uC/OS-III 应用开发指南

——基于 STM32F4 系列

修订历史

日期	版本	更新内容
2015/10/28	1.0.0	-

uC/OS-III 应用开发指南.....1
前言.....6
第 1 章 uC/OS-III 概要.....8
1.1 uC/OS-III 文件结构.....8
1.2 uC/OS-III 数据结构.....8
1.3 uC/OS-III 内核对象.....12
1.3.1 任务.....12
1.3.2 软件定时器.....14
1.3.3 多值信号量.....14
1.3.4 互斥信号量.....14
1.3.5 消息队列.....14
1.3.6 事件标志组.....15
1.3.7 任务信号量.....15
1.3.8 任务消息队列.....15
1.3.9 内存管理（分区）.....15
1.4 uC/OS-III 常用程序段.....15
1.4.1 临界段.....15
1.4.2 中断嵌套管理.....17
1.5 章末总结.....18
第 2 章 移植 uC/OS-III 到 STM32.....19
2.1 下载官方 uC/OS-III 源码.....19
2.2 移植过程.....22
2.3 建立多任务工程.....48
2.4 章末总结.....53
第 3 章 时钟节拍.....54
3.1 原理简述.....54
3.2 实例演示.....58
3.2.1 实例 1.....58
3.3 章末总结.....63
第 4 章 时间管理.....64
4.1 原理简述.....64
4.1.1 OSTimeDly().....64
4.1.2 OSTimeDlyHMSM().....67
4.1.3 OSTimeDlyResume().....71
4.1.4 OSTimeGet ().....73
4.1.5 OSTimeSet ().....74
4.2 实例演示.....75
4.2.1 实例 1.....75
4.2.2 实例 2.....80
4.3 章末总结.....83
第 5 章 软件定时器.....84
5.1 原理简述.....84
5.1.1 OSTmrCreate ().....84
5.1.2 OSTmrStart ().....87

5.1.3 OSTmrStop () 91
5.1.4 OSTmrDel () 93
5.2 实例演示 95
5.2.1 实例 1 95
5.3 章末总结 97
第 6 章 多值信号量 99
6.1 原理简述 99
6.1.1 OSSemCreate () 99
6.1.2 OSSemPost () 101
6.1.3 OSSemPend () 106
6.1.4 OSSemPendAbort () 110
6.1.5 OSSemDel() 113
6.1.6 OSSemSet() 117
6.2 实例演示 119
6.2.1 实例 1 119
6.2.2 实例 2 122
6.3 章末总结 127
第 7 章 互斥信号量 128
7.1 原理简述 128
7.1.1 OSMutexCreate () 128
7.1.2 OSMutexPost () 130
7.1.3 OSMutexPend () 133
7.1.4 OSMutexPendAbort() 138
7.1.5 OSMutexDel() 141
7.2 实例演示 146
7.2.1 实例 1 146
7.3 章末总结 153
第 8 章 消息队列 154
8.1 原理简述 154
8.1.1 OSQCreate () 156
8.1.2 OSQPost () 158
8.1.3 OSQPend () 164
8.1.4 OSQPendAbort () 170
8.1.5 OSQDel () 173
8.1.6 OSQFlush () 176
8.2 实例演示 178
8.2.1 实例 1 178
8.3 章末总结 181
第 9 章 事件标志组 182
9.1 原理简述 182
9.1.1 OSFlagCreate () 182
9.1.2 OSFlagPost () 184
9.1.3 OSFlagPend () 189
9.1.4 OSFlagPendAbort () 197

9.1.5 OSFlagDel ()....200
9.2 实例演示....203
9.2.1 实例 1....203
9.3 章末总结....206
第 10 章 等待多个内核对象....207
10.1 原理简述....207
10.1.1 OSPendMulti ()....207
10.2 实例演示....213
10.2.1 实例 1....213
10.3 章末总结....217
第 11 章 任务信号量....218
11.1 原理简述....218
11.1.1 OSTaskSemPost ()....218
11.1.2 OSTaskSemPend ()....223
11.1.3 OSTaskSemPendAbort ()....226
11.2 实例演示....229
11.2.1 实例 1....229
11.3 章末总结....232
第 12 章 任务消息队列....233
12.1 原理简述....233
12.1.1 OSTaskQPost ()....233
12.1.2 OSTaskQPend ()....239
12.1.3 OSTaskQPendAbort ()....243
12.2 实例演示....246
12.2.1 实例 1....246
12.3 章末总结....249
第 13 章 内存管理....250
13.1 原理简述....250
13.1.1 OSMemCreate ()....250
13.1.2 OSMemGet ()....253
13.1.3 OSMemPut ()....254
13.2 实例演示....256
13.2.1 实例 1....256
13.3 章末总结....259
第 14 章 任务管理....260
14.1 原理简述....260
14.1.1 OSTaskCreate ()....260
14.1.2 OSTaskSuspend ()....265
14.1.3 OSTaskResume ()....267
14.1.4 OSTaskChangePrio ()....270
14.1.5 OSTaskDel ()....272
14.1.6 OSSchedRoundRobinCfg ()....275
14.1.7 OSSchedRoundRobinYield ()....277
14.1.8 OSTaskTimeQuantaSet ()....279

14.1.9 OSTaskRegSet ()....280
14.1.10 OSTaskRegSet ()....282
14.2 实例演示....284
14.2.1 实例 1....284
14.3 章末总结....287
第 15 章 中断管理....288
15.1 原理简述....288
15.1.1 OSIntEnter ()....292
15.1.2 OSIntExit ()....293
15.1.3 CPU_IntDisMeasMaxGet ()....294
15.1.4 CPU_IntDisMeasMaxCurReset ()....296
15.1.5 CPU_IntDisMeasMaxCurGet ()....297
15.2 实例演示....298
15.2.1 实例 1....298
15.3 章末总结....303
第 16 章 统计信息....304
16.1 原理简述....304
16.1.1 统计任务....304
16.1.2 CPU 主频....308
16.1.3 uC/OS 版本号....309
16.1.4 其他统计信息....310
16.2 实例演示....310
16.2.1 实例 1....310
16.3 章末总结....315

前言

本书讲述的 uC/OS 是一个操作系统，既然是系统，那就是一个整体，所以书中各章节并不完全独立，章与章之间的内容皆有所交叉，只是各章均侧重本章标题对应的系统内核机制。

自从 2009 年 uC/OS-III 实时操作系统面世以来，uC/OS-III 的性能和稳定性已经日趋完善。uC/OS-III 对前面版本取精弃粕，推陈出新，功能和性能比起 uC/OS-II 已经上了很大一步台阶。uC/OS-III 正在逐渐取代 uC/OS-II，学习 uC/OS-III 已经成为许多嵌入式工程师迫在眉睫的一件事。下面是 uC/OS 系统各个版本的对比。

表 1 uC/OS 系统各版本对比

功能	uC/OS-I	uC/OS-II	uC/OS-III
诞生年份	1992	1998	2009
书	有	有	有
提供源码	是	是	是
抢占式多任务	是	是	是
最大任务数	64	256	无限制
每个优先级的任务数	1	1	无限制
时间片轮转调度	否	否	是
软件定时器	否	是	是
多值信号量	是	是	是
互斥信号量	否	是	是（可嵌套）
时间标志组	否	是	是
消息邮箱	是	是	否（消息队列可实现）
消息队列	是	是	是
内存管理	否	是	是
任务信号量	否	否	是
任务消息队列	否	否	是
挂起/恢复任务	否	是	是（可嵌套）
死锁预防	是	是	是
可扩展	是	是	是
代码段需求	3K 到 8K	6K 到 26K	6K 到 20K
数据段需求	1K+	1K+	1K+
可固化	是	是	是
在运行时配置	否	否	是
在编译时配置	是	是	是
对象可命名	否	是	是
可挂起多个任务	是	是	是
任务寄存器	否	是	是
内置性能测量	否	少量	大量
用户可定义的 hook 函数	否	是	是
时间戳	否	否	是

嵌入的内核调试	否	是	是
汇编可优化	否	否	是
时基任务	否	否	是
提供的服务	~20	~90	~70

uC/OS-III 的源码已经跟前面版本的源码相差很大，很多方面都做了更加规范的修改，很多处理问题的机制也有很大的突破。因此，如果通过前面版本的书籍来理解 uC/OS-III，还是会比较费解。现在市面上讲解 uC/OS-III 的书籍或资料都寥寥无几，讲解源码原理的更是少之又少。

特此问题，秉火科技推出了本书籍，带领有兴趣的学者挖掘 uC/OS-III 实时系统。本书的讲解思路是以 uC/OS-III 的内核机制分章，每章先带领学者了解内核机制的源码运作原理，然后再在秉火 STM32F4 系列的开发板上演示相关的实例，让学者能够更贴切体会到内核机制的功能。秉火 STM32F4 系列的开发板使用的芯片是 STM32F429IGT。本书已为这套开发板配套了相关的例程。

本书的读者定位非常适合初学 uC/OS-III 的学者，或希望快速应用 uC/OS-III 实时系统的嵌入式开发人员，也适合想要深入了解 uC/OS-III 源码原理的技术员。

秉火团队

2015 年 10 月 15 日

第1章 uC/OS-III 概要

本章主要对 uC/OS-III 实时操作系统做一些概要介绍，使读者对 uC/OS-III 有个整体的浅认识，为后面的章节的详细讲解做一个铺垫。

1.1 uC/OS-III 文件结构

下图是 uC/OS-III 系统从底层到上层的文件结构。

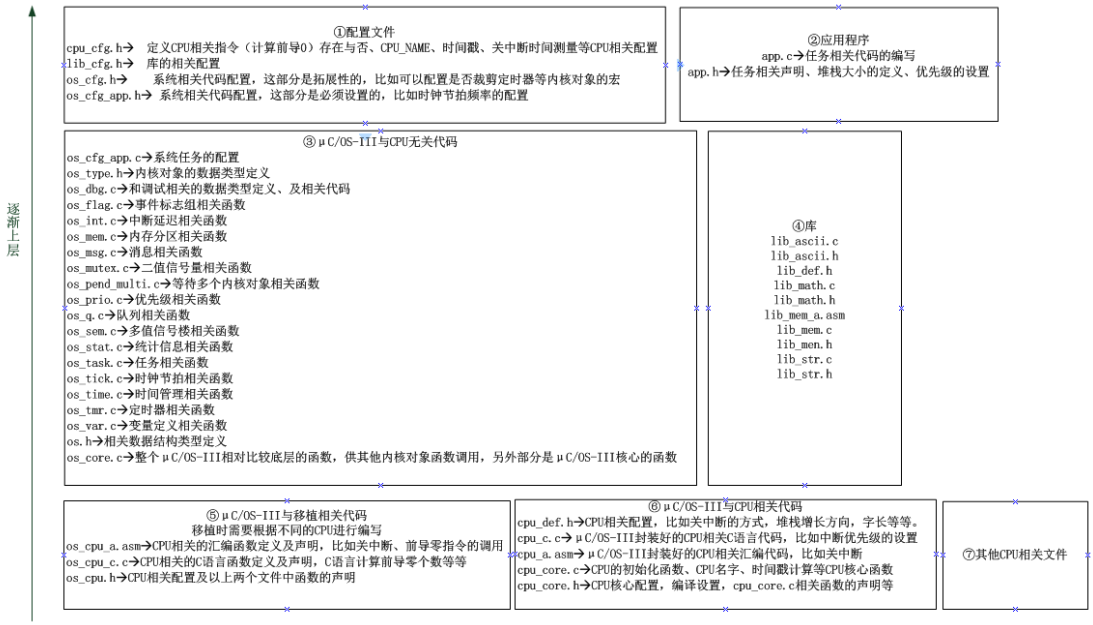

图 1-1 uC/OS-III 文件结构

①配置文件，通过定义这些文件里宏的值可以轻易地裁剪 uC/OS-III 的功能。
②用户应用文件，定义和声明应用任务。
③内核服务文件，其代码与 CPU 无关，可以不做任何修改移植到任何 CPU。本书主要讲解这部分内容。
④底层函数库，比如字符串的常规操作，常用的数学计算，等等。
⑤CPU 移植文件，用户如果想要移植 uC/OS-III 到不同平台上，需要修改这部分代码。
⑥CPU 配置文件，主要是 CPU 的一些工作模式和服务函数。
⑦其他 CPU 相关文件。

1.2 uC/OS-III 数据结构

uC/OS-III 中的内核对象大多都是以结构体的形式存在的，例如出现最多的任务的任务控制块的数据结构如下所示，结构体中的每个成员代表任务具有的一种属性。在结构体中，可以看到很多宏，通过定义这些宏，就可以轻易地裁剪任务控制块具有的属性（任务的功能）。

代码 1-1 任务控制块结构体

```
01 struct os_tcb {
02    CPU_STK            *StkPtr;
03
04    void               *ExtPtr;
05
06    CPU_STK            *StkLimitPtr;
07
08    OS_TCB             *NextPtr;
09    OS_TCB             *PrevPtr;
10
11    OS_TCB             *TickNextPtr;
12    OS_TCB             *TickPrevPtr;
13
14    OS_TICK_SPOKE       *TickSpokePtr;
15
16    CPU_CHAR           *NamePtr;
17
18    CPU_STK            *StkBasePtr;
19
20 #if defined(OS_CFG_TLS_TBL_SIZE) && (OS_CFG_TLS_TBL_SIZE > 0u)
21    OS_TLS              TLS_Tbl[OS_CFG_TLS_TBL_SIZE];
22 #endif
23
24    OS_TASK_PTR         TaskEntryAddr;
25    void               *TaskEntryArg;
26
27    OS_PEND_DATA       *PendDataTblPtr;
28    OS_STATE            PendOn;
29    OS_STATUS           PendStatus;
30
31    OS_STATE            TaskState;
32    OS_PRIO             Prio;
33    CPU_STK_SIZE        StkSize;
34    OS_OPT              Opt;
35
36    OS_OBJ_QTY          PendDataTblEntries;
37
38    CPU_TS              TS;
39
40    OS_SEM_CTR          SemCtr;
41
42
43    OS_TICK             TickCtrPrev;
44    OS_TICK             TickCtrMatch;
45    OS_TICK             TickRemain;
46
47    OS_TICK             TimeQuanta;
48    OS_TICK             TimeQuantaCtr;
49
50 #if OS_MSG_EN > 0u
51    void               *MsgPtr;
52    OS_MSG_SIZE         MsgSize;
53 #endif
54
55 #if OS_CFG_TASK_Q_EN > 0u
56    OS_MSG_Q            MsgQ;
57 #if OS_CFG_TASK_PROFILE_EN > 0u
58    CPU_TS              MsgQPendTime;
59    CPU_TS              MsgQPendTimeMax;
60 #endif
61 #endif
62
63 #if OS_CFG_TASK_REG_TBL_SIZE > 0u
64    OS_REG              RegTbl[OS_CFG_TASK_REG_TBL_SIZE];
65 #endif
```

```
66
67 #if OS_CFG_FLAG_EN > 0u
68    OS_FLAGS           FlagsPend;
69    OS_FLAGS           FlagsRdy;
70    OS_OPT             FlagsOpt;
71 #endif
72
73 #if OS_CFG_TASK_SUSPEND_EN > 0u
74    OS_NESTING_CTR      SuspendCtr;
75 #endif
76
77 #if OS_CFG_TASK_PROFILE_EN > 0u
78    OS_CPU_USAGE        CPUUsage;
79    OS_CPU_USAGE        CPUUsageMax;
80    OS_CTX_SW_CTR       CtxSwCtr;
81    CPU_TS             CyclesDelta;
82    CPU_TS             CyclesStart;
83    OS_CYCLES           CyclesTotal;
84    OS_CYCLES           CyclesTotalPrev;
85
86    CPU_TS             SemPendTime;
87    CPU_TS             SemPendTimeMax;
88 #endif
89
90 #if OS_CFG_STAT_TASK_STK_CHK_EN > 0u
91    CPU_STK_SIZE        StkUsed;
92    CPU_STK_SIZE        StkFree;
93 #endif
94
95 #ifdef CPU_CFG_INT_DIS_MEAS_EN
96    CPU_TS             IntDisTimeMax;
97 #endif
98 #if OS_CFG_SCHED_LOCK_TIME_MEAS_EN > 0u
99    CPU_TS             SchedLockTimeMax;
100 #endif
101
102 #if OS_CFG_DBG_EN > 0u
103    OS_TCB             *DbgPrevPtr;
104    OS_TCB             *DbgNextPtr;
105    CPU_CHAR           *DbgNamePtr;
106 #endif
107 };
```

在 uC/OS-III 中，对内核对象的管理大多采用线性链表的数据结构，包括单向链表和双向链表。链表就是将要管理的对象按照方便管理的规则一个接一个串联在一起，提高管理效率。比如下图的节拍列表就是一个双向链表，其中每个 OSCfgTickWheel 数组元素代表着一个时间点的节拍列表，相邻 OSCfgTickWheel 元素代表的时间点的对 OSCfgTickWheel 数组长度的余数相差一个时钟节拍。对于那些延时或有期限等待的任务的任务控制块 OS_TCB，会先根据其延时或等待的节拍对 OSCfgTickWheel 数组长度的余数插入到哪个节拍列表，然后再把该节拍列表里的任务控制块 OS_TCB 按照其延时或等待的节拍的大小顺序串联成双向链表。

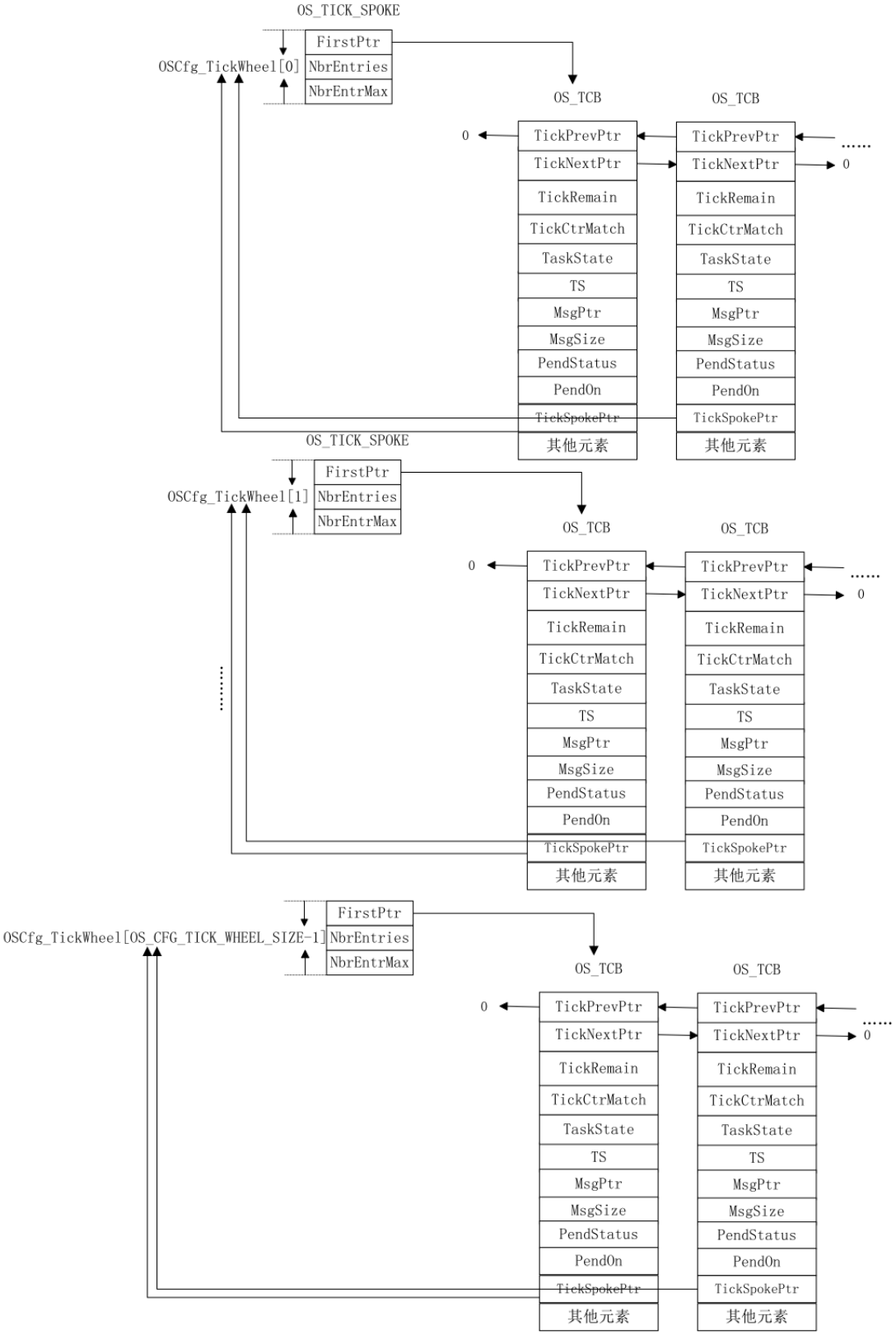

图 1-2 节拍列表数据结构

1.3 uC/OS-III 内核对象

1.3.1 任务

在 uC/OS-III 中，可以创建无数多个任务，让这些任务并发运行，就好像有多个主函数在运行一样。在 uC/OS-III 初始化的时候，至少会创建空闲任务 OS_IdleTask()和时基任务 OS_TickTask()这两个任务，另外还有三个可选择的内部任务，软件定时器任务 OS_TmrTaks() 、中断延迟提交任务 OS_IntQTask()和统计任务 OS_StatTask()。

从用户的角度来看，uC/OS-III 中的任务可以分为 5 种状态，分别是休眠态、就绪态、运行态、挂起态和中断态，如下表所示。

表 2 用户角度的任务状态

任务状态	描述
休眠态	声明了任务，但任务尚未被 OSTaskCreate()函数正式创建，该任务不受 uC/OS 系统管理。
就绪态	任务已被正式创建，而且已插入就绪列表，一旦获得 CPU 使用权，就可以运行。
运行态	任务正占有 CPU，正在运行。
等待态	任务被延时执行，需要等待某个事件（内核对象），或者被强制挂起时，就会进入等待态。
中断服务态	正在运行的任务突然被中断打断，CPU 被中断服务程序占有，该任务就进入了中断服务态。

任务状态之间的具体切换情况如下图所示。

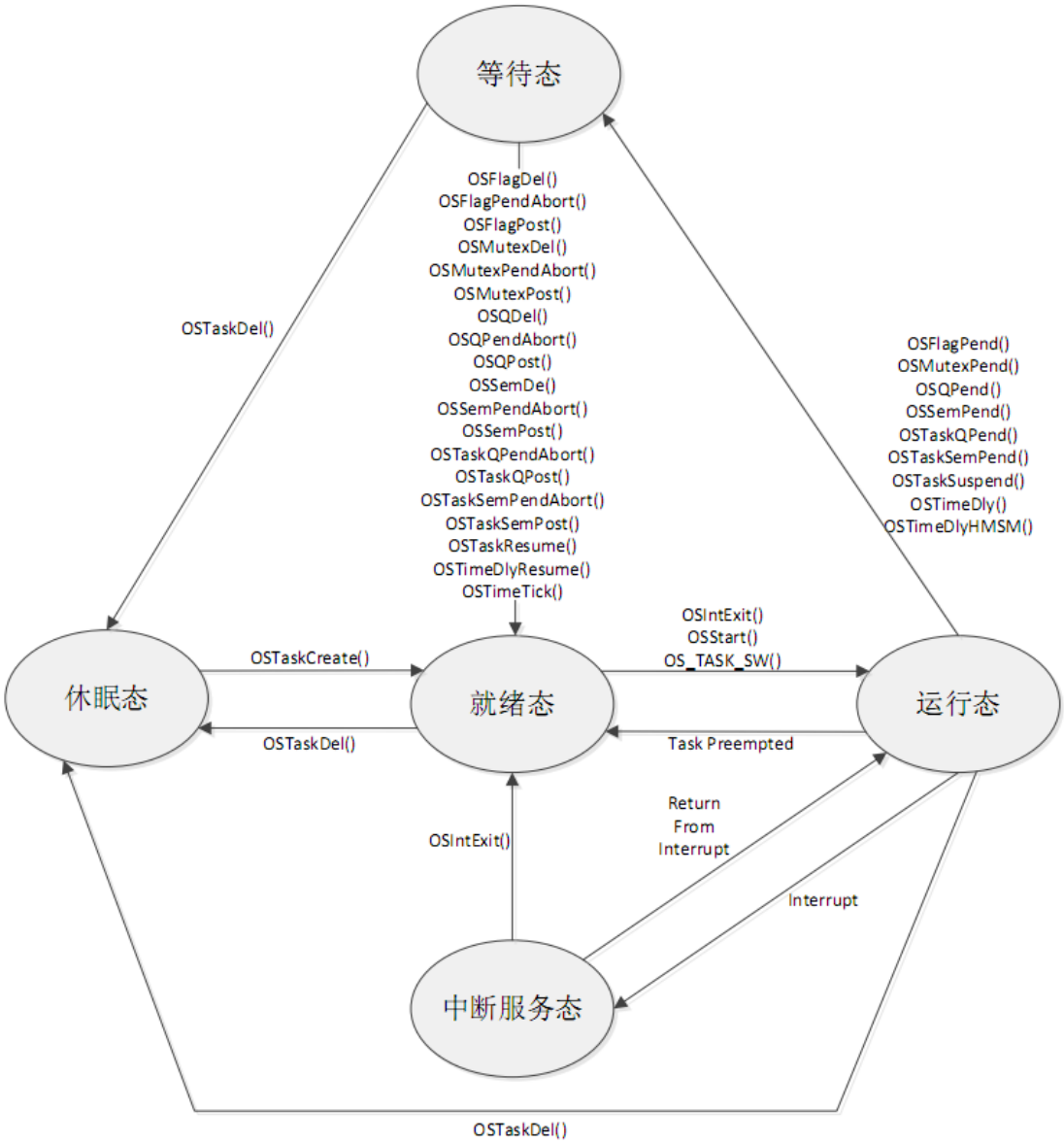

图 1-3 任务状态的切换

从 uC/OS-III 任务管理的角度来看，uC/OS-III 中的任务 9 种状态，如下表所示。分别是休眠态、就绪态、运行态、挂起态和中断态，如下表所示。

表 3 uC/OS-III 任务管理角度任务状态

任务状态	描述
OS_TASK_STATE_RDY	就绪状态。处于该状态的任务按照优先级高低先后占有 CPU 运行。
OS_TASK_STATE_DLY	延时状态。任务调用 uC/OS 的延时韩式 OSTimeDly()或 OSTimeDlyHMSM 后，就停止运行，进入延时状态。
OS_TASK_STATE_PEND	无期限等待状态。任务需要停止运行，等待某个事件（内核对象），直到等到才继续运行。

OS_TASK_STATE_PEND_TIMEOUT	有期限等待状态。任务需要停止运行，在一定时间内等待某个事件（内核对象），如果超时或事件发生了，就继续运行。
OS_TASK_STATE_SUSPENDED	挂起状态。任务被强制暂停运行，直到被恢复才可继续运行。
OS_TASK_STATE_DLY_SUSPENDED	延时中被挂起状态。任务在延时状态时，又被其它任务挂起。
OS_TASK_STATE_PEND_SUSPENDED	无期限等待中被挂起状态。任务在无期限等待某个事件（内核对象）时，又被其它任务挂起。
OS_TASK_STATE_PEND_TIMEOUT_SUSPENDED	有期限等待中被挂起状态。任务在有期限等待某个事件（内核对象）时，又被其它任务挂起。
OS_TASK_STATE_DEL	删除状态。任务被删除，不再参与任务管理，重新创建后才收 uC/OS 管理。

1.3.2 软件定时器

软件定时器的功能跟硬件定时器一样，主要用于定时，但其精度达不到硬件定时器的标准，可以用于定时一些精度要求不是特别严格的事件。理论上，uC/OS-III 可以创建无数个软件定时器，这是硬件定时器无法媲美的。

1.3.3 多值信号量

多值信号量主要用于管理资源和标志事件的发生。管理资源的一个常用仿例就是停车场，把总停车位看做信号量，每次申请一个停车位信号量就减 1，如果停车位为 0，就申请不到，但可以等待其它汽车释放停车位。标志事件的发生类似于裸机里常用的事件标志变量，就是标志某事是否发生，然后通知任务。

1.3.4 互斥信号量

互斥信号量的作用是保护共享资源，避免共享资源正在被重写时被其它任务读取，这样读取到的数据就有错误。互斥信号量的作用跟多值信号量的作用有些重叠，多值信号量的执行时间少于互斥信号量，但多个任务访问共享资源时，容易出现优先级反转的问题，这会降低系统的可预知性，而互斥信号量可以防止优先级反转，所以建议在互斥信号量可以解决需要时，就优先使用互斥信号量。

1.3.5 消息队列

消息队列是由多个消息串联而成的一个机制，需要消息的任务就从消息队列的出口端获

取，如果消息队列里没有消息了，可以选择等待或者不等待消息的到来。消息可以比信号量携带更丰富的信息，可以是任意长度的消息内容。

1.3.6 事件标志组

事件标志组用于标志若干个事件否发生的组合。这个功能可以轻易地实现键盘的按键组合。

1.3.7 任务信号量

任务信号量的作用与多值信号量的一样，但多值信号量是所有任务都可以申请使用，而任务信号量却只能给一个特定任务使用，也就是说任务信号量是一个任务本身的属性，但其他任务都可以给这个任务发送任务信号量。

1.3.8 任务消息队列

任务消息队列的作用与（普通）消息队列的一样，但（普通）消息队列是所有任务都可以申请它的消息，而任务任务消息队列的消息却只能给一个特定任务使用，也就是说任务消息队列是一个任务本身的属性，但其他任务都可以给这个任务发送任务消息。

1.3.9 内存管理（分区）

内存管理（分区）主要是为了尽量减少内存在不断分配和释放过程造成的内存碎片，避免过多的浪费内存。内存分区就是一次性开辟一大块连续内存，然后将内存分区平均分成若干个内存块，需要使用内存时就申请一个内存块，用完了再释放回内存分区，这样就实现内存块的循环使用。

1.4 uC/OS-III 常用程序段

1.4.1 临界段

临界段主要是为了某段代码在执行时避免被其它任务或中断打断。临界段根据是否是使能了中断延迟提交（OS_CFG_ISR_POST_DEFERRED_EN）大体可以分为两种，如下面代码所示。当使能中断延迟提交时，中断级任务会转换成任务级任务，这种情况下进入和退出临界段主要分别是锁调度器和解锁调度器。当禁用中断延迟提交时，进入和退出临界段的方式分别是关中断和开中断。

代码 1-2 临界段

```
01 #if OS_CFG_ISR_POST_DEFERRED_EN > 0u
02
```

```
#define  OS_CRITICAL_ENTER()                                        \
        do {                                                       \
           CPU_CRITICAL_ENTER();                                   \
           OSSchedLockNestingCtr++;                                \
           if (OSSchedLockNestingCtr == 1u) {                      \
              OS_SCHED_LOCK_TIME_MEAS_START();                     \
           }                                                       \
           CPU_CRITICAL_EXIT();                                    \
        } while (0)

#define  OS_CRITICAL_ENTER_CPU_EXIT()                               \
        do {                                                       \
           OSSchedLockNestingCtr++;                                \
                                                                   \
           if (OSSchedLockNestingCtr == 1u) {                      \
              OS_SCHED_LOCK_TIME_MEAS_START();                     \
           }                                                       \
           CPU_CRITICAL_EXIT();                                    \
        } while (0)


#define  OS_CRITICAL_EXIT()                                         \
        do {                                                       \
           CPU_CRITICAL_ENTER();                                   \
           OSSchedLockNestingCtr--;                                \
           if (OSSchedLockNestingCtr == (OS_NESTING_CTR)0) {       \
              OS_SCHED_LOCK_TIME_MEAS_STOP();                      \
              if (OSIntQNbrEntries > (OS_OBJ_QTY)0) {              \
                 CPU_CRITICAL_EXIT();                              \
                 OS_Sched0();                                      \
              } else {                                             \
                 CPU_CRITICAL_EXIT();                              \
              }                                                    \
           } else {                                                \
              CPU_CRITICAL_EXIT();                                 \
           }                                                       \
        } while (0)

#define  OS_CRITICAL_EXIT_NO_SCHED()                                \
        do {                                                       \
           CPU_CRITICAL_ENTER();                                   \
           OSSchedLockNestingCtr--;                                \
           if (OSSchedLockNestingCtr == (OS_NESTING_CTR)0) {       \
              OS_SCHED_LOCK_TIME_MEAS_STOP();                      \
           }                                                       \
           CPU_CRITICAL_EXIT();                                    \
        } while (0)


#else


#define  OS_CRITICAL_ENTER()                  CPU_CRITICAL_ENTER()

#define  OS_CRITICAL_ENTER_CPU_EXIT()

#define  OS_CRITICAL_EXIT()                   CPU_CRITICAL_EXIT()

#define  OS_CRITICAL_EXIT_NO_SCHED()           CPU_CRITICAL_EXIT()

#endif
```

其中，OS_CRITICAL_ENTER()和 OS_CRITICAL_ENTER_CPU_EXIT() 为进入临界段，OS_CRITICAL_EXIT() 和 OS_CRITICAL_EXIT_NO_SCHED() 为退出临界段，CPU_CRITICAL_ENTER()为关中断，CPU_CRITICAL_EXIT()为开中断。

1.4.2 中断嵌套管理

其中，CPU_CRITICAL_ENTER()为关中断，CPU_CRITICAL_EXIT()为开中断。为方便 uC/OS 对中断的管理，在进入中断服务函数时需要调用 OSIntEnter() 函数将中断嵌套计数 OSIntNestingCtr 加 1，并且在退出中断服务函数时需要调用 OSIntExit() 函数将中断嵌套计数 OSIntNestingCtr 减 1。

代码 1-3 OSIntEnter() 函数

```
01 void  OSIntEnter (void)
02 {
03    if (OSRunning != OS_STATE_OS_RUNNING) {
04       return;
05    }
06
07    if (OSIntNestingCtr >= (OS_NESTING_CTR)250u) {
08       return;
09    }
10
11    OSIntNestingCtr++;
12 }
```

代码 1-4 OSIntExit() 函数

```
01 void  OSIntExit (void)
02 {
03    CPU_SR_ALLOC();
04
05
06
07    if (OSRunning != OS_STATE_OS_RUNNING) {
08       return;
09    }
10
11    CPU_INT_DIS();
12    if (OSIntNestingCtr == (OS_NESTING_CTR)0) {
13       CPU_INT_EN();
14       return;
15    }
16    OSIntNestingCtr--;
17    if (OSIntNestingCtr > (OS_NESTING_CTR)0) {
18       CPU_INT_EN();
19       return;
20    }
21
22    if (OSSchedLockNestingCtr > (OS_NESTING_CTR)0) {
23       CPU_INT_EN();
24       return;
25    }
26
27    OSPrioHighRdy   = OS_PrioGetHighest();
28    OSTCBHighRdyPtr = OSRdyList[OSPrioHighRdy].HeadPtr;
29    if (OSTCBHighRdyPtr == OSTCBCurPtr) {
30       CPU_INT_EN();
31       return;
32    }
33
34 #if OS_CFG_TASK_PROFILE_EN > 0u
35    OSTCBHighRdyPtr->CtxSwCtr++;
36 #endif
37    OSTaskCtxSwCtr++;
38
39 #if defined(OS_CFG_TLS_TBL_SIZE) && (OS_CFG_TLS_TBL_SIZE > 0u)
```

```
40      OS_TLS_TaskSw();
41  #endif
42
43      OSIntCtxSw();
44      CPU_INT_EN();
45  }
```

1.5 章末总结

本章主要对 uC/OS-III 实时操作系统做一个整体的概要认识。uC/OS-III 系统的文件结构非常清晰，从底层到上层，把底层驱动和上层应用逐层分开，很方便系统的移植和应用开发。uC/OS-III 系统的内核对象大多是采用结构体，对内核对象的管理大多采用单向链表或双向链表的数据结构。uC/OS-III 系统的常用内核对象有任务、软件定时器、多值信号量、互斥信号量、消息队列、事件标志组、任务信号量、任务消息队列和内存管理，这也是本书的重点讲解对象，后面章节会进行详细阐述。用户不希望被任务或中断打断的程序段成为临界段，在进入和退出临界段时分别需要调用进入和退出临界段函数。为方便中断嵌套管理，进入和退出中断服务函数时分别需要调用 OSIntEnter()和 OSIntExit() 函数。

第2章 移植 uC/OS-III 到 STM32

鉴于后面要在秉火 STM32 开发板上演示 uC/OS-III 功能，所以本章先把 uC/OS-III 移植到秉火的 STM32 开发板上。

2.1 下载官方 uC/OS-III 源码

首先，打开 Micrium 公司官方网站（http://micrium.com/），点击“Downloads”选项卡进入下载页面，在“Brouse by MCU Manufacturer”栏目展开“STMicroelectronics”，单击“View all STMicroelectronics”。

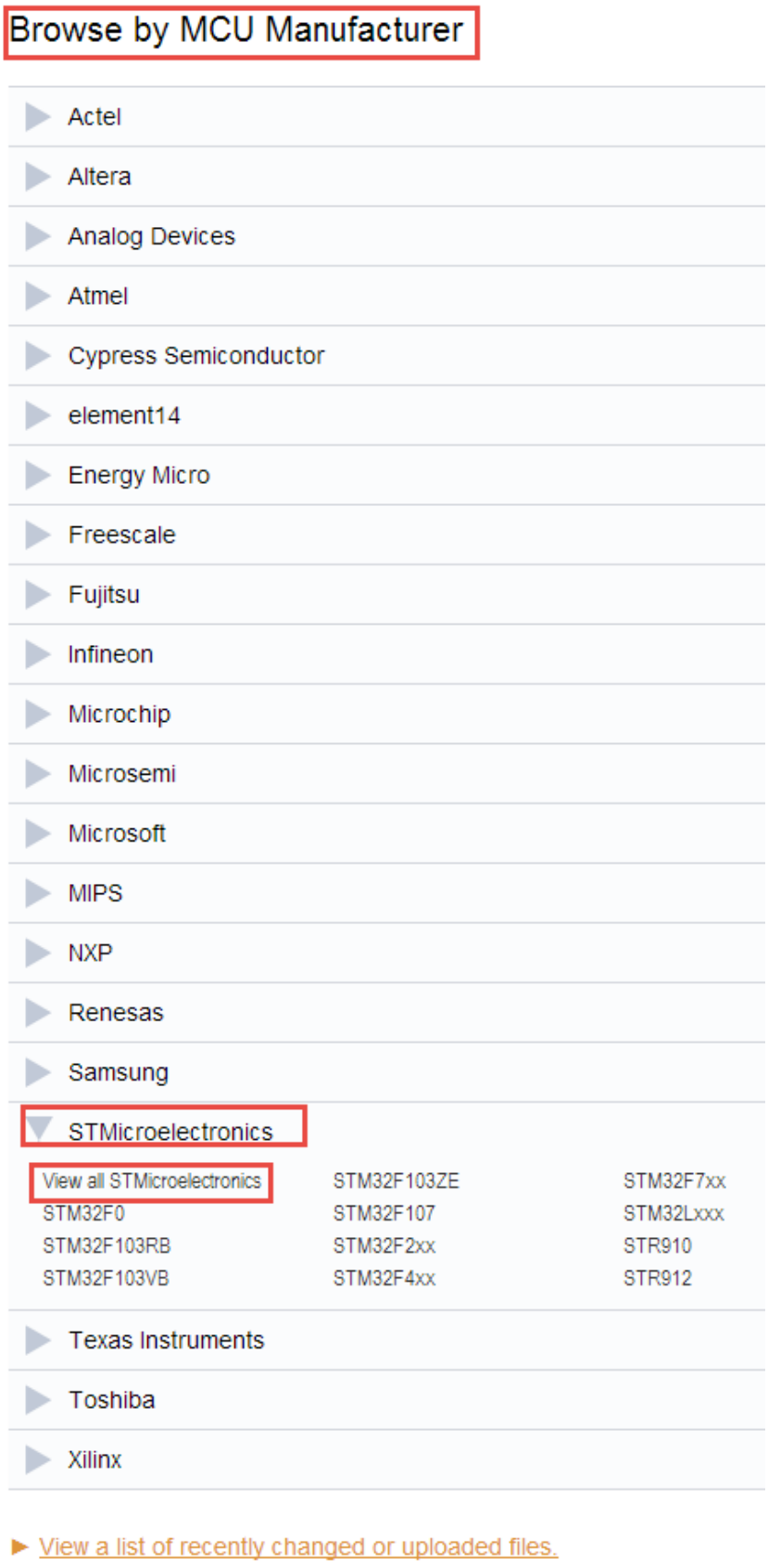
Browse by MCU Manufacturer
Actel
Altera
Analog Devices
Atmel
Cypress Semiconductor
element14
Energy Micro
Freescale
Fujitsu
Infineon
Microchip
Microsemi
Microsoft
MIPS
NXP
Renesas
Samsung
STMicroelectronics
View all STMicroelectronics
STM32F0
STM32F103RB
STM32F103VB
STM32F103ZE
STM32F107
STM32F2xx
STM32F4xx
STM32F7xx
STM32Lxxx
STR910
STR912
Texas Instruments
Toshiba
Xilinx
View a list of recently changed or uploaded files.

图 2-1

由于在“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 uC/OS-III 源码，单击即可。

Projects

MCU	Micrium Product	Evaluation Board	Toolchain	Date
STMicroelectronics STM32F0	µC/OS-III µC/OS-III V3.03.01	STM320518-EVAL	Atollic TrueSTUDIO V3.x IAR (EWARM) V6.x Keil MDK V4.x	2013/02/05
STMicroelectronics STM32F0	µC/OS-II µC/OS-II V2.92	STM320518-EVAL	Atollic TrueSTUDIO V3.x IAR (EWARM) V6.x Keil MDK V4.x	2012/11/14
STMicroelectronics STM32F103RB STM32F103VB	µC/OS-II µC/OS-II V2.86	IAR STM32-SK STM3210B-EVAL	IAR (EWARM) V5.x	2012/12/05
STMicroelectronics STM32F103ZE	µC/OS-II µC/OS-II V2.86	IAR STM32F103ZE-SK	IAR (EWARM) V5.x	2012/12/05
STMicroelectronics STM32F107	µC/TCP-IP µC/OS-III V3.04.04 µC/TCPIP V3.01.00 Library Project	Micrium uC-Eval-STM32F107	IAR (EWARM) V7.x	2014/08/15
STMicroelectronics STM32F107	µC/USB Device µC/OS-III V3.03.01 µC/USBD V4.01.01 Library Project	Micrium uC-Eval-STM32F107	IAR (EWARM) V6.x	2013/04/05
STMicroelectronics STM32F107	µC/TCP-IP µC/OS-III V3.03.01 µC/TCPIP V2.13.02 Library Project	Micrium uC-Eval-STM32F107	IAR (EWARM) V6.x	2013/04/05
STMicroelectronics STM32F107	µC/FS µC/FS V4.05.03 µC/OS-III V3.03.01 Library Project	Micrium uC-Eval-STM32F107	IAR (EWARM) V6.x	2013/04/02
STMicroelectronics STM32F107	µC/OS-III µC/OS-III V3.03.01	Micrium uC-Eval-STM32F107	Atollic TrueSTUDIO V3.x IAR (EWARM) V6.x Keil MDK V3.x	2013/02/08
STMicroelectronics STM32F107	µC/OS-II µC/OS-II V2.92.07	Micrium uC-Eval-STM32F107	Atollic TrueSTUDIO V3.x IAR (EWARM) V6.x Keil MDK V4.x	2013/02/08

图 2-2

页面跳转后，点击“Download”按钮即可下载，下载前要登录该网站，未账户的注册后登录。

Download “Micrium_uC-Eval-STM32F107_uCOS-III”

Download

File Type	**Project**
Processor	**STMicroelectronics STM32F107**
Micrium Product	**µC/OS-III** µC/OS-III V3.03.01
Version	**3.03.01**
Updated	**February 8, 2013**

图 2-3

秉火已经将此源码下载并附带于本书配套资料里，方便用户直接使用。

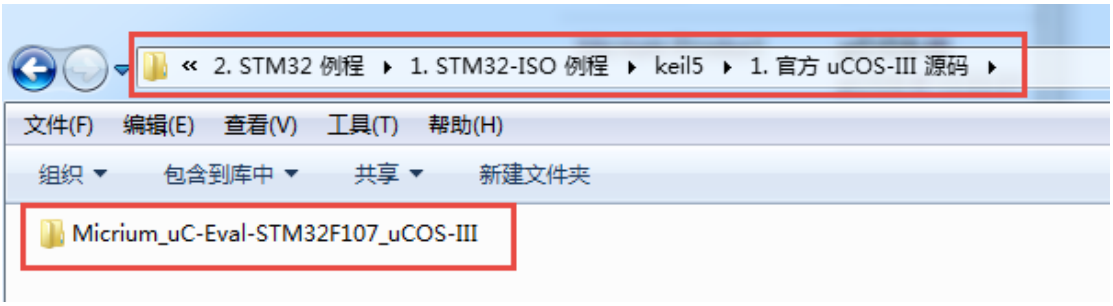

图 2-4

2.2 移植过程

选择一个秉火 STM32 开发板可用的裸机例程作为程序模板，这里准备了一个简单的“LED 流水灯”例程，该例程存放在本书配套资料里。

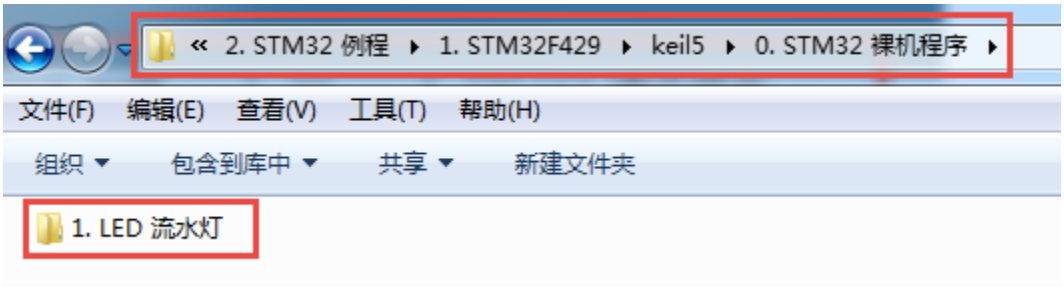

图 2-5

在该例程的“User”文件夹下建立下面几个文件夹。

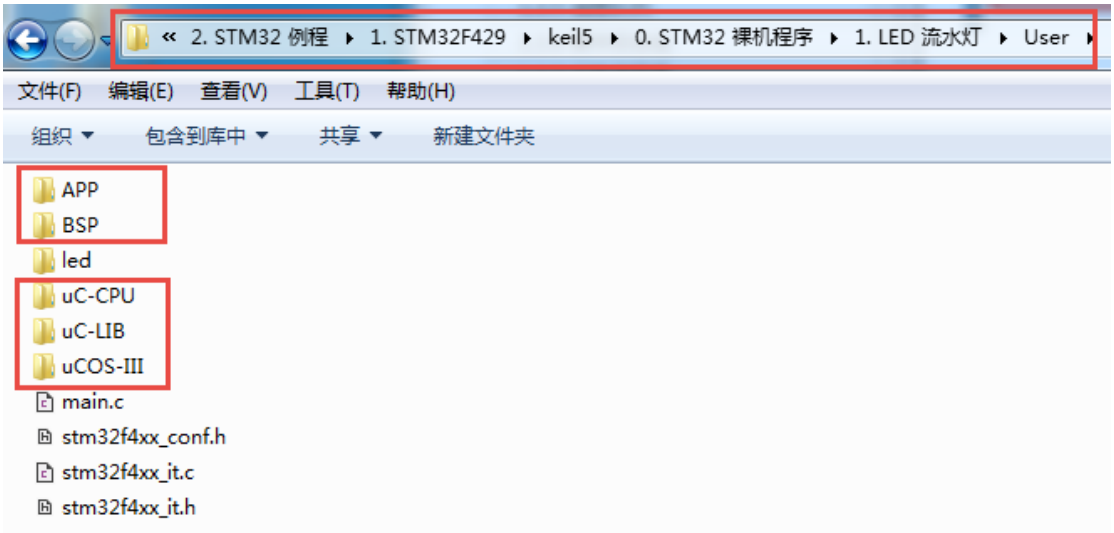

图 2-6

拷贝下图路径下 uC/OS-III 源码文件到“APP”文件夹下。

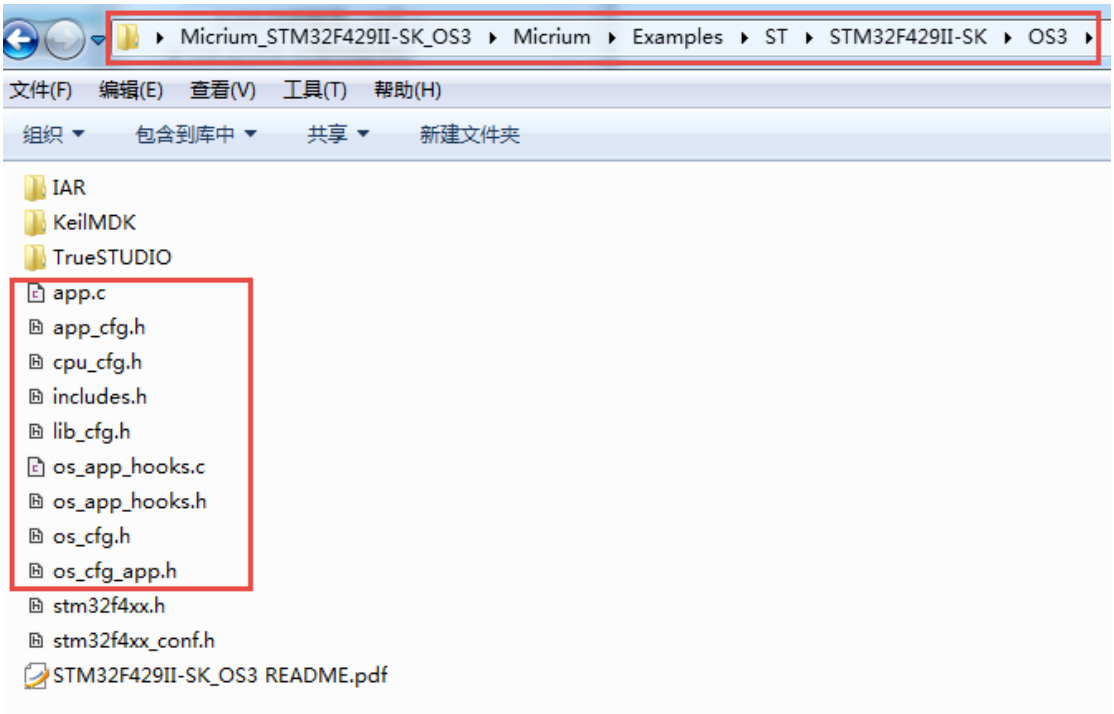

图 2-7

图 2-8

拷贝下图路径下 uC/OS-III 源码文件到“BSP”文件夹下。

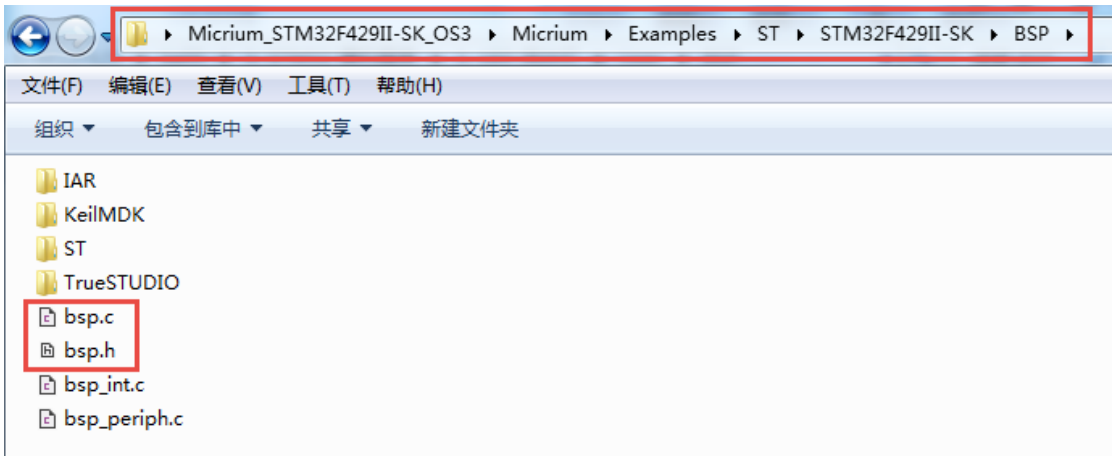

图 2-9

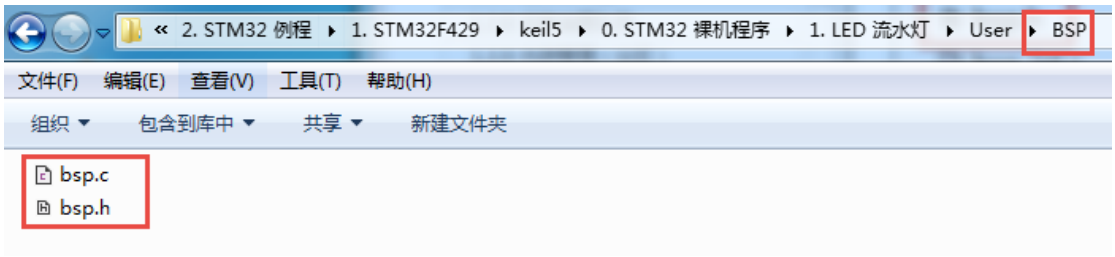

图 2-10

拷贝下图路径下 uC/OS-III 源码文件到“uC-CPU”文件夹下。

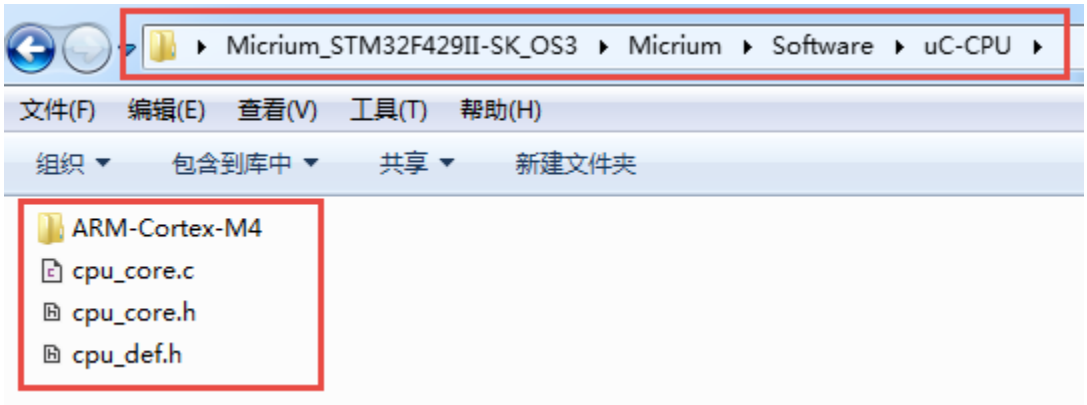

图 2-11

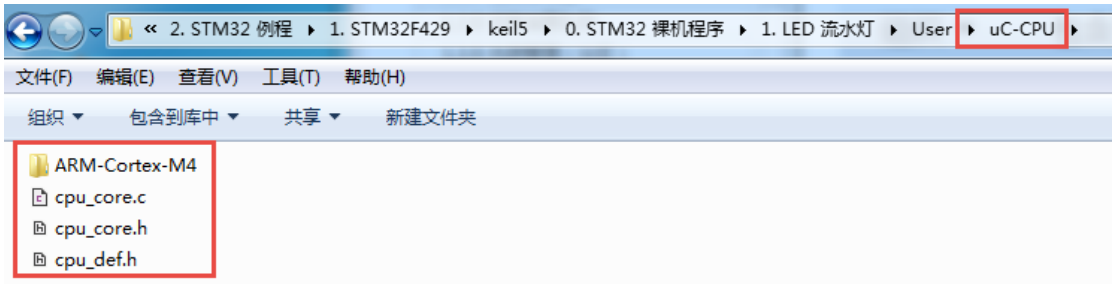

图 2-12

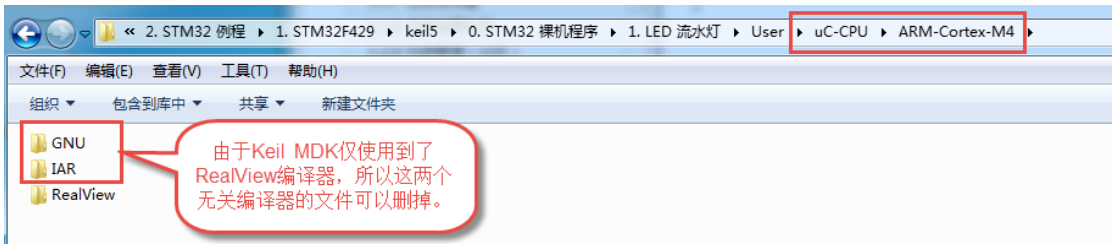

图 2-13

拷贝下图路径下 uC/OS-III 源码文件到“uC-LIB”文件夹下。

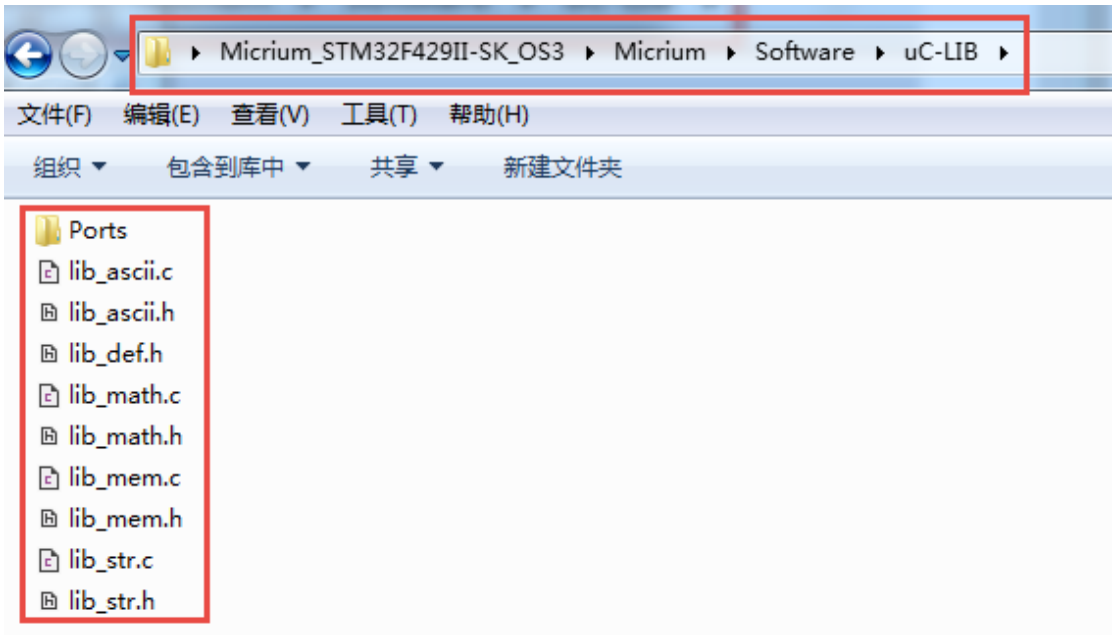

图 2-14

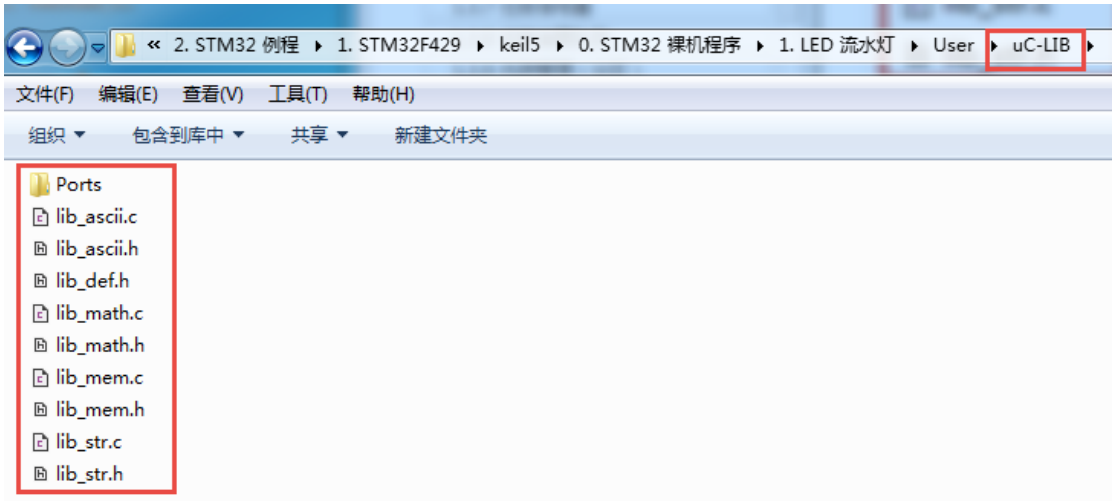

图 2-15

图 2-16

拷贝下图路径下 uC/OS-III 源码文件到“uCOS-III”文件夹下。

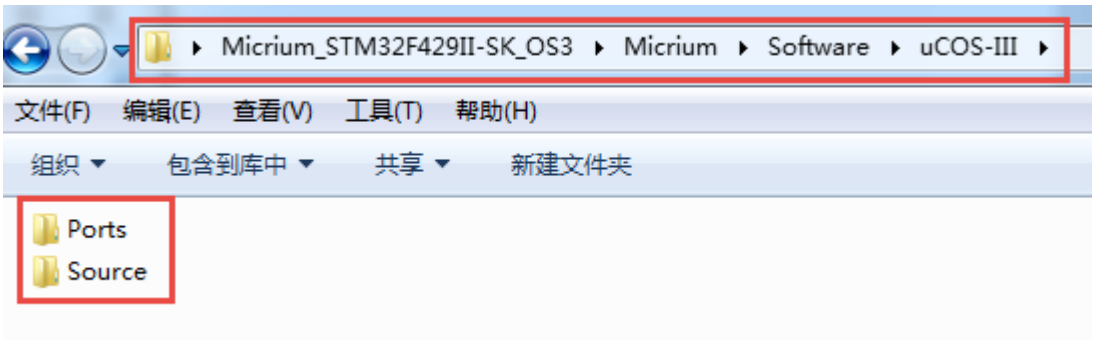

图 2-17

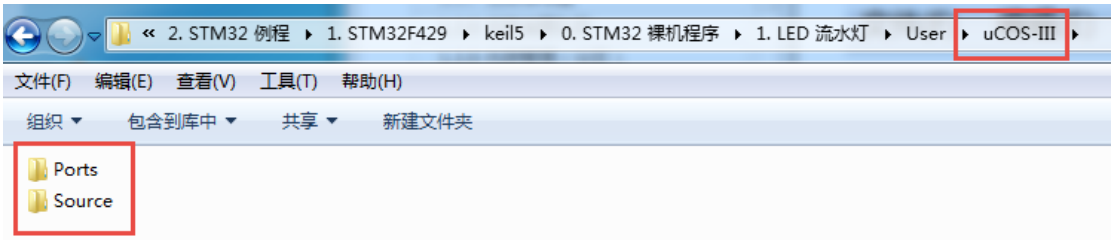

图 2-18

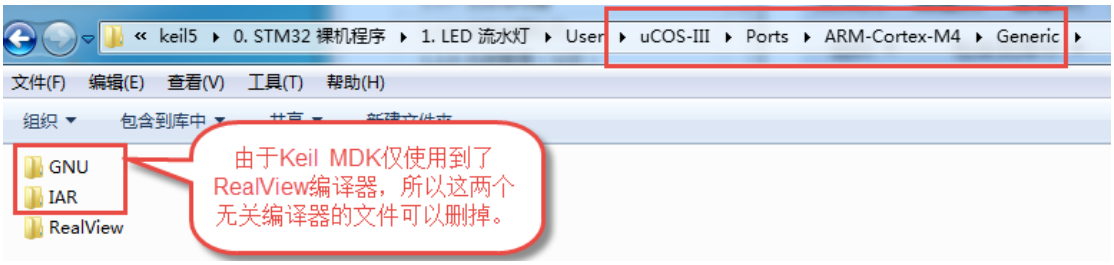

图 2-19

将“User”文件夹下的“led”文件夹剪切到“BSP”文件夹里面，并且删除“main.c”文件。

图 2-20

图 2-21

打开工程，首先移除废弃的文件。

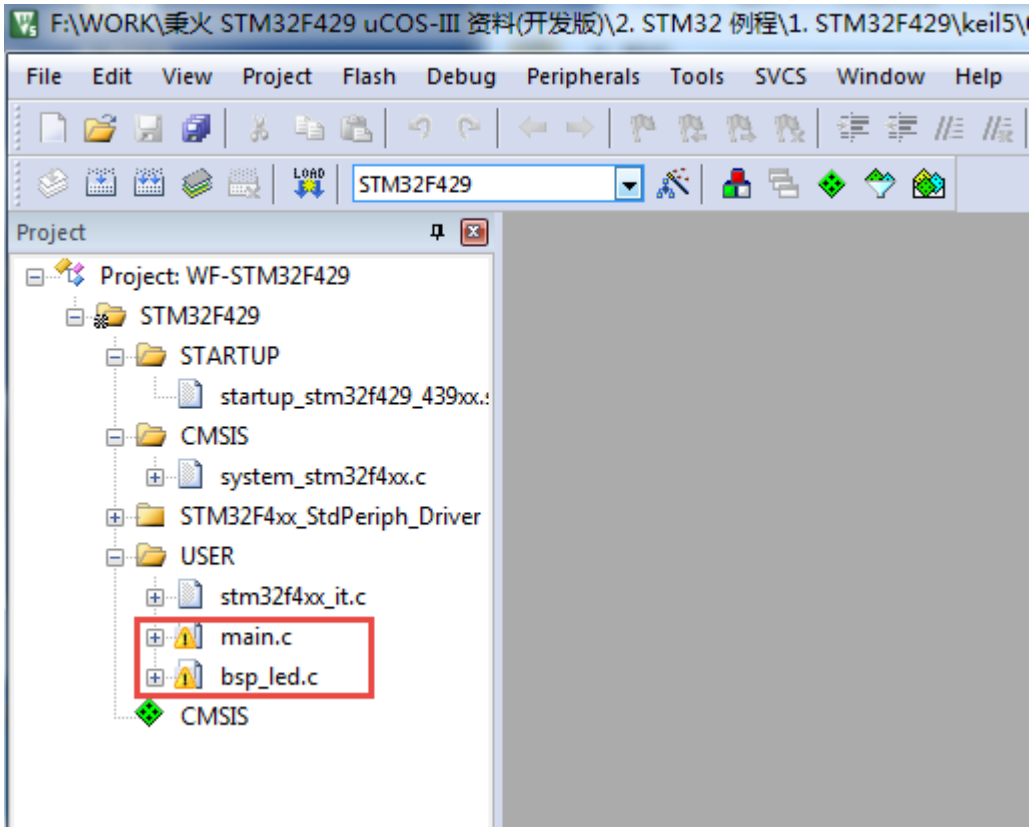

图 2-22

给工程增加下面的组件。

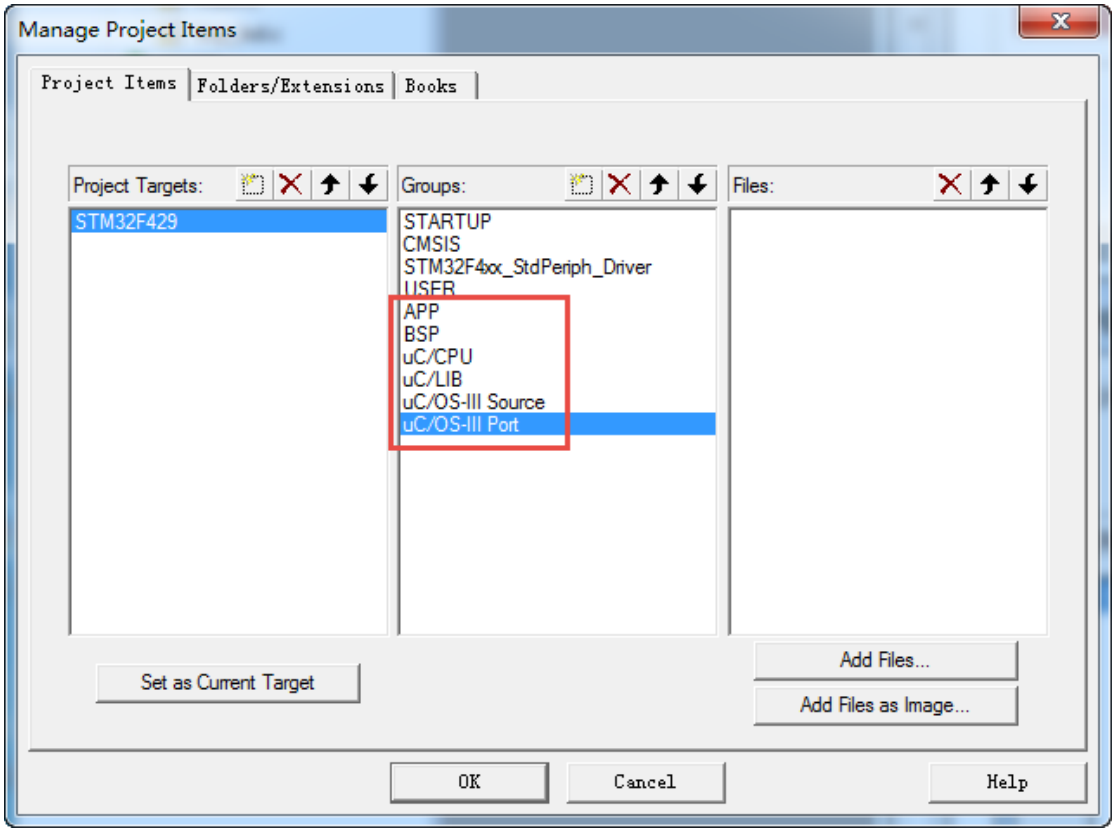

图 2-23

为“APP”组件添加“\User\APP”文件夹下的所有文件。

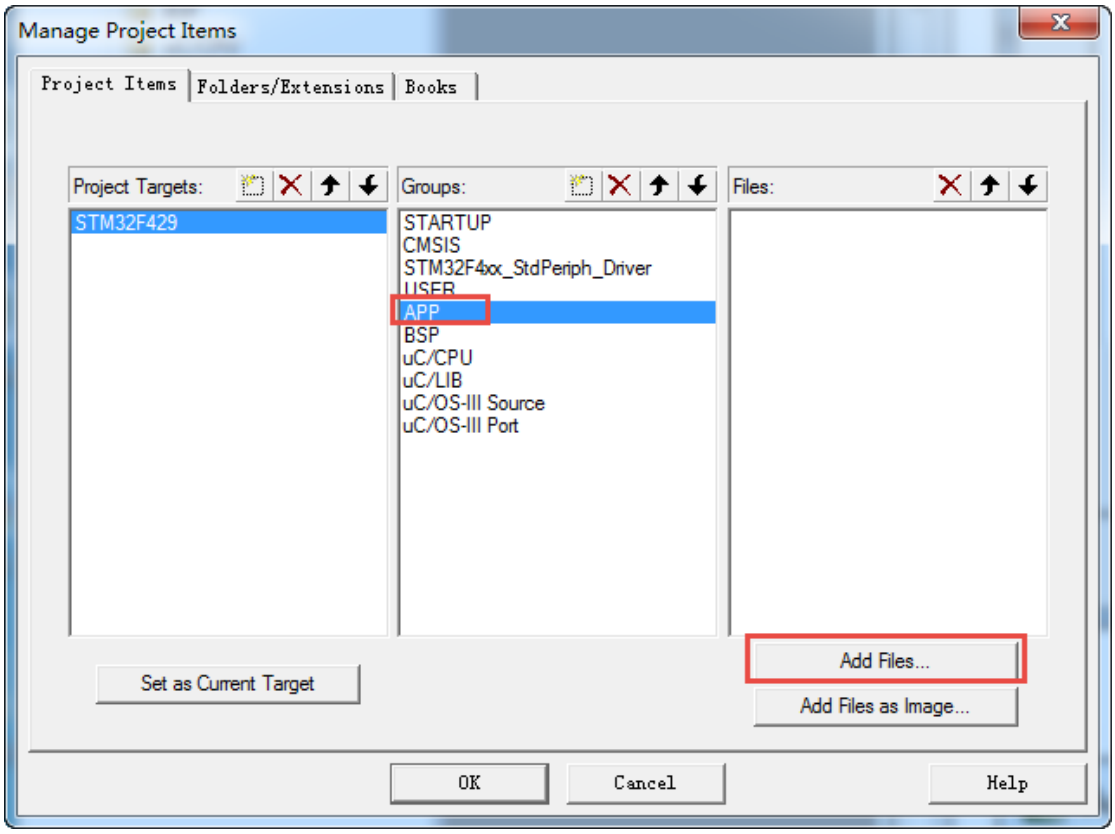
Manage Project Items
Project Items
Folders/Extensions
Books
Project Targets:
STM32F429
Groups:
STARTUP
CMSIS
STM32F4xx_StdPeriph_Driver
USER
APP
BSP
uC/CPU
uC/LIB
uC/OS-III Source
uC/OS-III Port
Files:
Set as Current Target
Add Files...
Add Files as Image...
OK
Cancel
Help

图 2-24

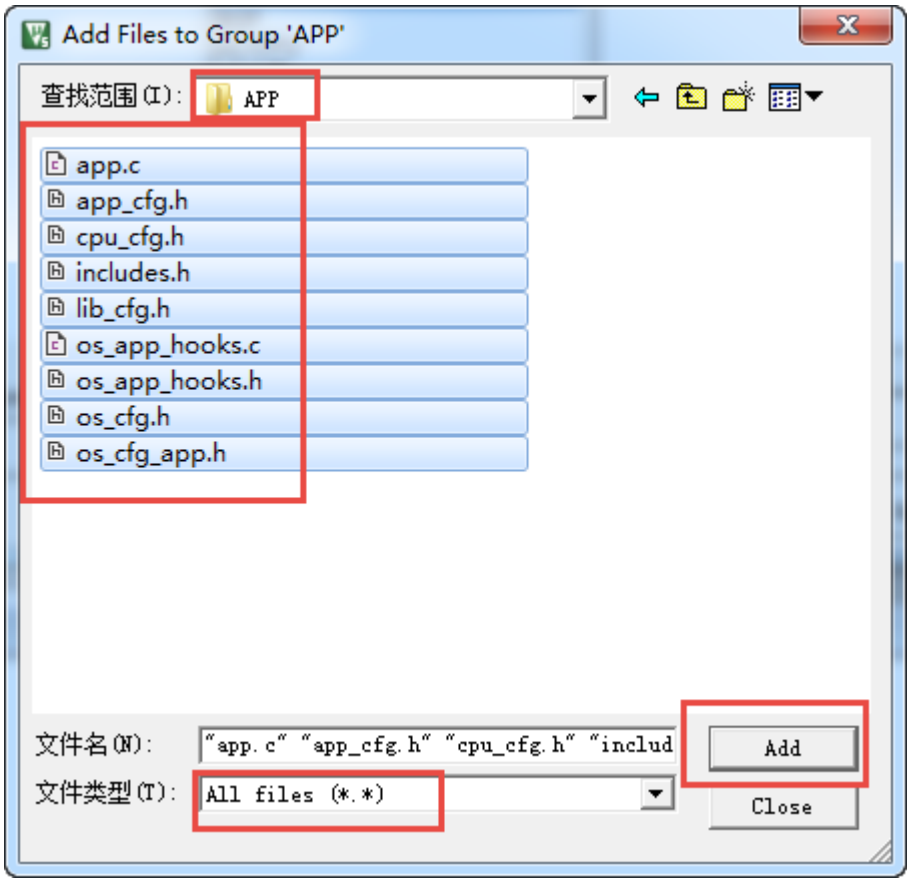
Add Files to Group 'APP'
查找范围(I):
APP
app.c
app_cfg.h
cpu_cfg.h
includes.h
lib_cfg.h
os_app_hooks.c
os_app_hooks.h
os_cfg.h
os_cfg_app.h
文件名(N):
"app.c" "app_cfg.h" "cpu_cfg.h" "includ
文件类型(T):
All files (*.*)
Add
Close

图 2-25

为“APP”组件添加“\User\BSP”文件夹下的所有文件和“\User\BSP\led”文件夹下的源文件。

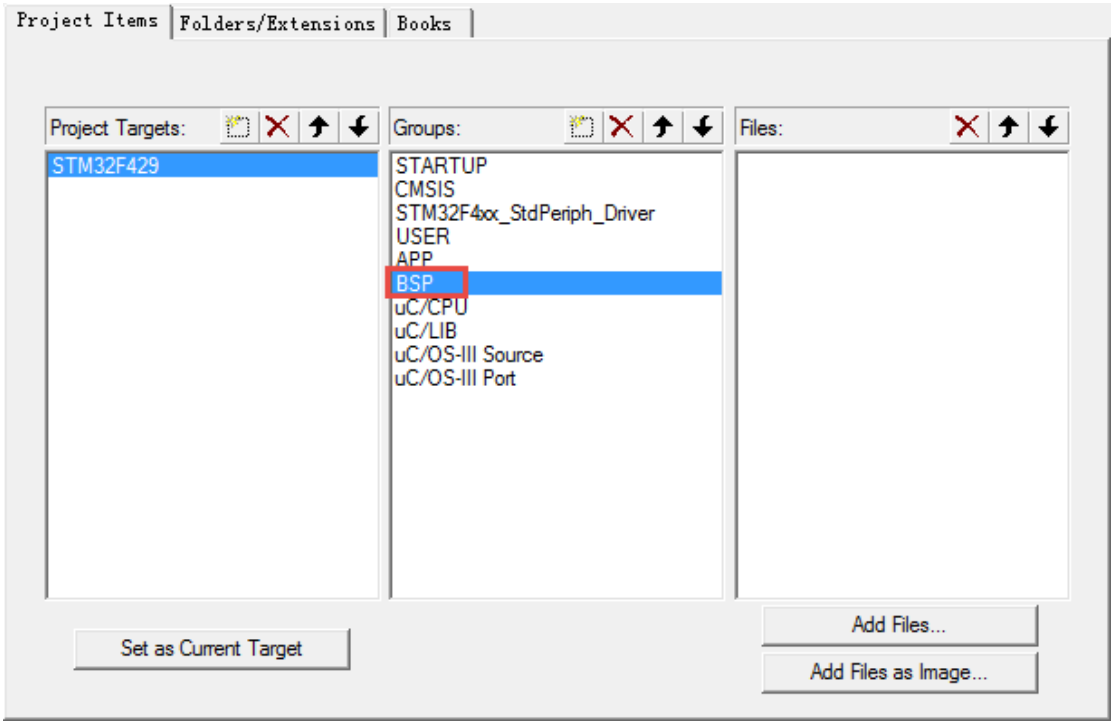

图 2-26

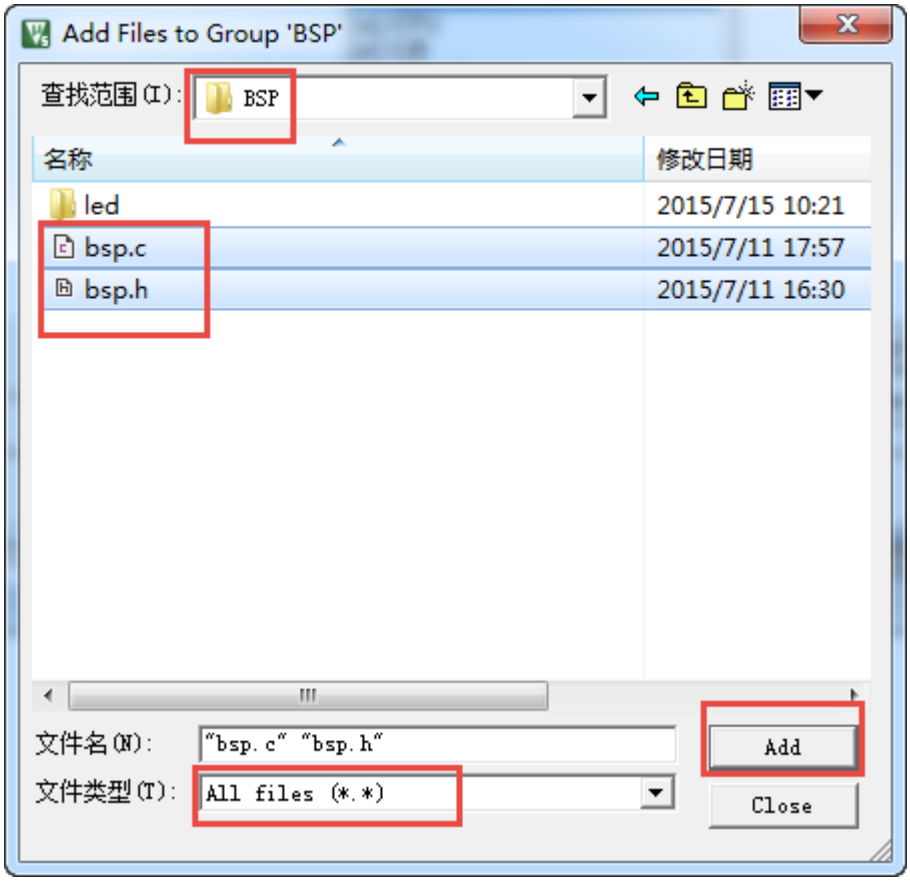

图 2-27

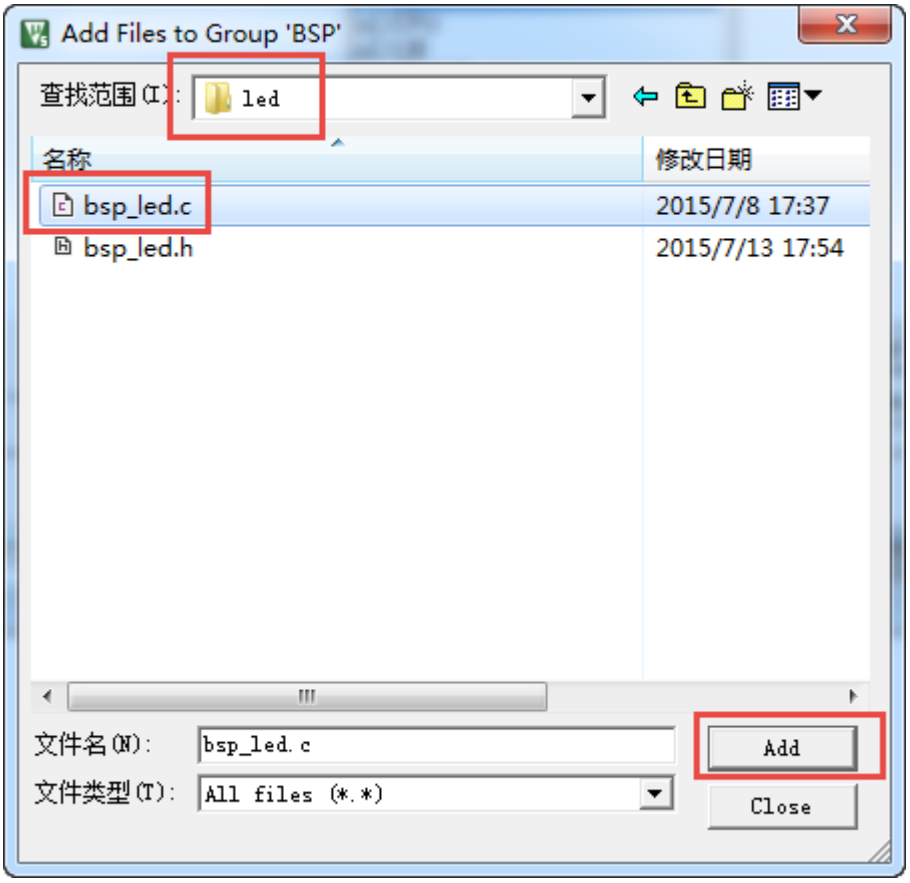

图 2-28

为“uC/CPU”组件添加“\User\ uC-CPU”文件夹下的所有文件和“\User\ uC-CPU\ ARM-Cortex-M3\ RealView”文件夹下的所有文件。

