RTOS入门

1，裸机与RTOS介绍（了解）

裸机

简介：裸机又称为前后台系统，前台系统指的中断服务函数，后台系统指的大循环，即应用程序

例子：打游戏和回复信息，需要打完游戏才可回复信息，或者回复完信息才可打游戏

特点

- 1，实时性差　(应用程序) 轮流执行
- 2，delay　空等待，CPU不执行其他代码（浪费资源）
- 3，结构臃肿　实现功能都放在无限循环

RTOS

简介：RTOS全称为：Real Time OS，就是实时操作系统，强调的是：实时性

例子

打游戏和回复信息，不需要等某一件事做完，可每间隔1ms(一个时间片时钟节拍)，然后交替做这两件事，因为速度很快，从宏观的意义上来看，类似同步执行！

特点

- 1，分而治之　实现功能划分为多个任务
- 2，延时函数　不会空等待，会让出CPU的使用权给其他任务，即任务调度　子主题
- 3，抢占式　高优先级任务抢占低优先级任务
- 4，任务堆栈　每个任务都有自己的栈空间，用于保存局部变量以及任务的上下文信息

注意1：中断可以打断任意任务

注意2：任务可以同等优先级

问题：如果高优先级的任务一直在运行，会怎么样？　会一直运行，使得低优先级任务无法运行

2，FreeRTOS简介（了解）

简介：FreeRTOS 是一个嵌入式实时操作系统

特点

- 免费开源　商业产品中使用，无潜在商业风险，无须担心
- 可裁剪　使得FreeRTOS的核心代码只有9000行左右
- 简单　简单易用，可移植性非常好
- 优先级不限　任务优先级分配没有限制，多任务可同一优先级
- 任务不限　可创建的实时任务数量没有软件限制
- 支持抢占/协程/时间片　支持抢占式，协程式、时间片流转任务调度
- … …

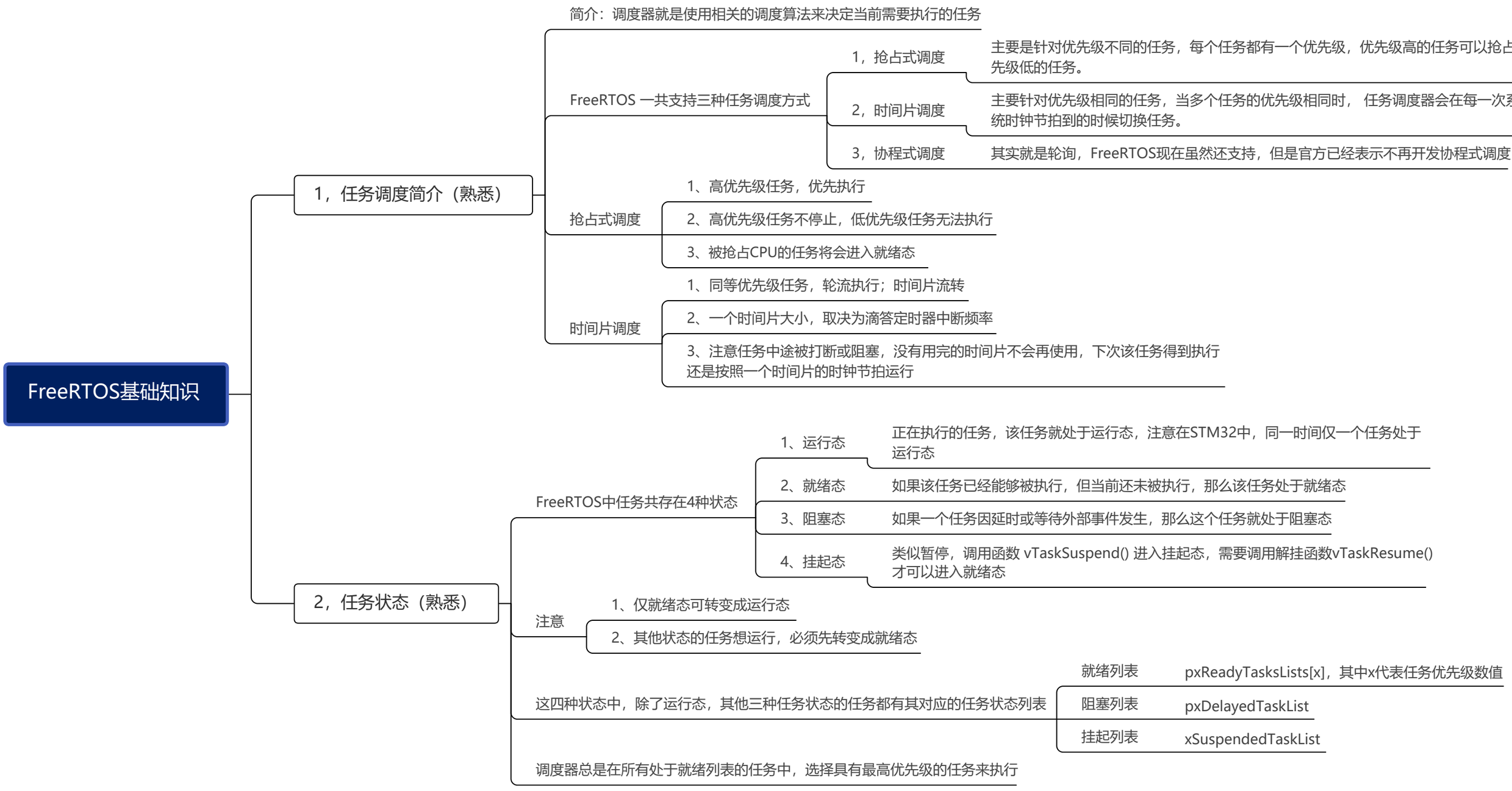
学习资料获取

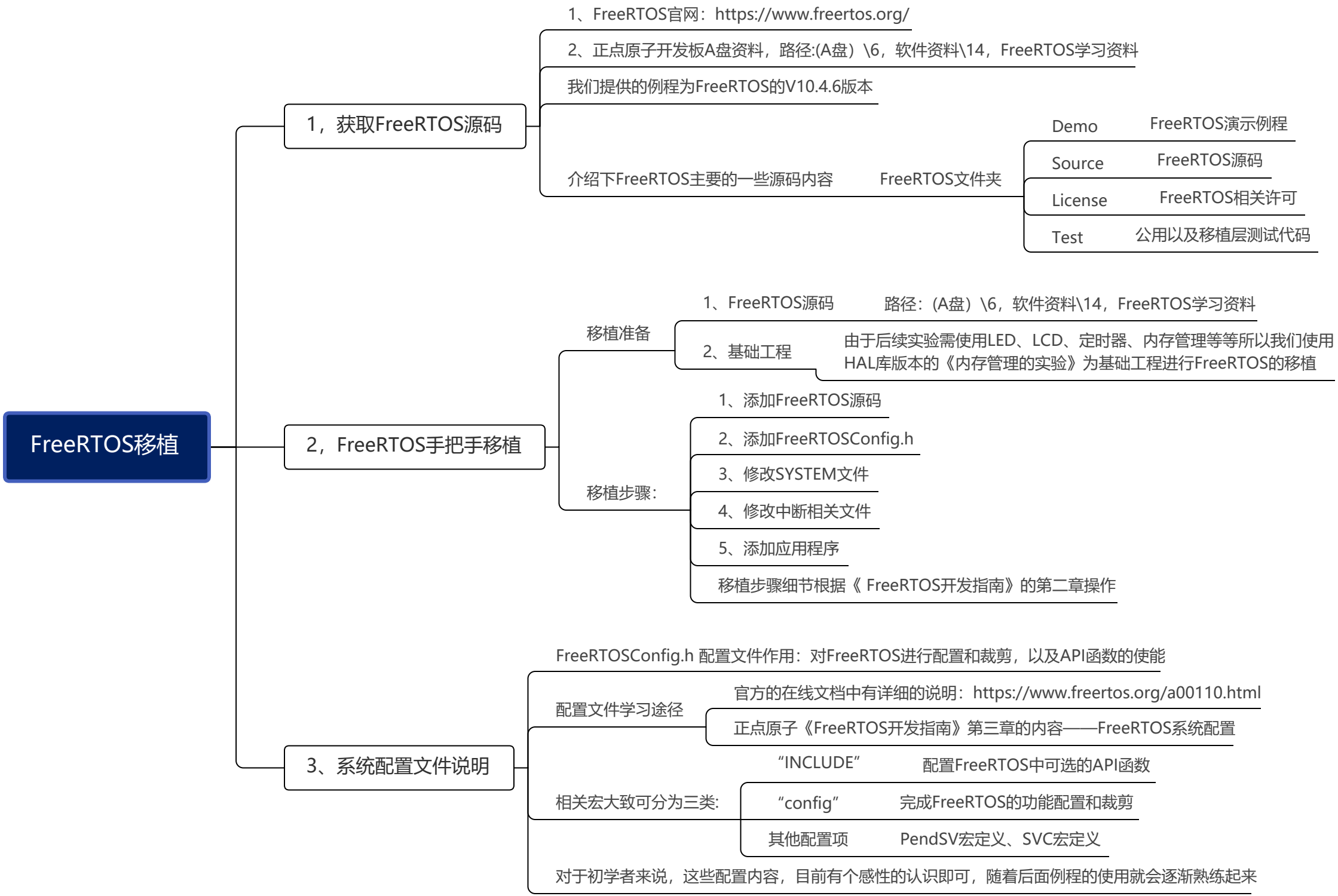
1、FreeRTOS官网：https://www.freertos.org/

- 英文文档（对英语较差的同学不友好）
- 文档较少（相对于UCOS来说文档比较少）

2、正点原子学习资料

- 教程文档　共430+页的详细教程文档，从基础移植，API函数介绍，到任务创建、队列、信号量…
- 程序源码　共25个例程源码，将FreeRTOS主要功能逐一验证
- 视频资料　正在录制当中… …


