[野火]uCOS-III 内核实现与应用开发实战指南——基于 STM32

EmbedFire
野火电子

2021年01月20日
Contents

关于本项目 ................................................................. 1
关于野火 ................................................................. 2
开源共享，共同进步 .................................................. 2
联系方式 ................................................................. 2
第 1 章 前言 ................................................................ 3
  1.1 如何学习本书 .................................................. 3
  1.2 本书的编写风格 .................................................. 3
  1.3 本书的技术论坛 .................................................. 4
  1.4 本书的参考资料 .................................................. 4
  1.5 本书的配套硬件 .................................................. 4
第 2 章 引言 .................................................................. 11
  2.1 为什么学习 RTOS ................................................. 11
  2.2 如何学习 RTOS .................................................. 11
  2.3 如何选择 RTOS .................................................. 12
第 3 章 第一部分从 0 到 1 教你写 µC/OS 内核 ..................... 14
第 4 章 新建工程—软件仿真 ......................................... 15
  4.1 新建本地工程文件夹 ............................................. 15
  4.2 使用 KEIL 新建工程 ............................................. 17
    4.2.1 New Project .................................................. 17
    4.2.2 Select Device For Target ................................ 17
    4.2.3 Manage Run-Time Environment ........................ 18
  4.3 在 KEIL 工程里面新建文件组 ............................... 20
  4.4 在 KEIL 工程里面添加文件 ................................ 21
    4.4.1 编写 main() 函数 ....................................... 22
  4.5 调试配置 .......................................................... 23
    4.5.1 设置软件仿真 ............................................. 23

论坛：https://www.firebbs.cn  天猫：https://yehuosm.tmall.com
第 5 章 裸机系统与多任务系统 ........................................... 27
  5.1 裸机系统 ......................................................... 27
     5.1.1 轮询系统 .................................................. 27
     5.1.2 前后台系统 ................................................ 28
  5.2 多任务系统 ...................................................... 31
第 6 章 任务的定义与任务切换的实现 ..................................... 35
  6.1 本章目标 ....................................................... 35
  6.2 什么是任务 ..................................................... 37
  6.3 创建任务 ....................................................... 38
     6.3.1 定义任务栈 ............................................... 38
     6.3.2 定义任务函数 ............................................... 40
     6.3.3 定义任务控制块 TCB ...................................... 41
     6.3.4 实现任务创建函数 .......................................... 42
  6.4 OS 系统初始化 ................................................ 50
  6.5 启动系统 ....................................................... 52
  6.6 任务切换 ....................................................... 58
  6.7 main() 函数 ................................................... 70
  6.8 实验现象 ....................................................... 76
第 7 章 任务时间片运行 ................................................... 80
  7.1 SysTick 简介 ................................................... 80
  7.2 初始化 SysTick ................................................ 82
  7.3 编写 SysTick 中断服务函数 ..................................... 84
  7.4 main() 函数 ................................................... 85
  7.5 实验现象 ....................................................... 88
第 8 章 阻塞延时与空闲任务 ............................................... 90
  8.1 实现空闲任务 .................................................. 90
     8.1.1 定义空闲任务栈 .......................................... 90
     8.1.2 定义空闲任务 TCB .......................................... 92
     8.1.3 定义空闲任务函数 ......................................... 92
第 9 章 时间截 ................................. 104
  9.1 时间截简介 .............................. 104
  9.2 时间截的实现 ............................ 104
  9.3 时间截代码讲解 ......................... 105
      9.3.1 CPU_Init() 函数 .................. 105
      9.3.2 CPU_TS_Init() 函数 ................ 107
      9.3.3 CPU_TS_TmrInit() 函数 .......... 109
      9.3.4 BSP_CPU_ClkFreq() 函数 .......... 111
      9.3.5 CPU_TS_TmrFreqSet() 函数 .......... 112
      9.3.6 CPU_TS_TmrRd() 函数 ............ 112
      9.3.7 OS_TS_GET() 函数 ............... 113
  9.4 main() 函数 ............................ 114
  9.5 实验现象 ............................... 116
第 10 章 临界段 ................................. 117
  10.1 临界段简介 ............................ 117
  10.2 Cortex-M 内核快速关中断指令 .... 118
  10.3 关中断 ................................. 119
  10.4 开中断 ................................. 119
  10.5 临界段代码的应用 ................... 120
  10.6 测量关中断时间 ....................... 126
      10.6.1 测量关中断时间初始化 ......... 127
      10.6.2 测量最大关中断时间 .......... 128
      10.6.3 获取最大关中断时间 .......... 130
  10.7 main() 函数 ............................ 132
  10.8 实验现象 ............................... 132
第 11 章 就绪列表 ............................... 133
  11.1 优先级表 ............................... 133
<table>
<thead>
<tr>
<th>章节</th>
<th>内容</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.1</td>
<td>优先级表函数讲解</td>
</tr>
<tr>
<td>11.2</td>
<td>就绪列表</td>
</tr>
<tr>
<td>11.2.1</td>
<td>就绪列表函数讲解</td>
</tr>
<tr>
<td>11.3</td>
<td>main() 函数</td>
</tr>
<tr>
<td>11.4</td>
<td>实验现象</td>
</tr>
<tr>
<td>12.1</td>
<td>定义优先级相关全局变量</td>
</tr>
<tr>
<td>12.2</td>
<td>修改 OSInit() 函数</td>
</tr>
<tr>
<td>12.3</td>
<td>修改任务控制块 TCB</td>
</tr>
<tr>
<td>12.4</td>
<td>修改 OSTaskCreate() 函数</td>
</tr>
<tr>
<td>12.5</td>
<td>修改 OS_IdleTaskInit() 函数</td>
</tr>
<tr>
<td>12.6</td>
<td>修改 OSStart() 函数</td>
</tr>
<tr>
<td>12.7</td>
<td>修改 PendSV_Handler() 函数</td>
</tr>
<tr>
<td>12.8</td>
<td>修改 OSTimeDly() 函数</td>
</tr>
<tr>
<td>12.9</td>
<td>修改 OSSched() 函数</td>
</tr>
<tr>
<td>12.10</td>
<td>修改 OSTimeTick() 函数</td>
</tr>
<tr>
<td>12.11</td>
<td>main() 函数</td>
</tr>
<tr>
<td>12.12</td>
<td>实验现象</td>
</tr>
<tr>
<td>13.1</td>
<td>实现时基列表</td>
</tr>
<tr>
<td>13.1.1</td>
<td>定义时基列表变量</td>
</tr>
<tr>
<td>13.1.2</td>
<td>修改任务控制块 TCB</td>
</tr>
<tr>
<td>13.1.3</td>
<td>实现时基列表相关函数</td>
</tr>
<tr>
<td>13.2</td>
<td>修改 OSTimeDly() 函数</td>
</tr>
<tr>
<td>13.3</td>
<td>修改 OSTimeDly() 函数</td>
</tr>
<tr>
<td>13.4</td>
<td>main 函数</td>
</tr>
<tr>
<td>13.5</td>
<td>实验现象</td>
</tr>
<tr>
<td>14.1</td>
<td>实现时间片</td>
</tr>
<tr>
<td>14.1.1</td>
<td>修改任务 TCB</td>
</tr>
<tr>
<td>14.1.2</td>
<td>实现时间片调度函数</td>
</tr>
</tbody>
</table>
第 15 章 任务的挂起和恢复.................................................. 216
  15.1 实现任务的挂起和恢复 ........................................... 216
    15.1.1 定义任务的状态 ........................................... 216
    15.1.2 修改任务控制块 TCB ................................... 217
    15.1.3 编写任务挂起和恢复函数 ................................. 219
  15.2 main() 函数 ....................................................... 227
  15.3 实验现象 .......................................................... 230
第 16 章 任务的删除 ...................................................... 231
  16.1 实现任务删除 .................................................... 231
    16.1.1 编写任务删除函数 ....................................... 231
  16.2 main() 函数 ....................................................... 235
  16.3 实验现象 .......................................................... 235
第 17 章 第二部分：μC/OS 内核应用开发 ............................. 236
第 18 章 移植μC/OS-III 到 STM32 ....................................... 237
  18.1 获取 STM32 的裸机工程模板 ................................... 237
  18.2 下载μC/OS-III 源码 ............................................ 238
  18.3 μC/OS-III 源码文件介绍 ....................................... 242
    18.3.1 EvalBoards ............................................... 243
    18.3.2 uC-CPU .................................................... 246
    18.3.3 uC-LIB ..................................................... 247
    18.3.4 μC/OS-III ............................................... 248
  18.4 移植到 STM32 工程 ............................................... 251
    18.4.1 在工程中添加文件分组 ................................... 251
    18.4.2 添加文件到对应分组 ..................................... 252
    18.4.3 添加头文件路径到工程中 ................................. 255
    18.4.4 具体的工程文件修改 ..................................... 256
[野火]μCOS-III 内核实现与应用开发实战指南

18.4.5  修改源码中的 bsp.c 与 bsp.h 文件 ......................................................... 259
18.5  按需配置最适的工程 ................................................................. 263
  18.5.1  os_cfg.h ................................................................. 263
  18.5.2  cpu_cfg.h ................................................................. 266
  18.5.3  os_cfg_app.h ................................................................. 268
18.6  修改 app.c ................................................................. 270
18.7  下载验证 ................................................................. 272

第 19 章  创建任务 ................................................................. 273
  19.1  硬件初始化 ................................................................. 273
  19.2  创建单任务 ................................................................. 276
    19.2.1  定义任务栈 ................................................................. 276
    19.2.2  定义任务控制块 ................................................................. 276
    19.2.3  定义任务主体函数 ................................................................. 277
    19.2.4  创建任务 ................................................................. 278
    19.2.5  启动任务 ................................................................. 280
    19.2.6  app.c 全貌 ................................................................. 281
  19.3  下载验证 ................................................................. 284
  19.4  创建多任务 ................................................................. 284
  19.5  下载验证 ................................................................. 292

第 20 章  μC/OS-III 的启动流程 ................................................................. 293
  20.1  万事俱备，只欠东风 ................................................................. 293
  20.2  小心翼翼，十分谨慎 ................................................................. 295
  20.3  孰优孰劣 ................................................................. 297
  20.4  系统的启动 ................................................................. 297
    20.4.1  系统初始化 ................................................................. 298
    20.4.2  CPU 初始化 ................................................................. 305
    20.4.3  SysTick 初始化 ................................................................. 310
    20.4.4  内存初始化 ................................................................. 311
    20.4.5  OSStart() ................................................................. 311
    20.4.6  app.c ................................................................. 313

论坛: https://www.firebbs.cn  vi  天猫: https://yehuosm.tmall.com
第 21 章 任务管理 .................................................. 318
  21.1 任务的基本概念 ............................................. 318
  21.2 任务调度器的基本概念 ...................................... 319
  21.3 任务状态迁移 ................................................. 320
  21.4 μC/OS 的任务状态 ............................................. 321
  21.5 常用的任务函数讲解 .......................................... 322
    21.5.1 任务挂起函数 OSTaskSuspend() ...................... 323
    21.5.2 任务恢复函数 OSTaskResume() ....................... 327
    21.5.3 删除任务函数 OSTD del() ............................. 334
    21.5.4 任务延时函数 ........................................... 340
  21.6 任务的设计要点 .............................................. 358
  21.7 任务管理实验 ............................................... 360
  21.8 实验现象 ................................................... 370

第 22 章 消息队列 .................................................. 372
  22.1 消息队列的基本概念 ....................................... 372
  22.2 消息队列工作过程 .......................................... 372
    22.2.1 消息池初始化 .......................................... 373
    22.2.2 消息队列的运作机制 .................................. 376
  22.3 消息队列的阻塞机制 ....................................... 377
  22.4 消息队列的应用场景 ....................................... 378
  22.5 消息队列的结构 ............................................. 379
  22.6 消息队列常用函数讲解 .................................... 383
    22.6.1 创建消息队列函数 OSQCreate() .................... 383
    22.6.2 消息队列删除函数 OSQDel() ........................ 389
    22.6.3 消息队列发送函数 OSQPost() ........................ 398
    22.6.4 消息队列获取函数 OSQPend() ....................... 414
  22.7 消息队列使用注意事项 .................................... 426
  22.8 消息队列实验 ............................................... 427
  22.9 消息队列实验现象 .......................................... 435

第 23 章 信号量 .................................................... 437
  23.1 信号量基本概念 ............................................. 437
23.1.1 二值信号量 .......................... 437
23.1.2 计数信号量 ........................... 438
23.2 信号量应用场景 ......................... 439
23.3 二值信号量运作机制 .................... 440
23.4 计数信号量运作机制 .................... 441
23.5 信号量控制块 ........................... 442
23.6 信号量函数接口讲解 .................... 444
   23.6.1 创建信号量函数 OSSemCreate() .... 444
   23.6.2 信号量删除函数 OSSemDel() ........ 449
   23.6.3 信号量释放函数 OSSemPost() ....... 455
   23.6.4 信号量获取函数 OSSemPend() ....... 464
23.7 使用信号量的注意事项 ................ 472
23.8 信号量实验 ............................ 473
   23.8.1 二值信号量同步实验 ............... 473
   23.8.2 计数信号量实验 .................. 482
23.9 信号量实验现象 ....................... 491
   23.9.1 二值信号量同步实验现象 ........... 491
   23.9.2 计数信号量实验现象 ............... 492

第 24 章 互斥量 .................................. 494
24.1 互斥量的基本概念 ....................... 494
24.2 互斥量的优先级继承机制 ............... 495
24.3 互斥量应用场景 ....................... 498
24.4 互斥量运作机制 ....................... 499
24.5 互斥量控制块 ......................... 500
24.6 互斥量函数接口讲解 .................... 503
   24.6.1 创建互斥量函数 OSMutexCreate() .. 503
   24.6.2 删除互斥量函数 OSMutexDel() .... 508
   24.6.3 获取互斥量函数 OSMutexPend() ..... 516
   24.6.4 释放互斥量函数 OSMutexPost() .... 526
24.7 实验 .................................. 533
   24.7.1 模拟优先级翻转实验 ............... 533
第 25 章 事件 ................................................. 553
  25.1 事件的基本概念 ..................................... 553
  25.2 事件的应用场景 ..................................... 554
  25.3 事件运作机制 ....................................... 555
  25.4 事件控制块 ......................................... 557
  25.5 事件函数接口 ....................................... 559
    25.5.1 事件创建函数 OSFlagCreate() .................. 559
    25.5.2 事件删除函数 OSFlagDel() ..................... 563
    25.5.3 事件设置函数 OSFlagPost() ................... 569
    25.5.4 事件等待函数 OSFlagPend() ................... 580
  25.6 事件实验 ........................................... 598
  25.7 事件实验现象 ....................................... 605

第 26 章 软件定时器 ............................................. 607
  26.1 软件定时器的基本概念 ................................ 607
  26.2 软件定时器应用场景 ................................ 609
  26.3 软件定时器的精度 ................................... 610
  26.4 软件定时器控制块 ................................... 610
  26.5 软件定时器函数接口讲解 ................................ 614
    26.5.1 创建软件定时器函数 OSTmrCreate() ............. 614
    26.5.2 启动软件定时器函数 OSTmrStart() ............... 618
    26.5.3 软件定时器列表管理 ............................. 626
    26.5.4 停止定时器函数 OSTmrStop() ................... 631
    26.5.5 删除软件定时器函数 OSTmrDel() ................. 637
  26.6 软件定时器任务 ..................................... 641
  26.7 软件定时器实验 ..................................... 650
  26.8 软件定时器实验现象 ................................ 656
[野火]uCOS-III 内核实现与应用开发实战指南

第 27 章 任务信号量 ........................................ 659
  27.1 任务信号量的基本概念 .................................. 659
  27.2 任务信号量的函数接口讲解 ................................. 660
    27.2.1 任务信号量释放函数 OSTaskSemPost() .............. 660
    27.2.2 获取任务信号量函数 OSTaskSemPend() .............. 668
  27.3 任务信号量实验 ........................................... 674
    27.3.1 任务信号量代替二值信号量 .......................... 674
    27.3.2 任务信号量代替计数信号量 .......................... 682
  27.4 任务信号量实验现象 ........................................ 689
    27.4.1 任务信号量代替二值信号量 .......................... 689
    27.4.2 任务信号量代替计数信号量 .......................... 690

第 28 章 任务消息队列 .......................................... 692
  28.1 任务消息队列的基本概念 .................................. 692
  28.2 任务消息队列的函数接口讲解 ................................. 693
    28.2.1 任务消息队列发送函数 OSTaskQPost() .......... 693
    28.2.2 任务消息队列获取函数 OSTaskQPend() ........... 699
  28.3 任务消息队列实验 ........................................... 705
  28.4 任务消息队列实验现象 ........................................ 712

第 29 章 内存管理 ................................................. 714
  29.1 内存管理的基本概念 ........................................ 714
  29.2 内存管理的运作机制 ........................................ 715
  29.3 内存管理的应用场景 ........................................ 717
  29.4 内存管理函数接口讲解 ...................................... 718
    29.4.1 内存池创建函数 ....................................... 718
    29.4.2 内存申请函数 OSMemGet() .......................... 724
    29.4.3 内存释放函数 ....................................... 727
  29.5 内存管理实验 ............................................... 730
  29.6 内存管理实验现象 ........................................... 737

第 30 章 中断管理 .................................................. 739
  30.1 异常与中断的基本概念 ..................................... 739

论坛：https://www.firebbs.cn
天猫：https://yehuosl.tmall.com
第31章 CPU利用率及栈检测统计

第32章 rst投稿示例文件

如何参与项目

项目源码

文档在线阅读与pdf

常见问题

版权说明
关于本项目

本项目通过 git 开源：

- github 仓库地址：https://github.com/Embedfire-ucos/ebf_uos_tutorial
- gitee 仓库地址：https://gitee.com/Embedfire-ucos/ebf_uos_tutorial

点击右侧链可在线阅读本项目文档：《[野火 ®]uCOS-III 内核实现与应用开发实战指南—基于 STM32 》

本书主要面向有 MCU 开发经验，希望从零开始学习 uCOS 开发的嵌入式软件工程师及在校学生。
关于野火

开源共享，共同进步

野火在发布第一块 STM32 开发板之初，就喊出 开源共享，共同进步 的口号，把代码和文档教程都免费提供给用户下载，而我们也一直把这个理念贯穿至今。

目前我们的产品已经包括STM32、i.MX RT 系列、GD32V、FPGA、Linux、emXGUI、操作系统、网络、下载器等分支，覆盖电子工程应用领域的各种常用技术，其中教学类产品的代码和文档一直保持着开源的姿态发布到网络上，为电子工程师排忧解难，让嵌入式没有难用的技术是我们最大的愿望。

联系方式

- 官网：http://www.embedfire.com
- 论坛：http://www.firebbs.cn
- github 主页：https://github.com/Embdefire
- gitee 主页：https://gitee.com/wildfireteam
- 淘宝：https://yehuosm.tmall.com
- 邮箱：embedfire@embedfire.com
- 电话：0769-33894118
第 1 章 前言

1.1 如何学习本书

本书与市面上的任何一本 μC/OS-III 相关的书都不一样，它们要么是翻译官方的参考手册，要么是讲如何使用 μC/OS-III，要么是讲 μC/OS-III 的源码，而本书是教你怎么从 0 开始把 μC/OS-III 写出来，既讲了源码实现，也讲了 API 如何使用。当你拿到本书开始学习时一定会惊讶，原来 RTOS 的学习并没有那么复杂，反而是那么有趣，原来自己也可以写 OS，成就感立马爆棚。

全书内容循序渐进，不断迭代，前一章都是后一章的基础，必须从头开始阅读，不能进行跳跃式的阅读。在学习的时候务必做到两点：一是不能一味地看书，要把代码和书本结合起来学习，一边看书，一边调试代码。如何调试代码呢？即单步执行每一条程序，看程序的执行流程和执行的效果与自己所想的是否一致；二是在每学完一章之后，必须将配套的例程重写一遍（切记不要复制，哪怕是一个分号，但可以照书录入），做到举一反三，确保真正理解。在自己写的时候肯定会错漏百出，这个时候要认真纠错，好好调试，这是你提高编程能力的最好机会。记住，编写程序不是一气呵成的，而是要一步一步地调试。

1.2 本书的编写风格

本书以 μC/OS-III 官方源码为蓝本，抽丝剥茧，不断迭代，教你如何从 0 开始把 μC/OS-III 写出来。书中涉及的数据类型、变量名称、函数名称，文件名称、文件存放的位置都完全按照 μC/OS-III 官方的方式来实现。学完这本书之后，可以无缝地切换到原版的 μC/OS-III 中使用。要注意的是，在实现的过程中，某些函数中会去掉一些形参和冗余的代码，只保留核心的功能，但这也并不会影响学习。注意，本书并不是教你怎么写一个自己的 OS，而是教你 μC/OS-III 是怎么写出来的，着重讲解原理实现，当你看完这本书之后，再学习其他 RTOS 将会事半功倍。

论坛：https://www.firebbs.cn
d天猫：https://yehuosm.tmall.com
1.3 本书的技术论坛

如果在学习过程中遇到问题，可以到野火电子论坛www.firebbs.cn 发帖交流，开源共享，共同进步。

鉴于水平有限，本书难免有错漏之处，热心的读者也可把勘误发送到论坛上以便改进。祝你学习愉快，μC/OS-III 的世界，野火与你同行。

1.4 本书的参考资料

1. μC/OS-III 官方源代码
2. μC/OS-III 中文翻译（电子版）
3. 嵌入式操作系统 μC/OS-II(第二版)（电子版）
4. 嵌入式实时操作系统 μC/OS-II 原理及应用任哲编著（电子版）
5. CM3 权威指南 CNR2（电子版）
6. STM32F10xxx Cortex-M3 programming manual（电子版）

1.5 本书的配套硬件

本书支持野火 STM32 开发板全套系列，具体型号见下表，具体图片见图野火 MINI 开发板、图野火指南者开发板、图野火霸道开发板、图野火霸天虎开发板 和图野火挑战者开发板。学习的时候如果配套这些硬件平台做实验，学习必会达到事半功倍的效果，可以省去中间硬件不一样时移植遇到的各种问题。

表格: 野火 STM32 开发板型号汇总

论坛: [https://www.firebbs.cn/](https://www.firebbs.cn/)  天猫: [https://yehuosm.tmall.com](https://yehuosm.tmall.com)
<table>
<thead>
<tr>
<th>型号</th>
<th>内核</th>
<th>引脚</th>
<th>RAM</th>
<th>ROM</th>
</tr>
</thead>
<tbody>
<tr>
<td>MINI</td>
<td>Cortex-M3</td>
<td>64</td>
<td>48KB</td>
<td>256KB</td>
</tr>
<tr>
<td>指南者</td>
<td>Cortex-M3</td>
<td>100</td>
<td>64KB</td>
<td>512KB</td>
</tr>
<tr>
<td>霸道</td>
<td>Cortex-M3</td>
<td>144</td>
<td>64KB</td>
<td>512KB</td>
</tr>
<tr>
<td>霸天虎</td>
<td>Cortex-M4</td>
<td>144</td>
<td>192KB</td>
<td>1MB</td>
</tr>
<tr>
<td>挑战者 F429</td>
<td>Cortex-M4</td>
<td>176</td>
<td>256KB</td>
<td>1MB</td>
</tr>
<tr>
<td>挑战者 F767</td>
<td>Cortex-M7</td>
<td>176</td>
<td>512KB</td>
<td>1MB</td>
</tr>
<tr>
<td>挑战者 H7</td>
<td>Cortex-M7</td>
<td>176</td>
<td>1MB</td>
<td>2MB</td>
</tr>
</tbody>
</table>
图野火【MINI】STM32F103RCT6 开发板
图野火【指南者】STM32F103VET6 开发板

图野火【霸道】STM32F103ZET6 开发板
图野火【霸天虎】STM32F407ZGT6 开发板
图野火【挑战者】STM32F429IGT6 开发板
TODO 和悬赏任务

本页记录待改进的事项，以后会在此页发布任务清单，完成任务可获得奖励，敬请期待！如果您想参与到项目，可按 如何参与野火开源项目 的说明操作。

- 格式问题
  - 代码改成使用 rst 的格式
  - 图片的引用需要修改
  - 把图片改成居中形式，去掉缩放

- 悬赏任务
若想领取悬赏任务，请发送要参与的任务和预计时间到邮箱：embedfire@embedfire.com。

论坛：https://www.firebbs.cn
天猫：https://yehuosm.tmall.com
第 2 章 引言

2.1 为什么学习 RTOS

当我们进入嵌入式这个领域时，首先接触的往往是单片机编程，单片机编程又支持 51 单片机的使用。这里面所说的嵌入式编程通常都是指裸机编程，即不使用任何RTOS（Real Time Operation System，实时操作系统）的编程。常用的RTOS 有国外的FreeRTOS、μC/OS、RTX 和国内的 Huawei LiteOS，和 RT-Thread 等，其中开源且免费的 FreeRTOS 的市场占有率最高，历史悠久的μC/OS 位居第二。在裸机系统中，所有的程序基本都是自己写的，所有的操作都是在一个无限大的循环里面实现的。现实生活中的很多中小型电子产品中用的都是裸机系统，而且能够满足需求。但是为什么还要学习RTOS 编程，要涉及一个操作系统呢？一是基于项目需求，随着产品功能的增强，产品的系统已经不能完美的解决问题，反而会使开发变得复杂。如果想降低开发的难度，可以考虑引入RTOS 实现多任务管理，这是使用RTOS 的最大优势。二是出于学习的需要，必须学习更高级的技术，实现更复杂的职业规划，为将来能有更好的职业发展做准备，而不是拘泥与裸机编程。作为一个合格的嵌入式软件工程师，学习是永远不能停止的，时刻都要为将来做准备。书到用时方恨少，希望当机会来临时，你不要有这种感觉。

为了帮大家厘清RTOS 编程的思路，本书将简单分析这两种编程方式的区别，我们称之为“为学习RTOS 的命门”，只要掌握这一关键内容，以后的RTOS 学习可以说是易如反掌。在讲解这两种编程方式的区别时，我们主要讲解方法论，不会涉及具体的代码，即主要还是通过伪代码来讲解。

2.2 如何学习RTOS

裸机编程和RTOS 编程的风格有些不一样，而且有很多人说学习RTOS 非常难，这就导致想要学习的人一听到RTOS 编程就会忌惮三分，结果就是“出师未捷身先死”。

那么到底如何学习一个RTOS？最简单的方法就是在别人移植好的系统上先看一看RTOS 中的

论坛：https://www.firebbs.cn

天猫：https://yehuosm.tmall.com
API 使用说明，然后调用这些 API 实现自己想要的功能，完全不用担心底层的移植，这是最简单、快速的人门方法。这种方法有利有弊。如果是做产品，好处是可以快速实现功能，将产品推向市场，赢得先机，弊端是当程序出现问题时，因对 RTOS 不够了解，会导致调试困难。如果想系统地学习 RTOS，那么只看简单地调用 API 是不可取的，我们应该深入学习其中一款 RTOS。

目前市场上的 RTOS，其内核实际方式差异不大，我们只需要深入学习其中一款即可。万变不离其宗，只要掌握了一款 RTOS，以后换到其他型号的 RTOS，使用起来自然也得心应手。那么如何深入地学习一款 RTOS 呢？这里有一个非常有效但也十分难的方法，就是阅读 RTOS 的源码，深入研究内核和每个组件的实现方式。这个过程枯燥且痛苦。但为了能够学到 RTOS 的精华，还是很值得一试的。

市面上虽然有一些讲解相关 RTOS 源码的书，但如果基础知识掌握得不够，且先前没有使用过该款 RTOS，那么只看源码还是会非常枯燥，并且不能从全局掌握整个 RTOS 的构成和实现。

现在，我们采用一种全新的方法来教大家学习一款 RTOS，既不是单纯地介绍其中的 API 如何使用，也不是单纯地拿里面的源码一句句地讲解，而是从 0 开始，层层叠加，不断完善，教大家如何把一个 RTOS 写 0 到 1 写出来，让你在每一个阶段都能享受到成功的喜悦。在这个 RTOS 实现的过程中，只需要具备 C 语言基础即可，然后就是按照本书所教学，最后定有所成。

这个用来教学的 RTOS，我们不会完全自己写一个，不会再重复的造轮子，而是选取年龄最大（26岁）、商业化最成功、安全验证最多的 μC/OS-III 为蓝本，将其抽丝剥茧，层层叠加，从 0 到 1 写出来。在实现的过程中，数据类型、变量名、函数名称、文件类型等都完全按照 μC/OS-III 里面的写法，不会自己再重新命名。这样学完我们这个课程之后，就可以无缝地过渡到 μC/OS-III 的使用。

### 2.3 如何选择 RTOS

选择一个 RTOS 要看下你是学习还是做产品，如果是学习则选择一个年龄最大，商业化最成功，安全验证最多的来学习，而且是深入学习。那么符合前面这几个标准的只有 μC/OS，所以，学一个 RTOS，首选 μC/OS，而且 μC/OS 的资料是最多的。当然，选择其他的 RTOS 来学习也是可以的。学完之后就要用了，如果是产品中使用 μC/OS 就要面临授权的问题，就是给版权费。一听到要给钱，大家肯定是不乐意了，所以开源免费的 FreeRTOS 就受到各个半导体厂商和开发者的青睐。目前，FreeRTOS 是市场占有率最高的 RTOS，非常适合用来做产品。另外，国内的
RT-Thread 也在迅速的崛起，同样是开源免费。
第 3 章　第一部分从 0 到 1 教你写 μC/OS 内核

本部分以 μC/OS-III 为蓝本，抽丝剥茧，不断迭代，教你如何从 0 开始把 μC/OS-III 写出来。这一部分着重讲解 μC/OS-III 实现的过程，当你学完这部分之后，再来重新使用 μC/OS-III 或者其他 RTOS，将会得心应手，不仅知其然，而且知其所以然。在源码实现的过程中，涉及的数据类型、变量名称、函数名称、文件名称以及文件的存放目录都会完全按照 μC/OS-III 的来实现，一些不必要的代码将会被剔除，但并不会影响我们理解整个操作系统功能。本部分几乎每一章都是前一章的基础，环环相扣，逐渐为你揭开 μC/OS-III 的神秘面纱，读起来会有一种豁然开朗的感觉。如果把代码都敲一遍，仿真时得出的效果与书中给出的一样，那从心里油然而生的成就感简直就要爆棚，恨不得一下子把本书读完，真是让人看了还想看，读了还想读。
第 4 章 新建工程—软件仿真

在开始写 RTOS 之前，先新建一个工程，Device 选择 Cortex-M3 内核的处理器，调试方式选择软件仿真，到最后写完整个 RTOS 之后，再把 RTOS 移植到野火 STM32 开发板上。最后的移植其实已经非常简单，只需要换一下启动文件和添加 bsp 驱动即可。

4.1 新建本地工程文件夹

在开始新建工程之前，我们先在本地计算机端新建一个文件夹用于存放工程。文件夹名设置为 RTOS，然后在该文件夹下新建各个文件夹和文件，有关这些文件夹的包含关系和作用如表所示。
<table>
<thead>
<tr>
<th>文件夹名称</th>
<th></th>
<th>文件夹作用</th>
</tr>
</thead>
<tbody>
<tr>
<td>Doc</td>
<td>.</td>
<td>用于存放对整个工程的说明文件，如 readme.txt。通常情况下，我们都要对整个工程实现的功能，如何编译，如何使用等做一个简要的说明。</td>
</tr>
<tr>
<td>Project</td>
<td>.</td>
<td>用于存放新建的工程文件</td>
</tr>
<tr>
<td>Mser</td>
<td>μC/OS-III</td>
<td>Source 用于存放 μC/OS-III 源码，其中的代码是纯软件相关的，与硬件无关</td>
</tr>
<tr>
<td></td>
<td>Ports</td>
<td>用于存放接口文件，即 μC/OS-III 与 CPU 连接的文件，也就是我们通常所说的移植文件。要想 μC/OS-III 在单片机上运行 μC/OS-III，这些移植文件必不可少</td>
</tr>
<tr>
<td>μC-CPU</td>
<td>.</td>
<td>用于存放 μC/OS-III 根据 CPU 总结的通用代码，只与 CPU 相关</td>
</tr>
<tr>
<td>μC-LIB</td>
<td>.</td>
<td>用于存放一些 C 语言函数库</td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>用于存放用户程序，如 app.c，main() 函数就放在 app.c 文件中</td>
</tr>
</tbody>
</table>

论坛: https://www.firebbs.cn/  
天猫: https://yehuosm.tmall.com
4.2 使用 KEIL 新建工程

开发环境我们使用 KEIL5，版本为 5.15，高于版本 5 即可。

4.2.1 New Project

首先打开 KEIL5 软件，新建一个工程，工程文件放在目录 ProjectRVMDK（uv5）下面，名称命名为 YH-μC/OS-III，其中 YH 是野火拼音首字母的缩写，当然你也可以换成其他名称，但是必须是英文，不能是中文，切记。

4.2.2 Select Device For Target

当设置好工程名称并确定之后会弹出 Select Device for Target 对话框，在该对话框中可以选择处理器，这里选择 ARMCM3，具体如图 Select_Device_For_Target 对话框所示。
4.2.3 Manage Run-Time Environment

选择好处理器，单击 OK 按钮后会弹出 Manage Run-Time Environment 选项框。这里我们在 CMSIS 栏选中 CORE 和 Device 栏选中 Startup 这两个文件即可，具体见图 Manage_Run-Time_Environment。
单击 OK，关闭 Manage Run-Time Environment 选项框之后，刚刚我们选择的 CORE 和 Startup 这两个文件就会添加到我们的工程组里面，具体见图 CORE 和 Startup 文件。

论坛：https://www.firebbs.cn/  19  天猫：https://yehuosm.tmall.com
其实这两个文件刚开始都是存放在 KEIL 的安装目录下，当我们配置 Manage Run-Time Environment 选项框之后，软件就会把选中好的文件从 KEIL 的安装目录复制到我们的工程目录：ProjectRTEDeviceARMCM3 下面。其中 startup_ARMCM3.s 是汇编编写的启动文件，system_ARMCM3.c 是 C 语言编写的跟时钟相关的文件。更加具体的可直接阅读这两个文件的源码。只要是 Cortex-M3 内核的单片机，这两个文件都适用。

4.3 在 KEIL 工程里面新建文件组

在工程里面添加 User、μC/OS-III Source、μC/OS-III Ports、μC/CPU、μC/LIB 和 Doc 这几个文件组，用于管理文件，具体见图新添加的文件组。

对于新手，这里有个问题就是如何添加文件组？具体的方法为鼠标右键 Target1，在弹出的选项里面选择 Add Group … 即可，具体见图如何添加组，需要多少个组就鼠标右击多少次 Target1。
4.4 在 KEIL 工程里面添加文件

在工程里面添加好组之后，我们需要把本地工程里面新建好的文件添加到工程里面。具体为把 readme.txt 文件添加到 Doc 组，app.c 添加到 User 组，至于 OS 相关的文件我们还没有编写，那么 OS 相关的组就暂时为空。具体见图往组里面添加好的文件。

对于新手，这里有个问题就是如何将本地工程里面的文件添加到工程组里里面？具体的方法为鼠标左键双击相应的组，在弹出的文件选择框中找到要添加的文件，默认的文件类型是 C 文件，
如果要添加的是文本或者汇编文件，那么此时将看不到，这个时候就需要把文件类型选择为 All Files，最后单击 Add 按钮即可，具体见图如何往组里面添加文件。

4.4.1 编写 main() 函数

一个工程如果没有 main() 函数是编译不成功的，会出错。因为系统在开始执行的时候先执行启动文件里面的复位程序，复位程序里面会调用 C 库函数 __main，__main 的作用是初始化好系统变量，如全局变量，只读的，可读可写的等等。__main 最后会调用 __rtentry，再由 __rtentry 调用 main() 函数，从而由汇编跳入到 C 的世界，这里面的 main() 函数就需要我们手动编写，如果没有编写 main() 函数，就会出现 main() 函数没有定义的错误，具体见图没定义 main 函数的错误。

论坛：https://www.firebbs.cn/  天猫：https://yehuosm.tmall.com
main() 函数我们写在 app.c 文件里面，因为是刚刚新建工程，所以 main() 函数暂时为空，具体见代码清单：

**列表 1: 代码清单：main 函数**

```c
int main(void)
{
    for (; ;)
    {
        /* 嘿事不干 */
    }
}
```

### 4.5 调试配置

#### 4.5.1 设置软件仿真

最后，我们再配置下调试相关的配置即可。为了方便，我们全部代码都用软件仿真，即不需要开发板也不需要仿真器，只需要一个 KEIL 软件即可，有关软件仿真的配置具体见图软件仿真的配置。

论坛：https://www.firebbs.cn/             天猫：https://yehuosm.tmall.com
4.5.2 修改时钟大小

在时钟相关文件 system_ARCMC3.c 的开头，有一段代码定义了系统时钟的大小为 25M，具体见代码清单: 时钟相关宏定义。在软件仿真时，确保时间的准确性，代码里面的系统时钟跟软件仿真的时钟必须一致，所以 Options for Target->Target 的时钟应该由默认的 12M 改成 25M，具体见图软件仿真时钟配置。

### 列表 2: 代码清单: 时钟相关宏定义

1. `#define __HSI         ( 8000000UL)`
2. `#define __XTAL       ( 5000000UL)`

(下页继续)
4.5.3 添加头文件路径

在 C/C++ 选项卡里面指定工程头文件的路径，不然编译会出错，头文件路径的具体指定方法见图指定头文件的路径。
至此，一个完整的基于 Cortex-M 内核的软件仿真的工程就建立完毕。
第 5 章  裸机系统与多任务系统

在真正开始动手写 RTOS 之前，我们先来讲解下单片机编程中的裸机系统和多任务系统的区别。

5.1 裸机系统

裸机系统通常分成轮询系统和前后台系统，有关这两者的具体实现方式请看下面的讲解。

5.1.1 轮询系统

轮询系统即是在裸机编程的时候，先初始化好相关的硬件，然后让主程序在一个死循环里面不断循环，顺序地做各种事情，大概的伪代码具体见代码清单: 轮询系统伪代码。轮询系统是一种非常简单的软件结构，通常只适用于那些只需要顺序执行代码且不需要外部事件来驱动的就能完成的事情。在代码清单: 轮询系统伪代码 中，如果只是实现 LED 翻转，串口输出，液晶显示等这些操作，那么使用轮询系统将会非常完美。但是，如果加入了按键操作等需要检测外部信号的事件，用来模拟紧急报警，那么整个系统的实时响应能力就不会那么好了。假设 DoSomething3 是按键扫描，当外部按键被按下，相当于一个警报，这个时候，需要立马响应，并做紧急处理，而这这个时候程序刚好执行到 DoSomething1，要命的是 DoSomething1 需要执行的时间比较久，久到按键释放之后都没有执行完毕，那么当执行到 DoSomething3 的时候就会丢失掉一次事件。足见，轮询系统只适合顺序执行的功能代码，当有外部事件驱动时，实时性就会降低。

列表 1: 代码清单: 轮询系统伪代码

```c
int main(void)
{
    /* 硬件相关初始化 */
    HardWareInit();
```

(下页继续)
/* 无限循环 */
for (;;) {
    /* 处理事情 1 */
    DoSomething1();

    /* 处理事情 2 */
    DoSomething2();

    /* 处理事情 3 */
    DoSomething3();
}

5.1.2 前后台系统

相比轮询系统，前后台系统是在轮询系统的基础上加入了中断。外部事件的响应在中断里面完成，事件的处理还是回到轮询系统中完成，中断在这里我们称为前台，main 函数里面的无限循环我们称为后台，大概的伪代码见代码清单: 前后台系统伪代码。

列表 2: 代码清单: 前后台系统伪代码

```c
int flag1 = 0;
int flag2 = 0;
int flag3 = 0;

int main(void)
{
    /* 硬件相关初始化 */
    HardWareInit();

    /* 无限循环 */
```
for (;;)
{
    if (flag1)
    {
        /* 处理事情 1 */
        DoSomething1();
    }

    if (flag2)
    {
        /* 处理事情 2 */
        DoSomething2();
    }

    if (flag3)
    {
        /* 处理事情 3 */
        DoSomething3();
    }
}

void ISR1(void)
{
    /* 置位标志位 */
    flag1 = 1;
    /* 如果事件处理时间很短，则在中断里面处理
    如果事件处理时间比较长，在回到前台处理 */
    DoSomething1();
}

void ISR2(void)
{  
    /* 置位标志位 */  
    flag2 = 1;

    /* 如果事件处理时间很短，则在中断里面处理  
    如果事件处理时间比较长，在回到前台处理 */  
    DoSomething2();
}

void ISR3(void)
{
    /* 置位标志位 */  
    flag3 = 1;

    /* 如果事件处理时间很短，则在中断里面处理  
    如果事件处理时间比较长，在回到前台处理 */  
    DoSomething3();
}

在顺序执行后台程序的时候，如果有中断来临，那么中断会打断后台程序的正常执行流，转而去执行中断服务程序，在中断服务程序里面标记事件，如果事件要处理的事情很简短，则可在中断服务程序里面处理，如果事件要处理的事情比较多，则返回到后台程序里面处理。虽然事件的响应和处理是分开了，但是事件的处理还是在后台里面顺序执行的，相比轮询系统，前后台系统确保了事件不会丢失，再加上中断具有可嵌套的功能，这可以大大的提高程序的实时响应能力。在大多数的中小型项目中，前后台系统运用的好，堪称有操作系统的效果。
5.2 多任务系统

相比前后台系统，多任务系统的事件响应也是在中断中完成的，但是事件的处理是在任务中完成的。在多任务系统中，任务跟中断一样，也具有优先级，优先级高的任务会被优先执行。当一个紧急的事件在中断被标记之后，如果事件对应的任务的优先级足够高，就会立马得到响应。相比前后台系统，多任务系统的实时性又被提高了。多任务系统大概的伪代码具体见代码清单：多任务系统伪代码。

列表 3: 代码清单: 多任务系统伪代码

```c
int flag1 = 0;
int flag2 = 0;
int flag3 = 0;

int main(void)
{
    /* 硬件相关初始化 */
    HardWareInit();

    /* OS 初始化 */
    RTOSInit();

    /* OS 启动，开始多任务调度，不再返回 */
    RTOSStart();
}

void ISR1(void)
{
    /* 置位标志位 */
    flag1 = 1;
}
```

(下页继续)
void ISR2(void)
{
    /* 置位标志位 */
    flag2 = 2;
}

void ISR3(void)
{
    /* 置位标志位 */
    flag3 = 1;
}

void DoSomething1(void)
{
    /* 无限循环，不能返回 */
    for (;;)
    {
        /* 任务实体 */
        if (flag1)
        {
            
        }
    }
}

void DoSomething2(void)
{
    /* 无限循环，不能返回 */
    for (;;)
    {
        /* 任务实体 */
        if (flag2)
        {
        }
    }
}
```c
voidDoSomething3(void)
{
    /* 无限循环，不能返回 */
    for (;;)
    {
        /* 任务实体 */
        if (flag3)
        {
        }
    }
}
```

相比前后台系统中后台顺序执行的程序主体，在多任务系统中，根据程序的功能，我们把这个程序主体分割成一个个独立的，无限循环且不能返回的小程序，这个小程序我们称之为任务。每个任务都是独立的，互不干扰的，且具备自身的优先级，它由操作系统调度管理。加入操作系统后，我们在编程的时候不需要精心地去设计程序的执行流，不用担心每个功能模块之间是否存在干扰。加入操作系统，我们的编程反而变得简单了。整个系统随之带来的额外开销就是操作系统占据的那一丁点的FLASH和RAM。现如今，单片机的FLASH和RAM是越来越大，完全足以抵挡RTOS那点开销。

无论是裸机系统中的单任务系统、前后台系统和多任务系统，我们不能一锤子的敲定孰优孰劣，它们是不同时代的产物，在各自的领域都还有相当大的应用价值，只有合适才是最好。有关这三者的软件模型区别具体见下表。

表：单任务、多任务和多任务系统软件模型区别
<table>
<thead>
<tr>
<th>模型</th>
<th>事件响应</th>
<th>事件处理</th>
<th>特点</th>
</tr>
</thead>
<tbody>
<tr>
<td>轮询系统</td>
<td>主程序</td>
<td>主程序</td>
<td>轮询响应事件，轮询处理事件</td>
</tr>
<tr>
<td>前后台系统</td>
<td>中断</td>
<td>主程序</td>
<td>实时响应事件，轮询处理事件</td>
</tr>
<tr>
<td>多任务系统</td>
<td>中断</td>
<td>任务</td>
<td>实时响应事件，实时处理事件</td>
</tr>
</tbody>
</table>
第 6 章 任务的定义与任务切换的实现

6.1 本章目标

本章是我们真正从 0 到 1 写 RTOS 的第一章，属于基础中的基础，必须要学会创建任务，重点掌握任务是如何切换的。因为任务的切换是由汇编代码来完成的，所以代码看起来比较难懂，但是我会尽力把代码讲得透彻。如果本章内容学不会，后面的内容根本无从下手。

在这章中，我们会创建两个任务，并让这两个任务不断地切换。任务的主体都是让一个变量按照一定的频率翻转，通过 KEIL 的软件仿真功能，在逻辑分析仪中观察变量的波形变化，最终的波形图具体见任务轮流切换波形图。

其实，图任务轮流切换波形图的波形图的效果，并不是真正的多任务系统中任务切换的效果图，这个效果其实可以完全由裸机代码来实现，具体见代码清单: 任务-1。

列表 1: 代码清单: 任务-1 裸机系统中两个变量轮流翻转

```c
/* flag 必须定义成全局变量才能添加到逻辑分析仪里面观察波形
   * 在逻辑分析仪中要设置以 bit 的模式才能看到波形，不能用默认的模拟量 */

uint32_t flag1;
uint32_t flag2;
```

（下页继续）
在多任务系统中，两个任务不断切换的效果图应该像多任务系统任务切换波形图所示那样，即两个变量的波形是完全一样的，就好像 CPU 在同时干两件事一样，这才是多任务的意义。虽然两者的波形图一样，但是，代码的实现方式是完全不一样的，原来的顺序执行变成了任务的主动切换，这是根本区别。这章只是开始，我们先掌握好任务是如何切换，在后面章节中，我们会陆续的完善功能代码，加入系统调度，实现真正的多任务。千里之行，始于本章节，不要急。
6.2 什么是任务

在裸机系统中，系统的主体就是 main 函数里面顺序执行的无限循环，这个无限循环里面 CPU 按照顺序完成各种事情。在多任务系统中，我们根据功能的不同，把整个系统分割成一个个独立的且无法返回的函数，这个函数我们称为任务。任务的大概形式具体见代码清单: 任务-2。
列表 2: 代码清单: 任务-2 多任务系统中任务的形式

```c
void task_entry (void *parg)
{
    /* 任务主体，无限循环且不能返回 */
    for (;;)
    {
        /* 任务主体代码 */
    }
}
```

### 6.3 创建任务

#### 6.3.1 定义任务栈

我们先回想下，在一个裸机系统中，如果有全局变量，有子函数调用，有中断发生。那么系统在运行的时候，全局变量放在哪里，子函数调用时，局部变量放在哪里，中断发生时，函数返回地址发哪里。如果只是单纯的裸机编程，它们放哪里我们不用管，但是如果要写一个RTOS，这些种种环境参数，我们必须弄清楚他们是如何存储的。在裸机系统中，他们统统放在一个叫栈的地方，栈是单片机 RAM 里面一段连续的内存空间，栈的大小由启动文件里面的代码配置，具体见代码清单: 任务-3，最后由 C 库函数 _main 进行初始化。它们在 RAM 空间里面的大概分布具体见。

列表 3: 代码清单: 任务-3 裸机系统中的栈分配

```assembly
Stack_Size       EQU  0x00000400
|
| AREA   STACK, NOINIT, READWRITE, ALIGN=3 |
Stack_Mem       SPACE  Stack_Size
__initial_sp
```

论坛: [https://www.firebbs.cn/](https://www.firebbs.cn/)  猫猫: [https://yehuosm.tmall.com](https://yehuosm.tmall.com)
但是，在多任务系统中，每个任务都是独立的，互不干扰的，所以要为每个任务都分配独立的栈空间，这个栈空间通常是一个预先定义好的全局数组。这些一个个的任务栈也是存在于 RAM 中，能够使用的最大的栈也是由代码清单: 任务-3 中的 Stack_Size 决定。只是多任务系统中任务的栈就是在统一的一个栈空间里面分配好一个个独立的房间，每个任务只能使用各自的房间，而裸机系统中需要使用栈的时候则可以天马行空，随便在栈里面找个空闲的空间使用，大概的区别具体见。

本章我们要实现两个变量按照一定的频率轮流的翻转，需要两个任务来实现，那么就需要定义两个任务栈，具体见代码清单: 任务-4。在多任务系统中，有多少个任务就需要定义多少个任务栈。

### 列表 4: 代码清单: 任务-4 定义任务栈

```c
#define TASK1_STK_SIZE 128   (1)
#define TASK2_STK_SIZE 128

static CPU_STK Task1Stk[TASK1_STK_SIZE]; (2)
static CPU_STK Task2Stk[TASK2_STK_SIZE];
```

- **代码清单: 任务-4** (1) 任务栈的大小由宏定义控制，在 μC/OS-III 中，空闲任务的栈最小应该大于 128，那么我们这里的任务的栈也暂且配置为 128。

- **代码清单: 任务-4** (2) 任务栈其实就是预先定义好的全局数据，数据类型为 CPU_STK。在 μC/OS-III 中，凡是涉及数据类型的地方，μC/OS-II 都会将标准的 C 数据类型用 typedef 重新取一个类型名，命名方式则采用见名之义的方式命名且统统大写。凡是与 CPU 类型相关的数据类型则统一在 cpu.h 中定义，与 OS 相关的数据类型则在 os_type.h 定义。CPU_STK 就是与 CPU 相关的数据类型，则在 cpu.h 中定义，具体见代码清单: 任务-5。cpu.h 首次使用则需要自行在 μC-CPU 文件夹中新建并添加到工程的 μC/CPU 这个组中。代码清单: 任务-5 中除了 CPU_STK 外，其他数据类型重定义是本章后面内容需要使用到，这里统一贴出来，后面将不再赘述。

### 列表 5: 代码清单: 任务-5 cpu.h 中的数据类型

```c
#ifndef CPU_H
#define CPU_H

#endif
```

(下页继续)
6.3.2 定义任务函数

任务是一个独立的函数，函数主体无限循环且不能返回。本章我们定义的两个任务具体见代码清单：任务-6。

列表 6: 代码清单: 任务-6 任务函数

```c
/* flag 必须定义成全局变量才能添加到逻辑分析仪里面观察波形 */
/* 在逻辑分析仪中要设置以 bit 的模式才能看到波形，不能用默认的模拟量 */
uint32_t flag1; (1)
uint32_t flag2;

/* 任务 1 */
void Task1( void *p_arg ) (2)
```


```c
{   for ( ;; ) {     flag1 = 1;     delay( 100 );     flag1 = 0;     delay( 100 );   }
}

/* 任务 2 */
void Task2( void *p_arg )(3) {

for ( ;; ) {     flag2 = 1;     delay( 100 );     flag2 = 0;     delay( 100 );   }
}
```

- **代码清单: 任务-6 (1):** 需要在 KEIL 的逻辑分析仪中观察波形的变量需要定义成全局变量，且要以 bit 的模式观察，不能使用默认的模拟量。
- **代码清单: 任务-6 (2) 和 (3):** 正如我们所说的那样，任务是一个独立的、无限循环且不能返回的函数。

### 6.3.3 定义任务控制块 TCB

在裸机系统中，程序的主体是 CPU 按顺序执行的。而在多任务系统中，任务的执行是由系统调度的。系统为了顺利的调度任务，为每个任务都额外定义了一个任务控制块 TCB (Task ControlBlock)，这个任务控制块相当于任务的身份证，里面存有任务的所有信息，比如任务的栈，任务名称，任务的形参等。有了这个任务控制块之后，以后系统对任务的全部操作都可以通过这
个 TCB 来实现。TCB 是一个新的数据类型，在 os.h（os.h 第一次使用需要自行在文件夹 μC/OS-IIISource 中新建并添加到工程的 μC/OS-III Source 组）这个头文件中声明，有关 TCB 具体的声明见代码清单: 任务-7，使用它可以为每个任务都定义一个 TCB 实体。

列表 7: 代码清单: 任务-7 任务控制块 TCB 类型声明

```c
/* 任务控制块重定义 */
typedef struct os_tcb OS_TCB; (1)

/* 任务控制块数据类型声明 */
struct os_tcb { (2)
    CPU_STK *StkPtr;
    CPU_STK_SIZE StkSize;
};
```

• 代码清单: 任务-7 (1): 在 μC/OS-III 中，所有的数据类型都会重新取一个名字且用大写字母表示。

• 代码清单: 任务-7 (2): 目前 TCB 里面的成员还比较少，只有栈指针和栈大小。其中为了以后操作方便，我们把栈指针作为 TCB 的第一个成员。

在本章实验中，我们在 app.c 文件中为两个任务定义的 TCB 具体见代码清单: 任务-8。

列表 8: 代码清单: 任务-8 任务 TCB 定义

```c
static OS_TCB Task1TCB;
static OS_TCB Task2TCB;
```

6.3.4 实现任务创建函数

任务的栈，任务的函数实体，任务的 TCB 最终需要联系起来才能由系统进行统一调度。那么这个联系的工作就由任务创建函数 OSTaskCreate 来实现，该函数在 os_task.c（os_task.c 第一次使用需要自行在文件夹 μC/OS-IIISource 中新建并添加到工程的 μC/OS-III Source 组）中定义，所有跟任务相关的函数都在这个文件定义。OSTaskCreate 函数的实现具体见代码清单: 任务-9。

论坛: https://www.firebbs.cn/  天猫: https://yehuosm.tmall.com
列表 9: 代码清单: 任务-9OSTaskCreate 函数

```c
void OSTaskCreate (OS_TCB *p_tcb, (1)
                 OS_TASK_PTR p_task, (2)
                        void *p_arg, (3)
                CPU_STK *p_stk_base, (4)
                CPU_STK_SIZE stk_size, (5)
                OS_ERR *p_err) (6)
{
    CPU_STK *p_sp;

    p_sp = OSTaskStkInit (p_task, (7)
                         p_arg,
                         p_stk_base,
                         stk_size);
    p_tcb->StkPtr = p_sp; (8)
    p_tcb->StkSize = stk_size; (9)

    *p_err = OS_ERR_NONE; (10)
}
```

代码清单: 任务-9: OSTaskCreate 函数遵循 μC/OS-III 中的函数命名规则，以大小的 OS 开头，表示这是一个外部函数，可以由用户调用。以 OS_ 开头的函数表示内部函数，只能由 μC/OS-III 内部使用。紧接着是文件名，表示该函数放在哪个文件，最后是函数功能名称。

- 代码清单: 任务-9 (1): p_tcb 是任务控制块指针。
- 代码清单: 任务-9 (2): p_task 是任务函数名，类型为 OS_TASK_PTR，原型声明在 os.h 中，具体见代码清单: 任务-10。
列表 10: 代码清单: 任务-10 OS_TASK_PTR 原型声明

typedef void (*OS_TASK_PTR)(void *p_arg);

• 代码清单: 任务-9 (3): p_arg 是任务形参，用于传递任务参数。
• 代码清单: 任务-9 (4): p_stk_base 用于指向任务栈的起始地址。
• 代码清单: 任务-9 (5): stk_size 表示任务栈的大小。
• 代码清单: 任务-9 (6): p_err 用于存错误码，μC/OS-III 中为函数的返回值预先定义了很多错误码，通过这些错误码我们可以知道函数是因为什么出错。为了方便，我们现在把 μC/OS-III 中所有的错误号都贴出来，错误码是枚举类型的数据，在 os.h 中定义，具体见 代码清单: 任务-11。

列表 11: 代码清单: 任务-11 错误码枚举定义

typedef enum os_err {
    OS_ERR_NONE = 0u,
    OS_ERR_A = 10000u,
    OS_ERR_ACCEPT_ISR = 10001u,
    OS_ERR_B = 11000u,
    OS_ERR_C = 12000u,
    OS_ERR_CREATE_ISR = 12001u,
    OS_ERR_X = 33000u,
    OS_ERR_Y = 34000u,
    OS_ERR_YIELD_ISR = 34001u,
}
代码清单: 任务-9 (7): OSTaskStkInit() 是任务栈初始化函数。当任务第一次运行的时候，加载到 CPU 寄存器的参数就放在任务栈里面，在任务创建的时候，预先初始化好栈。OSTaskStkInit() 函数在 os_cpu_c.c (os_cpu_c.c 第一次使用需要自行在文件夹 μC-CPU 中新建并添加到工程的 μC/CPU 组) 中定义，具体见代码清单: 任务-12。

列表 12: 代码清单: 任务-12OSTaskStkInit() 函数

```c
CPU_STK *OSTaskStkInit (OS_TASK_PTR p_task,(1)
    void *p_arg, (2)
    CPU_STK *p_stk_base, (3)
    CPU_STK_SIZE stk_size)(4)
{
    CPU_STK *p_stk;

    p_stk = &p_stk_base[stk_size];(5)
    /* 异常发生时自动保存的寄存器 */(6)
    *--p_stk = (CPU_STK)0x01000000u; /* xPSR 的 bit24 必须置 1 */
    *--p_stk = (CPU_STK)p_task; /* R15(PC) 任务的入口地址 */
    *--p_stk = (CPU_STK)0x14141414u; /* R14 (LR) */
    *--p_stk = (CPU_STK)0x12121212u; /* R12 */
    *--p_stk = (CPU_STK)0x03030303u; /* R3 */
    *--p_stk = (CPU_STK)0x02020202u; /* R2 */
    *--p_stk = (CPU_STK)0x01010101u; /* R1 */
    *--p_stk = (CPU_STK)p_arg; /* R0 : 任务形参 */
    /* 异常发生时需手动保存的寄存器 */(7)
    *--p_stk = (CPU_STK)0x11111111u; /* R11 */
    *--p_stk = (CPU_STK)0x10101010u; /* R10 */
    *--p_stk = (CPU_STK)0x09090909u; /* R9 */
```
22 *--p_stk = (CPU_STK)0x08080808u; /* R8 */
23 *--p_stk = (CPU_STK)0x07070707u; /* R7 */
24 *--p_stk = (CPU_STK)0x06060606u; /* R6 */
25 *--p_stk = (CPU_STK)0x05050505u; /* R5 */
26 *--p_stk = (CPU_STK)0x04040404u; /* R4 */
27
28    return (p_stk); (8)
29 }

- 代码清单: 任务-12 (1): p_task 是任务名，指示着任务的入口地址，在任务切换的时候，需要加载到 R15，即 PC 寄存器，这样 CPU 就可以找到要运行的任务。

- 代码清单: 任务-12 (2): p_arg 是任务的形参，用于传递参数，在任务切换的时候，需要加载到寄存器 R0。R0 寄存器通常用来传递参数。

- 代码清单: 任务-12 (3): p_stk_base 表示任务栈的起始地址。

- 代码清单: 任务-12 (4): stk_size 表示任务栈的大小，数据类型为 CPU_STK_SIZE，在 Cortex-M3 内核的处理器中等于 4 个字节，即一个字。

- 代码清单: 任务-12 (5): 获取任务栈的栈顶地址，ARMCM3 处理器的栈是由高地址向低地址生长的。所以初始化栈之前，要获取到栈顶地址，然后栈地址逐一递减即可。

- 代码清单: 任务-12 (6): 任务第一次运行的时候，加载到 CPU 寄存器的环境参数我们要预先初始化好。初始化的顺序固定，首先是异常发生时自动保存的 8 个寄存器，即 xPSR、R15、R14、R12、R3、R2、R1 和 R0。其中 xPSR 寄存器的位 24 必须是 1，R15PC 指针必须存的是任务的入口地址，R0 必须是任务形参，剩下的 R14、R12、R3、R2 和 R1 为了调试方便，填入与寄存器号相对应的 16 进制数。

- 代码清单: 任务-12 (7): 剩下的是 8 个需要手动加载到 CPU 寄存器的参数，为了调试方便填入与寄存器号相对应的 16 进制数。

- 代码清单: 任务-12 (8): 返回栈指针 p_stk，这个时候 p_stk 指向剩余栈的栈顶。

- 代码清单: 任务-9 (8): 将剩余栈的栈顶指针 p_sp 保存到任务控制块 TCB 的第一个成员 StkPtr 中。

论坛: https://www.firebbs.cn/
天猫: https://yehuosm.tmall.com
• 代码清单: 任务-9 (9): 将任务栈的大小保存到任务控制块 TCB 的成员 StkSize 中。

• 代码清单: 任务-9 (10): 函数执行到这里表示没有错误，即 OS_ERR_NONE。

任务创建好之后，我们需要把任务添加到一个叫就绪列表的数组里面，表示任务已经就绪，系统随时可以调度。将任务添加到就绪列表的代码具体见代码清单: 任务-13。

列表 13: 代码清单: 任务-13 将任务添加到就绪列表

```c
/* 将任务加入到就绪列表 */
OSRdyList[0].HeadPtr = &Task1TCB;(1)
OSRdyList[1].HeadPtr = &Task2TCB;(2)
```

• 代码清单: 任务-13 (1) 和 (2): 把任务 TCB 指针放到 OSRDYList 数组里面。OSRDYList 是一个类型为 OS_RDY_LIST 的全局变量，在 os.h 中定义，具体见代码清单: 任务-14。

列表 14: 代码清单: 任务-14 全局变量 OSRDYList 定义

```c
OS_EXT OS_RDY_LIST OSRdyList[OS_CFG_PRIO_MAX];
```

• 代码清单: 任务-14 (3): OS_CFG_PRIO_MAX 是一个定义，表示这个系统支持多少个优先级（刚开始暂时不支持多个优先级，往后章节会支持），目前这里仅用来表示这个就绪列表可以存多少个任务的 TCB 指针。具体的宏在 os_cfg.h（os_cfg.h 第一次使用需要自行在文件夹 μC/OS-IIISource 中新建并添加到工程的 μC/OS-III Source 组）中定义，具体见代码清单: 任务-15。

论坛: https://www.firebbs.cn/ 天猫: https://yehuosm.tmall.com
列表 15: 代码清单: 任务-15 OS_CFG_PRIO_MAX 宏定义

```c
#ifndef OS_CFG_H
#define OS_CFG_H

/* 支持最大的优先级 */
#define OS_CFG_PRIO_MAX 32u

#endif /* OS_CFG_H */
```

- 代码清单: 任务-14 (2): OS_RDY_LIST 是就绪列表的数据类型，在 os.h 中声明，具体见代码清单: 任务-16。

列表 16: 代码清单: 任务-16 OS_RDY_LIST 数据类型声明

```c
typedef struct os_rdy_list OS_RDY_LIST; (1)

struct os_rdy_list {
    OS_TCB *HeadPtr;
    OS_TCB *TailPtr;
};
```

- 代码清单: 任务-16 (1): μC/OS-III 中中会为每个数据类型重新取一个大写的名字。

- 代码清单: 任务-16 (2): OS_RDY_LIST 里面目前暂时只有两个 TCB 类型的指针，一个是头指针，一个是尾指针。本章实验只用到头指针，用来指向任务的 TCB。只有当后面讲到同一个优先级支持多个任务的时候才需要使用头尾指针来将 TCB 串成一个双向链表。

代码清单: 任务-14 (1): OS_EXT 是一个在 os.h 中定义的宏，具体见代码清单: 任务-17。

列表 17: 代码清单: 任务-17OS_EXT 宏定义

```c
#ifdef OS_GLOBALS
#define OS_EXT
#else
#define OS_EXT extern
#endif
```

代码清单: 任务-17：该段代码的意思是，如果没有定义 OS_GLOBALS 这个宏，那么 OS_EXT 就为空，否则就为 extern。

在 μC/OS-III 中，需要使用很多全局变量，这些全局变量都在 os.h 这个头文件中定义，但是 os.h 会被包含进很多的文件中，那么编译的时候，os.h 里面定义的全局变量就会出现重复定义的情况。而我们要的只是 os.h 里面定义的全局变量只定义一次，其他包含 os.h 头文件的时候只是声明。有人说，那我可以加 extern，那你告诉我怎么加？

通常我们的做法都是在 C 文件里面定义全局变量，然后在头文件里面加 extern 声明，哪里需要使用就在哪里加 extern 声明。但是 μC/OS-III 中，文件非常多，这种方法可行，但不现实。所以就有了现在在 os.h 头文件中定义全局变量，然后在 os.h 文件的开头加上代码清单: 任务-17 的宏定义的方法。但是到了这里还没成功，μC/OS-III 再另外新建了一个 os_var.c（os_var.c 第一次使用需要在文件下 μC/OS-IIISource 中新建并添加到工程的 μC/OS-IIISource 组）的文件，在里面包含 os.h，且只在这个文件里面定义 OS_GLOBALS 这个宏，具体见代码清单: 任务-18。

列表 18: 代码清单: 任务-18 os_var.c 文件内容

```c
#define OS_GLOBALS
#include "os.h"
```

经过这样处理之后，在编译整个工程的时候，只有 var.c 里面的 os.h 的 OS_EXT 才会被替换为空，即变量的定义。其他包含 os.h 的文件因为没有定义 OS_GLOBALS 这个宏，则 OS_EXT 会被替换成 extern，即变成了变量的声明。这样就实现了在头文件中定义变量。

在 μC/OS-III 中，将任务添加到就绪列表其实是在 OSTaskCreate() 函数中完成的。每当任务创建好

论坛: https://www.firebbs.cn/  
天猫: https://yehuosm.tmall.com
就把任务添加到就绪列表，表示任务已经就绪。只是目前这里的就绪列表的实现还是比较简单，不支持优先级，不支持双向链表，只是简单的将任务控制块放到就绪列表的数组里面。后面会有独立的章节来讲解就绪列表，等我们完善就绪列表之后，再把这部分的操作放在 OSTaskCreate() 函数里面。

### 6.4 OS 系统初始化

OS 系统初始化一般是在硬件初始化完成后之后来做的，主要做的工作就是初始化 μC/OS-III 中定义的全局变量。OSInit() 函数在文件 os_core.c（os_core.c 第一次使用需要自行在文件夹 μC/OS-IIISource 中新建并添加到工程的 μC/OS-IIISource 组）中定义，具体实现见代码清单: 任务-19。

#### 列表 19: 代码清单: 任务-19OSInit() 函数

```c
void OSInit (OS_ERR *p_err)
{
    OSRunning = OS_STATE_OS_STOPPED; (1)

    OSTCBCurPtr = (OS_TCB *) 0; (2)
    OSTCBHighRdyPtr = (OS_TCB *) 0; (3)

    OS_RdyListInit(); (4)

    *p_err = OS_ERR_NONE; (5)
}
```

- **代码清单: 任务-19** (1): 系统用一个全局变量 OSRunning 来指示系统的运行状态，刚开始系统初始化的时候，默认为停止状态，即 OS_STATE_OS_STOPPED。

- **代码清单: 任务-19** (2): 全局变量 OSTCBCurPtr 是系统用于指向当前正在运行的任务的 TCB 指针，在任务切换的时候用得到。

- **代码清单: 任务-19** (3): 全局变量 OSTCBHighRdyPtr 用于指向就绪任务中优先级最高的任务的 TCB，任务切换的时候用得到。本章暂时不支持优先级，则用于指向第一个运行的
任务的 TCB。

- **代码清单: 任务-19 (4):** OS_RdyListInit() 用于初始化全局变量 OSRdyList[], 即初始化就绪列表。OS_RdyListInit() 在 os_core.c 文件中定义，具体实现见代码清单: 任务-20

列表 20: 代码清单: 任务-20 OS_RdyListInit() 函数

```c
void OS_RdyListInit(void)
{
    OS_PRIO i;
    OS_RDY_LIST *p_rdy_list;

    for ( i=0u; i<OS_CFG_PRIO_MAX; i++ )
    {
        p_rdy_list = &OSRdyList[i];
        p_rdy_list->HeadPtr = (OS_TCB *)0;
        p_rdy_list->TailPtr = (OS_TCB *)0;
    }
}
```

- **代码清单: 任务-19 (5):** 代码运行到这里表示没有错误，即 OS_ERR_NONE。

代码清单: 任务-19 中的全局变量 OSTCBCurPtr 和 OSTCBHighRdyPtr 均在 os.h 中定义，具体见代码清单: 任务-21。OS_STATE_OS_STOPPED 这个表示系统运行状态的宏也在 os.h 中定义，具体见代码清单: 任务-22。

列表 21: 代码清单: 任务-21 OSInit() 函数中出现的全局变量的定义

```c
OS_EXT OS_TCB *OSTCBCurPtr;
OS_EXT OS_TCB *OSTCBHighRdyPtr;
OS_EXT OS_RDY_LIST OSRdyList[OS_CFG_PRIO_MAX];
OS_EXT OS_STATE OSRunning;
```
6.5 启动系统

任务创建好，系统初始化完毕之后，就可以开始启动系统了。系统启动函数 OSStart() 在 os_core.c 中定义，具体实现见代码清单: 任务-23。

列表 23: 代码清单: 任务-23OSStart() 函数

```
void OSStart (OS_ERR *p_err)
{
    if ( OSRunning == OS_STATE_OS_STOPPED ) {(1)
        /* 手动配置任务 1 先运行 */
        OSTCBHighRdyPtr = OSRdyList[0].HeadPtr; (2)

        /* 启动任务切换，不会返回 */
        OSStartHighRdy(); (3)

        /* 不会运行到这里，运行到这里表示发生了致命的错误 */
        *p_err = OS_ERR_FATAL_RETURN;
    }
    else
    {
        *p_err = OS_STATE_OS_RUNNING;
    }
}
```

• 代码清单: 任务-23 (1): 系统是第一次启动的话，if 肯定为真，则继续往下运行。
代码清单: 任务-23 (2)：OSTCBHghRdyPtr 指向第一个要运行的任务的 TCB。因为暂时不支持优先级，所以系统启动时先手动指定第一个要运行的任务。

代码清单: 任务-23 (3)：OSStartHighRdy() 用于启动任务切换，即配置 PendSV 的优先级为最低，然后触发 PendSV 异常，在 PendSV 异常服务函数中进行任务切换。该函数不再返回，在文件 os_cpu_a.s（os_cpu_a.s 第一次使用需要自行在文件夹 μC/OS-IIIPorts 中新建并添加到工程的 μC/OS-III Ports 组）中定义，由汇编语言编写，具体实现见代码清单: 任务-24。os_cpu_a.s 文件中涉及的 ARM 汇编指令的用法具体见表常用的 ARM 汇编指令讲解。
<table>
<thead>
<tr>
<th>指令名称作用</th>
<th>作用</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQU</td>
<td>给数字常量取一个符号名，相当于C语言中的define</td>
</tr>
<tr>
<td>AREA</td>
<td>汇编一个新的代码段或者数据段</td>
</tr>
<tr>
<td>SPACE</td>
<td>分配内存空间</td>
</tr>
<tr>
<td>PRESERVE8</td>
<td>当前文件栈需按照8字节对齐</td>
</tr>
<tr>
<td>EXPORT</td>
<td>声明一个标号具有全局属性，可被外部的文件使用</td>
</tr>
<tr>
<td>DCD</td>
<td>以字为单位分配内存，要求4字节对齐，并要求初始化这些内存</td>
</tr>
<tr>
<td>PROC</td>
<td>定义子程序，与ENDP成对使用，表示子程序结束</td>
</tr>
<tr>
<td>WEAK</td>
<td>弱定义，如果外部文件声明了一个标号，则优先使用外部文件定义的标号，如果外部文件没有定义也不出错。要注意的是：这个不是ARM的指令，是编译器的，这里放在一起只是为了方便。</td>
</tr>
<tr>
<td>IMPORT</td>
<td>声明标号来自外部文件，跟C语言中的EXTERN关键字类似</td>
</tr>
<tr>
<td>B</td>
<td>跳转到一个标号</td>
</tr>
<tr>
<td>ALIGN</td>
<td>编译器对指令或者数据的存放地址进行对齐，一般需要跟一个立即数，缺省表示4字节对齐。要注意的是：这个不是ARM的指令，是编译器的，这里放在一起只是为了方便。</td>
</tr>
<tr>
<td>END</td>
<td>到达文件的末尾，文件结束</td>
</tr>
<tr>
<td>IF,ELSE,ENDIF</td>
<td>汇编条件分支语句，跟C语言的if else类似</td>
</tr>
</tbody>
</table>
列表 24: 代码清单: 任务-24OSStartHighRdy() 函数

```assembly
;***************************************************************************
; 开始第一次上下文切换
; 1、配置 PendSV 异常的优先级为最低
; 2、在开始第一次上下文切换之前，设置 psp=0
; 3、触发 PendSV 异常，开始上下文切换
;***************************************************************************
OSStartHighRdy
LDR  R0, =NVIC_SYSPRI14  ; 设置 PendSV 异常优先级为最低(1)
LDR  R1, =NVIC_PENDSV_PRI
STRB R1, [R0]

MOV   R0, #0            ; 设置 psp 的值为 0，开始第一次上下文切换 (2)
MSR   PSP, R0

LDR  R0, =NVIC_INT_CTRL  ; 触发 PendSV 异常(3)
LDR  R1, =NVIC_PENDSVSET
STR   R1, [R0]

CPSIE I              ; 启用总中断，NMI 和 HardFault 除外(4)

OSStartHang
B   OSStartHang        ; 程序应永远不会运行到这里
```

代码清单: 任务-24 中涉及的 NVIC_INT_CTRL、NVIC_SYSPRI14、NVIC_PENDSV_PRI 和 NVIC_PENDSVSET 这四个常量在 os_cpu_a.s 的开头定义，具体见代码清单: 任务-25，有关这四个常量的含义看代码注释即可。
列表 25: 代码清单: 任务-25 NVIC_INT_CTRL、
NVIC_SYSPRI14、NVIC_PENDSV_PRI 和
NVIC_PENDSVSET 常量定义

```plaintext
; ******************************************************************************
; 常量
; ******************************************************************************

; 有些内核外设寄存器定义的参考官方文档: STM32F10xxx Cortex-M3 programming manual
; 系统控制块外设 SCB 地址范围: 0xE000ED00-0xE000ED3F

NVIC_INT_CTRL EQU 0xE000ED04 ; 中断控制及状态寄存器 SCB_ICSR。
NVIC_SYSPRI14 EQU 0xE000ED22 ; 系统优先级寄存器 SCB_SHPR3:

; bit16~23
NVIC_PENDSV_PRI EQU 0xFF ; PendSV 优先级的值 (最低)。
NVIC_PENDSVSET EQU 0x10000000 ; 触发 PendSV 异常的值 Bit28: PENDSVSET
```

- **代码清单: 任务-24 (1):** 配置 PendSV 的优先级为 0xFF，即最低。在 μC/OS-III 中，上下文切换是在 PendSV 异常服务程序中执行的，配置 PendSV 的优先级为最低，从而消灭了在中断服务程序中执行上下文切换的可能。

- **代码清单: 任务-24 (2):** 设置 PSP 的值为 0，开始第一个任务切换。在任务中，使用的栈指针都是 PSP，如果判断出 PSP 为 0，则表示第一次任务切换。

- **代码清单: 任务-24 (3):** 触发 PendSV 异常，如果中断启用且有编写 PendSV 异常服务函数的话，则内核会响应 PendSV 异常，去执行 PendSV 异常服务函数。

- **代码清单: 任务-24 (4):** 开中断，因为有些用户在 main() 函数开始会先关掉中断，等全部初始化完成后，在启动 OS 的时候才开中断。为了快速地开关中断，CM3 专门设置了一条 CPS 指令，有 4 种用法，具体见代码清单: 任务-26。
列表 26: 代码清单: 任务-26 CPS 指令用法

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CPSID I : PRIMASK=1 ; 关中断</td>
</tr>
<tr>
<td>2</td>
<td>CPSIE I : PRIMASK=0 ; 开中断</td>
</tr>
<tr>
<td>3</td>
<td>CPSID F ; FAULTMASK=1 ; 关异常</td>
</tr>
<tr>
<td>4</td>
<td>CPSIE F ; FAULTMASK=0 ; 开异常</td>
</tr>
</tbody>
</table>

代码清单: 任务-26 中 PRIMASK 和 FAULTMAST 是 CM3 里面三个中断屏蔽寄存器中的两个，还有一个是 BASEPRI，有关这三个寄存器的详细用法见表 CM3 中断屏蔽寄存器组描述。

<table>
<thead>
<tr>
<th>名字</th>
<th>功能描述</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRIMASK</td>
<td>这是个只有单一比特的寄存器。在它被置 1 后，就关掉所有可屏蔽的异常，只剩下 NMI 和硬 FAULT 可以响应。它的缺省值是 0，表示没有关中断。</td>
</tr>
<tr>
<td>FAULTMASK</td>
<td>这是个只有 1 个位的寄存器。当它置 1 时，只有 NMI 才能响应，所有其他的异常，甚至是硬 FAULT，也通通闭嘴。它的缺省值也是 0，表示没有关异常。</td>
</tr>
<tr>
<td>BASEPRI</td>
<td>这个寄存器最多有 9 位（由表达优先级的位数决定）。它定义了被屏蔽优先级的阈值。当它被设成某个值后，所有优先级号大于等于此值的中断都被关（优先级号越大，优先级越低）。但若被设成 0，则不关闭任何中断，0 也是缺省值。</td>
</tr>
</tbody>
</table>

论坛：https://www.firebbs.cn/  
天猫：https://yehuosm.tmall.com
### 6.6 任务切换

当调用 OSStartHighRdy() 函数，触发 PendSV 异常后，就需要编写 PendSV 异常服务函数，然后在里面进行任务切换。PendSV 异常服务函数具体见代码清单: 任务-27。PendSV 异常服务函数名称必须与启动文件里面向量表中 PendSV 的向量名一致，如果不一致则内核是响应不了用户编写的PendSV 异常服务函数的，只响应启动文件里面默认的 PendSV 异常服务函数。启动文件里面为每个异常都编写好默认的异常服务函数，函数体都是一个死循环，当你发现代码跳转到这些启动文件里面默认的异常服务函数的时候，就要检查下异常函数名称是否写错了，没有跟向量表里面的一致。PendSV_Handler 函数里面涉及的 ARM 汇编指令的讲解具体见表 PendSV_Handler 函数中涉及的 ARM 汇编指令讲解。

列表 27: 代码清单: 任务-27PendSV 异常服务函数

```c
;***************************************************************
PendSVHandler 异常

PendSV_Handler
; 关中断，NMI 和 HardFault 除外，防止上下文切换被中断
CPSID   I(1)

; 将 psp 的值加载到 R0
MRS     R0, PSP(2)

; 判断 R0，如果值为 0 则跳转到 OS_CPU_PendSVHandler_nosave
; 进行第一次任务切换的时候，R0 肯定为 0
CBZ     R0, OS_CPU_PendSVHandler_nosave (3)

;---------------------------------------------一、保存上下文---------------------------------------------
; 任务的切换，即把下一个要运行的任务的栈内容加载到 CPU 寄存器中
;---------------------------------------------
; 在进入 PendSV 异常的时候，当前 CPU 的 xPSR, PC (任务入口地址)，
; R14, R12, R3, R2, R1, R0 会自动存储到当前任务栈，
```
; 同时递减 PSP 的值，随便通过代码：MRS R0, PSP 把 PSP 的值传给 R0

; 手动存储 CPU 寄存器 R4-R11 的值到当前任务的栈
STMDB R0!, {R4-R11}(15)

; 加载 OSTCBCurPtr 指针的地址到 R1，这里 LDR 属于伪指令
LDR R1, = OSTCBCurPtr(16)

; 加载 OSTCBCurPtr 指针到 R1，这里 LDR 属于 ARM 指令
LDR R1, [R1](17)

; 存储 R0 的值到 OSTCBCurPtr->OSTCBStkPtr，这个时候 R0 存的是任务空闲栈的栈顶
STR R0, [R1](18)

; -------------------------------二、切换下文-------------------------------

; 实现 OSTCBCurPtr = OSTCBHighRdyPtr
; 把下一个要运行的任务的栈内容加载到 CPU 寄存器中

OS_CPU_PendSVHandler_nosave (4)

; 加载 OSTCBCurPtr 指针的地址到 R0，这里 LDR 属于伪指令
LDR R0, = OSTCBCurPtr(5)

; 加载 OSTCBHighRdyPtr 指针的地址到 R1，这里 LDR 属于伪指令
LDR R1, = OSTCBHighRdyPtr(6)

; 加载 OSTCBHighRdyPtr 指针到 R2，这里 LDR 属于 ARM 指令
LDR R2, [R1](7)

; 存储 OSTCBHighRdyPtr 到 OSTCBCurPtr
STR R2, [R0](8)

; 加载 OSTCBHighRdyPtr 到 R0
LDR R0, [R2](9)

; 加载需要手动保存的信息到 CPU 寄存器 R4-R11
LDMIA R0!, {R4-R11}(10)

(下页继续)
; 更新 PSP 的值，这个时候 PSP 指向下下一个要执行的任务的栈底
; (这个栈底已经加上刚刚手动加载到 CPU 寄存器 R4-R11 的偏移)
MSR      PSP, R0(11)

; 确保异常返回使用的栈指针是 PSP，即 LR 寄存器的位 2 要为 1
ORR      LR, LR, #0x04 (12)

; 开中断
CPSIE    I (13)

; 异常返回，这个时候任务栈中的剩下内容将会自动加载到 xPSR，
; PC (任务入口地址)，R14, R12, R3, R2, R1, R0 (任务的形参)
; 同时 PSP 的值也将更新，即指向任务栈的栈顶。
; 在 STM32 中，栈是由高地址向低地址生长的。
BX        LR (14)

代码清单: 任务-27 PendSV 异常服务中主要完成两个工作，一是保存上文，即保存当前正在运行的任务的环境参数；二是切换下文，即把下一个需要运行的任务的环境参数从任务栈中加载到 CPU 寄存器，从而实现任务的切换。接下来具体讲解下代码清单: 任务-27 每句代码的含义。

代码清单: 任务-27 PendSV 异常服务中用到了 OSTCBCurPtr 和 OSTCBHighRdyPtr 这两个全局变量，这两个全局变量在 os.h 中定义，要想在汇编文件 os_cpu_a.s 中使用，必须将这两个全局变量导入到 os_cpu_a.s 中，具体如何导入见代码清单: 任务-28。

列表 28: 代码清单: 任务-28 导入 OSTCBCurPtr 和
OSTCBHighRdyPtr 到 os_cpu_a.s

;**************************************************************************************
; 全局变量 & 函数
;**************************************************************************************
IMPORT OSTCBCurPtr ; 外部文件引人的参考(1)
IMPORT OSTCBHighRdyPtr

EXPORT OSStartHighRdy ; 该文件定义的函数(2)

EXPORT PendSV_Handler

• 代码清单: 任务-28 (1): 使用 IMPORT 关键字将 os.h 中的 OSTCBCurPtr 和 OSTCBHighRdyPtr 两个全局变量导入到该汇编文件, 从而该汇编文件可以使用这两个变量。如果是函数也可以使用 IMPORT 导入的方法。

• 代码清单: 任务-28 (2): 使用 EXPORT 关键字导出该汇编文件里面的 OSStartHighRdy 和 PendSV_Handler 这两个函数，让外部文件可见。除了使用 EXPORT 导出外，还要在某个 C 的头文件里面声明下这两个函数（在 µC/OS-III 中是在 os_cpu.h 中声明），这样才可以在 C 文件里里面调用这两个函数。

• 代码清单: 任务-27 (1): 关中断，NMI 和 HardFault 除外，防止上下文切换被中断。在上下文切换完毕之后，会重新开中断。

• 代码清单: 任务-27 (2): 将 PSP 的值加载到 R0 寄存器。MRS 是 ARM 32 位数据加载指令，功能是加载特殊功能寄存器的值到通用寄存器。

• 代码清单: 任务-27 (3): 判断 R0，如果值为 0 则跳转到 OS_CPU_PendSVHandler_nosave。进行第一次任务切换的时候，PSP 在 OSStartHighRdy 初始化为 0，所以这个时候 R0 肯定为 0，则跳转到 OS_CPU_PendSVHandler_nosave。CBZ 是 ARM16 位转移指令，用于比较，结果为 0 则跳转。

• 代码清单: 任务-27 (4): 当第一次任务切换的时候，会跳转到这里运行。当执行过一次任务切换之后，则顺序执行到这里。这个标号以后的内容属于下文切换。

• 代码清单: 任务-27 (5): 加载 OSTCBCurPtr 指针的地址到 R0。在 ARM 汇编中，操作变量都属于间接操作，即要先获取到这个变量的地址。这里 LDR 属于伪指令，不是 ARM 指令。举例：LDR Rd, = label，如果 label 是立即数，那 Rd 等于立即数，如果 label 是一个标识符，比如指针，那存到 Rd 的就是 label 这个标识符的地址。

• 代码清单: 任务-27 (6): 加载 OSTCBHighRdyPtr 指针的地址到 R1，这里 LDR 也属于伪指令。

论坛: https://www.firebbs.cn/  天猫: https://yehuosm.tmall.com
[野火]uCOS-III 内核实现与应用开发实战指南

- 代码清单：任务-27 (7)：加载 OSTCBHighRdyPtr 指针到 R2，这里 LDR 属于 ARM 指令。

- 代码清单：任务-27 (8)：存储 OSTCBHighRdyPtr 到 OSTCBCurPtr，实现下一个要运行的任务的 TCB 存储到 OSTCBCurPtr。

- 代码清单：任务-27 (9)：加载 OSTCBHighRdyPtr 到 R0。TCB 中第一个成员是栈指针 StkPtr，所以这个时候 R0 等于 StkPtr，后续操作任务栈都是通过操作 R0 来实现，不需要操作 StkPtr。

- 代码清单：任务-27 (10)：将任务栈中需要手动加载的内容加载到 CPU 寄存器 R4-R11，同时会递增 R0，让 R0 指向空闲栈的栈顶。LDMIA 中的 I 是 increase 的缩写，A 是 after 的缩小，R0 后面的感叹号 “!” 表示会自动调节 R0 里面存的指针。当任务被创建的时候，任务的栈会被初始化，初始化的流程是：先让栈指针 StkPtr 指向栈顶，然后从栈顶开始依次存储异常退出时会自动加载到 CPU 寄存器的值和需要手动加载到 CPU 寄存器的值，具体代码实现见代码清单：任务-12OSTaskStkInit() 函数，栈空间的分布情况具体见任务创建成功后栈空间的分布图。当把需要手动加载到 CPU 的栈内容加载完毕之后，栈空间的分布图和栈指针指向具体见图手动加载栈内容到 CPU 寄存器后的栈空间分布图，注意这个时候 StkPtr 不变，变的是 R0。
<table>
<thead>
<tr>
<th>高地址 TaskStk[TASK_STK_SIZE-1]</th>
<th>xPSR的位24，必须置1</th>
</tr>
</thead>
<tbody>
<tr>
<td>R15(PC)任务的入口地址</td>
<td></td>
</tr>
<tr>
<td>R14 (LR)</td>
<td></td>
</tr>
<tr>
<td>R12</td>
<td></td>
</tr>
<tr>
<td>R3</td>
<td></td>
</tr>
<tr>
<td>R2</td>
<td></td>
</tr>
<tr>
<td>R1</td>
<td></td>
</tr>
<tr>
<td>R0: 任务参数</td>
<td></td>
</tr>
<tr>
<td>R11</td>
<td></td>
</tr>
<tr>
<td>R10</td>
<td></td>
</tr>
<tr>
<td>R9</td>
<td></td>
</tr>
<tr>
<td>R8</td>
<td></td>
</tr>
<tr>
<td>R7</td>
<td></td>
</tr>
<tr>
<td>R6</td>
<td></td>
</tr>
<tr>
<td>R5</td>
<td></td>
</tr>
<tr>
<td>R4</td>
<td></td>
</tr>
<tr>
<td>栈指针 StkPtr</td>
<td></td>
</tr>
<tr>
<td>高地址 TaskStk[0]</td>
<td></td>
</tr>
</tbody>
</table>

自动加载到CPU寄存器

需手动加载到CPU寄存器

栈指针 StkPtr

空闲堆栈

论坛：https://www.firebbs.cn/  
天猫：https://yehuosm.tmall.com
【野火】uCOS-III 内核实现与应用开发实战指南

<table>
<thead>
<tr>
<th>高地址 TaskStk[TASK_STK_SIZE-1]</th>
<th>xPSR的位24，必须置1</th>
</tr>
</thead>
<tbody>
<tr>
<td>R15(PC)任务的入口地址</td>
<td>R14 (LR)</td>
</tr>
<tr>
<td>R12</td>
<td>R13</td>
</tr>
<tr>
<td>R3</td>
<td>R2</td>
</tr>
<tr>
<td>R1</td>
<td>R0: 任务形参</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>空闲堆栈</td>
</tr>
<tr>
<td></td>
<td>栈指针 StkPtr</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>低地址 TaskStk[0]</td>
<td>...</td>
</tr>
</tbody>
</table>

- **代码清单: 任务-27** (11): 更新 PSP 的值，这个时候 PSP 与图 3-4 中 R0 的指向一致。
- **代码清单: 任务-27** (12): 设置 LR 寄存器的位 2 为 1，确保异常退出时使用的栈指针是 PSP。当异常退出后，就切换到就绪任务中优先级最高的任务继续运行。
- **代码清单: 任务-27** (13): 开中断。上下文切换已经完成了四分之三，剩下的就是异常退出时自动保存的部分。
- **代码清单: 任务-27** (14): 异常返回，这个时候任务栈中的剩下内容将会自动加载到 xPSR, PC (任务入口地址), R14, R12, R3, R2, R1, R0 (任务的形参) 这些寄存器。同时 PSP 的值也将更新，即指向任务栈的栈顶。这样就切换到了新的任务。这个时候栈空间的分布具体见图刚切换完成即将运行的任务的栈空间分布和栈指针指向。
### 代码清单：任务-27 (15): 手动存储 CPU 寄存器 R4-R11 的值到当前任务的栈。当异常发生，进入 PendSV 异常服务函数的时候，当前 CPU 寄存器 xPSR, PC (任务入口地址), R14, R12, R3, R2, R1, R0 会自动存储到当前的任务栈，同时递减 PSP 的值。这个时候当前任务的栈空间分布具体见图进入 PendSV 异常时当前任务的栈空间分布。当执行 STMDB R0!, [R4-R11] 代码后，当前任务的栈空间分布图具体见当前任务执行完上文保存时的栈空间分布。
<table>
<thead>
<tr>
<th>高地址 TaskStk[TASK_STK_SIZE-1]</th>
<th>xPSR的位24，必须置1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R15(PC)任务的入口地址</td>
</tr>
<tr>
<td></td>
<td>R14 (LR)</td>
</tr>
<tr>
<td></td>
<td>R12</td>
</tr>
<tr>
<td></td>
<td>R3</td>
</tr>
<tr>
<td></td>
<td>R2</td>
</tr>
<tr>
<td></td>
<td>R1</td>
</tr>
<tr>
<td></td>
<td>R0:任务形参</td>
</tr>
<tr>
<td></td>
<td>PSP</td>
</tr>
<tr>
<td></td>
<td>......</td>
</tr>
<tr>
<td></td>
<td>空闲堆栈</td>
</tr>
<tr>
<td>低地址</td>
<td>TaskStk[0]</td>
</tr>
<tr>
<td></td>
<td>......</td>
</tr>
</tbody>
</table>
### [野火]uCOS-III 内核实现与应用开发实战指南

<table>
<thead>
<tr>
<th>高地址 TaskStk[TASK_STK_SIZE-1]</th>
<th>xPSR的位24，必须置1</th>
</tr>
</thead>
<tbody>
<tr>
<td>R15(PC)任务的入口地址</td>
<td>R14 (LR)</td>
</tr>
<tr>
<td>R12</td>
<td>R1</td>
</tr>
<tr>
<td>R3</td>
<td>R2</td>
</tr>
<tr>
<td>R1</td>
<td>R0: 任务形参</td>
</tr>
<tr>
<td>PSP</td>
<td></td>
</tr>
<tr>
<td>......</td>
<td></td>
</tr>
<tr>
<td>空闲堆栈</td>
<td></td>
</tr>
<tr>
<td>低地址 TaskStk[0]</td>
<td></td>
</tr>
</tbody>
</table>

- **代码清单: 任务-27 (16)**：加载 OSTCBCurPtr 指针的地址到 R1，这里 LDR 属于伪指令。
- **代码清单: 任务-27 (17)**：加载 OSTCBCurPtr 指针到 R1，这里 LDR 属于 ARM 指令。
- **代码清单: 任务-27 (18)**：存储 R0 的值到 OSTCBCurPtr->OSTCBSstkPtr，这个时候 R0 存的是任务空闲栈的栈顶。到了这里，上文的保存就总算完成。这个时候当前任务的栈空间分布和栈指针指向具体见图当前任务执行完上文保存时的栈空间分布和 StkPtr 指向。

论坛: [https://www.firebbs.cn/](https://www.firebbs.cn/)  
天猫: [https://yehuosm.tmall.com](https://yehuosm.tmall.com)
<table>
<thead>
<tr>
<th>高地址 TaskStk[TASK_STK_SIZE-1]</th>
<th>xPSR的位24，必须置1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R15(PC)任务的入口地址</td>
</tr>
<tr>
<td></td>
<td>R14 (LR)</td>
</tr>
<tr>
<td></td>
<td>R12</td>
</tr>
<tr>
<td></td>
<td>R3</td>
</tr>
<tr>
<td></td>
<td>R2</td>
</tr>
<tr>
<td></td>
<td>R1</td>
</tr>
<tr>
<td></td>
<td>R0: 任务参数</td>
</tr>
<tr>
<td></td>
<td>PSP</td>
</tr>
<tr>
<td></td>
<td>R11</td>
</tr>
<tr>
<td></td>
<td>R10</td>
</tr>
<tr>
<td></td>
<td>R9</td>
</tr>
<tr>
<td></td>
<td>R8</td>
</tr>
<tr>
<td></td>
<td>R7</td>
</tr>
<tr>
<td></td>
<td>R6</td>
</tr>
<tr>
<td></td>
<td>R5</td>
</tr>
<tr>
<td></td>
<td>R4</td>
</tr>
<tr>
<td></td>
<td>R0、StkPtr</td>
</tr>
<tr>
<td></td>
<td>......</td>
</tr>
<tr>
<td>低地址</td>
<td>......</td>
</tr>
</tbody>
</table>

图示说明：
- 自动从CPU寄存器存储到栈
- 手动从CPU寄存器存储到栈
- 空闲堆栈

论坛：[https://www.firebbs.cn/](https://www.firebbs.cn/)
天猫：[https://yehuosm.tmall.com](https://yehuosm.tmall.com)
<table>
<thead>
<tr>
<th>指令名称</th>
<th>作用</th>
</tr>
</thead>
<tbody>
<tr>
<td>MRS</td>
<td>加载特殊功能寄存器的值到通用寄存器</td>
</tr>
<tr>
<td>MSR</td>
<td>存储通用寄存器的值到特殊功能寄存器</td>
</tr>
<tr>
<td>CBZ</td>
<td>比较，如果结果为 0 就转移</td>
</tr>
<tr>
<td>CBNZ</td>
<td>比较，如果结果非 0 就转移</td>
</tr>
<tr>
<td>LDR</td>
<td>从存储器中加载字到一个寄存器中</td>
</tr>
<tr>
<td>LDR[伪指令] 加</td>
<td>一个立即数或者一个地址值到一个寄存器。举例：LDR</td>
</tr>
<tr>
<td>LDRH</td>
<td>从存储器中加载半字到一个寄存器中</td>
</tr>
<tr>
<td>LDRB</td>
<td>从存储器中加载字节到一个寄存器中</td>
</tr>
<tr>
<td>STR</td>
<td>把一个寄存器按字存储到存储器中</td>
</tr>
<tr>
<td>STRH</td>
<td>把一个寄存器存器的低半字存储到存储器中</td>
</tr>
<tr>
<td>STRB</td>
<td>把一个寄存器的低字节存储到存储器中</td>
</tr>
<tr>
<td>LDMIA</td>
<td>加载多个字，并且在加载后自增基址寄存器</td>
</tr>
<tr>
<td>STMIA</td>
<td>存储多个字，并且在存储后自增基址寄存器</td>
</tr>
<tr>
<td>ORR</td>
<td>按位或</td>
</tr>
<tr>
<td>BX</td>
<td>直接跳转到由寄存器给定的地址</td>
</tr>
<tr>
<td>BL</td>
<td>跳转到标号对应的地址，并且把跳转前的下条指令地址保存到 LR</td>
</tr>
<tr>
<td>BLX</td>
<td>跳转到由寄存器 REG 给出的地址，并根据 REG 的 LSB 切换处理器状态，还要把转移前的下条指令地址保存到 LR。ARM((\text{LSB}=0)), Thumb((\text{LSB}=1))。CM3 只在 Thumb 中运行，就必须保证 reg 的 LSB=1，否则一个 fault 打过来</td>
</tr>
</tbody>
</table>

论坛：https://www.firebbs.cn/  
天猫：https://yehuosm.tmall.com
6.7 main() 函数

main() 函数在文件 app.c 中编写，其中 app.c 文件中的所有代码具体见代码清单: 任务-29。

列表 29: 代码清单: 任务-29 app.c 文件中的代码

```c
/*
 * @(#) os.h
 * @(#) ARMCM3.h
 */

/* 函数声明
*/

uint32_t flag1;
uint32_t flag2;

/* TCB & STACK & 任务声明
*/
```

论坛: https://www.firebbs.cn/ 70 天猫: https://yehuosm.tmall.com
```c
#define TASK1_STK_SIZE 20
#define TASK2_STK_SIZE 20

static CPU_STK Task1Stk[TASK1_STK_SIZE];
static CPU_STK Task2Stk[TASK2_STK_SIZE];
static OS_TCB Task1TCB;
static OS_TCB Task2TCB;

void Task1( void *p_arg );
void Task2( void *p_arg );

void delay(uint32_t count);
```

```
/*
  * 函数声明
  */

/*
  * main() 函数
  */

/* 注意事项：1. 该工程使用软件仿真，debug 需选择 Ude Simulator
  * 2. 在 Target 选项卡里面把晶振 Xtal (Mhz) 的值改为 25，默认是 12，
  * 改成 25 是为了跟 system_ARMCM3.c 中定义的__SYSTEM_CLOCK 相同，
  * 确保仿真时钟时钟一致
*/
```
int main(void)
{
    OS_ERR err;

    /* 初始化相关的全局变量 */
    OSInit(&err);

    /* 创建任务 */
    OSTaskCreate ((OS_TCB*) &Task1TCB,
                   (OS_TASK_PTR ) Task1,
                   (void *) 0,
                   (CPU_STK*) &Task1Stk[0],
                   (CPU_STK_SIZE) TASK1_STK_SIZE,
                   (OS_ERR *) &err);

    OSTaskCreate ((OS_TCB*) &Task2TCB,
                   (OS_TASK_PTR ) Task2,
                   (void *) 0,
                   (CPU_STK*) &Task2Stk[0],
                   (CPU_STK_SIZE) TASK2_STK_SIZE,
                   (OS_ERR *) &err);

    /* 将任务加入到就绪列表 */
    OSRdyList[0].HeadPtr = &Task1TCB;
    OSRdyList[1].HeadPtr = &Task2TCB;

    /* 启动 OS，将不再返回 */
    OSStart(&err);
}

/*
/*
** 软件延时 */
void delay (uint32_t count)
{
    for (; count!=0; count--);
}

/* 任务 1 */
void Task1 ( void *p_arg )
{
    for ( ;; ) {
        flag1 = 1;
        delay ( 100 );
        flag1 = 0;
        delay ( 100 );

        /* 任务切换，这里是手动切换 */
        OSSched();
    }
}

/* 任务 2 */
void Task2 ( void *p_arg )
{
    for ( ;; ) {
        flag2 = 1;
        delay ( 100 );
        flag2 = 0;
代码清单：任务-29 中的所有代码在本小节之前都有循序渐进的讲解，这里这是融合在一起放在 main() 函数中。其实现在 Task1 和 Task2 并不会真正的自动切换，而是在各自的函数体里面加入了 OSSched() 函数来实现手动切换。OSSched() 函数的实现具体见代码清单：任务-30。

列表 30：代码清单：任务-30 OSSched() 函数

```c
/* 任务切换，实际就是触发 PendSV 异常，然后在 PendSV 异常中进行上下文切换 */
void OSSched (void)
{
    if ( OSTCBCurPtr == OSRdyList[0].HeadPtr )
    {
        OSTCBHighRdyPtr = OSRdyList[1].HeadPtr;
    }
    else
    {
        OSTCBHighRdyPtr = OSRdyList[0].HeadPtr;
    }

    OS_TASK_SW();
}
```

OSSched() 函数的调度算法很简单，即如果当前任务是任务 1，那么下一个任务就是任务 2，如果当前任务是任务 2，那么下一个任务就是任务 1，然后再调用 OS_TASK_SW() 函数触发 PendSV 异常，然后在 PendSV 异常里面实现任务的切换。在往后的章节中，我们将继续完善，加入 SysTick 中断，从而实现系统调度的自动切换。OS_TASK_SW() 函数其实是一个宏定义，具体是往中断及状态控制寄存器 SCB_ICSR 的位 28（PendSV 异常启用位）写入 1，从而触发 PendSV 异常。

论坛：https://www.firebbs.cn/    天猫：https://yehuosm.tmall.com
OS_TASK_SW() 函数在 os_cpu.h 文件中实现，os_cpu.h（os_cpu.h 第一次使用需要自行在文件夹 μC-CPU 中新建并添加到工程的 μC-CPU 组）文件内容具体见代码清单: 任务-31。

列表 31: 代码清单: 任务-31 os_cpu.h 文件代码清单

```c
#ifndef OS_CPU_H
#define OS_CPU_H

/*
 * ******************************************************************************
 *  * 宏定义
 *  ******************************************************************************
 */
#endif

#ifndef NVIC_INT_CTRL
/* 中断控制及状态寄存器 SCB_ICSR */
#define NVIC_INT_CTRL (*((CPU_REG32 *)0xE000ED04)
#endif

#ifndef NVIC_PENDSVSET
/* 触发 PendSV 异常的值 Bit28: PENDSVSET */
#define NVIC_PENDSVSET 0x10000000
#endif

/* 触发 PendSV 异常 */
#define OS_TASK_SW() NVIC_INT_CTRL = NVIC_PENDSVSET
/* 触发 PendSV 异常 */
#define OSIntCtxSw() NVIC_INT_CTRL = NVIC_PENDSVSET

/*
 * ******************************************************************************
 *  * 函数声明
 *  ******************************************************************************
 */
#endif
```

(下页继续)
6.8 实验现象

本章代码讲解完毕，接下来是软件调试仿真，具体过程见

- 图点击 Debug 按钮进入调试界面
- 图点击逻辑分析仪按钮调出逻辑分析仪
- 图将要观察的变量添加到逻辑分析仪
- 图将变量设置为 Bit 模式，默认是 Analog
- 图点击全速运行按钮，即可看到波形，Zoom 栏的 In Out All 可放大和缩小波形
[野火]uCOS-III 内核实现与应用开发实战指南

论坛：https://www.firebbs.cn/
天猫：https://yehuosm.tmall.com
至此，本章讲解完毕。但是，只是把本章的内容看完，然后再仿真看看波形是远远不够的，应该是把任务的栈、TCB、OSTCBCurPtr 和 OSTCBHighRdyPtr 这些变量统统添加到观察窗口，然后单步执行程序，看看这些变量是怎么变化的。特别是任务切换时，CPU 寄存器、任务栈和 PSP 这些是怎么变化的，让机器执行代码的过程在自己的脑子里面过一遍。图软件调试仿真时的 Watch 窗口 就是我在仿真调试时的观察窗口。
第 7 章 任务时间片运行

本章在上一章的基础上，加入 SysTick 中断，在 SysTick 中断服务函数里面进行任务切换，从而实现双任务的时间片运行，即每个任务运行的时间都是一样的。

7.1 SysTick 简介

RTOS 需要一个时基来驱动，系统任务调度的频率等于该时基的频率。通常该时基由一个定时器来提供，也可以从其他周期性的信号源获得。刚好 Cortex-M 内核中有一个系统定时器 SysTick，它内嵌在 NVIC 中，是一个 24 位的递减的计数器，计数器每计数一次的时间为 1/SYSCLK。当重装载数值寄存器的值递减到 0 的时候，系统定时器就产生一次中断，以此循环往复。因为 SysTick 是嵌套在内核中的，所以使得 OS 在 Cortex-M 器件中编写的定时器代码不必修改，使移植工作一下子变得简单很多。所以 SysTick 是最适合给操作系统提供时基，用于维护系统心跳的定时器。有关 SysTick 的寄存器汇总见表 4-1，各个寄存器的用法见表 4-2、表 4-3、表 4-4 和表 4-5。

表 4-1 SysTick 寄存器汇总

<table>
<thead>
<tr>
<th>寄存器名称</th>
<th>寄存器描述</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTRL</td>
<td>SysTick 控制及状态寄存器</td>
</tr>
<tr>
<td>LOAD</td>
<td>SysTick 重装载数值寄存器</td>
</tr>
<tr>
<td>VAL</td>
<td>SysTick 当前数值寄存器</td>
</tr>
</tbody>
</table>

表 4-2 SysTick 控制及状态寄存器
<table>
<thead>
<tr>
<th>位段</th>
<th>名称</th>
<th>类型</th>
<th>复位值</th>
<th>描述</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>COUNTFLAG</td>
<td>R/W</td>
<td>0</td>
<td>如果在上次读取本寄存器后，SysTick 已经计到 0，则该位为 1。</td>
</tr>
<tr>
<td>2</td>
<td>CLKSOURCE</td>
<td>R/W</td>
<td>0</td>
<td>时钟源选择位，0=AHB/8，1=处理器时钟 AHB</td>
</tr>
<tr>
<td>1</td>
<td>TICKINT</td>
<td>R/W</td>
<td>0</td>
<td>1=SysTick 倒数计数到 0 时产生 SysTick 异常请求，0=数到 0 时无动作。也可以通过读取 COUNTFLAG 标志位来确定计数器是否递减到 0</td>
</tr>
<tr>
<td>0</td>
<td>ENABLE</td>
<td>R/W</td>
<td>0</td>
<td>SysTick 定时器的启用位</td>
</tr>
</tbody>
</table>

表 4-3SysTick 重装载数值寄存器

<table>
<thead>
<tr>
<th>位段</th>
<th>名称</th>
<th>类型</th>
<th>复位值</th>
<th>描述</th>
</tr>
</thead>
<tbody>
<tr>
<td>23:0</td>
<td>RELOAD</td>
<td>R/W</td>
<td>0</td>
<td>当倒数计数至零时，将被重装载的值</td>
</tr>
</tbody>
</table>

表 4-4SysTick 当前数值寄存器
### 7.2 初始化 SysTick

使用 SysTick 非常简单，只需一个初始化函数搞定，OS_CPU_SysTickInit 函数在 os_cpu.c 中定义，具体实现见代码清单: 时间片运行-1。在这里，SysTick 初始化函数我们没有使用 μC/OS-III 官方的，我们是自己另外编写了一个。区别是 uC/OS-III 官方的 OS_CPU_SysTickInit 函数里面涉及 SysTick 寄存器都是重新在 cpu.h 中定义，而我们自己编写的则是使用 ARMCM3.h（记得在 os_cpu.c 的开头包含 ARMCM3.h 这个头文件）这个固件库文件里面定义的寄存器，仅此区别而已。

列表 1: 代码清单: 时间片运行-1SysTick 初始化

```c
#if 0 /* 不用 μC/OS-III 自带的 */
void OS_CPU_SysTickInit (CPU_INT32U cnts)
{
    CPU_INT32U prio;

    /* 填写 SysTick 的重载计数值 */
    CPU_REG_NVIC_ST_RELOAD = cnts - 1u;

    /* 设置 SysTick 中断优先级 */
```

（下页继续）
prio = CPU_REG_NVIC_SHPRI3;
prio &= DEF_BIT_FIELD(24, 0);
prio |= DEF_BIT_MASK(OS_CPU_CFG_SYSTICK_PRIO, 24);

CPU_REG_NVIC_SHPRI3 = prio;

/* 启用 SysTick 的时钟源和启动计数器 */
CPU_REG_NVIC_ST_CTRL |= CPU_REG_NVIC_ST_CTRL_CLKSOURCE |
                         CPU_REG_NVIC_ST_CTRL_ENABLE;
/* 启用 SysTick 的定时中断 */
CPU_REG_NVIC_ST_CTRL |= CPU_REG_NVIC_ST_CTRL_TICKINT;
}

#else/* 直接使用头文件 ARMCM3.h 里面现有的寄存器定义和函数来实现 */
void OS_CPU_SysTickInit (CPU_INT32U ms)
{
    /* 设置装载寄存器的值 */
    SysTick->LOAD = ms * SystemCoreClock / 1000 - 1; (1)

    /* 配置中断优先级为最低 */
    NVIC_SetPriority (SysTick_IRQn, (1<<__NVIC_PRIO_BITS) - 1); (2)

    /* 复位当前计数器的值 */
    SysTick->VAL = 0; (3)

    /* 选择时钟源、启用中断、启用计数器 */
    SysTick->CTRL = SysTick_CTRL_CLKSOURCE_Msk | (4)
                   SysTick_CTRL_TICKINT_Msk | (5)
                   SysTick_CTRL_ENABLE_Msk; (6)
}
#endif
• 代码清单: 时间片运行-1 (1): 配置重装载寄存器的值，我们配合函数形参 ms 来配置，如果需要配置为 10ms 产生一次中断，形参设置为 10 即可。

• 代码清单: 时间片运行-1 (2): 配置 SysTick 的优先级，这里配置为 15，即最低。

• 代码清单: 时间片运行-1 (3): 复位当前计数器的值。

• 代码清单: 时间片运行-1 (4): 选择时钟源，这里选择 SystemCoreClock。

• 代码清单: 时间片运行-1 (5): 启用中断。

• 代码清单: 时间片运行-1 (6): 启用计数器开始计数。

### 7.3 编写 SysTick 中断服务函数

SysTick 中断服务函数也是在 os_cpu.c 中定义，具体实现见代码清单: 时间片运行-2。

列表 2: 代码清单: 时间片运行-2 SysTick 中断服务函数

```c
/* SysTick 中断服务函数 */
void SysTick_Handler(void)
{
    OSTimeTick();
}
```

SysTick 中断服务函数很简单，里面仅调用了函数 OSTimeTick()。OSTimeTick() 是与时间相关的函数，在 os_time.c（os_time.c 第一次使用需要自行在文件夹 \uC/OS-IIISource 中新建并添加到工程的 \uC/OS-IIISource 组）文件中定义，具体实现见代码清单: 时间片运行-3。

列表 3: 代码清单: 时间片运行-3 OSTimeTick() 函数

```c
void OSTimeTick (void)
{
    /* 任务调度 */
    OSSched();
}
```
OSTimeTick() 很简单，里面仅调用了函数 OSSched，OSSched 函数暂时没有修改，与上一章一样，具体见代码清单: 时间片运行-4。

列表 4: 代码清单: 时间片运行-4 OSSched 函数

```c
void OSSched (void)
{
    if ( OSTCBCurPtr == OSRdyList[0].HeadPtr )
    {
        OSTCBHighRdyPtr = OSRdyList[1].HeadPtr;
    }
    else
    {
        OSTCBHighRdyPtr = OSRdyList[0].HeadPtr;
    }
    OS_TASK_SW();
}
```

### 7.4 main() 函数

main() 函数与上一章区别不大，仅仅是加入了 SysTick 相关的内容，具体见代码清单: 时间片运行-5。

列表 5: 代码清单: 时间片运行-5 main() 函数和任务代码

```c
int main(void)
{
    OS_ERR err;
```

论坛: [https://www.firebbs.cn/](https://www.firebbs.cn/)  
天猫: [https://yehuosm.tmall.com](https://yehuosm.tmall.com)
/* 关闭中断 */
CPU_IntDis();(1)

/* 配置 SysTick 10ms 中断一次 */
OS_CPU_SysTickInit (10);(2)

/* 初始化相关的全局变量 */
OSInit(&err);

/* 创建任务 */
OSTaskCreate ((OS_TCB*) &Task1TCB,
              (OS_TASK_PTR ) Task1,
              (void *) 0,
              (CPU_STK*) &Task1Stk[0],
              (CPU_STK_SIZE) TASK1_STK_SIZE,
              (OS_ERR *) &err);

OSTaskCreate ((OS_TCB*) &Task2TCB,
              (OS_TASK_PTR ) Task2,
              (void *) 0,
              (CPU_STK*) &Task2Stk[0],
              (CPU_STK_SIZE) TASK2_STK_SIZE,
              (OS_ERR *) &err);

/* 将任务加入到就绪列表 */
OSRdyList[0].HeadPtr = &Task1TCB;
OSRdyList[1].HeadPtr = &Task2TCB;

/* 启动 OS，将不再返回 */
OSStart(&err);
```c
/* 任务 1 */
void Task1 ( void *p_arg )
{
    for ( ;; ) {
        flag1 = 1;
        delay( 100 );
        flag1 = 0;
        delay( 100 );

        /* 任务切换，这里是手动切换 */
        //OSSched();(3)
    }
}

/* 任务 2 */
void Task2 ( void *p_arg )
{
    for ( ;; ) {
        flag2 = 1;
        delay( 100 );
        flag2 = 0;
        delay( 100 );

        /* 任务切换，这里是手动切换 */
        //OSSched();(4)
    }
}
```

论坛：https://www.firebbs.cn/  天猫：https://yehuosm.tmall.com


• 代码清单: 时间片运行-5 (1): 关闭中断。因为在 OS 系统初始化之前我们启用了 SysTick 定时器产生 10ms 的中断，在中断里面触发任务调度，如果一开始我们不关闭中断，就会在 OS 还有启动之前就进入 SysTick 中断，然后发生任务调度，既然 OS 都还没启动，跳调度是不允许发生的，所以应先关闭中断。系统启动后，中断由 OSStart() 函数里面的 OSStartHighRdy() 重新开启。

• 代码清单: 时间片运行-5 (2): 配置 SysTick 为 10ms 中断一次。任务的调度是在 SysTick 的中断服务函数中完成的，中断的频率越高就意味着 OS 的调度越高，系统的负荷就越重，一直在不断的进入中断，执行任务的时间就减小。选择合适的 SysTick 中断频率会提供系统的运行效率，μC/OS-III 官方推荐为 10ms，或者高点也行。

• 代码清单: 时间片运行-5 (3)、(4): 任务调度将不再在各自的任务里面实现，而是放到了 SysTick 中断服务函数中。从而实现每个任务都运行相同的时间片，平等的享有 CPU。

7.5 实验现象

进入软件调试，单击全速运行按钮就可看到实验波形，具体见图实验现象。

从图实验现象 我们可以看到，两个任务轮流的占有 CPU，享有相同的时间片。其实目前的实验
现象与上一章的实验现象还没有本质上的区别，加入 SysTick 只是为了后续章节做准备。上一章两个任务也是轮流的占有 CPU，也是享有相同的时间片，该时间片是任务单次运行的时间。不同的是本章任务的时间片等于 SysTick 定时器的时基，是很多个任务单次运行时间的综合。即在这个时间片里面任务运行了非常多次，如果我们把波形放大，就会发现大波形里面包含了很多小波形，具体见图实验现象 2。
第 8 章 阻塞延时与空闲任务

在上一章节中，任务体内的延时使用的是软件延时，即还是让 CPU 空等来达到延时的效果。使用 RTOS 的很大优势就是榨干 CPU 的性能，永远不能让它闲着，任务如果需要延时也就不能再让 CPU 空等来实现延时的效果。RTOS 中的延时叫阻塞延时，即任务需要延时的时候，任务会放弃 CPU 的使用权，CPU 可以去做其他的事情。当任务延时时间到，重新获取 CPU 使用权，任务继续运行，这样就充分利用了 CPU 的资源，而不是干等着。

当任务需要延时，进入阻塞状态，那 CPU 又去干什么事情了？如果没有其他任务可以运行，RTOS 都会为 CPU 创建一个空闲任务。这个时候 CPU 就运行空闲任务。在 μC/OS-III 中，空闲任务是系统在初始化的时候创建的优先级最低的任务，空闲任务主体很简单，只是对一个全局变量进行计数。鉴于空闲任务的这种特性，在实际应用中，当系统进入空闲任务的时候，可在空闲任务中让单片机进入休眠或者低功耗等操作。

8.1 实现空闲任务

8.1.1 定义空闲任务栈

空闲任务栈在 os_cfg_app.c（os_cfg_app.c 第一次使用需要自行在文件夹 μC/OS-III Source 中新建并添加到工程的 μC/OS-III Source 组）文件中定义，具体见代码清单: 空闲任务-1。

列表 1: 代码清单: 空闲任务-1os_cfg_app.c 文件代码

```c
/*******************************************************************************
 *
****************************************************************************/

CPU_STK    OSCfg_IdleTaskStk[OS_CFG_IDLE_TASK_STK_SIZE];(1)
```

(下页继续)
CPU_STK * const Oscfg_IdleTaskStkBasePtr = (CPU_STK *) &OSCfg_IdleTaskStk[0];

CPU_STK_SIZE const Oscfg_IdleTaskStkSize = (CPU_STK_SIZE) OS_CFG_IDLE_TASK_STK_SIZE;

• 代码清单: 全局变量-1 (1) : 全局变量的数组大小由 OS_CFG_IDLE_TASK_STK_SIZE 定义，大小由 OS_CFG_IDLE_TASK_STK_SIZE 宏控制。OS_CFG_IDLE_TASK_STK_SIZE 在 os_cfg_app.h 头文件中定义，大小为 128，具体见代码清单: 全局变量-2。

列表 2: 代码清单: 全局变量-2 os_cfg_app.h 文件代码

```c
#ifndef OS_CFG_APP_H
#define OS_CFG_APP_H

/*

常量

*/

/* 全局变量大小 */
#define OS_CFG_IDLE_TASK_STK_SIZE 128u
```

(下页继续)
### 8.1.2 定义空闲任务 TCB

任务控制块 TCB 是每一个任务必须的，空闲任务的 TCB 在 os.h 中定义，是一个全局变量，具体见代码清单: 空闲任务-4。

<table>
<thead>
<tr>
<th>行号</th>
<th>代码</th>
<th>描述</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>/* 空闲任务 TCB */</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>OS_EXT OS_TCB OSIdleTaskTCB;</td>
<td>定义空闲任务 TCB</td>
</tr>
</tbody>
</table>

### 8.1.3 定义空闲任务函数

空闲任务正如其名，空闲，任务体里面只是对全局变量 OSIdleTaskCtr ++ 操作，具体实现见代码清单: 空闲任务-5。
列表 5: 代码清单: 空闲任务-5 空闲任务函数

```c
/* 空闲任务 */
void OS_IdleTask (void *p_arg)
{
    p_arg = p_arg;

    /* 空闲任务什么都不做，只对全局变量 OSIdleTaskCtr ++ 操作 */
    for (;;) {
        OSIdleTaskCtr++;
    }
}
```

代码清单: 空闲任务-5 中的全局变量 OSIdleTaskCtr 在 os.h 中定义，具体见代码清单: 空闲任务-6。

列表 6: 代码清单: 空闲任务-6 OSIdleTaskCtr 定义

```c
/* 空闲任务计数变量 */
OS_EXT OS_IDLE_CTR OSIdleTaskCtr;
```

代码清单: 空闲任务-6 中的 OS_IDLE_CTR 是在 os_type.h 中重新定义的数据类型，具体见代码清单: 空闲任务-7。
8.1.4 空闲任务初始化

空闲任务的初始化在 OSInit() 完成，意味着在系统还没有启动之前空闲任务就已经创建好，具体在 os_core.c 定义，具体代码见代码清单: 空闲任务-8。

列表 8: 代码清单: 空闲任务-8 空闲任务初始化函数

```c
void OSInit (OS_ERR *p_err)
{
    /* 配置 OS 初始状态为停止态 */
    OSRunning = OS_STATE_OS_STOPPED;

    /* 初始化两个全局 TCB，这两个 TCB 用于任务切换 */
    OSTCBCurPtr = (OS_TCB *)0;
    OSTCBHighRdyPtr = (OS_TCB *)0;

    /* 初始化就绪列表 */
    OS_RdyListInit();

    /* 初始化空闲任务 */
    OS_IdleTaskInit(p_err);(1)
    if (p_err != OS_ERR_NONE) {
        return;
    }
}
```

(下页继续)
void OS_IdleTaskInit(OS_ERR *p_err)
{
    /* 初始化空闲任务计数器 */
    OSIdleTaskCtr = (OS_IDLE_CTR)0; (2)

    /* 创建空闲任务 */
    OSTaskCreate((OS_TCB *)&OSIdleTaskTCB, (3)
        (OS_TASK_PTR )OS_IdleTask,
        (void *)0,
        (CPU_STK *)OSCfg_IdleTaskStkBasePtr,
        (CPU_STK_SIZE)OSCfg_IdleTaskStkSize,
        (OS_ERR *)p_err );
}

• 代码清单: 空闲任务-8 (1): 空闲任务初始化函数在 OSInit 中调用，在系统还没有启动之前就被创建。

• 代码清单: 空闲任务-8 (2): 初始化空闲任务计数器，我们知道，这个是预先在 os.h 中定义好的全局变量。

• 代码清单: 空闲任务-8 (3): 创建空闲任务，把栈，TCB，任务函数联系在一起。

8.2 实现阻塞延时

阻塞延时的阻塞是指任务调用该延时函数后，任务会被剥夺 CPU 使用权，然后进入阻塞状态，直到延时结束，任务重新获取 CPU 使用权才可以继续运行。在任务阻塞的这段时间，CPU 可以去执行其他的任务，如果其他的任务也在延时状态，那么 CPU 就将运行空闲任务。阻塞延时函数在 os_time.c 中定义，具体代码实现见代码清单: 空闲任务-9。

论坛：https://www.firebbs.cn/  95  天猫：https://yehuosm.tmall.com
列表 9: 代码清单: 空闲任务-9 阻塞延时代码

```c
/* 阻塞延时 */
void OSTimeDly(OS_TICK dly)
{
    /* 设置延时时间 */
    OSTCBCurPtr->TaskDelayTicks = dly; (1)

    /* 进行任务调度 */
    OSSched(); (2)
}
```

- **代码清单: 空闲任务-9 (1)**: TaskDelayTicks 是任务控制块的一个成员，用于记录任务需要延时的时间，单位为 SysTick 的中断周期。比如我们本书当中 SysTick 的中断周期为 10ms，调用 OSTimeDly(2) 则完成 2*10ms 的延时。TaskDelayTicks 的定义具体见代码清单: 空闲任务-10。

列表 10: 代码清单: 空闲任务-10 TaskDelayTicks 定义

```c
struct os_tcb {
    CPU_STK     *StkPtr;
    CPU_STK_SIZE StkSize;

    /* 任务延时周期个数 */
    OS_TICK      TaskDelayTicks;
};
```

- **代码清单: 空闲任务-9 (2)**: 任务调度。这个时候的任务调度与上一章节的不一样，具体见代码清单: 空闲任务-11，其中加粗部分为上一章节的代码，现已用条件编译屏蔽掉。
列表 11: 代码清单: 空闲任务-11 任务调度

```c
void OSSched(void)
{
#if 0 /* 非常简单任务调度: 两个任务轮流执行 */
    if ( OSTCBCurPtr == OSRdyList[0].HeadPtr )
    {
        OSTCBHighRdyPtr = OSRdyList[1].HeadPtr;
    }
    else
    {
        OSTCBHighRdyPtr = OSRdyList[0].HeadPtr;
    }
#endif

/* 如果当前任务是空闲任务，那么就去尝试执行任务 1 或者任务 2，看看他们的延时时间是否结束，如果任务的延时时间均没有到期，那就返回继续执行当前任务 */
    if ( OSTCBCurPtr == &OSIdleTaskTCB ) (1)
    {
        if (OSRdyList[0].HeadPtr->TaskDelayTicks == 0)
        {
            OSTCBHighRdyPtr = OSRdyList[0].HeadPtr;
        }
        else if (OSRdyList[1].HeadPtr->TaskDelayTicks == 0)
        {
            OSTCBHighRdyPtr = OSRdyList[1].HeadPtr;
        }
        else
        {
            /* 任务延时均没有到期则返回，继续执行当前任务 */
            return;
        }
    }
```

论坛: [https://www.firebbs.cn/](https://www.firebbs.cn/)  天猫: [https://yehuosm.tmall.com](https://yehuosm.tmall.com)
```c
else (2)
{
/* 如果是 task1 或者 task2 的话，检查下另外一个任务，
如果另外的任务不在延时中，就切换到该任务
否则，判断下当前任务是否应该进入延时状态，
如果是的话，就切换到空闲任务。否则就不进行任何切换 */
    if (OSTCBCurPtr == OSRdyList[0].HeadPtr)
    {
        if (OSRdyList[1].HeadPtr->TaskDelayTicks == 0)
        {
            OSTCBHighRdyPtr = OSRdyList[1].HeadPtr;
        }
        else if (OSTCBCurPtr->TaskDelayTicks != 0)
        {
            OSTCBHighRdyPtr = &OSIdleTaskTCB;
        }
    }
    else
    {
        /* 返回，不进行切换，因为两个任务都处于延时中 */
        return;
    }
}
else if (OSTCBCurPtr == OSRdyList[1].HeadPtr)
{
    if (OSRdyList[0].HeadPtr->TaskDelayTicks == 0)
    {
        OSTCBHighRdyPtr = OSRdyList[0].HeadPtr;
    }
    else if (OSTCBCurPtr->TaskDelayTicks != 0)
    {
```
OSTCHighRdyPtr = &OSIdleTaskTCB;
}
else
{
    /* 返回，不进行切换，因为两个任务都处于延时中 */
    return;
}
}

/* 任务切换 */
OS_TASK_SW(); (3)

• 代码清单: 空闲任务-11 (1): 如果当前任务是空闲任务，那么就去尝试执行任务 1 或者任务 2，看看他们的延时时间是否结束，如果任务的延时时间均没有到期，那就返回继续执行空闲任务。

• 代码清单: 空闲任务-11 (2): 如果当前任务不是空闲任务则会执行到此，那就看看当前任务是哪个任务。无论是哪个任务，都要检查下另外一个任务是否在延时中，如果没有在延时，那就切换到该任务，如果有在延时，那就判断下当前任务是否应该进入延时状态，如果是的话，就切换到空闲任务。否则就不进行任务切换。

• 代码清单: 空闲任务-11 (3): 任务切换，实际就是触发 PendSV 异常。

### 8.3 main() 函数

main() 函数和任务代码变动不大，具体见代码清单: 空闲任务-12，有变动部分代码已加粗。
列表 12: 代码清单: 空闲任务-12 main() 函数

```c
int main(void)
{
    OS_ERR err;

    /* 关闭中断 */
    CPU_IntDis();

    /* 配置 SysTick 10ms 中断一次 */
    OS_CPU_SysTickInit (10);

    /* 初始化相关的全局变量 */
    OSInit(&err);(1)

    /* 创建任务 */
    OSTaskCreate ((OS_TCB*) &Task1TCB,
                   (OS_TASK_PTR ) Task1,
                   (void *) 0,
                   (CPU_STK*) &Task1Stk[0],
                   (CPU_STK_SIZE) TASK1_STK_SIZE,
                   (OS_ERR *) &err);

    OSTaskCreate ((OS_TCB*) &Task2TCB,
                   (OS_TASK_PTR ) Task2,
                   (void *) 0,
                   (CPU_STK*) &Task2Stk[0],
                   (CPU_STK_SIZE) TASK2_STK_SIZE,
                   (OS_ERR *) &err);

    /* 将任务加入到就绪列表 */
    OSRdyList[0].HeadPtr = &Task1TCB;
}```
OSRdyList[1].HeadPtr = &Task2TCB;

/* 启动 OS，将不再返回 */
OSStart(&err);
}

/* 任务 1 */

void Task1(void *p_arg)
{
    for ( ;; ) {
        flag1 = 1;
        //delay( 100 );
        OSTimeDly(2);(2)
        flag1 = 0;
        //delay( 100 );
        OSTimeDly(2);

        /* 任务切换，这里是手动切换 */
        //OSSched();
    }
}

/* 任务 2 */

void Task2(void *p_arg)
{
    for ( ;; ) {
        flag2 = 1;
        //delay( 100 );
        OSTimeDly(2);(3)
        flag2 = 0;
        //delay( 100 );
        OSTimeDly(2);
• 代码清单: 空闲任务-12 (1): 空闲任务初始化函数在 OSInit 中调用，在系统启动之前创建好空闲任务。

• 代码清单: 空闲任务-12 (2) 和 (3): 延时函数均替代为阻塞延时，延时时间均为 2 个 SysTick 中断周期，即 20ms。

8.4 实验现象

进入软件调试，全速运行程序，从逻辑分析仪中可以看到两个任务的波形是完全同步，就好像CPU 在同时干两件事情，具体仿真的波形图见图实验现象 1 和图实验现象 2。
从图实验现象 1 和图实验现象 2 可以看出，flag1 和 flag2 的高电平的时间为 (0.1802-0.1602)s，刚好等于阻塞延时的 20ms，所以实验现象跟代码要实现的功能是一致的。
第9章  时间戳

本章实现时间戳用的是 ARM Cortex-M 系列内核中的 DWT 这个外设的功能，有关这个外设的功能和寄存器说明具体见手册 “STM32F10xxx Cortex-M3 programming manual”

9.1 时间戳简介

在 μC/OS-III 中，很多地方的代码都加入了时间测量的功能，比如任务关中断的时间，关调度器的时间等。知道了某段代码的运行时间，就明显地知道该代码的执行效率，如果时间过长就可以优化或者调整代码策略。如果要测量一段代码 A 的时间，那么可以在代码段 A 运行前记录一个时间点 TimeStart，在代码段 A 运行完记录一个时间点 TimeEnd，那么代码段 A 的运行时间 TimeUse 就等于 TimeEnd 减去 TimeStart。这里面的两个时间点 TimeEnd 和 TimeStart，就叫作时间戳，时间戳实际上就是一个时间点。

9.2 时间戳的实现

通常执行一条代码是需要多个时钟周期的，即是 ns 级别。要想准确测量代码的运行时间，时间戳的精度就很重要。通常单片机中的硬件定时器的精度都是 us 级别，远达不到测量几条代码运行时间的精度。

在 ARM Cortex-M 系列内核中，有一个 DWT 的外设，该外设有一个 32 位的寄存器叫 CYCCNT，它是一个向上的计数器，记录的是内核时钟 HCLK 运行的个数，当 CYCCNT 溢出之后，会清零重新开始向上计数。该计数器在 μC/OS-III 中正好被用来实现时间戳的功能。

在 STM32F103 系列的单片机中，HCLK 时钟最高为 72M，单个时钟的周期为 1/72us = 0.0139us = 14ns，CYCCNT 总共能记录的时间为 $2^{32} * 14 = 60S$。在 μC/OS-III 中，要测量的时间都是很短的，都是 ms 级别，根本不需要考虑定时器溢出的问题。如果内核代码执行的时间超过 s 的级别，那就背离了实时操作系统实时的设计初衷了，没有意义。
9.3 时间戳代码讲解

9.3.1 CPU_Init() 函数

CPU_Init() 函数在 cpu_core.c（cpu_core.c 文件第一次使用需要自行在文件夹 uC-CPU 中新建并添加到工程的 uC-CPU 组）中实现，主要做三件事：1、初始化时间戳，2、初始化中断禁用时间测量，3、初始化 CPU 名字。第 2 和 3 个功能目前还没有使用到，只实现了第 1 个初始化时间戳的代码，具体见代码清单: 时间戳-1。  

列表 1: 代码清单: 时间戳-1 CPU_Init() 函数

```c
/* CPU 初始化函数 */
void CPU_Init (void)
{
    /* CPU 初始化函数中总共做了三件事
       1. 初始化时间戳
       2. 初始化中断禁用时间测量
       3. 初始化 CPU 名字
       这里只讲时间戳功能，剩下两个的初始化代码则删除不讲 */
#if ((CPU_CFG_TS_EN == DEF_ENABLED) || (CPU_CFG_TS_TMR_EN == DEF_ENABLED))
    CPU_TS_Init();
#endif
}
```

- 代码清单: 时间戳-1 (1)：CPU_CFG_TS_EN 和 CPU_CFG_TS_TMR_EN 这两个宏在 cpu_core.h 中定义，用于控制时间戳相关的功能代码，具体定义见代码清单: 时间戳-2。
列表2：代码清单：时间戳-CPU_CFG_TS_EN 和 CPU_CFG_TS_TMR_EN 宏定义

```c
#if ((CPU_CFG_TS_32_EN == DEF_ENABLED) || (CPU_CFG_TS_64_EN == DEF_ENABLED))
#define CPU_CFG_TS_EN DEF_ENABLED
#else
#define CPU_CFG_TS_EN DEF_DISABLED
#endif

#if ((CPU_CFG_TS_EN == DEF_ENABLED) || (defined(CPU_CFG_INT_DIS_MEAS_EN)))
#define CPU_CFG_TS_TMR_EN DEF_ENABLED
#else
#define CPU_CFG_TS_TMR_EN DEF_DISABLED
#endif
```

- 代码清单：时间戳-2（1）：CPU_CFG_TS_32_EN 和 CPU_CFG_TS_64_EN 这两个宏在 cpu_cfg.h（cpu_cfg.h 文件第一次使用需要自行在文件夹 uC-CPU 中新建并添加到工程的 uC-CPU 组）文件中定义，用于控制时间戳是 32 位还是 64 位的，默认启用 32 位，具体见代码清单：时间戳-3。

列表3：代码清单：时间戳-3CPU_CFG_TS_32_EN 和 CPU_CFG_TS_64_EN 宏定义

```c
#ifndef CPU_CFG_MODULE_PRESENT
#define CPU_CFG_MODULE_PRESENT
#endif

#define CPU_CFG_TS_32_EN DEF_ENABLED
#define CPU_CFG_TS_64_EN DEF_DISABLED
```

(下页继续)


### 9.3.2 CPU_TS_Init() 函数

- 代码清单: 时间戳-1 (2): CPU_TS_Init() 是时间戳初始化函数，在 cpu_core.c 中实现，具体见代码清单: 时间戳-4。

列表4: 代码清单: 时间戳-4 CPU_TS_Init() 函数

```c
#define CPU_CFG_TS_TMR_SIZE CPU_WORD_SIZE_32

#endif /* CPU_CFG_MODULE_PRESENT */

if ((CPU_CFG_TS_EN == DEF_ENABLED) ||
    (CPU_CFG_TS_TMR_EN == DEF_ENABLED))

static void CPU_TS_Init (void)
{

#if (CPU_CFG_TS_TMR_EN == DEF_ENABLED)
    CPU_TS_TmrFreq_Hz = 0u; (1)
    CPU_TS_TmrInit(); (2)
#endif

