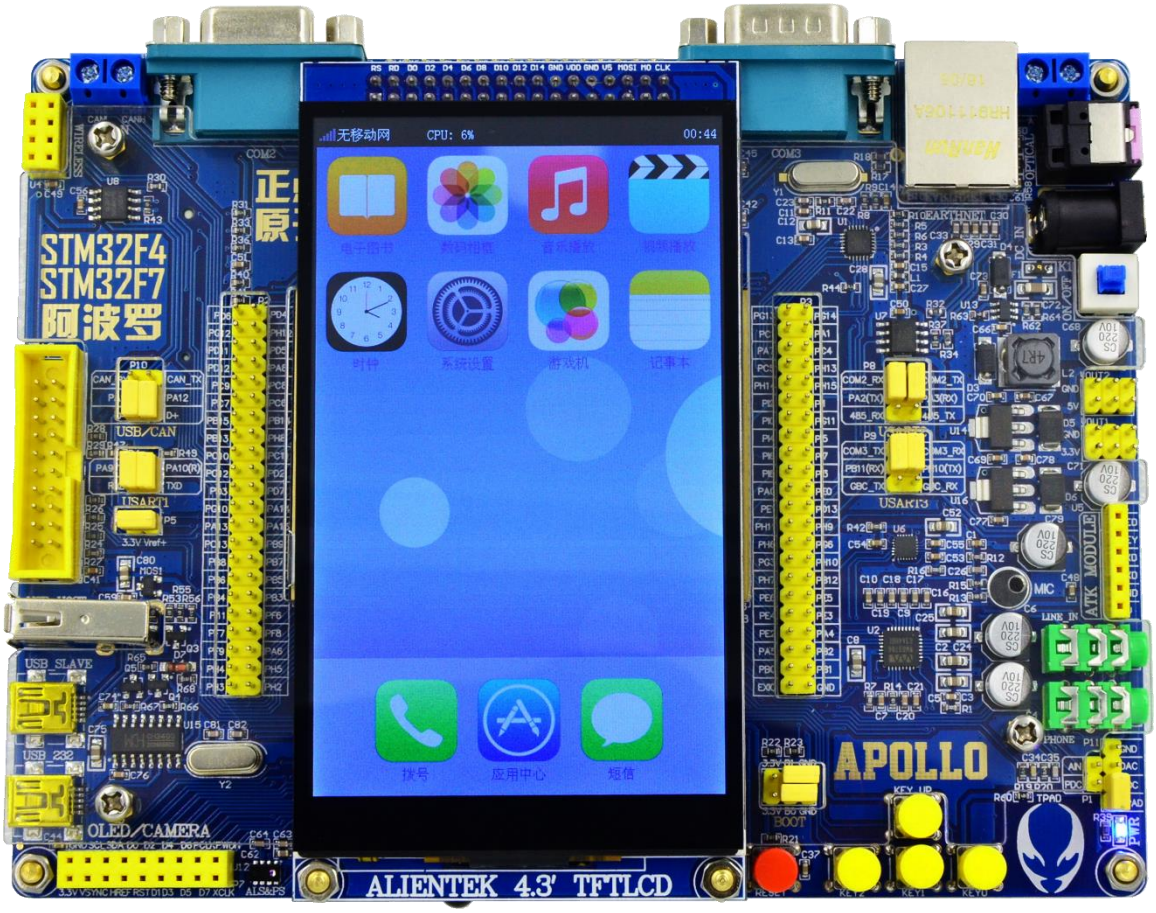
STM32F429

FreeRTOS 开发手册

V1.2

-ALIENTEK STM32F429 FreeRTOS 开发教程



广州市星翼电子科技有限公司

淘宝店铺 1：http://eboard.taobao.com

淘宝店铺 2：http://openedv.taobao.com

技术支持论坛（开源电子网）：www.openedv.com

官方网站：www.alientek.com

最新资料下载链接：http://www.openedv.com/posts/list/13912.htm

E-mail: 389063473@qq.com**QQ:** 389063473

咨询电话：020-38271790

传真号码：020-36773971

团队：正点原子团队

正点原子，做最全面、最优秀的嵌入式开发平台软硬件供应商。

友情提示

如果您想及时免费获取“正点原子”最新资料，敬请关注正点原子微信公众平台，我们将及时给您发布最新消息和重要资料。

关注方法：

(1)微信“扫一扫”，扫描右侧二维码，添加关注

(2)微信→添加朋友→公众号→输入“正点原子”→关注

(3)微信→添加朋友→输入“alientek_stm32”→关注

文档更新说明

版本	版本更新说明	作者	校审	发布日期
V1.0	初稿： 第一章 FreeRTOS 简介 第二章 FreeRTOS 移植 第三章 FreeRTOS 系统配置 第四章 FreeRTOS 中断配置和临界段 第五章 FreeRTOS 任务基础知识 第六章 FreeRTOS 任务相关 API 函数 第七章 FreeRTOS 列表和列表项 第八章 FreeRTOS 任务创建和调度器开启 第九章 FreeRTOS 任务切换 第十章 FreeRTOS 系统内核控制函数 第十一章 FreeRTOS 其他任务 API 函数 第十二章 FreeRTOS 时间管理 第十三章 FreeRTOS 队列 第十四章 FreeRTOS 信号量 第十五章 FreeRTOS 软件定时器 第十六章 FreeRTOS 事件标志组 第十七章 FreeRTOS 内存管理	左忠凯	刘军	2016.11.14
V1.1	新增 第十八章 FreeRTOS 任务通知	左忠凯	刘军	2016.11.22
V1.2	调整 V1.1 版本中的第十八章调整为本版本的第二十章。 第八章部分小节顺序做了调整。	左忠凯	刘军	2017.3.14

	新增 第八章 8.4，8.5，8.6 小节 第十八章 FreeRTOS 低功耗 Tickless 模式 第十九章 FreeRTOS 空闲任务			

目录

STM32F429 FreeRTOS 开发手册 V1.2 1

声明 14

第一章 FreeRTOS 简介 15

1.1 初识 FreeRTOS 16
1.1.1 什么是 FreeRTOS? 16
1.1.2 为什么选择 FreeRTOS？ 16
1.1.3 FreeRTOS 特点 17
1.1.4 商业许可 17
1.2 磨刀不误砍柴工 18
1.2.1 资料查找 18
1.2.2 FreeRTOS 官方文档 20
1.2.3 Cortex-M 架构资料 22
1.3 FreeRTOS 源码初探 22
1.3.1 FreeRTOS 源码下载 22
1.3.2 FreeRTOS 文件预览 24

第二章 FreeRTOS 移植 29

2.1 准备工作 30
2.1.1 准备基础工程 30
2.1.2 FreeRTOS 系统源码 30
2.2 FreeRTOS 移植 30
2.2.1 向工程中添加相应文件 30
2.2.2 修改 SYSTEM 文件 34
2.3 移植验证实验 38
2.3.1 实验程序设计 38
2.3.2 实验程序运行结果分析 41

第三章 FreeRTOS 系统配置 43

3.1 FreeRTOSConfig.h 文件 44
3.1 “INCLUDE_”开始的宏 44

3.2 “config”开始的宏 45

第四章 FreeRTOS 中断配置和临界段 53

4.1 Cortex-M 中断 54
4.1.1 中断简介 54
4.1.2 中断管理简介 54
4.1.3 优先级分组定义 55
4.1.4 优先级设置 57
4.1.5 用于中断屏蔽的特殊寄存器 58
4.2 FreeRTOS 中断配置宏 59
4.2.1 configPRIO_BITS 59
4.2.2 configLIBRARY_LOWEST_INTERRUPT_PRIORITY 59
4.2.3 configKERNEL_INTERRUPT_PRIORITY 59
4.2.4 configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY 61
4.2.5 configMAX_SYSCALL_INTERRUPT_PRIORITY 61
4.3 FreeRTOS 开关中断 62
4.4 临界段代码 62
4.4.1 任务级临界段代码保护 63
4.4.2 中断级临界段代码保护 64
4.5 FreeRTOS 中断测试实验 65
4.5.1 实验程序设计 65
4.5.2 实验程序运行结果 69

第五章 FreeRTOS 任务基础知识 71

5.1 什么是多任务系统？ 72
5.2 FreeRTOS 任务与协程 73
5.2.1 任务(Task)的特性 73
5.2.2 协程(Co-routine)的特性 74
5.3 任务状态 74
5.4 任务优先级 75
5.5 任务实现 75
5.6 任务控制块 76
5.7 任务堆栈 78

第六章 FreeRTOS 任务相关 API 函数 80

6.1 任务创建和删除 API 函数 81
6.2 任务创建和删除实验(动态方法) 83
6.2.1 实验程序设计 83
6.2.2 程序运行结果分析 87
6.3 任务创建和删除实验(静态方法) 88
6.3.1 实验程序设计 88
6.3.2 程序运行结果分析 93
6.4 任务挂起和恢复 API 函数 93
6.5 任务挂起和恢复实验 94
6.5.1 实验程序设计 94
6.5.2 程序运行结果分析 100

第七章 FreeRTOS 列表和列表项 101

7.1 什么是列表和列表项？ 102
7.1.1 列表 102
7.1.2 列表项 102
7.1.3 迷你列表项 103
7.2 列表和列表项初始化 104
7.2.1 列表初始化 104
7.2.2 列表项初始化 105
7.3 列表项插入 105
7.3.1 列表项插入函数分析 105
7.3.2 列表项插入过程图示 107
7.4 列表项末尾插入 108
7.4.1 列表项末尾插入函数分析 108
7.4.2 列表项末尾插入图示 109
7.5 列表项的删除 110
7.6 列表的遍历 111
7.7 列表项的插入和删除实验 112
7.7.1 实验程序设计 112
7.7.2 程序运行结果分析 116

第八章 FreeRTOS 调度器开启和任务相关函数详解 120

8.1 阅读本章所必备的知识 121
8.2 调度器开启过程分析 121
8.2.1 任务调度器开启函数分析 121
8.2.2 内核相关硬件初始化函数分析 123
8.2.3 使能 FPU 函数分析 124
8.2.4 启动第一个任务 124
8.2.5 SVC 中断服务函数 125
8.2.6 空闲任务 130
8.3 任务创建过程分析 130
8.3.1 任务创建函数分析 130
8.3.2 任务初始化函数分析 132
8.3.3 任务堆栈初始化函数分析 135
8.3.4 添加任务到就绪列表 137
8.4 任务删除过程分析 139
8.5 任务挂起过程分析 141
8.6 任务恢复过程分析 143

第九章 FreeRTOS 任务切换 145

9.1 PendSV 异常 146
9.2 FreeRTOS 任务切换场合 147
9.2.1 执行系统调用 147
9.2.2 系统滴答定时器(SysTick)中断 148
9.3 PendSV 中断服务函数 149
9.4 查找下一个要运行的任务 150
9.6 FreeRTOS 时间片调度 153
9.6 时间片调度实验 155
9.6.1 实验程序设计 155
9.6.2 程序运行结果分析 158

第十章 FreeRTOS 系统内核控制函数 159

10.1 内核控制函数预览 160
10.2 内核控制函数详解 160

第十一章 FreeRTOS 其他任务 API 函数 165

11.1 任务相关 API 函数预览 166
11.2 任务相关 API 函数详解 167
11.3 任务壮态查询 API 函数实验 174
11.3.1 实验程序设计 174
11.3.2 程序运行结果分析 179
11.4 任务运行时间信息统计实验 180
11.4.1 相关宏的设置 180
11.4.2 实验程序设计 182
11.4.3 程序运行结果分析 186

第十二章 FreeRTOS 时间管理 187

12.1 FreeRTOS 延时函数 188
12.1 函数 vTaskDelay() 188
12.2 函数 prvAddCurrentTaskToDelayedList() 189
12.3 函数 vTaskDelayUntil() 191
12.2 FreeRTOS 系统时钟节拍 195

第十三章 FreeRTOS 队列 201

13.1 队列简介 202
13.2 队列结构体 204
13.3 队列创建 205
13.3.1 函数原型 205
13.3.2 队列创建函数详解 207
13.3.3 队列初始化函数 208
13.3.4 队列复位函数 209
13.4 向队列发送消息 211
13.4.1 函数原型 211
13.4.2 任务级通用入队函数 215
13.4.3 中断级通用入队函数 219
13.5 队列上锁和解锁 221
13.6 从队列读取消息 223
13.7 队列操作实验 227
13.7.1 实验程序设计 227
13.7.2 程序运行结果分析 235

第十四章 FreeRTOS 信号量 237

14.1 信号量简介 238
14.2 二值信号量 238
14.2.1 二值信号量简介 238
14.2.2 创建二值信号量 240
14.2.3 二值信号量创建过程分析 241
14.2.4 释放信号量 242
14.2.5 获取信号量 243
14.3 二值信号量操作实验 245
14.3.1 实验程序设计 245
14.3.2 程序运行结果分析 251
14.4 计数型信号量 252
14.4.1 计数型信号量简介 252
14.4.2 创建计数型信号量 252
14.4.3 计数型信号量创建过程分析 253
14.4.4 释放和获取计数信号量 254
14.5 计数型信号量操作实验 254
14.5.1 实验程序设计 254
14.5.2 程序运行结果分析 258
14.6 优先级翻转 259
14.7 优先级翻转实验 260
14.7.1 实验程序设计 260
14.7.2 程序运行结果分析 265
14.8 互斥信号量 267
14.8.1 互斥信号量简介 267
14.8.2 创建互斥信号量 267
14.8.3 互斥信号量创建过程分析 268
14.8.4 释放互斥信号量 270
14.8.5 获取互斥信号量 273
14.9 互斥信号量操作实验 278
14.9.1 实验程序设计 278
14.9.2 程序运行结果分析 281
14.10 递归互斥信号量 282

14.10.1 递归互斥信号量简介 282
14.10.2 创建互斥信号量 282
14.10.3 递归信号量创建过程分析 283
14.10.4 释放递归互斥信号量 283
14.10.5 获取递归互斥信号量 284
14.10.6 递归互斥信号量使用示例 286

第十五章 FreeRTOS 软件定时器 288

15.1 软件定时器简介 289
15.2 定时器服务/Daemon 任务 289
15.2.1 定时器服务任务与队列 289
15.2.2 定时器相关配置 289
15.3 单次定时器和周期定时器 290
15.4 复位软件定时器 290
15.5 创建软件定时器 292
15.6 开启软件定时器 294
15.7 停止软件定时器 295
15.8 软件定时器实验 296
15.8.1 实验程序设计 296
15.8.2 程序运行结果分析 300

第十六章 FreeRTOS 事件标志组 301

16.1 事件标志组简介 302
16.2 创建事件标志组 303
16.3 设置事件位 303
16.4 获取事件标志组值 305
16.5 等待指定的事件位 306
16.6 事件标志组实验 307
16.6.1 实验程序设计 307
16.6.2 程序运行结果分析 312

第十七章 FreeRTOS 任务通知 315

17.1 任务通知简介 316
17.2 发送任务通知 316

17.3 任务通知通用发送函数 319
17.3.1 任务级任务通知通用发送函数 319
17.3.2 中断级任务通知发送函数 322
17.4 获取任务通知 325
17.5 任务通知模拟二值信号量实验 329
17.5.1 实验程序设计 329
17.5.2 实验程序运行结果 332
17.6 任务通知模拟计数型信号量实验 332
17.6.1 实验程序设计 333
17.6.2 实验程序运行结果 334
17.7 任务通知模拟消息邮箱实验 334
17.7.1 实验程序设计 334
17.7.2 实验程序运行结果 338
17.8 任务通知模拟事件标志组实验 338
17.8.1 实验程序设计 339
17.8.2 实验程序运行结果 342

第十八章 FreeRTOS 低功耗 Tickless 模式 343

18.1 STM32F4 低功耗模式 344
18.1.1 睡眠(Sleep)模式 344
18.1.2 停止(Stop)模式 345
18.1.3 待机(Standby)模式 345
18.2 Tickless 模式详解 346
18.2.1 如何降低功耗？ 346
18.2.2 Tickless 具体实现 346
18.3 低功耗 Tickless 模式实验 352
18.3.1 实验程序设计 352
18.3.2 实验程序运行结果 357

第十九章 FreeRTOS 空闲任务 359

19.1 空闲任务详解 360
19.1.1 空闲任务简介 360
19.1.2 空闲任务的创建 360
19.1.3 空闲任务函数 361

19.2 空闲任务钩子函数详解 364
19.2.1 钩子函数 364
19.2.2 空闲任务钩子函数 365
19.3 空闲任务钩子函数实验 366
19.3.1 实验程序设计 366
19.3.2 实验程序运行结果 368

第二十章 FreeRTOS 内存管理 369

20.1 FreeRTOS 内存管理简介 370
20.2 内存碎片 370
20.3 heap_1 内存分配方法 371
20.3.1 分配方法简介 371
20.3.2 内存申请函数详解 371
20.3.3 内存释放函数详解 374
20.4 heap_2 内存分配方法 374
20.4.1 分配方法简介 374
20.4.2 内存块详解 375
20.4.3 内存堆初始化函数详解 376
20.4.4 内存块插入函数详解 377
20.4.5 内存申请函数详解 378
20.4.6 内存释放函数详解 380
20.5 heap_3 内存分配方法 381
20.6 heap_4 内存分配方法 383
20.6.1 分配方法简介 383
20.6.2 内存堆初始化函数详解 383
20.6.3 内存块插入函数详解 385
20.6.4 内存申请函数详解 388
20.6.5 内存释放函数详解 392
20.7 heap_5 内存分配方法 393
20.8 内存管理实验 394
20.8.1 实验程序设计 394
20.8.2 实验程序运行结果 398

声明

本教程中关于 FreeRTOS 的原理性知识均参考自 FreeRTOS 官方手册：《FreeRTOS Reference Manual》和《Using the FreeRTOS Real Time Kernel - A Practical Guide》，其中大多数的理论知识都是翻译自这两本手册，有关 Cortex-M 的内核知识参考自《The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors, 3rd Edition》这本教程。本教程作为 FreeRTOS 学习、参考资料全部免费公开，包括教程中的实验源码！

第一章　FreeRTOS 简介

从本章开始我们就踏入了 FreeRTOS 的大门，FreeRTOS 是一个 RTOS 类的嵌入式实时操作系统。在此之前 ALIENTEK 已经推出了 uC/OS 操作系统的教程和例程，但是现在为什么又要学 FreeROTS 呢？这就是本章的目的，本章分为如下几部分：

1.1 初始 FreeRTOS

1.2 磨刀不误砍柴工

1.3 FreeRTOS 源码初衷

1.1 初识 FreeRTOS

1.1.1 什么是 FreeRTOS?

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 uC/OS，FreeRTOS，RTX，RT-Thread 等这些都是 RTOS 类操作系统。

操作系统允许多个任务同时运行，这个叫做多任务，实际上，一个处理器核心在某一时刻只能运行一个任务。操作系统中任务调度器的责任就是决定在某一时刻究竟运行哪个任务，任务调度在各个任务之间的切换非常快!这就给人们造成了同一时刻有多个任务同时运行的错觉。

操作系统的分类方式可以由任务调度器的工作方式决定，比如有的操作系统给每个任务分配同样的运行时间，时间到了就轮到下一个任务，Unix 操作系统就是这样的。 RTOS 的任务调度器被设计为可预测的，而这正是嵌入式实时操作系统所需要的，实时环境中要求操作系统必须对某一个事件做出实时的响应，因此系统任务调度器的行为必须是可预测的。像 FreeRTOS 这种传统的 RTOS 类操作系统是由用户给每个任务分配一个任务优先级，任务调度器就可以根据此优先级来决定下一刻应该运行哪个任务。

FreeRTOS 是 RTOS 系统的一种，FreeRTOS 十分的小巧，可以在资源有限的微控制器中运行，当然了，FreeRTOS 不仅局限于在微控制器中使用。但从文件数量上来看 FreeRTOS 要比 uC/OSII 和 uC/OSIII 小的多。

1.1.2 为什么选择 FreeRTOS?

上面我们说了 RTOS 类系统有很多，为什么要选择 FreeRTOS 呢？在 uC/OS 教程中，我们说过学习 RTOS 首选 uC/OS，因为 uC/OS 的资料很多，尤其是中文资料！但是 FreeRTOS 的资料少，而且大多数是英文的，我为何要选择它？原因如下：

1、FreeRTOS 免费！这是最重要的一点，uC/OS 是要收费的，学习 RTOS 系统的话 uC/OS 是首选，但是做产品的话就要考虑一下成本了。显而易见的，FreeRTOS 在此时就是一个很好的选择，当然了也可以选择其他的免费的 RTOS 系统。

2、许多其他半导体厂商产品的 SDK 包就使用 FreeRTOS 作为其操作系统，尤其是 WIFI、蓝牙这些带协议栈的芯片或模块。

3、许多软件厂商也使用 FreeRTOS 做本公司软件的操作系统，比如著名的 TouchGFX，其所有的例程都是基于 FreeRTOS 操作系统的。ST 公司的所有要使用到 RTOS 系统的例程也均采用了 FreeRTOS，由此可见免费的力量啊！

3、简单，FreeRTOS 的文件数量很少，这个在我们后面的具体学习中就会看到，和 uC/OS 系统相比要少很多！

4、文档相对齐全，在 FreeRTOS 的官网（www.freertos.org）上可以找到所需的文档和源码，但是所有的文档都是英文版本的，而且下载 pdf 文档的时候是要收费的。

5、FreeRTOS 被移植到了很多不同的微处理器上，比如我们使用的 STM32，F1、F3、F4 和最新的 F7 都有移植，这个极大的方便了我们学习和使用。

6、社会占有量很高，EEtimes 统计的 2015 年 RTOS 系统占有量中 FreeRTOS 已经跃升至第一位，如图 1.1.1 所示。

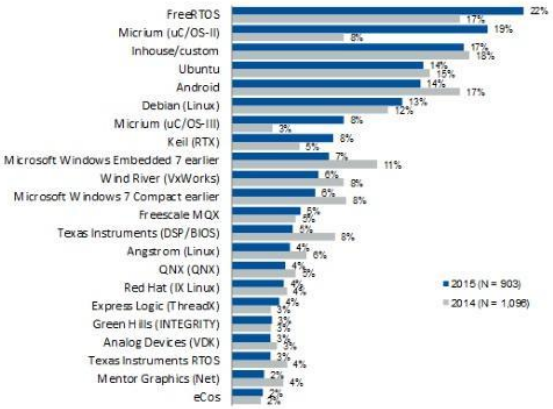


图 1.1.2.1 2014/2015 年 ROTS 系统使用情况

1.1.3 FreeRTOS 特点

FreeRTOS 是一个可裁剪的小型 RTOS 系统，其特点包括：

- FreeRTOS 的内核支持抢占式，合作式和时间片调度。
- SafeRTOS 衍生自 FreeRTOS，SafeRTOS 在代码完整性上相比 FreeRTOS 更胜一筹。
- 提供了一个用于低功耗的 Tickless 模式。
- 系统的组件在创建时可以选择动态或者静态的 RAM，比如任务、消息队列、信号量、软件定时器等等。
- 已经在超过 30 种架构的芯片上进行了移植。
- FreeRTOS-MPU 支持 Corex-M 系列中的 MPU 单元，如 STM32F429。
- FreeRTOS 系统简单、小巧、易用，通常情况下内核占用 4k-9k 字节的空间。
- 高可移植性，代码主要 C 语言编写。
- 支持实时任务和协程(co-routines 也有称为合作式、协同程序，本教程均成为协程)。
- 任务与任务、任务与中断之间可以使用任务通知、消息队列、二值信号量、数值型信号量、递归互斥信号量和互斥信号量进行通信和同步。
- 创新的事件组(或者事件标志)。
- 具有优先级继承特性的互斥信号量。
- 高效的软件定时器。
- 强大的跟踪执行功能。
- 堆栈溢出检测功能。
- 任务数量不限。
- 任务优先级不限。

1.1.4 商业许可

FreeRTOS 衍生出来了另外两个系统：OpenRTOS 和 SafeTROS，FreeRTOS 开源许可协议允许在商业应用中使用 FreeRTOS 系统，并且不需要公开你的私有代码。如果有以下需求的话可以使用 OpenRTOS：

1、你不能接受 FreeRTOS 的开源许可协议条件，具体参见表 1.4.1。

2、你需要技术支持。

3、你想获得开发帮助

4、你需要法律保护或者其他的保护。

使用 OpenRTOS 的话需要遵守商业协议，FreeRTOS 的开源许可和 OpenRTOS 的商业许可区别如表 1.4.1 所示：

	FreeRTOS 开源许可	OpenRTOS 商业许可
是否免费	是	否
是否可以用在商业应用中	是	是
是否免版权费	是	是
是否提供质保	否	是
是否提供专业的技术支持	否(仅提供论坛技术支持)	是
是否提供法律保护	否	是
是否需要开源自己的代码	否	否
自行修改源码以后是否需要开源	是	否
是否需要记录使用了系统	是，如果要发布自己的代码	否
是否需要给我工程用户提供FreeRTOS代码	是，如果要发布自己的代码	否

表 1.4.1 开源许可和商业许可区别

OpenRTOS 是 FreeRTOS 的商业化版本，OpenRTOS 的商业许可协议不包含任何 GPL 条款。还有另外一个系统：SafeRTOS，SafeRTOS 看名字有个 Safe，安全的意思！SafeRTOS 也是 FreeRTOS 的衍生版本，只是 SafeRTOS 过了一些安全认证，比如 IEC61508。

1.2 磨刀不误砍柴工

1.2.1 资料查找

不管学习什么，第一件事就是找资料，可能有的朋友会说“找资料还不容易吗？”，打开谷歌或者百度直接搜索不就行了。方法是没错，但是你会发现搜索出来的资料很凌乱，尤其是国内大部分开发者不喜欢看英文，都想找中文资料。而 FreeRTOS 的中文资料大多数都是老版本的，要知道 FreeRTOS 的更新是很快的，在笔者写教程的时候最新版本的 FreeRTOS 是 V9.0.0 的，而且大多数资料都是零零散散，断断续续的。当然了，也有很多很不错的资料，比如有的博主写的博客，有些技术论坛上别人分享的帖子等，但是从这么多资料中找出这些精华还是很费时间的。其实找资料没有这么复杂，官网是最好的地方，FreeRTOS 的官网是 www.freertos.org，打开以后如图 1.2.1.1 所示。

图 1.2.1.1 FreeRTOS 官网

显而易见的，FreeRTOS 的官网是全英文的，由此可见英语对于一个电子工程师的重要性，不管你喜不喜欢，英文就在这里！所以在这里建议大家不要谈英色变，像我这种四级考了六次都没过的人此时也只能静下心来慢慢啃了。

点击网页导航栏中的“Quick Start”，直译就是“快速开始”，点击以后打开 Quick Start 页面，此页面就是简单的介绍如何快速的上手 FreeRTOS，这个大家还是要看一下的，对 FreeRTOS 可以有一个简单的了解。导航栏中的“Suppotred MCUs”可以查看 FreeRTOS 所支持的 MCU，打开以后如图 1.2.1.2 所示。

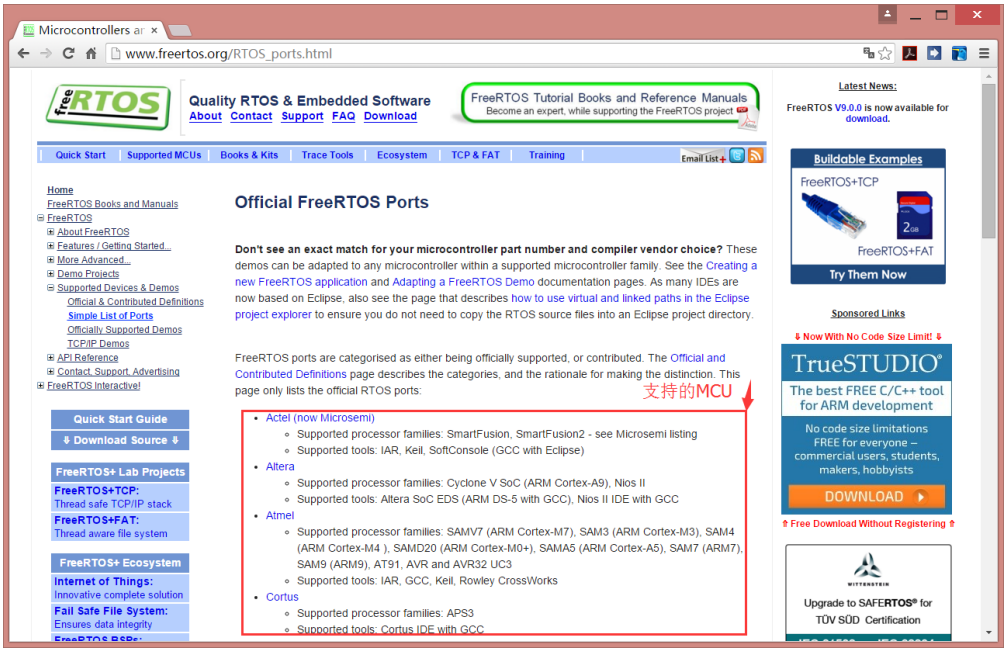


图 1.2.1.2 FreeRTOS 所支持的 MCU

从这里就可以找到 FreeRTOS 是否支持我们所使用的 MCU，支持的意思就是官方已经移植好了，用户直接拿过来针对自己的板子简单修改一下就可以在自己的板子上跑起来，后面我们

讲解的 FreeRTOS 在 ALIENTEK STM32F429 板子上的移植就是用的这种方法。

1.2.2 FreeRTOS 官方文档

如果有学习过 uC/OS 的朋友会知道，uC/OS 官方的文档和书籍做非常好，我们去 uC/OS 官网看一下就知道，如图 1.2.2.1 所示。

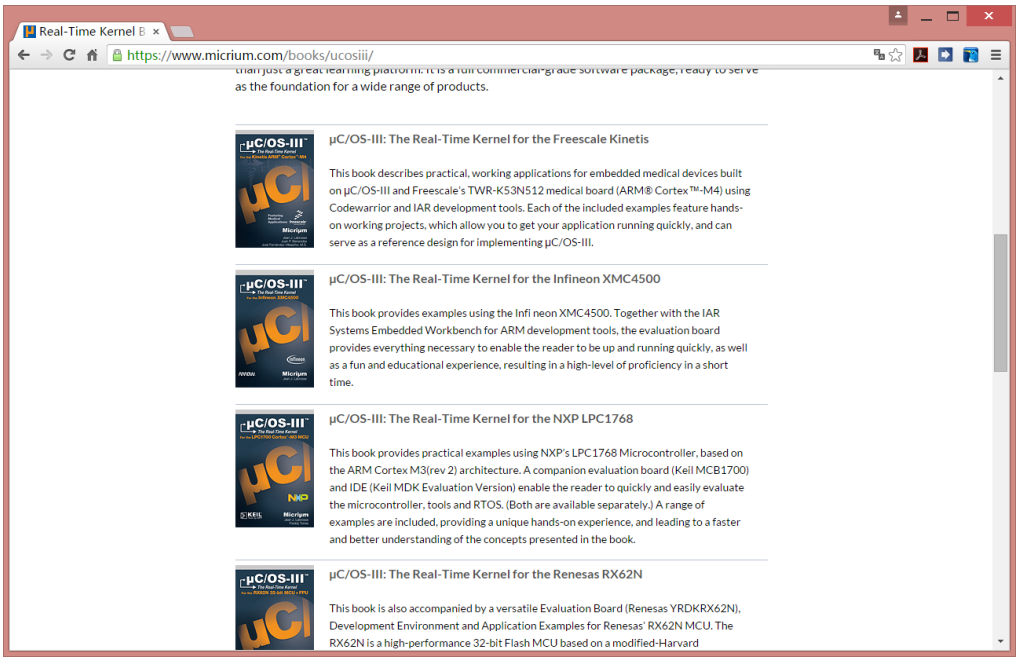


图 1.2.2.1 Micrium 官方文档和书籍

图 1.2.2.1 中只是 uC/OS 官方一部分的文档和书籍，详细的情况可以到 uC/OS 官网 www.micrium.com 上查看，可以看出 Micrium 官方的文档和书籍非常多，Micrium 官方所有的产品都有书籍，不仅仅是 uC/OS，像 uC/TCP、uC/USB 这些组件也都有相应的文档和书籍。并且 uC/OS 的本地化做得也很好，这些书籍大多数都有中文翻译版本出售，具体大家可以到当当上查找一下。

那么问题来了，FreeRTOS 的官方文档和教程怎么样？点击导航栏中的“PDF Books”，打开以后如图 1.2.2.2 所示：

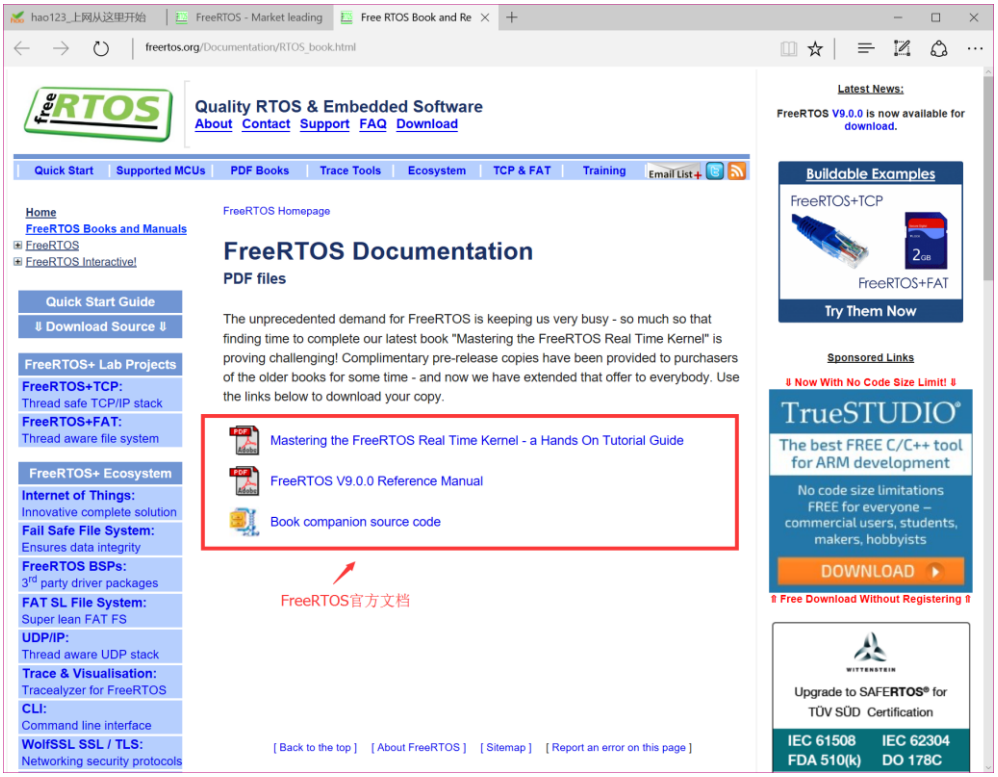


图 1.2.2.2 FreeRTOS 官方文档和教程

从图 1.2.2.2 可以看到，FreeRTOS 官方有两份 PDF 文档，一份是 FreeRTOS 的指导手册，一份是 FreeRTOS 的 API 函数参考手册。相比 uC/OS,FreeRTOS 的官方文档确实有点少。FreeRTOS 还有一个在线文档，可以直接在官网浏览，点击“Quick Start”，打开以后如图 1.2.2.3 所示

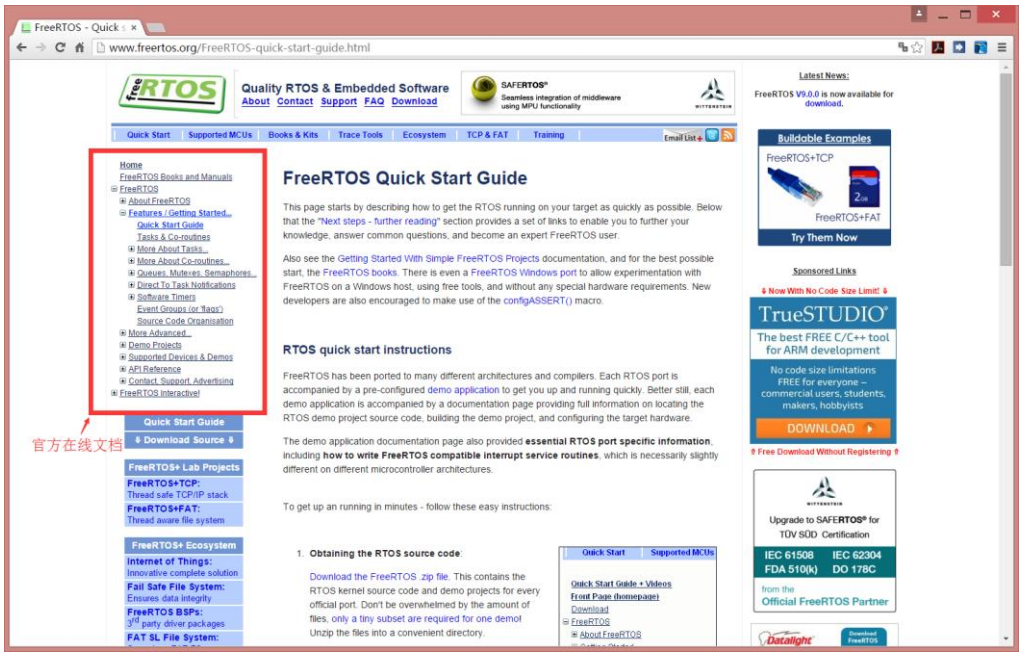


图 1.2.2.3

图 1.2.2.3 红框中的部分就是 FreeRTOS 的在线文档，我们可以直接阅读在线文档来学习 FreeRTOS，但是由于 FreeRTOS 是国外网站，所以在线阅读文档的话加载页面会很慢，这点大

家要做好心理准备。

1.2.3 Cortex-M 架构资料

在后面学习 FreeRTOS 任务切换的时候需要我们了解 Cortex-M 内核架构相关的知识，否则的话根本看不懂任务切换的过程。关于 Cortex-M 架构的讲解在 ARM 官网上就有，不过是英文的，叫做《The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors, 3rd Edition》。不过幸运的是这本教程已经有中文翻译版了，叫做《ARM Cortex-M3 与 Cortex-M4 权威指南(第三版)》，Joseph Yiu 著，吴常玉、曹孟娟、王丽红译,清华大学出版社。这本教程在淘宝或者当当都有出售，此书对于 Cortex-M 架构的讲解非常详细，强烈建议那些想深入了解 Cortex-M 架构的同学看一下，如果英语 NB 的可以直接看英文原版的，ARM 官网直接下载，完全免费的。此书中文翻译版如图 1.2.3.1 所示。

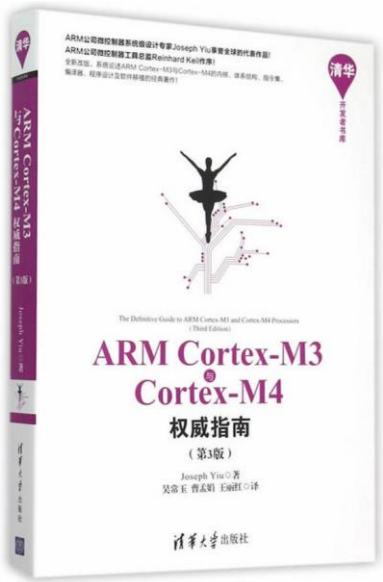


图 1.2.3.1 权威指南中文版

后面的学习中涉及到 Cortex-M 架构的知识均参考自这本教程，后面简称这本教程为《权威指南》。注意和宋岩翻译的那本《ARM Cortex-M3 权威指南》的区别，宋岩翻译的那本虽然也是 Joseph Yiu 编著的，但是只是针对 Cortex-M3 的，M4 添加的新功能没有讲解，尤其是 FPU 部分。

1.3 FreeRTOS 源码初探

1.3.1 FreeRTOS 源码下载

了解过 FreeRTOS 的一些基础知识后接下来就是要见识一下 FreeRTOS 的庐山真面目，是骡子是马拉出来溜溜嘛。FreeRTOS 真身在哪里呢？很明显，官网下载嘛！进入 FreeRTOS 首页，有两个下载提示点，一个是左上角部分的 FreeRTOS 下载提示，一个是右上角部分的下载提示，如图 1.3.1.1 所示：

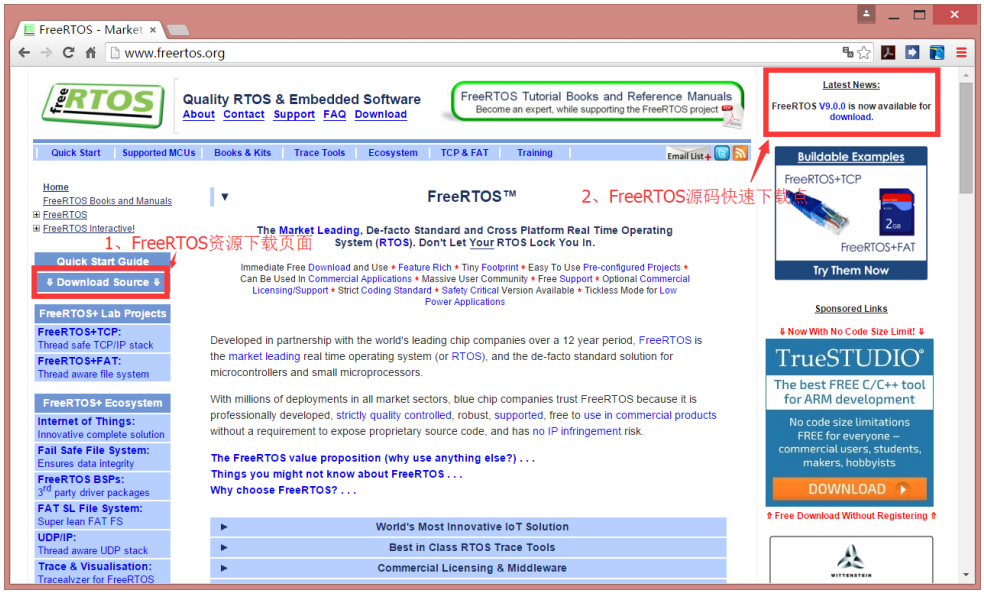


图 1.3.1.1 FreeRTOS 下载

这两个下载点都可以进入 FreeRTOS 源码下载页面，下载页面如图 1.3.1.2 所示：

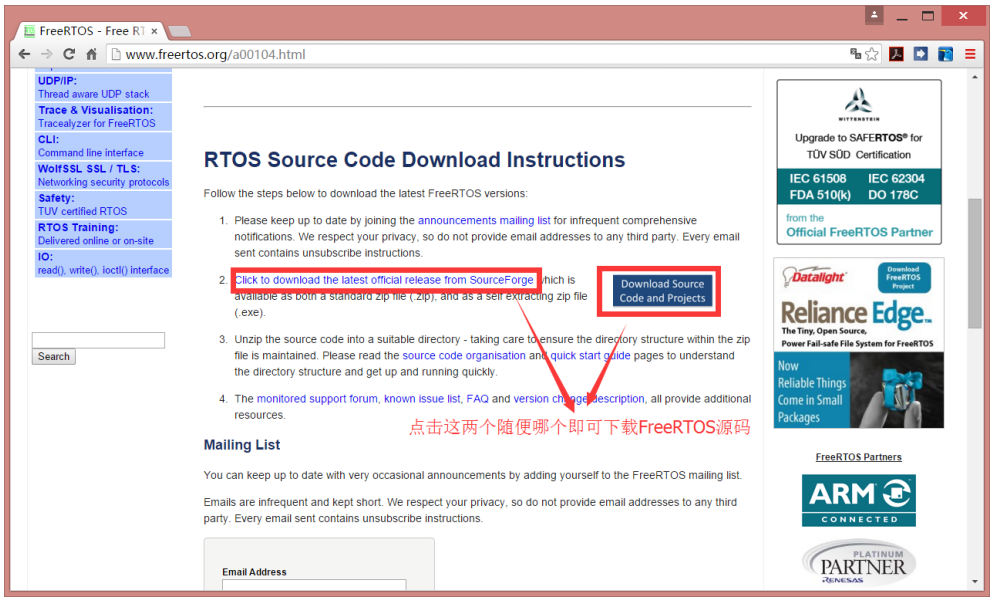


图 1.3.1.2 FreeRTOS 下载

点击图 1.3.1.2 中的下载链接以后会提示我们下载“FreeRTOSv9.0.0.exe”这个东东，“.exe”结尾的？这不是个软件吗？跟源码有什么关系？其实这个软件会自动下载 FreeRTOS 的源码的，把这个软件下载下来以后双击打开如图 1.3.1.3 所示：

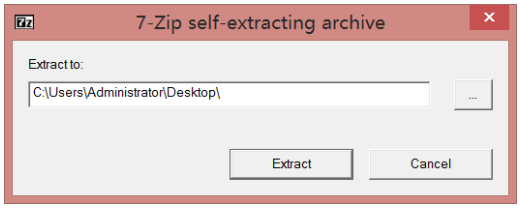


图 1.3.1.3 FreeRTOS 源码存放位置

图 1.3.1.3 是用来设置 FreeRTOS 源码存放位置的，自行选择一个位置即可，设置好以后点

击“Extract”即可开始自动下载 FreeRTOS 源码，下载过程如图 1.3.1.4 所示：

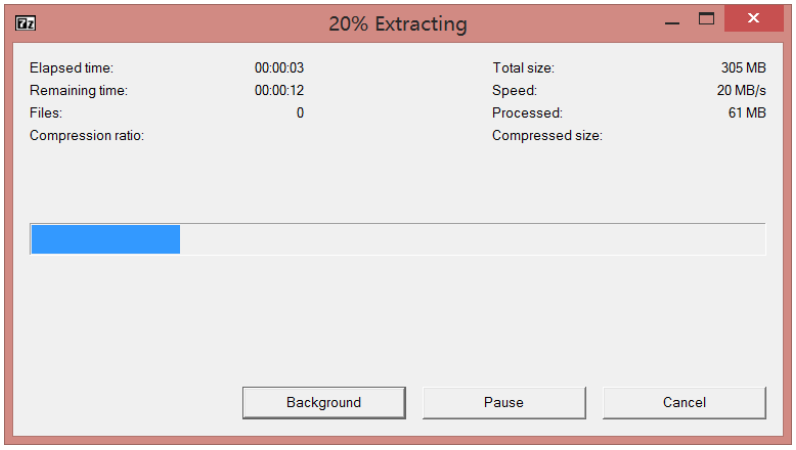


图 1.3.1.4 FreeRTOS 下载过程

静静的等待 FreeRTOS 源码下载完成，下载完成以后图 1.3.1.4 会自动关闭，这个时候查看你在图 1.3.1.3 中设置的路径，里面会多个 FreeRTOSv9.0.0 文件夹。FreeRTOS 源码就在这个文件夹里面，打开如图 1.3.1.5 所示：

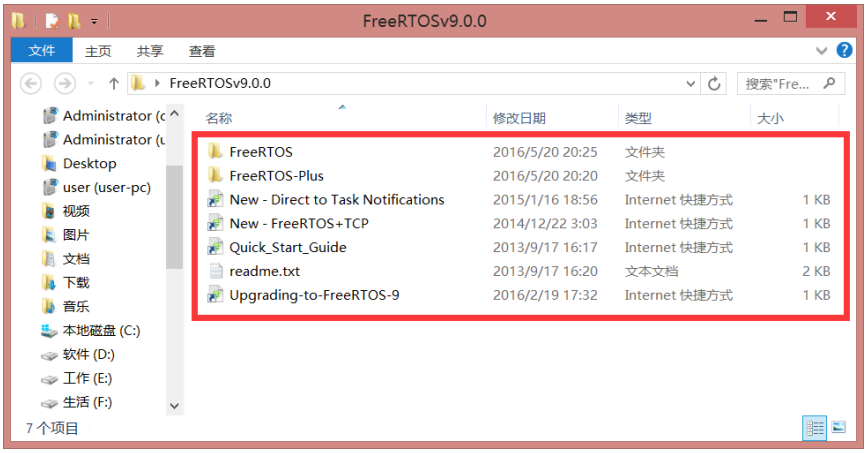


图 1.3.1.5 FreeRTOS 源码文件

至此我们就得到了 FreeRTOS 源码。

1.3.2 FreeRTOS 文件预览

在 1.3.1 小节中我们已经获取到了 FreeRTOS 的源码了，这一小节我们就来初步的预览一下这个源码文件，大致看一下都是些什么东西。从图 1.3.1.5 可以看出 FreeRTOS 源码中有两个文件夹，4 个 HTML 格式的网页和一个 txt 文档，HTML 网页和 txt 文档就不用介绍了，看名字就知道是什么东西了，重点在于上面那两个文件夹:FreeRTOS 和 FreeRTOS-Plus，这两个文件夹里面的东西就是 FreeRTOS 的源码。我们知道苹果从 Iphone6 以后分为了 Iphone6 和 Iphone6 Plus 两个版本，区别就是 Plus 比普通的功能多一点，配置强大一点。现在 FreeRTOS 也这么分，是不是 Plus 版本比 FreeRTOS 功能强一点啊，强大到哪里？是不是源码都不同了呀？

1、FreeRTOS 文件夹

打开 FreeRTOS 文件夹，如图 1.3.2.1 所示：

图 1.3.2.1 FreeRTOS 文件夹

图 1.3.2.1 中有三个文件夹，Demo、License 和 Source，从名字上就可以很容易的得出他们都是些什么。

- **Demo 文件夹**

Demo 文件夹里面就是 FreeRTOS 的相关例程，打开以后如图 1.3.2.2 所示：

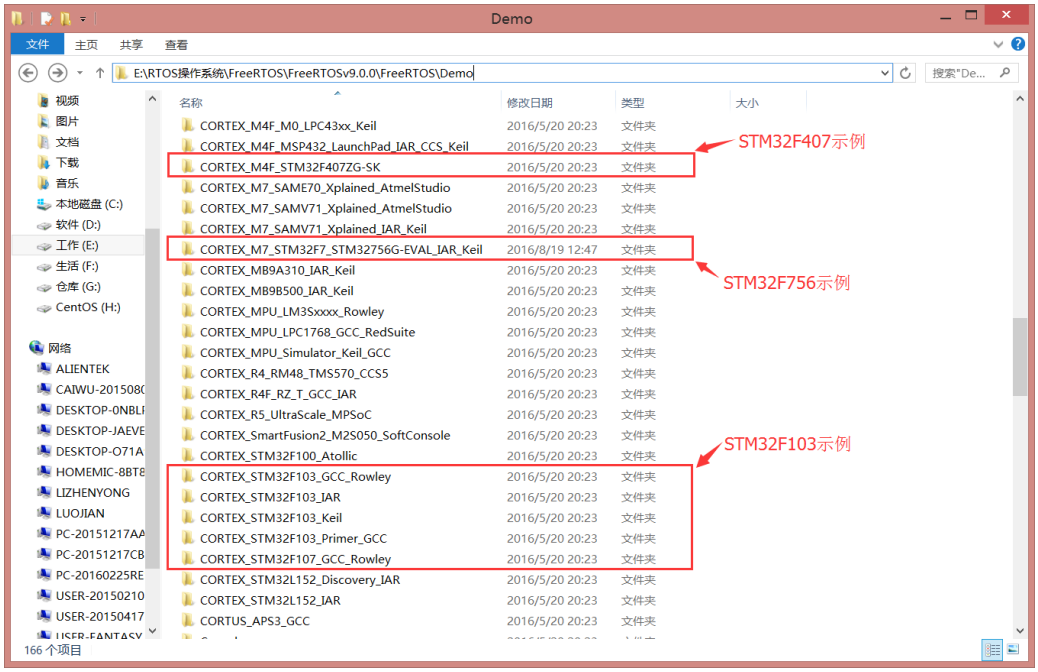


图 1.3.2.2 Demo 文件夹

可以看出 FreeRTOS 针对不同的 MCU 提供了非常多的 Demo，其中就有 ST 的 F1、F4 和 F7 的相关例程，这对于我们学习来说是非常友好的，我们在移植的时候就会参考这些例程，虽然我们用的 STM32F429 的芯片，但是是可以用 F407 的工程的，因为他们都是 Cortem-M4F 的内核。

- **License 文件夹**

这个文件夹里面就是相关的许可信息，要用 FreeRTOS 做产品的得仔细看看，尤其是要出口的产品。

- **Source 文件夹**

看名字就知道了，这个就是 FreeRTOS 的本尊了，打开后如图 1.3.2.3 所示：

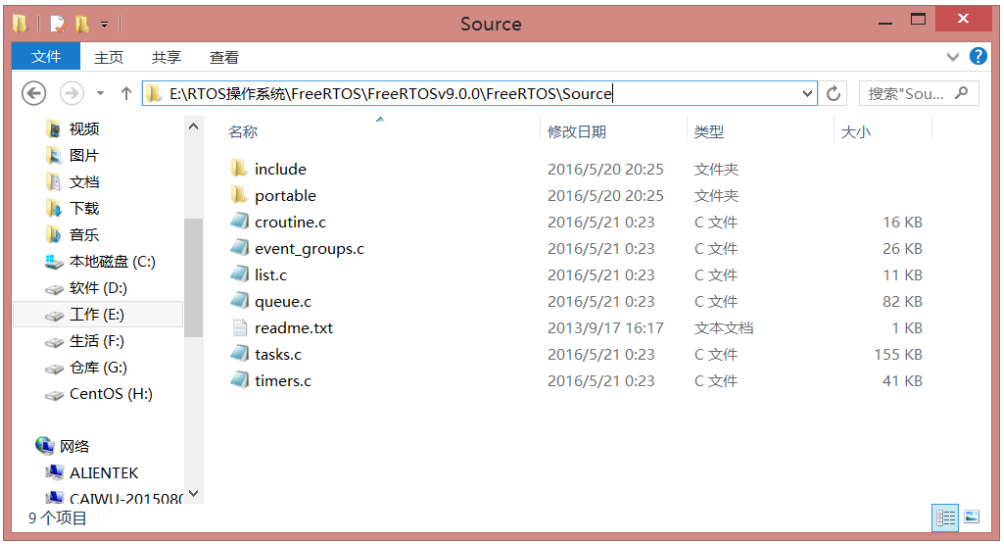


图 1.3.2.3 FreeRTOS 源码

图 1.3.2.3 就是 FreeRTOS 的源码文件，也是我们以后打交道的，可以看出，相比于 uC/OS 来说 FreeRTOS 的文件非常少！include 文件夹是一些头文件，移植的时候是需要的，下面的这些.C 文件就是 FreeRTOS 的源码文件了，移植的时候肯定也是需要的。重点来看一下 portable 这个文件夹，我们知道 FreeRTOS 是个系统，归根结底就是个纯软件的东西，它是怎么和硬件联系在一起的呢？软件到硬件中间必须有一个桥梁，portable 文件夹里面的东西就是 FreeRTOS 系统和具体的硬件之间的连接桥梁！不同的编译环境，不同的 MCU，其桥梁应该是不同的，打开 portable 文件夹，如图 1.3.2.4 所示：

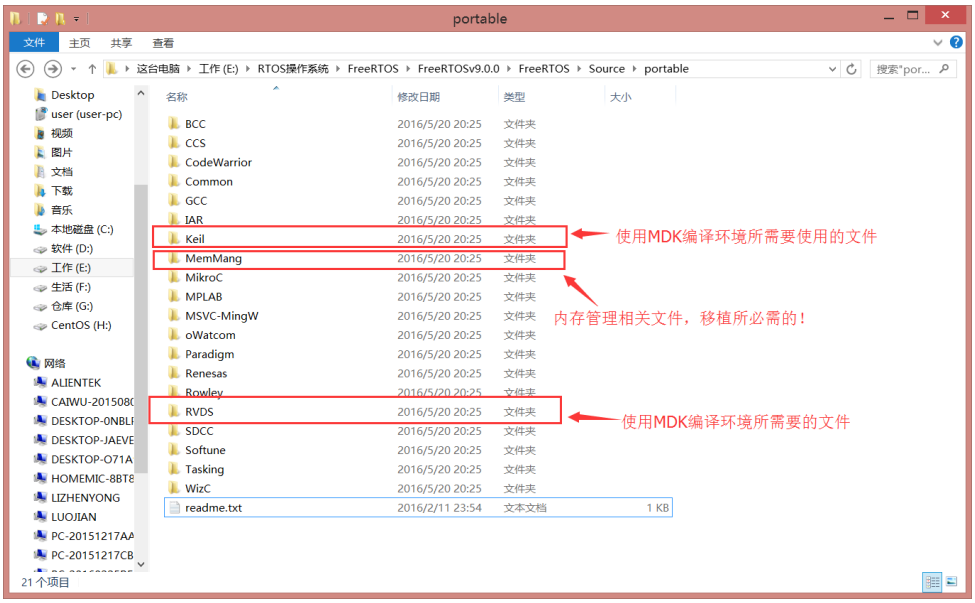


图 1.3.2.4 portable 文件夹

从图 1.3.2.4 中可以看出 FreeRTOS 针对不同的编译环境和 MCU 都有不同的“桥梁”，我们这里就以 MDK 编译环境下的 STM32F429 为例。MemMang 这个文件夹是跟内存管理相关的，我们移植的时候是必须的，具体内容我们后面会专门有一章来讲解。Keil 文件夹里面的东西肯定也是必须的，但是我们打开 Keil 文件夹以后里面只有一个文件：See-also-the-RVDS-directory.txt。这个 txt 文件是什么鬼？别急嘛！看文件名字“See-also-the-RVDS-directory”，意思就是参考 RVDS 文件夹里面的东西！哎，好吧，在打开 RVDS 文件夹，如图 1.3.2.5 所示：

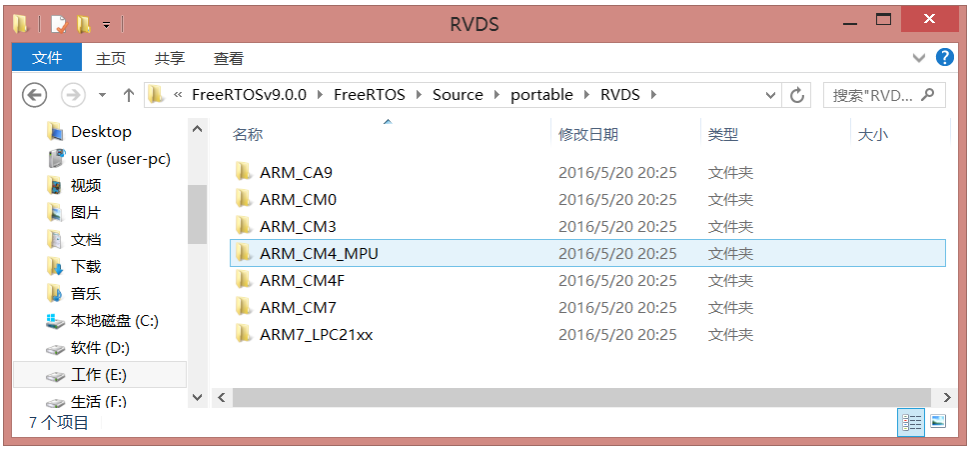


图 1.3.2.5 RVDS 文件夹

RVDS 文件夹针对不同的架构的 MCU 做了详细的分类，STM32F429 就参考 ARM_CM4F，打开 ARM_CM4F 文件夹，如图 1.3.2.6 所示：

图 1.3.2.6 ARM_CM4F 文件夹

ARM_CM4F 有两个文件，这两个文件就是我们移植的时候所需要的！

2、FreeRTOS-Plus 文件夹

上面我们分析完了 FreeRTOS 文件夹，接下来看一下 FreeRTOS-Plus，打开以后如图 1.3.2.7 所示：

图 1.3.2.7 FreeRTOS-Plus 文件夹

同样，FreeRTOS-Plus 也有 Demo 和 Source，Demo 就不看了，肯定是一些例程。我们看一下 Source，打开以后如图 1.3.2.8 所示：

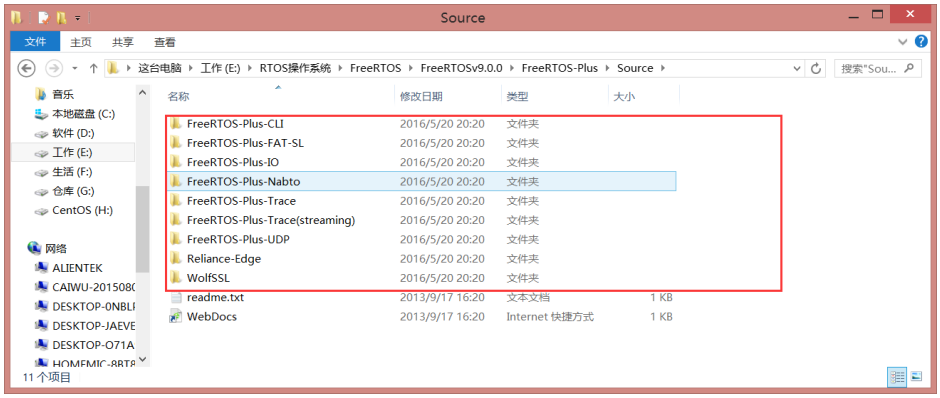


图 1.2.3.8 Source 文件夹

可以看出，FreeRTOS-Plus 中的源码其实并不是 FreeRTOS 系统的源码，是在这个 FreeRTOS 系统上另外增加的一些功能代码，比如 CLI、FAT、Trace 等等。就系统本身而言，和 FreeRTOS 里面的一模一样的，所以我们如果只是学习 FreeRTOS 这个系统的话，FreeRTOS-Plus 就没必要看了。

第二章 FreeRTOS 移植

上一章中我们初步的了解了一下 FreeRTOS，本章就正式踏上 FreeRTOS 的学习之路，首先肯定是把 FreeRTOS 移植到我们所使用的平台上，这里以 ALIENTEK 的 STM32F429 开发板为例，本章分为如下几部分：

2.1 准备工作

2.2 FreeRTOS 移植

2.3 移植验证实验

2.1 准备工作

2.1.1 准备基础工程

要移植 FreeRTOS，肯定需要一个基础工程，基础工程越简单越好，这里我们就用基础例程中的跑马灯实验来作为基础工程。

2.1.2 FreeRTOS 系统源码

FreeRTOS 系统源码在上一章已经详细的讲解过如何获取了，这里我们会将 FreeRTOS 的系统源码放到开发板光盘中去，路径为：6，软件资料->14，FreeRTOS 学习资料->FreeRTOS 源码。

2.2 FreeRTOS 移植

2.2.1 向工程中添加相应文件

1、添加 FreeRTOS 源码

在基础工程中新建一个名为 FreeRTOS 的文件夹，如图 2.2.1.1 所示：

图 2.2.1.1 新建 FreeRTOS 文件夹

创建 FreeRTOS 文件夹以后就可以将 FreeRTOS 的源码添加到这个文件夹中，添加完以后如图 2.2.1.2 所示：

图 2.2.1.2 添加 FreeRTOS 源码

在 1.3.2 小节中详细的讲解过 portable 文件夹，我们只需要留下 keil、MemMang 和 RVDS 这三个文件夹，其他的都可以删除掉，完成以后如图 2.2.1.3 所示：

图 2.2.1.3 portable 文件夹

2、向工程分组中添加文件

打开基础工程，新建分组 FreeRTOS_CORE 和 FreeRTOS_PORTABLE，然后向这两个分组中添加文件，如图 2.2.1.4 所示：

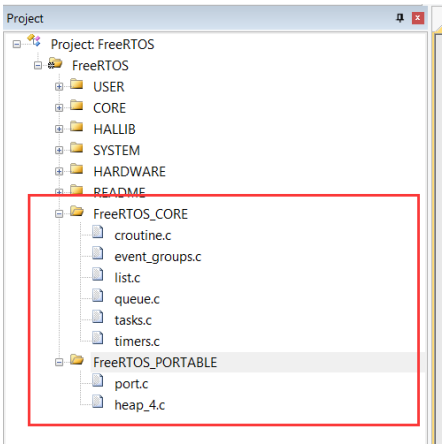


图 2.2.1.4 添加文件

分组 FreeRTOS_CORE 中的文件在什么地方就不说了，打开 FreeRTOS 源码一目了然。重点来说说 FreeRTOS_PORTABLE 分组中的 port.c 和 heap_4.c 是怎么来的，port.c 是 RVDS 文件夹下的 ARM_CM4F 中的文件，因为 STM32F429 是 Cortex-M4 内核并且带有 FPU，因此要选择 ARM_CM4F 中的 port.c 文件。heap_4.c 是 MemMang 文件夹中的，前面说了 MemMang 是跟内存管理相关的，里面有 5 个 c 文件：heap_1.c、heap_2.c、heap_3.c、heap_4.c 和 heap_5.c。这 5 个 c 文件是五种不同的内存管理方法，就像从北京到上海你可以坐火车、坐飞机，如果心情好的话也可以走路，反正有很多种方法，只要能到上海就行。这里也一样的，这 5 个文件都可以用来作为 FreeRTOS 的内存管理文件，只是它们的实现原理不同，各有利弊。这里我们选择 heap_4.c，至于原因，后面会有一章节专门来讲解 FreeRTOS 的内存管理，到时候大家就知道原因了。这里就先选择 heap_4.c，毕竟本章的重点是 FreeRTOS 的移植。

3、添加相应的头文件路径

添加完 FreeRTOS 源码中的 C 文件以后还要添加 FreeRTOS 源码的头文件路径，头文件路径如图 2.2.1.5 所示：

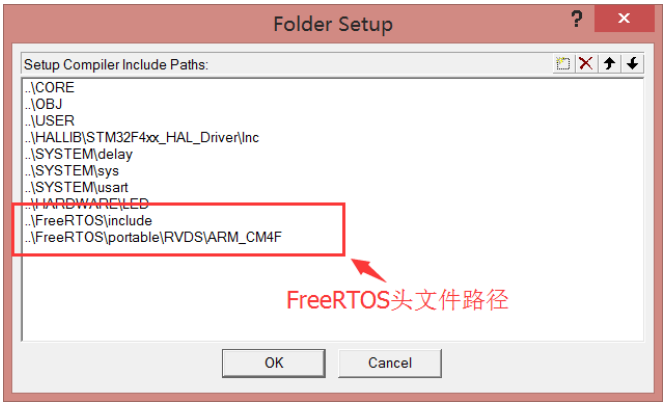


图 2.2.1.5 FreeRTOS 头文件路径

头文件路径添加完成以后编译一下，看看有没有什么错误，结果会发现提示打不开“FreeRTOSConfig.h”这个文件，如图 2.2.1.6 所示：

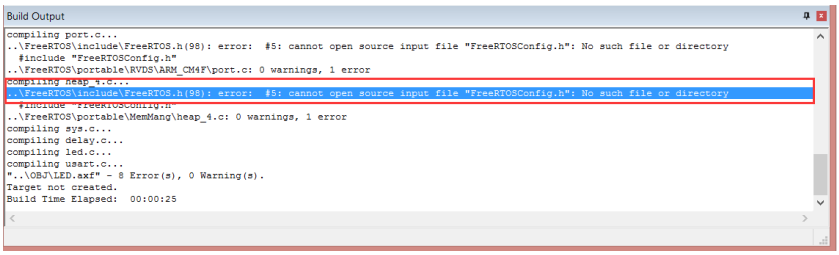


图 2.2.1.6 错误提示

这是因为缺少 FreeRTOSConfig.h 文件，这个文件在哪里找呢？你可以自己创建，显然这不是一个明智的做法。我们可以找找 FreeRTOS 的官方移植工程中会不会有这个文件，打开 FreeRTOS 针对 STM32F407 的移植工程文件，文件夹是 CORTEX_M4F_STM32F407ZG-SK，打开以后如图 2.2.1.7 所示：

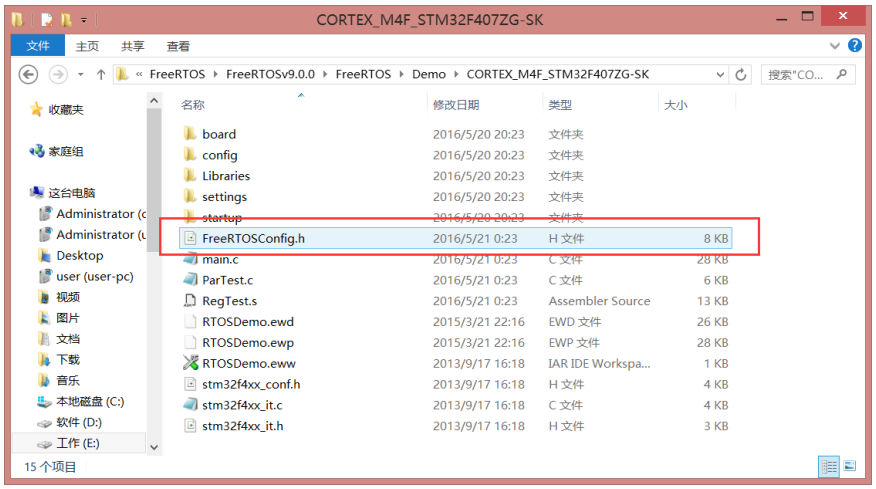


图 2.2.1.7 STM32F407 移植工程

果然！官方的移植工程中有这个文件，二话不说复制到我们的工程中去，至于复制到什么地方大家可以自行决定，这里我为了方便放到了 FreeRTOS 源码中的 include 文件夹下。

FreeRTOSConfig.h 是何方神圣？看名字就知道，他是 FreeRTOS 的配置文件，一般的操作系统都有裁剪、配置功能，而这些裁剪及配置都是通过一个文件来完成的，基本都是通过宏定义来完成对系统的配置和裁剪的，关于 FreeRTOS 的配置文件 FreeRTOSConfig.h 后面也会有一章节来详细的讲解。

到这里我们再编译一次，发现还是会提示有错误存在，如图 2.2.1.8 所示：

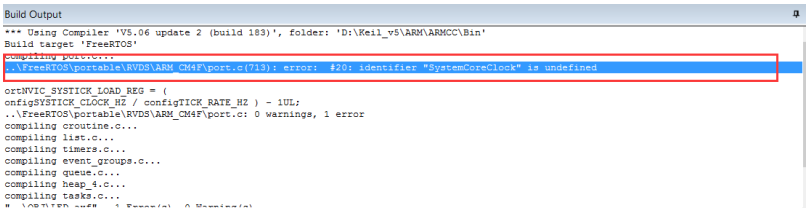


图 2.2.1.8 错误提示

这次的错误是 SystemCoreClock 未定义，这是因为在 FreeRTOSConfig.h 中使用到了 SystemCoreClock 来标记 MCU 的频率，如图 2.2.1.9 所示：

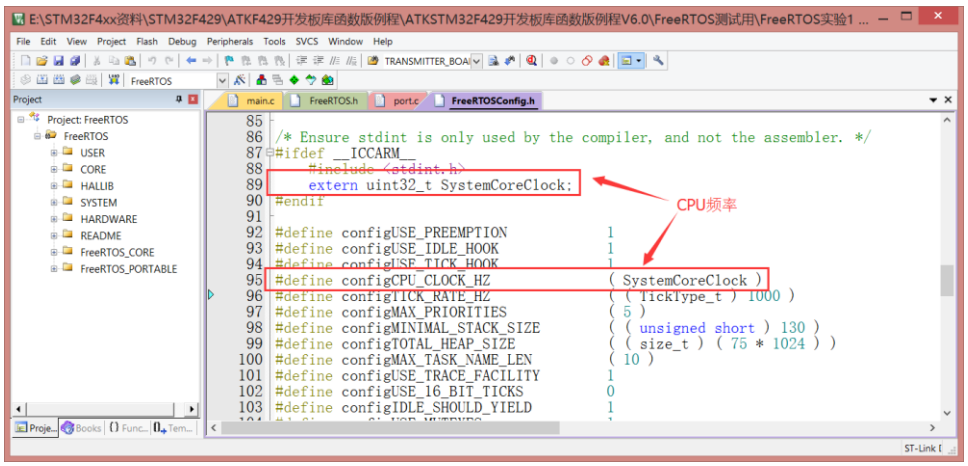


图 2.2.1.9 CPU 频率定义

注意，这里有个条件编译！只有定义__ICCARM__以后下面的代码才有效：

```
#include <stdint.h>
extern uint32_t SystemCoreClock;
```

我们需要修改这个条件编译，修改后的代码如下：

```
#if defined(__ICCARM__) || defined(__CC_ARM) || defined(__GNUC__)
    #include <stdint.h>
    extern uint32_t SystemCoreClock;
#endif
```

继续编译一次，还是有错误，如图 2.1.1.10：

图 2.2.1.10

这是因为 port.c 和 stm32f4xx_it.c 这两个文件中有重复定义的函数：PendSV_Handler()、SVC_Handler()和 Systick_Handler()，这里屏蔽掉 stm32f4xx_it.c 中的 PendSV_Handler()、SVC_Handler()和 Systick_Handler()这三个函数。

在编译一次，继续有错误，这次的错误是提示有一些未定义的函数，如图 2.2.1.11 所示：

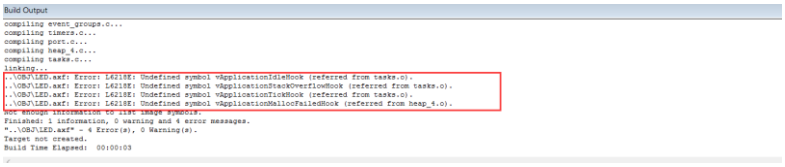


图 2.2.1.11 函数未定义

注意看这些未定义的函数，它们都是 Hook 结尾的，这些函数有个共同的名称：钩子函数，这是因为在 FreeRTOSConfig.h 中开启了这些钩子函数，但是却没有定义这些钩子函数而导致的，我们在 FreeRTOSConfig.h 中关闭这些钩子函数就行了，关闭的方法很简单，将相应的宏定义改为 0 即可，这里将宏 configUSE_IDLE_HOOK、configUSE_TICK_HOOK、configUSE_MALLOC_FAILED_HOOK 和 configCHECK_FOR_STACK_OVERFLOW 定义为 0。

最后编译一下，应该就没有错误了，如果还有错误的话大家自行根据错误类型查找和修改错误。

2.2.2 修改 SYSTEM 文件

SYSTEM 文件夹里面的文件一开始是针对 uC/OS 而编写的，所以如果使用 FreeRTOS 的话就需要做相应的修改。本来打算让 SYSTEM 文件夹也支持 FreeRTOS，但是这样的话会导致 SYSTEM 里面的文件太过于复杂，这样非常不利于初学者学习，所以这里就专门针对 FreeRTOS 修改了 SYSTEM 里面的文件。

1、修改 sys.h 文件

sys.h 文件修改很简单，在 sys.h 文件里面用宏 SYSTEM_SUPPORT_OS 来定义是否使用 OS，我们使用了 FreeRTOS，所以应该将宏 SYSTEM_SUPPORT_OS 改为 1。

```
//0,不支持 os
//1,支持 os
#define SYSTEM_SUPPORT_OS       1           //定义系统文件夹是否支持 OS
```

2、修改 usart.c 文件

usart.c 文件修改也很简单，usart.c 文件有两部分要修改，一个是添加 FreeRTOS.h 头文件，默认是添加的 uC/OS 中的 includes.h 头文件，修改以后如下：

```
//如果使用 os,则包括下面的头文件即可.
#if SYSTEM_SUPPORT_OS
#include "FreeRTOS.h"                   //os 使用
#endif
```

另外一个就是 USART1 的中断服务函数，在使用 uC/OS 的时候进出中断的时候需要添加 OSIntEnter()和 OSIntExit()，使用 FreeRTOS 的话就不需要了，所以将这两行代码删除掉，修改以后如下：

```
//串口 1 中断服务程序
void USART1_IRQHandler(void)
{
    u32 timeout=0;
    u32 maxDelay=0x1FFFF;

    HAL_UART_IRQHandler(&UART1_Handler);  //调用 HAL 库中断处理公用函数

    timeout=0;
```

```
    while (HAL_UART_GetState(&UART1_Handler) != HAL_UART_STATE_READY)//等待就绪
    {
        timeout++;////超时处理
        if(timeout>maxDelay) break;
    }

    timeout=0;
    //一次处理完成之后，重新开启中断并设置 RxXferCount 为 1
    while(HAL_UART_Receive_IT(&UART1_Handler, (u8 *)aRxBuffer, \
          RXBUFFERSIZE) != HAL_OK)
    {
        timeout++; //超时处理
        if(timeout>maxDelay) break;
    }
}
```

3、修改 delay.c 文件

delay.c 文件修改的就比较大了，因为涉及到 FreeRTOS 的系统时钟，delay.c 文件里面有 4 个函数，先来看一下函数 SysTick_Handler()，此函数是滴答定时器的中断服务函数，代码如下：

```
extern void xPortSysTickHandler(void);
//systick 中断服务函数,使用 OS 时用到
void SysTick_Handler(void)
{
    if(xTaskGetSchedulerState()!=taskSCHEDULER_NOT_STARTED)//系统已经运行
    {
        xPortSysTickHandler();
    }
    HAL_IncTick();
}
```

FreeRTOS 的心跳就是由滴答定时器产生的，根据 FreeRTOS 的系统时钟节拍设置好滴答定时器的周期，这样就会周期触发滴答定时器中断了。在滴答定时器中断服务函数中调用 FreeRTOS 的 API 函数 xPortSysTickHandler()，如果使用 HAL 库的话还需要调用 HAL 库中的函数 HAL_IncTick()！因为 HAL 库中的延时是靠 HAL_IncTick()来完成的。

delay_init()是用来初始化滴答定时器和延时函数，代码如下：

```
//初始化延时函数
//当使用 FreeRTOS 的时候,此函数会初始化 FreeRTOS 的时钟节拍
//SYSTICK 的时钟固定为 AHB 时钟
//SYSCLK:系统时钟频率
void delay_init(u8 SYSCLK)
{
    u32 reload;
    //SysTick 频率为 HCLK
    HAL_SYSTICK_CLKSourceConfig(SYSTICK_CLKSOURCE_HCLK);
    fac_us=SYSCLK;                          //不论是否使用 OS,fac_us 都需要使用
```

```
    reload=SYSCLK;                                  //每秒钟的计数次数 单位为 K
    reload*=1000000/configTICK_RATE_HZ;             //根据 configTICK_RATE_HZ 设定溢出时间
                                                    //reload 为 24 位寄存器,最大值:16777216,
                                                    //在 180M 下,约合 0.745s 左右
    fac_ms=1000/configTICK_RATE_HZ;                 //代表 OS 可以延时的最少单位
    SysTick->CTRL|=SysTick_CTRL_TICKINT_Msk;//开启 SYSTICK 中断
    SysTick->LOAD=reload;                           //每 1/configTICK_RATE_HZ 断一次
    SysTick->CTRL|=SysTick_CTRL_ENABLE_Msk; //开启 SYSTICK
}
```

前面我们说了 FreeRTOS 的系统时钟是由滴答定时器提供的，那么肯定要根据 FreeRTOS 的系统时钟节拍来初始化滴答定时器了，delay_init()就是来完成这个功能的。FreeRTOS 的系统时钟节拍由宏 configTICK_RATE_HZ 来设置，这个值我们可以自由设置，但是一旦设置好以后我们就要根据这个值来初始化滴答定时器，其实就是设置滴答定时器的中断周期。由于我们使用了 HAL 库，所以在设置滴答定时器的时候要照顾到 HAL 库，HAL 库里的延时函数要求滴答定时器周期为 1ms，因此 FreeRTOS 的系统节拍应该设置为 1000HZ，也就是 1ms 的周期。

接下来的三个函数都是延时的，代码如下：

```
//延时 nus
//nus:要延时的 us 数.
//nus:0~190887435(最大值即 2^32/fac_us@fac_us=22.5)
void delay_us(u32 nus)
{
    u32 ticks;
    u32 told,tnow,tcnt=0;
    u32 reload=SysTick->LOAD;                       //LOAD 的值
    ticks=nus*fac_us;                               //需要的节拍数
    told=SysTick->VAL;                              //刚进入时的计数器值
    while(1)
    {
        tnow=SysTick->VAL;
        if(tnow!=told)
        {
            //这里注意一下 SYSTICK 是一个递减的计数器就可以了.
            if(tnow<told)tcnt+=told-tnow;
            else tcnt+=reload-tnow+told;
            told=tnow;
            if(tcnt>=ticks)break;                   //时间超过/等于要延迟的时间,则退出.
        }
    };
}

//延时 nms,会引起任务调度
//nms:要延时的 ms 数
//nms:0~65535
```

```
void delay_ms(u32 nms)
{
    if(xTaskGetSchedulerState()!=taskSCHEDULER_NOT_STARTED)//系统已经运行
    {
        if(nms>=fac_ms)                 //延时的时间大于 OS 的最少时间周期
        {
            vTaskDelay(nms/fac_ms);     //FreeRTOS 延时
        }
        nms%=fac_ms;                    //OS 已经无法提供这么小的延时了,
                                        //采用普通方式延时
    }
    delay_us((u32)(nms*1000));          //普通方式延时
}
//延时 nms,不会引起任务调度
//nms:要延时的 ms 数
void delay_xms(u32 nms)
{
    u32 i;
    for(i=0;i<nms;i++) delay_us(1000);
}
```

delay_us()是 us 级延时函数，delay_ms 和 delay_xms()都是 ms 级的延时函数，delay_us()和 delay_xms()不会导致任务切换。delay_ms()其实就是对 FreeRTOS 中的延时函数 vTaskDelay()的简单封装，所以在使用 delay_ms()的时候就会导致任务切换。

delay.c 修改完成以后编译一下，会提示如图 2.2.2.1 所示错误：

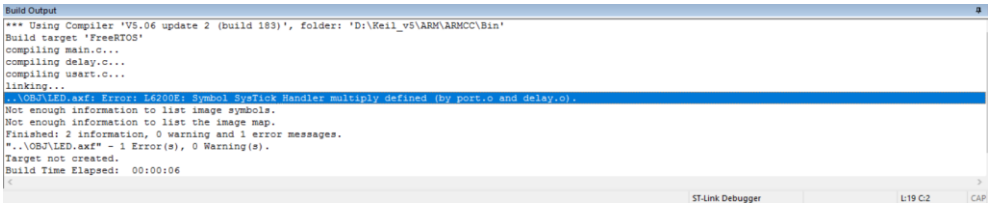


图 2.2.2.1 错误提示

从图 2.2.2.1 可以看出在 port.c 和 delay.c 中有重复定义的函数：SysTick_Handler()，二选一！很明显 delay.c 中的 SysTick_Handler()得留下来，打开 FreeRTOSConfig.h 文件，找到如下一个宏定义：

```
#define xPortSysTickHandler SysTick_Handler
```

屏蔽掉此宏定义，如图 2.2.2.2 所示：

```
152 to all Cortex-M ports, and do not rely on any particular library functions. */
153 #define configKERNEL_INTERRUPT_PRIORITY         ( configLIBRARY_LOWEST_INTERRUPT_PRIORITY << (8 - c
154 /* !!!! configMAX_SYSCALL_INTERRUPT_PRIORITY must not be set to zero !!!!
155 See http://www.FreeRTOS.org/RTOS-Cortex-M3-M4.html. */
156 #define configMAX_SYSCALL_INTERRUPT_PRIORITY    ( configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY << (
157
158 /* Normal assert() semantics without relying on the provision of an assert.h
159 header file. */
160 #define configASSERT( x ) if( ( x ) == 0 ) { taskDISABLE_INTERRUPTS(); for( ;; ); }
161
162 /* Definitions that map the FreeRTOS port interrupt handlers to their CMSIS
163 standard names. */
164 #define vPortSVCHandler SVC_Handler
165 #define xPortPendSVHandler PendSV_Handler
166 #define xPortSysTickHandler SysTick_Handler
167
168 #endif /* FREERTOS_CONFIG_H */
169
170
```

屏蔽掉此行代码

图 2.2.2.2 屏蔽掉宏定义

至此，SYSTEM 文件夹就修改完成了。

2.3 移植验证实验

2.3.1 实验程序设计

1、实验目的

编写简单的 FreeRTOS 应用代码，测试 FreeRTOS 的移植是否成功。鉴于大家还没正式学习 FreeRTOS，可以直接将本实验代码复制粘贴到自己的移植工程中。

2、实验设计

本实验设计四个任务：start_task()、led0_task ()、led1_task ()和 float_task()，这四个任务的任务功能如下：

start_task()：用来创建其他三个任务。

led0_task ()：控制 LED0 的闪烁，提示系统正在运行。

led1_task ()：控制 LED1 的闪烁。

float_task()：简单的浮点测试任务，用于测试 STM32F4 的 FPU 是否工作正常。

3、实验工程

FreeRTOS 实验 2-1 FreeRTOS 移植实验。

4、实验程序与分析

●任务设置

```
#include "sys.h"
#include "delay.h"
#include "usart.h"
#include "led.h"
#include "FreeRTOS.h"
#include "task.h"
/************************************************
 ALIENTEK 阿波罗 STM32F429 开发板 FreeRTOS 实验 2-1
 FreeRTOS 移植实验-HAL 库版本
 技术支持：www.openedv.com
 淘宝店铺：http://eboard.taobao.com
```

```
 关注微信公众平台微信号："正点原子"，免费获取 STM32 资料。
 广州市星翼电子科技有限公司
 作者：正点原子 @ALIENTEK
**********************************************/

#define START_TASK_PRIO          1          //任务优先级
#define START_STK_SIZE           128        //任务堆栈大小
TaskHandle_t StartTask_Handler;             //任务句柄
void start_task(void *pvParameters);        //任务函数

#define LED0_TASK_PRIO           2          //任务优先级
#define LED0_STK_SIZE            50         //任务堆栈大小
TaskHandle_t LED0Task_Handler;              //任务句柄
void led0_task(void *p_arg);                //任务函数


#define LED1_TASK_PRIO           3          //任务优先级
#define LED1_STK_SIZE            50         //任务堆栈大小
TaskHandle_t LED1Task_Handler;              //任务句柄
void led1_task(void *p_arg);                //任务函数

#define FLOAT_TASK_PRIO          4          //任务优先级
#define FLOAT_STK_SIZE           128        //任务堆栈大小
TaskHandle_t FLOATTask_Handler;             //任务句柄
void float_task(void *p_arg);               //任务函数
```

- **main()函数**

```
int main(void)
{
    HAL_Init();                             //初始化 HAL 库
    Stm32_Clock_Init(360,25,2,8);           //设置时钟,180Mhz
    delay_init(180);                        //初始化延时函数
    LED_Init();                             //初始化 LED
    uart_init(115200);                      //初始化串口
    //创建开始任务
    xTaskCreate((TaskFunction_t     )start_task,            //任务函数
                (const char*        )"start_task",          //任务名称
                (uint16_t           )START_STK_SIZE,        //任务堆栈大小
                (void*              )NULL,                  //传递给任务函数的参数
                (UBaseType_t        )START_TASK_PRIO,       //任务优先级
                (TaskHandle_t*      )&StartTask_Handler);   //任务句柄
    vTaskStartScheduler();                                  //开启任务调度
}
```

- **任务函数**

```
//开始任务任务函数
void start_task(void *pvParameters)
{
    taskENTER_CRITICAL();           //进入临界区
    //创建 LED0 任务
    xTaskCreate((TaskFunction_t     )led0_task,
                (const char*        )"led0_task",
                (uint16_t           )LED0_STK_SIZE,
                (void*              )NULL,
                (UBaseType_t        )LED0_TASK_PRIO,
                (TaskHandle_t*      )&LED0Task_Handler);
    //创建 LED1 任务
    xTaskCreate((TaskFunction_t     )led1_task,
                (const char*        )"led1_task",
                (uint16_t           )LED1_STK_SIZE,
                (void*              )NULL,
                (UBaseType_t        )LED1_TASK_PRIO,
                (TaskHandle_t*      )&LED1Task_Handler);
    //浮点测试任务
    xTaskCreate((TaskFunction_t     )float_task,
                (const char*        )"float_task",
                (uint16_t           )FLOAT_STK_SIZE,
                (void*              )NULL,
                (UBaseType_t        )FLOAT_TASK_PRIO,
                (TaskHandle_t*      )&FLOATTask_Handler);
    vTaskDelete(StartTask_Handler);     //删除开始任务
    taskEXIT_CRITICAL();            //退出临界区
}

//LED0 任务函数
void led0_task(void *pvParameters)
{
    while(1)
    {
        LED0=~LED0;
        vTaskDelay(500);
    }
}

//LED1 任务函数
void led1_task(void *pvParameters)
{
    while(1)
```

```
    {
        LED1=0;
        vTaskDelay(200);
        LED1=1;
        vTaskDelay(800);
    }
}

//浮点测试任务
void float_task(void *p_arg)
{
    static float float_num=0.00;
    while(1)
    {
        float_num+=0.01f;
        printf("float_num 的值为: %.4f\r\n",float_num);
        vTaskDelay(1000);
    }
}
```

测试代码中创建了 3 个任务：LED0 测试任务、LED1 测试任务和浮点测试任务，它们的任务函数分别为：led0_task()、led1_task()和 float_task()。led0_task()和 led1_task()任务很简单，就是让 LED0 和 LED1 周期性闪烁。因为 STM32F429 支持 FPU，所以必须测试一下 FreeRTOS 是否也支持 FPU。float_task()任务就是用来完成这个功能，定义一个 float 变量，然后每隔 1s 加 0.01，并且通过串口打印出来。

由于我们只是用测试代码来测试 FreeRTOS 是否移植成功的，所以关于具体的函数的调用方法这些不要深究，后面会有详细的讲解！

2.3.2 实验程序运行结果分析

编译并下载代码到 STM32F429 开发板中，下载进去以后会看到 LED0 和 LED1 开始闪烁，LED0 均匀闪烁，那是因为我们在 LED0 的任务代码中设置好的 LED0 亮 500ms，灭 500ms。LED1 亮的时间短，灭的时间长，这是因为在 LED1 的任务代码中设置好的亮 200ms，灭 800ms。在打开串口调试助手，串口调试助手接收到开发板发送过来的数据，如图 2.3.2.1 所示：

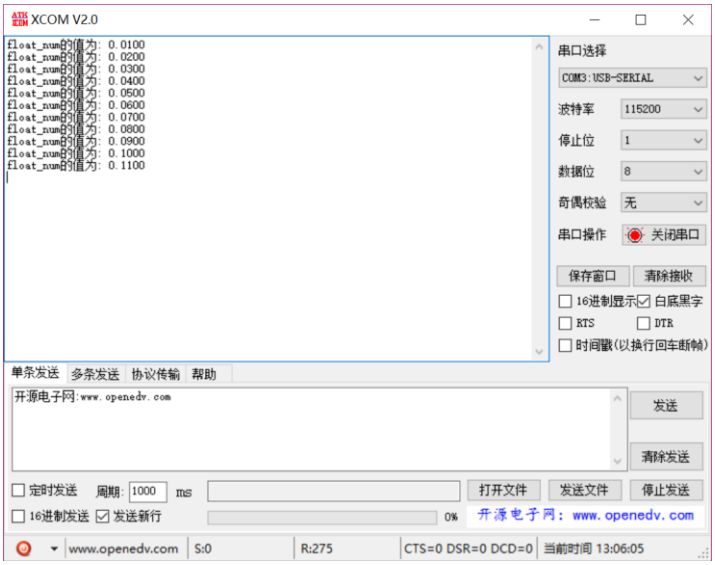


图 2.3.2.1 接受到的数据

从图 2.3.2.1 中可以看出，float_num 的值在不断的增加，每次增加 0.01，这和我们程序中设置的相符。但是这个不足以说明 FreeRTOS 支持 FPU，我们还需要查看是否使用到了浮点寄存器和相关的浮点指令，这个通过硬件调试来查看，进入调试界面，打开汇编窗口:View->Disassembly,在如图 2.3.2.2 所示地方设置断点。

```
90  }
91  //浮点测试任务
92  void float_task(void *p_arg)
93  {
94      static float float_num=0.00;
95      while(1)
96      {
97          float_num+=0.01f;
98          printf("float_num的值为: %.4f\r\n",float_num);
99          vTaskDelay(1000);
100     }
101 }
```

图 2.3.2.2 设置断点

点击全速运行图标，程序运行到图 2.3.2.2 设置的断点处停下来，查看汇编窗口，如图 2.3.2.3 所示：

```
0x080025A0 ED2D8B02  VPUSH.64      {d8-d8}
    97:                float_num+=0.01f;
    98:                printf("float_num的值为: %.4f\r\n",float_num);
0x080025A4 4C0C      LDR           r4,[pc,#48]  ; @0x080025D8
0x080025A6 ED9F8A0D  VLDR          s16,[pc,#0x34]
    99:            vTaskDelay(1000);
0x080025AA F44F757A  MOV           r5,#0x3E8
    97:                float_num+=0.01f;
0x080025AE ED940A00  VLDR          s0,[r4,#0x00]
0x080025B2 EE300A08  VADD.F32      s0,s0,s16
0x080025B6 ED840A00  VSTR          s0,[r4,#0x00]
```

图 2.3.2.3 汇编窗口

从图 2.3.2.3 中可以看出使用到了浮点寄存器 s0、s16，还使用到了浮点指令如：VLDR、VADD.F32,这说明 FreeRTOS 支持 FPU。至此，FreeRTOS 就移植到了 ALIENTEK 的 STM32F429 开发板上了。

第三章 FreeRTOS 系统配置

在实际使用 FreeRTOS 的时候我们时常需要根据自己需求来配置 FreeRTOS，而且不同架构的 MCU 在使用的时候配置也不同。FreeRTOS 的系统配置文件为 FreeRTOSConfig.h，在此配置文件中可以完成 FreeRTOS 的裁剪和配置，这是非常重要的一个文件，本章就来讲解这个这个文件，本章分为如下几部分：

3.1 FreeRTOSConfig.h 文件
3.2 “INCLUDE_”开始的宏
3.3 “config”开始的宏

3.1 FreeRTOSConfig.h 文件

FreeRTOS 的配置基本是通过在 FreeRTOSConfig.h 中使用“#define”这样的语句来定义宏定义实现的。在 FreeRTOS 的官方 demo 中，每个工程都有一个 FreeRTOSConfig.h 文件，我们在使用的时候可以参考这个文件，甚至直接复制粘贴使用。

3.1 “INCLUDE_”开始的宏

使用“INCLUDE_”开头的宏用来表示使能或除能 FreeRTOS 中相应的 API 函数，作用就是用来配置 FreeRTOS 中的可选 API 函数的。比如当宏 INCLUDE_vTaskPrioritySet 设置为 0 的时候表示不能使用函数 vTaskPrioritySet()，当设置为 1 的时候就表示可以使用函数 vTaskPrioritySet()。这个功能其实就是条件编译，在文件 tasks.c 中有如下图 3.1.1 所示的代码。

```
1400 #if ( INCLUDE_vTaskPrioritySet == 1 )
1401
1402     void vTaskPrioritySet( TaskHandle_t xTask, UBaseType_t uxNewPriority )
1403     {
1561 #endif /* INCLUDE_vTaskPrioritySet */
```

条件编译

图 3.1.1 条件编译

从图 3.1.1 可以看出当满足条件：NCLUDE_vTaskPrioritySet == 1 的时候，函数 vTaskPrioritySet()才会被编译，注意，这里为了缩小篇幅将函数 vTaskPrioritySet()的内容进行了折叠。FreeRTOS 中的裁剪和配置就是这种用条件编译的方法来实现的，不止 FreeRTOS 这么干，很多的协议栈、RTOS 系统和 GUI 库等都是使用条件编译的方法来完成配置和裁剪的。条件编译的好处就是节省空间，不需要的功能就不用编译，这样就可以根据实际需求来减少系统占用的 ROM 和 RAM 大小，根据自己所使用的 MCU 来调整系统消耗，降低成本。

下面来看看“INCLUDE_”开始的都有哪些宏，它们的功能都是什么。

1、INCLUDE_xSemaphoreGetMutexHolder

如果要使用函数 xQueueGetMutexHolder()的话宏 INCLUDE_xSemaphoreGetMutexHolder 必须定义为 1。

2、INCLUDE_xTaskAbortDelay

如果要使用函数 xTaskAbortDelay()的话将宏 INCLUDE_xTaskAbortDelay 定义为 1。

3、INCLUDE_vTaskDelay

如果要使用函数 vTaskDelay()的话需要将宏 INCLUDE_vTaskDelay 定义为 1。

4、INCLUDE_vTaskDelayUntil

如果要使用函数 vTaskDelayUntil()的话需要将宏 INCLUDE_vTaskDelayUntil 定义为 1。

5、INCLUDE_vTaskDelete

如果要使用函数 vTaskDelete()的话需要将宏 INCLUDE_vTaskDelete 定义为 1。

6、INCLUDE_xTaskGetCurrentTaskHandle

如果要使用函数 xTaskGetCurentTaskHandle()的话需要将宏 INCLUDE_xTaskGetCurrentTaskHandle 定义为 1。

7、INCLUDE_xTaskGetHandle

如果要使用函数 xTaskGetHandle()的话需要将宏 INCLUDE_xTaskGetHandle 定义为 1。

8、INCLUDE_xTaskGetIdleTaskHandle

如果要使用函数 xTaskGetIdleTaskHandle()的话需要将宏 INCLUDE_xTaskGetIdleTaskHandle 定义为 1。

9、INCLUDE_xTaskGetSchedulerState

如果要使用函数 xTaskGetSchedulerState()的话需要将宏 INCLUDE_xTaskGetSchedulerState 定义为 1。

10、INCLUDE_uxTaskGetStackHighWaterMark

如果要使用函数 uxTaskGetStackHighWaterMark()的话需要将宏 INCLUDE_uxTaskGetStackHighWaterMark 定义为 1。

11、INCLUDE_uxTaskPriorityGet

如果要使用函数 uxTaskPriorityGet()的话需要将宏 INCLUDE_uxTaskPriorityGet 定义为 1。

12、INCLUDE_vTaskPrioritySet

如果要使用函数 vTaskPrioritySet()的话需要将宏 INCLUDE_vTaskPrioritySet 定义为 1。

13、INCLUDE_xTaskResumeFromISR

如果要使用函数 xTaskResumeFromISR()的话需要将宏 INCLUDE_xTaskResumeFromISR 和 INCLUDE_vTaskSuspend 都定义为 1。

14、INCLUDE_eTaskGetState

如果要使用函数 eTaskGetState()的话需要将宏 INCLUDE_eTaskGetState 定义为 1。

15、INCLUDE_vTaskSuspend

如果要使用函数 vTaskSuspend()、vTaskResume()、prvTaskIsTaskSuspended()、xTaskResumeFromISR()的话宏 INCLUDE_vTaskSuspend 要定义为 1。

如果要使用函数 xTaskResumeFromISR()的话宏 INCLUDE_xTaskResumeFromISR 和 INCLUDE_vTaskSuspend 都必须定义为 1。

16、INCLUDE_xTimerPendFunctionCall

如果要使用函数 xTimerPendFunctionCall()和 xTimerPendFunctionCallFromISR()的话宏 INCLUDE_xTimerPendFunctionCall 和 configUSE_TIMERS 都必须定义为 1。

3.2 “config”开始的宏

“config”开始的宏和“INCLUDE_”开始的宏一样，都是用来完成 FreeRTOS 的配置和裁剪的，接下来我们就看一下这些“config”开始的宏。

1、configAPPLICATION_ALLOCATED_HEAP

默认情况下 FreeRTOS 的堆内存是由编译器来分配的，将宏

configAPPLICATION_ALLOCATED_HEAP 定义为 1 的话堆内存可以由用户自行设置，堆内存在 heap_1.c、heap_2.c、heap_3.c、heap_4.c 和 heap_5.c 中有定义，具体在哪个文件取决于用户的选择哪种内存管理方式。比如我们的例程选择了 heap_4.c，那么在 heap_4.c 中就有如图 3.2.1 所示定义：

```
100 /* Allocate the memory for the heap. */
101 #if( configAPPLICATION_ALLOCATED_HEAP == 1 )
102     /* The application writer has already defined the array used for the RTOS
103     heap - probably so it can be placed in a special segment or address. */
104     extern uint8_t ucHeap[ configTOTAL_HEAP_SIZE ];
105 #else
106     static uint8_t ucHeap[ configTOTAL_HEAP_SIZE ];
107 #endif /* configAPPLICATION_ALLOCATED_HEAP */
```

图 3.2.1 堆内存

从图 3.2.1 可以看出当宏 configAPPLICATION_ALLOCATED_HEAP 定义为 1 的话需要用户自行堆内存 ucHeap，否则的话就是编译器来分配的。

2、configASSERT

断言，类似 C 标准库中的 assert()函数，调试代码的时候可以检查传入的参数是否合理，FreeRTOS 内核中的关键点都会调用 configASSERT(x)，当 x 为 0 的时候说明有错误发生，使用断言的话会导致开销加大，一般在调试阶段使用。configASSERT()需要在 FreeRTOSConfig.h 文件中定义，如下实例：

```
#define configASSERT((x))    if((x)==0) vAssertCalled(__FILE_,__LINE__);
```

注意，vAssertCalled()函数需要用户自行去定义，可以是显示到 LCD 上的函数，也可以是通过串口打印出来的函数，本教程的所有例程采用如下的定义：

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

当参数 x 错误的时候就通过串口打印出发生错误的文件名和错误所在的行号，调试代码的可以使用断言，当调试完成以后尽量去掉断言，防止增加开销！

3、configCHECK_FOR_STACK_OVERFLOW

设置堆栈溢出检测，每个任务都有一个任务堆栈，如果使用函数 xTaskCreate()创建一个任务的话那么这个任务的堆栈是自动从 FreeRTOS 的堆(ucHeap)中分配的，堆栈的大小是由函数 xTaskCreate()的参数 usStackDepth 来决定的。如果使用函数 xTaskCreateStatic()创建任务的话任务堆栈是由用户设置的，参数 pxStackBuffer 为任务堆栈，一般是一个数组。

堆栈溢出是导致应用程序不稳定的主要因素，FreeRTOS 提供了两种可选的机制来帮助检测和调试堆栈溢出，不管使用哪种机制都要设置宏 configCHECK_FOR_STACK_OVERFLOW。如果使能了堆栈检测功能的话，即宏 configCHECK_FOR_STACK_OVERFLOW 不为 0，那么用户必须提供一个钩子函数(回调函数)，当内核检测到堆栈溢出以后就会调用这个钩子函数，此钩子函数原型如下：

```
void vApplicationStackOverflowHook( TaskHandle_t     xTask,
                                    char *           pcTaskName );
```

参数 xTask 是任务句柄，pcTaskName 是任务名字，要注意的是堆栈溢出太严重的话可能会损毁这两个参数，如果发生这种情况的话可以直接查看变量 pxCurrentTCB 来确定哪个任务发生了堆栈溢出。有些处理器可能在堆栈溢出的时候生成一个 fault 中断来提示这种错误，另外，堆栈溢出检测会增加上下文切换的开销，建议在调试的时候使用。

configCHECK_FOR_STACK_OVERFLOW==1，使用堆栈溢出检测方法 1。

上下文切换的时候需要保存现场，现场是保存在堆栈中的，这个时候任务堆栈使用率很可能达到最大值，方法一就是不断的检测任务堆栈指针是否指向有效空间，如果指向了无效空间的话就会调用钩子函数。方法一的优点就是快！但是缺点就是不能检测所有的堆栈溢出。

configCHECK_FOR_STACK_OVERFLOW==2，使用堆栈溢出检测方法 2。

使用方法二的话在创建任务的时候会向任务堆栈填充一个已知的标记值，方法二会一直检测堆栈后面的几个 bytes(标记值)是否被改写，如果被改写的话就会调用堆栈溢出钩子函数，方法二也会使用方法一中的机制！方法二比方法一要慢一些，但是对用户而言还是很快的！方法二能检测到几乎所有的堆栈溢出，但是也有一些情况检测不到，比如溢出值和标记值相同的时候。

3、configCPU_CLOCK_HZ

设置 CPU 的频率。

4、configSUPPORT_DYNAMIC_ALLOCATION

定义为 1 的话在创建 FreeRTOS 的内核对象的时候所需要的 RAM 就会从 FreeRTOS 的堆中动态的获取内存，如果定义为 0 的话所需的 RAM 就需要用户自行提供，默认情况下宏 configSUPPORT_DYNAMIC_ALLOCATION 为 1。

5、configENABLE_BACKWARD_COMPATIBILITY

FreeRTOS.h 中由一些列的#define 宏定义，这些宏定义都是一些数据类型名字，如下图 3.2.2 所示：

```
823 #if configENABLE_BACKWARD_COMPATIBILITY == 1
824     #define eTaskStateGet eTaskGetState
825     #define portTickType TickType_t
826     #define xTaskHandle TaskHandle_t
827     #define xQueueHandle QueueHandle_t
828     #define xSemaphoreHandle SemaphoreHandle_t
829     #define xQueueSetHandle QueueSetHandle_t
830     #define xQueueSetMemberHandle QueueSetMemberHandle_t
831     #define xTimeOutType TimeOut_t
832     #define xMemoryRegion MemoryRegion_t
833     #define xTaskParameters TaskParameters_t
834     #define xTaskStatusType TaskStatus_t
835     #define xTimerHandle TimerHandle_t
836     #define xCoRoutineHandle CoRoutineHandle_t
837     #define pdTASK_HOOK_CODE TaskHookFunction_t
838     #define portTICK_RATE_MS portTICK_PERIOD_MS
839     #define pcTaskGetTaskName pcTaskGetName
840     #define pcTimerGetTimerName pcTimerGetName
841     #define pcQueueGetQueueName pcQueueGetName
842     #define vTaskGetTaskInfo vTaskGetInfo
843
844     /* Backward compatibility within the scheduler code only - these definitic
845     are not really required but are included for completeness. */
846     #define tmrTIMER_CALLBACK TimerCallbackFunction_t
847     #define pdTASK_CODE TaskFunction_t
848     #define xListItem ListItem_t
849     #define xList List_t
850 #endif /* configENABLE_BACKWARD_COMPATIBILITY */
```

图 3.2.2 宏定义

在 V8.0.0 之前的 FreeRTOS 中会使用到这些数据类型，这些宏保证了你的代码从 V8.0.0 之前的版本升级到最新版本的时候不需要做出修改，默认情况下宏 configENABLE_BACKWARD_COMPATIBILITY 为 1。

6、configGENERATE_RUN_TIME_STATS

设置为 1 开启时间统计功能，相应的 API 函数会被编译，为 0 时关闭时间统计功能。如果宏 configGENERATE_RUN_TIME_STATS 为 1 的话还需要定义表 3.2.1 中的宏。

宏	描述
portCONFIGURE_TIMER_FOR_RUN_TIME_STATS()	此宏用来初始化一个外设来作为时间统计的基准时钟。
portGET_RUN_TIME_COUNTER_VALUE()或 portALT_GET_RUN_TIME_COUNTER_VALUE(Time)	此宏用来返回当前基准时钟的时钟值。

表 3.2.1 宏定义

7、configIDLE_SHOULD_YIELD

此宏定义了与空闲任务(idle Task)处于同等优先级的其他用户任务的行为，当为 0 的时候空闲任务不会为其他处于同优先级的任务让出 CPU 使用权。当为 1 的时候空闲任务就会为处于同等优先级的用户任务让出 CPU 使用权，除非没有就绪的用户任务，这样花费在空闲任务上的时间就会很少，但是这种方法也带了副作用，见图 3.2.3。

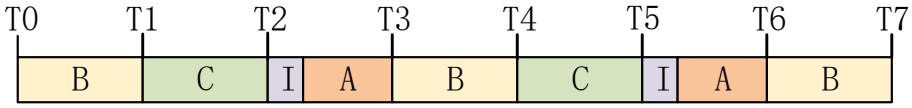


图 3.2.3 任务运行图

图 3.2.3 中有三个用户任务：A、B、C，还有一个空闲任务 I，用户任务和空闲任务处于同一优先级，任务切换发生在 T0~T7 时刻。T0~T1 之间的时间为一个时间片，从图中可以看出一开始任务 B、C 都执行了一个完成的时间片，在 T2 时刻空闲任务 I 开始执行，I 任务运行了一段时间以后被 A 任务抢走了 CPU 使用权，A 任务运行到 T3 时刻发生任务切换，B 任务开始运行。可以看出其实任务 I 和任务 A 一起使用了一个时间片，所以任务 A 运行的时间就比其他任务少！

一般建议大家关闭这个功能，毕竟空闲任务用不了多少时间，而且现在的 MCU 性能都很强！

8、configKERNEL_INTERRUPT_PRIORITY、
configMAX_SYSCALL_INTERRUPT_PRIORITY、
configMAX_API_CALL_INTERRUPT_PRIORITY

这三个宏和 FreeRTOS 的中断配置有关，后面会有专门的章节来讲解！

9、configMAX_CO_ROUTINE_PRIORITIES

设置可以分配给协程的最大优先级，也就是协程的优先级数。设置号以后协程的优先级可以从 0 到 configMAX_CO_ROUTINE_PRIORITIES-1，其中 0 是最低的优先级，configMAX_CO_ROUTINE_PRIORITIES-1 为最高的优先级。

10、configMAX_PRIORITIES

设置任务的优先级数量，设置好以后任务就可以使用从 0 到 configMAX_PRIORITIES-1 的优先级，其中 0 是最低优先级，configMAX_PRIORITIES-1 是最高优先级。注意和 uC/OS 的区别，uC/OS 中 0 是最高优先级！

11、configMAX_TASK_NAME_LEN

设置任务名最大长度。

12、configMINIMAL_STACK_SIZE

设置空闲任务的最小任务堆栈大小，以字为单位，不是字节。比如在 STM32 上设置为 100 的话，那么真正的堆栈大小就是 100*4=400 字节。

13、configNUM_THREAD_LOCAL_STORAGE_POINTERS

设置每个任务的本地存储指针数组大小，任务控制块中有本地存储数组指针，用户应用程序可以在这些本地存储中存入一些数据。

14、configQUEUE_REGISTRY_SIZE

设置可以注册的队列和信号量的最大数量，在使用内核调试器查看信号量和队列的时候需要设置此宏，而且要先将消息队列和信号量进行注册，只有注册了的队列和信号量才会再内核调试器中看到，如果不使用内核调试器的话此宏设置为 0 即可。