FreeRTOS基础知识
1，任务调度简介（熟悉）
简介：调度器就是使用相关的调度算法来决定当前需要执行的任务
FreeRTOS 一共支持三种任务调度方式
1，抢占式调度
主要是针对优先级不同的任务，每个任务都有一个优先级，优先级高的任务可以抢占优先级低的任务。
2，时间片调度
主要针对优先级相同的任务，当多个任务的优先级相同时， 任务调度器会在每一次系统时钟节拍到的时候切换任务。
3，协程式调度
其实就是轮询，FreeRTOS现在虽然还支持，但是官方已经表示不再开发协程式调度
抢占式调度
1、高优先级任务，优先执行
2、高优先级任务不停止，低优先级任务无法执行
3、被抢占CPU的任务将会进入就绪态
时间片调度
1、同等优先级任务，轮流执行；时间片流转
2、一个时间片大小，取决为滴答定时器中断频率
3、注意任务中途被打断或阻塞，没有用完的时间片不会再使用，下次该任务得到执行还是按照一个时间片的时钟节拍运行
2，任务状态（熟悉）
FreeRTOS中任务共存在4种状态
1、运行态
正在执行的任务，该任务就处于运行态，注意在STM32中，同一时间仅一个任务处于运行态
2、就绪态
如果该任务已经能够被执行，但当前还未被执行，那么该任务处于就绪态
3、阻塞态
如果一个任务因延时或等待外部事件发生，那么这个任务就处于阻塞态
4、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
注意
1、仅就绪态可转变成运行态
2、其他状态的任务想运行，必须先转变成就绪态
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
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


FreeRTOS移植
1，获取FreeRTOS源码
1、FreeRTOS官网：https://www.freertos.org/
2、正点原子开发板A盘资料，路径:(A盘）\6，软件资料\14，FreeRTOS学习资料
我们提供的例程为FreeRTOS的V10.4.6版本
介绍下FreeRTOS主要的一些源码内容
FreeRTOS文件夹
Demo
FreeRTOS演示例程
Source
FreeRTOS源码
License
FreeRTOS相关许可
Test
公用以及移植层测试代码
2，FreeRTOS手把手移植
移植准备
1、FreeRTOS源码
路径：(A盘）\6，软件资料\14，FreeRTOS学习资料
2、基础工程
由于后续实验需使用LED、LCD、定时器、内存管理等等所以我们使用HAL库版本的《内存管理的实验》为基础工程进行FreeRTOS的移植
移植步骤：
1、添加FreeRTOS源码
2、添加FreeRTOSConfig.h
3、修改SYSTEM文件
4、修改中断相关文件
5、添加应用程序
移植步骤细节根据《 FreeRTOS开发指南》的第二章操作
3、系统配置文件说明
FreeRTOSConfig.h 配置文件作用：对FreeRTOS进行配置和裁剪，以及API函数的使能
配置文件学习途径
官方的在线文档中有详细的说明：https://www.freertos.org/a00110.html
正点原子《FreeRTOS开发指南》第三章的内容——FreeRTOS系统配置
相关宏大致可分为三类:
“INCLUDE”
配置FreeRTOS中可选的API函数
“config”
完成FreeRTOS的功能配置和裁剪
其他配置项
PendSV宏定义、SVC宏定义
对于初学者来说，这些配置内容，目前有个感性的认识即可，随着后面例程的使用就会逐渐熟练起来

- FreeRTOS任务创建与删除
  - 任务创建和删除的API函数解析
    - 1、动态创建任务其内部实现
      - 1、 申请堆栈内存（返回首地址）
      - 2、 申请任务控制块内存（返回首地址）
      - 3、 把前面申请的堆栈地址，赋值给控制块的堆栈成员
      - 4、 调用prvInitialiseNewTask 初始化任务控制块中的成员
        - 1、初始化堆栈为0xa5（可选）
        - 2、记录栈顶，保存在pxTopOfStack
        - 3、保存任务名字到pxNewTCB->pcTaskName[x]中
        - 4、保存任务优先级到pxNewTCB->uxPriority
        - 5、设置状态列表项的所属控制块，设置事件列表项的值
        - 6、列表项的插入是从小到大插入，所以这里将越高优先级的任务他的事件列表项值设置越小，这样就可以拍到前面
        - 7、调用pxPortInitialiseStack初始化任务堆栈，用于保存当前任务上下文寄存器信息，已备后续任务切换使用
        - 8、将任务句柄等于任务控制块
      - 5、 调用prvAddNewTaskToReadyList 添加新创建任务到就绪列表中
        - 1、记录任务数量uxCurrentNumberOfTasks++
        - 2、判断新创建的任务是否为第一个任务
          - 如果创建的是第一个任务，初始化任务列表prvInitialiseTaskLists
          - 如果创建的不是第一个任务，并且调度器还未开始启动，比较新任务与正在执行的任务优先级大小，新任务优先级大的话，将当前控制块重新指向新的控制块
        - 3、将新的任务控制块添加到就绪列表中，使用函数prvAddTaskToReadyList
          - 将uxTopReadyPriority相应bit置一，表示相应优先级有就绪任务，比如任务优先级为5，就将该变量的位5置一，方便后续任务切换判断，对应的就绪列表是否有任务存在
          - 将新创建的任务插入对应的就绪列表末尾
        - 4、如果调度器已经开始运行，并且新任务的优先级更大的话，进行一次任务切换
    - 2、删除任务的内部实现
      - 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
        - 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
        - 删除其他任务，当前任务数量--　　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、删除的任务为其他任务则直接释放内存prvDeleteTCB( )
      - 5、调度器正在运行且删除任务自身，则需要进行一次任务切换
    - 3、静态创建任务其内部实现
      - 1、获取控制块内存（首地址）
      - 2、获取堆栈内存（首地址）
      - 3、标记使用的静态的方式申请的TCB和堆栈内存
      - 4、调用prvInitialiseNewTask 初始化任务块，并将控制块信息返回给任务句柄，以便后续返回句柄信息
      - 5、调用prvAddNewTaskToReadyList 添加新创建任务到就绪列表中
  - 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
    - 动态方式创建任务：xTaskCreate()
    - 静态方式创建任务：xTaskCreateStatic()
    - 删除任务：vTaskDelete()
    - 动态静态创建区别
      - 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
    - 实现动态创建任务流程
      - 使用动态创建任务，需将宏configSUPPORT_DYNAMIC_ALLOCATION 配置为 1
      - 1、定义函数入口参数
      - 2、编写任务函数
    - 静态创建任务流程
      - 使用静态创建任务，需将宏configSUPPORT_STATIC_ALLOCATION 配置为 1
      - 1、定义空闲任务&定时器任务的任务堆栈及TCB
      - 2、实现两个接口函数
        - vApplicationGetIdleTaskMemory( )
        - vApplicationGetTimerTaskMemory ( )
      - 3、编写任务函数
    - 删除任务流程
      - 使用删除任务函数，需将宏INCLUDE_vTaskDelete 配置为 1
      - 1、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）
      - 2、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，任务创建和删除（动态方法）（掌握）
    - 1、实验目的：学会 xTaskCreate( ) 和 vTaskDelete( ) 的使用
    - 2、将设计四个任务：start_task、task1、task2、task3
      - start_task：用来创建其他的两个任务
      - task1：实现LED0每500ms闪烁一次
      - task2：实现LED1每500ms闪烁一次
      - task3：判断按键KEY0是否按下，按下则删掉task1
  - 3，任务创建和删除（静态方法）（掌握）
    - 1、实验目的：学会 xTaskCreateStatic( ) 和 vTaskDelete( ) 的使用
    - 2、将设计四个任务：start_task、task1、task2、task3
      - start_task：用来创建其他的两个任务
      - task1：实现LED0每500ms闪烁一次
      - task2：实现LED1每500ms闪烁一次
      - task3：判断按键KEY0是否按下，按下则删掉task1
  - 4、需注意
    - 1、在实际的应用中，动态方式创建任务是比较常用的，除非有特殊的需求，一般都会使用动态方式创建任务
    - 2、动态创建相对简单，更为常用
    - 3、静态创建：可将任务堆栈放置在特定的内存位置，并且无需关心对内存分配失败的处理
    - 4、临界区保护，保护那些不想被打断的程序段，关闭freertos所管理的中断，中断无法打断，滴答中断和PendSV中断无法进行不能实现任务调度