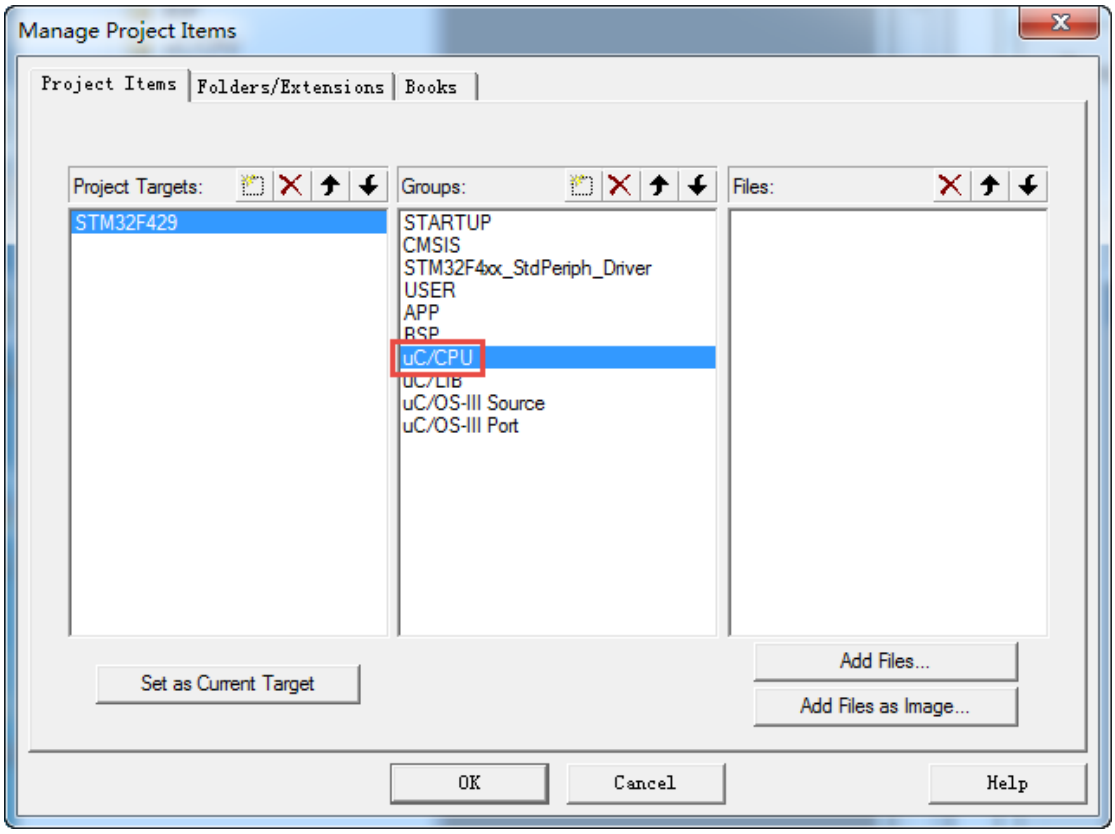
Manage Project Items
Project Items
Folders/Extensions
Books
Project Targets:
STM32F429
Groups:
STARTUP
CMSIS
STM32F4xx_StdPeriph_Driver
USER
APP
BSP
uC/CPU
uC/LIB
uC/OS-III Source
uC/OS-III Port
Files:
Set as Current Target
Add Files...
Add Files as Image...
OK
Cancel
Help

图 2-29

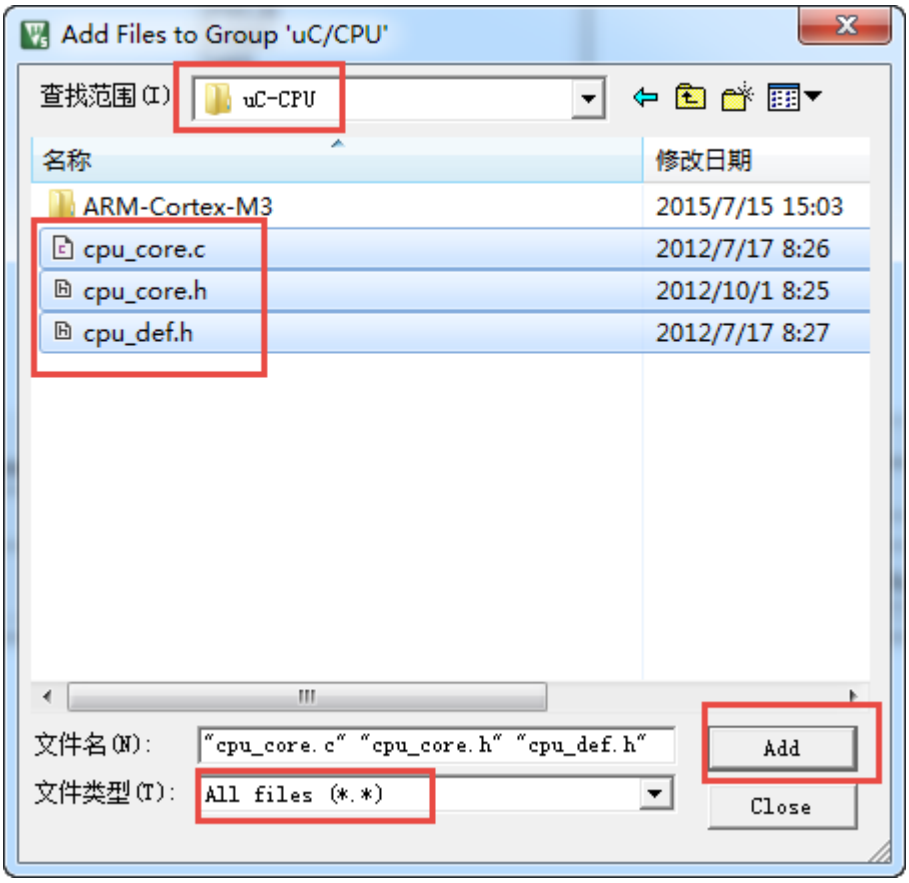
Add Files to Group 'uC/CPU'
查找范围(I):
uC-CPU
名称
修改日期
ARM-Cortex-M3
2015/7/15 15:03
cpu_core.c
2012/7/17 8:26
cpu_core.h
2012/10/1 8:25
cpu_def.h
2012/7/17 8:27
文件名(N):
"cpu_core.c" "cpu_core.h" "cpu_def.h"
文件类型(T):
All files (*.*)
Add
Close

图 2-30

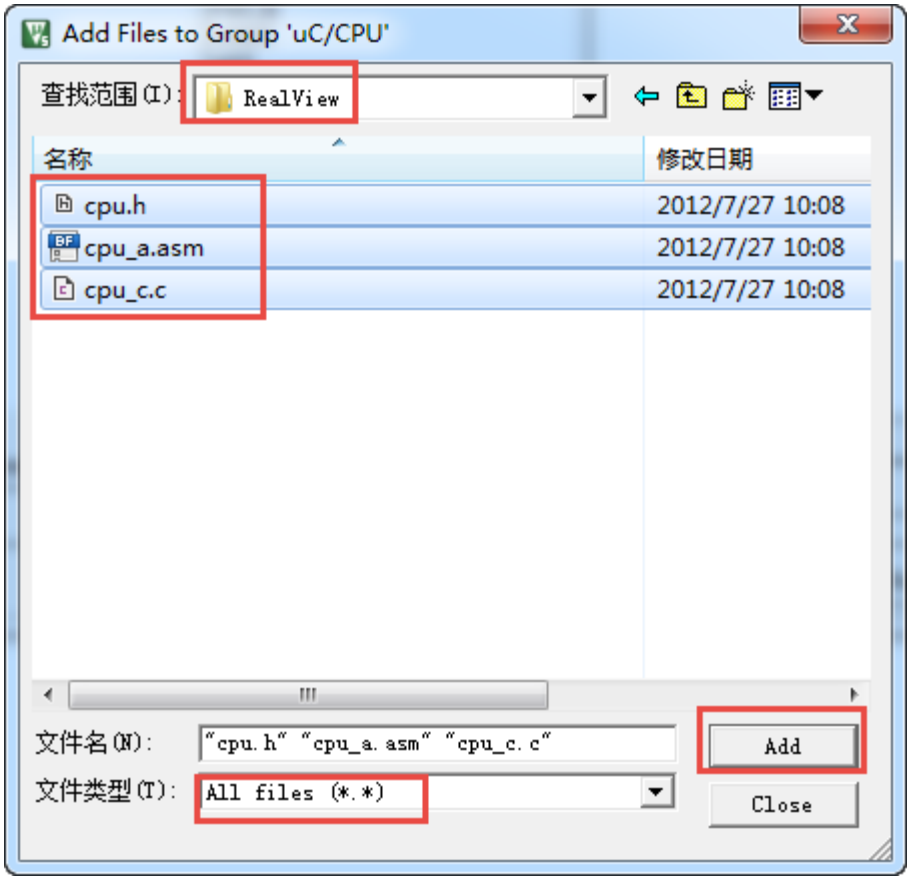

图 2-31

为“uC/LIB”组件添加“\User\ uC-LIB”文件夹下的所有文件和“\User\ uC-LIB\Ports\ARM-Cortex-M3\ RealView”文件夹下的所有文件。

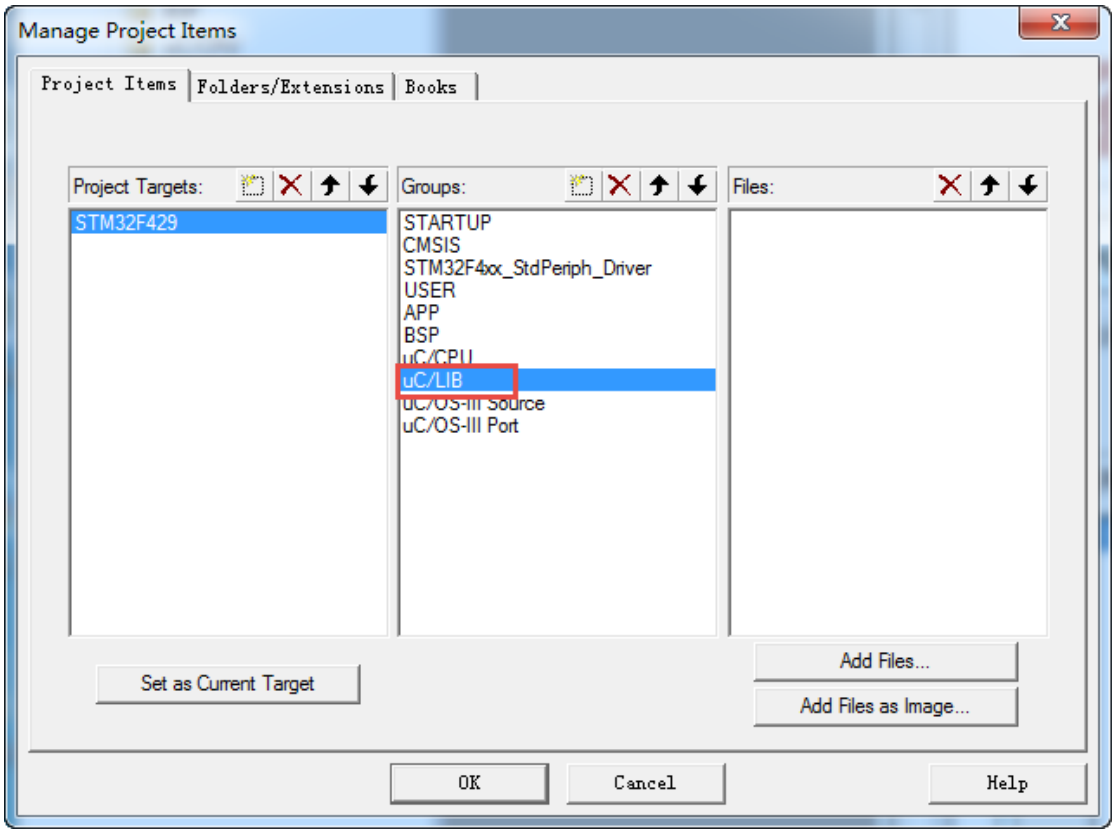
Manage Project Items
Project Items
Folders/Extensions
Books
Project Targets:
STM32F429
Groups:
STARTUP
CMSIS
STM32F4xx_StdPeriph_Driver
USER
APP
BSP
uC/CPU
uC/LIB
uC/OS-III Source
uC/OS-III Port
Files:
Set as Current Target
Add Files...
Add Files as Image...
OK
Cancel
Help

图 2-32

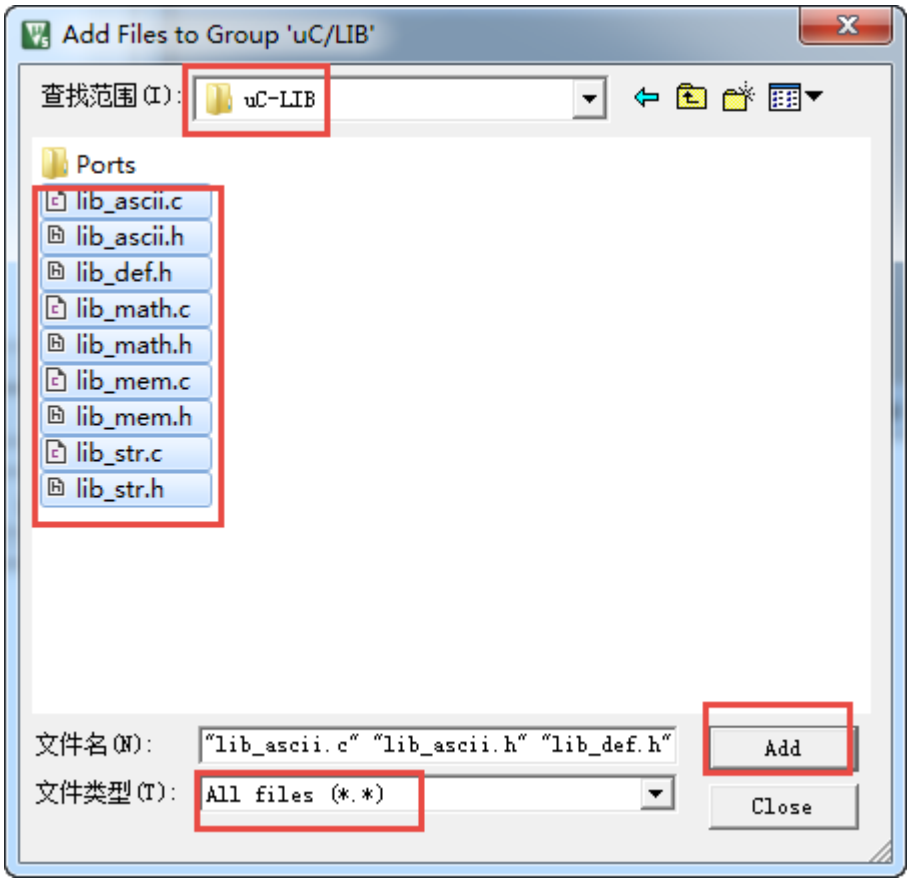
Add Files to Group 'uC/LIB'
查找范围(I):
uC-LIB
Ports
lib_ascii.c
lib_ascii.h
lib_def.h
lib_math.c
lib_math.h
lib_mem.c
lib_mem.h
lib_str.c
lib_str.h
文件名(N):
"lib_ascii.c" "lib_ascii.h" "lib_def.h"
文件类型(T):
All files (*.*)
Add
Close

图 2-33

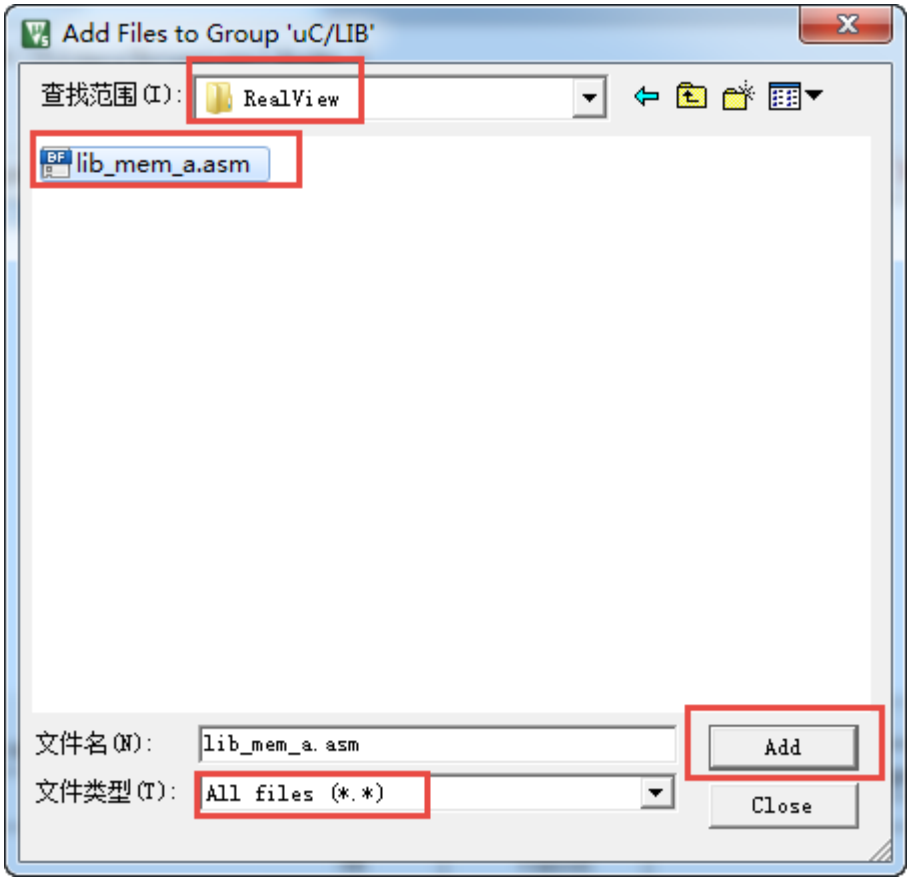

图 2-34

为“uC/OS-III Source”组件添加“\User\ uCOS-III\ Source”文件夹下的所有文件。

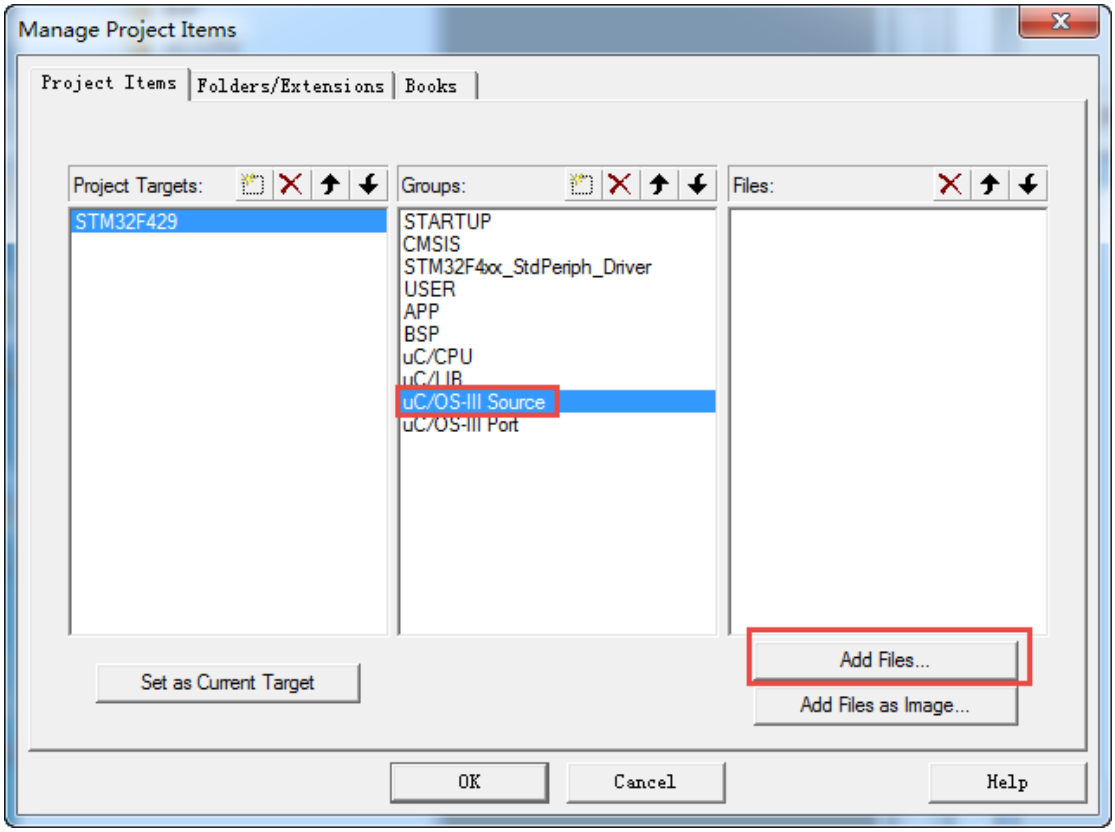

图 2-35

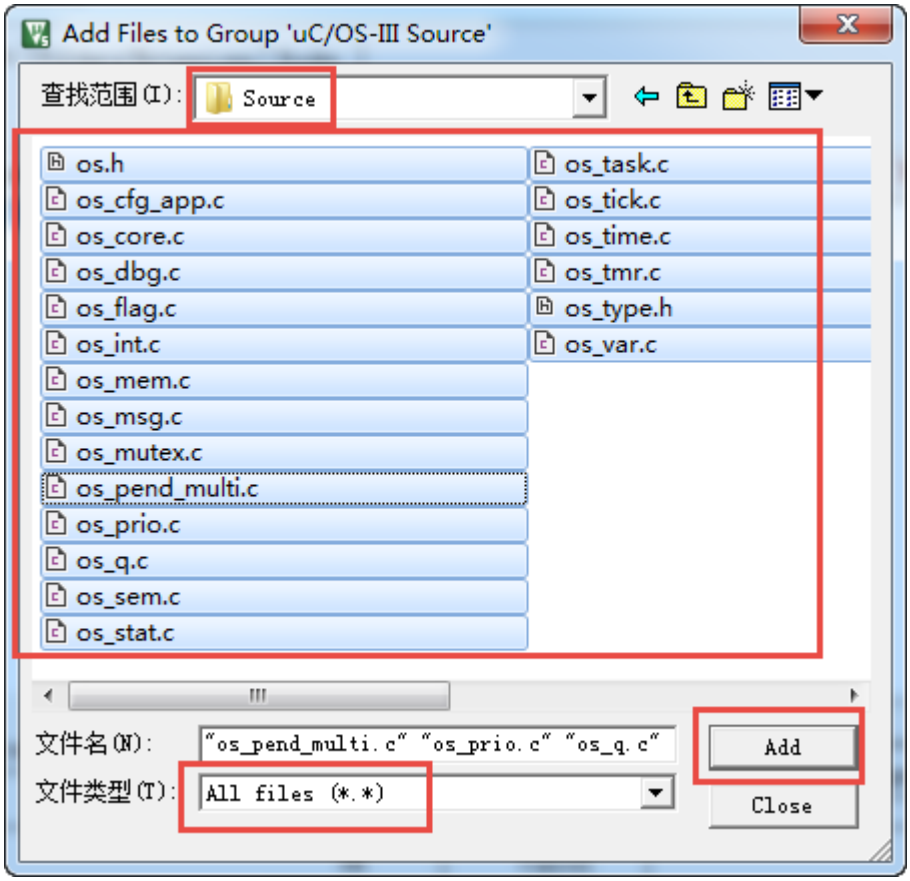

图 2-36

为“uC/OS-III Port”组件添加“\User\uCOS-III\Ports\ARM-Cortex-M3\Generic\RealView”文件夹下的所有文件。

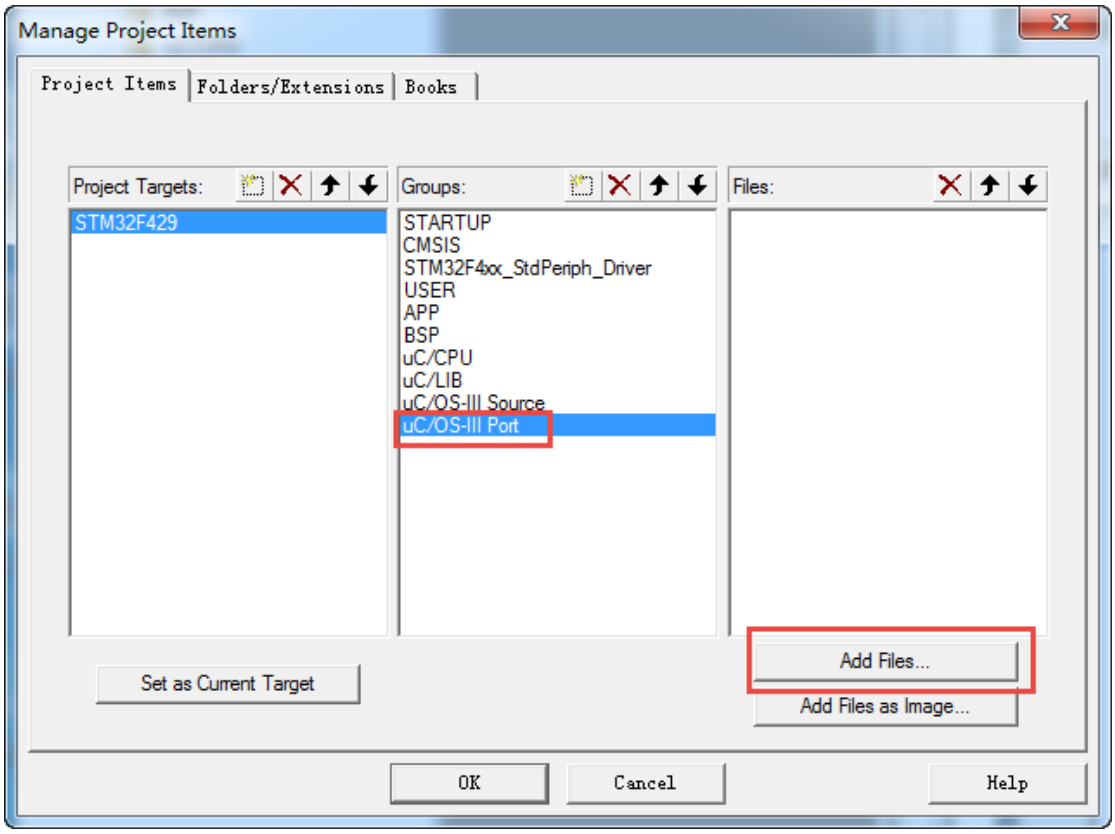
Manage Project Items
Project Items
Folders/Extensions
Books
Project Targets:
STM32F429
Groups:
STARTUP
CMSIS
STM32F4xx_StdPeriph_Driver
USER
APP
BSP
uC/CPU
uC/LIB
uC/OS-III Source
uC/OS-III Port
Files:
Add Files...
Add Files as Image...
Set as Current Target
OK
Cancel
Help

图 2-37

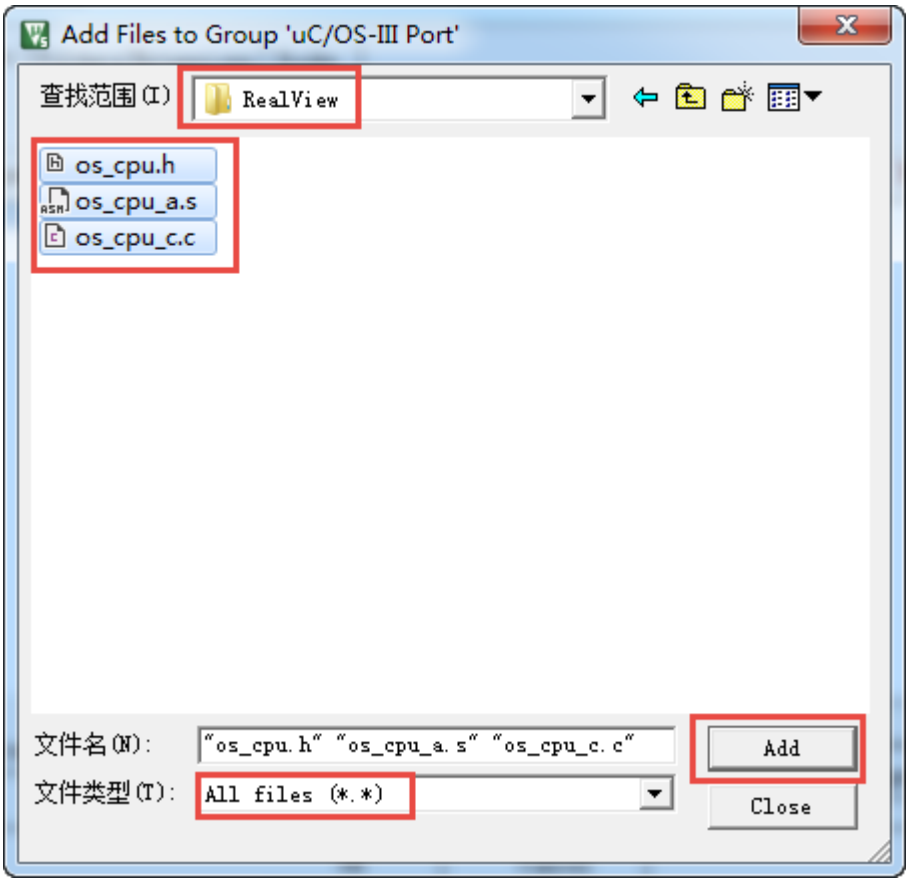
Add Files to Group 'uC/OS-III Port'
查找范围(I)
RealView
os_cpu.h
os_cpu_a.s
os_cpu_c.c
文件名(N):
"os_cpu.h" "os_cpu_a.s" "os_cpu_c.c"
文件类型(T):
All files (*.*)
Add
Close

图 2-38

打开“Options for Target”窗口，给工程添加包含路径。

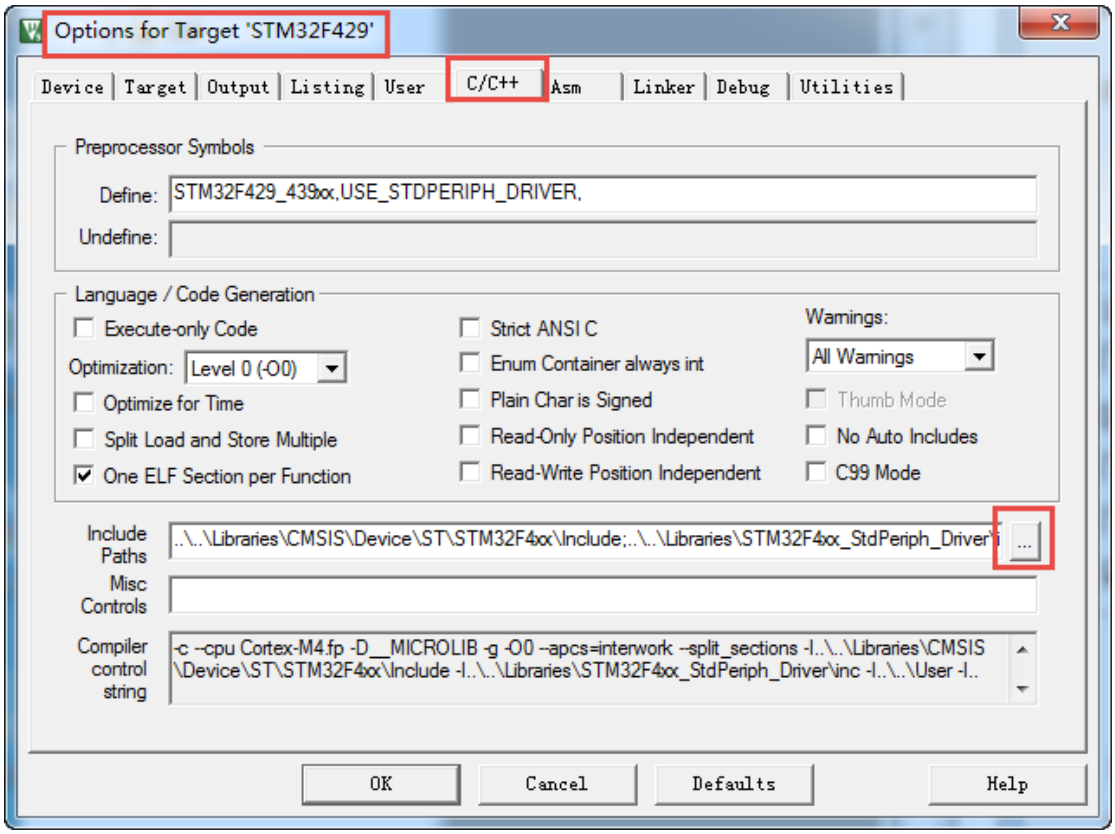

图 2-39

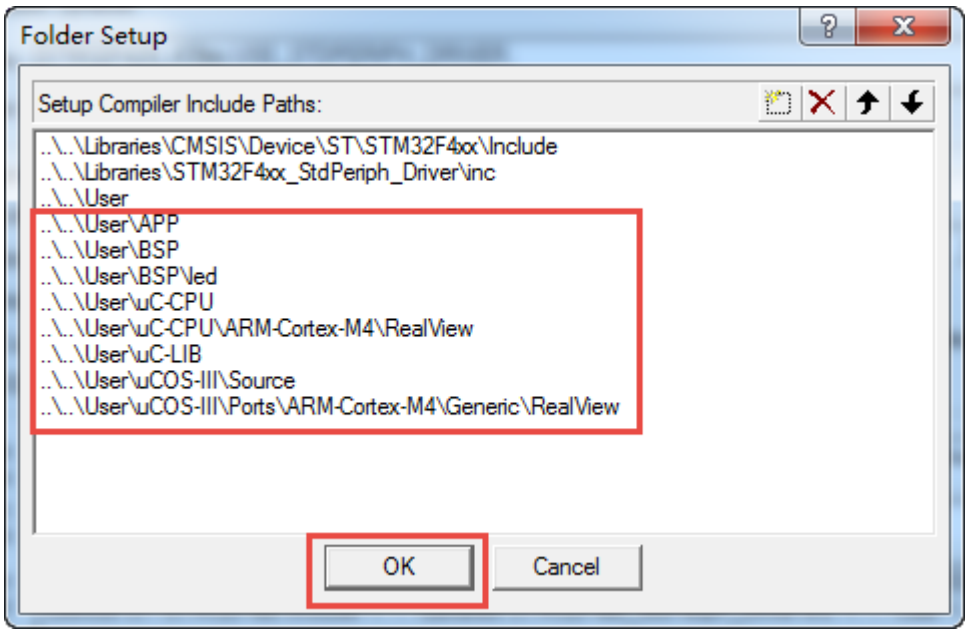

图 2-40

这时可以编译一下整个工程，但肯定会有错误的，uC/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_stm32f429_439xx.s”。其中将 PendSV_Handler 和 SysTick_Handler 分别改为 OS_CPU_PendSVHandler 和

OS_CPU_SysTickHandler，共两处。还有在复位时使能浮点支持。

修改一：

startup_stm32f429_439xx.s | includes.h | app.c

```
76                DCD     0                       ; Reserved
77                DCD     0                       ; Reserved
78                DCD     0                       ; Reserved
79                DCD     0                       ; Reserved
80                DCD     SVC_Handler             ; SVCall Handler
81                DCD     DebugMon_Handler        ; Debug Monitor Handler
82                DCD     0                       ; Reserved
83                DCD     PendSV_Handler          ; PendSV Handler
84                DCD     SysTick_Handler         ; SysTick Handler
85
```

修改前

图 2-41

startup_stm32f429_439xx.s | includes.h | app.c

```
76                DCD     0                       ; Reserved
77                DCD     0                       ; Reserved
78                DCD     0                       ; Reserved
79                DCD     0                       ; Reserved
80                DCD     SVC_Handler             ; SVCall Handler
81                DCD     DebugMon_Handler        ; Debug Monitor Handler
82                DCD     0                       ; Reserved
83                DCD     OS_CPU_PendSVHandler        ; PendSV Handler
84                DCD     OS_CPU_SysTickHandler       ; SysTick Handler
85
```

修改后

图 2-42

修改二：

bsp.c | app_cfg.h | app.c | bsp.h | cpu_cfg.h | startup_stm32f429_439xx.s | includes.h

```
183                 AREA    |.text|, CODE, READONLY
184
185 ; Reset handler
186 Reset_Handler    PROC
187                 EXPORT  Reset_Handler             [WEAK]
188         IMPORT  SystemInit
189         IMPORT  __main
190
191                 IF {FPU} != "SoftVFP"
192                                                   ; Enable Floating Point Support at reset for FPU
193                 LDR.W   R0, =0xE000ED88           ; Load address of CPACR register
194                 LDR     R1, [R0]                  ; Read value at CPACR
195                 ORR     R1,  R1, #(0xF <<20)      ; Set bits 20-23 to enable CP10 and CP11 coprocessors
196                                                   ; Write back the modified CPACR value
197                 STR     R1, [R0]                  ; Wait for store to complete
198                 DSB
199
200                                                   ; Disable automatic FP register content
201                                                   ; Disable lazy context switch
202                 LDR.W   R0, =0xE000EF34           ; Load address to FPCCR register
203                 LDR     R1, [R0]
204                 AND     R1,  R1, #(0x3FFFFFFF)    ; Clear the LSPEN and ASPEN bits
205                 STR     R1, [R0]
206                 ISB                               ; Reset pipeline now the FPU is enabled
207                 ENDIF
208
209                 LDR     R0, =SystemInit
210                 BLX     R0
211                 LDR     R0, =__main
212                 BX      R0
213                 ENDP
214
215 ; Dummy Exception Handlers (infinite loops which can be modified)
```

插入该段汇编代码

图 2-43

修改三：

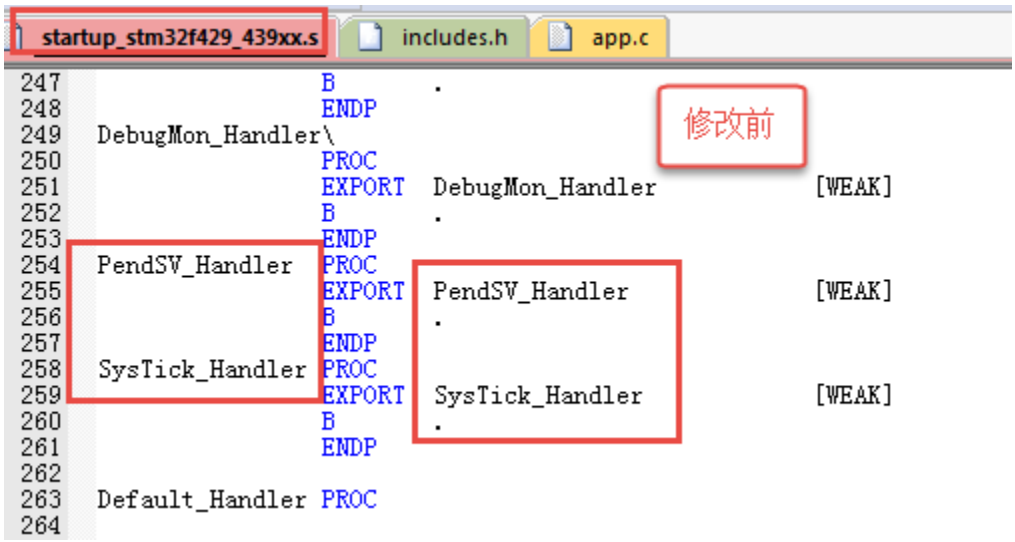

图 2-44

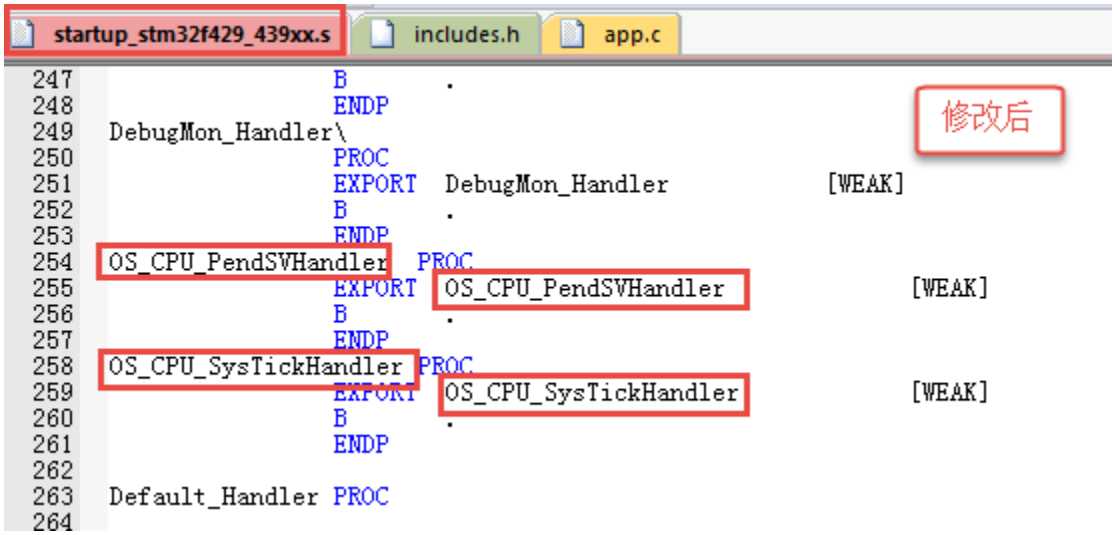

图 2-45

“bsp.h”和“bsp.c”文件使用的都是 uC/OS-III 源码的 STM32 板载驱动代码，这里要改成自己的裸机板载驱动代码。“bsp.h”的修改如下：

修改一：


```
58
59
60  /*
61  *********************************************************************************************************
62  *                                                INCLUDE FILES
63  *********************************************************************************************************
64  */
65
66  #include  <stdio.h>
67  #include  <stdarg.h>
68
69  #include  <cpu.h>
70  #include  <cpu_core.h>
71
72  #include  <lib_def.h>
73  #include  <lib_ascii.h>
74
75
76  #include  <stm32f4xx_conf.h>
77
78  /*
79  *********************************************************************************************************
80  *                                                CONSTANTS
81  *********************************************************************************************************
82  */
```

图 2-46


```
58
59
60  /*
61  *********************************************************************************************************
62  *                                                INCLUDE FILES
63  *********************************************************************************************************
64  */
65
66  #include  <stdio.h>
67  #include  <stdarg.h>
68
69  #include  <cpu.h>
70  #include  <cpu_core.h>
71
72  #include  <lib_def.h>
73  #include  <lib_ascii.h>
74
75
76  #include "stm32f4xx.h"
77
78  #include "bsp_led.h"
79
80
81  /*
82  *********************************************************************************************************
83  *                                                CONSTANTS
84  *********************************************************************************************************
85  */
```

图 2-47

修改二：

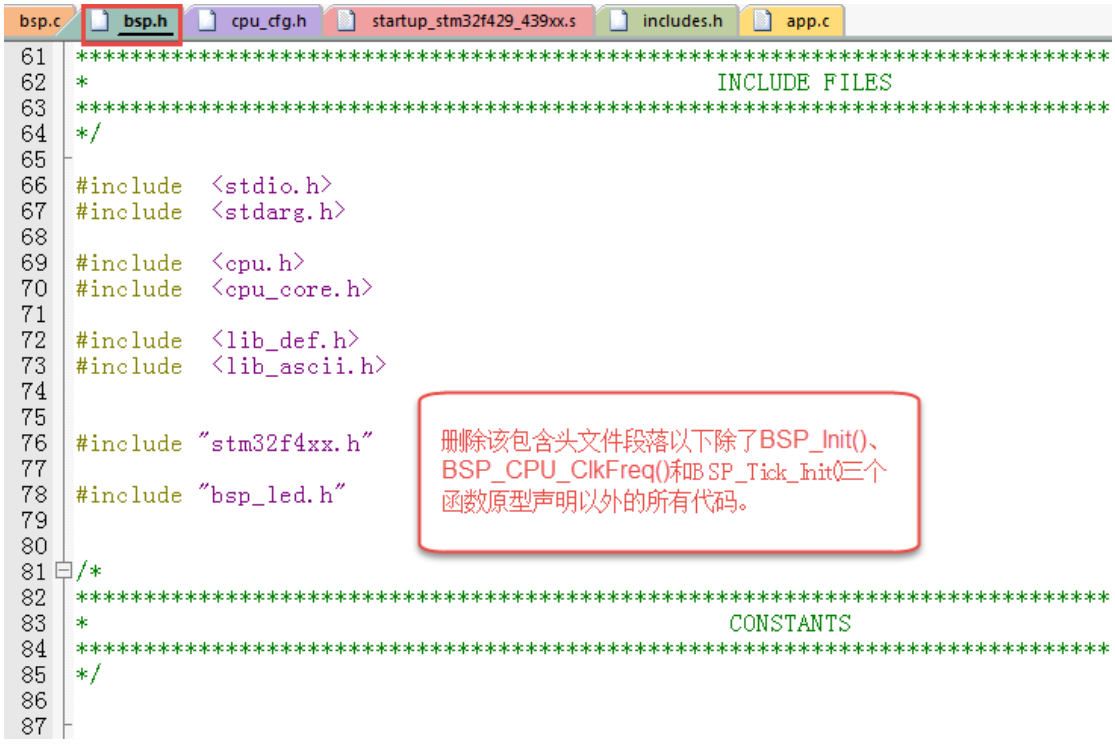

图 2-48

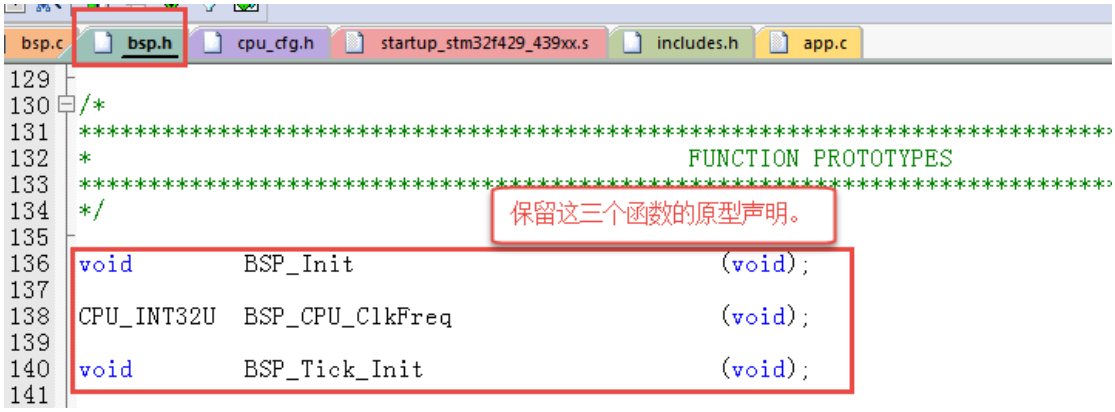

图 2-49

“bsp.c”的修改如下：

修改一：

app.c | bsp.c | bsp.h | cpu_cfg.h | startup_stm32f429_439xx.s | includes.h

```
/*
*********************************************************************************************************
*                                                LOCAL DEFINES
*********************************************************************************************************
*/


#define  BSP_GPIOA_LED1                                 DEF_BIT_04
#define  BSP_GPIOG_LED2                                 DEF_BIT_03
#define  BSP_GPIOE_LED3                                 DEF_BIT_02
#define  BSP_GPIOE_LED4                                 DEF_BIT_03

/*
```

删掉uC/OS-III自带的STM32板载驱动的宏定义。

图 2-50

修改二：

app.c | bsp.c | bsp.h | cpu_cfg.h | startup_stm32f429_439xx.s | includes.h

```
/*
*********************************************************************************************************
*                                          LOCAL FUNCTION PROTOTYPES
*********************************************************************************************************
*/

static  void  BSP_LED_Init  (void);
```

删掉uC/OS-III自带的STM32板载驱动函数。

图 2-51

修改三：

app.c | bsp.c | bsp.h | cpu_cfg.h | startup_stm32f429_439xx.s | includes.h

```
/*
*********************************************************************************************************
*                                            BSP_Init()
*
* Description : Initialize the Board Support Package (BSP).
*
* Argument(s) : none.
*
* Return(s)   : none.
*
* Caller(s)   : Application.
*
* Note(s)     : (1) This function SHOULD be called before any other BSP function is called.
*
*               (2) CPU instruction / data tracing requires the use of the following pins :
*                   (a) (1) Asynchronous      :  PB[3]
*                       (2) Synchronous 1-bit :  PE[3:2]
*                       (3) Synchronous 2-bit :  PE[4:2]
*                       (4) Synchronous 4-bit :  PE[6:2]
*
*                   (c) The application may wish to adjust the trace bus width depending on I/O
*                       requirements.
*********************************************************************************************************
*/

void  BSP_Init (void)
{
    CPU_INT32U reg_val;
    CPU_INT32U hse_rdyctr;


    BSP_IntInit();

                                                                /* ---------- RESET CLOCK CONFIG. REGISTERS ----------- */
```

修改前

清空板载初始化函数的全部定义体，改用裸机驱动函数。

图 2-52

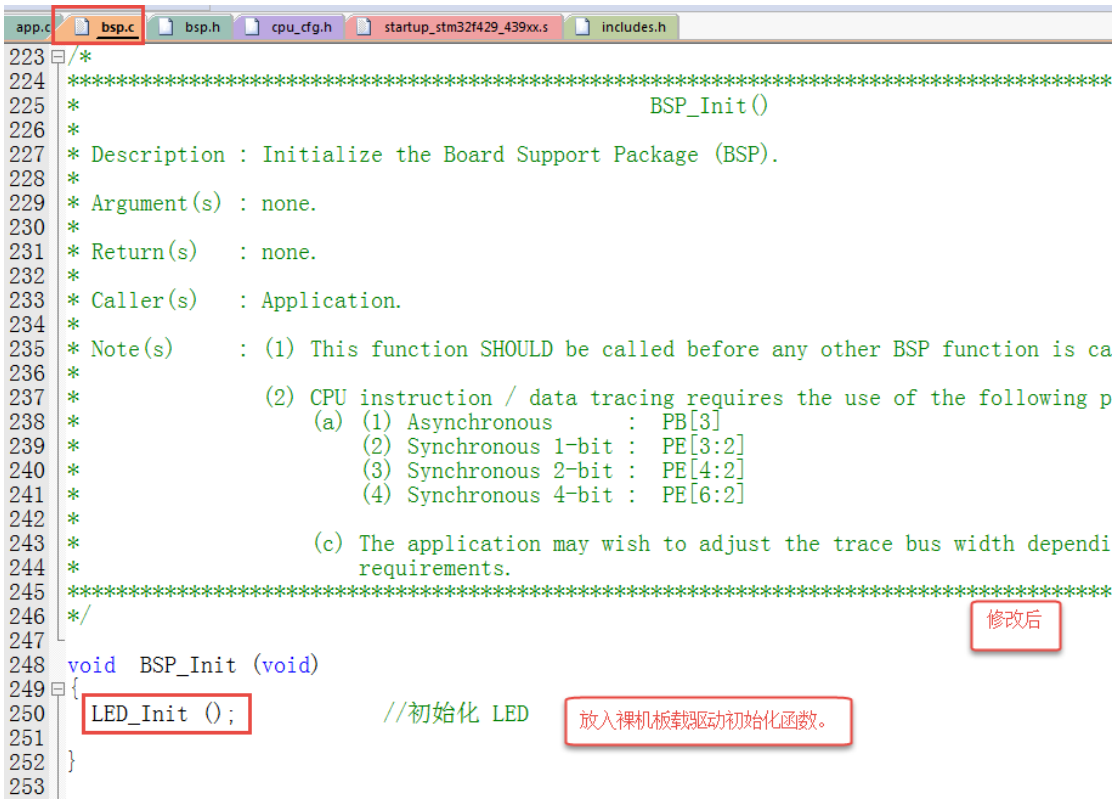

图 2-53

修改四：

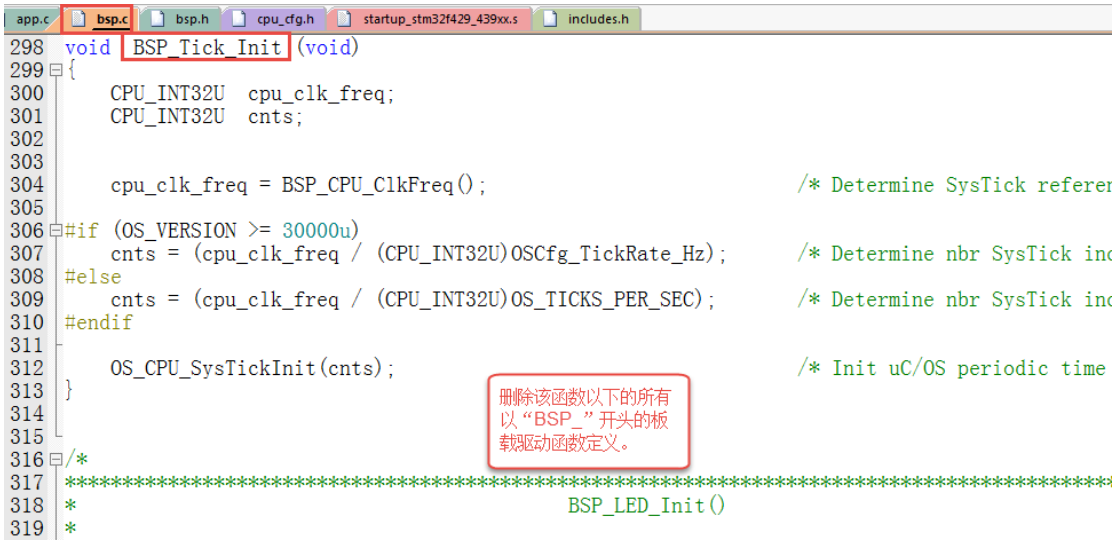

图 2-54

接下来修改应用文件“app_cfg.h”和“app.c”。在这里创建一个任务，叫起始任务，每隔 5 秒切换一次 LED1 的亮灭状态，以此来验证 uC/OS-III 系统是否移植成功。

“app_cfg.h”的修改如下：

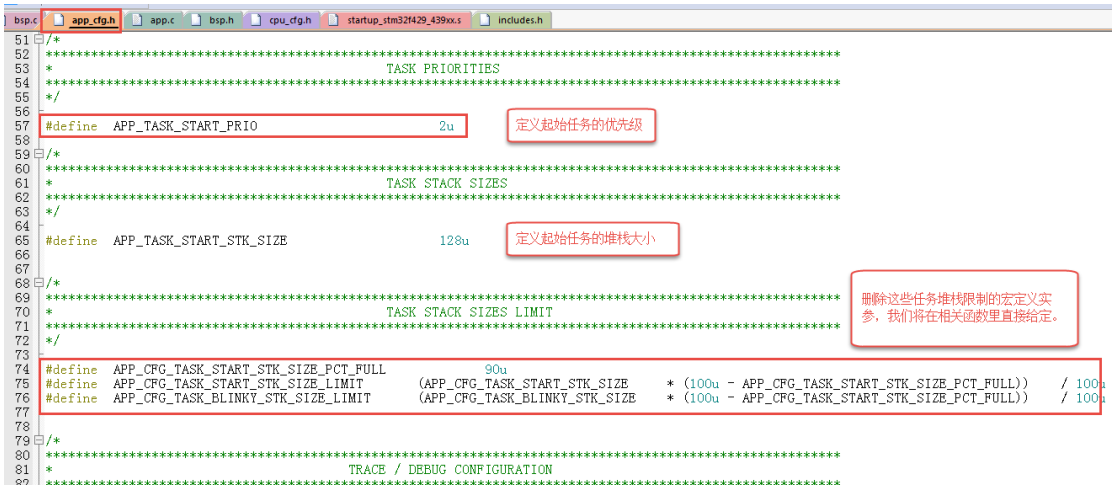

图 2-55

“app.c”的修改如下，原文件的应用稍微比较复杂，在此做整体修改，改成简单的应用。在该文件只创建和运行一个起始任务，用于每隔 5s 切换一次 LED1 的亮灭状态。

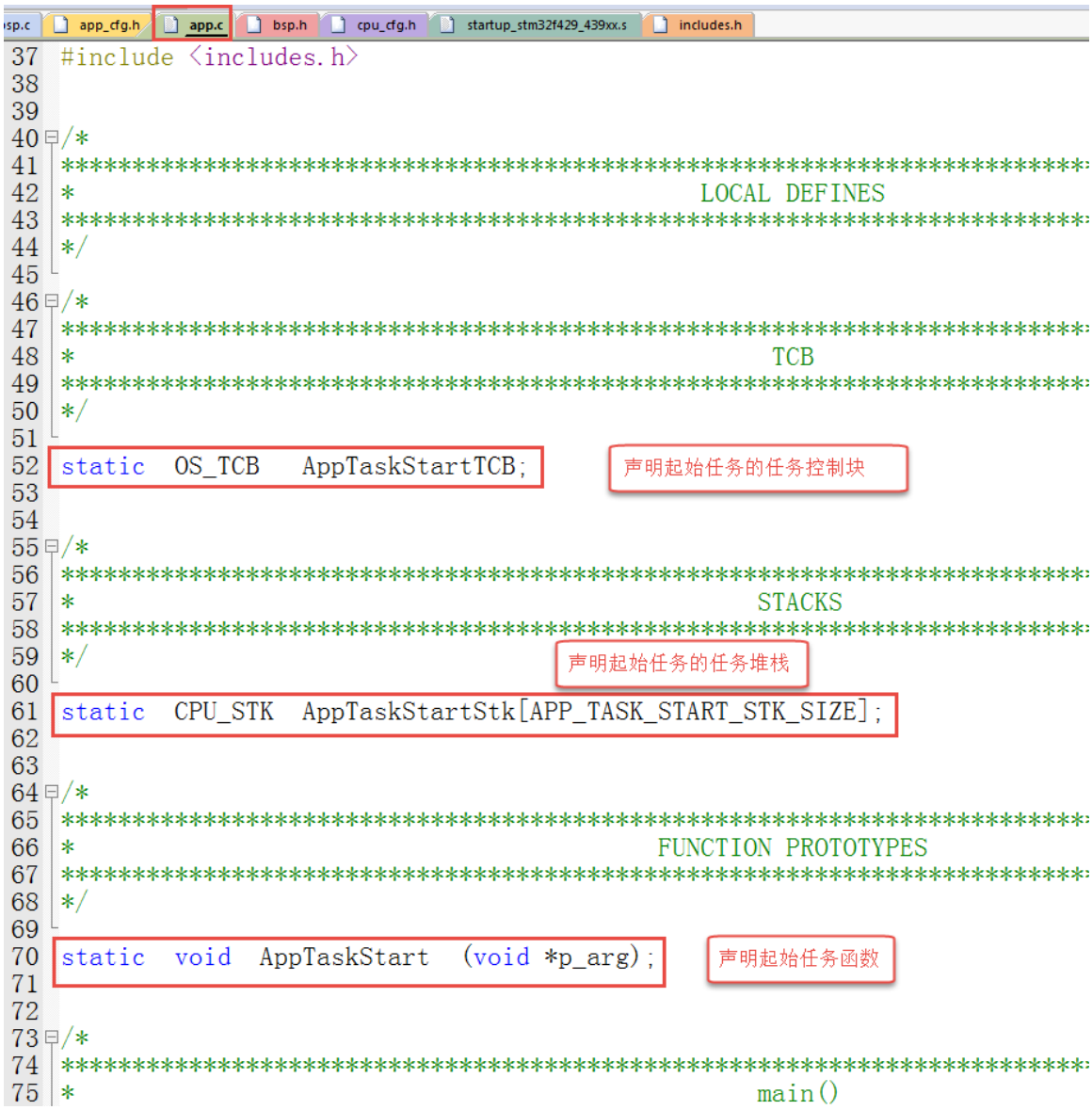

图 2-56

bsp.c | app_cfg.h | app.c | bsp.h | cpu_cfg.h | startup_stm32f429_439xx.s | includes.h

```
/*
*********************************************************************************************
*                                                main()
*
* Description : This is the standard entry point for C code.  It is assumed that your coc
*               main() once you have performed all necessary initialization.
*
* Arguments   : none
*
* Returns     : none
*********************************************************************************************
*/

int  main (void)
{
    OS_ERR  err;


    OSInit(&err);                                               /* Init uC/OS-III.
                 (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.


}
```

在主函数中创建和运行起始任务

初始化uC/OS系统

创建起始任务

开始运行系统

图 2-57

bsp.c | app_cfg.h | app.c | bsp.h | cpu_cfg.h | startup_stm32f429_439xx.s | includes.h

```
129 static  void  AppTaskStart (void *p_arg)
130 {
131     CPU_INT32U  cpu_clk_freq;
132     CPU_INT32U  cnts;
133     OS_ERR      err;
134
135
136    (void)p_arg;
137
138     BSP_Init();
139     CPU_Init();
140
141     cpu_clk_freq = BSP_CPU_ClkFreq();
142     cnts = cpu_clk_freq / (CPU_INT32U)OSCfg_TickRate_Hz;
143     OS_CPU_SysTickInit(cnts);
144
145     Mem_Init();
146
147 #if OS_CFG_STAT_TASK_EN > 0u
148     OSStatTaskCPUUsageInit(&err);
149 #endif
150
151 #ifdef CPU_CFG_INT_DIS_MEAS_EN
152     CPU_IntDisMeasMaxCurReset();
153 #endif
154
155
156     while (DEF_TRUE) {
157       macLED1_TOGGLE ();
158       OSTimeDly ( 5000, OS_OPT_TIME_DLY, & err );
159     }
160
161
162 }
163
```

定义起始任务函数体，就是起始任务要做的事情。

该任务中不停地每隔5s切换一次LED1的亮灭状态

图 2-58

编译工程，没有错误和警告，下载程序到秉火 STM32-ISO 开发板，可以观察到 LED1 每隔 5 秒切换一次亮灭状态，移植成功。

```
Build Output
compiling os_stat.c...
compiling os_task.c...
compiling os_tick.c...
compiling os_time.c...
compiling os_tmr.c...
compiling os_var.c...
assembling os_cpu_a.asm...
compiling os_cpu_c.c...
linking...
Program Size: Code=14544 RO-data=972 RW-data=232 ZI-data=12960
FromELF: creating hex file...
".\Output\WF-STM32F429.axf" - 0 Error(s), 0 Warning(s).
Build Time Elapsed:  00:00:36
```

图 2-59

秉火已将本移植工程文件存放在下图所示路径。

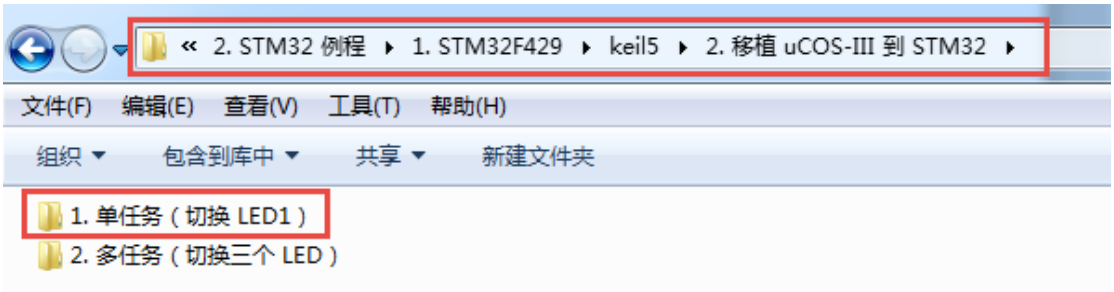

图 2-60 例程路径

2.3 建立多任务工程

uC/OS 在实际应用中，更多的是采用多任务形式，下面秉火将在上述 uC/OS-III 单任务工程的基础上，将其改为多任务工程。

在多任务工程里，秉火一共使用四个应用任务，分别是起始任务、LED1 任务、LED2 任务何 LED3 任务。主函数运行时创建起始任务，起始任务运行时进行初始化操作和创建三个 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。

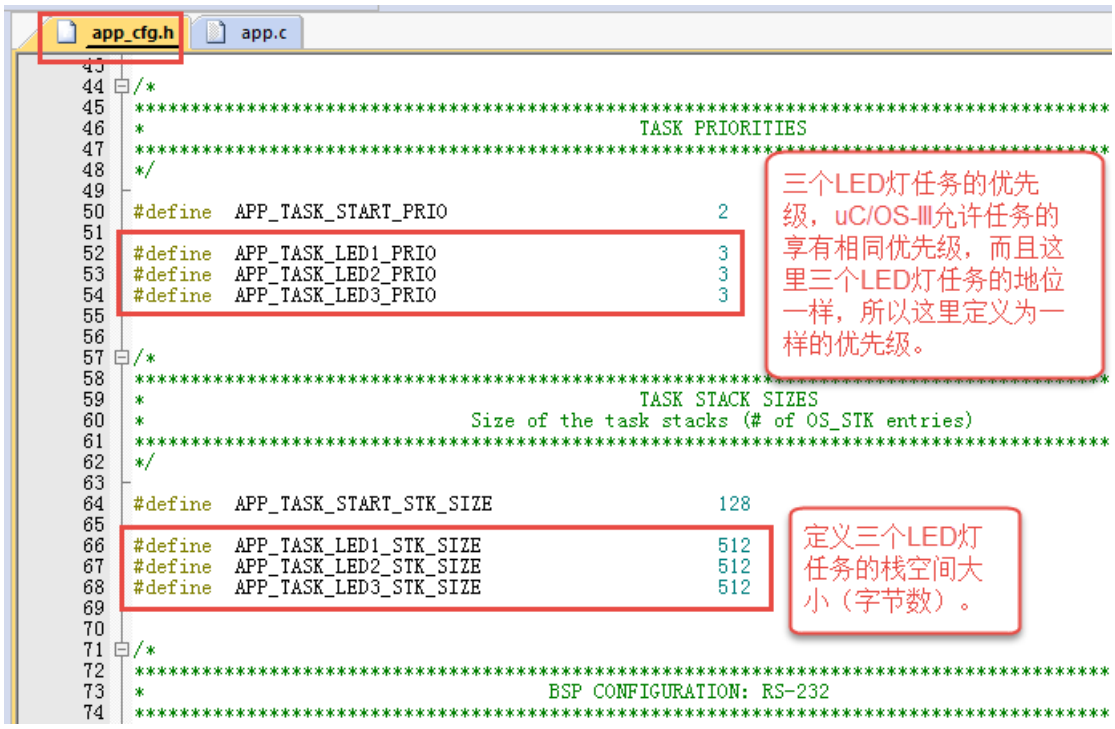

图 2-61

在“app.c”里，增加声明三个 LED 灯任务的任务控制块、栈数组和任务函数原型。

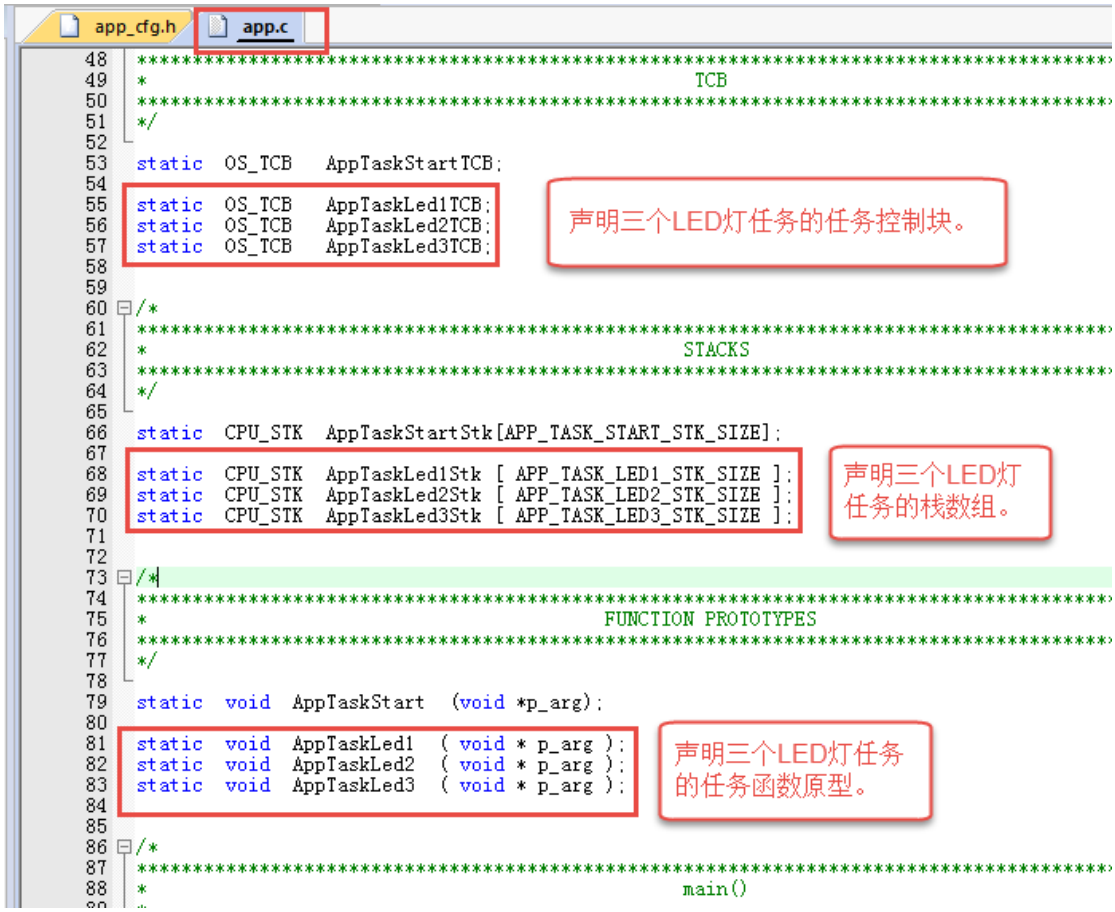

图 2-62

在“app.c”里，增加定义三个 LED 灯任务的任务函数。

app_cfg.h | app.c

```
/*
*********************************************************************************************************
*                                              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.       */
      macLED1_TOGGLE ();
      OSTimeDly ( 1000, OS_OPT_TIME_DLY, & err );
    }

}
/*
*********************************************************************************************************
*                                              LED2 TASK
*********************************************************************************************************
*/
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 );
    }

}
/*
*********************************************************************************************************
*                                              LED3 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 );
    }

}
```

在AppTaskStart()定义体下方增加定义三个LED灯任务的任务函数。

图 2-63

在“app.c”里，把起始任务函数 AppTaskStart()定义体内的 while 循环改为创建三个 LED 灯任务和删除起始任务本身。

app_cfg.h | app.c

```
125 *********************************************************************************************************
126 *                                          STARTUP TASK
127 *
128 * Description : This is an example of a startup task.  As mentioned in the book's text, you MUST
129 *               initialize the ticker only once multitasking has started.
130 *
131 * Arguments   : p_arg   is the argument passed to 'AppTaskStart()' by 'OSTaskCreate()'.
132 *
133 * Returns     : none
134 *
135 * Notes       : 1) The first line of code is used to prevent a compiler warning because 'p_arg' is not
136 *                  used.  The compiler should not generate any code for this statement.
137 *********************************************************************************************************
138 */
139
140 static  void  AppTaskStart (void *p_arg)
141 {
142     CPU_INT32U  cpu_clk_freq;
143     CPU_INT32U  cnts;
144     OS_ERR      err;
145
146
147     (void)p_arg;
148
149     BSP_Init();                                                 /* Initialize BSP functions                         */
150     CPU_Init();
151
152     cpu_clk_freq = BSP_CPU_ClkFreq();                           /* Determine SysTick reference freq.                */
153     cnts = cpu_clk_freq / (CPU_INT32U)OSCfg_TickRate_Hz;        /* Determine nbr SysTick increments                 */
154     OS_CPU_SysTickInit(cnts);                                   /* Init uC/OS periodic time src (SysTick).          */
155
156     Mem_Init();                                                 /* Initialize Memory Management Module              */
157
158 #if OS_CFG_STAT_TASK_EN > 0u
159     OSStatTaskCPUUsageInit(&err);                               /* Compute CPU capacity with no task running        */
160 #endif
161
162     CPU_IntDisMeasMaxCurReset();
163
164     while (DEF_TRUE) {                                          /* Task body, always written as an infinite loop.   */
165       macLED1_TOGGLE ();
166       OSTimeDly ( 5000, OS_OPT_TIME_DLY, & err );
167     }
168
169
170 }
```

修改前

修改该部分

图 2-64

app_cfg.h | app.c

```
137 *********************************************************************************************************
138 */
139
140 static  void  AppTaskStart (void *p_arg)
141 {
142     CPU_INT32U  cpu_clk_freq;
143     CPU_INT32U  cnts;
144     OS_ERR      err;
145
146
147    (void)p_arg;
148
149     BSP_Init();                                                 /* Initialize BSP functions                         */
150     CPU_Init();
151
152     cpu_clk_freq = BSP_CPU_ClkFreq();                           /* Determine SysTick reference freq.                */
153     cnts = cpu_clk_freq / (CPU_INT32U)OSCfg_TickRate_Hz;        /* Determine nbr SysTick increments                 */
154     OS_CPU_SysTickInit(cnts);                                   /* Init uC/OS periodic time src (SysTick).          */
155
156     Mem_Init();                                                 /* Initialize Memory Management Module              */
157
158 #if OS_CFG_STAT_TASK_EN > 0u
159     OSStatTaskCPUUsageInit(&err);                               /* Compute CPU capacity with no task running        */
160 #endif
161
162     CPU_IntDisMeasMaxCurReset();
163
164     OSTaskCreate((OS_TCB     *)&AppTaskLed1TCB,                 /* Create the Led1 task                             */
165                  (CPU_CHAR   *)"App Task Led1",
166                  (OS_TASK_PTR ) AppTaskLed1,
167                  (void       *) 0,
168                  (OS_PRIO     ) APP_TASK_LED1_PRIO,
169                  (CPU_STK    *)&AppTaskLed1Stk[0],
170                  (CPU_STK_SIZE) APP_TASK_LED1_STK_SIZE / 10,
171                  (CPU_STK_SIZE) APP_TASK_LED1_STK_SIZE,
172                  (OS_MSG_QTY  ) 5u,
173                  (OS_TICK     ) 0u,
174                  (void       *) 0,
175                  (OS_OPT      )(OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
176                  (OS_ERR     *)&err);
177
178     OSTaskCreate((OS_TCB     *)&AppTaskLed2TCB,                 /* Create the Led2 task                             */
179                  (CPU_CHAR   *)"App Task Led2",
180                  (OS_TASK_PTR ) AppTaskLed2,
181                  (void       *) 0,
182                  (OS_PRIO     ) APP_TASK_LED2_PRIO,
183                  (CPU_STK    *)&AppTaskLed2Stk[0],
184                  (CPU_STK_SIZE) APP_TASK_LED2_STK_SIZE / 10,
185                  (CPU_STK_SIZE) APP_TASK_LED2_STK_SIZE,
186                  (OS_MSG_QTY  ) 5u,
187                  (OS_TICK     ) 0u,
188                  (void       *) 0,
189                  (OS_OPT      )(OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
190                  (OS_ERR     *)&err);
191
192     OSTaskCreate((OS_TCB     *)&AppTaskLed3TCB,                 /* Create the Led3 task                             */
193                  (CPU_CHAR   *)"App Task Led3",
194                  (OS_TASK_PTR ) AppTaskLed3,
195                  (void       *) 0,
196                  (OS_PRIO     ) APP_TASK_LED3_PRIO,
197                  (CPU_STK    *)&AppTaskLed3Stk[0],
198                  (CPU_STK_SIZE) APP_TASK_LED3_STK_SIZE / 10,
199                  (CPU_STK_SIZE) APP_TASK_LED3_STK_SIZE,
200                  (OS_MSG_QTY  ) 5u,
201                  (OS_TICK     ) 0u,
202                  (void       *) 0,
203                  (OS_OPT      )(OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
204                  (OS_ERR     *)&err);
205
206
207     OSTaskDel ( & AppTaskStartTCB, & err );
208
209
210 }
```

修改后

创建三个LED灯任务，最后删除起始任务自身。

图 2-65

编译工程，没有错误和警告，下载程序到秉火 STM32-ISO 开发板，可以观察到 LED1、LED2 和 LED3 分别每隔 1 秒、5 秒和 10 秒切换一次亮灭状态，工程修改成功。

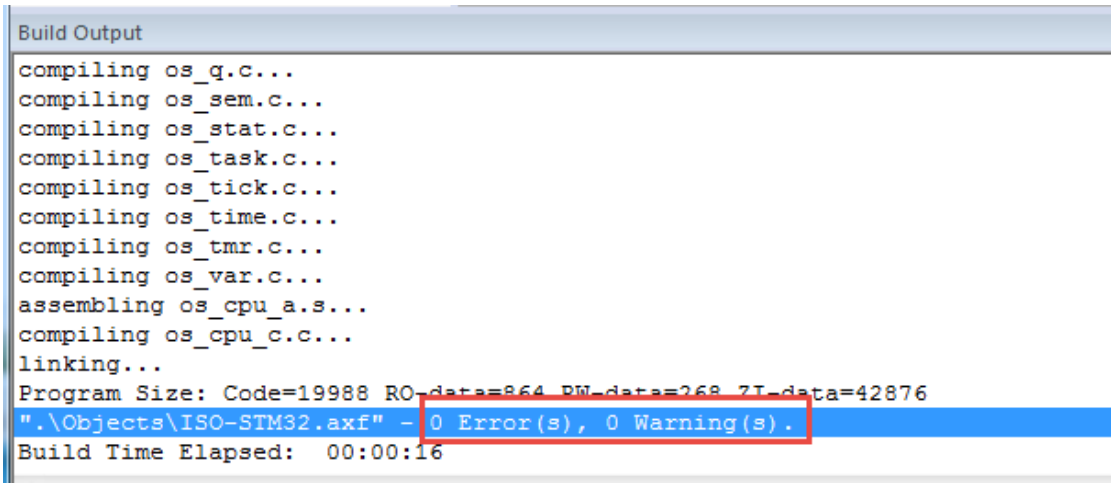

图 2-66

秉火已将本工程文件存放在下图所示路径。

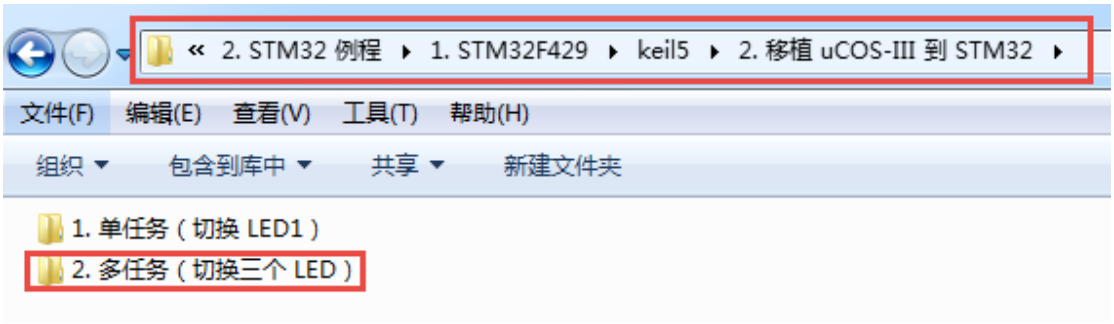

图 2-67 例程路径

2.4 章末总结

本章主要讲解了如何将 uC/OS-III 系统移植到秉火 STM32 开发板。首先到官方下载 uC/OS-III 源码，为了降低移植门槛，用户可以下载相应内核评估板的 uC/OS-III 源码，这很适合初学者首次移植 uC/OS-III 系统。然后借助开发板可行的裸机例程为平台，将 uC/OS-III 源码移植到例程上，选择一个硬件定时器驱动 uC/OS-III 的时基时钟 SysTick，带动整个 uC/OS-III 运行。最后在应用中调用 uC/OS-III 的延时函数测试延时时间是否正确，如果正确，移植 uC/OS-III 系统一般就成功了。

