

title: 超详细的FreeRTOS移植全教程——基于srm32 author: 杰杰 top: false cover: false toc: true mathjax: false date: 2019-10-05 09:23:58 img: coverImg: password: summary: tags:

- FreeRTOS
- RTOS
- 操作系统 categories: 操作系统

准备






在移植之前，我们首先要获取到FreeRTOS的官方的源码包。这里我们提供两个下载链接:

```
一个是官网：http://www.freertos.org/ 另外一个 是代码托管网站：  
https://sourceforge.net/projects/freertos/files/FreeRTOS/
```

这里我们演示如何在代码托管网站里面下载。打开网站链接之后，我们选择FreeRTOS的最新版本V9.0.0（2016年），尽管现在FreeRTOS的版本已经更新到V10.0.1了，但是我们还是选择V9.0.0，因为内核很稳定，并且网上资料很多，因为V10.0.0版本之后是亚马逊收购了FreeRTOS之后才出来的版本，主要添加了一些云端组件，我们本书所讲的FreeRTOS是实时内核，采用V9.0.0版本足以。

简单介绍FreeRTOS

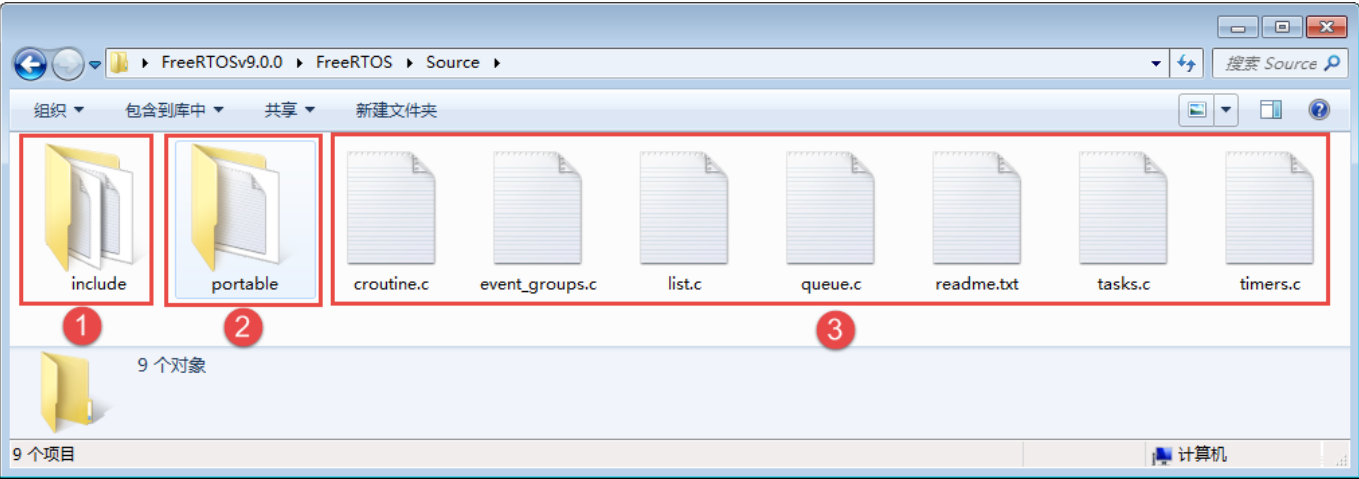
FreeRTOS包含Demo例程和内核源码（比较重要，我们就需要提取该目录下的大部分文件）。**Source**文件夹里面包含的是FreeRTOS内核的源代码，我们移植FreeRTOS的时候就需要这部分源代码；**Demo**文件夹里面包含了FreeRTOS官方为各个单片机移植好的工程代码，FreeRTOS为了推广自己，会给各种半导体厂商的评估板写好完整的工程程序，这些程序就放在Demo这个目录下，这部分Demo非常有参考价值。

| > FreeRTOS > FreeRTOS > Demo > FreeRTOSv9.0.0 > FreeRTOS | | |
|---|------------------|---------------|
| 名称 | 修改日期 | 类型 |
|  Demo | 2018/9/2 13:51 | 文件夹 |
|  License | 2018/9/2 13:51 | 文件夹 |
|  Source | 2018/9/2 13:51 | 文件夹 |
|  links_to_doc_pages_for_the_demo_pr... | 2014/10/30 23:56 | Internet 快捷方式 |
|  readme.txt | 2013/9/17 16:20 | TXT 文件 |