- FreeRTOS任务挂起与恢复
  - 1，任务的挂起与恢复的API函数（熟悉）
    - 任务挂起与恢复API函数解析
      - 1、任务挂起函数vTaskSuspend( )
        - 1、需将宏INCLUDE_vTaskSuspend 配置为 1
        - 2、根据任务句柄获取任务控制块，如果任务句柄为NULL，表示挂起任务自身
        - 3、将要挂起的任务从相应的状态列表和事件列表中移除
        - 4、将待挂起任务的任务状态列表向插入到挂起态任务列表末尾
        - 5、判断任务调度器是否运行，在运行，更新下一次阻塞时间，防止被挂起任务为下一次阻塞超时任务
        - 6、如果挂起的是任务自身，且调度器正在运行，需要进行一次任务切换；
调度器没有运行，判断挂起任务数是否等于任务总数，是：当前控制块赋值为NULL，
否：寻找下一个最高优先级任务
      - 2、（任务中调用）任务恢复函数vTaskResume( )
        - 1、需将宏INCLUDE_vTaskSuspend 配置为 1
        - 2、恢复任务不能是正在运行任务
        - 3、判断任务是否在挂起列表中，是：就会将该任务在挂起列表中移除， 将该任务添加到就绪列表中
        - 4、判断恢复的任务优先级是否大于当前正在运行的 是的话执行任务切换
      - 3、（中断中调用）任务恢复函数xTaskResumeFromISR( )
        - 1、关闭freertos可管理中断，防止被其他的中断打断，并返回关闭前basepri寄存器的值
        - 2、判断是否有挂起任务
          - 有挂起任务　检查调度器是否被挂起
            - 调度器未挂起
              - 1、判断恢复的这个任务优先级是否大于正在执行的任务是的话将xYieldRequired标记为pdTRUE，表示需要进行一次任务切换
              - 2、将被恢复的任务从挂起列表中移除
              - 3、插入到就绪列表
            - 调度器挂起　如果调度器被挂起了，就将恢复的任务插入等待就绪列表，直到调度器被恢复再进行任务的处理
          - 无挂起任务　不需操作
        - 3、将前面保存的basepri的值，恢复回来
        - 4、返回xYieldRequired的值 用于决定是否需要进行任务切换
    - 挂起任务　void vTaskSuspend(TaskHandle_t xTaskToSuspend)
      - 此函数用于挂起任务，使用时需将宏 INCLUDE_vTaskSuspend 配置为 1。
      - 无论优先级如何，被挂起的任务都将不再被执行，直到任务被恢复 。
      - 注意：当传入的参数为NULL，则代表挂起任务自身（当前正在运行的任务）
    - 恢复被挂起的任务　void vTaskResume(TaskHandle_t xTaskToResume)
      - 注意宏：INCLUDE_vTaskSuspend必须定义为 1
      - 函数：vTaskResume无返回值
      - 使用：在任务中恢复被挂起函数
      - 注意：任务无论被 vTaskSuspend() 挂起多少次，只需用 vTakResume() 恢复一次，就可以继续运行。
    - 在中断中恢复被挂起的任务　BaseType_t xTaskResumeFromISR(TaskHandle_t xTaskToResume)
      - 注意宏：INCLUDE_vTaskSuspend 和 INCLUDE_xTaskResumeFromISR 必须定义为 1
      - 使用：中断中恢复被挂起函数
      - 函数：xTaskResumeFromISR返回值描述
        - 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
    - 挂起简介：挂起任务类似暂停，可恢复； 删除任务，无法恢复，类似“人死两清”
    - 函数带有“FromISR”后缀的，为中断服务函数中专用API函数
  - 2，任务挂起与恢复实验（掌握）
    - 1、实验目的：学会 使用FreeRTOS中的任务挂起与恢复相关API函数：vTaskSuspend( )、vTaskResume( )、xTaskResumeFromISR( )
    - 实验设计：将设计四个任务：start_task、task1、task2、task3
    - start_task　用来创建其他的三个任务
    - task1　实现LED0每500ms闪烁一次
    - task2　实现LED1每500ms闪烁一次
    - task3　判断按键按下逻辑，KEY0按下，挂起task1，按下按键KEY1在任务中恢复task1
按下KEY2，在中断中恢复task1（外部中断线实现）

FreeRTOS中断管理

1，什么是中断？（了解）

简介：让CPU打断正常程序的运行，转而紧急处理的事件（程序），就叫中断

中断执行机制，可简单概括为三步：

1，中断请求：外设产生中断请求（GPIO外部中断、定时器中断等）

2，响应中断：CPU停止执行当前程序，转而去执行中断处理程序（ISR）

3，退出中断：执行完毕，返回被打断的程序处，继续往下执行

2，中断优先级分组设置（熟悉）

ARM Cortex-M 使用了 8 位宽的寄存器来配置中断的优先等级，这个寄存器就是中断优先级配置寄存器，所以中断优先级配置范围在0~255

STM32，只用了中断优先级配置寄存器的高4位 [7 : 4]，所以STM32提供了最大16级的中断优先等级

STM32 的中断优先级可以分为抢占优先级和子优先级

抢占优先级： 抢占优先级高的中断可以打断正在执行但抢占优先级低的中断

子优先级：当同时发生具有相同抢占优先级的两个中断时，子优先级数值小的优先执行

共有5种优先级分组分配方式

NVIC_PriorityGroup_0	0bit 用于抢占优先级，4bit 用于子优先级
NVIC_PriorityGroup_1	1bit 用于抢占优先级 3bit 用于子优先级
NVIC_PriorityGroup_2	2bit 用于抢占优先级 2bit 用于子优先级
NVIC_PriorityGroup_3	3bit 用于抢占优先级 1bit 用于子优先级
NVIC_PriorityGroup_4	4bit 用于抢占优先级 0bit 用于子优先级

相关说明可查看官网：https://www.freertos.org/RTOS-Cortex-M3-M4.html

特点

1、低于configMAX_SYSCALL_INTERRUPT_PRIORITY优先级的中断里才允许调用FreeRTOS 的API函数

2、建议将所有优先级位指定为抢占优先级位，方便FreeRTOS管理

3、中断优先级数值越小越优先，任务优先级数值越大越优先

3，中断相关寄存器（熟悉）

三个系统中断优先级配置寄存器，分别为分别为 SHPR1、 SHPR2、 SHPR3

通过SHPR3将PendSV和SysTick的中断优先级设置为最低优先级，保证系统任务切换不会阻塞系统其他中断的响应

三个中断屏蔽寄存器，分别为 PRIMASK、 FAULTMASK 和BASEPRI

FreeRTOS所使用的中断管理就是利用的BASEPRI这个寄存器

BASEPRI：屏蔽优先级低于某一个阈值的中断

比如： BASEPRI设置为0x50，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

学习资料参考《Cortex M3权威指南(中文)》手册

4，FreeRTOS中断管理实验（掌握）

实验目的：学会使用FreeRTOS的中断管理

本实验会使用两个定时器，一个优先级为4，一个优先级为6，注意：系统所管理的优先级范围：5~15，现象：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。

2、实验设计：将设计2个任务：start_task、task1

start_task	用来创建task1任务
task1	中断测试任务，任务中将调用关中断和开中断函数来体现对中断的管理作用！


临界区保护及调度器的挂起和恢复
1，临界段代码保护简介（熟悉）
什么是临界段：临界段代码也叫做临界区，是指那些必须完整运行，不能被打断的代码段
适用场合
1，外设：需严格按照时序初始化的外设：IIC、SPI等等
2，系统：系统自身需求
3，用户：用户需求
2，临界段代码保护函数介绍（掌握）
FreeRTOS 在进入临界段代码的时候需要关闭中断，当处理完临界段代码以后再打开中断
有关于临界区的API函数
任务级进入临界段：taskENTER_CRITICAL()
任务级退出临界段：taskEXIT_CRITICAL()
中断级进入临界段：taskENTER_CRITICAL_FROM_ISR()
中断级退出临界段：taskEXIT_CRITICAL_FROM_ISR()
特点
1、成对使用
2、支持嵌套（与单纯的开关中断最大的区别）
3、尽量保持临界段耗时短
强悍！临界区是直接屏蔽了中断，系统任务调度靠中断，ISR也靠中断
3，任务调度器的挂起和恢复（熟悉）
挂起任务调度器， 只单纯挂起调度器使任务无法调度，但中断正常
调度器挂起和恢复相关API函数
挂起任务调度器：vTaskSuspendAll()
恢复任务调度器：xTaskResumeAll()
特点
1、与临界区不一样的是，挂起任务调度器，未关闭中断；
2、它仅仅是防止了任务之间的资源争夺，中断照样可以直接响应；
3、挂起调度器的方式，适用于临界区位于任务与任务之间；既不用去延时中断，又可以做到临界区的安全
调度器挂起和恢复API函数
挂起任务调度器：vTaskSuspendAll()
调用一次挂起调度器，该变量uxSchedulerSuspended就加一
变量uxSchedulerSuspended的值，将会影响Systick触发PendSV中断，即影响任务调度
恢复任务调度器：xTaskResumeAll()
调用一次恢复调度器，该变量uxSchedulerSuspended就减一
如果等于0，则允许调度
1、将所有在xPendingReadyList中的任务移到对应的就绪链表中
2、移除等待就绪列表中的列表项，恢复至就绪列表，直到xPendingReadyList列表为空

FreeRTOS列表和列表项

1，列表和列表项的简介（熟悉）

- 列表简介：列表是 FreeRTOS 中的一个数据结构，概念上和链表有点类似，列表被用来跟踪 FreeRTOS中的任务。
- 列表项简介：列表项就是存放在列表中的项目
- 特点：
  - 列表相当于链表，列表项相当于节点，FreeRTOS 中的列表是一个双向环形链表
  - 列表的特点：列表项间的地址非连续的，是人为的连接到一起的。列表项的数目是由后期添加的个数决定的，随时可以改变
  - 数组的特点：数组成员地址是连续的，数组在最初确定了成员数量后期无法改变
- 在OS中任务的数量是不确定的，并且任务状态是会发生改变的，所以非常适用列表(链表)这种数据结构
- 列表结构体
- 列表项结构体
- 迷你列表项结构体

2，列表相关API函数介绍（掌握）

- 列表初始化函数vListInitialise( )
  1. 1、初始化时，列表中只有 xListEnd，因此 pxIndex 指向 xListEnd
  2. 2、xListEnd 的值初始化为最大值，用于列表项升序排序时，排在最后
  3. 3、初始化时，列表中只有 xListEnd，因此上一个和下一个列表项都为 xListEnd 本身
  4. 4、初始化时，列表中的列表项数量为 0（不包含 xListEnd）
- 列表项初始化函数vListInitialiseItem( )
  1. 1、初始化时，列表项不属于任何一个列表，所以为空
- 列表项插入函数vListInsert( )
  1. 1、获取新插入的列表项的值
  2. 2、判断新插入的列表项数值大小
     - 如果数值等于末尾列表项的数值。就插入到末尾列表项前面
     - 否则遍历列表中的列表项，找到插入的位置
  3. 3、将列表项插入前面所找到的位置
  4. 4、更新待插入列表项所在列表
  5. 5、更新列表中列表项的数量
- 末尾列表项插入函数vListInsertEnd( )
  1. 1、获取列表 pxIndex 指向的列表项
  2. 2、将待插入的列表项插入到 pxIndex所指向的列表项前面
  3. 3、更新待插入列表项的所在列表
  4. 4、更新列表中列表项的数量
- 列表项移除函数uxListRemove( )
  1. 1、获取所要移除的列表项的所在列表
  2. 2、从列表中移除列表项
  3. 3、如果 pxIndex 正指向待移除的列表项，将其指向带移除的上一个列表项
  4. 4、将待移除列表项的所在列表指针清空
  5. 5、列表的列表项数目减一
  6. 6、返回列表项移除后列表中列表项的数量

