目录

[第1章 FreeRTOS介绍 2](#_Toc202888289)

[1.1 操作系统概述 2](#_Toc202888290)

[1.1.1 何为操作系统 2](#_Toc202888291)

[1.1.2 嵌入式操作系统 2](#_Toc202888292)

[1.2 FreeRTOS概述 3](#_Toc202888293)

[1.2.1 简介 3](#_Toc202888294)

[1.2.2 FreeRTOS特点 3](#_Toc202888295)

[1.2.1 FreeRTOS操作系统的调度原则 4](#_Toc202888296)

[1.2.2 FreeRTOS操作系统的程序结构 4](#_Toc202888297)

[1.2.3 FreeRTOS操作系统的任务结构 4](#_Toc202888298)

[第2章 FreeRTOS移植 5](#_Toc202888299)

[2.1 FreeRTOS在STM32F4上移植 5](#_Toc202888300)

[2.1.1 准备工作 5](#_Toc202888301)

[2.1.2 获取源码 5](#_Toc202888302)

[2.1.3 文件夹介绍 5](#_Toc202888303)

[2.1.4 添加文件 6](#_Toc202888304)

[2.1.5 配置和裁剪 6](#_Toc202888305)

[2.1.6 宏说明 7](#_Toc202888306)

[第3章 任务基础 8](#_Toc202888307)

[3.1 认识任务 8](#_Toc202888308)

[3.2 任务的状态 8](#_Toc202888309)

[3.3 任务的优先级 9](#_Toc202888310)

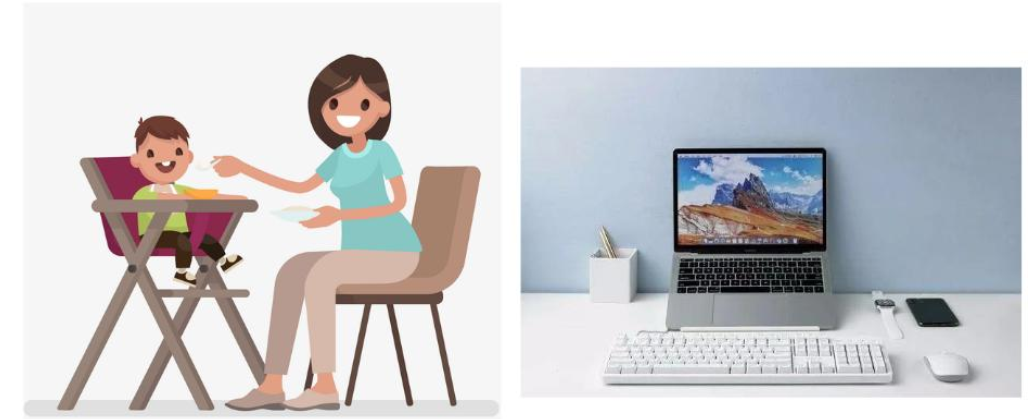
[3.4 任务的创建和删除 9](#_Toc202888311)

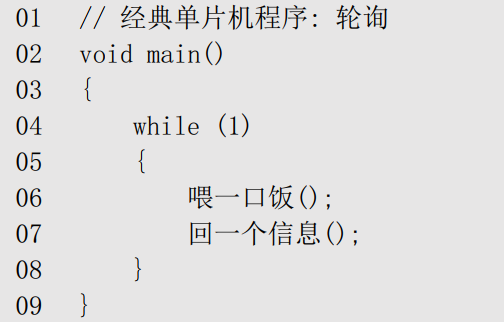
# FreeRTOS介绍

## 裸机与RTOS的举例对比

裸机程序的设计模式可以分为： 轮询、 前后台、 定时器驱动、 基于状态机。 前面三种方法都无法解决一个问题： 假设有 A、 B 两个都很耗时的函数， 无法降低它们相互之间的影响。第 4 种方法可以解决这个问题， 但是实践起来有难度。

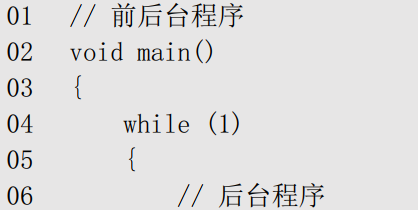
假设一位职场妈妈需要同时解决 2 个问题： 给小孩喂饭、 回复工作信息， 场景如图所示，后面将会演示各类模式下如何写程序：

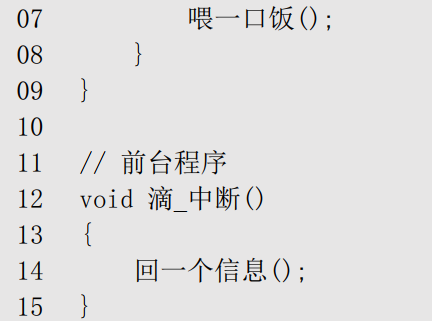




在 main 函数中是一个 while 循环， 里面依次调用 2 个函数， 这两个函数相互之间有影响： 如果“喂一口饭” 太花时间， 就会导致迟迟无法“回一个信息”； 如果“回一个信息”太花时间， 就会导致迟迟无法“喂下一口饭”。

使用轮询模式编写程序看起来很简单， 但是要求 while 循环里调用到的函数要执行得非常快，在复杂场景里反而增加了编程难度。





所谓“前后台” 就是使用中断程序。 假设收到同事发来的信息时， 电脑会发出“滴” 的一声， 这时候妈妈才需要去回复信息。 示例程序如下：

main 函数里 while 循环里的代码是后台程序， 平时都是 while 循环在运行；

当同事发来信息， 电脑发出“滴” 的一声， 触发了中断。 妈妈暂停喂饭， 去执行“滴\_

中断” 给同事回复信息；

在这个场景里， 给同事回复信息非常及时： 即使正在喂饭也会暂停下来去回复信息。“喂一口饭” 无法影响到“回一个信息”。 但是， 如果“回一个信息” 太花时间， 就会导致 “喂一口饭” 迟迟无法执行。

对于裸机程序， 无论使用哪种模式进行精心的设计， 在最差的情况下都无法解决这个问题： 假设有 A、 B 两个都很耗时的函数， 无法降低它们相互之间的影响。 使用状态机模式时，如果函数拆分得不好， 也会导致这个问题。 本质原因是： 函数是轮流执行的。 假设“喂一口饭” 需要 t1~t5 这 5 段时间， “回一个信息需要” ta~te 这 5 段时间， 轮流执行时： 先执行完 t1~t5， 再执行 ta~te，如下图所示：



对于职场妈妈， 她怎么解决这个问题呢？ 她是一个眼明手快的人， 可以一心多用， 她这

样做：

 左手拿勺子， 给小孩喂饭

 右手敲键盘， 回复同事

 两不耽误， 小孩“以为” 妈妈在专心喂饭， 同事“以为” 她在专心聊天

 但是脑子只有一个啊， 虽然说“一心多用”， 但是谁能同时思考两件事？

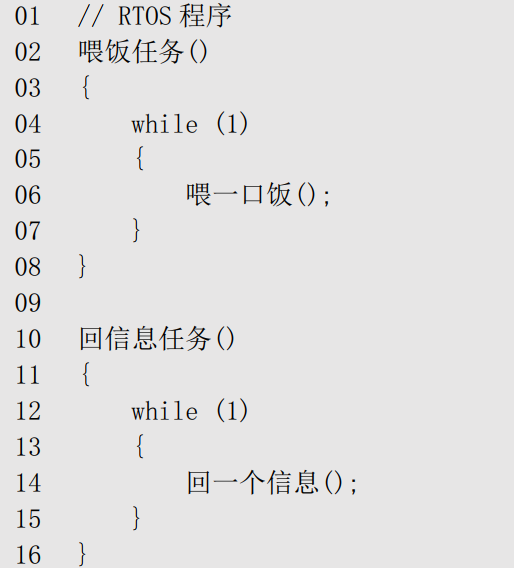
 只是她反应快， 上一秒钟在考虑夹哪个菜给小孩， 下一秒钟考虑给同事回复什么信息

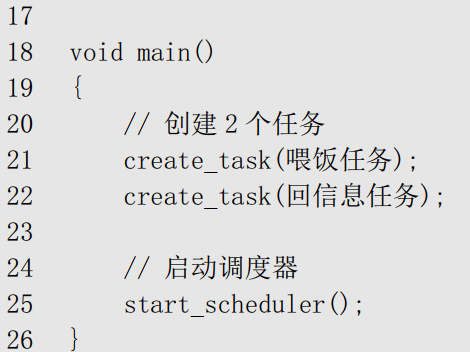
 本质是： 交叉执行， t1~t5 和 ta~te 交叉执行， 如下图所示：

时间片：1ms 快速的切换任务执行，就可以达到伪并行的效果。



这时厨房着火了，该任务优先级相比喂孩子和回消息优先级高，这时候会涉及到优先级的概念。喂孩子和回消息优先级相同，当有高优先级任务出现时，妈妈会去灭火，直到火熄灭为止。（只要高优先级任务一直在执行，CPU会一直执行高优先级任务，直到高优先级任务被删除或者挂起（暂停）或者阻塞，CPU才会执行其他低优先级任务）。