15、configSUPPORT_STATIC_ALLOCATION

当此宏定义为 1，在创建一些内核对象的时候需要用户指定 RAM，当为 0 的时候就会自使用 heap.c 中的动态内存管理函数来自动的申请 RAM。

16、configTICK_RATE_HZ

设置 FreeRTOS 的系统时钟节拍频率，单位为 HZ，此频率就是滴答定时器的中断频率，需要使用此宏来配置滴答定时器的中断，前面在讲 delay.c 文件的时候已经说过了。为了兼容 ST 最新的 HAL 库，我们将此宏设置为 1000，周期就是 1ms。

17、configTIMER_QUEUE_LENGTH

此宏是配置 FreeRTOS 软件定时器的，FreeRTOS 的软件定时器 API 函数会通过命令队列向软件定时器任务发送消息，此宏用来设置这个软件定时器的命令队列长度。

18、configTIMER_TASK_PRIORITY

设置软件定时器任务的任务优先级。

19、configTIMER_TASK_STACK_DEPTH

设置定时器服务任务的任务堆栈大小。

20、configTOTAL_HEAP_SIZE

设置堆大小，如果使用了动态内存管理的话，FreeRTOS 在创建任务、信号量、队列等的时候就会使用 heap_x.c(x 为 1~5)中的内存申请函数来申请内存。这些内存就是从堆 ucHeap[configTOTAL_HEAP_SIZE]中申请的，堆的大小由 configTOTAL_HEAP_SIZE 来定义。

21、configUSE_16_BIT_TICKS

设置系统节拍计数器变量数据类型，系统节拍计数器变量类型为 TickType_t，当 configUSE_16_BIT_TICKS 为 1 的时候 TickType_t 就是 16 位的，当 configUSE_16_BIT_TICKS 为 0 的话 TickType_t 就是 32 位的。

22、configUSE_APPLICATION_TASK_TAG

此宏设置为 1 的话函数 configUSE_APPLICATION_TASK_TAGF() 和

xTaskCallApplicationTaskHook()就会被编译。

23、configUSE_CO_ROUTINES

此宏为 1 的时候启用协程，协程可以节省开销，但是功能有限，现在的 MCU 性能已经非常强大了，建议关闭协程。

24、configUSE_COUNTING_SEMAPHORES

设置为 1 的时候启用计数型信号量，相关的 API 函数会被编译。

25、configUSE_DAEMON_TASK_STARTUP_HOOK

当宏 configUSE_TIMERS 和 configUSE_DAEMON_TASK_STARTUP_HOOK 都为 1 的时需要定义函数 vApplicationDaemonTaskStartupHook()，函数原型如下：

```
void vApplicationDaemonTaskStartupHook( void )
```

26、configUSE_IDLE_HOOK

为 1 时使用空闲任务钩子函数，用户需要实现空闲任务钩子函数，函数的原型如下：

```
void vApplicationIdleHook( void )
```

27、configUSE_MALLOC_FAILED_HOOK

为 1 时使用内存分配失败钩子函数，用户需要实现内存分配失败钩子函数，函数原型如下：

```
void vApplicationMallocFailedHook( void )
```

28、configUSE_MUTEXES

为 1 时使用互斥信号量，相关的 API 函数会被编译。

29、configUSE_PORT_OPTIMISED_TASK_SELECTION

FreeRTOS 有两种方法来选择下一个要运行的任务，一个是通用的方法，另外一个是特殊的方法，也就是硬件方法，使用 MCU 自带的硬件指令来实现。

通用方法：

- 当宏 configUSE_PORT_OPTIMISED_TASK_SELECTION 为 0，或者硬件不支持的时候。
- 希望所有硬件通用的时候。
- 全部用 C 语言来实现，但是效率比特殊方法低。
- 不限制最大优先级数目的时候。

特殊方法：

- 不是所有的硬件都支持。
- 当宏 configUSE_PORT_OPTIMISED_TASK_SELECTION 为 1 的时候。
- 硬件拥有特殊的指令，比如计算前导零(CLZ)指令。
- 比通用方法效率高。
- 会限制优先级数目，一般是 32 个。

STM32 有计算前导零的指令，所以我们可以使用特殊方法，即将宏 configUSE_PORT_OPTIMISED_TASK_SELECTION 定义为 1。计算前导零的指令在 uC/OSIII 也用到了，也是用来查找下一个要运行的任务的。

30、configUSE_PREEMPTION

为 1 时使用抢占式调度器，为 0 时使用协程。如果使用抢占式调度器的话内核会在每个时钟节拍中断中进行任务切换，当使用协程的话会在如下地方进行任务切换：

- 一个任务调用了函数 taskYIELD()。
- 一个任务调用了可以使任务进入阻塞态的 API 函数。
- 应用程序明确定义了在中断中执行上下文切换。

31、configUSE_QUEUE_SETS

为 1 时启用队列集功能。

32、configUSE_RECURSIVE_MUTEXES

为 1 时使用递归互斥信号量，相关的 API 函数会被编译。

33、configUSE_STATS_FORMATTING_FUNCTIONS

宏 configUSE_TRACE_FACILITY 和 configUSE_STATS_FORMATTING_FUNCTIONS 都为 1 的时候函数 vTaskList()和 vTaskGetRunTimeStats()会被编译。

34、configUSE_TASK_NOTIFICATIONS

为 1 的时候使用任务通知功能，相关的 API 函数会被编译，开启了此功能的话每个任务会多消耗 8 个字节。

35、configUSE_TICK_HOOK

为 1 时使能时间片钩子函数，用户需要实现时间片钩子函数，函数的原型如下：

```
void vApplicationTickHook( void )
```

36、configUSE_TICKLESS_IDLE

为 1 时使能低功耗 tickless 模式。

37、configUSE_TIMERS

为 1 时使用软件定时器，相关的 API 函数会被编译，当宏 configUSE_TIMERS 为 1 的话，那么宏 configTIMER_TASK_PRIORITY 、configTIMER_QUEUE_LENGTH 和 configTIMER_TASK_STACK_DEPTH 必须定义。

38、configUSE_TIME_SLICING

默认情况下，FreeRTOS 使用抢占式调度器，这意味着调度器永远都在执行已经就绪了的最高优先级任务，优先级相同的任务在时钟节拍中断中进行切换，当宏 configUSE_TIME_SLICING 为 0 的时候不会在时钟节拍中断中执行相同优先级任务的任务切换，默认情况下宏 configUSE_TIME_SLICING 为 1。

39、configUSE_TRACE_FACILITY

为 1 启用可视化跟踪调试，会增加一些结构体成员和 API 函数。

FreeRTOS 的配置文件基本就这些，还有一些其他的配置宏由于使用的比较少这里并没有列

出来，这些配置具体使用到的时候在具体查看就行了。

第四章 FreeRTOS 中断配置和临界段

FreeRTOS 的中断配置是一个很重要的内容，需要根据所使用的 MCU 来具体配置。这需要了解 MCU 架构中有关中断的知识，本章会结合 Cortex-M 的 NVIC 来讲解 STM32 平台下的 FreeRTOS 中断配置，本章分为如下几部分：

4.1 Cortex-M 中断
4.2 FreeRTOS 中断配置宏
4.3 FreeRTOS 开关中断
4.4 临界段代码
4.5 FreeRTOS 中断测试实验

4.1 Cortex-M 中断

4.1.1 中断简介

中断是微控制器一个很常见的特性，中断由硬件产生，当中断产生以后 CPU 就会中断当前的流程转而去处理中断服务，Cortex-M 内核的 MCU 提供了一个用于中断管理的嵌套向量中断控制器(NVIC)。

Cotex-M3 和 M4 的 NVIC 最多支持 240 个 IRQ(中断请求)、1 个不可屏蔽中断(NMI)、1 个 Systick(滴答定时器)定时器中断和多个系统异常。

4.1.2 中断管理简介

Cortex-M 处理器有多个用于管理中断和异常的可编程寄存器，这些寄存器大多数都在 NVIC 和系统控制块(SCB)中，CMSIS 将这些寄存器定义为结构体。以 STM32F429 为例，打开 core_cm4.h，有两个结构体，NVIC_Type 和 SCB_Type，如下：

```
typedef struct
{
  __IOM uint32_t ISER[8U];              /*!< Offset: 0x000 (R/W)  Interrupt Set Enable Register */
        uint32_t RESERVED0[24U];
  __IOM uint32_t ICER[8U];              /*!< Offset: 0x080 (R/W)  Interrupt Clear Enable Register */
        uint32_t RSERVED1[24U];
  __IOM uint32_t ISPR[8U];              /*!< Offset: 0x100 (R/W)  Interrupt Set Pending Register */
        uint32_t RESERVED2[24U];
  __IOM uint32_t ICPR[8U];              /*!< Offset: 0x180 (R/W)  Interrupt Clear Pending Register */
        uint32_t RESERVED3[24U];
  __IOM uint32_t IABR[8U];              /*!< Offset: 0x200 (R/W)  Interrupt Active bit Register */
        uint32_t RESERVED4[56U];
  __IOM uint8_t  IP[240U];     /*!< Offset: 0x300 (R/W)  Interrupt Priority Register (8Bit wide) */
        uint32_t RESERVED5[644U];
  __OM  uint32_t STIR;          /*!< Offset: 0xE00 ( /W)  Software Trigger Interrupt Register */
}  NVIC_Type;

typedef struct
{
  __IM  uint32_t CPUID;         /*!< Offset: 0x000 (R/ )  CPUID Base Register */
  __IOM uint32_t ICSR;          /*!< Offset: 0x004 (R/W)  Interrupt Control and State Register */
  __IOM uint32_t VTOR;          /*!< Offset: 0x008 (R/W)  Vector Table Offset Register */
  __IOM uint32_t AIRCR;         /*!< Offset: 0x00C (R/W)  Application Interrupt and Reset
                                     Control Register */
  __IOM uint32_t SCR;           /*!< Offset: 0x010 (R/W)  System Control Register */
  __IOM uint32_t CCR;           /*!< Offset: 0x014 (R/W)  Configuration Control Register */
  __IOM uint8_t  SHP[12U];      /*!< Offset: 0x018 (R/W)  System Handlers Priority
                                Registers (4-7, 8-11, 12-15) */
  __IOM uint32_t SHCSR;         /*!< Offset: 0x024 (R/W) System Handler Control and
                                State Register */
```

```
  __IOM uint32_t CFSR;          /*!< Offset: 0x028 (R/W)  Configurable Fault Status Register */
  __IOM uint32_t HFSR;          /*!< Offset: 0x02C (R/W)  HardFault Status Register */
  __IOM uint32_t DFSR;          /*!< Offset: 0x030 (R/W)  Debug Fault Status Register */
  __IOM uint32_t MMFAR;         /*!< Offset: 0x034 (R/W)  MemManage Fault Address Register */
  __IOM uint32_t BFAR;          /*!< Offset: 0x038 (R/W)  BusFault Address Register */
  __IOM uint32_t AFSR;          /*!< Offset: 0x03C (R/W)  Auxiliary Fault Status Register */
  __IM  uint32_t PFR[2U];       /*!< Offset: 0x040 (R/ )  Processor Feature Register */
  __IM  uint32_t DFR;           /*!< Offset: 0x048 (R/ )  Debug Feature Register */
  __IM  uint32_t ADR;           /*!< Offset: 0x04C (R/ )  Auxiliary Feature Register */
  __IM  uint32_t MMFR[4U];      /*!< Offset: 0x050 (R/ )  Memory Model Feature Register */
  __IM  uint32_t ISAR[5U];      /*!< Offset: 0x060 (R/ )  Instruction Set Attributes Register */
  uint32_t RESERVED0[5U];
  __IOM uint32_t CPACR;         /*!< Offset: 0x088 (R/W)  Coprocessor Access Control Register */
} SCB_Type;
```

NVIC 和 SCB 都位于系统控制空间(SCS)内，SCS 的地址从 0XE000E000 开始，SCB 和 NVIC 的地址也在 core_cm4.h 中有定义，如下：

```
#define SCS_BASE      (0xE000E000UL)              /*!< System Control Space Base Address */
#define NVIC_BASE     (SCS_BASE + 0x0100UL)       /*!< NVIC Base Address */
#define SCB_BASE      (SCS_BASE + 0x0D00UL)       /*!< System Control Block Base Address

#define SCnSCB    ((SCnSCB_Type*) SCS_BASE)       /*!< System control Register not in SCB */
#define SCB         ((SCB_Type    *) SCB_BASE)    /*!< SCB configuration struct */
#define NVIC        ((NVIC_Type *) NVIC_BASE    ) /*!< NVIC configuration struct *//
```

以上的中断控制寄存器我们在移植 FreeRTOS 的时候是不需要关心的，这里只是提一下，大家要是感兴趣的话可以参考 Cortex-M 的权威指南，我们重点关心的是是三个中断屏蔽寄存器：PRIMASK、FAULTMASK 和 BASEPRI，这三个寄存器后面会详细的讲解。

4.1.3 优先级分组定义

当多个中断来临的时候处理器应该响应哪一个中断是由中断的优先级来决定的，高优先级的中断(优先级编号小)肯定是首先得到响应，而且高优先级的中断可以抢占低优先级的中断，这个就是中断嵌套。Cortex-M 处理器的有些中断是具有固定的优先级的，比如复位、NMI、HardFault，这些中断的优先级都是负数，优先级也是最高的。

Cortex-M 处理器有三个固定优先级和 256 个可编程的优先级，最多有 128 个抢占等级，但是实际的优先级数量是由芯片厂商来决定的。但是，绝大多数的芯片都会精简设计的，以致实际上支持的优先级数会更少，如 8 级、16 级、32 级等，比如 STM32 就只有 16 级优先级。在设计芯片的时候会裁掉表达优先级的几个低端有效位，以减少优先级数，所以不管用多少位来表达优先级，都是 MSB 对齐的，如图 4.1.3.1 就是使用三位来表达优先级。

Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
用于表达优先级			没有实现，读回零				

图 4.1.3.1 使用三位表达优先级

在图 4.1.3.1 中，Bit0~Bit4 没有实现，所以读它们总是返回零，写如它们的话则会忽略写入

的值。因此，对于 3 个位的情况，可是使用的优先级就是 8 个：0X00(最高优先级)、0X20、0X40、0X60、0X80、0XA0、0XC0 和 0XE0。注意，这个是芯片厂商来决定的！不是我们能决定的，比如 STM32 就选择了 4 位作为优先级！

有读者可能就会问，优先级配置寄存器是 8 位宽的，为什么却只有 128 个抢占等级？8 位不应该是 256 个抢占等级吗？为了使抢占机能变得更可控，Cortex-M 处理器还把 256 个优先级按位分为高低两段：抢占优先级(分组优先级)和亚优先级(子优先级)，NVIC 中有一个寄存器是“应用程序中断及复位控制寄存器(AIRCR)”，AIRCR 寄存器里面有个位段名为“优先级组”，如表 4.1.3.1 所示：

位段	名称	类型	复位值	描述
[31:16]	VECTKEY	RW	-	访问钥匙：任何对该寄存器的写操作，都必须同时 0X05FA 写入此段，否则写操作被忽略。如读取此半字，则读回值为 0XFA05
15	ENDIANESS	R	-	指示端设置：1==大端(BE8)，0==小端，此值是在复位时确定的，不能更改。
[10:8]	PRIGROUP	R/W	0	优先级分组
2	SYSRESETREQ	W	-	请求芯片控制逻辑产生一次复位。
1	VECTCLRACTIVE	W	-	清零所有异常的活动状态信息，通常只在调试时用，或者在 OS 从错误中恢复时用。
0	VECTRESET	W	-	复位微控制器内核(调试逻辑除外)，但此复位不影响芯片上在内核以外的电路

表 4.1.3.1 AIRCR 寄存器

表 4.1.3.1 中 PRIGROUP 就是优先级分组，它把优先级分为两个位段：MSB 所在的位段(左边的)对应抢占优先级，LSB 所在的位段(右边的)对应亚优先级，如表 4.1.3.2 所示。

分组位置	表达抢占优先级的位段	表达亚优先级的位段
0(默认)	[7:1]	[0:0]
1	[7:2]	[1:0]
2	[7:3]	[2:0]
3	[7:4]	[3:0]
4	[7:5]	[4:0]
5	[7:6]	[5:0]
6	[7:7]	[6:0]
7	无	[7:0]

表 4.1.3.2 抢占优先级和亚优先级的表达，位数与分组位置的关系

在看一下 STM32 的优先级分组情况，我们前面说了 STM32 使用了 4 位，因此最多有 5 组优先级分组设置，这 5 个分组在 stm32f4xx_hal_cortex.h 中有定义，如下：

```
#define NVIC_PRIORITYGROUP_0   ((uint32_t)0x00000007) /*!< 0 bits for pre-emption priority
                                                          4 bits for subpriority */
#define NVIC_PRIORITYGROUP_1   ((uint32_t)0x00000006) /*!< 1 bits for pre-emption priority
                                                          3 bits for subpriority */
#define NVIC_PRIORITYGROUP_2   ((uint32_t)0x00000005) /*!< 2 bits for pre-emption priority
                                                          2 bits for subpriority */
#define NVIC_PRIORITYGROUP_3   ((uint32_t)0x00000004) /*!< 3 bits for pre-emption priority
```

```
                                                    1 bits for subpriority */
#define NVIC_PRIORITYGROUP_4    ((uint32_t)0x00000003) /*!< 4 bits for pre-emption priority
```

可以看出 STM32 有 5 个分组，但是一定要注意！STM32 中定义的分组 0 对应的值是 7！如果我们选择分组 4，即 NVIC_PRIORITYGROUP_4 的话，那 4 位优先级就都全是抢占优先级了，没有亚优先级，那么就有 0~15 共 16 个优先级。而移植 FreeRTOS 的时候我们配置的就是组 4，ST 官方默认的也是组 4，优先级分组配置在 HAL_Init()中，如图 4.1.3.2 所示：

```
166   */
167 HAL_StatusTypeDef HAL_Init(void)
168 {
169   /* Configure Flash prefetch, Instruction cache, Data cache */
170 #if (INSTRUCTION_CACHE_ENABLE != 0)
171     __HAL_FLASH_INSTRUCTION_CACHE_ENABLE();
172 #endif /* INSTRUCTION_CACHE_ENABLE */
173
174 #if (DATA_CACHE_ENABLE != 0)
175     __HAL_FLASH_DATA_CACHE_ENABLE();
176 #endif /* DATA_CACHE_ENABLE */
177
178 #if (PREFETCH_ENABLE != 0)
179     __HAL_FLASH_PREFETCH_BUFFER_ENABLE();
180 #endif /* PREFETCH_ENABLE */
181
182   /* Set Interrupt Group Priority */
183   HAL_NVIC_SetPriorityGrouping(NVIC_PRIORITYGROUP_4);//中断优先级分组4
184
185   /* Use systick as time base source and configure 1ms tick (default clock after Reset is HSI) */
186   HAL_InitTick(TICK_INT_PRIORITY);
187
188   /* Init the low level hardware */
189   HAL_MspInit();
190
191   /* Return function status */
192   return HAL_OK;
193 }
```

图 4.1.3.2 优先级分组配置

如果使用 ALIENTEK 的基础例程的话默认配置的组 2，所以在将基础例程中的外设驱动移植到 FreeRTOS 下面的时候需要修改优先级配置。主要是 FreeRTOS 的中断配置没有处理亚优先级这种情况，所以只能配置为组 4，直接就 16 个优先级，使用起来也简单！

4.1.4 优先级设置

每个外部中断都有一个对应的优先级寄存器，每个寄存器占 8 位，因此最大宽度是 8 位，但是最小为 3 位。4 个相临的优先级寄存器拼成一个 32 位寄存器。如前所述，根据优先级组的设置，优先级又可以分为高、低两个位段，分别抢占优先级和亚优先级。STM32 我们已经设置位组 4，所以就只有抢占优先级了。优先级寄存器都可以按字节访问，当然也可以按半字/字来访问，有意义的优先级寄存器数目由芯片厂商来实现，如表 4.1.4.1 和 4.1.4.2 所示：

名称	类型	地址	复位值	描述
PRI_0	R/W	0xE000_E400	0(8 位)	外中断#0 的优先级
PRI_1	R/W	0xE000_E401	0(8 位)	外中断#1 的优先级
⋮	⋮	⋮	⋮	⋮
PRI_239	R/W	0xE000_E4EF	0(8 位)	外中断#239 的优先级

表 4.1.4.1 中断优先级寄存器阵列(地址：0xE000_E400~0xE000_E4EF)

名称	类型	地址	复位值	描述
PRI_4		0xE000_ED18		存储管理 fault 的优先级
PRI_5		0xE000_ED19		总线 fault 的优先级
PRI_6		0xE000_ED1A		用法 fault 的优先级
-	-	0xE000_ED1B	-	-
-	-	0xE000_ED1C	-	-
-	-	0xE000_ED1D	-	-
-	-	0xE000_ED1E	-	-

PRI_11		0xE000_ED1F		SVC 优先级
PRI_12		0xE000_ED20		调试监视器的优先级
-	-	0xE000_ED21	-	-
PRI_14		0xE000_ED22		PendSV 的优先级
PRI_15		0xE000_ED23		SysTick 的优先级

表 4.1.4.2 系统异常优先级阵列(地址：0XE000_ED18~0xE000_ED23)

上面说了，4 个相临的寄存器可以拼成一个 32 位的寄存器，因此地址 0xE000_ED20~0xE000_ED23 这四个寄存器就可以拼接成一个地址为 0xE000_ED20 的 32 位寄存器。这一点很重要！因为 FreeRTOS 在设置 PendSV 和 SysTick 的中断优先级的时候都是直接操作的地址 0xE000_ED20。

4.1.5 用于中断屏蔽的特殊寄存器

在 4.1.2 小节中说了我们在 STM32 上移植 FreeRTOS 的时候需要重点关注 PRIMASK、FAULTMASK 和 BASEPRI 这三个寄存器，本节就来学习一下这三个寄存器。

1、PRIMASK 和 FAULTMASK 寄存器

在许多应用中，需要暂时屏蔽所有的中断一执行一些对时序要求严格的任务，这个时候就可以使用 PRIMASK 寄存器，PRIMASK 用于禁止除 NMI 和 HardFalut 外的所有异常和中断，汇编编程的时候可以使用 CPS(修改处理器状态)指令修改 PRIMASK 寄存器的数值：

```
CPSIE   I;   //清除 PRIMASK(使能中断)
CPSID   I;   //设置 PRIMASK(禁止中断)
```

PRIMASK 寄存器还可以通过 MRS 和 MSR 指令访问，如下：

```
MOVS    R0,  #1
MSR     PRIMASK,    R0   ;//将 1 写入 PRIMASK 禁止所有中断
```

以及：

```
MOVS    R0,  #0
MSR     PRIMASK,    R0   ;//将 0 写入 PRIMASK 以使能中断
```

uC/OS 中的临界区代码代码保护就是通过开关中断实现的(uC/OSIII 也可以使用禁止任务调度的方法来实现临界区代码保护，这里不讨论这种情况)，而开关中断就是直接操作 PRIMASK 寄存器的，所以在 uC/OS 中关闭中断的时候时关闭了除复位、NMI 和 HardFault 以外的所有中断！

FAULTMASK 比 PRIMASK 更狠，它可以连 HardFault 都屏蔽掉，使用方法和 PRIMASK 类似，FAULTMASK 会在退出时自动清零。

汇编编程的时候可以利用 CPS 指令修改 FAULTMASK 的当前状态：

```
CPSIE   F   ;清除 FAULTMASK
CPSID   F   ;设置 FAULTMASK
```

还可以利用 MRS 和 MSR 指令访问 FAULTMASK 寄存器：

```
MOVS    R0,  #1
MSR FAULTMASK,  R0   ;将 1 写入 FAULTMASK 禁止所有中断
```

以及：

```
MOVS    R0,  #0
MSR     FAULTMASK,  R0   ;将 0 写入 FAULTMASK 使能中断
```

2、BASEPRI 寄存器

PRIMASK 和 FAULTMASK 寄存器太粗暴了，直接关闭除复位、NMI 和 HardFault 以外的其他所有中断，但是在有些场合需要对中断屏蔽进行更细腻的控制，比如只屏蔽优先级低于某一个阈值的中断。那么这个作为阈值的优先级值存储在哪里呢？在 BASEPRI 寄存器中，不过如果向 BASEPRI 写 0 的话就会停止屏蔽中断。比如，我们要屏蔽优先级不高于 0X60 的中断，则可以使用如下汇编编程：

```
MOV     R0，     #0X60
MSR     BASEPRI, R0
```

如果需要取消 BASEPRI 对中断的屏蔽，可以使用如下代码：

```
MOV     R0,      #0
MSR     BASEPRI, R0
```

注意！FreeRTOS 的开关中断就是操作 BASEPRI 寄存器来实现的！它可以关闭低于某个阈值的中断，高于这个阈值的中断就不会被关闭！

4.2 FreeRTOS 中断配置宏

4.2.1 configPRIO_BITS

此宏用来设置 MCU 使用几位优先级，STM32 使用的是 4 位，因此此宏为 4！

4.2.2 configLIBRARY_LOWEST_INTERRUPT_PRIORITY

此宏是用来设置最低优先级，前面说了，STM32 优先级使用了 4 位，而且 STM32 配置的使用组 4，也就是 4 位都是抢占优先级。因此优先级数就是 16 个，最低优先级那就是 15。所以此宏就是 15，注意！不同的 MCU 此值不同，具体是多少要看所使用的 MCU 的架构，本教程只针对 STM32 讲解！

4.2.3 configKERNEL_INTERRUPT_PRIORITY

此宏用来设置内核中断优先级，此宏定义如下：

```
#define  configKERNEL_INTERRUPT_PRIORITY
    ( configLIBRARY_LOWEST_INTERRUPT_PRIORITY << (8 - configPRIO_BITS) )
```

宏 configKERNEL_INTERRUPT_PRIORITY 为，宏 configLIBRARY_LOWEST_INTERRUPT_PRIORITY 左移 8-configPRIO_BITS 位，也就是左移 4 位。为什么要左移 4 位呢？前面我们说了，STM32 使用了 4 位作为优先级，而这 4 位是高 4 位，因此要左移 4 位才是真正的优先级。当然了也可以不用移位，直接将宏 configLIBRARY_LOWEST_INTERRUPT_PRIORITY 定义为 0XF0！不过这样看起来不直观。

宏 configKERNEL_INTERRUPT_PRIORITY 用来设置 PendSV 和滴答定时器的中断优先级，port.c 中有如下定义：

```
#define portNVIC_PENDSV_PRI     ( ( ( uint32_t ) configKERNEL_INTERRUPT_PRIORITY ) << 16UL )
#define portNVIC_SYSTICK_PRI    ( ( ( uint32_t ) configKERNEL_INTERRUPT_PRIORITY ) << 24UL )
```

可以看出，portNVIC_PENDSV_PRI 和 portNVIC_SYSTICK_PRI 都是使用了宏 configKERNEL_INTERRUPT_PRIORITY，为什么宏 portNVIC_PENDSV_PRI 是宏 configKERNEL_INTERRUPT_PRIORITY 左移 16 位呢？宏 portNVIC_SYSTICK_PRI 也同样是左移 24 位。4.1.4 小节讲过了，PendSV 和 SysTcik 的中断优先级设置是操作 0xE000_ED20 地址

的，这样一次写入的是个 32 位的数据， SysTick 和 PendSV 的优先级寄存器分别对应这个 32 位数据的最高 8 位和次高 8 位，不就是一个左移 16 位，一个左移 24 位了。

PendSV 和 SysTick 优先级是在哪里设置的呢？在函数 xPortStartScheduler()中设置，此函数在文件 port.c 中，函数如下：

```
BaseType_t xPortStartScheduler( void )
{
    configASSERT( configMAX_SYSCALL_INTERRUPT_PRIORITY );
    configASSERT( portCPUID != portCORTEX_M7_r0p1_ID );
    configASSERT( portCPUID != portCORTEX_M7_r0p0_ID );

    #if( configASSERT_DEFINED == 1 )
    {
        volatile uint32_t ulOriginalPriority;
        volatile uint8_t * const pucFirstUserPriorityRegister = ( uint8_t * )
                                        ( portNVIC_IP_REGISTERS_OFFSET_16 +
                                        portFIRST_USER_INTERRUPT_NUMBER );
        volatile uint8_t ucMaxPriorityValue;
        ulOriginalPriority = *pucFirstUserPriorityRegister;
        *pucFirstUserPriorityRegister = portMAX_8_BIT_VALUE;
        ucMaxPriorityValue = *pucFirstUserPriorityRegister;
        configASSERT( ucMaxPriorityValue == ( configKERNEL_INTERRUPT_PRIORITY &
                            ucMaxPriorityValue ) );
        ucMaxSysCallPriority = configMAX_SYSCALL_INTERRUPT_PRIORITY &
                            ucMaxPriorityValue;
        ulMaxPRIGROUPValue = portMAX_PRIGROUP_BITS;
        while( ( ucMaxPriorityValue & portTOP_BIT_OF_BYTE ) == portTOP_BIT_OF_BYTE )
        {
            ulMaxPRIGROUPValue--;
            ucMaxPriorityValue <<= ( uint8_t ) 0x01;
        }

        ulMaxPRIGROUPValue <<= portPRIGROUP_SHIFT;
        ulMaxPRIGROUPValue &= portPRIORITY_GROUP_MASK;

        *pucFirstUserPriorityRegister = ulOriginalPriority;
    }
    #endif /* conifgASSERT_DEFINED */

    portNVIC_SYSPRI2_REG |= portNVIC_PENDSV_PRI;        //设置 PendSV 中断优先级
    portNVIC_SYSPRI2_REG |= portNVIC_SYSTICK_PRI;       //设置 SysTick 中断优先级

    vPortSetupTimerInterrupt();
```

```
    uxCriticalNesting = 0;
    prvEnableVFP();
    *( portFPCCR ) |= portASPEN_AND_LSPEN_BITS;
    prvStartFirstTask();
    return 0;
}
```

上述代码中红色部分就是设置 PendSV 和 SysTick 优先级的，它们是直接向地址 portNVIC_SYSPRI2_REG 写入优先级数据，portNVIC_SYSPRI2_REG 是个宏，在文件 port.c 中由定义，如下：

```
#define portNVIC_SYSPRI2_REG        ( * ( ( volatile uint32_t * ) 0xe000ed20 ) )
```

可以看到宏 portNVIC_SYSPRI2_REG 就是地址 0XE000ED20！同时也可以看出在 FreeRTOS 中 PendSV 和 SysTick 的中断优先级都是最低的！

4.2.4 configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY

此宏用来设置 FreeRTOS 系统可管理的最大优先级，也就是我们在 4.1.5 小节中讲解 BASEPRI 寄存器说的那个阈值优先级，这个大家可以自由设置，这里我设置为了 5。也就是高于 5 的优先级(优先级数小于 5)不归 FreeRTOS 管理！

4.2.5 configMAX_SYSCALL_INTERRUPT_PRIORITY

此宏是 configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY 左移 4 位而来的，原因和宏 configKERNEL_INTERRUPT_PRIORITY 一样。此宏设置好以后，低于此优先级的中断可以安全的调用 FreeRTOS 的 API 函数，高于此优先级的中断 FreeRTOS 是不能禁止的，中断服务函数也不能调用 FreeRTOS 的 API 函数！

以 STM32 为例，有 16 个优先级，0 为最高优先级，15 为最低优先级，配置如下：

- configMAX_SYSCALL_INTERRUPT_PRIORITY==5
- configKERNEL_INTERRUPT_PRIORITY==15

结果如图 4.2.5.1 所示：

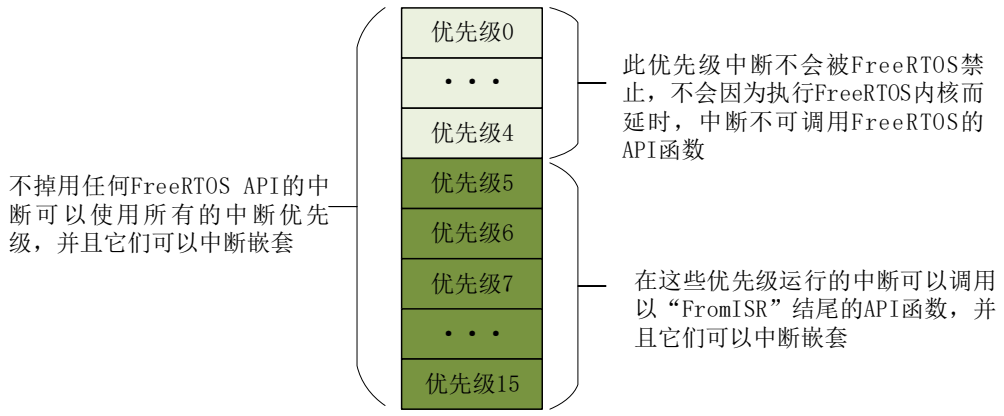


图 4.2.5.1 中断配置结果图

由于高于 configMAX_SYSCALL_INTERRUPT_PRIORITY 的优先级不会被 FreeRTOS 内核屏蔽，因此那些对实时性要求严格的任务就可以使用这些优先级，比如四轴飞行器中的壁障检测。

4.3 FreeRTOS 开关中断

FreeRTOS 开关中断函数为 portENABLE_INTERRUPTS ()和 portDISABLE_INTERRUPTS()，这两个函数其实是宏定义，在 portmacro.h 中有定义，如下：

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

可以看出开关中断实际上是通过函数 vPortSetBASEPRI(0)和 vPortRaiseBASEPRI()来实现的，这两个函数如下：

```
static portFORCE_INLINE void vPortSetBASEPRI( uint32_t ulBASEPRI )
{
    __asm
    {
        msr basepri, ulBASEPRI
    }
}
/*-----------------------------------------------------------*/

static portFORCE_INLINE void vPortRaiseBASEPRI( void )
{
uint32_t ulNewBASEPRI = configMAX_SYSCALL_INTERRUPT_PRIORITY;

    __asm
    {
        msr basepri, ulNewBASEPRI
        dsb
        isb
    }
}
```

函数 vPortSetBASEPRI()是向寄存器 BASEPRI 写入一个值，此值作为参数 ulBASEPRI 传递进来，portENABLE_INTERRUPTS()是开中断，它传递了个 0 给 vPortSetBASEPRI()，根据我们前面讲解 BASEPRI 寄存器可知，结果就是开中断。

函数 vPortRaiseBASEPRI() 是向寄存器 BASEPRI 写入宏 configMAX_SYSCALL_INTERRUPT_PRIORITY ，那么优先级低于 configMAX_SYSCALL_INTERRUPT_PRIORITY 的中断就会被屏蔽！

4.4 临界段代码

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，比如有的外设的初始化需要严格的时序，初始化过程中不能被打断。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。FreeRTOS 系统本身就有很多的临界段代码，这些代码都加了临界段代码保护，我们在写自己的用户程序的时候有些地方也需要添加临界段代码保护。

FreeRTOS 与临界段代码保护有关的函数有 4 个：taskENTER_CRITICAL()、taskEXIT_CRITICAL()、taskENTER_CRITICAL_FROM_ISR()和

taskEXIT_CRITICAL_FROM_ISR()，这四个函数其实是宏定义，在 task.h 文件中有定义。这四个函数的区别是前两个是任务级的临界段代码保护，后两个是中断级的临界段代码保护。

4.4.1 任务级临界段代码保护

taskENTER_CRITICAL()和 taskEXIT_CRITICAL()是任务级的临界代码保护，一个是进入临界段，一个是退出临界段，这两个函数是成对使用的，这函数的定义如下：

```
#define taskENTER_CRITICAL()        portENTER_CRITICAL()
#define taskEXIT_CRITICAL()         portEXIT_CRITICAL()
```

而 portENTER_CRITICAL()和 portEXIT_CRITICAL()也是宏定义，在文件 portmacro.h 中有定义，如下：

```
#define portENTER_CRITICAL()        vPortEnterCritical()
#define portEXIT_CRITICAL()         vPortExitCritical()
```

函数 vPortEnterCritical()和 vPortExitCritical()在文件 port.c 中，函数如下：

```
void vPortEnterCritical( void )
{
    portDISABLE_INTERRUPTS();
    uxCriticalNesting++;

    if( uxCriticalNesting == 1 )
    {
        configASSERT( ( portNVIC_INT_CTRL_REG & portVECTACTIVE_MASK ) == 0 );
    }
}

void vPortExitCritical( void )
{
    configASSERT( uxCriticalNesting );
    uxCriticalNesting--;
    if( uxCriticalNesting == 0 )
    {
        portENABLE_INTERRUPTS();
    }
}
```

可以看出在进入函数 vPortEnterCritical()以后会首先关闭中断，然后给变量 uxCriticalNesting 加一，uxCriticalNesting 是个全局变量，用来记录临界段嵌套次数的。函数 vPortExitCritical()是退出临界段调用的，函数每次将 uxCriticalNesting 减一，只有当 uxCriticalNesting 为 0 的时候才会调用函数 portENABLE_INTERRUPTS()使能中断。这样保证了在有多个临界段代码的时候不会因为某一个临界段代码的退出而打乱其他临界段的保护，只有所有的临界段代码都退出以后才会使能中断！

任务级临界代码保护使用方法如下：

```
void taskcritical_test(void)
{
    while(1)
```

```
    {
        taskENTER_CRITICAL();                                   (1)
        total_num+=0.01f;
        printf("total_num 的值为: %.4f\r\n",total_num);
        taskEXIT_CRITICAL();                                    (2)
        vTaskDelay(1000);
    }
}
```

(1)、进入临界区。

(2)、退出临界区。

(1)和(2)中间的代码就是临界区代码，注意临界区代码一定要精简！因为进入临界区会关闭中断，这样会导致优先级低于 configMAX_SYSCALL_INTERRUPT_PRIORITY 的中断得不到及时的响应！

4.4.2 中断级临界段代码保护

函数 taskENTER_CRITICAL_FROM_ISR()和 taskEXIT_CRITICAL_FROM_ISR()中断级别临界段代码保护，是用在中断服务程序中的，而且这个中断的优先级一定要低于 configMAX_SYSCALL_INTERRUPT_PRIORITY！原因前面已经说了。这两个函数在文件 task.h 中有如下定义：

```
#define taskENTER_CRITICAL_FROM_ISR()  portSET_INTERRUPT_MASK_FROM_ISR()
#define taskEXIT_CRITICAL_FROM_ISR( x ) portCLEAR_INTERRUPT_MASK_FROM_ISR( x )
```

接着找 portSET_INTERRUPT_MASK_FROM_ISR()和 portCLEAR_INTERRUPT_MASK_FROM_ISR()，这两个在文件 portmacro.h 中有如下定义：

```
#define portSET_INTERRUPT_MASK_FROM_ISR()          ulPortRaiseBASEPRI()
#define portCLEAR_INTERRUPT_MASK_FROM_ISR(x)       vPortSetBASEPRI(x)
```

vPortSetBASEPRI()前面已经讲解了，就是给 BASEPRI 寄存器中写入一个值。

函数 ulPortRaiseBASEPRI()在文件 portmacro.h 中定义的，如下：

```
static portFORCE_INLINE uint32_t ulPortRaiseBASEPRI( void )
{
uint32_t ulReturn, ulNewBASEPRI = configMAX_SYSCALL_INTERRUPT_PRIORITY;

    __asm
    {
        mrs ulReturn, basepri                   (1)
        msr basepri, ulNewBASEPRI               (2)
        dsb
        isb
    }

    return ulReturn;                            (3)
}
```

(1)、先读出 BASEPRI 的值，保存在 ulReturn 中。

(2)、将 configMAX_SYSCALL_INTERRUPT_PRIORITY 写入到寄存器 BASEPRI 中。

(3)、返回 ulReturn，退出临界区代码保护的时候要使用到此值！

中断级临界代码保护使用方法如下：

```
//定时器 3 中断服务函数
void TIM3_IRQHandler(void)
{
    HAL_TIM_IRQHandler(&TIM3_Handler);
}

static u32 total_num=0;
//回调函数，定时器中断服务函数调用
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    uint32_t status_value;
    if(htim==(&TIM3_Handler))
    {
        status_value=taskENTER_CRITICAL_FROM_ISR();        (1)
        total_num+=1;
        printf("float_num 的值为: %d\r\n",total_num);
        taskEXIT_CRITICAL_FROM_ISR(status_value);          (2)
    }
}
```

(1)、进入临界区。

(2)、退出临界区。

4.5 FreeRTOS 中断测试实验

4.5.1 实验程序设计

1、实验目的

上面我们讲了在 FreeRTOS 中优先级低于 configMAX_SYSCALL_INTERRUPT_PRIORITY 的中断会被屏蔽掉，高于的就不会，那么本节我们就写个简单的例程测试一下。使用两个定时器，一个优先级为 4，一个优先级为 5，两个定时器每隔 1s 通过串口输出一串字符串。然后在某个任务中关闭中断一段时间，查看两个定时器的输出情况。。

2、实验设计

本实验设计了两个任务 start_task()和 interrupt_task(), 这两个任务的任务功能如下：

start_task()：创建另外一个任务。

interrupt_task()：中断测试任务，任务中会调用 FreeRTOS 的关中断函数 portDISABLE_INTERRUPTS()来将中断关闭一段时间。

3、实验工程

FreeRTOS 实验 4-1 FreeRTOS 中断测试实验。

4、实验程序与分析

● **任务设置**

```
#define START_TASK_PRIO                 1          //任务优先级
#define START_STK_SIZE                  256        //任务堆栈大小
TaskHandle_t StartTask_Handler;                    //任务句柄
void start_task(void *pvParameters);               //任务函数

#define INTERRUPT_TASK_PRIO             2          //任务优先级
#define INTERRUPT_STK_SIZE              256        //任务堆栈大小
TaskHandle_t INTERRUPTTask_Handler;                //任务句柄
void interrupt_task(void *p_arg);                  //任务函数
```

- **main()函数**

```
int main(void)
{
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(360,25,2,8);   //设置时钟,180Mhz
    delay_init(180);                //初始化延时函数
    LED_Init();                     //初始化 LED
    uart_init(115200);              //初始化串口
    TIM3_Init(10000-1,9000-1);      //初始化定时器 3，定时周期 1S
    TIM5_Init(10000-1,9000-1);      //初始化定时器 5，定时周期 1S

    //创建开始任务
    xTaskCreate((TaskFunction_t     )start_task,            //任务函数
                (const char*        )"start_task",          //任务名称
                (uint16_t           )START_STK_SIZE,        //任务堆栈大小
                (void*              )NULL,                  //传递给任务函数的参数
                (UBaseType_t        )START_TASK_PRIO,       //任务优先级
                (TaskHandle_t*      )&StartTask_Handler);   //任务句柄
    vTaskStartScheduler();                                  //开启任务调度
}
```

- **任务函数**

```
//开始任务任务函数
void start_task(void *pvParameters)
{
    taskENTER_CRITICAL();           //进入临界区
    //创建中断测试任务
    xTaskCreate((TaskFunction_t     )interrupt_task,            //任务函数          (1)
                (const char*        )"interrupt_task",          //任务名称
                (uint16_t           )INTERRUPT_STK_SIZE,        //任务堆栈大小
                (void*              )NULL,                      //传递给任务函数的参数
                (UBaseType_t        )INTERRUPT_TASK_PRIO,       //任务优先级
                (TaskHandle_t*      )&INTERRUPTTask_Handler);   //任务句柄
    vTaskDelete(StartTask_Handler);     //删除开始任务
```

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

//中断测试任务函数
void interrupt_task(void *pvParameters)
{
    static u32 total_num=0;
    while(1)
    {
        total_num+=1;
        if(total_num==5)                                        (2)
        {
            printf("关闭中断.....\r\n");
            portDISABLE_INTERRUPTS();       //关闭中断          (3)
            delay_xms(5000);                //延时 5s           (4)
            printf("打开中断.....\r\n");    //打开中断
            portENABLE_INTERRUPTS();                            (5)
        }
        LED0=~LED0;
        vTaskDelay(1000);
    }
}
```

(1)、创建一个任务来执行开关中断的动作，任务函数为 interrupt_task()。

(2)、当任务 interrupt_task()运行 5 次以后关闭中断。

(3)、调用函数 portDISABLE_INTERRUPTS()关闭中断。优先级低于 configMAX_SYSCALL_INTERRUPT_PRIORITY 的中断都会被关闭，高于的不会受任何影响。

(4)、调用函数 delay_xms()延时 5S，此函数是对 delay_us()的简单封装，delay_xms()会用来模拟关闭中断一段时间，此函数不会引起任务调度！

(5)、调用函数 portENABLE_INTERRUPTS()重新打开中断。

● **中断初始化及处理过程**

```
TIM_HandleTypeDef TIM3_Handler;         //定时器句柄
TIM_HandleTypeDef TIM5_Handler;         //定时器句柄

//通用定时器 3 中断初始化
//arr：自动重装值。
//psc：时钟预分频数
//定时器溢出时间计算方法:Tout=((arr+1)*(psc+1))/Ft us.
//Ft=定时器工作频率,单位:Mhz
//这里使用的是定时器 3!(定时器 3 挂在 APB1 上，时钟为 HCLK/2)
void TIM3_Init(u16 arr,u16 psc)
{
    TIM3_Handler.Instance=TIM3;                         //通用定时器 3
    TIM3_Handler.Init.Prescaler=psc;                        //分频系数
```

```
    TIM3_Handler.Init.CounterMode=TIM_COUNTERMODE_UP;      //向上计数器
    TIM3_Handler.Init.Period=arr;                          //自动装载值
    TIM3_Handler.Init.ClockDivision=TIM_CLOCKDIVISION_DIV1; //时钟分频因子
    HAL_TIM_Base_Init(&TIM3_Handler);

    //使能定时器 3 和定时器 3 更新中断：TIM_IT_UPDATE
    HAL_TIM_Base_Start_IT(&TIM3_Handler);
}

//通用定时器 5 中断初始化
//arr：自动重装值(TIM2,TIM5 是 32 位的!!)
//psc：时钟预分频数
void TIM5_Init(u32 arr,u16 psc)
{
    TIM5_Handler.Instance=TIM5;                            //通用定时器 5
    TIM5_Handler.Init.Prescaler=psc;                       //分频系数
    TIM5_Handler.Init.CounterMode=TIM_COUNTERMODE_UP;      //向上计数器
    TIM5_Handler.Init.Period=arr;                          //自动装载值
    TIM5_Handler.Init.ClockDivision=TIM_CLOCKDIVISION_DIV1; //时钟分频因子
    HAL_TIM_Base_Init(&TIM5_Handler);

    //使能定时器 5 和定时器 5 更新中断：TIM_IT_UPDATE
    HAL_TIM_Base_Start_IT(&TIM5_Handler); }

//定时器底册驱动，开启时钟，设置中断优先级
//此函数会被 HAL_TIM_Base_Init()函数调用
void HAL_TIM_Base_MspInit(TIM_HandleTypeDef *htim)
{
    if(htim->Instance==TIM3)
    {
        __HAL_RCC_TIM3_CLK_ENABLE();            //使能 TIM3 时钟
        HAL_NVIC_SetPriority(TIM3_IRQn,4,0);    //抢占优先级 4，子优先级 0     (1)
        HAL_NVIC_EnableIRQ(TIM3_IRQn);          //开启 ITM3 中断
    }
    else if(htim->Instance==TIM5)
    {
        __HAL_RCC_TIM5_CLK_ENABLE();            //使能 TIM5 时钟
        HAL_NVIC_SetPriority(TIM5_IRQn,5,0);    //抢占优先级 5，子优先级 0      (2)
        HAL_NVIC_EnableIRQ(TIM5_IRQn);          //开启 ITM5 中断
    }
}
```

```
//定时器 3 中断服务函数
void TIM3_IRQHandler(void)
{
    HAL_TIM_IRQHandler(&TIM3_Handler);
}

//定时器 5 中断服务函数
void TIM5_IRQHandler(void)
{
    HAL_TIM_IRQHandler(&TIM5_Handler);
}

//回调函数，定时器中断服务函数调用
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    if(htim==(&TIM3_Handler))
    {
        printf("TIM3 输出.....\r\n");                    (3)
    }
    else if(htim==(&TIM5_Handler))
    {
        printf("TIM5 输出.....\r\n");                    (4)
    }
}
```

(1)、设置定时器 3 的抢占优先级为 4，高于 configMAX_SYSCALL_INTERRUPT_PRIORITY，因此在调用函数 portDISABLE_INTERRUPTS()关闭中断的时候定时器 3 是不会受影响的。

(2)、设置定时器 5 的抢占优先级为 5，等于 configMAX_SYSCALL_INTERRUPT_PRIORITY，因此在调用函数 portDISABLE_INTERRUPTS()关闭中断的时候定时器 5 中断肯定会被关闭的。

(3)和(4)、定时器 3 和定时 5 串口输出信息。

4.5.2 实验程序运行结果

编译并下载代码到开发板中，打开串口调试助手查看数据输出，结果如图 4.5.2.1 所示：

图 4.5.2.1 串口调试助手

从图 4.5.2.1 可以看出，一开始没有关闭中断，所以 TIM3 和 TIM5 都正常运行，红框所示部分。当任务 interrupt_task()运行了 5 次以后就关闭了中断，此时由于 TIM5 的中断优先级为 5，等于 configMAX_SYSCALL_INTERRUPT_PRIORITY，因此 TIM5 被关闭。但是，TIM3 的中断优先级高于 configMAX_SYSCALL_INTERRUPT_PRIORITY，不会被关闭，所以 TIM3 正常运行，绿框所示部分。中断关闭 5S 以后就会调用函数 portENABLE_INTERRUPTS()重新打开中断，重新打开中断以后 TIM5 恢复运行，蓝框所示部分。

第五章 FreeRTOS 任务基础知识

RTOS 系统的核心就是任务管理，FreeRTOS 也不例外，而且大多数学习 RTOS 系统的工程师或者学生主要就是为了使用 RTOS 的多任务处理功能，初步上手 RTOS 系统首先必须掌握的也是任务的创建、删除、挂起和恢复等操作，由此可见任务管理的重要性。由于任务相关的知识很多，所以接下来我们将用几章的内容来讲解 FreeRTOS 的任务。本章先学习一下 FreeRTOS 的任务基础知识，本章是后面学习的基础，所以一定要掌握本章关于 FreeRTOS 任务管理的基础知识，本章分为如下几部分：

5.1 什么是多任务系统
5.2 FreeRTOS 任务与协程
5.3 初次使用
5.3 任务状态
5.4 任务优先级
5.5 任务实现
5.6 任务控制块
5.7 任务堆栈

5.1 什么是多任务系统？

回想一下我们以前在使用 51、AVR、STM32 单片机裸机(未使用系统)的时候一般都是在 main 函数里面用 while(1)做一个大循环来完成所有的处理，即应用程序是一个无限的循环，循环中调用相应的函数完成所需的处理。有时候我们也需要中断中完成一些处理。相对于多任务系统而言，这个就是单任务系统，也称作前后台系统，中断服务函数作为前台程序，大循环 while(1)作为后台程序，如图 5.1.1 所示：

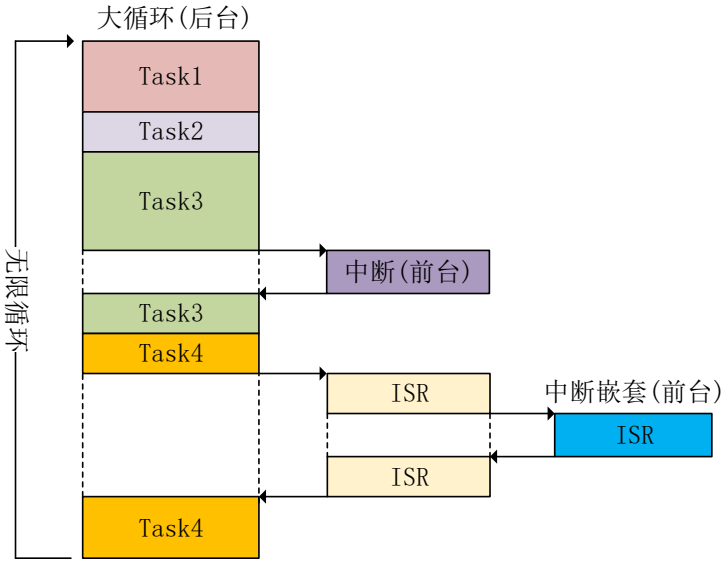


图 5.1.1 前后台系统

前后台系统的实时性差，前后台系统各个任务(应用程序)都是排队等着轮流执行，不管你这个程序现在有多紧急，没轮到你就只能等着！相当于所有任务(应用程序)的优先级都是一样的。但是前后台系统简单啊，资源消耗也少啊！在稍微大一点的嵌入式应用中前后台系统就明显力不从心了，此时就需要多任务系统出马了。

多任务系统会把一个大问题(应用)“分而治之”，把大问题划分成很多个小问题，逐步的把小问题解决掉，大问题也就随之解决了，这些小问题可以单独的作为一个小任务来处理。这些小任务是并发处理的，注意，并不是说同一时刻一起执行很多个任务，而是由于每个任务执行的时间很短，导致看起来像是同一时刻执行了很多个任务一样。多个任务带来了一个新的问题，究竟哪个任务先运行，哪个任务后运行呢？完成这个功能的东西在 RTOS 系统中叫做任务调度器。不同的系统其任务调度器的实现方法也不同，比如 FreeRTOS 是一个抢占式的实时多任务系统，那么其任务调度器也是抢占式的，运行过程如图 5.1.2 所示：

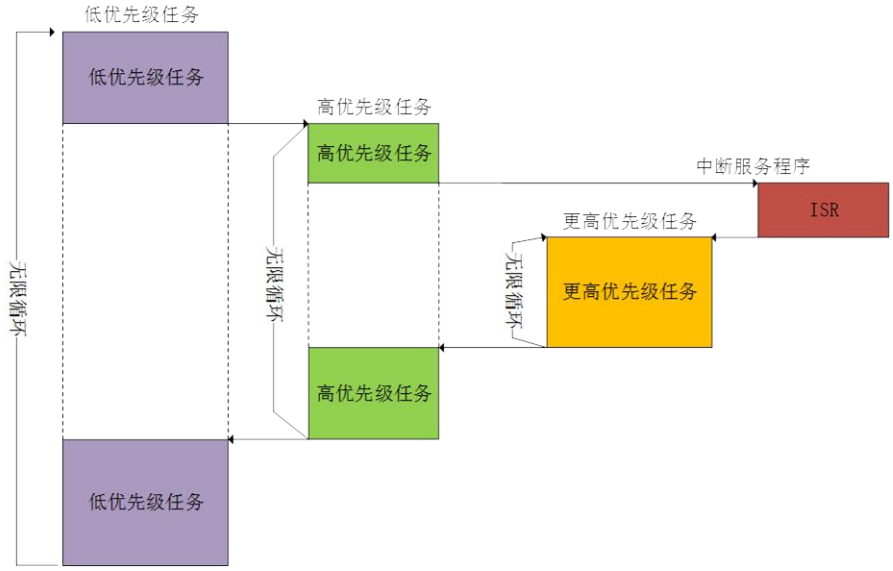


图 5.1.2 抢占式多任务系统

在图 5.1.2 中，高优先级的任务可以打断低优先级任务的运行而取得 CPU 的使用权，这样就保证了那些紧急任务的运行。这样我们就可以为那些对实时性要求高的任务设置一个很高的优先级，比如自动驾驶中的障碍物检测任务等。高优先级的任务执行完成以后重新把 CPU 的使用权归还给低优先级的任务，这个就是抢占式多任务系统的基本原理。

5.2 FreeRTOS 任务与协程

再 FreeRTOS 中应用既可以使用任务，也可以使用协程(Co-Routine)，或者两者混合使用。但是任务和协程使用不同的 API 函数，因此不能通过队列(或信号量)将数据从任务发送给协程，反之亦然。协程是为那些资源很少的 MCU 准备的，其开销很小，但是 FreeRTOS 官方已经不打算再更新协程了，所以本教程只讲解任务。

5.2.1 任务(Task)的特性

在使用 RTOS 的时候一个实时应用可以作为一个独立的任务。每个任务都有自己的运行环境，不依赖于系统中其他的任务或者 RTOS 调度器。任何一个时间点只能有一个任务运行，具体运行哪个任务是由 RTOS 调度器来决定的，RTOS 调度器因此就会重复的开启、关闭每个任务。任务不需要了解 RTOS 调度器的具体行为，RTOS 调度器的职责是确保当一个任务开始执行的时候其上下文环境(寄存器值，堆栈内容等)和任务上一次退出的时候相同。为了做到这一点，每个任务都必须有个堆栈，当任务切换的时候将上下文环境保存在堆栈中，这样当任务再次执行的时候就可以从堆栈中取出上下文环境，任务恢复运行。

任务特性：

1、简单。
2、没有使用限制。
3、支持抢占
4、支持优先级
5、每个任务都拥有堆栈导致了 RAM 使用量增大。
6、如果使用抢占的话的必须仔细的考虑重入的问题。

5.2.2 协程(Co-routine)的特性

协程是为那些资源很少的 MCU 而做的，但是随着 MCU 的飞速发展，性能越来越强大，现在协程几乎很少用到了！但是 FreeRTOS 目前还没有把协程移除的计划，但是 FreeRTOS 是绝对不会再更新和维护协程了，因此协程大家了解一下就行了。在概念上协程和任务是相似的，但是有如下根本上的不同：

1、堆栈使用

所有的协程使用同一个堆栈(如果是任务的话每个任务都有自己的堆栈)，这样就比使用任务消耗更少的 RAM。

2、调度器和优先级

协程使用合作式的调度器，但是可以在使用抢占式的调度器中使用协程。

3、宏实现

协程是通过宏定义来实现的。

4、使用限制

为了降低对 RAM 的消耗做了很多的限制。

5.3 任务状态

FreeRTOS 中的任务永远处于下面几个状态中的某一个：

- **运行态**

当一个任务正在运行时，那么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的话那么不管在任何时刻永远都只有一个任务处于运行态。

- **就绪态**

处于就绪态的任务是那些已经准备就绪(这些任务没有被阻塞或者挂起)，可以运行的任务，但是处于就绪态的任务还没有运行，因为有一个同优先级或者更高优先级的任务正在运行！

- **阻塞态**

如果一个任务当前正在等待某个外部事件的话就说它处于阻塞态，比如说如果某个任务调用了函数 vTaskDelay()的话就会进入阻塞态，直到延时周期完成。任务在等待队列、信号量、事件组、通知或互斥信号量的时候也会进入阻塞态。任务进入阻塞态会有一个超时时间，当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来临！

- **挂起态**

像阻塞态一样，任务进入挂起态以后也不能被调度器调用进入运行态，但是进入挂起态的任务没有超时时间。任务进入和退出挂起态通过调用函数 vTaskSuspend()和 xTaskResume()。

任务状态之间的转换如图 5.4.1 所示：

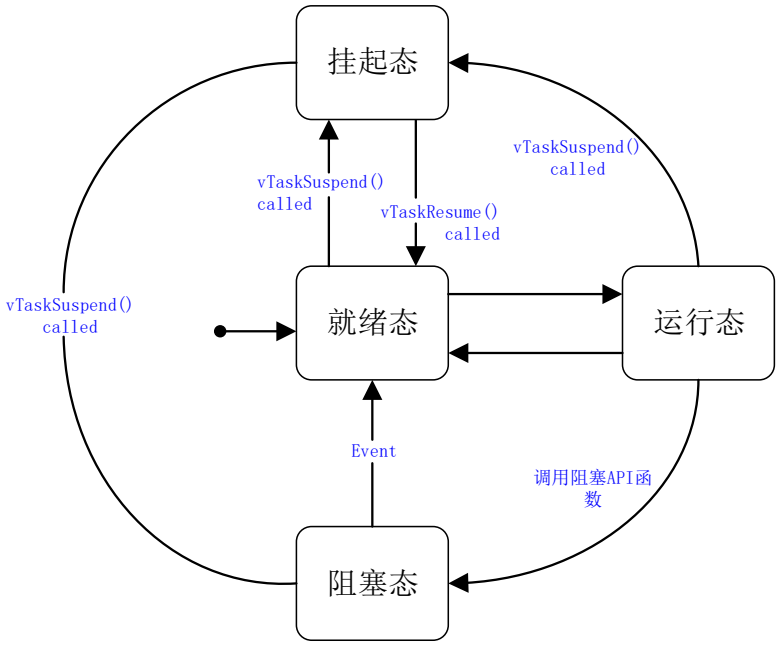


图 5.4.1 任务状态之间的转换

5.4 任务优先级

每个任务都可以分配一个从 0~(configMAX_PRIORITIES-1) 的优先级，configMAX_PRIORITIES 在文件 FreeRTOSConfig.h 中有定义，前面我们讲解 FreeRTOS 系统配置的时候已经讲过了。如果所使用的硬件平台支持类似计算前导零这样的指令(可以通过该指令选择下一个要运行的任务，Cortex-M 处理器是支持该指令的)，并且宏 configUSE_PORT_OPTIMISED_TASK_SELECTION 也设置为了 1，那么宏 configMAX_PRIORITIES 不能超过 32！也就是优先级不能超过 32 级。其他情况下宏 configMAX_PRIORITIES 可以为任意值，但是考虑到 RAM 的消耗，宏 configMAX_PRIORITIES 最好设置为一个满足应用的最小值。

优先级数字越低表示任务的优先级越低，0 的优先级最低，configMAX_PRIORITIES-1 的优先级最高。空闲任务的优先级最低，为 0。

FreeRTOS 调度器确保处于就绪态或运行态的高优先级的任务获取处理器使用权，换句话说就是处于就绪态的最高优先级的任务才会运行。当宏 configUSE_TIME_SLICING 定义为 1 的时候多个任务可以共用一个优先级，数量不限。默认情况下宏 configUSE_TIME_SLICING 在文件 FreeRTOS.h 中已经定义为 1。此时处于就绪态的优先级相同的任务就会使用时间片轮转调度器获取运行时间。

5.5 任务实现

在使用 FreeRTOS 的过程中，我们要使用函数 xTaskCreate()或 xTaskCreateStatic()来创建任务，这两个函数的第一个参数 pxTaskCode，就是这个任务的任务函数。什么是任务函数？任务函数就是完成本任务工作的函数。我这个任务要干嘛？要做什么？要完成什么样的功能都是在这个任务函数中实现的。 比如我要做个任务，这个任务要点个流水灯，那么这个流水灯的程序就是任务函数中实现的。FreeRTOS 官方给出的任务函数模板如下：

```
void vATaskFunction(void *pvParameters)                    (1)
{
```

```
    for( ; ; )                                          (2)
    {
        --任务应用程序--                                (3)
        vTaskDelay();                                   (4)
    }
    /*不能从任务函数中返回或者退出，从任务函数中返回或退出的话就会调用
configASSERT()，前提是你定义了 configASSERT()。如果一定要从任务函数中退出的话那一定
要调用函数 vTaskDelete(NULL)来删除此任务。*/

    vTaskDelete(NULL);                                  (5)

}
```

(1)、任务函数本质也是函数，所以肯定有任务名什么的，不过这里我们要注意：任务函数的返回类型一定要为 void 类型，也就是无返回值，而且任务的参数也是 void 指针类型的！任务函数名可以根据实际情况定义。

(2)、任务的具体执行过程是一个大循环，for(; ;)就代表一个循环，作用和 while(1)一样，笔者习惯用 while(1)。

(3)、循环里面就是真正的任务代码了，此任务具体要干的活就在这里实现！

(4)、FreeRTOS 的延时函数，此处不一定要用延时函数，其他只要能让 FreeRTOS 发生任务切换的 API 函数都可以，比如请求信号量、队列等，甚至直接调用任务调度器。只不过最常用的就是 FreeRTOS 的延时函数。

(5)、任务函数一般不允许跳出循环，如果一定要跳出循环的话在跳出循环以后一定要调用函数 vTaskDelete(NULL)删除此任务！

FreeRTOS 的任务函数和 uC/OS 的任务函数模式基本相同的，不止 FreeRTOS，其他 RTOS 的任务函数基本也是这种方式的。

5.6 任务控制块

FreeRTOS 的每个任务都有一些属性需要存储，FreeRTOS 把这些属性集合到一起用一个结构体来表示，这个结构体叫做任务控制块：TCB_t，在使用函数 xTaskCreate()创建任务的时候就会自动的给每个任务分配一个任务控制块。在老版本的 FreeRTOS 中任务控制块叫做 tskTCB，新版本重命名为 TCB_t，但是本质上还是 tskTCB，本教程后面提到任务控制块的话均用 TCB_t 表示，此结构体在文件 tasks.c 中有定义，如下：

```
typedef struct tskTaskControlBlock
{
    volatile StackType_t    *pxTopOfStack;          //任务堆栈栈顶
    #if ( portUSING_MPU_WRAPPERS == 1 )
        xMPU_SETTINGS xMPUSettings;                 //MPU 相关设置
    #endif

    ListItem_t              xStateListItem;         //状态列表项
    ListItem_t              xEventListItem;         //事件列表项
    UBaseType_t             uxPriority;             //任务优先级
    StackType_t             *pxStack;               //任务堆栈起始地址
```

```
    char                    pcTaskName[ configMAX_TASK_NAME_LEN ];//任务名字

    #if ( portSTACK_GROWTH > 0 )
        StackType_t         *pxEndOfStack;          //任务堆栈栈底
    #endif

    #if ( portCRITICAL_NESTING_IN_TCB == 1 )
        UBaseType_t         uxCriticalNesting;      //临界区嵌套深度
    #endif

    #if ( configUSE_TRACE_FACILITY == 1 )       //trace 或到 debug 的时候用到
        UBaseType_t         uxTCBNumber;
        UBaseType_t         uxTaskNumber;
    #endif

    #if ( configUSE_MUTEXES == 1 )
        UBaseType_t         uxBasePriority;         //任务基础优先级,优先级反转的时候用到
        UBaseType_t         uxMutexesHeld;          //任务获取到的互斥信号量个数
    #endif

    #if ( configUSE_APPLICATION_TASK_TAG == 1 )
        TaskHookFunction_t pxTaskTag;
    #endif

    #if( configNUM_THREAD_LOCAL_STORAGE_POINTERS > 0 ) //与本地存储有关
        void
*pvThreadLocalStoragePointers[ configNUM_THREAD_LOCAL_STORAGE_POINTERS ];
    #endif

    #if( configGENERATE_RUN_TIME_STATS == 1 )
        uint32_t        ulRunTimeCounter;           //用来记录任务运行总时间
    #endif

    #if ( configUSE_NEWLIB_REENTRANT == 1 )
        struct      _reent xNewLib_reent;           //定义一个 newlib 结构体变量
    #endif

    #if( configUSE_TASK_NOTIFICATIONS == 1 )//任务通知相关变量
        volatile uint32_t ulNotifiedValue;          //任务通知值
        volatile uint8_t ucNotifyState;             //任务通知状态
    #endif

    #if( tskSTATIC_AND_DYNAMIC_ALLOCATION_POSSIBLE != 0 )
```

```
        //用来标记任务是动态创建的还是静态创建的，如果是静态创建的此变量就为 pdTURE，
        //如果是动态创建的就为 pdFALSE
          uint8_t    ucStaticallyAllocated;
    #endif

    #if( INCLUDE_xTaskAbortDelay == 1 )
          uint8_t ucDelayAborted;
    #endif

} tskTCB;

//新版本的 FreeRTOS 任务控制块重命名为 TCB_t，但是本质上还是 tskTCB，主要是为了兼容
//旧版本的应用。
typedef    tskTCB    TCB_t;
```

可以看出来 FreeRTOS 的任务控制块中的成员变量相比 uC/OSIII 要少很多，而且大多数与裁剪有关，当不使用某些功能的时候与其相关的变量就不参与编译，任务控制块大小就会进一步的减小。

5.7 任务堆栈

FreeRTOS 之所以能正确的恢复一个任务的运行就是因为有任务堆栈在保驾护航，任务调度器在进行任务切换的时候会将当前任务的现场(CPU 寄存器值等)保存在此任务的任务堆栈中，等到此任务下次运行的时候就会先用堆栈中保存的值来恢复现场，恢复现场以后任务就会接着从上次中断的地方开始运行。

创建任务的时候需要给任务指定堆栈，如果使用的函数 xTaskCreate()创建任务(动态方法)的话那么任务堆栈就会由函数 xTaskCreate()自动创建，后面分析 xTaskCreate()的时候会讲解。如果使用函数 xTaskCreateStatic()创建任务(静态方法)的话就需要程序员自行定义任务堆栈，然后堆栈首地址作为函数的参数 puxStackBuffer 传递给函数，如下：

```
TaskHandle_t xTaskCreateStatic(  TaskFunction_t      pxTaskCode,
                                 const char * const  pcName,
                                 const uint32_t      ulStackDepth,
                                 void * const        pvParameters,
                                 UBaseType_t         uxPriority,
                                 StackType_t *       const puxStackBuffer,        (1)
                                 StaticTask_t *      const pxTaskBuffer       )
```

(1)、任务堆栈，需要用户定义，然后将堆栈首地址传递给这个参数。

堆栈大小：

我们不管是使用函数 xTaskCreate()还是 xTaskCreateStatic()创建任务都需要指定任务堆栈大小。任务堆栈的数据类型为 StackType_t，StackType_t 本质上是 uint32_t，在 portmacro.h 中有定义，如下：

```
#define   portSTACK_TYPE      uint32_t
#define   portBASE_TYPE       long

typedef   portSTACK_TYPE      StackType_t;
```

```
typedef    long              BaseType_t;
typedef    unsigned long     UBaseType_t;
```

可以看出 StackType_t 类型的变量为 4 个字节，那么任务的实际堆栈大小就应该是我们所定义的 4 倍

第六章 FreeRTOS 任务相关 API 函数

上一章我们学习了 FreeRTOS 的任务基础知识，本章就正式学习如何使用 FreeRTOS 中有关任务的 API 函数。本来本章想讲解 FreeRTOS 的任务原理知识的，但是很多初学者还没使用过 FreeRTOS，甚至其他的 RTOS 系统都没有使用过，所以一上来就是苦涩的原理很可能会吓跑一大批初学者。所以本章做了调整，先学习怎么用，先知其然，后面在知其所以然。使用过以后再学习原理、看源码就会轻松很多。本章分为如下几部分：

6.1 任务创建和删除 API 函数
6.2 任务创建和删除实验(动态方法)
6.3 任务创建和删除实验(静态方法)
6.4 任务挂起和恢复 API 函数
6.5 任务挂起和恢复实验

6.1 任务创建和删除 API 函数

FreeRTOS 最基本的功能就是任务管理，而任务管理最基本的操作就是创建和删除任务，FreeRTOS 的任务创建和删除 API 函数如表 6.1.1.1 所示：

函数	描述
xTaskCreate()	使用动态的方法创建一个任务。
xTaskCreateStatic()	使用静态的方法创建一个任务。
xTaskCreateRestricted()	创建一个使用 MPU 进行限制的任务，相关内存使用动态内存分配。
vTaskDelete()	删除一个任务。

表 6.1.1.1 任务创建和删除 API 函数

1、函数 xTaxkCreate()

此函数用来创建一个任务，任务需要 RAM 来保存与任务有关的状态信息(任务控制块)，任务也需要一定的 RAM 来作为任务堆栈。如果使用函数 xTaskCreate()来创建任务的话那么这些所需的 RAM 就会自动的从 FreeRTOS 的堆中分配，因此必须提供内存管理文件，默认我们使用 heap_4.c 这个内存管理文件，而且宏 configSUPPORT_DYNAMIC_ALLOCATION 必须为 1。如果使用函数 xTaskCreateStatic()创建的话这些 RAM 就需要用户来提供了。新创建的任务默认就是就绪态的，如果当前没有比它更高优先级的任务运行那么此任务就会立即进入运行态开始运行，不管在任务调度器启动前还是启动后，都可以创建任务。此函数也是我们以后经常用到的，本教程所有例程均用此函数来创建任务，函数原型如下：

```
BaseType_t xTaskCreate(     TaskFunction_t          pxTaskCode,
                            const char * const      pcName,
                            const uint16_t          usStackDepth,
                            void * const            pvParameters,
                            UBaseType_t             uxPriority,
                            TaskHandle_t * const    pxCreatedTask )
```

参数：

pxTaskCode： 任务函数。

pcName： 任务名字，一般用于追踪和调试，任务名字长度不能超过。configMAX_TASK_NAME_LEN。

usStackDepth： 任务堆栈大小，注意实际申请到的堆栈是 usStackDepth 的 4 倍。其中空闲任务的任务堆栈大小为 configMINIMAL_STACK_SIZE。

pvParameters: 传递给任务函数的参数。

uxPriotiry: 任务优先级，范围 0~ configMAX_PRIORITIES-1。

pxCreatedTask: 任务句柄，任务创建成功以后会返回此任务的任务句柄，这个句柄其实就是任务的任务堆栈。此参数就用来保存这个任务句柄。其他 API 函数可能会使用到这个句柄。

返回值：

pdPASS: 任务创建成功。

errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY： 任务创建失败，因为堆内存不足！

2、函数 xTaskCreateStatic()

此函数和 xTaskCreate()的功能相同，也是用来创建任务的，但是使用此函数创建的任务所需的 RAM 需要用用户来提供。如果要使用此函数的话需要将宏 configSUPPORT_STATIC_ALLOCATION 定义为 1。函数原型如下：

```
TaskHandle_t xTaskCreateStatic(  TaskFunction_t          pxTaskCode,
                                 const char * const      pcName,
                                 const uint32_t          ulStackDepth,
                                 void * const            pvParameters,
                                 UBaseType_t             uxPriority,
                                 StackType_t * const     puxStackBuffer,
                                 StaticTask_t * const    pxTaskBuffer )
```

参数：

pxTaskCode： 任务函数。
pcName： 任务名字，一般用于追踪和调试，任务名字长度不能超过。configMAX_TASK_NAME_LEN。
usStackDepth： 任务堆栈大小，由于本函数是静态方法创建任务，所以任务堆栈由用户给出，一般是个数组，此参数就是这个数组的大小。
pvParameters: 传递给任务函数的参数。
uxPriotiry: 任务优先级，范围 0~ configMAX_PRIORITIES-1。
puxStackBuffer: 任务堆栈，一般为数组，数组类型要为 StackType_t 类型。
pxTaskBuffer: 任务控制块。

返回值：

NULL： 任务创建失败，puxStackBuffer 或 pxTaskBuffer 为 NULL 的时候会导致这个错误的发生。
其他值: 任务创建成功，返回任务的任务句柄。

3、函数 xTaskCreateRestricted()

此函数也是用来创建任务的，只不过此函数要求所使用的 MCU 有 MPU(内存保护单元)，用此函数创建的任务会受到 MPU 的保护。其他的功能和函数 xTaxkCreate()一样。

```
BaseType_t xTaskCreateRestricted( const TaskParameters_t * const   pxTaskDefinition,
                                  TaskHandle_t *                   pxCreatedTask )
```

参数：

pxTaskDefinition: 指向一个结构体 TaskParameters_t，这个结构体描述了任务的任务函数、堆栈大小、优先级等。此结构体在文件 task.h 中有定义。
pxCreatedTask: 任务句柄。

返回值：

pdPASS: 任务创建成功。
其他值: 任务未创建成功，很有可能是因为 FreeRTOS 的堆太小了。

4、函数 vTaskDelete()

删除一个用函数 xTaskCreate()或者 xTaskCreateStatic()创建的任务，被删除了的任务不再存在，也就是说再也不会进入运行态。任务被删除以后就不能再使用此任务的句柄！如果此任务是使用动态方法创建的，也就是使用函数 xTaskCreate()创建的，那么在此任务被删除以后此任务之前申请的堆栈和控制块内存会在空闲任务中被释放掉，因此当调用函数 vTaskDelete()删除任务以后必须给空闲任务一定的运行时间。

只有那些由内核分配给任务的内存才会在任务被删除以后自动的释放掉，用户分配给任务的内存需要用户自行释放掉，比如某个任务中用户调用函数 pvPortMalloc()分配了 500 字节的内存，那么在此任务被删除以后用户也必须调用函数 vPortFree()将这 500 字节的内存释放掉，否则会导致内存泄露。此函数原型如下：

```
vTaskDelete( TaskHandle_t    xTaskToDelete )
```

参数：

xTaskToDelete:　　要删除的任务的任务句柄。

返回值:

无

6.2 任务创建和删除实验(动态方法)

6.2.1 实验程序设计

1、实验目的

上一小节讲解了 FreeRTOS 的任务创建和删除的 API 函数，本小节就来学习如何使用这些 API 函数，本小节学习 xTaskCreate()和 vTaskDelete()这两个函数的使用

2、实验设计

本实验设计三个任务：start_task、task1_task 和 task2_task ，这三个任务的任务功能如下：

start_task：用来创建其他两个任务。

task1_task ：当此任务运行 5 此以后就会调用函数 vTaskDelete()删除任务 task2_task，此任务也会控制 LED0 的闪烁，并且周期性的刷新 LCD 指定区域的背景颜色。

task2_task ：此任务普通的应用任务，此任务也会控制 LED1 的闪烁，并且周期性的刷新 LCD 指定区域的背景颜色。

3、实验工程

FreeRTOS 实验 6-1 FreeRTOS 任务创建和删除实验(动态方法)。

4、实验程序与分析

● **任务设置**

```
#define START_TASK_PRIO          1          //任务优先级                    (1)
#define START_STK_SIZE           128        //任务堆栈大小                  (2)
TaskHandle_t StartTask_Handler;             //任务句柄                      (3)
void start_task(void *pvParameters);        //任务函数                      (4)

#define TASK1_TASK_PRIO          2          //任务优先级
```

```
#define TASK1_STK_SIZE              128         //任务堆栈大小
TaskHandle_t Task1Task_Handler;                 //任务句柄
void task1_task(void *pvParameters);            //任务函数

#define TASK2_TASK_PRIO             3           //任务优先级
#define TASK2_STK_SIZE              128         //任务堆栈大小
TaskHandle_t Task2Task_Handler;                 //任务句柄
void task2_task(void *pvParameters);            //任务函数

//LCD 刷屏时使用的颜色
int lcd_discolor[14]={  WHITE,      BLACK,      BLUE,       BRED,
                        GRED,       GBLUE,      RED,        MAGENTA,
                        GREEN,      CYAN,       YELLOW,     BROWN,
                        BRRED,      GRAY };
```

(1) 、start_task 任务的任务优先级，此处用宏来表示，以后所有的任务优先级都用宏来表示。创建任务设置优先级的时候就用这个宏，当然了也可以直接在创建任务的时候指定任务优先级。

(2)、start_task 任务的任务堆栈大小。

(3)、start_task 任务的任务句柄。

(4)、start_task 任务的任务函数声明。

● **main()函数**

```
int main(void)
{
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(360,25,2,8);   //设置时钟,180Mhz
    delay_init(180);                //初始化延时函数
    uart_init(115200);              //初始化串口
    LED_Init();                     //初始化 LED
    KEY_Init();                     //初始化按键
    SDRAM_Init();                   //初始化 SDRAM
    LCD_Init();                     //初始化 LCD

    POINT_COLOR = RED;
    LCD_ShowString(30,10,200,16,16,"Apollo STM32F4/F7");
    LCD_ShowString(30,30,200,16,16,"FreeRTOS Examp 5-1");
    LCD_ShowString(30,50,200,16,16,"Task Creat and Del");
    LCD_ShowString(30,70,200,16,16,"ATOM@ALIENTEK");
    LCD_ShowString(30,90,200,16,16,"2016/9/24");

    //创建开始任务
    xTaskCreate((TaskFunction_t     )start_task,            //任务函数      (1)
                (const char*        )"start_task",          //任务名称
                (uint16_t           )START_STK_SIZE,        //任务堆栈大小
```

```
                (void*          )NULL,                  //传递给任务函数的参数
                (UBaseType_t    )START_TASK_PRIO,       //任务优先级
                (TaskHandle_t*  )&StartTask_Handler);   //任务句柄
    vTaskStartScheduler();                              //开启任务调度      (2)
}
```

(1)、调用函数 xTaskCreate()创建 tart_task 任务，函数中的各个参数就是上面的任务设置中定义的，其他任务的创建也用这种方法。

(2)、调用函数 vTaskStartScheduler()开启 FreeRTOS 的任务调度器，FreeRTOS 开始运行。

● **任务函数**

```
//开始任务任务函数
void start_task(void *pvParameters)                                 (1)
{
    taskENTER_CRITICAL();           //进入临界区
    //创建 TASK1 任务
    xTaskCreate((TaskFunction_t     )task1_task,
                (const char*        )"task1_task",
                (uint16_t           )TASK1_STK_SIZE,
                (void*              )NULL,
                (UBaseType_t        )TASK1_TASK_PRIO,
                (TaskHandle_t*      )&Task1Task_Handler);
    //创建 TASK2 任务
    xTaskCreate((TaskFunction_t     )task2_task,
                (const char*        )"task2_task",
                (uint16_t           )TASK2_STK_SIZE,
                (void*              )NULL,
                (UBaseType_t        )TASK2_TASK_PRIO,
                (TaskHandle_t*      )&Task2Task_Handler);
    vTaskDelete(StartTask_Handler); //删除开始任务                   (2)
    taskEXIT_CRITICAL();            //退出临界区
}

//task1 任务函数
void task1_task(void *pvParameters)                                 (3)
{
    u8 task1_num=0;

    POINT_COLOR = BLACK;

    LCD_DrawRectangle(5,110,115,314);  //画一个矩形
    LCD_DrawLine(5,130,115,130);       //画线
    POINT_COLOR = BLUE;
    LCD_ShowString(6,111,110,16,16,"Task1 Run:000");
    while(1)
```

```
    {
        task1_num++; //任务执 1 行次数加 1  注意 task1_num1 加到 255 的时候会清零!!
        LED0=!LED0;
        printf("任务 1 已经执行：%d 次\r\n",task1_num);
        if(task1_num==5)
        {
            vTaskDelete(Task2Task_Handler);//任务 1 执行 5 次删除任务 2      (4)
            printf("任务 1 删除了任务 2!\r\n");
        }
        LCD_Fill(6,131,114,313,lcd_discolor[task1_num%14]); //填充区域
        LCD_ShowxNum(86,111,task1_num,3,16,0x80); //显示任务执行次数
        vTaskDelay(1000);                                  //延时 1s，也就是 1000 个时钟节拍
    }
}

//task2 任务函数
void task2_task(void *pvParameters)                                    (5)
{
    u8 task2_num=0;

    POINT_COLOR = BLACK;

    LCD_DrawRectangle(125,110,234,314);     //画一个矩形
    LCD_DrawLine(125,130,234,130);          //画线
    POINT_COLOR = BLUE;
    LCD_ShowString(126,111,110,16,16,"Task2 Run:000");
    while(1)
    {
        task2_num++; //任务 2 执行次数加 1  注意 task1_num2 加到 255 的时候会清零!!
        LED1=!LED1;
        printf("任务 2 已经执行：%d 次\r\n",task2_num);
        LCD_ShowxNum(206,111,task2_num,3,16,0x80);  //显示任务执行次数
        LCD_Fill(126,131,233,313,lcd_discolor[13-task2_num%14]); //填充区域
        vTaskDelay(1000);           //延时 1s，也就是 1000 个时钟节拍
    }
}
```

(1)、start_task 任务的任务函数，在此任务函数中我们创建了另外两个任务 task1_task 和 task2_task。start_task 任务的职责就是用来创建其他的任务或者信号量、消息队列等的，当创建完成以后就可以删除掉 start_task 任务。

(2)、删除 start_task 任务，注意函数 vTaskDelete()的参数就是 start_task 任务的任务句柄 StartTask_Handler。

(3)、task1_task 任务函数(任务 1)，任务比较简单，每隔 1 秒钟 task1_num 加一并且 LED0 反转，串口输出任务运行的次数，其实就是 task1_num 的值。当 task1_task 运行 5 次以后就调

用函数 vTaskDelete()删除任务 task2_task。

(4)、任务 task1_task 运行了 5 次，调用函数 vTaskDelete()删除任务 task2_task。

(5)、task2_task 任务函数(任务 2)，和 task1_task 差不多。

简单的总结分析一下此例程的流程，因为这是我们使用 FreeRTOS 写的第一个程序，很多习惯是我们后面要用到的。比如使用任务宏定义任务优先级，堆栈大小等，一般有关一个任务的东西我们的放到一起，比如任务堆栈、任务句柄、任务函数声明等，这样方便修改。这些东西可以放到一个.h 头文件里面去，只是例程里面任务数比较少，所以就直接放到 main.c 文件里面了，要是工程比较大的话最好做一个专用的头文件来管理。

在 main 函数中一开始肯定是初始化各种硬件外设，初始化完外设以后调用函数 xTaskCreate()创建一个开始任务，注意创建开始任务是在调用函数 vTaskStartScheduler()开启任务调度器之前，这样当后面开启任务调度器以后就会直接运行开始任务了。其他任务的创建就放到开始任务的任务函数中，由于开始任务的职责就是创建其他应用任务和信号量、队列等这些内核对象的，所以它只需要执行一次，当这些东西创建完成以后就可以删除掉开始任务了。看过我们的 uC/OS 教程的话就会发现这个过程和 uC/OS 里面一样的。

6.2.2 程序运行结果分析

编译程序并下载到开发板中，查看任务 1 和任务 2 的运行情况，下载完成以后以后 LCD 显示如图 6.2.2.1 所示：

图 6.2.2.1 LCD 默认界面

图中左边的框为任务 1 的运行区域，右边的框为任务 2 的运行区域，可以看出任务 2 运行了 5 次就停止了，而任务 1 运行了 17 次了。打开串口调试助手，显示如图 6.2.2.2 所示：

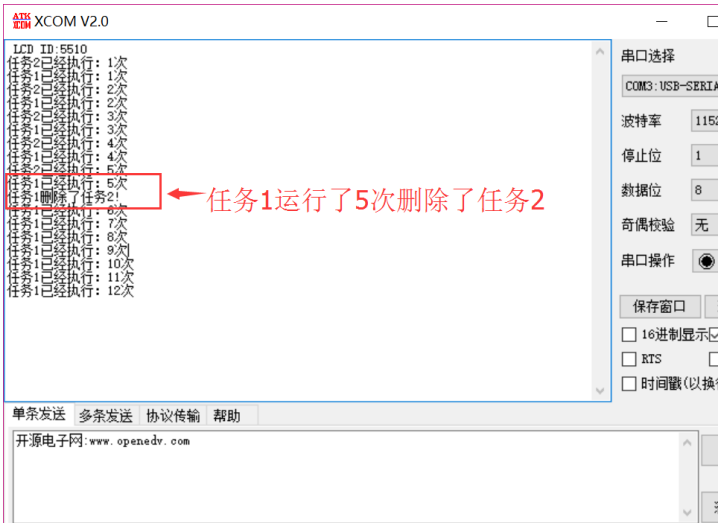


图 6.2.2.2 窗口调试助手输出信息

从图 6.2.2.2 中可以看出，一开始任务 1 和任务 2 是同时运行的，由于任务 2 的优先级比任务 1 的优先级高，所以任务 2 先输出信息。当任务 1 运行了 5 次以后任务 1 就删除了任务 2，最后只剩下了任务 1 在运行了。

6.3 任务创建和删除实验(静态方法)

6.3.1 实验程序设计

1、实验目的

上一小节我们讲了使用函数 xTaskCreate()来创建任务，本节在上一小节的基础上做简单的修改，使用函数 xTaskCreateStatic()来创建任务，也就是静态方法，任务的堆栈、任务控制块就需要由用户来指定了。

2、实验设计

参考实验：FreeRTOS 实验 6-1 FreeRTOS 任务创建和删除实验(动态方法)。

3、实验工程

FreeRTOS 实验 6-2 FreeRTOS 任务创建和删除实验(动态方法)。

4、实验程序与分析

● **系统设置**

使用静态方法创建任务的时候需要将宏 configSUPPORT_STATIC_ALLOCATION 设置为 1，在文件 FreeRTOSConfig.h 中设置，如下所示：

```
#define configSUPPORT_STATIC_ALLOCATION         1                       //静态内存
```

宏 configSUPPORT_STATIC_ALLOCATION 定义为 1 以后编译一次，会提示我们有两个函数未定义，如图 6.3.1.1 所示：

```
..\OBJ\LED.axf: Error: L6218E: Undefined symbol vApplicationGetIdleTaskMemory (referred from tasks.o).
..\OBJ\LED.axf: Error: L6218E: Undefined symbol vApplicationGetTimerTaskMemory (referred from timers.o).
Not enough information to list image symbols.
Finished: 1 information, 0 warning and 2 error messages.
"..\OBJ\LED.axf" - 2 Error(s), 4 Warning(s).
Target not created.
Build Time Elapsed:  00:00:04
```

图 6.3.3.1 错误提示

这个在我们讲 FreeRTOS 的配置文件 FreeRTOSConfig.h 的时候就说过了，如果使用静态方法的话需要用户实现两个函数 vApplicationGetIdleTaskMemory() 和 vApplicationGetTimerTaskMemory()。通过这两个函数来给空闲任务和定时器服务任务的任务堆栈和任务控制块分配内存，这两个函数我们在 mainc.c 中定义，定义如下：

```
//空闲任务任务堆栈
static StackType_t IdleTaskStack[configMINIMAL_STACK_SIZE];
//空闲任务控制块
static StaticTask_t IdleTaskTCB;

//定时器服务任务堆栈
static StackType_t TimerTaskStack[configTIMER_TASK_STACK_DEPTH];
//定时器服务任务控制块
static StaticTask_t TimerTaskTCB;

//获取空闲任务地任务堆栈和任务控制块内存，因为本例程使用的
//静态内存，因此空闲任务的任务堆栈和任务控制块的内存就应该
//有用户来提供，FreeRTOS 提供了接口函数 vApplicationGetIdleTaskMemory()
//实现此函数即可。
//ppxIdleTaskTCBBuffer:任务控制块内存
//ppxIdleTaskStackBuffer:任务堆栈内存
//pulIdleTaskStackSize:任务堆栈大小
void vApplicationGetIdleTaskMemory(StaticTask_t **ppxIdleTaskTCBBuffer,
                                   StackType_t **ppxIdleTaskStackBuffer,
                                   uint32_t *pulIdleTaskStackSize)
{
    *ppxIdleTaskTCBBuffer=&IdleTaskTCB;
    *ppxIdleTaskStackBuffer=IdleTaskStack;
    *pulIdleTaskStackSize=configMINIMAL_STACK_SIZE;
}

//获取定时器服务任务的任务堆栈和任务控制块内存
//ppxTimerTaskTCBBuffer:任务控制块内存
//ppxTimerTaskStackBuffer:任务堆栈内存
//pulTimerTaskStackSize:任务堆栈大小
void vApplicationGetTimerTaskMemory(StaticTask_t **ppxTimerTaskTCBBuffer,
                                    StackType_t **ppxTimerTaskStackBuffer,
                                    uint32_t *pulTimerTaskStackSize)
{
    *ppxTimerTaskTCBBuffer=&TimerTaskTCB;
    *ppxTimerTaskStackBuffer=TimerTaskStack;
    *pulTimerTaskStackSize=configTIMER_TASK_STACK_DEPTH;
}
```

可以看出这两个函数很简单，用户定义静态的任务堆栈和任务控制块内存，然后将这些内

存传递给函数参数。最后创建空闲任务和定时器服务任务的 API 函数会调用 vApplicationGetIdleTaskMemory()和 vApplicationGetTimerTaskMemory()来获取这些内存。

● **任务设置**

```
#define START_TASK_PRIO                 1          //任务优先级
#define START_STK_SIZE                  128        //任务堆栈大小
StackType_t StartTaskStack[START_STK_SIZE];        //任务堆栈          (1)
StaticTask_t StartTaskTCB;                         //任务控制块        (2)
TaskHandle_t StartTask_Handler;                    //任务句柄
void start_task(void *pvParameters);               //任务函数

#define TASK1_TASK_PRIO                 2          //任务优先级
#define TASK1_STK_SIZE                  128        //任务堆栈大小
StackType_t Task1TaskStack[TASK1_STK_SIZE];        //任务堆栈
StaticTask_t Task1TaskTCB;                         //任务控制块
TaskHandle_t Task1Task_Handler;                    //任务句柄
void task1_task(void *pvParameters);               //任务函数

#define TASK2_TASK_PRIO                 3          //任务优先级
#define TASK2_STK_SIZE                  128        //任务堆栈大小
StackType_t Task2TaskStack[TASK2_STK_SIZE];        //任务堆栈
StaticTask_t Task2TaskTCB;                         //任务控制块
TaskHandle_t Task2Task_Handler;                    //任务句柄
void task2_task(void *pvParameters);               //任务函数

//LCD 刷屏时使用的颜色
int lcd_discolor[14]={  WHITE,      BLACK,      BLUE,       BRED,
                        GRED,       GBLUE,      RED,        MAGENTA,
                        GREEN,      CYAN,       YELLOW,     BROWN,
                        BRRED,      GRAY };
```

(1)、静态创建任务需要用户提供任务堆栈，这里定义一个数组作为任务堆栈，堆栈数组为 StackType_t 类型。

(2)、定义任务控制块，注意任务控制块类型要用 StaticTask_t，而不是 TCB_t 或 tskTCB！这里已经要切记！

● **main()函数**

```
int main(void)
{
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(360,25,2,8);   //设置时钟,180Mhz
    delay_init(180);                //初始化延时函数
    uart_init(115200);              //初始化串口
    LED_Init();                     //初始化 LED
    KEY_Init();                     //初始化按键
```

```
    SDRAM_Init();                           //初始化 SDRAM
    LCD_Init();                             //初始化 LCD

    POINT_COLOR = RED;
    LCD_ShowString(30,10,200,16,16,"Apollo STM32F4/F7");
    LCD_ShowString(30,30,200,16,16,"FreeRTOS Examp 5-2");
    LCD_ShowString(30,50,200,16,16,"Task Creat and Del");
    LCD_ShowString(30,70,200,16,16,"ATOM@ALIENTEK");
    LCD_ShowString(30,90,200,16,16,"2016/9/24");
    //创建开始任务
    StartTask_Handler=xTaskCreateStatic((TaskFunction_t)start_task,             //任务函数   (1)
                                        (const char*    )"start_task",          //任务名称
                                        (uint32_t       )START_STK_SIZE,        //任务堆栈大小
                                        (void*          )NULL,          //传递给任务函数的参数
                                        (UBaseType_t    )START_TASK_PRIO,       //任务优先级
                                        (StackType_t*   )StartTaskStack,        //任务堆栈   (2)
                                        (StaticTask_t*  )&StartTaskTCB);        //任务控制块(3)
    vTaskStartScheduler();          //开启任务调度
}
```

(1)、调用函数 xTaskCreateStatic()创建任务。

(2)、将定义的任务堆栈数组传递给函数。

(3)、将定义的任务控制块传递给函数。

可以看出在用法上 xTaskCreateStatic()和 xTaskCreate()没有太大的区别，大多数的参数都相同。学习过 uC/OS 的同学应该会对函数 xTaskCreateStatic()感到熟悉，因为 uC/OS 中创建任务的函数和 xTaskCreateStatic()类似，也需要用户来指定任务堆栈和任务控制块的内存的，然后将其作为参数传递给任务创建函数。不过我们后面所有的例程不管是创建任务、信号量还是队列都使用动态方法。

● **任务函数**

```
//开始任务任务函数
void start_task(void *pvParameters)
{
    taskENTER_CRITICAL();           //进入临界区
    //创建 TASK1 任务
    Task1Task_Handler=xTaskCreateStatic((TaskFunction_t     )task1_task,        (1)
                                        (const char*        )"task1_task",
                                        (uint32_t           )TASK1_STK_SIZE,
                                        (void*              )NULL,
                                        (UBaseType_t        )TASK1_TASK_PRIO,
                                        (StackType_t*       )Task1TaskStack,
                                        (StaticTask_t*      )&Task1TaskTCB);
    //创建 TASK2 任务
    Task2Task_Handler=xTaskCreateStatic((TaskFunction_t     )task2_task,        (2)
                                        (const char*        )"task2_task",
```

```
                                (uint32_t           )TASK2_STK_SIZE,
                                (void*              )NULL,
                                (UBaseType_t        )TASK2_TASK_PRIO,
                                (StackType_t*       )Task2TaskStack,
                                (StaticTask_t*      )&Task2TaskTCB);
    vTaskDelete(StartTask_Handler);         //删除开始任务
    taskEXIT_CRITICAL();                    //退出临界区
}

//task1 任务函数
void task1_task(void *pvParameters)
{
    u8 task1_num=0;

    POINT_COLOR = BLACK;

    LCD_DrawRectangle(5,110,115,314);  //画一个矩形
    LCD_DrawLine(5,130,115,130);       //画线
    POINT_COLOR = BLUE;
    LCD_ShowString(6,111,110,16,16,"Task1 Run:000");
    while(1)
    {
        task1_num++; //任务执 1 行次数加 1  注意 task1_num1 加到 255 的时候会清零!!
        LED0=!LED0;
        printf("任务 1 已经执行：%d 次\r\n",task1_num);
        if(task1_num==5)
        {
            vTaskDelete(Task2Task_Handler);//任务 1 执行 5 次删除任务 2
            printf("任务 1 删除了任务 2!\r\n");
        }
        LCD_Fill(6,131,114,313,lcd_discolor[task1_num%14]);    //填充区域
        LCD_ShowxNum(86,111,task1_num,3,16,0x80);              //显示任务执行次数
        vTaskDelay(1000);  //延时 1s，也就是 1000 个时钟节拍
    }
}

//task2 任务函数
void task2_task(void *pvParameters)
{
    u8 task2_num=0;

    POINT_COLOR = BLACK;
```

```
    LCD_DrawRectangle(125,110,234,314); //画一个矩形
    LCD_DrawLine(125,130,234,130);          //画线
    POINT_COLOR = BLUE;
    LCD_ShowString(126,111,110,16,16,"Task2 Run:000");
    while(1)
    {
        task2_num++; //任务 2 执行次数加 1  注意 task1_num2 加到 255 的时候会清零!!
        LED1=!LED1;
        printf("任务 2 已经执行：%d 次\r\n",task2_num);
        LCD_ShowxNum(206,111,task2_num,3,16,0x80);  //显示任务执行次数
        LCD_Fill(126,131,233,313,lcd_discolor[13-task2_num%14]); //填充区域
        vTaskDelay(1000);                       //延时 1s，也就是 1000 个时钟节拍
    }
}
```

(1)、使用静态任务创建函数 xTaskCreateStatic()来创建任务 task1_task。

(2)、使用静态任务创建函数 xTaskCreateStatic()来创建任务 task2_task。

6.3.2 程序运行结果分析

参考 6.2.2 小节。

6.4 任务挂起和恢复 API 函数

有时候我们需要暂停某个任务的运行，过一段时间以后在重新运行。这个时候要是使用任务删除和重建的方法的话那么任务中变量保存的值肯定丢失了！FreeRTOS 给我们提供了解决这种问题的方法，那就是任务挂起和恢复，当某个任务要停止运行一段时间的话就将这个任务挂起，当要重新运行这个任务的话就恢复这个任务的运行。FreeRTOS 的任务挂起和恢复 API 函数如表 6.2.1.1 所示：

函数	描述
vTaskSuspend()	挂起一个任务。
vTaskResume()	恢复一个任务的运行。
xTaskResumeFromISR()	中断服务函数中恢复一个任务的运行。

表 6.2.1.1 任务挂起和恢复 API 函数

1、函数 vTaskSuspend()

此函数用于将某个任务设置为挂起态，进入挂起态的任务永远都不会进入运行态。退出挂起态的唯一方法就是调用任务恢复函数 vTaskResume()或 xTaskResumeFromISR()。，函数原型如下：

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

参数：

xTaskToSuspend： 要挂起的任务的任务句柄，创建任务的时候会为每个任务分配一个任务句柄。如果使用函数 xTaskCreate()创建任务的话那么函数的参数 pxCreatedTask 就是此任务的任务句柄，如果使用函数 xTaskCreateStatic()创建任务的话那么函数的返回值就是此任务的任务句柄。也可以通过函

数 xTaskGetHandle()来根据任务名字来获取某个任务的任务句柄。
注意！如果参数为 NULL 的话表示挂起任务自己。

返回值：

无。

2、函数 vTaskResume()

将一个任务从挂起态恢复到就绪态，只有通过函数 vTaskSuspend()设置为挂起态的任务才可以使用 vTaskRexume()恢复！函数原型如下：

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

参数：

xTaskToResume： 要恢复的任务的任务句柄。

返回值：

无。

3、函数 xTaskResumeFromISR()

此函数是 vTaskResume()的中断版本，用于在中断服务函数中恢复一个任务。函数原型如下：

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

参数：

xTaskToResume: 要恢复的任务的任务句柄。

返回值：

pdTRUE: 恢复运行的任务的任务优先级等于或者高于正在运行的任务(被中断打断的任务)，这意味着在退出中断服务函数以后必须进行一次上下文切换。

pdFALSE: 恢复运行的任务的任务优先级低于当前正在运行的任务(被中断打断的任务)，这意味着在退出中断服务函数的以后不需要进行上下文切换。

6.5 任务挂起和恢复实验

6.5.1 实验程序设计

1、实验目的

学习使用 FreeRTOS 的任务挂起和恢复相关 API 函数，包括 vTaskSuspend()、vTaskResume()和 xTaskResumeFromISR()。

2、实验设计

本实验设计 4 个任务：start_task、key_task、task1_task 和 task2_task，这四个任务的任务功能如下：

start_task：用来创建其他 3 个任务。

key_task： 按键服务任务，检测按键的按下结果，根据不同的按键结果执行不同的操作。

task1_task：应用任务 1。
task2_task:　应用任务 2。
实验需要四个按键，KEY0、KEY1、KEY2 和 KEY_UP，这四个按键的功能如下：
KEY0:　此按键为中断模式，在中断服务函数中恢复任务 2 的运行。
KEY1:　此按键为输入模式，用于恢复任务 1 的运行。
KEY2:　此按键为输入模式，用于挂起任务 2 的运行。
KEY_UP:　此按键为输入模式，用于挂起任务 1 的运行。

3、实验工程

FreeRTOS 实验 6-3 FreeRTOS 任务挂起和恢复实验。

4、实验程序与分析

● 任务设置

实验中任务优先级、堆栈大小和任务句柄等的设置如下：

```
#define START_TASK_PRIO          1        //任务优先级
#define START_STK_SIZE           128      //任务堆栈大小
TaskHandle_t StartTask_Handler;           //任务句柄
void start_task(void *pvParameters);      //任务函数

#define KEY_TASK_PRIO            2        //任务优先级
#define KEY_STK_SIZE             128      //任务堆栈大小
TaskHandle_t KeyTask_Handler;             //任务句柄
void key_task(void *pvParameters);        //任务函数

#define TASK1_TASK_PRIO          3        //任务优先级

#define TASK1_STK_SIZE           128      //任务堆栈大小
TaskHandle_t Task1Task_Handler;           //任务句柄
void task1_task(void *pvParameters);      //任务函数

#define TASK2_TASK_PRIO          4        //任务优先级
#define TASK2_STK_SIZE           128      //任务堆栈大小
TaskHandle_t Task2Task_Handler;           //任务句柄
void task2_task(void *pvParameters);      //任务函数
```

● main()函数

```
int main(void)
{
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(360,25,2,8);   //设置时钟,180Mhz
    delay_init(180);                //初始化延时函数
    uart_init(115200);              //初始化串口
    LED_Init();                     //初始化 LED
    KEY_Init();                     //初始化按键
```

```
    EXTI_Init();                        //初始化外部中断
    SDRAM_Init();                       //初始化 SDRAM
    LCD_Init();                         //初始化 LCD

    POINT_COLOR = RED;
    LCD_ShowString(30,10,200,16,16,"Apollo STM32F4/F7");
    LCD_ShowString(30,30,200,16,16,"FreeRTOS Examp 5-1");
    LCD_ShowString(30,50,200,16,16," Task Susp and Resum");
    LCD_ShowString(30,70,200,16,16,"ATOM@ALIENTEK");
    LCD_ShowString(30,90,200,16,16,"2016/9/24");
    //创建开始任务
    xTaskCreate((TaskFunction_t     )start_task,            //任务函数
                (const char*        )"start_task",          //任务名称
                (uint16_t           )START_STK_SIZE,        //任务堆栈大小
                (void*              )NULL,                  //传递给任务函数的参数
                (UBaseType_t        )START_TASK_PRIO,       //任务优先级
                (TaskHandle_t*      )&StartTask_Handler);   //任务句柄
    vTaskStartScheduler();                                  //开启任务调度
}
```

在 main 函数中我们主要完成硬件的初始化，在硬件初始化完成以后创建了任务 start_task()并且开启了 FreeRTOS 的任务调度。

● **任务函数**

```
//开始任务任务函数
void start_task(void *pvParameters)                                    (1)
{
    taskENTER_CRITICAL();               //进入临界区
    //创建 KEY 任务
    xTaskCreate((TaskFunction_t     )key_task,
                (const char*        )"key_task",
                (uint16_t           )KEY_STK_SIZE,
                (void*              )NULL,
                (UBaseType_t        )KEY_TASK_PRIO,
                (TaskHandle_t*      )&KeyTask_Handler);
    //创建 TASK1 任务
    xTaskCreate((TaskFunction_t     )task1_task,
                (const char*        )"task1_task",
                (uint16_t           )TASK1_STK_SIZE,
                (void*              )NULL,
                (UBaseType_t        )TASK1_TASK_PRIO,
                (TaskHandle_t*      )&Task1Task_Handler);
    //创建 TASK2 任务
    xTaskCreate((TaskFunction_t     )task2_task,
                (const char*        )"task2_task",
```

```
                (uint16_t           )TASK2_STK_SIZE,
                (void*              )NULL,
                (UBaseType_t        )TASK2_TASK_PRIO,
                (TaskHandle_t*      )&Task2Task_Handler);
    vTaskDelete(StartTask_Handler);         //删除开始任务
    taskEXIT_CRITICAL();                    //退出临界区
}

//key 任务函数
void key_task(void *pvParameters)
{
    u8 key;
    while(1)
    {
        key=KEY_Scan(0);
        switch(key)
        {
            case WKUP_PRES:
                vTaskSuspend(Task1Task_Handler);    //挂起任务 1                (2)
                printf("挂起任务 1 的运行!\r\n");
                break;
            case KEY1_PRES:
                vTaskResume(Task1Task_Handler);     //恢复任务 1                (3)
                printf("恢复任务 1 的运行!\r\n");
                break;
            case KEY2_PRES:
                vTaskSuspend(Task2Task_Handler);//挂起任务 2                    (4)
                printf("挂起任务 2 的运行!\r\n");
                break;
        }
        vTaskDelay(10);                 //延时 10ms
    }
}

//task1 任务函数
void task1_task(void *pvParameters)                                             (5)
{
    u8 task1_num=0;

    POINT_COLOR = BLACK;

    LCD_DrawRectangle(5,110,115,314);  //画一个矩形
    LCD_DrawLine(5,130,115,130);       //画线
```

```
    POINT_COLOR = BLUE;
    LCD_ShowString(6,111,110,16,16,"Task1 Run:000");
    while(1)
    {
        task1_num++; //任务执 1 行次数加 1  注意 task1_num1 加到 255 的时候会清零!!
        LED0=!LED0;
        printf("任务 1 已经执行：%d 次\r\n",task1_num);
        LCD_Fill(6,131,114,313,lcd_discolor[task1_num%14]); //填充区域
        LCD_ShowxNum(86,111,task1_num,3,16,0x80); //显示任务执行次数
        vTaskDelay(1000);                          //延时 1s，也就是 1000 个时钟节拍
    }
}

//task2 任务函数
void task2_task(void *pvParameters)                                    (6)
{
    u8 task2_num=0;

    POINT_COLOR = BLACK;

    LCD_DrawRectangle(125,110,234,314);     //画一个矩形
    LCD_DrawLine(125,130,234,130);          //画线
    POINT_COLOR = BLUE;
    LCD_ShowString(126,111,110,16,16,"Task2 Run:000");
    while(1)
    {
        task2_num++; //任务 2 执行次数加 1  注意 task1_num2 加到 255 的时候会清零!!
        LED1=!LED1;
        printf("任务 2 已经执行：%d 次\r\n",task2_num);
        LCD_ShowxNum(206,111,task2_num,3,16,0x80);          //显示任务执行次数
        LCD_Fill(126,131,233,313,lcd_discolor[13-task2_num%14]);  //填充区域
        vTaskDelay(1000);                          //延时 1s，也就是 1000 个时钟节拍
    }
}
```

(1)、start_task 任务，用于创建其他 3 个任务。

(2)、在 key_tssk 任务里面，KEY_UP 被按下，调用函数 vTaskSuspend()挂起任务 1。

(3)、KEY1 被按下，调用函数 vTaskResume()恢复任务 1 的运行。

(4)、KEY2 被按下，调用函数 vTaskSuspend()挂起任务 2。

(5)、任务 1 的任务函数，用于观察任务挂起和恢复的过程。

(6)、任务 2 的任务函数，用于观察任务挂起和恢复的过程(中断方式)。

● **中断初始化及处理过程**

```
//外部中断初始化
void EXTI_Init(void)
```

```
{
    GPIO_InitTypeDef GPIO_Iniure;

    __HAL_RCC_GPIOH_CLK_ENABLE();              //开启 GPIOH 时钟

    GPIO_Iniure.Pin=GPIO_PIN_3;                //PH3
    GPIO_Iniure.Mode=GPIO_MODE_IT_FALLING;     //下降沿触发
    GPIO_Iniure.Pull=GPIO_PULLUP;
    HAL_GPIO_Init(GPIOH,&GPIO_Iniure);

    //中断线 3-PH3
    HAL_NVIC_SetPriority(EXTI3_IRQn,6,0);      //抢占优先级为 6，子优先级为 0   (1)
    HAL_NVIC_EnableIRQ(EXTI3_IRQn);            //使能中断线 3
}

//任务句柄
extern TaskHandle_t Task2Task_Handler;