}
#endif

```

- 代码清单: 时间戳-4 (1): CPU_TS_TmrFreq_Hz 是一个在 cpu_core.h 中定义的全局变量, 表示 CPU 的系统时钟, 具体大小跟硬件相关, 如果使用 STM32F103 系列, 那就等于 72000000HZ。CPU_TS_TmrFreq_Hz 变量的定义和时间戳相关的数据类型的定义具体见代码清单: 时间戳-5。

论坛: [https://www.firebbs.cn/](https://www.firebbs.cn/)  
天猫: [https://yehuosm.tmall.com](https://yehuosm.tmall.com)
列表 5: 代码清单: 时间戳-CPU_TS_TmrFreq_Hz 和时间戳相关的数据类型定义

```c
/*
*******************************************************************
*  EXTERNS
*  在 cpu_core.h 开头定义
*******************************************************************
*/

#define CPU_TS32
#define CPU_TS_TMR
#define CPU_TS
#define CPU_TS_TMR_FREQ

/*
*******************************************************************
* 时间戳数据类型
* 在 cpu_core.h 文件定义
*******************************************************************
*/
typedef CPU_INT32U CPU_TS32;
typedef CPU_INT32U CPU_TS_TMR_FREQ;
typedef CPU_INT32U CPU_TS;
typedef CPU_INT32U CPU_TS_TMR;

/*
*******************************************************************
* 全局变量
*******************************************************************
*/
```

(下页继续)
9.3.3 CPU_TS_TmrInit() 函数

- 代码清单：时间戳-4（2）：时间戳定时器初始化函数 CPU_TS_TmrInit() 在 cpu_core.c 实现，具体见代码清单：时间戳-6。

列表 6：代码清单：时间戳-6CPU_TS_TmrInit() 函数

```c
/* 时间戳定时器初始化 */
#if (CPU_CFG_TS_TMR_EN == DEF_ENABLED)
void CPU_TS_TmrInit (void)
{
    CPU_INT32U fclk_freq;
    fclk_freq = BSP_CPU_ClkFreq(); (2)

    /* 启用 DWT 外设 */
    BSP_REG_DEM_CR |=(CPU_INT32U)BSP_BIT_DEM_CR_TRCENA; (1)
    /* DWT CYCCNT 寄存器计数清零 */
    BSP_REG_DWT_CYCCNT = (CPU_INT32U)0u;
    /* 注意：当使用软件仿真全速运行的时候，会先停在这里，
    就好像在这里设置了一个断点一样，需要手动运行才能跳过，
    当使用硬件仿真时则不会 */
    /* 启用 Cortex-M3 DWT CYCCNT 寄存器 */
    BSP_REG_DWT_CR |=(CPU_INT32U)BSP_BIT_DWT_CR_CYCCNTENA;
#endif
```
17  CPU_TS_TmrFreqSet((CPU_TS_TMR_FREQ)fclk_freq);(3)
18  }
19  #endif
20

- 代码清单: 时间戳-6 (1): 初始化时间戳计数器 CYCCNT，启用 CYCCNT 计数的操作步骤:
   1. 先启用 DWT 外设，这个由另外内核调试寄存器 DEMCR 的位 24 控制，写 1 启用。
   2. 启用 CYCCNT 寄存器之前，先清零。
   3. 启用 CYCCNT 寄存器，这个由 DWT_CTRL(代码宏定义为 DWT_CR) 的位 0 控制，写 1 启用。这三个步骤里面涉及的寄存器定义在 cpu_core.c 文件的开头，具体见代码清单: 时间戳-7。

列表 7: 代码清单: 时间戳-7 DWT 外设相关寄存器定义

```c
/*
 * **************************************************************
 * 寄存器定义
 * **************************************************************
 */
#define BSP_REG_DEM_CR (*(CPU_REG32 *)0xE000EDFC)
#define BSP_REG_DWT_CR  (*(CPU_REG32 *)0xE0001000)
#define BSP_REG_DWT_CYCCNT (*(CPU_REG32 *)0xE0001004)
#define BSP_REG_DBGMCU_CR (*(CPU_REG32 *)0xE0042004)

/*
 * **************************************************************
 * 寄存器位定义
 * **************************************************************
 */
#define BSP_DBGMCU_CR_TRACE_IOEN_MASK 0x10
```

论坛: https://www.firebbs.cn/  
天猫: https://yehuosm.tmall.com


9.3.4 BSP_CPU_ClkFreq() 函数

- 代码清单: 时间戳-6 (2): BSP_CPU_ClkFreq() 是一个用于获取 CPU 的 HCLK 时钟的 BSP 函数，具体跟硬件相关，目前只是使用软件仿真，则把硬件相关的代码注释掉，直接手动设置
- CPU 的 HCLK 的时钟等于软件仿真的时钟 25000000HZ。BSP_CPU_ClkFreq() 在 cpu_core.c
- 实现，具体定义见 代码清单: 时间戳-8。

列表 8: 代码清单: 时间戳-8BSP_CPU_ClkFreq() 函数

/* 获取 CPU 的 HCLK 时钟
这个是跟硬件相关的，目前我们是软妹仿真，我们暂时把跟硬件相关的代码屏蔽掉，
直接手动设置 CPU 的 HCLK 时钟 */
CPU_INT32U BSP_CPU_ClkFreq (void)
{
    #if 0
        RCC_ClocksTypeDef rcc_clocks;
        RCC_GetClocksFreq(&rcc_clocks);
        return ((CPU_INT32U)rcc_clocks.HCLK_Frequency);
    #else
        CPU_INT32U CPU_HCLK;
    
 */
9.3.5 CPU_TS_TmrFreqSet() 函数

- 代码清单: 时间戳-6：CPU_TS_TmrFreqSet() 函数在 cpu_core.c 定义，具体的作用是把函数 BSP_CPU_ClkFreq() 获取到的 CPU 的 HCLK 时钟赋值给全局变量 CPU_TS_TmrFreq_Hz，具体实现见代码清单: 时间戳-9。

列表 9: 代码清单: 时间戳-9CPU_TS_TmrFreqSet() 函数

```c
/* 初始化 CPU_TS_TmrFreq_Hz，这个就是系统的时钟，单位为 HZ */
#if (CPU_CFG_TS_TMR_EN == DEF_ENABLED)
void CPU_TS_TmrFreqSet (CPU_TS_TMR_FREQ freq_hz)
{
    CPU_TS_TmrFreq_Hz = freq_hz;
}
#endif
```

9.3.6 CPU_TS_TmrRd() 函数

CPU_TS_TmrRd() 函数用于获取 CYCNNT 计数器的值，在 cpu_core.c 中定义，具体实现见代码清单: 时间戳-10。

论坛: [https://www.firebbs.cn/](https://www.firebbs.cn/)  
天猫: [https://yehuosm.tmall.com](https://yehuosm.tmall.com)
列表 10: 代码清单: 时间戳-10CPU_TS_TmrRd() 函数

```c
#if (CPU_CFG_TS_TMR_EN == DEF_ENABLED)
CPU_TS_TMR CPU_TS_TmrRd (void)
{
    CPU_TS_TMR ts_tmr_cnts;
    ts_tmr_cnts = (CPU_TS_TMR)BSP_REG_DWT_CYCCNT;
    return (ts_tmr_cnts);
}
#endif
```

### 9.3.7 OS_TS_GET() 函数

OS_TS_GET() 函数用于获取 CYCNNT 计数器的值，实际上是一个宏定义，将 CPU 底层的函数 CPU_TS_TmrRd() 重新取个名字封装，供内核和用户函数使用，在 os_cpu.h 头文件定义，具体实现见代码清单: 时间戳-11。

列表 11: 代码清单: 时间戳-11OS_TS_GET() 函数

```c
/*
 **********************************************
 * 时间戳配置
 **********************************************
*/
/* 启用时间戳，在 os_cfg.h 头文件中启用 */
#define OS_CFG_TS_EN 1u
#if OS_CFG_TS_EN == 1u
#define OS_TS_GET() (CPU_TS)CPU_TS_TmrRd()
#else
#define OS_TS_GET() (CPU_TS)0u
#endif
```
9.4 main() 函数

主函数与上一章区别不大，首先在 main() 函数开头加入 CPU_Init() 函数，然后在任务 1 中对延时函数的执行时间进行测量。新加入的代码做了加粗显示，具体见代码清单: 时间戳-12。

列表 12: 代码清单: 时间戳-12 主函数

```c
uint32_t TimeStart; /* 定义三个全局变量 */
uint32_t TimeEnd;
uint32_t TimeUse;

/*
*******************************************************************
* main() 函数
*******************************************************************
*/

int main(void)
{
    OS_ERR err;

    /* CPU 初始化: 1. 初始化时间戳 */
    CPU_Init();

    /* 关闭中断 */
    CPU_IntDis();

    /* 配置 SysTick 10ms 中断一次 */
    OS_CPU_SysTickInit (10);

    /* 初始化相关的全局变量 */
```

(下页继续)
OSInit(&err);

/* 创建任务 */
OSTaskCreate ((OS_TCB*) &Task1TCB,
(OS_TASK_PTR ) Task1,
(void *) 0,
(CPU_STK*) &Task1Stk[0],
(CPU_STK_SIZE) TASK1_STK_SIZE,
(OS_ERR *) &err);

OSTaskCreate ((OS_TCB*) &Task2TCB,
(OS_TASK_PTR ) Task2,
(void *) 0,
(CPU_STK*) &Task2Stk[0],
(CPU_STK_SIZE) TASK2_STK_SIZE,
(OS_ERR *) &err);

/* 将任务加入到就绪列表 */
OSRdyList[0].HeadPtr = &Task1TCB;
OSRdyList[1].HeadPtr = &Task2TCB;

/* 启动 OS，将不再返回 */
OSStart(&err);
}

/* 任务 1 */
void Task1( void *p_arg )
{
    for ( ;; ) {
        flag1 = 1;
        TimeStart = OS_TS_GET();
        
        /* 干一些事情 */
    }
}

*/
9.5 实验现象

时间戳时间测量功能在软件仿真时候使用不了，只能硬件仿真，这里仅能够讲解代码功能。有关硬件仿真，本书有提供一个测量 SysTick 定时时间的例程，名称叫 “7-SysTick—系统定时器 STM32 时间戳【硬件仿真】”，在配套的程序源码里面可以找到。
第 10 章 临界段

10.1 临界段简介

临界段代码，也称作临界域，是一段不可分割的代码。μC/OS 中包含了很多临界段代码。如果临界段可能被中断，那么就需要关中断以保护临界段。如果临界段可能被任务级代码打断，那么需要锁调度器保护临界段。

临界段用一句话概括就是一段在执行的时候不能被中断的代码段。在 μC/OS 里面，这个临界段最常出现的就是对全局变量的操作，全局变量就像是一个枪把子，谁都可以对他开枪；但是我开枪的时候，你就不能开枪，否则就不知道是谁命中了靶子。可能有人会说我可以在子弹上面做个标记，我说你能不能不要瞎扯淡。

那么什么情况下临界段会被打断？一个是系统调度，还有一个就是外部中断。在 μC/OS 的系统调度，最终也是产生 PendSV 中断，在 PendSV Handler 里面实现任务的切换，所以还是可以归结为中断。既然这样，μC/OS 对临界段的保护最终还是回到对中断的开和关的控制。

μC/OS 中定义了一个进入临界段的宏和两个出临界段的宏，用户可以通过这些宏定义进入临界段和退出临界段。

- OS_CRITICAL_ENTER()
- OS_CRITICAL_EXIT()
- OS_CRITICAL_EXIT_NO_SCHED()

此外还有一个开中断但是锁住调度器的宏定义 OS_CRITICAL_ENTER_CPU_EXIT()。
10.2 Cortex-M 内核快速关中断指令

为了快速地开关中断，Cortex-M 内核专门设置了一条 CPS 指令，有 4 种用法，具体见代码清单：临界段-1。

列表 1: 代码清单: 临界段-1 CPS 指令用法

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CPSID I ; PRIMASK=1</td>
<td>关中断</td>
</tr>
<tr>
<td>CPSIE I ; PRIMASK=0</td>
<td>开中断</td>
</tr>
<tr>
<td>CPSID F ; FAULTMASK=1</td>
<td>关异常</td>
</tr>
<tr>
<td>CPSIE F ; FAULTMASK=0</td>
<td>关异常</td>
</tr>
</tbody>
</table>

代码清单: 临界段-1 中 PRIMASK 和 FAULTMASK 是 Cortex-M 内核里面三个中断屏蔽寄存器中的两个，还有一个是 BASEPRI，有关这三个寄存器的详细用法见表 Cortex-M 内核中断屏蔽寄存器组描述。

表 7-1Cortex-M 内核中断屏蔽寄存器组描述

<table>
<thead>
<tr>
<th>名字</th>
<th>功能描述</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRIMASK</td>
<td>这是个只有单一比特的寄存器，在它被置 1 后，就关掉所有可屏蔽的异常，只剩下 NMI 和硬 FAULT 可以响应。它的默认值是 0，表示没有关中断。</td>
</tr>
<tr>
<td>FAULTMASK</td>
<td>这是个只有 1 个位的寄存器。当它置 1 时，只有 NMI 才能响应，所有其他的异常，甚至是硬 FAULT，也通通闭嘴。它的默认值也是 0，表示没有关异常。</td>
</tr>
<tr>
<td>BASEPRI</td>
<td>这个寄存器最多有 9 位（由表达优先级的位数决定）。它定义了被屏蔽优先级的阈值。当它被设成某个值后，所有优先级号大于等于此值的中断都被关（优先级号越大，优先级越低）。但若被设成 0，则不关闭任何中断，0 也是默认值。</td>
</tr>
</tbody>
</table>

论坛：https://www.firebbs.cn/ 118 天猫：https://yehuosm.tmall.com
但是，在μC/OS 中，对中断的开和关是通过操作 PRIMASK 寄存器来实现的，使用 CPSID I 指令就能立即关闭中断。很是方便。

### 10.3 关中断

μC/OS 中关中断的函数在 cpu_a.asm 中定义，无论上层的宏定义是怎么实现的，底层操作关中断的函数还是 CPU_SR_Save()，具体实现见代码清单: 临界段-2。

列表 2: 代码清单: 临界段-2 关中断

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU_SR_Save</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MRSR0, PRIMASK</td>
</tr>
<tr>
<td></td>
<td>CPSID I</td>
</tr>
<tr>
<td></td>
<td>BX LR</td>
</tr>
</tbody>
</table>

- **代码清单: 临界段-2 (1):** 通过 MRS 指令将特殊寄存器 PRIMASK 寄存器的值存储到通用寄存器 r0。当在 C 中调用汇编的子程序返回时，会将 r0 作为函数的返回值。所以在 C 中调用 CPU_SR_Save() 的时候，需要事先声明一个变量用来存储 CPU_SR_Save() 的返回值，即 r0 寄存器的值，也就是 PRIMASK 的值。

- **代码清单: 临界段-2 (2):** 关闭中断，即使用 CPS 指令将 PRIMASK 寄存器的值置 1。在这里，我敢肯定，一定会有人有这样一个疑问：关中断，不就是直接使用 CPSID I 指令就行了嘛，为什么还要第一步，即在执行 CPSIDI 指令前，要先把 PRIMASK 的值保存起来？这个疑问接下来在“临界段代码的应用”这个小结揭晓。

- **代码清单: 临界段-2 (3):** 子程序返回。

### 10.4 开中断

开中断要与关中断配合使用，μC/OS 中开中断的函数在 cpu_a.asm 中定义，无论上层的宏定义是怎么实现的，底层操作关中断的函数还是 CPU_SR_Restore()，具体实现见代码清单: 临界段-3。

论坛: [https://www.firebbs.cn/](https://www.firebbs.cn/)  119  天猫: [https://yehuosm.tmall.com](https://yehuosm.tmall.com)
列表 3: 代码清单: 临界段-3 开中断

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CPU_SR_Restore</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>MSR PRIMASK, R0</td>
<td>(1)</td>
</tr>
<tr>
<td>3</td>
<td>BX LR</td>
<td>(2)</td>
</tr>
</tbody>
</table>

- **代码清单: 临界段-2 (1):** 通过 MSR 指令将通用寄存器 r0 的值存储到特殊寄存器 PRIMASK。当在 C 中调用汇编的子程序返回时, 会将第一个形参传入到通用寄存器 r0。所以在 C 中调用 CPU_SR_Restore() 的时候, 需要传入一个形参, 该形参是进入临界段之前保存的 PRIMASK 的值。这个时候又有人会问, 开中断, 不就是使用 CPSIE I 指令就行了嘛, 为啥跟我等凡人想的不一样? 其中奥妙将在接下来“临界段代码的应用”这个小结揭晓。

- **代码清单: 临界段-2 (2):** 子程序返回。

### 10.5 临界段代码的应用

在进入临界段之前, 我们会先把中断关闭, 退出临界段时再把中断打开。而且 Cortex-M 内核设置了快速关中断的 CPS 指令, 那么按照我们的第一思维, 开关中断的函数的实现和临界段代码的保护应该是像 **代码清单: 临界段-4** 那样的。

列表 4: 代码清单: 临界段-4 开关中断的函数的实现和临界段代码的保护

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>; // 开关中断函数的实现</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>; /*</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>; * void CPU_SR_Save();</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>; */</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>CPU_SR_Save</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>CPSID I</td>
<td>(1)</td>
</tr>
<tr>
<td>7</td>
<td>BX LR</td>
<td></td>
</tr>
</tbody>
</table>

(下页继续)
```c
*/
* void CPU_SR_Restore(void);
*/

CPU_SR_Restore

    CPSIE I
    BX    LR

PRIMASK = 0; /* PRIMASK 初始值为 0，表示没有关中断 */

/* 临界段代码保护 */
{
    /* 临界段开始 */
    CPU_SR_Save(); /* 关中断，PRIMASK = 1 */
{
    /* 执行临界段代码，不可中断 */
}
    /* 临界段结束 */
    CPU_SR_Restore(); /* 开中断，PRIMASK = 0 */
}
```

- **代码清单: 临界段-4 (1)**：关中断直接使用了 CPSID I，没有跟代码清单: 临界段-2 一样事先将 PRIMASK 的值保存在 r0 中。

- **代码清单: 临界段-4 (2)**：开中断直接使用了 CPSIE I，而不是像代码清单: 临界段-3 那样从传进来的形参来恢复 PRIMASK 的值。

- **代码清单: 临界段-4 (4)**：假设 PRIMASK 初始值为 0，表示没有关中断。

- **代码清单: 临界段-4 (4)**：临界段开始，调用关中断函数 CPU_SR_Save()，此时 PRIMASK 的值等于 1，确实中断已经关闭。

- **代码清单: 临界段-4 (5)**：执行临界段代码，不可中断。

- **代码清单: 临界段-4 (5)**：临界段结束，调用开中断函数 CPU_SR_Restore()，此时 PRIMASK 的值等于 0，确实中断已经开启。

论坛: [https://www.firebbs.cn/](https://www.firebbs.cn/)  
天猫: [https://yehuosm.tmall.com](https://yehuosm.tmall.com)
乍一看，代码清单: 临界段-4 的这种实现开关中断的方法确实有效，没有什么错误，但是我们忽略了一种情况，就是当临界段出现嵌套的时候，这种开关中断的方法就不了了，具体怎么样不行具体见代码清单: 临界段-5。

列表 5: 代码清单: 临界段-5 开关中断的函数的实现和嵌套临界段代码的保护（有错误，只为讲解）

```c
; //开关中断函数的实现
; *
; * void CPU_SR_Save();
; */
CPU_SR_Save  
  CPSID  I
  BX  LR

; /*
; * void CPU_SR_Restore(void);
; */
CPU_SR_Restore  
  CPSIE  I
  BX  LR

PRIMASK = 0; /* PRIMASK 初始化值为 0，表示没有关中断 */

/* 临界段代码 */
{
  /* 临界段 1 开始 */
  CPU_SR_Save(); /* 关中断,PRIMASK = 1 */
  {
    /* 临界段 2 */
    CPU_SR_Save(); /* 关中断,PRIMASK = 1 */
  }
}
```

(下页继续)
论坛: https://www.firebbs.cn/ 123 天猫: https://yehuosm.tmall.com
```c
BX    LR

PRIMASK = 0;    /* PRIMASK 初始值为 0，表示没有关中断 */ (1)

CPU_SR  cpu_sr1 = (CPU_SR)0
CPU_SR  cpu_sr2 = (CPU_SR)0  (2)

/* 临界段代码 */
{
    /* 临界段 1 开始 */
    cpu_sr1 = CPU_SR_Save();    /* 关中断, cpu_sr1=0,PRIMASK=1 */ (3)
    {
        /* 临界段 2 */
        cpu_sr2 = CPU_SR_Save(); /* 关中断, cpu_sr2=1,PRIMASK=1 */ (4)
    }
    CPU_SR_Restore(cpu_sr2); /* 开中断, cpu_sr2=1,PRIMASK=1 */ (5)
}
/* 临界段 1 结束 */
CPU_SR_Restore(cpu_sr1);  /* 开中断, cpu_sr1=0,PRIMASK=0 */ (6)
```

- 代码清单: 临界段-6 (1): 假设 PRIMASK 初始值为 0，表示没有关中断。
- 代码清单: 临界段-6 (2): 定义两个变量，留着后面用。
- 代码清单: 临界段-6 (3): 临界段 1 开始，调用关中断函数 CPU_SR_Save(), CPU_SR_Save()函数先将 PRIMASK 的值存储在通用寄存器 r0，一开始我们假设 PRIMASK 的值等于 0，所以此时 r0 的值即为 0。然后执行汇编指令 CPSIDI 关闭中断，即设置 PRIMASK 等于 1，在返回的时候 r0 当做函数的返回值存储在 cpu_sr1，所以 cpu_sr1 等于 r0 等于 0。
- 代码清单: 临界段-6 (4): 临界段 2 开始，调用关中断函数 CPU_SR_Save(), CPU_SR_Save()函数先将 PRIMASK 的值存储在通用寄存器 r0，临界段 1 开始的时候我们关闭了中断，即
设置 PRIMASK 等于 1，所以此时 r0 的值等于 1。然后执行汇编指令 CPSIDI 关闭中断，即设置 PRIMASK 等于 1，在返回的时候 r0 当做函数的返回值存储在 cpu_sr2，所以 cpu_sr2 等于 r0 等于 1。

- 代码清单: 临界段-6 (5): 临界段 2 结束，调用开中断函数 CPU_SR_Restore(cpu_sr2)，cpu_sr2 作为函数的形参传入到通用寄存器 r0，然后执行汇编指令 MSR r0, PRIMASK 恢复 PRIMASK 的值。此时 PRIMASK = r0 = cpu_sr2 =1。关键点来了，为什么临界段 2 结束了，PRIMASK 还是等于 1，按道理应该是等于 0。因为此时临界段 2 是嵌套在临界段 1 中的，还是没有完全离开临界段的范畴，所以不能把中断打开，如果临界段是没有嵌套的，使用当前的开关中断的方法的话，那么 PRIMASK 确实是等于 1，具体举例见 代码清单: 临界段-7。

列表 7: 代码清单: 临界段-7 开关中断的函数的实现和一重临界段代码的保护（正确）

```c

1; //开关中断函数的实现
2; /*
3; * void CPU_SR_Save();
4; */
5CPU_SR_Save
6 MRS R0, PRIMASK
7 CPSID I
8 BX LR
9
10; /*
11; * void CPU_SR_Restore(void);
12; */
13CPU_SR_Restore
14 MSR PRIMASK, R0
15 BX LR
16
17 PRIMASK = 0; /* PRIMASK 初始化为 0，表示没有关中断 */
18
19CPU_SR cpu_sr1 = (CPU_SR)0
```

(下页继续)
```c
/* 临界段代码 */
{
    /* 临界段开始 */
cpu_sr1 = CPU_SR_Save(); /* 关中断, cpu_sr1=0,PRIMASK=1 */
}

*/

/* 临界段结束 */
CPU_SR_Restore(cpu_sr1); /* 开中断, cpu_sr1=0,PRIMASK=0 */
}
```

- 代码清单: 临界段-6 (6): 临界段 1 结束，PRIMASK 等于 0，开启中断，与进入临界段 1 遥相呼应。

### 10.6 测量关中断时间

μC/OS 提供了测量关中断时间的功能，通过设置 cpu_cfg.h 中的宏定义 CPU_CFG_INT_DIS_MEAS_EN 为 1 就表示启用该功能。

系统会在每次关中断前开始测量，开中段后结束测量，测量功能保存了 2 个方面的测量值，总的关中断时间与最近一次关中断的时间。因此，用户可以根据得到的关中断时间对其加以优化。时间戳的速率决定于 CPU 的速率。例如，如果 CPU 速率为 72MHz，时间戳的速率就为 72MHz，那么时间戳的分辨率是 1/72M 微秒，大约为 13.8 纳秒 (ns)。显然，系统测出的关中断时间还包括了测量时消耗的额外时间，那么测量得到的时间减掉测量时所耗时间就是实际上的关中断时间。关中断时间跟处理器的指令速度、速度、内存访问速度有很大的关系。
10.6.1 测量关中断时间初始化

关中断之前要用函数 CPU_IntDisMeasInit() 函数进行初始化，可以直接调用函数 CPU_Init() 函数
进行初始化，具体见代码清单: 临界段-8。

<table>
<thead>
<tr>
<th>动作</th>
<th>描述</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU_Init()</td>
<td>初始化CPU</td>
</tr>
<tr>
<td>CPU_IntDisMeasInit()</td>
<td>初始化中断测量</td>
</tr>
</tbody>
</table>

### CPU_IntDisMeasInit() 源码

```c
#define CPU_CFG_INT_DIS_MEAS_OVRHD_NBR

static void CPU_IntDisMeasInit (void)
{
    CPU_TS_TMR time_meas_tot_cnts;
    CPU_INT16U i;
    CPU_SR_ALLOC();

    CPU_IntDisMeasCtr = 0u;
    CPU_IntDisNestCtr = 0u;
    CPU_IntDisMeasStart_cnts = 0u;
    CPU_IntDisMeasStop_cnts = 0u;
    CPU_IntDisMeasMaxCur_cnts = 0u;
    CPU_IntDisMeasMax_cnts = 0u;
    CPU_IntDisMeasOvrhd_cnts = 0u;

    time_meas_tot_cnts = 0u;
    CPU_INT_DIS();           /* 关中 */
    for (i = 0u; i < CPU_CFG_INT_DIS_MEAS_OVRHD_NBR; i++)
    {
        CPU_IntDisMeasMaxCur_cnts = 0u;
        CPU_IntDisMeasStart();    /* 执行多个连续的开始/停止时间测量 */
        CPU_IntDisMeasStop();
        time_meas_tot_cnts += CPU_IntDisMeasMaxCur_cnts; /* 计算总的时间 */
    }

    CPU_IntDisMeasOvrhd_cnts = (time_meas_tot_cnts + (CPU_CFG_INT_DIS_MEAS_OVRHD_NBR / 2u))/CPU_CFG_INT_DIS_MEAS_OVRHD_NBR;
}
```

论坛: https://www.firebbs.cn/  天猫: https://yehuosm.tmall.com
因为关中断测量本身也会耗费一定的时间，这些时间实际是加入到我们测量到的最大关中断时间里面，如果能够计算出这段时间，后面计算的时候将其减去可以得到更加准确的结果。这段代码的核心思想很简单，就是重复多次开始测量与停止测量，然后多次之后，取得平均值，那么这个值就可以看作一次开始测量与停止测量的时间，保存在 CPU_IntDisMeasOvrhd_cnts 变量中。

### 10.6.2 测量最大关中断时间

如果用户启用了 CPU_CFG_INT_DIS_MEAS_EN 这个宏定义，那么系统在关中断的时候会调用开始测量关中断最大时间的函数 CPU_IntDisMeasStart()，开中断的时候调用停止测量关中断最大时间的函数 CPU_IntDisMeasStop()。从代码中我们能看到，只要在关中断且嵌套层数 OSSched-LockNestingCtr 为 0 的时候保存下时间戳，如果嵌套层数不为 0，肯定不是刚刚进入中断，退出中断且嵌套层数为 0 的时候，这个时候才算是真正的退出中断，把测得的时间戳减去一次测量额外消耗的时间，便得到这次关中断的时间，再将这个时间跟历史保存下的最大的关中断的时间对比，刷新最大的关中断时间。源码具体见代码清单：临界段-9。

**列表 9: 代码清单: 临界段-9 开始/停止测量关中断时间**

```c
/* 开始测量关中断时间 */
#elseif CPU_CFG_INT_DIS_MEAS_EN
void CPU_IntDisMeasStart (void)
{
    CPU_IntDisMeasCtr++;
    if (CPU_IntDisNestCnt == 0u) /* 嵌套层数为 0 */
```
CPU_IntDisMeasStart_cnts = CPU_TS_TmrRd();    /* 保存时间戳  */
}
CPU_IntDisNestCtr++;
}
#endif

/* 停止测量关中断时间 */
#ifdef CPU_CFG_INT_DIS_MEAS_EN
void CPU_IntDisMeasStop (void)
{
    CPU_TS_TMR      time_ints_disd_cnts;
    CPU_IntDisNestCtr--;
    if (CPU_IntDisNestCtr == 0u)    /* 嵌套层数为 0*/
    {
        CPU_IntDisMeasStop_cnts = CPU_TS_TmrRd();    /* 保存时间戳  */
        time_ints_disd_cnts = CPU_IntDisMeasStop_cnts -
                              CPU_IntDisMeasStart_cnts; /* 得到关中断时间 */
        /* 更新最大关中断时间 */
        if (CPU_IntDisMeasMaxCur_cnts < time_ints_disd_cnts)
        {
            CPU_IntDisMeasMaxCur_cnts = time_ints_disd_cnts;
        }
        if (CPU_IntDisMeasMax_cnts < time_ints_disd_cnts)
        {
            CPU_IntDisMeasMax_cnts = time_ints_disd_cnts;
        }
    }
}
#endif
10.6.3 获取最大关中断时间

现在得到了关中断时间，那么μC/OS 也提供了三个与获取关中断时间有关的函数，分别是:

- CPU_IntDisMeasMaxCurReset()
- CPU_IntDisMeasMaxCurGet()
- CPU_IntDisMeasMaxGet()

如果想直接获取整个程序运行过程中的最大关中断时间的话，直接调用函数 CPU_IntDisMeasMaxGet() 获取即可。

如果想要测量某段程序执行的最大关中断时间，那么在这段程序的前面调用 CPU_IntDisMeasMaxCurReset() 函数将 CPU_IntDisMeasMaxCur_cnts 变量清 0，在这段程序结束的时候调用函数 CPU_IntDisMeasMaxCurGet() 即可。

这些函数的源码很简单，具体见代码清单: 临界段-10。

### 列表 10: 代码清单: 临界段-10 获取最大关中断时间相关

```c
#define CPU_CFG_INT_DIS_MEAS_EN //如果启用了关中断时间测量
CPU_TS_TMR CPU_IntDisMeasMaxCurGet (void) //获取测量的程序段的最大关中断时间
{
    CPU_TS_TMR time_tot_cnts;
    CPU_TS_TMR time_max_cnts;
    CPU_SR_ALLOC(); //使用到临界段 (在关/开中断时) 时必须用到该宏，该宏声明和
    //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
    // SR (临界段关中断只需保存 SR)，开中断时将该值还原。
    CPU_INT_DIS(); //关中断
    time_tot_cnts = CPU_IntDisMeasMaxCur_cnts;
    //获取未处理的程序段最大关中断时间
    CPU_INT_EN (); //开中断
    time_max_cnts = CPU_IntDisMeasMaxCalc (time_tot_cnts);
    //获取减去测量时间后的最大关中断时间
```

return (time_max_cnts);     //返回程序段的最大关中断时间
}
#endif

 ifndef CPU_CFG_INT_DIS_MEAS_EN //如果启用了关中断时间测量
CPU_TS_TMR CPU_IntDisMeasMaxGet (void)
//获取整个程序目前最大的关中断时间
{
    CPU_TS_TMR time_tot_cnts;
    CPU_TS_TMR time_max_cnts;
    CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必须用到该宏，该宏声明和
    //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
    // SR（临界段关中断只需保存 SR），开中断时将该值还原。
    CPU_INT_DIS();       //关中断
    time_tot_cnts = CPU_IntDisMeasMax_cnts;
    //获取尚未处理的最大关中断时间
    CPU_INT_EN();        //开中断
    time_max_cnts = CPU_IntDisMeasMaxCalc(time_tot_cnts);
    //获取减去测量时间后的最大关中断时间

    return (time_max_cnts);     //返回目前最大关中断时间
}
#endif

 ifndef CPU_CFG_INT_DIS_MEAS_EN //如果启用了关中断时间测量
CPU_TS_TMR CPU_IntDisMeasMaxCurReset (void)
//初始化（复位）测量程序段的最大关中断时间
{
    CPU_TS_TMR time_max_cnts;
    CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必须用到该宏，该宏声明和
    //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
// SR (临界段关中断只需保存 SR)，开中断时将该值还原。
time_max_cnts = CPU_IntDisMeasMaxCurGet(); // 获取复位前的程序段最大关中断时间
CPU_INT_DIS(); // 关中断
CPU_IntDisMeasMaxCur_cnts = 0u; // 清零程序段的最大关中断时间
CPU_INT_EN(); // 开中断

return (time_max_cnts); // 返回复位前的程序段最大关中断时间
}
#endif

10.7 main() 函数

本章 main() 函数没有添加新的测试代码，只需理解章节内容即可。

10.8 实验现象

本章没有实验，只需理解章节内容即可。
第 11 章 就绪列表

在μC/OS-III 中，任务被创建后，任务的 TCB 会被放入就绪列表中，表示任务在就绪，随时可能被运行。就绪列表包含一个表示任务优先级的优先级表，一个存储任务 TCB 的 TCB 双向链表。

11.1 优先级表

优先级表在代码层面上来看，就是一个数组，在文件 os_prio.c（os_prio.c 第一次使用需要自行在文件夹μC/OS-III Source 中新建并添加到工程的μC/OS-III Source 组）的开头定义，具体见代码清单: 就绪列表-1。

列表 1: 代码清单: 就绪列表-1 优先级表 OSPrioTbl[] 定义

```c
/* 定义优先级表，在 os.h 中用 extern 声明 */
CPU_DATA OSPrioTbl[OS_PRIO_TBL_SIZE];(1)
```


列表 2: 代码清单: 就绪列表-2 OS_PRIO_TBL_SIZE 宏定义

```c
#define OS_PRIO_TBL_SIZE((OS_CFG_PRIO_MAX - 1u) / (DEF_INT_CPU_NBR_BITS) + 1u)
```

（下页继续）
### 野火]uCOS-III 内核实现与应用开发实战指南

(续上页)

- **代码清单: 就绪列表-2 (1):** OS_CFG_PRIO_MAX 表示支持多少个优先级，在 os_cfg.h 中定义，本书设置为 32，即最大支持 32 个优先级。

- **代码清单: 就绪列表-2 (2):** DEF_INT_CPU_NBR_BITS 定义 CPU 整型数据有多少位，本书适配的是基于 Cortex-M 系列的 MCU，宏展开为 32 位。

所以，经过 OS_CFG_PRIO_MAX 和 DEF_INT_CPU_NBR_BITS 这两个宏展开运算之后，可得出 OS_PRIO_TBL_SIZE 的值为 1，即优先级表只需要一个成员即可表示 32 个优先级。如果要支持 64 个优先级，即需要两个成员，以此类推。如果 MCU 的类型是 16 位、8 位或者 64 位，只需要把优先级表的数据类型 CPU_DATA 改成相应的位数即可。

那么优先级表又是如何跟任务的优先级联系在一起的？具体的优先级表的示意图见图优先级表。

![优先级表示意图](image)

在图优先级表中，优先级表的成员是 32 位的，每个成员可以表示 32 个优先级。如果优先级超过 32 个，那么优先级表的成员就要相应的增加。以本书为例，CPU 的类型为 32 位，支持最大的优先级为 32 个，优先级表只需要一个成员即可，即只有 OSPrioTbl[0]。假如创建一个优先级为 Prio 的任务，那么就在 OSPrioTbl[0] 的位 [31-prio] 置 1 即可。如果 Prio 等于 3，那么就将位 28 置 1。OSPrioTbl[0] 的位 31 表示的是优先级最高的任务，以此递减，直到 OSPrioTbl[OS_PRIO_TBL_SIZE-1] 的位 0，OSPrioTbl[OS_PRIO_TBL_SIZE-1] 的位 0 表示的是最低的优先级。

论坛: [https://www.firebbs.cn/](https://www.firebbs.cn/)  
天猫: [https://yehuosm.tmall.com](https://yehuosm.tmall.com)
11.1.1 优先级表函数讲解

优先级表相关的函数在 os_prio.c 文件中实现，在 os.h 文件中声明，函数汇总具体见下表。

<table>
<thead>
<tr>
<th>函数名称</th>
<th>函数作用</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS_PrioInit</td>
<td>初始化优先级表</td>
</tr>
<tr>
<td>OS_PrioInsert</td>
<td>设置优先级表中相应的位</td>
</tr>
<tr>
<td>OS_PrioRemove</td>
<td>清除优先级表中相应的位</td>
</tr>
<tr>
<td>OS_PrioGetHighest</td>
<td>查找最高的优先级</td>
</tr>
</tbody>
</table>

11.1.1.1 OS_PrioInit() 函数

OS_PrioInit() 函数用于初始化优先级表，在 OSIinit() 函数中被调用，具体实现见代码清单: 就绪列表-3。

```c
/* 初始化优先级表 */
void OS_PrioInit( void )
{
    CPU_DATA i;

    /* 默认全部初始化为 0 */
    for ( i=0u; i<OS_PRIO_TBL_SIZE; i++ )
    {
        OSPrioTbl[i] = (CPU_DATA)0;
    }
}
```

本书中，优先级表 OS_PrioTbl[] 只有一个成员，即 OS_PRIO_TBL_SIZE 等于 1 经过 代码清单: 就绪列表-3 初始化之后，具体示意图见图优先级表初始化后的示意图。

论坛：https://www.firebbs.cn/ 　　天猫：https://yehuosm.tmall.com
11.1.1.2 OS_PrioInsert() 函数

OS_PrioInsert() 函数用于置位优先级表中相应的位，会被 OSTaskCreate() 函数调用，具体实现见代码清单: 就绪列表-4。

列表 4: 代码清单: 就绪列表-4OS_PrioInsert() 函数

```c
/* 置位优先级表中相应的位 */
void OS_PrioInsert (OS_PRIO prio)
{
    CPU_DATA bit;
    CPU_DATA bit_nbr;
    OS_PRIO ix;

    /* 求模操作，获取优先级表数组的下标索引 */
    ix = prio / DEF_INT_CPU_NBR_BITS; (1)

    /* 求余操作，将优先级限制在 DEF_INT_CPU_NBR_BITS 之内 */
    bit_nbr = (CPU_DATA)prio & (DEF_INT_CPU_NBR_BITS - 1u); (2)

    /* 获取优先级在优先级表中对应的位的位置 */ (3)
    bit = 1u;
    bit <<= (DEF_INT_CPU_NBR_BITS - 1u) - bit_nbr;

    /* 将优先级在优先级表中对应的位置 1 */
    OSPrioTbl[ix] |= bit; (4)
}
```

• 代码清单: 就绪列表-4 (1): 求模操作，获取优先级表数组的下标索引。即求定 prio 这个优先级对应优先级表数组的哪个成员。假设 prio 等于 3，DEF_INT_CPU_NBR_BITS（用于表示 CPU 一个整型数有多少位）等于 32，那么 ix 就等于 0，即对应 OSPrioTBL[0]。

• 代码清单: 就绪列表-4 (2): 求余操作，将优先级限制在 DEF_INT_CPU_NBR_BITS 之内，超过 DEF_INT_CPU_NBR_BITS 的优先级就肯定要增加优先级表的数组成员了。假设 prio 等于 3，DEF_INT_CPU_NBR_BITS（用于表示 CPU 一个整型数有多少位）等于 32，那么 bit_nbr 就等于 3，但是这个还不是真正需要被置位的位。

• 代码清单: 就绪列表-4 (3): 获取优先级在优先级表中对应的位的位置。置位优先级对应的位是从高位开始的，不是从低位开始。位 31 对应的是优先级 0，在μC/OS-III 中，优先级数值越小，逻辑优先级就越高。假设 prio 等于 3，DEF_INT_CPU_NBR_BITS（用于表示 CPU 一个整型数有多少位）等于 32，那么 bit 就等于 28。

• 代码清单: 就绪列表-4 (4): 将优先级在优先级表中对应的位置 1。假设 prio 等于 3，DEF_INT_CPU_NBR_BITS（用于表示 CPU 一个整型数有多少位）等于 32，那么置位的就是 OSPrioTbl[0] 的位 28。

在优先级最大是 32，DEF_INT_CPU_NBR_BITS 等于 32 的情况下，如果分别创建了优先级 3、5、8 和 11 这四个任务，任务创建成功后，优先级表的设置情况是怎么样的？具体见图创建优先级 3-5-8 和 11 后优先级表的设置情况。有一点要注意的是，在μC/OS-III 中，最高优先级和最低优先级是留给系统任务使用的，用户任务不能使用。

![优先级表](image)

### 11.1.1.3 OS_PrioRemove() 函数

OS_PrioRemove() 函数用于清除优先级表中相应的位，与 OS_PrioInsert() 函数的作用刚好相反，具体实现见代码清单: 就绪列表-5，有关代码的讲解参考代码清单: 就绪列表-4 即可，不同的是置位操作改成了清零。

论坛: [https://www.firebbs.cn/](https://www.firebbs.cn/)  天猫: [https://yehuosm.tmall.com](https://yehuosm.tmall.com)
列表 5: 代码清单：就绪列表-5OS_PrioRemove() 函数

```c
/* 清除优先级表中相应的位 */
void OS_PrioRemove (OS_PRIO prio)
{
    CPU_DATA bit;
    CPU_DATA bit_nbr;
    OS_PRIO ix;

    /* 求模操作，获取优先级数组的下标索引 */
    ix = prio / DEF_INT_CPU_NBR_BITS;

    /* 求余操作，将优先级限制在 DEF_INT_CPU_NBR_BITS 之内 */
    bit_nbr = (CPU_DATA)prio & (DEF_INT_CPU_NBR_BITS - 1u);

    /* 获取优先级在优先级表中对应的位的位置 */
    bit = 1u;
    bit <<= (DEF_INT_CPU_NBR_BITS - 1u) - bit_nbr;

    /* 将优先级在优先级表中对应的位清零 */
    OSPrioTbl[ix] &= ~bit;
}
```

11.1.1.4 OS_PrioGetHighest() 函数

OS_PrioGetHighest() 函数用于从优先级表中查找最高的优先级，具体实现见代码清单：就绪列表-6。
列表 6: 代码清单: 就绪列表-6OS_PrioGetHighest() 函数

```c
/* 获取最高的优先级 */
OS_PRIO OS_PrioGetHighest (void)
{
    CPU_DATA  *p_tbl;
    OS_PRIO   prio;

    prio  = (OS_PRIO)0;
    /* 获取优先级表首地址 */
    p_tbl = &OSPrioTbl[0]; (1)

    /* 找到数值不为 0 的数组成员 */(2)
    while (*p_tbl == (CPU_DATA)0)
    {
        prio += DEF_INT_CPU_NBR_BITS;
        p_tbl++;
    }

    /* 找到优先级表中置位的最高的优先级 */
    prio += (OS_PRIO)CPU_CntLeadZeros(*p_tbl); (3)
    return (prio);
}
```

- 代码清单: 就绪列表-6 (1): 获取优先级表的首地址，从头开始搜索整个优先级表，直到找到最高的优先级。
- 代码清单: 就绪列表-6 (2): 找到优先级表中数值不为 0 的数组成员，只要不为 0 就表示该成员里面至少有一个位是置位的。我们知道，在图优先级表的优先级表中，优先级按照从左到右，从上到下依次减小，左上角为最高的优先级，右下角为最低的优先级，所以我们只需要找到第一个不是 0 的优先级表成员即可。
代码清单: 就绪列表-6 (3): 确定好优先级表中第一个不为 0 的成员后，然后再找出该成员
中第一个置 1 的位（从高位到低位开始找）就算找到最高优先级。在一个变量中，按照从高
位到低位的顺序查找第一个置 1 的位的方法是通过计算前导 0 函数 CPU_CntLeadZeros() 来
实现的。从高位开始找 1 叫计算前导 0，从低位开始找 1 叫计算后导 0。如果分别创建了优
先级 3、5、8 和 11 这四个任务，任务创建成功后，优先级表的设置情况具体见图创建优先级
3_5_8 和 11 后优先级表的设置情况。调用 CPU_CntLeadZeros() 可以计算出 OSPrioTbl[0]
第一个置 1 的位前面有 3 个 0，那么这个 3 就是我们要查找的最高优先级，至于后面还有
多少个位置 1 我们都不用管，只需要找到第一个 1 即可。

CPU_CntLeadZeros() 函数可由汇编或者 C 来实现，如果使用的处理器支持前导零指令 CLZ，可
由汇编来实现，加快指令运算。如果不支持则由 C 来实现。在 μCOS-III 中，这两种实现方法
均有提供代码，到底使用哪种方法由 CPU_CFG_LEAD_ZEROS_ASM_PRESEN 这个宏来控制，定义了
这个宏则使用汇编来实现，没有定义则使用 C 来实现。

Cortex-M 系列处理器自带 CLZ 指令，所以 CPU_CntLeadZeros() 函数默认由汇编编写，具体在
cpu_a.asm 文件实现，在 cpu.h 文件声明，具体见代码清单: 就绪列表-7。

列表 7: 代码清单: 就绪列表-7CPU_CntLeadZeros() 函数
实现与声明

```c
PUBLIC FUNCTIONS
```
EXPORT CPU_CntLeadZeros
EXPORT CPU_CntTrailZeros

; 计算前导 0 函数

; 描述:

; 函数声明: CPU_DATA CPU_CntLeadZeros(CPU_DATA val);

CPU_CntLeadZeros
    CLZ     R0, R0 ; Count leading zeros
    BX      LR

; 计算后导 0 函数

; 描述:

; 函数声明: CPU_DATA CPU_CntTrailZeros(CPU_DATA val);

CPU_CntTrailZeros
    RBIT    R0, R0 ; Reverse bits
    CLZ     R0, R0 ; Count trailing zeros
    BX      LR
如果处理器不支持前导 0 指令，CPU_CntLeadZeros() 函数就得由 C 编写，具体在 cpu_core.c 文件实现，在 cpu.h 文件声明，具体见代码清单: 就绪列表-8。

### 列表 8：代码清单：就绪列表-8 由 C 实现的

CPU_CntLeadZeros() 函数

```c
#ifndef CPU_CFG_LEAD_ZEROS_ASM_PRESENT
CPU_DATA CPU_CntLeadZeros (CPU_DATA val)
{
    CPU_DATA    nbr_lead_zeros;
    CPU_INT08U  ix;

    /* 检查高 16 位 */
    if (val > 0x0000FFFFu) { (1)
    /* 检查 bits [31:24] : */
    if (val > 0x00FFFFFFu) { (2)

    /* 获取 bits [31:24] 的值，并转换成 8 位 */
    ix = (CPU_INT08U)(val >> 24u); (3)
    /* 查表找到优先级 */
    nbr_lead_zeros = (CPU_DATA)(CPU_CntLeadZerosTbl[ix] + 0u); (4)
```


```c
}    /* 检查 bits [23:16] : */
else {
    /* 获取 bits [23:16] 的值，并转换成 8 位 */
    ix = (CPU_INT08U)(val >> 16u);
    /* 查表找到优先级 */
    nbr_lead_zeros = (CPU_DATA )(CPU_CntLeadZerosTbl[ix] + 8u);
}

} /* 检查低 16 位 */
else {
    /* 检查 bits [15:08] : */
    if (val > 0x000000FFu) {
        /* 获取 bits [15:08] 的值，并转换成 8 位 */
        ix = (CPU_INT08U)(val >> 8u);
        /* 查表找到优先级 */
        nbr_lead_zeros = (CPU_DATA )(CPU_CntLeadZerosTbl[ix] + 16u);
    }
    /* 检查 bits [07:00] : */
    else {
        /* 获取 bits [15:08] 的值，并转换成 8 位 */
        ix = (CPU_INT08U)(val >> 0u);
        /* 查表找到优先级 */
        nbr_lead_zeros = (CPU_DATA )(CPU_CntLeadZerosTbl[ix] + 24u);
    }
}

/* 返回优先级 */
return (nbr_lead_zeros);
```
在 μC/OS-III 中，由 C 实现的 CPU_CntLeadZeros() 函数支持 8 位、16 位、32 位和 64 位的变量的前导 0 计算，但最终的代码实现都是分离成 8 位来计算。这里我们只讲解 32 位的，其他几种情况都类似。

- 代码清单: 就绪列表-8 (1): 分离出高 16 位，else 则为低 16 位。
- 代码清单: 就绪列表-8 (2): 分离出高 16 位的高 8 位，else 则为高 16 位的低 8 位。
- 代码清单: 就绪列表-8 (3): 将高 16 位的高 8 位通过移位强制转化为 8 位的变量，用于后面的查表操作。
- 代码清单: 就绪列表-8 (4): 将 8 位的变量 ix 作为数组 CPU_CntLeadZerosTbl[] 的索引，返回索引对应的值，那么该值就是 8 位变量 ix 对应的前导 0，然后再加上 (24-右移的位数) 就等于优先级。数组 CPU_CntLeadZerosTbl[] 在 cpu_core.c 的开头定义，具体见 代码清单: 就绪列表-9。

列表 9: 代码清单: 就绪列表-9CPU_CntLeadZerosTbl[] 定义

```c
#define CPU_CFG_LEAD_ZEROS_ASM_PRESENT

static const CPU_INT08U CPU_CntLeadZerosTbl[256] = { /* 索引 */
  0x00, /* 0x00 to 0xFF */
  0x10, /* 0x00 to 0xFF */
  0x20, /* 0x00 to 0xFF */
  0x30, /* 0x00 to 0xFF */
  0x40, /* 0x00 to 0xFF */
  0x50, /* 0x00 to 0xFF */
  0x60, /* 0x00 to 0xFF */
  0x70, /* 0x00 to 0xFF */
  0x80, /* 0x00 to 0xFF */
  0x90, /* 0x00 to 0xFF */
```
代码清单: 就绪列表-8 中，对一个 32 位的变量算前导 0 个数的时候都是分离成 8 位的变量来计算，然后将这个 8 位的变量作为数组 CPU_CntLeadZerosTbl[] 的索引，索引下对应的值就是这个 8 位变量的前导 0 个数。一个 8 位的变量的取值范围为 0~0xFF，这些值作为数组 CPU_CntLeadZerosTbl[] 的索引，每一个值的前导 0 个数都预先算出来作为该数组索引下的值。通过查 CPU_CntLeadZerosTbl[] 这个表就可以很快的知道一个 8 位变量的前导 0 个数，根本不用计算，只是浪费了定义 CPU_CntLeadZerosTbl[] 这个表的一点点空间而已，在处理器内存很充足的情况下，则优先选择这种空间换时间的方法。

### 11.2 就绪列表

准备好运行的任务的 TCB 都会被放到就绪列表中，系统可随时调度任务运行。就绪列表在代码的层面上看就是一个 OS_RDY_LIST 数据类型的数组 OSRdyList[]，数组的大小由宏 OS_CFG_PRIO_MAX 决定，支持多少个优先级，OSRdyList[] 就有多少个成员。任务的优先级与 OSRdyList[] 的索引一一对应，比如优先级 3 的任务的 TCB 会被放到 OSRdyList[3] 中。OSRdyList[] 是一个在 os.h 文件中定义的全局变量，具体见代码清单: 就绪列表-10。
列表 10: 代码清单: 就绪列表-10 OSRdyList[] 数组定义

```c
/* 就绪列表定义 */
OS_EXT OS_RDY_LIST OSRdyList[OS_CFG_PRIO_MAX];
```

代码清单: 就绪列表-10 中的数据类型 OS_RDY_LIST 在 os.h 中定义，专用于就绪列表，具体实现见代码清单: 就绪列表-11。

列表 11: 代码清单: 就绪列表-11 OS_RDY_LIST 定义

```c
typedef struct os_rdy_list OS_RDY_LIST; (1)

struct os_rdy_list {
    OS_TCB *HeadPtr; (2)
    OS_TCB *TailPtr;
    OS_OBJ_QTY NbrEntries; (3)
};
```

- 代码清单: 就绪列表-11 (1): 在 μC/OS-III 中，内核对象的数据类型都会用大写字母重新定义。
- 代码清单: 就绪列表-11 (2): OSRdyList[] 的成员与任务的优先级一一对应，同一个优先级的多个任务会以双向链表的形式存在 OSRdyList[] 同一个索引下，那么 HeadPtr 就用于指向链表的头节点，TailPtr 用于指向链表的尾节点，该优先级下的索引成员的地址则称为该优先级下双向链表的根节点，知道根节点的地址就可以查找到该链表下的每一个节点。
- 代码清单: 就绪列表-11 (3): NbrEntries 表示 OSRdyList[] 同一个索引下有多少个任务。

一个空的就绪列表，OSRdyList[] 索引下的 HeadPtr、TailPtr 和 NbrEntries 都会被初始化为 0，具体见图空的就绪列表。
就绪列表相关的所有函数都在 os_core.c 实现，这些函数都是以“OS_”开头，表示是 OS 的内部函数，用户不能调用，这些函数的汇总具体见下表。

表：就绪列表相关函数汇总

<table>
<thead>
<tr>
<th>函数名称</th>
<th>函数作用</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS_RdyListInit</td>
<td>初始化就绪列表为空</td>
</tr>
<tr>
<td>OS_RdyListInsert</td>
<td>插入一个 TCB 到就绪列表</td>
</tr>
<tr>
<td>OS_RdyListInsertHead</td>
<td>插入一个 TCB 到就绪列表的头部</td>
</tr>
<tr>
<td>OS_RdyListInsertTail</td>
<td>插入一个 TCB 到就绪列表的尾部</td>
</tr>
<tr>
<td>OS_RdyListMoveHeadToTail</td>
<td>将 TCB 从就绪列表的头部移到尾部</td>
</tr>
<tr>
<td>OS_RdyListRemove</td>
<td>将 TCB 从就绪列表中移除</td>
</tr>
</tbody>
</table>

论坛：https://www.firebbs.cn/  天猫：https://yehuosm.tmall.com
11.2.1 就绪列表函数讲解

在实现就绪列表相关函数之前，我们需要在结构体 os_tcb 中添加 Prio、NextPtr 和 PrevPtr 这三个成员，然后在 os.h 中定义两个全局变量 OSPrioCur 和 OSPrioHighRdy，具体定义见代码清单: 就绪列表-12。接下来要实现的就绪列表相关的函数会用到几个变量。

列表 12: 代码清单：就绪列表-12 就绪列表函数需要用到的变量定义

```c
struct os_tcb {
    CPU_STK    *StkPtr;
    CPU_STK_SIZE StkSize;

    /* 任务延时周期个数 */
    OS_TICK      TaskDelayTicks;

    /* 任务优先级 */
    OS_PRIO      Prio;

    /* 就绪列表双向链表的下一个指针 */
    OS_TCB      *NextPtr;

    /* 就绪列表双向链表的前一个指针 */
    OS_TCB      *PrevPtr;
};

/* 在 os.h 中定义 */
OS_EXT      OS_PRIO OSPrioCur;    /* 当前优先级 */
OS_EXT      OS_PRIO OSPrioHighRdy; /* 最高优先级 */
```
11.2.1.1 OS_RdyListInit() 函数

OS_RdyListInit() 用于将就绪列表 OSRdyList[] 初始化为空，初始化完毕之后具体示意图见图空的
就绪列表，具体实现见代码清单: 就绪列表-13。

列表 13: 代码清单: 就绪列表-13OS_RdyListInit() 函数

```c
void OS_RdyListInit(void)
{
    OS_PRIO i;
    OS_RDY_LIST *p_rdy_list;

    /* 循环初始化，所有成员都初始化为 0 */
    for ( i=0; i<OS_CFG_PRIO_MAX; i++ ) {
        p_rdy_list = &OSRdyList[i];
        p_rdy_list->NbrEntries = (OS_OBJ_QTY)0;
        p_rdy_list->HeadPtr = (OS_TCB *)0;
        p_rdy_list->TailPtr = (OS_TCB *)0;
    }
}
```

11.2.1.2 OS_RdyListInsertHead() 函数

OS_RdyListInsertHead() 用于在链表头部插入一个 TCB 节点，插入的时候分两种情况，第一种是
链表是空链表，第二种是链表中已有节点，具体示意图见图在链表的头部插入一个 TCB 节点前
链表的可能情况，具体的代码实现见代码清单: 就绪列表-14，阅读代码的时候最好配套示意图
来理解。
void OS_RdyListInsertHead (OS_TCB *p_tcb)
{
    OS_RDY_LIST *p_rdy_list;
    OS_TCB     *p_tcb2;

    /* 获取链表根部 */
    p_rdy_list = &OSRdyList[p_tcb->Prio];
/* CASE 0: 链表是空链表 */
if (p_rdy_list->NbrEntries == (OS_OBJ_QTY)0) {
    p_rdy_list->NbrEntries = (OS_OBJ_QTY)1;
    p_tcb->NextPtr = (OS_TCB *)0;
    p_tcb->PrevPtr = (OS_TCB *)0;
    p_rdy_list->HeadPtr = p_tcb;
    p_rdy_list->TailPtr = p_tcb;
}
/* CASE 1: 链表已有节点 */
else {
    p_rdy_list->NbrEntries++;
    p_tcb->NextPtr = p_rdy_list->HeadPtr;
    p_tcb->PrevPtr = (OS_TCB *)0;
    p_tcb2 = p_rdy_list->HeadPtr;
    p_tcb2->PrevPtr = p_tcb;
    p_rdy_list->HeadPtr = p_tcb;
}

11.2.1.3 OS_RdyListInsertTail() 函数

OS_RdyListInsertTail() 用于在链表尾部插入一个 TCB 节点，插入的时候分两种情况，第一种是链表是空链表，第二种是链表中已有节点，具体示意图见图在链表的尾部插入一个 TCB 节点前链表的可能情况，具体的代码实现见，阅读代码的时候最好配合示意图来理解。
/*
* 在就绪链表尾部插入一个TCB 有两种情况
* CASE 0: 链表是空的
* OS_RDY_LIST
*  | TailPtr   | HeadPtr    | NbrEntries |
*  | -> 0      | -> 0       | = 0        |
*
* CASE 1: 链表已有节点
* OS_RDY_LIST
*  | TailPtr   | HeadPtr    | OS_TCB
*  |  | NextPtr     |
*  |  | PrevPtr     |
*  |  | NbrEntries=|
*  |  | 0<-
*  |  | 0<-
*  |  | 0<-
*
*/

void OS_RdyListInsertTail(OS_TCB *p_tcb)
{
    OS_RDY_LIST *p_rdy_list;
    OS_TCB *p_tcb2;

    /* 获取链表根部 */
    p_rdy_list = &OSRdyList[p_tcb->Prio];
/* CASE 0: 链表是空链表 */
if (p_rdy_list->NbrEntries == (OS_OBJ_QTY)0) {
    p_rdy_list->NbrEntries = (OS_OBJ_QTY)1;
    p_tcb->NextPtr = (OS_TCB *)0;
    p_tcb->PrevPtr = (OS_TCB *)0;
    p_rdy_list->HeadPtr = p_tcb;
    p_rdy_list->TailPtr = p_tcb;
}

/* CASE 1: 链表已有节点 */
else {
    p_rdy_list->NbrEntries++;
    p_tcb->NextPtr = (OS_TCB *)0;
    p_tcb2 = p_rdy_list->TailPtr;
    p_tcb->PrevPtr = p_tcb2;
    p_tcb2->NextPtr = p_tcb;
    p_rdy_list->TailPtr = p_tcb;
}

11.2.1.4 OS_RdyListInsert() 函数

OS_RdyListInsert() 用于将任务的 TCB 插入就绪列表，插入的时候分成两步，第一步是根据优先级将优先级表中的相应位置位，这个调用 OS_PrioInsert() 函数来实现，第二步是根据优先级将任务的 TCB 放到 OSrdyList[优先级] 中，如果优先级等于当前的优先级则插入链表的尾部，否则插入链表的头部，具体实现见代码清单: 就绪列表-16。

列表 16: 代码清单: 就绪列表-16 OS_RdyListInsert() 函数

/* 在就绪链表中插入一个 TCB */
void OS_RdyListInsert (OS_TCB *p_tcb)
{  
    /* 将优先级插入优先级表 */  
    OS_PrioInsert(p_tcb->Prio);  

    if (p_tcb->Prio == OSPrioCur)  
    {  
        /* 如果是当前优先级则插入链表尾部 */  
        OS_RdyListInsertTail(p_tcb);  
    }  
    else  
    {  
        /* 否则插入链表头部 */  
        OS_RdyListInsertHead(p_tcb);  
    }  
}

11.2.1.5 OS_RdyListMoveHeadToTail( ) 函数

OS_RdyListMoveHeadToTail( ) 函数用于将节点从链表头部移动到尾部，移动的时候分四种情况，
第一种是链表为空，无事可做；第二种是链表只有一个节点，也是无事可做；第三种是链表只有
两个节点；第四种是链表有两个以上节点，具体示意图见图将节点从链表头部移动到尾部前链表
的可能情况，具体代码实现见代码清单: 就绪列表-17，阅读代码的时候最好配套示意图来理解。
void OS_RdyListMoveHeadToTail (OS_RDY_LIST *p_rdy_list)
{
    OS_TCB *p_tcb1;
    OS_TCB *p_tcb2;
    OS_TCB *p_tcb3;

    switch (p_rdy_list->NbrEntries) {
        case 0:
        case 1:
            break;
    }

    // 以下继续
case 2:
    p_tcb1 = p_rdy_list->HeadPtr;
    p_tcb2 = p_rdy_list->TailPtr;
    p_tcb1->PrevPtr = p_tcb2;
    p_tcb1->NextPtr = (OS_TCB *)0;
    p_tcb2->PrevPtr = (OS_TCB *)0;
    p_tcb2->NextPtr = p_tcb1;
    p_rdy_list->HeadPtr = p_tcb2;
    p_rdy_list->TailPtr = p_tcb1;
break;

default:
    p_tcb1 = p_rdy_list->HeadPtr;
    p_tcb2 = p_rdy_list->TailPtr;
    p_tcb3 = p_tcb1->NextPtr;
    p_tcb3->PrevPtr = (OS_TCB *)0;
    p_tcb1->NextPtr = (OS_TCB *)0;
    p_tcb1->PrevPtr = p_tcb2;
    p_tcb2->NextPtr = p_tcb1;
    p_rdy_list->HeadPtr = p_tcb3;
    p_rdy_list->TailPtr = p_tcb1;
break;

}
### 11.2.1.6 OS_RdyListRemove() 函数

OS_RdyListRemove() 函数用于从链表中移除一个节点，移除的时候分为三种情况，第一种是链表为空，无事可做；第二种是链表只有一个节点；第三种是链表有两个以上节点，具体示意图见图从链表中移除一个节点前链表的可能情况，具体代码实现见，阅读代码的时候最好配合示意图来理解。

```c
void OS_RdyListRemove (OS_TCB *p_tcb)
{
    OS_RDY_LIST  *p_rdy_list;
    OS_TCB      *p_tcb1;
    OS_TCB      *p_tcb2;
```

论坛: https://www.firebbs.cn/  
天猫: https://yehuosm.tmall.com
p_rdy_list = &OSRdyList[p_tcb->Prio];

/* 保存要删除的 TCB 节点的前一个和后一个节点 */
p_tcb1 = p_tcb->PrevPtr;
p_tcb2 = p_tcb->NextPtr;

/* 要移除的 TCB 节点是链表中的第一个节点 */
if (p_tcb1 == (OS_TCB*)0)
{
    /* 且该链表中只有一个节点 */
    if (p_tcb2 == (OS_TCB*)0)
    {
        /* 根节点全部初始化为 0 */
        p_rdy_list->NbrEntries = (OS_OBJ_QTY)0;
        p_rdy_list->HeadPtr = (OS_TCB*)0;
        p_rdy_list->TailPtr = (OS_TCB*)0;

        /* 清除在优先级表中相应的位 */
        OS_PrioRemove(p_tcb->Prio);
    } /* 该链表中不止一个节点 */
    else
    {
        /* 节点减 1 */
        p_rdy_list->NbrEntries--;
        p_tcb2->PrevPtr = (OS_TCB*)0;
        p_rdy_list->HeadPtr = p_tcb2;
    }
} /* 要移除的 TCB 节点不是链表中的第一个节点 */
else
{

}
p_rdy_list->NbrEntries--;  
p_tcb1->NextPtr = p_tcb2;

/* 如果要删除的节点的下一个节点是 0，即删除的节点是最后一个节点 */
if (p_tcb2 == (OS_TCB *)0) {
    p_rdy_list->TailPtr = p_tcb1;
}
else {
    p_tcb2->PrevPtr = p_tcb1;
}

/* 复位从就绪列表中删除的 TCB 的 PrevPtr 和 NextPtr 这两个指针 */
p_tcb->PrevPtr = (OS_TCB *)0;
p_tcb->NextPtr = (OS_TCB *)0;

11.3 main() 函数

本章 main() 函数没有添加新的测试代码，只需理解章节内容即可。

11.4 实验现象

本章没有实验，只需理解章节内容即可。
第 12 章 支持多优先级

在本章之前，OS 还没有到优先级，只支持两个任务互相切换，从本章开始，任务中我们开始加入优先级的功能。在 μC/OS-III 中，数字优先级越小，逻辑优先级越高。

12.1 定义优先级相关全局变量

在支持任务多优先级的时候，需要在 os.h 头文件添加两个优先级相关的全局变量，具体定义见代 码清单: 优先级-1。

列表 1: 代码清单: 优先级-1 定义优先级相关全局变量

```c
/* 在 os.h 中定义 */
/* 当前优先级 */
#define OS_PRIO OSPrioCur;
/* 最高优先级 */
#define OS_EXT OS_PRIO OSPrioHighRdy;
```

12.2 修改 OSInit() 函数

刚刚新添加的优先级相关的全部变量，需要在 OSInit() 函数中进行初始化，具体见代 码清单: 优先级-2 中的加粗部分代码，其实 OS 中定义的所有的全局变量都是在 OSInit() 中初始化的。

列表 2: 代码清单: 优先级-2OSInit() 函数

```c
void OSInit (OS_ERR *p_err)
{
    /* 配置 OS 初始状态为停止态 */

```
12.3 修改任务控制块 TCB

在任务控制块中，加入优先级字段 Prio，具体见代码清单: 优先级-3 中的加粗代码。优先级 Prio 的数据类型为 OS_PRIO，展开后是 8 位的整型，所以只支持 255 个优先级。
12.4 修改 OSTaskCreate() 函数

修改 OSTaskCreate() 函数，在里面加入优先级相关的处理，具体见代码清单: 优先级-4 的加粗部分。

列表 4: 代码清单: 优先级-4OSTaskCreate() 函数加入优先级处理

```c
void OSTaskCreate (OS_TCB *p_tcb,
                   OS_TASK_PTR p_task,
                   void *p_arg,
                   OS_PRIO prio, (1)
                   CPU_STK *p_stk_base,
```

(下页继续)
CPU_STK_SIZE stk_size,
OS_ERR *p_err)
{
    CPU_STK *p_sp;
    CPU_SR_ALLOC(); (2)

    /* 初始化 TCB 为默认值 */
    OS_TaskInitTCB(p_tcb); (3)

    /* 初始化栈 */
    p_sp = OSTaskStkInit( p_task,
        p_arg,
        p_stk_base,
        stk_size );

    p_tcb->Prio = prio; (4)

    p_tcb->StkPtr = p_sp;
    p_tcb->StkSize = stk_size;

    /* 进入临界段 */
    OS_CRITICAL_ENTER(); (5)

    /* 将任务添加到就绪列表 */
    OS_PrioInsert(p_tcb->Prio);
    OS_RdyListInsertTail(p_tcb);

    /* 退出临界段 */
    OS_CRITICAL_EXIT(); (7)

    *p_err = OS_ERR_NONE;
}
- 代码清单: 优先级-4 (1): 在函数形参中，加入优先级字段。任务的优先级由用户在创建任务的时候通过形参 Prio 传进来。

- 代码清单: 优先级-4 (2): 定义一个局部变量，用来存 CPU 关中断前的中断状态。因为接下来将任务添加到就绪列表，这段代码属于临界段代码，需要关中断。

- 代码清单: 优先级-4 (3): 初始化 TCB 为默认值，其实就是全部初始化为 0, OS_TaskInitTCB() 函数在 os_task.c 的开头定义，具体见代码清单: 优先级-5。

列表 5: 代码清单: 优先级-5OS_TaskInitTCB() 函数

```c
void OS_TaskInitTCB(OS_TCB *p_tcb) {
    p_tcb->StkPtr = (CPU_STK *) 0;
    p_tcb->StkSize = (CPU_STK_SIZE ) 0u;
    p_tcb->TaskDelayTicks = (OS_TICK ) 0u;
    p_tcb->Prio = (OS_PRIO ) OS_PRIO_INIT; (1)
    p_tcb->NextPtr = (OS_TCB ) 0;
    p_tcb->PrevPtr = (OS_TCB ) 0;
}
```

- 代码清单: 优先级-5 (1): OS_PRIO_INIT 是任务 TCB 初始化的时候给的默认的一个优先级宏展开等于 OS_CFG_PRIO_MAX，这是一个不会被 OS 使用到的优先级。OS_PRIO_INIT 具体在 os.h 中定义。

- 代码清单: 优先级-4 (4): 将形参传进来的优先级存到任务控制块 TCB 的优先级字段。

- 代码清单: 优先级-4 (5): 进入临界段。


- 代码清单: 优先级-4 (7): 退出临界段。

论坛: https://www.firebbs.cn/  164  天猫: https://yehuosm.tmall.com
12.5 修改 OS_IdleTaskInit() 函数

修改 OS_IdleTaskInit() 函数，是因为该函数调用了任务创建函数 OSTaskCreate()，OSTaskCreate()我们刚刚加入了优先级，所以这里我们要分配一个优先级，具体见。代码清单: 优先级-6 的加粗部分。

列表 6: 代码清单: 优先级-6 OS_IdleTaskInit() 函数

```c
/* 空闲任务初始化 */
void OS_IdleTaskInit(OS_ERR *p_err)
{
    /* 初始化空闲任务计数器 */
    OSIdleTaskCtr = (OS_IDLE_CTR)0;

    /* 创建空闲任务 */
    OSTaskCreate( (OS_TCB *)&OSIdleTaskTCB,
                   (OS_TASK_PTR )OS_IdleTask,
                   (void *)0,
                   (OS_PRIO)(OS_CFG_PRIO_MAX - 1u), (1)
                   (CPU_STK *)OSCfg_IdleTaskStkBasePtr,
                   (CPU_STK_SIZE)OSCfg_IdleTaskStkSize,
                   (OS_ERR *)p_err );
}
```

- 代码清单: 优先级-6 (1): 空闲任务是 μC/OS-III 的内部任务，在 OSInit() 中被创建，系统没有任何用户任务运行的情况下，空闲任务就会被运行，优先级最低，即等于 OS_CFG_PRIO_MAX - 1u。
12.6 修改 OSStart() 函数

加入优先级之后，OSStart() 函数需要修改，具体哪一个任务最先运行，由优先级决定，新加入的代码具体见代码清单: 优先级-7 的加粗部分。

列表 7: 代码清单: 优先级-7OSStart() 函数

```c
/* 启动RTOS，将不再返回 */
void OSStart (OS_ERR *p_err)
{
    if (OSRunning == OS_STATE_OS_STOPPED) {
        #if 0
            /* 手动配置任务1先运行 */
            OSTCBHighRdyPtr = OSRdyList[0].HeadPtr;
        #endif
        /* 寻找最高的优先级 */
        OSPrioHighRdy = OS_PrioGetHighest();(1)
        OSPrioCur = OSPrioHighRdy;

        /* 找到最高优先级的TCB */
        OSTCBHighRdyPtr = OSRdyList[OSPrioHighRdy].HeadPtr;(2)
        OSTCBCurPtr = OSTCBHighRdyPtr;

        /* 标记OS开始运行 */
        OSRunning = OS_STATE_OS_RUNNING;

        /* 启动任务切换，不会返回 */
        OSStartHighRdy();

        /* 不会运行到这里，运行到这里表示发生了致命的错误 */
        *p_err = OS_ERR_FATAL_RETURN;
    } else {
        *p_err = OS_STATE_OS_RUNNING;
    }
```

(下页继续)
• 代码清单: 优先级-7 (1): 调取 OS_PrioGetHighest() 函数从全局变量优先级表 OSPrioTbl[] 获取最高的优先级，放到 OSPrioHighRdy 这个全局变量中，然后把 OSPrioHighRdy 的值再赋给当前优先级 OSPrioCur 这个全局变量。在任务切换的时候需要用到 OSPrioHighRdy 和 OSPrioCur 这两个全局变量。

• 代码清单: 优先级-7 (2): 根据 OSPrioHighRdy 的值，作为全局变量 OSRdyList[] 的下标索引找到最高优先级任务的 TCB，传给全局变量 OSTCBHighRdyPtr，然后再将 OSTCBHighRdyPtr 赋值给 OSTCBCurPtr。在任务切换的时候需要使用到 OSTCBHighRdyPtr 和 OSTCBCurPtr 这两个全局变量。

12.7 修改 PendSV_Handler() 函数

PendSV_Handler() 函数中添加了优先级相关的代码，具体见代码清单: 优先级-8 中加粗部分。有关 PendSV_Handler() 这个函数的具体讲解要参考《任务的定义与任务切换的实现》这个章节，这里不再赘述。

列表 8: 代码清单: 优先级-8PendSV_Handler() 函数

```assembly
; ******************************************************
; PendSVHandler 异常
; ******************************************************

OS_CPU_PendSVHandler_nosave

; OSPrioCur = OSPrioHighRdy
LDR  R0, =OSPrioCur
LDR  R1, =OSPrioHighRdy
LDRB R2, [R1]
```

论坛: [https://www.firebbs.cn/](https://www.firebbs.cn/)  天猫: [https://yehuosm.tmall.com](https://yehuosm.tmall.com)
12.8 修改 OSTimeDly() 函数

任务调用 OSTimeDly() 函数之后，任务就处于阻塞态，需要将任务从就绪列表中移除，具体修改的代码见代码清单: 优先级-9 的加粗部分。

列表 9: 代码清单: 优先级-9OSTimeDly() 函数

```c
/* 阻塞延时 */
void OSTimeDly(OS_TICK dly)
```
```c
{  
  #if 0
    /* 设置延时时间 */
    OSTCBCurPtr->TaskDelayTicks = dly;

    /* 进行任务调度 */
    OSSched();
  #endif

  CPU_SR_ALLOC();(1)

  /* 进入临界区 */
  OS_CRITICAL_ENTER();(2)

  /* 设置延时时间 */
  OSTCBCurPtr->TaskDelayTicks = dly;

  /* 从就绪列表中移除 */
  //OS_RdyListRemove(OSTCBCurPtr);
  OS_PrioRemove(OSTCBCurPtr->Prio);(3)

  /* 退出临界区 */
  OS_CRITICAL_EXIT();(4)

  /* 任务调度 */
  OSSched();
}
```

- 代码清单: 优先级-9 (1): 定义一个局部变量，用来存CPU关中断前的中断状态，因为接下来将任务从就绪列表移除这段代码属于临界短代码，需要关中断。
- 代码清单: 优先级-9 (2): 进入临界段
• 代码清单: 优先级-9 (3): 将任务从就绪列表移除，这里只需将任务在优先级表中对应的位清除即可，暂时不需要把任务 TCB 从 OSRdyList[] 中移除，因为接下来 OSTimeTick() 函数还是通过扫描 OSRdyList[] 来判断任务的延时时间是否到期。当我们加入了时基列表之后，当任务调用 OSTimeDly() 函数进行延时，就可以把任务的 TCB 从就绪列表删除，然后把任务 TCB 插入时基列表，OSTimeTick() 函数判断任务的延时是否到期只需通过扫描时基列表即可，时基列表在下一个章节实现。所以这里暂时不能把 TCB 从就绪列表中删除，只是将任务优先级在优先级表中对应的位清除来达到任务不处于就绪态的目的。

• 代码清单: 优先级-9 (4): 退出临界段。

12.9 修改 OSSched() 函数

任务调度函数 OSSched() 不再是之前的两个任务轮流切换，需要根据优先级来调度，具体修改部分见代码清单: 优先级-10 的加粗部分，被迭代的代码已经通过条件编译屏蔽。

列表 10: 代码清单: 优先级-10 OSSched() 函数

```
void OSSched(void)
{
  #if 0
      /* 如果当前任务是空闲任务，那么就去尝试执行任务 1 或者任务 2，
      看看他们的延时时间是否结束，如果任务的延时时间均没有到期，
      那就返回继续执行空闲任务 */
      if (OSTCBCurPtr == &OSIdleTaskTCB )
      {
          if (OSRdyList[0].HeadPtr->TaskDelayTicks == 0)
          {
              OSTCBHighRdyPtr = OSRdyList[0].HeadPtr;
          }
          else if (OSRdyList[1].HeadPtr->TaskDelayTicks == 0)
          {
              OSTCBHighRdyPtr = OSRdyList[1].HeadPtr;

```

论坛: https://www.firebbs.cn/ 170 天猫: https://yehuosm.tmall.com
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47

} else
{ return; /* 任务延时均没有到期则返回，继续执行空闲任务 */
}
} else
{
/* 如果是 task1 或 task2 的话，检查下另外一个任务，
如果另外的任务不在延时中，就切换到该任务，
否则，判断下当前任务是否应该进入延时状态，
如果是的话，就切换到空闲任务。否则就不进行任何切换 */
if (OSTCBCurPtr == OSRdyList[0].HeadPtr)
{
    if (OSRdyList[1].HeadPtr->TaskDelayTicks == 0)
    {
        OSTCBHighRdyPtr = OSRdyList[1].HeadPtr;
    } else if (OSTCBCurPtr->TaskDelayTicks != 0) {
        OSTCBHighRdyPtr = &OSIdleTaskTCB;
    } else {
        /* 返回，不进行切换，因为两个任务都处于延时中 */
        return;
    }
}
else if (OSTCBCurPtr == OSRdyList[1].HeadPtr)
{
    if (OSRdyList[0].HeadPtr->TaskDelayTicks == 0)
    {
        OSTCBHighRdyPtr = OSRdyList[0].HeadPtr;
    }
    else if (OSTCBCurPtr->TaskDelayTicks != 0)
    {
    
}
OSTCBHighRdyPtr = &OSIdleTaskTCB;
else
{
    /* 返回，不进行切換，因为两个任务都处于延时中 */
    return;
}
}

/* 任务切換 */
OS_TASK_SW();
#endif

CPU_SR_ALLOC();

/* 进入临界区 */
OS_CRITICAL_ENTER();

/* 查找最高优先级的任务 */
OSPrioHighRdy = OS_PrioGetHighest();
OSTCBHighRdyPtr = OSRdyList[OSPrioHighRdy].HeadPtr;

/* 如果最高优先级的任务是当前任务则直接返回，不进行任务切換 */
if (OSTCBHighRdyPtr == OSTCBCurPtr)
{
    /* 退出临界区 */
    OS_CRITICAL_EXIT();

    return;
}
/* 退出临界区 */
80  OS_CRITICAL_EXIT(); (5)

81

82 /* 任务切换 */
83  OS_TASK_SW(); (6)
84 }

- 代码清单: 优先级-10 (1): 定义一个局部变量, 用来存 CPU 关中断前的中断状态, 因为接下来查找最高优先级这段代码属于临界短代码, 需要关中断。

- 代码清单: 优先级-10 (2): 进入临界段。

- 代码清单: 优先级-10 (3): 查找最高优先级任务。

- 代码清单: 优先级-10 (4): 判断最高优先级任务是不是当前任务, 如果是则直接返回, 否则将继续往下执行, 最后执行任务切换。

- 代码清单: 优先级-10 (5): 退出临界段。

- 代码清单: 优先级-10 (6): 任务切换。

12.10 修改 OSTimeTick() 函数

OSTimeTick() 函数在 SysTick 中断服务函数中被调用, 是一个周期函数, 具体用于扫描就绪列表 OSRdyList[], 判断任务的延时时间是否到期, 如果到期则将任务在优先级表中对应的位置位, 修改部分的代码见 代码清单: 优先级-11 的加粗部分, 被迭代的代码则通过条件编译屏蔽。

列表 11: 代码清单: 优先级-11OSTimeTick() 函数

```
void OSTimeTick (void)
{
  unsigned int i;
  CPU_SR_ALLOC(); (1)
}
```

(下页继续)
/* 进入临界区 */
OS_CRITICAL_ENTER();

/* 扫描就绪列表中所有任务的 TaskDelayTicks，如果不为 0，则减 1 */
#if 0
for (i=0; i<OS_CFG_PRIO_MAX; i++)
{
    if (OSRdyList[i].HeadPtr->TaskDelayTicks > 0)
    {
        OSRdyList[i].HeadPtr->TaskDelayTicks --;
    }
}
#endif

for (i=0; i<OS_CFG_PRIO_MAX; i++)
{
    if (OSRdyList[i].HeadPtr->TaskDelayTicks > 0)
    {
        OSRdyList[i].HeadPtr->TaskDelayTicks --;
        if (OSRdyList[i].HeadPtr->TaskDelayTicks == 0)
        {
            /* 为 0 则表示延时时间到，让任务就绪 */
            //OS_RdyListInsert (OSRdyList[i].HeadPtr);
            OS_PrioInsert(i);
        }
    }
}

/* 退出临界区 */
OS_CRITICAL_EXIT();

/* 任务调度 */
• 代码清单: 优先级-II (1): 定义一个局部变量, 用来存 CPU 关中断前的中断状态, 因为接下来扫描就绪列表 OSRdyList[]. 这段代码属于临界短代码，需要关中断。

• 代码清单: 优先级-II (2): 进入临界段。

• 代码清单: 优先级-II (3): 扫描就绪列表 OSRdyList[], 判断任务的延时时间是否到期，如果到期则将任务在优先级表中对应的位置位。

• 代码清单: 优先级-II (4): 退出临界段。

### 12.11 main() 函数

main() 函数具体见代码清单: 优先级-II 12，修改部分代码已经加粗显示。

列表 12: 代码清单: 优先级-12 main() 函数

```c
/*
 ******************************************
 * 全局变量
 ******************************************
 */

uint32_t flag1;
uint32_t flag2;
uint32_t flag3;

/*
 ******************************************
 * TCB & STACK & 任务声明
 ******************************************
 */
```
/*
#define TASK1_STK_SIZE 128
#define TASK2_STK_SIZE 128
#define TASK3_STK_SIZE 128

static OS_TCB Task1TCB;
static OS_TCB Task2TCB;
static OS_TCB Task3TCB;

static CPU_STK Task1Stk[TASK1_STK_SIZE];
static CPU_STK Task2Stk[TASK2_STK_SIZE];
static CPU_STK Task3Stk[TASK2_STK_SIZE];

void Task1( void *p_arg );
void Task2( void *p_arg );
void Task3( void *p_arg );

void delay(uint32_t count);

*/
int main(void)
{
    OS_ERR err;

    /* CPU 初始化：1. 初始化时间戳 */
    CPU_Init();

    /* 关闭中断 */
    CPU_IntDis();

    /* 配置 SysTick 10ms 中断一次 */
    OS_CPU_SysTickInit (10);

    /* 初始化相关的全局变量 */
    OSInit(&err); (1)

    /* 创建任务 */
    OSTaskCreate( (OS_TCB*)&Task1TCB,
                   (OS_TASK_PTR )Task1,
                   (void *)0,
                   (OS_PRIO)1, (2)
                   (CPU_STK*)&Task1Stk[0],
                   (CPU_STK_SIZE) TASK1_STK_SIZE,
                   (CPU_STK_SIZE) TASK1_STK_SIZE,
78   (OS_ERR *) &err);
79
80   OSTaskCreate( (OS_TCB*) &Task2TCB,
81       (OS_TASK_PTR ) Task2,
82           (void *) 0,
83       (OS_Prio) 2,(3)
84       (CPU_STK*) &Task2Stk[0],
85       (CPU_STK_SIZE) TASK2_STK_SIZE,
86   (OS_ERR *) &err );
87
88   OSTaskCreate( (OS_TCB*) &Task3TCB,
89       (OS_TASK_PTR ) Task3,
90           (void *) 0,
91       (OS_Prio) 3,(4)
92       (CPU_STK*) &Task3Stk[0],
93       (CPU_STK_SIZE) TASK3_STK_SIZE,
94   (OS_ERR *) &err );
95
96   #if 0
97   /* 将任务加入到就绪列表 */(5)
98   OSRdyList[0].HeadPtr = &Task1TCB;
99   OSRdyList[1].HeadPtr = &Task2TCB;
100  #endif
101
102  /* 启动 OS，将不再返回 */
103  OSStart(&err);
104  }
105
106  /*
107  *******************************************
108  * 函数实现
109  *******************************************
109  */
/* 软件延时 */
void delay (uint32_t count)
{
    for (; count!=0; count--);
}

void Task1 ( void *p_arg )
{
    for ( ;; ) {
        flag1 = 1;
        OSTimeDly(2);
        flag1 = 0;
        OSTimeDly(2);
    }
}

void Task2 ( void *p_arg )
{
    for ( ;; ) {
        flag2 = 1;
        OSTimeDly(2);
        flag2 = 0;
        OSTimeDly(2);
    }
}

void Task3 ( void *p_arg )
{
    for ( ;; ) {
        flag3 = 1;
    }
}
12.12 实验现象

进入软件调试，全速运行程序，从逻辑分析仪中可以看到三个任务的波形是完全同步，就好像 CPU 在同时干三件事情，具体仿真的波形图见图实验现象-宏观。任务启动的启动过程具体见图任务的启动过程-微观，这个启动过程要认真地理解下。

论坛: [https://www.firebbs.cn/](https://www.firebbs.cn/) 天猫: [https://yehuosm.tmall.com](https://yehuosm.tmall.com)

```c
OSTimeDly(2);
flag3 = 0;
OSTimeDly(2);
}
```
图任务的启动过程-微观 是任务 1、2 和 3 刚开始启动时的软件仿真波形图，系统从启动到任务 1 开始运行前花的时间为 TIME1，等于 0.26MS。任务 1 开始运行，然后调用 OSTimeDly(1) 进入延时，随后进行任务切换，切换到任务 2 开始运行，从任务 1 切换到任务 2 花费的时间等于 TIME2-TIME1，等于 0.01MS。任务 2 开始运行，然后调用 OSTimeDly(1) 进入延时，随后进行任务切换，切换到任务 3 开始运行，从任务 2 切换到任务 3 花费的时间等于 TIME3-TIME1，等于 0.01MS。任务 3 开始运行，然后调用 OSTimeDly(1) 进入延时，随后进行任务切换，这个时候我们创建的 3 个任务都处于延时状态，系统就切换到空闲任务，在三个任务延时未到期之前，系统一直都是在运行空闲任务。当第一个 SysTick 中断产生，中断服务函数会调用 OSTimeTick() 函数扫描每个任务的延时是否到期，因是是延时 1 个 SysTick 周期，所以第一个 SysTick 中断产生就意味着延时都到期，任务 1、2 和 3 依次进入就绪态，再次回到任务本身接着运行，将自身的 Flag 清零，然后任务 1、2 和 3 又依次调用 OSTimeDly(1) 进入延时状态，直到下一个 SysTick 中断产生前，系统都处在空闲任务中，一直这样循环下去。

但是，有些同学肯定就会问图实验现象-宏观 中任务 1、2 和 3 的波形图是同步的，而图任务的启动过程-微观 中任务的波形就不同步，有先后顺序？答案是图任务的启动过程-微观 是将两个任务切换花费的时间 0.01ms 进行放大后观察的波形，就好像我们用放大镜看微小的东西一样，如果不用放大镜，在宏观层面观察就是图实验现象-宏观 的实验现象。

论坛: [https://www.firebbs.cn/]  181  天猫: [https://yehuosm.tmall.com]
第 13 章 实现时基列表

从本章开始，我们在 OS 中加入时基列表，时基列表是跟时间相关的，处于延时的任务和等待事件有超时限制的任务都会从就绪列表中移除，然后插入时基列表。时基列表在 OSTimeTick 中更新，如果任务的延时时间结束或者超时到期，就会让任务就绪，从时基列表移除，插入就绪列表。到目前为止，我们在 OS 中只实现了两个列表，一个是就绪列表，一个是本章将要实现的时基列表，在本章之前，任务要么在就绪列表，要么在时基列表。

13.1 实现时基列表

13.1.1 定义时基列表变量

时基列表在代码层面上由全局数组 OSCfg_TickWheel[] 和全局变量 OSTickCtr 构成，一个空的时基列表示意图见图空的时基列表，时基列表的代码实现具体见代码清单: 时基列表-1。

<table>
<thead>
<tr>
<th>[0]</th>
<th>[1]</th>
<th>[2]</th>
</tr>
</thead>
<tbody>
<tr>
<td>NbrEntriesMax=0</td>
<td>NbrEntries=0</td>
<td>FirstPtr ● → 0</td>
</tr>
<tr>
<td>NbrEntriesMax=0</td>
<td>NbrEntries=0</td>
<td>FirstPtr ● → 0</td>
</tr>
<tr>
<td>NbrEntriesMax=0</td>
<td>NbrEntries=0</td>
<td>FirstPtr ● → 0</td>
</tr>
<tr>
<td>NbrEntriesMax=0</td>
<td>NbrEntries=0</td>
<td>FirstPtr ● → 0</td>
</tr>
<tr>
<td>NbrEntriesMax=0</td>
<td>NbrEntries=0</td>
<td>FirstPtr ● → 0</td>
</tr>
<tr>
<td>NbrEntriesMax=0</td>
<td>NbrEntries=0</td>
<td>FirstPtr ● → 0</td>
</tr>
<tr>
<td>NbrEntriesMax=0</td>
<td>NbrEntries=0</td>
<td>FirstPtr ● → 0</td>
</tr>
<tr>
<td>NbrEntriesMax=0</td>
<td>NbrEntries=0</td>
<td>FirstPtr ● → 0</td>
</tr>
<tr>
<td>NbrEntriesMax=0</td>
<td>NbrEntries=0</td>
<td>FirstPtr ● → 0</td>
</tr>
<tr>
<td>NbrEntriesMax=0</td>
<td>NbrEntries=0</td>
<td>FirstPtr ● → 0</td>
</tr>
<tr>
<td>NbrEntriesMax=0</td>
<td>NbrEntries=0</td>
<td>FirstPtr ● → 0</td>
</tr>
</tbody>
</table>

| OSTickCtr |

列表 1: 代码清单: 时基列表-1 时基列表定义

```
/* 时基列表大小，在 os_cfg_app.h 定义 */
#define OS_CFG_TICK_WHEEL_SIZE   17u
```

(下页继续)
/* 在 os_cfg_app.c 定义 */
/* 时基列表 */
(1)(2)
OS_TICK_SPOKE OSCfg_TickWheel[OS_CFG_TICK_WHEEL_SIZE];
/* 时基列表大小 */
OS_OBJ_QTY const OSCfg_TickWheelSize = (OS_OBJ_QTY )OS_CFG_TICK_WHEEL_SIZE;

/* 在 os.h 中声明 */
/* 时基列表 */
extern OS_TICK_SPOKE OSCfg_TickWheel[];
/* 时基列表大小 */
extern OS_OBJ_QTY const OSCfg_TickWheelSize;

/* Tick 计数器，在 os.h 中定义 */
OS_EXT OS_TICK OSTickCtr; (3)

- 代码清单: 时基列表-1（1）: OS_TICK_SPOKE 为时基列表数组 OSCfg_TickWheel[] 的数据类型，在 os.h 文件定义，具体见代码清单: 时基列表-2。

列表2: 代码清单: 时基列表-2OS_TICK_SPOKE 定义

typedef struct os_tick_spoke OS_TICK_SPOKE; (1)

struct os_tick_spoke {
  OS_TCB *FirstPtr; (2)
  OS_OBJ_QTY NbrEntries; (3)
  OS_OBJ_QTY NbrEntriesMax; (4)
};

- 代码清单: 时基列表-2（1）: 在 μC/OS-III 中，内核对象的数据类型都会用大写字母重新定义。

论坛: https://www.firebbs.cn/ 183 天猫: https://yehuosm.tmall.com
• 代码清单: 时基列表-2 (2): 时基列表 OSCfg_TickWheel[] 的每个成员都包含一条单向链表，被插入该条链表的 TCB 会按照延时时间做升序排列。FirstPtr 用于指向这条单向链表的第一个节点。

• 代码清单: 时基列表-2 (3): 时基列表 OSCfg_TickWheel[] 的每个成员都包含一条单向链表，NbrEntries 表示该条单向链表当前有多少个节点。

• 代码清单: 时基列表-2 (4): 时基列表 OSCfg_TickWheel[] 的每个成员都包含一条单向链表，NbrEntriesMax 记录该条单向链表最多的时候有多少个节点，在增加节点的时候会刷新，在删除节点的时候不刷新。

• 代码清单: 时基列表-1 (2): OS_CFG_TICK_WHEEL_SIZE 是一个宏，在 os_cfg_app.h 中定义，用于控制时基列表的大小。OS_CFG_TICK_WHEEL_SIZE 的推荐值为任务数/4，不推荐使用偶数，如果算出来是偶数，则加 1 变成质数，实际上质数是一个很好的选择。

• 代码清单: 时基列表-1 (3): OSTickCtr 为 SysTick 周期计数器，记录系统启动到现在或者从上一次复位到现在经过了多少个 SysTick 周期。

### 13.1.2 修改任务控制块 TCB

时基列表 OSCfg_TickWheel[] 的每个成员都包含一条单向链表，被插入该条链表的 TCB 会按照延时时间做升序排列，为了 TCB 能按照延时时间从小到大串接在一起，需要在 TCB 中加入几个成员，具体见代码清单: 时基列表-3 的加粗部分。
列表 3: 代码清单: 时基列表-3 在 TCB 中加入时基列表

相关字段

```c
struct os_tcb {
    CPU_STK     *StkPtr;
    CPU_STK_SIZE StkSize;

    /* 任务延时周期个数 */
    OS_TICK      TaskDelayTicks;

    /* 任务优先级 */
    OS_PRIOR    Prio;

    /* 就绪列表双向链表的下一个指针 */
    OS_TCB      *NextPtr;

    /* 就绪列表双向链表的前一个指针 */
    OS_TCB      *PrevPtr;

    /* 时基列表相关字段 */
    OS_TCB      *TickNextPtr;  // (1)
}
```
代码清单: 时基列表-3 加粗部分的字段可以配合图时基列表中有两个 TCB 一起理解，这样会比较容易。图时基列表中有两个 TCB 是在时基列表 OSCfg_TickWheel[] 索引 11 这条链表里面插进了两个 TCB，一个需要延时 1 个时钟周期，另外一个需要延时 13 个时钟周期。

- 代码清单: 时基列表-3 (1): TickNextPtr 用于指向链表中的下一个 TCB 节点。
- 代码清单: 时基列表-3 (2): TickPrevPtr 用于指向链表中的上一个 TCB 节点。
- 代码清单: 时基列表-3 (3): TickRemain 用于设置任务还需要等待多少个时钟周期，每到来一个时钟周期，该值会递减。
- 代码清单: 时基列表-3 (4): TickCtrMatch 的值等于是时基计数器 OSTickCtr 的值加上 TickRemain 的值，当 TickCtrMatch 的值等于 OSTickCtr 的值的时候，表示等待到期，TCB 会从链表中删除。
- 代码清单: 时基列表-3 (5): 每个被插入链表的 TCB 都包含一个字段 TickSpokePtr，用于回指到链表的根部。

### 13.1.3 实现时基列表相关函数

时基列表相关函数在 os_tick.c 实现，在 os.h 中声明。如果 os_tick.c 文件是第一次使用，需要自行在文件夹 μC/OS-III 复制 C/OS-III 资源中新建并添加到工程的 μC/OS-III Source 组。
13.1.3.1 OS_TickListInit() 函数

OS_TickListInit() 函数用于初始化时基列表，即将全局变量 OSCfg_TickWheel[] 的数据域全部初始化为 0，一个初始化为 0 的时基列表见图时基列表的数据域全部被初始化为 0。

列表 4: 代码清单: 时基列表-4OS_TickListInit() 函数

```c
/* 初始化时基列表的数据域 */
void OS_TickListInit (void)
{
    OS_TICK_SPOKE_IX   i;
    OS_TICK_SPOKE      *p_spoke;

    for (i = 0u; i < OSCfg_TickWheelSize; i++) {
        p_spoke = (OS_TICK_SPOKE *) &OSCfg_TickWheel[i];
        p_spoke->FirstPtr = (OS_TCB *) 0;
        p_spoke->NbrEntries = (OS_OBJ_QTY ) 0u;
        p_spoke->NbrEntriesMax = (OS_OBJ_QTY ) 0u;
    }
}
```

<table>
<thead>
<tr>
<th>[0]</th>
<th>[1]</th>
<th>[2]</th>
<th>[OS_CFG_TICK_WHEEL_SIZE-1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>NbrEntriesMax=0</td>
<td>NbrEntries=0</td>
<td>FirstPtr ● → 0</td>
<td>NbrEntriesMax=0</td>
</tr>
<tr>
<td>NbrEntriesMax=0</td>
<td>NbrEntries=0</td>
<td>FirstPtr ● → 0</td>
<td>NbrEntriesMax=0</td>
</tr>
<tr>
<td>NbrEntriesMax=0</td>
<td>NbrEntries=0</td>
<td>FirstPtr ● → 0</td>
<td>NbrEntriesMax=0</td>
</tr>
<tr>
<td>NbrEntriesMax=0</td>
<td>NbrEntries=0</td>
<td>FirstPtr ● → 0</td>
<td>NbrEntriesMax=0</td>
</tr>
</tbody>
</table>

论坛: https://www.firebbs.cn/  187  天猫: https://yehuosm.tmall.com
13.1.3.2 OS_TickListInsert() 函数

OS_TickListInsert() 函数用于往时基列表中插入一个任务 TCB，具体实现见代码清单: 时基列表-5。代码清单: 时基列表-5 可配和图时基列表中有三个 TCB 一起阅读，这样理解起来会容易很多。

列表 5: 代码清单: 时基列表-5OS_TickListInsert() 函数

```c
/* 将一个任务插入时基列表，根据延时时间的大小升序排列 */
void OS_TickListInsert(OS_TCB *p_tcb, OS_TICK time) {
    OS_TICK_SPOKE_IX spoke;
    OS_TICK_SPOKE *p_spoke;
    OS_TCB *p_tcb0;
    OS_TCB *p_tcb1;

    p_tcb->TickCtrMatch = OSTickCtr + time;  // (1)
    p_tcb->TickRemain = time;  // (2)

    spoke = (OS_TICK_SPOKE_IX)(p_tcb->TickCtrMatch % OSCfg_TickWheelSize);  // (3)
    p_spoke = &OSCfg_TickWheel[spoke];  // (4)

    /* 插入 OSCfg_TickWheel[spoke] 的第一个节点 */
}
```

论坛: https://www.firebbs.cn/
天猫: https://yehuosm.tmall.com
if (p_spoke->NbrEntries == (OS_OBJ_QTY)0u) (5)
{
    p_tcb->TickNextPtr = (OS_TCB *)0;
    p_tcb->TickPrevPtr = (OS_TCB *)0;
    p_spoke->FirstPtr = p_tcb;
    p_spoke->NbrEntries = (OS_OBJ_QTY)1u;
}
/* 如果插入的第一个节点，则按照 TickRemain 大小升序排列 */
else (6)
{
    /* 获取第一个节点指针 */
    p_tcb1 = p_spoke->FirstPtr;
    while (p_tcb1 != (OS_TCB *)0)
    {
        /* 计算比较节点的剩余时间 */
        p_tcb1->TickRemain = p_tcb1->TickCtrMatch - OSTickCtr;

        /* 插入比较节点的后面 */
        if (p_tcb->TickRemain > p_tcb1->TickRemain)
        {
            if (p_tcb1->TickNextPtr != (OS_TCB *)0)
            {
                /* 寻找下一个比较节点 */
                p_tcb1 = p_tcb1->TickNextPtr;
            }
        else
        { /* 在最后一个节点插入 */
            p_tcb->TickNextPtr = (OS_TCB *)0;
            p_tcb->TickPrevPtr = p_tcb1;
            p_tcb1->TickNextPtr = p_tcb;
            p_tcb1 = (OS_TCB *)0; (7)
        }
    }
}  
/* 插入比较节点的前面 */  
else  
{  
/* 在第一个节点插入 */  
if (p_tcb1->TickPrevPtr == (OS_TCB *)0) {  
p_tcb->TickPrevPtr = (OS_TCB *)0;  
p_tcb->TickNextPtr = p_tcb1;  
p_tcb1->TickPrevPtr = p_tcb;  
p_spoke->FirstPtr = p_tcb;  
}  
else  
{  
/* 插入两个节点之间 */  
p_tcb0 = p_tcb1->TickPrevPtr;  
p_tcb->TickPrevPtr = p_tcb0;  
p_tcb->TickNextPtr = p_tcb1;  
p_tcb0->TickNextPtr = p_tcb;  
p_tcb1->TickPrevPtr = p_tcb;  
}  
/* 跳出 while 循环 */  
p_tcb1 = (OS_TCB *)0;  
}  
}  
/* 节点成功插入 */  
p_spoke->NbrEntries++;  
}  

/* 刷新 NbrEntriesMax 的值 */  
if (p_spoke->NbrEntriesMax < p_spoke->NbrEntries)  
{  
}
p_spoke->NbrEntriesMax = p_spoke->NbrEntries;
}

/* 任务 TCB 中的 TickSpokePtr 回指根节点 */
p_tcb->TickSpokePtr = p_spoke;  // (11)

论坛: https://www.firebbs.cn/  191  天猫: https://yehuosm.tmall.com
值做升序排列，插入的时候分三种情况，第一是在最后一个节点之间插入，第二是在第一个节点插入，第三是在两个节点之间插入。

- 代码清单: 时基列表-5 (7) (8): 节点成功插入 p_tcb1 指针，跳出 while 循环
- 代码清单: 时基列表-5 (9): 节点成功插入，记录当前链表节点个数的计数器 NbrEntries 加一。
- 代码清单: 时基列表-5 (10): 刷新 NbrEntriesMax 的值，NbrEntriesMax 用于记录当前链表曾经最多有多少个节点，只有在增加节点的时候才刷新，在删除节点的时候是不刷新的。
- 代码清单: 时基列表-5 (11): 任务 TCB 被成功插入链表，TCB 中的 TickSpokePtr 回指所在链表的根指针。

### 13.1.3.3 OS_TickListRemove() 函数

OS_TickListRemove() 用于从时基列表删除一个指定的 TCB 节点，具体实现见。代码清单: 时基列表-6

<table>
<thead>
<tr>
<th>1</th>
<th>/* 从时基列表中移除一个任务 */</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td><strong>void</strong> <strong>OS_TickListRemove</strong>( <strong>OS_TCB</strong> <em>p_tcb</em>)</td>
</tr>
</tbody>
</table>
| 3 | {
| 4 | **OS_TICK_SPOKE** *p_spoke*;
| 5 | **OS_TCB** *p_tcb1*;
| 6 | **OS_TCB** *p_tcb2*;
| 7 | |
| 8 | /* 获取任务 TCB 所在链表的根指针 */ |
| 9 | p_spoke = p_tcb->TickSpokePtr; (1) |
| 10 | |
| 11 | /* 确保任务在链表中 */ |
| 12 | **if** (p_spoke != (**OS_TICK_SPOKE**) 0) |
| 13 | {
| 14 | /* 将剩余时问清零 */ |

(下页继续)
p_tcb->TickRemain = (OS_TICK)0u;

/* 要移除的刚好是第一个节点 */
if (p_spoke->FirstPtr == p_tcb) (2)
{
    /* 更新第一个节点，原来的第一个节点需要被移除 */
    p_tcb1 = (OS_TCB *)p_tcb->TickNextPtr;
    p_spoke->FirstPtr = p_tcb1;
    if (p_tcb1 != (OS_TCB *)0)
    {
        p_tcb1->TickPrevPtr = (OS_TCB *)0;
    }
}
/* 要移除的不是第一个节点 */(3)
else
{
    /* 保存要移除的节点的前后节点的指针 */
    p_tcb1 = p_tcb->TickPrevPtr;
    p_tcb2 = p_tcb->TickNextPtr;

    /* 节点移除，将节点前后的两个节点连接在一起 */
    p_tcb1->TickNextPtr = p_tcb2;
    if (p_tcb2 != (OS_TCB *)0)
    {
        p_tcb2->TickPrevPtr = p_tcb1;
    }
}

/* 复位任务 TCB 中时基列表相关的字段成员 */(4)
p_tcb->TickNextPtr = (OS_TCB *)0;
p_tcb->TickPrevPtr = (OS_TCB *)0;
p_tcb->TickSpokePtr = (OS_TICK_SPOKE *)0;
void OS_TickListUpdate ( void )
{
    OS_TICK_SPOKE_IX spoke;
    OS_TICK_SPOKE *p_spoke;
    OS_TCB *p_tcb;
    OS_TCB *p_tcb_next;
    CPU_BOOLEAN done;

    p_tcb->TickCtrlMatch = (OS_TICK) 0;
    /* 节点减 1 */
    p_spoke->NbrEntries--; (5)
}

- 代码清单: 时基列表-6 (1): 获取任务 TCB 所在链表的根指针。
- 代码清单: 时基列表-6 (2): 要删除的节点是链表的第一个节点, 这个操作很好处理, 只需更新下第一个节点即可。
- 代码清单: 时基列表-6 (3): 要删除的节点不是链表的第一个节点, 则先保存要删除的节点的前后节点, 然后把这前后两个节点相连即可。
- 代码清单: 时基列表-6 (4): 复位任务 TCB 中时基列表相关的字段成员。
- 代码清单: 时基列表-6 (5): 节点删除成功, 链表中的节点计数器 NbrEntries 减一。

### 13.1.3.4 OS_TickListUpdate() 函数

OS_TickListUpdate() 在每个 SysTick 周期到来时在 OSTimeTick() 被调用, 用于更新时基计数器 OSTickCtr, 扫描时基列表中的任务延时是否到期, 具体实现见代码清单: 时基列表-7。

**列表 7: 代码清单: 时基列表-7OS_TickListUpdate() 函数**

```c
void OS_TickListUpdate ( void )
{
    OS_TICK_SPOKE_IX spoke;
    OS_TICK_SPOKE *p_spoke;
    OS_TCB *p_tcb;
    OS_TCB *p_tcb_next;
    CPU_BOOLEAN done;
}
```
CPU_SR_ALLOC();

/* 进入临界段 */
OS_CRITICAL_ENTER();

/* 时基计数器 ++ */
OSTickCtr++; (1)

spoke = (OS_TICK_SPOKE_IX)(OSTickCtr % OSCfg_TickWheelSize); (2)
p_spoke = &OSCfg_TickWheel[spoke];

p_tcb = p_spoke->FirstPtr;
done = DEF_FALSE;

while (done == DEF_FALSE)
{
    if (p_tcb != (OS_TCB *)0) (3)
    {
        p_tcb_next = p_tcb->TickNextPtr;

        p_tcb->TickRemain = p_tcb->TickCtrMatch - OSTickCtr; (4)

        /* 节点延时时间到 */
        if (OSTickCtr == p_tcb->TickCtrMatch) (5)
        {
            /* 让任务就绪 */
            OS_TaskRdy(p_tcb);
        }
        else (6)
        {
            /* 如果第一个节点延时期未满，则退出 while 循环
                因为链表是根据升序排列的，第一个节点延时期未满，那后面的肯定未满 */
        }
    }
}
done = DEF_TRUE;
}

/* 如果第一个节点延时期满，则继续遍历链表，看看还有没有延时期满的任务
如果有，则让它就绪 */
p_tcb = p_tcb_next;(7)
}
else
{
    done = DEF_TRUE;(8)
}
}

/* 退出临界段 */
OS_CRITICAL_EXIT();

- 代码清单: 时基列表-7 (1): 每到来一个 SysTick 时钟周期，时基计数器 OSTickCtr 都要加一操作。
- 代码清单: 时基列表-7 (2): 计算要扫描的时基列表的索引，每次只扫描一条链表。时基列表里面有可能有多条链表，为啥只扫描其中一条链表就可以？因为任务在插入时基列表的时候，插入的索引值 spoke_insert 是通过 TickCtrMatch 对 OSCfg_TickWheelSize 求余得出，现在需要扫描的索引值 spoke_update 是通过 OSTickCtr 对 OSCfg_TickWheelSize 求余得出，TickCtrMatch 的值等于 OSTickCtr 加上 TickRemain，只有在经过 TickRemain 个时钟周期后， spoke_update 的值才有可能等于 spoke_insert。如果算出的 spoke_update 小于 spoke_insert，且 OSCfg_TickWheel[spoke_update] 下的链表的任务没有到期，那后面的肯定都没有到期，不用继续扫描。

举例，在时基列表中有三个 TCB，时基列表 OSCfg_TickWheel[] 的大小 OSCfg_TickWheelSize 等于 12，当前时基计数器 OSTickCtr 的值为 7，有三个任务分别需要延时 TickTemain=16，TickTemain=28 和 TickTemain=40 个时钟周期，三个任务的 TickRemain 加上 OSTickCtr 可分别得出它们的 TickCtrMatch 等于 23，35 和 47，这三个任务的 TickCtrMatch 对 OSCfg_TickWheelSize 求余

- 代码清单: 时基列表-7 (3): 判断链表是否为空，为空则跳转到第（8）步骤。
- 代码清单: 时基列表-7 (4): 链表不为空，递减第一个节点的 TickRemain。
- 代码清单: 时基列表-7 (5): 判断第一个节点的延时时间是否到，如果到期，让任务就绪，即将任务从时基列表删除，插入就绪列表，这两步由函数 OS_TaskRdy() 来完成，该函数在 os_core.c 中定义，具体实现见代码清单: 时基列表-8。
列表 8: 代码清单: 时基列表-8OS_TaskRdy() 函数

```c
void OS_TaskRdy(OS_TCB *p_tcb)
{
    /* 从时基列表删除 */
    OS_TickListRemove(p_tcb);

    /* 插入就绪列表 */
    OS_RdyListInsert(p_tcb);
}
```

- **代码清单: 时基列表-7 (6):** 如果第一个节点延时期未满，则退出 while 循环，因为链表是根据升序排列的，第一个节点延时期未满，那后面的肯定未满。

- **代码清单: 时基列表-7 (7):** 如果第一个节点延时到期，则继续判断下一个节点延时是否到期。

- **代码清单: 时基列表-7 (8):** 链表为空，退出扫描，因为其他还没到期。

论坛: https://www.firebbs.cn/  
天猫: https://yehuosm.tmall.com
13.2 修改 OSTimeDly() 函数

加入时基列表之后，OSTimeDly() 函数需要被修改，具体见代码清单: 时基列表-9 的加粗部分，被迭代的代码已经用条件编译屏蔽。

列表 9: 代码清单: 时基列表-9OSTimeDly() 函数

```c
void OSTimeDly(OS_TICK dly)
{
    CPU_SR_ALLOC();

    /* 进入临界区 */
    OS_CRITICAL_ENTER();
#if 0
    /* 设置延时时间 */
    OSTCBCurPtr->TaskDelayTicks = dly;

    /* 从就绪列表中移除 */
    //OS_RdyListRemove(OSTCBCurPtr);
    OS_PrioRemove(OSTCBCurPtr->Prio);
#endif

    /* 插入时基列表 */
    OS_TickListInsert(OSTCBCurPtr, dly);

    /* 从就绪列表移除 */
    OS_RdyListRemove(OSTCBCurPtr);

    /* 退出临界区 */
    OS_CRITICAL_EXIT();

    /* 任务调度 */
    OSSched();
}
```

(下页继续)
### 13.3 修改 OSTimeTick() 函数

加入时基列表之后，OSTimeTick() 函数需要被修改，具体见代码清单: 时基列表-10 的加粗部分，被迭代的代码已经用条件编译屏蔽。

<table>
<thead>
<tr>
<th>代码清单: 时基列表-10OSTimeTick() 函数</th>
</tr>
</thead>
<tbody>
<tr>
<td>void OSTimeTick (void)</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>#if 0</td>
</tr>
<tr>
<td>unsigned int i;</td>
</tr>
<tr>
<td>CPU_SR_ALLOC();</td>
</tr>
<tr>
<td>/* 进入临界区 */</td>
</tr>
<tr>
<td>OS_CRITICAL_ENTER();</td>
</tr>
<tr>
<td>for (i=0; i&lt;OS_CFG_PRIO_MAX; i++)</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>if (OSRdyList[i].HeadPtr-&gt;TaskDelayTicks &gt; 0)</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>OSRdyList[i].HeadPtr-&gt;TaskDelayTicks -=;</td>
</tr>
<tr>
<td>if (OSRdyList[i].HeadPtr-&gt;TaskDelayTicks == 0)</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>/* 为 0 则表示延时时间到，让任务就绪 */</td>
</tr>
<tr>
<td>//OS_RdyListInsert (OSRdyList[i].HeadPtr);</td>
</tr>
<tr>
<td>OS_PrioInsert (i);</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>
13.4 main 函数

main() 函数同上一章一样。

13.5 实验现象

实验现象同上一章一样，实验现象虽然一样，但是任务在就是延时状态时，任务的 TCB 不再继续放在就绪列表，而是放在了时基列表中。
第 14 章 实现时间片

本章开始，我们让 OS 支持同一个优先级下可以有多个任务的功能，这些任务可以分配不同的时间片，当任务时间片用完的时候，任务会从链表的头部移动到尾部，让下一个任务共享时间片，以此循环。

14.1 实现时间片

14.1.1 修改任务 TCB

为了实现时间片功能，我们需要先在任务控制块 TCB 中添加两个时间片相关的变量，具体见代码清单: 阻塞延时-1 的加粗部分。

列表 1: 代码清单: 阻塞延时-1 在 TCB 中添加时间片相关的变量

```c
struct os_tcb {
    CPU_STK *StkPtr;
    CPU_STK_SIZE StkSize;

    /* 任务延时周期数 */
    OS_TICK TaskDelayTicks;

    /* 任务优先级 */
    OS_PRIO Prio;

    /* 就绪列表双向链表的下一个指针 */
    OS_TCB *NextPtr;
    /* 就绪列表双向链表的前一个指针 */
}
```

(下页继续)
OS_TCB *PrevPtr;

/* 时基列表相关字段 */
OS_TCB *TickNextPtr;
OS_TCB *TickPrevPtr;
OS_TICK_SPOKE *TickSpokePtr;

OS_TICK TickCtrMatch;
OS_TICK TickRemain;

/* 时间片相关字段 */
OS_TICK TimeQuanta; (1)
OS_TICK TimeQuantaCtr; (2)

};

- 代码清单: 阻塞延时-J (1): TimeQuanta 表示任务需要多少个时间片，单位为系统时钟周期 Tick。

- 代码清单: 阻塞延时-J (2): TimeQuantaCtr 表示任务还剩下多少个时间片，每到来一个系统时钟周期，TimeQuantaCtr 会减一，当 TimeQuantaCtr 等于零的时候，表示时间片用完，任务的 TCB 会从就绪列表链表的头部移动到尾部，好让下一个任务共享时间片。
14.1.2 实现时间片调度函数

14.1.2.1 OS_SchedRoundRobin() 函数

时间片调度函数 OS_SchedRoundRobin() 在 os_core.c 中实现，在 OSTimeTick() 调用，具体见代码清单: 阻塞延时-2。在阅读代码清单: 阻塞延时-2 的时候，可配图时间片调度函数讲解配套一起理解，该图画的是在一个就绪链表中，有三个任务就绪，其中在优先级 2 下面有两个任务，均分配了两个时间片，其中任务 3 的时间片已用完，则位于链表的末尾；任务 2 的时间片还剩一个，则位于链表的头部。当下一个时钟周期到来的时候，任务 2 的时间片将耗完，相应的 TimeQuantaCtr 会递减为 0，任务 2 的 TCB 会被移动到链表的末尾，任务 3 则被成为链表的头部，然后重置任务
3 的时间片计数器 TimeQuantaCtr 的值为 2，重新享有时间片。

列表 2: 代码清单：阻塞延时-2 时间片调度函数

```c
#if OS_CFG_SCHED_ROUND_ROBIN_EN > 0u(1)
void OS_SchedRoundRobin(OS_RDY_LIST *p_rdy_list)
{
    OS_TCB  *p_tcb;
    CPU_SR_ALLOC();

    /* 进入临界段 */
    CPU_CRITICAL_ENTER();
}
```

(下页继续)
p_tcb = p_rdy_list->HeadPtr; (2)

/* 如果 TCB 节点为空，则退出 */
if (p_tcb == (OS_TCB *)0) (3)
{
    CPU_CRITICAL_EXIT();
    return;
}

/* 如果是空闲任务，也退出 */
if (p_tcb == &OSIdleTaskTCB) ((4)
    CPU_CRITICAL_EXIT();
    return;
}

/* 时间片自减 */
if (p_tcb->TimeQuantaCtr > (OS_TICK)0) ((5)
    p_tcb->TimeQuantaCtr--;
}

/* 时间片没有用完，则退出 */
if (p_tcb->TimeQuantaCtr > (OS_TICK)0) ((6)
    CPU_CRITICAL_EXIT();
    return;
}

/* 如果当前优先级只有一个任务，则退出 */
if (p_rdy_list->NbrEntries < (OS_OBJ_QTY)2) ((7)
    CPU_CRITICAL_EXIT();
    return;
)
/* 时间片耗完，将任务放到链表的最后一个节点 */
OS_RdyListMoveHeadToTail(p_rdy_list); (8)

/* 重新获取任务节点 */
p_tcb = p_rdy_list->HeadPtr; (9)
/* 重载默认的时间片计数值 */
p_tcb->TimeQuantaCtr = p_tcb->TimeQuanta;

/* 退出临界段 */
CPU_CRITICAL_EXIT();
}
#endif /* OS_CFG_SCHED_ROUND_ROBIN_EN > 0u */

• 代码清单: 阻塞延时-2 (1) : 时间片是一个可选的功能，是否选择由
  OS_CFG_SCHED_ROUND_ROBIN_EN 控制，该宏在 os_cfg.h 定义。
  
• 代码清单: 阻塞延时-2 (2) : 获取链表的第一个节点。
  
• 代码清单: 阻塞延时-2 (3) : 如果节点为空，则退出。
  
• 代码清单: 阻塞延时-2 (4) : 如果节点不为空，看看是否是空闲任务，如果是则退出。
  
• 代码清单: 阻塞延时-2 (5) : 如果不是空闲任务，则时间片计数器 TimeQuantaCtr 减一操作。
  
• 代码清单: 阻塞延时-2 (6) : 时间片计数器 TimeQuantaCtr 递减之后，则判断下时间片是否
  用完，如果没有用完，则退出。
  
• 代码清单: 阻塞延时-2 (7) : 如果时间片用完，则判断性该优先级下有多少个任务，如果是
  一个，就退出。
  
• 代码清单: 阻塞延时-2 (8) : 时间片用完，如果该优先级下有两个以上任务，则将刚刚耗完
  时间片的节点移到链表的末尾，此时位于末尾的任务的 TCB 字段中的 TimeQuantaCtr 是等
  于 0 的，只有等它下一次运行的时候值才会重置为 TimeQuanta。
  
• 代码清单: 阻塞延时-2 (9) : 重新获取链表的第一个节点，重置时间片计数器 TimeQuantaCtr
的值等于 TimeQuanta，任务重新享有时间片。

### 14.2 修改 OSTimeTick() 函数

任务的时间片的单位在每个系统时钟周期到来的时候被更新，时间片调度函数则由时基周期处理函数 OSTimeTick() 调用，只需要在更新时基列表之后调用时间片调度函数即可，具体修改见代码清单: 阻塞延时-3 的加粗部分。

列表 3: 代码清单: 阻塞延时-3OSTimeTick() 函数

```c
void OSTimeTick (void) {
    /* 更新时基列表 */
    OS_TickListUpdate();

#if OS_CFG_SCHED_ROUND_ROBIN_EN > 0u
    /* 时间片调度 */
    OS_SchedRoundRobin(&OSRdyList[OSPrioCur]);
#endif

    /* 任务调度 */
    OSSched();
}
```

### 14.3 修改 OSTaskCreate() 函数

任务的时间片在函数创建的时候被指定，具体修改见代码清单: 阻塞延时-4 中的加粗部分。
列表 4: 代码清单: 阻塞延时-4OSTaskCreate() 函数

```c
void OSTaskCreate (OS_TCB *p_tcb,
    OS_TASK_PTR p_task,
    void *p_arg,
    OS_PRIO prio,
    CPU_STK *p_stk_base,
    CPU_STK_SIZE stk_size,
    OS_TICK time_quanta,(1)
    OS_ERR *p_err)
{
    CPU_STK *p_sp;
    CPU_SR_ALLOC();

    /* 初始化 TCB 为默认值 */
    OS_TaskInitTCB(p_tcb);

    /* 初始化栈 */
    p_sp = OSTaskStkInit( p_task,
        p_arg,
        p_stk_base,
        stk_size );

    p_tcb->Prio = prio;

    p_tcb->StkPtr = p_sp;
    p_tcb->StkSize = stk_size;

    /* 时间片相关初始化 */
    p_tcb->TimeQuanta = time_quanta;(2)
    #if OS_CFG_SCHED_ROUND_ROBIN_EN > 0u
    p_tcb->TimeQuantaCtr = time_quanta;(3)
    #endif
}
```

(下页继续)
```
#include "ucos_iii.h"

/* 阻塞延时 */
void OS_Delay(unsigned long time)
{
    OS_ENTER_CRITICAL();
    ucos_3陷入阻塞模式，等待延时。
    将运行状态设置为睡眠模式。
    *p_err = OS_ERR_NONE;
    OS_EXIT_CRITICAL();
}
```

- **代码清单: 阻塞延时-4** (1): 时间片在任务创建的时候由函数形参 time_quanta 指定。
- **代码清单: 阻塞延时-4** (2): 初始化任务 TCB 字段的时间片变量 TimeQuanta，该变量表示
  任务能享有的最大的时间片是多少，该值一旦初始化后就不会变，除非认为修改。
- **代码清单: 阻塞延时-4** (3): 初始化时间片计数器 TimeQuantaCtr 的值等于 TimeQuanta，每
  经过一个系统时钟周期，该值会递减，如果该值为 0，则表示时间片耗完。

### 14.4 修改 OS_IdleTaskInit() 函数

因为在 OS_IdleTaskInit() 函数中创建了空闲任务，所以该函数也需要修改，只需在空闲任务创建
函数中，添加一个时间片的形参就可，时间片我们分配为 0，因为在空闲任务优先级下只有空闲
任务一个任务，没有其他的任务，具体修改见代码清单: 阻塞延时-5 的加粗部分。

论坛: https://www.firebbs.cn/ 天猫: https://yehuosm.tmall.com
14.5 main() 函数

这里，我们创建任务 1、2 和 3，其中任务 1 的优先级为 1，时间片为 0，任务 2 和任务 3 的优先级相同，均为 2，均分配两个两个时间片，当任务创建完毕后，就绪列表的分布图具体见图 main 函数代码讲解配图。
int main(void)
{
    OS_ERR err;

    /* CPU 初始化: 1. 初始化时间戳 */
    CPU_Init();

    /* 关闭中断 */
    CPU_IntDis();

    (下页继续)
/* 配置 SysTick 10ms 中断一次 */
OS_CPU_SysTickInit (10);

/* 初始化相关的全局变量 */
OSInit(&err);

/* 创建任务 */
OSTaskCreate( (OS_TCB *)&Task1TCB,
(OS_TASK_PTR )Task1,
(void *)0,
(OS_PRIO )1,(1)
(CPU_STK *)&Task1Stk[0],
(CPU_STK_SIZE )TASK1_STK_SIZE,
(OS_TICK )0,(1)
(OS_ERR )&err );

OSTaskCreate( (OS_TCB *)&Task2TCB,
(OS_TASK_PTR )Task2,
(void *)0,
(OS_PRIO )2,(2)
(CPU_STK *)&Task2Stk[0],
(CPU_STK_SIZE )TASK2_STK_SIZE,
(OS_TICK )1,(2)
(OS_ERR )&err );

OSTaskCreate( (OS_TCB *)&Task3TCB,
(OS_TASK_PTR )Task3,
(void *)0,
(OS_PRIO )2,(2)
(CPU_STK *)&Task3Stk[0],
(CPU_STK_SIZE )TASK3_STK_SIZE,
(OS_TICK)1,(2)
(OS_ERR)*&err);

/* 启动 OS，将不再返回 */
OSStart(&err);
}

void Task1(void *p_arg)
{
    for (;; ) {
        flag1 = 1;
        OSTimeDly(2);
        flag1 = 0;
        OSTimeDly(2);
    }
}

void Task2(void *p_arg)
{
    for (;; ) {
        flag2 = 1;
        //OSTimeDly(1); (3)
        delay(0xff);
        flag2 = 0;
        //OSTimeDly(1);
        delay(0xff);
    }
}

void Task3(void *p_arg)
{
    for (;; ) {

flag3 = 1;

//OSTimeDly(1); (3)
delay(0xff);
flag3 = 0;

//OSTimeDly(1);
delay(0xff);
}
}

- 代码清单: 阻塞延时-6 (1): 任务 1 的优先级为 1，时间片为 0。当同一个优先级下有多个任务的时候才需要时间片功能。
- 代码清单: 阻塞延时-6 (2): 任务 2 和任务 3 的优先级相同，均为 2，且分配相同的时间片，时间片也可以不同。
- 代码清单: 阻塞延时-6 (3): 因为任务 2 和 3 的优先级相同，分配了相同的时间片，也可以分配不同的时间片，并把阻塞延时换成软件延时，不管是阻塞延时还是软件延时，延时的时间都必须小于时间片，因为相同优先级的任务在运行的时候最大不能超过时间片的时间。

### 14.6 实验现象

进入软件调试，单击全速运行按钮就可看到实验波形，具体见图实验现象。在图中我们可以看到，在任务 1 的 flag1 置 1 和置 0 的两个时间片内，任务 2 和 3 都各运行了一次，运行的时间均为 1 个时间片，在这 1 个时间片内任务 2 和 3 的 flag 变量翻转了好多次，即任务运行了好多次。

[论坛: https://www.firebbs.cn/  番茄: https://yehuosm.tmall.com]
第 15 章 任务的挂起和恢复

本章开始，我们让 OS 的任务支持挂起和恢复的功能，挂起就相当于暂停，暂停后任务从就绪列表中移除，恢复即重新将任务插入就绪列表。一个任务挂起多少次就要被恢复多少次才能重新运行。

### 15.1 实现任务的挂起和恢复

#### 15.1.1 定义任务的状态

在任务实现挂起和恢复的时候，要根据任务的状态来操作，任务的状态不同，操作也不同，有关任务状态的宏定义在 os.h 中实现，总共有 9 种状态，具体定义见代码清单: 挂起和恢复-1。

列表 1: 代码清单: 挂起和恢复-1 定义任务的状态

```c
/* -------- 任务的状态 --------*/
#define OS_TASK_STATE_BIT_DLY (OS_STATE)(0x01u) /* ------
   挂起位 */

#define OS_TASK_STATE_BIT_PEND (OS_STATE)(0x02u) /* |------
   等待位 */

#define OS_TASK_STATE_BIT_SUSPENDED (OS_STATE)(0x04u) /* | |------
   延时/超时位 */

#define OS_TASK_STATE_RDY (OS_STATE)(0u) /* 0 0 0 就绪 */

#define OS_TASK_STATE_DLY (OS_STATE)(1u) /* 0 0 1 延时或者超时 */
```

(下页继续)
10.1 define OS_TASK_STATE_PEND (OS_STATE) ( 2u) /* 0 1 0 等
    待 */
11.1 define OS_TASK_STATE_PEND_TIMEOUT (OS_STATE) ( 3u) /* 0 1 1 等
    待 + 超时 */
12.1 define OS_TASK_STATE_SUSPENDED (OS_STATE) ( 4u) /* 1 0 0 挂
    起 */
13.1 define OS_TASK_STATE_DLY_SUSPENDED (OS_STATE) ( 5u) /* 1 0 1 挂
    起 + 延时或者超时 */
14.1 define OS_TASK_STATE_PEND_SUSPENDED (OS_STATE) ( 6u) /* 1 1 0 挂
    起 + 等待 */
15.1 define OS_TASK_STATE_PEND_TIMEOUT_SUSPENDED (OS_STATE) ( 7u) /* 1 1 1 挂
    起 + 等待 + 超时 */
16.1 define OS_TASK_STATE_DEL (OS_STATE) (255u)

15.1.2 修改任务控制块 TCB

为了实现任务的挂起和恢复，需要先在任务控制中 TCB 中添加任务的状态 TaskState 和任务挂起计数器 SusPendCtr 这两个成员，具体见代码清单: 挂起和恢复-2 的加粗部分。

列表 2: 代码清单: 挂起和恢复-2 任务 TCB

```
struct os_tcb {
    CPU_STK *StkPtr;
    CPU_STK_SIZE StkSize;

    /* 任务延时周期个数 */
    OS_TICK TaskDelayTicks;

    /* 任务优先级 */
    OS_PRIO Prio;
}
```
11 /* 就绪列表双向链表的下一个指针 */
12 OS_TCB *NextPtr;
13 /* 就绪列表双向链表的前一个指针 */
14 OS_TCB *PrevPtr;
15
16 /* 时基列表相关字段 */
17 OS_TCB *TickNextPtr;
18 OS_TCB *TickPrevPtr;
19 OS_TICK_SPOKE *TickSpokePtr;
20
21 OS_TICK TickCtrMatch;
22 OS_TICK TickRemain;
23
24 /* 时间片相关字段 */
25 OS_TICK TimeQuanta;
26 OS_TICK TimeQuantaCtr;
27
28 OS_STATE TaskState; (1)
29
30 #if OS_CFG_TASK_SUSPEND_EN > 0u (2)
31 /* 任务挂起函数 OSTaskSuspend() 计数器 */
32 OS_NESTING_CTR SuspendCtr; (3)
33 #endif
34
35
• 代码清单: 挂起和恢复-2 (1): TaskState 用来表示任务的状态，在本章之前，任务出现了两种状态，一是任务刚刚创建好的时候，处于就绪态，调用阻塞延时函数的时候处于延时态。本章要实现的是任务的挂起态，再往后的章节中还会有等待态，超时态，删除态等。TaskState 能够取的值具体见代码清单: 挂起和恢复-1。
• 代码清单: 挂起和恢复-2 (2): 任务挂起功能是可选的，通过宏 OS_CFG_TASK_SUSPEND_EN 来控制，该宏在 os_cfg.h 文件中定义。
• 代码清单: 挂起和恢复-2 (3): 任务挂起计数器，任务每被挂起一次，SuspendCtr 递增一次，一个任务挂起多少次就要被恢复多少次才能重新运行。

15.1.3 编写任务挂起和恢复函数

15.1.3.1 OSTaskSuspend() 函数

OSTaskSuspend() 函数

```
#if OS_CFG_TASK_SUSPEND_EN > 0u

void OSTaskSuspend (OS_TCB *p_tcb, 
                    OS_ERR   *p_err)
{
    CPU_SR_ALLOC();

    #if 0/* 屏蔽开始 */
    #ifdef OS_SAFETY_CRITICAL
        /* 安全检查，OS_SAFETY_CRITICAL_EXCEPTION() 函数需要用户自行编写 */
        if (p_err == (OS_ERR *)0)
        {
            OS_SAFETY_CRITICAL_EXCEPTION();
            return;
        }
    #endif
    #endif

#if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u
    /* 不能在 ISR 程序中调用该函数 */
    if (OSIntNestingCtr > (OS_NESTING_CTR)0)
    {
```

(下页继续)
*p_err = OS_ERR_TASK_SUSPEND_ISR;
   return;
}
#endif

/* 不能挂起空闲任务 */
if (p_tcb == &OSIdleTaskTCB)
{
   *p_err = OS_ERR_TASK_SUSPEND_IDLE;
   return;
}

#if OS_CFG_ISR_POST_DEFERRED_EN > 0u
   /* 不能挂起中断处理任务 */
   if (p_tcb == &OSIntQTaskTCB)
   {
      *p_err = OS_ERR_TASK_SUSPEND_INT_HANDLER;
      return;
   }
#endif

#endif/* 屏蔽结束 */

CPU_CRITICAL_ENTER();

/* 是否挂起自己 */
if (p_tcb == (OS_TCB *)0) {
   p_tcb = OSTCBCurPtr;
}

if (p_tcb == OSTCBCurPtr) {
(下页继续)
/* 如果调度器锁住则不能挂起自己 */
if (OSSchedLockNestingCtr > (OS_NESTING_CTR)0) {
    CPU_CRITICAL_EXIT();
    *p_err = OS_ERR_SCHED_LOCKED;
    return;
}
*p_err = OS_ERR_NONE;

/* 根据任务的状态来决定挂起的动作 */(4)
switch (p_tcb->TaskState)
{
    case OS_TASK_STATE_RDY: (5)
        OS_CRITICAL_ENTER_CPU_CRITICAL_EXIT();
        p_tcb->TaskState = OS_TASK_STATE_SUSPENDED;
        p_tcb->SuspendCtr = (OS_NESTING_CTR)1;
        OS_RdyListRemove(p_tcb);
        OS_CRITICAL_EXIT_NO_SCHED();
        break;

    case OS_TASK_STATE_DLY: (6)
        p_tcb->TaskState = OS_TASK_STATE_DLY_SUSPENDED;
        p_tcb->SuspendCtr = (OS_NESTING_CTR)1;
        CPU_CRITICAL_EXIT();
        break;

    case OS_TASK_STATE_PEND: (7)
        p_tcb->TaskState = OS_TASK_STATE_PEND_SUSPENDED;
        p_tcb->SuspendCtr = (OS_NESTING_CTR)1;
        CPU_CRITICAL_EXIT();
        break;
```c

    case OS_TASK_STATE_PEND_TIMEOUT: (8)
        p_tcb->TaskState = OS_TASK_STATE_PEND_TIMEOUT_SUSPENDED;
        p_tcb->SuspendCtr = (OS_NESTING_CTR)1;
        CPU_CRITICAL_EXIT();
        break;

    case OS_TASK_STATE_SUSPENDED: (9)
    case OS_TASK_STATE_DLY_SUSPENDED:
    case OS_TASK_STATE_PEND_SUSPENDED:
    case OS_TASK_STATE_PEND_TIMEOUT_SUSPENDED:
        p_tcb->SuspendCtr++;
        CPU_CRITICAL_EXIT();
        break;

    default: (10)
        CPU_CRITICAL_EXIT();
        *p_err = OS_ERR_STATE_INVALID;
        return;
    }