第 21、 22 行， 创建 2 个任务；

第 25 行， 启动调度器；

之后， 这 2 个任务就会交叉执行了；

基于多任务系统编写程序时， 反而更简单了：

上面第 2~8 行是“喂饭任务” 的代码；

第 10~16 行是“回信息任务” 的代码， 编写它们时甚至都不需要考虑它和其他函数

的相互影响。 就好像有 2 个单板： 一个只运行“喂饭任务” 这个函数、 另一个只运

行“回信息任务” 这个函数。

多任务系统会依次给这些任务分配时间： 你执行一会， 我执行一会， 如此循环。 只要切换的间隔足够短， 用户会“感觉这些任务在同时运行”。 如下图所示：



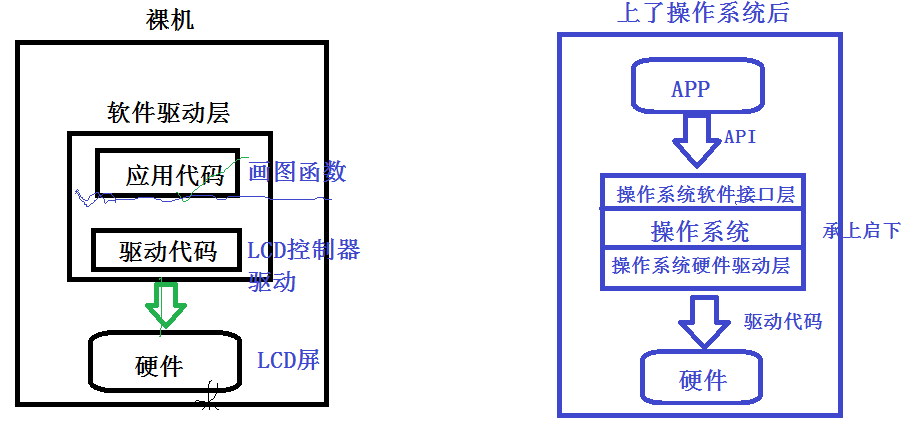
**RTOS处理方式：**时间片：1ms 快速的切换任务执行，就可以达到伪并行的效果。在高优先级出现时，只要高优先级任务一直在执行，CPU会一直执行高优先级任务，直到高优先级任务被删除或者挂起（暂停）或者阻塞，CPU才会执行其他低优先级任务）。

## 操作系统概述

### 何为操作系统

什么是操作系统？

操作系统是管理和控制计算机硬件与软件资源的计算机程序，是直接运行在“裸机”上的最基本的系统软件，任何其他软件都必须在操作系统的支持下才能运行。介于APP和硬件之间。



1. 为什么要用操作系统？
2. 相比裸机，CPU的灵活性大大提高了。
3. 相比裸机，可以用少量代码实现更加复杂的功能
4. 屏蔽硬件：使得上层应用APP移植性更好。

### 嵌入式操作系统

1. **实时操作系统（RTOS）**

**注重实时性，小型的操作系统（**嵌入式**）**

系统的任务调度（任务切换）靠优先级完成，任务的执行是没有时间限制，想结束任务，直到高优先级任务被删除或者挂起（暂停）或者阻塞，CPU才会执行其他低优先级任务）。

RTT、UCOSII、FreeRTOS

**2.分时操作系统**

任务的执行和任务调度操作靠时间片完成。

比如 时间片1ms 给任务A分配10个时间片（任务A执行10ms）

Windows95/98/2000、LINUX2.6内核之前

**3.半分时半实时操作系统**

Windows7/8/10、LINUX2.6内核之后

**用哪一种操作系统比较好？**

由平台决定。

在嵌入式中，使用的芯片的主频都不是特别的快，为了能实现任务间快速的切换，就需要选择实时操作系统。

在消费电子领域：采用的芯片都是比较高端的，主频非常快，1S可以轮询任务很多次，人眼分辨不出来，感觉不到卡顿。

举例：在下载大型软件时，用输入法输入文字会出现卡顿的情况，这是因为系统分配大量时间给下载进程。

嵌入式操作系统和通用计算机操作系统有什区别？

嵌入式：专向领域（智能手表，汽车，家电） 计算机：电脑，服务器，笔记本

嵌入式：微型，精简，所需要内存空间小。 计算机：需要更大的内存空间（磁盘）

嵌入式：配置简单，易裁剪。 计算机：系统庞大，配置复杂，不易裁剪

## FreeRTOS概述

### 简介

我们看一下FreeRTOS的名字，可以分为两部分:Free和 RTOS，Free就是免费的、自由的、不受约束的意思，RTOS全称是Real Time Operating System，中文名就是实时操作系统。可以看出FreeROTS就是一个免费的RTOS类系统。这里要注意，RTOS不是指某一个确定的系统，而是指一类系统。比如UCOS，FreeRTOS，RTX，RT-Thread等这些都是RTOS类操作系统。

### FreeRTOS特点

FreeRTOS 是可裁剪的小型嵌入式实时操作系统，除开源、免费以外，还具有以下特点。

(1) FreeRTOS 的内核支持抢占式、合作式和时间片三种调度方式。  
(2)支持的芯片种类多，已经在超过30种架构的芯片上进行了移植。  
(3)系统简单、小巧、易用，通常情况下其内核仅占用4~9KB的ROM空间。

(4) 代码主要用C语言编写，可移植性高。  
(5)支持 ARM Cortex-M系列中的MPU，如STM32F429等有MPU的芯片。

(6)任务数量不限。  
(7)任务优先级不限。(在移植到芯片中，会受到CPU能力的限制)  
(8)任务与任务、任务与中断之间可以使用任务通知、队列、二值信号量、计数信号量、互斥信号量和递归互斥信号量进行通信和同步。  
(9)有高效的软件定时器。  
(10)有强大的跟踪执行功能。

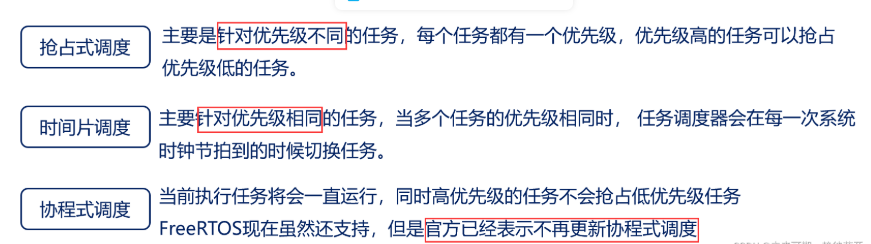
(11)有堆栈溢出检测功能。  
(12)适用于低功耗应用。FreeRTOS 提供了一个低功耗tickless模式。  
(13)在创建任务通知、队列、信号量、软件定时器等系统组件时，可以选择动态或静态 RAM。  
(14) SafeRTOS 作为FreeRTOS 的衍生品，具有比FreeRTOS更高的代码完整性。

为什么选择FreeRTOS?

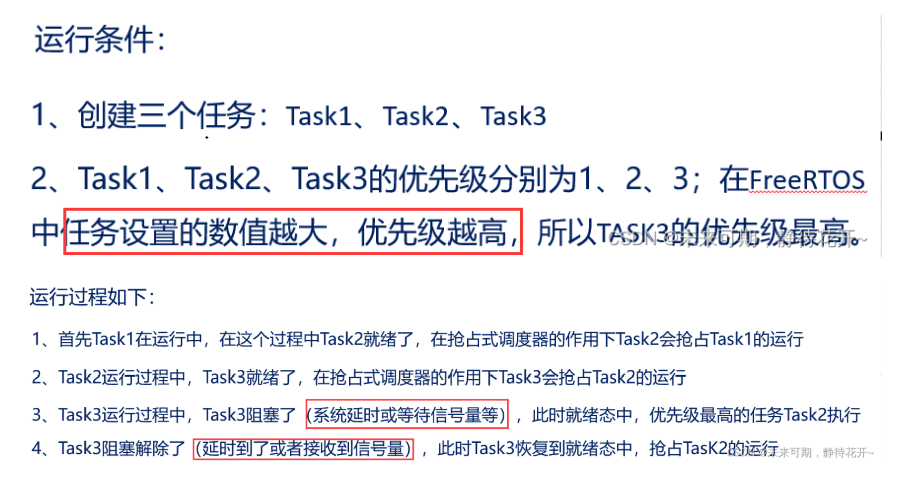
1. FreeRTOS是免费的。
2. FreeRTOS得到众多半导体厂商支持。
3. 越来越多的软件厂商使用FreeRTOS。
4. FreeRTOS很容易移植到不同架构的处理器中。
5. FreeRTOS文件数量少，占用内存少，使用简单，高效。
6. FreeRTOS的社会占有量高。

