}
```

14.4 任务通知模拟消息邮箱实验（掌握）

14.4.1 实验目标

学习将任务通知用作轻量级邮箱：

- start_task：用来创建其他 2 个任务。
- task1：用于按键扫描，将按下的按键键值通过任务通知发送给指定任务。
- task2：用于接收任务通知，并根据接收到的数据做相应动作。

14.4.2 freertos_demo.c 代码清单

1) 任务配置

```
/* 启动任务函数 */
#define START_TASK_PRIORITY 1
#define START_TASK_STACK_DEPTH 128
TaskHandle_t start_task_handler;
void Start_Task(void *pvParameters);
```

更多 [Java](#) - [大数据](#) - [前端](#) - [python](#) 人工智能资料下载，可百度访问：[尚硅谷官网](#)

```
/* Task1 任务 配置 */
#define TASK1_PRIORITY 2
#define TASK1_STACK_DEPTH 128
TaskHandle_t task1_handler;
void Task1(void *pvParameters);

/* Task2 任务 配置 */
#define TASK2_PRIORITY 3
#define TASK2_STACK_DEPTH 128
TaskHandle_t task2_handler;
void Task2(void *pvParameters);
```

2) 入口函数

```
/**
 * @description: FreeRTOS 入口函数：创建任务函数并开始调度
 * @return {*}
 */
void FreeRTOS_Start(void)
{
    xTaskCreate((TaskFunction_t)Start_Task,
                (char *)"Start_Task",
                (configSTACK_DEPTH_TYPE)START_TASK_STACK_DEPTH,
                (void *)NULL,
                (UBaseType_t)START_TASK_PRIORITY,
                (TaskHandle_t *)&start_task_handler);
    vTaskStartScheduler();
}
```

3) 初始任务函数

```
void Start_Task( void * pvParameters )
{
    taskENTER_CRITICAL();          /* 进入临界区 */
    xTaskCreate((TaskFunction_t)   ) Task1,
                (char *             ) "Task1",
                (configSTACK_DEPTH_TYPE) TASK1_STACK_DEPTH,
                (void *             ) NULL,
                (UBaseType_t        ) TASK1_PRIORITY,
                (TaskHandle_t *      ) &task1_handler );

    xTaskCreate((TaskFunction_t)   ) Task2,
                (char *             ) "Task2",
```

```
        (configSTACK_DEPTH_TYPE )   TASK2_STACK_DEPTH,  
        (void *                     )   NULL,  
        (UBaseType_t                )   TASK2_PRIORITY,  
        (TaskHandle_t *             )   &task2_handler );  
vTaskDelete(NULL);  
taskEXIT_CRITICAL();           /* 退出临界区 */  
}
```

4) task1 任务函数

```
/**  
 * @description: 发送任务通知值  
 * @param {void *} pvParameters  
 * @return {*}  
 */  
void Task1(void *pvParameters)  
{  
    uint8_t key = 0;  
  
    while (1)  
    {  
        key = Key_Detect();  
        if ((key != 0) && (task2_handler != NULL))  
        {  
            printf("任务通知模拟消息邮箱发送, 发送的键值为: %d\r\n", key);  
            xTaskNotify(task2_handler, key, eSetValueWithOverwrite);  
        }  
        vTaskDelay(10);  
    }  
}
```

5) task2 任务函数

```
/**  
 * @description: 接收任务通知值  
 * @param {void *} pvParameters  
 * @return {*}  
 */  
void Task2(void *pvParameters)  
{  
    uint32_t notify_val = 0;  
    while (1)  
    {  
        xTaskNotifyWait(0, 0xFFFFFFFF, &notify_val, portMAX_DELAY);  
    }  
}
```

```
switch (noyify_val)
{
case KEY1_PRESS:
{
    printf("接收到的通知值为: %d\r\n", noyify_val);
    LED_Toggle(LED1_Pin);
    break;
}
case KEY2_PRESS:
{
    printf("接收到的通知值为: %d\r\n", noyify_val);
    LED_Toggle(LED2_Pin);
    break;
}
default:
    break;
}
}
```

14.5 任务通知模拟事件标志组实验（掌握）

14.5.1 实验目标

学习将任务通知用作轻量级事件标志组：

- start_task：用来创建其他 2 个任务。
- task1：用于按键扫描，当检测到按键按下时，发送任务通知设置不同标志位。
- task2：用于接收任务通知，并打印相关提示信息。

14.5.2 freertos_demo.c 代码清单

1) 任务配置

```
/* 启动任务函数 */
#define START_TASK_PRIORITY 1
#define START_TASK_STACK_DEPTH 128
TaskHandle_t start_task_handler;
void Start_Task(void *pvParameters);

/* Task1 任务 配置 */
#define TASK1_PRIORITY 2
#define TASK1_STACK_DEPTH 128
```



```
TaskHandle_t task1_handler;
void Task1(void *pvParameters);

/* Task2 任务 配置 */
#define TASK2_PRIORITY 3
#define TASK2_STACK_DEPTH 128
TaskHandle_t task2_handler;
void Task2(void *pvParameters);