第3章　时钟节拍

时钟节拍可谓是 uC/OS 操作系统的心脏，它若不跳动，整个系统都将会瘫痪。时钟节拍就是操作系统的时基，操作系统要实现时间上的管理，必须依赖于时基。

3.1 原理简述

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担加重，所以用户需要权衡该频率的设置。秉火在这里采用默认的 1000 Hz（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。

```
59
60  #define  OS_CFG_TICK_RATE_HZ              1000u      /* ---------------------- TICKS ---------------------- */
61  #define  OS_CFG_TICK_TASK_PRIO              10u      // 时钟节拍频率 (10 to 1000 Hz)
62  #define  OS_CFG_TICK_TASK_STK_SIZE         128u      // 时钟节拍任务 OS_TickTask() 的优先级
63  #define  OS_CFG_TICK_WHEEL_SIZE             17u      // 时钟节拍任务 OS_TickTask() 的栈空间大小
64                                                       // OSCfg_TickWheel 数组的大小，推荐使用任务总数/4，且为质数
```

图 3-1 设置时钟节拍的频率

在本书例程中，均在“app.c”中的起始任务 AppTaskStart() 中初始化时钟节拍定时器，其实就是初始化 STM32 内核的 SysTick 时钟。

```
153
154     cpu_clk_freq = BSP_CPU_ClkFreq();                    //获取 CPU 内核时钟频率（SysTick 工作时钟）
155     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
156     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
157
```

图 3-2 初始化 SysTick 时钟

OS_CPU_SysTickInit() 函数的定义位于“os_cpu_c.c”。

```
390
391  void  OS_CPU_SysTickInit (CPU_INT32U  cnts)
392  {
393      CPU_INT32U  prio;
394
395      /* 填写 SysTick 的重载计数值 */
396      CPU_REG_NVIC_ST_RELOAD = cnts - 1u;                  // SysTick 以该计数值为周期循环计数定时
397
398      /* 设置 SysTick 中断优先级 */
399      prio  = CPU_REG_NVIC_SHPRI3;
400      prio &= DEF_BIT_FIELD(24, 0);
401      prio |= DEF_BIT_MASK(OS_CPU_CFG_SYSTICK_PRIO, 24);   //设置为默认的最高优先级0，在裸机例程中该优先级默认为最低
402
403      CPU_REG_NVIC_SHPRI3 = prio;
404
405      /* 使能 SysTick 的时钟源和启动计数器 */
406      CPU_REG_NVIC_ST_CTRL |= CPU_REG_NVIC_ST_CTRL_CLKSOURCE |
407                              CPU_REG_NVIC_ST_CTRL_ENABLE;
408      /* 使能 SysTick 的定时中断 */
409      CPU_REG_NVIC_ST_CTRL |= CPU_REG_NVIC_ST_CTRL_TICKINT;
410  }
```

图 3-3 SysTick 初始化函数 OS_CPU_SysTickInit()

SysTick 定时中断函数 OS_CPU_SysTickHandler() 的定义也位于“os_cpu_c.c”，就毗邻

OS_CPU_SysTickInit() 函数定义体的上方。OSTimeTick ()的定义位于“os_time.c”。

os_time.c | os_int.c | os_tmr.c | os_tick.c | bsp.h | app.c | os_cfg_app.h | os_cpu.h | os_cfg.h | os_cpu_c.c | cpu.h | lib_

```
362
363 void  OS_CPU_SysTickHandler (void)
364 {
365     CPU_SR_ALLOC();       //分配保存中断状态的局部变量，后面关中断的时候可以保存中断状态
366
367
368     CPU_CRITICAL_ENTER(); // CPU_CRITICAL_ENTER() 和 CPU_CRITICAL_EXIT() 之间形成临界段，避免期间程序运行时受到干扰
369     OSIntNestingCtr++;    //进入中断时中断嵌套数要加1
370     CPU_CRITICAL_EXIT();
371
372     OSTimeTick();         //调用 OSTimeTick() 函数
373
374     OSIntExit();          //退出中断，里面回家中断嵌套数减1
375 }
```

图 3-4 SysTick 中断函数 OS_CPU_SysTickHandler()

os_time.c | os_int.c | os_tmr.c | os_tick.c | bsp.h | app.c | os_cfg_app.h | os_cpu.h | os_cfg.h | os_cpu_c.c | cpu.h | lib_def.h

```
527
528 void  OSTimeTick (void)
529 {
530     OS_ERR  err;
531 #if OS_CFG_ISR_POST_DEFERRED_EN > 0u
532     CPU_TS  ts;
533 #endif
534
535
536     OSTimeTickHook();                                   //调用用户可自定义的钩子函数，可在此函数中定义在时钟节拍到来时的事件
537
538 #if OS_CFG_ISR_POST_DEFERRED_EN > 0u                    //如果使能（默认使能）了中断发送延迟
539
540     ts = OS_TS_GET();                                   //获取时间戳
541     OS_IntQPost((OS_OBJ_TYPE) OS_OBJ_TYPE_TICK,         //任务信号量暂时发送到中断队列，退出中断后由优先级最高的延迟发布任务
542                 (void      *)&OSRdyList[OSPrioCur],     //就绪发送给时钟节拍任务 OS_TickTask()，OS_TickTask() 接收到该信号量
543                 (void      *) 0,                        //就会继续执行。中断发送延迟可以减少中断时间，将中断级事件转为任务级
544                 (OS_MSG_SIZE) 0u,                       //，提高了操作系统的实时性。
545                 (OS_FLAGS   ) 0u,
546                 (OS_OPT     ) 0u,
547                 (CPU_TS     ) ts,
548                 (OS_ERR    *)&err);
549
550 #else                                                   //如果禁用（默认使能）了中断发送延迟
551
552    (void)OSTaskSemPost((OS_TCB *)&OSTickTaskTCB,        //直接发送信号量给时钟节拍任务 OS_TickTask()
553                        (OS_OPT  ) OS_OPT_POST_NONE,
554                        (OS_ERR *)&err);
555
556
557 #if OS_CFG_SCHED_ROUND_ROBIN_EN > 0u                    //如果使能（默认使能）了（同优先级任务）时间片轮转调度
558     OS_SchedRoundRobin(&OSRdyList[OSPrioCur]);          //检查当前任务的时间片是否耗尽，如果耗尽就调用同优先级的其他任务运行
559 #endif
560
561 #if OS_CFG_TMR_EN > 0u                                  //如果使能（默认使能）了软件定时器
562     OSTmrUpdateCtr--;                                   //软件定时器计数器自减
563     if (OSTmrUpdateCtr == (OS_CTR)0u) {                 //如果软件定时器计数器减至0
564         OSTmrUpdateCtr = OSTmrUpdateCnt;                //重载软件定时器计数器
565         OSTaskSemPost((OS_TCB *)&OSTmrTaskTCB,          //发送信号量给软件定时器任务 OS_TmrTask()
566                       (OS_OPT  ) OS_OPT_POST_NONE,
567                       (OS_ERR *)&err);
568     }
569 #endif
570
571 #endif
572 }
```

图 3-5 OSTimeTick ()

在函数 OSTimeTick () 会发送信号量给时基任务 OS_TickTask() ，任务 OS_TickTask() 接收到信号量后就会进入就绪状态，准备运行。

os_cfg_app.c | os_time.c | os_int.c | os_tmr.c | os_tick.c | bsp.h | app.c | os_cfg_app.h | os_cpu.h | os_cfg.h

```
60
61  void  OS_TickTask (void  *p_arg)
62  {
63      OS_ERR  err;
64      CPU_TS  ts;
65
66
67      p_arg = p_arg;                                          //预防编译警告，没有实际意义
68
69      while (DEF_ON) {                                        //循环运行
70          (void)OSTaskSemPend((OS_TICK  )0,                   //等待来自时基中断的信号量，接收到信号量后继续运行
71                              (OS_OPT   )OS_OPT_PEND_BLOCKING,
72                              (CPU_TS  *)&ts,
73                              (OS_ERR  *)&err);
74          if (err == OS_ERR_NONE) {                           //如果上面接受的信号量没有错误
75              if (OSRunning == OS_STATE_OS_RUNNING) {         //如果操作系统正在运行
76                  OS_TickListUpdate();                        //更新所有任务的时间等待时间（如延时、超时等）
77              }
78          }
79      }
80  }
```

图 3-6 时基任务 OS_TickTask ()

OS_TickListUpdate() 函数的定义位于“os_tick.c”。在一个任务将要进行延时或超时检测的时候，内核会将这些任务插入 OSCfg_TickWheel 数组的不同元素（一个元素组织一个节拍列表）中。插入操作位于 OS_TickListInsert() 函数（函数的定义也位于“os_tick.c”，就在 OS_TickListUpdate() 函数定义的上方），通过任务的 TickCtrMatch（TickCtrMatch=OSTickCtr 当前+需延时或超时节拍数）对 OSCfg_TickWheelSize 的取余（哈希算法）来决定将其插入 OSCfg_TickWheel 数组的哪个元素（列表）。相对应的，在 OS_TickListUpdate() 函数中查找到期任务时，为了能快速检测到到期的任务，通过 OSTickCtr 对 OSCfg_TickWheelSize 的取余来决定操作 OSCfg_TickWheel 数组的哪个元素（列表）。TickCtrMatch 不变，OSTickCtr 一直在计数（逢一个时钟节拍加 1），OSTickCtr 等于 TickCtrMatch 时，延时或超时完成，所以此时它俩对 OSCfg_TickWheelSize 的取余肯定相等，也就找到了到期任务在 OSCfg_TickWheel 数组的哪个元素了。这样就大大缩小了查找范围了，不用遍历 OSCfg_TickWheel 整个数组，缩小为 1/OSCfg_TickWheelSize。但是在代码中，OSTickCtr 和 TickCtrMatch 对 OSCfg_TickWheelSize 的取余相等，不一定该两变量就相等，只是可能相等，所以还得进一步判断 OSTickCtr 和 TickCtrMatch 是否相等，所以在代码中可以看到对查找到元素（列表）还进行了进一步的判断（遍历）。在一个节拍列表中，是 TickCtrMatch 从小到大排序的，所以当遍历到 OSTickCtr 和 TickCtrMatch 相等时，还要继续遍历，因为下一个 TickCtrMatch 可能和当前的 TickCtrMatch 相等；如若当遍历到 OSTickCtr 和 TickCtrMatch 不相等时，后面的肯定也不相等，就无需继续遍历了。

os_tick.c

```
271
272     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 OSCfg_TickWheel 数组
273     p_spoke = &OSCfg_TickWheel[spoke];                                       //的哪个元素（节拍列表），与查找到期任务时对应，可方便查找。
274
```

图 3-7 OS_TickListInsert() 函数中将任务插入 OSCfg_TickWheel 数组

os_tick.c | os.h | os_cfg.h

```
415  void  OS_TickListUpdate (void)
416  {
417      CPU_BOOLEAN        done;
418      OS_TICK_SPOKE     *p_spoke;
419      OS_TCB            *p_tcb;
420      OS_TCB            *p_tcb_next;
421      OS_TICK_SPOKE_IX   spoke;
422      CPU_TS             ts_start;
423      CPU_TS             ts_end;
424      CPU_SR_ALLOC();                                                  //使用到临界段（在关/开中断时）时必需该宏，该宏声明和定义一个局部变
425                                                                       //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）
426                                                                       //，开中断时将该值还原。
427      OS_CRITICAL_ENTER();                                             //进入临界段
428      ts_start = OS_TS_GET();                                          //获取 OS_TickTask() 任务的起始时间戳
429      OSTickCtr++;                                                     //时钟节拍数自加
430      spoke    = (OS_TICK_SPOKE_IX)(OSTickCtr % OSCfg_TickWheelSize);  //使用哈希算法（取余）缩小查找到期任务位于 OSCfg_TickWheel 数组的
431      p_spoke  = &OSCfg_TickWheel[spoke];                              //哪个元素（一个节拍列表），与任务插入数组时对应，下面只操作该列表。
432      p_tcb    = p_spoke->FirstPtr;                                    //获取节拍列表的首个任务控制块的地址
433      done     = DEF_FALSE;                                            //使下面 while 体得到运行
434      while (done == DEF_FALSE) {
435          if (p_tcb != (OS_TCB *)0) {                                  //如果该任务不空（存在）
436              p_tcb_next = p_tcb->TickNextPtr;                         //获取该列表中紧邻该任务的下一个任务控制块的地址
437              switch (p_tcb->TaskState) {                              //根据该任务的任务状态处理
438                  case OS_TASK_STATE_RDY:                              //如果任务状态均是与时间事件无关，就无需理会
439                  case OS_TASK_STATE_PEND:
440                  case OS_TASK_STATE_SUSPENDED:
441                  case OS_TASK_STATE_PEND_SUSPENDED:
442                       break;
443
444                  case OS_TASK_STATE_DLY:                              //如果是延时状态
445                       p_tcb->TickRemain = p_tcb->TickCtrMatch         //计算延时的的剩余时间
446                                         - OSTickCtr;
447                       if (OSTickCtr == p_tcb->TickCtrMatch) {         //如果任务期满
448                           p_tcb->TaskState = OS_TASK_STATE_RDY;       //修改任务状态量为就绪状态
449                           OS_TaskRdy(p_tcb);                          //让任务就绪
450                       } else {                                        //如果任务未期满（由于升序排列，该列表后面的任务肯定也未期满）
451                           done             = DEF_TRUE;                //不再遍历该列表，退出 while 循环
452                       }
453                       break;
```

os_tick.c | os.h | os_cfg.h

```
454
455                  case OS_TASK_STATE_PEND_TIMEOUT:                     //如果是有期限等待状态
456                       p_tcb->TickRemain = p_tcb->TickCtrMatch         //计算期限的的剩余时间
457                                         - OSTickCtr;
458                       if (OSTickCtr == p_tcb->TickCtrMatch) {         //如果任务期满
459  #if (OS_MSG_EN > 0u)                                                 //如果使能了消息队列（普通消息队列或任务消息队列）
460                           p_tcb->MsgPtr     = (void       *)0;       //把任务保存接收到消息的地址的成员清空
461                           p_tcb->MsgSize    = (OS_MSG_SIZE)0u;       //把任务保存接收到消息的长度的成员清零
462  #endif
463                           p_tcb->TS         = OS_TS_GET();           //记录任务结束等待的时间戳
464                           OS_PendListRemove(p_tcb);                  //从等待列表移除该任务
465                           OS_TaskRdy(p_tcb);                         //让任务就绪
466                           p_tcb->TaskState  = OS_TASK_STATE_RDY;       //修改任务状态量为就绪状态
467                           p_tcb->PendStatus = OS_STATUS_PEND_TIMEOUT;  //记录等待状态为超时
468                           p_tcb->PendOn     = OS_TASK_PEND_ON_NOTHING; //记录等待内核对象变量为空
469                       } else {                                       //如果任务未期满（由于升序排列，该列表后面的任务肯定也未期满）
470                           done              = DEF_TRUE;              //不再遍历该列表，退出 while 循环
471                       }
472                       break;
473
474                  case OS_TASK_STATE_DLY_SUSPENDED:                    //如果是延时中被挂起状态
475                       p_tcb->TickRemain = p_tcb->TickCtrMatch         //计算延时的的剩余时间
476                                         - OSTickCtr;
477                       if (OSTickCtr == p_tcb->TickCtrMatch) {         //如果任务期满
478                           p_tcb->TaskState  = OS_TASK_STATE_SUSPENDED; //修改任务状态量为被挂起状态
479                           OS_TickListRemove(p_tcb);                  //从节拍列表移除该任务
480                       } else {                                       //如果任务未期满（由于升序排列，该列表后面的任务肯定也未期满）
481                           done              = DEF_TRUE;              //不再遍历该列表，退出 while 循环
482                       }
483                       break;
```

os_tick.c | os.h | os_cfg.h

```
484
485                  case OS_TASK_STATE_PEND_TIMEOUT_SUSPENDED:           //如果是有期限等待中被挂起状态
486                       p_tcb->TickRemain = p_tcb->TickCtrMatch         //计算期限的的剩余时间
487                                         - OSTickCtr;
488                       if (OSTickCtr == p_tcb->TickCtrMatch) {         //如果任务期满
489  #if (OS_MSG_EN > 0u)                                                 //如果使能了消息队列（普通消息队列或任务消息队列）
490                           p_tcb->MsgPtr     = (void       *)0;       //把任务保存接收到消息的地址的成员清空
491                           p_tcb->MsgSize    = (OS_MSG_SIZE)0u;       //把任务保存接收到消息的长度的成员清零
492  #endif
493                           p_tcb->TS         = OS_TS_GET();           //记录任务结束等待的时间戳
494                           OS_PendListRemove(p_tcb);                  //从等待列表移除该任务
495                           OS_TickListRemove(p_tcb);                  //从节拍列表移除该任务
496                           p_tcb->TaskState  = OS_TASK_STATE_SUSPENDED; //修改任务状态量为被挂起状态
497                           p_tcb->PendStatus = OS_STATUS_PEND_TIMEOUT;  //记录等待状态为超时
498                           p_tcb->PendOn     = OS_TASK_PEND_ON_NOTHING; //记录等待内核对象变量为空
499                       } else {                                       //如果任务未期满（由于升序排列，该列表后面的任务肯定也未期满）
500                           done              = DEF_TRUE;              //不再遍历该列表，退出 while 循环
501                       }
502                       break;
503
504                  default:
505                       break;
506              }
507              p_tcb = p_tcb_next;                                      //遍历节拍列表的下一个任务
508          } else {                                                     //如果该任务为空（节拍列表后面肯定也都是空的）
509              done  = DEF_TRUE;                                        //不再遍历该列表，退出 while 循环
510          }
511      }
512      ts_end = OS_TS_GET() - ts_start;                                 //获取 OS_TickTask() 任务的结束时间戳，并计算其执行时间
513      if (OSTickTaskTimeMax < ts_end) {                                //更新 OS_TickTask() 任务的最大运行时间
514          OSTickTaskTimeMax = ts_end;
515      }
516      OS_CRITICAL_EXIT();                                              //退出临界段
517  }
```

图 3-8 节拍列表更新函数 OS_TickListUpdate()

3.2 实例演示

3.2.1 实例 1

本节实例将在一个任务里延时 1000 个时钟节拍（1s），在延时前后获取时间戳，然后通过时间戳来计算延时时间。该例程已经存放在配套资料的下图路径。

图 3-9 例程路径

本例程需用使用 USART1，所以工程中需要添加改 USART1 的驱动文件到“...\User\BSP”路径下，在 MDK 上添加驱动文件的源文件和头文件路径，再在“bsp.h”中包含驱动文件的头文件，然后在“bsp.c”的 BSP_Init() 函数的定义体内添加 USART1 初始化函数。

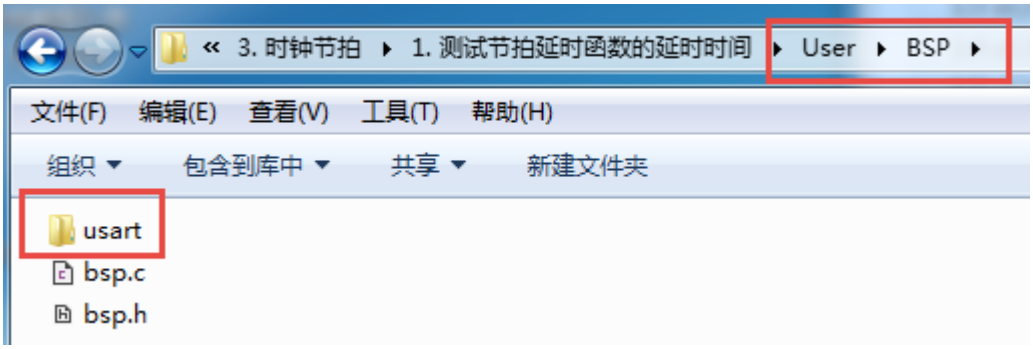

图 3-10 板级驱动文件路径

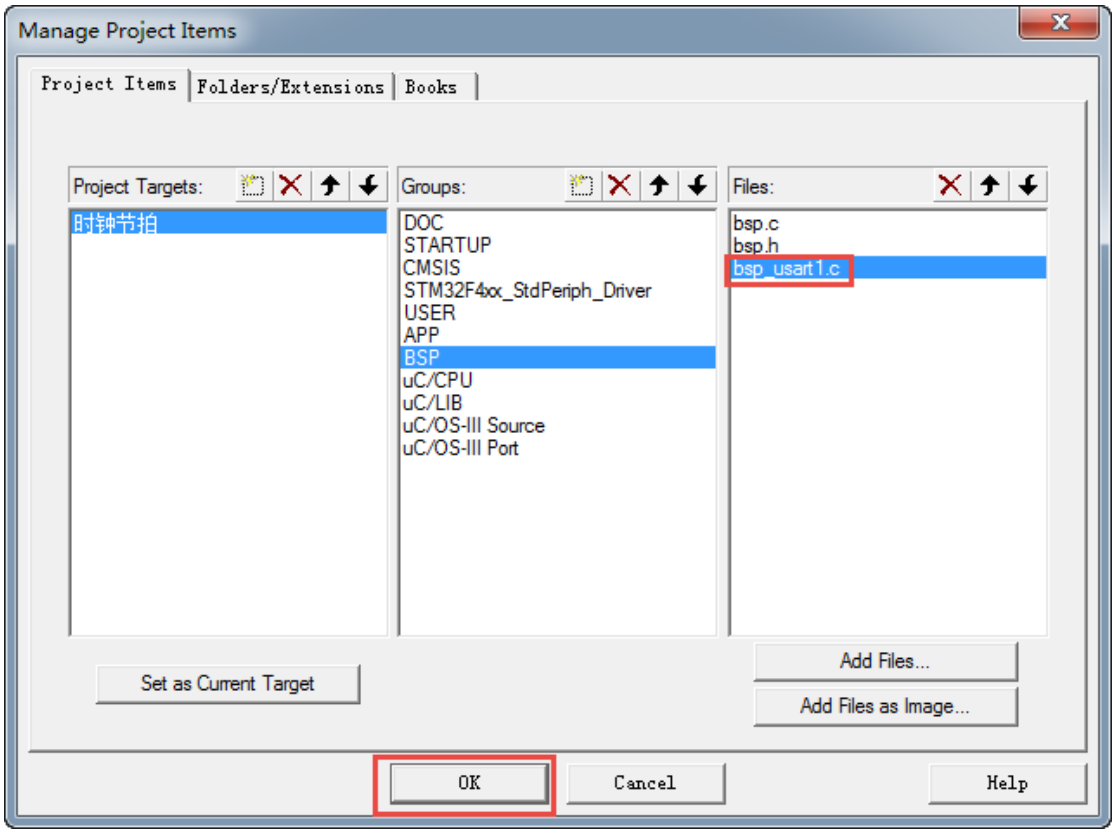

图 3-11 在 MDK 增加驱动文件的源文件到组件

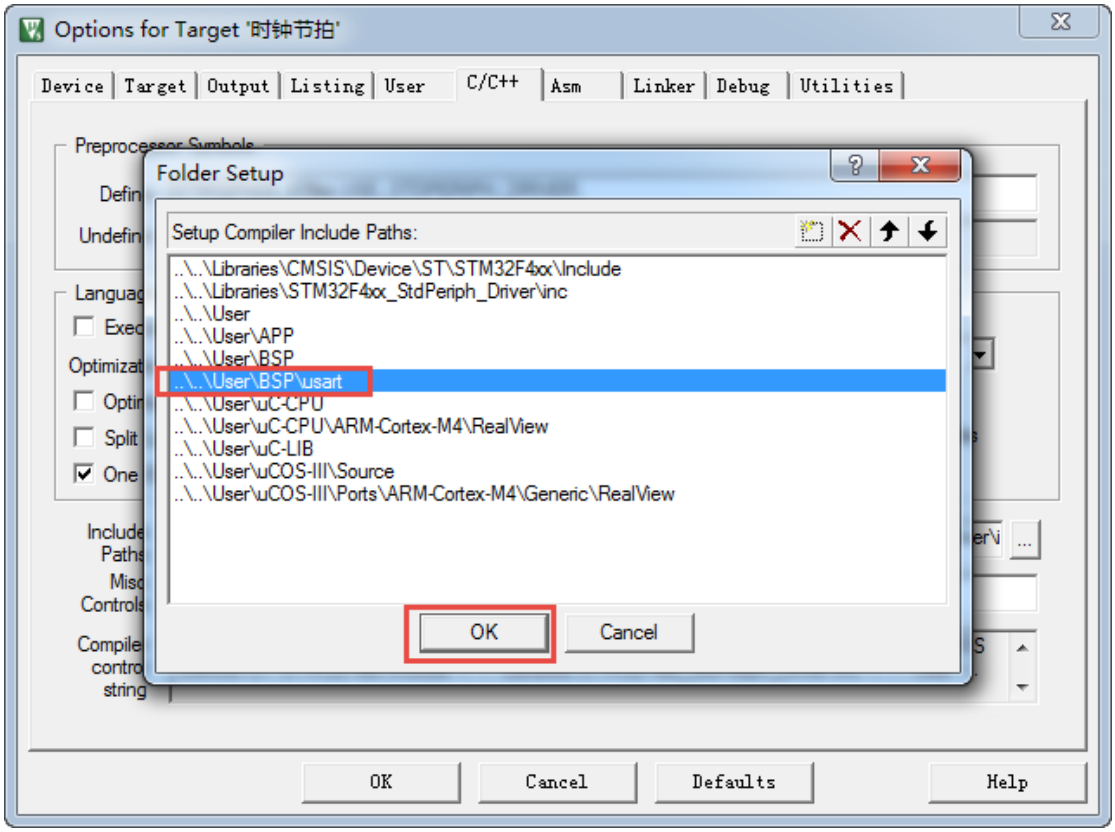

图 3-12 在 MDK 增加驱动文件的头文件路径

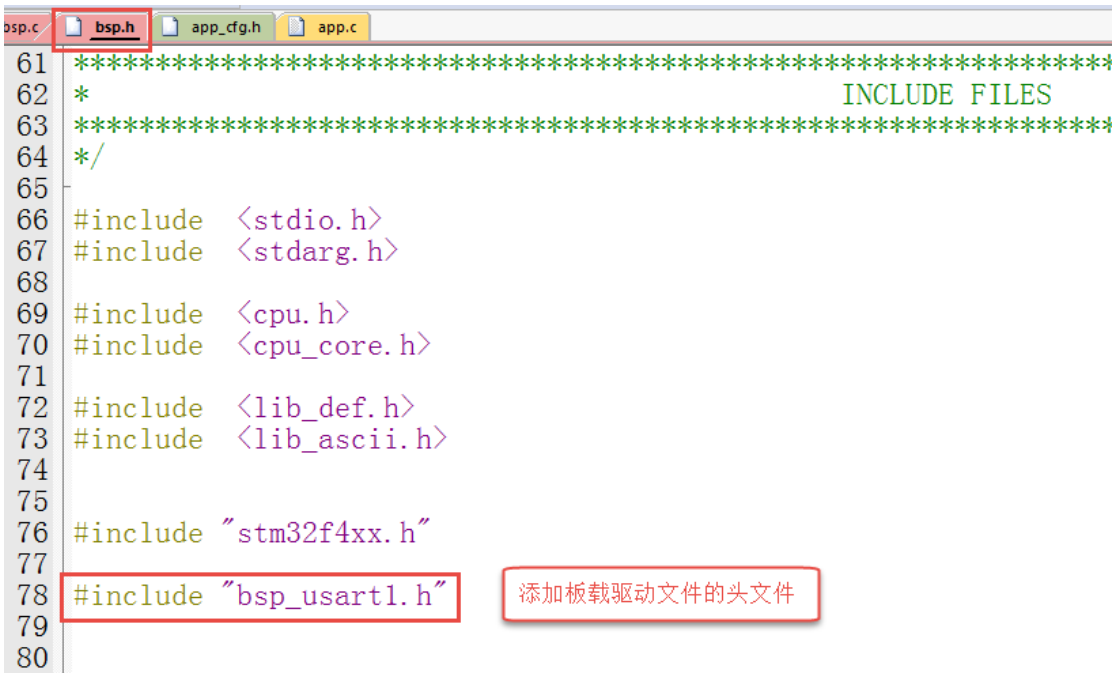

图 3-13 包含板级驱动文件的头文件

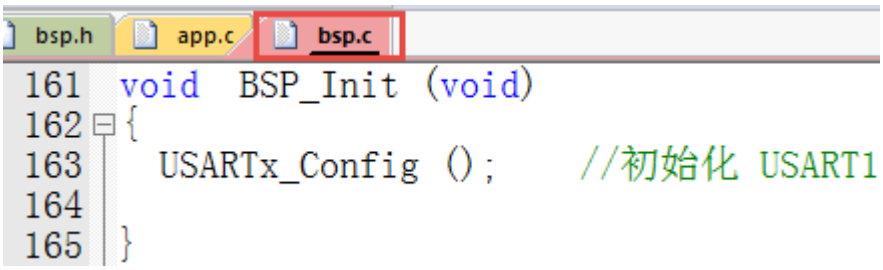

图 3-14 添加板级初始化函数

在“app.c”中，新建了一个任务 AppTaskTest() ，任务在 while 死循环里延时了 1000 个节拍（1s），并通过时间戳计算该延时的实际执行时间，并将该时间通过串口打印到串口调试助手。

app_cfg.h | app.c

```
/*
*********************************************************************************************************
*                                            TEST TASK
*********************************************************************************************************
*/

static  void  AppTaskTest ( void * p_arg )
{
  OS_ERR          err;
  CPU_INT32U      cpu_clk_freq;
  CPU_TS          ts_start;
  CPU_TS          ts_end;
  CPU_SR_ALLOC();                                   //使用到临界段（在关/开中断时）时必需该宏，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。
 (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，通常都写成一个死循环
    ts_start = OS_TS_GET();                         //获取延时前时间戳

    OSTimeDly ( 1000, OS_OPT_TIME_DLY, & err );     //延时1000个时钟节拍（1s）

    ts_end = OS_TS_GET() - ts_start;                //获取延时后的时间戳（以CPU时钟进行计数的一个计数值），并计算延时时间

    OS_CRITICAL_ENTER();                            //进入临界段，不希望下面串口打印遭到中断

    printf ( "\r\n延时1000个时钟节拍（1s），通过时间戳测得延时 %07d us，即 %04d ms。",
             ts_end / ( cpu_clk_freq / 1000000 ),   //将延时时间折算成 us
             ts_end / ( cpu_clk_freq / 1000 ) );    //将延时时间折算成 ms

    OS_CRITICAL_EXIT();                             //进入临界段，不希望下面串口打印遭到中断

  }


}
```

图 3-15 AppTaskTest() 任务函数

程序在主函数 main() 中创建起始任务，起始任务运行时会创建 AppTaskTest() 任务，然后删除自身，之后就只有 AppTaskTest() 一个应用任务在运行。

app_cfg.h | app.c

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


    OSInit(&err);                                                   //初始化 uC/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],                 //任务堆栈的基地址
                 (CPU_STK_SIZE) APP_TASK_START_STK_SIZE / 10,       //任务堆栈空间剩下1/10时限制其增长
                 (CPU_STK_SIZE) APP_TASK_START_STK_SIZE,            //任务堆栈空间（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uC/OS-III控制）


}
```

图 3-16 主函数

app_cfg.h | app.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、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                                    //如果使能（默认使能）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


    /* 创建测试任务 */
    OSTaskCreate((OS_TCB     *)&AppTaskTestTCB,                                 //任务控制块地址
                 (CPU_CHAR   *)"App Task Test",                                 //任务名称
                 (OS_TASK_PTR ) AppTaskTest,                                    //任务函数
                 (void       *) 0,                                              //传递给任务函数（形参p_arg）的实参
                 (OS_PRIO     ) APP_TASK_TEST_PRIO,                             //任务的优先级
                 (CPU_STK    *)&AppTaskTestStk[0],                              //任务堆栈的基地址
                 (CPU_STK_SIZE) APP_TASK_TEST_STK_SIZE / 10,                    //任务堆栈空间剩下1/10时限制其增长
                 (CPU_STK_SIZE) APP_TASK_TEST_STK_SIZE,                         //任务堆栈空间（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


}
```

图 3-17 起始任务 AppTaskStart()

把 STM32 的 USART1 连接至电脑的串口调试助手，编译和下载程序到秉火 STM32 开发板，运行程序。用户可以观察到下图的实验现象。

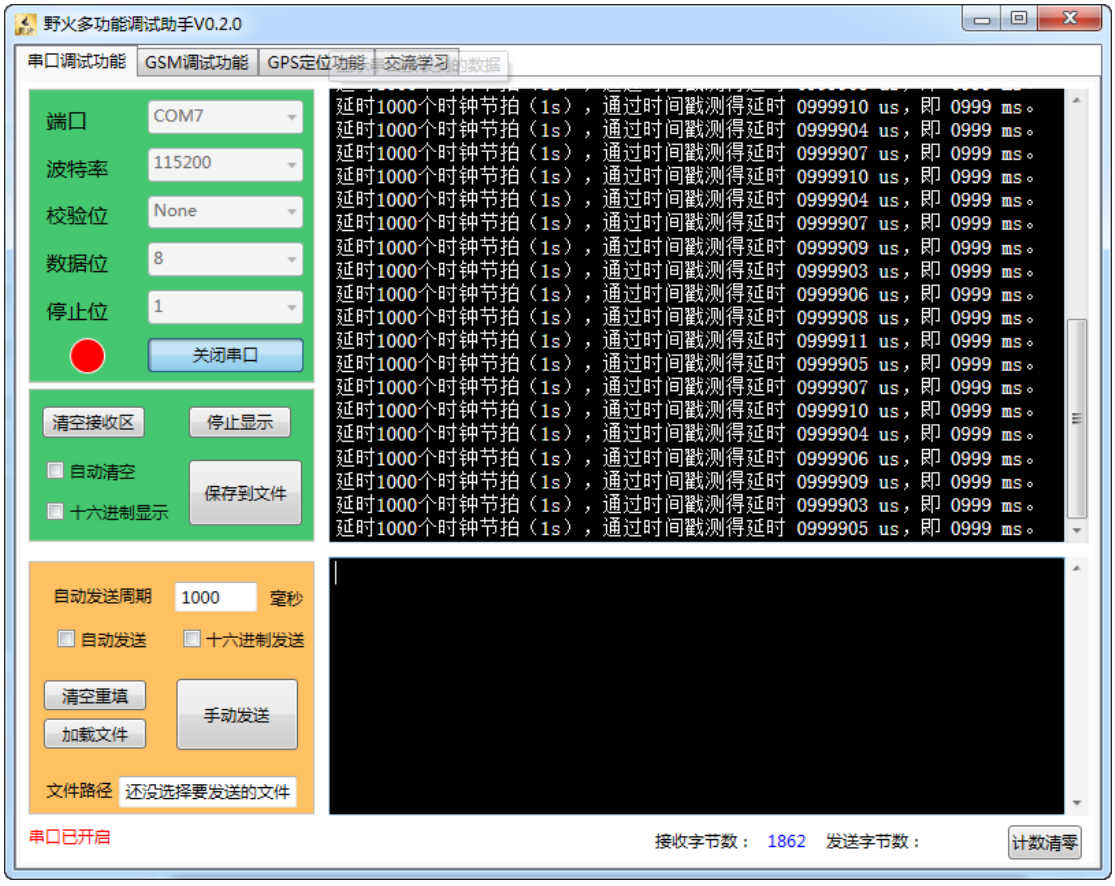

图 3-18 串口调试助手

从打印信息可以看到，测出来的延时时间并非刚好 1 s，而是略少了 6 ms 多。这一方面是由于延时操作的插入可能在两个时钟节拍的任意时刻上，如下图所示，所以它可能会存在一个节拍的误差。另外这是软件延时，精度会比硬件延时略逊一筹。但是，在 uC/OS 实际应用中，该延时的精度已经算非常高了。

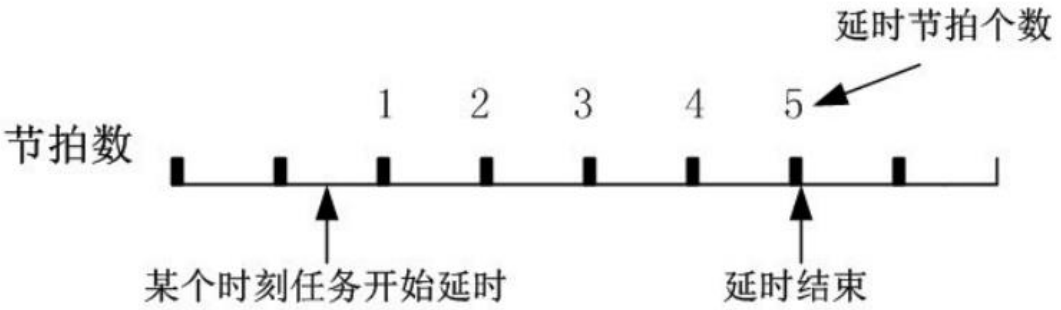

图 3-19 延时误差示意图

3.3 章末总结

本章阐述了时钟节拍的工作原理，时钟节拍看似微小，却是整个 uC/OS 系统的命脉。时钟节拍的运行依赖于 CPU 的定时器，STM32 专门为此量身定制了 SysTick 时钟，用户移植时需要将节拍时钟移植到 SysTick 时钟。每个时钟节拍到来时，节拍任务就会运行，节拍任务的重点是更新节拍列表。在节拍列表中，存放的均是与时间事件（如延时或超时等）相关的任务，在节拍任务中会检查这些任务的时间事件是否到期，如有到期，立即更新任务的任务状态。在查找到期任务时，还巧妙地使用了哈希算法大大缩小了查找范围，提高了操作系统的工作效率。

最后，通过实例，为大家演示了时钟节拍在延时函数中的应用。

第4章 时间管理

在上一章的示例演示中使用到延时函数，其实已经是 uC/OS 操作系统时间管理的范畴了。前一章撒下的影子，也算为本章做了一个铺垫。时间管理就是一种建立在时钟节拍上，对操作系统任务的运行实现时间上管理的一种系统内核机制。

4.1 原理简述

由于下面很多内核服务函数在实际开发应用中要使用到，所以本节主要以服务函数为主导讲解时间管理的工作原理。

4.1.1 OSTimeDly()

OSTimeDly() 函数用于停止当前任务进行的运行，延时一段时间后再运行。OSTimeDly() 函数的信息如下表所示。

表 4 OSTimeDly()

函数原型	void OSTimeDly (OS_TICK dly, OS_OPT opt, OS_ERR *p_err);			
功能	延时执行当前任务，延时结束后再回来执行当前任务。			
参数	dly	延时的时钟节拍数。		
	opt	选项	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。
			OS_OPT_TIME_TIMEOUT	跟 OS_OPT_TIME_DLY 情况一样。
			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。
			OS_OPT_TIME_PERIODIC	周期性延时，跟 OS_OPT_TIME_DLY 差不多。如果是长时间延时，该选项更精准一些。
	p_err	返回错误类型	OS_ERR_NONE	没错误。
			OS_ERR_OPT_INVALID	选项不可用。
			OS_ERR_SCHED_LOCKED	调度器被锁
			OS_ERR_TIME_DLY_ISR	在中断中使用该函数。
			OS_ERR_TIME_ZERO_DLY	Dly 为 0。
返回值	无。			
注意事项	不可以在中断中调用该函数。			

OSTimeDly() 函数的定义位于“os_time.c”。

```
startup_stm32f10x_hd.s | app.c | os.h | os_time.c | os_task.c | os_core.c | os_cpu.h | os_int.c | os_tmr.c | os_tick.c
 80  void  OSTimeDly (OS_TICK    dly,                        //延时的时钟节拍数
 81                   OS_OPT     opt,                        //选项
 82                   OS_ERR    *p_err)                      //返回错误类型
 83  {
 84      CPU_SR_ALLOC();                                     //使用到临界段（在关/开中断时）时必需该宏，该宏声明和定义一个局部变
 85                                                          //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）
 86                                                          //，开中断时将该值还原。
 87
 88  #ifdef OS_SAFETY_CRITICAL                               //如果使能（默认禁用）了安全检测
 89      if (p_err == (OS_ERR *)0) {                         //如果错误类型实参为空
 90          OS_SAFETY_CRITICAL_EXCEPTION();                 //执行安全检测异常函数
 91          return;                                         //返回，不执行延时操作
 92      }
 93  #endif
 94
 95  #if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u                  //如果使能（默认使能）了中断中非法调用检测
 96      if (OSIntNestingCtr > (OS_NESTING_CTR)0u) {         //如果该延时函数是在中断中被调用
 97         *p_err = OS_ERR_TIME_DLY_ISR;                    //错误类型为“在中断函数中延时”
 98          return;                                         //返回，不执行延时操作
 99      }
100  #endif
101      /* 当调度器被锁时任务不能延时 */
102      if (OSSchedLockNestingCtr > (OS_NESTING_CTR)0u) {   //如果调度器被锁
103         *p_err = OS_ERR_SCHED_LOCKED;                    //错误类型为“调度器被锁”
104          return;                                         //返回，不执行延时操作
105      }

startup_stm32f10x_hd.s | app.c | os.h | os_time.c | os_task.c | os_core.c | os_cpu.h | os_int.c | os_tmr.c | os_tick.c
106
107     switch (opt) {                                      //根据延时选项参数 opt 分类操作
108         case OS_OPT_TIME_DLY:                           //如果选择相对时间（从现在起延时多长时间）
109         case OS_OPT_TIME_TIMEOUT:                       //如果选择超时（实际同上）
110         case OS_OPT_TIME_PERIODIC:                      //如果选择周期性延时
111              if (dly == (OS_TICK)0u) {                  //如果参数 dly 为0（0意味不延时）
112                 *p_err = OS_ERR_TIME_ZERO_DLY;          //错误类型为“0延时”
113                  return;                                //返回，不执行延时操作
114              }
115              break;
116
117         case OS_OPT_TIME_MATCH:                         //如果选择绝对时间（匹配系统开始运行（OSStart()）后的时钟节拍数）
118              break;
119
120         default:                                        //如果选项超出范围
121             *p_err = OS_ERR_OPT_INVALID;                //错误类型为“选项非法”
122              return;                                    //返回，不执行延时操作
123     }
124
125     OS_CRITICAL_ENTER();                                //进入临界段
126     OSTCBCurPtr->TaskState = OS_TASK_STATE_DLY;         //修改当前任务的任务状态为延时状态
127     OS_TickListInsert(OSTCBCurPtr,                      //将当前任务插入到节拍列表
128                       dly,
129                       opt,
130                       p_err);
131     if (*p_err != OS_ERR_NONE) {                        //如果当前任务插入节拍列表时出现错误
132          OS_CRITICAL_EXIT_NO_SCHED();                   //退出临界段（无调度）
133          return;                                        //返回，不执行延时操作
134     }
135     OS_RdyListRemove(OSTCBCurPtr);                      //从就绪列表移除当前任务
136     OS_CRITICAL_EXIT_NO_SCHED();                        //退出临界段（无调度）
137     OSSched();                                          //任务切换
138    *p_err = OS_ERR_NONE;                                //错误类型为“无错误”
139 }
```

图 4-1 OSTimeDly() 函数

OSTimeDly() 函数中，如果函数的实参和调用场合均合法，就会调用 OS_TickListInsert() 函数将当前任务插入到节拍列表进行管理，这与上一章所讲的更新节拍列表是对应的。OS_TickListInsert() 函数的定义位于“os_tick.c”。要插入节拍列表的任务的等待时间有个门限 OS_TICK_TH_RDY，该宏为 0Xffff0000。如果以时钟节拍最高频率 1000 Hz 来算，该宏所代表时间都有 49 天（至少数），任务等待这么长的时间在实际应用中是不可能存在的，所以 uC/OS 设置了该门限，凡是超过该门限的等待时间均视为 0 延时，防止恶性等待，浪费资源。

startup_stm32f10x_hd.s | app.c | os.h | os_time.c | os_task.c | os_core.c | os_cpu.h | os_int.c | os_tmr.c | os_tick.c | os_type.h | lib_def.h | cpu.h | cpu_def

```
220 void  OS_TickListInsert (OS_TCB    *p_tcb, //任务控制块
221                         OS_TICK   time,  //时间
222                         OS_OPT    opt,   //选项
223                         OS_ERR    *p_err) //返回错误类型
224 {
225     OS_TICK            tick_delta;
226     OS_TICK            tick_next;
227     OS_TICK_SPOKE     *p_spoke;
228     OS_TCB            *p_tcb0;
229     OS_TCB            *p_tcb1;
230     OS_TICK_SPOKE_IX   spoke;
231
232
233
234     if (opt == OS_OPT_TIME_MATCH) {                                 //如果 time 是个绝对时间
235         tick_delta = time - OSTickCtr - 1u;                         //计算离到期还有多长时间
236         if (tick_delta > OS_TICK_TH_RDY) {                          //如果延时时间超过了门限
237             p_tcb->TickCtrMatch = (OS_TICK        )0u;              //将任务的时钟节拍的匹配变量置0
238             p_tcb->TickRemain   = (OS_TICK        )0u;              //将任务的延时还需时钟节拍数置0
239             p_tcb->TickSpokePtr = (OS_TICK_SPOKE *)0;               //该任务不插入节拍列表
240            *p_err               =  OS_ERR_TIME_ZERO_DLY;            //错误类型相当于“0延时”
241             return;                                                 //返回，不将任务插入节拍列表
242         }
243         p_tcb->TickCtrMatch = time;                                 //任务等待的匹配点为 OSTickCtr = time
244         p_tcb->TickRemain   = tick_delta + 1u;                      //计算任务离到期还有多长时间
245
246     } else if (time > (OS_TICK)0u) {                                //如果 time > 0
247         if (opt == OS_OPT_TIME_PERIODIC) {                          //如果 time 是周期性时间
248             tick_next  = p_tcb->TickCtrPrev + time;                 //计算任务接下来要匹配的时钟节拍总计数
249             tick_delta = tick_next - OSTickCtr - 1u;                //计算任务离匹配还有个多长时间
250             if (tick_delta < time) {                                //如果 p_tcb->TickCtrPrev < OSTickCtr + 1
251                 p_tcb->TickCtrMatch = tick_next;                    //将 p_tcb->TickCtrPrev + time 设为时钟节拍匹配点
252             } else {                                                //如果 p_tcb->TickCtrPrev >= OSTickCtr + 1
253                 p_tcb->TickCtrMatch = OSTickCtr + time;             //将 OSTickCtr + time 设为时钟节拍匹配点
254             }
255             p_tcb->TickRemain   = p_tcb->TickCtrMatch - OSTickCtr; //计算任务离到期还有多长时间
256             p_tcb->TickCtrPrev  = p_tcb->TickCtrMatch;              //保存当前匹配值为下一周期延时用
257
258         } else {                                                    //如果 time 是相对时间
259             p_tcb->TickCtrMatch = OSTickCtr + time;                 //任务等待的匹配点为 OSTickCtr + time
260             p_tcb->TickRemain   = time;                             //计算任务离到期的时间就是 time
261         }
262
263     } else {                                                        //如果 time = 0
264         p_tcb->TickCtrMatch = (OS_TICK        )0u;                  //将任务的时钟节拍的匹配变量置0
265         p_tcb->TickRemain   = (OS_TICK        )0u;                  //将任务的延时还需时钟节拍数置0
266         p_tcb->TickSpokePtr = (OS_TICK_SPOKE *)0;                   //该任务不插入节拍列表
267        *p_err               =  OS_ERR_TIME_ZERO_DLY;                //错误类型为“0延时”
268         return;                                                     //返回，不将任务插入节拍列表
269     }
270
```

startup_stm32f10x_hd.s | app.c | os.h | os_time.c | os_task.c | os_core.c | os_cpu.h | os_int.c | os_tmr.c | os_tick.c | os_type.h | lib_def.h | cpu.h | cpu_def.h

```
271
272     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
273     p_spoke = &OSCfg_TickWheel[spoke];                                       //OSCfg_TickWheel的哪个元素（组织一个节拍列表）
274                                                                              //与更新节拍列表相对应，可方便查找到期任务。
```

```
275     if (p_spoke->NbrEntries == (OS_OBJ_QTY)0u) {                //如果当前节拍列表为空
276         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
277         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
278         p_spoke->FirstPtr     =  p_tcb;                         //当前任务被列为该节拍列表的第一个任务
279         p_spoke->NbrEntries   = (OS_OBJ_QTY)1u;                 //节拍列表中的元素数目为1
280     } else {                                                    //如果当前节拍列表非空
281         p_tcb1     = p_spoke->FirstPtr;                         //获取列表中的第一个任务
282         while (p_tcb1 != (OS_TCB *)0) {                         //如果该任务存在
283             p_tcb1->TickRemain = p_tcb1->TickCtrMatch           //计算该任务的剩余等待时间
284                                - OSTickCtr;
285             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
286                 if (p_tcb1->TickNextPtr != (OS_TCB *)0) {       //如果该任务不是列表的最后一个元素
287                     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
288                 } else {                                        //如果该任务是列表的最后一个元素
289                     p_tcb->TickNextPtr  = (OS_TCB *)0;          //当前任务为列表的最后一个元素
290                     p_tcb->TickPrevPtr  =  p_tcb1;              //该任务是当前任务的前一个元素
291                     p_tcb1->TickNextPtr =  p_tcb;               //当前任务是该任务的后一个元素
292                     p_tcb1              = (OS_TCB *)0;          //插入完成，退出 while 循环
293                 }
294             } else {                                            //如果当前任务的剩余等待时间不大于该任务的
295                 if (p_tcb1->TickPrevPtr == (OS_TCB *)0) {       //如果该任务是列表的第一个元素
296                     p_tcb->TickPrevPtr  = (OS_TCB *)0;          //当前任务就作为列表的第一个元素
297                     p_tcb->TickNextPtr  =  p_tcb1;              //该任务是当前任务的后一个元素
298                     p_tcb1->TickPrevPtr =  p_tcb;               //当前任务是该任务的前一个元素
299                     p_spoke->FirstPtr   =  p_tcb;               //当前任务是列表的第一个元素
300                 } else {                                        //如果该任务也不是是列表的第一个元素
301                     p_tcb0              =  p_tcb1->TickPrevPtr; // p_tcb0 暂存该任务的前一个任务
302                     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
303                     p_tcb->TickNextPtr  =  p_tcb1;              //该任务作为当前任务的后一个任务
304                     p_tcb0->TickNextPtr =  p_tcb;               // p_tcb0 暂存的任务的下一个任务改为当前任务
305                     p_tcb1->TickPrevPtr =  p_tcb;               // 该任务的前一个任务也改为当前任务
306                 }
307                 p_tcb1 = (OS_TCB *)0;                           //插入完成，退出 while 循环
308             }
309         }
310         p_spoke->NbrEntries++;                                  //节拍列表中的元素数目加1
311     }
312     if (p_spoke->NbrEntriesMax < p_spoke->NbrEntries) {         //更新节拍列表的元素数目的最大记录
313         p_spoke->NbrEntriesMax = p_spoke->NbrEntries;
314     }
315     p_tcb->TickSpokePtr = p_spoke;                              //记录当前任务存放于哪个节拍列表
316    *p_err               = OS_ERR_NONE;                          //错误类型为“无错误”
317 }
```

图 4-2 OS_TickListInsert() 函数

如果 OSTimeDly() 函数调用 OS_TickListInsert() 函数将当前任务插入节拍列表成功的话，就会调用 OS_RdyListRemove() 函数将当前任务从任务就绪列表中移除，并将系统切换至其他任务。OS_RdyListRemove() 函数属于任务管理范畴，该函数的原理可参考“任务管理”章节。

4.1.2 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，用户若要使用 OSTimeDlyHMSM() 函数，得事先将宏 OS_CFG_TIME_DLY_HMSM_EN（位于“os_cfg.h”）设为 1。

```
96
97  #define OS_CFG_TIME_DLY_HMSM_EN          1u   /* ------------------------- TIME MANAGEMENT ---
                                                  //使能/禁用 OSTimeDlyHMSM() 函数
98  #define OS_CFG_TIME_DLY_RESUME_EN        1u   //使能/禁用 OSTimeDlyResume() 函数
```

图 4-3 使能 OSTimeDlyHMSM() 函数

OSTimeDlyHMSM () 函数的信息如下表所示。

表 5 OSTimeDlyHMSM()

函数	void OSTimeDlyHMSM (CPU_INT16U hours, CPU_INT16U minutes,

<table>
<tr><td>原型</td><td colspan="5">CPU_INT16U seconds,
CPU_INT32U milli,
OS_OPT opt,
OS_ERR *p_err);</td></tr>
<tr><td>功能</td><td colspan="5">延时执行当前任务，延时结束后再回来执行当前任务。</td></tr>
<tr><td rowspan="11">参数</td><td>hours</td><td colspan="4">小时数。</td></tr>
<tr><td>minutes</td><td colspan="4">分钟数。</td></tr>
<tr><td>seconds</td><td colspan="4">秒数。</td></tr>
<tr><td>milli</td><td colspan="4">毫秒数。</td></tr>
<tr><td rowspan="5">opt</td><td rowspan="5">选项</td><td>OS_OPT_TIME_DLY</td><td colspan="2">相对性时间，就是从现在起延时多长时间。</td></tr>
<tr><td>OS_OPT_TIME_TIMEOUT</td><td colspan="2">跟 OS_OPT_TIME_DLY 情况一样。</td></tr>
<tr><td>OS_OPT_TIME_MATCH</td><td colspan="2">绝对性时间，就把系统开始运行（调用 OSStart()）时作为延时时间的起点。</td></tr>
<tr><td>OS_OPT_TIME_PERIODIC</td><td colspan="2">周期性延时，跟 OS_OPT_TIME_DLY 差不多。如果是长时间延时，该选项更精准一些。</td></tr>
<tr><td>OS_OPT_TIME_HMSM_STRICT</td><td colspan="2">延时时间取值比较严格。
hours (0...99)
minutes (0...59)
seconds (0...59)
milliseconds (0...999)</td></tr>
<tr><td></td><td></td><td>OS_OPT_TIME_HMSM_NON_STRICT</td><td colspan="2">延时时间取值比较宽松。
hours (0...999)
minutes (0...9999)
seconds (0...65535)
milliseconds (0...4294967295)</td></tr>
<tr><td>p_err</td><td colspan="2">返回错误类型</td><td>OS_ERR_NONE</td><td>没错误。</td></tr>
<tr><td colspan="3"></td><td>OS_ERR_OPT_INVALID</td><td>选项不可用。</td></tr>
<tr><td colspan="3"></td><td>OS_ERR_SCHED_LOCKED</td><td>调度器被锁</td></tr>
<tr><td colspan="3"></td><td>OS_ERR_TIME_DLY_ISR</td><td>在中断中使用该函数。</td></tr>
<tr><td colspan="3"></td><td>OS_ERR_TIME_INVALID_HOURS</td><td>小时数不可用。</td></tr>
<tr><td colspan="3"></td><td>OS_ERR_TIME_INVALID_MINUTES</td><td>分钟数不可用。</td></tr>
<tr><td colspan="3"></td><td>OS_ERR_TIME_INVALID_SECONDS</td><td>秒数不可用。</td></tr>
<tr><td colspan="3"></td><td>OS_ERR_TIME_INVALID_MILLISECONDS</td><td>毫秒数不可用。</td></tr>
<tr><td colspan="3"></td><td>OS_ERR_TIME_ZERO_DLY</td><td>延时时间为 0。</td></tr>
<tr><td>返</td><td colspan="5">无。</td></tr>
</table>

回值	
注意事项	不可以在中断中调用该函数。

OSTimeDlyHMSM() 函数的定义位于“os_time.c”。

os_cfg_app.h | os_cfg.h | os_time.c | lib_def.h | os.h

```
199 #if OS_CFG_TIME_DLY_HMSM_EN > 0u                    //如果使能（默认使能）了 OSTimeDlyHMSM() 函数
200 void  OSTimeDlyHMSM (CPU_INT16U   hours,           //延时小时数
201                       CPU_INT16U   minutes,         //分钟数
202                       CPU_INT16U   seconds,         //秒数
203                       CPU_INT32U   milli,           //毫秒数
204                       OS_OPT       opt,             //选项
205                       OS_ERR      *p_err)           //返回错误类型
206 {
207 #if OS_CFG_ARG_CHK_EN > 0u                          //如果使能（默认使能）了参数检测功能
208     CPU_BOOLEAN  opt_invalid;                       //声明变量用于参数检测
209     CPU_BOOLEAN  opt_non_strict;
210 #endif
211     OS_OPT       opt_time;
212     OS_RATE_HZ   tick_rate;
213     OS_TICK      ticks;
214     CPU_SR_ALLOC();
215
216
217
218 #ifdef OS_SAFETY_CRITICAL                           //如果使能（默认禁用）了安全检测
219     if (p_err == (OS_ERR *)0) {                     //如果错误类型实参为空
220         OS_SAFETY_CRITICAL_EXCEPTION();             //执行安全检测异常函数
221         return;                                     //返回，不执行延时操作
222     }
223 #endif
```

os_cfg_app.h | os_cfg.h | os_time.c | lib_def.h | os.h

```
224
225 #if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u                  //如果使能（默认使能）了中断中非法调用检测
226     if (OSIntNestingCtr > (OS_NESTING_CTR)0u) {         //如果该延时函数是在中断中被调用
227         *p_err = OS_ERR_TIME_DLY_ISR;                   //错误类型为“在中断函数中延时”
228          return;                                        //返回，不执行延时操作
229     }
230 #endif
231     /* 当调度器被锁时任务不能延时 */
232     if (OSSchedLockNestingCtr > (OS_NESTING_CTR)0u) {   //如果调度器被锁
233         *p_err = OS_ERR_SCHED_LOCKED;                   //错误类型为“调度器被锁”
234          return;                                        //返回，不执行延时操作
235     }
236
237     opt_time = opt & OS_OPT_TIME_MASK;                  //检测除选项中与延时时间性质有关的位
238     switch (opt_time) {                                 //根据延时选项参数 opt 分类操作
239         case OS_OPT_TIME_DLY:                           //如果选择相对时间（从现在起延时多长时间）
240         case OS_OPT_TIME_TIMEOUT:                       //如果选择超时（实际同上）
241         case OS_OPT_TIME_PERIODIC:                      //如果选择周期性延时
242              if (milli == (CPU_INT32U)0u) {             //如果毫秒数为0
243                  if (seconds == (CPU_INT16U)0u) {       //如果秒数为0
244                      if (minutes == (CPU_INT16U)0u) {   //如果分钟数为0
245                          if (hours == (CPU_INT16U)0u) { //如果小时数为0
246                              *p_err = OS_ERR_TIME_ZERO_DLY; //错误类型为“0延时”
247                               return;                   //返回，不执行延时操作
248                          }
249                      }
250                  }
251              }
252              break;
253
254         case OS_OPT_TIME_MATCH:                         //如果选择绝对时间（把系统开始运行（OSStart()时做为起点）
255              break;
256
257         default:                                        //如果选项超出范围
258             *p_err = OS_ERR_OPT_INVALID;                //错误类型为“选项非法”
259              return;                                    //返回，不执行延时操作
260     }
```

```
os_cfg_app.h | os_cfg.h | os_time.c | lib_def.h | os.h
261
262 #if OS_CFG_ARG_CHK_EN > 0u                                    //如果使能（默认使能）了参数检测功能
263     opt_invalid = DEF_BIT_IS_SET_ANY(opt, ~OS_OPT_TIME_OPTS_MASK);  //检测除选项位以后其他位是否被置位
264     if (opt_invalid == DEF_YES) {                             //如果除选项位以后其他位有被置位的
265         *p_err = OS_ERR_OPT_INVALID;                          //错误类型为“选项非法”
266          return;                                              //返回，不执行延时操作
267     }
268
269     opt_non_strict = DEF_BIT_IS_SET(opt, OS_OPT_TIME_HMSM_NON_STRICT); //检测有关时间参数取值范围的选项位
270     if (opt_non_strict != DEF_YES) {                          //如果选项选择了 OS_OPT_TIME_HMSM_STRICT
271         if (milli   > (CPU_INT32U)999u) {                     //如果毫秒数>999
272             *p_err = OS_ERR_TIME_INVALID_MILLISECONDS;        //错误类型为“毫秒数不可用”
273              return;                                          //返回，不执行延时操作
274         }
275         if (seconds > (CPU_INT16U)59u) {                      //如果秒数>59
276             *p_err = OS_ERR_TIME_INVALID_SECONDS;             //错误类型为“秒数不可用”
277              return;                                          //返回，不执行延时操作
278         }
279         if (minutes > (CPU_INT16U)59u) {                      //如果分钟数>59
280             *p_err = OS_ERR_TIME_INVALID_MINUTES;             //错误类型为“分钟数不可用”
281              return;                                          //返回，不执行延时操作
282         }
283         if (hours   > (CPU_INT16U)99u) {                      //如果小时数>99
284             *p_err = OS_ERR_TIME_INVALID_HOURS;               //错误类型为“小时数不可用”
285              return;                                          //返回，不执行延时操作
286         }
287     } else {                                                  //如果选项选择了 OS_OPT_TIME_HMSM_  NON_STRICT
288         if (minutes > (CPU_INT16U)9999u) {                    //如果分钟数>9999
289             *p_err = OS_ERR_TIME_INVALID_MINUTES;             //错误类型为“分钟数不可用”
290              return;                                          //返回，不执行延时操作
291         }
292         if (hours   > (CPU_INT16U)999u) {                     //如果小时数>999
293             *p_err = OS_ERR_TIME_INVALID_HOURS;               //错误类型为“小时数不可用”
294              return;                                          //返回，不执行延时操作
295         }
296     }
297 #endif
298
```

```
os_cfg_app.h | os_cfg.h | os_time.c | lib_def.h | os.h
299
300     /*将延时时间转换成时钟节拍数*/
301     tick_rate = OSCfg_TickRate_Hz;                                                    //获取时钟节拍的频率
302     ticks     = ((OS_TICK)hours * (OS_TICK)3600u + (OS_TICK)minutes * (OS_TICK)60u + (OS_TICK)seconds) * tick_rate //将延时时间转换成时钟节拍数
303               + (tick_rate * ((OS_TICK)milli + (OS_TICK)500u / tick_rate)) / (OS_TICK)1000u;
```

```
os_cfg_app.h | os_cfg.h | os_time.c | lib_def.h | os.h
304
305     if (ticks > (OS_TICK)0u) {                                //如果延时节拍数>0
306         OS_CRITICAL_ENTER();                                  //进入临界段
307         OSTCBCurPtr->TaskState = OS_TASK_STATE_DLY;            //修改当前任务的任务状态为延时状态
308         OS_TickListInsert(OSTCBCurPtr,                        //将当前任务插入到节拍列表
309                           ticks,
310                           opt_time,
311                           p_err);
312         if (*p_err != OS_ERR_NONE) {                          //如果当前任务插入节拍列表时出现错误
313             OS_CRITICAL_EXIT_NO_SCHED();                      //退出临界段（无调度）
314             return;                                           //返回，不执行延时操作
315         }
316         OS_RdyListRemove(OSTCBCurPtr);                        //从就绪列表移除当前任务
317         OS_CRITICAL_EXIT_NO_SCHED();                          //退出临界段（无调度）
318         OSSched();                                            //任务切换
319        *p_err = OS_ERR_NONE;                                  //错误类型为“无错误”
320     } else {                                                  //如果延时节拍数=0
321        *p_err = OS_ERR_TIME_ZERO_DLY;                         //错误类型为“0延时”
322     }
323 }
324 #endif
```

图 4-4 OSTimeDlyHMSM() 函数

OSTimeDlyHMSM() 函数中，如果函数的实参和调用场合均合法，就会调用 OS_TickListInsert() 函数将当前任务插入到节拍列表进行管理。OS_TickListInsert() 函数的原理参考“OSTimeDly()”章节。

如果 OSTimeDlyHMSM() 函数调用 OS_TickListInsert() 函数将当前任务插入节拍列表成功的话，就会调用 OS_RdyListRemove() 函数将当前任务从任务就绪列表中移除，并将系统切换至其他任务。

4.1.3 OSTimeDlyResume()

OSTimeDlyResume () 函数用于直接结束其他任务（非当前任务）的延时。用户若要使用 OSTimeDlyResume () 函数，得事先将宏 OS_CFG_TIME_DLY_RESUME_EN（位于“os_cfg.h”）设为 1。

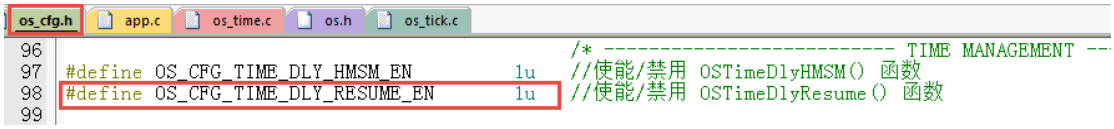

图 4-5 使能 OSTimeDlyResume () 函数

OSTimeDlyResume () 函数的信息如下表所示。

表 6 OSTimeDlyResume ()

函数原型	void OSTimeDlyResume (OS_TCB *p_tcb, OS_ERR *p_err);			
功能	结束其他任务的延时。			
参数	p_tcb	任务的任务控制块指针。		
	p_err	返回错误类型	OS_ERR_NONE	没错误。
			OS_ERR_STATE_INVALID	任务状态非法。
			OS_ERR_TIME_DLY_RESUME_ISR	在中断中结束延时。
			OS_ERR_ TASK_NOT_DLY	任务不在延时。
			OS_ERR_TASK_SUSPENDED	任务被挂起。
返回值	无。			
注意事项	✧ 该函数的 p_tcb 参数不可以是当前任务。 ✧ 不可以在中断中调用该函数。 ✧ 不可以用该函数结束等待事件。			

OSTimeDlyResume () 函数的定义位于“os_time.c”。

```
os_cfg.h  app.c  os_time.c  os.h  os_tick.c
348 #if OS_CFG_TIME_DLY_RESUME_EN > 0u                    //如果使能（默认使能）了 OSTimeDlyResume() 函数
349 void  OSTimeDlyResume (OS_TCB  *p_tcb,                //任务控制块
350                        OS_ERR  *p_err)                //错误类型
351 {
352     CPU_SR_ALLOC();                                   //使用到临界段（在关/开中断时）时必需该宏，该宏声明和定义一个局部变
353                                                       //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）
354                                                       //，开中断时将该值还原。
355
356 #ifdef OS_SAFETY_CRITICAL                             //如果使能（默认禁用）了安全检测
357     if (p_err == (OS_ERR *)0) {                       //如果错误类型实参为空
358         OS_SAFETY_CRITICAL_EXCEPTION();               //执行安全检测异常函数
359         return;                                       //返回，函数执行不成功
360     }
361 #endif
362
363 #if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u                //如果使能（默认使能）了中断中非法调用检测
364     if (OSIntNestingCtr > (OS_NESTING_CTR)0u) {       //如果函数是在中断中被调用
365         *p_err = OS_ERR_TIME_DLY_RESUME_ISR;          //错误类型为“在中断中结束延时”
366         return;                                       //返回，函数执行不成功
367     }
368 #endif
369
370 #if OS_CFG_ARG_CHK_EN > 0u                            //如果使能（默认使能）了参数检测功能
371     if (p_tcb == (OS_TCB *)0) {                       //如果任务为空
372         *p_err = OS_ERR_TASK_NOT_DLY;                 //错误类型为“任务不在延时”
373         return;                                       //返回，函数执行不成功
374     }
375 #endif
376
377     CPU_CRITICAL_ENTER();                             //关中断
378     if (p_tcb == OSTCBCurPtr) {                       //如果任务为当前任务
379         *p_err = OS_ERR_TASK_NOT_DLY;                 //错误类型为“任务不在延时”
380         CPU_CRITICAL_EXIT();                          //开中断
381         return;                                       //返回，函数执行不成功
382     }
```

os_cfg.h | app.c | os_time.c | os.h | os_tick.c

```
383
384     switch (p_tcb->TaskState) {                                 //根据任务的任务状态分类处理
385         case OS_TASK_STATE_RDY:                                 //如果任务处于就绪状态
386              CPU_CRITICAL_EXIT();                               //开中断
387             *p_err = OS_ERR_TASK_NOT_DLY;                       //错误类型为“任务不在延时”
388              break;
389
390         case OS_TASK_STATE_DLY:                                 //如果任务为延时状态
391              OS_CRITICAL_ENTER_CPU_EXIT();                      //进入临界段并开中断
392              p_tcb->TaskState = OS_TASK_STATE_RDY;              //修改任务为就绪状态
393              OS_TickListRemove(p_tcb);                          //将该任务从节拍列表移除
394              OS_RdyListInsert(p_tcb);                           //将该任务插入到就绪列表
395              OS_CRITICAL_EXIT_NO_SCHED();                       //退出临界段（无调度）
396             *p_err = OS_ERR_NONE;                               //错误类型为“无错误”
397              break;
398
399         case OS_TASK_STATE_PEND:                                //如果任务为无期限等待状态
400              CPU_CRITICAL_EXIT();                               //开中断
401             *p_err = OS_ERR_TASK_NOT_DLY;                       //错误类型为“任务不在延时”
402              break;
403
404         case OS_TASK_STATE_PEND_TIMEOUT:                        //如果任务为有期限等待状态
405              CPU_CRITICAL_EXIT();                               //开中断
406             *p_err = OS_ERR_TASK_NOT_DLY;                       //错误类型为“任务不在延时”
407              break;
408
409         case OS_TASK_STATE_SUSPENDED:                           //如果任务为被挂起状态
410              CPU_CRITICAL_EXIT();                               //开中断
411             *p_err = OS_ERR_TASK_NOT_DLY;                       //错误类型为“任务不在延时”
412              break;
```

os_cfg.h | app.c | os_time.c | os.h | os_tick.c

```
413
414         case OS_TASK_STATE_DLY_SUSPENDED:                       //如果任务为延时中被挂起状态
415              OS_CRITICAL_ENTER_CPU_EXIT();                      //进入临界段并开中断
416              p_tcb->TaskState = OS_TASK_STATE_SUSPENDED;        //修改任务为被挂起状态
417              OS_TickListRemove(p_tcb);                          //将该任务从节拍列表移除
418              OS_CRITICAL_EXIT_NO_SCHED();                       //退出临界段（无调度）
419             *p_err            = OS_ERR_TASK_SUSPENDED;          //错误类型为“任务被挂起”
420              break;
421
422         case OS_TASK_STATE_PEND_SUSPENDED:                      //如果任务为无期限等待中被挂起状态
423              CPU_CRITICAL_EXIT();                               //开中断
424             *p_err = OS_ERR_TASK_NOT_DLY;                       //错误类型为“任务不在延时”
425              break;
426
427         case OS_TASK_STATE_PEND_TIMEOUT_SUSPENDED:              //如果任务为有期限等待中被挂起状态
428              CPU_CRITICAL_EXIT();                               //开中断
429             *p_err = OS_ERR_TASK_NOT_DLY;                       //错误类型为“任务不在延时”
430              break;
431
432         default:                                                //如果任务状态超范围
433              CPU_CRITICAL_EXIT();                               //开中断
434             *p_err = OS_ERR_STATE_INVALID;                      //错误类型为“任务状态非法”
435              break;
436     }
437
438     OSSched();                                                  //任务切换
439 }
440 #endif
```

图 4-6 OSTimeDlyResume () 函数

如果任务的任务状态中包括延时，就调用 OS_TickListRemove() 函数将任务从节拍列表中移除。OS_TickListRemove() 函数的定义位于“os_tick.c”。

os_cfg.h | app.c | os_time.c | os.h | os_tick.c

```
337  void  OS_TickListRemove (OS_TCB  *p_tcb)                    //把任务从节拍列表移除
338  {
339      OS_TICK_SPOKE  *p_spoke;
340      OS_TCB         *p_tcb1;
341      OS_TCB         *p_tcb2;
342
343
344
345      p_spoke = p_tcb->TickSpokePtr;                          //获取任务位于哪个节拍列表
346      if (p_spoke != (OS_TICK_SPOKE *)0) {                    //如果任务的确在节拍列表中
347          p_tcb->TickRemain = (OS_TICK)0u;                    //将任务的延时还需时钟节拍数置0
348          if (p_spoke->FirstPtr == p_tcb) {                   //如果任务为节拍列表的第一个任务
349              p_tcb1           = (OS_TCB *)p_tcb->TickNextPtr; //获取任务的后一个任务为 p_tcb1
350              p_spoke->FirstPtr = p_tcb1;                     //把 p_tcb1 作为节拍列表的第一个任务
351              if (p_tcb1 != (OS_TCB *)0) {                    //如果 p_tcb1 非空
352                  p_tcb1->TickPrevPtr = (OS_TCB *)0;          //p_tcb1 前面已不存在任务
353              }
354          } else {                                            //如果任务不为节拍列表的第一个任务
355              p_tcb1              = p_tcb->TickPrevPtr;       //获取任务的前一个任务为 p_tcb1
356              p_tcb2              = p_tcb->TickNextPtr;       //获取任务的后一个任务为 p_tcb2
357              p_tcb1->TickNextPtr = p_tcb2;                   //将 p_tcb2 作为 p_tcb1 的后一个任务
358              if (p_tcb2 != (OS_TCB *)0) {                    //如果 p_tcb2 非空
359                  p_tcb2->TickPrevPtr = p_tcb1;               //把 p_tcb1 作为 p_tcb2 的前一个任务
360              }
361          }
362          p_tcb->TickNextPtr  = (OS_TCB        *)0;           //清空任务的后一个任务
363          p_tcb->TickPrevPtr  = (OS_TCB        *)0;           //清空任务的前一个任务
364          p_tcb->TickSpokePtr = (OS_TICK_SPOKE *)0;           //任务不再属于任何节拍列表
365          p_tcb->TickCtrMatch = (OS_TICK        )0u;          //将任务的时钟节拍的匹配变量置0
366          p_spoke->NbrEntries--;                              //节拍列表中的元素数目减1
367      }
368  }
```

图 4-7 OS_TickListRemove() 函数

4.1.4 OSTimeGet ()

OSTimeGet () 函数用于获取当前的时钟节拍计数值。OSTimeGet () 函数的信息如下表所示。

表 7 OSTimeGet ()

函数原型	OS_TICK OSTimeGet (OS_ERR *p_err);			
功能	获取当前的时钟节拍计数值。			
参数	p_err	返回错误类型	OS_ERR_NONE	没错误。
返回值	当前的时钟节拍计数值。			

OSTimeGet () 函数的定义位于“os_time.c”。

```
457 OS_TICK  OSTimeGet (OS_ERR  *p_err) //获取当前的时钟节拍计数值
458 {
459     OS_TICK  ticks;
460     CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必需该宏，该宏声明和定义一个局部变
461                     //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）
462                     //，开中断时将该值还原。
463
464 #ifdef OS_SAFETY_CRITICAL               //如果使能（默认禁用）了安全检测
465     if (p_err == (OS_ERR *)0) {         //如果错误类型实参为空
466         OS_SAFETY_CRITICAL_EXCEPTION(); //执行安全检测异常函数
467         return ((OS_TICK)0);            //返回0，函数执行不成功
468     }
469 #endif
470
471     CPU_CRITICAL_ENTER();               //关中断
472     ticks = OSTickCtr;                  //获取当前的时钟节拍计数值
473     CPU_CRITICAL_EXIT();                //开中断
474    *p_err = OS_ERR_NONE;                //错误类型为“无错误”
475     return (ticks);                     //返回当前的时钟节拍计数值
476 }
```

图 4-8 OSTimeGet () 函数

4.1.5 OSTimeSet ()

OSTimeSet () 函数用于设置当前的时钟节拍计数值。OSTimeSet () 函数的信息如下表所示。

表 8 OSTimeSet ()

函数原型	void OSTimeSet (OS_TICK ticks, OS_ERR *p_err);			
功能	设置当前的时钟节拍计数值。			
参数	ticks	时钟节拍数。		
	p_err	返回错误类型	OS_ERR_NONE	没错误。
返回值	无。			
注意事项	虽然 uC/OS-III 允许该操作，但建议慎用该函数。			

OSTimeSet () 函数的定义位于“os_time.c”。

```
494 void  OSTimeSet (OS_TICK   ticks,  //设置当前的时钟节拍计数值
495                  OS_ERR   *p_err)  //返回错误类型
496 {
497     CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必需该宏，该宏声明和定义一个局部变
498                     //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）
499                     //，开中断时将该值还原。
500
501 #ifdef OS_SAFETY_CRITICAL               //如果使能（默认禁用）了安全检测
502     if (p_err == (OS_ERR *)0) {         //如果错误类型实参为空
503         OS_SAFETY_CRITICAL_EXCEPTION(); //执行安全检测异常函数
504         return;                         //返回，函数执行不成功
505     }
506 #endif
507
508     CPU_CRITICAL_ENTER();               //关中断
509     OSTickCtr = ticks;                  //设置当前的时钟节拍计数值
510     CPU_CRITICAL_EXIT();                //开中断
511    *p_err     = OS_ERR_NONE;            //错误类型为“无错误”
512 }
```

图 4-9 OSTimeSet () 函数

4.2 实例演示

4.2.1 实例 1

本节实例创建三个应用任务，分别为 LED1 任务、LED2 任务和 LED3 任务。LED1 任务调用 OSTimeDly() 函数相对性延时 1s 切换一次 LED1 的亮灭状态，LED2 任务 OSTimeDly() 函数周期性延时 5s 切换一次 LED2 的亮灭状态，LED3 任务 OSTimeDlyHMSM () 函数相对性延时 10s 切换一次 LED3 的亮灭状态。该例程已经存放在配套资料的下图路径。

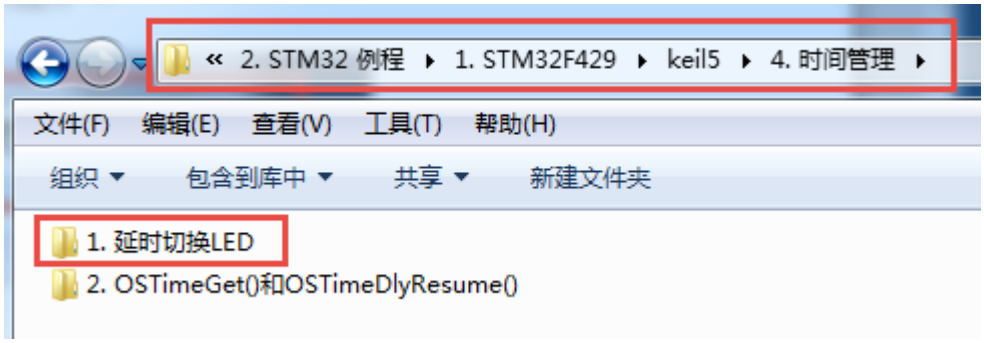

图 4-10 例程路径

本例程需用使用 LED，所以工程中需要添加改 LED 的驱动文件到“...\User\BSP”路径下，在 MDK 上添加驱动文件的源文件和头文件路径，再在“bsp.h”中包含驱动文件的头文件，然后在“bsp.c”的 BSP_Init() 函数的定义体内添加 LED 初始化函数。