3，列表项的插入和删除实验（掌握）

1、实验目的：学会对FreeRTOS 列表和列表项的操作函数使用，并观察运行结果和理论分析是否一致

2、实验设计：将设计三个任务：start_task、task1、task2

start_task	用来创建其他的2个任务
task1	实现LED0每500ms闪烁一次，用来提示系统正在运行
task2	调用列表和列表项相关API函数，并且通过串口输出相应的信息，进行观察

FreeRTOS任务调度

启动第一个任务流程：

创建开始任务：start_tsak

启动任务调度器：vTaskStartScheduler

1.创建空闲任务：prvIdleTask
2.创建软件定时器任务：xTimerCreateTimerTask
3.关中断（在启动第一个任务时开启）
4.初始化一些全局变量...
5.初始化任务运行时间统计功能的时基定时器
6.调用函数xPortStartScheduler完成启动任务调度器
1.配置 PendSV 和 SysTick 的中断优先级为最低优先级
2.调用函数 vPortSetupTimerInterrupt()配置 SysTick
3.调用函数 prvEnableVFP()使能 FPU
4.将 FPCCR 寄存器的[31:30]置 1，这样在进出异常时， FPU 的相关寄存器就会自动地保存和恢复
5.调用函数 prvStartFirstTask()启动第一个任务
1.复位MSP初始值
2.使能中断
3.触发SVC中断
获取当前优先级最高的任务控制块pxCurrentTCB
将该任务的寄存器值出栈至CPU寄存器中
设置PSP
返回r14　　执行第一个任务函数

任务切换流程

触发PendSV中断，主要途径

1、滴答定时器中断触发
2、调用FreeRTOS的API函数触发，如：portYIELD( )

PendSV中断函数触发，执行任务切换

当前的psp是正在运行的任务的栈指针，读取当前psp进程指针，存入r0
压栈（保存现场）
获取当前最高优先级任务的任务控制块
出栈（恢复现场）
更新切换后的任务的的栈指针给PSP
bx r14 执行新任务函数

1，开启任务调度器（熟悉）

函数：vTaskStartScheduler()，用于启动任务调度器，任务调度器启动后， FreeRTOS便会开始进行任务调度

1、创建空闲任务
2、如果使能软件定时器，则创建定时器任务
3、关闭中断，防止调度器开启之前或过程中，受中断干扰，会在运行第一个任务时打开中断
4、初始化全局变量，并将任务调度器的运行标志设置为已运行
5、初始化任务运行时间统计功能的时基定时器
6、调用函数 xPortStartScheduler()

xPortStartScheduler()该函数用于完成启动任务调度器中与硬件架构相关的配置部分，以及启动第一个任务

1、检测用户在 FreeRTOSConfig.h 文件中对中断的相关配置是否有误
2、配置 PendSV 和 SysTick 的中断优先级为最低优先级
3、调用函数 vPortSetupTimerInterrupt()配置 SysTick
4、初始化临界区嵌套计数器为 0
5、调用函数 prvEnableVFP()使能 FPU
6、调用函数 prvStartFirstTask()启动第一个任务

2，启动第一个任务（掌握）

想象下应该如何启动第一个任务？

1、找到优先级最高的任务
2、将该任务的寄存器值恢复到CPU寄存器中

2.1，prvStartFirstTask () 该函数用于初始化启动第一个任务前的环境，主要是重新设置MSP 指针，并使能全局中断

1、什么是MSP指针？
主堆栈指针（MSP）：它由 OS 内核、异常服务例程以及所有需要特权访问的应用程序代码来使用。
进程堆栈指针（PSP）：用于常规的应用程序代码（不处于异常服用例程中时）。
注意：在FreeRTOS中，中断使用MSP（主堆栈），中断以外使用PSP（进程堆栈）

2、为什么是 0xE000ED08?
因为需从 0xE000ED08 获取向量表的偏移，为啥要获得向量表呢？因为向量表的第一个是 MSP 指针！
取 MSP 的初始值的思路是先根据向量表的位置寄存器 VTOR (0xE000ED08) 来获取向量表存储的地址；在根据向量表存储的地址，来访问第一个元素，也就是初始的 MSP

2.2，vPortSVCHandler () /* SVC中断服务函数 */

1. 通过 pxCurrentTCB 获取优先级最高的就绪态任务的任务栈地址，优先级最高的就绪态任务是系统将要运行的任务 。
2. 通过任务的栈顶指针，将任务栈中的内容出栈到 CPU 寄存器中，任务栈中的内容在调用任务创建函数的时候，已初始化，然后设置 PSP 指针 。
3. 通过往 BASEPRI 寄存器中写 0，允许中断。
4、执行 bx R14，告诉处理器 ISR 完成，需要返回，此刻处理器便会使用 PSP 做为堆栈指针，进行出栈操作，将xPSR、PC、LR、R12、R3~R0 出栈，初始化的时候，PC被我们赋值成为了执行任务的函数的入口，所以呢，就正常跳入到了优先级最高的就绪状态的第一个任务的入口函数了

注意
返回R14 R14 是链接寄存器 LR，在 ISR 中（此刻我们在 SVC 的 ISR 中），它记录了异常返回值 EXC_RETURN
SVC中断只在启动第一次任务时会调用一次，以后均不调用

3，任务切换（掌握）

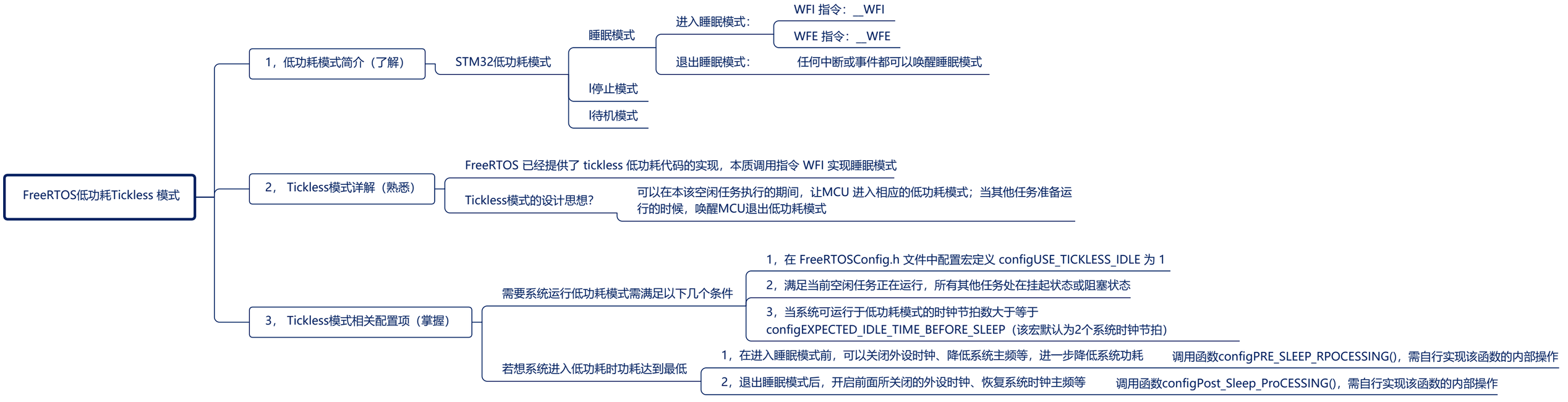
任务切换的本质：就是CPU寄存器的切换。

假设当由任务A切换到任务B时，主要分为两步：
第一步：需暂停任务A的执行，并将此时任务A的寄存器保存到任务堆栈，这个过程叫做保存现场；
第二步：将任务B的各个寄存器值（被存于任务堆栈中）恢复到CPU寄存器中，这个过程叫做恢复现场；
对任务A保存现场，对任务B恢复现场，这个整体的过程称之为：上下文切换

通过函数vTaskSwitchContext（）查找最高优先级任务
利用函数taskSELECT_HIGHEST_PRIORITY_TASK( )，前导置零指令找到最高优先级
利用函数listGET_OWNER_OF_NEXT_ENTRY( ) ，获取最高优先级的任务句柄

任务切换的整个过程：

1.mrs r0, psp　　把psp存到r0，当前的psp是任务A的栈指针，这时的psp指向的是任务栈A中的R0
2. ldr r3, =pxCurrentTCB
ldr r2, [r3]　　获取当前运行任务的栈顶地址即R2保存的栈顶地址，注意R3等于pxCurrentTCB的地址
3.stmdb r0!, {r4-r11, r14}　　压栈，从上往下压，将r0的值，当压栈的起始地址，开始压栈（保存现场）
4.str r0, [r2]　　将r0的值(前面的底部地址)，写到r2地址所指向的内存中（即栈顶地址指向的内存，pxTopOfStack中）
5.stmdb sp!, {r0, r3}　　/* 把R0和R3的值压入MSP */
6.bl vTaskSwitchContext　　通过该函数，获取下一个执行任务的任务控制块，赋值给pxCurrentTCB
7.ldmia sp!, {r0, r3}　　从sp(MSP)恢复r3，即把r3恢复成&pxCurrentTCB。后续就可以利用r3得到新的任务控制块了。
8.ldmia r0!, {r4-r11, r14}　　出栈，以寻址地址开始，从下往上进行出栈，将保存在这些地址的值恢复到寄存器里边去
9.msr psp, r0
bx r14　　①将r0更新给psp线程堆栈
②返回线程模式，执行新任务