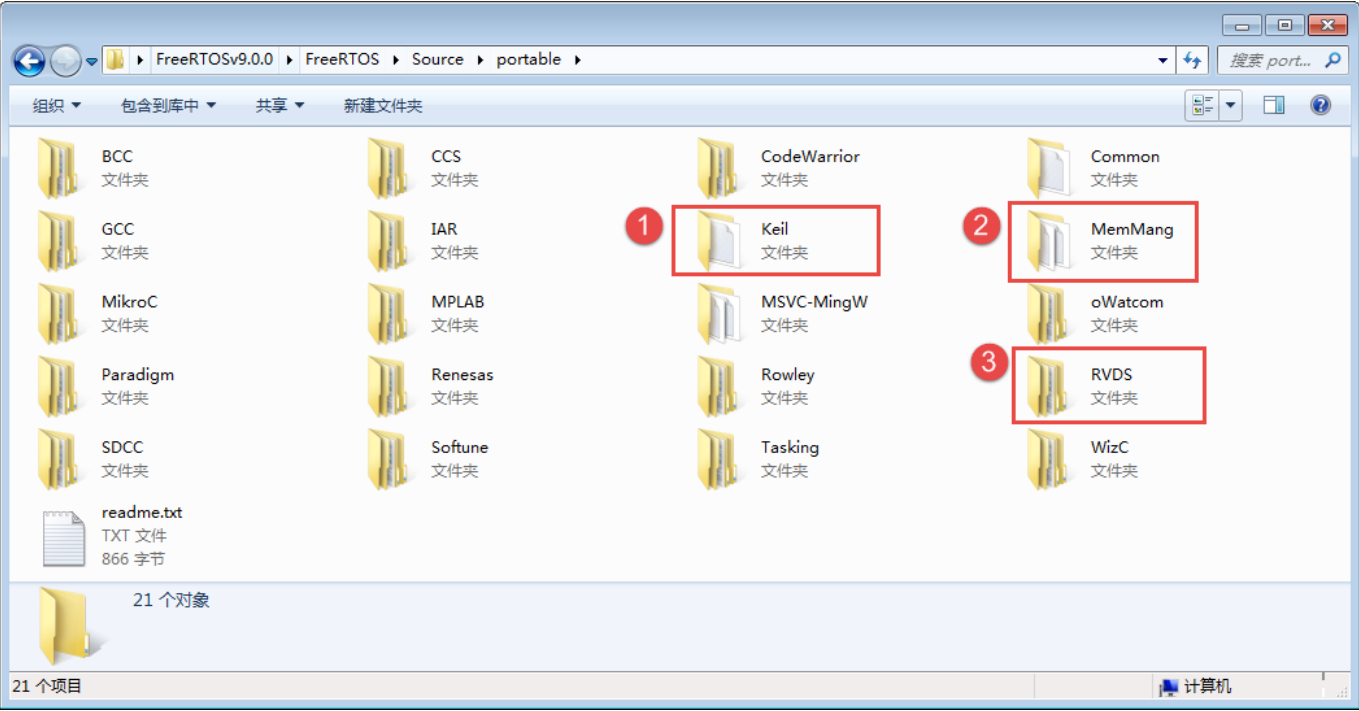
Source文件夹

这里我们再重点分析下FreeRTOS/ Source文件夹下的文件，①和③包含的是FreeRTOS的通用的头文件和C文件，这两部分的文件试用于各种编译器和处理器，是通用的。需要移植的头文件和C文件放在②portblle这个文件