//中断服务函数
void EXTI3_IRQHandler(void)
{
    BaseType_t YieldRequired;

    delay_xms(50);                             //消抖
    if(KEY0==0)
    {
        YieldRequired=xTaskResumeFromISR(Task2Task_Handler);//恢复任务 2          (2)
        printf("恢复任务 2 的运行!\r\n");
        if(YieldRequired==pdTRUE)
        {
            /*如果函数 xTaskResumeFromISR()返回值为 pdTRUE，那么说明要恢复的这个
            任务的任务优先级等于或者高于正在运行的任务(被中断打断的任务),所以在
            退出中断的时候一定要进行上下文切换！ */
            portYIELD_FROM_ISR(YieldRequired);                                  (3)
        }
    }
    __HAL_GPIO_EXTI_CLEAR_IT(GPIO_PIN_3);     //清除中断标志位
}
```

(1)、设置中断优先级，前面在讲解 FreeRTOS 中断的时候就讲过，如果中断服务函数要使用 FreeRTOS 的 API 函数的话那么中断优先级一定要低于 configMAX_SYSCALL_INTERRUPT_PRIORITY！这里设置为 6。

(2)、调用函数 xTaskResumeFromISR()来恢复任务 2 的运行。

(3)、根据函数 xTaskResumeFromISR()的返回值来确定是否需要进行上下文切换。当返回值

为 pdTRUE 的时候就需要调用函数 portYIELD_FROM_ISR()进行上下文切换，否则的话不需要。

6.5.2 程序运行结果分析

编译并下载程序到开发板中，通过按不同的按键来观察任务的挂起和恢复的过程，如图 6.5.2.1 所示：

图 6.5.2.1 程序运行结果

从图 6.5.2.1 可以看出，一开始任务 1 和任务 2 都正常运行，当挂起任务 1 或者任务 2 以后任务 1 或者任务 2 就会停止运行，直到下一次重新恢复任务 1 或者任务 2 的运行。重点是，保存任务运行次数的变量都没有发生数据丢失，如果用任务删除和重建的方法这些数据必然会丢失掉的！

第七章 FreeRTOS 列表和列表项

要想看懂 FreeRTOS 源码并学习其原理，有一个东西绝对跑不了，那就是 FreeRTOS 的列表和列表项。列表和列表项是 FreeRTOS 的一个数据结构，FreeRTOS 大量使用到了列表和列表项，它是 FreeRTOS 的基石。要想深入学习并理解 FreeRTOS，那么列表和列表项就必须首先掌握，否则后面根本就没法进行。本章我们就来学习一下 FreeRTOS 的列表和列表项，包括对列表和列表项的操作，本章分为如下几部分：

7.1 什么是列表和列表项
7.2 列表和列表项的初始化
7.3 列表项的插入
7.4 列表项末尾插入
7.5 列表项的删除
7.6 列表项的遍历
7.7 列表项的插入和删除实验

7.1 什么是列表和列表项？

7.1.1 列表

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。与列表相关的全部东西都在文件 list.c 和 list.h 中。在 list.h 中定义了一个叫 List_t 的结构体，如下：

```
typedef struct xLIST
{
    listFIRST_LIST_INTEGRITY_CHECK_VALUE                (1)
    configLIST_VOLATILE UBaseType_t      uxNumberOfItems;   (2)
    ListItem_t * configLIST_VOLATILE     pxIndex;           (3)
    MiniListItem_t                       xListEnd;          (4)
    listSECOND_LIST_INTEGRITY_CHECK_VALUE               (5)
} List_t;
```

(1) 和 (5)、这两个都是用来检查列表完整性的，需要将宏 configUSE_LIST_DATA_INTEGRITY_CHECK_BYTES 设置为 1，开启以后会向这两个地方分别添加一个变量 xListIntegrityValue1 和 xListIntegrityValue2，在初始化列表的时候会这两个变量中写入一个特殊的值，默认不开启这个功能。以后我们在学习列表的时候不讨论这个功能！

(2)、uxNumberOfItems 用来记录列表中列表项的数量。

(3)、pxIndex 用来记录当前列表项索引号，用于遍历列表。

(4)、列表中最后一个列表项，用来表示列表结束，此变量类型为 MiniListItem_t，这是一个迷你列表项，关于列表项稍后讲解。

列表结构示意图如图 7.1.1.1 所示：

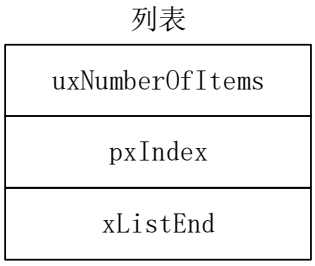


图 7.1.1.1 列表示意图

注意！图 7.1.1.1 中并未列出用于列表完整性检查的成员变量。

7.1.2 列表项

列表项就是存放在列表中的项目，FreeRTOS 提供了两种列表项：列表项和迷你列表项。这两个都在文件 list.h 中有定义，先来看一下列表项，定义如下：

```
struct xLIST_ITEM
{
    listFIRST_LIST_ITEM_INTEGRITY_CHECK_VALUE           (1)
    configLIST_VOLATILE TickType_t       xItemValue;    (2)
    struct xLIST_ITEM * configLIST_VOLATILE  pxNext;    (3)
    struct xLIST_ITEM * configLIST_VOLATILE  pxPrevious;(4)
    void *                               pvOwner;       (5)
```

```
    void * configLIST_VOLATILE                  pvContainer;            (6)
    listSECOND_LIST_ITEM_INTEGRITY_CHECK_VALUE                          (7)
};
typedef struct  xLIST_ITEM    ListItem_t;
```

(1)和(7)、用法和列表一样，用来检查列表项完整性的。以后我们在学习列表项的时候不讨论这个功能！

(2)、xItemValue 为列表项值。

(3)、pxNext 指向下一个列表项。

(4)、pxPrevious 指向前一个列表项，和 pxNext 配合起来实现类似双向链表的功能。

(5)、pvOwner 记录此链表项归谁拥有，通常是任务控制块。

(6)、pvContainer 用来记录此列表项归哪个列表。注意和 pvOwner 的区别，在前面讲解任务控制块 TCB_t 的时候说了在 TCB_t 中有两个变量 xStateListItem 和 xEventListItem，这两个变量的类型就是 ListItem_t，也就是说这两个成员变量都是列表项。以 xStateListItem 为例，当创建一个任务以后 xStateListItem 的 pvOwner 变量就指向这个任务的任务控制块，表示 xSateListItem 属于此任务。当任务就绪态以后 xStateListItem 的变量 pvContainer 就指向就绪列表，表明此列表项在就绪列表中。举个通俗一点的例子：小王在上二年级，他的父亲是老王。如果把小王比作列表项，那么小王的 pvOwner 属性值就是老王，小王的 pvContainer 属性值就是二年级。

列表项结构示意图如图 7.1.2.1 所示：

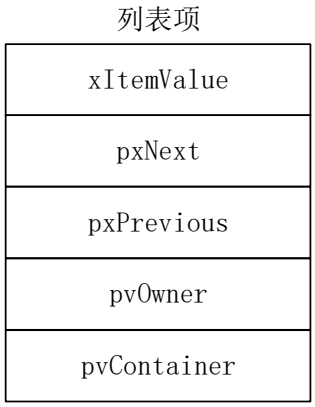


图 7.1.2.1 列表项示意图

注意！图 7.1.2.1 中并未列出用于列表项完整性检查的成员变量！

7.1.3 迷你列表项

上面我们我们说了列表项，现在来看一下迷你列表项，迷你列表项在文件 list.h 中有定义，如下：

```
struct xMINI_LIST_ITEM
{
    listFIRST_LIST_ITEM_INTEGRITY_CHECK_VALUE                   (1)
    configLIST_VOLATILE TickType_t              xItemValue;     (2)
    struct xLIST_ITEM * configLIST_VOLATILE     pxNext;         (3)
    struct xLIST_ITEM * configLIST_VOLATILE     pxPrevious;     (4)
};
typedef struct xMINI_LIST_ITEM MiniListItem_t;
```

(1)、用于检查迷你列表项的完整性。

(2)、xItemValue 记录列表列表项值。

(3)、pxNext 指向下一个列表项。

(4)、pxPrevious 指向上一个列表项。

可以看出迷你列表项只是比列表项少了几个成员变量，迷你列表项有的成员变量列表项都有的，没感觉有什么本质区别啊？那为什么要弄个迷你列表项出来呢？那是因为有些情况下我们不需要列表项这么全的功能，可能只需要其中的某几个成员变量，如果此时用列表项的话会造成内存浪费！比如上面列表结构体 List_t 中表示最后一个列表项的成员变量 xListEnd 就是 MiniListItem_t 类型的。

迷你列表项结构示意图如图 7.1.3.1 所示：

图 7.1.3.1 迷你列表项示意图

注意！图 7.1.3.1 中并未列出用于迷你列表项完整性检查的成员变量！

7.2 列表和列表项初始化

7.2.1 列表初始化

新创建或者定义的列表需要对其做初始化处理，列表的初始化其实就是初始化列表结构体 List_t 中的各个成员变量，列表的初始化通过使函数 vListInitialise()来完成，此函数在 list.c 中有定义，函数如下：

```
void vListInitialise( List_t * const pxList )
{
    pxList->pxIndex = ( ListItem_t * ) &( pxList->xListEnd );                    (1)
    pxList->xListEnd.xItemValue = portMAX_DELAY;                                 (2)
    pxList->xListEnd.pxNext = ( ListItem_t * ) &( pxList->xListEnd );            (3)
    pxList->xListEnd.pxPrevious = ( ListItem_t * ) &( pxList->xListEnd );        (4)
    pxList->uxNumberOfItems = ( UBaseType_t ) 0U;                                (5)

    listSET_LIST_INTEGRITY_CHECK_1_VALUE( pxList );                              (6)
    listSET_LIST_INTEGRITY_CHECK_2_VALUE( pxList );                              (7)
}
```

(1)、xListEnd 用来表示列表的末尾，而 pxIndex 表示列表项的索引号，此时列表只有一个列表项，那就是 xListEnd，所以 pxIndex 指向 xListEnd。

(2)、xListEnd 的列表项值初始化为 portMAX_DELAY， portMAX_DELAY 是个宏，在文件 portmacro.h 中有定义。根据所使用的 MCU 的不同，portMAX_DELAY 值也不相同，可以为 0xffff 或者 0xffffffffUL，本教程中为 0xffffffffUL。

(3)、初始化列表项 xListEnd 的 pxNext 变量，因为此时列表只有一个列表项 xListEnd，因此 pxNext 只能指向自身。

(4)、同(3)一样，初始化 xListEnd 的 pxPrevious 变量，指向 xListEnd 自身。

(5)、由于此时没有其他的列表项，因此 uxNumberOfItems 为 0，注意，这里没有算 xListEnd。

(6)和(7)、初始化列表项中用于完整性检查字段，只有宏 configUSE_LIST_DATA_INTEGRITY_CHECK_BYTES 为 1 的时候才有效。同样的根据所选的 MCU 不同其写入的值也不同，可以为 0x5a5a 或者 0x5a5a5a5aUL。STM32 是 32 位系统写入的是 0x5a5a5a5aUL，列表初始化完以后如图 7.2.1.1 所示：

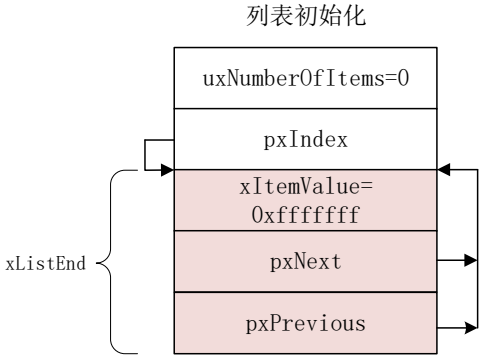


图 7.2.1.1 列表初始化

注意，图 7.2.1.1 为了好分析，将 xListEnd 中的各个成员变量都写了出来！

7.2.2 列表项初始化

同列表一样，列表项在使用的时候也需要初始化，列表项初始化由函数 vListInitialiseItem()来完成，函数如下：

```
void vListInitialiseItem( ListItem_t * const pxItem )
{
    pxItem->pvContainer = NULL;                    //初始化 pvContainer 为 NULL

    //初始化用于完整性检查的变量，如果开启了这个功能的话。
    listSET_FIRST_LIST_ITEM_INTEGRITY_CHECK_VALUE( pxItem );
    listSET_SECOND_LIST_ITEM_INTEGRITY_CHECK_VALUE( pxItem );
}
```

列表项的初始化很简单，只是将列表项成员变量 pvContainer 初始化为 NULL，并且给用于完整性检查的变量赋值。有朋友可能会问，列表项的成员变量比列表要多，怎么初始化函数就这么短？其他的成员变量什么时候初始化呢？这是因为列表项要根据实际使用情况来初始化，比如任务创建函数 xTaskCreate()就会对任务堆栈中的 xStateListItem 和 xEventListItem 这两个列表项中的其他成员变量在做初始化，任务创建过程后面会详细讲解。

7.3 列表项插入

7.3.1 列表项插入函数分析

列表项的插入操作通过函数 vListInsert()来完成，函数原型如下：

```
void vListInsert( List_t * const           pxList,
                  ListItem_t * const       pxNewListItem )
```

参数：

pxList: 列表项要插入的列表。
pxNewListItem: 要插入的列表项。

返回值:

无

函数 vListInsert()的参数 pxList 决定了列表项要插入到哪个列表中，pxNewListItem 决定了要插入的列表项，但是这个列表项具体插入到什么地方呢？要插入的位置由列表项中成员变量 xItemValue 来决定。列表项的插入根据 xItemValue 的值按照升序的方式排列！接下来我们来具体看一下函数 vListInsert()的整个运行过程，函数代码如下：

```
void vListInsert( List_t * const pxList, ListItem_t * const pxNewListItem )
{
    ListItem_t *pxIterator;
    const TickType_t xValueOfInsertion = pxNewListItem->xItemValue;              (1)

    listTEST_LIST_INTEGRITY( pxList );                                           (2)
    listTEST_LIST_ITEM_INTEGRITY( pxNewListItem );

    if( xValueOfInsertion == portMAX_DELAY )                                     (3)
    {
        pxIterator = pxList->xListEnd.pxPrevious;                                (4)
    }
    else
    {

        for( pxIterator = ( ListItem_t * ) &( pxList->xListEnd ); pxIterator->\  (5)
                pxNext->xItemValue <=xValueOfInsertion; pxIterator = pxIterator->pxNext )
        {
            //空循环，什么也不做！
        }
    }

    pxNewListItem->pxNext = pxIterator->pxNext;                                  (6)
    pxNewListItem->pxNext->pxPrevious = pxNewListItem;
    pxNewListItem->pxPrevious = pxIterator;
    pxIterator->pxNext = pxNewListItem;

    pxNewListItem->pvContainer = ( void * ) pxList;                              (7)

    ( pxList->uxNumberOfItems )++;                                               (8)
}
```

(1)、获取要插入的列表项值，即列表项成员变量 xItemValue 的值，因为要根据这个值来确定列表项要插入的位置。

(2)、这一行和下一行代码用来检查列表和列表项的完整性的。其实就是检查列表和列表项

中用于完整性检查的变量值是否被改变。这些变量的值在列表和列表项初始化的时候就被写入了，这两行代码需要实现函数 configASSERT()！

(3)、要插入列表项，第一步就是要获取该列表项要插入到什么位置！如果要插入的列表项的值等于 portMAX_DELAY，也就是说列表项值为最大值，这种情况最好办了，要插入的位置就是列表最末尾了。

(4)、获取要插入点，注意！列表中的 xListEnd 用来表示列表末尾，在初始化列表的时候 xListEnd 的列表值也是 portMAX_DELAY，此时要插入的列表项的列表值也是 portMAX_DELAY。这两个的顺序该怎么放啊？通过这行代码可以看出要插入的列表项会被放到 xListEnd 前面。

(5)、要插入的列表项的值如果不等于 portMAX_DELAY 那么就需要在列表中一个一个的找自己的位置，这个 for 循环就是找位置的过程，当找到合适列表项的位置的时候就会跳出。由于这个 for 循环是用来寻找列表项插入点的，所以 for 循环体里面没有任何东西。这个查找过程是按照升序的方式查找列表项插入点的。

(6)、经过上面的查找，我们已经找到列表项的插入点了，从本行开始接下来的四行代码就是将列表项插入到列表中，插入过程和数据结构中双向链表的插入类似。像 FreeRTOS 这种 RTOS 系统和一些协议栈都会大量用到数据结构的知识，所以建议大家没事的时候多看看数据结构方面的书籍，否则的话看源码会很吃力的。

(7)、列表项已经插入到列表中了，那么列表项的成员变量 pvContainer 也该记录此列表项属于哪个列表的了。

(8)、列表的成员变量 uxNumberOfItems 加一，表示又添加了一个列表项。

7.3.2 列表项插入过程图示

上一小节我们分析了列表项插入函数 vListInsert()，本小节我们就通过图片来演示一下这个插入过程，本小节我们会向一个空的列表中插入三个列表项，这三个列表项的值分别为 40，60 和 50。

1、插入值为 40 的列表项

在一个空的列表 List 中插入一个列表值为 40 的列表项 ListItem1，插入完成以后如图 7.3.2.1 所示：

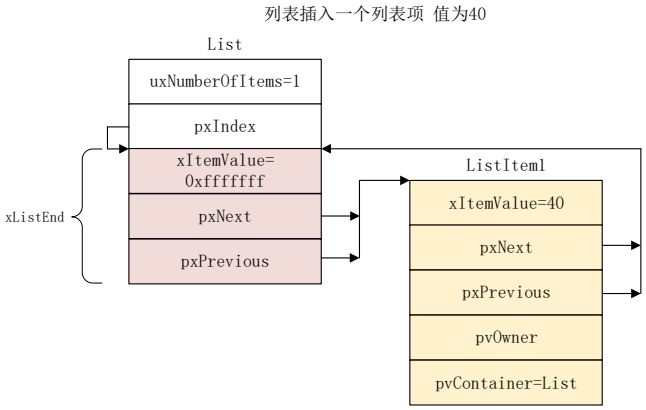


图 7.3.2.1 插入列表项 ListItem1

注意观察插入完成以后列表 List 和列表项 ListItem1 中各个成员变量之间的变化，比如列表 List 中的 uxNumberOfItems 变为了 1，表示现在列表中有一个列表项。列表项 ListItem1 中的 pvContainer 变成了 List，表示此列表项属于列表 List。通过图 7.3.2.1 可以看出，列表是一个环形的，即环形列表！

2、插入值为 60 的列表项

接着再插入一个值为 60 的列表项 ListItem2，插入完成以后如图 7.3.2.2 所示：

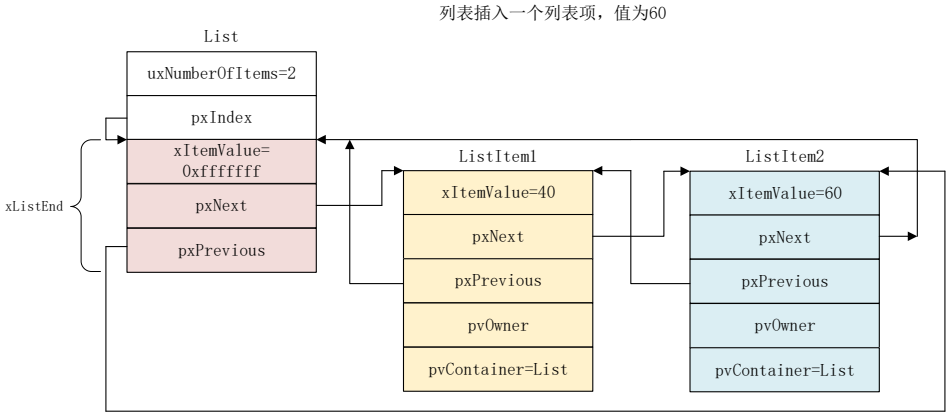


图 7.3.2.2 插入列表项 ListItem2

上面再讲解函数 vListInsert()的时候说过了列表项是按照升序的方式插入的，所以 ListItem2 肯定是插入到 ListItem1 的后面、xListEnd 的前面。同样的，列表 List 的 uxNumberOfItems 再次加一变为 2 了，说明此时列表中有两个列表项。

3、插入值为 50 的列表项

在上面的列表中再插入一个值为 50 的列表项 ListItem3，插入完成以后如图 7.3.2.3 所示：

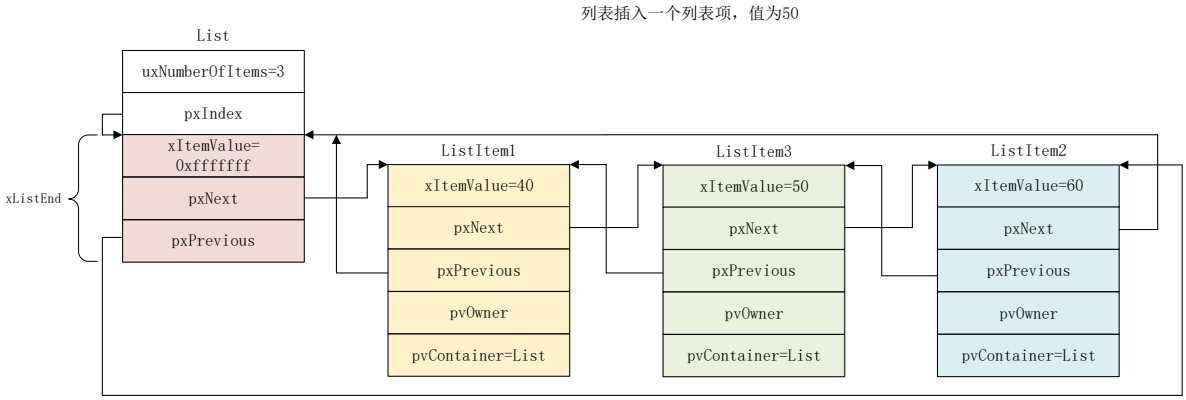


图 7.3.2.3 插入列表项 ListItem3

按照升序排列的方式，ListItem3 应该放到 ListItem1 和 ListItem2 中间，大家最好通过对照这三幅图片来阅读函数 vListInsert()的源码，这样就会对函数有一个直观的认识。

7.4 列表项末尾插入

7.4.1 列表项末尾插入函数分析

列表末尾插入列表项的操作通过函数 vListInsertEnd ()来完成，函数原型如下：

```
void vListInsertEnd( List_t * const       pxList,
                     ListItem_t * const   pxNewListItem )
```

参数：

pxList: 列表项要插入的列表。

pxNewListItem: 要插入的列表项。

返回值:

无

函数 vListInsertEnd()源码如下：

```
void vListInsertEnd( List_t * const pxList, ListItem_t * const pxNewListItem )
{
ListItem_t * const pxIndex = pxList->pxIndex;

    listTEST_LIST_INTEGRITY( pxList );                                  (1)
    listTEST_LIST_ITEM_INTEGRITY( pxNewListItem );

    pxNewListItem->pxNext = pxIndex;                                    (2)
    pxNewListItem->pxPrevious = pxIndex->pxPrevious;

    mtCOVERAGE_TEST_DELAY();

    pxIndex->pxPrevious->pxNext = pxNewListItem;
    pxIndex->pxPrevious = pxNewListItem;

    pxNewListItem->pvContainer = ( void * ) pxList;                     (3)

    ( pxList->uxNumberOfItems )++;                                      (4)
}
```

(1)、与下面的一行代码完成对列表和列表项的完整性检查。

(2)、从本行开始到(3)之间的代码就是将要插入的列表项插入到列表末尾。使用函数 vListInsert()向列表中插入一个列表项的时候这个列表项的位置是通过列表项的值，也就是列表项成员变量 xItemValue 来确定。vListInsertEnd()是往列表的末尾添加列表项的，我们知道列表中的 xListEnd 成员变量表示列表末尾的，那么函数 vListInsertEnd()插入一个列表项是不是就是插到 xListEnd 的前面或后面啊？这个是不一定的，这里所谓的末尾要根据列表的成员变量 pxIndex 来确定的！前面说了列表中的 pxIndex 成员变量是用来遍历列表的，pxIndex 所指向的列表项就是要遍历的开始列表项，也就是说 pxIndex 所指向的列表项就代表列表头！由于是个环形列表，所以新的列表项就应该插入到 pxIndex 所指向的列表项的前面。

(3)、标记新的列表项 pxNewListItem 属于列表 pxList。

(4)、记录列表中的列表项数目的变量加一，更新列表项数目。

7.4.2 列表项末尾插入图示

跟函数 vListInsert()一样，我们也用图片来看一下函数 vListInsertEnd()的插入过程。

1、默认列表

在插入列表项之前我们先准备一个默认列表，如图 7.4.2.1 所示：

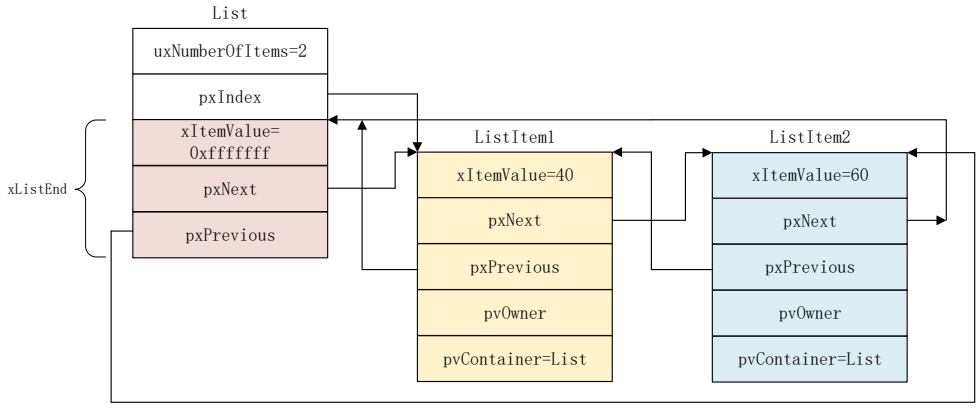


图 7.4.2.1 默认列表

注意图 7.4.2.1 中列表的 pxIndex 所指向的列表项，这里为 ListItem1，不再是 xListEnd 了。

3、插入值为 50 的列表项

在上面的列表中插入一个值为 50 的列表项 ListItem3，插入完成以后如图 7.4.2.2 所示：

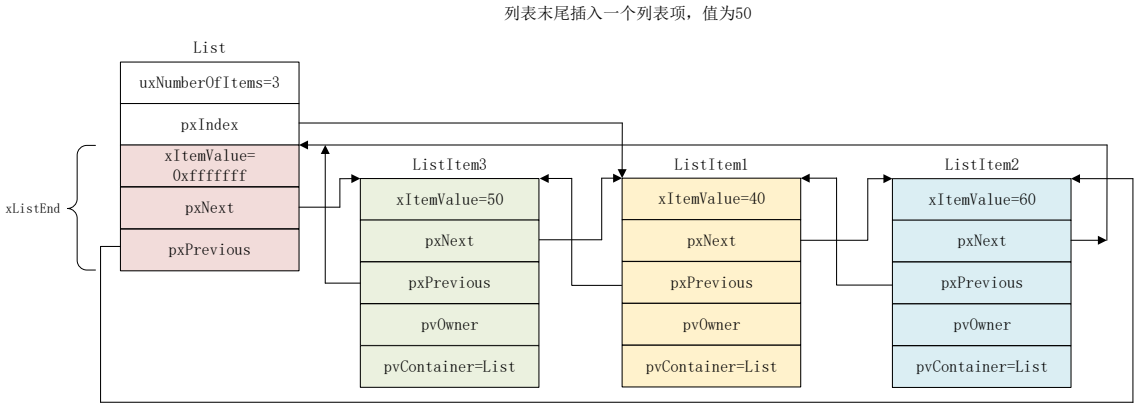


图 7.4.2.2 插入列表项 ListItem3

列表 List 的 pxIndex 指向列表项 ListItem1，因此调用函数 vListInsertEnd()插入 ListItem3 的话就会在 ListItem1 的前面插入。

7.5 列表项的删除

有列表项的插入，那么必然有列表项的删除，列表项的删除通过函数 uxListRemove()来完成，函数原型如下：

```
UBaseType_t uxListRemove( ListItem_t * const pxItemToRemove )
```

参数：

pxItemToRemove: 要删除的列表项。

返回值： 返回删除列表项以后的列表剩余列表项数目。

注意，列表项的删除只是将指定的列表项从列表中删除掉，并不会将这个列表项的内存给释放掉！如果这个列表项是动态分配内存的话。函数 uxListRemove()的源码如下：

```
UBaseType_t uxListRemove( ListItem_t * const pxItemToRemove )
{
    List_t * const pxList = ( List_t * ) pxItemToRemove->pvContainer;        (1)
```

```
    pxItemToRemove->pxNext->pxPrevious = pxItemToRemove->pxPrevious;      (2)
    pxItemToRemove->pxPrevious->pxNext = pxItemToRemove->pxNext;

    mtCOVERAGE_TEST_DELAY();

    if( pxList->pxIndex == pxItemToRemove )
    {
        pxList->pxIndex = pxItemToRemove->pxPrevious;                     (3)
    }
    else
    {
        mtCOVERAGE_TEST_MARKER();
    }

    pxItemToRemove->pvContainer = NULL;                                   (4)
    ( pxList->uxNumberOfItems )--;

    return pxList->uxNumberOfItems;                                       (5)
}
```

(1)、要删除一个列表项我们得先知道这个列表项处于哪个列表中，直接读取列表项中的成员变量 pvContainer 就可以得到此列表项处于哪个列表中。

(2)、与下面一行完成列表项的删除，其实就是将要删除的列表项的前后两个列表项“连接”在一起。

(3)、如果列表的 pxIndex 正好指向要删除的列表项，那么在删除列表项以后要重新给 pxIndex 找个“对象”啊，这个新的对象就是被删除的列表项的前一个列表项。

(4)、被删除列表项的成员变量 pvContainer 清零。

(5)、返回新列表的当前列表项数目。

7.6 列表的遍历

介绍列表结构体的时候说过列表 List_t 中的成员变量 pxIndex 是用来遍历列表的，FreeRTOS 提供了一个函数来完成列表的遍历，这个函数是 listGET_OWNER_OF_NEXT_ENTRY()。每调用一次这个函数列表的 pxIndex 变量就会指向下一个列表项，并且返回这个列表项的 pxOwner 变量值。这个函数本质上是一个宏，这个宏在文件 list.h 中如下定义：

```
#define listGET_OWNER_OF_NEXT_ENTRY( pxTCB, pxList )                      \   (1)
{                                                                         \
    List_t * const pxConstList = ( pxList );                              \
    ( pxConstList )->pxIndex = ( pxConstList )->pxIndex->pxNext;          \   (2)
    if( ( void * ) ( pxConstList )->pxIndex == ( void * ) &( ( pxConstList )->xListEnd ) )\   (3)
    {                                                                     \
        ( pxConstList )->pxIndex = ( pxConstList )->pxIndex->pxNext;      \   (4)
    }                                                                     \
    ( pxTCB ) = ( pxConstList )->pxIndex->pvOwner;                        \   (5)
```

```
}
```

(1)、pxTCB 用来保存 pxIndex 所指向的列表项的 pvOwner 变量值，也就是这个列表项属于谁的？通常是一个任务的任务控制块。pxList 表示要遍历的列表。

(2)、列表的 pxIndex 变量指向下一个列表项。

(3)、如果 pxIndex 指向了列表的 xListEnd 成员变量，表示到了列表末尾。

(4)、如果到了列表末尾的话就跳过 xListEnd，pxIndex 再一次重新指向处于列表头的列表项，这样就完成了一次对列表的遍历。

(5)、将 pxIndex 所指向的新列表项的 pvOwner 赋值给 pxTCB。

此函数用于从多个同优先级的就绪任务中查找下一个要运行的任务。

7.7 列表项的插入和删除实验

上面我们通过分析源码的方式了解了 FreeRTOS 的列表和列表项的操作过程，但是实际上究竟是不是这样的、我们的分析有没有错误？最好的检验方法就是写一段测试代码测试一下，观察在对列表和列表项操作的时候其变化情况和我们分析的是否一致。

7.7.1 实验程序设计

1、实验目的

学习使用 FreeRTOS 列表和列表项相应的操作函数的使用，观察这些操作函数的运行结果和我们理论分析的是否一致。

2、实验设计

本实验设计 3 个任务：start_task、task1_task 和 list_task，这三个任务的任务功能如下：

start_task：用来创建其他 2 个任务。

task1_task：应用任务 1，控制 LED0 闪烁，用来提示系统正在运行。

task2_task: 列表和列表项操作任务，调用列表和列表项相关的 API 函数，并且通过串口输出相应的信息来观察这些 API 函数的运行过程。

实验需要用到 KEY_UP 按键，用于控制任务的运行。

3、实验工程

FreeRTOS 实验 7-1 FreeRTOS 列表项的插入和删除实验。

4、实验程序与分析

● **任务设置**

实验中任务优先级、堆栈大小和任务句柄等的设置如下：

```
#define START_TASK_PRIO          1        //任务优先级
#define START_STK_SIZE           128      //任务堆栈大小
TaskHandle_t StartTask_Handler;           //任务句柄
void start_task(void *pvParameters);      //任务函数

#define TASK1_TASK_PRIO          2        //任务优先级
#define TASK1_STK_SIZE           128      //任务堆栈大小
TaskHandle_t Task1Task_Handler;           //任务句柄
void task1_task(void *pvParameters);      //任务函数
```

```
#define LIST_TASK_PRIO          3       //任务优先级
#define LIST_STK_SIZE           128     //任务堆栈大小
TaskHandle_t ListTask_Handler;          //任务句柄
void list_task(void *pvParameters);     //任务函数
```

● **列表和列表项的定义**

```
//定义一个测试用的列表和 3 个列表项
List_t TestList;                        //测试用列表
ListItem_t ListItem1;                   //测试用列表项 1
ListItem_t ListItem2;                   //测试用列表项 2
ListItem_t ListItem3;                   //测试用列表项 3
```

● **main()函数**

```
int main(void)
{
    HAL_Init();                         //初始化 HAL 库
    Stm32_Clock_Init(360,25,2,8);       //设置时钟,180Mhz
    delay_init(180);                    //初始化延时函数
    uart_init(115200);                  //初始化串口
    LED_Init();                         //初始化 LED
    KEY_Init();                         //初始化按键
    SDRAM_Init();                       //初始化 SDRAM
    LCD_Init();                         //初始化 LCD

    POINT_COLOR = RED;
    LCD_ShowString(30,10,200,16,16,"Apollo STM32F4/F7");
    LCD_ShowString(30,30,200,16,16,"FreeRTOS Examp 7-1");
    LCD_ShowString(30,50,200,16,16,"list and listItem");
    LCD_ShowString(30,70,200,16,16,"ATOM@ALIENTEK");
    LCD_ShowString(30,90,200,16,16,"2016/10/9");
    //创建开始任务
    xTaskCreate((TaskFunction_t     )start_task,            //任务函数
                (const char*        )"start_task",          //任务名称
                (uint16_t           )START_STK_SIZE,        //任务堆栈大小
                (void*              )NULL,                  //传递给任务函数的参数
                (UBaseType_t        )START_TASK_PRIO,       //任务优先级
                (TaskHandle_t*      )&StartTask_Handler);   //任务句柄
    vTaskStartScheduler();                                  //开启任务调度
}
```

● **任务函数**

任务函数 start_task()和 task1_task()都比较简单，这里为了缩减篇幅就不列出来了，重点看一下任务函数 list_task()，函数如下：

```
//list 任务函数
void list_task(void *pvParameters)
{
    //第一步：初始化列表和列表项
    vListInitialise(&TestList);
    vListInitialiseItem(&ListItem1);
    vListInitialiseItem(&ListItem2);
    vListInitialiseItem(&ListItem3);

    ListItem1.xItemValue=40;            //ListItem1 列表项值为 40
    ListItem2.xItemValue=60;            //ListItem2 列表项值为 60
    ListItem3.xItemValue=50;            //ListItem3 列表项值为 50

    //第二步：打印列表和其他列表项的地址
    printf("/**********列表和列表项地址**********/\r\n");
    printf("项目                                地址      \r\n");
    printf("TestList                            %#x       \r\n",(int)&TestList);
    printf("TestList->pxIndex                   %#x       \r\n",(int)TestList.pxIndex);
    printf("TestList->xListEnd                  %#x       \r\n",(int)(&TestList.xListEnd));
    printf("ListItem1                           %#x       \r\n",(int)&ListItem1);
    printf("ListItem2                           %#x       \r\n",(int)&ListItem2);
    printf("ListItem3                           %#x       \r\n",(int)&ListItem3);
    printf("/****************结束****************/\r\n");
    printf("按下 KEY_UP 键继续!\r\n\r\n\r\n");
    while(KEY_Scan(0)!=WKUP_PRES) delay_ms(10);  //等待 KEY_UP 键按下

    //第三步：向列表 TestList 添加列表项 ListItem1，并通过串口打印所有
    //列表项中成员变量 pxNext 和 pxPrevious 的值，通过这两个值观察列表
    //项在列表中的连接情况。
    vListInsert(&TestList,&ListItem1);          //插入列表项 ListItem1
    printf("/*********添加列表项 ListItem1**********/\r\n");
    printf("项目                                地址      \r\n");
    printf("TestList->xListEnd->pxNext          %#x       \r\n",(int)(TestList.xListEnd.pxNext));
    printf("ListItem1->pxNext                   %#x       \r\n",(int)(ListItem1.pxNext));
    printf("/**********前后向连接分割线**********/\r\n");
    printf("TestList->xListEnd->pxPrevious      %#x       \r\n",(int)(TestList.xListEnd.pxPrevious));
    printf("ListItem1->pxPrevious               %#x       \r\n",(int)(ListItem1.pxPrevious));
    printf("/****************结束****************/\r\n");
    printf("按下 KEY_UP 键继续!\r\n\r\n\r\n");
    while(KEY_Scan(0)!=WKUP_PRES) delay_ms(10);  //等待 KEY_UP 键按下

    //第四步：向列表 TestList 添加列表项 ListItem2，并通过串口打印所有
    //列表项中成员变量 pxNext 和 pxPrevious 的值，通过这两个值观察列表
```

```
//项在列表中的连接情况。
vListInsert(&TestList,&ListItem2);     //插入列表项 ListItem2
printf("/*********添加列表项 ListItem2**********/\r\n");
printf("项目                                地址      \r\n");
printf("TestList->xListEnd->pxNext          %#x       \r\n",(int)(TestList.xListEnd.pxNext));
printf("ListItem1->pxNext                   %#x       \r\n",(int)(ListItem1.pxNext));
printf("ListItem2->pxNext                   %#x       \r\n",(int)(ListItem2.pxNext));
printf("/***********前后向连接分割线**********/\r\n");
printf("TestList->xListEnd->pxPrevious      %#x       \r\n",(int)(TestList.xListEnd.pxPrevious));
printf("ListItem1->pxPrevious               %#x       \r\n",(int)(ListItem1.pxPrevious));
printf("ListItem2->pxPrevious               %#x       \r\n",(int)(ListItem2.pxPrevious));
printf("/****************结束****************/\r\n");
printf("按下 KEY_UP 键继续!\r\n\r\n\r\n");
while(KEY_Scan(0)!=WKUP_PRES) delay_ms(10);  //等待 KEY_UP 键按下

//第五步：向列表 TestList 添加列表项 ListItem3，并通过串口打印所有
//列表项中成员变量 pxNext 和 pxPrevious 的值，通过这两个值观察列表
//项在列表中的连接情况。
vListInsert(&TestList,&ListItem3);     //插入列表项 ListItem3
printf("/*********添加列表项 ListItem3**********/\r\n");
printf("项目                                地址      \r\n");
printf("TestList->xListEnd->pxNext          %#x       \r\n",(int)(TestList.xListEnd.pxNext));
printf("ListItem1->pxNext                   %#x       \r\n",(int)(ListItem1.pxNext));
printf("ListItem3->pxNext                   %#x       \r\n",(int)(ListItem3.pxNext));
printf("ListItem2->pxNext                   %#x       \r\n",(int)(ListItem2.pxNext));
printf("/**********前后向连接分割线**********/\r\n");
printf("TestList->xListEnd->pxPrevious      %#x       \r\n",(int)(TestList.xListEnd.pxPrevious));
printf("ListItem1->pxPrevious               %#x       \r\n",(int)(ListItem1.pxPrevious));
printf("ListItem3->pxPrevious               %#x       \r\n",(int)(ListItem3.pxPrevious));
printf("ListItem2->pxPrevious               %#x       \r\n",(int)(ListItem2.pxPrevious));
printf("/****************结束****************/\r\n");
printf("按下 KEY_UP 键继续!\r\n\r\n\r\n");
while(KEY_Scan(0)!=WKUP_PRES) delay_ms(10);  //等待 KEY_UP 键按下

//第六步：删除 ListItem2，并通过串口打印所有列表项中成员变量 pxNext 和
//pxPrevious 的值，通过这两个值观察列表项在列表中的连接情况。
uxListRemove(&ListItem2);                    //删除 ListItem2
printf("/**********删除列表项 ListItem2*********/\r\n");
printf("项目                                地址      \r\n");
printf("TestList->xListEnd->pxNext          %#x       \r\n",(int)(TestList.xListEnd.pxNext));
printf("ListItem1->pxNext                   %#x       \r\n",(int)(ListItem1.pxNext));
printf("ListItem3->pxNext                   %#x       \r\n",(int)(ListItem3.pxNext));
printf("/***********前后向连接分割线**********/\r\n");
```

```
    printf("TestList->xListEnd->pxPrevious     %#x     \r\n",(int)(TestList.xListEnd.pxPrevious));
    printf("ListItem1->pxPrevious              %#x     \r\n",(int)(ListItem1.pxPrevious));
    printf("ListItem3->pxPrevious              %#x     \r\n",(int)(ListItem3.pxPrevious));
    printf("/****************结束****************/\r\n");
    printf("按下 KEY_UP 键继续!\r\n\r\n\r\n");
    while(KEY_Scan(0)!=WKUP_PRES) delay_ms(10);  //等待 KEY_UP 键按下

    //第七步：删除 ListItem2，并通过串口打印所有列表项中成员变量 pxNext 和
    //pxPrevious 的值，通过这两个值观察列表项在列表中的连接情况。
    TestList.pxIndex=TestList.pxIndex->pxNext;//pxIndex 向后移一项，
                                              //这样 pxIndex 就会指向 ListItem1。
    vListInsertEnd(&TestList,&ListItem2);     //列表末尾添加列表项 ListItem2
    printf("/******在末尾添加列表项 ListItem2*******/\r\n");
    printf("项目                               地址     \r\n");
    printf("TestList->pxIndex                  %#x     \r\n",(int)TestList.pxIndex);
    printf("TestList->xListEnd->pxNext         %#x     \r\n",(int)(TestList.xListEnd.pxNext));
    printf("ListItem2->pxNext                  %#x     \r\n",(int)(ListItem2.pxNext));
    printf("ListItem1->pxNext                  %#x     \r\n",(int)(ListItem1.pxNext));
    printf("ListItem3->pxNext                  %#x     \r\n",(int)(ListItem3.pxNext));
    printf("/***********前后向连接分割线**********/\r\n");
    printf("TestList->xListEnd->pxPrevious     %#x     \r\n",(int)(TestList.xListEnd.pxPrevious));
    printf("ListItem2->pxPrevious              %#x     \r\n",(int)(ListItem2.pxPrevious));
    printf("ListItem1->pxPrevious              %#x     \r\n",(int)(ListItem1.pxPrevious));
    printf("ListItem3->pxPrevious              %#x     \r\n",(int)(ListItem3.pxPrevious));
    printf("/****************结束****************/\r\n\r\n\r\n");
    while(1)
    {
        LED1=!LED1;
        vTaskDelay(1000);             //延时 1s，也就是 1000 个时钟节拍
    }
}
```

任务函数 list_task()通过调用与列表和列表项相关的 API 函数来对列表和列表项做相应的操作，并且通过串口打印出每调用一个 API 函数以后列表和列表项的连接信息，通过这些信息我们可以直观的判断出列表项在插入、删除和末尾插入的时候这个列表的变化情况。

7.7.2 程序运行结果分析

编译并下载实验代码到开发板中，打开串口调试助手，然后按照任务函数 list_task()中的步骤一步步的测试分析。

● **第一步和第二步**

第一步和第二步是用来初始化列表和列表项的，并且通过串口输出列表和列表项的地址，这一步是开发板复位后默认运行的，串口调试助手信息如下所示：

```
 LCD ID:5510
/*******************列表和列表项地址*******************/
项目                            地址
TestList                        0x200000bc
TestList->pxIndex               0x200000c4
TestList->xListEnd              0x200000c4
ListItem1                       0x200000d0
ListItem2                       0x200000e4
ListItem3                       0x200000f8
/*************************结束*************************/
按下KEY_UP键继续!
```

图 7.7.2.1 串口调试助手输出信息

由于这些列表和列表项地址前六位都为 0X200000，只有最低 2 位不同，所以我们就用最低 2 位代表这些列表和列表项的地址。注意！列表和列表项的地址在不同的硬件平台或编译软件上是不同的，以自己的实际实验结果为准！简单的分析一下图 7.7.2.1 可以得到如下信息：

1、列表 TestList 地址为 bc。

2、列表项 ListItem1、ListItem2 和 ListItem3 的地址分别为 d0、e4 和 f8。

3、列表 TestList 的 xListEnd 地址为 c4。

4、列表 TestList 的 pxIndex 指向地址 c4，而这个地址正是迷你列表项 xListEnd，说明 pxIndex 指向 xListEnd，这个和我们分析列表初始化函数 vListInitialise()的时候得到的结果是一致的。

● **第三步**

按一下 KEY_UP 键，执行第三步，第三步是向列表 TestList 中插入列表项 ListItem1，列表项 ListItem1 的成员变量 xItemValue 的值为 40。第三步执行完以后串口调试助手输出如图 7.7.2.2 所示：

```
/*******************添加列表项ListItem1*******************/
项目                            地址
TestList->xListEnd->pxNext      0x200000d0
ListItem1->pxNext               0x200000c4
/*******************前后向连接分割线*********************/
TestList->xListEnd->pxPrevious  0x200000d0
ListItem1->pxPrevious           0x200000c4
/*************************结束*************************/
按下KEY_UP键继续!
```

图 7.7.2.2 串口调试助手输出信息

分析图 7.7.2.2 可以得出一下信息：

1、xListEnd 的 pxNext 指向地址 d0，而 d0 是 ListItem1 的地址，说明 xListEnd 的 pxNext 指向 ListItem1。

2、列表项 ListItem1 的 pxNext 指向地址 c4，而 c4 是 xListEnd 的地址，说明 ListItem1 的 pxNext 指向 xListEnd。

3、xListEnd 的 pxPrevious 指向地址 d0，而 d0 是 ListItem1 的地址，说明 xListEnd 的 pxPrevious 指向 ListItem2。

4、ListItem1 的 pxPrevious 指向地址 c4，c4 是 xListEnd 的地址，说明 ListItem1 的 pxPrevious 指向 xListEnd。

用简易示意图表示这一步的话如图 7.7.2.3 所示：

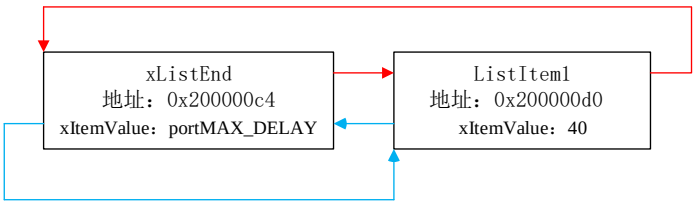


图 7.7.2.3 简易示意图

●第四步

按一下 KEY_UP 键，执行第四步，第四步是向列表 TestList 中插入列表项 ListItem2，列表项 ListItem2 的成员变量 xItemValue 的值为 60。第四步执行完以后串口调试助手输出如图 7.7.2.4 所示：

```
/******************添加列表项ListItem2*****************/
项目                                地址
TestList->xListEnd->pxNext          0x200000d0
ListItem1->pxNext                   0x200000e4
ListItem2->pxNext                   0x200000c4
/********************前后向连接分割线*********************/
TestList->xListEnd->pxPrevious      0x200000e4
ListItem1->pxPrevious               0x200000c4
ListItem2->pxPrevious               0x200000d0
/************************结束*************************/
按下KEY_UP键继续!
```

图 7.7.2.4 串口调试助手输出信息

分析图 7.7.2.4 可以得出一下信息：

1、xListEnd 的 pxNext 指向 ListItem1。

2、ListItem1 的 pxNext 指向 ListItem2。

3、ListItem2 的 pxNext 指向 xListEnd。

4、列表项的 pxPrevious 分析过程类似，后面的步骤中就不做分析了，只看 pxNext 成员变量。用简易示意图表示这一步的话如图 7.7.2.5 所示：

图 7.7.2.5 简易示意图

●第五步

按一下 KEY_UP 键，执行第五步，第五步是向列表 TestList 中插入列表项 ListItem3，列表项 ListItem3 的成员变量 xItemValue 的值为 50。第四步执行完以后串口调试助手输出如图 7.7.2.6 所示：

```
/******************添加列表项ListItem3*****************/
项目                                地址
TestList->xListEnd->pxNext          0x200000d0
ListItem1->pxNext                   0x200000f8
ListItem3->pxNext                   0x200000e4
ListItem2->pxNext                   0x200000c4
/********************前后向连接分割线*********************/
TestList->xListEnd->pxPrevious      0x200000e4
ListItem1->pxPrevious               0x200000c4
ListItem3->pxPrevious               0x200000d0
ListItem2->pxPrevious               0x200000f8
/************************结束*************************/
按下KEY_UP键继续!
```

图 7.7.2.6 串口调试助手输出信息

分析图 7.7.2.6 可以得出一下信息：

1、xListEnd 的 pxNext 指向 ListItem1。

2、ListItem1 的 pxNext 指向 ListItem3。

3、ListItem3 的 pxNext 指向 ListItem2。

4、ListItem2 的 pxNext 指向 xListEnd。

用简易示意图表示这一步的话如图 7.7.2.7 所示：

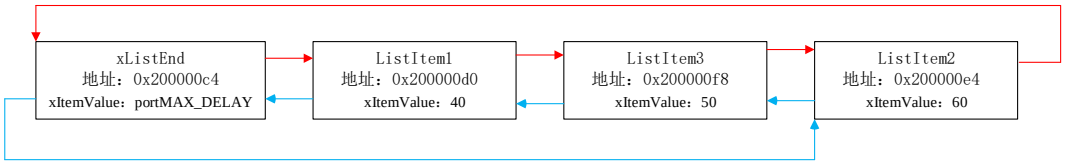


图 7.7.2.7 简易示意图

通过这几步可以看出列表项的插入是根据 xItemValue 的值做升序排列的，这个和我们分析函数 vListInsert()得到的结果一样，说明我们的分析是对的！

●第六步和第七步

这两步是观察函数 uxListRemove()和 vListInsertEnd()的运行过程的，分析过程和前五步一样。这里就不做分析了，大家自行根据串口调试助手输出的信息做分析。

第八章 FreeRTOS 调度器开启和任务相关函数详解

我们已经学会了 FreeRTOS 的任务创建和删除，挂起和恢复等基本操作，并且也学习了分析 FreeRTOS 源码所必须掌握的知识：列表和列表项。但是任务究竟如何被创建、删除、挂起和恢复的？系统是怎么启动的等等这些我们还不了解，一个操作系统最核心的内容就是多任务管理，所以我们非常有必要去学习一下 FreeRTOS 的任务创建、删除、挂起、恢复和系统启动等，这样才能对 FreeRTOS 有一个更深入的了解。本章分为如下几部分：

8.1 阅读本章所必备的知识
8.2 调度器开启过程分析
8.3 任务创建过程分析
8.4 任务删除过程分析
8.5 任务挂起过程分析
8.6 任务恢复过程分析

8.1 阅读本章所必备的知识

本章和下一章要讲解的内容和 Cortex-M 处理器的内核架构联系非常紧密！阅读本章必须先对 Cortex-M 处理器的架构有一定的了解，在学习本章的时候一定要配合《权威指南》来学习，推荐大家仔细阅读《权威指南》中的如下章节：

1、第 3 章 技术综述，通过阅读本章可以对 Cortex-M 处理器的架构有一个大体的了解。

2、第 4 章 架构，强烈建议仔细阅读本章内容，尤其是要理解其中讲解到的各个寄存器。

3、第 5 章 指令集，本章和下一章的内容会涉及到一些有关 ARM 的汇编指令，在阅读的时遇到不懂的指令可以查阅《权威指南》的第 5 章中相关指令的讲解。

4、第 7 章 异常和中断，大概了解一下 。

5、第 8 章 深入了解异常处理，强烈建议仔细阅读！

6、第 10 章 OS 支持特性， 强烈建议仔细阅读！

《权威指南》中的其他章节大家依据个人爱好来阅读，由于《权威指南》讲解的内容非常的“底层”，所以看起来可能会感觉晦涩难懂，如果看不懂的话不要着急，看不懂的地方就跳过，先对 Cortex-M 的处理器有一个大概的了解就行了。笔者第一次看宋岩翻译的那本《ARM Cortex-M3 权威指南》的时候就一点都没看懂，在后面的工作中因为工作需要才硬着头皮看的，不知道看了多少遍，反正书已经翻烂了，现在看第三版的《权威指南》估计也就能看懂个 40%~50%吧。

8.2 调度器开启过程分析

在本节中会涉及到 ARM 的汇编指令，有关涉及到的 ARM 指令的详细使用情况请参考《权威指南》的“第 5 章 指令集”。《权威指南》的这一章节对 Cortex-M3/M4 内核的所有指令做了非常详细的接收，包括指令的含义、使用方法和参考案例等等。

8.2.1 任务调度器开启函数分析

前面的所有例程中我们都是在 main()函数中先创建一个开始任务 start_task，后面紧接着调用函数 vTaskStartScheduler()。这个函数的功能就是开启任务调度器的，这个函数在文件 tasks.c 中有定义，缩减后的函数代码如下：

```
void vTaskStartScheduler( void )
{
    BaseType_t xReturn;

    xReturn = xTaskCreate( prvIdleTask,                                    (1)
                           "IDLE", configMINIMAL_STACK_SIZE,
                           ( void * ) NULL,
                           ( tskIDLE_PRIORITY | portPRIVILEGE_BIT ),
                           &xIdleTaskHandle );

    #if ( configUSE_TIMERS == 1 )              //使用软件定时器使能
    {
        if( xReturn == pdPASS )
        {
            xReturn = xTimerCreateTimerTask();                             (2)
        }
```

```
        else
        {
            mtCOVERAGE_TEST_MARKER();
        }
    }
    #endif /* configUSE_TIMERS */

    if( xReturn == pdPASS )              //空闲任务和定时器任务创建成功。
    {
        portDISABLE_INTERRUPTS();                                          (3)

        #if ( configUSE_NEWLIB_REENTRANT == 1 )      //使能 NEWLIB
        {
            _impure_ptr = &( pxCurrentTCB->xNewLib_reent );
        }
        #endif /* configUSE_NEWLIB_REENTRANT */

        xNextTaskUnblockTime = portMAX_DELAY;
        xSchedulerRunning = pdTRUE;                                        (4)
        xTickCount = ( TickType_t ) 0U;

        portCONFIGURE_TIMER_FOR_RUN_TIME_STATS();                          (5)

        if( xPortStartScheduler() != pdFALSE )                             (6)
        {
            //如果调度器启动成功的话就不会运行到这里，函数不会有返回值的
        }
        else
        {
            //不会运行到这里，除非调用函数 xTaskEndScheduler()。
        }
    }
    else
    {
        //程序运行到这里只能说明一点，那就是系统内核没有启动成功，导致的原因是在创建
        //空闲任务或者定时器任务的时候没有足够的内存。
        configASSERT( xReturn != errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY );
    }

    //防止编译器报错，比如宏 INCLUDE_xTaskGetIdleTaskHandle 定义为 0 的话编译器就会提
    //示 xIdleTaskHandle 未使用。
    ( void ) xIdleTaskHandle;
}
```

(1)、创建空闲任务，如果使用静态内存的话使用函数 xTaskCreateStatic()来创建空闲任务，优先级为 tskIDLE_PRIORITY，宏 tskIDLE_PRIORITY 为 0，也就是说空闲任务的优先级为最低。

(2)、如果使用软件定时器的话还需要通过函数 xTimerCreateTimerTask()来创建定时器服务任务。定时器服务任务的具体创建过程是在函数 xTimerCreateTimerTask()中完成的，这个函数很简单，大家就 自行查阅一下。

(3)、关闭中断，在 SVC 中断服务函数 vPortSVCHandler()中会打开中断。

(4)、变量 xSchedulerRunning 设置为 pdTRUE，表示调度器开始运行。

(5)、当宏 configGENERATE_RUN_TIME_STATS 为 1 的时候说明使能时间统计功能，此时需要用户实现宏 portCONFIGURE_TIMER_FOR_RUN_TIME_STATS，此宏用来配置一个定时器/计数器。

(6)、调用函数 xPortStartScheduler()来初始化跟调度器启动有关的硬件，比如滴答定时器、FPU 单元和 PendSV 中断等等。

8.2.2 内核相关硬件初始化函数分析

FreeRTOS 系统时钟是由滴答定时器来提供的，而且任务切换也会用到 PendSV 中断，这些硬件的初始化由函数 xPortStartScheduler()来完成，缩减后的函数代码如下：

```
BaseType_t xPortStartScheduler( void )
{
    /*************************************************************************/
    /***************此处省略一大堆的条件编译代码*********************/
    /*************************************************************************/
    portNVIC_SYSPRI2_REG |= portNVIC_PENDSV_PRI;                (1)
    portNVIC_SYSPRI2_REG |= portNVIC_SYSTICK_PRI;               (2)
    vPortSetupTimerInterrupt();                                 (3)
    uxCriticalNesting = 0;                                      (4)
    prvEnableVFP();                                             (5)
    *( portFPCCR ) |= portASPEN_AND_LSPEN_BITS;                 (6)
    prvStartFirstTask();                                        (7)

    //代码正常执行的话是不会到这里的！
    return 0;
}
```

(1)、设置 PendSV 的中断优先级，为最低优先级。

(2)、设置滴答定时器的中断优先级，为最低优先级。

(3)、调用函数 vPortSetupTimerInterrupt()来设置滴答定时器的定时周期，并且使能滴答定时器的中断，函数比较简单，大家自行查阅分析。

(4)、初始化临界区嵌套计数器。

(5)、调用函数 prvEnableVFP()使能 FPU。

(6)、设置寄存器 FPCCR 的 bit31 和 bit30 都为 1，这样 S0~S15 和 FPSCR 寄存器在异常入口和退出时的壮态自动保存和恢复。并且异常流程使用惰性压栈的特性以保证中断等待。关于 FPCCR 寄存器和惰性压栈的知识请参考《权威指南》的“第 12 章 浮点运算”。

8.2.3 使能 FPU 函数分析

在函数 xPortStartScheduler()中会通过调用 prvEnableVFP()来使能 FPU，这个函数是汇编形式的，在文件 port.c 中有定义，函数如下：

```
__asm void prvEnableVFP( void )
{
    PRESERVE8

    ldr.w r0, =0xE000ED88           ;R0=0XE000ED88                  (1)
    ldr   r1, [r0]                  ;从 R0 的地址读取数据赋给 R1     (2)

    orr   r1, r1, #( 0xf << 20 )    ;R1=R1|(0xf<<20)                (3)
    str r1, [r0]                    ;R1 中的值写入 R0 保存的地址中   (4)
    bx    r14                                                       (5)
    nop
}
```

(1)、利用寄存器 CPACR 可以使能或禁止 FPU，此寄存器的地址为 0XE000ED88(具体参考《权威指南》“第 13 章 浮点运算”13.2.3 章节)，此寄存器的 CP10(bit20 和 bit21)和 CP11(bit22 和 bit23)用于控制 FPU。这 4 个 bit 的具体含义请参考《权威指南》，通常将这 4 个 bit 都设置为 1 来开启 FPU，表示全访问。此行代码将地址 0XE000ED88 保存在寄存器 R0 中。

(2)、读取 R0 中保存的存储地址处的数据，也就是 CPACR 寄存器的值，并将结果保存在 R1 寄存器中。

(3)、R1 中的值与(0xf<<20)进行按位或运算，也就是 R1=R1|0X00F00000。此时 R1 所保存的值的 bit20~bit23 就都为 1 了，将这个值写入寄存器 CPACR 中就可开启 FPU。

(4)、将 R1 中的值写入 R0 中保存的地址处，也就是寄存器 CPACR 中。

(5)、函数返回。bx 为间接跳转指令，一般为 BX <Rm>，也就是跳转到存放在 Rm 中的地址处，此处是跳转到 R14 存放的地址处。R14 寄存器也叫做链接寄存(LR)，也可以用 LR 表示。这个寄存器用于函数或子程序调用时返回地址的保存。关于此寄存器的详细介绍请参考《权威指南》“第 4 章 架构”的 4.2.2 章节。

8.2.4 启动第一个任务

经过上面的操作以后我们就可以启动第一个任务了，函数 prvStartFirstTask()用于启动第一个任务，这是一个汇编函数，函数源码如下：

```
__asm void prvStartFirstTask( void )
{
    PRESERVE8

    ldr r0, =0xE000ED08     ;R0=0XE000ED08                  (1)
    ldr r0, [r0]            ;取 R0 所保存的地址处的值赋给 R0 (2)
    ldr r0, [r0]            ;获取 MSP 初始值                (3)

    msr msp, r0             ;复位 MSP                       (4)
```

```
    cpsie I                 ;使能中断(清除 PRIMASK)          (5)
    cpsie f                 ;使能中断(清除 FAULTMASK)        (6)
    dsb                     ;数据同步屏障                    (7)
    isb                     ;指令同步屏障                    (8)

    svc 0                   ;触发 SVC 中断(异常)             (9)
    nop
    nop
}
```

(1)、将 0XE000ED08 保存在寄存器 R0 中。一般来说向量表应该是从起始地址(0X00000000)开始存储的，不过，有些应用可能需要在运行时修改或重定义向量表，Cortex-M 处理器为此提供了一个叫做向量表重定位的特性。向量表重定位特性提供了一个名为向量表偏移寄存器(VTOR)的可编程寄存器。VTOR 寄存器的地址就是 0XE000ED08，通过这个寄存器可以重新定义向量表，比如在 STM32F429 的 ST 官方库中会通过函数 SystemInit()来设置 VTOR 寄存器，代码如下：

```
SCB->VTOR = FLASH_BASE | VECT_TAB_OFFSET;    //VTOR=0x08000000+0X00
```

通过上面一行代码就将向量表开始地址重新定义到了 0X08000000，向量表的起始地址存储的就是 MSP 初始值。关于向量表和向量表重定位的详细内容请参阅《权威指南》的“第 7 章 异常和中断”的 7.5 小节。

(2)、读取 R0 中存储的地址处的数据并将其保存在 R0 寄存器，也就是读取寄存器 VTOR 中的值，并将其保存在 R0 寄存器中。这一行代码执行完就以后 R0 的值应该为 0X08000000。

(3)、读取 R0 中存储的地址处的数据并将其保存在 R0 寄存器，也就是读取地址 0X08000000 处存储的数据，并将其保存在 R0 寄存器中。我们知道向量表的起始地址保存的就是主栈指针 MSP 的初始值，这一行代码执行完以后寄存器 R0 就存储 MSP 的初始值。现在来看(1)、(2)、(3)这三步起始就是为了获取 MSP 的初始值而已！

(4)、复位 MSP，R0 中保存了 MSP 的初始值，将其赋值给 MSP 就相当于复位 MSP。

(5)和(6)、使能中断，关于这两个指令的详细内容请参考《权威指南》的“第 4 章 架构”的第 4.2.3 小节。

(7)和(8)、数据同步和指令同步屏障，这两个指令的详细内容请参考《权威指南》的“第 5 章 指令集”的 5.6.13 小节。

(9)，调用 SVC 指令触发 SVC 中断，SVC 也叫做请求管理调用，SVC 和 PendSV 异常对于 OS 的设计来说非常重要。SVC 异常由 SVC 指令触发。关于 SVC 的详细内容请参考《权威指南》的“第 10 章 OS 支持特性”的 10.3 小节。在 FreeRTOS 中仅仅使用 SVC 异常来启动第一个任务，后面的程序中就再也用不到 SVC 了。

8.2.5 SVC 中断服务函数

在函数 prvStartFirstTask()中通过调用 SVC 指令触发了 SVC 中断，而第一个任务的启动就是在 SVC 中断服务函数中完成的，SVC 中断服务函数应该为 SVC_Handler()，但是 FreeRTOSConfig.h 中通过#define 的方式重新定义为了 xPortPendSVHandler()，如下：

```
#define xPortPendSVHandler      PendSV_Handler
```

函数 vPortSVCHandler()在文件 port.c 中定义，这个函数也是用汇编写的，函数源码如下：

```
__asm void vPortSVCHandler( void )
{
```

```
    PRESERVE8

    ldr   r3, =pxCurrentTCB      ;R3=pxCurrentTCB 的地址                (1)
    ldr r1, [r3]                 ;取 R3 所保存的地址处的值赋给 R1       (2)
    ldr r0, [r1]                 ;取 R1 所保存的地址处的值赋给 R0       (3)

    ldmia r0!, {r4-r11, r14}     ;出栈 ， R4~R11 和 R14                 (4)
    msr psp, r0                  ;进程栈指针 PSP 设置为任务的堆栈       (5)
    isb                          ;指令同步屏障
    mov r0, #0                   ;R0=0                                  (6)
    msr  basepri, r0             ;寄存器 basepri=0，开启中断            (7)
    bx r14                                                              (8)
}
```

(1)、获取 pxCurrentTCB 指针的存储地址，pxCurrentTCB 是一个指向 TCB_t 的指针，这个指针永远指向正在运行的任务。这里先获取这个指针存储的地址，比如我现在的代码测试出来这个指针是存放在 0X2000007C，如图 8.2.5.1 所示。

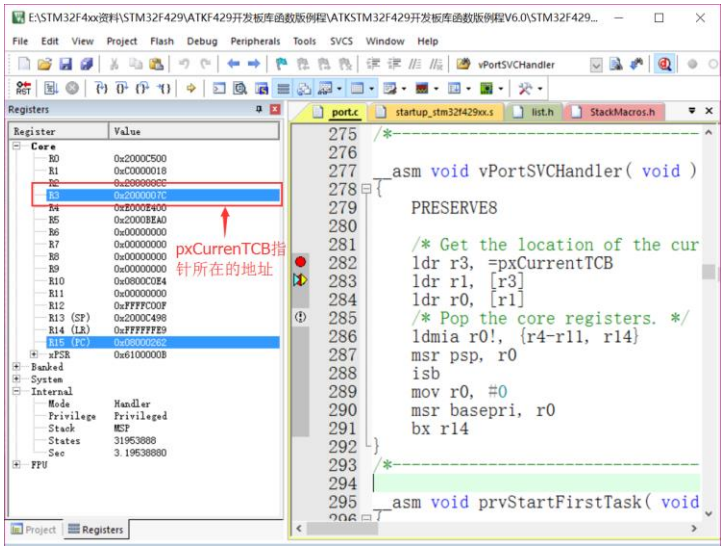


图 8.2.5.1 pxCurrentTCB 指针所在位置

(2)、取 R3 所保存的地址处的值赋给 R1。通过这一步就获取到了当前任务的任务控制块的存储地址。比如当前我的程序中这个地址就为 0X20001058，如图 8.2.5.2 所示：

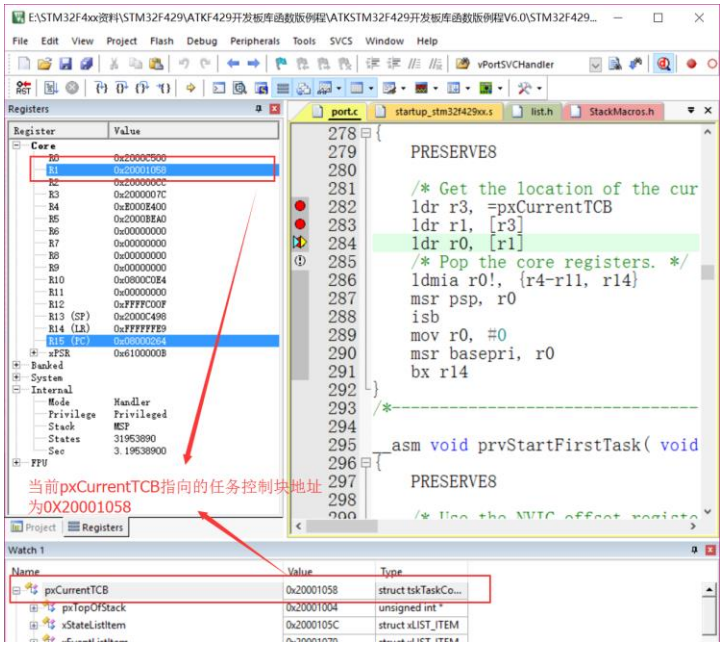


图 8.2.5.2 pxCurrentTCB 指向的 TCB 地址

(3)、取 R3 所保存的地址处的值赋给 R0，我们知道任务控制块的第一个字段就是任务堆栈的栈顶指针 pxTopOfStack 所指向的位置，所以读取任务控制块所在的首地址(0X20001058)得到的就是栈顶指针所指向的地址，当前我的程序中这个栈顶指针(pxTopOfStack)所指向的地址为 0X20001004，如图 8.2.5.3 所示

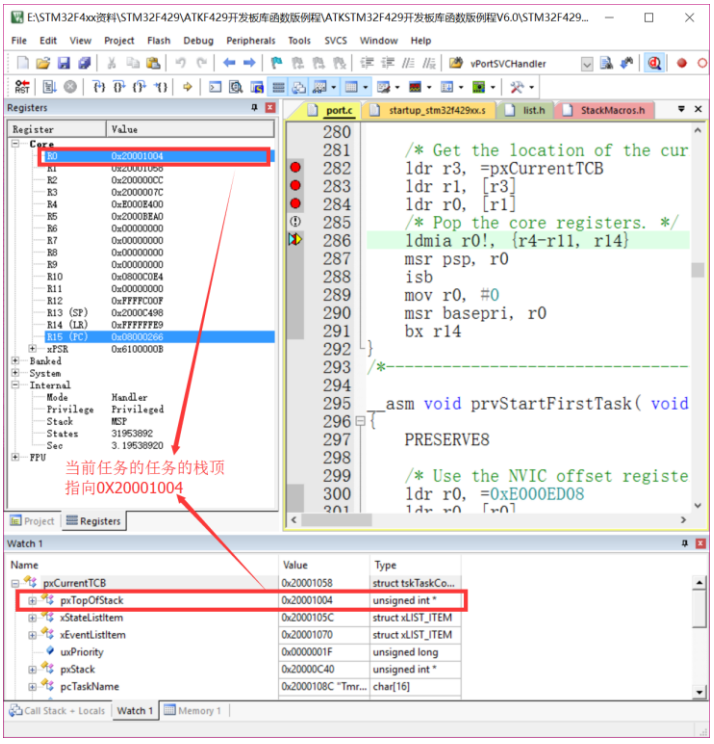


图 8.2.5.3 栈顶指针

可以看出(1)、(2)和(3)的目的就是获取要切换到的这个任务的任务栈顶指针，因为任务所对应的寄存器值，也就是现场都保存在任务的任务堆栈中，所以需要获取栈顶指针来恢复这些寄存器值！

(4)、R4~R11，R14 这些寄存器出栈。这里使用了指令 LDMIA，LDMIA 指令是多加载/存储指令，不过这里使用的是具有回写的多加载/存储访问指令，用法如下：

```
LDMIA   Rn! ,      {reg list}
```

表示从 Rn 指定的存储器位置读取多个字，地址在每次读取后增加(IA),Rn 在传输完成以后写回。对于 STM32 来说地址一次增加 4 字节，比如如下代码：

```
LDR      R0,  =0X800
LDMIA   R0!, {R2~R4}
```

上面两行代码就是将 0X800 地址的数据赋值给寄存器 R2，0X804 地址的数据赋值给寄存器 R3，0X8008 地址的数据赋值给 R4 寄存器，然后，重点来了！此时 R0 为 800A！

通过这一步我们就从任务堆栈中将 R4~R11,R14 这几个寄存器的值给恢复了，注意 R14 的值为 0XFFFFFFFD，这个值就是我们在初始化任务堆栈的时候保存的 EXC_RETURN 的值！如图 8.2.5.4 所示：

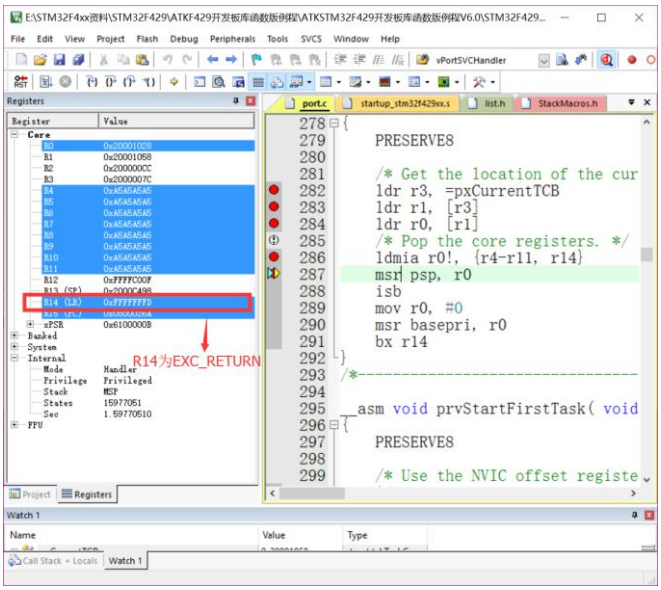


图 8.2.5.4 R14 的值

这里有朋友就要问了，R0~R3，R12，PC,xPSR 这些寄存器怎么没有恢复？这是因为这些寄存器会在退出中断的时候 MCU 自动出栈(恢复)的，而 R4~R11 需要由用户手动出栈。如果使用 FPU 的话还要考虑到 FPU 寄存器，这个我们在分析 PendSV 中断服务函数的时候会讲到。到这步以后我们来看一下堆栈的栈顶指针指到哪里了？如图 8.2.5.5 所示：

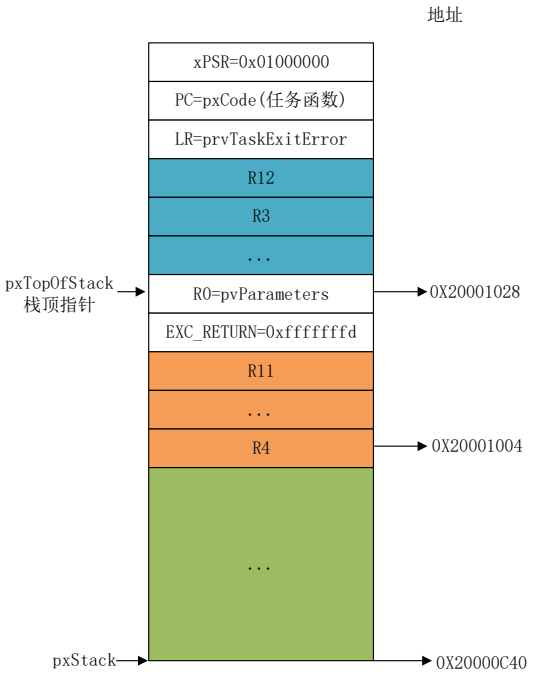


图 8.2.5.5 栈顶指针新值

从图 8.2.5.5 可以看出恢复 R4~R11 和 R14 以后堆栈的栈顶指针应该指向地址 0X20001028，也就是保存寄存器 R0 值的存储地址。退出中断服务函数以后进程栈指针 PSP 应该从这个地址开始恢复其他的寄存器值。

(5)、设置进程栈指针 PSP，PSP=R0=0X20001028，如图 8.2.5.6 所示：

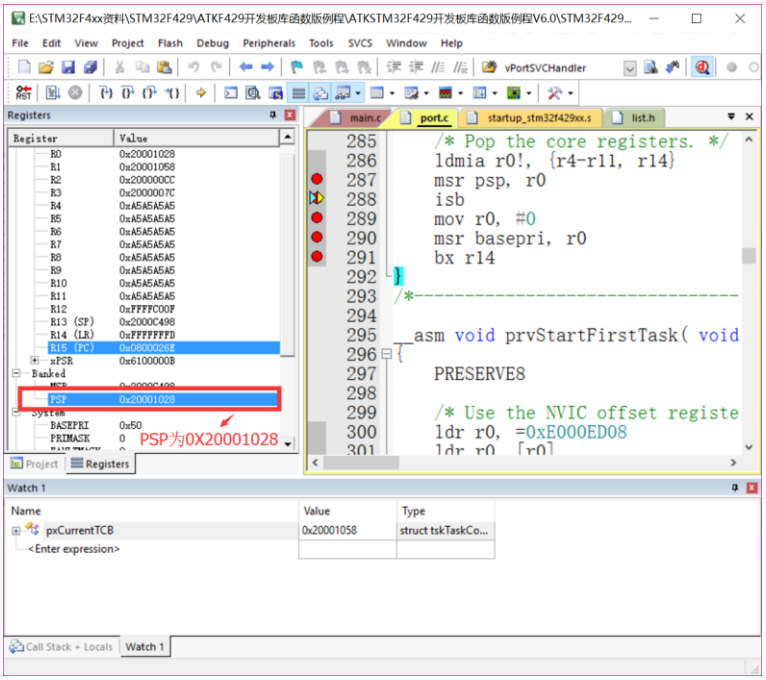


图 8.2.5.6 设置进程栈指针 PSP 值

(6)、设置寄存器 R0 为 0。

(7)、设置寄存器 BASEPRI 为 R0，也就是 0，打开中断！

(8)、执行此行代码以后硬件自动恢复寄存器 R0~R3、R12、LR、PC 和 xPSR 的值，堆栈使

用进程栈 PSP，然后执行寄存器 PC 中保存的任务函数。至此，FreeRTOS 的任务调度器正式开始运行！

8.2.6 空闲任务

在 8.2.1 小节讲解函数 vTaskStartScheduler()说过，此函数会创建一个名为“IDLE”的任务，这个任务叫做空闲任务。顾名思义，空闲任务就是空闲的时候运行的任务，也就是系统中其他的任务由于各种原因不能运行的时候空闲任务就在运行。空闲任务是 FreeRTOS 系统自动创建的，不需要用户手动创建。任务调度器启动以后就必须有一个任务运行！但是空闲任务不仅仅是为了满足任务调度器启动以后至少有一个任务运行而创建的，空闲任务中还会去做一些其他的事情，如下：

1、判断系统中是否有任务删除自己，如果有的话就在空闲任务中释放被删除任务的任务堆栈和任务控制块的内存。

2、运行用户设置的空闲任务钩子函数。

3、判断是否开启低功耗 tickless 模式，如果开启的话还需要做相应的处理

空闲任务的任务优先级是最低的，为 0，任务函数为 prvIdleTask()，有关空闲任务的详细内容我们后面会有专门的章节讲解，这里大家只要知道有这个任务就行了。

8.3 任务创建过程分析

8.3.1 任务创建函数分析

前面学了任务创建可以使用动态方法或静态方法(不讨论使用 MPU 的情况)，它们分别使用函数 xTaskCreate()和 xTaskCreateStatic()。本节我们就以函数 xTaskCreate()为例来分析一下 FreeRTOS 的任务创建过程，函数 xTaskCreateStatic()类似，这里不做分析。函数 xTaskCreate()代码如下，注意这里为了缩小篇幅去掉了函数中的条件编译等不重要的语句！

```
BaseType_t xTaskCreate(TaskFunction_t          pxTaskCode,
                       const char * const      pcName,
                       const uint16_t          usStackDepth,
                       void * const            pvParameters,
                       UBaseType_t             uxPriority,
                       TaskHandle_t * const    pxCreatedTask )
{
    TCB_t *pxNewTCB;
    BaseType_t xReturn;

    /*****************************************************************/
    /***************使用条件编译的向上增长堆栈相关代码省略***************/
    /*****************************************************************/

    StackType_t *pxStack;
    pxStack = ( StackType_t * ) pvPortMalloc( ( ( ( size_t ) usStackDepth ) *\          (1)
              sizeof( StackType_t ) ) );

    if( pxStack != NULL )
```

```
    {
        pxNewTCB = ( TCB_t * ) pvPortMalloc( sizeof( TCB_t ) );                    (2)
        if( pxNewTCB != NULL )
        {
            pxNewTCB->pxStack = pxStack;                                           (3)
        }
        else
        {
            vPortFree( pxStack );                                                  (4)
        }
    }
    else
    {
        pxNewTCB = NULL;
    }

    if( pxNewTCB != NULL )
    {
        #if( tskSTATIC_AND_DYNAMIC_ALLOCATION_POSSIBLE != 0 )
        {
            pxNewTCB->ucStaticallyAllocated =\                                     (5)
            tskDYNAMICALLY_ALLOCATED_STACK_AND_TCB;
        }
        #endif /* configSUPPORT_STATIC_ALLOCATION */

        prvInitialiseNewTask( pxTaskCode, pcName, ( uint32_t ) usStackDepth, \     (6)
                              pvParameters, uxPriority, pxCreatedTask, pxNewTCB, NULL );
        prvAddNewTaskToReadyList( pxNewTCB );                                      (7)
        xReturn = pdPASS;
    }
    else
    {
        xReturn = errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY;
    }

    return xReturn;
}
```

(1)、使用函数 pvPortMalloc()给任务的任务堆栈申请内存，申请内存的时候会做字节对齐处理。

(2)、如果堆栈的内存申请成功的话就接着给任务控制块申请内存，同样使用函数 pvPortMalloc()。

(3)、任务控制块内存申请成功的话就初始化内存控制块中的任务堆栈字段 pxStack，使用(1)中申请到的任务堆栈。

(4)、如果任务控制块内存申请失败的话就释放前面已经申请成功的任务堆栈的内存。

(5)、标记任务堆栈和任务控制块是使用动态内存分配方法得到的。

(6)、使用函数 prvInitialiseNewTask()初始化任务，这个函数完成对任务控制块中各个字段的初始化工作！

(7)、使用函数 prvAddNewTaskToReadyList()将新创建的任务加入到就绪列表中。

8.3.2 任务初始化函数分析

函数prvInitialiseNewTask()用于完成对任务的初始化，缩减后的函数源码如下：

```
static void prvInitialiseNewTask(  TaskFunction_t              pxTaskCode,
                                   const char * const          pcName,
                                   const uint32_t              ulStackDepth,
                                   void * const                pvParameters,
                                   UBaseType_t                 uxPriority,
                                   TaskHandle_t * const        pxCreatedTask,
                                   TCB_t *                     pxNewTCB,
                                   const MemoryRegion_t * const xRegions )
{
    StackType_t *pxTopOfStack;
    UBaseType_t x;

    #if( ( configCHECK_FOR_STACK_OVERFLOW > 1 ) || ( configUSE_TRACE_FACILITY ==\
        1 ) || ( INCLUDE_uxTaskGetStackHighWaterMark == 1 ) )
    {
        ( void ) memset( pxNewTCB->pxStack, ( int ) tskSTACK_FILL_BYTE,\        (1)
        ( size_t ) ulStackDepth * sizeof( StackType_t ) );
    }
    #endif

    pxTopOfStack = pxNewTCB->pxStack + ( ulStackDepth - ( uint32_t ) 1 );       (2)
    pxTopOfStack = ( StackType_t * ) ( ( ( portPOINTER_SIZE_TYPE ) pxTopOfStack ) &\
                    ( ~( ( portPOINTER_SIZE_TYPE ) portBYTE_ALIGNMENT_MASK ) ) );

    for( x = ( UBaseType_t ) 0; x < ( UBaseType_t ) configMAX_TASK_NAME_LEN; x++ )
    {
        pxNewTCB->pcTaskName[ x ] = pcName[ x ];                                (3)
        if( pcName[ x ] == 0x00 )
        {
            break;
        }
        else
        {
            mtCOVERAGE_TEST_MARKER();
        }
```

```
}
pxNewTCB->pcTaskName[ configMAX_TASK_NAME_LEN - 1 ] = '\0';                    (4)

if( uxPriority >= ( UBaseType_t ) configMAX_PRIORITIES )                       (5)
{
    uxPriority = ( UBaseType_t ) configMAX_PRIORITIES - ( UBaseType_t ) 1U;
}
else
{
    mtCOVERAGE_TEST_MARKER();
}
pxNewTCB->uxPriority = uxPriority;                                             (6)

#if ( configUSE_MUTEXES == 1 )                                                 (7)
{
    pxNewTCB->uxBasePriority = uxPriority;
    pxNewTCB->uxMutexesHeld = 0;
}
#endif /* configUSE_MUTEXES */

vListInitialiseItem( &( pxNewTCB->xStateListItem ) );                          (8)
vListInitialiseItem( &( pxNewTCB->xEventListItem ) );                          (9)

listSET_LIST_ITEM_OWNER( &( pxNewTCB->xStateListItem ), pxNewTCB );            (10)
listSET_LIST_ITEM_VALUE( &( pxNewTCB->xEventListItem ), \                      (11)
                         ( TickType_t ) configMAX_PRIORITIES - ( TickType_t ) uxPriority );
listSET_LIST_ITEM_OWNER( &( pxNewTCB->xEventListItem ), pxNewTCB );            (12)

#if ( portCRITICAL_NESTING_IN_TCB == 1 )              //使能临界区嵌套
{
    pxNewTCB->uxCriticalNesting = ( UBaseType_t ) 0U;
}
#endif /* portCRITICAL_NESTING_IN_TCB */

#if ( configUSE_APPLICATION_TASK_TAG == 1 )           //使能任务标签功能
{
    pxNewTCB->pxTaskTag = NULL;
}
#endif /* configUSE_APPLICATION_TASK_TAG */

#if ( configGENERATE_RUN_TIME_STATS == 1 )            //使能时间统计功能
{
    pxNewTCB->ulRunTimeCounter = 0UL;
```

```
    }
    #endif /* configGENERATE_RUN_TIME_STATS */

    #if( configNUM_THREAD_LOCAL_STORAGE_POINTERS != 0 )
    {
        for( x = 0; x < ( UBaseType_t ) configNUM_THREAD_LOCAL_STORAGE_POINTERS;\
             x++ )
        {
            pxNewTCB->pvThreadLocalStoragePointers[ x ] = NULL;                (12)
        }
    }
    #endif

    #if ( configUSE_TASK_NOTIFICATIONS == 1 )      //使能任务通知功能
    {
        pxNewTCB->ulNotifiedValue = 0;
        pxNewTCB->ucNotifyState = taskNOT_WAITING_NOTIFICATION;
    }
    #endif

    #if ( configUSE_NEWLIB_REENTRANT == 1 )        //使能 NEWLIB
    {
        _REENT_INIT_PTR( ( &( pxNewTCB->xNewLib_reent ) ) );
    }
    #endif

    #if( INCLUDE_xTaskAbortDelay == 1 )                //使能函数 xTaskAbortDelay()
    {
        pxNewTCB->ucDelayAborted = pdFALSE;
    }
    #endif

    pxNewTCB->pxTopOfStack = pxPortInitialiseStack( pxTopOfStack, pxTaskCode,\  (13)
                                                    pvParameters );
    if( ( void * ) pxCreatedTask != NULL )
    {
        *pxCreatedTask = ( TaskHandle_t ) pxNewTCB;                            (14)
    }
    else
    {
        mtCOVERAGE_TEST_MARKER();
    }
}
```

(1)、如果使能了堆栈溢出检测功能或者追踪功能的话就使用一个定值 tskSTACK_FILL_BYTE 来填充任务堆栈，这个值为 0xa5U。

(2)、计算堆栈栈顶 pxTopOfStack，后面初始化堆栈的时候需要用到。

(3)、保存任务的任务名。

(4)、任务名数组添加字符串结束符'\0'。

(5)、判断任务优先级是否合法，如果设置的任务优先级大于 configMAX_PRIORITIES，则将优先级修改为 configMAX_PRIORITIES-1。

(6)、初始化任务控制块的优先级字段 uxPriority。

(7)、使能了互斥信号量功能，需要初始化相应的字段。

(8)和(9)、初始化列表项 xStateListItem 和 xEventListItem，任务控制块结构体中有两个列表项，这里对这两个列表项做初始化。

(10)和(12)、设置列表项 xStateListItem 和 xEventListItem 属于当前任务的任务控制块，也就是设置这两个列表项的字段 pvOwner 为新创建的任务的任务控制块。

(11)、设置列表项 xEventListItem 的字段 xItemValue 为 configMAX_PRIORITIES- uxPriority，比如当前任务优先级 3，最大优先级为 32，那么 xItemValue 就为 32-3=29，这就意味着 xItemValue 值越大，优先级就越小。上一章学习列表和列表项的时候我们说过，列表的插入是按照 xItemValue 的值升序排列的。

(12)、初始化线程本地存储指针，如果使能了这个功能的话。

(13)、调用函数 pxPortInitialiseStack()初始化任务堆栈。

(14)、生成任务句柄，返回给参数 pxCreatedTask，从这里可以看出任务句柄其实就是任务控制块。

8.3.3 任务堆栈初始化函数分析

在任务初始化函数中会对任务堆栈初始化，这个过程通过调用函数 pxPortInitialiseStack()来完成，函数 pxPortInitialiseStack()就是堆栈初始化函数，函数源码如下：

```
StackType_t *pxPortInitialiseStack(  StackType_t *  pxTopOfStack,
                                     TaskFunction_t  pxCode,
                                     void *          pvParameters )
{
    pxTopOfStack--;
    *pxTopOfStack = portINITIAL_XPSR;                                   (1)

    pxTopOfStack--;
    *pxTopOfStack = ( ( StackType_t ) pxCode ) & portSTART_ADDRESS_MASK;  (2)

    pxTopOfStack--;
    *pxTopOfStack = ( StackType_t ) prvTaskExitError;                   (3)

    pxTopOfStack -= 5;                                                  (4)
    *pxTopOfStack = ( StackType_t ) pvParameters;                       (5)

    pxTopOfStack--;
    *pxTopOfStack = portINITIAL_EXEC_RETURN;                            (6)
```

```
    pxTopOfStack -= 8;                                          (7)
    return pxTopOfStack;
}
```

堆栈是用来在进行上下文切换的时候保存现场的，一般在新创建好一个堆栈以后会对其先进行初始化处理，即对 Cortex-M 内核的某些寄存器赋初值。这些初值就保存在任务堆栈中，保存的顺序按照：xPSR、R15(PC)、R14(LR)、R12、R3~R0、R11~R14。

(1)、寄存器 xPSR 值为 portINITIAL_XPSR，其值为 0x01000000。xPSR 是 Cortex-M4 的一个内核寄存器，叫做程序状态寄存器，0x01000000 表示这个寄存器的 bit24 为 1，表示处于 Thumb 状态，即使用的 Thumb 指令。

(2)、寄存器 PC 初始化为任务函数 pxCode。

(3)、寄存器 LR 初始化为函数 prvTaskExitError。

(4)、跳过 4 个寄存器，R12，R3，R2，R1，这四个寄存器不初始化。

(5)、寄存器 R0 初始化为 pvParameters，一般情况下，函数调用会将 R0~R3 作为输入参数，R0 也可用作返回结果，如果返回值为 64 位，则 R1 也会用于返回结果，这里的 pvParameters 是作为任务函数的参数，保存在寄存器 R0 中。

(6)、保存 EXC_RETURN 值，用于设置退出 SVC 或 PendSV 中断的时候处理器应该处于什么壮态。处理器进入异常或中断服务程序(ISR)时，链接寄存器 R14(LR)的数值会被更新为 EXC_RETURN 数值，之后该数值会在异常处理结束时触发异常返回。这里人为的设置为 0XFFFFFFFD，表示退出异常以后 CPU 进入线程模式并且使用进程栈！关于 EXC_RETURN 的更多信息请参考《权威指南》“第 8 章 深入了解异常处理”。

(7)、跳过 8 个寄存器，R11、R10、R8、R7、R6、R5、R4。

经过上面的初始化之后，此时的堆栈结果如图 8.3.3.1 所示：

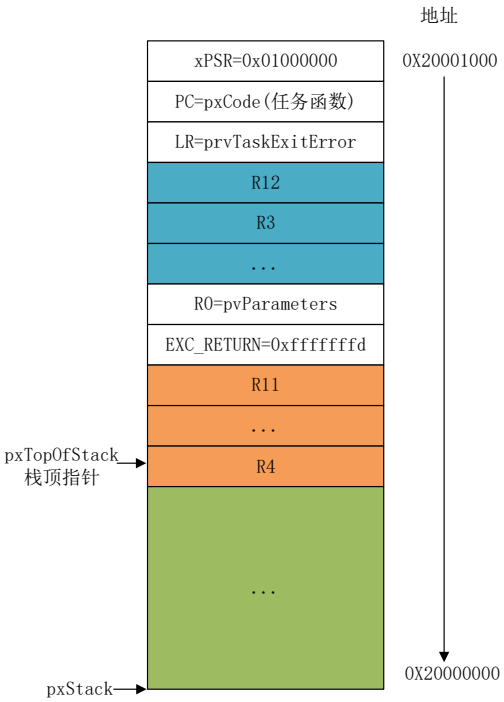


图 8.3.3.1 堆栈初始化

注意，图 8.3.3.1 中以 STM32 为例，堆栈为向下增长模式。

8.3.4 添加任务到就绪列表

任务创建完成以后就会被添加到就绪列表中，FreeRTOS 使用不同的列表表示任务的不同状态，在文件 tasks.c 中就定义了多个列表和指向列表的指针来完成不同的功能，这些列表如下：

```
PRIVILEGED_DATA static List_t       pxReadyTasksLists[ configMAX_PRIORITIES ];
PRIVILEGED_DATA static List_t       xDelayedTaskList1;
PRIVILEGED_DATA static List_t       xDelayedTaskList2;
PRIVILEGED_DATA static List_t *     volatile pxDelayedTaskList;
PRIVILEGED_DATA static List_t *     volatile pxOverflowDelayedTaskList;
PRIVILEGED_DATA static List_t       xPendingReadyList;
```

列表数组 pxReadyTasksLists[]就是任务就绪列表，数组大小为 configMAX_PRIORITIES，也就是说一个优先级一个列表，这样相同优先级的任务就使用一个列表。将一个新创建的任务添加到就绪列表中通过函数 prvAddNewTaskToReadyList()来完成，函数如下：

```
static void prvAddNewTaskToReadyList( TCB_t *pxNewTCB )
{
    taskENTER_CRITICAL();
    {
        uxCurrentNumberOfTasks++;                                    (1)
        if( pxCurrentTCB == NULL )//正在运行任务块为 NULL，说明没有任务运行！
        {
            pxCurrentTCB = pxNewTCB;//将新任务的任务控制块赋值给 pxCurrentTCB
            //新创建的任务是第一个任务！！！
            if( uxCurrentNumberOfTasks == ( UBaseType_t ) 1 )
            {
                prvInitialiseTaskLists();                            (2)
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
            }
        }
        else
        {
            if( xSchedulerRunning == pdFALSE )
            {
                //新任务的任务优先级比正在运行的任务优先级高。
                if( pxCurrentTCB->uxPriority <= pxNewTCB->uxPriority )
                {
                    pxCurrentTCB = pxNewTCB;                         (3)
                }
                else
                {
                    mtCOVERAGE_TEST_MARKER();
                }
```

```
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
            }
        }

        uxTaskNumber++;             //uxTaskNumber 加一，用作任务控制块编号。
        #if ( configUSE_TRACE_FACILITY == 1 )
        {
            pxNewTCB->uxTCBNumber = uxTaskNumber;
        }
        #endif /* configUSE_TRACE_FACILITY */
        prvAddTaskToReadyList( pxNewTCB );                              (4)
    }
    taskEXIT_CRITICAL();

    if( xSchedulerRunning != pdFALSE )
    {
        //新任务优先级比正在运行的任务优先级高
        if( pxCurrentTCB->uxPriority < pxNewTCB->uxPriority )
        {
            taskYIELD_IF_USING_PREEMPTION();                            (5)
        }
        else
        {
            mtCOVERAGE_TEST_MARKER();
        }
    }
    else
    {
        mtCOVERAGE_TEST_MARKER();
    }
}
```

(1)、变量 uxCurrentNumberOfTasks 为全局变量，用来统计任务数量。

(2)、变量 uxCurrentNumberOfTasks 为 1 说明正在创建的任务是第一个任务！那么就需要先初始化相应的列表，通过调用函数 prvInitialiseTaskLists()来初始化相应的列表。这个函数很简单，本质就是调用上一章讲的列表初始化函数 vListInitialise()来初始化几个列表，大家可以自行分析一下。

(3)、新创建的任务优先级比正在运行的任务优先级高，所以需要修改 pxCurrentTCB 为新建任务的任务控制块。

(4)、调用函数 prvAddTaskToReadyList()将任务添加到就绪列表中，这个其实是个宏，如下：

```
#define prvAddTaskToReadyList( pxTCB )                                  \
```

```
    traceMOVED_TASK_TO_READY_STATE( pxTCB );                                \
    taskRECORD_READY_PRIORITY( ( pxTCB )->uxPriority );                     \
    vListInsertEnd( &( pxReadyTasksLists[ ( pxTCB )->uxPriority ] ),        \
                    &( ( pxTCB )->xStateListItem ) );                       \
    tracePOST_MOVED_TASK_TO_READY_STATE( pxTCB )
```

其中宏 portRECORD_READY_PRIORITY()用来记录处于就绪态的任务，具体是通过操作全局变量 uxTopReadyPriority 来实现的。这个变量用来查找处于就绪态的优先级最高任务，具体操作过程后面讲解任务切换的时候会讲。接下来使用函数 vListInsertEnd()将任务添加到就绪列表末尾。

(5)、如果新任务的任务优先级最高，而且调度器已经开始正常运行了，那么就调用函数 taskYIELD_IF_USING_PREEMPTION()完成一次任务切换。

8.4 任务删除过程分析

前面我们已经学习了如何使用 FreeRTOS 的任务删除函数 vTaskDelete()，本节我们来详细的学习一下 vTaskDelete()这个函数的具体实现过程，函数源码如下：

```
void vTaskDelete( TaskHandle_t xTaskToDelete )
{
    TCB_t *pxTCB;

    taskENTER_CRITICAL();
    {
        //如果参数为 NULL 的话那么说明调用函数 vTaskDelete()的任务要删除自身。
        pxTCB = prvGetTCBFromHandle( xTaskToDelete );                       (1)

        //将任务从就绪列表中删除。
        if( uxListRemove( &( pxTCB->xStateListItem ) ) == ( UBaseType_t ) 0 )   (2)
        {
            taskRESET_READY_PRIORITY( pxTCB->uxPriority );
        }
        else
        {
            mtCOVERAGE_TEST_MARKER();
        }

        //任务是否在等待某个事件？
        if( listLIST_ITEM_CONTAINER( &( pxTCB->xEventListItem ) ) != NULL )  (3)
        {
            ( void ) uxListRemove( &( pxTCB->xEventListItem ) );
        }
        else
        {
            mtCOVERAGE_TEST_MARKER();
        }
```

```
            uxTaskNumber++;

            if( pxTCB == pxCurrentTCB )                                    (4)
            {
                vListInsertEnd( &xTasksWaitingTermination, &( pxTCB->\     (5)
                                                        xStateListItem ) );

                ++uxDeletedTasksWaitingCleanUp;                            (6)

                portPRE_TASK_DELETE_HOOK( pxTCB, &xYieldPending );         (7)
            }
            else
            {
                --uxCurrentNumberOfTasks;                                  (8)
                prvDeleteTCB( pxTCB );                                     (9)
                prvResetNextTaskUnblockTime();                             (10)
            }

            traceTASK_DELETE( pxTCB );
        }
        taskEXIT_CRITICAL();

        //如果删除的是正在运行的任务那么就需要强制进行一次任务切换。
        if( xSchedulerRunning != pdFALSE )
        {
            if( pxTCB == pxCurrentTCB )
            {
                configASSERT( uxSchedulerSuspended == 0 );
                portYIELD_WITHIN_API();                                    (11)
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
            }
        }
    }
```

(1)、调用函数 prvGetTCBFromHandle()获取要删除任务的任务控制块，参数为任务句柄。如果参数为当前正在执行的任务句柄那么返回值就为 NULL。

(2)、将任务从任务就绪列表中删除。

(3)、查看任务是否正在等待某个事件（如信号量、队列等），因为如果任务等待某个事件的话这个任务会被放到相应的列表中，这里需要将其从相应的列表中删除掉。

(4)、要删除的是当前正在运行的任务，

(5)、要删除任务，那么任务的任务控制块和任务堆栈所占用的内存肯定要被释放掉(如果使用动态方法创建的任务)，但是当前任务正在运行，显然任务控制块和任务堆栈的内存不能被立即释放掉！必须等到当前任务运行完成才能释放相应的内存，所以需要打一个“标记”，标记出有任务需要处理。这里将当前任务添加到列表 xTasksWaitingTermination 中，如果有任务要删除自身的话都会被添加到列表 xTasksWaitingTermination 中。那么问题来了？内存释放在哪里完成呢？空闲任务！空闲任务会依次将需要释放的内存都释放掉。

(6)、uxDeletedTasksWaitingCleanUp 是一个全局变量，用来记录有多少个任务需要释放内存。

(7)、调用任务删除钩子函数，钩子函数的具体内容需要用户自行实现。

(8)、删除的是别的任务，变量 uxCurrentNumberOfTasks 减一，也就是当前任务数减一。

(9)、因为是删除别的任务，所以可以直接调用函数 prvDeleteTCB()删除任务控制块。

(10)、重新计算一下还要多长时间执行下一个任务，也就是下一个任务的解锁时间，防止有任务的解锁时间参考了刚刚被删除的那个任务。

(11)、如果删除的是正在运行的任务那么删除完以后肯定需要强制进行一次任务切换。

8.5 任务挂起过程分析

挂起任务使用函数 vTaskSuspend()，函数源码如下：

```
void vTaskSuspend( TaskHandle_t xTaskToSuspend )
{
    TCB_t *pxTCB;

    taskENTER_CRITICAL();
    {
        //如果参数为 NULL 的话说明挂起自身
        pxTCB = prvGetTCBFromHandle( xTaskToSuspend );                    (1)
        traceTASK_SUSPEND( pxTCB );

        //将任务从就绪或者延时列表中删除，并且将任务放到挂起列表中
        if( uxListRemove( &( pxTCB->xStateListItem ) ) == ( UBaseType_t ) 0 )    (2)
        {
            taskRESET_READY_PRIORITY( pxTCB->uxPriority );
        }
        else
        {
            mtCOVERAGE_TEST_MARKER();
        }

        //任务是否还在等待其他事件
        if( listLIST_ITEM_CONTAINER( &( pxTCB->xEventListItem ) ) != NULL )  (3)
        {
            ( void ) uxListRemove( &( pxTCB->xEventListItem ) );
        }
        else
        {
```

```
            mtCOVERAGE_TEST_MARKER();
        }

        vListInsertEnd( &xSuspendedTaskList, &( pxTCB->xStateListItem ) );          (4)
    }
    taskEXIT_CRITICAL();

    if( xSchedulerRunning != pdFALSE )
    {
        taskENTER_CRITICAL();
        {
            prvResetNextTaskUnblockTime();                                          (5)
        }
        taskEXIT_CRITICAL();
    }
    else
    {
        mtCOVERAGE_TEST_MARKER();
    }

    if( pxTCB == pxCurrentTCB )
    {
        if( xSchedulerRunning != pdFALSE )
        {
            configASSERT( uxSchedulerSuspended == 0 );
            portYIELD_WITHIN_API();                                                 (6)
        }
        else
        {
            if( listCURRENT_LIST_LENGTH( &xSuspendedTaskList ) ==\                  (7)
                                                uxCurrentNumberOfTasks )
            {
                    pxCurrentTCB = NULL;                                            (8)
            }
            else
            {
                vTaskSwitchContext();                                               (9)
            }
        }
    }
    else
    {
        mtCOVERAGE_TEST_MARKER();
```

```
    }
}
```

(1)、通过函数 prvGetTCBFromHandle()获取要删除任务的任务控制块。

(2)、将任务从任务就绪列表延时列表中删除。

(3)、查看任务是否正在等待某个事件（如信号量、队列等），如果任务还在等待某个事件的话就将其从相应的事件列表中删除。

(4)、将任务添加到挂起任务列表尾，挂起任务列表为 xSuspendedTaskList，所有被挂起的任务都会被放到这个列表中。

(5)、重新计算一下还要多长时间执行下一个任务，也就是下一个任务的解锁时间。防止有任务的解锁时间参考了刚刚被挂起的那个任务。

(6)、如果刚刚挂起的任务是正在运行的任务，并且任务调度器运行正常，那么这里就需要调用函数 portYIELD_WITHIN_API()强制进行一次任务切换。

(7)、pxCurrentTCB 指向正在运行的任务，但是正在运行的任务要挂起了，所以必须给 pxCurrentTCB 重新找一个“对象”。也就是查找下一个将要运行的任务，本来这个工作是由任务切换函数来完成的，但是程序运行到这一行说明任务调度器被挂起了，任务切换函数也无能为力了，必须手动查找下一个要运行的任务了。调用函数 listCURRENT_LIST_LENGTH()判断一下系统中所有的任务是不是都被挂起了，也就是查看列表 xSuspendedTaskList 的长度是不是等于 uxCurrentNumberOfTasks。如果等于的话就说明系统中所有的任务都被挂起了(实际上不存在这种情况，因为最少都有一个空闲任务是可以运行的，空闲任务执行期间不会调用任何可以阻塞或者挂起空闲任务的 API 函数，为的就是保证系统中永远都有一个可运行的任务)。

(8)、如果所有任务都被挂起的话 pxCurrentTCB 就只能等于 NULL 了，这样当有新任务被创建的时候 pxCurrentTCB 就可以指向这个新任务。

(9)、有其他的没有被挂起的任务，调用 vTaskSwitchContext()获取下一个要运行的任务，函数 vTaskSwitchContext()会在下一章详细讲解。

8.6 任务恢复过程分析

任务恢复函数有两个 vTaskResume()和 xTaskResumeFromISR()，一个是用在任务中的，一个是用在中断中的，但是基本的处理过程都是一样的，我们就以函数 vTaskResume()为例来讲解一下任务恢复详细过程。

```
void vTaskResume( TaskHandle_t xTaskToResume )
{
    TCB_t * const pxTCB = ( TCB_t * ) xTaskToResume;                    (1)

    configASSERT( xTaskToResume );

    //函数参数不可能为 NULL。
    if( ( pxTCB != NULL ) && ( pxTCB != pxCurrentTCB ) )                (2)
    {
        taskENTER_CRITICAL();                                           (3)
        {
            if( prvTaskIsTaskSuspended( pxTCB ) != pdFALSE )            (4)
            {
                traceTASK_RESUME( pxTCB );
```