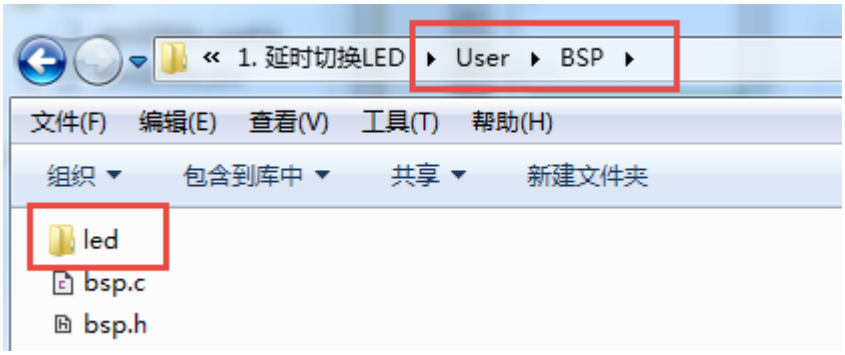

图 4-11 板级驱动文件路径

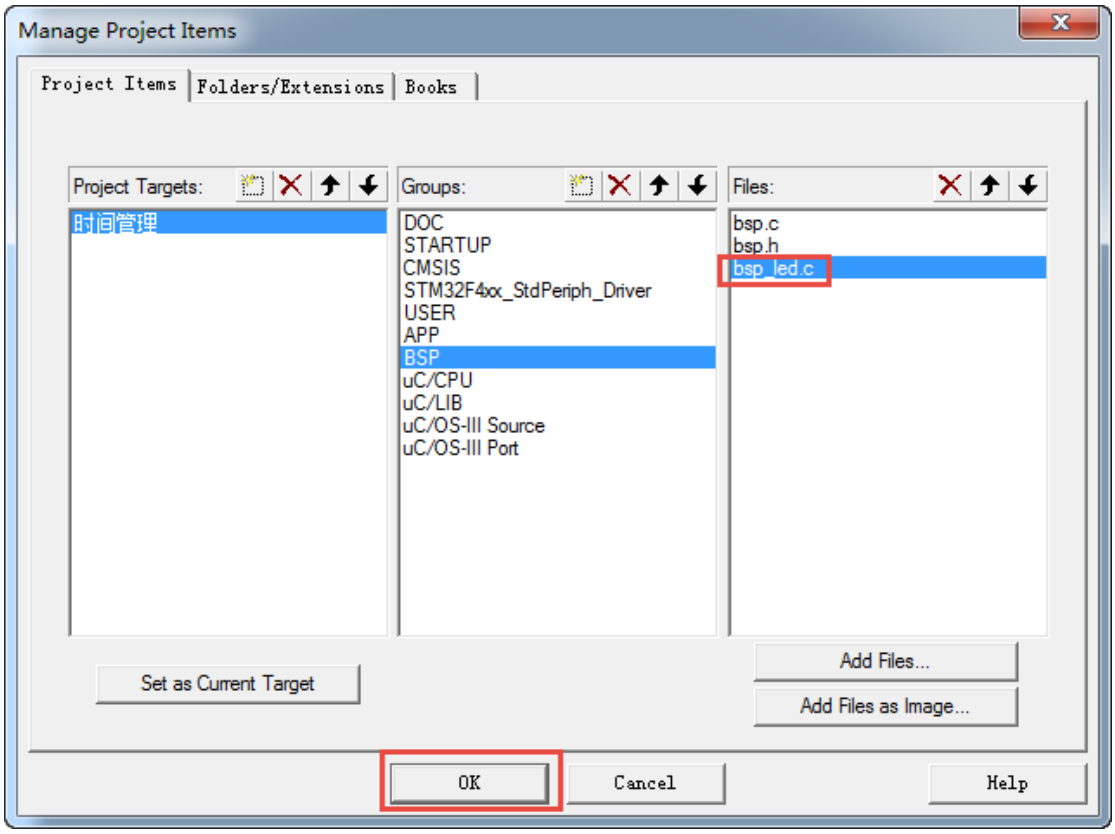

图 4-12 在 MDK 增加驱动文件的源文件到组件

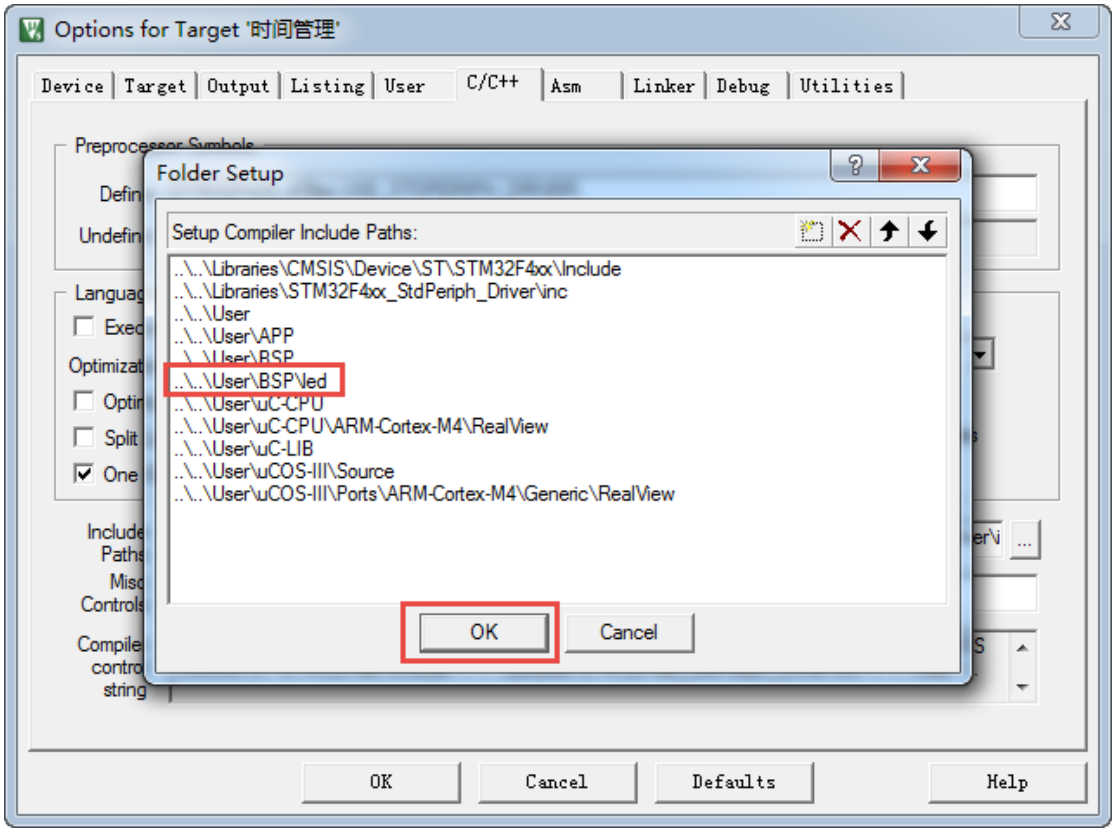

图 4-13 在 MDK 增加驱动文件的头文件路径

```
app_cfg.h | app.c | bsp.c | bsp.h
61  *********************************************************************
62  *                                                  INCLUDE FILES
63  *********************************************************************
64  */
65
66  #include  <stdio.h>
67  #include  <stdarg.h>
68
69  #include  <cpu.h>
70  #include  <cpu_core.h>
71
72  #include  <lib_def.h>
73  #include  <lib_ascii.h>
74
75
76  #include "stm32f4xx.h"
77
78  #include "bsp_led.h"        添加板载驱动文件的头文件
79
80
```

图 4-14 包含板级驱动文件的头文件

```
bsp.c | bsp.h | app_cfg.h | app.c
161  void  BSP_Init (void)
162  {
163    LED_Init ();    //初始化 LED
164
165  }
```

图 4-15 添加板级初始化函数

在“app.c”中，新建了任务 AppTaskLed1()、AppTaskLed2() 和 AppTaskLed1()。任务 AppTaskLed1() 在 while 死循环里每隔 1000 个时钟节拍（1s）切换一次 LED1 的亮灭状态。任务 AppTaskLed2() 在 while 死循环里每隔 5000 个时钟节拍（5s）切换一次 LED2 的亮灭状态。任务 AppTaskLed3() 在 while 死循环里每隔 10s 切换一次 LED3 的亮灭状态。

```
bsp.c | bsp.h | app_cfg.h | app.c
220 /*
221 *********************************************************************************************
222 *                                              LED1 TASK
223 *********************************************************************************************
224 */
225
226 static  void  AppTaskLed1 ( void * p_arg )
227 {
228     OS_ERR      err;
229
230
231     (void)p_arg;
232
233
234     while (DEF_TRUE) {                                  //任务体，通常写成一个死循环
235       macLED1_TOGGLE ();                                //切换 LED1 的亮灭状态
236       OSTimeDly ( 1000, OS_OPT_TIME_DLY, & err );       //相对性延时1000个时钟节拍（1s）
237     }
238
239
240 }
```

图 4-16 AppTaskLed1() 任务函数

app.c os_time.c

```
/*
*********************************************************************************************************
*                                                LED2 TASK
*********************************************************************************************************
*/

static  void  AppTaskLed2 ( void * p_arg )
{
    OS_ERR      err;


   (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
      OSTimeDly ( 5000, OS_OPT_TIME_PERIODIC, & err );          //周期性延时5000个时钟节拍（5s）
    }


}
```

图 4-17 AppTaskLed2() 任务函数

bsp.c bsp.h app_cfg.h app.c

```
/*
*********************************************************************************************************
*                                                LED3 TASK
*********************************************************************************************************
*/

static  void  AppTaskLed3 ( void * p_arg )
{
    OS_ERR      err;


   (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
      OSTimeDlyHMSM ( 0, 0, 10, 0, OS_OPT_TIME_DLY, & err );    //相对性延时10s
    }


}
```

图 4-18 AppTaskLed3() 任务函数

程序在主函数 main() 中创建起始任务，起始任务运行时会创建 AppTaskTest() 任务，然后删除自身，之后任务 AppTaskLed1()、AppTaskLed2() 和 AppTaskLed1() 并行运行。

app_cfg.h | app.c

```
93  int  main (void)
94  {
95      OS_ERR  err;
96
97
98      OSInit(&err);                                               //初始化 uC/OS-III
99
100     /* 创建起始任务 */
101     OSTaskCreate((OS_TCB     *)&AppTaskStartTCB,                //任务控制块地址
102                  (CPU_CHAR   *)"App Task Start",                //任务名称
103                  (OS_TASK_PTR ) AppTaskStart,                   //任务函数
104                  (void       *) 0,                              //传递给任务函数（形参p_arg）的实参
105                  (OS_PRIO     ) APP_TASK_START_PRIO,            //任务的优先级
106                  (CPU_STK    *)&AppTaskStartStk[0],             //任务堆栈的基地址
107                  (CPU_STK_SIZE) APP_TASK_START_STK_SIZE / 10,   //任务堆栈空间剩下1/10时限制其增长
108                  (CPU_STK_SIZE) APP_TASK_START_STK_SIZE,        //任务堆栈空间（单位：sizeof(CPU_STK)）
109                  (OS_MSG_QTY  ) 5u,                             //任务可接收的最大消息数
110                  (OS_TICK     ) 0u,                             //任务的时间片节拍数（0表默认值OSCfg_TickRate_Hz/10）
111                  (void       *) 0,                              //任务扩展（0表不扩展）
112                  (OS_OPT      )(OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR), //任务选项
113                  (OS_ERR     *)&err);                           //返回错误类型
114
115     OSStart(&err);                                              //启动多任务管理（交由uC/OS-III控制）
116
117
118 }
```

图 4-19 主函数

bsp.c | bsp.h | app_cfg.h | app.c

```
143 static  void  AppTaskStart (void *p_arg)
144 {
145     CPU_INT32U  cpu_clk_freq;
146     CPU_INT32U  cnts;
147     OS_ERR      err;
148
149
150     (void)p_arg;
151
152     BSP_Init();                                                 //板级初始化
153     CPU_Init();                                                 //初始化 CPU 组件（时间戳、关中断时间测量和主机名）
154
155     cpu_clk_freq = BSP_CPU_ClkFreq();                           //获取 CPU 内核时钟频率（SysTick 工作时钟）
156     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
157     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
158
159     Mem_Init();                                                 //初始化内存管理组件（堆内存池和内存池表）
160
161 #if OS_CFG_STAT_TASK_EN > 0u                                    //如果使能（默认使能）了统计任务
162     OSStatTaskCPUUsageInit(&err);                               //计算没有应用任务（只有空闲任务）运行时 CPU 的（最大）
163 #endif                                                          //容量（决定 OS_Stat_IdleCtrMax 的值，为后面计算 CPU
164                                                                 //使用率使用）。
165     CPU_IntDisMeasMaxCurReset();                                //复位（清零）当前最大关中断时间
166
```

bsp.c | bsp.h | app_cfg.h | app.c

```
167
168     /* 创建 LED1 任务 */
169     OSTaskCreate((OS_TCB     *)&AppTaskLed1TCB,                                //任务控制块地址
170                  (CPU_CHAR   *)"App Task Led1",                                //任务名称
171                  (OS_TASK_PTR ) AppTaskLed1,                                   //任务函数
172                  (void       *) 0,                                             //传递给任务函数（形参p_arg）的实参
173                  (OS_PRIO     ) APP_TASK_LED1_PRIO,                            //任务的优先级
174                  (CPU_STK    *)&AppTaskLed1Stk[0],                             //任务堆栈的基地址
175                  (CPU_STK_SIZE) APP_TASK_LED1_STK_SIZE / 10,                   //任务堆栈空间剩下1/10时限制其增长
176                  (CPU_STK_SIZE) APP_TASK_LED1_STK_SIZE,                        //任务堆栈空间（单位：sizeof(CPU_STK)）
177                  (OS_MSG_QTY  ) 5u,                                            //任务可接收的最大消息数
178                  (OS_TICK     ) 0u,                                            //任务的时间片节拍数（0表默认值OSCfg_TickRate_Hz/10）
179                  (void       *) 0,                                             //任务扩展（0表不扩展）
180                  (OS_OPT      )(OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),    //任务选项
181                  (OS_ERR     *)&err);                                          //返回错误类型
182
183     /* 创建 LED2 任务 */
184     OSTaskCreate((OS_TCB     *)&AppTaskLed2TCB,                                //任务控制块地址
185                  (CPU_CHAR   *)"App Task Led2",                                //任务名称
186                  (OS_TASK_PTR ) AppTaskLed2,                                   //任务函数
187                  (void       *) 0,                                             //传递给任务函数（形参p_arg）的实参
188                  (OS_PRIO     ) APP_TASK_LED2_PRIO,                            //任务的优先级
189                  (CPU_STK    *)&AppTaskLed2Stk[0],                             //任务堆栈的基地址
190                  (CPU_STK_SIZE) APP_TASK_LED2_STK_SIZE / 10,                   //任务堆栈空间剩下1/10时限制其增长
191                  (CPU_STK_SIZE) APP_TASK_LED2_STK_SIZE,                        //任务堆栈空间（单位：sizeof(CPU_STK)）
192                  (OS_MSG_QTY  ) 5u,                                            //任务可接收的最大消息数
193                  (OS_TICK     ) 0u,                                            //任务的时间片节拍数（0表默认值OSCfg_TickRate_Hz/10）
194                  (void       *) 0,                                             //任务扩展（0表不扩展）
195                  (OS_OPT      )(OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),    //任务选项
196                  (OS_ERR     *)&err);                                          //返回错误类型
197
198     /* 创建 LED3 任务 */
199     OSTaskCreate((OS_TCB     *)&AppTaskLed3TCB,                                //任务控制块地址
200                  (CPU_CHAR   *)"App Task Led3",                                //任务名称
201                  (OS_TASK_PTR ) AppTaskLed3,                                   //任务函数
202                  (void       *) 0,                                             //传递给任务函数（形参p_arg）的实参
203                  (OS_PRIO     ) APP_TASK_LED3_PRIO,                            //任务的优先级
204                  (CPU_STK    *)&AppTaskLed3Stk[0],                             //任务堆栈的基地址
205                  (CPU_STK_SIZE) APP_TASK_LED3_STK_SIZE / 10,                   //任务堆栈空间剩下1/10时限制其增长
206                  (CPU_STK_SIZE) APP_TASK_LED3_STK_SIZE,                        //任务堆栈空间（单位：sizeof(CPU_STK)）
207                  (OS_MSG_QTY  ) 5u,                                            //任务可接收的最大消息数
208                  (OS_TICK     ) 0u,                                            //任务的时间片节拍数（0表默认值OSCfg_TickRate_Hz/10）
209                  (void       *) 0,                                             //任务扩展（0表不扩展）
210                  (OS_OPT      )(OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),    //任务选项
211                  (OS_ERR     *)&err);                                          //返回错误类型
212
213
214     OSTaskDel ( & AppTaskStartTCB, & err );                    //删除起始任务本身，该任务不再运行
215
216
217 }
```

图 4-20 起始任务 AppTaskStart()

编译和下载程序到秉火 STM32 开发板，运行程序。用户可以分别单独观察三个 LED，LED1 每隔 1s 切换一次亮灭状态，LED2 每隔 5s 切换一次亮灭状态，LED3 每隔 10s 切换一次亮灭状态。这说明任务的延时起到了作用。另外，若以 LED1 的切换作为参考，可以发现 LED1 没切换 5 次，LED2 切换一次，这说明周期性延时和相对性延时基本一样。但在实际应用中，如果是很长时间的延时，一般会采用周期性延时，因为它会进行自动调整，避免长时间的累积误差，而比较短暂的延时就采用相对性延时，其他的选项一般很少用。

4.2.2 实例 2

本节例程以实例 1 为基础，在 LED1 任务中增加 OSTimeGet() 读取当前时钟节拍计数值，并使用 USART1 打印出来；在 LED2 任务中增加 OSTimeDlyResume() 函数结束 LED3 任务的延时，使 LED3 随同 LED2 一起切换，而不再时 10s 切换一次。该例程已经存放在配套资料的下图路径。

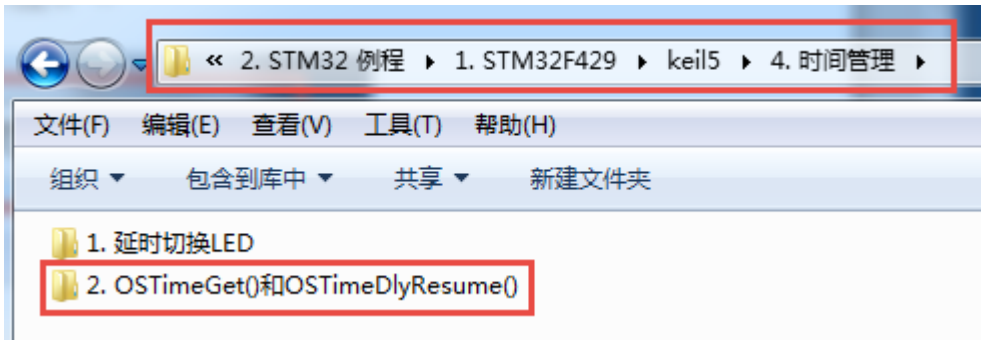

图 4-21 例程路径

本例程需用使用 LED 和 USART1，所以工程中需要添加其驱动文件和初始化。用户可参照实例 1，这里不再赘述。

任务函数 AppTaskLed1() 和 AppTaskLed1() 需要改动，AppTaskLed3() 不变。

os_time.c | os.h | bsp.c | bsp.h | bsp_usart1.c | app.c

```
/*
*********************************************************************************************************
*                                              LED1 TASK
*********************************************************************************************************
*/

static  void  AppTaskLed1 ( void * p_arg )
{
  OS_ERR      err;
  OS_TICK     ticks;
  CPU_SR_ALLOC();            //使用到临界段（在关/开中断时）时必需该宏，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。
  (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

    OS_CRITICAL_ENTER();                                //进入临界段，不希望下面串口打印遭到中断

    ticks = OSTimeGet ( & err );                        //获取当前时钟节拍计数值

    printf ( "\r\n当前时钟节拍计数值为：%d", ticks );    //打印当前时钟节拍计数值

    OS_CRITICAL_EXIT();                                 //进入临界段，不希望下面串口打印遭到中断

    OSTimeDly ( 1000, OS_OPT_TIME_DLY, & err );         //相对性延时1000个时钟节拍（1s）

  }


}
```

图 4-22 AppTaskLed1() 任务函数

os_time.c | os.h | bsp.c | bsp.h | bsp_usart1.c | app.c

```
/*
*********************************************************************************************************
*                                                LED2 TASK
*********************************************************************************************************
*/

static  void  AppTaskLed2 ( void * p_arg )
{
  OS_ERR      err;


  (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

    OSTimeDlyResume ( &AppTaskLed3TCB, & err );         //结束任务 AppTaskLed3 的延时

    OSTimeDly ( 5000, OS_OPT_TIME_PERIODIC, & err ); //周期性延时5000个时钟节拍（5s）

  }


}
```

图 4-23 AppTaskLed2() 任务函数

把 STM32 的 USART1 连接至电脑的串口调试助手，编译和下载程序到秉火 STM32 开发板，运行程序。用户可以串口调试助手观察到打印的当前的时钟节拍计数值，打印的当前的时钟节拍计数值每次递增 1000，与程序延时的 1000 个时钟节拍吻合。另外，用户可以观察到 LED3 不再是每隔 10s 切换一次亮灭状态，而是随同 LED2 每隔 5s 切换一次，这说明 LED2 任务中的 OSTimeDlyResume () 函数起到了作用。

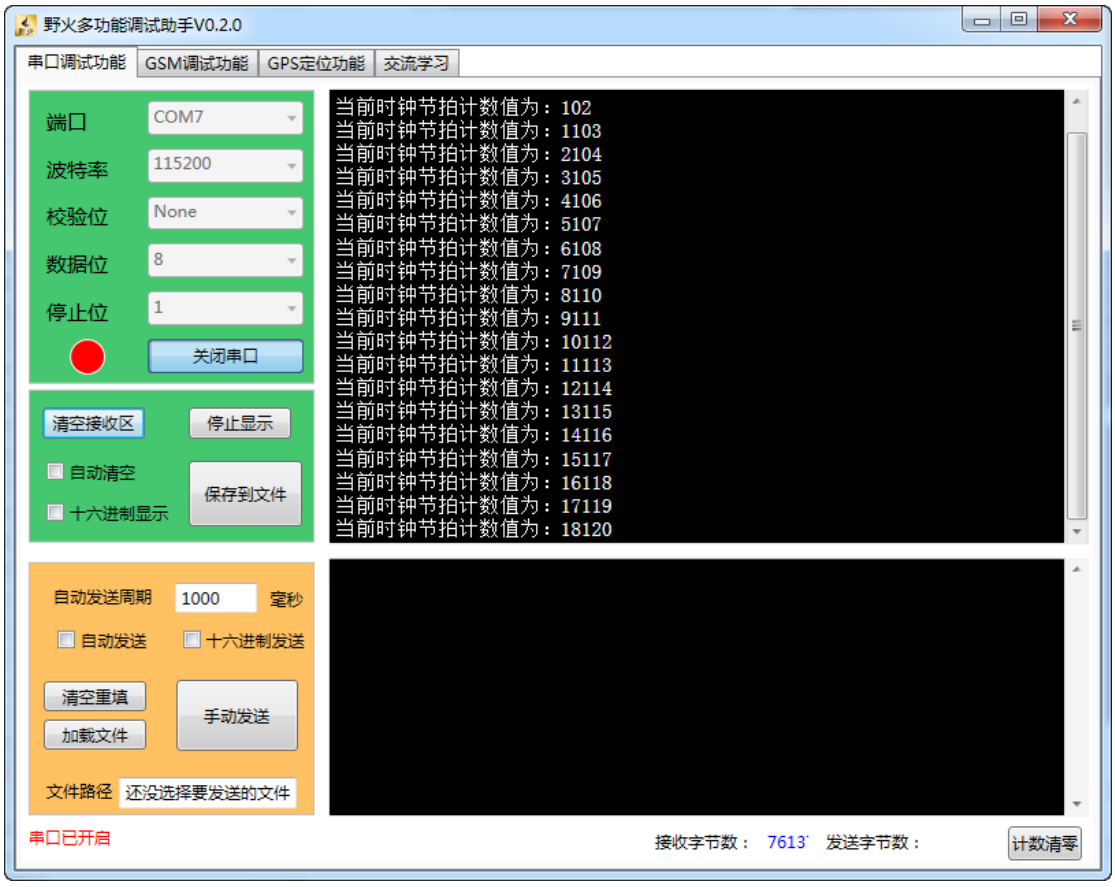

图 4-24 串口调试助手

4.3 章末总结

本章阐述了时间管理的工作原理，时间管理主要负责管理任务的延时和有期限等待。当任务要延时或有期限等待时，就会被插入到节拍列表里进行管理。每当时钟节拍到来时，系统就检查节拍列表中是否有任务的延时或者等待已经到期。如果有任务已经期满，则改变该任务的状态，去掉它包含的延时或等待状态，并将任务从节拍列表移除。

uC/OS 系统的时间事件主要有延时和等待。本章侧重讲解延时，等待主要属于信号量和消息队列的范畴，在后面相关章节侧重讲解。延时的函数主要有 OSTimeDly() 和 OSTimeDlyHMSM()。前者以时钟节拍作为时间参数，后者直接以时分秒和毫秒作为时间参数，具有更好的交互性。其实两者的原理都一样，OSTimeDlyHMSM() 函数的时间参数最终也是要转换成节拍数的。这两个函数的选项参数一般使用 OS_OPT_TIME_DLY 或 OS_OPT_TIME_PERIODIC。短时间延时通常使用 OS_OPT_TIME_DLY，短时间延时通常使用 OS_OPT_TIME_PERIODIC。

OSTimeDlyResume () 函数可以用于结束其他任务因调用 OSTimeDly() 或 OSTimeDlyHMSM() 导致的延时。但必须切记，OSTimeDlyResume () 函数的操作对象为其他任务，不能是当前任务。

OSTimeGet () 函数用于获取当前的时钟节拍计数值。OSTimeSet () 函数用于设置当前的时钟节拍计数值，但必须慎用该函数。

第5章 软件定时器

软件定时器是 uC/OS 操作系统的一个内核对象，软件定时器是基于时钟节拍和系统管理创建的软件性定时器，理论上可以创建无限多个，但精准度肯定比硬件定时稍逊一筹。使用硬件定时器往往需要查阅芯片的相关数据手册，比较繁琐，而使用 uC/OS 的软件定时非常方便。

5.1 原理简述

软件定时器启动之后是由软件定时器任务 OS_TmrTask() 统一管理，在创建软件定时器之前必须先使能软件定时器和配置软件定时器的相关参数。

软件定时器的使能位于“os_cfg.h”。

os_tmr.c | app.c | os_cfg.h

```
100                                                  /* ------------------------ TIMER MANAGEMENT
101 #define OS_CFG_TMR_EN                    1u     //使能/禁用软件定时器
102 #define OS_CFG_TMR_DEL_EN                1u     //使能/禁用 OSTmrDel() 函数
103
```

图 5-1

其有关参数的配置位于“os_cpu_app.h”。

os_cfg_app.h | app.c

```
65
66                                                   /* ---------------------- TIMERS ---------------------- */
67 #define  OS_CFG_TMR_TASK_PRIO             11u    //定时器任务的优先级
68 #define  OS_CFG_TMR_TASK_RATE_HZ          10u    //定时器的时基（一般不能大于 OS_CFG_TICK_RATE_HZ ）
69 #define  OS_CFG_TMR_TASK_STK_SIZE        128u    //定时器任务的栈空间大小（单位：CPU_STK）
70 #define  OS_CFG_TMR_WHEEL_SIZE            17u    // OSCfg_TmrWheel 数组的大小，推荐使用任务总数/4，且为质数
71
```

图 5-2

5.1.1 OSTmrCreate ()

要使用 uC/OS 的软件定时器必须先声明和创建软件定时器，调用 OSTmrCreate () 函数可以创建一个软件定时器。OSTmrCreate () 函数的信息如下表所示。

表 9 OSTimeDly()

函数原型	void OSTmrCreate (OS_TMR *p_tmr, CPU_CHAR *p_name, OS_TICK dly, OS_TICK period, OS_OPT opt, OS_TMR_CALLBACK_PTR p_callback, void *p_callback_arg, OS_ERR *p_err);
功能	创建一个软件定时器。

<table>
<tr><td rowspan="15">参数</td><td>p_tmr</td><td colspan="3">定时器控制块指针。</td></tr>
<tr><td>p_name</td><td colspan="3">定时器名称。</td></tr>
<tr><td>dly</td><td colspan="3">初始定时节拍数。</td></tr>
<tr><td>period</td><td colspan="3">周期定时重载节拍数。</td></tr>
<tr><td rowspan="2">opt</td><td rowspan="2">选项</td><td>OS_OPT_TMR_ONE_SHOT</td><td>周期性定时。</td></tr>
<tr><td>OS_OPT_TMR_PERIODIC</td><td>一次性定时。</td></tr>
<tr><td>p_callback</td><td colspan="3">定时到期时的回调函数。</td></tr>
<tr><td>p_callback_arg</td><td colspan="3">传给回调函数的参数。</td></tr>
<tr><td rowspan="9">p_err</td><td rowspan="9">返回错误类型</td><td>OS_ERR_NONE</td><td>没错误</td></tr>
<tr><td>OS_ERR_ILLEGAL_CREATE_RUN_TIME</td><td>非法创建内核对象</td></tr>
<tr><td>OS_ERR_OBJ_CREATED</td><td>该定时器已被创建过</td></tr>
<tr><td>OS_ERR_OBJ_PTR_NULL</td><td>定时器对象为空</td></tr>
<tr><td>OS_ERR_OBJ_TYPE</td><td>定时器对象无效</td></tr>
<tr><td>OS_ERR_OPT_INVALID</td><td>选项不可用。</td></tr>
<tr><td>OS_ERR_TMR_INVALID_DLY</td><td>定时初始实参无效</td></tr>
<tr><td>OS_ERR_TMR_INVALID_PERIOD</td><td>周期重载实参无效</td></tr>
<tr><td>OS_ERR_TMR_ISR</td><td>在中断函数中定时</td></tr>
<tr><td>返回值</td><td colspan="3">无。</td></tr>
<tr><td>注意事项</td><td colspan="3">✧ 创建前必须先为 p_tmr 声明一个软件定时器对象（OS_TMR）。
✧ 一次性定时时 dly 不能为 0。
✧ 周期性定时时 period 不能为 0。
✧ 不可以在中断中调用该函数。</td></tr>
</table>

OSTmrCreate () 函数的定义位于“os_tmr.c”。

os_tmr.c | os_time.c | os_cfg.h | app.c | os.h | os_core.c | lib_def.h

```
104 void  OSTmrCreate (OS_TMR                *p_tmr,           //定时器控制块指针
105                   CPU_CHAR              *p_name,          //命名定时器，有助于调试
106                   OS_TICK                dly,             //初始定时节拍数
107                   OS_TICK                period,          //周期定时重载节拍数
108                   OS_OPT                 opt,             //选项
109                   OS_TMR_CALLBACK_PTR    p_callback,      //定时到期时的回调函数
110                   void                  *p_callback_arg,  //传给回调函数的参数
111                   OS_ERR                *p_err)           //返回错误类型
112 {
113     CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必需该宏，该宏声明和定义一个局部变
114                     //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）
115                     //，开中断时将该值还原。
116
117 #ifdef OS_SAFETY_CRITICAL                 //如果使能（默认禁用）了安全检测
118     if (p_err == (OS_ERR *)0) {           //如果错误类型实参为空
119         OS_SAFETY_CRITICAL_EXCEPTION(); //执行安全检测异常函数
120         return;                           //返回，不执行定时操作
121     }
122 #endif
```

```
123
124 #ifdef OS_SAFETY_CRITICAL_IEC61508          //如果使能（默认禁用）了安全关键
125     if (OSSafetyCriticalStartFlag == DEF_TRUE) { //如果是在调用 OSSafetyCriticalStart() 后创建该定时器
126         *p_err = OS_ERR_ILLEGAL_CREATE_RUN_TIME;  //错误类型为“非法创建内核对象”
127         return;                                   //返回，不执行定时操作
128     }
129 #endif
130
131 #if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u      //如果使能（默认使能）了中断中非法调用检测
132     if (OSIntNestingCtr > (OS_NESTING_CTR)0) {  //如果该函数是在中断中被调用
133         *p_err = OS_ERR_TMR_ISR;              //错误类型为“在中断函数中定时”
134         return;                               //返回，不执行定时操作
135     }
136 #endif
```

```
137
138 #if OS_CFG_ARG_CHK_EN > 0u                          //如果使能（默认使能）了参数检测
139     if (p_tmr == (OS_TMR *)0) {                     //如果参数 p_tmr 为空
140         *p_err = OS_ERR_OBJ_PTR_NULL;               //错误类型为“定时器对象为空”
141         return;                                     //返回，不执行定时操作
142     }
143
144     switch (opt) {                                  //根据延时选项参数 opt 分类操作
145         case OS_OPT_TMR_PERIODIC:                   //如果选择周期性定时
146             if (period == (OS_TICK)0) {             //如果周期重载实参为0
147                 *p_err = OS_ERR_TMR_INVALID_PERIOD; //错误类型为“周期重载实参无效”
148                 return;                             //返回，不执行定时操作
149             }
150             break;
151
152         case OS_OPT_TMR_ONE_SHOT:                   //如果选择一次性定时
153             if (dly == (OS_TICK)0) {                //如果定时初始实参为0
154                 *p_err = OS_ERR_TMR_INVALID_DLY;    //错误类型为“定时初始实参无效”
155                 return;                             //返回，不执行定时操作
156             }
157             break;
158
159         default:                                    //如果选项超出预期
160             *p_err = OS_ERR_OPT_INVALID;            //错误类型为“选项非法”
161             return;                                 //返回，不执行定时操作
162     }
163 #endif
```

```
164 {
165     OS_CRITICAL_ENTER();                //进入临界段
166     p_tmr->State           = (OS_STATE           )OS_TMR_STATE_STOPPED;  //初始化定时器指标
167     p_tmr->Type            = (OS_OBJ_TYPE        )OS_OBJ_TYPE_TMR;
168     p_tmr->NamePtr         = (CPU_CHAR          *)p_name;
169     p_tmr->Dly             = (OS_TICK            )dly;
170     p_tmr->Match           = (OS_TICK            )0;
171     p_tmr->Remain          = (OS_TICK            )0;
172     p_tmr->Period          = (OS_TICK            )period;
173     p_tmr->Opt             = (OS_OPT             )opt;
174     p_tmr->CallbackPtr     = (OS_TMR_CALLBACK_PTR)p_callback;
175     p_tmr->CallbackPtrArg = (void               *)p_callback_arg;
176     p_tmr->NextPtr         = (OS_TMR            *)0;
177     p_tmr->PrevPtr         = (OS_TMR            *)0;
178
179 #if OS_CFG_DBG_EN > 0u              //如果使能（默认使能）了调试代码和变量
180     OS_TmrDbgListAdd(p_tmr);        //将该定时添加到定时器双向调试链表
181 #endif
182     OSTmrQty++;                     //定时器个数加1
183
184     OS_CRITICAL_EXIT_NO_SCHED(); //退出临界段（无调度）
185    *p_err = OS_ERR_NONE;         //错误类型为“无错误”
186 }
```

图 5-3 OSTmrCreate () 函数

5.1.2 OSTmrStart ()

创建完软件定时器后，如果要使用该软件定时器，需要调用 OSTmrStart () 函数启动该软件定时器。OSTmrStart () 函数的信息如下表所示。

表 10 OSTmrStart ()

函数原型	CPU_BOOLEAN OSTmrStart (OS_TMR *p_tmr, OS_ERR *p_err);			
功能	启动一个软件定时器。			
参数	p_tmr	定时器控制块指针。		
	p_err	返回错误类型	OS_ERR_NONE	没错误
			OS_ERR_ILLEGAL_CREATE_RUN_TIME	非法创建内核对象
			OS_ERR_OBJ_CREATED	该定时器已被创建过
			OS_ERR_OBJ_PTR_NULL	定时器对象为空
			OS_ERR_OBJ_TYPE	定时器对象无效
			OS_ERR_OPT_INVALID	选项不可用。
			OS_ERR_TMR_INVALID_DLY	定时初始实参无效
			OS_ERR_TMR_INVALID_PERIOD	周期重载实参无效
			OS_ERR_TMR_ISR	在中断函数中定时
返回值	✧ DEF_TRUE，执行成功。 ✧ DEF_FALSE，执行失败。			
注意事项	✧ 创建前必须先为 p_tmr 声明一个定时器对象（OS_TMR）。 ✧ 一次性定时时 dly 不能为 0。 ✧ 周期性定时时 period 不能为 0。 ✧ 不可以在中断中调用该函数。			

OSTmrCreate () 函数的定义也位于“os_tmr.c”。

os_tmr.c | os_time.c | os_cfg.h | app.c | os.h | os_core.c | lib_def.h | cpu_core.h | os_cfg_app.c

```
421 CPU_BOOLEAN  OSTmrStart (OS_TMR   *p_tmr,   //定时器控制块指针
422                          OS_ERR   *p_err)   //返回错误类型
423 {
424     OS_ERR       err;
425     CPU_BOOLEAN  success; //暂存函数执行结果
426
427
428
429 #ifdef OS_SAFETY_CRITICAL                  //如果使能（默认禁用）了安全检测
430     if (p_err == (OS_ERR *)0) {            //如果错误类型实参为空
431         OS_SAFETY_CRITICAL_EXCEPTION(); //执行安全检测异常函数
432         return (DEF_FALSE);                //返回 DEF_FALSE，不继续执行
433     }
434 #endif
435
436 #if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u          //如果使能（默认使能）了中断中非法调用检测
437     if (OSIntNestingCtr > (OS_NESTING_CTR)0) { //如果该函数是在中断中被调用
438        *p_err = OS_ERR_TMR_ISR;                //错误类型为“在中断函数中定时”
439         return (DEF_FALSE);                    //返回 DEF_FALSE，不继续执行
440     }
441 #endif


442
443 #if OS_CFG_ARG_CHK_EN > 0u              //如果使能（默认使能）了参数检测
444     if (p_tmr == (OS_TMR *)0) {         //如果使能 p_tmr 的实参为空
445        *p_err = OS_ERR_TMR_INVALID;     //错误类型为“无效的定时器”
446         return (DEF_FALSE);             //返回 DEF_FALSE，不继续执行
447     }
448 #endif
449
450 #if OS_CFG_OBJ_TYPE_CHK_EN > 0u             //如果使能（默认使能）了对象类型检测
451     if (p_tmr->Type != OS_OBJ_TYPE_TMR) { //如果该定时器的对象类型有误
452        *p_err = OS_ERR_OBJ_TYPE;            //错误类型为“对象类型错误”
453         return (DEF_FALSE);                 //返回 DEF_FALSE，不继续执行
454     }
455 #endif
```

```
456
457     OSSchedLock(&err);                                  //锁住调度器
458     switch (p_tmr->State) {                             //根据定时器的状态分类处理
459         case OS_TMR_STATE_RUNNING:                      //如果定时器正在运行，则重启
460              OS_TmrUnlink(p_tmr);                       //从定时器轮里移除该定时器
461              OS_TmrLink(p_tmr, OS_OPT_LINK_DLY);        //将该定时器重新插入到定时器轮
462              OSSchedUnlock(&err);                       //解锁调度器
463             *p_err   = OS_ERR_NONE;                     //错误类型为“无错误”
464              success = DEF_TRUE;                        //执行结果暂为 DEF_TRUE
465              break;
466
467         case OS_TMR_STATE_STOPPED:                      //如果定时器已被停止，则开启
468         case OS_TMR_STATE_COMPLETED:                    //如果定时器已完成了，则开启
469              OS_TmrLink(p_tmr, OS_OPT_LINK_DLY);        //将该定时器重新插入到定时器轮
470              OSSchedUnlock(&err);                       //解锁调度器
471             *p_err   = OS_ERR_NONE;                     //错误类型为“无错误”
472              success = DEF_TRUE;                        //执行结果暂为 DEF_TRUE
473              break;
474
475         case OS_TMR_STATE_UNUSED:                       //如果定时器未被创建
476              OSSchedUnlock(&err);                       //解锁调度器
477             *p_err   = OS_ERR_TMR_INACTIVE;             //错误类型为“定时器未激活”
478              success = DEF_FALSE;                       //执行结果暂为 DEF_FALSE
479              break;
480
481         default:                                        //如果定时器的状态超出预期
482              OSSchedUnlock(&err);                       //解锁调度器
483             *p_err = OS_ERR_TMR_INVALID_STATE;          //错误类型为“定时器无效”
484              success = DEF_FALSE;                       //执行结果暂为 DEF_FALSE
485              break;
486     }
487     return (success);                                   //返回执行结果
488 }
```

图 5-4 OSTmrStart () 函数

这里涉及到两个函数，OS_TmrLink() 和 OS_TmrUnlink()。所有的软件定时器是通过定时器轮来实现管理的，定时器轮与时钟节拍列表数组一样，就是有若干个定时器列表组成的数组。OS_TmrLink() 函数是将软件定时器插入到定时器轮的列表，相反 OS_TmrUnlink() 函数是将软件定时器从定时器轮的列表移除。该操作与时钟节拍插入和移除节拍列表类似。

OS_TmrLink() 函数的定义位于“os_tmr.c”。

os_tmr.c

```
906 void  OS_TmrLink (OS_TMR  *p_tmr,  //定时器控制块指针
907                   OS_OPT   opt)    //选项
908 {
909     OS_TMR_SPOKE     *p_spoke;
910     OS_TMR           *p_tmr0;
911     OS_TMR           *p_tmr1;
912     OS_TMR_SPOKE_IX   spoke;
913
914
915
916     p_tmr->State = OS_TMR_STATE_RUNNING;                          //重置定时器为运行状态
917     if (opt == OS_OPT_LINK_PERIODIC) {                            //如果定时器是再次插入
918         p_tmr->Match = p_tmr->Period + OSTmrTickCtr;              //匹配时间加个周期重载值
919     } else {                                                      //如果定时器是首次插入
920         if (p_tmr->Dly == (OS_TICK)0) {                           //如果定时器的 Dly = 0
921             p_tmr->Match = p_tmr->Period + OSTmrTickCtr;          //匹配时间加个周期重载值
922         } else {                                                  //如果定时器的 Dly != 0
923             p_tmr->Match = p_tmr->Dly    + OSTmrTickCtr;          //匹配时间加个 Dly
924         }
925     }
926     spoke  = (OS_TMR_SPOKE_IX)(p_tmr->Match % OSCfg_TmrWheelSize); //通过哈希算法觉得将该定时器
927     p_spoke = &OSCfg_TmrWheel[spoke];                             //插入到定时器轮的哪个列表。
```

```
928
929     if (p_spoke->FirstPtr ==  (OS_TMR *)0) {                //如果列表为空，
930         p_tmr->NextPtr      = (OS_TMR *)0;                  //直接将该定时器作为列表的第一个元素。
931         p_tmr->PrevPtr      = (OS_TMR *)0;
932         p_spoke->FirstPtr   = p_tmr;
933         p_spoke->NbrEntries = 1u;
934     } else {                                                //如果列表非空
935         p_tmr->Remain  = p_tmr->Match                       //算出定时器 p_tmr 的剩余时间
936                        - OSTmrTickCtr;
937         p_tmr1         = p_spoke->FirstPtr;                 //取列表的首个元素到 p_tmr1
938         while (p_tmr1 != (OS_TMR *)0) {                     //如果 p_tmr1 非空
939             p_tmr1->Remain = p_tmr1->Match                  //算出 p_tmr1 的剩余时间
940                            - OSTmrTickCtr;
941             if (p_tmr->Remain > p_tmr1->Remain) {           //如果 p_tmr 的剩余时间大于 p_tmr1 的
942                 if (p_tmr1->NextPtr  != (OS_TMR *)0) {      //如果 p_tmr1 后面非空
943                     p_tmr1           = p_tmr1->NextPtr;     //取p_tmr1后一个定时器为新的p_tmr1进行下一次循环
944                 } else {                                    //如果 p_tmr1 后面为空
945                     p_tmr->NextPtr   = (OS_TMR *)0;         //将 p_tmr 插到 p_tmr1 的后面，结束循环
946                     p_tmr->PrevPtr   =  p_tmr1;
947                     p_tmr1->NextPtr  =  p_tmr;
948                     p_tmr1           = (OS_TMR *)0;
949                 }
950             } else {                                        //如果 p_tmr 的剩余时间不大于 p_tmr1 的，
951                 if (p_tmr1->PrevPtr == (OS_TMR *)0) {       //将 p_tmr 插入到 p_tmr1 的前一个，结束循环。
952                     p_tmr->PrevPtr   = (OS_TMR *)0;
953                     p_tmr->NextPtr   = p_tmr1;
954                     p_tmr1->PrevPtr  = p_tmr;
955                     p_spoke->FirstPtr = p_tmr;
956                 } else {
957                     p_tmr0           = p_tmr1->PrevPtr;
958                     p_tmr->PrevPtr   = p_tmr0;
959                     p_tmr->NextPtr   = p_tmr1;
960                     p_tmr0->NextPtr  = p_tmr;
961                     p_tmr1->PrevPtr  = p_tmr;
962                 }
963                 p_tmr1 = (OS_TMR *)0;
964             }
965         }
966         p_spoke->NbrEntries++;                              //列表元素成员数加1
967     }
968     if (p_spoke->NbrEntriesMax < p_spoke->NbrEntries) {     //更新列表成员数最大值历史记录
969         p_spoke->NbrEntriesMax = p_spoke->NbrEntries;
970     }
971 }
```

图 5-5 OS_TmrLink() 函数

OS_TmrUnlink() 函数的定义也位于“os_tmr.c”。

```
os_tmr.c
1017 void  OS_TmrUnlink (OS_TMR  *p_tmr)    //定时器控制块指针
1018 {
1019     OS_TMR_SPOKE    *p_spoke;
1020     OS_TMR          *p_tmr1;
1021     OS_TMR          *p_tmr2;
1022     OS_TMR_SPOKE_IX  spoke;
1023
1024
1025
1026     spoke   = (OS_TMR_SPOKE_IX)(p_tmr->Match % OSCfg_TmrWheelSize); //与插入时一样，通过哈希算法找出
1027     p_spoke = &OSCfg_TmrWheel[spoke];                               //该定时器在定时器轮的哪个列表。
1028
1029     if (p_spoke->FirstPtr == p_tmr) {                       //如果 p_tmr 是列表的首个元素
1030         p_tmr1           = (OS_TMR *)p_tmr->NextPtr;        //取 p_tmr 后一个元素为 p_tmr1(可能为空)
1031         p_spoke->FirstPtr = (OS_TMR *)p_tmr1;               //表首改为 p_tmr1
1032         if (p_tmr1 != (OS_TMR *)0) {                        //如果 p_tmr1 确定非空
1033             p_tmr1->PrevPtr = (OS_TMR *)0;                  //p_tmr1 的前面清空
1034         }
1035     } else {                                                //如果 p_tmr 不是列表的首个元素
1036         p_tmr1          = (OS_TMR *)p_tmr->PrevPtr;         //将 p_tmr 从列表移除，并将 p_tmr
1037         p_tmr2          = (OS_TMR *)p_tmr->NextPtr;         //前后的两个元素连接在一起.
1038         p_tmr1->NextPtr = p_tmr2;
1039         if (p_tmr2 != (OS_TMR *)0) {
1040             p_tmr2->PrevPtr = (OS_TMR *)p_tmr1;
1041         }
1042     }
1043     p_tmr->State    = OS_TMR_STATE_STOPPED;    //复位 p_tmr 的指标
1044     p_tmr->NextPtr = (OS_TMR *)0;
1045     p_tmr->PrevPtr = (OS_TMR *)0;
1046     p_spoke->NbrEntries--;                     //列表元素成员减1
1047 }
```

图 5-6 OS_TmrUnlink() 函数

5.1.3 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 () 函数的信息如下表所示。

表 11 OSTmrStop ()

<table>
<tr><td>函数原型</td><td colspan="5">CPU_BOOLEAN OSTmrStop (OS_TMR *p_tmr,
OS_OPT opt,
void *p_callback_arg,
OS_ERR *p_err);</td></tr>
<tr><td>功能</td><td colspan="5">停止一个软件定时器。</td></tr>
<tr><td rowspan="11">参数</td><td>p_tmr</td><td colspan="4">定时器控制块指针。</td></tr>
<tr><td>opt</td><td rowspan="3">选项</td><td>OS_OPT_TMR_NONE</td><td colspan="2">只需停止定时器，不需执行指定事件。</td></tr>
<tr><td></td><td>OS_OPT_TMR_CALLBACK</td><td colspan="2">停止定时器，并执行回调函数。</td></tr>
<tr><td></td><td>OS_OPT_TMR_CALLBACK_ARG</td><td colspan="2">停止定时器，并执行回调函数，且将 p_callback_arg 作为新实参。</td></tr>
<tr><td>p_callback_arg</td><td colspan="4">opt 为 OS_OPT_TMR_CALLBACK_ARG 时，作为回调函数的新实参。</td></tr>
<tr><td rowspan="9">p_err</td><td colspan="2" rowspan="9">返回错误类型</td><td>OS_ERR_NONE</td><td>没错误</td></tr>
<tr><td>OS_ERR_OBJ_TYPE</td><td>p_tmr 不是一个定时器指针</td></tr>
<tr><td>OS_ERR_OPT_INVALID</td><td>opt 非法</td></tr>
<tr><td>OS_ERR_TMR_INACTIVE</td><td>定时器为被创建</td></tr>
<tr><td>OS_ERR_TMR_INVALID</td><td>p_tmr 为空</td></tr>
<tr><td>OS_ERR_TMR_INVALID_STATE</td><td>定时器状态非法</td></tr>
<tr><td>OS_ERR_TMR_ISR</td><td>在中断中被调用</td></tr>
<tr><td>OS_ERR_TMR_NO_CALLBACK</td><td>定时器不存在回调函数</td></tr>
<tr><td>OS_ERR_TMR_STOPPED</td><td>定时器已被停止</td></tr>
<tr><td>返回值</td><td colspan="5">✧ DEF_TRUE，停止成功（包括定时器已被停止，即错误类型为“OS_ERR_TMR_STOPPED”）。
✧ DEF_FALSE，停止失败。</td></tr>
<tr><td>注意事项</td><td colspan="5">✧ 不可以在中断中调用该函数。</td></tr>
</table>

OSTmrStop () 函数的定义也位于“os_tmr.c”。

os_tmr.c | app.c | os_cfg.h

```
607 CPU_BOOLEAN  OSTmrStop (OS_TMR  *p_tmr,           //定时器控制块指针
608                         OS_OPT   opt,            //选项
609                         void    *p_callback_arg, //传给回调函数的新参数
610                         OS_ERR  *p_err)          //返回错误类型
611 {
612     OS_TMR_CALLBACK_PTR  p_fnct;
613     OS_ERR               err;
614     CPU_BOOLEAN          success;  //暂存函数执行结果
615
616
617
618 #ifdef OS_SAFETY_CRITICAL                  //如果使能（默认禁用）了安全检测
619     if (p_err == (OS_ERR *)0) {            //如果错误类型实参为空
620         OS_SAFETY_CRITICAL_EXCEPTION();    //执行安全检测异常函数
621         return (DEF_FALSE);                //返回 DEF_FALSE，不继续执行
622     }
623 #endif
```

```
624
625 #if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u          //如果使能（默认使能）了中断中非法调用检测
626     if (OSIntNestingCtr > (OS_NESTING_CTR)0) { //如果该函数是在中断中被调用
627         *p_err = OS_ERR_TMR_ISR;               //错误类型为“在中断函数中定时”
628         return (DEF_FALSE);                    //返回 DEF_FALSE，不继续执行
629     }
630 #endif
631
632 #if OS_CFG_ARG_CHK_EN > 0u             //如果使能（默认使能）了参数检测
633     if (p_tmr == (OS_TMR *)0) {        //如果使能 p_tmr 的实参为空
634         *p_err = OS_ERR_TMR_INVALID;   //错误类型为“无效的定时器”
635         return (DEF_FALSE);            //返回 DEF_FALSE，不继续执行
636     }
637 #endif
638
639 #if OS_CFG_OBJ_TYPE_CHK_EN > 0u              //如果使能（默认使能）了对象类型检测
640     if (p_tmr->Type != OS_OBJ_TYPE_TMR) {    //如果该定时器的对象类型有误
641         *p_err = OS_ERR_OBJ_TYPE;            //错误类型为“对象类型错误”
642         return (DEF_FALSE);                  //返回 DEF_FALSE，不继续执行
643     }
644 #endif
```

```
645
646     OSSchedLock(&err);                                          //锁住调度器
647     switch (p_tmr->State) {                                     //根据定时器的状态分类处理
648         case OS_TMR_STATE_RUNNING:                              //如果定时器正在运行
649              OS_TmrUnlink(p_tmr);                               //从定时器轮列表里移除该定时器
650             *p_err = OS_ERR_NONE;                               //错误类型为“无错误”
651              switch (opt) {                                     //根据选项分类处理
652                  case OS_OPT_TMR_CALLBACK:                      //执行回调函数，使用创建定时器时的实参
653                       p_fnct = p_tmr->CallbackPtr;              //取定时器的回调函数
654                       if (p_fnct != (OS_TMR_CALLBACK_PTR)0) {   //如果回调函数存在
655                         (*p_fnct)((void *)p_tmr, p_tmr->CallbackPtrArg); //使用创建定时器时的实参执行回调函数
656                       } else {                                  //如果回调函数不存在
657                          *p_err = OS_ERR_TMR_NO_CALLBACK;       //错误类型为“定时器没有回调函数”
658                       }
659                       break;
660
661                  case OS_OPT_TMR_CALLBACK_ARG:                  //执行回调函数，使用 p_callback_arg 作为实参
662                       p_fnct = p_tmr->CallbackPtr;              //取定时器的回调函数
663                       if (p_fnct != (OS_TMR_CALLBACK_PTR)0) {   //如果回调函数存在
664                         (*p_fnct)((void *)p_tmr, p_callback_arg); //使用 p_callback_arg 作为实参执行回调函数
665                       } else {                                  //如果回调函数不存在
666                          *p_err = OS_ERR_TMR_NO_CALLBACK;       //错误类型为“定时器没有回调函数”
667                       }
668                       break;
```

```
669
670                     case OS_OPT_TMR_NONE:             //只需停掉定时器
671                          break;
672
673                     default:                          //情况超出预期
674                         OSSchedUnlock(&err);          //解锁调度器
675                        *p_err = OS_ERR_OPT_INVALID; //错误类型为“选项无效”
676                         return (DEF_FALSE);           //返回 DEF_FALSE，不继续执行
677                 }
678                 OSSchedUnlock(&err);
679                 success = DEF_TRUE;
680                 break;
681
682             case OS_TMR_STATE_COMPLETED:              //如果定时器已完成第一次定时
683             case OS_TMR_STATE_STOPPED:                //如果定时器已被停止
684                  OSSchedUnlock(&err);                 //解锁调度器
685                 *p_err    = OS_ERR_TMR_STOPPED;       //错误类型为“定时器已被停止”
686                  success = DEF_TRUE;                  //执行结果暂为 DEF_TRUE
687                  break;
688
689             case OS_TMR_STATE_UNUSED:                 //如果该定时器未被创建过
690                  OSSchedUnlock(&err);                 //解锁调度器
691                 *p_err    = OS_ERR_TMR_INACTIVE;      //错误类型为“定时器未激活”
692                  success = DEF_FALSE;                 //执行结果暂为 DEF_FALSE
693                  break;
694
695             default:                                  //如果定时器状态超出预期
696                  OSSchedUnlock(&err);                 //解锁调度器
697                 *p_err    = OS_ERR_TMR_INVALID_STATE;//错误类型为“定时器状态非法”
698                  success = DEF_FALSE;                 //执行结果暂为 DEF_FALSE
699                  break;
700     }
701     return (success);                                 //返回执行结果
702 }
```

图 5-7 OSTmrStop () 函数

5.1.4 OSTmrDel ()

OSTmrDel () 函数用于删除一个软件定时器。OSTmrDel () 函数的信息如下表所示。

表 12 OSTmrDel ()

函数原型	CPU_BOOLEAN OSTmrDel (OS_TMR *p_tmr, OS_ERR *p_err);			
功能	删除一个软件定时器。			
参数	p_tmr	定时器控制块指针。		
	p_err	返回错误类型	OS_ERR_NONE	没错误
			OS_ERR_OBJ_TYPE	p_tmr 不是一个定时器类型
			OS_ERR_TMR_INVALID	p_tmr 为空
			OS_ERR_TMR_ISR	在中断中被调用
			OS_ERR_TMR_INACTIVE	定时器未被创建过
			OS_ERR_TMR_INVALID_STATE	定时器处于非法状态
返回	✧ DEF_TRUE，删除成功；			

值	✧ DEF_FALSE，删除失败，或者有错误。
注意事项	✧ 不可以在中断中调用该函数。

OSTmrDel () 函数的定义位于“os_tmr.c”。

os_tmr.c | app.c | os_cfg.h

```
211 #if OS_CFG_TMR_DEL_EN > 0u                  //如果使能（默认是嫩）了 OSTmrDel() 函数
212 CPU_BOOLEAN  OSTmrDel (OS_TMR  *p_tmr, //定时器控制块指针
213                        OS_ERR  *p_err) //返回错误类型
214 {
215     OS_ERR       err;
216     CPU_BOOLEAN  success;  //暂存函数执行结果
217
218
219
220 #ifdef OS_SAFETY_CRITICAL                  //如果使能（默认禁用）了安全检测
221     if (p_err == (OS_ERR *)0) {            //如果错误类型实参为空
222         OS_SAFETY_CRITICAL_EXCEPTION(); //执行安全检测异常函数
223         return (DEF_FALSE);                //返回 DEF_FALSE，不继续执行
224     }
225 #endif
```

```
226
227 #if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u          //如果使能（默认使能）了中断中非法调用检测
228     if (OSIntNestingCtr > (OS_NESTING_CTR)0) {  //如果该函数是在中断中被调用
229         *p_err  = OS_ERR_TMR_ISR;               //错误类型为“在中断函数中定时”
230         return (DEF_FALSE);                     //返回 DEF_FALSE，不继续执行
231     }
232 #endif
233
234 #if OS_CFG_ARG_CHK_EN > 0u                  //如果使能（默认使能）了参数检测
235     if (p_tmr == (OS_TMR *)0) {             //如果使能 p_tmr 的实参为空
236         *p_err = OS_ERR_TMR_INVALID; //错误类型为“无效的定时器”
237         return (DEF_FALSE);                 //返回 DEF_FALSE，不继续执行
238     }
239 #endif
240
241 #if OS_CFG_OBJ_TYPE_CHK_EN > 0u             //如果使能（默认使能）了对象类型检测
242     if (p_tmr->Type != OS_OBJ_TYPE_TMR) {   //如果该定时器的对象类型有误
243         *p_err = OS_ERR_OBJ_TYPE;           //错误类型为“对象类型错误”
244         return (DEF_FALSE);                 //返回 DEF_FALSE，不继续执行
245     }
246 #endif
```

```
247
248     OSSchedLock(&err);              //锁住调度器
249 #if OS_CFG_DBG_EN > 0u             //如果使能（默认使能）了调试代码和变量
250     OS_TmrDbgListRemove(p_tmr); //将该定时从定时器双向调试链表移除
251 #endif
252     OSTmrQty--;                     //定时器个数减1
253
254     switch (p_tmr->State) {                         //根据定时器的状态分类处理
255         case OS_TMR_STATE_RUNNING:                  //如果定时器正在运行
256              OS_TmrUnlink(p_tmr);                   //从当前定时器轮列表移除定时器
257              OS_TmrClr(p_tmr);                      //复位定时器的指标
258              OSSchedUnlock(&err);                   //解锁调度器
259             *p_err    = OS_ERR_NONE;                //错误类型为“无错误”
260              success = DEF_TRUE;                   //执行结果暂为 DEF_TRUE
261              break;
262
263         case OS_TMR_STATE_STOPPED:                  //如果定时器已被停止
264         case OS_TMR_STATE_COMPLETED:                //如果定时器已完成第一次定时
265              OS_TmrClr(p_tmr);                      //复位定时器的指标
266              OSSchedUnlock(&err);                   //解锁调度器
267             *p_err    = OS_ERR_NONE;                //错误类型为“无错误”
268              success = DEF_TRUE;                   //执行结果暂为 DEF_TRUE
269              break;
270
271         case OS_TMR_STATE_UNUSED:                   //如果定时器已被删除
272              OSSchedUnlock(&err);                   //解锁调度器
273             *p_err    = OS_ERR_TMR_INACTIVE;        //错误类型为“定时器未激活”
274              success = DEF_FALSE;                  //执行结果暂为 DEF_FALSE
275              break;
276
277         default:                                    //如果定时器的状态超出预期
278              OSSchedUnlock(&err);                   //解锁调度器
279             *p_err    = OS_ERR_TMR_INVALID_STATE;   //错误类型为“定时器无效”
280              success = DEF_FALSE;                  //执行结果暂为 DEF_FALSE
281              break;
282     }
283     return (success);                               //返回执行结果
284 }
285 #endif
```

图 5-8 OSTmrDel () 函数

5.2 实例演示

5.2.1 实例 1

本节实例在“app.c”创建一个应用任务 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。该例程已经存放在配套资料的下图路径。

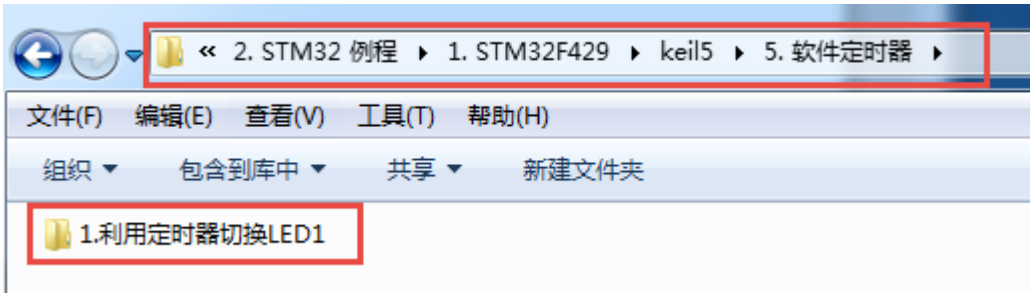

图 5-9 例程路径

本例程需用使用 LED 和 USART1，所以工程中需要添加其驱动文件和初始化。用户可参照前面相关实例，这里不再赘述。

任务函数 AppTaskTmr() 的定义如下。在任务中创建了一个软件定时器，并启动该软件定时器，之后改定时器相当于另一个任务独立运行。

```
220 static  void  AppTaskTmr ( void * p_arg )
221 {
222   OS_ERR      err;
223   OS_TMR      my_tmr;   //声明软件定时器对象
224
225
226   (void)p_arg;
227
228
229   /* 创建软件定时器 */
230   OSTmrCreate ((OS_TMR              *)&my_tmr,              //软件定时器对象
231                (CPU_CHAR            *)"MySoftTimer",        //命名软件定时器
232                (OS_TICK              )10,                   //定时器初始值，依10Hz时基计算，即为1s
233                (OS_TICK              )10,                   //定时器周期重载值，依10Hz时基计算，即为1s
234                (OS_OPT               )OS_OPT_TMR_PERIODIC, //周期性定时
235                (OS_TMR_CALLBACK_PTR  )TmrCallback,          //回调函数
236                (void                *)"Timer Over!",        //传递实参给回调函数
237                (OS_ERR              *)err);                 //返回错误类型
238
239   /* 启动软件定时器 */
240   OSTmrStart ((OS_TMR   *)&my_tmr, //软件定时器对象
241               (OS_ERR   *)err);    //返回错误类型
242
243   ts_start = OS_TS_GET();                       //获取定时前时间戳
244
245   while (DEF_TRUE) {                           //任务体，通常写成一个死循环
246
247     OSTimeDly ( 1000, OS_OPT_TIME_DLY, & err ); //不断阻塞该任务
248
249   }
250
251 }
```

图 5-10 AppTaskTmr() 任务函数

创建软件定时器 my_tmr 时为其指定了回调函数 TmrCallback()，每次定时完成都会调用该函数。TmrCallback() 函数的定义如下。

```
193 void TmrCallback (OS_TMR *p_tmr, void *p_arg) //软件定时器MyTmr的回调函数
194 {
195   CPU_INT32U       cpu_clk_freq;
196   CPU_SR_ALLOC();       //使用到临界段（在关/开中断时）时必需该宏，该宏声明和定义一个局部变
197                         //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）
198                         //，开中断时将该值还原。
199   printf ( "%s", ( char * ) p_arg );
200
201   cpu_clk_freq = BSP_CPU_ClkFreq();                    //获取CPU时钟，时间戳是以该时钟计数
202
203   macLED1_TOGGLE ();
204
205   ts_end = OS_TS_GET() - ts_start;     //获取定时后的时间戳（以CPU时钟进行计数的一个计数值）
206                                        //，并计算定时时间。
207   OS_CRITICAL_ENTER();                 //进入临界段，不希望下面串口打印遭到中断
208
209   printf ( "\r\n定时1s，通过时间戳测得定时 %07d us，即 %04d ms。\r\n",
210            ts_end / ( cpu_clk_freq / 1000000 ),      //将定时时间折算成 us
211            ts_end / ( cpu_clk_freq / 1000 ) );       //将定时时间折算成 ms
212
213   OS_CRITICAL_EXIT();
214
215   ts_start = OS_TS_GET();                            //获取定时前时间戳
216
217 }
```

图 5-11 TmrCallback() 函数

把 STM32 的 USART1 连接至电脑的串口调试助手，编译和下载程序到秉火 STM32 开发

板，运行程序。用户可以看到开发板上的 LED1 每隔 1s 切换一次亮灭状态，并且串口调试助手打印内容如下。其中的“Timer Over!”正是调用 OSTmrCreate()创建定时器时传递给回调函数 TmrCallback()的实参。另一个消息打印了时间戳测试出来的实际定时长度，可以观察到比预期的 1s 略少，但还是蛮精准的。

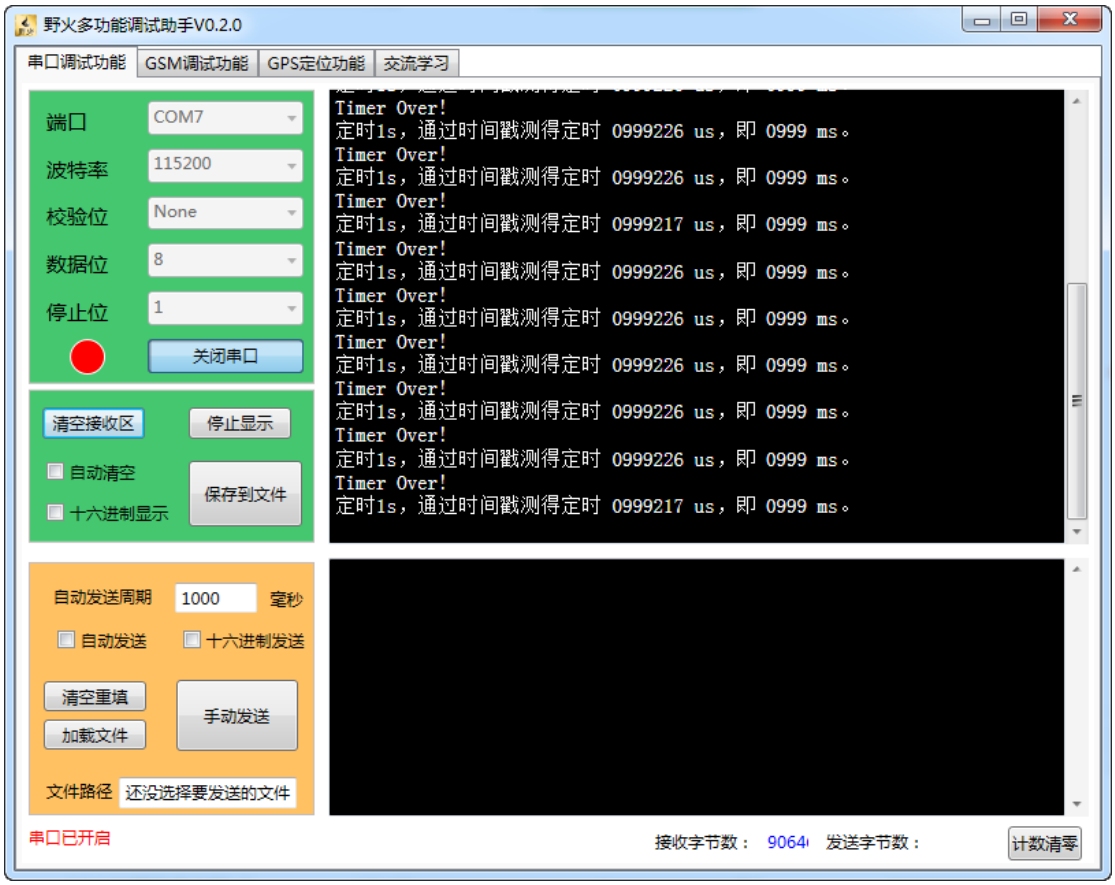

图 5-12 串口调试助手

5.3 章末总结

软件定时器是 uC/OS 操作系统的一种软件性定时器，也就是通过代码实现的具有定时功能的一种内核机制。它与硬件定时器不同，精准度也无法跟硬件定时器媲美。

要使用软件定时器，首先要声明一个 OS_TMR 对象，并通过 OSTmrCreate () 函数创建该软件定时器。软件定时器分 OS_OPT_TMR_ONE_SHOT 和 OS_OPT_TMR_PERIODIC 两种类型。前者相当于相当于硬件定时器的突发模式，执行的是一次性定时，定时完成之后软件定时器就停止工作，需要再次启动才能再次工作。后者相当于硬件定时器的周期模式，具有周期重载值，可以不断周期工作，不需再次启动。

软件定时器创建完成之后，还无法立即工作，需要调用 OSTmrStart () 函数来启动它投入工作。

OSTmrStop () 函数可以停止一个软件定时器，但它只是被停止了定时，而并没有被删除。如果还想继续使用该软件定时器定时，调用 OSTmrStart () 函数启动它即可，但启动后是重新定时，而不是紧接着停止时的时间继续计时。

OSTmrDel () 函数用于删除一个软件定时器，删除之后该软件定时器不能再被使用。

第6章 多值信号量

多值信号量是 uC/OS 操作系统的一个内核对象，主要用于标志事件的发生和共享资源管理。

6.1 原理简述

如果想要使用多值信号量，就必须事先使能多值信号量。多值信号量的使能位于“os_cfg.h”。

os_cfg.h | os_sem.c | os.h | os_core.c

```
74
75                                          /* ---------------------------- SEMAPHORES -----
76  #define OS_CFG_SEM_EN                1u   //使能或禁用多值信号量
77  #define OS_CFG_SEM_DEL_EN            1u   //使能或禁用 OSSemDel() 函数
78  #define OS_CFG_SEM_PEND_ABORT_EN     1u   //使能或禁用 OSSemPendAbort() 函数
79  #define OS_CFG_SEM_SET_EN            1u   //使能或禁用 OSSemSet() 函数
80
```

图 6-1

6.1.1 OSSemCreate ()

要使用 uC/OS 的多值信号量必须先声明和创建多值信号量，调用 OSSemCreate () 函数可以创建一个多值信号量。OSSemCreate () 函数的信息如下表所示。

表 13 OSSemCreate ()

函数原型	void OSSemCreate (OS_SEM *p_sem, CPU_CHAR *p_name, OS_SEM_CTR cnt, OS_ERR *p_err);			
功能	创建一个多值信号量。			
参数	p_sem	多值信号量指针。		
	p_name	多值信号量名称。		
	cnt	如果信号量用于分享资源，则该参数为资源数目。		
		如果信号量用于标志事件的是否发生，则该参数设为 0。		
	p_err	返回错误类型	OS_ERR_NONE	没错误
			OS_ERR_CREATE_ISR	在中断中调用该函数
			OS_ERR_ILLEGAL_CREATE_RUN_TIME	在调用 OSSafetyCriticalStart() 函数后创建内核对象

			OS_ERR_NAME	p_name 为空指针
			OS_ERR_OBJ_CREATED	信号量已经被创建过
			OS_ERR_OBJ_PTR_NULL	p_sem 是个空指针
			OS_ERR_OBJ_TYPE	p_sem 已被初始化到另一种对象类型
返回值	无。			
注意事项	✧ 创建前必须先为 p_sem 声明一个多值信号量对象（OS_SEM）。 ✧ 不可以在中断中调用该函数。			

OSSemCreate () 函数的定义位于“os_sem.c”。

os_sem.c | os_cfg.h | os_cfg_app.h | app.c | os_core.c | os.h

```
73  void  OSSemCreate (OS_SEM      *p_sem,   //多值信号量控制块指针
74                     CPU_CHAR    *p_name,  //多值信号量名称
75                     OS_SEM_CTR   cnt,     //资源数目或事件是否发生标志
76                     OS_ERR      *p_err)   //返回错误类型
77  {
78      CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必需该宏，该宏声明和定义一个局部变
79                      //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）
80                      //，开中断时将该值还原。
81
82  #ifdef OS_SAFETY_CRITICAL                 //如果使能（默认禁用）了安全检测
83      if (p_err == (OS_ERR *)0) {           //如果错误类型实参为空
84          OS_SAFETY_CRITICAL_EXCEPTION(); //执行安全检测异常函数
85          return;                           //返回，不继续执行
86      }
87  #endif

88
89  #ifdef OS_SAFETY_CRITICAL_IEC61508        //如果使能（默认禁用）了安全关键
90      if (OSSafetyCriticalStartFlag == DEF_TRUE) {  //如果是在调用 OSSafetyCriticalStart() 后创建该多值信号量
91          *p_err = OS_ERR_ILLEGAL_CREATE_RUN_TIME;  //错误类型为“非法创建内核对象”
92          return;                           //返回，不继续执行
93      }
94  #endif
95
96  #if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u    //如果使能（默认使能）了中断中非法调用检测
97      if (OSIntNestingCtr > (OS_NESTING_CTR)0) {    //如果该函数是在中断中被调用
98          *p_err = OS_ERR_CREATE_ISR;       //错误类型为“在中断函数中定时”
99          return;                           //返回，不继续执行
100     }
101 #endif
102
103 #if OS_CFG_ARG_CHK_EN > 0u                //如果使能（默认使能）了参数检测
104     if (p_sem == (OS_SEM *)0) {           //如果参数 p_sem 为空
105         *p_err = OS_ERR_OBJ_PTR_NULL;     //错误类型为“多值信号量对象为空”
106         return;                           //返回，不继续执行
107     }
108 #endif
```

```
109
110     OS_CRITICAL_ENTER();                    //进入临界段
111     p_sem->Type     = OS_OBJ_TYPE_SEM;  //初始化多值信号量指标
112     p_sem->Ctr      = cnt;
113     p_sem->TS       = (CPU_TS)0;
114     p_sem->NamePtr = p_name;
115     OS_PendListInit(&p_sem->PendList); //初始化该多值信号量的等待列表
116
117 #if OS_CFG_DBG_EN > 0u          //如果使能（默认使能）了调试代码和变量
118     OS_SemDbgListAdd(p_sem); //将该定时添加到多值信号量双向调试链表
119 #endif
120     OSSemQty++;                 //多值信号量个数加1
121
122     OS_CRITICAL_EXIT_NO_SCHED();      //退出临界段（无调度）
123    *p_err = OS_ERR_NONE;                //错误类型为“无错误”
124 }
```

图 6-2 OSSemCreate () 函数

其中，调用了 OS_PendListInit() 函数初始化了多值信号量的等待列表。每个多值信号量都有一个等待列表，凡是等待该多值信号量的的任务都会被插入到这个等待列表，方便高效管理。OS_PendListInit() 函数的定义位于“os_core.c”。

```
1319 void  OS_PendListInit (OS_PEND_LIST  *p_pend_list)
1320 {
1321     p_pend_list->HeadPtr    = (OS_PEND_DATA *)0;    //复位等待列表的所有成员
1322     p_pend_list->TailPtr    = (OS_PEND_DATA *)0;
1323     p_pend_list->NbrEntries = (OS_OBJ_QTY    )0;
1324 }
```

图 6-3 OS_PendListInit() 函数

如果使能了 OS_CFG_DBG_EN（位于“os_cfg.h”），创建多值信号量时还会调用 OS_SemDbgListAdd() 函数将该多值信号量插入到一个多值信号量调试列表，是为方便调试所设。OS_SemDbgListAdd() 函数的定义位于“os_sem.c”。

```
771 #if OS_CFG_DBG_EN > 0u                   //如果使能（默认使能）了调试代码和变量
772 void  OS_SemDbgListAdd (OS_SEM  *p_sem)  //将该多值信号量插入到多值信号量列表的最前端
773 {
774     p_sem->DbgNamePtr               = (CPU_CHAR *)((void *)" ");//先不指向任何任务的名称
775     p_sem->DbgPrevPtr               = (OS_SEM   *)0;            //将该信号量作为列表的最前端
776     if (OSSemDbgListPtr == (OS_SEM *)0) {                       //如果列表为空
777         p_sem->DbgNextPtr           = (OS_SEM   *)0;            //该信号量的下一个元素也为空
778     } else {                                                    //如果列表非空
779         p_sem->DbgNextPtr           =  OSSemDbgListPtr;         //列表原来的首元素作为该信号量的下一个元素
780         OSSemDbgListPtr->DbgPrevPtr =  p_sem;                   //原来的首元素的前面变为了该信号量
781     }
782     OSSemDbgListPtr                 =  p_sem;                   //该信号量成为列表的新首元素
783 }
```

图 6-4 OS_SemDbgListAdd() 函数

6.1.2 OSSemPost ()

OSSemPost () 函数用于发布多值信号量。OSSemPost () 函数的信息如下表所示。

表 14 OSSemPost ()

函 数	OS_SEM_CTR　OSSemPost (OS_SEM　*p_sem,

原型	OS_OPT opt, OS_ERR *p_err);			
功能	发布多值信号量。			
参数	p_sem	定时器控制块指针。		
	opt	OS_OPT_POST_1	发布给等待该信号量中最高优先级的任务。	
		OS_OPT_POST_ALL	发布给等待该信号量的所有任务。	
		OS_OPT_POST_1 | OS_OPT_POST_NO_SCHED	发布给等待该信号量中最高优先级的任务，但不进行任务调度。	
		OS_OPT_POST_ALL | OS_OPT_POST_NO_SCHED	发布给等待该信号量的所有任务，但不进行任务调度。	
	p_err	返回错误类型	OS_ERR_NONE	没错误
			OS_ERR_OBJ_PTR_NULL	p_sem 为空
			OS_ERR_OBJ_TYPE	p_sem 不是多值信号量类型对象
			OS_ERR_SEM_OVF	该发布将导致了信号量的计数值溢出
返回值	✧ 0，有错误。 ✧ 其他值，信号量的计数值。			

OSSemPost () 函数的定义也位于“os_sem.c”。

os_sem.c | os_cfg.h | os_cfg_app.h | app.c | os_core.c | os.h | os_type.h | lib_def.h | os_int.c

```
584 OS_SEM_CTR  OSSemPost (OS_SEM  *p_sem,      //多值信号量控制块指针
585                        OS_OPT   opt,        //选项
586                        OS_ERR  *p_err)      //返回错误类型
587 {
588     OS_SEM_CTR  ctr;
589     CPU_TS      ts;
590
591
592
593 #ifdef OS_SAFETY_CRITICAL                   //如果使能（默认禁用）了安全检测
594     if (p_err == (OS_ERR *)0) {             //如果错误类型实参为空
595         OS_SAFETY_CRITICAL_EXCEPTION();     //执行安全检测异常函数
596         return ((OS_SEM_CTR)0);             //返回0（有错误），不继续执行
597     }
598 #endif
```

```
599
600 #if OS_CFG_ARG_CHK_EN > 0u                    //如果使能（默认使能）了参数检测功能
601     if (p_sem == (OS_SEM *)0) {                //如果 p_sem 为空
602         *p_err  = OS_ERR_OBJ_PTR_NULL;         //返回错误类型为“内核对象指针为空”
603          return ((OS_SEM_CTR)0);               //返回0（有错误），不继续执行
604     }
605     switch (opt) {                                          //根据选项情况分类处理
606         case OS_OPT_POST_1:                                 //如果选项在预期内，不处理
607         case OS_OPT_POST_ALL:
608         case OS_OPT_POST_1    | OS_OPT_POST_NO_SCHED:
609         case OS_OPT_POST_ALL  | OS_OPT_POST_NO_SCHED:
610              break;
611
612         default:                                            //如果选项超出预期
613             *p_err =  OS_ERR_OPT_INVALID;                   //返回错误类型为“选项非法”
614              return ((OS_SEM_CTR)0u);                       //返回0（有错误），不继续执行
615     }
616 #endif
617
618 #if OS_CFG_OBJ_TYPE_CHK_EN > 0u                 //如果使能了对象类型检测
619     if (p_sem->Type != OS_OBJ_TYPE_SEM) {       //如果 p_sem 的类型不是多值信号量类型
620         *p_err = OS_ERR_OBJ_TYPE;               //返回错误类型为“对象类型错误”
621          return ((OS_SEM_CTR)0);                //返回0（有错误），不继续执行
622     }
623 #endif
624
625     ts = OS_TS_GET();                                       //获取时间戳
626
627 #if OS_CFG_ISR_POST_DEFERRED_EN > 0u                        //如果使能了中断延迟发布
628     if (OSIntNestingCtr > (OS_NESTING_CTR)0) {              //如果该函数是在中断中被调用
629         OS_IntQPost((OS_OBJ_TYPE)OS_OBJ_TYPE_SEM,           //将该信号量发布到中断消息队列
630                     (void       *)p_sem,
631                     (void       *)0,
632                     (OS_MSG_SIZE)0,
633                     (OS_FLAGS    )0,
634                     (OS_OPT      )opt,
635                     (CPU_TS      )ts,
636                     (OS_ERR     *)p_err);
637         return ((OS_SEM_CTR)0);                             //返回0（尚未发布），不继续执行
638     }
639 #endif
640
641     ctr = OS_SemPost(p_sem,                                 //将信号量按照普通方式处理
642                      opt,
643                      ts,
644                      p_err);
645
646     return (ctr);                                           //返回信号的当前计数值
647 }
```

图 6-5 OSSemPost () 函数

其实，不管是否使能了中断延迟发布，最终都是调用 OS_SemPost() 函数进行发布信号量。只是使能了中断延迟发布的发布过程会比较曲折，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。OS_SemPost() 函数的定义位于“os_sem.c”。

os_sem.c | os.h

```
884 OS_SEM_CTR  OS_SemPost (OS_SEM  *p_sem, //多值信号量指针
885                         OS_OPT   opt,   //选项
886                         CPU_TS   ts,    //时间戳
887                         OS_ERR  *p_err) //返回错误类型
888 {
889     OS_OBJ_QTY      cnt;
890     OS_SEM_CTR      ctr;
891     OS_PEND_LIST   *p_pend_list;
892     OS_PEND_DATA   *p_pend_data;
893     OS_PEND_DATA   *p_pend_data_next;
894     OS_TCB         *p_tcb;
895     CPU_SR_ALLOC();
896
897
898
```

```
899     CPU_CRITICAL_ENTER();                                   //关中断
900     p_pend_list = &p_sem->PendList;                         //取出该信号量的等待列表
901     if (p_pend_list->NbrEntries == (OS_OBJ_QTY)0) {         //如果没有任务在等待该信号量
902         switch (sizeof(OS_SEM_CTR)) {                       //判断是否将导致该信号量计数值溢出，
903             case 1u:                                        //如果溢出，则开中断，返回错误类型为
904                  if (p_sem->Ctr == DEF_INT_08U_MAX_VAL) {   //“计数值溢出”，返回0（有错误），
905                      CPU_CRITICAL_EXIT();                   //不继续执行。
906                     *p_err = OS_ERR_SEM_OVF;
907                      return ((OS_SEM_CTR)0);
908                  }
909                  break;
910
911             case 2u:
912                  if (p_sem->Ctr == DEF_INT_16U_MAX_VAL) {
913                      CPU_CRITICAL_EXIT();
914                     *p_err = OS_ERR_SEM_OVF;
915                      return ((OS_SEM_CTR)0);
916                  }
917                  break;
918
919             case 4u:
920                  if (p_sem->Ctr == DEF_INT_32U_MAX_VAL) {
921                      CPU_CRITICAL_EXIT();
922                     *p_err = OS_ERR_SEM_OVF;
923                      return ((OS_SEM_CTR)0);
924                  }
925                  break;
926
927             default:
928                  break;
929         }
930         p_sem->Ctr++;                                       //信号量计数值不溢出则加1
931         ctr       = p_sem->Ctr;                             //获取信号量计数值到 ctr
932         p_sem->TS = ts;                                     //保存时间戳
933         CPU_CRITICAL_EXIT();                                //则开中断
934        *p_err     = OS_ERR_NONE;                            //返回错误类型为“无错误”
935         return (ctr);                                       //返回信号量的计数值，不继续执行
936     }
```

```
937
938     OS_CRITICAL_ENTER_CPU_EXIT();                                   //加锁调度器，但开中断
939     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
940         cnt = p_pend_list->NbrEntries;                              //获取等待任务数目到 cnt
941     } else {                                                        //如果要将信号量发布给优先级最高的等待任务
942         cnt = (OS_OBJ_QTY)1;                                        //将要操作的任务数为1，cnt 置1
943     }
944     p_pend_data = p_pend_list->HeadPtr;                             //获取等待列表的首个任务到 p_pend_data
945     while (cnt > 0u) {                                              //逐个处理要发布的任务
946         p_tcb            = p_pend_data->TCBPtr;                     //取出当前任务
947         p_pend_data_next = p_pend_data->NextPtr;                    //取出下一个任务
948         OS_Post((OS_PEND_OBJ *)((void *)p_sem),                     //发布信号量给当前任务
949                 p_tcb,
950                 (void      *)0,
951                 (OS_MSG_SIZE)0,
952                 ts);
953         p_pend_data = p_pend_data_next;                             //处理下一个任务
954         cnt--;
955     }
956     ctr = p_sem->Ctr;                                               //获取信号量计数值到 ctr
957     OS_CRITICAL_EXIT_NO_SCHED();                                    //减锁调度器，但不执行任务调度
958     if ((opt & OS_OPT_POST_NO_SCHED) == (OS_OPT)0) {                //如果 opt 没选择“发布时不调度任务”
959         OSSched();                                                  //任务调度
960     }
961    *p_err = OS_ERR_NONE;                                            //返回错误类型为“无错误”
962     return (ctr);                                                   //返回信号量的当前计数值
963 }
```