夹。



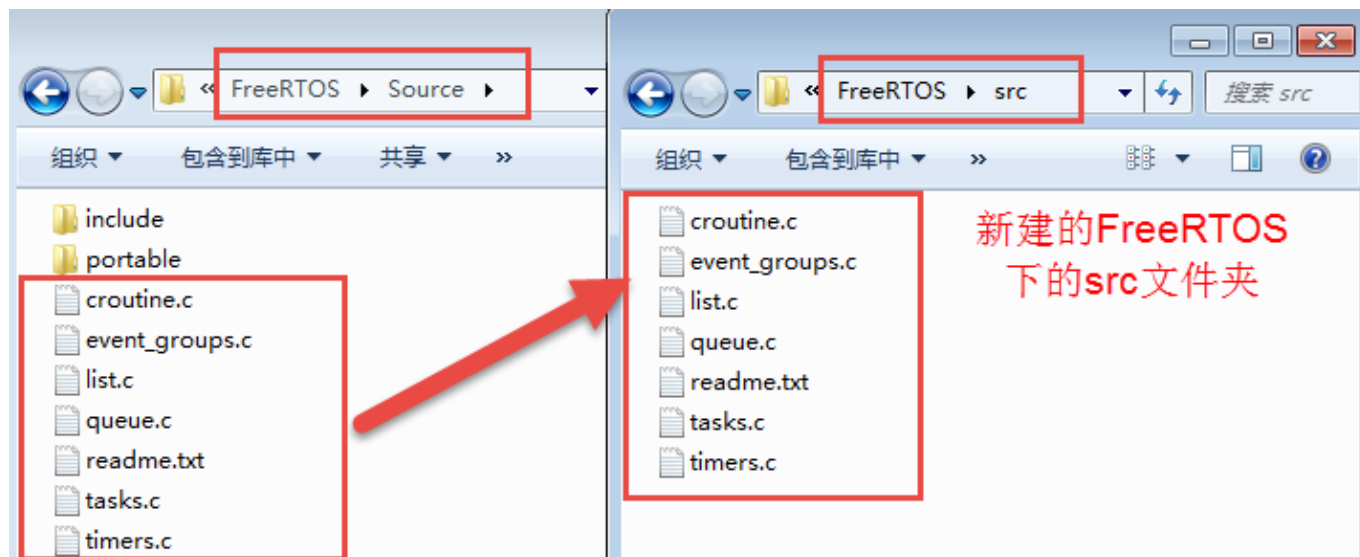
portble文件夹，是与编译器相关的文件夹，在不同的编译器中使用不同的支持文件。①中的KEIL就是我们就是使用的编译器，其实KEIL里面的内容跟RVDS里面的内容一样，所以我们只需要③RVDS文件夹里面的内容即可，里面包含了各种处理器相关的文件夹，从文件夹的名字我们就非常熟悉了，我们学习的STM32有M0、M3、M4等各种系列，FreeRTOS是一个软件，单片机是一个硬件，FreeRTOS要想运行在一个单片机上面，它们就必须关联在一起。MemMang文件夹下存放的是跟内存管理相关的源文件。