    /* 任务切换 */
    OSSched(); (11)
} #endif
```

- **代码清单: 挂起和恢复-3 (1) 和 (2):** 这部分代码是为了程序的健壮性写的代码，即是在各种判断，避免用户的误操作。在 μC/OS-III 中，这段代码随处可见，但为了讲解方便，我们把这部分代码注释掉，里面涉及的一些宏和函数我们均不实现，只需要了解即可，在后面的讲解中，要是出现这段代码，我们直接删除掉，删除掉也不会影响核心功能。

- **代码清单: 挂起和恢复-3 (3):** 如果任务挂起的是自己，则判断下调度器是否锁住，如果锁住则退出返回错误码，没有锁则继续往下执行。
• 代码清单: 挂起和恢复-3 (4): 根据任务的状态来决定挂起操作。

• 代码清单: 挂起和恢复-3 (5): 任务在就绪状态，则将任务的状态改为挂起态，挂起计数器置 1，然后从就绪列表删除。

• 代码清单: 挂起和恢复-3 (6): 任务在延时状态，则将任务的状态改为延时加挂起态，挂起计数器置 1，不用改变 TCB 的位置，即还是在延时的时基列表。

• 代码清单: 挂起和恢复-3 (7): 任务在等待状态，则将任务的状态改为等待加挂起态，挂起计数器置 1，不用改变 TCB 的位置，即还是在等待列表等待。等待列表暂时还没有实现，将会在后面的章节实现。

• 代码清单: 挂起和恢复-3 (8): 任务在等待加超时态，则将任务的状态改为等待加超时加挂起态，挂起计数器置 1，不用改变 TCB 的位置，即还在等待和时基这两个列表中。

• 代码清单: 挂起和恢复-3 (9): 只要有一个是挂起状态，则将挂起计数器加一操作，不用改变 TCB 的位置。

• 代码清单: 挂起和恢复-3 (10): 其他状态则无效，退出返回状态无效错误码。

• 代码清单: 挂起和恢复-3 (11): 任务切换。凡是涉及改变任务状态的地方，都需要进行任务切换。

15.1.3.2 OSTaskResume() 函数

OSTaskResume() 函数用于恢复被挂起的函数，但是不能恢复自己，挂起倒是可以挂起自己，具体实现见代码清单: 挂起和恢复-4。

列表 4: 代码清单: 挂起和恢复-4OSTaskResume() 函数

```c
#if OS_CFG_TASK_SUSPEND_EN > 0u
void OSTaskResume (OS_TCB *p_tcb,
                   OS_ERR *p_err)
{
    CPU_SR_ALLOC();
}
```

(下页继续)
#if 0 /* 屏蔽开始 */(1)
#define OS_SAFETY_CRITICAL
/ * 安全检查，OS_SAFETY_CRITICAL_EXCEPTION() 函数需要用户自行编写 */
if (p_err == (OS_ERR *)0) {
    OS_SAFETY_CRITICAL_EXCEPTION();
    return;
}
#endif

#if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u
/ * 不能在 ISR 程序中调用该函数 */
if (OSIntNestingCtr > (OS_NESTING_CTR)0) {
    *p_err = OS_ERR_TASK_RESUME_ISR;
    return;
}
#endif

CPU_CRITICAL_ENTER();
#if OS_CFG_ARG_CHK_EN > 0u
/ * 不能自己恢复自己 */
if ((p_tcb == (OS_TCB *)0) ||
    (p_tcb == OSTCBCurPtr)) {
    CPU_CRITICAL_EXIT();
    *p_err = OS_ERR_TASK_RESUME_SELF;
    return;
}
#endif
#endif/* 屏蔽结束 */(2)
39  *p_err = OS_ERR_NONE;
40  /* 根据任务的状态来决定挂起的动作 */
41  switch (p_tcb->TaskState) {
42    case OS_TASK_STATE_RDY:
43    case OS_TASK_STATE_DLY:
44    case OS_TASK_STATE_PEND:
45    case OS_TASK_STATE_PEND_TIMEOUT:
46        CPU_CRITICAL_EXIT();
47        *p_err = OS_ERR_TASK_NOT_SUSPENDED;
48    break;
49
50    case OS_TASK_STATE_SUSPENDED:
51        OS_CRITICAL_ENTER_CPU_CRITICAL_EXIT();
52            p_tcb->SuspendCtr--;
53            if (p_tcb->SuspendCtr == (OS_NESTING_CTR)0) {
54                p_tcb->TaskState = OS_TASK_STATE_RDY;
55                OS_TaskRdy(p_tcb);
56            }
57            OS_CRITICAL_EXIT_NO_SCHED();
58            break;
59
60    case OS_TASK_STATE_DLY_SUSPENDED:
61            p_tcb->SuspendCtr--;
62            if (p_tcb->SuspendCtr == (OS_NESTING_CTR)0) {
63                p_tcb->TaskState = OS_TASK_STATE_DLY;
64            }
65            CPU_CRITICAL_EXIT();
66            break;
67
68    case OS_TASK_STATE_PEND_SUSPENDED:
69            p_tcb->SuspendCtr--;
70            if (p_tcb->SuspendCtr == (OS_NESTING_CTR)0) {

p_tcb->TaskState = OS_TASK_STATE_PEND;
}
CPU_CRITICAL_EXIT();
break;

case OS_TASK_STATE_PEND_TIMEOUT_SUSPENDED:
    p_tcb->SuspendCtr--;
    if (p_tcb->SuspendCtr == (OS_NESTING_CTR)0) {
        p_tcb->TaskState = OS_TASK_STATE_PEND_TIMEOUT;
    }
    CPU_CRITICAL_EXIT();
break;

default:
    CPU_CRITICAL_EXIT();
    *p_err = OS_ERR_STATE_INVALID;
return;

/* 任务切换 */
OSSched();

#endif

• 代码清单：挂起和恢复-4  (1) 和 (2): 这部分代码是为了程序的健壮性写的代码，即是加了各种判断，避免用户的误操作。在µC/OS-III 中，这段代码随处可见，但为了讲解方便，我们把这部分代码注释掉，里面涉及的一些宏和函数我们均不实现，只需要了解即可，在后面的讲解中，要是出现这段代码，我们直接删除掉，删除掉也不会影响核心功能。

• 代码清单：挂起和恢复-4  (3): 根据任务的状态来决定恢复操作。

• 代码清单：挂起和恢复-4  (4): 只要任务没有被挂起，则退出返回任务没有被挂起的错误码。

论坛：https://www.firebbs.cn/  天猫：https://yehuosm.tmall.com
• 代码清单: 挂起和恢复-4 (5): 任务只在挂起态，则递减挂起计数器 SuspendCtr，如果 SuspendCtr 等于 0，则将任务的状态改为就绪态，并让任务就绪。

• 代码清单: 挂起和恢复-4 (6): 任务在延时加挂起态，则递减挂起计数器 SuspendCtr，如果 SuspendCtr 等于 0，则将任务的状态改为延时态。

• 代码清单: 挂起和恢复-4 (7): 任务在延时加等待态，则递减挂起计数器 SuspendCtr，如果 SuspendCtr 等于 0，则将任务的状态改为等待态。

• 代码清单: 挂起和恢复-4 (8): 任务在等待加超时加挂起态，则递减挂起计数器 SuspendCtr，如果 SuspendCtr 等于 0，则将任务的状态改为等待加超时态

• 代码清单: 挂起和恢复-4 (9): 其他状态则无效，退出返回状态无效错误码。

• 代码清单: 挂起和恢复-4 (10): 任务切换。凡是涉及改变任务状态的地方，都需要进行任务切换。

15.2 main() 函数

这里，我们创建任务 1、2 和 3，其中任务的优先级为 1，任务 2 的优先级为 2，任务 3 的优先级为 3。任务 1 将自身的 flag 每翻转一次后均将自己挂起，任务 2 在经过两个时钟周期后将任务 1 恢复，任务 3 每隔一个时钟周期翻转一次。具体代码见代码清单: 挂起和恢复-5。

列表 5: 代码清单: 挂起和恢复-5 main() 函数

```c
int main(void)
{
    OS_ERR err;

    /* CPU 初始化: 1, 初始化时间戳 */
    CPU_Init();

    /* 关闭中断 */
```

(下页继续)
CPU_IntDis();

/* 配置 SysTick 10ms 中断一次 */
OS_CPU_SysTickInit (10);

/* 初始化相关的全局变量 */
OSInit(&err);

/* 创建任务 */
OSTaskCreate( (OS_TCB *)&Task1TCB,
              (OS_TASK_PTR )Task1,
              (void       *)0,
              (OS_PRIO    )1,
              (CPU_STK    )&Task1Stk[0],
              (CPU_STK_SIZE )TASK1_STK_SIZE,
              (OS_TICK    )0,
              (OS_ERR     *)&err );

OSTaskCreate( (OS_TCB *)&Task2TCB,
              (OS_TASK_PTR )Task2,
              (void       *)0,
              (OS_PRIO    )2,
              (CPU_STK    )&Task2Stk[0],
              (CPU_STK_SIZE )TASK2_STK_SIZE,
              (OS_TICK    )0,
              (OS_ERR     *)&err );

OSTaskCreate( (OS_TCB *)&Task3TCB,
              (OS_TASK_PTR )Task3,
              (void       *)0,
              (OS_PRIO    )3,
              (CPU_STK    )&Task3Stk[0],
              (CPU_STK    ),
              (CPU_STK_SIZE ) TASK3_STK_SIZE,
              (OS_TICK    )0,
              (OS_ERR     *)&err );
(CPU_STK_SIZE  ) TASK3_STK_SIZE,
(OS_TICK       ) 0,
(OS_ERR       *) &err);

/* 启动 OS，将不再返回 */
OSStart(&err);
}

void Task1( void *p_arg )
{
    OS_ERR err;

    for ( ;; ) {
        flag1 = 1;
        OSTaskSuspend(&Task1TCB,&err);
        flag1 = 0;
        OSTaskSuspend(&Task1TCB,&err);
    }
}

void Task2( void *p_arg )
{
    OS_ERR err;

    for ( ;; ) {
        flag2 = 1;
        OSTimeDly(1);
        //OSTaskResume(&Task1TCB,&err);
        flag2 = 0;
        OSTimeDly(1);
        OSTaskResume(&Task1TCB,&err);
    }
}
void Task3( void *p_arg )
{
    for ( ;; ) {
        flag3 = 1;
        OSTimeDly(1);
        flag3 = 0;
        OSTimeDly(1);
    }
}

### 15.3 实验现象

进入软件调试，单击全速运行按钮就可看到实验波形，具体见图实验现象。在图实验现象中，可以看到任务 2 和任务 3 的波形图是一样的，任务 1 的波形周期是任务 2 的两倍，与代码实现相符。如果想实现其他效果可自行修改代码实现。
第 16 章 任务的删除

本章开始，我们让 OS 的任务支持删除操作，一个任务被删除后就进入休眠态，要想继续运行必须创新创建。

16.1 实现任务删除

16.1.1 编写任务删除函数

16.1.1.1 OSTaskDel() 函数

任务删除函数 OSTaskDel() 用于删除一个指定的任务，也可以删除自身，在 os_task.c 中定义，具体实现见代码清单: 任务的删除。

列表 1: 代码清单: 任务的删除 OSTaskDel() 函数

```c
#if OS_CFG_TASK_DEL_EN > 0u(1)
void OSTaskDel (OS_TCB *p_tcb,
               OS_ERR   *p_err)
{
    CPU_SR_ALLOC();

    /* 不允许删除空闲任务 */(2)
    if (p_tcb == &OSIdleTaskTCB)
    {
        *p_err = OS_ERR_TASK_DEL_IDLE;
        return;
    }
```

(下页继续)
/* 删除自己 */
if (p_tcb == (OS_TCB *)0) {
    CPU_CRITICAL_ENTER();
    p_tcb = OSTCBCurPtr;
    CPU_CRITICAL_EXIT();
}

OS_CRITICAL_ENTER();

/* 根据任务的状态来决定删除的动作 */
switch (p_tcb->TaskState) {
    case OS_TASK_STATE_RDY: { OS_RdyListRemove(p_tcb); break; }
    case OS_TASK_STATE_SUSPENDED: { break; }
    /* 任务只是在延时，并没有在任何等待列表 */
    case OS_TASK_STATE_DLY: case OS_TASK_STATE_DLY_SUSPENDED: { OS_TickListRemove(p_tcb); break; }
    case OS_TASK_STATE_PEND: case OS_TASK_STATE_PEND_SUSPENDED: case OS_TASK_STATE_PEND_TIMEOUT: case OS_TASK_STATE_PEND_TIMEOUT_SUSPENDED: { OS_TickListRemove(p_tcb); }
}
```c
#if 0 /* 目前我们还没有实现等待列表，暂时先把这部分代码注释 */
/* 看着在等待什么 */
switch (p_tcb->PendOn) {
    case OS_TASK_PEND_ON NOTHING:
        /* 任务信号量和队列没有等待队列，直接退出 */
        case OS_TASK_PEND_ON_TASK_Q:
        case OS_TASK_PEND_ON_TASK SEM:
            break;

        /* 从等待列表移除 */
        case OS_TASK_PEND_ON_FLAG:
        case OS_TASK_PEND_ON_MULTI:
        case OS_TASK_PEND_ON_MUTEX:
        case OS_TASK_PEND_ON_Q:
        case OS_TASK_PEND_ON SEM:
            OS_PendListRemove(p_tcb);
            break;

        default:
        break;
}
break;
#endif
default:
    OS_CRITICAL_EXIT();
    *p_err = OS_ERR STATE INVALID;
    return;
}

/* 初始化 TCB 为默认值 */
OS_TaskInitTCB(p_tcb);
/* 修改任务的状态为删除态，即处于休眠 */
```
p_tcb->TaskState = (OS_STATE)OS_TASK_STATE_DEL; (9)

OS_CRITICAL_EXIT_NO_SCHED();
/* 任务切换，寻找最高优先级的任   */
OSSched(); (10)

*p_err = OS_ERR_NONE;
}
#endif /* OS_CFG_TASK_DEL_EN > 0u */

- 代码清单: 任务的删除 (1): 任务删除是一个可选功能，由 OS_CFG_TASK_DEL_EN 控制，该宏在 os_cfg.h 中定义。

- 代码清单: 任务的删除 (2): 空闲任务不能被删除。系统必须至少有一个任务在运行，当没有其他用户任务运行的时候，系统就会运行空闲任务。

- 代码清单: 任务的删除 (3): 删除自己。

- 代码清单: 任务的删除 (4): 任务只在就绪态，则从就绪列表移除。

- 代码清单: 任务的删除 (5): 任务是被挂起，则退出返回，不用做什么。

- 代码清单: 任务的删除 (6): 任务在延时或者是延时加挂起，则从时基列表移除。

- 代码清单: 任务的删除 (7): 任务在多种状态，但只要有一种是等待状态，就需要从等待列表移除。如果任务等待是任务自身的信号量和消息，则直接退出返回，因为任务信号量和消息是没有等待列表的。等待列表我们暂时还没实现，所以暂时将等待部分相关的代码用条件编译屏蔽掉。

- 代码清单: 任务的删除 (8): 初始化 TCB 为默认值。

- 代码清单: 任务的删除 (9): 修改任务的状态为删除态，即处于休眠。

- 代码清单: 任务的删除 (10): 任务调度，寻找优先级最高的任务来运行。
16.2 main() 函数

本章 main() 函数没有添加新的测试代码，只需理解章节内容即可。

16.3 实验现象

本章没有实验，只需理解章节内容即可。
第 17 章  第二部分：μC/OS 内核应用开发

本书第二部分以野火 STM32 全系列开发板（包括 M3、M4 和 M7）为硬件平台，来讲解 μC/OS-III 的内核应用。这部分不会再深究源码的实现，着重讲解 μC/OS-III 各个内核对象的使用，例如任务如何创建、优先级如何分配、内部 IPC 通信机制如何使用等 RTOS 知识点。
第 18 章 移植 μC/OS-III 到 STM32

本章开始，先新建一个基于野火 STM32 全系列（包含 M3/4/7）开发板的 μC/OS-III 的工程模板，让 μC/OS-III 先跑起来。以后所有的 μC/OS-III 相关的例程我们都在此模板上修改和添加代码，不用再反反复复地新建。在本书配套的例程中，每一章的例程对野火 STM32 的每一个板子都会有一个对应的例程，但是区别都很小，如果有区别的地方我会在教程里面详细指出，如果没有特别备注那么都是一样的。

18.1 获取 STM32 的裸机工程模板

STM32 的裸机工程模板我们直接使用野火 STM32 开发板配套的固件库例程即可。这里我们选取比较简单的例程—“GPIO 输出一使用固件库点亮 LED”作为裸机工程模板。该裸机工程模板均可以在对应板子的 A 盘/程序源码/固件库例程的目录下获取到。下面以野火 F103-霸道板子的光盘目录为例，具体见图 STM32 裸机工程模板在光盘资料中的位置。
18.2 下载 μC/OS-III 源码

μC/OS™
RTOS and Stacks
Reliable. Efficient. Trusted.
μC/OS-III 是一个操作系统，其实也可以理解成一个软件库，它可以移植到多种硬件平台，如 M3、M4、M7 内核的 STM32，或者 ARM9 等等其他芯片。核心代码肯定是一致的，但是针对不同的处理器肯定要不同的实现部分。这里选择与我们开发板最为接近的版本（STMicroelectronics STM32F107），因为我们的野火 STM32 霸道开发板是 M3 内核的，而 μC/OS-III 的这个官方代码

论坛：https://www.firebbs.cn/ 240 天猫：https://yehuosm.tmall.com
就满足我们的需求，目的也在于少花费工夫，要知道，若要从 0 开始移植 μC/OS-III 到目标硬件平台，需要极大的精力和软件水平。

在“Projects”栏目中选择一个基于 Keil MDK 平台在 cortex-M3 内核 MCU 评估板上测试的 μC/OS-III 源码，单击即可。我们选择“STMicroelectronics STM32F107”这个项目的代码，单击后进入下载即可，不过 μC/OS 官网下载这些源码是需要注册账号的，而我们野火已经将这些源码下载了，在配套源码中，这样子就免去下载这一步了，直接拿来使用即可，具体见图 STMicroelectronics_STM32F107工程。图 STMicroelectronics_STM32F107工程下载与图 μCOS-III 源码。

<table>
<thead>
<tr>
<th>MCU</th>
<th>Micrium Product</th>
<th>Evaluation Board</th>
<th>Toolchain</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>STMicroelectronics STM32F0</td>
<td>μCOS-III</td>
<td>STM32018-EVAL</td>
<td>Atollic TrueSTUDIO V3.x</td>
<td>2013/02/05</td>
</tr>
<tr>
<td>STMicroelectronics STM32F0</td>
<td>μCOS-II</td>
<td>STM32018-EVAL</td>
<td>Atollic TrueSTUDIO V3.x</td>
<td>2013/02/05</td>
</tr>
<tr>
<td>STMicroelectronics STM32F103R8</td>
<td>μCOS-II</td>
<td>STMicroelectronics STM32F103R8</td>
<td>IAR STM32-LINK V3.x</td>
<td>2012/11/14</td>
</tr>
<tr>
<td>STMicroelectronics STM32F103ZE</td>
<td>μCOS-II</td>
<td>STMicroelectronics STM32F103ZE</td>
<td>IAR STM32-LINK V3.x</td>
<td>2012/12/05</td>
</tr>
<tr>
<td>STMicroelectronics STM32F107</td>
<td>μCOS-I</td>
<td>STMicroelectronics STM32F107</td>
<td>IAR STM32-LINK V3.x</td>
<td>2014/08/15</td>
</tr>
</tbody>
</table>

论坛：https://www.firebbs.cn/ 241 天猫：https://yehuosm.tmall.com
18.3 μC/OS-III 源码文件介绍

我们从μC/OS-III 源码下面的文件夹中看到，里面的文件夹不多，只有4个，分别是EvalBoards、
uC-CPU、uC-LIB、μC/OS-III，下面我们就来介绍一下这几个文件夹的作用。
18.3.1 EvalBoards

EvalBoards 文件夹里面包含评估板相关文件，在移植时我们只提取部分文件，具体见图 EvalBoards 中提取的代码，然后我们在我工程模板中的 User 文件夹下新建一个 APP 文件夹，然后将这9 个文件复制到 APP 文件夹下，具体见图复制 EvalBoards 中的文件到 APP 文件夹下。

![文件路径]

这些是我们需要的代码
将 “EvalBoardsMicriumuC-Eval-STM32F107BSP” 路径下的 bsp.c, bsp.h 文件复制到我们工程中的 UserBSP 文件夹下，具体见图提取 bsp 源码与图复制到工程中的 User_BSP 文件夹下。
18.3.2 uC-CPU

这是和 CPU 紧密相关的文件，里面的一些文件很重要，都是我们需要使用的。在 ARM-Cortex-M3 文件夹下，存在 cpu_c.c 一些对不同编译器移植相关的文件，有 GNU、IAR、RealView，里面都有一些很重要的文件，目前我们使用的开发环境是 MDK（keil），所以我们选择 RealView 文件夹下的源码文件作为我们的讲解，下面具体来介绍一下里面的文件，具体见图 uC-CPU 文件夹下的源码文件。

18.3.2.1 cpu_c.h 文件

包含了一些数据类型的定义，让 μC/OSIII 与 CPU 架构和编译器的字宽无关。同时还指定了 CPU 使用的是大端模式还是小端模式，还包括一些与 CPU 架构相关的函数的声明。
18.3.2.2 cpu_c.c 文件与 cpu_a.asm 文件

这两个文件主要是 CPU 底层相关的一些 CPU 函数，cpu_c.c 文件中放的是 C 函数，包含了一些 CPU 架构相关的代码，为什么要用 C 语言实现呢？μC/OS 是为了移植方便而采用 C 语言编写；而 cpu_a.asm 存放的是汇编代码，有一些代码只能用汇编实现，包含一些用来开关中断，前导零指令等。

18.3.2.3 cpu_core.c

cpu_core.c 文件包含了适用于所有的 CPU 架构的 C 代码，也就是常说的通用代码。是一个很重要的文件。主要包含的函数是 CPU 名字的命名，时间戳的计算等等，跟 CPU 底层的移植没有太大的关系，主要保留的是 CPU 前导零的 C 语言计算函数以及一些其他的函数，因为前导零指令是依靠硬件实现的，这里采用 C 语言方式实现，以防止某些 CPU 不支持前导零指令。

18.3.2.4 cpu_core.h

主要是对 cpu_core.c 文件里面一些函数的说明，以及一些时间戳相关等待定义。

18.3.2.5 cpu_def.h

包含 CPU 相关的一些宏定义，常量，利用 #define 进行定义的相关信息。

18.3.3 uC-LIB

Micrium 公司提供的官方库，诸如字符串操作、内存操作等接口，可用可不用。一般能用于代替标准库中的一些函数，使得在嵌入式中应用更加方便安全。
18.3.4 μC/OS-III

这是关键目录，我们下来着重分析的文件位于此目录下。

首先先看看 μC/OS-IIIPortsARM-Cortex-M3GenericRealView 目录下的文件，具体见图 μCOS-III_Ports 下的文件。

μC/OS 是软件，我们的开发板是硬件，软硬件必须有桥梁来连接，这些与处理器架构相关的代码，可以称之为 RTOS 硬件接口层，它们位于 μC/OS-IIIPorts 文件夹下，在不同的编译器中选择不同的文件，我们使用了 MDK，就使用 RealView 文件夹下的文件，这些文件不需要我们去修改，也不需要我们去理会，都是官方给我们写好的，直接使用即可。

18.3.4.1 os_cpu.h

定义数据类型、处理器相关代码、声明函数原型。

18.3.4.2 oc_cpu_a.asm

与处理器相关的汇编代码，主要是与任务切换相关。
18.3.4.3 os_cpu_c.c

定义用户钩子函数，提供扩充软件功能的接口。

打开 Source 文件，这个是μC/OS 的源码文件，我们可以看到这些就是μC/OS 核心文件，是非常重要的，我们在移植的时候必须将这里面的文件添加到我们的工程中去，具体见图 μCOS 源码。

下面介绍一下每个文件的功能作用，具体见下表
<table>
<thead>
<tr>
<th>文件</th>
<th>功能作用</th>
</tr>
</thead>
<tbody>
<tr>
<td>os.h</td>
<td>包含μC/OS-III的主要头文件，定义了一些与系统相关的宏定义、常量，声明了一些全局变量、函数原型等。</td>
</tr>
<tr>
<td>os_cfg_app.c</td>
<td>根据 os_cfg_app.h 中的配置来定义变量和数组。</td>
</tr>
<tr>
<td>os_core.c</td>
<td>内核数据结构管理，μC/OS-III的核心，任务切换等。</td>
</tr>
<tr>
<td>os_dbg.c</td>
<td>μC/OS-III 内核调试相关的代码。</td>
</tr>
<tr>
<td>os_flag.c</td>
<td>事件块管理、事件标志组管理等功能相关代码。</td>
</tr>
<tr>
<td>os_int.c</td>
<td>涵盖内核的初始化相关代码。</td>
</tr>
<tr>
<td>os_mem.c</td>
<td>系统内存管理相关代码。</td>
</tr>
<tr>
<td>os_msg.c</td>
<td>消息处理相关代码。</td>
</tr>
<tr>
<td>os_mutex.c</td>
<td>互斥量相关代码。</td>
</tr>
<tr>
<td>os_pend_multi.c</td>
<td>在多个消息队列、信号量等待的相关代码。</td>
</tr>
<tr>
<td>os_prio.c</td>
<td>这是一个内部调用的文件，关于任务就绪相关的代码。</td>
</tr>
<tr>
<td>os_q.c</td>
<td>消息队列相关代码。</td>
</tr>
<tr>
<td>os_sem.c</td>
<td>信号量相关代码。</td>
</tr>
<tr>
<td>os_stat.c</td>
<td>任务状态统计相关代码。</td>
</tr>
<tr>
<td>os_task.c</td>
<td>任务管理相关代码。</td>
</tr>
<tr>
<td>os_tick.c</td>
<td>处理处于延时、阻塞状态任务的相关代码。</td>
</tr>
<tr>
<td>os_time.c</td>
<td>时间管理相关代码，阻塞延时等。</td>
</tr>
<tr>
<td>os_tmrr.c</td>
<td>软件定时器相关代码。</td>
</tr>
<tr>
<td>os_var.c</td>
<td>μC/OS-III 定义的全局变量。</td>
</tr>
<tr>
<td>os_type.h</td>
<td>μC/OS-III 数据类型声明相关代码。</td>
</tr>
</tbody>
</table>

至此，关于μC/OS-III源码的文件就简单介绍完成，下面我们需要将其复制到我们的工程中，将μC-CPU、μC-LIB、μC/OS-III这3个文件夹复制到工程中的User文件夹下，然后进行移植，具体见图复制源码到工程中。
18.4 移植到 STM32 工程

在前一章节中，我们看到了μC/OS-III 源码中那么多文件，一开始学我们根本看不过来那么多文件，我们需要提取源码中的最简洁的部分代码，方便同学们学习，更何况我们学习的只是μC/OS-III 的实时内核中的知识，因为这才是μC/OS-III 的核心，那些 demo 都是基于此移植而来的，上一章节我们只是将μC/OS-III 的源码放到了本地工程目录下，还没有添加到开发环境里面的组文件夹里面，所以μC/OS-III 也就没有移植到我们的工程中去，现在开始讲μC/OS-III 的源码添加到工程中。

18.4.1 在工程中添加文件分组

我们需要先在工程中创建一些分组，方便我们分模块管理μC/OS-III 中的文件，就按照μC/OS-III 官方的命名方式创建我们的文件分组即可，具体见图在工程中添加文件分组。
18.4.2 添加文件到对应分组

向“APP”分组添加“UserAPP”文件夹下的所有文件，具体见图APP分组的文件。

向“BSP”分组添加“UserBSP”文件夹下的所有文件和“UserBSPled”文件夹下的源文件，具体见图BSP分组的文件。

向“μC/CPU”分组添加“UseruC-CPU”文件夹下的所有文件和“UseruC-CPUARM-Cortex-
M3RealView” 文件夹下的所有文件，具体见图 μC_CPU 分组的文件。

向 “μC/LIB” 分组添加 “UseruC-LIB” 文件夹下的所有文件和 “UseruC-LIBPortsARM-Cortex-M3RealView” 文件夹下的所有文件，具体见图 μC_LIB 分组的文件。

向 “μC/OS-III Source” 分组添加 “\UserμC/OS-III\Source” 文件夹下的所有文件，具体见图 μCOS-III_Source 分组的文件。
向“μC/OS-III Port”分组添加“UserμC/OS-III Ports ARM-Cortex-M3 Generic RealView”文件夹下的所有文件，具体见图μCOS-III Port分组的文件。

至此，我们的源码文件就添加到工程中了，当然此时仅仅是添加而已，并不是移植成功了，如果你编译一下工程就会发现一大堆错误，所以还需努力移植工程才行。
18.4.3 添加头文件路径到工程中

MC/OS-III 的源码已经添加到开发环境的组文件夹下面，编译的时候需要为这些源文件指定头文件的路径，不然编译会报错，此时我们先将头文件添加到我们的工程中，具体见图添加头文件路径到工程中。

至此，MC/OS 的整体工程基本移植完毕，我们需要修改 MC/OS 配置文件，按照我们的需求来进行

论坛：https://www.firebbs.cn/               天猫：https://yehuosm.tmall.com
行修改。

18.4.4 具体的工程文件修改

添加完头文件路径后，我们可以编译一下整个工程，但肯定会有错误的，μC/OS-III 的移植尚未完毕，接下来需要对工程文件进行修改。首先修改工程的启动文件 “startup_stm32f10x_hd.s” 。其中将 PendSV_Handler 和 SysTick_Handler 分别改为 OS_CPU_PendSVHandler 和 OS_CPU_SysTickHandler，共两处，因为 μC/OS 官方已经给我们处理好对应的中断函数，就无需我们自己处理与系统相关的中断了，同时我们还需要将 stm32f10x_it.c 文件中的 PendSV_Handler 和 SysTick_Handler 函数注释掉（当然不注释掉也没问题的），具体见图修改 startup_stm32f10x_hd.s 文件第 76-77 行，图修改 startup_stm32f10x_hd.s 文件第 193-197 行 与图注释掉 PendSV_Handler 和 SysTick_Handler 函数。

论坛：https://www.firebbs.cn/ 256 天猫：https://yehuosm.tmall.com
如果使用的是 M4/M7 内核带有 FPU（浮点运算单元）的处理器，那么还需要修改一下启动文件，如果想要使用 FPU，我们必须要在启动文件中添加以下代码，处理器必须处于特权模式才能读取和写入 CPACR，具体见代码清单: 移植-1 与图启动文件中插入代码。

注：关于具体的 FPU 相关说明请参考《ARM-Cortex-M4 内核参考手册》第 7 章相关内容。

<table>
<thead>
<tr>
<th>列表 1: 代码清单: 移植-1 启用 FPU（汇编）</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
</tbody>
</table>

(下页继续)
同时将对应芯片头文件中启用 FPU 的宏定义 __FPU_PRESENT 配置为 1（默认是启用的），然后在 Option->Target->Floating Point Hardware 中选择启用浮点运算，具体见图启用浮点运算。

论坛：https://www.firebbs.cn/  天猫：https://yehuosm.tmall.com
18.4.5 修改源码中的 bsp.c 与 bsp.h 文件

我们知道 bsp 就是板级相关的文件，也就是对应开发板的文件，而 μC/OS-III 源码的 bsp 肯定是与我们的板子不一样，所以就需要进行修改，而且以后我们的板级外设都在 bsp.c 文件进行初始化，所以按照我们修改好的源码进行修改即可，具体见代码清单: 移植-2 加粗部分。
列表2: 代码清单：移植-2 修改后的bsp.c 文件（已删掉注释）

```c
#define BSP_MODULE
#include <bsp.h>

CPU_INT32U BSP_CPU_ClkFreq_MHz;

#define DWT_CR *(CPU_REG32 *)0xE0001000
#define DWT_CYCCNT *(CPU_REG32 *)0xE0001004
#define DEM_CR *(CPU_REG32 *)0xE000EDFC
#define DBGMCU_CR *(CPU_REG32 *)0xE0042004

#define DBGMCU_CR_TRACE_IOEN_MASK 0x10
#define DBGMCU_CR_TRACE_MODE_ASYNC 0x00
#define DBGMCU_CR_TRACE_MODE_SYNC_01 0x40
#define DBGMCU_CR_TRACE_MODE_SYNC_02 0x80
#define DBGMCU_CR_TRACE_MODE_SYNC_04 0xC0
#define DBGMCU_CR_TRACE_MODE_MASK 0xC0

#define DEM_CR_TRCENA (1 << 24)
#define DWT_CR_CYCCNTENA (1 << 0)

void BSP_Init (void)
{
    LED_Init ();
}

CPU_INT32U BSP_CPU_ClkFreq (void)
{
    RCC_ClocksTypeDef rcc_clocks;
}
```

(下页继续)
RCC_GetClocksFreq(&rcc_clocks);

return ((CPU_INT32U)rcc_clocks.HCLK_Frequency);
}

#if ((APP_CFG_PROBE_OS_PLUGIN_EN == DEF_ENABLED) && (OS_PROBE_HOOKS_EN == 1))
void OSProbe_TmrInit (void)
{
}
#endif

#if ((APP_CFG_PROBE_OS_PLUGIN_EN == DEF_ENABLED) && (OS_PROBE_HOOKS_EN == 1))
CPU_INT32U OSProbe_TmrRd (void)
{
    return ((CPU_INT32U)DWT_CYCCNT);
}
#endif

#if (CPU_CFG_TS_TMR_EN == DEF_ENABLED)
void CPU_TS_TmrInit (void)
{
    CPU_INT32U cpu_clk_freq_hz;

    DEM_CR = (CPU_INT32U)DEM_CR_TRCENA;
    DWT_CYCCNT = (CPU_INT32U)0u;
    DWT_CR = (CPU_INT32U)DWT_CR_CYCCNTENA;
}
cpu_clk_freq_hz = BSP_CPU_ClkFreq();
CPU_TS_TmrFreqSet(cpu_clk_freq_hz);
}
#endif

#if (CPU_CFG_TS_TMR_EN == DEF_ENABLED)
CPU_TS_TMR CPU_TS_TmrRd (void)
{
    return ((CPU_TS_TMR)DWT_CYCCNT);
}
#endif

bsp.h 文件中需要添加我们自己的板级驱动头文件，头文件代码具体见代码清单: 移植-3。
列表 3: 代码清单: 移植-3bsp.h 文件添加我们自己的板级头文件

```c
#include "stm32f10x.h" // Modified by fire
#include <app_cfg.h>
#include "bsp_led.h" // Modified by fire
```

18.5 按需配置最适的工程

虽然前面的编译是没有错误的，并且工程模板也是可用的，但是此时还不是我们最适合使用的工程模板，最适合的工程往往是根据需要进行配置的，而 μC/OS 提供裁剪的功能，我们可以按需对系统进行裁剪。

18.5.1 os_cfg.h

os_cfg.h 文件是系统的配置文件，主要是让用户自己配置一些系统默认的功能，用户可以选择某些或者全部的功能，比如消息队列、信号量、互斥量、事件标志位等，系统默认全部使用的，如果如果用户不需要的话，则可以直接关闭，在对应的宏定义中设置为 0 即可，这样子就不会占用系统的 SRAM，以节省系统资源，os_cfg.h 文件的配置说明具体见代码清单: 移植-4。

列表 4: 代码清单: 移植-4os_cfg.h

```c
#ifndef OS_CFG_H
#define OS_CFG_H

/* --- 其他配置 --- */
#define OS_CFG_APP_HOOKS_EN 1u/* 是否使用钩子函数 */
```

(下页继续)
```c
# define OS_CFG_ARG_CHK_EN 1u/* 是否使用参数检查 */
# define OS_CFG_CALLED_FROM_ISR_CHK_EN 1u/* 是否使用中断调用检查 */
# define OS_CFG_DBG_EN 1u/* 是否使用 debug */
# define OS_CFG_ISR_POST_DEFERRED_EN 1u/* 是否使用中断延迟 post 操作 */
# define OS_CFG_OBJ_TYPE_CHK_EN 1u/* 是否使用对象类型检查 */
# define OS_CFG_TS_EN 1u/* 是否使用时间戳 */
# define OS_CFG_PEND_MULTI_EN 1u/* 是否使用支持多个任务 pend 操作 */
# define OS_CFG_PRIO_MAX 32u/* 定义任务的最大优先级 */
# define OS_CFG_SCHED_LOCK_TIME_MEAS_EN 1u/* 是否使用支持测量调度器锁定时间 */
# define OS_CFG_SCHED_ROUND_ROBIN_EN 1u/* 是否支持循环调度 */
# define OS_CFG_STK_SIZE_MIN 64u/* 最小的任务栈大小 */

/* ---------- 事件标志位---------- */
# define OS_CFG_FLAG_EN 1u/* 是否使用事件标志位 */
# define OS_CFG_FLAG_DEL_EN 1u/* 是否包含 OSFlagDel() 的代码 */
# define OS_CFG_FLAG_MODE_CLR_EN 1u/* 是否包含清除事件标志位的代码 */
# define OS_CFG_FLAG_PEND_ABORT_EN 1u/* 是否包含 OSFlagPendAbort() 的代码 */

/* 内存管理 --- */
# define OS_CFG_MEM_EN 1u/* 是否使用内存管理 */

/* 互斥量 ----- */
# define OS_CFG_MUTEX_EN 1u/* 是否使用互斥量 */
# define OS_CFG_MUTEX_DEL_EN 1u/* 是否包含 OSMutexDel() 的代码 */
```

论坛: https://www.firebbs.cn/ 天猫: https://yehuosm.tmall.com
# define OS_CFG_MUTEX_PEND_ABORT_EN 1u /* 是否包含 OSMutexPendAbort() 的代码 */

/* ------- 消息队列 --------------- */
#define OS_CFG_Q_EN 1u /* 是否使用消息队列 */
#define OS_CFG_Q_DEL_EN 1u /* 是否包含 OSQDel() 的代码 */
#define OS_CFG_Q_FLUSH_EN 1u /* 是否包含 OSQFlush() 的代码 */
#define OS_CFG_Q_PEND_ABORT_EN 1u /* 是否包含 OSQPendAbort() 的代码 */

/* --------------- 信号量 ------------ */
#define OS_CFG_SEM_EN 1u /* 是否使用信号量 */
#define OS_CFG_SEM_DEL_EN 1u /* 是否包含 OSSemDel() 的代码 */
#define OS_CFG_SEM_PEND_ABORT_EN 1u /* 是否包含 OSSemPendAbort() 的代码 */
#define OS_CFG_SEM_SET_EN 1u /* 是否包含 OSSemSet() 的代码 */

/* ----------------- 任务管理 ------------ */
#define OS_CFG_STAT_TASK_EN 1u /* 是否使用任务统计功能 */
#define OS_CFG_STAT_TASK_STK_CHK_EN 1u /* 从统计任务中检查任务栈 */
#define OS_CFG_TASK_CHANGE_PRIO_EN 1u /* 是否包含 OSTaskChangePrio() 的代码 */
#define OS_CFG_TASK_DEL_EN 1u /* 是否包含 OSTaskDel() 的代码 */
#define OS_CFG_TASK_Q_EN 1u /* 是否包含 OSTaskQXXX() 的代码 */
#define OS_CFG_TASK_Q_PEND_ABORT_EN 1u /* 是否包含 OSTaskQPendAbort() 的代码 */
#define OS_CFG_TASK_PROFILE_EN 1u /* 是否在 OS_TCB 中包含变量以进行性能分析 */
#define OS_CFG_TASK_REG_TBL_SIZE 1u /* 任务特定寄存器的数量 */
```c
#define OS_CFG_TASK_SEM_PEND_ABORT_EN 1u /* 是否包含 OSTaskSemPendAbort() 的代码 */
#define OS_CFG_TASK_SUSPEND_EN 1u /* 是否包含 OSTaskSuspend() 和 OSTaskResume() 的代码 */

/* ------- 时间管理 ------- */
#define OS_CFG_TIME_DLY_HMSM_EN 1u /* 是否包含 OTimeDlyHMSM() 的代码 */
#define OS_CFG_TIME_DLY_RESUME_EN 1u /* 是否包含 OTimeDlyResume() 的代码 */

/* ---------- 定时器管理 ---------- */
#define OS_CFG_TMR_EN 1u /* 是否使用定时器 */
#define OS_CFG_TMR_DEL_EN 1u /* 是否支持 OStmrDel() */

#endif
```

### 18.5.2 cpu_cfg.h

cpu_cfg.h 文件主要是配置 CPU 相关的一些宏定义，我们可以选择对不同的 CPU 进行配置，当然，如果我们没有对 CPU 很熟悉的话，就直接忽略这个文件即可，在这里我们只需要注意关于时间截与前导零指令相关的内容，我们使用的 CPU 是 STM32，是 32 位的 CPU，那么时间截我们使用 32 位的变量即可，而且 STM32 支持前导零指令，可以使用它让系统进行寻找最高优先级的任务更加快捷，具体见代码清单: 移植-5。

MC/OS 支持两种方法选择下一个要执行的任务：一个采用 C 语言实现前导零指令，这种方法我们通常称为通用方法，CPU_CFG_LEAD_ZEROS_ASM_PRESENT 没有被定义的时候使用才使用通用方法获取下一个即将运行的任务，通用方法可以用于所有 μC/OS 支持的硬件平台，因为这种方法是完全用 C 语言实现，所以效率略低于特殊方法，但不强制要求限制最大可用优先级数目; 另一个是硬件方式查找下一个要运行的任务，必须定义 CPU_CFG_LEAD_ZEROS_ASM_PRESENT 这个宏，因为这种方法是必须依赖一个或多个特定架构的汇编指令（一般是类似计算前导零 [CLZ] 指令，在 M3，M4，M7 内核中都有，这个指令是用来计算一个变量从最高位开始的连续零的个
数)，所以效率略高于通用方法，但受限于硬件平台。

列表 5: 代码清单: 移植-5cpu_cfg.h

```c
#ifndef CPU_CFG_MODULE_PRESENT
#define CPU_CFG_MODULE_PRESENT

/* 是否使用 CPU 名字: DEF_ENABLED 或者 DEF_DISABLED */
#define CPU_CFG_NAME_EN DEF_ENABLED

/* CPU 名字大小 (ASCII 字符串形式) */
#define CPU_CFG_NAME_SIZE 16u

/* CPU 时间戳功能配置（只能选择其中一个） */
/* 是否使用 32 位的时间戳变量: DEF_ENABLED 或者 DEF_DISABLED */
#define CPU_CFG_TS_32_EN DEF_ENABLED

/* 是否使用 64 位的时间戳变量: DEF_ENABLED 或者 DEF_DISABLED */
#define CPU_CFG_TS_64_EN DEF_DISABLED

/* 配置 CPU 时间戳定时器字大小 */
#define CPU_CFG_TS_TMR_SIZE CPU_WORD_SIZE_32

/* 是否使用测量 CPU 禁用中断的时间 */
#if 0
#define CPU_CFG_INT_DIS_MEAS_EN
#endif

/* 配置测量的次数 */
#define CPU_CFG_INT_DIS_MEAS_OVRHD_NBR 1u
```

(下页继续)
/* 是否使用 CPU 前导零指令（需要硬件支持，在 stm32 我们可以使用这个指令） */
#if 1
#define CPU_CFG_LEAD_zeros_asm_PRESENT
#endif
#endif

18.5.3 os_cfg_app.h

os_cfg_app.h 是系统应用配置的头文件，简单来说就是系统默认的任务配置，如任务的优先级、栈大小等基本信息，但是有两个任务是必须开启的，一个就是空闲任务，另一个就是时钟节拍任务，这两个是让系统正常运行的最基本任务，而其他任务我们自己按需配置即可。

列表 6：代码清单：移植-os_cfg_app.h

```
#ifndef OS_CFG_APP_H
#define OS_CFG_APP_H

/* --------------------- MISCELLANEOUS ------------------ */
#define OS_CFG_MSG_POOL_SIZE 100u/* 支持的最大消息数量 */
#define OS_CFG_ISR_STK_SIZE 128u/*ISR 栈的大小 */
#define OS_CFG_TASK_STK_LIMIT_PCT_EMPTY 10u/* 检查栈的剩余大小（百分百形式，此处是 10%） */

/* ---------------------- 空闲任务 ---------------------- */
#define OS_CFG_IDLE_TASK_STK_SIZE 128u/* 空闲任务栈大小 */
```

(下页继续)
/* ------------------ 中断处理任务 ------------------ */
#define OS_CFG_INT_Q_SIZE 10u /* 中断处理任务队列大小 */
#define OS_CFG_INT_Q_TASK_STK_SIZE 128u /* 中断处理任务的栈大小 */

/* ------------------ 统计任务 ------------------ */
#define OS_CFG_STAT_TASK_PRIO 11u /* 统计任务的优先级 */
#define OS_CFG_STAT_TASK_RATE_HZ 10u /* 统计任务的周期频率（10Hz） */
#define OS_CFG_STAT_TASK_STK_SIZE 128u /* 统计任务的栈大小 */

/* ------------------ 时钟节拍任务 ------------------ */
#define OS_CFG_TICK_RATE_HZ 1000u /* 系统的时钟节拍（一般为10到1000Hz） */
#define OS_CFG_TICK_TASK_PRIO 1u /* 时钟节拍任务的优先级 */
#define OS_CFG_TICK_TASK_STK_SIZE 128u /* 时钟节拍任务的栈大小 */
#define OS_CFG_TICK_WHEEL_SIZE 17u /* 时钟节拍任务的列表大小 */

/* ------------------ 定时器任务 ------------------ */
#define OS_CFG_TMR_TASK_PRIO 11u /* 定时器任务的优先级 */
#define OS_CFG_TMR_TASK_RATE_HZ 10u /* 定时器频率（10Hz 是典型值） */
#define OS_CFG_TMR_TASK_STK_SIZE 128u /* 定时器任务的栈大小 */
#define OS_CFG_TMR_WHEEL_SIZE 17u /* 定时器任务的列表大小 */

#endif

此处要注意时钟节拍任务，μC/OS 的时钟节拍任务是用于管理时钟节拍的，建议将其优先级设置更高一些，这样子在调度的时候，时钟节拍任务能抢占其他任务执行，从而能够更新任务，相对于其他操作系统，寻找处于最高优先级的就绪任务都是在中断中，μC/OS 将其放于任务中能更好解决关中断时间过长的问题。

论坛：https://www.firebbs.cn/ 269 天猫：https://yehuosm.tmall.com
18.6 修改 app.c

我们将原来裸机工程里面 app.c 的文件内容全部删除，新增如下内容，具体见代码清单: 移植-7。

列表 7: 代码清单: 移植-7app.c 文件内容

```c
/**
 * @file  app.c
 * @author fire
 * @version V1.0
 * @date  2018-xx-xx
 * @brief MC/OS-III + STM32 工程模版
**/

#include <includes.h>

```

(下页继续)
变量

/*
 */

/*
 */

main 函数

/*
 */

/**
 * @brief 主函数
 * @param 无
 * @retval 无
 */

int main(void)
{
    /* 暂时没有在 main 里面创建任务应用任务 */
}

END OF FILE*/
18.7 下载验证

将程序编译好，用 DAP 仿真器把程序下载到野火 STM32 开发板（具体型号根据购买的板子而定，每个型号的板子都配套有对应的程序），一看，啥现象都没有，一脸懵逼，我说，你急个啥，目前我们还没有在 main() 函数里面创建任务，系统也没有跑起来，main() 函数中什么都没有，那当然是没有现象。如果要想看现象，得自己在 main 创建里面应用任务，并且让 μC/OS 跑起来，关于如何使用 μC/OS 创建任务，请看下一章“创建任务”
第 19 章  创建任务

在上一章，我们已经基于野火 STM32 开发板创建好了 μC/OS-III 的工程模板，这章开始我们将真正进入如何使用 μC/OS 的征程，先从最简单的创建任务开始，点亮一个 LED，以慰藉下尔等初学者弱小的心灵。

19.1 硬件初始化

本章创建的任务需要用到开发板上的 LED，所以先要将 LED 相关的函数初始化好，为了方便以后统一管理板级外设的初始化，我们在 bsp.c 文件中创建一个 BSP_Init() 函数，专门用于存放板级外设初始化函数，具体见代码清单: 创建任务-1 的加粗部分。

列表 1: 代码清单: 创建任务-1 BSP_Init() 中添加硬件初始化函数

```c
/*
 * @ 函数名:  BSP_Init
 * @ 功能说明: 板级外设初始化，所有板子上的初始化均可放在这个函数里面
 * @ 参数:
 * @ 返回值: 无
 */

static void BSP_Init(void)
{

    /*
    * STM32 中断优先级分组为 4，即 4bit 都用来表示抢占优先级，范围为: 0~15
    * 优先级分组只需要分组一次即可，以后如果有其他的任务需要用到中断，
    * 都统一用这个优先级分组，千万不要再分组，切忌。
    */
    NVIC_PriorityGroupConfig( NVIC_PriorityGroup_4 );
}
```

(下页继续)
执行到 BSP_Init() 函数的时候，操作系统完全都还没有涉及，即 BSP_Init() 函数所做的事情跟我们以前编写的裸机工程里面的硬件初始化工作是一模一样的。运行完 BSP_Init() 函数，接下来才慢慢启动操作系统，最后运行创建好的任务。有时候任务创建好，整个系统跑起来了，可想要的实验现象就是出不来，比如 LED 不会亮，串口没有输出，LCD 没有显示等等。如果是初学者，这个时候就会心急如焚，四处求救，那怎么办？这个时候如何排除是硬件的问题还是系统的问题，这里面有个小小的技巧，即在硬件初始化好之后，顺便测试下硬件，测试方法跟裸机编程一样，具体实现见代码清单: 创建任务-2 的加粗部分。

### 列表 2: 代码清单: 创建任务-2BSP_Init() 中添加硬件测试函数

```c
/* 开发板硬件 bsp 头文件 */
#include"bsp_led.h"
#include"bsp_usart.h"

/***********************************************************************************/
* @ 函数名： BSP_Init
* @ 功能说明：板级外设初始化，所有板子上的初始化均可放在这个函数里面
* @ 参数：
* @ 返回值：无
***********************************************************************************/
static void BSP_Init(void)
{
    /*
```
```
/* STM32 中断优先级分组为 4，即 4bit 都用来表示抢占优先级，范围为：0~15
* 优先级分组只需要分组一次即可，以后如果有其他的任务需要用到中断，
* 都统一用这个优先级分组，千万不要再分组，切忌。
*/
NVIC_PriorityGroupConfig( NVIC_PriorityGroup_4 );

/* LED 初始化 */
LED_GPIO_Config();

/* 测试硬件是否正常工作 */
LED1_ON;

/* 其他硬件初始化和测试 */

/* 让程序停留在这里，不再继续往下执行 */
while (1);

/* 串口初始化 */
USART_Config();

}  

• 代码清单: 创建任务-2 (1): 初始化硬件后，顺便测试硬件，看下硬件是否正常工作。

• 代码清单: 创建任务-2 (2): 可以继续添加其他的硬件初始化和测试。硬件确认没有问题之后，硬件测试代码可删可不删，因为 BSP_Init() 函数只执行一遍。

• 代码清单: 创建任务-2 (3): 方便测试硬件好坏，让程序停留在这里，不再继续往下执行，当测试完毕后，这个 while(1); 必须删除。

注意: 以上仅仅是测试代码，以实际工程代码为准。
19.2 创建单任务

这里，我们创建一个单任务，任务使用的栈和任务控制块都使用静态内存，即预先定义好的全局变量，这些预先定义好的全局变量都存在内部的 SRAM 中。

19.2.1 定义任务栈

目前我们只创建了一个任务，当任务进入延时的时候，因为没有另外就绪的用户任务，那么系统就会进入空闲任务，空闲任务是 uCOS 系统自己创建并且启动的一个任务，优先级最低。当整个系统都没有就绪任务的时候，系统必须保证有一个任务在运行，空闲任务就是为这个设计的。当用户任务延时到期，又会从空闲任务切换回用户任务。

在 uCOS 系统中，每一个任务都是独立的，他们的运行环境都单独的保存在他们的栈空间当中。那么在定义好任务函数之后，我们还要为任务定义一个栈，目前我们使用的是静态内存，所以任务栈是一个独立的全局变量，具体见代码清单: 创建任务-3。任务的栈占用的是 MCU 内部的 RAM，当任务越多的时候，需要使用的栈空间就越大，即需要使用的 RAM 空间就越多。一个 MCU 能够支持多少任务，就得看你的 RAM 空间有多少。

列表 3: 代码清单: 创建任务-3 定义任务栈

```c
#define APP_TASK_START_STK_SIZE 128
static CPU_STK AppTaskStartStk[APP_TASK_START_STK_SIZE];
```

19.2.2 定义任务控制块

定义好任务函数和任务栈之后，我们还需要为任务定义一个任务控制块，通常我们称这个任务控制块为任务的身份证。在 C 代码上，任务控制块就是一个结构体，里面有非常多的成员，这些成员共同描述了任务的全部信息，具体见代码清单: 创建任务-4。

论坛: https://www.firebbs.cn/ 276 天猫: https://yehuosm.tmall.com
列表 4: 代码清单: 创建任务-4 定义任务控制块

```
static OS_TCB AppTaskStartTCB;
```

### 19.2.3 定义任务主体函数

任务实际上就是一个无限循环且不带返回值的 C 函数。目前，我们创建一个这样的任务，让开发板上面的 LED 灯以 500ms 的频率闪烁，具体实现见代码清单: 创建任务-5。

列表 5: 代码清单: 创建任务-5 定义任务函数（此处为伪代码，以工程代码为准）

```
static void LED_Task (void* parameter)
{
    while (1)      (1)
    {
        LED1_ON;
        OSTimeDly (500,OS_OPT_TIME_DLY,&err); /* 延时 500 个 tick */ (2)
        LED1_OFF;
        OSTimeDly (500,OS_OPT_TIME_DLY,&err); /* 延时 500 个 tick */
    }
}
```

- 代码清单: 创建任务-5 (1): 任务必须是一个死循环，否则任务将通过 LR 返回，如果 LR 指向了非法的内存就会产生 HardFault_Handler，而 μC/OS 指向一个任务退出函数 OS_TaskReturn()，它如果支持任务删除的话，则进行任务删除操作，否则就进入死循环中，这样子的任务是不安全的，所以避免这种情况，任务一般都是死循环并且无返回值的，只执行一次的任务在执行完毕要记得及时删除。
• 代码清单: 创建任务-5 (2): 任务里面的延时函数必须使用 μC/OS 里面提供的阻塞延时函数，并不能使用我们裸机编程中的那种延时。这两种的延时的区别是 μC/OS 里面的延时是阻塞延时，即调用 OSTimeDly() 函数的时候，当前任务会被挂起，调度器会切换到其他就绪的任务，从而实现多任务。如果还是使用裸机编程中的那种延时，那么整个任务就成为了一个死循环，如果恰好该任务的优先级是最高的，那么系统永远都是在这个任务中运行，比它优先级更低的任务无法运行，根本无法实现多任务，因此任务中必须有能阻塞任务的函数，才能切换到其他任务中。

19.2.4 创建任务

一个任务的三要素是任务主体函数，任务栈，任务控制块，那么怎么样把这个三要素联合在一起？μC/OS 里面有一个叫任务创建函数 OSTaskCreate()，它就是干这个活的。它将任务主体函数，任务栈和任务控制块这三者联系在一起，让任务在创建之后可以随时被系统启动与调度，具体见代码清单: 创建任务-6。

<table>
<thead>
<tr>
<th>代码清单: 创建任务-6 创建任务</th>
</tr>
</thead>
<tbody>
<tr>
<td>OTaskCreate((OS_TCB *) &amp;AppTaskStartTCB,</td>
</tr>
<tr>
<td>(CPU_CHAR *) &quot;App Task Start&quot;,</td>
</tr>
<tr>
<td>(OS_TASK_PTR) AppTaskStart,</td>
</tr>
<tr>
<td>(void *) 0,</td>
</tr>
<tr>
<td>(OS_PRIO) APP_TASK_START_PRIO,</td>
</tr>
<tr>
<td>(CPU_STK *) &amp;AppTaskStartStk[0],</td>
</tr>
<tr>
<td>(CPU_STK_SIZE) APP_TASK_START_STK_SIZE / 10,</td>
</tr>
<tr>
<td>(CPU_STK_SIZE) APP_TASK_START_STK_SIZE,</td>
</tr>
<tr>
<td>(OS_MSG_QTY) 5u,</td>
</tr>
<tr>
<td>(OS_TICK) 0u,</td>
</tr>
<tr>
<td>(void *) 0,</td>
</tr>
<tr>
<td>(OS_OPT) (OS_OPT_TASK_STK_CHK</td>
</tr>
<tr>
<td>(OS_ERR *) &amp;err);</td>
</tr>
</tbody>
</table>

• 代码清单: 创建任务-6 (1): 任务控制块，由用户自己定义。

论坛: https://www.firebbs.cn/ 天猫: https://yehuosm.tmall.com


- **代码清单: 创建任务-6 (2)**: 任务名称，字符串形式，这里任务名称最好要与任务函数入口名字一致，方便进行调试。

- **代码清单: 创建任务-6 (3)**: 任务入口函数，即任务函数的名称，需要我们自己定义并且实现。

- **代码清单: 创建任务-6 (4)**: 任务入口函数形参，不用的时候配置为 0 或者 NULL 即可，p_arg 是指向可选数据区域的指针，用于将参数传递给任务，因为任务一旦执行，那必须是在一个死循环中，所以传参只在首次执行时有效。

- **代码清单: 创建任务-6 (5)**: 任务的优先级，由用户自己定义。

- **代码清单: 创建任务-6 (6)**: 指向栈基址的指针（即栈的起始地址）。

- **代码清单: 创建任务-6 (7)**: 设置栈深度的限制位置。这个值表示任务的栈满溢之前剩余的栈容量。例如，指定 stk_size 值的 10% 表示将达到栈限制，当栈达到 90% 满就表示任务的栈已满。

- **代码清单: 创建任务-6 (8)**: 任务栈大小，单位由用户决定，如果 CPU_STK 被设置为 CPU_INT08U，则单位为字节，而如果 CPU_STK 被设置为 CPU_INT16U，则单位为半字，同理，如果 CPU_STK 被设置为 CPU_INT32U，单位为字。在 32 位的处理器下（STM32），一个字等于 4 个字节，那么任务大小就为 APP_TASK_START_STK_SIZE * 4 字节。

- **代码清单: 创建任务-6 (9)**: 设置可以发送到任务的最大消息数，按需设置即可。

- **代码清单: 创建任务-6 (10)**: 在任务之间循环时的时间片的时间量（以滴答为单位）。指定 0 则使用默认值。

- **代码清单: 创建任务-6 (11)**: 是指向用户提供的内存位置的指针，用作 TCB 扩展。例如，该用户存储器可以保存浮点寄存器的内容在上下文切换期间，每个任务执行的时间，次数、任务已经切换等。

- **代码清单: 创建任务-6 (12)**: 用户可选的任务特定选项，具体见代码清单: 创建任务-7。

### 列表 7: 代码清单: 创建任务-7 任务特定选项

```c
#define OS_OPT_TASK_NONE (OS_OPT)(0x0000u) (1)  
#define OS_OPT_TASK_STK_CHK (OS_OPT)(0x0001u) (2)
```

(下页继续)
### 19.2.5 启动任务

当任务创建好后，是处于任务就绪，在就绪态的任务可以参与操作系统的调度。任务调度器只启动一次，之后就不会再次执行了，μC/OS 中启动任务调度器的函数是 OSStart()，并且启动任务调度器的时候就不会返回，从此任务都由 μC/OS 管理，此时才是真正进入实时操作系统中的第一步，具体见代码清单: 创建任务-8。
19.2.6 app.c 全貌

现在我们把任务主体，任务栈，任务控制块这三部分代码统一放到 app.c 中，我们在 app.c 文件中创建一个 AppTaskStart 任务，这个任务是仅是用于测试用户任务，以后为了方便管理，我们的所有的任务创建都统一放在这个任务中，在这个任务中创建成功的任务就可以直接参与任务调度了，具体内容见 代码清单: 创建任务-9。

列表 9: 代码清单: 创建任务-9 app.c 全貌

```c
#include <includes.h>

int main (void)
{
    OS_ERR err;

    OSInit(&err); /* Init μC/OS-III.*/

    OSTaskCreate((OS_TCB *)&AppTaskStartTCB, /* Create the start task */
                 (CPU_CHAR *)"App Task Start",
                 (OS_TASK_PTR ) AppTaskStart,
                 (void *) 0,
                 (OS_PRIO   ) APP_TASK_START_PRIO,
                 (CPU_STK   *) &AppTaskStartStk[0],
```

(下页继续)
(续上页)

(CPU_STK_SIZE) APP_TASK_START_STK_SIZE / 10,
(CPU_STK_SIZE) APP_TASK_START_STK_SIZE,
(OS_MSG_QTY ) 5u,
(OS_TICK ) 0u,
(void *) 0,
(OS_OPT ) (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
(OS_ERR *) &err);

OSStart(&err); /* Start multitasking (i.e. give control to μC/OS-III). */
}
/*

*****************************************************
 * STARTUP TASK
 * Description : This is an example of a startup task. As mentioned in
 * the book's text, you MUST initialize the ticker only once
 * multitasking has started.
 *
 * Arguments : p_arg is the argument passed to 'AppTaskStart()' by
 * 'OSTaskCreate()'.
 * Returns : none
 * Notes: 1) The first line of code is used to prevent a compiler warning
 * because 'p_arg' is not
 * used. The compiler should not generate any code for this statement.
 *****************************************************
*/

static void AppTaskStart (void *p_arg)
{
    CPU_INT32U cpu_clk_freq;

CPU_INT32U cnts;
OS_ERR err;

(void)p_arg;

BSP_Init(); /* Initialize BSP functions */
CPU_Init();
/* Determine SysTick reference freq*/
cpu_clk_freq = BSP_CPU_ClkFreq();
/* Determine nbr SysTick increments */
cnts = cpu_clk_freq / (CPU_INT32U)OSCfg_TickRate_Hz;

OS_CPU_SysTickInit(cnts); /* Init μC/OS periodic time src (SysTick). */

Mem_Init(); /* Initialize Memory Management Module */

#if OS_CFG_STAT_TASK_EN > 0u
OSStatTaskCPUUsageInit(&err); /* Compute CPU capacity with no task running */
#endif

CPU_IntDisMeasMaxCurReset();

while (DEF_TRUE) { /* Task body, always written as an infinite loop. */
    macLED1_TOGGLE();
    OSTimeDly (5000, OS_OPT_TIME_DLY, &err);
}
19.3 下载验证

将程序编译好，用 DAP 仿真器把程序下载到野火 STM32 开发板（具体型号根据购买的板子而定，每个型号的板子都配套有对应的程序），可以看到板子上面的 LED 灯已经在闪烁，说明我们创建的单任务已经跑起来了。

19.4 创建多任务

创建多任务只需要按照创建单任务的套路依葫芦画瓢即可，接下来我们创建四个任务，分别是起始任务、LED1 任务、LED2 任务和 LED3 任务。任务 1 让一个 LED 灯闪烁，任务 2 让另外一个 LED 闪烁，两个 LED 闪烁的频率不一样，三个任务的优先级不一样。主函数运行时创建起始任务，起始任务运行时进行创建三个 LED 灯的任务和删除自身，之后就运行三个 LED 灯的任务。三个 LED 灯的任务优先级不一样，LED1 任务为 LED1 每隔 1 秒切换一次亮灭状态，LED2 任务为 LED2 每隔 5 秒切换一次亮灭状态，LED3 任务为 LED3 每隔 10 秒切换一次亮灭状态，首先在“app_cfg.h”里，增加定义三个 LED 灯任务的优先级和栈空间大小，然后修改 app.c 的源码，具体见代码清单: 创建任务-10 加粗部分。

列表 10: 代码清单: 创建任务-10app.c 全貌

```c
#include <includes.h>

/*
 ********************************************
 * LOCAL DEFINES
 ********************************************
 */
```
/*
 *************************************************************/
* TCB
 *************************************************************/
*/

static OS_TCB AppTaskStartTCB;
static OS_TCB AppTaskLed1TCB;
static OS_TCB AppTaskLed2TCB;
static OS_TCB AppTaskLed3TCB;

/*
 *************************************************************/
* STACKS
 *************************************************************/
*/

static CPU_STK AppTaskStartStk[APP_TASK_START_STK_SIZE];
static CPU_STK AppTaskLed1Stk [ APP_TASK_LED1_STK_SIZE ];
static CPU_STK AppTaskLed2Stk [ APP_TASK_LED2_STK_SIZE ];
static CPU_STK AppTaskLed3Stk [ APP_TASK_LED3_STK_SIZE ];

/*
 ***********************************************/
* FUNCTION PROTOTYPES
 ***********************************************/
*/

static void AppTaskStart (void *p_arg);
static void AppTaskLed1 ( void * p_arg );
static void AppTaskLed2 ( void * p_arg );
static void AppTaskLed3 ( void * p_arg );

/*
   ***********************************************
   * main()
   *
   * Description : This is the standard entry point for C code. It is
   * assumed that your code will call main() once you have
   * performed all necessary initialization.
   * Arguments : none
   *
   * Returns : none
   ***********************************************
   */

int main (void)
{
    OS_ERR err;

    OSInit(&err);    /* Init μC/OS-III.
     */

    OSTaskCreate((OS_TCB *) &AppTaskStartTCB, /*Create the
      start task */
        (CPU_CHAR *) "App Task Start",
        (OS_TASK_PTR ) AppTaskStart,
        (void *) 0,
        (OS_PRIO ) APP_TASK_START_PRIO,

    )
(CPU_STK *) &AppTaskStartStk[0],
(CPU_STK_SIZE) APP_TASK_START_STK_SIZE / 10,
(CPU_STK_SIZE) APP_TASK_START_STK_SIZE,
(OS_MSG_QTY ) 5u,
(OS_TICK ) 0u,
(void ) 0,
(OS_OPT ) (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
(OS_ERR ) &err);

OSStart(&err);                      /* Start multitasking (i.e. givecontrol_...
                                 to μC/OS-III). */
*/

/*
******************************************************************************
* STARTUP TASK
* Description : This is an example of a startup task. As mentioned in
* the book's text, you MUST initialize the ticker only once
* multitasking has started.
* Arguments : p_arg is the argument passed to 'AppTaskStart()' by
* 'OSTaskCreate()'.
* Returns : none
* Notes : 1) The first line of code is used to prevent a compiler
* warning because 'p_arg' is not used. The compiler should
* not generate any code for this statement.
******************************************************************************
*/
### Code Snippet

```c
static void AppTaskStart (void *p_arg)
{
    CPU_INT32U cpu_clk_freq;
    CPU_INT32U cnts;
    OS_ERR err;

    (void)p_arg;

    BSP_Init(); /* Initialize BSP functions */
    CPU_Init();

    cpu_clk_freq = BSP_CPU_ClkFreq(); /* Determine SysTick reference freq. */
    cnts = cpu_clk_freq / (CPU_INT32U)OSCfg_TickRate_Hz; /* Determine nbrSysTick increments */
    OS_CPU_SysTickInit(cnts); /* Init μC/OS periodic time src(SysTick). */

    Mem_Init(); /* Initialize Memory Management Module */

    #if OS_CFG_STAT_TASK_EN > 0u
    OSStatTaskCPUUsageInit(&err); /* Compute CPU capacity with no task running */
    #endif

    CPU_IntDisMeasMaxCurReset();

    OSTaskCreate((OS_TCB *)AppTaskLed1TCB, /*Create the Led1 task */
```
(CPU_CHAR *)"App Task Led1",
(OS_TASK_PTR) AppTaskLed1,
(void *) 0,
(OS_PRIO) APP_TASK_LED1_PRIO,
(CPU_STK *)&AppTaskLed1Stk[0],
(CPU_STK_SIZE) APP_TASK_LED1_STK_SIZE / 10,
(CPU_STK_SIZE) APP_TASK_LED1_STK_SIZE,
(OS_MSG_QTY) 5u,
(OS_TICK) 0u,
(void *) 0,
(OS_OPT)(OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
(OS_ERR *)&err);

OSTaskCreate((OS_TCB *)&AppTaskLed2TCB, /*Create the Led2 task*/
(CPU_CHAR *)"App Task Led2",
(OS_TASK_PTR) AppTaskLed2,
(void *) 0,
(OS_PRIO) APP_TASK_LED2_PRIO,
(CPU_STK *)&AppTaskLed2Stk[0],
(CPU_STK_SIZE) APP_TASK_LED2_STK_SIZE / 10,
(CPU_STK_SIZE) APP_TASK_LED2_STK_SIZE,
(OS_MSG_QTY) 5u,
(OS_TICK) 0u,
(void *) 0,
(OS_OPT)(OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
(OS_ERR *)&err);

OSTaskCreate((OS_TCB *)&AppTaskLed3TCB, /*Create the Led3 task*/
(CPU_CHAR *)"App Task Led3",
(OS_TASK_PTR) AppTaskLed3,
(void *) 0,
(OS_PRIO) APP_TASK_LED3_PRIO,
(CPU_STK *) &AppTaskLed3Stk[0],
(CPU_STK_SIZE) APP_TASK_LED3_STK_SIZE / 10,
(OS_MSG_QTY ) 5u,
(OS_TICK ) 0u,
(OS_OPT ) (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
(OS_ERR * ) &err);

OSTaskDel ( & AppTaskStartTCB, & err );

*/
*******************************************************************************/
LED1 TASK
*******************************************************************************/

static void AppTaskLed1 ( void * p_arg )
{
    OS_ERR err;

    (void) p_arg;

    while (DEF_TRUE) {    /* Task body, always written as an infinite loop. */

    }
}
static void AppTaskLed2 ( void * p_arg )
{
    OS_ERR err;

    (void)p_arg;

    while (DEF_TRUE) { /* Task body, always written as an infinite loop. */
        macLED2_TOGGLE ();
        OSTimeDly ( 5000, OS_OPT_TIME_DLY, & err );
    }
}

/*
***********************************************************************
* LED2 TASK
***********************************************************************
*/
static void AppTaskLed3 ( void * p_arg )
{
    OS_ERR err;

    (void)p_arg;

    while (DEF_TRUE) {
        /* Task body, always written as an infinite loop. */
        macLED3_TOGGLE ();
        OSTimeDly ( 10000, OS_OPT_TIME_DLY, & err );
    }
}

19.5 下载验证

将程序编译好，用 DAP 仿真器把程序下载到野火 STM32 开发板（具体型号根据购买的板子而定，每个型号的板子都配套有对应的程序），可以看到板子上面的三个 LED 灯以不同的频率在闪烁，说明我们创建的多任务已经跑起来了。
第 20 章  μC/OS-III 的启动流程

在目前的 RTOS 中，主要有两种比较流行的启动方式，暂时还没有看到第三种，接下来我将通过伪代码的方式来讲解下这两种启动方式的区别，然后再具体分析下 μC/OS 的启动流程。

20.1 万事俱备，只欠东风

第一种我称之为万事俱备，只欠东风法。这种方法是在 main() 函数中将硬件初始化，RTOS 系统初始化，所有任务的创建这些都弄好，这个我称之为万事都已经准备好。最后只欠一道东风，即启动 RTOS 的调度器，开始多任务的调度，具体的伪代码实现见代码清单: 启动-1。

列表 1: 代码清单: 启动-1 万事俱备，只欠东风法伪代码

实现

```c
int main (void)
{

    /* 硬件初始化 */
    HardWare_Init();(1)

    /* RTOS 系统初始化 */
    RTOS_Init();(2)

    /* 创建任务 1，但任务 1 不会执行，因为调度器还没有开启 */(3)
    RTOS_TaskCreate(Task1);

    /* 创建任务 2，但任务 2 不会执行，因为调度器还没有开启 */
    RTOS_TaskCreate(Task2);

    /* ....... 继续创建各种任务 */
}
```

(下页继续)
/* 启动 RTOS，开始调度 */
RTOS_Start();(4)
}

voidTask1( void *arg )(5)
{
    while (1)
    {
        /* 任务实体，必须有阻塞的情况出现 */
    }
}

voidTask1( void *arg )(6)
{
    while (1)
    {
        /* 任务实体，必须有阻塞的情况出现 */
    }
}

• 代码清单: 启动-1 (1): 硬件初始化。硬件初始化这一步还属于裸机的范畴，我们可以把需要使用到的硬件都初始化好而且测试好，确保无误。

• 代码清单: 启动-1 (2): RTOS 系统初始化。比如 RTOS 里面的全局变量的初始化，空闲任务的创建等。不同的 RTOS，它们的初始化有细微的差别。

• 代码清单: 启动-1 (3): 创建各种任务。这里把所有要用到的任务都创建好，但还不会进入调度，因为这个时候 RTOS 的调度器还没有开启。

• 代码清单: 启动-1 (4): 启动 RTOS 调度器，开始任务调度。这个时候调度器就从刚刚创建好的任务中选择一个优先级最高的任务开始运行。

• 代码清单: 启动-1 (5)(6): 任务实体通常是一个不带返回值的无限循环的 C 函数，函数体必须有阻塞的情况出现，不然任务（如果优先权恰好是最高）会一直在 while 循环里面执行,
导致其他任务没有执行的机会。

20.2 小心翼翼，十分谨慎

第二种我称之为小心翼翼，十分谨慎法。这种方法是在 main() 函数中将硬件和 RTOS 系统先初始化好，然后创建一个启动任务后就启动调度器，然后在启动任务里面创建各种应用任务，当所有任务都创建成功后，启动任务把自己删除，具体的伪代码实现见代码清单: 启动-2。

列表 2: 代码清单: 启动-2 小心翼翼，十分谨慎法伪代码实现

```c
int main (void)
{
    /* 硬件初始化 */
    HardWare_Init();(1)

    /* RTOS 系统初始化 */
    RTOS_Init();(2)

    /* 创建一个任务 */
    RTOS_TaskCreate(AppTaskCreate);(3)

    /* 启动 RTOS，开始调度 */
    RTOS_Start();(4)
}

/* 启始任务，在里面创建任务 */
void AppTaskCreate( void *arg )(5)
{
    /* 创建任务 1，然后执行 */
    RTOS_TaskCreate(Task1);(6)
}
```

(下页继续)
/* 当任务 1 阻塞时，继续创建任务 2，然后执行 */
RTOS_TaskCreate(Task2);

/* ....... 继续创建各种任务 */

/* 当任务创建完成，删除起始任务 */
RTOS_TaskDelete(AppTaskCreate); (7)
}

void Task1( void *arg ) (8)
{
    while (1)
    {
        /* 任务实体，必须有阻塞的情况出现 */
    }
}

void Task2( void *arg ) (9)
{
    while (1)
    {
        /* 任务实体，必须有阻塞的情况出现 */
    }
}

• 代码清单: 启动-2 (1): 硬件初始化。来到硬件初始化这一步还属于裸机的范畴，我们可以把需要使用到的硬件都初始化好而且测试好，确保无误。
• 代码清单: 启动-2 (2): RTOS 系统初始化。比如 RTOS 里面的全局变量的初始化，空闲任务的创建等。不同的 RTOS，它们的初始化有细微的差别。
• 代码清单: 启动-2 (3): 创建一个开始任务。然后在这个初始任务里面创建各种应用任务。
• 代码清单: 启动-2 (4): 启动 RTOS 调度器，开始任务调度。这个时候调度器就去执行刚刚创建好的初始任务。

• 代码清单: 启动-2 (5): 我们通常说任务是一个不带返回值的无限循环的 C 函数，但是因为初始任务的特殊性，它不能是无限循环的，只执行一次后就关闭。在初始任务里面我们创建我们需要的各种任务。

• 代码清单: 启动-2 (6): 创建任务。每创建一个任务后它都将进入就绪态，系统会进行一次调度，如果新创建的任务的优先级比初始任务的优先级高的话，那将去执行新创建的任务，当新的任务阻塞时再回到初始任务被打断的地方继续执行。反之，则继续往下创建新的任务，直到所有任务创建完成。

• 代码清单: 启动-2 (7): 各种应用任务创建完成后，初始任务自己关闭自己，使命完成。

• 代码清单: 启动-2 (8)(9): 任务实体通常是一个不带返回值的无限循环的 C 函数，函数体必须有阻塞的情况出现，不然任务（如果优先权恰好是最高的）会一直在 while 循环里面执行，其他任务没有执行的机会。

20.3 熟优孰劣

那有关这两种方法孰优孰劣? 我暂时没发现，我个人还是比较喜欢使用第二种。COS 和 LiteOS 第一种和第二种都可以使用，由用户选择，RT-Thread 和 FreeRTOS 则默认使用第二种。接下来我们详细讲解下 μC/OS 的启动流程。

20.4 系统的启动

我们知道，在系统上电的时候第一个执行的是启动文件里面由汇编编写的复位函数 Reset_Handler，具体见代码清单: 启动-3。复位函数的最后会调用 C 库函数 _main，具体见代码清单: 启动-3 的加粗部分。_main() 函数的主要工作是初始化系统的堆和栈，最后调用 C 中的 main() 函数，从而去到 C 的世界。

论坛: https://www.firebbs.cn/ 297 天猫: https://yehuosm.tmall.com
列表 3: 代码清单：启动-3Reset_Handler 函数

```assembly
Reset_Handler  PROC
EXPORT  Reset_Handler          [WEAK]
IMPORT  __main
IMPORT  SystemInit
LDRR0, =SystemInit
    BLX  R0
LDRR0,=__main
    BX  R0
ENDP
```

20.4.1 系统初始化

在调用创建任务函数之前，我们必须要对系统进行一次初始化，而系统的初始化是根据我们配置
宏定义进行初始化的，有一些则是系统必要的初始化，如空闲任务，时钟节拍任务等，下面我们
来看看系统初始化的源码，具体见代码清单：启动-4。

列表 4: 代码清单：启动-4 系统初始化（已删减）

```c
void  OSInit (OS_ERR   *p_err)
{
    CPU_STK    *p_stk;
    CPU_STK_SIZE  size;

    if  (p_err == (OS_ERR  *))
    {
        OS_SAFETY_CRITICAL_EXCEPTION();
        return;
    }
```

(下页继续)
OSInitHook();  // 初始化钩子函数相关的代码 */

OSIntNestingCtr = (OS_NESTING_CTR)0;  // 清除中断嵌套计数器 */

OSRunning = OS_STATE_OS_STOPPED;  // 未启动多任务处理 */

OSSchedLockNestingCtr = (OS_NESTING_CTR)0;  // 清除锁定计数器 */

OSTCBCurPtr = (OS_TCB *)0;  // 将 OS_TCB 指针初始化为已知状态 */

OSTCBHighRdyPtr = (OS_TCB *)0;

OSPrioCur = (OS_PRIO)0;  // 将优先级变量初始化为已知状态 */

OSPrioHighRdy = (OS_PRIO)0;

OSPrioSaved = (OS_PRIO)0;

if (OSCfg_ISRStkSize > (CPU_STK_SIZE))
{
    p_stk = OSCfg_ISRStkBasePtr;  // 清除异常栈以进行栈检查 */
    if (p_stk != (CPU_STK *)0)
    {
        size = OSCfg_ISRStkSize;
        while (size > (CPU_STK_SIZE))
        {
            size--;
            *p_stk = (CPU_STK)0;
            p_stk++;
        }
    }
}

OS_PrioInit();  // 初始化优先级位图表 */

(下页继续)
OS_RdyListInit();    /* 初始化就绪列表 */

OS_TaskInit(p_err);   /* 初始化任务管理器 */
if (*p_err != OS_ERR_NONE)
{
    return;
}

OS_IdleTaskInit(p_err);   /* 初始化空闲任务 */    (1)
if (*p_err != OS_ERR_NONE)
{
    return;
}

OS_TickTaskInit(p_err);   /* 初始化时钟节拍任务 */    (2)
if (*p_err != OS_ERR_NONE)
{
    return;
}

OSCfg_Init();

在这个系统初始化中，我们主要看两个地方，一个是空闲任务的初始化，一个是时钟节拍任务的初始化，这两个任务是必须存在的任务，否则系统无法正常运行。
20.4.1.1 空闲任务

- 代码清单: 启动-4 (1): 其实初始化就是创建一个空闲任务，空闲任务的相关信息由系统默认指定，用户不能修改，OS_IdleTaskInit() 源码具体见代码清单: 启动-5。

列表 5: 代码清单: 启动-5 OS_IdleTaskInit() 源码

```c
void OS_IdleTaskInit (OS_ERR *p_err)
{
    #ifdef OS_SAFETY_CRITICAL
        if (p_err == (OS_ERR *)0)
        {
            OS_SAFETY_CRITICAL_EXCEPTION();
            return;
        }
    #endif

    OSIdleTaskCtr = (OS_IDLE_CTR)0;  (1)

    /* ---------------- CREATE THE IDLE TASK ---------------- */
    OSTaskCreate((OS_TCB *)&OSIdleTaskTCB,
                  (CPU_CHAR *)((void *)"μC/OS-III Idle Task"),
                  (OS_TASK_PTR)OS_IdleTask,
                  (void *)0,
                  (OS_PRIO ) (OS_CFG_PRIO_MAX - 1u),
                  (CPU_STK  *)OSCfg_IdleTaskStkBasePtr,
                  (CPU_STK_SIZE)OSCfg_IdleTaskStkLimit,
                  (CPU_STK_SIZE)OSCfg_IdleTaskStkSize,
                  (OS_MSG_QTY ) 0u,
                  (OS_TICK  ) 0u,
                  (void *)0,
                  (OS_OPT) (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR | OS_OPT_TASK_NO_TLS),
                  (OS_ERR  *)p_err);  (2)
```

(下页继续)
代码清单: 启动-5 (1): OSIdleTaskCtr 在 os.h 头文件中定义, 是一个 32 位无符号整型变量, 该变量的作用是用于统计空闲任务的运行的, 怎么统计呢, 我们在空闲任务中讲解。现在初始化空闲任务，系统就将 OSIdleTaskCtr 清零。

代码清单: 启动-5 (2): 我们可以很容易看到系统只是调用了 OSTaskCreate() 函数来创建一个任务, 这个任务就是空闲任务, 任务优先级为 OS_CFG_PRIO_MAX-1, OS_CFG_PRIO_MAX 是一个宏, 该宏定义表示 μC/OS 的任务优先级数值的最大值, 我们知道, 在 μC/OS 系统中, 任务的优先级数值越大, 表示任务的优先级越低, 所以空闲任务的优先级是最低的。空闲任务栈大小为 OSCfg_IdleTaskStkSize, 它也是一个宏, 在 os_cfg_app.c 文件中定义，默认为 128, 则空闲任务栈默认为 128*4=512 字节。

空闲任务其实就是一个函数，其函数入口是 OS_IdleTask，源码具体见代码清单: 启动-6。

列表 6: 代码清单: 启动-6 OS_IdleTask() 源码

```c
void OS_IdleTask (void *p_arg)
{
    CPU_SR_ALLOC();

    /* Prevent compiler warning for not using 'p_arg'*/
    p_arg = p_arg;

    while (DEF_ON)
    {
        CPU_CRITICAL_ENTER();
        OSIdleTaskCtr++;
        #if OS_CFG_STAT_TASK_EN > 0u
        OSStatTaskCtr++;
        #endif
        CPU_CRITICAL_EXIT();
    }
```
空闲任务的作用还是很大的，它是一个无限的死循环，因为其优先级是最低的，所以任何优先级比它高的任务都能抢占它从而取得 CPU 的使用权，为什么系统要空闲任务呢？因为 CPU 是不会停下来的，即使啥也不干，CPU 也不会停下来，此时系统就必须保证有一个随时处于就绪态的任务，而且这个任务不会抢占其他任务，当且仅当系统的其他任务处于阻塞中，系统才会运行空闲任务，这个任务可以做很多事情，任务统计，钩入用户自定义的钩子函数实现用户自定义的功能等，但是需要注意的是，在钩子函数中用户不允许调用任何可以使空闲任务阻塞的函数接口，空闲任务是不允许被阻塞的。

- 代码清单: 启动-4 (2): 同样的，OS_TickTaskInit() 函数也是创建一个时钟节拍任务，具体见代码清单: 启动-7。

列表 7: 代码清单: 启动-7 OS_TickTaskInit() 源码

```c
void OS_TickTaskInit (OS_ERR *p_err)
{
    #ifdef OS_SAFETY_CRITICAL
        if (p_err == (OS_ERR *)0)
            {
                OS_SAFETY_CRITICAL_EXCEPTION();
                return;
            }
    #endif

    OSTickCtr = (OS_TICK)0u;  /* Clear the tick counter */

    OSTickTaskTimeMax = (CPU_TS)0u;
}
```
OS_TickListInit();/* Initialize the tick list data structures */

/* ---------------- CREATE THE TICK TASK ---------------- */
if (OSCfg_TickTaskStkBasePtr == (CPU_STK *)0)
{
    *p_err = OS_ERR_TICK_STK_INVALID;
    return;
}

if (OSCfg_TickTaskStkSize < OSCfg_StkSizeMin)
{
    *p_err = OS_ERR_TICK_STK_SIZE_INVALID;
    return;
}

/* Only one task at the 'Idle Task' priority */
if (OSCfg_TickTaskPrio >= (OS_CFG_PRIO_MAX - 1u))
{
    *p_err = OS_ERR_TICK_PRIO_INVALID;
    return;
}

OSTaskCreate((OS_TCB *)&OSTickTaskTCB,
            (CPU_CHAR (*)((void *)"μC/OS-III Tick Task")),
            (OS_TASK_PTR )OS_TickTask,
            (void *)0,
            (OS_PRIO )OSCfg_TickTaskPrio,
            (CPU_STK *)OSCfg_TickTaskStkBasePtr,
            (CPU_STK_SIZE)OSCfg_TickTaskStkLimit,
            (CPU_STK_SIZE)OSCfg_TickTaskStkSize,
            (OS_MSG_QTY )0u,
            (OS_TICK )0u,
当然啦，系统的初始化远远不止这两个任务，系统的其他资源也是会进行初始化的，我们在这里就暂时不讲解，有兴趣的图像可以自行查看系统初始化的源码。

### 20.4.2 CPU 初始化

在 main() 函数中，我们除了需要对板级硬件进行初始化，还需要进行一些系统相关的初始化，如 CPU 的初始化，在 μC/OS 中，有一个很重要的功能就是时间截，它的精度高达 ns 级别，是 CPU 内核的一个资源，所以使用的时候要对 CPU 进行相关的初始化，具体见代码清单: 启动-8。

#### 列表 8: 代码清单: 启动-8CPU 初始化源码

```
void CPU_Init (void)
{
/* --------------------- INIT TS ---------------------- */
#if ((CPU_CFG_TS_EN == DEF_ENABLED) ||
     (CPU_CFG_TS_TMR_EN == DEF_ENABLED))
   CPU_TS_Init(); /* 时间截测量的初始化 */
#endif
/* -------------- INIT INT DIS TIME MEAS -------------- */
#ifdef CPU_CFG_INT_DIS_MEAS_EN
   CPU_IntDisMeasInit(); /* 最大关中断时间测量初始化 */
#endif
```

/* ------------------ INIT CPU NAME ------------------- */
#if (CPU_CFG_NAME_EN == DEF_ENABLED)
    CPU_NameInit();       //CPU 名字初始化
#endif

我们重点来介绍一下 μC/OS 的时间戳。

在 Cortex-M（注意：M0 内核不可用）内核中有一个外设叫 DWT(Data Watchpoint and Trace)，是用于系统调试及跟踪，它有一个 32 位的寄存器叫 CYCCNT，它是一个向上的计数器，记录的是内核时钟运行的个数，内核时钟跳动一次，该计数器就加 1，当 CYCCNT 溢出之后，会清零重新开始向上升数。CYCCNT 的精度非常高，其精度取决于内核的频率是多少，如果是 STM32F1 系列，内核时钟是 72M，那精度就是 1/72M=14ns，而程序的运行时间都是微秒级别的，所以 14ns 的精度是远远不够的。最长能记录的时间为：60s=2 的 32 次方/72000000(假设内核频率为 72M，内核跳一次的时间大概为 1/72M=14ns)，而如果是 STM32H7 系列这种 400M 主频的芯片，那它的时钟精度高达 2.5ns (1/400000000 = 2.5)，而如果是 i.MX RT1052 这种比较厉害的处理器，最长能记录的时间为：8.13s=2 的 32 次方/528000000(假设内核频率为 528M，内核跳一次的时间大概为 1/528M=1.9ns)。

想要启用 DWT 外设，需要由另外的内核调试寄存器 DEMCR 的位 24 控制，写 1 启用，DEMCR 的地址是 0xE000EDFC。
启用 DWT_CYCCNT 寄存器之前，先清零。让我们看看 DWT_CYCCNT 的基地址，从 ARM-Cortex-M 手册中可以看到其基地址是 0xE000 1004，复位默认值是 0，而且它的类型是可读可写的，我们往 0xE000 1004 这个地址写 0 就将 DWT_CYCCNT 清零了。
关于 CYCCNTENA，它是 DWT 控制寄存器的第一位，写 1 启用，则启用 CYCCNT 计数器，否则 CYCCNT 计数器将不会工作，它的地址是 0xE000EDFC。
所以想要使用 DWT 的 CYCCNT 步骤：

1. 先启用 DWT 外设，这个由另外内核调试寄存器 DEMCR 的位 24 控制，写 1 启用

2. 在启用 CYCCNT 寄存器之前，先清零。

3. 启用 CYCCNT 寄存器，这个由 DWT 的 CYCCNTENA 控制，也就是 DWT 控制寄存器的位 0 控制，写 1 启用

这样子，我们就能去看看 μC/OS 的时间戳的初始化了，具体见 代码清单: 启动-9

列表 9: 代码清单: 启动-9 CPU_TS_TmrInit() 源码

```c
#define DWT_CR *(CPU_REG32 *)0xE0001000
#define DWT_CYCCNT *(CPU_REG32 *)0xE0001004
#define DEM_CR *(CPU_REG32 *)0xE000EDFC
```

(下页继续)
# define DEM_CR_TRCENA
           (1 << 24)

# define DWT_CR_CYCCNTENA
           (1 << 0)

#if (CPU_CFG_TS_TMR_EN == DEF_ENABLED)
void CPU_TS_TmrInit (void)
{
    CPU_INT32U cpu_clk_freq_hz;

    /* Enable Cortex-M3's DWT CYCCNT reg. */
    DEM_CR |= (CPU_INT32U)DEM_CR_TRCENA;

    DWT_CYCCNT = (CPU_INT32U)0u;
    DWT_CR |= (CPU_INT32U)DWT_CR_CYCCNTENA;

    cpu_clk_freq_hz = BSP_CPU_ClkFreq();
    CPU_TS_TmrFreqSet (cpu_clk_freq_hz);
}
#endif

## 20.4.3 SysTick 初始化

时钟节拍的频率表示操作系统每 1 秒钟产生多少个 tick。tick 即是操作系统节拍的时钟周期，时钟节拍就是系统以固定的频率产生中断（时基中断），并在中断中处理与时间相关的事件，推动所有任务向前运行。时钟节拍需要依赖于硬件定时器，在 STM32 裸机程序中经常使用的 SysTick 时钟是 MCU 的内核定时器，通常都使用该定时器产生操作系统的时钟节拍。用户需要先在 “os_cfg_app.h” 中设定时钟节拍的频率，该频率越高，操作系统检测事件就越频繁，可以增强任务的实时性，但太频繁也会增加操作系统内核的负荷加重，所以用户需要权衡该频率的设置。我们在这里采用默认的 1000Hz（本书之后若无特别声明，均采用 1000 Hz），也就是时钟节拍的周期为 1 ms。
函数 OS_CPU_SysTickInit() 用于初始化时钟节拍中断，初始化中断的优先级，SysTick 中断的启用等等，这个函数要跟不同的 CPU 进行编写，并且在系统任务的第一个任务开始的时候进行调用，如果在此之前进行调用，可能会造成系统奔溃，因为系统还没有初始化好就进入中断，可能在进入和退出中断的时候会调用系统未初始化好的一些模块，具体见代码清单: 启动-10。

### 列表 10: 代码清单: 启动-10 SysTick 初始化

```c
1  cpu_clk_freq = BSP_CPU_ClkFreq(); /* Determine SysTick reference freq. */
2  cnts = cpu_clk_freq / (CPU_INT32U)OSCfg_TickRate_Hz;
3  OS_CPU_SysTickInit(cnts); /* Init μC/OS periodic time src (SysTick). */
```

#### 20.4.4 内存初始化

我们都知道，内存存在嵌入式中是很珍贵的存在，而一个系统它是软件，则必须要有一块内存属于系统所管理的，所以在系统创建任务之前，就必须将系统必要的东西进行初始化，μC/OS 采用一块连续的大数组作为系统管理的内存，CPU_INT08U Mem_Heap[LIB_MEM_CFG_HEAP_SIZE]，在使用之前就需要先将管理的内存进行初始化，具体见代码清单: 启动-11。

### 列表 11: 代码清单: 启动-11 内存初始化

```c
1  Mem_Init(); /* Initialize Memory Management Module */
```

#### 20.4.5 OSStart()

在创建完任务的时候，我们需要开启调度器，因为创建仅仅只是把任务添加到系统中，还没真正调度，那怎么才能让系统支持运行呢，μC/OS 为我们提供一个系统启动的函数接口——OSStart()，我们使用 OSStart() 函数就能让系统开始运行，具体见代码清单: 启动-12。

论坛: https://www.firebbs.cn/  
天猫: https://yehuosm.tmall.com
列表 12: 代码清单: 启动-12vTaskStartScheduler() 函数

```
void OSStart (OS_ERR *p_err)
{
    #ifdef OS_SAFETY_CRITICAL
        if (p_err == (OS_ERR *)0) {
            OS_SAFETY_CRITICAL_EXCEPTION();
            return;
        }
    #endif

    if (OSRunning == OS_STATE_OS_STOPPED) {
        OSPrioHighRdy = OS_PrioGetHighest(); /* Find the highest priority...
                                            */
        OSPrioCur     = OSPrioHighRdy;
        OSTCBHighRdyPtr = OSRdyList[OSPrioHighRdy].HeadPtr;
        OSTCBCurPtr   = OSTCBHighRdyPtr;
        OSRunning     = OS_STATE_OS_RUNNING;
        OSStartHighRdy(); /* Execute target specific code to start task */
        *p_err         = OS_ERR_FATAL_RETURN;
        /* OSStart() is not supposed to return */
    }
    else
    {
        *p_err         = OS_ERR_OS_RUNNING; /* OS is already running */
    }
}
```

关于任务切换的详细过程在第一部分已经讲解完毕，此处就不再重复赘述。
当我们拿到一个移植好μC/OS的例程的时候，不出意外，你首先看到的是main()函数，当你认真一看main()函数里面只是让系统初始化和硬件初始化，然后创建并启动一些任务，具体见代码清单: 启动-13。因为这样子高度封装的函数让我们使用起来非常方便，防止用户一不小心忘了初始化系统的某些必要资源，造成系统启动失败，而作为用户，如果只是单纯使用μC/OS的话，无需太过于关注μC/OS接口函数里面的实现过程，但是我们还是建议需要深入了解μC/OS然后再去使用，避免出现问题。

列表13: 代码清单: 启动-13 main() 函数

```c
int main (void)
{
    OS_ERR err;
    OSInit(&err);    /* Init μC/OS-III. */
    OSTaskCreate((OS_TCB *) &AppTaskStartTCB, /*Create the start task*/
                 (CPU_CHAR *) "App Task Start",
                 (OS_TASK_PTR) AppTaskStart,
                 (void *) 0,
                 (OS_PRIO) APP_TASK_START_PRIO,
                 (CPU_STK *) &AppTaskStartStk[0],
                 (CPU_STK_SIZE) APP_TASK_START_STK_SIZE / 10,
                 (CPU_STK_SIZE) APP_TASK_START_STK_SIZE,
                 (OS_MSG_QTY) 5u,
                 (OS_TICK) 0u,
                 (void *) 0,
                 (OS_OPT) (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
                 (OS_ERR) &err);
    /*Start multitasking (i.e. give control to μC/OS-III)*/
    OSStart(&err);
}
```
• 代码清单: 启动-13 (1): 系统初始化完成，就创建一个 AppTaskStart 任务，在 AppTaskStart 任务中创建各种应用任务，具体见代码清单: 启动-14。

列表 14: 代码清单: 启动-14 AppTaskCreate 函数

```
static void AppTaskStart (void *p_arg)
{
    CPU_INT32U cpu_clk_freq;
    CPU_INT32U cnts;
    OS_ERR err;

    (void)p_arg;

    BSP_Init(); /* Initialize BSP functions */

    CPU_Init();

    cpu_clk_freq = BSP_CPU_ClkFreq(); /* Determine SysTick reference freq */
    /* Determine nbr SysTick increments */
    cnts = cpu_clk_freq / (CPU_INT32U)OSCfg_TickRate_Hz;

    OS_CPU_SysTickInit(cnts); /* Init μC/OS periodic time src (SysTick) */

    Mem_Init(); /* Initialize Memory Management Module */

#if OS_CFG_STAT_TASK_EN > 0u
    /* Compute CPU capacity with no task running*/
    OSStatTaskCPUUsageInit(&err);
#endif
```

CPU_IntDisMeasMaxCurReset();

OSTaskCreate((OS_TCB *)&AppTaskLed1TCB, "Create the Led1 task",  
(CPU_CHAR *)"App Task Led1",  
(OS_TASK_PTR ) AppTaskLed1,  
(void *) 0,  
(OS_PRIO ) APP_TASK_LED1_PRIO,  
(CPU_STK *)&AppTaskLed1Stk[0],  
(CPU_STK_SIZE) APP_TASK_LED1_STK_SIZE / 10,  
(CPU_STK_SIZE) APP_TASK_LED1_STK_SIZE,  
(OS_MSG_QTY ) 5,  
(OS_TICK ) 0,  
(void *) 0,  
(OS_OPT ) (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),  
(OS_ERR *)&err);

OSTaskCreate((OS_TCB *)&AppTaskLed2TCB, "Create the Led2 task",  
(CPU_CHAR *)"App Task Led2",  
(OS_TASK_PTR ) AppTaskLed2,  
(void *) 0,  
(OS_PRIO ) APP_TASK_LED2_PRIO,  
(CPU_STK *)&AppTaskLed2Stk[0],  
(CPU_STK_SIZE) APP_TASK_LED2_STK_SIZE / 10,  
(CPU_STK_SIZE) APP_TASK_LED2_STK_SIZE,  
(OS_MSG_QTY ) 5,  
(OS_TICK ) 0,  
(void *) 0,  
(OS_OPT ) (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),  
(OS_ERR *)&err);

OSTaskCreate((OS_TCB *)&AppTaskLed3TCB, "Create the Led3 task",  
(CPU_CHAR *)"App Task Led3",  
(OS_TASK_PTR ) AppTaskLed3,  
(void *) 0,  
(OS_PRIO ) APP_TASK_LED3_PRIO,  
(CPU_STK *)&AppTaskLed3Stk[0],  
(CPU_STK_SIZE) APP_TASK_LED3_STK_SIZE / 10,  
(CPU_STK_SIZE) APP_TASK_LED3_STK_SIZE,  
(OS_MSG_QTY ) 5,  
(OS_TICK ) 0,  
(void *) 0,  
(OS_OPT ) (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),  
(OS_ERR *)&err);

(下页继续)
在 AppTaskStart 中创建的应用任务的优先级比 AppTaskStart 任务的优先级高、低或者相等时候，程序是如执行的？假如像我们代码一样在临界区创建任务，任务只能在退出临界区的时候才执行最高优先级任务。假如没使用临界区的话，就会分三种情况：

1. 应用任务的优先级比初始任务的优先级高，那创建完后立马去执行刚刚创建的应用任务，当应用任务被阻塞时，继续回到初始任务被打断的地方继续往下执行，直到所有应用任务创建完成，最后初始任务把自己删除，完成自己的使命；

2. 应用任务的优先级与初始任务的优先级一样，那创建完后根据任务的时间片来执行，直到所有应用任务创建完成，最后初始任务把自己删除，完成自己的使命；

3. 应用任务的优先级比初始任务的优先级低，那创建完后任务不会被执行，如果还有应用任务紧接着创建应用任务，如果应用任务的优先级出现了比初始任务高或者相等的情况，请参考 1 和 2 的处理方式，直到所有应用任务创建完成，最后初始任务把自己删除，完成自己的使命。

- 代码清单: 启动-13 (2): 在启动任务调度器的时候，假如启动成功的话，任务就不会有返回了，假如启动没成功，则通过 LR 寄存器指定的地址退出，在创建 AppTaskStart 任务的时
当任务栈对应 LR 寄存器指向是任务退出函数 OS_TaskReturn()，当系统启动没能成功的话，系统就不会运行。
第 21 章 任务管理

21.1 任务的基本概念

从系统角度看，任务是竞争系统资源的最小运行单元。μC/OS 是一个支持多任务的操作系统。在 μC/OS 中，任务可以使用或等待 CPU、使用内存空间等系统资源，并独立于其他任务运行，任何数量的任务可以共享同一个优先级，处于就绪态的多个相同优先级任务将会以时间片切换的方式共享处理器。

简而言之：μC/OS 的任务可认为是一系列独立任务的集合。每个任务在自己的环境中运行。在任何时刻，只有一个任务得到运行，μC/OS 调度器决定运行哪个任务。调度器会不断的启动、停止每个任务，宏观看上去所有的任务都在同时在执行。作为任务，不需要对调度器的活动有所了解，在任务切入切出时保存上下文环境（寄存器值、栈内容）是调度器主要的职责。为了实现这点，每个 μC/OS 任务都需要有自己的栈空间。当任务切出时，它的执行环境会被保存在该任务的栈空间中，这样当任务再次运行时，就能从栈中正确的恢复上次的运行环境。任务越多，需要的栈空间就越大，而一个系统能运行多少个任务，取决于系统的可用的 SRAM。

μC/OS 的可以为用户提供多个任务单独享有独立的栈空间，系统可用确定任务的状态，决定任务是否可以运行，同时还能运用内核的 IPC 通信资源，实现了任务之间的通信，帮助用户管理业务程序流程。这样用户可以将更多的精力投入到业务功能的实现中。

μC/OS 中的任务是抢占式调度机制，高优先级的任务可打断低优先级任务，低优先级任务必须在高优先级任务阻塞或结束后才能得到调度。同时 μC/OS 也支持时间片轮转调度方式，只不过时间片的调度是不允许抢占任务的 CPU 使用权。

任务通常会运行在一个死循环中，也不会退出，如果一个任务不再需要，可以调用 μC/OS 中的任务删除 API 函数接口显式地将其删除。
21.2 任务调度器的基本概念

μC/OS 中提供的任务调度器是基于优先级的全抢占式调度：在系统中除了中断处理函数、调度器上锁部分的代码和禁止中断的代码是不可抢占的之外，系统的其他部分都是可以抢占的。系统理论上可以支持无数个优先级 (0 ~ N，优先级数值越大的任务优先级越低，(OS_CFG_PRIO_MAX - 1)) 为最低优先级，分配给空闲任务使用，一般不建议用户来使用这个优先级。一般系统默认的最大可用优先级数目为 32。在一些资源比较紧张的系统中，用户可以根据实际情况选择只支持 8 个或自定义个数优先级的系统配置。在系统中，当有比当前任务优先级更高的任务就绪时，当前任务将立刻被切出，高优先级任务抢占处理器运行。

一个操作系统如果只是具备了高优先级任务能够“立即”获得处理器并得到执行的特点，那么它仍然不算是实时操作系统。因为这个查找最高优先级任务的过程决定了调度时间是否具有确定性，例如一个包含 n 个就绪任务的系统中，如果仅仅从头找到尾，那么这个时间将直接和 n 相关，而下一个就绪任务执行时间的长短将会极大的影响系统的实时性。

μC/OS 内核中采用两种方法寻找最高优先级的任务，第一种是通用的方法，因为 μC/OS 防止 CPU 平台不支持前导零指令，就采用 C 语言模仿前导零指令的效果实现了快速查找到最高优先级任务的方法。而第二种方法则是特殊方法，利用硬件计算前导零指令 CLZ，这样子一次就能知道哪一个优先级任务能够运行，这种调度算法比普通方法更快捷，但受限于平台（在 STM32 中我们就使用这种方法）。

如果分别创建了优先级 3、5、8 和 11 这四个任务，任务创建成功后，调用 CPU_CntLeadZeros() 可以计算出 OSPrioTbl[0] 第一个置 1 的位置有 3 个 0，那么这个 3 就是我们要查找的最高优先级，至于后面还有多少个位置 1 我们都不用管，只需要找到第一个 1 即可。

μC/OS 内核中也允许创建相同优先级的任务。相同优先级的任务采用时间片轮转方式进行调度（也就是说的分时调度器），时间片轮转调度仅在当前系统中无更高优先级就绪任务存在的情况下才有效。为了保证系统的实时性，系统尽最大可能地保证高优先级的任务得以运行。任务调度的原则是一旦任务状态发生了改变，并且当前运行的任务优先级小于优先级队列组中任务最高优先级时，立刻进行任务切换（除非当前系统处于中断处理程序中或禁止任务切换的状态）。
21.3 任务状态迁移

μC/OS 系统中的每一个任务都有多种运行状态，他们之间的转换关系是怎么样的呢？从运行态任务变成阻塞态，或者从阻塞态变成就绪态，这些任务状态是如何进行迁移的呢？下面就让我们一起了解一下任务状态迁移吧，具体见图任务状态迁移图。

- 任务状态迁移图 (1): 创建任务 → 就绪态 (Ready): 任务创建完成后进入就绪态，表明任务已准备就绪，随时可以运行，只等待调度器进行调度。
- 任务状态迁移图 (2): 就绪态 → 运行态 (Running): 发生任务切换时，就绪列表中最高优先级的任务被执行，从而进入运行态。
- 任务状态迁移图 (3): 运行态 → 就绪态: 有更高优先级任务创建或者恢复后，会发生任务调度，此刻就绪列表中最高优先级任务变为运行态，那么原先运行的任务由运行态变为就
绪态，依然在就绪列表中，等待最高优先级的任务运行完毕继续运行原来的任务（此处可以看作 CPU 使用权被更高优先级的任务抢占了）。

- 任务状态迁移图 (4): 运行态 → 阻塞态（或者称为挂起态 Suspended）：正在运行的任务发生阻塞（挂起、延时、读信号量等待）时，该任务会从就绪列表中删除，任务状态由运行态变成阻塞态，然后发生任务切换，运行就绪列表中当前最高优先级任务。

- 任务状态迁移图 (5): 阻塞态 → 就绪态：阻塞的任务被恢复后（任务恢复、延时时间超时、读信号量超时或读到信号量等），此时被恢复的任务会被加入就绪列表，从而由阻塞态变成就绪态；如果此时被恢复任务的优先级高于正在运行任务的优先级，则会发生任务切换，将该任务将再次转换任务状态，由就绪态变成运行态。

- 任务状态迁移图 (6)(7)(8): 就绪态、阻塞态、运行态 → 删除态 (Delete)：任务可以通过调用 OSTaskDel() API 函数都可以将处于任何状态的任务删除，被删除后的任务将不能再次使用，关于任务的资源都会被系统回收。

- 任务状态迁移图 (9): 删除态 → 就绪态：这就是创建任务的过程，一个任务将会从无到有，创建成功的任务可以参与系统的调度。

注意：此处的任务状态只是大致的任务状态而并非 μC/OS 的所有任务状态，下面会具体介绍 \u201c\u201d\u00a0μC/OS 中具体的任务的状态。

### 21.4 μC/OS 的任务状态

μC/OS 系统中的每一任务都有一种运行状态。系统初始化完成后，创建的任务就可以在系统中竞争一定的资源，由内核进行调度。

μC/OS 的任务状态通常分为以下几种:

- 就绪 (OS_TASK_STATE_RDY)：该任务在就绪列表中，就绪的任务已经具备执行的能力，只等待调度器进行调度，新创建的任务会初始化为就绪态。

- 延时 (OS_TASK_STATE_DLY)：该任务处于延时调度状态。

- 等待 (OS_TASK_STATE_PEND)：任务调用 OSQPend()、OSSemPend() 这类等待函数，系统就会设置一个超时时间让该任务处于等待状态，如果超时时间设置为 0，任务的状态，无

[论坛: https://www.firebbs.cn/] [天猫: https://yehuosm.tmall.com]
限期等下去，直到事件发生。如果超时时间为 N(N>0)，在 N 个时间内任务等待的事件或信号都没发生，就退出等待状态转为就绪状态。

- 运行 (Running)：该状态表明任务正在执行，此时它占用处理器，MC/OS 调度器选择运行的永远是处于最高优先级的就绪态任务，当任务被运行的一刻，它的任务状态就变成了运行态，其实运行态的任务也是处于就绪列表中的。

- 挂起 (OS_TASK_STATE_SUSPENDED)：任务通过调用 OSTaskSuspend() 函数能够挂起自己或其它任务，调用 OSTaskResume() 是使被挂起的任务回复运行的唯一的方法。挂起一任务意味着该任务再被恢复运行以前不能够取得 CPU 的使用权，类似强行暂停一个任务。

- 延时 + 挂起 (OS_TASK_STATE_DLY_SUSPENDED)：任务先产生一个延时，延时没结束的时候被其它任务挂起，挂起的效果叠加，当且仅当延时结束并且挂起被恢复了，该任务才能够再次运行。

- 等待 + 挂起 (OS_TASK_STATE_PEND_SUSPENDED)：任务先等待一个事件或信号的发生 (无限期等待)，还没等待到就被其它任务挂起，挂起的效果叠加，当且仅当任务等待到事件或信号并且挂起被恢复了，该任务才能够再次运行。

- 超时等待 + 挂起 (OS_TASK_STATE_PEND_TIMEOUT_SUSPENDED)：任务在指定时间内等待事件或信号的产生，但是任务已经被其它任务挂起。

- 删除 (OS_TASK_STATE_DEL)：任务被删除后的状态，任务被删除后将不再运行，除非重新创建任务。

### 21.5 常用的任务函数讲解

相信大家通过第一部分章节的学习，对任务创建以及任务调度的实现已然掌握了，下面就补充一些 MC/OS 提供给我们对任务操作的一些常用函数。
21.5.1 任务挂起函数 OS_TaskSuspend()

挂起指定任务。被挂起的任务绝不会得到 CPU 的使用权，不管该任务具有什么优先级。

任务可以通过调用 vTaskSuspend() 函数都可以将处于任何状态的任务挂起，被挂起的任务得不到 CPU 的使用权，也不会参与调度，它相对于调度器而言是不可见的，除非它从挂起状态中解除。任务挂起是经常使用的一个函数，需要使用的就必须将宏定义 OS_CFG_TASK_SUSPEND_EN 启用，这样在编译的时候才会包含 OS_TaskSuspend() 这个函数。下面一起看看任务挂起的源代码吧，具体见代码清单：任务管理-1。

列表 1：代码清单：任务管理-1 任务挂起函数
OS_TaskSuspend() 源码

```c
#if OS_CFG_TASK_SUSPEND_EN > 0u // 如果启用了函数 OSTaskSuspend()

void OS_TaskSuspend (OS_TCB *p_tcb, OS_ERR *p_err) // 任务控制块指针
{ // 返回错误类型
    CPU_SR_ALLOC(); // 使用到临界段（处在开中断状态）时必须用到该宏，该宏声明和
    // 定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
    // SR（临界段关中断时需保存 SR），开中断时将该值还原。

    CPU_CRITICAL_ENTER(); // 关中断
    if (p_tcb == (OS_TCB *)0) { // 如果 p_tcb 为空
        p_tcb = OSTCBCurPtr; // 挂起自身
    }

    if (p_tcb == OSTCBCurPtr) { // 如果是挂起自身
        if (OSSchedLockNestingCtr > (OS_NESTING_CTR)0) { // 如果调度器被锁
            CPU_CRITICAL_EXIT(); // 开中断
            *p_err = OS_ERR_SCHED_LOCKED; // 错误类型为“调度器被锁”
            return; // 返回，停止执行
        }
    }
}
```

(下页继续)
*p_err = OS_ERR_NONE;  //错误类型为“无错误”

switch (p_tcb->TaskState) {  //根据 p_tcb 的任务状态分类处理
    case OS_TASK_STATE_RDY:  //如果是就绪状态
        OS_CRITICAL_ENTER_CPU_EXIT();  //锁调度器，重开中断
        p_tcb->TaskState = OS_TASK_STATE_SUSPENDED;  //任务状态改为“挂起状态”
        p_tcb->SuspendCtr = (OS_NESTING_CTR)1;  //挂起前套数为 1
        OS_RdyListRemove(p_tcb);  //将任务从就绪列表移除
        OS_CRITICAL_EXIT_NO_SCHED();  //开调度器，不进行调度
        break;  //跳出

    case OS_TASK_STATE_DLY:  //如果是延时状态将改为“延时中被挂起”
        p_tcb->TaskState = OS_TASK_STATE_DLY_SUSPENDED;
        p_tcb->SuspendCtr = (OS_NESTING_CTR)1;  //挂起前套数为 1
        CPU_CRITICAL_EXIT();  //开中断
        break;  //跳出

    case OS_TASK_STATE_PEND:  //如果是无期限等待状态将改为“无期限等待中被挂起”
        p_tcb->TaskState = OS_TASK_STATE_PEND_SUSPENDED;
        p_tcb->SuspendCtr = (OS_NESTING_CTR)1;  //挂起前套数为 1
        CPU_CRITICAL_EXIT();  //开中断
        break;  //跳出

    case OS_TASK_STATE_PEND_TIMEOUT:  //如果是有限期限等待将改为“有限期限等待中被挂起”
        p_tcb->TaskState = OS_TASK_STATE_PEND_TIMEOUT_SUSPENDED;
        p_tcb->SuspendCtr = (OS_NESTING_CTR)1;  //挂起前套数为 1
        CPU_CRITICAL_EXIT();  //开中断
        break;  //跳出

    case OS_TASK_STATE_SUSPENDED:  //如果状态中有挂起


```c
    case OS_TASK_STATE_DLY_SUSPENDED:
    case OS_TASK_STATE_PEND_SUSPENDED:
    case OS_TASK_STATE_PEND_TIMEOUT_SUSPENDED:
        p_tcb->SuspendCtr++;  // 挂起计数加 1
        CPU_CRITICAL_EXIT();  // 开中断
        break;  // 跳出

    default:  // (11) // 如果任务状态超时超期
        CPU_CRITICAL_EXIT();  // 开中断
        *p_err = OS_ERR_STATE_INVALID;  // 错误类型为 "状态非法"
        return;  // 返回，停止执行
    }

    OSSched();  // (12) // 调度任务

#endif

• 代码清单: 任务管理-1 (1): 任务控制块指针，该指针指向要挂起的任务，也可以是任务自身，但是不能是空闲任务，空闲任务永远不允许挂起。

• 代码清单: 任务管理-1 (2): 用于存放返回错误代码，如果挂起任务失败，则返回对应的错误代码。

• 代码清单: 任务管理-1 (3): 如果传递进来的任务控制块指针是 NULL 或者是 0，则表明要挂起的任务是任务自身，将任务控制块的指针指向当前任务。

• 代码清单: 任务管理-1 (4): 如果的任务是当前任务，也就是挂起任务自身，那么需要判断一下调度器有没有被锁定，因为挂起任务自身之后，就肯定需要切换任务，而如果调度器被锁定的话，就无法切换任务了，所以会返回错误类型“调度器被锁”，然后退出。

• 代码清单: 任务管理-1 (5): 根据要挂起的任务状态分类处理，这样处理逻辑简单，更加方便快捷。

• 代码清单: 任务管理-1 (6): 如果任务处于就绪状态，那么该任务能直接挂起，但是接下来我们要操作进行列表，时间是不确定的，我们不能将中断关闭太久，这样子会影响系统对
```
论坛: https://www.firebbs.cn/  天猫: https://yehuosm.tmall.com
中断的响应，此时系统就会打开中断，但是系统又不想其他任务来影响我们操作就绪列表，所以系统还会锁定调度器，不进行任务切换，这样子就不会有任务打扰我们的操作了，然后将任务状态变挂起态，挂起次数为 1 次，然后调用 OS_RdyListRemove() 函数将任务从就绪列表移除，再打开调度器，然后跳出，最后才进行任务的调度。

- **代码清单: 任务管理-1 (7):** 如果任务当前处于延时状态，那么也能被挂起，任务状态将改为“延时中被挂起”状态，挂起次数为 1 次，然后打开中断，退出。

- **代码清单: 任务管理-1 (8):** 如果任务当前处于无限期等待状态，那么也能被挂起，任务状态将改为“无限期等待中被挂起”状态，挂起次数为 1 次，然后打开中断，退出。

- **代码清单: 任务管理-1 (9):** 如果任务当前处于有限期等待状态，那么也能被挂起，任务状态将改为“有限期等待中被挂起”状态，挂起次数为 1 次，然后打开中断，退出。

- **代码清单: 任务管理-1 (10):** 如果要挂起的任务本身就处于挂起态，那么再次挂起就要记录挂起的次数，将挂起的次数加一，然后打开中断，退出。

- **代码清单: 任务管理-1 (11):** 对于其他的任务状态，返回状态非法错误，然后退出。

- **代码清单: 任务管理-1 (12):** 进行一次任务调度。

注: 任务可以调用 OS_TaskSuspend() 这个函数来挂起任务自身，但是在挂起自身的时候会进行一次任务上下文切换，需要挂起自身就将任务控制块指针设置为 NULL 或 0 传递进来即可。无论任务是什么状态都可以被挂起，只要调用了 OS_TaskSuspend() 这个函数就会挂起成功，不论是挂起其他任务还是挂起任务自身。

任务的挂起与恢复函数在很多时候都是很有用的，比如我们想暂停某个任务运行一段时间，但是我们需要在其恢复的时候继续工作，那么删除任务是不可能的，因为删除了任务的话，任务的所有信息都是不可能恢复的了，删除是完完全全删除了，里面的资源都被系统释放掉，但是挂起任务就不会这样子，调用挂起任务函数，仅仅是将任务进入挂起态，其内部的资源都会保留下来，同时也不会参与系统中任务的调度，当调用恢复函数的时候，整个任务立即从挂起态进入就绪态，并且参与任务的调度，如果该任务的优先级是当前就绪态优先级最高的任务，那么立即会按照挂起前的任务状态继续执行该任务，从而达到我们需要的效果，注意，是继续执行，也就是说，挂起任务之前是什么状态，都会被系统保留下来，在恢复的瞬间，继续执行。这个任务函数的使用方法是很简单的，只需把任务句柄传递进来即可，OS_TaskSuspend() 会根据任务句柄的信息将对应的任务挂起，具体见 **代码清单: 任务管理-2 加粗部分**。
列表 2：代码清单·任务管理-2·任务挂起函数

OS_TaskSuspend() 使用实例

```c
/**************************** 任务句柄 ****************************/
/*
* 任务句柄是一个指针，用于指向一个任务，当任务创建好之后，它就具有了一个任务句柄
* 以后我们要想操作这个任务都需要通过这个任务句柄，如果是自身的任务操作自己，那么
* 这个句柄可以为 NULL。
*/
staticOS_TCB AppTaskLed1TCB; /* LED 任务句柄 */
static void KEY_Task(void* parameter)
{
    OS_ERR err;
    while (1) {
        if ( Key_Scan(KEY1_GPIO_PORT,KEY1_GPIO_PIN) == KEY_ON ) {
            /* KEY1 被按下 */
            printf("挂起 LED 任务！\n");
            OSTaskSuspend (AppTaskLed1TCB, & err );   /* 挂起 LED1 任务 */
        }
        OSTimeDly ( 20, OS_OPT_TIME_DLY, & err ); /* 延时 20 个 tick */
    }
}
```

21.5.2 任务恢复函数 OSTaskResume()

既然有任务的挂起，那么当然一样有恢复。不然任务怎么恢复呢，任务恢复就是让挂起的任务重新进入就绪状态，恢复的任务会保留挂起前的状态信息，在恢复的时候根据挂起时的状态继续运行。如果被恢复任务在所有就绪态任务中，处于最高优先级列表的第一位，那么系统将进行任务上下文的切换。下面一起看看任务恢复函数 OSTaskResume() 的源码，具体见代码清单·任务管理-3。
列表 3: 代码清单: 任务管理-3 任务恢复函数 OSTaskResume() 源码

```c
#if OS_CFG_TASK_SUSPEND_EN > 0u //如果启用了函数 OSTaskResume()

void OSTaskResume (OS_TCB *p_tcb, OS_ERR *p_err) //任务控制块指针
{ //返回错误类型

   CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必须用到该宏，该宏声明和
   //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
   //SR（临界段关中断需保存SR），开中断时将该值还原。

#ifdef OS_SAFETY_CRITICAL //如果启用了安全检测
   if (p_err == (OS_ERR *)0) { //如果 p_err 为空
      OS_SAFETY_CRITICAL_EXCEPTION(); //执行安全检测异常函数
      return; //返回，停止执行
   }
#endif

#ifdef OS_SAFETY_CRITICAL //如果启用了安全检测
   if (p_tcb == (OS_TCB *)0) && 
       (OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u) 
       //如果在中断中调用该函数
       *p_err = OS_ERR_TASK_RESUME_ISR; //错误类型为“在中断中恢复任务”
       return; //返回，停止执行
   #endif

CPU_CRITICAL_ENTER(); //关中断

#if OS_CFG_ARG_CHK_EN > 0u //如果启用了参数检测
   if ((p_tcb == (OS_TCB *)0) || //如果被恢复任务为空或是自身
       (p_tcb == OSTMBCurPtr)) { //开中断
      CPU_CRITICAL_EXIT();
   }
#endif
```

(下页继续)
30 *p_err = OS_ERR_TASK_RESUME_SELF;  //错误类型为“恢复自身”
31     return;                       //返回，停止执行
32 }
33 #endif
34
35 CPU_CRITICAL_EXIT();                //关中断
36
37 #if OS_CFG_ISR_POST_DEFERRED_EN > 0u(6)  //如果启用了中断延迟发布
38     if (OSIntNestingCtr > (OS_NESTING_CTR)0) {  //如果该函数在中断中被调用
39         OS_IntQPost((OS_OBJ_TYPE)OS_OBJ_TYPE_TASK_RESUME,
40         (void *)p_tcb,
41         (void *)0,
42         (OS_MSG_SIZE)0,
43         (OS_FLAGS )0,
44         (OS_OPT )0,
45         (CPU_TS )0,
46         (OS_ERR *p_err));  //把恢复任务命令发布到中断消息队列
47     return;                      //返回，停止执行
48 }
49 #endif
50 /* 如果禁用了中断延迟发布或不是在中断中调用该函数 */
51 OS_TaskResume(p_tcb, p_err);        //直接将任务 p_tcb 恢复 (7)
52 } #endif

- 代码清单: 任务管理-3 (1): 任务控制块指针，该指针指向要恢复的任务，与挂起任务不同的是，该指针不允许指向任务自身。

- 代码清单: 任务管理-3 (2): 用于存放返回错误代码，如果恢复任务失败，则返回对应的错误代码。

- 代码清单: 任务管理-3 (3): 如果启用了安全检测 (OS_SAFETY_CRITICAL) 这个宏定义，那么在编译代码的时候会包含安全检测。如果 p_err 指针为空，系统会执行安全检测异常函数 OS_SAFETY_CRITICAL_EXCEPTION()，然后退出。

论坛: https://www.firebbs.cn/  天猫: https://yehuosm.tmall.com
· 代码清单: 任务管理-3 (4): 如果禁用了中断延迟发布和中断中非法调用检测，那么在中断中恢复任务则是非法的，会直接返回错误类型为“在中断中恢复任务”，并且退出。而如果启用了中断延迟发布的话呢，就可以在中断中恢复任务，因为中断延迟发布的真正操作是在中断发布任务中。

· 代码清单: 任务管理-3 (5): 如果启用了参数检测（OS_CFG_ARG_CHK_EN）这个宏定义，如果被恢复任务为空或是自身，也是不允许的，会返回错误类型为“恢复自身”，并且退出操作。

· 代码清单: 任务管理-3 (6): 如果启用了中断延迟发布，并且如果该函数在中断中被调用，系统就会把恢复任务命令发布到中断消息队列中，唤醒中断发布任务，在任务中恢复指定任务，并且退出。

· 代码清单: 任务管理-3 (7): 如果禁用了中断延迟发布或不是在中断中调用该函数，直接调用 OS_TaskResume() 函数恢复任务，该函数源码具体见代码清单: 任务管理-4。

列表 4: 代码清单: 任务管理-4 OS_TaskResume() 源码

```c
#if OS_CFG_TASK_SUSPEND_EN > 0u; //如果启用了函数 OSTaskResume()

void OS_TaskResume (OS_TCB *p_tcb, //任务控制块指针
                  OS_ERR *p_err) //返回错误类型
{
    CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必须用到该宏，该宏声明和
    //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
    // SR (临界段关中断只需保存 SR)，开中断时将该值还原。
    CPU_CRITICAL_ENTER(); //关中断
    *p_err = OS_ERR_NONE; //错误类型为“无错误”
    switch (p_tcb->TaskState) { (1) //根据 p_tcb 的任务状态分类处理
        case OS_TASK_STATE_RDY: //如果状态中没有挂起状态
        case OS_TASK_STATE_DLY:
        case OS_TASK_STATE_PEND:
        case OS_TASK_STATE_PEND_TIMEOUT:
            CPU_CRITICAL_EXIT(); //开中断
            *p_err = OS_ERR_TASK_NOT_SUSPENDED; (2) //错误类型为“任务未被挂起”
    }
}
```

(下页继续)
break;  //跳出

case OS_TASK_STATE_SUSPENDED:  //如果是“挂起状态"
    OS_CRITICAL_ENTER_CPU_EXIT();  //锁调度器，重开中断
    p_tcb->SuspendCtr--;  //任务的挂起嵌套数减 1
    if (p_tcb->SuspendCtr == (OS_NESTING_CTR)0) {  //如果挂起前套数为 0
        p_tcb->TaskState = OS_TASK_STATE_RDY;  //修改状态为“就绪状态”
        OS_TaskRdy(p_tcb);  //把 p_tcb 插入就绪列表
    }
    OS_CRITICAL_EXIT_NO_SCHED();  //开调度器，不调度任务
    break;  //跳出

    case OS_TASK_STATE_DLY_SUSPENDED:  //如果是“延时中被挂起”
    p_tcb->SuspendCtr--;  //任务的挂起嵌套数减 1
    if (p_tcb->SuspendCtr == (OS_NESTING_CTR)0) {  //如果挂起前套数为 0
        p_tcb->TaskState = OS_TASK_STATE_DLY;  //修改状态为“延时状态”
    }
    CPU_CRITICAL_EXIT();  //开中断
    break;  //跳出

    case OS_TASK_STATE_PEND_SUSPENDED:  //如果是“无期限等待中被挂起”
    p_tcb->SuspendCtr--;  //任务的挂起嵌套数减 1
    if (p_tcb->SuspendCtr == (OS_NESTING_CTR)0) {  //如果挂起前套数为 0
        p_tcb->TaskState = OS_TASK_STATE_PEND;  //修改状态为“无期限等待状态”
    }
    CPU_CRITICAL_EXIT();  //开中断
    break;  //跳出

    case OS_TASK_STATE_PEND_TIMEOUT_SUSPENDED:  //如果是“有限期等待中被挂起”
    p_tcb->SuspendCtr--;  //任务的挂起嵌套数减 1
    if (p_tcb->SuspendCtr == (OS_NESTING_CTR)0) {  //如果挂起前套数为 0
    (下页继续)
p_tcb->TaskState = OS_TASK_STATE_PEND_TIMEOUT;
}
CPU_CRITICAL_EXIT(); //开中断
break; //跳出

default: //如果 p_tcb 任务状态超出预期
CPU_CRITICAL_EXIT(); //开中断
*p_err = OS_ERR_STATE_INVALID; //错误类型为“状态非法"
return; //跳出
}

OSSched(); //调度任务

• 代码清单: 任务管理-4 (1): 根据要挂起的任务状态分类处理, 这样处理逻辑简单, 更加方便快捷。

• 代码清单: 任务管理-4 (2): 如果要恢复的的任务状态中没有挂起状态, 那表示任务没有被挂起, 根本不需要恢复任务, 返回错误类型为“任务未被挂起”, 并且退出操作。

• 代码清单: 任务管理-4 (3): 如果要恢复的任务是单纯的挂起状态, 那么可以恢复任务。

• 代码清单: 任务管理-4 (4): 任务的挂起记录次数减 1, 如果挂起前次数为 0, 表示任务已经完全恢复了, 那么就可以参与系统的调度, 此时就要把任务添加到就绪列表中, 并且将任务的状态变为就绪状态, 操作完成之后就跳出 switch 语句, 打开中断但是不进行任务调度, 因为在最后面才会进行任务调度。

• 代码清单: 任务管理-4 (5): 如果任务在延时的时候被挂起了, 也可以进行恢复任务操作, 任务的挂起记录次数减 1, 如果挂起前次数为 0, 表示任务已经完全恢复了, 那就会恢复挂起前的状态——延时状态, 然后退出。

• 代码清单: 任务管理-4 (6): 同理, 如果任务在无期限等待的时候被挂起了, 也可以进行恢复任务操作, 任务的挂起记录次数减 1, 如果挂起前次数为 0, 表示任务已经完全恢复了,
那就会恢复挂起前的状态——无期限等待状态，然后退出。

- **代码清单: 任务管理-4 (7)**：如果任务在有期限等待的时候被挂起了，也可以进行恢复任务操作，任务的挂起记录次数减 1，如果挂起前次数为 0，表示任务已经完全恢复了，那就会恢复挂起前的状态——有期限等待状态，然后退出。

- **代码清单: 任务管理-4 (8)**：对于其他的任务状态，返回状态非法错误，然后退出。

- **代码清单: 任务管理-4 (9)**：进行一次任务调度。

OSTaskResume() 函数用于恢复挂起的任务。任务在挂起时候调用过多少次的 OS_TaskSuspend() 函数，那么就需要调用多少次 OSTaskResume() 函数才能将任务恢复运行，下面来看看任务恢复函数 OSTaskResume() 的使用实例，具体见 **代码清单: 任务管理-5** 加粗部分。

### 列表 5: 代码清单: 任务管理-5 任务恢复函数 OSTaskResume() 实例

```c
/*
 * 任务句柄是一个指针，用于指向一个任务，当任务创建好之后，它就具有了一个任务句柄
 * 以后我们只要想操作这个任务都需要通过这个任务句柄，如果是自身的任务操作自己，那么
 * 这个句柄可以为 NULL。
 */

staticOS_TCB AppTaskLed1TCB; /* LED 任务句柄 */

static void KEY_Task(void* parameter)
{
    OS_ERR err;
    while (1) {
        if ( Key_Scan(KEY2_GPIO_PORT, KEY2_GPIO_PIN) == KEY_ON ) {
            /* KEY2 被按下 */
            printf("恢复 LED 任务！\n");
            OSTaskResume ( &AppTaskLed1TCB, & err ); /* 恢复 LED 任务！ */
        }
        OSTimeDly ( 20, OS_OPT_TIME_DLY, & err ); /* 延时 20 个 tick */
    }
}
```

论坛: [https://www.firebbs.cn/](https://www.firebbs.cn/)  
天猫: [https://yehuosm.tmall.com](https://yehuosm.tmall.com)
21.5.3 删除任务函数 OSTaskDel()

OSTaskDel() 用于删除一个任务。当一个任务删除另外一个任务时，形参为要删除任务创建时返回的任务句柄，如果是删除自身，则形参为 NULL。要想使用该函数必须在 os_cfg.h 中把 OS_CFG_TASK_DEL_EN 宏定义配置为 1，删除的任务将从所有就绪、阻塞、挂起和事件列表中删除，任务删除函数 OSTaskDel() 源码具体见代码清单: 任务管理-6。

列表 6: 代码清单: 任务管理-6 任务删除函数
taskDelete() 源码

```c
#if OS_CFG_TASK_DEL_EN > 0u  
void OSTaskDel (OS_TCB *p_tcb,          // 目标任务控制块指针
                OS_ERR *p_err)        // 返回错误类型
{
    CPU_SR_ALLOC();  // 使用到临界段 (在关/开中断时) 时必须用到该宏，该宏声明和
    // 定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
    // SR (临界段关中断只需保存 SR)，开中断时将该值还原。

#ifdef OS_SAFETY_CRITICAL  
    if (p_err == (OS_ERR *)0) {  // 如果 p_err 为空
        OS_SAFETY_CRITICAL_EXCEPTION();  // 执行安全检测异常函数
        return;  // 返回，停止执行
    }
#endif

#if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u(1)  // 如果启用了中断中非法调用检测
    if (OSIntNestingCtr > (OS_NESTING_CTR)0) {  // 如果该函数在中断中被调用
        *p_err = OS_ERR_TASK_DEL_ISR;  // 错误类型为 “在中断中删除任务”
        return;  // 返回，停止执行
    }
#endif

if (p_tcb == &OSIdleTaskTCB) { (2)  // 如果目标任务是空闲任务

(下页继续)
```
*p_err = OS_ERR_TASK_DEL_IDLE;    //错误类型为“删除空闲任务”
return;                          //返回，停止执行
}

#if OS_CFG_ISR_POST_DEFERRED_EN > 0u(3) //如果启用了中断延迟发布
    if (p_tcb == &OSIntQTaskTCB) {
        *p_err = OS_ERR_TASK_DEL_INVALID; //错误类型为“非法删除任务”
        return;                           //返回，停止执行
    }
#endif

if (p_tcb == (OS_TCB *)0) {          //如果 p_tcb 为空
    CPU_CRITICAL_ENTER();            //关中断
    p_tcb = OSTCBCurPtr;             //目标任务设为自身
    CPU_CRITICAL_EXIT();             //开中断
}