#define EVENTBIT_0 (1 << 0)
#define EVENTBIT_1 (1 << 1)
```

2) 入口函数

```
/**
 * @description: FreeRTOS 入口函数：创建任务函数并开始调度
 * @return {*}
 */
void FreeRTOS_Start(void)
{
    xTaskCreate((TaskFunction_t)Start_Task,
                (char *)"Start_Task",
                (configSTACK_DEPTH_TYPE)START_TASK_STACK_DEPTH,
                (void *)NULL,
                (UBaseType_t)START_TASK_PRIORITY,
                (TaskHandle_t *)&start_task_handler);
    vTaskStartScheduler();
}
```

3) 初始任务函数

```
void Start_Task( void * pvParameters )
{
    taskENTER_CRITICAL();          /* 进入临界区 */
    xTaskCreate((TaskFunction_t)   ) Task1,
                (char *             ) "Task1",
                (configSTACK_DEPTH_TYPE) TASK1_STACK_DEPTH,
                (void *             ) NULL,
                (UBaseType_t        ) TASK1_PRIORITY,
                (TaskHandle_t *      ) &task1_handler );

    xTaskCreate((TaskFunction_t)   ) Task2,
                (char *             ) "Task2",
                (configSTACK_DEPTH_TYPE) TASK2_STACK_DEPTH,
```

```
        (void *          ) NULL,  
        (UBaseType_t     ) TASK2_PRIORITY,  
        (TaskHandle_t *  ) &task2_handler );  
vTaskDelete(NULL);  
taskEXIT_CRITICAL();          /* 退出临界区 */  
}
```

4) task1 任务函数

```
/**  
 * @description: 发送任务通知值  
 * @param {void *} pvParameters  
 * @return {*}  
 */  
void Task1(void *pvParameters)  
{  
    uint8_t key = 0;  
    while (1)  
    {  
        key = Key_Detect();  
        if (key == KEY1_PRESS)  
        {  
            printf("将 bit0 位置 1\r\n");  
            xTaskNotify(task2_handler, EVENTBIT_0, eSetBits);  
        }  
        else if (key == KEY2_PRESS)  
        {  
            printf("将 bit1 位置 1\r\n");  
            xTaskNotify(task2_handler, EVENTBIT_1, eSetBits);  
        }  
        vTaskDelay(10);  
    }  
}
```

5) task2 任务函数

```
/**  
 * @description: 接收任务通知值  
 * @param {void *} pvParameters  
 * @return {*}  
 */  
void Task2(void *pvParameters)  
{  
    uint32_t notify_val = 0, event_bit = 0;
```

```
while(1)
{
    xTaskNotifyWait( 0, 0xFFFFFFFF, &notify_val, portMAX_DELAY );
    if(notify_val & EVENTBIT_0)
    {
        event_bit |= EVENTBIT_0;
    }
    if(notify_val & EVENTBIT_1)
    {
        event_bit |= EVENTBIT_1;
    }
    if(event_bit == (EVENTBIT_0|EVENTBIT_1))
    {
        printf("任务通知模拟事件标志组接收成功\r\n");
        event_bit = 0;
    }
}
```

第 15 章 FreeRTOS 软件定时器

15.1 软件定时器的简介（了解）

FreeRTOS 中的软件定时器是一种轻量级的时间管理工具，用于在任务中创建和管理定时器。软件定时器是基于 FreeRTOS 内核提供的时间管理功能实现的，允许开发者创建、启动、停止、删除和管理定时器，从而实现在任务中对时间的灵活控制。

软件定时器与硬件定时器的主要区别如下：

软件定时器	硬件定时器
FreeRTOS 提供的功能来模拟定时器，依赖系统的任务调度器来进行计时和任务调度	由芯片或微控制器提供，独立于 CPU，可以在后台运行，不受任务调度器的影响
精度和分辨率可能受到任务调度的影响	具有更高的精度和分辨率
不需要额外的硬件资源，但可能会增加系统的负载	占用硬件资源，不会增加 CPU 的负载

软件定时器能够让函数在未来的设定时间执行。由定时器执行的函数称为定时器的回调函数。从定时器启动到其回调函数执行之间的时间被称为定时器的周期。简而言之，当定时器的周期到期时，定时器的回调函数会被执行。

定时器回调函数在定时器服务任务的上下文中执行，在定时器回调函数中不能调用导致阻塞的 API 函数。

软件定时器服务任务是任务调度器中的一个特殊任务，专门用于管理和维护软件定时器的正常运行。如果 `configUSE_TIMERS` 设置为 1，在开启任务调度器的时候，会自动创建软件定时器服务的任务。它主要负责软件定时器超时的逻辑判断、调用超时软件定时器的超时回调函数、处理软件定时器命令队列。

15.2 软件定时器的状态（熟悉）

FreeRTOS 中的软件定时器有三种状态，分别是：

- 未创建（Uncreated）：软件定时器被创建之前的状态。在这个状态下，定时器的数据结构已经被定义，但尚未通过 `xTimerCreate()` 函数创建。
- 已创建（Created）：软件定时器已被成功创建，但尚未启动。在这个状态下，可以对定时器进行配置，如设置定时器的周期、回调函数等，但定时器并未开始计时。
- 已运行（Running）：软件定时器已经被启动，正在运行中。在这个状态下，定时器会按照预定的周期定时触发超时事件，执行注册的回调函数。

15.3 单次定时器和周期定时器（熟悉）

在 FreeRTOS 中，软件定时器主要有两种类型：一次性定时器和周期性定时器。

- 一次性定时器（One-shot Timer）：这种定时器在触发一次超时后就会停止，不再执行。适用于只需在特定时间执行一次任务或动作的场景。
- 周期性定时器（Periodic Timer）：这种定时器会在每个超时周期都触发一次，循环执行。适用于需要在固定的时间间隔内重复执行任务或动作的场景。

15.4 FreeRTOS 软件定时器相关 API 函数（熟悉）

软件定时器相关函数如下：

函数	描述
<code>xTimerCreate()</code>	动态方式创建软件定时器
<code>xTimerCreateStatic()</code>	静态方式创建软件定时器
<code>xTimerStart()</code>	开启软件定时器定时
<code>xTimerStartFromISR()</code>	在中断中开启软件定时器定时

xTimerStop()	停止软件定时器定时
xTimerStopFromISR()	在中断中停止软件定时器定时
xTimerReset()	复位软件定时器定时
xTimerResetFromISR()	在中断中复位软件定时器定时
xTimerChangePeriod()	更改软件定时器的定时超时时间
xTimerChangePeriodFromISR()	在中断中更改定时超时时间

15.5 FreeRTOS 软件定时器实验（掌握）

15.5.1 实验目标

学习使用 FreeRTOS 软件定时器的函数：

- start_task：用来创建 task1 任务，并创建一次性定时器和周期性定时器。
- task1：用于按键扫描，并对软件定时器进行开启、停止操作。

15.5.2 FreeRTOSConfig.h 代码清单