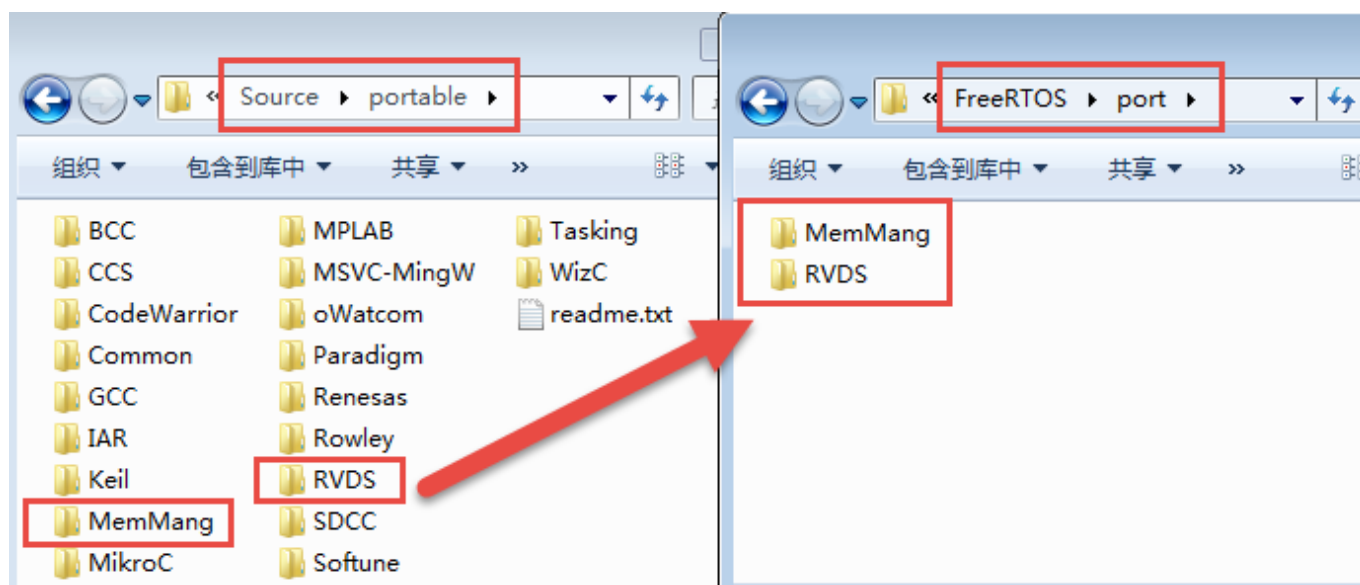
移植过程

提取源码

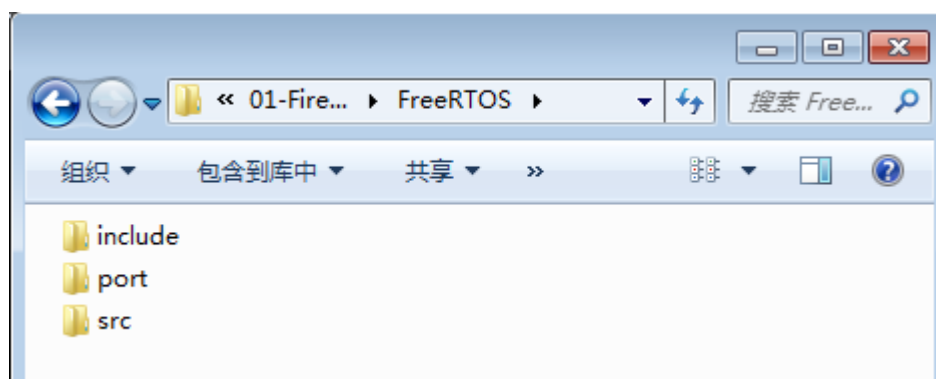
- 1. 首先在我们的STM32裸机工程模板根目录下新建一个文件夹，命名为“FreeRTOS”，并且在FreeRTOS文件夹下新建两个空文件夹，分别命名为“src”与“port”，src文件夹用于保存FreeRTOS中的核心源文件，也就是我们常说的'.c文件’，port文件夹用于保存内存管理以及处理器架构相关代码，这些代码FreeRTOS官方已经提供给我们的，直接使用即可，在前面已经说了，FreeRTOS是软件，我们的开发版是硬件，软硬件必须有桥梁来连接，这些与处理器架构相关的代码，可以称之为RTOS硬件接口层，它们位于FreeRTOS/Source/Portable文件夹下。
- 2. 打开FreeRTOS V9.0.0源码，在“FreeRTOSv9.0.0\FreeRTOS\Source”目录下找到所有的'.c文件’，将它们拷贝到我们新建的src文件夹中，



3. 打开FreeRTOS V9.0.0源码，在“FreeRTOSv9.0.0\FreeRTOS\Source\portable”目录下找到“MemMang”文件夹与“RVDS”文件夹，将它们拷贝到我们新建的port文件夹中



4. 打开FreeRTOS V9.0.0源码，在“FreeRTOSv9.0.0\FreeRTOS\Source”目录下找到“include”文件夹，它是我们需要用到FreeRTOS的一些头文件，将它直接拷贝到我们新建的FreeRTOS文件夹中，完成这一步之后就可以看到我们新建的FreeRTOS文件夹已经有3个文件夹，这3个文件夹就包含FreeRTOS的核心文件，至此，FreeRTOS的源码就提取完成。

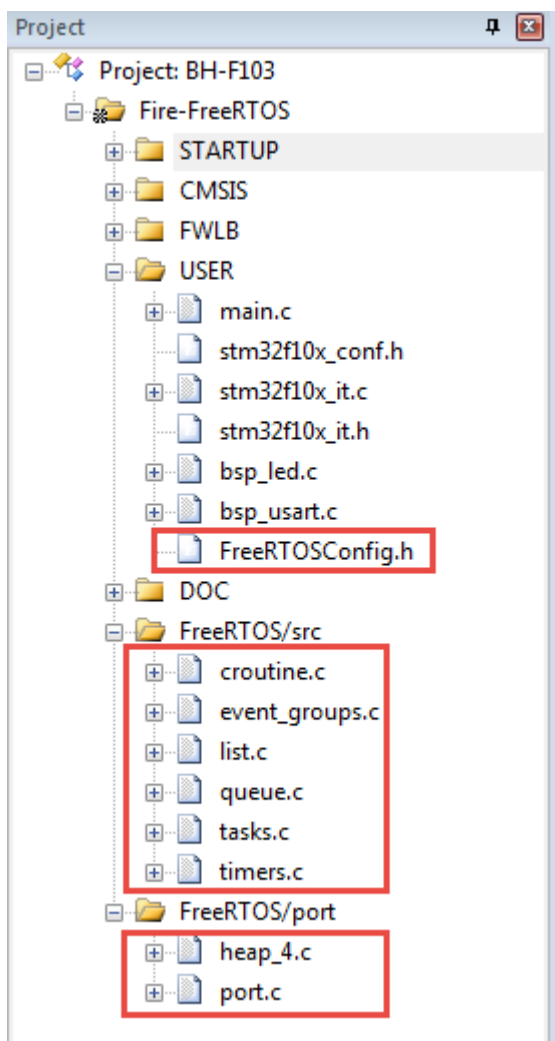


添加到工程

添加**FreeRTOSConfig.h**文件 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。打开FreeRTOSv9.0.0源码，