```
                ( void ) uxListRemove( &( pxTCB->xStateListItem ) );          (5)
                prvAddTaskToReadyList( pxTCB );                               (6)

                if( pxTCB->uxPriority >= pxCurrentTCB->uxPriority )           (7)
                {
                    taskYIELD_IF_USING_PREEMPTION();                          (8)
                }
                else
                {
                    mtCOVERAGE_TEST_MARKER();
                }
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
            }
        }
        taskEXIT_CRITICAL();                                                  (9)
    }
    else
    {
        mtCOVERAGE_TEST_MARKER();
    }
}
```

(1)、根据参数获取要恢复的任务的任务控制块，因为不存在恢复正在运行的任务这种情况所以参数也不可能为 NULL(你强行给个为 NULL 的参数那也没办法)，这里也就不需要使用函数 prvGetTCBFromHandle()来获取要恢复的任务控制块，prvGetTCBFromHandle()会处理参数为 NULL 这种情况。

(2)、任务控制块不能为 NULL 和 pxCurrentTCB，因为不存在说恢复当前正在运行的任务。

(3)、调用函数 taskENTER_CRITICAL()进入临界段

(4)、调用函数 prvTaskIsTaskSuspended()判断要恢复的任务之前是否已经被挂起了，恢复的肯定是被挂起的任务，没有挂起就不用恢复。

(5)、首先将要恢复的任务从原来的列表中删除，任务被挂起以后都会放到任务挂起列表 xSuspendedTaskList 中。

(6)、将要恢复的任务添加到就绪任务列表中。

(7)、要恢复的任务优先级高于当前正在运行的任务优先级。

(8)、因为要恢复的任务其优先级最高，所以需要调用函数 taskYIELD_IF_USING_PREEMPTION()来完成一次任务切换。

(9)、调用函数 taskEXIT_CRITICAL()退出临界区。

第九章 FreeRTOS 任务切换

RTOS 系统的核心是任务管理，而任务管理的核心是任务切换，任务切换决定了任务的执行顺序，任务切换效率的高低也决定了一款系统的性能，尤其是对于实时操作系统。而对于想深入了解 FreeRTOS 系统运行过程的同学其任务切换是必须掌握的知识点。本章我们就来学习一下 FreeRTOS 的任务切换过程，本章分为如下几部分：

9.1 PendSV 异常
9.2 FreeRTOS 任务切换场合
9.3 PendSV 中断服务函数
9.4 查找下一个要运行的任务

9.1 PendSV 异常

本小节参考自《权威指南》的“第 10 章 OS 支持特性”的第 10.4 小节。

PendSV(可挂起的系统调用)异常对 OS 操作非常重要，其优先级可以通过编程设置。可以通过将中断控制和壮态寄存器 ICSR 的 bit28，也就是 PendSV 的挂起位置 1 来触发 PendSV 中断。与 SVC 异常不同，它是不精确的，因此它的挂起壮态可在更高优先级异常处理内设置，且会在高优先级处理完成后执行。

利用该特性，若将 PendSV 设置为最低的异常优先级，可以让 PendSV 异常处理在所有其他中断处理完成后执行，这对于上下文切换非常有用，也是各种 OS 设计中的关键。

在具有嵌入式 OS 的典型系统中，处理时间被划分为了多个时间片。若系统中只有两个任务，这两个任务会交替执行，如图 9.1.1 所示：

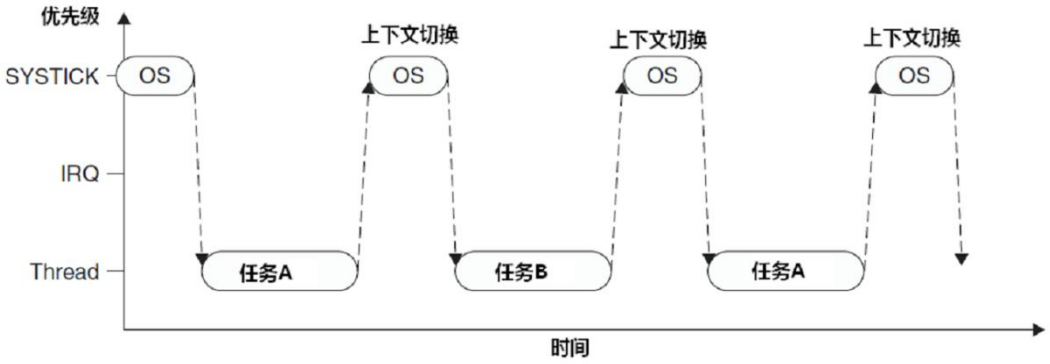


图 9.1.1 上下文切换简单实例

上下文切换被触发的场合可以是：

- 执行一个系统调用
- 系统滴答定时器(SysTick)中断。

在 OS 中，任务调度器决定是否应该执行上下文切换，如图 9.1.1 中任务切换都是由 SysTick 中断中执行，每次它都会决定切换到一个不同的任务中。

若中断请求(IRQ)在 SysTick 异常前产生，则 SysTick 异常可能会抢占 IRQ 的处理，在这种情况下，OS 不应该执行上下文切换，否则中断请求 IRQ 处理就会被延迟，而且在真实系统中延迟时间还往往不可预知——任何有一丁点实时要求的系统都决不能容忍这种事。对于 Cortex-M3 和 Cortex-M4 处理器，当存在活跃的异常服务时，设计默认不允许返回到线程模式，若存在活跃中断服务，且 OS 试图返回到线程模式，则将触发用法 fault，如图 9.1.2 所示。

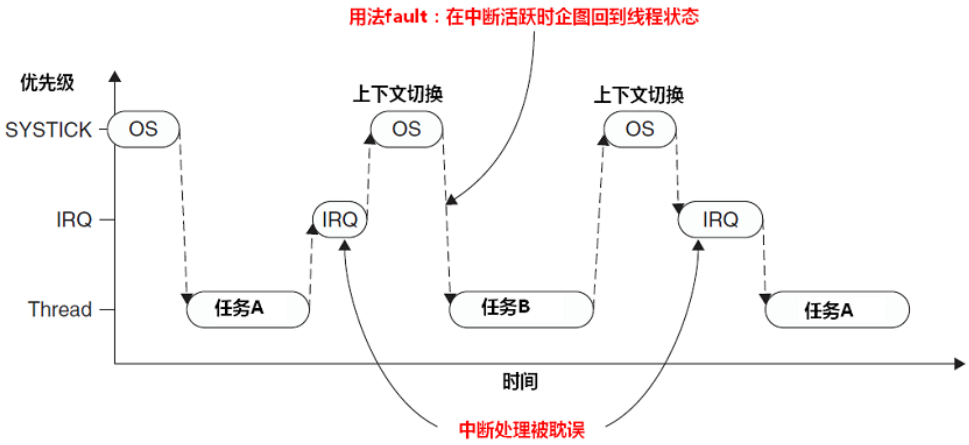


图 9.1.2 ISR 执行期间的上下文切换会延迟中断服务

在一些 OS 设计中，要解决这个问题，可以在运行中断服务时不执行上下文切换，此时可以检查栈帧中的压栈 xPSR 或 NVIC 中的中断活跃壮态寄存器。不过，系统的性能可能会受到影响，特别时当中断源在 SysTick 中断前后持续产生请求时，这样上下文切换可能就没有执行的机会了。

为了解决这个问题，PendSV 异常将上下文切换请求延迟到所有其他 IRQ 处理都已经完成后，此时需要将 PendSV 设置为最低优先级。若 OS 需要执行上下文切换，他会设置 PendSV 的挂起壮态，并在 PendSV 异常内执行上下文切换。如图 9.1.3 所示：

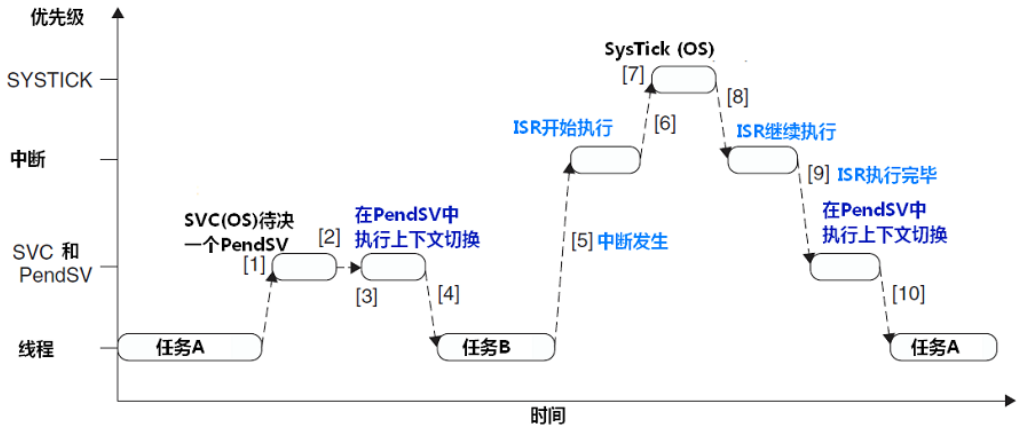


图 9.1.3 PendSV 上下文切换

图 9.1.3 中事件的流水账记录如下：

(1) 任务 A 呼叫 SVC 来请求任务切换（例如，等待某些工作完成）

(2) OS 接收到请求，做好上下文切换的准备，并且 pend 一个 PendSV 异常。

(3) 当 CPU 退出 SVC 后，它立即进入 PendSV，从而执行上下文切换。

(4) 当 PendSV 执行完毕后，将返回到任务 B，同时进入线程模式。

(5) 发生了一个中断，并且中断服务程序开始执行

(6) 在 ISR 执行过程中，发生 SysTick 异常，并且抢占了该 ISR。

(7) OS 执行必要的操作，然后 pend 起 PendSV 异常以作好上下文切换的准备。

(8) 当 SysTick 退出后，回到先前被抢占的 ISR 中， ISR 继续执行

(9) ISR 执行完毕并退出后， PendSV 服务例程开始执行，并且在里面执行上下文切换。

(10) 当 PendSV 执行完毕后，回到任务 A，同时系统再次进入线程模式。

讲解 PendSV 异常的原因就是让大家知道，FreeRTOS 系统的任务切换最终都是在 PendSV 中断服务函数中完成的，uC/OS 也是在 PendSV 中断中完成任务切换的。

9.2 FreeRTOS 任务切换场合

在 9.1 小节中讲解 PendSV 中断的时候提到了上下文(任务)切换被触发的场合：

- 可以执行一个系统调用
- 系统滴答定时器(SysTick)中断。

9.2.1 执行系统调用

执行系统调用就是执行 FreeRTOS 系统提供的相关 API 函数，比如任务切换函数 taskYIELD()，FreeRTOS 有些 API 函数也会调用函数 taskYIELD()，这些 API 函数都会导致任务切换，这些 API 函数和任务切换函数 taskYIELD()都统称为系统调用。函数 taskYIELD()其实就是个宏，在文件 task.h 中有如下定义：

```
#define taskYIELD()                    portYIELD()
```

函数 portYIELD()也是个宏，在文件 portmacro.h 中有如下定义：

```
#define portYIELD()                                                    \
{                                                                      \
    portNVIC_INT_CTRL_REG = portNVIC_PENDSVSET_BIT;                    \        (1)
                                                                       \
    __dsb( portSY_FULL_READ_WRITE );                                   \
    __isb( portSY_FULL_READ_WRITE );                                   \
}
```

(1)、通过向中断控制和壮态寄存器 ICSR 的 bit28 写入 1 挂起 PendSV 来启动 PendSV 中断。这样就可以在 PendSV 中断服务函数中进行任务切换了。

中断级的任务切换函数为 portYIELD_FROM_ISR()，定义如下：

```
#define    portEND_SWITCHING_ISR( xSwitchRequired ) \
           if( xSwitchRequired != pdFALSE )    portYIELD()
#define portYIELD_FROM_ISR( x ) portEND_SWITCHING_ISR( x )
```

可以看出 portYIELD_FROM_ISR()最终也是通过调用函数 portYIELD()来完成任务切换的。

9.2.2 系统滴答定时器(SysTick)中断

FreeRTOS 中滴答定时器(SysTick)中断服务函数中也会进行任务切换，滴答定时器中断服务函数如下：

```
void SysTick_Handler(void)
{
    if(xTaskGetSchedulerState()!=taskSCHEDULER_NOT_STARTED)//系统已经运行
    {
        xPortSysTickHandler();
    }
    HAL_IncTick();
}
```

在滴答定时器中断服务函数中调用了 FreeRTOS 的 API 函数 xPortSysTickHandler()，此函数源码如下：

```
void xPortSysTickHandler( void )
{
    vPortRaiseBASEPRI();                                        (1)
    {
        if( xTaskIncrementTick() != pdFALSE )      //增加时钟计数器 xTickCount 的值
        {
            portNVIC_INT_CTRL_REG = portNVIC_PENDSVSET_BIT;     (2)
        }
    }
    vPortClearBASEPRIFromISR();                                 (3)
}
```

(1)、关闭中断

(2)、通过向中断控制和壮态寄存器 ICSR 的 bit28 写入 1 挂起 PendSV 来启动 PendSV 中断。这样就可以在 PendSV 中断服务函数中进行任务切换了。

(3)、打开中断。

9.3 PendSV 中断服务函数

前面说了 FreeRTOS 任务切换的具体过程是在 PendSV 中断服务函数中完成的，本节我们就来学习一个 PendSV 的中断服务函数，看看任务切换过程究竟是怎么进行的。PendSV 中断服务函数本应该为 PendSV_Handler()，但是 FreeRTOS 使用#define 重定义了，如下：

```
#define xPortPendSVHandler        PendSV_Handler
```

函数 xPortPendSVHandler()源码如下：

```
__asm void xPortPendSVHandler( void )
{
    extern uxCriticalNesting;
    extern pxCurrentTCB;
    extern vTaskSwitchContext;

    PRESERVE8

    mrs r0, psp                                          (1)
    isb

    ldr    r3, =pxCurrentTCB                             (2)
    ldr    r2, [r3]                                      (3)

    tst r14, #0x10                                       (4)
    it eq                                                (5)
    vstmdbeq r0!, {s16-s31}                              (6)
    stmdb r0!, {r4-r11, r14}                             (7)
    str r0, [r2]                                         (8)

    stmdb sp!, {r3}                                      (9)
    mov r0, #configMAX_SYSCALL_INTERRUPT_PRIORITY        (10)
    msr basepri, r0                                      (11)
    dsb
    isb
    bl vTaskSwitchContext                                (12)
    mov r0, #0                                           (13)
    msr basepri, r0                                      (14)
    ldmia sp!, {r3}                                      (15)

    ldr r1, [r3]                                         (16)
    ldr r0, [r1]                                         (17)
```

```
    ldmia r0!, {r4-r11, r14}                    (18)

    tst r14, #0x10                              (19)
    it eq                                       (20)
    vldmiaeq r0!, {s16-s31}                     (21)

    msr psp, r0                                 (22)
    isb
    bx r14                                      (23)
}
```

(1)、读取进程栈指针，保存在寄存器 R0 里面。

(2)和(3)，获取当前任务的任务控制块，并将任务控制块的地址保存在寄存器 R2 里面。

(4)和(5)、判断任务是否使用了 FPU，如果任务使用了 FPU 的话在进行任务切换的时候就需要将 FPU 寄存器 s16~s31 手动保存到任务堆栈中，其中 s0~s15 和 FPSCR 是自动保存的。

(6)、保存 s16~s31 这 16 个 FPU 寄存器。

(7)、保存 r4~r11 和 R14 这几个寄存器的值。

(8)、将寄存器 R0 的值写入到寄存器 R2 所保存的地址中去，也就是将新的栈顶保存在任务控制块的第一个字段中。此时的寄存器 R0 保存着最新的堆栈栈顶指针值，所以要将这个最新的栈顶指针写入到当前任务的任务控制块第一个字段，而经过(2)和(3)已经获取到了任务控制块，并将任务控制块的首地址写如到了寄存器 R2 中。

(9)、将寄存器 R3 的值临时压栈，寄存器 R3 中保存了当前任务的任务控制块，而接下来要调用函数 vTaskSwitchContext()，为了防止 R3 的值被改写，所以这里临时将 R3 的值先压栈。

(10)和(11)、关闭中断，进入临界区

(12)、调用函数 vTaskSwitchContext()，此函数用来获取下一个要运行的任务，并将 pxCurrentTCB 更新为这个要运行的任务。

(13)和(14)、打开中断，退出临界区。

(15)、刚刚保存的寄存器 R3 的值出栈，恢复寄存器 R3 的值。注意，经过(12)步，此时 pxCurrentTCB 的值已经改变了，所以读取 R3 所保存的地址处的数据就会发现其值改变了，成为了下一个要运行的任务的任务控制块。

(16)和(17)、获取新的要运行的任务的任务堆栈栈顶,并将栈顶保存在寄存器 R0 中。

(18)、R4~R11,R14 出栈，也就是即将运行的任务的现场。

(19)、(20)和(21)、判断即将运行的任务是否有使用到 FPU，如果有的话还需要手工恢复 FPU 的 s16~s31 寄存器。

(22)、更新进程栈指针 PSP 的值。

(23)、执行此行代码以后硬件自动恢复寄存器 R0~R3、R12、LR、PC 和 xPSR 的值，确定异常返回以后应该进入处理器模式还是进程模式，使用主栈指针(MSP)还是进程栈指针(PSP)。很明显这里会进入进程模式，并且使用进程栈指针(PSP)，寄存器 PC 值会被恢复为即将运行的任务的任务函数，新的任务开始运行！至此，任务切换成功。

9.4 查找下一个要运行的任务

在 PendSV 中断服务程序中有调用函数 vTaskSwitchContext()来获取下一个要运行的任务，也就是查找已经就绪了的优先级最高的任务，缩减后(去掉条件编译)函数源码如下：

```
void vTaskSwitchContext( void )
```

```
{
    if( uxSchedulerSuspended != ( UBaseType_t ) pdFALSE )                (1)
    {
        xYieldPending = pdTRUE;
    }
    else
    {
        xYieldPending = pdFALSE;
        traceTASK_SWITCHED_OUT();
        taskCHECK_FOR_STACK_OVERFLOW();

        taskSELECT_HIGHEST_PRIORITY_TASK();                              (2)

        traceTASK_SWITCHED_IN();
    }
}
```

(1)、如果调度器挂起那就不能进行任务切换。

(2)、调用函数 taskSELECT_HIGHEST_PRIORITY_TASK()获取下一个要运行的任务。taskSELECT_HIGHEST_PRIORITY_TASK()本质上是一个宏，在 tasks.c 中有定义。

FreeRTOS 中查找下一个要运行的任务有两种方法：一个是通用的方法，另外一个就是使用硬件的方法，这个在我们讲解 FreeRTOSCofnig.h 文件的时候就提到过了，至于选择哪种方法通过宏 configUSE_PORT_OPTIMISED_TASK_SELECTION 来决定的。当这个宏为 1 的时候就使用硬件的方法，否则的话就是使用通用的方法，我们来看一下这两个方法的区别。

1、通用方法

顾名思义，就是所有的处理器都可以用的方法，方法如下：

```
#define taskSELECT_HIGHEST_PRIORITY_TASK()                                \
{                                                                         \
    UBaseType_t uxTopPriority = uxTopReadyPriority;                       \
    while( listLIST_IS_EMPTY( &( pxReadyTasksLists[ uxTopPriority ] ) ) ) \     (1)
    {                                                                     \
        configASSERT( uxTopPriority );                                    \
        --uxTopPriority;                                                  \
    }                                                                     \
    listGET_OWNER_OF_NEXT_ENTRY( pxCurrentTCB,                            \     (2)
                                 &( pxReadyTasksLists[ uxTopPriority ] ) );\
    uxTopReadyPriority = uxTopPriority;                                   \
}
```

(1)、在前面的 8.2.4 小节中说了 pxReadyTasksLists[]为就绪任务列表数组，一个优先级一个列表，同优先级的就绪任务都挂到相对应的列表中。uxTopReadyPriority 代表处于就绪态的最高优先级值，每次创建任务的时候都会判断新任务的优先级是否大于 uxTopReadyPriority，如果大于的话就将这个新任务的优先级赋值给变量 uxTopReadyPriority。函数 prvAddTaskToReadyList()也会修改这个值，也就是说将某个任务添加到就绪列表中的时候都会用 uxTopReadyPriority 来记录就绪列表中的最高优先级。这里就从这个最高优先级开始判断，看看哪个列表不为空就说

明哪个优先级有就绪的任务。函数 listLIST_IS_EMPTY()用于判断某个列表是否为空，uxTopPriority 用来记录这个有就绪任务的优先级。

(2)、已经找到了有就绪任务的优先级了，接下来就是从对应的列表中找出下一个要运行的任务，查找方法就是使用函数 listGET_OWNER_OF_NEXT_ENTRY()来获取列表中的下一个列表项，然后将获取到的列表项所对应的任务控制块赋值给 pxCurrentTCB，这样我们就确定了下一个要运行的任务了。

可以看出通用方法是完全通过 C 语言来实现的，肯定适用于不同的芯片和平台，而且对于任务数量没有限制，但是效率肯定相对于使用硬件方法的要低很多。

2、硬件方法

硬件方法就是使用处理器自带的硬件指令来实现的，比如 Cortex-M 处理器就带有的计算前导 0 个数指令：CLZ，函数如下：

```
#define taskSELECT_HIGHEST_PRIORITY_TASK()                          \
{                                                                   \
    UBaseType_t uxTopPriority;                                      \
    portGET_HIGHEST_PRIORITY( uxTopPriority, uxTopReadyPriority );  \    (1)
    configASSERT( listCURRENT_LIST_LENGTH( &                        \
                        ( pxReadyTasksLists[ uxTopPriority ] ) )> 0 ); \
    listGET_OWNER_OF_NEXT_ENTRY( pxCurrentTCB,                      \    (2)
                        &( pxReadyTasksLists[ uxTopPriority ] ) );  \
}
```

(1)、通过函数 portGET_HIGHEST_PRIORITY()获取处于就绪态的最高优先级，portGET_HIGHEST_PRIORITY 本质上是个宏，定义如下：

```
#define portGET_HIGHEST_PRIORITY( uxTopPriority, uxReadyPriorities ) uxTopPriority = ( 31UL\
                                        - ( uint32_t ) __clz( ( uxReadyPriorities ) ) )
```

使用硬件方法的时候 uxTopReadyPriority 就不代表处于就绪态的最高优先级了，而是使用每个 bit 代表一个优先级，bit0 代表优先级 0，bit31 就代表优先级 31，当某个优先级有就绪任务的话就将其对应的 bit 置 1。从这里就可以看出，如果使用硬件方法的话最多只能有 32 个优先级。__clz(uxReadyPriorities)就是计算 uxReadyPriorities 的前导零个数，前导零个数就是指从最高位开始(bit31)到第一个为 1 的 bit，其间 0 的个数，如下例子：

二进制数 1000 0000 0000 0000 的前导零个数就为 0。

二进制数 0000 1001 1111 0001 的前导零个数就是 4。

得到 uxTopReadyPriority 的前导零个数以后在用 31 减去这个前导零个数得到的就是处于就绪态的最高优先级了，比如优先级 30 为此时的处于就绪态的最高优先级，30 的前导零个数为 1，那么 31-1=30，得到处于就绪态的最高优先级为 30。

(2)、已经找到了处于就绪态的最高优先级了，接下来就是从对应的列表中找出下一个要运行的任务，查找方法就是使用函数 listGET_OWNER_OF_NEXT_ENTRY()来获取列表中的下一个列表项，然后将获取到的列表项所对应的任务控制块赋值给 pxCurrentTCB，这样我们就确定了下一个要运行的任务了。

可以看出硬件方法借助一个指令就可以快速的获取处于就绪态的最高优先级，但是会限制任务的优先级数，比如 STM32 只能有 32 个优先级，不过 32 个优先级已经完全够用了。要知道 FreeRTOS 是支持时间片的，每个优先级可以支持无限多个任务。

9.6 FreeRTOS 时间片调度

前面多次提到 FreeRTOS 支持多个任务同时拥有一个优先级，这些任务的调度是一个值得考虑的问题，不过这不是我们要考虑的。在 FreeRTOS 中允许一个任务运行一个时间片(一个时钟节拍的长度)后让出 CPU 的使用权，让拥有同优先级的下一个任务运行，至于下一个要运行哪个任务？在 9.4 小节里面已经分析过了，FreeRTOS 中的这种调度方法就是时间片调度。图 9.6.1 展示了运行在同一优先级下的执行时间图，在优先级 N 下有 3 个就绪的任务。

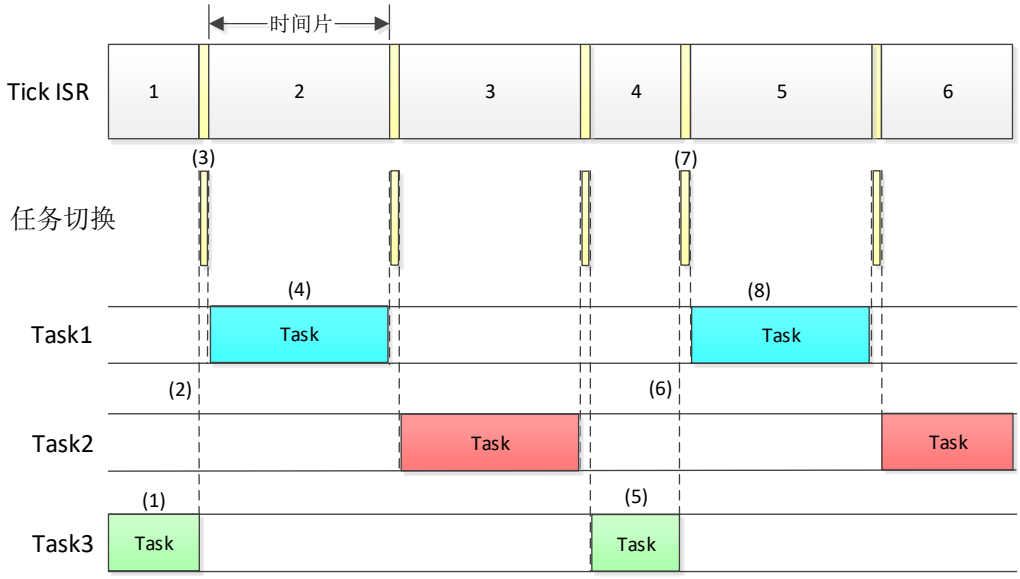


图 9.6.1 时间片调度

1、任务 3 正在运行。

2、这时一个时钟节拍中断(滴答定时器中断)发生，任务 3 的时间片用完，但是任务 3 还没有执行完。

3、FreeRTOS 将任务切换到任务 1，任务 1 是优先级 N 下的下一个就绪任务。

4、任务 1 连续运行至时间片用完。

5、任务 3 再次获取到 CPU 使用权，接着运行。

6、任务 3 运行完成，调用任务切换函数 portYIELD()强行进行任务切换放弃剩余的时间片，从而使优先级 N 下的下一个就绪的任务运行。

7、FreeRTOS 切换到任务 1。

8、任务 1 执行完其时间片。

要使用时间片调度的话宏 configUSE_PREEMPTION 和宏 configUSE_TIME_SLICING 必须为 1。时间片的长度由宏 configTICK_RATE_HZ 来确定，一个时间片的长度就是滴答定时器的中断周期，比如本教程中 configTICK_RATE_HZ 为 1000，那么一个时间片的长度就是 1ms。时间片调度发生在滴答定时器的中断服务函数中，前面讲解滴答定时器中断服务函数的时候说了在中断服务函数 SysTick_Handler()中会调用 FreeRTOS 的 API 函数 xPortSysTickHandler()，而函数 xPortSysTickHandler()会引发任务调度，但是这个任务调度是有条件的，函数 xPortSysTickHandler()如下：

```
void xPortSysTickHandler( void )
{
    vPortRaiseBASEPRI();
    {
        if( xTaskIncrementTick() != pdFALSE )
```

```
            {
                portNVIC_INT_CTRL_REG = portNVIC_PENDSVSET_BIT;
            }
        }
        vPortClearBASEPRIFromISR();
}
```

上述代码中红色部分表明只有函数 xTaskIncrementTick()的返回值不为 pdFALSE 的时候就会进行任务调度！查看函数 xTaskIncrementTick()会发现有如下条件编译语句：

```
BaseType_t xTaskIncrementTick( void )
{
    TCB_t * pxTCB;
    TickType_t xItemValue;
    BaseType_t xSwitchRequired = pdFALSE;

    if( uxSchedulerSuspended == ( UBaseType_t ) pdFALSE )
    {
    /*************************************************************************/
    /*************************此处省去一大堆代码*****************************/
    /*************************************************************************/
        #if ( ( configUSE_PREEMPTION == 1 ) && ( configUSE_TIME_SLICING == 1 ) ) (1)
        {
            if( listCURRENT_LIST_LENGTH( &( \
              pxReadyTasksLists[ pxCurrentTCB->uxPriority ] ) ) > ( UBaseType_t ) 1 )      (2)
            {
                xSwitchRequired = pdTRUE;                                                  (3)
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
            }
        }
        #endif /* ( ( configUSE_PREEMPTION == 1 ) && ( configUSE_TIME_SLICING == 1 ) )
    }
    return xSwitchRequired;
}
```

(1)、当宏 configUSE_PREEMPTION 和宏 configUSE_PREEMPTION 都为 1 的时候下面的代码才会编译。所以要想使用时间片调度的话这这两个宏都必须为 1，缺一不可！

(2)、判断当前任务所对应的优先级下是否还有其他的任务。

(3)、如果当前任务所对应的任务优先级下还有其他的任务那么就返回 pdTRUE。

从上面的代码可以看出，如果当前任务所对应的优先级下有其他的任务存在，那么函数 xTaskIncrementTick() 就会返回 pdTURE，由于函数返回值为 pdTURE 因此函数 xPortSysTickHandler()就会进行一次任务切换。

9.6 时间片调度实验

9.6.1 实验程序设计

1、实验目的

学习使用 FreeRTOS 的时间片调度。

2、实验设计

本实验设计三个任务：start_task、task1_task 和 task2_task ，其中 task1_task 和 task2_task 的任务优先级相同，都为 2，这三个任务的任务功能如下：

start_task：用来创建其他 2 个任务。

task1_task ：控制 LED0 灯闪烁，并且通过串口打印 task1_task 的运行次数。

task2_task ：控制 LED1 灯闪烁，并且通过串口打印 task2_task 的运行次数。

3、实验工程

FreeRTOS 实验 9-1 FreeRTOS 时间片调度。

4、实验程序与分析

● **系统设置**

为了观察方便，将系统的时钟节拍频率设置为 20，也就是将宏 configTICK_RATE_HZ 设置为 20：

```
#define configTICK_RATE_HZ                    (20)
```

这样设置以后滴答定时器的中断周期就是 50ms 了，也就是说时间片值为 50ms，这个时间片还是很大的，不过大一点我们到时候观察的时候方便。

● **任务设置**

```
#define START_TASK_PRIO          1        //任务优先级
#define START_STK_SIZE           128      //任务堆栈大小
TaskHandle_t StartTask_Handler;           //任务句柄
void start_task(void *pvParameters);      //任务函数

#define TASK1_TASK_PRIO          2        //任务优先级          (1)
#define TASK1_STK_SIZE           128      //任务堆栈大小
TaskHandle_t Task1Task_Handler;           //任务句柄
void task1_task(void *pvParameters);      //任务函数

#define TASK2_TASK_PRIO          2        //任务优先级          (2)
#define TASK2_STK_SIZE           128      //任务堆栈大小
TaskHandle_t Task2Task_Handler;           //任务句柄
void task2_task(void *pvParameters);      //任务函数
```

(1)和(2)、任务 task1_task 和 task2_task 的任务优先级设置为相同的，这里都设置为 2。

● **main()函数**

```
int main(void)
{
```

```
    HAL_Init();                             //初始化 HAL 库
    Stm32_Clock_Init(360,25,2,8);           //设置时钟,180Mhz
    delay_init(180);                        //初始化延时函数
    uart_init(115200);                      //初始化串口
    LED_Init();                             //初始化 LED
    KEY_Init();                             //初始化按键
    SDRAM_Init();                           //初始化 SDRAM
    LCD_Init();                             //初始化 LCD

    POINT_COLOR = RED;
    LCD_ShowString(30,10,200,16,16,"Apollo STM32F4/F7");
    LCD_ShowString(30,30,200,16,16,"FreeRTOS Examp 9-1");
    LCD_ShowString(30,50,200,16,16,"FreeRTOS Round Robin");
    LCD_ShowString(30,70,200,16,16,"ATOM@ALIENTEK");
    LCD_ShowString(30,90,200,16,16,"2016/10/20");

    //创建开始任务
    xTaskCreate((TaskFunction_t     )start_task,            //任务函数
                (const char*        )"start_task",          //任务名称
                (uint16_t           )START_STK_SIZE,        //任务堆栈大小
                (void*              )NULL,                  //传递给任务函数的参数
                (UBaseType_t        )START_TASK_PRIO,       //任务优先级
                (TaskHandle_t*      )&StartTask_Handler);   //任务句柄
    vTaskStartScheduler();                                  //开启任务调度
}
```

在 main 函数中我们主要完成硬件的初始化，在硬件初始化完成以后创建了任务 start_task 并且开启了 FreeRTOS 的任务调度。

● **任务函数**

```
//开始任务任务函数
void start_task(void *pvParameters)
{
    taskENTER_CRITICAL();           //进入临界区
    //创建 TASK1 任务
    xTaskCreate((TaskFunction_t     )task1_task,
                (const char*        )"task1_task",
                (uint16_t           )TASK1_STK_SIZE,
                (void*              )NULL,
                (UBaseType_t        )TASK1_TASK_PRIO,
                (TaskHandle_t*      )&Task1Task_Handler);
    //创建 TASK2 任务
    xTaskCreate((TaskFunction_t     )task2_task,
                (const char*        )"task2_task",
                (uint16_t           )TASK2_STK_SIZE,
```

```
                (void*              )NULL,
                (UBaseType_t        )TASK2_TASK_PRIO,
                (TaskHandle_t*      )&Task2Task_Handler);
    vTaskDelete(StartTask_Handler);         //删除开始任务
    taskEXIT_CRITICAL();                    //退出临界区
}

//task1 任务函数
void task1_task(void *pvParameters)
{
    u8 task1_num=0;
    while(1)
    {
        task1_num++;      //任务 1 执行次数加 1  注意 task1_num1 加到 255 的时候会清零!!
        LED0=!LED0;
        taskENTER_CRITICAL();   //进入临界区
        printf("任务 1 已经执行：%d 次\r\n",task1_num);
        taskEXIT_CRITICAL();      //退出临界区
        //延时 10ms，模拟任务运行 10ms，此函数不会引起任务调度
        delay_xms(10);                                                    (1)
    }
}

//task2 任务函数
void task2_task(void *pvParameters)
{
    u8 task2_num=0;
    while(1)
    {
        task2_num++; //任务 2 执行次数加 1  注意 task2_num1 加到 255 的时候会清零!!
        LED1=!LED1;
        taskENTER_CRITICAL();   //进入临界区
        printf("任务 2 已经执行：%d 次\r\n",task2_num);
        taskEXIT_CRITICAL();      //退出临界区
        //延时 10ms，模拟任务运行 10ms，此函数不会引起任务调度
        delay_xms(10);                                                    (2)
    }
}
```

(1)、调用函数 delay_xms()延时 10ms。在一个时间片内如果任务不主动放弃 CPU 使用权的话那么就会一直运行这一个任务，直到时间片耗尽。在 task1_task 任务中我们通过串口打印字符串的方式提示 task1_task 在运行，但是这个过程对于 CPU 来说执行速度很快，不利于观察，所以这里通过调用函数 delay_xms()来默认任务占用 10ms 的 CPU。函数 delay_xm()不会引起任务调度，这样的话相当于 task1_task 的执行周期＞10ms，基本可以看作等于 10ms，因为其他的

函数执行速度还是很快的。一个时间片的长度是 50ms，任务执行所需的时间以 10ms 算，理论上在一个时间片内 task1_task 可以执行 5 次，但是事实上很少能执行 5 次，基本上是 4 次。

(2)、同(1)。

9.6.2 程序运行结果分析

编译并下载实验代码到开发板中，打开串口调试助手，开发板上电，串口调试助手显示如图 9.6.2.1 所示：

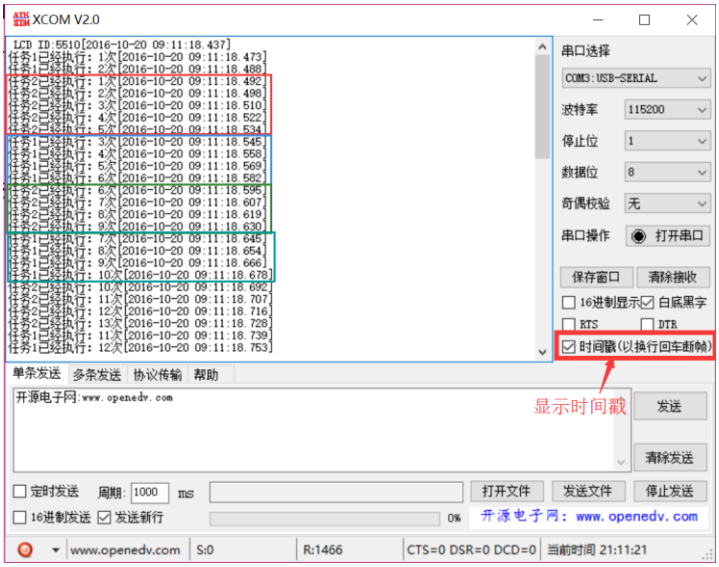


图 9.6.2.1 串口调试助手接收到的信息

从图 9.6.2.1 可以看出，不管是 task1_task 还是 task2_task 都是连续执行 4，5 次，和前面程序设计的一样，说明在一个时间片内一直在运行一个任务，当时间片用完后就切换到下一个任务运行。注意，接收到的信息后面显示的时间是串口调试助手统计的接收到数据的时间，并不是开发板真实的运行时间，这个时间戳值仅供参考。

第十章 FreeRTOS 系统内核控制函数

FreeRTOS 中有一些函数只供系统内核使用，用户应用程序一般不允许使用，这些 API 函数就是系统内核控制函数。本章我们就来学习一下这些内核控制函数，本章分为如下几部分：

10.1 内核控制函数预览

10.2 内核控制函数详解

10.1 内核控制函数预览

顾名思义，内核控制函数就是 FreeRTOS 内核所使用的函数，一般情况下应用层程序不使用这些函数，在 FreeRTOS 官网可以找到这些函数，如图 10.1.1 所示：

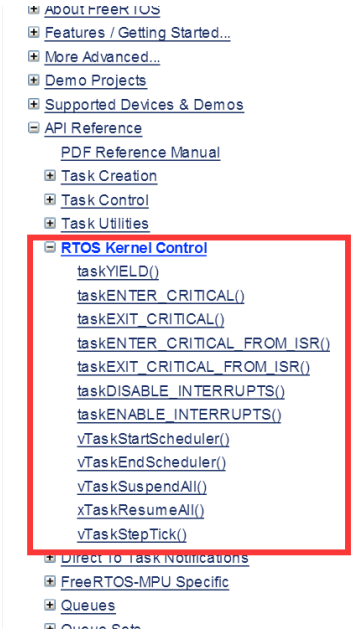


图 10.1.1 内核控制函数

这些函数的含义如表 10.1.1 所示：

函数	描述
taskYIELD()	任务切换。
taskENTER_CRITICAL()	进入临界区，用于任务中。
taskEXIT_CRITICAL()	退出临界区，用于任务中。
taskENTER_CRITICAL_FROM_ISR()	进入临界区，用于中断服务函数中。
taskEXIT_CRITICAL_FROM_ISR()	退出临界区，用于中断服务函数中。
taskDISABLE_INTERRUPTS()	关闭中断。
taskENABLE_INTERRUPTS()	打开中断。
vTaskStartScheduler()	开启任务调度器。
vTaskEndScheduler()	关闭任务调度器。
vTaskSuspendAll()	挂起任务调度器。
xTaskResumeAll()	恢复任务调度器。
vTaskStepTick()	设置系统节拍值。

表 10.1.1 内核控制函数

10.2 内核控制函数详解

1、函数 taskYIELD()

此函数用于进行任务切换，此函数本质上是一个宏，此函数的详细讲解请参考 9.2.1 小节。

2、函数 taskENTER_CRITICAL()

进入临界区，用于任务函数中，本质上是一个宏，此函数的详细讲解请参考 4.4.1 小节。

3、函数 taskEXIT_CRITICAL()

退出临界区，用于任务函数中，本质上是一个宏，此函数的详细讲解请参考 4.4.1 小节。

4、函数 taskENTER_CRITICAL_FROM_ISR()

进入临界区，用于中断服务函数中，此函数本质上是一个宏，此函数的详细讲解请参考 4.4.2 小节。

5、函数 taskEXIT_CRITICAL_FROM_ISR()

退出临界区，用于中断服务函数中，此函数本质上是一个宏，此函数的详细讲解请参考 4.4.2 小节。

6、函数 taskDISABLE_INTERRUPTS()

关闭可屏蔽的中断，此函数本质上是一个宏，此函数的详细讲解请参考 4.3 小节。

7、函数 taskENABLE_INTERRUPTS()

打开可屏蔽的中断，此函数本质上是一个宏，此函数的详细讲解请参考 4.3 小节。

8、函数 vTaskStartScheduler()

启动任务调度器，此函数的详细讲解请参考 8.3 小节。

9、函数 vTaskEndScheduler()

关闭任务调度器，FreeRTOS 中对于此函数的解释如图 10.2.1 所示：

```
/**
 * task. h
 * <pre>void vTaskEndScheduler( void );</pre>
 *
 * NOTE: At the time of writing only the x86 real mode port, which runs on a PC
 * in place of DOS, implements this function.
 *
```

图 10.2.1 函数解释

可以看出此函数仅用于 X86 架构的处理器，调用此函数以后所有系统时钟就会停止运行，所有创建的任务都会自动的删除掉(FreeRTOS 对此函数的解释是会自动删除所有的任务，但是在 FreeRTOS 的源码中没有找到相关的处理过程，有可能要根据实际情况编写相关代码，亦或是 X86 的硬件会自动处理？笔者不了解 X86 架构)，多任务性能关闭。可以调用函数 vTaskStartScheduler()来重新开启任务调度器。此函数在文件 tasks.c 中有如下定义：

```
void vTaskEndScheduler( void )
{
    portDISABLE_INTERRUPTS();              //关闭中断
    xSchedulerRunning = pdFALSE;           //标记任务调度器停止运行
    vPortEndScheduler();                   //调用硬件层关闭中断的处理函数
}
```

函数 vPortEndScheduler()在 port.c 中有定义，这个函数在移植 FreeRTOS 的时候要根据实际使用的处理器来编写，此处没有实现这个函数，只是简单的加了一行断言，函数如下：

```
void vPortEndScheduler( void )
```

```
{
    configASSERT( uxCriticalNesting == 1000UL );
}
```

10、函数 vTaskSuspendAll()

挂起任务调度器，调用此函数不需要关闭可屏蔽中断即可挂起任务调度器，此函数在文件 tasks.c 中有如下定义：

```
void vTaskSuspendAll( void )
{
    ++uxSchedulerSuspended;
}
```

可看出，此函数只是简单的将变量 uxSchedulerSuspended 加一，uxSchedulerSuspended 是挂起嵌套计数器，调度器挂起是支持嵌套的。使用函数 xTaskResumeAll()可以恢复任务调度器，调用了几次 vTaskSuspendAll()挂起调度器，同样的也得调用几次 xTaskResumeAll()才会最终恢复任务调度器。

假设现在有这样一种情况，任务 1 的优先级为 10，此时任务 1 由于等待队列(关于队列的知识后面会有专门的章节讲)TestQueue 而处于阻塞态。但是有段其他的代码调用函数 vTaskSuspendAll()挂起了任务调度器，在还没有调用 xTaskResumeAll()恢复任务调度器之前，有个在外部中断发生了，在中断服务程序里面调用函数 xQueueSendFromISR()向任务 1 发送了队列 TestQueue。如果任务调度器没有阻塞的话函数 xQueueSendFromISR()会使任务 1 进入就绪态，也就是将任务 1 添加到优先级 10 对应的就绪列表 pxReadyTasksLists[10]中，这样当任务切换的时候任务 1 就会运行。但是现在任务调度器由于函数 vTaskSuspendAll()而挂起，这个时候任务 1 就不是添加到任务就绪列表 pxReadyTasksLists[10]中了，而是添加到另一个叫做 xPendingReadyList 的列表中，xPendingReadyList 是个全局变量，在文件 tasks.c 中有定义。当调用函数 xTaskResumeAll()恢复调度器的时候就会将挂到列表 xPendingReadyList 中的任务重新移动到它们所对应的就绪列表 pxReadyTasksLists 中。

11、函数 xTaskResumeAll()

此函数用于将任务调度器从挂起壮态恢复，缩减后的函数代码如下：

```
BaseType_t xTaskResumeAll( void )
{
    TCB_t *pxTCB = NULL;
    BaseType_t xAlreadyYielded = pdFALSE;

    configASSERT( uxSchedulerSuspended );
    taskENTER_CRITICAL();                                                    (1)
    {
        --uxSchedulerSuspended;                                              (2)
        if( uxSchedulerSuspended == ( UBaseType_t ) pdFALSE )                (3)
        {
            if( uxCurrentNumberOfTasks > ( UBaseType_t ) 0U )
            {
                while( listLIST_IS_EMPTY( &xPendingReadyList ) == pdFALSE )  (4)
                {
```

```
                    pxTCB = ( TCB_t * ) listGET_OWNER_OF_HEAD_ENTRY\  (5)
                                ( ( &xPendingReadyList ) );
                    ( void ) uxListRemove( &( pxTCB->xEventListItem ) );  (6)
                    ( void ) uxListRemove( &( pxTCB->xStateListItem ) );  (7)
                    prvAddTaskToReadyList( pxTCB );                       (8)
                    if( pxTCB->uxPriority >= pxCurrentTCB->uxPriority )   (9)
                    {
                        xYieldPending = pdTRUE;
                    }
                    else
                    {
                        mtCOVERAGE_TEST_MARKER();
                    }
                }
    /*************************************************************************/
    /*************************省略部分代码*************************************/
    /*************************************************************************/
                if( xYieldPending != pdFALSE )                            (10)
                {
                    #if( configUSE_PREEMPTION != 0 )
                    {
                        xAlreadyYielded = pdTRUE;                         (11)
                    }
                    #endif
                    taskYIELD_IF_USING_PREEMPTION();                      (12)
                }
                else
                {
                    mtCOVERAGE_TEST_MARKER();
                }
            }
        }
        else
        {
            mtCOVERAGE_TEST_MARKER();
        }
    }
    taskEXIT_CRITICAL();                                                  (13)
    return xAlreadyYielded;                                               (14)
}
```

(1)、进入临界区。

(2)、调度器挂起嵌套计数器 uxSchedulerSuspended 减一。

(3)、如果 uxSchedulerSuspended 为 0 说明所有的挂起都已经解除，调度器可以开始运行了。

(4)、while()循环处理列表 xPendingReadyList，只要列表 xPendingReadyList 不为空，说明还有任务挂到了列表 xPendingReadyList 上，这里需要将这些任务从列表 xPendingReadyList 上移除并添加到这些任务所对应的就绪列表中。

(5)、遍历列表 xPendingReadyList，获取挂到列表 xPendingReadyList 上的任务对应的任务控制块。

(6)、将任务从事件列表上删除。

(7)、将任务从壮态列表上移除。

(8)、调用函数 prvAddTaskToReadyList()将任务添加到就绪列表中。

(9)、判断任务的优先级是否高于当前正在运行的任务，如果是的话需要将 xYieldPending 标记为 pdTRUE，表示需要进行任务切换。

(10)、根据(9)得出需要进行任务切换。

(11)、标记在函数 xTaskResumeAll()中进行了任务切换，变量 xAlreadyYielded 用于标记在函数 xTaskResumeAll()中是否有进行任务切换。

(12)、调用函数 taskYIELD_IF_USING_PREEMPTION()进行任务切换，此函数本质上是一个宏，其实最终调用是通过调用函数 portYIELD()来完成任务切换的。

(13)、退出临界区。

(14)、返回变量 xAlreadyYielded，如果为 pdTRUE 的话表示在函数 xTaskResumeAll()中进行了任务切换，如果为 pdFALSE 的话表示没有进行任务切换。

12、函数 vTaskStepTick()

此函数在使用 FreeRTOS 的低功耗 tickless 模式的时候会用到，即宏 configUSE_TICKLESS_IDLE 为 1。当使能低功耗 tickless 模式以后在执行空闲任务的时候系统时钟节拍中断就会停止运行，系统时钟中断停止运行的这段时间必须得补上，这个工作就是由函数 vTaskStepTick()来完成的，此函数在文件 tasks.c 中有如下定义：

```
void vTaskStepTick( const TickType_t xTicksToJump )
{
    configASSERT( ( xTickCount + xTicksToJump ) <= xNextTaskUnblockTime );
    xTickCount += xTicksToJump;                                          (1)
    traceINCREASE_TICK_COUNT( xTicksToJump );
}
```

(1)、函数参数 xTicksToJump 是要加上的时间值，系统节拍计数器 xTickCount 加上这个时间值得到新的系统时间。关于 xTicksToJump 这个时间值的确定后面在讲解 FreeRTOS 的低功耗模式的时候会详细的讲解。

第十一章 FreeRTOS 其他任务 API 函数

前面几章我们花费了大量的精力来学习 FreeRTOS 的任务管理，但是真正涉及到的与任务相关的 API 函数只有那么几个。但是 FreeRTOS 还有很多与任务相关的 API 函数，不过这些 API 函数大多都是辅助函数了，本章我们就来看一下这些与任务相关的其他的 API 函数。本章分为如下几部分：

11.1 任务相关 API 函数预览
11.2 任务相关 API 函数详解
11.3 任务壮态查询 API 函数实验
11.4 任务运行时间壮态统计实验

11.1 任务相关 API 函数预览

先通过一个表 11.1.1 来看一下这些与任务相关的其他 API 函数都有哪些：

函数	描述
uxTaskPriorityGet()	查询某个任务的优先级。
vTaskPrioritySet()	改变某个任务的任务优先级。
uxTaskGetSystemState()	获取系统中任务状态。
vTaskGetInfo()	获取某个任务信息。
xTaskGetApplicationTaskTag()	获取某个任务的标签(Tag)值。
xTaskGetCurrentTaskHandle()	获取当前正在运行的任务的任务句柄。
xTaskGetHandle()	根据任务名字查找某个任务的句柄
xTaskGetIdleTaskHandle()	获取空闲任务的任务句柄。
uxTaskGetStackHighWaterMark()	获取任务的堆栈的历史剩余最小值，FreeRTOS 中叫做“高水位线”
eTaskGetState()	获取某个任务的壮态，这个壮态是 eTaskState 类型。
pcTaskGetName()	获取某个任务的任务名字。
xTaskGetTickCount()	获取系统时间计数器值。
xTaskGetTickCountFromISR()	在中断服务函数中获取时间计数器值
xTaskGetSchedulerState()	获取任务调度器的壮态，开启或未开启。
uxTaskGetNumberOfTasks()	获取当前系统中存在的任务数量。
vTaskList()	以一种表格的形式输出当前系统中所有任务的详细信息。
vTaskGetRunTimeStats()	获取每个任务的运行时间。
vTaskSetApplicationTaskTag()	设置任务标签(Tag)值。
SetThreadLocalStoragePointer()	设置线程本地存储指针
GetThreadLocalStoragePointer()	获取线程本地存储指针

表 11.1.1 任务相关 API 函数

这些 API 函数在 FreeRTOS 官网上都有，如图 11.1.2 所示：

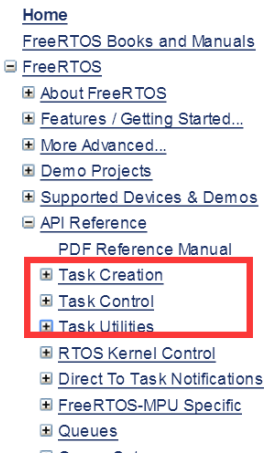


图 11.1.2 任务相关 API 函数

11.2 任务相关 API 函数详解

1、函数 uxTaskPriorityGet()

此函数用来获取指定任务的优先级，要使用此函数的话宏 INCLUDE_uxTaskPriorityGet 应该定义为 1，函数原型如下：

```
UBaseType_t  uxTaskPriorityGet( TaskHandle_t  xTask )
```

参数：

xTask： 要查找的任务的任务句柄。

返回值： 获取到的对应的任务的优先级。

2、函数 vTaskPrioritySet()

此函数用于改变某一个任务的任务优先级，要使用此函数的话宏 INCLUDE_vTaskPrioritySet 应该定义为 1，函数原型如下：

```
void vTaskPrioritySet( TaskHandle_t    xTask,
                       UBaseType_t     uxNewPriority )
```

参数：

xTask： 要查找的任务的任务句柄。

uxNewPriority: 任务要使用的新的优先级，可以是 0~ configMAX_PRIORITIES – 1。

返回值： 无。

3、uxTaskGetSystemState()

此函数用于获取系统中所有任务的任务壮态，每个任务的壮态信息保存在一个 TaskStatus_t 类型的结构体里面，这个结构体里面包含了任务的任务句柄、任务名字、堆栈、优先级等信息，要使用此函数的话宏 configUSE_TRACE_FACILITY 应该定义为 1，函数原型如下：

```
UBaseType_t uxTaskGetSystemState( TaskStatus_t * const      pxTaskStatusArray,
                                  const UBaseType_t         uxArraySize,
                                  uint32_t * const          pulTotalRunTime )
```

参数：

pxTaskStatusArray： 指向 TaskStatus_t 结构体类型的数组首地址，每个任务至少需要一个 TaskStatus_t 结构体，任务的数量可以使用函数 uxTaskGetNumberOfTasks()。结构体 TaskStatus_t 在文件 task.h 中有如下定义：

```
typedef struct xTASK_STATUS
{
    TaskHandle_t    xHandle;        //任务句柄
    const char *    pcTaskName;     //任务名字
    UBaseType_t     xTaskNumber;    //任务编号
```

```
    eTaskState      eCurrentState;      //当前任务壮态，eTaskState 是一个枚举类型
    UBaseType_t     uxCurrentPriority;  //任务当前的优先级
    UBaseType_t     uxBasePriority;     //任务基础优先级
    uint32_t        ulRunTimeCounter;//任务运行的总时间
    StackType_t *   pxStackBase;        //堆栈基地址
    uint16_t        usStackHighWaterMark;//从任务创建以来任务堆栈剩余的最小大小，此
                                         //值如果太小的话说明堆栈有溢出的风险。
} TaskStatus_t;
```

uxArraySize: 保存任务壮态数组的数组的大小。

pulTotalRunTime: 如果 configGENERATE_RUN_TIME_STATS 为 1 的话此参数用来保存系统总的运行时间。

返回值： 统计到的任务壮态的个数，也就是填写到数组 pxTaskStatusArray 中的个数，此值应该等于函数 uxTaskGetNumberOfTasks()的返回值。如果参数 uxArraySize 太小的话返回值可能为 0。

4、函数 vTaskGetInfo()

此函数也是用来获取任务壮态的，但是是获取指定的单个任务的壮态的，任务的壮态信息填充到参数 pxTaskStatus 中，这个参数也是 TaskStatus_t 类型的。要使用此函数的话宏 configUSE_TRACE_FACILITY 要定义为 1，函数原型如下：

```
void vTaskGetInfo( TaskHandle_t     xTask,
                   TaskStatus_t *   pxTaskStatus,
                   BaseType_t       xGetFreeStackSpace,
                   eTaskState       eState )
```

参数：

xTask： 要查找的任务的任务句柄。

pxTaskStatus: 指向类型为 TaskStatus_t 的结构体变量。

xGetFreeStackSpace: 在结构体 TaskStatus_t 中有个字段 usStackHighWaterMark 来保存自任务运行以来任务堆栈剩余的历史最小大小，这个值越小说明越接近堆栈溢出，但是计算这个值需要花费一点时间，所以我们可以通过将 xGetFreeStackSpace 设置为 pdFALSE 来跳过这个步骤，当设置为 pdTRUE 的时候就会检查堆栈的历史剩余最小值。

eState: 结构体 TaskStatus_t 中有个字段 eCurrentState 用来保存任务运行壮态，这个字段是 eTaskState 类型的，这是个枚举类型，在 task.h 中有如下定义：

```
typedef enum
{
    eRunning = 0,   //运行壮态
    eReady,         //就绪态
    eBlocked,       //阻塞态
    eSuspended,     //挂起态
    eDeleted,       //任务被删除
```

```
    eInvalid            //无效
} eTaskState;
```

获取任务运行壮态会耗费不少时间，所以为了加快函数 vTaskGetInfo()的执行速度结构体 TaskStatus_t 中的字段 eCurrentState 就可以由用户直接赋值，参数 eState 就是要赋的值。如果不在乎这点时间，那么可以将 eState 设置为 eInvalid，这样任务的壮态信息就由函数 vTaskGetInfo()去想办法获取。

返回值： 无。

5、函数 xTaskGetApplicationTaskTag()

此函数用于获取任务的 Tag(标签)值，任务控制块中有个成员变量 pxTaskTag 来保存任务的标签值。标签的功能由用户自行决定，此函数就是用来获取这个标签值的，FreeRTOS 系统内核是不会使用到这个标签的。要使用此函数的话宏 configUSE_APPLICATION_TASK_TAG 必须为 1，函数原型如下：

```
TaskHookFunction_t    xTaskGetApplicationTaskTag( TaskHandle_t    xTask )
```

参数：

xTask： 要获取标签值的任务对应的任务句柄，如果为 NULL 的话就获取当前正在运行的任务标签值。

返回值： 任务的标签值。

6、函数 xTaskGetCurrentTaskHandle()

此函数用于获取当前任务的任务句柄，其实获取到的就是任务控制块，在前面讲解任务创建函数的时候说过任务句柄就是任务控制。如果要使用此函数的话宏 INCLUDE_xTaskGetCurrentTaskHandle 应该为 1，函数原型如下：

```
TaskHandle_t   xTaskGetCurrentTaskHandle( void )
```

参数： 无

返回值： 当前任务的任务句柄。

7、函数 xTaskGetHandle()

此函数根据任务名字获取任务的任务句柄，在使用函数 xTaskCreate()或 xTaskCreateStatic()创建任务的时候都会给任务分配一个任务名，函数 xTaskGetHandle()就是使用这个任务名字来查询其对应的任务句柄的。要使用此函数的话宏 INCLUDE_xTaskGetHandle 应该设置为 1，此函数原型如下：

```
TaskHandle_t xTaskGetHandle( const char *       pcNameToQuery )
```

参数：

pcNameToQuery： 任务名，C 语言字符串。

返回值：

NULL： 没有任务名 pcNameToQuery 所对应的任务。
其他值： 任务名 pcNameToQuery 所对应的任务句柄

8、函数 xTaskGetIdleTaskHandle()

此函数用于返回空闲任务的任务句柄，要使用此函数的话宏 INCLUDE_xTaskGetIdleTaskHandle 必须为 1，函数原型如下：

```
TaskHandle_t xTaskGetIdleTaskHandle( void )
```

参数： 无

返回值： 空闲任务的任务句柄。

9、函数 uxTaskGetStackHighWaterMark()

每个任务都有自己的堆栈，堆栈的总大小在创建任务的时候就确定了，此函数用于检查任务从创建好到现在的历史剩余最小值，这个值越小说明任务堆栈溢出的可能性就越大！FreeRTOS 把这个历史剩余最小值叫做“高水位线”。此函数相对来说会多耗费一点时间，所以在代码调试阶段可以使用，产品发布的时候最好不要使用。要使用此函数的话宏 INCLUDE_uxTaskGetStackHighWaterMark 必须为 1，此函数原型如下：

```
UBaseType_t uxTaskGetStackHighWaterMark( TaskHandle_t    xTask )
```

参数：

xTask： 要查询的任务的任务句柄，当这个参数为 NULL 的话说明查询自身任务(即调用函数 uxTaskGetStackHighWaterMark()的任务)的“高水位线”。

返回值： 任务堆栈的“高水位线”值，也就是堆栈的历史剩余最小值。

10、函数 eTaskGetState()

此函数用于查询某个任务的运行壮态，比如：运行态、阻塞态、挂起态、就绪态等，返回值是个枚举类型。要使用此函数的话宏 INCLUDE_eTaskGetState 必须为 1，函数原型如下：

```
eTaskState eTaskGetState( TaskHandle_t      xTask )
```

参数：

xTask： 要查询的任务的任务句柄。

返回值： 返回值为 eTaskState 类型，这是个枚举类型，在文件 task.h 中有定义，前面讲解函数 vTaskGetInfo()的时候已经讲过了。

11、函数 pcTaskGetName()

根据某个任务的任务句柄来查询这个任务对应的任务名，函数原型如下：

```
char *pcTaskGetName( TaskHandle_t   xTaskToQuery )
```

参数：

xTaskToQuery： 要查询的任务的任务句柄，此参数为 NULL 的话表示查询自身任务(调用函数 pcTaskGetName())的任务名字

返回值： 返回任务所对应的任务名。

12、函数 xTaskGetTickCount()

此函数用于查询任务调度器从启动到现在时间计数器 xTickCount 的值。xTickCount 是系统的时钟节拍值，并不是真实的时间值。每个滴答定时器中断 xTickCount 就会加 1，一秒钟滴答定时器中断多少次取决于宏 configTICK_RATE_HZ。理论上 xTickCount 存在溢出的问题，但是这个溢出对于 FreeRTOS 的内核没有影响，但是如果用户的应用程序有使用到的话就要考虑溢出了。什么时候溢出取决于宏 configUSE_16_BIT_TICKS，当此宏为 1 的时候 xTixkCount 就是个 16 位的变量，当为 0 的时候就是个 32 位的变量。函数原型如下：

```
TickType_t    xTaskGetTickCount( void )
```

参数： 无。

返回值： 时间计数器 xTickCount 的值。

13、函数 xTaskGetTickCountFromISR()

此函数是 xTaskGetTickCount()的中断级版本，用于在中断服务函数中获取时间计数器 xTickCount 的值，函数原型如下：

```
TickType_t    xTaskGetTickCountFromISR( void )
```

参数： 无。

返回值： 时间计数器 xTickCount 的值。

14、函数 xTaskGetSchedulerState()

此函数用于获取 FreeRTOS 的任务调度器运行情况：运行？关闭？还是挂起！要使用此函数的话宏 INCLUDE_xTaskGetSchedulerState 必须为 1，此函数原型如下：

```
BaseType_t    xTaskGetSchedulerState( void )
```

参数： 无。

返回值：

taskSCHEDULER_NOT_STARTED： 调度器未启动，调度器的启动是通过函数 vTaskStartScheduler() 来完成，所以在函数 vTaskStartScheduler() 未调用之前调用函数

xTaskGetSchedulerState()的话就会返回此值。

taskSCHEDULER_RUNNING： 调度器正在运行。

taskSCHEDULER_SUSPENDED： 调度器挂起。

15、函数 uxTaskGetNumberOfTasks()

此函数用于查询系统当前存在的任务数量，函数原型如下：

```
UBaseType_t  uxTaskGetNumberOfTasks( void )
```

参数： 无。

返回值： 当前系统中存在的任务数量， 此值=挂起态的任务+阻塞态的任务+就绪态的任务+空闲任务+运行态的任务。

16、函数 vTaskList()

此函数会创建一个表格来描述每个任务的详细信息，如图 11.2.1 所示：

```
Name          State   Priority  Stack  Num
*********************************************
Print           R       4       331     29
Math7           R       0       417     7
Math8           R       0       407     8
QConsB2         R       0       53      14
QProdB5         R       0       52      17
QConsB4         R       0       53      16
SEM1            R       0       50      27
SEM1            R       0       50      28
IDLE            R       0       64      0
Math1           R       0       436     1
Math2           R       0       436     2
```

图 11.2.1 任务壮态信息表

表中的信息如下：

Name： 创建任务的时候给任务分配的名字。

State： 任务的壮态信息，B 是阻塞态，R 是就绪态，S 是挂起态，D 是删除态。

Priority：任务优先级。

Stack： 任务堆栈的“高水位线”，就是堆栈历史最小剩余大小。

Num： 任务编号，这个编号是唯一的，当多个任务使用同一个任务名的时候可以通过此编号来做区分。

函数原型如下：

```
void vTaskList( char * pcWriteBuffer )
```

参数：

pcWriteBuffer： 保存任务壮态信息表的存储区。存储区要足够大来保存任务状态信息表。

返回值： 无

17、函数 vTaskGetRunTimeStats()

FreeRTOS 可以通过相关的配置来统计任务的运行时间信息，任务的运行时间信息提供了每个任务获取到 CPU 使用权总的时间。函数 vTaskGetRunTimeStats()会将统计到的信息填充到一个表里面，表里面提供了每个任务的运行时间和其所占总时间的百分比，如图 11.2.2 所示：

```
Task            Abs Time        % Time
****************************************

uIP             12050           <1%
IDLE            587724          24%
QProdB2         2172            <1%
QProdB3         10002           <1%
QProdB5         11504           <1%
QConsB6         11671           <1%
PolSEM1         60033           2%
PolSEM2         59957           2%
IntMath         349246          14%
MuLow           36619           1%
Gen0            579715          24%
```

图 11.2.2 任务运行时间表

函数 vTaskGetRunTimeStats() 是一个很实用的函数，要使用此函数的话宏 configGENERATE_RUN_TIME_STATS 和 configUSE_STATS_FORMATTING_FUNCTIONS 必须都为 1。如果宏 configGENERATE_RUN_TIME_STATS 为 1 的话还需要实现一下几个宏定义：

- **portCONFIGURE_TIMER_FOR_RUN_TIME_STATS()**，此宏用来初始化一个外设来提供时间统计功能所需的时基，一般是定时器/计数器。这个时基的分辨率一定要比 FreeRTOS 的系统时钟高，一般这个时基的时钟精度比系统时钟的高 10~20 倍就可以了。
- **portGET_RUN_TIME_COUNTER_VALUE()或者 portALT_GET_RUN_TIME_COUNTER_VALUE(Time)**，这两个宏实现其中一个就行了，这两个宏用于提供当前的时基的时间值。

函数原型如下：

```
void vTaskGetRunTimeStats( char *pcWriteBuffer )
```

参数：

pcWriteBuffer： 保存任务时间信息的存储区。存储区要足够大来保存任务时间信息。

返回值： 无

18、函数 vTaskSetApplicationTaskTag()

此函数是为高级用户准备的，此函数用于设置某个任务的标签值 ，这个标签值的具体函数和用法由用户自行决定，FreeRTOS 内核不会使用这个标签值，如果要使用此函数的话宏 configUSE_APPLICATION_TASK_TAG 必须为 1，函数原型如下：

```
void vTaskSetApplicationTaskTag(    TaskHandle_t            xTask,
                                    TaskHookFunction_t      pxHookFunction )
```

参数：

xTask： 要设置标签值的任务，此值为 NULL 的话表示设置自身任务的标签值。

pxHookFunction： 要设置的标签值，这是一个 TaskHookFunction_t 类型的函数指针，但是也可以设置为其他值。

返回值： 无

19、函数 SetThreadLocalStoragePointer()

此函数用于设置线程本地存储指针的值，每个任务都有它自己的指针数组来作为线程本地存储，使用这些线程本地存储可以用来在任务控制块中存储一些应用信息，这些信息只属于任务自己的。线程本地存储指针数组的大小由宏 configNUM_THREAD_LOCAL_STORAGE_POINTERS 来决定的。如果要使用此函数的话宏 configNUM_THREAD_LOCAL_STORAGE_POINTERS 不能为 0，宏的具体值是本地存储指针数组的大小，函数原型如下：

```
void vTaskSetThreadLocalStoragePointer( TaskHandle_t    xTaskToSet,
                                        BaseType_t      xIndex,
                                        void *          pvValue )
```

参数：

xTaskToSet： 要设置线程本地存储指针的任务的任务句柄，如果是 NULL 的话表示设置任务自身的线程本地存储指针。

xIndex： 要设置的线程本地存储指针数组的索引。

pvValue: 要存储的值。

返回值： 无

20、函数 GetThreadLocalStoragePointer()

此函数用于获取线程本地存储指针的值，如果要使用此函数的话宏 configNUM_THREAD_LOCAL_STORAGE_POINTERS 不能为 0，函数原型如下：

```
void *pvTaskGetThreadLocalStoragePointer( TaskHandle_t    xTaskToQuery,
                                          BaseType_t      xIndex )
```

参数：

xTaskToSet： 要获取的线程本地存储指针的任务句柄，如果是 NULL 的话表示获取任务自身的线程本地存储指针。

xIndex： 要获取的线程本地存储指针数组的索引。

返回值： 获取到的线程本地存储指针的值。

11.3 任务壮态查询 API 函数实验

11.3.1 实验程序设计

FreeRTOS 与任务相关的 API 函数中有很多是与任务壮态或者信息查询有关的，比如函数 uxTaskGetSystemState()、vTaskGetInfo()、eTaskGetState()和 vTaskList()。本实验我们就来学习这

些函数的使用方法。

1、实验目的

学习使用 FreeRTOS 与任务壮态或者信息查询有关的 API 函数，包括 uxTaskGetSystemState()、vTaskGetInfo()、eTaskGetState()和 vTaskList()。

2、实验设计

本实验设计三个任务：start_task、led0_task 和 query_task ，这三个任务的任务功能如下：

start_task：用来创建其他 2 个任务。

led0_task ：控制 LED0 灯闪烁，提示系统正在运行。

query_task ：任务壮态和信息查询任务，在此任务中学习使用与任务的壮态和信息查询有关的 API 函数。

实验需要一个按键 KEY_UP，这四个按键的功能如下：

KEY_UP: 控制程序的运行步骤。

3、实验工程

FreeRTOS 实验 11-1 FreeRTOS 任务壮态或信息查询。

4、实验程序与分析

●任务设置

实验中任务优先级、堆栈大小和任务句柄等的设置如下：

```
#define START_TASK_PRIO          1        //任务优先级
#define START_STK_SIZE           128      //任务堆栈大小
TaskHandle_t StartTask_Handler;           //任务句柄
void start_task(void *pvParameters);      //任务函数

#define LED0_TASK_PRIO           2        //任务优先级
#define LED0_STK_SIZE            128      //任务堆栈大小
TaskHandle_t Led0Task_Handler;            //任务句柄
void led0_task(void *pvParameters);       //任务函数

#define QUERY_TASK_PRIO          3        //任务优先级
#define QUERY_STK_SIZE           256      //任务堆栈大小
TaskHandle_t QueryTask_Handler;           //任务句柄
void query_task(void *pvParameters);      //任务函数

char InfoBuffer[1000];                    //保存信息的数组
```

● **main()函数**

```
int main(void)
{
    HAL_Init();                           //初始化 HAL 库
    Stm32_Clock_Init(360,25,2,8);         //设置时钟,180Mhz
    delay_init(180);                      //初始化延时函数
    uart_init(115200);                    //初始化串口
```

```
    LED_Init();                                 //初始化 LED
    KEY_Init();                                 //初始化按键
    SDRAM_Init();                               //初始化 SDRAM
    LCD_Init();                                 //初始化 LCD

    POINT_COLOR = RED;
    LCD_ShowString(30,10,200,16,16,"Apollo STM32F4/F7");
    LCD_ShowString(30,30,200,16,16,"FreeRTOS Examp 11-1");
    LCD_ShowString(30,50,200,16,16,"Task Info Query");
    LCD_ShowString(30,70,200,16,16,"ATOM@ALIENTEK");
    LCD_ShowString(30,90,200,16,16,"2016/10/19");

    //创建开始任务
    xTaskCreate((TaskFunction_t     )start_task,            //任务函数
                (const char*        )"start_task",          //任务名称
                (uint16_t           )START_STK_SIZE,        //任务堆栈大小
                (void*              )NULL,                  //传递给任务函数的参数
                (UBaseType_t        )START_TASK_PRIO,       //任务优先级
                (TaskHandle_t*      )&StartTask_Handler);   //任务句柄
    vTaskStartScheduler();                                  //开启任务调度
}
```

在 main 函数中我们主要完成硬件的初始化，在硬件初始化完成以后创建了任务 start_task()并且开启了 FreeRTOS 的任务调度。

● **任务函数**

```
//led0 任务函数
void led0_task(void *pvParameters)
{

    while(1)
    {
        LED0=~LED0;
        vTaskDelay(500);                    //延时 500ms，也就是 500 个时钟节拍
    }
}

//query 任务函数
void query_task(void *pvParameters)
{
    u32 TotalRunTime;
    UBaseType_t ArraySize,x;
    TaskStatus_t *StatusArray;

    //第一步：函数 uxTaskGetSystemState()的使用
```

```
printf("/********第一步：函数 uxTaskGetSystemState()的使用**********/\r\n");
ArraySize=uxTaskGetNumberOfTasks();                    //获取系统任务数量      (1)
StatusArray=pvPortMalloc(ArraySize*sizeof(TaskStatus_t));    //申请内存          (2)
if(StatusArray!=NULL)                                  //内存申请成功
{
    ArraySize=uxTaskGetSystemState((TaskStatus_t*    )StatusArray,             (3)
                                   (UBaseType_t      )ArraySize,
                                   (uint32_t*        )&TotalRunTime);
    printf("TaskName\t\tPriority\t\tTaskNumber\t\t\r\n");
    for(x=0;x<ArraySize;x++)
    {
        //通过串口打印出获取到的系统任务的有关信息，比如任务名称、
        //任务优先级和任务编号。
        printf("%s\t\t%d\t\t\t%d\t\t\t\r\n",                                  (4)
                StatusArray[x].pcTaskName,
                (int)StatusArray[x].uxCurrentPriority,
                (int)StatusArray[x].xTaskNumber);
    }
}
vPortFree(StatusArray); //释放内存                                            (5)
printf("/*************************结束**************************/\r\n");
printf("按下 KEY_UP 键继续!\r\n\r\n\r\n");
while(KEY_Scan(0)!=WKUP_PRES) delay_ms(10);            //等待 KEY_UP 键按下

//第二步：函数 vTaskGetInfo()的使用
TaskHandle_t TaskHandle;
TaskStatus_t TaskStatus;

printf("/************第二步：函数 vTaskGetInfo()的使用*************/\r\n");
TaskHandle=xTaskGetHandle("led0_task");      //根据任务名获取任务句柄。        (6)
//获取 LED0_Task 的任务信息
vTaskGetInfo((TaskHandle_t      )TaskHandle,  //任务句柄                      (7)
             (TaskStatus_t*     )&TaskStatus, //任务信息结构体
             (BaseType_t        )pdTRUE,      //允许统计任务堆栈历史最小剩余大小
             (eTaskState        )eInvalid);   //函数自己获取任务运行壮态
//通过串口打印出指定任务的有关信息。
printf("任务名:                %s\r\n",TaskStatus.pcTaskName);                (8)
printf("任务编号:              %d\r\n",(int)TaskStatus.xTaskNumber);
printf("任务壮态:              %d\r\n",TaskStatus.eCurrentState);
printf("任务当前优先级:        %d\r\n",(int)TaskStatus.uxCurrentPriority);
printf("任务基优先级:          %d\r\n",(int)TaskStatus.uxBasePriority);
printf("任务堆栈基地址:        %#x\r\n",(int)TaskStatus.pxStackBase);
printf("任务堆栈历史剩余最小值:%d\r\n",TaskStatus.usStackHighWaterMark);
```

```
    printf("/*************************结束**************************/\r\n");
    printf("按下 KEY_UP 键继续!\r\n\r\n\r\n");
    while(KEY_Scan(0)!=WKUP_PRES) delay_ms(10);        //等待 KEY_UP 键按下

    //第三步：函数 eTaskGetState()的使用
    eTaskState TaskState;
    char TaskInfo[10];
    printf("/************第三步：函数 eTaskGetState()的使用*************/\r\n");
    TaskHandle=xTaskGetHandle("query_task");      //根据任务名获取任务句柄。          (9)
    TaskState=eTaskGetState(TaskHandle);          //获取 query_task 任务的任务壮态    (10)
    memset(TaskInfo,0,10);                        //数组清零
    switch((int)TaskState)                                                          (11)
    {
        case 0:sprintf(TaskInfo,"Running");break;
        case 1:sprintf(TaskInfo,"Ready");break;
        case 2:sprintf(TaskInfo,"Suspend");break;
        case 3:sprintf(TaskInfo,"Delete");break;
        case 4:sprintf(TaskInfo,"Invalid");break;
    }
    printf("任务壮态值:%d,对应的壮态为:%s\r\n",TaskState,TaskInfo);                 (12)
    printf("/*************************结束**************************/\r\n");
    printf("按下 KEY_UP 键继续!\r\n\r\n\r\n");
    while(KEY_Scan(0)!=WKUP_PRES) delay_ms(10);        //等待 KEY_UP 键按下

    //第四步：函数 vTaskList()的使用
    printf("/*************第三步：函数 vTaskList()的使用*************/\r\n");
    vTaskList(InfoBuffer);                        //获取所有任务的信息              (13)
    printf("%s\r\n",InfoBuffer);                  //通过串口打印所有任务的信息      (14)

    while(1)
    {
        LED1=~LED1;
        vTaskDelay(1000);                          //延时 1s，也就是 1000 个时钟节拍
    }
}
```

(1)、使用函数 uxTaskGetNumberOfTasks()获取当前系统中的任务数量，因为要根据任务数量给任务信息数组 StatusArray 分配内存。注意，这里 StatusArray 是个指向 TaskStatus_t 类型的指针，但是在使用的时候会把他当作一个数组来用。

(2)、调用函数 pvPortMalloc()给任务信息数组 StatusArray 分配内存，数组是 TaskStatus_t 类型。

(3)、调用函数 uxTaskGetSystemState()获取系统中所有任务的信息，并将获取到的信息保存在 StatusArray 中。

(4)、通过串口将获取到的所有任务的部分信息打印出来，这里并没有把所获取到的信息都

输出，只是将任务的任务名、任务优先级和任务编号做了输出。

(5)、任务信息数组 StatusArray 使用完毕，释放其内存。

(6)、调用函数 xTaskGetHandle()根据任务名来获取任务句柄，这里获取任务名为“led0_task”的任务句柄。我们在创建任务的时候一般都会保存任务的句柄，如果保存了任务句柄的话就可以直接使用。

(7)、调用函数 vTaskGetInfo()获取任务名为“led0_task”的任务信息，任务信息保存在 TaskStatus 中。获取任务信息的时候允许统计任务堆栈历史最小剩余大小，任务的运行壮态也是由函数 vTaskGetInfo()来统计。

(8)、通过串口输出获取到的任务 led0_task 的任务信息。

(9)、通过函数 xTaskGetHandle()获取任务名为“query_task”的任务句柄。

(10)、调用函数 eTaskGetState()获取任务的运行壮态。

(11)、通过函数 eTaskGetState()获取到的任务运行壮态是个枚举类型：eTaskState，枚举类型不同的值表示不同的含义，这里用字符串来描述这些枚举值的含义。

(12)、通过串口输出任务 query_task 的运行壮态信息。

(13)、调用函数 vTaskList()统计所有任务的信息，统计出来的任务信息存储在缓冲区 InfoBuffer 中，这些任务信息以表格的形式呈现，

(14)、通过串口输出保存在缓冲区 InfoBuffer 中的任务信息。

11.3.2 程序运行结果分析

编译并下载实验代码到开发板中，打开串口调试助手，然后按照任务函数 query_task()中的步骤一步步的测试分析。

● **第一步**

第一步是学习 uxTaskGetSystemState()函数的使用，通过此函数获取到系统中当前所有任务的信息，并且通过串口输出其中的一部分信息，这里我们输出了任务名、任务的优先级和任务的编号，如图 11.3.2.1 所示：

```
 LCD ID:5510
/********第一步：函数uxTaskGetSystemState()的使用**********/
TaskName                Priority                TaskNumber
query_task              3                       5
led0_task               2                       4
IDLE            0                       2
start_task              1                       1
Tmr Svc         31                      3
/**************************结束***************************/
按下KEY_UP键继续!
```

图 11.3.2.1 第一步

从图 11.3.2.1 可以看出空闲任务的任务优先级最低，为 0，定时器服务任务优先级最高，为 31。而且任务的优先级和任务的编号是不同的，优先级是用户自行设定的，而编号是根据任务创建的先后顺序来自动分配的。

● **第二步**

第二步是通过函数 vTaskGetInfo()来获取任务名为“led0_task”的任务信息，并通过串口输出，如图 11.3.2.2 所示：

```
/************第二步：函数vTaskGetInfo()的使用**************/
任务名:                led0_task
任务编号:              4
任务壮态:              2
任务当前优先级:        2
任务基优先级:          2
任务堆栈基地址:        0x20001478
任务堆栈历史剩余最小值:107
/****************************结束*****************************/
按下KEY_UP键继续!
```

图 11.3.2.2 第二步

图 11.3.2.2 中可以看出有关任务 led0_task 的详细信息，包括任务名、任务编号、任务当前优先级、任务基优先级，任务堆栈基地址和堆栈历史剩余最小值。

● **第三步**

第三步通过函数 eTaskGetStat()来任务 query_task 的运行壮态，串口调试助手显示如图 11.3.2.3 所示：

```
/************第三步：函数eTaskGetState()的使用*************/
任务壮态值:0,对应的壮态为:Running
/****************************结束*****************************/
按下KEY_UP键继续!
```

图 11.3.2.3 第三步

● **第四步**

第四步是通过函数 vTaskList()获取系统中当前所有任务的详细信息，并且将这些信息按照表格的形式组织在一起存放在用户提供的缓冲区中，例程中将这些信息放到了缓冲区 InfoBuffer 中。最后通过串口输出缓冲区 InfoBuffer 中的信息，如图 11.3.2.4 所示：

```
/*************第三步：函数vTaskList()的使用*************/
query_task      R       3       192     5
IDLE            R       0       111     2
led0_task       B       2       107     4
Tmr Svc         S       31      235     3

/****************************结束*****************************/
```

图 11.3.2.4 第四步

图 11.3.2.4 中各列的含义我们在讲解函数 vTaskList()的时候已经详细的讲解过了。

11.4 任务运行时间信息统计实验

FreeRTOS 可以通过函数 vTaskGetRunTimeStats()来统计每个任务使用 CPU 的时间，以及所使用的时间占总时间的比例。在调试代码的时候我们可以根据这个时间使用值来分析哪个任务的 CPU 占用率高，然后合理的分配或优化任务。本实验我们就来学习如何使用 FreeRTOS 的这个运行时间壮态统计功能。

11.4.1 相关宏的设置

要使用此功能的话宏 configGENERATE_RUN_TIME_STATS 必须为 1，还需要在定义其他两个宏：

portCONFIGURE_TIMER_FOR_RUN_TIME_STATS()：配置一个高精度定时器/计数器提供时基。

portGET_RUN_TIME_COUNTER_VALUE()：读取时基的时间值。

这三个宏在 FreeRTOSConfig.h 中定义，如下:

```
#define configGENERATE_RUN_TIME_STATS                1
```

```
#define portCONFIGURE_TIMER_FOR_RUN_TIME_STATS()    ConfigureTimeForRunTimeStats()
#define portGET_RUN_TIME_COUNTER_VALUE()                FreeRTOSRunTimeTicks
```

其中函数 ConfigureTimeForRunTimeStats()和变量 FreeRTOSRunTimeTicks 在 timer.c 里面定义，如下：

```
//FreeRTOS 时间统计所用的节拍计数器
volatile unsigned long long FreeRTOSRunTimeTicks;

//初始化 TIM3 使其为 FreeRTOS 的时间统计提供时基
void ConfigureTimeForRunTimeStats(void)
{
    //定时器 3 初始化，定时器时钟为 90M，分频系数为 90-1，所以定时器 3 的频率
    //为 90M/90=1M，自动重装载为 50-1，那么定时器周期就是 50us
    FreeRTOSRunTimeTicks=0;
    TIM3_Init(50-1,90-1);              //初始化 TIM3
}
```

函数 ConfigureTimeForRunTimeStats()其实就是初始化定时器，因为时间统计功能需要用户提供一个高精度的时钟，这里使用定时器 3。前面在讲函数 vTaskGetRunTimeStats()的时候说过，这个时钟的精度要比 FreeRTOS 的系统时钟高，大约 10~20 倍即可。FreeRTOS 系统时钟我们配置的是 1000HZ，周期 1ms，这里我们将定时器 3 的中断频率配置为 20KHZ，周期 50us，刚好是系统时钟频率的 20 倍。

定时器 3 初始化函数如下：

```
//通用定时器 3 中断初始化
//arr：自动重装值。
//psc：时钟预分频数
//定时器溢出时间计算方法:Tout=((arr+1)*(psc+1))/Ft us.
//Ft=定时器工作频率,单位:Mhz
//这里使用的是定时器 3!(定时器 3 挂在 APB1 上，时钟为 HCLK/2)
void TIM3_Init(u16 arr,u16 psc)
{
    TIM3_Handler.Instance=TIM3;                          //通用定时器 3
    TIM3_Handler.Init.Prescaler=psc;                     //分频系数
    TIM3_Handler.Init.CounterMode=TIM_COUNTERMODE_UP;    //向上计数器
    TIM3_Handler.Init.Period=arr;                        //自动装载值
    TIM3_Handler.Init.ClockDivision=TIM_CLOCKDIVISION_DIV1;//时钟分频因子
    HAL_TIM_Base_Init(&TIM3_Handler);

    //使能定时器 3 和定时器 3 更新中断：TIM_IT_UPDATE
    HAL_TIM_Base_Start_IT(&TIM3_Handler);
}

//定时器底册驱动，开启时钟，设置中断优先级
//此函数会被 HAL_TIM_Base_Init()函数调用
void HAL_TIM_Base_MspInit(TIM_HandleTypeDef *htim)
```

```
{
    if(htim->Instance==TIM3)
    {
        __HAL_RCC_TIM3_CLK_ENABLE();      //使能 TIM3 时钟
        HAL_NVIC_SetPriority(TIM3_IRQn,1,0); //抢占优先级 1，子优先级 0
        HAL_NVIC_EnableIRQ(TIM3_IRQn);      //开启 ITM3 中断
    }
}

//定时器 3 中断服务函数
void TIM3_IRQHandler(void)
{
    HAL_TIM_IRQHandler(&TIM3_Handler);
}

//回调函数，定时器中断服务函数调用
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    if(htim==(&TIM3_Handler))
    {
        FreeRTOSRunTimeTicks++;        //运行时间统计时基计数器加一
    }
}
```

FreeRTOSRunTimeTicks 是个全局变量，用来为时间统计功能提供时间，在定时器 3 的中断服务函数中进行更新。

11.4.2 实验程序设计

1、实验目的

学习使用 FreeRTOS 运行时间壮态统计函数 vTaskGetRunTimeStats()的使用。

2、实验设计

本实验设计四个任务：start_task、task1_task 、task2_task 和 RunTimeStats_task，这四个任务的任务功能如下：

start_task：用来创建其他 3 个任务。

task1_task ：应用任务 1，控制 LED0 灯闪烁，并且刷新 LCD 屏幕上指定区域的颜色。

task2_task ：应用任务 2，控制 LED1 灯闪烁，并且刷新 LCD 屏幕上指定区域的颜色。

RunTimeStats_task：获取按键值，当 KEY_UP 键按下以后就调用函数 vTaskGetRunTimeStats()获取任务的运行时间信息，并且将其通过串口输出到串口调试助手上。

实验需要一个按键 KEY_UP，用来获取系统中任务运行时间信息。

3、实验工程

FreeRTOS 实验 11-2 FreeRTOS 任务运行时间统计。

4、实验程序与分析

●任务设置

```
#define START_TASK_PRIO                 1          //任务优先级
#define START_STK_SIZE                  128        //任务堆栈大小
TaskHandle_t StartTask_Handler;                    //任务句柄
void start_task(void *pvParameters);               //任务函数

#define TASK1_TASK_PRIO                 2          //任务优先级
#define TASK1_STK_SIZE                  128        //任务堆栈大小
TaskHandle_t Task1Task_Handler;                    //任务句柄
void task1_task(void *pvParameters);               //任务函数

#define TASK2_TASK_PRIO                 3          //任务优先级
#define TASK2_STK_SIZE                  128        //任务堆栈大小
TaskHandle_t Task2Task_Handler;                    //任务句柄
void task2_task(void *pvParameters);               //任务函数

#define RUNTIMESTATS_TASK_PRIO          4          //任务优先级
#define RUNTIMESTATS_STK_SIZE           128        //任务堆栈大小
TaskHandle_t RunTimeStats_Handler;                 //任务句柄
void RunTimeStats_task(void *pvParameters);        //任务函数

char RunTimeInfo[400];                             //保存任务运行时间信息
```

● **main()函数**

```
int main(void)
{
    HAL_Init();                                    //初始化 HAL 库
    Stm32_Clock_Init(360,25,2,8);                  //设置时钟,180Mhz
    delay_init(180);                               //初始化延时函数
    uart_init(115200);                             //初始化串口
    LED_Init();                                    //初始化 LED
    KEY_Init();                                    //初始化按键
    SDRAM_Init();                                  //初始化 SDRAM
    LCD_Init();                                    //初始化 LCD

    POINT_COLOR = RED;
    LCD_ShowString(30,10,200,16,16,"Apollo STM32F4/F7");
    LCD_ShowString(30,30,200,16,16,"FreeRTOS Examp 11-2");
    LCD_ShowString(30,50,200,16,16,"Get Run Time Stats");
    LCD_ShowString(30,70,200,16,16,"ATOM@ALIENTEK");
    LCD_ShowString(30,90,200,16,16,"2016/10/21");

    //创建开始任务
```

```
    xTaskCreate((TaskFunction_t     )start_task,            //任务函数
                (const char*        )"start_task",          //任务名称
                (uint16_t           )START_STK_SIZE,        //任务堆栈大小
                (void*              )NULL,                  //传递给任务函数的参数
                (UBaseType_t        )START_TASK_PRIO,       //任务优先级
                (TaskHandle_t*      )&StartTask_Handler);   //任务句柄
    vTaskStartScheduler();                                  //开启任务调度
}
```

在 main 函数中我们主要完成硬件的初始化，在硬件初始化完成以后创建了任务 start_task()并且开启了 FreeRTOS 的任务调度。

● **任务函数**

```
//开始任务任务函数
void start_task(void *pvParameters)
{
    taskENTER_CRITICAL();           //进入临界区
    //创建 TASK1 任务
    xTaskCreate((TaskFunction_t     )task1_task,
                (const char*        )"task1_task",
                (uint16_t           )TASK1_STK_SIZE,
                (void*              )NULL,
                (UBaseType_t        )TASK1_TASK_PRIO,
                (TaskHandle_t*      )&Task1Task_Handler);
    //创建 TASK2 任务
    xTaskCreate((TaskFunction_t     )task2_task,
                (const char*        )"task2_task",
                (uint16_t           )TASK2_STK_SIZE,
                (void*              )NULL,
                (UBaseType_t        )TASK2_TASK_PRIO,
                (TaskHandle_t*      )&Task2Task_Handler);
    //创建 RunTimeStats 任务
    xTaskCreate((TaskFunction_t     )RunTimeStats_task,
                (const char*        )"RunTimeStats_task",
                (uint16_t           )RUNTIMESTATS_STK_SIZE,
                (void*              )NULL,
                (UBaseType_t        )RUNTIMESTATS_TASK_PRIO,
                (TaskHandle_t*      )&RunTimeStats_Handler);
    vTaskDelete(StartTask_Handler); //删除开始任务
    taskEXIT_CRITICAL();            //退出临界区
}

//task1 任务函数
void task1_task(void *pvParameters)
{
```

```
    u8 task1_num=0;

    POINT_COLOR = BLACK;
    LCD_DrawRectangle(5,110,115,314);  //画一个矩形
    LCD_DrawLine(5,130,115,130);       //画线
    POINT_COLOR = BLUE;
    LCD_ShowString(6,111,110,16,16,"Task1 Run:000");
    while(1)
    {
        task1_num++; //任务执 1 行次数加 1  注意 task1_num1 加到 255 的时候会清零!!
        LED0=!LED0;
        LCD_Fill(6,131,114,313,lcd_discolor[task1_num%14]); //填充区域
        LCD_ShowxNum(86,111,task1_num,3,16,0x80); //显示任务执行次数
        vTaskDelay(1000);                  //延时 1s，也就是 1000 个时钟节拍
    }
}

//task2 任务函数
void task2_task(void *pvParameters)
{
    u8 task2_num=0;

    POINT_COLOR = BLACK;
    LCD_DrawRectangle(125,110,234,314);    //画一个矩形
    LCD_DrawLine(125,130,234,130);         //画线
    POINT_COLOR = BLUE;
    LCD_ShowString(126,111,110,16,16,"Task2 Run:000");
    while(1)
    {
        task2_num++; //任务 2 执行次数加 1  注意 task1_num2 加到 255 的时候会清零!!
        LED1=!LED1;
        LCD_ShowxNum(206,111,task2_num,3,16,0x80);  //显示任务执行次数
        LCD_Fill(126,131,233,313,lcd_discolor[13-task2_num%14]); //填充区域
        vTaskDelay(1000);                  //延时 1s，也就是 1000 个时钟节拍
    }
}

//RunTimeStats 任务
void RunTimeStats_task(void *pvParameters)
{
    u8 key=0;
    while(1)
    {
```

```
        key=KEY_Scan(0);
        if(key==WKUP_PRES)
        {
            memset(RunTimeInfo,0,400);                  //信息缓冲区清零
            vTaskGetRunTimeStats(RunTimeInfo);          //获取任务运行时间信息        (1)
            printf("任务名\t\t\t 运行时间\t 运行所占百分比\r\n");
            printf("%s\r\n",RunTimeInfo);                                           (2)
        }
        vTaskDelay(10);                     //延时 10ms，也就是 1000 个时钟节拍
    }
}
```

(1)、调用函数 vTaskGetRunTimeStats()获取任务运行时间信息，此函数会统计任务的运行时间，并且将统计到的运行时间信息按照表格的形式组织在一起并存放在用户设置的缓冲区里面，缓冲区的首地址通过参数传递给函数 vTaskGetRunTimeStats()。

(2)、通过串口输出统计到的任务运行时间信息。

11.4.3 程序运行结果分析

编译并下载实验代码到开发板中，打开串口调试助手，按下 KEY_UP 按键输出任务的运行时间信息，如图 11.4.3.1 所示：

```
 LCD ID:5510
任务名                  运行时间        运行所占百分比
RunTimeStats_task       0               <1%
IDLE                    15572           96%
task1_task              311             1%
task2_task              311             1%
Tmr Svc                 1               <1%

任务名                  运行时间        运行所占百分比
RunTimeStats_task       349             1%
IDLE                    23293           94%
task1_task              546             2%
task2_task              546             2%
Tmr Svc                 1               <1%
```

图 11.4.3.1 任务运行时间信息

要注意，函数 vTaskGetRunTimeStats()相对来说会很耗时间，所以不要太过于频繁的调用此函数，测试阶段可以使用此函数来分析任务的运行情况。还有就是运行时间不是真正的运行时间，真正的时间值要乘以 50us。

第十二章 FreeRTOS 时间管理

在使用 FreeRTOS 的过程中我们通常会在一个任务函数中使用延时函数对这个任务延时，当执行延时函数的时候就会进行任务切换，并且此任务就会进入阻塞态，直到延时完成，任务重新进入就绪态。延时函数属于 FreeRTOS 的时间管理，本章我们就来学习一些 FreeRTOS 的这个时间管理过程，看看在调用延时函数以后究竟发生了什么？任务是如何进入阻塞态的，在延时完成以后任务又是如何从阻塞态恢复到就绪态的，本章分为如下几部分：

12.1 FreeRTOS 延时函数

12.3 FreeRTOS 系统时钟节拍

12.1 FreeRTOS 延时函数

12.1 函数 vTaskDelay()

学习过 uC/OSIII 的朋友应该知道，在 uC/OSIII 中延时函数 OSTimeDly()可以设置为三种模式：相对模式、周期模式和绝对模式。在 FreeRTOS 中延时函数也有相对模式和绝对模式，不过在 FreeRTOS 中不同的模式用的函数不同，其中函数 vTaskDelay()是相对模式(相对延时函数)，函数 vTaskDelayUntil()是绝对模式(绝对延时函数)。函数 vTaskDelay()在文件 tasks.c 中有定义，要使用此函数的话宏 INCLUDE_vTaskDelay 必须为 1，函数代码如下：

```
void vTaskDelay( const TickType_t xTicksToDelay )
{
    BaseType_t xAlreadyYielded = pdFALSE;
    //延时时间要大于 0。
    if( xTicksToDelay > ( TickType_t ) 0U )                          (1)
    {
        configASSERT( uxSchedulerSuspended == 0 );
        vTaskSuspendAll();                                           (2)
        {
            traceTASK_DELAY();
            prvAddCurrentTaskToDelayedList( xTicksToDelay, pdFALSE ); (3)
        }
        xAlreadyYielded = xTaskResumeAll();                          (4)
    }
    else
    {
        mtCOVERAGE_TEST_MARKER();
    }
    if( xAlreadyYielded == pdFALSE )                                 (5)
    {
        portYIELD_WITHIN_API();                                      (6)
    }
    else
    {
        mtCOVERAGE_TEST_MARKER();
    }
}
```

(1)、延时时间由参数 xTicksToDelay 来确定，为要延时的时间节拍数，延时时间肯定要大于 0。否则的话相当于直接调用函数 portYIELD()进行任务切换。

(2)、调用函数 vTaskSuspendAll()挂起任务调度器。

(3)、调用函数 prvAddCurrentTaskToDelayedList()将要延时的任务添加到延时列表 pxDelayedTaskList 或者 pxOverflowDelayedTaskList()中。后面会具体分析函数 prvAddCurrentTaskToDelayedList()。

(4)、调用函数 xTaskResumeAll()恢复任务调度器。

(5)、如果函数 xTaskResumeAll()没有进行任务调度的话那么在这里就得进行任务调度。

(6)、调用函数 portYIELD_WITHIN_API()进行一次任务调度。

12.2 函数 prvAddCurrentTaskToDelayedList()

函数 prvAddCurrentTaskToDelayedList()用于将当前任务添加到等待列表中，函数在文件 tasks.c 中有定义，缩减后的函数如下：

```
static void prvAddCurrentTaskToDelayedList( TickType_t  x            TicksToWait,
                                            const BaseType_t  xCanBlockIndefinitely )
{
    TickType_t xTimeToWake;
    const TickType_t xConstTickCount = xTickCount;                                    (1)

    #if( INCLUDE_xTaskAbortDelay == 1 )
    {
        //如果使能函数 xTaskAbortDelay()的话复位任务控制块的 ucDelayAborted 字段为
        //pdFALSE。
        pxCurrentTCB->ucDelayAborted = pdFALSE;
    }
    #endif

    if( uxListRemove( &( pxCurrentTCB->xStateListItem ) ) == ( UBaseType_t ) 0 )      (2)
    {
        portRESET_READY_PRIORITY( pxCurrentTCB->uxPriority, uxTopReadyPriority ); (3)
    }
    else
    {
        mtCOVERAGE_TEST_MARKER();
    }

    #if ( INCLUDE_vTaskSuspend == 1 )
    {
        if( ( xTicksToWait == portMAX_DELAY ) && ( xCanBlockIndefinitely != pdFALSE ) )(4)
        {
            vListInsertEnd( &xSuspendedTaskList, &( pxCurrentTCB->xStateListItem ) );  (5)
        }
        else
        {
            xTimeToWake = xConstTickCount + xTicksToWait;                             (6)
            listSET_LIST_ITEM_VALUE( &( pxCurrentTCB->xStateListItem ), \             (7)
                                     xTimeToWake );
            if( xTimeToWake < xConstTickCount )                                       (8)
            {
                vListInsert( pxOverflowDelayedTaskList, &( pxCurrentTCB->\            (9)
                             xStateListItem ) );
```

```
            }
            else
            {
                vListInsert( pxDelayedTaskList, &( pxCurrentTCB->xStateListItem ) );    (10)
                if( xTimeToWake < xNextTaskUnblockTime )                                (11)
                {
                    xNextTaskUnblockTime = xTimeToWake;                                 (12)
                }
                else
                {
                    mtCOVERAGE_TEST_MARKER();
                }
            }
        }
    }
    /*-----------------------------------------------------------*/
    /*-------------------------其他条件编译语句--------------------------*/
    /*-----------------------------------------------------------*/
}
```

(1)、读取进入函数 prvAddCurrentTaskToDelayedList()的时间点并保存在 xConstTickCount 中，后面计算任务唤醒时间点的时候要用到。xTickCount 是时钟节拍计数器，每个滴答定时器中断 xTickCount 都会加一。

(2)、要将当前正在运行的任务添加到延时列表中，肯定要先将当前任务从就绪列表中移除。

(3)、将当前任务从就绪列表中移除以后还要取消任务在 uxTopReadyPriority 中的就绪标记。也就是将 uxTopReadyPriority 中对应的 bit 清零。

(4)、延时时间为最大值 portMAX_DELAY，并且 xCanBlockIndefinitely 不为 pdFALSE(xCanBlockIndefinitely 不为 pdFALSE 的话表示允许阻塞任务)的话直接将当前任务添加到挂起列表中，任务就不用添加到延时列表中。

(5)、将当前任务添加到挂起列表 xSuspendedTaskList 的末尾。

(6)、计算任务唤醒时间点，也就是(1)中获取到的进入函数 prvAddCurrentTaskToDelayedList()的时间值 xConstTickCount 加上延时时间值 xTicksToWait。

(7)、将计算到的任务唤醒时间点值 xTimeToWake 写入到任务列表中壮态列表项的相应字段中。

(8)、计算得到的任务唤醒时间点小于 xConstTickCount，说明发生了溢出。全局变量 xTickCount 是 TickType_t 类型的，这是个 32 位的数据类型，因此在用 xTickCount 计算任务唤醒时间点 xTimeToWake 的时候的肯定会出现溢出的现象。FreeRTOS 针对此现象专门做了处理，在 FreeROTS 中定义了两个延时列表 xDelayedTaskList1 和 xDelayedTaskList2，并且也定义了两个指针 pxDelayedTaskList 和 pxOverflowDelayedTaskList 来访问这两个列表，在初始化列表函数 prvInitialiseTaskLists() 中指针 pxDelayedTaskList 指向了列表 xDelayedTaskList1，指针 pxOverflowDelayedTaskList 指向了列表 xDelayedTaskList2。这样发生溢出的话就将任务添加到 pxOverflowDelayedTaskList 所指向的列表中，如果没有溢出的话就添加到 pxDelayedTaskList 所指向的列表中。

(9)、如果发生了溢出的话就将当前任务添加到 pxOverflowDelayedTaskList 所指向的列表中。

(10)、如果没有发生溢出的话就将当前任务添加到 pxDelayedTaskList 所指向的列表中。

(11)、xNextTaskUnblockTime 是个全局变量，保存着距离下一个要取消阻塞的任务最小时间点值。 当 xTimeToWake 小于 xNextTaskUnblockTime 的话说明有个更小的时间点来了。

(12)、更新 xNextTaskUnblockTime 为 xTimeToWake。

12.3 函数 vTaskDelayUntil()

函数 vTaskDelayUntil()会阻塞任务，阻塞时间是一个绝对时间，那些需要按照一定的频率运行的任务可以使用函数 vTaskDelayUntil()。此函数再文件 tasks.c 中有如下定义：

```
void vTaskDelayUntil( TickType_t * const    pxPreviousWakeTime,
                      const TickType_t      xTimeIncrement )
{
    TickType_t xTimeToWake;
    BaseType_t xAlreadyYielded, xShouldDelay = pdFALSE;

    configASSERT( pxPreviousWakeTime );
    configASSERT( ( xTimeIncrement > 0U ) );
    configASSERT( uxSchedulerSuspended == 0 );

    vTaskSuspendAll();                                                    (1)
    {
        const TickType_t xConstTickCount = xTickCount;                    (2)
        xTimeToWake = *pxPreviousWakeTime + xTimeIncrement;               (3)

        if( xConstTickCount < *pxPreviousWakeTime )                       (4)
        {
            if( ( xTimeToWake < *pxPreviousWakeTime ) && ( xTimeToWake >\ (5)
                                                    xConstTickCount ) )
            {
                xShouldDelay = pdTRUE;                                    (6)
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
            }
        }
        else
        {
            if( ( xTimeToWake < *pxPreviousWakeTime ) || ( xTimeToWake > \ (7)
                                                    xConstTickCount ) )
            {
                xShouldDelay = pdTRUE;                                    (8)
            }
            else
```

```
            {
                mtCOVERAGE_TEST_MARKER();
            }
        }
        *pxPreviousWakeTime = xTimeToWake;                                  (9)

        if( xShouldDelay != pdFALSE )                                       (10)
        {
            traceTASK_DELAY_UNTIL( xTimeToWake );
            prvAddCurrentTaskToDelayedList( xTimeToWake - xConstTickCount, pdFALSE );(11)
        }
        else
        {
            mtCOVERAGE_TEST_MARKER();
        }
    }
    xAlreadyYielded = xTaskResumeAll();                                     (12)

    if( xAlreadyYielded == pdFALSE )
    {
            ortYIELD_WITHIN_API();
    }
    else
    {
        mtCOVERAGE_TEST_MARKER();
    }
}
```

参数：

pxPreviousWakeTime： 上一次任务延时结束被唤醒的时间点，任务中第一次调用函数 vTaskDelayUntil 的话需要将 pxPreviousWakeTime 初始化进入任务的 while()循环体的时间点值。在以后的运行中函数 vTaskDelayUntil()会自动更新 pxPreviousWakeTime。

xTimeIncrement： 任务需要延时的时间节拍数(相对于 pxPreviousWakeTime 本次延时的节拍数)。

(1)、挂起任务调度器。

(2)、记录进入函数 vTaskDelayUntil()的时间点值，并保存在 xConstTickCount 中。

(3)、根据延时时间 xTimeIncrement 来计算任务下一次要唤醒的时间点，并保存在 xTimeToWake 中。可以看出这个延时时间是相对于 pxPreviousWakeTime 的，也就是上一次任务被唤醒的时间点。pxPreviousWakeTime、xTimeToWake、xTimeIncrement 和 xConstTickCount 的关系如图 12.3.1 所示。

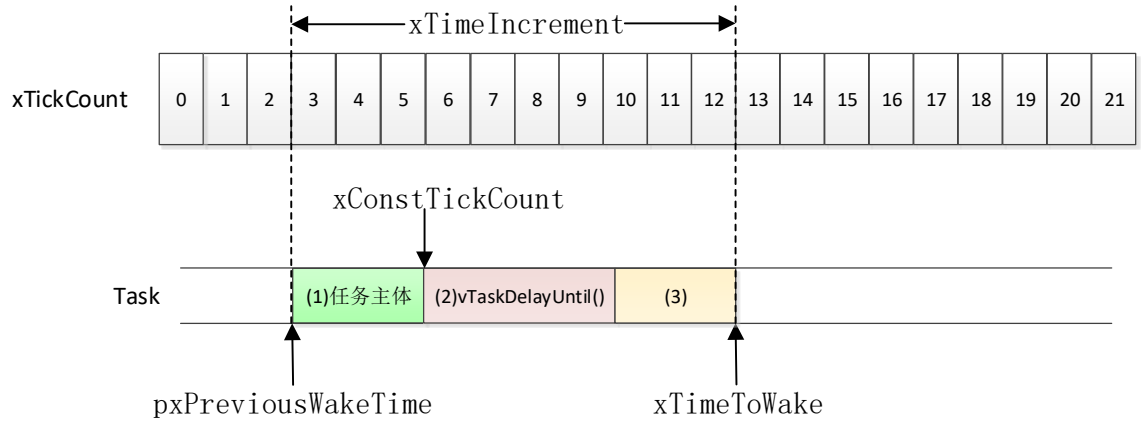


图 12.3.1 变量之间的关系

如 12.3.1 中 (1)为任务主体，也就是任务真正要做的工作，(2)是任务函数中调用 vTaskDelayUntil()对任务进行延时，(3)为其他任务在运行。任务的延时时间是 xTimeIncrement，这个延时时间是相对于 pxPreviousWakeTime 的，可以看出任务总的执行时间一定要小于任务的延时时间 xTimeIncrement！也就是说如果使用 vTaskDelayUntil()的话任务相当于任务的执行周期永远都是 xTimeIncrement，而任务一定要在这个时间内执行完成。这样就保证了任务永远按照一定的频率运行了，这个延时值就是绝对延时时间，因此函数 vTaskDelayUntil()也叫做绝对延时函数。

(4)、根据图 12.3.1 可以看出，理论上 xConstTickCount 要大于 pxPreviousWakeTime 的，但是也有一种情况会导致 xConstTickCount 小于 pxPreviousWakeTime，那就是 xConstTickCount 溢出了！

(5)、既然 xConstTickCount 都溢出了，那么计算得到的任务唤醒时间点肯定也是要溢出的，并且 xTimeToWake 肯定也是要大于 xConstTickCount 的。这种情况如图 12.3.2 所示：

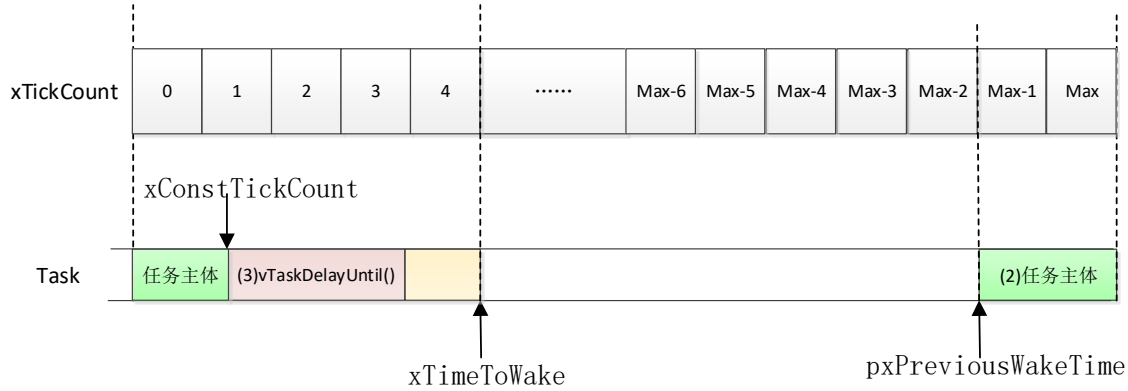


图 12.3.2 变量溢出

(6)、如果满足(5)条件的话就将 pdTRUE 赋值给 xShouldDelay，标记允许延时。

(7)、还有其他两种情况，一：只有 xTimeToWake 溢出，二：都没有溢出。只有 xTimeToWake 溢出的话如图 12.3.3 所示：

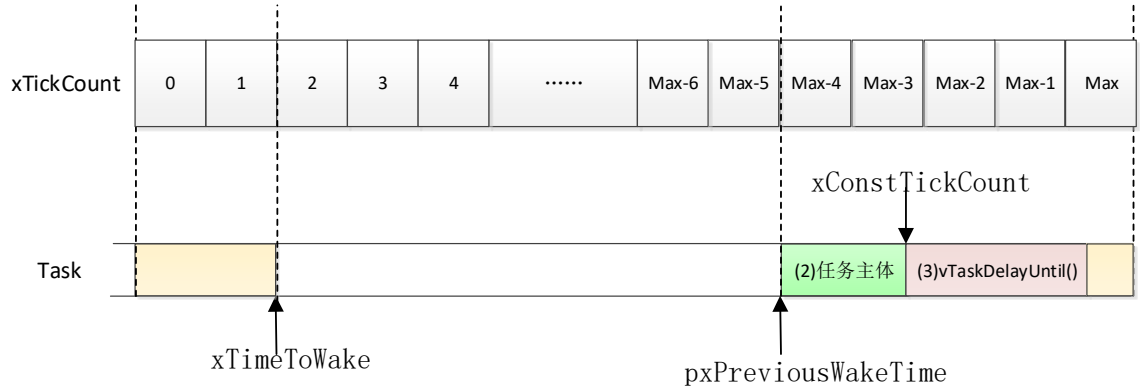


图 12.3.3 xTimeToWake 溢出

都不溢出的话就如图 12.3.1 所示，这两种情况都允许进行延时。

(8)、将 pdTRUE 赋值给 xShouldDelay，标记允许延时。

(9)、更新 pxPreviousWakeTime 的值，更新为 xTimeToWake，为本函数的下一次执行做准备。

(10)、经过前面的判断，允许进行任务延时。

(11)、调用函数 prvAddCurrentTaskToDelayedList()进行延时。函数的第一个参数是设置任务的阻塞时间，前面我们已经计算出了任务下一次唤醒时间点了，那么任务还需要阻塞的时间就是下一次唤醒时间点 xTimeToWake 减去当前的时间 xConstTickCount。而在函数 vTaskDelay()中只是简单的将这参数设置为 xTicksToDelay。

(12)、调用函数 xTaskResumeAll()恢复任务调度器。

函数 vTaskDelayUntil()的使用方法如下：