```
/* 软件定时器相关定义 */
#define configUSE_TIMERS 1 /* 1:
使能软件定时器，默认：0。使能后需指定下面 3 个 */
#define configTIMER_TASK_PRIORITY (configMAX_PRIORITIES - 1) /*
定义软件定时器任务的优先级 */
#define configTIMER_QUEUE_LENGTH 5 /* 定
义软件定时器命令队列的长度*/
#define configTIMER_TASK_STACK_DEPTH (configMINIMAL_STACK_SIZE * 2) /*
定义软件定时器任务的栈空间大小*/
```

15.5.3 freertos_demo.c 代码清单

1) 引入头文件

```
#include "timers.h"
```

2) 任务配置

```
/* 启动任务函数 */
#define START_TASK_PRIORITY 1
#define START_TASK_STACK_DEPTH 128
TaskHandle_t start_task_handler;
void Start_Task(void *pvParameters);
```

```
/* Task1 任务 配置 */
#define TASK1_PRIORITY 2
#define TASK1_STACK_DEPTH 128
TaskHandle_t task1_handler;
void Task1(void *pvParameters);

void timer1_callback( TimerHandle_t pxTimer );
void timer2_callback( TimerHandle_t pxTimer );
```

3) 入口函数

```
/**
 * @description: FreeRTOS 入口函数：创建任务函数并开始调度
 * @return {*}
 */
void FreeRTOS_Start(void)
{
    xTaskCreate((TaskFunction_t)Start_Task,
                (char *)"Start_Task",
                (configSTACK_DEPTH_TYPE)START_TASK_STACK_DEPTH,
                (void *)NULL,
                (UBaseType_t)START_TASK_PRIORITY,
                (TaskHandle_t *)&start_task_handler);
    vTaskStartScheduler();
}
```

4) 初始任务函数

```
TimerHandle_t timer1_handle = 0; /* 单次定时器 */
TimerHandle_t timer2_handle = 0; /* 周期定时器 */
void Start_Task(void *pvParameters)
{
    taskENTER_CRITICAL(); /* 进入临界区 */
    /* 创建单次定时器 */
    timer1_handle = xTimerCreate("timer1",
                                  500,
                                  pdFALSE,
                                  (void *)1,
                                  timer1_callback);

    /* 创建周期定时器 */
    timer2_handle = xTimerCreate("timer2",
                                  2000,
                                  pdTRUE,
```

```
        (void *)2,  
        timer2_callback);  
  
xTaskCreate((TaskFunction_t)Task1,  
            (char *)"Task1",  
            (configSTACK_DEPTH_TYPE)TASK1_STACK_DEPTH,  
            (void *)NULL,  
            (UBaseType_t)TASK1_PRIORITY,  
            (TaskHandle_t *)&task1_handler);  
vTaskDelete(NULL);  
taskEXIT_CRITICAL(); /* 退出临界区 */  
}
```

5) task1 任务函数

```
/**  
 * @description: 根据按键控制软件定时器  
 * @param {void *} pvParameters  
 * @return {*}  
 */  
void Task1(void *pvParameters)  
{  
    uint8_t key = 0;  
    while (1)  
    {  
        key = Key_Detect();  
        if (key == KEY1_PRESS)  
        {  
            xTimerStart(timer1_handle, portMAX_DELAY);  
            xTimerStart(timer2_handle, portMAX_DELAY);  
        }  
        else if (key == KEY2_PRESS)  
        {  
            xTimerStop(timer1_handle, portMAX_DELAY);  
            xTimerStop(timer2_handle, portMAX_DELAY);  
        }  
        vTaskDelay(10);  
    }  
}
```

6) 超时回调函数

```
/**  
 * @description: timer1 的超时回调函数
```

```
* @param {TimerHandle_t} pxTimer 定时器句柄
* @return {*}
*/
void timer1_callback(TimerHandle_t pxTimer)
{
    static uint32_t timer = 0;
    printf("timer1 的运行次数=%d\r\n", ++timer);
}