FreeRTOS低功耗Tickless 模式
1，低功耗模式简介（了解）
STM32低功耗模式
睡眠模式
进入睡眠模式：
WFI 指令：__WFI
WFE 指令：__WFE
退出睡眠模式：
任何中断或事件都可以唤醒睡眠模式
l停止模式
l待机模式
2， Tickless模式详解（熟悉）
FreeRTOS 已经提供了 tickless 低功耗代码的实现，本质调用指令 WFI 实现睡眠模式
Tickless模式的设计思想?
可以在本该空闲任务执行的期间，让MCU 进入相应的低功耗模式；当其他任务准备运行的时候，唤醒MCU退出低功耗模式
3， Tickless模式相关配置项（掌握）
需要系统运行低功耗模式需满足以下几个条件
1，在 FreeRTOSConfig.h 文件中配置宏定义 configUSE_TICKLESS_IDLE 为 1
2，满足当前空闲任务正在运行，所有其他任务处在挂起状态或阻塞状态
3，当系统可运行于低功耗模式的时钟节拍数大于等于 configEXPECTED_IDLE_TIME_BEFORE_SLEEP（该宏默认为2个系统时钟节拍）
若想系统进入低功耗时功耗达到最低
1，在进入睡眠模式前，可以关闭外设时钟、降低系统主频等，进一步降低系统功耗
调用函数configPRE_SLEEP_RPOCESSING()，需自行实现该函数的内部操作
2，退出睡眠模式后，开启前面所关闭的外设时钟、恢复系统时钟主频等
调用函数configPost_Sleep_ProCESSING()，需自行实现该函数的内部操作

FreeRTOS内存管理

1，FreeRTOS内存管理简介（了解）

在使用 FreeRTOS 创建任务、队列、信号量等对象的时，一般都提供了两种方法：

① 动态方法创建：自动地从 FreeRTOS 管理的内存堆中申请创建对象所需的内存，并且在对象删除后，可将这块内存释放回FreeRTOS管理的内存堆

② 静态方法创建：需用户提供各种内存空间，并且使用静态方式占用的内存空间一般固定下来了，即使任务、队列等被删除后，这些被占用的内存空间也没有其他用途

总结：动态方式管理内存相比与静态方式，更加灵活。

为啥不用标准的 C 库自带的内存管理算法?

1.占用大量的代码空间 不适合用在资源紧缺的嵌入式系统中

2.没有线程安全的相关机制

3.运行有不确定性，每次调用这些函数时花费的时间可能都不相同

4.内存碎片化

5.... ...

6.因此，FreeRTOS 提供了动态内存管理的方法，并且针对不同的嵌入式系统，提供了多种内存管理算法！

2，FreeRTOS内存管理算法（熟悉）

FreeRTOS提供了5种动态内存管理算法，分别为： heap_1、heap_2、heap_3、heap_4、heap_5 。

heap_1：只实现了pvPortMalloc，没有实现vPortFree；也就是说，它只能申请内存，无法释放内存

heap_1：如果你的工程，创建好的任务、队列、信号量等都不需要被删除，那么可以使用heap_1内存管理算法

heap_2：相比于 heap_1 内存管理算法， heap_2 内存管理算法使用最适应算法，并且支持释放内存

heap_2：heap_2 内存管理算法并不能将相邻的空闲内存块合并成一个大的空闲内存块；因此 heap_2 内存管理算法不可避免地会产生内存碎片

heap_2：最适应算法　　根据需要申请的内存大小，找出最小的且能满足内存要求的内存块

heap_2：适用场景：频繁的创建和删除任务，且所创建的任务堆栈都相同，这类场景下Heap_2没有碎片化的问题

heap_3：调用C库函数malloc()和 free()，Heap_3中先暂停FreeRTOS的调度器，再去调用这些函数，使用这种方法实现了线程安全

heap_4：heap_4 内存管理算法使用了首次适应算法，也支持内存的申请与释放，并且能够将空闲且相邻的内存进行合并，从而减少内存碎片的现象。

heap_5：heap_5 内存管理算法是在 heap_4 内存管理算法的基础上实现的，但是 heap_5 内存管理算法在 heap_4 内存管理算法的基础上实现了管理多个非连续内存区域的能力

在我们FreeRTOS例程中，使用的均为heap_4内存管理算法

3，FreeRTOS内存管理相关API函数介绍（熟悉）

申请内存：void * pvPortMalloc(size_t xWantedSize);

释放内存：void vPortFree(void * pv);

获取当前空闲内存的大小：size_t xPortGetFreeHeapSize(void);

注意：在一段内存没有被释放之前绝对不能再调用一次函数pvPortMalloc()为其再次分配内存，否则会导致内存泄露

heap_4 内存管理算法

定义一个大数组作为heap_4 管理的内存堆：uint8_t ucHeap[configTOTAL_HEAP_SIZE];

内存块结构体介绍：指向下一个空闲的内存块：struct A_BLOCK_LINK * pxNextFreeBlock;
最高位表示内存块是否已经被分配 其余为位表示内存块大小：size_t xBlockSize;

初始化内存堆：prvHeapInit()

- 获取内存堆大小
- 获取内存堆的起始地址
- 进行8字节对齐，并算出对齐后的起始地址和内存堆大小
- 将xStart 内存块指向下一个空闲地址 即字节对齐后的地址，并且xStart 内存块大小为0
- 将pxEnd内存块的初始地址赋值为：末尾地址 - 减掉内存块结构体大小
- 设置End
- 设置第一个空闲内存块的地址和大小，以及指向的下一个空闲内存块
- 记录历史堆栈剩余最小值和当前堆栈剩余值
- 将变量xBlockAllocatedBit的最高位置一，限制内存块大小

空闲内存链表的插入：prvInsertBlockIntoFreeList( )

- 作用：用于将空闲内存块插入空闲块链表 由地址从低到高排列，并将相邻的两个内存块合并
- 从头开始遍历空闲块链表， 找到第一个且下一个内存块的起始地址比待插入内存块高的内存块
- 判断找到的这个内存块是否与待插入内存块的低地址相邻　　相邻就将两个相邻的内存块合并
- 获取待插入内存块的起始地址
- 判断找到的这个内存块的下一个内存块始于待插入内存块的高地址相邻
  - 相邻：判断要合并的内存块是不是pxEnd
    - 是：将待插入内存块插入到 pxEnd 前面
    - 不是：将两个内存块合并
  - 不相邻：将待插入内存块，插入到找到的内存块的下一个内存块前面
- 判断找到的内存块是否不因为与待插入内存块的低地址相邻　　不相邻：将找到的内存块的下一个内存块指向待插入内存块

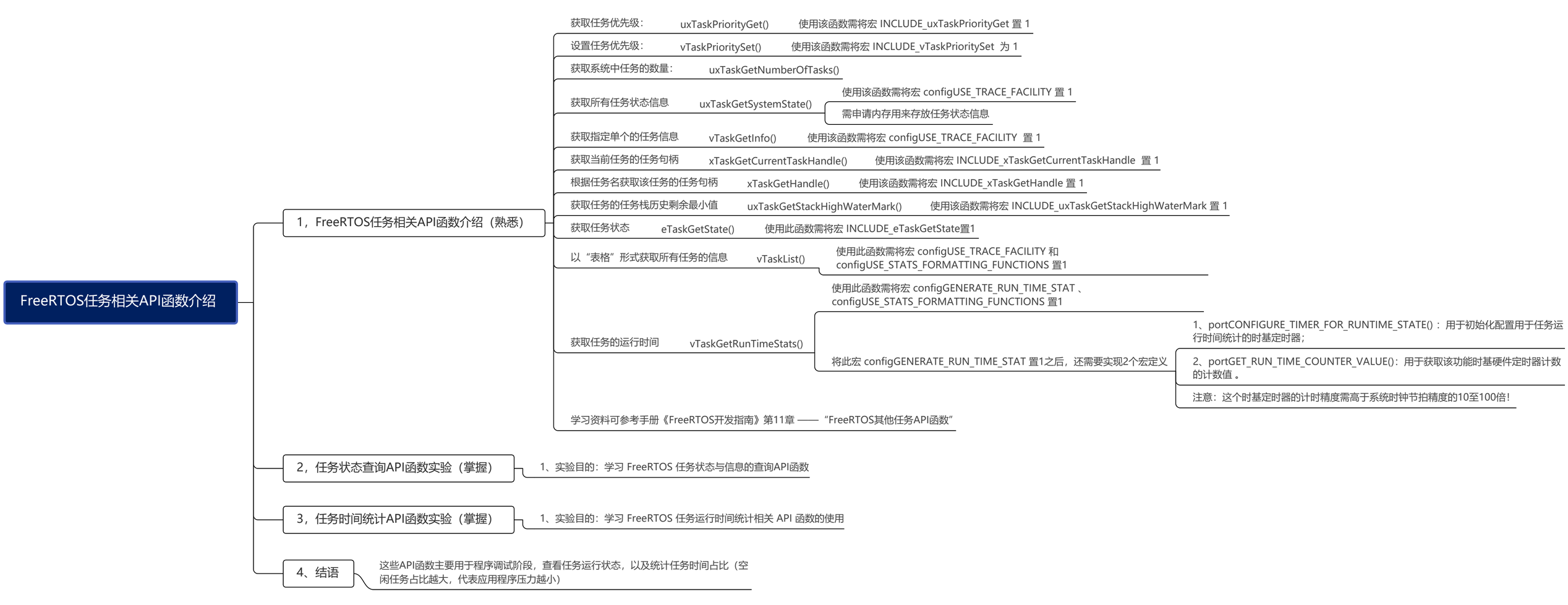
内存申请：pvPortMalloc( )