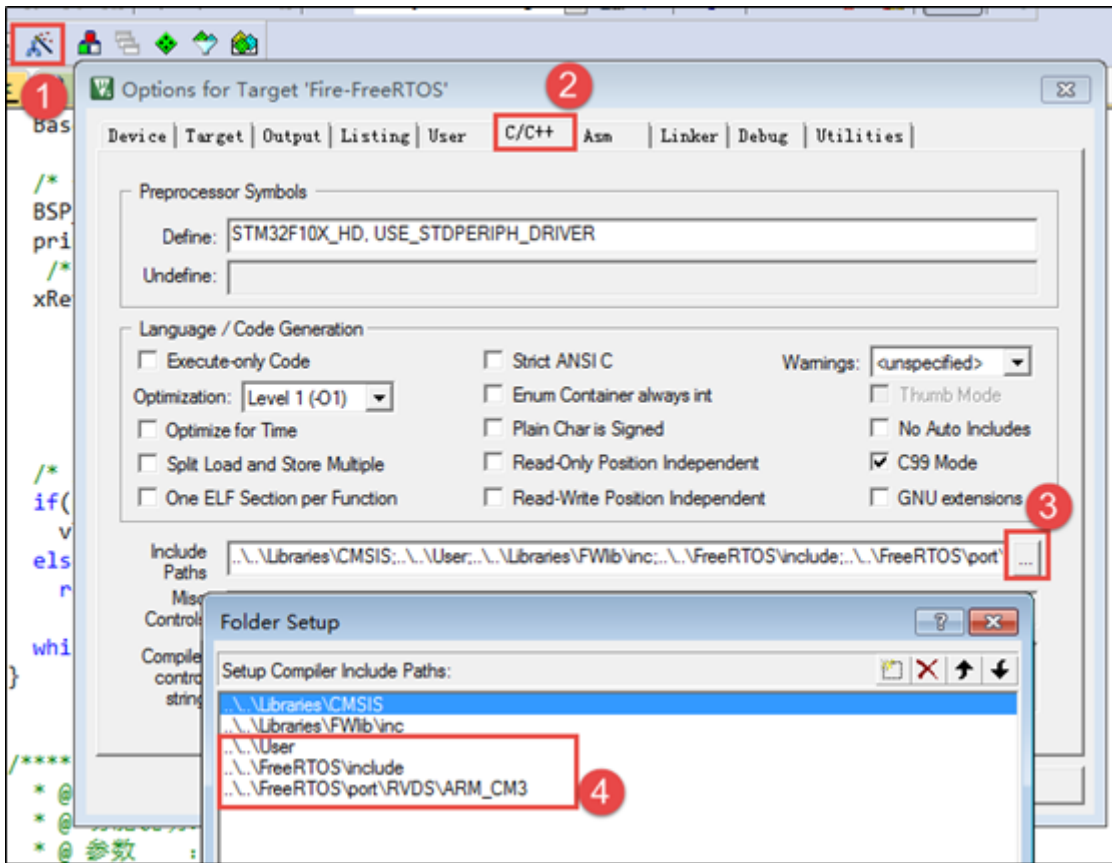
在“FreeRTOSv9.0.0\FreeRTOS\Demo”文件夹下面找到“CORTEX_STM32F103_Keil”这个文件夹，双击打开，在其根目录下找到这个“FreeRTOSConfig.h”文件，然后拷贝到我们工程的user文件夹下即可，等下我们需要对这个文件进行修改。

创建工程分组 接下来我们在mdk里面新建FreeRTOS/src和FreeRTOS/port两个组文件夹，其中FreeRTOS/src用于存放src文件夹的内容，FreeRTOS/port用于存放port\MemMang文件夹 与port\RVDS\ARM_CM3文件夹的内容。然后将工程文件中FreeRTOS的内容添加到工程中去，按照已经新建的分组添加我们的FreeRTOS工程源码。在FreeRTOS/port分组中添加MemMang文件夹中的文件只需选择其中一个即可，我们选择“heap_4.c”，这是FreeRTOS的一个内存管理源码文件。添加完成后：



**** 添加头文件路径**** 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FreeRTOS\include和FreeRTOS\port\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user文件夹下，所以user的路径