/**
* @description: timer2 的超时回调函数
* @param {TimerHandle_t} pxTimer 定时器句柄
* @return {*}
*/
void timer2_callback(TimerHandle_t pxTimer)
{
    static uint32_t timer = 0;
    printf("timer2 的运行次数=%d\r\n", ++timer);
}
```

第 16 章 Tickless 低功耗模式

16.1 低功耗模式简介（了解）

FreeRTOS 的 Tickless 模式是一种特殊的运行模式，用于最小化系统的时钟中断频率，以降低功耗。在 Tickless 模式下，系统只在有需要时才会启动时钟中断，而在无任务要运行时则完全进入休眠状态，从而降低功耗。在滴答中断重启时，会对 RTOS 滴答计数值进行校正调整。

Tickless 模式的实现方式通常依赖于微控制器的硬件特性，尤其是低功耗定时器或实时时钟单元。以下是 Tickless 模式的一般工作原理：

- 空闲任务检测：FreeRTOS 会通过空闲任务（Idle Task）来检测系统是否有任务需要执行。如果没有任务需要执行，系统可以进入休眠状态。
- 时钟中断：当有任务需要执行时，系统会启动时钟中断，唤醒处理器。
- 时钟中断处理：在时钟中断处理函数中，FreeRTOS 将检查任务的状态并决定是否继续执行。
- 休眠状态：如果没有任务需要执行，系统可以进入休眠状态，关闭时钟中断。在休眠状态下，处理器可以进入更低功耗的模式。

更多 [Java](#) - [大数据](#) - [前端](#) - [python](#) 人工智能资料下载，可百度访问：[尚硅谷官网](#)

➤ 任务唤醒：当有任务需要执行时，系统会再次启动时钟中断，唤醒处理器，然后执行相应的任务。

在 Tickless 模式下，系统的时钟中断频率明显降低，从而降低了系统的平均功耗。Tickless 模式适用于那些对功耗要求较高、需要长时间运行在低功耗状态的嵌入式系统。比如：电池驱动设备、物联网（IoT）设备、低功耗传感器节点、无线通信模块等。

16.2 Tickless 模式详解（熟悉）

STM32F103xC、STM32F103xD 和 STM32F103xE 增强型产品支持三种低功耗模式，可以在要求低功耗、短启动时间和多种唤醒事件之间达到最佳的平衡。

（1）睡眠模式（Sleep Mode）

只有 CPU 停止，所有外设处于工作状态并可在发生中断/事件时唤醒 CPU。

（2）停机模式（Stop Mode）

在保持 SRAM 和寄存器内容不丢失的情况下，停机模式可以达到最低的电能消耗。在停机模式下，停止所有内部 1.8V 部分的供电，PLL、HSI 的 RC 振荡器和 HSE 晶体振荡器被关闭，调压器可以被置于普通模式或低功耗模式。可以通过任一配置成 EXTI 的信号把微控制器从停机模式中唤醒，EXTI 信号可以是 16 个外部 I/O 口之一、PVD 的输出、RTC 闹钟或 USB 的唤醒信号。

（3）待机模式（Standby Mode）

在待机模式下可以达到最低的电能消耗。内部的电压调压器被关闭，因此所有内部 1.8V 部分的供电被切断；PLL、HSI 的 RC 振荡器和 HSE 晶体振荡器也被关闭；进入待机模式后，SRAM 和寄存器的内容将消失，但后备寄存器的内容仍然保留，待机电路仍工作。从待机模式退出的条件是：NRST 上的外部复位信号、IWDG 复位、WKUP 引脚上的一个上升边沿或 RTC 的闹钟到时。

注意：在进入停机或待机模式时，RTC、IWDG 和对应的时钟不会被停止。

模式	进入	唤醒	对1.8V区域时钟的影响	对VDD区域时钟的影响	电压调节器
睡眠 (SLEEP-NOW或 SLEEP-ON-EXIT)	WFI	任一中断	CPU时钟关，对其他时钟和ADC时钟无影响	无	开
	WFE	唤醒事件			
停机	PDDS和LPDS位 +SLEEPDEEP位 +WFI或WFE	任一外部中断(在外部中断寄存器中设置)	关闭所有1.8V区域的时钟	HSI 和HSE的振荡器关闭	开启或处于低功耗模式(依据电源控制寄存器(PWR_CR)的设置)
待机	PDDS位 +SLEEPDEEP位 +WFI或WFE	WKUP引脚的上升沿、RTC闹钟事件、NRST引脚上的外部复位、IWDG复位			关

主要使用睡眠模式，任何中断或事件都可以唤醒睡眠模式。Tickless 低功耗模式通过调用指令 `__WFI` 实现睡眠模式

FreeRTOS 系统中的所有其它任务都不在运行时（处于阻塞或挂起），会运行空闲任务。所以想不影响系统运行又降低功耗，可以在空闲任务执行的期间，让 MCU 进入相应的低功耗模式。

由于滴答定时器频繁中断则会影响低功耗，所以 FreeRTOS 的 Tickless 低功耗模式会自动把滴答定时器的中断周期修改为低功耗运行时间，退出低功耗后再补上系统时钟节拍数。

16.3 Tickless 模式相关配置项（掌握）

配置项	说明
<code>configUSE_TICKLESS_IDLE</code>	使能低功耗 Tickless 模式，默认 0
<code>configEXPECTED_IDLE_TIME_BEFORE_SLEEP</code>	系统进入相应低功耗模式的最短时长，默认 2
<code>configPRE_SLEEP_PROCESSING(x)</code>	在系统进入低功耗模式前执行的事务，比如关闭外设时钟
<code>configPOSR_SLEEP_PROCESSING(x)</code>	系统退出低功耗模式后执行的事务，比如开启之前关闭的外设时钟

16.4 Tickless 低功耗模式实验（掌握）

16.4.1 实验目标

学习使用 FreeRTOS 中的 Tickless 低功耗模式：

在 11.3 二值信号量实验案例中，加入低功耗模式，对比功耗结果，观察是否降低功耗。

16.4.2 FreeRTOSConfig.h 代码清单