```
void TestTask( void * pvParameters )
{
    TickType_t    PreviousWakeTime;
    //延时 50ms，但是函数 vTaskDelayUntil()的参数需要设置的是延时的节拍数，不能直接
    //设置延时时间，因此使用函数 pdMS_TO_TICKS 将时间转换为节拍数。
    const TickType_t TimeIncrement = pdMS_TO_TICKS( 50 );

    PreviousWakeTime = xTaskGetTickCount();       //获取当前的系统节拍值
    for( ;; )
    {
        /*********************************************************************/
        /*************************任务主体**********************************/
        /*********************************************************************/

        //调用函数 vTaskDelayUntil 进行延时
        vTaskDelayUntil( &PreviousWakeTime, TimeIncrement);
    }
}
```

其实使用函数 vTaskDelayUntil()延时的任务也不一定就能周期性的运行，使用函数 vTaskDelayUntil()只能保证你按照一定的周期取消阻塞，进入就绪态。如果有更高优先级或者中断的话你还是得等待其他的高优先级任务或者中断服务函数运行完成才能轮到你。这个绝对延

时只是相对于 vTaskDelay()这个简单的延时函数而言的。

12.2 FreeRTOS 系统时钟节拍

不管是什么系统，运行都需要有个系统时钟节拍，前面已经提到多次了，xTickCount 就是 FreeRTOS 的系统时钟节拍计数器。每个滴答定时器中断中 xTickCount 就会加一，xTickCount 的具体操作过程是在函数 xTaskIncrementTick()中进行的，此函数在文件 tasks.c 中有定义，如下：

```
BaseType_t xTaskIncrementTick( void )
{
    TCB_t * pxTCB;
    TickType_t xItemValue;
    BaseType_t xSwitchRequired = pdFALSE;

    //每个时钟节拍中断(滴答定时器中断)调用一次本函数，增加时钟节拍计数器 xTickCount 的
    //值，并且检查是否有任务需要取消阻塞。
    traceTASK_INCREMENT_TICK( xTickCount );
    if( uxSchedulerSuspended == ( UBaseType_t ) pdFALSE )                    (1)
    {
        const TickType_t xConstTickCount = xTickCount + 1;                   (2)

        //增加系统节拍计数器 xTickCount 的值，当为 0，也就是溢出的话就交换延时和溢出列
        //表指针值。
        xTickCount = xConstTickCount;
        if( xConstTickCount == ( TickType_t ) 0U )                           (3)
        {
            taskSWITCH_DELAYED_LISTS();                                      (4)
        }
        else
        {
            mtCOVERAGE_TEST_MARKER();
        }

        //判断是否有任务延时时间到了，任务都会根据唤醒时间点值按照顺序(由小到大的升
        //序排列)添加到延时列表中，这就意味这如果延时列表中第一个列表项对应的任务的
        //延时时间都没有到的话后面的任务就不用看了，肯定也没有到。
        if( xConstTickCount >= xNextTaskUnblockTime )                        (5)
        {
            for( ;; )
            {
                if( listLIST_IS_EMPTY( pxDelayedTaskList ) != pdFALSE )      (6)
                {
                    //延时列表为空,设置 xNextTaskUnblockTime 为最大值。
                    xNextTaskUnblockTime = portMAX_DELAY;                    (7)
                    break;
```

```
    }
    else
    {
        //延时列表不为空，获取延时列表的第一个列表项的值，根据判断这个值
        //判断任务延时时间是否到了， 如果到了的话就将任务移除延时列表。
        pxTCB = ( TCB_t * )\                                          (8)
                listGET_OWNER_OF_HEAD_ENTRY( pxDelayedTaskList );
        xItemValue =\                                                 (9)
                listGET_LIST_ITEM_VALUE( &( pxTCB->xStateListItem ) );

        if( xConstTickCount < xItemValue )                            (10)
        {
            //任务延时时间还没到，但是 xItemValue 保存着下一个即将解除
            //阻塞态的任务对应的解除时间点，所以需要用 xItemValue 来更新
            //变量 xNextTaskUnblockTime
            xNextTaskUnblockTime = xItemValue;                        (11)
            break;
        }
        else
        {
            mtCOVERAGE_TEST_MARKER();
        }
        //将任务从延时列表中移除
        ( void ) uxListRemove( &( pxTCB->xStateListItem ) );          (12)

        //任务是否还在等待其他事件？如信号量、队列等，如果是的话就将这些
        //任务从相应的事件列表中移除。相当于等待事件超时退出！
        if( listLIST_ITEM_CONTAINER( &( pxTCB->xEventListItem ) ) !\ (13)
                                                    = NULL )
        {
            ( void ) uxListRemove( &( pxTCB->xEventListItem ) );      (14)
        }
        else
        {
            mtCOVERAGE_TEST_MARKER();
        }

        //将任务添加到就绪列表中
        prvAddTaskToReadyList( pxTCB );                               (15)

        #if (    configUSE_PREEMPTION == 1 )
        {
            //使用抢占式内核，判断解除阻塞的任务优先级是否高于当前正在
```

```
                        //运行的任务优先级，如果是的话就需要进行一次任务切换!
                        if( pxTCB->uxPriority >= pxCurrentTCB->uxPriority )          (16)
                        {
                            xSwitchRequired = pdTRUE;
                        }
                        else
                        {
                            mtCOVERAGE_TEST_MARKER();
                        }
                    }
                    #endif /* configUSE_PREEMPTION */
                }
            }
        }

        //如果使能了时间片的话还需要处理同优先级下任务之间的调度
        #if ( ( configUSE_PREEMPTION == 1 ) && ( configUSE_TIME_SLICING == 1 ) )(17)
        {
            if( listCURRENT_LIST_LENGTH( &( \
                    pxReadyTasksLists[ pxCurrentTCB->uxPriority ] ) ) > ( UBaseType_t ) 1 )
            {
                xSwitchRequired = pdTRUE;
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
            }
        }
        #endif

        //使用时钟节拍钩子函数
        #if ( configUSE_TICK_HOOK == 1 )
        {
            if( uxPendedTicks == ( UBaseType_t ) 0U )
            {
                vApplicationTickHook();                                          (18)
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
            }
        }
        #endif /* configUSE_TICK_HOOK */
```

```
    }
    else        //任务调度器挂起                                          (19)
    {
        ++uxPendedTicks;                                                (20)
        #if ( configUSE_TICK_HOOK == 1 )
        {
            vApplicationTickHook();
        }
        #endif
    }

    #if ( configUSE_PREEMPTION == 1 )
    {
        if( xYieldPending != pdFALSE )                                  (21)
        {
            xSwitchRequired = pdTRUE;
        }
        else
        {
            mtCOVERAGE_TEST_MARKER();
        }
    }
    #endif /* configUSE_PREEMPTION */

    return xSwitchRequired;                                             (22)
}
```

(1)、判断任务调度器是否被挂起。

(2)、将时钟节拍计数器 xTickCount 加一，并将结果保存在 xConstTickCount 中，下一行程序会将 xConstTickCount 赋值给 xTickCount，相当于给 xTickCount 加一。

(3)、xConstTickCount 为 0，说明发生了溢出！

(4)、如果发生了溢出的话使用函数 taskSWITCH_DELAYED_LISTS 将延时列表指针 pxDelayedTaskList 和溢出列表指针 pxOverflowDelayedTaskList 所指向的列表进行交换，函数 taskSWITCH_DELAYED_LISTS()本质上是个宏，在文件 tasks.c 中有定义，将这两个指针所指向的列表交换以后还需要更新 xNextTaskUnblockTime 的值。

(5)、变量 xNextTaskUnblockTime 保存着下一个要解除阻塞的任务的时间点值，如果 xConstTickCount 大于 xNextTaskUnblockTime 的话就说明有任务需要解除阻塞了。

(6)、判断延时列表是否为空。

(7)、如果延时列表为空的话就将 xNextTaskUnblockTime 设置为 portMAX_DELAY。

(8)、延时列表不为空，获取延时列表第一个列表项对应的任务控制块。

(9)、获取(8)中获取到的任务控制块中的壮态列表项值。

(10)、任务控制块中的壮态列表项值保存了任务的唤醒时间点，如果这个唤醒时间点值大于当前的系统时钟(时钟节拍计数器值)，说明任务的延时时间还未到。

(11)、任务延时时间还未到，而且 xItemValue 已经保存了下一个要唤醒的任务的唤醒时间

点，所以需要用 xItemValue 来更新 xNextTaskUnblockTime。

(12)、任务延时时间到了，所以将任务先从延时列表中移除。

(13)、检查任务是否还等待某个事件，比如等待信号量、队列等。如果还在等待的话就任务从相应的事件列表中移除。因为超时时间到了！

(14)、将任务从相应的事件列表中移除。

(15)、任务延时时间到了，并且任务已经从延时列表或者事件列表中已经移除。所以这里需要将任务添加到就绪列表中。

(16)、延时时间到的任务优先级高于正在运行的任务优先级，所以需要进行任务切换了，标记 xSwitchRequired 为 pdTRUE，表示需要进行任务切换。

(17)、如果使能了时间片调度的话，还要处理跟时间片调度有关的工作，具体过程参考 9.6 小节。

(18)、如果使能了时间片钩子函数的话就执行时间片钩子函数 vApplicationTickHook()，函数的具体内容由用户自行编写。

(19)、如果调用函数 vTaskSuspendAll()挂起了任务调度器的话在每个滴答定时器中断就不不会更新 xTickCount 了。取而代之的是用 uxPendedTicks 来记录调度器挂起过程中的时钟节拍数。这样在调用函数 xTaskResumeAll()恢复任务调度器的时候就会调用 uxPendedTicks 次函数 xTaskIncrementTick()，这样 xTickCount 就会恢复，并且那些应该取消阻塞的任务都会取消阻塞。函数 xTaskResumeAll()中相应的处理代码如下：

```
BaseType_t xTaskResumeAll( void )
{
    TCB_t *pxTCB = NULL;
    BaseType_t xAlreadyYielded = pdFALSE;
    configASSERT( uxSchedulerSuspended );

    taskENTER_CRITICAL();

    /*************************************************************************/
    /*************************省略部分代码*************************************/
    /*************************************************************************/

    UBaseType_t uxPendedCounts = uxPendedTicks;
    if( uxPendedCounts > ( UBaseType_t ) 0U )
    {
        //do-while()循环体，循环次数为 uxPendedTicks
        do
        {
            if( xTaskIncrementTick() != pdFALSE )      //调用函数 xTaskIncrementTick
            {
                xYieldPending = pdTRUE;                //标记需要进行任务调度。
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
```

```
            }
            --uxPendedCounts;                           //变量减一
        } while( uxPendedCounts > ( UBaseType_t ) 0U );
        uxPendedTicks = 0;                              //循环执行完毕，uxPendedTicks 清零
    }
    else
    {
        mtCOVERAGE_TEST_MARKER();
    }

    /*----------------------------------------------------------------------*/
    /*----------------------------省略部分代码----------------------------*/
    /*----------------------------------------------------------------------*/

    taskEXIT_CRITICAL();
    return xAlreadyYielded;
}
```

(20)、uxPendedTicks 是个全局变量，在文件 tasks.c 中有定义，任务调度器挂起以后此变量用来记录时钟节拍数。

(21)、有时候调用其他的 API 函数会使用变量 xYieldPending 来标记是否需要进行上下文切换，后面具体遇到具体分析。

(22)、返回 xSwitchRequired 的值，xSwitchRequired 保存了是否进行任务切换的信息，如果为 pdTRUE 的话就需要进行任务切换，pdFALSE 的话就不需要进行任务切换。函数 xPortSysTickHandler()中调用 xTaskIncrementTick()的时候就会判断返回值，并且根据返回值决定是否进行任务切换。

第十三章 FreeRTOS 队列

在实际的应用中，常常会遇到一个任务或者中断服务需要和另外一个任务进行“沟通交流”，这个“沟通交流”的过程其实就是消息传递的过程。在没有操作系统的时候两个应用程序进行消息传递一般使用全局变量的方式，但是如果在使用操作系统的应用中用全局变量来传递消息就会涉及到“资源管理”的问题。FreeRTOS 对此提供了一个叫做“队列”的机制来完成任务与任务、任务与中断之间的消息传递。本章我们就来学习 FreeRTOS 队列，本章分为如下几部分：

13.1 队列简介
13.2 队列结构体
13.3 队列创建
13.4 向队列发送消息
13.5 队列上锁和解锁
13.6 从队列读取消息
13.7 队列操作实验

13.1 队列简介

队列是为了任务与任务、任务与中断之间的通信而准备的，可以在任务与任务、任务与中断之间传递消息，队列中可以存储有限的、大小固定的数据项目。任务与任务、任务与中断之间要交流的数据保存在队列中，叫做队列项目。队列所能保存的最大数据项目数量叫做队列的长度，创建队列的时候会指定数据项目的大小和队列的长度。由于队列用来传递消息的，所以也称为消息队列。FreeRTOS 中的信号量的也是依据队列实现的！所以有必要深入的了解 FreeRTOS 的队列。

1、数据存储

通常队列采用先进先出(FIFO)的存储缓冲机制，也就是往队列发送数据的时候(也叫入队)永远都是发送到队列的尾部，而从队列提取数据的时候(也叫出队)是从队列的头部提取的。但是也可以使用 LIFO 的存储缓冲，也就是后进先出，FreeRTOS 中的队列也提供了 LIFO 的存储缓冲机制。

数据发送到队列中会导致数据拷贝，也就是将要发送的数据拷贝到队列中，这就意味着在队列中存储的是数据的原始值，而不是原数据的引用(即只传递数据的指针)，这个也叫做值传递。学过 uC/OS 的同学应该知道，uC/OS 的消息队列采用的是引用传递，传递的是消息指针。采用引用传递的话消息内容就必须一直保持可见性，也就是消息内容必须有效，那么局部变量这种可能会随时被删掉的东西就不能用来传递消息，但是采用引用传递会节省时间啊！因为不用进行数据拷贝。

采用值传递的话虽然会导致数据拷贝，会浪费一点时间，但是一旦将消息发送到队列中原始的数据缓冲区就可以删除掉或者覆写，这样的话这些缓冲区就可以被重复的使用。FreeRTOS 中使用队列传递消息的话虽然使用的是数据拷贝，但是也可以使用引用来传递消息啊，我直接往队列中发送指向这个消息的地址指针不就可以了！这样当我要发送的消息数据太大的时候就可以直接发送消息缓冲区的地址指针，比如在网络应用环境中，网络的数据量往往都很大的，采用数据拷贝的话就不现实。

1、多任务访问

队列不是属于某个特别指定的任务的，任何任务都可以向队列中发送消息，或者从队列中提取消息。

2、出队阻塞

当任务尝试从一个队列中读取消息的时候可以指定一个阻塞时间，这个阻塞时间就是当任务从队列中读取消息无效的时候任务阻塞的时间。出队就是就从队列中读取消息，出队阻塞是针对从队列中读取消息的任务而言的。比如任务 A 用于处理串口接收到的数据，串口接收到数据以后就会放到队列 Q 中，任务 A 从队列 Q 中读取数据。但是如果此时队列 Q 是空的，说明还没有数据，任务 A 这时候来读取的话肯定是获取不到任何东西，那该怎么办呢？任务 A 现在有三种选择，一：二话不说扭头就走，二：要不我在等等吧，等一会看看，说不定一会就有数据了，三：死等，死也要等到你有数据！选哪一个就是由这个阻塞时间决定的，这个阻塞时间单位是时钟节拍数。阻塞时间为 0 的话就是不阻塞，没有数据的话就马上返回任务继续执行接下来的代码，对应第一种选择。如果阻塞时间为 0~portMAX_DELAY，当任务没有从队列中获取到消息的话就进入阻塞态，阻塞时间指定了任务进入阻塞态的时间，当阻塞时间到了以后还没有接收到数据的话就退出阻塞态，返回任务接着运行下面的代码，如果在阻塞时间内接收到了数据就立即返回，执行任务中下面的代码，这种情况对应第二种选择。当阻塞时间设置为 portMAX_DELAY 的话，任务就会一直进入阻塞态等待，直到接收到数据为止！这个就是第三

种选择。

3、入队阻塞

入队说的是向队列中发送消息，将消息加入到队列中。和出队阻塞一样，当一个任务向队列发送消息的话也可以设置阻塞时间。比如任务 B 向消息队列 Q 发送消息，但是此时队列 Q 是满的，那肯定是发送失败的。此时任务 B 就会遇到和上面任务 A 一样的问题，这两种情况的处理过程是类似的，只不过一个是向队列 Q 发送消息，一个是从队列 Q 读取消息而已。

4、队列操作过程图示

下面几幅图简单的演示了一下队列的入队和出队过程。

● **创建队列**

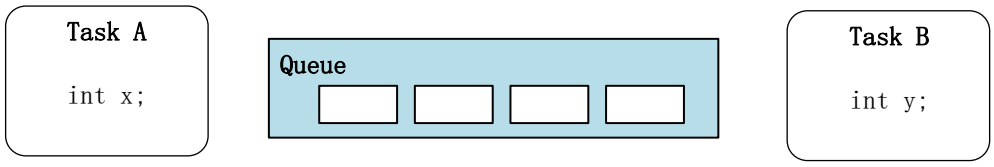


图 13.1.1 创建队列

图 13.1.1 中任务 A 要向任务 B 发送消息，这个消息是 x 变量的值。首先创建一个队列，并且指定队列的长度和每条消息的长度。这里我们创建了一个长度为 4 的队列，因为要传递的是 x 值，而 x 是个 int 类型的变量，所以每条消息的长度就是 int 类型的长度，在 STM32 中就是 4 字节，即每条消息是 4 个字节的。

● **向队列发送第一个消息**

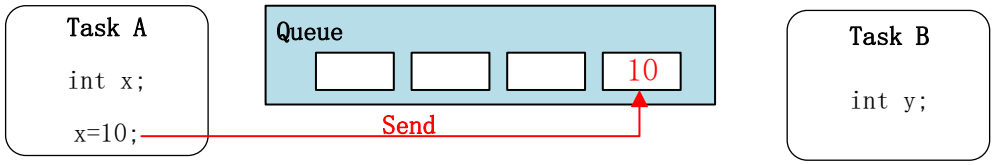


图 13.1.2 向队列发送第一个消息

图 13.1.2 中任务 A 的变量 x 值为 10，将这个值发送到消息队列中。此时队列剩余长度就是 3 了。前面说了向队列中发送消息是采用拷贝的方式，所以一旦消息发送完成变量 x 就可以再次被使用，赋其他的值。

● **向队列发送第二个消息**

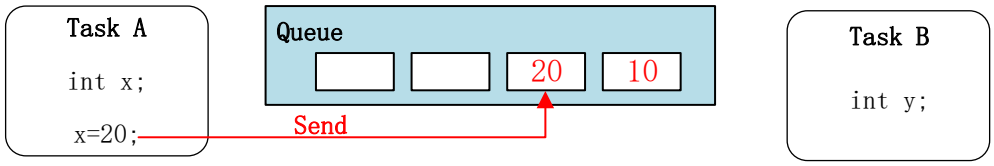


图 13.1.3 向队列发送第二个消息

图 13.1.3 中任务 A 又向队列发送了一个消息，即新的 x 的值，这里是 20。此时队列剩余长度为 2。

● **从队列中读取消息**

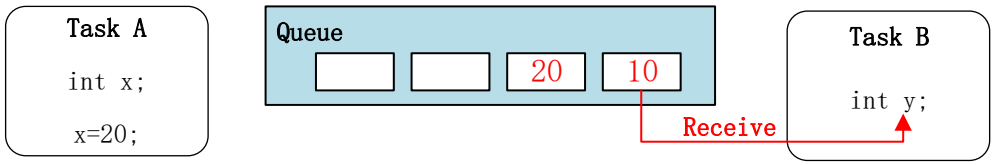


图 13.1.4 从队列中读取消息

图 13.1.4 中任务 B 从队列中读取消息，并将读取到的消息值赋值给 y，这样 y 就等于 10 了。任务 B 从队列中读取消息完成以后可以选择清除掉这个消息或者不清除。当选择清除这个消息的话其他任务或中断就不能获取这个消息了，而且队列剩余大小就会加一，变成 3。如果不清除的话其他任务或中断也可以获取这个消息，而队列剩余大小依旧是 2。

13.2 队列结构体

有一个结构体用于描述队列，叫做 Queue_t，这个结构体在文件 queue.c 中定义如下：

```
typedef struct QueueDefinition
{
    int8_t *pcHead;                 //指向队列存储区开始地址。
    int8_t *pcTail;                 //指向队列存储区最后一个字节。
    int8_t *pcWriteTo;              //指向存储区中下一个空闲区域。

    union
    {
        int8_t *pcReadFrom;         //当用作队列的时候指向最后一个出队的队列项首地址
        UBaseType_t uxRecursiveCallCount;//当用作递归互斥量的时候用来记录递归互斥量被
                                         //调用的次数。
    } u;

    List_t xTasksWaitingToSend;     //等待发送任务列表，那些因为队列满导致入队失败而进
                                    //入阻塞态的任务就会挂到此列表上。
    List_t xTasksWaitingToReceive;  //等待接收任务列表，那些因为队列空导致出队失败而进
                                    //入阻塞态的任务就会挂到此列表上。

    volatile UBaseType_t uxMessagesWaiting;  //队列中当前队列项数量，也就是消息数
    UBaseType_t uxLength;           //创建队列时指定的队列长度，也就是队列中最大允许的
                                    //队列项(消息)数量
    UBaseType_t uxItemSize;         //创建队列时指定的每个队列项(消息)最大长度，单位字节

    volatile int8_t cRxLock;        //当队列上锁以后用来统计从队列中接收到的队列项数
                                    //量，也就是出队的队列项数量，当队列没有上锁的话此字
                                    //段为 queueUNLOCKED

    volatile int8_t cTxLock;        //当队列上锁以后用来统计发送到队列中的队列项数量，
                                    //也就是入队的队列项数量，当队列没有上锁的话此字
                                    //段为 queueUNLOCKED

    #if( ( configSUPPORT_STATIC_ALLOCATION == 1 ) &&\
         ( configSUPPORT_DYNAMIC_ALLOCATION == 1 ) )
          uint8_t ucStaticallyAllocated;//如果使用静态存储的话此字段设置为 pdTURE。
    #endif
```

```
    #if ( configUSE_QUEUE_SETS == 1 )       //队列集相关宏
        struct QueueDefinition *pxQueueSetContainer;
    #endif

    #if ( configUSE_TRACE_FACILITY == 1 ) //跟踪调试相关宏
        UBaseType_t uxQueueNumber;
        uint8_t ucQueueType;
    #endif

} xQUEUE;

typedef    xQUEUE       Queue_t;
```

老版本的 FreeRTOS 中队列可能会使用 xQUEUE 这个名字，新版本 FreeRTOS 中队列的名字都使用 Queue_t。

13.3 队列创建

13.3.1 函数原型

在使用队列之前必须先创建队列，有两种创建队列的方法，一种是静态的，使用函数 xQueueCreateStatic()；另一个是动态的，使用函数 xQueueCreate()。这两个函数本质上都是宏，真正完成队列创建的函数是 xQueueGenericCreate()和 xQueueGenericCreateStatic()，这两个函数在文件 queue.c 中有定义，这四个函数的原型如下。

1、函数 xQueueCreate()

此函数本质上是一个宏，用来动态创建队列，此宏最终调用的是函数 xQueueGenericCreate()，函数原型如下：

```
QueueHandle_t xQueueCreate(UBaseType_t      uxQueueLength,
                           UBaseType_t      uxItemSize)
```

参数：

uxQueueLength：要创建的队列的队列长度，这里是队列的项目数。
uxItemSize：队列中每个项目(消息)的长度，单位为字节

返回值：

其他值：队列创捷成功以后返回的队列句柄！
NULL：队列创建失败。

2、函数 xQueueCreateStatic()

此函数也是用于创建队列的，但是使用的静态方法创建队列，队列所需要的内存由用户自行分配，此函数本质上也是一个宏，此宏最终调用的是函数 xQueueGenericCreateStatic()，函数原型如下：

```
QueueHandle_t xQueueCreateStatic(UBaseType_t      uxQueueLength,
```

```
                                UBaseType_t     uxItemSize,
                                uint8_t *       pucQueueStorageBuffer,
                                StaticQueue_t * pxQueueBuffer)
```

参数：

uxQueueLength： 要创建的队列的队列长度，这里是队列的项目数。

uxItemSize： 队列中每个项目(消息)的长度，单位为字节

pucQueueStorage: 指向队列项目的存储区，也就是消息的存储区，这个存储区需要用户自行分配。此参数必须指向一个 uint8_t 类型的数组。这个存储区要大于等于(uxQueueLength * uxItemsSize)字节。

pxQueueBuffer: 此参数指向一个 StaticQueue_t 类型的变量，用来保存队列结构体。

返回值：

其他值： 队列创捷成功以后的队列句柄！

NULL: 队列创建失败。

3、函数 xQueueGenericCreate()

函数 xQueueGenericCreate()用于动态创建队列，创建队列过程中需要的内存均通过 FreeRTOS 中的动态内存管理函数 pvPortMalloc()分配，函数原型如下：

```
QueueHandle_t xQueueGenericCreate( const UBaseType_t uxQueueLength,
                                   const UBaseType_t uxItemSize,
                                   const uint8_t     ucQueueType )
```

参数：

uxQueueLength： 要创建的队列的队列长度，这里是队列的项目数。

uxItemSize： 队列中每个项目(消息)的长度，单位为字节。

ucQueueType： 队列类型，由于 FreeRTOS 中的信号量等也是通过队列来实现的，创建信号量的函数最终也是使用此函数的，因此在创建的时候需要指定此队列的用途，也就是队列类型，一共有六种类型：

queueQUEUE_TYPE_BASE	普通的消息队列
queueQUEUE_TYPE_SET	队列集
queueQUEUE_TYPE_MUTEX	互斥信号量
queueQUEUE_TYPE_COUNTING_SEMAPHORE	计数型信号量
queueQUEUE_TYPE_BINARY_SEMAPHORE	二值信号量
queueQUEUE_TYPE_RECURSIVE_MUTEX	递归互斥信号量

函数 xQueueCreate()创建队列的时候此参数默认选择的就是 queueQUEUE_TYPE_BASE。

返回值：

其他值： 队列创捷成功以后的队列句柄！

NULL: 队列创建失败。

4、函数 xQueueGenericCreateStatic()

此函数用于动态创建队列，创建队列过程中需要的内存需要由用户自行分配好，函数原型如下：

```
QueueHandle_t xQueueGenericCreateStatic( const UBaseType_t      uxQueueLength,
                                         const UBaseType_t      uxItemSize,
                                         uint8_t *              pucQueueStorage,
                                         StaticQueue_t *        pxStaticQueue,
                                         const uint8_t          ucQueueType )
```

参数：

uxQueueLength： 要创建的队列的队列长度，这里是队列的项目数。

uxItemSize： 队列中每个项目(消息)的长度，单位为字节

pucQueueStorage: 指向队列项目的存储区，也就是消息的存储区，这个存储区需要用户自行分配。此参数必须指向一个 uint8_t 类型的数组。这个存储区要大于等于(uxQueueLength * uxItemsSize)字节。

pxStaticQueue: 此参数指向一个 StaticQueue_t 类型的变量，用来保存队列结构体。

ucQueueType： 队列类型。

返回值：

其他值： 队列创捷成功以后队列句柄！

NULL: 队列创建失败。

13.3.2 队列创建函数详解

最终完成队列创建的函数有两个，一个是静态方法的 xQueueGenericCreateStatic()，另外一个就是动态方法的 xQueueGenericCreate()。我们来详细的分析一下动态创建函数 xQueueGenericCreate()，静态方法大同小异，大家可以自行分析一下。函数 xQueueGenericCreate()在文件 queue.c 中有如下定义：

```
QueueHandle_t xQueueGenericCreate( const UBaseType_t      uxQueueLength,
                                   const UBaseType_t      uxItemSize,
                                   const uint8_t          ucQueueType )
{
    Queue_t *pxNewQueue;
    size_t xQueueSizeInBytes;
    uint8_t *pucQueueStorage;

    configASSERT( uxQueueLength > ( UBaseType_t ) 0 );

    if( uxItemSize == ( UBaseType_t ) 0 )
    {
        //队列项大小为 0，那么就不需要存储区。
        xQueueSizeInBytes = ( size_t ) 0;
    }
    else
    {
```

```
        //分配足够的存储区，确保随时随地都可以保存所有的项目(消息)，
        xQueueSizeInBytes = ( size_t ) ( uxQueueLength * uxItemSize );                (1)
    }

    pxNewQueue = ( Queue_t * ) pvPortMalloc( sizeof( Queue_t ) + xQueueSizeInBytes );  (2)

    //内存申请成功
    if( pxNewQueue != NULL )
    {
        pucQueueStorage = ( ( uint8_t * ) pxNewQueue ) + sizeof( Queue_t );           (3)

        #if( configSUPPORT_STATIC_ALLOCATION == 1 )
        {
            //队列是使用动态方法创建的，所以队列字段 ucStaticallyAllocated 标
            //记为 pdFALSE。
            pxNewQueue->ucStaticallyAllocated = pdFALSE;
        }
        #endif
        prvInitialiseNewQueue( uxQueueLength, uxItemSize, pucQueueStorage, \      (4)
                               ucQueueType, pxNewQueue );
    }
    return pxNewQueue;
}
```

(1)、队列是要存储消息的，所以必须要有消息的存储区，函数的参数 uxQueueLength 和 uxItemSize 指定了队列中最大队列项目(消息)数量和每个消息的长度，两者相乘就是消息存储区的大小。

(2)、调用函数 pvPortMalloc()给队列分配内存，注意这里申请的内存大小是队列结构体和队列中消息存储区的总大小。

(3)、计算出消息存储区的首地址，(2)中申请到的内存是队列结构体和队列中消存储区的总大小，队列结构体内存在前，紧跟在后面的就是消息存储区内存。

(4)、调用函数 prvInitialiseNewQueue()初始化队列。

可以看出函数 xQueueGenericCreate()重要的工作就是给队列分配内存，当内存分配成功以后调用函数 prvInitialiseNewQueue()来初始化队列。

13.3.3 队列初始化函数

队列初始化函数 prvInitialiseNewQueue()用于队列的初始化，此函数在文件 queue.c 中有定义，函数代码如下：

```
static void prvInitialiseNewQueue(  const UBaseType_t    uxQueueLength,    //队列长度
                                    const UBaseType_t    uxItemSize,       //队列项目长度
                                    uint8_t *            pucQueueStorage,  //队列项目存储区
                                    const uint8_t        ucQueueType,      //队列类型
                                    Queue_t *            pxNewQueue )      //队列结构体
{
```

```
    //防止编译器报错
    ( void ) ucQueueType;

    if( uxItemSize == ( UBaseType_t ) 0 )
    {
        //队列项(消息)长度为 0，说明没有队列存储区，这里将 pcHead 指向队列开始地址
        pxNewQueue->pcHead = ( int8_t * ) pxNewQueue;
    }
    else
    {
        //设置 pcHead 指向队列项存储区首地址
        pxNewQueue->pcHead = ( int8_t * ) pucQueueStorage;                    (1)
    }

    //初始化队列结构体相关成员变量
    pxNewQueue->uxLength = uxQueueLength;                                     (2)
    pxNewQueue->uxItemSize = uxItemSize;
    ( void ) xQueueGenericReset( pxNewQueue, pdTRUE );                        (3)

    #if ( configUSE_TRACE_FACILITY == 1 )       //跟踪调试相关字段初始化
    {
        pxNewQueue->ucQueueType = ucQueueType;
    }
    #endif /* configUSE_TRACE_FACILITY */

    #if( configUSE_QUEUE_SETS == 1 )            //队列集相关字段初始化
    {
        pxNewQueue->pxQueueSetContainer = NULL;
    }
    #endif /* configUSE_QUEUE_SETS */

    traceQUEUE_CREATE( pxNewQueue );
}
```

(1)、队列结构体中的成员变量 pcHead 指向队列存储区中首地址。

(2)、初始化队列结构体中的成员变量 uxQueueLength 和 uxItemSize，这两个成员变量保存队列的最大队列项目和每个队列项大小。

(3)、调用函数 xQueueGenericReset()复位队列。PS:发一句牢骚，绕来绕去的，函数调了一个又一个的。

13.3.4 队列复位函数

队列初始化函数 prvInitialiseNewQueue()中调用了函数 xQueueGenericReset()来复位队列，函数 xQueueGenericReset()代码如下：

```
BaseType_t xQueueGenericReset( QueueHandle_t xQueue, BaseType_t xNewQueue )
```

```
{
Queue_t * const pxQueue = ( Queue_t * ) xQueue;

    configASSERT( pxQueue );

    taskENTER_CRITICAL();
    {
        //初始化队列相关成员变量
        pxQueue->pcTail = pxQueue->pcHead + ( pxQueue->uxLength * pxQueue->\          (1)
                          uxItemSize );
        pxQueue->uxMessagesWaiting = ( UBaseType_t ) 0U;
        pxQueue->pcWriteTo = pxQueue->pcHead;
        pxQueue->u.pcReadFrom = pxQueue->pcHead + ( ( pxQueue->uxLength - \
                                ( UBaseType_t ) 1U ) * pxQueue->uxItemSize );
        pxQueue->cRxLock = queueUNLOCKED;
        pxQueue->cTxLock = queueUNLOCKED;

        if( xNewQueue == pdFALSE )                                                   (2)
        {
            //由于复位队列以后队列依旧是空的，所以对于那些由于出队(从队列中读取消
            //息)而阻塞的任务就依旧保持阻塞壮态。但是对于那些由于入队(向队列中发送
            //消息)而阻塞的任务就不同了，这些任务要解除阻塞壮态，从队列的相应列表中
            //移除。
            if( listLIST_IS_EMPTY( &( pxQueue->xTasksWaitingToSend ) ) == pdFALSE )
            {
                if( xTaskRemoveFromEventList( &( pxQueue->\
                        xTasksWaitingToSend ) ) != pdFALSE )
                {
                    queueYIELD_IF_USING_PREEMPTION();
                }
                else
                {
                    mtCOVERAGE_TEST_MARKER();
                }
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
            }
        }
        else
        {
            //初始化队列中的列表
```

```
            vListInitialise( &( pxQueue->xTasksWaitingToSend ) );                    (3)
            vListInitialise( &( pxQueue->xTasksWaitingToReceive ) );
        }
    }
    taskEXIT_CRITICAL();
    return pdPASS;
}
```

(1)、初始化队列中的相关成员变量。

(2)、根据参数 xNewQueue 确定要复位的队列是否是新创建的队列，如果不是的话还需要做其他的处理

(3)、初始化队列中的列表 xTasksWaitingToSend 和 xTasksWaitingToReceive。

至此，队列创建成功，比如我们创建一个有 4 个队列项，每个队列项长度为 32 个字节的队列 TestQueue，创建成功的队列如图 13.3.4.1 所示：

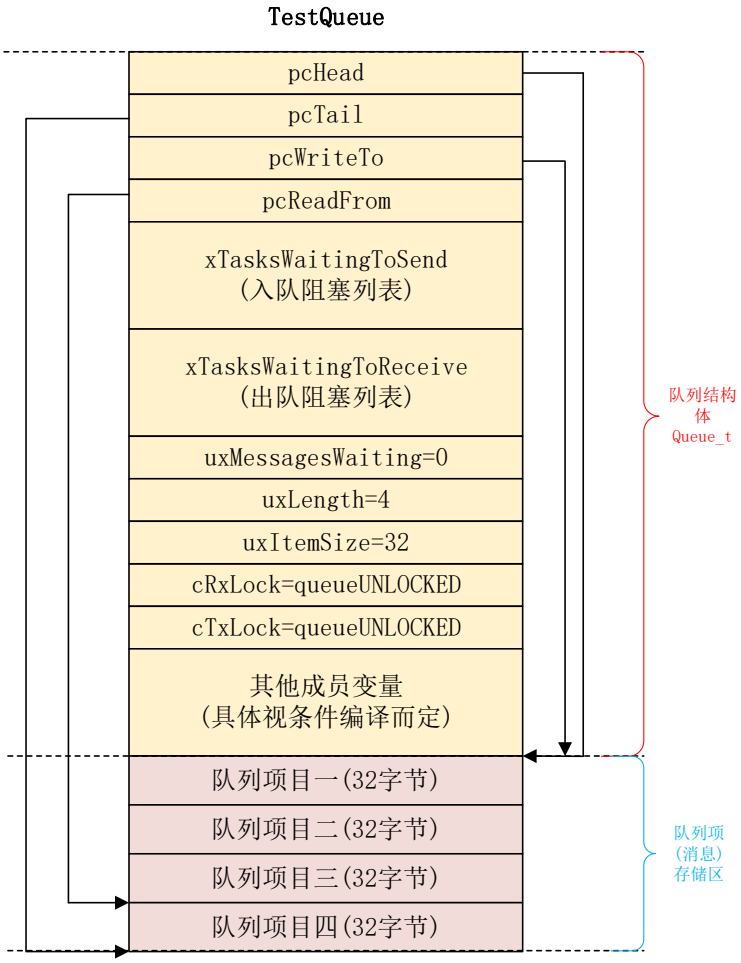


图 13.3.4.1 创建完成后的初始队列

13.4 向队列发送消息

13.4.1 函数原型

创建好队列以后就可以向队列发送消息了，FreeRTOS 提供了 8 个向队列发送消息的 API 函

数，如表 13.4.1 所示：

分类	函数	描述
任务级入队函数	xQueueSend()	发送消息到队列尾部（后向入队），这两个函数是一样的。
	xQueueSendToBack()	
	xQueueSendToFront()	发送消息到队列头（前向入队）。
	xQueueOverwrite()	发送消息到队列，带覆写功能，当队列满了以后自动覆盖掉旧的消息。
中断级入队函数	xQueueSendFromISR()	发送消息到队列尾（后向入队），这两个函数是一样的，用于中断服务函数
	xQueueSendToBackFromISR()	
	xQueueSendToFrontFromISR()	发送消息到队列头（前向入队），用于中断服务函数
	xQueueOverwriteFromISR()	发送消息到队列，带覆写功能，当队列满了以后自动覆盖掉旧的消息，用于中断服务函数。

表 13.4.1.1 入队函数

1、函数 xQueueSend()、xQueueSendToBack()和 xQueueSendToFront()

这三个函数都是用于向队列中发送消息的，这三个函数本质都是宏，其中函数 xQueueSend()和 xQueueSendToBack()是一样的，都是后向入队，即将新的消息插入到队列的后面。函数 xQueueSendToToFront()是前向入队，即将新消息插入到队列的前面。然而！这三个函数最后都是调用的同一个函数：xQueueGenericSend()。这三个函数只能用于任务函数中，不能用于中断服务函数，中断服务函数有专用的函数，它们以“FromISR”结尾，这三个函数的原型如下：

```
BaseType_t   xQueueSend(   QueueHandle_t     xQueue,
                           const void *      pvItemToQueue,
                           TickType_t        xTicksToWait);

BaseType_t xQueueSendToBack(QueueHandle_t xQueue,
                            const void*     pvItemToQueue,
                            TickType_t      xTicksToWait);

BaseType_t xQueueSendToToFront(QueueHandle_t    xQueue,
                               const void       *pvItemToQueue,
                               TickType_t       xTicksToWait);
```

参数：

xQueue： 队列句柄，指明要向哪个队列发送数据，创建队列成功以后会返回此队列的队列句柄。

pvItemToQueue： 指向要发送的消息，发送时候会将这个消息拷贝到队列中。

xTicksToWait： 阻塞时间，此参数指示当队列满的时候任务进入阻塞态等待队列空闲的最大时间。如果为 0 的话当队列满的时候就立即返回；当为 portMAX_DELAY 的话就会一直等待，直到队列有空闲的队列项，也就是死等，但是宏 INCLUDE_vTaskSuspend 必须为 1。

返回值：

pdPASS： 向队列发送消息成功！

errQUEUE_FULL： 队列已经满了，消息发送失败。

2、函数 xQueueOverwrite()

此函数也是用于向队列发送数据的，当队列满了以后会覆写掉旧的数据，不管这个旧数据有没有被其他任务或中断取走。这个函数常用于向那些长度为 1 的队列发送消息，此函数也是一个宏，最终调用的也是函数 xQueueGenericSend()，函数原型如下：

```
BaseType_t xQueueOverwrite(QueueHandle_t    xQueue,
                           const void *     pvItemToQueue);
```

参数：

xQueue： 队列句柄，指明要向哪个队列发送数据，创建队列成功以后会返回此队列的队列句柄。

pvItemToQueue： 指向要发送的消息，发送的时候会将这个消息拷贝到队列中。

返回值：

pdPASS： 向队列发送消息成功，此函数也只会返回 pdPASS！因为此函数执行过程中不在乎队列满不满，满了的话我就覆写掉旧的数据，总之肯定能成功。

3、函数 xQueueGenericSend()

此函数才是真正干活的，上面讲的所有的任务级入队函数最终都是调用的此函数，此函数也是我们后面重点要讲解的，先来看一下函数原型：

```
BaseType_t xQueueGenericSend( QueueHandle_t       xQueue,
                              const void * const  pvItemToQueue,
                              TickType_t          xTicksToWait,
                              const BaseType_t    xCopyPosition )
```

参数：

xQueue： 队列句柄，指明要向哪个队列发送数据，创建队列成功以后会返回此队列的队列句柄。

pvItemToQueue： 指向要发送的消息，发送的过程中会将这个消息拷贝到队列中。

xTicksToWait： 阻塞时间。

xCopyPosition： 入队方式，有三种入队方式：

queueSEND_TO_BACK： 后向入队

queueSEND_TO_FRONT： 前向入队

queueOVERWRITE： 覆写入队。

上面讲解的入队 API 函数就是通过此参数来决定采用哪种入队方式的。

返回值：

pdTRUE： 向队列发送消息成功！

errQUEUE_FULL： 队列已经满了，消息发送失败。

4、函数 xQueueSendFromISR()、

xQueueSendToBackFromISR()、

xQueueSendToFrontFromISR()

这三个函数也是向队列中发送消息的，这三个函数用于中断服务函数中。这三个函数本质也宏，其中函数 xQueueSendFromISR ()和 xQueueSendToBackFromISR ()是一样的，都是后向入队，即将新的消息插入到队列的后面。函数 xQueueSendToFrontFromISR ()是前向入队，即将新消息插入到队列的前面。这三个函数同样调用同一个函数 xQueueGenericSendFromISR ()。这三个函数的原型如下：

```
BaseType_t xQueueSendFromISR(QueueHandle_t    xQueue,
                             const void *     pvItemToQueue,
                             BaseType_t *     pxHigherPriorityTaskWoken);

BaseType_t xQueueSendToBackFromISR(QueueHandle_t  xQueue,
                                   const void *   pvItemToQueue,
                                   BaseType_t *   pxHigherPriorityTaskWoken);

BaseType_t xQueueSendToFrontFromISR(QueueHandle_t     xQueue,
                                    const void *      pvItemToQueue,
                                    BaseType_t *      pxHigherPriorityTaskWoken);
```

参数：

xQueue： 队列句柄，指明要向哪个队列发送数据，创建队列成功以后会返回此队列的队列句柄。

pvItemToQueue： 指向要发送的消息，发送的时候会将这个消息拷贝到队列中。

pxHigherPriorityTaskWoken： 标记退出此函数以后是否进行任务切换，这个变量的值由这三个函数来设置的，用户不用进行设置，用户只需要提供一个变量来保存这个值就行了。当此值为 pdTRUE 的时候在退出中断服务函数之前一定要进行一次任务切换。

返回值：

pdTRUE： 向队列中发送消息成功！

errQUEUE_FULL: 队列已经满了，消息发送失败。

我们注意观察，可以看出这些函数都没有设置阻塞时间值。原因很简单，这些函数都是在中断服务函数中调用的，并不是在任务中，所以也就没有阻塞这一说了！

5、函数 xQueueOverwriteFromISR()

此函数是 xQueueOverwrite()的中断级版本，用在中断服务函数中，在队列满的时候自动覆写掉旧的数据，此函数也是一个宏，实际调用的也是函数 xQueueGenericSendFromISR()，此函数原型如下：

```
BaseType_t xQueueOverwriteFromISR(QueueHandle_t    xQueue,
                                  const void *     pvItemToQueue,
```

```
                                    BaseType_t *        pxHigherPriorityTaskWoken);
```

此函数的参数和返回值同上面三个函数相同。

6、函数 xQueueGenericSendFromISR()

上面说了 4 个中断级入队函数最终都是调用的函数 xQueueGenericSendFromISR()，这是真正干活的主啊，也是我们下面会详细讲解的函数，先来看一下这个函数的原型，如下：

```
BaseType_t xQueueGenericSendFromISR(QueueHandle_t      xQueue,
                                    const void*        pvItemToQueue,
                                    BaseType_t*        pxHigherPriorityTaskWoken,
                                    BaseType_t         xCopyPosition);
```

参数：

xQueue： 队列句柄，指明要向哪个队列发送数据，创建队列成功以后会返回此队列的队列句柄。

pvItemToQueue： 指向要发送的消息，发送的过程中会将这个消息拷贝到队列中。

pxHigherPriorityTaskWoken： 标记退出此函数以后是否进行任务切换，这个变量的值由这三个函数来设置的，用户不用进行设置，用户只需要提供一个变量来保存这个值就行了。当此值为 pdTRUE 的时候在退出中断服务函数之前一定要进行一次任务切换。

xCopyPosition： 入队方式，有三种入队方式：

queueSEND_TO_BACK： 后向入队

queueSEND_TO_FRONT： 前向入队

queueOVERWRITE： 覆写入队。

返回值：

pdTRUE： 向队列发送消息成功！

errQUEUE_FULL： 队列已经满了，消息发送失败。

13.4.2 任务级通用入队函数

不管是后向入队、前向入队还是覆写入队，最终调用的都是通用入队函数 xQueueGenericSend()，这个函数在文件 queue.c 文件中由定义，缩减后的函数代码如下：

```
BaseType_t xQueueGenericSend( QueueHandle_t      xQueue,
                              const void * const pvItemToQueue,
                              TickType_t         xTicksToWait,
                              const BaseType_t   xCopyPosition )
{
    BaseType_t xEntryTimeSet = pdFALSE, xYieldRequired;
    TimeOut_t xTimeOut;
    Queue_t * const pxQueue = ( Queue_t * ) xQueue;

    for( ;; )
    {
        taskENTER_CRITICAL();           //进入临界区
        {
```

```
        //查询队列现在是否还有剩余存储空间，如果采用覆写方式入队的话那就不用在
        //乎队列是不是满的啦。
        if( ( pxQueue->uxMessagesWaiting < pxQueue->uxLength ) ||\                    (1)
            ( xCopyPosition == queueOVERWRITE ) )
        {
            traceQUEUE_SEND( pxQueue );
            xYieldRequired = prvCopyDataToQueue( pxQueue, pvItemToQueue,\            (2)
                                                 xCopyPosition );
/*****************************************************************************/
/*************************省略掉与队列集相关代码*************************/
/*****************************************************************************/
            {
                //检查是否有任务由于等待消息而进入阻塞态
                if( listLIST_IS_EMPTY( &( pxQueue->xTasksWaitingToReceive ) ) ==\(3)
                                       pdFALSE )
                {
                    if( xTaskRemoveFromEventList( &( pxQueue->\                      (4)
                      xTasksWaitingToReceive ) ) != pdFALSE )
                    {
                        //解除阻塞态的任务优先级最高，因此要进行一次任务切换
                        queueYIELD_IF_USING_PREEMPTION();                            (5)
                    }
                    else
                    {
                        mtCOVERAGE_TEST_MARKER();
                    }
                }
                else if( xYieldRequired != pdFALSE )
                {
                    queueYIELD_IF_USING_PREEMPTION();
                }
                else
                {
                    mtCOVERAGE_TEST_MARKER();
                }
            }
            taskEXIT_CRITICAL();
            return pdPASS;                                                           (6)
        }
        else
        {
            if( xTicksToWait == ( TickType_t ) 0 )                                   (7)
            {
```

```
                    //队列是满的，并且没有设置阻塞时间的话就直接返回
                    taskEXIT_CRITICAL();
                    traceQUEUE_SEND_FAILED( pxQueue );
                    return errQUEUE_FULL;                                    (8)
                }
                else if( xEntryTimeSet == pdFALSE )                          (9)
                {
                    //队列是满的并且指定了任务阻塞时间的话就初始化时间结构体
                    vTaskSetTimeOutState( &xTimeOut );
                    xEntryTimeSet = pdTRUE;
                }
                else
                {
                    //时间结构体已经初始化过了，
                    mtCOVERAGE_TEST_MARKER();
                }
            }
        }
        taskEXIT_CRITICAL();        //退出临界区

        vTaskSuspendAll();                                                   (10)
        prvLockQueue( pxQueue );                                             (11)

        //更新时间壮态，检查是否有超时产生
        if( xTaskCheckForTimeOut( &xTimeOut, &xTicksToWait ) == pdFALSE )    (12)
        {
            if( prvIsQueueFull( pxQueue ) != pdFALSE )                       (13)
            {
                traceBLOCKING_ON_QUEUE_SEND( pxQueue );
                vTaskPlaceOnEventList( &( pxQueue->xTasksWaitingToSend ), \  (14)
                                       xTicksToWait );
                prvUnlockQueue( pxQueue );                                   (15)
                if( xTaskResumeAll() == pdFALSE )                            (16)
                {
                    portYIELD_WITHIN_API();
                }
            }
            else
            {
                //重试一次
                prvUnlockQueue( pxQueue );                                   (17)
                ( void ) xTaskResumeAll();
            }
```

```
            }
            else
            {
                //超时产生
                prvUnlockQueue( pxQueue );                                    (18)
                ( void ) xTaskResumeAll();

                traceQUEUE_SEND_FAILED( pxQueue );
                return errQUEUE_FULL;                                         (19)
            }
        }
    }
```

(1)、要向队列发送数据，肯定要先检查一下队列是不是满的，如果是满的话肯定不能发送的。当队列未满或者是覆写入队的话就可以将消息入队了。

(2)、调用函数 prvCopyDataToQueue()将消息拷贝到队列中。前面说了入队分为后向入队、前向入队和覆写入队，他们的具体实现就是在函数 prvCopyDataToQueue()中完成的。如果选择后向入队 queueSEND_TO_BACK 的话就将消息拷贝到队列结构体成员 pcWriteTo 所指向的队列项，拷贝成功以后 pcWriteTo 增加 uxItemSize 个字节，指向下一个队列项目。当选择前向入队 queueSEND_TO_FRONT 或者 queueOVERWRITE 的话就将消息拷贝到 u.pcReadFrom 所指向的队列项目，同样的需要调整 u.pcReadFrom 的位置。当向队列写入一个消息以后队列中统计当前消息数量的成员 uxMessagesWaiting 就会加一，但是选择覆写入队 queueOVERWRITE 的话还会将 uxMessagesWaiting 减一，这样一减一加相当于队列当前消息数量没有变。

(3)、检查是否有任务由于请求队列消息而阻塞，阻塞的任务会挂在队列的 xTasksWaitingToReceive 列表上。

(4)、有任务由于请求消息而阻塞，因为在(2)中已将向队列中发送了一条消息了，所以调用函数 xTaskRemoveFromEventList()将阻塞的任务从列表 xTasksWaitingToReceive 上移除，并且把这个任务添加到就绪列表中，如果调度器上锁的话这些任务就会挂到列表 xPendingReadyList 上。如果取消阻塞的任务优先级比当前正在运行的任务优先级高还要标记需要进行任务切换。当函数 xTaskRemoveFromEventList()返回值为 pdTRUE 的话就需要进行任务切换。

(5)、进行任务切换。

(6)、返回 pdPASS，标记入队成功。

(7)、(2)到(6)都是非常理想的效果，即消息队列未满，入队没有任何障碍。但是队列满了以后呢？首先判断设置的阻塞时间是否为 0，如果为 0 的话就说明没有阻塞时间。

(8)、由(7)得知阻塞时间为 0，那就直接返回 errQUEUE_FULL，标记队列已满就可以了。

(9)、如果阻塞时间不为 0 并且时间结构体还没有初始化的话就初始化一次超时结构体变量，调用函数 vTaskSetTimeOutState()完成超时结构体变量 xTimeOut 的初始化。其实就是记录当前的系统时钟节拍计数器的值 xTickCount 和溢出次数 xNumOfOverflows。

(10)、任务调度器上锁，代码执行到这里说明当前的状况是队列已满了，而且设置了不为 0 的阻塞时间。那么接下来就要对任务采取相应的措施了，比如将任务加入到队列的 xTasksWaitingToSend 列表中。

(11)、调用函数 prvLockQueue()给队列上锁，其实就是将队列中的成员变量 cRxLock 和 cTxLock 设置为 queueLOCKED_UNMODIFIED。

(12)、调用函数 xTaskCheckForTimeOut()更新超时结构体变量 xTimeOut，并且检查阻塞时

间是否到了。

(13)、阻塞时间还没到，那就检查队列是否还是满的。

(14)、经过(12)和(13)得出阻塞时间没到，而且队列依旧是满的，那就调用函数 vTaskPlaceOnEventList()将任务添加到队列的 xTasksWaitingToSend 列表中和延时列表中，并且将任务从就绪列表中移除。注意！如果阻塞时间是 portMAX_DELAY 并且宏 INCLUDE_vTaskSuspend 为 1 的话，函数 vTaskPlaceOnEventList()会将任务添加到列表 xSuspendedTaskList 上。

(15)、操作完成，调用函数 prvUnlockQueue()解锁队列。

(16)、调用函数 xTaskResumeAll()恢复任务调度器

(17)、阻塞时间还没到，但是队列现在有空闲的队列项，那么就在重试一次。

(18)、相比于第(12)步，阻塞时间到了！那么任务就不用添加到那些列表中了，那就解锁队列，恢复任务调度器。

(19)、返回 errQUEUE_FULL，表示队列满了。

13.4.3 中断级通用入队函数

讲完任务级入队函数再来看一下中断级入队函数 xQueueGenericSendFromISR()，其他的中断级入队函数都是靠此函数来实现的。中断级入队函数和任务级入队函数大同小异，函数代码如下：

```
BaseType_t xQueueGenericSendFromISR( QueueHandle_t              xQueue,
                                     const void * const         pvItemToQueue,
                                     BaseType_t * const         pxHigherPriorityTaskWoken,
                                     const BaseType_t           xCopyPosition )
{
    BaseType_t xReturn;
    UBaseType_t uxSavedInterruptStatus;
    Queue_t * const pxQueue = ( Queue_t * ) xQueue;

    portASSERT_IF_INTERRUPT_PRIORITY_INVALID();

    uxSavedInterruptStatus = portSET_INTERRUPT_MASK_FROM_ISR();
    {
        if( ( pxQueue->uxMessagesWaiting < pxQueue->uxLength ) ||\                    (1)
                                        ( xCopyPosition == queueOVERWRITE ) )
        {
            const int8_t cTxLock = pxQueue->cTxLock;                                 (2)
            traceQUEUE_SEND_FROM_ISR( pxQueue );
            ( void ) prvCopyDataToQueue( pxQueue, pvItemToQueue, xCopyPosition );    (3)

            //队列上锁的时候就不能操作事件列表，队列解锁的时候会补上这些操作的。
            if( cTxLock == queueUNLOCKED )                                           (4)
            {

    /*****************************************************************************/
```

```
/*************************省略掉与队列集相关代码*************************/
/************************************************************************/

            if( listLIST_IS_EMPTY( &( pxQueue->xTasksWaitingToReceive ) ) == \  (5)
                                                    pdFALSE )
            {
                if( xTaskRemoveFromEventList( &( pxQueue->\                     (6)
                            xTasksWaitingToReceive ) ) != pdFALSE )
                {
                    //刚刚从事件列表中移除的任务对应的任务优先级更高，所以
                    标记要进行任务切换
                    if( pxHigherPriorityTaskWoken != NULL )
                    {
                        *pxHigherPriorityTaskWoken = pdTRUE;                    (7)
                    }
                    else
                    {
                        mtCOVERAGE_TEST_MARKER();
                    }
                }
                else
                {
                    mtCOVERAGE_TEST_MARKER();
                }
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
            }
        }
        else
        {
            //cTxLock 加一，这样就知道在队列上锁期间向队列中发送了数据
            pxQueue->cTxLock = ( int8_t ) ( cTxLock + 1 );                      (8)
        }
        xReturn = pdPASS;                                                       (9)
    }
    else
    {
        traceQUEUE_SEND_FROM_ISR_FAILED( pxQueue );
        xReturn = errQUEUE_FULL;                                                (10)
    }
```

```
    }
    portCLEAR_INTERRUPT_MASK_FROM_ISR( uxSavedInterruptStatus );
    return xReturn;
}
```

(1)、队列未满或者采用的覆写的入队方式，这是最理想的壮态。

(2)、读取队列的成员变量 xTxLock，用于判断队列是否上锁。

(3)、将数据拷贝到队列中。

(4)、队列上锁了，比如任务级入队函数在操作队列中的列表的时候就会对队列上锁。

(5)、判断队列列表 xTasksWaitingToReceive 是否为空，如果不为空的话说明有任务在请求消息的时候被阻塞了。

(6)、将相应的任务从列表 xTasksWaitingToReceive 上移除。跟任务级入队函数处理过程一样。

(7)、如果刚刚从列表 xTasksWaitingToReceive 中移除的任务优先级比当前任务的优先级高，那么标记 pxHigherPriorityTaskWoken 为 pdTRUE，表示要进行任务切换。如果要进行任务切换的话就需要在退出此函数以后，退出中断服务函数之前进行一次任务切换。

(8)、如果队列上锁的话那就将队列成员变量 cTxLock 加一，表示进行了一次入队操作，在队列解锁(prvUnlockQueue())的时候会对其做相应的处理。

(9)、返回 pdPASS，表示入队完成。

(10)、如果队列满的话就直接返回 errQUEUE_FULL，表示队列满。

13.5 队列上锁和解锁

在上面讲解任务级通用入队函数和中断级通用入队函数的时候都提到了队列的上锁和解锁，队列的上锁和解锁是两个 API 函数：prvLockQueue()和 prvUnlockQueue()。首先来看一下队列上锁函数 prvLockQueue()，此函数本质上就是一个宏，定义如下：

```
#define prvLockQueue( pxQueue )                                 \
    taskENTER_CRITICAL();                                       \
    {                                                           \
        if( ( pxQueue )->cRxLock == queueUNLOCKED )             \
        {                                                       \
            ( pxQueue )->cRxLock = queueLOCKED_UNMODIFIED;\
        }                                                       \
        if( ( pxQueue )->cTxLock == queueUNLOCKED )             \
        {                                                       \
            ( pxQueue )->cTxLock = queueLOCKED_UNMODIFIED;\
        }                                                       \
    }                                                           \
    taskEXIT_CRITICAL()
```

prvLockQueue()函数很简单，就是将队列中的成员变量 cRxLock 和 cTxLock 设置为 queueLOCKED_UNMODIFIED 就行了。

在来看一下队列的解锁函数 prvUnlockQueue()，函数如下：

```
static void prvUnlockQueue( Queue_t * const pxQueue )
{
    //上锁计数器(cTxLock 和 cRxLock)记录了在队列上锁期间，入队或出队的数量，当队列
```

```
//上锁以后队列项是可以加入或者移除队列的，但是相应的列表不会更新。
taskENTER_CRITICAL();
{
    //处理 cTxLock。
    int8_t cTxLock = pxQueue->cTxLock;
    while( cTxLock > queueLOCKED_UNMODIFIED )                          (1)
    {

/*****************************************************************************/
/*************************省略掉与队列集相关代码*************************/
/*****************************************************************************/

        {
            //将任务从事件列表中移除
            if( listLIST_IS_EMPTY( &( pxQueue->xTasksWaitingToReceive ) ) == \   (2)
                                                                  pdFALSE )
            {
                if( xTaskRemoveFromEventList( &( pxQueue->\                    (3)
                    xTasksWaitingToReceive ) ) != pdFALSE )
                {
                    //从列表中移除的任务优先级比当前任务的优先级高，因此要
                    //进行任务切换。
                    vTaskMissedYield();                                        (4)
                }
                else
                {
                    mtCOVERAGE_TEST_MARKER();
                }
            }
            else
            {
                break;
            }
        }
        --cTxLock;                                                             (5)
    }
    pxQueue->cTxLock = queueUNLOCKED;                                          (6)
}
taskEXIT_CRITICAL();

//处理 cRxLock。
taskENTER_CRITICAL();
{
```

```
        int8_t cRxLock = pxQueue->cRxLock;
        while( cRxLock > queueLOCKED_UNMODIFIED )                    (7)
        {
            if( listLIST_IS_EMPTY( &( pxQueue->xTasksWaitingToSend ) ) == pdFALSE )
            {
                if( xTaskRemoveFromEventList( &( pxQueue->xTasksWaitingToSend ) ) !=\
                                    pdFALSE )
                {
                    vTaskMissedYield();
                }
                else
                {
                    mtCOVERAGE_TEST_MARKER();
                }
                --cRxLock;
            }
            else
            {
                break;
            }
        }
        pxQueue->cRxLock = queueUNLOCKED;
    }
    taskEXIT_CRITICAL();
}
```

(1)、判断是否有中断向队列发送了消息,在 13.2.3 小节讲解中断级通用入队函数的时候说了，如果当队列上锁的话那么向队列发送消息成功以后会将入队计数器 cTxLock 加一。

(2)、判断列表 xTasksWaitingToReceive 是否为空，如果不为空的话就要将相应的任务从列表中移除。

(3)、将任务从列表 xTasksWaitingToReceive 中移除。

(4)、如果刚刚从列表 xTasksWaitingToReceive 中移除的任务优先级比当前任务的优先级高，那么就要标记需要进行任务切换。这里调用函数 vTaskMissedYield()来完成此任务，函数 vTaskMissedYield()只是简单的将全局变量 xYieldPending 设置为 pdTRUE。那么真正的任务切换是在哪里完成的呢？在时钟节拍处理函数 xTaskIncrementTick()中，此函数会判断 xYieldPending 的值，从而决定是否进行任务切换，具体内容可以参考 12.2 小节。

(5)、每处理完一条就将 cTxLock 减一，直到处理完所有的。

(6)、当处理完以后标记 cTxLock 为 queueUNLOCKED，也就说 cTxLock 是没有上锁的了。

(7)、处理完 cTxLock 以后接下来就要处理 xRxLock 了，处理过程和 xTxLock 很类似，大家自行分析一下。

13.6 从队列读取消息

有入队就有出队，出队就是从队列中获取队列项(消息)，FreeRTOS 中出队函数如表 13.6.1.1 所示：

分类	函数	描述
任务级出队函数	xQueueReceive()	从队列中读取队列项(消息)，并且读取完以后删除掉队列项(消息)
	xQueuePeek()	从队列中读取队列项(消息)，并且读取完以后不删除队列项(消息)
中断级出队函数	xQueueReceiveFromISR()	从队列中读取队列项(消息)，并且读取完以后删除掉队列项(消息)，用于中断服务函数中
	xQueuePeekFromISR ()	从队列中读取队列项(消息)，并且读取完以后不删除队列项(消息)，用于中断服务函数中。

表 13.6.1.1 出队函数原型

1、函数 xQueueReceive()

此函数用于在任务中从队列中读取一条(请求)消息，读取成功以后就会将队列中的这条数据删除，此函数的本质是一个宏，真正执行的函数是 xQueueGenericReceive()。此函数在读取消息的时候是采用拷贝方式的，所以用户需要提供一个数组或缓冲区来保存读取到的数据，所读取的数据长度是创建队列的时候所设定的每个队列项目的长度，函数原型如下：

```
BaseType_t xQueueReceive(QueueHandle_t    xQueue,
                         void *           pvBuffer,
                         TickType_t       xTicksToWait);
```

参数：

xQueue： 队列句柄，指明要读取哪个队列的数据，创建队列成功以后会返回此队列的队列句柄。

pvBuffer： 保存数据的缓冲区，读取队列的过程中会将读取到的数据拷贝到这个缓冲区中。

xTicksToWait： 阻塞时间，此参数指示当队列空的时候任务进入阻塞态等待队列有数据的最大时间。如果为 0 的话当队列空的时候就立即返回；当为 portMAX_DELAY 的话就会一直等待，直到队列有数据，也就是死等，但是宏 INCLUDE_vTaskSuspend 必须为 1。

返回值：

pdTRUE： 从队列中读取数据成功。

pdFALSE: 从队列中读取数据失败。

2、函数 xQueuePeek()

此函数用于从队列读取一条(请求)消息，只能用在任务中！此函数在读取成功以后不会将消息删除，此函数是一个宏，真正执行的函数是 xQueueGenericReceive()。此函数在读取消息的时候是采用拷贝方式的，所以用户需要提供一个数组或缓冲区来保存读取到的数据，所读取的数据长度是创建队列的时候所设定的每个队列项目的长度，函数原型如下：

```
BaseType_t xQueuePeek(QueueHandle_t    xQueue,
```

```
                    void *          pvBuffer,
                    TickType_t      xTicksToWait);
```

参数：

xQueue： 队列句柄，指明要读取哪个队列的数据，创建队列成功以后会返回此队列的队列句柄。

pvBuffer： 保存数据的缓冲区，读取队列的过程中会将读取到的数据拷贝到这个缓冲区中。

xTicksToWait： 阻塞时间，此参数指示当队列空的时候任务进入阻塞态等待队列有数据的最大时间。如果为 0 的话当队列空的时候就立即返回；当为 portMAX_DELAY 的话就会一直等待，直到队列有数据，也就是死等，但是宏 INCLUDE_vTaskSuspend 必须为 1。

返回值：

pdTRUE： 从队列中读取数据成功。

pdFALSE: 从队列中读取数据失败。

3、函数 xQueueGenericReceive()

不管是函数 xQueueReceive() 还是 xQueuePeek()，最终都是调用的函数 xQueueGenericReceive()，此函数是真正干事的，函数原型如下：

```
BaseType_t xQueueGenericReceive(QueueHandle_t   xQueue,
                                void*           pvBuffer,
                                TickType_t      xTicksToWait
                                BaseType_t      xJustPeek)
```

参数：

xQueue： 队列句柄，指明要读取哪个队列的数据，创建队列成功以后会返回此队列的队列句柄。

pvBuffer： 保存数据的缓冲区，读取队列的过程中会将读取到的数据拷贝到这个缓冲区中。

xTicksToWait： 阻塞时间，此参数指示当队列空的时候任务进入阻塞态等待队列有数据的最大时间。如果为 0 的话当队列空的时候就立即返回；当为 portMAX_DELAY 的话就会一直等待，直到队列有数据，也就是死等，但是宏 INCLUDE_vTaskSuspend 必须为 1。

xJustPeek： 标记当读取成功以后是否删除掉队列项，当为 pdTRUE 的时候就不用删除，也就是说你后面再调用函数 xQueueReceive()获取到的队列项是一样的。当为 pdFALSE 的时候就会删除掉这个队列项。

返回值：

pdTRUE： 从队列中读取数据成功。

pdFALSE: 从队列中读取数据失败。

4、函数 xQueueReceiveFromISR()

此函数是 xQueueReceive()的中断版本，用于在中断服务函数中从队列中读取(请求)一条消息，读取成功以后就会将队列中的这条数据删除。此函数在读取消息的时候是采用拷贝方式的，所以需要用户提供一个数组或缓冲区来保存读取到的数据，所读取的数据长度是创建队列的时候所设定的每个队列项目的长度，函数原型如下：

```
BaseType_t xQueueReceiveFromISR(QueueHandle_t    xQueue,
                                void*            pvBuffer,
                                BaseType_t *     pxTaskWoken);
```

参数：

xQueue： 队列句柄，指明要读取哪个队列的数据，创建队列成功以后会返回此队列的队列句柄。

pvBuffer： 保存数据的缓冲区，读取队列的过程中会将读取到的数据拷贝到这个缓冲区中。

pxTaskWoken： 标记退出此函数以后是否进行任务切换，这个变量的值是由函数来设置的，用户不用进行设置，用户只需要提供一个变量来保存这个值就行了。当此值为 pdTRUE 的时候在退出中断服务函数之前一定要进行一次任务切换。

返回值：

pdTRUE： 从队列中读取数据成功。

pdFALSE: 从队列中读取数据失败。

5、函数 xQueuePeekFromISR()

此函数是 xQueuePeek()的中断版本，此函数在读取成功以后不会将消息删除，此函数原型如下：

```
BaseType_t xQueuePeekFromISR(QueueHandle_t    xQueue,
                             void *           pvBuffer)
```

参数：

xQueue： 队列句柄，指明要读取哪个队列的数据，创建队列成功以后会返回此队列的队列句柄。

pvBuffer： 保存数据的缓冲区，读取队列的过程中会将读取到的数据拷贝到这个缓冲区中。

返回值：

pdTRUE： 从队列中读取数据成功。

pdFALSE: 从队列中读取数据失败。

出队函数的具体过程和入队函数类似，具体的过程就不在详细的分析了，有兴趣的，大家自行对照着源码看一下就可以了。

13.7 队列操作实验

13.7.1 实验程序设计

1、实验目的

学习使用 FreeRTOS 的队列相关 API 函数，学会如何在任务或中断中向队列发送消息或者从队列中接收消息。

2、实验设计

本实验设计三个任务：start_task、task1_task 、Keyprocess_task 这三个任务的任务功能如下：

start_task：用来创建其他 2 个任务。

task1_task ：读取按键的键值，然后将键值发送到队列 Key_Queue 中，并且检查队列的剩余容量等信息。

Keyprocess_task ：按键处理任务，读取队列 Key_Queue 中的消息，根据不同的消息值做相应的处理。

实验需要三个按键 KEY_UP、KEY2 和 KEY0，不同的按键对应不同的按键值， 任务 task1_task 会将这些值发送到队列 Key_Queue 中。

实验中创建了两个队列 Key_Queue 和 Message_Queue，队列 Key_Queue 用于传递按键值，队列 Message_Queue 用于传递串口发送过来的消息。

实验还需要两个中断，一个是串口 1 接收中断，一个是定时器 9 中断，他们的作用如下：串口 1 接收中断：接收串口发送过来的数据，并将接收到的数据发送到队列 Message_Queue 中。定时器 9 中断：定时周期设置为 500ms，在定时中断中读取队列 Message_Queue 中的消息，并将其显示在 LCD 上。

3、实验工程

FreeRTOS 实验 13-1 FreeRTOS 队列操作实验。

4、实验程序与分析

●任务设置

```
#define START_TASK_PRIO              1          //任务优先级
#define START_STK_SIZE               256        //任务堆栈大小
TaskHandle_t StartTask_Handler;                 //任务句柄
void start_task(void *pvParameters);            //任务函数

#define TASK1_TASK_PRIO              2          //任务优先级
#define TASK1_STK_SIZE               256        //任务堆栈大小
TaskHandle_t Task1Task_Handler;                 //任务句柄
void task1_task(void *pvParameters);            //任务函数

#define KEYPROCESS_TASK_PRIO         3          //任务优先级
#define KEYPROCESS_STK_SIZE          256        //任务堆栈大小
TaskHandle_t Keyprocess_Handler;                //任务句柄
void Keyprocess_task(void *pvParameters);       //任务函数
```

```
//按键消息队列的数量
#define KEYMSG_Q_NUM        1                   //按键消息队列的数量          (1)
#define MESSAGE_Q_NUM       4                   //发送数据的消息队列的数量    (2)
QueueHandle_t Key_Queue;                        //按键值消息队列句柄
QueueHandle_t Message_Queue;                    //信息队列句柄
```

(1)、队列 Key_Queue 用来传递按键值的，也就是一个 u8 变量，所以队列长度为 1 就行了。并且消息长度为 1 个字节。

(2)、队列 Message_Queue 用来传递串口接收到的数据，队列长度设置为 4，每个消息的长度为 USART_REC_LEN(在 usart.h 中有定义)。

● **其他应用函数**

在 main.c 中还有一些其他的函数，如下：

```
//用于在 LCD 上显示接收到的队列的消息
//str:  要显示的字符串(接收到的消息)
void disp_str(u8* str)
{
    LCD_Fill(5,230,110,245,WHITE);          //先清除显示区域
    LCD_ShowString(5,230,100,16,16,str);
}

//加载主界面
void freertos_load_main_ui(void)
{
    POINT_COLOR = RED;
    LCD_ShowString(10,10,200,16,16,"Apollo STM32F4/F7");
    LCD_ShowString(10,30,200,16,16,"FreeRTOS Examp 13-1");
    LCD_ShowString(10,50,200,16,16,"Message Queue");
    LCD_ShowString(10,70,220,16,16,"KEY_UP:LED1 KEY0:Refresh LCD");
    LCD_ShowString(10,90,200,16,16,"KEY1:SendMsg KEY2:BEEP");

    POINT_COLOR = BLACK;
    LCD_DrawLine(0,107,239,107);            //画线
    LCD_DrawLine(119,107,119,319);          //画线
    LCD_DrawRectangle(125,110,234,314);     //画矩形
    POINT_COLOR = RED;
    LCD_ShowString(0,130,120,16,16,"DATA_Msg Size:");
    LCD_ShowString(0,170,120,16,16,"DATA_Msg rema:");
    LCD_ShowString(0,210,100,16,16,"DATA_Msg:");
    POINT_COLOR = BLUE;
}

//查询 Message_Queue 队列中的总队列数量和剩余队列数量
void check_msg_queue(void)
{
```

```
    u8 *p;
    u8 msgq_remain_size;                //消息队列剩余大小
    u8 msgq_total_size;                 //消息队列总大小

    taskENTER_CRITICAL();               //进入临界区
    msgq_remain_size=uxQueueSpacesAvailable(Message_Queue);//得到队列剩余大小     (1)
    msgq_total_size=uxQueueMessagesWaiting(Message_Queue)+\                       (2)
     uxQueueSpacesAvailable(Message_Queue);//得到队列总大小，总大小=使用+剩余的。
    p=mymalloc(SRAMIN,20);  //申请内存
    sprintf((char*)p,"Total Size:%d",msgq_total_size); //显示 DATA_Msg 消息队列总的大小
    LCD_ShowString(10,150,100,16,16,p);
    sprintf((char*)p,"Remain Size:%d",msgq_remain_size);    //显示 DATA_Msg 剩余大小
    LCD_ShowString(10,190,100,16,16,p);
    myfree(SRAMIN,p);           //释放内存
    taskEXIT_CRITICAL();        //退出临界区
}
```

定时器 9 的中断服务函数会调用函数 disp_str()在 LCD 上显示从队列 Message_Queue 接收到的消息。函数 freertos_load_main_ui()就是在屏幕上画出实验的初始 UI 界面。函数 check_msg_queue()用于查询队列 Message_Queue 的相关信息，比如队列总大小，队列当前剩余大小。

(1)、调用函数 uxQueueSpacesAvailable()获取队列 Message_Queue 的剩余大小。

(2)、调用函数 uxQueueMessagesWaiting()获取队列当前消息数量，也就是队列的使用量，将其与函数 uxQueueSpacesAvailable()获取到的队列剩余大小相加就是队列的总大小。

● **main()函数**

```
int main(void)
{
    HAL_Init();                         //初始化 HAL 库
    Stm32_Clock_Init(360,25,2,8);       //设置时钟,180Mhz
    delay_init(180);                    //初始化延时函数
    uart_init(115200);                  //初始化串口
    LED_Init();                         //初始化 LED
    KEY_Init();                         //初始化按键
    PCF8574_Init();                     //初始化 PCF8574
    SDRAM_Init();                       //初始化 SDRAM
    LCD_Init();                         //初始化 LCD
    TIM9_Init(5000,18000-1);            //初始化定时器 9，周期 500ms
    my_mem_init(SRAMIN);                //初始化内部内存池
    freertos_load_main_ui();            //加载主 UI

    //创建开始任务
    xTaskCreate((TaskFunction_t     )start_task,            //任务函数
                (const char*        )"start_task",          //任务名称
                (uint16_t           )START_STK_SIZE,        //任务堆栈大小
```

```
                (void*          )NULL,              //传递给任务函数的参数
                (UBaseType_t    )START_TASK_PRIO,   //任务优先级
                (TaskHandle_t*  )&StartTask_Handler); //任务句柄
    vTaskStartScheduler();                          //开启任务调度
}
```

● 任务函数

```
//开始任务任务函数
void start_task(void *pvParameters)
{
    taskENTER_CRITICAL();                   //进入临界区

    //创建消息 Key_Queue
    Key_Queue=xQueueCreate(KEYMSG_Q_NUM,sizeof(u8));                    (1)
    //创建消息 Message_Queue,队列项长度是串口接收缓冲区长度
    Message_Queue=xQueueCreate(MESSAGE_Q_NUM,USART_REC_LEN);            (2)

    //创建 TASK1 任务
    xTaskCreate((TaskFunction_t     )task1_task,
                (const char*        )"task1_task",
                (uint16_t           )TASK1_STK_SIZE,
                (void*              )NULL,
                (UBaseType_t        )TASK1_TASK_PRIO,
                (TaskHandle_t*      )&Task1Task_Handler);
    //创建 TASK2 任务
    xTaskCreate((TaskFunction_t     )Keyprocess_task,
                (const char*        )"keyprocess_task",
                (uint16_t           )KEYPROCESS_STK_SIZE,
                (void*              )NULL,
                (UBaseType_t        )KEYPROCESS_TASK_PRIO,
                (TaskHandle_t*      )&Keyprocess_Handler);
    vTaskDelete(StartTask_Handler); //删除开始任务
    taskEXIT_CRITICAL();            //退出临界区
}

//task1 任务函数
void task1_task(void *pvParameters)
{
    u8 key,i=0;
    BaseType_t err;
    while(1)
    {
        key=KEY_Scan(0);            //扫描按键
        if((Key_Queue!=0)&&(key))   //消息队列 Key_Queue 创建成功,并且按键被按下
```

```
        {
            err=xQueueSend(Key_Queue,&key,10);                          (3)
            if(err==errQUEUE_FULL)    //发送按键值
            {
                printf("队列 Key_Queue 已满，数据发送失败!\r\n");
            }
        }
        i++;
        if(i%10==0) check_msg_queue();//检 Message_Queue 队列的容量          (4)
        if(i==50)
        {
            i=0;
            LED0=!LED0;
        }
        vTaskDelay(10);                                  //延时 10ms，也就是 10 个时钟节拍
    }
}

//Keyprocess_task 函数
void Keyprocess_task(void *pvParameters)
{
    u8 num,key,beepsta=1;
    while(1)
    {
        if(Key_Queue!=0)
        {
            //请求消息 Key_Queue
            if(xQueueReceive(Key_Queue,&key,portMAX_DELAY))             (5)
            {
                switch(key)                                             (6)
                {
                    case WKUP_PRES:          //KEY_UP 控制 LED1
                        LED1=!LED1;
                        break;
                    case KEY2_PRES:          //KEY2 控制蜂鸣器
                        beepsta=!beepsta;
                        PCF8574_WriteBit(BEEP_IO,beepsta);
                        break;
                    case KEY0_PRES:          //KEY0 刷新 LCD 背景
                        num++;
                        LCD_Fill(126,111,233,313,lcd_discolor[num%14]);
                        break;
                }
```

```
            }
        }
        vTaskDelay(10);          //延时 10ms，也就是 10 个时钟节拍
    }
}
```

(1)、在使用队列之前要先创建队列，调用函数 xQueueCreate()创建队列 Key_Queue，队列长度为 1，每个队列项(消息)长度为 1 个字节。

(2)、同样的创建队列 Message_Queue，队列长度为 4，每个队列项(消息)长度为 USART_REC_LEN，USART_REC_LEN 为 50。

(3)、获取到按键键值以后就调用函数 xQueueSend()发送到队列 Key_Queue 中，由于只有一个队列 Key_Queue 只有一个队列项，所以此处也可以用覆写入队函数 xQueueOverwrite()。

(4)、调用函数 check_msg_queue()检查队列信息，并将相关的信息显示在 LCD 上，如队列总大小，队列剩余大小等。

(5)、调用函数 xQueueReceive()获取队列 Key_Queue 中的消息

(6)、变量 key 保存着获取到的消息，也就是按键值，这里根据不同的按键值做不同的处理。

●中断初始化及处理过程

本实验用到了两个中断，串口 1 的接收中断和定时器 9 的定时中断，串口 1 的具体配置看基础例程中的串口实验就可以了，这里要将串口中断接收缓冲区大小改为 50，如下：

```
#define USART_REC_LEN              50        //定义最大接收字节数 50
#define EN_USART1_RX               1         //使能（1）/禁止（0）串口 1 接收
```

还要注意！由于要在中断服务函数中使用 FreeRTOS 中的 API 函数，所以一定要注意中断优先级的设置，这里设置如下：

```
void HAL_UART_MspInit(UART_HandleTypeDef *huart)
{
    //GPIO 端口设置
    GPIO_InitTypeDef GPIO_Initure;

    if(huart->Instance==USART1)//如果是串口 1，进行串口 1 MSP 初始化
    {
        __HAL_RCC_GPIOA_CLK_ENABLE();            //使能 GPIOA 时钟
        __HAL_RCC_USART1_CLK_ENABLE();           //使能 USART1 时钟

        GPIO_Initure.Pin=GPIO_PIN_9;             //PA9
        GPIO_Initure.Mode=GPIO_MODE_AF_PP;       //复用推挽输出
        GPIO_Initure.Pull=GPIO_PULLUP;           //上拉
        GPIO_Initure.Speed=GPIO_SPEED_FAST;      //高速
        GPIO_Initure.Alternate=GPIO_AF7_USART1;  //复用为 USART1
        HAL_GPIO_Init(GPIOA,&GPIO_Initure);      //初始化 PA9

        GPIO_Initure.Pin=GPIO_PIN_10;            //PA10
        HAL_GPIO_Init(GPIOA,&GPIO_Initure);      //初始化 PA10

#if EN_USART1_RX
```

```
        HAL_NVIC_EnableIRQ(USART1_IRQn);            //使能 USART1 中断通道
        HAL_NVIC_SetPriority(USART1_IRQn,7,0);      //抢占优先级 7，子优先级 0
#endif
    }
}
```

注意红色代码的串口优先级设置，这里设置抢占优先级为 7，子优先级为 0。这个优先级中可以调用 FreeRTOS 中的 API 函数。串口 1 的中断服务函数如下：

```
//串口 1 中断服务程序
void USART1_IRQHandler(void)
{
    u32 timeout=0;
    u32 maxDelay=0x1FFFF;
    BaseType_t xHigherPriorityTaskWoken;

    HAL_UART_IRQHandler(&UART1_Handler);  //调用 HAL 库中断处理公用函数

    timeout=0;
    while (HAL_UART_GetState(&UART1_Handler) != HAL_UART_STATE_READY)//等待就绪
    {
        timeout++;////超时处理
        if(timeout>maxDelay) break;
    }

    timeout=0;
    while(HAL_UART_Receive_IT(&UART1_Handler, (u8 *)aRxBuffer, RXBUFFERSIZE) !=\
                HAL_OK)//一次处理完成之后，重新开启中断并设置 RxXferCount 为 1
    {
        timeout++; //超时处理
        if(timeout>maxDelay) break;
    }

    //就向队列发送接收到的数据
    if(USART_RX_STA&0x8000)                                          (1)
    {
        //向队列中发送数据
       xQueueSendFromISR(Message_Queue,USART_RX_BUF,&xHigherPriorityTaskWoken);(2)
       USART_RX_STA=0;     //数据处理完成，USART_RX_STA 清零
       //清除数据接收缓冲区 USART_RX_BUF,用于下一次数据接收
        memset(USART_RX_BUF,0,USART_REC_LEN);                        (3)
        //如果需要的话进行一次任务切换
        portYIELD_FROM_ISR(xHigherPriorityTaskWoken);                (4)
    }
}
```

(1)、判断是否接收到数据，如果接收到数据的话就要将数据发送到队列 Message_Queue 中，

(2)、调用函数 xQueueSendFromISR()将串口接收缓冲区 USART_RX_BUF[]中接收到的数据发送到队列 Message_Queue 中。

(3)、发送完成以后要将串口接收缓冲区 USART_RX_BUF[]清零。

(4)、如果需要进行任务调度的话在退出串口中断服务函数之前调用函数 portYIELD_FROM_ISR()进行一次任务调度。

在定时 9 的中断服务函数中请求队列 Message_Queue 中的数据并将请求到的消息显示在 LCD 上，定时器 9 的定时周期设置为 500ms，定时器初始化很简单，唯一要注意的就是中断优先级的设置，如下：

```
__HAL_RCC_TIM9_CLK_ENABLE();                    //使能 TIM9 时钟
HAL_NVIC_SetPriority(TIM1_BRK_TIM9_IRQn,8,0);   //设置中断优先级，抢占优先级 8
HAL_NVIC_EnableIRQ(TIM1_BRK_TIM9_IRQn);         //开启 ITM9 中断
```

定时器 9 的中断服务函数是重点，代码如下：

```
//定时器 9 中断服务函数
void TIM1_BRK_TIM9_IRQHandler(void)
{
    HAL_TIM_IRQHandler(&TIM9_Handler);                                    (1)
}

extern QueueHandle_t Message_Queue;     //信息队列句柄
extern void disp_str(u8* str);

//回调函数，定时器中断服务函数调用
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    u8 *buffer;
    BaseType_t xTaskWokenByReceive=pdFALSE;
    BaseType_t err;

    if(htim==(&TIM3_Handler))
    {
        FreeRTOSRunTimeTicks++;
    }
    else if(htim==(&TIM9_Handler))                                        (2)
    {
        buffer=mymalloc(SRAMIN,USART_REC_LEN);                            (3)
        if(Message_Queue!=0)
        {
            memset(buffer,0,USART_REC_LEN); //清除缓冲区                   (4)
            //请求消息 Message_Queue
            err=xQueueReceiveFromISR(Message_Queue,buffer,&xTaskWokenByReceive);(5)
            if(err==pdTRUE)             //接收到消息
            {
```

```
                disp_str(buffer);          //在 LCD 上显示接收到的消息                (6)
            }
        }
        myfree(SRAMIN,buffer);             //释放内存                                 (7)

        //如果需要的话进行一次任务切换
        portYIELD_FROM_ISR(xTaskWokenByReceive);                                      (8)
    }
}
```

(1)、根据 HAL 库的设计思路，定时器 9 的中断服务函数中调用函数 HAL_TIM_IRQHandler()，此函数会调用 HAL_TIM_PeriodElapsedCallback()，周期性中断要做的事情就放到函数 HAL_TIM_PeriodElapsedCallback()中完成，比如本例程中的从队列 Message_Queue 中获取消息。

(2)、定时器 9 的中断服务函数调用的函数 HAL_TIM_PeriodElapsedCallback()。

(3)、从队列中获取消息也是采用数据拷贝的方式，所以我们要先准备一个数据缓冲区用来保存从队列中获取到的消息。这里通过动态内存管理的方式分配一个数据缓冲区，这个动态内存管理方法是 ALIENTEK 编写的，具体原理和使用方法请参考基础例程中的内存管理实验。当然了，也可以使用 FreeRTOS 提供的动态内存管理函数。直接提供一个数组也行，这个数据缓冲区的大小一定要和队列中队列项大小相同，比如本例程就是 USART_REC_LEN。

(4)、清除缓冲区。

(5)、调用函数 xQueueReceiveFromISR()从队列 Message_Queue 中获取消息。

(6)、如果获取消息成功的话就调用函数 disp_str()将获取到的消息显示在 LCD 上。

(7)、使用完成以后就释放(3)中申请到的数据缓冲区内存。

(8)、如果需要进行任务调度的话在退出定时器的中断服务函数之前调用函数 portYIELD_FROM_ISR()进行一次任务调度。

13.7.2 程序运行结果分析

编译并下载实验代码到开发板中，打开串口调试助手，LCD 默认显示如图 13.7.2.1 所示，

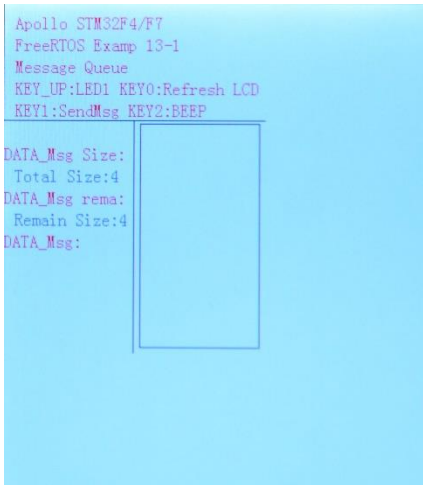


图 13.7.2.2 LCD 默认显示

通过串口调试助手给开发板发送一串字符串，比如“ALIENTEK”，由于定时器 9 会周期性的读取队列 Message_Queue 中的数据，当读取成功以后就会将相应的数据显示在 LCD 上，所以 LCD 上会显示字符串“ALENTEK”,如图 13.7.2.3 所示：

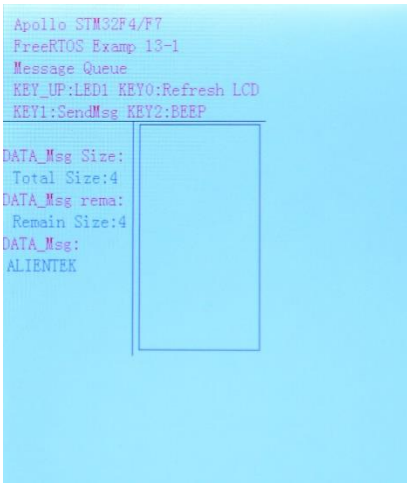


图 13.7.2.3 显示队列中的消息

通过串口向开发板发送数据的时候注意观察队列 Message_Queue 剩余大小的变化，最后按不同的按键看看有什么反应，是否和我们的代码中设置的相同。

第十四章 FreeRTOS 信号量

信号量是操作系统中重要的一部分，信号量一般用来进行资源管理和任务同步，FreeRTOS 中信号量又分为二值信号量、计数型信号量、互斥信号量和递归互斥信号量。不同的信号量其应用场景不同，但有些应用场景是可以互换着使用的， 本章我们就来学习一下 FreeRTOS 的信号量，本章分为如下几部分：

14.1 信号量简介
14.2 二值信号量
14.3 二值信号量操作实验
14.4 计数型信号量
14.5 计数型信号量操作实验
14.6 优先级翻转
14.7 优先级翻转实验
14.8 互斥信号量
14.9 互斥信号量操作实验
14.10 递归互斥信号量

14.1 信号量简介

信号量常常用于控制对共享资源的访问和任务同步。举一个很常见的例子，某个停车场有100 个停车位，这 100 个停车位大家都可以用，对于大家来说这 100 个停车位就是共享资源。假设现在这个停车场正常运行，你要把车停到这个这个停车场肯定要先看一下现在停了多少车了？还有没有停车位？当前停车数量就是一个信号量，具体的停车数量就是这个信号量值，当这个值到 100 的时候说明停车场满了。停车场满的时你可以等一会看看有没有其他的车开出停车场，当有车开出停车场的时候停车数量就会减一，也就是说信号量减一，此时你就可以把车停进去了，你把车停进去以后停车数量就会加一，也就是信号量加一。这就是一个典型的使用信号量进行共享资源管理的案例，在这个案例中使用的就是计数型信号量。再看另外一个案例：使用公共电话，我们知道一次只能一个人使用电话，这个时候公共电话就只可能有两个状态：使用或未使用，如果用电话的这两个状态作为信号量的话，那么这个就是二值信号量。

信号量用于控制共享资源访问的场景相当于一个上锁机制，代码只有获得了这个锁的钥匙才能够执行。

上面我们讲了信号量在共享资源访问中的使用，信号量的另一个重要的应用场合就是任务同步，用于任务与任务或中断与任务之间的同步。在执行中断服务函数的时候可以通过向任务发送信号量来通知任务它所期待的事件发生了，当退出中断服务函数以后在任务调度器的调度下同步的任务就会执行。在编写中断服务函数的时候我们都知道一定要快进快出，中断服务函数里面不能放太多的代码，否则的话会影响的中断的实时性。裸机编写中断服务函数的时候一般都只是在中断服务函数中打个标记，然后在其他的地方根据标记来做具体的处理过程。在使用 RTOS 系统的时候我们就可以借助信号量完成此功能，当中断发生的时候就释放信号量，中断服务函数不做具体的处理。具体的处理过程做成一个任务，这个任务会获取信号量，如果获取到信号量就说明中断发生了，那么就开始完成相应的处理，这样做的好处就是中断执行时间非常短。这个例子就是中断与任务之间使用信号量来完成同步，当然了，任务与任务之间也可以使用信号量来完成同步。

FreeRTOS 中还有一些其他特殊类型的信号量，比如互斥信号量和递归互斥信号量，这些具体遇到的时候在讲解。有关信号量的知识在 FreeRTOS 的官网上都有详细的讲解，包括二值信号量、计数型信号量、互斥信号量和递归互斥信号量，我们下面要讲解的这些涉及到理论性的知识都是翻译自 FreeRTOS 官方资料，感兴趣的可以去官网看原版的英文资料。

14.2 二值信号量

14.2.1 二值信号量简介

二值信号量通常用于互斥访问或同步，二值信号量和互斥信号量非常类似，但是还是有一些细微的差别，互斥信号量拥有优先级继承机制，二值信号量没有优先级继承。因此二值信号另更适合用于同步(任务与任务或任务与中断的同步)，而互斥信号量适合用于简单的互斥访问，有关互斥信号量的内容后面会专门讲解，本节只讲解二值信号量在同步中的应用。

和队列一样，信号量 API 函数允许设置一个阻塞时间，阻塞时间是当任务获取信号量的时候由于信号量无效从而导致任务进入阻塞态的最大时钟节拍数。如果多个任务同时阻塞在同一一个信号量上的话那么优先级最高的哪个任务优先获得信号量，这样当信号量有效的时候高优先级的任务就会解除阻塞状态。

二值信号量其实就是一个只有一个队列项的队列，这个特殊的队列要么是满的，要么是空的，这不正好就是二值的吗？ 任务和中断使用这个特殊队列不用在乎队列中存的是什么消息，

只需要知道这个队列是满的还是空的。可以利用这个机制来完成任务与中断之间的同步。

在实际应用中通常会使用一个任务来处理 MCU 的某个外设，比如网络应用中，一般最简单的方法就是使用一个任务去轮询的查询 MCU 的 ETH(网络相关外设，如 STM32 的以太网 MAC)外设是否有数据，当有数据的时候就处理这个网络数据。这样使用轮询的方式是很浪费 CPU 资源的，而且也阻止了其他任务的运行。最理想的方法就是当没有网络数据的时候网络任务就进入阻塞态，把 CPU 让给其他的任务，当有数据的时候网络任务才去执行。现在使用二值信号量就可以实现这样的功能，任务通过获取信号量来判断是否有网络数据，没有的话就进入阻塞态，而网络中断服务函数(大多数的网络外设都有中断功能，比如 STM32 的 MAC 专用 DMA 中断，通过中断可以判断是否接收到数据)通过释放信号量来通知任务以太网外设接收到了网络数据，网络任务可以去提取处理了。网络任务只是在一直的获取二值信号量，它不会释放信号量，而中断服务函数是一直在释放信号量，它不会获取信号量。在中断服务函数中发送信号量可以使用函数 xSemaphoreGiveFromISR()，也可以使用任务通知功能来替代二值信号量，而且使用任务通知的话速度更快，代码量更少，有关任务通知的内容后面会有专门的章节介绍。

使用二值信号量来完成中断与任务同步的这个机制中，任务优先级确保了外设能够得到及时的处理，这样做相当于推迟了中断处理过程。也可以使用队列来替代二值信号量，在外设事件的中断服务函数中获取相关数据，并将相关的数据通过队列发送给任务。如果队列无效的话任务就进入阻塞态，直至队列中有数据，任务接收到数据以后就开始相关的处理过程。下面几个步骤演示了二值信号量的工作过程。

1、二值信号量无效

图 14.2.1.1 请求二值信号量

在图 14.2.1.1 中任务 Task 通过函数 xSemaphoreTake()获取信号量，但是此时二值信号量无效，所以任务 Task 进入阻塞态。

2、中断释放信号量

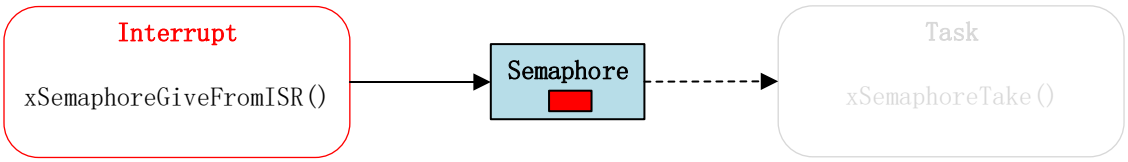


图 14.2.1.2

此时中断发生了，在中断服务函数中通过函数 xSemaphoreGiveFromISR()释放信号量，因此信号量变为有效。

3、任务获取信号量成功

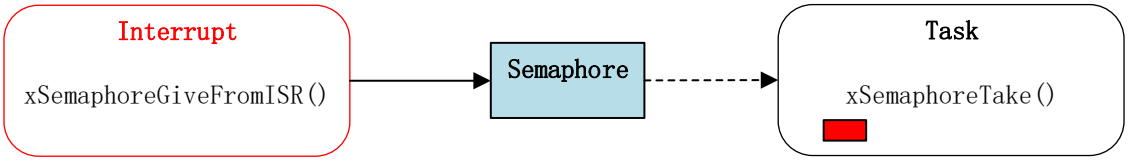


图 14.2.1.3 任务请求信号量成功

由于信号量已经有效了，所以任务 Task 获取信号量成功，任务从阻塞态解除，开始执行相关的处理过程。

4、任务再次进入阻塞态

由于任务函数一般都是一个大循环，所以在任务做完相关的处理以后就会再次调用函数 xSemaphoreTake()获取信号量。在执行完第三步以后二值信号量就已经变为无效的了，所以任务将再次进入阻塞态，和第一步一样，直至中断再次发生并且调用函数 xSemaphoreGiveFromISR()释放信号量。

14.2.2 创建二值信号量

同队列一样，要想使用二值信号量就必须先创建二值信号量，二值信号量创建函数如表 14.2.2 所示：

函数	描述
vSemaphoreCreateBinary ()	动态创建二值信号量，这个是老版本 FreeRTOS 中使用的创建二值信号量的 API 函数。
xSemaphoreCreateBinary()	动态创建二值信号量，新版 FreeRTOS 使用此函数创建二值信号量。
xSemaphoreCreateBinaryStatic()	静态创建二值信号量。

表 14.2.2 创建二值信号量

1、函数 vSemaphoreCreateBinary ()

此函数是老版本 FreeRTOS 中的创建二值信号量函数，新版本已经不再使用了，新版本的 FreeRTOS 使用 xSemaphoreCreateBinary()来替代此函数，这里还保留这个函数是为了兼容那些基于老版本 FreeRTOS 而做的应用层代码。此函数是个宏，具体创建过程是由函数 xQueueGenericCreate()来完成的，在文件 semphr.h 中有如下定义：

```
void vSemaphoreCreateBinary( SemaphoreHandle_t xSemaphore )
```

参数：

xSemaphore：保存创建成功的二值信号量句柄。

返回值：

NULL: 二值信号量创建失败。
其他值: 二值信号量创建成功。

2、函数 xSemaphoreCreateBinary()

此函数是 vSemaphoreCreateBinary()的新版本，新版本的 FreeRTOS 中统一用此函数来创建二值信号量。使用此函数创建二值信号量的话信号量所需要的 RAM 是由 FreeRTOS 的内存管理部分来动态分配的。此函数创建好的二值信号量默认是空的，也就是说刚创建好的二值信号量使用函数 xSemaphoreTake()是获取不到的，此函数也是个宏，具体创建过程是由函数 xQueueGenericCreate()来完成的，函数原型如下：

```
SemaphoreHandle_t    xSemaphoreCreateBinary( void )
```

参数：

无。

返回值：

NULL: 二值信号量创建失败。
其他值: 创建成功的二值信号量的句柄。

3、函数 xSemaphoreCreateBinaryStatic()

此函数也是创建二值信号量的，只不过使用此函数创建二值信号量的话信号量所需要的 RAM 需要由用户来分配，此函数是个宏，具体创建过程是通过函数 xQueueGenericCreateStatic()来完成的，函数原型如下：

```
SemaphoreHandle_t xSemaphoreCreateBinaryStatic( StaticSemaphore_t *pxSemaphoreBuffer )
```

参数：

pxSemaphoreBuffer：此参数指向一个 StaticSemaphore_t 类型的变量，用来保存信号量结构体。

返回值：

NULL: 二值信号量创建失败。
其他值: 创建成功的二值信号量句柄。

14.2.3 二值信号量创建过程分析

上一小节讲了三个用于二值信号量创建的函数，两个动态的创建函数和一个静态的创建函数。本节就来分析一下这两个动态的创建函数，静态创建函数和动态的类似，就不做分析了。首先来看一下老版本的二值信号量动态创建函数 vSemaphoreCreateBinary()，函数代码如下：

```
#if( configSUPPORT_DYNAMIC_ALLOCATION == 1 )
    #define vSemaphoreCreateBinary( xSemaphore )                    \
        {                                                           \
            ( xSemaphore ) = xQueueGenericCreate( ( UBaseType_t ) 1,    \       (1)
            semSEMAPHORE_QUEUE_ITEM_LENGTH,                         \
            queueQUEUE_TYPE_BINARY_SEMAPHORE );                     \
            if( ( xSemaphore ) != NULL )                            \
            {                                                       \
                ( void ) xSemaphoreGive( ( xSemaphore ) );          \       (2)
            }                                                       \
        }
#endif
```

(1)、上面说了二值信号量是在队列的基础上实现的，所以创建二值信号量就是创建队列的过程。这里使用函数 xQueueGenericCreate()创建了一个队列，队列长度为 1，队列项长度为 0，队列类型为 queueQUEUE_TYPE_BINARY_SEMAPHORE，也就是二值信号量。

(2)、当二值信号量创建成功以后立即调用函数 xSemaphoreGive()释放二值信号量，此时新创建的二值信号量有效。

在来看一下新版本的二值信号量创建函数 xSemaphoreCreateBinary()，函数代码如下：

```
#if( configSUPPORT_DYNAMIC_ALLOCATION == 1 )
    #define    xSemaphoreCreateBinary()                             \
```

```
                xQueueGenericCreate( ( UBaseType_t ) 1,                 \
                semSEMAPHORE_QUEUE_ITEM_LENGTH,                         \
                queueQUEUE_TYPE_BINARY_SEMAPHORE )                      \
#endif
```

可以看出新版本的二值信号量创建函数也是使用函数 xQueueGenericCreate()来创建一个类型为 queueQUEUE_TYPE_BINARY_SEMAPHORE、长度为 1、队列项长度为 0 的队列。这一步和老版本的二值信号量创建函数一样，唯一不同的就是新版本的函数在成功创建二值信号量以后不会立即调用函数 xSemaphoreGive()释放二值信号量。也就是说新版函数创建的二值信号量默认是无效的，而老版本是有效的。

大家注意看，创建的队列是个没有存储区的队列，前面说了使用队列是否为空来表示二值信号量，而队列是否为空可以通过队列结构体的成员变量 uxMessagesWaiting 来判断。

14.2.4 释放信号量

释放信号量的函数有两个，如表 14.2.4.1 所示：

函数	描述
xSemaphoreGive()	任务级信号量释放函数
xSemaphoreGiveFromISR()	中断级信号量释放函数

表 14.2.4.1 释放信号量

同队列一样，释放信号量也分为任务级和中断级。还有！不管是二值信号量、计数型信号量还是互斥信号量，它们都使用表 14.2.4.1 中的函数释放信号量，递归互斥信号量有专用的释放函数。

1、函数 xSemaphoreGive()

此函数用于释放二值信号量、计数型信号量或互斥信号量，此函数是一个宏，真正释放信号量的过程是由函数 xQueueGenericSend()来完成的，函数原型如下：

```
BaseType_t    xSemaphoreGive( xSemaphore )
```

参数：

xSemaphore：要释放的信号量句柄。

返回值：

pdPASS: 释放信号量成功。
errQUEUE_FULL: 释放信号量失败。

我们再来看一下函数 xSemaphoreGive()的具体内容，此函数在文件 semphr.h 中有如下定义：

```
#define xSemaphoreGive( xSemaphore )                                    \
    xQueueGenericSend( ( QueueHandle_t ) ( xSemaphore ),                \
    NULL,                                                               \
    semGIVE_BLOCK_TIME,                                                 \
    queueSEND_TO_BACK )                                                 \
```

可以看出任务级释放信号量就是向队列发送消息的过程，只是这里并没有发送具体的消息，阻塞时间为 0(宏 semGIVE_BLOCK_TIME 为 0)，入队方式采用的后向入队。具体入队过程第十三章已经做了详细的讲解，入队的时候队列结构体成员变量 uxMessagesWaiting 会加一，对于二

值信号量通过判断 uxMessagesWaiting 就可以知道信号量是否有效了，当 uxMessagesWaiting 为 1 的话说明二值信号量有效，为 0 就无效。如果队列满的话就返回错误值 errQUEUE_FULL，提示队列满，入队失败。

2、函数 xSemaphoreGiveFromISR()

此函数用于在中断中释放信号量，此函数只能用来释放二值信号量和计数型信号量，绝对不能用来在中断服务函数中释放互斥信号量！此函数是一个宏，真正执行的是函数 xQueueGiveFromISR()，此函数原型如下：

```
BaseType_t xSemaphoreGiveFromISR( SemaphoreHandle_t    xSemaphore,
                                  BaseType_t *         pxHigherPriorityTaskWoken)
```

参数：

xSemaphore： 要释放的信号量句柄。

pxHigherPriorityTaskWoken： 标记退出此函数以后是否进行任务切换，这个变量的值由这三个函数来设置的，用户不用进行设置，用户只需要提供一个变量来保存这个值就行了。当此值为 pdTRUE 的时候在退出中断服务函数之前一定要进行一次任务切换。

返回值：

pdPASS: 释放信号量成功。

errQUEUE_FULL: 释放信号量失败。

在中断中释放信号量真正使用的是函数 xQueueGiveFromISR()，此函数和中断级通用入队函数 xQueueGenericSendFromISR()极其类似！只是针对信号量做了微小的改动。函数 xSemaphoreGiveFromISR()不能用于在中断中释放互斥信号量，因为互斥信号量涉及到优先级继承的问题，而中断不属于任务，没法处理中断优先级继承。大家可以参考第十三章分析函数 xQueueGenericSendFromISR()的过程来分析 xQueueGiveFromISR()。

14.2.5 获取信号量

获取信号量也有两个函数，如表 14.2.5.1 所示：

函数	描述
xSemaphoreTake()	任务级获取信号量函数
xSemaphoreTakeFromISR()	中断级获取信号量函数

表 14.2.5.1 获取信号量

同释放信号量的 API 函数一样，不管是二值信号量、计数型信号量还是互斥信号量，它们都使用表 14.2.5.1 中的函数获取信号量

1、函数 xSemaphoreTake()

此函数用于获取二值信号量、计数型信号量或互斥信号量，此函数是一个宏，真正获取信号量的过程是由函数 xQueueGenericReceive ()来完成的，函数原型如下：

```
BaseType_t xSemaphoreTake(SemaphoreHandle_t    xSemaphore,
                          TickType_t           xBlockTime)
```

参数：

xSemaphore：要获取的信号量句柄。
xBlockTime: 阻塞时间。

返回值：

pdTRUE: 获取信号量成功。
pdFALSE: 超时，获取信号量失败。

再来看一下函数 xSemaphoreTake ()的具体内容，此函数在文件 semphr.h 中有如下定义：

```
#define xSemaphoreTake( xSemaphore, xBlockTime )        \
xQueueGenericReceive( ( QueueHandle_t ) ( xSemaphore ),  \
                      NULL,                              \
                      ( xBlockTime ),                    \
                      pdFALSE )                          \
```

获取信号量的过程其实就是读取队列的过程，只是这里并不是为了读取队列中的消息。在第十三章讲解函数 xQueueGenericReceive()的时候说过如果队列为空并且阻塞时间为 0 的话就立即返回 errQUEUE_EMPTY，表示队列满。如果队列为空并且阻塞时间不为 0 的话就将任务添加到延时列表中。如果队列不为空的话就从队列中读取数据(获取信号量不执行这一步)，数据读取完成以后还需要将队列结构体成员变量 uxMessagesWaiting 减一，然后解除某些因为入队而阻塞的任务，最后返回 pdPASS 表示出对成功。互斥信号量涉及到优先级继承，处理方式不同，后面讲解互斥信号量的时候在详细的讲解。

2、函数 xSemaphoreTakeFromISR ()

此函数用于在中断服务函数中获取信号量，此函数用于获取二值信号量和计数型信号量，绝对不能使用此函数来获取互斥信号量！此函数是一个宏，真正执行的是函数 xQueueReceiveFromISR ()，此函数原型如下：