图 6-6 OS_SemPost() 函数

当没有任务正在等待当发布的多值信号量，多值信号量的资源计数变量 Ctr 就会加 1，如果有任务正在等待要发布的多值信号量，那么多值信号量的资源计数变量 Ctr 就不加 1，而是直接将信号量发布给正在等待的任务，解除它们的等待状态。如果选项 opt 选择了 OS_OPT_POST_1，就只解除所有等待任务中优先级最高的那个任务；如果选择了 OS_OPT_POST_ALL，就解除所有等待任务。在插入等待任务时，如果等待列表中已有同一优先级的任务，会将该任务插在这些同优先级任务的后面。也就是说，如果选项 opt 选择了 OS_OPT_POST_1，就只解除优先级最高任务中最早插入的那个任务。

在 OS_SemPost() 函数中，又会调用 OS_Post() 函数发布内核对象。OS_Post() 函数是一个底层的发布函数，它不仅仅用来发布多值信号量，还可以发布互斥信号量、消息队列、事件标志组、任务消息队列和任务信号量。OS_Post() 函数的定义位于“os_core.c”。

os_sem.c | os.h | os_core.c | os_cfg.h

```
1844 void  OS_Post (OS_PEND_OBJ  *p_obj,      //内核对象类型指针
1845             OS_TCB       *p_tcb,      //任务控制块
1846             void         *p_void,     //消息
1847             OS_MSG_SIZE   msg_size,   //消息大小
1848             CPU_TS        ts)         //时间戳
1849 {
1850     switch (p_tcb->TaskState) {                                     //根据任务状态分类处理
1851         case OS_TASK_STATE_RDY:                                     //如果任务处于就绪状态
1852         case OS_TASK_STATE_DLY:                                     //如果任务处于延时状态
1853         case OS_TASK_STATE_SUSPENDED:                               //如果任务处于挂起状态
1854         case OS_TASK_STATE_DLY_SUSPENDED:                           //如果任务处于延时中被挂起状态
1855              break;                                                 //不用处理，直接跳出
```

```
1856
1857            case OS_TASK_STATE_PEND:                                    //如果任务处于无期限等待状态
1858            case OS_TASK_STATE_PEND_TIMEOUT:                            //如果任务处于有期限等待状态
1859                 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
1860                     OS_Post1(p_obj,                                    //标记哪个内核对象被发布
1861                              p_tcb,
1862                              p_void,
1863                              msg_size,
1864                              ts);
1865                 } else {                                               //如果任务不是在等待多个信号量或消息队列
1866#if (OS_MSG_EN > 0u)
1867                     p_tcb->MsgPtr  = p_void;                           //保存消息到等待任务
1868                     p_tcb->MsgSize = msg_size;
1869#endif
1870                     p_tcb->TS      = ts;                               //保存时间戳到等待任务
1871                 }
1872                 if (p_obj != (OS_PEND_OBJ *)0) {                       //如果内核对象为空
1873                     OS_PendListRemove(p_tcb);                          //从等待列表移除该等待任务
1874#if OS_CFG_DBG_EN > 0u
1875                     OS_PendDbgNameRemove(p_obj,                        //移除内核对象的调试名
1876                                          p_tcb);
1877#endif
1878                 }
1879                 OS_TaskRdy(p_tcb);                                     //让该等待任务准备运行
1880                 p_tcb->TaskState  = OS_TASK_STATE_RDY;                 //任务状态改为就绪状态
1881                 p_tcb->PendStatus = OS_STATUS_PEND_OK;                 //清除等待状态
1882                 p_tcb->PendOn     = OS_TASK_PEND_ON_NOTHING;           //标记不再等待
1883                 break;
```

```
1884
1885            case OS_TASK_STATE_PEND_SUSPENDED:                          //如果任务在无期限等待中被挂起
1886            case OS_TASK_STATE_PEND_TIMEOUT_SUSPENDED:                  //如果任务在有期限等待中被挂起
1887                 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
1888                     OS_Post1(p_obj,                                    //标记哪个内核对象被发布
1889                              p_tcb,
1890                              p_void,
1891                              msg_size,
1892                              ts);
1893                 } else {                                               //如果任务不在等待多个信号量或消息队列
1894#if (OS_MSG_EN > 0u)
1895                     p_tcb->MsgPtr  = p_void;                           //如果使能了调试代码和变量
1896                     p_tcb->MsgSize = msg_size;                         //保存消息到等待任务
1897#endif
1898                     p_tcb->TS      = ts;                               //保存时间戳到等待任务
1899                 }
1900                 OS_TickListRemove(p_tcb);                              //从节拍列表移除该等待任务
1901                 if (p_obj != (OS_PEND_OBJ *)0) {                       //如果内核对象为空
1902                     OS_PendListRemove(p_tcb);                          //从等待列表移除该等待任务
1903#if OS_CFG_DBG_EN > 0u
1904                     OS_PendDbgNameRemove(p_obj,                        //如果使能了调试代码和变量
1905                                          p_tcb);                       //移除内核对象的调试名
1906#endif
1907                 }
1908                 p_tcb->TaskState  = OS_TASK_STATE_SUSPENDED;           //任务状态改为被挂起状态
1909                 p_tcb->PendStatus = OS_STATUS_PEND_OK;                 //清除等待状态
1910                 p_tcb->PendOn     = OS_TASK_PEND_ON_NOTHING;           //标记不再等待
1911                 break;
1912
1913            default:                                                    //如果任务状态超出预期
1914                 break;                                                 //直接跳出
1915        }
1916    }
```

图 6-7 OS_Post() 函数

6.1.3 OSSemPend ()

与 OSSemPost () 多值信号量发布函数相对应，OSSemPend() 函数用于等待一个多值信号量。

表 15 OSSemPend()

函数原	OS_SEM_CTR OSSemPend (OS_SEM *p_sem, OS_TICK timeout, OS_OPT opt,

<table>
<tr><td>型</td><td colspan="5">CPU_TS *p_ts,
OS_ERR *p_err);</td></tr>
<tr><td>功能</td><td colspan="5">等待一个多值信号量。</td></tr>
<tr><td rowspan="8">参数</td><td>p_sem</td><td colspan="4">多值信号量指针。</td></tr>
<tr><td>timeout</td><td colspan="4">等待超时时间（单位：时钟节拍），0 代表无期限等待。opt 为 OS_OPT_PEND_BLOCKING 时该参数才起作用。</td></tr>
<tr><td rowspan="2">opt</td><td rowspan="2">选项</td><td>OS_OPT_PEND_BLOCKING</td><td colspan="2">如果不能立即获得信号量，就堵塞当前任务，继续等待信号量。</td></tr>
<tr><td>OS_OPT_PEND_NON_BLOCKING</td><td colspan="2">如果不能立即获得信号量，不堵塞当前任务，不继续等待信号量。</td></tr>
<tr><td>p_ts</td><td>时间戳</td><td colspan="3">用于存储信号量最后一次被发布的时间戳，或者等待被中止的时间戳，或者信号量被删除时的时间戳，具体返回哪个时间戳，要根据返回的 p_err 判断。该参数可以为 NULL，表示用户不需要获得时间戳。</td></tr>
<tr><td rowspan="11">p_err</td><td rowspan="11">返回错误类型</td><td>OS_ERR_NONE</td><td colspan="2">没错误，获得信号量。</td></tr>
<tr><td>OS_ERR_OBJ_DEL</td><td colspan="2">p_sem 被删除。</td></tr>
<tr><td>OS_ERR_OBJ_PTR_NULL</td><td colspan="2">p_sem 为空。</td></tr>
<tr><td>OS_ERR_OBJ_TYPE</td><td colspan="2">p_sem 不是信号量类型对象。</td></tr>
<tr><td>OS_ERR_OPT_INVALID</td><td colspan="2">opt 非法。</td></tr>
<tr><td>OS_ERR_PEND_ABORT</td><td colspan="2">等待被另一个任务中止。</td></tr>
<tr><td>OS_ERR_PEND_ISR</td><td colspan="2">在中断中被调用。</td></tr>
<tr><td>OS_ERR_PEND_WOULD_BLOCK</td><td colspan="2">缺乏堵塞。</td></tr>
<tr><td>OS_ERR_SCHED_LOCKED</td><td colspan="2">调度器被锁。</td></tr>
<tr><td>OS_ERR_STATUS_INVALID</td><td colspan="2">等待状态非法。</td></tr>
<tr><td>OS_ERR_TIMEOUT</td><td colspan="2">等待超时。</td></tr>
<tr><td>返回值</td><td colspan="5">✧ 0，信号量的当前计数值为 0，或有错误。
✧ 其他值，信号量的当前计数值。</td></tr>
<tr><td>注意事项</td><td colspan="5">✧ 不可以在中断中调用该函数。</td></tr>
</table>

OSSemPend() 函数的定义也位于“os_sem.c”。

os_sem.c | os.h | os_core.c | os_cfg.h

```
311 OS_SEM_CTR  OSSemPend (OS_SEM   *p_sem,    //多值信号量指针
312                        OS_TICK   timeout,  //等待超时时间
313                        OS_OPT    opt,      //选项
314                        CPU_TS   *p_ts,     //等到信号量时的时间戳
315                        OS_ERR   *p_err)    //返回错误类型
316 {
317     OS_SEM_CTR    ctr;
318     OS_PEND_DATA  pend_data;
319     CPU_SR_ALLOC();
320
321
322
323 #ifdef OS_SAFETY_CRITICAL                          //如果使能（默认禁用）了安全检测
324     if (p_err == (OS_ERR *)0) {                    //如果错误类型实参为空
325         OS_SAFETY_CRITICAL_EXCEPTION();            //执行安全检测异常函数
326         return ((OS_SEM_CTR)0);                    //返回0（有错误），不继续执行
327     }
328 #endif
329
330 #if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u             //如果使能了中断中非法调用检测
331     if (OSIntNestingCtr > (OS_NESTING_CTR)0) {     //如果该函数在中断中被调用
332         *p_err = OS_ERR_PEND_ISR;                  //返回错误类型为“在中断中等待”
333         return ((OS_SEM_CTR)0);                    //返回0（有错误），不继续执行
334     }
335 #endif

336
337 #if OS_CFG_ARG_CHK_EN > 0u                         //如果使能了参数检测
338     if (p_sem == (OS_SEM *)0) {                    //如果 p_sem 为空
339         *p_err = OS_ERR_OBJ_PTR_NULL;              //返回错误类型为“内核对象为空”
340         return ((OS_SEM_CTR)0);                    //返回0（有错误），不继续执行
341     }
342     switch (opt) {                                 //根据选项分类处理
343         case OS_OPT_PEND_BLOCKING:                 //如果选择“等待不到对象进行堵塞”
344         case OS_OPT_PEND_NON_BLOCKING:             //如果选择“等待不到对象不进行堵塞”
345             break;                                 //直接跳出，不处理
346
347         default:                                   //如果选项超出预期
348             *p_err = OS_ERR_OPT_INVALID;           //返回错误类型为“选项非法”
349             return ((OS_SEM_CTR)0);                //返回0（有错误），不继续执行
350     }
351 #endif
352
353 #if OS_CFG_OBJ_TYPE_CHK_EN > 0u                    //如果使能了对象类型检测
354     if (p_sem->Type != OS_OBJ_TYPE_SEM) {          //如果 p_sem 不是多值信号量类型
355         *p_err = OS_ERR_OBJ_TYPE;                  //返回错误类型为“内核对象类型错误”
356         return ((OS_SEM_CTR)0);                    //返回0（有错误），不继续执行
357     }
358 #endif

359
360     if (p_ts != (CPU_TS *)0) {                     //如果 p_ts 非空
361         *p_ts  = (CPU_TS)0;                        //初始化（清零）p_ts，待用于返回时间戳
362     }
363     CPU_CRITICAL_ENTER();                          //关中断
364     if (p_sem->Ctr > (OS_SEM_CTR)0) {              //如果资源可用
365         p_sem->Ctr--;                              //资源数目减1
366         if (p_ts != (CPU_TS *)0) {                 //如果 p_ts 非空
367             *p_ts  = p_sem->TS;                    //获取该信号量最后一次发布的时间戳
368         }
369         ctr    = p_sem->Ctr;                       //获取信号量的当前资源数目
370         CPU_CRITICAL_EXIT();                       //开中断
371         *p_err = OS_ERR_NONE;                      //返回错误类型为“无错误”
372         return (ctr);                              //返回信号量的当前资源数目，不继续执行
373     }
374
375     if ((opt & OS_OPT_PEND_NON_BLOCKING) != (OS_OPT)0) {    //如果没有资源可用，而且选择了不堵塞任务
376         ctr    = p_sem->Ctr;                               //获取信号量的资源数目到 ctr
377         CPU_CRITICAL_EXIT();                               //开中断
378         *p_err = OS_ERR_PEND_WOULD_BLOCK;                  //返回错误类型为“等待渴求堵塞”
379         return (ctr);                                      //返回信号量的当前资源数目，不继续执行
380     } else {                                               //如果没有资源可用，但选择了堵塞任务
381         if (OSSchedLockNestingCtr > (OS_NESTING_CTR)0) {   //如果调度器被锁
382             CPU_CRITICAL_EXIT();                           //开中断
383             *p_err = OS_ERR_SCHED_LOCKED;                  //返回错误类型为“调度器被锁”
384             return ((OS_SEM_CTR)0);                        //返回0（有错误），不继续执行
385         }
386     }
```

```
387
388     OS_CRITICAL_ENTER_CPU_EXIT();                       //锁调度器，并重开中断
389     OS_Pend(&pend_data,                                 //堵塞等待任务，将当前任务脱离就绪列表，
390             (OS_PEND_OBJ *)((void *)p_sem),             //并插入到节拍列表和等待列表。
391             OS_TASK_PEND_ON_SEM,
392             timeout);
393
394     OS_CRITICAL_EXIT_NO_SCHED();                        //开调度器，但不进行调度
395
396     OSSched();                                          //找到并调度最高优先级就绪任务

397     /* 当前任务（获得信号量）得以继续运行 */
398     CPU_CRITICAL_ENTER();                               //关中断
399     switch (OSTCBCurPtr->PendStatus) {                  //根据当前运行任务的等待状态分类处理
400         case OS_STATUS_PEND_OK:                         //如果等待状态正常
401              if (p_ts != (CPU_TS *)0) {                 //如果 p_ts 非空
402                 *p_ts  =  OSTCBCurPtr->TS;              //获取信号被发布的时间戳
403              }
404             *p_err = OS_ERR_NONE;                       //返回错误类型为“无错误”
405              break;
406
407         case OS_STATUS_PEND_ABORT:                      //如果等待被终止中止
408              if (p_ts != (CPU_TS *)0) {                 //如果 p_ts 非空
409                 *p_ts  =  OSTCBCurPtr->TS;              //获取等待被中止的时间戳
410              }
411             *p_err = OS_ERR_PEND_ABORT;                 //返回错误类型为“等待被中止”
412              break;
413
414         case OS_STATUS_PEND_TIMEOUT:                    //如果等待超时
415              if (p_ts != (CPU_TS *)0) {                 //如果 p_ts 非空
416                 *p_ts  = (CPU_TS  )0;                   //清零 p_ts
417              }
418             *p_err = OS_ERR_TIMEOUT;                    //返回错误类型为“等待超时”
419              break;
420
421         case OS_STATUS_PEND_DEL:                        //如果等待的内核对象被删除
422              if (p_ts != (CPU_TS *)0) {                 //如果 p_ts 非空
423                 *p_ts  =  OSTCBCurPtr->TS;              //获取内核对象被删除的时间戳
424              }
425             *p_err = OS_ERR_OBJ_DEL;                    //返回错误类型为“等待对象被删除”
426              break;
427
428         default:                                        //如果等待状态超出预期
429             *p_err = OS_ERR_STATUS_INVALID;             //返回错误类型为“等待状态非法”
430              CPU_CRITICAL_EXIT();                       //开中断
431              return ((OS_SEM_CTR)0);                    //返回0（有错误），不继续执行
432     }
433     ctr = p_sem->Ctr;                                   //获取信号量的当前资源数目
434     CPU_CRITICAL_EXIT();                                //开中断
435     return (ctr);                                       //返回信号量的当前资源数目
436 }
```

图 6-8 OSSemPend() 函数

OSSemPend() 函数会调用一个更加底层的等待函数来执行当前任务对多值信号量的等待，该函数就是 OS_Pend()。与 OS_Post() 函数一样，OS_Pend() 函数不仅仅用来等待多值信号量，还可以等待互斥信号量、消息队列、事件标志组、任务消息队列和任务信号量。OS_Pend() 函数的定义位于“os_core.c”。

```
os_sem.c | os.h | os_core.c | os_cfg.h | os_type.h
873 void  OS_Pend (OS_PEND_DATA  *p_pend_data,   //待插入等待列表的元素
874              OS_PEND_OBJ   *p_obj,         //等待的内核对象
875              OS_STATE       pending_on,    //等待哪种对象内核
876              OS_TICK        timeout)       //等待期限
877 {
878     OS_PEND_LIST  *p_pend_list;
879
880
881
882     OSTCBCurPtr->PendOn     = pending_on;                    //资源不可用，开始等待
883     OSTCBCurPtr->PendStatus = OS_STATUS_PEND_OK;             //正常等待中
884
885     OS_TaskBlock(OSTCBCurPtr,                                //阻塞当前运行任务，
886                  timeout);                                   //如果 timeout 非0，把任务插入的节拍列表
887
888     if (p_obj != (OS_PEND_OBJ *)0) {                         //如果等待对象非空
889         p_pend_list           = &p_obj->PendList;            //获取对象的等待列表到 p_pend_list
890         p_pend_data->PendObjPtr = p_obj;                     //保存要等待的对象
891         OS_PendDataInit((OS_TCB      *)OSTCBCurPtr,          //初始化 p_pend_data（待插入等待列表）
892                         (OS_PEND_DATA *)p_pend_data,
893                         (OS_OBJ_QTY    )1);
894         OS_PendListInsertPrio(p_pend_list,                   //按优先级将 p_pend_data 插入到等待列表
895                               p_pend_data);
896     } else {                                                 //如果等待对象为空
897         OSTCBCurPtr->PendDataTblEntries = (OS_OBJ_QTY    )0; //清零当前任务的等待域数据
898         OSTCBCurPtr->PendDataTblPtr     = (OS_PEND_DATA *)0;
899     }
900 #if OS_CFG_DBG_EN > 0u                                       //如果使能了调试代码和变量
901     OS_PendDbgNameAdd(p_obj,                                 //更新信号量的 DbgNamePtr 元素为其等待
902                       OSTCBCurPtr);                          //列表中优先级最高的任务的名称。
903 #endif
904 }
```

图 6-9 OS_Pend() 函数

6.1.4 OSSemPendAbort ()

OSSemPendAbort() 函数用于中止任务对一个多值信号量的等待。要使用 OSSemPendAbort () 函数，还得事先使能 OS_CFG_SEM_PEND_ABORT_EN（位于“os_cfg.h”），如下图所示。

```
os_cfg.h | os_sem.c | os.h | os_core.c
74
75                                          /* ---------------------------- SEMAPHORES -----
76 #define OS_CFG_SEM_EN                 1u   //使能或禁用多值信号量
77 #define OS_CFG_SEM_DEL_EN             1u   //使能或禁用 OSSemDel() 函数
78 #define OS_CFG_SEM_PEND_ABORT_EN      1u   //使能或禁用 OSSemPendAbort() 函数
79 #define OS_CFG_SEM_SET_EN             1u   //使能或禁用 OSSemSet() 函数
80
```

图 6-10

OSSemPendAbort () 函数的信息如下表所示。

表 16 OSSemPendAbort ()

函数原型	OS_OBJ_QTY OSSemPendAbort (OS_SEM *p_sem, OS_OPT opt, OS_ERR *p_err);			
功能	中止对一个多值信号量的等待。			
参数	p_sem	多值信号量指针。		
	opt	选项。	OS_OPT_PEND_ABORT_1	只中止该信号量等待列表中的最高优先级任务。
			OS_OPT_PEND_ABORT_ALL	中止该信号量等待列表中的所有

				优先级任务。
			OS_OPT_PEND_ABORT_1 | OS_OPT_POST_NO_SCHED	只中止该信号量等待列表中的最高优先级任务，但不进行任务调度。
			OS_OPT_PEND_ABORT_ALL | OS_OPT_POST_NO_SCHED	中止该信号量等待列表中的所有优先级任务，但不进行任务调度。
	p_err	返回错误类型	OS_ERR_NONE	没错误。
			OS_ERR_OBJ_PTR_NULL	p_sem 为空。
			OS_ERR_OBJ_TYPE	p_sem 不是多值信号量类型。
			OS_ERR_OPT_INVALID	选项非法。
			OS_ERR_PEND_ABORT_ISR	该函数在中断中被调用。
			OS_ERR_PEND_ABORT_NONE	没有任务在等待该信号量。
返回值	✧ 0，没有任务在等待该信号量，或者有错误产生。 ✧ >0，被中止的任务数。			
注意事项	✧ 不可以在中断中调用该函数。			

OSSemPendAbort () 函数的定义位于“os_sem.c”。

os_cfg.h | os_sem.c | os.h | os_core.c

```
470 #if OS_CFG_SEM_PEND_ABORT_EN > 0u                //如果使能了 OSSemPendAbort() 函数
471 OS_OBJ_QTY  OSSemPendAbort (OS_SEM  *p_sem,  //多值信号量指针
472                             OS_OPT   opt,    //选项
473                             OS_ERR  *p_err) //返回错误类型
474 {
475     OS_PEND_LIST  *p_pend_list;
476     OS_TCB        *p_tcb;
477     CPU_TS         ts;
478     OS_OBJ_QTY     nbr_tasks;
479     CPU_SR_ALLOC();
480
481
482
483 #ifdef OS_SAFETY_CRITICAL                  //如果使能（默认禁用）了安全检测
484     if (p_err == (OS_ERR *)0) {            //如果错误类型实参为空
485         OS_SAFETY_CRITICAL_EXCEPTION();  //执行安全检测异常函数
486         return ((OS_OBJ_QTY)0u);           //返回0（有错误），不继续执行
487     }
488 #endif
```

```
489
490 #if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u                    //如果使能了中断中非法调用检测
491     if (OSIntNestingCtr > (OS_NESTING_CTR)0u) {          //如果该函数在中断中被调用
492         *p_err =  OS_ERR_PEND_ABORT_ISR;                 //返回错误类型为“在中断中中止等待”
493         return ((OS_OBJ_QTY)0u);                         //返回0（有错误），不继续执行
494     }
495 #endif
496
497 #if OS_CFG_ARG_CHK_EN > 0u                               //如果使能了参数检测
498     if (p_sem == (OS_SEM *)0) {                          //如果 p_sem 为空
499         *p_err =  OS_ERR_OBJ_PTR_NULL;                   //返回错误类型为“内核对象为空”
500         return ((OS_OBJ_QTY)0u);                         //返回0（有错误），不继续执行
501     }
502     switch (opt) {                                                  //根据选项分类处理
503         case OS_OPT_PEND_ABORT_1:                                   //如果选项在预期内
504         case OS_OPT_PEND_ABORT_ALL:
505         case OS_OPT_PEND_ABORT_1   | OS_OPT_POST_NO_SCHED:
506         case OS_OPT_PEND_ABORT_ALL | OS_OPT_POST_NO_SCHED:
507              break;                                             //不处理，直接跳出
508
509         default:                                                //如果选项超出预期
510             *p_err =  OS_ERR_OPT_INVALID;                       //返回错误类型为“选项非法”
511              return ((OS_OBJ_QTY)0u);                           //返回0（有错误），不继续执行
512     }
513 #endif
514
515 #if OS_CFG_OBJ_TYPE_CHK_EN > 0u                 //如果使能了对象类型检测
516     if (p_sem->Type != OS_OBJ_TYPE_SEM) {       //如果 p_sem 不是多值信号量类型
517         *p_err =  OS_ERR_OBJ_TYPE;              //返回错误类型为“内核对象类型错误”
518         return ((OS_OBJ_QTY)0u);                //返回0（有错误），不继续执行
519     }
520 #endif
```

```
521
522     CPU_CRITICAL_ENTER();                                   //关中断
523     p_pend_list = &p_sem->PendList;                         //获取 p_sem 的等待列表到 p_pend_list
524     if (p_pend_list->NbrEntries == (OS_OBJ_QTY)0u) {        //如果没有任务在等待 p_sem
525         CPU_CRITICAL_EXIT();                                //开中断
526         *p_err =  OS_ERR_PEND_ABORT_NONE;                   //返回错误类型为“没有任务在等待”
527         return ((OS_OBJ_QTY)0u);                            //返回0（有错误），不继续执行
528     }
529
530     OS_CRITICAL_ENTER_CPU_EXIT();                           //加锁调度器，并开中断
531     nbr_tasks = 0u;
532     ts        = OS_TS_GET();                                //获取时间戳
533     while (p_pend_list->NbrEntries > (OS_OBJ_QTY)0u) {      //如果有任务在等待 p_sem
534         p_tcb = p_pend_list->HeadPtr->TCBPtr;               //获取优先级最高的等待任务
535         OS_PendAbort((OS_PEND_OBJ *)((void *)p_sem),        //中止该任务对 p_sem 的等待
536                      p_tcb,
537                      ts);
538         nbr_tasks++;
539         if (opt != OS_OPT_PEND_ABORT_ALL) {                 //如果不是选择了中止所有等待任务
540             break;                                          //立即跳出，不再继续中止
541         }
542     }
543     OS_CRITICAL_EXIT_NO_SCHED();                            //减锁调度器，但不调度
544
545     if ((opt & OS_OPT_POST_NO_SCHED) == (OS_OPT)0u) {       //如果选择了任务调度
546         OSSched();                                          //进行任务调度
547     }
548
549    *p_err = OS_ERR_NONE;    //返回错误类型为“无错误”
550     return (nbr_tasks);     //返回被中止的任务数目
551 }
552 #endif
```

图 6-11 OSSemPendAbort () 函数

OSSemPendAbort () 函数会调用一个更加底层的中止等待函数来执行当前任务对多值信号量的等待，该函数就是 OS_PendAbort()。OS_PendAbort() 函数不仅仅用来中止对多值信号量的等待，还可以中止对互斥信号量、消息队列、事件标志组、任务消息队列或任务信号量的等待。OS_PendAbort() 函数的定义位于“os_core.c”。

os_cfg.h | os_sem.c | os.h | os_core.c

```
927  void  OS_PendAbort (OS_PEND_OBJ  *p_obj,    //被等待对象的类型
928                      OS_TCB       *p_tcb,    //任务控制块指针
929                      CPU_TS        ts)       //等待被中止时的时间戳
930  {
931      switch (p_tcb->TaskState) {                                 //根据任务状态分类处理
932          case OS_TASK_STATE_RDY:                                 //如果任务是就绪状态
933          case OS_TASK_STATE_DLY:                                 //如果任务是延时状态
934          case OS_TASK_STATE_SUSPENDED:                           //如果任务是挂起状态
935          case OS_TASK_STATE_DLY_SUSPENDED:                       //如果任务是在延时中被挂起
936               break;                                             //这些情况均与等待无关，直接跳出
937
938          case OS_TASK_STATE_PEND:                                //如果任务是无期限等待状态
939          case OS_TASK_STATE_PEND_TIMEOUT:                        //如果任务是有期限等待状态
940               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
941                   OS_PendAbort1(p_obj,                           //强制解除任务对某一对象的等待
942                                 p_tcb,
943                                 ts);
944               }
945  #if (OS_MSG_EN > 0u)                                            //如果使能了任务队列或消息队列
946               p_tcb->MsgPtr     = (void      *)0;               //清除（复位）任务的消息域
947               p_tcb->MsgSize    = (OS_MSG_SIZE)0u;
948  #endif
949               p_tcb->TS         = ts;                            //保存等待被中止时的时间戳到任务控制块
950               if (p_obj != (OS_PEND_OBJ *)0) {                   //如果等待对象非空
951                   OS_PendListRemove(p_tcb);                      //将任务从所有等待列表中移除
952               }
953               OS_TaskRdy(p_tcb);                                 //让任务进准备运行
954               p_tcb->TaskState  = OS_TASK_STATE_RDY;             //修改任务状态为就绪状态
955               p_tcb->PendStatus = OS_STATUS_PEND_ABORT;          //标记任务的等待被中止
956               p_tcb->PendOn     = OS_TASK_PEND_ON_NOTHING;       //标记任务目前没有等待任何对象
957               break;                                             //跳出
958
959          case OS_TASK_STATE_PEND_SUSPENDED:                      //如果任务在无期限等待中被挂起
960          case OS_TASK_STATE_PEND_TIMEOUT_SUSPENDED:              //如果任务在有期限等待中被挂起
961               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
962                   OS_PendAbort1(p_obj,                           //强制解除任务对某一对象的等待
963                                 p_tcb,
964                                 ts);
965               }
966  #if (OS_MSG_EN > 0u)                                            //如果使能了任务队列或消息队列
967               p_tcb->MsgPtr     = (void      *)0;               //清除（复位）任务的消息域
968               p_tcb->MsgSize    = (OS_MSG_SIZE)0u;
969  #endif
970               p_tcb->TS         = ts;                            //保存等待被中止时的时间戳到任务控制块
971               if (p_obj != (OS_PEND_OBJ *)0) {                   //如果等待对象非空
972                   OS_PendListRemove(p_tcb);                      //将任务从所有等待列表中移除
973               }
974               OS_TickListRemove(p_tcb);                          //让任务脱离节拍列表
975               p_tcb->TaskState  = OS_TASK_STATE_SUSPENDED;       //修改任务状态为挂起状态
976               p_tcb->PendStatus = OS_STATUS_PEND_ABORT;          //标记任务的等待被中止
977               p_tcb->PendOn     = OS_TASK_PEND_ON_NOTHING;       //标记任务目前没有等待任何对象
978               break;                                             //跳出
979
980          default:                                                //如果任务状态超出预期
981               break;                                             //不需处理，直接跳出
982      }
983  }
```

图 6-12 OS_PendAbort() 函数

6.1.5 OSSemDel()

OSSemDel () 函数用于删除一个多值信号量。要使用 OSSemDel () 函数，还得事先使能 OS_CFG_SEM_DEL_EN（位于“os_cfg.h”），如下图所示。

os_cfg.h | os_sem.c | os.h | os_core.c

```
74
75                                          /* ---------------------------- SEMAPHORES -----
76   #define  OS_CFG_SEM_EN                  1u   //使能或禁用多值信号量
77   #define  OS_CFG_SEM_DEL_EN              1u   //使能或禁用 OSSemDel() 函数
78   #define  OS_CFG_SEM_PEND_ABORT_EN       1u   //使能或禁用 OSSemPendAbort() 函数
79   #define  OS_CFG_SEM_SET_EN              1u   //使能或禁用 OSSemSet() 函数
80
```

图 6-13

OSSemDel () 函数的信息如下表所示。

表 17 OSSemDel ()

函数原型	OS_OBJ_QTY OSSemDel (OS_SEM *p_sem, OS_OPT opt, OS_ERR *p_err);			
功能	删除一个多值信号量。			
参数	p_sem	多值信号量指针。		
	opt	选项	OS_OPT_DEL_NO_PEND	如果没有任务等待 p_sem，才删除 p_sem。
			OS_OPT_DEL_ALWAYS	必须删除 p_sem。
	p_err	返回错误类型	OS_ERR_NONE	没错误。
			OS_ERR_DEL_ISR	该函数在中断中被调用。
			OS_ERR_OBJ_PTR_NULL	p_sem 为空。
			OS_ERR_OBJ_TYPE	p_sem 不是多值信号量类型。
			OS_ERR_OPT_INVALID	选项非法。
			OS_ERR_TASK_WAITING	还有任务在等待该信号量。
返回值	✧ 0，没有任务在等待该信号量，或者有错误产生。 ✧ >0，信号量被删除前等待其的任务数。			
注意事项	✧ 不可以在中断中调用该函数。			

OSSemDel () 函数的定义位于“os_sem.c”。

```
os_cfg_app.c  os_sem.c  os_core.c*  os_cfg.h
162 #if OS_CFG_SEM_DEL_EN > 0u                   //如果使能了 OSSemDel() 函数
163 OS_OBJ_QTY  OSSemDel (OS_SEM  *p_sem,       //多值信号量指针
164                       OS_OPT   opt,         //选项
165                       OS_ERR  *p_err)       //返回错误类型
166 {
167     OS_OBJ_QTY      cnt;
168     OS_OBJ_QTY      nbr_tasks;
169     OS_PEND_DATA   *p_pend_data;
170     OS_PEND_LIST   *p_pend_list;
171     OS_TCB         *p_tcb;
172     CPU_TS          ts;
173     CPU_SR_ALLOC();
174
175
176
177 #ifdef OS_SAFETY_CRITICAL                       //如果使能（默认禁用）了安全检测
178     if (p_err == (OS_ERR *)0) {                 //如果错误类型实参为空
179         OS_SAFETY_CRITICAL_EXCEPTION();         //执行安全检测异常函数
180         return ((OS_OBJ_QTY)0);                 //返回0（有错误），不继续执行
181     }
182 #endif
183
```

```
184 #if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u          //如果使能了中断中非法调用检测
185     if (OSIntNestingCtr > (OS_NESTING_CTR)0) { //如果该函数在中断中被调用
186        *p_err = OS_ERR_DEL_ISR;                //返回错误类型为“在中断中删除”
187         return ((OS_OBJ_QTY)0);                //返回0（有错误），不继续执行
188     }
189 #endif
190 
191 #if OS_CFG_ARG_CHK_EN > 0u                     //如果使能了参数检测
192     if (p_sem == (OS_SEM *)0) {                //如果 p_sem 为空
193        *p_err = OS_ERR_OBJ_PTR_NULL;           //返回错误类型为“内核对象为空”
194         return ((OS_OBJ_QTY)0);                //返回0（有错误），不继续执行
195     }
196     switch (opt) {                             //根据选项分类处理
197         case OS_OPT_DEL_NO_PEND:               //如果选项在预期之内
198         case OS_OPT_DEL_ALWAYS:
199              break;                            //直接跳出
200 
201         default:                               //如果选项超出预期
202             *p_err = OS_ERR_OPT_INVALID;       //返回错误类型为“选项非法”
203              return ((OS_OBJ_QTY)0);           //返回0（有错误），不继续执行
204     }
205 #endif
206 
207 #if OS_CFG_OBJ_TYPE_CHK_EN > 0u                //如果使能了对象类型检测
208     if (p_sem->Type != OS_OBJ_TYPE_SEM) {      //如果 p_sem 不是多值信号量类型
209        *p_err = OS_ERR_OBJ_TYPE;               //返回错误类型为“内核对象类型错误”
210         return ((OS_OBJ_QTY)0);                //返回0（有错误），不继续执行
211     }
212 #endif
```

```
213 
214     CPU_CRITICAL_ENTER();                      //关中断
215     p_pend_list = &p_sem->PendList;            //获取信号量的等待列表到 p_pend_list
216     cnt         = p_pend_list->NbrEntries;     //获取等待该信号量的任务数
217     nbr_tasks   = cnt;
218     switch (opt) {                             //根据选项分类处理
219         case OS_OPT_DEL_NO_PEND:               //如果只在没有任务等待的情况下删除信号量
220             if (nbr_tasks == (OS_OBJ_QTY)0) { //如果没有任务在等待该信号量
221 #if OS_CFG_DBG_EN > 0u                         //如果使能了调试代码和变量
222                 OS_SemDbgListRemove(p_sem);    //将该信号量从信号量调试列表移除
223 #endif
224                 OSSemQty--;                    //信号量数目减1
225                 OS_SemClr(p_sem);              //清除信号量内容
226                 CPU_CRITICAL_EXIT();           //开中断
227                *p_err = OS_ERR_NONE;           //返回错误类型为“无错误”
228             } else {                           //如果有任务在等待该信号量
229                 CPU_CRITICAL_EXIT();           //开中断
230                *p_err = OS_ERR_TASK_WAITING;   //返回错误类型为“有任务在等待该信号量”
231             }
232             break;
233 
```

```
234         case OS_OPT_DEL_ALWAYS:                                 //如果必须删除信号量
235              OS_CRITICAL_ENTER_CPU_EXIT();                      //锁调度器，并开中断
236              ts = OS_TS_GET();                                  //获取时间戳
237              while (cnt > 0u) {                                 //逐个移除该信号量等待列表中的任务
238                  p_pend_data = p_pend_list->HeadPtr;
239                  p_tcb       = p_pend_data->TCBPtr;
240                  OS_PendObjDel((OS_PEND_OBJ *)((void *)p_sem),
241                                p_tcb,
242                                ts);
243                  cnt--;
244              }
245 #if OS_CFG_DBG_EN > 0u                                           //如果使能了调试代码和变量
246              OS_SemDbgListRemove(p_sem);                        //将该信号量从信号量调试列表移除
247 #endif
248              OSSemQty--;                                        //信号量数目减1
249              OS_SemClr(p_sem);                                  //清除信号量内容
250              OS_CRITICAL_EXIT_NO_SCHED();                       //减锁调度器，但不进行调度
251              OSSched();                                         //任务调度，执行最高优先级的就绪任务
252             *p_err = OS_ERR_NONE;                               //返回错误类型为“无错误”
253              break;
254
255         default:                                                //如果选项超出预期
256              CPU_CRITICAL_EXIT();                               //开中断
257             *p_err = OS_ERR_OPT_INVALID;                        //返回错误类型为“选项非法”
258              break;
259     }
260     return ((OS_OBJ_QTY)nbr_tasks);                             //返回删除信号量前等待其的任务数
261 }
262 #endif
```

图 6-14 OSSemDel () 函数

OSSemDel () 函数会调用一个更加底层的删除等待对象的函数来执行对互斥信号量的删除，该函数就是 OS_PendObjDel ()。OS_PendObjDel () 函数不仅仅用来删除互斥信号量，还可以删除多值信号量、消息队列、事件标志组、任务消息队列或任务信号量。OS_PendObjDel () 函数的定义位于“os_core.c”。

os_cfg_app.c | os_sem.c | os_core.c | os_cfg.h

```
1693 void  OS_PendObjDel (OS_PEND_OBJ  *p_obj,   //被删除对象的类型
1694                     OS_TCB       *p_tcb,   //任务控制块指针
1695                     CPU_TS        ts)      //信号量被删除时的时间戳
1696 {
1697     switch (p_tcb->TaskState) {                                 //根据任务状态分类处理
1698         case OS_TASK_STATE_RDY:                                 //如果任务是就绪状态
1699         case OS_TASK_STATE_DLY:                                 //如果任务是延时状态
1700         case OS_TASK_STATE_SUSPENDED:                           //如果任务是挂起状态
1701         case OS_TASK_STATE_DLY_SUSPENDED:                       //如果任务是在延时中被挂起
1702              break;                                             //这些情况均与等待无关，直接跳出
1703
1704         case OS_TASK_STATE_PEND:                                //如果任务是无期限等待状态
1705         case OS_TASK_STATE_PEND_TIMEOUT:                        //如果任务是有期限等待状态
1706              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
1707                  OS_PendObjDel1(p_obj,                          //强制解除任务对某一对象的等待
1708                                 p_tcb,
1709                                 ts);
1710              }
1711 #if (OS_MSG_EN > 0u)                                            //如果使能了任务队列或消息队列
1712              p_tcb->MsgPtr     = (void *)0;                    //清除（复位）任务的消息域
1713              p_tcb->MsgSize    = (OS_MSG_SIZE)0u;
1714 #endif
1715              p_tcb->TS         = ts;                           //保存等待被中止时的时间戳到任务控制块
1716              OS_PendListRemove(p_tcb);                         //将任务从所有等待列表中移除
1717              OS_TaskRdy(p_tcb);                                //让任务进准备运行
1718              p_tcb->TaskState  = OS_TASK_STATE_RDY;            //修改任务状态为就绪状态
1719              p_tcb->PendStatus = OS_STATUS_PEND_DEL;           //标记任务的等待对象被删除
1720              p_tcb->PendOn     = OS_TASK_PEND_ON_NOTHING;      //标记任务目前没有等待任何对象
1721              break;                                            //跳出
1722
```

```
1723          case OS_TASK_STATE_PEND_SUSPENDED:                    //如果任务在无期限等待中被挂起
1724          case OS_TASK_STATE_PEND_TIMEOUT_SUSPENDED:            //如果任务在有期限等待中被挂起
1725               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
1726                   OS_PendObjDel1(p_obj,                        //强制解除任务对某一对象的等待
1727                                  p_tcb,
1728                                  ts);
1729               }
1730 #if (OS_MSG_EN > 0u)                                  //如果使能了任务队列或消息队列
1731               p_tcb->MsgPtr     = (void      *)0;  //清除（复位）任务的消息域
1732               p_tcb->MsgSize    = (OS_MSG_SIZE)0u;
1733 #endif
1734               p_tcb->TS         = ts;                           //保存等待被中止时的时间戳到任务控制块
1735               OS_TickListRemove(p_tcb);                         //让任务脱离节拍列表
1736               OS_PendListRemove(p_tcb);                         //将任务从所有等待列表中移除
1737               p_tcb->TaskState  = OS_TASK_STATE_SUSPENDED;  //修改任务状态为挂起状态
1738               p_tcb->PendStatus = OS_STATUS_PEND_DEL;       //标记任务的等待对象被删除
1739               p_tcb->PendOn     = OS_TASK_PEND_ON_NOTHING;  //标记任务目前没有等待任何对象
1740               break;                                            //跳出
1741
1742          default:                                               //如果任务状态超出预期
1743               break;                                            //不需处理，直接跳出
1744      }
1745 }
```

图 6-15 OS_PendObjDel () 函数

6.1.6 OSSemSet()

OSSemSet () 函数用于设置多值信号量的计数值。要使用 OSSemSet () 函数，还得事先使能 OS_CFG_SEM_SET_EN（位于“os_cfg.h”），如下图所示。

os_cfg.h | os_sem.c | os.h | os_core.c

```
74
75                                                  /* ---------------------------- SEMAPHORES -----
76  #define OS_CFG_SEM_EN                    1u    //使能或禁用多值信号量
77  #define OS_CFG_SEM_DEL_EN                1u    //使能或禁用 OSSemDel() 函数
78  #define OS_CFG_SEM_PEND_ABORT_EN         1u    //使能或禁用 OSSemPendAbort() 函数
79  #define OS_CFG_SEM_SET_EN                1u    //使能或禁用 OSSemSet() 函数
80
```

图 6-16

OSSemSet () 函数的信息如下表所示。

表 18 OSSemSet ()

函数原型	void OSSemSet (OS_SEM *p_sem, OS_SEM_CTR cnt, OS_ERR *p_err);			
功能	设置多值信号量的计数值。			
参数	p_sem	定时器控制块指针。		
	cnt	要设置的信号量计数值。		
	p_err	返回错误类型	OS_ERR_NONE	没错误。
			OS_ERR_SET_ISR	函数在中断中被调用。
			OS_ERR_OBJ_PTR_NULL	p_sem 为空。
			OS_ERR_OBJ_TYPE	p_sem 不是多值信号量类型。
			OS_ERR_TASK_WAITING	还有任务在等待 p_sem。
返回	无。			

值	
注 意 事项	✧ 不可以在中断中调用该函数。

OSSemSet () 函数的定义位于“os_sem.c”。

os_cfg_app.c | os_sem.c | os_core.c | os_cfg.h

```
675 #if OS_CFG_SEM_SET_EN > 0u                //如果使能 OSSemSet() 函数
676 void  OSSemSet (OS_SEM      *p_sem,      //多值信号量指针
677                 OS_SEM_CTR   cnt,        //信号量计数值
678                 OS_ERR      *p_err)      //返回错误类型
679 {
680     OS_PEND_LIST  *p_pend_list;
681     CPU_SR_ALLOC();
682
683
684
685 #ifdef OS_SAFETY_CRITICAL                //如果使能（默认禁用）了安全检测
686     if (p_err == (OS_ERR *)0) {          //如果错误类型实参为空
687         OS_SAFETY_CRITICAL_EXCEPTION();  //执行安全检测异常函数
688         return;                          //返回0（有错误），不继续执行
689     }
690 #endif
691
692 #if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u           //如果使能了中断中非法调用检测
693     if (OSIntNestingCtr > (OS_NESTING_CTR)0) {   //如果该函数在中断中被调用
694         *p_err = OS_ERR_SET_ISR;                 //返回错误类型为“在中断中设置”
695         return;                                  //返回0（有错误），不继续执行
696     }
697 #endif

698
699 #if OS_CFG_ARG_CHK_EN > 0u               //如果使能了参数检测
700     if (p_sem == (OS_SEM *)0) {          //如果 p_sem 为空
701         *p_err = OS_ERR_OBJ_PTR_NULL;    //返回错误类型为“内核对象为空”
702         return;                          //返回0（有错误），不继续执行
703     }
704 #endif
705
706 #if OS_CFG_OBJ_TYPE_CHK_EN > 0u              //如果使能了对象类型检测
707     if (p_sem->Type != OS_OBJ_TYPE_SEM) {    //如果 p_sem 不是多值信号量类型
708         *p_err = OS_ERR_OBJ_TYPE;            //返回错误类型为“内核对象类型错误”
709         return;                              //返回0（有错误），不继续执行
710     }
711 #endif

712
713     *p_err = OS_ERR_NONE;                                   //返回错误类型为“无错误”
714     CPU_CRITICAL_ENTER();                                   //关中断
715     if (p_sem->Ctr > (OS_SEM_CTR)0) {                       //如果信号量计数值>0，说明没
716         p_sem->Ctr = cnt;                                   //被等待可以直接设置计数值。
717     } else {                                                //如果信号量计数值=0
718         p_pend_list = &p_sem->PendList;                     //获取信号量的等待列表
719         if (p_pend_list->NbrEntries == (OS_OBJ_QTY)0) {     //如果没任务在等待信号量
720             p_sem->Ctr = cnt;                               //可以直接设置信号计数值
721         } else {                                            //如果有任务在等待信号量
722             *p_err      = OS_ERR_TASK_WAITING;              //返回错误类型为“有任务等待”
723         }
724     }
725     CPU_CRITICAL_EXIT();
726 }
727 #endif
```

图 6-17 OSSemSet () 函数

6.2 实例演示

6.2.1 实例 1

本节实例将多值信号量用于标志事件，标志按键 KEY1 是否被单击，如果 KEY1 被单击，就切换 LED1 的亮灭状态。本例程中在“app.c”创建两个应用任务 AppTaskKey 和 AppTaskLed1。AppTaskKey 任务负责扫描按键，当 KEY1 被单击时，就发布多值信号量。AppTaskLed1 任务则在等待信号量，当检测到多值信号量被发布时，就会切换 LED1 的亮灭状态。

该例程已经存放在配套资料的下图路径。

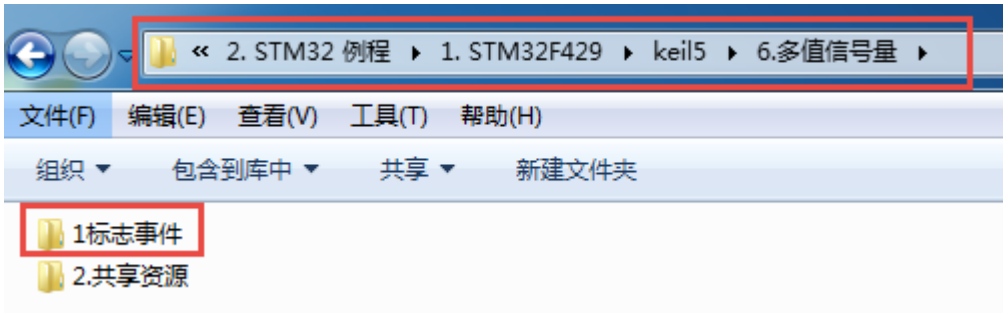

图 6-18 例程路径

本例程需用使用 LED、USART1 和按键，所以工程中需要添加其驱动文件和初始化。用户可参照前面相关实例，这里不再赘述。另外，值得注意，这里按键驱动文件的添加与 LED 和 USART1 略有不同，按键驱动文件“bsp_key.c”里的延时函数需要改用 uC/OS 的延时函数（须包含头文件“includes.h”），如下图所示。

```
69 uint8_t Key_Scan ( GPIO_TypeDef * GPIOx, uint16_t GPIO_Pin, uint8_t ucPushState, uint8_t * pKeyPress )
70 {
71   uint8_t ucKeyState, ucRet = 0;
72
73   OS_ERR      err;
74
75   ucKeyState = GPIO_ReadInputDataBit ( GPIOx, GPIO_Pin );
76
77   if ( ucKeyState == ucPushState )
78   {
79 //    Key_Delay( 10000 );          //改成uC/OS的延时函数
80     OSTimeDlyHMSM ( 0, 0, 0, 20, OS_OPT_TIME_DLY, & err );
81
82     ucKeyState = GPIO_ReadInputDataBit ( GPIOx, GPIO_Pin );
83
84     if ( ucKeyState == ucPushState )
85       * pKeyPress = 1;
86   }
87
88   if ( ( ucKeyState != ucPushState ) && ( * pKeyPress == 1 ) )
89   {
90     ucRet = 1;
91     * pKeyPress = 0;
92   }
93
94   return ucRet;
95
96 }
```

图 6-19 板载驱动文件改用 uC/OS 的延时函数

在起始任务 AppTaskStart() 中，在创建应用任务之前，创建了应用任务需要使用到的多值信号量 SemOfKey（必须保证该信号量在被使用到之前创建好）。

```
os_sem.c | bsp_key.c | app_cfg.h | app.c
142 static  void  AppTaskStart (void *p_arg)
143 {
144     CPU_INT32U  cpu_clk_freq;
145     CPU_INT32U  cnts;
146     OS_ERR      err;
147
148
149    (void)p_arg;
150
151     BSP_Init();                                                 //板级初始化
152     CPU_Init();                                                 //初始化 CPU 组件（时间戳、关中断时间测
153
154     cpu_clk_freq = BSP_CPU_ClkFreq();                           //获取 CPU 内核时钟频率（SysTick 工作时
155     cnts = cpu_clk_freq / (CPU_INT32U)OSCfg_TickRate_Hz;        //根据用户设定的时钟节拍频率计算 SysTic
156     OS_CPU_SysTickInit(cnts);                                   //调用 SysTick 初始化函数，设置定时器计
157
158     Mem_Init();                                                 //初始化内存管理组件（堆内存池和内存池表
159
160 #if OS_CFG_STAT_TASK_EN > 0u                                    //如果使能（默认使能）了统计任务
161     OSStatTaskCPUUsageInit(&err);                               //计算没有应用任务（只有空闲任务）运行时
162 #endif                                                          //容量（决定 OS_Stat_IdleCtrMax 的值，
163                                                                 //使用率使用）。
164     CPU_IntDisMeasMaxCurReset();                                //复位（清零）当前最大关中断时间
165
166
167     /* 创建多值信号量 SemOfKey */
168     OSSemCreate((OS_SEM      *)&SemOfKey,   //指向信号量变量的指针
169                 (CPU_CHAR    *)"SemOfKey",  //信号量的名字
170                 (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
171                 (OS_ERR      *)&err);       //错误类型
172
173
174     /* 创建 AppTaskKey 任务 */
```

图 6-20 创建多值信号量

任务函数 AppTaskKey() 的定义如下。任务中每隔 20ms 扫描一次 KEY1，如果 KEY1 被单击，就发布信号量 SemOfKey。

```
os_sem.c | bsp_key.c | app_cfg.h | app.c
211 *********************************************************************************************************
212 *                                             KEY TASK
213 *********************************************************************************************************
214 */
215 static  void  AppTaskKey ( void * p_arg )
216 {
217   OS_ERR      err;
218
219   uint8_t ucKey1Press = 0;
220
221
222   (void)p_arg;
223
224
225   while (DEF_TRUE) {                                                    //任务体
226     if( Key_Scan ( macKEY1_GPIO_PORT, macKEY1_GPIO_PIN, 1, & ucKey1Press ) ) //如果KEY1被单击
227       OSSemPost((OS_SEM  *)&SemOfKey,                                   //发布SemOfKey
228                 (OS_OPT   )OS_OPT_POST_ALL,                             //发布给所有等待任务
229                 (OS_ERR  *)&err);                                       //返回错误类型
230
231     OSTimeDlyHMSM ( 0, 0, 0, 20, OS_OPT_TIME_DLY, & err );              //每20ms扫描一次
232
233   }
234
235 }
```

图 6-21 AppTaskKey() 任务函数

任务函数 AppTaskLed1 () 的定义如下。任务中会等待多值信号量 SemOfKey 的发布，直到 SemOfKey 被发布，任务才继续运行。当任务接收到 SemOfKey 时，就会切换 LED1 的亮灭状态，并打印 SemOfKey 从被发布到被接收的时间段。

os_sem.c | bsp_key.c | app_cfg.h | app.c

```
239 *********************************************************************************************************
240 *                                              LED1 TASK
241 *********************************************************************************************************
242 */
243
244 static  void  AppTaskLed1 ( void * p_arg )
245 {
246     OS_ERR         err;
247     CPU_INT32U     cpu_clk_freq;
248     CPU_TS         ts_sem_post, ts_sem_get;
249     CPU_SR_ALLOC();  //使用到临界段（在关/开中断时）时必需该宏，该宏声明和定义一个局部变
250                      //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）
251                      //，开中断时将该值还原。
252     (void)p_arg;
253
254
255     cpu_clk_freq = BSP_CPU_ClkFreq();                   //获取CPU时钟，时间戳是以该时钟计数
256
257
258     while (DEF_TRUE) {                                  //任务体
259
260       OSSemPend ((OS_SEM   *)&SemOfKey,                 //等待该信号量被发布
261                  (OS_TICK   )0,                         //无期限等待
262                  (OS_OPT    )OS_OPT_PEND_BLOCKING,      //如果没有信号量可用就等待
263                  (CPU_TS   *)&ts_sem_post,              //获取信号量最后一次被发布的时间戳
264                  (OS_ERR   *)&err);                     //返回错误类型
265
266       ts_sem_get = OS_TS_GET();                         //获取解除等待时的时间戳
267
268       macLED1_TOGGLE ();                                //切换LED1的亮灭状态
269
270       OS_CRITICAL_ENTER();                              //进入临界段，不希望下面串口打印遭到中断
271
272       printf ( "\r\n发布信号量的时间戳是%d", ts_sem_post );
273       printf ( "\r\n解除等待状态的时间戳是%d", ts_sem_get );
274       printf ( "\r\n接收到信号量与发布信号量的时间相差%dus\r\n",
275               ( ts_sem_get - ts_sem_post ) / ( cpu_clk_freq / 1000000 ) );
276
277       OS_CRITICAL_EXIT();
278
279     }
```

图 6-22 AppTaskLed1 () 任务函数

把 STM32 的 USART1 连接至电脑的串口调试助手，编译和下载程序到秉火 STM32 开发板，运行程序。当用户每单击一次 KEY1 时，就会看到 LED1 切换一次亮灭状态，并且串口调试助手上会打印 SemOfKey 最后一次被发布的时间戳、被接收时的时间戳和从被发布到被接收的时间段。

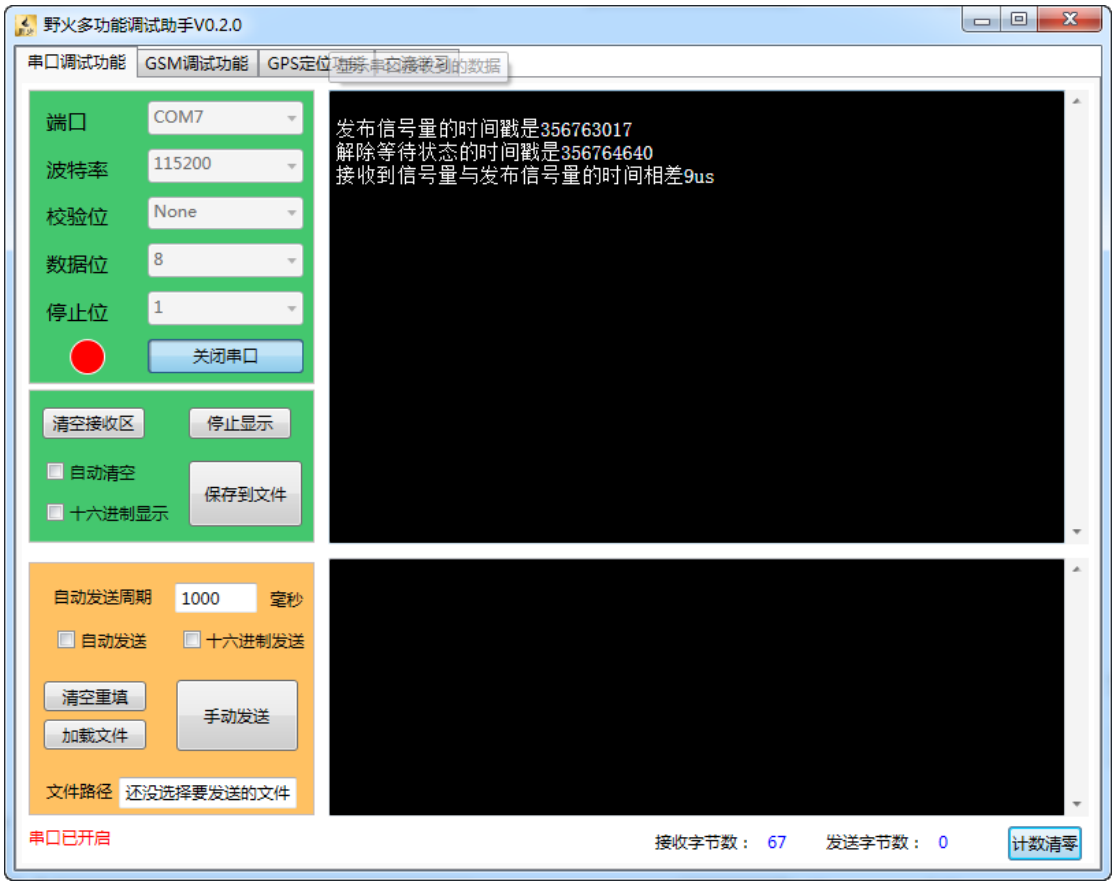

图 6-23 串口调试助手

6.2.2 实例 2

本节实例将多值信号量用于共享资源，例程模拟停车场资源管理，多值信号量的计数值代表停车场还剩下的空位。当单击按键 KEY1 时，表示有汽车要占位，如果还有空位，就获得一个停车位。当单击按键 KEY2 时，表示有汽车开走，释放了一个停车位，停车场就多了一个空位。

本例程中在“app.c”创建两个应用任务 AppTaskKey1 和 AppTaskKey2。AppTaskKey1 任务负责处理按键 KEY1 的单击事件，表示有新进汽车要占位。AppTaskKey2 负责处理按键 KEY2 的单击事件，表示有汽车开离停车场，释放了一个停车位。

该例程已经存放在配套资料的下图路径。

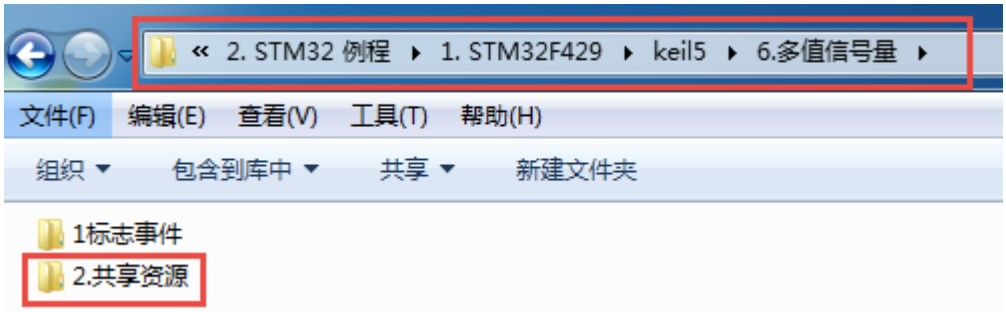

图 6-24 例程路径

本例程需用使用 USART1 和按键，所以工程中需要添加其驱动文件和初始化。用户可参照前面相关实例，这里不再赘述。另外，值得注意，这里按键驱动文件的添加与 USART1 略有不同，按键驱动文件“bsp_key.c”里的延时函数需要改用 uC/OS 的延时函数（须包含头文件“includes.h”），如下图所示。

```
69 uint8_t Key_Scan ( GPIO_TypeDef * GPIOx, uint16_t GPIO_Pin, uint8_t ucPushState, uint8_t * pKeyPress )
70 {
71   uint8_t ucKeyState, ucRet = 0;
72
73   OS_ERR      err;
74
75   ucKeyState = GPIO_ReadInputDataBit ( GPIOx, GPIO_Pin );
76
77   if ( ucKeyState == ucPushState )
78   {
79 //    Key_Delay( 10000 );          //改成uC/OS的延时函数
80     OSTimeDlyHMSM ( 0, 0, 0, 20, OS_OPT_TIME_DLY, & err );
81
82     ucKeyState = GPIO_ReadInputDataBit ( GPIOx, GPIO_Pin );
83
84     if ( ucKeyState == ucPushState )
85       * pKeyPress = 1;
86   }
87
88   if ( ( ucKeyState != ucPushState ) && ( * pKeyPress == 1 ) )
89   {
90     ucRet = 1;
91     * pKeyPress = 0;
92   }
93
94   return ucRet;
95
96 }
```

图 6-25 板载驱动文件改用 uC/OS 的延时函数

在起始任务 AppTaskStart() 中，在创建应用任务之前，创建了应用任务需要使用到的多值信号量 SemOfKey（必须保证该信号量在被使用到之前创建好）。这里 SemOfKey 的计数值初始化为 5，表示停车场目前还剩下 5 个空位。

app_cfg.h | app.c

```
142 static  void  AppTaskStart (void *p_arg)
143 {
144     CPU_INT32U  cpu_clk_freq;
145     CPU_INT32U  cnts;
146     OS_ERR      err;
147
148
149     (void)p_arg;
150
151     BSP_Init();                                                 //板
152     CPU_Init();                                                 //初
153
154     cpu_clk_freq = BSP_CPU_ClkFreq();                           //获
155     cnts = cpu_clk_freq / (CPU_INT32U)OSCfg_TickRate_Hz;        //根
156     OS_CPU_SysTickInit(cnts);                                   //计
157
158     Mem_Init();                                                 //初
159
160 #if OS_CFG_STAT_TASK_EN > 0u                                    //如
161     OSStatTaskCPUUsageInit(&err);                               //计
162 #endif                                                          //空
163                                                                 //使
164     CPU_IntDisMeasMaxCurReset();                                //复
165
166
167     /* 创建多值信号量 SemOfKey */
168     OSSemCreate((OS_SEM      *)&SemOfKey,     //指向信号量变量的指针
169                 (CPU_CHAR    *)"SemOfKey",    //信号量的名字
170                 (OS_SEM_CTR   )5,             //表示现有资源数目
171                 (OS_ERR      *)&err);         //错误类型
172
173
174     /* 创建 AppTaskKey1 任务 */
175     OSTaskCreate((OS_TCB     *)&AppTaskKey1TCB,
```

图 6-26 创建多值信号量

任务函数 AppTaskKey1() 的定义如下。任务主要处理 KEY1 的单击事件，如果 KEY1 被单击，就请求多值信号量 SemOfKey（新进汽车想要一个停车位）。如果不能立即请求到 SemOfKey，也不原地等待（立马开离停车场）。

app_cfg.h | app.c

```
211 /*********************************************************************************************************
212 *                                              KEY1 TASK
213 *********************************************************************************************************
214 */
215 static  void  AppTaskKey1 ( void * p_arg )
216 {
217   OS_ERR      err;
218   OS_SEM_CTR  ctr;
219   CPU_SR_ALLOC();  //使用到临界段（在关/开中断时）时必需该宏，该宏声明和定义一个局部变
220                    //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）
221                   //，开中断时将该值还原。
222   uint8_t ucKey1Press = 0;
223
224
225   (void)p_arg;
226
227
228   while (DEF_TRUE) {                                                  //任务体
229     if( Key_Scan ( macKEY1_GPIO_PORT, macKEY1_GPIO_PIN, 1, & ucKey1Press ) ) //如果KEY1被单击
230     {
231       ctr = OSSemPend ((OS_SEM   *)&SemOfKey,                 //等待该信号量 SemOfKey
232                        (OS_TICK   )0,                         //下面选择不等待，该参无效
233                        (OS_OPT    )OS_OPT_PEND_NON_BLOCKING,//如果没信号量可用不等待
234                        (CPU_TS    *)0,                        //不获取时间戳
235                        (OS_ERR    *)&err);                    //返回错误类型
236
237       OS_CRITICAL_ENTER();                                    //进入临界段
238
239       if ( err == OS_ERR_NONE )
240         printf ( "\r\nKEY1被单击：成功申请到停车位，剩下%d个停车位。\r\n", ctr );
241       else if ( err == OS_ERR_PEND_WOULD_BLOCK )
242         printf ( "\r\nkey1被单击：不好意思，现在停车场已满，请等待！\r\n" );
243
244       OS_CRITICAL_EXIT();
245
246     }
247
248     OSTimeDlyHMSM ( 0, 0, 0, 20, OS_OPT_TIME_DLY, & err );              //每20ms扫描一次
249
250   }
251
252 }
```

图 6-27 AppTaskKey1() 任务函数

任务函数 AppTaskKey2() 的定义如下。任务主要处理 KEY2 的单击事件，如果 KEY2 被单击，就发布多值信号量 SemOfKey（有汽车开离停车场，释放了一个停车位）。

app_cfg.h | app.c

```
256 *********************************************************************************************************
257 *                                          KEY2 TASK
258 *********************************************************************************************************
259 */
260 static  void  AppTaskKey2 ( void * p_arg )
261 {
262   OS_ERR      err;
263   OS_SEM_CTR  ctr;
264   CPU_SR_ALLOC();  //使用到临界段（在关/开中断时）时必需该宏，该宏声明和定义一个局部变
265                    //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）
266                    //，开中断时将该值还原。
267   uint8_t ucKey2Press = 0;
268
269
270   (void)p_arg;
271
272
273   while (DEF_TRUE) {                                                  //任务体
274     if( Key_Scan ( macKEY2_GPIO_PORT, macKEY2_GPIO_PIN, 1, & ucKey2Press ) ) //如果KEY1被单击
275     {
276       ctr = OSSemPost((OS_SEM  *)&SemOfKey,                           //发布SemOfKey
277                       (OS_OPT   )OS_OPT_POST_ALL,                     //发布给所有等待任务
278                       (OS_ERR  *)&err);                               //返回错误类型
279
280       OS_CRITICAL_ENTER();                                            //进入临界段
281
282       printf ( "\r\nKEY2被单击：释放1个停车位，剩下%d个停车位。\r\n", ctr );
283
284       OS_CRITICAL_EXIT();
285
286     }
287
288     OSTimeDlyHMSM ( 0, 0, 0, 20, OS_OPT_TIME_DLY, & err );             //每20ms扫描一次
289
290   }
291
292 }
```

图 6-28 AppTaskKey2() 任务函数

把 STM32 的 USART1 连接至电脑的串口调试助手，编译和下载程序到秉火 STM32 开发板，运行程序。用户看通过按键 KEY1 和 KEY2 分别申请和释放“停车位”，并通过串口调试助手查看打印信息。

图 6-29 串口调试助手

6.3 章末总结

多值信号量是 uC/OS 操作系统的一个内核对象，主要用于标志事件的是否发生和资源的共享管理。

使用多值信号量之前必须先创建它，创建多值信号量使用 OSSemCreate () 函数。

OSSemPend() 函数用于申请享用多值信号量，如果多值信号量不可用，可以选择等待或者不等待。与之相对应，OSSemPost () 函数则用于发布多值信号量，也就是增加多值信号量的可用资源。

OSSemPendAbort () 函数用于中止任务对一个多值信号量的等待。OSSemDel () 函数用于删除一个多值信号量。OSSemSet () 函数用于设置多值信号量的计数值。

第7章 互斥信号量

互斥信号量是 uC/OS 操作系统的一个内核对象，与多值信号量非常相似，但它是二值的，只能是 0 或 1，所以也叫二值信号量，主要用于保护资源。

7.1 原理简述

如果想要使用互斥信号量，就必须事先使能互斥信号量。互斥信号量的使能位于“os_cfg.h”。

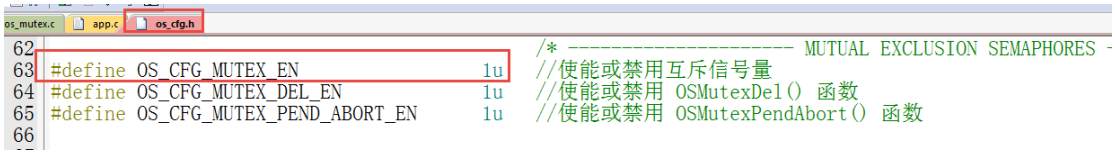

图 7-1

7.1.1 OSMutexCreate ()

要使用 uC/OS 的互斥信号量必须先声明和创建互斥信号量，调用 OSMutexCreate () 函数可以创建一个多值信号量。OSMutexCreate () 函数的信息如下表所示。

表 19 OSMutexCreate ()

函数原型	void OSMutexCreate (OS_MUTEX *p_mutex, CPU_CHAR *p_name, OS_ERR *p_err);			
功能	创建一个互斥信号量。			
参数	p_mutex	互斥信号量指针。		
	p_name	互斥信号量名称。		
	p_err	返回错误类型	OS_ERR_NONE	没错误
			OS_ERR_CREATE_ISR	在中断中调用该函数
			OS_ERR_ILLEGAL_CREATE_RUN_TIME	在调用 OSSafetyCriticalStart() 函数后创建内核对象
			OS_ERR_NAME	p_name 为空指针
			OS_ERR_OBJ_CREATED	信号量已经被创建过
			OS_ERR_OBJ_PTR_NULL	p_mutex 是个空指针
返回	无。			

值	
注意事项	✧ 创建前必须先为 p_mutex 声明一个互斥信号量对象（OS_MUTEX）。 ✧ 不可以在中断中调用该函数。

OSMutexCreate () 函数的定义位于“os_mutex.c”。

```
67  void  OSMutexCreate (OS_MUTEX  *p_mutex, //互斥信号量指针
68                       CPU_CHAR  *p_name,  //取信号量的名称
69                       OS_ERR    *p_err)   //返回错误类型
70  {
71      CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必需该宏，该宏声明和定义一个局部变
72                      //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）
73                      //，开中断时将该值还原。
74
75  #ifdef OS_SAFETY_CRITICAL                //如果使能（默认禁用）了安全检测
76      if (p_err == (OS_ERR *)0) {          //如果错误类型实参为空
77          OS_SAFETY_CRITICAL_EXCEPTION(); //执行安全检测异常函数
78          return;                          //返回，不继续执行
79      }
80  #endif
81
82  #ifdef OS_SAFETY_CRITICAL_IEC61508                //如果使能（默认禁用）了安全关键
83      if (OSSafetyCriticalStartFlag == DEF_TRUE) { //如果是在调用 OSSafetyCriticalStart() 后创建
84          *p_err = OS_ERR_ILLEGAL_CREATE_RUN_TIME;  //错误类型为“非法创建内核对象”
85          return;                                   //返回，不继续执行
86      }
87  #endif
88
89  #if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u            //如果使能（默认使能）了中断中非法调用检测
90      if (OSIntNestingCtr > (OS_NESTING_CTR)0) {    //如果该函数是在中断中被调用
91          *p_err = OS_ERR_CREATE_ISR;               //错误类型为“在中断函数中定时”
92          return;                                   //返回，不继续执行
93      }
94  #endif

95
96  #if OS_CFG_ARG_CHK_EN > 0u                //如果使能（默认使能）了参数检测
97      if (p_mutex == (OS_MUTEX *)0) {       //如果参数 p_mutex 为空
98          *p_err = OS_ERR_OBJ_PTR_NULL;     //错误类型为“创建对象为空”
99          return;                           //返回，不继续执行
100     }
101 #endif
102
103     OS_CRITICAL_ENTER();                  //进入临界段，初始化互斥信号量指标
104     p_mutex->Type              =  OS_OBJ_TYPE_MUTEX;  //标记创建对象数据结构为互斥信号量
105     p_mutex->NamePtr           =  p_name;
106     p_mutex->OwnerTCBPtr       = (OS_TCB       *)0;
107     p_mutex->OwnerNestingCtr   = (OS_NESTING_CTR)0;   //互斥信号量目前可用
108     p_mutex->TS                = (CPU_TS        )0;
109     p_mutex->OwnerOriginalPrio =  OS_CFG_PRIO_MAX;
110     OS_PendListInit(&p_mutex->PendList);              //初始化该互斥信号量的等待列表
111
112 #if OS_CFG_DBG_EN > 0u                //如果使能（默认使能）了调试代码和变量
113     OS_MutexDbgListAdd(p_mutex); //将该信号量添加到互斥信号量双向调试链表
114 #endif
115     OSMutexQty++;                 //互斥信号量个数加1
116
117     OS_CRITICAL_EXIT_NO_SCHED(); //退出临界段（无调度）
118    *p_err = OS_ERR_NONE;          //错误类型为“无错误”
119 }
```

图 7-2 OSMutexCreate () 函数

其中，调用了 OS_PendListInit() 函数初始化了互斥信号量的等待列表。每个互斥信号量都有一个等待列表，凡是等待该互斥信号量的的任务都会被插入到这个等待列表，方便高效管理。OS_PendListInit() 函数的定义位于“os_core.c”。

```
1319 void  OS_PendListInit (OS_PEND_LIST  *p_pend_list)
1320 {
1321     p_pend_list->HeadPtr    = (OS_PEND_DATA *)0;    //复位等待列表的所有成员
1322     p_pend_list->TailPtr    = (OS_PEND_DATA *)0;
1323     p_pend_list->NbrEntries = (OS_OBJ_QTY    )0;
1324 }
```

图 7-3 OS_PendListInit() 函数

如果使能了 OS_CFG_DBG_EN（位于“os_cfg.h”），创建互斥信号量时还会调用 OS_MutexDbgListAdd() 函数将该多值信号量插入到一个多值信号量调试列表，是为方便调试所设。OS_MutexDbgListAdd() 函数的定义位于“os_mutex.c”。

```
794 #if OS_CFG_DBG_EN > 0u                       //如果使能（默认使能）了调试代码和变量
795 void  OS_MutexDbgListAdd (OS_MUTEX  *p_mutex) //将该互斥信号量插入到互斥信号量列表的最前端
796 {
797     p_mutex->DbgNamePtr               = (CPU_CHAR *)((void *)" "); //先不指向任何任务的名称
798     p_mutex->DbgPrevPtr               = (OS_MUTEX *)0;             //将该信号量作为列表的最前端
799     if (OSMutexDbgListPtr == (OS_MUTEX *)0) {                      //如果列表为空
800         p_mutex->DbgNextPtr           = (OS_MUTEX *)0;             //该信号量的下一个元素也为空
801     } else {                                                       //如果列表非空
802         p_mutex->DbgNextPtr           =  OSMutexDbgListPtr;        //列表原来的首元素作为该信号量的下一个元素
803         OSMutexDbgListPtr->DbgPrevPtr =  p_mutex;                  //原来的首元素的前面变为了该信号量
804     }
805     OSMutexDbgListPtr                 =  p_mutex;                  //该信号量成为列表的新首元素
806 }
```

图 7-4 OS_MutexDbgListAdd() 函数

7.1.2 OSMutexPost ()

OSMutexPost () 函数用于释放（发布，提交）互斥信号量。OSMutexPost () 函数的信息如下表所示。

表 20 OSMutexPost ()

<table>
<tr><td>函数原型</td><td colspan="4">void OSMutexPost (OS_MUTEX *p_mutex,
OS_OPT opt,
OS_ERR *p_err);</td></tr>
<tr><td>功能</td><td colspan="4">释放互斥信号量。</td></tr>
<tr><td rowspan="6">参数</td><td>p_mutex</td><td colspan="3">互斥信号量指针。</td></tr>
<tr><td rowspan="2">opt</td><td>OS_OPT_POST_NONE</td><td colspan="2">释放信号量后，如果信号量可用，而且有任务正在等待，就（默认）进行任务调度。</td></tr>
<tr><td>OS_OPT_POST_NO_SCHED</td><td colspan="2">释放信号量后，如果信号量可用，而且有任务正在等待，不进行任务调度，继续运行当前任务。</td></tr>
<tr><td rowspan="4">p_err</td><td rowspan="4">返回错误类型</td><td>OS_ERR_NONE</td><td>无错误。</td></tr>
<tr><td>OS_ERR_MUTEX_NESTING</td><td>p_mutex 被嵌套。</td></tr>
<tr><td>OS_ERR_MUTEX_NOT_OWNER</td><td>当前任务不持有 p_mutex。</td></tr>
<tr><td>OS_ERR_OBJ_PTR_NULL</td><td>p_mutex 为空。</td></tr>
<tr><td></td><td></td><td>OS_ERR_OBJ_TYPE</td><td>p_mutex 不是多值信号量类型。</td></tr>
</table>

			OS_ERR_POST_ISR	在中断中释放多值信号量。
返回值	无。			
注意事项	✧ 当前任务必须在先有 p_mutex 的情况下才能去释放它。 ✧ 不可以在中断中调用该函数。			

OSMutexPost () 函数的定义也位于“os_mutex.c

```
648 void  OSMutexPost (OS_MUTEX  *p_mutex, //互斥信号量指针
649                    OS_OPT     opt,      //选项
650                    OS_ERR    *p_err)    //返回错误类型
651 {
652     OS_PEND_LIST  *p_pend_list;
653     OS_TCB        *p_tcb;
654     CPU_TS         ts;
655     CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必需该宏，该宏声明和定义一个局部变
656                     //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）
657                     //，开中断时将该值还原。
658
659 #ifdef OS_SAFETY_CRITICAL                 //如果使能（默认禁用）了安全检测
660     if (p_err == (OS_ERR *)0) {           //如果错误类型实参为空
661         OS_SAFETY_CRITICAL_EXCEPTION(); //执行安全检测异常函数
662         return;                           //返回，不继续执行
663     }
664 #endif
665
666 #if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u          //如果使能了中断中非法调用检测
667     if (OSIntNestingCtr > (OS_NESTING_CTR)0) { //如果该函数在中断中被调用
668        *p_err = OS_ERR_POST_ISR;                //错误类型为“在中断中等待”
669         return;                                 //返回，不继续执行
670     }
671 #endif

672
673 #if OS_CFG_ARG_CHK_EN > 0u                  //如果使能了参数检测
674     if (p_mutex == (OS_MUTEX *)0) {         //如果 p_mutex 为空
675        *p_err = OS_ERR_OBJ_PTR_NULL;        //错误类型为“内核对象为空”
676         return;                             //返回，不继续执行
677     }
678     switch (opt) {                          //根据选项分类处理
679         case OS_OPT_POST_NONE:              //如果选项在预期内，不处理
680         case OS_OPT_POST_NO_SCHED:
681              break;
682
683         default:                            //如果选项超出预期
684             *p_err =  OS_ERR_OPT_INVALID;   //错误类型为“选项非法”
685              return;                        //返回，不继续执行
686     }
687 #endif
688
689 #if OS_CFG_OBJ_TYPE_CHK_EN > 0u                 //如果使能了对象类型检测
690     if (p_mutex->Type != OS_OBJ_TYPE_MUTEX) { //如果 p_mutex 的类型不是互斥信号量类型
691        *p_err = OS_ERR_OBJ_TYPE;                //返回，不继续执行
692         return;
693     }
694 #endif
```

```
695
696     CPU_CRITICAL_ENTER();                                   //关中断
697     if (OSTCBCurPtr != p_mutex->OwnerTCBPtr) { //如果当前运行任务不持有该信号量
698         CPU_CRITICAL_EXIT();                                //开中断
699        *p_err = OS_ERR_MUTEX_NOT_OWNER;                     //错误类型为“任务不持有该信号量”
700         return;                                             //返回，不继续执行
701     }
702
703     OS_CRITICAL_ENTER_CPU_EXIT();                                   //锁调度器，开中断
704     ts          = OS_TS_GET();                                      //获取时间戳
705     p_mutex->TS = ts;                                               //存储信号量最后一次被释放的时间戳
706     p_mutex->OwnerNestingCtr--;                                     //信号量的嵌套数减1
707     if (p_mutex->OwnerNestingCtr > (OS_NESTING_CTR)0) { //如果信号量仍被嵌套
708         OS_CRITICAL_EXIT();                                         //解锁调度器
709        *p_err = OS_ERR_MUTEX_NESTING;                               //错误类型为“信号量被嵌套”
710         return;                                                     //返回，不继续执行
711     }
712     /* 如果信号量未被嵌套，已可用 */
713     p_pend_list = &p_mutex->PendList;                       //获取信号量的等待列表
714     if (p_pend_list->NbrEntries == (OS_OBJ_QTY)0) {  //如果没有任务在等待该信号量
715         p_mutex->OwnerTCBPtr     = (OS_TCB       *)0;//请空信号量持有者信息
716         p_mutex->OwnerNestingCtr = (OS_NESTING_CTR)0;
717         OS_CRITICAL_EXIT();                                 //解锁调度器
718        *p_err = OS_ERR_NONE;                                //错误类型为“无错误”
719         return;                                             //返回，不继续执行
720     }
721     /* 如果有任务在等待该信号量 */
722     if (OSTCBCurPtr->Prio != p_mutex->OwnerOriginalPrio) { //如果当前任务的优先级被改过
723         OS_RdyListRemove(OSTCBCurPtr);                      //从就绪列表移除当前任务
724         OSTCBCurPtr->Prio = p_mutex->OwnerOriginalPrio;     //还原当前任务的优先级
725         OS_PrioInsert(OSTCBCurPtr->Prio);                   //在优先级表格插入这个优先级
726         OS_RdyListInsertTail(OSTCBCurPtr);                  //将当前任务插入就绪列表尾端
727         OSPrioCur           = OSTCBCurPtr->Prio;            //更改当前任务优先级变量的值
728     }
729
730     p_tcb                       = p_pend_list->HeadPtr->TCBPtr; //获取等待列表的首端任务
731     p_mutex->OwnerTCBPtr        = p_tcb;                        //将信号量交给该任务
732     p_mutex->OwnerOriginalPrio = p_tcb->Prio;
733     p_mutex->OwnerNestingCtr    = (OS_NESTING_CTR)1;            //开始嵌套
734     /* 释放信号量给该任务 */
735     OS_Post((OS_PEND_OBJ *)((void *)p_mutex),
736             (OS_TCB      *)p_tcb,
737             (void        *)0,
738             (OS_MSG_SIZE  )0,
739             (CPU_TS       )ts);
740
741     OS_CRITICAL_EXIT_NO_SCHED();                            //减锁调度器，但不执行任务调度
742
743     if ((opt & OS_OPT_POST_NO_SCHED) == (OS_OPT)0) { //如果 opt 没选择“发布时不调度任务”
744         OSSched();                                          //任务调度
745     }
746
747    *p_err = OS_ERR_NONE;                                    //错误类型为“无错误”
748 }
```