```
#define configUSE_TICKLESS_IDLE 1
#include "freertos_demo.h"
#define configPRE_SLEEP_PROCESSING( x ) PRE_SLEEP_PROCESSING()
#define configPOST_SLEEP_PROCESSING( x ) POST_SLEEP_PROCESSING()
```

16.4.3 freertos_demo.c 代码清单

1) 引入信号量头文件

```
#include "semphr.h"
```

2) 低功耗处理函数

```
/* 进入低功耗前所需要执行的操作 */
void PRE_SLEEP_PROCESSING(void)
{
    __HAL_RCC_GPIOA_CLK_DISABLE();
    __HAL_RCC_GPIOB_CLK_DISABLE();
    __HAL_RCC_GPIOC_CLK_DISABLE();
    __HAL_RCC_GPIOD_CLK_DISABLE();
    __HAL_RCC_GPIOE_CLK_DISABLE();
    __HAL_RCC_GPIOF_CLK_DISABLE();
    __HAL_RCC_GPIOG_CLK_DISABLE();
}

/* 退出低功耗后所需要执行的操作 */
void POST_SLEEP_PROCESSING(void)
{
    __HAL_RCC_GPIOA_CLK_ENABLE();
    __HAL_RCC_GPIOB_CLK_ENABLE();
    __HAL_RCC_GPIOC_CLK_ENABLE();
    __HAL_RCC_GPIOD_CLK_ENABLE();
    __HAL_RCC_GPIOE_CLK_ENABLE();
    __HAL_RCC_GPIOF_CLK_ENABLE();
    __HAL_RCC_GPIOG_CLK_ENABLE();
}
```

3) 任务配置

```
/* 启动任务函数 */
#define START_TASK_PRIORITY 1
#define START_TASK_STACK_DEPTH 128
TaskHandle_t start_task_handler;
void Start_Task(void *pvParameters);
```

```
/* Task1 任务 配置 */
#define TASK1_PRIORITY 2
#define TASK1_STACK_DEPTH 128
TaskHandle_t task1_handler;
void Task1(void *pvParameters);

/* Task2 任务 配置 */
#define TASK2_PRIORITY 3
#define TASK2_STACK_DEPTH 128
TaskHandle_t task2_handler;
void Task2(void *pvParameters);
```

4) 入口函数

```
QueueHandle_t semaphore_handle;
/**
 * @description: FreeRTOS 入口函数：创建任务函数并开始调度
 * @return {*}
 */
void FreeRTOS_Start(void)
{
    semaphore_handle = xSemaphoreCreateBinary();
    if (semaphore_handle != NULL)
    {
        printf("二值信号量创建成功\r\n");
    }
    xTaskCreate((TaskFunction_t)Start_Task,
                (char *)"Start_Task",
                (configSTACK_DEPTH_TYPE)START_TASK_STACK_DEPTH,
                (void *)NULL,
                (UBaseType_t)START_TASK_PRIORITY,
                (TaskHandle_t *)&start_task_handler);
    vTaskStartScheduler();
}
```

5) 初始任务函数

```
void Start_Task(void *pvParameters)
{
    taskENTER_CRITICAL(); /* 进入临界区 */
    xTaskCreate((TaskFunction_t)Task1,
                (char *)"Task1",
                (configSTACK_DEPTH_TYPE)TASK1_STACK_DEPTH,
                (void *)NULL,
```

```
        (UBaseType_t)TASK1_PRIORITY,  
        (TaskHandle_t *)&task1_handler);  
  
    xTaskCreate((TaskFunction_t)Task2,  
                (char *)"Task2",  
                (configSTACK_DEPTH_TYPE)TASK2_STACK_DEPTH,  
                (void *)NULL,  
                (UBaseType_t)TASK2_PRIORITY,  
                (TaskHandle_t *)&task2_handler);  
    vTaskDelete(NULL);  
    taskEXIT_CRITICAL(); /* 退出临界区 */  
}
```

6) task1 任务函数

```
/**  
 * @description: 释放二值信号量  
 * @param {void *} pvParameters  
 * @return {*}  
 */  
void Task1(void *pvParameters)  
{  
    uint8_t key = 0;  
    BaseType_t err;  
    while (1)  
    {  
        key = Key_Detect();  
        if (key == KEY1_PRESS)  
        {  
            if (semaphore_handle != NULL)  
            {  
                err = xSemaphoreGive(semaphore_handle);  
                if (err == pdPASS)  
                {  
                    printf("信号量释放成功\r\n");  
                }  
                else  
                {  
                    printf("信号量释放失败\r\n");  
                }  
            }  
            vTaskDelay(10);  
        }  
    }  
}
```

7) task2 任务函数