- 判断是否进行过初始化，如果未初始化，则进行一次初始化
- 需要申请的内存大小不能超过内存块的最大大小限制　　申请的内存大小需大于 0，并且未溢出
  - 将所需申请的内存大小加上内存块结构体的大小
  - 将所需申请的内存大小按 portBYTE_ALIGNMENT 字节对齐
- 所需的内存大小需大于 0，且小于内存堆中可分配内存大小
  - 从头遍历内存块链表，找到第一个内存大小适合的内存块
  - 判断是否找到了符合条件的内存块
    - 将返回值设置为符合添加内存块中可分配内存的起始地址
    - 将符合条件的内存块从空闲块链表中移除
    - 如果内存块中可分配内存比需要申请的内存大，那么这个内存块可以被分配两个内存块一个作为申请到的内存块，一个作为空闲块重新添加到空闲块链表中
    - 更新xFreeBytesRemaining内存堆中可分配的内存大小
    - 更新历史剩余最小堆栈

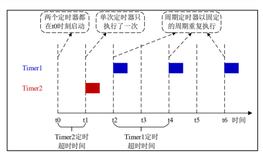
内存释放：vPortFree( )

- 将需要释放的内存块的地址进行偏移，偏移到起始地址（包含内存块结构体的地内存）
- 判断该内存块是否已被申请，是才能释放
  - 判断待释放的内存块是否不在空闲块链表中
  - 将待释放的内存块标记为未被分配
  - 更新空闲内存大小
  - 将释放的内存块插入空闲列表里，并更新释放成功次数

- FreeRTOS任务调度
  - 1，任务通知的简介（了解）
    - 任务通知：用来通知任务的，任务控制块中的结构体成员变量 ulNotifiedValue就是这个通知值。
    - 队列、信号量、事件标志组与任务通知的区别
      - 使用队列、信号量、事件标志组时都需另外创建一个结构体，通过中间的结构体进行间接通信！
      - 使用任务通知时，任务结构体TCB中就包含了内部对象，可以直接接收别人发过来的"通知"
    - 任务通知的特点
      - 不覆盖接受任务的通知值
      - 覆盖接受任务的通知值
      - 类似队列
      - 更新接受任务通知值的一个或多个bit
      - 类似事件标志组
      - 增加接受任务的通知值
      - 类似信号量
      - 总结：只要合理，灵活的利用任务通知的特点，可以在一些场合中替代队列、信号量、事件标志组！
    - 任务通知的优势和劣势
      - 优势
        - 1、效率更高
        - 2、使用内存更小
      - 劣势
        - 无法发送数据给ISR
        - 无法广播给多个任务
        - 无法缓存多个数据
        - 发送受阻不支持阻塞
  - 2，任务通知值和通知状态（熟悉）
    - 任务控制块TCB，它里边有两个结构体成员变量：一个是 uint32_t 类型，用来表示通知值
      一个是 uint8_t 类型，用来表示通知状态
    - 通知值更新方式
      - 计数值（数值累加，类似信号量）
      - 相应位置一（类似事件标志组）
      - 任意数值（支持覆写和不覆写）
    - 通知状态
      - 任务没有在等待通知
      - 等待通知：接收方已经准备好了，只需要你发送方给个通知
      - 等待接收：发送方已经发送出去，等待接收方接收
  - 3，任务通知相关API函数介绍（熟悉）
    - 任务通知API函数主要有两类：①发送通知 ，②接收通知。
    - 注意：发送通知API函数可以用于任务和中断服务函数，接收通知API函数只能用在任务中。
    - ①发送通知函数
      - 发送通知，带有通知值：xTaskNotify()
      - 发送通知，带有通知值并且保留接收任务的原通知值：xTaskNotifyAndQuery()
      - 发送通知，不带通知值用于模拟信号量：xTaskNotifyGive()
      - 均调用函数xTaskGenericNotify( )
    - ②接收通知函数
      - 获取任务通知，此函数适用于获取信号量：ulTaskNotifyTake( )
      - 获取任务通知，此函数适用于模拟队列，事件标志组：xTaskNotifyWait( )
  - 4，实战编程
    - 任务通知功能模拟二值信号量和计数型信号量：使用API函数：xTaskNotifyGive()、ulTaskNotifyTake( )
    - 任务通知功能模拟消息邮箱
    - 任务通知功能模拟事件标志组
    - 使用的API函数：xTaskNotify()、xTaskNotifyWait( )
  - 函数解析
    - 发送通知函数均调用函数xTaskGenericNotify( )
      - 1，判断是否需要保存原先的任务通知值
      - 2，记录目标任务先前的通知状态，然后赋值当前的任务状态（等待接收状态）
      - 3，更新通知值，有四种更新方式
        - 1，更新通知值某些位
        - 类似事件标志组
        - 2，通知值++
        - 类似信号量
        - 3，覆写的方式更新通知值
        - 4，不覆写的方式更新通知值
        - 类似队列
      - 4，如果目标任务为等待通知状态（代表先调用了任务接受函数，
        此时没有发送任务通知值，所以目标任务会阻塞）
        - 此时需将该任务解除阻塞状态，添加到就绪列表
        - 判断刚恢复的任务是否比当前正在执行的任务优先级更高，是的话就执行一次任务切换
    - （用于信号量获取）接收任务通知值：ulTaskNotifyTake( )
      - 1，判断任务通知的通知值是否为 0， 为 0 表示没有收到任务通知
        - 1，将当前任务的任务状态赋值为等待通知状态
        - 2，如果阻塞时间大于0代表需要阻塞，将任务添加到阻塞列表，并执行一次任务切换
      - 2，如果在此之前，任务被阻塞则解除阻塞后会执行到这
        - 收到通知值
        - 1，获取到通知值后，判断是否需要在成功读取通知后，
          将通知值清零
        - 1，xClearCountOnExit为pdture将通知值清0
        - 用于二值信号量
        - 2，xClearCountOnExit为pdfalse将通知值减1
        - 用于计数型信号量
      - 3，不论接收通知成功或者失败都将任务通知的状态标记为未等待通知状态
    - （用于将通知值指定位置一和获取通知值）接收任务通知值：xTaskNotifyWait( )
      - 1，判断任务通知的状态是否不为等待接收通知状态，如果是代表没有任务发送通知值
        - 1，等待任务通知前将任务通知通知值的指定比特位清零
        - 2，将任务通知状态更新为等待通知状态
        - 3，阻塞时间如果大于0，那么就将任务从就绪列表移除添加到阻塞列表（因为此时没有任务发送通知值）
        - 4，执行一次任务切换
      - 2，如果在此之前，任务被阻塞则解除阻塞后会执行到这
        - 1，判断pulNotificationValue是否不为NULL，是的话将通知值保存下来
        - 2，再次判断任务状态是否为等待接收状态
          因为如果接收到通知，任务通知值的状态会被置为等待接收通知状态
        - 不等于则未接收到通知
        - 接收到通知，在成功接收到通知后将任务通知通知值的指定比特位清零
      - 3，不论接收通知成功或者失败都将任务通知的状态标记为未等待通知状态


FreeRTOS任务相关API函数介绍
1，FreeRTOS任务相关API函数介绍（熟悉）
获取任务优先级：
uxTaskPriorityGet()
使用该函数需将宏 INCLUDE_uxTaskPriorityGet 置 1
设置任务优先级：
vTaskPrioritySet()
使用该函数需将宏 INCLUDE_vTaskPrioritySet 为 1
获取系统中任务的数量：
uxTaskGetNumberOfTasks()
获取所有任务状态信息
uxTaskGetSystemState()
使用该函数需将宏 configUSE_TRACE_FACILITY 置 1
需申请内存用来存放任务状态信息
获取指定单个的任务信息
vTaskGetInfo()
使用该函数需将宏 configUSE_TRACE_FACILITY 置 1
获取当前任务的任务句柄
xTaskGetCurrentTaskHandle()
使用该函数需将宏 INCLUDE_xTaskGetCurrentTaskHandle 置 1
根据任务名获取该任务的任务句柄
xTaskGetHandle()
使用该函数需将宏 INCLUDE_xTaskGetHandle 置 1
获取任务的任务栈历史剩余最小值
uxTaskGetStackHighWaterMark()
使用该函数需将宏 INCLUDE_uxTaskGetStackHighWaterMark 置 1
获取任务状态
eTaskGetState()
使用此函数需将宏 INCLUDE_eTaskGetState置1
以“表格”形式获取所有任务的信息
vTaskList()
使用此函数需将宏 configUSE_TRACE_FACILITY 和 configUSE_STATS_FORMATTING_FUNCTIONS 置1
获取任务的运行时间
vTaskGetRunTimeStats()
使用此函数需将宏 configGENERATE_RUN_TIME_STAT 、configUSE_STATS_FORMATTING_FUNCTIONS 置1
将此宏 configGENERATE_RUN_TIME_STAT 置1之后，还需要实现2个宏定义
1、portCONFIGURE_TIMER_FOR_RUNTIME_STATE()：用于初始化配置用于任务运行时间统计的时基定时器；
2、portGET_RUN_TIME_COUNTER_VALUE()：用于获取该功能时基硬件定时器计数的计数值 。
注意：这个时基定时器的计时精度需高于系统时钟节拍精度的10至100倍!
学习资料可参考手册《FreeRTOS开发指南》第11章 —— “FreeRTOS其他任务API函数”
2，任务状态查询API函数实验（掌握）
1、实验目的：学习 FreeRTOS 任务状态与信息的查询API函数
3，任务时间统计API函数实验（掌握）
1、实验目的：学习 FreeRTOS 任务运行时间统计相关 API 函数的使用
4、结语
这些API函数主要用于程序调试阶段，查看任务运行状态，以及统计任务时间占比（空闲任务占比越大，代表应用程序压力越小）