OS_CRITICAL_ENTER();                //进入临界段
switch (p_tcb->TaskState) {         //根据目标任务的任务状态分类处理
        case OS_TASK_STATE_RDY:    //如果是就绪状态
            OS_RdyListRemove(p_tcb); //将任务从就绪列表移除
            break;                  //跳出

        case OS_TASK_STATE_SUSPENDED: //如果是挂起状态
            break;                  //直接跳出

        case OS_TASK_STATE_DLY:      //如果包含延时状态
            case OS_TASK_STATE_DLY_SUSPENDED: //如果包含延时状态
                OS_TickListRemove(p_tcb); //将任务从节拍列表移除
                break;                  //跳出

        case OS_TASK_STATE_PEND:     //如果包含等待状态
            break;                  //跳跃

(下页继续)
case OS_TASK_STATE_PEND_SUSPENDED:
case OS_TASK_STATE_PEND_TIMEOUT:
case OS_TASK_STATE_PEND_TIMEOUT_SUSPENDED:
    OS_TickListRemove(p_tcb);  //将任务从节拍列表移除
switch (p_tcb->PendOn) {  //根据任务的等待对象分类处理
    case OS_TASK_PEND_ON NOTHING:  //如果没在等待内核对象
    case OS_TASK_PEND_ON_TASK_Q:  //如果等待的是任务消息队列
    case OS_TASK_PEND_ON_TASK_SEM:  //如果等待的是任务信号量
            break;  //直接跳出

    case OS_TASK_PEND_ON_FLAG:  //如果等待的是事件
    case OS_TASK_PEND_ON_MULTI:  //如果等待多个内核对象
    case OS_TASK_PEND_ON_MUTEX:  //如果等待的是互斥量
    case OS_TASK_PEND_ON_Q:  //如果等待的是消息队列
    case OS_TASK_PEND_ON_SEM:  //如果等待的是信号量
            OS_PendListRemove(p_tcb);  //将任务从等待列表移除
            break;  //跳出

    default:  //如果等待对象超出预期
            break;  //直接跳出

            break;  //跳出

default:  //如果目标任务状态超出预期
    OS_CRITICAL_EXIT();  //退出临界段
    *p_err = OS_ERR_STATE_INVALID;  //错误类型为“状态非法”
    return;  //返回，停止执行
}  //}

#if OS_CFG_TASK_Q_EN > 0u  //如果启用了任务消息队列
    (void) OS_MsgQFreeAll(&p_tcb->MsgQ);  //释放任务的所有任务消息
#endif
OSTaskDelHook(p_tcb); (15) // 调用用户自定义的钩子函数

#if defined(OS_CFG_TLS_TBL_SIZE) && (OS_CFG_TLS_TBL_SIZE > 0u)
  OS_TLS_TaskDel(p_tcb); /* Call TLSk */
#endif

#if OS_CFG_DBG_EN > 0u(16) // 如果启用了调试代码和变量
  OS_TaskDbgListRemove(p_tcb); // 将任务从任务调试双向列表移除
#endif

OSTaskQty--; (17) // 任务数目减 1

OS_TaskInitTCB(p_tcb); (18) // 初始化任务控制块
p_tcb->TaskState = (OS_STATE)OS_TASK_STATE_DEL; // 标定任务已被删除

OS_CRITICAL_EXIT_NO_SCHD(); // 退出临界段（无调度）

*p_err = OS_ERR_NONE; // 错误类型为 “无错误”

OSSched(); (19) // 调度任务

# endif

- **代码清单：任务管理-6 (1)**：如果启用了中断中非法调用检测，那么在中断中删除任务则是非法的，会直接返回错误类型为 “在中断中删除任务”，并且退出。

- **代码清单：任务管理-6 (2)**：如果要删除的目标任务是空闲任务，这是绝对不允许的，系统中空闲任务的存在是必然的，绝对不允许删除空闲任务，会返回错误类型为“删除空闲任务”的错误代码，并且退出。

- **代码清单：任务管理-6 (3)**：如果启用了中断延迟发布，但是要删除的目标任务是中断延迟发布任务，这也是绝对不允许的，因为启用了中断延迟发布，则代表着系统中必须有一个

论坛：https://www.firebbs.cn/  337  天猫：https://yehuosm.tmall.com
中断延迟发布任务处理在中断中的发布的事情，所以会返回错误类型为“非法删除任务”的错误代码，并且退出。

- **代码清单: 任务管理-6 (4):** 如果传递进来的任务控制块指针为 0，表示要删除的任务是任务自身，将任务控制块指针指向当前任务，目标任务设为任务自身。

- **代码清单: 任务管理-6 (5):** 根据目标任务的任务状态分类处理。

- **代码清单: 任务管理-6 (6):** 如果任务是处于就绪态的，就将任务从就绪列表移除。

- **代码清单: 任务管理-6 (7):** 如果任务是处于挂起状态就直接跳出 switch 语句。

- **代码清单: 任务管理-6 (8):** 如果任务包含延时状态，那么将任务从节拍列表移除。

- **代码清单: 任务管理-6 (9):** 如果任务包含等待状态。

- **代码清单: 任务管理-6 (10):** 系统首先会将任务从节拍列表移除。

- **代码清单: 任务管理-6 (11):** 然后根据任务的等待对象分类处理，如果没在等待内核对象或者等待的是任务消息队列或者等待的是任务信号量，那么直接跳出 switch 语句。

- **代码清单: 任务管理-6 (12):** 而任务如果是在等待内核资源这些，如事件、消息队列、信号量等，系统会直接将任务从等待列表移除，然后跳出 switch 语句。

- **代码清单: 任务管理-6 (13):** 如果目标任务状态超出预期，直接返回错误类型为“状态非法”的错误，并且退出删除操作。

- **代码清单: 任务管理-6 (14):** 如果启用了任务消息队列，将释放任务的所有任务消息。

- **代码清单: 任务管理-6 (15):** 在删除任务的时候，系统还会调用用户自定义的钩子函数，用户可以通过该钩子函数进行自定义的操作。

- **代码清单: 任务管理-6 (16):** 如果启用了调试代码和变量，将任务从任务调试双向列表移除。

- **代码清单: 任务管理-6 (17):** 到here就进行任务的删除操作，系统的任务数目减 1。

- **代码清单: 任务管理-6 (18):** 初始化对应的任务控制块，将任务状态变为删除态，退出临界段但不进行调度，返回错误类型为“无错误”的错误代码。

- **代码清单: 任务管理-6 (19):** 进行一次任务调度。
删除任务是说任务将返回并处以删除（休眠）状态，任务的代码不再被μC/OS 调用，删除任务不是删除代码，删除任务和挂起任务有些相似，其实有着本质的区别，根本来说，最大的不同就是删除任务队任务控制块的操作，我们知道在任务创建的时候，需要给每个任务分配一个任务控制块，这个任务控制块存储有关这个任务重要的信息，对任务间有至关重要的作用，挂起任务根本不会动任务控制块，但删除任务就会把任务控制块进行初始化，这样子关于任务的任何信息都被抹去。

注意了，删除任务并不会释放任务的栈空间。

删除任务函数的使用实例具体见代码清单: 任务管理-7。

列表 7: 代码清单: 任务管理-7 删除任务函数OSTaskDel()
使用实例

```c
/*
 * 任务句柄是一个指针，用于指向一个任务，当任务创建好之后，它就具有了一个任务句柄
 * 以后我们想要操作这个任务都需要通过这个任务句柄，如果是自身的任务操作自己，那么
 * 这个句柄可以为 NULL。
 */

staticOS_TCB AppTaskLed1TCB; /* LED 任务句柄 */

static void KEY_Task(void* parameter)
{OS_ERR err;
 while (1) {
   if ( Key_Scan(KEY2_GPIO_PORT,KEY2_GPIO_PIN) == KEY_ON ) {
     /* KEY2 被按下 */
     printf("删除 LED 任务!\n");
     OSTaskDel( &AppTaskLed1TCB, & err ); /* 删除 LED 任务! */
   }
   OSTimeDly ( 20, OS_OPT_TIME_DLY, & err ); /* 延时 20 个 tick */
 }
}
```

论坛: https://www.firebbs.cn/ 339 天猫: https://yehuosm.tmall.com
21.5.4 任务延时函数

21.5.4.1 OSTimeDly()

OSTimeDly() 任务在任务中用得非常之多，每个任务都必须是死循环，并且是必须要有阻塞的情况，否则低优先级的任务就无法被运行了，OSTimeDly() 函数常用于停止当前任务进行的运行，延时一段时间后再运行，OSTimeDly() 函数源码具体见代码清单: 任务管理-8。

列表 8: 代码清单: 任务管理-8OSTimeDly() 函数源码

```c
void OSTimeDly (OS_TICK dly,  //延时的时间值
    OS_OPT opt,       //选项
    OS_ERR *p_err)    //返回错误类型
{
    CPU_SR_ALLOC();
    //使用到临界段（在关/开中断时）时必须用到该宏，该宏声明和定义一个局部变量，用于保存关中断前的 CPU 状态寄存器 SR（临界段关中断只保存 SR）
    //，开中断时将该值还原。

#ifdef OS_SAFETY_CRITICAL(4) //如果启用（默认禁用）了安全检测
    if (p_err == (OS_ERR *)0) {  //如果错误类型实参为空
        OS_SAFETY_CRITICAL_EXCEPTION(); //执行安全检测异常函数
        return;  //返回，不执行延时操作
    }
#endif

#if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u //如果启用（默认启用）了中断中非法调用检测
    if (OSIntNestingCtr > (OS_NESTING_CTR)0u) {  //如果该延时函数是在中断中被调用
        *p_err = OS_ERR_TIME_DLY_ISR;  //错误类型为“在中断函数中延时”
        return;  //返回，不执行延时操作
    }
#endif
/* 当调度器被锁时任务不能延时 */
```

(下页继续)
if (OSSchedLockNestingCtr > (OS_NESTING_CTR)0u) { //如果调度器被锁
    *p_err = OS_ERR_SCHED_LOCKED; //错误类型为“调度器被锁”
    return; //返回，不执行延时操作
}

switch (opt) { (7) //根据延时选项参数 opt 分类操作
    case OS_OPT_TIME_DLY: //如果选择相对时间（从现在起延时多长时间）
    case OS_OPT_TIME_TIMEOUT: //如果选择超时（实际时间）
    case OS_OPT_TIME_PERIODIC: //如果选择周期性延时
        if (dly == (OS_TICK)0u) { (8) //如果参数 dly 为 0 (0 意味不延时)
            *p_err = OS_ERR_TIME_ZERO_DLY; //错误类型为“0 延时”
            return; //返回，不执行延时操作
        }
        break;

    case OS_OPT_TIME_MATCH: (9)
        //如果选择绝对时间（匹配系统开始运行（OSStart()）后的时钟节拍数）
        break;

    default: (10) //如果选项超出范围
        *p_err = OS_ERR_OPT_INVALID; //错误类型为“选项非法”
        return; //返回，不执行延时操作
}

OS_CRITICAL_ENTER(); //进入临界段
OSTCBCurPtr->TaskState = OS_TASK_STATE_DLY; (11) //修改当前任务的任务状态为延时状态
OS_TickListInsert(OSTCBCurPtr, dly, //将当前任务插入节拍列表
代码清单: 任务管理-8 (1): 任务延时的时钟节拍数，也就是延时的时间。

代码清单: 任务管理-8 (2): 任务延时的可选选项，在os.h中有定义，具体见代码清单: 任务管理-9。

列表9: 代码清单: 任务管理-9 任务延时的可选选项

```
#define OS_OPT_TIME_DLY DEF_BIT_NONE (1)
#define OS_OPT_TIME_TIMEOUT ((OS_OPT)DEF_BIT_01) (2)
#define OS_OPT_TIME_MATCH ((OS_OPT)DEF_BIT_02) (3)
#define OS_OPT_TIME_PERIODIC ((OS_OPT)DEF_BIT_03) (4)
```

代码清单: 任务管理-9 (1): OS_OPT_TIME_DLY: dly为相对时间，就是从现在起延时多长时间，到时钟节拍总计数OSTickCtr = OSTickCtr 当前 + dly 时延时结束。

代码清单: 任务管理-9 (2): OS_OPT_TIME_TIMEOUT: 跟OS_OPT_TIME_DLY的作用情况一样。

代码清单: 任务管理-9 (3): OS_OPT_TIME_MATCH: dly为绝对时间，就是从系统开始运行（调用OSStart())时到节拍总计数OSTickCtr = dly 时延时结束。

代码清单: 任务管理-9 (4): OS_OPT_TIME_PERIODIC: 周期性延时，跟OS_OPT_TIME_DLY的作用差不多，如果是长时间延时，该选项更精准一些。
• 代码清单: 任务管理-8 (3): 用于存放返回错误代码，如果挂起任务失败，则返回对应的错误代码。

• 代码清单: 任务管理-8 (4): 如果启用（默认禁用）了安全检测，系统就会执行安全检测的代码，如果错误类型实参为空，就执行安全检测异常函数，然后返回，不执行延时操作。

• 代码清单: 任务管理-8 (5): 如果启用（默认启用）了中断中非法调用检测，如果该延时函数是在中断中被调用，将返回错误类型为“在中断函数中延时”的错误代码，退出，不执行延时操作。

• 代码清单: 任务管理-8 (6): 如果调度器被锁，则不允许进行延时操作，返回错误类型为“调度器被锁”的错误代码，并且退出延时操作。因为延时就必须进行任务的切换，所以在延时的时候不能锁定调度器。

• 代码清单: 任务管理-8 (7): 根据延时选项参数 opt 分类操作。

• 代码清单: 任务管理-8 (8): 如果选择相对时间（从现在起延时多长时间）或者选择超时时间或者选择周期性延时，那么这表示延时时间，如果参数 dly 为 0（0 意味不延时），就会返回错误类型为“0 延时”的错误代码，并且退出不执行延时操作。

• 代码清单: 任务管理-8 (9): 如果选择绝对时间（匹配系统开始运行（OSStart()）后的时钟节拍数。

• 代码清单: 任务管理-8 (10): 如果选项超出范围，则视为非法，返回错误类型为“选项非法”的错误代码，并且退出不执行延时操作。

• 代码清单: 任务管理-8 (11): 程序能执行到这里，说明能正常进行延时操作，那么系统就会修改当前任务的任务状态为延时状态。

• 代码清单: 任务管理-8 (12): 调用 OS_TickListInsert() 函数将当前任务插入节拍列表，加入节拍列表的任务会按照延时时间进行升序排列，OS_TickListInsert() 源码具体见代码清单: 任务管理-10。

注：此源码看注释即可，就不过多讲解。
```c
void OS_TickListInsert(OS_TCB *p_tcb, //任务控制块
                      OS_TICK time, //时间
                      OS_OPT opt, //选项
                      OS_ERR *p_err) //返回错误类型
{
    OS_TICK tick_delta;
    OS_TICK tick_next;
    OS_TICK_SPOKE *p_spoke;
    OS_TCB *p_tcb0;
    OS_TCB *p_tcb1;
    OS_TICK_SPOKE_IX spoke;

    if (opt == OS_OPT_TIME_MATCH)
    {
        //如果 time 是个绝对时间
        tick_delta = time - OSTickCtr - 1u; //计算离到期还有多长时间
        if (tick_delta > OS_TICK_TH_RDY)
        {
            p_tcb->TickCtrMatch = (OS_TICK) 0u; //将任务的时钟节拍的匹配变量置 0
            p_tcb->TickRemain = (OS_TICK) 0u; //将任务的延时还需时钟节拍数量置 0
            p_tcb->TickSpokePtr = (OS_TICK_SPOKE *) 0; //该任务不插入节拍列表
            *p_err = OS_ERR_TIME_ZERO_DLY; //错误类型相当于 “0 延时”
            return; //返回，不将任务插入节拍列表
        }
        p_tcb->TickCtrMatch = time; //任务等待的匹配点为 OSTickCtr = time
        p_tcb->TickRemain = tick_delta + 1u; //计算任务离到期还有多长时间
    }
    else if (time > (OS_TICK) 0u)
    {
        //如果 time > 0
    }
}
```

(下页继续)
if (opt == OS_OPT_TIME_PERIODIC)
{
   //如果 time 是周期性时间
   tick_next = p_tcb->TickCtrPrev + time;
   //计算任务接下来要匹配的时钟节拍总数
   tick_delta = tick_next - OSTickCtr - 1u;  //计算任务离匹配还有多少个长
   time

   if (tick_delta < time)
   {
      //如果 p_tcb->TickCtrPrev<OSTickCtr+1
      p_tcb->TickCtrlMatch = tick_next;  //将 p_tcb->TickCtrlPrev +
   }
   else
   {
      //如果 p_tcb->TickCtrlPrev >= OSTickCtrl + 1
      p_tcb->TickCtrlMatch = OSTickCtrl + time;  //将 OSTickCtrl +
   }

   p_tcb->TickRemain = p_tcb->TickCtrlMatch - OSTickCtr;  //计算任务离到期还有多长时间
   p_tcb->TickCtrPrev = p_tcb->TickCtrlMatch;  //保存当前匹配值为下一周
   期延时用

}
else
{
   //如果 time 是相对时间
   p_tcb->TickCtrlMatch = OSTickCtrl + time;  //任务等待的匹配点为
   OSTickCtr + time
   p_tcb->TickRemain = time;  //计算任务离到期的时间就是 time
}

else
{
   //如果 time = 0

}
p_tcb->TickCtrMatch = (OS_TICK) 0;  //将任务的时钟节拍的匹配变量置0
p_tcb->TickRemain = (OS_TICK) 0;  //将任务的延时还需时钟节拍数置0
p_tcb->TickSpokePtr = (OS_TICK_SPOKE *) 0;  //该任务不插入节拍列表
p_err = OS_ERR_TIME_ZERO_DLY;  //错误类型为“0 延时”
return;  //返回，不将任务插入节拍列表

spoke = (OS_TICK_SPOKE_IX)(p_tcb->TickCtrMatch % OSCfg_TickWheelSize);  //使用哈希算法（取余）来决定任务存于数组
p_spoke = &OSCfg_TickWheel[spoke];  //OSCfg_TickWheel 的哪个元素（组织一个节拍列表），
//与更新节拍列表相对应，可方便查找到期任务。
if (p_spoke->NbrEntries == (OS_OBJ_QTY) 0) {
    //如果当前节拍列表为空
    p_tcb->TickNextPtr = (OS_TCB *) 0;  //任务中指向节拍列表中下一个任务的指针置空
    p_tcb->TickPrevPtr = (OS_TCB *) 0;  //任务中指向节拍列表中前一个任务的指针置空
    p_spoke->FirstPtr = p_tcb;  //当前任务被列为该节拍列表的第一个任务
    p_spoke->NbrEntries = (OS_OBJ_QTY) 1;  //节拍列表中的元素数目为1
} else {
    //如果当前节拍列表非空
    p_tcb1 = p_spoke->FirstPtr;  //获取列表中的第一个任务
    while (p_tcb1 != (OS_TCB *) 0) {
        //如果该任务存在
        p_tcb1->TickRemain = p_tcb1->TickCtrMatch  //计算该任务的剩余等待时间
        p_tcb1 = p_tcb1->TickNextPtr;
    }
}
```c
- OSTickCtr;

if (p_tcb->TickRemain > p_tcb1->TickRemain)
{
    //如果当前任务的剩余等待时间大于该任务的
    if (p_tcb1->TickNextPtr != (OS_TCB *)0)
    {
        //如果该任务不是列表的最后一个元素
        p_tcb1 = p_tcb1->TickNextPtr;
        //让当前任务继续与该任务的下一个任务作比较
    }
    else
    {
        //如果该任务是列表的最后一个元素
        p_tcb->TickNextPtr = (OS_TCB *)0; //当前任务为列表的最后一个
        p_tcb->TickPrevPtr = p_tcb1; //该任务是当前任务的前一个元素
        p_tcb1->TickNextPtr = p_tcb; //当前任务是该任务的后一个元素

        p_tcb1 = (OS_TCB *)0; //插入完成，退出 while...
    }
}
else
{
    //如果当前任务的剩余等待时间不大于该任务的
    if (p_tcb1->TickPrevPtr == (OS_TCB *)0)
    {
        //如果该任务是列表的第一个元素
        p_tcb->TickPrevPtr = (OS_TCB *)0; //当前任务就作为列表的第一个元素
        p_tcb->TickNextPtr = p_tcb1; //该任务是当前任务的后一个元素
        p_tcb1->TickPrevPtr = p_tcb; //当前任务是该任务的前一个元素
        p_spoke->FirstPtr = p_tcb; //当前任务是列表的第一个元素
    }
}
```

```c
else
{
    // 如果该任务也不是是列表的第一个元素
    p_tcb0 = p_tcb1->TickPrevPtr; // p_tcb0 暂存该任务的前一个任务
    p_tcb->TickPrevPtr = p_tcb0;
    // 该任务的前一个任务作为当前任务的前一个任务
    p_tcb->TickNextPtr = p_tcb1; // 该任务作为当前任务的后一个任务
    p_tcb0->TickNextPtr = p_tcb; // p_tcb0 暂存的任务的下一个任务改为当前任务
    p_tcb1->TickPrevPtr = p_tcb; // 该任务的前一个任务也改为当前任务
}
p_tcb1 = (OS_TCB *)0; // 插入完成，退出 while 循环
}
}
p_spoke->NbrEntries++; // 节拍列表中的元素数目加 1
} // 更新节拍列表的元素数目的最大记录
if (p_spoke->NbrEntriesMax < p_spoke->NbrEntries) {
    p_spoke->NbrEntriesMax = p_spoke->NbrEntries;
}
p_tcb->TickSpokePtr = p_spoke; // 记录当前任务存放于哪个节拍列表
*p_err = OS_ERR_NONE; // 错误类型为“无错误”
}
```

- **代码清单: 任务管理-8 (13)**: 调用 OS_RdyListRemove() 函数从就绪列表移除当前任务，进行延时操作。

- **代码清单: 任务管理-8 (14)**: 进行一次任务切换。

任务的延时在实际中运用特别多，因为需要暂停一个任务，让任务放弃 CPU，延时结束后再继续运行该任务，如果任务中没有阻塞的话，比该任务优先级低的任务则无法得到 CPU 的使用权。
就无法运行，具体见代码清单:任务管理-11 加粗部分。

列表 11: 代码清单: 任务管理-11 延时函数 vTaskDelay()的使用实例

```c
void vTaskA( void * pvParameters )
{
    while (1) {
        // ...
        // 这里为任务主体代码
        // ...
        /* 调用相对延时函数，阻塞 1000 个 tick */
        OSTimeDly( 1000, OS_OPT_TIME_DLY, & err );
    }
}
```

21.5.4.2 OSTimeDlyHMSM()

OSTimeDlyHMSM() 函数与 OSTimeDly() 函数的功能类似，也是用于停止当前任务进行的运行，延时一段时间后再运行，但是 OSTimeDlyHMSM() 函数会更加直观，延时多少个小时、分钟、秒、毫秒。但是，用户若要使用 OSTimeDlyHMSM() 函数，必须将宏 OS_CFG_TIME_DLY_HMSM_EN 设为 1，该宏定义位于 os_cfg.h 中，OSTimeDlyHMSM() 函数源码具体见代码清单: 任务管理-12。

列表 12: 代码清单: 任务管理-12OSTimeDlyHMSM() 源码

```c
#ifdef OS_CFG_TIME_DLY_HMSM_EN > 0
//如果启用（默认启用）了 OSTimeDlyHMSM() 函数

void OSTimeDlyHMSM (CPU_INT16U hours, (1) //延时小时
    CPU_INT16U minutes, (2) //分钟数
    CPU_INT16U seconds, (3) //秒数
    CPU_INT32U milli, (4) //毫秒数
    OS_OPT opt, (5) //选项
```

(下页继续)
OS_ERR *p_err) (6) //返回错误类型
{
    #if OS_CFG_ARG_CHK_EN > 0u(7) //如果启用（默认启用）了参数检测功能
        CPU_BOOLEAN opt_invalid;  //声明变量用于参数检测
        CPU_BOOLEAN opt_non_strict;
    #endif
    OS_OPT opt_time;
    OS_RATE_HZ tick_rate;
    OS_TICK ticks;
    CPU_SR_ALLOC();

    #ifdef OS_SAFETY_CRITICAL(8) //如果启用（默认禁用）了安全检测
        if (p_err == (OS_ERR *)0) { //如果错误类型实参为空
            OS_SAFETY_CRITICAL_EXCEPTION(); //执行安全检测异常函数
            return; //返回，不执行延时操作
        }
    #endif

    #if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u (9) //如果启用（默认启用）了中断中非法调用检测
        if (OSIntNestingCtr > (OS_NESTING_CTR)0u) { //如果该延时函数是在中断中被调用
            *p_err = OS_ERR_TIME_DLY_ISR; //错误类型为“在中断函数中延时”
            return; //返回，不执行延时操作
        }
    #endif

    /* 当调度器被锁时任务不能延时 */
    if (OSSchedLockNestingCtr > (OS_NESTING_CTR)0u) { (10) //如果调度器被锁
        *p_err = OS_ERR_SCHED_LOCKED; //错误类型为“调度器被锁”
        return; //返回，不执行延时操作
    }
(下页继续)
opt_time = opt & OS_OPT_TIME_MASK; (11) //检测除选项中与延时时间性质有关的位
switch (opt_time) {
    //根据延时选项参数 opt 分类操作
    case OS_OPT_TIME_DLY: //如果选择相对时间（从现在起延时多长时间）
    case OS_OPT_TIME_TIMEOUT: //如果选择超时（实际同上）
    case OS_OPT_TIME_PERIODIC: //如果选择周期性延时
        if (milli == (CPU_INT32U)0u) { //如果毫秒数为 0
            if (seconds == (CPU_INT16U)0u) { //如果秒数为 0
                if (minutes == (CPU_INT16U)0u) { //如果分钟数为 0
                    if (hours == (CPU_INT16U)0u) { //如果小时数为 0
                        *p_err = OS_ERR_TIME_ZERO_DLY; //错误类型为"0 延时"
                        return; (12) //返回，不执行延时操作
                    }
                }
            }
        }
        break;
        case OS_OPT_TIME_MATCH: (13) //如果选择绝对时间（把系统开始运行 (OSStart()) 时做为起点）
        break;
        default: (14) //如果选项超出范围
            *p_err = OS_ERR_OPT_INVALID; //错误类型为"选项非法"
            return; //返回，不执行延时操作
    }
#if OS_CFG_ARG_CHK_EN > 0u (15)
//如果启用（默认启用）了参数检测功能
    opt_invalid = DEF_BIT_IS_SET_ANY(opt, ~OS_OPT_TIME_OPTS_MASK);
    //检测除选项位以外其他位是否被置位
    if (opt_invalid == DEF_YES) { (16)
        //如果除选项位以外其他位有被置位的
        *p_err = OS_ERR_OPT_INVALID; //错误类型为“选项非法”
        return; //返回，不执行延时操作
    }

    opt_non_strict = DEF_BIT_IS_SET(opt, OS_OPT_TIME_HMSM_NON_STRICT); (17)
    //检测有关时间参数取值范围的选项位
    if (opt_non_strict != DEF_YES) { //如果选项选择了 OS_OPT_TIME_HMSM_STRICT
        if (milli > (CPU_INT32U)9999u) { (18) //如果毫秒数>9999
            *p_err = OS_ERR_TIME_INVALID_MILLISECONDS; //错误类型为“毫秒数不可用”
            return; //返回，不执行延时操作
        }
    }

    if (seconds > (CPU_INT16U)59u) { (19) //如果秒数>59
        *p_err = OS_ERR_TIME_INVALID SECONDS; //错误类型为“秒数不可用”
        return; //返回，不执行延时操作
    }

    if (minutes > (CPU_INT16U)59u) { (20) //如果分钟数>59
        *p_err = OS_ERR_TIME_INVALID_MINUTES; //错误类型为“分钟数不可用”
        return; //返回，不执行延时操作
    }

    if (hours > (CPU_INT16U)99u) { (21) //如果小时数>99
        *p_err = OS_ERR_TIME_INVALID_HOURS; //错误类型为“小时数不可用”
        return; //返回，不执行延时操作
    }
}
    else { //如果选项选择了 OS_OPT_TIME_HMSM_NON_STRICT
    if (minutes > (CPU_INT16U)9999u) { (22) //如果分钟数>9999
*p_err = OS_ERR_TIME_INVALID_MINUTES; //错误类型为“分钟数不可用”
return;
}
if (hours > (CPU_INT16U) 999u) { (23) //如果小时数>999
*p_err = OS_ERR_TIME_INVALID_HOURS; //错误类型为“小时数不可用”
return; //返回，不执行延时操作
}
#endif

*/ 将延时时间转换成时钟节拍数 */
tick_rate = OSCfg_TickRate_Hz; (24) //获取时钟节拍的频率
ticks = ((OS_TICK) hours * (OS_TICK) 3600u + (OS_TICK) minutes *
(OS_TICK) 60u + (OS_TICK) seconds) * tick_rate +
(tick_rate * ((OS_TICK) milli + (OS_TICK) 500u /
tick_rate)) / (OS_TICK) 1000u; (25) //将延时时间转换成时钟节拍数

if (ticks > (OS_TICK) 0u) {
    (26) //如果延时节拍数>0
    OS_CRITICAL_ENTER(); //进入临界段
    OSTCBCurPtr->TaskState = OS_TASK_STATE_DLY; //修改当前任务的任务状态为延时状态
    OS_TickListInsert(OSTCBCurPtr, //将当前任务插入节拍列表
ticks,
    opt_time,
    p_err); (27)
    if (*p_err != OS_ERR_NONE) { //如果当前任务插入节拍列表时出现错误
        OS_CRITICAL_EXIT_NO_SCHED(); //退出临界段（无调度）
        return; //返回，不执行延时操作
    }
    OS_RdyListRemove(OSTCBCurPtr); (28) //从就绪列表移除当前任务
    OS_CRITICAL_EXIT_NO_SCHED(); //退出临界段（无调度）


```c
OSSched(); // (29) 任务切换
*p_err = OS_ERR_NONE;  // 错误类型为 “无错误”
} else {
  *p_err = OS_ERR_TIME_ZERO_DLY;  // 错误类型为 “0 延时”
}
#endif
```

- 代码清单: 任务管理-12 (1): 延时时间——小时数。
- 代码清单: 任务管理-12 (2): 延时时间——分钟数
- 代码清单: 任务管理-12 (3): 延时时间——秒数
- 代码清单: 任务管理-12 (4): 延时时间——毫秒数
- 代码清单: 任务管理-12 (5): 任务延时的可选选项，在 os.h 中有定义，具体见代码清单: 任务管理-13。

### 列表 13: 代码清单: 任务管理-13 任务延时的可选选项

```c
#define OS_OPT_TIME_DLY        DEF_BIT_NONE (1)
#define OS_OPT_TIME_TIMEOUT     ((OS_OPT)DEF_BIT_01) (2)
#define OS_OPT_TIME.Match       ((OS_OPT)DEF_BIT_02) (3)
#define OS_OPT_TIME.PERIODIC    ((OS_OPT)DEF_BIT_03) (4)
#define OS_OPT_TIME.HMSM.Strict ((OS_OPT)DEF_BIT_NONE) (5)
#define OS_OPT_TIME.HMSM.NonStrict ((OS_OPT)DEF_BIT_04) (6)
```

- 代码清单: 任务管理-13 (1): OS_OPT_TIME_DLY: dly 为相对时间，就是从现在起延时多长时间，到时钟节拍累计数 OSTickCtr = OSTickCtr 当前 + dly 时延时结束。
- 代码清单: 任务管理-13 (2): OS_OPT_TIME_TIMEOUT: 跟 OS_OPT_TIME_DLY 的作用情况一样。

论坛: [https://www.firebbs.cn/](https://www.firebbs.cn/)  
天猫: [https://yehuosm.tmall.com](https://yehuosm.tmall.com)
[野火]uCOS-III 内核实现与应用开发实战指南

- 代码清单: 任务管理-13 (3): OS_OPT_TIME_MATCH: dly 为绝对时间，就是从系统开始运行（调用 OSStart()）时到节拍总计数 OSTickCtr = dly 时延时结束。

- 代码清单: 任务管理-13 (4): OS_OPT_TIME_PERIODIC: 周 期 性 延 时，跟 OS_OPT_TIME_DLY 的作用差不多，如果是长时间延时，该选项更精准一些。

- 代码清单: 任务管理-13 (5): 延时时间取值比较严格:
  - 小时数 hours: (0-99)
  - 分钟数 minutes: (0-59)
  - 秒数 seconds: (0-59)
  - 毫秒数 milliseconds: (0-999)

- 代码清单: 任务管理-13 (6): 延时时间取值比较宽松。
  - 小时数 hours: (0-999)
  - 分钟数 minutes: (0-9999)
  - 秒数 seconds: (0-65535)
  - 毫秒数 milliseconds: (0-4294967295)

- 代码清单: 任务管理-12 (6): 用于存放返回错误代码，如果挂起任务失败，则返回对应的错误代码。

- 代码清单: 任务管理-12 (7): 如果启用（默认启用）了参数检测功能，则定义一些变量用于参数检测。

- 代码清单: 任务管理-12 (8): 如果启用（默认禁用）了安全检测，就会包含安全检测的代码，如果错误类型实参为空，执行安全检测异常函数，然后返回，不执行延时操作。

- 代码清单: 任务管理-12 (9): 如果启用（默认启用）了中断中非法调用检测，并且如果该延时函数是在中断中被调用，则被视为非法，返回错误类型为“在中断函数中延时”的错误，然后返回，不执行延时操作。

- 代码清单: 任务管理-12 (10): 当调度器被锁时任务不能延时，任务延时后会进行任务调度，如果调度器被锁，就会返回错误类型为“调度器被锁”的错误，然后返回，不执行延时操
作。

- **代码清单: 任务管理-J2 (11)**: 检测到选项中与延时时间性质有关的位，并且根据延时选项参数 opt 分类操作。

- **代码清单: 任务管理-J2 (12)**: 如果选择相对延时（从现在起延时多长时间）、超时延时、周期性延时等延时类型，就会检测一下延时的时间是多少，如果是 0，则是不允许的，返回错误类型为“0 延时”的错误，不进行延时操作。

- **代码清单: 任务管理-J2 (13)**: 如果选择绝对时间，会把系统开始运行 OSStart() 时做为起点。

- **代码清单: 任务管理-J2 (14)**: 如果选项超出范围，返回错误类型为“选项非法”的错误，然后退出，不进行延时操作。

- **代码清单: 任务管理-J2 (15)**: 如果启用（默认启用）了参数检测功能，就会检测到选项位以外其他位是否被置位。

- **代码清单: 任务管理-J2 (16)**: 如果除选项位外其他位有被置位的，则返回错误类型为“选项非法”的错误，然后退出，不执行延时操作。

- **代码清单: 任务管理-J2 (17)**: 检测有关时间参数取值范围的选项位，如果选项选择了 OS_OPT_TIME_HMSM_STRICT，就是比较严格的参数范围。

- **代码清单: 任务管理-J2 (18)**: 如果毫秒数大于 999，返回错误类型为“毫秒数不可用”的错误，然后退出，不执行延时操作。

- **代码清单: 任务管理-J2 (19)**: 如果秒数大于 59，返回错误类型为“秒数不可用”的错误，然后退出，不执行延时操作。

- **代码清单: 任务管理-J2 (20)**: 如果分钟数大于 59，返回错误类型为“分钟数不可用”的错误，然后退出，不执行延时操作。

- **代码清单: 任务管理-J2 (21)**: 如果小时数大于 99，返回错误类型为“小时数不可用”的错误，然后退出，不执行延时操作。

- **代码清单: 任务管理-J2 (22)**: 如果选项选择了 OS_OPT_TIME_HMSM_NON_STRICT，就是比较宽松的延时操作，如果分钟数大于 9999，返回错误类型为“分钟数不可用”的错误，然后退出，不执行延时操作。
• **代码清单: 任务管理-12 (23)**：如果小时数大于 999，返回错误类型为“小时数不可用”的错误，然后退出，不执行延时操作。

• **代码清单: 任务管理-12 (24)**：因为我们延时的时间是时、分、秒、毫秒，但是系统的时间单位是时钟节拍，所以需要将延时时间转换成时钟节拍数，首先获取时钟节拍的频率。

• **代码清单: 任务管理-12 (25)**：然后根据我们延时的时间进行计算转换，将延时时间转换成时钟节拍数 tick。

• **代码清单: 任务管理-12 (26)**：如果延时节拍数大于 0，表示可以延时，修改当前任务的任务状态为延时状态。

• **代码清单: 任务管理-12 (27)**：调用 OS_TickListInsert() 函数将当前任务插入节拍列表。

• **代码清单: 任务管理-12 (28)**：调用 OS_RdyListRemove() 函数从就绪列表移除当前任务。

• **代码清单: 任务管理-12 (29)**：进行一次任务切换。

任务延时函数 OSTimeDlyHMSM() 的使用实例具体见代码清单: 任务管理-14。
表14：代码清单：任务管理-14 延时函数 vTaskDelay()的使用实例

```c
void vTaskA( void * pvParameters )
{
    while (1) {
        // ...
        // 这里为任务主体代码
        // ...

        /* 调用延时函数，延时 1s */
        OSTimeDlyHMSM(0,0,1,0, OS_OPT_TIME_DLY, & err );
    }
}
```

### 21.6 任务的设计要点

作为一个嵌入式开发人员，要对自己设计的嵌入式系统了如指掌，任务的优先级信息，任务与中断的处理，任务的运行时间、逻辑、状态等都要知道，才能设计出好的系统，所以，在设计的时候需要根据需求制定框架。在设计之初就应该考虑下面几点因素：任务运行的上下文环境、任务的执行时间合理设计。

μC/OS 中程序运行的上下文包括：

- 中断服务函数。
- 普通任务。
- 空闲任务。

1. 中断服务函数：

中断服务函数是一种需要特别注意的上下文环境，它运行在非任务的执行环境下（一般为芯片的一种特殊运行模式（也被称作特权模式）），在这个上下文环境中不能使用挂起当前任务的操作，
不允许调用任何会阻塞运行的 API 函数接口。另外需要注意的是，中断服务程序最好保持精简短小，快进快出。一般在中断服务函数中只做标记事件的发生，然后通知任务，让对应任务去执行相关处理。因为中断服务函数的优先级高于任何优先级的任务，如果中断处理时间过长，将会导致整个系统的任务无法正常运行。所以在设计的时候必须考虑中断的频率、中断的处理时间等重要因素，以便配合对应中断处理任务的工作。

μC/OS 支持中断延迟发布，使得原本在中断中发布的信息变成任务级发布，这样子会使得中断服务函数的处理更加快速，屏蔽中断的时间更短，这样子能快速响应其他的中断，真正称得上实时操作系统。

2. 任务

任务看似没有什么限制程序执行的因素，似乎所有的操作都可以执行。但是作为一个优先级明确的实时系统，如果一个任务中的程序出现了死循环操作（此处的死循环是指没有阻塞机制的任务循环体），那么比这个任务优先级低的任务都将无法执行，当然也包括了空闲任务，因为死循环的时候，任务不会主动让出 CPU，低优先级的任务是不可能得到 CPU 的使用权的，而高优先级的任务就可以抢占 CPU。这个情况在实时操作系统中是必须注意的一点，所以在任务中不允许出现死循环。如果一个任务只有就绪态而无阻塞态，势必会影响到其他低优先级任务的执行，所以在进行任务设计时，就应该保证任务在不活跃的时候，任务可以进入阻塞态以交出 CPU 使用权，这就需要我们自己明确知道什么情况下让任务进入阻塞态，保证低优先级任务可以正常运行。在实际设计中，一般会将紧急的处理事件的任务优先级设置得高一些。

3. 空闲任务

空闲任务（idle 任务）是 μC/OS 系统中没有其他工作进行时自动进入的系统任务。因为处理器总是需要代码来执行——所以至少要有一个任务处于运行态。μC/OS 为了保证这一点，当调用 OSInit() 函数进行系统初始化时，系统会自动创建一个空闲任务，空闲任务是一个非常短小的循环。用户可以通过空闲任务钩子方式，在空闲任务上钩入自己的功能函数。通常这个空闲任务钩子能够完成一些额外的特殊功能，例如系统运行状态的指示，系统省电模式等。空闲任务是唯一一个不允许出现阻塞情况的任务，因为 μC/OS 需要保证系统永远都有一个可运行的任务。

对于空闲任务钩子上挂接的空闲钩子函数，它应该满足以下的条件:

- 永远不会挂起空闲任务;
- 不应该陷入死循环，需要留出部分时间用于统计系统的运行状态等。
4. 任务的执行时间:

任务的执行时间一般是指两个方面，一是任务从开始到结束的时间，二是任务的周期。

在系统设计的时候这两个时间我们都需要考虑，例如，对于事件 A 对应的服务任务 Ta，系统要求的实时响应指标是 10ms，而 Ta 的最大运行时间是 1ms，那么 10ms 就是任务 Ta 的周期了，1ms 则是任务的运行时间，简单来说任务 Ta 在 10ms 内完成对事件 A 的响应即可。此时，系统中还存在着以 50ms 为周期的另一任务 Tb，它每次运行的最大时间长度是 100us。在这种情况下，即使把任务 Tb 的优先级设置比 Ta 更高，对系统的实时性指标也没什么影响，因为即使在 Ta 的运行过程中，Tb 抢占了 Ta 的资源，等到 Tb 执行完毕，消耗的时间也只不过是 100us，还是在事件 A 指定的响应时间内 (10ms)，Ta 能够安全完成对事件 A 的响应。但是假如系统中还存在任务 Tc，其运行时间为 20ms，假如将 Tc 的优先级设置比 Ta 更高，那么在 Ta 运行的时候，突然间被 Tc 打断，等到 Tc 执行完毕，那 Ta 已经错过对事件 A (10ms) 的响应了，这是不允许的。所以在我们设计的时候，必须考虑任务的时间，一般来说处理时间更短的任务优先级应设置更高一些。

21.7 任务管理实验

任务管理实验是将任务常用的函数进行一次实验，在野火 STM32 开发板上进行该实验，创建 LED1、LED2 和 LED3 三个应用任务，三个任务的优先级均是 3，本实例使用时间片轮转调度它们运行。系统开始运行后，三个任务均每隔 1s 切换一次自己的 LED 灯的灭亮状态。当 LED2 和 LED3 两个任务切换 5 次后就均挂起自身，停止切换。而 LED1 依然继续切换 LED1，当 LED1 切换 10 次时，会恢复 LED2 和 LED3 两个任务运行。依此循环。具体见代码清单: 任务管理-15。

列表 15: 代码清单: 任务管理-15 任务管理实验

```c
#include <includes.h>
#include <string.h>

//OS_MEM  mem;  //声明内存管理对象
//uint8_t ucArray [ 3 ] [ 20 ];  //声明内存分区大小
```

（下页继续）
static OS_TCB AppTaskStartTCB;  //任务控制块
static OS_TCB AppTaskLed1TCB;
static OS_TCB AppTaskLed2TCB;
static OS_TCB AppTaskLed3TCB;

static CPU_STK AppTaskStartStk[APP_TASK_START_STK_SIZE];    //任务栈
static CPU_STK AppTaskLed1Stk [ APP_TASK_LED1_STK_SIZE ];
static CPU_STK AppTaskLed2Stk [ APP_TASK_LED2_STK_SIZE ];
static CPU_STK AppTaskLed3Stk [ APP_TASK_LED3_STK_SIZE ];

static void AppTaskStart ( void *p_arg);            //任务函数声明
static void AppTaskLed1 ( void * p_arg );
static void AppTaskLed2 ( void * p_arg );
static void AppTaskLed3 ( void * p_arg );

int main ( void)
{
    OS_ERR err;
    OSInit(&err);                     //初始化 μC/OS-III

    /* 创建起始任务 */
    OSTaskCreate((OS_TCB *) &AppTaskStartTCB,    //任务控制块地址
                  (CPU_CHAR *) "App Task Start",   //任务名称
                  (OS_TASK_PTR ) AppTaskStart,     //任务函数
                  (void *) 0,
                  (OS_PRIO ) APP_TASK_START_PRIO,  //任务的优先级
                  (CPU_STK *) &AppTaskStartStk[0],

    )
//任务栈的基地址
(CPU_STK_SIZE) APP_TASK_START_STK_SIZE / 10,
//任务栈空间剩余 1/10 时限制其增长
(CPU_STK_SIZE) APP_TASK_START_STK_SIZE,
//任务栈空间（单位：sizeof(CPU_STK)
(OS_MSG_QTY ) 5u,
//任务可接收的最大消息数
(OS_TICK ) 0u,
//任务的时间片轮转数（0 表默认值 OSCfg_TickRate_Hz/10）
(void *) 0,
//任务扩展（0 表不扩展）
(OS_OPT ) (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR), //任务
选
项
(OS_ERR *) &err);  //返回错误类型
OSStart(&err);
//启动多任务管理（交由 μC/OS-III 控制）
}
static void AppTaskStart (void *p_arg)
{
    CPU_INT32U cpu_clk_freq;
    CPU_INT32U cnts;
    OS_ERR err;

    (void)p_arg;

    BSP_Init(); //板级初始化
    CPU_Init();
    //初始化 CPU 组件（时间戳、中断时间测量和主机名）
cpu_clk_freq = BSP_CPU_ClkFreq();
//获取 CPU 内核时钟频率（SysTick 工作时钟）

Vcnts = cpu_clk_freq / ((CPU_INT32U)OSCfg_TickRate_Hz);
//根据用户设定的时钟节拍频率计算 SysTick 定时器的计数值

OS_CPU_SysTickInit(cnts);
//调用 SysTick 初始化函数，设置定时器计数值和启动定时器

Mem_Init();
//初始化内存管理组件（堆内存池和内存池表）

#if OS_CFG_STAT_TASK_EN > 0u
//如果启用（默认启用）了统计任务

OSStatTaskCPUUsageInit(&err);
//计算没有应用任务（只有空闲任务）运行时 CPU 的（最大）容量（决定 OS_Stat_~IdleCtrMax 的值，为后面计算 CPU 使用率使用）。
#endif

CPU_IntDisMeasMaxCurReset();
//复位（清零）当前最大中断时间

/* 配置时间片轮转调度 */
OSSchedRoundRobinCfg((CPU_BOOLEAN )DEF_ENABLED,  //启用时间片轮转调度
(OS_TICK )0,  //把 OSCfg_TickRate_Hz / 10 设为默认时间片值
(OS_ERR )&err );  //返回错误类型

/* 创建 LED1 任务 */
OSTaskCreate((OS_TCB )&AppTaskLed1TCB,  //任务控制块地址
(CPU_CHAR *)"App Task Led1",
(OS_TASK_PTR ) AppTaskLed1,  //任务函数
(void ) 0,
//传递给任务函数（形参 p_arg）的实参
(OS_PRIO ) APP_TASK_LED1_PRIO, //任务的优先级
(CPU_STK  *) &AppTaskLed1Stk[0],
//任务栈的基地址
(CPU_STK_SIZE) APP_TASK_LED1_STK_SIZE / 10,
//任务栈空间剩下 1/10 时限制其增长
(CPU_STK_SIZE) APP_TASK_LED1_STK_SIZE,
//任务栈空间 (单位: sizeof(CPU_STK))
(OS_MSG_QTY ) 5u,
//任务可接收的最大消息数
(OS_TICK ) 0u,
//任务的时间片轮转数 (0 表默认值)
(void *) 0,
//任务扩展 (0 表不扩展)
(OS_OPT ) (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
(OS_ERR *) &err; //返回错误类型

/* 创建 LED2 任务 */
OSTaskCreate((OS_TCB *) &AppTaskLed2TCB, //任务控制块地址
  (CPU_CHAR *) "App Task Led2", //任务名称
  (OS_TASK_PTR ) AppTaskLed2, //任务函数
  (void *) 0,
  //传递给任务函数 (形参 p_arg) 的实参
  (OS_PRIO ) APP_TASK_LED2_PRIO, //任务的优先级
  (CPU_STK  *) &AppTaskLed2Stk[0],
  //任务栈的基地址
  (CPU_STK_SIZE) APP_TASK_LED2_STK_SIZE / 10,
  //任务栈空间剩下 1/10 时限制其增长
  (CPU_STK_SIZE) APP_TASK_LED2_STK_SIZE,
  //任务栈空间 (单位: sizeof(CPU_STK))
  (OS_MSG_QTY ) 5u,
  //任务可接收的最大消息数
  (OS_TICK ) 0u,
//任务的时间片轮转数 (0 表默认值)
(void *) 0,
//任务扩展 (0 表不扩展)
(OS_OPT ) (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
(OS_ERR *) & err);  //返回错误类型

/* 创建 LED3 任务 */
OSTaskCreate((OS_TCB * ) & AppTaskLed3TCB,  //任务控制块地址
(CPU_CHAR *) "App Task Led3", //任务名称
(OS_TASK_PTR ) AppTaskLed3,          //任务函数
(void *) 0,
//传递给任务函数(形参 p_arg) 的实参
(OS_PRIO ) APP_TASK_LED3_PRIO,       //任务的优先级
(CPU_STK *) & AppTaskLed3Stk[0],
//任务栈的地址
(CPU_STK_SIZE) APP_TASK_LED3_STK_SIZE / 10,
//任务栈空间剩下 1/10 时限制其增长
(CPU_STK_SIZE) APP_TASK_LED3_STK_SIZE,
//任务栈空间 (单位: sizeof(CPU_STK))
(OS_MSG_QTY ) 5u,
//任务可接收的最大消息数
(OS_TICK ) 0u,
//任务的时间片轮转数 (0 表默认值)
(void *) 0,
//任务扩展 (0 表不扩展)
(OS_OPT ) (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
(OS_ERR *) & err);  //返回错误类型

OSTaskDel ( 0, & err );  //删除起始任务本身，该任务不再运行
static void AppTaskLed1 ( void * p_arg )
{
    OS_ERR err;
    OS_REG value;

    (void)p_arg;

    while (DEF_TRUE) //任务体，通常写成一个死循环
    {
        macLED1_TOGGLE (); //切换 LED1 的亮灭状态

        value = OSTaskRegGet ( 0, 0, & err ); //获取自身任务寄存器值

        if ( value < 10 ) //如果任务寄存器值<10
        {
            OSTaskRegSet ( 0, 0, ++ value, & err ); //继续累加任务寄存器值
        }
        else //如果累加到 10
        {
            OSTaskRegSet ( 0, 0, 0, & err ); //将任务寄存器值归 0

            OSTaskResume ( & AppTaskLed2TCB, & err ); //恢复 LED2 任务
            printf("恢复 LED2 任务! \n");
        }
    }
}
OSTaskResume ( & AppTaskLed3TCB, & err );  //恢复 LED3 任务
        printf("恢复 LED3 任务! \n");
    }

    OSTimeDly ( 1000, OS_OPT_TIME_DLY, & err );
    //相对性延时 1000 个时钟节拍 (1s)
}

/*
**********************************************************************
**  LED2 TASK
***********************************************************************/

static void AppTaskLed2 ( void * p_arg )
{
    OS_ERR err;
    OS_REG value;

    (void)p_arg;

    while (DEF_TRUE)  //任务体，通常写成一个死循环
    {
        macLED2_TOGGLE ();  //切换 LED2 的亮灭状态
    }
value = OSTaskRegGet(0, 0, &err); // 获取自身任务寄存器值

if (value < 5) // 如果任务寄存器值<5
{
    OSTaskRegSet(0, 0, ++value, &err); // 继续累加任务寄存器值
}
else // 如果累加到 5
{
    OSTaskRegSet(0, 0, 0, &err); // 将任务寄存器值归 0

    OSTaskSuspend(0, &err); // 挂起自身
    printf("挂起 LED2 任务 (自身)! \n");
}

OSTimeDly(1000, OS_OPT_TIME_DLY, &err); // 相对性延时 1000 个时钟节拍 (1s)
static void AppTaskLed3 ( void * p_arg )
{
    OS_ERR err;
    OS_REG value;

    (void)p_arg;

    while (DEF_TRUE) //任务体，通常写成一个死循环
    {
        macLED3_TOGGLE (); //切换LED3 的亮灭状态

        value = OSTaskRegGet ( 0, 0, & err ); //获取自身任务寄存器值

        if ( value < 5 ) //如果任务寄存器值<5
        {
            OSTaskRegSet ( 0, 0, ++ value, & err ); //继续累加任务寄存器值
        }
        else //如果累加到5
        {
            OSTaskRegSet ( 0, 0, 0, & err ); //将任务寄存器值归零

            OSTaskSuspend ( 0, & err ); //挂起自身
            printf("挂起 LED3 任务 (自身)! \n");
        }

        OSTimeDly ( 1000, OS_OPT_TIME_DLY, & err ); //相对性延时1000个时钟节拍(1s)
    }
}
21.8 实验现象

将程序编译好，用 USB 线连接计算机和开发板的 USB 接口（对应丝印为 USB 转串口），用 DAP 仿真器把配套程序下载到野火 STM32 开发板（具体型号根据购买的板子而定，每个型号的板子都配套有对应的程序），在计算机上打开串口调试助手，然后复位开发板就可以在调试助手中看到串口的打印信息，在开发板可以看到，LED 在闪烁，同时在串口调试助手也输出了相应的信息，说明任务已经被挂起与恢复，具体见图任务管理实验现象。
[野火]uCOS-III 内核实现与应用开发实战指南

论坛：https://www.firebbs.cn/ 天猫：https://yehuosm.tmall.com
第 22 章 消息队列

同学们，回想一下，在我们裸机的编程中，我们是怎么样用全局的一个数组的呢？

22.1 消息队列的基本概念

队列又称消息队列，是一种常用于任务间通信的数据结构，队列可以在任务与任务间、中断和任务间传递信息。实现了任务接受来自其他任务或中断的不定长度的消息，任务能够从队列里面读取消息。队列中的消息是空时，读取消息的任务将被阻塞，用户还可以指定阻塞的时间定时。在这段时间中，如果队列为空，该任务将保持阻塞状态以等待队列数据有效。当队列中有新消息时，被阻塞的任务会被唤醒并处理新消息；当等待的时间超过了指定的阻塞时间，即使队列中尚无有效数据，任务也会自动从阻塞态转为就绪态。消息队列是一种异步的通信方式。

通过消息队列服务，任务中断服务程序可以将消息放入消息队列中。同样，一个或多个任务可以从消息队列中获得消息。当有多个消息发送到消息队列时，通常是将先进入消息队列的消息先传给任务，也就是说，任务先得到的是最先进入消息队列的消息，即先进先出原则（FIFO）。但是 πC/OS 也支持后进先出原则（LIFO）。

\mu C/OS 中使用队列数据结构实现任务异步通信工作，具有如下特性：

22.2 消息队列工作过程

在 πC/OS-III 中定义了一个数组 OS_CFG_MsgPool[OS_CFG_MSG_POOL_SIZE]，因为在使用消息队列的时候存取消息比较频繁，在系统初始化的时候就将这个大数组的各个元素串成单向链表，组成我们说的消息池，而这些元素我们称之为消息。为什么这里是单向链表而不是我们之前在各种列表中看到的双向链表？因为消息的存取并不需要从链表中间，只需在链表的首尾存取即可，单向链表即够用，使用双向链表反而更复杂。消息池的大小 OS_CFG_MSG_POOL_SIZE 由用户自己定义，该宏定义在 os_cfg_app.h 头文件中。
可能很多同学有疑问，为什么μC/OS 的消息队列要搞一个消息池呢？因为这样子的处理很快，并
且共用了资源，系统中所有被创建的队列都可以从消息池中取出消息，挂载到自身的队列上，以
表示消息队列拥有消息，当消息使用完毕，则又会被释放回到消息池中，其他队列也可以从中取
出消息，这样子的消息资源是能被系统所有的消息队列反复使用。

### 22.2.1 消息池初始化

在系统初始化 (OSInit()) 的时候，系统就会将消息池进行初始化，其中，OS_MsgPoolInit() 函数
就是用来初始化消息池的，OS_MsgPoolInit() 函数的定义位于 os_msg.c 文件中，其源码具体见代
码清单: 消息队列-1。

<table>
<thead>
<tr>
<th>列表 1: 代码清单: 消息队列-1OS_MsgPoolInit() 源码</th>
</tr>
</thead>
<tbody>
<tr>
<td>void OS_MsgPoolInit (OS_ERR *p_err) //返回错误类型</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>OS_MSG *p_msg1;</td>
</tr>
<tr>
<td>OS_MSG *p_msg2;</td>
</tr>
<tr>
<td>OS_MSG_QTY i;</td>
</tr>
<tr>
<td>OS_MSG_QTY loops;</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>#ifdef OS_SAFETY_CRITICAL(1)//如果启用了安全检测</td>
</tr>
<tr>
<td>if (p_err == (OS_ERR *)0) { //如果错误类型为0</td>
</tr>
<tr>
<td>OS_SAFETY_CRITICAL_EXCEPTION(); //执行安全检测异常函数</td>
</tr>
<tr>
<td>return; //返回，停止执行</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>#endif</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>#if OS_CFG_ARG_CHK_EN &gt; 0u(2)//如果启用了参数检测</td>
</tr>
<tr>
<td>if (OSCfg_MsgPoolBasePtr == (OS_MSG *)0) { //如果消息池不存在</td>
</tr>
<tr>
<td>*p_err = OS_ERR_MSG_POOL_NULL_PTR; //错误类型为“消息池指针为空”</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>(下页继续)</td>
</tr>
</tbody>
</table>
20 return;  // 返回，停止执行
21 }
22 if (OSCfg_MsgPoolSize == (OS_MSG_QTY)0) {  // 如果消息池不能存放消息
23 *p_err = OS_ERR_MSG_POOL_EMPTY;  // 错误类型为 “消息池为空”
24 return;  // 返回，停止执行
25 }
26 #endif
27 /* 将消息池里的消息逐条串成单向链表，方便管理 */
28 p_msg1 = OSCfg_MsgPoolBasePtr;
29 p_msg2 = OSCfg_MsgPoolBasePtr;
30 p_msg2++;
31 loops = OSCfg_MsgPoolSize - 1u;
32 for (i = 0u; i < loops; i++) {  // 初始化每一条消息
33     p_msg1->NextPtr = p_msg2;
34     p_msg1->MsgPtr = (void *)0;
35     p_msg1->MsgSize = (OS_MSG_SIZE)0u;
36     p_msg1->MsgTS = (CPU_TS)0u;
37     p_msg1++;
38     p_msg2++;
39 }
40 p_msg1->NextPtr = (OS_MSG*)0;  // 最后一条消息
41 p_msg1->MsgPtr = (void*)0;
42 p_msg1->MsgSize = (OS_MSG_SIZE)0u;
43 p_msg1->MsgTS = (CPU_TS)0u;
44 /* 初始化消息池数据 */
45 OMsgPool.NextPtr = OSCfg_MsgPoolBasePtr; (5)
46 OMsgPool.NbrFree = OSCfg_MsgPoolSize;
47 OMsgPool.NbrUsed = (OS_MSG_QTY)0;
48 OMsgPool.NbrUsedMax = (OS_MSG_QTY)0;
49 *p_err = OS_ERR_NONE;  // 错误类型为 “无错误”
• 代码清单：消息队列-1 (1)：如果启用了安全检测（OS_SAFETY_CRITICAL）这个宏定义，那么在编译代码的时候会包含安全检测，如果 p_err 指针为空，系统会执行安全检测异常函数 OS_SAFETY_CRITICAL_EXCEPTION()，然后退出。

• 代码清单：消息队列-1 (2)：如果启用了参数检测（OS_CFG_ARG_CHK_EN）这个宏定义，那么在编译代码的时候会包含参数检测，如果消息池不存在，系统会返回错误类型为“消息池指针为空”的错误代码，然后退出，不执行初始化操作；如果消息池不能存放消息，系统会返回错误类型为“消息池为空”的错误代码，然后退出，也不执行初始化操作。

• 代码清单：消息队列-1 (3)：系统会将消息池里的消息逐条串成单向链表，方便管理，通过 for 循环将消息池中的每个消息元素（消息）进行初始化，并且通过单链表连接起来。

• 代码清单：消息队列-1 (4)：初始化最后一个消息，每个消息有四个元素，具体见图 OS_MSG
  - NextPtr：指向下一个可用的消息。
  - MsgPtr：指向实际的消息。
  - MsgSize：记录消息的大小（以字节为单位）。
  - MsgTS：记录发送消息时的时间戳。

```latex
\begin{center}
\begin{tabular}{|c|c|c|c|}
\hline
& NextPtr & MsgPtr & MsgSize & MsgTS \\
\hline
OS_MSG \\
\hline
\end{tabular}
\end{center}
```

• 代码清单：消息队列-1 (5)：OSMsgPool 是个全局变量，用来管理内存池的存取操作，它包含以下四个元素，具体见图 OSMsgPool。
  - NextPtr：指向下一个可用的消息。
  - NbrFree：记录消息池中可用的消息个数。

论坛：https://www.firebbs.cn/ 375 天猫：https://yehuosm.tmall.com
- NbrUsed：记录已用的消息个数。
- NbrUsedMax：记录使用的消息峰值数量。

初始化完成的消息池示意图具体见图初始化完成的消息池。

### 22.2.2 消息队列的运作机制

μC/OS 的消息队列控制块由多个元素组成，当消息队列被创建时，编译器会静态为消息队列分配对应的内存空间（因为我们需要自己定义一个消息队列控制块），用于保存消息队列的一些信息，如队列的名字，队列可用的最大消息个数，入队指针、出队指针等。在创建成功的时候，这些内存就被占用了，创建队列的时候用户指定队列的最大消息个数，无法再次更改，每个消息空间可以存放任意类型的数据。

任务或者中断服务程序都可以给消息队列发送消息，当发送消息时，如果队列未满，μC/OS 会将
从消息池中取出一个消息，将消息挂载到队列的尾部，消息中的成员变量 MsgPtr 指向要发送的消息。如果队列已满，则返回错误代码，入队失败。

μC/OS 还支持发送紧急消息，也就是我们所说的后进先出（LIFO）排队，其过程与发送消息几乎一样，唯一的不同是，当发送紧急消息时，发送的消息会挂载到队列的队头而非队尾，这样，接收者就能够优先接收到紧急消息，从而及时进行消息处理。

当某个任务试图读一个队列时，可以指定一个阻塞超时时间。在这段时间中，如果队列为空，该任务将保持阻塞状态以等待队列数据有效。当其他任务或中断服务程序往其等待的队列中写入了数据，该任务将自动由阻塞态转移为就绪态。当等待的时间超过了指定的阻塞时间，即使队列中尚无有效数据，任务也会自动从阻塞态转移为就绪态。

当消息队列不再被使用时，可以对它进行删除操作，一旦删除操作完成，消息队列将被永久性的删除，所有关于队列的信息会被清空，知道再次创建才可使用。

消息队列的运作过程具体见图消息队列运作过程。

22.3 消息队列的阻塞机制

我们使用的消息队列一般是属于某个任务的队列，在很多时候，我们创建的队列，是每个任务都可以去对他进行读写操作的，但是为了保护每个任务对它进行读操作的过程（μC/OS 队列的写操作是没有阻塞的），我们必须要有限制机制，在某个任务对它读操作的时候，必须保证该任务
能正常完成读操作，而不受后来的任务干扰，凡事都有先来后到嘛！

那么，如何实现这个先来后到的机制呢，很简单，因为 μC/OS 已经为我们做好了，我们直接使用就好了，每个对消息队列读的函数，都有这种机制，我称之为阻塞机制。假设一个任务 A 对某个队列进行读操作的时候（也就是我们所说的出队），发现它没有消息，那么此时任务 A 有 3 个选择：第一个选择，任务 A 扭头就走，既然队列没有消息，那我也不等了，干其他事情去，这样子任务 A 不会进入阻塞态；第二个选择，任务 A 还是在这里等等吧，可能过一会队列就有消息，此时任务 A 会进入阻塞状态，在等待着消息的道来，而任务 A 的等待时间就由我们自己定义，比如设置 1000 个系统时钟节拍 tick 的等待，在这 1000 个 tick 到来之前任务 A 都是处于阻塞态，当阻塞的这段时间任务 A 等到了队列的消息，那么任务 A 就会从阻塞态变成就绪态，如果此时任务 A 比当前运行的任务优先级还高，那么，任务 A 就会得到消息并且运行；假如 1000 个 tick 都过去了，队列还没消息，那任务 A 就不等了，从阻塞态中唤醒，返回一个没等到消息的错误代码，然后继续执行任务 A 的其他代码；第三个选择，任务 A 死等，不等到消息就不走了，这样子任务 A 就会进入阻塞态，直到完成读取队列的消息。

假如有多个任务阻塞在一个消息队列中，那么这些阻塞的任务将按照任务优先级进行排序，优先级高的任务将优先获得队列的访问权。

如果发送消息的时候用户选择广播消息，那么在等待中的任务都会收到一样的消息。

### 22.4 消息队列的应用场景

消息队列可以应用于发送不定长消息的场合，包括任务与任务间的消息交换，队列是 μC/OS 中任务与任务间、中断与任务间主要的通讯方式，发送到队列的消息是通过引用方式实现的，这意味这队列存储的数据的地址，我们可以通过这个地址将这个数据读取出来，这样子，无论数据量是多大，其操作时间都是一定的，只是一个指向数据地址指针。
22.5 消息队列的结构

μC/OS 的消息队列由多个元素组成，在信号量被创建时，需要由我们自己定义消息队列（也可以称之为消息队列句柄），因为它是用于保存消息队列的一些信息的，其数据结构 `OS_Q` 除了队列必须的一些基本信息外，还有 `PendList` 链表与 `MsgQ`，为的是方便系统来管理消息队列。其数据结构具体见代码清单: 消息队列-2，示意图具体见消息队列的结构。
列表 2: 代码清单: 消息队列-2 消息队列结构

```c
struct os_q {
    /* ------------------ GENERIC MEMBERS ------------------ */
    OS_OBJ_TYPE Type; (1)
    CPU_CHAR *NamePtr; (2)
    OS_PEND_LIST PendList; (3)
    #if OS_CFG_DBG_EN > 0u
    OS_Q *DbgPrevPtr;
    OS_Q *DbgNextPtr;
    CPU_CHAR *DbgNamePtr;
    #endif
    /* ------------------ SPECIFIC MEMBERS ------------------ */
    OS_MSG_Q MsgQ; (4)
};
```

- 代码清单: 消息队列-2 (1): 消息队列的类型，用户无需理会。
- 代码清单: 消息队列-2 (2): 消息队列的名字。
- 代码清单: 消息队列-2 (3): 等待消息队列的任务列表。

列表 3: 代码清单: 消息队列-3 os_msg_q 结构

```c
struct os_msg_q {
    /* OS_MSG_Q */
    OS_MSG *InPtr; (1) /* 指向要插入队列的下一个 OS_MSG 的指针 */
    OS_MSG *OutPtr; (2) /* 指向要从队列中提取的下一个 OS_MSG 的指针 */
    OS_MSG_QTY NbrEntriesSize; (3) /* 队列中允许的最大消息个数 */
    OS_MSG_QTY NbrEntries; (4) /* 队列中当前的消息个数 */
    OS_MSG_QTY NbrEntriesMax; (5) /* 队列中的消息个数阈值 */
};
```

论坛: [https://www.firebbs.cn/](https://www.firebbs.cn/)
天猫: [https://yehuosm.tmall.com](https://yehuosm.tmall.com)
• 代码清单: 消息队列-3 (1)、(2)：队列中消息也是用单向链表串联起来的，但存取消息不像消息池只是从固定的一端。队列存取消息有两种方式，一种是 FIFO 模式，即先进先出，这个时候消息的存取是在单向链表的两端，一个头一个尾，存取位置可能不一样产生了这四个输入指针和输出指针，具体见图 FIFO 模式。另一种是 LIFO 模式，后进先出，这个时候消息的存取都是在单向链表的一端，仅仅用 OutPtr 就足够指示存取的位置，具体见图 LIFO 模式。当队列中已经存在比较多的消息没有处理，这个时候有个紧急的消息需要马上传送到其他任务去的时候就可以在发布消息的时候选择 LIFO 模式。

• 代码清单: 消息队列-3 (3)：消息队列最大可用的消息个数，消息队列创建的时候由用户指定这个值的大小。

• 代码清单: 消息队列-3 (4)：记录消息队列中当前的消息个数，每发送一个消息，若没有任务在等待该消息队列的消息，那么新发送的消息被插入此消息队列后此值加 1，NbrEntries 的大小不能超过 NbrEntriesSize。

• 代码清单: 消息队列-3 (5)：记录队列最多的时候拥有的消息个数。
22.6 消息队列常用函数讲解

22.6.1 创建消息队列函数 OSQCreate()

要使用 μC/OS 的消息队列必须先声明和创建消息队列，OSQCreate() 用于创建一个新的队列。队列就是一个数据结构，用于任务间的数据的传递。每创建一个新的队列都需要为其分配 RAM，在创建的时候我们需要自己定义一个消息队列结构体，其内存是由编译器自动分配的，OSQCreate 的源码具体见代码清单: 消息队列-4。
列表 4: 代码清单: 消息队列-4OSQCreate() 源码

```c
void OSQCreate (OS_Q *p_q, (1) // 消息队列指针
        CPU_CHAR *p_name, (2) // 消息队列名称
        OS_MSG_QTY max_qty, (3) // 消息队列大小（不能为 0）
        OS_ERR *p_err) (4) // 返回错误类型
{
    CPU_SR_ALLOC(); (5) // 使用到临界段 (在关 / 开中断时) 时必须用到该宏，该宏声明和
    // 定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
    // SR (临界段关中断只需保存 SR)，开中断时将该值还原。

    #ifdef OS_SAFETY_CRITICAL (6) // 如果启用了安全检测
        if (p_err == (OS_ERR *)0) { // 如果错误类型实例为空
            OS_SAFETY_CRITICAL_EXCEPTION(); // 执行安全检测异常函数
            return; // 返回，停止执行
        }
    #endif

    #ifdef OS_SAFETY_CRITICAL_IEC61508 // 如果启用了安全关键
        // 如果在调用 OSSafetyCriticalStart() 后创建
        if (OSSafetyCriticalStartFlag == DEF_TRUE) {
            *p_err = OS_ERR_ILLEGAL_CREATE_RUN_TIME; // 错误类型为 “非法创建内核对象”
            return; // 返回，停止执行
        }
    #endif

    #if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u(7) // 如果启用了中断中非法调用检测
        if (OSIntNestingCtr > (OS_NESTING_CTR)0) { // 如果该函数是在中断中被调用
            *p_err = OS_ERR_CREATE_ISR; // 错误类型为 “在中断中创建对象”
            return; // 返回，停止执行
        }
    #endif
```

(下页继续)
```c
#if OS_CFG_ARG_CHK_EN > 0u(8)  //如果启用了参数检测
    if (p_q == (OS_Q *)0) {  //如果 p_q 为空
        *p_err = OS_ERR_OBJ_PTR_NULL;  //错误类型为“创建对象为空”
        return;  //返回，停止执行
    }
    if (max_qty == (OS_MSG_QTY)0) {  //如果 max_qty = 0
        *p_err = OS_ERR_Q_SIZE;  //错误类型为“队列空间为 0”
        return;  //返回，停止执行
    }
#endif

OS_CRITICAL_ENTER();  //进入临界段
p_q->Type = OS_OBJ_TYPE_Q;  //标记创建对象数据结构为消息队列
p_q->NamePtr = p_name;  //标记消息队列的名称
OS_MsgQInit(&p_q->MsgQ,  //初始化消息队列
    max_qty);
OS_PendListInit(&p_q->PendList);  //初始化该消息队列的等待列表

#if OS_CFG_DBG_EN > 0u  //如果启用了调试代码和变量
    OS_QDbgListAdd(p_q);  //将该队列添加到消息队列双向调试链表
#endif

OSQQty++;  //消息队列个数加 1

OS_CRITICAL_EXIT_NO_SCHED();  //退出临界段（无调度）
*p_err = OS_ERR_NONE;  //错误类型为“无错误”
```

- **代码清单: 消息队列-1(1)**：消息队列指针，在创建之前我们要定义一个队列的数据结构，然后将消息队列指针指向该队列。
- **代码清单: 消息队列-1(2)**：消息队列的名称，字符串形式，这个名称一般与消息队列名称一致，为了方便调试。
• 代码清单: 消息队列-4 (3): 消息队列的大小，也就是消息队列的可用消息个数最大为多少，一旦确定无法修改。

• 代码清单: 消息队列-4 (4): 用于保存返回的错误类型。

• 代码清单: 消息队列-4 (5): 使用到临界段 (在关/开中断时) 时必须用到该宏，该宏声明和定义一个局部变量，用于保存关中断前的 CPU 状态寄存器 SR (临界段关中断只需保存 SR)，开中断时将该值还原。

• 代码清单: 消息队列-4 (6): 如果启用了安全检测，在编译时则会包含安全检测相关的代码，如果错误类型实参为空，系统会执行安全检测异常函数，然后返回，停止执行。

• 代码清单: 消息队列-4 (7): 如果启用了中断中非法调用检测，在编译时则会包含中断非法调用检测相关的代码，如果该函数是在中断中被调用，则是非法的，返回错误类型为“在中断中创建对象”的错误代码，并且退出，不执行创建队列操作。

• 代码清单: 消息队列-4 (8): 如果启用了参数检测，在编译时则会包含参数检测相关的代码，如果 p q 参数为空，返回错误类型为“创建对象为空”的错误代码，并且退出，不执行创建队列操作。

• 代码清单: 消息队列-4 (9): 如果 max_qty 参数为 0，表示不存在消息空间，这也是错误的，返回错误类型为“队列空间为 0”的错误代码，并且退出，不执行创建队列操作。

• 代码清单: 消息队列-4 (10): 标记创建对象数据结构为消息队列。

• 代码清单: 消息队列-4 (11): 初始化消息队列的名称。

• 代码清单: 消息队列-4 (12): 调用 OS_MsgQInit() 函数初始化消息队列，其实就是初始化消息队列结构的相关信息，该函数源码具体见 代码清单: 消息队列-5。

列表 5: 代码清单: 消息队列-5OS_MsgQInit() 源码

```c
void OS_MsgQInit (OS_MSG_Q *p_msg_q, //消息队列指针
                  OS_MSG_QTY size)   //消息队列空间
{
    p_msg_q->NbrEntriesSize = (OS_MSG_QTY)size; //消息队列可存放消息数目
    p_msg_q->NbrEntries     = (OS_MSG_QTY)0;    //消息队列目前可用消息数
}
```

(下页继续)
p_msg_q->NbrEntriesMax = (OS_MSG_QTY)0;    //可用消息数的最大历史记录
p_msg_q->InPtr   = (OS_MSG *)0;            //队列的入队指针
p_msg_q->OutPtr  = (OS_MSG *)0;            //队列的出队指针
}

- **代码清单: 消息队列-4 (13)**: 初始化消息队列的阻塞列表，消息队列的阻塞列表是用于记录阻塞在此消息队列上的任务。

- **代码清单: 消息队列-4 (14)**: OSQQty 是系统中的一个全局变量，用于记录已经创建的消息队列个数，现在创建队列完毕，所以该变量要加一。

消息队列创建完成的示意图具体见图消息队列创建完成。
在创建消息队列的时候，是需要用户自己定义消息队列的句柄的。但是注意了，定义了队列的句柄并不等于创建了队列，创建队列必须是调用消息队列创建函数进行创建，否则，以后根据队列句柄使用消息队列的其他函数的时候会发生错误，用户通过消息队列句柄就可使用消息队列进行发送与获取消息的操作，用户可以根据返回的错误代码进行判断消息队列是否创建成功，消息队列创建函数 OSQCreate() 使用实例具体见代码清单: 消息队列-6。
列表 6: 代码清单: 消息队列-6 OSQCreate() 使用实例

```c
OS_Q queue;            //声明消息队列
OS_ERR err;

/* 创建消息队列 queue */
OSQCreate ((OS_Q     *)&queue,       //指向消息队列的指针
    (CPU_CHAR  *)&"Queue For Test",  //队列的名字
    (OS_MSG_QTY)20,                   //最多可存放消息的数目
    (OS_ERR    *)&err);              //返回错误类型
```

### 22.6.2 消息队列删除函数 OSQDel()

队列删除函数是根据队列结构 (队列句柄) 直接删除的，删除之后这个消息队列的所有信息都会被系统清空，而且不能再次使用这个消息队列了，但是需要注意的是，如果某个消息队列没有被定义，那也是无法被删除的。想要使用消息队列删除函数就必须将 OS_CFG_Q_DEL_EN 宏定义配置为 1，其函数源码具体见 代码清单: 消息队列-7。

列表 7: 代码清单: 消息队列-7OSQDel() 源码

```c
#if OS_CFG_Q_DEL_EN > 0u  //如果启用了 OSQDel() 函数
OS_OBJ_QTY OSQDel (OS_Q *p_q, (1)//消息队列指针
                     OS_OPT opt, (2)//选项
                     OS_ERR *p_err) (3)//返回错误类型
{
    OS_OBJ_QTY cnt;
    OS_OBJ_QTY nbr_tasks;
    OS_PEND_DATA *p_pend_data;
    OS_PEND_LIST *p_pend_list;
    OS_TCB *p_tcb;
```

(下页继续)
CPU_TS    ts;
CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必须用到该宏
//定义一个局部变量，用于保存关中断前的CPU状态寄存器
//SR（临界段关中断只需保存SR），开中断时将该值还原。
#endif OS_SAFETY_CRITICAL(4) //如果启用（默认禁用）了安全检测
if (p_err == (OS_ERR *)) {  //如果错误类型实参为空
    OS_SAFETY_CRITICAL_EXCEPTION(); //执行安全检测异常函数
    return ((OS_OBJ_QTY)0);  //返回 0 (有错误)，停止执行
}  #endif

#if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u(5) //如果启用了中断中非法调用检测
if (OSIntNestingCtr > (OS_NESTING_CTR)0) { //如果该函数在中断中被调用
    *p_err = OS_ERR_DEL_ISR;  //错误类型为 “在中断中中止等待”
    return ((OS_OBJ_QTY)0);  //返回 0 (有错误)，停止执行
}  #endif

#if OS_CFG_ARG_CHK_EN > 0u(6) //如果启用了参数检测
if (p_q == (OS_Q *)) { //如果 p_q 为空
    *p_err = OS_ERR_OBJ_PTR_NULL;  //错误类型为 “对象为空”
    return ((OS_OBJ_QTY)0u);  //返回 0 (有错误)，停止执行
}
switch (opt) { //根据选项分类处理
    case OS_OPT_DEL_NO_PEND:  //如果选项在预期内
        break;  //直接跳出
    case OS_OPT_DEL_ALWAYS:
        default:  //如果选项超出预期
            *p_err = OS_ERR_OPT_INVALID;  //返回 0 (有错误)，停止执行
            return ((OS_OBJ_QTY)0u);  //返回 0 (有错误)，停止执行

if OS_CFG_OBJ_TYPE_CHK_EN > 0u(9) //如果启用了对象类型检测
    if (p_q->Type != OS_OBJ_TYPE_Q) { //如果 p_q 不是消息队列类型
        *p_err = OS_ERR_OBJ_TYPE; //错误类型为“对象类型有误”
        return ((OS_OBJ_QTY)0); //返回 0 (有错误)，停止执行
    }
#endif

CPU_CRITICAL_ENTER(); //关中断
p_pend_list = &p_q->PendList; (10) //获取消息队列的等待列表
cnt = p_pend_list->NbrEntries; (11) //获取等待该队列的任务数
nbr_tasks = cnt; (12) //按照任务数目逐个处理
switch (opt) { (13) //根据选项分类处理
    case OS_OPT_DEL_NO_PEND: (14) //如果只在没有任务等待的情况下删除队列
        if (nbr_tasks == (OS_OBJ_QTY)0) { (15) //如果没有任务在等待该队列
            #if OS_CFG_DBG_EN > 0u //如果启用了调试代码和变量
                OS_QDbgListRemove(p_q); //将该队列从消息队列调试列表移除
            #endif
                OSQQty--; //消息队列数目减 1
                OS_QClr(p_q); (16) //清除该队列的内容
                CPU_CRITICAL_EXIT(); //开中断
                *p_err = OS_ERR_NONE; (17) //错误类型为“无错误”
            } else { (18) //如果有任务在等待该队列
                CPU_CRITICAL_EXIT(); //开中断
                *p_err = OS_ERR_TASK_WAITING; //错误类型为“有任务在等待该队列”
            }
        break;
    case OS_OPT_DEL_ALWAYS: (19) //如果必须删除信号量
        OS_CRITICAL_ENTER_CPU_EXIT(); //进入临界段，重开中断
        break;
    #endif
#endif

论坛: https://www.firebbs.cn/天猫: https://yehuosm.tmall.com

(续上页)
ts = OS_TS_GET(); //获取时闰戳

while (cnt > 0u) { 
    p_pend_data = p_pend_list->HeadPtr;
    p_tcb = p_pend_data->TCBPtr;
    OS_PendObjDel((OS_PEND_OBJ *)((void *)p_q),
                   p_tcb,
                   ts);
    cnt--;
} //逐个移除该队列等待列表中的任务

if OS_CFG_DBG_EN > 0u //如果启用了调试代码和变量
    OS_QDbgListRemove(p_q); //将该队列从消息队列调试列表中删除
#endif

OSQQty--; //消息队列数日减 1
OS_QClr(p_q); //清除消息队列内容
OS_CRITICAL_EXIT_NO_SCHED(); //退出临界段 (无调度)
OSSched(); //调度任务
*p_err = OS_ERR_NONE; //错误类型为 “无错误”
break; //跳出

default: //如果选项超出了预期
    CPU_CRITICAL_EXIT(); //开中断
    *p_err = OS_ERR_OPT_INVALID; //错误类型为 “选项非法”
    break; //跳出
}

return (nbr_tasks); //返回删除队列前等待其的任务数

• 代码清单: 消息队列-7 (1): 消息队列指针，指向要删除的消息队列。
• 代码清单: 消息队列-7 (2): 操作消息队列的选项，具体在后面讲解。
• 代码清单: 消息队列-7 (3): 用于保存返回的错误类型。

论坛: https://www.firebbs.cn/ 　　天猫: https://yehuosm.tmall.com
• 代码清单: 消息队列-7 (4): 如果启用（默认禁用）了安全检测，在编译时则会包含安全检测相关的代码，如果错误类型实参为空，系统会执行安全检测异常函数，然后返回，停止执行。

• 代码清单: 消息队列-7 (5): 如果启用了中断中非法调用检测，在编译时则会包含中断非法调用检测相关的代码，如果该函数是在中断中被调用，则是非法的，返回错误类型为“在中断中删除”的错误代码，并且退出，不执行删除队列操作。

• 代码清单: 消息队列-7 (6): 如果启用了参数检测，在编译时则会包含参数检测相关的代码，如果 p/q 参数为空，返回错误类型为“删除对象为空”的错误代码，并且退出，不执行删除队列操作。

• 代码清单: 消息队列-7 (7): 根据选项分类处理，如果选项在预期内，直接跳出 switch 语句。

• 代码清单: 消息队列-7 (8): 如果选项超出预期，就退出，不执行删除队列操作。

• 代码清单: 消息队列-7 (9): 如果启用了对象类型检查，在编译时则会包含对象类型检查相关代码，如果 p/q 不是消息队列类型，那么返回错误类型为“对象类型有误”的错误代码，并且退出，不执行删除队列操作。

• 代码清单: 消息队列-7 (10): 程序能执行到这里，说明传入的参数都是正确的，此时可以执行删除队列操作，系统首先获取消息队列中的等待列表，通过 p_pend_list 变量进行消息队列等待列表的访问。

• 代码清单: 消息队列-7 (11): 获取阻塞在该队列上的任务个数。

• 代码清单: 消息队列-7 (12): 按照任务数目逐个处理。

• 代码清单: 消息队列-7 (13): 根据选项分类处理。

• 代码清单: 消息队列-7 (14): 如果删除选项是只在没有任务等待的情况下删除队列，系统就会判断有没有任务阻塞在改队列上。

• 代码清单: 消息队列-7 (15): 如果没有任务在等待该队列，那就执行删除操作。

• 代码清单: 消息队列-7 (16): 系统的消息队列数目减 1。

• 代码清单: 消息队列-7 (17): 清除该队列的内容。

• 代码清单: 消息队列-7 (18): 返回错误类型为“无错误”的错误代码。
• 代码清单: 消息队列-7 (19): 如果有任务在等待该队列，那么就无法进行删除操作，返回错误类型为“有任务在等待该队列”的错误代码。

• 代码清单: 消息队列-7 (20): 如果删除操作的选项是必须删除消息队列，无论是否有任务阻塞在该消息队列上，系统都会进行删除操作。

• 代码清单: 消息队列-7 (21): 根据消息队列当前等待的任务个数，逐个移除该队列待列表中的任务。

• 代码清单: 消息队列-7 (22): 调用 OS_PendObjDel() 函数将阻塞在内核对象（如信号量）上的任务从阻塞态恢复，此时系统在删除内核对象，删除之后，这些等待事件的任务需要被恢复，其源码具体见 代码清单: 消息队列-8。每移除一个，消息队列的任务个数就减一，当没有任务阻塞在该队列上，就进行删除队列操作。

列表 8: 代码清单: 消息队列-8OS_PendObjDel() 源码

```
void OS_PendObjDel (OS_PEND_OBJ *p_obj, (1) //被删除对象的类型
                      OS_TCB   *p_tcb, (2) //任务控制块指针
                      CPU_TS   ts) (3) //信号量被删除时的时间戳
{
  switch (p_tcb->TaskState) (4) //根据任务状态分类处理
  {
    case OS_TASK_STATE_RDY: //如果任务是就绪状态
    case OS_TASK_STATE_DLY: //如果任务是延时状态
    case OS_TASK_STATE_SUSPENDED: //如果任务是挂起状态
    case OS_TASK_STATE_DLY_SUSPENDED: //如果任务是在延时中被挂起
      break; (5)
    //这些情况均与等待无关，直接跳出
    case OS_TASK_STATE_PEND: //如果任务是无限期等待状态
    case OS_TASK_STATE_PEND_TIMEOUT: //如果任务是有限期等待状态
      if (p_tcb->PendOn == OS_TASK_PEND_ON_MULTI)
        //如果任务在等待多个信号量或消息队列
        {
          OS_PendObjDel1(p_obj, (6) //强制解除任务对某一对象的等待
```

论坛: https://www.firebbs.cn/  394  天猫: https://yehuosm.tmall.com
p_tcb,
    ts);  (6)
}

#if (OS_MSG_EN > 0u) (7)//如果启用了任务队列或消息队列
    p_tcb->MsgPtr  = (void *)0;  //清除（复位）任务的消息域
    p_tcb->MsgSize = (OS_MSG_SIZE)0u;
#endif

    p_tcb->TS  = ts;  (8)
    //保存等待被中止时的时间戳到任务控制块
    OS_PendListRemove(p_tcb); (9)//将任务从所有等待列表中移除
    OS_TaskRdy(p_tcb);  (10)//让任务进准备运行
    p_tcb->TaskState = OS_TASK_STATE_RDY; (11)//修改任务状态为就绪状态
    p_tcb->PendStatus = OS_STATUS_PEND_DEL; (12)//标记任务的等待对象被删除
    p_tcb->PendOn = OS_TASK_PEND_ON_NOTHING; (13)//标记任务目前没有等待任何对象
    break;  //跳出

    case OS_TASK_STATE_PEND_SUSPENDED: //如果任务在无期限等待中被挂起
        case OS_TASK_STATE_PEND_TIMEOUT_SUSPENDED: //如果任务在有限期等待中被挂起
            if (p_tcb->PendOn == OS_TASK_PEND_ON_MULTI) //如果任务在等待多个信号量或消息队列
                {
                    OS_PendObjDel1(p_obj,  //强制解除任务对某一对象的等待
                        p_tcb,
                        ts);  (14)
                }
#endif (OS_MSG_EN > 0u) (15)//如果启用了任务队列或消息队列
    p_tcb->MsgPtr  = (void*)0;  //清除（复位）任务的消息域
    p_tcb->MsgSize = (OS_MSG_SIZE)0u;
#endif

    p_tcb->TS  = ts;  (17)
    //保存等待被中止时的时间戳到任务控制块

(下页继续)
OS_TickListRemove(p_tcb);      //让任务脱离节拍列表
OS_PendListRemove(p_tcb);      //将任务从所有等待列表中移除
p_tcb->TaskState  = OS_TASK_STATE_SUSPENDED; (20)   //修改任务状态为挂起状态
p_tcb->PendStatus = OS_STATUS_PEND_DEL; (21)   //标记任务的等待对象被删除
p_tcb->PendOn    = OS_TASK_PEND_ON_NOTHING;      //标记任务目前没有等待任何对象
break;                           //跳出

default:
break;                           //不需处理，直接跳出
}

• 代码清单: 消息队列-8 (1): 被删除对象的类型（如消息队列、信号量、互斥量、事件等）。
• 代码清单: 消息队列-8 (2): 任务控制块指针。
• 代码清单: 消息队列-8 (3): 内核对象被删除时的时间戳。
• 代码清单: 消息队列-8 (4): 根据任务状态分类处理。
• 代码清单: 消息队列-8 (5): 如果任务是就绪状态、延时状态、挂起状态或者是在延时中被挂起，这些任务状态与等待内核对象是无关的，在内核对象被删除的时候无需进行任何操作。
• 代码清单: 消息队列-8 (6): 如果任务是无限期等待状态或者是有限期等待状态，那么在内核对象被删除的时候需要将这些任务恢复。如果这些任务在等待多个内核对象（信号量或消息队列等），那么就需要强制解除任务对某一对象的等待，比如现在删除的是消息队列，那么就将该任务对消息队列的等待进行解除。
• 代码清单: 消息队列-8 (7): 如果启用了任务队列或消息队列，清除（复位）任务的消息指针，任务等待的消息大小为 0。
• 代码清单: 消息队列-8 (8): 保存等待被中止时的时间戳到任务控制块。

论坛: https://www.firebbs.cn/ 天猫: https://yehuosm.tmall.com
• 代码清单: 消息队列-8 (9): 调用 OS_PendListRemove() 函数将任务从所有等待列表中移除。
• 代码清单: 消息队列-8 (10): 调用 OS_TaskRdy() 函数让任务进入就绪态参与系统调度，准备运行。
• 代码清单: 消息队列-8 (11): 修改任务状态为就绪状态。
• 代码清单: 消息队列-8 (12): 标记任务的等待对象被删除。
• 代码清单: 消息队列-8 (13): 标记任务目前没有等待任何对象。
• 代码清单: 消息队列-8 (14): 如果任务在无期限等待中被挂起或者在有限期等待中被挂起，也是需要将这些等待内核对象的任务从等待中移除，但是由于在等待中被挂起，那么就不会将这些任务恢复为就绪状态，仅仅是将任务从等待列表中移除。如果任务在等待多个信号量或消息队列，同样也是将任务从等待的对象中移除即可。
• 代码清单: 消息队列-8 (15): 如果启用了任务队列或消息队列。
• 代码清单: 消息队列-8 (16): 需要清除（复位）任务的消息指针，任务等待的消息大小为 0。
• 代码清单: 消息队列-8 (17): 保存等待被中止时的时间戳到任务控制块。
• 代码清单: 消息队列-8 (18): 调用 OS_TickListRemove() 函数让任务脱离节拍列表。
• 代码清单: 消息队列-8 (19): 调用 OS_PendListRemove() 函数将任务从所有等待列表中移除。
• 代码清单: 消息队列-8 (20): 修改任务状态为挂起状态，因为在等待中被挂起，此时即使任务不等的内核对象了，它还是处于挂起态。
• 代码清单: 消息队列-8 (21): 任务的等待对象被删除，标记任务目前没有等待任何对象。
• 代码清单: 消息队列-8 (22): 如果任务状态超出预期，不需处理，直接跳出。
• 代码清单: 消息队列-7 (23): 系统的消息队列数目减 1。
• 代码清单: 消息队列-7 (24): 清除消息队列内容。
• 代码清单: 消息队列-7 (25): 发起一次调度任务。
• 代码清单: 消息队列-7 (26): 返回错误类型为“无错误”的错误代码。
• 代码清单: 消息队列-7 (27): 而如果选项超出预期，返回错误类型为“选项非法”的错误代码，然后退出。
消息队列删除函数 OSQDel() 的使用也是很简单的，只需要传入要删除的消息队列的句柄与选项，还有保存返回的错误类型即可，调用函数时，系统将删除这个消息队列。需要注意的是在调用删除消息队列函数前，系统应存在已创建的消息队列。如果删除消息队列时，有任务正在等待消息，则不应该进行删除操作，删除之后的消息队列就不可用了，删除消息队列函数 OSQDel() 的使用实例具体见代码清单: 消息队列-9。

列表 9: 代码清单: 消息队列-9 消息队列删除函数 OSQDel() 使用实例

```c
OS_Q queue;          //声明消息队列
OS_ERR err;

/* 删除消息队列 queue */
OSQDel ((OS_Q *) &queue,           //指向消息队列的指针
OS_OPT_DEL_NO_PEND,
(OS_ERR *) &err);          //返回错误类型
```

22.6.3 消息队列发送函数 OSQPost()

任务或者中断服务程序都可以给消息队列发送消息，当发送消息时，如果队列未满，就说明运行信息入队。μC/OS 会从消息池中取出一个消息，挂载到消息队列的末尾（FIFO 发送方式），如果是 LIFO 发送方式，则将消息挂载到消息队列的头部，然后将消息中 MsgPtr 成员变量指向要发送的消息（此处可以理解为添加要发送的信息到消息（块）中），如果系统有任务阻塞在消息队列中，要么在发送了消息队列的时候，会将任务解除阻塞，其源码具体见代码清单: 消息队列-10。

列表 10: 代码清单: 消息队列-10 OSQPost() 源码

```c
void OSQPost (OS_Q *p_q,    //消息队列指针
              void *p_void, (2) //消息指针
              OS_MSG_SIZE msg_size,(3) //消息大小 (单位：字节)
              OS_OPT opt, (4) //选项
```

论坛: https://www.firebbs.cn/ 398 天猫: https://yehuosm.tmall.com
OS_ERR *p_err) (5) //返回错误类型
{
    CPU_TS ts;
}

#endif OS_SAFETY_CRITICAL(6) //如果启用（默认禁用）了安全检测
if (p_err == (OS_ERR *)0) { //如果错误类型实参为空
    OS_SAFETY_CRITICAL_EXCEPTION(); //执行安全检测异常函数
    return; //返回，停止执行
}
#endif

#endif OS_CFG_ARG_CHK_EN > 0u(7) //如果启用了参数检测
if (p_q == (OS_Q *)0) { //如果 p_q 为空
    *p_err = OS_ERR_OBJ_PTR_NULL; //错误类型为“内核对象为空”
    return; //返回，停止执行
}
switch (opt) { (8) //根据选项分类处理
    case OS_OPT_POST_FIFO: //如果选项在预期内
    case OS_OPT_POST_LIFO:
    case OS_OPT_POST_FIFO | OS_OPT_POST_ALL:
    case OS_OPT_POST_LIFO | OS_OPT_POST_ALL:
    case OS_OPT_POST_FIFO | OS_OPT_POST_NO_SCHED:
    case OS_OPT_POST_LIFO | OS_OPT_POST_NO_SCHED:
    case OS_OPT_POST_FIFO | OS_OPT_POST_ALL | OS_OPT_POST_NO_SCHED:
    case OS_OPT_POST_LIFO | OS_OPT_POST_ALL | OS_OPT_POST_NO_SCHED:
    break; //直接跳出

    default: (9) //如果选项超出预期
        *p_err = OS_ERR_OPT_INVALID; //错误类型为“选项非法”
        return; //返回，停止执行

(下页继续)


```c
#endif

#if OS_CFG_OBJ_TYPE_CHK_EN > 0u(10) // 如果启用了对象类型检测
    if (p_q->Type != OS_OBJ_TYPE_Q) { // 如果 p_q 不是消息队列类型
        p_err = OS_ERR_OBJ_TYPE;        // 错误类型为“对象类型错误”
        return;                         // 返回，停止执行
    }
#endif

ts = OS_TS_GET();                      // 获取时间戳

#if OS_CFG_ISR_POST_DEFERRED_EN > 0u(11) // 如果启用了中断延迟发布
    if (OSIntNestingCtr > (OS_NESTING_CTR)0) { // 如果该函数在中断中被调用
        OS_IntQPost((OS_OBJ_TYPE)OS_OBJ_TYPE_Q, // 将该消息发布到中断消息队列
                     (void  *)p_q,
                     (void  *)p_void,
                     (OS_MSG_SIZE)msg_size,
                     (OS_FLAGS )0,
                     (OS_OPT  )opt,
                     (CPU_TS   )ts,
                     (OS_ERR   *)p_err);
        return;                           // 返回（尚未发布），停止执行
    }
#endif

OS_QPost(p_q,                          // 将消息按照普通方式
         p_void,                         //
         msg_size,
         opt,
         ts,
         p_err);                         //
```

(下页继续)


- **代码清单: 消息队列-10 (1)**: 消息队列指针，指向要发送消息的队列。

- **代码清单: 消息队列-10 (2)**: 消息指针，指向任何类型的消息数据。

- **代码清单: 消息队列-10 (3)**: 消息的大小（单位：字节）。

- **代码清单: 消息队列-10 (4)**: 发送消息的选项，在 os.h 中定义，具体见代码清单: 消息队列-11。

### 列表11: 代码清单: 消息队列-11 发送消息的选项

```c
#define OS_OPT_POST_FIFO (OS_OPT)(0x0000u) /* 默认采用 FIFO 方式发送 */
#define OS_OPT_POST_LIFO (OS_OPT)(0x0010u) /* 采用 LIFO 方式发送消息 */
#define OS_OPT_POST_1   (OS_OPT)(0x0000u) /* 将消息发布到最高优先级的等待任务 */
#define OS_OPT_POST_ALL (OS_OPT)(0x0200u) /* 向所有等待的任务广播消息 */
#define OS_OPT_POST_NO_SCHED (OS_OPT)(0x8000u) /* 发送消息但是不进行任务调度 */
```

- **代码清单: 消息队列-10 (5)**: 保存返回的错误类型，用户可以根据此变量得知错误的原因。

- **代码清单: 消息队列-10 (6)**: 如果启用（默认禁用）了安全检测，在编译时则会包含安全检测相关的代码，如果错误类型实参为空，系统会执行安全检测异常函数，然后返回，停止执行。

- **代码清单: 消息队列-10 (7)**: 如果启用了参数检测，在编译时则会包含参数检测相关的代码，如果 p_q 参数为空，返回错误类型为“内核对象为空”的错误代码，并且退出，不执行发送消息操作。

- **代码清单: 消息队列-10 (8)**: 根据 opt 选项进行分类处理，如果选项在预期内，直接退出，其实在这里只是对选项的一个检查，看看传入的选项参数是否正确。

- **代码清单: 消息队列-10 (9)**: 如果 opt 选项超出预期，返回错误类型为“选项非法”的错误代码，并且退出，不执行发送消息操作。
• 代码清单: 消息队列-10 (10): 如果启用了对象类型检测，在编译时则会包含对象类型检测相关代码，如果 p_q 不是消息队列类型，那么返回错误类型为“对象类型有误”的错误代码，并且退出，不执行发送消息操作。

• 代码清单: 消息队列-10 (11): 如果启用了中断延迟发布，并且发送消息的函数是在中断中被调用，此时就不该立即发送消息，而是将消息的发送放在指定发布任务中，此时系统将将消息发布到租单消息队列中，等待到中断发布任务唤醒再发送消息，该函数会在中断管理章节详细讲解。

• 代码清单: 消息队列-10 (12): 而如果不是在中断中调用 OSQPost() 函数，或者未启用中断延迟发布，则直接调用 OS_QPost() 函数进行消息的发送。OS_QPost() 函数源码具体见 代码清单: 消息队列-12。

列表 12: 代码清单: 消息队列-12 OS_QPost() 源码

```c
void OS_QPost(OS_Q *p_q,    //消息队列指针
    void *p_void,       //消息指针
    OS_MSG_SIZE msg_size, //消息大小（单位：字节）
    OS_OPT opt,        //选项
    CPU_TS ts,         //消息被发布时的时间戳
    OS_ERR *p_err)     //返回错误类型
{
    OS_OBJ_QTY cnt;
    OS_OPT post_type;
    OS_PEND_LIST *p_pend_list;
    OS_PEND_DATA *p_pend_data;
    OS_PEND_DATA *p_pend_data_next;
    OS_TCB *p_tcb;
    CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必须用到该宏，该宏声明和
    //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
    //SR（临界段关中断只需保存 SR），开中断时将该值还原。
    OS_CRITICAL_ENTER(); //进入临界段
    p_pend_list = &p_q->PendList; //取出该队列的等待列表
    ...
}
```

(下页继续)
if (p_pend_list->NbrEntries == (OS_OBJ_QTY)0)     (1) //如果没有任务在等待该队列
{
    if (((opt & OS_OPT_POST_LIFO) == (OS_OPT)0) //把消息发布到队列的末端
    {
        post_type = OS_OPT_POST_FIFO;     (2)
    }
    else //把消息发布到队列的前端
    {
        post_type = OS_OPT_POST_LIFO;     (3)
    }
}

OS_MsgQPut(&p_q->MsgQ, //把消息放入消息队列
            p_void,
            msg_size,
            post_type,
            ts,
            p_err);     (4)
OS_CRITICAL_EXIT(); //退出临界段
return; //返回，执行完毕
}
/* 如果有任务在等待该队列 */
if ((opt & OS_OPT_POST_ALL) != (OS_OPT)0)     (5) //如果要把消息发布给所有等待任务
{
    cnt = p_pend_list->NbrEntries; //获取等待任务数目
}
else //如果要把消息发布给一个等待任务
{
    cnt = (OS_OBJ_QTY)1;     (6) //要处理的任务数目为 1
}
p_pend_data = p_pend_list->HeadPtr; //获取等待列表的头部（任务）
while (cnt > 0u) {  
    p_tcb = p_pend_data->TCBPtr;  
    p_pend_data_next = p_pend_data->NextPtr;  
    OS_Post((OS_PEND_OBJ *)(void *)p_q),  //把消息发布给任务  
       p_tcb,  
       p_void,  
       msg_size,  
       ts);  
    p_pend_data = p_pend_data_next;  
    cnt--;  
}  
OS_CRITICAL_EXIT_NO_SCHED();  //退出临界段（无调度）  
if ((opt & OS_OPT_POST_NO_SCHED) == (OS_OPT))  //如果没选择“发布完不调度任务”  
    {  
        OSSched();  
    }  
*p_err = OS_ERR_NONE;  //错误类型为“无错误”

- 代码清单: 消息队列-12 (1): 使用局部变量 p_pend_list 获取队列的等待列表，然后查看等待列表中是否有任务在等待，分情况处理，因为没有任务等待就直接将消息放入队列中即可，而有任务在等待则有可能需要唤醒该任务。

- 代码清单: 消息队列-12 (2): 如果没有任务在等待，系统就会看看用户发送消息的选项是什么，如果是发送到细细道来的末端（队尾，FIFO 方式），那么表示发送类型的 post_type 变量就被设置为 OS_OPT_POST_FIFO。

- 代码清单: 消息队列-12 (3): 否则就设置为 OS_OPT_POST_LIFO，采用 LIFO 方式发送消息。将消息发送到队列的前端（对头）。

列表 13: 代码清单: 消息队列-13 OS_MsgQPut() 源码

```c
void OS_MsgQPut (OS_MSG_Q *p_msg_q, //消息队列指针
                 void *p_void,     //消息指针
                 OS_MSG_SIZE msg_size, //消息大小 (单位: 字节)
                 OS_OPT opt,       //选项
                 CPU_TS ts,        //消息被发布时的时间戳
                 OS_ERR *p_err)    //返回错误类型
{
    OS_MSG *p_msg;
    OS_MSG *p_msg_in;

    #ifdef OS_SAFETY_CRITICAL //如果启用了安全检测
        if (p_err == (OS_ERR *)0)        //如果错误类型实参为空
            {
                OS_SAFETY_CRITICAL_EXCEPTION(); //执行安全检测异常函数
                return;                       //返回, 停止执行
            }
    #endif

    if (p_msg_q->NbrEntries >= p_msg_q->NbrEntriesSize) //如果消息队列已没有可用空间
        {
            *p_err = OS_ERR_Q_MAX;       //错误类型为“队列已满”
            return;                     //返回, 停止执行
        }

    if (OSMsgPool.NbrFree == (OS_MSG_QTY)0) //如果消息池没有可用消息
        {
            *p_err = OS_ERR_MSG_POOL_EMPTY; //错误类型为“消息池没有消息”
        }
```

(下页继续)
return;             //返回，停止执行
}

/* 从消息池获取一个消息（暂存于 p_msg ） */
p_msg = OMSgpool.NextPtr; (1) //将消息控制块从消息池移除
OMSgpool.NextPtr = p_msg->NextPtr; (2) //指向下一个消息（取走首个消息）
OMSgpool.NbrFree--;  (3) //消息池可用消息数减 1
OMSgpool.NbrUsed++;  (4) //消息池被用消息数加 1
if (OMSgpool.NbrUsedMax < OMSgpool.NbrUsed) (5) //更新消息被用最大数目的历史记录
{
    OMSgpool.NbrUsedMax = OMSgpool.NbrUsed;
}

/* 将获取的消息插入消息队列 */
if (p_msg_q->NbrEntries == (OS_MSG_QTY)0) (6) //如果消息队列目前没有消息
{
    p_msg_q->InPtr          = p_msg;      //将其入队指针指向该消息
    p_msg_q->OutPtr         = p_msg;      //出队指针也指向该消息
    p_msg_q->NbrEntries     = (OS_MSG_QTY)1; //队列的消息数为 1
    p_msg->NextPtr          = (OS_MSG *)0; //该消息的下一个消息为空
}
else(7) //如果消息队列目前已有消息
{
    if ((opt & OS_OPT_POST_LIFO) == OS_OPT_POST_FIFO) //如果用 FIFO 方式插入队列，
    {
        p_msg_in          = p_msg_q->InPtr; //将消息插入入队端，入队
        p_msg_in->NextPtr  = p_msg;       //指针指向该消息。
        p_msg_q->InPtr     = p_msg;
        p_msg_q->NextPtr   = (OS_MSG *)0;
    }
    else(8) //如果用 LIFO 方式插入队列，
    {
        //后续代码
```c
p_msg->NextPtr = p_msg_q->OutPtr;  // 将消息插入出队端，出队
p_msg->OutPtr = p_msg;           // 指针指向该消息。
}
p_msg_q->NbrEntries++;          // 消息队列的消息数目加 1

if (p_msg_q->NbrEntriesMax < p_msg_q->NbrEntries) (10) // 更新消息队列的最大消息
{
    p_msg_q->NbrEntriesMax = p_msg_q->NbrEntries;       // 数目的历史记录。
}
p_msg->MsgPtr = p_void;          // 给该消息填写消息内容
p_msg->MsgSize = msg_size;       // 给该消息填写消息大小
p_msg->MsgTS = ts;               // 填写发布该消息时的时间戳
*p_err = OS_ERR_NONE;            // 错误类型为 “无错误”
```
• 代码清单: 消息队列-13 (7): 如果消息队列目前已有消息，那么又分两种入队的选项，是先进先出排队呢还是后进先出排队呢？如果采用 FIFO 方式插入队列，那么就将消息插入入队端，消息队列的最后一个消息的 NextPtr 指针就指向该消息，然后入队的消息成为队列中排队的最后一个消息，那么需要更新它的下一个消息为空。

• 代码清单: 消息队列-13 (8): 而如果采用 LIFO 方式插入队列，将消息插入出队端，队列中出队指针 OutPtr 指向该消息，需要出队的时候就是该消息首先出队，这就是后进先出原则。

• 代码清单: 消息队列-13 (9): 无论是采用哪种方式入队，消息队列的消息数目都要加 1。

• 代码清单: 消息队列-13 (10): 更新改消息队列的最大消息。

• 代码清单: 消息队列-13 (11): 既然消息已经入队了，那肯定得添加我们自己的消息内容啊，需要给该消息填写消息内容，消息中的 MsgPtr 指针指向我们的消息内容。

• 代码清单: 消息队列-13 (12): 给该消息填写我们发送的消息大小。

• 代码清单: 消息队列-13 (13): 填写发布该消息时的时间戳。

• 代码清单: 消息队列-13 (14): 当程序执行到这里，表面就是没有错误，返回错误类型为“无错误”的错误代码。

• 代码清单: 消息队列-12 (5): 如果有任务在等待消息，会有两种情况，一种是将消息发送到所有等待任务（广播消息），另一种是只将消息发送到等待任务中最高优先级的任务。根据 opt 选项选择其中一种方式进行发送消息，如果要把消息发送给所有等待任务，那就首先获取到等待任务个数，保存在要处理任务个数 cnt 变量中。

• 代码清单: 消息队列-12 (6): 否则就是把消息发布给一个等待任务，要处理任务个数 cnt 变量为 1。

• 代码清单: 消息队列-12 (7): 获取等待列表中的第一个任务。

• 代码清单: 消息队列-12 (8): 根据要处理任务个数 cnt 逐个将消息发送出去。

• 代码清单: 消息队列-12 (9): 获取任务的控制块。

• 代码清单: 消息队列-12 (10): 调用 OS_Post() 函数把消息发送给任务，其源码具体见代码清单: 消息队列-14。


- **代码清单: 消息队列-12 (11):** 每处理完一个任务, cnt 变量就要减一, 等到为 0 的时候退出 while 循环。

- **代码清单: 消息队列-12 (12):** 如果没选择 “发送完不调度任务”, 在发送消息完成的时候就要进行一次任务调度。

### 列表 14: 代码清单: 消息队列-14OS_Post() 源码

```c
void OS_Post (OS_PEND_OBJ *p_obj, (1) //内核对象类型指针
  OS_TCB *p_tcb, (2) //任务控制块
void *p_void, (3) //消息
OS_MSG_SIZE msg_size, (4) //消息大小
CPU_TS ts) (5) //时间戳
{
  switch (p_tcb->TaskState) (6) //根据任务状态分类处理
  {
    case OS_TASK_STATE_RDY: //如果任务处于就绪状态
    case OS_TASK_STATE_DLY: //如果任务处于延时状态
    case OS_TASK_STATE_SUSPENDED: //如果任务处于挂起状态
    case OS_TASK_STATE_DLY_SUSPENDED: //如果任务处于延时中被挂起状态
      break; (7) //不用处理, 直接跳出
    case OS_TASK_STATE_PEND: //如果任务处于无期限等待状态
    case OS_TASK_STATE_PEND_TIMEOUT: //如果任务处于有限期等待状态
      if (p_tcb->PendOn == OS_TASK_PEND_ON_MULTI) (8)
        //如果任务在等待多个信号量或消息队列
        {
          OS_Post1(p_obj, //标记哪个内核对象被发布
            p_tcb,
            p_void,
            msg_size,
            ts); (9)
        }
  }
}
```

(下页继续)
else (10)
    //如果任务不是在等待多个信号量或消息队列
{
    #if (OS_MSG_EN > 0u)
        //如果启用了任务队列或消息队列
        p_tcb->MsgPtr = p_void; (11) //保存消息到等待任务
        p_tcb->MsgSize = msg_size;
    #endif
    p_tcb->TS = ts; (12) //保存时间戳到等待任务
}
if (p_obj != (OS_PEND_OBJ *)) 0) //如果内核对象不为空
{
    OS_PendListRemove(p_tcb); (13) //从等待列表移除该等待任务
    #if OS_CFG_DBG_EN > 0u //如果启用了调试代码和变量
        OS_PendDbgNameRemove(p_obj, (14) //移除内核对象的调试名
            p_tcb);
    #endif
}
OS_TaskRdy(p_tcb); (14) //让该等待任务准备运行
p_tcb->TaskState = OS_TASK_STATE_RDY; (15) //任务状态改为就绪状态
p_tcb->PendStatus = OS_STATUS_PEND_OK; (16) //清除等待状态
p_tcb->PendOn = OS_TASK_PEND_ON NOTHING; (17) //标记不再等待
break;

case OS_TASK_STATE_PEND_SUSPENDED:
    //如果任务在无期限等待中被挂起
    case OS_TASK_STATE_PEND_TIMEOUT_SUSPENDED:
        //如果任务在有限期等待中被挂起
        if (p_tcb->PendOn == OS_TASK_PEND_ON_MULTI) (18)
            //如果任务在等待多个信号量或消息队列
            
            OS_Post1(p_obj, //标记哪个内核对象被发布
            //


79  else (20)
80     //如果任务不在等待多个信号量或消息队列
81     {
82         #if (OS_MSG_EN > 0u) //如果启用了调试代码和变量
83             p_tcb->MsgPtr = p_void; (21) //保存消息到等待任务
84             p_tcb->MsgSize = msg_size;
85         #endif
86         p_tcb->TS = ts; //保存时间戳到等待任务
87     }
88     OS_TickListRemove(p_tcb); (22) //从节拍列表移除该等待任务
89     if (p_obj != (OS_PEND_OBJ *)0) //如果内核对象为空
90         {
91             OS_PendListRemove(p_tcb); (23) //从等待列表移除该等待任务
92             #if OS_CFG_DBG_EN > 0u //如果启用了调试代码和变量
93                 OS_PendDbgNameRemove(p_obj,       //移除内核对象的调试名
94                     p_tcb);
95             #endif
96         }
97     p_tcb->TaskState = OS_TASK_STATE_SUSPENDED; (24) //任务状态改为被挂起状态
98     p_tcb->PendStatus = OS_STATUS_PEND_OK; (25) //清除等待状态
99     p_tcb->PendOn = OS_TASK_PEND_ON_NOTHING; (26) //标记不再等待
100    break;
101  default: (27) //如果任务状态超出预期
102    break;
103    }

论坛: https://www.firebbs.cn/                天猫: https://yehuosm.tmall.com
• 代码清单: 消息队列-14 (1)：内核对象类型指针，表示是哪个内核对象进行发布（释放/发送）操作。

• 代码清单: 消息队列-14 (2)：任务控制块指针，指向被操作的任务。

• 代码清单: 消息队列-14 (3)：消息指针。

• 代码清单: 消息队列-14 (4)：消息大小。

• 代码清单: 消息队列-14 (5)：时间戳。

• 代码清单: 消息队列-14 (6)：根据任务状态分类处理。

• 代码清单: 消息队列-14 (7)：如果任务处于就绪状态、延时状态、挂起状态或者是延时中被挂起状态，都不用处理，直接退出，因为现在这个操作是内核对象进行发布（释放）操作，而这些状态的任务是与内核对象无关的状态。

   也就是这些任务没在等待相关的内核对象（如消息队列、信号量等）。

• 代码清单: 消息队列-14 (8)：如果任务处于无限期等待状态或者是有限期等待状态，那么就需要处理了，先看看任务是不是在等待多个内核对象。

• 代码清单: 消息队列-14 (9)：如果任务在等待多个信号量或消息队列，就调用 OS_Post() 函数标记一下是哪个内核对象进行发布（释放）操作。

• 代码清单: 消息队列-14 (10)：如果任务不是在等待多个信号量或消息队列，就直接操作即可。

• 代码清单: 消息队列-14 (11)：如果启用了任务队列或消息队列（启用了 OS_MSG_EN 宏定义），保存消息到等待任务控制块的 MsgPtr 成员变量中，将消息的大小保存在等待任务控制块的 MsgSize 成员变量中。

• 代码清单: 消息队列-14 (12)：保存时间戳到等待任务控制块的 TS 成员变量中。

• 代码清单: 消息队列-14 (13)：如果内核对象不为空，调用 OS_PendListRemove() 函数从等待列表移除该等待任务。

论坛: https://www.firebbs.cn/  天猫: https://yehuosm.tmall.com
• 代码清单: 消息队列-14 (14): 调用 OS_TaskRdy() 函数让该等待任务准备运行。
• 代码清单: 消息队列-14 (15): 任务状态改为就绪状态。
• 代码清单: 消息队列-14 (16): 清除任务的等待状态。
• 代码清单: 消息队列-14 (17): 标记任务不再等待。
• 代码清单: 消息队列-14 (18): 如果任务在无期限等待中被挂起，或者任务在有限期等待中被挂起，反正任务就是在等待中被挂起了，也能进行内核对象发布（释放）操作，同理，先看看任务是不是在等待多个内核对象。
• 代码清单: 消息队列-14 (19): 如果任务在等待多个信号量或消息队列，就调用 OS_Post1() 函数标记一下是哪个内核对象进行发布（释放）操作。
• 代码清单: 消息队列-14 (20): 如果任务不在等待多个信号量或消息队列，就直接操作即可。
• 代码清单: 消息队列-14 (21): 如果启用了任务队列或消息队列（启用了 OS_MSG_EN 宏定义），保存消息到等待任务控制块的 MsgPtr 成员变量中，将消息的大小保存到等待任务控制块的 MsgSize 成员变量中。
• 代码清单: 消息队列-14 (22): 调用 OS_TickListRemove() 函数将任务从节拍列表中移除。
• 代码清单: 消息队列-14 (23): 从等待列表移除该等待任务。
• 代码清单: 消息队列-14 (24): 任务状态改为被挂起状态。
• 代码清单: 消息队列-14 (25): 清除任务的等待状态。
• 代码清单: 消息队列-14 (26): 标记任务不再等待。
• 代码清单: 消息队列-14 (27): 如果任务状态超出预期，直接跳出。

从消息队列的入队操作（发送消息）我们可以看出：μC/OS 支持向所有任务发送消息，也支持只向一个任务发送消息，这样子系统的灵活性就会大大提高，与此同时，μC/OS 还支持中断延迟发布，不在中断中直接发送消息。

消息队列的发送函数 OSQPost() 使用实例具体见代码清单: 消息队列-15.
22.6.4 消息队列获取函数 OSQPend()

当任务试图从队列中的获取消息时，用户可以指定一个阻塞超时时间，当且仅当消息队列中有消息的时候，任务才能获取到消息。在这段时间中，如果队列为空，该任务将保持阻塞状态以等待队列消息有效。当其他任务或中断服务程序往其等待的队列中写入了数据，该任务将自动由阻塞状态转为就绪态。当任务等待的时间超过了用户指定的阻塞时间，即使队列中尚无有效消息，任务也会自动从阻塞状态转为就绪态。OSQPend() 函数源码具体见代码清单: 消息队列-16。
// 定义一个局部变量，用于保存中断前的 CPU 状态寄存器
// SR (临界段口的中断需保存 SR)，开中断时将该值还原。

#ifndef OS_SAFETY_CRITICAL(7) // 如果启用（默认禁用）了安全检测
    if (p_err == (OS_ERR *)0) // 如果错误类型实参为空
        {
            OS_SAFETY_CRITICAL_EXCEPTION(); // 执行安全检测异常函数
            return ((void *)0); // 返回 0 (有错误)，停止执行
        }
#endif

#ifndef OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u(8) // 如果启用了中断中非法调用检测
    if (OSIntNestingCtr > (OS_NESTING_CTR)0) // 如果该函数在中断中被调用
        {
            *p_err = OS_ERR_PEND_ISR; // 错误类型为“在中断中中止等待”
            return ((void *)0); // 返回 0 (有错误)，停止执行
        }
#endif

#ifndef OS_CFG_ARG_CHK_EN > 0u(9) // 如果启用了参数检测
    if (p_q == (OS_Q *)0) // 如果 p_q 为空
        {
            *p_err = OS_ERR_OBJ_PTR_NULL; // 错误类型为“对象为空”
            return ((void *)0); // 返回 0 (有错误)，停止执行
        }
    if (p_msg_size == (OS_MSG_SIZE *)0) // 如果 p_msg_size 为空
        {
            *p_err = OS_ERR_PTR_INVALID; // 错误类型为“指针不可用”
            return ((void *)0); // 返回 0 (有错误)，停止执行
        }
    switch (opt) (10) // 根据选项分类处理
    {

case OS_OPT_PEND_BLOCKING: //如果选项在预期内
    break; //直接跳出

default: //如果选项超出预期
    *p_err = OS_ERR_OPT_INVALID; //返回错误类型为“选项非法”
    return ((void *)0); //返回0(有错误)，停止执行
#endif

#if OS_CFG_OBJ_TYPE_CHK_EN > 0u //如果启用了对象类型检测
    if (p_q->Type != OS_OBJ_TYPE_Q) //如果p_q不是消息队列类型
    {
        *p_err = OS_ERR_OBJ_TYPE; //错误类型为“对象类型有误”
        return ((void *)0); //返回0(有错误)，停止执行
    }
#endif

if (p_ts != (CPU_TS *)0) //如果p_ts非空
{
    *p_ts = (CPU_TS *)0; //初始化(清零)p_ts，待用于返回时间戳
}

CPU_CRITICAL_ENTER(); //关中断
p_void = OS_MsgQGet(&p_q->MsgQ, //从消息队列获取一个消息
             p_msg_size,
             p_ts,
             p_err);
if (*p_err == OS_ERR_NONE) //如果获取消息成功
{
    CPU_CRITICAL_EXIT(); //开中断
    return (p_void); //返回消息内容
/* 如果获取消息不成功 */
if ((opt & OS_OPT_PEND_NON_BLOCKING) != (OS_OPT)0) //如果选择了不阻塞任务
{
    CPU_CRITICAL_EXIT(); //开中断
    *p_err = OS_ERR_PEND_WOULD_BLOCK; //错误类型为“等待渴求阻塞”
    return ((void *)0); //返回 0 (有错误), 停止执行
} //如果选择了阻塞任务

else //如果调度器被锁
{
    if (OSSchedLockNestingCtr > (OS_NESTING_CTR)0) //如果调度器被锁
    {
        CPU_CRITICAL_EXIT(); //开中断
        *p_err = OS_ERR_SCHED_LOCKED; //错误类型为“调度器被锁”
        return ((void *)0); //返回 0 (有错误), 停止执行
    } //如果调度器未被锁

    /* 如果调度器未被锁 */
    OS_CRITICAL_ENTER_CPU_EXIT(); //锁调度器, 重开中断
    OS_Pend(&pend_data,
    //阻塞当前任务, 等待消息队列,
    (OS_PEND_OBJ *)((void *)p_q), //将当前任务脱离就绪列表, 并
    OS_TASK_PEND_ON_Q, //插入就绪列表和等待列表。
    timeout);
    OS_CRITICAL_EXIT_NO_SCHED(); //开调度器, 但不进行调度
    OSSched(); //找到并调度最高优先级就绪任务
    /* 当前任务（获得消息队列的消息）得以继续运行 */
    CPU_CRITICAL_ENTER(); //关中断
    switch (OSTCBCurPtr->PendStatus) //根据当前运行任务的等待状态分类处理
    {...}
{ 
  case OS_STATUS_PEND_OK: (24) // 如果等待状态正常
      p_void     = OSTCBCurPtr->MsgPtr; (25)
      // 从 (发布时放于) 任务控制块提取消息
      *p_msg_size = OSTCBCurPtr->MsgSize; // 提取消息大小
      if (p_ts != (CPU_TS *)0) // 如果 p_ts 非空
      {
        *p_ts     = OSTCBCurPtr->TS; // 获取任务等到消息时的时间戳
      }
      *p_err    = OS_ERR_NONE; // 错误类型为 "无错误"
      break; // 跳出
  
  case OS_STATUS_PEND_ABORT: (26) // 如果等待被中止
      p_void     = (void *)0; // 返回消息内容为空
      *p_msg_size = (OS_MSG_SIZE)0; // 返回消息大小为 0
      if (p_ts != (CPU_TS *)0) // 如果 p_ts 非空
      {
        *p_ts     = OSTCBCurPtr->TS; // 获取等待被中止时的时间戳
      }
      *p_err    = OS_ERR_PEND_ABORT; // 错误类型为 "等待被中止"
      break; // 跳出
  
  case OS_STATUS_PEND_TIMEOUT: (27) // 如果等待超时
      p_void     = (void *)0; // 返回消息内容为空
      *p_msg_size = (OS_MSG_SIZE)0; // 返回消息大小为 0
      if (p_ts != (CPU_TS *)0) // 如果 p_ts 非空
      {
        *p_ts     = (CPU_TS *)0; // 清零 p_ts
      }
      *p_err    = OS_ERR_TIMEOUT; // 错误类型为 "等待超时"
      break; // 跳出
}
```c
 case OS_STATUS_PEND_DEL:  //如果等待的内核对象被删除
   p_void = (void *)0;       //返回消息内容为空
   *p_msg_size = (OS_MSG_SIZE)0; //返回消息大小为 0
   if (p_ts != (CPU_TS *)0)  //如果 p_ts 非空
   {
     *p_ts = OSTCBCurPtr->TS;  //获取对象被删时的时间戳
   }
   *p_err = OS_ERR_OBJ_DEL;    //错误类型为 “等待对象被删”
   break;                     //跳出

 default:                    //如果等待状态超出预期
   p_void = (void *)0;        //返回消息内容为空
   *p_msg_size = (OS_MSG_SIZE)0; //返回消息大小为 0
   *p_err = OS_ERR_STATUS_INVALID; //错误类型为 “状态非法”
   break;                     //跳出
   CPU_CRITICAL_EXIT(); //开中断
 return (p_void);
```

- 代码清单: 消息队列-16 (1): 消息队列指针，指向要获取消息的队列。
- 代码清单: 消息队列-16 (2): 指定阻塞时间（单位: 时钟节拍）。
- 代码清单: 消息队列-16 (3): 获取消息的选项，在 os.h 中有定义。
- 代码清单: 消息队列-16 (4): 用于保存返回获取的消息大小（单位：字节）。
- 代码清单: 消息队列-16 (5): 用于保存返回等到消息时的时间戳。
- 代码清单: 消息队列-16 (6): 用于保存返回的错误类型，用户可以根据此变量得知错误的原因。
- 代码清单: 消息队列-16 (7): 如果启用（默认禁用）了安全检测，在编译时会包含安全检测相关的代码，如果错误类型实参为空，系统会执行安全检测异常函数，然后返回，停止
执行。

- **代码清单: 消息队列-16 (8):** 如果启用了中断中非法调用检测，并且如果该函数在中断中被调用，则返回错误类型为“在中断获取消息”的错误代码，然后退出，停止执行。

- **代码清单: 消息队列-16 (9):** 如果启用了参数检测，在编译时则会包含参数检测相关的代码，如果 p_q 参数为空，返回错误类型为“内核对象为空”的错误代码，并且退出，不执行获取消息操作。

- **代码清单: 消息队列-16 (10):** 根据 opt 选项进行分类处理，如果选项在预期内，直接退出，其实在这里只是对选项的一个检查，看看传入的选项参数是否正确。

- **代码清单: 消息队列-16 (11):** 如果 opt 选项超出预期，返回错误类型为“选项非法”的错误代码，并且退出，不执行获取消息操作。

- **代码清单: 消息队列-16 (12):** 如果启用了对象类型检测，在编译时则会包含对象类型检测相关代码，如果 p_q 不是消息队列类型，那么返回错误类型为“对象类型有误”的错误代码，并且退出，不执行获取消息操作。

- **代码清单: 消息队列-16 (13):** 如果 p_ts 非空，就初始化（清零）p_ts，待用于返回时间戳。

- **代码清单: 消息队列-16 (14):** 调用 OS_MsgQGet() 函数从消息队列获取一个消息，其源码具体见代码清单: 消息队列-17。

### 列表 17: 代码清单: 消息队列-17OS_MsgQGet() 源码

```c
void *OS_MsgQGet (OS_MSG_Q *p_msg_q,  //消息队列
    OS_MSG_SIZE *p_msg_size,  //返回消息大小
    CPU_TS *p_ts,  //返回某些操作的时间戳
    OS_ERR *p_err)  //返回错误类型
{
    OS_MSG *p_msg;
    void *p_void;
```

(下页继续)
```c
#define OS_SAFETY_CRITICAL //#如果启用(默认禁用)了安全检测

if (p_err == (OS_ERR *)0) //如果错误类型实参为空
{
    OS_SAFETY_CRITICAL_EXCEPTION(); //执行安全检测异常函数
    return ((void *)0); //返回空消息，停止执行
}
#endif

if (p_msg_q->NbrEntries == (OS_MSG_QTY)0) (1)//如果消息队列没有消息
{
    *p_msg_size = (OS_MSG_SIZE)0; //返回消息长度为 0
    if (p_ts != (CPU_TS *)0) //如果 p_ts 非空
    {
        *p_ts = (CPU_TS )0; //清零 p_ts
    }
    *p_err = OS_ERR_Q_EMPTY; //错误类型为“队列没消息”
    return ((void *)0); //返回空消息，停止执行
}
/* 如果消息队列有消息 */
p_msg = p_msg_q->OutPtr; (2)//从队列的出口端提取消息
p_void = p_msg->MsgPtr; (3)//提取消息内容
*p_msg_size = p_msg->MsgSize; (4)//提取消息长度
if (p_ts != (CPU_TS *)0) (5)//如果 p_ts 非空
{
    *p_ts = p_msg->MsgTS; //获取消息被发布时的时间戳
}

p_msg_q->OutPtr = p_msg->NextPtr; (6)//修改队列的出队指针
if (p_msg_q->OutPtr == (OS_MSG *)0) (7)//如果队列没有消息了
{
    p_msg_q->InPtr = (OS_MSG *)0; //清零出队指针
}
```

(下页继续)


```c
p_msg_q->NbrEntries = (OS_MSG_QTY) 0; //清零消息数
}
else //如果队列还有消息
{
    p_msg_q->NbrEntries--; //队列的消息数减 1
}
/* 从消息队列提取完消息信息后，将消息释放回消息池供继续使用 */
p_msg->NextPtr = OMsgPool.NextPtr; //消息插回消息池
OSMsgPool.NextPtr = p_msg;
OSMsgPool.NbrFree++; //消息池的可用消息数加 1
OSMsgPool.NbrUsed--; //消息池的已用消息数减 1
*p_err = OS_ERR_NONE; //错误类型为“无错误”
return (p_void); //返回消息内容
```

- **代码清单: 消息队列-17 (1)**：如果消息队列目前没有可用消息，返回消息长度为 0，并且返回错误类型为“队列没消息”的错误代码和空消息，停止执行。
- **代码清单: 消息队列-17 (2)**：而如果队列中有消息，则从队列的出口端提取消息。
- **代码清单: 消息队列-17 (3)**：提取消息内容。
- **代码清单: 消息队列-17 (4)**：提取消息长度。
- **代码清单: 消息队列-17 (5)**：如果 p_ts 非空，获取消息入队时的时间戳。
- **代码清单: 消息队列-17 (6)**：修改队列的出队指针。
- **代码清单: 消息队列-17 (7)**：如果队列没有消息了，就将出队指针与消息个数清零。
- **代码清单: 消息队列-17 (8)**：如果队列还有消息，队列的消息个数减 1。
- **代码清单: 消息队列-17 (9)**：消息插回消息池，以便重复利用。
- **代码清单: 消息队列-17 (10)**：消息池的可用消息数加 1。
- **代码清单: 消息队列-17 (11)**：消息池的已用消息数减 1。
• 代码清单: 消息队列-17 (12): 返回消息内容。

• 代码清单: 消息队列-16 (15): 如果获取消息成功，则返回消息的内容。

• 代码清单: 消息队列-16 (16): 如果获取消息不成功，并且用户选择了阻塞等待，则返回错误类型为“等待渴求阻塞 (OS_ERR_PEND_WOULD_BLOCK)”的错误代码，并且返回 0，表示没有获取到消息。

• 代码清单: 消息队列-16 (17): 当获取消息不成功的时候，用户选择了阻塞等待，那么就会将任务状态变为阻塞态以等待消息。

• 代码清单: 消息队列-16 (18): 判断一下调度器是否被锁，如果被锁了，则返回错误类型为“调度器被锁”的错误代码，然后退出。

• 代码清单: 消息队列-16 (19): 如果调度器未被锁，就锁定调度器，重新打开中断。此次可能有同学就会问了，为什么刚刚调度器被锁就错误的呢，而现在又要锁定调度？那是因为之前锁定的调度器不是由这个函数进行锁定的，这是不允许的，因为现在要阻塞当前任务，而调度器锁定了就表示无法进行任务调度，这也是不允许的。那为什么又要关闭调度器呢，因为接下来的操作是需要操作队列与任务的列表，这个时间就不会很短，系统不可能把其他任务来操作任务列表，因为可能引起其他任务解除阻塞，这可能会发生优先级翻转。比如任务 A 的优先级低于当前任务，但是在当前任务进入阻塞的过程中，任务 A 却因为其他原因解除阻塞了，那系统肯定是会去运行任务 A，这显然是要绝对禁止的，因为挂起调度器意味着任务不能切换并且不准调用可能引起任务切换的 API 函数，所以，锁定调度器，打开中断这样的处理，既不会影响中断的响应，又避免了其他任务来操作队列与任务的列表。

• 代码清单: 消息队列-16 (20): 调用 OS_Pend() 函数将当前任务脱离就绪列表，并根据用户指定的阻塞时间插入就绪列表和队列等待列表，然后打开调度器，但不进行调度，OS_Pend() 源码具体见代码清单: 消息队列-18。

注：此源码的注释很丰富，就不再讲解源码了。

列表 18: 代码清单: 消息队列-18OS_Pend() 源码

```c
void OS_Pend (OS_PEND_DATA *p_pend_data, //待插入等待列表的元素
              OS_PEND_OBJ *p_obj,       //等待的内核对象
```

(下页继续)
OS_STATE pending_on,  //等待哪种对象内核
OS_TICK timeout)   //等待期限
{
    OS_PEND_LIST *p_pend_list;
    OSTCBCurPtr->PendOn = pending_on;  //资源不可用，开启等待
    OSTCBCurPtr->PendStatus = OS_STATUS_PEND_OK;  //正常等待中
    OS_TaskBlock(OSTCBCurPtr,timeout);  //阻塞当前运行任务，如果 timeout 非
    if (p_obj != (OS_PEND_OBJ *)0)   //如果等待对象非空
    {
        p_pend_list = &p_obj->PendList;  //获取对象的等待列表到
    }
    p_pend_data->PendObjPtr = p_obj;  //保存要等待的对象
    OS_PendDataInit((OS_TCB *)OSTCBCurPtr,  //初始化 p_pend_
        (OS_PEND_DATA *)p_pend_data,
        (OS_OBJ_QTY )1);
    OS_PendListInsertPrio(p_pend_list,
        p_pend_data);
}
else //如果等待对象为空
{
    OSTCBCurPtr->PendDataTableEntries = (OS_OBJ_QTY )0;  //清零当前任务的
    OSTCBCurPtr->PendDataTablePtr = (OS_PEND_DATA *)0;
}
#if OS_CFG_DBG_EN > 0u //如果启用了调试代码和变量
    OS_PendDbgNameAdd(p_obj,    //更新信号量的 DbgNamePtr 元素为其等待
        OSTCBCurPtr);  //列表中优先级最高的任务的名称。
#endif

论坛：https://www.firebbs.cn/ 天猫：https://yehuosm.tmall.com
• 代码清单: 消息队列-16 (21): 在这里就进行一次任务的调度。