### FreeRTOS操作系统的调度原则

FreeRTOS 一共支持三种任务调度方式：



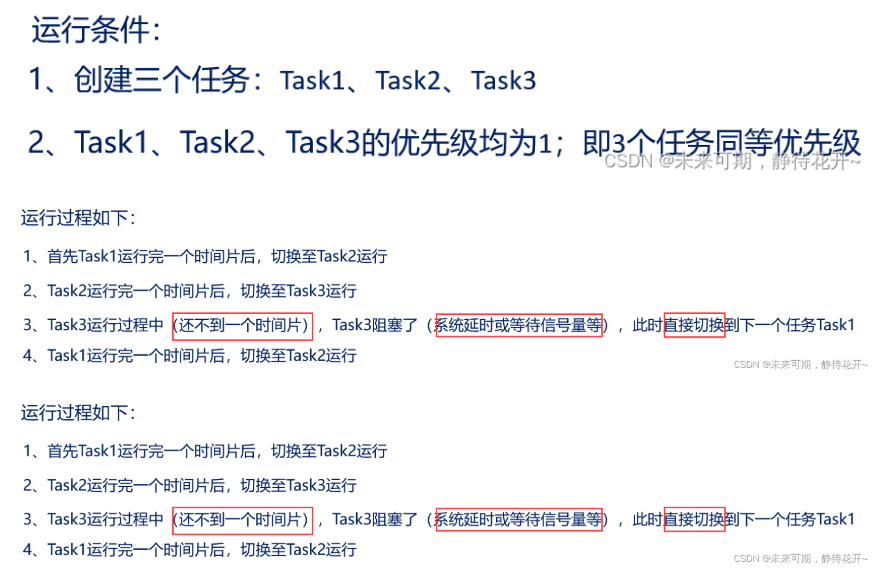
**抢占式调度场景：**



**总结：**

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，除非高优先级的任务进入阻塞态或者被挂起，这时便会让出CPU的使用权给就绪态中最高的优先级任务使用，运行它；**
3. **被抢占的任务将会进入就绪态；**

**时间片调度场景：**



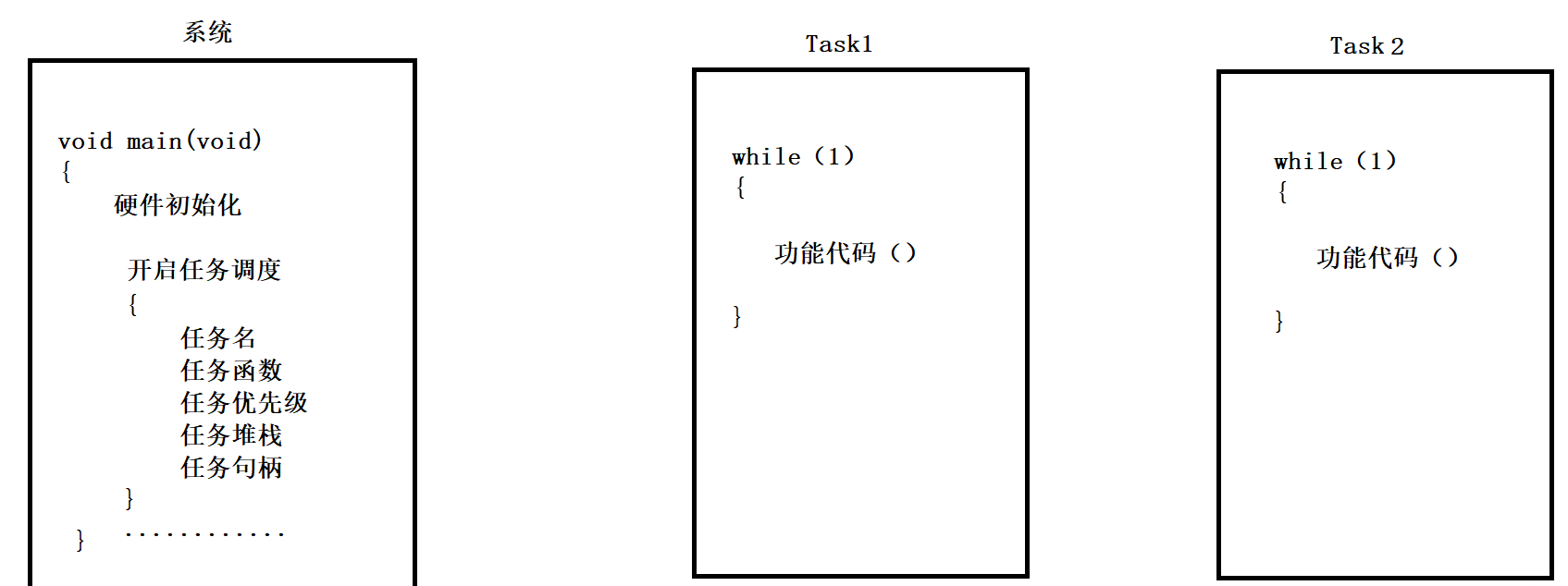
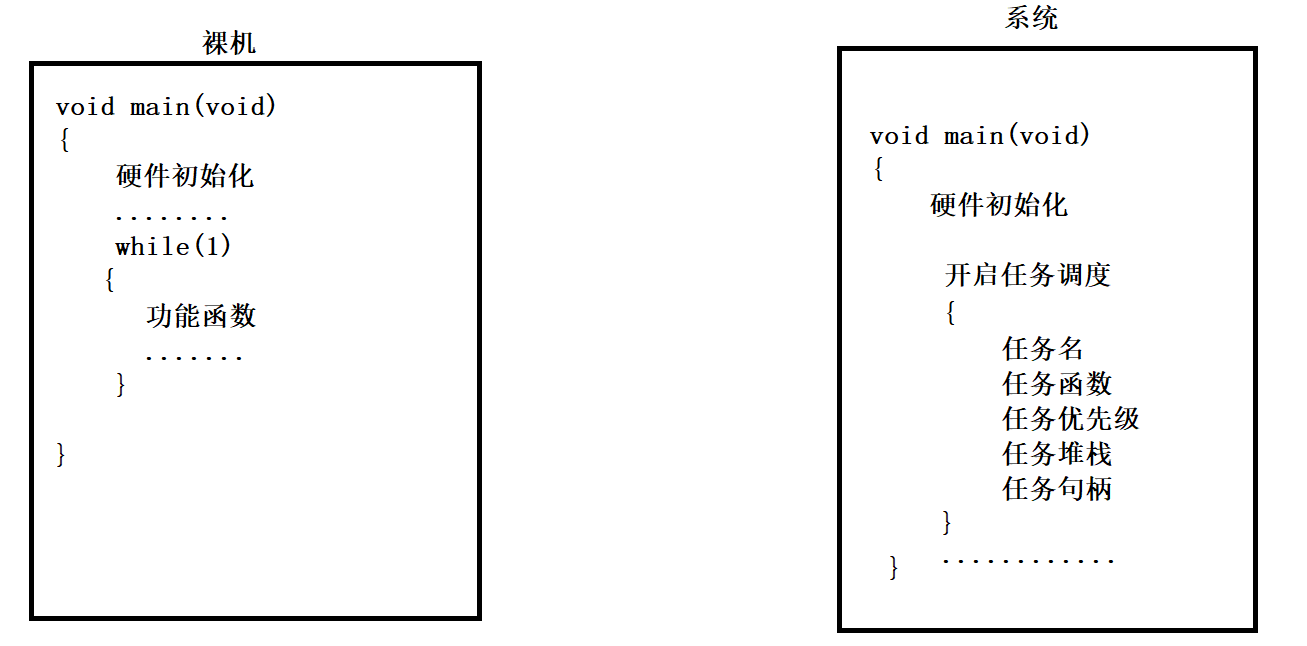
**总结：**

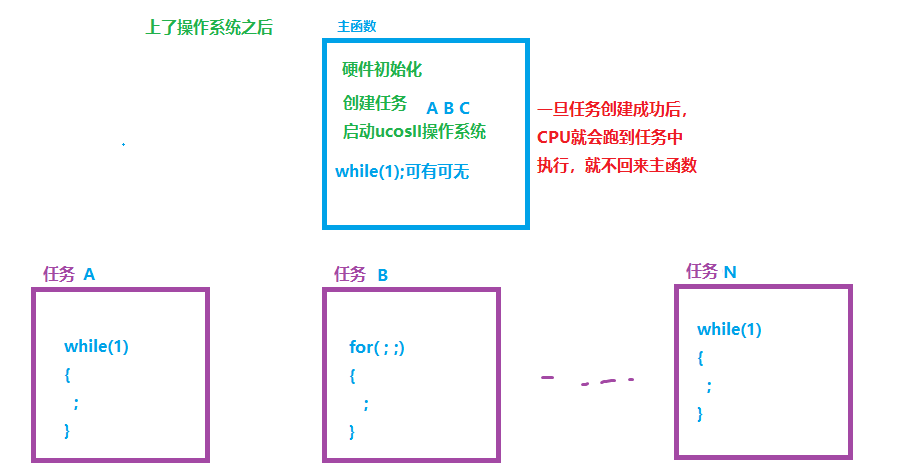
1. **同等优先级任务，轮流执行；时间片流转；**
2. **一个时间片大小，取决于滴答定时器中断周期；**
3. **注意没有用完的时间片不会再使用，下次任务Task3得到执行**，**还是按照一个时间片的时钟节拍运**