```
BaseType_t xSemaphoreTakeFromISR(SemaphoreHandle_t    xSemaphore,
                                 BaseType_t *         pxHigherPriorityTaskWoken)
```

参数：

xSemaphore： 要获取的信号量句柄。
pxHigherPriorityTaskWoken： 标记退出此函数以后是否进行任务切换，这个变量的值由这三个函数来设置的，用户不用进行设置，用户只需要提供一个变量来保存这个值就行了。当此值为 pdTRUE 的时候在退出中断服务函数之前一定要进行一次任务切换。

返回值：

pdPASS: 获取信号量成功。
pdFALSE: 获取信号量失败。

在中断中获取信号量真正使用的是函数 xQueueReceiveFromISR ()，这个函数就是中断级出队函数！当队列不为空的时候就拷贝队列中的数据(用于信号量的时候不需要这一步)，然后将队列结构体中的成员变量 uxMessagesWaiting 减一，如果有任务因为入队而阻塞的话就解除阻塞态，当解除阻塞的任务拥有更高优先级的话就将参数 pxHigherPriorityTaskWoken 设置为 pdTRUE，最后返回 pdPASS 表示出队成功。如果队列为空的话就直接返回 pdFAIL 表示出队失败！这个函数还是很简单的。

14.3 二值信号量操作实验

14.3.1 实验程序设计

1、实验目的

二值信号量的使命就是同步，完成任务与任务或中断与任务之间的同步。大多数情况下都是中断与任务之间的同步。本节就学习一下如何使用二值信号量来完成中断与任务之间的同步。

2、实验设计

本节我们设计一个通过串口发送指定的指令来控制开发板上的 LED1 和 BEEP 开关的实验，指令如下(不区分大小写)：

LED1ON：打开 LED1。

LED1OFF：关闭 LED1。

BEEPON：打开蜂鸣器。

BEEPOFF：关闭蜂鸣器。

这些指令通过串口发送给开发板，指令是不分大小写的！开发板使用中断接收，当接收到数据以后就释放二值信号量。任务 DataProcess_task()用于处理这些指令，任务会一直尝试获取二值信号量，当获取到信号量就会从串口接收缓冲区中提取这些指令，然后根据指令控制相应的外设。

本实验设计三个任务：start_task、task1_task 、DataProcess_task 这三个任务的任务功能如下：

start_task：用来创建其他 2 个任务。

task1_task ：控制 LED0 闪烁，提示系统正在运行。

DataProcess_task ：指令处理任务，根据接收到的指令来控制不同的外设。

实验中还创建了一个二值信号量 BinarySemaphore 用于完成串口中断和任务 DataProcess_task 之间的同步。

3、实验工程

FreeRTOS 实验 14-1 FreeRTOS 二值信号量操作实验。

4、实验程序与分析

●任务设置

```
#define START_TASK_PRIO          1        //任务优先级
#define START_STK_SIZE           256      //任务堆栈大小
TaskHandle_t StartTask_Handler;           //任务句柄
void start_task(void *pvParameters);      //任务函数

#define TASK1_TASK_PRIO          2        //任务优先级
#define TASK1_STK_SIZE           256      //任务堆栈大小
TaskHandle_t Task1Task_Handler;           //任务句柄
void task1_task(void *pvParameters);      //任务函数

#define DATAPROCESS_TASK_PRIO 3           //任务优先级
#define DATAPROCESS_STK_SIZE   256        //任务堆栈大小
```

```
TaskHandle_t DataProcess_Handler;              //任务句柄
void DataProcess_task(void *pvParameters);     //任务函数

//二值信号量句柄
SemaphoreHandle_t BinarySemaphore;//二值信号量句柄

//用于命令解析用的命令值
#define LED1ON              1
#define LED1OFF             2
#define BEEPON              3
#define BEEPOFF             4
#define COMMANDERR          0XFF
```

- **其他应用函数**

```
//将字符串中的小写字母转换为大写
//str:要转换的字符串
//len：字符串长度
void LowerToCap(u8 *str,u8 len)
{
    u8 i;
    for(i=0;i<len;i++)
    {
        if((96<str[i])&&(str[i]<123))       //小写字母
        str[i]=str[i]-32;                   //转换为大写
    }
}

//命令处理函数，将字符串命令转换成命令值
//str：命令
//返回值: 0XFF，命令错误；其他值，命令值
u8 CommandProcess(u8 *str)
{
    u8 CommandValue=COMMANDERR;
    if(strcmp((char*)str,"LED1ON")==0) CommandValue=LED1ON;
    else if(strcmp((char*)str,"LED1OFF")==0) CommandValue=LED1OFF;
    else if(strcmp((char*)str,"BEEPON")==0) CommandValue=BEEPON;
    else if(strcmp((char*)str,"BEEPOFF")==0) CommandValue=BEEPOFF;
    return CommandValue;
}
```

函数 LowerToCap()用于将串口发送过来的命令中的小写字母统一转换成大写字母，这样就可以在发送命令的时候不用区分大小写，因为开发板会统一转换成大写。函数 CommandProcess()用于将接收到的命令字符串转换成命令值，比如命令“LED1ON”转换成命令值就是 0(宏 LED1ON 为 0)。

- **main()函数**

```
int main(void)
{
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(360,25,2,8);   //设置时钟,180Mhz
    delay_init(180);                //初始化延时函数
    uart_init(115200);              //初始化串口
    LED_Init();                     //初始化 LED
    KEY_Init();                     //初始化按键
    PCF8574_Init();                 //初始化 PCF8574
    SDRAM_Init();                   //初始化 SDRAM
    LCD_Init();                     //初始化 LCD
    my_mem_init(SRAMIN);            //初始化内部内存池

    POINT_COLOR=RED;
    LCD_ShowString(10,10,200,16,16,"Apollo STM32F4/F7");
    LCD_ShowString(10,30,200,16,16,"FreeRTOS Examp 14-1");
    LCD_ShowString(10,50,200,16,16,"Binary Semap");
    LCD_ShowString(10,70,200,16,16,"Command data:");

    //创建开始任务
    xTaskCreate((TaskFunction_t     )start_task,            //任务函数
                (const char*        )"start_task",          //任务名称
                (uint16_t           )START_STK_SIZE,        //任务堆栈大小
                (void*              )NULL,                  //传递给任务函数的参数
                (UBaseType_t        )START_TASK_PRIO,       //任务优先级
                (TaskHandle_t*      )&StartTask_Handler);   //任务句柄
    vTaskStartScheduler();          //开启任务调度
}
```

● **任务函数**

```
//开始任务任务函数
void start_task(void *pvParameters)
{
    taskENTER_CRITICAL();           //进入临界区

    //创建二值信号量
    BinarySemaphore=xSemaphoreCreateBinary();
    //创建 TASK1 任务
    xTaskCreate((TaskFunction_t     )task1_task,
                (const char*        )"task1_task",
                (uint16_t           )TASK1_STK_SIZE,
                (void*              )NULL,
                (UBaseType_t        )TASK1_TASK_PRIO,
                (TaskHandle_t*      )&Task1Task_Handler);
```

```
    //创建 TASK2 任务
    xTaskCreate((TaskFunction_t     )DataProcess_task,
                (const char*        )"keyprocess_task",
                (uint16_t           )DATAPROCESS_STK_SIZE,
                (void*              )NULL,
                (UBaseType_t        )DATAPROCESS_TASK_PRIO,
                (TaskHandle_t*      )&DataProcess_Handler);
    vTaskDelete(StartTask_Handler); //删除开始任务
    taskEXIT_CRITICAL();                //退出临界区
}

//task1 任务函数
void task1_task(void *pvParameters)
{
    while(1)
    {
        LED0=!LED0;
        vTaskDelay(500);        //延时 500ms，也就是 500 个时钟节拍
    }
}

//DataProcess_task 函数
void DataProcess_task(void *pvParameters)
{
    u8 len=0;
    u8 CommandValue=COMMANDERR;
    BaseType_t err=pdFALSE;

    u8 *CommandStr;
    POINT_COLOR=BLUE;
    while(1)
    {
        if(BinarySemaphore!=NULL)
        {
            err=xSemaphoreTake(BinarySemaphore,portMAX_DELAY);//获取信号量      (1)
            if(err==pdTRUE)                                   //获取信号量成功
            {
                len=USART_RX_STA&0x3fff;                      //得到此次接收到的数据长度
                CommandStr=mymalloc(SRAMIN,len+1);    //申请内存
                sprintf((char*)CommandStr,"%s",USART_RX_BUF);
                CommandStr[len]='\0';                         //加上字符串结尾符号
                LowerToCap(CommandStr,len);                   //将字符串转换为大写       (2)
                CommandValue=CommandProcess(CommandStr);      //命令解析          (3)
```

```
                if(CommandValue!=COMMANDERR)                    //接收到正确的命令
                {
                    LCD_Fill(10,90,210,110,WHITE);              //清除显示区域
                    LCD_ShowString(10,90,200,16,16,CommandStr);//在 LCD 上显示命令
                    printf("命令为:%s\r\n",CommandStr);
                    switch(CommandValue)                        //处理命令          (4)
                    {
                        case LED1ON:
                            LED1=0;
                            break;
                        case LED1OFF:
                            LED1=1;
                            break;
                        case BEEPON:
                            PCF8574_WriteBit(BEEP_IO,0);
                            break;
                        case BEEPOFF:
                            PCF8574_WriteBit(BEEP_IO,1);
                            break;
                    }
                }
                else
                {
                    printf("无效的命令，请重新输入!!\r\n");
                }
                USART_RX_STA=0;
                memset(USART_RX_BUF,0,USART_REC_LEN);//串口接收缓冲区清零
                myfree(SRAMIN,CommandStr);                      //释放内存
            }
        }
        else if(err==pdFALSE)
        {
            vTaskDelay(10);         //延时 10ms，也就是 10 个时钟节拍
        }
    }
}
```

(1)、使用函数 xSemaphoreTake()获取二值信号量 BinarySemaphore，延时时间为 portMAX_DELAY。

(2)、调用函数 LowerToCap()将命令字符串中的小写字母转换成大写的。

(3)、调用函数 CommandProcess()处理命令字符串，其实就是将命令字符串转换为命令值。

(4)、根据不同的命令值执行不同的操作，如开关 LED1，开关 BEEP。

●中断初始化及处理过程

本实验中串口 1 是通过中断方式来接收数据的，所以需要初始化串口 1，串口的初始化很

简单，前面已经讲了很多次了。不过要注意串口 1 的中断优先级！因为我们要在串口 1 的中断服务函数中使用 FeeRTOS 的 API 函数，本实验设置串口 1 的抢占优先级为 7，子优先级为 0，如下：

```
HAL_NVIC_EnableIRQ(USART1_IRQn);        //使能 USART1 中断通道
HAL_NVIC_SetPriority(USART1_IRQn,7,0);  //抢占优先级 7，子优先级 0
```

串口 1 的中断服务函数如下：

```
//串口 1 中断服务程序
void USART1_IRQHandler(void)
{
    u32 timeout=0;
    u32 maxDelay=0x1FFFF;
    BaseType_t xHigherPriorityTaskWoken;

    HAL_UART_IRQHandler(&UART1_Handler);  //调用 HAL 库中断处理公用函数

    timeout=0;
    while (HAL_UART_GetState(&UART1_Handler) != HAL_UART_STATE_READY)//等待就绪
    {
     timeout++;////超时处理
     if(timeout>maxDelay) break;

    }

    timeout=0;
    while(HAL_UART_Receive_IT(&UART1_Handler, (u8 *)aRxBuffer, RXBUFFERSIZE) !=\
          HAL_OK)//一次处理完成之后，重新开启中断并设置 RxXferCount 为 1
    {
     timeout++; //超时处理
     if(timeout>maxDelay) break;
    }

    //接收到数据，并且二值信号量有效
    if((USART_RX_STA&0x8000)&&(BinarySemaphore!=NULL))
    {
        //释放二值信号量
        xSemaphoreGiveFromISR(BinarySemaphore,&xHigherPriorityTaskWoken);        (1)
        portYIELD_FROM_ISR(xHigherPriorityTaskWoken);//如果需要的话进行一次任务切换
    }
}
```

(1)、当串口接收到数据以后就调用函数 xSemaphoreGiveFromISR()释放信号量 BinarySemaphore。

14.3.2 程序运行结果分析

编译并下载实验代码到开发板中，打开串口调试助手，通过串口调试助手发送命令，比如命令“led1ON”，开发板接收到命令以后就会将命令中的小写字母转换为大写，并且显示在 LCD 上，如图 14.3.2.1 所示：

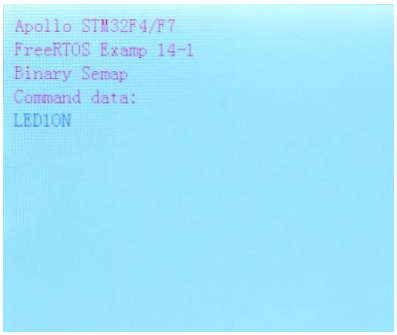


图 14.3.2.1 控制命令字符串

当命令正确的时候 LED1 就会亮，同时开发板向串口调试助手发送经过大小写转换后的命令字符串，如图 14.3.2.2 所示：

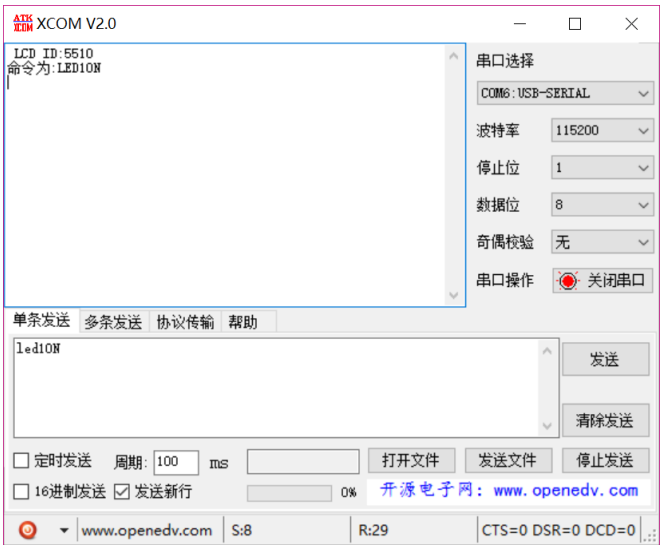


图 14.3.2.2 串口调试助手

当命令错误的时候开发板就会向串口调试助手发送命令错误的提示信息，比如我们发送“led1_off”这个命令,串口调试助手显示如图 14.3.2.3 所示：

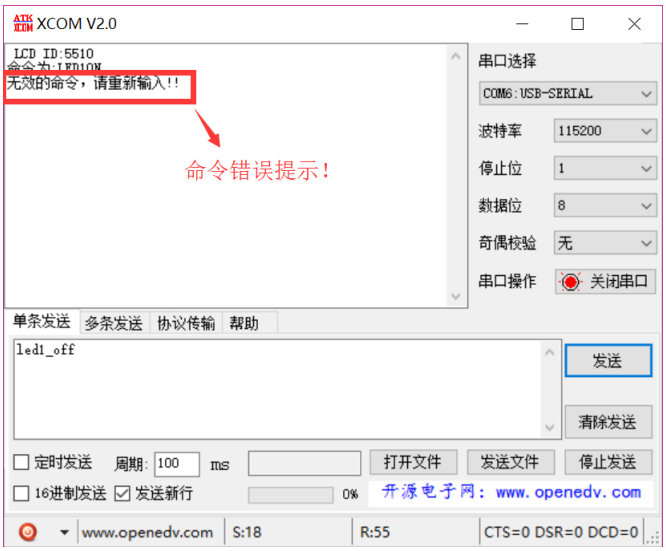


图 14.3.2.3 命令错误提示

14.4 计数型信号量

14.4.1 计数型信号量简介

有些资料中也将计数型信号量叫做数值信号量，二值信号量相当于长度为 1 的队列，那么计数型信号量就是长度大于 1 的队列。同二值信号量一样，用户不需要关心队列中存储了什么数据，只需要关心队列是否为空即可。计数型信号量通常用于如下两个场合：

1、事件计数

在这个场合中，每次事件发生的时候就在事件处理函数中释放信号量(增加信号量的计数值)，其他任务会获取信号量(信号量计数值减一，信号量值就是队列结构体成员变量 uxMessagesWaiting)来处理事件。在这种场合中创建的计数型信号量初始计数值为 0。

2、资源管理

在这个场合中，信号量值代表当前资源的可用数量，比如停车场当前剩余的停车位数量。一个任务要想获得资源的使用权，首先必须获取信号量，信号量获取成功以后信号量值就会减一。当信号量值为 0 的时候说明没有资源了。当一个任务使用完资源以后一定要释放信号量，释放信号量以后信号量值会加一。在这个场合中创建的计数型信号量初始值应该是资源的数量，比如停车场一共有 100 个停车位，那么创建信号量的时候信号量值就应该初始化为 100。

14.4.2 创建计数型信号量

FreeRTOS 提供了两个计数型信号量创建函数，如表 14.4.2.1 所示：

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

表 14.4.2.1 计数型信号量创建函数

1、函数 xSemaphoreCreateCounting()

此函数用于创建一个计数型信号量，所需要的内存通过动态内存管理方法分配。此函数本

质是一个宏，真正完成信号量创建的是函数 xQueueCreateCountingSemaphore()，此函数原型如下：

```
SemaphoreHandle_t xSemaphoreCreateCounting(UBaseType_t      uxMaxCount,
                                           UBaseType_t      uxInitialCount )
```

参数：

uxMaxCount： 计数信号量最大计数值，当信号量值等于此值的时候释放信号量就会失败。
uxInitialCount： 计数信号量初始值。

返回值：

NULL： 计数型信号量创建失败。
其他值： 计数型信号量创建成功，返回计数型信号量句柄。

2、函数 xSemaphoreCreateCountingStatic()

此函数也是用来创建计数型信号量的，使用此函数创建计数型信号量的时候所需要的内存需要由用户分配。此函数也是一个宏，真正执行的是函数 xQueueCreateCountingSemaphoreStatic()，函数原型如下：

```
SemaphoreHandle_t xSemaphoreCreateCountingStatic( UBaseType_t          uxMaxCount,
                                                  UBaseType_t          uxInitialCount,
                                                  StaticSemaphore_t *  pxSemaphoreBuffer )
```

参数：

uxMaxCount： 计数信号量最大计数值，当信号量值等于此值的时候释放信号量就会失败。
uxInitialCount： 计数信号量初始值。
pxSemaphoreBuffer： 指向一个 StaticSemaphore_t 类型的变量，用来保存信号量结构体。

返回值：

NULL： 计数型信号量创建失败。
其他值： 计数型号量创建成功，返回计数型信号量句柄。

14.4.3 计数型信号量创建过程分析

这里只分析动态创建计数型信号量函数 xSemaphoreCreateCounting()，此函数是个宏，定义如下：

```
#if( configSUPPORT_DYNAMIC_ALLOCATION == 1 )
    #define xSemaphoreCreateCounting( uxMaxCount, uxInitialCount )                \
            xQueueCreateCountingSemaphore( ( uxMaxCount ), ( uxInitialCount ) )   \
#endif
```

可以看出，真正干事的是函数 xQueueCreateCountingSemaphore()，此函数在文件 queue.c 中有如下定义：

```
QueueHandle_t xQueueCreateCountingSemaphore( const UBaseType_t    uxMaxCount,
                                             const UBaseType_t    uxInitialCount )
{
```

```
    QueueHandle_t xHandle;

    configASSERT( uxMaxCount != 0 );
    configASSERT( uxInitialCount <= uxMaxCount );

    xHandle = xQueueGenericCreate( uxMaxCount,\                                    (1)
                                   queueSEMAPHORE_QUEUE_ITEM_LENGTH, \
                                   queueQUEUE_TYPE_COUNTING_SEMAPHORE );

    if( xHandle != NULL )
    {
        ( ( Queue_t * ) xHandle )->uxMessagesWaiting = uxInitialCount;             (2)
        traceCREATE_COUNTING_SEMAPHORE();
    }
    else
    {
        traceCREATE_COUNTING_SEMAPHORE_FAILED();
    }

    return xHandle;
}
```

(1)、计数型信号量也是在队列的基础上实现的，所以需要调用函数 xQueueGenericCreate()创建一个队列，队列长度为 uxMaxCount，对列项长度为 queueSEMAPHORE_QUEUE_ITEM_LENGTH(此宏为 0)，队列的类型为 queueQUEUE_TYPE_COUNTING_SEMAPHORE，表示是个计数型信号量。

(2)、队列结构体的成员变量 uxMessagesWaiting 用于计数型信号量的计数，根据计数型信号量的初始值来设置 uxMessagesWaiting。

14.4.4 释放和获取计数信号量

计数型信号量的释放和获取与二值信号量相同，具体请参考 14.2.3 和 14.2.4 小节。

14.5 计数型信号量操作实验

14.5.1 实验程序设计

1、实验目的

计数型信号量一般用于事件计数和资源管理，计数型信号量在这个场景中的使用方法基本一样，本实验就来学习一下计数型信号量在事件计数中的使用方法。

2、实验设计

本实验中用 KEY_UP 按键来模拟事件，当 KEY_UP 按下以后就表示事件发生，当检测到 KEY_UP 按下以后就释放计数型信号量，按键的检测和信号量的释放做成一个任务。另外一个任务获取信号量，当信号量获取成功以后就刷新 LCD 上指定区域的背景颜色，并且显示计数型信号量的值。

本实验设计三个任务：start_task、SemapGive_task 、SemapTake_task 这三个任务的任务功能如下：

start_task：用来创建其他 2 个任务。

SemapGive_task ：获取按键状态，当 KEY_UP 键按下去以后就释放信号量 CountSemaphore，此任务还用来控制 LED0 的亮灭来提示程序正在运行中。

SemapTake_task ： 获取信号量 CountSemaphore，当获取信号量成功以后就刷新 LCD 指定区域的背景色。

实验中创建了一个计数型信号量 CountSemaphore，此信号量用于记录 KEY_UP 按下的次数。硬件部分需要用到 KEY_UP 按键，用于模拟事件发生。

3、实验工程

FreeRTOS 实验 14-2 FreeRTOS 计数型信号量操作实验。

4、实验程序与分析

●任务设置

```
#define START_TASK_PRIO             1          //任务优先级
#define START_STK_SIZE              256        //任务堆栈大小
TaskHandle_t StartTask_Handler;                //任务句柄
void start_task(void *pvParameters);           //任务函数

#define SEMAPGIVE_TASK_PRIO         2          //任务优先级
#define SEMAPGIVE_STK_SIZE          256        //任务堆栈大小
TaskHandle_t SemapGiveTask_Handler;            //任务句柄
void SemapGive_task(void *pvParameters);       //任务函数

#define SEMAPTAKE_TASK_PRIO         3          //任务优先级
#define SEMAPTAKE_STK_SIZE          256        //任务堆栈大小
TaskHandle_t SemapTakeTask_Handler;            //任务句柄
void SemapTake_task(void *pvParameters);       //任务函数

//计数型信号量句柄
SemaphoreHandle_t CountSemaphore;//计数型信号量

//LCD 刷屏时使用的颜色
int lcd_discolor[14]={  WHITE,      BLACK,      BLUE,       BRED,
                        GRED,       GBLUE,      RED,        MAGENTA,
                        GREEN,      CYAN,       YELLOW,     BROWN,
                        BRRED,      GRAY };
```

● main()函数

```
int main(void)
{
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(360,25,2,8);   //设置时钟,180Mhz
```

```
    delay_init(180);                    //初始化延时函数
    uart_init(115200);                  //初始化串口
    LED_Init();                         //初始化 LED
    KEY_Init();                         //初始化按键
    PCF8574_Init();                     //初始化 PCF8574
    SDRAM_Init();                       //初始化 SDRAM
    LCD_Init();                         //初始化 LCD
    my_mem_init(SRAMIN);                //初始化内部内存池

    POINT_COLOR = RED;
    LCD_ShowString(30,10,200,16,16,"Apollo STM32F4/F7");
    LCD_ShowString(30,30,200,16,16,"FreeRTOS Examp 14-2");
    LCD_ShowString(30,50,200,16,16,"Count Semaphore");
    LCD_ShowString(30,70,200,16,16,"ATOM@ALIENTEK");
    LCD_ShowString(30,90,200,16,16,"2016/11/7");

    POINT_COLOR = BLACK;
    LCD_DrawRectangle(5,110,234,314);
    LCD_DrawLine(5,130,234,130);
    POINT_COLOR = RED;
    LCD_ShowString(30,111,200,16,16,"COUNT_SEM Value: 0");
    POINT_COLOR = BLUE;

    //创建开始任务
    xTaskCreate((TaskFunction_t     )start_task,            //任务函数
                (const char*        )"start_task",          //任务名称
                (uint16_t           )START_STK_SIZE,        //任务堆栈大小
                (void*              )NULL,                  //传递给任务函数的参数
                (UBaseType_t        )START_TASK_PRIO,       //任务优先级
                (TaskHandle_t*      )&StartTask_Handler);   //任务句柄
    vTaskStartScheduler();                                  //开启任务调度
}
```

● **任务函数**

```
//开始任务任务函数
void start_task(void *pvParameters)
{
    taskENTER_CRITICAL();               //进入临界区

    //创建计数型信号量
    CountSemaphore=xSemaphoreCreateCounting(255,0);                    (1)
    //创建释放信号量任务
    xTaskCreate((TaskFunction_t     )SemapGive_task,
                (const char*        )"semapgive_task",
```

```
                (uint16_t           )SEMAPGIVE_STK_SIZE,
                (void*              )NULL,
                (UBaseType_t        )SEMAPGIVE_TASK_PRIO,
                (TaskHandle_t*      )&SemapGiveTask_Handler);
    //创建获取信号量任务
    xTaskCreate((TaskFunction_t     )SemapTake_task,
                (const char*        )"semaptake_task",
                (uint16_t           )SEMAPTAKE_STK_SIZE,
                (void*              )NULL,
                (UBaseType_t        )SEMAPTAKE_TASK_PRIO,
                (TaskHandle_t*      )&SemapTakeTask_Handler);
    vTaskDelete(StartTask_Handler);         //删除开始任务
    taskEXIT_CRITICAL();                    //退出临界区
}

//释放计数型信号量任务函数
void SemapGive_task(void *pvParameters)
{
    u8 key,i=0;
    u8 semavalue;
    BaseType_t err;
    while(1)
    {
        key=KEY_Scan(0);                    //扫描按键
        if(CountSemaphore!=NULL)            //计数型信号量创建成功
        {
            switch(key)
            {
                case WKUP_PRES:
                    err=xSemaphoreGive(CountSemaphore);//释放计数型信号量   (2)
                    if(err==pdFALSE)
                    {
                        printf("信号量释放失败!!!\r\n");
                    }
                    //获取计数型信号量值
                    semavalue=uxSemaphoreGetCount(CountSemaphore);          (3)
                    LCD_ShowxNum(155,111,semavalue,3,16,0);//显示信号量值   (4)
                    break;
            }
        }
        i++;
        if(i==50)
        {
```

```
            i=0;
            LED0=!LED0;
        }
        vTaskDelay(10);        //延时 10ms，也就是 10 个时钟节拍
    }
}
//获取计数型信号量任务函数
void SemapTake_task(void *pvParameters)
{
    u8 num;
    u8 semavalue;
    while(1)
    {
        xSemaphoreTake(CountSemaphore,portMAX_DELAY);    //等待数值信号量        (5)
        num++;
        semavalue=uxSemaphoreGetCount(CountSemaphore);    //获取数值信号量值    (6)
        LCD_ShowxNum(155,111,semavalue,3,16,0);           //显示信号量值
        LCD_Fill(6,131,233,313,lcd_discolor[num%14]);     //刷屏
        LED1=!LED1;
        vTaskDelay(1000);                          //延时 1s，也就是 1000 个时钟节拍
    }
}
```

(1)、要使用计数型信号量，首先肯定是要先创建，调用函数 xSemaphoreCreateCounting()创建一个计数型信号量 CountSemaphore。计数型信号量计数最大值设置为 255，由于本实验中计数型信号量是用于事件计数的，所以计数型信号量的初始值设置为 0。如果计数型信号量用于资源管理的话那么事件计数型信号量的初始值就应该根据资源的实际数量来设置了。

(2)、如果 KEY_UP 键按下的话就表示事件发生了，事件发生以后就调用函数 xSemaphoreGive()释放信号量 CountSemaphore。

(3)、调用函数 uxSemaphoreGetCount()获取信号量 CountSemaphore 的信号量值，释放信号量的话信号量值就会加一。函数 uxSemaphoreGetCount()是用来获取信号量值的，这个函数是个宏，是对函数 uxQueueMessagesWaiting()的一个简单封装，其实就是返回队列结构体成员变量 uxMessagesWaiting 的值。

(4)、在 LCD 上显示信号量 CountSemaphore 的信号量值，可以直观的观察信号量的变化过程。

(5)、调用函数 xSemaphoreTake()获取信号量 CountSemaphore。

(6)、同样的在获取信号量以后调用函数 uxSemaphoreGetCount()获取信号量值，并且在 LCD 上显示出来，因为获取信号量成功以后信号量值会减一。

14.5.2 程序运行结果分析

编译并下载实验代码到开发板中，默认情况下 LCD 显示如图 14.5.2.1 所示。

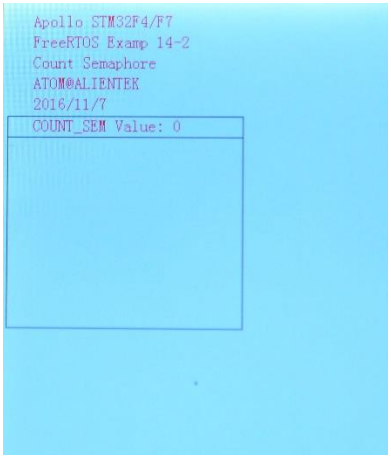


图 14.5.2.1 LCD 默认显示画面

按下 KEY_UP 键释放信号量 CountSemaphore，注意观察 LCD 上的信号量值的变化。信号量有效以后任务 SemapTake_task()就可以获取到信号量，获取到信号量的话就会刷新 LCD 指定区域的背景色。释放信号量的话信号量值就会增加，获取信号量的话信号量值就会减少，如图 14.5.2.2 所示：

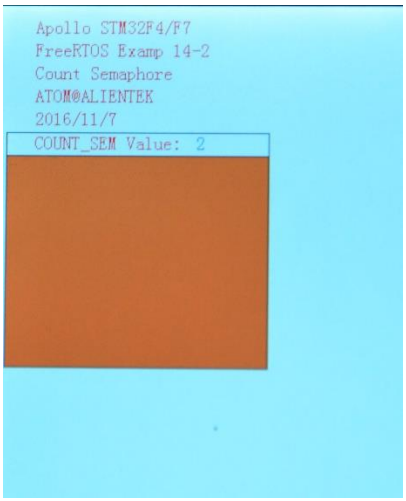


图 14.5.2.2 信号量变化

当信号量值减为 0 的时候就表示信号量无效，任务 SemapTake_task()获取信号量失败，任务因此进入阻塞态，LCD 指定区域背景色刷新停止，看起来任务就好像是“停止”运行了。

14.6 优先级翻转

在使用二值信号量的时候会遇到很常见的一个问题——优先级翻转，优先级翻转在可剥夺内核中是非常常见的，在实时系统中不允许出现这种现象，这样会破坏任务的预期顺序，可能会导致严重的后果，图 14.6.1 就是一个优先级翻转的例子。，

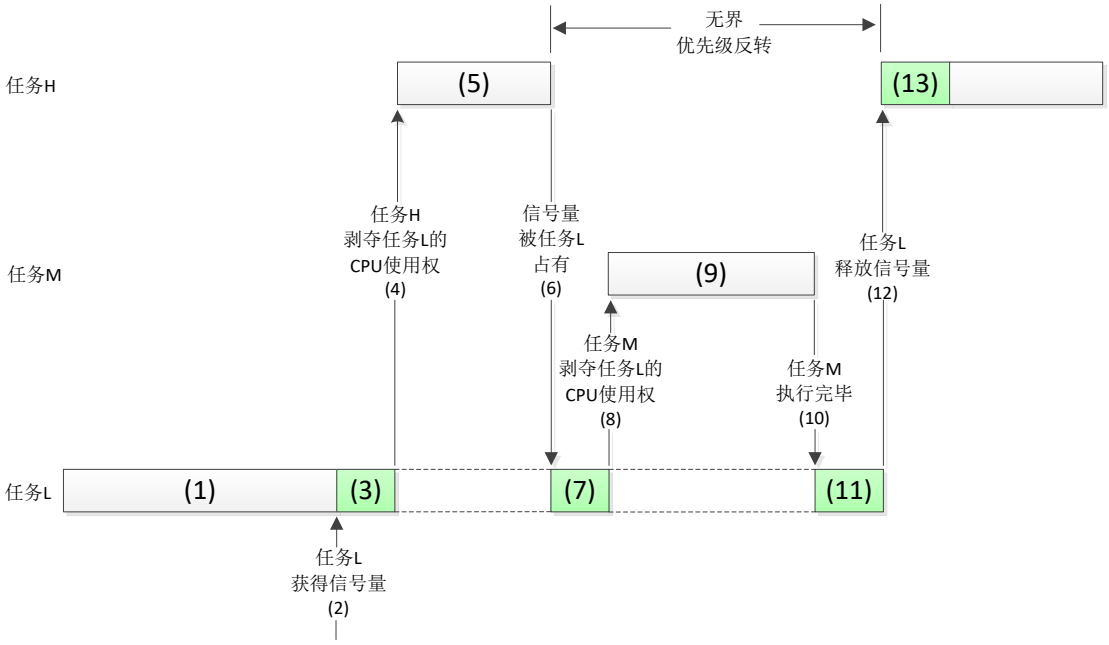


图 14.6.1 优先级翻转示意图

(1) 任务 H 和任务 M 处于挂起状态，等待某一事件的发生，任务 L 正在运行。

(2) 某一时刻任务 L 想要访问共享资源，在此之前它必须先获得对应该资源的信号量。

(3) 任务 L 获得信号量并开始使用该共享资源。

(4) 由于任务 H 优先级高，它等待的事件发生后便剥夺了任务 L 的 CPU 使用权。

(5) 任务 H 开始运行。

(6) 任务 H 运行过程中也要使用任务 L 正在使用着的资源，由于该资源的信号量还被任务 L 占用着，任务 H 只能进入挂起状态，等待任务 L 释放该信号量。

(7) 任务 L 继续运行。

(8) 由于任务 M 的优先级高于任务 L，当任务 M 等待的事件发生后，任务 M 剥夺了任务 L 的 CPU 使用权。

(9) 任务 M 处理该处理的事。

(10) 任务 M 执行完毕后，将 CPU 使用权归还给任务 L。

(11) 任务 L 继续运行。

(12) 最终任务 L 完成所有的工作并释放了信号量，到此为止，由于实时内核知道有个高优先级的任务在等待这个信号量，故内核做任务切换。

(13) 任务 H 得到该信号量并接着运行。

在这种情况下，任务 H 的优先级实际上降到了任务 L 的优先级水平。因为任务 H 要一直等待直到任务 L 释放其占用的那个共享资源。由于任务 M 剥夺了任务 L 的 CPU 使用权，使得任务 H 的情况更加恶化，这样就相当于任务 M 的优先级高于任务 H，导致优先级翻转。

14.7 优先级翻转实验

14.7.1 实验程序设计

1、实验目的

在使用二值信号量的时候会存在优先级翻转的问题，本实验通过模拟的方式实现优先级翻转，观察优先级翻转对抢占式内核的影响。

2、实验设计

本实验设计四个任务：start_task、high_task 、middle_task ，low_task，这四个任务的任务功能如下：

start_task：用来创建其他 3 个任务。

high_task ：高优先级任务，会获取二值信号量，获取成功以后会进行相应的处理，处理完成以后就会释放二值信号量。

middle_task ：中等优先级的任务，一个简单的应用任务。

low_task：低优先级任务，和高优先级任务一样，会获取二值信号量，获取成功以后会进行相应的处理，不过不同之处在于低优先级的任务占用二值信号量的时间要久一点(软件模拟占用)。

实验中创建了一个二值信号量 BinarySemaphore，高优先级和低优先级这两个任务会使用这个二值信号量。

3、实验工程

FreeRTOS 实验 14-3 FreeRTOS 优先级翻转实验。

4、实验程序与分析

●任务设置

```
#define START_TASK_PRIO          1          //任务优先级
#define START_STK_SIZE           256        //任务堆栈大小
TaskHandle_t StartTask_Handler;             //任务句柄
void start_task(void *pvParameters);        //任务函数

#define LOW_TASK_PRIO            2          //任务优先级
#define LOW_STK_SIZE             256        //任务堆栈大小
TaskHandle_t LowTask_Handler;               //任务句柄
void low_task(void *pvParameters);          //任务函数

#define MIDDLE_TASK_PRIO         3          //任务优先级
#define MIDDLE_STK_SIZE          256        //任务堆栈大小
TaskHandle_t MiddleTask_Handler;            //任务句柄
void middle_task(void *pvParameters);       //任务函数

#define HIGH_TASK_PRIO           4          //任务优先级
#define HIGH_STK_SIZE            256        //任务堆栈大小
TaskHandle_t HighTask_Handler;              //任务句柄
void high_task(void *pvParameters);         //任务函数

//二值信号量句柄
SemaphoreHandle_t BinarySemaphore;//二值信号量

//LCD 刷屏时使用的颜色
int lcd_discolor[14]={   WHITE,      BLACK,      BLUE,       BRED,
```

```
                    GRED,       GBLUE,      RED,        MAGENTA,
                    GREEN,      CYAN,       YELLOW,     BROWN,
                    BRRED,      GRAY };
```

● **main()函数**

```
int main(void)
{
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(360,25,2,8);   //设置时钟,180Mhz
    delay_init(180);                //初始化延时函数
    uart_init(115200);              //初始化串口
    LED_Init();                     //初始化 LED
    KEY_Init();                     //初始化按键
    PCF8574_Init();                 //初始化 PCF8574
    SDRAM_Init();                   //初始化 SDRAM
    LCD_Init();                     //初始化 LCD
    my_mem_init(SRAMIN);            //初始化内部内存池

    POINT_COLOR = RED;
    LCD_ShowString(30,10,200,16,16,"Apollo STM32F4/F7");
    LCD_ShowString(30,30,200,16,16,"FreeRTOS Examp 14-3");
    LCD_ShowString(30,50,200,16,16,"Priority Overturn");
    LCD_ShowString(30,70,200,16,16,"ATOM@ALIENTEK");
    LCD_ShowString(30,90,200,16,16,"2016/11/8");

    //创建开始任务
    xTaskCreate((TaskFunction_t     )start_task,            //任务函数
                (const char*        )"start_task",          //任务名称
                (uint16_t           )START_STK_SIZE,        //任务堆栈大小
                (void*              )NULL,                  //传递给任务函数的参数
                (UBaseType_t        )START_TASK_PRIO,       //任务优先级
                (TaskHandle_t*      )&StartTask_Handler);   //任务句柄
    vTaskStartScheduler();                                  //开启任务调度
}
```

● **任务函数**

```
//开始任务任务函数
void start_task(void *pvParameters)
{
    taskENTER_CRITICAL();           //进入临界区

    //创建二值信号量
    BinarySemaphore=xSemaphoreCreateBinary();                   (1)
    //二值信号量创建成功以后要先释放一下
```

```
    if(BinarySemaphore!=NULL)xSemaphoreGive(BinarySemaphore);          (2)

    //创建高优先级任务
    xTaskCreate((TaskFunction_t     )high_task,
                (const char*        )"high_task",
                (uint16_t           )HIGH_STK_SIZE,
                (void*              )NULL,
                (UBaseType_t        )HIGH_TASK_PRIO,
                (TaskHandle_t*      )&HighTask_Handler);
    //创建中等优先级任务
    xTaskCreate((TaskFunction_t     )middle_task,
                (const char*        )"middle_task",
                (uint16_t           )MIDDLE_STK_SIZE,
                (void*              )NULL,
                (UBaseType_t        )MIDDLE_TASK_PRIO,
                (TaskHandle_t*      )&MiddleTask_Handler);
    //创建低优先级任务
    xTaskCreate((TaskFunction_t     )low_task,
                (const char*        )"low_task",
                (uint16_t           )LOW_STK_SIZE,
                (void*              )NULL,
                (UBaseType_t        )LOW_TASK_PRIO,
                (TaskHandle_t*      )&LowTask_Handler);
    vTaskDelete(StartTask_Handler);         //删除开始任务
    taskEXIT_CRITICAL();                    //退出临界区
}

//高优先级任务的任务函数
void high_task(void *pvParameters)
{
    u8 num;

    POINT_COLOR = BLACK;
    LCD_DrawRectangle(5,110,115,314);  //画一个矩形
    LCD_DrawLine(5,130,115,130);       //画线
    POINT_COLOR = BLUE;
    LCD_ShowString(6,111,110,16,16,"High Task");

    while(1)
    {
        vTaskDelay(500);  //延时 500ms，也就是 500 个时钟节拍
        num++;
        printf("high task Pend Sem\r\n");
```

```
        xSemaphoreTake(BinarySemaphore,portMAX_DELAY);  //获取二值信号量        (3)
        printf("high task Running!\r\n");
        LCD_Fill(6,131,114,313,lcd_discolor[num%14]);      //填充区域
        LED1=!LED1;
        xSemaphoreGive(BinarySemaphore);                   //释放信号量          (4)
        vTaskDelay(500);  //延时 500ms，也就是 500 个时钟节拍
    }
}

//中等优先级任务的任务函数
void middle_task(void *pvParameters)
{
    u8 num;

    POINT_COLOR = BLACK;
    LCD_DrawRectangle(125,110,234,314);     //画一个矩形
    LCD_DrawLine(125,130,234,130);          //画线
    POINT_COLOR = BLUE;
    LCD_ShowString(126,111,110,16,16,"Middle Task");
    while(1)
    {
        num++;
        printf("middle task Running!\r\n");
        LCD_Fill(126,131,233,313,lcd_discolor[13-num%14]); //填充区域
        LED0=!LED0;
        vTaskDelay(1000); //延时 1s，也就是 1000 个时钟节拍
    }
}

//低优先级任务的任务函数
void low_task(void *pvParameters)
{
    static u32 times;

    while(1)
    {
        xSemaphoreTake(BinarySemaphore,portMAX_DELAY);  //获取二值信号量        (5)
        printf("low task Running!\r\n");
        for(times=0;times<20000000;times++) //模拟低优先级任务占用二值信号量     (6)
        {
            taskYIELD();                                //发起任务调度
        }
        xSemaphoreGive(BinarySemaphore);                //释放二值信号量        (7)
```

```
        vTaskDelay(1000); //延时 1s，也就是 1000 个时钟节拍
    }
}
```

(1)、调用函数 xSemaphoreCreateBinary()创建二值信号量。

(2)、默认创建的二值信号量是无效的，这里需要先调用函数 xSemaphoreGive()释放一次二值信号量。否则任务 high_task()和 low_task()都会获取不到信号量。

(3)、高优先级任务调用函数 xSemaphoreTake()获取二值信号量。

(4)、使用完以后需要调用函数 xSemaphoreGive()释放二值信号量。

(5)、低优先级任务获取二值信号量 BinarySemaphore。

(6)、低优先级任务模拟长时间占用二值信号量。

(7)、低优先级任务释放二值信号量。

14.7.2 程序运行结果分析

编译并下载实验代码到开发板中，打开串口调试助手，默认情况下 LCD 显示如图 14.7.2.1 所示。

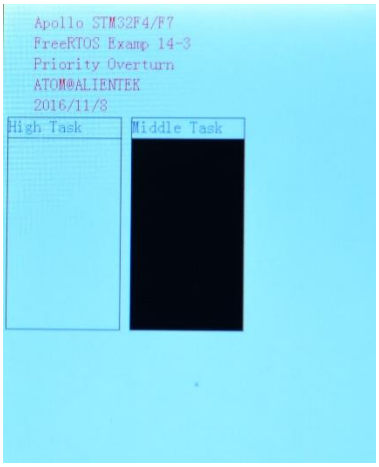


图 14.7.2.1 LCD 默认画面

从 LCD 上不容易看出优先级翻转的现象，我们可以通过串口很方便的观察优先级翻转，串口输出如图 14.7.2.2 所示。

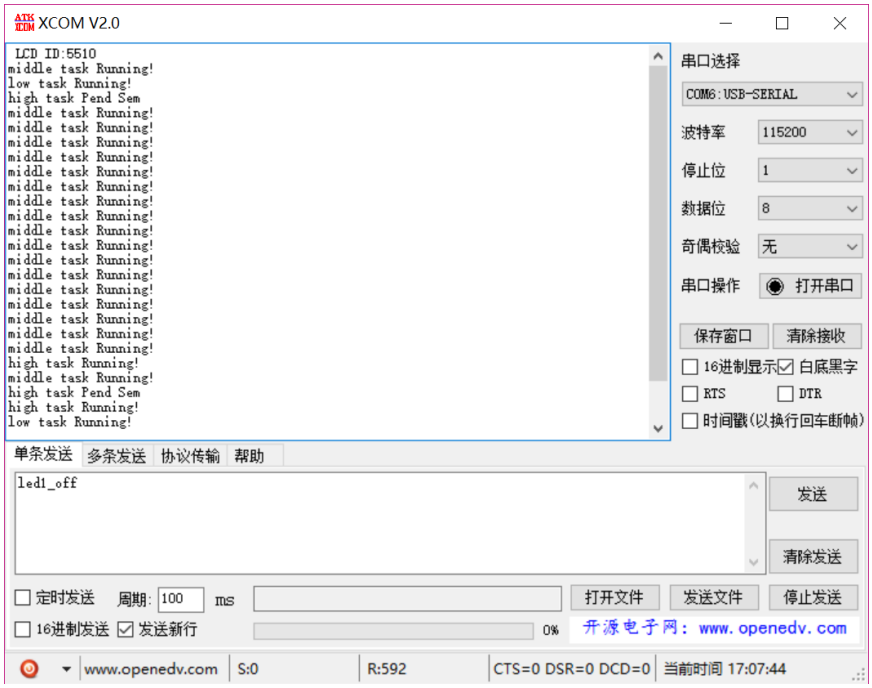


图 14.7.2.2 串口输出

为了方便分析，我们将串口输出复制出来，如下：

```
LCD ID:5510
middle task Running!
low task Running!                    (1)
high task Pend Sem                   (2)
middle task Running!                 (3)
middle task Running!
middle task Running!
middle task Running!
middle task Running!
middle task Running!
middle task Running!
middle task Running!
middle task Running!
middle task Running!
middle task Running!
middle task Running!
middle task Running!
middle task Running!
middle task Running!
middle task Running!
middle task Running!
middle task Running!
high task Running!                   (4)
middle task Running!
high task Pend Sem
```

(1)、low_task 任务获取到二值信号量 BinarySemaphore 开始运行。

(2)、high_task 获取信号量 BinarySemaphore，但是此时信号量 BinarySemaphore 被任务 low_task 占用着，因此 high_task 就要一直等待，直到 low_task 任务释放信号量 BinarySemaphore。

(3)、由于 high_task 没有获取到信号量 BinarySemaphore，只能一直等待，红色部分代码中 high_task 没有运行，而 middle_task 一直在运行，给人的感觉就是 middle_task 的任务优先级高于 high_task。但是事实上 high_task 任务的任务优先级是高于 middle_task 的，这个就是优先级反转！

(4)、high_task 任务因为获取到了信号量 BinarySemaphore 而运行

从本例程中可以看出，当一个低优先级任务和一个高优先级任务同时使用同一个信号量，而系统中还有其他中等优先级任务时。如果低优先级任务获得了信号量，那么高优先级的任务就会处于等待状态，但是，中等优先级的任务可以打断低优先级任务而先于高优先级任务运行(此时高优先级的任务在等待信号量 ，所以不能运行)，这是就出现了优先级翻转的现象。

既然优先级翻转是个很严重的问题，那么有没有解决方法呢？有！这就要引出另外一种信号量——互斥信号量！

14.8 互斥信号量

14.8.1 互斥信号量简介

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(任务与任务或中断与任务之间的同步)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。在互斥访问中互斥信号量相当于一个钥匙，当任务想要使用资源的时候就必须先获得这个钥匙，当使用完资源以后就必须归还这个钥匙，这样其他的任务就可以拿着这个钥匙去使用资源。

互斥信号量使用和二值信号量相同的 API 操作函数，所以互斥信号量也可以设置阻塞时间，不同于二值信号量的是互斥信号量具有优先级继承的特性。当一个互斥信号量正在被一个低优先级的任务使用，而此时有个高优先级的任务也尝试获取这个互斥信号量的话就会被阻塞。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，这个过程就是优先级继承。优先级继承尽可能的降低了高优先级任务处于阻塞态的时间，并且将已经出现的“优先级翻转”的影响降到最低。

优先级继承并不能完全的消除优先级翻转，它只是尽可能的降低优先级翻转带来的影响。硬实时应用应该在设计之初就要避免优先级翻转的发生。互斥信号量不能用于中断服务函数中，原因如下：

- 互斥信号量有优先级继承的机制，所以只能用在任务中，不能用于中断服务函数。
- 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。

14.8.2 创建互斥信号量

FreeRTOS 提供了两个互斥信号量创建函数，如表 14.8.2.1 所示：

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。

表 14.8.2.1 互斥信号量创建函数

1、函数 xSemaphoreCreateMutex()

此函数用于创建一个互斥信号量，所需要的内存通过动态内存管理方法分配。此函数本质

是一个宏，真正完成信号量创建的是函数 xQueueCreateMutex()，此函数原型如下：

```
SemaphoreHandle_t      xSemaphoreCreateMutex( void )
```

参数：

无。

返回值：

NULL: 互斥信号量创建失败。
其他值: 创建成功的互斥信号量的句柄。

2、函数 xSemaphoreCreateMutexStatic()

此函数也是创建互斥信号量的，只不过使用此函数创建互斥信号量的话信号量所需要的 RAM 需要由用户来分配，此函数是个宏，具体创建过程是通过函数 xQueueCreateMutexStatic () 来完成的，函数原型如下：

```
SemaphoreHandle_t      xSemaphoreCreateMutexStatic( StaticSemaphore_t *pxMutexBuffer )
```

参数：

pxMutexBuffer：此参数指向一个 StaticSemaphore_t 类型的变量，用来保存信号量结构体。

返回值：

NULL: 互斥信号量创建失败。
其他值: 创建成功的互斥信号量的句柄。

14.8.3 互斥信号量创建过程分析

这里只分析动态创建互斥信号量函数 xSemaphoreCreateMutex ()，此函数是个宏，定义如下：

```
#define   xSemaphoreCreateMutex()   xQueueCreateMutex( queueQUEUE_TYPE_MUTEX )
```

可以看出，真正干事的是函数 xQueueCreateMutex()，此函数在文件 queue.c 中有如下定义，

```
QueueHandle_t xQueueCreateMutex( const uint8_t ucQueueType )
{
    Queue_t *pxNewQueue;
    const UBaseType_t uxMutexLength = ( UBaseType_t ) 1, uxMutexSize = ( UBaseType_t ) 0;

    pxNewQueue = ( Queue_t * ) xQueueGenericCreate( uxMutexLength, uxMutexSize,\      (1)
                                        ucQueueType );
    prvInitialiseMutex( pxNewQueue );                                                 (2)

    return pxNewQueue;
}
```

(1)、调用函数 xQueueGenericCreate()创建一个队列，队列长度为 1，队列项长度为 0，队列类型为参数 ucQueueType。由于本函数是创建互斥信号量的，所以参数 ucQueueType 为 queueQUEUE_TYPE_MUTEX。

(2)、调用函数 prvInitialiseMutex()初始化互斥信号量。

函数 prvInitialiseMutex()代码如下：

```
static void prvInitialiseMutex( Queue_t *pxNewQueue )
{
    if( pxNewQueue != NULL )
    {
        //虽然创建队列的时候会初始化队列结构体的成员变量，但是此时创建的是互斥
        //信号量，因此有些成员变量需要重新赋值，尤其是那些用于优先级继承的。
        pxNewQueue->pxMutexHolder = NULL;                    (1)
        pxNewQueue->uxQueueType = queueQUEUE_IS_MUTEX;       (2)

        //如果是递归互斥信号量的话。
        pxNewQueue->u.uxRecursiveCallCount = 0;              (3)

        traceCREATE_MUTEX( pxNewQueue );

        //释放互斥信号量
        ( void ) xQueueGenericSend( pxNewQueue, NULL, ( TickType_t ) 0U,\
                                    queueSEND_TO_BACK );
    }
    else
    {
        traceCREATE_MUTEX_FAILED();
    }
}
```

(1)和(2)、这里大家可能会疑惑，队列结构体 Queue_t 中没有 pxMutexHolder 和 uxQueueType 这两个成员变量吖？这两个东西是哪里来的妖孽？这两个其实是宏，专门为互斥信号量准备的，在文件 queue.c 中有如下定义：

```
#define pxMutexHolder                   pcTail
#define uxQueueType                     pcHead
#define queueQUEUE_IS_MUTEX             NULL
```

当 Queue_t 用于表示队列的时候 pcHead 和 pcTail 指向队列的存储区域，当 Queue_t 用于表示互斥信号量的时候就不需要 pcHead 和 pcTail 了。当用于互斥信号量的时候将 pcHead 指向 NULL 来表示 pcTail 保存着互斥队列的所有者，pxMutexHolder 指向拥有互斥信号量的那个任务的任务控制块。重命名 pcTail 和 pcHead 就是为了增强代码的可读性。

(3)、如果创建的互斥信号量是互斥信号量的话，还需要初始化队列结构体中的成员变量 u.uxRecursiveCallCount。

互斥信号量创建成功以后会调用函数 xQueueGenericSend()释放一次信号量，说明互斥信号量默认就是有效的！互斥信号量创建完成以后如图 14.8.3.1 所示：

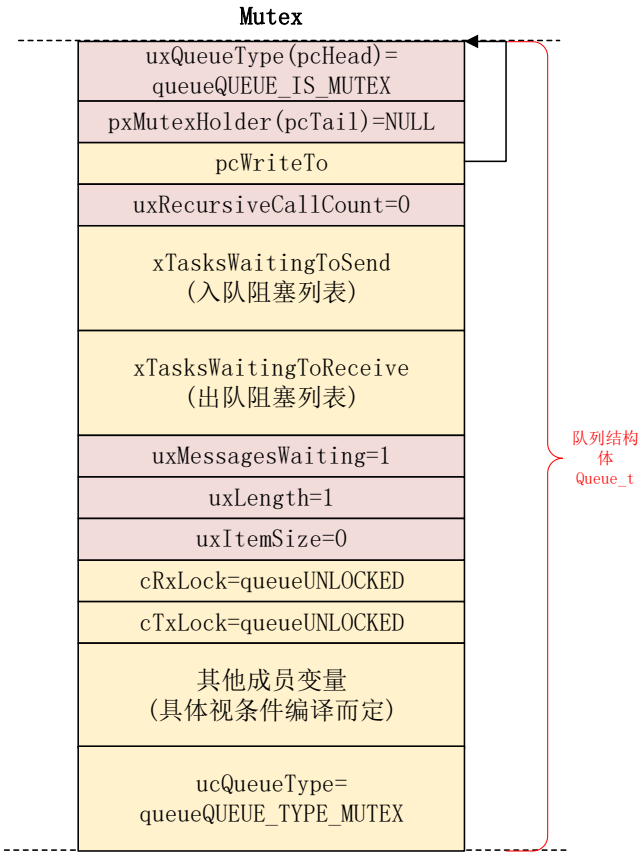


图 14.8.3.1 互斥信号量初始化结构图

14.8.4 释放互斥信号量

释放互斥信号量的时候和二值信号量、计数型信号量一样，都是用的函数 xSemaphoreGive()(实际上完成信号量释放的是函数 xQueueGenericSend())。不过由于互斥信号量涉及到优先级继承的问题，所以具体处理过程会有点区别。使用函数 xSemaphoreGive()释放信号量最重要的一步就是将 uxMessagesWaiting 加一，而这一步就是通过函数 prvCopyDataToQueue() 来完成的，释放信号量的函数 xQueueGenericSend() 会调用 prvCopyDataToQueue()。互斥信号量的优先级继承也是在函数 prvCopyDataToQueue()中完成的，此函数中有如下一段代码：

```
static BaseType_t prvCopyDataToQueue( Queue_t * const        pxQueue,
                                      const void *           pvItemToQueue,
                                      const BaseType_t       xPosition )
{
    BaseType_t xReturn = pdFALSE;
    UBaseType_t uxMessagesWaiting;

    uxMessagesWaiting = pxQueue->uxMessagesWaiting;

    if( pxQueue->uxItemSize == ( UBaseType_t ) 0 )
    {
        #if ( configUSE_MUTEXES == 1 )                    //互斥信号量
```

```
        {
            if( pxQueue->uxQueueType == queueQUEUE_IS_MUTEX )                    (1)
            {
                xReturn = xTaskPriorityDisinherit( ( void * ) pxQueue->pxMutexHolder );(2)
                pxQueue->pxMutexHolder = NULL;                                   (3)
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
            }
        }
        #endif /* configUSE_MUTEXES */
    }

    /*****************************************************************************/
    /************************省略掉其他处理代码*************************************/
    /*****************************************************************************/

    pxQueue->uxMessagesWaiting = uxMessagesWaiting + 1;

    return xReturn;
}
```

(1)、当前操作的是互斥信号量。

(2)、调用函数 xTaskPriorityDisinherit()处理互斥信号量的优先级继承问题。

(3)、互斥信号量释放以后，互斥信号量就不属于任何任务了，所以 pxMutexHolder 要指向 NULL。

在来看一下函数 xTaskPriorityDisinherit()是怎么具体的处理优先级继承的，函数 xTaskPriorityDisinherit()代码如下：

```
BaseType_t xTaskPriorityDisinherit( TaskHandle_t const pxMutexHolder )
 {
    TCB_t * const pxTCB = ( TCB_t * ) pxMutexHolder;
    BaseType_t xReturn = pdFALSE;

    if( pxMutexHolder != NULL )                                                  (1)
    {
        //当一个任务获取到互斥信号量以后就会涉及到优先级继承的问题，正在释放互斥
        //信号量的任务肯定是当前正在运行的任务 pxCurrentTCB。
        configASSERT( pxTCB == pxCurrentTCB );
        configASSERT( pxTCB->uxMutexesHeld );

        ( pxTCB->uxMutexesHeld )--;                                              (2)

        //是否存在优先级继承？如果存在的话任务当前优先级肯定和任务基优先级不同。
```

```
        if( pxTCB->uxPriority != pxTCB->uxBasePriority )                    (3)
        {
            //当前任务只获取到了一个互斥信号量
            if( pxTCB->uxMutexesHeld == ( UBaseType_t ) 0 )                 (4)
            {
                if( uxListRemove( &( pxTCB->xStateListItem ) ) == ( UBaseType_t ) 0 )  (5)
                {
                    taskRESET_READY_PRIORITY( pxTCB->uxPriority );          (6)
                }
                else
                {
                    mtCOVERAGE_TEST_MARKER();
                }

                //使用新的优先级将任务重新添加到就绪列表中
                traceTASK_PRIORITY_DISINHERIT( pxTCB, pxTCB->uxBasePriority );
                pxTCB->uxPriority = pxTCB->uxBasePriority;                  (7)

                    /* Reset the event list item value.  It cannot be in use for
                    any other purpose if this task is running, and it must be
                    running to give back the mutex. */
                listSET_LIST_ITEM_VALUE( &( pxTCB->xEventListItem ), \      (8)
                                         ( TickType_t ) configMAX_PRIORITIES - \
                                         ( TickType_t ) pxTCB->uxPriority );
                prvAddTaskToReadyList( pxTCB );                             (9)
                xReturn = pdTRUE;                                           (10)
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
            }
        }
        else
        {
            mtCOVERAGE_TEST_MARKER();
        }
    }
    else
    {
        mtCOVERAGE_TEST_MARKER();
    }

    return xReturn;
```

```
}
```

(1)、函数的参数 pxMutexHolder 表示拥有此互斥信号量任务控制块，所以要先判断此互斥信号量是否已经被其他任务获取。

(2)、有的任务可能会获取多个互斥信号量，所以就需要标记任务当前获取到的互斥信号量个数，任务控制块结构体的成员变量 uxMutexesHeld 用来保存当前任务获取到的互斥信号量个数。任务每释放一次互斥信号量，变量 uxMutexesHeld 肯定就要减一。

(3)、判断是否存在优先级继承，如果存在的话任务的当前优先级肯定不等于任务的基优先级。

(4)、判断当前释放的是不是任务所获取到的最后一个互斥信号量，因为如果任务还获取了其他互斥信号量的话就不能处理优先级继承。优先级继承的处理必须是在释放最后一个互斥信号量的时候。

(5)、优先级继承的处理说白了就是将任务的当前优先级降低到任务的基优先级，所以要把当前任务先从任务就绪表中移除。当任务优先级恢复为原来的优先级以后再重新加入到就绪表中。

(6)、如果任务继承来的这个优先级对应的就绪表中没有其他任务的话就将取消这个优先级的就绪态。

(7)、重新设置任务的优先级为任务的基优先级 uxBasePriority。

(8)、复位任务的事件列表项。

(9)、将优先级恢复后的任务重新添加到任务就绪表中。

(10)、返回 pdTRUE，表示需要进行任务调度。

14.8.5 获取互斥信号量

获取互斥信号量的函数同获取二值信号量和计数型信号量的函数相同，都是 xSemaphoreTake()(实际执行信号量获取的函数是 xQueueGenericReceive())，获取互斥信号量的过程也需要处理优先级继承的问题，函数 xQueueGenericReceive()在文件 queue.c 中有定义，在第十三章讲解队列的时候我们没有分析这个函数，本节就来简单的分析一下这个函数，缩减后的函数代码如下：

```
BaseType_t xQueueGenericReceive( QueueHandle_t xQueue, void * const pvBuffer, TickType_t xTicksToWait, const BaseType_t xJustPeeking )
{
    BaseType_t xEntryTimeSet = pdFALSE;
    TimeOut_t xTimeOut;
    int8_t *pcOriginalReadPosition;
    Queue_t * const pxQueue = ( Queue_t * ) xQueue;

    for( ;; )
    {
        taskENTER_CRITICAL();
        {
            const UBaseType_t uxMessagesWaiting = pxQueue->uxMessagesWaiting;

            //判断队列是否有消息
            if( uxMessagesWaiting > ( UBaseType_t ) 0 )                    (1)
```

```
{
    pcOriginalReadPosition = pxQueue->u.pcReadFrom;
    prvCopyDataFromQueue( pxQueue, pvBuffer );                    (2)

    if( xJustPeeking == pdFALSE )                                 (3)
    {
        traceQUEUE_RECEIVE( pxQueue );

        //移除消息
        pxQueue->uxMessagesWaiting = uxMessagesWaiting - 1;       (4)
        #if ( configUSE_MUTEXES == 1 )                            (5)
        {
            if( pxQueue->uxQueueType == queueQUEUE_IS_MUTEX )
            {
                pxQueue->pxMutexHolder =                          (6)
                  ( int8_t * ) pvTaskIncrementMutexHeldCount();
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
            }
        }
        #endif /* configUSE_MUTEXES */

        //查看是否有任务因为入队而阻塞，如果有的话就需要解除阻塞态。
        if( listLIST_IS_EMPTY( &( pxQueue->xTasksWaitingToSend ) ) == (7)
                                    pdFALSE )
        {
            if( xTaskRemoveFromEventList( &
                ( pxQueue->xTasksWaitingToSend ) ) != pdFALSE )
            {
                //如果解除阻塞的任务优先级比当前任务优先级高的话就需要
                //进行一次任务切换
                queueYIELD_IF_USING_PREEMPTION();
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
            }
        }
        else
        {
            mtCOVERAGE_TEST_MARKER();
```

```
            }
        }
        else                                                    (8)
        {
            traceQUEUE_PEEK( pxQueue );
            //读取队列中的消息以后需要删除消息
            pxQueue->u.pcReadFrom = pcOriginalReadPosition;

            //如果有任务因为出队而阻塞的话就解除任务的阻塞态。
            if( listLIST_IS_EMPTY( &( pxQueue->xTasksWaitingToReceive ) ) == (9)
                                                                pdFALSE )
            {
                if( xTaskRemoveFromEventList( &
                  ( pxQueue->xTasksWaitingToReceive ) ) != pdFALSE )
                {
                    //如果解除阻塞的任务优先级比当前任务优先级高的话就需要
                    //进行一次任务切换
                    queueYIELD_IF_USING_PREEMPTION();
                }
                else
                {
                    mtCOVERAGE_TEST_MARKER();
                }
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
            }
        }

        taskEXIT_CRITICAL();
        return pdPASS;
    }
    else                                    //队列为空            (10)
    {
        if( xTicksToWait == ( TickType_t ) 0 )
        {
            //队列为空，如果阻塞时间为 0 的话就直接返回 errQUEUE_EMPTY
            taskEXIT_CRITICAL();
            traceQUEUE_RECEIVE_FAILED( pxQueue );
            return errQUEUE_EMPTY;
        }
        else if( xEntryTimeSet == pdFALSE )
```

```
            {
                //队列为空并且设置了阻塞时间，需要初始化时间状态结构体。
                vTaskSetTimeOutState( &xTimeOut );
                xEntryTimeSet = pdTRUE;
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
            }
        }
    }
    taskEXIT_CRITICAL();

    vTaskSuspendAll();
    prvLockQueue( pxQueue );

    //更新时间状态结构体，并且检查超时是否发生
    if( xTaskCheckForTimeOut( &xTimeOut, &xTicksToWait ) == pdFALSE )       (11)
    {
        if( prvIsQueueEmpty( pxQueue ) != pdFALSE )                         (12)
        {
            traceBLOCKING_ON_QUEUE_RECEIVE( pxQueue );

            #if ( configUSE_MUTEXES == 1 )
            {
                if( pxQueue->uxQueueType == queueQUEUE_IS_MUTEX )     (13)
                {
                    taskENTER_CRITICAL();
                    {
                        vTaskPriorityInherit( ( void * ) pxQueue->pxMutexHolder );(14)
                    }
                    taskEXIT_CRITICAL();
                }
                else
                {
                    mtCOVERAGE_TEST_MARKER();
                }
            }
            #endif

            vTaskPlaceOnEventList( &( pxQueue->xTasksWaitingToReceive ),     (15)
                                   xTicksToWait );
            prvUnlockQueue( pxQueue );
```

```
                    if( xTaskResumeAll() == pdFALSE )
                    {
                        portYIELD_WITHIN_API();
                    }
                    else
                    {
                        mtCOVERAGE_TEST_MARKER();
                    }
                }
                else
                {
                    //重试一次
                    prvUnlockQueue( pxQueue );
                    ( void ) xTaskResumeAll();
                }
            }
            else
            {
                prvUnlockQueue( pxQueue );
                ( void ) xTaskResumeAll();

                if( prvIsQueueEmpty( pxQueue ) != pdFALSE )
                {
                    traceQUEUE_RECEIVE_FAILED( pxQueue );
                    return errQUEUE_EMPTY;
                }
                else
                {
                    mtCOVERAGE_TEST_MARKER();
                }
            }
        }
    }
}
```

(1)、队列不为空，可以从队列中提取数据。

(2)、调用函数 prvCopyDataFromQueue()使用数据拷贝的方式从队列中提取数据。

(3)、数据读取以后需要将数据删除掉。

(4)、队列的消息数量计数器 uxMessagesWaiting 减一，通过这一步就将数据删除掉了。

(5)、表示此函数是用于获取互斥信号量的。

(6)、获取互斥信号量成功，需要标记互斥信号量的所有者，也就是给 pxMutexHolder 赋值，pxMutexHolder 应该是当前任务的任务控制块。但是这里是通过函数 pvTaskIncrementMutexHeldCount()来赋值的，此函数很简单，只是将任务控制块中的成员变量 uxMutexesHeld 加一，表示任务获取到了一个互斥信号量，最后此函数返回当前任务的任务控制块。

(7)、出队成功以后判断是否有任务因为入队而阻塞的，如果有的话就需要解除任务的阻塞态，如果解除阻塞的任务优先级比当前任务的优先级高还需要进行一次任务切换。

(8)、出队的时候不需要删除消息。

(9)、如果出队的时候不需要删除消息的话那就相当于刚刚出队的那条消息接着有效！既然还有有效的消息存在队列中，那么就判断是否有任务因为出队而阻塞，如果有的话就解除任务的阻塞态。同样的，如果解除阻塞的任务优先级比当前任务的优先级高的话还需要进行一次任务切换。

(10)、上面分析的都是队列不为空的时候，那当队列为空的时候该如何处理呢？处理过程和队列的任务级通用入队函数 xQueueGenericSend()类似。如果阻塞时间为 0 的话就就直接返回 errQUEUE_EMPTY，表示队列空，如果设置了阻塞时间的话就进行相关的处理。

(11)、检查超时是否发生，如果没有的话就需要将任务添加到队列的 xTasksWaitingToReceive 列表中。

(12)、检查队列是否继续为空？如果不为空的话就会在重试一次出队。

(13)、表示此函数是用于获取互斥信号量的。

(14)、调用函数 vTaskPriorityInherit()处理互斥信号量中的优先级继承问题，如果函数 xQueueGenericReceive()用于获取互斥信号量的话，此函数执行到这里说明互斥信号量正在被其他的任务占用。此函数和 14.8.4 小节中的函数 xTaskPriorityDisinherit()过程相反。此函数会判断当前任务的任务优先级是否比正在拥有互斥信号量的那个任务的任务优先级高，如果是的话就会把拥有互斥信号量的那个低优先级任务的优先级调整为与当前任务相同的优先级！

(15)、经过(12)步判断，队列依旧为空，那么就将任务添加到列表 xTasksWaitingToReceive 中。

在上面的分析中，红色部分就是当函数 xQueueGenericReceive()用于互斥信号量的时候的处理过程，其中(13)和(14)条详细的分析了互斥信号量优先级继承的过程。我们举个例子来简单的演示一下这个过程，假设现在有两个任务 HighTask 和 LowTask，HighTask 的任务优先级为 4，LowTask 的任务优先级为 2。这两个任务都会操同一个互斥信号量 Mutex，LowTask 先获取到互斥信号量 Mutex。此时任务 HighTask 也要获取互斥信号量 Mutex，任务 HighTask 调用函数 xSemaphoreTake()尝试获取互斥信号量 Mutex，发现此互斥信号量正在被任务 LowTask 使用，并且 LowTask 的任务优先级为 2，比自己的任务优先级小，因为任务 HighTask 就会将 LowTask 的任务优先级调整为与自己相同的优先级，即 4，然后任务 HighTask 进入阻塞态等待互斥信号量有效。

14.9 互斥信号量操作实验

14.9.1 实验程序设计

1、实验目的

学习使用互斥信号量，并且观察互斥信号量是否可以解决或者缓解优先级翻转。

2、实验设计

本实验在“FreeRTOS 实验 14-3 FreeRTOS 优先级翻转实验”的基础上完成，只是将其中的二值信号量更换为互斥信号量，其他部分完全相同。

3、实验工程

FreeRTOS 实验 14-4 FreeRTOS 互斥信号量操作实验。

4、实验程序与分析

本实验是在实验“FreeRTOS 实验 14-3 FreeRTOS 优先级翻转实验”的基础上修改的，除了任务函数以外其他的部分都相同。

● **任务函数**

```
//开始任务任务函数
void start_task(void *pvParameters)
{
    taskENTER_CRITICAL();           //进入临界区

    //创建互斥信号量
    MutexSemaphore=xSemaphoreCreateMutex();                    (1)

    //创建高优先级任务
    xTaskCreate((TaskFunction_t     )high_task,
                (const char*        )"high_task",
                (uint16_t           )HIGH_STK_SIZE,
                (void*              )NULL,
                (UBaseType_t        )HIGH_TASK_PRIO,
                (TaskHandle_t*      )&HighTask_Handler);
    //创建中等优先级任务
    xTaskCreate((TaskFunction_t     )middle_task,
                (const char*        )"middle_task",
                (uint16_t           )MIDDLE_STK_SIZE,
                (void*              )NULL,
                (UBaseType_t        )MIDDLE_TASK_PRIO,
                (TaskHandle_t*      )&MiddleTask_Handler);
    //创建低优先级任务
    xTaskCreate((TaskFunction_t     )low_task,
                (const char*        )"low_task",
                (uint16_t           )LOW_STK_SIZE,
                (void*              )NULL,
                (UBaseType_t        )LOW_TASK_PRIO,
                (TaskHandle_t*      )&LowTask_Handler);
    vTaskDelete(StartTask_Handler);     //删除开始任务
    taskEXIT_CRITICAL();            //退出临界区
}

//高优先级任务的任务函数
void high_task(void *pvParameters)
{
    u8 num;
```

```
    POINT_COLOR = BLACK;
    LCD_DrawRectangle(5,110,115,314);  //画一个矩形
    LCD_DrawLine(5,130,115,130);       //画线
    POINT_COLOR = BLUE;
    LCD_ShowString(6,111,110,16,16,"High Task");

    while(1)
    {
        vTaskDelay(500);  //延时 500ms，也就是 500 个时钟节拍
        num++;
        printf("high task Pend Sem\r\n");
        xSemaphoreTake(MutexSemaphore,portMAX_DELAY);   //获取互斥信号量        (2)
        printf("high task Running!\r\n");
        LCD_Fill(6,131,114,313,lcd_discolor[num%14]);           //填充区域
        LED1=!LED1;
        xSemaphoreGive(MutexSemaphore);                         //释放信号量        (3)
        vTaskDelay(500);  //延时 500ms，也就是 500 个时钟节拍
    }
}

//中等优先级任务的任务函数
void middle_task(void *pvParameters)
{
    u8 num;

    POINT_COLOR = BLACK;
    LCD_DrawRectangle(125,110,234,314);    //画一个矩形
    LCD_DrawLine(125,130,234,130);         //画线
    POINT_COLOR = BLUE;
    LCD_ShowString(126,111,110,16,16,"Middle Task");
    while(1)
    {
        num++;
        printf("middle task Running!\r\n");
        LCD_Fill(126,131,233,313,lcd_discolor[13-num%14]); //填充区域
        LED0=!LED0;
        vTaskDelay(1000); //延时 1s，也就是 1000 个时钟节拍
    }
}

//低优先级任务的任务函数
void low_task(void *pvParameters)
{
```

```
    static u32 times;

    while(1)
    {
        xSemaphoreTake(MutexSemaphore,portMAX_DELAY);   //获取互斥信号量          (4)
        printf("low task Running!\r\n");
        for(times=0;times<20000000;times++) //模拟低优先级任务占用互斥信号量      (5)
        {
            taskYIELD();                    //发起任务调度
        }
        xSemaphoreGive(MutexSemaphore);   //释放互斥信号量                      (6)
        vTaskDelay(1000); //延时 1s，也就是 1000 个时钟节拍
    }
}
```

(1)、调用函数 xSemaphoreCreateMutex()创建互斥信号量 MutexSemaphore。
(2)、任务 high_task 获取互斥信号量。
(3)、互斥信号量使用完成以后一定要释放！
(4)、任务 low_task 获取互斥信号量 MutexSemaphore，阻塞时间为 portMAX_DELAY。
(5)、模拟任务 low_task 长时间占用互斥信号量 portMAX_DELAY。
(6)、任务 low_task 使用完互斥信号量，释放！

14.9.2 程序运行结果分析

编译并下载实验代码到开发板中，打开串口调试助手，串口调试助手如图 14.9.2.1 所示：

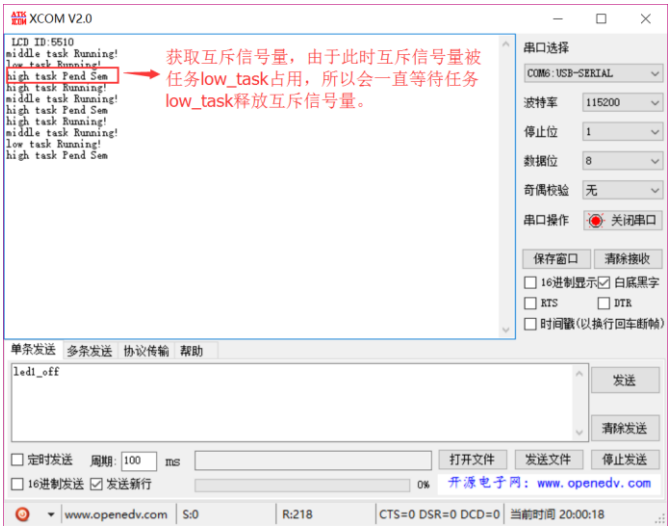


图 14.9.2.1 串口调试助手

为了方便分析，我们将串口调试助手中的数据复制下来，如下：

```
LCD ID:5510
middle task Running!                    (1)
low task Running!                       (2)
high task Pend Sem                      (3)
high task Running!                      (4)
```

```
middle task Running!
high task Pend Sem
high task Running!
middle task Running!
low task Running!
```

(1)、middle_task 任务运行。

(2)、low_task 获得互斥信号量运行。

(3)、high_task 请求信号量，在这里会等待一段时间，等待 low_task 任务释放互斥信号量。但是 middle_task 不会运行。因为由于 low_task 正在使用互斥信号量，所以 low_task 任务优先级暂时提升到了与 high_task 相同的优先级，这个优先级比任务 middle_task 高，所以 middle_task 任务不能再打断 low_task 任务的运行了！

(4)、high_task 任务获得互斥信号量而运行。

从上面的分析可以看出互斥信号量有效的抑制了优先级翻转现象的发生。

14.10 递归互斥信号量

14.10.1 递归互斥信号量简介

递归互斥信号量可以看作是一个特殊的互斥信号量，已经获取了互斥信号量的任务就不能再次获取这个互斥信号量，但是递归互斥信号量不同，已经获取了递归互斥信号量的任务可以再次获取这个递归互斥信号量，而且次数不限！一个任务使用函数 xSemaphoreTakeRecursive()成功的获取了多少次递归互斥信号量就得使用函数 xSemaphoreGiveRecursive()释放多少次！比如某个任务成功的获取了 5 次递归信号量，那么这个任务也得同样的释放 5 次递归信号量。

递归互斥信号量也有优先级继承的机制，所以当任务使用完递归互斥信号量以后一定要记得释放。同互斥信号量一样，递归互斥信号量不能用在中断服务函数中。

● 由于优先级继承的存在，就限定了递归互斥信号量只能用在任务中，不能用在中断服务函数中！

● 中断服务函数不能设置阻塞时间。

要使用递归互斥信号量的话宏 configUSE_RECURSIVE_MUTEXES 必须为 1！

14.10.2 创建互斥信号量

FreeRTOS 提供了两个互斥信号量创建函数，如表 14.10.2.1 所示：

函数	描述
xSemaphoreCreateRecursiveMutex()	使用动态方法创建递归互斥信号量。
xSemaphoreCreateRecursiveMutexStatic()	使用静态方法创建递归互斥信号量。

表 14.10.2.1 创建递归互斥信号量

1、函数 xSemaphoreCreateRecursiveMutex()

此函数用于创建一个递归互斥信号量，所需要的内存通过动态内存管理方法分配。此函数本质是一个宏，真正完成信号量创建的是函数 xQueueCreateMutex ()，此函数原型如下：

```
SemaphoreHandle_t    xSemaphoreCreateRecursiveMutex( void )
```

参数：

无。

返回值：

NULL: 互斥信号量创建失败。
其他值: 创建成功的互斥信号量的句柄。

2、函数 xSemaphoreCreateRecursiveMutexStatic()

此函数也是创建递归互斥信号量的，只不过使用此函数创建递归互斥信号量的话信号量所需要的 RAM 需要由用户来分配，此函数是个宏，具体创建过程是通过函数 xQueueCreateMutexStatic ()来完成的，函数原型如下：

```
SemaphoreHandle_t xSemaphoreCreateRecursiveMutexStatic( StaticSemaphore_t *pxMutexBuffer )
```

参数：

pxMutexBuffer：此参数指向一个 StaticSemaphore_t 类型的变量，用来保存信号量结构体。

返回值：

NULL: 互斥信号量创建失败。
其他值: 创建成功的互斥信号量的句柄。

14.10.3 递归信号量创建过程分析

这里只分析动态创建互斥信号量函数 xSemaphoreCreateRecursiveMutex ()，此函数是个宏，定义如下：

```
#define    xSemaphoreCreateRecursiveMutex()
           xQueueCreateMutex( queueQUEUE_TYPE_RECURSIVE_MUTEX )
```

可以看出，真正干事的是函数 xQueueCreateMutex()，互斥信号量的创建也是用的这个函数，只是在创建递归互斥信号量的时候类型选择为 queueQUEUE_TYPE_RECURSIVE_MUTEX。具体的创建过程参考 14.8.3 小节。

14.10.4 释放递归互斥信号量

递归互斥信号量有专用的释放函数：xSemaphoreGiveRecursive()，此函数为宏，如下：

```
#define    xSemaphoreGiveRecursive( xMutex )    xQueueGiveMutexRecursive( ( xMutex ) )
```

函数的参数就是就是要释放的递归互斥信号量，真正的释放是由函数 xQueueGiveMutexRecursive()来完成的，此函数代码如下：

```
BaseType_t xQueueGiveMutexRecursive( QueueHandle_t xMutex )
{
    BaseType_t xReturn;
    Queue_t * const pxMutex = ( Queue_t * ) xMutex;

    configASSERT( pxMutex );

    //检查递归互斥信号量是不是被当前任务获取的，要释放递归互斥信号量的任务肯定是当
    //前正在运行的任务。 因为同互斥信号量一样，递归互斥信号量的获取和释放要在同一个
    //任务中完成！如果当前正在运行的任务不是递归互斥信号量的拥有者就不能释放！
```

```
if( pxMutex->pxMutexHolder == ( void * ) xTaskGetCurrentTaskHandle() )          (1)
{
    traceGIVE_MUTEX_RECURSIVE( pxMutex );

    ( pxMutex->u.uxRecursiveCallCount )--;                                      (2)
    if( pxMutex->u.uxRecursiveCallCount == ( UBaseType_t ) 0 )                  (3)
    {

        ( void ) xQueueGenericSend( pxMutex, NULL, \                            (4)
          queueMUTEX_GIVE_BLOCK_TIME, queueSEND_TO_BACK );
    }
    else
    {
        mtCOVERAGE_TEST_MARKER();
    }
        xReturn = pdPASS;                                                       (5)
    }
    else
    {
        xReturn = pdFAIL;                                                       (6)
        traceGIVE_MUTEX_RECURSIVE_FAILED( pxMutex );
    }

    return xReturn;
}
```

(1)、哪个任务获取到的递归互斥信号量，哪个任务就释放！要释放递归互斥信号量的任务肯定是当前正在运行的任务。检查这个任务是不是递归互斥信号量的拥有者，如果不是的话就不能完成释放。

(2)、uxRecursiveCallCount 减一，uxRecursiveCallCount 用来记录递归信号量被获取的次数。由于递归互斥信号量可以被一个任务多次获取，因此在释放的时候也要多次释放，但是只有在最后一次释放的时候才会调用函数 xQueueGenericSend()完成释放过程，其他的时候只是简单的将 uxRecursiveCallCount 减一即可。

(3)、当 uxRecursiveCallCount 为 0 的时候说明是最后一次释放了。

(4)、如果是最后一次释放的话就调用函数 xQueueGenericSend()完成真正的释放过程。阻塞时间是 queueMUTEX_GIVE_BLOCK_TIME，宏 queueMUTEX_GIVE_BLOCK_TIME 为 0。

(5)、递归互斥信号量释放成功，返回 pdPASS。

(6)、递归互斥信号量释放未成功，返回 pdFAIL。

由于递归互斥信号量可以被一个任务重复的获取，因此在释放的时候也要释放多次，但是只有在最后一次释放的时候才会调用函数 xQueueGenericSend()完成真正的释放。其他释放的话只是简单的将 uxRecursiveCallCount 减一。

14.10.5 获取递归互斥信号量

递归互斥信号量的获取使用函数 xSemaphoreTakeRecursive()，此函数是个宏，定义如下：

```
#define xSemaphoreTakeRecursive( xMutex, xBlockTime )
        xQueueTakeMutexRecursive( ( xMutex ), ( xBlockTime ) )
```

函数第一个参数是要获取的递归互斥信号量句柄，第二个参数是阻塞时间。真正的获取过程是由函数 xQueueTakeMutexRecursive()来完成的，此函数如下：

```
BaseType_t xQueueTakeMutexRecursive( QueueHandle_t  xMutex,         //要获取的信号量
                                     TickType_t     xTicksToWait )//阻塞时间
{
    BaseType_t xReturn;
    Queue_t * const pxMutex = ( Queue_t * ) xMutex;
    configASSERT( pxMutex );

    traceTAKE_MUTEX_RECURSIVE( pxMutex );

    if( pxMutex->pxMutexHolder == ( void * ) xTaskGetCurrentTaskHandle() )          (1)
    {
        ( pxMutex->u.uxRecursiveCallCount )++;                                      (2)
        xReturn = pdPASS;
    }
    else
    {
        xReturn = xQueueGenericReceive( pxMutex, NULL, xTicksToWait, pdFALSE );     (3)
        if( xReturn != pdFAIL )
        {
            ( pxMutex->u.uxRecursiveCallCount )++;                                  (4)
        }
        else
        {
                traceTAKE_MUTEX_RECURSIVE_FAILED( pxMutex );
        }
    }

        return xReturn;
    }
```

(1)、判断当前要获取递归互斥信号量的任务是不是已经是递归互斥信号量的拥有者。通过这一步就可以判断出当前任务是第一次获取递归互斥信号量还是重复获取。

(2)、如果当前任务已经是递归互斥信号量的拥有者，那就说明任务已经获取了递归互斥信号量，本次是重复获取递归互斥信号量，那么就简单的将 uxRecursiveCallCount 加一，然后返回 pdPASS 表示获取成功。

(3)、如果任务是第一次获取递归互斥信号量的话就需要调用函数 xQueueGenericReceive()完成真正的获取过程。

(4)、第一次获取递归互斥信号量成功以后将 uxRecursiveCallCount 加一。

14.10.6 递归互斥信号量使用示例

互斥信号量使用很简单，就不做专门的实验了，FreeRTOS 官方提供了一个简单的示例，大家可以参考一下，示例如下：

```
SemaphoreHandle_t RecursiveMutex;        //递归互斥信号量句柄

//某个任务中创建一个递归互斥信号量
void vATask( void * pvParameters )
{
    //没有创建创建递归互斥信号量之前不要使用！
    RecursiveMutex = xSemaphoreCreateRecursiveMutex();    //创建递归互斥信号量
    for( ;; )
    {
        /************任务代码**************/
    }
}

//任务调用的使用递归互斥信号量的功能函数。
void vAFunction( void )
{
    /**********其他处理代码*****************/
    if( xMutex != NULL )
    {
        //获取递归互斥信号量，阻塞时间为 10 个节拍
        if( xSemaphoreTakeRecursive( RecursiveMutex, 10 ) == pdTRUE )
        {
            /***********其他处理过程*************/

            //这里为了演示，所以是顺序的获取递归互斥信号量，但是在实际的代码中肯定
            //不是这么顺序的获取的，真正的代码中是混合着其他程序调用的。
            xSemaphoreTakeRecursive( RecursiveMutex, ( TickType_t ) 10 );
            xSemaphoreTakeRecursive( RecursiveMutex, ( TickType_t ) 10 );

            //任务获取了三次递归互斥信号量，所以就得释放三次！
            xSemaphoreGiveRecursive( RecursiveMutex);
            xSemaphoreGiveRecursive( RecursiveMutex);
            xSemaphoreGiveRecursive( RecursiveMutex);

            //递归互斥信号量释放完成，可以被其他任务获取了
        }
        else
        {
            /**********递归互斥信号量获取失败***********/
        }
```

```
    }
}
```

第十五章 FreeRTOS 软件定时器

定时器可以说是每个 MCU 都有的外设，有的 MCU 其定时器功能异常强大，比如提供 PWM、输入捕获等功能。但是最常用的还是定时器最基础的功能——定时，通过定时器来完成需要周期性处理的事务。MCU 自带的定时器属于硬件定时器，不同的 MCU 其硬件定时器数量不同，因为要考虑成本的问题。FreeRTOS 也提供了定时器功能，不过是软件定时器，软件定时器的精度肯定没有硬件定时器那么高，但是对于普通的精度要求不高的周期性处理的任务来说够了。当 MCU 的硬件定时器不够的时候就可以考虑使用 FreeRTOS 的软件定时器，本章就来学习一下 FreeRTOS 的软件定时器，本章分为如下几部分：

15.1 软件定时器简介
15.2 定时器服务/Daemon 任务
15.3 单次定时器和周期定时器
15.4 复位软件定时器
15.5 创建软件定时器
15.6 开启软件定时器
15.7 停止软件定时器
15.8 软件定时器实验

15.1 软件定时器简介

1、软件定时器概述

软件定时器允许设置一段时间，当设置的时间到达之后就执行指定的功能函数，被定时器调用的这个功能函数叫做定时器的回调函数。回调函数的两次执行间隔叫做定时器的定时周期，简而言之，当定时器的定时周期到了以后就会执行回调函数。

2、编写回调函数的注意事项

软件定时器的回调函数是在定时器服务任务中执行的，所以一定不能在回调函数中调用任何会阻塞任务的 API 函数！比如，定时器回调函数中千万不能调用 vTaskDelay()、vTaskDelayUnti()，还有一些访问队列或者信号量的非零阻塞时间的 API 函数也不能调用。

15.2 定时器服务/Daemon 任务

15.2.1 定时器服务任务与队列

定时器是一个可选的、不属于 FreeRTOS 内核的功能，它是由定时器服务(或 Daemon)任务来提供的。FreeRTOS 提供了很多定时器有关的 API 函数，这些 API 函数大多都使用 FreeRTOS 的队列发送命令给定时器服务任务。这个队列叫做定时器命令队列。定时器命令队列是提供给 FreeRTOS 的软件定时器使用的，用户不能直接访问！图 15.2.1 描述了这个过程：

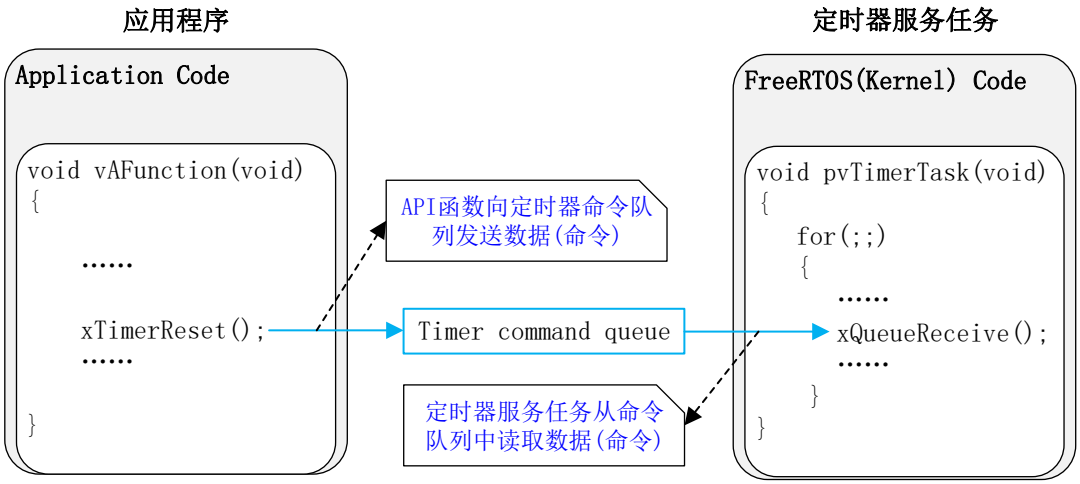


图 15.2.1 定时器命令队列操作

图 15.2.1 左侧部分属于用户应用程序的一部分，并且会在某个用户创建的用户任务中调用。图中右侧部分是定时器服务任务的任务函数，定时器命令队列将用户应用任务和定时器服务任务连接在一起。在这个例子中，应用程序调用了函数 xTimerReset()，结果就是复位命令会被发送到定时器命令队列中，定时器服务任务会处理这个命令。应用程序是通过函数 xTimerReset()间接的向定时器命令队列发送了复位命令，并不是直接调用类似 xQueueSend()这样的队列操作函数发送的。

15.2.2 定时器相关配置

上一小节我们知道了软件定时器有一个定时器服务任务和定时器命令队列，这两个东西肯定是要配置的，配置方法和我们前面讲解的 FreeRTOSCofig.h 一样，而且相关的配置也是放到文件 FreeRTOSConfig.h 中的，涉及到的配置如下：

1、configUSE_TIMERS

如果要使用软件定时器的话宏 configUSE_TIMERS 一定要设置为 1，当设置为 1 的话定时器服务任务就会在启动 FreeRTOS 调度器的时候自动创建。

2、configTIMER_TASK_PRIORITY

设置软件定时器服务任务的任务优先级，可以为 0~(configMAX_PRIORITIES-1)。优先级一定要根据实际的应用要求来设置。如果定时器服务任务的优先级设置的高的话，定时器命令队列中的命令和定时器回调函数就会及时的得到处理。

3、configTIMER_QUEUE_LENGTH

此宏用来设置定时器命令队列的队列长度。

4、configTIMER_TASK_STACK_DEPTH

此宏用来设置定时器服务任务的任务堆栈大小，单位为字，不是字节！，对于 STM32 来说一个字是 4 字节。由于定时器服务任务中会执行定时器的回调函数，因此任务堆栈的大小一定要根据定时器的回调函数来设置。

15.3 单次定时器和周期定时器

软件定时器分两种：单次定时器和周期定时器，单次定时器的话定时器回调函数就执行一次，比如定时 1s，当定时时间到了以后就会执行一次回调函数，然后定时器就会停止运行。对于单次定时器我们可以再次手动重新启动(调用相应的 API 函数即可)，但是单次定时器不能自动重启。相反的，周期定时器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，这样回调函数就会周期性的执行。图 15.3.1 描述了单次定时器和周期定时器的不同：

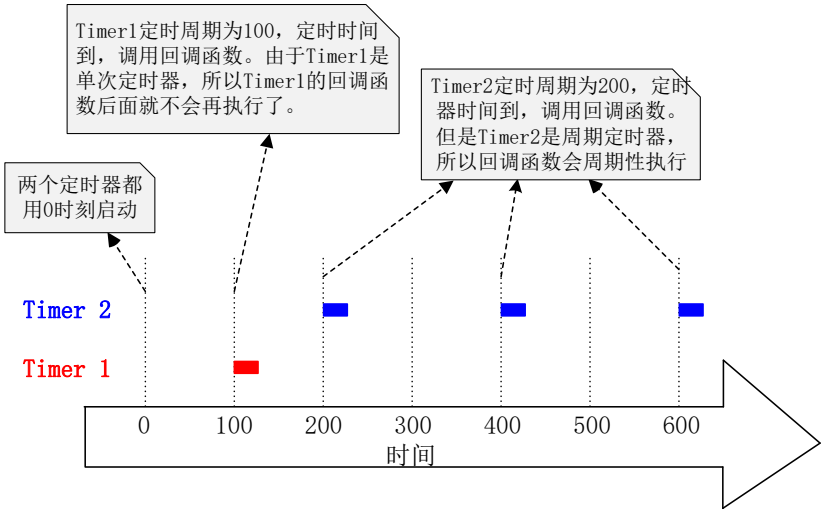


图 15.3.1 单次定时器和周期定时器

图中 Timer1 为单次定时器，定时器周期为 100，Timer2 为周期定时器，定时器周期为 200。

15.4 复位软件定时器

有时候我们可能会在定时器正在运行的时候需要复位软件定时器，复位软件定时器的话会

重新计算定时周期到达的时间点，这个新的时间点是相对于复位定时器的那个时刻计算的，并不是第一次启动软件定时器的那个时间点。图 15.4.1 演示了这个过程，Timer1 是单次定时器，定时周期是 5s：

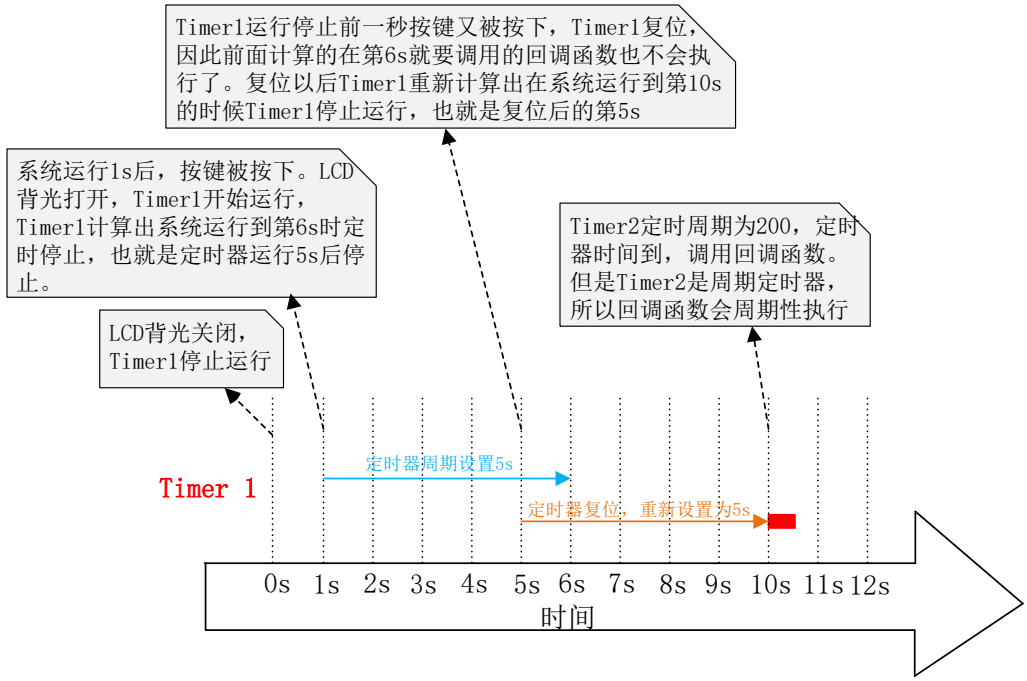


图 15.4.1 定时器复位

在图 15.4.1 中我们展示了定时器复位过程，这是一个通过按键打开 LCD 背光的例子，我们假定当唤醒键被按下的时候应用程序打开 LCD 背光，当 LCD 背光点亮以后如果 5s 之内唤醒键没有再次按下就自动熄灭。如果在这 5s 之内唤醒键被按下了，LCD 背光就从按下的这个时刻起再亮 5s。

FreeRTOS 提供了两个 API 函数来完成软件定时器的复位，如表 15.4.1 所示：

函数	描述
xTimerReset()	复位软件定时器，用在任务中。
xTimerResetFromISR()	复位软件定时器，用在中断服务函数中

表 15.4.1 复位软件定时器

1、函数 xTimerReset()

复位一个软件定时器，此函数只能用在任务中，不能用于中断服务函数！此函数是一个宏，真正执行的是函数 xTimerGenericCommand()，函数原型如下：

```
BaseType_t xTimerReset( TimerHandle_t    xTimer,
                        TickType_t       xTicksToWait )
```

参数：

xTimer： 要复位的软件定时器的句柄。

xTicksToWait： 设置阻塞时间，调用函数 xTimerReset ()开启软件定时器其实就是向定时器命令队列发送一条 tmrCOMMAND_RESET 命令，既然是向队列发送消息，那肯定会涉及到入队阻塞时间的设置。

返回值：

pdPASS: 软件定时器复位成功，其实就是命令发送成功。

pdFAIL: 软件定时器复位失败，命令发送失败。

2、函数 xTimerResetFromISR()

此函数是 xTimerReset()的中断版本，此函数用于中断服务函数中！此函数是一个宏，真正执行的是函数 xTimerGenericCommand()，函数原型如下：

```
BaseType_t xTimerResetFromISR( TimerHandle_t     xTimer,
                               BaseType_t *      pxHigherPriorityTaskWoken );
```

参数：

xTimer： 要复位的软件定时器的句柄。

pxHigherPriorityTaskWoken： 记退出此函数以后是否进行任务切换，这个变量的值函数会自动设置的，用户不用进行设置，用户只需要提供一个变量来保存这个值就行了。当此值为 pdTRUE 的时候在退出中断服务函数之前一定要进行一次任务切换。

返回值：

pdPASS: 软件定时器复位成功，其实就是命令发送成功。

pdFAIL: 软件定时器复位失败，命令发送失败。

15.5 创建软件定时器

使用软件定时器之前要先创建软件定时器，软件定时器创建函数如表 15.5.1 所示：

函数	描述
xTimerCreate()	使用动态方法创建软件定时器。
xTimerCreateStatic()	使用静态方法创建软件定时器。

表 15.5.1 创建软件定时器

1、函数 xTiemrCreate()

此函数用于创建一个软件定时器，所需要的内存通过动态内存管理方法分配。新创建的软件定时器处于休眠状态，也就是未运行的。函数 xTimerStart()、xTimerReset()、xTimerStartFromISR()、xTimerResetFromISR()、xTimerChangePeriod()和 xTimerChangePeriodFromISR()可以使新创建的定时器进入活动状态，此函数的原型如下：

```
TimerHandle_t xTimerCreate( const char * const    pcTimerName,
                            TickType_t            xTimerPeriodInTicks,
                            UBaseType_t           uxAutoReload,
                            void *                pvTimerID,
                            TimerCallbackFunction_t pxCallbackFunction )
```

参数：

pcTimerName： 软件定时器名字，名字是一串字符串，用于调试使用。

xTimerPeriodInTicks： 软件定时器的定时器周期，单位是时钟节拍数。可以借助

portTICK_PERIOD_MS 将 ms 单位转换为时钟节拍数。举个例子，定时器的周期为 100 个时钟节拍的话，那么 xTimerPeriodInTicks 就为 100，当定时器周期为 500ms 的时候 xTimerPeriodInTicks 就可以设置为(500/ portTICK_PERIOD_MS)。

uxAutoReload： 设置定时器模式，单次定时器还是周期定时器？当此参数为 pdTRUE 的时候表示创建的是周期定时器。如果为 pdFALSE 的话表示创建的是单次定时器。

pvTimerID： 定时器 ID 号，一般情况下每个定时器都有一个回调函数，当定时器定时周期到了以后就会执行这个回调函数。但是 FreeRTOS 也支持多个定时器共用同一个回调函数，在回调函数中根据定时器的 ID 号来处理不同的定时器。

pxCallbackFunction： 定时器回调函数，当定时器定时周期到了以后就会调用这个函数。

返回值：

NULL: 软件定时器创建失败。

其他值: 创建成功的软件定时器句柄。

2、函数 xTimerCreateStatic()

此函数用于创建一个软件定时器，所需要的内存需要用户自行分配。新创建的软件定时器处于休眠状态，也就是未运行的。函数 xTimerStart()、xTimerReset()、xTimerStartFromISR()、xTimerResetFromISR()、xTimerChangePeriod()和 xTimerChangePeriodFromISR()可以使新创建的定时器进入活动状态，此函数的原型如下：

```
TimerHandle_t xTimerCreateStatic(const char * const    pcTimerName,
                                 TickType_t            xTimerPeriodInTicks,
                                 UBaseType_t           uxAutoReload,
                                 void *                pvTimerID,
                                 TimerCallbackFunction_t pxCallbackFunction,
                                 StaticTimer_t *       pxTimerBuffer )
```

参数：

pcTimerName： 软件定时器名字，名字是一串字符串，用于调试使用。

xTimerPeriodInTicks： 软件定时器的定时器周期，单位是时钟节拍数。可以借助 portTICK_PERIOD_MS 将 ms 单位转换为时钟节拍数。举个例子，定时器的周期为 100 个时钟节拍的话，那么 xTimerPeriodInTicks 就为 100，当定时器周期为 500ms 的时候 xTimerPeriodInTicks 就可以设置为(500/ portTICK_PERIOD_MS)。

uxAutoReload： 设置定时器模式，单次定时器还是周期定时器？当此参数为 pdTRUE 的时候表示创建的是周期定时器。如果为 pdFALSE 的话表示创建的是单次定时器。

pvTimerID： 定时器 ID 号，一般情况下每个定时器都有一个回调函数，当定时器定时周期到了以后就会执行这个回调函数。当时 FreeRTOS 也支持多个定时器共用同一个回调函数，在回调函数中根据定时器的 ID 号来处理不同的定时器。

pxCallbackFunction： 定时器回调函数，当定时器定时周期到了以后就会调用这个函数。
pxTimerBuffer： 参数指向一个 StaticTimer_t 类型的变量，用来保存定时器结构体。

返回值：

NULL: 软件定时器创建失败。
其他值: 创建成功的软件定时器句柄。

15.6 开启软件定时器

如果软件定时器停止运行的话可以使用 FreeRTOS 提供的两个开启函数来重新启动软件定时器，这两个函数表 15.6.1 所示：

函数	描述
xTimerStart()	开启软件定时器，用于任务中。
xTimerStartFromISR()	开启软件定时器，用于中断中。

表 15.6.1 开启软件定时器

1、函数 xTimerStart()

启动软件定时器，函数 xTimerStartFromISR()是这个函数的中断版本，可以用在中断服务函数中。如果软件定时器没有运行的话调用函数 xTimerStart()就会计算定时器到期时间，如果软件定时器正在运行的话调用函数 xTimerStart()的结果和 xTimerReset()一样。此函数是个宏，真正执行的是函数 xTimerGenericCommand，函数原型如下：

```
BaseType_t xTimerStart( TimerHandle_t    xTimer,
                        TickType_t       xTicksToWait )
```

参数：

xTimer： 要开启的软件定时器的句柄。
xTicksToWait： 设置阻塞时间，调用函数 xTimerStart()开启软件定时器其实就是向定时器命令队列发送一条 tmrCOMMAND_START 命令，既然是向队列发送消息，那肯定会涉及到入队阻塞时间的设置。

返回值：

pdPASS: 软件定时器开启成功，其实就是命令发送成功。
pdFAIL: 软件定时器开启失败，命令发送失败。

2、函数 xTimerStartFromISR()

此函数是函数 xTimerStart()的中断版本，用在中断服务函数中，此函数是一个宏，真正执行的是函数 xTimerGenericCommand()，此函数原型如下：

```
BaseType_t xTimerStartFromISR( TimerHandle_t    xTimer,
                               BaseType_t *     pxHigherPriorityTaskWoken );
```

参数：

xTimer： 要开启的软件定时器的句柄。
pxHigherPriorityTaskWoken： 标记退出此函数以后是否进行任务切换，这个变量的值函数会

自动设置的，用户不用进行设置，用户只需要提供一个变量来保存这个值就行了。当此值为 pdTRUE 的时候在退出中断服务函数之前一定要进行一次任务切换。

返回值：

pdPASS: 软件定时器开启成功，其实就是命令发送成功。
pdFAIL: 软件定时器开启失败，命令发送失败。

15.7 停止软件定时器

既然有开启软件定时器的 API 函数，那么肯定也有停止软件定时器的函数，FreeRTOS 也提供了两个用于停止软件定时器的 API 函数，如表 15.7.1 所示：

函数	描述
xTimerStop()	停止软件定时器，用于任务中。
xTimerStopFromISR()	停止软件定时器，用于中断服务函数中。

表 15.7.1 关闭软件定时器

1、函数 xTimerStop()

此函数用于停止一个软件定时器，此函数用于任务中，不能用在中断服务函数中！此函数是一个宏，真正调用的是函数 xTimerGenericCommand()，函数原型如下：

```
BaseType_t xTimerStop ( TimerHandle_t    xTimer,
                        TickType_t       xTicksToWait )
```

参数：

xTimer： 要停止的软件定时器的句柄。
xTicksToWait： 设置阻塞时间，调用函数 xTimerStop()停止软件定时器其实就是向定时器命令队列发送一条 tmrCOMMAND_STOP 命令，既然是向队列发送消息，那肯定会涉及到入队阻塞时间的设置。

返回值：

pdPASS: 软件定时器停止成功，其实就是命令发送成功。
pdFAIL: 软件定时器停止失败，命令发送失败。

1、函数 xTimerStopFromISR()

此函数是 xTimerStop()的中断版本，此函数用于中断服务函数中！此函数是一个宏，真正执行的是函数 xTimerGenericCommand()，函数原型如下：

```
BaseType_t xTimerStopFromISR( TimerHandle_t     xTimer,
                              BaseType_t *      pxHigherPriorityTaskWoken );
```

参数：

xTimer： 要停止的软件定时器句柄。
pxHigherPriorityTaskWoken： 标记退出此函数以后是否进行任务切换，这个变量的值函数会自动设置的，用户不用进行设置，用户只需要提供一个变量来

保存这个值就行了。当此值为 pdTRUE 的时候在退出中断服务函数之前一定要进行一次任务切换。

返回值：

pdPASS: 软件定时器停止成功，其实就是命令发送成功。

pdFAIL: 软件定时器停止失败，命令发送失败。

15.8 软件定时器实验

15.8.1 实验程序设计

1、实验目的

学习 FreeROTS 软件定时器的使用，包括软件定时器的创建、开启和停止。

2、实验设计

本实验设计两个任务：start_task 和 timercontrol_task 这两个任务的任务功能如下：

start_task：用来创建任务 timercontrol_task()和两个软件定时器。

timercontrol_task：控制两个软件定时器的开启和停止。

实验中还创建了两个软件定时器：AutoReloadTimer_Handle 和 OneShotTimer_Handle，这两个定时器分别为周期定时器和单次定时器。定时器 AutoReloadTimer_Handle 的定时器周期为 1000 个时钟节拍(1s)，定时器 OneShotTimer_Handle 的定时器周期为 2000 个时钟节拍(2s)。

3、实验工程

FreeRTOS 实验 15-1 FreeRTOS 软件定时器实验。

4、实验程序与分析

●任务设置

```
#define START_TASK_PRIO                 1          //任务优先级
#define START_STK_SIZE                  256        //任务堆栈大小
TaskHandle_t StartTask_Handler;                    //任务句柄
void start_task(void *pvParameters);               //任务函数

#define TIMERCONTROL_TASK_PRIO          2          //任务优先级
#define TIMERCONTROL_STK_SIZE           256        //任务堆栈大小
TaskHandle_t TimerControlTask_Handler;             //任务句柄
void timercontrol_task(void *pvParameters);        //任务函数

//////////////////////////////////////////////////////
TimerHandle_t   AutoReloadTimer_Handle;            //周期定时器句柄
TimerHandle_t   OneShotTimer_Handle;               //单次定时器句柄

void AutoReloadCallback(TimerHandle_t xTimer);     //周期定时器回调函数
void OneShotCallback(TimerHandle_t xTimer);        //单次定时器回调函数
```

```
//LCD 刷屏时使用的颜色
int lcd_discolor[14]={  WHITE,      BLACK,      BLUE,   BRED,
                        GRED,       GBLUE,      RED,    MAGENTA,
                        GREEN,      CYAN,   YELLOW,     BROWN,
                        BRRED,      GRAY };
```

● **main()函数**

```
int main(void)
{
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(360,25,2,8);   //设置时钟,180Mhz
    delay_init(180);                //初始化延时函数
    uart_init(115200);              //初始化串口
    LED_Init();                     //初始化 LED
    KEY_Init();                     //初始化按键
    PCF8574_Init();                 //初始化 PCF8574
    SDRAM_Init();                   //初始化 SDRAM
    LCD_Init();                     //初始化 LCD
    my_mem_init(SRAMIN);            //初始化内部内存池