• 代码清单: 消息队列-16 (22): 程序能执行到这里，就说明大体上有两种情况，要么是消息队列中有消息入队，任务获取到消息了；任务还没获取到消息（任务没获取到消息的情况有很多种），无论是哪种情况，都先把中断关掉再说。

• 代码清单: 消息队列-16 (23): 根据当前运行任务的等待状态分类处理。

• 代码清单: 消息队列-16 (24): 如果任务状态是 OS_STATUS_PEND_OK，则表示任务获取到消息了。

• 代码清单: 消息队列-16 (25): 从任务控制块中提取消息，这是因为在发送消息给任务的时候，会将消息放入任务控制块的 MsgPtr 成员变量中，然后继续提取消息大小，如果 p_ts 非空，记录获取任务等到消息时的时间戳，返回错误类型为“无错误”的错误代码，跳出 switch 语句。

• 代码清单: 消息队列-16 (26): 如果任务在等待（阻塞）被中止，则返回消息内容为空，返回消息大小为 0，如果 p_ts 非空，获取等待被中止时的时间戳，返回错误类型为“等待被中止”的错误代码，跳出 switch 语句。

• 代码清单: 消息队列-16 (27): 如果等待（阻塞）超时，说明等待的时间过去了，任务也没获取到消息，则返回消息内容为空，返回消息大小为 0，如果 p_ts 非空，将 p_ts 清零，返回错误类型为“等待超时”的错误代码，跳出 switch 语句。

• 代码清单: 消息队列-16 (28): 如果等待的内核对象被删除，则返回消息内容为空，返回消息大小为 0，如果 p_ts 非空，获取对象被删时的时间戳，返回错误类型为“等待对象被删”的错误代码，跳出 switch 语句。

• 代码清单: 消息队列-16 (29): 如果等待状态超出预期，则返回消息内容为空，返回消息大小为 0，返回错误类型为“状态非法”的错误代码，跳出 switch 语句。

• 代码清单: 消息队列-16 (30): 打开中断，返回消息内容。

消息队列获取函数的使用实例具体见代码清单: 消息队列-19。

论坛: [https://www.firebbs.cn/](https://www.firebbs.cn/)  425  天猫: [https://yehuosm.tmall.com](https://yehuosm.tmall.com)
列表 19: 代码清单: 消息队列-19 OSQPend() 使用实例

```c
OS_Q queue;                        //声明消息队列
OS_ERR err;
OS_MSG_SIZE msg_size;

/* 获取消息队列 queue 的消息 */
pMsg = OSQPend ((OS_Q*)&queue,        //消息变量指针
   (OS_TICK) 0,                        //等待时长为无限
   (OS_OPT)OS_OPT_PEND_BLOCKING,      //如果没有获取到信号量就等待
   (OS_MSG_SIZE*)&msg_size,           //获取消息的字节大小
   (CPU_TS*)0,                        //获取任务发送时的时间戳
   (OS_ERR*)&err);                    //返回错误
```

### 22.7 消息队列使用注意事项

在使用μC/OS 提供的消息队列函数的时候，需要了解以下几点:

1. 使用 OSQPend()、OSQPost() 等这些函数之前应先创建需消息队列，并根据队列句柄（队列控制块）进行操作。

2. 队列读取采用的是先进先出（FIFO）模式，会先读取先存储在队列中的数据。当然也μC/OS 也支持后进先出（LIFO）模式，那么读取的时候就会读取到后进队列的数据。

3. 无论是发送或者是接收消息都是以数据引用的方式进行。

4. 队列是具有自己独立权限的内核对象，并不属于任何任务。所有任务都可以向同一队列写入和读出。一个队列由多任务或中断写入是经常的事，但由多个任务读出倒是用的比较少。

5. 消息的传递实际上只是传递传送内容的指针和传送内容的字节大小。这在使用消
息队列的时候就要注意了，获取消息之前不能释放存储在消息中的指针内容，比如中断定义了一个局部变量，然后将该地址放在消息中进行传递，中断退出之前消息并没有被其他任务获取，退出中断的时候 CPU 已经释放了中断中的这个局部变量，后面任务获取这个地址的内容就会出错。所以一定要保证在获取内容地址之前不能释放内容这个内存单元。有三种方式可以避免这种情况：

- 将变量定义为静态变量，即在其前面加上 static，这样内存单元就不会被释放。
- 将变量定义为全局变量。
- 将要传递的内容当做指针传递过去。比如地址 0x12345678 存放一个变量的值为 5，常规是把 0x12345678 这个地址传递给接收消息的任务，任务接收到这个消息后，取出这个地址的内容 5。但是如果我们将 5 拼成“地址”传递给任务，最后接收消息的任务直接拿着这个“地址”当做内容去处理即可。不过这种方法不能传递结构体等比较复杂的数据结构，因为消息中存放地址的变量内存大小是有限的（一个指针大小）。

### 22.8 消息队列实验

消息队列实验是在 µC/OS 中创建了两个任务 AppTaskPost() 和 AppTaskPend()，任务 AppTaskPost() 用于发送消息，任务 AppTaskPend() 用于接收消息，两个任务独立运行，并把接收到的消息通过串口调试助手打印出来。具体见 代码清单: 消息队列-20。

#### 列表 20: 代码清单: 消息队列-20 消息队列实验

```c
#include <includes.h>

/**************************************************************************
LOCAL DEFINES
**************************************************************************

OS_Q queue;           //声明消息队列

/**************************************************************************
```

(下页继续)
static OS_TCB AppTaskStartTCB; //任务控制块
static OS_TCB AppTaskPostTCB;
static OS_TCB AppTaskPendTCB;

static CPU_STK AppTaskStartStk[APP_TASK_START_STK_SIZE]; //任务栈
static CPU_STK AppTaskPostStk[APP_TASK_POST_STK_SIZE];
static CPU_STK AppTaskPendStk[APP_TASK_PEND_STK_SIZE];

static void AppTaskStart (void *p_arg);
static void AppTaskPost (void *p_arg);
static void AppTaskPend (void *p_arg);

/*
 */

main()
*
* Description : This is the standard entry point for C code. It is assumed that
* your code will call main() once you have performed all necessary
* initialization.
* Arguments : none
int main (void)
{
    OS_ERR err;
    OSInit(&err); // 初始化uC/OS-III

    /* 创建起始任务 */
    OSTaskCreate((OS_TCB *) &AppTaskStartTCB,
                 (CPU_CHAR *) "App Task Start",
                 (OS_TASK_PTR) AppTaskStart,
                 (void *) 0,
                 (OS_PRIO) APP_TASK_START_PRIO,
                 (CPU_STK *) &AppTaskStartStk[0],
                 (CPU_STK_SIZE) APP_TASK_START_STK_SIZE / 10,
                 (CPU_STK_SIZE) APP_TASK_START_STK_SIZE,
                 (OS_MSG_QTY) 5u,
                 (OS_TICK) 0u,
                 (void *) 0,
                 (OS_OPT) (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
                 (OS_ERR) &err); // 返回错误类型

    OSStart(&err);
}
// 启动多任务管理 ( 由 µC/OS-III 控制 )

}  

/***** STARTUP TASK *******/

/*
 * Description : This is an example of a startup task. As mentioned in
 * the book's text, you MUST initialize the ticker only once mu
 * 1ltitasking has started.
 * Arguments : p_arg is the argument passed to 'AppTaskStart()' by
 * 'OSTaskCreate()'.
 * Returns : none
 * Notes : 1) The first line of code is used to prevent a compiler
 * warning because 'p_arg' is not
 * used. The compiler should not generate any code for
 * this statement.
 */

static void AppTaskStart (void *p_arg)
{
    CPU_INT32U cpu_clk_freq;
    CPU_INT32U cnts;
    OS_ERR err;

    (void)p_arg;

    BSP_Init(); // 板级初始化
CPU_Init();
//初始化CPU组件（时间戳，中断时间测量和主机名）

cpu_clk_freq = BSP_CPU_ClkFreq();
//获取CPU内核时钟频率（SysTick工作时钟）
cnts = cpu_clk_freq / (CPU_INT32U)OSCfg_TickRate_Hz;
//根据用户设定的时钟节拍频率计算SysTick定时器的计数值
OS_CPU_SysTickInit(cnts);
//调用SysTick初始化函数，设置定时器计数值和启动定时器

Mem_Init();
//初始化内存管理组件（堆内存池和内存池表）

#ifdef OS_CFG_STAT_TASK_EN > 0u
//如果启用（默认启用）了统计任务
OSStatTaskCPUUsageInit(&err);
//计算没有应用任务（只有空闲任务）运行时CPU的（最大）
#endif //容量（决定OS_Stat_IdleCtrMax的值，为后面计算CPU使用率使用）。

CPU_IntDisMeasMaxCurReset();
//复位（清零）当前最大关中断时间

/* 创建消息队列 queue */
OSQCreate ((OS_Q   *)&queue,                   //指向消息队列的指针
    (CPU_CHAR   *)"Queue For Test",       //队列的名字
    (OS_MSG_QTY )20,                      //最多可存放消息的数目
    (OS_ERR     *)&err);                //返回错误类型

/* 创建AppTaskPost任务 */
OSTaskCreate((OS_TCB  *)&AppTaskPostTCB,    //任务控制块地址
(CPU_CHAR *) "App Task Post",       //任务名称
(OS_TASK_PTR ) AppTaskPost,          //任务函数
(void *) 0,                          //传递给任务函数（形参 p_arg）的实参
(OS_PRIO ) APP_TASK_POST_PRIO,       //任务的优先级
(CPU_STK *) &AppTaskPostStk[0],      //任务栈的基地址
(CPU_STK_SIZE) APP_TASK_POST_STK_SIZE / 10, //任务栈空间剩下 1/10 时限制其增长
(CPU_STK_SIZE) APP_TASK_POST_STK_SIZE, //任务栈空间（单位：sizeof(CPU_STK)）
(OS_MSG_QTY ) 5u,                     //任务可接收的最大消息数
(OS_TICK ) 0u,                         //任务的时间片节拍数（0 表默认值 OSCfg_TickRate_Hz/10）
(void *) 0,                            //任务扩展（0 表不扩展）
(OS_OPT ) (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
(OS_ERR *) &err);                     //返回错误类型

/* 创建 AppTaskPend 任务 */
OSTaskCreate((OS_TCB *) &AppTaskPendTCB,      //任务控制块地址
              (CPU_CHAR *) "App Task Pend",     //任务名称
              (OS_TASK_PTR ) AppTaskPend,       //任务函数
              (void *) 0,
              //传递给任务函数（形参 p_arg）的实参
              (OS_PRIO ) APP_TASK_PEND_PRIO,    //任务的优先级
              (CPU_STK *) &AppTaskPendStk[0],   //任务栈的基地址
              (CPU_STK_SIZE) APP_TASK_PEND_STK_SIZE / 10, //任务栈空间剩下 1/10 时限制其增长
              (CPU_STK_SIZE) APP_TASK_PEND_STK_SIZE, //任务栈空间（单位：sizeof(CPU_STK)）
              (OS_OPT ) (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
              (OS_ERR *) &err);
//任务栈空间（单位：sizeof(CPU_STK))
(OS_MSG_QTY) 5u,
//任务可接收的最大消息数
(OS_TICK) 0u,
//任务的时间片节拍数（0 表默认值 OSCfg_TickRate_Hz/10)
(void*) 0,
//任务扩展（0 表不扩展）
(OS_OPT)(OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
(OS_ERR*) &err);  //返回错误类型

OSTaskDel (&AppTaskStartTCB, &err);
//删除起始任务本身，该任务不再运行

OSTaskDel (&AppTaskStartTCB, &err);
//删除起始任务本身，该任务不再运行

} /* END AppTaskDel */

/***************************************************************************/
/* POST TASK */
/***************************************************************************/
static void AppTaskPost ( void * p_arg )
{
  OS_ERR err;
  (void)p_arg;

  while (DEF_TRUE)  //任务体
  {
    /* 发送消息到消息队列 queue */
    OSQPost ((OS_Q*)&queue,  //消息变量指针
              (void*)"Fire μC/OS-III",
              (OS_MSG_SIZE)sizeof("Fire μC/OS-III"), //数据字节大小
(OS_OPT | OS_OPT_POST_FIFO | OS_OPT_POST_ALL),
//先进先出和发布给全部任务的形式
(OS_ERR *)&err);  //返回错误类型

OSTimeDlyHMSM ( 0, 0, 0, 500, OS_OPT_TIME_DLY, & err );
}
}

/******************************************************************************
PEND TASK
******************************************************************************/
static void AppTaskPend ( void * p_arg )
{
  OS_ERR err;
  OS_MSG_SIZE msg_size;
  CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必须用到该宏，该宏声明和
  //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
  // SR (临界段关中断只需保存 SR), 开中断时将该值还原。
  char * pMsg;
  (void)p_arg;

  while (DEF_TRUE) //任务体
  {
    /* 请求消息队列 queue 的消息 */
    pMsg = OSQPend ((OS_Q *)&queue, //消息变量指针
      (OS_TICK )0,                  //等待时长为无限
      (OS_OPT | OS_OPT_PEND_BLOCKING),
      //如果没有获取到信号量就等待
      (OS_MSG_SIZE *)&msg_size,     //获取消息的字节大小
      (CPU_TS *)&CPU_TS,           //获取任务发送时的时间戳
      (OS_ERR *)&err);            //返回错误

22.9 消息队列实验现象

将程序编译好，用 USB 线连接计算机和开发板的 USB 接口（对应丝印为 USB 转串口），用 DAP 仿真器把配套程序下载到野火 STM32 开发板（具体型号根据购买的板子而定，每个型号的板子都配套有对应的程序），在计算机上打开串口调试助手，然后复位开发板就可以在调试助手中看到串口的打印信息，具体见图消息队列实验现象。
论坛：https://www.firebbs.cn/  天猫：https://yehuosm.tmall.com
第 23 章 信号量

同志们，回想一下，你是否在裸机编程中这样使用过一个变量：用于标记某个事件是否发生，或者标志一下某个东西是否正在被使用，如果是被占用了的或者没发生，我们就不对它进行操作。

23.1 信号量基本概念

信号量（Semaphore）是一种实现任务间通信的机制，可以实现任务之间同步或临界资源的互斥访问，常用于协助一组相互竞争的任务来访问临界资源。在多任务系统中，各任务之间需要同步或互斥实现临界资源的保护，信号量功能可以为用户提供这方面的支持。

抽象的来讲，信号量是一个非负整数，所有获取它的任务都会将该整数减一（获取它当然是为了使用资源），当该整数值为零时，所有试图获取它的任务都将处于阻塞状态。通常一个信号量的计数值用于对应有效的资源数，表示剩余可被占用的临界资源数，其值的含义分两种情况：

• 0：表示没有积累下来的释放信号量操作，且有可能有在此信号量上阻塞的任务。
• 正值：表示有一个或多个释放信号量操作。

注意：μC/OS 的信号量并没有区分二值信号量与计数信号量，下面是作者为了更详细解释信号量的相关内容，自行区分二值信号量与计数信号量，其实原理都是一样的，只不过用途不一样而已。

注意：μC/OS 中的信号量不具备传递数据的功能。

23.1.1 二值信号量

二值信号量既可以用于临界资源访问也可以用于同步功能。

二值信号量和互斥信号量（以下使用互斥量表示互斥信号量）非常相似，但是有一些细微差别：互斥量有优先级继承机制，二值信号量则没有这个机制。这使得二值信号量更偏向应用于同步功能（任务与任务间的同步或任务和中断间同步），而互斥量更偏向应用于临界资源的互斥访问。
用作同步时，信号量在创建后应被置为空，任务 1 获取信号量而进入阻塞，任务 2 在某种条件发生后，释放信号量，于是任务 1 获得信号量得以进入就绪态，如果任务 1 的优先级是最高的，那么就会立即切换任务，从而达到了两个任务间的同步。同样的，在中断服务函数中释放信号量，任务 1 也会得到信号量，从而达到任务与中断间的同步。

还记得我们经常说的中断要快进快出吗，在裸机开发中我们经常是在中断中做一个标记，然后在退出的时候进行轮询处理，这个就是类似我们使用信号量进行同步的，当标记发生了，我们再做其他事情。在 μC/OS 中我们用信号量用于同步，任务与任务的同步，中断与任务的同步，可以大大提高效率。

### 23.1.2 计数信号量

顾名思义，计数信号量肯定是用于计数的。在实际的使用中，我们常将计数信号量用于事件计数与资源管理。每当某个事件发生时，任务或者中断将释放一个信号量（信号量计数值加 1），当处理被事件时（一般在任务中处理），处理任务会取走该信号量（信号量计数值减 1），信号量的计数值则表示还有多少个事件未被处理。此外，系统还有很多资源，我们也可以使用计数信号量进行资源管理，信号量的计数值表示系统中可用的资源数目，任务必须先获取到信号量才能获取资源访问权，当信号量的计数值为零时表示系统没有可用的资源，但是要注意，在使用完资源的时候必须归还信号量，否则当计数值为 0 的时候任务就无法访问该资源了。

计数信号量允许多个任务对其进行操作，但限制了任务的数量。比如有一个停车场，里面只有 100 个车位，那么能停的车只有 100 辆，也相当于我们的信号量有 100 个，假如一开始停车场的车位还有 100 个，那么每进去一辆车就要消耗一个停车位，车位的数量就要减一，对应的，我们的信号量在使用之后也需要减一，当停车场停满了 100 辆车的时候，此时的停车位为 0，来的车就不能进去了，否则将造成事故，也相当于我们的信号量为 0，后面的任务对这个停车场资源的访问也无法进行，当有车从停车场离开的时候，车位又空余出来了，那么，后面的车就能进去了，我们信号量的操作也是一样的，当我们释放了这个资源，后面的任务才能对这个资源进行访问。
23.2 信号量应用场景

在嵌入式操作系统中二值信号量是任务间、任务与中断间同步的重要手段，信号量使用最多的一般都是二值信号量与互斥量（互斥量在下一章讲解）。为什么叫二值信号量呢？因为信号量资源被获取了，信号量值就是 0，信号量资源被释放，信号量值就是 1，把这种只有 0 和 1 两种情况的信号量称之为二值信号量。

在多任务系统中，我们经常会使用这个二值信号量，比如，某个任务需要等待一个标记，那么任务可以在轮询中查询这个标记有没有被置位，但是这样子做，就会很消耗 CPU 资源并且妨碍其他任务执行，更好的做法是任务的大部分时间处于阻塞状态（允许其他任务执行），直到某些事件发生该任务才被唤醒去执行。可以使用二进制信号量实现这种同步，当任务取信号量时，因为此时尚未发生特定事件，信号量为空，任务会进入阻塞状态；当事件的条件满足后，任务/中断便会释放信号量，告知任务这个事件发生了，任务取得信号量便被唤醒去执行对应的操作，任务执行完毕并不需要归还信号量，这样子的 CPU 的效率可以大大提高，而且实时响应也是最快的。

再比如某个任务使用信号量在等中断的标记的发生，在这之前任务已经进入了阻塞态，在等待着中断的发生，当在中断发生之后，释放一个信号量，也就是我们常说的标记，当它退出中断之后，操作系统会进行任务的调度，如果这个任务能够运行，系统就会去执行这个任务，这样子就大大提高了我们的效率。

二值信号量在任务与任务中同步的应用场景：假设我们有一个温湿度的传感器，假设是 1s 采集一次数据，那么我们让它们在液晶屏中显示数据出来，这个周期也是要 1s 一次的，如果液晶屏刷新的周期是 100ms 更新一次，那么此时的温湿度的数据还没更新，液晶屏根本无需刷新，只需要在 1s 后温湿度数据更新的时候刷新即可，否则 CPU 就是白白做了多次的无效数据更新，CPU 的资源就无处释放这个任务占用了大半，造成 CPU 资源浪费，如果液晶屏刷新的周期是 10s 更新一次，那么温湿度的数据都变化了 10 次，液晶屏才来更新数据，那拿这个产品有啥用，根本就是不准确的，所以，还是需要同步协调工作，在温湿度采集完毕之后，进行液晶屏数据的刷新，这样子，才是最准确的，并且不会浪费 CPU 的资源。

同理，二值信号量在任务与中断同步的应用场景：我们在串口接收中，我们不知道啥时候有数据发送过来，有一个任务是做接收这些数据处理，总不能在任务中每时每刻都在任务查询有没有数据到来，那样会浪费 CPU 资源，所以在这种情况下使用二值信号量是很好的办法，当没有数据到来的时候，任务就进入阻塞态，不参与任务的调度，等到数据到来了，释放一个二值信号量，
任务就立即从阻塞态中解除，进入就绪态，然后运行的时候处理数据，这样子系统的资源就会很好的被利用起来。

而计数信号量则用于资源统计，比如当前任务来了很多个消息，但是这些消息都放在缓冲区中，尚未处理，这时候就可以利用计数信号量对这些资源进行统计，每来一个消息就加一，每处理完一个消息就减一，这样子系统就知道有多少资源未处理的。

23.3 二值信号量运作机制

创建信号量时，系统会为创建的信号量对象分配内存，并把可用信号量初始化为用户自定义的个数，二值信号量的最大可用信号量个数为 1。

二值信号量获取，任何任务都可以从创建的二值信号量资源中获取一个二值信号量，获取成功则返回正确，否则任务会根据用户指定的阻塞超时时间来等待其他任务/中断释放信号量。在等待这段时间，系统将任务变成阻塞态，任务将被挂到该信号量的阻塞等待列表中。

在二值信号量无效的时候，假如此时有任务获取该信号量的话，那么任务将进入阻塞状态，具体见图信号量无效时候获取。

假如某个时间中断/任务释放了信号量，其过程具体见图中断_任务释放信号量，那么，由于获取无效信号量而进入阻塞态的任务将获得信号量并且恢复为就绪态，其过程具体见图二值信号量运作机制。
23.4 计数信号量运作机制

计数信号量可以用于资源管理，允许多个任务获取信号量访问共享资源，但会限制任务的最大数目。访问的任务数达到可支持的最大数目时，会阻塞其他试图获取该信号量的任务，直到有任务释放了信号量。这就是计数型信号量的运作机制，虽然计数信号量允许多个任务访问同一个资源，但也有限定，比如某个资源限定只能有 3 个任务访问，那么第 4 个任务访问的时候，会因为获取不到信号量而进入阻塞，等到有任务（比如任务 1）释放掉该资源的时候，第 4 个任务才能获取到信号量从而进行资源的访问，其运作的机制具体见图计数信号量运作示意图。
23.5 信号量控制块

μC/OS 的信号量由多个元素组成，在信号量被创建时，需要由我们自己定义信号量控制块（也可以称之为信号量句柄），因为它是用于保存信号量的一些信息的，其数据结构 OS_SEM 除了信号量必须的一些基本信息外，还有 PendList 链表与 Ctr，为的是方便系统来管理信号量。其数据结构具体见 代码清单: 消息队列-2，示意图具体见图信号量的控制块数据结构。
列表 1: 代码清单: 信号量-1 消息队列结构

```
struct os_sem
{
    OS_OBJ_TYPE Type;  // (1)
    CPU_CHAR *NamePtr; // (2)
    OS_PEND_LIST PendList; // (3)

#if OS_CFG_DBG_EN > 0u
    OS_SEM *DbgPrevPtr;
    OS_SEM *DbgNextPtr;
    CPU_CHAR *DbgNamePtr;
#endif
```

(下页继续)
OS_SEM_CTR Ctr; (4)
CPU_TS TS; (5)

- 代码清单: 信号量-1 (1): 信号量的类型，用户无需理会。
- 代码清单: 信号量-1 (2): 信号量的名字。
- 代码清单: 信号量-1 (3): 等待信号量的任务列表。
- 代码清单: 信号量-1 (4): 可用信号量的个数，如果为0则表示无可用信号量。
- 代码清单: 信号量-1 (5): 用于记录时间戳。

### 23.6 信号量函数接口讲解

#### 23.6.1 创建信号量函数 OSSemCreate()

在定义完信号量结构体变量后就可以调用 OSSemCreate() 函数进行创建一个信号量，跟消息队列的创建差不多，我们知道，其实这里的“创建信号量”指的就是对内核对象（信号量）的一些初始化。要特别注意的是内核对象使用之前一定要先创建，这个创建过程必须要保证在所有可能使用内核对象的任务之前，所以一般我们都是在创建任务之前就创建好系统需要的内核对象（如信号量等），创建信号量函数 OSSemCreate() 源码具体见代码清单: 信号量-2。

| 1 | void OSSemCreate (OS_SEM *p_sem, (1) \//信号量控制块指针 |
| 2 | CPU_CHAR *p_name, (2) \//信号量名称 |
| 3 | OS_SEM_CTR cnt, (3) \//资源数目或事件是否发生标志 |
| 4 | OS_ERR *p_err) (4) \//返回错误类型 |
| 5 | |

（下页继续）
CPU_SR_ALLOC();

// 使用到临界段（在关/开中断时）时必须用到该宏，该宏声明和定义一个局部变量，用于保存关中断前的CPU状态寄存器SR（临界段关中断前需保存SR）
//，开中断时将该值还原。

#ifdef OS_SAFETY_CRITICAL(5) //如果启用（默认禁用）了安全检测
    if (p_err == (OS_ERR *) 0) //如果错误类型实参为空
    {
        OS_SAFETY_CRITICAL_EXCEPTION(); //执行安全检测异常函数
        return; //返回，不继续执行
    }
#endif

#ifdef OS_SAFETY_CRITICAL_IEC61508(6) //如果启用（默认禁用）了安全关键
//如果是在调用OSSafetyCriticalStart()后创建该信号量
    if (OSSafetyCriticalStartFlag == DEF_TRUE)
    {
        *p_err = OS_ERR_ILLEGAL_CREATE_RUN_TIME; //错误类型为“非法创建内核对象”
        return; //返回，不继续执行
    }
#endif

#if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u (7)
//如果启用（默认启用）了中断中非法调用检测
    if (OSIntNestingCtr > (OS_NESTING_CTR) 0) //如果该函数是在中断中被调用
    {
        *p_err = OS_ERR_CREATE_ISR; //错误类型为“在中断函数中创建对象”
        return; //返回，不继续执行
    }
#endif

#if OS_CFG_ARG_CHK_EN > 0u(8) //如果启用（默认启用）了参数检测

```c
if (p_sem == (OS_SEM *) 0)  //如果参数 p_sem 为空
{
    *p_err = OS_ERR_OBJ_PTR_NULL;  //错误类型为“信号量对象为空”
    return;  //返回，不继续执行
}

#if OS_CFG_DBG_EN > 0u  //如果启用（默认启用）了调试代码和变量
    OS_SemDbgListAdd(p_sem);  //将该定时添加到信号量双向调试链表
#endif

OS_CRITICAL_ENTER();  //进入临界段
p_sem->Type     = OS_OBJ_TYPE_SEM;  (9)  //初始化信号量指标
p_sem->Ctr     = cnt;
p_sem->TS     = (CPU_TS) 0;
p_sem->NamePtr = p_name;
    OS_PendListInit(&p_sem->PendList);  (10) //初始化该信号量的等待列表

#if OS_CFG_DBG_EN > 0u //如果启用（默认启用）了调试代码和变量
    OS_SemDbgListAdd(p_sem);  //将该定时添加到信号量双向调试链表
#endif

    OSSemQty++;  (11) //信号量个数加 1

OS_CRITICAL_EXIT_NO_SCHED();  //退出临界段（无调度）
*p_err = OS_ERR_NONE;  //错误类型为“无错误”
```

- **代码清单: 信号量-2 (1)**: 信号量控制块指针，指向我们定义的信号量控制块结构体变量，所以在创建之前我们需要先定义一个信号量控制块变量。

- **代码清单: 信号量-2 (2)**: 信号量名称，字符串形式。

- **代码清单: 信号量-2 (3)**: 这个值表示初始化时候资源的个数或事件是否发生标志，一般信号量是二值信号量的时候，这个值一般为 0 或者为 1，而如果信号量作为计数信号量的时候，这个值一般定义为初始资源的个数。

- **代码清单: 信号量-2 (4)**: 用于保存返回错误类型。
· 代码清单: 信号量-2 (5): 如果启用了安全检测（默认禁用），在编译时则会包含安全检测相关的代码，如果错误类型实参为空，系统会执行安全检测异常函数，然后返回，不执行创建信号量操作。

· 代码清单: 信号量-2 (6): 如果启用（默认禁用）了安全关键检测，在编译时则会包含安全关键检测相关的代码，如果在调用 OSSafetyCriticalStart() 后创建该信号量，则是非法的，返回错误类型为“非法创建内核对象”错误代码，并且退出，不执行创建信号量操作。

· 代码清单: 信号量-2 (7): 如果启用了中断中非法调用检测（默认启用），在编译时则会包含中断非法调用检测相关的代码，如果该函数是在中断中被调用，则是非法的，返回错误类型为“在中断中创建对象”的错误代码，并且退出，不执行创建信号量操作。

· 代码清单: 信号量-2 (8): 如果启用了参数检测（默认启用），在编译时则会包含参数检测相关的代码，如果 p_sem 参数为空, 返回错误类型为“创建对象为空”的错误代码，并且退出，不执行创建信号量操作。

· 代码清单: 信号量-2 (9): 进入临界段，然后进行初始化信号量相关信息，如初始化信号量的类型、名字、可用信号量 Ctr、记录时间戳的变量 TS 等。

· 代码清单: 信号量-2 (10): 调用 OS_PendListInit() 函数初始化该信号量的等待列表。

· 代码清单: 信号量-2 (11): 系统信号量个数加 1。

如果我们创建一个初始可用信号量个数为 5 的信号量，那么信号量创建成功的示意图具体见图信号量创建成功示意图。

论坛: https://www.firebbs.cn/ 447 天猫: https://yehuosm.tmall.com
创建信号量函数 OSSemCreate() 的使用实例具体见代码清单: 信号量-3。
列表 3: 代码清单: 信号量-3 OSSemCreate() 使用实例

```c
OS_SEM SemOfKey;          // 标志 KEY1 是否被按下的信号量
/* 创建信号量 SemOfKey */
OSSemCreate((OS_SEM *) &SemOfKey,   // 指向信号量变量的指针
            (CPU_CHAR *) "SemOfKey", // 信号量的名字
            (OS_SEM_CTR ) 0,
            // 信号量这里是指示事件发生，所以赋值为 0，表示事件还没有发生
            (OS_ERR *) &err);        // 错误类型
```

### 23.6.2 信号量删除函数 OSSemDel()

OSSemDel() 用于删除一个信号量，信号量删除函数是根据信号量结构（信号量句柄）直接删除的，删除之后这个信号量的所有信息都会被系统清空，而且不能再次使用这个信号量了，但是需要注意的是，如果某个信号量没有被定义，那也是无法被删除的，如果有任务阻塞在该信号量上，那么尽量不要删除该信号量。想要使用互斥量删除函数就必须将 OS_CFG_SEM_DEL_EN 宏定义配置为 1，其函数源码具体见代码清单: 信号量-4。

列表 4: 代码清单: 信号量-4 OSSemDel() 源码

```c
#if OS_CFG_SEM_DEL_EN > 0u // 如果启用了 OSSemDel() 函数
OS_OBJ_QTY OSSemDel (OS_SEM *p_sem,   (1)    // 信号量指针
             OS_OPT      opt,      (2)    // 选项
             OS_ERR      *p_err)   (3)    // 返回错误类型
{
    OS_OBJ_QTY cnt;
    OS_OBJ_QTY nbr_tasks;
    OS_PEND_DATA *p_pend_data;
    OS_PEND_LIST *p_pend_list;
    OS_TCB        *p_tcb;
```

(下页继续)
CPU_TS    ts;
CPU_SR_ALLOC();

#ifdef OS_SAFETY_CRITICAL(4) //如果启用（默认禁用）了安全检测
    if (p_err == (OS_ERR *)0)  //如果错误类型实参为空
    {
        OS_SAFETY_CRITICAL_EXCEPTION();  //执行安全检测异常函数
        return ((OS_OBJ_QTY)0);  //返回 0 (有错误)，不继续执行
    }
#endif

#if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u(5) //如果启用了中断中非法调用检测
    if (OSIntNestingCtr > (OS_NESTING_CTR)0)  //如果该函数在中断中被调用
    {
        *p_err = OS_ERR_DEL_ISR;  //返回错误类型为“在中断中删除”
        return ((OS_OBJ_QTY)0);  //返回 0 (有错误)，不继续执行
    }
#endif

#if OS_CFG_ARG_CHK_EN > 0u(6) //如果启用了参数检测
    if (p_sem == (OS_SEM *)0)  //如果 p_sem 为空
    {
        *p_err = OS_ERR_OBJ_PTR_NULL;  //返回错误类型为“内核对象为空”
        return ((OS_OBJ_QTY)0);  //返回 0 (有错误)，不继续执行
    }
switch (opt) (7) //根据选项分类处理
{
    case OS_OPT_DEL_NO_PEND:  //如果选项在预期之内
    case OS_OPT_DEL_ALWAYs:
        break;  //直接跳出

default: (8) //如果选项超出预期
        *p_err = OS_ERR_OPT_INVALID; //返回错误类型为“选项非法”
    return ((OS_OBJ_QTY)0); //返回 0 (有错误)，不继续执行
}
#endif

#if OS_CFG_OBJ_TYPE_CHK_EN > 0u(9) //如果启用了对象类型检测

if (p_sem->Type != OS_OBJ_TYPE_SEM) //如果 p_sem 不是信号量类型
{
    *p_err = OS_ERR_OBJ_TYPE; //返回错误类型为“内核对象类型错误”
    return ((OS_OBJ_QTY)0); //返回 0 (有错误)，不继续执行
}
#endif

CPU_CRITICAL_ENTER(); //关中断
p_pend_list = &p_sem->PendList; (10) //获取信号量的等待列表到 p_pend_list
cnt = p_pend_list->NbrEntries; //获取等待该信号量的任务数
nbr_tasks = cnt;
switch (opt) (11) //根据选项分类处理
{
    case OS_OPT_DEL_NO_PEND: (12)
        //如果只在没有任务等待的情况下删除信号量
        if (nbr_tasks == (OS_OBJ_QTY)0) //如果没有任务在等待该信号量
            {
            #if OS_CFG_DBG_EN > 0u //如果启用了调试代码和变量
            OS_SemDbgListRemove(p_sem); //将该信号量从信号量调试列表移除
            #endif
            OSSemQty--; //信号量数目减 1
            OS_SemClr(p_sem); //清除信号量内容
            CPU_CRITICAL_EXIT(); //开中断
            *p_err = OS_ERR_NONE; (15) //返回错误类型为“无错误”

else \(16\) //如果有任务在等待该信号量
{
    CPU_CRITICAL_EXIT(); \(//\)开中断
    \*p_err = OS_ERR_TASK_WAITING;
    //返回错误类型为“有任务在等待该信号量”
}
break;

case OS_OPT_DEL_ALWAYS: \(17\)//如果必须删除信号量
    OS_CRITICAL_ENTER_CPU_EXIT(); \(//\)锁调度器，并开中断
    ts = OS_TS_GET(); \(18\)//获取时间戳
    while (cnt > 0u) \(19\)
        //逐个移除该信号量等待列表中的任务
        {
            p_pend_data = p_pend_list->HeadPtr;
            p_tcb = p_pend_data->TCBPtr;
            OS_PendObjDel((OS_PEND_OBJ *)((\*void *)p_sem),
                p_tcb,
                ts); \((20)\)
            cnt--;  
        }
#endif OS_CFG_DBG_EN > 0u//如果启用了调试代码和变量
    OS_SemDbgListRemove(p_sem); \(//\)将该信号量从信号量调试列表移除
#endif
OSSemQty--; \(21\)//信号量数目减 1
OS_SemClr(p_sem); \(22\)//清除信号量内容
OS_CRITICAL_EXIT_NO_SCHED(); \(//\)锁调度器，但不进行调度
OSSched(); \(23\)
//任务调度，执行最高优先级的就绪任务
\*p_err = OS_ERR_NONE; \(//\)返回错误类型为“无错误”
break;

default: (24) //如果选项超出预期
    CPU_CRITICAL_EXIT(); //开中断
    *p_err = OS_ERR_OPT_INVALID; //返回错误类型为“选项非法”
break;

return ((OS_OBJ_QTY)nbr_tasks); (25) //返回删除信号量前等待其的任务数
}
#endif

- **代码清单: 信号量-4 (1)**: 信号量控制块指针，指向我们定义的信号量控制块结构体变量，所以在删除之前我们需要先定义一个信号量控制块变量，并且成功创建信号量后再进行删除操作。

- **代码清单: 信号量-4 (2)**: 删除的选项。

- **代码清单: 信号量-4 (3)**: 用于保存返回的错误类型。

- **代码清单: 信号量-4 (4)**: 如果启用了安全检测（默认），在编译时则会包含安全检测相关的代码，如果错误类型实参为空，系统会执行安全检测异常函数，然后返回，不执行删除信号量操作。

- **代码清单: 信号量-4 (5)**: 如果启用了中断中非法调用检测（默认启用），在编译时则会包含中断非法调用检测相关的代码，如果该函数是在中断中被调用，则是非法的，返回错误类型为“在中断中删除对象”的错误代码，并且退出，不执行删除信号量操作。

- **代码清单: 信号量-4 (6)**: 如果启用了参数检测（默认启用），在编译时则会包含参数检测相关的代码，如果 p_sem 参数为空，返回错误类型为“内核对象为空”的错误代码，并且退出，不执行删除信号量操作。

- **代码清单: 信号量-4 (7)**: 判断 opt 选项是否合理，该选项有两个，OS_OPT_DEL_ALWAYS 与 OS_OPT_DEL_NO_PEND，在 os.h 文件中定义。此处是判断一下选项是否在预期之内，如果在则跳出 switch 语句。
• 代码清单: 信号量-4 (8): 如果选项超出预期，则返回错误类型为“数据非法”的错误代码，退出，不继续执行。

• 代码清单: 信号量-4 (9): 如果启用了信号量检测，在构建时则会包含信号量检测的代码，如果 p_sem 非信号量类型，返回错误类型为“内核对象类型错误”的错误代码，并且退出，不执行删除信号量操作。

• 代码清单: 信号量-4 (10): 程序执行到这里，表示可以删除信号量了，系统首先获取信号量的等待列表保存到 p_pend_list 变量中。然后再获取该信号量的任务数。

• 代码清单: 信号量-4 (11): 根据选项分类处理

• 代码清单: 信号量-4 (12): 如果 opt 是 OS_OPT_DEL_NO_PEND，则表示只在没有任务等待的情况下删除信号量，如果当前系统中有任务阻塞在该信号量上，则不能删除，反之，则可以删除信号量。

• 代码清单: 信号量-4 (13): 如果没有任务在等待该信号量，信号量数目减 1。

• 代码清单: 信号量-4 (14): 清除信号量内容。

• 代码清单: 信号量-4 (15): 删除成功，返回错误类型为“无错误”的错误代码。

• 代码清单: 信号量-4 (16): 如果有任务在等待该信号量，则返回错误类型为“有任务在等待该信号量”错误代码。

• 代码清单: 信号量-4 (17): 如果 opt 是 OS_OPT_DEL_ALWAYS，则表示无论如何都必须删除信号量，那么在删除之前，系统会把所有阻塞在该信号量上的任务恢复。

• 代码清单: 信号量-4 (18): 调用 OS_PendObjDel() 函数将阻塞在内核对象（如信号量）上的任务从阻塞态恢复，此时系统在删除内核对象，删除之后，这些等待事件的任务需要被恢复，其源码具体见代码清单: 消息队列-8。

• 代码清单: 信号量-4 (20): 执行到这里，表示已经删除了信号量了，系统信号量个数减 1。

• 代码清单: 信号量-4 (22): 清除信号量内容。

论坛: https://www.firebbs.cn/ 454 天猫: https://yehuosm.tmall.com
• 代码清单: 信号量-4 (23): 进行一次任务调度。

• 代码清单: 信号量-4 (24): 如果选项超出预期，返回错误类型为“选项非法”的错误代码，退出。

• 代码清单: 信号量-4 (25): 返回删除信号量前阻塞在该信号量上的任务个数。

信号量删除函数 OSSemDel() 的使用也是很简单的，只需要传入要删除的信号量的句柄与选项还有保存返回的错误类型即可，调用函数时，系统将删除这个信号量。需要注意的是在调用删除信号量函数前，系统应存在已创建的信号量。如果删除信号量时，系统中有任务正在等待该信号量，则不应该进行删除操作，因为删除之后的信号量就不可用了，删除信号量函数 OSSemDel() 的使用实例具体见代码清单: 信号量-5。

列表 5: 代码清单: 信号量-5OSSemDel() 使用实例

```c
OS_SEM SemOfKey; //声明信号量
OS_ERR err;

/* 删除信号量 sem*/
OSSemDel ((OS_SEM *)&SemOfKey, //指向信号量的指针
OS_OPT_DEL_NO_PEND,
OS_ERR *)&err); //返回错误类型
```

23.6.3 信号量释放函数 OSSemPost()

与消息队列的操作一样，信号量的释放可以在任务、中断中使用。

在前面的讲解中，我们知道，当信号量有效的时候，任务才能获取信号量，那么，是什么函数使得信号量变得有效？其实有两个方式，一个是在创建的时候进行初始化，将它可用的信号量个数设置一个初始值；如果该信号量用作二值信号量，那么我们在创建信号量的时候其初始值的范围是 0~1，假如初始值为 1 个可用的信号量的话，被获取一次就变得无效了，那就需要我们释放信号量，μC/OS 提供了信号量释放函数，每调用一次该函数就释放一个信号量。但是有个问题，能不能一直释放？很显然如果用作二值信号量的话，一直释放信号量就达不到同步或者互斥访问的
效果，虽然说μC/OS 的信号量是允许一直释放的，但是，信号量的范围还需我们用户自己根据需求进行决定，当用作二值信号量的时候，必须确保其可用值在 0-1 范围内；而用作计数信号量的话，其范围是由用户根据实际情况来决定的，在写代码的时候，我们要注意代码的严谨性罢了，信号量释放函数的源码具体见代码清单: 信号量-6。

| 1 | OS_SEM_CTR OSSemPost (OS_SEM *p_sem, (1)     //信号量控制块指针 |
| 2 |     OS_OPT opt, (2)     //选项 |
| 3 |     OS_ERR *p_err)     //返回错误类型 |
| 4 | { |
| 5 |     OS_SEM_CTR ctr; |
| 6 |     CPU_TS ts; |
| 7 | |
| 8 | #ifdef OS_SAFETY_CRITICAL//如果启用（默认禁用）了安全检测 |
| 9 |     if (p_err == (OS_ERR *0))     //如果错误类型实参为空 |
| 10 |     { |
| 11 |         OS_SAFETY_CRITICAL_EXCEPTION();     //执行安全检测异常函数 |
| 12 |         return ((OS_SEM_CTR)0);     //返回 0 (有错误)，不继续执行 |
| 13 |     } |
| 14 | #endif |
| 15 | #if OS_CFG_ARG_CHK_EN > 0u//如果启用（默认启用）了参数检测功能 |
| 16 |     if (p_sem == (OS_SEM *0))     //如果 p_sem 为空 |
| 17 |     { |
| 18 |         *p_err = OS_ERR_OBJ_PTR_NULL;     //返回错误类型为“内核对象指针为空” |
| 19 |         return ((OS_SEM_CTR)0);     //返回 0 (有错误)，不继续执行 |
| 20 |     } |
| 21 |     switch (opt)     //根据选项情况分类处理 |
| 22 |     { |
| 23 |         case OS_OPT_POST_1:     //如果选项在预期内，不处理 |
| 24 |         //

论坛: https://www.firebbs.cn/
天猫: https://yehuosm.tmall.com

(下页继续)
```c
    case OS_OPT_POST_ALL:
    case OS_OPT_POST_1 | OS_OPT_POST_NO_SCHED:
    case OS_OPT_POST_ALL | OS_OPT_POST_NO_SCHED:
        break;

    default: // 如果选项超出预期
        *p_err = OS_ERR_OPT_INVALID; // 返回错误类型为“选项非法”
        return ((OS_SEM_CTR)0u); // 返回 0 (有错误), 不继续执行
    }
#endif

#if OS_CFG_OBJ_TYPE_CHK_EN > 0u // 如果启用了对象类型检测
    if (p_sem->Type != OS_OBJ_TYPE_SEM) // 如果 p_sem 的类型不是信号量类型
        {
            *p_err = OS_ERR_OBJ_TYPE; // 返回错误类型为“对象类型错误”
            return ((OS_SEM_CTR)0); // 返回 0 (有错误), 不继续执行
        }
#endif

ts = OS_TS_GET();  // 获取时间戳

#if OS_CFG_ISR_POST_DEFERRED_EN > 0u // 如果启用了中断延迟发布
    if (OSIntNestingCtr > (OS_NESTING_CTR)0) // 如果该函数是在中断中被调用
        {
            OS_IntQPost((OS_OBJ_TYPE)OS_OBJ_TYPE_SEM, // 将该信号量发布到中断消息队列
                (void  *)p_sem,
                (void  *)0,
                (OS_MSG_SIZE)0,
                (OS_FLAGS )0,
                (OS_OPT   )opt,
                (CPU_TS   )ts,
                (OS_ERR   *)p_err); (4)
```
59    return ((OS_SEM_CTR)0);  //返回 0 (尚未发布), 不继续执行
60 }
61 #endif
62
63    ctr = OS_SemPost(p_sem,  //将信号量按照普通方式处理
64           opt,
65           ts,
66           p_err);        (5)
67
68    return (ctr);          //返回信号的当前计数值
69 }

- 代码清单: 信号量-6 (1): 信号量控制块指针。
- 代码清单: 信号量-6 (2): 释放信号量的选项，该选项在 os.h 中定义，具体见代码清单: 信号量-7。

列表 7: 代码清单: 信号量-7 释放信号量选项

1    #define OS_OPT_POST_FIFO  (OS_OPT) (0x0000u) (1)
2
3    #define OS_OPT_POST_LIFO  (OS_OPT) (0x0010u) (2)
4
5    #define OS_OPT_POST_1     (OS_OPT) (0x0000u) (3)
6
7    #define OS_OPT_POST_ALL   (OS_OPT) (0x0200u) (4)

- 代码清单: 信号量-7 (1): 默认采用 FIFO 方式发布信号量
- 代码清单: 信号量-7 (2): μC/OS 也支持采用 FIFO 方式发布信号量。
- 代码清单: 信号量-7 (3): 发布给一个任务。
- 代码清单: 信号量-7 (4): 发布给所有等待的任务，也叫广播信号量。
- 代码清单: 信号量-6 (3): 用于保存返回错误类型。

论坛: https://www.firebbs.cn/  458  天猫: https://yehuosm.tmall.com
• 代码清单: 信号量-6 (4): 如果启用了中断延迟发布，并且该函数在中断中被调用，则使用 OS_IntQPost() 函数将信号量发布到中断消息队列中，OS_IntQPost() 函数源码我们在前面已经将近过，就不再重复赘述。

• 代码清单: 信号量-6 (5): 将信号量按照普通方式处理，OS_SemPost() 源码具体见

列表 8: 代码清单: 信号量-8OS_SemPost() 源码

```c
OS_SEM_CTR OS_SemPost (OS_SEM *p_sem,
                        OS_OPT opt,     // 选项
                        CPU_TS ts,      // 时间戳
                        OS_ERR *p_err); // 返回错误类型
{
    OS_OBJ_QTY cnt;
    OS_SEM_CTR ctr;
    OS_PEND_LIST *p_pend_list;
    OS_PEND_DATA *p_pend_data;
    OS_PEND_DATA *p_pend_data_next;
    OS_TCB *p_tcb;

    CPU_SR_ALLOC();

    CPU_CRITICAL_ENTER(); // 关中断
    p_pend_list = &p_sem->PendList; (1) // 取出该信号量的等待列表
    if (p_pend_list->NbrEntries == (OS_OBJ_QTY) 0) (2)
    {
        // 判断是否将导致该信号量计数值溢出，
        switch (sizeof(OS_SEM_CTR)) (3)
        {
            case 1u: (4)
                // 如果溢出，则开中断，返回错误类型为
```

论坛: https://www.firebbs.cn/  天猫: https://yehuosm.tmall.com
if (p_sem->Ctr == DEF_INT_08U_MAX_VAL)
  // “计数值溢出”, 返回 0 (有错误),
  {
    CPU_CRITICAL_EXIT(); // 不继续执行。
    *p_err = OS_ERR_SEM_OVF;
    return ((OS_SEM_CTR)0);
  }
break;

case 2u:
  if (p_sem->Ctr == DEF_INT_16U_MAX_VAL)
  {
    CPU_CRITICAL_EXIT();
    *p_err = OS_ERR_SEM_OVF;
    return ((OS_SEM_CTR)0);
  }
break;

case 4u:
  if (p_sem->Ctr == DEF_INT_32U_MAX_VAL)
  {
    CPU_CRITICAL_EXIT();
    *p_err = OS_ERR_SEM_OVF;
    return ((OS_SEM_CTR)0);
  }
break;

default:
  break;
}
p_sem->Ctr++; // (5) // 信号量计数值不溢出则加 1
ctr = p_sem->Ctr; // 获取信号量计数值到 ctr

p_sem->TS = ts;  //保存时间戳
CPU_CRITICAL_EXIT();  //则开中断
*p_err = OS_ERR_NONE;  //返回错误类型为“无错误”
return (ctr);  //返回信号量的计数值，不继续执行

OS_CRITICAL_ENTER_CPU_EXIT();  //加锁调度器，但开中断
if ((opt & OS_OPT_POST_ALL) != (OS_OPT) 0)
//如果要将信号量发布给所有等待任务
{
    cnt = p_pend_list->NbrEntries;  //获取等待任务数目到 cnt
}
else
//如果要将信号量发布给优先级最高的等待任务
{
    cnt = (OS_OBJ_QTY) 1;  //将要操作的任务数为 1, cnt 置 1
}
p_pend_data = p_pend_list->HeadPtr;  //获取等待列表的首个任务到 p_pend_data

while (cnt > 0u)  //逐个处理要发布的任务
{
    p_tcb = p_pend_data->TCBPtr;  //取出当前任务
    p_pend_data_next = p_pend_data->NextPtr;  //取出下一个任务
    OS_Post((OS_PEND_OBJ *)((void *)p_sem),
        p_tcb,
        (void *)0,
        (OS_MSG_SIZE) 0,
        ts);  //发布信号量给当前任务

    p_pend_data = p_pend_data_next;  //处理下一个任务
    cnt--;  //
91 } 
92 ctr = p_sem->Ctr; //获取信号量计数值到 ctr 
93 OS_CRITICAL_EXIT_NO_SCHED(); (14) 
94 //减锁调度器，但不执行任务调度 
95 //如果 opt 没选择“发布时不调度任务” 
96 if ((opt & OS_OPT_POST_NO_SCHED) == (OS_OPT)0) 
97 { 
98 OSSched(); (15) //任务调度 
99 } 
100 *p_err = OS_ERR_NONE; (16) //返回错误类型为“无错误” 
101 return (ctr); //返回信号量的当前计数值

• 代码清单: 信号量-8 (1): 取出该信号量的等待列表保存在 p_pend_list 变量中。

• 代码清单: 信号量-8 (2): 判断一下有没有任务在等待该信号量，如果没有任务在等待该信号量，则要先看看信号量的信号量计数值是否即将溢出。

• 代码清单: 信号量-8 (3): 怎么判断计数值是否溢出呢？μC/OS 支持多个数据类型的信号量计数值，可以是 8 位的，16 位的，32 位的，具体是多少位是由我们自己定义的。

• 代码清单: 信号量-8 (4): 先看看 OS_SEM_CTR 的大小是个字节，如果是 1 个字节，表示 Ctr 计数值是 8 位的，则判断一下 Ctr 是否到达了 DEF_INT_08U_MAX_VAL，如果到达了，再释放信号量就会溢出，那么就会返回错误类型为“计数值溢出”的错误代码。而对于 OS_SEM_CTR 是 2 字节、4 字节的也是同样的判断操作。

• 代码清单: 信号量-8 (5): 程序能执行到这里，说明信号量的计数值不溢出，此时释放信号量则需要将 Ctr 加 1。

• 代码清单: 信号量-8 (6): 保存释放信号量时的时间戳。

• 代码清单: 信号量-8 (7): 返回错误类型为“无错误”的错误代码，然后返回信号量的计数值，不继续执行。

• 代码清单: 信号量-8 (8): 而程序能执行到这里，说明系统中有任务阻塞在该信号量上，此时我们释放了一个信号量，就要将等待的任务进行恢复，但是恢复一个还是恢复所有任务得
看用户自定义的释放信号量选项。所以此时不管三七二十一将调度器锁定，但开中断，因为接下来的操作是需要操作任务与信号量的列表，系统不希望其他任务来打乱。

- 代码清单: 信号量-8 (9): 如果要将信号量释放给所有等待任务，首先获取等待该信号量的任务个数到变量 cnt 中，用来记录即将要进行释放信号量操作的任务个数。
- 代码清单: 信号量-8 (10): 如果要将信号量释放给优先级最高的等待任务，将要操作的任务数为 1，所以将 cnt 置 1。
- 代码清单: 信号量-8 (11): 逐个处理要释放信号量的任务。
- 代码清单: 信号量-8 (12): 调用 OS_Post() 函数进行对应的任务释放信号量，因为该源码在前面就讲解过了，此处就不再重复赘述，具体见 代码清单: 消息队列-14。
- 代码清单: 信号量-8 (13): 处理下一个任务。
- 代码清单: 信号量-8 (14): 减锁调度器，但不执行任务调度。
- 代码清单: 信号量-8 (15): 如果 opt 没选择“发布时不调度任务”，那么就进行任务调度。
- 代码清单: 信号量-8 (16): 操作完成，返回错误类型为“无错误”的错误代码，并且返回信号量的当前计数值。

如果可用信号量未满，信号量控制块结构体成员变量 Ctr 就会加 1，然后判断是否有阻塞的任务，如果有的话就会恢复阻塞的任务，然后返回成功信息，用户可以选择只释放（发布）给一个任务或者是释放（发布）给所有在等待信号量的任务（广播信号量），并且用户可以选择在释放（发布）完成后要不要进行任务调度，如果信号量在中断中释放，用户可以选择是否需要延迟释放（发布）。

信号量的释放函数的使用很简单，具体见 代码清单: 信号量-9

列表 9: 代码清单: 信号量-9OSSemPost() 使用实例

```
OS_SEM SemOfKey;           //标志 KEY1 是否被按下的信号量
OSSemPost((OS_SEM *)&SemOfKey,  //发布 SemOfKey
           (OS_OPT )OS_OPT_POST_ALL,    //发布给所有等待任务
           (OS_ERR *)&err);           //返回错误类型
```
23.6.4 信号量获取函数 OSSemPend()

与消息队列的操作一样，信号量的获取可以在任务中使用。

与释放信号量对应的是获取信号量，我们知道，当信号量有效的时候，任务才能获取信号量，当任务获取了某个信号量的时候，该信号量的可用个数就减一，当它减到 0 的时候，任务就无法再获取了，并且获取的任务会进入阻塞态（假如用户指定了阻塞超时时间的话）。如果某个信号量中当前拥有 1 个可用的信号量的话，被获取一次就变得无效了，那么此时另外一个任务获取该信号量的时候，就会无法获取成功，该任务便会进入阻塞态，阻塞时间由用户指定。

μC/OS 支持系统中多个任务获取同一个信号量，假如信号量中已有多个任务在等待，那么这些任务会按照优先级顺序进行排列，如果信号量在释放的时候选择只释放给一个任务，那么在所有等待任务中最高优先级的任务优先获得信号量，而如果信号量在释放的时候选择释放给所有任务，则所有等待的任务都会获取到信号量。信号量获取函数 OSSemPend() 源码具体见。

列表 10: 代码清单: 信号量-10 OSSemPend() 源码

```c
OS_SEM_CTR OSSemPend (OS_SEM *p_sem, (1) //信号量指针
    OS_TICK timeout, (2) //等待超时时间
    OS_OPT opt, (3) //选项
    CPU_TS *p_ts, (4) //等到信号量时的时间戳
    OS_ERR *p_err) (5) //返回错误类型
{
    OS_SEM_CTR ctr;
    OS_PEND_DATA pend_data;
    CPU_SR_ALLOC();

    #ifdef OS_SAFETY_CRITICAL(6)//如果启用（默认禁用）了安全检测
        if (p_err == (OS_ERR *)) (6) //如果错误类型实参为空
            {
                OS_SAFETY_CRITICAL_EXCEPTION(); //执行安全检测异常函数
                return ((OS_SEM_CTR)0); //返回 0 (有错误)，不继续执行
            } (下页继续)
```
```c
#endif

#if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u(7) //如果启用了中断中非法调用检测
    if (OSIntNestingCtr > (OS_NESTING_CTR)0) //如果该函数在中断中被调用
        { 
            *p_err = OS_ERR_PEND_ISR; //返回错误类型为 “在中断中等待”
            return ((OS_SEM_CTR)0); //返回 0 (有错误)，不继续执行
        }
#endif

#if OS_CFG_ARG_CHK_EN > 0u(8) //如果启用了参数检测
    if (p_sem == (OS_SEM *)0) //如果 p_sem 为空
        { 
            *p_err = OS_ERR_OBJ_PTR_NULL; //返回错误类型为 “内核对象为空”
            return ((OS_SEM_CTR)0); //返回 0 (有错误)，不继续执行
        }
    switch (opt) { //根据选项分类处理
        case OS_OPT_PEND_BLOCKING: //如果选择 “等待不到对象进行阻塞”
            case OS_OPT_PEND_NON_BLOCKING: //如果选择 “等待不到对象不进行阻塞”
                break; //直接跳出，不处理
        default: //如果选项超出预期
            *p_err = OS_ERR_OPT_INVALID; //返回错误类型为 “选项非法”
            return ((OS_SEM_CTR)0); //返回 0 (有错误)，不继续执行
    }
#endif

#if OS_CFG_OBJ_TYPE_CHK_EN > 0u(11) //如果启用了对象类型检测
    if (p_sem->Type != OS_OBJ_TYPE_SEM) //如果 p_sem 不是信号量类型
        {
```

*p_err = OS_ERR_OBJ_TYPE;  // 返回错误类型为“内核对象类型错误”
return ((OS_SEM_CTR)0);  // 返回 0 (有错误)，不继续执行
}
#endif

if (p_ts != (CPU_TS *)0)  (12)// 如果 p_ts 非空
{
  *p_ts = (CPU_TS)0;
  // 初始化 (清零) p_ts，待用于返回时间戳
}
CPU_CRITICAL_ENTER();  // 关中断
if (p_sem->Ctr > (OS_SEM_CTR)0)  (13)// 如果资源可用
{
  p_sem->Ctr--;  (14)// 资源数目减 1
  if (p_ts != (CPU_TS *)0)  (15)// 如果 p_ts 非空
  {
    *p_ts = p_sem->TS;  // 获取该信号量最后一次发布的知识
  }
  ctr = p_sem->Ctr;  (16)// 获取信号量的当前资源数目
CPU_CRITICAL_EXIT();  // 开中断
*p_err = OS_ERR_NONE;  // 返回错误类型为“无错误”
return (ctr);  // 返回信号量的当前资源数目，不继续执行
}
#if ((opt & OS_OPT_PEND_NON_BLOCKING) != (OS_OPT)0)  (17)
// 如果没有资源可用，而且选择了不阻塞任务
{
  ctr = p_sem->Ctr;  // 获取信号量的资源数目到 ctr
CPU_CRITICAL_EXIT();  // 开中断
  *p_err = OS_ERR_PEND_WOULD_BLOCK;
  // 返回错误类型为“等待渴求阻塞”
}
return (ctr);
    // 返回信号量的当前资源数目，不继续执行
}
else
    // 如果没有资源可用，但选择了阻塞任务
{
    if (OSSchedLockNestingCtr > (OS_NESTING_CTR) 0) // 如果调度器被锁
    {
        CPU_CRITICAL_EXIT(); // 开中断
        *p_err = OS_ERR_SCHED_LOCKED;
        // 返回错误类型为“调度器被锁”
        return ((OS_SEM_CTR) 0);
        // 返回 0（有错误），不继续执行
    }
}
OS_CRITICAL_ENTER_CPU_EXIT(); // 锁调度器，并重开中断
OS_Pend(&pend_data,
    // 阻塞等待任务，将当前任务脱离就绪列表，
    (OS_PEND_OBJ *) (void *) p_sem),
    // 并插入节拍列表和等待列表。
    OS_TASK_PEND_ON_SEM,
    timeout); // (21)

OS_CRITICAL_EXIT_NO_SCHED(); // 开调度器，但不进行调度

OSSched(); // 找到并调度最高优先级就绪任务
/* 当前任务（获得信号量）得以继续运行 */
CPU_CRITICAL_ENTER(); // 关中断
switch (OSTCBCurPtr->PendStatus) // (23)
    // 根据当前运行任务的等待状态分类处理


{  
  case OS_STATUS_PEND_OK: //如果等待状态正常  
    if (p_ts != (CPU_TS *)0) //如果 p_ts 非空  
    {  
      *p_ts = OSTCBCurPtr->TS; //获取信号被发布的时间戳  
    }  
    *p_err = OS_ERR_NONE; //返回错误类型为“无错误”  
    break;  
  
  case OS_STATUS_PEND_ABORT: //如果等待被终止中止  
    if (p_ts != (CPU_TS *)0) //如果 p_ts 非空  
    {  
      *p_ts = OSTCBCurPtr->TS; //获取等待被中止的时间戳  
    }  
    *p_err = OS_ERR_PEND_ABORT; //返回错误类型为“等待被中止”  
    break;  
  
  case OS_STATUS_PEND_TIMEOUT: //如果等待超时  
    if (p_ts != (CPU_TS *)0) //如果 p_ts 非空  
    {  
      *p_ts = (CPU_TS )0; //清零 p_ts  
    }  
    *p_err = OS_ERR_TIMEOUT; //返回错误类型为“等待超时”  
    break;  
  
  case OS_STATUS_PEND_DEL: //如果等待的内核对象被删除  
    if (p_ts != (CPU_TS *)0) //如果 p_ts 非空  
    {  
      *p_ts = OSTCBCurPtr->TS; //获取内核对象被删除的时间戳  
    }  
    *p_err = OS_ERR_OBJ_DEL; //返回错误类型为“等待对象被删除”  
}
break;

default: //如果等待状态超出预期
    *p_err = OS_ERR_STATUS_INVALID;
    //返回错误类型为“等待状态非法”
    CPU_CRITICAL_EXIT(); //开中断
    return ((OS_SEM_CTR)0); //返回 0 (有错误)，不继续执行
}

ctr = p_sem->Ctr; //获取信号量的当前资源数目
CPU_CRITICAL_EXIT(); //开中断
return (ctr); //返回信号量的当前资源数目

- 代码清单: 信号量-10 (1): 信号量指针。
- 代码清单: 信号量-10 (2): 用户自定义的阻塞超时时间
- 代码清单: 信号量-10 (3): 获取信号量的选项，当信号量不可用的时候，用户可以选择阻塞或者不阻塞。
- 代码清单: 信号量-10 (4): 用于保存返回等到信号量时的时间戳。
- 代码清单: 信号量-10 (5): 用于保存返回的错误类型，用户可以根据此变量得知错误的原因。
- 代码清单: 信号量-10 (6): 如果启用（默认禁用）了安全检测，在编译时则会包含安全检测相关的代码，如果错误类型实参为空，系统会执行安全检测异常函数，然后返回，停止执行。
- 代码清单: 信号量-10 (7): 如果启用了中断中非法调用检测，并且如果该函数在中断中被调用，则返回错误类型为“在中断获取信号量”的错误代码，然后退出，停止执行。
- 代码清单: 信号量-10 (8): 如果启用了参数检测，在编译时则会包含参数检测相关的代码，如果 p_sem 参数为空，返回错误类型为“内核对象为空”的错误代码，并且退出，不执行获取消息操作。
- 代码清单: 信号量-10 (9): 判断一下 opt 选项是否合理，如果选择“等待不到对
象进行阻塞”（OS_OPT_PEND_BLOCKING）或者选择“等待不到对象不进行阻塞”（OS_OPT_PEND_NON_BLOCKING），则是合理的，跳过 switch 语句。

- 代码清单: 信号量-10 (10): 如果选项超出预期，返回错误类型为“选项非法”的错误代码，并且推出。
- 代码清单: 信号量-10 (11): 如果启用了对象类型检测，在编译时则会包含对象类型检测相关代码，如果 p_sem 不是信号量类型，那么返回错误类型为“对象类型有误”的错误代码，并且退出，不执行获取信号量操作。
- 代码清单: 信号量-10 (12): 如果 p_ts 非空，就初始化（清零）p_ts，待用于返回时间戳。
- 代码清单: 信号量-10 (13): 如果当前信号量资源可用。
- 代码清单: 信号量-10 (14): 那么被获取的信号量资源中的 Ctr 成员变量个数就要减一。
- 代码清单: 信号量-10 (15): 如果 p_ts 非空，获取该信号量最后一次发布的时钟。
- 代码清单: 信号量-10 (16): 获取信号量的当前资源数目用于返回，执行完成，那么返回错误类型为“无错误”的错误代码，退出。
- 代码清单: 信号量-10 (17): 如果没有资源可用，而且用户选择了不阻塞任务，获取信号量的资源数目到 ctr 变量用于返回，然后返回错误类型为“等待渴求阻塞”的错误代码，退出操作。
- 代码清单: 信号量-10 (18): 如果没有资源可用，但用户选择了阻塞任务，则需要判断一下调度器是否被锁。
- 代码清单: 信号量-10 (19): 如果调度器被锁，返回错误类型为“调度器被锁”的错误代码，然后退出，不执行信号量获取操作。
- 代码清单: 信号量-10 (20): 如果调度器未被锁，就锁定调度器，重新打开中断。此次可能有同学就会问了，为什么刚刚调度器被锁就错误的呢，而现在又要锁定调度器？那是因为之前锁定的调度器不是由这个函数进行锁定的，这是不允许的，因为现在要阻塞当前任务，而调度器锁定了表示无法进行任务调度，这也是不允许的。那为什么又要关闭调度器呢，因为接下来的操作是需要操作队列与任务的列表，这个时间就不会很短，系统不希望有其他任务来操作任务列表，因为可能引起其他任务解除阻塞，这可能会发生优先级翻转。比如任务 A 的优先级低于当前任务，但是在当前任务进入阻塞的过程中，任务 A 却因为其
他原因解除阻塞了，那系统肯定是会去运行任务 A，这显然是要绝对禁止的，因为挂起调度器意味着任务不能切换并且不准调用可能引起任务切换的 API 函数，所以，锁定调度器，打开中断这样的处理，既不会影响中断的响应，又避免了其他任务来操作队列与任务的列表。

- **代码清单: 信号量-10 (21)**: 调用 OS_Pend() 函数将当前任务脱离就绪列表，并根据用户指定的阻塞时间插入节拍列表和队列等待列表，然后打开调度器，但不进行调度，OS_Pend()源码具体见 代码清单: 消息队列-18。

注: OS_Pend() 源码注释很丰富，就不讲解了。

- **代码清单: 信号量-10 (22)**：当前任务阻塞了，就要进行一次任务的调度。

- **代码清单: 信号量-10 (23)**：当程序能执行到这里，就说明大体上有两种情况，要么是信号量中可以的信号量了，任务获取到信号量了；要么任务还没获取到信号量（任务没获取到信号量的情况下有很多种），无论是哪种情况，都先把中断关掉再说，再根据当前运行任务的等待状态分类处理。

- **代码清单: 信号量-10 (24)**：如果任务状态是 OS_STATUS_PEND_OK，则表示任务获取到信号量了，获取信号被释放时候的时间戳，返回错误类型为“无错误”的错误代码。

- **代码清单: 信号量-10 (25)**：如果任务在等待（阻塞）被中止，则表示任务没有获取到信号量，如果 p_ts 非空，获取等待被中止的时间戳，返回错误类型为“等待被中止”的错误代码，跳出 switch 语句。

- **代码清单: 信号量-10 (26)**：如果等待（阻塞）超时，说明等待的时间过去了，任务也没获取到信号量，如果 p_ts 非空，将 p_ts 清零，返回错误类型为“等待超时”的错误代码，跳出 switch 语句。

- **代码清单: 信号量-10 (27)**：如果等待的内核对象被删除，如果 p_ts 非空，获取对象被删时的时间戳，返回错误类型为“等待对象被删”的错误代码，跳出 switch 语句。

- **代码清单: 信号量-10 (28)**：如果等待状态超出预期，则返回错误类型为“状态非法”的错误代码，跳出 switch 语句。

- **代码清单: 信号量-10 (29)**：打开中断，返回信号量的当前资源数目。

当有任务试图获取信号量的时候，当且仅当信号量有效的时候，任务才能获取到信号量。如果信
号量无效，在用户指定的阻塞超时时间中，该任务将保持阻塞状态以等待信号量有效。当其他任务或中断释放了有效的信号量，该任务将自动由阻塞态转移为就绪态。当任务在等待的时间超过了指定的阻塞时间，即使信号量中还是没有可用信号量，任务也会自动从阻塞态转移为就绪态。

信号量获取函数 OSSemPend() 的使用实例具体见代码清单: 信号量-11。

列表11: 代码清单: 信号量-11 OSSemPend() 使用实例

```
OSSemPend ((OS_SEM*) &SemOfKey, //等待该信号量被发布
             (OS_TICK) ) 0,      //无期限等待
(OS_OPT) OS_OPT_PEND_BLOCKING, //如果没有信号量可用就等待
(CPU_TS*) &ts_sem_post, //获取信号量最后一次被发布的时时间戳
(OS_ERR*) &err);        //返回错误类型
```

23.7 使用信号量的注意事项

- 信号量访问共享资源不会导致中断延迟。当任务在执行信号量所保护的共享资源时，ISR 或高优先级任务可以抢占该任务。

- 应用中可以有任意个信号量用于保护共享资源。然而，推荐将信号量用于 I/O 端口的保护，而不是内存地址。

- 信号量经常会被过度使用。很多情况下，访问一个简短的共享资源时推荐使用信号量，请求和释放信号量会消耗 CPU 时间。通过关/开中断能更有效地执行这些操作。假设两个任务共享一个 32 位的整型变量。第一个任务将这个整型变量加 1，第二个任务将这个变量清零。考虑到执行这些操作每时很短，不需要使用信号量。执行这个操作前任务只需关中断，执行完毕后再开中断。但是若操作浮点数变量且处理器不支持硬件浮点操作时，就需要用到信号量。因为在这种情况下处理浮点数变量需较长时间。

- 信号量会导致一种严重的问题：优先级翻转。
23.8 信号量实验

23.8.1 二值信号量同步实验

信号量同步实验是在 μC/OS 中创建了两个任务，一个是获取信号量任务，一个是释放信号量任务，两个任务独立运行，获取信号量任务是等待信号量，其等待时间是无期限等待，直到获取到信号量之后，任务开始执行任务代码。释反复等待另外任务释放的信号量。

释放信号量任务在检测按键是否按，如果按下则释放信号量，此时释放信号量会唤醒获取任务，获取任务开始运行，然后形成两个任务间的同步，LED 进行翻转，如果按钮按，则释放，信号量就不会释放，只有当信号量释放的时候，获取信号量的任务才会被唤醒，如此一来就达到任务与任务的同步，同时程序的运行会在串口打印出相关信息，具体见代码清单: 信号量-12 加粗部分。

列表 12: 代码清单: 信号量-12 二值信号量同步实验

```c
#include <includes.h>

/***************************************************************************/
/* LOCAL DEFINES */
/***************************************************************************/

OS_SEM SemOfKey;  //标志 KEY1 是否被按下的信号量

/***************************************************************************/
/* TCB */
/***************************************************************************/

static OS_TCB AppTaskStartTCB;  //任务控制块
static OS_TCB AppTaskKeyTCB;
static OS_TCB AppTaskLed1TCB;
```

(下页继续)
 /**************************************************************************
 * STACKS
**************************************************************************/

static CPU_STK AppTaskStartStk[APP_TASK_START_STK_SIZE];  //任务栈
static CPU_STK AppTaskKeyStk [ APP_TASK_KEY_STK_SIZE ];
static CPU_STK AppTaskLed1Stk [ APP_TASK_LED1_STK_SIZE ];

 /**************************************************************************
 * FUNCTION PROTOTYPES
**************************************************************************/

static void  AppTaskStart ( void *p_arg);  //任务函数声明
static void  AppTaskKey ( void * p_arg);
static void  AppTaskLed1 ( void * p_arg);

 /**************************************************************************
 * main()
**************************************************************************/

int main (void)
{ 
    OS_ERR err;
    OSInit(&err);
    //初始化μC/OS-III

    /* 创建起始任务 */
    OSTaskCreate((OS_TCB *) &AppTaskStartTCB,
        //任务控制块地址
        (CPU_CHAR *) "App Task Start",
        //任务名称
        (OS_TASK_PTR) AppTaskStart,
        //任务函数
        (void *) 0,
        //传递给任务函数（形参 p_arg）的实参
        (OS_PRIO) APP_TASK_START_PRIO,
        //任务的优先级
        (CPU_STK *) &AppTaskStartStk[0],
        //任务栈的基地址
        (CPU_STK_SIZE) APP_TASK_START_STK_SIZE / 10,
        //任务栈空间剩下 1/10 时限制其增长
        (CPU_STK_SIZE) APP_TASK_START_STK_SIZE,
        //任务栈空间（单位：sizeof(CPU_STK)
        (OS_MSG_QTY) 5u,
        //任务可接收的最大消息数
        (OS_TICK) 0u,
        //任务的时间片节拍数 (0 表默认值 OSCfg_TickRate_Hz/10)
        (void *) 0,
        //任务扩展 (0 表不扩展)
        (OS_OPT) (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
        //任务选项
        (OS_ERR *) &err);
    //返回错误类型
}

81    OSStart(&err);
82    //启动多任务管理（交由μC/OS-III控制）
83
84 {

85 static void AppTaskStart (void *p_arg)
86 {
87     CPU_INT32U cpu_clk_freq;
88     CPU_INT32U cnts;
89     OS_ERR err;
90
91     (void)p_arg;
92
93     BSP_Init();
94     //板级初始化
95     CPU_Init();
96     //初始化CPU组件（时间戳、中断时间测量和主机名）
97
98     cpu_clk_freq = BSP_CPU_ClkFreq();
99     //获取CPU内核时钟频率（SysTick工作时钟）
100    cnts = cpu_clk_freq / (CPU_INT32U)OS Cf g_TickRate_Hz;
101    //根据用户设定的时钟节拍频率计算SysTick定时器的计数值
102    OS_CPU_SysTickInit(cnts);
103    //调用SysTick初始化函数，设置定时器计数值和启动定时器
104
105    Mem_Init();
106    //初始化内存管理组件（堆内存池和内存池表）
107
108    #if OS_CFG_STAT_TASK_EN > 0u
109    //如果启用（默认启用）了统计任务
110    (续上页)
OSStatTaskCPUUsageInit(&err);

//计算没有应用任务（只有空闲任务）运行时 CPU 的（最大）
#endif//容量（决定 OS_Stat_IdleCtrMax 的值，为后面计算 CPU 使用率使用）。
CPU_IntDisMeasMaxCurReset();
//复位（清零）当前最大关中断时间

/* 创建信号量 SemOfKey */
OSSemCreate((OS_SEM *)&SemOfKey, //指向信号量变量的指针
    (CPU_CHAR *)"SemOfKey", //信号量的名字
    (OS_SEM_CTR )0,
    //信号量这里是指示事件发生，所以赋值为 0，表示事件还没有发生
    (OS_ERR *)&err); //错误类型

/* 创建 AppTaskKey 任务 */
OSTaskCreate((OS_TCB *)&AppTaskKeyTCB,
    //任务控制块地址
    (CPU_CHAR *)"App Task Key",
    //任务名称
    (OS_TASK_PTR ) AppTaskKey,
    //任务函数
    (void *) 0,
    //传递给任务函数（形参 p_arg）的实参
    (OS_PRIO ) APP_TASK_KEY_PRIO,
    //任务的优先级
    (CPU_STK *)&AppTaskKeyStk[0],
    //任务栈的基地址
    (CPU_STK_SIZE) APP_TASK_KEY_STK_SIZE / 10,
    //任务栈空间剩下 1/10 时限制其增长
    (CPU_STK_SIZE) APP_TASK_KEY_STK_SIZE,
    //任务栈空间（单位：sizeof(CPU_STK)
    (OS_MSG_QTY ) 5u,
// 任务可接收的最大消息数
(OS_TICK *) 0u,

// 任务的时间片节拍数（0 表默认值 OSCfg_TickRate_Hz/10）
(void *) 0,

// 任务扩展（0 表不扩展）
(OS_OPT *) (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),

// 任务选项
(OS_ERR *) &err);

/* 创建 LED1 任务 */
OSTaskCreate((OS_TCB *) &AppTaskLed1TCB,

// 任务控制块地址
(CPU_CHAR *) "App Task Led1",

// 任务名称
(OS_TASK_PTR) AppTaskLed1,

// 任务函数
(void *) 0,

// 传递给任务函数（形参 p_arg）的实参
(OS_PRIO) APP_TASK_LED1_PRIO,

// 任务的优先级
(CPU_STK *) &AppTaskLed1Stk[0],

// 任务栈的基地址
(CPU_STK_SIZE) APP_TASK_LED1_STK_SIZE / 10,

// 任务栈空间剩下 1/10 时限制其增长
(CPU_STK_SIZE) APP_TASK_LED1_STK_SIZE,

// 任务栈空间（单位: sizeof(CPU_STK))
(OS_MSG_QTY) 5u,

// 任务可接收的最大消息数
(OS_TICK) 0u,

// 任务的时间片节拍数（0 表默认值 OSCfg_TickRate_Hz/10）
(void *) 0,
//任务扩展 (0 表不扩展)
(OS_OPT ) (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
//任务选项
(OS_ERR *) &err);
//返回错误类型

OSTaskDel (& AppTaskStartTCB, & err);
//删除起始任务本身，该任务不再运行

/************************************************************************
* KEY TASK
************************************************************************/
static void AppTaskKey ( void * p_arg )
{
    OS_ERR err;
    uint8_t ucKey1Press = 0;
    (void)p_arg;

    while (DEF_TRUE)
    //任务体
    {
        if ( Key_Scan ( macKEY1_GPIO_PORT, macKEY1_GPIO_PIN, 1, &
                        ucKey1Press ) )
            //如果 KEY1 被按下
            OSSemPost((OS_SEM *) &SemOfKey,
                        //发布 SemOfKey
                        (OS_OPT ) OS_OPT_POST_ALL,
                        //发布给所有等待任务
                        (OS_ERR *) &err);
                            (下页继续)
// 返回错误类型

OSTimeDlyHMSM ( 0, 0, 0, 20, OS_OPT_TIME_DLY, & err );
// 每 20ms 扫描一次

}

/************************************************************************
* LED1 TASK
*************************************************************************/

static void AppTaskLed1 ( void * p_arg )
{
    OS_ERR err;
    CPU_INT32U cpu_clk_freq;
    CPU_TS ts_sem_post, ts_sem_get;
    CPU_SR_ALLOC();
    // 使用到临界段（开中断时）时必须用到该宏，该宏声明和定义一个局部变量。
    // 用于保存关中断前的 CPU 状态寄存器 SR（临界段关中断只保存 SR）开中断时将该值还原。
    (void)p_arg;

    cpu_clk_freq = BSP_CPU_ClkFreq();
    // 获取 CPU 时钟，时间戳是以该时钟计数

    while (DEF_TRUE)
    // 任务体
    {
        OSSemPend ((OS_SEM *) & SemOfKey,

//等待该信号量被发布
(OS_TICK  ) 0,
//无期限等待
(OS_OPT ) OS_OPT_PEND_BLOCKING,
//如果没有信号量可用就等待
(CPU_TS *) &ts_sem_post,
//获取信号量最后一次被发布的时间戳
(OS_ERR *) &err);
//返回错误类型

   ts_sem_get = OS_TS_GET();
   //获取解除等待时的时间戳
   
   macLED1_TOGGLE();
   //切换 LED1 的亮灭状态

   OS_CRITICAL_ENTER();
   //进入临界段，不希望下面串口打印遭到中断

   printf ( "\n发布信号量的时间戳是%d", ts_sem_post );
   printf ( "\n解除等待状态的时间戳是%d", ts_sem_get );
   printf ( "\n接收到信号量与发布信号量的时间相差%dus\n", ( ts_sem_get -
                  ts_sem_post ) / ( cpu_clk_freq / 1000000 ) );

   OS_CRITICAL_EXIT();

}
23.8.2 计数信号量实验

计数型信号量实验是模拟停车场工作运行。在创建信号量的时候初始化 5 个可用的信号量，并且创建了两个任务：一个是获取信号量任务，一个是释放信号量任务，两个任务独立运行，获取信号量任务是通过按下 KEY1 按键进行信号量的获取，模拟停车场停车操作，其等待时间是 0，在串口调试助手输出相应信息。

释放信号量任务则是信号量的释放，释放信号量任务也是通过按下 KEY2 按键进行信号量的释放，模拟停车场取车操作，在串口调试助手输出相应信息，实验源码具体见。

列表 13: 代码清单: 信号量-13 计数信号量实验

```c
#include <includes.h>

/*
 **********************************************
 *      LOCAL DEFINES
 **********************************************
 */

OS_SEM SemOfKey;    //标志 KEY1 是否被按下的信号量

/*
 **********************************************
 *         TCB
 **********************************************
 */

static OS_TCB AppTaskStartTCB;    //任务控制块
static OS_TCB AppTaskKey1TCB;
static OS_TCB AppTaskKey2TCB;
```

(下页继续)
*/

static CPU_STK AppTaskStartStk[APP_TASK_START_STK_SIZE]; // 任务栈
static CPU_STK AppTaskKey1Stk [ APP_TASK_KEY1_STK_SIZE ];
static CPU_STK AppTaskKey2Stk [ APP_TASK_KEY2_STK_SIZE ];

/*
 *****************************************
  FUNCTION PROTOTYPES
 *****************************************
*/

static void AppTaskStart (void *p_arg);   // 任务函数声明
static void AppTaskKey1 ( void * p_arg );
static void AppTaskKey2 ( void * p_arg );

int main (void)
{
    OS_ERR err;
    OSInit(&err); // 初始化 μC/OS-III

    /* 创建起始任务 */

    (下页继续)
OSTaskCreate((OS_TCB*)&AppTaskStartTCB,
    // 任务控制块地址
    (CPU_CHAR*)"App Task Start",
    // 任务名称
    (OS_TASK_PTR)AppTaskStart,
    // 任务函数
    (void*)0,
    // 传递给任务函数（形参 p_arg）的实参
    (OS_PRIO)APP_TASK_START_PRIO,
    // 任务的优先级
    (CPU_STK*)&AppTaskStartStk[0],
    // 任务栈的基地址
    (CPU_STK_SIZE)APP_TASK_START_STK_SIZE / 10,
    // 任务栈空间剩下 1/10 时限制其增长
    (CPU_STK_SIZE)APP_TASK_START_STK_SIZE,
    // 任务栈空间 (单位: sizeof(CPU_STK))
    (OS_MSG_QTY)5u,
    // 任务可接收的最大消息数
    (OS_TICK)0u,
    // 任务的时间片节拍数 (0 表默认值 OSCfg_TickRate_Hz/10)
    (void*)0,
    // 任务扩展 (0 表不扩展)
    (OS_OPT)(OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
    // 任务选项
    (OS_ERR*)&err);
    // 返回错误类型

OSStart(&err);
    // 启动多任务管理（交由 μC/OS-III 控制）


```c
static void AppTaskStart (void *p_arg)
{
    CPU_INT32U cpu_clk_freq;
    CPU_INT32U cnts;
    OS_ERR err;

    (void)p_arg;

    BSP_Init();
    //板级初始化
    CPU_Init(); //初始化CPU组件（时间戳、中断时间测量和主机名）

    cpu_clk_freq = BSP_CPU_ClkFreq();
    //获取CPU内核时钟频率（SysTick工作时钟）
    cnts = cpu_clk_freq / (CPU_INT32U)OSCfg_TickRate_Hz;
    //根据用户设定的时钟节拍频率计算SysTick定时器的计数值
    OS_CPU_SysTickInit(cnts); //调用SysTick初始化函数，设置定时器计数值和启动定时器

    Mem_Init(); //初始化内存管理组件（堆内存池和内存池表）

#if OS_CFG_STAT_TASK_EN > 0u
    //如果启用（默认启用）了统计任务
    OSStatTaskCPUUsageInit(&err);
    //计算没有应用任务（只有空闲任务）运行时CPU的（最大）
#endif
    //容量（决定OS_Stat_IdleCtrMax的值，为后面计算CPU使用率使用）。
```

论坛：https://www.firebbs.cn/  天猫：https://yehuosm.tmall.com
CPU_IntDisMeasMaxCurReset();
  //复位（清零）当前最大关中断时间

/* 创建信号量 SemOfKey */
OSSemCreate((OS_SEM *) &SemOfKey,    //指向信号量变量的指针
    (CPU_CHAR *) "SemOfKey",   //信号量的名字
    (OS_SEM_CTR *) 5,           //表示现有资源数目
    (OS_ERR *) &err);          //错误类型

/* 创建 AppTaskKey1 任务 */
OSTaskCreate((OS_TCB *) &AppTaskKey1TCB,
    //任务控制块地址
    (CPU_CHAR *) "App Task Key1",  //任务名称
    (OS_TASK_PTR ) AppTaskKey1,    //任务函数
    (void *) 0,        //传递给任务函数（形参 p_arg）的实参
    (OS_PRIO ) APP_TASK_KEY1_PRIO, //任务的优先级
    (CPU_STK *) &AppTaskKey1Stk[0], //任务栈的基地址
    (CPU_STK_SIZE) APP_TASK_KEY1_STK_SIZE / 10, //任务栈空间剩下 1/10 时限制其增长
    (CPU_STK_SIZE) APP_TASK_KEY1_STK_SIZE, //任务栈空间（单位：sizeof(CPU_STK)）
    (OS_MSG_QTY ) 5u, //任务可接收的最大消息数
    (OS_TICK ) 0u,    //任务的时间片节拍数（0 表默认值 OSCfg_TickRate_Hz/10）
(void *) 0,
//任务扩展 (0 表不扩展)
(OS_OPT ) (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
//任务选项
(OS_ERR * ) &err);
//返回错误类型

/* 创建 AppTaskKey2 任务 */
OSTaskCreate((OS_TCB *) &AppTaskKey2TCB,
//任务控制块地址
(CPU_CHAR *) "App Task Key2",
//任务名称
(OS_TASK_PTR ) AppTaskKey2,
//任务函数
(void *) 0,
//传递给任务函数（形参 p_arg）的实参
(OS_PRIO ) APP_TASK_KEY2_PRIO,
//任务的优先级
(CPU_STK * ) &AppTaskKey2Stk[0],
//任务栈的基地址
(CPU_STK_SIZE) APP_TASK_KEY2_STK_SIZE / 10,
//任务栈空间剩余 1/10 时限制其增长
(CPU_STK_SIZE) APP_TASK_KEY2_STK_SIZE,
//任务栈空间（单位：sizeof(CPU_STK)）
(OS_MSG_QTY ) 5u,
//任务可接收的最大消息数
(OS_TICK ) 0u,
//任务的时间片节拍数 (0 表默认值 OSCfg_TickRate_Hz/10)
(void *) 0,
//任务扩展 (0 表不扩展)
(OS_OPT ) (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
//任务选项
182 (OS_ERR *) &err);
// 返回错误类型
183
184 OSTaskDel ( & AppTaskStartTCB, & err );
// 删除起始任务本身，该任务不再运行
185
186
187
188
189
190
191
192 /*
193 ******************************************************************************
194 *
195 KEY1 TASK
196 ******************************************************************************
197 */
198 static void AppTaskKey1 ( void * p_arg )
199 {
200     OS_ERR err;
201     OS_SEM_CTR ctr;
202     CPU_SR_ALLOC();
203     // 使用到临界段（在关/开中断时）时必须用到该宏，该宏声明和定义一个局部变量，用于保存关中断前的 CPU 状态寄存器 SR（临界段关中断只需保存 SR），开中断时将该值还原。
204     uint8_t ucKey1Press = 0;
205
206     (void)p_arg;
207
208     while (DEF_TRUE)
209     // 任务体
210     {
211     
212     
213     (下页继续)
if ( Key_Scan ( macKEY1_GPIO_PORT, macKEY1_GPIO_PIN, 1, &ucKey1Press ) )
    // 如果 KEY1 被按下
    {
        ctr = OSSemPend ((OS_SEM *)&SemOfKey, // 等待该信号量 SemOfKey
                         (OS_TICK ) 0,
                         // 下面选择不等待，该参数无效
                         (OS_OPT ) OS_OPT_PEND_NON_BLOCKING,
                         // 如果没信号量可用不等待
                         (CPU_TS *) 0,          // 不获取时间戳
                         (OS_ERR *) &err);     // 返回错误类型

        OS_CRITICAL_ENTER();     // 进入临界段

        if ( err == OS_ERR_NONE )
            printf ( "\n\nKEY1 被按下：成功申请到停车位，剩余%d 个停车位。\n", ctr );
        else if ( err == OS_ERR_PEND_WOULD_BLOCK )
            printf ( "\n\nKEY1 被按下：不好意思，现在停车场已满，请等待！\n", err );

        OS_CRITICAL_EXIT();
    }

OSTimeDlyHMSM ( 0, 0, 0, 20, OS_OPT_TIME_DLY, & err );
static void AppTaskKey2 ( void * p_arg )
{
  OS_ERR err;
  OS_SEM_CTR ctr;
  CPU_SR_ALLOC();

  // 使用到临界段在开中断时必须用到该宏，该宏声明和定义一个局部变量，用于保存关中断前的 CPU 状态寄存器 SR（临界段关中断只需保存 SR）。
  // 开中断时将该值还原。
  uint8_t ucKey2Press = 0;

  (void)p_arg;

  while (DEF_TRUE)
  {
    if ( Key_Scan ( macKEY2_GPIO_PORT, macKEY2_GPIO_PIN, 1, &ucKey2Press ) )
      // 如果 KEY2 被按下
      {
        ctr = OSSemPost((OS_SEM *) SemOfKey,
                         (OS_OPT ) OS_OPT_POST_ALL,
                         (OS_ERR *) &err);
23.9 信号量实验现象

23.9.1 二值信号量同步实验现象

将程序编译好，用 USB 线连接计算机和开发板的 USB 接口（对应丝印为 USB 转串口），用 DAP 仿真器把配套程序下载到野火 STM32 开发板（具体型号根据购买的板子而定，每个型号的板子都配套有对应的程序），在计算机上打开串口调试助手，然后复位开发板，我们按下开发板的按键，串口打印任务运行的信息，表明两个任务同步成功，具体见图二值信号量同步实验现象。
23.9.2 计数信号量实验现象

将程序编译好，用 USB 线连接计算机和开发板的 USB 接口（对应丝印为 USB 转串口），用 DAP 仿真器将配套程序下载到野火 STM32 开发板（具体型号根据购买的板子而定，每个型号的板子都配套有对应的程序），在计算机上打开串口调试助手，然后复位开发板就可以在调试助手中看到串口的打印信息，按下开发板的 KEY1 按键获取信号量模拟停车，按下 KEY2 按键释放信号量模拟取车：我们按下 KEY1 与 KEY2 试试，在串口调试助手中可以看到运行结果，具体见图计数信号量实验现象。
野火多功能调试助手V1.3
技术论坛：www.fireBBS.cn 官网：www.embedFire.com

端口：COM3
波特率：115200
校验位：None
数据位：8
停止位：1

KEY1被按下：成功申请到停车位，剩下4个停车位。
KEY1被按下：成功申请到停车位，剩下3个停车位。
KEY1被按下：成功申请到停车位，剩下2个停车位。
KEY1被按下：成功申请到停车位，剩下1个停车位。
KEY1被按下：成功申请到停车位，剩下0个停车位。
KEY1被按下：不好意思，现在停车场已满，请等待！
KEY2被按下：释放1个停车位，剩下1个停车位。
KEY2被按下：释放1个停车位，剩下2个停车位。
KEY1被按下：成功申请到停车位，剩下1个停车位。
KEY1被按下：成功申请到停车位，剩下0个停车位。

论坛：https://www.firebbs.cn/ 493 天猫：https://yehuosm.tmall.com
第 24 章 互斥量

24.1 互斥量的基本概念

互斥量又称互斥信号量（本质也是一种信号量，不具备传递数据功能），是一种特殊的二值信号量，它和信号量不同的是，它支持互斥量所有权，递归访问及防止优先级翻转的特性，用于实现对临界资源的独占式处理。任意时刻互斥量的状态只有两种，开锁或闭锁。当互斥量被任务持有时，该互斥量处于闭锁状态，这个任务获得互斥量的所有权。当该任务释放这个互斥量时，该互斥量处于开锁状态，任务失去该互斥量的所有权。当一个任务持有互斥量时，其他任务将不能再对该互斥量进行开锁或持有。持有该互斥量的任务也能够再次获得该锁而不被挂起，这就是递归访问，也就是递归互斥量的特性。这个特性与一般的信号量有很大的不同，在信号量中，由于已经不存在可用的信号量，任务递归获取信号量时会发生主动挂起任务最终形成死锁。

如果想要用于实现同步（任务之间或者任务与中断之间），二值信号量或许是更好的选择，虽然互斥量也可以用于任务与任务间同步，但是互斥量更多的是用于保护资源的互锁。

用于互锁的互斥量可以充当保护资源的令牌，当一个任务希望访问某个资源时，它必须先获取令牌。当任务使用完资源后，必须还回令牌，以便其他任务可以访问该资源。是不是很熟悉，在我们的二值信号量里面也是一样的，用于保护临界资源，保证多任务的访问井然有序。当任务获取到信号量的时候才能开始使用被保护的资源，使用完就释放信号量，下一个任务才能获取到信号量从而可用使用被保护的资源。但是信号量会导致的另一个潜在问题，那就是任务优先级翻转（其具体在下文讲解）。而 μC/OS 提供的互斥量可以通过优先级继承算法，可以降低优先级翻转问题产生的影响，所以，用于临界资源的保护一般建议使用互斥量。

论坛：https://www.firebbs.cn        494        天猫：https://yehuosm.tmall.com
24.2 互斥量的优先级继承机制

在μC/OS 操作系统中为了降低优先级翻转问题利用了优先级继承算法。优先级继承算法是指，暂时提高某个占有某种资源的低优先级任务的优先级，使之与在所有等待该资源的任务中优先级最高那个任务的优先级相等，而当这个低优先级任务执行完毕释放该资源时，优先级重新回到初始设定值。因此，继承优先级的任务避免了系统资源被任何中间优先级的任务抢占。

互斥量与二值信号量最大的不同是：互斥量具有优先级继承机制，而信号量没有。也就是说，某个临界资源受到一个互斥量保护，如果这个资源正在被一个低优先级任务使用，那么此时的互斥量是闭锁状态，代表了没有任务能申请到这个互斥量，如果此时一个高优先级任务想要对这个资源进行访问，去申请这个互斥量，那么高优先级任务会因为申请不到互斥量而进入阻塞态；那么系统会将现在持有该互斥量的任务的优先级临时提升到与高优先级任务的优先级相同，这个优先级提升的过程叫作优先级继承。这个优先级继承机制确保高优先级任务进入阻塞状态的时间尽可能短，以及将已经出现的“优先级翻转”危害降低到最小。

没有理解？没问题，结合过程示意图再说一遍。我们知道任务的优先级在创建的时候就已经是设置好的，高优先级的任务可以打断低优先级的任务，抢占 CPU 的使用权。但是在很多场合中，某些资源只有一个，当低优先级任务正在占用该资源的时候，即使高优先级任务也只能乖乖的等待低优先级任务使用完该资源后释放资源。这里高优先级任务无法执行而低优先级任务可以运行的现象称为“优先级翻转”。

为什么说优先级翻转在操作系统中危害很大？因为在我们刚开始创建这个系统的时候，我们已经设置好了任务的优先级了，越重要的任务优先级越高。但是发生优先级翻转，对我们操作系统是致命的，会导致系统的高优先级任务阻塞时间过长。

举个例子，现在有 3 个任务分别为 H 任务 (High)、M 任务 (Middle)、L 任务 (Low)，3 个任务的优先级顺序为 H 任务 > M 任务 > L 任务。正常运行的时候 H 任务可以打断 M 任务与 L 任务，M 任务可以打断 L 任务，假设系统中有一个资源被保护了，此时该资源被 L 任务正在使用中，某一刻，H 任务需要使用该资源，但是 L 任务还没使用完，H 任务则因为申请不到资源而进入阻塞态，L 任务继续使用该资源，此时已经出现了“优先级翻转”现象，高优先级任务在等着低优先级的任务执行，如果在 L 任务执行的时候刚好 M 任务被唤醒了，由于 M 任务优先级比 L 任务优先级高，那么会打断 L 任务，抢占了 CPU 的使用权，直到 M 任务执行完，再把 CUP 使用权归还给 L 任务，L 任务继续执行，等到执行完毕之后释放该资源，H 任务此时才从阻塞态解除，使用
该资源。这个过程，本来是最高优先级的 H 任务，在等待了更低优先级的 L 任务与 M 任务，其阻塞的时间是 M 任务运行时间 +L 任务运行时间，这只是只有 3 个任务的系统，假如很多个这样的任务打断最低优先级的任务，那这个系统最高优先级任务岂不是崩溃了，这个现象是绝不允许出现的，高优先级的任务必须能及时响应。所以，没有优先级继承的情况下，使用资源保护，其危害极大，具体见图优先级翻转图解。

- 优先级翻转图解 (1): L 任务正在使用某临界资源，H 任务被唤醒，执行 H 任务。但 L 任务并未执行完毕，此时临界资源还未释放。

- 优先级翻转图解 (2): 这个时刻 H 任务也要对该临界资源进行访问，但 L 任务还未释放资源，由于保护机制，H 任务进入阻塞态，L 任务得以继续运行，此时已经发生了优先级翻转现象。

- 优先级翻转图解 (3): 某个时刻 M 任务被唤醒，由于 M 任务的优先级高于 L 任务，M 任务抢占了 CPU 的使用权，M 任务开始执行，此时 L 任务尚未执行完，临界资源还没被释放。

- 优先级翻转图解 (4): M 任务运行结束，归还 CPU 使用权，L 任务继续运行。

- 优先级翻转图解 (5): L 任务运行结束，释放临界资源，H 任务得以对资源进行访问，H 任务

论坛: [https://www.firebbs.cn/](https://www.firebbs.cn/)  天猫: [https://yehuosm.tmall.com](https://yehuosm.tmall.com)
务开始运行。

在这过程中，H 任务的等待时间过长，这对系统来说这是致命的，所以这种情况不允许出现，而互斥量就是用来降低优先级翻转的产生的危害。

假如有优先级继承呢？那么，在 H 任务申请该资源的时候，由于申请不到资源会进入阻塞态，那么系统就会把当前正在使用的任务的优先级临时提高到与 H 任务优先级相同，此时 M 任务被唤醒了，因为它的优先级比 H 任务低，所以无法打断 L 任务，因为此时 L 任务的优先级被临时提升到 H，所以当 L 任务使用完该资源了，进行释放，那么此时 H 任务优先级最高，将接着抢占 CPU 的使用权，H 任务的阻塞时间仅仅是 L 任务的执行时间，此时的优先级的危害降到了最低，看！这就是优先级继承的优势，具体见图优先级继承。

### 任务优先级

- **优先级继承 (1)**: L 任务正在使用某临界资源，L 任务正在使用某临界资源，H 任务被唤醒，执行 H 任务。但 L 任务并未执行完毕，此时临界资源还未释放。
- **优先级继承 (2)**: 某一时刻 H 任务也要对该资源进行访问，由于保护机制，H 任务进入阻塞态。此时发生优先级继承，系统将 L 任务的优先级暂时提升到与 H 任务优先级相同，L 任务继续执行。

论坛: [https://www.firebbs.cn/](https://www.firebbs.cn/)  
天猫: [https://yehuosm.tmall.com](https://yehuosm.tmall.com)
• 优先级继承 (3)：在某一时刻 M 任务被唤醒，由于此时 M 任务的优先级暂时低于 L 任务，所以 M 任务仅在就绪态，而无法获得 CPU 使用权。

• 优先级继承 (4)：L 任务运行完毕，H 任务获得对资源的访问权，H 任务从阻塞态变成运行态，此时 L 任务的优先级会变回原来的优先级。

• 优先级继承 (5)：当 H 任务运行完毕，M 任务得到 CPU 使用权，开始执行。

• 优先级继承 (6)：系统正常运行，按照设定好的优先级运行。

但是使用互斥量的时候一定需要注意：在获得互斥量后，请尽快释放互斥量，同时需要注意的是在任务持有互斥量的这段时间，不得更改任务的优先级。MC/OS 的优先级继承机制不能解决优先级翻转，只能将这种情况的影响降低到最小，硬实时系统在一开始设计时就要避免优先级翻转发生。

### 24.3 互斥量应用场景

互斥量的使用比较单一，因为它是一种信号量的一种，并且它是以锁的形式存在。在初始化的时候，互斥量处于开锁的状态，而被任务持有的时候则立刻转为闭锁的状态。互斥量更适合于：

• 可能会引起优先级翻转的情况。

• 任务可能会多次获取互斥量的情况下，这样可以避免同一任务多次递归持有而造成死锁的问题。

多任务环境下往往存在多个任务竞争同一临界资源的应用场景，互斥量可被用于对临界资源的保护从而实现独占式访问。另外，互斥量可以降低信号量存在的优先级翻转问题带来的影响。

比如有两个任务需要对串口进行发送数据，其硬件资源只有一个，那么两个任务肯定不能同时发送，不然导致数据错误，那么，就可以用互斥量对串口资源进行保护，当一个任务正在使用串口的时候，另一个任务则无法使用串口，等到任务使用串口完毕之后，另外一个任务才能获得串口的使用权。

另外需要注意的是互斥量不能在中断服务函数中使用，因为其特有的优先级继承机制只在任务起作用，而在中断的上下文环境中毫无意义。

论坛：https://www.firebbs.cn/ 498 天猫：https://yehuosm.tmall.com
24.4 互斥量运作机制

多任务环境下会存在多个任务访问同一临界资源的场景，该资源会被任务独占处理。其他任务在资源被占用的情况下不允许对该临界资源进行访问，这个时候就需要用到 μC/OS 的互斥量来进行资源保护，那么互斥量是怎样来避免这种冲突？

用互斥量处理不同任务对临界资源的同步访问时，任务想要获得互斥量才能进行资源访问，如果一旦有任务成功获得了互斥量，则互斥量立即变为闭锁状态，此时其他任务会因为获取不到互斥量而不能访问这个资源，任务会根据用户自定义的等待时间进行等待，直到互斥量被持有的任务释放后，其他任务才能获得互斥量从而得以访问该临界资源，此时互斥量再次上锁，如此一来就可以确保每个时刻只有一个任务正在访问这个临界资源，保证了临界资源操作的安全性。

- 互斥量运作机制 (1)：因为互斥量具有优先级继承机制，一般选择使用互斥量对资源进行保护，如果资源被占用的时候，无论是什么优先级的任务想要使用该资源都会被阻塞。

- 互斥量运作机制 (2)：假如正在使用该资源的任务 1 比阻塞中的任务 2 的优先级还低，那么任务 1 将被系统临时提升到与高优先级任务 2 相等的优先级 (任务 1 的优先级从 L 变成 H)。

论坛：https://www.firebbs.cn/  499  天猫：https://yehuosm.tmall.com
• 互斥量运作机制(3)：当任务1使用完资源之后，释放互斥量，此时任务1的优先级会从H变回原来的L。

• 互斥量运作机制(4)-(5)：任务2此时可以获得互斥量，然后进行资源的访问，当任务2访问了资源的时候，该互斥量的状态又为闭锁状态，其他任务无法获取互斥量。

24.5 互斥量控制块

μC/OS的互斥量由多个元素组成，在互斥量被创建时，需要由我们自己定义互斥量（也可以称之为互斥量句柄），因为它是用于保存互斥量的一些信息的，其数据结构OS_MUTEX除了互斥量必须的一些基本信息外，还有指向任务控制块的指针OwnerTCBPtr、任务优先级变量OwnerOriginalPrio、PendList链表与OwnerNestingCtr变量等，为的是方便系统来管理互斥量。其数据结构具体见代码清单:互斥量-1，示意图具体见图互斥量的控制块数据结构。
列表 1: 代码清单: 互斥量-1 互斥量控制块数据结构

```c
struct os_mutex
{
    /* ------------------ GENERIC MEMBERS ------------------ */
    OS_OBJ_TYPE Type; (1)
    CPU_CHAR *NamePtr; (2)
    OS_PEND_LIST PendList; (3)
}
```

(下页继续)
```c
#if OS_CFG_DBG_EN > 0u
    OS_MUTEX     *DbgPrevPtr;
    OS_MUTEX     *DbgNextPtr;
    CPU_CHAR     *DbgNamePtr;
#endif

/* ------------------ SPECIFIC MEMBERS ------------------ */
OS_TCB        *OwnerTCBPtr;  (4)
OS_PRIO       OwnerOriginalPrio;  (5)
OS_NESTING_CTR OwnerNestingCtr;  (6)
CPU_TS        TS;  (7)
};
```

- **代码清单: 互斥量-I (1):** 互斥量的类型，用户无需理会，μC/OS 能识别它是一个 mutex。
- **代码清单: 互斥量-I (2):** 互斥量的名字，每个内核对象都被分配一个名。
- **代码清单: 互斥量-I (3):** 等待互斥量的任务列表。
- **代码清单: 互斥量-I (4):** 指向持有互斥量任务控制块的指针，如果任务占用这个 mutex，那么该变量 OwnerTCBPtr 会指向占用这个 mutex 的任务的 OS_TCB。
- **代码清单: 互斥量-I (5):** 用于记录持有互斥量任务的优先级，如果任务占用这个 mutex，那么该变量 OwnerOriginalPrio 中存放着任务的原优先级，当占用 mutex 任务的优先级被提升时就会用到这个变量。
- **代码清单: 互斥量-I (6):** 表示互斥量是否可用，当该值为 0 的时候表示互斥量处于开锁状态，互斥量可用。μC/OS 允许任务递归调用同一个 mutex 多达 256 次，每递归调用一次 mutex 该值就会加一，但也需要释放相同次数才能真正释放掉这个 mutex。
- **代码清单: 互斥量-I (7):** mutex 中的变量 TS 用于保存该 mutex 最后一次被释放的时间戳。当 mutex 被释放，读取时基计数值并存放到该变量中。

注意：用户代码不能直接访问这个结构体，必须通过 μC/OS 提供的 API 访问。
24.6 互斥量函数接口讲解

24.6.1 创建互斥量函数 OSMutexCreate()

在定义完互斥量结构体变量后就可以调用 OSMutexCreate() 函数进行创建一个互斥量，跟信号量的创建差不多，我们知道，其实这里的“创建互斥量”指的对内核对象（互斥量）的一些初始化。要特别注意的是内核对象使用之前一定要先创建，这个创建过程必须要保证在所有可能使用内核对象的任务之前，所以我们都是在创建任务之前就创建好系统需要的内核对象（如互斥量等），创建互斥量函数 OSMutexCreate() 源码具体见代码清单: 互斥量-2。

列表 2: 代码清单: 互斥量-2 OSMutexCreate() 源码

```c
void OSMutexCreate (OS_MUTEX *p_mutex, (1) //互斥量指针
    CPU_CHAR *p_name, (2) //取互斥量的名称
    OS_ERR *p_err) (3) //返回错误类型
{
    CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必须用到该宏，该宏声明和
    //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
    // SR（临界段关中断只保存 SR），开中断时将该值还原。

    #ifdef OS_SAFETY_CRITICAL(4) //如果启用（默认禁用）了安全检测
        if (p_err == (OS_ERR *)) 0) //如果错误类型实参为空
            {
                OS_SAFETY_CRITICAL_EXCEPTION(); //执行安全检测异常函数
                return; //返回，不继续执行
            }
    #endif

    #ifdef OS_SAFETY_CRITICAL_IEC61508(5) //如果启用（默认禁用）了安全关键
        //如果是在调用 OSSafetyCriticalStart() 后创建
        if (OSSafetyCriticalStartFlag == DEF_TRUE)
            {
```
*p_err = OS_ERR_ILLEGAL_CREATE_RUN_TIME;  //错误类型为“非法创建内核对象”
return;
}
#endif

#if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u (6)
  //如果启用（默认启用）了中断中非法调用检测
  if (OSIntNestingCtr > (OS_NESTING_CTR)0)  //如果该函数是在中断中被调用
  {
    *p_err = OS_ERR_CREATE_ISR;  //错误类型为“在中断函数中定时”
    return;  //返回，不继续执行
  }
#endif

#if OS_CFG_ARG_CHK_EN > 0u(7) //如果启用（默认启用）了参数检测
  if (p_mutex == (OS_MUTEX *)0)  //如果参数 p_mutex 为空
  {
    *p_err = OS_ERR_OBJ_PTR_NULL;  //错误类型为“创建对象为空”
    return;  //返回，不继续执行
  }
#endif

OS_CRITICAL_ENTER();  //进入临界段，初始化互斥量指标

//标记创建对象数据结构为互斥量
p_mutex->Type = OS_OBJ_TYPE_MUTEX;  (8)
p_mutex->NamePtr = p_name;  (9)
p_mutex->OwnerTCBPtr = (OS_TCB *)0;  (10)
p_mutex->OwnerNestingCtr = (OS_NESTING_CTR)0;  (11)
p_mutex->TS = (CPU_TS )0;  (12)
p_mutex->OwnerOriginalPrio = OS_CFG_PRIO_MAX;
OS_PendListInit(&p_mutex->PendList);  //初始化该互斥量的等待列表

(下页继续)
```c
#if OS_CFG_DBG_EN > 0u
   //如果启用（默认启用）了调试代码和变量
   OS_MutexDbgListAdd(p_mutex); //将该互斥量添加到互斥量双向调试链表
#endif

OSMutexQty++; 
   //互斥量个数加1

OS_CRITICAL_EXIT_NO_SCHED();
   //退出临界段 (无调度)
*p_err = OS_ERR_NONE;
   //错误类型为 “无错误”

```

- **代码清单: 互斥量-2 (1):** 互斥量控制块指针，指向我们定义的互斥量控制块结构体变量，所以在创建之前我们需要先定义一个互斥量控制块变量。

- **代码清单: 互斥量-2 (2):** 互斥量名称，字符串形式。

- **代码清单: 互斥量-2 (3):** 用于保存返回的错误类型。

- **代码清单: 互斥量-2 (4):** 如果启用了安全检测（默认禁用），在编译时则会包含安全检测相关的代码，如果错误类型实参为空，系统会执行安全检测异常函数，然后返回，不执行创建互斥量操作。

- **代码清单: 互斥量-2 (5):** 如果启用（默认禁用）了安全关键检测，在编译时则会包含安全关键检测相关的代码，如果是在调用 OSSafetyCriticalStart() 后创建该互斥量，则是非法的，返回错误类型为“非法创建内核对象”错误代码，并且退出，不执行创建互斥量操作。

- **代码清单: 互斥量-2 (6):** 如果启用了中断中非法调用检测（默认启用），在编译时则会包含中断非法调用检测相关的代码，如果该函数是在中断中被调用，则是非法的，返回错误类型为“在中断中创建对象” 的错误代码，并且退出，不执行创建互斥量操作。

- **代码清单: 互斥量-2 (7):** 如果启用了参数检测（默认启用），在编译时则会包含参数检测相关的代码，如果 p_mutex 参数为空，返回错误类型为“创建对象为空”的错误代码，并且退出，不执行创建互斥量操作。

- **代码清单: 互斥量-2 (8):** 标记创建对象数据结构为互斥量。

- **代码清单: 互斥量-2 (9):** 初始化互斥量的名称。
• 代码清单: 互斥量-2 (10): 初始化互斥量结构体中的 OwnerTCBPtr 成员变量，目前系统中尚未任务持有互斥量。

• 代码清单: 互斥量-2 (11): 初始化互斥量结构体中的 OwnerNestingCtr 成员变量为 0，表示互斥量可用。

• 代码清单: 互斥量-2 (12): 记录时间戳的变量 TS 初始化为 0。初始化互斥量结构体中的 OwnerOriginalPrio 成员变量为 OS_CFG_PRIO_MAX（最低优先级）。初始化该互斥量的等待列表等。

• 代码清单: 互斥量-2 (13): 系统中互斥量个数加 1。

• 代码清单: 互斥量-2 (14): 退出临界段（无调度），创建互斥量成功。

如果我们创建一个互斥量，那么互斥量创建成功的示意图具体见图互斥量创建完成示意图。
互斥量创建函数的使用实例具体见代码清单: 互斥量-3。
列表 3: 代码清单: 互斥量-3OSMutexCreate() 使用实例

```c
OS_MUTEX mutex;                           //声明互斥量
/* 创建互斥量 mutex */
OSMutexCreate ((OS_MUTEX *)&mutex,       //指向互斥量变量的指针
               (CPU_CHAR *)"Mutex For Test"); //互斥量的名字
(OS_ERR    *)&err);                      //错误类型
```

### 24.6.2 删除互斥量函数 OSMutexDel()

OSSemDel() 用于删除一个互斥量，互斥量删除函数是根据互斥量结构（互斥量句柄）直接删除的，删除之后这个互斥量的所有信息都会被系统清空，而且不能再次使用这个互斥量了，但是需要注意的是，如果某个互斥量没有被定义，那也是无法被删除的，如果有任务阻塞在该互斥量上，那么尽量不要删除该互斥量。想要使用互斥量删除函数就必须将 OS_CFG_MUTEX_DEL_EN 宏定义配置为 1，其函数源码具体见 代码清单: 互斥量-4。

列表 4: 代码清单: 互斥量-4 OSMutexDel() 源码

```c
#if OS_CFG_MUTEX_DEL_EN > 0u                  //如果启用了 OSMutexDel()
OS_OBJ_QTY OSMutexDel (OS_MUTEX *p_mutex, (1) //互斥量指针
                       OS_OPT opt, (2)    //选项
                       OS_ERR *p_err) (3) //返回错误类型
{
    OS_OBJ_QTY cnt;
    OS_OBJ_QTY nbr_tasks;
    OS_PEND_DATA *p_pend_data;
    OS_PEND_LIST *p_pend_list;
    OS_TCB *p_tcb;
    OS_TCB *p_tcb_owner;
    CPU_TS ts;
    CPU_SRALLOC(); //使用到临界段（在关/开中断时）时必须用到该宏，该宏声明和
```
// 定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
// SR (临界段关中断只需保存 SR)，开中断时将该值还原。

#ifdef OS_SAFETY_CRITICAL(4)
// 如果启用（默认禁用）了安全检测
if (p_err == (OS_ERR *)0) // 如果错误类型实参为空
{
    OS_SAFETY_CRITICAL_EXCEPTION(); // 执行安全检测异常函数
    return ((OS_OBJ_QTY)0); // 返回 0 (有错误)，停止执行
}
#endif

#if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u(5)
// 如果启用了中断中非法调用检测
if (OSIntNestingCtr > (OS_NESTING_CTR)0) // 如果该函数在中断中被调用
{
    *p_err = OS_ERR_DEL_ISR; // 错误类型为“在中断中中止等待”
    return ((OS_OBJ_QTY)0); // 返回 0 (有错误)，停止执行
}
#endif

#if OS_CFG_ARG_CHK_EN > 0u(6)
// 如果启用了参数检测
if (p_mutex == (OS_MUTEX *)0) // 如果 p_mutex 为空
{
    *p_err = OS_ERR_OBJ_PTR_NULL; // 错误类型为“对象为空”
    return ((OS_OBJ_QTY)0); // 返回 0 (有错误)，停止执行
}
#endif

switch (opt) (7) // 根据选项分类处理
{
    case OS_OPT_DEL_NO_PEND: // 如果选项在预期内
    case OS_OPT_DEL_ALWAYS:
        break; // 直接跳出

    default: (8) // 如果选项超出预期
*p_err = OS_ERR_OPT_INVALID; //错误类型为“选项非法”
return ((OS_OBJ_QTY)0);  //返回 0 (有错误), 停止执行
}
#endif

#if OS_CFG_OBJ_TYPE_CHK_EN > 0u(9)//如果启用了对象类型检测
if (p_mutex->Type != OS_OBJ_TYPE_MUTEX)  //如果 p_mutex 非互斥量类型
{
    *p_err = OS_ERR_OBJ_TYPE;  //错误类型为“对象类型错误”
    return ((OS_OBJ_QTY)0);  //返回 0 (有错误), 停止执行
}
#endif

OS_CRITICAL_ENTER(); //进入临界段
p_pend_list = &p_mutex->PendList; (10)//获取互斥量的等待列表
cnt = p_pend_list->NbrEntries; (11)//获取等待该互斥量的任务数
nbr_tasks = cnt;
switch (opt) (12)//根据选项分类处理
{
    case OS_OPT_DEL_NO_PEND: (13)//如果只在没任务等待时删除互斥量
        if (nbr_tasks == (OS_OBJ_QTY)0)  //如果没有任务在等待该互斥量
        {
#if OS_CFG_DBG_EN > 0u//如果启用了调试代码和变量
            OS_MutexDbgListRemove(p_mutex); //将该互斥量从互斥量调试列表移除
#endif
            OSMutexQty--; (14)//互斥量数目减 1
            OS_MutexClr(p_mutex); (15)//清除互斥量内容
            OS_CRITICAL_EXIT();  //退出临界段
            *p_err = OS_ERR_NONE; (16)//错误类型为“无错误”
        }
        else(17)//如果其它任务在等待该互斥量
        {

(下页继续)
OS_CRITICAL_EXIT(); //退出临界段
*p_err = OS_ERR_TASK_WAITING; //错误类型为“有任务正在等待”
}
break; //跳出

case OS_OPT_DEL_ALWAYS: /*如果必须删除互斥量*/
p_tcb_owner = p_mutex->OwnerTCBPtr; //获取互斥量持有任务
if ((p_tcb_owner != (OS_TCB *))0) && //如果持有任务存在，
   (p_tcb_owner->Prio != p_mutex->OwnerOriginalPrio)) //而且优先级被提升过。
{
    switch (p_tcb_owner->TaskState) //根据其任务状态处理
    {
        case OS_TASK_STATE_RDY: //如果是就绪状态
            OS_RdyListRemove(p_tcb_owner); //将任务从就绪列表移除
            p_tcb_owner->Prio = p_mutex->OwnerOriginalPrio; (23) //还原任务的优先级
            OS_PrioInsert(p_tcb_owner->Prio); (24) //将该优先级插入优先级表格
            OS_RdyListInsertTail(p_tcb_owner); (25) //将任务重插入就绪列表
            break; //跳出
        case OS_TASK_STATE_DLY: //如果是延时状态
        case OS_TASK_STATE_SUSPENDED: //如果是被挂起状态
        case OS_TASK_STATE_DLY_SUSPENDED: //如果是延时中被挂起状态
            p_tcb_owner->Prio = p_mutex->OwnerOriginalPrio; //还原任务的优先级
            break;
        case OS_TASK_STATE_PEND: //如果是无限期等待状态
        case OS_TASK_STATE_PEND_TIMEOUT: //如果是有限期等待状态
            break;
    case OS_TASK_STATE_RDONLY: //如果是读取状态
            break;
    case OS_TASK_STATE_DLY_RDONLY: //如果是延时读取状态
            break;
    case OS_TASK_STATE_SUSPENDED_RDONLY: //如果是被挂起读取状态
            break;
    case OS_TASK_STATE_DLY_SUSPENDED_RDONLY: //如果是延时中被挂起读取状态
            break;
    case OS_TASK_STATE_PEND_RDONLY: //如果是无限期等待读取状态
        break;
    case OS_TASK_STATE_PEND_TIMEOUT_RDONLY: //如果是有限期等待读取状态
            break;
    }
}
case OS_TASK_STATE_PEND_SUSPENDED:
  //如果是无期等待中被挂状态

case OS_TASK_STATE_PEND_TIMEOUT_SUSPENDED:
  //如果是有期等待中被挂状态
  OS_PendListChangePrio(p_tcb_owner,
  //改变任务在等待列表的位置
  p_mutex->OwnerOriginalPrio);
  break;

  //如果状态超出预期
  OS_CRITICAL_EXIT();
  ^p_err = OS_ERR_STATE_INVALID;
  //错误类型为“任务状态非法”
  return ((OS_OBJ_QTY)0);
  //返回 0 (有错误), 停止执行

default:                     //28)
  OS_CRITICAL_EXIT();
  ^p_err = OS_ERR_STATE_INVALID;
  //错误类型为“任务状态非法”
  return ((OS_OBJ_QTY)0);
  //返回 0 (有错误), 停止执行

}{

  ts = OS_TS_GET();        //29)

  while(cnt > 0u)          //30)
    //移除该互斥量等待列表中的所有任务。
    {
      p_pend_data = p_pend_list->HeadPtr;
      p_tcb = p_pend_data->TCBPtr;
      OS_PendObjDel((OS_PEND_OBJ *)((void *)p_mutex),
      p_tcb,
      ts);          //31)
      cnt--;
    }

#if OS_CFG_DBG_EN > 0u //如果启用了调试代码和变量
  OS_MutexDbgListRemove(p_mutex); //将互斥量从互斥量调试列表移除
#endif
OSMutexQty--;  \( \text{(32)} \) //互斥量数目减 1
OS_MutexClr(p_mutex); \( \text{(33)} \) //清除互斥量内容
OS_CRITICAL_EXIT_NO_SCHED(); \( \text{(34)} \) //退出临界段，但不调度
OS_Sched(); \( \text{(35)} \) //调度最高优先级任务运行
*p_err = OS_ERR_NONE; \( \text{(36)} \) //错误类型为“无错误”
break;  //跳出
default:\( \text{(37)} \) //如果选项超出预期
    OS_CRITICAL_EXIT(); \( \text{//退出临界段} \)
    *p_err = OS_ERR_OPT_INVALID; \( \text{//错误类型为“选项非法”} \)
break;  //跳出
}

return (nbr_tasks); \( \text{(38)} \) //返回删除前互斥量等待列表中的任务数

• 代码清单: 互斥量-4 (1): 互斥量控制块指针，指向我们定义的互斥量控制块结构体变量，所以在删除之前我们需要先定义一个互斥量控制块变量，并且成功创建互斥量后再进行删除操作。

• 代码清单: 互斥量-4 (2): 互斥量删除的选项。

• 代码清单: 互斥量-4 (3): 用于保存返回的错误类型。

• 代码清单: 互斥量-4 (4): 如果启用了安全检测 (默认)，在编译时则会包含安全检测相关的代码，如果错误类型实参为空，系统会执行安全检测异常函数，然后返回，不执行删除互斥量操作。

• 代码清单: 互斥量-4 (5): 如果启用了中断中非法调用检测 (默认启用)，在编译时则会包含中断非法调用检测相关的代码，如果该函数是在中断中被调用，则是非法的，返回错误类型为 “在中断中删除对象” 的错误代码，并且退出，不执行删除互斥量操作。

• 代码清单: 互斥量-4 (6): 如果启用了参数检测 (默认启用)，在编译时则会包含参数检测相关的代码，如果 p_mutex 参数为空，返回错误类型为 “内核对象为空” 的错误代码，并且
退出，不执行删除互斥量操作。

- 代码清单: 互斥量-4 (7): 判断 opt 选项是否合理，该选项有两个，OS_OPT_DEL_ALWAYS
  与 OS_OPT_DEL_NO_PEND，在 os.h 文件中定义。此处是判断一下选项是否在预期之内，
  如果在则跳出 switch 语句。

- 代码清单: 互斥量-4 (8): 如果选项超出预期，则返回错误类型为“选项非法”的错误代码，
  退出，不继续执行。

- 代码清单: 互斥量-4 (9): 如果启用了对象类型检测，在编译时则会包含对象类型检测相关的
  代码，如果 p_mutex 不是互斥量类型，返回错误类型为“内核对象类型错误”的错误代码，
  并且退出，不执行删除互斥量操作。

- 代码清单: 互斥量-4 (10): 程序执行到这里，表示可以删除互斥量了，系统首先获取互斥量
  的等待列表保存到 p_pend_list 变量中。

- 代码清单: 互斥量-4 (11): 然后再获取等待该互斥量的任务数。

- 代码清单: 互斥量-4 (12): 根据选项分类处理。

- 代码清单: 互斥量-4 (13): 如果 opt 是 OS_OPT_DEL_NO_PEND，则表示只在没有任务等待
  的情况下删除互斥量，如果当前系统中有任务阻塞在该互斥量上，则不能删除，反之，则
  可以删除互斥量。

- 代码清单: 互斥量-4 (14): 如果没有任务在等待该互斥量，互斥量数目减 1。

- 代码清单: 互斥量-4 (15): 清除互斥量内容

- 代码清单: 互斥量-4 (16): 删除成功，返回错误类型为“无错误”的错误代码。

- 代码清单: 互斥量-4 (17): 如果有任务在等待该互斥量，则返回错误类型为“有任务在等待
  该互斥量”错误代码。

- 代码清单: 互斥量-4 (18): 如果 opt 是 OS_OPT_DEL_ALWAYS，则表示无论如何都必须删除
  互斥量，那么在删除之前，系统会把所有阻塞在该互斥量上的任务恢复。

- 代码清单: 互斥量-4 (19): 首先获取一下持有互斥量的任务。

- 代码清单: 互斥量-4 (20): 如果该互斥量被任务持有了，并且优先级也被提升了（发生优先
  级继承）。
• 代码清单: 互斥量-4 (21): 根据持有互斥量任务的状态进行分类处理。

• 代码清单: 互斥量-4 (22): 如果任务处于就绪状态。

• 代码清单: 互斥量-4 (23): 那么就将任务从就绪列表移除，然后还原任务的优先级，互斥量控制块中的 OwnerOriginalPrio 成员变量保存的就是持有互斥量任务的原本优先级。

• 代码清单: 互斥量-4 (24): 调用 OS_PrioInsert() 函数将任务按照其原本的优先级插入优先级列表中。

• 代码清单: 互斥量-4 (25): 将任务重新插入就绪列表。

• 代码清单: 互斥量-4 (26): 如果任务处于延时状态、被挂起状态或者是延时中被挂起状态，直接将任务的优先级恢复即可，并不用进行任务列表相关的操作。

• 代码清单: 互斥量-4 (27): 如果任务处于无限期等待状态、有限期等待状态、无期等待中被挂状态或者是有期等待中被挂状态，那么就调用 OS_PendListChangePrio() 函数改变任务在等待列表的位置，根据任务的优先级进行修改即可。

• 代码清单: 互斥量-4 (28): 如果状态超出预期，则返回错误类型为“任务状态非法”的错误代码。

• 代码清单: 互斥量-4 (29): 获取时间戳，记录一下删除的时间。

• 代码清单: 互斥量-4 (30): 然后根据前面 cnt 记录阻塞在该互斥量上的任务个数，逐个移除该互斥量等待列表中的任务。

• 代码清单: 互斥量-4 (31): 调用 OS_PendObjDel() 函数将阻塞在内核对象（如互斥量）上的任务从阻塞态恢复，此时系统在删除内核对象，删除之后，这些等待事件的任务需要被恢复，其源码具体见 代码清单: 消息队列-18。

• 代码清单: 互斥量-4 (32): 系统中互斥量数目减 1。

• 代码清单: 互斥量-4 (33): 清除互斥量中的内容。

• 代码清单: 互斥量-4 (34): 退出临界段，但不调度。

• 代码清单: 互斥量-4 (35): 调度最高优先级任务运行。

• 代码清单: 互斥量-4 (36): 删除互斥量完成，返回错误类型为“无错误”的错误代码。


- **代码清单: 互斥量-4 (37)**：如果选项超出预期则返回错误类型为“任务状态非法”的错误代码。

- **代码清单: 互斥量-4 (38)**：返回删除前互斥量等待列表中的任务数。

互斥量删除函数 OSMutexDel() 的使用也是简单的，只需要传入要删除的互斥量的句柄与选项还有保存返回的错误类型即可，调用函数时，系统将删除这个互斥量。需要注意的是在调用删除互斥量函数前，系统应存在已创建的互斥量。如果删除互斥量时，系统中有任务正在等待该互斥量，则不应该进行删除操作，因为删除之后的互斥量就不可用了。删除互斥量函数 OSMutexDel() 的使用实例具体见 代码清单: 互斥量-5。

列表 5: 代码清单: 互斥量-5 OSMutexDel() 函数使用实例

```c
OS_SEM mutex;         //声明互斥量
OS_ERR err;

/* 删除互斥量 mutex*/
OSMutexDel ((OS_MUTEX *)&mutex,             //指向互斥量的指针
OS_OPT_DEL_NO_PEND,
(OS_ERR *)&err);                //返回错误类型
```

### 24.6.3 获取互斥量函数 OSMutexPend()

我们知道，当互斥量处于开锁的状态，任务才能获取互斥量成功，当任务持有了某个互斥量的时候，其他任务就无法获取这个互斥量，需要等到持有互斥量的任务进行释放后，其他任务才能获取成功，任务通过互斥量获取函数来获取互斥量的所有权。任务对互斥量的所有权是独占的，任意时刻互斥量只能被一个任务持有，如果互斥量处于开锁状态，那么获取该互斥量的任务将成功获得该互斥量，并拥有互斥量的使用权；如果互斥量处于闭锁状态，获取该互斥量的任务将无法获得互斥量，任务将被挂起，在任务被挂起之前，会进行优先级继承。如果当前任务优先级比拥有互斥量的任务优先级高，那么将会临时提升持有互斥量任务的优先级。互斥量的获取函数就是 OSMutexPend()，其源码具体见 代码清单: 互斥量-6。
列表 6: 代码清单: 互斥量-OSMutexPend() 源码

```c
void OSMutexPend (OS_MUTEX *p_mutex, (1)  //互斥量指针
    OS_TICK timeout, (2)  //超时时间 (节拍)
    OS_OPT opt, (3)  //选项
    CPU_TS *p_ts, (4)  //时间戳
    OS_ERR *p_err) (5)  //返回错误类型
{
    OS_PEND_DATA pend_data;
    OS_TCB *p_tcb;
    CPU_SR_ALLOC();  //使用到临界段 (在关 / 开中断时) 时必须用到该宏，该宏声明和
    //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
    //SR (临界段关中断只需保存 SR)，开中断时将该值还原。

    #ifdef OS_SAFETY_CRITICAL  //如果启用（默认禁用）了安全检测
        if (p_err == (OS_ERR *)0)  //如果错误类型实参为空
        {
            OS_SAFETY_CRITICAL_EXCEPTION();  //执行安全检测异常函数
            return;  //返回，不继续执行
        }
    #endif

    #if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u  //如果启用了中断中非法调用检测
        if (OSIntNestingCtr > (OS_NESTING_CTR)0)  //如果该函数在中断中被调用
        {
            *p_err = OS_ERR_PEND_ISR;  //错误类型为 “在中断中等待”
            return;  //返回，不继续执行
        }
    #endif

    #if OS_CFG_ARG_CHK_EN > 0u  //如果启用了参数检测
        if (p_mutex == (OS_MUTEX *)0)  //如果 p_mutex 为空
        {
            OS_ARG_CHK_EXCEPTION();  //执行参数检测异常函数
            return;  //返回，不继续执行
        }
    #endif
```

(下页继续)


```c
31 {  
32     *p_err = OS_ERR_OBJ_PTR_NULL;  //返回错误类型为“内核对象为空”  
33     return;  //返回，不继续执行
34 }
35 switch (opt)  //根据选项分类处理
36 {
37 case OS_OPT_PEND_BLOCKING:  //如果选项在预期内
38     case OS_OPT_PEND_NON_BLOCKING:
39         break;
40     default:  //如果选项超出预期
41         *p_err = OS_ERR_OPT_INVALID;  //错误类型为“选项非法”
42         return;  //返回，不继续执行
43 }
44 #endif
45
46 #if OS_CFG_OBJ_TYPE_CHK_EN > 0u//如果启用了对象类型检测
47     if (p_mutex->Type != OS_OBJ_TYPE_MUTEX)  //如果 p_mutex 非互斥量类型
48         {  
49             *p_err = OS_ERR_OBJ_TYPE;  //错误类型为“内核对象类型错误”
50             return;  //返回，不继续执行
51         }
52 #endif
53
54     if (p_ts != (CPU_TS *)(CPU_TS *))  //如果 p_ts 非空
55         {  
56             *p_ts = (CPU_TS *)0;  //初始化（清零）p_ts，待用于返回时间戳
57         }
58
59     CPU_CRITICAL_ENTER();  //关中断
60     if (p_mutex->OwnerNestingCtr == (OS_NESTING_CTR)0) (6)  //如果互斥量可用
61     {
```

论坛: [https://www.firebbs.cn/](https://www.firebbs.cn/)  |  [https://yehuosm.tmall.com](https://yehuosm.tmall.com)
p_mutex->OwnerTCBPtr = OSTCBCurPtr;  //让当前任务持有互斥量
p_mutex->OwnerOriginalPrio = OSTCBCurPtr->Prio;  //保存持有任务的优先级
p_mutex->OwnerNestingCtr = (OS_NESTING_CTR)1;  //开始嵌套

if (p_ts != (CPU_TS *)0)
{
    *p_ts = p_mutex->TS;  //返回互斥量的时间戳记录
}
CPU_CRITICAL_EXIT();  //开中断
*p_err = OS_ERR_NONE;  //错误类型为“无错误”
return;

/* 如果互斥量不可用 */
if (OSTCBCurPtr == p_mutex->OwnerTCBPtr)  //如果当前任务已经持有该互斥量
{
    p_mutex->OwnerNestingCtr++;  //互斥量嵌套数加 1
    if (p_ts != (CPU_TS *)0)  //如果 p_ts 非空
    {
        *p_ts = p_mutex->TS;  //返回互斥量的时间戳记录
    }
    CPU_CRITICAL_EXIT();  //开中断
    *p_err = OS_ERR_MUTEX_OWNER;  //错误类型为“任务已持有互斥量”
    return;
}
/* 如果当前任务非持有该互斥量 */
if ((opt & OS_OPT_PEND_NON_BLOCKING) != (OS_OPT)0)  //如果选择了不阻塞任务
{
    CPU_CRITICAL_EXIT();  //开中断
    *p_err = OS_ERR_PEND_WOULD_BLOCK;  //错误类型为“渴求阻塞”
    return;
}
else  //如果选择了阻塞任务