### FreeRTOS操作系统的程序结构

裸机：有且只有一个主函数main()(作为程序的入口)，在主函数有一个死循环；把想要实现的功能函数放在主函数里面。

上了操作系统之后：有且只有一个主函数main()(作为程序入口)，在主函数里面死循环可有可无，在程序中，需要创建多个任务，每一个任务其实就是一个功能代码，每个任务需要一个死循环

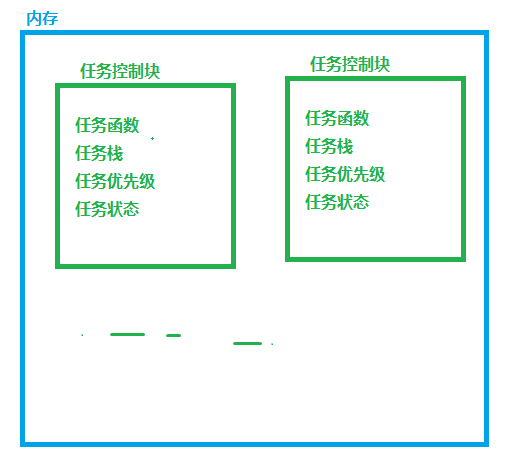




### FreeRTOS操作系统的任务结构

任务结构：任务控制块、任务函数、任务栈、任务优先级、任务状态

任务控制块：当任务创建成功，系统会自动分配一段内存空间，这一段内存空间就是任务控制块，在这一段空间中，就是存放该任务的存放任务函数、任务栈、任务优先级、任务状态



任务函数(由用户决定)：其实就是普通函数，就是将你想要实现的功能代码写到该任务函数中，系统是根据任务函数名来找到该任务函数

任务优先级(由用户决定)：每一个任务都有一个唯一的编号，是作为系统调度和系统切换唯一依据，编号越小优先级越高

任务栈(由自己设置)：当发生任务切换，用来保存任务环境(寄存器配置、变量等等)和恢复任务环境

任务状态：停止/休眠、挂起/等待、就绪、运行、中断 任务肯定是处于这五个状态中一种

# FreeRTOS移植

## FreeRTOS在STM32F4上移植

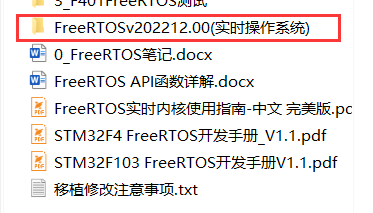
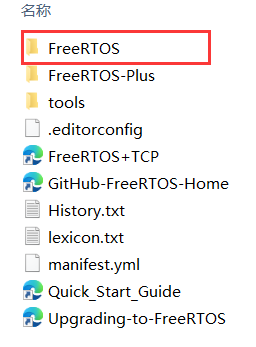
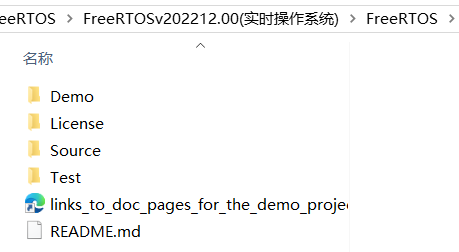
### 准备工作

要移植FreeRTOS，肯定需要一个基础工程，基础工程越简单越好。

### 获取源码

下发的资料中有，也可以从FreeRTOS官网下载得到

[FreeRTOS™ - FreeRTOS™](https://www.freertos.org/)

### 文件夹介绍

Demo:存放的是FreeRTOS的相关示例程序

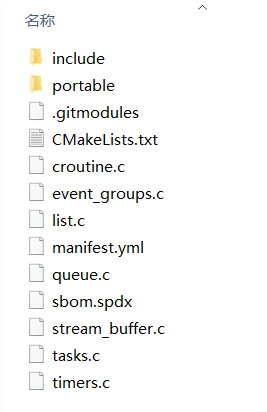
License:存放的是相关的许可信息

Source:存放的是FreeRTOS内核文件

这里我们需要用Source文件夹

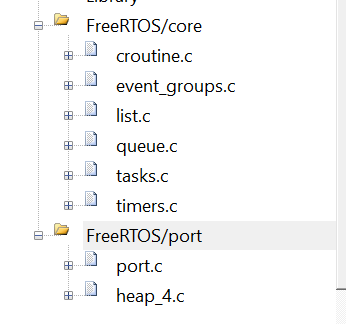
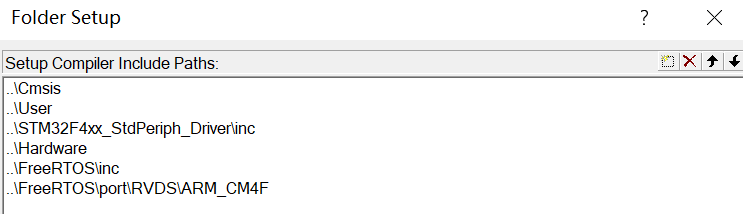
include 存放.c对应头文件

porttable文件夹是FreeRTOS和具体硬件练习在一起的桥梁，也叫移植层文件夹。

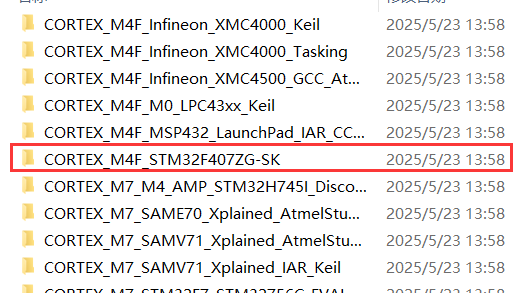
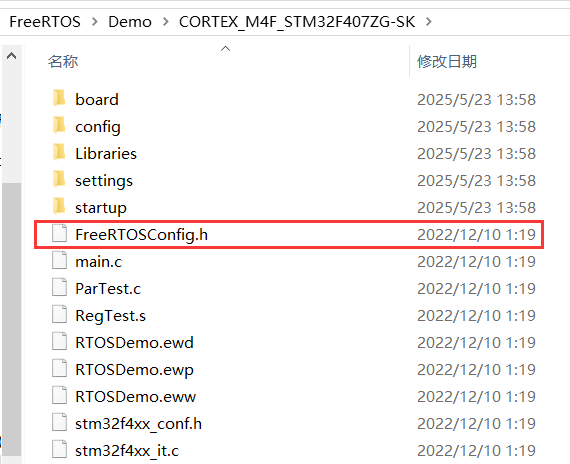


Portable文件夹中只需使用Keil、MemMang和RVDS三个文件夹。

### 添加文件

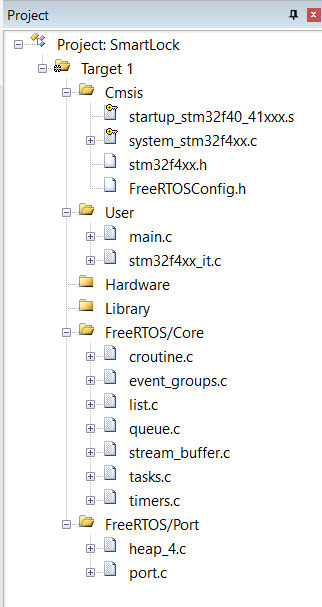
 

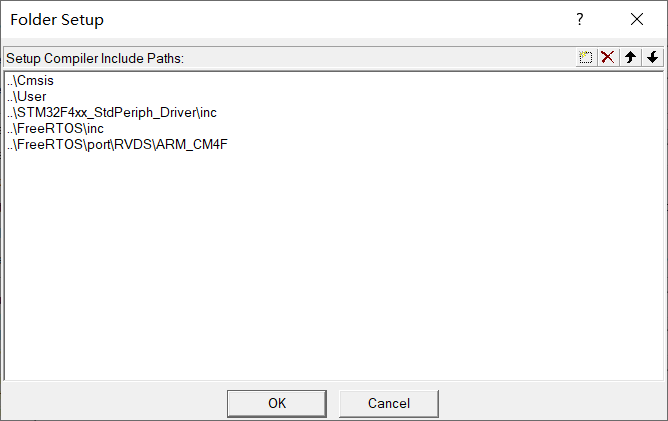
这时还缺少一个FreeRTOSConfig.h文件，请到Demo文件夹找到：

不同系列的芯片所用的FreeRTOSConfig.h文件也有差别。

移植后虚拟树





### 配置和裁剪

---------------------------------------------------------------------------------