```
/**
 * @description: 获取二值信号量
 * @param {void *} pvParameters
 * @return {*}
 */
void Task2(void *pvParameters)
{
    uint32_t i = 0;
    BaseType_t err;
    while (1)
    {
        /* 一直等待获取信号量 */
        err = xSemaphoreTake(semaphore_handle, portMAX_DELAY);
        if (err == pdTRUE)
        {
            printf("获取信号量成功\r\n");
        }
        else
        {
            printf("已超时%d\r\n", ++i);
        }
    }
}
```

第 17 章 FreeRTOS 内存管理

17.1 FreeRTOS 内存管理简介（了解）

在使用 FreeRTOS 创建任务、队列、信号量等对象时，通常都有动态创建和静态创建的方式。动态方式提供了更灵活的内存管理，而静态方式则更注重内存的静态分配和控制。

如果是动态创建的，那么标准 C 库 malloc() 和 free() 函数有时可用于此目的，但是有以下缺点：

- 它们在嵌入式系统上并不总是可用。
- 它们占用了宝贵的代码空间。
- 它们不是线程安全的。
- 它们不是确定性的（执行函数所需时间将因调用而异）。
- ...

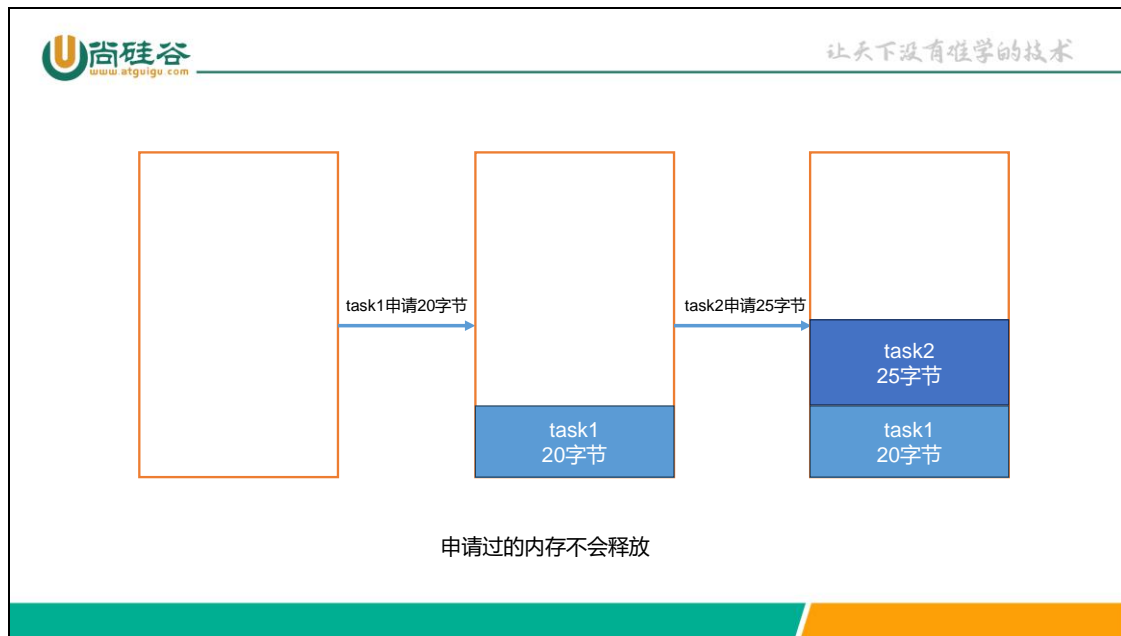
所以更多的时候需要的不是一个替代的内存分配实现。一个嵌入式/实时系统的 RAM 和定时要求可能与另一个非常不同，所以单一的 RAM 分配算法将永远只适用于一个应用程序子集。为了避免此问题，FreeRTOS 将内存分配 API 保留在其可移植层，提供了五种内存管理算法：

- heap_1: 最简单，不允许释放内存。
- heap_2: 允许释放内存，但不会合并相邻的空闲块。
- heap_3: 简单包装了标准 malloc() 和 free(), 以保证线程安全。
- heap_4: 合并相邻的空闲块以避免碎片化。包含绝对地址放置选项。
- heap_5: 如同 heap_4, 能够跨越多个不相邻内存区域的堆。

17.2 FreeRTOS 内存管理算法（熟悉）

17.2.1 heap_1 算法

heap_1 是最简单的实现方式。内存一经分配，它不允许内存再被释放。尽管如此，heap_1.c 还是适用于大量嵌入式应用程序。这是因为许多小型和深度嵌入的应用程序在系统启动时创建了所需的所有任务、队列、信号量等，并在程序的生命周期内使用所有这些对象（直到应用程序再次关闭或重新启动）。任何内容都不会被删除。



17.2.2 heap_2 算法

heap_2 使用最佳适应算法，并且与方案 1 不同，它允许释放先前分配的块，它不将相

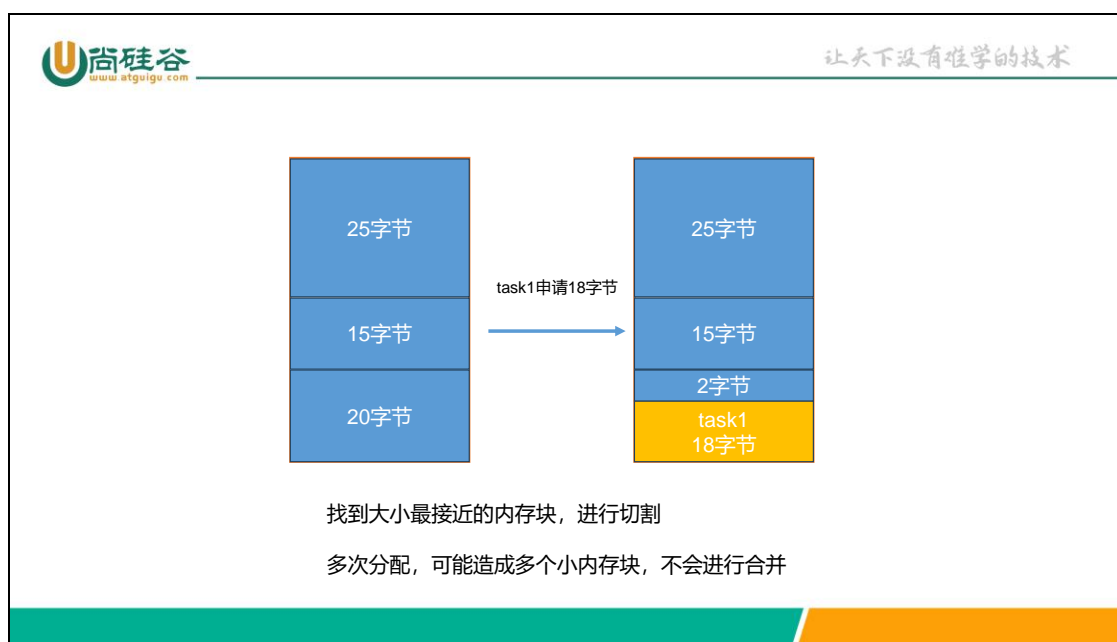
邻的空闲块组合成一个大块。

heap_2.c 适用于许多必须动态创建对象的小型实时系统。

➤ 如果动态地创建和删除任务，且分配给正在创建任务的堆栈大小总是相同的，那么 heap2.c 可以在大多数情况下使用。

➤ 但是，如果分配给正在创建任务的堆栈的大小不是总相同，那么可用的空闲内存可能会被碎片化成许多小块，最终导致分配失败。