图 7-5 OSMutexPost () 函数

在 OSMutexPost () 函数中，会调用 OS_Post() 函数发布内核对象。OS_Post() 函数是一个底层的发布函数，它不仅仅用来发布互斥信号量，还可以发布多值信号量、消息队列、事件标志组、任务消息队列和任务信号量。OS_Post() 函数的定义位于“os_core.c”。

os_sem.c | os.h | os_core.c | os_cfg.h

```
1844 void  OS_Post (OS_PEND_OBJ  *p_obj,      //内核对象类型指针
1845             OS_TCB       *p_tcb,      //任务控制块
1846             void         *p_void,     //消息
1847             OS_MSG_SIZE   msg_size,   //消息大小
1848             CPU_TS        ts)         //时间戳
1849 {
1850     switch (p_tcb->TaskState) {                             //根据任务状态分类处理
1851         case OS_TASK_STATE_RDY:                             //如果任务处于就绪状态
1852         case OS_TASK_STATE_DLY:                             //如果任务处于延时状态
1853         case OS_TASK_STATE_SUSPENDED:                       //如果任务处于挂起状态
1854         case OS_TASK_STATE_DLY_SUSPENDED:                   //如果任务处于延时中被挂起状态
1855              break;                                         //不用处理，直接跳出
```

```
1856
1857            case OS_TASK_STATE_PEND:                                    //如果任务处于无期限等待状态
1858            case OS_TASK_STATE_PEND_TIMEOUT:                            //如果任务处于有期限等待状态
1859                 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
1860                     OS_Post1(p_obj,                                    //标记哪个内核对象被发布
1861                              p_tcb,
1862                              p_void,
1863                              msg_size,
1864                              ts);
1865                 } else {                                               //如果任务不是在等待多个信号量或消息队列
1866#if (OS_MSG_EN > 0u)
1867                     p_tcb->MsgPtr  = p_void;                           //保存消息到等待任务
1868                     p_tcb->MsgSize = msg_size;
1869#endif
1870                     p_tcb->TS      = ts;                               //保存时间戳到等待任务
1871                 }
1872                 if (p_obj != (OS_PEND_OBJ *)0) {                       //如果内核对象为空
1873                     OS_PendListRemove(p_tcb);                          //从等待列表移除该等待任务
1874#if OS_CFG_DBG_EN > 0u                                                  //如果使能了调试代码和变量
1875                     OS_PendDbgNameRemove(p_obj,                        //移除内核对象的调试名
1876                                          p_tcb);
1877#endif
1878                 }
1879                 OS_TaskRdy(p_tcb);                                     //让该等待任务准备运行
1880                 p_tcb->TaskState  = OS_TASK_STATE_RDY;                 //任务状态改为就绪状态
1881                 p_tcb->PendStatus = OS_STATUS_PEND_OK;                 //清除等待状态
1882                 p_tcb->PendOn     = OS_TASK_PEND_ON_NOTHING;           //标记不再等待
1883                 break;
```

```
1884
1885            case OS_TASK_STATE_PEND_SUSPENDED:                          //如果任务在无期限等待中被挂起
1886            case OS_TASK_STATE_PEND_TIMEOUT_SUSPENDED:                  //如果任务在有期限等待中被挂起
1887                 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
1888                     OS_Post1(p_obj,                                    //标记哪个内核对象被发布
1889                              p_tcb,
1890                              p_void,
1891                              msg_size,
1892                              ts);
1893                 } else {                                               //如果任务不在等待多个信号量或消息队列
1894#if (OS_MSG_EN > 0u)                                                   //如果使能了调试代码和变量
1895                     p_tcb->MsgPtr  = p_void;                           //保存消息到等待任务
1896                     p_tcb->MsgSize = msg_size;
1897#endif
1898                     p_tcb->TS      = ts;                               //保存时间戳到等待任务
1899                 }
1900                 OS_TickListRemove(p_tcb);                              //从节拍列表移除该等待任务
1901                 if (p_obj != (OS_PEND_OBJ *)0) {                       //如果内核对象为空
1902                     OS_PendListRemove(p_tcb);                          //从等待列表移除该等待任务
1903#if OS_CFG_DBG_EN > 0u                                                  //如果使能了调试代码和变量
1904                     OS_PendDbgNameRemove(p_obj,                        //移除内核对象的调试名
1905                                          p_tcb);
1906#endif
1907                 }
1908                 p_tcb->TaskState  = OS_TASK_STATE_SUSPENDED;           //任务状态改为被挂起状态
1909                 p_tcb->PendStatus = OS_STATUS_PEND_OK;                 //清除等待状态
1910                 p_tcb->PendOn     = OS_TASK_PEND_ON_NOTHING;           //标记不再等待
1911                 break;
1912
1913            default:                                                    //如果任务状态超出预期
1914                 break;                                                 //直接跳出
1915        }
1916    }
```

图 7-6 OS_Post() 函数

7.1.3 OSMutexPend ()

与 OSMutexPost () 互斥信号量释放函数相对应，OSMutexPend() 函数用于申请（等待，请求）获取一个互斥信号量。

表 21 OSMutexPend()

函数原	void OSMutexPend (OS_MUTEX *p_mutex, OS_TICK timeout, OS_OPT opt,

<table>
<tr><td>型</td><td colspan="5">CPU_TS *p_ts,
OS_ERR *p_err);</td></tr>
<tr><td>功能</td><td colspan="5">申请一个互斥信号量。</td></tr>
<tr><td rowspan="8">参数</td><td>p_mutex</td><td colspan="4">互斥信号量指针。</td></tr>
<tr><td>timeout</td><td colspan="4">等待超时时间（单位：时钟节拍），0 代表无期限等待。opt 为 OS_OPT_PEND_BLOCKING 时该参数才起作用。</td></tr>
<tr><td rowspan="2">opt</td><td rowspan="2">选项</td><td>OS_OPT_PEND_BLOCKING</td><td colspan="2">如果不能立即获得信号量，就堵塞当前任务，继续等待信号量。</td></tr>
<tr><td>OS_OPT_PEND_NON_BLOCKING</td><td colspan="2">如果不能立即获得信号量，不堵塞当前任务，不继续等待信号量。</td></tr>
<tr><td>p_ts</td><td>时间戳</td><td colspan="3">用于存储信号量最后一次被发布的时间戳，或者等待被中止的时间戳，或者信号量被删除时的时间戳，具体返回哪个时间戳，要根据返回的 p_err 判断。该参数可以为 NULL，表示用户不需要获得时间戳。</td></tr>
<tr><td rowspan="11">p_err</td><td rowspan="11">返回错误类型</td><td>OS_ERR_NONE</td><td colspan="2">没错误，获得信号量。</td></tr>
<tr><td>OS_ERR_OBJ_DEL</td><td colspan="2">p_sem 被删除。</td></tr>
<tr><td>OS_ERR_OBJ_PTR_NULL</td><td colspan="2">p_sem 为空。</td></tr>
<tr><td>OS_ERR_OBJ_TYPE</td><td colspan="2">p_sem 不是信号量类型对象。</td></tr>
<tr><td>OS_ERR_OPT_INVALID</td><td colspan="2">opt 非法。</td></tr>
<tr><td>OS_ERR_PEND_ABORT</td><td colspan="2">等待被另一个任务中止。</td></tr>
<tr><td>OS_ERR_PEND_ISR</td><td colspan="2">在中断中被调用。</td></tr>
<tr><td>OS_ERR_PEND_WOULD_BLOCK</td><td colspan="2">缺乏堵塞。</td></tr>
<tr><td>OS_ERR_SCHED_LOCKED</td><td colspan="2">调度器被锁。</td></tr>
<tr><td>OS_ERR_STATUS_INVALID</td><td colspan="2">等待状态非法。</td></tr>
<tr><td>OS_ERR_TIMEOUT</td><td colspan="2">等待超时。</td></tr>
<tr><td>返回值</td><td colspan="5">✧ 0，有错误。
✧ 其他值，信号量的计数值。</td></tr>
<tr><td>注意事项</td><td colspan="5">✧ 不可以在中断中调用该函数。</td></tr>
</table>

OSMutexPend() 函数的定义也位于“os_mutex.c”。

os_mutex.c | os_sem.c | os_core.c | os_prio.c | os.h

```
339 void  OSMutexPend (OS_MUTEX  *p_mutex, //互斥信号量指针
340                     OS_TICK    timeout, //超时时间（节拍）
341                     OS_OPT     opt,     //选项
342                     CPU_TS    *p_ts,    //时间戳
343                     OS_ERR    *p_err)   //返回错误类型
344 {
345     OS_PEND_DATA  pend_data;
346     OS_TCB       *p_tcb;
347     CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必需该宏，该宏声明和
348                     //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
349                     // SR（临界段关中断只需保存SR），开中断时将该值还原。
350
351 #ifdef OS_SAFETY_CRITICAL                 //如果使能（默认禁用）了安全检测
352     if (p_err == (OS_ERR *)0) {           //如果错误类型实参为空
353         OS_SAFETY_CRITICAL_EXCEPTION(); //执行安全检测异常函数
354         return;                           //返回，不继续执行
355     }
356 #endif
357
358 #if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u          //如果使能了中断中非法调用检测
359     if (OSIntNestingCtr > (OS_NESTING_CTR)0) { //如果该函数在中断中被调用
360         *p_err = OS_ERR_PEND_ISR;               //错误类型为“在中断中等待”
361         return;                                 //返回，不继续执行
362     }
363 #endif

364
365 #if OS_CFG_ARG_CHK_EN > 0u                  //如果使能了参数检测
366     if (p_mutex == (OS_MUTEX *)0) {         //如果 p_mutex 为空
367         *p_err = OS_ERR_OBJ_PTR_NULL;       //返回错误类型为“内核对象为空”
368         return;                             //返回，不继续执行
369     }
370     switch (opt) {                          //根据选项分类处理
371         case OS_OPT_PEND_BLOCKING:          //如果选项在预期内
372         case OS_OPT_PEND_NON_BLOCKING:
373              break;
374
375         default:                            //如果选项超出预期
376             *p_err = OS_ERR_OPT_INVALID; //错误类型为“选项非法”
377              return;                        //返回，不继续执行
378     }
379 #endif
380
381 #if OS_CFG_OBJ_TYPE_CHK_EN > 0u                 //如果使能了对象类型检测
382     if (p_mutex->Type != OS_OBJ_TYPE_MUTEX) { //如果 p_mutex 非互斥信号量类型
383         *p_err = OS_ERR_OBJ_TYPE;               //错误类型为“内核对象类型错误”
384         return;                                 //返回，不继续执行
385     }
386 #endif
```

```
387
388     if (p_ts != (CPU_TS *)0) {  //如果 p_ts 非空
389         *p_ts  = (CPU_TS  )0;    //初始化（清零）p_ts，待用于返回时间戳
390     }
391
392     CPU_CRITICAL_ENTER();                                       //关中断
393     if (p_mutex->OwnerNestingCtr == (OS_NESTING_CTR)0) { //如果信号量可用
394         p_mutex->OwnerTCBPtr       =  OSTCBCurPtr;          //让当前任务持有信号量
395         p_mutex->OwnerOriginalPrio =  OSTCBCurPtr->Prio; //保存持有任务的优先级
396         p_mutex->OwnerNestingCtr   = (OS_NESTING_CTR)1;  //开始嵌套
397         if (p_ts != (CPU_TS *)0) {                              //如果 p_ts 非空
398             *p_ts  = p_mutex->TS;                               //返回信号量的时间戳记录
399         }
400         CPU_CRITICAL_EXIT();                                    //开中断
401        *p_err = OS_ERR_NONE;                                    //错误类型为“无错误”
402         return;                                                 //返回，不继续执行
403     }
404     /* 如果信号量不可用 */
405     if (OSTCBCurPtr == p_mutex->OwnerTCBPtr) { //如果当前任务已经持有该信号量
406         p_mutex->OwnerNestingCtr++;             //信号量前套数加1
407         if (p_ts != (CPU_TS *)0) {              //如果 p_ts 非空
408             *p_ts  = p_mutex->TS;               //返回信号量的时间戳记录
409         }
410         CPU_CRITICAL_EXIT();                    //开中断
411        *p_err = OS_ERR_MUTEX_OWNER;             //错误类型为“任务已持有信号量”
412         return;                                 //返回，不继续执行
413     }

414     /* 如果当前任务非持有该信号量 */
415     if ((opt & OS_OPT_PEND_NON_BLOCKING) != (OS_OPT)0) {//如果选择了不堵塞任务
416         CPU_CRITICAL_EXIT();                            //开中断
417        *p_err = OS_ERR_PEND_WOULD_BLOCK;                //错误类型为“渴求堵塞”
418         return;                                         //返回，不继续执行
419     } else {                                            //如果选择了堵塞任务
420         if (OSSchedLockNestingCtr > (OS_NESTING_CTR)0) {//如果调度器被锁
421             CPU_CRITICAL_EXIT();                        //开中断
422            *p_err = OS_ERR_SCHED_LOCKED;                //错误类型为“调度器被锁”
423             return;                                     //返回，不继续执行
424         }
425     }

426     /* 如果调度器未被锁 */
427     OS_CRITICAL_ENTER_CPU_EXIT();                       //锁调度器，并重开中断
428     p_tcb = p_mutex->OwnerTCBPtr;                       //获取信号量持有任务
429     if (p_tcb->Prio > OSTCBCurPtr->Prio) {              //如果持有任务优先级低于当前任务
430         switch (p_tcb->TaskState) {                     //根据持有任务的任务状态分类处理
431             case OS_TASK_STATE_RDY:                     //如果是就绪状态
432                  OS_RdyListRemove(p_tcb);               //从就绪列表移除持有任务
433                  p_tcb->Prio = OSTCBCurPtr->Prio;       //提升持有任务的优先级到当前任务
434                  OS_PrioInsert(p_tcb->Prio);            //将该优先级插入优先级表格
435                  OS_RdyListInsertHead(p_tcb);           //将持有任务插入就绪列表
436                  break;                                 //跳出
437
438             case OS_TASK_STATE_DLY:                     //如果是延时状态
439             case OS_TASK_STATE_DLY_SUSPENDED:           //如果是延时中被挂起状态
440             case OS_TASK_STATE_SUSPENDED:               //如果是被挂起状态
441                  p_tcb->Prio = OSTCBCurPtr->Prio;       //提升持有任务的优先级到当前任务
442                  break;                                 //跳出
443
444             case OS_TASK_STATE_PEND:                    //如果是无期限等待状态
445             case OS_TASK_STATE_PEND_TIMEOUT:            //如果是有期限等待状态
446             case OS_TASK_STATE_PEND_SUSPENDED:          //如果是无期限等待中被挂起状态
447             case OS_TASK_STATE_PEND_TIMEOUT_SUSPENDED:  //如果是有期限等待中被挂起状态
448                  OS_PendListChangePrio(p_tcb,           //改变持有任务在等待列表的位置
449                                        OSTCBCurPtr->Prio);
450                  break;                                 //跳出
451

452              default:                                   //如果任务状态超出预期
453                   OS_CRITICAL_EXIT();                   //开中断
454                  *p_err = OS_ERR_STATE_INVALID;         //错误类型为“任务状态非法”
455                   return;                               //返回，不继续执行
456         }
457     }
```

```
    /* 堵塞任务，将当前任务脱离就绪列表，并插入到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);

    OS_CRITICAL_EXIT_NO_SCHED();              //开调度器，但不进行调度

    OSSched();                                //调度最高优先级任务运行

    CPU_CRITICAL_ENTER();                     //开中断
    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:               //如果等待正常（获得信号量）
             if (p_ts != (CPU_TS *)0) {       //如果 p_ts 非空
                *p_ts  = OSTCBCurPtr->TS;     //返回信号量最后一次被释放的时间戳
             }
            *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
             }
            *p_err = OS_ERR_PEND_ABORT;       //错误类型为“等待被中止”
             break;                           //跳出

        case OS_STATUS_PEND_TIMEOUT:          //如果超时内为获得信号量
             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;                           //跳出

         case OS_STATUS_PEND_DEL:             //如果信号量已被删除
              if (p_ts != (CPU_TS *)0) {      //如果 p_ts 非空
                 *p_ts  = OSTCBCurPtr->TS;    //返回信号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;                          //跳出

         default:                             //根据等待状态超出预期
             *p_err = OS_ERR_STATUS_INVALID;  //错误类型为“状态非法”
              break;                          //跳出
    }
    CPU_CRITICAL_EXIT();                      //开中断
}
```

图 7-7 OSMutexPend() 函数

OSMutexPend() 函数会调用一个更加底层的等待函数来执行当前任务对互斥信号量的等待，该函数就是 OS_Pend()。与 OS_Post() 函数一样，OS_Pend() 函数不仅仅用来等待互斥信号量，还可以等待多值信号量、消息队列、事件标志组、任务消息队列和任务信号量。OS_Pend() 函数的定义位于“os_core.c”。

```
873 void  OS_Pend (OS_PEND_DATA  *p_pend_data,   //待插入等待列表的元素
874              OS_PEND_OBJ   *p_obj,         //等待的内核对象
875              OS_STATE       pending_on,    //等待哪种对象内核
876              OS_TICK        timeout)       //等待期限
877 {
878     OS_PEND_LIST  *p_pend_list;
879
880
881
882     OSTCBCurPtr->PendOn     = pending_on;                    //资源不可用，开始等待
883     OSTCBCurPtr->PendStatus = OS_STATUS_PEND_OK;             //正常等待中
884
885     OS_TaskBlock(OSTCBCurPtr,                                //阻塞当前运行任务，
886                  timeout);                                   //如果 timeout 非0，把任务插入的节拍列表
887
888     if (p_obj != (OS_PEND_OBJ *)0) {                         //如果等待对象非空
889         p_pend_list          = &p_obj->PendList;             //获取对象的等待列表到 p_pend_list
890         p_pend_data->PendObjPtr = p_obj;                     //保存要等待的对象
891         OS_PendDataInit((OS_TCB      *)OSTCBCurPtr,          //初始化 p_pend_data（待插入等待列表）
892                         (OS_PEND_DATA *)p_pend_data,
893                         (OS_OBJ_QTY    )1);
894         OS_PendListInsertPrio(p_pend_list,                   //按优先级将 p_pend_data 插入到等待列表
895                               p_pend_data);
896     } else {                                                 //如果等待对象为空
897         OSTCBCurPtr->PendDataTblEntries = (OS_OBJ_QTY    )0; //清零当前任务的等待域数据
898         OSTCBCurPtr->PendDataTblPtr     = (OS_PEND_DATA *)0;
899     }
900 #if OS_CFG_DBG_EN > 0u                                       //如果使能了调试代码和变量
901     OS_PendDbgNameAdd(p_obj,                                 //更新信号量的 DbgNamePtr 元素为其等待
902                       OSTCBCurPtr);                          //列表中优先级最高的任务的名称。
903 #endif
904 }
```

图 7-8 OS_Pend() 函数

7.1.4 OSMutexPendAbort()

OSMutexPendAbort() 函数用于中止任务对一个互斥信号量的等待。要使用 OSMutexPendAbort () 函数，还得事先使能 OS_CFG_MUTEX_PEND_ABORT_EN（位于“os_cfg.h”），如下图所示。

```
62                                          /* ---------------------- MUTUAL EXCLUSION SEMAPHORES -
63 #define OS_CFG_MUTEX_EN                1u   //使能或禁用互斥信号量
64 #define OS_CFG_MUTEX_DEL_EN            1u   //使能或禁用 OSMutexDel() 函数
65 #define OS_CFG_MUTEX_PEND_ABORT_EN     1u   //使能或禁用 OSMutexPendAbort() 函数
66
```

图 7-9

OSMutexPendAbort() 函数的信息如下表所示。

表 22 OSMutexPendAbort()

函数原型	OS_OBJ_QTY OSMutexPendAbort (OS_MUTEX *p_mutex, OS_OPT opt, OS_ERR *p_err);			
功能	中止对一个互斥信号量的等待。			
参数	p_mutex	互斥信号量指针。		
	opt	选项。	OS_OPT_PEND_ABORT_1	只中止该信号量等待列表中的最高优先级任务。
			OS_OPT_PEND_ABORT_ALL	中止该信号量等待列表中的所有优先级任务。

			OS_OPT_PEND_ABORT_1 | OS_OPT_POST_NO_SCHED	只中止该信号量等待列表中的最高优先级任务，但不进行任务调度。
			OS_OPT_PEND_ABORT_ALL | OS_OPT_POST_NO_SCHED	中止该信号量等待列表中的所有优先级任务，但不进行任务调度。
	p_err	返回错误类型	OS_ERR_NONE	没错误。
			OS_ERR_OBJ_PTR_NULL	p_mutex 为空。
			OS_ERR_OBJ_TYPE	p_mutex 不是多值信号量类型。
			OS_ERR_OPT_INVALID	选项非法。
			OS_ERR_PEND_ABORT_ISR	该函数在中断中被调用。
			OS_ERR_PEND_ABORT_NONE	没有任务在等待该信号量。
返回值	✧ 0，没有任务在等待该信号量，或者有错误产生。 ✧ >0，被中止的任务数。			
注意事项	✧ 不可以在中断中调用该函数。			

OSMutexPendAbort() 函数的定义位于“os_mutex.c”。

```
537 #if OS_CFG_MUTEX_PEND_ABORT_EN > 0u                  //如果使能了 OSMutexPendAbort()
538 OS_OBJ_QTY  OSMutexPendAbort (OS_MUTEX  *p_mutex,  //互斥信号量指针
539                               OS_OPT     opt,      //选项
540                               OS_ERR    *p_err)    //返回错误类型
541 {
542     OS_PEND_LIST  *p_pend_list;
543     OS_TCB        *p_tcb;
544     CPU_TS         ts;
545     OS_OBJ_QTY     nbr_tasks;
546     CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必需该宏，该宏声明和
547                     //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
548                     // SR（临界段关中断只需保存SR），开中断时将该值还原。
549
550 #ifdef OS_SAFETY_CRITICAL                          //如果使能（默认禁用）了安全检测
551     if (p_err == (OS_ERR *)0) {                    //如果错误类型实参为空
552         OS_SAFETY_CRITICAL_EXCEPTION(); //执行安全检测异常函数
553         return ((OS_OBJ_QTY)0u);                   //返回0（有错误），不继续执行
554     }
555 #endif
```

```
556
557 #if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u                  //如果使能了中断中非法调用检测
558     if (OSIntNestingCtr > (OS_NESTING_CTR)0u) {  //如果该函数在中断中被调用
559         *p_err =  OS_ERR_PEND_ABORT_ISR;                 //错误类型为“在中断中中止等待”
560          return ((OS_OBJ_QTY)0u);                        //返回0（有错误），不继续执行
561     }
562 #endif
563
564 #if OS_CFG_ARG_CHK_EN > 0u                          //如果使能了参数检测
565     if (p_mutex == (OS_MUTEX *)0) {                 //如果 p_sem 为空
566         *p_err =  OS_ERR_OBJ_PTR_NULL;              //错误类型为“对象为空”
567          return ((OS_OBJ_QTY)0u);                   //返回0（有错误），不继续执行
568     }
569     switch (opt) {                                          //根据选项分类处理
570         case OS_OPT_PEND_ABORT_1:                           //如果选项在预期内
571         case OS_OPT_PEND_ABORT_ALL:
572         case OS_OPT_PEND_ABORT_1   | OS_OPT_POST_NO_SCHED:
573         case OS_OPT_PEND_ABORT_ALL | OS_OPT_POST_NO_SCHED:
574              break;                                         //直接跳出
575
576         default:                                    //如果选项超出预期
577             *p_err =  OS_ERR_OPT_INVALID;           //错误类型为“选项非法”
578              return ((OS_OBJ_QTY)0u);               //返回0（有错误），不继续执行
579     }
580 #endif

581
582 #if OS_CFG_OBJ_TYPE_CHK_EN > 0u                     //如果使能了对象类型检测
583     if (p_mutex->Type != OS_OBJ_TYPE_MUTEX) { //如果 p_mutex 不是互斥信号量类型
584         *p_err =  OS_ERR_OBJ_TYPE;                  //错误类型为“对象类型错误”
585          return ((OS_OBJ_QTY)0u);                   //返回0（有错误），不继续执行
586     }
587 #endif

588
589     CPU_CRITICAL_ENTER();                                   //关中断
590     p_pend_list = &p_mutex->PendList;                       //获取 p_mutex 的等待列表
591     if (p_pend_list->NbrEntries == (OS_OBJ_QTY)0u) { //如果没有任务在等待 p_mutex
592         CPU_CRITICAL_EXIT();                                //开中断
593        *p_err =  OS_ERR_PEND_ABORT_NONE;                    //错误类型为“没有任务在等待”
594         return ((OS_OBJ_QTY)0u);                            //返回0（有错误），不继续执行
595     }
596

597     OS_CRITICAL_ENTER_CPU_EXIT();                           //加锁调度器，重开中断
598     nbr_tasks = 0u;
599     ts        = OS_TS_GET();                                //获取时间戳
600     while (p_pend_list->NbrEntries > (OS_OBJ_QTY)0u) { //如果有任务在等待 p_mutex
601         p_tcb = p_pend_list->HeadPtr->TCBPtr;               //获取优先级最高的等待任务
602         OS_PendAbort((OS_PEND_OBJ *)((void *)p_mutex), //中止该任务对p_mutex的等待
603                      p_tcb,
604                      ts);
605         nbr_tasks++;
606         if (opt != OS_OPT_PEND_ABORT_ALL) {                 //如果非选择中止所有等待任务
607             break;                                          //立即跳出，不再继续中止
608         }
609     }
610     OS_CRITICAL_EXIT_NO_SCHED();                            //减锁调度器，但不调度
611
612     if ((opt & OS_OPT_POST_NO_SCHED) == (OS_OPT)0u) {  //如果选择了任务调度
613         OSSched();                                          //进行任务调度
614     }
615
616    *p_err = OS_ERR_NONE;                                    //返错误类型为“无错误”
617     return (nbr_tasks);                                     //返回被中止的任务数目
618 }
619 #endif
```

图 7-10 OSMutexPendAbort() 函数

OSMutexPendAbort() 函数会调用一个更加底层的中止等待函数来执行当前任务对多值

信号量的等待，该函数就是 OS_PendAbort()。OS_PendAbort() 函数不仅仅用来中止对多值信号量的等待，还可以中止对互斥信号量、消息队列、事件标志组、任务消息队列或任务信号量的等待。OS_PendAbort() 函数的定义位于“os_core.c”。

os_cfg.h | os_sem.c | os.h | os_core.c

```
927 void  OS_PendAbort (OS_PEND_OBJ  *p_obj,    //被等待对象的类型
928                     OS_TCB        *p_tcb,    //任务控制块指针
929                     CPU_TS         ts)       //等待被中止时的时间戳
930 {
931     switch (p_tcb->TaskState) {                                 //根据任务状态分类处理
932         case OS_TASK_STATE_RDY:                                 //如果任务是就绪状态
933         case OS_TASK_STATE_DLY:                                 //如果任务是延时状态
934         case OS_TASK_STATE_SUSPENDED:                           //如果任务是挂起状态
935         case OS_TASK_STATE_DLY_SUSPENDED:                       //如果任务是在延时中被挂起
936              break;                                             //这些情况均与等待无关，直接跳出
937
938         case OS_TASK_STATE_PEND:                                //如果任务是无期限等待状态
939         case OS_TASK_STATE_PEND_TIMEOUT:                        //如果任务是有期限等待状态
940              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
941                  OS_PendAbort1(p_obj,                           //强制解除任务对某一对象的等待
942                                p_tcb,
943                                ts);
944              }
945 #if (OS_MSG_EN > 0u)                                            //如果使能了任务队列或消息队列
946              p_tcb->MsgPtr     = (void      *)0;                //清除（复位）任务的消息域
947              p_tcb->MsgSize    = (OS_MSG_SIZE)0u;
948 #endif
949              p_tcb->TS         = ts;                            //保存等待被中止时的时间戳到任务控制块
950              if (p_obj != (OS_PEND_OBJ *)0) {                   //如果等待对象非空
951                  OS_PendListRemove(p_tcb);                      //将任务从所有等待列表中移除
952              }
953              OS_TaskRdy(p_tcb);                                 //让任务进准备运行
954              p_tcb->TaskState  = OS_TASK_STATE_RDY;             //修改任务状态为就绪状态
955              p_tcb->PendStatus = OS_STATUS_PEND_ABORT;          //标记任务的等待被中止
956              p_tcb->PendOn     = OS_TASK_PEND_ON_NOTHING;       //标记任务目前没有等待任何对象
957              break;                                             //跳出
958
959         case OS_TASK_STATE_PEND_SUSPENDED:                      //如果任务在无期限等待中被挂起
960         case OS_TASK_STATE_PEND_TIMEOUT_SUSPENDED:              //如果任务在有期限等待中被挂起
961              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
962                  OS_PendAbort1(p_obj,                           //强制解除任务对某一对象的等待
963                                p_tcb,
964                                ts);
965              }
966 #if (OS_MSG_EN > 0u)                                            //如果使能了任务队列或消息队列
967              p_tcb->MsgPtr     = (void      *)0;                //清除（复位）任务的消息域
968              p_tcb->MsgSize    = (OS_MSG_SIZE)0u;
969 #endif
970              p_tcb->TS         = ts;                            //保存等待被中止时的时间戳到任务控制块
971              if (p_obj != (OS_PEND_OBJ *)0) {                   //如果等待对象非空
972                  OS_PendListRemove(p_tcb);                      //将任务从所有等待列表中移除
973              }
974              OS_TickListRemove(p_tcb);                          //让任务脱离节拍列表
975              p_tcb->TaskState  = OS_TASK_STATE_SUSPENDED;       //修改任务状态为挂起状态
976              p_tcb->PendStatus = OS_STATUS_PEND_ABORT;          //标记任务的等待被中止
977              p_tcb->PendOn     = OS_TASK_PEND_ON_NOTHING;       //标记任务目前没有等待任何对象
978              break;                                             //跳出
979
980         default:                                                //如果任务状态超出预期
981              break;                                             //不需处理，直接跳出
982     }
983 }
```

图 7-11 OS_PendAbort() 函数

7.1.5 OSMutexDel()

OSMutexDel() 函数用于删除一个互斥信号量。要使用 OSMutexDel() 函数，还得事先使能 OS_CFG_MUTEX_DEL_EN（位于“os_cfg.h”），如下图所示。

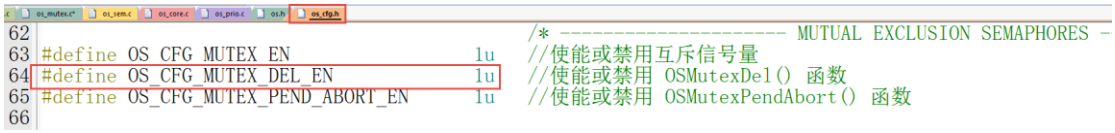

图 7-12

OSMutexDel() 函数的信息如下表所示。

表 23 OSMutexDel()

函数原型	OS_OBJ_QTY OSMutexDel (OS_MUTEX *p_mutex, OS_OPT opt, OS_ERR *p_err);			
功能	删除一个互斥信号量。			
参数	p_sem	互斥信号量指针。		
	opt	选项	OS_OPT_DEL_NO_PEND	如果没有任务等待 p_mutex，才删除 p_mutex。
			OS_OPT_DEL_ALWAYS	必须删除 p_mutex。
	p_err	返回错误类型	OS_ERR_NONE	无错误。
			OS_ERR_DEL_ISR	该函数在中断中被调用。
			OS_ERR_OBJ_PTR_NULL	p_mutex 为空。
			OS_ERR_OBJ_TYPE	p_mutex 不是互斥信号量类型。
			OS_ERR_OPT_INVALID	选项非法。
			OS_ERR_STATE_INVALID	持有信号量任务状态非法。
			OS_ERR_TASK_WAITING	还有任务在等待该信号量。
返回值	✧ 0，没有任务在等待该信号量，或者有错误产生。 ✧ >0，信号量被删除前等待其的任务数。			
注意事项	✧ 不可以在中断中调用该函数。			

OSMutexDel() 函数的定义位于“os_mutex.c”。

os_mutex.c os_cfg.h os_core.c

```
159 #if OS_CFG_MUTEX_DEL_EN > 0u                    //如果使能了 OSMutexDel()
160 OS_OBJ_QTY  OSMutexDel (OS_MUTEX  *p_mutex,  //互斥信号量指针
161                         OS_OPT     opt,      //选项
162                         OS_ERR    *p_err)    //返回错误类型
163 {
164     OS_OBJ_QTY     cnt;
165     OS_OBJ_QTY     nbr_tasks;
166     OS_PEND_DATA  *p_pend_data;
167     OS_PEND_LIST  *p_pend_list;
168     OS_TCB        *p_tcb;
169     OS_TCB        *p_tcb_owner;
170     CPU_TS         ts;
171     CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必需该宏，该宏声明和
172                     //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
173                     // SR（临界段关中断只需保存SR），开中断时将该值还原。
174
175 #ifdef OS_SAFETY_CRITICAL                      //如果使能（默认禁用）了安全检测
176     if (p_err == (OS_ERR *)0) {                //如果错误类型实参为空
177         OS_SAFETY_CRITICAL_EXCEPTION(); //执行安全检测异常函数
178         return ((OS_OBJ_QTY)0);                //返回0（有错误），停止执行
179     }
180 #endif

181
182 #if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u         //如果使能了中断中非法调用检测
183     if (OSIntNestingCtr > (OS_NESTING_CTR)0) {//如果该函数在中断中被调用
184         *p_err = OS_ERR_DEL_ISR;               //错误类型为“在中断中中止等待”
185         return ((OS_OBJ_QTY)0);                //返回0（有错误），停止执行
186     }
187 #endif
188
189 #if OS_CFG_ARG_CHK_EN > 0u                     //如果使能了参数检测
190     if (p_mutex == (OS_MUTEX *)0) {            //如果 p_mutex 为空
191         *p_err = OS_ERR_OBJ_PTR_NULL;          //错误类型为“对象为空”
192         return ((OS_OBJ_QTY)0);                //返回0（有错误），停止执行
193     }
194     switch (opt) {                             //根据选项分类处理
195         case OS_OPT_DEL_NO_PEND:               //如果选项在预期内
196         case OS_OPT_DEL_ALWAYS:
197              break;                            //直接跳出
198
199         default:                               //如果选项超出预期
200             *p_err =  OS_ERR_OPT_INVALID; //错误类型为“选项非法”
201              return ((OS_OBJ_QTY)0);           //返回0（有错误），停止执行
202     }
203 #endif

204
205 #if OS_CFG_OBJ_TYPE_CHK_EN > 0u                //如果使能了对象类型检测
206     if (p_mutex->Type != OS_OBJ_TYPE_MUTEX) { //如果 p_mutex 非互斥信号量类型
207         *p_err = OS_ERR_OBJ_TYPE;              //错误类型为“对象类型错误”
208         return ((OS_OBJ_QTY)0);                //返回0（有错误），停止执行
209     }
210 #endif
211
```

```
212     OS_CRITICAL_ENTER();                                    //进入临界段
213     p_pend_list = &p_mutex->PendList;                       //获取信号量的等待列表
214     cnt         = p_pend_list->NbrEntries;                  //获取等待该信号量的任务数
215     nbr_tasks   = cnt;
216     switch (opt) {                                          //根据选项分类处理
217         case OS_OPT_DEL_NO_PEND:                            //如果只在没任务等待时删除信号量
218              if (nbr_tasks == (OS_OBJ_QTY)0) {              //如果没有任务在等待该信号量
219 #if OS_CFG_DBG_EN > 0u                                      //如果使能了调试代码和变量
220                  OS_MutexDbgListRemove(p_mutex);            //将该信号量从信号量调试列表移除
221 #endif
222                  OSMutexQty--;                              //互斥信号量数目减1
223                  OS_MutexClr(p_mutex);                      //清除信号量内容
224                  OS_CRITICAL_EXIT();                        //退出临界段
225                 *p_err = OS_ERR_NONE;                       //错误类型为“无错误”
226              } else {                                       //如果有任务在等待该信号量
227                  OS_CRITICAL_EXIT();                        //退出临界段
228                 *p_err = OS_ERR_TASK_WAITING;               //错误类型为“有任务正在等待”
229              }
230              break;                                         //跳出
231
```

```
232         case OS_OPT_DEL_ALWAYS:                                         //如果必须删除信号量
233              p_tcb_owner = p_mutex->OwnerTCBPtr;                        //获取信号量持有任务
234              if ((p_tcb_owner       != (OS_TCB *)0) &&                  //如果持有任务存在，
235                  (p_tcb_owner->Prio !=  p_mutex->OwnerOriginalPrio)) {  //而且优先级被提升过。
236                  switch (p_tcb_owner->TaskState) {                      //根据其任务状态处理
237                      case OS_TASK_STATE_RDY:                            //如果是就绪状态
238                           OS_RdyListRemove(p_tcb_owner);                //将任务从就绪列表移除
239                           p_tcb_owner->Prio = p_mutex->OwnerOriginalPrio;//还原任务的优先级
240                           OS_PrioInsert(p_tcb_owner->Prio);             //将该优先级插入优先级表格
241                           OS_RdyListInsertTail(p_tcb_owner);            //将任务重插入就绪列表
242                           break;                                        //跳出
243
244                      case OS_TASK_STATE_DLY:                            //如果是延时状态
245                      case OS_TASK_STATE_SUSPENDED:                      //如果是被挂起状态
246                      case OS_TASK_STATE_DLY_SUSPENDED:                  //如果是延时中被挂起状态
247                           p_tcb_owner->Prio = p_mutex->OwnerOriginalPrio;//还原任务的优先级
248                           break;
249
250                      case OS_TASK_STATE_PEND:                           //如果是无期限等待状态
251                      case OS_TASK_STATE_PEND_TIMEOUT:                   //如果是有期限等待状态
252                      case OS_TASK_STATE_PEND_SUSPENDED:                 //如果是无期等待中被挂状态
253                      case OS_TASK_STATE_PEND_TIMEOUT_SUSPENDED:         //如果是有期等待中被挂状态
254                           OS_PendListChangePrio(p_tcb_owner,            //改变任务在等待列表的位置
255                                                 p_mutex->OwnerOriginalPrio);
256                           break;
257
258                      default:                                           //如果状态超出预期
259                          OS_CRITICAL_EXIT();
260                         *p_err = OS_ERR_STATE_INVALID;                  //错误类型为“任务状态非法”
261                          return ((OS_OBJ_QTY)0);                        //返回0（有错误），停止执行
262                  }
263              }
```

```
264
265              ts = OS_TS_GET();                                          //获取时间戳
266              while (cnt > 0u) {                                         //移除该互斥信号量等待列表
267                  p_pend_data = p_pend_list->HeadPtr;                    //中的所有任务。
268                  p_tcb       = p_pend_data->TCBPtr;
269                  OS_PendObjDel((OS_PEND_OBJ *)((void *)p_mutex),
270                                p_tcb,
271                                ts);
272                  cnt--;
273              }
274 #if OS_CFG_DBG_EN > 0u                                  //如果使能了调试代码和变量
275              OS_MutexDbgListRemove(p_mutex);            //将信号量从互斥信号量调试列表移除
276 #endif
277              OSMutexQty--;                              //互斥信号量数目减1
278              OS_MutexClr(p_mutex);                      //清除信号量内容
279              OS_CRITICAL_EXIT_NO_SCHED();               //退出临界段，但不调度
280              OSSched();                                 //调度最高优先级任务运行
281             *p_err = OS_ERR_NONE;                       //错误类型为“无错误”
282              break;                                     //跳出
283
```

```
284         default:                                            //如果选项超出预期
285              OS_CRITICAL_EXIT();                            //退出临界段
286             *p_err = OS_ERR_OPT_INVALID;                    //错误类型为“选项非法”
287              break;                                         //跳出
288     }
289     return (nbr_tasks);                                     //返回删除前信号量等待列表中的任务数
290 }
291 #endif
```

图 7-13 OSMutexDel() 函数

OSMutexDel() 函数会调用一个更加底层的删除等待对象的函数来执行对互斥信号量的删除，该函数就是 OS_PendObjDel ()。OS_PendObjDel () 函数不仅仅用来删除互斥信号量，还可以删除多值信号量、消息队列、事件标志组、任务消息队列或任务信号量。OS_PendObjDel () 函数的定义位于“os_core.c”。

os_cfg_app.c | os_sem.c | os_core.c | os_cfg.h

```
1693 void  OS_PendObjDel (OS_PEND_OBJ  *p_obj,   //被删除对象的类型
1694                     OS_TCB       *p_tcb,   //任务控制块指针
1695                     CPU_TS        ts)      //信号量被删除时的时间戳
1696 {
1697     switch (p_tcb->TaskState) {                                 //根据任务状态分类处理
1698         case OS_TASK_STATE_RDY:                                 //如果任务是就绪状态
1699         case OS_TASK_STATE_DLY:                                 //如果任务是延时状态
1700         case OS_TASK_STATE_SUSPENDED:                           //如果任务是挂起状态
1701         case OS_TASK_STATE_DLY_SUSPENDED:                       //如果任务是在延时中被挂起
1702              break;                                             //这些情况均与等待无关，直接跳出
1703
1704         case OS_TASK_STATE_PEND:                               //如果任务是无期限等待状态
1705         case OS_TASK_STATE_PEND_TIMEOUT:                       //如果任务是有期限等待状态
1706              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
1707                  OS_PendObjDel1(p_obj,                         //强制解除任务对某一对象的等待
1708                                 p_tcb,
1709                                 ts);
1710              }
1711 #if (OS_MSG_EN > 0u)                                    //如果使能了任务队列或消息队列
1712              p_tcb->MsgPtr     = (void *)0;             //清除（复位）任务的消息域
1713              p_tcb->MsgSize    = (OS_MSG_SIZE)0u;
1714 #endif
1715              p_tcb->TS         = ts;                           //保存等待被中止时的时间戳到任务控制块
1716              OS_PendListRemove(p_tcb);                         //将任务从所有等待列表中移除
1717              OS_TaskRdy(p_tcb);                                //让任务进准备运行
1718              p_tcb->TaskState  = OS_TASK_STATE_RDY;            //修改任务状态为就绪状态
1719              p_tcb->PendStatus = OS_STATUS_PEND_DEL;           //标记任务的等待对象被删除
1720              p_tcb->PendOn     = OS_TASK_PEND_ON_NOTHING;      //标记任务目前没有等待任何对象
1721              break;                                            //跳出
1722
1723         case OS_TASK_STATE_PEND_SUSPENDED:                     //如果任务在无期限等待中被挂起
1724         case OS_TASK_STATE_PEND_TIMEOUT_SUSPENDED:             //如果任务在有期限等待中被挂起
1725              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
1726                  OS_PendObjDel1(p_obj,                         //强制解除任务对某一对象的等待
1727                                 p_tcb,
1728                                 ts);
1729              }
1730 #if (OS_MSG_EN > 0u)                                    //如果使能了任务队列或消息队列
1731              p_tcb->MsgPtr     = (void      *)0;       //清除（复位）任务的消息域
1732              p_tcb->MsgSize    = (OS_MSG_SIZE)0u;
1733 #endif
1734              p_tcb->TS         = ts;                           //保存等待被中止时的时间戳到任务控制块
1735              OS_TickListRemove(p_tcb);                         //让任务脱离节拍列表
1736              OS_PendListRemove(p_tcb);                         //将任务从所有等待列表中移除
1737              p_tcb->TaskState  = OS_TASK_STATE_SUSPENDED;      //修改任务状态为挂起状态
1738              p_tcb->PendStatus = OS_STATUS_PEND_DEL;           //标记任务的等待对象被删除
1739              p_tcb->PendOn     = OS_TASK_PEND_ON_NOTHING;      //标记任务目前没有等待任何对象
1740              break;                                            //跳出
1741
1742         default:                                               //如果任务状态超出预期
1743              break;                                            //不需处理，直接跳出
1744     }
1745 }
```

图 7-14 OS_PendObjDel () 函数

7.2 实例演示

7.2.1 实例 1

在进行本节实例之前，我们来想象这样一个问题。例如 AD 采样，采样结果是 16 位，我们要把它的高八位和低八位分别写在两个八位的变量里供上位机读取。可是当程序刚写完高八位，还没写低八位时，上位机就发来指令，触发接收中断要返回 AD 采样结果。这个时候倘若返回的 AD 采样结果肯定是不对的，这是一个很严重的问你。如果能让该 AD 采样结果的读写动作在时间上不重叠，这个问题也就可以迎刃而解。

本节实例就是解决上述问题的一个写照。本节实例创建读写两个任务，AppTaskWrite()和 AppTaskRead()。在任务 AppTaskWrite()中对全局数组 ucValue 的两个元素（初始值均是 0）自加，两者之间间隔 100ms。在任务 AppTaskRead()读取这两个元素，如果相等，则表示这两个元素得到了同时更新，获取数据正确，打印“SUCCESSFUL”；如果不想打，则表示这两个元素没有得到同时更新，数据传送存在漏洞，打印“FAIL”。

该例程已经存放在配套资料的下图路径。

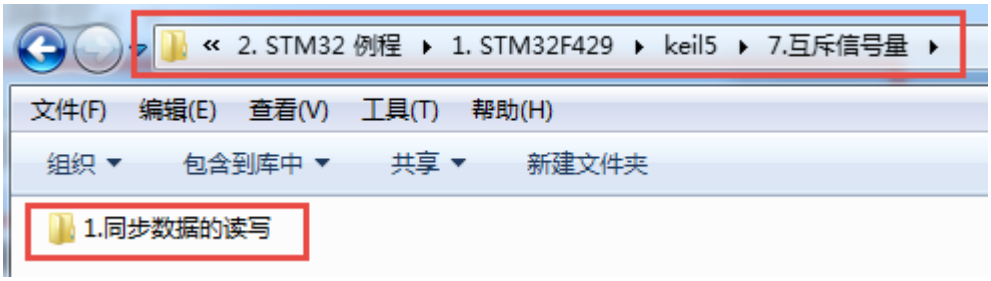

图 7-15 例程路径

本例程需用使用 USART1，所以工程中需要添加其驱动文件和初始化。用户可参照前面相关实例，这里不再赘述。

在起始任务 AppTaskStart() 中，在创建应用任务之前，创建了应用任务需要使用到的互斥信号量 mutex（必须保证该信号量在被使用到之前创建好）。

```
Build (F7)
Build target files
app.c  os_mutex.c
145 static  void  AppTaskStart (void *p_arg)
146 {
147     CPU_INT32U  cpu_clk_freq;
148     CPU_INT32U  cnts;
149     OS_ERR      err;
150
151
152     (void)p_arg;
153
154     BSP_Init();                                                 //板级初始化
155     CPU_Init();                                                 //初始化 CPU
156
157     cpu_clk_freq = BSP_CPU_ClkFreq();                           //获取 CPU 内
158     cnts = cpu_clk_freq / (CPU_INT32U)OSCfg_TickRate_Hz;        //根据用户设定
159     OS_CPU_SysTickInit(cnts);                                   //调用 SysTick
160
161     Mem_Init();                                                 //初始化内存管
162
163 #if OS_CFG_STAT_TASK_EN > 0u                                    //如果使能（默
164     OSStatTaskCPUUsageInit(&err);                               //计算没有应用
165 #endif                                                          //容量（决定（
166                                                                 //使用率使用）
167     CPU_IntDisMeasMaxCurReset();                                //复位（清零）
168
169
170     /* 创建互斥信号量 mutex */
171     OSMutexCreate ((OS_MUTEX  *)&mutex,                 //指向信号量变量的指针
172                    (CPU_CHAR  *)"Mutex For Test",       //信号量的名字
173                    (OS_ERR    *)&err);                  //错误类型
174
175
176     /* 创建 AppTaskWrite 任务 */
```

图 7-16 创建互斥信号量

任务函数 AppTaskWrite() 的定义如下。

app_cfg.h | app.c | os_mutex.c

```
213 ************************************************************************************************
214 *                                            WRITE TASK
215 ************************************************************************************************
216 */
217 static  void  AppTaskWrite ( void * p_arg )
218 {
219   OS_ERR      err;
220
221
222   (void)p_arg;
223
224
225   while (DEF_TRUE) {                                          //任务体
226     OSMutexPend ((OS_MUTEX  *)&mutex,                         //申请互斥信号量 mutex
227                  (OS_TICK    )0,                              //无期限等待
228                  (OS_OPT     )OS_OPT_PEND_BLOCKING,           //如果不能申请到信号量就堵塞任务
229                  (CPU_TS     *)0,                             //不想获得时间戳
230                  (OS_ERR     *)&err);                         //返回错误类型
231
232     ucValue [ 0 ] ++;
233
234     OSTimeDly ( 100, OS_OPT_TIME_DLY, & err );               //延时100个时钟节拍（100ms）
235
236     ucValue [ 1 ] ++;
237
238     OSMutexPost ((OS_MUTEX  *)&mutex,                         //释放互斥信号量 mutex
239                  (OS_OPT     )OS_OPT_POST_NONE,               //进行任务调度
240                  (OS_ERR     *)&err);                         //返回错误类型
241
242   }
243
244 }
```

图 7-17 AppTaskWrite() 任务函数

任务函数 AppTaskRead() 的定义如下。

app_cfg.h | app.c | os_mutex.c

```
248 *********************************************************************************************************
249 *                                                READ TASK
250 *********************************************************************************************************
251 */
252 static  void  AppTaskRead ( void * p_arg )
253 {
254   OS_ERR      err;
255   CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必需该宏，该宏声明和
256                   //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
257                   // SR（临界段关中断只需保存SR），开中断时将该值还原。
258   (void)p_arg;
259
260
261   while (DEF_TRUE) {                                    //任务体
262     OSMutexPend ((OS_MUTEX  *)&mutex,                   //申请互斥信号量 mutex
263                  (OS_TICK    )0,                        //无期限等待
264                  (OS_OPT     )OS_OPT_PEND_BLOCKING,     //如果申请不到就堵塞任务
265                  (CPU_TS     *)0,                       //不想获得时间戳
266                  (OS_ERR     *)&err);                   //返回错误类型
267
268     if ( ucValue [ 0 ] == ucValue [ 1 ] )
269     {
270       OS_CRITICAL_ENTER();                              //进入临界段
271
272       printf ( "\r\nSUCCESSFUL\r\n" );
273
274       OS_CRITICAL_EXIT();
275
276     }
277     else
278     {
279       OS_CRITICAL_ENTER();                              //进入临界段
280
281       printf ( "\r\nFAIL\r\n" );
282
283       OS_CRITICAL_EXIT();
284
285     }
286
287     OSMutexPost ((OS_MUTEX  *)&mutex,                   //释放互斥信号量 mutex
288                  (OS_OPT     )OS_OPT_POST_NONE,         //进行任务调度
289                  (OS_ERR     *)&err);                   //返回错误类型
290
291     OSTimeDly ( 1000, OS_OPT_TIME_DLY, & err );          //延时1000个时钟节拍（1s）
292
293   }
294
295 }
```

图 7-18 AppTaskRead() 任务函数

把 STM32 的 USART1 连接至电脑的串口调试助手，编译和下载程序到秉火 STM32 开发板，运行程序。用户可以看到，串口调试助手上打印的都是“SUCCESSFUL”，这说明程序达到了目的，数组 ucValue 的两个元素均是在被更新完后才被一起读取。

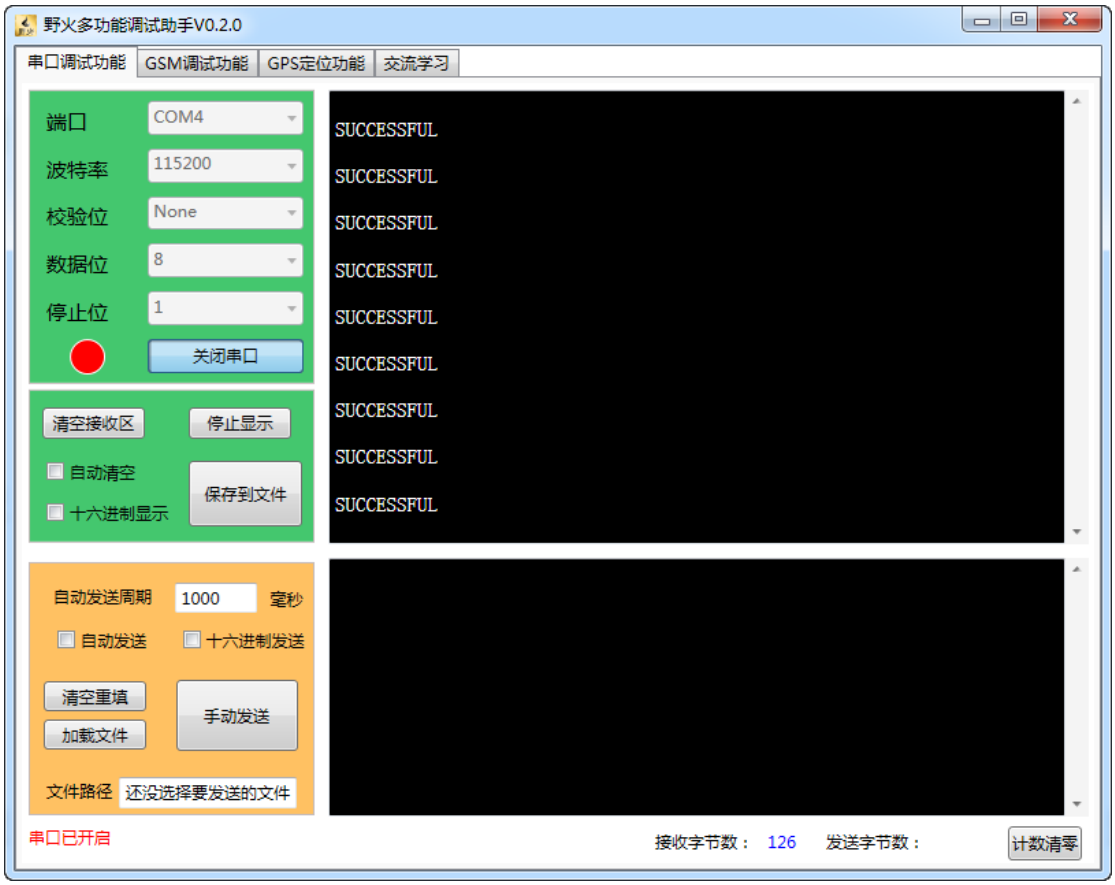

图 7-19 串口调试助手

为了验证互斥信号量 mutex 在这里的作用，用户可以注释掉 mutex 的作用代码，如下图所示。

app_cfg.h app.c os_mutex.c

```
213 ************************************************************************************************************
214 *                                                WRITE TASK
215 ************************************************************************************************************
216 */
217 static  void  AppTaskWrite ( void * p_arg )
218 {
219   OS_ERR      err;
220 
221 
222   (void)p_arg;
223 
224 
225   while (DEF TRUE) {                                          //任务体
226 //    OSMutexPend ((OS_MUTEX  *)&mutex,                       //申请互斥信号量 mutex
227 //                 (OS_TICK    )0,                            //无期限等待
228 //                 (OS_OPT     )OS_OPT_PEND_BLOCKING,         //如果不能申请到信号量就堵塞任务
229 //                 (CPU_TS     *)0,                           //不想获得时间戳
230 //                 (OS_ERR     *)&err);                       //返回错误类型
231 
232     ucValue [ 0 ] ++;
233 
234     OSTimeDly ( 100, OS_OPT_TIME_DLY, & err );              //延时100个时钟节拍（100ms）
235 
236     ucValue [ 1 ] ++;
237 
238 //    OSMutexPost ((OS_MUTEX  *)&mutex,                       //释放互斥信号量 mutex
239 //                 (OS_OPT     )OS_OPT_POST_NONE,             //进行任务调度
240 //                 (OS_ERR     *)&err);                       //返回错误类型
241 
242   }
243 
244 }
```

app_cfg.h | app.c | os_mutex.c

```
248 *********************************************************************************************
249 *                                          READ TASK
250 *********************************************************************************************
251 */
252 static  void  AppTaskRead ( void * p_arg )
253 {
254   OS_ERR      err;
255   CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必需该宏，该宏声明和
256                   //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
257                   // SR（临界段关中断只需保存SR），开中断时将该值还原。
258   (void)p_arg;
259
260
261   while (DEF_TRUE) {                                   //任务体
262 //    OSMutexPend ((OS_MUTEX  *)&mutex,                  //申请互斥信号量 mutex
263 //                 (OS_TICK    )0,                       //无期限等待
264 //                 (OS_OPT     )OS_OPT_PEND_BLOCKING,    //如果申请不到就堵塞任务
265 //                 (CPU_TS     *)0,                      //不想获得时间戳
266 //                 (OS_ERR     *)&err);                  //返回错误类型
267
268     if ( ucValue [ 0 ] == ucValue [ 1 ] )
269     {
270       OS_CRITICAL_ENTER();                             //进入临界段
271
272       printf ( "\r\nSUCCESSFUL\r\n" );
273
274       OS_CRITICAL_EXIT();
275
276     }
277     else
278     {
279       OS_CRITICAL_ENTER();                             //进入临界段
280
281       printf ( "\r\nFAIL\r\n" );
282
283       OS_CRITICAL_EXIT();
284
285     }
286
287 //    OSMutexPost ((OS_MUTEX  *)&mutex,                  //释放互斥信号量 mutex
288 //                 (OS_OPT     )OS_OPT_POST_NONE,        //进行任务调度
289 //                 (OS_ERR     *)&err);                  //返回错误类型
290
291     OSTimeDly ( 1000, OS_OPT_TIME_DLY, & err );          //延时1000个时钟节拍（1s）
292
293   }
294
295 }
```

图 7-20 注释代码

重新编译和下载程序到秉火 STM32 开发板，运行程序。用户可以看到，串口调试助手上打印内容变成了“FAIL”，这说明程序失去互斥信号量 mutex 的作用后，没法达到目的，数组 ucValue 的两个元素还没更新完就被一起读取了。

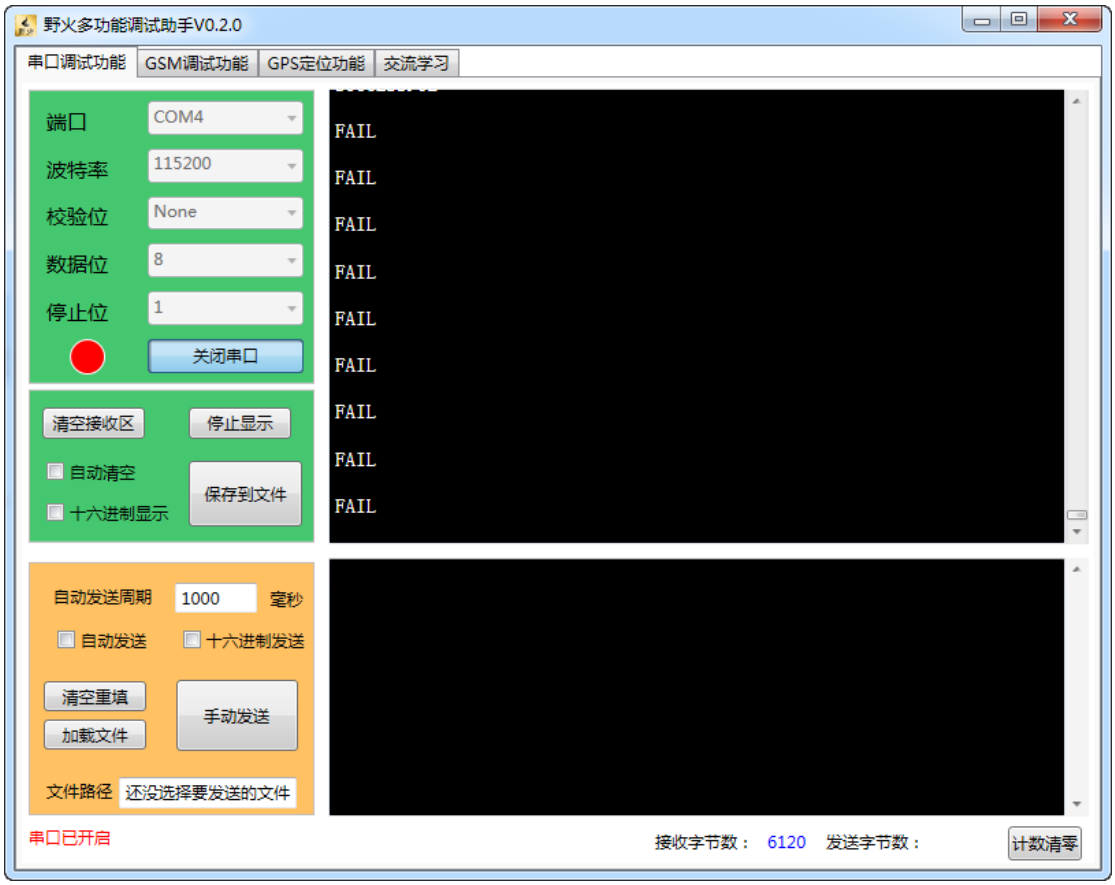

图 7-21 串口调试助手

7.3 章末总结

互斥信号量与多值信号量非常相似。多值信号量具有计数成员，可以管理资源计数。互斥信号量只有两个值，就是 0 和 1，所以互斥信号量也被称为二值信号量。互斥信号量主要用来同步数据信息。

使用互斥信号量之前必须先创建它，创建互斥信号量使用 OSMutexCreate () 函数。

OSMutexPend() 函数用于申请互斥信号量，如果互斥信号量不可用，可以选择等待或者不等待。与之相对应，OSMutexPost () 函数则用于释放互斥信号量。拥有互斥信号量的任务可以继续使用 OSMutexPend() 函数对其进行嵌套，最多可以嵌套 250 层。嵌套层数越大，表示该任务对信号量的拥有程度越高。任务对互斥信号量嵌套多少层，也值有调用多少次 OSMutexPost () 函数才能释放该信号量。

OSMutexPendAbort() 函数用于中止任务对一个互斥信号量的等待。OSSemDel () 函数用于删除一个互斥信号量。

第8章 消息队列

前面所说的多值信号量和和互斥信号量主要用来标志事件是否发生和协调资源的访问。如果要给资源赋予内容进行传递，信号量就力有所不及了。这时候就需要用到 uC/OS 操作系统的另一个内核机制了，那就是消息队列。

8.1 原理简述

如果想要使用消息队列，就必须事先使能消息队列。消息队列的使能位于“os_cfg.h”。

```
68                                                  /* ------------------------- MESSAGE QUEUES -
69  #define OS_CFG_Q_EN                         1u  //使能/禁用消息队列
70  #define OS_CFG_Q_DEL_EN                     1u  //使能或禁用 OSQDel() 函数
71  #define OS_CFG_Q_FLUSH_EN                   1u  //使能或禁用 OSQFlush() 函数
72  #define OS_CFG_Q_PEND_ABORT_EN              1u  //使能或禁用 OSQPendAbort() 函数
73
```

图 8-1

消息队列的消息都是从消息池获取的。消息队列可以有一个或多个，但消息队列却只有一个。当有消息队列要发布消息时，就从消息池（OS_MSG 类型数组的元素依次组成的单向列表）获取一个元素用于存放消息，并将该消息插入到消息队列。当有消息队列释放（等到）消息时，又会把这个元素释放回消息池，该元素又可以继续供发布消息使用。

如果用户使能了消息队列，在调用 OSInit() 函数对系统进行初始化时，就会初始化消息池。

```
147 #if (OS_MSG_EN) > 0u                    /* Initialize the free list of OS_MSGs */
148     OS_MsgPoolInit(p_err);
149     if (*p_err != OS_ERR_NONE) {
150         return;
151     }
152 #endif
153
```

图 8-2 初始化消息池

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”。

```
61 void  OS_MsgPoolInit (OS_ERR  *p_err)    //返回错误类型
62 {
63     OS_MSG      *p_msg1;
64     OS_MSG      *p_msg2;
65     OS_MSG_QTY   i;
66     OS_MSG_QTY   loops;
67
68
69
70 #ifdef OS_SAFETY_CRITICAL                 //如果使能（默认禁用）了安全检测
71     if (p_err == (OS_ERR *)0) {           //如果错误类型实参为空
72         OS_SAFETY_CRITICAL_EXCEPTION(); //执行安全检测异常函数
73         return;                           //返回，停止执行
74     }
75 #endif
```

```
76
77 #if OS_CFG_ARG_CHK_EN > 0u                    //如果使能了参数检测
78     if (OSCfg_MsgPoolBasePtr == (OS_MSG *)0) {//如果消息池不存在
79        *p_err = OS_ERR_MSG_POOL_NULL_PTR;     //错误类型为“消息池指针为空”
80         return;                               //返回，停止执行
81     }
82     if (OSCfg_MsgPoolSize == (OS_MSG_QTY)0) { //如果消息池不能存放消息
83        *p_err = OS_ERR_MSG_POOL_EMPTY;        //错误类型为“消息池为空”
84         return;                               //返回，停止执行
85     }
86 #endif
```

```
87     /* 将消息池里的消息逐条串成单向链表，方便管理 */
88     p_msg1 = OSCfg_MsgPoolBasePtr;
89     p_msg2 = OSCfg_MsgPoolBasePtr;
90     p_msg2++;
91     loops  = OSCfg_MsgPoolSize - 1u;
92     for (i = 0u; i < loops; i++) {             //初始化每一条消息
93         p_msg1->NextPtr = p_msg2;
94         p_msg1->MsgPtr  = (void      *)0;
95         p_msg1->MsgSize = (OS_MSG_SIZE)0u;
96         p_msg1->MsgTS   = (CPU_TS     )0u;
97         p_msg1++;
98         p_msg2++;
99     }
100    p_msg1->NextPtr = (OS_MSG    *)0;          //最后一条消息
101    p_msg1->MsgPtr  = (void      *)0;
102    p_msg1->MsgSize = (OS_MSG_SIZE)0u;
103    p_msg1->MsgTS   = (CPU_TS     )0u;
104    /* 初始化消息池数据 */
105    OSMsgPool.NextPtr     =  OSCfg_MsgPoolBasePtr;
106    OSMsgPool.NbrFree     =  OSCfg_MsgPoolSize;
107    OSMsgPool.NbrUsed     = (OS_MSG_QTY)0;
108    OSMsgPool.NbrUsedMax  = (OS_MSG_QTY)0;
109   *p_err                 =  OS_ERR_NONE;      //错误类型为“无错误”
110 }
```

图 8-3 OS_MsgPoolInit() 函数

8.1.1 OSQCreate ()

要使用 uC/OS 的消息队列必须先声明和创建消息队列，调用 OSQCreate () 函数可以创建一个消息队列。OSQCreate () 函数的信息如下表所示。

表 24 OSQCreate ()

函数原型	void OSQCreate (OS_Q *p_q, CPU_CHAR *p_name, OS_MSG_QTY max_qty, OS_ERR *p_err);			
功能	创建一个消息队列。			
参数	p_q	消息队列指针。		
	p_name	消息队列名称。		
	max_qty	最大消息数目。		
	p_err	返回错误类型	OS_ERR_NONE	无错误
			OS_ERR_CREATE_ISR	在中断中调用该函数
			OS_ERR_ILLEGAL_CREATE_RUN_TIME	在调用 OSSafetyCriticalStart() 函数后创建内核对象
			OS_ERR_NAME	p_name 为空指针
			OS_ERR_OBJ_CREATED	该消息队列已经被创建过
			OS_ERR_OBJ_PTR_NULL	p_q 是个空指针
			OS_ERR_Q_SIZE	max_qty 为 0.
返回值	无。			
注意事项	✧ 创建前必须先为 p_q 声明一个消息队列对象（OS_Q）。 ✧ 不可以在中断中调用该函数。			

OSQCreate () 函数的定义位于“os_q.c”。

```
71  void  OSQCreate (OS_Q         *p_q,      //消息队列指针
72                   CPU_CHAR     *p_name,   //消息队列名称
73                   OS_MSG_QTY    max_qty,  //消息队列大小（不能为0）
74                   OS_ERR       *p_err)    //返回错误类型
75
76  {
77      CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必需该宏，该宏声明和
78                      //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
79                      // SR（临界段关中断只需保存SR），开中断时将该值还原。
80
81  #ifdef OS_SAFETY_CRITICAL                  //如果使能了安全检测
82      if (p_err == (OS_ERR *)0) {            //如果错误类型实参为空
83          OS_SAFETY_CRITICAL_EXCEPTION(); //执行安全检测异常函数
84          return;                            //返回，停止执行
85      }
86  #endif

87
88  #ifdef OS_SAFETY_CRITICAL_IEC61508         //如果使能了安全关键
89      if (OSSafetyCriticalStartFlag == DEF_TRUE) { //如果在调用OSSafetyCriticalStart()后创建
90          *p_err = OS_ERR_ILLEGAL_CREATE_RUN_TIME;  //错误类型为“非法创建内核对象”
91          return;                            //返回，停止执行
92      }
93  #endif

94
95  #if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u     //如果使能了中断中非法调用检测
96      if (OSIntNestingCtr > (OS_NESTING_CTR)0) { //如果该函数是在中断中被调用
97          *p_err = OS_ERR_CREATE_ISR;        //错误类型为“在中断中创建对象”
98          return;                            //返回，停止执行
99      }
100 #endif
101
102 #if OS_CFG_ARG_CHK_EN > 0u                 //如果使能了参数检测
103     if (p_q == (OS_Q *)0) {                //如果 p_q 为空
104         *p_err = OS_ERR_OBJ_PTR_NULL;      //错误类型为“创建对象为空”
105         return;                            //返回，停止执行
106     }
107     if (max_qty == (OS_MSG_QTY)0) {        //如果 max_qty = 0
108         *p_err = OS_ERR_Q_SIZE;            //错误类型为“队列空间为0”
109         return;                            //返回，停止执行
110     }
111 #endif

112
113     OS_CRITICAL_ENTER();                   //进入临界段
114     p_q->Type    = OS_OBJ_TYPE_Q;          //标记创建对象数据结构为消息队列
115     p_q->NamePtr = p_name;                 //标记消息队列的名称
116     OS_MsgQInit(&p_q->MsgQ,                //初始化消息队列
117                 max_qty);
118     OS_PendListInit(&p_q->PendList);       //初始化该消息队列的等待列表
119
120 #if OS_CFG_DBG_EN > 0u                     //如果使能了调试代码和变量
121     OS_QDbgListAdd(p_q);                   //将该队列添加到消息队列双向调试链表
122 #endif
123     OSQQty++;                              //消息队列个数加1
124
125     OS_CRITICAL_EXIT_NO_SCHED();           //退出临界段（无调度）
126    *p_err = OS_ERR_NONE;                   //错误类型为“无错误”
127 }
```

图 8-4 OSQCreate () 函数

其中，OSQCreate () 函数调用了 OS_MsgQInit () 函数初始化了消息队列。OS_MsgQInit() 函数的定义位于“os_msg.c”。

```
168 void  OS_MsgQInit (OS_MSG_Q    *p_msg_q, //消息队列指针
169                   OS_MSG_QTY   size)    //消息队列空间
170 {
171     p_msg_q->NbrEntriesSize = (OS_MSG_QTY)size; //消息队列可存放消息数目
172     p_msg_q->NbrEntries     = (OS_MSG_QTY)0;    //消息队列目前可用消息数
173     p_msg_q->NbrEntriesMax  = (OS_MSG_QTY)0;    //可用消息数的最大历史记录
174     p_msg_q->InPtr          = (OS_MSG    *)0;   //队列的入队指针
175     p_msg_q->OutPtr         = (OS_MSG    *)0;   //队列的出队指针
176 }
```

图 8-5 OS_MsgQInit () 函数

另外，OSQCreate () 函数还调用了 OS_PendListInit() 函数初始化了消息队列的等待列表。每个消息队列都有一个等待列表，凡是等待该消息队列的的任务都会被插入到这个等待列表，方便高效管理。OS_PendListInit() 函数的定义位于“os_core.c”。

```
1319 void  OS_PendListInit (OS_PEND_LIST  *p_pend_list)
1320 {
1321     p_pend_list->HeadPtr    = (OS_PEND_DATA *)0;   //复位等待列表的所有成员
1322     p_pend_list->TailPtr    = (OS_PEND_DATA *)0;
1323     p_pend_list->NbrEntries = (OS_OBJ_QTY    )0;
1324 }
```

图 8-6 OS_PendListInit() 函数

如果使能了 OS_CFG_DBG_EN（位于“os_cfg.h”），创建消息队列时还会调用 OS_QDbgListAdd() 函数将该消息队列插入到一个消息队列调试列表，是为方便调试所设。OS_QDbgListAdd () 函数的定义位于“os_q.c”。

```
802 #if OS_CFG_DBG_EN > 0u              //如果使能（默认使能）了调试代码和变量
803 void  OS_QDbgListAdd (OS_Q  *p_q)  //将消息队列插入到消息队列列表的最前端
804 {
805     p_q->DbgNamePtr               = (CPU_CHAR *)((void *)" ");//先不指向任何任务的名称
806     p_q->DbgPrevPtr               = (OS_Q     *)0;            //将该队列作为列表的最前端
807     if (OSQDbgListPtr == (OS_Q *)0) {                         //如果列表为空
808         p_q->DbgNextPtr           = (OS_Q     *)0;            //该队列的下一个元素也为空
809     } else {                                                  //如果列表非空
810         p_q->DbgNextPtr           =  OSQDbgListPtr;           //列表原来的首元素作为该队列的下一个元素
811         OSQDbgListPtr->DbgPrevPtr =  p_q;                     //原来的首元素的前面变为了该队列
812     }
813     OSQDbgListPtr                 =  p_q;                     //该队列成为列表的新首元素
814 }
```

图 8-7 OS_QDbgListAdd() 函数

8.1.2 OSQPost ()

OSQPost () 函数用于向消息队列发布一个消息。OSQPost () 函数的信息如下表所示。

表 25 OSQPost ()

函数原型	void OSQPost (OS_Q *p_q, void *p_void, OS_MSG_SIZE msg_size, OS_OPT opt, OS_ERR *p_err);

功能	向消息队列发布一个消息。			
参数	p_q	消息队列指针。		
	p_void,	消息指针。		
	msg_size	消息大小（单位：字节）。		
	opt	选项	OS_OPT_POST_FIFO	把消息发布到队列的入口端，并且只唤醒一个等待任务。
			OS_OPT_POST_LIFO	把消息发布到队列的出口端，并且只唤醒一个等待任务。
			OS_OPT_POST_FIFO | OS_OPT_POST_ALL	把消息发布到队列的入口端，并且唤醒全部等待任务。
			OS_OPT_POST_LIFO | OS_OPT_POST_ALL	把消息发布到队列的出口端，并且唤醒全部等待任务。
			OS_OPT_POST_FIFO | OS_OPT_POST_NO_SCHED	把消息发布到队列的入口端；只唤醒一个等待任务；不进行任务调度，继续运行当前任务。
			OS_OPT_POST_LIFO | OS_OPT_POST_NO_SCHED	把消息发布到队列的出口端；只唤醒一个等待任务；不进行任务调度，继续运行当前任务。
			OS_OPT_POST_FIFO | OS_OPT_POST_ALL | OS_OPT_POST_NO_SCHED	把消息发布到队列的入口端；唤醒全部等待任务；不进行任务调度，继续运行当前任务。
			OS_OPT_POST_LIFO | OS_OPT_POST_ALL | OS_OPT_POST_NO_SCHED	把消息发布到队列的出口端；唤醒全部等待任务；不进行任务调度，继续运行当前任务。
	p_err	返回错误类型	OS_ERR_NONE	调用成功，消息被发布了。
			OS_ERR_MSG_POOL_EMPTY	消息池没可用消息。
			OS_ERR_OBJ_PTR_NULL	p_q 为空。
			OS_ERR_OBJ_TYPE	p_q 不是消息队列类型。
			OS_ERR_Q_MAX	消息队列已满。
返回值	无。			

OSQPost () 函数的定义也位于“os_q.c

```
688 void  OSQPost (OS_Q          *p_q,        //消息队列指针
689                 void         *p_void,     //消息指针
690                 OS_MSG_SIZE   msg_size,   //消息大小（单位：字节）
691                 OS_OPT        opt,        //选项
692                 OS_ERR       *p_err)      //返回错误类型
693 {
694     CPU_TS  ts;
695
696
697
698 #ifdef OS_SAFETY_CRITICAL                 //如果使能（默认禁用）了安全检测
699     if (p_err == (OS_ERR *)0) {           //如果错误类型实参为空
700         OS_SAFETY_CRITICAL_EXCEPTION();   //执行安全检测异常函数
701         return;                           //返回，停止执行
702     }
703 #endif
704

705 #if OS_CFG_ARG_CHK_EN > 0u                //如果使能了参数检测
706     if (p_q == (OS_Q *)0) {               //如果 p_q 为空
707        *p_err = OS_ERR_OBJ_PTR_NULL;      //错误类型为“内核对象为空”
708         return;                           //返回，停止执行
709     }
710     switch (opt) {                              //根据选项分类处理
711         case OS_OPT_POST_FIFO:                  //如果选项在预期内
712         case OS_OPT_POST_LIFO:
713         case OS_OPT_POST_FIFO | OS_OPT_POST_ALL:
714         case OS_OPT_POST_LIFO | OS_OPT_POST_ALL:
715         case OS_OPT_POST_FIFO | OS_OPT_POST_NO_SCHED:
716         case OS_OPT_POST_LIFO | OS_OPT_POST_NO_SCHED:
717         case OS_OPT_POST_FIFO | OS_OPT_POST_ALL | OS_OPT_POST_NO_SCHED:
718         case OS_OPT_POST_LIFO | OS_OPT_POST_ALL | OS_OPT_POST_NO_SCHED:
719              break;                             //直接跳出
720
721         default:                                //如果选项超出预期
722             *p_err =  OS_ERR_OPT_INVALID;       //错误类型为“选项非法”
723              return;                            //返回，停止执行
724     }
725 #endif
726

727 #if OS_CFG_OBJ_TYPE_CHK_EN > 0u           //如果使能了对象类型检测
728     if (p_q->Type != OS_OBJ_TYPE_Q) {     //如果 p_q 不是消息队列类型
729        *p_err = OS_ERR_OBJ_TYPE;          //错误类型为“对象类型错误”
730         return;                           //返回，停止执行
731     }
732 #endif
733
734     ts = OS_TS_GET();                     //获取时间戳
735
736 #if OS_CFG_ISR_POST_DEFERRED_EN > 0u              //如果使能了中断延迟发布
737     if (OSIntNestingCtr > (OS_NESTING_CTR)0) {    //如果该函数在中断中被调用
738         OS_IntQPost((OS_OBJ_TYPE)OS_OBJ_TYPE_Q,   //将该消息发布到中断消息队列
739                     (void      *)p_q,
740                     (void      *)p_void,
741                     (OS_MSG_SIZE)msg_size,
742                     (OS_FLAGS   )0,
743                     (OS_OPT     )opt,
744                     (CPU_TS     )ts,
745                     (OS_ERR    *)p_err);
746         return;                                   //返回（尚未发布），停止执行
747     }
748 #endif
```

```
749
750     OS_QPost(p_q,                                   //将消息按照普通方式
751              p_void,
752              msg_size,
753              opt,
754              ts,
755              p_err);
756 }
```