将FreeRTOSconfig.h

#ifdef \_\_ICCARM\_\_

#include <stdint.h>

extern uint32\_t SystemCoreClock;

#endif

改成

#if defined(\_\_ICCARM\_\_) || defined(\_\_CC\_ARM) || defined(\_\_GNUC\_\_)

#include <stdint.h>

extern uint32\_t SystemCoreClock;

#endif

-----------------------------------------------------------------------------------

屏蔽stm32f4xx\_it.c中

void SVC\_Handler(void)

void PendSV\_Handler(void)

void SysTick\_Handler(void)

--------------------------------------------------------------------------------------

修改FreeRTOSconfig.h中

configUSE\_IDLE\_HOOK 0

configUSE\_TICK\_HOOK 0

configCHECK\_FOR\_STACK\_OVERFLOW 0

configUSE\_MALLOC\_FAILED\_HOOK 0

### 宏说明

这段代码是FreeRTOS操作系统的配置宏定义。这些宏定义用于配置FreeRTOS的行为和功能。下面是每个宏定义的简要说明：

configUSE\_PREEMPTION：定义为1时，启用抢占式调度，否则为合作式调度。

configUSE\_IDLE\_HOOK：定义为1时，启用空闲任务钩子函数，可以在这个钩子函数中添加用户代码。

configUSE\_TICK\_HOOK：定义为1时，启用滴答（tick）钩子函数，可以在这个钩子函数中添加用户代码。

configCPU\_CLOCK\_HZ：定义CPU时钟频率，通常设置为SystemCoreClock宏，后者通常在启动代码中定义。

configTICK\_RATE\_HZ：定义系统滴答频率，通常设置为1000Hz，即每1ms产生一个滴答。

configMAX\_PRIORITIES：定义任务优先级的最大数量。

configMINIMAL\_STACK\_SIZE：定义最小任务堆栈大小。

configTOTAL\_HEAP\_SIZE：定义系统堆的总大小。

configMAX\_TASK\_NAME\_LEN：定义任务名称的最大长度。

configUSE\_TRACE\_FACILITY：定义为1时，启用跟踪设施，用于跟踪任务和队列等。

configUSE\_16\_BIT\_TICKS：定义为1时，使用16位计数器，适用于滴答计数不会溢出的应用。

configIDLE\_SHOULD\_YIELD：定义为1时，空闲任务会在执行完所有任务后让出CPU。

configUSE\_MUTEXES：定义为1时，启用互斥量。

configQUEUE\_REGISTRY\_SIZE：定义队列注册表的大小，用于跟踪队列的状态。

configCHECK\_FOR\_STACK\_OVERFLOW：定义为2时，启用堆栈溢出检测。

configUSE\_RECURSIVE\_MUTEXES：定义为1时，启用递归互斥量。

configUSE\_MALLOC\_FAILED\_HOOK：定义为1时，启用内存分配失败钩子函数。

configUSE\_APPLICATION\_TASK\_TAG：定义为1时，允许给任务设置标签。

configUSE\_COUNTING\_SEMAPHORES：定义为1时，启用计数信号量。

configGENERATE\_RUN\_TIME\_STATS：定义为1时，生成运行时统计信息。

若系统需要多达十几个任务优先级，要修改哪些宏？

# 任务基础

## 认识任务

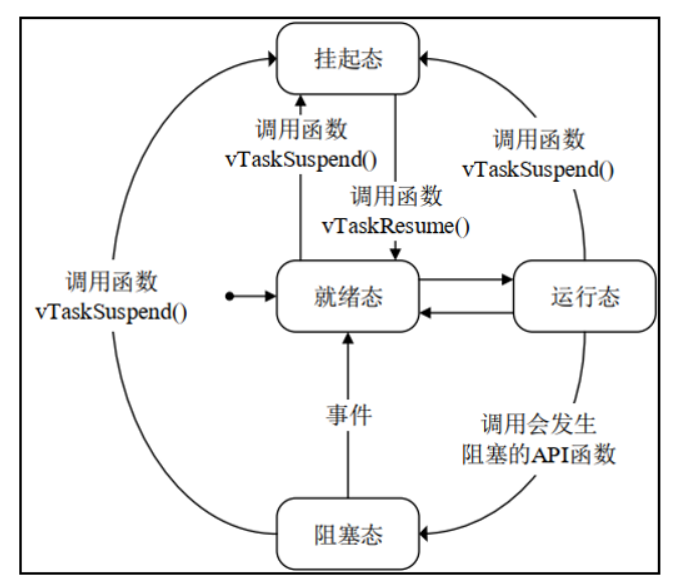
在之前我们学习STM32裸机编程时，程序是怎么运行的？我们通常写一个很大的 **main()** 函数，里面是一个 **while(1)** 超级循环。这种一个超级循环有各种短板：代码臃肿复杂，响应性差，资源利用率低，实时性差。

FreeRTOS中的任务(Task)，可以理解为一个独立的、无限循环的小程序。它是操作系统调度和运行的基本单位。每个任务都有自己的运行环境，为了能在任务之间进行切换，需要使用堆栈来保存任务的上下文环境，以便调度器在恢复任务时能使该任务从被切出去的地方继续执行。

1. 简单，每个任务尽可能完成简单操作。
2. 没有使用限制，每个任务都有可能被调度器选中运行。
3. 支持优先级，不同任务可设置不同优先级。
4. 支持抢占，高优先级任务能抢占（打断）低优先级任务
5. 每个任务都有堆栈，将导致RAM使用量增大。

## 任务的状态

FreeRTOS 的任务有4种状态，分别是运行态，就绪态，阻塞态，挂起态。  
运行态：若一个任务正在运行，就说这个任务处于运行态。处于运行态的任务就是当前正在使用处理器的任务。如果使用的是单核处理器，那么无论在什么时候都只有一个任务处于运行态。  
就绪态：处于就绪态的任务是指那些已经准备就绪，可以运行的任务，但当前未被调度器选中从而未运行，因为有其他优先级更高的任务正在运行。  
阻塞态：若一个任务当前正在等待某个外部事件（信号）发生，就说这个任务处于阻塞态。任务进入阻塞态会有一个超时时间，如果超过这个超时时间，任务就会退出阻塞态，即使所等待的事件还没有发生。  
挂起态：挂起态类似于阻塞态，但任务进入挂起态后不能被调度器选中从而进入运行态，而且进入挂起态的任务没有超时时间。  
FreeRTOS的任务永远处于这4种状态之一，各状态之间可以在等待事件或 API函数调用中进行转换。



在FreeRTOS中，任务有几种状态，有哪些状态？

阻塞态和挂起态有什么区别？

## 任务的优先级

任务可以设置不同的优先级，FreeRTOS对任务的优先级数量没有限制。优先级值越大，代表优先级级越高，这与使用ARM Cortex-M内核的STM32微控制器的中断优先级正好相反，务必高度重视。