(续上页)
{  
    if (OSSchedLockNestingCtr > (OS_NESTING_CTR)0) //如果调度器被锁
    {
        CPU_CRITICAL_EXIT();  //开中断
        *p_err = OS_ERR_SCHED_LOCKED;  //错误类型为“调度器被锁”
        return;  //返回，不继续执行
    }
}
/* 如果调度器未被锁 */
OS_CRITICAL_ENTER_CPU_EXIT();  //锁调度器，并重开中断
p_tcb = p_mutex->OwnerTCBPtr;  //获取互斥量持有任务
if (p_tcb->Prio > OSTCBCurPtr->Prio)  //如果持有任务优先级低于当前任务
{
    switch (p_tcb->TaskState)  //根据持有任务的任务状态分类处理
    {
    case OS_TASK_STATE_RDY:  //如果是就绪状态
        OS_RdyListRemove(p_tcb);  //从就绪列表移除持有任务
        p_tcb->Prio = OSTCBCurPtr->Prio;  //提升持有任务的优先级到当前任务
        OS_PrioInsert(p_tcb->Prio);  //将该优先级插入优先级表格
        OS_RdyListInsertHead(p_tcb);  //将持有任务插入就绪列表
        break;  //跳出

    case OS_TASK_STATE_DLY:  //如果是延时状态
    case OS_TASK_STATE_DLY_SUSPENDED:  //如果是延时中被挂起状态

    case OS_TASK_STATE_SUSPENDED:  //如果是被挂起状态
        p_tcb->Prio = OSTCBCurPtr->Prio;  //提升持有任务的优先级到当前任务
        break;  //跳出
case OS_TASK_STATE_PEND:      //如果是无期限等待状态
  case OS_TASK_STATE_PEND_TIMEOUT:    //如果是有期限等待状态
  case OS_TASK_STATE_PEND_SUSPENDED:  //如果是无期限等待中被挂起状态
    case OS_TASK_STATE_PEND_TIMEOUT_SUSPENDED:  //如果是有期限等待中被挂起状态
    OS_PendListChangePrio(p_tcb,       //改变持有任务在等待列表的位置
        OSTCBCurPtr->Prio); (23)
    break;                          //跳出

    default: (24) //如果任务状态超出预期
        OS_CRITICAL_EXIT();          //开中断
        *p_err = OS_ERR_STATE_INVALID;  //错误类型为“任务状态非法”
        return;                      //返回，不继续执行
  }
}
/*
阻塞任务，将当前任务脱离就绪列表，并插入节拍列表和等待列表。
*/
OS_Pend(&pend_data,
   (OS_PEND_OBJ *)((void *)p_mutex),
   OS_TASK_PEND_ON_MUTEX,
   timeout); (25)

OS_CRITICAL_EXIT_NO_SCHED();     //开调度器，但不进行调度

OSSched(); (26) //调度最高优先级任务运行

CPU_CRITICAL_ENTER();           //开中断
switch (OSTCBCurPtr->PendStatus) (27) //根据当前运行任务的等待状态分类处理
{
  case OS_STATUS_PEND_OK: (28) //如果等待正常（获得互斥量）
if (p_ts != (CPU_TS *)0)  //如果 p_ts 非空
{
    *p_ts = OSTCBCurPtr->TS;  //返回互斥量最后一次被释放的时间戳
}
*p_err = OS_ERR_NONE;  (29)  //错误类型为“无错误”
break;  //跳过

case OS_STATUS_PEND_ABORT:  (30)  //如果等待被中止
if (p_ts != (CPU_TS *)0)  //如果 p_ts 非空
{
    *p_ts = OSTCBCurPtr->TS;  //返回等待被中止时的时间戳
}
*p_err = OS_ERR_PEND_ABORT;  //错误类型为“等待被中止”
break;  //跳过

case OS_STATUS_PEND_TIMEOUT:  (31)  //如果超时内为获得互斥量
if (p_ts != (CPU_TS *)0)  //如果 p_ts 非空
{
    *p_ts = (CPU_TS )0;  //清零 p_ts
}
*p_err = OS_ERR_TIMEOUT;  //错误类型为“超时”
break;  //跳过

case OS_STATUS_PEND_DEL:  (32)  //如果互斥量已被删除
if (p_ts != (CPU_TS *)0)  //如果 p_ts 非空
{
    *p_ts = OSTCBCurPtr->TS;  //返回互斥量被删除时的时间戳
}
*p_err = OS_ERR_OBJ_DEL;  //错误类型为“对象被删除”
break;  //跳过

default:  (33)  //根据等待状态超时预期
*p_err = OS_ERR_STATUS_INVALID; //错误类型为“状态非法”
break; //跳出
}
CPU_CRITICAL_EXIT(); //开中断

- 代码清单: 互斥量-6 (1): 互斥量指针。
- 代码清单: 互斥量-6 (2): 用户自定义的阻塞超时时间，单位为系统时钟节拍。
- 代码清单: 互斥量-6 (3): 获取互斥量的选项，当互斥量不可用的时候，用户可以选择阻塞或者不阻塞。
- 代码清单: 互斥量-6 (4): 用于保存返回等到互斥量时的时间戳。
- 代码清单: 互斥量-6 (5): 用于保存返回的错误类型，用户可以根据此变量得知错误的原因。
- 代码清单: 互斥量-6 (6): 如果互斥量可用，互斥量控制块中的OwnerNestingCtr 变量为 0 则表示互斥量处于开锁状态，互斥量可用被任务获取。
- 代码清单: 互斥量-6 (7): 让当前任务持有互斥量。
- 代码清单: 互斥量-6 (8): 保存一下持有互斥量任务的优先级。如果发生了优先级继承，就会用到这个变量。
- 代码清单: 互斥量-6 (9): 开始嵌套，这其实是将互斥量变为闭锁状态，而其他任务就不能获取互斥量，但是本身持有互斥量的任务就拥有该互斥量的所有权，能递归获取该互斥量，每获取一次已经持有的互斥量，OwnerNestingCtr 的值就会加一，以表示互斥量嵌套，任务获取了多少次互斥量就需要释放多少次互斥量。
- 代码清单: 互斥量-6 (10): 保存并且返回互斥量的时间戳记录，记录错误类型为“无错误”，退出，不继续执行。
- 代码清单: 互斥量-6 (11): 而如果任务想要获取的互斥量处于闭锁状态 (OwnerNestingCtr 变量不为 0)，那么就判断一下当前任务是否已经持有该互斥量。
- 代码清单: 互斥量-6 (12): 如果当前任务已经持有该互斥量，那么任务就拥有互斥量的所有权，能递归获取互斥量，那么互斥量嵌套数就加 1。

论坛:  https://www.firebbs.cn/  
天猫:  https://yehuosm.tmall.com
• 代码清单: 互斥量-6 (13): 返回互斥量的时间戳记录与错误类型为“任务已持有互斥量”的错误代码，然后退出。

• 代码清单: 互斥量-6 (14): 如果当前任务并没有持有该互斥量，那肯定是不能获取到的，就看看用户有没有选过阻塞任务，如果选择了不阻塞任务，那么就返回错误类型为“渴求阻塞”的错误代码，退出，不继续执行。

• 代码清单: 互斥量-6 (15): 而用户如果选择了阻塞任务，就判断一下调度器是否被锁，如果调度器被锁了，就返回错误类型为“调度器被锁”的错误代码。

• 代码清单: 互斥量-6 (16): 如果调度器未被锁，就锁调度器，并重开中断，至于为什么，在前面的章节就讲解过了，此处就不再重复赘述，具体见 代码清单：信号量-13(20)。

• 代码清单: 互斥量-6 (17): 获取持有互斥量的任务，判断一下当前任务与持有互斥量的任务优先级情况，如果持有互斥量的任务优先级低于当前任务，就会临时将持有互斥量任务的优先级提升，提升到与当前任务优先级一致，这就是优先级继承。

• 代码清单: 互斥量-6 (18): 根据持有互斥量任务的任务状态分类处理。

• 代码清单: 互斥量-6 (19): 如果该任务处于就绪状态，那么从就绪列表中移除该任务，然后将该任务的优先级到与当前任务优先级一致。

• 代码清单: 互斥量-6 (20): 将该优先级插入优先级表格。

• 代码清单: 互斥量-6 (21): 再将该任务按照优先级顺序插入就绪列表。

• 代码清单: 互斥量-6 (22): 如果持有互斥量任务处于延时状态、延时中被挂起状态或者是被挂起状态，仅仅是提升持有互斥量任务的优先级与当前任务优先级一致即可，不需要操作就绪列表。

• 代码清单: 互斥量-6 (23): 如果持有互斥量任务无期限等状态、有期限等状态、无期限等状态中被挂起状态或者是有期限等状态中被挂起状态，那么就直接根据任务的优先级来改变持有互斥量任务在等待列表的位置即可。

• 代码清单: 互斥量-6 (24): 如果任务状态超出预期，返回错误类型为“任务状态非法”的错误代码，不继续执行。

• 代码清单: 互斥量-6 (25): 程序执行到这里，就表示如果需要优先级继承的就已经处理完毕了，否则就不用优先级继承，那么可以直接调用 OS_Pend() 函数阻塞任务，将当前任务脱
离就绪列表，并插入节拍列表和等待列表中。

- **代码清单: 互斥量-6 (26)**: 进行一次任务调度，以运行处于最高优先级的就绪任务。
- **代码清单: 互斥量-6 (27)**: 程序能执行到这里，表示任务已经从阻塞中恢复了，但是恢复的原因有多种，需要根据当前运行任务的等待状态分类处理。
- **代码清单: 互斥量-6 (28)**: 如果任务等待正常（获得了互斥量），这是最好的结果了，任务等到了互斥量。
- **代码清单: 互斥量-6 (29)**: 保存一下获取的时间戳与错误类型为“无错误”的错误代码，就跳出 switch 语句继续执行。
- **代码清单: 互斥量-6 (30)**: 如果等待被中止，返回等待被中止时的时间戳与错误类型为“等待被中止”的错误代码，跳出 switch 语句。
- **代码清单: 互斥量-6 (31)**: 如果超时时间内未获得互斥量，就返回错误类型为“阻塞超时”的错误代码，然后跳出 switch 语句。
- **代码清单: 互斥量-6 (32)**: 如果互斥量已被删除，返回互斥量被删除时的时间戳与错误类型为“对象被删除”的错误代码，跳出 switch 语句。
- **代码清单: 互斥量-6 (33)**: 根据等待状态超出预期，返回错误类型为“状态非法”的错误代码，退出。

至此，获取互斥量的操作就完成了，如果任务获取互斥量成功，那么在使用完毕需要立即释放，否则很容易造成其他任务无法获取互斥量，因为互斥量的优先级继承机制是只能将优先级危害降低，而不能完全消除。同时还需注意的是，互斥量是不允许在中断中操作的，因为互斥量特有的优先级继承机制在中断是毫无意义的，互斥量获取函数的使用实例具体见代码清单: 互斥量-7。

**列表 7: 代码清单: 互斥量-7OSMutexPend() 函数使用实例**

```
1  OS_MUTEX mutex;                //声明互斥量
2
3  OS_ERR err;
```

(下页继续)
24.6.4 释放互斥量函数 OSMutexPost()

任务想要访问某个资源的时候，需要先获取互斥量，然后进行资源访问，在任务使用完该资源的时候，必须要及时归还互斥量，这像别的任务才能对资源进行访问。在前面的讲解中，我们知道，当互斥量有效的的时候，任务才能获取互斥量，那么，是什么函数使得互斥量变得有效呢？μC/OS 给我们提供了互斥量释放函数 OSMutexPost()，任务可以调用该函数进行释放互斥量，表示我已经用完了，别人可以申请使用，但是要注意的是，互斥量的释放只能在任务中，不允许在中断中释放互斥量。

使用该函数接口时，只有已持有互斥量所有权的任务才能释放它，当任务调用 OSMutexPost() 函数时会释放一次互斥量，当互斥量的成员变量 OwnerNestingCtr 为 0 的时候，互斥量状态才会成为开锁状态，等待获取该互斥量的任务将被唤醒。如果任务的优先级被互斥量的优先级翻转机制临时提升，那么当互斥量被完全释放后，任务的优先级将恢复为原本设定的优先级，其源码具体见代码清单: 互斥量-8。

列表 8: 代码清单: 互斥量-8 OSMutexPost() 源码

```c
void OSMutexPost (OS_MUTEX *p_mutex, (1) //互斥量指针
  OS_OPT opt, (2) //选项
  OS_ERR *p_err) (3) //返回错误类型
{
  OS_PEND_LIST *p_pend_list;
  OS_TCB *p_tcb;
  CPU_TS ts;
```

CPU_SR_ALLOC();
//使用到临界段（在关/开中断时）时必须用到该宏，该宏声明和定义一个局部变
//量，用于保存关中断前的 CPU 状态寄存器 SR（临界段关中断只需保存 SR）
//，开中断时将该值还原。

#define OS_SAFETY_CRITICAL(4)//如果启用（默认禁用）了安全检测
  if (p_err == (OS_ERR *)0) //如果错误类型实参为空
  {
      OS_SAFETY_CRITICAL_EXCEPTION(); //执行安全检测异常函数
      return; //返回，不继续执行
  }
#endif

#if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u(5)//如果启用了中断中非法调用检测
  if (OSIntNestingCtr > (OS_NESTING_CTR)0) //如果该函数在中断中被调用
  {
      *p_err = OS_ERR_POST_ISR; //错误类型为“在中断中等待”
      return; //返回，不继续执行
  }
#endif

#if OS_CFG_ARG_CHK_EN > 0u(6)//如果启用了参数检测
  if (p_mutex == (OS_MUTEX *)0) //如果 p_mutex 为空
  {
      *p_err = OS_ERR_OBJ_PTR_NULL; //错误类型为“内核对象为空”
      return; //返回，不继续执行
  }
  switch (opt) //根据选项分类处理
  {
    case OS_OPT_POST_NONE: //如果选项在预期内，不处理
    case OS_OPT_POST_NO_SCHED:
      break;
  }
}
default:                      // 如果选项超出预期
*p_err = OS_ERR_OPT_INVALID;  // 错误类型为 "选项非法"
return;                      // 返回，不继续执行

#endif

#if OS_CFG_OBJ_TYPE_CHK_EN > 0u(7) // 如果启用了对象类型检测

if (p_mutex->Type != OS_OBJ_TYPE_MUTEX) // 如果 p_mutex 的类型不是互斥量类型
{
    *p_err = OS_ERR_OBJ_TYPE;       // 返回，不继续执行
    return;
}
#endif

CPU_CRITICAL_ENTER();         // 关中断
if (OSTCBCurPtr != p_mutex->OwnerTCBPtr) (8) // 如果当前运行任务不持有该互斥量
{
    CPU_CRITICAL_EXIT();          // 关中断
    *p_err = OS_ERR_MUTEX_NOT_OWNER; (9) // 错误类型为 "任务不持有该互斥量"
    return;
}

OS_CRITICAL_ENTER_CPU_EXIT();  // 锁调度器，关中断
ts = OS_TS_GET();             // (10) 获取时间戳
p_mutex->TS = ts;
// 存储互斥量最后一次被释放的时间戳
p_mutex->OwnerNestingCtr--;    // (11) 互斥量的嵌套数减 1
if (p_mutex->OwnerNestingCtr > (OS_NESTING_CTR)0) // 如果互斥量仍被嵌套
{
    OS_CRITICAL_EXIT();          // 解锁调度器
}
*p_err = OS_ERR_MUTEX_NESTING; \(12\) //错误类型为“互斥量被嵌套”
    return; \(12\) //返回，不继续执行
}
/* 如果互斥量未被嵌套，已可用 */
p_pend_list = &p_mutex->PendList; \(13\) //获取互斥量的等待列表
if (p_pend_list->NbrEntries == (OS_OBJ_QTY)0) \(13\) //如果没有任务在等待该互斥量
{
    p_mutex->OwnerTCBPtr = (OS_TCB *)0; \(14\) //清除互斥量持有者信息
    p_mutex->OwnerNestingCtr = (OS_NESTING_CTR)0; \(15\)
    OS_CRITICAL_EXIT(); \(16\) //解锁调度器
    *p_err = OS_ERR_NONE; \(16\) //错误类型为“无错误”
    return; \(16\) //返回，不继续执行
}
/* 如果有任务在等待该互斥量 */
if (OSTCBCurPtr->Prio != p_mutex->OwnerOriginalPrio) \(17\) //如果当前任务的优先级被改过
{
    OS_RdyListRemove(OSTCBCurPtr); \(18\) //从就绪列表移除当前任务
    OSTCBCurPtr->Prio = p_mutex->OwnerOriginalPrio; \(19\) //还原当前任务的优先级
    OS_PrioInsert (OSTCBCurPtr->Prio); \(20\) //在优先级表格插入这个优先级
    OS_RdyListInsertTail (OSTCBCurPtr); \(21\) //将当前任务插入就绪列表尾端
    OSPrioCur = OSTCBCurPtr->Prio; \(22\) //更改当前任务优先级变量的值
}
p_tcb = p_pend_list->HeadPtr->TCBPtr; \(23\) //获取等待列表的首端任务
p_mutex->OwnerTCBPtr = p_tcb; \(24\) //将互斥量交给该任务
p_mutex->OwnerOriginalPrio = p_tcb->Prio; \(25\)
\[p_mutex->OwnerNestingCtr = (OS_NESTING_CTR)1; \(26\) //开始嵌套
/* 释放互斥量给该任务 */
OS_Post((OS_PEND_OBJ *)((void *)p_mutex),

论坛：https://www.firebbs.cn/  
天猫：https://yehuosm.tmall.com
(OS_TCB  *) p_tcb,
(void  *) 0,
(OS_MSG_SIZE ) 0,
(CPU_TS   ) ts);   (27)

OS_CRITICAL_EXIT_NO_SCHED();   //取消调度，但不执行任务调度

if ((opt & OS_OPT_POST_NO_SCHED) == (OS_OPT)0)  //如果 opt 没选择 “发布时不调度任务”
{
    OSSched();   (28)   //任务调度
}

*p_err = OS_ERR_NONE;   //错误类型为 “无错误”

论坛：https://www.firebbs.cn/   530   天猫：https://yehuosm.tmall.com

- 代码清单: 互斥量-8 (1): 互斥量指针。
- 代码清单: 互斥量-8 (2): 释放互斥量的选项。
- 代码清单: 互斥量-8 (3): 用于保存返回的错误类型，用户可以根据此变量得知错误的原因。
- 代码清单: 互斥量-8 (4): 如果启用（默认禁用）了安全检测，在编译时则会包含安全检测相关的代码，如果错误类型实参为空，系统会执行安全检测异常函数，然后返回，停止执行。
- 代码清单: 互斥量-8 (5): 如果在中断中非法调用检测，并且如果该函数在中断中被调用，则返回错误类型为 “在中断中释放”的错误代码，然后退出不继续执行。消息、信号量等内核对象可以在中断中释放，但是唯一互斥量是不可以的，因为其具备的优先级继承特性在中断的上下文环境中毫无意义。
- 代码清单: 互斥量-8 (6): 如果启用参数检测，在编译时则会包含参数检测相关的代码，如果 p_mutex 参数为空，返回错误类型为 “内核对象为空”的错误代码，并且退出，不执行释放互斥量操作。
- 代码清单: 互斥量-8 (7): 如果启用对象类型检测，在编译时则会包含对象类型检测相关
代码，如果 p_mutex 不是互斥量类型，那么返回错误类型为“对象类型有误”的错误代码，并且退出，不执行释放互斥量操作。

- **代码清单: 互斥量-8 (8):** 程序能运行到这里，说明传递进来的参数是正确的，此时，系统会判断一下调用互斥量释放函数的任务是否持有该互斥量，如果是则进行互斥量的释放，否则就返回错误。

- **代码清单: 互斥量-8 (9):** 如果当前运行任务不持有该互斥量，返回错误类型为“任务不持有该互斥量”的错误代码，然后退出，不继续执行。

- **代码清单: 互斥量-8 (10):** 获取时间戳，保存一下互斥量最后一次被释放的时间戳。

- **代码清单: 互斥量-8 (11):** 互斥量控制块中的 OwnerNestingCtr 成员变量减一，也就是互斥量的嵌套数减 1，当该变量为 0 的时候，互斥量才变为开锁状态。

- **代码清单: 互斥量-8 (12):** 如果互斥量仍被嵌套，也就是 OwnerNestingCtr 不为 0，那还是表明当前任务还是持有互斥量的，并未完全释放，返回错误类型为“互斥量仍被嵌套”的错误代码，然后退出，不继续执行。

- **代码清单: 互斥量-8 (13):** 如果互斥量未被嵌套，已可用（OwnerNestingCtr 为 0），那么就获取互斥量的等待列表保存在 p_pend_list 变量中，通过该变量访问互斥量等待列表。

- **代码清单: 互斥量-8 (14):** 如果没有任务在等待该互斥量，那么就清空互斥量持有者信息，互斥量中的 OwnerTCBPtr 成员变量重置为 0。

- **代码清单: 互斥量-8 (15):** 互斥量中的 OwnerNestingCtr 成员变量重置为 0，表示互斥量处于开锁状态。

- **代码清单: 互斥量-8 (16):** 执行到这里，表示当前任务已经完全释放互斥量了，返回错误类型为“无错误”的错误代码。

- **代码清单: 互斥量-8 (17):** 如果有任务在等待该互斥量，那么就很有可能发生了优先级继承，先查看当前任务的优先级是否被修改过，如果有则说明发生了优先级继承，就需要重新恢复任务原本的优先级。

- **代码清单: 互斥量-8 (18):** 从就绪列表移除当前任务。

- **代码清单: 互斥量-8 (19):** 还原当前任务的优先级。
• 代码清单: 互斥量-8 (20): 在优先级表格插入这个优先级。
• 代码清单: 互斥量-8 (21): 将当前任务插入就绪列表尾端。
• 代码清单: 互斥量-8 (22): 更改当前任务优先级变量的值。
• 代码清单: 互斥量-8 (23): 获取等待列表的首端任务。
• 代码清单: 互斥量-8 (24): 将互斥量交给该任务。
• 代码清单: 互斥量-8 (25): 保存一下该任务的优先级。
• 代码清单: 互斥量-8 (26): 互斥量的 OwnerNestingCtr 成员变量设置为 1，表示互斥量处于闭锁状态。
• 代码清单: 互斥量-8 (27): 调用 OS_Post() 函数释放互斥量给该任务。
• 代码清单: 互斥量-8 (28): 进行一次任务调度。

已经获取到互斥量的任务拥有互斥量的所有权，能重复获取同一个互斥量，但是任务获取了多少次互斥量就要释放多少次互斥量才能彻底释放掉互斥量，互斥量的状态才会变成开锁状态，否则在此之前互斥量都处于无效状态，别的任务就无法获取该互斥量。使用该函数接口时，只有已持有互斥量所有权的任务才能释放它，每释放一次该互斥量，它的 OwnerNestingCtr 成员变量就减 1。当该互斥量的 OwnerNestingCtr 成员变量为 0 时（即持有任务已经释放所有的持有操作），互斥量则变为开锁状态，等待在该互斥量上的任务将被唤醒。如果任务的优先级被互斥量的优先级翻转机制临时提升，那么当互斥量被释放后，任务的优先级将恢复为原本设定的优先级，下面看看互斥量释放函数是如何使用的，具体见代码清单: 互斥量-9。
列表 9: 代码清单: 互斥量-9 OSMutexPost() 使用实例

```
OS_MUTEX mutex;  //声明互斥互斥量
OS_ERR err;
OSMutexPost ((OS_MUTEX *) &mutex,  //释放互斥互斥量 mutex
(OS_OPT ) OS_OPT_POST_NONE,  //进行任务调度
(OS_ERR *) &err);  //返回错误类型
```

### 24.7 实验

#### 24.7.1 模拟优先级翻转实验

模拟优先级翻转实验是在µC/OS 中创建了三个任务与一个二值信号量，任务分别是高优先级任务 AppTaskLed3，中优先级任务 AppTaskLed2，低优先级任务 AppTaskLed1，用于模拟产生优先级翻转。低优先级任务在获取信号量的时候，被中优先级打断，中优先级的任务开始执行，因为低优先级还未释放信号量，那么高优先级任务就无法取得信号量继续运行，此时就发生了优先级翻转，任务在运行中，使用串口打印出相关信息，具体见代码清单: 互斥量-10。

列表 10: 代码清单: 互斥量-10 模拟优先级翻转实验

```
#include <includes.h>

/*

******************************************************************************
* LOCAL DEFINES
******************************************************************************
*/

OS_SEM TestSem;  //信号量
```

(下页继续)
/*
 * TCB
 */

static OS_TCB AppTaskStartTCB;
static OS_TCB AppTaskLed1TCB;
static OS_TCB AppTaskLed2TCB;
static OS_TCB AppTaskLed3TCB;

/*
 * STACKS
 */

static CPU_STK AppTaskStartStk[APP_TASK_START_STK_SIZE];
static CPU_STK AppTaskLed1Stk[APP_TASK_LED1_STK_SIZE];
static CPU_STK AppTaskLed2Stk[APP_TASK_LED2_STK_SIZE];
static CPU_STK AppTaskLed3Stk[APP_TASK_LED3_STK_SIZE];

/*
 * FUNCTION PROTOTYPES
 */

static void AppTaskStart (void *p_arg);
static void AppTaskLed1 (void *p_arg);
static void AppTaskLed2 ( void * p_arg );

static void AppTaskLed3 ( void * p_arg );

int main (void)
{
    OS_ERR err; /* Init μC/OS-III. */
    OSInit(&err);
    OSTaskCreate((OS_TCB *) &AppTaskStartTCB, /* Create the start task*/
        (CPU_CHAR *) "App Task Start",
        (OS_TASK_PTR) AppTaskStart,
        (void *) 0,
        (OS_PRIO) APP_TASK_START_PRIO,
        (CPU_STK *) &AppTaskStartStk[0],
        (CPU_STK_SIZE) APP_TASK_START_STK_SIZE / 10,
        (CPU_STK_SIZE) APP_TASK_START_STK_SIZE,
        (OS_MSG_QTY) 5u,
        (OS_TICK) 0u,
        (void *) 0,
        (OS_OPT) (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
        (OS_ERR) *&err);
    OSStart(&err);
}

static void AppTaskStart (void *p_arg)
{
    CPU_INT32U cpu_clk_freq;
    CPU_INT32U cnts;
    OS_ERR err;

    (void)p_arg;
    BSP_Init(); /* Initialize BSP functions */
CPU_Init();

cpu_clk_freq = BSP_CPU_ClkFreq();

cnts = cpu_clk_freq / (CPU_INT32U)OSCfg_TickRate_Hz;

OS_CPU_SysTickInit(cnts);

Mem_Init();

#if OS_CFG_STAT_TASK_EN > 0u
    OSStatTaskCPUUsageInit(&err);
#endif

CPU_IntDisMeasMaxCurReset();

/* 创建信号量 TestSem */
OSSemCreate((OS_SEM  *) &TestSem, // 指向信号量变量的指针
             (CPU_CHAR  *) "TestSem ", // 信号量的名字
             (OS_SEM_CTR  ) 1,
             // 信号量这里是指示事件发生，所以赋值为 0，表示事件还没有发生
             (OS_ERR  *) &err); // 错误类型

/* Create the Led1 task */
OSTaskCreate((OS_TCB  *) &AppTaskLed1TCB,
              (CPU_CHAR  *) "App Task Led1",
              (OS_TASK_PTR ) AppTaskLed1,
              (void  *) 0,
              (OS_PRIO  ) APP_TASK_LED1_PRIO,
(CPU_STK *) &AppTaskLed1Stk[0],
(CPU_STK_SIZE) APP_TASK_LED1_STK_SIZE / 10,
(CPU_STK_SIZE) APP_TASK_LED1_STK_SIZE,
(OS_MSG_QTY ) 5u,
(OS_TICK ) 0u,
(void *) 0,
(OS_OPT ) (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
(OS_ERR *) &err);

/* Create the Led2 task */
OSTaskCreate((OS_TCB *) &AppTaskLed2TCB,
(CPU_CHAR *) "App Task Led2",
(OS_TASK_PTR ) AppTaskLed2,
(void *) 0,
(OS_PRIO ) APP_TASK_LED2_PRIO,
(CPU_STK *) &AppTaskLed2Stk[0],
(CPU_STK_SIZE) APP_TASK_LED2_STK_SIZE / 10,
(CPU_STK_SIZE) APP_TASK_LED2_STK_SIZE,
(OS_MSG_QTY ) 5u,
(OS_TICK ) 0u,
(void *) 0,
(OS_OPT ) (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
(OS_ERR *) &err);

/* Create the Led3 task */
OSTaskCreate((OS_TCB *) &AppTaskLed3TCB,
(CPU_CHAR *) "App Task Led3",
(OS_TASK_PTR ) AppTaskLed3,
(void *) 0,
(OS_PRIO ) APP_TASK_LED3_PRIO,
(CPU_STK *) &AppTaskLed3Stk[0],
(CPU_STK_SIZE) APP_TASK_LED3_STK_SIZE / 10,
(CPU_STK_SIZE) APP_TASK_LED3_STK_SIZE,
(OS_MSG_QTY ) 5u,
(OS_TICK ) 0u,
(void *) 0,
(OS_OPT ) (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
(OS_ERR *) &err);

OSTaskDel ( & AppTaskStartTCB, & err );
}

;/*
***********************************************************************/
/*
    LED1 TASK
***********************************************************************/
*/

static void AppTaskLed1 ( void * p_arg )
{
    OS_ERR err;
    static uint32_t i;
    CPU_TS ts_sem_post;

    (void)p_arg;

    while ( DEF_TRUE )
{
        printf("AppTaskLed1 获取信号量\n");
        //获取二值信号量 TestSem，没获取到则一直等待
        OSSemPend ((OS_SEM *) &TestSem, //等待该信号量被发布
                    (OS_TICK ) 0, //无期限等待
                    (OS_OPT ) OS_OPT_PEND_BLOCKING,
                    //如果没有信号量可用就等待
                    (OS_ERR ) &err);

        /*
         ***************************************************************/
         /*
             LED1 TASK
         ***************************************************************/
         */

(CPU_TS *) &ts_sem_post,
// 获取信号量最后一次被发布的时间戳
(OS_ERR *) &err);    // 返回错误类型

for (i = 0; i < 600000; i++)  // 模拟低优先级任务占用信号量
{
    OSSched(); // 发起任务调度
}

printf("AppTaskLed1 释放信号量!
");
OSSemPost((OS_SEM *) &TestSem,
    // 发布 SemOfKey
    (OS_OPT ) OS_OPT_POST_1,
    // 发布给所有等待任务
    (OS_ERR *) &err);

    macLED1_TOGGLE();
    OTimDelayHMSM (0, 0, 1, 0, OS_OPT_TIME_PERIODIC, &err);
}

/*
**********************************************************************************************
* LED2 TASK
***********************************************************************************************/

static void AppTaskLed2 ( void * p_arg )
{
    OS_ERR err;
(void)p_arg;

while (DEF_TRUE)
{
    printf("AppTaskLed2 Running\n");
    macLED2_TOGGLE();
    OSTimeDlyHMSM (0,0,0,200,OS_OPT_TIME_PERIODIC,&err);
}

/*
************************************************************
* LED3 TASK
************************************************************
*/

static void AppTaskLed3 ( void * p_arg )
{
    OS_ERR       err;
    CPU_TS       ts_sem_post;
    (void)p_arg;

    while (DEF_TRUE)
    {
        printf("AppTaskLed3 获取信号量\n");
        //获取二值信号量 TestSem，没获取到则一直等待
        OSSemPend ((OS_SEM *)&TestSem,    //等待该信号量被发布
                    (OS_TICK ) 0,       //无期限等待
                    (OS_OPT ) OS_OPT_PEND_BLOCKING,
                    //如果没有信号量可用就等待
                    (CPU_TS    *)&ts_sem_post,
                    //获取信号量最后一次被发布的时间戳
                    //下页继续)
24.7.2 互斥量实验

互斥量实验是基于优先级反转实验进行修改的，将信号量改为互斥量，目的是为了测试互斥量的优先级继承机制是否有效，具体见代码清单: 互斥量-11

列表 11: 代码清单: 互斥量-11 互斥量实验

```c
#include <includes.h>

/*
 * *************************************************
 * LOCAL DEFINES
 * *************************************************
 */

OS_SEM TestMutex;       //互斥量
```
/*
 * TCB
 *
 */

static OS_TCB AppTaskStartTCB;
static OS_TCB AppTaskLed1TCB;
static OS_TCB AppTaskLed2TCB;
static OS_TCB AppTaskLed3TCB;

/
* STACKS
*/

static CPU_STK AppTaskStartStk[APP_TASK_START_STK_SIZE];
static CPU_STK AppTaskLed1Stk [APP_TASK_LED1_STK_SIZE];
static CPU_STK AppTaskLed2Stk [APP_TASK_LED2_STK_SIZE];
static CPU_STK AppTaskLed3Stk [APP_TASK_LED3_STK_SIZE];

/*
* FUNCTION PROTOTYPES
*/

static void AppTaskStart (void *p_arg);
static void AppTaskLed1 (void *p_arg);
static void AppTaskLed2 ( void * p_arg );
static void AppTaskLed3 ( void * p_arg );

int main (void)
{
    OS_ERR err;

    OSInit (&err);

    OSTaskCreate ((OS_TCB *) &AppTaskStartTCB,
                   (CPU_CHAR *) "App Task Start",
                   (OS_TASK_PTR ) AppTaskStart,
                   (void *) 0,
                   (OS_PRIO ) APP_TASK_START_PRIO,
                   (CPU_STK *) &AppTaskStartStk[0],
                   (CPU_STK_SIZE) APP_TASK_START_STK_SIZE / 10,
                   (CPU_STK_SIZE) APP_TASK_START_STK_SIZE,
                   (OS_MSG_QTY ) 5u,
                   (OS_TICK ) 0u,
                   (void *) 0,
                   (OS_OPT ) (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
                   (OS_ERR *) &err);

    OSStart (&err);
}

static void AppTaskStart (void * p_arg)
{
    CPU_INT32U cpu_clk_freq;
    CPU_INT32U cnts;
OS_ERR err;  

(void)p_arg;  

BSP_Init();  

CPU_Init();  

cpu_clk_freq = BSP_CPU_ClkFreq();  

cnts = cpu_clk_freq / (CPU_INT32U)OSCfg_TickRate_Hz;  

OS_CPU_SysTickInit(cnts);  

Mem_Init();  

#if OS_CFG_STAT_TASK_EN > 0u  

OSStatTaskCPUUsageInit(&err);  
#endif  

CPU_INTDisMeasMaxCurReset();  

/* 创建互斥信号量 mutex */  
OSMutexCreate ((OS_MUTEX *) &TestMutex,  
                (CPU_CHAR *) "Mutex For Test",  
                (OS_ERR *) &err);  

/* Create the Led1 task */  
OSTaskCreate((OS_TCB *) &AppTaskLed1TCB,  
              (CPU_CHAR *) "App Task Led1",  
              (OS_TASK_PTR ) AppTaskLed1,  
              (void *) 0,  

(续上页)
(OS_PRIO ) APP_TASK_LED1_PRIO,
(CPU_STK   *)&AppTaskLed1Stk[0],
(CPU_STK_SIZE) APP_TASK_LED1_STK_SIZE / 10,
(CPU_STK_SIZE) APP_TASK_LED1_STK_SIZE,
(OS_MSG_QTY  ) 5u,
(OS_TICK     ) 0u,
(void)       0,
(OS_OPT ) (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
(OS_ERR)& err);

/* Create the Led2 task */
OSTaskCreate((OS_TCB   *)&AppTaskLed2TCB,
(CPU_CHAR  )"App Task Led2",
(OS_TASK_PTR ) AppTaskLed2,
(void)       0,
(OS_PRIO    ) APP_TASK_LED2_PRIO,
(CPU_STK   *)&AppTaskLed2Stk[0],
(CPU_STK_SIZE) APP_TASK_LED2_STK_SIZE / 10,
(CPU_STK_SIZE) APP_TASK_LED2_STK_SIZE,
(OS_MSG_QTY  ) 5u,
(OS_TICK    ) 0u,
(void)       0,
(OS_OPT ) (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
(OS_ERR) & err);

/* Create the Led3 task */
OSTaskCreate((OS_TCB   *)&AppTaskLed3TCB,
(CPU_CHAR  )"App Task Led3",
(OS_TASK_PTR ) AppTaskLed3,
(void)       0,
(OS_PRIO    ) APP_TASK_LED3_PRIO,
(CPU_STK   *)&AppTaskLed3Stk[0],
(CPU_STK_SIZE) APP_TASK_LED3_STK_SIZE / 10,
(CPU_STK_SIZE) APP_TASK_LED3_STK_SIZE,
(OS_MSG_QTY ) 5u,
(OS_TICK ) 0u,
(void ) 0,
(OS_OPT ) (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
(OS_ERR ) &err);

OSTaskDel ( & AppTaskStartTCB, & err );
}

/*
*****************************************************************************************/

LED1 TASK
*****************************************************************************************/

static void AppTaskLed1 ( void * p_arg )
{
    OS_ERR err;
    static uint32_t i;

    (void)p_arg;

    while ( DEF_TRUE )
    {
        printf("AppTaskLed1 获取互斥量
");
        //获取互斥量，没获取到则一直等待
        OSMutexPend ((OS_Mutex ) &TestMutex, //申请互斥量
(OS_TICK ) 0, //无期限等待
(OS_OPT ) OS_OPT_PEND_BLOCKING,
//如果不能申请到信号量就阻塞任务

(续上页)

(CPU_TS  *)0,  //不想获得时间戳
(OS_ERR  *)&err);  //返回错误类型

for (i=0; i<600000; i++)  //模拟低优先级任务占用互斥量
{
  OSSched();  //发起任务调度
}

printf("AppTaskLed1 释放互斥量\n");
OSMutexPost ((OS_MUTEX  *)&TestMutex,  //释放互斥量
  (OS_OPT  )OS_OPT_POST_NONE,  //进行任务调度
  (OS_ERR  *)&err);  //返回错误类型

macLED1_TOGGLE ();
OSTimeDlyHMSM (0,0,1,0,OS_OPT_TIME_PERIODIC,&err);
}

/*
 *****************************************************************************/
/*   LED2 TASK
*****************************************************************************/
/*

static  void  AppTaskLed2 ( void  * p_arg )
{
  OS_ERR  err;
  (void)p_arg;

  while (DEF_TRUE)
  {
    printf("AppTaskLed2 Running\n");
  }

论坛:  https://www.firebbs.cn/  547  天猫:  https://yehuosm.tmall.com
macLED2_TOGGLE ();
OSTimeDlyHMSM (0,0,0,200,OS_OPT_TIME_PERIODIC,&err);
}
}

/*
*********************************************************************
* LED3 TASK
*********************************************************************
*/
static void AppTaskLed3 ( void * p_arg )
{
    OS_ERR err;
    (void)p_arg;

    while (DEF_TRUE)
    {
        printf("AppTaskLed3 获取互斥量\n");
        // 获取互斥量，没获取到则一直等待
        OSMutexPend ((OS_MUTEX *)&TestMutex,
                      (OS_TICK *)0,
                      (OS_OPT )OS_OPT_PEND_BLOCKING,
                      // 如果不能申请到信号量就阻塞任务
                      (CPU_TS *)0,
                      (OS_ERR *)&err);
        // 返回错误类型
        macLED3_TOGGLE ();

        printf("AppTaskLed3 释放互斥量\n");
        OSMutexPost ((OS_MUTEX *)&TestMutex,
                     // 释放互斥量
                     // 返回错误类型
                     // 释放互斥量
                     // 返回错误类型
)
}


24.8 实验现象

24.8.1 模拟优先级翻转实验现象

将程序编译好，用 USB 线连接计算机和开发板的 USB 接口（对应丝印为 USB 转串口），用 DAP 仿真器把配套程序下载到野火 STM32 开发板（具体型号根据购买的板子而定，每个型号的板子都配套有对应的程序），在计算机上打开串口调试助手，然后复位开发板就可以在调试助手中看到串口的打印信息，它里面输出了信息表明任务正在运行中，并且很明确可以看到高优先级任务在等待低优先级任务运行完毕才能得到信号量继续运行，具体见图优先级翻转实验现象。
24.8.2 互斥量实验现象

将程序编译好，用 USB 线连接计算机和开发板的 USB 接口（对应丝印为 USB 转串口），用 DAP 仿真器把配套程序下载到野火 STM32 开发板（具体型号根据购买的板子而定，每个型号的板子都配套有对应的程序），在计算机上打开串口调试助手，然后复位开发板就可以在调试助手中看到串口的打印信息，它里面输出了信息表明任务正在运行中，并且很明确可以看到在低优先级任务运行的时候，中优先级任务无法抢占低优先级的任务，这是因为互斥量的优先级继承机制，从而最大程度降低了优先级翻转产生的危害，具体见图互斥量实验现象。
24.9 总结

互斥量更适用于保护各个任务间对共享资源的互斥访问。当然，系统中对于这种互斥访问的资源可以使用很多种保护方式，如关闭中断方式、关调度器方式、信号量保护或者采用互斥量保护。但是这些方式各有好坏，下面就简单说明一下这4种方式的使用情况。具体见
### 共享资源保护方式

<table>
<thead>
<tr>
<th>保护方式</th>
<th>说明</th>
</tr>
</thead>
<tbody>
<tr>
<td>关闭中断方式</td>
<td>什么时候该用：当系统能很快地结束访问该共享资源时，如一些共享的全局变量的操作，可以关闭中断，操作完成再打开中断即可。但我们一般不推荐使用这种方法，因为会导致中断延迟。</td>
</tr>
<tr>
<td>锁调度器方式</td>
<td>当访问共享资源较久的时候，比如对一些列表的操作，如遍历列表、插入、删除等操作，对于操作时间是不确定的，如一些 os 中的内存分配，都可以采用锁定调度器这种方式进行共享资源的保护。</td>
</tr>
<tr>
<td>信号量保护方式</td>
<td>当该共享资源经常被多个被使用时可以使用这种方式。但信号量可能会导致优先级翻转，并且信号量是无法解决这种危害的。</td>
</tr>
<tr>
<td>互斥量保护方式</td>
<td>推荐使用这种方法访问共享资源，尤其当任务要访问的共享资源有阻塞时间的时候。μC/OS-III 的互斥量有内置的优先级，这样可防止优先级翻转。然而，互斥量方式慢于信号量方式，因为互斥量需执行额外的操作，改变任务的优先级。</td>
</tr>
</tbody>
</table>

论坛：[https://www.firebbs.cn/](https://www.firebbs.cn/)  
天猫：[https://yehuosm.tmall.com](https://yehuosm.tmall.com)
第 25 章 事件

25.1 事件的基本概念

事件是一种实现任务间通信的机制，主要用于实现多任务间的同步，但事件通信只能是事件类型的通信，无数据传输。与信号量不同的是，它可以实现一对多、多对多的同步。即一个任务可以等待多个事件的发生：可以是任一一个事件发生时唤醒任务进行事件处理；也可以是几个事件都发生后才唤醒任务进行事件处理。同样，也可以是多个任务同步多个事件。

每一个事件组只需要很少的 RAM 空间来保存事件组的状态。事件组存储在一个 OS_FLAGS 类型的 Flags 变量中，该变量在事件结构体中定义。而变量的宽度由我们自己定义，可以是 8 位、16位、32 位的变量，取决于 os_type.h 中的 OS_FLAGS 的位数。在 STM32 中，我们一般将其定义为 32 位的变量，有 32 个位用来实现事件标志组。每一位代表一个事件，任务通过“逻辑与”或“逻辑或”与一个或多个事件建立关联，形成一个事件组。事件的“逻辑或”也被称作是独立型同步，指的是任务感兴趣的事件任一事件发生即可被唤醒；事件“逻辑与”则被称为是关联型同步，指的是任务感兴趣的若干事件都发生时才被唤醒，并且事件发生的时间可以不同步。

多任务环境下，任务、中断之间往往需要同步操作，一个事件发生会告知等待中的任务，即形成一个任务与任务、中断与任务间的同步。事件可以提供一对多、多对多的同步操作。一对多同步模型：一个任务等待多个事件的触发，这种情况是比较常见的；多对多同步模型：多个任务等待多个事件的触发。

任务可以通过设置事件位来实现事件的触发和等待操作。μC/OS 的事件仅用于同步，不提供数据传输功能。

μC/OS 提供的事件具有如下特点:

• 事件只与任务相关联，事件相互独立，一个 32 位（数据宽度由用户定义）的事件集合用于标识该任务发生的事件类型，其中每一位表示一种事件类型（0 表示该事件类型未发生，1 表示该事件类型已经发生），一共 32 种事件类型。

• 事件仅用于同步，不提供数据传输功能。
• 事件无排队性，即多次向任务设置同一事件（如果任务还未得及读走），等效于只设置一次。
• 允许多个任务对同一事件进行读写操作。
• 支持事件等待超时机制。
• 支持显式清除事件。

在μC/OS 的等待事件中，用户可以选择感兴趣的事件，并且选择等待事件的选项，它有 4 个属性，分别是逻辑与、逻辑或、等待所有事件清除或者等待任意事件清除。当任务等待事件同步时，可以通过任务感兴趣的事件位和事件选项来判断当前获取的事件是否满足要求，如果满足则说明任务等待到对应的事件，系统将唤醒等待的任务；否则，任务会根据用户指定的阻塞超时时间继续等待下去。

25.2 事件的应用场景

μC/OS 的事件用于事件类型的通讯，无数据传输，也就是说，我们可以用事件来做标志位，判断某些事件是否发生了，然后根据结果做处理，那很多人又会问了，为什么我不直接用变量做标志呢，岂不是更好更有效率？非也非也，若是在裸机编程中，用全局变量是最为有效的方法，这点我不否认，但是在操作系统中，使用全局变量就要考虑以下问题了:

• 如何对全局变量进行保护呢，如何处理多任务同时对它进行访问?
• 如何让内核对事件进行有效管理呢？使用全局变量的话，就需要在任务中轮询查看事件是否发送，这简直就是浪费 CPU 资源啊，还有等待超时机制，使用全局变量的话需要用户自己去实现。

所以，在操作系统中，还是使用操作系统给我们提供的通信机制就好了，简单方便还实用。

在某些场合，可能需要多个时间发生了才能进行下一步操作，比如一些危险机器的启动，需要检查各项指标，当指标不达标的时候，无法启动，但是检查各个指标的时候，不能一下子检测完毕啊，所以，需要事件来做统一的等待，当所有的事件都完成了，那么机器才允许启动，这只是事件的其中一个应用。

事件可使用于多种场合，它能够在一定程度上替代信号量，用于任务与任务间，中断与任务间的
同步。一个任务或中断服务例程发送一个事件给事件对象，而后等待的任务被唤醒并处理相应的事件。但是它与信号量不同的是，事件的发送操作是不可累计的，而信号量的释放动作是可累计的。事件另外一个特性是，接收任务可等待多种事件，即多个事件对应一个任务或多个任务。同时按照任务等待的参数，可选择是“逻辑或”触发还是“逻辑与”触发。这个特性也是信号量等所不具备的，信号量只能识别单一同步动作，而不能同时等待多个事件的同步。

各个事件可分别发送或一起发送给事件对象，而任务可以等待多个事件，任务仅对感兴趣的事件进行关注。当有它们感兴趣的事件发生时并且符合感兴趣的条件，任务将被唤醒并进行后续的处理动作。

### 25.3 事件运作机制

等待（接收）事件时，可以根据感兴趣的参数事件类型等待事件的单个或者多个事件类型。事件等待成功后，必须使用 `OS_OPT_PEND_FLAG_CONSUME` 选项来清除已接收到的事件类型，否则不会清除已接收到的事件，这样就需要用户显式清除事件位。用户可以自定义通过传入 `opt` 选项来选择读取模式，是等待所有感兴趣的事件还是等待感兴趣的一个事件。

设置事件时，对指定事件写入指定的事件类型，设置事件集合的对应事件位为 1，可以一次同时写多个事件类型，设置事件成功可能会触发任务调度。

清除事件时，根据入参事件句柄待清除的事件类型，对事件对应位进行清零操作。

事件不与任务相关联，事件相互独立，一个 32 位的变量就是事件的集合，用于标识该任务发生的事件类型，其中每一位表示一种事件类型（0 表示该事件类型未发生，1 表示该事件类型已经发生），一共 32 种事件类型具体见图事件集 `Flags`。

![图示](image)

事件唤醒机制，当任务因为等待某个或者多个事件发生而进入阻塞态，当事件发生的时候会被唤醒，其过程具体见图事件唤醒任务示意图。
任务1对事件3或事件5感兴趣（逻辑或），当发生其中的某一个事件都会被唤醒，并且执行相应操作。而任务2对事件3与事件5感兴趣（逻辑与），当且仅当事件3与事件5都发生的时候，任务2才会被唤醒，如果只有一个其中一个事件发生，那么任务还是会继续等待事件发生。如果在接收事件函数中设置了清除事件位选项OS_OPT_PEND_FLAG_CONSUME，那么当任务唤醒后将把事件3和事件5的事件标志清零，否则事件标志将依然存在。

论坛：https://www.firebbs.cn/ 
天猫：https://yehuosm.tmall.com
25.4 事件控制块

理论上用户可以创建任意个事件（仅限制于处理器的 RAM 大小）。通过设置 os_cfg.h 中的宏定义 OS_CFG_FLAG_EN 为 1 即可开启事件功能。事件是一个内核对象，由数据类型 OS_FLAG_GRP 定义，该数据类型由 os_flag_grp 定义（在 os.h 文件）。

μC/OS 的事件由多个元素组成，在事件被创建时，需要由我们自己定义事件（也可以称之为事件句柄），因为它是用于保存事件的一些信息的，其数据结构 OS_FLAG_GRP 除了事件必须的一些基本信息外，还有 PendList 链表与一个 32 位的事件组变量 Flags 等，为的是方便系统来管理事件。其数据结构具体见代码清单: 事件-1，示意图具体见图事件控制块数据结构。

列表 1: 代码清单: 事件-1 事件控制块数据结构

```c
struct os_flag_grp
```

(下页继续)
```c
{  
/* ------------------ GENERIC MEMBERS ------------------ */
  OS_OBJ_TYPE Type; (1)
  CPU_CHAR *NamePtr; (2)
  OS_PEND_LIST PendList; (3)
#if OS_CFG_DBG_EN > 0u
  OS_FLAG_GRP *DbgPrevPtr;
  OS_FLAG_GRP *DbgNextPtr;
  CPU_CHAR *DbgNamePtr;
#endif
/* ------------------ SPECIFIC MEMBERS ------------------ */
  OS_FLAGS Flags; (4)
  CPU_TS TS; (5)
};
```

- **代码清单: 事件-1 (1)**：事件的类型，用户无需理会，μC/OS 用于识别它是一个事件。
- **代码清单: 事件-1 (2)**：事件的名字，每个内核对象都会被分配一个名，采用字符串形式记录下来。
- **代码清单: 事件-1 (3)**：因为可以有多个任务同时等待系统中的事件，所以事件中包含了一个用于控制挂起任务列表的结构体，用于记录阻塞在此事件上的任务。
- **代码清单: 事件-1 (4)**：事件中包含了很多标志位，Flags 这个变量中保存了当前这些标志位的状态。这个变量可以为 8 位、16 位或 32 位。
- **代码清单: 事件-1 (5)**：事件中的变量 TS 用于保存该事件最后一次被释放的时间戳。当事件被释放时，读取时基计数值并存放到该变量中。

注意：用户代码不能直接访问这个结构体，必须通过 μC/OS 提供的 API 访问。

---

论坛: [https://www.firebbs.cn/](https://www.firebbs.cn/)  
天猫: [https://yehuosm.tmall.com](https://yehuosm.tmall.com)
25.5 事件函数接口

25.5.1 事件创建函数 OSFlagCreate()

事件创建函数，顾名思义，就是创建一个事件，与其他内核对象一样，都是需要先创建才能使用的资源。μC/OS 给我们提供了一个创建事件的函数 OSFlagCreate()，当创建一个事件时，系统会对我们定义的事件控制块进行基本的初始化，所以，在使用创建函数之前，我们需要先定义一个事件控制块（句柄），事件创建函数的源码具体见代码清单：事件-2。

列表 2: 代码清单: 事件-2OSFlagCreate() 源码

```c
void OSFlagCreate (OS_FLAG_GRP *p_grp, (1) //事件指针
CPU_CHAR *p_name, (2) //命名事件
OS_FLAGS flags, (3) //标志初始值
OS_ERR *p_err) (4) //返回错误类型
{
  CPU_SR_ALLOC(); //使用到临界段（关中断时）时必须用到该宏，该宏声明和
  //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
  //SR (临界段关中断只需保存 SR)，开中断时将该值还原。

#ifdef OS_SAFETY_CRITICAL(5)//如果启用了安全检测
  if (p_err == (OS_ERR *))0 //如果错误类型实参为空
  {
    OS_SAFETY_CRITICAL_EXCEPTION(); //执行安全检测异常函数
    return; //返回，停止执行
  }
#endif

#ifdef OS_SAFETY_CRITICAL_IEC61508(6)//如果启用了安全关键
  if (OSSafetyCriticalStartFlag == DEF_TRUE) //如果
    OSSafetyCriticalStart() 后创建
  {
```

*p_err = OS_ERR_ILLEGAL_CREATE_RUN_TIME;  //错误类型为“非法创建内核对象”

return;  //返回，停止执行

#endif

#if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u(7) //如果启用了中断中非法调用检测
    if (OSIntNestingCtr > (OS_NESTING_CTR)0)  //如果该函数是在中断中被调用
    {
        *p_err = OS_ERR_CREATE_ISR;  //错误类型为“在中断中创建对象”
        return;  //返回，停止执行
    }
#endif

#if OS_CFG_ARG_CHK_EN > 0u(8) //如果启用了参数检测
    if (p_grp == (OS_FLAG_GRP *)0)  //如果 p grp 为空
    {
        *p_err = OS_ERR_OBJ_PTR_NULL;  //错误类型为“创建对象为空”
        return;  //返回，停止执行
    }
#endif

OS_CRITICAL_ENTER();  //进入临界段
p_grp->Type = OS_OBJ_TYPE_FLAG;  //标记创建对象数据结构为事件
p_grp->NamePtr = p_name;  //标记事件的名称
p_grp->Flags = flags;  //设置标志初始值
p_grp->TS = (CPU_TS)0;  //清零事件的时间戳
OS_PendListInit(&p_grp->PendList);  //初始化该事件的等待列表

#if OS_CFG_DBG_EN > 0u //如果启用了调试代码和变量
    OS_FlagDbgListAdd(p_grp);  //将该事件添加到事件双向调试链表
#endif

(下页继续)
OSFlagQty++;  
(14) //事件个数加 1

OS_CRITICAL_EXIT_NO_SCHED();  
(15) //退出临界段（无调度）

*p_err = OS_ERR_NONE;  

{  

• 代码清单: 事件-2 (1): 事件控制块指针，指向我们定义的事件控制块结构体变量，所以在创建之前我们需要先定义一个事件控制块变量。

• 代码清单: 事件-2 (2): 事件名称，字符串形式。

• 代码清单: 事件-2 (3): 事件标志位的初始值，一般为常为 0。

• 代码清单: 事件-2 (4): 用于保存返回的错误类型。

• 代码清单: 事件-2 (5): 如果启用了安全检测（默认禁用），在编译时则会包含安全检测相关的代码，如果错误类型实参为空，系统会执行安全检测异常函数，然后返回，不执行创建互斥量操作。

• 代码清单: 事件-2 (6): 如果启用（默认禁用）了安全关键检测，在编译时则会包含安全关键检测相关的代码，如果是在调用 OSSafetyCriticalStart() 后创建该事件，则是非法的，返回错误类型为“非法创建内核对象”错误代码，并且退出，不执行创建事件操作。

• 代码清单: 事件-2 (7): 如果启用了中断中非法调用检测（默认启用），在编译时则会包含中断非法调用检测相关的代码，如果该函数是在中断中被调用，则是非法的，返回错误类型为“在中断中创建对象”的错误代码，并且退出，不执行创建事件操作。

• 代码清单: 事件-2 (8): 如果启用了参数检测（默认启用），在编译时则会包含参数检测相关的代码，如果 p_grp 参数为空，返回错误类型为“创建对象为空”的错误代码，并且退出，不执行创建事件操作。

• 代码清单: 事件-2 (9): 进入临界段，标记创建对象数据结构为事件。

• 代码清单: 事件-2 (10): 初始化事件的名称。

• 代码清单: 事件-2 (11): 设置事件标志的初始值。

• 代码清单: 事件-2 (12): 记录时间戳的变量 TS 初始化为 0。

论坛: https://www.firebbs.cn/  561    天猫: https://yehuosm.tmall.com
[野火]uCOS-III 内核实现与应用开发实战指南

- **代码清单: 事件-2 (13)**: 初始化该事件的等待列表。
- **代码清单: 事件-2 (14)**: 系统事件个数加 1。
- **代码清单: 事件-2 (15)**: 退出临界段（无调度），创建事件成功。

如果我们创建一个事件，那么事件创建成功的示意图具体见图事件创建完成示意图。

事件创建函数的使用实例具体见 代码清单: 事件-3。
列表 3: 代码清单: 事件-3OSFlagCreate() 使用实例

```c
OS_FLAG_GRP flag_grp;  //声明事件
OS_ERR err;

/* 创建事件 flag_grp */
OSFlagCreate ((OS_FLAG_GRP *)&flag_grp, //指向事件的指针
    (CPU_CHAR *)"FLAG For Test",  //事件的名字
    (OS_FLAGS )0,  //事件的初始值
    (OS_ERR *)&err);  //返回错误类型
```

### 25.5.2 事件删除函数 OSFlagDel()

在很多场合，某些事件只用一次的，就好比在事件应用场景说的危险机器的启动，假如各项指标都达到了，并且机器启动成功了，那这个事件之后可能就没用了，那就可以进行销毁了。想要删除事件怎么办？μC/OS 给我们提供了一个删除事件的函数——OSFlagDel()，使用它就能将事件进行删除了。当系统不再使用事件对象时，可以通过删除事件对象控制块来进行删除，具体见代码清单: 事件-4。

注意，想要使用删除事件函数则必须将 OS_CFG_FLAG_DEL_EN 宏定义配置为 1，该宏定义在 os_cfg.h 文件中。

列表 4: 代码清单: 事件-4OSFlagDel() 源码

```c
#if OS_CFG_FLAG_DEL_EN > 0u  //如果启用了 OSFlagDel() 函数
OS_OBJ_QTY OSFlagDel (OS_FLAG_GRP *p_grp, (1)  //事件指针
    OS_OPT opt, (2)  //选项
    OS_ERR *p_err) (3)  //返回错误类型
{
    OS_OBJ_QTY cnt;
    OS_OBJ_QTY nbr_tasks;
```

(下页继续)
OS_PEND_DATA  *p_pend_data;
OS_PEND_LIST   *p_pend_list;
OS_TCB         *p_tcb;
CPU_TS         ts;
CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必须用到该宏，该宏声明和
//定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
// SR（临界段关中断只需保存 SR），开中断时将该值还原。

#ifdef OS_SAFETY_CRITICAL(4) //如果启用（默认禁用）了安全检测
  if (p_err == (OS_ERR *)0)  //如果错误类型实参为空
    {
      OS_SAFETY_CRITICAL_EXCEPTION(); //执行安全检测异常函数
      return ((OS_OBJ_QTY)0); //返回 0（有错误），停止执行
    }
#endif

#if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u(5) //如果启用了中断中非法调用检测
  if (OSIntNestingCtr > (OS_NESTING_CTR)0) //如果该函数在中断中被调用
    {
      *p_err = OS_ERR_DEL_ISR; //错误类型为“在中断中删除对象”
      return ((OS_OBJ_QTY)0); //返回 0（有错误），停止执行
    }
#endif

#if OS_CFG_ARG_CHK_EN > 0u(6) //如果启用了参数检测
  if (p_grp == (OS_FLAG_GRP *)0) //如果 p_grp 为空
    {
      *p_err = OS_ERR_OBJ_PTR_NULL; //错误类型为“对象为空”
      return ((OS_OBJ_QTY)0); //返回 0（有错误），停止执行
    }
  switch (opt) (7) //根据选项分类处理
    {

case OS_OPT_DEL_NO_PEND: //如果选项在预期内
    case OS_OPT_DEL_ALWAYS:
        break; //直接跳出

default: (8) //如果选项超出预期
    *p_err = OS_ERR_OPT_INVALID; //错误类型为“选项非法”
    return ((OS_OBJ_QTY)0); //返回 0 (有错误)，停止执行
} #endif

#if OS_CFG_OBJ_TYPE_CHK_EN > 0u(9) //如果启用了对象类型检测
    if (p_grp->Type != OS_OBJ_TYPE_FLAG) //如果 p_grp 不是事件类型
    {
        *p_err = OS_ERR_OBJ_TYPE; //错误类型为“对象类型有误”
        return ((OS_OBJ_QTY)0); //返回 0 (有错误)，停止执行
    }
#endif

OS_CRITICAL_ENTER(); //进入临界段
p_pend_list = &p_grp->PendList; (10) //获取消息队列的等待列表
cnt = p_pend_list->NbrEntries; (11) //获取等待该队列的任务数
nbr_tasks = cnt; //按照任务数目逐个处理
switch (opt) (12) //根据选项分类处理
{
    case OS_OPT_DEL_NO_PEND: (13) //如果只在没任务等待时进行删除
        if (nbr_tasks == (OS_OBJ_QTY)0) //如果没有任务在等待该事件
        {
#if OS_CFG_DBG_EN > 0u //如果启用了调试代码和变量
        OS_FlagDbgListRemove(p_grp); (14) //将该事件从事件调试列表移除
#endif
        OSFlagQty--; (15) //事件数目减 1
        OS_FlagClr(p_grp); (16) //清除该事件的内容
    }
OS_CRITICAL_EXIT(); //退出临界段
*p_err = OS_ERR_NONE; (17) //错误类型为“无错误”
}
else
{
OS_CRITICAL_EXIT(); //退出临界段
*p_err = OS_ERR_TASK_WAITING; (18) //错误类型为“有任务在等待事件”
}
break; //跳出

case OS_OPT_DEL_ALWAYS: (19) //如果必须删除事件
ts = OS_TS_GET(); (20) //获取时间戳
while (cnt > 0u) (21) //逐个移除该事件等待列表中的任务
{
    p_pend_data = p_pend_list->HeadPtr;
p_tcb = p_pend_data->TCBPtr;
    OS_PendObjDel((OS_PEND_OBJ *)(void *)p_grp),
        p_tcb,
        ts); (22)
    cnt--;
}
#if OS_CFG_DBG_EN > 0u //如果启用了调试代码和变量
    OS_FlagDbgListRemove(p_grp); //将该事件从事件调试列表移除
#endif
    OSFlagQty--; (23) //事件数目减 1
    OS_FlagClr(p_grp); (24) //清除该事件的内容
    OS_CRITICAL_EXIT_NO_SCHED(); //退出临界段 (无调度)
    OSSched(); (25) //调度任务
    *p_err = OS_ERR_NONE; (26) //错误类型为“无错误”
    break; //跳出

default: (27) //如果选项超出预期
OS_CRITICAL_EXIT();  //退出临界段
*p_err = OS_ERR_OPT_INVALID;  //错误类型为“选项非法”
break;  //跳出
}
return (nbr_tasks);  //返回删除事件前等待其的任务数
#endif

• 代码清单: 事件-4 (1): 事件控制块指针，指向我们定义的事件控制块结构体变量，所以在删除之前我们需要先定义一个事件控制块变量，并且成功创建事件后再进行删除操作。

• 代码清单: 事件-4 (2): 事件删除的选项。

• 代码清单: 事件-4 (3): 用于保存返回的错误类型。

• 代码清单: 事件-4 (4): 如果启用了安全检测（默认），在编译时会包含安全检测相关的代码，如果错误类型实参为空，系统会执行安全检测异常函数，然后返回，不执行删除互斥量操作。

• 代码清单: 事件-4 (5): 如果启用了中断中非法调用检测（默认启用），在编译时则会包含中断非法调用检测相关的代码，如果该函数是在中断中被调用，则非法的，返回错误类型为“在中断中删除对象”的错误代码，并且退出，不执行删除事件操作。

• 代码清单: 事件-4 (6): 如果启用了参数检测（默认启用），在编译时则会包含参数检测相关的代码，如果 p_grp 参数为空，返回错误类型为“内核对象为空”的错误代码，并且退出，不执行删除事件操作。

• 代码清单: 事件-4 (7): 判断 opt 选项是否合理，该选项有两个，OS_OPT_DEL_ALWAYS 与 OS_OPT_DEL_NO_PEND，在 os.h 文件中定义。此处是判断一下选项是否在预期之内，如果在则跳出 switch 语句。

• 代码清单: 事件-4 (8): 如果选项超出预期，则返回错误类型为“选项非法”的错误代码，退出，不继续执行。

• 代码清单: 事件-4 (9): 如果启用了对象类型检测，在编译时则会包含对象类型检测相关的代码，如果 p_grp 不是事件类型，返回错误类型为“内核对象类型错误”的错误代码，并且
退出，不执行删除事件操作。

- **代码清单: 事件-4 (10)**：进入临界段，程序执行到这里，表示可以删除事件了，系统首先获取互斥量的等待列表保存到 p_pend_list 变量中，μC/OS 在删除事件的时候是通过该变量访问事件等待列表的任务的。

- **代码清单: 事件-4 (11)**：获取等待该队列的任务数，按照任务个数逐个处理。

- **代码清单: 事件-4 (12)**：根据选项分类处理。

- **代码清单: 事件-4 (13)**：如果 opt 是 OS_OPT_DEL_NO_PEND，则表示只在没有任务等待的情况下删除事件，如果当前系统中没有任务还在等待该事件的某些位，则不能进行删除操作，反之，则可以删除事件。

- **代码清单: 事件-4 (14)**：如果启用了调试代码和变量，将该事件从事件调试列表移除。

- **代码清单: 事件-4 (15)**：系统的事件个数减一。

- **代码清单: 事件-4 (16)**：清除该事件的内容。

- **代码清单: 事件-4 (17)**：删除成功，返回错误类型为“无错误”的错误代码。

- **代码清单: 事件-4 (18)**：如果已有任务在等待该事件，则返回错误类型为“有任务在等待该事件”错误代码。

- **代码清单: 事件-4 (19)**：如果 opt 是 OS_OPT_DEL_ALWAYS，则表示无论如何都必须删除事件，那么在删除之前，系统会把所有阻塞在该事件上的任务恢复。

- **代码清单: 事件-4 (20)**：获取删除时候的时间截。

- **代码清单: 事件-4 (21)**：根据前面 cnt 记录阻塞在该事件上的任务个数，逐个移除该事件等待列表中的任务。

- **代码清单: 事件-4 (22)**：调用 OS_PendObjDel() 函数将阻塞在内核对象（如事件）上的任务从阻塞恢复，此时系统在删除内核对象，删除之后，这些等待事件的任务需要被恢复，其源码具体 **代码清单: 消息队列-1**。

- **代码清单: 事件-4 (23)**：系统事件数目减 1

- **代码清单: 事件-4 (24)**：清除该事件的内容。
• 代码清单: 事件-4 (25): 进行一次任务调度。
• 代码清单: 事件-4 (26): 删除事件完成，返回错误类型为“无错误”的错误代码。
• 代码清单: 事件-4 (27): 如果选项超出预期则返回错误类型为“任务状态非法”的错误代码。
• 代码清单: 事件-4 (28): 返回删除事件前等待其的任务数

事件删除函数 OSFlagDel() 的使用也是很简单的，只需要传入要删除的事件的句柄与选项还有保存返回的错误类型即可，调用函数时，系统将删除这个事件。需要注意的是在调用删除事件函数前，系统应存在已创建的事件。如果删除事件时，系统中有任务正在等待该事件，则不应该进行删除操作，删除事件函数 OSFlagDel() 的使用实例具体见代码清单: 事件-5。

列表 5: 代码清单: 事件-5OSFlagDel() 函数使用实例

```c
OS_FLAG_GRP flag_grp;;
  //声明事件句柄
OS_ERR err;

/* 删除事件*/
OSFlagDel((OS_FLAG_GRP*)&flag_grp,  //指向事件的指针
  OS_OPT_DEL_NO_PEND,
  (OS_ERR*)&err);  //返回错误类型
```

25.5.3 事件设置函数 OSFlagPost()

OSFlagPost() 用于设置事件组中指定的位，当位被置位之后，并且满足任务的等待事件，那么等待在事件该标志位上的任务将会被恢复。使用该函数接口时，通过参数指定的事件标志来设置事件的标志位，然后遍历等待在事件对象上的事件等待列表，判断是否有任务的事件激活要求与当前事件对象标志值匹配，如果有，则唤醒该任务。简单来说，就是设置我们自己定义的事件标志位为 1，并且看看有没有任务在等待这个事件，有的话就唤醒它，OSFlagPost() 函数源码具体见代码清单: 事件-6。
列表 6: 代码清单: 事件-6 OSFlagPost() 源码

```c
OS_FLAGS OSFlagPost (OS_FLAG_GRP *p_grp,   //事件指针
    OS_FLAGS flags,   //选定要操作的标志位
    OS_OPT opt,       //选项
    OS_ERR *p_err)    //返回错误类型
{

    OS_FLAGS flags_cur;
    CPU_TS ts;

    #ifdef OS_SAFETY_CRITICAL //如果启用（默认禁用）了安全检测
        if (p_err == (OS_ERR *)) 0)   //如果错误类型实参为空
            {
                OS_SAFETY_CRITICAL_EXCEPTION(); //执行安全检测异常函数
                return ((OS_FLAGS)0);    //返回 0，停止执行
            }
    #endif

    #if OS_CFG_ARG_CHK_EN > 0u //如果启用（默认启用）了参数检测
        if (p_grp == (OS_FLAG_GRP *)0)   //如果参数 p_grp 为空
            {
                *p_err = OS_ERR_OBJ_PTR_NULL; //错误类型为“事件对象为空”
                return ((OS_FLAGS)0);    //返回 0，停止执行
            }
    #endif

    switch (opt)   //根据选项分类处理
    {
        case OS_OPT_POST_FLAG_SET:   //如果选项在预期之内
            case OS_OPT_POST_FLAG_CLR:
            case OS_OPT_POST_FLAG_SET | OS_OPT_POST_NO_SCHED:
            case OS_OPT_POST_FLAG_CLR | OS_OPT_POST_NO_SCHED:
```

(下页继续)
break;  // 直接跳出

default:  // 如果选项超出预期
    *p_err = OS_ERR_OPT_INVALID;  // 错误类型为“选项非法"
    return ((OS_FLAGS)0);  // 返回 0，停止执行
}
#endif

#if OS_CFG_OBJ_TYPE_CHK_EN > 0u
// 如果启用了对象类型检测
if (p_grp->Type != OS_OBJ_TYPE_FLAG)  // 如果 p_grp 不是事件类型
{
    *p_err = OS_ERR_OBJ_TYPE;  // 错误类型“对象类型有误”
    return ((OS_FLAGS)0);  // 返回 0，停止执行
}
#endif

// 获取时间戳
if (OS_CFG_ISR_POST_DEFERRED_EN > 0u)  // 如果启用了中断延迟发布
    if (OSIntNestingCtr > (OS_NESTING_CTR))
    {
        OS_IntQPost((OS_OBJ_TYPE)OS_OBJ_TYPE_FLAG,  // 将该事件发布到中断消息队列
            (void   *)p_grp,
            (void   *)0,
            (OS_MSG_SIZE)0,
            (OS_FLAGS )flags,
            (OS_OPT   )opt,
            (CPU_TS   )ts,
            (OS_ERR   *)p_err);
        return ((OS_FLAGS)0);  // 返回 0，停止执行
    }
#endif

/* 如果没有启用中断延迟发布 */
注：因为程序大体与之前的程序差不多，此处仅介绍重点。

- 代码清单: 事件-6 (1)：如果启用了中断延迟发布并且该函数在中断被调用，则将该事件发布到中断消息队列。

- 代码清单: 事件-6 (2)：如果没有启用中断延迟发布，则直接将该事件对应的标志位置位。

OS_FlagPost() 函数源码具体见代码清单: 事件-7。

列表 7: 代码清单: 事件-7 OS_FlagPost() 源码

```c
flags_cur = OS_FlagPost(p_grp, flags, opt, ts, p_err); (2)

return (flags_cur);  // 返回当前标志位的值
```

```c
OS_FLAGS OS_FlagPost (OS_FLAG_GRP *p_grp, (1)  // 事件指针
OS_FLAGS flags, (2)  // 选定时要操作的标志位
OS_OPT opt, (3)  // 选项
CPU_TS ts, (4)  // 时隙
OS_ERR *p_err) (5)  // 返回错误类型
{
    OS_FLAGS flags_cur;
    OS_FLAGS flags_rdy;
    OS_OPT mode;
    OS_PEND_DATA *p_pend_data;
    OS_PEND_DATA *p_pend_data_next;
    OS_PEND_LIST *p_pend_list;
    OS_TCB *p_tcb;
    CPU_SR_ALLOC();  // 使用到临界段（在关/开中断时）时必须用到该宏，该宏声明和
```
// 定义一个局部变量，用于保存中断前的 CPU 状态寄存器
// SR（临界段关中断只需保存 SR），开中断时将该值还原。

CPU_CRITICAL_ENTER();  // 关中断

switch (opt) {          // 根据选项分类处理

    case OS_OPT_POST_FLAG_SET:    // 如果要求将选定位 1
        break;

    case OS_OPT_POST_FLAG_CLR:    // 如果要求将选定位 0
        break;

    default:                      // 如果选项超出预期
        CPU_CRITICAL_EXIT();    // 开中断
        ^p_err = OS_ERR_OPT_INVALID;  // 错误类型为“选项非法”

        return ((OS_FLAGS)0);    // 返回 0，停止执行

}  

p_grp->TS = ts;           // 将时间戳存入事件

p_pend_list = &p_grp->PendList;  // 获取事件的等待列表

if (p_pend_list->NbrEntries == 0u) {  // 如果没有任务在等待事件

    CPU_CRITICAL_EXIT();    // 开中断

    ^p_err = OS_ERR_NONE;    // 错误类型为“无错误”

    return (p_grp->Flags);  // 返回事件的标志值

} /* 如果有任务在等待事件 */

OS_CRITICAL_ENTER_CPU_EXIT();  // 进入临界段，重开中断

(下页继续)
p_pend_data = p_pend_list->HeadPtr; (14) //获取等待列表头个等待任务
p_tcb = p_pend_data->TCBPtr;
while (p_tcb != (OS_TCB *)0) (15) //从头至尾遍历等待列表的所有任务
{
    p_pend_data_next = p_pend_data->NextPtr;
    mode = p_tcb->FlagsOpt & OS_OPT_PEND_FLAG_MASK; //获取任务的标志选项
    switch (mode) (16) //根据任务的标志选项分类处理
    {
        case OS_OPT_PEND_FLAG_SET_ALL: (17) //如果要求任务等待的标志位都得置 1
            flags_rdy = (OS_FLAGS)(p_grp->Flags & p_tcb->FlagsPend);
            if (flags_rdy == p_tcb->FlagsPend) //如果任务等待的标志位都置 1 了
            {
                OS_FlagTaskRdy(p_tcb, //让该任务准备运行
                               flags_rdy,
                               ts); (18)
            }
            break; //跳出
        case OS_OPT_PEND_FLAG_SET_ANY: (19)
            //如果要求任务等待的标志位有 1 位置 1 即可
            flags_rdy = (OS_FLAGS)(p_grp->Flags & p_tcb->FlagsPend); (20)
            if (flags_rdy != (OS_FLAGS)0) //如果任务等待的标志位有 1 的
            {
                OS_FlagTaskRdy(p_tcb, //让该任务准备运行
                               flags_rdy,
                               ts); (21)
            }
            break; //跳出
        #if OS_CFG_FLAG_MODE_CLR_EN > 0u (22) //如果启用了标志位清零触发模式
        case OS_OPT_PEND_FLAG_CLR_ALL: (23) //如果要求任务等待的标志位都得清 0
            break; //跳出
        #endif
    }
}
flags_rdy = (OS_FLAGS)(~p_grp->Flags & p_tcb->FlagsPend);
if (flags_rdy == p_tcb->FlagsPend) //如果任务等待的标志位都清 0 了
{
    OS_FlagTaskRdy(p_tcb,  //让该任务准备运行
       flags_rdy,
       ts);  (24)
}
break;  //跳出

case OS_OPT_PEND_FLAG_CLR_ANY:  (25)
    //如果要求任务等待的标志位有 1 位清 0 即可
    flags_rdy = (OS_FLAGS)(~p_grp->Flags & p_tcb->FlagsPend);
    if (flags_rdy != (OS_FLAGS)0) //如果任务等待的标志位有清 0 的
    {
        OS_FlagTaskRdy(p_tcb,  //让该任务准备运行
           flags_rdy,
           ts);  (26)
    }
    break;  //跳出
#endif

default:  (27) //如果标志选项超出预期
    OS_CRITICAL_EXIT();  //退出临界段
    *p_err = OS_ERR_FLAG_PEND_OPT;  //错误类型为“标志选项非法”
    return ((OS_FLAGS)0);  //返回 0，停止运行
}
p_pend_data = p_pend_data_next;  (28) //准备处理下一个等待任务
if (p_pend_data != (OS_PEND_DATA *)0) //如果该任务存在
{
    p_tcb = p_pend_data->TCBPtr;  (29) //获取该任务的任务控制块
}
else //如果该任务不存在
{
p_tcb = (OS_TCB *)0;  \(30\) //清空 p_tcb，退出 while 循环
}

OS_CRITICAL_EXIT_NO_SCHED();  \(31\) //退出临界段（无调度）

if ((opt & OS_OPT_POST_NO_SCHED) == (OS_OPT)0) \(32\) //如果 opt 没选择“发布时不调度任务”
{
    OSSched();  \(33\) //任务调度
}

CPU_CRITICAL_ENTER();  \(34\) //中断
flags_cur = p_grp->Flags;  \(35\) //获取事件的标志值
CPU_CRITICAL_EXIT();  \(36\) //开中断
*p_err = OS_ERR_NONE;  \(37\) //错误类型为“无错误”
return (flags_cur);  \(38\) //返回事件的当前标志值

- 代码清单: 事件-7 (1): 事件指针。
- 代码清单: 事件-7 (2): 选定要操作的标志位。
- 代码清单: 事件-7 (3): 设置事件标志位的选项。
- 代码清单: 事件-7 (4): 时间戳。
- 代码清单: 事件-7 (5): 返回错误类型。
- 代码清单: 事件-7 (6): 根据选项分类处理。
- 代码清单: 事件-7 (7): 如果要求将选定位 1，则置 1 即可，然后跳出 switch 语句。
- 代码清单: 事件-7 (8): 如果要求将选定位 0，将选定位清零即可，然后跳出 switch 语句。
- 代码清单: 事件-7 (9): 如果选项超出预期，返回错误类型为“选项非法”的错误代码，退出。

论坛: https://www.firebbs.cn/  天猫: https://yehuosm.tmall.com
• 代码清单: 事件-7 (10): 将时间戳存入事件的 TS 成员变量中。
• 代码清单: 事件-7 (11): 获取事件的等待列表。
• 代码清单: 事件-7 (12): 如果当前没有任务在等待事件，置位后直接退出即可，并且返回事件的标志值。
• 代码清单: 事件-7 (13): 如果有任务在等待事件，那么进入临界段，重开中断。
• 代码清单: 事件-7 (14): 获取等待列表头个等待任务，然后获取到对应的任务控制块，保存在 p_tcb 变量中。
• 代码清单: 事件-7 (15): 当事件等待列表中有任务的时候，就从头至尾遍历等待列表的所有任务。
• 代码清单: 事件-7 (16): 获取任务感兴趣的事件标志选项，根据任务的标志选项分类处理。
• 代码清单: 事件-7 (17): 如果要求任务等待的标志位都得置 1，就获取一下任务已经等待到的事件标志，保存在 flags_rdy 变量中。
• 代码清单: 事件-7 (18): 如果任务等待的标志位都置 1 了，就调用 OS_FlagTaskRdy() 函数让该任务恢复为就绪态，准备运行，然后跳出 switch 语句。
• 代码清单: 事件-7 (19): 如果要求任务等待的标志位有任意一个位置 1 即可。
• 代码清单: 事件-7 (20): 那么就获取一下任务已经等待到的事件标志，保存在 flags_rdy 变量中。
• 代码清单: 事件-7 (21): 如果任务等待的标志位有置 1 的，也就是满足了任务唤醒的条件，就调用 OS_FlagTaskRdy() 函数让该任务恢复为就绪态，准备运行，然后跳出 switch 语句。
• 代码清单: 事件-7 (22): 如果启用了标志位清零触发模式，在编译的时候就会包含事件标志位清零触发的代码。
• 代码清单: 事件-7 (23): 如果要求任务等待的标志位都得清 0，那就看看等待任务对应的标志位是否清零了。
• 代码清单: 事件-7 (24): 如果任务等待的标志位都清 0 了，就调用 OS_FlagTaskRdy() 函数让该任务恢复为就绪态，准备运行，然后跳出 switch 语句。
• 代码清单: 事件-7 (25): 如果要求任务等待的标志位有 1 位清 0 即可。

论坛: https://www.firebbs.cn/ 577 天猫: https://yehuosm.tmall.com
• 代码清单: 事件-7 (26): 那么如果任务等待的标志位有请 0 的，就让任务恢复为就绪态。

• 代码清单: 事件-7 (27): 如果标志选项超出预期，返回错误类型为“标志选项非法”的错误代码，并且推出。

• 代码清单: 事件-7 (28): 准备处理下一个等待任务。

• 代码清单: 事件-7 (29): 如果该任务存在，获取该任务的运行控制块。

• 代码清单: 事件-7 (30): 如果该任务不存在，清空 p_tcb，退出 while 循环。

• 代码清单: 事件-7 (31): 进行一次任务调度。

• 代码清单: 事件-7 (32): 事件标志位设置完成，返回事件的当前标志值。

OSFlagPost() 的运用很简单，举个例子，比如我们要记录一个事件的发生，这个事件在事件组的位置是 bit0，当它还未发生的时候，那么事件组 bit0 的值也是0，当它发生的时候，我们往事件标志组的 bit0 位中写入这个事件，也就是 0x01，那这就表示事件已经发生了，当然，μC/OS 也支持事件清零触发。为了便于理解，一般操作我们都是用宏定义来实现 #define EVENT (0x01 << x)，“<< x”表示写入事件集合的 bit x，在使用该函数之前必须先创建事件，具体见 代码清单: 事件-8 。

### 列表 8: 代码清单: 事件-8xEvtGrpSetBits() 函数使用实例

```c
#define KEY1_EVENT  (0x01 << 0) //设置事件掩码的位 0
#define KEY2_EVENT  (0x01 << 1) //设置事件掩码的位 1

OS_FLAG_GRP flag_grp; //声明事件标志组

static void AppTaskPost ( void * p_arg )
{
    OS_ERR err;

    (void)p_arg;
}
```

(下页继续)
while (DEF_TRUE) {
    //任务体
    //如果 KEY1 被按下
    if (Key_ReadStatus ( macKEY1_GPIO_PORT, macKEY1_GPIO_PIN, 1 ) == 1)
    {
        macLED1_ON (); //点亮 LED1
        OSFlagPost ((OS_FLAG_GRP *)&flag_grp,
                    //将标志组的 BIT0 置 1
                    (OS_FLAGS )KEY1_EVENT,
                    (OS_OPT )OS_OPT_POST_FLAG_SET,
                    (OS_ERR  *)&err);
    }
    else //如果 KEY1 被释放
    {
        macLED1_OFF (); //熄灭 LED1
        OSFlagPost ((OS_FLAG_GRP *)&flag_grp,
                    //将标志组的 BIT0 清零
                    (OS_FLAGS )KEY1_EVENT,
                    (OS_OPT )OS_OPT_POST_FLAG_CLR,
                    (OS_ERR  *)&err);
    }
    //如果 KEY2 被按下
    if (Key_ReadStatus ( macKEY2_GPIO_PORT, macKEY2_GPIO_PIN, 1 ) == 1)
    {
        macLED2_ON (); //点亮 LED2
        OSFlagPost ((OS_FLAG_GRP *)&flag_grp,

25.5.4 事件等待函数 OSFlagPend()  

既然标记了事件的发生，那么我们怎么知道他到底有没有发生，这也是需要一个函数来获取事件是否已经发生，μC/OS 提供了一个等待指定事件的函数——OSFlagPend()，通过这个函数，任务可以知道事件标志组中的哪些位，有什么事件发生了，然后通过“逻辑与”、“逻辑或”等操作对感兴趣的事件进行获取，并且这个函数实现了等待超时机制，当且仅当任务等待的事件发生时，任务才能获取到事件信息。在这段时间中，如果事件一直没发生，该任务将保持阻塞状态以等待
事件发生。当其他任务或中断服务程序正在等待的事件设置对应的标志位，该任务将自动由阻塞态转为就绪态。当任务等待的时间超过了指定的阻塞时间，即使事件还未发生，任务也会自动从阻塞态转为就绪态。这样有效的体现了操作系统的实时性，如果事件正确获取（等待到）则返回对应的事件标志位；由用户判断再做处理，因为在事件超时的时候也可能会返回一个不能确定的事件值，所以最好判断一下任务所等待的事件是否真的发生。

OSFlagPend() 函数源码具体见代码清单:事件-9。

注意: OSFlagPend() 函数源码比较长，我们只挑重点进行讲解。

### 列表 9: 代码清单: 事件-9 OSFlagPend() 源码

```c
OS_FLAGS OSFlagPend (OS_FLAG_GRP *p_grp, (1) //事件指针
    OS_FLAGS flags, (2) //选定要操作的标志位
    OS_TICK timeout, (3) //等待期限 (单位: 时钟节拍)
    OS_OPT opt, (4) //选项
    CPU_TS *p_ts, (5) //返回等待事件标志时的时间戳
    OS_ERR *p_err) (6) //返回错误类型
{
    CPU_BOOLEAN consume;
    OS_FLAGS flags_rdy;
    OS_OPT mode;
    OS_PEND_DATA pend_data;
    CPU_SR_ALLOC(); //使用到临界段 (在调/开中断时) 时必须用到该宏，该宏声明和
    //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
    // SR (在临界段关中断时,最后将该值还原。

    #ifdef OS_SAFETY_CRITICAL //如果启用 (默认禁用) 了安全检测
    if (p_err == (OS_ERR *)0) //如果错误类型实参为空
    {
        OS_SAFETY_CRITICAL_EXCEPTION(); //执行安全检测异常函数
        return ((OS_FLAGS)0); //返回 0 (有错误), 停止执行
    }
    #endif
```
```c
#if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u //如果启用了中断中非法调用检测
    if (OSIntNestingCtr > (OS_NESTING_CTR)0) //如果该函数在中断中被调用
    {
        *p_err = OS_ERR_PEND_ISR; //错误类型为“在中断中中止等待”
        return ((OS_FLAGS)0); //返回 0 (有错误), 停止执行
    }
#endif

#if OS_CFG_ARG_CHK_EN > 0u //如果启用了参数检测
    if (p_grp == (OS_FLAG_GRP *)0) //如果 p_grp 为空
    {
        *p_err = OS_ERR_OBJ_PTR_NULL; //错误类型为“对象为空”
        return ((OS_FLAGS)0); //返回 0 (有错误), 停止执行
    }
#endif

    switch (opt) (7) //根据选项分类处理
    {
        case OS_OPT_PEND_FLAG_CLR_ALL: //如果选项在预期内
        case OS_OPT_PEND_FLAG_CLR_ANY:
        case OS_OPT_PEND_FLAG_SET_ALL:
        case OS_OPT_PEND_FLAG_SET_ANY:
        case OS_OPT_PEND_FLAG_CLR_ALL | OS_OPT_PEND_FLAG_CONSUME:
        case OS_OPT_PEND_FLAG_CLR_ANY | OS_OPT_PEND_FLAG_CONSUME:
        case OS_OPT_PEND_FLAG_SET_ALL | OS_OPT_PEND_FLAG_CONSUME:
        case OS_OPT_PEND_FLAG_SET_ANY | OS_OPT_PEND_FLAG_CONSUME:
        case OS_OPT_PEND_FLAG_CLR_ALL | OS_OPT_PEND_NON_BLOCKING:
        case OS_OPT_PEND_FLAG_CLR_ANY | OS_OPT_PEND_NON_BLOCKING:
        case OS_OPT_PEND_FLAG_SET_ALL | OS_OPT_PEND_NON_BLOCKING:
        case OS_OPT_PEND_FLAG_SET_ANY | OS_OPT_PEND_NON_BLOCKING:
        case OS_OPT_PEND_FLAG_CLR_ALL | OS_OPT_PEND_FLAG_CONSUME | OS_OPT_PEND_NON_BLOCKING:
        case OS_OPT_PEND_FLAG_CLR_ANY | OS_OPT_PEND_FLAG_CONSUME | OS_OPT_PEND_NON_BLOCKING:
```
```c
        case OS_OPT_PEND_FLAG_SET_ALL | OS_OPT_PEND_FLAG_CONSUME | OS_OPT_PEND_ 
            NON_BLOCKING:
          case OS_OPT_PEND_FLAG_SET_ANY | OS_OPT_PEND_FLAG_CONSUME | OS_OPT_PEND_ 
            NON_BLOCKING:
            break;            // 直接跳出

        default:            (8) // 如果选项超出预期
          *p_err = OS_ERR_OPT_INVAL; // 错误类型为“选项非法”
          return ((OS_OBJ_QTY)0);    // 返回 0 (有错误)，停止执行
        }
    #endif

    #if OS_CFG_OBJ_TYPE_CHK_EN > 0u // 如果启用了对象类型检测
      if (p_grp->Type != OS_OBJ_TYPE_FLAG) // 如果 p_grp 不是事件类型
        {
          *p_err = OS_ERR_OBJ_TYPE; // 错误类型为“对象类型有误”
          return ((OS_FLAGS)0);      // 返回 0 (有错误)，停止执行
        }
    #endif

      if ((opt & OS_OPT_PEND_FLAG_CONSUME) !=(OS_OPT)0) (9) // 选择了标志位匹配后自动取反
        {
        consume = DEF_TRUE;
      }
    else(10) // 未选择标志位匹配后自动取反
      {
        consume = DEF_FALSE;
      }

      if (p_ts != (CPU_TS *)0) // 如果 p_ts 非空
      {
```

(下页继续)
*\p_ts = (CPU_TS)*0;  //初始化（清除）p_ts，待用于返回时间戳
*
*
mode = opt & OS_OPT_PEND_FLAG_MASK; (11) //从选项中提取对标志位的要求
CPU_CRITICAL_ENTER();  //关中断
switch (mode) (12) //根据事件触发模式分类处理
{
  case OS_OPT_PEND_FLAG_SET_ALL: (13) //如果要求所有标志位均要置 1
    flags_rdy = (OS_FLAGS)(p_grp->Flags & flags); //提取想要的标志位的值
    if (flags_rdy == flags) (14) //如果该值与期望值匹配
    {
      if (consume == DEF_TRUE)(15) //如果要求将标志位匹配后取反
      {
        p_grp->Flags &= ~flags_rdy; //清除事件的相关标志位
      }
      OSTCBCurPtr->FlagsRdy = flags_rdy; (16) //保存让任务脱离等待的标志值
      if (p_ts != (CPU_TS*)0) //如果 p_ts 非空
      {
        *p_ts = p_grp->TS;  //获取任务等待时的时间戳
      }
      CPU_CRITICAL_EXIT();  //关中断
      *p_err = OS_ERR_NONE;  //错误类型为“无错误”
      return (flags_rdy); (17) //返回让任务脱离等待的标志值
    }
  else(18)
    //如果想要标志位的值与期望值不匹配
    {
      if ((opt & OS_OPT_PEND_NON_BLOCKING) != (OS_OPT)0) //如果选择了不阻塞任务
      {
        CPU_CRITICAL_EXIT();  //关中断
        *p_err = OS_ERR_PEND_WOULD_BLOCK; //错误类型为“渴求阻塞”
      }
    }
}
```c
return ((OS_FLAGS)0);      // (19) 返回 0 (有错误)，停止执行
}
else (20) // 如果选择了阻塞任务
{
    if (OSSchedLockNestingCtr > (OS_NESTING_CTR)0)  // 如调度器被锁
    {
        CPU_CRITICAL_EXIT();             // 关中断
        *p_err = OS_ERR_SCHED_LOCKED;     // 错误类型为“调度器被锁”
        return ((OS_FLAGS)0);      // (21) 返回 0 (有错误)，停止执行
    }
} /* 如果调度器未被锁 */
OS_CRITICAL_ENTER_CPU_EXIT();  // 进入临界段，重开中断
OS_FlagBlock(&pend_data,  // 阻塞当前运行任务，等待事件
            p_grp,
            flags,
            opt,
            timeout);    // (22)
OS_CRITICAL_EXIT_NO_SCHED();   // 退出临界段 (无调度)
}
break;                   // 跳出

case OS_OPT_PEND_FLAG_SET_ANY:   // (23) 如果要求有标志位被置 1 即可
flags_rdy = (OS_FLAGS)(p_grp->Flags & flags);  // 提取想要的标志位的值
if (flags_rdy != (OS_FLAGS)0)    // (24) 如果有位被置 1
{
    if (consume == DEF_TRUE)      // 如果要求将标志位匹配后取反
    {
        p_grp->Flags &= ~flags_rdy; // 清零事件的相关标志位
    }

    OSTCBCurPtr->FlagsRdy = flags_rdy; // 保存让任务脱离等待的标志值
```
if (p_ts != (CPU_TS *))  //如果 p_ts 非空
{
    *p_ts = p_grp->TS;  //获取任务等到事件时的时间戳
}
CPU_CRITICAL_EXIT();  //开中断
*p_err = OS_ERR_NONE;  //错误类型为“无错误”
return (flags_rdy);  //（25）返回任务脱离等待的标志值
}
else//如果没有位被置 1
{
    if ((opt & OS_OPT_PEND_NON_BLOCKING) != (OS_OPT)0)  //如果没设置阻塞任务
    {
        CPU_CRITICAL_EXIT();  //关中断
        *p_err = OS_ERR_PEND_WOULD_BLOCK;  //错误类型为“渴求阻塞”
        return ((OS_FLAGS)0);  //（26）返回 0（有错误），停止执行
    }
else//如果设置了阻塞任务
{
    if (OSSchedLockNestingCtr > (OS_NESTING_CTR)0)  //如果调度器被锁
    {
        CPU_CRITICAL_EXIT();  //关中断
        *p_err = OS_ERR_SCHED_LOCKED;  //错误类型为“调度器被锁”
        return ((OS_FLAGS)0);  //（27）返回 0（有错误），停止执行
    }
    /* 如果调度器没被锁 */
    OS_CRITICAL_ENTER_CPU_EXIT();  //进入临界段，重开中断
    OS_FlagBlock(&pend_data,  //阻塞当前运行任务，等待事件
    p_grp,
    flags,
    /* */
if (flags_rdy == flags)  //如果该值与期望值匹配
{
    if (consume == DEF_TRUE)  //如果要求将标志位匹配后取反
    {
        p_grp->Flags | flags_rdy;  //置 1 事件的相关标志位
    }
    OSTCBCurPtr->FlagsRdy = flags_rdy;  //保存让任务脱离等待的标志值
    if (p_ts != (CPU_TS*)0)  //如果 p_ts 非空
    {
        *p_ts = p_grp->TS;  //获取任务等到事件时的时间戳
    }
    CPU_CRITICAL_EXIT();  //开中断
    *p_err = OS_ERR_NONE;  //错误类型为“无错误”
    return (flags_rdy);  //返回 0 (有错误)，停止执行
}
else  //如果想要标志位的值与期望值不匹配
{
    if ((opt & OS_OPT_PEND_NON_BLOCKING) != (OS_OPT)0)  //如果选择了不阻塞任务
    {
        OS_CRITICAL_EXIT_NO_SCHED();  //退出中断 (无调度)
    }
}
CPU_CRITICAL_EXIT();  // 关中断
*p_err = OS_ERR_PEND_WOULD_BLOCK;  // 错误类型为“渴求阻塞”
return ((OS_FLAGS)0);  (32) // 返回 0 (有错误)，停止执行
}
else // 如果选择了阻塞任务
{
    if (OSSchedLockNestingCtr > (OS_NESTING_CTR)0)  // 如果调度器被锁
    {
        CPU_CRITICAL_EXIT();  // 关中断
        *p_err = OS_ERR_SCHED_LOCKED;  // 错误类型为“调度器被锁”
        return ((OS_FLAGS)0);  (33) // 返回 0 (有错误)，停止执行
    }
}
/* 如果调度器未被锁 */
OS_CRITICAL_ENTER_CPU_EXIT();  // 进入临界段，重开中断
OS_FlagBlock(&pend_data, p_grp, flags, opt, timeout);  (34)
OS_CRITICAL_EXIT_NO_SCHED();  // 退出临界段 (无调度)
break;  // 跳出

case OS_OPT_PEND_FLAG_CLR_ANY:  (35) // 如果要求有标志位被清零即可
    flags_rdy = (OS_FLAGS)~p_grp->Flags & flags; // 提取想要的标志位的值
    if (flags_rdy != (OS_FLAGS)0)  // 如果有位被清零
    {
        if (consume == DEF_TRUE)  // 如果要求将标志位匹配后取反
        {
            p_grp->Flags |= flags_rdy;  (36) // 置 1 事件的相关标志位
236）
OSTCBCurPtr->FlagsRdy = flags_rdy;  // 保存让任务脱离等待的标志值
237
if (p_ts != (CPU_TS *)0)  // 如果 p_ts 非空
238
{  
  *p_ts = p_grp->TS;  // 获取任务等到事件时的时间戳
239
}
240
CPU_CRITICAL_EXIT();  // 开中断
241
*p_err = OS_ERR_NONE;  // 错误类型为“无错误”
242
return (flags_rdy);  // 返回 0 (有错误)，停止执行
243}
244
else // 如果没有位被清零
245
{
  if ((opt & OS_OPT_PEND_NON_BLOCKING) != (OS_OPT)0)  // 如果没设置阻塞任务
246
  {
    CPU_CRITICAL_EXIT();  // 开中断
247    *p_err = OS_ERR_PEND_WOULD_BLOCK;  // 错误类型为“渴求阻塞”
248    return ((OS_FLAGS)0);  // 返回 0 (有错误)，停止执行
249  }
250
  else // 如果设置了阻塞任务
251
  {
    if (OSSchedLockNestingCtr > (OS_NESTING_CTR)0)  // 如果调度器被锁
252
    {
      CPU_CRITICAL_EXIT();  // 开中断
253      *p_err = OS_ERR_SCHED_LOCKED;  // 错误类型为“调度器被锁”
254      return ((OS_FLAGS)0);  // 返回 0 (有错误)，停止执行
255    }
256
    /* 如果调度器没被锁 */
257    OS_CRITICAL_ENTER_CPU_EXIT();  // 进入临界段，重开中断
258    OS_FlagBlock(&pend_data,  // 阻塞当前运行任务，等待事件
259      // 结束
p_grp,
flags,
opt,
timeout);        // 退出中断（无调度）

OS_CRITICAL_EXIT_NO_SCHED();
}  
break;            // 跳出
#endif

default:           // 如果要求超出预期
CPU_CRITICAL_EXIT();
*p_err = OS_ERR_OPT_INVALID;    // 错误类型为“选项非法”
return ((OS_FLAGS) 0);        // 返回 0 (有错误)，停止执行
}

OSSched();        // 任务调度
*/ 任务等到了事件后得以继续运行 /*
CPU_CRITICAL_ENTER();     // 关中断
switch (OSTCBCurPtr->PendStatus) // 根据运行任务的等待状态分类处理
{
    case OS_STATUS_PEND_OK:       // 如果等到了事件
        if (p_ts != (CPU_TS *) 0) // 如果 p_ts 非空
        {
            *p_ts = OSTCBCurPtr->TS; // 返回等到事件时的时间戳
        }
    *p_err = OS_ERR_NONE;         // 错误类型为“无错误”
    break;                       // 跳出

    case OS_STATUS_PEND_ABORT:     // 如果等待被中止
        if (p_ts != (CPU_TS *) 0) // 如果 p_ts 非空
        {


*p_ts = OSTCBCurPtr->TS;    //返回等待被中止时的时间戳
}
CPU_CRITICAL_EXIT();    //开中断
*p_err = OS_ERR_PEND_ABORT;    //错误类型为“等待被中止”
break;

case OS_STATUS_PEND_TIMEOUT:      //46 //如果等待超时
    if (p_ts != (CPU_TS *)0)    //如果 p_ts 非空
    {
        *p_ts = (CPU_TS *)0;    //清零 p_ts
    }
    CPU_CRITICAL_EXIT();    //开中断
    *p_err = OS_ERR_TIMEOUT;    //错误类型为“超时”
    break;    //跳出

case OS_STATUS_PEND_DEL:      //47 //如果等待对象被删除
    if (p_ts != (CPU_TS *)0)    //如果 p_ts 非空
    {
        *p_ts = OSTCBCurPtr->TS;    //返回对象被删时的时间戳
    }
    CPU_CRITICAL_EXIT();    //开中断
    *p_err = OS_ERR_OBJ_DEL;    //错误类型为“对象被删”
    break;    //跳出

default:      //48 //如果等待状态超出预期
    CPU_CRITICAL_EXIT();    //开中断
    *p_err = OS_ERR_STATUS_INVALID;    //错误类型为“状态非法”
    break;    //跳出
}
if (*p_err != OS_ERR_NONE)      //49 //如果有错误存在
{
    return ((OS_FLAGS)0);    //返回 0 (有错误)，停止执行
}
flags_rdy = OSTCBCurPtr->FlagsRdy;  //读取让任务脱离等待的标志值
if (consume == DEF_TRUE)
  //如果需要取反触发事件的标志位
{
  switch (mode)  //根据事件触发模式分类处理
  {
    case OS_OPT_PEND_FLAG_SET_ALL:    //如果是通过置 1 来标志事件的发生
      case OS_OPT_PEND_FLAG_SET_ANY:
        p_grp->Flags &= ~flags_rdy;     //置 1 事件里触发事件的标志位
        break;                         //跳出
    
#if OS_CFG_FLAG_MODE_CLR_EN > 0u  //如果启用了标志位清零触发模式
    case OS_OPT_PEND_FLAG_CLR_ALL:    //如果是通过清零来标志事件的发生
      case OS_OPT_PEND_FLAG_CLR_ANY:
        p_grp->Flags |= flags_rdy;     //置 1 事件里触发事件的标志位
        break;                        //跳出
#endif
    default:                         //如果触发模式超出预期
      CPU_CRITICAL_EXIT();          //开中断
      *p_err = OS_ERR_OPT_INVALID;  //错误类型为“选项非法”
      return ((OS_FLAGS)0);         //返回 0 (有错误)，停止执行
  }
}
CPU_CRITICAL_EXIT();                  //开中断
*p_err = OS_ERR_NONE;                 //错误类型为“无错误”
return (flags_rdy);                   //返回让任务脱离等待的标志值
· 代码清单: 事件-9 (1): 事件指针。
· 代码清单: 事件-9 (2): 选定要等待的标志位。
· 代码清单: 事件-9 (3): 等待不到事件时指定阻塞时间（单位: 时钟节拍）。
· 代码清单: 事件-9 (4): 等待的选项。
· 代码清单: 事件-9 (5): 保存返回等到事件标志时的时间戳。
· 代码清单: 事件-9 (6): 保存返回错误类型。
· 代码清单: 事件-9 (7): 此处是判断一下等待的选项是否在预期内，如果在预期内则继续操作，跳出 switch 语句。
· 代码清单: 事件-9 (8): 如果选项超出预期，返回错误类型为“选项非法”的错误代码，并且退出，不继续执行等待事件操作。
· 代码清单: 事件-9 (9): 如果用户选择了标志位匹配后自动取反，变量 consume 就为 DEF_TRUE。
· 代码清单: 事件-9 (10): 如果未选择标志位匹配后自动取反，变量 consume 则为 DEF_FALSE。
· 代码清单: 事件-9 (11): 从选项中提取对标志位的要求，利用“&”运算操作符获取选项并且保存在 mode 变量中。
· 代码清单: 事件-9 (12): 根据事件触发模式分类处理。
· 代码清单: 事件-9 (13): 如果任务要求所有标志位均要置 1，那么就提取想要的标志位的值保存在 flags_rdy 变量中。
· 代码清单: 事件-9 (14): 如果该值与任务的期望值匹配。
· 代码清单: 事件-9 (15): 如果要求将标志位匹配后取反，就将事件的相关标志位清零。
· 代码清单: 事件-9 (16): 保存让任务脱离等待的标志值，此时已经等到任务要求的事件了，就可以退出了。
· 代码清单: 事件-9 (17): 返回错误类型为“无错误”的错误代码与让任务脱离等待的标志值。
· 代码清单: 事件-9 (18): 如果想要标志位的值与期望值不匹配。

论坛: https://www.firebbs.cn/ 593 天猫: https://yehuosm.tmall.com
• 代码清单: 事件-9 (19): 并且如果用户选择了不阻塞任务，那么返回错误类型为“渴求阻塞”的错误代码，退出。

• 代码清单: 事件-9 (20): 而如果用户选择了阻塞任务。

• 代码清单: 事件-9 (21): 那就判断一下调度器是否被锁，如果被锁了，返回错误类型为“调度器被锁”的错误代码，并且退出。

• 代码清单: 事件-9 (22): 如果调度器没有被锁，则调用 OS_FlagBlock() 函数阻塞当前任务，在阻塞中继续等待任务需要的事件。

• 代码清单: 事件-9 (23): 如果要求有标志位被置 1 即可，那就提取想要的标志位的值保存在 flags_rdy 变量中。

• 代码清单: 事件-9 (24): 如果有任何一位被置 1，就表示等待到了事件。如果要求将标志位匹配后取反，将事件的相关标志位清零。

• 代码清单: 事件-9 (25): 等待成功，就返回让任务脱离等待的标志值。

• 代码清单: 事件-9 (26): 如果没有位被置 1，并且用户没有设置阻塞时间，那么就返回错误类型为“渴求阻塞”的错误代码，然后退出。

• 代码清单: 事件-9 (27): 如果设置了阻塞任务，但是调度器被锁了，返回错误类型为“调度器被锁”的错误代码，并且退出。

• 代码清单: 事件-9 (28): 如果调度器没被锁，则调用 OS_FlagBlock() 函数阻塞当前任务，在阻塞中继续等待任务需要的事件。

• 代码清单: 事件-9 (29): 如果启用了标志位清零触发模式（宏定义 OS_CFG_FLAG_MODE_CLR_EN 被配置为 1），则在编译的时候会包含事件清零触发相关代码。

• 代码清单: 事件-9 (30): 如果要求所有标志位均要清零，首先提取想要的标志位的值保存在 flags_rdy 变量中，如果该值与任务的期望值匹配，那么就表示等待的事件。

• 代码清单: 事件-9 (31): 如果要求将标志位匹配后取反，就置 1 事件的相关标志位，因为现在是清零触发的，事件标志位取反就是将对应标志位置一。

• 代码清单: 事件-9 (32): 如果想要标志位的值与期望值不匹配，并且如果用户选择了不阻塞
任务，那么返回错误类型为“渴求阻塞”的错误代码，退出。

- **代码清单: 事件-9 (33):** 如果调度器被锁，返回错误类型为“调度器被锁”的错误代码，并且退出。

- **代码清单: 事件-9 (34):** 如果调度器没有被锁，则调用 OS_FlagBlock() 函数阻塞当前任务，在阻塞中继续等待任务需要的事件。

- **代码清单: 事件-9 (35):** 如果要求有标志位被清零即可，提取想要的标志位的值，如果有位被清零则表示等待到事件。

- **代码清单: 事件-9 (36):** 如果要求将标志位匹配后取反，将事件的相关标志位置 1。

- **代码清单: 事件-9 (37):** 等待到事件就返回对应的事件标志位。

- **代码清单: 事件-9 (38):** 如果没有位被清零，并且如果用户没设置阻塞任务，那么就返回错误类型为“渴求阻塞”的错误代码，然后退出。

- **代码清单: 事件-9 (39):** 如果设置了阻塞任务，但是调度器被锁了，返回错误类型为“调度器被锁”的错误代码，并且退出。

- **代码清单: 事件-9 (40):** 如果调度器没有被锁，则调用 OS_FlagBlock() 函数阻塞当前任务，在阻塞中继续等待任务需要的事件。

- **代码清单: 事件-9 (41):** 如果要求超出预期，返回错误类型为“选项非法”的错误代码，退出。

- **代码清单: 事件-9 (42):** 执行到这里，说明任务没有等待到事件，并且用户还选择了阻塞任务，那么就进行一次任务调度。

- **代码清单: 事件-9 (43):** 当程序能执行到这里，就说明大体上有两种情况，要么是任务获取到对应的事件了；要么任务还没获取到事件（任务没获取到事件的情况有很多种），无论是哪种情况，都先把中断关掉再说，再根据当前运行任务的等待状态分类处理。

- **代码清单: 事件-9 (44):** 如果等到了事件，返回等到事件时的时间戳，然后退出。

- **代码清单: 事件-9 (45):** 如果任务在等待事件中被中止，返回等待被中止时的时间戳，记录错误类型为“等待被中止”的错误代码，然后退出。

- **代码清单: 事件-9 (46):** 如果等待超时，返回错误类型为“等待超时”的错误代码，退出。
• 代码清单: 事件-9 (47): 如果等待对象被删除，返回对象被删时的时间戳，记录错误类型为“对象被删”的错误代码，退出。

• 代码清单: 事件-9 (48): 如果等待状态超出预期，记录错误类型为“状态非法”的错误代码，退出。

• 代码清单: 事件-9 (49): 如果有错误存在，返回 0，表示没有等到到事件。

• 代码清单: 事件-9 (50): 如果没有错误存在，如果需要取反触发事件的标志位。

• 代码清单: 事件-9 (51): 根据事件触发模式分类处理。

• 代码清单: 事件-9 (52): 如果是通过置 1 来标志事件的发生，将事件里触发事件的标志位清零。

• 代码清单: 事件-9 (53): 如果是通过清零来标志事件的发生，那么就将事件里触发事件的标志位置 1。

• 代码清单: 事件-9 (54): 如果触发模式超出预期，返回错误类型为“选项非法”的错误代码。

• 代码清单: 事件-9 (55): 返回让任务脱离等待的标志值。

至此，任务等待事件函数就已经讲解完毕，其实 μC/OS 这种利用状态机的方法等待事件，根据不一样的情况进行处理，是很好的，省去很多逻辑的代码。

下面简单分析处理过程：当用户调用这个函数接口时，系统首先根据用户指定参数和接收选项来判断它要等待的事件是否发生，如果已经发生，则根据等待选项来决定是否清除事件的相应标志位，并且返回事件标志位的值，但是这个值可能不是一个稳定的值，所以在等待到对应事件的时候，我们最好要判断事件是否与任务需要的一致；如果事件没有发生，则把任务添加到事件等待列表中，将当前任务阻塞，直到事件发生或等待时间超时，事件等待函数 OSFlagPend() 使用实例具体见代码清单: 事件-10。

列表 10: 代码清单: 事件-10OSFlagPend() 使用实例

```c
#define KEY1_EVENT (0x01 << 0)  //设置事件掩码的位 0
#define KEY2_EVENT (0x01 << 1)  //设置事件掩码的位 1

OS_FLAG_GRP flag_grp;  //声明事件标志组
```

论坛: https://www.firebbs.cn/  天猫: https://yehuosm.tmall.com
static void AppTaskPend ( void * p_arg )
{
    OS_ERR err;

    (void)p_arg;

    //任务体
    while (DEF_TRUE)
    {
        //等待标志组的 BIT0 和 BIT1 均被置 1
        OSFlagPend ((OS_FLAG_GRP *)&flag_grp,
            (OS_FLAGS )(( KEY1_EVENT | KEY2_EVENT ),
            (OS_TICK )0,
            (OS_OPT )OS_OPT_PEND_FLAG_SET_ALL |
            OS_OPT_PEND_BLOCKING,
            (CPU_TS *)0,
            (OS_ERR *)&err);

        LED3_ON ();     //点亮 LED3

        //等待标志组的 BIT0 和 BIT1 有一个被清零
        OSFlagPend ((OS_FLAG_GRP *)&flag_grp,
            (OS_FLAGS )(( KEY1_EVENT | KEY2_EVENT ),
            (OS_TICK )0,
            (OS_OPT )OS_OPT_PEND_FLAG_CLR_ANY |
            OS_OPT_PEND_BLOCKING,
            (CPU_TS *)0,
            (OS_ERR *)&err);

        LED3_OFF ();     //熄灭 LED3
    }
25.6 事件实验

事件标志组实验是在 μC/OS 中创建了两个任务，一个是设置事件任务，一个是等待事件任务，两个任务独立运行，设置事件任务通过检测按键的按下情况设置不同的事件标志位，等待事件任务则获取这两个事件标志位，并且判断两个事件是否都发生，如果是则输出相应信息，LED 进行翻转。等待事件任务一直在等待事件的发生，等待到事件之后清除对应的事件标记位，具体见代码清单: 事件-II。

列表11: 代码清单: 事件-11 事件实验

```c
#include <includes.h>

OS_FLAG_GRP flag_grp;       //声明事件标志组

#define KEY1_EVENT (0x01 << 0) //设置事件掩码的位 0
#define KEY2_EVENT (0x01 << 1) //设置事件掩码的位 1

static OS_TCB AppTaskStartTCB;  //任务控制块
static OS_TCB AppTaskPostTCB;
static OS_TCB AppTaskPendTCB;

static CPU_STK AppTaskStartStk[APP_TASK_START_STK_SIZE];   //任务栈
static CPU_STK AppTaskPostStk [ APP_TASK_POST_STK_SIZE ];
static CPU_STK AppTaskPendStk [ APP_TASK_PEND_STK_SIZE ];
```
static void AppTaskStart ( void *p_arg );
static void AppTaskPost ( void * p_arg );
static void AppTaskPend ( void * p_arg );

int main ( void )
{
  OS_ERR err;
  OSInit(&err); //初始化 μC/OS-III

  /* 创建起始任务 */
  OSTaskCreate((OS_TCB *)&AppTaskStartTCB,
                (CPU_CHAR *)"App Task Start",
                (OS_TASK_PTR ) AppTaskStart,
                (void *) 0,
                (OS_PRIO ) APP_TASK_START_PRIO,
                (CPU_STK *)&AppTaskStartStk[0],
                (CPU_STK_SIZE) APP_TASK_START_STK_SIZE / 10,
                (OS_MSG_QTY ) 5u,
                (OS_TICK ) 0u,
                (void *) 0,
                (OS_TICK ) OSCfg_TickRate_Hz / 10)
}

论坛: https://www.firebbs.cn/
天猫: https://yehuosm.tmall.com
 static void AppTaskStart (void *p_arg)  
{
    CPU_INT32U cpu_clk_freq;
    CPU_INT32U cnts;
    OS_ERR err;

    (void)p_arg;

    BSP_Init();
    // 基板初始化
    CPU_Init();
    // 初始化 CPU 组件（时间戳、关中断时间测量和主机名）

    cpu_clk_freq = BSP_CPU_ClkFreq();
    // 获取 CPU 内核时钟频率（SysTick 工作时钟）
    cnts = cpu_clk_freq / (CPU_INT32U)OSCfg_TickRate_Hz;
    // 根据用户设定的时钟节拍频率计算 SysTick 定时器的计数值
    OS_CPU_SysTickInit(cnts);
    // 调用 SysTick 初始化函数，设置定时器计数值和启动定时器

    Mem_Init();
    // 初始化内存管理组件（堆内存池和内存池表）
```c
#if OS_CFG_STAT_TASK_EN > 0u
    // 如果启用（默认启用）了统计任务
    OSStatTaskCPUPUsageInit(&err);
#endif

CPU_IntDisMeasMaxCurReset();
    // 复位（清零）当前最大关中断时间

/* 创建事件标志组 flag_grp */
OSFlagCreate ((OS_FLAG_GRP *)flag_grp,
    (CPU_CHAR *)"FLAG For Test",
    (OS_FLAGS )0,
    (OS_ERR  *)&err);
    // 返回错误类型

/* 创建 AppTaskPost 任务 */
OSTaskCreate((OS_TCB  *)&AppTaskPostTCB,
    (CPU_CHAR  *)"App Task Post",
    (OS_TASK_PTR ) AppTaskPost,
    (void        *) 0,
    (OS_PRIO     ) APP_TASK_POST_PRIO,
    (CPU_STK     *)&AppTaskPostStk[0],
    (CPU_STK_SIZE) APP_TASK_POST_STK_SIZE / 10,
    // 任务栈空间剩下 1/10 时限制其增长
```
(CPU_STK_SIZE) APP_TASK_POST_STK_SIZE,
//任务栈空间（单位：sizeof(CPU(STK))
(OS_MSG_QTY) 5u,
//任务可接收的最大消息数
(OS_TICK) 0u,
//任务的时间片节拍数（0 表默认值 OSCfg_TickRate_Hz/10）
(void *) 0,
//任务扩展（0 表不扩展）
(OS_OPT) (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
//任务选项
(OS_ERR *) &err;
//返回错误类型

/* 创建 AppTaskPend 任务 */
OSTaskCreate((OS_TCB *) &AppTaskPendTCB,
//任务控制块地址
(CPU_CHAR *) "App Task Pend",
//任务名称
(OS_TASK_PTR) AppTaskPend,
//任务函数
(void *) 0,
//传递给任务函数（形参 p_arg）的实参
(OS_PRIO) APP_TASK_PEND_PRIO,
//任务的优先级
(CPU_STK *) &AppTaskPendStk[0],
//任务栈的基地址
(CPU_STK_SIZE) APP_TASK_PEND_STK_SIZE / 10,
//任务栈空间剩下 1/10 时限制其增长
(CPU_STK_SIZE) APP_TASK_PEND_STK_SIZE,
//任务栈空间（单位：sizeof(CPU(STK))
(OS_MSG_QTY) 5u,
//任务可接收的最大消息数

(下页继续)
(OS_TICK) 0u,
//任务的时间片节拍数 (0 表默认值 OSCfg_TickRate_Hz/10)
(void *) 0,
//任务扩展 (0 表不扩展)
(OS_OPT)(OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
//任务选项
(OS_ERR *)&err);
//返回错误类型

OSTaskDel (&AppTaskStartTCB, &err);
//删除起始任务本身，该任务不再运行
}

static void AppTaskPost (void * p_arg)
{
    OS_ERR err;
    (void)p_arg;

    while (DEF_TRUE) //任务体
    {
        if (Key_ReadStatus (macKEY1_GPIO_PORT, macKEY1_GPIO_PIN, 1) == 1)
        //如果 KEY1 被按下
        {
            //点亮 LED1
            printf("KEY1 被按下\n");
            OSFlagPost ((OS_FLAG_GRP *)&flag_grp,
                        //将标志组的 BIT0 置 1
                        (OS_FLAGS)KEY1_EVENT,
                        (OS_OPT)OS_OPT_POST_FLAG_SET,
                        (OS_ERR *)&err);

            (下页继续)
if ( Key_ReadStatus ( macKEY2_GPIO_PORT, macKEY2_GPIO_PIN, 1 ) == 1 )

//如果 KEY2 被按下
{
    //点亮 LED2
    printf("KEY2 被按下\n");
    OSFlagPost ((OS_FLAG_GRP *)flag_grp,
        //将标志组的 BIT1 置1
        (OS_FLAGS )KEY2_EVENT,
        (OS_OPT )OS_OPT_POST_FLAG_SET,
        (OS_ERR *)&err);

    OSTimeDlyHMSM ( 0, 0, 0, 20, OS_OPT_TIME_DLY, & err );
}

static void AppTaskPend ( void * p_arg )
{
    OS_ERR err;
    OS_FLAGS flags_rdy;
    (void)p_arg;
    while ( DEF_TRUE )
        //任务体
    
}
25.7 事件实验现象

程序编译好，用 USB 线连接计算机和开发板的 USB 接口（对应丝印为 USB 转串口），用 DAP 仿真器把配套程序下载到野火 STM32 开发板（具体型号根据购买的板子而定，每个型号的板子都配套有对应的程序），在计算机上打开串口调试助手，然后复位开发板就可以在调试助手中看到串口的打印信息，按下开发板的 KEY1 按键发送事件 1，按下 KEY2 按键发送事件 2；我们按下 KEY1 与 KEY2 试试，在串口调试助手中可以看到运行结果，并且当事件 1 与事件 2 都发生的时候，开发板的 LED 会进行翻转，具体见图事件标志组实验现象。
第 26 章  软件定时器

26.1 软件定时器的基本概念

定时器，是指从指定的时刻开始，经过一个指定时间，然后触发一个超时事件，用户可以自定义定时器的周期与频率。类似生活中的闹钟，我们可以设置闹钟每天什么时候响，还能设置响的次数，是响一次还是每天都响。

定时器有硬件定时器和软件定时器之分：

硬件定时器是芯片本身提供的定时功能。一般是由外部晶振提供给芯片输入时钟，芯片向软件模块提供一组配置寄存器，接受控制输入，到达设定时间值后芯片中断控制器产生时钟中断。硬件定时器的精度一般很高，可以达到纳秒级别，并且是中断触发方式。

软件定时器，软件定时器是由操作系统提供的一类系统接口，它构建在硬件定时器基础之上，使系统能够提供不受硬件定时器资源限制的定时器服务，它实现的功能与硬件定时器也是类似的。

使用硬件定时器时，每次在定时时间到达之后就会自动触发一个中断，用户在中断中处理信息；而使用软件定时器时，需要我们在创建软件定时器时指定时间到达后要调用的函数（也称超时函数/回调函数），为了统一，下文均用回调函数描述，在回调函数中处理信息。

注意：软件定时器回调函数的上下文是任务，下文所说的定时器均为软件定时器。

软件定时器在被创建之后，当经过设定的时钟计数值后会触发用户定义的回调函数。定时精度与系统时钟的周期有关。一般系统利用 SysTick 作为软件定时器的基础时钟，软件定时器的回调函数类似硬件的中断服务函数，所以，回调函数也要快进快出，而且回调函数中不能有任何阻碍任务运行的情况（软件定时器回调函数的上下文环境是任务），比如 OSTimeDly() 以及其他能阻塞任务运行的函数，两次触发回调函数的时间间隔 period 叫定器的定时周期。

μC/OS 操作系统提供软件定时器功能，软件定时器的使用相当于扩展了定时器的数量，允许创建更多的定时业务。μC/OS 软件定时器功能上支持：

• 裁剪：能通过宏关闭软件定时器功能。
μC/OS 提供的软件定时器支持单次模式和周期模式，单次模式和周期模式的定时时间到之后都会调用软件定时器的回调函数，用户可以在回调函数中加入要执行的工程代码。

单次模式：当用户创建了定时器并启动了定时器后，定时时间到了，只执行一次回调函数之后就将不再重复执行，当然用户还是可以调用软件定时器启动函数 OSTmrStart() 来启动一次软件定时器。

周期模式：这个定时器会按照设置的定时时间循环执行回调函数，直到用户将定时器删除，具体见图软件定时器的单次模式与周期模式。

当然，μC/OS 中软件定时器的周期模式也分为两种，一种是有初始化延迟的周期模式，另一种是无初始化延迟的周期模式，由 OSTmrCreate() 中的 “dly” 参数设置，这两种周期模式基本是一致的，但是有个细微的差别。

有初始化延迟的周期模式：在软件定时器创建的时候，其第一个定时周期是由定时器中的 dly 参数决定，然后在运行完第一个周期后，其以后的定时周期均由 period 参数决定。

无初始化延迟的周期模式：该定时器从始至终都按照周期运行。

论坛：https://www.firebbs.cn/ 608 天猫：https://yehuosm.tmall.com
比如我们创建两个周期定时器，定时器 1 是无初始化延迟的定时器，周期为 100 个 tick（时钟节拍），定时器 2 是有初始化延迟的定时器，其初始化延迟的 dly 参数为 150 个 tick，周期为 100 个 tick，从 tick 为 0 的时刻就启动了两个软件定时器。定时器 1 从始至终都按照正常的周期运行，但是定时器 2 则在第一个周期中的运行周期为 dly，从第二个运行周期开始按照正常的 100 个 tick 来运行。其示意图具体见两种周期模式运行示意图。

\( \mu\text{C/OS} \) 通过一个 OS_TmrTask 任务（也叫软件定时器任务）来管理软件定时器，它是在系统初始化时（OSInit() 函数中）自动创建的，为了满足用户定时需求。TmrTask 任务会在定时器节拍到来的时候检查定时器列表，看看是否有定时器时间到了，如果到了就调用其回调函数。只有设置 os_cfg.h 中的宏定义 OS_CFGDBG_EN 设置为 1，才会将软件定时器相关代码编译进来，才能正常使用软件定时器相关功能。

### 26.2 软件定时器应用场景

在很多应用中，我们需要一些定时器任务，硬件定时器受硬件的限制，数量上不足以满足用户的实际需求，无法提供更多的定时器，那么可以采用软件定时器来完成，由软件定时器代替硬件定时器任务。但需要注意的是软件定时器的精度是无法和硬件定时器相比的，因为在软件定时器的定时过程中是极有可能被其他中断所打断，因为软件定时器的执行上下文环境是任务。所以，软件定时器更适用于对时间精度要求不高的任务，一些辅助型的任务。
26.3 软件定时器的精度

在操作系统中，通常软件定时器以系统节拍为计时的时基单位。系统节拍是系统的心跳节拍，表示系统时钟的频率，就类似心跳。1s 能跳动多少次，系统节拍配置为 OS_CFG_TICK_RATE_HZ，该宏在 os_app_cfg.h 中有定义，默认是 1000。那么系统的时钟节拍周期就为 1ms（1s 跳动 1000 下，每一刻就为 1ms）。

μC/OS 软件定时器的精度（分辨率）取决于系统时基频率，也就是变量 OS_CFG_TMR_TASK_RATE_HZ 的值，它是以 Hz 为单位的。如果软件定时器任务的频率（OS_CFG_TMR_TASK_RATE_HZ）设置为 10Hz，系统中所有软件定时器的精度为十分之一秒。事实上，这是用于软件定时器的推荐值，因为软件定时器常用于不精确时间尺度的任务。

注：为了书写简便，下文统一采用定时器表示软件定时器，如非特别说明，本章所有的定时器均为软件定时器。

而且定时器所定时的数值必须是这个定时器任务精度的整数倍，例如，定时器任务的频率为 10Hz，那么上层软件定时器定时数值只能是 100ms, 200ms, 1000ms 等，而不能取值为 150ms。由于系统节拍与软件定时器频率决定了系统中定时器能够分辨的精确度，用户可以根据实际 CPU 的处理能力和实时性需求设置合适的数值，软件定时器频率的值越大，精度越高，但是系统开销也将越大，因为这代表在 1 秒中系统进入定时器任务的次数也就越多。

注意：定时器任务的频率 OS_CFG_TMR_TASK_RATE_HZ 的值不能大于系统时基频率 OS_CFG_TMR_TASK_RATE_HZ 的值。

26.4 软件定时器控制块

本章先了解软件定时器的使用再讲解软件定时器的运作机制。

μC/OS 的软件定时器也属于内核对象，是一个可以裁剪的功能模块，同样在系统中由一个控制块管理其相关信息，软件定时器的控制块中包含创建的软件定时器基本信息，在使用定时器前我们需要通过 OSTmrCreate() 函数创建一个软件定时器，但是在创建前需要我们定义一个定时器的句柄（控制块），下面来看看软件定时器控制块的成员变量，具体见代码清单: 定时器-1。
列表 1: 代码清单: 定时器-1 软件定时器控制块

```c
struct os_tmr
{
    OS_OBJ_TYPE Type; (1)
    CPU_CHAR *NamePtr; (2)
    OS_TMR_CALLBACK_PTR CallbackPtr; (3)
    void *CallbackPtrArg; (4)
    OS_TMR *NextPtr; (5)
    OS_TMR *PrevPtr; (6)
    OS_TICK Match; (7)
    OS_TICK Remain; (8)
    OS_TICK Dly; (9)
    OS_TICK Period; (10)
    OS_OPT Opt; (11)
    OS_STATE State; (12)

#if OS_CFG_DBG_EN > 0u
    OS_TMR *DbgPrevPtr;
    OS_TMR *DbgNextPtr;
#endif
};
```

- **代码清单: 定时器-1 (1):** 结构体开始于一个“Type”域，μC/OS 可以通过这个域辨认它是个定时器（其他内核对象的结构体首部也有“Type”）。如果函数需传递一种内核对象，μC/OS 会检测“Type”域是否为参数所需的类型。

- **代码清单: 定时器-1 (2):** 每个内核对象都可以被命名，以便于用户调试，这是一个指向内核对象名的指针。

- **代码清单: 定时器-1 (3):** CallbackPtr 是一个指向函数的指针，被指向的函数称作回调函数，当定时器定时间间到达后，其指向的回调函数将被调用。如果定时器创建时该指针值为 NULL，回调函数将不会被调用。

- **代码清单: 定时器-1 (4):** 当回调函数需要接受一个参数时（CallbackPtr 不为 NULL），这个
参数通过该指针传递给回调函数，简单来说就是指向回调函数中的形参。

- **代码清单: 定时器-1 (5)**: NextPtr 指针指向下一个定时器

- **代码清单: 定时器-1 (6)**: PrevPtr 指针指向下一个定时器，与 NextPtr 指针联合工作将定时器链接成一个双向链表。

- **代码清单: 定时器-1 (7)**: 当定时器管理器中的变量 OSTmrTickCtr 的值等于定时器中的 Match 值时，表示定时器时间到了，Match 也被称为匹配时间（唤醒时间）。

- **代码清单: 定时器-1 (8)**: Remain 中保存了距定时器定时时间到达还有多少个时基。

- **代码清单: 定时器-1 (9)**: Dly 这个值包含了定时器的初次定时值（可以看作是第一次延迟的值），这个值以定时器时基为最小单位。

- **代码清单: 定时器-1 (10)**: Period 是定时器的定时周期（当被设置为周期模式时）。这个值以定时器时基为最小单位。

- **代码清单: 定时器-1 (11)**: Opt 是定时器的选项，可选参数。

- **代码清单: 定时器-1 (12)**: State 记录定时器的状态。

软件定时器控制块示意图具体见软件定时器控制块。
<table>
<thead>
<tr>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>NamePtr</td>
</tr>
<tr>
<td>CallbackPtr</td>
</tr>
<tr>
<td>CallbackPtrArg</td>
</tr>
<tr>
<td>NextPtr</td>
</tr>
<tr>
<td>PrevPtr</td>
</tr>
<tr>
<td>Match</td>
</tr>
<tr>
<td>Remain</td>
</tr>
<tr>
<td>Dly</td>
</tr>
<tr>
<td>Period</td>
</tr>
<tr>
<td>Opt</td>
</tr>
<tr>
<td>State</td>
</tr>
<tr>
<td>Opt</td>
</tr>
<tr>
<td>State</td>
</tr>
</tbody>
</table>

注意：即使了解 OS_TMR 结构体的内容，用户也不允许直接访问这些内容，必须通过 μC/OS 提供的 API 进行访问。
26.5 软件定时器函数接口讲解

26.5.1 创建软件定时器函数 OSTmrCreate()

软件定时器也是内核对象，与消息队列、信号量等内核对象一样，都是需要创建之后才能使用的资源。我们在创建的时候需要指定定时器延时初始值 dly、定时器周期、定时器工作模式、回调函数等。每个软件定时器只需少许的 RAM 空间，理论上 µC/OS 支持无限多个软件定时器，只要 RAM 足够即可。

创建软件定时器函数 OSTmrCreate() 源码具体见代码清单: 定时器-3。

列表 2: 代码清单: 定时器-3 OSTmrCreate() 源码

```c
void OSTmrCreate (OS_TMR *p_tmr, OS_OPT opt, OS_TICK period, OS_TICK dly, CPU_CHAR *p_name, OS_TMR_CALLBACK_PTR p_callback, OS_ERR *p_err) {
    CPU_SR_ALLOC();
    // 使用到临界段（在关/开中断时）时必须用到该宏，该宏声明和定义一个局部变量，用于保存关中断前的 CPU 状态寄存器 SR (临界段关中断只需保存 SR)
    //，开中断时将该值还原。

#ifdef OS_SAFETY_CRITICAL //如果启用（默认禁用）了安全检测
    if (p_err == (OS_ERR *)0) { //如果错误类型实参为空
        OS_SAFETY_CRITICAL_EXCEPTION(); //执行安全检测异常函数
        return; //返回，不执行定时操作
    }
#endif
```

(下页继续)
```c
#pragma once

#if defined(OS_SAFETY_CRITICAL_IEC61508) // 如果启用（默认禁用）了安全关键
  // 如果是在调用 OSSafetyCriticalStart() 后创建该定时器
  if (OSSafetyCriticalStartFlag == DEF_TRUE)
  {
    *p_err = OS_ERR_ILLEGAL_CREATE_RUN_TIME; // 错误类型为“非法创建内核对象”
    return; // 返回，不执行定时操作
  }
#endif

#if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u
  // 如果启用（默认启用）了中断进入非法调用检测
  if (OSIntNestingCtr > (OS_NESTING_CTR)0) // 如果该函数是在中断中被调用
  {
    *p_err = OS_ERR_TMR_ISR; // 错误类型为“在中断函数中定时”
    return; // 返回，不执行定时操作
  }
#endif

#if OS_CFG_ARG_CHK_EN > 0u // 如果启用（默认启用）了参数检测
  if (p_tmrr == (OS_TMR *)0) // 如果参数 p_tmrr 为空
  {
    *p_err = OS_ERR_OBJ_PTR_NULL; // 错误类型为“定时器对象为空”
    return; // 返回，不执行定时操作
  }

  switch (opt) // 根据延时选项参数 opt 分类操作
  {
    case OS_OPT_TMR_PERIODIC: // 如果选择周期性定时
    if (period == (OS_TICK)0) // 如果周期重载实参为 0
    {
```

论坛: https://www.firebbs.cn/   615   天猫: https://yehuosm.tmall.com
*p_err = OS_ERR_TMR_INVALID_PERIOD;  //错误类型为“周期载荷参数无效”
    return;                          //返回，不执行定时操作

} break;

case OS_OPT_TMR_ONE_SHOT:              //如果选择一次性定时
    if (dly == (OS_TICK)0)                  //如果定时初始实参为 0
        { *p_err = OS_ERR_TMR_INVALID_DLY;   //错误类型为“定时初始实参无效”
           return;                          //返回，不执行定时操作
        }
    break;

default:                                //如果选项超出预期
    *p_err = OS_ERR_OPT_INVALID;         //错误类型为“选项非法”
    return;                            //返回，不执行定时操作
}
#endif