也要加到开发环境里面。



修改FreeRTOSConfig.h

FreeRTOSConfig.h是直接从demo文件夹下面拷贝过来的，该头文件对裁剪整个FreeRTOS所需的功能的宏均做了定义，有些宏定义被使能，有些宏定义被失能，一开始我们只需要配置最简单的功能即可。要想随心所欲的配置FreeRTOS的功能，我们必须对这些宏定义的功能有所掌握，下面我们先简单的介绍下这些宏定义的含义，然后再对这些宏定义进行修改。

```
#ifndef FREERTOS_CONFIG_H
#define FREERTOS_CONFIG_H

#include "stm32f10x.h"
#include "bsp_usart.h"

//针对不同的编译器调用不同的stdint.h文件
#if defined(__ICCARM__) || defined(__CC_ARM) || defined(__GNUC__)
#include <stdint.h>
extern uint32_t SystemCoreClock;
#endif

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

/*****
*
* FreeRTOS基础配置配置选项
*****/
```

```

/* 置1: RTOS使用抢占式调度器; 置0: RTOS使用协作式调度器(时间片)
*
* 注: 在多任务管理机制上, 操作系统可以分为抢占式和协作式两种。
* 协作式操作系统是任务主动释放CPU后, 切换到下一个任务。
* 任务切换的时机完全取决于正在运行的任务。
*/
#define configUSE_PREEMPTION 1

//1使能时间片调度(默认式使能的)
#define configUSE_TIME_SLICING 1

/* 某些运行FreeRTOS的硬件有两种方法选择下一个要执行的任务:
* 通用方法和特定于硬件的方法(以下简称“特殊方法”)。
*
* 通用方法:
* 1.configUSE_PORT_OPTIMISED_TASK_SELECTION 为 0 或者硬件不支持这种特殊方法。
* 2.可以用于所有FreeRTOS支持的硬件
* 3.完全用C实现, 效率略低于特殊方法。
* 4.不强制要求限制最大可用优先级数目
* 特殊方法:
* 1.必须将configUSE_PORT_OPTIMISED_TASK_SELECTION设置为1。
* 2.依赖一个或多个特定架构的汇编指令(一般是类似计算前导零[CLZ]指令)。
* 3.比通用方法更高效
* 4.一般强制限定最大可用优先级数目为32
* 一般是硬件计算前导零指令, 如果所使用的, MCU没有这些硬件指令的话此宏应该设置为0!
*/
#define configUSE_PORT_OPTIMISED_TASK_SELECTION 1

/* 置1: 使能低功耗tickless模式; 置0: 保持系统节拍(tick)中断一直运行
* 假设开启低功耗的话可能会导致下载出现问题, 因为程序在睡眠中, 可用以下办法解决
*
* 下载方法:
* 1.将开发版正常连接好
* 2.按住复位按键, 点击下载瞬间松开复位按键
*
* 1.通过跳线帽将 BOOT 0 接高电平(3.3V)
* 2.重新上电, 下载
*
* 1.使用FlyMcu擦除一下芯片, 然后进行下载
* STMISP -> 清除芯片(z)
*/
#define configUSE_TICKLESS_IDLE 0

/*
* 写入实际的CPU内核时钟频率, 也就是CPU指令执行频率, 通常称为Fclk
* Fclk为供给CPU内核的时钟信号, 我们所说的cpu主频为 XX MHz,
* 就是指的这个时钟信号, 相应的, 1/Fclk即为cpu时钟周期;
*/
#define configCPU_CLOCK_HZ (SystemCoreClock)

//RTOS系统节拍中断的频率。即一秒中断的次数, 每次中断RTOS都会进行任务调度
#define configTICK_RATE_HZ ((

```

```

TickType_t )1000)

//可使用的最大优先级
#define configMAX_PRIORITIES                (32)

//空闲任务使用的堆栈大小
#define configMINIMAL_STACK_SIZE            ((unsigned
short)128)

//任务名字字符串长度
#define configMAX_TASK_NAME_LEN            (16)

//系统节拍计数器变量数据类型，1表示为16位无符号整形，0表示为32位无符号整形
#define configUSE_16_BIT_TICKS              0

//空闲任务放弃CPU使用权给其他同优先级的用户任务
#define configIDLE_SHOULD_YIELD              1

//启用队列
#define configUSE_QUEUE_SETS                1

//开启任务通知功能，默认开启
#define configUSE_TASK_NOTIFICATIONS        1

//使用互斥信号量
#define configUSE_MUTEXES                    1

//使用递归互斥信号量
#define configUSE_RECURSIVE_MUTEXES        1

//为1时使用计数信号量
#define configUSE_COUNTING_SEMAPHORES        1

/* 设置可以注册的信号量和消息队列个数 */
#define configQUEUE_REGISTRY_SIZE            10

#define configUSE_APPLICATION_TASK_TAG      0

/*****
FreeRTOS与内存申请有关配置选项
*****/
//支持动态内存申请
#define configSUPPORT_DYNAMIC_ALLOCATION      1
//支持静态内存
#define configSUPPORT_STATIC_ALLOCATION        0
//系统所有总的堆大小
#define configTOTAL_HEAP_SIZE                ((size_t)
(36*1024))

/*****
FreeRTOS与钩子函数有关的配置选项
*****/