**注意两点：**

**1、任务可以是同等优先级的；**

**2、中断可以打断任意任务；**

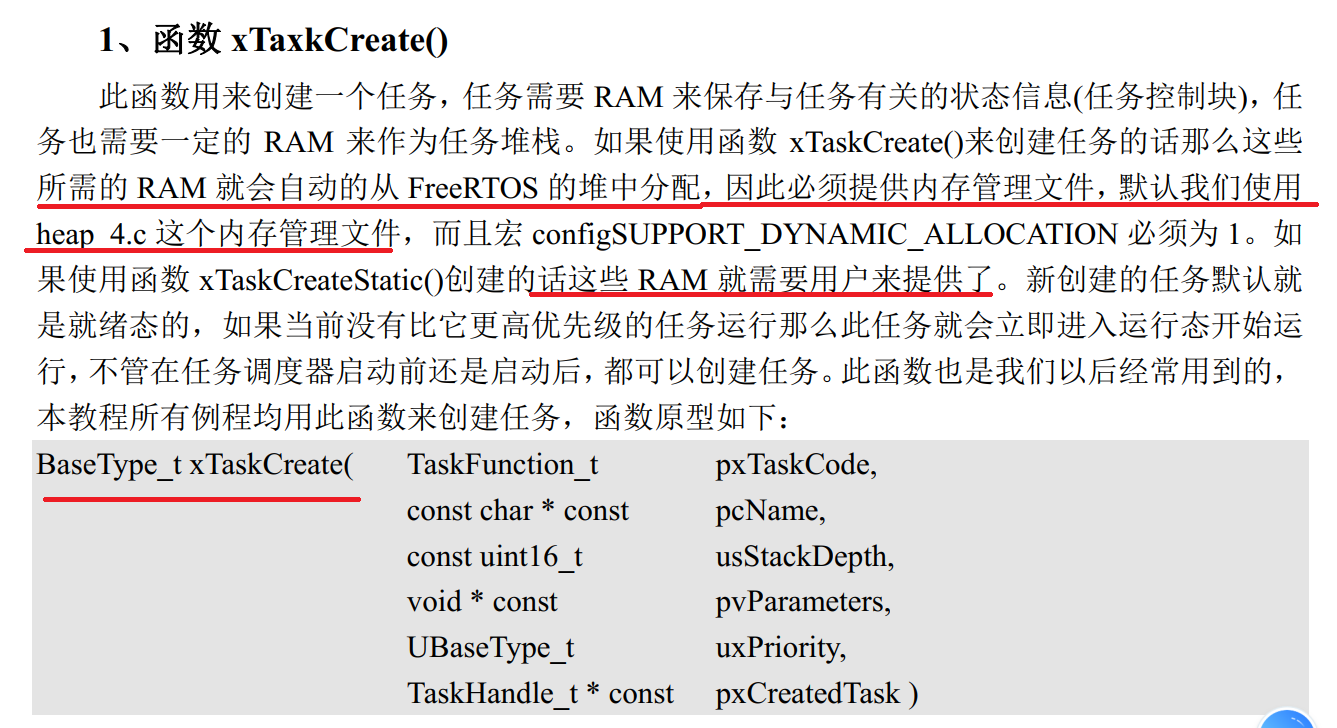
**在FreeRTOS中任务优先级不限，但会收到CPU的限制，本项目中：**

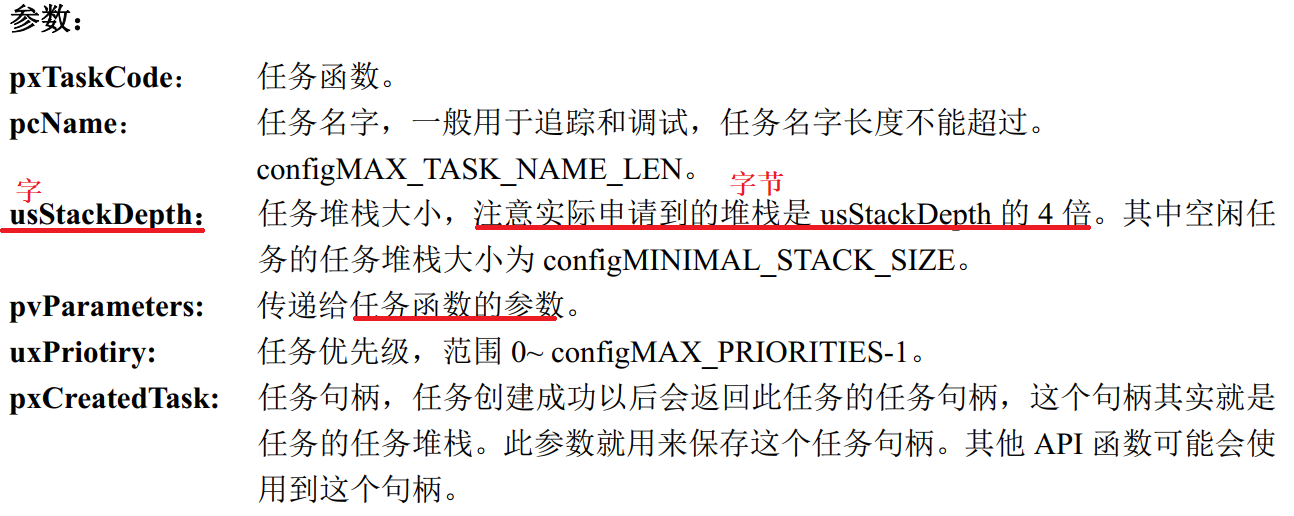


**思考：如果高优先级的任务一直在运行，会怎么样？**

如果高任务的优先级里面没有延时函数，那么，它就不会让出CPU的使用权，并且高优先级任务是死循环的，它将会一直执行下去，使得低优先级任务无法运行！！！

## 任务的创建和删除





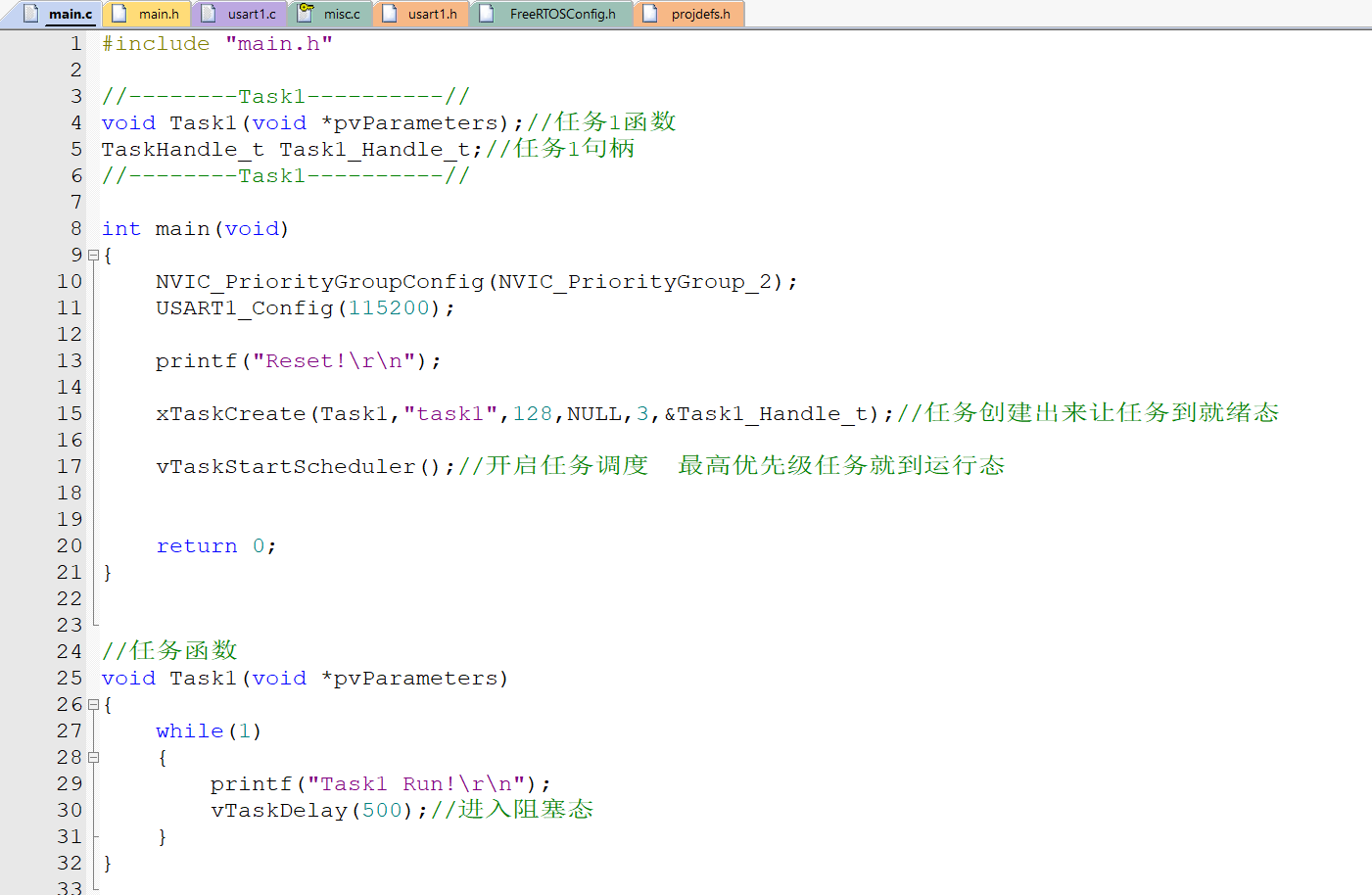
任务A和任务B

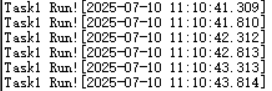
任务调度：不一定是从A调到B，有A再到A

任务切换：从A跳到B,或者从B跳到A

切换一定会调度，调度不一定切换

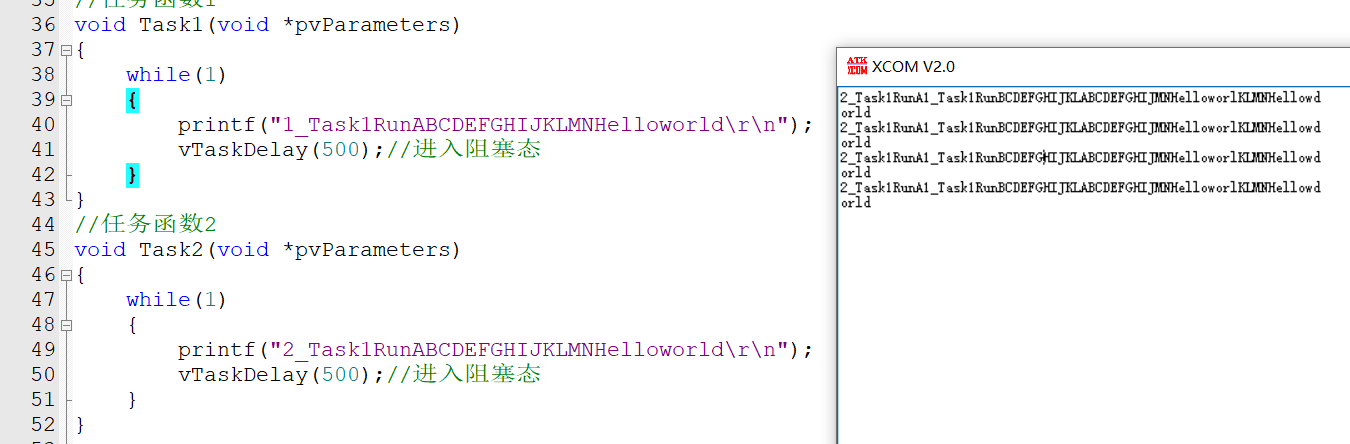
**任务创建示例：**





**时间片调度中，两个任务使用USART1**





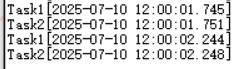
**该现象的原因：**任务2的1ms时间片内数据还没打印完，就轮到任务1的1ms时间片开始打印数据了，任务1的1ms时间片内数据还没打印完，任务2在用1ms时间片接着打印。

**任务优先级不同情况下：**



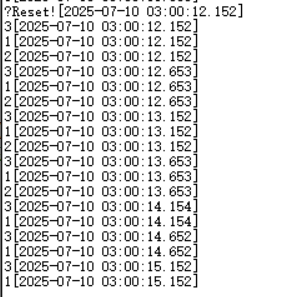


现象：任务1优先级高，任务1先执行。然后2执行



**任务删除：**

|  |
| --- |
| #include "main.h"  //--------Task1----------//  void Task1(void \*pvParameters);//任务1函数  TaskHandle\_t Task1\_Handle\_t;//任务1句柄  //--------Task1----------//  //--------Task2----------//  void Task2(void \*pvParameters);//任务2函数  TaskHandle\_t Task2\_Handle\_t;//任务2句柄  //--------Task2----------//  //--------Task3----------//  void Task3(void \*pvParameters);//任务3函数  TaskHandle\_t Task3\_Handle\_t;//任务3句柄  //--------Task3----------//  int main(void)  {  NVIC\_PriorityGroupConfig(NVIC\_PriorityGroup\_2);  USART1\_Config(115200);    printf("Reset!\r\n");    xTaskCreate(Task1,"task1",128,NULL,4,&Task1\_Handle\_t);//任务创建出来让任务到就绪态  xTaskCreate(Task2,"task2",128,NULL,4,&Task2\_Handle\_t);//任务创建出来让任务到就绪态  xTaskCreate(Task3,"task3",128,NULL,4,&Task3\_Handle\_t);//任务创建出来让任务到就绪态    vTaskStartScheduler();//开启任务调度 最高优先级任务就到运行态    return 0;  }  //任务函数1  void Task1(void \*pvParameters)  {  int i = 0;  while(1)  {  printf("1\r\n");  i++;  if(i==5)  vTaskDelete(Task2\_Handle\_t);//挂起（暂停）  vTaskDelay(500);//进入阻塞态  }  }  //任务函数2  void Task2(void \*pvParameters)  {  while(1)  {  printf("2\r\n");  vTaskDelay(500);//进入阻塞态  }  }  //任务函数3  void Task3(void \*pvParameters)  {  while(1)  {  printf("3\r\n");  vTaskDelay(500);//进入阻塞态  }  } |