- FreeRTOS软件定时器
  - 软件定时器整体流程
    - 启动任务调度器：vTaskStartScheduler( )
    - 创建软件定时器服务任务：xTimerCreateTimerTask( )
      - 创建列表与消息队列：prvCheckForValidListAndQueue( )
        - vListInitialise(&xActiveTimerList1);
        - vListInitialise(&xActiveTimerList2);
        - xTimerQueue = xQueueCreate( )
      - xTaskCreate( )
      - for(; ;)
        - 获取最近一次定时器超时时间：prvGetNextExpireTime( );
        - 处理超时的定时器或者让队列阻塞：prvProcessTimerOrBlockTask( );
          - 第一次运行软件定时器任务在此阻塞进入休眠态
        - 处理队列接收到的命令：prvProcessReceivedCommands();
    - 发送命令队列
      - 软件定时器启动：xTimerStart( )
      - 软件定时器停止：xTimerStop( )
      - 软件定时器复位：xTimerReset( )
      - 软件定时器删除：xTimerDelete( )
      - ... ...
      - 当发送命令队列或某个定时器超时了，会唤醒软件定时器任务
  - 1，软件定时器的简介（了解）
    - 定时器：从指定的时刻开始，经过一个指定时间，然后触发一个超时事件，用户可自定义定时器的周期
    - 硬件定时器：芯片本身自带的定时器模块，硬件定时器的精度一般很高，每次在定时时间到达之后就会自动触发一个中断，用户在中断服务函数中处理信息。
    - 软件定时器：是指具有定时功能的软件，可设置定时周期，当指定时间到达后要调用回调函数（也称超时函数），用户在回调函数中处理信息
    - 软件定时器的优缺点：
      - 优点：
        - 硬件定时器数量有限，而软件定时器理论上只需有足够内存，就可以创建多个；
        - 使用简单、成本低
      - 缺点：
        - 软件定时器相对硬件定时器来说，精度没有那么高（因为它以系统时钟为基准，系统时钟中断优先级又是最低，容易被打断）。 对于需要高精度要求的场合，不建议使用软件定时器。
    - FreeRTOS软件定时器特点
      - 可裁剪
      - 单次和周期
      - 注意：软件定时器的超时回调函数是由软件定时器服务任务调用的，软件定时器的超时回调函数本身不是任务，因此不能在该回调函数中使用可能会导致任务阻塞的 API 函数。
      - 软件定时器服务任务：在调用函数 vTaskStartScheduler()开启任务调度器的时候，会创建一个用于管理软件定时器的任务，这个任务就叫做软件定时器服务任务。
      - 软件定时器服务任务作用
        - 1、负责软件定时器超时的逻辑判断
        - 2、调用超时软件定时器的超时回调函数
        - 3、处理软件定时器命令队列
  - 2，软件定时器的状态（熟悉）
    - 休眠态　　软件定时器可以通过其句柄被引用，但因为没有运行，所以其定时超时回调函数不会被执行
    - 运行态　　运行态的定时器，当指定时间到达之后，它的回调函数会被调用
    - 注意：新创建的软件定时器处于休眠状态 ，也就是未运行的！
    - 问题：如何让软件定时器从休眠态转变为运行态？　　解答：发送命令队列
  - 3，单次定时器和周期定时器（熟悉）
    - 单次定时器
      - 单次定时器的一旦定时超时，只会执行一次其软件定时器超时回调函数，不会自动重新开启定时，不过可以被手动重新开启。
    - 周期定时器
      - 周期定时器的一旦启动以后就会在执行完回调函数以后自动的重新启动 ，从而周期地执行其软件定时器回调函数。
    - 示意图：



      - Timer1：周期定时器，定时超时时间为 2 个单位时间，开启后，一直以2个时间单位间隔重复执行；
      - Timer2：单次定时器，定时超时时间为 1 个单位时间，开启后，则在第一个超时后就不在执行了。
  - 4，软件定时器结构体成员介绍（熟悉）
    - 详情见PPT
  - 5， FreeRTOS软件定时器相关API函数（熟悉）
    - 5.1，创建软件定时器API函数
      - 动态方式创建软件定时器：xTimerCreate()
      - 静态方式创建软件定时器：xTimerCreateStatic()
    - 5.2，开启软件定时器API函数
      - 开启软件定时器定时：xTimerStart()
      - 在中断中开启软件定时器定时：xTimerStartFromISR()
    - 5.3，停止软件定时器API函数
      - 停止软件定时器定时：xTimerStop()
      - 在中断中停止软件定时器定时：xTimerStopFromISR()
    - 5.4，复位软件定时器API函数
      - 复位软件定时器定时：xTimerReset()
      - 在中断中复位软件定时器定时：xTimerResetFromISR()
    - 5.5，更改软件定时器超时时间API函数
      - 更改软件定时器的定时超时时间：xTimerChangePeriod()
      - 在中断中更改定时超时时间：xTimerChangePeriodFromISR()


FreeRTOS时间管理
1，延时函数介绍（了解）
1.相对延时函数：vTaskDelay( )
2.绝对延时函数：xTaskDelayUntil( )
3.相对延时：指每次延时都是从执行函数vTaskDelay()开始，直到延时指定的时间结束
4.绝对延时：指将整个任务的运行周期看成一个整体，适用于需要按照一定频率运行的任务
2，延时函数解析（熟悉）
1.判断延时时间是否大于0，大于0才有效
2.挂起调度器
3.将当前正在运行的任务从就绪列表移除，添加到阻塞列表
prvAddCurrentTaskToDelayedList( )
1.将该任务从就绪列表中移除
2.如果使能挂起操作，并且延时时间为0XFFFF FFFF，并且xCanBlockIndefinitely等于pdTRUE，就代表此时是一直等，相当于挂起，所以添加到挂起列表
3.如果延时时间小于0XFFFF FFFF
1.记录阻塞超时时间，并记录在列表项值里（通过该值确定插入阻塞列表的位置）
2.如果阻塞超时时间溢出，将该任务状态列表项添加到溢出阻塞列表
3.如果没溢出，则将该任务状态列表项添加到阻塞列表，并判断阻塞超时时间是否小于下一个阻塞超时时间，是的话就更新当前这个时间为下一个阻塞超时时间
4.恢复任务调度器
5.进行一次任务切换
延时函数的流程
正在运行的任务
调用延时函数
此时将该任务移除就绪列表，并添加到阻塞列表中
滴答中断里边进行计时
判断阻塞时间是否到达，如果到达将从阻塞列表移除，添加到就绪列表

FreeRTOS事件标志组

1，事件标志组简介（了解）

简介：事件标志组是一组事件标志的集合， 可以简单的理解事件标志组，就是一个整数

特点：

- 它的每一个位表示一个事件（高8位不算）
- 每一位事件的含义，由用户自己决定，如：bit0表示按键是否按下，bit1表示是否接受到消息 … …（这些位值为1：表示事件发生了；值为0：表示事件未发生）
- 任意任务或中断都可以读写这些位
- 可以等待某一位成立，或者等待多位同时成立

一个事件组就包含了一个 EventBites_t 数据类型的变量，EventBits_t 实际上是一个16 位或 32 位无符号的数据类型（STM32中这个变量类型为32位的）

虽然使用了 32 位无符号的数据类型变量来存储事件标志， 但其中的高8位用作存储事件标志组的控制信息，低24位用作存储事件标志 ，所以说一个事件组最多可以存储 24 个事件标志！

事件标志组与队列、信号量的区别？

- 队列、信号量：事件发生时，只会唤醒一个任务，并且是消耗型的资源，队列的数据被读走就没了；信号量被获取后就减少了
- 事件组：事件发生时，会唤醒所有符号条件的任务，简单地说它有"广播"的作用，并且被唤醒的任务有两个选择，可以让事件保留不动，也可以清除事件

2，事件标志组相关API函数介绍（熟悉）

动态方式创建事件标志组API函数：EventGroupHandle_t　xEventGroupCreate (void) ;

等待事件标志位API函数：EventBits_t xEventGroupWaitBits(EventGroupHandle_t xEventGroup,
const EventBits_t uxBitsToWaitFor,
const BaseType_t xClearOnExit,
const BaseType_t xWaitForAllBits,
TickType_t xTicksToWait)

设置事件标志位API函数：EventBits_t xEventGroupSetBits(EventGroupHandle_t xEventGroup,
const EventBits_t uxBitsToSet)

同步函数：EventBits_t xEventGroupSync(EventGroupHandle_t xEventGroup,
const EventBits_t uxBitsToSet,
const EventBits_t uxBitsToWaitFor,
TickType_t xTicksToWait)

更多事件标志组相关的API函数介绍请查阅《FreeRTOS开发指南》-- 第十六章“FreeRTOS事件标志组”

FreeRTOS消息队列

队列相关API函数解析

创建队列：xQueueCreate( )

- 实际执行的是xQueueGenericCreate( )
- xQueueGenericCreate((uxQueueLength), (uxItemSize), (queueQUEUE_TYPE_BASE))
- 1、计算队列需要多大内存 xQueueSizeInBytes = (size_t) (uxQueueLength * uxItemSize)
- 2、为队列申请内存，申请大小：sizeof(Queue_t) + xQueueSizeInBytes ，前面部分存放结构体成员，后面就是队列项大小
- 3、判断内存是否申请成功，成功即计算出队列项存储区的首地址
- 4、调用prvInitialiseNewQueue（）初始化新队列pxNewQueue
  - 1、初始化队列结构体成员变量
  - 2、调用xQueueGenericReset（）复位队列
    - 1、初始化其他队列结构体成员变量
    - 2、判断要复位的队列是否为新创建的队列
      - 不是新创建队列，那就复位它，将列表xTasksWaitingToSend移除
      - 是新创建的队列，那就初始化这两个列表xTasksWaitingToSend和xTasksWaitingToReceive