heap_2 使用最佳适应算法，该算法在空闲内存中选择与请求的内存大小最接近的块来分配内存。下面是一个简单的例子来说明最佳适应算法：



假设有一个空闲内存，其中包含以下块：

- 大小为 20 字节的空闲块。
- 大小为 15 字节的空闲块。
- 大小为 25 字节的空闲块。

现在有一个任务请求分配 18 字节的内存。最佳适应算法将选择大小为 20 字节的块，因为它与请求的大小最接近。在选择这个块后，分配器可能会将该块分割为两部分，一部分大小为 18 字节，用于任务的内存，另一部分大小为 2 字节，留作未分配的块。

17.2.3 heap_3 算法

heap_3 使用 C 库的 malloc 和 free 函数来进行内存分配和释放。它通过分配固定大小的块来管理内存，这些块的大小在配置 FreeRTOS 时进行定义，不会动态改变。

更多 [Java](#) - [大数据](#) - [前端](#) - [python](#) 人工智能资料下载，可百度访问：[尚硅谷官网](#)

假设我们使用 Heap_3 管理内存，其中块的大小固定为 32 字节。初始时，整个内存被分割成大小为 32 字节的块：

- 块 1（32 字节）。
- 块 2（32 字节）。
- 块 3（32 字节）。

现在，有一个任务请求分配 20 字节的内存。Heap_3 算法将选择块 1，并将其分割成两部分：

- 分配给任务的内存块（20 字节）。
- 剩余未分配的块（12 字节）。

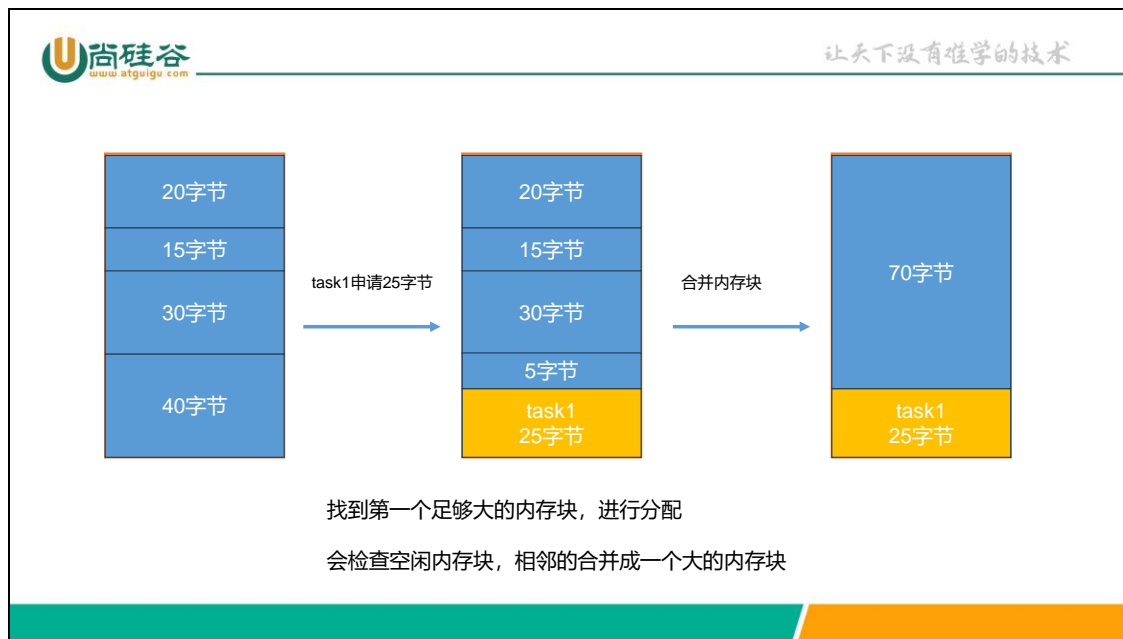
再假设另一个任务请求分配 40 字节的内存。由于没有足够大的块可供分配，heap_3 将返回分配失败的状态。

heap_3 的特点是块大小固定，这样可以简化内存管理。然而，也因为块大小不可变，可能导致内存碎片问题，即一些块可能无法完全被利用，从而浪费了一些内存。

17.2.4 heap_4 算法

heap_4 使用第一适应算法，并且会将相邻的空闲内存块合并成大内存块，减少内存碎片。

第一适应算法会在可用内存块中选择第一个足够大的内存块进行分配。



假设有一个内存块链表，其中包含以下顺序的内存块：

- 大小为 40 字节的块。
- 大小为 30 字节的块。
- 大小为 15 字节的块。
- 大小为 20 字节的块。

如果一个任务需要申请 25 字节的内存，第一适应算法将选择大小为 40 字节的块，因为它是第一个足够大以容纳任务需求的内存块。（如果是 heap_2 的最佳适应算法，会选择 30 字节的块）

17.2.5 heap_5 算法

heap_5 使用与 heap_4 相同的第一适应和内存合并算法，允许堆跨越多个不相邻（非连续）内存区域。适用于内存地址不连续的复杂场景。

17.3 FreeRTOS 内存管理相关 API 函数介绍（熟悉）

内存管理相关函数如下：

函数	描述
void * pvPortMalloc(size_t xWantedSize);	申请内存
void vPortFree(void * pv);	释放内存
size_t xPortGetFreeHeapSize(void);	获取当前空闲内存的大小

17.4 FreeRTOS 内存管理实验（掌握）

17.4.1 实验目标

学习 FreeRTOS 内存管理的函数，观察内存变化情况：

- start_task：用来创建其他 1 个任务。
- task1：用于按键扫描，当 KEY1 按下则申请内存，当 KEY2 按下则释放内存，并打印剩余内存信息。

17.4.2 freertos_demo.c 代码清单

1) 任务配置