## 空闲任务

为什么会有空闲任务？

思考⼀个问题:在我们的Free RTOS中可以将所有任务阻塞吗？

不能。

①所有任务都阻塞而没有可运行的任务，会导致系统⽆法处理外部响应，这与实时操作系统的设计理念相悖。

②可能会出现死锁或系统崩溃。

因此，必须始终至少有⼀个任务可以进⼊运行状态，这就是空闲任务。

空闲任务（Idle Task）

在FreeRTOS中，空闲任务是在我们调用任务调度器函数的内部自动创建的，它是优先级最低的任务。它在系统中没有其他任务可以运行时执行。其主要作用包括：

1. CPU利用率：空闲任务的运行表明系统没有其他更高优先级的任务需要运行，因此空闲任务的执行时间可以被用来计算CPU的空闲时间，从而得出CPU利用率。
2. 内存管理：空闲任务负责清理被删除的任务堆栈内存。

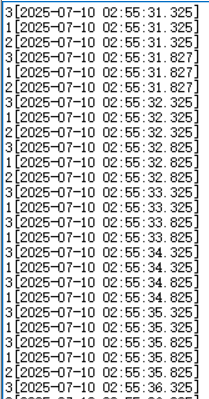
注意：空闲任务不需要我们进行创建！！！

注意：空闲任务它也是一个任务，只不过比较特殊而已，它是任务调度器自动帮我们进行创建的，在静态创建任务时，我们首先使能空闲任务的[API函数](https://so.csdn.net/so/search?q=API%E5%87%BD%E6%95%B0&spm=1001.2101.3001.7020)



## 任务的挂起和恢复

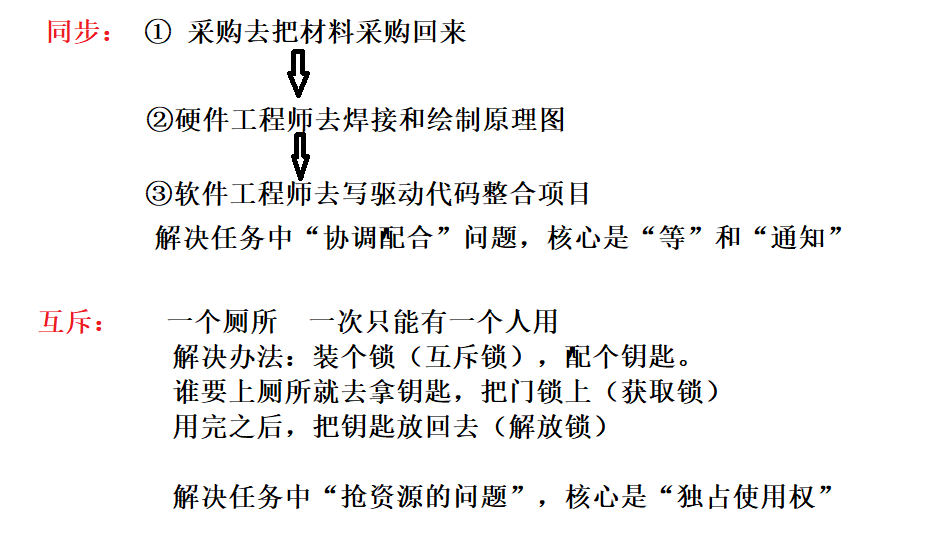
|  |
| --- |
| #include "main.h"  //--------Task1----------//  void Task1(void \*pvParameters);//任务1函数  TaskHandle\_t Task1\_Handle\_t;//任务1句柄  //--------Task1----------//  //--------Task2----------//  void Task2(void \*pvParameters);//任务2函数  TaskHandle\_t Task2\_Handle\_t;//任务2句柄  //--------Task2----------//  //--------Task3----------//  void Task3(void \*pvParameters);//任务3函数  TaskHandle\_t Task3\_Handle\_t;//任务3句柄  //--------Task3----------//  int main(void)  {  NVIC\_PriorityGroupConfig(NVIC\_PriorityGroup\_2);  USART1\_Config(115200);    printf("Reset!\r\n");    xTaskCreate(Task1,"task1",128,NULL,4,&Task1\_Handle\_t);//任务创建出来让任务到就绪态  xTaskCreate(Task2,"task2",128,NULL,4,&Task2\_Handle\_t);//任务创建出来让任务到就绪态  xTaskCreate(Task3,"task3",128,NULL,4,&Task3\_Handle\_t);//任务创建出来让任务到就绪态    vTaskStartScheduler();//开启任务调度 最高优先级任务就到运行态    return 0;  }  //任务函数1  void Task1(void \*pvParameters)  {  int i = 0;  while(1)  {  printf("1\r\n");  i++;  if(i==5)  vTaskSuspend(Task2\_Handle\_t);//挂起（暂停）  if(i==10)  vTaskResume(Task2\_Handle\_t);//恢复  vTaskDelay(500);//进入阻塞态  }  }  //任务函数2  void Task2(void \*pvParameters)  {  while(1)  {  printf("2\r\n");  vTaskDelay(500);//进入阻塞态  }  }  //任务函数3  void Task3(void \*pvParameters)  {  while(1)  {  printf("3\r\n");  vTaskDelay(500);//进入阻塞态  }  } |