OS_CRITICAL_ENTER();                  //进入临界段，初始化定时器指针
    p_tmr->State     = (OS_STATE )OS_TMR_STATE_STOPPED;
    p_tmr->Type      = (OS_OBJ_TYPE )OS_OBJ_TYPE_TMR;
    p_tmr->NamePtr   = (CPU_CHAR  *)p_name;
    p_tmr->Dly       = (OS_TICK  )dly;
    p_tmr->Match     = (OS_TICK  )0;
    p_tmr->Remain    = (OS_TICK  )0;
    p_tmr->Period    = (OS_TICK  )period;
    p_tmr->Opt       = (OS_OPT   )opt;
    p_tmr->CallbackPtr = (OS_TMR_CALLBACK_PTR)p_callback;
    p_tmr->CallbackPtrArg = (void  *)p_callback_arg;
    p_tmr->NextPtr   = (OS_TMR   *)0;
    p_tmr->PrevPtr   = (OS_TMR   *)0;

(下页继续)
# if OS_CFG_DBG_EN > 0u
  //如果启用（默认启用）了调试代码和变量
  OS_TmrDbgListAdd(p_tmr); //将该定时添加到定时器双向调试链表
#endif

OSmrQty++; //定时器个数加 1

OS_CRITICAL_EXIT_NO_SCHED(); //退出临界段（无调度）
*p_err = OS_ERR_NONE; //错误类型为“无错误”
}

定时器创建函数比较简单，主要是根据用户指定的参数将定时器控制块进行相关初始化，并且定时器状态会被设置为 OS_TMR_STATE_STOPPED，具体见源码注释即可。

该函数的使用也是很简单，具体见 代码清单: 定时器-4

论坛：https://www.firebbs.cn/  617  天猫：https://yehuosm.tmall.com
列表 3: 代码清单: 定时器-4 软件定时器创建函数 OSTmrCreate() 使用实例

```c
OS_ERR err;
OS_TMR my_tmr;    //声明软件定时器对象

/* 创建软件定时器 */
OSTmrCreate ((OS_TMR*)&my_tmr,         //软定时器对象
              (CPU_CHAR*)"MySoftTimer",  //命名软定时器
              (OS_TICK)10,              //定时器初始值，依 10Hz 时基计算，即 1s
              (OS_TICK)10,              //定时器周期重载值，依 10Hz 时基计算，即 1s
              (OS_OPT)OS_OPT_TMR_PERIODIC,  //周期性定时
              (OS_TMR_CALLBACK_PTR&TmrCallback,  //回调函数
               (void*)"Timer Over!",   //传递参数给回调函数
               (OS_ERR*)err);          //返回错误类型
```

26.5.2 启动软件定时器函数 OSTmrStart()

我们知道，在系统初始化的时候，系统会帮我们自动创建一个软件定时器任务，在这个任务中，
如果暂时没有运行中的定时器，任务会进入阻塞状态等待定时器任务节拍的信号量。我们在创建一个
软件定时器之后，如果没有启动它，该定时器就不会被添加到软件定时器列表中，那么在定时器任务
就不会运行该定时器，而 OSTmrStart() 函数就是将已经创建的软件定时器添加到定时器列表中，
这样子被创建的定时器就会被系统运行，其源码具体见 代码清单: 定时器-5。

列表 4: 代码清单: 定时器-5 OSTmrStart() 源码

```c
CPU_BOOLEAN OSTmrStart (OS_TMR *p_tmr, (1)    //定时器控制块指针
        OS_ERR *p_err) (2)     //返回错误类型
{
```

论坛: https://www.firebbs.cn/ 618  天猫: https://yehuosm.tmall.com
OS_ERR err;
CPU_BOOLEAN success; //存储函数执行结果

#endif OS_SAFETY_CRITICAL //如果启用（默认禁用）了安全检测
    if (p_err == (OS_ERR *)0) //如果错误类型实参为空
        { //执行安全检测异常函数
            OS_SAFETY_CRITICAL_EXCEPTION();
            return (DEF_FALSE); //返回 DEF_FALSE，不继续执行
        }
#endif

#if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u
    //如果启用（默认启用）了中断中非法调用检测
    if (OSIntNestingCtr > (OS_NESTING_CTR)0) //如果该函数是在中断中被调用
        { //错误类型为“在中断函数中定时”
            *p_err = OS_ERR_TMR_ISR;
            return (DEF_FALSE); //返回 DEF_FALSE，不继续执行
        }
#endif

#if OS_CFG_ARG_CHK_EN > 0u //如果启用（默认启用）了参数检测
    (p_tmr == (OS_TMR *)0) //如果启用 p_tmr 的实参为空
        { //错误类型为“无效的定时器”
            *p_err = OS_ERR_TMR_INVALID;
            return (DEF_FALSE); //返回 DEF_FALSE，不继续执行
        }
#endif

#if OS_CFG_OBJ_TYPE_CHK_EN > 0u //如果启用（默认启用）了对象类型检测
    if (p_tmr->Type != OS_OBJ_TYPE_TMR) //如果该定时器的对象类型有误
        { //错误类型为“无效的定时器”
            *p_err = OS_ERR_TMR_INVALID;
            return (DEF_FALSE); //返回 DEF_FALSE，不继续执行
        }
#endif
```c
    {
        *p_err = OS_ERR_OBJ_TYPE;        // 错误类型为“对象类型错误”
        return (DEF_FALSE);              // 返回 DEF_FALSE，不继续执行
    }
#endif

OSSchedLock(&err);                    // 锁住调度器
switch (p_tmr->State) (3)// 根据定时器的状态分类处理
{
    case OS_TMR_STATE_RUNNING:        // 如果定时器正在运行，则重启
        OS_TmrUnlink(p_tmr);        (5)// 从定时器列表中移除该定时器
        OS_TmrLink(p_tmr, OS_OPT_LINK_DLY); (4)// 将该定时器重新插入定时器列表
        OSSchedUnlock(&err);        // 解锁调度器
        *p_err = OS_ERR_NONE;        // 错误类型为“无错误”
        success = DEF_TRUE;         // 执行结果暂为 DEF_TRUE
        break;

    case OS_TMR_STATE_STOPPED:        // 如果定时器已被停止，则开启
    case OS_TMR_STATE_COMPLETED:      (6)// 如果定时器已完成了，则开启
        OS_TmrLink(p_tmr, OS_OPT_LINK_DLY); // 将该定时器重新插入定时器列表
        OSSchedUnlock(&err);            // 解锁调度器
        *p_err = OS_ERR_NONE;           // 错误类型为“无错误”
        success = DEF_TRUE;            // 执行结果暂为 DEF_TRUE
        break;

    case OS_TMR_STATE_UNUSED:         (7)// 如果定时器未被创建
        OSSchedUnlock(&err);          // 解锁调度器
        *p_err = OS_ERR_TMR_INACTIVE;  // 错误类型为“定时器未激活”
        success = DEF_FALSE;          // 执行结果暂为 DEF_FALSE
        break;

    default:                 (8)// 如果定时器的状态超出预期
        OSSchedUnlock(&err);      // 解锁调度器
        *p_err = OS_ERR_TMR_INACTIVE;  // 错误类型为“定时器未激活”
        success = DEF_FALSE;         // 执行结果暂为 DEF_FALSE
        break;
```
OSSchedUnlock(&err); //解锁调度器
*p_err = OS_ERR_TMR_INVALID_STATE; //错误类型为“定时器无效”
success = DEF_FALSE; //执行结果暂为 DEF_FALSE
break;
}
return (success); //返回执行结果

• 代码清单: 定时器-5 (1): 定时器控制块指针，指向要启动的软件定时器。

• 代码清单: 定时器-5 (2): 保存返回错误类型。

• 代码清单: 定时器-5 (3): 锁定调度器，因为接下来的操作是需要操作定时器列表的，此时
  应该锁定调度器，不被其他任务打扰，然后根据定时器的状态分类处理。

注: 源码中先看插入函数再看删除函数，代码清单: 定时器-5 (4)(5) 顺序在讲解中颠倒的。

• 代码清单: 定时器-5 (4): 在然后移除之后需要将软件定时器重新按照周期插入定时器列表
  中，调用 OS_TmrLink() 函数即可将软件定时器插入定时器列表，其源码具体见代码清单: 定时器-6。

列表 5: 代码清单: 定时器-6OS_TmrLink() 源码

```c
void OS_TmrLink (OS_TMR *p_tmr, (1) OS_OPT opt) (2) { //定时器控制块指针

  OS_TMR_SPOKE *p_spoke;
  OS_TMR *p_tmr0;
  OS_TMR *p_tmr1;
  OS_TMR_SPOKE_IX spoke;

  //重置定时器为运行状态
  p_tmr->State = OS_TMR_STATE_RUNNING;
}
```

(下页继续)
if (opt == OS_OPT_LINK_PERIODIC)
{
    // 如果定时器是再次插入，匹配时间加个周期重载值
    p_tmr->Match = p_tmr->Period + OSTmrTickCtr;(3)
}
else
{
    // 如果定时器是首次插入
    if (p_tmr->Dly == (OS_TICK)0)
    {
        // 如果定时器的 Dly = 0，匹配时间加个 Dly
        p_tmr->Match = p_tmr->Dly + OSTmrTickCtr;(4)
    }
    else
    {
        // 如果定时器的 Dly != 0，匹配时间加个 Dly
        p_tmr->Match = p_tmr->Dly + OSTmrTickCtr;(5)
    }
}

// 通过哈希算法决定将该定时器插入定时器轮的哪个列表。
spoke = (OS_TMR_SPOKE_IX)(p_tmr->Match % OSCfg_TmrWheelSize);(6)
p_spoke = &OSCfg_TmrWheel[spoke];

if (p_spoke->FirstPtr == (OS_TMR *))0)(7)
{
    // 如果列表为空，直接将该定时器作为列表的第一个元素。
    p_tmr->NextPtr = (OS_TMR *)0;
    p_tmr->PrevPtr = (OS_TMR *)0;
    p_spoke->FirstPtr = p_tmr;
    p_spoke->NbrEntries = 1u;
}

else
{
    //如果列表非空，算出定时器 p_tmr 的剩余时间
    p_tmr->Remain = p_tmr->Match - OSTmrTickCtr; (8)
    //取列表的首元素到 p_tmr1
    p_tmr1 = p_spoke->FirstPtr; (9)
    while (p_tmr1 != (OS_TMR *))
    {
        //如果 p_tmr1 非空，算出 p_tmr1 的剩余时间
        p_tmr1->Remain = p_tmr1->Match - OSTmrTickCtr; (10)
        if (p_tmr1->Remain > p_tmr1->Remain)
        {
            //如果 p_tmr 的剩余时间大于 p_tmr1 的
            if (p_tmr1->NextPtr != (OS_TMR *))
            {
                //如果 p_tmr1 后面非空，取 p_tmr1 后一个定时器为新的 p_tmr1 进行下一次循环
                p_tmr1 = p_tmr1->NextPtr; (11)
            }
        }
        else
        {
            //如果 p_tmr1 后面为空，将 p_tmr 插到 p_tmr1 的后面，结束循环
            p_tmr->NextPtr = (OS_TMR *)0;
            p_tmr->PrevPtr = p_tmr1;
            p_tmr1->NextPtr = p_tmr;
            p_tmr1 = (OS_TMR *)0; (12)
        }
    }
}
if (p_tmr1->PrevPtr == (OS_TMR *)0)  (13)
{
    //将p_tmr插入p_tmr1的前一个，结束循环。
    p_tmr->PrevPtr = (OS_TMR *)0;
    p_tmr->NextPtr = p_tmr1;
    p_tmr1->PrevPtr = p_tmr;
    p_spoke->FirstPtr = p_tmr;
}
else
{
    p_tmr0 = p_tmr1->PrevPtr;
    p_tmr->PrevPtr = p_tmr0;
    p_tmr->NextPtr = p_tmr1;
    p_tmr0->NextPtr = p_tmr;
    p_tmr1->PrevPtr = p_tmr;  (14)
}
p_tmr1 = (OS_TMR *)0;
}
//列表元素成员数加1
p_spoke->NbrEntries++;  (15)
}
if (p_spoke->NbrEntriesMax < p_spoke->NbrEntries)
{
    //更新列表成员数最大值历史记录
    p_spoke->NbrEntriesMax = p_spoke->NbrEntries;  (16)
}

• 代码清单: 定时器-6 (1): 定时器控制块指针。
• 代码清单: 定时器-6 (2): 插入定时器列表中的选项。
[野火]uCOS-III 内核实现与应用开发实战指南

- **代码清单: 定时器-6 (3)**：重置定时器为运行状态，如果定时器是再次插入，肯定是周期性定时器，延时时间为 Period，定时器的匹配时间（唤醒时间）Match 等于周期重载值 Period 加上当前的定时器计时时间。

- **代码清单: 定时器-6 (4)**：如果定时器是首次插入，如果定时器的延时时间 Dly 等于 0，定时器的匹配时间 Match 也等于周期重载值加上当前的定时器计时时间。

- **代码清单: 定时器-6 (5)**：而如果定时器的 Dly 不等于 0，定时器的匹配时间 Match 则等于 Dly 的值加上当前的定时器计时时间。

- **代码清单: 定时器-6 (6)**：通过哈希算法决定将该定时器插入定时器的哪个列表，这与第一部分讲解的时基列表很像。既然是哈希算法，开始插入的时候也要根据余数进行操作，根据软件定时器的到达时间（匹配时间或者称为唤醒时间也可以）对 OSCfg_TmrWheelSize 的余数取出 OSCfg_TmrWheel[OS_CFG_TMR_WHEEL_SIZE] 中对应的定时器列表记录，然后将定时器插入对应的列表中。

- **代码清单: 定时器-6 (7)**：如果定时器列表为空，直接将该定时器作为列表的第一个元素。

- **代码清单: 定时器-6 (8)**：如果列表非空，算出定时器 p_tmr 的剩余时间，按照即将唤醒的时间插入定时器列表中。

- **代码清单: 定时器-6 (9)**：取列表的首个元素到 p_tmr1，遍历定时器列表。

- **代码清单: 定时器-6 (10)**：如果 p_tmr1 非空，算出 p_tmr1 的剩余时间，对比 p_tmr 与 p_tmr1 的时间，按照升序进行插入列表中。

- **代码清单: 定时器-6 (11)**：如果 p_tmr 的剩余时间大于 p_tmr1 的，取 p_tmr1 后一个定时器作为新的 p_tmr1 进行下一次循环，直到 p_tmr 找到合适的地方就插入定时器列表。

- **代码清单: 定时器-6 (12)**：如果 p_tmr1 后面为空，将 p_tmr 插到 p_tmr1 的后面，结束循环。这些插入操作都是双向链表的插入操作，此处就不重复赘述。

- **代码清单: 定时器-6 (13)**：如果 p_tmr 的剩余时间不大于 p_tmr1 的，并且 p_tmr1 的前一个定时器为空，就直接将 p_tmr 插入 p_tmr1 的前一个位置，并且软件定时器列表的第一个定时器就是 p_tmr。

- **代码清单: 定时器-6 (14)**：而如果的上一个不为空，将 p_tmr 插入 p_tmr1 的前一个位置。

- **代码清单: 定时器-6 (15)**：对应定时器列表元素成员个数加 1。

论坛: https://www.firebbs.cn/  625  天猫: https://yehuosm.tmall.com
26.5.3 软件定时器列表管理

有些情况下，当系统中有多个软件定时器的时候，μC/OS 可能要维护上百个定时器。使用定时器
列表会大大降低更新定时器列表所占用的 CPU 时间，一个一个检测是否到期效率很低，有没有
什么办法让系统快速查找到到期的软件定时器？μC/OS 对软件定时器列表的管理就跟时间节拍
一样，采用哈希算法。OS_TmrLink 将不同的定时器变量根据其对 OSCfg_TmrWheelSize 余数的不
同插入数组 OSCfg_TmrWheel[OS_CFG_TMR_WHEEL_SIZE] 中，μC/OS 的软件定时器列表示
意图具体见图定时器列表。

定时器列表中包含了 OS_CFG_TMR_WHEEL_SIZE 条记录，该值是一个宏定义，由用户指定，在
os_cfg_app.h 文件中。能记录定时器的多少仅限于处理器的 RAM 空间，推荐的设置值为定时器
数 /4。定时器列表的每个记录都由 3 部分组成：NbrEntriesMax 表明该记录中有多少个定时器；
NbrEntriesMax 表明该记录中最大时存放了多少个定时器；FirstPtr 指向当前记录的定时器链表。
可能这样子讲述的不够清晰，下面举个例子来讲述软件定时器采用哈希算法插入对应的定时器
列表中的过程。

如图插入一个定时器 所示，我们先假定此时的定时器列表是空的，设置的宏定义

论坛：https://www.firebbs.cn/ 626 天猫：https://yehuosm.tmall.com


列表 6: 代码清单: 定时器-7 OS_TmrUnlink() 源码

```c
void OS_TmrUnlink (OS_TMR *p_tmr) (1) //定时器控制块指针
{
    OS_TMR_SPOKE   *p_spoke;
    OS_TMR         *p_tmr1;
    OS_TMR         *p_tmr2;
    OS_TMR_SPOKE_IX spoke;

    spoke = (OS_TMR_SPOKE_IX)(p_tmr->Match % OSCfg_TmrWheelSize); //与插入时一样，通过哈希算法找出
    p_spoke = &OSCfg_TmrWheel[spoke]; (2) //该定时器在定时器的哪个列表。

    if (p_spoke->FirstPtr == p_tmr) (3) //如果 p_tmr 是列表的首个元素
    {
        //取 p_tmr 后一个元素为 p_tmr1(可能为空)
    }

    OS_TMR_SPOKE_IX p_spoke2 = p_spoke->FirstPtr;
    do
    {
        if (p_spoke2 == p_tmr)
        {
            p_spoke2 = p_spoke2->FirstPtr;
        }
        else
        {
            p_spoke2 = p_spoke2->NextPtr;
        }
    }
    while (p_spoke2 != p_tmr);

    OS_TMR_SPOKE p_spoke3 = p_spoke;
    do
    {
        if (p_spoke3->FirstPtr == p_tmr)
        {
            p_spoke3 = p_spoke3->FirstPtr;
        }
        else
        {
            p_spoke3 = p_spoke3->NextPtr;
        }
    }
    while (p_spoke3 != p_tmr);

    //取 p_tmr 前一个元素为 p_tmr2(可能为空)

    p_tmr = p_tmr1;
}
```

(下页继续)
p_tmr1 = (OS_TMR *)p_tmr->NextPtr;
p_spoke->FirstPtr = (OS_TMR *)p_tmr1;    //表首改为 p_tmr1
if (p_tmr1 != (OS_TMR *)0) //如果 p_tmr1 确定非空
{
    p_tmr1->PrevPtr = (OS_TMR *)0;     //p_tmr1 的前面清空
}
}
else(4) //如果 p_tmr 不是列表的首个元素
{
    //将 p_tmr 从列表移除，并将 p_tmr 前后的两个元素连接在一起.
    p_tmr1 = (OS_TMR *)p_tmr->PrevPtr;

    p_tmr2 = (OS_TMR *)p_tmr->NextPtr;
    p_tmr1->NextPtr = p_tmr2;
    if (p_tmr2 != (OS_TMR *)0)
    {
        p_tmr2->PrevPtr = (OS_TMR *)p_tmr1;
    }
}
p_tmr->State = OS_TMR_STATE_STOPPED;    //复位 p_tmr 的指标
p_tmr->NextPtr = (OS_TMR *)0;
p_tmr->PrevPtr = (OS_TMR *)0;
p_spoke->NbrEntries--;               (5) //列表元素成员减 1

- 代码清单: 定时器-7 (1): 定时器控制块指针，指向要移除的定时器。
- 代码清单: 定时器-7 (2): 与插入时一样，通过哈希算法找出该定时器在定时器的哪个列表。
- 代码清单: 定时器-7 (3): 如果 p_tmr 是列表的首个元素，取 p_tmr 后一个元素为 p_tmr1(可能为空)，软件定时器列表头部的定时器改为 p_tmr1，如果 p_tmr1 确定非空，那就将 p_tmr 删除（p_tmr1 的前一个定时器就是 p_tmr)。
- 代码清单: 定时器-7 (4): 如果 p_tmr 不是列表的首个元素，那就将 p_tmr 从列表移除，并将
p_tmr 前后的两个元素连接在一起，这其实是双向链表的操作。

- **代码清单: 定时器-7 (5)**：清除定时器 p_tmr 的相关信息，定时器列表元素成员减 1。
- **代码清单: 定时器-5 (6)**：如果定时器已创建完成了，则开启即可，开启也是将定时器按照周期插入定时器列表中。
- **代码清单: 定时器-5 (7)**：如果定时器未被创建，那是不可能开启定时器的，使用会返回错误类型为“定时器未激活”的错误代码，用户需要先创建软件定时器再来开启。
- **代码清单: 定时器-5 (8)**：如果定时器的状态超出预期，返回错误类型为“定时器无效”的错误代码。

至此，软件定时器的启动函数就讲解完毕，我们在创建一个软件定时器后可以调用 OSTmrStart() 函数启动它，软件定时器启动函数的使用实例具体见 **代码清单: 定时器-8**。

**列表 7: 代码清单: 定时器-8OSTmrStart() 使用实例**

```c
void OS_TmrUnlink (OS_TMR *p_tmr) (1) //定时器控制块指针
{
    OS_TMR_SPOKE *p_spoke;
    OS_TMR *p_tmr1;
    OS_TMR *p_tmr2;
    OS_TMR_SPOKE_IX spoke;

    spoke = (OS_TMR_SPOKE_IX)(p_tmr->Match % OSCfg_TmrWheelSize); //与插入时一样，通过哈希算法找出
    p_spoke = &OSCfg_TmrWheel[spoke]; //该定时器在定时器的哪个列表。
    if (p_spoke->FirstPtr == p_tmr) (3) //如果 p_tmr 是列表的第一个元素
    {
        //取 p_tmr 后一个元素为 p_tmr1(可能为空)
        p_tmr1 = (OS_TMR *)p_tmr->NextPtr;
...
```

(下页继续)
26.5.4 停止定时器函数 OSTmrStop()

OSTmrStop() 函数用于停止一个软件定时器。软件定时器被停掉之后可以调用 OSTmrStart() 函数重启，但是重启之后定时器是从头计时，而不是接着上次停止的时刻继续计时。OSTmrStop() 源码具体见代码清单: 定时器-9。
列表 8: 代码清单: 定时器-OSTmrStop() 源码

```c
CPU_BOOLEAN OSTmrStop (OS_TMR *p_tmr, (1)//定时器控制块指针
    OS_OPT opt, (2)//选项
void *p_callback_arg, (3)//传给回调函数的新参数
    OS_ERR *p_err) (4)//返回错误类型
{
    OS_TMR_CALLBACK_PTR p_fnct;
    OS_ERR err;
    CPU_BOOLEAN success; //函数执行结果

    #ifdef OS_SAFETY_CRITICAL//如果启用（默认禁用）了安全检测
        if (p_err == (OS_ERR *)0) //如果错误类型为实参为空
        {
            OS_SAFETY_CRITICAL_EXCEPTION(); //执行安全检测异常函数
            return (DEF_FALSE); //返回 DEF_FALSE，不继续执行
        }
    #endif

    #if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u
    //如果启用（默认启用）了中断中非法调用检测
        if (OSIntNestingCtr > (OS_NESTING_CTR)0) //如果该函数是在中断中被调用
        {
            *p_err = OS_ERR_TMR_ISR; //错误类型为“在中断函数中定时”
            return (DEF_FALSE); //返回 DEF_FALSE，不继续执行
        }
    #endif

    #if OS_CFG_ARG_CHK_EN > 0u//如果启用（默认启用）了参数检测
        if (p_tmr == (OS_TMR *)0) //如果启用 p_tmr 的实参为空
            //...
    
    #endif

    //...
```

(下页继续)


```c
{  *p_err = OS_ERR_TMR_INVALID; //错误类型为“无效的定时器”
    return (DEF_FALSE);        //返回 DEF_FALSE，不继续执行
}
#endif

#if OS_CFG_OBJ_TYPE_CHK_EN > 0u //如果启用了（默认启用）了对象类型检测
    if (p_tmr->Type != OS_OBJ_TYPE_TMR)  //如果该定时器的对象类型有误
        {
        *p_err = OS_ERR_OBJ_TYPE;       //错误类型为“对象类型错误”
        return (DEF_FALSE);            //返回 DEF_FALSE，不继续执行
    }
#endif

OSSchedLock(&err);  //锁住调度器
switch (p_tmr->State)  (5)
{
    //根据定时器的状态分类处理
    case OS_TMR_STATE_RUNNING:   (6)
        //如果定时器正在运行
        OS_TmrUnlink(p_tmr);
        //从定时器队列表中移除该定时器
        *p_err = OS_ERR_NONE;
        //错误类型为“无错误”
        switch (opt)
        {
            //根据选项分类处理
            case OS_OPT_TMR_CALLBACK: (7)
                //执行回调函数，使用创建定时器时的实参
                p_fnct = p_tmr->CallbackPtr;
                //获取定时器的回调函数
                if (p_fnct != (OS_TMR_CALLBACK_PTR)0)
```
(续上页)

```c
63 {
64     // 如果回调函数存在
65     (*p_fnct)((void *)p_tmr, p_tmr->CallbackPtrArg);
66     // 使用创建定时器时的实参执行回调函数
67 }
68 else
69 {
70     // 如果回调函数不存在
71     *p_err = OS_ERR_TMR_NO_CALLBACK;
72     // 错误类型为“定时器没有回调函数”
73 }
74 break;
75
76 case OS_OPT_TMR_CALLBACK_ARG: (9)
77     // 执行回调函数，使用 p_callback_arg 作为实参
78     p_fnct = p_tmr->CallbackPtr;
79     // 取定时器的回调函数
80     if (p_fnct != (OS_TMR_CALLBACK_PTR)0)
81 {
82         // 如果回调函数存在
83         (*p_fnct)((void *)p_tmr, p_callback_arg);
84         // 使用 p_callback_arg 作为实参执行回调函数
85 }
86 else
87 {
88         // 如果回调函数不存在
89         *p_err = OS_ERR_TMR_NO_CALLBACK;
90         // 错误类型为“定时器没有回调函数”
91 }
92 break;
93
94 case OS_OPT_TMR_NONE: // 只需停掉定时器
(下页继续)
```

论坛: https://www.firebbs.cn/ 634 天猫: https://yehuosm.tmall.com
break;

default: (10) //情况超出预期
OSSchedUnlock(&err);  //解锁调度器
*p_err = OS_ERR_OPT_INVALID; //错误类型为“选项无效”
return (DEF_FALSE); //返回 DEF_FALSE，不继续执行
}
OSSchedUnlock(&err);
success = DEF_TRUE;
break;

case OS_TMR_STATE_COMPLETED: (11)
//如果定时器已完成第一次定时

case OS_TMR_STATE_STOPPED:
//如果定时器已被停止
OSSchedUnlock(&err);  //解锁调度器
*p_err = OS_ERR_TMR_STOPPED; //错误类型为“定时器已被停止”
success = DEF_TRUE; //执行结果暂为 DEF_TRUE
break;

case OS_TMR_STATE_UNUSED: (12)
//如果该定时器未被创建过
OSSchedUnlock(&err);  //解锁调度器
*p_err = OS_ERR_TMR_INACTIVE; //错误类型为“定时器未激活”
success = DEF_FALSE; //执行结果暂为 DEF_FALSE
break;

default: (13) //如果定时器状态超出预期
OSSchedUnlock(&err);  //解锁调度器
*p_err = OS_ERR_TMR_INVALID_STATE; //错误类型为“定时器状态非法”
success = DEF_FALSE; //执行结果暂为 DEF_FALSE
break;
```c
) return (success); //返回执行结果
```

- **代码清单: 定时器-9 (1)**: 定时器控制块指针，指向要停止的定时器。
- **代码清单: 定时器-9 (2)**: 停止的选项。
- **代码清单: 定时器-9 (3)**: 传给回调函数的新参数。
- **代码清单: 定时器-9 (4)**: 保存返回的错误类型。
- **代码清单: 定时器-9 (5)**: 锁定调度器，然后根据定时器的状态分类处理。
- **代码清单: 定时器-9 (6)**: 如果定时器正在运行，那么就调用 OS_TmrUnlink() 函数将该定时器从定时器列表中移除。
- **代码清单: 定时器-9 (7)**: 根据选项分类处理，如果需要执行回调函数，并且使用创建定时器时的实参，那就取定时器的回调函数，如果回调函数存在，就根据创建定时器指定的实参执行回调函数。
- **代码清单: 定时器-9 (8)**: 如果回调函数不存在，返回错误类型为“定时器没有回调函数”的错误代码。
- **代码清单: 定时器-9 (9)**: 如果需要执行回调函数，但是却是使用 p_callback_arg 作为实参，先取定时器的回调函数，如果回调函数存在就将 p_callback_arg 作为实参传递进去并且执行回调函数，否则就返回错误类型为“定时器没有回调函数”的错误代码。
- **代码清单: 定时器-9 (10)**: 如果情况超出预期，返回错误类型为“选项无效”的错误代码。
- **代码清单: 定时器-9 (11)**: 如果定时器已完成第一次定时或者如果定时器已被停止，返回错误类型为“定时器已被停止”的错误代码。
- **代码清单: 定时器-9 (12)**: 如果该定时器未被创建过，返回错误类型为“定时器未激活”的错误代码。
- **代码清单: 定时器-9 (13)**: 如果定时器状态超出预期，返回错误类型为“定时器状态非法”的错误代码。
软件定时器停止函数的使用很简单，在使用该函数前需确认定时器已经开启，停止后的软件定时器可以通过调用定时器启动函数来重新启动，OSTmrStop() 函数的使用实例具体见代码清单: 定时器-10 。（

### 列表 9: 代码清单: 定时器-10 OSTmrStop() 使用实例

```c
OS_ERR err;
OS_TMR my_tmr; //声明软件定时器对象
OSTmrStop ((OS_TMR *) &my_tmr, //定时器控制块指针
(OS_OPT *) OS_OPT_TMR_NONE, //选项
(void *) "Timer Over!", //传递回调函数的新参数
(OS_ERR *) err); //返回错误类型
```

#### 26.5.5 删除软件定时器函数 OSTmrDel()

OSTmrDel() 用于删除一个已经被创建成功的软件定时器，删除之后就无法使用该定时器，并且定时器相应的信息也会被系清空。要想使用 OSTmrDel() 函数必须在头文件 os_cfg.h 中把宏 OS_CFG_TMR_DEL_EN 定义为 1，该函数的源码具体见代码清单: 定时器-11 。（

### 列表 10: 代码清单: 定时器-11 OSTmrDel() 源码

```c
#if OS_CFG_TMR_DEL_EN > 0u //如果启用了 OSTmrDel() 函数
CPU_BOOLEAN OSTmrDel (OS_TMR *p_tmr, (1) //定时器控制块指针
OS_ERR *p_err) (2) //返回错误类型
{
  OS_ERR err;
  CPU_BOOLEAN success; //暂存函数执行结果

#ifdef OS_SAFETY_CRITICAL //如果启用 按认禁用）了安全检测
  if (p_err == (OS_ERR *) 0) //如果错误类型实参为空
```

(下页继续)


```c
{
   OS_SAFETY_CRITICAL_EXCEPTION();  //执行安全检测异常函数
   return (DEF_FALSE);              //返回 DEF_FALSE，不继续执行
}
#endif

#if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u
//如果启用（默认启用）了中断中非法调用检测
   if (OSIntNestingCtr > (OS_NESTING_CTR)0)  //如果该函数是在中断中被调用
   {
      *p_err = OS_ERR_TMR_ISR;                 //错误类型为“在中断函数中定时”
      return (DEF_FALSE);                      //返回 DEF_FALSE，不继续执行
   }
#endif

#if OS_CFG_ARG_CHK_EN > 0u//如果启用（默认启用）了参数检测
   if (p_tmr == (OS_TMR *)0)                  //如果启用 p_tmr 的实参为空
   {
      *p_err = OS_ERR_TMR_INVALID;            //错误类型为“无效的定时器”
      return (DEF_FALSE);                     //返回 DEF_FALSE，不继续执行
   }
#endif

#if OS_CFG_OBJ_TYPE_CHK_EN > 0u//如果启用（默认启用）了对象类型检测
   if (p_tmr->Type != OS_OBJ_TYPE_TMR)       //如果该定时器的对象类型有误
   {
      *p_err = OS_ERR_OBJ_TYPE;               //错误类型为“对象类型错误”
      return (DEF_FALSE);                     //返回 DEF_FALSE，不继续执行
   }
#endif

OSSchedLock(&err);                     //锁住调度器
```

(续上页)
# if OS_CFG_DBG_EN > 0u //如果启用（默认启用）了调试代码和变量
    OS_TmrDbgListRemove(p_tmr); //将该定时从定时器双向调试链表移除
#endif

OSTmrQty--; (3)  //定时器个数减 1

switch (p_tmr->State) //根据定时器的状态分类处理
{
    case OS_TMR_STATE_RUNNING: //如果定时器正在运行
        OS_TmrUnlink(p_tmr); (4) //从当前定时器列表列表移除定时器
        OS_TmrClr(p_tmr); (5) //复位定时器的指标
        OSSchedUnlock(&err); //解锁调度器
        *p_err = OS_ERR_NONE; //错误类型为“无错误”
        success = DEF_TRUE; //执行结果暂为 DEF_TRUE
        break;

    case OS_TMR_STATE_STOPPED: //如果定时器已被停止
    case OS_TMR_STATE_COMPLETED: //如果定时器已完成第一次定时
        OS_TmrClr(p_tmr); //复位定时器的指标
        OSSchedUnlock(&err); //解锁调度器
        *p_err = OS_ERR_NONE; //错误类型为“无错误”
        success = DEF_TRUE; //执行结果暂为 DEF_TRUE
        break;

    case OS_TMR_STATEUNUSED: //如果定时器已被删除
        OSSchedUnlock(&err); //解锁调度器
        *p_err = OS_ERR_TMR_INACTIVE; //错误类型为“定时器未激活”
        success = DEF_FALSE; //执行结果暂为 DEF_FALSE
        break;

    default: //如果定时器的状态超出预期
        OSSchedUnlock(&err); //解锁调度器
        *p_err = OS_ERR_TMR_INVALID_STATE; //错误类型为“定时器无效”
        break;
}
success = DEF_FALSE; //执行结果暂为 DEF_FALSE
break;
}
return (success); //返回执行结果
#endif

- 代码清单: 定时器-I1 (1): 定时器控制块指针，指向要删除的软件定时器。
- 代码清单: 定时器-I1 (2): 用于保存返回的错误类型。
- 代码清单: 定时器-I1 (3): 如果程序能执行到这里，说明能正常删除软件定时器，将系统的软件定时器个数减一。
- 代码清单: 定时器-I1 (4): 调用 OS_TmrUnlink() 函数从当前定时器列表移除定时器。
- 代码清单: 定时器-I1 (5): OS_TmrClr() 清除软件定时器控制块的相关信息，表示定时器删除完成。

软件定时器的删除函数使用很简单，具体见代码清单: 定时器-I2。
列表 11: 代码清单: 定时器-12 软件定时器删除函数
OSTmrDel() 使用实例

```c
OS_ERR err;
OS_TMR my_tmr;     //声明软件定时器对象
OSTmrDel((OS_TMR *)&my_tmr, //软件定时器对象
(OS_ERR *)&err);       //返回错误类型
```

本章讲解了那么多，是不是还是不知道软件定时器是怎么运作的，别担心，下面我们就来看看软件定时器是怎么运作的。

26.6 软件定时器任务

我们知道，软件定时器的回调函数的上下文是在任务中，所有，系统中必须要一个任务来管理所有的软件定时器，等到定时时间到达后就调用定时器对应的回调函数，那么软件定时器任务又是一个什么东西呢，它是在系统初始化的时候系统就帮我们创建的一个任务，具体见代码清单: 定时器-13 加粗部分。

列表 12: 代码清单: 定时器-13 创建软件定时器任务。

```c
void OS_TmrInit (OS_ERR *p_err)
{
    OS_TMR_SPOKE_IX i;
    OS_TMR_SPOKE    *p_spoke;

    #ifdef OS_SAFETY_CRITICAL
        if (p_err == (OS_ERR *)&OS_ERR_OK)
        {
```

(下页继续)
OS_SAFETY_CRITICAL_EXCEPTION();

    return;

} #endif

#if OS_CFG_DBG_EN > 0u
    OSTmrDbgListPtr = (OS_TMR *)0;
#endif

if (OSCfg_TmrTaskRate_Hz > (OS_RATE_HZ)0)    (1)
{
    OSTmrUpdateCnt = OSCfg_TickRate_Hz / OSCfg_TmrTaskRate_Hz;
}
else     (2)
{
    OSTmrUpdateCnt = OSCfg_TickRate_Hz / (OS_RATE_HZ)10;
}
OSTmrUpdateCtr = OSTmrUpdateCnt;

OSTmrTickCtr = (OS_TICK)0;

OSTmrTaskTimeMax = (CPU_TS)0;

for (i = 0u; i < OSCfg_TmrWheelSize; i++)    (3)
{
    p_spoke = &OSCfg_TmrWheel[i];
    p_spoke->NbrEntries = (OS_OBJ_QTY)0;
    p_spoke->NbrEntriesMax = (OS_OBJ_QTY)0;
    p_spoke->FirstPtr = (OS_TMR *)0;
}

/* ---------------- CREATE THE TIMER TASK --------------- */
if (OSCfg_TmrTaskStkBasePtr == (CPU_STK*)0)
{
    *p_err = OS_ERR_TMR_STK_INVALID;
    return;
}

if (OSCfg_TmrTaskStkSize < OSCfg_StkSizeMin)
{
    *p_err = OS_ERR_TMR_STK_SIZE_INVALID;
    return;
}

if (OSCfg_TmrTaskPrio >= (OS_CFG_PRIO_MAX - 1u))
{
    *p_err = OS_ERR_TMR_PRIO_INVALID;
    return;
}

OSTaskCreate((OS_TCB*)&OSTmrTaskTCB,
              (CPU_CHAR*)((void*)"μC/OS-III Timer Task"),
              (OS_TASK_PTR)OS_TmrTask,
              (void*)0,
              (OS_PRIO*)OSCfg_TmrTaskPrio,
              (CPU_STK*)OSCfg_TmrTaskStkBasePtr,
              (CPU_STK_SIZE)OSCfg_TmrTaskStkLimit,
              (CPU_STK_SIZE)OSCfg_TmrTaskStkSize,
              (OS_MSG_QTY)0,
              (OS_TICK)0,
              (void*)0,
              (OS_OPT)(OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR | OS_OPT_TASK_NO_TLS),
              (OS_ERR*)p_err);
• 代码清单: 定时器-13 (1): 正常来说定时器任务的执行频率 OSCfg_TmrTaskRate_Hz 是大于 0 的，并且能被 OSCfg_TickRate_Hz 整除，才能比较准确得到定时器任务运行的频率。如果 OSCfg_TmrTaskRate_Hz 设置为大于 0，就配置定时器任务的频率。

• 代码清单: 定时器-13 (2): 否则就配置为系统时钟频率的十分之一 (1/10)。不过设定的定时器的频率大于时节拍的执行频率的时候，定时器运行就会出错，但是这里没有进行判断，我们自己在写代码的时候注意一下即可。

我们举个例子，系统的 OSCfg_TickRate_Hz 是 1000，OSCfg_TmrTaskRate_Hz 是 10，那么计算得到 OSTmrUpdateCnt 就是 100，开始的时候 OSTmrUpdateCtr 是跟 OSTmrUpdateCnt 一样大的，都是 100，每当时钟节拍到来的时候 OSTmrUpdateCtr 就减 1，减到 0 的话就运行定时器任务，这样子就实现了从时间节拍中分频得到定时器任务频率。如果 OSCfg_TmrTaskRate_Hz 不能被 OSCfg_TickRate_Hz 整除，比如 OSCfg_TickRate_Hz 设置为 1000，OSCfg_TmrTaskRate_Hz 设置为 300，这样子设置是想要定时器任务执行频率是 300Hz，但是 OSTmrUpdateCnt 计算出来是 3，这样子定时器任务执行频率大约就是 330Hz，定时的单位本来想设置为 3.3ms，可实际运行的单位却是 3ms，这样子肯定导致定时器不是很精确的，这些处理还是需要我们根据实际情况进行调整的。

• 代码清单: 定时器-13 (3): 利用 for 循环初始化定时器列表。

• 代码清单: 定时器-13 (4): 创建 OS_TmrTask 任务。

我们来看看定时器任务是在做什么事情，OS_TmrTask() 源码具体见代码清单: 定时器-14。

列表 13: 代码清单: 定时器-14 OS_TmrTask() 源码

```c
void OS_TmrTask (void *p_arg)
{
    CPU_BOOLEAN done;
    OS_ERR err;
    OS_TMR_CALLBACK_PTR p_fnct;
    OS_TMR_SPOKE *p_spoke;
}
```

(下页继续)
OS_TMR *p_tmr;
OS_TMR *p_tmr_next;
OS_TMR_SPOKE_IX spoke;
CPU_TS ts;
CPU_TS ts_start;
CPU_TS ts_end;
p_arg = p_arg; /* Not using 'p_arg', prevent compiler warning */

while (DEF_ON)
{
    /* 等待信号指示更新定时器的时间 */
    (void)OSTaskSemPend((OS_TICK )0,
                        (OS_OPT )OS_OPT_PEND_BLOCKING,
                        (CPU_TS *)&ts,
                        (OS_ERR *)&err);  (1)

    OSSchedLock(&err);
    ts_start = OS_TS_GET();
    /* 增加当前定时器时间 */
    OSTmrTickCtr++; (2)

    /* 通过哈希算法找到对应时间唤醒的列表 */
    spoke = (OS_TMR_SPOKE_IX)(OSTmrTickCtr % OSCfg_TmrWheelSize);
    p_spoke = &OSCfg_TmrWheel[spoke]; (3)

    /* 获取列表头部的定时器 */
    p_tmr = p_spoke->FirstPtr; (4)

    done = DEF_FALSE;
    while (done == DEF_FALSE)
    {

    (下页继续)
if (p_tmr != (OS_TMR *)0) {

    /* 指向下—个定时器以进行更新，
因为可能当前定时器到时了会从列表中移除 */
    p_tmr_next = (OS_TMR *)p_tmr->NextPtr;

    /* 确认是定时时间到达 */
    if (OSTmrTickCtr == p_tmr->Match) {
        /* 先移除定时器 */
        OS_TmrUnlink(p_tmr);

        /* 如果是周期定时器 */
        if (p_tmr->Opt == OS_OPT_TMR_PERIODIC) {
            /* 重新按照唤醒时间插入定时器列表 */
            OS_TmrLink(p_tmr,
                        OS_OPT_LINK_PERIODIC); (7)
        } else {

            /* 定时器状态设置为已完成 */
            p_tmr->State = OS_TMR_STATE_COMPLETED; (8)
        }

        /* 执行回调函数（如果可用）*/
        p_fnct = p_tmr->CallbackPtr;
        if (p_fnct != (OS_TMR_CALLBACK_PTR)0) {
            (*p_fnct)((void *)p_tmr,
                       p_tmr->CallbackPtrArg); (9)
        }
    }
/* 看看下一个定时器是否匹配 */
p_tmr = p_tmr_next;  
}  
else  
{  
done = DEF_TRUE;  
}  
}  
else  
{  
done = DEF_TRUE;  
}  
}  
/* 测量定时器任务的执行时间 */
ts_end = OS_TS_GET() - ts_start;  
OSSchedUnlock(&err);  
if (OSTmrTaskTimeMax < ts_end)  
{  
OSTmrTaskTimeMax = ts_end;  
}  
}  
}

• 代码清单: 定时器-14 (1): 调用 OSTaskSemPend() 函数在一直等待定时器跳拍的信号量，等到到了才运行，那定时器节拍是怎么样运行的呢，系统的时钟节拍是基于 SysTick 定时器上的，μC/OS 采用 Tick 任务 (OS_TickTask) 管理系统的时钟节拍，而我们定时器节拍是由系统节拍分频而来，那么其发送信号量的地方当然也是在 SysTick 中断服务函数中，但是 μC/OS 支持采用中断延迟，如果使用了中断延迟，那么发送任务信号量的地方就会在中断发布任务中 (OS_IntQTTask)，从代码中，我们可以看到当 OSTmrUpdateCtr 减到 0 的时候才会发送一次信号量，这也就是为什么我们的定时器节拍是基于系统时钟节拍分频而来的原因，具体见代码清单: 定时器-15 。

论坛: [https://www.firebbs.cn/](https://www.firebbs.cn/)  
天猫: [https://yehuosm.tmall.com](https://yehuosm.tmall.com)
注意：此处的信号量获取是任务信号量而非内核对象的信号量，在后文会讲解这种任务信号量，此处就先了解即可，与系统内核对象信号量是作用是一样的。

列表 14: 代码清单: 定时器-15 定时器任务的发送信号量

```c
// *************************** 在 SysTick 中断服务函数中 
***************************

#if OS_CFG_TMR_EN > 0u

// 如果启用 (默认启用) 了软件定时器
OSTmrUpdateCtr--;  // 软件定时器计数器自减

if (OSTmrUpdateCtr == (OS_CTR)0u)  // 如果软件定时器计数器减至 0
{
    OSTmrUpdateCtr = OSTmrUpdateCnt;  // 重载软件定时器计数器

    // 发送信号量给软件定时器任务 OS_TmrTask()
    OSTaskSemPost((OS_TCB *)OSTmrTaskTCB,
                   (OS_OPT ) OS_OPT_POST_NONE,
                   (OS_ERR *) &err);
}
#endif

// *************************** 在中断发布任务中 ***************************

#if OS_CFG_TMR_EN > 0u

OSTmrUpdateCtr--;  // 软件定时器计数器自减

if (OSTmrUpdateCtr == (OS_CTR)0u)
{
    OSTmrUpdateCtr = OSTmrUpdateCnt;

    ts = OS_TS_GET();  // 获取时间戳 */

    /* 发送信号量给软件定时器任务 OS_TmrTask() */
(void)OSTaskSemPost((OS_TCB *)OSTmrTaskTCB,
                   (OS_OPT ) OS_OPT_POST_NONE,
                   (OS_ERR *) &err);
}
```

(下页继续)
(CPU_TS ) ts,
(OS_ERR *) &err);

```c
#endif
```

- 代码清单: 定时器-14 (2): 当任务获取到信号量的时候，任务开始运行，增加当前定时器时间记录 OSTmrTickCtr。

- 代码清单: 定时器-14 (3): 通过哈希算法找到对应时间唤醒的列表，比如按照我们前面添加的定时器 1 与定时器 2，具体见图插入第二个定时器，当 OSTmrTickCtr 到达 13 的时候，通过哈希算法的运算之后，我们能得到 spoke 等于 4，这样子就直接找到我们插入的定时器列表了。

- 代码清单: 定时器-14 (4): 获取列表头部的定时器。

- 代码清单: 定时器-14 (5): 如果定时器列表中有定时器的话，就将 p_tmr_next 变量指向下一个定时器以准备进行更新，因为当前定时器可能到时了，就会从列表中移除。

- 代码清单: 定时器-14 (6): 如果当前定时器时间（OSTmrTickCtr）与定时器中的匹配时间（Match）是一样的，那么确认是到时时间已经到达。

- 代码清单: 定时器-14 (7): 不管三七二十一调用 OS_TmrUnlink() 函数移除定时器，如果该定时器是周期定时器，那么就调用 OS_TmrLink() 函数按照唤醒时间重新插入定时器列表。

- 代码清单: 定时器-14 (8): 否则就是单次定时器，那么将定时器状态设置为定时已完成。

- 代码清单: 定时器-14 (9): 如果回调函数存在，执行回调函数。

- 代码清单: 定时器-14 (10): 看看下一个定时器的定时时间是否也到达了，如果是就需要唤醒。

- 代码清单: 定时器-14 (11): 测量定时器任务的执行时间。

当定时器任务被执行，它首先递增 OSTmrTickCtr 变量，然后通过哈希算法决定哪个定时器列表需被更新。然后，如果这个定时器列表中存在定时器 (FirstPtr 不为 NULL)，系统会检查定时器中的匹配时间 Match 是否与当前定时器时间 OSTmrTickCtr 相等，如果相等，这个定时器会被移出该定时器，然后调用这个定时器的回调函数（假定这个定时器被创建时有回调函数），再根据
26.7 软件定时器实验

软件定时器实验是在μC/OS 中创建了一个应用任务 AppTaskTmr，在该任务中创建一个软件定时器，周期性定时 1s，每次定时完成切换 LED1 的亮灭状态，并且打印时间戳的计时，检验定时的精准度，具体见代码清单: 定时器-16。

```c
#include <includes.h>

CPU_TS ts_start; //时间戳变量
CPU_TS ts_end;

static OS_TCB AppTaskStartTCB; //任务控制块
static OS_TCB AppTaskTmrTCB;
static CPU_STK AppTaskStartStk[APP_TASK_START_STK_SIZE];  //任务栈
static CPU_STK AppTaskTmrStk [ APP_TASK_TMR_STK_SIZE ];

static void AppTaskStart (void *p_arg); //任务函数声明
static void AppTaskTmr  ( void * p_arg );
```

(下页继续)
```c
int main (void)
{
    OSS_ERR err;

    OSInit(&err);   // 初始化μC/OS-III

    /* 创建起始任务 */
    OSTaskCreate((OS_TCB *)&AppTaskStartTCB,
        // 任务控制块地址
        (CPU_CHAR *)"App Task Start",
        // 任务名称
        (OS_TASK_PTR ) AppTaskStart,
        // 任务函数
        (void *) 0,
        // 传给任务函数 (形参 p_arg) 的实参
        (OS_PRIO ) APP_TASK_START_PRIO,
        // 任务的优先级
        (CPU_STK *)&AppTaskStartStk[0],
        // 任务栈的地址
        (CPU_STK_SIZE) APP_TASK_START_STK_SIZE / 10,
        // 任务栈空间剩下 1/10 时限制其增长
        (CPU_STK_SIZE) APP_TASK_START_STK_SIZE,
        // 任务栈空间（单位：sizeof(CPU_STK))
        (OS_MSG_QTY ) 5u,
        // 任务可接收的最大消息数
        (OS_TICK ) 0u,
        // 任务的时间片节拍数 (0 表默认值 OSCfg_TickRate_Hz/10)
        (void *) 0,
        // 任务扩展 (0 表不扩展)
        (OS_OPT ) (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
```

（下页继续）
//任务选项
(OS_ERR *)&err);
//返回错误类型

OSStart(&err);
//启动多任务管理（交由 μC/OS-III 控制）
}

static void AppTaskStart (void *p_arg)
{
    CPU_INT32U cpu_clk_freq;
    CPU_INT32U cnts;
    OS_ERR err;

    (void)p_arg;
    //板级初始化
    BSP_Init();
    //初始化 CPU 组件（时间戳、关中断时间测量和主机名）
    CPU_Init();

    //获取 CPU 内核时钟频率（SysTick 工作时钟）
    cpu_clk_freq = BSP_CPU_ClkFreq();
    //根据用户设定的时钟节拍频率计算 SysTick 定时器的计数值
    cnts = cpu_clk_freq / (CPU_INT32U)OSCfg_TickRate_Hz;
    //调用 SysTick 初始化函数，设置定时器计数值和启动定时器
    OS_CPU_SysTickInit(cnts);

    //初始化内存管理组件（堆内存池和内存池表）
    Mem_Init();
}

(下页继续)
```c
#define OS_CFG_STAT_TASK_EN > 0u
OSStatTaskCPUUsageInit(&err);
#endif

CPU_IntDisMeasMaxCurReset();
//复位（清零）当前最大关中断时间

/* 创建 AppTaskTmr 任务 */
OSTaskCreate((OS_TCB    *@)&AppTaskTmrTCB,
    //任务控制块地址
    (CPU_CHAR    @)*"App Task Tmr",
    //任务名称
    (OS_TASK_PTR ) AppTaskTmr,
    //任务函数
    (void        @) 0,
    //传递给任务函数 (形参 p_arg) 的实参
    (OS_PRIO     ) APP_TASK_TMR_PRIO,
    //任务的优先级
    (CPU_STK      @)&AppTaskTmrStk[0],
    //任务栈的地址
    (CPU_STK_SIZE) APP_TASK_TMR_STK_SIZE / 10,
    //任务栈空间剩下 1/10 时限制其增长
    (CPU_STK_SIZE) APP_TASK_TMR_STK_SIZE,
    //任务栈空间（单点: sizeof(CPU_STK)
    (OS_MSG_QTY    ) 5u,
    //任务可接收的最大消息数
    (OS_TICK       ) 0u,
    //任务的时间片数（0 表默认值 OSCfg_TickRate_Hz/10）
    (void        @) 0,
    //任务扩展（0 表不扩展）
    (OS_OPT       ) (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
```

(下页继续)
```c
    // 任务选项
    (OS_ERR[*]&err);
    // 返回错误类型

    OSTaskDel(&AppTaskStartTCB,&err);
    // 删除起始任务本身，该任务不再运行
}

// 软件定时器 MyTmr 的回调函数
void TmrCallback(OS_TMR*p_tmr,void *p_arg)
{
    CPU_INT32U cpu_clk_freq;
    // 使用到临界段（在关/开中断时）时必须用到该宏。该宏声明和定义一

    // 个局部变量，用于保存关中断前的 CPU 状态寄存器

    // SR（临界段关中断只保存 SR），开中断时将该值还原。
    CPU_SR_ALLO();
    printf("%s",(char*)p_arg);

    cpu_clk_freq = BSP_CPU_ClkFreq();
    // 获取 CPU 时钟，时间戳是以该时钟计数

    macLED1_TOGGLE();

    ts_end = OS_TS_GET() - ts_start;
    // 获取定时后的的时间戳（以 CPU 时钟进行计数的一个计数值）

    // 并计算定时时间。
    OS_CRITICAL_ENTER();
    // 进入临界段，不希望下面串口打印遭到中断
```

(下页继续)
printf ( "\r\n定时 1s，通过时间戳测得定时 \%07d us，即 \%04d ms。\r\n", 
    ts_end / ( cpu_clk_freq / 1000000 ),  //将定时时间折算成 us
    ts_end / ( cpu_clk_freq / 1000 ) );  //将定时时间折算成 ms

OS_CRITICAL_EXIT();

    ts_start = OS_TS_GET(); /*获取定时器时间戳*/
}

static void AppTaskTmr ( void * p_arg )
{
    OS_ERR err;
    OS_TMR my_tmr;  //声明软件定时器对象

    (void)p_arg;

    /* 创建软件定时器 */
    OSTmrCreate ((OS_TMR *) &my_tmr,  //软件定时器对象
        (CPU_CHAR *)"MySoftTimer", //命名软件定时器
        (OS_TICK )10,
        //定时器初始值，10Hz 基本计算，即为 1s
        (OS_TICK )10,
        //定时器周期重装值，10Hz 基本计算，即为 1s
        (OS_OPT )OS_OPT_TMR_PERIODIC, //周期性定时
        (OS_TMR_CALLBACK_PTR )TmrCallback,  //回调函数
        (void)"Timer Over!", //传递实参给回调函数
        (OS_ERR *) err); //返回错误类型
26.8 软件定时器实验现象

程序编译好，用 USB 线连接计算机和开发板的 USB 接口（对应丝印为 USB 转串口），用 DAP 仿真器把配套程序下载到野火 STM32 开发板（具体型号根据购买的板子而定，每个型号的板子都配套有对应的程序），在计算机上打开串口调试助手，然后复位开发板就可以在调试助手中看到串口的打印信息，在串口调试助手中可以看到运行结果我们可以看到，每 1S 时间到的时候，软件定时器就会触发一次回调函数，具体见图软件定时器实验现象。

论坛：https://www.firebbs.cn/  656  天猫：https://yehuosm.tmall.com
26.9 总结

从一开始的定时器相关函数的使用和分析到后面定时器运作机制的分析，想必大家对定时器整个运作有了更深的了解。定时器的创建、删除、启动、停止这些操作无非就是在操作定时器列表的双向列表和根据不同的设置进行定时器状态的转化以及相关的处理。至于检测定时器到期，系统将时间节拍进行分数得到定时器任务执行的频率，在定时器任务中，系统采用了哈希算法进行快速检测有没有定时器到期，然后执行其对应的回调函数等操作。软件定时器最核心的一点是底层的一个硬件定时器（SysTick 内核定时器）上进行软件分频，这也是μC/OS 写的很好的一点，
大家也可以学习它的这种编程思想。

μC/OS 允许用户建立任意数量的定时器（只限制于处理器的 RAM 大小）。

回调函数是在定时器任务中被调用，所以回调函数的上下文环境是在任务中，并且运行回调函数时调度器处于被锁状态。一般我们编写的回调函数越简短越好，并且不能在回调函数中等待消息队列、信号量、事件等操作，否则定时器任务会被挂起，导致定时器任务崩溃，这是绝对不允许的。

此外我们还需要注意几点：

1. 回调函数是在定时器任务中被执行的，这意味着定时器任务需要有足够的栈空间供回调函数去执行。

2. 回调函数是在根据定时器队列中依次存放的，所以在定时器时间到达后回调函数是依次被执行的。

3. 定时器任务的执行时间决定于：有多少个定时器周期，执行定时器中的回调函数需多少时间。因为回调函数是由用户提供的，它可能很大程度上影响了定时器任务的执行时间。

4. 回调函数被执行时会锁调度器，所以我们必须让回调函数尽可能地短，以便其他任务能正常运行。
第 27 章 任务信号量

27.1 任务信号量的基本概念

μC/OS 提供任务信号量这个功能，每个任务都有一个 32 位（用户可以自定义位宽，我们使用 32 位的 CPU，此处就是 32 位）的信号量值 SemCtr，这个信号量值是在任务控制块中包含的，是任务独有的一个信号量通知值，在大多数情况下，任务信号量可以替代内核对象的二值信号量、计数信号量等。

注：本章主要讲解任务信号量，而非内核对象信号量，如非特别说明，本章中的信号量都指的是内核对象信号量。前面所讲的信号量是单独的内核对象，是独立于任务存在的；本章要讲述的任务信号量是任务特有的属性，紧紧依赖于一个特定任务。

相对于前面使用 μC/OS 内核通信的资源，必须创建二进制信号量、计数信号量等情况，使用任务信号量显然更灵活。因为使用任务信号量比通过内核对象信号量通信方式解除阻塞的任务的速度要快，并且更加节省 RAM 内存空间，任务信号量的使用无需单独创建信号量。

通过对任务信号量的合理使用，可以在一定场合下替代 μC/OS 的信号量，用户只需向任务内部的信号量发送一个信号而不用通过外部的信号量进行发送，这样子处理就会很方便而且更加高效，当然，凡事都有利弊，不然的话 μC/OS 还要内核的 IPC 通信机制干嘛，任务信号量虽然处理更快，RAM 开销更小，但也有限制：只能有一个任务接收任务信号量，因为必须指定接收信号量的任务，才能正确发送信号量；而内核对象的信号量则没有这个限制，用户在释放信号量，可以采用广播的方式，让所有等待信号量的任务都获取到信号量。

在实际任务间的通信中，一个或多个任务发送一个信号量给另一个任务是非常常见的，而一个任务给多个任务发送信号量的情况相对比较少。这种情况就很适合采用任务信号量进行传递信号，如果任务信号量可以满足设计需求，那么尽量不要使用普通信号量，这样子设计的系统会更加高效。

任务信号量的运作机制与普通信号量一样，没什么差别。

论坛：https://www.firebbs.cn  659  天猫：https://yehuosm.tmall.com
27.2 任务信号量的函数接口讲解

27.2.1 任务信号量释放函数 OSTaskSemPost()

函数 OSTaskSemPost() 用来释放任务信号量，虽然只有拥有任务信号量的任务才可以等待该任务信号量，但是其他所有的任务或者中断都可以向该任务释放信号量，其源码具体见代码清单: 任务信号量-1。

列表 1: 代码清单: 任务信号量-1OSTaskSemPost()

```c
OS_SEM_CTR  OSTaskSemPost (OS_TCB  *p_tcb, (1)  //目标任务
    OS_OPT   opt,      (2)  //选项
    OS_ERR   *p_err) (3)  //返回错误类型
{
    OS_SEM_CTR  ctr;
    CPU_TS      ts;

    #ifdef OS_SAFETY_CRITICAL //如果启用（默认禁用）了安全检测
    if (p_err == (OS_ERR  *)0)  //如果 p_err 为空
    {
        OS_SAFETY_CRITICAL_EXCEPTION();  //执行安全检测异常函数
        return ((OS_SEM_CTR)0);  //返回 0 (有错误)，停止执行
    }
    #endif

    #if OS_CFG_ARG_CHK_EN > 0u //如果启用（默认启用）了参数检测功能
    switch (opt)  //根据选项分类处理
    {
        case OS_OPT_POST_NONE: //如果选项在预期之内
        case OS_OPT_POST_NO_SCHED:
```

论坛: https://www.firebbs.cn/                       天猫: https://yehuosm.tmall.com
break; //跳出

default: //如果选项超出预期
    *p_err = OS_ERR_OPT_INVALID; //错误类型为“选项非法”
    return ((OS_SEM_CTR)0u); //返回 0 (有错误)，停止执行
#endif

ts = OS_TS_GET(); //获取时间戳
#if OS_CFG_ISR_POST_DEFERRED_EN > 0u //如果启用了中断延迟发布
    if (OSIntNestingCtr > (OS_NESTING_CTR)0) //如果该函数是在中断中被调用
    {
        OS_IntQPost((OS_OBJ_TYPE)OS_OBJ_TYPE_TASK_SIGNAL,
        //将信号量发布到中断消息队列
        (void *)p_tcb,
        (void *)0,
        (OS_MSG_SIZE)0,
        (OS_FLAGS)0,
        (OS_OPT)0,
        (CPU_TS)ts,
        (OS_ERR *)p_err); (4)
        return ((OS_SEM_CTR)0); //返回 0 (尚未发布)
    }
#endif

ctr = OS_TaskSemPost(p_tcb, //将信号量按照普通方式处理
    opt,
    ts,
    p_err); (5)

return (ctr); //返回信号的当前计数值

论坛: https://www.firebbs.cn/ 661 天猫: https://yehuosm.tmall.com
• 代码清单: 任务信号量-1 (1): 目标任务控制块指针，指向要释放任务信号量的任务。

• 代码清单: 任务信号量-1 (2): 释放任务信号量的选项。

• 代码清单: 任务信号量-1 (3): 用于返回保存错误代码。

• 代码清单: 任务信号量-1 (4): 如果启用了中断延迟发布，并且该函数在中断中被调用，那就将信号量发布到中断消息队列，由中断消息队列发布任务信号量。


列表 2: 代码清单: 任务信号量-2 OS_TaskSemPost() 源码

```c
OS_SEM_CTR OS_TaskSemPost (OS_TCB *p_tcb, OS_OPT opt, CPU_TS ts, OS_ERR *p_err) //目标任务
{  OS_SEM_CTR ctr;
    CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必须用到该宏，该宏声明和
    //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
    //SR (临界段关中断无需保存 SR)，开中断时将该值还原。
    OS_CRITICAL_ENTER();  //进入临界段
    if (p_tcb == (OS_TCB *) 0) //如果 p_tcb 为空
    {  p_tcb = OSTCBCurPtr; //将任务信号量发给自己 (任务)
    }
    p_tcb->TS = ts; //记录信号量被发布的时间戳
    *p_err = OS_ERR_NONE; //错误类型为“无错误”
    switch (p_tcb->TaskState) //返回错误类型
    {  //目标任务
    
    } //返回错误类型
    
```

// 跟吴目标任务的任务状态分类处理
{
    case OS_TASK_STATE_RDY:  // 如果目标任务没有等待状态
        //
        break;
    case OS_TASK_STATE_DLY:
        //
        break;
    case OS_TASK_STATE_SUSPENDED:
        //
        break;
    case OS_TASK_STATE_DLY_SUSPENDED:
        //
        switch (sizeof(OS_SEM_CTR))
        {
            case 1u:  // 号量计数值溢出，如
                if (p_tcb->SemCtr == DEF_INT_08U_MAX_VAL)  // 果溢出，则开中断，
                {
                    OS_CRITICAL_EXIT();  // 返错误类型为“计
                    *p_err = OS_ERR_SEM_OVF;
                    return ((OS_SEM_CTR)0);  // (有错误)，不继续
                }
                break;
            case 2u:
                if (p_tcb->SemCtr == DEF_INT_16U_MAX_VAL)
                {
                    OS_CRITICAL_EXIT();  // 返错误类型为“计
                    *p_err = OS_ERR_SEM_OVF;
                    return ((OS_SEM_CTR)0);  // (有错误)，不继续
                }
                break;
            case 4u:
                if (p_tcb->SemCtr == DEF_INT_32U_MAX_VAL)
                {
                    OS_CRITICAL_EXIT();  // 返错误类型为“计
                    *p_err = OS_ERR_SEM_OVF;
                    return ((OS_SEM_CTR)0);
                }
                break;
        }
}
break;

default:
    break;
}
p_tcb->SemCtr++;
    //信号计数值不溢出则加 1
ctr = p_tcb->SemCtr;
    //获取信号量的当前计数值
OS_CRITICAL_EXIT();
    //退出临界段
break;
    //跳出

case OS_TASK_STATE_PEND:      //如果任务有等待状态
    break;

case OS_TASK_STATE_PEND_TIMEOUT:
    break;

case OS_TASK_STATE_PEND_SUSPENDED:
    break;

case OS_TASK_STATE_PEND_TIMEOUT_SUSPENDED:
    if (p_tcb->PendOn == OS_TASK_PEND_ON_TASK_SEM)      //如果正等待任务信号量
    {
        OSPost((OS_PEND_OBJ*)0,                   //发布信号量给目标任务
            (OS_TCB*)p_tcb,
            (void*)0,
            (OS_MSG_SIZE)0u,
            (CPU_TS)ts);                 //获取信号量的当前计数值
        OS_CRITICAL_EXIT_NO_SCHED();             //退出临界段（无调度）
        if ((opt & OS_OPT_POST_NO_SCHED) == (OS_OPT)0)   //如果选择了调度任务
            {OSSched(); //调度任务
            }
    }
    else //如果没等待任务信号量
        break;

(下页继续)
switch (sizeof(OS_SEM_CTR))  
{
    case 1u:  // 该信号量计数值
        if (p_tcb->SemCtr == DEF_INT_08U_MAX_VAL)  // 如果溢出，
        {
            OS_CRITICAL_EXIT();  // 则开中断，返回
            *p_err = OS_ERR_SEM_OVF;  // 错误类型为“计
            return ((OS_SEM_CTR)0);  // 数值溢出”，返
        }  // 回 0 (有错误)，
        break;  // 不继续执行。
    case 2u:
        if (p_tcb->SemCtr == DEF_INT_16U_MAX_VAL)
        {
            OS_CRITICAL_EXIT();
            *p_err = OS_ERR_SEM_OVF;
            return ((OS_SEM_CTR)0);
        }
        break;
    case 4u:
        if (p_tcb->SemCtr == DEF_INT_32U_MAX_VAL)
        {
            OS_CRITICAL_EXIT();
            *p_err = OS_ERR_SEM_OVF;
            return ((OS_SEM_CTR)0);
        }
        break;
    default:
        break;
} } 
  p_tcb->SemCtr++;  //信号量计数值不溢出则加 1 
  ctr = p_tcb->SemCtr;  //获取信号量的当前计数值 
  OS_CRITICAL_EXIT();  //退出临界段 
} 
  break;  //跳出 

  default:  (14) //如果任务状态超出预期 
  OS_CRITICAL_EXIT();  //退出临界段 
  *p_err = OS_ERR_STATE_INVALID;  //错误类型为“状态非法” 
  ctr = (OS_SEM_CTR)0;  //清零 ctr 
  break;  //跳出 
} 
return (ctr);  //返回信号量的当前计数值
• 代码清单: 任务信号量-2 (10): 如果任务有等待状态，并且如果正等待任务信号量。

• 代码清单: 任务信号量-2 (11): 调用 OS_Post() 函数发布信号量给目标任务，该函数在前面章节有讲解。

• 代码清单: 任务信号量-2 (12): 如果选择了调度任务，就进行一次任务调度。

• 代码清单: 任务信号量-2 (13): 如果不是等待任务信号量，判断一下是否即将导致该信号量计数值溢出，如果溢出，则开中断，返回错误类型为“计数值溢出”的错误代码，退出不再继续执行，如果信号量还没溢出，信号量计数值加 1。

• 代码清单: 任务信号量-2 (14): 如果任务状态超出预期，返回错误类型为“状态非法”的错误代码。

在释放任务信号量的时候，系统首先判断目标任务的状态，只有处于等待状态并且等待的是任务信号量那就调用 OS_Post() 函数让等待的 task 结束（如果内核对象信号量的话，还会让任务脱离等待列表），所以任务信号量的操作是非常高效的；如果没有处于等待状态或者等待的不是任务信号量，那就直接将任务控制块的元素 SemCtr 加 1。最后返回任务信号量计数值。

其实，不管是否启用了中断延迟发布，最终都是调用 OS_TaskSemPost() 函数进行释放任务信号量。只是启用了中断延迟发布的释放过程会比较曲折，中间会有许多插曲，这是中断管理范畴的内容，留到后面再作介绍。在 OS_TaskSemPost() 函数中，又会调用 OS_Post() 函数释放内核对象。OS_Post() 函数是一个底层的释放（发布）函数，它不仅仅用来释放（发布）任务信号量，还可以释放信号量、互斥信号量、消息队列、事件标志组或任务消息队列。注意：在这里，OS_Post() 函数将任务信号量直接释放给目标任务。

释放任务互斥量函数的使用实例具体见 代码清单: 任务信号量-3 。

论坛：https://www.firebbs.cn/ 667 天猫：https://yehuosm.tmall.com
列表 3: 代码清单: 任务信号量-3OSTaskSemPost() 使用实例

```
OSTaskSemPost((OS_TCB *) &AppTaskPendTCB, // 目标任务
(OS_OPT ) OS_OPT_POST_NONE, // 没选项要求
(OS_ERR *) &err); // 返回错误类型
```

### 27.2.2 获取任务信号量函数 OSTaskSemPend()

与 OSTaskSemPost() 任务信号量释放函数相对应，OSTaskSemPend() 函数用于获取一个任务信号量，参数中没有指定某个任务去获取信号量，实际上就是当前运行任务获取它自己拥有的任务信号量，OSTaskSemPend() 源码具体见代码清单: 任务信号量-4。

列表 4: 代码清单: 任务信号量-4OSTaskSemPend() 源码

```
OS_SEM_CTR OSTaskSemPend (OS_TICK timeout, (1) // 等待超时时间
    OS_OPT opt, (2) // 选项
    CPU_TS *p_ts, (3) // 返回时间戳
    OS_ERR *p_err) (4) // 返回错误类型
{
    OS_SEM_CTR ctr;
    CPU_SR_ALLOC(); // 使用到临界段（关/开中断时）时必须用到该宏，该宏声明和
    // 定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
    // SR（临界段关中断只需保存 SR），开中断时将该值还原。

    #ifdef OS_SAFETY_CRITICAL // 如果启用了安全检测
        if (p_err == (OS_ERR *) 0) // 如果错误类型实参为空
        {
            OS_SAFETY_CRITICAL_EXCEPTION(); // 执行安全检测异常函数
            return ((OS_SEM_CTR) 0); // 返回 0 (有错误), 停止执行
        }
    #endif
```

(下页继续)
```c
#define

#if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u //如果启用了中断中非法调用检测
    if (OSIntNestingCtr > (OS_NESTING_CTR)0) //如果该函数在中断中被调用
    {
        *p_err = OS_ERR_PEND_ISR; //返回错误类型为“在中断中等待”
        return ((OS_SEM_CTR)0); //返回 0 (有错误)，停止执行
    }
#endif

#if OS_CFG_ARG_CHK_EN > 0u //如果启用了参数检测
    switch (opt) //根据选项分类处理
    {
        case OS_OPT_PEND_BLOCKING: //如果选项在预期内
            break; //直接跳出
        case OS_OPT_PEND_NON_BLOCKING:
            break; //直接跳出
        default: //如果选项超出预期
            *p_err = OS_ERR_OPT_INVALID; //错误类型为“选项非法”
            return ((OS_SEM_CTR)0); //返回 0 (有错误)，停止执行
    }
#endif

if (p_ts != (CPU_TS *)0) //如果 p_ts 非空
{
    *p_ts = (CPU_TS)0; //清零 (初始化) p_ts
}

CPU_CRITICAL_ENTER(); //关中断
if (OSTCBCurPtr->SemCtr > (OS_SEM_CTR)0) //如果任务信号量当前可用
{
    OSTCBCurPtr->SemCtr--; (5) //信号量计数器减 1
```
ctr = OSTCBCurPtr->SemCtr;  //获取信号量的当前计数值
if (p_ts != (CPU_TS *)) {  //如果 p_ts 非空
    *p_ts = OSTCBCurPtr->TS;  //返回信号量被发布的时戳
}
#endif  // if OS_CFG_TASK_PROFILE_EN > 0u  (8)

OSTCBCurPtr->SemPendTime = OS_TS_GET() - OSTCBCurPtr->TS;  //更新任务等待

if (OSTCBCurPtr->SemPendTimeMax < OSTCBCurPtr->SemPendTime)  //任务信号量的
    OSTCBCurPtr->SemPendTimeMax = OSTCBCurPtr->SemPendTime;  //最长时间记录。
#endif  // if OS_CFG_TASK_PROFILE_EN > 0u  (8)

CPU_CRITICAL_EXIT();  //开中断
*p_err = OS_ERR_NONE;  //错误类型为“无错误”
return (ctr);  //返回信号量的当前计数值

/* 如果任务信号量当前不可用 */  (9)
if ((opt & OS_OPT_PEND_NON_BLOCKING) != (OS_OPT)0) //如果选择了不阻塞任务
{
    CPU_CRITICAL_EXIT();  //开中断
    *p_err = OS_ERR_PEND_WOULD_BLOCK;  //错误类型为“缺乏阻塞”
    return ((OS_SEM_CTR)0);  //返回 0 (有错误), 停止执行
}

else (10) //如果选择了阻塞任务
{
    if (OSSchedLockNestingCtr > (OS_NESTING_CTR)0)  //如果调度器被锁
    {
        CPU_CRITICAL_EXIT();  //开中断
        *p_err = OS_ERR_SCHED_LOCKED;  //错误类型为“调度器被锁”
    }
}
```c
    return ((OS_SEM_CTR)0);               //返回 0 (有错误), 停止执行
}

/* 如果调度器未被锁 */
OS_CRITICAL_ENTER_CPU_EXIT();        //锁调度器, 重开中断
OS_Pend((OS_PEND_DATA *)0,          //阻塞任务, 等待信号量。
    (OS_PEND_OBJ *)0,                  //不需插入等待列表。
    (OS_STATE )OS_TASK_PEND_ON_TASK_SEM,
    (OS_TICK )timeout);                //开调度器 (无调度)
OS_CRITICAL_EXIT_NO_SCHED();

OSSched();                            //调度任务
/* 任务获得信号量后得以继续运行 */
CPU_CRITICAL_ENTER();                //关中断
switch (OSTCBCurPtr->PendStatus)     //根据任务的等待状态分类处理
{
    case OS_STATUS_PEND_OK:           //如果任务成功获得信号量
        if (p_ts != (CPU_TS *)0)       //返回信号量被发布的时间戳
            {
                *p_ts = OSTCBCurPtr->TS;

#if OS_CFG_TASK_PROFILE_EN > 0u //更新最长等待时间记录
                OSTCBCurPtr->SemPendTime = OS_TS_GET() - OSTCBCurPtr->TS;
                if (OSTCBCurPtr->SemPendTimeMax < OSTCBCurPtr->SemPendTime)
                    {
                        OSTCBCurPtr->SemPendTimeMax = OSTCBCurPtr->SemPendTime;
                    }
#endif
            }
            *p_err = OS_ERR_NONE;         //错误类型为“无错误”
            break;                      //跳出
    case OS_STATUS_PEND_ABORT:        //如果等待被中止
```

if (p_ts != (CPU_TS *)) {  // 返回被终止时的时间戳
    *p_ts = OSTCBCurPtr->TS;
}  
*p_err = OS_ERR_PEND_ABORT;  // 错误类型为“等待被中止”
break;  // 跳出

case OS_STATUS_PEND_TIMEOUT:  // (16) 如果等待超时
    if (p_ts != (CPU_TS *)) {  // 返回时间戳为 0
        *p_ts = (CPU_TS) 0;
    }
    *p_err = OS_ERR_TIMEOUT;  // 错误类型为“等待超时”
break;  // 跳出

default:  // (17) 如果等待状态超时超出预期
    *p_err = OS_ERR_STATUS_INVALID;  // 错误类型为“状态非法”
break;  // 跳出

} ctr = OSTCBCurPtr->SemCtr;  // 获取信号量的当前计数值
CPU_CRITICAL_EXIT();  // 开中断
return (ctr);  // (18) 返回信号量的当前计数值

• 代码清单: 任务信号量-4 (1): 等待超时时间。
• 代码清单: 任务信号量-4 (2): 等待的选项。
• 代码清单: 任务信号量-4 (3): 保存返回的时间戳。
• 代码清单: 任务信号量-4 (4): 保存返回错误的类型。
• 代码清单: 任务信号量-4 (5): 如果任务信号量当前可用，那就信号量计数值 SemCtr 减一。
• 代码清单: 任务信号量-4 (6): 获取信号量的当前计数值保存在 ctr 变量中，用于返回。
- 代码清单: 任务信号量-4 (7): 返回信号量被发布的时间戳。
- 代码清单: 任务信号量-4 (8): 如果启用任务统计的宏，计算任务信号量从被释放到获取所用时间及最大时间。
- 代码清单: 任务信号量-4 (9): 如果任务信号量当前不可用，并且如果用户选择了不阻塞任务，那么就返回错误类型为“缺乏阻塞”错误代码。
- 代码清单: 任务信号量-4 (10): 如果选择了阻塞任务，判断一下调度器是否被锁，如果被锁，则返回错误类型为“调度器被锁”的错误代码。
- 代码清单: 任务信号量-4 (11): 如果调度器未被锁，锁调度器，重开中断，调用 OS_Pend()函数将当前任务进入阻塞状态以等待任务信号量，该函数在前面的章节已经讲解过，此处就不再重复赘述。
- 代码清单: 任务信号量-4 (12): 进行一次任务调度。
- 代码清单: 任务信号量-4 (13): 当程序能执行到这里，就说明大体上有两种情况，要么是任务获取到任务信号量了；要么任务还没获取到任务信号量（任务没获取到任务信号量的情况有很多种），无论是哪种情况，都先把中断关掉再说，再根据当前运行任务的等待状态分类处理。
- 代码清单: 任务信号量-4 (14): 如果任务成功获得任务信号量，返回信号量被发布的时间簇，然后跳出 switch 语句。
- 代码清单: 任务信号量-4 (15): 如果任务在等待中被中止，返回被终止时的时间簇，返回错误类型为“等待被中止”的错误代码，跳出 switch 语句。
- 代码清单: 任务信号量-4 (16): 如果任务等待超时，返回错误类型为“等待超时”的错误代码，跳出 switch 语句。
- 代码清单: 任务信号量-4 (17): 如果等待状态超出预期，返回错误类型为“状态非法”的错误代码。
- 代码清单: 任务信号量-4 (18): 获取并返回任务信号量的当前计数值。

在调用该函数的时候，系统先判断任务信号量是否可用，即检查任务信号量的计数值是否大于 0，如果大于 0，即表示可用，这个时候获取信号量，即记计数值减 1 后直接返回。如果信号量不可用，且当调度器没有被锁住时，用户希望在任务信号量不可用的时候进行阻塞任务以等待任务信
号量可用，那么系统就会调用 OS_Pend() 函数将任务脱离就绪列表，如果用户有指定超时时间，系统还要将该任务插入等待列表。注意：此处系统并没有将任务插入等待列表。然后切换任务，处于就绪列表中最高优先级的任务通过任务调度获得 CPU 使用权，等到出现任务信号量被释放、任务等待任务信号量被强制停止、等待超时等情况，任务会从阻塞中恢复，等待任务信号量的任务重新获得 CPU 使用权，返回相关错误代码和任务信号量计数值，用户可以根据返回的错误知道任务退出等待状态的情况。

获取任务信号量函数的使用实例具体见 代码清单: 任务信号量-5

列表 5: 代码清单: 任务信号量-5 OSTaskSemPend()

```
OSTaskSemPend ((OS_TICK ) 0, //无期限等待
    (OS_OPT ) OS_OPT_PEND_BLOCKING, //如果信号量不可用就等待
    (CPU_TS ) * &ts, //获取信号量被发布的时间戳
    (OS_ERR ) * &err); //返回错误类型
```

27.3 任务信号量实验

27.3.1 任务信号量代替二值信号量

任务通知代替消息队列是在 MC/OS 中创建了两个任务，其中一个任务是用于接收任务信号量，另一个任务发送任务信号量。两个任务独立运行，发送任务信号量的任务是通过检测按键的按下情况发送，等待任务在任务信号量中没有可用的信号量之前就一直等待，获取到信号量以后后续执行。这样子是为了代替二值信号量，任务同步成功则继续执行，然后在串口调试助手里将运行信息打印出来，具体见 代码清单: 任务信号量-6 加粗部分。

列表 6: 代码清单: 任务信号量-6 任务通知代替二值信号量

```
#include <includes.h>

static OS_TCB AppTaskStartTCB; //任务控制块
```

论坛：https://www.firebbs.cn/ 674 天猫：https://yehuosm.tmall.com
static OS_TCB AppTaskPostTCB;
static OS_TCB AppTaskPendTCB;

static CPU_STK AppTaskStartStk[APP_TASK_START_STK_SIZE];  // 任务栈

static CPU_STK AppTaskPostStk [ APP_TASK_POST_STK_SIZE ];
static CPU_STK AppTaskPendStk [ APP_TASK_PEND_STK_SIZE ];

static void AppTaskStart (void *p_arg);  // 任务函数声明
static void AppTaskPost ( void *p_arg );
static void AppTaskPend ( void *p_arg );

int main (void)
{
  OS_ERR err;

  OSInit(&err);  // 初始化 μC/OS-III

  /* 创建起始任务 */
  OSTaskCreate((OS_TCB *)&AppTaskStartTCB,
  // 任务控制块地址
  (CPU_CHAR *)"App Task Start",
  // 任务名称
  (OS_TASK_PTR ) AppTaskStart,
  // 任务函数
  (void *) 0,
  // 传递给任务函数（形参 p_arg）的实参
  (OS_PRIO ) APP_TASK_START_PRIO,
  // 任务的优先级
(下页继续)
(CPU_STK *) AppTaskStartStk[0],
  //任务栈的基地址
(CPU_STK_SIZE) APP_TASK_START_STK_SIZE / 10,
  //任务栈空间剩下 1/10 时限制其增长
(CPU_STK_SIZE) APP_TASK_START_STK_SIZE,
  //任务栈空间（单位：sizeof(CPU_STK))
(OS_MSG_QTY ) 5u,
  //任务可接收的最大消息数
(OS_TICK ) 0u,
  //任务的时间片节拍数（0 表默认值 OSCfg_TickRate_Hz/10)
(void *) 0,
  //任务扩展（0 表不扩展）
(OS_OPT ) (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
  //任务选项
(OS_ERR *) &err);
  //返回错误类型

OSStart(&err);
  //启动多任务管理（交由 µC/OS-III 控制)
}

static void AppTaskStart (void *p_arg)
{
  CPU_INT32U cpu_clk_freq;
  CPU_INT32U cnts;
  OS_ERR err;

  (void)p_arg;

  //板级初始化
  BSP_Init();
//初始化CPU组件（时间戳、中断时间测量和主机名）
CPU_Init();

//获取CPU内核时钟频率(SysTick工作时钟)
cpu_clk_freq = BSP_CPU_ClkFreq();
//根据用户设定的时钟节拍频率计算SysTick定时器的计数值
cnts = cpu_clk_freq / (CPU_INT32U)OSCfg_TickRate_Hz;
//调用SysTick初始化函数，设置定时器计数值和启动定时器
OS_CPU_SysTickInit(cnts);

Mem_Init();
//初始化内存管理组件（堆内存池和内存池表）

#if OS_CFG_STAT_TASK_EN > 0u
//如果启用（默认启用）了统计任务
OSStatTaskCPUUsageInit(&err);
#endif

CPU_IntDisMeasMaxCurReset();
//复位（清零）当前最大关中断时间

/*创建AppTaskPost任务*/
OSTaskCreate((OS_TCB *) &AppTaskPostTCB,
    //任务控制块地址
    (CPU_CHAR * )"App Task Post",
    //任务名称
    (OS_TASK_PTR ) AppTaskPost,
    //任务函数
    (void *) 0,
    //传递给任务函数（形参p_arg）的实参
    (下页继续)
(OS_PRIO ) APP_TASK_POST_PRIO,
  // 任务的优先级
  (CPU_STK *) &AppTaskPostStk[0],
  // 任务栈的地址
  (CPU_STK_SIZE) APP_TASK_POST_STK_SIZE / 10,
  // 任务栈空间剩下 1/10 限时其增长
  (CPU_STK_SIZE) APP_TASK_POST_STK_SIZE,
  // 任务栈空间（单位：sizeof(CPU_STK))
  (OS_MSG_QTY ) 5u,
  // 任务可接收的最大消息数
  (OS_TICK ) 0u,
  // 任务的时间片节拍数（0 表默认值 OSCfg_TickRate_Hz/10）
  (void *) 0,
  // 任务扩展（0 表不扩展）
  (OS_OPT ) (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
  // 任务选项
  (OS_ERR *) &err);
  // 返回错误类型

  /* 创建 AppTaskPend 任务 */
OSTaskCreate((OS_TCB *) &AppTaskPendTCB,
  // 任务控制块地址
  (CPU_CHAR *) "App Task Pend",
  // 任务名称
  (OS_TASK_PTR ) AppTaskPend,
  // 任务函数
  (void *) 0,
  // 传递给任务函数的参数 p_arg 的实参
  (OS_PRIO ) APP_TASK_PEND_PRIO,
  // 任务的优先级
  (CPU_STK *) &AppTaskPendStk[0],
  // 任务栈的地址

(下页继续)
(CPU_STK_SIZE) APP_TASK_PEND_STK_SIZE / 10,
//任务栈空间下 1/10 时限制其增长
(CPU_STK_SIZE) APP_TASK_PEND_STK_SIZE,
//任务栈空间（单位：sizeof(CPU_STK)）
(OS_MSG_QTY ) 5u,
//任务可接收的最大消息数
(OS_TICK ) 0u,
//任务的时间片节拍数 (0 表默认值 OSCfg_TickRate_Hz/10)
(void *) 0,
//任务扩展 (0 表不扩展)
(OS_OPT ) (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
//任务选项
(OS_ERR *) &err);
//返回错误类型
OSTaskDel ( & AppTaskStartTCB, & err );
//删除起始任务本身，该任务不再运行

static void AppTaskPost ( void * p_arg )
{
    OS_ERR       err;

    uint8_t ucKey1Press = 0;    //记忆按键 KEY1 状态

    (void)p_arg;

    while ( DEF_TRUE )
        //任务体
if ( Key_Scan ( macKEY1_GPIO_PORT, macKEY1_GPIO_PIN, 1, &ucKey1Press ) ) {
    //如果 KEY1 被按下
    printf("发送任务信号量\n");
    /* 发送任务信号量 */
    OSTaskSemPost((OS_TCB *)&AppTaskPendTCB,
                   //目标任务
                   (OS_OPT )OS_OPT_POST_NONE,
                   //没选项要求
                   (OS_ERR *)&err);  //返回错误类型
}

OSTimeDlyHMSM ( 0, 0, 0, 20, OS_OPT_TIME_DLY, & err );  //每 20ms 扫描一次

static void AppTaskPend ( void * p_arg )
{
    OS_ERR err;
    CPU_TS ts;
    CPU_INT32U cpu_clk_freq;
    CPU_SR_ALLOC();

    (void)p_arg;
cpu clk_freq = BSP_CPU_ClkFreq();
//获取 CPU 时钟，时间戳是以该时钟计数

while (DEF_TRUE) //任务体
{
    /* 阻塞任务，直到 KEY1 被按下 */
    OSTaskSemPend ((OS_TICK )0, //无期限等待
        (OS_OPT )OS_OPT_PEND_BLOCKING,
        //如果信号量不可用就等待
        (CPU_TS *)&ts,
        //获取信号量被发布的时间戳
        (OS_ERR *)&err); //返回错误类型

    ts = OS_TS_GET() - ts;
    //计算信号量从发布到接收的时间差

    macLED1_TOGGLE (); //切换 LED1 的亮灭状态

    OS_CRITICAL_ENTER();
    //进入临界段，避免串口打印被打断

    printf ( "任务信号量从被发送到被接收的时间差是%dus\n\n", ts / ( cpu_clk_freq / 1000000 ) );

    OS_CRITICAL_EXIT(); //退出临界段
}

27.3.2 任务信号量代替计数信号量

任务通知代替计数信号量是基于计数信号量实验修改而来，模拟停车场工作运行。并且在μC/OS中创建了两个任务：一个是获取信号量任务，一个是发送信号量任务，两个任务独立运行，获取任务信号量的任务是通过按下 KEY1 按键获取，模拟停车场停车操作，其等待时间是 0；发送任务信号量的任务则是通过检测 KEY2 按键按下进行信号量的发送（发送到获取任务），模拟停车场取车操作，并且在串口调试助手输出相应信息，实验源码具体见 代码清单: 任务信号量-7。

列表 7: 代码清单: 任务信号量-7 任务通知代替计数信号量

```c
#include <includes.h>

static OS_TCB AppTaskStartTCB; //任务控制块
static OS_TCB AppTaskPostTCB;
static OS_TCB AppTaskPendTCB;
static CPU_STK AppTaskStartStk[APP_TASK_START_STK_SIZE]; //任务栈
static CPU_STK AppTaskPostStk[APP_TASK_POST_STK_SIZE];
static CPU_STK AppTaskPendStk[APP_TASK_PEND_STK_SIZE];
static void AppTaskStart ( void *p_arg); //任务函数声明
static void AppTaskPost ( void *p_arg);
static void AppTaskPend ( void *p_arg);

int main (void)
{
    OS_ERR err;
    OSInit(&err);
    //初始化μC/OS-III

    /* 创建起始任务 */
    OSTaskCreate((OS_TCB *) &AppTaskStartTCB,
                 //任务控制块地址
                 (CPU_CHAR *) "App Task Start",
```

// 任务名称
(OS_TASK_PTR ) AppTaskStart,
// 任务函数
(void * ) 0,
// 传递给任务函数 (形参 p_arg) 的实参
(OS_PRIO ) APP_TASK_START_PRIO,
// 任务的优先级
(CPU_STK * ) AppTaskStartStk[0],
// 任务栈的基地址
(CPU_STK_SIZE) APP_TASK_START_STK_SIZE / 10,
// 任务栈空间剩下 1/10 时限制其增长
(CPU_STK_SIZE) APP_TASK_START_STK_SIZE,
// 任务栈空间 (单位: sizeof (CPU_STK))
(OS_MSG_QTY ) 5u,
// 任务可接收的最大消息数
(OS_TICK ) 0u,
// 任务的时间片节拍数 (0 表默认值 OSCfg_TickRate_Hz/10)
(void * ) 0,
// 任务扩展 (0 表不扩展)
(OS_OPT ) (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
// 任务选项
(OS_ERR * ) &err);
// 返回错误类型
OSStart(&err);
// 启动多任务管理 (交由 μC/OS-III 控制)

static void AppTaskStart (void * p_arg)
{
    CPU_INT32U cpu_clk_freq;
}
CPU_INT32U cnts;
OS_ERR err;

(void)p_arg;

BSP_Init();
//板级初始化
CPU_Init();
//初始化 CPU 组件（时钟数、关中断时间测量和主机名）

cpu_clk_freq = BSP_CPU_ClkFreq();
//获取 CPU 内核时钟频率（SysTick 工作时钟）
cnts = cpu_clk_freq / (CPU_INT32U)OSCfg_TickRate_Hz;
//根据用户设定的时钟节拍频率计算 SysTick 定时器的计数值
//调用 SysTick 初始化函数，设置定时器计数值和启动定时器
OS_CPU_SysTickInit(cnts);

Mem_Init();
//初始化内存管理组件（堆内存池和内核表）

#if OS_CFG_STAT_TASK_EN > 0u
//如果启用（默认启用）了统计任务
OSStatTaskCPUUsageInit(&err);
#endif
//复位（清零）当前最大关中断时间
CPU_IntDisMeasMaxCurReset();

/* 创建 AppTaskPost 任务 */
OSTaskCreate((OS_TCB *)&AppTaskPostTCB,
    //任务控制块地址
    (CPU_CHAR *)"App Task Post",
    //任务名称
    (CPU_CHAR *)"App Task Post",
    //任务名称
87 (OS_TASK_PTR) AppTaskPost,
88 //任务函数
89 (void *) 0,
90 //传递给任务函数（形参 p_arg）的实参
91 (OS_PRIO) APP_TASK_POST_PRIO,
92 //任务的优先级
93 (CPU_STK *) &AppTaskPostStk[0],
94 //任务栈的基地址
95 (CPU_STK_SIZE) APP_TASK_POST_STK_SIZE / 10,
96 //任务栈空间剩下 1/10 时限制其增长
97 (CPU_STK_SIZE) APP_TASK_POST_STK_SIZE,
98 //任务栈空间（单位: sizeof(CPU_STK)）
99 (OS_MSG_QTY) 5u,
100 //任务可接收的最大消息数
101 (OS_TICK) 0u,
102 //任务的时间片节拍数 (0 表默认值 OSCfg_TickRate_Hz/10)
103 (void *) 0,
104 //任务扩展 (0 表不扩展)
105 (OS_OPT) (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
106 //任务选项
107 (OS_ERR *) &err);
108 //返回错误类型

/* 创建 AppTaskPend 任务 */
OSTaskCreate((OS_TCB *) &AppTaskPendTCB,
            //任务控制块地址
            (CPU_CHAR *) "App Task Pend",
            //任务名称
            (OS_TASK_PTR) AppTaskPend,
            //任务函数
            (void *) 0,
            //传递给任务函数（形参 p_arg）的实参

(下页继续)
(OS_PRIO) APP_TASK_PEND_PRIO,
//任务的优先级
(CPU_STK *) &AppTaskPendStk[0],
//任务栈的基地址
(CPU_STK_SIZE) APP_TASK_PEND_STK_SIZE / 10,
//任务栈空间剩下 1/10 时限制其增长
(CPU_STK_SIZE) APP_TASK_PEND_STK_SIZE,
//任务栈空间（单位: sizeof(CPU_STK)）
(OS_MSG_QTY) 5u,
//任务可接收的最大消息数
(OS_TICK) 0u,
//任务的时间片节拍数（0 表默认值 OSCfg_TickRate_Hz/10）
(void *) 0,
//任务扩展（0 表不扩展）
(OS_OPT) (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
//任务选项
(OS_ERR *) &err);
//返回错误类型
OSTaskDel (& AppTaskStartTCB, & err);
//删除起始任务本身，该任务不再运行

static void AppTaskPost (void * p_arg)
{
    OS_ERR err;
    OS_SEM_CTR ctr;
    uint8_t ucKey2Press = 0;  //记忆按键 KEY2 状态
    CPU_SR_ALLOC();
void p_arg;

while (DEF_TRUE)
    // 任务体
    {
        if ( Key_Scan ( macKEY2_GPIO_PORT, macKEY2_GPIO_PIN, 1, &ucKey2Press ) )
            // 如果 KEY2 被按下
            {
                /* 发布任务信号 */
                ctr = OSTaskSemPost((OS_TCB *)&AppTaskPendTCB,
                    // 目标任务
                    (OS_OPT )OS_OPT_POST_NONE,
                    // 没选项要求
                    (OS_ERR *)&err);
                // 返回错误类型

                macLED2_TOGGLE();
                OS_CRITICAL_ENTER();
                // 进入临界段，避免串口打印被打断

                printf( "KEY2 被按下，释放 1 个停车位，当前车位为 %d 个\n", ctr);

                OS_CRITICAL_EXIT();        // 退出临界段

            }

        OSTimeDlyHMSM ( 0, 0, 0, 20, OS_OPT_TIME_DLY, & err );
        // 每 20ms 扫描一次
static void AppTaskPend ( void * p_arg )
{
    OS_ERR err;
    CPU_SR_ALLOC();
    OS_SEM_CTR ctr; // 当前任务信号量计数

    uint8_t ucKey1Press = 0; // 记忆按键 KEY1 状态
    (void)p_arg;

    while (DEF_TRUE) // 任务体
    {
        if ( Key_Scan ( macKEY1_GPIO_PORT, macKEY1_GPIO_PIN, 1, &ucKey1Press ) )
        { // 如果 KEY2 被按下
            ctr = OSTaskSemPend ((OS_TICK )0,
                (OS_OPT )OS_OPT_PEND_NON_BLOCKING,
                (CPU_TS *)0, // 获取信号量被发布的时问戳
                (OS_ERR *)&err); // 返回错误类型
            macLED1_TOGGLE ();
            // 切换 LED1 的亮灭状态
        }
    }
}
OS_CRITICAL_ENTER();
//进入临界段，避免串口打印被打断

if (OS_ERR_NONE == err)
    printf("KEY1 被按下，申请车位成功，当前剩余车位为 %d 个\n", ctr);
else
    printf("申请车位失败，请按 KEY2 释放车位\n");

OS_CRITICAL_EXIT();  //退出临界段

OSTimeDlyHMSM ( 0, 0, 0, 20, OS_OPT_TIME_DLY, & err );
}

27.4 任务信号量实验现象

27.4.1 任务信号量代替二值信号量

将程序编译好，用 USB 线连接计算机和开发板的 USB 接口（对应丝印为 USB 转串口），用 DAP 仿真器把配套程序下载到野火 STM32 开发板（具体型号根据购买的板子而定，每个型号的板子都配套有对应的程序），在计算机上打开串口调试助手，然后复位开发板就可以在调试助手中看到串口的打印信息，它里面输出了信息表明任务正在运行中，我们按下开发板的按键，串口打印任务运行的信息，表明两个任务同步成功，具体见图任务信号量代替二值信号量实验现象。
27.4.2 任务信号量代替计数信号量

将程序编译好，用 USB 线连接计算机和开发板的 USB 接口（对应丝印为 USB 转串口），用 DAP 仿真器把配套程序下载到野火 STM32 开发板（具体型号根据购买的板子而定，每个型号的板子都配套有对应的程序），在计算机上打开串口调试助手，然后复位开发板就可以在调试助手中看到串口的打印信息，按下开发板的 KEY1 按键获取信号量模拟停车，按下 KEY2 按键释放信号量模拟取车，因为是使用任务信号量代替信号量，所以任务通信信号量默认为 0，表当前车位为 0；我们按下 KEY1 与 KEY2 试试，在串口调试助手中可以看到运行信息，具体见图任务信号量代替计数信号量实验现象。

论坛：https://www.firebbs.cn/ 690 天猫：https://yehuosm.tmall.com
第 28 章 任务消息队列

28.1 任务消息队列的基本概念

任务消息队列跟任务信号量一样，均隶属于某一个特定任务，不需单独创建，任务在则任务消息队列在，只有该任务才可以获取（接收）这个任务消息队列的消息，其它任务只能给这个任务消息队列发送消息，却不能获取。任务消息队列与前面讲解的（普通）消息队列极其相似，只是任务消息队列已隶属于一个特定任务，所以它不具有等待列表，在操作的过程中省去了等待任务插入和移除列表的动作，所以工作原理相对更简单一点，效率也比较高一些。

注意：本书所提的“消息队列”，若无特别说明，均指前面的（普通）消息队列（属于内核对象），而非任务消息队列。

通过对任务消息队列的合理使用，可以在一定场合下替代 μC/OS 的消息队列，用户只需向任务内部的消息队列发送一个消息而不用通过外部的消息队列进行发送，这样子处理就会很方便并且更加高效，当然，凡事都有利弊，任务消息队列虽然处理更快，RAM 开销更小，但也有限制：只能指定消息发送的对象，有且只有一个任务接收消息；而内核对象的消息队列则没有这个限制，用户在发送消息的时候，可以采用广播消息的方式，让所有等待该消息的任务都获取到消息。

在实际任务间的通信中，一个或多个任务发送一个消息给另一个任务是非常常见的，而一个任务给多个任务发送消息的情况相对比较少，前者就很适合采用任务消息队列进行传递消息，如果任务消息队列可以满足设计需求，那么尽量不要使用普通消息队列，这样子设计的系统会更加高效。

（内核对象）消息队列是用结构体 OS_Q 来管理的，包含了管理消息的元素 MsgQ 和管理等待列表的元素 PendList 等。而任务消息队列的结构体成员变量就少了 PendList，因为等待任务消息队列只有拥有任务消息队列本身的任务才可以进行获取，故任务消息队列不需要等待列表的相关数据结构，具体见代码清单：任务消息队列-1。

注意：想要使用任务消息队列，就必须将 OS_CFG_TASK_Q_EN 宏定义配置为 1，该宏定义位于 os_cfg.h 文件中。
列表1：代码清单：任务消息队列-1任务消息队列数据结构

```c
struct os_msg_q
{
    OS.MSG *InPtr; (1)
    OS.MSG *OutPtr; (2)
    OS.MSG_QTY NbrEntriesSize; (3)
    OS.MSG_QTY NbrEntries; (4)
    OS.MSG_QTY NbrEntriesMax; (5)
};
```

- **代码清单：任务消息队列-1** (1)、(2)：任务消息队列中进出消息指针。
- **代码清单：任务消息队列-1** (3)：任务消息队列中最大可用的消息个数，在创建任务的时候由用户指定这个值的大小。
- **代码清单：任务消息队列-1** (4)：记录任务消息队列中当前的消息个数，每当发送一个消息到任务消息队列的时候，若任务没有在等待该消息，那么新发送的消息被插入任务消息队列后此值加1，NbrEntries 的大小不能超过 NbrEntriesSize。
- **代码清单：任务消息队列-1** (5)：记录任务消息队列最多的时候拥有的消息个数。

任务消息队列的运作机制与普通消息队列一样，没什么差别。

### 28.2 任务消息队列的函数接口讲解

#### 28.2.1 任务消息队列发送函数 OSTaskQPost()

函数OSTaskQPost() 用来发送任务消息队列，参数中有指向消息要发送给的任务控制块的指针，任何任务都可以发送消息给拥有任务消息队列的任务（任务在被创建的时候，要设置参数 q_size 大于 0），其源码具体见**代码清单：任务消息队列-2**。

论坛：https://www.firebbs.cn/  693  天猫：https://yehuosm.tmall.com
列表 2: 代码清单: 任务消息队列-2 OSTaskQPost() 源码

```c
#include OS_CFG_TASK_Q_EN > 0u //如果启用了任务消息队列

void OSTaskQPost (OS_TCB *p_tcb, (1) //目标任务
    void *p_void, (2) //消息内容地址
    OS_MSG_SIZE msg_size, (3) //消息长度
    OS_OPT opt, (4) //选项
    OS_ERR *p_err) (5) //返回错误类型
{
    CPU_TS ts;

    #ifdef OS_SAFETY_CRITICAL //如果启用了安全检测
    if (p_err == (OS_ERR *)0) //如果错误类型实例为空
    {
        OS_SAFETY_CRITICAL_EXCEPTION(); //执行安全检测异常函数
        return; //返回，停止执行
    }
    #endif

    #if OS_CFG_ARG_CHK_EN > 0u //如果启用了参数检测
    switch (opt) //根据选项分类处理
    {
    case OS_OPT_POST_FIFO: //如果选项在预期内
        case OS_OPT_POST_LIFO:
        case OS_OPT_POST_FIFO | OS_OPT_POST_NO_SCHED:
        case OS_OPT_POST_LIFO | OS_OPT_POST_NO_SCHED:
            break; //直接跳出

    default: //如果选项超出预期
        *p_err = OS_ERR_OPT_INVALID; //错误类型为“选项非法”
        return; //返回，停止执行
    }

    (下页继续)
```
```c
#endif

  ts = OS_TS_GET();    //获取时间戳

#if OS_CFG_ISR_POST_DEFERRED_EN > 0u //如果启用了中断延迟发布
  if (OSIntNestingCtr > (OS_NESTING_CTR)0)  //如果该函数在中断中被调用
  {
    OS_IntQPost((OS_OBJ_TYPE)OS_OBJ_TYPE_TASK_MSG, //将消息先发布到中断消息队列
        (void  *)p_tcb,
        (void  *)p_void,
        (OS_MSG_SIZE)msg_size,
        (OS_FLAGS  )0,
        (OS_OPT   )opt,
        (CPU_TS   )ts,
        (OS_ERR   *)p_err);         //返回
  }
#endif

OS_TaskQPost(p_tcb, //将消息直接发布
  p_void,
  msg_size,
  opt,
  ts,
  p_err);         //返回
  //返回

#endif
```

- 代码清单：任务消息队列-2 (1)：目标任务。
- 代码清单：任务消息队列-2 (2)：任务消息内容指针。

论坛：https://www.firebbs.cn/  
天猫：https://yehuosm.tmall.com
野火®

[野火]uCOS-III 内核实现与应用开发实战指南

- 代码清单: 任务消息队列-2 (3): 任务消息的大小。
- 代码清单: 任务消息队列-2 (4): 发送的选项。
- 代码清单: 任务消息队列-2 (5): 用于保存返回的错误类型。
- 代码清单: 任务消息队列-2 (6): 如果启用了中断延迟发布，并且如果该函数在中断中被调用，就先将消息先发布到中断消息队列。
- 代码清单: 任务消息队列-2 (7): 调用 OS_TaskQPost() 函数将消息直接发送，其源码具体见 代码清单: 任务消息队列-3。

列表 3: 代码清单: 任务消息队列-3 OS_TaskQPost() 源码

```c
#if OS_CFG_TASK_Q_EN > 0u//如果启用了任务消息队列
void OS_TaskQPost (OS_TCB *p_tcb, //目标任务
                   void *p_void, //消息内容地址
                   OS_MSG_SIZE msg_size, //消息长度
                   OS_OPT opt, //选项
                   CPU_TS ts, //时间戳
                   OS_ERR *p_err) //返回错误类型
{
    CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必须用到该宏，该宏声明和
                         //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
                         // SR (临界段关中断只保存 SR)，开中断时将该值还原。
    OS_CRITICAL_ENTER(); //进入临界段
    if (p_tcb == (OS_TCB *)0) (1)//如果 p_tcb 为空
    {
        p_tcb = OSTCBCurPtr; //目标任务为自身
    }
    *p_err = OS_ERR_NONE; //错误类型为“无错误”
    switch (p_tcb->TaskState) (2)//根据任务状态分类处理
    {
        case OS_TASK_STATE_RDY: //如果目标任务没等待状态
                        
    (下页继续)
```

论坛: https://www.firebbs.cn/ 696 天猫: https://yehuosm.tmall.com
case OS_TASK_STATE_DLY:
case OS_TASK_STATE_SUSPENDED:
case OS_TASK_STATE_DLY_SUSPENDED:
    OS_MsgQPut(&p_tcb->MsgQ, // 把消息放入任务消息队列
               p_void,
               msg_size,
               opt,
               ts,
               p_err);          (3)
    OS_CRITICAL_EXIT(); // 退出临界段
    break;              // 跳出

case OS_TASK_STATE_PEND: // 如果目标任务有等待状态

case OS_TASK_STATE_PEND_TIMEOUT:
case OS_TASK_STATE_PEND_SUSPENDED:
case OS_TASK_STATE_PEND_TIMEOUT_SUSPENDED:
    if (p_tcb->PendOn == OS_TASK_PEND_ON_TASK_Q) // 如果等的是任务消息队列
        {
            OS_Post((OS_PEND_OBJ *)0, // 把消息发布给目标任务
                     p_tcb,
                     p_void,
                     msg_size,
                     ts);         (4)
            OS_CRITICAL_EXIT_NO_SCHED(); // 退出临界段（无调度）
            if ((opt & OS_OPT_POST_NO_SCHED) == (OS_OPT)0u) // 如果要调度任务
                {
                    OSSched(); // 调度任务
                }
        }
    else(5) // 如果没在等待任务消息队列
        {
        }
OS_MsgQPut(&p_tcb->MsgQ, p_void, msg_size, opt, ts, p_err);
OS_CRITICAL_EXIT();

break;  //跳出

default:
OS_CRITICAL_EXIT();  //退出临界段
*p_err = OS_ERR_STATE_INVALID;  //错误类型为“状态非法”
break;  //跳出
}
#endif

- 代码清单: 任务消息队列-3 (1): 如果目标任务为空，则表示将任务消息释放给自己，那么p_tcb 就指向当前任务。

- 代码清单: 任务消息队列-3 (2): 根据任务状态分类处理。

- 代码清单: 任务消息队列-3 (3): 如果目标任务没等待状态，就调用 OS_MsgQPut() 函数将消息放入队列中，执行完毕就退出。

- 代码清单: 任务消息队列-3 (4): 如果目标任务有等待状态，那就看看是不是在等待任务消息队列，如果是的话，调用 OS_Post() 函数把任务消息发送给目标任务。

- 代码清单: 任务消息队列-3 (5): 如果任务并不是在等待任务消息队列，那么调用 OS_MsgQPut() 函数将消息放入任务消息队列中即可。

- 代码清单: 任务消息队列-3 (6): 如果状态超出预期，返回错误类型为“状态非法”的错误代码。
任务消息队列的发送过程是跟消息队列发送过程差不多，先检查目标任务的状态，如果该任务刚刚好在等待任务消息队列的消息，那么直接让任务脱离等待状态即可。如果任务没有在等待任务消息队列的消息，那么就将消息插入要发送消息的任务消息队列。

任务消息队列发送函数的使用实例具体见代码清单: 任务消息队列-4。

列表4: 代码清单: 任务消息队列-4 OSTaskQPost() 使用实例

```c
OS_ERR err;

/* 发布消息到任务 AppTaskPend */
OSTaskQPost ((OS_TCB *)AppTaskPendTCB, //目标任务的控制块
    (void *)"YeHuo μC/OS-III",
    (OS_MSG_SIZE )sizeof ( "YeHuo μC/OS-III" ), //消息长度
    (OS_OPT )OS_OPT_POST_FIFO,
    //发布到任务消息队列的入口端
    (OS_ERR *)&err); //返回错误类型
```

28.2.2 任务消息队列获取函数 OSTaskQPend()

与OSTaskQPost()任务消息队列发送函数相对应，OSTaskQPend()函数用于获取一个任务消息队列，函数的参数中没有指定哪个任务获取任务消息，实际上就是当前执行的任务，当任务调用了这个函数就表明这个任务需要获取任务消息，OSTaskQPend()源码具体见代码清单: 任务消息队列-5。

列表5: 代码清单: 任务消息队列-5OSTaskQPend() 源码

```c
#if OS_CFG_TASK_Q_EN > 0u//如果启用了任务消息队列
void *OSTaskQPend (OS_TICK timeout, (1) //等待期限（单位：时钟节拍）
    OS_OPT opt, (2) //选项
    OS_MSG_SIZE *p_msg_size, (3) //返回消息长度
    CPU_TS *p_ts, (4) //返回时间戳
```

(下页继续)
```c
OS_ERR *p_err) (5) //返回错误类型
{
    OS_MSG_Q *p_msg_q;
    void *p_void;
    CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必须用到该宏，该宏声明和
    //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
    // SR (临界段关中断只保存 SR)，开中断时将该值还原。
#endif OS_SAFETY_CRITICAL //如果启用（默认禁用）了安全检测
    if (p_err == (OS_ERR *)0) //如果错误类型实参为空
    {
        OS_SAFETY_CRITICAL_EXCEPTION(); //执行安全检测异常函数
        return ((void *)0); //返回 0 (有错误)，停止执行
    }
#endif
#endif OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u //如果启用了中断中非法调用检测
    if (OSIntNestingCtr > (OS_NESTING_CTR)0) //如果该函数在中断中被调用
    {
        *p_err = OS_ERR_PEND_ISR; //错误类型为“在中断中中止等待”
        return ((void *)0); //返回 0 (有错误)，停止执行
    }
#endif
#if OS_CFG_ARG_CHK_EN > 0u //如果启用了参数检测
    if (p_msg_size == (OS_MSG_SIZE *)0) //如果 p_msg_size 为空
    {
        *p_err = OS_ERR_PTR_INVALID; //错误类型为“指针不可用”
        return ((void *)0); //返回 0 (有错误)，停止执行
    }
    switch (opt) //根据选项分类处理
    {
```

论坛：https://www.firebbs.cn/
天猫：https://yehuosm.tmall.com
case OS_OPT_PEND_BLOCKING:  // 如果选项在预期内
    break;  // 直接跳出

default:  // 如果选项超出预期内
    *p_err = OS_ERR_OPT_INVALID;  // 错误类型为“选项非法”
    return ((void *)0);  // 返回 0 (有错误)，停止执行
#endif

if (p_ts != (CPU_TS *)0)  // 如果 p_ts 非空
{
    *p_ts = (CPU_TS *)0;  // 初始化 (清零) p_ts，待用于返回时间戳
}

CPU_CRITICAL_ENTER();  // 关中断
p_msg_q = &OSTCBCurPtr->MsgQ;  // 获取当前任务的消息队列
p_void = OS_MsgQGet(p_msg_q,  // 从队列中获取一个消息
    p_msg_size,
    p_ts,
    p_err);  // (7)
if (*p_err == OS_ERR_NONE)  // 如果获取消息成功
{
#if OS_CFG_TASK_PROFILE_EN > 0u

    if (p_ts != (CPU_TS *)0)
    {
        OSTCBCurPtr->MsgQPendTime = OS_TS_GET() - *p_ts;
        if (OSTCBCurPtr->MsgQPendTimeMax < OSTCBCurPtr->MsgQPendTime)
        {
            OSTCBCurPtr->MsgQPendTimeMax = OSTCBCurPtr->MsgQPendTime;
        }
    }
}
endif

CPU_CRITICAL_EXIT(); // 开中断
return (p_void); // 返回消息内容

/* 如果获取消息不成功（队列里没有消息） */ (8)
if ((opt & OS_OPT_PEND_NON_BLOCKING) != (OS_OPT) 0) // 如果选择了不阻塞任务
{
    *p_err = OS_ERR_PEND_WOULD_BLOCK; // 错误类型为“缺乏阻塞”
    CPU_CRITICAL_EXIT(); // 开中断
    return ((void *) 0); // 返回 0 (有错误)，停止执行
}
else(9) // 如果选择了阻塞任务
{
    if (OSSchedLockNestingCpr > (OS_NESTING_CTR) 0) // 如果调度器被锁
    {
        CPU_CRITICAL_EXIT(); // 开中断
        *p_err = OS_ERR_SCHED_LOCKED; // 错误类型为“调度器被锁”
        return ((void *) 0); // 返回 0 (有错误)，停止执行
    }
    /* 如果调度器未被锁 */ (10) // 锁调度器，重开中断
OS_CRITICAL_ENTER_CPU_EXIT();
OS_Pend((OS_PEND_DATA *) 0,
        (OS_PEND_OBJ *) 0,
        (OS_STATE ) OS_TASK_PEND_ON_TASK_Q,
        (OS_TICK ) timeout);
OS_CRITICAL_EXIT_NO_SCHED(); // 解锁调度器（无调度）

OSSched(); (12) // 调度任务
/* 当前任务（获得消息队列的消息）得以继续运行 */
CPU_CRITICAL_ENTER(); (13) // 关中断

(下页继续)
switch (OSTCBCurPtr->PendStatus) //根据任务的等待状态分类处理
{
  case OS_STATUS_PEND_OK:  (14)//如果任务已成功获得消息
    p_void = OSTCBCurPtr->MsgPtr; //提取消息内容地址
    *p_msg_size = OSTCBCurPtr->MsgSize; //提取消息长度
    if (p_ts != (CPU_TS *)0) //如果 p_ts 非空
    {
      *p_ts = OSTCBCurPtr->TS; //获取任务等到消息时的时间戳
    }
    #if OS_CFG_TASK_PROFILE_EN > 0u
      OSTCBCurPtr->MsgQPendTime = OS_TS_GET() - OSTCBCurPtr->TS;
      if (OSTCBCurPtr->MsgQPendTimeMax < OSTCBCurPtr->MsgQPendTime)
      {
        OSTCBCurPtr->MsgQPendTimeMax = OSTCBCurPtr->MsgQPendTime;
      }
    #endif
  }
  *p_err = OS_ERR_NONE; //错误类型为“无错误”
  break; //跳过

  case OS_STATUS_PEND_ABORT:  (15)//如果等待被中止
    p_void = (void *)0; //返回消息内容为空
    *p_msg_size = (OS_MSG_SIZE)0; //返回消息大小为 0
    if (p_ts != (CPU_TS *)0) //如果 p_ts 非空
    {
      *p_ts = (CPU_TS )0; //清零 p_ts
    }
    *p_err = OS_ERR_PEND_ABORT; //错误类型为“等待被中止”
    break; //跳过

  case OS_STATUS_PEND_TIMEOUT: (16)//如果等待超时，
    default: //或者任务状态超出预

p_void = (void *)0; //返回消息内容为空  
*p_msg_size = (OS_MSG_SIZE)0; //返回消息大小为 0  
if (p_ts != (CPU_TS *)0) //如果 p_ts 非空  
{  
    *p_ts = OSTCBCurPtr->TS;  
}  
*p_err = OS_ERR_TIMEOUT; //错误类为 “等待超时”  
break; //跳出  
CPU_CRITICAL_EXIT(); //开中断  
return (p_void); (17) //返回消息内容地址  
#endif

- 代码清单: 任务消息队列-5 (2): 获取任务消息队列的选项。  
- 代码清单: 任务消息队列-5 (3): 返回消息大小。  
- 代码清单: 任务消息队列-5 (4): 返回时间截。  
- 代码清单: 任务消息队列-5 (5): 返回错误类型。  
- 代码清单: 任务消息队列-5 (6): 获取当前任务的消息队列保存在 p_msg_q 变量中。  
- 代码清单: 任务消息队列-5 (7): 调用 OS_MsgQGet() 函数从消息队列获取一个消息，如果获取消息成功，则返回指向消息的指针。  
- 代码清单: 任务消息队列-5 (8): 如果获取消息不成功（任务消息队列里没有消息），并且如果用户选择了不阻塞任务，那么返回错误类型为 “缺乏阻塞” 的错误代码，然后退出。  
- 代码清单: 任务消息队列-5 (9): 如果选择了阻塞任务，先判断一下调度器是否被锁，如果被锁了也就不能继续执行。  
- 代码清单: 任务消息队列-5 (10): 如果调度器未被锁，系统会锁调度器，重开中断。
代码清单: 任务消息队列-5 (11): 调用 OS_Pend() 函数将当前任务脱离就绪列表，并根据用户指定的阻塞时间插入就绪列表，但是不会插入队列等待列表，然后打开调度器，但不进行调度，OS_Pend() 源码具体见代码清单 18-18。

代码清单: 任务消息队列-5 (12): 进行一次任务调度。

代码清单: 任务消息队列-5 (13): 程序能执行到这里，就说明大体上有两种情况，要么是任务获取到消息了；任务还没获取到消息（任务没获取到消息的情况有很多种），无论是哪种情况，都先把中断关掉再说，然后根据当前运行任务的等待状态分类处理。

代码清单: 任务消息队列-5 (14): 如果任务状态是 OS_STATUS_PEND_OK，则表示任务获取到消息了，那么就从任务控制块中提取消息，这是因为在发送消息给任务的时候，会将消息放入任务控制块的 MsgPtr 成员变量中，然后继续提取消息大小，如果 p_ts 非空，记录获取任务待到消息时的时间戳，返回错误类型为“无错误”的错误代码，跳出 switch 语句。

代码清单: 任务消息队列-5 (15): 如果任务在等待（阻塞）重被中止，则返回消息内容为空，返回消息大小为 0，返回错误类型为“等待被中止”的错误代码，跳出 switch 语句。

代码清单: 任务消息队列-5 (16): 如果任务等待（阻塞）超时，说明等待的时间过去了，任务也没获取到消息，则返回消息内容为空，返回消息大小为 0，返回错误类型为“等待超时”的错误代码，跳出 switch 语句。

代码清单: 任务消息队列-5 (17): 打开中断，返回消息内容。

### 28.3 任务消息队列实验

任务通知代替消息队列是在 MCOS 中创建了两个任务，其中一个任务是用于接收任务消息，另一个任务发送任务消息。两个任务独立运行，发送消息任务每秒发送一次任务消息，接收任务在就一直在等待消息，一旦获取到消息通知就把消息打印在串口调试助手里，具体见代码清单: 任务消息队列-6。
列表 6: 代码清单: 任务消息队列-6 任务通知代替消息队列

```c
#include <includes.h>

static OS_TCB AppTaskStartTCB; // 任务控制块
static OS_TCB AppTaskPostTCB;
static OS_TCB AppTaskPendTCB;
static CPU_STK AppTaskStartStk[APP_TASK_START_STK_SIZE]; // 任务栈
static CPU_STK AppTaskPostStk [ APP_TASK_POST_STK_SIZE ];
static CPU_STK AppTaskPendStk [ APP_TASK_PEND_STK_SIZE ];
static void AppTaskStart ( void *p_arg); // 任务函数声明
static void AppTaskPost ( void * p_arg );
static void AppTaskPend ( void * p_arg );

int main (void)
{
  OS_ERR err;
  OSInit(&err); // 初始化 μC/OS

  /* 创建起始任务 */
  OSTaskCreate((OS_TCB *) &AppTaskStartTCB,
    // 任务控制块地址
    (CPU_CHAR *)"App Task Start", // 任务名称
    (OS_TASK_PTR ) AppTaskStart, // 任务函数
    (void *) 0,
    // 传送给任务函数（形参 p_arg）的实参
    (OS_PRIO ) APP_TASK_START_PRIO, // 任务的优先级
    (CPU_STK *) &AppTaskStartStk[0],
    // 任务栈的基地址
    (CPU_STK_SIZE) APP_TASK_START_STK_SIZE / 10,
    // 任务栈空间剩下 1/10 时限制其增长
  )
}
```

(下页继续)
(CPU_STK_SIZE) APP_TASK_START_STK_SIZE,
    // 任务栈空间（单位：sizeof(CPU_STK))
(OS_MSG_QTY) 5u,
    // 任务可接的最多消息数
(OS_TICK) 0u,
    // 任务的时间片数（0 表默认值 OSCfg_TickRate_Hz/10)
(void *) 0,
    // 任务扩展（0 表不扩展）
(OS_OPT) (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR), //
    // 任务选项
(OS_ERR *) &err;                                     // 返回错误类型

OSStart(&err);
    // 启动多任务管理（交由 μC/OS-III 控制）
}

static void AppTaskStart (void *p_arg)
{
    CPU_INT32U cpu_clk_freq;
    CPU_INT32U cnts;
    OS_ERR err;

    (void)p_arg;

    BSP_Init();                                            // 板级初始化
    CPU_Init();                                            // 初始化 CPU 组件（时钟、关中断时间测量和主机名）

    cpu_clk_freq = BSP_CPU_ClkFreq();
    // 获取 CPU 内核时钟频率 (SysTick 工作时钟)
    cnts = cpu_clk_freq / (CPU_INT32U)OSCfg_TickRate_Hz;}
// 根据用户设定的时钟节拍频率计算 SysTick 定时器的计数值
OS_CPU_SysTickInit(cnts);    // 调用 SysTick 初始化函数，设置定时器计数值和启动定时器

Mem_Init();                  // 初始化内存管理组件（堆内存池和内存池表）

#if OS_CFG_STAT_TASK_EN > 0u
    // 如果启用（默认启用）了统计任务
    OSStatTaskCPUUsageInit(&err);
#endif

CPU_IntDisMeasMaxCurReset();  // 复位（清零）当前最大关中断时间

/* 创建 AppTaskPost 任务 */
OSTaskCreate((OS_TCB   *)&AppTaskPostTCB,
              // 任务控制块地址
              (CPU_CHAR   *)"App Task Post",     // 任务名称
              (OS_TASK_PTR ) AppTaskPost,        // 任务函数
              (void    *) 0,
              // 传递给任务函数（形参 p_arg）的实参
              (OS_PRIO    ) APP_TASK_POST_PRIO,   // 任务的优先级
              (CPU_STK    *)&AppTaskPostStk[0],  // 任务栈的基地址
              (CPU_STK_SIZE) APP_TASK_POST_STK_SIZE / 10,
              // 任务栈空间剩下 1/10 时限制其增长
              (CPU_STK_SIZE) APP_TASK_POST_STK_SIZE,
              // 任务栈空间（单位：sizeof(CPU_STK)）
/* 创建 AppTaskPend 任务 */
OSTaskCreate((OS_TCB *) &AppTaskPendTCB,
   // 任务控制块地址
   (CPU_CHAR *) "App Task Pend",       // 任务名称
   (OS_TASK_PTR) AppTaskPend,          // 任务函数
   (void *) 0,                         // 传递给任务函数（形参 p_arg）的实参
   (OS_PRIO) APP_TASK_PEND_PRIO,       // 任务的优先级
   (CPU_STK *) &AppTaskPendStk[0],    // 任务栈的基地址
   (CPU_STK_SIZE) APP_TASK_PEND_STK_SIZE / 10, // 任务栈空间剩下 1/10 时限制其增长
   (CPU_STK_SIZE) APP_TASK_PEND_STK_SIZE, // 任务栈空间（单位: sizeof(CPU_STK)）
   (OS_MSG_QTY) 50u,                   // 任务可接收的最大消息数
   (OS_TICK) 0u,                       // 任务的时间片节拍数 (0 表默认值 Oscfg_TickRate_Hz/10)
   (void *) 0,                         // 任务扩展 (0 表不扩展)
   (OS_OPT) (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR), // 任务选项
   (OS_ERR *) &err);                   // 返回错误类型

(93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123)

论坛: https://www.firebbs.cn/  709  天猫: https://yehuosm.tmall.com
OSTaskDel (& AppTaskStartTCB, & err);
// 删除起始任务本身，该任务不再运行

static void AppTaskPost ( void * p_arg )
{
    OS_ERR err;
    (void)p_arg;

    while (DEF_TRUE) // 任务体
    {
        /* 发送消息到任务 AppTaskPend */
        OSTaskQPost ((OS_TCB *)&AppTaskPendTCB, // 目标任务的控制块
                      (void *)"Fire μC/OS-III", // 消息内容
                      (OS_MSG_SIZE )sizeof( "Fire μC/OS-III" ), // 消息长度
                      (OS_OPT )OS_OPT_POST_FIFO,
                      // 发送到任务消息队列的入口端
                      (OS_ERR *)&err); // 返回错误类型

        OSTimeDlyHMSM ( 0, 0, 1, 0, OS_OPT_TIME_DLY, & err );
    }
}

static void AppTaskPend ( void * p_arg )
{
    OS_ERR err;

OS_MSG_SIZE msg_size;
CPU_TS ts;
CPU_INT32U cpu_clk_freq;
CPU_SR_ALLOC();

cpu_clk_freq = BSP_CPU_ClkFreq();
// 获取 CPU 时钟，时间戳是以该时钟计数

while (DEF_TRUE) { // 任务体
    /* 阻塞任务，等待任务消息 */
    pMsg = OSTaskQPend ((OS_TICK)0, // 无期限等待
    (OS_OPT)OS_OPT_PEND_BLOCKING, // 没有消息就阻塞任务
    (OS_MSG_SIZE *)&msg_size, // 返回消息长度
    (CPU_TS *)&ts,
    // 返回消息被发送的时间戳
    (OS_ERR *)&err); // 返回错误类型

    ts = OS_TS_GET() - ts;
    // 计算消息从发送到被接收的时间差

    macLED1_TOGGLE(); // 切换 LED1 的亮灭状态

    OS_CRITICAL_ENTER();
    // 进入临界段，避免串口打印被打断
}
28.4 任务消息队列实验现象

将程序编译好，用 USB 线连接计算机和开发板的 USB 接口（对应丝印为 USB 转串口），用 DAP 仿真器把配套程序下载到野火 STM32 开发板（具体型号根据购买的板子而定，每个型号的板子都配套有对应的程序），在计算机上打开串口调试助手，然后复位开发板就可以在调试助手中看到串口的运行打印信息，具体见图任务通知代替消息队列实验现象。
<table>
<thead>
<tr>
<th>端口</th>
<th>COM3</th>
</tr>
</thead>
<tbody>
<tr>
<td>波特率</td>
<td>115200</td>
</tr>
<tr>
<td>校验位</td>
<td>None</td>
</tr>
<tr>
<td>数据位</td>
<td>8</td>
</tr>
<tr>
<td>停止位</td>
<td>1</td>
</tr>
</tbody>
</table>

接收到的消息内容为：Fire uC/OS-III，长度是：15字节。
任务消息从被发送到被接收的时间差是43us

接收到的消息内容为：Fire uC/OS-III，长度是：15字节。
任务消息从被发送到被接收的时间差是43us

接收到的消息内容为：Fire uC/OS-III，长度是：15字节。
任务消息从被发送到被接收的时间差是43us

接收到的消息内容为：Fire uC/OS-III，长度是：15字节。
任务消息从被发送到被接收的时间差是43us

接收到的消息内容为：Fire uC/OS-III，长度是：15字节。
任务消息从被发送到被接收的时间差是43us

接收到的消息内容为：Fire uC/OS-III，长度是：15字节。
任务消息从被发送到被接收的时间差是43us

接收到的消息内容为：Fire uC/OS-III，长度是：15字节。
任务消息从被发送到被接收的时间差是43us

接收到的消息内容为：Fire uC/OS-III，长度是：15字节。
任务消息从被发送到被接收的时间差是43us

接收到的消息内容为：Fire uC/OS-III，长度是：15字节。
任务消息从被发送到被接收的时间差是43us

接收到的消息内容为：Fire uC/OS-III，长度是：15字节。
任务消息从被发送到被接收的时间差是43us

接收到的消息内容为：Fire uC/OS-III，长度是：15字节。
任务消息从被发送到被接收的时间差是43us

接收到的消息内容为：Fire uC/OS-III，长度是：15字节。
任务消息从被发送到被接收的时间差是43us

接收到的消息内容为：Fire uC/OS-III，长度是：15字节。
任务消息从被发送到被接收的时间差是43us

接收到的消息内容为：Fire uC/OS-III，长度是：15字节。
任务消息从被发送到被接收的时间差是43us

接收到的消息内容为：Fire uC/OS-III，长度是：15字节。
任务消息从被发送到被接收的时间差是43us

接收到的消息内容为：Fire uC/OS-III，长度是：15字节。
任务消息从被发送到被接收的时间差是43us

接收到的消息内容为：Fire uC/OS-III，长度是：15字节。
任务消息从被发送到被接收的时间差是43us

接收到的消息内容为：Fire uC/OS-III，长度是：15字节。
任务消息从被发送到被接收的时间差是43us

接收到的消息内容为：Fire uC/OS-III，长度是：15字节。
任务消息从被发送到被接收的时间差是43us

接收到的消息内容为：Fire uC/OS-III，长度是：15字节。
任务消息从被发送到被接收的时间差是43us

接收到的消息内容为：Fire uC/OS-III，长度是：15字节。
任务消息从被发送到被接收的时间差是43us

接收到的消息内容为：Fire uC/OS-III，长度是：15字节。
任务消息从被发送到被接收的时间差是43us

接收到的消息内容为：Fire uC/OS-III，长度是：15字节。
任务消息从被发送到被接收的时间差是43us

接收到的消息内容为：Fire uC/OS-III，长度是：15字节。
任务消息从被发送到被接收的时间差是43us

接收到的消息内容为：Fire uC/OS-III，长度是：15字节。
任务消息从被发送到被接收的时间差是43us

接收到的消息内容为：Fire uC/OS-III，长度是：15字节。
任务消息从被发送到被接收的时间差是43us

接收到的消息内容为：Fire uC/OS-III，长度是：15字节。
任务消息从被发送到被接收的时间差是43us

接收到的消息内容为：Fire uC/OS-III，长度是：15字节。
任务消息从被发送到被接收的时间差是43us

接收到的消息内容为：Fire uC/OS-III，长度是：15字节。
任务消息从被发送到被接收的时间差是43us

接收到的消息内容为：Fire uC/OS-III，长度是：15字节。
任务消息从被发送到被接收的时间差是43us

接收到的消息内容为：Fire uC/OS-III，长度是：15字节。
任务消息从被发送到被接收的时间差是43us

接收到的消息内容为：Fire uC/OS-III，长度是：15字节。
任务消息从被发送到被接收的时间差是43us

接收到的消息内容为：Fire uC/OS-III，长度是：15字节。
任务消息从被发送到被接收的时间差是43us

接收到的消息内容为：Fire uC/OS-III，长度是：15字节。
任务消息从被发送到被接收的时间差是43us

接收到的消息内容为：Fire uC/OS-III，长度是：15字节。
任务消息从被发送到被接收的时间差是43us

接收到的消息内容为：Fire uC/OS-III，长度是：15字节。
任务消息从被发送到被接收的时间差是43us

接收到的消息内容为：Fire uC/OS-III，长度是：15字节。
任务消息从被发送到被接收的时间差是43us

接收到的消息内容为：Fire uC/OS-III，长度是：15字节。
任务消息从被发送到被接收的时间差是43us

接收到的消息内容为：Fire uC/OS-III，长度是：15字节。
任务消息从被发送到被接收的时间差是43us

接收到的消息内容为：Fire uC/OS-III，长度是：15字节。
任务消息从被发送到被接收的时间差是43us

接收到的消息内容为：Fire uC/OS-III，长度是：15字节。
任务消息从被发送到被接收的时间差是43us

接收到的消息内容为：Fire uC/OS-III，长度是：15字节。
任务消息从被发送到被接收的时间差是43us

接收到的消息内容为：Fire uC/OS-III，长度是：15字节。
任务消息从被发送到被接收的时间差是43us

接收到的消息内容为：Fire uC/OS-III，长度是：15字节。
任务消息从被发送到被接收的时间差是43us

接收到的消息内容为：Fire uC/OS-III，长度是：15字节。
任务消息从被发送到被接收的时间差是43us

接收到的消息内容为：Fire uC/OS-III，长度是：15字节。
任务消息从被发送到被接收的时间差是43us

接收到的消息内容为：Fire uC/OS-III，长度是：15字节。
任务消息从被发送到被接收的时间差是43us

接收到的消息内容为：Fire uC/OS-III，长度是：15字节。
任务消息从被发送到被接收的时间差是43us

接收到的消息内容为：Fire uC/OS-III，长度是：15字节。
任务消息从被发送到被接收的时间差是43us

接收到的消息内容为：Fire uC/OS-III，长度是：15字节。
任务消息从被发送到被接收的时间差是43us

接收到的消息内容为：Fire uC/OS-III，长度是：15字节。
任务消息从被发送到被接收的时间差是43us

接收到的消息内容为：Fire uC/OS-III，长度是：15字节。
任务消息从被发送到被接收的时间差是43us

接收到的消息内容为：Fire uC/OS-III，长度是：15字节。
任务消息从被发送到被接收的时间差是43us

接收到的消息内容为：Fire uC/OS-III，长度是：15字节。
任务消息从被发送到被接收的时间差是43us

接收到的字节数： 3994
发送字节数：
第 29 章 内存管理

29.1 内存管理的基本概念

在计算系统中，变量、中间数据一般存放在系统存储空间中，只有在实际使用时才将它们从存储空间调入到中央处理器内部进行运算。通常存储空间可以分为两种：内部存储空间和外部存储空间。内部存储空间访问速度比较快，能够按照变量地址随机地访问，也就是我们通常所说的 RAM（随机存储器），或计算机的内存；而外部存储空间内所保存的内容相对来说比较固定，即使掉电后数据也不会丢失，可以把它理解为计算机的硬盘。在这一章中我们主要讨论内部存储空间（RAM）的管理——内存管理。