图 8-8 OSQPost () 函数

其实，不管是否使能了中断延迟发布，最终都是调用 OS_QPost () 函数进行发布消息。只是使能了中断延迟发布的发布过程会比较曲折，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。OS_QPost () 函数的定义位于“os_q.c”。

```
922 void  OS_QPost (OS_Q          *p_q,       //消息队列指针
923                 void          *p_void,    //消息指针
924                 OS_MSG_SIZE    msg_size,  //消息大小（单位：字节）
925                 OS_OPT         opt,       //选项
926                 CPU_TS         ts,        //消息被发布时的时间戳
927                 OS_ERR        *p_err)     //返回错误类型
928 {
929     OS_OBJ_QTY      cnt;
930     OS_OPT          post_type;
931     OS_PEND_LIST   *p_pend_list;
932     OS_PEND_DATA   *p_pend_data;
933     OS_PEND_DATA   *p_pend_data_next;
934     OS_TCB         *p_tcb;
935     CPU_SR_ALLOC();   //使用到临界段（在关/开中断时）时必需该宏，该宏声明和
936                       //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
937                       // SR（临界段关中断只需保存SR），开中断时将该值还原。
938
```

```
939     OS_CRITICAL_ENTER();                                //进入临界段
940     p_pend_list = &p_q->PendList;                       //取出该队列的等待列表
941     if (p_pend_list->NbrEntries == (OS_OBJ_QTY)0) {     //如果没有任务在等待该队列
942         if ((opt & OS_OPT_POST_LIFO) == (OS_OPT)0) {    //把消息发布到队列的末端
943             post_type = OS_OPT_POST_FIFO;
944         } else {                                        //把消息发布到队列的前端
945             post_type = OS_OPT_POST_LIFO;
946         }
947         OS_MsgQPut(&p_q->MsgQ,                          //把消息放入消息队列
948                    p_void,
949                    msg_size,
950                    post_type,
951                    ts,
952                    p_err);
953         OS_CRITICAL_EXIT();                            //退出临界段
954         return;                                        //返回，执行完毕
955     }
```

```
956     /* 如果有任务在等待该队列 */
957     if ((opt & OS_OPT_POST_ALL) != (OS_OPT)0) {      //如果要把消息发布给所有等待任务
958         cnt = p_pend_list->NbrEntries;               //获取等待任务数目
959     } else {                                         //如果要把消息发布给一个等待任务
960         cnt = (OS_OBJ_QTY)1;                         //要处理的任务数目为1
961     }
962     p_pend_data = p_pend_list->HeadPtr;              //获取等待列表的头部（任务）
963     while (cnt > 0u) {                               //根据要发布的任务数目逐个发布
964         p_tcb           = p_pend_data->TCBPtr;
965         p_pend_data_next = p_pend_data->NextPtr;
966         OS_Post((OS_PEND_OBJ *)((void *)p_q),        //把消息发布给任务
967                 p_tcb,
968                 p_void,
969                 msg_size,
970                 ts);
971         p_pend_data = p_pend_data_next;
972         cnt--;
973     }
974     OS_CRITICAL_EXIT_NO_SCHED();                     //退出临界段（无调度）
975     if ((opt & OS_OPT_POST_NO_SCHED) == (OS_OPT)0) {//如果没选择“发布完不调度任务”
976         OSSched();                                   //任务调度
977     }
978    *p_err = OS_ERR_NONE;                             //错误类型为“无错误”
979 }
```

图 8-9 OS_QPost () 函数

在 OS_QPost () 函数中，会调用 OS_MsgQPut () 函数从消息池获取一个消息插入到消息队列。OS_MsgQPut () 函数的定义位于“os_msg.c”。

```
287 void  OS_MsgQPut (OS_MSG_Q      *p_msg_q,     //消息队列指针
288                   void          *p_void,      //消息指针
289                   OS_MSG_SIZE    msg_size,    //消息大小（单位：字节）
290                   OS_OPT         opt,         //选项
291                   CPU_TS         ts,          //消息被发布时的时间戳
292                   OS_ERR        *p_err)       //返回错误类型
293 {
294     OS_MSG  *p_msg;
295     OS_MSG  *p_msg_in;
296
297
298
299 #ifdef OS_SAFETY_CRITICAL                      //如果使能了安全检测
300     if (p_err == (OS_ERR *)0) {                //如果错误类型实参为空
301         OS_SAFETY_CRITICAL_EXCEPTION();        //执行安全检测异常函数
302         return;                                //返回，停止执行
303     }
304 #endif
```

```
305
306     if (p_msg_q->NbrEntries >= p_msg_q->NbrEntriesSize) { //如果消息队列已没有可用空间
307        *p_err = OS_ERR_Q_MAX;                             //错误类型为“队列已满”
308         return;                                           //返回，停止执行
309     }
310
311     if (OSMsgPool.NbrFree == (OS_MSG_QTY)0) {  //如果消息池没有可用消息
312        *p_err = OS_ERR_MSG_POOL_EMPTY;         //错误类型为“消息池没有消息”
313         return;                                //返回，停止执行
314     }
315     /* 从消息池获取一个消息（暂存于 p_msg ）*/
316     p_msg             = OSMsgPool.NextPtr;          //将消息控制块从消息池移除
317     OSMsgPool.NextPtr = p_msg->NextPtr;             //指向下一个消息（取走首个消息）
318     OSMsgPool.NbrFree--;                            //消息池可用消息数减1
319     OSMsgPool.NbrUsed++;                            //消息池被用消息数加1
320     if (OSMsgPool.NbrUsedMax < OSMsgPool.NbrUsed) { //更新消息被用最大数目的历史记录
321         OSMsgPool.NbrUsedMax = OSMsgPool.NbrUsed;
322     }
```

```
323     /* 将获取的消息插入到消息队列 */
324     if (p_msg_q->NbrEntries == (OS_MSG_QTY)0) {             //如果消息队列目前没有消息
325         p_msg_q->InPtr         = p_msg;                     //将其入队指针指向该消息
326         p_msg_q->OutPtr        = p_msg;                     //出队指针也指向该消息
327         p_msg_q->NbrEntries    = (OS_MSG_QTY)1;             //队列的消息数为1
328         p_msg->NextPtr         = (OS_MSG *)0;               //该消息的下一个消息为空
329     } else {                                                //如果消息队列目前已有消息
330         if ((opt & OS_OPT_POST_LIFO) == OS_OPT_POST_FIFO) { //如果用FIFO方式插入队列，
331             p_msg_in           = p_msg_q->InPtr;            //将消息插入到入队端，入队
332             p_msg_in->NextPtr  = p_msg;                     //指针指向该消息。
333             p_msg_q->InPtr     = p_msg;
334             p_msg->NextPtr     = (OS_MSG *)0;
335         } else {                                            //如果用LIFO方式插入队列，
336             p_msg->NextPtr     = p_msg_q->OutPtr;           //将消息插入到出队端，出队
337             p_msg_q->OutPtr    = p_msg;                     //指针指向该消息。
338         }
339         p_msg_q->NbrEntries++;                              //消息队列的消息数目加1
340     }
341     if (p_msg_q->NbrEntriesMax < p_msg_q->NbrEntries) {     //更新改消息队列的最大消息
342         p_msg_q->NbrEntriesMax = p_msg_q->NbrEntries;       //数目的历史记录。
343     }
344     p_msg->MsgPtr  = p_void;                                //给该消息填写消息内容
345     p_msg->MsgSize = msg_size;                              //给该消息填写消息大小
346     p_msg->MsgTS   = ts;                                    //填写发布该消息时的时间戳
347    *p_err          = OS_ERR_NONE;                           //错误类型为“无错误”
348 }
```

图 8-10 OS_MsgQPut () 函数

另外，OS_QPost () 函数还调用了 OS_Post() 函数发布内核对象。OS_Post() 函数是一个底层的发布函数，它不仅仅用来发布消息队列，还可以发布多值信号量、互斥信号量、事件标志组、任务消息队列和任务信号量。OS_Post() 函数的定义位于“os_core.c”。

os_sem.c | os.h | os_core.c | os_cfg.h

```
1844 void  OS_Post (OS_PEND_OBJ  *p_obj,      //内核对象类型指针
1845              OS_TCB       *p_tcb,      //任务控制块
1846              void         *p_void,     //消息
1847              OS_MSG_SIZE   msg_size,   //消息大小
1848              CPU_TS        ts)         //时间戳
1849 {
1850     switch (p_tcb->TaskState) {                          //根据任务状态分类处理
1851         case OS_TASK_STATE_RDY:                          //如果任务处于就绪状态
1852         case OS_TASK_STATE_DLY:                          //如果任务处于延时状态
1853         case OS_TASK_STATE_SUSPENDED:                    //如果任务处于挂起状态
1854         case OS_TASK_STATE_DLY_SUSPENDED:                //如果任务处于延时中被挂起状态
1855              break;                                      //不用处理，直接跳出
```

```
1856
1857          case OS_TASK_STATE_PEND:                                    //如果任务处于无期限等待状态
1858          case OS_TASK_STATE_PEND_TIMEOUT:                            //如果任务处于有期限等待状态
1859               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
1860                    OS_Post1(p_obj,                                   //标记哪个内核对象被发布
1861                             p_tcb,
1862                             p_void,
1863                             msg_size,
1864                             ts);
1865               } else {                                               //如果任务不是在等待多个信号量或消息队列
1866#if (OS_MSG_EN > 0u)                                                  //如果使能了任务队列或消息队列
1867                    p_tcb->MsgPtr  = p_void;                          //保存消息到等待任务
1868                    p_tcb->MsgSize = msg_size;
1869#endif
1870                    p_tcb->TS      = ts;                              //保存时间戳到等待任务
1871               }
1872               if (p_obj != (OS_PEND_OBJ *)0) {                       //如果内核对象为空
1873                    OS_PendListRemove(p_tcb);                         //从等待列表移除该等待任务
1874#if OS_CFG_DBG_EN > 0u                                                //如果使能了调试代码和变量
1875                    OS_PendDbgNameRemove(p_obj,                       //移除内核对象的调试名
1876                                         p_tcb);
1877#endif
1878               }
1879               OS_TaskRdy(p_tcb);                                     //让该等待任务准备运行
1880               p_tcb->TaskState  = OS_TASK_STATE_RDY;                 //任务状态改为就绪状态
1881               p_tcb->PendStatus = OS_STATUS_PEND_OK;                 //清除等待状态
1882               p_tcb->PendOn     = OS_TASK_PEND_ON_NOTHING;           //标记不再等待
1883               break;
```

```
1884
1885          case OS_TASK_STATE_PEND_SUSPENDED:                          //如果任务在无期限等待中被挂起
1886          case OS_TASK_STATE_PEND_TIMEOUT_SUSPENDED:                  //如果任务在有期限等待中被挂起
1887               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
1888                    OS_Post1(p_obj,                                   //标记哪个内核对象被发布
1889                             p_tcb,
1890                             p_void,
1891                             msg_size,
1892                             ts);
1893               } else {                                               //如果任务不在等待多个信号量或消息队列
1894#if (OS_MSG_EN > 0u)                                                  //如果使能了调试代码和变量
1895                    p_tcb->MsgPtr  = p_void;                          //保存消息到等待任务
1896                    p_tcb->MsgSize = msg_size;
1897#endif
1898                    p_tcb->TS      = ts;                              //保存时间戳到等待任务
1899               }
1900               OS_TickListRemove(p_tcb);                              //从节拍列表移除该等待任务
1901               if (p_obj != (OS_PEND_OBJ *)0) {                       //如果内核对象为空
1902                    OS_PendListRemove(p_tcb);                         //从等待列表移除该等待任务
1903#if OS_CFG_DBG_EN > 0u                                                //如果使能了调试代码和变量
1904                    OS_PendDbgNameRemove(p_obj,                       //移除内核对象的调试名
1905                                         p_tcb);
1906#endif
1907               }
1908               p_tcb->TaskState  = OS_TASK_STATE_SUSPENDED;           //任务状态改为被挂起状态
1909               p_tcb->PendStatus = OS_STATUS_PEND_OK;                 //清除等待状态
1910               p_tcb->PendOn     = OS_TASK_PEND_ON_NOTHING;           //标记不再等待
1911               break;
1912
1913          default:                                                    //如果任务状态超出预期
1914               break;                                                //直接跳出
1915     }
1916 }
```

图 8-11 OS_Post() 函数

8.1.3 OSQPend ()

与 OSQPost () 消息队列发布消息函数相对应，OSQPend () 函数用于等待获取消息队列的消息。

表 26 OSQPend ()

函数原	void *OSQPend (OS_Q	*p_q,
	OS_TICK	timeout,
	OS_OPT	opt,

<table>
<tr><td>型</td><td colspan="4">OS_MSG_SIZE *p_msg_size,
CPU_TS *p_ts,
OS_ERR *p_err);</td></tr>
<tr><td>功能</td><td colspan="4">等待一个消息队列的消息。</td></tr>
<tr><td rowspan="11">参数</td><td>p_q</td><td colspan="3">消息队列指针。</td></tr>
<tr><td>timeout</td><td colspan="3">等待超时时间（单位：时钟节拍），0 代表无期限等待。opt 为 OS_OPT_PEND_BLOCKING 时该参数才起作用。</td></tr>
<tr><td rowspan="2">opt</td><td rowspan="2">选项</td><td>OS_OPT_PEND_BLOCKING</td><td>如果不能立即获得消息，就堵塞当前任务，继续等待消息。</td></tr>
<tr><td>OS_OPT_PEND_NON_BLOCKING</td><td>如果不能立即获得消息，不堵塞当前任务，不继续等待消息。</td></tr>
<tr><td>p_msg_size</td><td colspan="3">消息大小（单位：字节）。</td></tr>
<tr><td>p_ts</td><td>时间戳</td><td colspan="2">用于存储消息队列最后一次被发布消息的时间戳，或者等待被中止的时间戳，或者消息队列被删除时的时间戳，具体返回哪个时间戳，要根据返回的 p_err 判断。该参数可以为 NULL，表示用户不需要获得时间戳。</td></tr>
<tr><td rowspan="10">p_err</td><td rowspan="10">返回错误类型</td><td>OS_ERR_NONE</td><td>没错误，任务获得消息。</td></tr>
<tr><td>OS_ERR_OBJ_PTR_NULL</td><td>p_q 为空。</td></tr>
<tr><td>OS_ERR_PTR_INVALID</td><td>p_msg_size 为空。</td></tr>
<tr><td>OS_ERR_OBJ_TYPE</td><td>p_q 不是消息队列类型对象。</td></tr>
<tr><td>OS_ERR_PEND_ABORT</td><td>等待被中止。</td></tr>
<tr><td>OS_ERR_PEND_ISR</td><td>在中断中被调用。</td></tr>
<tr><td>OS_ERR_PEND_WOULD_BLOCK</td><td>缺乏堵塞。</td></tr>
<tr><td>OS_ERR_SCHED_LOCKED</td><td>调度器被锁。</td></tr>
<tr><td>OS_ERR_STATUS_INVALID</td><td>等待状态非法。</td></tr>
<tr><td>OS_ERR_TIMEOUT</td><td>等待超时。</td></tr>
<tr><td>返回值</td><td colspan="4">✧ != (void *)0，接收到的消息的指针（首地址）。
✧ == (void *)0，接收到一个空消息，或者没接收到消息，或者所等待的消息队列不存在，或者用户传给 p_q 的不是消息队列类型的对象。</td></tr>
<tr><td>注意事项</td><td colspan="4">✧ 不可以在中断中调用该函数。</td></tr>
</table>

OSQPend () 函数的定义也位于“os_q.c”。

```
384 void  *OSQPend (OS_Q         *p_q,         //消息队列指针
385                 OS_TICK       timeout,     //等待期限（单位：时钟节拍）
386                 OS_OPT        opt,         //选项
387                 OS_MSG_SIZE  *p_msg_size,//返回消息大小（单位：字节）
388                 CPU_TS       *p_ts,        //获取等到消息时的时间戳
389                 OS_ERR       *p_err)       //返回错误类型
390 {
391     OS_PEND_DATA  pend_data;
392     void         *p_void;
393     CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必需该宏，该宏声明和
394                     //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
395                     // SR（临界段关中断只需保存SR），开中断时将该值还原。
396
397 #ifdef OS_SAFETY_CRITICAL                 //如果使能（默认禁用）了安全检测
398     if (p_err == (OS_ERR *)0) {           //如果错误类型实参为空
399         OS_SAFETY_CRITICAL_EXCEPTION(); //执行安全检测异常函数
400         return ((void *)0);               //返回0（有错误），停止执行
401     }
402 #endif
403
```

```
404 #if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u          //如果使能了中断中非法调用检测
405     if (OSIntNestingCtr > (OS_NESTING_CTR)0) { //如果该函数在中断中被调用
406        *p_err = OS_ERR_PEND_ISR;                //错误类型为“在中断中中止等待”
407         return ((void *)0);                     //返回0（有错误），停止执行
408     }
409 #endif
410
411 #if OS_CFG_ARG_CHK_EN > 0u                  //如果使能了参数检测
412     if (p_q == (OS_Q *)0) {                 //如果 p_q 为空
413        *p_err = OS_ERR_OBJ_PTR_NULL;        //错误类型为“对象为空”
414         return ((void *)0);                 //返回0（有错误），停止执行
415     }
416     if (p_msg_size == (OS_MSG_SIZE *)0) { //如果 p_msg_size 为空
417        *p_err = OS_ERR_PTR_INVALID;         //错误类型为“指针不可用”
418         return ((void *)0);                 //返回0（有错误），停止执行
419     }
420     switch (opt) {                          //根据选项分类处理
421         case OS_OPT_PEND_BLOCKING:          //如果选项在预期内
422         case OS_OPT_PEND_NON_BLOCKING:
423              break;                         //直接跳出
424
425         default:                            //如果选项超出预期
426             *p_err = OS_ERR_OPT_INVALID;   //返回错误类型为“选项非法”
427              return ((void *)0);            //返回0（有错误），停止执行
428     }
429 #endif
```

```
430
431 #if OS_CFG_OBJ_TYPE_CHK_EN > 0u          //如果使能了对象类型检测
432     if (p_q->Type != OS_OBJ_TYPE_Q) {  //如果 p_q 不是消息队列类型
433        *p_err = OS_ERR_OBJ_TYPE;         //错误类型为“对象类型有误”
434         return ((void *)0);              //返回0（有错误），停止执行
435     }
436 #endif
437
```

```
    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) {  //如果选择了不堵塞任务
        CPU_CRITICAL_EXIT();                              //开中断
       *p_err = OS_ERR_PEND_WOULD_BLOCK;                  //错误类型为“等待渴求堵塞”
        return ((void *)0);                               //返回0（有错误），停止执行
    } else {                                              //如果选择了堵塞任务
        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”
            return ((void *)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(&pend_data,                                   //堵塞当前任务，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,                            //插入到节拍列表和等待列表。
            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:                           //如果等待状态正常
             p_void     = OSTCBCurPtr->MsgPtr;            //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:                        //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
```

```
493         case OS_STATUS_PEND_TIMEOUT:                        //如果等待超时
494              p_void     = (void      *)0;                    //返回消息内容为空
495             *p_msg_size = (OS_MSG_SIZE)0;                    //返回消息大小为0
496              if (p_ts  != (CPU_TS *)0) {                     //如果 p_ts 非空
497                 *p_ts   = (CPU_TS  )0;                       //清零 p_ts
498              }
499             *p_err      = OS_ERR_TIMEOUT;                    //错误类型为“等待超时”
500              break;                                          //跳出
501
502         case OS_STATUS_PEND_DEL:                            //如果等待的内核对象被删除
503              p_void     = (void      *)0;                    //返回消息内容为空
504             *p_msg_size = (OS_MSG_SIZE)0;                    //返回消息大小为0
505              if (p_ts  != (CPU_TS *)0) {                     //如果 p_ts 非空
506                 *p_ts   =  OSTCBCurPtr->TS;                  //获取对象被删时的时间戳
507              }
508             *p_err      = OS_ERR_OBJ_DEL;                    //错误类型为“等待对象被删”
509              break;                                          //跳出
510
511         default:                                            //如果等待状态超出预期
512              p_void     = (void      *)0;                    //返回消息内容为空
513             *p_msg_size = (OS_MSG_SIZE)0;                    //返回消息大小为0
514             *p_err      = OS_ERR_STATUS_INVALID;             //错误类型为“状态非法”
515              break;                                          //跳出
516     }
517     CPU_CRITICAL_EXIT();                                    //开中断
518     return (p_void);                                        //返回消息内容
519 }
```

图 8-12 OSQPend () 函数

在 OSQPend () 函数中，会调用 OS_MsgQGet () 函数从消息队列获取一个消息。OS_MsgQGet () 函数的定义位于“os_msg.c”。

```
203 void  *OS_MsgQGet (OS_MSG_Q      *p_msg_q,      //消息队列
204                    OS_MSG_SIZE  *p_msg_size,   //返回消息大小
205                    CPU_TS       *p_ts,         //返回某些操作的时间戳
206                    OS_ERR       *p_err)        //返回错误类型
207 {
208     OS_MSG  *p_msg;
209     void    *p_void;
210
211
212
213 #ifdef OS_SAFETY_CRITICAL                      //如果使能（默认禁用）了安全检测
214     if (p_err == (OS_ERR *)0) {                //如果错误类型实参为空
215         OS_SAFETY_CRITICAL_EXCEPTION();        //执行安全检测异常函数
216         return ((void *)0);                    //返回空消息，停止执行
217     }
218 #endif
219
```

```
220     if (p_msg_q->NbrEntries == (OS_MSG_QTY)0) {  //如果消息队列没有消息
221        *p_msg_size = (OS_MSG_SIZE)0;             //返回消息长度为0
222         if (p_ts != (CPU_TS *)0) {               //如果 p_ts 非空
223            *p_ts  = (CPU_TS  )0;                 //清零 p_ts
224         }
225        *p_err = OS_ERR_Q_EMPTY;                  //错误类型为“队列没消息”
226         return ((void *)0);                      //返回空消息，停止执行
227     }
```

```
228     /* 如果消息队列有消息 */
229     p_msg            = p_msg_q->OutPtr;          //从队列的出口端提取消息
230     p_void           = p_msg->MsgPtr;            //提取消息内容
231    *p_msg_size       = p_msg->MsgSize;           //提取消息长度
232     if (p_ts != (CPU_TS *)0) {                   //如果 p_ts 非空
233        *p_ts  = p_msg->MsgTS;                    //获取消息被发布时的时间戳
234     }
235
236     p_msg_q->OutPtr = p_msg->NextPtr;            //修改队列的出队指针
237
238     if (p_msg_q->OutPtr == (OS_MSG *)0) {        //如果队列没有消息了
239         p_msg_q->InPtr       = (OS_MSG    *)0;   //清零出队指针
240         p_msg_q->NbrEntries = (OS_MSG_QTY)0;     //清零消息数
241     } else {                                     //如果队列还有消息
242         p_msg_q->NbrEntries--;                   //队列的消息数减1
243     }
244     /* 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 */
245     p_msg->NextPtr     = OSMsgPool.NextPtr;      //消息插回消息池
246     OSMsgPool.NextPtr = p_msg;
247     OSMsgPool.NbrFree++;                         //消息池的可用消息数加1
248     OSMsgPool.NbrUsed--;                         //消息池的已用消息数减1
249
250    *p_err             = OS_ERR_NONE;            //错误类型为“无错误”
251     return (p_void);                            //返回罅隙内容
252 }
```

图 8-13 OS_MsgQGet () 函数

OSQPend () 函数会调用一个更加底层的等待函数来执行当前任务对消息队列的等待，该函数就是 OS_Pend()。与 OS_Post() 函数一样，OS_Pend() 函数不仅仅用来等待消息队列，还可以等待多值信号量、互斥信号量、事件标志组、任务消息队列和任务信号量。OS_Pend() 函数的定义位于“os_core.c”。

```
os_sem.c  os.h  os_core.c  os_cfg.h  os_type.h
873 void  OS_Pend (OS_PEND_DATA  *p_pend_data,   //待插入等待列表的元素
874               OS_PEND_OBJ   *p_obj,         //等待的内核对象
875               OS_STATE       pending_on,    //等待哪种对象内核
876               OS_TICK        timeout)       //等待期限
877 {
878     OS_PEND_LIST  *p_pend_list;
879
880
881
882     OSTCBCurPtr->PendOn     = pending_on;                    //资源不可用，开始等待
883     OSTCBCurPtr->PendStatus = OS_STATUS_PEND_OK;             //正常等待中
884
885     OS_TaskBlock(OSTCBCurPtr,                                //阻塞当前运行任务，
886                  timeout);                                   //如果 timeout 非0，把任务插入的节拍列表
887
888     if (p_obj != (OS_PEND_OBJ *)0) {                         //如果等待对象非空
889         p_pend_list             = &p_obj->PendList;          //获取对象的等待列表到 p_pend_list
890         p_pend_data->PendObjPtr = p_obj;                     //保存要等待的对象
891         OS_PendDataInit((OS_TCB       *)OSTCBCurPtr,         //初始化 p_pend_data（待插入等待列表）
892                         (OS_PEND_DATA *)p_pend_data,
893                         (OS_OBJ_QTY    )1);
894         OS_PendListInsertPrio(p_pend_list,                   //按优先级将 p_pend_data 插入到等待列表
895                               p_pend_data);
896     } else {                                                 //如果等待对象为空
897         OSTCBCurPtr->PendDataTblEntries = (OS_OBJ_QTY    )0; //清零当前任务的等待域数据
898         OSTCBCurPtr->PendDataTblPtr     = (OS_PEND_DATA *)0;
899     }
900 #if OS_CFG_DBG_EN > 0u                                       //如果使能了调试代码和变量
901     OS_PendDbgNameAdd(p_obj,                                 //更新信号量的 DbgNamePtr 元素为其等待
902                       OSTCBCurPtr);                          //列表中优先级最高的任务的名称。
903 #endif
904 }
```

图 8-14 OS_Pend() 函数

8.1.4 OSQPendAbort ()

OSQPendAbort () 函数用于中止任务对一个消息队列的等待。要使用 OSQPendAbort () 函数，还得事先使能 OS_CFG_Q_PEND_ABORT_EN（位于“os_cfg.h”），如下图所示。

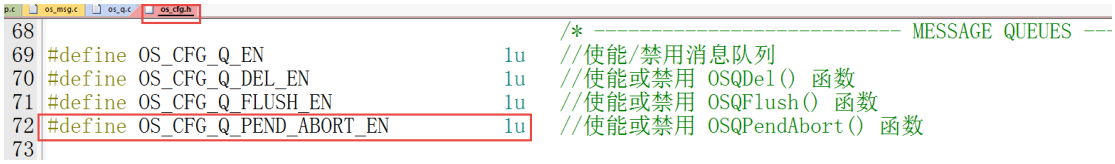

图 8-15

OSQPendAbort () 函数的信息如下表所示。

表 27 OSQPendAbort ()

函数原型	OS_OBJ_QTY OSQPendAbort (OS_Q *p_q, OS_OPT opt, OS_ERR *p_err);			
功能	中止任务对一个消息队列的等待。			
参数	p_q	消息队列指针。		
	opt	选项。	OS_OPT_PEND_ABORT_1	只中止该消息队列等待列表中的最高优先级任务。
			OS_OPT_PEND_ABORT_ALL	中止该消息队列等待列表中的所有优先级任务。
			OS_OPT_PEND_ABORT_1 | OS_OPT_POST_NO_SCHED	只中止该消息队列等待列表中的最高优先级任务，但不进行任务调度。
			OS_OPT_PEND_ABORT_ALL | OS_OPT_POST_NO_SCHED	中止该消息队列等待列表中的所有优先级任务，但不进行任务调度。
	p_err	返回错误类型	OS_ERR_NONE	无错误。
			OS_ERR_OPT_INVALID	选项非法。
			OS_ERR_OBJ_PTR_NULL	p_q 为空。
			OS_ERR_OBJ_TYPE	p_q 不是消息队列类型。
			OS_ERR_PEND_ABORT_ISR	该函数在中断中被调用。
			OS_ERR_PEND_ABORT_NONE	没有任务在等待该消息队列。
返回值	✧ 0，没有任务在等待该信号量，或者有错误产生。 ✧ >0，被中止的任务数。			
注意事项	✧ 不可以在中断中调用该函数。			

OSQPendAbort () 函数的定义位于“os_q.c”。

```
554 #if OS_CFG_Q_PEND_ABORT_EN > 0u                  //如果使能了 OSQPendAbort() 函数
555 OS_OBJ_QTY  OSQPendAbort (OS_Q     *p_q,        //消息队列
556                           OS_OPT    opt,        //选项
557                           OS_ERR   *p_err)      //返回错误类型
558 {
559     OS_PEND_LIST  *p_pend_list;
560     OS_TCB        *p_tcb;
561     CPU_TS         ts;
562     OS_OBJ_QTY     nbr_tasks;
563     CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必需该宏，该宏声明和
564                     //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
565                     // SR（临界段关中断只需保存SR），开中断时将该值还原。
566
567 #ifdef OS_SAFETY_CRITICAL                       //如果使能（默认禁用）了安全检测
568     if (p_err == (OS_ERR *)0) {                 //如果错误类型实参为空
569         OS_SAFETY_CRITICAL_EXCEPTION();         //执行安全检测异常函数
570         return ((OS_OBJ_QTY)0u);                //返回0（有错误），停止执行
571     }
572 #endif
573
574 #if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u          //如果使能了中断中非法调用检测
575     if (OSIntNestingCtr > (OS_NESTING_CTR)0u) {  //如果该函数在中断中被调用
576        *p_err =  OS_ERR_PEND_ABORT_ISR;         //错误类型为“在中断中中止等待”
577         return ((OS_OBJ_QTY)0u);                //返回0（有错误），停止执行
578     }
579 #endif
580
581 #if OS_CFG_ARG_CHK_EN > 0u                                  //返回0（有错误），停止执行
582     if (p_q == (OS_Q *)0) {                                 //如果 p_q 为空
583        *p_err =  OS_ERR_OBJ_PTR_NULL;                       //错误类型为“对象为空”
584         return ((OS_OBJ_QTY)0u);                            //返回0（有错误），停止执行
585     }
586     switch (opt) {                                          //根据选项分类处理
587         case OS_OPT_PEND_ABORT_1:                           //如果选项在预期之内
588         case OS_OPT_PEND_ABORT_ALL:
589         case OS_OPT_PEND_ABORT_1   | OS_OPT_POST_NO_SCHED:
590         case OS_OPT_PEND_ABORT_ALL | OS_OPT_POST_NO_SCHED:
591              break;                                         //直接跳出
592
593         default:                                            //如果选项超出预期
594             *p_err =  OS_ERR_OPT_INVALID;                   //错误类型为“选项非法”
595              return ((OS_OBJ_QTY)0u);                       //返回0（有错误），停止执行
596     }
597 #endif
598
599 #if OS_CFG_OBJ_TYPE_CHK_EN > 0u          //如果使能了对象类型检测
600     if (p_q->Type != OS_OBJ_TYPE_Q) {    //如果 p_q 不是消息队列类型
601        *p_err =  OS_ERR_OBJ_TYPE;        //错误类型为“对象类型有误”
602         return ((OS_OBJ_QTY)0u);         //返回0（有错误），停止执行
603     }
604 #endif
605
606     CPU_CRITICAL_ENTER();                                   //关中断
607     p_pend_list = &p_q->PendList;                           //获取消息队列的等待列表
608     if (p_pend_list->NbrEntries == (OS_OBJ_QTY)0u) {        //如果没有任务在等待
609         CPU_CRITICAL_EXIT();                                //开中断
610        *p_err =  OS_ERR_PEND_ABORT_NONE;                    //错误类型为“没任务在等待”
611         return ((OS_OBJ_QTY)0u);
612     }
```

```
613     /* 如果有任务在等待 */
614     OS_CRITICAL_ENTER_CPU_EXIT();                          //锁调度器，重开中断
615     nbr_tasks = 0u;                                        //准备计数被中止的等待任务
616     ts        = OS_TS_GET();                               //获取时间戳
617     while (p_pend_list->NbrEntries > (OS_OBJ_QTY)0u) {     //如果还有任务在等待
618         p_tcb = p_pend_list->HeadPtr->TCBPtr;              //获取头端（最高优先级）任务
619         OS_PendAbort((OS_PEND_OBJ *)((void *)p_q),         //中止该任务对 p_q 的等待
620                      p_tcb,
621                      ts);
622         nbr_tasks++;
623         if (opt != OS_OPT_PEND_ABORT_ALL) {                //如果不是选择了中止所有等待任务
624             break;                                         //立即跳出，不再继续中止
625         }
626     }
627     OS_CRITICAL_EXIT_NO_SCHED();                           //减锁调度器，但不调度
628
629     if ((opt & OS_OPT_POST_NO_SCHED) == (OS_OPT)0u) {      //如果选择了任务调度
630         OSSched();                                         //进行任务调度
631     }
632
633    *p_err = OS_ERR_NONE;                                   //错误类型为“无错误”
634     return (nbr_tasks);                                    //返回被中止的任务数目
635 }
636 #endif
```

图 8-16 OSQPendAbort () 函数

OSQPendAbort () 函数会调用一个更加底层的中止等待函数来执行当前任务对消息队列的等待，该函数就是 OS_PendAbort()。OS_PendAbort() 函数不仅仅用来中止对消息队列的等待，还可以中止对多值信号量、互斥信号量、事件标志组、任务消息队列或任务信号量的等待。OS_PendAbort() 函数的定义位于“os_core.c”。

os_cfg.h | os_sem.c | os.h | os_core.c

```
927 void  OS_PendAbort (OS_PEND_OBJ  *p_obj,    //被等待对象的类型
928                     OS_TCB       *p_tcb,    //任务控制块指针
929                     CPU_TS        ts)       //等待被中止时的时间戳
930 {
931     switch (p_tcb->TaskState) {                            //根据任务状态分类处理
932         case OS_TASK_STATE_RDY:                            //如果任务是就绪状态
933         case OS_TASK_STATE_DLY:                            //如果任务是延时状态
934         case OS_TASK_STATE_SUSPENDED:                      //如果任务是挂起状态
935         case OS_TASK_STATE_DLY_SUSPENDED:                  //如果任务是在延时中被挂起
936              break;                                        //这些情况均与等待无关，直接跳出
937
938         case OS_TASK_STATE_PEND:                           //如果任务是无期限等待状态
939         case OS_TASK_STATE_PEND_TIMEOUT:                   //如果任务是有期限等待状态
940              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
941                  OS_PendAbort1(p_obj,                      //强制解除任务对某一对象的等待
942                                p_tcb,
943                                ts);
944              }
945 #if (OS_MSG_EN > 0u)                                       //如果使能了任务队列或消息队列
946              p_tcb->MsgPtr     = (void      *)0;           //清除（复位）任务的消息域
947              p_tcb->MsgSize    = (OS_MSG_SIZE)0u;
948 #endif
949              p_tcb->TS         = ts;                       //保存等待被中止时的时间戳到任务控制块
950              if (p_obj != (OS_PEND_OBJ *)0) {              //如果等待对象非空
951                  OS_PendListRemove(p_tcb);                 //将任务从所有等待列表中移除
952              }
953              OS_TaskRdy(p_tcb);                            //让任务进准备运行
954              p_tcb->TaskState  = OS_TASK_STATE_RDY;        //修改任务状态为就绪状态
955              p_tcb->PendStatus = OS_STATUS_PEND_ABORT;     //标记任务的等待被中止
956              p_tcb->PendOn     = OS_TASK_PEND_ON_NOTHING;  //标记任务目前没有等待任何对象
957              break;                                        //跳出
958
```

```
959         case OS_TASK_STATE_PEND_SUSPENDED:                  //如果任务在无期限等待中被挂起
960         case OS_TASK_STATE_PEND_TIMEOUT_SUSPENDED:          //如果任务在有期限等待中被挂起
961             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
962                 OS_PendAbort1(p_obj,                        //强制解除任务对某一对象的等待
963                               p_tcb,
964                               ts);
965             }
966 #if (OS_MSG_EN > 0u)                                        //如果使能了任务队列或消息队列
967             p_tcb->MsgPtr     = (void      *)0;             //清除（复位）任务的消息域
968             p_tcb->MsgSize    = (OS_MSG_SIZE)0u;
969 #endif
970             p_tcb->TS         = ts;                         //保存等待被中止时的时间戳到任务控制块
971             if (p_obj != (OS_PEND_OBJ *)0) {                //如果等待对象非空
972                 OS_PendListRemove(p_tcb);                   //将任务从所有等待列表中移除
973             }
974             OS_TickListRemove(p_tcb);                       //让任务脱离节拍列表
975             p_tcb->TaskState  = OS_TASK_STATE_SUSPENDED;    //修改任务状态为挂起状态
976             p_tcb->PendStatus = OS_STATUS_PEND_ABORT;       //标记任务的等待被中止
977             p_tcb->PendOn     = OS_TASK_PEND_ON_NOTHING;    //标记任务目前没有等待任何对象
978             break;                                          //跳出
979
980         default:                                            //如果任务状态超出预期
981             break;                                          //不需处理，直接跳出
982     }
983 }
```

图 8-17 OS_PendAbort() 函数

8.1.5 OSQDel ()

OSQDel() 函数用于删除一个消息队列。要使用 OSQDel () 函数，还得事先使能 OS_CFG_Q_DEL_EN（位于“os_cfg.h”），如下图所示。

```
68                                          /* ---------------------------- MESSAGE QUEUES ---
69 #define OS_CFG_Q_EN                  1u  //使能/禁用消息队列
70 #define OS_CFG_Q_DEL_EN              1u  //使能或禁用 OSQDel() 函数
71 #define OS_CFG_Q_FLUSH_EN            1u  //使能或禁用 OSQFlush() 函数
72 #define OS_CFG_Q_PEND_ABORT_EN       1u  //使能或禁用 OSQPendAbort() 函数
73
```

图 8-18

OSQDel () 函数的信息如下表所示。

表 28 OSQDel ()

函数原型	OS_OBJ_QTY OSQDel (OS_Q *p_q, OS_OPT opt, OS_ERR *p_err);			
功能	删除一个消息队列。			
参数	p_q	消息队列指针。		
	opt	选项	OS_OPT_DEL_NO_PEND	如果没有任务等待 p_q，才删除 p_q。
			OS_OPT_DEL_ALWAYS	必须删除 p_q。
	p_err	返回错误类型	OS_ERR_NONE	无错误。
			OS_ERR_DEL_ISR	该函数在中断中被调用。
			OS_ERR_OBJ_PTR_NULL	p_q 为空。
			OS_ERR_OBJ_TYPE	p_q 不是消息队列类型。
			OS_ERR_OPT_INVALID	选项非法。
			OS_ERR_TASK_WAITING	还有任务在等待该消息队列。

返回值	✧ 0，没有任务在等待该消息队列，或者有错误产生。 ✧ >0，消息队列被删除前等待其的任务数。
注意事项	✧ 不可以在中断中调用该函数。

OSQDel() 函数的定义位于“os_q.c”。

```
166 #if OS_CFG_Q_DEL_EN > 0u                    //如果使能了 OSQDel() 函数
167 OS_OBJ_QTY  OSQDel (OS_Q     *p_q,         //消息队列指针
168                     OS_OPT    opt,         //选项
169                     OS_ERR   *p_err)       //返回错误类型
170 {
171     OS_OBJ_QTY      cnt;
172     OS_OBJ_QTY      nbr_tasks;
173     OS_PEND_DATA   *p_pend_data;
174     OS_PEND_LIST   *p_pend_list;
175     OS_TCB         *p_tcb;
176     CPU_TS          ts;
177     CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必需该宏，该宏声明和
178                     //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
179                     // SR（临界段关中断只需保存SR），开中断时将该值还原。
180
181 #ifdef OS_SAFETY_CRITICAL                  //如果使能（默认禁用）了安全检测
182     if (p_err == (OS_ERR *)0) {            //如果错误类型实参为空
183         OS_SAFETY_CRITICAL_EXCEPTION(); //执行安全检测异常函数
184         return ((OS_OBJ_QTY)0);            //返回0（有错误），停止执行
185     }
186 #endif
187
188 #if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u          //如果使能了中断中非法调用检测
189     if (OSIntNestingCtr > (OS_NESTING_CTR)0) {  //如果该函数在中断中被调用
190         *p_err = OS_ERR_DEL_ISR;                //错误类型为“在中断中中止等待”
191         return ((OS_OBJ_QTY)0);                 //返回0（有错误），停止执行
192     }
193 #endif
194
195 #if OS_CFG_ARG_CHK_EN > 0u                  //如果使能了参数检测
196     if (p_q == (OS_Q *)0) {                 //如果 p_q 为空
197         *p_err =  OS_ERR_OBJ_PTR_NULL;      //错误类型为“对象为空”
198         return ((OS_OBJ_QTY)0u);            //返回0（有错误），停止执行
199     }
200     switch (opt) {                          //根据选项分类处理
201         case OS_OPT_DEL_NO_PEND:            //如果选项在预期内
202         case OS_OPT_DEL_ALWAYS:
203             break;                          //直接跳出
204
205         default:
206             *p_err =  OS_ERR_OPT_INVALID; //如果选项超出预期
207             return ((OS_OBJ_QTY)0u);      //返回0（有错误），停止执行
208     }
209 #endif
210

211 #if OS_CFG_OBJ_TYPE_CHK_EN > 0u         //如果使能了对象类型检测
212     if (p_q->Type != OS_OBJ_TYPE_Q) { //如果 p_q 不是消息队列类型
213         *p_err = OS_ERR_OBJ_TYPE;       //错误类型为“对象类型有误”
214         return ((OS_OBJ_QTY)0);         //返回0（有错误），停止执行
215     }
216 #endif
217
```

```
    CPU_CRITICAL_ENTER();                                       //关中断
    p_pend_list = &p_q->PendList;                               //获取消息队列的等待列表
    cnt         = p_pend_list->NbrEntries;                      //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) {                                              //根据选项分类处理
        case OS_OPT_DEL_NO_PEND:                                //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) {                  //如果没有任务在等待该队列
#if OS_CFG_DBG_EN > 0u                                          //如果使能了调试代码和变量
                 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);                                  //清除该队列的内容
                 CPU_CRITICAL_EXIT();                           //开中断
                *p_err = OS_ERR_NONE;                           //错误类型为“无错误”
             } else {                                           //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:                                 //如果必须删除信号量
             CPU_CRITICAL_ENTER_CPU_EXIT();                     //进入临界段，重开中断
             ts = OS_TS_GET();                                  //获取时间戳
             while (cnt > 0u) {                                 //逐个移除该队列等待列表中的任务
                 p_pend_data = p_pend_list->HeadPtr;
                 p_tcb       = p_pend_data->TCBPtr;
                 OS_PendObjDel((OS_PEND_OBJ *)((void *)p_q),
                               p_tcb,
                               ts);
                 cnt--;
             }
#if OS_CFG_DBG_EN > 0u                                          //如果使能了调试代码和变量
             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);                                      //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
}
#endif
```

图 8-19 OSQDel() 函数

OSQDel() 函数会调用一个更加底层的删除等待对象的函数来执行对消息队列的删除，该函数就是 OS_PendObjDel ()。OS_PendObjDel () 函数不仅仅用来删除消息队列，还可以删除多值信号量、互斥信号量、事件标志组、任务消息队列或任务信号量。OS_PendObjDel () 函数的定义位于“os_core.c”。

os_cfg_app.c | os_sem.c | os_core.c | os_cfg.h

```
void  OS_PendObjDel (OS_PEND_OBJ  *p_obj,   //被删除对象的类型
                     OS_TCB       *p_tcb,   //任务控制块指针
                     CPU_TS        ts)      //信号量被删除时的时间戳
{
    switch (p_tcb->TaskState) {                                 //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;                                             //这些情况均与等待无关，直接跳出
```

```
1704            case OS_TASK_STATE_PEND:                                    //如果任务是无期限等待状态
1705            case OS_TASK_STATE_PEND_TIMEOUT:                            //如果任务是有期限等待状态
1706                 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
1707                     OS_PendObjDel1(p_obj,                              //强制解除任务对某一对象的等待
1708                                    p_tcb,
1709                                    ts);
1710                 }
1711#if (OS_MSG_EN > 0u)                                   //如果使能了任务队列或消息队列
1712                 p_tcb->MsgPtr     = (void *)0;        //清除（复位）任务的消息域
1713                 p_tcb->MsgSize    = (OS_MSG_SIZE)0u;
1714#endif
1715                 p_tcb->TS         = ts;                                 //保存等待被中止时的时间戳到任务控制块
1716                 OS_PendListRemove(p_tcb);                               //将任务从所有等待列表中移除
1717                 OS_TaskRdy(p_tcb);                                      //让任务进准备运行
1718                 p_tcb->TaskState  = OS_TASK_STATE_RDY;                  //修改任务状态为就绪状态
1719                 p_tcb->PendStatus = OS_STATUS_PEND_DEL;                 //标记任务的等待对象被删除
1720                 p_tcb->PendOn     = OS_TASK_PEND_ON_NOTHING;            //标记任务目前没有等待任何对象
1721                 break;                                                  //跳出
1722
1723            case OS_TASK_STATE_PEND_SUSPENDED:                          //如果任务在无期限等待中被挂起
1724            case OS_TASK_STATE_PEND_TIMEOUT_SUSPENDED:                  //如果任务在有期限等待中被挂起
1725                 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
1726                     OS_PendObjDel1(p_obj,                              //强制解除任务对某一对象的等待
1727                                    p_tcb,
1728                                    ts);
1729                 }
1730#if (OS_MSG_EN > 0u)                                   //如果使能了任务队列或消息队列
1731                 p_tcb->MsgPtr     = (void      *)0;  //清除（复位）任务的消息域
1732                 p_tcb->MsgSize    = (OS_MSG_SIZE)0u;
1733#endif
1734                 p_tcb->TS         = ts;                                 //保存等待被中止时的时间戳到任务控制块
1735                 OS_TickListRemove(p_tcb);                               //让任务脱离节拍列表
1736                 OS_PendListRemove(p_tcb);                               //将任务从所有等待列表中移除
1737                 p_tcb->TaskState  = OS_TASK_STATE_SUSPENDED;            //修改任务状态为挂起状态
1738                 p_tcb->PendStatus = OS_STATUS_PEND_DEL;                 //标记任务的等待对象被删除
1739                 p_tcb->PendOn     = OS_TASK_PEND_ON_NOTHING;            //标记任务目前没有等待任何对象
1740                 break;                                                  //跳出
1741
1742            default:                                                     //如果任务状态超出预期
1743                 break;                                                  //不需处理，直接跳出
1744        }
1745}
```

图 8-20 OS_PendObjDel () 函数

8.1.6 OSQFlush ()

OSQFlush () 函数用于清空一个消息队列。要使用 OSQFlush () 函数，还得事先使能 OS_CFG_Q_FLUSH_EN（位于“os_cfg.h”），如下图所示。

```
68                                                  /* ---------------------------- MESSAGE QUEUES ---
69  #define OS_CFG_Q_EN                     1u      //使能/禁用消息队列
70  #define OS_CFG_Q_DEL_EN                 1u      //使能或禁用 OSQDel() 函数
71  #define OS_CFG_Q_FLUSH_EN               1u      //使能或禁用 OSQFlush() 函数
72  #define OS_CFG_Q_PEND_ABORT_EN          1u      //使能或禁用 OSQPendAbort() 函数
73
```

图 8-21

OSQFlush () 函数的信息如下表所示。

表 29 OSQFlush ()

函 数 原型	OS_MSG_QTY OSQFlush (OS_Q *p_q, OS_ERR *p_err);	
功能	清空一个消息队列（的消息）。	
参数	p_q	消息队列指针。

	p_err	返回错误类型	OS_ERR_NONE	无错误，执行成功。
			OS_ERR_FLUSH_ISR	该函数在中断中被调用。
			OS_ERR_OBJ_PTR_NULL	p_q 为空。
			OS_ERR_OBJ_TYPE	p_q 不是消息队列类型。
返回值	清空消息队列前队列里的消息数目。			
注意事项	✧ 不可以在中断中调用该函数			

OSQFlush () 函数的定义也位于“os_q.c

```
293 #if OS_CFG_Q_FLUSH_EN > 0u                //如果使能了 OSQFlush() 函数
294 OS_MSG_QTY  OSQFlush (OS_Q     *p_q,     //消息队列指针
295                       OS_ERR   *p_err)   //返回错误类型
296 {
297     OS_MSG_QTY  entries;
298     CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必需该宏，该宏声明和
299                     //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
300                     // SR（临界段关中断只需保存SR），开中断时将该值还原。
301
302 #ifdef OS_SAFETY_CRITICAL                  //如果使能（默认禁用）了安全检测
303     if (p_err == (OS_ERR *)0) {            //如果错误类型实参为空
304         OS_SAFETY_CRITICAL_EXCEPTION(); //执行安全检测异常函数
305         return ((OS_MSG_QTY)0);            //返回0（有错误），停止执行
306     }
307 #endif
308
309 #if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u          //如果使能了中断中非法调用检测
310     if (OSIntNestingCtr > (OS_NESTING_CTR)0) {  //如果该函数在中断中被调用
311        *p_err = OS_ERR_FLUSH_ISR;               //错误类型为“在中断中请空队列”
312         return ((OS_MSG_QTY)0);                 //返回0（有错误），停止执行
313     }
314 #endif
315
```

```
316 #if OS_CFG_ARG_CHK_EN > 0u                  //如果使能了参数检测
317     if (p_q == (OS_Q *)0) {                 //如果 p_q 为空
318         *p_err = OS_ERR_OBJ_PTR_NULL;       //错误类型为“对象为空”
319          return ((OS_MSG_QTY)0);            //返回0（有错误），停止执行
320     }
321 #endif
322
323 #if OS_CFG_OBJ_TYPE_CHK_EN > 0u             //如果使能了对象类型检测
324     if (p_q->Type != OS_OBJ_TYPE_Q) {       //如果 p_q 不是消息队列类型
325         *p_err = OS_ERR_OBJ_TYPE;           //错误类型为“对象类型有误”
326          return ((OS_MSG_QTY)0);            //返回0（有错误），停止执行
327     }
328 #endif
329
330     OS_CRITICAL_ENTER();                    //进入临界段
331     entries = OS_MsgQFreeAll(&p_q->MsgQ); //把队列的所有消息均释放回消息池
332     OS_CRITICAL_EXIT();                     //退出临界段
333    *p_err   = OS_ERR_NONE;                  //错误类型为“无错误”
334     return ((OS_MSG_QTY)entries);           //返回清空队列前队列的消息数目
335 }
336 #endif
```

图 8-22 OSQFlush () 函数

8.2 实例演示

8.2.1 实例 1

本节实例非常简单，就是用消息队列进行消息的发送和接收。本节实例创建读写两个任务，AppTaskPost ()和 AppTaskPend ()。任务 AppTaskPost () 用于发送消息。任务 AppTaskPend () 用于接收消息，并把接收到的消息通过串口调试助手打印出来。

该例程已经存放在配套资料的下图路径。

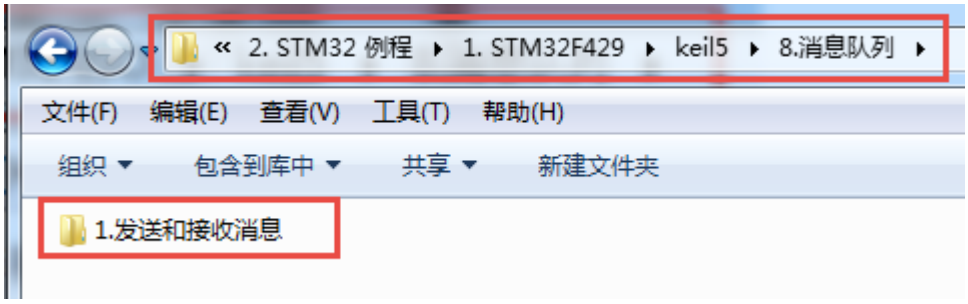

图 8-23 例程路径

本例程需用使用 USART1，所以工程中需要添加其驱动文件和初始化。用户可参照前面相关实例，这里不再赘述。

在起始任务 AppTaskStart() 中，在创建应用任务之前，创建了应用任务需要使用到的消息队列 queue（必须保证该消息队列在被使用到之前创建好）。

```
142 static  void  AppTaskStart (void *p_arg)
143 {
144     CPU_INT32U  cpu_clk_freq;
145     CPU_INT32U  cnts;
146     OS_ERR      err;
147
148
149     (void)p_arg;
150
151     BSP_Init();                                                 //板级初始化
152     CPU_Init();                                                 //初始化 CPU
153
154     cpu_clk_freq = BSP_CPU_ClkFreq();                           //获取 CPU
155     cnts = cpu_clk_freq / (CPU_INT32U)OSCfg_TickRate_Hz;        //根据用户设
156     OS_CPU_SysTickInit(cnts);                                   //调用 SysT
157
158     Mem_Init();                                                 //初始化内存
159
160 #if OS_CFG_STAT_TASK_EN > 0u                                    //如果使能（
161     OSStatTaskCPUUsageInit(&err);                               //计算没有应
162 #endif                                                          //容量（决定
163                                                                 //使用率使用
164     CPU_IntDisMeasMaxCurReset();                                //复位（清零
165
166
167     /* 创建消息队列 queue */
168     OSQCreate ((OS_Q         *)&queue,                  //指向消息队列的指针
169                (CPU_CHAR     *)"Queue For Test",        //队列的名字
170                (OS_MSG_QTY    )20,                      //最多可存放消息的数目
171                (OS_ERR       *)&err);                   //返回错误类型
172
173
174     /* 创建 AppTaskPost 任务 */
```

图 8-24 创建消息队列

任务函数 AppTaskPost () 的定义如下。任务每隔 500ms 向消息队列 queue 发送一个消息。

```
211 ************************************************************************************************************
212 *                                             POST TASK
213 ************************************************************************************************************
214 */
215 static  void  AppTaskPost ( void * p_arg )
216 {
217   OS_ERR      err;
218
219
220   (void)p_arg;
221
222
223   while (DEF_TRUE) {                                            //任务体
224     /* 发布消息到消息队列 queue */
225     OSQPost ((OS_Q         *)&queue,                            //消息变量指针
226              (void         *)"Binghuo uC/OS-III",               //要发送的数据的指针，将内存块首地址通过队列“发送出去”
227              (OS_MSG_SIZE   )sizeof ( "Binghuo uC/OS-III" ),    //数据字节大小
228              (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
229              (OS_ERR       *)&err);                             //返回错误类型
230
231     OSTimeDlyHMSM ( 0, 0, 0, 500, OS_OPT_TIME_DLY, & err );    //每隔500ms发送一次
232
233   }
234
235 }
```

图 8-25 AppTaskPost () 任务函数

任务函数 AppTaskPend () 的定义如下。任务等待接收消息队列的消息，如果成功接收到消息，就把消息的长度和内容打印到串口调试助手上。

app_cfg.h app.c

```
239 *********************************************************************************************************
240 *                                              PEND TASK
241 *********************************************************************************************************
242 */
243 static  void  AppTaskPend ( void * p_arg )
244 {
245   OS_ERR      err;
246   OS_MSG_SIZE msg_size;
247   CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必需该宏，该宏声明和
248                   //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
249                   // SR（临界段关中断只需保存SR），开中断时将该值还原。
250   char * pMsg;
251
252
253   (void)p_arg;
254
255
256   while (DEF_TRUE) {                                          //任务体
257     /* 请求消息队列 queue 的消息 */
258     pMsg = OSQPend ((OS_Q         *)&queue,                   //消息变量指针
259                     (OS_TICK       )0,                        //等待时长为无限
260                     (OS_OPT        )OS_OPT_PEND_BLOCKING,     //如果没有获取到信号量就等待
261                     (OS_MSG_SIZE  *)&msg_size,                //获取消息的字节大小
262                     (CPU_TS       *)0,                        //获取任务发送时的时间戳
263                     (OS_ERR       *)&err);                    //返回错误
264
265     if ( err == OS_ERR_NONE )                                 //如果接收成功
266     {
267       OS_CRITICAL_ENTER();                                    //进入临界段
268
269       printf ( "\r\n接收消息的长度：%d字节，内容：%s\r\n", msg_size, pMsg );
270
271       OS_CRITICAL_EXIT();
272
273     }
274
275   }
276
277 }
```

图 8-26 AppTaskPend () 任务函数

把 STM32 的 USART1 连接至电脑的串口调试助手，编译和下载程序到秉火 STM32 开发板，运行程序。用户可以看到串口调试助手上打印任务 AppTaskPend () 接收到的消息长度和内容。

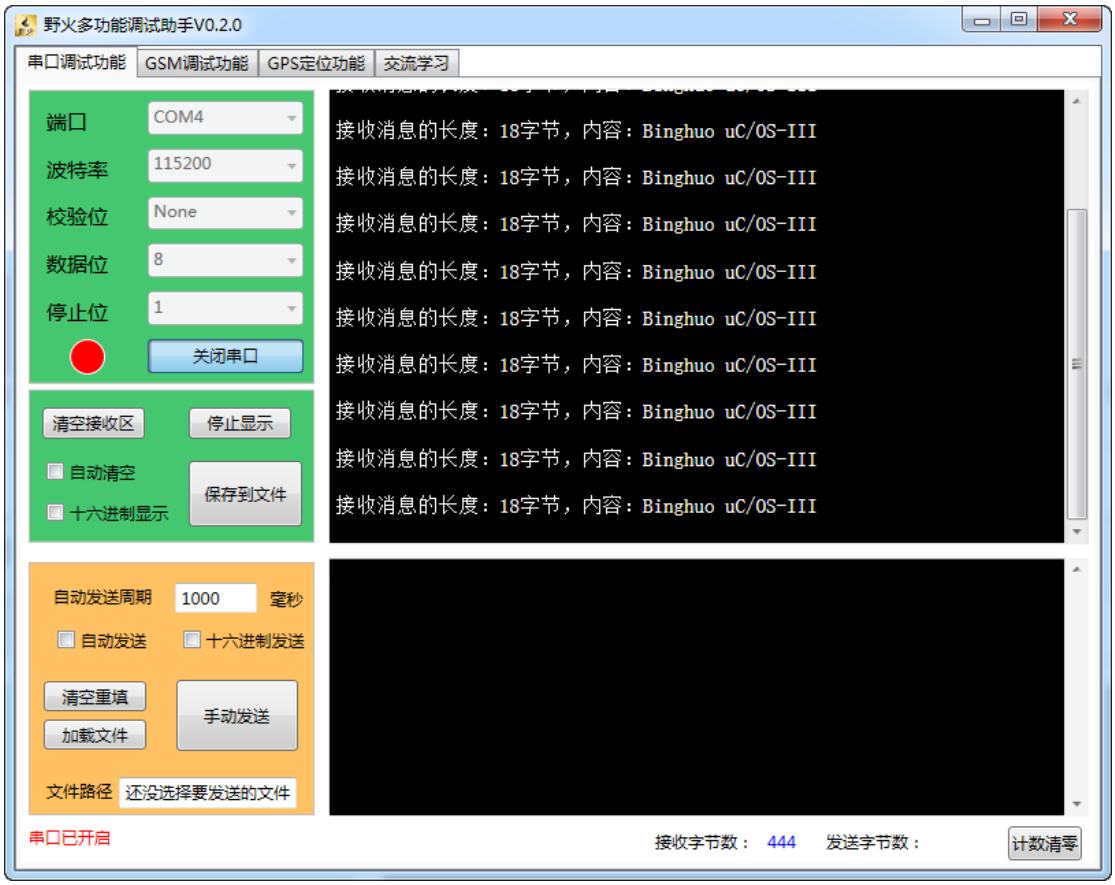

图 8-27 串口调试助手

8.3 章末总结

与信号量相比，消息队列可以传递更多的数据和信息。消息队列可以有多个，但消息池只有一个，所有的消息队列都共用一个消息池，当有消息队列需要使用消息时，就从消息池“舀”一个，用完了再“倒”回消息池。

使用消息队列之前必须先创建它，创建消息队列使用 OSQCreate () 函数。

OSQPend () 函数用于等待消息队列的消息，如果消息队列里没有消息，可以选择等待或者不等待。与之相对应，OSQPost () 函数则用于发布消息到消息队列。

OSQPendAbort () 函数用于中止任务对一个消息队列的等待。OSQDel() 函数用于删除一个消息队列。OSQFlush () 函数用于清空一个消息队列，清空的是消息队列里的消息，消息队列依然存在。

第9章 事件标志组

事件标志组，顾名思义，就是若干个事件标志的组合，代表若干个事件是否发生，通常用于集合两个或两个以上事件的状态。

9.1 原理简述

如果想要使用事件标志组，就必须事先使能事件标志组。消息队列的使能位于“os_cfg.h”。

```
51                                          /* ---------------------------- EVENT FLAGS -
52  #define OS_CFG_FLAG_EN                   1u   //使能/禁用事件标志组
53  #define OS_CFG_FLAG_DEL_EN               1u   //使能/禁用 OSFlagDel() 函数
54  #define OS_CFG_FLAG_MODE_CLR_EN          1u   //使能/禁用标志位清0触发模式
55  #define OS_CFG_FLAG_PEND_ABORT_EN        1u   //使能/禁用 OSFlagPendAbort() 函数
56
```

图 9-1

9.1.1 OSFlagCreate ()

要使用 uC/OS 的事件标志组必须先声明和创建事件标志组，调用 OSFlagCreate () 函数可以创建一个事件标志组。OSFlagCreate () 函数的信息如下表所示。

表 30 OSFlagCreate ()

函数原型	void OSFlagCreate (OS_FLAG_GRP *p_grp, CPU_CHAR *p_name, OS_FLAGS flags, OS_ERR *p_err);			
功能	创建一个事件标志组。			
参数	p_grp	事件标志组指针。		
	p_name	事件标志组名称。		
	flags	事件标志初始值。		
	p_err	返回错误类型	OS_ERR_NONE	无错误，创建成功。
			OS_ERR_CREATE_ISR	在中断中调用该函数
			OS_ERR_ILLEGAL_CREATE_RUN_TIME	在调用 OSSafetyCriticalStart() 函数后创建内核对象
			OS_ERR_NAME	p_name 为空指针
			OS_ERR_OBJ_CREATED	该事件标志组已经被创建过
			OS_ERR_OBJ_PTR_NULL	p_q 是个空指针
返	无。			

回值	
注意事项	✧ 创建前必须先为 p_grp 声明一个消息队列对象（OS_FLAG_GRP）。 ✧ 不可以在中断中调用该函数。

OSFlagCreate () 函数的定义位于“os_flag.c”。

```
70  void  OSFlagCreate (OS_FLAG_GRP  *p_grp,  //事件标志组指针
71                      CPU_CHAR     *p_name, //命名事件标志组
72                      OS_FLAGS      flags,  //标志初始值
73                      OS_ERR       *p_err)  //返回错误类型
74  {
75      CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必需该宏，该宏声明和
76                      //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
77                      // SR（临界段关中断只需保存SR），开中断时将该值还原。
78
79  #ifdef OS_SAFETY_CRITICAL                //如果使能了安全检测
80      if (p_err == (OS_ERR *)0) {          //如果错误类型实参为空
81          OS_SAFETY_CRITICAL_EXCEPTION(); //执行安全检测异常函数
82          return;                          //返回，停止执行
83      }
84  #endif
85
86  #ifdef OS_SAFETY_CRITICAL_IEC61508               //如果使能了安全关键
87      if (OSSafetyCriticalStartFlag == DEF_TRUE) { //如果在调用OSSafetyCriticalStart()后创建
88         *p_err = OS_ERR_ILLEGAL_CREATE_RUN_TIME;  //错误类型为“非法创建内核对象”
89          return;                                  //返回，停止执行
90      }
91  #endif
```

```
92
93  #if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u           //如果使能了中断中非法调用检测
94      if (OSIntNestingCtr > (OS_NESTING_CTR)0) { //如果该函数是在中断中被调用
95         *p_err = OS_ERR_CREATE_ISR;             //错误类型为“在中断中创建对象”
96          return;                                //返回，停止执行
97      }
98  #endif
99
100 #if OS_CFG_ARG_CHK_EN > 0u             //如果使能了参数检测
101     if (p_grp == (OS_FLAG_GRP *)0) { //如果 p_grp 为空
102        *p_err = OS_ERR_OBJ_PTR_NULL; //错误类型为“创建对象为空”
103         return;                      //返回，停止执行
104     }
105 #endif
106
```

```
107     OS_CRITICAL_ENTER();                  //进入临界段
108     p_grp->Type    = OS_OBJ_TYPE_FLAG; //标记创建对象数据结构为事件标志组
109     p_grp->NamePtr = p_name;           //标记事件标志组的名称
110     p_grp->Flags   = flags;            //设置标志初始值
111     p_grp->TS      = (CPU_TS)0;        //清零事件标志组的时间戳
112     OS_PendListInit(&p_grp->PendList); //初始化该事件标志组的等待列表
113
114 #if OS_CFG_DBG_EN > 0u                 //如果使能了调试代码和变量
115     OS_FlagDbgListAdd(p_grp);          //将该标志组添加到事件标志组双向调试链表
116 #endif
117     OSFlagQty++;                       //事件标志组个数加1
118
119     OS_CRITICAL_EXIT_NO_SCHED();       //退出临界段（无调度）
120    *p_err = OS_ERR_NONE;               //错误类型为“无错误”
121 }
```

图 9-2 OSFlagCreate () 函数

其中，OSFlagCreate () 函数调用了 OS_PendListInit() 函数初始化了事件标志组的等待列表。每个事件标志组都有一个等待列表，凡是等待该事件标志组的的任务都会被插入到这个等待列表，方便高效管理。OS_PendListInit() 函数的定义位于“os_core.c”。

```
1319 void  OS_PendListInit (OS_PEND_LIST  *p_pend_list)
1320 {
1321     p_pend_list->HeadPtr    = (OS_PEND_DATA *)0;    //复位等待列表的所有成员
1322     p_pend_list->TailPtr    = (OS_PEND_DATA *)0;
1323     p_pend_list->NbrEntries = (OS_OBJ_QTY    )0;
1324 }
```

图 9-3 OS_PendListInit() 函数

如果使能了 OS_CFG_DBG_EN（位于“os_cfg.h”），创建事件标志组时还会调用 OS_FlagDbgListAdd() 函数将该事件标志组插入到一个事件标志组调试列表，是为方便调试所设。OS_FlagDbgListAdd () 函数的定义位于“os_flag.c”。

```
1020 #if OS_CFG_DBG_EN > 0u                          //如果使能（默认使能）了调试代码和变量
1021 void  OS_FlagDbgListAdd (OS_FLAG_GRP  *p_grp) //将事件标志组插入到事件标志组调试列表的最前端
1022 {
1023     p_grp->DbgNamePtr               = (CPU_CHAR    *)((void *)" ");//先不指向任何任务的名称
1024     p_grp->DbgPrevPtr               = (OS_FLAG_GRP *)0;            //将该标志组作为列表的最前端
1025     if (OSFlagDbgListPtr == (OS_FLAG_GRP *)0) {                    //如果列表为空
1026         p_grp->DbgNextPtr           = (OS_FLAG_GRP *)0;            //该队列的下一个元素也为空
1027     } else {                                                       //如果列表非空
1028         p_grp->DbgNextPtr           =  OSFlagDbgListPtr;           //列表原来的首元素作为该队列的下一个元素
1029         OSFlagDbgListPtr->DbgPrevPtr =  p_grp;                     //原来的首元素的前面变为了该队列
1030     }
1031     OSFlagDbgListPtr                =  p_grp;                      //该标志组成为列表的新首元素
1032 }
```

图 9-4 OS_FlagDbgListAdd () 函数

9.1.2 OSFlagPost ()

OSFlagPost () 函数用于发布一个事件标志组。OSFlagPost () 函数的信息如下表所示。

表 31 OSFlagPost ()

函数原型	OS_FLAGS OSFlagPost (OS_FLAG_GRP *p_grp, OS_FLAGS flags, OS_OPT opt,, OS_ERR *p_err);		
功能	发布一个事件标志组。		
参数	p_grp	事件标志组指针。	
	flags	选定要操作的标志位。	
	opt	OS_OPT_POST_FLAG_SET	把选定的标志位置 1。
		OS_OPT_POST_FLAG_CLR	把选定的标志位清 0。
		OS_OPT_POST_FLAG_SET | OS_OPT_POST_NO_SCHED	把消息发布到队列的末端，并且唤醒全部等待任务；不进行任务调度，继续运行当前任务。
		OS_OPT_POST_FLAG_CLR |	把消息发布到队列的前端，并且唤醒全部等待

		OS_OPT_POST_NO_SCHED	任务；不进行任务调度，继续运行当前任务。	
	p_err	返回错误类型	OS_ERR_NONE	调用成功。
			OS_ERR_OBJ_PTR_NULL	p_grp 为空。
			OS_ERR_OBJ_TYPE	p_grp 不是事件标志组类型。
			OS_ERR_OPT_INVALID	选项非法。
返回值	事件标志组的标志值。			

OSFlagPost () 函数的定义也位于“os_flag.c

```
820 OS_FLAGS  OSFlagPost (OS_FLAG_GRP  *p_grp, //事件标志组指针
821                       OS_FLAGS     flags, //选定要操作的标志位
822                       OS_OPT       opt,   //选项
823                       OS_ERR       *p_err) //返回错误类型
824 {
825     OS_FLAGS  flags_cur;
826     CPU_TS    ts;
827
828
829
830 #ifdef OS_SAFETY_CRITICAL                 //如果使能（默认禁用）了安全检测
831     if (p_err == (OS_ERR *)0) {           //如果错误类型实参为空
832         OS_SAFETY_CRITICAL_EXCEPTION();   //执行安全检测异常函数
833         return ((OS_FLAGS)0);             //返回0，停止执行
834     }
835 #endif
836
837 #if OS_CFG_ARG_CHK_EN > 0u                //如果使能（默认使能）了参数检测
838     if (p_grp == (OS_FLAG_GRP *)0) {      //如果参数 p_grp 为空
839        *p_err  = OS_ERR_OBJ_PTR_NULL;     //错误类型为“事件标志组对象为空”
840         return ((OS_FLAGS)0);             //返回0，停止执行
841     }
842     switch (opt) {                        //根据选项分类处理
843         case OS_OPT_POST_FLAG_SET:        //如果选项在预期之内
844         case OS_OPT_POST_FLAG_CLR:
845         case OS_OPT_POST_FLAG_SET | OS_OPT_POST_NO_SCHED:
846         case OS_OPT_POST_FLAG_CLR | OS_OPT_POST_NO_SCHED:
847              break;                       //直接跳出
848
849         default:                          //如果选项超出预期
850             *p_err = OS_ERR_OPT_INVALID; //错误类型为“选项非法”
851              return ((OS_FLAGS)0);        //返回0，停止执行
852     }
853 #endif
854
855 #if OS_CFG_OBJ_TYPE_CHK_EN > 0u           //如果使能了对象类型检测
856     if (p_grp->Type != OS_OBJ_TYPE_FLAG) { //如果 p_grp 不是事件标志组类型
857        *p_err = OS_ERR_OBJ_TYPE;          //错误类型“对象类型有误”
858         return ((OS_FLAGS)0);             //返回0，停止执行
859     }
860 #endif
861
```

```
862     ts = OS_TS_GET();                                           //获取时间戳
863 #if OS_CFG_ISR_POST_DEFERRED_EN > 0u                           //如果使能了中断延迟发布
864     if (OSIntNestingCtr > (OS_NESTING_CTR)0) {                  //如果该函数是在中断中被调用
865         OS_IntQPost((OS_OBJ_TYPE)OS_OBJ_TYPE_FLAG,              //将该标志组发布到中断消息队列
866                     (void       *)p_grp,
867                     (void       *)0,
868                     (OS_MSG_SIZE)0,
869                     (OS_FLAGS    )flags,
870                     (OS_OPT      )opt,
871                     (CPU_TS      )ts,
872                     (OS_ERR     *)p_err);
873         return ((OS_FLAGS)0);                                   //返回0，停止执行
874     }
875 #endif
876     /* 如果没有使能中断延迟发布 */
877     flags_cur = OS_FlagPost(p_grp,                              //将标志组直接发布
878                             flags,
879                             opt,
880                             ts,
881                             p_err);
882
883     return (flags_cur);                                         //返回当前标志位的值
884 }
```

图 9-5 OSFlagPost () 函数

其实，不管是否使能了中断延迟发布，最终都是调用 OS_FlagPost() 函数进行发布事件标志组。只是使能了中断延迟发布的发布过程会比较曲折，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。OS_FlagPost() 函数的定义位于“os_flag.c”。

os_core.c | os_flag.c

```
1109 OS_FLAGS  OS_FlagPost (OS_FLAG_GRP  *p_grp, //事件标志组指针
1110                        OS_FLAGS      flags, //选定要操作的标志位
1111                        OS_OPT        opt,   //选项
1112                        CPU_TS        ts,    //时间戳
1113                        OS_ERR       *p_err) //返回错误类型
1114 {
1115     OS_FLAGS         flags_cur;
1116     OS_FLAGS         flags_rdy;
1117     OS_OPT           mode;
1118     OS_PEND_DATA    *p_pend_data;
1119     OS_PEND_DATA    *p_pend_data_next;
1120     OS_PEND_LIST    *p_pend_list;
1121     OS_TCB          *p_tcb;
1122     CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必需该宏，该宏声明和
1123                     //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
1124                     // SR（临界段关中断只需保存SR），开中断时将该值还原。
1125
```

```
1126      CPU_CRITICAL_ENTER();                                    //关中断
1127      switch (opt) {                                           //根据选项分类处理
1128          case OS_OPT_POST_FLAG_SET:                           //如果要求将选定位置1
1129          case OS_OPT_POST_FLAG_SET | OS_OPT_POST_NO_SCHED:
1130               p_grp->Flags |=  flags;                         //将选定位置1
1131               break;                                          //跳出
1132
1133          case OS_OPT_POST_FLAG_CLR:                           //如果要求将选定位请0
1134          case OS_OPT_POST_FLAG_CLR | OS_OPT_POST_NO_SCHED:
1135               p_grp->Flags &= ~flags;                         //将选定位请0
1136               break;                                          //跳出
1137
1138          default:                                             //如果选项超出预期
1139               CPU_CRITICAL_EXIT();                            //开中断
1140              *p_err = OS_ERR_OPT_INVALID;                     //错误类型为“选项非法”
1141               return ((OS_FLAGS)0);                           //返回0，停止执行
1142      }
1143      p_grp->TS   = ts;                                        //将时间戳存入事件标志组
1144      p_pend_list = &p_grp->PendList;                          //获取事件标志组的等待列表
1145      if (p_pend_list->NbrEntries == 0u) {                     //如果没有任务在等待标志组
1146          CPU_CRITICAL_EXIT();                                 //开中断
1147         *p_err = OS_ERR_NONE;                                 //错误类型为“无错误”
1148          return (p_grp->Flags);                               //返回事件标志组的标志值
1149      }

1150      /* 如果有任务在等待标志组 */
1151      OS_CRITICAL_ENTER_CPU_EXIT();                            //进入临界段，重开中断
1152      p_pend_data = p_pend_list->HeadPtr;                      //获取等待列表头个等待任务
1153      p_tcb       = p_pend_data->TCBPtr;
1154      while (p_tcb != (OS_TCB *)0) {                           //从头至尾遍历等待列表的所有任务
1155          p_pend_data_next = p_pend_data->NextPtr;
1156          mode             = p_tcb->FlagsOpt & OS_OPT_PEND_FLAG_MASK; //获取任务的标志选项
1157          switch (mode) {                                      //根据任务的标志选项分类处理
1158              case OS_OPT_PEND_FLAG_SET_ALL:                   //如果要求任务等待的标志位都得置1
1159                   flags_rdy = (OS_FLAGS)(p_grp->Flags & p_tcb->FlagsPend);
1160                   if (flags_rdy == p_tcb->FlagsPend) {        //如果任务等待的标志位都置1了
1161                       OS_FlagTaskRdy(p_tcb,                   //让该任务准备运行
1162                                      flags_rdy,
1163                                      ts);
1164                   }
1165                   break;                                      //跳出
1166
1167              case OS_OPT_PEND_FLAG_SET_ANY:                   //如果要求任务等待的标志位有1位置1即可
1168                   flags_rdy = (OS_FLAGS)(p_grp->Flags & p_tcb->FlagsPend);
1169                   if (flags_rdy != (OS_FLAGS)0) {             //如果任务等待的标志位有置1的
1170                       OS_FlagTaskRdy(p_tcb,                   //让该任务准备运行
1171                                      flags_rdy,
1172                                      ts);
1173                   }
1174                   break;                                     //跳出
1175

1176  #if OS_CFG_FLAG_MODE_CLR_EN > 0u                             //如果使能了标志位清0检测
1177              case OS_OPT_PEND_FLAG_CLR_ALL:                   //如果要求任务等待的标志位都得请0
1178                   flags_rdy = (OS_FLAGS)(~p_grp->Flags & p_tcb->FlagsPend);
1179                   if (flags_rdy == p_tcb->FlagsPend) {//如果任务等待的标志位都请0了
1180                       OS_FlagTaskRdy(p_tcb,                   //让该任务准备运行
1181                                      flags_rdy,
1182                                      ts);
1183                   }
1184                   break;            //跳出
1185
1186              case OS_OPT_PEND_FLAG_CLR_ANY:                  //如果要求任务等待的标志位有1位请0即可
1187                   flags_rdy = (OS_FLAGS)(~p_grp->Flags & p_tcb->FlagsPend);
1188                   if (flags_rdy != (OS_FLAGS)0) {    //如果任务等待的标志位有请0的
1189                       OS_FlagTaskRdy(p_tcb,           //让该任务准备运行
1190                                      flags_rdy,
1191                                      ts);
1192                   }
1193                   break;                                    //跳出
1194  #endif
```

```
1195            default:                                    //如果标志选项超出预期
1196                 OS_CRITICAL_EXIT();                    //退出临界段
1197                *p_err = OS_ERR_FLAG_PEND_OPT;          //错误类型为“标志选项非法”
1198                 return ((OS_FLAGS)0);                  //返回0，停止运行
1199        }
1200        p_pend_data = p_pend_data_next;                 //准备处理下一个等待任务
1201        if (p_pend_data != (OS_PEND_DATA *)0) {         //如果该任务存在
1202            p_tcb = p_pend_data->TCBPtr;                //获取该任务的任务控制块
1203        } else {                                        //如果该任务不存在
1204            p_tcb = (OS_TCB *)0;                        //清空 p_tcb，退出 while 循环
1205        }
1206    }
1207    OS_CRITICAL_EXIT_NO_SCHED();                    //退出临界段（无调度）
1208
1209    if ((opt & OS_OPT_POST_NO_SCHED) == (OS_OPT)0) {  //如果 opt 没选择“发布时不调度任务”
1210        OSSched();                                      //任务调度
1211    }
1212
1213    CPU_CRITICAL_ENTER();           //关中断
1214    flags_cur = p_grp->Flags;       //获取事件标志组的标志值
1215    CPU_CRITICAL_EXIT();            //开中断
1216   *p_err     = OS_ERR_NONE;        //错误类型为“无错误”
1217    return (flags_cur);             //返回事件标志组的当前标志值
1218}
```

图 9-6 OS_FlagPost() 函数

在 OS_FlagPost() 函数中，会逐个遍历事件标志组的等待列表里的等待任务。如果已有任务在等待正在发布的事件标志组，而且该发布刚好吻合任务等待的条件，就会调用 OS_FlagTaskRdy() 函数让等待任务脱离等待列表，去除该等待任务的等待状态。OS_FlagTaskRdy() 函数的定义位于“os_flag.c”。

```
os_core.c  os_flag.c
1241 void   OS_FlagTaskRdy (OS_TCB     *p_tcb,              //任务控制块指针
1242                        OS_FLAGS    flags_rdy,          //让任务就绪的标志位
1243                        CPU_TS      ts)                 //事件标志组被发布时的时间戳
1244 {
1245     p_tcb->FlagsRdy   = flags_rdy;                     //标记让任务就绪的事件标志位
1246     p_tcb->PendStatus = OS_STATUS_PEND_OK;             //清除任务的等待状态
1247     p_tcb->PendOn     = OS_TASK_PEND_ON_NOTHING;       //标记任务没有等待任何对象
1248     p_tcb->TS         = ts;                            //记录任务脱离等待时的时间戳
1249     switch (p_tcb->TaskState) {                        //根据任务的任务状态分类处理
1250         case OS_TASK_STATE_RDY:                        //如果任务是就绪状态
1251         case OS_TASK_STATE_DLY:                        //如果任务是延时状态
1252         case OS_TASK_STATE_DLY_SUSPENDED:              //如果任务是延时中被挂起状态
1253         case OS_TASK_STATE_SUSPENDED:                  //如果任务是被挂起状态
1254              break;                                    //直接跳出，不需处理
1255
1256         case OS_TASK_STATE_PEND:                     //如果任务是无期限等待状态
1257         case OS_TASK_STATE_PEND_TIMEOUT:             //如果任务是有期限等待状态
1258              OS_TaskRdy(p_tcb);                      //让任务进入就绪状态
1259              p_tcb->TaskState = OS_TASK_STATE_RDY;   //修改任务的状态为就绪状态
1260              break;                                  //跳出
1261
1262         case OS_TASK_STATE_PEND_SUSPENDED:                //如果任务是无期限等待中被挂起状态
1263         case OS_TASK_STATE_PEND_TIMEOUT_SUSPENDED:        //如果任务是有期限等待中被挂起状态
1264              p_tcb->TaskState = OS_TASK_STATE_SUSPENDED;//修改任务的状态为被挂起状态
1265              break;                                    //跳出
1266
1267         default:                                       //如果任务状态超出预期
1268              break;                                    //直接跳出
1269     }
1270     OS_PendListRemove(p_tcb);                          //将任务从等待列表移除
1271 }
```

图 9-7 OS_FlagTaskRdy () 函数

9.1.3 OSFlagPend ()

与 OSFlagPost () 事件标志组发布函数相对应，OSFlagPend () 函数用于等待事件标志组。

表 32 OSFlagPend ()

函数原型	OS_FLAGS OSFlagPend (OS_FLAG_GRP *p_grp, OS_FLAGS flags, OS_TICK timeout, OS_OPT opt, CPU_TS *p_ts, OS_ERR *p_err);			
功能	等待一个事件标志组的事件组合发生。			
参数	p_grp	事件标志组指针。		
	flags	要等待的事件（位）的组合		
	timeout	等待超时时间（单位：时钟节拍），0 代表无期限等待。opt 为 OS_OPT_PEND_BLOCKING 时该参数才起作用。		
	opt	选项	OS_OPT_PEND_FLAG_CLR_ALL	等待 flags 指定位均被清 0。
			OS_OPT_PEND_FLAG_CLR_ANY	等待 flags 指定位有一位被清 0 即可。
			OS_OPT_PEND_FLAG_SET_ALL	等待 flags 指定位均被置 1。
			OS_OPT_PEND_FLAG_SET_ANY	等待 flags 指定位有一位被置 1 即可。
			OS_OPT_PEND_FLAG_CLR_ALL | OS_OPT_PEND_FLAG_CONSUME	等待 flags 指定位均被清 0；等到后把触发位取反。
			OS_OPT_PEND_FLAG_CLR_ANY | OS_OPT_PEND_FLAG_CONSUME	等待 flags 指定位有一位被清 0 即可；等到后把触发位取反。
			OS_OPT_PEND_FLAG_SET_ALL | OS_OPT_PEND_FLAG_CONSUME	等待 flags 指定位均被置 1；等到后把触发位取反。
			OS_OPT_PEND_FLAG_SET_ANY | OS_OPT_PEND_FLAG_CONSUME	等待 flags 指定位有一位被置 1 即可；等到后把触发位取反。
			OS_OPT_PEND_FLAG_CLR_ALL | OS_OPT_PEND_NON_BLOCKING	要求 flags 指定位均被清 0；不符合要求不等待。
			OS_OPT_PEND_FLAG_CLR_ANY | OS_OPT_PEND_NON_BLOCKING	要求 flags 指定位有一位被清 0；不符合要求不等待。
			OS_OPT_PEND_FLAG_SET_ALL | OS_OPT_PEND_NON_BLOCKING	要求 flags 指定位均被置 1；不符合要求不等待。
			OS_OPT_PEND_FLAG_SET_ANY | OS_OPT_PEND_NON_BLOCKING	要求 flags 指定位有一位被置 1；不符合要求不等待。
			OS_OPT_PEND_FLAG_CLR_ALL | OS_OPT_PEND_FLAG_CONSUME | OS_OPT_PEND_NON_BLOCKING	要求 flags 指定位均被清 0；符合要求就把触发位取反；不符合要求不等待；。