    POINT_COLOR = RED;
    LCD_ShowString(30,10,200,16,16,"Apollo STM32F4/F7");
    LCD_ShowString(30,30,200,16,16,"FreeRTOS Examp 15-1");
    LCD_ShowString(30,50,200,16,16,"KEY_UP:Start Tmr1");
    LCD_ShowString(30,70,200,16,16,"KEY0:Start Tmr2");
    LCD_ShowString(30,90,200,16,16,"KEY1:Stop Tmr1 and Tmr2");

    LCD_DrawLine(0,108,239,108);    //画线
    LCD_DrawLine(119,108,119,319);  //画线

    POINT_COLOR = BLACK;
    LCD_DrawRectangle(5,110,115,314); //画一个矩形
    LCD_DrawLine(5,130,115,130);    //画线

    LCD_DrawRectangle(125,110,234,314); //画一个矩形
    LCD_DrawLine(125,130,234,130);  //画线
    POINT_COLOR = BLUE;
    LCD_ShowString(6,111,110,16,16,"AutoTim:000");
    LCD_ShowString(126,111,110,16,16,"OneTim: 000");

    //创建开始任务
    xTaskCreate((TaskFunction_t     )start_task,            //任务函数
                (const char*        )"start_task",          //任务名称
                (uint16_t           )START_STK_SIZE,        //任务堆栈大小
```

```
                (void*          )NULL,              //传递给任务函数的参数
                (UBaseType_t    )START_TASK_PRIO,   //任务优先级
                (TaskHandle_t*  )&StartTask_Handler); //任务句柄
    vTaskStartScheduler();                          //开启任务调度
}
```

● 任务函数

```
//开始任务任务函数
void start_task(void *pvParameters)
{
    taskENTER_CRITICAL();           //进入临界区
    //创建软件周期定时器 ，周期定时器，周期 1s(1000 个时钟节拍)，周期模式
    AutoReloadTimer_Handle=xTimerCreate((const char*            )"AutoReloadTimer",     (1)
                                        (TickType_t             )1000,
                                        (UBaseType_t            )pdTRUE,
                                        (void*                  )1,
                                        (TimerCallbackFunction_t)AutoReloadCallback);

    //创建单次定时器，单次定时器，周期 2s(2000 个时钟节拍)，单次模式
    OneShotTimer_Handle=xTimerCreate((const char*               )"OneShotTimer",        (2)
                                     (TickType_t                )2000,
                                     (UBaseType_t               )pdFALSE,
                                     (void*                     )2,
                                     (TimerCallbackFunction_t)OneShotCallback);
    //创建 TASK1 任务
    xTaskCreate((TaskFunction_t     )timercontrol_task,
                (const char*        )"timercontrol_task",
                (uint16_t           )TIMERCONTROL_STK_SIZE,
                (void*              )NULL,
                (UBaseType_t        )TIMERCONTROL_TASK_PRIO,
                (TaskHandle_t*      )&TimerControlTask_Handler);
    vTaskDelete(StartTask_Handler);     //删除开始任务
    taskEXIT_CRITICAL();                //退出临界区
}

//TimerControl 的任务函数
void timercontrol_task(void *p_arg)
{
    u8 key,num;
    while(1)
    {
        //只有两个定时器都创建成功了才能对其进行操作
        if((AutoReloadTimer_Handle!=NULL)&&(OneShotTimer_Handle!=NULL))
        {
```

```
            key = KEY_Scan(0);
            switch(key)
            {
                case WKUP_PRES:          //当 key_up 按下的话打开周期定时器
                    xTimerStart(AutoReloadTimer_Handle,0);      //开启周期定时器 (3)
                    printf("开启定时器 1\r\n");
                    break;
                case KEY0_PRES:          //当 key0 按下的话打开单次定时器
                    xTimerStart(OneShotTimer_Handle,0);         //开启单次定时器 (4)
                    printf("开启定时器 2\r\n");
                    break;
                case KEY1_PRES:          //当 key1 按下话就关闭定时器
                    xTimerStop(AutoReloadTimer_Handle,0);       //关闭周期定时器 (5)
                    xTimerStop(OneShotTimer_Handle,0);          //关闭单次定时器
                    printf("关闭定时器 1 和 2\r\n");
                    break;
            }
        }
        num++;
        if(num==50)  //每 500msLED0 闪烁一次
        {
            num=0;
            LED0=!LED0;
        }
        vTaskDelay(10); //延时 10ms，也就是 10 个时钟节拍
    }
}
```

(1)、调用函数 xTimerCreate()创建定时器 AutoReloadTimer_Handle，这是一个周期定时器，定时周期是 1000 个时钟节拍，在本例程中就是 1s。

(2)、调用函数 xTimerCreate()创建定时器 OneShotTimer_Handle，这是一个单次定时器，定时器周期为 2000 个时钟节拍，在本例程中就是 2s。

(3)、当 KEY_UP 按下以后调用函数 xTimerStart()开启周期定时器 AutoReloadTimer_Handle。

(4)、当 KEY0 按下以后调用函数 xTimerStart()开启单次定时器 OneShotTimer_Handle。

(5)、当 KEY1 按下以后就调用函数 xTimerStop()同时关闭定时器 AutoReloadTimer_Handle 和 OneShotTimer_Handle。

● **定时器回调函数**

```
//周期定时器的回调函数
void AutoReloadCallback(TimerHandle_t xTimer)
{
    static u8 tmr1_num=0;

    tmr1_num++;                                    //周期定时器执行次数加 1
    LCD_ShowxNum(70,111,tmr1_num,3,16,0x80);       //显示周期定时器的执行次数
```

```
    LCD_Fill(6,131,114,313,lcd_discolor[tmr1_num%14]);     //填充区域
}

//单次定时器的回调函数
void OneShotCallback(TimerHandle_t xTimer)
{
    static u8 tmr2_num = 0;

    tmr2_num++;                                            //周期定时器执行次数加 1
    LCD_ShowxNum(190,111,tmr2_num,3,16,0x80);              //显示单次定时器执行次数
    LCD_Fill(126,131,233,313,lcd_discolor[tmr2_num%14]); //填充区域
    LED1=!LED1;
    printf("定时器 2 运行结束\r\n");
}
```

15.8.2 程序运行结果分析

编译并下载实验代码到开发板中，默认情况下 LCD 显示如图 15.8.2.1 所示。

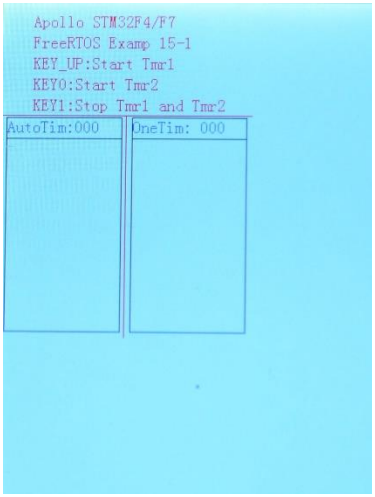


图 15.8.2.1 LCD 默认显示

当按下 KEY0 键以后定时器 OneShotTimer_Handle 开始运行，当定时器计时时间到了以后就会调用函数 OneShotCallback()，屏幕右侧区域的背景色会被刷新为其他颜色，背景颜色刷新完成以后定时器 OneShotTimer_Handle 就会停止运行。当按下 KEY_UP 键的话定时器 AutoReloadTimer_Handle 就会开始运行，定时器周期到了以后屏幕左侧区域的背景色会被刷新为其他颜色。由于定时器 AutoReloadTimer_Handle 是周期定时器，所以不会停止运行，除非按下 KEY1 键同时关闭定时器 AutoReloadTimer_Handle 和 OneShotTimer_Handle。

第十六章 FreeRTOS 事件标志组

前面我们学习了使用信号量来完成同步，但是使用信号量来同步的话任务只能与单个的事件或任务进行同步。有时候某个任务可能会需要与多个事件或任务进行同步，此时信号量就无能为力了。FreeRTOS 为此提供了一个可选的解决方法，那就是事件标志组。本章我们就来学习一下 FreeRTOS 中事件标志组的使用，本章分为如下几部分：

16.1 事件标志组简介
16.2 创建事件标志组
16.3 设置事件位
16.4 获取事件标志组值
16.5 等待指定的事件位
16.6 事件标志组实验

16.1 事件标志组简介

1、事件位(事件标志)

事件位用来表明某个事件是否发生，事件位通常用作事件标志，比如下面的几个例子：

● 当收到一条消息并且把这条消息处理掉以后就可以将某个位(标志)置 1，当队列中没有消息需要处理的时候就可以将这个位(标志)置 0。

● 当把队列中的消息通过网络发送输出以后就可以将某个位(标志)置 1，当没有数据需要从网络发送出去的话就将这个位(标志)置 0。

● 现在需要向网络中发送一个心跳信息，将某个位(标志)置 1。现在不需要向网络中发送心跳信息，这个位(标志)置 0。

2、事件组

一个事件组就是一组的事件位，事件组中的事件位通过位编号来访问，同样，以上面列出的三个例子为例：

● 事件标志组的 bit0 表示队列中的消息是否处理掉。

● 事件标志组的 bit1 表示是否有消息需要从网络中发送出去。

● 事件标志组的 bit2 表示现在是否需要向网络发送心跳信息。

3、事件标志组和事件位的数据类型

事件标志组的数据类型为 EventGroupHandle_t，当 configUSE_16_BIT_TICKS 为 1 的时候事件标志组可以存储 8 个事件位，当 configUSE_16_BIT_TICKS 为 0 的时候事件标志组存储 24 个事件位。

事件标志组中的所有事件位都存储在一个无符号的 EventBits_t 类型的变量中，EventBits_t 在 event_groups.h 中有如下定义：

```
typedef    TickType_t     EventBits_t;
```

数据类型 TickType_t 在文件 portmacro.h 中有如下定义：

```
#if( configUSE_16_BIT_TICKS == 1 )
    typedef uint16_t TickType_t;
    #define portMAX_DELAY ( TickType_t ) 0xffff
#else
    typedef uint32_t TickType_t;
    #define portMAX_DELAY ( TickType_t ) 0xffffffffUL
    #define portTICK_TYPE_IS_ATOMIC 1
#endif
```

可以看出当 configUSE_16_BIT_TICKS 为 0 的时候 TickType_t 是个 32 位的数据类型，因此 EventBits_t 也是个 32 位的数据类型。EventBits_t 类型的变量可以存储 24 个事件位，另外的那高 8 位有其他用。事件位 0 存放在这个变量的 bit0 上，变量的 bit1 就是事件位 1，以此类推。对于 STM32 来说一个事件标志组最多可以存储 24 个事件位，如图 16.1.1 所示：

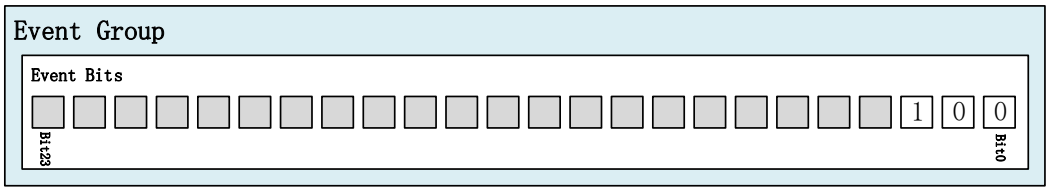


图 16.1.1 事件标志组和事件位

16.2 创建事件标志组

FreeRTOS 提供了两个用于创建事件标志组的函数，如表 16.2.1 所示：

函数	描述
xEventGroupCreate()	使用动态方法创建事件标志组。
xEventGroupCreateStatic()	使用静态方法创建事件标志组

表 16.2.1 事件标志组创建函数

1、函数 xEventGroupCreate()

此函数用于创建一个事件标志组，所需要的内存通过动态内存管理方法分配。由于内部处理的原因，事件标志组可用的 bit 数取决于 configUSE_16_BIT_TICKS，当 configUSE_16_BIT_TICKS1 为 1 的时候事件标志组有 8 个可用的位(bit0~bit7)，当 configUSE_16_BIT_TICKS 为 0 的时候事件标志组有 24 个可用的位(bit0~bit23)。EventBits_t 类型的变量用来存储事件标志组中的各个事件位，函数原型如下：

```
EventGroupHandle_t xEventGroupCreate( void )
```

参数：

无。

返回值：

NULL: 事件标志组创建失败。
其他值: 创建成功的事件标志组句柄。

2、函数 xEventGroupCreateStatic()

此函数用于创建一个事件标志组定时器，所需要的内存需要用户自行分配，此函数原型如下：

```
EventGroupHandle_t xEventGroupCreateStatic( StaticEventGroup_t *pxEventGroupBuffer )
```

参数：

pxEventGroupBuffer: 参数指向一个 StaticEventGroup_t 类型的变量，用来保存事件标志组结构体。

返回值：

NULL: 事件标志组创建失败。
其他值: 创建成功的事件标志组句柄。

16.3 设置事件位

FreeRTOS 提供了 4 个函数用来设置事件标志组中事件位(标志)，事件位(标志)的设置包括清零和置 1 两种操作，这 4 个函数如表 16.3.1 所示：

函数	描述
xEventGroupClearBits()	将指定的事件位清零，用在任务中。
xEventGroupClearBitsFromISR()	将指定的事件位清零，用在中断服务函数中

xEventGroupSetBits()	将指定的事件位置 1，用在任务中。
xEventGroupSetBitsFromISR()	将指定的事件位置 1，用在中断服务函数中。

表 16.3.1 事件位(标志)操作函数

1、函数 xEventGroupClearBits()

将事件标志组中的指定事件位清零，此函数只能用在任务中，不能用在中断服务函数中！中断服务函数有其他的 API 函数。函数原型如下：

```
EventBits_t xEventGroupClearBits( EventGroupHandle_t   xEventGroup,
                                  const EventBits_t    uxBitsToClear );
```

参数：

xEventGroup： 要操作的事件标志组的句柄。

uxBitsToClear： 要清零的事件位，比如要清除 bit3 的话就设置为 0X08。可以同时清除多个 bit，如设置为 0X09 的话就是同时清除 bit3 和 bit0。

返回值：

任何值： 将指定事件位清零之前的事件组值。

2、函数 xEventGroupClearBitsFromISR()

此函数为函数 xEventGroupClearBits()的中断级版本，也是将指定的事件位(标志)清零。此函数用在中断服务函数中，此函数原型如下：

```
BaseType_t xEventGroupClearBitsFromISR( EventGroupHandle_t   xEventGroup,
                                        const EventBits_t    uxBitsToSet );
```

参数：

xEventGroup： 要操作的事件标志组的句柄。

uxBitsToClear： 要清零的事件位，比如要清除 bit3 的话就设置为 0X08。可以同时清除多个 bit，如设置为 0X09 的话就是同时清除 bit3 和 bit0。

返回值：

pdPASS： 事件位清零成功。

pdFALSE: 事件位清零失败。

2、函数 xEventGroupSetBits()

设置指定的事件位为 1，此函数只能用在任务中，不能用于中断服务函数，此函数原型如下：

```
EventBits_t xEventGroupSetBits( EventGroupHandle_t   xEventGroup,
                                const EventBits_t    uxBitsToSet );
```

参数：

xEventGroup： 要操作的事件标志组的句柄。

uxBitsToClear： 指定要置 1 的事件位，比如要将 bit3 值 1 的话就设置为 0X08。可以同时将

多个 bit 置 1，如设置为 0X09 的话就是同时将 bit3 和 bit0 置 1。

返回值：

任何值： 在将指定事件位置 1 后的事件组值。

3、函数 xEventGroupSetBitsFromISR()

此函数也用于将指定的事件位置 1，此函数是 xEventGroupSetBits()的中断版本，用在中断服务函数中，函数原型如下：

```
BaseType_t xEventGroupSetBitsFromISR( EventGroupHandle_t    xEventGroup,
                                      const EventBits_t     uxBitsToSet,
                                      BaseType_t *          pxHigherPriorityTaskWoken );
```

参数：

xEventGroup： 要操作的事件标志组的句柄。

uxBitsToClear： 指定要置 1 的事件位，比如要将 bit3 值 1 的话就设置为 0X08。可以同时将多个 bit 置 1，如设置为 0X09 的话就是同时将 bit3 和 bit0 置 1。

pxHigherPriorityTaskWoken： 标记退出此函数以后是否进行任务切换，这个变量的值函数会自动设置的，用户不用进行设置，用户只需要提供一个变量来保存这个值就行了。当此值为 pdTRUE 的时候在退出中断服务函数之前一定要进行一次任务切换。

返回值：

pdPASS： 事件位置 1 成功。

pdFALSE: 事件位置 1 失败。

16.4 获取事件标志组值

我们可以通过 FreeRTOS 提供的 API 函数来查询事件标准组值，FreeRTOS 一共提供了两个这样的 API 函数，如表 16.4.1 所示：

函数	描述
xEventGroupGetBits()	获取当前事件标志组的值(各个事件位的值)，用在任务中。
xEventGroupGetBitsFromISR()	获取当前事件标志组的值，用在中断服务函数中。

表 16.4.1 获取事件标志组值

1、函数 xEventGroupGetBits()

此函数用于获取当前事件标志组的值，也就是各个事件位的值。此函数用在任务中，不能用在中断服务函数中。此函数是个宏，真正执行的是函数 xEventGroupClearBits()，函数原型如下：

```
EventBits_t xEventGroupGetBits( EventGroupHandle_t xEventGroup )
```

参数：

xEventGroup： 要获取的事件标志组的句柄。

返回值：

任何值：当前事件标志组的值。

2、函数 xEventGroupGetBitsFromISR()

获取当前事件标志组的值，此函数是 xEventGroupGetBits()的中断版本，函数原型如下：

```
EventBits_t xEventGroupGetBitsFromISR( EventGroupHandle_t xEventGroup )
```

参数：

xEventGroup：　要获取的事件标志组的句柄。

返回值：

任何值：　当前事件标志组的值。

16.5 等待指定的事件位

某个任务可能需要与多个事件进行同步，那么这个任务就需要等待并判断多个事件位(标志)，使用函数 xEventGroupWaitBits()可以完成这个功能。调用函数以后如果任务要等待的事件位还没有准备好(置 1 或清零)的话任务就会进入阻塞态，直到阻塞时间到达或者所等待的事件位准备好。函数原型如下：

```
EventBits_t xEventGroupWaitBits( EventGroupHandle_t   xEventGroup,
                                 const EventBits_t    uxBitsToWaitFor,
                                 const BaseType_t     xClearOnExit,
                                 const BaseType_t     xWaitForAllBits,
                                 const TickType_t     xTicksToWait );
```

参数：

xEventGroup：　指定要等待的事件标志组。

uxBitsToWaitFord：指定要等待的事件位，比如要等待 bit0 和(或)bit2 的时候此参数就是 0X05，如果要等待 bit0 和(或)bit1 和(或)bit2 的时候此参数就是 0X07，以此类推。

xClearOnExit：　此参数要是为 pdTRUE 的话，那么在退出此函数之前由参数 uxBitsToWaitFor 所设置的这些事件位就会清零。如果设置位 pdFALSE 的话这些事件位就不会改变。

xWaitForAllBits：　此参数如果设置为 pdTRUE 的话，当 uxBitsToWaitFor 所设置的这些事件位都置 1，或者指定的阻塞时间到的时候函数 xEventGroupWaitBits()才会返回。当此函数为 pdFALSE 的话，只要 uxBitsToWaitFor 所设置的这些事件位其中的任意一个置 1，或者指定的阻塞时间到的话函数 xEventGroupWaitBits()就会返回。

xTicksToWait：　设置阻塞时间，单位为节拍数。

返回值：

任何值：　返回当所等待的事件位置 1 以后的事件标志组的值，或者阻塞时间到。根据这个值我们就知道哪些事件位置 1 了。如果函数因为阻塞时间到而返回

的话那么这个返回值就不代表任何的含义。

16.6 事件标志组实验

16.6.1 实验程序设计

1、实验目的

学习 FreeROTS 事件标志组的使用，包括创建事件标志组、将相应的事件位置 1、等待相应的事件位置 1 等操作。

2、实验设计

本实验设计四个任务：start_task、eventsetbit_task、eventgroup_task 和 eventquery_task 这四个任务的任务功能如下：

start_task：用来创建其他三个任务和事件标志组。

eventsetbit_task：读取按键值，根据不同的按键值将事件标志组中相应的事件位置 1，用来模拟事件的发生。

eventgroup_task：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 的话就执行相应的处理，例程中是刷新 LCD 指定区域的背景色。

eventquery_task：查询事件组的值，也就是各个事件位的值。获取到事件组值以后就将其显示到 LCD 上，并且也通过串口打印出来。

实验中还创建了一个事件标志组： EventGroupHandler，实验中用到了这个事件标志组的三个事件位，分别位 bit0，bit1 和 bit2。

实验中会用到 3 个按键：KEY0、KEY1 和 KEY2，其中按键 KEY1 和 KEY2 为普通的输入模式。按键 KEY0 为中断输入模式，KEY0 用来演示如何在中断服务程序调用事件标志组的 API 函数。

3、实验工程

FreeRTOS 实验 16-1 FreeRTOS 事件标志组实验。

4、实验程序与分析

●任务设置

```
#define START_TASK_PRIO                 1          //任务优先级
#define START_STK_SIZE                  256        //任务堆栈大小
TaskHandle_t StartTask_Handler;                    //任务句柄
void start_task(void *pvParameters);               //任务函数

#define EVENTSETBIT_TASK_PRIO           2          //任务优先级
#define EVENTSETBIT_STK_SIZE            256        //任务堆栈大小
TaskHandle_t EventSetBit_Handler;                  //任务句柄
void eventsetbit_task(void *pvParameters);         //任务函数

#define EVENTGROUP_TASK_PRIO            3          //任务优先级
#define EVENTGROUP_STK_SIZE             256        //任务堆栈大小
TaskHandle_t EventGroupTask_Handler;               //任务句柄
void eventgroup_task(void *pvParameters);          //任务函数
```

```
#define EVENTQUERY_TASK_PRIO        4          //任务优先级
#define EVENTQUERY_STK_SIZE         256        //任务堆栈大小
TaskHandle_t EventQueryTask_Handler;           //任务句柄
void eventquery_task(void *pvParameters);      //任务函数

//////////////////////////////////////////////////////
EventGroupHandle_t EventGroupHandler;          //事件标志组句柄

#define EVENTBIT_0   (1<<0)                    //事件位
#define EVENTBIT_1   (1<<1)
#define EVENTBIT_2   (1<<2)
#define EVENTBIT_ALL     (EVENTBIT_0|EVENTBIT_1|EVENTBIT_2)

//LCD 刷屏时使用的颜色
int lcd_discolor[14]={   WHITE,      BLACK,      BLUE,       BRED,
                         GRED,       GBLUE,      RED,        MAGENTA,
                         GREEN,      CYAN,       YELLOW,     BROWN,
                         BRRED,      GRAY };
```

- **main()函数**

```
int main(void)
{
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(360,25,2,8);   //设置时钟,180Mhz
    delay_init(180);                //初始化延时函数
    uart_init(115200);              //初始化串口
    LED_Init();                     //初始化 LED
    KEY_Init();                     //初始化按键
    PCF8574_Init();                 //初始化 PCF8574
    EXTI_Init();                    //初始化外部中断
    SDRAM_Init();                   //初始化 SDRAM
    LCD_Init();                     //初始化 LCD
    my_mem_init(SRAMIN);            //初始化内部内存池

    POINT_COLOR = RED;
    LCD_ShowString(30,10,200,16,16,"Apollo STM32F4/F7");
    LCD_ShowString(30,30,200,16,16,"FreeRTOS Examp 16-1");
    LCD_ShowString(30,50,200,16,16,"Event Group");
    LCD_ShowString(30,70,200,16,16,"ATOM@ALIENTEK");
    LCD_ShowString(30,90,200,16,16,"2016/11/11");

    POINT_COLOR = BLACK;
    LCD_DrawRectangle(5,130,234,314);       //画矩形
```

```
    POINT_COLOR = BLUE;
    LCD_ShowString(30,110,220,16,16,"Event Group Value:0");

    //创建开始任务
    xTaskCreate((TaskFunction_t     )start_task,              //任务函数
                (const char*        )"start_task",            //任务名称
                (uint16_t           )START_STK_SIZE,          //任务堆栈大小
                (void*              )NULL,                    //传递给任务函数的参数
                (UBaseType_t        )START_TASK_PRIO,         //任务优先级
                (TaskHandle_t*      )&StartTask_Handler);     //任务句柄
    vTaskStartScheduler();                                    //开启任务调度
}
```

● 任务函数

```
//开始任务任务函数
void start_task(void *pvParameters)
{
    taskENTER_CRITICAL();           //进入临界区
    //创建事件标志组
    EventGroupHandler=xEventGroupCreate();  //创建事件标志组                    (1)

    //创建设置事件位的任务
    xTaskCreate((TaskFunction_t     )eventsetbit_task,
                (const char*        )"eventsetbit_task",
                (uint16_t           )EVENTSETBIT_STK_SIZE,
                (void*              )NULL,
                (UBaseType_t        )EVENTSETBIT_TASK_PRIO,
                (TaskHandle_t*      )&EventSetBit_Handler);
    //创建事件标志组处理任务
    xTaskCreate((TaskFunction_t     )eventgroup_task,
                (const char*        )"eventgroup_task",
                (uint16_t           )EVENTGROUP_STK_SIZE,
                (void*              )NULL,
                (UBaseType_t        )EVENTGROUP_TASK_PRIO,
                (TaskHandle_t*      )&EventGroupTask_Handler);
    //创建事件标志组查询任务
    xTaskCreate((TaskFunction_t     )eventquery_task,
                (const char*        )"eventquery_task",
                (uint16_t           )EVENTQUERY_STK_SIZE,
                (void*              )NULL,
                (UBaseType_t        )EVENTQUERY_TASK_PRIO,
                (TaskHandle_t*      )&EventQueryTask_Handler);
    vTaskDelete(StartTask_Handler); //删除开始任务
    taskEXIT_CRITICAL();            //退出临界区
```

```
}


//设置事件位的任务
void eventsetbit_task(void *pvParameters)
{
    u8 key;
    while(1)
    {
        if(EventGroupHandler!=NULL)
        {
            key=KEY_Scan(0);
            switch(key)
            {
                case KEY1_PRES:
                    xEventGroupSetBits(EventGroupHandler,EVENTBIT_1);        (2)
                    break;
                case KEY2_PRES:
                    xEventGroupSetBits(EventGroupHandler,EVENTBIT_2);        (3)
                    break;
            }
        }
        vTaskDelay(10); //延时 10ms，也就是 10 个时钟节拍
    }
}

//事件标志组处理任务
void eventgroup_task(void *pvParameters)
{
    u8 num;
    EventBits_t EventValue;
    while(1)
    {

        if(EventGroupHandler!=NULL)
        {
            //等待事件组中的相应事件位
            EventValue=xEventGroupWaitBits((EventGroupHandle_t  )EventGroupHandler, (4)
                                           (EventBits_t         ) EVENTBIT_ALL,
                                           (BaseType_t          )pdTRUE,
                                           (BaseType_t          )pdTRUE,
                                           (TickType_t          )portMAX_DELAY);
            printf("事件标志组的值:%d\r\n",EventValue);
```

```
                LCD_ShowxNum(174,110,EventValue,1,16,0);

                num++;
                LED1=!LED1;
                LCD_Fill(6,131,233,313,lcd_discolor[num%14]);
            }
            else
            {
                vTaskDelay(10); //延时 10ms，也就是 10 个时钟节拍
            }
        }
}

//事件查询任务
void eventquery_task(void *pvParameters)
{
    u8 num=0;
    EventBits_t NewValue,LastValue;
    while(1)
    {
        if(EventGroupHandler!=NULL)
        {
            NewValue=xEventGroupGetBits(EventGroupHandler);     //获取事件组的     (5)
            if(NewValue!=LastValue)
            {
                LastValue=NewValue;
                printf("事件标志组的值:%d\r\n",NewValue);
                LCD_ShowxNum(174,110,NewValue,1,16,0);
            }
        }
        num++;
        if(num==0)    //每 500msLED0 闪烁一次
        {
            num=0;
            LED0=!LED0;
        }
        vTaskDelay(50); //延时 50ms，也就是 50 个时钟节拍
    }
}
```

(1)、首先调用函数 xEventGroupCreate()创建一个事件标志组 EventGroupHandler。

(2)、按下 KEY1 键的时候就调用函数 xEventGroupSetBits()将事件标志组的 bit1 置 1。

(3)、按下 KEY2 键的时候调用函数 xEventGroupSetBits()将事件标志组的 bit2 值 1。

(4)、调用函数 xEventGroupWaitBits()同时等待事件标志组的 bit0，bit1 和 bit2，只有当这三

个事件都置 1 的时候才会执行任务中的其他代码。

(5)、调用函数 xEventGroupGetBits()查询事件标志组 EventGroupHandler 的值变化，通过查看这些值的变化就可以分析出当前哪个事件位置 1 了。

● **中断初始化及处理过程**

事件标志组 EventGroupHandler 的事件位 bit0 是通过 KEY0 的外部中断服务函数来设置的，注意中断优先级的设置！本例程的中断优先级设置如下：

```
//中断线 3-PH3
HAL_NVIC_SetPriority(EXTI3_IRQn,6,0);          //抢占优先级为 6，子优先级为 0
HAL_NVIC_EnableIRQ(EXTI3_IRQn);                //使能中断线 3
```

KEY0 的外部中断服务函数如下：

```
//事件标志组句柄
extern EventGroupHandle_t EventGroupHandler;

//中断服务函数
void EXTI3_IRQHandler(void)
{
    BaseType_t Result,xHigherPriorityTaskWoken;

    delay_xms(50);                             //消抖
    if(KEY0==0)
    {
        Result=xEventGroupSetBitsFromISR(EventGroupHandler,EVENTBIT_0,\      (1)
                                         &xHigherPriorityTaskWoken);
        if(Result!=pdFAIL)
        {
            portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
        }
    }
    __HAL_GPIO_EXTI_CLEAR_IT(GPIO_PIN_3);     //清除中断标志位
}
```

(1)、在中断服务函数中通过调用 xEventGroupSetBitsFromISR()来将事件标志组的事件位 bit0 置 1。

16.6.2 程序运行结果分析

编译并下载实验代码到开发板中，打开串口调试助手，默认情况下 LCD 显示如图 16.6.2.1 所示：

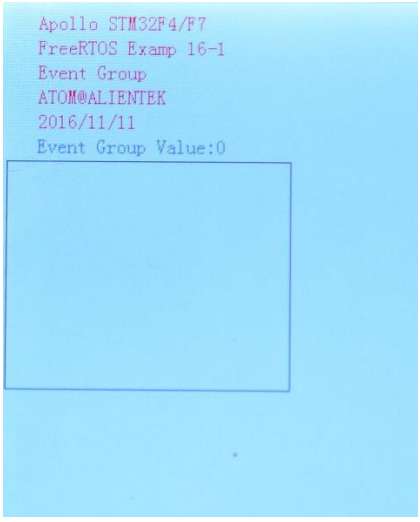


图 16.6.2.1 LCD 默认界面

通过按下 KEY0、KEY1 和 KEY2 来观察 LCD 的变化和串口调试助手收到的信息，当 KEY0、KEY1 和 KEY2 都有按下的时候 LCD 指定区域的背景颜色就会刷新，因为三个事件位都置 1 了，LCD 显示如图 16.6.2.2 所示：

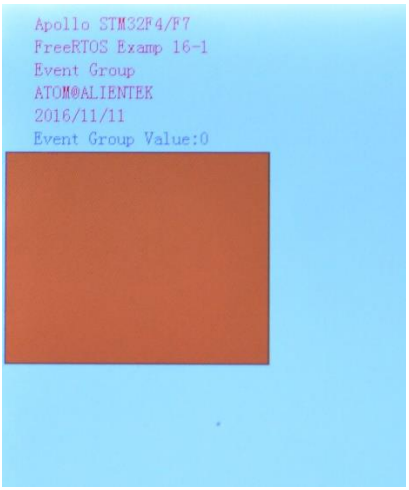


图 16.6.2.2 LCD 指定区域背景颜色改变

注意观察，当 LCD 背景颜色刷新完成以后事件标志组就会清零，通过串口调试助手可以很清楚的观察到事件标志组值的变化情况，如图 16.6.2.3 所示：

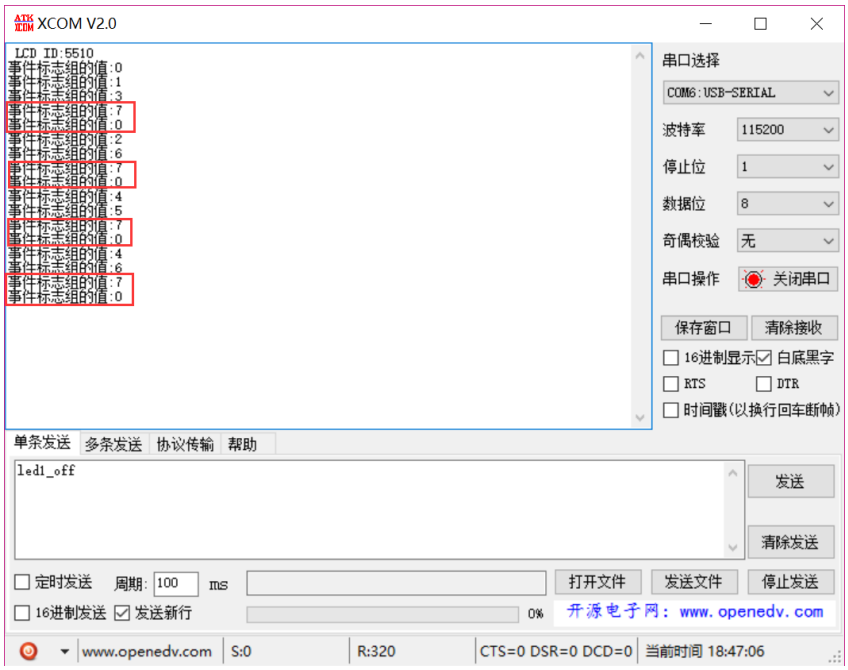


图 16.6.2.3 串口调试助手

从图 16.6.2.3 中可以看出，每次事件标志组的值为 7 的话就会紧接着将事件标志组的值清零。这是因为当事件标志组的值为 7 的话说明事件位 bit0，bit1 和 bit2 都置 1 了，任务 eventgroup_task()所等待的事件满足了，然后函数 xEventGroupWaitBits()就会将事件组清零，因为我们在调用函数 xEventGroupWaitBits()的时候就设置的要将事件组值清零的。

第十七章 FreeRTOS 任务通知

从 v8.2.0 版本开始，FreeRTOS 新增了任务通知(Task Notifictions)这个功能，可以使用任务通知来代替信号量、消息队列、事件标志组等这些东西。使用任务通知的话效率会更高，本章我们就来学习一下 FreeRTOS 的任务通知功能，本章分为如下几部分：

17.1 任务通知简介
17.2 发送任务通知
17.3 任务通知通用发送函数
17.4 获取任务通知
17.5 任务通知模拟二值信号量实验
17.6 任务通知模拟计数型信号量实验
17.7 任务通知模拟消息邮箱实验
17.8 任务通知模拟事件标志组实验

17.1 任务通知简介

任务通知在 FreeRTOS 中是一个可选的功能，要使用任务通知的话就需要将宏 configUSE_TASK_NOTIFICATIONS 定义为 1。

FreeRTOS 的每个任务都有一个 32 位的通知值，任务控制块中的成员变量 ulNotifiedValue 就是这个通知值。任务通知是一个事件，假如某个任务通知的接收任务因为等待任务通知而阻塞的话，向这个接收任务发送任务通知以后就会解除这个任务的阻塞状态。也可以更新接收任务的任务通知值，任务通知可以通过如下方法更新接收任务的通知值：

- 不覆盖接收任务的通知值(如果上次发送给接收任务的通知还没被处理)。
- 覆盖接收任务的通知值。
- 更新接收任务通知值的一个或多个 bit。
- 增加接收任务的通知值。

合理、灵活的使用上面这些更改任务通知值的方法可以在一些场合中替代队列、二值信号量、计数型信号量和事件标志组。使用任务通知来实现二值信号量功能的时候，解除任务阻塞的时间比直接使用二值信号量要快 45%(FreeRTOS 官方测试结果，使用 v8.1.2 版本中的二值信号量，GCC 编译器，-O2 优化的条件下测试的，没有使能断言函数 configASSERT())，并且使用的 RAM 更少！

任务通知的发送使用函数 xTaskNotify()或者 xTaskNotifyGive()(还有此函数的中断版本)来完成，这个通知值会一直被保存着，直到接收任务调用函数 xTaskNotifyWait()或者 ulTaskNotifyTake()来获取这个通知值。假如接收任务因为等待任务通知而阻塞的话那么在接收到任务通知以后就会解除阻塞态。

任务通知虽然可以提高速度，并且减少 RAM 的使用，但是任务通知也是有使用限制的：

- FreeRTOS 的任务通知只能有一个接收任务，其实大多数的应用都是这种情况。
- 接收任务可以因为接收任务通知而进入阻塞态，但是发送任务不会因为任务通知发送失败而阻塞。

17.2 发送任务通知

任务通知发送函数有 6 个，如表 17.2.1 所示：

函数	描述
xTaskNotify()	发送通知，带有通知值并且不保留接收任务原通知值，用在任务中。
xTaskNotifyFromISR()	发送通知，函数 xTaskNotify()的中断版本。
xTaskNotifyGive()	发送通知，不带通知值并且不保留接收任务的通知值，此函数会将接收任务的通知值加一，用于任务中。
vTaskNotifyGiveFromISR()	发送通知，函数 xTaskNotifyGive()的中断版本。
xTaskNotifyAndQuery()	发送通知，带有通知值并且保留接收任务的原通知值，用在任务中。
xTaskNotiryAndQueryFromISR()	发送通知，函数 xTaskNotifyAndQuery()的中断版本，用在中断服务函数中。

表 17.2.1 任务通知发送函数

1、函数 xTaskNotify()

此函数用于发送任务通知，此函数发送任务通知的时候带有通知值，此函数是个宏，真正执行的函数 xTaskGenericNotify()，函数原型如下：

```
BaseType_t xTaskNotify( TaskHandle_t     xTaskToNotify,
                        uint32_t         ulValue,
                        eNotifyAction    eAction )
```

参数：

xTaskToNotify： 任务句柄，指定任务通知是发送给哪个任务的。

ulValue： 任务通知值。

eAction： 任务通知更新的方法，eNotifyAction 是个枚举类型，在文件 task.h 中有如下定义：

```
typedef enum
{
    eNoAction = 0,
    eSetBits,                   //更新指定的 bit
    eIncrement,                 //通知值加一
    eSetValueWithOverwrite,     //覆写的方式更新通知值
    eSetValueWithoutOverwrite   //不覆写通知值
} eNotifyAction;
```

此参数可以选择枚举类型中的任意一个，不同的应用环境其选择也不同。

返回值：

pdFAIL: 当参数 eAction 设置为 eSetValueWithoutOverwrite 的时候，如果任务通知值没有更新成功就返回 pdFAIL。

pdPASS: eAction 设置为其他选项的时候统一返回 pdPASS。

2、函数 xTaskNotifyFromISR()

此函数用于发送任务通知，是函数 xTaskNotify()的中断版本，此函数是个宏，真正执行的是函数 xTaskGenericNotifyFromISR()，此函数原型如下：

```
BaseType_t xTaskNotifyFromISR( TaskHandle_t     xTaskToNotify,
                               uint32_t         ulValue,
                               eNotifyAction    eAction,
                               BaseType_t *     pxHigherPriorityTaskWoken );
```

参数：

xTaskToNotify： 任务句柄，指定任务通知是发送给哪个任务的。

ulValue： 任务通知值。

eAction： 任务通知更新的方法。

pxHigherPriorityTaskWoken: 记退出此函数以后是否进行任务切换，这个变量的值函数会自动设置的，用户不用进行设置，用户只需要提供一个变量来保存这个值就行了。当此值为 pdTRUE 的时候在退出中断服务函数之前一定要进行一次任务切换。

返回值：

pdFAIL: 当参数 eAction 设置为 eSetValueWithoutOverwrite 的时候，如果任务通知值没有更新成功就返回 pdFAIL。

pdPASS: eAction 设置为其他选项的时候统一返回 pdPASS。

3、函数 xTaskNotifyGive()

发送任务通知，相对于函数 xTaskNotify()，此函数发送任务通知的时候不带有通知值。此函数只是将任务通知值简单的加一，此函数是个宏，真正执行的是函数 xTaskGenericNotify()，此函数原型如下：

```
BaseType_t xTaskNotifyGive( TaskHandle_t xTaskToNotify );
```

参数：

xTaskToNotify： 任务句柄，指定任务通知是发送给哪个任务的。

返回值：

pdPASS: 此函数只会返回 pdPASS。

4、函数 vTaskNotifyGiveFromISR()

此函数为 xTaskNotifyGive()的中断版本，用在中断服务函数中，函数原型如下：

```
void vTaskNotifyGiveFromISR( TaskHandle_t     xTaskHandle,
                             BaseType_t *     pxHigherPriorityTaskWoken );
```

参数：

xTaskToNotify： 任务句柄，指定任务通知是发送给哪个任务的。

pxHigherPriorityTaskWoken: 记退出此函数以后是否进行任务切换，这个变量的值函数会自动设置的，用户不用进行设置，用户只需要提供一个变量来保存这个值就行了。当此值为 pdTRUE 的时候在退出中断服务函数之前一定要进行一次任务切换。

返回值：

无。

5、函数 xTaskNotifyAndQuery()

此函数和 xTaskNotify()很类似，此函数比 xTaskNotify()多一个参数，此参数用来保存更新前的通知值。此函数是个宏，真正执行的是函数 xTaskGenericNotify()，此函数原型如下：

```
BaseType_t xTaskNotifyAndQuery ( TaskHandle_t     xTaskToNotify,
                                 uint32_t         ulValue,
                                 eNotifyAction    eAction
                                 uint32_t *       pulPreviousNotificationValue);
```

参数：

xTaskToNotify： 任务句柄，指定任务通知是发送给哪个任务的。

ulValue： 任务通知值。
eAction： 任务通知更新的方法。
pulPreviousNotificationValue： 用来保存更新前的任务通知值。

返回值：

pdFAIL: 当参数 eAction 设置为 eSetValueWithoutOverwrite 的时候，如果任务通知值没有更新成功就返回 pdFAIL。
pdPASS: eAction 设置为其他选项的时候统一返回 pdPASS。

6、函数 xTaskNotifyAndQueryFromISR()

此函数为 xTaskNorityAndQuery()的中断版本，用在中断服务函数中。此函数同样为宏，真正执行的是函数 xTaskGenericNotifyFromISR()，此函数的原型如下：

```
BaseType_t xTaskNotifyAndQueryFromISR ( TaskHandle_t     xTaskToNotify,
                                        uint32_t         ulValue,
                                        eNotifyAction    eAction,
                                        uint32_t *       pulPreviousNotificationValue
                                        BaseType_t *     pxHigherPriorityTaskWoken );
```

参数：

xTaskToNotify： 任务句柄，指定任务通知是发送给哪个任务的。
ulValue： 任务通知值。
eAction： 任务通知更新的方法。
pulPreviousNotificationValue： 用来保存更新前的任务通知值。
pxHigherPriorityTaskWoken: 记退出此函数以后是否进行任务切换，这个变量的值函数会自动设置的，用户不用进行设置，用户只需要提供一个变量来保存这个值就行了。当此值为 pdTRUE 的时候在退出中断服务函数之前一定要进行一次任务切换。

返回值：

pdFAIL: 当参数 eAction 设置为 eSetValueWithoutOverwrite 的时候，如果任务通知值没有更新成功就返回 pdFAIL。
pdPASS: eAction 设置为其他选项的时候统一返回 pdPASS。

17.3 任务通知通用发送函数

17.3.1 任务级任务通知通用发送函数

在 17.2 小节中我们学习了 3 个任务级任务通知发送函数：xTaskNotify()、xTaskNotifyGive()和 xTaskNotifyAndQuery()，这三个函数最终调用的都是函数 xTaskGenericNotify()！此函数在文件 tasks.c 中有如下定义，缩减后的函数如下：

```
BaseType_t xTaskGenericNotify( TaskHandle_t     xTaskToNotify,    //任务句柄
                               uint32_t         ulValue,          //任务通知值
                               eNotifyAction    eAction,          //任务通知更新方式
```

```
                                    uint32_t *          pulPreviousNotificationValue )//保存更新前的
                                                                                      //任务通知值
{
    TCB_t * pxTCB;
    BaseType_t xReturn = pdPASS;
    uint8_t ucOriginalNotifyState;

    configASSERT( xTaskToNotify );
    pxTCB = ( TCB_t * ) xTaskToNotify;

    taskENTER_CRITICAL();
    {
        if( pulPreviousNotificationValue != NULL )                              (1)
        {
            *pulPreviousNotificationValue = pxTCB->ulNotifiedValue;             (2)
        }
        ucOriginalNotifyState = pxTCB->ucNotifyState;                           (3)

        pxTCB->ucNotifyState = taskNOTIFICATION_RECEIVED;                       (4)

        switch( eAction )
        {
            case eSetBits   :                                                   (5)
                pxTCB->ulNotifiedValue |= ulValue;
                break;
            case eIncrement :                                                   (6)
                ( pxTCB->ulNotifiedValue )++;
                break;
            case eSetValueWithOverwrite:                                        (7)
                pxTCB->ulNotifiedValue = ulValue;
                break;
            case eSetValueWithoutOverwrite :                                    (8)
                if( ucOriginalNotifyState != taskNOTIFICATION_RECEIVED )
                {
                    pxTCB->ulNotifiedValue = ulValue;
                }
                else
                {
                    xReturn = pdFAIL;
                }
                break;
            case eNoAction:
                break;
```

```
        }

        traceTASK_NOTIFY();

        //如果任务因为等待任务通知而进入阻塞态的话就需要解除阻塞
        if( ucOriginalNotifyState == taskWAITING_NOTIFICATION )                    (9)
        {
            ( void ) uxListRemove( &( pxTCB->xStateListItem ) );                   (10)
            prvAddTaskToReadyList( pxTCB );                                        (11)
        /*************************************************************************/
        /*********************省略相关的条件编译代码*************************/
        /*************************************************************************/
            if( pxTCB->uxPriority > pxCurrentTCB->uxPriority )                     (12)
            {
                //解除阻塞的任务优先级比当前运行的任务优先级高，所以需要进行
                //任务切换。
                taskYIELD_IF_USING_PREEMPTION();
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
            }
        }
        else
        {
            mtCOVERAGE_TEST_MARKER();
        }
    }
    taskEXIT_CRITICAL();
    return xReturn;                                                            (13)
}
```

(1)、判断参数 pulPreviousNotificationValue 是否有效，因为此参数用来保存更新前的任务通知值。

(2)、如果参数 pulPreviousNotificationValue 有效的话就用此参数保存更新前的任务通知值。

(3)、保存任务通知状态，因为下面会修改这个状态，后面我们要根据这个状态来确定是否将任务从阻塞态解除。

(4)、更新任务通知状态为 taskNOTIFICATION_RECEIVED。

(5)、根据不同的更新方式做不同的处理，如果为 eSetBits 的话就将指定的 bit 置 1。也就是更新接收任务通知值的一个或多个 bit。

(6)、如果更新方式为 eIncrement 的话就将任务通知值加一。

(7)、如果更新方式为 eSetValueWithOverwrite 的话就直接覆写原来的任务通知值。

(8)、如果更新方式为 eSetValueWithoutOverwrite 的话就需要判断原来的任务通知值是否被处理，如果已经被处理了就更新为任务通知值。如果此前的任务通知值话没有被处理的话就标

记 xReturn 为 pdFAIL，后面会返回这个值。

(9)、根据(3)中保存的接收任务之前的状态值来判断是否有任务需要解除阻塞，如果在任务通知值被更新前任务处于 taskWAITING_NOTIFICATION 状态的话就说明有任务因为等待任务通知值而进入了阻塞态。

(10)、将任务从状态列表中移除。

(11)、将任务重新添加到就绪列表中。

(12)、判断刚刚解除阻塞的任务优先级是否比当前正在运行的任务优先级高，如果是的话需要进行一次任务切换。

(13)、返回 xReturn 的值，pdFAIL 或 pdPASS。

17.3.2 中断级任务通知发送函数

中断级任务通知发送函数也有三个，分别为：xTaskNotifyFromISR()、xTaskNotifyAndQueryFromISR()和 vTaskNotifyGiveFromISR()。其中函数 xTaskNotifyFromISR()和 xTaskNotifyAndQueryFromISR()最终调用的都是函数 xTaskGenericNotifyFromISR()，此函数的原型如下:

```
BaseType_t xTaskGenericNotifyFromISR( TaskHandle_t        xTaskToNotify,
                                      uint32_t            ulValue,
                                      eNotifyAction       eAction,
                                      uint32_t *          pulPreviousNotificationValue,
                                      BaseType_t *        pxHigherPriorityTaskWoken )
```

参数：

xTaskToNotify： 任务句柄，指定任务通知是发送给哪个任务的。

ulValue： 任务通知值。

eAction： 任务通知更新的方法。

pulPreviousNotificationValue： 用来保存更新前的任务通知值。

pxHigherPriorityTaskWoken： 记退出此函数以后是否进行任务切换，这个变量的值函数会自动设置的，用户不用进行设置，用户只需要提供一个变量来保存这个值就行了。当此值为 pdTRUE 的时候在退出中断服务函数之前一定要进行一次任务切换。

返回值：

pdFAIL： 当参数 eAction 设置为 eSetValueWithoutOverwrite 的时候，如果任务通知值没有更新成功就返回 pdFAIL。

pdPASS： eAction 设置为其他选项的时候统一返回 pdPASS。

函数 xTaskGenericNotifyFromISR()在文件 tasks.c 中有定义，函数源码如下：

```
BaseType_t xTaskGenericNotifyFromISR( TaskHandle_t        xTaskToNotify,
                                      uint32_t            ulValue,
                                      eNotifyAction       eAction,
                                      uint32_t *          pulPreviousNotificationValue,
                                      BaseType_t *        pxHigherPriorityTaskWoken )
{
    TCB_t * pxTCB;
```

```
uint8_t ucOriginalNotifyState;
BaseType_t xReturn = pdPASS;
UBaseType_t uxSavedInterruptStatus;

configASSERT( xTaskToNotify );

portASSERT_IF_INTERRUPT_PRIORITY_INVALID();

pxTCB = ( TCB_t * ) xTaskToNotify;

uxSavedInterruptStatus = portSET_INTERRUPT_MASK_FROM_ISR();
{
    if( pulPreviousNotificationValue != NULL )                    (1)
    {
        *pulPreviousNotificationValue = pxTCB->ulNotifiedValue;
    }

    ucOriginalNotifyState = pxTCB->ucNotifyState;                 (2)
    pxTCB->ucNotifyState = taskNOTIFICATION_RECEIVED;             (3)

    switch( eAction )                                             (4)
    {
        case eSetBits   :
            pxTCB->ulNotifiedValue |= ulValue;
            break;
        case eIncrement     :
            ( pxTCB->ulNotifiedValue )++;
            break;
        case eSetValueWithOverwrite:
            pxTCB->ulNotifiedValue = ulValue;
            break;
        case eSetValueWithoutOverwrite :
            if( ucOriginalNotifyState != taskNOTIFICATION_RECEIVED )
            {
                pxTCB->ulNotifiedValue = ulValue;
            }
            else
            {
                xReturn = pdFAIL;
            }
            break;
        case eNoAction :
            break;
```

```
        }

        traceTASK_NOTIFY_FROM_ISR();

        //如果任务因为等待任务通知而进入阻塞态的话就需要解除阻塞
        if( ucOriginalNotifyState == taskWAITING_NOTIFICATION )                    (5)
        {
            configASSERT( listLIST_ITEM_CONTAINER( &( pxTCB->xEventListItem ) ) ==\
                                                                NULL );

            if( uxSchedulerSuspended == ( UBaseType_t ) pdFALSE )                  (6)
            {
                ( void ) uxListRemove( &( pxTCB->xStateListItem ) );
                prvAddTaskToReadyList( pxTCB );
            }
            else                                                                   (7)
            {
                vListInsertEnd( &( xPendingReadyList ), &( pxTCB->xEventListItem ) );
            }

            if( pxTCB->uxPriority > pxCurrentTCB->uxPriority )                     (8)
            {
                //解除阻塞的任务优先级比当前运行任务的优先级高，所以需要标记
                //在退出中断服务函数的时候需要做任务切换。
                if( pxHigherPriorityTaskWoken != NULL )
                {
                    *pxHigherPriorityTaskWoken = pdTRUE;
                }
                else
                {
                    xYieldPending = pdTRUE;
                }
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
            }
        }
    }
    portCLEAR_INTERRUPT_MASK_FROM_ISR( uxSavedInterruptStatus );
    return xReturn;
}
```

(1)、判断参数 pulPreviousNotificationValue 是否有效，因为此参数用来保存更新前的任务通

知值。如果参数 pulPreviousNotificationValue 有效的话就用此参数保存更新前的任务通知值。

(2)、保存任务通知状态，因为下面会修改这个状态，后面我们要根据这个状态来确定是否将任务解除阻塞态。

(3)、更新任务通知状态 taskNOTIFICATION_RECEIVED。

(4)、根据不同的通知值更新方式来做不同的处理，与函数 xTaskGenericNotify()的处理过程一样。

(5)、根据(2)中保存的接收任务之前的状态值来判断是否有任务需要解除阻塞，如果在任务通知值被更新前任务处于 taskWAITING_NOTIFICATION 状态的话就说明有任务因为等待任务通知值而进入了阻塞态。

(6)、判断任务调度器是否上锁，如果调度器没有上锁的话就将任务从状态列表中移除，然后重新将任务添加到就绪列表中。

(7)、如果任务调度器上锁了的话就将任务添加到列表 xPendingReadyList 中。

(8)、判断任务解除阻塞的任务优先级是否比当前任务优先级高，如果是的话就将 pxHigherPriorityTaskWoken 标记 pdTRUE。如果参数 pxHigherPriorityTaskWoken 无效的话就将全局变量 xYieldPending 标记为 pdTRUE。

还有另外一个用于中断服务函数的任务通知发送函数 vTaskNotifyGiveFromISR()，此函数和 xTaskGenericNotifyFromISR()极其类似。此函数用于将任务通知值加一，大家可以自行分析一下此函数。

17.4 获取任务通知

获取任务通知的函数有两个，如表 17.4.1 所示：

函数	描述
ulTaskNotifyTake()	获取任务通知，可以设置在退出此函数的时候将任务通知值清零或者减一。当任务通知用作二值信号量或者计数信号量的时候使用此函数来获取信号量。
xTaskNotifyWait()	等待任务通知，比 ulTaskNotifyTak()更为强大，全功能版任务通知获取函数。

表 17.4.1 任务通知获取函数

1、函数 ulTaskNotifyTake()

此函数为获取任务通知函数，当任务通知用作二值信号量或者计数型信号量的时候可以使用此函数来获取信号量，函数原型如下：

```
uint32_t ulTaskNotifyTake( BaseType_t    xClearCountOnExit,
                           TickType_t    xTicksToWait );
```

参数：

xClearCountOnExit： 参数为 pdFALSE 的话在退出函数 ulTaskNotifyTake()的时候任务通知值减一，类似计数型信号量。当此参数为 pdTRUE 的话在退出函数的时候任务任务通知值清零，类似二值信号量。

xTickToWait: 阻塞时间。

返回值：

任何值 ： 任务通知值减少或者清零之前的值。

此函数在文件 tasks.c 中有定义，代码如下：

```
uint32_t ulTaskNotifyTake( BaseType_t xClearCountOnExit, TickType_t xTicksToWait )
{
    uint32_t ulReturn;

    taskENTER_CRITICAL();
    {

        if( pxCurrentTCB->ulNotifiedValue == 0UL )                    (1)
        {
            pxCurrentTCB->ucNotifyState = taskWAITING_NOTIFICATION;   (2)
            if( xTicksToWait > ( TickType_t ) 0 )                     (3)
            {
                prvAddCurrentTaskToDelayedList( xTicksToWait, pdTRUE );
                traceTASK_NOTIFY_TAKE_BLOCK();
                portYIELD_WITHIN_API();
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
            }
        }
        else
        {
            mtCOVERAGE_TEST_MARKER();
        }
    }
    taskEXIT_CRITICAL();

    taskENTER_CRITICAL();
    {
        traceTASK_NOTIFY_TAKE();
        ulReturn = pxCurrentTCB->ulNotifiedValue;                     (4)

        if( ulReturn != 0UL )                                         (5)
        {
            if( xClearCountOnExit != pdFALSE )                        (6)
            {
                pxCurrentTCB->ulNotifiedValue = 0UL;
            }
            else
            {
```

```
                    pxCurrentTCB->ulNotifiedValue = ulReturn - 1;                    (7)
                }
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
            }

            pxCurrentTCB->ucNotifyState = taskNOT_WAITING_NOTIFICATION;          (8)
        }
        taskEXIT_CRITICAL();
        return ulReturn;
}
```

(1)、判断任务通知值是否为 0，如果为 0 的话说明还没有接收到任务通知。
(2)、修改任务通知状态为 taskWAITING_NOTIFICATION。
(3)、如果阻塞时间不为 0 的话就将任务添加到延时列表中，并且进行一次任务调度。
(4)、如果任务通知值不为 0 的话就先获取任务通知值。
(5)、任务通知值大于 0。
(6)、参数 xClearCountOnExit 不为 pdFALSE，那就将任务通知值清零。
(7)、如果参数 xClearCountOnExit 为 pdFALSE 的话那就将任务通知值减一。
(8)、更新任务通知状态为 taskNOT_WAITING_NOTIFICATION。

2、函数 xTaskNotifyWait()

此函数也是用来获取任务通知的，不过此函数比 ulTaskNotifyTake()更为强大，不管任务通知用作二值信号量、计数型信号量、队列和事件标志组中的哪一种，都可以使用此函数来获取任务通知。但是当任务通知用作位置信号量和计数型信号量的时候推荐使用函数 ulTaskNotifyTake()。此函数原型如下：

```
BaseType_t xTaskNotifyWait( uint32_t      ulBitsToClearOnEntry,
                            uint32_t      ulBitsToClearOnExit,
                            uint32_t *    pulNotificationValue,
                            TickType_t    xTicksToWait );
```

参数：

ulBitsToClearOnEntry： 当没有接收到任务通知的时候将任务通知值与此参数的取反值进行按位与运算，当此参数为 0xffffffff 或者 ULONG_MAX 的时候就会将任务通知值清零。

ulBitsToClearOnExit： 如果接收到了任务通知，在做完相应的处理退出函数之前将任务通知值与此参数的取反值进行按位与运算，当此参数为 0xffffffff 或者 ULONG_MAX 的时候就会将任务通知值清零。

pulNotificationValue： 此参数用来保存任务通知值。

xTickToWait: 阻塞时间。

返回值：

pdTRUE： 获取到了任务通知。

pdFALSE： 任务通知获取失败。

此函数在文件 tasks.c 中有定义，代码如下：

```
BaseType_t xTaskNotifyWait( uint32_t      ulBitsToClearOnEntry,
                            uint32_t      ulBitsToClearOnExit,
                            uint32_t *    pulNotificationValue,
                            TickType_t    xTicksToWait )
{
    BaseType_t xReturn;

    taskENTER_CRITICAL();
    {
        if( pxCurrentTCB->ucNotifyState != taskNOTIFICATION_RECEIVED )        (1)
        {
            pxCurrentTCB->ulNotifiedValue &= ~ulBitsToClearOnEntry;           (2)
            pxCurrentTCB->ucNotifyState = taskWAITING_NOTIFICATION;           (3)
            if( xTicksToWait > ( TickType_t ) 0 )                             (4)
            {
                prvAddCurrentTaskToDelayedList( xTicksToWait, pdTRUE );
                traceTASK_NOTIFY_WAIT_BLOCK();
                portYIELD_WITHIN_API();
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
            }
        }
        else
        {
            mtCOVERAGE_TEST_MARKER();
        }
    }
    taskEXIT_CRITICAL();

    taskENTER_CRITICAL();
    {
        traceTASK_NOTIFY_WAIT();

        if( pulNotificationValue != NULL )                                    (5)
        {
            *pulNotificationValue = pxCurrentTCB->ulNotifiedValue;
        }
```

```
        if( pxCurrentTCB->ucNotifyState == taskWAITING_NOTIFICATION )          (6)
        {
            xReturn = pdFALSE;
        }
        else
        {
            pxCurrentTCB->ulNotifiedValue &= ~ulBitsToClearOnExit;             (7)
            xReturn = pdTRUE;
        }

        pxCurrentTCB->ucNotifyState = taskNOT_WAITING_NOTIFICATION;            (8)
    }
    taskEXIT_CRITICAL();
    return xReturn;
}
```

(1)、任务同通状态不为 taskNOTIFICATION_RECEIVED。

(2)、将任务通知值与参数 ulBitsToClearOnEntry 的取反值进行按位与运算。

(3)、任务通知状态改为 taskWAITING_NOTIFICATION。

(4)、如果阻塞时间大于 0 的话就要将任务添加到延时列表中，并且进行一次任务切换。

(5)、如果任务通知状态为 taskNOTIFICATION_RECEIVED，并且参数 pulNotificationValue 有效的话就保存任务通知值。

(6)、如果任务通知的状态又变为 taskWAITING_NOTIFICATION 的话就标记 xRetur 为 pdFALSE。

(7)、如果任务通知的状态一直为 taskNOTIFICATION_RECEIVED 的话就将任务通知的值与参数 ulBitsToClearOnExit 的取反值进行按位与运算，并且标记 xReturn 为 pdTRUE，表示获取任务通知成功。

(8)、标记任务通知的状态为 taskNOT_WAITING_NOTIFICATION。

17.5 任务通知模拟二值信号量实验

前面说了，根据 FreeRTOS 官方的统计，使用任务通知替代二值信号量的时候任务解除阻塞的时间要快 45%，并且需要的 RAM 也更少。其实通过我们上面分析任务通知发送和获取函数的过程可以看出，任务通知的代码量很少，所以执行时间与所需的 RAM 也就相应的会减少。

二值信号量就是值最大为 1 的信号量，这也是名字中“二值”的来源。当任务通知用于替代二值信号量的时候任务通知值就会替代信号量值，函数 ulTaskNotifyTake()就可以替代信号量获取函数 xSemaphoreTake()，函数 ulTaskNotifyTake()的参数 xClearCountOnExit 设置为 pdTRUE。这样在每次获取任务通知的时候模拟的信号量值就会清零。函数 xTaskNotifyGive()和 vTaskNotifyGiveFromISR()用于替代函数 xSemaphoreGive()和 xSemaphoreGiveFromISR()。接下来我们通过一个实验来演示一下任务通知是如何用作二值信号量的。

17.5.1 实验程序设计

1、实验目的

FreeRTOS 中的任务通知可以用来模拟二值信号量，本实验就来学习如何使用任务通知功

能模拟二值信号量。

2、实验设计

本实验是在“FreeRTOS 实验 14-1 FreeRTOS 二值信号量实验”的基础上修改而来的，只是将其中的二值信号量相关 API 函数改为任务通知的 API 函数。

3、实验工程

FreeRTOS 实验 17-1 FreeRTOS 任务通知模拟二值信号量。

4、实验程序与分析

本实验是在“FreeRTOS 实验 14-1 FreeRTOS 二值信号量实验”上修改而来的，所以几乎所有的代码都相同，只是将二值信号量的发送和获取换成了任务通知的发送和获取。我们就挑其中重要的代码讲解一下。

● **任务函数**

```
//DataProcess_task 函数
void DataProcess_task(void *pvParameters)
{
    u8 len=0;
    u8 CommandValue=COMMANDERR;
    u32 NotifyValue;

    u8 *CommandStr;
    POINT_COLOR=BLUE;
    while(1)
    {
        NotifyValue=ulTaskNotifyTake(pdTRUE,portMAX_DELAY);  //获取任务通知      (1)
        if(NotifyValue==1)      //清零之前的任务通知值为 1，说明任务通知有效      (2)
        {
            len=USART_RX_STA&0x3fff;                        //得到此次接收到的数据长度
            CommandStr=mymalloc(SRAMIN,len+1);              //申请内存
            sprintf((char*)CommandStr,"%s",USART_RX_BUF);
            CommandStr[len]='\0';                           //加上字符串结尾符号
            LowerToCap(CommandStr,len);                     //将字符串转换为大写
            CommandValue=CommandProcess(CommandStr);        //命令解析
            if(CommandValue!=COMMANDERR)
            {
                LCD_Fill(10,90,210,110,WHITE);              //清除显示区域
                LCD_ShowString(10,90,200,16,16,CommandStr);//在 LCD 上显示命令
                printf("命令为:%s\r\n",CommandStr);
                switch(CommandValue)                        //处理命令
                {
                    case LED1ON:
                        LED1=0;
                        break;
```

```
                    case LED1OFF:
                        LED1=1;
                        break;
                    case BEEPON:
                        PCF8574_WriteBit(BEEP_IO,0);
                        break;
                    case BEEPOFF:
                        PCF8574_WriteBit(BEEP_IO,1);
                        break;
                }
            }
            else
            {
                printf("无效的命令，请重新输入!!\r\n");
            }
            USART_RX_STA=0;
            memset(USART_RX_BUF,0,USART_REC_LEN);    //串口接收缓冲区清零
            myfree(SRAMIN,CommandStr);               //释放内存
        }
        else
        {
            vTaskDelay(10);         //延时 10ms，也就是 10 个时钟节拍
        }
    }
}
```

(1)、调用函数 ulTaskNotifyTake()获取本任务的任务通知，函数 ulTaskNotifyTake()的第一个参数设置为 pdTRUE。因为本实验是用任务通知模拟二值信号量的，所以需要在获取任务通知以后将任务通知值清零。

(2)、函数 ulTaskNotifyTake()的返回值就是清零之前的任务通知值，如果为 1 的话就说明任务通知值有效，相当于二值信号量有效。

● **中断处理**

任务通知的发送是在串口 1 的接收中断服务函数中完成的，中断服务程序如下：

```
extern TaskHandle_t DataProcess_Handler;; //接收任务通知的任务句柄

//串口 1 中断服务程序
void USART1_IRQHandler(void)
{
    u32 timeout=0;
    u32 maxDelay=0x1FFFF;
    BaseType_t xHigherPriorityTaskWoken;

    HAL_UART_IRQHandler(&UART1_Handler);  //调用 HAL 库中断处理公用函数
```

```
    timeout=0;
    while (HAL_UART_GetState(&UART1_Handler) != HAL_UART_STATE_READY)//等待就绪
    {
        timeout++;////超时处理
        if(timeout>maxDelay) break;
    }

    timeout=0;
    //一次处理完成之后，重新开启中断并设置 RxXferCount 为 1
    while(HAL_UART_Receive_IT(&UART1_Handler, (u8 *)aRxBuffer, RXBUFFERSIZE) !=\
                                        HAL_OK)
    {
        timeout++; //超时处理
        if(timeout>maxDelay) break;
    }

    //接收到数据，并且接收任务通知的任务有效
    if((USART_RX_STA&0x8000)&&(DataProcess_Handler!=NULL))
    {
        vTaskNotifyGiveFromISR(DataProcess_Handler,&xHigherPriorityTaskWoken);      (1)
        portYIELD_FROM_ISR(xHigherPriorityTaskWoken);//如果需要的话进行一次任务切换
    }
}
```

(1)、调用函数 vTaskNotifyGiveFromISR()向任务 DataProcess_task 发送任务通知，此处取代的是二值信号量发送函数。其他部分的代码和“FreeRTOS 实验 14-1 FreeRTOS 二值信号量实验”基本完全相同。

可以看出，使用任务通知替代二值信号量的过程还是非常简单的，只需要替换掉几个 API 函数，然后做简单的处理即可！

17.5.2 实验程序运行结果

参考实验“FreeRTOS 实验 14-1 FreeRTOS 二值信号量实验”。

17.6 任务通知模拟计数型信号量实验

不同与二值信号量，计数型信号量值可以大 1，这个最大值在创建信号量的时候可以设置。当计数型信号量有效的时候任务可以获取计数型信号量，信号量值只要大于 0 就表示计数型信号量有效。

当任务通知用作计数型信号量的时候获取信号量相当于获取任务通知值，使用函数 ulTaskNotifyTake()来替代函数 xSemaphoreTake()。函数 ulTaskNotifyTake()的参数 xClearOnExit 要设置为 pdFLASE，这样每次获取任务通知成功以后任务通知值就会减一。使用任务通知发送函数 xTaskNotifyGive() 和 vTaskNotifyGiveFromISR() 来替代计数型信号量释放函数 xSemaphoreGive()和 xSemaphoreGiveFromISR()。下面通过一个实验来演示一下任务通知是如何用作计数型信号量。

17.6.1 实验程序设计

1、实验目的

FreeRTOS 中的任务通知可以用来模拟计数型信号量，本实验就来学习如何使用任务通知功能模拟计数型信号量。

2、实验设计

本实验是在“FreeRTOS 实验 14-2 FreeRTOS 计数型信号量实验”的基础上修改而来的，只是将其中的计数型信号量相关 API 函数改为任务通知的 API 函数。

3、实验工程

FreeRTOS 实验 17-2 FreeRTOS 任务通知模拟计数型信号量。

4、实验程序与分析

本实验是在“FreeRTOS 实验 14-2 FreeRTOS 计数型信号量实验”上修改而来的，大部分的代码都是相同，我们就挑其中重要的代码讲解一下。

● **任务函数**

```
//释放计数型信号量任务函数
void SemapGive_task(void *pvParameters)
{
    u8 key,i=0;
    while(1)
    {
        key=KEY_Scan(0);                    //扫描按键
        if(SemapTakeTask_Handler!=NULL)
        {
            switch(key)
            {
                case WKUP_PRES:
                    xTaskNotifyGive(SemapTakeTask_Handler);//发送任务通知          (1)
                    break;
            }
        }
        i++;
        if(i==50)
        {
            i=0;
            LED0=!LED0;
        }
        vTaskDelay(10);         //延时 10ms，也就是 10 个时钟节拍
    }
}

//获取计数型信号量任务函数
```

```
void SemapTake_task(void *pvParameters)
{
    u8 num;
    uint32_t NotifyValue;
    while(1)
    {
        NotifyValue=ulTaskNotifyTake(pdFALSE,portMAX_DELAY);//获取任务通知      (2)
        num++;
        LCD_ShowxNum(166,111,NotifyValue-1,3,16,0);      //显示当前任务通知值      (3)
        LCD_Fill(6,131,233,313,lcd_discolor[num%14]);      //刷屏
        LED1=!LED1;
        vTaskDelay(1000);                          //延时 1s，也就是 1000 个时钟节拍
    }
}
```

(1)、以前发送计数型信号量的函数改为发送任务通知的函数 xTaskNotifyGive()。

(2)、调用任务通知获取函数 ulTaskNotifyTake()，由于我们是用任务通知来模拟计数型信号量的，所以函数 ulTaskNotifyTake()的第一个参数要设置为 pdFALSE。这样每获取成功一次任务通知，任务通知值就会减一，类似技术型信号量。函数 ulTaskNotifyTake()的返回值为任务通知值减少之前的值。

(3)、在 LCD 上显示当前任务通知值，注意，NotifyValue 要减一才是当前的任务通知值。

17.6.2 实验程序运行结果

参考实验“FreeRTOS 实验 14-1 FreeRTOS 二值信号量实验”。

17.7 任务通知模拟消息邮箱实验

任务通知也可用来向任务发送数据，但是相对于用队列发送消息，任务通知向任务发送消息会受到很多限制！

1、只能发送 32 位的数据值。

2、消息被保存为任务的任务通知值，而且一次只能保存一个任务通知值，相当于队列长度为 1。

因此说任务通知可以模拟一个轻量级的消息邮箱而不是轻量级的消息队列。任务通知值就是消息邮箱的值。

发送数据可以使用函数 xTaskNotify()或者 xTaskNotifyFromISR()，函数的参数 eAction 设置 eSetValueWithOverwrite 或者 eSetValueWithoutOverwrite。如果参数 eAction 为 eSetValueWithOverwrite 的话不管接收任务的通知值是否已经被处理，这个通知值都会被更新。参数 eAction 为 eSetValueWithoutOverwrite 的话如果上一个任务通知值话还没有被处理，那么新的任务通知值就不会更新。如果要读取任务通知值的话就使用函数 xTaskNotifyWait()。下面通过一个实验来演示一下任务通知是如何用作消息邮箱。

17.7.1 实验程序设计

1、实验目的

FreeRTOS 中的任务通知可以用来模拟消息邮箱，本实验就来学习如何使用任务通知功能

模拟消息邮箱。

2、实验设计

本实验设计三个任务：start_task、task1_task 、Keyprocess_task 这三个任务的任务功能如下：

start_task：用来创建其他 2 个任务。

task1_task ：读取按键的键值，然后将按键值作为任务通知发生给任务 Keyprocess_task。

Keyprocess_task：按键处理任务，读取任务通知值，根据不同的通知值做相应的处理。

实验需要三个按键 KEY_UP、KEY2 和 KEY0，不同的按键对应不同的按键值， 任务 task1_task()会将这些值作为任务通知发送给任务 Keyprocess_task。

本实验是在“FreeRTOS 实验 13-1 FreeRTOS 队列操作实验”的基础上修改而来的。

3、实验工程

FreeRTOS 实验 17-3 FreeRTOS 任务通知模拟消息邮箱实验。

4、实验程序与分析

●任务设置

```
#define START_TASK_PRIO             1          //任务优先级
#define START_STK_SIZE              256        //任务堆栈大小
TaskHandle_t StartTask_Handler;                //任务句柄
void start_task(void *pvParameters);           //任务函数

#define TASK1_TASK_PRIO             2          //任务优先级
#define TASK1_STK_SIZE              256        //任务堆栈大小
TaskHandle_t Task1Task_Handler;                //任务句柄
void task1_task(void *pvParameters);           //任务函数

#define KEYPROCESS_TASK_PRIO        3          //任务优先级
#define KEYPROCESS_STK_SIZE         256        //任务堆栈大小
TaskHandle_t Keyprocess_Handler;               //任务句柄
void Keyprocess_task(void *pvParameters);      //任务函数

//LCD 刷屏时使用的颜色
int lcd_discolor[14]={  WHITE,      BLACK,      BLUE,       BRED,
                        GRED,       GBLUE,      RED,        MAGENTA,
                        GREEN,      CYAN,       YELLOW,     BROWN,
                        BRRED,      GRAY };
```

● main()函数

```
int main(void)
{
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(360,25,2,8);   //设置时钟,180Mhz
    delay_init(180);                //初始化延时函数
    uart_init(115200);              //初始化串口
    LED_Init();                     //初始化 LED
```

```
    KEY_Init();                         //初始化按键
    PCF8574_Init();                     //初始化 PCF8574
    SDRAM_Init();                       //初始化 SDRAM
    LCD_Init();                         //初始化 LCD
    my_mem_init(SRAMIN);                //初始化内部内存池

    POINT_COLOR = RED;
    LCD_ShowString(10,10,200,16,16,"Apollo STM32F4/F7");
    LCD_ShowString(10,30,200,16,16,"FreeRTOS Examp 18-3");
    LCD_ShowString(10,50,200,16,16,"Task Notify Maibox");
    LCD_ShowString(10,70,220,16,16,"KEY_UP:LED1    KEY2:BEEP");
    LCD_ShowString(10,90,200,16,16,"KEY0:Refresh LCD");

    POINT_COLOR = BLACK;
    LCD_DrawRectangle(5,125,234,314);  //画矩形

    //创建开始任务
    xTaskCreate((TaskFunction_t     )start_task,            //任务函数
                (const char*        )"start_task",          //任务名称
                (uint16_t           )START_STK_SIZE,        //任务堆栈大小
                (void*              )NULL,                  //传递给任务函数的参数
                (UBaseType_t        )START_TASK_PRIO,       //任务优先级
                (TaskHandle_t*      )&StartTask_Handler);   //任务句柄
    vTaskStartScheduler();                                  //开启任务调度
}
```

● **任务函数**

```
//开始任务任务函数
void start_task(void *pvParameters)
{
    taskENTER_CRITICAL();           //进入临界区

    //创建 TASK1 任务
    xTaskCreate((TaskFunction_t     )task1_task,
                (const char*        )"task1_task",
                (uint16_t           )TASK1_STK_SIZE,
                (void*              )NULL,
                (UBaseType_t        )TASK1_TASK_PRIO,
                (TaskHandle_t*      )&Task1Task_Handler);
    //创建按键处理任务
    xTaskCreate((TaskFunction_t     )Keyprocess_task,
                (const char*        )"keyprocess_task",
                (uint16_t           )KEYPROCESS_STK_SIZE,
                (void*              )NULL,
```

```
                (UBaseType_t        )KEYPROCESS_TASK_PRIO,
                (TaskHandle_t*      )&Keyprocess_Handler);
    vTaskDelete(StartTask_Handler);         //删除开始任务
    taskEXIT_CRITICAL();                    //退出临界区
}

//task1 任务函数
void task1_task(void *pvParameters)
{
    u8 key,i=0;
    BaseType_t err;
    while(1)
    {
        key=KEY_Scan(0);            //扫描按键
        if((Keyprocess_Handler!=NULL)&&(key))
        {
            err=xTaskNotify((TaskHandle_t   )Keyprocess_Handler,        //任务句柄    (1)
                            (uint32_t       )key,                       //任务通知值
                            (eNotifyAction  )eSetValueWithOverwrite);   //覆写的方式
            if(err==pdFAIL)
            {
                printf("任务通知发送失败\r\n");
            }
        }
        i++;
        if(i==50)
        {
            i=0;
            LED0=!LED0;
        }
        vTaskDelay(10);             //延时 10ms，也就是 10 个时钟节拍
    }
}

//Keyprocess_task 函数
void Keyprocess_task(void *pvParameters)
{
    u8 num,beepsta=1;
    uint32_t NotifyValue;
    BaseType_t err;
    while(1)
    {
        err=xTaskNotifyWait((uint32_t           )0x00,    //进入函数的时候不清除任务 bit  (2)
```

```
                                (uint32_t          )ULONG_MAX,//退出函数的时候清除所有的 bit
                                (uint32_t*         )&NotifyValue, //保存任务通知值
                                (TickType_t        )portMAX_DELAY);    //阻塞时间
        if(err==pdTRUE)                     //获取任务通知成功
        {
            switch((u8)NotifyValue)
            {
                case WKUP_PRES:     //KEY_UP 控制 LED1
                    LED1=!LED1;
                    break;
                case KEY2_PRES:     //KEY2 控制蜂鸣器
                    beepsta=!beepsta;
                    PCF8574_WriteBit(BEEP_IO,beepsta);
                    break;
                case KEY0_PRES:     //KEY0 刷新 LCD 背景
                    num++;
                    LCD_Fill(6,126,233,313,lcd_discolor[num%14]);
                    break;
            }
        }
    }
}
```

(1)、调用函数 xTaskNotify()发送任务通知，任务通知值为按键值，发送方式采用覆写的方式。

(2)、调用函数 xTaskNotifyWait()获取任务通知，获取到的任务通知其实就是(1)中发送的按键值，然后根据不同的按键值做不同的处理。

17.7.2 实验程序运行结果

编译并下载实验代码到开发板中，通过按键来控制 LED1 和蜂鸣器的开关、LCD 指定区域背景色的刷新。本实验的运行结果和“FreeRTOS 实验 13-1 FreeRTOS 队列操作实验”类似，只是不能接收串口发送过来的数据而已。

17.8 任务通知模拟事件标志组实验

事件标志组其实就是一组二进制事件标志(位)，每个事件标志位的具体意义由应用程序编写者来决定。当一个任务等待事件标志组中的某几个标志(位)的时候可以进入阻塞态，当任务因为等待事件标志(位)而进入阻塞态以后这个任务就不会消耗 CPU。

当任务通知用作事件标志组的话任务通知值就相当于事件组，这个时候任务通知值的每个 bit 用作事件标志(位)。函数 xTaskNotifyWait()替代事件标志组中的 API 函数 xEventGroupWaitBits()。函数 xTaskNotify()和 xTaskNotifyFromISR()(函数的参数 eAction 为 eSetBits)替代事件标志组中的 API 函数 xEventGroupSetBits()和 xEventGroupSetBitsFromISR()。下面通过一个实验来演示一下任务通知是如何用作事件标志组。

17.8.1 实验程序设计

1、实验目的

FreeRTOS 中的任务通知可以用来模拟事件标志组，本实验就来学习如何使用任务通知功能模拟事件标志组。

2、实验设计

本实验是在“FreeRTOS 实验 16-1 FreeRTOS 事件标志组实验”的基础上修改而来的。

3、实验工程

FreeRTOS 实验 17-4 FreeRTOS 任务通知模拟事件标志组实验。

4、实验程序与分析

本实验是在“FreeRTOS 实验 16-1 FreeRTOS 事件标志组实验”上修改而来的，大部分的代码都是相同，我们就挑其中重要的代码讲解一下:

● **任务函数**

```
//开始任务任务函数
void start_task(void *pvParameters)
{
    taskENTER_CRITICAL();           //进入临界区
    //创建设置事件位的任务
    xTaskCreate((TaskFunction_t     )eventsetbit_task,
                (const char*        )"eventsetbit_task",
                (uint16_t           )EVENTSETBIT_STK_SIZE,
                (void*              )NULL,
                (UBaseType_t        )EVENTSETBIT_TASK_PRIO,
                (TaskHandle_t*      )&EventSetBit_Handler);
    //创建事件标志组处理任务
    xTaskCreate((TaskFunction_t     )eventgroup_task,
                (const char*        )"eventgroup_task",
                (uint16_t           )EVENTGROUP_STK_SIZE,
                (void*              )NULL,
                (UBaseType_t        )EVENTGROUP_TASK_PRIO,
                (TaskHandle_t*      )&EventGroupTask_Handler);
    vTaskDelete(StartTask_Handler);     //删除开始任务
    taskEXIT_CRITICAL();            //退出临界区
}

//设置事件位的任务
void eventsetbit_task(void *pvParameters)
{
    u8 key,i;
    while(1)
    {
```

```
        if(EventGroupTask_Handler!=NULL)
        {
            key=KEY_Scan(0);
            switch(key)
            {
                case KEY1_PRES:
                    xTaskNotify((TaskHandle_t   )EventGroupTask_Handler,        (1)
                                (uint32_t       )EVENTBIT_1,
                                (eNotifyAction  )eSetBits);
                    break;
                case KEY2_PRES:
                    xTaskNotify((TaskHandle_t   )EventGroupTask_Handler,        (2)
                                (uint32_t       )EVENTBIT_2,
                                (eNotifyAction  )eSetBits);
                    break;
            }
        }
        i++;
        if(i==50)
        {
            i=0;
            LED0=!LED0;
        }
        vTaskDelay(10); //延时 10ms，也就是 10 个时钟节拍
    }
}

//事件标志组处理任务
void eventgroup_task(void *pvParameters)
{
    u8 num=0,enevtvalue;
    static u8 event0flag,event1flag,event2flag;
    uint32_t NotifyValue;
    BaseType_t err;

    while(1)
    {
        //获取任务通知值
        err=xTaskNotifyWait((uint32_t   )0x00,          //进入函数的时候不清除任务 bit  (3)
                            (uint32_t   )ULONG_MAX,     //退出函数的时候清除所有的 bit
                            (uint32_t*  )&NotifyValue,  //保存任务通知值
                            (TickType_t )portMAX_DELAY);//阻塞时间
```

```
        if(err==pdPASS)        //任务通知获取成功
        {
            if((NotifyValue&EVENTBIT_0)!=0)          //事件 0 发生        (4)
            {
                event0flag=1;
            }
            else if((NotifyValue&EVENTBIT_1)!=0)     //事件 1 发生
            {
                event1flag=1;
            }
            else if((NotifyValue&EVENTBIT_2)!=0)     //事件 2 发生
            {
                event2flag=1;
            }

            enevtvalue=event0flag|(event1flag<<1)|(event2flag<<2);    //模拟事件标志组值(5)
            printf("任务通知值为:%d\r\n",enevtvalue);
            LCD_ShowxNum(174,110,enevtvalue,1,16,0);          //在 LCD 上显示当前事件值

            if((event0flag==1)&&(event1flag==1)&&(event2flag==1))//三个事件都同时发生(6)
            {
                num++;
                LED1=!LED1;
                LCD_Fill(6,131,233,313,lcd_discolor[num%14]);
                event0flag=0;                                 //标志清零              (7)
                event1flag=0;
                event2flag=0;
            }
        }
        else
        {
            printf("任务通知获取失败\r\n");
        }
    }
}
```

(1)、按下 KEY1 键以后调用函数 xTaskNotify()发生任务通知，因为是模拟事件标志组的，所以这里设置的是将任务通知值的 bit1 置 1。

(2)、按下 KEY2 键以后调用函数 xTaskNotify()发生任务通知，因为是模拟事件标志组的，所以这里设置的是将任务通知值的 bit2 置 1。

(3)、调用函数 xTaskNotifyWait()获取任务通知值，相当于获取事件标志组值。

(4)、根据 NotifyValue 来判断是哪个事件发生了，如果是事件 0 发生的话(也就是任务通知值的 bit0 置一)就给 event0flag 赋 1，表示事件 0 发生了。因为任务通知不像事件标志组那样可以同时等待多个事件位。只要有任何一个事件位置 1，函数 xTaskNotifyWait()就会因为获取任务

通知成功而在退出函数的时候将这个事件位清零(函数 xTaskNotifyWait()的参数设置的)。所以这里我们要对每个发生的事件做个标志，比如本试验中用到了三个事件标志位：EVENTBIT_0、EVENTBIT_1、EVENTBIT_2。这三个事件标志位代表三个事件，每个事件都有一个变量来记录此事件是否已经发生，这三个变量在任务函数 eventgroup_task()中有定义，它们分别为：event0flag、event1flag 和 event2flag。当这三个变量都为 1 的时候就说明三个事件都发生了，这不就模拟出来了事件标志组的同时等待多个事件位吗？

(5)、通过这三个事件发生标志来模拟事件标志组值。

(6)、所等待的三个事件都发生了，执行相应的处理。

(7)、处理完成以后将事件发生标志清零。

● **中断处理过程**

```
//事件标志组句柄
extern TaskHandle_t EventGroupTask_Handler;

//中断服务函数
void EXTI3_IRQHandler(void)
{
    BaseType_t xHigherPriorityTaskWoken;

    delay_xms(50);                              //消抖
    if(KEY0==0)
    {
        xTaskNotifyFromISR((TaskHandle_t   )EventGroupTask_Handler,  //任务句柄    (1)
                           (uint32_t       )EVENTBIT_0,              //要更新的 bit
                           (eNotifyAction  )eSetBits,                //更新指定的 bit
                           (BaseType_t*    )xHigherPriorityTaskWoken);

        portYIELD_FROM_ISR(xHigherPriorityTaskWoken);

    }
    __HAL_GPIO_EXTI_CLEAR_IT(GPIO_PIN_3);     //清除中断标志位
}
```

(1)、中断服务函数中调用 xTaskNotifyFromISR()来发送任务通知。

17.8.2 实验程序运行结果

参考实验“FreeRTOS 实验 16-1 FreeRTOS 事件标志组实验”。

第十八章 FreeRTOS 低功耗 Tickless 模式

很多应用场合对于空耗的要求很严格，比如长期无人照看的数据采集仪器，可穿戴设备等。其实很多 MCU 都有相应的低功耗模式，以此来降低设备运行时的功耗，进行裸机开发的时候就可以使用这些低功耗模式。但是现在我们要使用操作系统，因此操作系统对于低功耗的支持也显得尤为重要，这样硬件与软件相结合，可以进一步降低系统的功耗。这样开发也会方便很多，毕竟系统已经原生支持低功耗了，我们只需要按照系统的要求来做编写相应的应用层代码即可。FreeRTOS 提供了一个叫做 Tickless 的低功耗模式，本章我们就来学习一下如何使用这个 Tickless 模式，本章分为如下几部分：

18.1 STM32F4 低功耗模式

18.2 Tickless 模式详解

18.3 低功耗 Tickless 模式实验

18.1 STM32F4 低功耗模式

STM32 本身就支持低功耗模式，以本教程使用的 STM32F429 为例，共有三种低功耗模式：

- 睡眠(Sleep)模式。
- 停止(Stop)模式。
- 待机(Standby)模式。

这三种模式对比如表 18.1.1 所示：

模式名称	进入	唤醒	对 1.2V 域时钟的影响	对 VDD 域时钟的影响	调压器
睡眠(立即休眠或退出是休眠)	WFI	任意中断	CPU CLK 关闭对其它时钟或模拟时钟源无影响	无	开启
	WFE	唤醒事件			
停止	PDDS 和 LPDS 位+SLEEPDEEP 位+WFI 或 WFE	任意 EXTI 线(在 EXTI 寄存器中配置，内部线和外部线)	所有 1.2 V 域时钟都关闭	HSI 和 HSE 振荡器关闭	开启或处于低功耗模式
待机	PDDS 位+SLEEPDEEP 位+WFI 或 WFE	WKUP 引脚上升沿、RTC 闹钟(闹钟 A 或闹钟 B)RTC 唤醒事件、RTC 入侵事件、RTC 时间戳事件、NRST 引脚外部复位、IWDG 复位	所有 1.2 V 域时钟都关闭	HSI 和 HSE 振荡器关闭	关闭

表 18.1.1 低功耗模式对比

这三种低功耗模式对应三种不同的功耗水平，根据实际的应用环境选择相对应的低功耗模式。接下来我们就详细的看一下这三者有何区别。

18.1.1 睡眠(Sleep)模式

- **进入睡眠模式**

进入睡眠模式有两种指令：WFI(等待中断)和 WFE(等待事件)。根据 Cortex-M 内核的 SCR(系统控制)寄存器可以选择使用立即休眠还是退出时休眠，当 SCR 寄存器的 SLEEPONEXIT(bit1)位为 0 的时候使用立即休眠，当为 1 的时候使用退出时休眠。关于立即休眠和退出时休眠的详细内容请参考《权威指南》“第 9 章 低功耗和系统控制特性”章节。

CMSIS(Cortex 微控制器软件接口标准)提供了两个函数来操作指令 WFI 和 WFE，我们可以直接使用这两个函数：__WFI 和__WFE。FreeRTOS 系统会使用 WFI 指令进入休眠模式。

- **退出休眠模式**

如果使用 WFI 指令进入休眠模式的话那么任意一个中断都会将 MCU 从休眠模式中唤醒，如果使用 WFE 指令进入休眠模式的话那么当有事件发生的话就会退出休眠模式，比如配置一

个 EXIT 线作为事件。

当 STM32F429 处于休眠模式的时候 Cortex-M4F 内核停止运行，但是其他外设运行正常，比如 NVIC、SRAM 等。休眠模式的功耗比其他两个高，但是休眠模式没有唤醒延时，应用程序可以立即运行。

18.1.2 停止(Stop)模式

停止模式基于 Cortex-M4F 的深度休眠模式与外设时钟门控，在此模式下 1.2V 域的所有时钟都会停止，PLL、HSI 和 HSE RC 振荡器会被禁止，但是内部 SRAM 的数据会被保留。调压器可以工作在正常模式，也可配置为低功耗模式。如果有必要的话可以通过将 PWR_CR 寄存器的 FPDS 位置 1 来使 Flash 在停止模式的时候进入掉电状态，当 Flash 处于掉电状态的时候 MCU 从停止模式唤醒以后需要更多的启动延时。停止模式的进入和退出如表 18.1.2.1 所示：

停止模式	描述
进入模式	WFI(等待中断)或 WF(等待事件)，且： — 将 Cortex-M4F 系统控制寄存器中的 SLEEPDEEP 位置 1 — 将电源控制寄存器 (PWR_CR) 中的 PDDS 位清零 — 通过配置 PWR_CR 中的 LPDS 位选择调压器模式。 注意：要进入停止模式，所有 EXTI 线挂起位、RTC 闹(闹钟 A 和闹钟 B)、RTC 唤醒、RTC 入侵和 RTC 时间戳标志必须复位。否则将忽略进入停止模式这一过程，继续执行程序。
退出模式	如果使用 WFI 进入： 所有配置为中断模式的 EXTI 线(必须在 NVIC 中使能对应的 EXTI 中断向量)。 如果使用 WFE 进入： 所有配置为事件模式的 EXTI 线。

表 18.1.2.1 进入和退出停止模式

18.1.3 待机(Standby)模式

相比于前面两种低功耗模式，待机模式的功耗最低。待机模式是基于 Cortex-M4F 的深度睡眠模式的，其中调压器被禁止。1.2V 域断电，PLL、HSI 振荡器和 HSE 振荡器也被关闭。除了备份区域和待机电路相关的寄存器外，SRAM 和其他寄存器的内容都将丢失。待机模式的进入和退出如表 18.1.3.1 所示：

待机模式	描述
进入模式	WFI(等待中断)或 WFE(等待事件)，且： — 将 Cortex-M4F 系统控制寄存器中的 SLEEPDEEP 位置 1 — 将电源控制寄存器 (PWR_CR) 中的 PDDS 位置 1 — 将电源控制/状态寄存器 (PWR_CSR) 中的 WUF 位清零 — 将与所选唤醒源（RTC 闹钟 A、RTC 闹钟 B、RTC 唤醒、RTC 入侵或 RTC 时间戳标志）对应的 RTC 标志清零
退出模式	WKUP 引脚上升沿、RTC 闹钟(闹钟 A 和闹钟 B)、RTC 唤醒事件、RTC 入侵事件、RTC 时间戳事件、NRST 引脚外部复位和 IWDG 复位。

表 18.1.3.1 进入和退出待机模式

退出待机模式的话会导致 STM32F4 重启，所以待机模式的唤醒延时也是最大的。实际应

用中要根据使用环境和要求选择合适的待机模式。关于 STM32 低功耗模式的详细介绍和使用请参考 ST 官方的参考手册。

18.2 Tickless 模式详解

18.2.1 如何降低功耗？

在 11.4 小节讲解获取任务运行时间信息的时候可以看出，一般的简单应用中处理器大量的时间都在处理空闲任务，所以我们就可以考虑当处理器处理空闲任务的时候就进入低功耗模式，当需要处理应用层代码的时候就将处理器从低功耗模式唤醒。FreeRTOS 就是通过在处理器处理空闲任务的时候将处理器设置为低功耗模式来降低能耗。一般会在空闲任务的钩子函数中执行低功耗相关处理，比如设置处理器进入低功耗模式、关闭其他外设时钟、降低系统主频等等。本章后面均以 STM32F429 三个低功耗模式中的睡眠模式为例讲解。

我们知道 FreeRTOS 的系统时钟是由滴答定时器中断来提供的，系统时钟频率越高，那么滴答定时器中断频率也就越高。18.1 小节讲过，中断是可以将 STM32F429 从睡眠模式中唤醒，周期性的滴答定时器中断就会导致 STM32F429 周期性的进入和退出睡眠模式。因此，如果滴答定时器中断频率太高的话会导致大量的能量和时间消耗在进出睡眠模式中，这样导致的结果就是低功耗模式的作用被大大的削弱。

为此，FreeRTOS 特地提供了一个解决方法——Tickless 模式，当处理器进入空闲任务周期以后就关闭系统节拍中断(滴答定时器中断)，只有当其他中断发生或者其他任务需要处理的时候处理器才会被从低功耗模式中唤醒。为此我们将面临两个问题：

问题一：关闭系统节拍中断会导致系统节拍计数器停止，系统时钟就会停止。

FreeRTOS 的系统时钟是依赖于系统节拍中断(滴答定时器中断)的，如果关闭了系统节拍中断的话就会导致系统时钟停止运行，这是绝对不允许的！该如何解决这个问题呢？我们可以记录下系统节拍中断的关闭时间，当系统节拍中断再次开启运行的时候补上这段时间就行了。这时候我们就需要另外一个定时器来记录这段该补上的时间，如果使用专用的低功耗处理器的话基本上都会有一个低功耗定时器，比如 STM32L4 系列(L 系列是 ST 的低功耗处理器)就有一个叫做 LPTIM(低功耗定时器)的定时器。STM32F429 没有这种定时器那么就接着使用滴答定时器来完成这个功能，具体实现方法后面会讲解。

问题二：如何保证下一个要运行的任务能被准确的唤醒？

即使处理器进入了低功耗模式，但是我的中断和应用层任务也要保证及时的响应和处理。中断自然不用说，本身就可以将处理器从低功耗模式中唤醒。但是应用层任务就不行了，它无法将处理器从低功耗模式唤醒，无法唤醒就无法运行！这个问题看来很棘手，既然应用层任务无法将处理器从低功耗模式唤醒，那么我们就借助其他的力量来完成这个功能。如果处理器在进入低功耗模式之前能够获取到还有多长时间运行下一个任务那么问题就迎刃而解了，我们只需要开一个定时器，定时器的定时周期设置为这个时间值就行了，定时时间到了以后产生定时中断，处理器不就从低功耗模式唤醒了。这里似乎又引出了一个新的问题，那就是如何知道还有多长时间执行下一个任务？这个时间也就是低功耗模式的执行时间，值得庆幸的是 FreeRTOS 已经帮我们完成了这个工作。

18.2.2 Tickless 具体实现

1、宏 configUSE_TICKLESS_IDLE

要想使用 Tickless 模式，首先必须将 FreeRTOSConfig.h 中的宏 configUSE_TICKLESS_IDLE 设置为 1，代码如下：

```
#define configUSE_TICKLESS_IDLE        1                    //1 启用低功耗 tickless 模式
```

2、宏 portSUPPRESS_TICKS_AND_SLEEP()

使能 Tickless 模式以后当下面两种情况都出现的时候 FreeRTOS 内核就会调用宏 portSUPPRESS_TICKS_AND_SLEEP()来处理低功耗相关的工作。

- 空闲任务是唯一可运行的任务，因为其他所有的任务都处于阻塞态或者挂起态。
- 系统处于低功耗模式的时间至少大于 configEXPECTED_IDLE_TIME_BEFORE_SLEEP 个时钟节拍，宏 configEXPECTED_IDLE_TIME_BEFORE_SLEEP 默认在文件 FreeRTOS.h 中定义为 2，我们可以在 FreeRTOSConfig.h 中重新定义，此宏必须大于 2！

portSUPPRESS_TICKS_AND_SLEEP()有个参数，此参数用来指定还有多长时间将有任务进入就绪态，其实就是处理器进入低功耗模式的时长(单位为时钟节拍数)，因为一旦有其他任务进入就绪态处理器就必须退出低功耗模式去处理这个任务。portSUPPRESS_TICKS_AND_SLEEP()应该是由用户根据自己所选择的平台来编写的，此宏会被空闲任务调用来完成具体的低功耗工作。但是！如果使用 STM32 的话编写这个宏的工作就不用我们来完成了，因为 FreeRTOS 已经帮我们做好了，有没有瞬间觉得好幸福啊。当然了你也可以自己去重新编写，不使用 FreeRTOS 提供的，如果自己编写的话需要先将 configUSE_TICKLESS_IDLE 设置为 2。

宏 portSUPPRESS_TICKS_AND_SLEEP 在文件 portmacro.h 中如下定义：

```
#ifndef portSUPPRESS_TICKS_AND_SLEEP
    extern void vPortSuppressTicksAndSleep( TickType_t xExpectedIdleTime );
    #define    portSUPPRESS_TICKS_AND_SLEEP( xExpectedIdleTime )
               vPortSuppressTicksAndSleep( xExpectedIdleTime )
#endif
```

从上面的代码可以看出 portSUPPRESS_TICKS_AND_SLEEP() 的本质就是函数 vPortSuppressTicksAndSleep()，此函数在文件 port.c 中有如下定义：

```
__weak void vPortSuppressTicksAndSleep( TickType_t    xExpectedIdleTime )
{
    uint32_t  ulReloadValue,      ulCompleteTickPeriods,      ulCompletedSysTickDecrements,\
              ulSysTickCTRL;
    TickType_t xModifiableIdleTime;

    //确保滴答定时器的 Reload(重装载)值不会溢出，也就是不能超过滴答定时器最大计数值。
    if( xExpectedIdleTime > xMaximumPossibleSuppressedTicks )                    (1)
    {
        xExpectedIdleTime = xMaximumPossibleSuppressedTicks;
    }

    //停止滴答定时器。
    portNVIC_SYSTICK_CTRL_REG &= ~portNVIC_SYSTICK_ENABLE_BIT;

    //根据参数 xExpectedIdleTime 来计算滴答定时器的重装载值。
    ulReloadValue = portNVIC_SYSTICK_CURRENT_VALUE_REG +\                        (2)
```

```
                    ( ulTimerCountsForOneTick * ( xExpectedIdleTime - 1UL ) );
if( ulReloadValue > ulStoppedTimerCompensation )                    (3)
{
    ulReloadValue -= ulStoppedTimerCompensation;
}

__disable_irq();                                                    (4)
__dsb( portSY_FULL_READ_WRITE );
__isb( portSY_FULL_READ_WRITE );

//确认是否可以进入低功耗模式
if( eTaskConfirmSleepModeStatus() == eAbortSleep )                  (5)
{
    //不能进入低功耗模式，重新启动滴答定时器
    portNVIC_SYSTICK_LOAD_REG = portNVIC_SYSTICK_CURRENT_VALUE_REG;
    portNVIC_SYSTICK_CTRL_REG |= portNVIC_SYSTICK_ENABLE_BIT;
    portNVIC_SYSTICK_LOAD_REG = ulTimerCountsForOneTick - 1UL;
    __enable_irq();                                                 (6)
}
else                                                                (7)
{
    //可以进入低功耗模式，设置滴答定时器
    portNVIC_SYSTICK_LOAD_REG = ulReloadValue;                      (8)
    portNVIC_SYSTICK_CURRENT_VALUE_REG = 0UL;
    portNVIC_SYSTICK_CTRL_REG |= portNVIC_SYSTICK_ENABLE_BIT;

    xModifiableIdleTime = xExpectedIdleTime;
    configPRE_SLEEP_PROCESSING( xModifiableIdleTime );              (9)
    if( xModifiableIdleTime > 0 )
    {
        __dsb( portSY_FULL_READ_WRITE );
        __wfi();                                                    (10)
        __isb( portSY_FULL_READ_WRITE );
    }

    //当代码执行到这里的时候说明已经退出了低功耗模式!
    configPOST_SLEEP_PROCESSING( xExpectedIdleTime );               (11)

    //停止滴答定时器
    ulSysTickCTRL = portNVIC_SYSTICK_CTRL_REG;                      (12)
    portNVIC_SYSTICK_CTRL_REG = ( ulSysTickCTRL &\
                                  ~portNVIC_SYSTICK_ENABLE_BIT );
```

```
        __enable_irq();                                                    (13)

        //判断导致退出低功耗的是由外部中断引起的还是滴答定时器计时时间到引起的
        if( ( ulSysTickCTRL & portNVIC_SYSTICK_COUNT_FLAG_BIT ) != 0 )     (14)
        {
            uint32_t    ulCalculatedLoadValue;
            ulCalculatedLoadValue = ( ulTimerCountsForOneTick - 1UL ) - ( ulReloadValue -\
                                    portNVIC_SYSTICK_CURRENT_VALUE_REG );

            if( ( ulCalculatedLoadValue < ulStoppedTimerCompensation ) ||\
                ( ulCalculatedLoadValue > ulTimerCountsForOneTick ) )
            {
                ulCalculatedLoadValue = ( ulTimerCountsForOneTick - 1UL );
            }
            portNVIC_SYSTICK_LOAD_REG = ulCalculatedLoadValue;
            ulCompleteTickPeriods = xExpectedIdleTime - 1UL;
        }
        else        //外部中断唤醒的，需要进行时间补偿
        {
            ulCompletedSysTickDecrements = ( xExpectedIdleTime * \
                                            ulTimerCountsForOneTick ) -\
                                            portNVIC_SYSTICK_CURRENT_VALUE_REG;

            ulCompleteTickPeriods = ulCompletedSysTickDecrements / \
                                    ulTimerCountsForOneTick;

            portNVIC_SYSTICK_LOAD_REG = ( ( ulCompleteTickPeriods + 1UL ) * \
                                ulTimerCountsForOneTick ) - ulCompletedSysTickDecrements;
        }

        //重新启动滴答定时器，滴答定时器的重装载值设置为正常值。
        portNVIC_SYSTICK_CURRENT_VALUE_REG = 0UL;
        portENTER_CRITICAL();
        {
            portNVIC_SYSTICK_CTRL_REG |= portNVIC_SYSTICK_ENABLE_BIT;
            vTaskStepTick( ulCompleteTickPeriods );                        (15)
            portNVIC_SYSTICK_LOAD_REG = ulTimerCountsForOneTick - 1UL;
        }
        portEXIT_CRITICAL();
    }
}
```

(1)、参数 xExpectedIdleTime 表示处理器将要在低功耗模式运行的时长(单位为时钟节拍数)，这个时间会使用滴答定时器来计时，但是滴答定时器的计数寄存器是 24 位的，因此这个时间值

不能超过滴答定时器的最大计数值。xMaximumPossibleSuppressedTicks 是个静态全局变量，在文件 port.c 中有定义，此变量会在函数 vPortSetupTimerInterrupt()中被重新赋值，代码如下：

```
ulTimerCountsForOneTick = ( configSYSTICK_CLOCK_HZ / configTICK_RATE_HZ );
xMaximumPossibleSuppressedTicks = portMAX_24_BIT_NUMBER / ulTimerCountsForOneTick;
```

经过计算 xMaximumPossibleSuppressedTicks=0xffffff/(180000000/1000)≈93，因此可以得出进入低功耗模式的最大时长为 93 个时钟节拍，注意！这个值要根据自己所使用的平台以及 FreeRTOS 的实际配置情况来计算。

(2)、根据参数 xExpectedIdleTime 来计算滴答定时器的重装载值，因为处理器进入低功耗模式以后的计时是由滴答定时器来完成的。

(3)、从滴答定时器停止运行到把统计得到的低功耗模式运行的这段时间补偿给 FreeRTOS 系统时钟也是需要时间的，这期间也是有程序在运行的。这段程序运行的时间我们要留出来，具体的时间没法去统计，因为平台不同、编译器的代码优化水平不同导致了程序的执行时间也不同。这里只能大概的留出一个时间值，这个时间值由变量 ulStoppedTimerCompensation 来确定，这是一个全局变量，在文件 port.c 中有定义。此变量也会在函数 vPortSetupTimerInterrupt()中被重新赋值，代码如下：

```
#define portMISSED_COUNTS_FACTOR  ( 45UL )

ulStoppedTimerCompensation = portMISSED_COUNTS_FACTOR / ( configCPU_CLOCK_HZ / \
                                                          configSYSTICK_CLOCK_HZ );
```

由上面的公式可以得出：ulStoppedTimerCompensation=45/(180000000/180000000)=45。如果要修改这个时间值的话直接修改宏 portMISSED_COUNTS_FACTOR 即可。

(4)、在执行 WFI 前设置寄存器 PRIMASK 的话处理器可以由中断唤醒但是不会处理这些中断，退出低功耗模式以后通过清除寄存器 PRIMASK 来使 ISR 得到执行，其实就是利用 PRIMASK 来延迟 ISR 的执行。函数__disable_irq()是用来设置寄存器 PRIMASK 的，清除寄存器 PRIMASK 使用函数__enable_irq()，这两个函数是由 CMSIS-Core 提供的，如果所使用的例程添加了 CMSIS 相关文件的话就可以直接拿来用，ALIENTEK 的所有例程都添加了。

(5)、调用函数 eTaskConfirmSleepModeStatus()来判断是否可以进入低功耗模式，此函数在文件 tasks.c 中有定义。此函数通过检查是否还有就绪任务来决定处理器能不能进入低功耗模式，如果返回 eAbortSleep 的话就表示不能进入低功耗模式，既然不能进入低功耗那就需要重新恢复滴答定时器的运行。

(6)、调用函数__enable_irq()重新打开中断，因为在(4)中我们调用函数__disable_irq()关闭了中断。

(7)、可以进入低功耗模式，完成低功耗相关设置。

(8)、进入低功耗模式的时间已经在(2)中计算出来了，这里将这个值写入到滴答定时器的重装载寄存器中。

(9)、configPRE_SLEEP_PROCESSING()是个宏，在进入低功耗模式之前可能有一些其他的事情要处理，比如降低系统时钟、关闭外设时钟、关闭板子某些硬件的电源等等，这些操作就可以在这个宏中完成，后面会讲解这个宏如何使用。

(10)、使用 WFI 指令使 STM32F429 进入睡眠模式。

(11)、代码执行到这里说明处理器已经退出了低功耗模式，退出低功耗模式以后也可能需要处理一些事情。比如恢复系统时钟，使能外设时钟，打开板子某些硬件的电源等等，这些操作在宏 configPOST_SLEEP_PROCESSING()中完成，后面会讲解这个宏如何使用。

(12)、读取滴答定时器 CTRL(控制和状态)寄存器，后面要用。

(13)、调用函数__enable_irq()打开中断。

(14)、判断退出低功耗模式是由滴答定时器中断引起的还是由其他中断引起的，因为这两种原因所对应的系统时钟赔偿值的计算方法不同，这个系统时钟补偿值的单位是时钟节拍。

(15)、调用函数 vTaskStepTick()补偿系统时钟，函数参数是要补偿的值，此函数在文件 tasks.c 中有如下定义：