往队列写入数据（入队）：xQueueSend( )

- 实际执行的是：xQueueGenericSend((xQueue), (pvItemToQueue), (xTicksToWait), queueSEND_TO_BACK)
- 进入临界区（关中断）
- 判断队列是否已满?
- 队列有空闲位置
  - 1,、只有在队列有空闲位置或为覆写的情况才能写入消息
  - 2、当有空闲位置或覆写时：将待写入消息按指定写入方式复制到队列中
  - 3、判断是否有因为读不到消息而阻塞的任务，有的话，将解除阻塞态，通过这个函数实现：xTaskRemoveFromEventList( )
    - 判断调度器是否被挂起
      - 没有挂起：将移除相应的事件列表项和状态列表项，并且将任务添加到就绪列表中
      - 挂起：将移除事件列表项，将事件列表项添加到等待就绪列表：xPendingReadyList，当调用恢复调度器时xTaskResumeAll( )，xPendingReadyList中多任务就会被处理
  - 退出临界区（开中断）
- 队列已满
  - 此时不能写入消息，因此要将任务阻塞
  - 如果阻塞时间为0 ，代表不阻塞，直接返回队列满错误
  - 如果阻塞时间不为0，任务需要阻塞，记录下此时系统节拍计数器的值和溢出次数，用于下面对阻塞时间进行补偿
  - 判断阻塞时间补偿后，是否还需要阻塞
    - 需要：
      - 将任务的事件列表项添加到等待发送列表中
      - 将任务状态列表项添加到阻塞列表中进行阻塞，队列解锁，恢复调度器
    - 不需要：队列解锁，恢复调度器，返回队列满错误

从队列读取数据（出队）：xQueueReceive( )

- 进入临界区（关中断）
- 判断队列是否为空
- 有数据
  - 使用函数prvCopyDataFromQueue( )拷贝数据
  - 队列项目个数减一
  - 因为前面已经减了一个队列项，所以队列已经有空位了，如果xTasksWaitingToSend等待发送列表中，有任务，则解除阻塞态，通过这个函数xTaskRemoveFromEventList( )
    - 判断调度器是否被挂起
      - 没有挂起：将移除相应的事件列表项和状态列表项，并且将任务添加到就绪列表中
      - 挂起：将移除事件列表项，将事件列表项添加到等待就绪列表：xPendingReadyList，当调用恢复调度器时xTaskResumeAll( )，xPendingReadyList中多任务就会被处理
  - 退出临界区（开中断）
- 为空
  - 此时读取不到消息，因此要将任务阻塞
  - 如果阻塞时间为0 ，代表不阻塞，直接返回队列空错误
  - 如果阻塞时间不为0，任务需要阻塞，记录下此时系统节拍计数器的值和溢出次数，用于下面对阻塞时间进行补偿
  - 判断阻塞时间补偿后，是否还需要阻塞
    - 需要：
      - 将任务的事件列表项添加到等待接收列表中
      - 将任务状态列表项添加到阻塞列表中进行阻塞，队列解锁，恢复调度器
    - 不需要：队列解锁，恢复调度器，返回队列空错误

FreeRTOS消息队列

1，队列简介（了解）

- 1.简介：队列是任务到任务、任务到中断、中断到任务数据交流的一种机制（消息传递）
- 2.在OS中如果使用全局变量存在弊端：数据无保护，导致数据不安全，当多个任务同时对该变量操作时，数据易受损
- 3.队列：读写队列做好了保护，防止多任务同时访问冲突；我们只需要直接调用API函数即可，简单易用!
- 4.FreeRTOS基于队列， 实现了多种功能，其中包括队列集、互斥信号量、计数型信号量、二值信号量、 递归互斥信号量，因此很有必要深入了解 FreeRTOS 的队列 。
- 5.特点：
  - 队列通常采用“先进先出”(FIFO)的数据存储缓冲机制，即先入队的数据会先从队 列中被读取，FreeRTOS中也可以配置为“后进先出”LIFO方式；
  - FreeRTOS中队列采用实际值传递，即将数据拷贝到队列中进行传递， FreeRTOS采用拷贝数据传递，也可以传递指针，所以在传递较大的数据的时候采用指针传递
  - 队列不属于某个任务，任何任务和中断都可以向队列发送消息 以及 读取消息
  - 当任务向一个队列发送消息时，可以指定一个阻塞时间，假设此时当队列已满无法入队
    - 若阻塞时间为0：直接返回不会等待；
    - 若阻塞时间为0~port_MAX_DELAY：等待设定的阻塞时间，若在该时间内还无法入队，超时后直接返回不再等待；
    - 若阻塞时间为port_MAX_DELAY：死等，一直等到可以入队为止。出队阻塞与入队阻塞类似；

2，队列结构体介绍（熟悉）

3，队列相关API函数介绍（熟悉）

- 1.使用队列的主要流程：创建队列 -->写队列 --> 读队列。
- 2.创建队列API函数
  - 动态方式创建队列：xQueueCreate()
  - 静态方式创建队列：xQueueCreateStatic()
  - 动态和静态创建队列之间的区别：队列所需的内存空间由 FreeRTOS 从 FreeRTOS 管理的堆中分配，而静态创建需要用户自行分配内存。
- 3.写队列API函数
  - 往队列的尾部写入消息：xQueueSend()
  - 往队列的尾部写入消息：xQueueSendToBack()
  - 往队列的头部写入消息：xQueueSendToFront()
  - 覆写队列消息（只用于队列长度为 1 的情况）：xQueueOverwrite()
  - 均调用xQueueGenericSend( )，只是写入位置不同
  - 在中断中往队列的尾部写入消息：xQueueSendFromISR()
  - 在中断中往队列的尾部写入消息：xQueueSendToBackFromISR()
  - 在中断中往队列的头部写入消息：xQueueSendToFrontFromISR()
  - 在中断中覆写队列消息：xQueueOverwriteFromISR()
  - 均调用xQueueGenericSendFromISR( )，只是写入位置不同
- 4.读队列API函数
  - 从队列头部读取消息，并删除消息：xQueueReceive()
  - 从队列头部读取消息：xQueuePeek()
  - 在中断中从队列头部读取消息，并删除消息：xQueueReceiveFromISR()
  - 在中断中从队列头部读取消息：xQueuePeekFromISR()

4，队列操作实验（掌握）

- 1、实验目的：学习 FreeRTOS 的队列相关API函数的使用 ，实现队列的入队和出队操作。

FreeRTOS信号量

1，信号量的简介（了解）

简介：信号量是一种解决同步问题的机制，可以实现对共享资源的有序访问

特点：

- 当计数值大于0，代表有信号量资源
- 当释放信号量，信号量计数值（资源数）加一
- 当获取信号量，信号量计数值（资源数）减一
- 信号量：用于传递状态
- 当信号量如果最大值被限定为1，那么它就是二值信号量；如果最大值不是1，它就是计数型信号量。

2，二值信号量（熟悉）

简介：二值信号量的本质是一个队列长度为 1 的队列 ，该队列就只有空和满两种情况，这就是二值。

相关API函数介绍

- 创建二值信号量函数：SemaphoreHandle_t　xSemaphoreCreateBinary(void)
- 释放二值信号量函数：BaseType_t　xSemaphoreGive(xSemaphore)
- 获取二值信号量函数：BaseType_t　xSemaphoreTake(xSemaphore, xBlockTime)

3，计数型信号量（熟悉）

简介：计数型信号量相当于队列长度大于1 的队列，因此计数型信号量能够容纳多个资源，这在计数型信号量被创建的时候确定的

计数型信号量适用场合：

- 事件计数：当每次事件发生后，在事件处理函数中释放计数型信号量（计数值+1），其他任务会获取计数型信号量（计数值-1） ，这种场合一般在创建时将初始计数值设置为 0
- 资源管理：信号量表示有效的资源数目。任务必须先获取信号量（信号量计数值-1 ）才能获取资源控制权。当计数值减为零时表示没有的资源。当任务使用完资源后，必须释放信号量（信号量计数值+1）。信号量创建时计数值应等于最大资源数目

相关API函数介绍

- 创建计数型信号量：xSemaphoreCreateCounting(uxMaxCount , uxInitialCount)
- 获取信号量当前计数值大小：uxSemaphoreGetCount(xSemaphore)
- 计数型信号量的释放和获取与二值信号量相同！

4，优先级翻转简介（熟悉）

简介：优先级翻转：高优先级的任务反而慢执行，低优先级的任务反而优先执行

- 优先级翻转在抢占式内核中是非常常见的，但是在实时操作系统中是不允许出现优先级翻转的，因为优先级翻转会破坏任务的预期顺序，可能会导致未知的严重后果。
- 总结：高优先级任务被低优先级任务阻塞，导致高优先级任务迟迟得不到调度。但其他中等优先级的任务却能抢到CPU资源。从现象上看，就像是中优先级的任务比高优先级任务具有更高的优先权（即优先级翻转）

5，互斥信号量（熟悉）

简介：互斥信号量其实就是一个拥有优先级继承的二值信号量，在同步的应用中二值信号量最适合。互斥信号量适合用于那些需要互斥访问的应用中！

优先级继承：当一个互斥信号量正在被一个低优先级的任务持有时， 如果此时有个高优先级的任务也尝试获取这个互斥信号量，那么这个高优先级的任务就会被阻塞。不过这个高优先级的任务会将低优先级任务的优先级提升到与自己相同的优先级。

注意：互斥信号量不能用于中断服务函数中，原因如下：

- (1) 互斥信号量有任务优先级继承的机制， 但是中断不是任务，没有任务优先级， 所以互斥信号量只能用与任务中，不能用于中断服务函数。
- (2) 中断服务函数中不能因为要等待互斥信号量而设置阻塞时间进入阻塞态。

注意：创建互斥信号量时，会主动释放一次信号量