在嵌入式系统设计中，内存分配应该是根据所设计系统的特点来决定选择使用静态内存分配还是动态内存分配算法，一些可靠性要求非常高的系统应选择使用静态的，而普通的业务系统可以使用动态来提高内存使用效率。静态可以保证设备的可靠性但是需要考虑内存上限，内存使用效率低，而动态则是相反。

μC/OS 的内存管理是采用内存池的方式进行管理，也就是创建一个内存池，静态划分一大块连续空间作为内存管理的空间，里面划分为很多个内存块，我们在使用的时候就从这个内存池中获取一个内存块，使用完毕的时候用户可以将其放回内存池中，这样子就不会导致内存碎片的产生。

μC/OS 内存管理模块管理用于系统中内存资源，它是操作系统的核心模块之一，主要包括内存池的创建、分配以及释放。

很多人会有疑问，为什么不直接使用 C 标准库中的内存管理函数呢？在计算机中我们可以用 malloc() 和 free() 这两个函数动态的分配内存和释放内存。但是，在嵌入式实时操作系统中，调用 malloc() 和 free() 却是危险的，原因有以下几点：

- 这些函数在小型嵌入式系统中并不总是可用的，小型嵌入式设备中的 RAM 不足。
- 它们的实现可能非常的大，占据了相当大的一块代码空间。
- 他们几乎都不是安全的。
它们并不是确定的，每次调用这些函数执行的时间可能都不一样。
它们有可能产生碎片。
这两个函数会使得链接器配置得复杂。
如果允许堆空间的生长方向覆盖其他变量占据的内存，它们会成为 debug 的灾难。

在一般的实时嵌入式系统中，由于实时性的要求，很少使用虚拟内存机制。所有的内存都需要用户参与分配，直接操作物理内存，所分配的内存不能超过系统的物理内存，所有的系统栈的管理，都由用户自己管理。

同时，在嵌入式实时操作系统中，对内存的分配时间要求更为苛刻，分配内存的时间必须是确定的。一般内存管理算法是根据需要存储的数据的长度在内存中去寻找一个与这段数据相适应的空闲内存块，然后将数据存储在里面，而寻找这样一个空闲内存块所耗费的时间是不确定的，因此对于实时系统来说，这就是不可接受的，实时系统必须要保证内存块的分配过程在可预测的确定时间内完成，否则实时任务对外部事件的响应也将变得不可确定。

在嵌入式系统中，内存是十分有限而且是十分珍贵的，用一块内存就少了一块内存，而在分配中随着内存不断被分配和释放，整个系统内存区域会产生越来越多的碎片，因为在使用过程中，申请了一些内存，其中一些释放了，导致内存空间中存在一些小的内存块，它们地址不连续，不能够作为整块的大内存分配出去，所以一定会在某个时间，系统已经无法分配到合适的内存了，导致系统瘫痪。其实系统中实际是还有内存的，但是因为小块的内存地址不连续，导致无法分配成功，所以我们需要一个优良的内存分配算法来避免这种情况的出现。所以 μC/OS 提供的内存分配算法是只允许用户分配固定大小的内存块，当使用完成就将其放回内存池中，这样子分配效率极高，时间复杂度是 O(1)，也就是一个固定的时间常数，并不会因为系统内存的多少而增加遍历内存块列表的时间，并且还不会导致内存碎片的出现，但是这样的内存分配机制会导致内存利用率的下降以及申请内存大小的限制。

### 29.2 内存管理的运行机制
内存池（Memory Pool）是一种用于分配大量大小相同的内存对象的技术，它可以极大加快内存分配/释放的速度。

在系统编译的时候，编译器就静态划分了一个大数组作为系统的内存池，然后在初始化的时候将
其分成大小相等的多个内存块，内存块直接通过链表连接起来（此链表也称为空闲内存块列表）。每次分配的时候，从空闲内存块列表中取出表头上第一个内存块，提供给申请者。物理内存中允许存在多个大小不同的内存池，每一个内存池又由多个大小相同的空闲内存块组成。我们必须先创建内存池才能去使用内存池里面的内存块，在创建的时候，我们必须定义一个内存池控制块，然后进行相关初始化，内存控制块的参数包括内存池名称，内存池起始地址，内存块大小，内存块数量等信息，在以后需要从内存池取出内存块或者释放内存块的时候，我们只需根据内存控制块的信息就能很容易做到，内存控制块的数据结构具体见代码清单: 内存管理-1。内存池一旦创建完成，其内部的内存块大小将不能再做调整，具体见图静态内存示意图。

列表1: 代码清单: 内存管理-1 内存控制块数据结构

```c
struct os_mem
{
    OS_OBJ_TYPE Type; (1)
    void *AddrPtr; (2)
    CPU_CHAR *NamePtr; (3)
    void *FreeListPtr; (4)
    OS_MEM_SIZE BlkSize; (5)
    OS_MEM_QTY NbrMax; (6)
    OS_MEM_QTY NbrFree; (7)

#if OS_CFG_DBG_EN > 0u
    OS_MEM *DbgPrevPtr;
    OS_MEM *DbgNextPtr;
#endif
};
```

- 代码清单: 内存管理-1 (1): 内核对象类型。
- 代码清单: 内存管理-1 (2): 内存池的起始地址。
- 代码清单: 内存管理-1 (3): 内存池名称。
- 代码清单: 内存管理-1 (4): 空闲内存块列表。
- 代码清单: 内存管理-1 (5): 内存块大小。

论坛: https://www.firebbs.cn/  716  天猫: https://yehuosm.tmall.com
• 代码清单: 内存管理-I (6): 内存池中内存块的总数量。
• 代码清单: 内存管理-I (7): 空闲内存块数量。

注意: 内存池中的内存块是通过单链表连接起来的，类似于消息池。内存池在创建的时候内存块地址是连续的，但是经过多次申请以及释放后，空闲内存块列表的内存块在地址上不一定是连续的。

29.3 内存管理的应用场景

首先，在使用内存分配前，必须明白自己在做什么，这样做与其他的方法有什么不同，特别是会产生哪些负面影响，在自己的产品面前，应当选择哪种分配策略。

内存管理的主要工作是动态划分并管理用户分配好的内存区间，主要是在用户需要使用大小不等的内存块的场景中使用。当用户需要分配内存时，可以通过操作系统的内存申请函数索取指定大小内存块，一旦使用完毕，通过动态内存释放函数归还所占用内存，使之可以重复使用（heap_1.c 的内存管理除外）。

例如我们需要定义一个 float 型数组：floatArr[];
但是，在使用数组的时候，总有一个问题困扰着我们：数组应该有多大？在很多的情况下，你并不能确定要使用多大的数组，可能为了避免发生错误你就需要把数组定义得足够大。即使你知道想利用的空间大小，但是如果因为某种特殊原因空间利用的大小有增加或者减少，你又必须重新去修改程序，扩大数组的存储范围。这种分配固定大小的内存分配方法称之为静态内存分配。这种内存分配的方法存在比较严重的缺陷，在大多数情况下会浪费大量的内存空间，在少数情况下，当你定义的数组不够大时，可能引起下标越界错误，甚至导致严重后果。

μC/OS 将系统静态分配的大数组作为内存池，然后进行内存池的初始化，然后分配固定大小的内存块。

注意：μC/OS 也不能很好解决这种问题，因为内存块的大小是固定的，无法解决这种弹性很大的内存需求，只能按照最大的内存块进行分配。但是 μC/OS 的内存分配能解决内存利用率的问题，在不需要使用内存的时候，将内存释放到内存池中，让其他任务能正常使用该内存块。

### 29.4 内存管理函数接口讲解

#### 29.4.1 内存池创建函数

在使用内存池的时候首先要创建一个内存池，需要用户静态分配一个数组空间作为系统的内存池，且用户还需定义一个内存控制块。创建内存池后，任务才可以通过系统的内存申请、释放函数从内存池中申请或释放内存，μC/OS 提供内存池创建函数 OSMemCreate()，内存池创建函数源码具体见 代码清单: 内存管理-2。

#### 列表 2: 代码清单: 内存管理-2 静态内存创建函数

```c
void OSMemCreate (OS_MEM *p_mem, (1) //内存池控制块
    CPU_CHAR *p_name, (2) //命名内存池
    void *p_addr, (3) //内存池首地址
    OS_MEM_QTY n_blks, (4) //内存块数目
    OS_MEM_SIZE blk_size, (5) //内存块大小 (单位: 字节)
    OS_ERR *p_err) (6) //返回错误类型
```

(下页继续)
```c
{ 
#if OS_CFG_ARG_CHK_EN > 0u
    CPU_DATA    align_msk;
#endif

OS_MEM_QTY    i;
OS_MEM_QTY    loops;
CPU_INT08U    *p_blk;

void         **p_link;       //二级指针，存放指针的指针
CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必须用到该宏，该宏声明和
//定义一个局部变量，用于保存关中断前的CPU状态寄存器
//SR（临界段关中断只需保存SR），开中断时将该值还原。

#ifdef OS_SAFETY_CRITICAL//如果启用了安全检测
    if (p_err == (OS_ERR *)0)     //如果错误类型实参为空
    {
        OS_SAFETY_CRITICAL_EXCEPTION(); //执行安全检测异常函数
        return;                        //返回，停止执行
    }
#endif

#ifdef OS_SAFETY_CRITICAL_IEC61508//如果启用了安全关键
    if (OSSafetyCriticalStartFlag == DEF_TRUE)
    {
        *p_err = OS_ERR_ILLEGAL_CREATE_RUN_TIME; //错误类型为“非法创建内核对象”
        return;                                   //返回，停止执行
    }
#endif

#if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u//如果启用了中断中非法调用检测
    if (OSIntNestingCtr > (OS_NESTING_CTR)0)   //如果该函数是在中断中被调用
    {
        *p_err = OS_ERR_MEM_CREATE_ISR;        //错误类型为“在中断中创建对象”
    }
}
```


```
return; //返回，停止执行

#endif

#if OS_CFG_ARG_CHK_EN > 0u //如果启用了参数检测
    if (p_addr == (void *)0) { //如果 p_addr 为空
        *p_err = OS_ERR_MEM_INVALID_P_ADDR; //错误类型为“内存池地址非法”
        return; //返回，停止执行
    }
    if (n_blks < (OS_MEM_QTY)2) { //如果内存池的内存块数目少于 2
        *p_err = OS_ERR_MEM_INVALID_BLKS; //错误类型为“内存块数目非法”
        return; //返回，停止执行
    }
    if (blk_size < sizeof(void *)) { //如果内存块空间小于指定的
        *p_err = OS_ERR_MEM_INVALID_SIZE; //错误类型为“内存空间非法”
        return; //返回，停止执行
    }
    align_msk = sizeof(void *) - 1u; //开始检查内存地址是否对齐
    if (align_msk > 0u) {
        if (((CPU_ADDR)p_addr & align_msk) != 0u) { //如果首地址没对齐
            *p_err = OS_ERR_MEM_INVALID_P_ADDR; //错误类型为“内存池地址非法”
            return; //返回，停止执行
        }
        if ((blk_size & align_msk) != 0u) { //如果内存块地址没对齐
            *p_err = OS_ERR_MEM_INVALID_SIZE; //错误类型为“内存块大小非法”
            return; //返回，停止执行
        }
    }
#endif
```
```c
/* 将空闲内存块串联成一个单向链表 */
void p_link;
CPU_INT08U p_blk;

for (i = 0; i < loops; i++) {
    p_blk += blk_size;
    *p_link = (void *)p_blk;
}

OS_CRITICAL_ENTER();

if OS_CFG_DBG_EN > 0u //如果启用了调试代码和变量
    OS_MemDbgListAdd(p_mem);

OSMemQty++;
```

OS_CRITICAL_EXIT_NO_SCHED();     //退出临界段（无调度）
*p_err = OS_ERR_NONE;              //错误类型为“无错误”
}

- 代码清单: 内存管理-2 (1): 内存控制块指针。
- 代码清单: 内存管理-2 (2): 内存池名字。
- 代码清单: 内存管理-2 (3): 内存池首地址。
- 代码清单: 内存管理-2 (4): 内存块数目。
- 代码清单: 内存管理-2 (5): 内存块大小（单位: 字节）。
- 代码清单: 内存管理-2 (6): 返回的错误类型。
- 代码清单: 内存管理-2 (7): 如果启用了参数检测，编译的时候回包含参数检测相关代码，如果 p_addr 为空，返回错误类型为“内存池地址非法”的错误代码。
- 代码清单: 内存管理-2 (8): 如果内存池的内存块数目少于 2，返回错误类型为“内存块数目非法”错误代码。
- 代码清单: 内存管理-2 (9): 如果内存块空间小于一个指针的大小（在 stm32 上是 4 字节），返回错误类型为“内存空间非法”的错误代码。sizeof(void *) 是求出 CPU 指针的字节大小, STM32 是 32 位单片机，求出的指针所占字节大小是 4，减去 1 后就是 3，3 的二进制数是 11(B)。如果一个地址或者内存块字节大小是 4 字节对齐的，那么用二进制表示地址或内存块大小最低两位都是 0，比如 11100(B)、101010100(B) 这些 4 字节对齐的都最低 2 位都是 0，那么 11(B) 与上一个低两位字节都是 0 的数结果肯定为 0，不为 0 说明不是 4 字节对齐。同理可以检测内存块的大小是否是 4 的倍数。
- 代码清单: 内存管理-2 (10): 开始检查内存地址是否对齐，如果内存池首地址没对齐，返回错误类型为“内存池地址非法”的错误代码。
- 代码清单: 内存管理-2 (11): 如果内存块地址没对齐，返回错误类型为“内存块大小非法”的错误代码。

论坛: https://www.firebbs.cn/  天猫: https://yehuosm.tmall.com
代码清单：内存管理-2 (12)：程序执行到这里，就表示传递进来的参数都是正确的，下面开始初始化内存池以及内存控制块的信息，将内存池首地址转为二级指针保存在 p_link 变量中。

代码清单：内存管理-2 (13)：获取内存池中首个内存块地址。

代码清单：内存管理-2 (14)：将空闲内存块逐个连接成一个单向链表，根据内存块起始地址与内存块大小获取下一个内存块的地址，然后在当前内存块中保存下一个内存块的地址，再将下一个内存块的地址转为二级指针，将这些内存块连接成一个单链表，也就是空闲内存块链表。

一个内存块的操作是先计算是下一个内存块的地址，因为此时数组元素的地址是连续的，所以开始的时候只要在前一个内存块的首地址加上内存块字节大小即可得到下一个内存块的首地址，然后把下一个内存块的首地址放在前一个内存块中，就将他们串起来了，如此循环反复即可串成空闲内存块列表。

代码清单：内存管理-2 (15)：然后将最后一个内存块存储的地址为空，表示到达空闲内存块列表尾部，连接完成的示意图具体见图空闲内存块列表初始化完成。

代码清单：内存管理-2 (16)：设置对象的类型。

代码清单：内存管理-2 (17)：保存内存池的名称。

代码清单：内存管理-2 (18)：保存内存池的首地址。

代码清单：内存管理-2 (19)：初始化空闲内存块列表的首地址，指向下一个可用的内存块。

代码清单：内存管理-2 (20)：保存空闲内存块的数目。

代码清单：内存管理-2 (21)：保存内存块的总数目。

代码清单：内存管理-2 (22)：保存内存块的空间大小。

代码清单：内存管理-2 (23)：创建完成，内存管理对象数目加 1。

论坛：https://www.firebbs.cn/ 723 天猫：https://yehuosm.tmall.com
整个内存池创建完成示意图具体见图内存池创建完成。

内存池创建函数的使用实例具体见代码清单: 内存管理-3。

列表 3: 代码清单: 内存管理-3 OSMemCreate() 使用实例

| 1 | OS_MEM mem; //声明内存管理对象 |
| 2 | uint8_t ucArray [ 3 ] [ 20 ]; //声明内存池大小 |
| 3 | |
| 4 | OS_ERR err; |
| 5 | /* 创建内存管理对象 mem */ |
| 6 | OSMemCreate ( (OS_MEM *) &mem, //指向内存管理对象 |
| 7 | (CPU_CHAR *) "Mem For Test", //命名内存管理对象 |
| 8 | (void *) ucArray, //内存池的首地址 |
| 9 | (OS_MEM_QTY ) 3, //内存池中内存块数目 |
| 10 | (OS_MEM_SIZE ) 20, //内存块的字节数目 |
| 11 | (OS_ERR *) &err); //返回错误类型 |

29.4.2 内存申请函数 OSMemGet()

这个函数用于申请固定大小的内存块，从指定的内存池中分配一个内存块给用户使用，该内存块的大小在内存池初始化的时候就已经决定的。如果内存池中有可用的内存块，则从内存池的空闲内存块列表上取下一个内存块并且返回对应的内存地址；如果内存池中已经没有可用内存块，则返回 0 与对应的错误码 OS_ERR_MEM_NO_FREE_BLKS，其源码具体见代码清单: 内存管理-4。

论坛：https://www.firebbs.cn/ 724 天猫：https://yehuosm.tmall.com
列表 4: 代码清单: 内存管理-4OSMemGet() 源码

```c
void  *OSMemGet (OS_MEM  *p_mem, (1)  //内存管理对象
  OS_ERR  *p_err)  (2)  //返回错误类型
{
  void   *p_blk;
  CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必须用到该宏，该宏声明和
  //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
  //SR（临界段关中断只保存SR），开中断时将该值还原。

  #ifdef OS_SAFETY_CRITICAL//如果启用了安全检测
    if (p_err == (OS_ERR *)0) //如果错误类型实参为空
      {  
        OS_SAFETY_CRITICAL_EXCEPTION(); //执行安全检测异常函数
        return ((void *)0); //返回 0 (有错误), 停止执行
      }
  #endif

  #if OS_CFG_ARG_CHK_EN > 0u//如果启用了参数检测
    if (p_mem == (OS_MEM *)0) //如果 p_mem 为空
      {
        *p_err = OS_ERR_MEM_INVALID_P_MEM; //错误类型为“内存池非法”
        return ((void *)0); //返回 0 (有错误), 停止执行
      }
  #endif

  CPU_CRITICAL_ENTER();  //关中断
  if (p_mem->NbrFree == (OS_MEM_QTY)0) (3) //如果没有空闲的内存块
    {
    CPU_CRITICAL_EXIT();  //开中断
    *p_err = OS_ERR_MEM_NO_FREE_BLKS; //错误类型为“没有空闲内存块”
    return ((void *)0); //返回 0 (有错误), 停止执行
  }
```

(下页继续)
• 代码清单: 内存管理-4 (1): 指定内存池对象。

• 代码清单: 内存管理-4 (2): 保存返回的错误类型。

• 代码清单: 内存管理-4 (3): 判断一下内存池控制块中 NbrFree 的值，如果没有空闲的内存块，就没法申请内存，保存错误类型为“没有空闲内存块”的错误代码，返回 0 表示没申请到内存块。

• 代码清单: 内存管理-4 (4): 如果内存池中还有空闲内存块，就获取它，获取的过程就是从空闲内存块中取出一个内存块，并且返回该内存块的地址。

• 代码清单: 内存管理-4 (5): 调整内存池控制块的空闲内存块指针，指向下一个可用的内存块。

• 代码清单: 内存管理-4 (6): 内存池中空闲内存块数目减 1。

• 代码清单: 内存管理-4 (7): 返回获取到的内存块地址。

假设我们在内存池创建完成后就调用 OSMemGet() 函数申请一个内存块，那么申请完毕后的内存块示意图具体见图申请内存块完成示意图，被申请出去的内存块会脱离空闲内存块列表，并且内存控制块中的 NbrFree 变量会减一。
OSMemGet() 函数的使用实例具体见代码清单: 内存管理-5。

列表 5: 代码清单: 内存管理-5 OSMemGet() 使用实例

```c
OS_MEM    mem;                           // 声明内存管理对象
OS_ERR    err;
/* 向 mem 获取内存块 */
p_mem_blk = OSMemGet ((OS_MEM *) &mem,   // 指向内存管理对象
                      (OS_ERR *) &err);  // 返回错误类型
```

### 29.4.3 内存释放函数

嵌入式系统的内存对我们来说是十分珍贵的，任何内存块使用完后都必须被释放，否则会造成内存泄漏，导致系统发生致命错误。μC/OS 提供了 OSMemPut() 函数进行内存的释放管理，使用该函数接口时，根据指定的内存控制块对象，将内存块插入内存池的空闲内存块列表中，然后增加该内存池的可用内存块数目，其源码具体见代码清单: 内存管理-6。

列表 6: 代码清单: 内存管理-6 OSMemPut() 源码

```c
void    OSMemPut (OS_MEM    *p_mem, (1)   // 内存管理对象
      void    *p_blk, (2)       // 要退回的内存块
      OS_ERR   *p_err) (3)      // 返回错误类型
{
```
CPU_SR_ALLOC(); // 使用到临界段（在关/开中断时）时必须用到该宏，该宏声明和
// 定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
// SR（临界段关中断只需保存 SR），开中断时将该值还原。

#ifndef OS_SAFETY_CRITICAL // 如果启用了安全检测
    if (p_err == (OS_ERR *)0) // 如果错误类型实参为空
    {
        OS_SAFETY_CRITICAL_EXCEPTION(); // 执行安全检测异常函数
        return; // 返回，停止执行
    }
#endif

#if OS_CFG_ARG_CHK_EN > 0u // 如果启用了参数检测
    if (p_mem == (OS_MEM *)0) // 如果 p_mem 为空
    {
        *p_err = OS_ERR_MEM_INVALID_P_MEM; // 错误类型为“内存池非法”
        return; // 返回，停止执行
    }
    if (p_blk == (void *)0) // 如果内存块为空
    {
        *p_err = OS_ERR_MEM_INVALID_P_BLK; // 错误类型为“内存块非法”
        return; // 返回，停止执行
    }
#endif

CPU_CRITICAL_ENTER(); // 关中断
    if (p_mem->NbrFree >= p_mem->NbrMax) (4) // 如果内存池已满
    {
        CPU_CRITICAL_EXIT(); // 开中断
        *p_err = OS_ERR_MEM_FULL; // 错误类型为“内存池已满”
        return; // 返回，停止执行
    }

(续上页)
*(void **)p_blk = p_mem->FreeListPtr; (5) //把内存块插入空闲内存块链表
p_mem->FreeListPtr = p_blk; (6) //内存块退回到链表的最前端
p_mem->NbrFree++; (7) //空闲内存块数目加 1
CPU_CRITICAL_EXIT();  //开中断
*p_err = OS_ERR_NONE;  //错误类型为“无错误”
}

- 代码清单: 内存管理-6 (1): 内存控制块指针，指向要操作的内存池。
- 代码清单: 内存管理-6 (2): 要释放的内存块。
- 代码清单: 内存管理-6 (3): 保存返回的错误类型。
- 代码清单: 内存管理-6 (4): 如果内存池已经满了，那是无法进行释放的，返回错误类型为“内存池已满”的错误代码。
- 代码清单: 内存管理-6 (5): 如果内存池没满，那么释放内存块到内存池中，把内存块插入空闲内存块列表。
- 代码清单: 内存管理-6 (6): 内存块退回到链表的最前端。
- 代码清单: 内存管理-6 (7): 空闲内存块数目加 1。

我们在释放一个内存块的时候，我们会将内存插入内存池中空闲内存块列表的首部，然后增加内存池中空闲内存块的数量，该函数的使用实例具体见代码清单: 内存管理-7 。

列表 7: 代码清单: 内存管理-7 OSMemPut() 使用实例

```c
OS_MEM  mem;  //声明内存管理对象
OS_ERR   err;

/* 释放内存块 */
OSMemPut ((OS_MEM  *)&mem,  //指向内存管理对象
    (void  *)&pMsg,  //内存块的首地址
    (OS_ERR  *)&err);  //返回错误类型
```
至此 µC/OS 常用的内存管理函数就讲解完，需要注意的是：我们想要使用内存管理相关的函数时，需要将 os_cfg.h 中的 OS_CFG_MEM_EN 宏定义配置为 1；OSMemCreate() 只能在任务级被调用，但是 OSMemGet() 和 OSMemPut() 可以在中断中被调用。

### 29.5 内存管理实验

本次的实验例程采用消息队列进行发送与接受消息，只不过存放消息的地方是在内存块中，在获取完消息的时候，就进行释放内存块，反复使用内存块，具体见代码清单：内存管理-8 加粗部分。

#### 列表 8：代码清单：内存管理-8 内存管理实验

```c
#include <includes.h>
#include <string.h>

OS_MEM mem;  // 声明内存管理对象
uint8_t ucArray [ 3 ] [ 20 ];  // 声明内存分区大小

static OS_TCB AppTaskStartTCB;  // 任务控制块
static OS_TCB AppTaskPostTCB;
static OS_TCB AppTaskPendTCB;

static CPU_STK AppTaskStartStk[APP_TASK_START_STK_SIZE];  // 任务栈
static CPU_STK AppTaskPostStk [ APP_TASK_POST_STK_SIZE ];
static CPU_STK AppTaskPendStk [ APP_TASK_PEND_STK_SIZE ];

static void AppTaskStart ( void *p_arg );  // 任务函数声明
static void AppTaskPost ( void * p_arg );
static void AppTaskPend ( void * p_arg );

int main ( void )
{
    OS_ERR err;
}
```

(下页继续)
OSInit(&err);
//初始化μC/OS-III

/* 创建起始任务 */
OSTaskCreate((OS_TCB *)&AppTaskStartTCB,
   //任务控制块地址
   (CPU_CHAR *)"App Task Start",
   //任务名称
   (OS_TASK_PTR ) AppTaskStart,
   //任务函数
   (void *) 0,
   //传递给任务函数（形参 p_arg）的实参
   (OS_PRIO ) APP_TASK_START_PRIO,
   //任务的优先级
   (CPU_STK *) &AppTaskStartStk[0],
   //任务栈的基地址
   (CPU_STK_SIZE) APP_TASK_START_STK_SIZE / 10,
   //任务栈空间剩下 1/10 时限制其增长
   (CPU_STK_SIZE) APP_TASK_START_STK_SIZE,
   //任务栈空间（单位：sizeof(CPU_STK)）
   (OS_MSG_QTY ) 5u,
   //任务可接收的最大消息数
   (OS_TICK ) 0u,
   //任务的时间片节拍数（0 表默认值 OSCfg_TickRate_Hz/10）
   (void *) 0,
   //任务扩展（0 表不扩展）
   (OS_OPT ) (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
   //任务选项
   (OS_ERR *) &err);
   //返回错误类型
OSStart(&err);
// 启动多任务管理（由 μC/OS-III 控制）
}

static void AppTaskStart (void *p_arg)
{
    CPU_INT32U cpu_clk_freq;
    CPU_INT32U cnts;
    OS_ERR err;

    (void)p_arg;

    BSP_Init();  // 板级初始化
    CPU_Init();  // 初始化 CPU 组件（时间戳、关中断时间测量和主机名）

    cpu_clk_freq = BSP_CPU_ClkFreq();
    cnts = cpu_clk_freq / (CPU_INT32U)OSCfg_TickRate_Hz;
    OS_CPU_SysTickInit(cnts);

    Mem_Init();  // 初始化内存管理组件（堆内存池和内存池表）

#if OS_CFG_STAT_TASK_EN > 0u  // 如果启用（默认启用）了统计任务
    OSStatTaskCPUUsageInit(&err);
#endif

    CPU_IntDisMeasMaxCurReset();  // 复位（清零）当前最大关中断时间

/* 创建内存管理对象 mem */
OSMemCreate ((OS_MEM     *) &mem,  // 指向内存管理对象
    (CPU_CHAR    *) "Mem For Test",  // 命名内存管理对象
    (void       *) ucArray,  // 内存分区的首地址
    (OS_MEM_QTY  ) 3,  // 内存分区内内存块数目
    (OS_MEM_SIZE ) 20,  // 内存块的字节数目

论坛：https://www.firebbs.cn/  732  天猫：https://yehuosm.tmall.com
/* 创建 AppTaskPost 任务 */
OSTaskCreate((OS_TCB *) &AppTaskPostTCB,
   // 任务控制块地址
   (CPU_CHAR *) "App Task Post",
   // 任务名称
   (OS_TASK_PTR ) AppTaskPost,
   // 任务函数
   (void *) 0,
   // 传送给任务函数 (形参 p_arg) 的实参
   (OS_PRIO ) APP_TASK_POST_PRIO,
   // 任务的优先级
   (CPU_STK *) &AppTaskPostStk[0],
   // 任务栈的基地址
   (CPU_STK_SIZE) APP_TASK_POST_STK_SIZE / 10,
   // 任务栈空间剩下 1/10 时限制其增长
   (CPU_STK_SIZE) APP_TASK_POST_STK_SIZE,
   // 任务栈空间（单位：sizeof(CPU_STK)）
   (OS_MSG_QTY ) 5u,
   // 任务可接收的最大消息数
   (OS_TICK ) 0u,
   // 任务的时间片节拍数（0 表默认值 OSCfg_TickRate_Hz/10）
   (void *) 0,
   // 任务扩展（0 表不扩展）
   (OS_OPT ) (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
   // 任务选项
   (OS_ERR *) &err);
   // 返回错误类型

/* 创建 AppTaskPend 任务 */
OSTaskCreate((OS_TCB    *) &AppTaskPendTCB,  
    //任务控制块地址
    (CPU_CHAR    *)"App Task Pend",  
    //任务名称
    (OS_TASK_PTR ) AppTaskPend,  
    //任务函数
    (void        *) 0,  
    //传递给任务函数（形参 p_arg）的实参
    (OS_PRIO      ) APP_TASK_PEND_P prio,  
    //任务的优先级
    (CPU_STK      *) &AppTaskPendStk[0],  
    //任务栈的地址
    (CPU_STK_SIZE) APP_TASK_PEND_STK_SIZE / 10,  
    //任务栈空间剩下 1/10 时限制其增长
    (CPU_STK_SIZE) APP_TASK_PEND_STK_SIZE,  
    //任务栈空间（单位: sizeof(CPU_STK)）
    (OS_MSG_QTY   ) 50u,  
    //任务可接收的最大消息数
    (OS_TICK      ) 0u,  
    //任务的时间片节拍数（0 表默认值 OSCfg_TickRate_Hz/10）
    (void        *) 0,  
    //任务扩展（0 表不扩展）
    (OS_OPT       ) (OS_OPT_TASK_STK_CHK |  OS_OPT_TASK_STK_CLR),  
    //任务选项
    (OS_ERR       *) &err);  
    //返回错误类型
    OSTaskDel (  & AppTaskStartTCB,  & err );  
    //删除起始任务本身，该任务不再运行
}

static void AppTaskPost ( void * p_arg )
{

OS_ERR err;
char * p_mem_blk;
uint32_t ulCount = 0;

(void)p_arg;

while (DEF_TRUE) //任务体
{
    /* 向 mem 获取内存块 */
    p_mem_blk = OSMemGet ((OS_MEM *)&mem,
                          (OS_ERR *)&err); //指向内存管理对象

    sprintf ( p_mem_blk, "%d", ulCount ++ );
    //向内存块存取计数值

    /* 发布任务消息到任务 AppTaskPend */
    OSTaskQPost ((OS_TCB *)&AppTaskPendTCB,
                 (void *)p_mem_blk,
                 (OS_MSG_SIZE )strlen ( p_mem_blk ), //消息长度
                 (OS_OPT )OS_OPT_POST_FIFO, //发布到任务消息队列的入口端
                 (OS_ERR *)&err); //返回错误类型
    OSTimeDlyHMSM ( 0, 0, 1, 0, OS_OPT_TIME_DLY, & err);
}

static void AppTaskPend ( void * p_arg )
{
    OS_ERR err;
OS_MSG_SIZE    msg_size;
CPU_TS         ts;
CPU_INT32U     cpu_clk_freq;
CPU_SR_ALLOC();

char * pMsg;
(void)p_arg;

cpu_clk_freq = BSP_CPU_ClkFreq();
//获取 CPU 时钟，时间戳是以该时钟计数

while (DEF_TRUE)     //任务体
{
    /* 阻塞任务，等待任务消息 */
    pMsg = OSTaskQPend ((OS_TICK )0,     //无期限等待
                         (OS_OPT )OS_OPT_PEND_BLOCKING,
                         (OS_MSG_SIZE *)&msg_size,   //返回消息长度
                         (CPU_TS *)&ts,           //返回消息被发出的时间戳
                         (OS_ERR *)&err);       //返回错误类型

    ts = OS_TS_GET() - ts;
    //计算消息从发布到被接收的时间差

    macLED1_TOGGLE ();    //切换 LED1 的亮灭状态

    OS_CRITICAL_ENTER(); //进入临界段，避免串口打印被打断

    printf ( "\n接收到的消息的内容为: %s，长度是: %d 字节。", pMsg, msg_size);
}
29.6 内存管理实验现象

程序编译好，用 USB 线连接计算机和开发板的 USB 接口（对应丝印为 USB 转串口），用 DAP 仿真器把配套程序下载到野火 STM32 开发板（具体型号根据购买的板子而定，每个型号的板子都配有对应的程序），在计算机上打开串口调试助手，然后复位开发板就可以在调试助手中看到串口的打印信息与运行结果，具体见图静态内存管理实验现象。
第 30 章 中断管理

30.1 异常与中断的基本概念

异常是导致处理器脱离正常运行转向执行特殊代码的任何事件，如果不及时进行处理，轻则系统出错，重则会导致系统毁灭性瘫痪。所以正确地处理异常，避免错误的发生是提高软件鲁棒性（稳定性）非常重要的一环，对于实时系统更是如此。

异常是指任何打断处理器正常执行，并且迫使处理器进入一个由有特权的特殊指令执行的事件。异常通常可以分成两类：同步异常和异步异常。由内部事件（像处理器指令运行产生的事件）引起的异常称为同步异常，例如造成被零除的算术运算引发一个异常，又如在某些处理器体系结构中，对于确定的数据尺寸必须从内存的偶数地址进行读和写操作。从一个奇数内存地址的读或写操作将引起存储器存取一个错误事件并引起一个异常（称为未准异常）。

异步异常主要是指由于外部异常源产生的异常，是一个由外部硬件装置产生的事件引起的异常。同步异常不同于异步异常的地方是事件的来源，同步异常事件是由于执行某些指令而从处理器内部产生的，而异步异常事件的来源是外部硬件装置。例如按下设备某个按钮产生的事件。同步异常与异步异常的区别还在于，同步异常触发后，系统必须立刻进行处理而不能够依然执行原有的程序指令步骤；而异步异常则可以延缓处理甚至是忽略，例如按键中断异常，虽然中断异常触发了，但是系统可以选择它继续运行（同样也忽略了相应的按键事件）。

中断，中断属于异步异常。所谓中断是指中央处理器 CPU 在处理某件事的时候，外部发生了某一事件，请求 CPU 迅速处理，CPU 暂时中断当前的工作，转入处理所发生的事件，处理完后，再回到原来被中断的地方，继续原来的工作，这样的过程称为中断。

中断能打断任务的执行，无论该任务具有什么样的优先级，因此中断一般用于处理比较紧急的事件，而且只做简单处理，例如标记该事件，在使用 μC/OS 系统时，一般建议使用信号量、消息或事件标志组等标志中断的发生，将这些内核对象发布给处理任务，处理任务再做具体处理。

通过中断机制，在外设不需要 CPU 介入时，CPU 可以执行其他任务，而当外设需要 CPU 时通过产生中断信号使 CPU 立即停止当前任务转而响应中断请求。这样可以使 CPU 避免把大量时间
耗费在等待、查询外设状态的操作上，因此将大大提高系统实时性以及执行效率。

此处读者要知道一点，μC/OS 源码中有许多处临界段的地方，临界段虽然保护了关键代码的执行不被打断，但也会影响系统的实时性，任何使用了操作系统的中断响应都不会比裸机快。比如，某
个时候有一个任务在运行中，并且该任务部分程序将中断屏蔽掉，也就是进入临界段中，这个时候
如果有一个紧急的中断事件被触发，这个中断就会被挂起，不能得到及时响应，必须等到中断
开启才可以得到响应，如果屏蔽中断时间超过了紧急中断能够容忍的限度，危害是可想而知的。
操作系统中的中断在某些时候会产生产必要的中断延迟，因此调用中断屏蔽函数进入临界段的时候，
也需快进快出。

μC/OS 的中断管理支持：

- 开/关中断。
- 恢复中断。
- 中断启用。
- 中断屏蔽。
- 中断嵌套。
- 中断延迟发布。

### 30.1.1 中断的介绍

与中断相关的硬件可以划分为三类：外设、中断控制器、CPU 本身。

外设：当外设需要请求 CPU 时，产生一个中断信号，该信号连接至中断控制器。

中断控制器：中断控制器是 CPU 众多外设中的一个，它一方面接受其他外设中断信号的输入，另一
一方面，它会发出中断信号给 CPU。可以通过对中断控制器编程实现对中断源的优先级、触发方
式、打开和关闭源等设置操作。在 Cortex-M 系列控制器中常用的中断控制器是 NVIC（内嵌向量
中断控制器 Nested Vectored Interrupt Controller）。

CPU：CPU 会响应中断源的请求，中断当前正在执行的任务，转而执行中断处理程序。NVIC 最
多支持 240 个中断，每个中断最多 256 个优先级。
30.1.2 和中断相关的名词解释

- 中断号：每个中断请求信号都会有特定的标志，使得计算机能够判断是哪个设备提出的中断请求，这个标志就是中断号。

- 中断请求：“紧急事件”需向 CPU 提出申请，要求 CPU 暂停当前执行的任务，转而处理该“紧急事件”，这一申请过程称为中断请求。

- 中断优先级：为使系统能够及时响应并处理所有中断，系统根据中断时间的重要性和紧迫程度，将中断源分为若干个级别，称作中断优先级。

- 中断处理程序：当外设产生中断请求后，CPU 暂停当前的任务，转而响应中断申请，即执行中断处理程序。

- 中断触发：中断源发出并送给 CPU 控制信号，将中断触发器置“1”，表明该中断源产生了中断，要求 CPU 去响应该中断，CPU 暂停当前任务，执行相应的中断处理程序。

- 中断触发类型：外部中断申请通过一个物理信号发送到 NVIC，可以是电平触发或边沿触发。

- 中断向量：中断服务程序的入口地址。

- 中断向量表：存储中断向量的存储区，中断向量与中断号对应，中断向量在中断向量表中按照中断号顺序存储。

- 临界段：代码的临界段也称为临界区，一旦这部分代码开始执行，则不允许任何中断打断。为确保临界段代码的执行不被中断，在进入临界段之前须关中断，而临界段代码执行完毕后，要立即开中断。

30.2 中断的运作机制

当中断产生时，处理机将按如下的顺序执行：

1. 保存当前处理机状态信息

2. 载入异常或中断处理函数到 PC 寄存器
3. 把控制权转交给处理函数并开始执行

4. 当处理函数执行完成时，恢复处理器状态信息

5. 从异常或中断中返回到前一个程序执行点

中断使得 CPU 可以在事件发生时才给予处理，而不必让 CPU 连续不断地查询是否有相应的事件发生。通过两条特殊指令：关中断和开中断可以让处理器不响应或响应中断，在关闭中断期间，通常处理器会把新产生的中断挂起，当中断打开时立刻进行响应，所以会有适当的延时响应中断，故用户在进入临界区的时候应退进快出。

中断发生的环境有两种情况：在任务的上下文中，在中断服务函数处理上下文中。

- 任务在工作的时候，如果此时发生了一个中断，无论中断的优先级是多大，都会打断当前任务的执行，从而转到对应的中断服务函数中执行，其过程具体见图中断发生在任务上下文。

- 中断发生在任务上下文 (1)、(3)：在任务运行的时候发生了中断，那么中断会打断任务的运行，那么操作系统将先保存当前任务的上下文环境，转而去处理中断服务函数。

- 中断发生在任务上下文 (2)、(4)：当且仅当中断服务函数处理完的时候才恢复任务的上下文环境，继续运行任务。

- 在执行中断服务例程的过程中，如果有更高优先级别的中断源触发中断，由于当前处于中

论坛：https://www.firebbs.cn/  742  天猫：https://yehuosm.tmall.com
断处理上下文环境中，根据不同的处理器构架可能有不同的处理方式，比如新的中断等待挂起直到当前中断处理离开后再行响应；或新的高优先级中断打断当前中断处理过程，而去直接响应这个更高优先级的新中断源。后面这种情况，称之为中断嵌套。在硬实时环境
中，前一种情况是不允许发生的，不能使响应中断的时间尽量的短。而在软件处理（软实时环境）上，μC/OS 允许中断嵌套，即在一个中断服务例程期间，处理器可以响应另外一个优先级更高的中断，过程如图中断嵌套发生 所示。

- 中断嵌套发生 (1): 当中断 1 的服务函数在处理的时候发生了中断 2，由于中断 2 的优先级比中断 1 更高，所以发生了中断嵌套，那么操作系统将先把保存当前中断服务函数的上下文环境，并且转向处理中断 2，当且仅当中断 2 执行完的时候图中断嵌套发生 (2)，才能继续执行中断 1。

论坛: https://www.firebbs.cn/    天猫: https://yehuosm.tmall.com
30.3 中断延迟的概念

即使操作系统的响应很快了，但对于中断的处理仍然存在着中断延迟响应的问题，我们称之为中断延迟 (Interrupt Latency)。

中断延迟是指从硬件中断发生到开始执行中断处理程序第一条指令之间的这段时间。也就是：系统接收到中断信号到操作系统作出响应，并完成换到转入中断服务程序的时间。也可以简单地理解为：（外部）硬件（设备）发生中断，到系统执行中断服务子程序（ISR）的第一条指令的时间。

中断的处理过程是：外界硬件发生了中断后，CPU 到中断处理器读取中断向量，并且查找中断向量表，找到对应的中断服务子程序（ISR）的首地址，然后跳转到对应的 ISR 去做相应处理。这部分时间，我称之为：识别中断时间。

在允许中断嵌套的实时操作系统中，中断也是基于优先级的，允许高优先级中断抢占正在处理的低优先级中断，所以，如果当前正在处理更高优先级的中断，即使此时有低优先级的中断，也系统不会立刻响应，而是等到高优先级的中断处理完之后，才会响应。而即使在不支持中断嵌套，即中断是没有优先级的，中断是不允许被中断的，所以，如果当前系统正在处理一个中断，而此时另一个中断到来了，系统也是不会立即响应的，而只是等处理完当前的中断之后，才会处理后来的中断。此部分时间，我称其为：等待中断打开时间。

在操作系统中，很多时候我们会主动进入临界段，系统不允许当前状态被中断打断，故而在临界区发生的中断会被挂起，直到退出临界段时候打开中断。此部分时间，我称其为：关闭中断时间。

中断延迟可以定义为，从中断开始的时刻到中断服务例程开始执行的时刻之间的时间段。中断延迟 = 识别中断时间 + [等待中断打开时间] + [关闭中断时间]。

注意：[ ] 的时间是不一定都存在的，此处为最大可能的中断延迟时间。

此外，中断恢复时间定义为：执行完 ISR 中最后一句代码后到恢复到任务级代码的这段时间。

任务延迟时间定义为：中断发生到恢复到任务级代码的这段时间。
30.4 中断的应用场景

中断在嵌入式处理器中应用非常之多，没有中断的系统不是一个好系统，因为有中断，才能启动或者停止某件事情，从而转去做另一件事情。我们可以举一个日常生活中的例子来说明，假如你正在给朋友写信，电话铃响了，这时你放下手中的笔去接电话，通话完毕再继续写信。这个例子就表现了中断及其处理的过程：电话铃声使你暂时中止当前的工作，而去处理更为急需处理的事情——接电话，当把急需处理的事情处理完毕之后，再回过头来继续原来的事情。在这个例子中，电话铃声就可以称为“中断请求”，而你暂停写信去接电话就称作“中断响应”，那么接电话的过程就是“中断处理”。由此我们可以看出，在计算机执行程序的过程中，由于出现某个特殊情况（或称为“特殊事件”），使得系统暂时中止现行程序，而转去执行处理这一特殊事件的程序，处理完毕之后再回到原来程序的中断点继续向下执行。

为什么说没有中断的系统不是好系统呢？我们可以再举一个例子来说明中断的作用。假设有一个朋友来拜访你，但是由于不知何时到达，你只能在门口等待，于是什么事情也干不了；但如果在门口装一个门铃，你就不必在门口等待而可以在家里去做其他的工作。朋友来了按门铃通知你，这时你才中断手中的工作去开门，这就避免了不必要的等待。CPU 也是一样，如果时间都浪费在查询的事情上，那这个 CPU 啥也干不了，要他何用。在嵌入式系统中合理利用中断，能更好利用 CPU 的资源。

30.5 中断管理讲解

ARM Cortex-M 系列内核的中断是由硬件管理的，而 μC/OS 是软件，它并不接管由硬件管理的相关中断（接管简单来说就是，所有的中断都由 RTOS 的软件管理，硬件来了中断时，由软件决定是否响应，当没有中断时，延迟响应或者不响应），只支持简单的开关中断等，所以 μC/OS 中的中断使用其实跟裸机差不多的，需要我们自己配置中断，并且启用中断，编写中断服务函数，在中断服务函数中使用内核 IPC 通信机制，一般建议使用信号量，消息或事件标志组等标志事件的发生，将事件发布给处理任务，等退出中断后再由相关处理任务具体处理中断，当然 μC/OS 为了能让系统更快退出中断，它支持中断延迟发布，将中断级的发布变成任务级（在后文讲解）。

ARM Cortex-M NVIC 支持中断嵌套功能：当一个中断触发并且系统进行响应时，处理器硬件会将当前运行的部分上下文寄存器自动压入中断栈中，这部分的寄存器包括 PSR，R0，R1，R2，R3
以及 R12 寄存器。当系统正在服务一个中断时，如果有一个更高优先级的中断触发，那么处理器同样的会打断当前运行的中断服务例程，然后把老的中断服务例程上下文的 PSR，R0，R1，R2，R3 和 R12 寄存器自动保存到中断栈中。这些部分上下文寄存器保存到中断栈的行为完全是硬件行为，这一点是与其他 ARM 处理器最大的区别（以往都需要依赖于软件保存上下文）。

另外，在 ARM Cortex-M 系列处理器上，所有中断都采用中断向量表的方式进行处理，即当一个中断触发时，处理器将直接判定是哪个中断源，然后直接跳转到相应的固定位置进行处理。而在 ARM7、ARM9 中，一般是先跳转进入 IRQ 入口，然后再由软件进行判断是哪个中断源触发，获得了相对应的中断服务例程入口地址后，再进行后续的中断处理。ARM7、ARM9 的好处在于，所有中断它们都有统一的入口地址，便于 OS 的统一管理。而 ARM Cortex-M 系列处理器则恰恰相反，每个中断服务例程必须排列在一起放在统一的地址上（这个地址必须要设置到 NVIC 的中断向量偏移寄存器中）。中断向量表一般由一个数组定义（或在起始代码中给出），在 STM32 上，默认采用起始代码给出：具体见代码清单: 中断管理-1。

### 列表 1: 代码清单: 中断管理-1 中断向量表（部分）

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>_Vectors</td>
<td>DCD</td>
<td>_initial_sp</td>
</tr>
<tr>
<td>DCD</td>
<td>Reset_Handler</td>
<td>; Reset Handler</td>
</tr>
<tr>
<td>DCD</td>
<td>NMI_Handler</td>
<td>; NMI Handler</td>
</tr>
<tr>
<td>DCD</td>
<td>HardFault_Handler</td>
<td>; Hard Fault Handler</td>
</tr>
<tr>
<td>DCD</td>
<td>MemManage_Handler</td>
<td>; MPU Fault Handler</td>
</tr>
<tr>
<td>DCD</td>
<td>BusFault_Handler</td>
<td>; Bus Fault Handler</td>
</tr>
<tr>
<td>DCD</td>
<td>UsageFault_Handler</td>
<td>; Usage Fault Handler</td>
</tr>
<tr>
<td>DCD</td>
<td>0</td>
<td>; Reserved</td>
</tr>
<tr>
<td>DCD</td>
<td>0</td>
<td>; Reserved</td>
</tr>
<tr>
<td>DCD</td>
<td>0</td>
<td>; Reserved</td>
</tr>
<tr>
<td>DCD</td>
<td>0</td>
<td>; Reserved</td>
</tr>
<tr>
<td>DCD</td>
<td>SVC_Handler</td>
<td>; SVC Handler</td>
</tr>
<tr>
<td>DCD</td>
<td>DebugMon_Handler</td>
<td>; Debug Monitor Handler</td>
</tr>
<tr>
<td>DCD</td>
<td>0</td>
<td>; Reserved</td>
</tr>
<tr>
<td>DCD</td>
<td>PendSV_Handler</td>
<td>; PendSV Handler</td>
</tr>
<tr>
<td>DCD</td>
<td>SysTick_Handler</td>
<td>; SysTick Handler</td>
</tr>
</tbody>
</table>

(下页继续)
μC/OS 在 Cortex-M 系列处理器上也遵循与裸机中断一致的方法，当用户需要使用自定义的中断服务例程时，只需要定义相同名称的函数覆盖弱化符号即可。所以，μC/OS 在 Cortex-M 系列处理器的中断控制其实与裸机没什么差别，不过在进入中断与退出中断的时候需要调用一下 OSIntEnter() 函数和 OSIntExit() 函数，方便中断嵌套管理。
30.6 中断延迟发布

30.6.1 中断延迟发布的概念

μC/OS-III 有两种方法处理来自于中断的事件，直接发布（或者称为释放）和延迟发布。通过 os_cfg.h 中的 OS_CFG_ISR_POST_DEFERRED_EN 来选择，当设置为 0 时，μC/OS 使用直接发布的方法。当设置为 1 时，使用延迟发布方法，用户可以根据自己设计系统的应用选择其中一种方法即可。

启用中断延时发布，可以将中断级发布转换成任务级发布，而且在进入临界段时也可以使用锁调度器代替关中断，这就大大减小了关中断时间，有利于提高系统的实时性（能实时响应中断而不受中断屏蔽导致响应延迟）。在前面提到的 OSTimeTick()、OSSemPost()、OSQPost()、OSFlagPost()、OSTaskSemPost()、OSTaskQPost()、OSTaskSuspend() 和 OSTaskResume() 等这些函数，如果没有使用中断延迟发布，那么调用这些函数意味着进入一段很长的临界段，也就是要关中断很长时间。在启用中断延时发布后，如果在中断中调用这些函数，系统就会将这些 post 提交函数必要的信息保存到中断延时提交的变量中去，为了配合中断延迟，μC/OS 还将创建了优先级最高（优先级为 0）的任务——中断发布函数 OS_IntQTask，退出中断后根据之前保存的参数，在任务中再次进行 post 相关操作。这个过程其实就是把中断中的临界段放到任务中来实现，这个时候进入临界段就可以用锁住调度器的方式代替了关中断，因此大大减少了关中断的时间，系统将 post 操作延迟了，中断延迟就是这么来的。

进入临界段的方式可以是关中断或者锁住调度器，系统中有些变量不可能在中断中被访问，所以只要保证其他任务不要使用这些变量即可，这个时候就可以用锁调度启动的方式，用锁住调度代替关中断，减少了关中断的时间，也能达到进入临界段的目的。中断延迟就是利用这种思想，让本该在中断中完成的事情切换到任务中完成，而且进入临界段的方式是锁定调度器，这样子中断就不会被屏蔽，系统能随时响应中断，并且，整个中断延迟发布的过程不影响 post 的效果，因为 μC/OS 已经设定中断发布任务的优先级为最高，在退出中断后会马上进行 post 操作，这与在中断中直接进行 post 操作的时间基本一致。

注：操作系统内核相关函数一般为了保证其操作的完整性，一般都会进入或长或短的临界段，所以在中断的要尽量少调用内核函数。部分 μC/OS 提供的函数是不允许在中断中调用的。

在直接发布方式中，μC/OS 访问临界段时是采用关中断方式。然而，在延迟提交方式中，μC/OS
访问临界段时是采用锁调度器方式。在延迟提交方式中，访问中断队列时μC/OS 仍需要关中断进入临界段，但是这段关中断时间是非常短的且是固定的。

下面来看看中断延迟发布与直接发布的区别，具体见图中断延迟发布 图中断直接发布。

- 中断延迟发布 (1): 进入中断，在中断中需要发布一个内核对象（如消息队列、信号量等），但是使用了中断延迟发布，在中断中值执行 OS_IntQPost() 函数，在这个函数中，采用关中断方式进入临界段，因此在这个时间段是不能响应中断的。

- 中断延迟发布 (2): 已经将内核对象发布到中断消息队列，那么将唤醒 OS_IntQTask 任务，因为该任务是最高优先级任务，所以能立即被唤醒，然后转到 OS_IntQTask 任务中发布内核对象，在该任务中，调用 OS_IntQRePost() 函数进行发布内核对象，进入临界段的方式采用锁调度器方式，那么在这个阶段，中断是可以被响应的。

- 中断延迟发布 (3): 系统正常运行，任务按优先级进行切换。
• 中断直接发布 (1)、(2)：而采用中断直接发布的情况是在中断中直接屏蔽中断以进入临界段，这段时间中，都不会响应中断，直到发布完成，系统任务正常运行才开启中断。

• 中断直接发布 (3)：系统正常运行，任务按照优先级正常切换

从两个图中我们可以看出，很明显，采用中断延迟发布的效果更好，将本该在中断中的处理转变成为在任务中处理，系统关中断的时间大大降低，使得系统能很好地响应外部中断，如果在应用中中断时间是关键性的，应用中有非常频繁的中断源，且应用不能接受直接发布方式那样较长的关中断时间，推荐使用中断延迟发布方式。

### 30.6.2 中断队列控制块

如果启用中断延迟发布，在中断中调用内核对象发布（释放）函数，系统会将发布的内容存放在中断队列中控制块中，源码具体见代码清单: 中断管理-2

<table>
<thead>
<tr>
<th>列表 2: 代码清单: 中断管理-2 中断队列信息块</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

(下页继续)
• 代码清单: 中断管理-2 (1): 用于发布的内核对象类型，例如消息队列、信号量、事件等。
• 代码清单: 中断管理-2 (2): 指向下一个中断队列控制块。
• 代码清单: 中断管理-2 (3): 指向内核对象变量指针。
• 代码清单: 中断管理-2 (4): 如果发布的是任务消息或者是内核对象消息，指向发布消息的指针。
• 代码清单: 中断管理-2 (5): 如果发布的是任务消息或者是内核对象消息，记录发布的消息的字节大小。
• 代码清单: 中断管理-2 (6): 如果发布的是事件标志，该成员变量记录要设置事件的标志位。
• 代码清单: 中断管理-2 (7): 记录发布内核对象时的选项。
• 代码清单: 中断管理-2 (8): 记录时间戳。
30.6.3 中断延迟发布任务初始化 OS_IntQTaskInit()

在系统初始化的时候，如果我们启用了中断延迟发布，那么系统会根据我们自定义配置中断延迟发布任务的宏定义 OS_CFG_INT_Q_SIZE 与 OS_CFG_INT_Q_TASK_STK_SIZE 进行相关初始化，这两个宏定义在 os_cfg_app.h 文件中，中断延迟发布任务的初始化具体见代码清单: 中断管理-3。

列表 3: 代码清单: 中断管理-3 中断延迟发布任务初始化

```c
void OS_IntQTaskInit (OS_ERR *p_err) {
    OS_INT_Q *p_int_q;
    OS_INT_Q *p_int_q_next;
    OS_OBJ_QTY i;

    #ifdef OS_SAFETY_CRITICAL
    if (p_err == (OS_ERR *)0) {
        OS_SAFETY_CRITICAL_EXCEPTION();
        return;
    }
    #endif

    /* 清空延迟提交过程中溢出的计数值 */
    OSIntQOvfCtr = (OS_QTY)0u;

    // 延迟发布信息队列的基地址必须不为空指针
    if (OSCfg_IntQBasePtr == (OS_INT_Q *)0) (1)
    {
        *p_err = OS_ERR_INT_Q;
        return;
    }
}
```

(下页继续)
// 延迟发布队列成员必须不小于 2 个
if (OSCfg_IntQSize < (OS_OBJ_QTY)2u) 
{
    *p_err = OS_ERR_INT_Q_SIZE;
    return;
}

// 初始化延迟发布任务每次运行的最长时间记录变量
OSIntQTaskTimeMax = (CPU_TS) 0;

// 将定义的数据连接成一个单向链表
p_int_q = OSCfg_IntQBasePtr; 
        (3)
p_int_q_next = p_int_q;
p_int_q_next++; 
for (i = 0u; i < OSCfg_IntQSize; i++)
{
    // 每个信息块都进行初始化
    p_int_q->Type = OS_OBJ_TYPE_NONE;
p_int_q->ObjPtr = (void *) 0;
p_int_q->MsgPtr = (void *) 0;
p_int_q->MsgSize = (OS_MSG_SIZE) 0u;
p_int_q->Flags = (OS_FLAGS) 0u;
p_int_q->Opt = (OS_OPT) 0u;
p_int_q->NextPtr = p_int_q_next;
p_int_q++;
p_int_q_next++;
}

// 将单向链表的首尾相连组成一个“圈
p_int_q--;
p_int_q_next = OSCfg_IntQBasePtr;
p_int_q->NextPtr = p_int_q_next; 
(4)
//队列出口和入口都指向第一个
OSIntQInPtr = p_int_q_next;
OSIntQOutPtr = p_int_q_next; (5)

//清空延迟发布队列中需要进行发布的内核对象个数
OSIntQNbrEntries = (OS_OBJ_QTY)0u;

//清空延迟发布队列中历史发布的内核对象个数
OSIntQNbrEntriesMax = (OS_OBJ_QTY)0u;

if (OSCfg_IntQTaskStkBasePtr == (CPU_STK *)0)
{
  *p_err = OS_ERR_INT_Q_STK_INVALID;
  return;
}

if (OSCfg_IntQTaskStkSize < OSCfg_StkSizeMin)
{
  *p_err = OS_ERR_INT_Q_STK_SIZE_INVALID;
  return;
}

//创建延迟发布任务
OSTaskCreate((OS_TCB *)&OSIntQTaskTCB,
              (CPU_CHAR *) (void *)"μC/OS-III ISR Queue Task",
              (OS_TASK_PTR )OS_IntQTask,
              (void * )0,
              (OS_PRIO )0u, //优先级最高
              (CPU_STK *)OSCfg_IntQTaskStkBasePtr,
              (CPU_STK_SIZE)OSCfg_IntQTaskStkLimit,
              (CPU_STK_SIZE)OSCfg_IntQTaskStkSize,
              (OS_MSG_QTY )0u,

(下页继续)
(OS_TICK ) 0u,
(void ) 0,
(OS_OPT ) (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
(OS_ERR ) p_err);

#endif

- 代码清单: 中断管理-3 (1): 延迟发布信息队列的基地址必须不为空指针，μC/OS 在编译的时候就已经静态分配一个存储的空间（大数组），具体见代码清单: 中断管理-4。
列表 4: 代码清单: 中断管理-4 中断延迟发布队列存储空间（位于 os_cfg_app.c）

```c
#if (OS_CFG_ISR_POST_DEFERRED_EN > 0u)
OS_INT_Q   OSCfg_IntQ  [OS_CFG_INT_Q_SIZE];
CPU_STK    OSCfg_IntQTaskStk [OS_CFG_INT_Q_TASK_STK_SIZE];
#endif

OS_INT_Q   * const  OSCfg_IntQBasePtr = (OS_INT_Q *) &OSCfg_IntQ[0];
OS_OBJ_QTY  const  OSCfg_IntQSize   = (OS_OBJ_QTY )OS_CFG_INT_Q_ SIZE;
```

- 代码清单: 中断管理-3 (2): 延迟发布队列成员 (OSCfg_IntQSize = OS_CFG_INT_Q_SIZE) 必须不小于 2 个，该宏在 os_cfg_app.h 文件中定义。

- 代码清单: 中断管理-3 (3): 将定义的数据连接成一个单向链表，并且初始化每一个信息块的内容。

- 代码清单: 中断管理-3 (4): 将单向链表的首尾相连组成一个 “圈”，环形单链表，处理完成示意图具体见图中断延迟发布队列初始化完成示意图。

论坛: https://www.firebbs.cn/    天猫: https://yehuosm.tmall.com
• 代码清单: 中断管理-3 (6)：创建延迟发布任务，任务的优先级是 0，是最高效优先级任务不允许用户修改。

30.6.4 中断延迟发布过程 OS_IntQPost()

如果启用了中断延迟发布，并且发送消息的函数是在中断中被调用，此时就不应该立即发送消息，而是将消息的发送放在指定发布任务中，此时系统将消息发布到任务消息队列中，等待到中断发布任务唤醒后再发送消息，OS_IntQPost() 源码具体见代码清单: 中断管理-5。

提示：为了阅读方便，将“中断延迟发布队列”简称为“中断队列”。

列表 5: 代码清单: 中断管理-5 OS_IntQPost() 源码

```c
void OS_IntQPost (OS_OBJ_TYPE type, (1) //内核对象类型
    void *p_obj, (2) //被发布的内核对象
    void *p_void, (3) //消息队列或任务消息
    OS_MSG_SIZE msg_size, (4) //消息的数目
    OS_FLAGS flags, (5) //事件
    OS_OPT opt, (6) //发布内核对象时的选项
    CPU_TS ts, (7) //发布内核对象时的时间戳
    OS_ERR *p_err) (8) //返回错误类型
{
    CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必须用到该宏，该宏声明和定义一个
    //局部变量，用于保存关中断前的 CPU 状态寄存器 SR（临界段关中断只需保存 SR）
    //，开中断时将该值还原。
    #ifdef OS_SAFETY_CRITICAL(9) //如果启用（默认禁用）了安全检测
    if (p_err == (OS_ERR *)0) { //如果错误类型参数为空
        OS_SAFETY_CRITICAL_EXCEPTION(); //执行安全检测异常函数
        return; //返回，不继续执行
    }
    #endif
```

(下页继续)
CPU_CRITICAL_ENTER();  // 关中断
if (OSIntQNbrEntries < OSCfg_IntQSize) { (10) // 如果中断队列未占满
    OSIntQNbrEntries++; (11)
    // 更新中断队列的最大使用数目的历史记录
} (12) // 如果中断队列未占满

if (OSIntQNbrEntriesMax < OSIntQNbrEntries) {
    OSIntQNbrEntriesMax = OSIntQNbrEntries;
} /* 将要重新提交的内核对象的信息放到中断队列人口的信息记录块 */

OSIntQInPtr->Type = type; /* 保存要发布的对象类型 */
OSIntQInPtr->ObjPtr = p_obj; /* 保存指向要发布的对象的指针 */
OSIntQInPtr->MsgPtr = p_void; /* 将信息保存到消息块的中 */
OSIntQInPtr->MsgSize = msg_size; /* 保存信息的大小 */
OSIntQInPtr->Flags = flags; /* 如果发布到事件记标志，则保存标志 */
OSIntQInPtr->Opt = opt; /* 保存选项 */
OSIntQInPtr->TS = ts; /* 保存时间戳信息 */ (13)

OSIntQInPtr = OSIntQInPtr->NextPtr; (14) // 指向下ー个中断队列入口
/* 让中断队列管理任务 OSIntQTask 就绪 */
OSRdyList[0].NbrEntries = (OS_OBJ_QTY); // 更新就绪列表上的优先级 0 的任务数为 1 个
/* 就绪列表的头尾指针都指向 OSIntQTask 任务 */
OSRdyList[0].HeadPtr = &OSIntQTaskTCB;
OSRdyList[0].TailPtr = &OSIntQTaskTCB; (15) // 在优先级列表中增加优先级 0
if (OSPrioCur != 0) { (16) // 如果当前运行的不是 OSIntQTask 任务
    OSPrioSaved = OSPrioCur; // 保存当前任务的优先级
}

*p_err = OS_ERR_NONE; // 返回错误类型为“无错误”
else { // 如果中断队列已被占满
    OSIntQOvfCtr++; (17) // 中断队列溢出数目加 1
} (下页继续)
```c
*p_err = OS_ERR_INT_Q_FULL; //返回错误类型为“中断队列已满”
}
CPU_CRITICAL_EXIT();  //开中断
```

- **代码清单: 中断管理-5 (1):** 内核对象类型。
- **代码清单: 中断管理-5 (2):** 被发布的内核对象。
- **代码清单: 中断管理-5 (3):** 消息队列或任务消息。
- **代码清单: 中断管理-5 (4):** 消息的数目、大小。
- **代码清单: 中断管理-5 (5):** 事件。
- **代码清单: 中断管理-5 (6):** 发布内核对象时的选项。
- **代码清单: 中断管理-5 (7):** 发布内核对象时的时间戳。
- **代码清单: 中断管理-5 (8):** 返回错误类型。
- **代码清单: 中断管理-5 (9):** 如果启用（默认禁用）了安全检测，在编译时则会包含安全检测相关的代码，如果错误类型实参为空，系统会执行安全检测异常函数，然后返回，停止执行。
- **代码清单: 中断管理-5 (10):** 如果中断队列未占满，则执行 (10)–(20) 操作。
- **代码清单: 中断管理-5 (11):** OSIntQNbrEntries 用于记录中断队列的入队数量，需要加一表示当前有信息记录块入队。
- **代码清单: 中断管理-5 (12):** 更新中断队列的最大使用数目的历史记录。
- **代码清单: 中断管理-5 (13)–(14):** 将要重新提交的内核对象的信息放入到中断队列的信息记录块中，记录的信息有发布的对象类型、发布的内核对象、要发布的消息、要发布的消息大小、要发布的事件、选项、时间戳等信息。
- **代码清单: 中断管理-5 (15):** 指向下一个中断队列入口。
• 代码清单: 中断管理-5 (16): 让中断队列管理任务 OISIntQTask 就绪，更新就绪列表上的优先级 0 的任务数为 1 个。

• 代码清单: 中断管理-5 (17): 就绪列表的头尾指针都指向 OISIntQTask 任务。

• 代码清单: 中断管理-5 (18): 调用 OS_PrioInsert() 函数在优先级列表中增加优先级 0。

• 代码清单: 中断管理-5 (19): 如果当前运行的不是 OISIntQTask 任务，则需要保存当前任务的优先级。

• 代码清单: 中断管理-5 (20): 程序能执行到这里，表示已经正确执行完毕，返回错误类型为“无错误”的错误代码。

• 代码清单: 中断管理-5 (21): 如果中断队列已占满，记录一下中断队列溢出数目，返回错误类型为“中断队列已满”的错误代码。

30.6.5 中断延迟发布任务 OS_IntQTask()

在中断中将消息放入中断队列，那么接下来又怎么样进行发布内核对象呢？原来 μC/OS 在中断中只是将要提交的内核对象的信息都暂时保存起来，然后就绪优先级最高的中断延迟发布任务，接着继续执行中断，在退出所有中断嵌套后，第一个执行的任务就是延迟发布任务，延迟发布任务源码具体见。

列表 6: 代码清单: 中断管理-6 延迟发布任务 OS_IntQTask() 源码

```c
void OS_IntQTask (void *p_arg)
{
    CPU_BOOLEAN done;
    CPU_TS ts_start;
    CPU_TS ts_end;
    CPU_SR_ALLOC (); // 使用到临界段（在关 CPU 中断时）时必须用到该宏，该宏声明和
    // 定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
    // SR (临界段关中断只需保存 SR), 开中断时将该值还原。
```

(下页继续)
p_arg = p_arg;
while (DEF_ON)
{
    done = DEF_FALSE;
    while (done == DEF_FALSE)
    {
        CPU_CRITICAL_ENTER();
        if (OSIntQNbrEntries == (OS_OBJ_QTY)0u) // (1)
        {
            //如果中断队列里的内核对象发布完毕
            //从就绪列表移除中断队列管理任务 OS_IntQTask
            OSRdyList[0].NbrEntries = (OS_OBJ_QTY)0u;
            OSRdyList[0].HeadPtr = (OS_TCB *)0;
            OSRdyList[0].TailPtr = (OS_TCB *)0;
            OS_PrioRemove(0u); // (2) //从优先级表格移除优先级 0
            CPU_CRITICAL_EXIT(); //开中断
            OSSched(); // (3) //任务调度
            done = DEF_TRUE; //退出循环
        }
    }
}
else
{
    CPU_CRITICAL_EXIT(); //开中断
    ts_start = OS_TS_GET(); //获取时间戳
    OS_IntQRePost(); // (4) //发布中断队列里的内核对象
    ts_end = OS_TS_GET() - ts_start; //计算该次发布时间
    if (OSIntQTaskTimeMax < ts_end)
    //更新中断队列发布内核对象的最大时间的历史记录
    {
        OSIntQTaskTimeMax = ts_end;
    }
}
void OS_IntQRePost (void)
{
    CPU_TS ts;
    OS_ERR err;
}

• 代码清单: 中断管理-6 (1): 如果中断队列里的内核对象发布完毕（OSIntQNbrEntries 变量
    的值为 0），从就绪列表移除中断延迟发布任务 OS_InitTask，这样子的操作相当于挂起
    OS_InitTask 任务。

• 代码清单: 中断管理-6 (2): 从优先级表格中移除优先级 0 的任务。

• 代码清单: 中断管理-6 (3): 进行一次任务调度，这就保证了从中断出来后如果需要发布会
    将相应的内核对象全部进行发布直到全部都发布完成，才会进行一次任务调度，然后让其
    他的任务占用 CPU。

• 代码清单: 中断管理-6 (4): 如果中断队列里还存在未发布的内核对象，就调用
    OS_IntQRePost() 函数发布中断队列里的内核对象，其实这个函数才是真正的发布操作，该
    函数源码具体见 代码清单: 中断管理-7。

• 代码清单: 中断管理-6 (5): 处理下一个要发布的内核对象，直到没有任何要发布的内核对
    象为止。

• 代码清单: 中断管理-6 (6): 中断队列的成员数目减 1。

列表 7: 代码清单: 中断管理-7 OS_IntQRePost() 源码

社区: https://www.firebbs.cn/ 762 天猫: https://yehuosm.tmall.com
switch (OSIntQOutPtr->Type) {  
// 根据内核对象类型分类处理
{
    case OS_OBJ_TYPE_FLAG:  
        // 如果对象类型是事件标志
        #if OS_CFG_FLAG_EN > 0u  
        (void)OS_FlagPost((OS_FLAG_GRP *) OSIntQOutPtr->ObjPtr,  
            (OS_FLAGS ) OSIntQOutPtr->Flags,  
            (OS_OPT ) OSIntQOutPtr->Opt,  
            (CPU_TS ) OSIntQOutPtr->TS,  
            (OS_ERR *) &err);  
        #endif
    break;  
// 跳出

    case OS_OBJ_TYPE_Q:  
        // 如果对象类型是消息队列
        #if OS_CFG_Q_EN > 0u  
        OS_QPost((OS_Q *) OSIntQOutPtr->ObjPtr,  
            (void ) OSIntQOutPtr->MsgPtr,  
            (OS_MSG_SIZE) OSIntQOutPtr->MsgSize,  
            (OS_OPT ) OSIntQOutPtr->Opt,  
            (CPU_TS ) OSIntQOutPtr->TS,  
            (OS_ERR *) &err);  
        #endif
    break;  
// 跳出

    case OS_OBJ_TYPE_SEM:  
        // 如果对象类型是信号量
        #if OS_CFG_SEM_EN > 0u  
        (void)OS_SemPost((OS_SEM *) OSIntQOutPtr->ObjPtr,  
            (OS_OPT ) OSIntQOutPtr->Opt,  
            (CPU_TS ) OSIntQOutPtr->TS,  
            (OS_ERR *) &err);  
        #endif
    break;  
// 跳出

    #endif

}
break; //跳出

case OS_OBJ_TYPE_TASK_MSG: //如果对象类型是任务消息
#if OS_CFG_TASK_Q_EN > 0u //如果启用了任务消息，则发布任务消息
  OS_TaskQPost((OS_TCB *) OSIntQOutPtr->ObjPtr,
                (void *) OSIntQOutPtr->MsgPtr,
                (OS_MSG_SIZE) OSIntQOutPtr->MsgSize,
                (OS_OPT ) OSIntQOutPtr->Opt,
                (CPU_TS ) OSIntQOutPtr->TS,
                (OS_ERR *) &err); (5)
#endif
break; //跳出

case OS_OBJ_TYPE_TASK_RESUME: //如果对象类型是恢复任务
#if OS_CFG_TASK_SUSPEND_EN > 0u //如果启用了函数OSTaskResume()，恢复该任务
  (void)OS_TaskResume((OS_TCB *) OSIntQOutPtr->ObjPtr,
                      (OS_ERR *) &err); (6)
#endif
break; //跳出

case OS_OBJ_TYPE_TASK_SIGNAL: //如果对象类型是任务信号量
  (void)OS_TaskSemPost((OS_TCB *) OSIntQOutPtr->ObjPtr, //发布任务信号量
                       (OS_OPT ) OSIntQOutPtr->Opt,
                       (CPU_TS ) OSIntQOutPtr->TS,
                       (OS_ERR *) &err); (7)
break; //跳出

case OS_OBJ_TYPE_TASK_SUSPEND: //如果对象类型是挂起任务
#if OS_CFG_TASK_SUSPEND_EN > 0u //如果启用了函数OSTaskSuspend()，挂起该任务
  (void)OS_TaskSuspend((OS_TCB *) OSIntQOutPtr->ObjPtr,
                       (OS_ERR *) &err); (8)
#endif
break;           //跳出

case OS_OBJ_TYPE_TICK:        (9) //如果对象类型是时钟节拍
  if OS_CFG_SCHED_ROUND_ROBIN_EN > 0u //如果启用了时间片轮转调度，
    OS_SchedRoundRobin(&OSRdyList[OSPrioSaved]); //轮转调度进中断前优先级任务
  #endif

    (void)OS_TaskSemPost((OS_TCB *)&OSTickTaskTCB, //发送信号量给时钟节拍任务
      (OS_OPT ) OS_OPT_POST_NONE,
      (CPU_TS ) OSIntQOutPtr->TS,
      (OS_ERR *)&err); (10)
  #if OS_CFG_TMR_EN > 0u
    //如果启用了软件定时器，发送信号量给定时器任务
    OSTmrUpdateCtr--;
    if (OSTmrUpdateCtr == (OS_CTR)0u)
      {
        OSTmrUpdateCtr = OSTmrUpdateCnt;
        ts               = OS_TS_GET();
        (void)OS_TaskSemPost((OS_TCB *)&OSTmrTaskTCB,    //发送信号量给软件定时器
          (OS_OPT ) OS_OPT_POST_NONE,
          (CPU_TS ) ts,
          (OS_ERR *)&err); (11)
      }
  #endif
  break;           //跳出

default:          (12)//如果内核对象类型超出预期
    break;          //直接跳出
  }
}
• 代码清单: 中断管理-7 (1): 根据内核对象类型分类处理。
• 代码清单: 中断管理-7 (2): 如果对象类型是事件标志，发布事件标志。
• 代码清单: 中断管理-7 (3): 如果对象类型是消息队列，发布消息队列。
• 代码清单: 中断管理-7 (4): 如果对象类型是信号量，发布信号量。
• 代码清单: 中断管理-7 (5): 如果对象类型是任务消息，发布任务消息。
• 代码清单: 中断管理-7 (6): 如果对象类型是恢复任务，恢复该任务。
• 代码清单: 中断管理-7 (7): 如果对象类型是任务信号量，发布任务信号量。
• 代码清单: 中断管理-7 (8): 如果对象类型是挂起任务，挂起该任务。
• 代码清单: 中断管理-7 (9): 如果对象类型是时钟节拍，如果启用了时间片轮转调度，轮转调度进中断前优先级任务。
• 代码清单: 中断管理-7 (10): 发送信号量给时钟节拍任务。
• 代码清单: 中断管理-7 (11): 如果启用了软件定时器，发送信号量给定时器任务。
• 代码清单: 中断管理-7 (12): 如果内核对象类型超出预期，直接跳出。

该函数的整个流程也是非常简单的，首先提取出中断队列中的一个信息块的信息，根据发布的内核对象类型分类处理，在前面我们已经讲解过了全部内核对象发布（释放）的过程，就直接在任务中调用这些发布函数根据对应的内核对象进行发布。值得注意的是时钟节拍类型 OS_OBJ_TYPE_TICK，如果没有启用中断延迟发布的宏定义，那么所有跟时钟节拍相关的，包括时间片轮转调度，定时器，发送消息给时钟节拍任务等都是在中断中执行，而使用延迟提交就把这些工作都放到延迟发布任务中执行。延迟发布之所以能够减少关中断的时间是因为在这些内核对象发布函数中，进入临界段都是采用锁调度器的方式，如果没有使用延迟发布，提交的整个过程都要关中断。

至此，中断延迟发布的内容就讲解完毕，无论是否选择中断延迟发布，都不需要我们修改用户代码，这个是 μC/OS 会根据我们的选择自动处理，无需我们用户理会。
### 30.7 中断管理实验

中断管理实验是在 μC/OS 中创建了两个任务分别获取信号量与消息队列，并且定义了两个按键 KEY1 与 KEY2 的触发方式为中断触发。其触发的中断服务函数则跟裸机一样，在中断触发的时候通过消息队列将消息传递给任务。任务接收到消息即开启串口调试助手显示出来。而且中断管理实验也实现了一个串口的 DMA 传输 + 空闲中断功能，当串口接收完不定长的数据之后产生一个空闲中断，在中断中将信号量传递给任务，任务在收到信号量的时候将串口的数据读取出来并且在串口调试助手中回显。具体见代码清单: 中断管理-8 加粗部分。

#### 代码清单: 中断管理-8 中断管理实验

```c
#include <includes.h>
#include <string.h>

static OS_TCB AppTaskStartTCB; // 任务控制块
static OS_TCB AppTaskUsartTCB;
static OS_TCB AppTaskKeyTCB;

static CPU_STK AppTaskStartStk[APP_TASK_START_STK_SIZE]; // 任务栈
static CPU_STK AppTaskUsartStk [APP_TASK_USART_STK_SIZE];
static CPU_STK AppTaskKeyStk [APP_TASK_KEY_STK_SIZE];

extern char Usart_Rx_Buf[USART_RBUFF_SIZE];

static void AppTaskStart (void *p_arg); // 任务函数声明
static void AppTaskUsart (void *p_arg);
static void AppTaskKey (void *p_arg);

int main (void)
{
    OS_ERR err;
    OSInit(&err);
}
```

论坛: [https://www.firebbs.cn/](https://www.firebbs.cn/)
//初始化µC/OS-III

/* 创建起始任务 */
OSTaskCreate((OS_TCB *) &AppTaskStartTCB,
    //任务控制块地址
    (CPU_CHAR *) "App Task Start",
    //任务名称
    (OS_TASK_PTR) AppTaskStart,
    //任务函数
    (void *) 0,
    //传递给任务函数（形参 p_arg）的实参
    (OS_PRIO) APP_TASK_START_PRIO,
    //任务的优先级
    (CPU_STK *) &AppTaskStartStk[0],
    //任务栈的基地址
    (CPU_STK_SIZE) APP_TASK_START_STK_SIZE / 10,
    //任务栈空间剩下1/10时限制其增长
    (CPU_STK_SIZE) APP_TASK_START_STK_SIZE,
    //任务栈空间（单位: sizeof(CPU_STK)）
    (OS_MSG_QTY) 5u,
    //任务可接收的最大消息数
    (OS_TICK) 0u,
    //任务的时间片节拍数（0 表默认值 OSCfg_TickRate_Hz/10）
    (void *) 0,
    //任务扩展（0 表不扩展）
    (OS_OPT) (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
    //任务选项
    (OS_ERR *) &err);

OSStart(&err);
//启动多任务管理（交由µC/OS-III控制）
static void AppTaskStart (void *p_arg)
{
    CPU_INT32U cpu_clk_freq;
    CPU_INT32U cnts;
    OS_ERR err;

    (void)p_arg;
    // 板级初始化
    BSP_Init();
    // 初始化 CPU 组件（时间戳、关中断时间测量和主机名）
    CPU_Init();

    // 获取 CPU 内核时钟频率 (SysTick 工作时钟)
    cpu_clk_freq = BSP_CPU_ClkFreq();
    // 根据用户设定的时钟节拍频率计算 SysTick 定时器的计数值
    cnts = cpu_clk_freq / (CPU_INT32U)OSCfg_TickRate_Hz;
    // 调用 SysTick 初始化函数，设置定时器计数值和启动定时器
    OS_CPU_SysTickInit(cnts);
    // 初始化内存管理组件 (堆内存池和内存池表)
    Mem_Init();

    // 如果启用 (默认启用) 了统计任务
#if OS_CFG_STAT_TASK_EN > 0u
    OSStatTaskCPUUsageInit(&err);
#endif

    // 复位 (清零) 当前最大关中断时间
    CPU_IntDisMeasMaxCurReset();
}
/* 配置时间片轮转调度 */
OSSchedRoundRobinCfg((CPU_BOOLEAN )DEF_ENABLED,
    //启用时间片轮转调度
    (OS_TICK )0,
    //把 OSCfg_TickRate_Hz/10 设为默认时间片值
    (OS_ERR *)&err );  //返回错误类型

/* 创建 AppTaskUsart 任务 */
OSTaskCreate((OS_TCB *)&AppTaskUsartTCB,
    //任务控制块地址
    (CPU_CHAR *)"App Task Usart",
    //任务名称
    (OS_TASK_PTR )AppTaskUsart,
    //任务函数
    (void *)0,
    //传递给任务函数（形参 p_arg）的实参
    (OS_PRIO )APP_TASK_USART_PRIO,
    //任务的优先级
    (CPU_STK *)&AppTaskUsartStk[0],
    //任务栈的基地址
    (CPU_STK_SIZE) APP_TASK_USART_STK_SIZE / 10,
    //任务栈空间剩下 1/10 时限制其增长
    (CPU_STK_SIZE) APP_TASK_USART_STK_SIZE,
    //任务栈空间（单位：sizeof(CPU_STK)）
    (OS_MSG_QTY ) 50u,
    //任务可接收的最大消息数
    (OS_TICK )0u,
    //任务的时间片节拍数（0 表默认值 OSCfg_TickRate_Hz/10）
    (void *)0,
    //任务扩展（0 表不扩展）
    (OS_OPT ) (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
    // (下页继续)
// 任务选项
(OS_ERR *) &err);
// 返回错误类型

/* 创建 AppTaskKey 任务 */
OSTaskCreate((OS_TCB *) &AppTaskKeyTCB,
    // 任务控制块地址
    (CPU_CHAR *) "App Task Key",
    // 任务名称
    (OS_TASK_PTR) AppTaskKey,
    // 任务函数
    (void *) 0,
    // 传递给任务函数 (形参 p_arg) 的实参
    (OS_PRIO) APP_TASK_KEY_PRIO,
    // 任务的优先级
    (CPU_STK *) &AppTaskKeyStk[0],
    // 任务栈的基地址
    (CPU_STK_SIZE) APP_TASK_KEY_STK_SIZE / 10,
    // 任务栈空间剩下 1/10 时限制其增长
    (CPU_STK_SIZE) APP_TASK_KEY_STK_SIZE,
    // 任务栈空间 (单位: sizeof(CPU_STK))
    (OS_MSG_QTY) 50U,
    // 任务可接收的最大消息数
    (OS_TICK) 0,
    // 任务的时间片节拍数 (0 表默认值 OSCfg_TickRate_Hz/10)
    (void *) 0,
    // 任务扩展 (0 表不扩展)
    (OS_OPT) (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
    // 任务选项
    (OS_ERR *) &err);
    // 返回错误类型
OSTaskDel (0, &err);
// 删除起始任务本身，该任务不再运行
}

static void AppTaskUsart ( void * p_arg )
{
    OS_ERR err;
    CPU_SR_ALLOC();
    (void)p_arg;

    while (DEF_TRUE) // 任务体
    {

        OSTaskSemPend ((OS_TICK )0,
                        (OS_OPT )OS_OPT_PEND_BLOCKING,
                        (CPU_TS *)0,
                        (OS_ERR *)&err); // 返回错误类型

        OS_CRITICAL_ENTER();
        // 进入临界段，避免串口打印被打断
        printf("收到数据:%s\n",Usart_Rx_Buf);
        memset(Usart_Rx_Buf,0,USART_RBUFF_SIZE); // 清零
        OS_CRITICAL_EXIT(); // 退出临界段
    }
}

static void AppTaskKey ( void * p_arg )
{
    OS_ERR err;
    CPU_TS_TMR ts_int;
    CPU_INT32U cpu_clk_freq;
CPU_SR_ALLOC();

(void)p_arg;

cpu_clk_freq = BSP_CPU_ClkFreq();
//获取CPU时钟，时间戳是以该时钟计数

while (DEF_TRUE) //任务体
{
    /* 阻塞任务，直到KEY1被按下 */
    OSTaskSemPend ((OS_TICK )0,
    (OS_OPT )OS_OPT_PEND_BLOCKING,
    //如果信号量不可用就等待
    (CPU_TS  *)&0,
    //获取信号量被发布的时间戳
    (OS_ERR  *)&err);          //返回错误类型

    ts_int = CPU_IntDisMeasMaxGet ();  //获取最大关中断时间

    OS_CRITICAL_ENTER();
    //进入临界段，避免串口打印被打断

    printf ( "触发按键中断，最大中断时间是%ds\r\n", ts_int / ( cpu_clk_freq_.
    /* 1000000 */ ));

    OS_CRITICAL_EXIT();          //退出临界段
}

而中断服务函数则需要我们自己编写，并且中断被触发的时候通过信号量告知任务，具体见代码清单：中断管理-9。

论坛：https://www.firebbs.cn/        天猫：https://yehuosm.tmall.com
列表 9: 代码清单: 中断管理-9 中断管理——中断服务函数

```c
#include "stm32f10x_it.h"
#include <includes.h>
#include "bsp_usart1.h"
#include "bsp_exti.h"

extern OS_TCB AppTaskUsartTCB;
extern OS_TCB AppTaskKeyTCB;

/**
 * @brief USART 中断服务函数
 * @param
 * @retval
 */
void macUSART_INT_FUN(void)
{
    OS_ERR err;
    OSIntEnter(); // 进入中断

    if ( USART_GetITStatus ( macUSARTx, USART_IT_IDLE ) != RESET )
    {
        DMA_Cmd(USART_RX_DMA_CHANNEL, DISABLE);
        USART_ReceiveData ( macUSARTx ); /* 清除标志位 */
        // 清 DMA 标志位
        DMA_ClearFlag( DMA1_FLAG_TC5 );
    }
}
```
// 重新赋值计数值，必须大于等于最大可能接收到的数据帧数目
USART_RX_DMA_CHANNEL->CNDTR = USART_RBUFSIZE;
DMA_Cmd(USART_RX_DMA_CHANNEL, ENABLE);

// 给出信号量，发送接收到新数据标志，供前台程序查询

/* 发送任务信号量到任务 AppTaskKey */
OSTaskSemPost((OS_TCB *)&AppTaskUsartTCB, // 目标任务
(OS_OPT )OS_OPT_POST_NONE, // 没选项要求
(OS_ERR *)&err); // 返回错误类型
}

OSIntExit(); // 退出中断
}

/**
 * @brief EXTI 中断服务函数
 * @param 无
 * @retval 无
 */
void macEXTI_INT_FUNCTION (void)
{
    OS_ERR  err;
    OSIntEnter(); // 进入中断

    if (EXTI_GetITStatus(macEXTI_LINE) != RESET) // 确保是否产生了 EXTI Line 中断
    {
        /* 发送任务信号量到任务 AppTaskKey */
    }
}
OSTaskSemPost((OS_SVC_tcb*) &AppTaskKeyTCB, // 目标任务
(OS_OPT ) OS_OPT_POST_NONE, // 没选项要求
(OS_ERR *) &err); // 返回错误类型

EXTI_ClearITPendingBit(macEXTI_LINE); // 清除中断标志位

)

OSIntExit(); // 退出中断

}

30.8 中断管理实验现象

程序编译好，用 USB 线连接计算机和开发板的 USB 接口（对应丝印为 USB 转串口），用 DAP 仿真器把配套程序下载到野火 STM32 开发板（具体型号根据购买的板子而定，每个型号的板子都配套有对应的程序），在计算机上打开串口调试助手，然后复位开发板就可以在调试助手中看到串口的打印信息，按下开发板的 KEY1 按键触发中断，在串口调试助手中可以看到运行结果，然后通过串口调试助手发送一段不定长信息，触发中断会在中断服务函数发送信号量通知任务，任务接收到信号量的时候将串口信息打印出来，具体见图中断管理的实验现象。
第 31 章 CPU 利用率及栈检测统计

31.1 CPU 利用率的基本概念及作用

CPU 利用率其实就是系统运行的程序占用的 CPU 资源，表示机器在某段时间程序运行的情况，如果这段时间中，程序一直在占用 CPU 的使用权，那么可以认为 CPU 的利用率是 100%。CPU 的利用率越高，说明机器在这个时间上运行了很多程序，反之较少。利用率的高低与 CPU 性能强弱有直接关系，就像一段一模一样的程序，如果使用运算速度很慢的 CPU，它可能要运行 1000ms,而使用很快运算速度很快的 CPU 可能只需要 10ms,那么在 1000ms 这段时间中，前者的 CPU 利用率就是100%，而后者的 CPU 利用率只有 1%，因为 1000ms 内前者都在使用 CPU 做运算，而后者只使用 10ms 的时间做运算。剩下的时间 CPU 可以做其他事情。

μC/OS 是多任务操作系统，对 CPU 都是分时使用的：比如 A 任务占用 10ms，然后 B 任务占用 30ms，然后空闲 60ms，再又是 A 任务占 10ms，B 任务占 30ms，空闲 60ms；如果在一段时间内都是如此，那么这段时间内的利用率为 40%，因为整个系统中只有 40% 的时间是 CPU 处理数据的时间。

一个系统设计的好坏，可以使用 CPU 利用率来衡量，一个好的系统必然是能完美响应急需的处理，并且系统的资源不会过于浪费（性价比高）。举个例子，假设一个系统的 CPU 利用率经常在 90%~100% 徘徊，那么系统就很少有空闲的时候，这时候突然有些事情急需 CPU 的处理，但是此时 CPU 都很可能被其他任务在占用了，那么这个紧急事件就有可能无法被相应，即启用被相应，那么占用 CPU 的任务又处于等待状态，这种系统就是不够完美的，因为它资源处理得过于紧迫。反过来，假如 CPU 利用率在 1% 以下，那么我们就可以认为这种产品的资源过于浪费，搞一个那么好的 CPU 去干着没啥意义的活（大部分时间处于空闲状态），作为产品的设计，既不能让资源过于浪费，也不能让资源过于紧迫，这种设计才是完美的，在需要的时候能及时处理完突发事件，而且资源也不会过剩，性价比更高。

μC/OS 提供的 CPU 利用率统计是一个可选功能，只有将 OS_CFG_STAT_TASK_EN 宏定义启用后用户才能使用 CPU 利用率统计相关函数，该宏定义位于 os_cfg.h 文件中。
31.2 CPU 利用率统计初始化

μC/OS 对 CPU 利用率进行统计是怎怎么实现的呢？简单来说，CPU 利用率统计的原理很简单，我们知道，系统中必须存在空闲任务，当且仅当 CPU 空闲的时候才会去执行空闲任务，那么我们就可以让 CPU 在空闲任务中一直做加法运算，假设某段时间 T 中 CPU 一直都在空闲任务中做加法运算（变量自加），那么这段时间算出来的值就是 CPU 空闲时候的最大值，我们假设为 100，那么当系统中有其他任务的时候，CPU 就不可能一直处于空闲任务做加法了，那么同样的一段时间 T 里，空闲任务算出来的值变成了 80，那么是不是可以说明空闲任务只占用了系统的 80% 的资源，剩下的 20% 被其他任务占用了，这是显而易见的，同样的，利用这个原理，我们就能知道 CPU 的利用率大约是多少了（这种计算不会很精确），假设 CPU 在 T 时间内空闲任务中运算的最大值为 OSStatTaskCtrMax（100），而有其他任务参与时 T 时间内空闲任务运算的值为 80（OSStatTaskCtr），那么 CPU 的利用率 CPUUsage 的公式应该为：CPUUsage（%） = 100 *（1 - OSStatTaskCtr / OSStatTaskCtrMax），假设没有一次空闲任务运算的值为 100（OSStatTaskCtr），说明没有其他任务参与，那么 CPU 的利用率就是 0%，如果 OSStatTaskCtr 的值为 0，那么表示这段时间里 CPU 都没在空闲任务中运算，那么 CPU 的利用率自然就是 100%。

注意：一般情况下时 T 由 OS_CFG_STAT_TASK_RATE_HZ 宏定义决定，是我们自己在 os_cfg_app.h 文件中定义的，我们的例程定义为 10，该宏定义决定了统计任务的执行频率，即决定了更新一次 CPU 利用率的时间为 1/OS_CFG_STAT_TASK_RATE_HZ，单位是秒。此外，统计任务的时钟节拍与软件定时器任务的时钟节拍一样的，都是由系统时钟节拍分频得到的，如果统计任务运行的频率设定并不是时钟节拍数倍，那么统计任务实际运行的频率跟设定的就会有误差，这点跟定时器是一样的。

在统计 CPU 利用率之前必须先调用 OSStatTaskCPUUsageInit() 函数进行相关初始化，这个函数的目的就是为了计算只有空闲任务时 CPU 在某段时间内的运算最大值，也就是 OSStatTaskCtrMax，其源码具体见代码清单: 利用率-I 。

列表 1: 代码清单: 利用率-I OSStatTaskCPUUsageInit() 源代码

```c
void OSStatTaskCPUUsageInit (OS_ERR *p_err)
{
```

（下页继续）
OS_ERR err;
OS_TICK dly;
CPU_SR_ALLOC(); // 使用到临界段 (在关/开中断时) 时必须用到该宏，该宏声明和
// 定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
// SR (临界段关中断只需保存 SR)，开中断时将该值还原。

#define OS_SAFETY_CRITICAL // 如果启用了安全检测
if (p_err == (OS_ERR *)0) // 如果 p_err 为空
{
    OS_SAFETY_CRITICAL_EXCEPTION(); // 执行安全检测异常函数
    return;
}
#endif

#if (OS_CFG_TMR_EN > 0u) // 如果启用了软件定时器
OSTaskSuspend(&OSTmrTaskTCB, &err); (1) // 挂起软件定时任务
if (err != OS_ERR_NONE) // 如果挂起失败
{
    *p_err = err; // 返回失败原因
    return; // 返回，停止执行
}
#endif

OSTimeDly((OS_TICK )2,
    // 先延时两个节拍，为后面延时同步时钟节拍，增加准确性
    (OS_OPT )OS_OPT_TIME_DLY,
    (OS_ERR *)&err); (2)
if (err != OS_ERR_NONE) // 如果延时失败
{
    *p_err = err; // 返回失败原因
    return; // 返回，停止执行
}
CPU_CRITICAL_ENTER();  // 关中断
OSStatTaskCtr = (OS_TICK)0;  // 清零空闲计数器
CPU_CRITICAL_EXIT();  // 开中断

/* 根据设置的宏计算统计任务的执行节拍数 */
dly = (OS_TICK)0;  (3)
if (OSCfg_TickRate_Hz > OSCfg_StatTaskRate_Hz)
{
    dly = (OS_TICK)(OSCfg_TickRate_Hz / OSCfg_StatTaskRate_Hz);
}
if (dly == (OS_TICK)0)
{
    dly = (OS_TICK)(OSCfg_TickRate_Hz / (OS_RATE_HZ)10);
}  (4)

/* 延时累加空闲计数，获取最大空闲计数值 */
OSTimeDly(dly,
    OS_OPT_TIME_DLY,
    &err);  (5)

#if (OS_CFG_TMR_EN > 0u)  // 如果启用了软件定时器
OSTaskResume(&OSTmrTaskTCB, &err);  (5)  // 恢复软件定时器任务
if (err != OS_ERR_NONE)  // 如果恢复失败
{
    *p_err = err;  // 返回错误原因
    return;  // 返回，停止执行
}  (6)
#endif

/* 如果上面没产生错误 */
CPU_CRITICAL_ENTER();  // 关中断
OSStatTaskTimeMax = (CPU_TS)0;  // 存储最大空闲计数值
OSStatTaskCtrMax = OSStatTaskCtr;  (6)
OSStatTaskRdy = OS_STATE_RDY;  (7)  // 准备就绪统计任务

/* 论坛：https://www.firebbs.cn/  天猫：https://yehuosm.tmall.com */
CPU_CRITICAL_EXIT();  //开中断
*p_err = OS_ERR_NONE;  //错误类型为“无错误”
}

- **代码清单: 利用率-1 (1)**: 如果启用了软件定时器，那么在系统初始化的时候就会创建软件定时器任务，此处不希望别的任务打扰空闲任务的运算，就暂时将软件定时器任务挂起。

- **代码清单: 利用率-1 (2)**: 先延时两个节拍，为后面延时同步时钟节拍，增加准确性，为什么要先延时两个节拍呢？因为是为了匹配后面一个延时的时间起点，当两个时钟节拍到达后，再继续延时 dly 个时钟节拍，这样子时间就比较精确。程序执行到这里的时候，我们并不知道时间过去了多少，所以此时的延时起点并不一定与系统的时钟节拍匹配，具体见图延时误差分析。

- **代码清单: 利用率-1 (3)**: 根据设置的宏计算统计任务的执行节拍数，也就是 T 时间。

- **代码清单: 利用率-1 (4)**: 延时 dly 个时钟节拍（这个时钟节拍的延时会比较准确），将当前任务阻塞，让空闲做累加运算，获取最大空闲运算数值 OSStatTaskCtrMax。

- **代码清单: 利用率-1 (5)**: 恢复软件定时器任务。

- **代码清单: 利用率-1 (6)**: 保存一下空闲任务最大的运算数值 OSStatTaskCtrMax

- **代码清单: 利用率-1 (7)**: 准备就绪统计任务。

注意，调用 OSStatTaskCPUUsageInit() 函数进行初始化的时候，一定要在创建用户任务之前，否则当系统有很多任务在调度的时候，空闲任务就没法在某段时间内完成运算并且得到准确的 OSStatTaskCtrMax，这样子的 CPU 利用率计算是不准确的。

注意：统计的过程在后文讲解。

论坛: https://www.firebbs.cn/  782  天猫: https://yehuosm.tmall.com


31.3 栈溢出检测概念及作用

如果处理器有 MMU 或者 MPU，检测栈是否溢出是非常简单的，MMU 和 MPU 是处理器上特殊的硬件设施，可以检测非法访问，如果任务企图访问未被允许的内存空间的话，就会产生警告。但是我们使用的 STM32 是没有 MMU 和 MPU 的，但是可以使用软件模拟栈检测，但是软件的模拟比较难以实现。但是 μC/OS 为我们提供了栈使用情况统计的功能，直接使用即可。如果需要使用栈溢出检测的功能，就需要用户自己在 App_OS_TaskSwHook() 钩子函数中自定义实现（我们不实现该功能），需要使用 μC/OS 为我们提供的栈检测功能，想要使用该功能就需要在 os_cfg_app.h 文件中将 OS_CFG_STAT_TASK_STK_CHK_EN 宏定义配置为 1。

某些处理器中有一些栈溢出检测相关的寄存器，当 CPU 的栈指针小于（或大于，取决于栈的生长方向）设置于这个寄存器的值时，就会产生一个异常（中断），异常处理程序就需要确保未允许访问空间代码的安全（可能会发送警告给用户，或者其他处理）。任务控制块中的成员变量 StkLimitPtr 就是为这种目的而设置的，如图栈溢出检测 _ 硬件 所示。每个任务的栈必须分配足够大的内存空间供任务使用，在大多数情况下，StkLimitPtr 指针的值可以设置接近于栈顶（&TaskStk[0]，假定栈是从高地址往低地址生长的，事实上 STM32 的栈生长方向就是向下生长，从高地址向低地址生长），StkLimitPtr 的值在创建任务的时候由用户指定。

论坛：https://www.firebbs.cn/  783  天猫：https://yehuosm.tmall.com
注意：此处的栈检测是对于带有 MPU 的处理器。

那么 μC/OS 中对于没有 MPU 的处理器是怎么做到栈检测的呢？

当 μC/OS 从一个任务切换到另一个任务的时候，它会调用一个钩子函数 OSTaskSwHook()，它允许用户扩展上下文切换时的功能。所以，如果处理器没有硬件支持溢出检测功能，就可以在该钩子函数中添加代码软件模拟该功能。在切换到任务 B 前，我们需要检测将要被载入 CPU 栈指针的值是否超过该任务 B 的任务控制块 StkLimitPtr 的限制。因为软件不能在溢出时就迅速地做出反应，所以应该设置 StkLimitPtr 的值尽可能远离栈顶，保证有足够的溢出缓冲，具体见栈溢出检测 _ 软件。软件检测不会像硬件检测那样有效，但也可以有效防止栈溢出。
31.4 栈溢出检测过程

在前面的章节中我们已经详细讲解了栈相关的知识，每个任务独立的栈空间对任务来说是至关重要的，栈空间中保存了任务运行过程中需要保存局部变量、寄存器等重要的信息，如果设置的栈太小，任务无法正运行，可能还会出现各种奇怪的错误，如果发现我们程序出现奇怪的错误，一定要检查栈空间，包括 MSP 的栈，系统任务的栈，用户任务的栈。

μC/OS 是怎么检测任务使用了多少栈的呢？以 STM32 的栈生长方向为例子（高地址向低地址生长），在任务初始化的时候先将任务所有的栈都置 0，使用后的栈不为 0，在检测的时候只需从栈的低地址开始对为 0 的栈空间进行计数统计，然后通过计算就可以得出任务的栈使用了多少，这
样子用户就可以根据实际情况进行调整任务栈的大小，具体见图栈检测示意图，这些信息同样也会在统计任务每隔 I/OSCfg_StatTaskRate_Hz 秒就进行更新。

31.5 统计任务 OS_StatTask()

μC/OS 提供了统计任务的函数，该函数为系统内部函数（任务），在启用宏定义 OS_CFG_STAT_TASK_EN 后，系统会自动创建一个统计任务——OS_StatTask()，它会在任务中计算整个系统的 CPU 利用率，各个任务的 CPU 利用率和各个任务的栈使用信息，其源码具体见代码清单：利用率-2。

论坛：https://www.firebbs.cn/  
天猫：https://yehuosm.tmall.com
void OS_StatTask (void *p_arg) //统计任务函数
{
    
    #if OS_CFG_DBG_EN > 0u
    
    #if OS_CFG_TASK_PROFILE_EN > 0u
        OS_CPU_USAGE usage;
        OS_CYCLES cycles_total;
        OS_CYCLES cycles_div;
        OS_CYCLES cycles_mult;
        OS_CYCLES cycles_max;
    #endif
    
    OS_TCB *p_tcb;
    
    if OS_CFG_DBG_EN > 0u
    if OS_CFG_TASK_PROFILE_EN > 0u
        OS_CPU_USAGE usage;
        OS_CYCLES cycles_total;
        OS_CYCLES cycles_div;
        OS_CYCLES cycles_mult;
        OS_CYCLES cycles_max;
    #endif
    
    OS_TCB *p_tcb;
    
    #endif
        OS_TICK ctr_max;
        OS_TICK ctr_mult;
        OS_TICK ctr_div;
        OS_TICK dly;
        OS_TICK ts_start;
        OS_TICK ts_end;
        CPU_TS ts_start;
        CPU_TS ts_end;
        CPU_SR_ALLOC();
        
        //使用到临界段（在关/开中断时）时必须用到该宏，该宏声明和
        //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
        //SR（临界段关中断只需保存 SR），开中断时将该值还原。
    
p_arg = p_arg;
    
    //没意义，仅为预防编译器警告
    while (OSStatTaskRdy != DEF_TRUE) //如果统计任务没被允许运行
        
        { 
            OSTimeDly (2u * OSCfg_StatTaskRate_Hz, OS_OPT_TIME_DLY, //一直延时
                
                
                Forum: https://www.firebbs.cn/ 天猫: https://yehuosm.tmall.com
```c
31 }&err);
32 
33 OSStatReset(&err); //如果统计任务已被就绪，复位统计，继续执行
34 /* 根据设置的宏计算统计任务的执行节拍数 */
35 dly = (OS_TICK)0;
36 if (OScfg_TickRate_Hz > OSCfg_StatTaskRate_Hz)
37 { dly = (OS_TICK) (OScfg_TickRate_Hz / OSCfg_StatTaskRate_Hz);
38 }
39 if (dly == (OS_TICK)0)
40 { dly = (OS_TICK) (OScfg_TickRate_Hz / (OS_RATE_HZ)10);
41 } (1)
42
43 while (DEF_ON) //进入任务体
44 {
45 ts_start = OS_TS_GET(); //获取时间戳
46 #ifdef CPU_CFG_INT_DIS_MEAS_EN//如果要测量中断时间
47 OSIntDisTimeMax = CPU_IntDisMeasMaxGet(); //获取最大的中断时间
48 #endif
49
50 CPU_CRITICAL_ENTER(); //中断
51 OSStatTaskCtrRun = OSStatTaskCtr; (3) //获取上一次空闲任务的计数值
52 OSStatTaskCtr = (OS_TICK)0; //进行下一次空闲任务计数清零
53 CPU_CRITICAL_EXIT(); //开中断
54 /* 计算 CPU 利用率 */
55 if (OSStatTaskCtrMax > OSStatTaskCtrRun) (4)
56 //如果空闲计数值大于最大空闲计数值
57 {
58 if (OSStatTaskCtrMax < 400000u)
59 //这些分类是为了避免计算 CPU 利用率过程中
```

{  
    ctr_mult = 10000u;  //产生溢出，就是避免相乘时超出 32 位寄存器。
    ctr_div = 1u;
}
else if (OSStatTaskCtrMax < 4000000u)
{
    ctr_mult = 1000u;
    ctr_div = 10u;
}
else if (OSStatTaskCtrMax < 40000000u)
{
    ctr_mult = 100u;
    ctr_div = 100u;
}
else if (OSStatTaskCtrMax < 400000000u)
{
    ctr_mult = 10u;
    ctr_div = 1000u;
}
else
{
    ctr_mult = 1u;
    ctr_div = 10000u;
}
ctr_max = OSStatTaskCtrMax / ctr_div;
OSStatTaskCPUUsage = (OS_CPU_USAGE)((OS_TICK)10000u -
ctr_mult * OSStatTaskCtrRun / ctr_max); (5)
if (OSStatTaskCPUUsageMax < OSStatTaskCPUUsage)
  //更新 CPU 利用率的最大历史记录
  {
    OSStatTaskCPUUsageMax = OSStatTaskCPUUsage;
}
else //如果空闲计数值大于或等于最大空闲计数值
{
    OSStatTaskCPUUsage = (OS_CPU_USAGE)10000u; //那么 CPU 利率为 0
}

OSStatTaskHook(); //用户自定义的钩子函数

/* 下面计算各个任务的 CPU 利用率，原理跟计算整体 CPU 利用率相似 */
#else if OS_CFG_DBG_EN > 0u//如果启用了调试代码和变量
#endif
#if OS_CFG_TASK_PROFILE_EN > 0u
//如果启用了允许统计任务信息
    cycles_total = (OS_CYCLES)0;

    CPU_CRITICAL_ENTER(); //关中断
    p_tcb = OSTaskDbgListPtr;
    //获取任务双向调试列表的首个任务
    CPU_CRITICAL_EXIT(); //开中断
    while (p_tcb != (OS_TCB *)0) //如果该任务非空
{
        OS_CRITICAL_ENTER(); //进入临界段
        p_tcb->CyclesTotalPrev = p_tcb->CyclesTotal; (7)//保存任务的运行周期
        p_tcb->CyclesTotal = (OS_CYCLES)0; //复位运行周期，为下次运行做准备
        OS_CRITICAL_EXIT(); //退出临界段

        cycles_total+=p_tcb->CyclesTotalPrev; (8)//所有任务运行周期的总和

        CPU_CRITICAL_ENTER(); //关中断
        p_tcb = p_tcb->DbgNextPtr;

        // 用于整体 CPU 利用率的计算
// 获取列表的下一个任务，进行下一次循环
CPU_CRITICAL_EXIT(); // 开中断
}
#endif

/* 使用算法计算各个任务的 CPU 利用率和任务栈用量 */
#if OS_CFG_TASK_PROFILE_EN > 0u
// 如果启用了任务的统计功能

if (cycles_total > (OS_CYCLES)0u) // 如果有任务占用过 CPU
{
    if (cycles_total < 400000u)
    // 这些分类是为了避免计算 CPU 利用率过程中
    {
        cycles_mult = 10000u; // 产生溢出，就是避免相乘时超出 32 位寄存器。
        cycles_div = 1u;
    }
    else if (cycles_total < 4000000u)
    {
        cycles_mult = 1000u;
        cycles_div = 10u;
    }
    else if (cycles_total < 40000000u)
    {
        cycles_mult = 100u;
        cycles_div = 100u;
    }
    else if (cycles_total < 400000000u)
    {
        cycles_mult = 10u;
        cycles_div = 1000u;
    }
}


```c
else
{
    cycles_mult = 1u;
    cycles_div = 10000u;
}
cycles_max = cycles_total / cycles_div;
}

else //如果没有任务占用过 CPU
{
    cycles_mult = 0u;
    cycles_max = 1u;
}
#endif
CPU_CRITICAL_ENTER();   //关中断
p_tcb = OSTaskDbgListPtr;
//获取任务双向调试列表的首个任务
CPU_CRITICAL_EXIT();     //开中断
while (p_tcb != (OS_TCB *)0) //如果该任务非空
{
    #if OS_CFG_TASK_PROFILE_EN > 0u
    //如果启用了任务控制块的简况变量
    usage = (OS_CPU_USAGE)(cycles_mult * //计算任务的 CPU 利用率
    p_tcb->CyclesTotalPrev / cycles_max);   (9)
    if (usage > 10000u) //任务的 CPU 利用率为 100%
    {
        usage = 10000u;
    }
    p_tcb->CPUUsage = usage;         //保存任务的 CPU 利用率
    if (p_tcb->CPUUsageMax < usage) //更新任务的最大 CPU 利用率的历史记录
    {
```

(续上页)

论坛: [https://www.firebbs.cn/](https://www.firebbs.cn/)  
天猫: [https://yehuosm.tmall.com](https://yehuosm.tmall.com)
189 p_tcb->CPUUsageMax = usage;
190 }
191 #endif
192 /* 找检测 */
193 #if OS_CFG_STAT_TASK_STK_CHK_EN > 0u//如果启用了任务栈检测
194 OSTaskStkChk( p_tcb,                   //计算被激活任务的栈用量
195  &p_tcb->StkFree,
196  &p_tcb->StkUsed,
197  &err); (10)
198 #endif

199 CPU_CRITICAL_ENTER();                //关中断
200 p_tcb = p_tcb->DbgNextPtr;
201 //获取列表的下一个任务，进行下一次循环
202 CPU_CRITICAL_EXIT();                //开中断
203 }
204 #endif

205 if (OSStatResetFlag == DEF_TRUE)      //如果需要复位统计
206 {
207     OSStatResetFlag = DEF_FALSE;
208     OSStatReset(&err);              //复位统计
209 }
210
211 ts_end = OS_TS_GET() - ts_start;     //计算统计任务的执行时间
212 if (OSStatTaskTimeMax < ts_end)      //更新统计任务的最大执行时间的历史记录
213 {
214     OSStatTaskTimeMax = ts_end;
215 }
216
217 OSTimeDly(dly);                     //按照先前计算的执行节拍数延时
OS_OPT_TIME_DLY,
   &err);             (11)
]

- **代码清单: 利用率-2 (1):** 如果统计任务被允许运行，就让它一直延时，直到允许被运行为止，当统计任务准备就绪，就会调用 OSSStatReset() 函数复位。

- **代码清单: 利用率-2 (2):** 根据设置的宏计算统计任务的执行频率，这与我们前面讲解的定时器任务很像。

- **代码清单: 利用率-2 (3):** 进入统计任务主体代码，获取上一次空闲任务的计数值保存在 OSSStatTaskCtrRun 变量中，然后进行下一次空闲任务计数清零。

- **代码清单: 利用率-2 (4):** 计算 CPU 利用率，如果空闲任务的计数值小于最大空闲的计数值，表示是正常的，然后根据算法得到 CPU 的利用率，对 OSSStatTaskCtrMax 值的大小进行分类是为了避免计算 CPU 利用率过程中产生溢出。

- **代码清单: 利用率-2 (5):** 通过算法得到 CPU 的利用率 OSSStatTaskCPUUsage。算法很简单，如果不会就代一个数值进去计算一下就能得到。

- **代码清单: 利用率-2 (6):** 如果空闲任务计数值大于或等于最大空闲的计数值，说明 CPU 利用率为 0，CPU 一直在空闲任务中计数。

- **代码清单: 利用率-2 (7):** 下面计算各个任务的 CPU 利用率，原理跟计算整体 CPU 利用率相似，不过却要启用 OS_CFG_DBG_EN 与 OS_CFG_TASK_PROFILE_EN 宏定义，保存任务的运行周期。

- **代码清单: 利用率-2 (8):** 所有被统计的任务运行周期相加得到一个总的运行周期。

- **代码清单: 利用率-2 (9):** 与计算整体 CPU 利用率一样，计算得到各个任务的 CPU 利用率。

- **代码清单: 利用率-2 (10):** 如果启用了任务栈检测，调用 OSTaskStkChk() 函数进行任务的栈检测，在下文讲解该函数。

- **代码清单: 利用率-2 (11):** 按照先前计算的执行节拍数延时，因为统计任务也是按照周期运行的。

论坛：https://www.firebbs.cn/  794  天猫：https://yehuosm.tmall.com
31.6 栈检测 OSTaskStkChk()

uC/OS 提供 OSTaskStkChk() 函数用来进行栈检测，在使用之前必须将宏定义 OS_CFG_STAT_TASK_STK_CHK_EN 配置为 1，对于需要进行任务栈检测的任务，在其被 OSTaskCreate() 函数创建时，选项参数 opt 还需包含 OS_OPT_TASK_STK_CHK。统计任务会以我们设定的运行频率不断更新栈使用的情况并且保存到任务控制块的 StkFree 和 StkUsed 成员变量中，这两个变量分别表示任务栈的剩余空间与已使用空间大小，单位为任务栈大小的单位 (在 STM32 中采用 4 字节)，其源码具体见 代码清单: 利用率-3。

列表 3: 代码清单: 利用率-3OSTaskStkChk() 源码

```c
#if OS_CFG_STAT_TASK_STK_CHK_EN > 0u //如果启用了任务栈检测
void OSTaskStkChk (OS_TCB *p_tcb, (1) //目标任务控制块的指针
    CPU_STK_SIZE *p_free, (2) //返回空闲栈大小
    CPU_STK_SIZE *p_used, (3) //返回已用栈大小
    OS_ERR *p_err) (4) //返回错误类型
{
    CPU_STK_SIZE free_stk;
    CPU_STK *p_stk;
    CPU_SR_ALLOC(); //使用到临界段 (在关/开中断时) 时必须用到该宏，该宏声明和
    //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
    // SR (临界段关中断只需保存 SR)，开中断时将该值还原。
#endif

#ifdef OS_SAFETY_CRITICAL //如果启用了安全检测
    if (p_err == (OS_ERR *)0) (5) //如果 p_err 为空
        { //执行安全检测异常函数
            OS_SAFETY_CRITICAL_EXCEPTION();
            return;
        }
#endif

#if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u //如果启用了中断中非法调用检测
```

(下页继续)
if (OSIntNestingCtr > (OS_NESTING_CTR)0)  //如果该函数是在中断中被调用
{
    *p_err = OS_ERR_TASK_STK_CHK_ISR;  //错误类型为“在中断中检测栈”
    return;  //返回，停止执行
}
#endif

#if OS_CFG_ARG_CHK_EN > 0u  //如果启用了参数检测
if (p_free == (CPU_STK_SIZE)0)  //如果 p_free 为空
{
    *p_err = OS_ERR_PTR_INVALID;  //错误类型为“指针非法”
    return;  //返回，停止执行
}
#endif

if (p_used == (CPU_STK_SIZE)0)  //如果 p_used 为空
{
    *p_err = OS_ERR_PTR_INVALID;  //错误类型为“指针非法”
    return;  //返回，停止执行
}
CPU_CRITICAL_ENTER();  //关中断
if (p_tcb == (OS_TCB *)0)  //如果 p_tcb 为空
{
    p_tcb = OSTCBCurPtr;  //目标任务为当前运行任务（自身）
}
if (p_tcb->StkPtr == (CPU_STK*)0)  //如果目标任务的栈为空
{
    CPU_CRITICAL_EXIT();  //开中断
    *p_free = (CPU_STK_SIZE)0;  //清零 p_free
}
```c
*p_used = (CPU_STK_SIZE) 0; //清零 p_used
*p_err = OS_ERR_TASK_NOT_EXIST; //错误类型为“任务不存在”
return;
}
/* 如果目标任务的栈非空 */
if ((p_tcb->Opt & OS_OPT_TASK_STK_CHK) == (OS_OPT) 0) (7)
//如果目标任务没有选择检测栈
{
    CPU_CRITICAL_EXIT(); //开中断
    *p_free = (CPU_STK_SIZE) 0; //清零 p_free
    *p_used = (CPU_STK_SIZE) 0; //清零 p_used
    *p_err = OS_ERR_TASK_OPT; //错误类型为“任务选项有误”
    return; //返回，停止执行
}
CPU_CRITICAL_EXIT();
//如果任务选择了检测栈，开中断
/* 开始计算目标任务的栈的空闲数目和已用数目 */
free_stk = 0; (8) //初始化计算栈工作
#if CPU_CFG_STK_GROWTH == CPU_STK_GROWTH_HI_TO_LO
//如果 CPU 的栈是从高向低增长
p_stk = p_tcb->StkBasePtr; (9)
//从目标任务栈最低地址开始计算
while (*p_stk == (CPU_STK) 0) //计算值为 0 的栈数目
{
    p_stk++;
    free_stk++; (10)
}
#else
//如果 CPU 的栈是从低向高增长
p_stk = p_tcb->StkBasePtr + p_tcb->StkSize - 1u;
//从目标任务栈最高地址开始计算
```

while (*p_stk == (CPU_STK)0)  //计算值为 0 的栈数目
{
    free_stk++;
    p_stk--;  (11)
}
#endif
*p_free = free_stk;
//返回目标任务栈的空闲数目
*p_used = (p_tcb->StkSize - free_stk);  (12)
//返回目标任务栈的已用数目
*p_err = OS_ERR_NONE;                  //错误类型为“无错误”
}
#endif

• 代码清单: 利用率-3 (1): 目标任务控制块的指针。
• 代码清单: 利用率-3 (2): p_free 用于保存返回空闲栈大小。
• 代码清单: 利用率-3 (3): p_used 用于保存返回已用栈大小。
• 代码清单: 利用率-3 (4): p_err 用于保存返回错误类型。
• 代码清单: 利用率-3 (5): 如果 p_tcb 为空，目标任务为当前运行任务（自身）。
• 代码清单: 利用率-3 (6): 如果目标任务的栈为空，系统将 p_free 与 p_used 清零，返回错误类型为“任务不存在”的错误代码。
• 代码清单: 利用率-3 (7): 如果目标任务的栈非空，但是用户在创建任务的时候没有选择检测栈，那么系统将 p_free 与 p_used 清零，返回错误类型为“任务选项有误”的错误代码。
• 代码清单: 利用率-3 (8): 初始化计算栈工作。
• 代码清单: 利用率-3 (9): 通过宏定义 CPU_CFG_STK_GROWTH 选择 CPU 栈生长的方向，如果 CPU 的栈是从高向低增长，从目标任务栈最低地址开始计算。
• 代码清单: 利用率-3 (10): 计算栈空间中内容为 0 的栈大小，栈空间地址递增。
[野火]uCOS-III 内核实现与应用开发实战指南

- 代码清单: 利用率-3 (11): 如果 CPU 的栈是从低向高增长，从目标任务栈最高地址开始计算内容为 0 的栈大小，栈空间地址递减。
- 代码清单: 利用率-3 (12): 返回目标任务栈的空闲大小与已用大小。

注意: 我们自己也可以调用该函数进行统计某个任务的栈空间使用情况。

31.7 任务栈大小的确定

任务栈的大小取决于该任务的需求，设定栈大小时，我们就需要考虑: 所有可能被栈调用的函数及其函数的嵌套层数，相关局部变量的大小，中断服务程序所需要的空间，另外，栈还需存入 CPU 寄存器，如果处理器有浮点数单元 FPU 寄存器的话还需存入 FPU 寄存器。

嵌入式系统的潜规则，避免将递归函数，这样子可以人为计算出一个任务需要的栈空间大小，逐级嵌套所有可能被调用的函数，计数被调用函数中所有的参数，计算上下文切换时的 CPU 寄存器空间，计算切换到中断时所需的 CPU 寄存器空间 (假如 CPU 没有独立的栈用于处理中断)，计算处理中断服务函数 (ISR) 所需的栈空间，将这些值相加即可得到任务最小的需求空间，但是我们不可能算得出来的准确的栈空间，我们通常会将这个值再乘以 1.5 到 2.0 以确保任务的安全运行。这个计算的值是基于任务所执行的执行路线都是已知的情况下的，但这在真正的应用中并不太可能，比如说，如果调用 printf() 函数或者其他的函数，这些函数所需要的空间是很难测得或者说就是不可能知道的，在这种情况下，我们这种人为计算任务栈大小的方法就变得不太可能了，那么我们可以在刚开始创建任务的时候给任务设置一个较大的栈空间，并监测该任务运行时栈空间的实际使用量，运行一段时间后得到任务的最大栈使用情况 (或者叫任务栈最坏结果)，然后用该值乘 1.5 到 2.0 作为栈空间大小就差不多可以作为任务栈的空间大小，这样子得到的值就会比较精确一点，在调试阶段可以这样子进行测试，发现崩溃就增大任务的栈空间，直到任务能正常稳定运行为止。

论坛: https://www.firebbs.cn/     799     天猫: https://yehuosm.tmall.com
31.8 CPU 利用率及栈检测统计实验

CPU 利用率及栈检测统计实验是在 μC/OS 中创建了四个任务，其中三个任务是普通任务，另一个任务用于获取 CPU 利用率与任务相关信息并通过串口打印出来。具体见代码清单: 利用率-4。

列表 4: 代码清单: 利用率-4CPU 利用率及栈检测统计实验

```c
#include <includes.h>

static OS_TCB AppTaskStartTCB;

static OS_TCB AppTaskLed1TCB;
static OS_TCB AppTaskLed2TCB;
static OS_TCB AppTaskLed3TCB;
static OS_TCB AppTaskStatusTCB;

static CPU_STK AppTaskStartStk[APP_TASK_START_STK_SIZE];

static CPU_STK AppTaskLed1Stk[APP_TASK_LED1_STK_SIZE];
static CPU_STK AppTaskLed2Stk[APP_TASK_LED2_STK_SIZE];
static CPU_STK AppTaskLed3Stk[APP_TASK_LED3_STK_SIZE];
static CPU_STK AppTaskStatusStk[APP_TASK_STATUS_STK_SIZE];

static void AppTaskStart (void *p_arg);
static void AppTaskLed1 (void *p_arg);
static void AppTaskLed2 (void *p_arg);
static void AppTaskLed3 (void *p_arg);
static void AppTaskStatus (void *p_arg);
```

论坛: https://www.firebbs.cn/  800  天猫: https://yehuosm.tmall.com
```c
int main (void)
{
    OS_ERR err;

    OSInit(&err); /* Init μC/OS-III. */

    OSTaskCreate((OS_TCB *)&AppTaskStartTCB,

        (CPU_CHAR *)"App Task Start",
        (OS_TASK_PTR ) AppTaskStart,
        (void *) 0,
        (OS_PRIO ) APP_TASK_START_PRIO,
        (CPU_STK *)&AppTaskStartStk[0],
        (CPU_STK_SIZE) APP_TASK_START_STK_SIZE / 10,
        (CPU_STK_SIZE) APP_TASK_START_STK_SIZE,
        (OS_MSG_QTY ) 5u,
        (OS_TICK ) 0u,
        (void *) 0,
        (OS_OPT ) (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
        (OS_ERR * )&err);

    OSStart(&err);
}
```
static void AppTaskStart (void *p_arg)
{
    CPU_INT32U cpu_clk_freq;
    CPU_INT32U cnts;
    OS_ERR err;

    (void)p_arg;

    BSP_Init();
    CPU_Init();

    cpu_clk_freq = BSP_CPU_ClkFreq();

    cnts = cpu_clk_freq / (CPU_INT32U)OSCfg_TickRate_Hz;

    OS_CPU_SysTickInit(cnts);

    Mem_Init();

#if OS_CFG_STAT_TASK_EN > 0u

    OSStatTaskCPUUsageInit(&err);

#endif
CPU_IntDisMeasMaxCurReset();

/* Create the Led1 task */
OSTaskCreate((OS_TCB*)&AppTaskLed1TCB,
        (CPU_CHAR*)"App Task Led1",
        (OS_TASK_PTR) AppTaskLed1,
        (void*) 0,
        (OS_PRIO) APP_TASK_LED1_PRIO,
        (CPU_STK*) &AppTaskLed1Stk[0],
        (CPU_STK_SIZE) APP_TASK_LED1_STK_SIZE / 10,
        (CPU_STK_SIZE) APP_TASK_LED1_STK_SIZE,
        (OS_MSG_QTY) 5u,
        (OS_TICK) 0u,
        (void*) 0,
        (OS_OPT) (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
        (OS_ERR*) &err);

/* Create the Led2 task */
OSTaskCreate((OS_TCB*)&AppTaskLed2TCB,
        (CPU_CHAR*)"App Task Led2",
        (OS_TASK_PTR) AppTaskLed2,
        (void*) 0,
        (OS_PRIO) APP_TASK_LED2_PRIO,
        (CPU_STK*) &AppTaskLed2Stk[0],
        (CPU_STK_SIZE) APP_TASK_LED2_STK_SIZE / 10,
        (CPU_STK_SIZE) APP_TASK_LED2_STK_SIZE,
        (OS_MSG_QTY) 5u,
        (OS_TICK) 0u,
/* Create the Led3 task */
OSTaskCreate((OS_TCB *) &AppTaskLed3TCB,
    (CPU_CHAR *) "App Task Led3",
    (OS_TASK_PTR ) AppTaskLed3,
    (void *) 0,
    (OS_PRIO ) APP_TASK_LED3_PRIO,
    (CPU_STK *) &AppTaskLed3Stk[0],
    (CPU_STK_SIZE) APP_TASK_LED3_STK_SIZE / 10,
    (CPU_STK_SIZE) APP_TASK_LED3_STK_SIZE,
    (OS_MSG_QTY ) 5u,
    (OS_TICK ) 0u,
    (void *) 0,
    (OS_OPT ) (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
    (OS_ERR ) &err);

/* Create the status task */
OSTaskCreate((OS_TCB *) &AppTaskStatusTCB,
    (CPU_CHAR *) "App Task Status",
    (OS_TASK_PTR ) AppTaskStatus,
    (void *) 0,
    (OS_PRIO ) APP_TASK_STATUS_PRIO,
    (CPU_STK *) &AppTaskStatusStk[0],
    (CPU_STK_SIZE) APP_TASK_STATUS_STK_SIZE / 10,
    (CPU_STK_SIZE) APP_TASK_STATUS_STK_SIZE,
    (OS_MSG_QTY ) 5u,
    (OS_TICK ) 0u,
    (void *) 0,
    (OS_OPT ) (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
    (OS_ERR ) &err);
(OS_ERR *) &err);

OSTaskDel (& AppTaskStartTCB, & err);

} //TaskDelEnd

static void AppTaskLed1 ( void * p_arg )
{
    OS_ERR err;
    uint32_t i;

    (void)p_arg;

    while (DEF_TRUE)
    {
        printf("AppTaskLed1 Running\n");

        for (i=0; i<10000; i++) //模拟任务占用 cpu
        {
            ;
        }

        macLED1_TOGGLE ();
        OSTimeDlyHMSM (0, 0, 0, 500, OS_OPT_TIME.PERIODIC, &err);
    }
}


static void AppTaskLed2 ( void * p_arg )
{
    OS_ERR err;
    uint32_t i;

    (void)p_arg;

    while (DEF_TRUE)
    {
        printf("AppTaskLed2 Running\n");

        for (i=0; i<100000; i++)  //模拟任务占用 cpu
        {
            ;
        }
        macLED2_TOGGLE ();
        OSTimeDlyHMSM (0,0,0,500,OS_OPT_TIME_PERIODIC,&err);
    }
}
static void AppTaskLed3 ( void * p_arg )
{
    OS_ERR err;

    uint32_t i;
    (void)p_arg;

    while (DEF_TRUE)
    {
        macLED3_TOGGLE ();

        for (i=0; i<500000; i++)  //模拟任务占用 cpu
        {
            ;
        }

        printf("AppTaskLed3 Running\n");

        OSTimeDlyHMSM (0,0,500,OS_OPT_TIME_PERIODIC,&err);
    }
}

static void AppTaskStatus ( void * p_arg )
{
    OS_ERR err;

    CPU_SR_ALLOC();
}
(void)p_arg;

while (DEF_TRUE)
{

    OS_CRITICAL_ENTER();
    //进入临界段，避免串口打印被打断
    printf("------------------------
    ");
    printf ( "CPU 利用率: %d.%d\r\n", 
            OSStatTaskCPUUsage / 100, OSStatTaskCPUUsage % 100 );
    printf ( "CPU 最大利用率: %d.%d\r\n", 
            OSStatTaskCPUUsageMax / 100, OSStatTaskCPUUsageMax % 100 );

    printf ( "LED1 任务的 CPU 利用率: %d.%d\r\n", 
            AppTaskLed1TCB.CPUUsageMax / 100, AppTaskLed1TCB.CPUUsageMax % 100 );
    printf ( "LED1 任务的 CPU 利用率: %d.%d\r\n", 
            AppTaskLed2TCB.CPUUsageMax / 100, AppTaskLed2TCB.CPUUsageMax % 100 );
    printf ( "LED1 任务的 CPU 利用率: %d.%d\r\n", 
            AppTaskLed3TCB.CPUUsageMax / 100, AppTaskLed3TCB.CPUUsageMax % 100 );
    printf ( "统计任务的 CPU 利用率: %d.%d\r\n", 
            AppTaskStatusTCB.CPUUsageMax / 100, AppTaskStatusTCB.CPUUsageMax % 100 );
    printf ( "LED1 任务的已用和空闲栈大小分别为: %d,%d\r\n", 
            AppTaskLed1TCB.StkUsed, AppTaskLed1TCB.StkFree );
    printf ( "LED2 任务的已用和空闲栈大小分别为: %d,%d\r\n", 
            AppTaskLed2TCB.StkUsed, AppTaskLed2TCB.StkFree );
}


AppTaskLed2TCB.StkUsed, AppTaskLed2TCB.StkFree );
printf ( "LED3 任务的已用和空闲栈大小分别为: %d,%d\r\n",
AppTaskLed3TCB.StkUsed, AppTaskLed3TCB.StkFree );
printf ( "统计任务的已用和空闲栈大小分别为: %d,%d\r\n",
AppTaskStatusTCB.StkUsed, AppTaskStatusTCB.StkFree );

printf("-----------------------------------------------\n"):
OS_CRITICAL_EXIT();                              //退出临界段
OSTimeDlyHMSM (0,0,0,500,OS_OPT_TIME_PERIODIC,&err);
}
}

### 31.9 CPU 利用率及栈检测统计实验现象

程序编译好，用 USB 线连接计算机和开发板的 USB 接口（对应丝印为 USB 转串口），用 DAP 仿真器把配套程序下载到野火 STM32 开发板（具体型号根据购买的板子而定，每个型号的板子都配套有对应的程序），在计算机上打开串口调试助手，然后复位开发板就可以在调试助手中看到串口的打印信息，具体见图 CPU 利用率及栈检测统计实验现象。
论坛：https://www.firebbs.cn/  810  天猫：https://yehuosm.tmall.com
第 32 章  rst 投稿示例文件
如何参与项目

本项目从代码到文档全部开源，期待大家的参与！

无论是小到发现错别字、发现代码写得不规范，有 bug 抑或是想参与到文档的编辑、投稿完整的文章，我们都热烈欢迎。

具体参与方式请阅读如下说明：如何参与野火开源项目

项目源码

本项目代码及文档源码托管在 github 和 gitee，维护时主要以 github 开发，gitee 主要是方便国内用户下载，可在这两个平台提交 pull request 参与到本项目。

- github 仓库地址：https://github.com/Embedfire-ucos/ebf_ucos_tutorial
- gitee 仓库地址：https://gitee.com/Embedfire-ucos/ebf_ucos_tutorial

文档在线阅读与 pdf

本项目的文档是托管在 readthedocs 网站的，可在该网站在线浏览文档或者生成 pdf 文档离线阅读。

论坛：https://www.firebbs.cn

天猫：https://yehuosm.tmall.com
常见问题
版权说明

野火电子保留本项目的所有版权。

公司组织有生存的压力，因为开源而导致生存不下去，是有 GEEK 精神的人不愿看到的事情，由于我们还不熟悉各种版权条例，所以关于版权的问题还在选择中。

目前我们保留本项目的所有版权，但我们秉承开源的心是不变的。如果你使用本项目不是用于商业目的，基本不需要考虑版权问题。

我们主要担忧的商业目的如下，列出来的意思是因为用于以下目的，我们极有可能会追究版权。

- 同业竞争，目前开发板是我们主要的盈利来源，禁止把本项目用于其它开发板项目。
- 文档出版，项目中的包含的文档我们都会提交至出版社出版，所以我们保留文档出版的权利。如果你只是在自己的文章中使用了本项目文档中的内容，需要注明来源。

论坛：https://www.firebbs.cn
天猫：https://yehuosm.tmall.com