```
void vTaskStepTick( const TickType_t xTicksToJump )
{
    configASSERT( ( xTickCount + xTicksToJump ) <= xNextTaskUnblockTime );
    xTickCount += xTicksToJump;
    traceINCREASE_TICK_COUNT( xTicksToJump );
}
```

可以看出，此函数很简单，只是给 FreeRTOS 的系统时钟节拍计数器 xTickCount 加上一个补偿值而已。

3、宏 configPRE_SLEEP_PROCESSING ()和 configPOST_SLEEP_PROCESSING()

在真正的低功耗设计中不仅仅是将处理器设置到低功耗模式就行了，还需要做一些其他的处理，比如：

● 将处理器降低到合适的频率，因为频率越低功耗越小，甚至可以在进入低功耗模式以后关闭系统时钟。

● 修改时钟源，晶振的功耗肯定比处理器内部的时钟源高，进入低功耗模式以后可以切换到内部时钟源，比如 STM32 的内部 RC 振荡器。

● 关闭其他外设时钟，比如 IO 口的时钟。

● 关闭板子上其他功能模块电源，这个需要在产品硬件设计的时候就要处理好，比如可以通过 MOS 管来控制某个模块电源的开关，在处理器进入低功耗模式之前关闭这些模块的电源。

有关产品低功耗设计的方法还有很多，大家可以上网查找一下，上面列举出的这几点在处理器进入低功耗模式之前就要完成处理。FreeRTOS 为我们提供了一个宏来完成这些操作，它就是 configPRE_SLEEP_PROCESSING()，这个宏的具体实现内容需要用户去编写。如果在进入低功耗模式之前我们降低了处理器频率、关闭了某些外设时钟等的话，那在退出低功耗模式以后就需要恢复处理器频率、重新打开外设时钟等，这个操作在宏 configPOST_SLEEP_PROCESSING()中完成，同样的这个宏的具体内容也需要用户去编写。这两个宏会被函数 vPortSuppressTicksAndSleep()调用，我们可以在 FreeRTOSConfig.h 定义这两个宏，如下：

```
/*************************************************************************/
/*                          FreeRTOS 与低功耗管理相关配置                */
/*************************************************************************/
extern void PreSleepProcessing(uint32_t ulExpectedIdleTime);
extern void PostSleepProcessing(uint32_t ulExpectedIdleTime);

//进入低功耗模式前要做的处理
#define  configPRE_SLEEP_PROCESSING    PreSleepProcessing
//退出低功耗模式后要做的处理
#define  configPOST_SLEEP_PROCESSING   PostSleepProcessing
```

函数 PreSleepProcessing()和 PostSleepProcessing()可以在任意一个 C 文件中编写，本章对应的例程是在 main.c 文件中，函数的具体内容在下一节详解。

4、宏 configEXPECTED_IDLE_TIME_BEFORE_SLEEP

处理器工作在低功耗模式的时间虽说没有任何限制，1 个时钟节拍也行，滴答定时器所能计时的最大值也行。但是时间太短的话意义也不大啊，就 1 个时钟节拍，我这刚进去就得出来！所以我们必须对工作在低功耗模式的时间做个限制，不能太短了，宏 configEXPECTED_IDLE_TIME_BEFORE_SLEEP 就是用来完成这个功能的。此宏默认在文件 FreeRTOS 中有定义，如下：

```
#ifndef configEXPECTED_IDLE_TIME_BEFORE_SLEEP
    #define configEXPECTED_IDLE_TIME_BEFORE_SLEEP 2
#endif

#if configEXPECTED_IDLE_TIME_BEFORE_SLEEP < 2
    #error configEXPECTED_IDLE_TIME_BEFORE_SLEEP must not be less than 2
#endif
```

默认情况下 configEXPECTED_IDLE_TIME_BEFORE_SLEEP 为 2 个时钟节拍，并且最小不能小于 2 个时钟节拍。如果要修改这个值的话可以在文件 FreeRTOSConfi.h 中对其重新定义。此宏会在空闲任务函数 prvIdleTask()中使用！

18.3 低功耗 Tickless 模式实验

18.3.1 实验程序设计

1、实验目的

学习如何使用 FreeRTOS 的低功耗 Tickless 模式，观察 Tickless 模式对于降低系统功耗有无帮助。

2、实验设计

对于功耗要求严格的场合一般不要求有太大的数据处理量，因为功耗与性能很难兼得。一般的低功耗场合都是简单的数据采集设备或者小型的终端控制设备。它们的功能都很简单，周期性的采集数据并且发送给上层，比如服务器，或者接收服务器发送来的指令执行相应的控制操作，比如开灯关灯、开关电机等。

本实验我们就设计一个通过串口发送指定的指令来控制开发板上的 LED1 和 BEEP 开关的实验，本实验就是对“FreeRTOS 实验 14-1 FreeRTOS 二值信号量实验”的简单修改，只是在实验 14-1 的基础上增加了低功耗模式。最后我们可以直接通过对这两个实验的结果对比，观察 Tickless 模式对于降低功耗是否有用。

3、实验工程

FreeRTOS 实验 18-1 FreeRTOS 低功耗 Tickless 模式实验。

4、实验程序与分析

● **低功耗相关函数**

```
//进入低功耗模式前需要处理的事情
//ulExpectedIdleTime：低功耗模式运行时间
void PreSleepProcessing(uint32_t ulExpectedIdleTime)
{
```

```
    //关闭某些低功耗模式下不使用的外设时钟
    __HAL_RCC_GPIOB_CLK_DISABLE();                    (1)
    __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();
    __HAL_RCC_GPIOH_CLK_DISABLE();
}

//退出低功耗模式以后需要处理的事情
//ulExpectedIdleTime：低功耗模式运行时间
void PostSleepProcessing(uint32_t ulExpectedIdleTime)
{
    //退出低功耗模式以后打开那些被关闭的外设时钟
    __HAL_RCC_GPIOB_CLK_ENABLE();                     (2)
    __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();
    __HAL_RCC_GPIOH_CLK_ENABLE();
}
```

(1)、进入低功耗模式以后关闭那些低功耗模式中不用的外设时钟，本实验中串口 1 需要在低功耗模式下使用，因此 USART1 和 GPIOA 的时钟不能被关闭，其他的外设时钟可以关闭。这个要根据实际使用情况来设置，也可以在此函数中执行一些其他有利于降低功耗的处理。

(2)、退出低功耗模式以后需要打开函数 PreSleepProcessing()中关闭的那些外设的时钟。

● **任务设置**

```
#define START_TASK_PRIO                 1       //任务优先级
#define START_STK_SIZE                  256     //任务堆栈大小
TaskHandle_t StartTask_Handler;                 //任务句柄
void start_task(void *pvParameters);            //任务函数

#define TASK1_TASK_PRIO                 2       //任务优先级
#define TASK1_STK_SIZE                  256     //任务堆栈大小
TaskHandle_t Task1Task_Handler;                 //任务句柄
void task1_task(void *pvParameters);            //任务函数

#define DATAPROCESS_TASK_PRIO           3       //任务优先级
#define DATAPROCESS_STK_SIZE            256     //任务堆栈大小
TaskHandle_t DataProcess_Handler;               //任务句柄
void DataProcess_task(void *pvParameters);      //任务函数
```

```
//二值信号量句柄
SemaphoreHandle_t BinarySemaphore;                    //二值信号量句柄

//用于命令解析用的命令值
#define LED1ON                  1
#define LED1OFF                 2
#define BEEPON                  3
#define BEEPOFF                 4
#define COMMANDERR              0XFF
```

● **其他应用函数**

```
//将字符串中的小写字母转换为大写
//str:要转换的字符串
//len：字符串长度
void LowerToCap(u8 *str,u8 len)
{
    u8 i;
    for(i=0;i<len;i++)
    {
        if((96<str[i])&&(str[i]<123))       //小写字母
        str[i]=str[i]-32;                   //转换为大写
    }
}

//命令处理函数，将字符串命令转换成命令值
//str：命令
//返回值: 0XFF，命令错误；其他值，命令值
u8 CommandProcess(u8 *str)
{
    u8 CommandValue=COMMANDERR;
    if(strcmp((char*)str,"LED1ON")==0) CommandValue=LED1ON;
    else if(strcmp((char*)str,"LED1OFF")==0) CommandValue=LED1OFF;
    else if(strcmp((char*)str,"BEEPON")==0) CommandValue=BEEPON;
    else if(strcmp((char*)str,"BEEPOFF")==0) CommandValue=BEEPOFF;
    return CommandValue;
}
```

● **main()函数**

```
int main(void)
{
    HAL_Init();                             //初始化 HAL 库
    Stm32_Clock_Init(360,25,2,8);           //设置时钟,180Mhz
    delay_init(180);                        //初始化延时函数
    uart_init(115200);                      //初始化串口
```

```
    LED_Init();                             //初始化 LED
    KEY_Init();                             //初始化按键
    PCF8574_Init();                         //初始化 PCF8574
    SDRAM_Init();                           //初始化 SDRAM
    my_mem_init(SRAMIN);                    //初始化内部内存池

    //创建开始任务
    xTaskCreate((TaskFunction_t     )start_task,            //任务函数
                (const char*        )"start_task",          //任务名称
                (uint16_t           )START_STK_SIZE,        //任务堆栈大小
                (void*              )NULL,                  //传递给任务函数的参数
                (UBaseType_t        )START_TASK_PRIO,       //任务优先级
                (TaskHandle_t*      )&StartTask_Handler);   //任务句柄
    vTaskStartScheduler();                                  //开启任务调度
}
```

● **任务函数**

```
//开始任务任务函数
void start_task(void *pvParameters)
{
    taskENTER_CRITICAL();               //进入临界区
    //创建二值信号量
    BinarySemaphore=xSemaphoreCreateBinary();
    //创建 TASK1 任务
    xTaskCreate((TaskFunction_t     )task1_task,
                (const char*        )"task1_task",
                (uint16_t           )TASK1_STK_SIZE,
                (void*              )NULL,
                (UBaseType_t        )TASK1_TASK_PRIO,
                (TaskHandle_t*      )&Task1Task_Handler);
    //创建 TASK2 任务
    xTaskCreate((TaskFunction_t     )DataProcess_task,
                (const char*        )"keyprocess_task",
                (uint16_t           )DATAPROCESS_STK_SIZE,
                (void*              )NULL,
                (UBaseType_t        )DATAPROCESS_TASK_PRIO,
                (TaskHandle_t*      )&DataProcess_Handler);
    vTaskDelete(StartTask_Handler);     //删除开始任务
    taskEXIT_CRITICAL();                //退出临界区
}

//task1 任务函数
void task1_task(void *pvParameters)
{
```

```
    while(1)
    {
        LED0=!LED0;
        vTaskDelay(500);                //延时 500ms，也就是 500 个时钟节拍
    }
}

//DataProcess_task 函数
void DataProcess_task(void *pvParameters)
{
    u8 len=0;
    u8 CommandValue=COMMANDERR;
    BaseType_t err=pdFALSE;

    u8 *CommandStr;
    while(1)
    {
        err=xSemaphoreTake(BinarySemaphore,portMAX_DELAY);    //获取信号量
        if(err==pdTRUE)                                       //获取信号量成功
        {
            len=USART_RX_STA&0x3fff;                          //得到此次接收到的数据长度
            CommandStr=mymalloc(SRAMIN,len+1);  //申请内存
            sprintf((char*)CommandStr,"%s",USART_RX_BUF);
            CommandStr[len]='\0';                             //加上字符串结尾符号
            LowerToCap(CommandStr,len);                       //将字符串转换为大写
            CommandValue=CommandProcess(CommandStr);     //命令解析
            if(CommandValue!=COMMANDERR)
            {
                printf("命令为:%s\r\n",CommandStr);
                switch(CommandValue)                          //处理命令
                {
                    case LED1ON:
                        LED1=0;
                        break;
                    case LED1OFF:
                        LED1=1;
                        break;
                    case BEEPON:
                        PCF8574_WriteBit(BEEP_IO,0);
                        break;
                    case BEEPOFF:
                        PCF8574_WriteBit(BEEP_IO,1);
                        break;
```

```
                }
            }
            else
            {
                printf("无效的命令，请重新输入!!\r\n");
            }
            USART_RX_STA=0;
            memset(USART_RX_BUF,0,USART_REC_LEN);     //串口接收缓冲区清零
            myfree(SRAMIN,CommandStr);                //释放内存
        }
    }
}
```

●中断初始化及处理过程

同实验“FreeRTOS 实验 14-1 FreeRTOS 二值信号量实验”一样。

总体来说，本实验代码和“FreeRTOS 实验 14-1 FreeRTOS 二值信号量实验”基本一样，只是有微小的改动。

18.3.2 实验程序运行结果

编译并下载实验代码到开发板中，打开串口调试助手，通过串口调试助手发送命令就可以控制 LED1 或者 BEEP 的开关，操作方法和“FreeRTOS 实验 14-1 FreeRTOS 二值信号量实验”一样，这不是本章的重点，本章的重点是观察 Tickless 模式是否能够降低系统功耗。

既然要观察功耗肯定需要一个能测量功耗的仪器，仪器的灵敏度尽可能高一点，这里我用的是网上买来的一个测量手机充电电流和功耗的小仪器，通过 USB 线连接到开发板的 USB_232 接口为开发板提供电能，仪器的显示界面内容如图 19.3.2.1 所示：

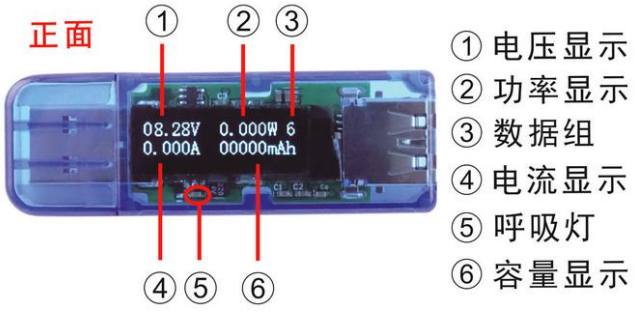


图 18.3.2.1 仪器界面内容

测量注意事项：

● 关闭板子其他所有的供电接口，确保板子的供电电路只有一条有效。本实验中笔者通过开发板的 USB_232 接口为开发板供电。

● 拔掉板子上的液晶屏、JLINK、STLink 等外接模块，防止对测量结果造成干扰。

● 由于开发板硬件并没有做低功耗处理，在通过命令控制 LED1 或者 BEEP 开关以后功耗可能会增大，即使进入低功耗模式功耗也没法降低到原来的水平。这是因为外设相关电路开始工作了，但是进入低功耗模式以后没法去关闭这些电路导致的，属于正常现象。

1、开启 Tickless 模式

打开 Tickless 模式，也就是将宏 configUSE_TICKLESS_IDLE 设置为 1，经过测量系统的整

体功耗如图 18.3.2.2 所示：

图 18.3.2.2 系统功耗

打开 Tickless 模式以后板子的工作电压为：5.13V，工作电流为：0.116A=116mA，功率为：0.595W=595mW。

2、关闭 Tickless 模式

关闭 Tickless 模式，将宏 configUSE_TICKLESS_IDLE 设置为 0，经过测量系统的整体功耗如图 18.3.2.3 所示：

图 18.2.2.3 系统功耗

关闭 Tickless 模式以后板子的工作电压为：5.13V，工作电流为：0.163A=163mA，功率为 0.836W=836mW。

3、总结

通过对比可以看出当开启了 FreeRTOS 的 Tickless 模式以后系统的工作电流降低了 163-116=47mA，功率降低了：836-595=241mW。这个还只是 STM32F429 芯片自身的睡眠模式带来的功耗收益，如果再配合硬件上的设计、选择其他低功耗模式那么功耗肯定会压到一个很低的水平。

第十九章 FreeRTOS 空闲任务

空闲任务是 FreeRTOS 必不可少的一个任务，其他 RTOS 类系统也有空闲任务，比如 uC/OS。看名字就知道，空闲任务是处理器空闲的时候去运行的一个任务，当系统中没有其他就绪任务的时候空闲任务就会开始运行，空闲任务最重要的作用就是让处理器在无事可做的时候找点事做，防止处理器无聊，因此，空闲任务的优先级肯定是最低的。当然了，实际上肯定不会这么浪费宝贵的处理器资源，FreeRTOS 空闲任务中也会执行一些其他的处理。本章我们就来学习一下 FreeRTOS 中的空闲任务，本章分为如下几部分：

19.1 空闲任务详解
19.2 空闲任务钩子函数详解
19.3 空闲任务钩子函数实验

19.1 空闲任务详解

19.1.1 空闲任务简介

当 FreeRTOS 的调度器启动以后就会自动的创建一个空闲任务，这样就可以确保至少有一任务可以运行。但是这个空闲任务使用最低优先级，如果应用中有其他高优先级任务处于就绪态的话这个空闲任务就不会跟高优先级的任务抢占 CPU 资源。空闲任务还有另外一个重要的职责，如果某个任务要调用函数 vTaskDelete()删除自身，那么这个任务的任务控制块 TCB 和任务堆栈等这些由 FreeRTOS 系统自动分配的内存需要在空闲任务中释放掉，如果删除的是别的任务那么相应的内存就会被直接释放掉，不需要在空闲任务中释放。因此，一定要给空闲任务执行的机会！除此以外空闲任务就没有什么特别重要的功能了，所以可以根据实际情况减少空闲任务使用 CPU 的时间(比如，当 CPU 运行空闲任务的时候使处理器进入低功耗模式)。

用户可以创建与空闲任务优先级相同的应用任务，当宏 configIDLE_SHOULD_YIELD 为 1 的话应用任务就可以使用空闲任务的时间片，也就是说空闲任务会让出时间片给同优先级的应用任务。这种方法在 3.2 节讲解 configIDLE_SHOULD_YIELD 的时候就讲过了，这种机制要求 FreeRTOS 使用抢占式内核。

19.1.2 空闲任务的创建

当调用函数 vTaskStartScheduler()启动任务调度器的时候此函数就会自动创建空闲任务，代码如下：

```
void vTaskStartScheduler( void )
{
    BaseType_t xReturn;

    //创建空闲任务，使用最低优先级
    #if( configSUPPORT_STATIC_ALLOCATION == 1 )                    (1)
    {
        StaticTask_t *pxIdleTaskTCBBuffer = NULL;
        StackType_t *pxIdleTaskStackBuffer = NULL;
        uint32_t ulIdleTaskStackSize;

        vApplicationGetIdleTaskMemory( &pxIdleTaskTCBBuffer, &pxIdleTaskStackBuffer, \
                                       &ulIdleTaskStackSize );
        xIdleTaskHandle = xTaskCreateStatic(  prvIdleTask,
                                              "IDLE",
                                              ulIdleTaskStackSize,
                                              ( void * ) NULL,
                                              ( tskIDLE_PRIORITY | portPRIVILEGE_BIT ),
                                              pxIdleTaskStackBuffer,
                                              pxIdleTaskTCBBuffer );

        if( xIdleTaskHandle != NULL )
        {
            xReturn = pdPASS;
```

```
        }
        else
        {
            xReturn = pdFAIL;
        }
    }
    #else                                                                    (2)
    {
        xReturn = xTaskCreate( prvIdleTask,
                               "IDLE",
                               configMINIMAL_STACK_SIZE,
                               ( void * ) NULL,
                               ( tskIDLE_PRIORITY | portPRIVILEGE_BIT ),
                               &xIdleTaskHandle );
    }
    #endif /* configSUPPORT_STATIC_ALLOCATION */

    /*********************************************************************/
    /**************************省略其他代码*******************************/
    /*********************************************************************/
}
```

(1)、使用静态方法创建空闲任务。

(2)、使用动态方法创建空闲任务，空闲任务的任务函数为 prvIdleTask()，任务堆栈大小为 configMINIMAL_STACK_SIZE，任务堆栈大小可以在 FreeRTOSConfig.h 中修改。任务优先级为 tskIDLE_PRIORITY，宏 tskIDLE_PRIORITY 为 0，说明空闲任务优先级最低，用户不能随意修改空闲任务的优先级！

19.1.3 空闲任务函数

空闲任务的任务函数为 prvIdleTask()，但是实际上是找不到这个函数的，因为它是通过宏定义来实现的，在文件 portmacro.h 中有如下宏定义：

```
#define portTASK_FUNCTION( vFunction, pvParameters ) void vFunction( void *pvParameters )
```

其中 portTASK_FUNCTION()在文件 tasks.c 中有定义，它就是空闲任务的任务函数，源码如下：

```
static portTASK_FUNCTION( prvIdleTask, pvParameters )                    (1)
{
    ( void ) pvParameters;    //防止报错

    //本函数为 FreeRTOS 的空闲任务任务函数，当任务调度器启动以后空闲任务会自动
    //创建

    for( ;; )
    {
```

```
//检查是否有任务要删除自己，如果有的话就释放这些任务的任务控制块 TCB 和
//任务堆栈的内存

prvCheckTasksWaitingTermination();                                    (2)

#if ( configUSE_PREEMPTION == 0 )
{

    //如果没有使用抢占式内核的话就强制进行一次任务切换查看是否有其他
    //任务有效，如果有使用抢占式内核的话就不需要这一步，因为只要有任
    //何任务有效(就绪)之后都会自动的抢夺 CPU 使用权

    taskYIELD();
}
#endif /* configUSE_PREEMPTION */

#if ( ( configUSE_PREEMPTION == 1 ) && ( configIDLE_SHOULD_YIELD == 1 ) ) (3)
{
    //如果使用抢占式内核并且使能时间片调度的话，当有任务和空闲任务共享
    //一个优先级的时候，并且此任务处于就绪态的话空闲任务就应该放弃本时
    //间片，将本时间片剩余的时间让给这个就绪任务。如果在空闲任务优先级
    //下的就绪列表中有多个用户任务的话就执行这些任务。

    if( listCURRENT_LIST_LENGTH( \                                    (4)
                &( pxReadyTasksLists[ tskIDLE_PRIORITY ] ) )> ( UBaseType_t ) 1 )
    {
        taskYIELD();
    }
    else
    {
        mtCOVERAGE_TEST_MARKER();
    }
}
#endif

#if ( configUSE_IDLE_HOOK == 1)
{
    extern void vApplicationIdleHook( void );

    //执行用户定义的空闲任务钩子函数，注意！钩子函数里面不能使用任何
    //可以引起阻塞空闲任务的 API 函数。

    vApplicationIdleHook();                                           (5)
```

```
        }
        #endif /* configUSE_IDLE_HOOK */

        //如果使能了 Tickless 模式的话就执行相关的处理代码
        #if ( configUSE_TICKLESS_IDLE != 0 )                                    (6)
        {
            TickType_t xExpectedIdleTime;

            xExpectedIdleTime = prvGetExpectedIdleTime();                       (7)
            if( xExpectedIdleTime >= configEXPECTED_IDLE_TIME_BEFORE_SLEEP ) (8)
            {
                vTaskSuspendAll();                                              (9)
                {

                    //调度器已经被挂起，重新采集一次时间值，这次的时间值可以
                    //使用

                    configASSERT( xNextTaskUnblockTime >= xTickCount );
                    xExpectedIdleTime = prvGetExpectedIdleTime();               (10)

                    if( xExpectedIdleTime >=\
                                configEXPECTED_IDLE_TIME_BEFORE_SLEEP )
                    {
                        traceLOW_POWER_IDLE_BEGIN();
                        portSUPPRESS_TICKS_AND_SLEEP( xExpectedIdleTime ); (11)
                        traceLOW_POWER_IDLE_END();
                    }
                    else
                    {
                        mtCOVERAGE_TEST_MARKER();
                    }
                }
                ( void ) xTaskResumeAll();                                      (12)
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
            }
        }
        #endif /* configUSE_TICKLESS_IDLE */
    }
}
```

(1)、将此行展开就是 static void prvIdleTask(void *pvParameters)，创建空闲任务的时候任务

函数名就是 prvIdleTask()。

(2)、调用函数 prvCheckTasksWaitingTermination()检查是否有需要释放内存的被删除任务，当有任务调用函数 vTaskDelete()删除自身的话，此任务就会添加到列表 xTasksWaitingTermination 中。函数 prvCheckTasksWaitingTermination()会检查列表 xTasksWaitingTermination 是否为空，如果不为空的话就依次将列表中所有任务对应的内存释放掉(任务控制块 TCB 和任务堆栈的内存)。

(3)、使用抢占式内核并且 configIDLE_SHOULD_YIELD 为 1，说明空闲任务需要让出时间片给同优先级的其他就绪任务。

(4)、检查优先级为 tskIDLE_PRIORITY(空闲任务优先级)的就绪任务列表是否为空，如果不为空的话就调用函数 taskYIELD()进行一次任务切换。

(5)、如果使能了空闲任务钩子函数的话就执行这个钩子函数，空闲任务钩子函数的函数名为 vApplicationIdleHook()，这个函数需要用户自行编写！在编写这个这个钩子函数的时候一定不能调用任何可以阻塞空闲任务的 API 函数。

(6)、configUSE_TICKLESS_IDLE 不为 0，说明使能了 FreeRTOS 的低功耗 Tickless 模式。

(7)、调用函数 prvGetExpectedIdleTime()获取处理器进入低功耗模式的时长，此值保存在变量 xExpectedIdleTime 中，单位为时钟节拍数。

(8)、xExpectedIdleTime 值要大于 configEXPECTED_IDLE_TIME_BEFORE_SLEEP 才有效，原因在上一章讲解宏 configEXPECTED_IDLE_TIME_BEFORE_SLEEP 的时候已经说过了。

(9)、处理 Tickless 模式，挂起任务调度器，其实就是起到临界段代码保护功能

(10)、重新获取一次时间值，这次的时间值是直接用于 portSUPPRESS_TICKS_AND_SLEEP()的。

(11)、调用 portSUPPRESS_TICKS_AND_SLEEP()进入低功耗 Tickless 模式，上一章已经详细的讲解过 Tickless 模式了。

(12)、恢复任务调度器。

19.2 空闲任务钩子函数详解

19.2.1 钩子函数

FreeRTOS 中有多个钩子函数，钩子函数类似回调函数，当某个功能(函数)执行的时候就会调用钩子函数，至于钩子函数的具体内容那就由用户来编写。如果不需要使用钩子函数的话就什么也不用管，钩子函数是一个可选功能，可以通过宏定义来选择使用哪个钩子函数，可选的钩子函数如表 19.2.1.1 所示：

宏定义	描述
configUSE_IDLE_HOOK	空闲任务钩子函数，空闲任务会调用此钩子函数。
configUSE_TICK_HOOK	时间片钩子函数，xTaskIncrementTick()会调用此钩子函数。此钩子函数最终会被节拍中断服务函数用，对于 STM32 来说就是滴答定时器中断服务函数。
configUSE_MALLOC_FAILED_HOOK	内存申请失败钩子函数，当使用函数 pvPortMalloc()申请内存失败的时候就会调用此钩子函数。
configUSE_DAEMON_TASK_STARTUP_HOOK	守护(Daemon)任务启动钩子函数，守护任务也就是定时器服务任务。

表 19.2.1.1 钩子函数使能宏

钩子函数的使用方法基本相同，用户使能相应的钩子函数，然后自行根据实际需求编写钩子函数的内容，下一节我们会以空闲任务钩子函数为例讲解如何使用钩子函数。

19.2.2 空闲任务钩子函数

在每个空闲任务运行周期都会调用空闲任务钩子函数，如果想在空闲任务优先级下处理某个任务有两种选择：

● 在空闲任务钩子函数中处理任务。

不管什么时候都要保证系统中至少有一个任务可以运行，因此绝对不能在空闲任务钩子函数中调用任何可以阻塞空闲任务的 API 函数，比如 vTaskDelay()，或者其他带有阻塞时间的信号量或队列操作函数。

● 创建一个与空闲任务优先级相同的任务。

创建一个任务是最好的解决方法，但是这种方法会消耗更多的 RAM。

要使用空闲任务钩子函数首先要在 FreeRTOSConfig.h 中将宏 configUSE_IDLE_HOOK 改为 1，然后编写空闲任务钩子函数 vApplicationIdleHook()。通常在空闲任务钩子函数中将处理器设置为低功耗模式来节省电能，为了与 FreeRTOS 自带的 Tickless 模式做区分，这里我暂且将这种低功耗的实现方法称之为通用低功耗模式(因为几乎所有的 RTOS 系统都可以使用这种方法实现低功耗)。这种通用低功耗模式和 FreeRTOS 自带的 Tickless 模式的区别我们通过下图 19.2.2.1 来对比分析一下。

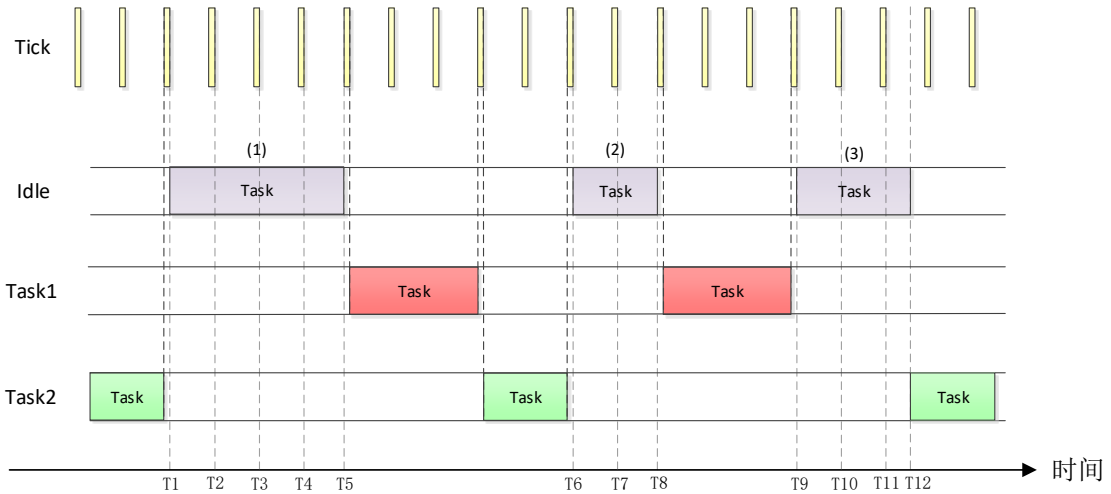


图 19.2.2.1 两种低功耗模式对比

图 19.2.2.1 中有三个任务，它们分别为一个空闲任务(Idle)，两个用户任务(Task1 和 Task2)，其中空闲任务一共有运行了三次，分别为(1)、(2)、(3)，其中 T1 到 T12 是 12 个时刻，下面我们分别从这两种低功耗的实现方法去分析一下整个过程。

1、通用低功耗模式

如果使用通用低功耗模式的话每个滴答定时器中断都会将处理器从低功耗模式中唤醒，以(1)为例，再 T2 时刻处理器从低功耗模式中唤醒，但是接下来由于没有就绪的其他任务所以处理器又再一次进入低功耗模式。T2、T3 和 T4 这三个时刻都一样，反复的进入低功耗、退出低功耗，最理想的情况应该是从 T1 时刻就进入低功耗，然后在 T5 时刻退出。

在(2)中空闲任务只工作了两个时钟节拍，但是也执行了低功耗模式的进入和退出，显然这个意义不大，因为进出低功耗也是需要时间的。

(3)中空闲任务在 T12 时刻被某个外部中断唤醒，中断的具体处理过程在任务 2(使用信号量实现中断与任务之间的同步)。

2、低功耗 Tickless 模式

在(1)中的 T1 时刻处理器进入低功耗模式，在 T5 时刻退出低功耗模式。相比通用低功耗模式少了 3 次进出低功耗模式的操作。

在(2)中由于空闲任务只运行了两个时钟节拍，所以就没必要进入低功耗模式。说明在 Tickless 模式中只有空闲任务要运行时间的超过某个最小阈值的时候才会进入低功耗模式，此阈值通过 configEXPECTED_IDLE_TIME_BEFORE_SLEEP 来设置，上一章已经讲过了。

(3)中的情况和通用低功耗模式一样。

可以看出相对与通用低功耗模式，FreeRTOS 自带的 Tickless 模式更加合理有效，所以如果有低功耗设计需求的话大家尽量使用 FreeRTOS 再带的 Tickless 模式。当然了，如果对于功耗要求不严格的话通用低功耗模式也可以使用，下一节将通过一个实验讲解如何在空闲任务钩子函数中实现低功耗。

19.3 空闲任务钩子函数实验

19.3.1 实验程序设计

1、实验目的

学习如何在 FreeRTOS 空闲任务钩子函数中实现低功耗。

2、实验设计

本实验在上一章实验“FreeRTOS 实验 18-1 FreeRTOS 低功耗 Tickless 模式实验”上做简单修改，关闭 Tickless 模式，在空闲任务钩子函数中使用 WFI 指令是处理器进入睡眠模式。

3、实验工程

FreeRTOS 实验 19-1 FreeRTOS 空闲任务钩子函数实验。

4、实验程序与分析

● **相关宏设置**

```
#define configUSE_TICKLESS_IDLE     0     //关闭低功耗 tickless 模式
#define configUSE_IDLE_HOOK         1     //使能空闲任务钩子函数
```

● 空闲任务钩子函数

```
//进入低功耗模式前需要处理的事情
void BeforeEnterSleep(void)
{
    //关闭某些低功耗模式下不使用的外设时钟
    __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();
    __HAL_RCC_GPIOH_CLK_DISABLE();
}

//退出低功耗模式以后需要处理的事情
void AfterExitSleep(void)
{
    //退出低功耗模式以后打开那些被关闭的外设时钟
    __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();
    __HAL_RCC_GPIOH_CLK_ENABLE();
}

//空闲任务钩子函数
void vApplicationIdleHook(void)
{
    __disable_irq();
    __dsb(portSY_FULL_READ_WRITE );
    __isb(portSY_FULL_READ_WRITE );

    BeforeEnterSleep();                 //进入睡眠模式之前需要处理的事情
    __wfi();                            //进入睡眠模式
    AfterExitSleep();                   //退出睡眠模式之后需要处理的事情

    __dsb(portSY_FULL_READ_WRITE );
    __isb(portSY_FULL_READ_WRITE );
    __enable_irq();
}
```

空闲任务钩子函数主要目的就是调用 WFI 指令使 STM32F429 进入睡眠模式，在进入和退

出低功耗模式的时候也可以做一些其他处理，比如关闭外设时钟等等，用法和 FreeRTOS 的 Tickless 模式类似。

● **其他任务函数和设置**

其他有关设置和任务函数的内容同“FreeRTOS 实验 18-1 FreeRTOS 低功耗 Tickless 模式实验”一样，这里就不列出来了。

19.3.2 实验程序运行结果

编译并下载实验代码到开发板中，打开串口调试助手，通过串口调试助手发送命令就可以控制 LED1 或者 BEEP 的开关。当然了，还有本实验的重点，功耗测量！功耗如图 19.3.2.1 所示：

图 19.3.2.1 系统功耗

当前的系统电压为 5.13V，工作电流为 0.119A=119mA，功率为 0.610W=610mW。跟上一章测量出的未使用低功耗模式的功耗相比，工作电流减少了 163-119=44mA，功率降低了 836-610=226mW。

第二十章 FreeRTOS 内存管理

内存管理是一个系统基本组成部分，FreeRTOS 中大量使用到了内存管理，比如创建任务、信号量、队列等会自动从堆中申请内存。用户应用层代码也可以 FreeRTOS 提供的内存管理函数来申请和释放内存，本章就来学习一下 FreeRTOS 自带的内存管理，本章分为如下几部分：

20.1 内存管理简介
20.2 内存碎片
20.3 heap_1 内存分配方法
20.4 heap_2 内存分配方法
20.5 heap_3 内存分配方法
20.6 heap_4 内存分配方法
20.7 heap_5 内存分配方法
20.8 内存管理实验

20.1 FreeRTOS 内存管理简介

FreeRTOS 创建任务、队列、信号量等的时候有两种方法，一种是动态的申请所需的 RAM。一种是由用户自行定义所需的 RAM，这种方法也叫静态方法，使用静态方法的函数一般以“Static”结尾，比如任务创建函数 xTaskCreateStatic()，使用此函数创建任务的时候需要由用户定义任务堆栈，本章我们不讨论这种静态方法。

使用动态内存管理的时候 FreeRTOS 内核在创建任务、队列、信号量的时候会动态的申请 RAM。标准 C 库中的 malloc()和 free()也可以实现动态内存管理，但是如下原因限制了其使用：

- 在小型的嵌入式系统中效率不高。
- 会占用很多的代码空间。
- 它们不是线程安全的。
- 具有不确定性，每次执行的时间不同。
- 会导致内存碎片。
- 使链接器的配置变得复杂。

不同的嵌入式系统对于内存分配和时间要求不同，因此一个内存分配算法可以作为系统的可选选项。FreeRTOS 将内存分配作为移植层的一部分，这样 FreeRTOS 使用者就可以使用自己的合适的内存分配方法。

当内核需要 RAM 的时候可以使用 pvPortMalloc()来替代 malloc()申请内存，不使用内存的时候可以使用 vPortFree()函数来替代 free()函数释放内存。函数 pvPortMalloc()、vPortFree()与函数 malloc()、free()的函数原型类似。

FreeRTOS 提供了 5 种内存分配方法，FreeRTOS 使用者可以其中的某一个方法，或者自己的内存分配方法。这 5 种方法是 5 个文件，分别为:heap_1.c、heap_2.c、heap_3.c、heap_4.c 和 heap_5.c。这 5 个文件再 FreeRTOS 源码中，路径：FreeRTOS->Source->portable->MemMang，后面会详细讲解这 5 种方法有何区别。

20.2 内存碎片

在看 FreeRTOS 的内存分配方法之前我们先来看一下什么叫做内存碎片，看名字就知道是小块的、碎片化的内存。那么内存碎片是怎么来的呢？内存碎片是伴随着内存申请和释放而来的，如图 20.2.1 所示。

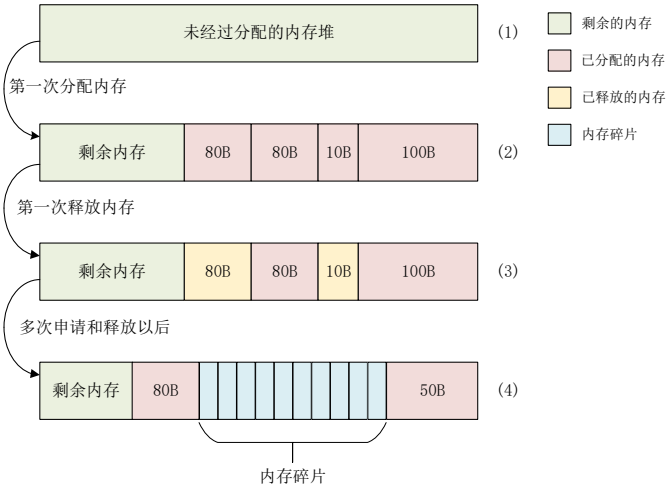


图 20.2.1 内存碎片产生过程

(1)、此时内存堆还没有经过任何操作，为全新的。

(2)、此时经过第一次内存分配，一共分出去了 4 块内存块，大小分别为 80B、80B、10B 和 100B。

(3)、有些应用使用完内存，进行了释放，从左往右第一个 80B 和后面的 10B 这两个内存块就是释放的内存。如果此时有个应用需要 50B 的内存，那么它可以从两个地方来获取到，一个是最前面的还没被分配过的剩余内存块，另一个就是刚刚释放出来的 80B 的内存块。但是很明显，刚刚释放出来的这个 10B 的内存块就没法用了，除非此时有另外一个应用所需要的内存小于 10B。

(4)、经过很多次的申请和释放以后，内存块被不断的分割、最终导致大量很小的内存块！也就是图中 80B 和 50B 这两个内存块之间的小内存块，这些内存块由于太小导致大多数应用无法使用，这些没法使用的内存块就沦为了内存碎片！

内存碎片是内存管理算法重点解决的一个问题，否则的话会导致实际可用的内存越来越少，最终应用程序因为分配不到合适的内存而奔溃！FreeRTOS 的 heap_4.c 就给我们提供了一个解决内存碎片的方法，那就是将内存碎片进行合并组成一个新的可用的大内存块。

20.3 heap_1 内存分配方法

20.3.1 分配方法简介

动态内存分配需要一个内存堆，FreeRTOS 中的内存堆为 ucHeap[]，大小为 configTOTAL_HEAP_SIZE，这个前面讲 FreeRTOS 配置的时候就讲过了。不管是哪种内存分配方法，它们的内存堆都为 ucHeap[],而且大小都是 configTOTAL_HEAP_SIZE。内存堆在文件 heap_x.c(x 为 1~5)中定义的，比如 heap_1.c 文件就有如下定义：

```
#if( configAPPLICATION_ALLOCATED_HEAP == 1 )
    extern uint8_t ucHeap[ configTOTAL_HEAP_SIZE ];      //需要用户自行定义内存堆
#else
    static uint8_t ucHeap[ configTOTAL_HEAP_SIZE ];      //编译器决定
#endif
```

当宏 configAPPLICATION_ALLOCATED_HEAP 为 1 的时候需要用户自行定义内存堆，否则的话由编译器来决定，默认都是由编译器来决定的。如果自己定义的话就可以将内存堆定义到外部 SRAM 或者 SDRAM 中。

heap_1 实现起来就是当需要 RAM 的时候就从一个大数组(内存堆)中分一小块出来，大数组(内存堆)的容量为 configTOTAL_HEAP_SIZE，上面已经说了。使用函数 xPortGetFreeHeapSize()可以获取内存堆中剩余内存大小。

heap_1 特性如下：

1、适用于那些一旦创建好任务、信号量和队列就再也不会删除的应用，实际上大多数的 FreeRTOS 应用都是这样的。

2、具有可确定性(执行所花费的时间大多数都是一样的)，而且不会导致内存碎片。

3、代码实现和内存分配过程都非常简单，内存是从一个静态数组中分配到的，也就是适合于那些不需要动态内存分配的应用。

20.3.2 内存申请函数详解

heap_1 的内存申请函数 pvPortMalloc()源码如下：

```
void *pvPortMalloc( size_t xWantedSize )
{
```

```
void *pvReturn = NULL;
static uint8_t *pucAlignedHeap = NULL;

//确保字节对齐
#if( portBYTE_ALIGNMENT != 1 )                                          (1)
{
    if( xWantedSize & portBYTE_ALIGNMENT_MASK )                         (2)
    {
        //需要进行字节对齐
        xWantedSize += ( portBYTE_ALIGNMENT - ( xWantedSize &\          (3)
                        portBYTE_ALIGNMENT_MASK ) );
    }
}
#endif

vTaskSuspendAll();                                                      (4)
{
    if( pucAlignedHeap == NULL )
    {
        //确保内存堆的开始地址是字节对齐的
        pucAlignedHeap = ( uint8_t * ) ( ( ( portPOINTER_SIZE_TYPE )\   (5)
                                        &ucHeap[ portBYTE_ALIGNMENT ] ) &\
                                        ( ~( ( portPOINTER_SIZE_TYPE )\
                                        portBYTE_ALIGNMENT_MASK ) ) );
    }

    //检查是否有足够的内存供分配，有的话就分配内存
    if( ( ( xNextFreeByte + xWantedSize ) < configADJUSTED_HEAP_SIZE ) && (6)
        ( ( xNextFreeByte + xWantedSize ) > xNextFreeByte ))
    {
        pvReturn = pucAlignedHeap + xNextFreeByte;                      (7)
        xNextFreeByte += xWantedSize;                                   (8)
    }

    traceMALLOC( pvReturn, xWantedSize );
}
( void ) xTaskResumeAll();                                              (9)

#if( configUSE_MALLOC_FAILED_HOOK == 1 )                                (10)
{
    if( pvReturn == NULL )
    {
        extern void vApplicationMallocFailedHook( void );
```

```
                vApplicationMallocFailedHook();
            }
        }
        #endif

        return pvReturn;                                    (11)
}
```

(1)、是否需要进行字节对齐，宏 portBYTE_ALIGNMENT 是需要对齐的字节数，默认为 8，需要进行 8 字节对齐，也就是说参数 xWantedSize 要为 8 的倍数，如果不是的话就需要调整为 8 的倍数。

(2)、参数 xWantedSize 与宏 portBYTE_ALIGNMENT_MASK 进行与运算来判断 xWantedSize 是否为 8 字节对齐，如果结果等于 0 就说明 xWantedSize 是 8 字节对齐的，否则的话不为 8 字节对齐，portBYTE_ALIGNMENT_MASK 为 0x0007。假如 xWantedSize 为 13，那么 13&0x0007=5，5 大于 0，所以 13 不为 8 的倍数，需要做字节对齐处理。当 xWantedSize 为 16 的时候，16&0x0007=0，所以 16 是 8 的倍数，无需字节对齐。

(3)、当 xWantedSize 不是 8 字节对齐的时候就需要调整为 8 字节对齐，调整方法就是找出大于它并且离它最近的那个 8 字节对齐的数，对于 13 来说就是 16。体现在代码中就是本行的这个公式，同样以 xWantedSize 为 13 为例，计算公式就是：

xWantedSize=13+(8-(13&0x0007))=13+(8-5)=16;

(4)、调用函数 vTaskSuspendAll()挂起任务调度器，因为申请内存过程中要做保护，不能被其他任务打断。

(5)、确保内存堆的可用起始地址也是 8 字节对齐的，内存堆 ucHeap 的起始地址是由编译器分配的，ucHeap 的起始地址不一定是 8 字节对齐的。但是我们在使用的时候肯定要使用一个 8 字节对齐的起始地址，这个地址用 pucAlignedHeap 表示，同样需要用公式计算一下，公式就是本行代码，ucHeap 和 pucAlignedHeap 如图 20.3.2.1 所示：

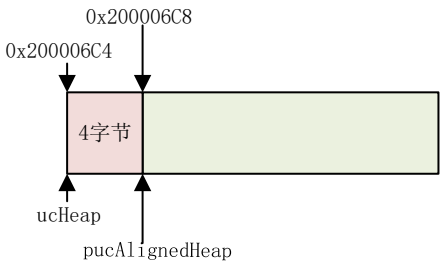


图 20.3.2.1 内存堆地址

图 20.3.2.1 中内存堆 ucHeap 实际起始地址为 0x200006C4，这个地址不是 8 字节对齐的，所以不能拿来使用，经过字节对齐以后可以使用的开始地址是 0x200006C8，所以 pucAlignedHeap 就为 0x200006C8。

(6)、检查一下可用内存是否够分配，分配完成以后是否会产生越界(超出内存堆范围)，xNextFreeByte 是个全局变量，用来保存 pucAlignedHeap 到内存堆剩余内存首地址之间的偏移值，如图 20.3.2.2 所示：

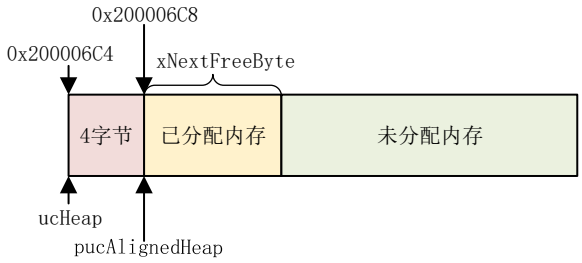


图 20.3.2.2 xNextFreeByte 示意图

(7)、如果内存够分配并且不会产生越界，那么就将申请到的内存首地址赋给 pvReturn，比如我们要申请 30 个字节(字节对齐以后实际需要申请 32 字节)的内存，申请过程如图 20.3.2.3 所示：

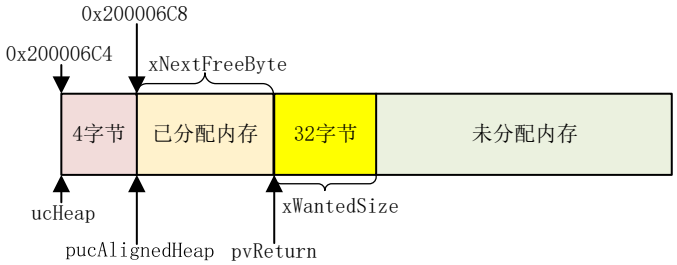


图 20.3.2.3 内存申请过程

(8)、内存申请完成以后更新一下变量 xNextFreeByte。

(9)、调用函数 xTaskResumeAll()恢复任务调度器。

(10)、宏 configUSE_MALLOC_FAILED_HOOK 为 1 的话就说明使能了内存申请失败钩子函数，因此会调用钩子函数 vApplicationMallocFailedHook()，此函数需要用户自行编写实现。

(11)、返回 pvRerurn 值，如果内存申请成功的话就是申请到的内存首地址，内存申请失败的话就返回 NULL。

20.3.3 内存释放函数详解

heap_1 的内存释放函数为 pvFree()，可以看一下 pvFree()的源码，如下：

```
void vPortFree( void *pv )
{
    ( void ) pv;
    configASSERT( pv == NULL );
}
```

可以看出 vPortFree()并没有具体释放内存的过程。因此如果使用 heap_1，一旦申请内存成功就不允许释放！但是 heap_1 的内存分配过程简单，如此看来 heap_1 似乎毫无任何使用价值啊。千万不能这么想，有很多小型的应用在系统一开始就创建好任务、信号量或队列等，在程序运行的整个过程这些任务和内核对象都不会删除，那么这个时候使用 heap_1 就很合适的。

20.4 heap_2 内存分配方法

20.4.1 分配方法简介

heap_2 提供了一个更好的分配算法，不像 heap_1 那样，heap_2 提供了内存释放函数。heap_2 不会把释放的内存块合并成一个大块，这样有一个缺点，随着你不断的申请内存，内存堆就会

被分为很多个大小不一的内存(块)，也就是会导致内存碎片！heap_4 提供了空闲内存块合并的功能。

heap_2 的特性如下：

1、可以使用在那些可能会重复的删除任务、队列、信号量等的应用中，要注意有内存碎片产生！

2、如果分配和释放的内存 n 大小是随机的，那么就要慎重使用了，比如下面的示例：

- 如果一个应用动态的创建和删除任务，而且任务需要分配的堆栈大小都是一样的，那么 heap_2 就非常合适。如果任务所需的堆栈大小每次都是不同，那么 heap_2 就不适合了，因为这样会导致内存碎片产生，最终导致任务分配不到合适的堆栈！不过 heap_4 就很适合这种场景了。
- 如果一个应用中所使用的队列存储区域每次都不同，那么 heap_2 就不适合了，和上面一样，此时可以使用 heap_4。
- 应用需要调用 pvPortMalloc()和 vPortFree()来申请和释放内存，而不是通过其他 FreeRTOS 的其他 API 函数来间接的调用，这种情况下 heap_2 不适合。

3、如果应用中的任务、队列、信号量和互斥信号量具有不可预料性(如所需的内存大小不能确定，每次所需的内存都不相同，或者说大多数情况下所需的内存都是不同的)的话可能会导致内存碎片。虽然这是小概率事件，但是还是要引起我们的注意！

4、具有不可确定性，但是也远比标准 C 中的 mallo()和 free()效率高！

heap_2 基本上可以适用于大多数的需要动态分配内存的工程中，而 heap_4 更是具有将内存碎片合并成一个大的空闲内存块(就是内存碎片回收)的功能。

20.4.2 内存块详解

同 heap_1 一样，heap_2 整个内存堆为 ucHeap[]，大小为 configTOTAL_HEAP_SIZE。可以通过函数 xPortGetFreeHeapSize()来获取剩余的内存大小。

为了实现内存释放，heap_2 引入了内存块的概念，每分出去的一段内存就是一个内存块，剩下的空闲内存也是一个内存块，内存块大小不定。为了管理内存块又引入了一个链表结构，链表结构如下：

```
typedef struct A_BLOCK_LINK
{
    struct A_BLOCK_LINK *pxNextFreeBlock;    //指向链表中下一个空闲内存块
    size_t xBlockSize;                       //当前空闲内存块大小
} BlockLink_t;
```

每个内存块前面都会有一个 BlockLink_t 类型的变量来描述此内存块，比如我们现在申请了一个 16 个字节的内存块，那么此内存块结构就如图 20.4.2.1 所示：

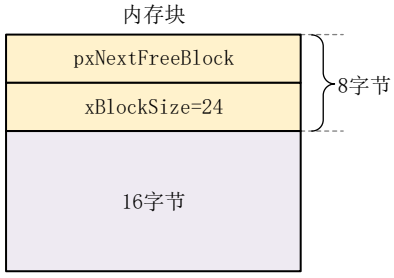


图 20.4.2.1 内存块

图 20.4.2.1 中内存块的总大小是 24 字节，虽然我们只申请了 16 个字节，但是还需要另外

8 字节来保存 BlockLink_t 类型的结构体变量，xBlockSize 记录的是整个内存块的大小。

为了方便管理，可用的内存块会被全部组织在一个链表内，局部静态变量 xStart, xEnd 用来记录这个链表的头和尾，这两个变量定义如下：

```
static BlockLink_t  xStart, xEnd;
```

20.4.3 内存堆初始化函数详解

内存堆初始化函数为 prvHeapInit()，函数源码如下：

```
static void prvHeapInit( void )
{
    BlockLink_t *pxFirstFreeBlock;
    uint8_t *pucAlignedHeap;

    //确保内存堆的开始地址是字节对齐的
    pucAlignedHeap = ( uint8_t * ) ( ( ( portPOINTER_SIZE_TYPE )\                    (1)
                                       &ucHeap[ portBYTE_ALIGNMENT ] ) & \
                                       ( ~( ( portPOINTER_SIZE_TYPE )\
                                       portBYTE_ALIGNMENT_MASK ) ) );

    //xStart 指向空闲内存块链表首。
    xStart.pxNextFreeBlock = ( void * ) pucAlignedHeap;                              (2)
    xStart.xBlockSize = ( size_t ) 0;

    //xEnd 指向空闲内存块链表尾。
    xEnd.xBlockSize = configADJUSTED_HEAP_SIZE;                                      (3)
    xEnd.pxNextFreeBlock = NULL;

    //刚开始只有一个空闲内存块，空闲内存块的总大小就是可用的内存堆大小。
    pxFirstFreeBlock = ( void * ) pucAlignedHeap;                                    (4)
    pxFirstFreeBlock->xBlockSize = configADJUSTED_HEAP_SIZE;
    pxFirstFreeBlock->pxNextFreeBlock = &xEnd;
}
```

(1)、同 heap_1 一样，确保内存堆的可用起始地址为 8 字节对齐。

(2)、初始化 xStart 变量。

(3)、初始化 xEnd 变量。

(4)、每个内存块前面都会保存一个 BlockLink_t 类型的结构体变量，这个结构体变量用来描述此内存块的大小和下一个空闲内存块的地址。

初始化以后的内存堆如图 20.4.3.1 所示：

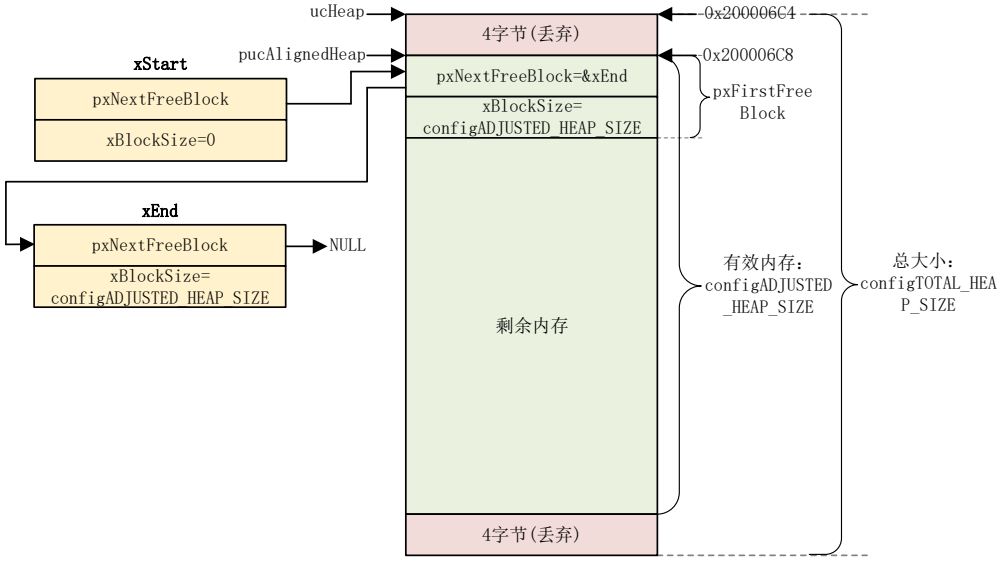


图 20.4.3.1 初始化后的内存堆

20.4.4 内存块插入函数详解

heap_2 允许内存释放，释放的内存肯定是要添加到空闲内存链表中的，宏 prvInsertBlockIntoFreeList()用来完成内存块的插入操作，宏定义如下：

```
#define prvInsertBlockIntoFreeList( pxBlockToInsert )
{
    BlockLink_t *pxIterator;
    size_t xBlockSize;

    xBlockSize = pxBlockToInsert->xBlockSize;

    //遍历链表，查找插入点
    for( pxIterator = &xStart; pxIterator->pxNextFreeBlock->xBlockSize < xBlockSize;        (1)
         pxIterator = pxIterator->pxNextFreeBlock )
    {
        //不做任何事情
    }

    //将内存块插入到插入点
    pxBlockToInsert->pxNextFreeBlock = pxIterator->pxNextFreeBlock;                        (2)
    pxIterator->pxNextFreeBlock = pxBlockToInsert;
}
```

(1)、寻找内存块的插入点，内存块是按照内存大小从小到大连接起来的，因为只是用来寻找插入点的，所以 for 循环体内没有任何代码。

(2)、找到内存插入点以后就将内存块插入到链表中。

假如我们现在需要将大小为 80 字节的内存块插入到链表中，过程如图 20.4.4.1 所示：

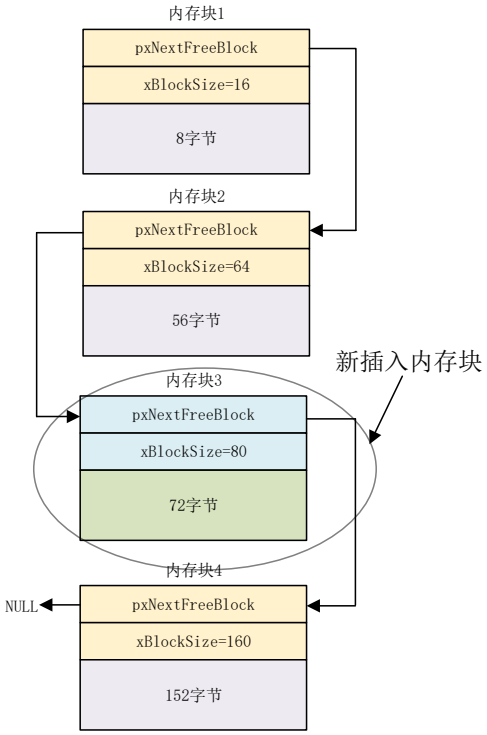


图 20.4.14.1 内存块插入

20.4.5 内存申请函数详解

heap_2 的内存申请函数源码如下：

```
void *pvPortMalloc( size_t xWantedSize )
{
    BlockLink_t *pxBlock, *pxPreviousBlock, *pxNewBlockLink;
    static BaseType_t xHeapHasBeenInitialised = pdFALSE;
    void *pvReturn = NULL;

    vTaskSuspendAll();
    {
        //如果是第一次申请内存的话需要初始化内存堆
        if( xHeapHasBeenInitialised == pdFALSE )                    (1)
        {
            prvHeapInit();
            xHeapHasBeenInitialised = pdTRUE;
        }

        //内存大小字节对齐，实际申请的内存大小还要加上结构体
        //BlockLink_t 的大小
        if( xWantedSize > 0 )                                       (2)
        {
            xWantedSize += heapSTRUCT_SIZE;                         (3)
```

```
            //xWantedSize 做字节对齐处理
            if( ( xWantedSize & portBYTE_ALIGNMENT_MASK ) != 0 )
            {
                xWantedSize += ( portBYTE_ALIGNMENT - ( xWantedSize &
                                portBYTE_ALIGNMENT_MASK ) );
            }
        }

        //所申请的内存大小合理，进行内存分配。
        if( ( xWantedSize > 0 ) && ( xWantedSize < configADJUSTED_HEAP_SIZE ) )
        {
            //从 xStart(最小内存块)开始，查找大小满足所需要内存的内存块。
            pxPreviousBlock = &xStart;
            pxBlock = xStart.pxNextFreeBlock;
            while( ( pxBlock->xBlockSize < xWantedSize ) &&\                    (4)
                   ( pxBlock->pxNextFreeBlock != NULL ) )
            {
                pxPreviousBlock = pxBlock;
                pxBlock = pxBlock->pxNextFreeBlock;
            }

            if( pxBlock != &xEnd )                                              (5)
            {
                //返回申请到的内存首地址
                pvReturn = ( void * ) ( ( ( uint8_t * ) pxPreviousBlock->pxNextFreeBlock ) +\(6)
                                        heapSTRUCT_SIZE );

                pxPreviousBlock->pxNextFreeBlock = pxBlock->pxNextFreeBlock;    (7)

                if( ( pxBlock->xBlockSize - xWantedSize ) >\                    (8)
                    heapMINIMUM_BLOCK_SIZE )
                {
                    pxNewBlockLink = ( void * ) ( ( ( uint8_t * ) pxBlock ) + xWantedSize );
                    pxNewBlockLink->xBlockSize = pxBlock->xBlockSize - xWantedSize;
                    pxBlock->xBlockSize = xWantedSize;
                    prvInsertBlockIntoFreeList( ( pxNewBlockLink ) );           (9)
                }
                xFreeBytesRemaining -= pxBlock->xBlockSize;                     (10)
            }
        }
        traceMALLOC( pvReturn, xWantedSize );
    }
    ( void ) xTaskResumeAll();
```

```
    #if( configUSE_MALLOC_FAILED_HOOK == 1 )                    (11)
    {
        if( pvReturn == NULL )
        {
            extern void vApplicationMallocFailedHook( void );
            vApplicationMallocFailedHook();
        }
    }
    #endif

    return pvReturn;
}
```

(1)、如果是第一次调用函数 pvPortMalloc()申请内存的话就需要先初始化一次内存堆。

(2)、所申请的内存大小进行字节对齐。

(3)、实际申请的内存大小需要再加上结构体 BlockLink_t 的大小，因为每个内存块都会保存一个 BlockLink_t 类型变量，BlockLink_t 结构体的大小为 heapSTRUCT_SIZE。

(4)、从空闲内存链表头 xStart 开始，查找满足所需内存大小的内存块，pxPreviousBlock 所指向的下一个内存块就是找到的可用内存块。

(5)、找到的可用内存块不能是链表尾 xEnd！

(6)、找到内存块以后就将可用内存首地址保存在 pvReturn 中，函数返回的时候返回此值，这个内存首地址要跳过结构体 BlockLink_t，如图 20.4.5.1 所示：

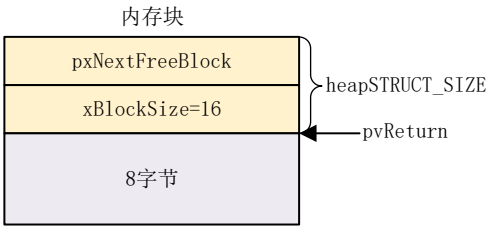


图 20.4.5.1 内存地址

(7)、内存块已经被申请了，所以需要将这个内存块从空闲内存块链表中移除。

(8)、存在这样一种情况(不考虑结构体 BlockLink_t 的大小)，我需要申请 100 个字节的内存，但是经过上面几步我得到了一个 1K 字节的内存块，实际使用中我只需要 100 个字节，剩下的 900 个字节就浪费掉了。这个明显是不合理的，所以需要判断，如果申请到的实际内存减去所需的内存大小(xBlockSize-xWantedSize)大于某个阈值的时候就把多余出来的内存重新组合成一个新的可用空闲内存块。这个阈值由宏 heapMINIMUM_BLOCK_SIZE 来设置，这个阈值要大于 heapSTRUCT_SIZE。

(9)、将新的空闲内存块插入到空闲内存块链表中。

(10)、更新全局变量 xFreeBytesRemaining，此变量用来保存内存堆剩余内存大小。

(11)、如果使能了钩子函数的话就调用钩子函数 vApplicationMallocFailedHook()。

20.4.6 内存释放函数详解

内存释放函数 vPortFree()的源码如下：

```
void vPortFree( void *pv )
{
```

```
    uint8_t *puc = ( uint8_t * ) pv;
    BlockLink_t *pxLink;

    if( pv != NULL )
    {
        puc -= heapSTRUCT_SIZE;                                    (1)
        pxLink = ( void * ) puc;                                   (2)
        vTaskSuspendAll();
        {
            //将内存块添加到空闲内存块链表中
            prvInsertBlockIntoFreeList( ( ( BlockLink_t * ) pxLink ) );   (3)
            xFreeBytesRemaining += pxLink->xBlockSize;             (4)
            traceFREE( pv, pxLink->xBlockSize );
        }
        ( void ) xTaskResumeAll();
    }
}
```

(1)、puc 为要释放的内存首地址，这个地址就是图 20.3.2.4 中 pvReturn 所指向的地址。所以必须减去 heapSTRUCT_SIZE 才是要释放的内存段所在内存块的首地址。

(2)、防止编译器报错。

(3)、将内存块添加到空闲内存块列表中。

(4)、更新变量 xFreeBytesRemaining。

内存释放函数 vPortFree()还是很简单的，主要目的就是将需要释放的内存所在的内存块添加到空闲内存块链表中。

20.5 heap_3 内存分配方法

这个分配方法是对标准 C 中的函数 malloc()和 free()的简单封装，FreeRTOS 对这两个函数做了线程保护，两个函数的源码如下：

```
void *pvPortMalloc( size_t xWantedSize )
{
    void *pvReturn;

    vTaskSuspendAll();                                             (1)
    {
        pvReturn = malloc( xWantedSize );                          (2)
        traceMALLOC( pvReturn, xWantedSize );
    }
    ( void ) xTaskResumeAll();                                     (3)

    #if( configUSE_MALLOC_FAILED_HOOK == 1 )
    {
        if( pvReturn == NULL )
        {
```

```
                extern void vApplicationMallocFailedHook( void );
                vApplicationMallocFailedHook();
            }
        }
        #endif

        return pvReturn;
}

void vPortFree( void *pv )
{
        if( pv )
        {
            vTaskSuspendAll();                                  (4)
            {
                free( pv );                                     (5)
                traceFREE( pv, 0 );
            }
            ( void ) xTaskResumeAll();                          (6)
        }
}
```

(1)和(4)、挂起任务调度器，为 malloc()和 free()提供线程保护

(2)、调用函数 malloc()来申请内存。

(3)和(6)、恢复任务调度器。

(5)、调用函数 free()释放内存。

heap_3 的特性如下：

1、需要编译器提供一个内存堆，编译器库要提供 malloc()和 free()函数。比如使用 STM32 的话可以通过修改启动文件中的 Heap_Size 来修改内存堆的大小，如图 20.5.1 所示。

```
48 Stack_Size      EQU     0x400;
49
50                 AREA    STACK, NOINIT, READWRITE, ALIGN=3
51 Stack_Mem       SPACE   Stack_Size
52 __initial_sp
53
54
55 ; <h> Heap Configuration
56 ;   <o>  Heap Size (in Bytes) <0x0-0xFFFFFFFF:8>
57 ; </h>
58                                          内存堆大小
59 Heap_Size      EQU     0x200;
60
```

图 20.5.1 内存堆大小

2、具有不确定性

3、可能会增加代码量。

注意，在 heap_3 中 configTOTAL_HEAP_SIZE 是没用的！

20.6 heap_4 内存分配方法

20.6.1 分配方法简介

heap_4 提供了一个最优的匹配算法，不像 heap_2，heap_4 会将内存碎片合并成一个大的可用内存块，它提供了内存块合并算法。内存堆为 ucHeap[]，大小同样为 configTOTAL_HEAP_SIZE。可以通过函数 xPortGetFreeHeapSize()来获取剩余的内存大小。

heap_4 特性如下：

1、可以用在那些需要重复创建和删除任务、队列、信号量和互斥信号量等的应用中。

2、不会像 heap_2 那样产生严重的内存碎片，即使分配的内存大小是随机的。

3、具有不确定性，但是远比 C 标准库中的 malloc()和 free()效率高。

heap_4 非常适合于那些需要直接调用函数 pvPortMalloc()和 vPortFree()来申请和释放内存的应用，注意，我们移植 FreeRTOS 的时候就选择的 heap_4！

heap_4 也使用链表结构来管理空闲内存块，链表结构体与 heap_2 一样。heap_4 也定义了两个局部静态变量 xStart 和 pxEnd 来表示链表头和尾，其中 pxEnd 是指向 BlockLink_t 的指针。

20.6.2 内存堆初始化函数详解

内存初始化函数 prvHeapInit()源码如下：

```
static void prvHeapInit( void )
{
    BlockLink_t *pxFirstFreeBlock;
    uint8_t *pucAlignedHeap;
    size_t uxAddress;
    size_t xTotalHeapSize = configTOTAL_HEAP_SIZE;

    //起始地址做字节对齐处理
    uxAddress = ( size_t ) ucHeap;

    if( ( uxAddress & portBYTE_ALIGNMENT_MASK ) != 0 )                    (1)
    {
        uxAddress += ( portBYTE_ALIGNMENT - 1 );
        uxAddress &= ~( ( size_t ) portBYTE_ALIGNMENT_MASK );
        xTotalHeapSize -= uxAddress - ( size_t ) ucHeap;                  (2)
    }

    pucAlignedHeap = ( uint8_t * ) uxAddress;                             (3)

    //xStart 为空闲链表头。
    xStart.pxNextFreeBlock = ( void * ) pucAlignedHeap;                   (4)
    xStart.xBlockSize = ( size_t ) 0;

    //pxEnd 为空闲内存块列表尾，并且将其放到到内存堆的末尾
    uxAddress = ( ( size_t ) pucAlignedHeap ) + xTotalHeapSize;           (5)
```

```
    uxAddress -= xHeapStructSize;
    uxAddress &= ~( ( size_t ) portBYTE_ALIGNMENT_MASK );
    pxEnd = ( void * ) uxAddress;
    pxEnd->xBlockSize = 0;
    pxEnd->pxNextFreeBlock = NULL;

    //开始的时候将内存堆整个可用空间看成一个空闲内存块。
    pxFirstFreeBlock = ( void * ) pucAlignedHeap;                          (6)
    pxFirstFreeBlock->xBlockSize = uxAddress - ( size_t ) pxFirstFreeBlock;
    pxFirstFreeBlock->pxNextFreeBlock = pxEnd;

    //只有一个内存块，而且这个内存块拥有内存堆的整个可用空间
    xMinimumEverFreeBytesRemaining = pxFirstFreeBlock->xBlockSize;         (7)
    xFreeBytesRemaining = pxFirstFreeBlock->xBlockSize;

    xBlockAllocatedBit = ( ( size_t ) 1 ) << ( ( sizeof( size_t ) * heapBITS_PER_BYTE ) - 1 );(8)
}
```

(1)、可用内存堆起始地址做字节对齐处理。

(2)、可用起始地址做字节对齐处理以后难免会有几个字节被抛弃掉，被抛弃的这几个字节不能使用，因此内存堆总的可用大小需要重新计算一下。

(3)、pucAlignedHeap 为内存堆字节对齐以后的可用起始地址。

(4)、初始化 xStart，xStart 为可用内存块链表头。

(5)、初始化 pxEnd，pxEnd 为可用内存块链表尾，pxEnd 放到了内存堆末尾。

(6)、同 heap_2 一样，内存块前面会有一个 BlockLink_t 类型的变量来描述内存块，这里是完成这个变量初始化的。

(7)、xMinimumEverFreeBytesRemaining 记录最小的那个空闲内存块大小，xFreeBytesRemaining 表示内存堆剩余大小。

(8)、初始化静态变量 xBlockAllocatedBit，初始化完成以后此变量值为 0X80000000,此变量是 size_t 类型的，其实就是将 size_t 类型变量的最高位置 1，对于 32 位 MCU 来说就是 0X80000000。此变量会用来标记某个内存块是被使用，BlockLink_t 中的成员变量 xBlockSize 是用来描述内存块大小的，在 heap_4 中其最高位表示此内存块是否被使用，如果为 1 的话就表示被使用了，所以在 heap_4 中一个内存块最大只能为 0x7FFFFFFF。

假设内存堆 ucHeap 的大小为 46KB，即 configTOTAL_HEAP_SIZE =46*1024，ucHeap 的起始地址为 0X200006D4，经过函数 prvHeapInit()初始化以后的内存堆如图 20.6.2.1 所示：

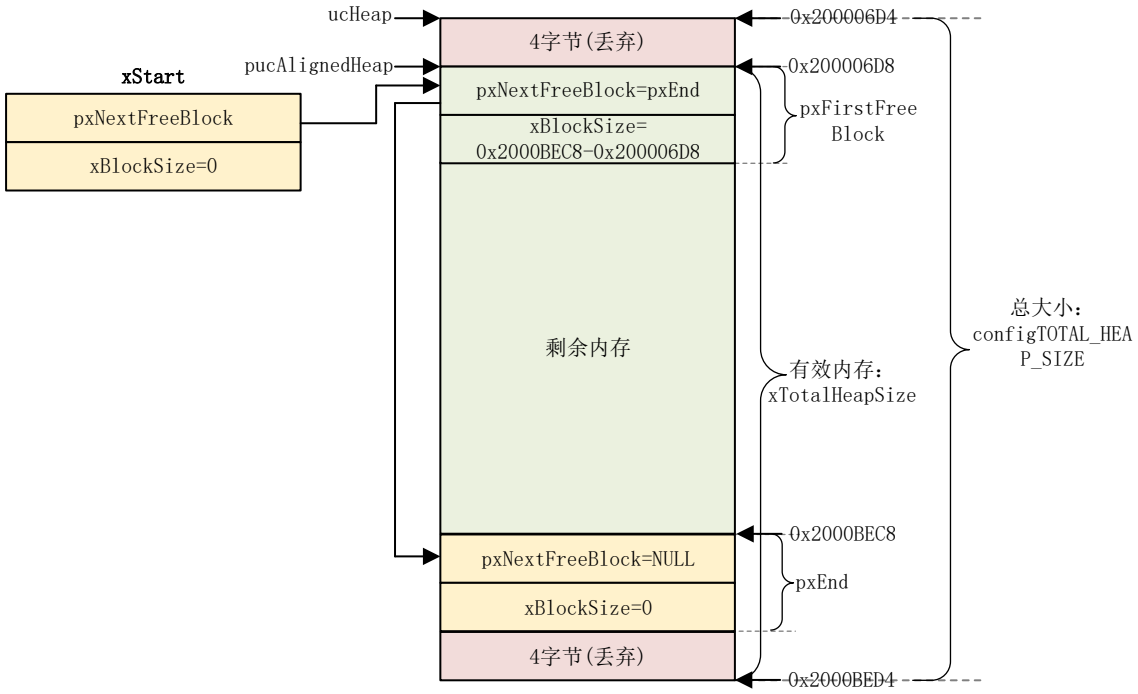


图 20.6.2.1 初始化完成以后的内存堆

20.6.3 内存块插入函数详解

内存块插入函数 prvInsertBlockIntoFreeList()用来将某个内存块插入到空闲内存块链表中，函数源码如下：

```
static void prvInsertBlockIntoFreeList( BlockLink_t *pxBlockToInsert )
{
    BlockLink_t *pxIterator;
    uint8_t *puc;

    //遍历空闲内存块链表，找出内存块插入点，内存块按照地址从低到高连接在一起
    for( pxIterator = &xStart; pxIterator->pxNextFreeBlock < pxBlockToInsert;\          (1)
                                pxIterator = pxIterator->pxNextFreeBlock )
    {
        //不做任何处理
    }

    //插入内存块，如果要插入的内存块可以和前一个内存块合并的话就
    //合并两个内存块
    puc = ( uint8_t * ) pxIterator;
    if( ( puc + pxIterator->xBlockSize ) == ( uint8_t * ) pxBlockToInsert )          (2)
    {
        pxIterator->xBlockSize += pxBlockToInsert->xBlockSize;
        pxBlockToInsert = pxIterator;
    }
```

```
    else
    {
        mtCOVERAGE_TEST_MARKER();
    }

    //检查是否可以和后面的内存块合并，可以的话就合并
    puc = ( uint8_t * ) pxBlockToInsert;.
    if( ( puc + pxBlockToInsert->xBlockSize ) == ( uint8_t * ) pxIterator->pxNextFreeBlock ) (3)
    {
        if( pxIterator->pxNextFreeBlock != pxEnd )
        {
            //将两个内存块组合成一个大的内存块
            pxBlockToInsert->xBlockSize += pxIterator->pxNextFreeBlock->xBlockSize;
            pxBlockToInsert->pxNextFreeBlock =\
            pxIterator->pxNextFreeBlock->pxNextFreeBlock;
        }
        else
        {
            pxBlockToInsert->pxNextFreeBlock = pxEnd;
        }
    }
    else
    {
        pxBlockToInsert->pxNextFreeBlock = pxIterator->pxNextFreeBlock;                (4)
    }

    if( pxIterator != pxBlockToInsert )
    {
        pxIterator->pxNextFreeBlock = pxBlockToInsert;
    }
    else
    {
        mtCOVERAGE_TEST_MARKER();
    }
}
```

(1)、遍历空闲内存块链表，找出当前内存块插入点，内存块是按照地址从低到高的顺序链接在一起的。

(2)、找到插入点以后判断是否可以和要插入的内存块合并，如果可以的话就合并在一起。如图 20.6.3.1 所示：

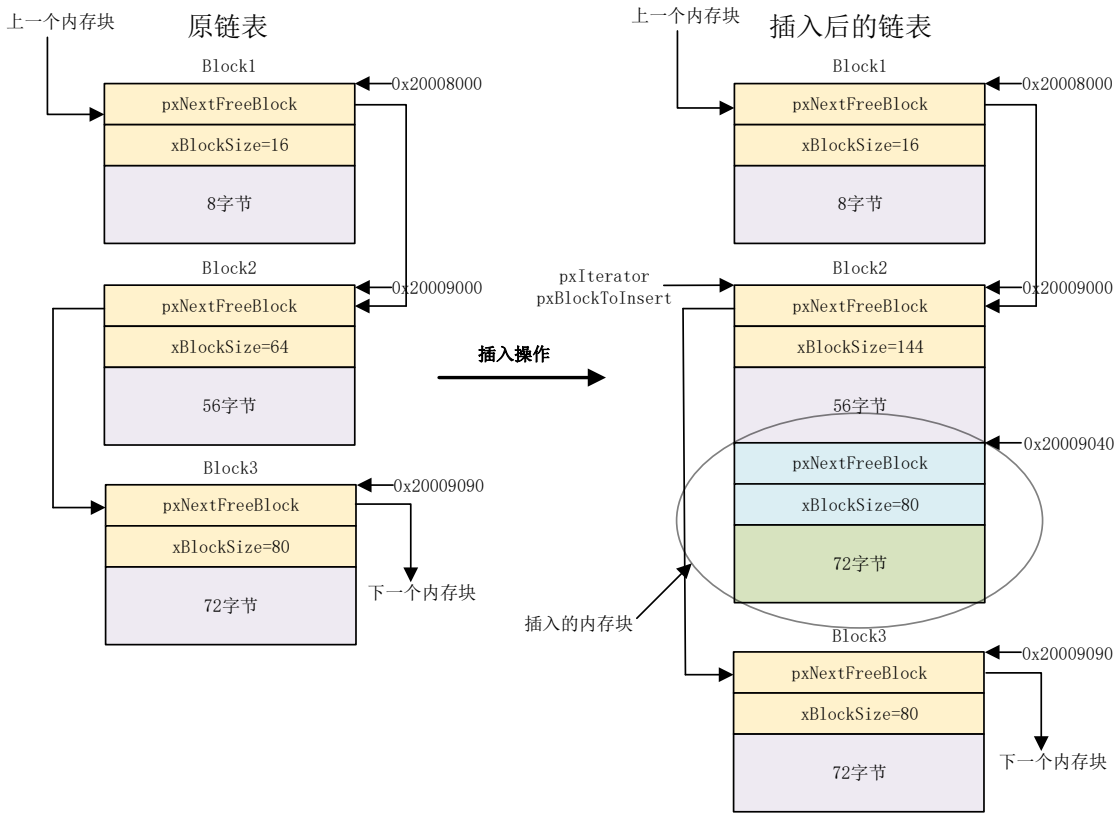


图 20.6.3.1 内存块插入 1

在图 20.6.3.1 中，右边椭圆圈起来的就是要插入的内存块，其起始地址为 0x20009040，该地址刚好和内存块 Block2 的末地址一样，所以这两个内存块可以合并在一起。合并以后 Block2 的大小 xBlockSize 要更新为最新的内存块大小，即 64+80=144。

(3)、再接着检查(2)中合并的新内存块是否可以和下一个内存块合并，也就是 Block3，如果可以的话就再次合并，合并完成以后如图 20.6.3.2 所示：

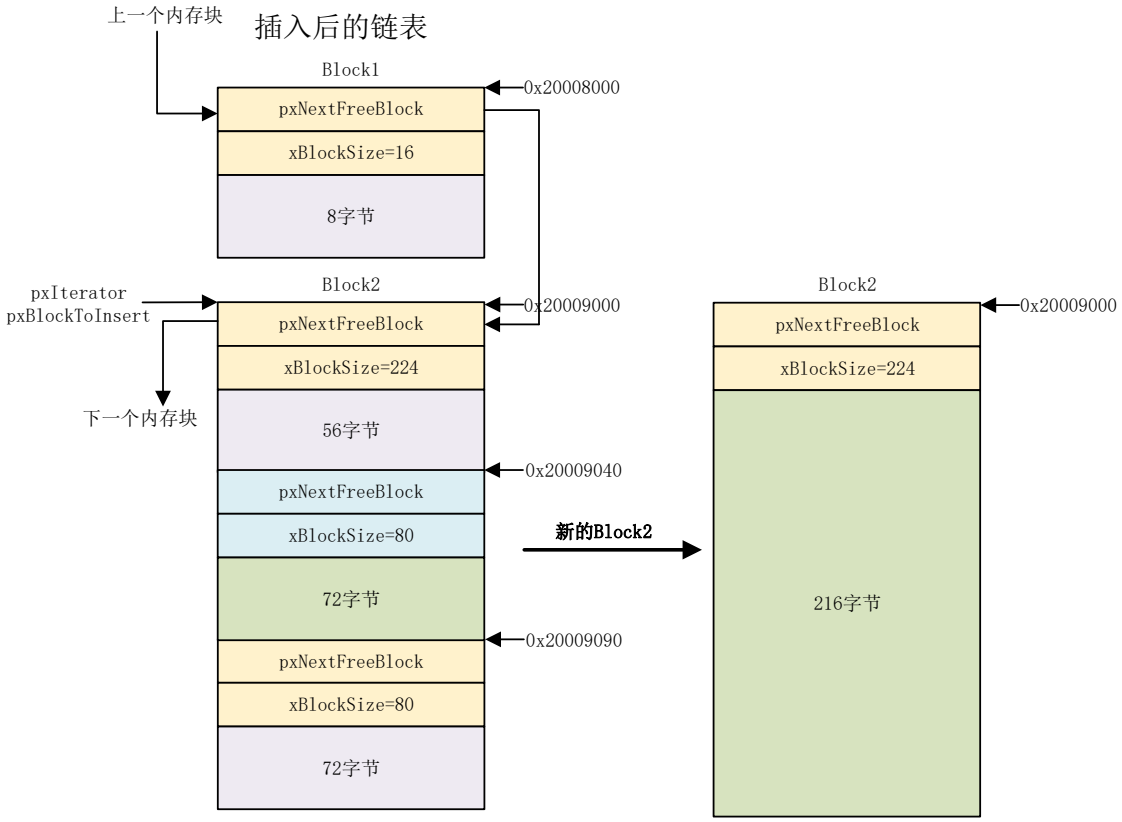


图 20.6.3.2 内存插入 2

在图 20.6.3.2 中可以看出，最终新插入的内存块和 Blokc2、Block3 合并成一个大小为 64+80+80=224 字节的大内存块，这个就是 heap_4 解决内存碎片的方法！

(4)、如果不能和 Block3 合并的话就将这两个内存块链接起来。

(5)、pxIterator 不等于 pxBlockToInsert 就意味着在内存块插入的过程中没有进行过一次内存合并，这样的话就使用最普通的处理方法。pxIterator 所指向的内存块在前，pxBlockToInsert 所指向的内存块在后，将两个内存块链接起来。

20.6.4 内存申请函数详解

heap_4 的内存申请函数源码如下：

```
void *pvPortMalloc( size_t xWantedSize )
{
    BlockLink_t *pxBlock, *pxPreviousBlock, *pxNewBlockLink;
    void *pvReturn = NULL;

    vTaskSuspendAll();
    {
        //第一次调用，初始化内存堆
        if( pxEnd == NULL )                                    (1)
        {
            prvHeapInit();
        }
        else
```

```
{
    mtCOVERAGE_TEST_MARKER();
}

//需要申请的内存块大小的最高位不能为 1，因为最高位用来表示内存块有没有
//被使用
if( ( xWantedSize & xBlockAllocatedBit ) == 0 )                    (2)
{
    if( xWantedSize > 0 )                                          (3)
    {
        xWantedSize += xHeapStructSize;
        if( ( xWantedSize & portBYTE_ALIGNMENT_MASK ) != 0x00 )
        {
            xWantedSize += ( portBYTE_ALIGNMENT - ( xWantedSize &\
                             portBYTE_ALIGNMENT_MASK ) );
            configASSERT( ( xWantedSize & portBYTE_ALIGNMENT_MASK ) == 0 );
        }
        else
        {
            mtCOVERAGE_TEST_MARKER();
        }
    }
    else
    {
        mtCOVERAGE_TEST_MARKER();
    }

    if( ( xWantedSize > 0 ) && ( xWantedSize <= xFreeBytesRemaining ) )
    {
        //从 xStart(内存块最小)开始，查找大小满足所需要内存的内存块。
        pxPreviousBlock = &xStart;
        pxBlock = xStart.pxNextFreeBlock;
        while( ( pxBlock->xBlockSize < xWantedSize ) &&\               (4)
               ( pxBlock->pxNextFreeBlock != NULL ) )
        {
            pxPreviousBlock = pxBlock;
            pxBlock = pxBlock->pxNextFreeBlock;
        }

        //如果找到的内存块是 pxEnd 的话就表示没有内存可以分配
        if( pxBlock != pxEnd )                                         (5)
        {
            pvReturn = ( void * ) ( ( ( uint8_t * ) pxPreviousBlock->\ (6)
```

```
                                        pxNextFreeBlock ) + xHeapStructSize );
                //将申请到的内存块从空闲内存链表中移除
                pxPreviousBlock->pxNextFreeBlock = pxBlock->pxNextFreeBlock;    (7)

                //如果申请到的内存块大于所需的内存，就将其分成两块
                if( ( pxBlock->xBlockSize - xWantedSize ) >\                    (8)
                            heapMINIMUM_BLOCK_SIZE )
                {
                    pxNewBlockLink = ( void * ) ( ( ( uint8_t * ) pxBlock ) + xWantedSize );
                    pxNewBlockLink->xBlockSize = pxBlock->xBlockSize - xWantedSize;
                    pxBlock->xBlockSize = xWantedSize;
                    prvInsertBlockIntoFreeList( pxNewBlockLink );               (9)
                }
                else
                {
                    mtCOVERAGE_TEST_MARKER();
                }
                xFreeBytesRemaining -= pxBlock->xBlockSize;                     (10)
                if( xFreeBytesRemaining < xMinimumEverFreeBytesRemaining )
                {
                    xMinimumEverFreeBytesRemaining = xFreeBytesRemaining;
                }
                else
                {
                    mtCOVERAGE_TEST_MARKER();
                }

                //内存块申请成功，标记此内存块已经被时候
                pxBlock->xBlockSize |= xBlockAllocatedBit;                      (11)
                pxBlock->pxNextFreeBlock = NULL;
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
            }
        }
        else
        {
            mtCOVERAGE_TEST_MARKER();
        }
    }
    else
    {
```

```
                mtCOVERAGE_TEST_MARKER();
            }
            traceMALLOC( pvReturn, xWantedSize );
        }
        ( void ) xTaskResumeAll();

        #if( configUSE_MALLOC_FAILED_HOOK == 1 )
        {
            if( pvReturn == NULL )
            {
                //调用钩子函数
                extern void vApplicationMallocFailedHook( void );
                vApplicationMallocFailedHook();
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
            }
        }
        #endif

        configASSERT( ( ( ( size_t ) pvReturn ) & ( size_t ) portBYTE_ALIGNMENT_MASK ) == 0 );
        return pvReturn;
}
```

(1)、pxEnd 为 NULL，说明内存堆还没初始化，所以需要调用函数 prvHeapInit()初始化内存堆。

(2)、BlockLink_t 中的变量 xBlockSize 是来描述内存块大小的，其最高位用来记录内存块有没有被使用，所以申请的内存块大小最高位不能为 1。

(3)、实际所需申请的内存数要加上结构体 BlockLink_t 的大小，因为内存块前面需要保存一个 BlockLink_t 类型的变量。最后还需要对最终的大小做字节对齐处理。这里有个疑问，假如 xWantedSize 为 0x7FFFFFFF，那么 xWantedSize 加上结构体 BlockLink_t 的大小就是 0x7FFFFFFF+8=0x80000007，在做一次 8 字节对齐 xWantedSize 就是 0x80000008，其最高位为 1。前面已经说了，BlockLink_t 中的变量 xBlockSize 的最高位是用来标记内存块是否被使用的，这里明显冲突了，但是 FreeRTOS 对此并没有做处理。

(4)、从空闲内存链表头 xStart 开始，查找满足所需内存大小的内存块，pxPreviousBlock 的下一个内存块就是找到的可用内存块。

(5)、找到的可用内存块不能是链表尾 pxEnd！

(6)、找到内存块以后就将内存首地址保存在 pvReturn 中，函数返回的时候返回此值。

(7)、内存块已经被申请了，所以需要将这个内存块从空闲内存块链表中移除。

(8)、申请到的内存块大于所需的大小，因此要把多余出来的内存重新组合成一个新的可用空闲内存块。

(9)、将新的空闲内存块插入到空闲内存块链表中。

(10)、更新全局变量 xFreeBytesRemaining 和 xMinimumEverFreeBytesRemaining。

(11)、xBlockSize 与 xBlockAllocatedBit 进行或运算，也就是将 xBlockSize 的最高位置 1，表示此内存块被使用。

20.6.5 内存释放函数详解

内存释放函数源码如下：

```
void vPortFree( void *pv )
{
    uint8_t *puc = ( uint8_t * ) pv;
    BlockLink_t *pxLink;

    if( pv != NULL )
    {
        puc -= xHeapStructSize;                                          (1)
        pxLink = ( void * ) puc;        //防止编译器报错

        configASSERT( ( pxLink->xBlockSize & xBlockAllocatedBit ) != 0 );
        configASSERT( pxLink->pxNextFreeBlock == NULL );

        if( ( pxLink->xBlockSize & xBlockAllocatedBit ) != 0 )           (2)
        {
            if( pxLink->pxNextFreeBlock == NULL )
            {
                pxLink->xBlockSize &= ~xBlockAllocatedBit;               (3)
                vTaskSuspendAll();
                {
                    xFreeBytesRemaining += pxLink->xBlockSize;
                    traceFREE( pv, pxLink->xBlockSize );
                    prvInsertBlockIntoFreeList( ( ( BlockLink_t * ) pxLink ) );   (4)
                }
                ( void ) xTaskResumeAll();
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
            }
        }
        else
        {
            mtCOVERAGE_TEST_MARKER();
        }
    }
}
```

(1)、获取内存块的 BlockLink_t 类型结构体。

(2)、要释放的内存块肯定是被使用了的，没有被使用的空闲内存块肯定没有释放这一说。这里通过判断 xBlockSize 的最高位是否等于 0 来得知要释放的内存块是否被应用使用。前面已经说了 BlockLink_t 中成员变量 xBlockSize 的最高位用来表示此内存块有没有被使用。

(3)、xBlockSize 的最高位清零，重新标记此内存块没有使用。xBlockSize 也表示内存块大小，就跟在 pvPortMalloc()函数里面分析的一样，如果最终申请的内存大小是 0x80000008，那么在经过这一行代码处理之后这个大小就变成了 0x00000008。所以一定要保证在使用函数 pvPortMalloc()申请内存的时候经过字节对齐等处理以后，最后申请大小不能超过 0x7FFFFFFF。

(4)、将内存块插到空闲内存链表中。

20.7 heap_5 内存分配方法

heap_5 使用了和 heap_4 相同的合并算法，内存管理实现起来基本相同，但是 heap_5 允许内存堆跨越多个不连续的内存段。比如 STM32 的内部 RAM 可以作为内存堆，但是 STM32 内部 RAM 比较小，遇到那些需要大容量 RAM 的应用就不行了，如音视频处理。不过 STM32 可以外接 SRAM 甚至大容量的 SDRAM，如果使用 heap_4 的话你就只能在内部 RAM 和外部 SRAM 或 SDRAM 之间二选一了，使用 heap_5 的话就不存在这个问题，两个都可以一起作为内存堆来用。

如果使用 heap_5 的话，在调用 API 函数之前需要先调用函数 vPortDefineHeapRegions ()来对内存堆做初始化处理，在 vPortDefineHeapRegions()未执行完之前禁止调用任何可能会调用 pvPortMalloc()的 API 函数！比如创建任务、信号量、队列等函数。函数 vPortDefineHeapRegions()只有一个参数，参数是一个 HeapRegion_t 类型的数组，HeapRegion 为一个结构体，此结构体在 portable.h 中有定义，定义如下：

```
typedef struct HeapRegion
{
    uint8_t    *pucStartAddress;        //内存块的起始地址
    size_t     xSizeInBytes;            //内存段大小
} HeapRegion_t;
```

上面说了，heap_5 允许内存堆跨越多个不连续的内存段，这些不连续的内存段就是由结构体 HeapRegion_t 来定义的。比如以 STM32F429 为例，现在有三个内存段：CCM、内部 SRAM、外部 SDRAM，起始分别为：0X10000000、0X20000000、0XC0000000，大小分别为：64KB、192KB、32MB，那么数组就如下：

```
HeapRegion_t xHeapRegions[] =
{
    { ( uint8_t * ) 0X10000000UL,    0x10000 }, //CCM 内存，起始地址 0X10000000,大小 64KB
    { ( uint8_t * ) 0X20000000UL,    0x30000 },//内部 SRAM 内存，起始地址 0X20000000,
                                               //大小为 192KB
    { ( uint8_t * ) 0XC0000000UL, 0x2000000},//外部 SDRAM 内存，起始地址 0XC0000000，
                                               //大小为 32MB
    { NULL, 0 }                                //数组结尾
};
```

注意，数组中成员顺序按照地址从低到高的顺序排列，而且最后一个成员必须使用 NULL。heap_5 允许内存堆不连续，说白了就是允许有多个内存堆。在 heap_2 和 heap_4 中只有一个内存堆，初始化的时候只也只需要处理一个内存堆。 heap_5 有多个内存堆，这些内存堆会被连接在一起，和空闲内存块链表类似，这个处理过程由函数 vPortDefineHeapRegions()完成。

使用 heap_5 的时候在一开始就应该先调用函数 vPortDefineHeapRegions()完成内存堆的初始化！然后才能创建任务、信号量这些东西，如下示例代码：

```
int main(void)
{
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(360,25,2,8);   //设置时钟,180Mhz
    delay_init(180);                //初始化延时函数
    uart_init(115200);              //初始化串口
    LED_Init();                     //初始化 LED
    KEY_Init();                     //初始化按键
    PCF8574_Init();                 //初始化 PCF8574
    SDRAM_Init();                   //初始化 SDRAM
    LCD_Init();                     //初始化 LCD

    //使用 heap_5 的时候在开启任务调度器、创建任务、创建信号量之前一定要先
    //调用函数 vPortDefineHeapRegions()初始化内存堆！
    vPortDefineHeapRegions((const HeapRegion_t *)xHeapRegions);

    //创建开始任务
    xTaskCreate((TaskFunction_t     )start_task,            //任务函数
                (const char*        )"start_task",          //任务名称
                (uint16_t           )START_STK_SIZE,        //任务堆栈大小
                (void*              )NULL,                  //传递给任务函数的参数
                (UBaseType_t        )START_TASK_PRIO,       //任务优先级
                (TaskHandle_t*      )&StartTask_Handler);   //任务句柄
    vTaskStartScheduler();                                  //开启任务调度
}
```

heap_5 的内存申请和释放函数和 heap_4 基本一样，这里就不详细讲解了，大家可以对照着前面 heap_4 的相关内容来自行分析。

至此，FreeRTOS 官方提供的 5 种内存分配方法已经讲完了，heap_1 最简单，但是只能申请内存，不能释放。heap_2 提供了内存释放函数，用户代码也可以直接调用函数 pvPortMalloc()和 vPortFree()来申请和释放内存，但是 heap_2 会导致内存碎片的产生！heap_3 是对标准 C 库中的函数 malloc()和 free()的简单封装，并且提供了线程保护。heap_4 相对与 heap_2 提供了内存合并功能，可以降低内存碎片的产生，我们移植 FreeRTOS 的时候就选择了 heap_4。heap_5 基本上和 heap_4 一样，只是 heap_5 支持内存堆使用不连续的内存块。

20.8 内存管理实验

20.8.1 实验程序设计

1、实验目的

本节我们设计一个小程序来学习使用 FreeRTOS 的内存申请和释放函数：pvPortMalloc()、vPortFree()，并且观察申请和释放的过程中内存大小的变化情况

2、实验设计

本实验设计两个任务：start_task 和 malloc_task ，这两个任务的任务功能如下：

start_task：用来创建另外一个任务。

malloc_task ：此任务用于完成内存的申请、释放和使用功能。任务会不断的获取按键情况，当检测到 KEY_UP 按下的时候就会申请内存，当 KEY0 按下以后就会使用申请到的内存，如果检测到 KEY1 按下的话就会释放申请到的内存。

实验中会用到 3 个按键：KEY0、KEY1 和 KEY_UP，KEY_UP 用于申请内存，KEY0 使用申请到的内存，KEY1 释放申请到的内存。

3、实验工程

FreeRTOS 实验 20-1 FreeRTOS 内存管理实验。

4、实验程序与分析

●任务设置

```
#define START_TASK_PRIO          1        //任务优先级
#define START_STK_SIZE           128      //任务堆栈大小
TaskHandle_t StartTask_Handler;           //任务句柄
void start_task(void *pvParameters);      //任务函数

#define MALLOC_TASK_PRIO         2        //任务优先级
#define MALLOC_STK_SIZE          128      //任务堆栈大小
TaskHandle_t MallocTask_Handler;          //任务句柄
void malloc_task(void *p_arg);            //任务函数
```

● main()函数

```
int main(void)
{
    HAL_Init();                           //初始化 HAL 库
    Stm32_Clock_Init(360,25,2,8);         //设置时钟,180Mhz
    delay_init(180);                      //初始化延时函数
    uart_init(115200);                    //初始化串口
    LED_Init();                           //初始化 LED
    KEY_Init();                           //初始化按键
    PCF8574_Init();                       //初始化 PCF8574
    SDRAM_Init();                         //初始化 SDRAM
    LCD_Init();                           //初始化 LCD
    my_mem_init(SRAMIN);                  //初始化内部内存池

    POINT_COLOR = RED;
    LCD_ShowString(30,10,200,16,16,"Apollo STM32F4/F7");
    LCD_ShowString(30,30,200,16,16,"FreeRTOS Examp 17-1");
    LCD_ShowString(30,50,200,16,16,"Mem Manage");
    LCD_ShowString(30,70,200,16,16,"KEY_UP:Malloc,KEY1:Free");
```

```
    LCD_ShowString(30,90,200,16,16,"KEY0:Use Mem");
    LCD_ShowString(30,110,200,16,16,"ATOM@ALIENTEK");
    LCD_ShowString(30,130,200,16,16,"2016/11/14");

    LCD_ShowString(30,170,200,16,16,"Total Mem:      Bytes");
    LCD_ShowString(30,190,200,16,16,"Free  Mem:      Bytes");
    LCD_ShowString(30,210,200,16,16,"Message:            ");
    POINT_COLOR = BLUE;

    //创建开始任务
    xTaskCreate((TaskFunction_t     )start_task,                //任务函数
                (const char*        )"start_task",              //任务名称
                (uint16_t           )START_STK_SIZE,            //任务堆栈大小
                (void*              )NULL,                      //传递给任务函数的参数
                (UBaseType_t        )START_TASK_PRIO,           //任务优先级
                (TaskHandle_t*      )&StartTask_Handler);       //任务句柄
    vTaskStartScheduler();                                      //开启任务调度
}
```

● **任务函数**

```
//开始任务任务函数
void start_task(void *pvParameters)
{
    taskENTER_CRITICAL();           //进入临界区
    //创建 TASK1 任务
    xTaskCreate((TaskFunction_t     )malloc_task,
                (const char*        )"malloc_task",
                (uint16_t           )MALLOC_STK_SIZE,
                (void*              )NULL,
                (UBaseType_t        )MALLOC_TASK_PRIO,
                (TaskHandle_t*      )&MallocTask_Handler);
    vTaskDelete(StartTask_Handler);     //删除开始任务
    taskEXIT_CRITICAL();            //退出临界区
}

//MALLOC 任务函数
void malloc_task(void *pvParameters)
{
    u8 *buffer;
    u8 times,i,key=0;
    u32 freemem;

    LCD_ShowxNum(110,170,configTOTAL_HEAP_SIZE,5,16,0);//显示内存总容量      (1)
    while(1)
```

```
    {
        key=KEY_Scan(0);
        switch(key)
        {
            case WKUP_PRES:
                buffer=pvPortMalloc(30);                //申请内存，30 个字节          (2)
                printf("申请到的内存地址为:%#x\r\n",(int)buffer);
                break;
            case KEY1_PRES:
                if(buffer!=NULL)vPortFree(buffer);      //释放内存                    (3)
                buffer=NULL;                                                        (4)
                break;
            case KEY0_PRES:
                if(buffer!=NULL)                        //buffer 可用,使用 buffer     (5)
                {
                    times++;
                    sprintf((char*)buffer,"User %d Times",times);//向 buffer 中填写一些数据
                    LCD_ShowString(94,210,200,16,16,buffer);
                }
                break;
        }
        freemem=xPortGetFreeHeapSize();             //获取剩余内存大小             (6)
        LCD_ShowxNum(110,190,freemem,5,16,0);//显示内存总容量
        i++;
        if(i==50)
        {
            i=0;
            LED0=~LED0;
        }
        vTaskDelay(10);
    }
}
```

(1)、显示内存堆的总容量，内存堆的容量由宏 configTOTAL_HEAP_SIZE 来确定的，所以直接显示 configTOTAL_HEAP_SIZE 的值就行了。

(2)、按下 KEY_UP 键，调用函数 pvPortMalloc()申请内存，大小为 30 字节。

(3)、按下 KEY1 键，释放(2)中申请到的内存。

(4)、释放内存以后将 buffer 设置为 NULL，函数 vPortFree()释放内存以后并不会将 buffer 清零，此时 buffer 还是上次申请到的内存地址，如果此时再调用指针 buffer 的话你会发现还是可以使用的！感觉好像没有释放内存，所以这里将 buffer 清零！

(5)、判断 buffer 是否有效，有效的话就使用 buffer。

(6)、调用函数 xPortGetFreeHeapSize()获取当前剩余内存大小并且显示到 LCD 上。

20.8.2 实验程序运行结果

编译并下载代码到开发板中，LCD 显示如图 20.8.2.1 所示：

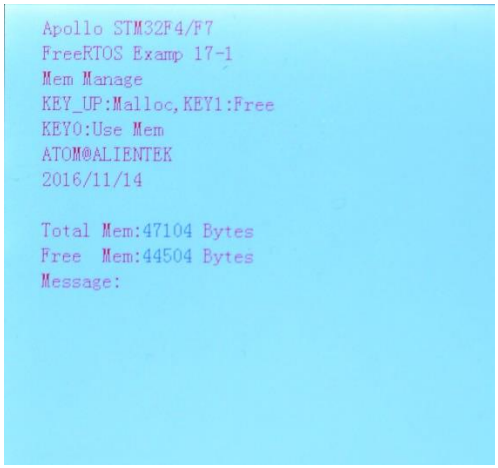


图 20.8.2.1 LCD 显示界面

可以看出内存堆的总容量为 47104 字节，再 FreeRTOSConfig.h 文件中我们设置的内存堆大小如下：

```
#define configTOTAL_HEAP_SIZE          ((size_t)(46*1024))     //46*1024=47104
```

此时剩余内存大小为 44504 字节，因为前面已经创建了一些用户任务和系统任务，所以内存肯定会被使用掉一部分。按下 KEY_UP 键，申请内存，内存申请成功 LCD 显示如图 20.8.2.2 所示：

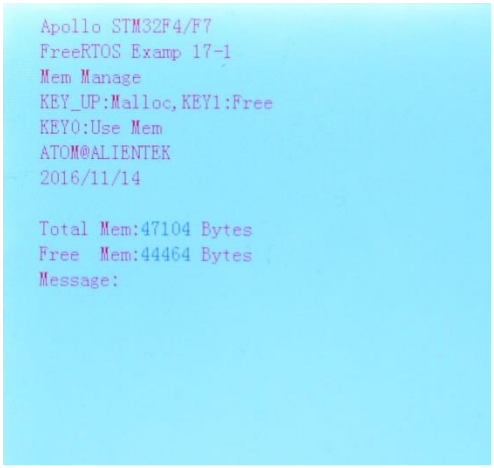


图 20.8.2.2 内存申请成功

内存申请成功以后会通过串口打印出当前申请到的内存首地址，串口调试助手如图 20.8.2.3 所示：

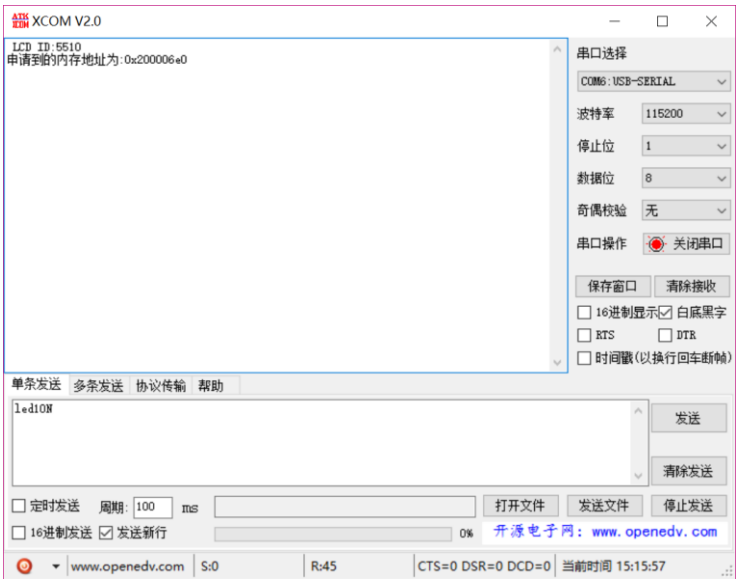


图 20.8.2.3 串口调试助手

从图 20.8.2.3 可以看出此时申请到的内存的首地址为 0x200006e0。不过此时有的同学可能注意到了，图 20.4.2.2 中剩余内存大小为 44464，44504-44464=40，而我们申请的是 30 个字节的内存！内存为什么会少 40 个字节呢？多余的 10 个字节是怎么回事？前面分析 heap_4 内存分配方法的时候已经说了，这是应用所需的内存大小上加上结构体 BlockLink_t 的大小然后做 8 字节对齐后导致的结果。

按下 KEY1 键释放内存，此时剩余内存大小又重新变成了 44504，说明内存释放成功！

内存泄露：

在使用内存管理的时候最常遇到的一个问题就是内存泄露，内存泄露的原因是没有正确的释放内存！以本实验为例，连续 5 次按下 KEY_UP 键，此时 LCD 上显示的剩余内存为 44304，如图 20.8.2.4 所示：

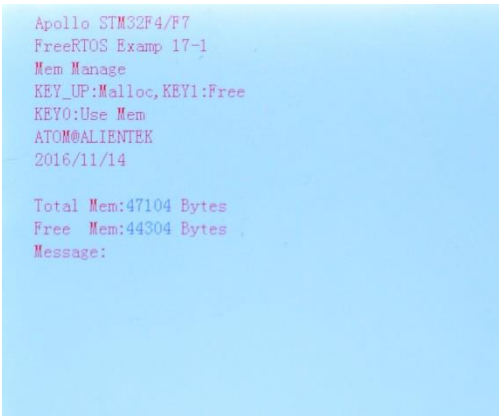


图 20.8.2.4 连续多次申请内存

44504-44304=200，连续 5 次一共申请了 200 字节的内存，上面说了一次申请 40 个字节的，5 次肯定就是 40*5=200 字节，这个没有错。然后释放内存，连续按 5 次 KEY1，此时 LCD 显示剩余内存为 44344，如图 20.8.2.5 所示:

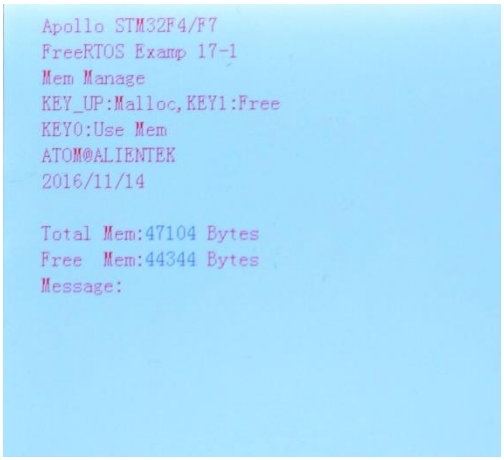


图 20.8.2.5 连续多次释放内存

44504-44344=160，竟然少了 160 个字节的内存，也就是说实际上只释放了一次内存，其他的 4 次是无效的！为什么会这样？这个就是内存泄露，泄露了 160 个字节的内存，这个是不正确的内存申请和释放导致的，内存申请和释放是要成对出现的，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！我们连续 5 次按 KEY_UP 为 buffer 申请内存，就犯了这个错误，正确的方法应该是，按一次 KEY_UP 按键申请内存成功以后，先按 KEY1 释放掉 buffer 的内存，然后再按 KEY_UP 为其重新分配内存。初学者很容易犯这样的错误，忘记释放内存，内存泄露严重的话应用可能因为申请不到合适的内存而导致死机！