```



```

/* 置1: 使用空闲钩子 (Idle Hook类似于回调函数); 置0: 忽略空闲钩子
*
* 空闲任务钩子是一个函数, 这个函数由用户来实现,
* FreeRTOS规定了函数的名字和参数: void vApplicationIdleHook(void ),
* 这个函数在每个空闲任务周期都会被调用
* 对于已经删除的RTOS任务, 空闲任务可以释放分配给它们的堆栈内存。
* 因此必须保证空闲任务可以被CPU执行
* 使用空闲钩子函数设置CPU进入省电模式是很常见的
* 不可以调用会引起空闲任务阻塞的API函数
*/
#define configUSE_IDLE_HOOK 0

/* 置1: 使用时间片钩子 (Tick Hook); 置0: 忽略时间片钩子
*
*
* 时间片钩子是一个函数, 这个函数由用户来实现,
* FreeRTOS规定了函数的名字和参数: void vApplicationTickHook(void )
* 时间片中断可以周期性的调用
* 函数必须非常短小, 不能大量使用堆栈,
* 不能调用以"FromISR" 或 "FROM_ISR"结尾的API函数
*/
/*xTaskIncrementTick函数是在xPortSysTickHandler中断函数中被调用的。因此,
vApplicationTickHook()函数执行的时间必须很短才行*/
#define configUSE_TICK_HOOK 0

//使用内存申请失败钩子函数
#define configUSE_MALLOC_FAILED_HOOK 0

/*
* 大于0时启用堆栈溢出检测功能, 如果使用此功能
* 用户必须提供一个栈溢出钩子函数, 如果使用的的话
* 此值可以为1或者2, 因为有两种栈溢出检测方法 */
#define configCHECK_FOR_STACK_OVERFLOW 0

/*****
FreeRTOS与运行时间和任务状态收集有关的配置选项
*****/
//启用运行时间统计功能
#define configGENERATE_RUN_TIME_STATS 0
//启用可视化跟踪调试
#define configUSE_TRACE_FACILITY 0
/* 与宏configUSE_TRACE_FACILITY同时为1时会编译下面3个函数
* prvWriteNameToBuffer()
* vTaskList(),
* vTaskGetRunTimeStats()
*/
#define configUSE_STATS_FORMATTING_FUNCTIONS 1

/*****
FreeRTOS与协程有关的配置选项
*****/
//启用协程, 启用协程以后必须添加文件croutine.c

```



```

#define configUSE_CO_ROUTINES                                0
//协程的有效优先级数目
#define configMAX_CO_ROUTINE_PRIORITIES                    ( 2 )

/*****
FreeRTOS与软件定时器有关的配置选项
*****/
//启用软件定时器
#define configUSE_TIMERS                                    1
//软件定时器优先级
#define configTIMER_TASK_PRIORITY                        (configMAX_PRIORITIES-1)
//软件定时器队列长度
#define configTIMER_QUEUE_LENGTH                        10
//软件定时器任务堆栈大小
#define configTIMER_TASK_STACK_DEPTH                    (configMINIMAL_STACK_SIZE*2)

/*****
FreeRTOS可选函数配置选项
*****/
#define INCLUDE_xTaskGetSchedulerState                    1
#define INCLUDE_vTaskPrioritySet                          1
#define INCLUDE_uxTaskPriorityGet                        1
#define INCLUDE_vTaskDelete                              1
#define INCLUDE_vTaskCleanUpResources                    1
#define INCLUDE_vTaskSuspend                            1
#define INCLUDE_vTaskDelayUntil                         1
#define INCLUDE_vTaskDelay                              1
#define INCLUDE_eTaskGetState                            1
#define INCLUDE_xTimerPendFunctionCall                    1
//#define INCLUDE_xTaskGetCurrentTaskHandle                1
//#define INCLUDE_uxTaskGetStackHighWaterMark                0
//#define INCLUDE_xTaskGetIdleTaskHandle                    0

/*****
FreeRTOS与中断有关的配置选项
*****/
#ifdef __NVIC_PRIO_BITS
    #define configPRIO_BITS                                __NVIC_PRIO_BITS
#else
    #define configPRIO_BITS                                4
#endif
//中断最低优先级
#define configLIBRARY_LOWEST_INTERRUPT_PRIORITY          15

//系统可管理的最高中断优先级
#define configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY    5

#define configKERNEL_INTERRUPT_PRIORITY                  (
configLIBRARY_LOWEST_INTERRUPT_PRIORITY << (8 - configPRIO_BITS) ) /* 240 */

#define configMAX_SYSCALL_INTERRUPT_PRIORITY            (
configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY << (8 - configPRIO_BITS) )