# 任务通信

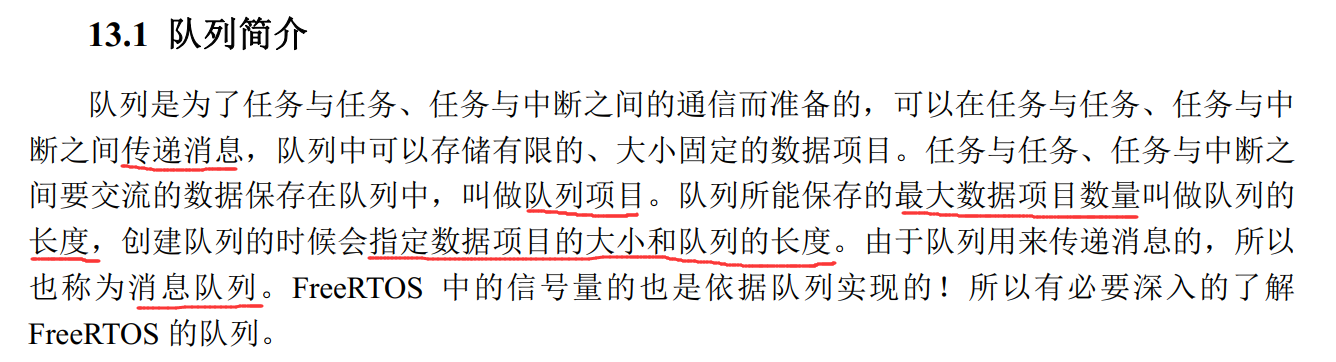
## 同步和互斥

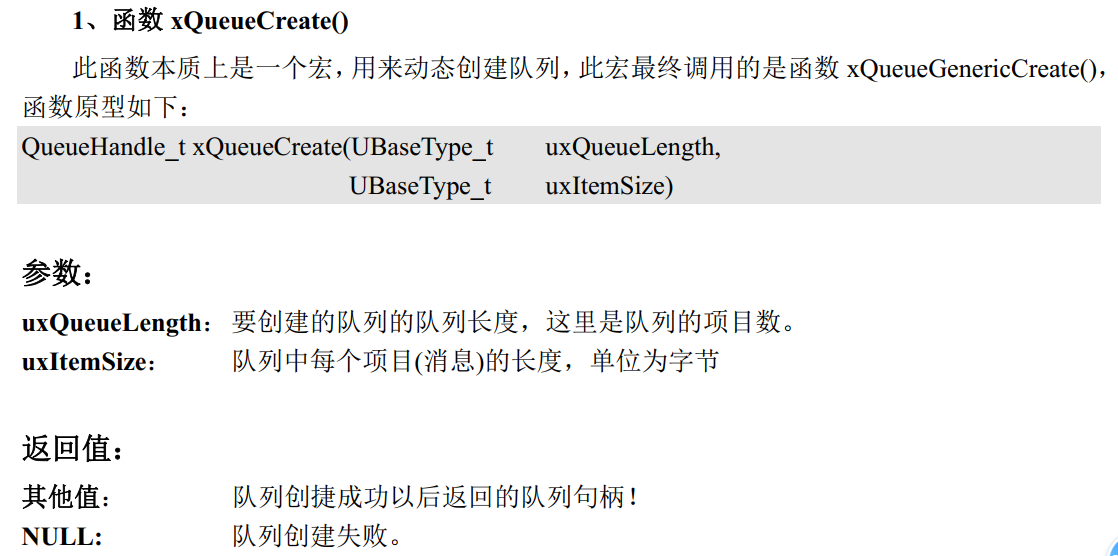
多个独立运行的任务如何安全的共享资源（互斥）以及如何有效的沟通和协调工作步骤（同步），FreeRTOS提供各种机制（队列，信号量，互斥量，任务通知，事件组）实现这些目标。

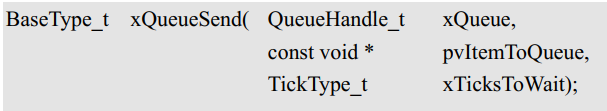


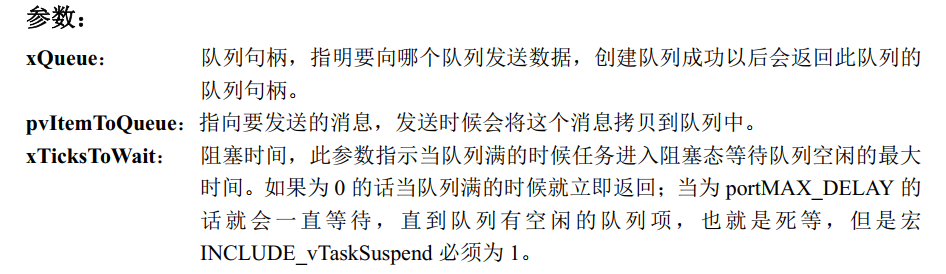
什么是同步？什么是互斥？

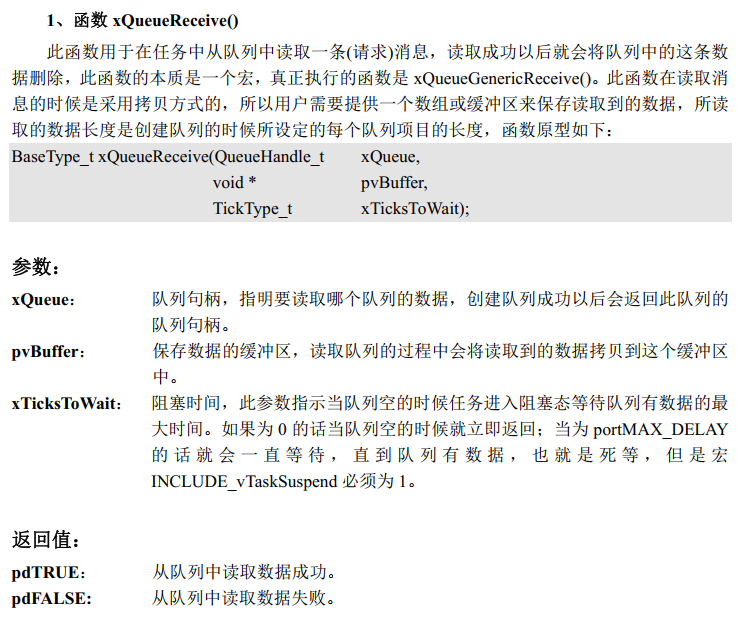
## 队列











|  |
| --- |
| #include "appTask.h"  //--------Task1----------//  void Task1(void \*pvParameters);//任务1函数  TaskHandle\_t Task1\_Handle\_t;//任务1句柄  //--------Task1----------//  //--------Task2----------//  void Task2(void \*pvParameters);//任务2函数  TaskHandle\_t Task2\_Handle\_t;//任务2句柄  //--------Task2----------//  ////--------Task3----------//  //void Task3(void \*pvParameters);//任务3函数  //TaskHandle\_t Task3\_Handle\_t;//任务3句柄  ////--------Task3----------//  //队列句柄  QueueHandle\_t queue;  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \*函 数 名：AppStartTask  \*函数功能：串口初始化  \*参 数：Baud  \*返 回 值：None  \*备 注：  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  void AppStartTask(void)  {  queue = xQueueCreate(64,sizeof(u8));    taskENTER\_CRITICAL();//进入临界区(关中断)  xTaskCreate(Task1,"task1",128,NULL,4,&Task1\_Handle\_t);//任务创建出来让任务到就绪态  xTaskCreate(Task2,"task2",128,NULL,4,&Task2\_Handle\_t);//任务创建出来让任务到就绪态  // xTaskCreate(Task3,"task3",128,NULL,4,&Task3\_Handle\_t);//任务创建出来让任务到就绪态    taskEXIT\_CRITICAL();//退出临界区(开中断)    vTaskStartScheduler();//开启任务调度 最高优先级任务就到运行态  }  //任务函数1  void Task1(void \*pvParameters)  {  int i = 0;  u8 data[5] = {1,2,3,4,5};  while(1)  {  for(i = 0;i<5;i++)  {  xQueueSend(queue,data+i,0);//向队列中发送消息  }  printf("T1\r\n");  delay\_ms(500);//进入阻塞态  }  }  //任务函数2  void Task2(void \*pvParameters)  {  u8 i = 0;  u8 data[5] = {0};  while(1)  {  for(i=0;i<5;i++)  {  xQueueReceive(queue,data+i,portMAX\_DELAY);  }  for(i=0;i<5;i++)  {  printf("%d\r\n",data[i]);  }  printf("T2\r\n");  delay\_ms(500);//进入阻塞态  }  }  ////任务函数3  //void Task3(void \*pvParameters)  //{  // while(1)  // {  // printf("T3\r\n");  // vTaskDelay(500);//进入阻塞态  // }  //} |