```
/* 启动任务函数 */
#define START_TASK_PRIORITY 1
#define START_TASK_STACK_DEPTH 128
TaskHandle_t start_task_handler;
void Start_Task(void *pvParameters);
```

更多 [Java](#) - [大数据](#) - [前端](#) - [python](#) 人工智能资料下载，可百度访问：尚硅谷官网

```
/* Task1 任务 配置 */
#define TASK1_PRIORITY 2
#define TASK1_STACK_DEPTH 128
TaskHandle_t task1_handler;
void Task1(void *pvParameters);
```

2) 入口函数

```
/**
 * @description: FreeRTOS 入口函数：创建任务函数并开始调度
 * @return {*}
 */
void FreeRTOS_Start(void)
{
    xTaskCreate((TaskFunction_t)Start_Task,
                (char *)"Start_Task",
                (configSTACK_DEPTH_TYPE)START_TASK_STACK_DEPTH,
                (void *)NULL,
                (UBaseType_t)START_TASK_PRIORITY,
                (TaskHandle_t *)&start_task_handler);
    vTaskStartScheduler();
}
```

3) 初始任务函数

```
void Start_Task( void * pvParameters )
{
    taskENTER_CRITICAL();           /* 进入临界区 */
    xTaskCreate((TaskFunction_t) Task1,
                (char *) "Task1",
                (configSTACK_DEPTH_TYPE) TASK1_STACK_DEPTH,
                (void *) NULL,
                (UBaseType_t) TASK1_PRIORITY,
                (TaskHandle_t *) &task1_handler );
    vTaskDelete(NULL);
    taskEXIT_CRITICAL();           /* 退出临界区 */
}
```

4) task1 任务函数

```
/**
 * @description: 申请及释放内存，并显示空闲内存大小
 * @param {void *} pvParameters
 * @return {*}
 */
```

```
void Task1(void * pvParameters)
{
    uint8_t key = 0, t = 0;
    uint8_t * buf = NULL;
    while(1)
    {
        key = Key_Detect();
        if(key == KEY1_PRESS)
        {
            /* 申请内存 */
            buf = pvPortMalloc(30);
            if(buf != NULL)
            {
                printf("申请内存成功\r\n");
            }
            else
            {
                printf("申请内存失败\r\n");
            }
        }
        else if(key == KEY2_PRESS)
        {
            if(buf != NULL)
            {
                /* 释放内存 */
                vPortFree(buf);
                printf("释放内存\r\n");
            }
        }
        if(t++ > 50)
        {
            t = 0;
            printf("剩余的空闲内存大小=%d\r\n", xPortGetFreeHeapSize());
        }
        vTaskDelay(10);
    }
}
```