```

```

/*****
FreeRTOS与中断服务函数有关的配置选项
*****/

#define xPortPendSVHandler      PendSV_Handler
#define vPortSVCHandler        SVC_Handler

/* 以下为使用Percepio Tracealyzer需要的东西，不需要时将 configUSE_TRACE_FACILITY 定义为 0 */
#if ( configUSE_TRACE_FACILITY == 1 )
#include "trcRecorder.h"
#define INCLUDE_xTaskGetCurrentTaskHandle      1    // 启用一个可选函数（该函数被 Trace源码使用，默认该值为0 表示不用）
#endif

#endif /* FREERTOS_CONFIG_H */

```

修改stm32f10x_it.c

SysTick中断服务函数是一个非常重要的函数，FreeRTOS所有跟时间相关的事情都在里面处理，SysTick就是FreeRTOS的一个心跳时钟，驱动着FreeRTOS的运行，就像人的心跳一样，假如没有心跳，我们就相当于“死了”，同样的，FreeRTOS没有了心跳，那么它就会卡死在某个地方，不能进行任务调度，不能运行任何东西，因此我们需要实现一个FreeRTOS的心跳时钟，FreeRTOS帮我们实现了SysTick的启动的配置：在port.c文件中已经实现vPortSetupTimerInterrupt()函数，并且FreeRTOS通用的SysTick中断服务函数也实现了：在port.c文件中已经实现xPortSysTickHandler()函数，所以移植的时候只需要我们在stm32f10x_it.c文件中实现我们对应（STM32）平台上的SysTick_Handler()函数即可。FreeRTOS为开发者考虑得特别多，PendSV_Handler()与SVC_Handler()这两个很重要的函数都帮我们实现了，在在port.c文件中已经实现xPortPendSV_Handler()与vPortSVCHandler()函数，防止我们自己实现不了，那么在stm32f10x_it.c中就需要我们注释掉PendSV_Handler()与SVC_Handler()这两个函数了。

```

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

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

extern void xPortSysTickHandler(void);

//systick中断服务函数
void SysTick_Handler(void)
{
    #if (INCLUDE_xTaskGetSchedulerState == 1 )
        if (xTaskGetSchedulerState() != taskSCHEDULER_NOT_STARTED)
        {
            #endif /* INCLUDE_xTaskGetSchedulerState */

```

```

        xPortSysTickHandler();
    #if (INCLUDE_xTaskGetSchedulerState == 1 )
    }
    #endif /* INCLUDE_xTaskGetSchedulerState */
}

```

创建任务

这里，我们创建一个单任务，任务使用的栈和任务控制块是在创建任务的时候FreeRTOS动态分配的。任务必须是一个死循环，否则任务将通过LR返回，如果LR指向了非法的内存就会产生HardFault_Handler，而FreeRTOS指向一个死循环，那么任务返回之后就在死循环中执行，这样子的任务是不安全的，所以避免这种情况，任务一般都是死循环并且无返回值的。并且每个任务循环主体中应该有阻塞任务的函数，否则就会饿死比它优先级更低的任务！！！！

```

/* FreeRTOS头文件 */
#include "FreeRTOS.h"
#include "task.h"
/* 开发板硬件bsp头文件 */
#include "bsp_led.h"

static void AppTaskCreate(void);/* AppTask任务 */

/* 创建任务句柄 */
static TaskHandle_t AppTask_Handle = NULL;

int main(void)
{
    BaseType_t xReturn = pdPASS; /* 定义一个创建信息返回值，默认为pdPASS */

    /* 开发板硬件初始化 */
    BSP_Init();

    /* 创建AppTaskCreate任务 */
    xReturn = xTaskCreate((TaskFunction_t )AppTask, /* 任务入口函数 */
                        (const char* )"AppTask", /* 任务名字 */
                        (uint16_t )512, /* 任务栈大小 */
                        (void* )NULL, /* 任务入口函数参数 */
                        (UBaseType_t )1, /* 任务的优先级 */
                        (TaskHandle_t* )&AppTask_Handle); /* 任务控制块指针 */

    /* 启动任务调度 */
    if(pdPASS == xReturn)
        vTaskStartScheduler(); /* 启动任务，开启调度 */
    else
        return -1;

    while(1); /* 正常不会执行到这里 */
}

static void AppTask(void* parameter)
{
    while (1)

```

```
{  
    LED1_ON;  
    vTaskDelay(500);    /* 延时500个tick */  
    LED1_OFF;  
    vTaskDelay(500);    /* 延时500个tick */  
}
```

喜欢就关注我吧！



相关代码可以在公众号后台获取。