			OS_OPT_PEND_FLAG_CLR_ANY | OS_OPT_PEND_FLAG_CONSUME | OS_OPT_PEND_NON_BLOCKING	要求 flags 指定位有一位被清 0；符合要求就把触发位取反；不符合要求不等待；。
			OS_OPT_PEND_FLAG_SET_ALL | OS_OPT_PEND_FLAG_CONSUME | OS_OPT_PEND_NON_BLOCKING	要求 flags 指定位均被置 1；符合要求就把触发位取反；不符合要求不等待；。
			OS_OPT_PEND_FLAG_SET_ANY | OS_OPT_PEND_FLAG_CONSUME | OS_OPT_PEND_NON_BLOCKING	要求 flags 指定位有一位被置 1；符合要求就把触发位取反；不符合要求不等待；。
	p_ts	时间戳	用于存储事件标志组最后一次被发布消息的时间戳，或者等待被中止的时间戳，或者事件标志组被删除时的时间戳，具体返回哪个时间戳，要根据返回的 p_err 判断。该参数可以为 NULL，表示用户不需要获得时间戳。	
	p_err	返回错误类型	OS_ERR_NONE	等待成功，指定事件组合发生。
			OS_ERR_OBJ_PTR_NULL	p_grp 为空。
			OS_ERR_OBJ_TYPE	p_grp 不是事件标志组类型对象。
			OS_ERR_PEND_ABORT	等待被中止。
			OS_ERR_PEND_ISR	在中断中被调用。
			OS_ERR_PEND_WOULD_BLOCK	缺乏堵塞。
			OS_ERR_SCHED_LOCKED	调度器被锁。
			OS_ERR_TIMEOUT	等待超时。
返回值	✧ != (void *)0，任务脱离等待时的事件标志组的标志成员值。 ✧ == (void *)0，有错误，或者等待超时。			
注意事项	✧ 不可以在中断中调用该函数。			

OSFlagPend () 函数的定义也位于“os_flag.c”。

```
315 OS_FLAGS  OSFlagPend (OS_FLAG_GRP  *p_grp,    //事件标志组指针
316                       OS_FLAGS      flags,    //选定要操作的标志位
317                       OS_TICK       timeout,  //等待期限（单位：时钟节拍）
318                       OS_OPT        opt,      //选项
319                       CPU_TS       *p_ts,     //返回等到事件标志时的时间戳
320                       OS_ERR       *p_err)    //返回错误类型
321 {
322     CPU_BOOLEAN   consume;
323     OS_FLAGS      flags_rdy;
324     OS_OPT        mode;
325     OS_PEND_DATA  pend_data;
326     CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必需该宏，该宏声明和
327                     //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
328                     // SR（临界段关中断只需保存SR），开中断时将该值还原。
329
330 #ifdef OS_SAFETY_CRITICAL                 //如果使能（默认禁用）了安全检测
331     if (p_err == (OS_ERR *)0) {           //如果错误类型实参为空
332         OS_SAFETY_CRITICAL_EXCEPTION(); //执行安全检测异常函数
333         return ((OS_FLAGS)0);             //返回0（有错误），停止执行
334     }
335 #endif
336
```

```
337 #if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u          //如果使能了中断中非法调用检测
338     if (OSIntNestingCtr > (OS_NESTING_CTR)0) {  //如果该函数在中断中被调用
339         *p_err = OS_ERR_PEND_ISR;               //错误类型为“在中断中中止等待”
340          return ((OS_FLAGS)0);                  //返回0（有错误），停止执行
341     }
342 #endif
343
```

```
344 #if OS_CFG_ARG_CHK_EN > 0u                  //如果使能了参数检测
345     if (p_grp == (OS_FLAG_GRP *)0) {        //如果 p_grp 为空
346         *p_err = OS_ERR_OBJ_PTR_NULL;       //错误类型为“对象为空”
347          return ((OS_FLAGS)0);              //返回0（有错误），停止执行
348     }
349     switch (opt) {                          //根据选项分类处理
350         case OS_OPT_PEND_FLAG_CLR_ALL:      //如果选项在预期内
351         case OS_OPT_PEND_FLAG_CLR_ANY:
352         case OS_OPT_PEND_FLAG_SET_ALL:
353         case OS_OPT_PEND_FLAG_SET_ANY:
354         case OS_OPT_PEND_FLAG_CLR_ALL | OS_OPT_PEND_FLAG_CONSUME:
355         case OS_OPT_PEND_FLAG_CLR_ANY | OS_OPT_PEND_FLAG_CONSUME:
356         case OS_OPT_PEND_FLAG_SET_ALL | OS_OPT_PEND_FLAG_CONSUME:
357         case OS_OPT_PEND_FLAG_SET_ANY | OS_OPT_PEND_FLAG_CONSUME:
358         case OS_OPT_PEND_FLAG_CLR_ALL | OS_OPT_PEND_NON_BLOCKING:
359         case OS_OPT_PEND_FLAG_CLR_ANY | OS_OPT_PEND_NON_BLOCKING:
360         case OS_OPT_PEND_FLAG_SET_ALL | OS_OPT_PEND_NON_BLOCKING:
361         case OS_OPT_PEND_FLAG_SET_ANY | OS_OPT_PEND_NON_BLOCKING:
362         case OS_OPT_PEND_FLAG_CLR_ALL | OS_OPT_PEND_FLAG_CONSUME | OS_OPT_PEND_NON_BLOCKING:
363         case OS_OPT_PEND_FLAG_CLR_ANY | OS_OPT_PEND_FLAG_CONSUME | OS_OPT_PEND_NON_BLOCKING:
364         case OS_OPT_PEND_FLAG_SET_ALL | OS_OPT_PEND_FLAG_CONSUME | OS_OPT_PEND_NON_BLOCKING:
365         case OS_OPT_PEND_FLAG_SET_ANY | OS_OPT_PEND_FLAG_CONSUME | OS_OPT_PEND_NON_BLOCKING:
366              break;                         //直接跳出
367
368         default:                            //如果选项超出预期
369             *p_err = OS_ERR_OPT_INVALID;//错误类型为“选项非法”
370              return ((OS_OBJ_QTY)0);        //返回0（有错误），停止执行
371     }
372 #endif
```

```
373
374 #if OS_CFG_OBJ_TYPE_CHK_EN > 0u                    //如果使能了对象类型检测
375     if (p_grp->Type != OS_OBJ_TYPE_FLAG) {        //如果 p_grp 不是事件标志组类型
376         *p_err = OS_ERR_OBJ_TYPE;                 //错误类型为“对象类型有误”
377          return ((OS_FLAGS)0);                    //返回0（有错误），停止执行
378     }
379 #endif
380
```

```
381     if ((opt & OS_OPT_PEND_FLAG_CONSUME) != (OS_OPT)0) { //选择了标志位匹配后自动取反
382         consume = DEF_TRUE;
383     } else {                                          //未选择标志位匹配后自动取反
384         consume = DEF_FALSE;
385     }
386
387     if (p_ts != (CPU_TS *)0) {          //如果 p_ts 非空
388        *p_ts = (CPU_TS)0;               //初始化（清零）p_ts，待用于返回时间戳
389     }
390
```

```
391     mode = opt & OS_OPT_PEND_FLAG_MASK;                          //从选项中提取对标志位的要求
392     CPU_CRITICAL_ENTER();                                        //关中断
393     switch (mode) {                                              //根据事件触发模式分类处理
394         case OS_OPT_PEND_FLAG_SET_ALL:                           //如果要求所有标志位均要置1
395              flags_rdy = (OS_FLAGS)(p_grp->Flags & flags);       //提取想要的标志位的值
396              if (flags_rdy == flags) {                           //如果该值与期望值匹配
397                  if (consume == DEF_TRUE) {                      //如果要求将标志位匹配后取反
398                      p_grp->Flags &= ~flags_rdy;                 //清0标志组的相关标志位
399                  }
400                  OSTCBCurPtr->FlagsRdy = flags_rdy;              //保存让任务脱离等待的标志值
401                  if (p_ts != (CPU_TS *)0) {                      //如果 p_ts 非空
402                     *p_ts  = p_grp->TS;                          //获取任务等到标志组时的时间戳
403                  }
404                  CPU_CRITICAL_EXIT();                            //开中断
405                 *p_err = OS_ERR_NONE;                            //错误类型为“无错误”
406                  return (flags_rdy);                             //返回让任务脱离等待的标志值
```

```
407              } else {                                            //如果想要标志位的值与期望值不匹配
408                  if ((opt & OS_OPT_PEND_NON_BLOCKING) != (OS_OPT)0) { //如果选择了不堵塞任务
409                      CPU_CRITICAL_EXIT();                        //关中断
410                     *p_err = OS_ERR_PEND_WOULD_BLOCK;            //错误类型为“渴求堵塞”
411                      return ((OS_FLAGS)0);                       //返回0（有错误），停止执行
412                  } else {                                        //如果选择了堵塞任务
413                      if (OSSchedLockNestingCtr > (OS_NESTING_CTR)0) { //如果调度器被锁
414                          CPU_CRITICAL_EXIT();                    //关中断
415                         *p_err = OS_ERR_SCHED_LOCKED;            //错误类型为“调度器被锁”
416                          return ((OS_FLAGS)0);                   //返回0（有错误），停止执行
417                      }
418                  }
419                  /* 如果调度器未被锁 */
420                  OS_CRITICAL_ENTER_CPU_EXIT();                   //进入临界段，重开中断
421                  OS_FlagBlock(&pend_data,                        //阻塞当前运行任务，等待事件标志组
422                               p_grp,
423                               flags,
424                               opt,
425                               timeout);
426                  OS_CRITICAL_EXIT_NO_SCHED();                    //退出临界段（无调度）
427              }
428              break;                                              //跳出
429
```

```
430         case OS_OPT_PEND_FLAG_SET_ANY:                          //如果要求有标志位被置1即可
431              flags_rdy = (OS_FLAGS)(p_grp->Flags & flags);      //提取想要的标志位的值
432              if (flags_rdy != (OS_FLAGS)0) {                    //如果有位被置1
433                  if (consume == DEF_TRUE) {                     //如果要求将标志位匹配后取反
434                      p_grp->Flags &= ~flags_rdy;                //清0湿巾标志组的相关标志位
435                  }
436                  OSTCBCurPtr->FlagsRdy = flags_rdy;             //保存让任务脱离等待的标志值
437                  if (p_ts != (CPU_TS *)0) {                     //如果 p_ts 非空
438                     *p_ts  = p_grp->TS;                         //获取任务等到标志组时的时间戳
439                  }
440                  CPU_CRITICAL_EXIT();                           //开中断
441                 *p_err = OS_ERR_NONE;                           //错误类型为“无错误”
442                  return (flags_rdy);                            //返回让任务脱离等待的标志值


443              } else {                                           //如果没有位被置1
444                  if ((opt & OS_OPT_PEND_NON_BLOCKING) != (OS_OPT)0) { //如果没设置堵塞任务
445                      CPU_CRITICAL_EXIT();                       //关中断
446                     *p_err = OS_ERR_PEND_WOULD_BLOCK;           //错误类型为“渴求堵塞”
447                      return ((OS_FLAGS)0);                      //返回0（有错误），停止执行
448                  } else {                                       //如果设置了堵塞任务
449                      if (OSSchedLockNestingCtr > (OS_NESTING_CTR)0) { //如果调度器被锁
450                          CPU_CRITICAL_EXIT();                   //关中断
451                         *p_err = OS_ERR_SCHED_LOCKED;           //错误类型为“调度器被锁”
452                          return ((OS_FLAGS)0);                  //返回0（有错误），停止执行
453                      }
454                  }
455                  /* 如果调度器没被锁 */
456                  OS_CRITICAL_ENTER_CPU_EXIT();                  //进入临界段，重开中断
457                  OS_FlagBlock(&pend_data,                       //阻塞当前运行任务，等待事件标志组
458                               p_grp,
459                               flags,
460                               opt,
461                               timeout);
462                  OS_CRITICAL_EXIT_NO_SCHED();                   //退出中断（无调度）
463              }
464              break;                                             //跳出
465


466 #if OS_CFG_FLAG_MODE_CLR_EN > 0u                                //如果使能了标志位清0触发模式
467          case OS_OPT_PEND_FLAG_CLR_ALL:                         //如果要求所有标志位均要清0
468               flags_rdy = (OS_FLAGS)(~p_grp->Flags & flags);//提取想要的标志位的值
469               if (flags_rdy == flags) {                         //如果该值与期望值匹配
470                   if (consume == DEF_TRUE) {                    //如果要求将标志位匹配后清0
471                       p_grp->Flags |= flags_rdy;                //置1标志组的相关标志位
472                   }
473                   OSTCBCurPtr->FlagsRdy = flags_rdy;            //保存让任务脱离等待的标志值
474                   if (p_ts != (CPU_TS *)0) {                    //如果 p_ts 非空
475                      *p_ts  = p_grp->TS;                        //获取任务等到标志组时的时间戳
476                   }
477                   CPU_CRITICAL_EXIT();                          //开中断
478                  *p_err = OS_ERR_NONE;                          //错误类型为“无错误”
479                   return (flags_rdy);                           //返回0（有错误），停止执行
```

```
480             } else {                                            //如果想要标志位的值与期望值不匹配
481                 if ((opt & OS_OPT_PEND_NON_BLOCKING) != (OS_OPT)0) {  //如果选择了不堵塞任务
482                     CPU_CRITICAL_EXIT();                        //关中断
483                    *p_err = OS_ERR_PEND_WOULD_BLOCK;            //错误类型为“渴求堵塞”
484                     return ((OS_FLAGS)0);                       //返回0（有错误），停止执行
485                 } else {                                        //如果选择了堵塞任务
486                     if (OSSchedLockNestingCtr > (OS_NESTING_CTR)0) { //如果调度器被锁
487                         CPU_CRITICAL_EXIT();                    //关中断
488                        *p_err = OS_ERR_SCHED_LOCKED;            //错误类型为“调度器被锁”
489                         return ((OS_FLAGS)0);                   //返回0（有错误），停止执行
490                     }
491                 }
492                 /* 如果调度器未被锁 */
493                 OS_CRITICAL_ENTER_CPU_EXIT();                   //进入临界段，重开中断
494                 OS_FlagBlock(&pend_data,                        //阻塞当前运行任务，等待事件标志组
495                              p_grp,
496                              flags,
497                              opt,
498                              timeout);
499                 OS_CRITICAL_EXIT_NO_SCHED();                  //退出临界段（无调度）
500             }
501             break;                                              //跳出
502
```

```
503         case OS_OPT_PEND_FLAG_CLR_ANY:                          //如果要求有标志位被清0即可
504              flags_rdy = (OS_FLAGS)(~p_grp->Flags & flags);//提取想要的标志位的值
505              if (flags_rdy != (OS_FLAGS)0) {                    //如果有位被清0
506                  if (consume == DEF_TRUE) {                     //如果要求将标志位匹配后取反
507                      p_grp->Flags |= flags_rdy;                 //置1湿巾标志组的相关标志位
508                  }
509                  OSTCBCurPtr->FlagsRdy = flags_rdy;             //保存让任务脱离等待的标志值
510                  if (p_ts != (CPU_TS *)0) {                     //如果 p_ts 非空
511                     *p_ts  = p_grp->TS;                         //获取任务等到标志组时的时间戳
512                  }
513                  CPU_CRITICAL_EXIT();                           //开中断
514                 *p_err = OS_ERR_NONE;                           //错误类型为“无错误”
515                  return (flags_rdy);                            //返回0（有错误），停止执行
```

```
516             } else {                                            //如果没有位被清0
517                 if ((opt & OS_OPT_PEND_NON_BLOCKING) != (OS_OPT)0) { //如果没设置堵塞任务
518                     CPU_CRITICAL_EXIT();                        //开中断
519                    *p_err = OS_ERR_PEND_WOULD_BLOCK;            //错误类型为“渴求堵塞”
520                     return ((OS_FLAGS)0);                       //返回0（有错误），停止执行
521                 } else {                                        //如果设置了堵塞任务
522                     if (OSSchedLockNestingCtr > (OS_NESTING_CTR)0) { //如果调度器被锁
523                         CPU_CRITICAL_EXIT();                    //开中断
524                        *p_err = OS_ERR_SCHED_LOCKED;            //错误类型为“调度器被锁”
525                         return ((OS_FLAGS)0);                   //返回0（有错误），停止执行
526                     }
527                 }
528                 /* 如果调度器没被锁 */
529                 OS_CRITICAL_ENTER_CPU_EXIT();                   //进入临界段，重开中断
530                 OS_FlagBlock(&pend_data,                        //阻塞当前运行任务，等待事件标志组
531                              p_grp,
532                              flags,
533                              opt,
534                              timeout);
535                 OS_CRITICAL_EXIT_NO_SCHED();                  //退出中断（无调度）
536             }
537             break;                                              //跳出
538 #endif
```

```
539
540         default:                                                //如果要求超出预期
541              CPU_CRITICAL_EXIT();
542             *p_err = OS_ERR_OPT_INVALID;                        //错误类型为“选项非法”
543              return ((OS_FLAGS)0);                              //返回0（有错误），停止执行
544     }
545
546     OSSched();                                                  //任务调度
```

```
    /* 任务等到了事件标志组后得以继续运行 */
    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:                          //如果等到了事件标志组
             if (p_ts != (CPU_TS *)0) {                  //如果 p_ts 非空
                *p_ts  = OSTCBCurPtr->TS;                //返回等到标志组时的时间戳
             }
            *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;                                      //跳出


        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
             }
             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:                         //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:                                         //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) {                         //如果有错误存在
        return ((OS_FLAGS)0);                            //返回0（有错误），停止执行
    }


    /* 如果没有错误存在 */
    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;             //清0标志组里触发事件的标志位
                 break;                                  //跳出

#if OS_CFG_FLAG_MODE_CLR_EN > 0u                         //如果使能了标志位清0触发模式
            case OS_OPT_PEND_FLAG_CLR_ALL:               //如果是通过清0来标志事件的发生
            case OS_OPT_PEND_FLAG_CLR_ANY:
                 p_grp->Flags |=  flags_rdy;             //置1标志组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
}
```

图 9-8 OSFlagPend () 函数

在 OSFlagPend () 函数中，当需要阻塞当前运行任务，等待事件标志组的事件时，会调用 OS_FlagBlock () 函数。OS_FlagBlock () 函数的定义位于“os_flag.c”。

```
923 void  OS_FlagBlock (OS_PEND_DATA  *p_pend_data,   //等待列表元素
924                     OS_FLAG_GRP   *p_grp,         //事件标志组
925                     OS_FLAGS       flags,         //要操作的标志位
926                     OS_OPT         opt,           //选项
927                     OS_TICK        timeout)       //等待期限
928 {
929     OSTCBCurPtr->FlagsPend = flags;               //保存需要等待的标志位
930     OSTCBCurPtr->FlagsOpt  = opt;                 //保存对标志位的要求
931     OSTCBCurPtr->FlagsRdy  = (OS_FLAGS)0;
932
933     OS_Pend(p_pend_data,                          //阻塞任务，等待事件标志组
934             (OS_PEND_OBJ *)((void *)p_grp),
935              OS_TASK_PEND_ON_FLAG,
936              timeout);
937 }
```

图 9-9 OS_FlagBlock () 函数

OS_FlagBlock () 函数会调用一个更加底层的等待函数来执行当前任务对事件标志组的等待，该函数就是 OS_Pend()。与 OS_Post() 函数一样，OS_Pend() 函数不仅仅用来等待事件标志组，还可以等待多值信号量、互斥信号量、消息队列、任务消息队列和任务信号量。OS_Pend() 函数的定义位于“os_core.c”。

```
873 void  OS_Pend (OS_PEND_DATA  *p_pend_data,   //待插入等待列表的元素
874              OS_PEND_OBJ   *p_obj,         //等待的内核对象
875              OS_STATE       pending_on,    //等待哪种对象内核
876              OS_TICK        timeout)       //等待期限
877 {
878     OS_PEND_LIST  *p_pend_list;
879
880
881
882     OSTCBCurPtr->PendOn     = pending_on;                    //资源不可用，开始等待
883     OSTCBCurPtr->PendStatus = OS_STATUS_PEND_OK;             //正常等待中
884
885     OS_TaskBlock(OSTCBCurPtr,                                //阻塞当前运行任务，
886                  timeout);                                   //如果 timeout 非0，把任务插入的节拍列表
887
888     if (p_obj != (OS_PEND_OBJ *)0) {                         //如果等待对象非空
889         p_pend_list             = &p_obj->PendList;          //获取对象的等待列表到 p_pend_list
890         p_pend_data->PendObjPtr = p_obj;                     //保存要等待的对象
891         OS_PendDataInit((OS_TCB       *)OSTCBCurPtr,         //初始化 p_pend_data（待插入等待列表）
892                         (OS_PEND_DATA *)p_pend_data,
893                         (OS_OBJ_QTY    )1);
894         OS_PendListInsertPrio(p_pend_list,                   //按优先级将 p_pend_data 插入到等待列表
895                               p_pend_data);
896     } else {                                                 //如果等待对象为空
897         OSTCBCurPtr->PendDataTblEntries = (OS_OBJ_QTY    )0; //清零当前任务的等待域数据
898         OSTCBCurPtr->PendDataTblPtr     = (OS_PEND_DATA *)0;
899     }
900 #if OS_CFG_DBG_EN > 0u                                       //如果使能了调试代码和变量
901     OS_PendDbgNameAdd(p_obj,                                 //更新信号量的 DbgNamePtr 元素为其等待
902                       OSTCBCurPtr);                          //列表中优先级最高的任务的名称。
903 #endif
904 }
```

图 9-10 OS_Pend() 函数

9.1.4 OSFlagPendAbort ()

OSFlagPendAbort() 函数用于中止任务对一个事件标志组的等待。要使用 OSFlagPendAbort () 函数，还得事先使能 OS_CFG_FLAG_PEND_ABORT_EN（位于“os_cfg.h”），如下图所示。

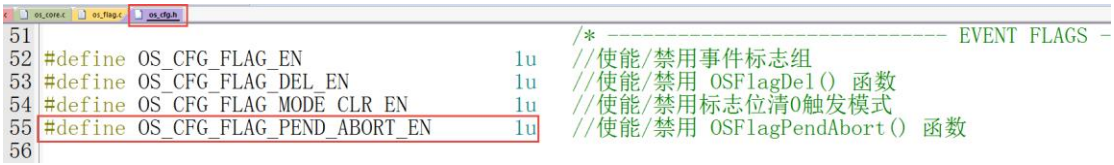

图 9-11

OSFlagPendAbort () 函数的信息如下表所示。

表 33 OSFlagPendAbort ()

函数原型	OS_OBJ_QTY OSFlagPendAbort (OS_FLAG_GRP *p_grp, OS_OPT opt, OS_ERR *p_err);			
功能	中止任务对一个事件标志组的等待。			
参数	p_grp	事件标志组指针。		
	opt	选项。	OS_OPT_PEND_ABORT_1	只中止该事件标志组等待列表中的最高优先级任务。
			OS_OPT_PEND_ABORT_ALL	中止该事件标志组等待列表中的所有优先级任务。
			OS_OPT_PEND_ABORT_1 | OS_OPT_POST_NO_SCHED	只中止该事件标志组等待列表中的最高优先级任务，但不进行任务调度。
			OS_OPT_PEND_ABORT_ALL | OS_OPT_POST_NO_SCHED	中止该事件标志组等待列表中的所有优先级任务，但不进行任务调度。
	p_err	返回错误类型	OS_ERR_NONE	无错误。
			OS_ERR_OBJ_PTR_NULL	p_grp 为空
			OS_ERR_OBJ_TYPE	p_grp 不是事件标志组类型。
			OS_ERR_OPT_INVALID	选项非法。
			OS_ERR_PEND_ABORT_ISR	该函数在中断中被调用。
			OS_ERR_PEND_ABORT_NONE	没有任务在等待该事件标志组。
返回值	✧ 0，没有任务在等待该事件标志组，或者有错误产生。 ✧ >0，被中止的任务数。			
注意事项	✧ 不可以在中断中调用该函数。			

OSFlagPendAbort () 函数的定义位于“os_flag.c”。

```
649 #if OS_CFG_FLAG_PEND_ABORT_EN > 0u                   //如果使能了 OSFlagPendAbort() 函数
650 OS_OBJ_QTY  OSFlagPendAbort (OS_FLAG_GRP  *p_grp, //事件标志组指针
651                              OS_OPT        opt,    //选项
652                              OS_ERR       *p_err) //返回错误类型
653 {
654     OS_PEND_LIST  *p_pend_list;
655     OS_TCB        *p_tcb;
656     CPU_TS         ts;
657     OS_OBJ_QTY     nbr_tasks;
658     CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必需该宏，该宏声明和
659                     //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
660                     // SR（临界段关中断只需保存SR），开中断时将该值还原。
661
662 #ifdef OS_SAFETY_CRITICAL                           //如果使能了安全检测
663     if (p_err == (OS_ERR *)0) {                     //如果错误类型实参为空
664         OS_SAFETY_CRITICAL_EXCEPTION();             //执行安全检测异常函数
665         return ((OS_OBJ_QTY)0u);                    //返回0（有错误），停止执行
666     }
667 #endif
668
669 #if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u              //如果使能了中断中非法调用检测
670     if (OSIntNestingCtr > (OS_NESTING_CTR)0u) {     //如果该函数是在中断中被调用
671         *p_err = OS_ERR_PEND_ABORT_ISR;             //错误类型为“在中断中创建对象”
672         return ((OS_OBJ_QTY)0u);                    //返回0（有错误），停止执行
673     }
674 #endif
675
676 #if OS_CFG_ARG_CHK_EN > 0u                          //如果使能了参数检测
677     if (p_grp == (OS_FLAG_GRP *)0) {                //如果 p_grp 为空
678         *p_err  =  OS_ERR_OBJ_PTR_NULL;             //错误类型为“创建对象为空”
679         return ((OS_OBJ_QTY)0u);                    //返回0（有错误），停止执行
680     }
681     switch (opt) {                                  //根据选项分类处理
682         case OS_OPT_PEND_ABORT_1:                   //如果选项在预期内
683         case OS_OPT_PEND_ABORT_ALL:
684         case OS_OPT_PEND_ABORT_1   | OS_OPT_POST_NO_SCHED:
685         case OS_OPT_PEND_ABORT_ALL | OS_OPT_POST_NO_SCHED:
686              break;                                 //直接跳出
687
688         default:                                    //如果选项超出预期
689             *p_err = OS_ERR_OPT_INVALID;            //错误类型为“选项非法”
690              return ((OS_OBJ_QTY)0u);               //返回0（有错误），停止执行
691     }
692 #endif
693
694 #if OS_CFG_OBJ_TYPE_CHK_EN > 0u                     //如果使能了对象类型检测
695     if (p_grp->Type != OS_OBJ_TYPE_FLAG) {          //如果 p_grp 不是事件标志组类型
696         *p_err = OS_ERR_OBJ_TYPE;                   //错误类型为“对象类型有误”
697         return ((OS_OBJ_QTY)0u);                    //返回0（有错误），停止执行
698     }
699 #endif
700
701     CPU_CRITICAL_ENTER();                           //关中断
702     p_pend_list = &p_grp->PendList;                 //获取消息队列的等待列表
703     if (p_pend_list->NbrEntries == (OS_OBJ_QTY)0u) { //如果没有任务在等待
704         CPU_CRITICAL_EXIT();                        //开中断
705        *p_err = OS_ERR_PEND_ABORT_NONE;             //错误类型为“没任务在等待”
706         return ((OS_OBJ_QTY)0u);                    //返回0（有错误），停止执行
707     }
```

```
708     /* 如果有任务在等待 */
709     OS_CRITICAL_ENTER_CPU_EXIT();                          //进入临界段，重开中断
710     nbr_tasks = 0u;                                        //准备计数被中止的等待任务
711     ts        = OS_TS_GET();                               //获取时间戳
712     while (p_pend_list->NbrEntries > (OS_OBJ_QTY)0u) {     //如果还有任务在等待
713         p_tcb = p_pend_list->HeadPtr->TCBPtr;              //获取头端（最高优先级）任务
714         OS_PendAbort((OS_PEND_OBJ *)((void *)p_grp),       //中止该任务对 p_grp 的等待
715                      p_tcb,
716                      ts);
717         nbr_tasks++;
718         if (opt != OS_OPT_PEND_ABORT_ALL) {                //如果不是选择了中止所有等待任务
719             break;                                         //立即跳出，不再继续中止
720         }
721     }
722     OS_CRITICAL_EXIT_NO_SCHED();                           //退出临界段（无调度）
723
724     if ((opt & OS_OPT_POST_NO_SCHED) == (OS_OPT)0u) {      //如果选择了任务调度
725         OSSched();                                         //进行任务调度
726     }
727
728    *p_err = OS_ERR_NONE;                                   //错误类型为“无错误”
729     return (nbr_tasks);                                    //返回被中止的任务数目
730 }
731 #endif
```

图 9-12 OSFlagPendAbort () 函数

OSFlagPendAbort () 函数会调用一个更加底层的中止等待函数来执行当前任务对事件标志组的等待，该函数就是 OS_PendAbort()。OS_PendAbort() 函数不仅仅用来中止对事件标志组的等待，还可以中止对多值信号量、互斥信号量、消息队列、任务消息队列或任务信号量的等待。OS_PendAbort() 函数的定义位于“os_core.c”。

os_cfg.h | os_sem.c | os.h | os_core.c

```
927 void  OS_PendAbort (OS_PEND_OBJ  *p_obj,    //被等待对象的类型
928                     OS_TCB        *p_tcb,    //任务控制块指针
929                     CPU_TS         ts)       //等待被中止时的时间戳
930 {
931     switch (p_tcb->TaskState) {                            //根据任务状态分类处理
932         case OS_TASK_STATE_RDY:                            //如果任务是就绪状态
933         case OS_TASK_STATE_DLY:                            //如果任务是延时状态
934         case OS_TASK_STATE_SUSPENDED:                      //如果任务是挂起状态
935         case OS_TASK_STATE_DLY_SUSPENDED:                  //如果任务是在延时中被挂起
936              break;                                        //这些情况均与等待无关，直接跳出
937
938         case OS_TASK_STATE_PEND:                           //如果任务是无期限等待状态
939         case OS_TASK_STATE_PEND_TIMEOUT:                   //如果任务是有期限等待状态
940              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
941                  OS_PendAbort1(p_obj,                      //强制解除任务对某一对象的等待
942                                p_tcb,
943                                ts);
944              }
945 #if (OS_MSG_EN > 0u)                                       //如果使能了任务队列或消息队列
946              p_tcb->MsgPtr     = (void       *)0;         //清除（复位）任务的消息域
947              p_tcb->MsgSize    = (OS_MSG_SIZE)0u;
948 #endif
949              p_tcb->TS         = ts;                       //保存等待被中止时的时间戳到任务控制块
950              if (p_obj != (OS_PEND_OBJ *)0) {              //如果等待对象非空
951                  OS_PendListRemove(p_tcb);                 //将任务从所有等待列表中移除
952              }
953              OS_TaskRdy(p_tcb);                            //让任务进准备运行
954              p_tcb->TaskState  = OS_TASK_STATE_RDY;        //修改任务状态为就绪状态
955              p_tcb->PendStatus = OS_STATUS_PEND_ABORT;     //标记任务的等待被中止
956              p_tcb->PendOn     = OS_TASK_PEND_ON_NOTHING;  //标记任务目前没有等待任何对象
957              break;                                        //跳出
958
```

```
959          case OS_TASK_STATE_PEND_SUSPENDED:                    //如果任务在无期限等待中被挂起
960          case OS_TASK_STATE_PEND_TIMEOUT_SUSPENDED:            //如果任务在有期限等待中被挂起
961               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
962                    OS_PendAbort1(p_obj,                        //强制解除任务对某一对象的等待
963                                  p_tcb,
964                                  ts);
965               }
966 #if (OS_MSG_EN > 0u)                                           //如果使能了任务队列或消息队列
967               p_tcb->MsgPtr     = (void      *)0;              //清除（复位）任务的消息域
968               p_tcb->MsgSize    = (OS_MSG_SIZE)0u;
969 #endif
970               p_tcb->TS         = ts;                         //保存等待被中止时的时间戳到任务控制块
971               if (p_obj != (OS_PEND_OBJ *)0) {                //如果等待对象非空
972                    OS_PendListRemove(p_tcb);                  //将任务从所有等待列表中移除
973               }
974               OS_TickListRemove(p_tcb);                       //让任务脱离节拍列表
975               p_tcb->TaskState  = OS_TASK_STATE_SUSPENDED;    //修改任务状态为挂起状态
976               p_tcb->PendStatus = OS_STATUS_PEND_ABORT;       //标记任务的等待被中止
977               p_tcb->PendOn     = OS_TASK_PEND_ON_NOTHING;    //标记任务目前没有等待任何对象
978               break;                                          //跳出
979
980          default:                                             //如果任务状态超出预期
981               break;                                          //不需处理，直接跳出
982     }
983 }
```

图 9-13 OS_PendAbort() 函数

9.1.5 OSFlagDel ()

OSFlagDel() 函数用于删除一个消息队列。要使用 OSFlagDel () 函数，还得事先使能 OS_CFG_FLAG_DEL_EN（位于“os_cfg.h”），如下图所示。

```
51                                                  /* ---------------------------- EVENT FLAGS -
52 #define OS_CFG_FLAG_EN                      1u   //使能/禁用事件标志组
53 #define OS_CFG_FLAG_DEL_EN                  1u   //使能/禁用 OSFlagDel() 函数
54 #define OS_CFG_FLAG_MODE_CLR_EN             1u   //使能/禁用标志位清0触发模式
55 #define OS_CFG_FLAG_PEND_ABORT_EN           1u   //使能/禁用 OSFlagPendAbort() 函数
56
```

图 9-14

OSFlagDel () 函数的信息如下表所示。

表 34 OSFlagDel ()

函数原型	OS_OBJ_QTY OSFlagDel (OS_FLAG_GRP *p_grp, OS_OPT opt, OS_ERR *p_err);			
功能	删除一个事件标志组。			
参数	p_q	事件标志组指针。		
	opt	选项	OS_OPT_DEL_NO_PEND	如果没有任务等待 p_grp，才删除 p_grp。
			OS_OPT_DEL_ALWAYS	必须删除 p_grp。
	p_err	返回错误类型	OS_ERR_NONE	无错误。
			OS_ERR_DEL_ISR	该函数在中断中被调用。
			OS_ERR_OBJ_PTR_NULL	p_grp 为空。
			OS_ERR_OBJ_TYPE	p_grp 不是事件标志组类型。
			OS_ERR_OPT_INVALID	选项非法。
			OS_ERR_TASK_WAITING	还有任务在等待该事件标志组。

返回值	✧ 0，没有任务在等待该事件标志组，或者有错误产生。 ✧ >0，事件标志组被删除前等待其的任务数。
注意事项	✧ 不可以在中断中调用该函数。

OSFlagDel () 函数的定义位于“os_flag.c”。

```
155 #if OS_CFG_FLAG_DEL_EN > 0u                          //如果使能了 OSFlagDel() 函数
156 OS_OBJ_QTY  OSFlagDel (OS_FLAG_GRP  *p_grp,  //事件标志组指针
157                        OS_OPT        opt,    //选项
158                        OS_ERR       *p_err) //返回错误类型
159 {
160     OS_OBJ_QTY         cnt;
161     OS_OBJ_QTY         nbr_tasks;
162     OS_PEND_DATA      *p_pend_data;
163     OS_PEND_LIST      *p_pend_list;
164     OS_TCB            *p_tcb;
165     CPU_TS             ts;
166     CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必需该宏，该宏声明和
167                     //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
168                     // SR（临界段关中断只需保存SR），开中断时将该值还原。
169
170 #ifdef OS_SAFETY_CRITICAL                           //如果使能（默认禁用）了安全检测
171     if (p_err == (OS_ERR *)0) {                     //如果错误类型实参为空
172         OS_SAFETY_CRITICAL_EXCEPTION(); //执行安全检测异常函数
173         return ((OS_OBJ_QTY)0);                     //返回0（有错误），停止执行
174     }
175 #endif
176
177 #if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u              //如果使能了中断中非法调用检测
178     if (OSIntNestingCtr > (OS_NESTING_CTR)0) { //如果该函数在中断中被调用
179         *p_err = OS_ERR_DEL_ISR;                    //错误类型为“在中断中删除对象”
180         return ((OS_OBJ_QTY)0);                     //返回0（有错误），停止执行
181     }
182 #endif
183
184 #if OS_CFG_ARG_CHK_EN > 0u                          //如果使能了参数检测
185     if (p_grp == (OS_FLAG_GRP *)0) {                //如果 p_grp 为空
186         *p_err  = OS_ERR_OBJ_PTR_NULL;              //错误类型为“对象为空”
187         return ((OS_OBJ_QTY)0);                     //返回0（有错误），停止执行
188     }
189     switch (opt) {                                  //根据选项分类处理
190         case OS_OPT_DEL_NO_PEND:                    //如果选项在预期内
191         case OS_OPT_DEL_ALWAYS:
192             break;                                  //直接跳出
193
194         default:                                    //如果选项超出预期
195             *p_err = OS_ERR_OPT_INVALID;            //错误类型为“选项非法”
196             return ((OS_OBJ_QTY)0);                 //返回0（有错误），停止执行
197     }
198 #endif
199
200 #if OS_CFG_OBJ_TYPE_CHK_EN > 0u                     //如果使能了对象类型检测
201     if (p_grp->Type != OS_OBJ_TYPE_FLAG) {//如果 p_grp 不是事件标志组类型
202         *p_err = OS_ERR_OBJ_TYPE;                   //错误类型为“对象类型有误”
203         return ((OS_OBJ_QTY)0);                     //返回0（有错误），停止执行
204     }
205 #endif
```

```
206     OS_CRITICAL_ENTER();                                    //进入临界段
207     p_pend_list = &p_grp->PendList;                         //获取消息队列的等待列表
208     cnt         = p_pend_list->NbrEntries;                  //获取等待该队列的任务数
209     nbr_tasks   = cnt;                                      //按照任务数目逐个处理
210     switch (opt) {                                          //根据选项分类处理
211         case OS_OPT_DEL_NO_PEND:                            //如果只在没任务等待时进行删除
212              if (nbr_tasks == (OS_OBJ_QTY)0) {              //如果没有任务在等待该标志组
213 #if OS_CFG_DBG_EN > 0u                                      //如果使能了调试代码和变量
214                  OS_FlagDbgListRemove(p_grp);               //将该标志组从标志组调试列表移除
215 #endif
216                  OSFlagQty--;                               //标志组数目减1
217                  OS_FlagClr(p_grp);                         //清除该标志组的内容
218
219                  OS_CRITICAL_EXIT();                        //退出临界段
220                 *p_err = OS_ERR_NONE;                       //错误类型为“无错误”
221              } else {
222                  OS_CRITICAL_EXIT();                        //退出临界段
223                 *p_err = OS_ERR_TASK_WAITING;               //错误类型为“有任务在等待标志组”
224              }
225              break;                                         //跳出
226

227         case OS_OPT_DEL_ALWAYS:                             //如果必须删除标志组
228              ts = OS_TS_GET();                              //获取时间戳
229              while (cnt > 0u) {                             //逐个移除该标志组等待列表中的任务
230                  p_pend_data = p_pend_list->HeadPtr;
231                  p_tcb       = p_pend_data->TCBPtr;
232                  OS_PendObjDel((OS_PEND_OBJ *)((void *)p_grp),
233                                p_tcb,
234                                ts);
235                  cnt--;
236              }
237 #if OS_CFG_DBG_EN > 0u                                      //如果使能了调试代码和变量
238              OS_FlagDbgListRemove(p_grp);                   //将该标志组从标志组调试列表移除
239 #endif
240              OSFlagQty--;                                   //标志组数目减1
241              OS_FlagClr(p_grp);                             //清除该标志组的内容
242              OS_CRITICAL_EXIT_NO_SCHED();                   //退出临界段（无调度）
243              OSSched();                                     //调度任务
244             *p_err = OS_ERR_NONE;                           //错误类型为“无错误”
245              break;                                         //跳出
246

247         default:                                            //如果选项超出预期
248              OS_CRITICAL_EXIT();                            //退出临界段
249             *p_err = OS_ERR_OPT_INVALID;                    //错误类型为“选项非法”
250              break;                                         //跳出
251     }
252     return (nbr_tasks);                                     //返回删除标志组前等待其的任务数
253 }
254 #endif
```

图 9-15 OSFlagDel () 函数

OSFlagDel () 函数会调用一个更加底层的删除等待对象的函数来执行对事件标志组的删除，该函数就是 OS_PendObjDel ()。OS_PendObjDel () 函数不仅仅用来删除事件标志组，还可以删除多值信号量、互斥信号量、消息队列、任务消息队列或任务信号量。OS_PendObjDel () 函数的定义位于“os_core.c”。

os_cfg_app.c | os_sem.c | os_core.c | os_cfg.h

```
1693 void  OS_PendObjDel (OS_PEND_OBJ  *p_obj,   //被删除对象的类型
1694                     OS_TCB       *p_tcb,   //任务控制块指针
1695                     CPU_TS        ts)      //信号量被删除时的时间戳
1696 {
1697     switch (p_tcb->TaskState) {                                 //根据任务状态分类处理
1698         case OS_TASK_STATE_RDY:                                 //如果任务是就绪状态
1699         case OS_TASK_STATE_DLY:                                 //如果任务是延时状态
1700         case OS_TASK_STATE_SUSPENDED:                           //如果任务是挂起状态
1701         case OS_TASK_STATE_DLY_SUSPENDED:                       //如果任务是在延时中被挂起
1702              break;                                             //这些情况均与等待无关，直接跳出
1703
```

```
1704            case OS_TASK_STATE_PEND:                                    //如果任务是无期限等待状态
1705            case OS_TASK_STATE_PEND_TIMEOUT:                            //如果任务是有期限等待状态
1706                 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
1707                     OS_PendObjDel1(p_obj,                              //强制解除任务对某一对象的等待
1708                                    p_tcb,
1709                                    ts);
1710                 }
1711#if (OS_MSG_EN > 0u)                                   //如果使能了任务队列或消息队列
1712                 p_tcb->MsgPtr     = (void *)0;          //清除（复位）任务的消息域
1713                 p_tcb->MsgSize    = (OS_MSG_SIZE)0u;
1714#endif
1715                 p_tcb->TS         = ts;                                 //保存等待被中止时的时间戳到任务控制块
1716                 OS_PendListRemove(p_tcb);                               //将任务从所有等待列表中移除
1717                 OS_TaskRdy(p_tcb);                                      //让任务进准备运行
1718                 p_tcb->TaskState  = OS_TASK_STATE_RDY;                  //修改任务状态为就绪状态
1719                 p_tcb->PendStatus = OS_STATUS_PEND_DEL;                 //标记任务的等待对象被删除
1720                 p_tcb->PendOn     = OS_TASK_PEND_ON_NOTHING;            //标记任务目前没有等待任何对象
1721                 break;                                                  //跳出
1722
1723            case OS_TASK_STATE_PEND_SUSPENDED:                          //如果任务在无期限等待中被挂起
1724            case OS_TASK_STATE_PEND_TIMEOUT_SUSPENDED:                  //如果任务在有期限等待中被挂起
1725                 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
1726                     OS_PendObjDel1(p_obj,                              //强制解除任务对某一对象的等待
1727                                    p_tcb,
1728                                    ts);
1729                 }
1730#if (OS_MSG_EN > 0u)                                   //如果使能了任务队列或消息队列
1731                 p_tcb->MsgPtr     = (void      *)0;  //清除（复位）任务的消息域
1732                 p_tcb->MsgSize    = (OS_MSG_SIZE)0u;
1733#endif
1734                 p_tcb->TS         = ts;                                 //保存等待被中止时的时间戳到任务控制块
1735                 OS_TickListRemove(p_tcb);                               //让任务脱离节拍列表
1736                 OS_PendListRemove(p_tcb);                               //将任务从所有等待列表中移除
1737                 p_tcb->TaskState  = OS_TASK_STATE_SUSPENDED;            //修改任务状态为挂起状态
1738                 p_tcb->PendStatus = OS_STATUS_PEND_DEL;                 //标记任务的等待对象被删除
1739                 p_tcb->PendOn     = OS_TASK_PEND_ON_NOTHING;            //标记任务目前没有等待任何对象
1740                 break;                                                  //跳出
1741
1742            default:                                                     //如果任务状态超出预期
1743                 break;                                                  //不需处理，直接跳出
1744        }
1745}
```

图 9-16 OS_PendObjDel () 函数

9.2 实例演示

9.2.1 实例 1

本节实例使用事件标志组制作一个按键组合。当 KEY1 被按下时，LED1 亮；当 KEY1 被释放时，LED1 灭。当 KEY2 被按下时，LED2 亮；当 KEY2 被释放时，LED2 灭。当 KEY1 和 KEY2 均被按下时，LED3 才亮，如果有按键被释放，LED3 就熄灭。

该例程已经存放在配套资料的下图路径。

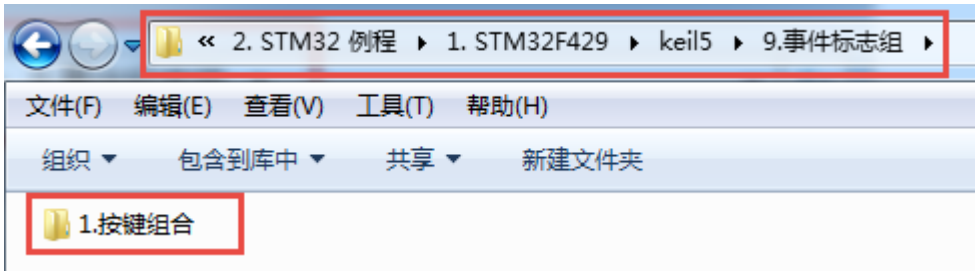

图 9-17 例程路径

该实例创建两个应用任务，AppTaskPost ()和 AppTaskPend ()任务。AppTaskPost () 用于采集按键信息和发布事件标志组：当 KEY1 被按下时，点亮 LED1；当 KEY1 被释放时，熄灭 LED1。当 KEY2 被按下时，点亮 LED2；当 KEY2 被释放时，熄灭 LED2。任务 AppTaskPend () 用于接收（等待）事件标志组：先等待 KEY1 和 KEY2 均被按下，点亮 LED3；然后再等待 KEY1 或 KEY2 被释放，熄灭 LED3；依次循环。

本例程需用使用按键和 LED，所以工程中需要添加其驱动文件和初始化。用户可参照前面相关实例，这里不再赘述。

在起始任务 AppTaskStart() 中，在创建应用任务之前，创建了应用任务需要使用到的事件标志组 flag_grp（必须保证该事件标志组在被使用到之前创建好）。

```
142 static  void  AppTaskStart (void *p_arg)
143 {
144     CPU_INT32U  cpu_clk_freq;
145     CPU_INT32U  cnts;
146     OS_ERR      err;
147
148
149     (void)p_arg;
150
151     BSP_Init();                                                 //板级初始化
152     CPU_Init();                                                 //初始化 CPU
153
154     cpu_clk_freq = BSP_CPU_ClkFreq();                           //获取 CPU 内
155     cnts = cpu_clk_freq / (CPU_INT32U)OSCfg_TickRate_Hz;        //根据用户设
156     OS_CPU_SysTickInit(cnts);                                   //调用 SysTic
157
158     Mem_Init();                                                 //初始化内存
159
160 #if OS_CFG_STAT_TASK_EN > 0u                                    //如果使能（
161     OSStatTaskCPUUsageInit(&err);                               //计算没有应
162 #endif                                                          //容量（决定
163                                                                 //使用率使用
164     CPU_IntDisMeasMaxCurReset();                                //复位（清零
165
166
167   /* 创建事件标志组 flag_grp */
168   OSFlagCreate ((OS_FLAG_GRP  *)&flag_grp,          //指向事件标志组的指针
169                 (CPU_CHAR     *)"FLAG For Test",    //事件标志组的名字
170                 (OS_FLAGS      )0,                  //事件标志组的初始值
171                 (OS_ERR       *)&err);              //返回错误类型
172
```

图 9-18 创建事件标志组

任务函数 AppTaskPost () 的定义如下。

```
211 ************************************************************************************************************
212 *                                              POST TASK
213 ************************************************************************************************************
214 */
215 static  void  AppTaskPost ( void * p_arg )
216 {
217   OS_ERR      err;
218 
219 
220   (void)p_arg;
221 
222 
223   while (DEF_TRUE) {                                                   //任务体
224     if( Key_ReadStatus ( macKEY1_GPIO_PORT, macKEY1_GPIO_PIN, 1 ) == 1 ) //如果KEY1被按下
225     {
226       macLED1_ON ();                                                   //点亮LED1
227 
228       OSFlagPost ((OS_FLAG_GRP  *)&flag_grp,                           //将标志组的BIT0置1
229                   (OS_FLAGS      )0x01,
230                   (OS_OPT        )OS_OPT_POST_FLAG_SET,
231                   (OS_ERR       *)&err);
232 
233     }
234     else                                                               //如果KEY1被释放
235     {
236       macLED1_OFF ();                                                  //熄灭LED1
237 
238       OSFlagPost ((OS_FLAG_GRP  *)&flag_grp,                           //将标志组的BIT0清0
239                   (OS_FLAGS      )0x01,
240                   (OS_OPT        )OS_OPT_POST_FLAG_CLR,
241                   (OS_ERR       *)&err);
242 
243     }


244 
245     if( Key_ReadStatus ( macKEY2_GPIO_PORT, macKEY2_GPIO_PIN, 0 ) == 1 ) //如果KEY2被按下
246     {
247       macLED2_ON ();                                                   //点亮LED2
248 
249       OSFlagPost ((OS_FLAG_GRP  *)&flag_grp,                           //将标志组的BIT1置1
250                   (OS_FLAGS      )0x02,
251                   (OS_OPT        )OS_OPT_POST_FLAG_SET,
252                   (OS_ERR       *)&err);
253 
254     }
255     else                                                               //如果KEY2被释放
256     {
257       macLED2_OFF ();                                                  //熄灭LED2
258 
259       OSFlagPost ((OS_FLAG_GRP  *)&flag_grp,                           //将标志组的BIT1清0
260                   (OS_FLAGS      )0x02,
261                   (OS_OPT        )OS_OPT_POST_FLAG_CLR,
262                   (OS_ERR       *)&err);
263 
264     }
265 
266     OSTimeDlyHMSM ( 0, 0, 0, 20, OS_OPT_TIME_DLY, & err );  //每20ms扫描一次
267 
268   }
269 
270 }
```

图 9-19 AppTaskPost () 任务函数

任务函数 AppTaskPend () 的定义如下。

```
app_cfg.h  app.c  os_core.c  os_flag.c
278 static  void  AppTaskPend ( void * p_arg )
279 {
280   OS_ERR      err;
281
282
283   (void)p_arg;
284
285
286   while (DEF_TRUE) {                                          //任务体
287     OSFlagPend ((OS_FLAG_GRP *)&flag_grp,                     //等待标志组的的BIT0和BIT1均被置1
288                 (OS_FLAGS     )( 0X01 | 0X02 ),
289                 (OS_TICK      )0,
290                 (OS_OPT       )OS_OPT_PEND_FLAG_SET_ALL |
291                                OS_OPT_PEND_BLOCKING,
292                 (CPU_TS       *)0,
293                 (OS_ERR       *)&err);
294
295     macLED3_ON ();                                            //点亮LED3
296
297     OSFlagPend ((OS_FLAG_GRP *)&flag_grp,                     //等待标志组的的BIT0和BIT1有一个被清0
298                 (OS_FLAGS     )( 0X01 | 0X02 ),
299                 (OS_TICK      )0,
300                 (OS_OPT       )OS_OPT_PEND_FLAG_CLR_ANY |
301                                OS_OPT_PEND_BLOCKING,
302                 (CPU_TS       *)0,
303                 (OS_ERR       *)&err);
304
305     macLED3_OFF ();                                           //熄灭LED3
306
307   }
308
309 }
```

图 9-20 AppTaskPend () 任务函数

编译和下载程序到秉火 STM32 开发板，运行程序。用户通过操作可以发现：当按下 KEY1 时，LED1 亮起，当释放 KEY1 时，LED1 熄灭；当按下 KEY2 时，LED2 亮起，当释放 KEY2 时，LED2 熄灭；当 KEY1 和 KEY2 均被按下时，LED3 亮起，当 KEY1 和 KEY2 有一个被释放时，LED3 就熄灭。

9.3 章末总结

事件标志组，顾名思义，就是用于集合多个事件的是否发生。

使用事件标志组之前必须先创建它，创建事件标志组使用 OSFlagCreate () 函数。

OSFlagPend () 函数用于等待事件标志组，如果事件标志组不符合等待选项的要求，可以选择等待或者不等待。与之相对应，OSFlagPost () 函数则用于发布事件标志组。

OSFlagPendAbort () 函数用于中止任务对一个事件标志组的等待。OSFlagDel() 函数用于删除一个事件标志组。

第10章 等待多个内核对象

前面所讲解的等待内核对象，均是等待一个内核对象，这一章要讲解的是同时等待多个内核对象。这里的多个内核对象是指多值信号量和消息队列的任意组合。

10.1 原理简述

如果想要使用“等待多个内核对象”，就必须事先使能“等待多个内核对象”。“等待多个内核对象”的使能位于“os_cfg.h”。

图 10-1

另外，值得注意，等待多个内核对象的内核对象指的是多值信号量或消息队列，要等待这两种对象，均须先使能它们，分别为 OS_CFG_SEM_EN 和 OS_CFG_Q_EN，均位于“os_cfg.h”，详情可参考前面相关章节。

10.1.1 OSPendMulti ()

OSPendMulti () 函数用于等待多个内核对象（多值信号量或消息队列）。OSPendMulti () 函数的信息如下表所示。

表 35 OSPendMulti ()

函数原型	OS_OBJ_QTY OSPendMulti (OS_PEND_DATA *p_pend_data_tbl, OS_OBJ_QTY tbl_size, OS_TICK timeout, OS_OPT opt, OS_ERR *p_err);	
功能	等待多个内核对象（多值信号量或消息队列）。	
参	p_pend_data_tbl	要等待的内核对象，等待两个或以上对象时一般为数组。

<table>
<tr><td rowspan="11">数</td><td>tbl_size</td><td colspan="3">等待对象的数目。</td></tr>
<tr><td>timeout</td><td colspan="3">等待超时时间（单位：时钟节拍），0 代表无期限等待。opt 为 OS_OPT_PEND_BLOCKING 时该参数才起作用。</td></tr>
<tr><td rowspan="2">opt</td><td>OS_OPT_PEND_BLOCKING</td><td colspan="2">如果目前没有对象已被发布，阻塞任务，等待对象被发布。</td></tr>
<tr><td>OS_OPT_PEND_NON_BLOCKING</td><td colspan="2">如果目前没有对象已被发布，不阻塞任务。</td></tr>
<tr><td rowspan="9">p_err</td><td rowspan="9">返回错误类型</td><td>OS_ERR_NONE</td><td>无错误，至少等到一个对象，通过查看等待对象的.RdyObjPtr 就可以知道该对象是否被发布。</td></tr>
<tr><td>OS_ERR_OBJ_TYPE</td><td>至少有一个等待对象不是多值信号量或消息队列类型。</td></tr>
<tr><td>OS_ERR_OPT_INVALID</td><td>选项非法。</td></tr>
<tr><td>OS_ERR_PEND_ABORT</td><td>对某对象等待被中止，通过查看等待对象的.RdyObjPtr 就可以知道该对象。</td></tr>
<tr><td>OS_ERR_PEND_DEL</td><td>某对象被删除，通过查看等待对象的.RdyObjPtr 就可以知道该对象。</td></tr>
<tr><td>OS_ERR_PEND_ISR</td><td>在中断中调用该函数。</td></tr>
<tr><td>OS_ERR_PEND_LOCKED</td><td>调度器被锁。</td></tr>
<tr><td>OS_ERR_PEND_WOULD_BLOCK</td><td>没有对象被发布，但又没选择阻塞任务。</td></tr>
<tr><td>OS_ERR_STATUS_INVALID</td><td>任务等待状态非法。</td></tr>
<tr><td>OS_ERR_PTR_INVALID</td><td>p_pend_data_tbl 为空。</td></tr>
<tr><td>OS_ERR_TIMEOUT</td><td>等待超时。</td></tr>
<tr><td>返回值</td><td colspan="3">✧ > 0，被发布对象的数目，或者被中止等待的对象的数目，或者被删除的对象的数目，具体对应哪个数目要根据返回错误类型判断。
✧ == 0，等待超时，或者有错误。</td></tr>
<tr><td>注意事项</td><td colspan="3">✧ 等待对象只能是多值信号量或者消息队列。
✧ 不可以在中断中调用该函数。</td></tr>
</table>

OSPendMulti () 函数的定义位于“os_pend_multi.c”。

```
121 OS_OBJ_QTY  OSPendMulti (OS_PEND_DATA  *p_pend_data_tbl,  //等待对象（数组）
122                          OS_OBJ_QTY     tbl_size,         //等待对象的数目
123                          OS_TICK        timeout,          //超时（单位：时钟借配）
124                          OS_OPT         opt,              //选项
125                          OS_ERR        *p_err)            //返回错误类型
126 {
127     CPU_BOOLEAN   valid;
128     OS_OBJ_QTY    nbr_obj_rdy;
129     CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必需该宏，该宏声明和定义一个局部变
130                     //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）
131                     //，开中断时将该值还原。
132
133 #ifdef OS_SAFETY_CRITICAL                 //如果使能（默认禁用）了安全检测
134     if (p_err == (OS_ERR *)0) {           //如果错误类型实参为空
135         OS_SAFETY_CRITICAL_EXCEPTION(); //执行安全检测异常函数
136         return ((OS_OBJ_QTY)0);           //返回0（有错误），停止执行
137     }
138 #endif
139
140 #if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u          //如果使能（默认使能）了中断中非法调用检测
141     if (OSIntNestingCtr > (OS_NESTING_CTR)0) { //如果该函数是在中断中被调用
142        *p_err = OS_ERR_PEND_ISR;                //错误类型为“在中断函数中定时”
143         return ((OS_OBJ_QTY)0);                 //返回0（有错误），停止执行
144     }
145 #endif


146
147 #if OS_CFG_ARG_CHK_EN > 0u                                  //如果使能（默认使能）了参数检测
148     if (p_pend_data_tbl == (OS_PEND_DATA *)0) {             //如果参数 p_pend_data_tbl 为空
149        *p_err = OS_ERR_PTR_INVALID;                         //错误类型为“等待对象不可用”
150         return ((OS_OBJ_QTY)0);                             //返回0（有错误），停止执行
151     }
152     if (tbl_size == (OS_OBJ_QTY)0) {                        //如果 tbl_size 为0
153        *p_err = OS_ERR_PTR_INVALID;                         //错误类型为“等待对象不可用”
154         return ((OS_OBJ_QTY)0);                             //返回0（有错误），停止执行
155     }
156     switch (opt) {                                          //根据选项分类处理
157         case OS_OPT_PEND_BLOCKING:                          //如果选项在预期内
158         case OS_OPT_PEND_NON_BLOCKING:
159              break;                                         //直接跳出
160
161         default:                                            //如果选项超出预期
162             *p_err = OS_ERR_OPT_INVALID;                    //错误类型为“选项非法”
163              return ((OS_OBJ_QTY)0);                        //返回0（有错误），停止执行
164     }
165 #endif


166     /* 证实等待对象是否只有多值信号量或消息队列 */
167     valid = OS_PendMultiValidate(p_pend_data_tbl,
168                                  tbl_size);
169     if (valid == DEF_FALSE) {                   //如果等待对象不是只有多值信号量或消息队列
170        *p_err = OS_ERR_OBJ_TYPE;                //错误类型为“对象类型有误”
171         return ((OS_OBJ_QTY)0);                 //返回0（有错误），停止执行
172     }


173     /* 如果等待对象确实只有多值信号量或消息队列 */
174 /*$PAGE*/FF
175     CPU_CRITICAL_ENTER();                                       //关中断
176     nbr_obj_rdy = OS_PendMultiGetRdy(p_pend_data_tbl,           //查看是否有对象被提交了
177                                      tbl_size);
178     if (nbr_obj_rdy > (OS_OBJ_QTY)0) {                          //如果有对象被发布
179         CPU_CRITICAL_EXIT();                                    //开中断
180        *p_err = OS_ERR_NONE;                                    //错误类型为“无错误”
181         return ((OS_OBJ_QTY)nbr_obj_rdy);                       //返回被发布的等待对象的数目
182     }
```

```
    /* 如果目前没有对象被发布 */
    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“渴望阻塞”
        return ((OS_OBJ_QTY)0);                           //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();                          //开中断
           *p_err = OS_ERR_SCHED_LOCKED;                  //错误类型为“调度器被锁”
            return ((OS_OBJ_QTY)0);                       //返回0（有错误），停止执行
        }
    }

    OS_CRITICAL_ENTER_CPU_EXIT();                         //锁调度器，重开中断

    OS_PendMultiWait(p_pend_data_tbl,                     //挂起当前任务等到有对象被发布或超时
                     tbl_size,
                     timeout);

    OS_CRITICAL_EXIT_NO_SCHED();                          //解锁调度器（无调度）

    OSSched();                                            //调度任务
```

```
    /* 任务等到了（一个）对象后得以继续运行 */
    CPU_CRITICAL_ENTER();                                 //关中断
    switch (OSTCBCurPtr->PendStatus) {                    //根据当前任务的等待状态分类处理
        case OS_STATUS_PEND_OK:                           //如果任务已经等到了对象
            *p_err = OS_ERR_NONE;                         //错误类型为“无错误”
             break;                                       //跳出

        case OS_STATUS_PEND_ABORT:                        //如果任务的等待被中止
            *p_err = OS_ERR_PEND_ABORT;                   //错误类型为“等待对象被中止”
             break;                                       //跳出

        case OS_STATUS_PEND_TIMEOUT:                      //如果等待超时
            *p_err = OS_ERR_TIMEOUT;                      //错误类型为“等待超时”
             break;                                       //跳出

        case OS_STATUS_PEND_DEL:                          //如果任务等待的对象被删除
            *p_err = OS_ERR_OBJ_DEL;                      //错误类型为“等待对象被删”
             break;                                       //跳出

        default:                                          //如果任务的等待状态超出预期
            *p_err = OS_ERR_STATUS_INVALID;               //错误类型为“状态非法”
             break;                                       //跳出
    }

    OSTCBCurPtr->PendStatus = OS_STATUS_PEND_OK;          //复位任务的等待状态
    CPU_CRITICAL_EXIT();                                  //开中断

    return ((OS_OBJ_QTY)1);                               //返回被发布对象数目为1
}
```

图 10-2 OSPendMulti () 函数

OSPendMulti () 函数中，会调用 OS_PendMultiValidate () 函数验证等待对象是否均属于多值信号量或消息队列，如果不是，就返回，不继续执行等待。OS_PendMultiValidate () 函数的定义位于“os_pend_multi.c”。

```
335 CPU_BOOLEAN  OS_PendMultiValidate (OS_PEND_DATA  *p_pend_data_tbl, //等待对象
336                                    OS_OBJ_QTY     tbl_size)         //等待对象数目
337 {
338     OS_OBJ_QTY  i;
339     OS_OBJ_QTY  ctr;
340 #if OS_CFG_SEM_EN  > 0u
341     OS_SEM      *p_sem;
342 #endif
343 #if OS_CFG_Q_EN > 0u
344     OS_Q        *p_q;
345 #endif
346
347
348     for (i = 0u; i < tbl_size; i++) {                                  //逐个判断等待对象
349         if (p_pend_data_tbl->PendObjPtr == (OS_PEND_OBJ *)0) {         //如果等待对象不存在
350             return (DEF_FALSE);                                        //返回，验证失败
351         }
352
353         ctr = 0u;                                                      //置0 ctr
354 #if OS_CFG_SEM_EN  > 0u                                                //如果使能了多值信号量
355         p_sem = (OS_SEM *)((void *)p_pend_data_tbl->PendObjPtr);       //获取等待对象
356         if (p_sem->Type == OS_OBJ_TYPE_SEM) {                          //如果该对象是多值信号量类型
357             ctr++;                                                     //ctr 为1
358         }
359 #endif

360
361 #if OS_CFG_Q_EN > 0u                                                   //如果使能了消息队列
362         p_q = (OS_Q *)((void *)p_pend_data_tbl->PendObjPtr);           //获取等待对象
363         if (p_q->Type == OS_OBJ_TYPE_Q) {                              //如果该对象是消息队列类型
364             ctr++;                                                     //ctr 为1
365         }
366 #endif
367
368         if (ctr == (OS_OBJ_QTY)0) {                                    //如果 ctr = 0
369             return (DEF_FALSE);                                        //返回，验证失败
370         }
371         p_pend_data_tbl++;                                             //验证下一个等待对象
372     }
373     return (DEF_TRUE);                                                 //返回，验证成功
374 }
```

图 10-3 OS_PendMultiValidate () 函数

在 OSPendMulti () 函数中，还会调用 OS_PendMultiGetRdy () 函数查看是否有等待对象已被发布可供立即使用。OS_PendMultiGetRdy () 函数的定义位于“os_pend_multi.c”。

```
254 OS_OBJ_QTY  OS_PendMultiGetRdy (OS_PEND_DATA  *p_pend_data_tbl, //等待对象
255                                 OS_OBJ_QTY     tbl_size)         //等待对象数目
256 {
257     OS_OBJ_QTY   i;
258     OS_OBJ_QTY   nbr_obj_rdy;
259 #if OS_CFG_Q_EN > 0u
260     OS_ERR       err;
261     OS_MSG_SIZE  msg_size;
262     OS_Q        *p_q;
263     void        *p_void;
264     CPU_TS       ts;
265 #endif
266 #if OS_CFG_SEM_EN  > 0u
267     OS_SEM      *p_sem;
268 #endif
269
270
271
```

```
272     nbr_obj_rdy = (OS_OBJ_QTY)0;
273     for (i = 0u; i < tbl_size; i++) {                           //逐个检测等待对象
274         p_pend_data_tbl->RdyObjPtr  = (OS_PEND_OBJ  *)0;        //清零等待对象的所有数据
275         p_pend_data_tbl->RdyMsgPtr  = (void         *)0;
276         p_pend_data_tbl->RdyMsgSize = (OS_MSG_SIZE   )0;
277         p_pend_data_tbl->RdyTS      = (CPU_TS        )0;
278         p_pend_data_tbl->NextPtr    = (OS_PEND_DATA *)0;
279         p_pend_data_tbl->PrevPtr    = (OS_PEND_DATA *)0;
280         p_pend_data_tbl->TCBPtr     = (OS_TCB       *)0;
281 #if OS_CFG_Q_EN > 0u                                            //如果使能了消息队列
282         p_q = (OS_Q *)((void *)p_pend_data_tbl->PendObjPtr);    //将该对象视为消息队列处理
283         if (p_q->Type == OS_OBJ_TYPE_Q) {                       //如果该对象确为消息队列类型
284             p_void = OS_MsgQGet(&p_q->MsgQ,                     //从该消息队列获取消息
285                                 &msg_size,
286                                 &ts,
287                                 &err);
288             if (err == OS_ERR_NONE) {                           //如果获取消息成功
289                 p_pend_data_tbl->RdyObjPtr  = p_pend_data_tbl->PendObjPtr;
290                 p_pend_data_tbl->RdyMsgPtr  = p_void;           //保存接收到的消息
291                 p_pend_data_tbl->RdyMsgSize = msg_size;
292                 p_pend_data_tbl->RdyTS      = ts;
293                 nbr_obj_rdy++;
294             }
295         }
296 #endif

297
298 #if OS_CFG_SEM_EN > 0u                                          //如果使能了多值信号量
299         p_sem = (OS_SEM *)((void *)p_pend_data_tbl->PendObjPtr); //将该对象视为多值信号量处理
300         if (p_sem->Type == OS_OBJ_TYPE_SEM) {                   //如果该对象确为多值信号量类型
301             if (p_sem->Ctr > 0u) {                              //如果该信号量可用
302                 p_sem->Ctr--;                                   //等待任务可以获得信号量
303                 p_pend_data_tbl->RdyObjPtr  = p_pend_data_tbl->PendObjPtr;
304                 p_pend_data_tbl->RdyTS      = p_sem->TS;
305                 nbr_obj_rdy++;
306             }
307         }
308 #endif
309
310         p_pend_data_tbl++;                                      //检测下一个等待对象
311     }
312     return (nbr_obj_rdy);                                       //返回被发布的对象的数目
313 }
```

图 10-4 OS_PendMultiGetRdy () 函数

如果 OS_PendMultiGetRdy () 函数发现已有等待对象可用，OSPendMulti () 函数就会返回，继续运行任务。如果发现没有可用等待对象，就会继续调用 OS_PendMultiWait() 函数阻塞当前运行任务，等待内核对象。OS_PendMultiWait() 函数的定义也位于“os_pend_multi.c”。

```
397 void  OS_PendMultiWait (OS_PEND_DATA  *p_pend_data_tbl, //等待对象
398                         OS_OBJ_QTY     tbl_size,        //等待对象数目
399                         OS_TICK        timeout)         //超时（单位：时钟节拍）
400 {
401     OS_OBJ_QTY      i;
402     OS_PEND_LIST   *p_pend_list;
403
404 #if OS_CFG_Q_EN > 0u
405     OS_Q           *p_q;
406 #endif
407
408 #if OS_CFG_SEM_EN > 0u
409     OS_SEM         *p_sem;
410 #endif
411
412
413
414     OSTCBCurPtr->PendOn             = OS_TASK_PEND_ON_MULTI;    //等待对象不可用，开始等待
415     OSTCBCurPtr->PendStatus         = OS_STATUS_PEND_OK;
416     OSTCBCurPtr->PendDataTblEntries = tbl_size;
417     OSTCBCurPtr->PendDataTblPtr     = p_pend_data_tbl;
418
419     OS_TaskBlock(OSTCBCurPtr,                                   //阻塞当前运行任务
420                  timeout);
421
```

```
422     for (i = 0u; i < tbl_size; i++) {                           //逐个将等待对象插入等待列表
423         p_pend_data_tbl->TCBPtr = OSTCBCurPtr;                  //所有的等待对象都绑定当前任务
424
425 #if OS_CFG_SEM_EN > 0u                                          //如果使能了多值信号量
426         p_sem = (OS_SEM *)((void *)p_pend_data_tbl->PendObjPtr);//将对象视为多值信号量处理
427         if (p_sem->Type == OS_OBJ_TYPE_SEM) {                   //如果该对象确为多值信号量
428             p_pend_list = &p_sem->PendList;                     //获取该信号量的等待列表
429             OS_PendListInsertPrio(p_pend_list,                  //将当前任务插入该等待列表
430                                   p_pend_data_tbl);
431         }
432 #endif
433
434 #if OS_CFG_Q_EN > 0u                                            //如果使能了消息队列
435         p_q = (OS_Q *)((void *)p_pend_data_tbl->PendObjPtr);    //将对象视为消息队列处理
436         if (p_q->Type == OS_OBJ_TYPE_Q) {                       //如果该对象确为消息队列
437             p_pend_list = &p_q->PendList;                       //获取该消息队列的等待列表
438             OS_PendListInsertPrio(p_pend_list,                  //将当前任务插入该等待列表
439                                   p_pend_data_tbl);
440         }
441 #endif
442
443         p_pend_data_tbl++;                                      //处理下一个等待对象
444     }
445 }
```

```
360
361 #if OS_CFG_Q_EN > 0u                                            //如果使能了消息队列
362         p_q = (OS_Q *)((void *)p_pend_data_tbl->PendObjPtr);    //获取等待对象
363         if (p_q->Type == OS_OBJ_TYPE_Q) {                       //如果该对象是消息队列类型
364             ctr++;                                              //ctr 为1
365         }
366 #endif
367
368         if (ctr == (OS_OBJ_QTY)0) {                             //如果 ctr = 0
369             return (DEF_FALSE);                                 //返回，验证失败
370         }
371         p_pend_data_tbl++;                                      //验证下一个等待对象
372     }
373     return (DEF_TRUE);                                          //返回，验证成功
374 }
```

图 10-5 OS_PendMultiWait() 函数

10.2 实例演示

10.2.1 实例 1

本节实例创建两个应用任务，AppTaskPost ()和 AppTaskPend ()。任务 AppTaskPost () 每隔 1s 向消息队列发送一次消息，并创建一个软件定时器，每隔 100ms 扫描一次按键，如果 KEY1 被单击，就发布多值信号量，标志 KEY1 被单击。任务 AppTaskPend () 用于等待消息队列和多值信号量，如果有消息或信号量可用，就向串口调试助手打印相关信息。

该例程已经存放在配套资料的下图路径。

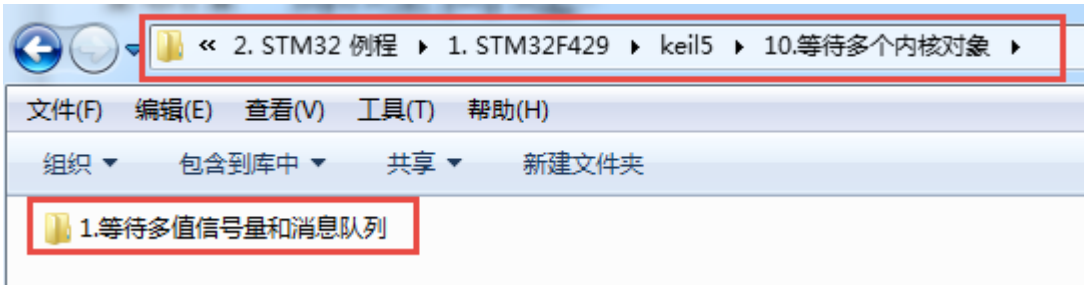

图 10-6 例程路径

本例程需用使用 USART1 和按键，所以工程中需要添加其驱动文件和初始化。用户可参照前面相关实例，这里不再赘述。

在起始任务 AppTaskStart() 中，在创建应用任务之前，创建了应用任务需要使用到的多值信号量 sem 和消息队列 queue（必须保证该多值信号量和消息队列在被使用到之前已被创建）。

```
166     CPU_IntDisMeasMaxCurReset();                        //复位（清零）当前最大关中断时间
167
168
169     /* 创建多值信号量 sem */
170     OSSemCreate ((OS_SEM      *)&sem,                   //指向要创建的多值信号量
171                  (CPU_CHAR    *)"Sem For Test",         //多值信号量的名字
172                  (OS_SEM_CTR   )0,                      //多值信号量初始不可用
173                  (OS_ERR      *)&err);                  //返回错误类型
174
175     /* 创建消息队列 queue */
176     OSQCreate ((OS_Q          *)&queue,                 //指向要创建的消息队列
177                (CPU_CHAR      *)"Queue For Test",       //消息队列的名字
178                (OS_MSG_QTY     )20,                     //最多可容20条消息
179                (OS_ERR        *)&err);                  //返回错误类型
180
181
```

图 10-7 创建多值信号量和消息队列

另外，还须声明和初始化等待对象数组（必须保证该数组在被使用到之前已被声明和初始化）。

```
43  *                                                   LOCAL DEFINES
44  **********************************************************************************
45  */
46
47  OS_Q            queue;                          //声明消息队列
48  OS_SEM          sem;                            //声明多值信号量
49  OS_PEND_DATA    mul_pend_array [ 2 ];           //声明等待对象数组
50
```

图 10-8 声明等待对象数组

```
181
182     /* 初始化要等待的多个内核对象 */
183     mul_pend_array [ 0 ] .PendObjPtr = ( OS_PEND_OBJ * ) & sem;
184     mul_pend_array [ 1 ] .PendObjPtr = ( OS_PEND_OBJ * ) & queue;
185
```

图 10-9 初始化等待对象数组

任务函数 AppTaskPost () 的定义如下。任务每隔 1s 向消息队列 queue 发送一个消息。任务还创建了一个软件定时器 tmr_for_key，循环定时 100ms，每次定时结束都会调用回调函数 TmrCallback() 扫描按键 KEY1，如果 KEY1 被单击，就发布多值信号量 sem，标志 KEY1 被单击。

```
246 static  void  AppTaskPost ( void * p_arg )
247 {
248   OS_ERR      err;
249   OS_TMR      tmr_for_key;                                  //声明软件定时器
250
251
252   (void)p_arg;
253
254
255   /* 创建软件定时器 */
256   OSTmrCreate ((OS_TMR              *)&tmr_for_key,         //软件定时器对象
257                (CPU_CHAR            *)"Tmr For Key",        //命名软件定时器
258                (OS_TICK              )0,                    //定时器初始值
259                (OS_TICK              )1,                    //周期重载值100ms
260                (OS_OPT               )OS_OPT_TMR_PERIODIC,  //周期性定时
261                (OS_TMR_CALLBACK_PTR  )TmrCallback,          //回调函数
262                (void                *)"Timer Over!",        //传递实参给回调函数
263                (OS_ERR              *)&err);                //返回错误类型
264
265   /* 启动软件定时器 */
266   OSTmrStart ((OS_TMR   *)&tmr_for_key,                     //软件定时器对象
267               (OS_ERR   *)err);                             //返回错误类型
268
269
270   while (DEF_TRUE) {                                        //任务体
271     /* 发布消息到消息队列 queue */
272     OSQPost ((OS_Q         *)&queue,                        //消息队列
273              (void         *)"One Message",                 //消息内容
274              (OS_MSG_SIZE   )sizeof ( "One Message" ),      //消息长度
275              (OS_OPT        )OS_OPT_POST_FIFO |             //发到入队端
276                              OS_OPT_POST_ALL,               //发给全部等待任务
277              (OS_ERR       *)&err);                        //返回错误类型
278
279     OSTimeDlyHMSM ( 0, 0, 1, 0, OS_OPT_TIME_DLY, & err );  //每隔1s发送一次消息
280
281   }
282
283 }
```

图 10-10 AppTaskPost () 任务函数

```
228 void TmrCallback (OS_TMR *p_tmr, void *p_arg)             //软件定时器 tmr_for_key 的回调函数
229 {
230   OS_ERR             err;
231
232   static uint8_t     ucKey1Press = 0;
233
234
235   if ( Key_Scan ( macKEY1_GPIO_PORT, macKEY1_GPIO_PIN, 1, & ucKey1Press ) )//如果KEY1被单击
236   {
237     /* 发布多值信号量 sem */
238     OSSemPost ((OS_SEM  *)&sem,                                 //多值信号量
239                (OS_OPT   )OS_OPT_POST_ALL,                      //发给全部等待任务
240                (OS_ERR  *)&err);                                //返回错误类型
241   }
242
243 }
```

图 10-11 定时器回调函数 TmrCallback()

任务函数 AppTaskPend () 的定义如下。任务同时等待多值信号量 sem 和消息队列，只要有一个对象可用，任务就脱离等待继续运行，查看哪些对象可用，并向串口调试助手打印有关信息。

```
291 static  void  AppTaskPend ( void * p_arg )
292 {
293   OS_ERR      err;
294
295
296   (void)p_arg;
297
298
299   while (DEF_TRUE) {                                                  //任务体
300     /* 等待多个内核对象 */
301     OSPendMulti ((OS_PEND_DATA  *)mul_pend_array,        //等待对象数组
302                  (OS_OBJ_QTY     )2,                     //等待对象数目
303                  (OS_TICK        )0,                     //无期限等待
304                  (OS_OPT         )OS_OPT_PEND_BLOCKING, //阻塞任务
305                  (OS_ERR        *)&err);                 //返回错误类型
306
307     /* 查看哪些等待对象可用 */
308     if ( mul_pend_array [ 0 ] .RdyObjPtr == mul_pend_array [ 0 ] .PendObjPtr ) //如果 sem 可用
309     {
310       printf("\r\nKEY1被单击\r\n");
311     }
312     if ( mul_pend_array [ 1 ] .RdyObjPtr == mul_pend_array [ 1 ] .PendObjPtr )//如果 queue 可用
313     {
314       printf("\r\n接收到一条消息，内容为：%s，长度为：%d字节\r\n",
315              ( char * ) mul_pend_array [ 1 ] .RdyMsgPtr,  mul_pend_array [ 1 ] .RdyMsgSize );
316     }
317
318   }
319
320 }
```

图 10-12 AppTaskPend () 任务函数

把 STM32 的 USART1 连接至电脑的串口调试助手，编译和下载程序到秉火 STM32 开发板，运行程序。用户可以看到串口调试助手上打印任务 AppTaskPend () 每隔 1s 接收到的消息长度和内容。当 KEY1 被单击时，串口调试助手还会打印提示信息。

图 10-13 串口调试助手

10.3 章末总结

等待多个内核对象，是指任务可以同时等待多个内核对象，当其中有一个对象被可用时，任务就可以脱离等待。需要注意的是，这里的多个内核对象，是指多值信号量和消息队列的任意组合，一般将该组合声明为一个数组。等待多个内核对象使用 OSPendMulti () 函数。多值信号量和消息队列的发布函数依然使用它们各自原有的发布函数，用法也不变。

第11章 任务信号量

在实际任务间的通信中，一个或多个任务发送一个信号量或者消息给另一个任务是比常见的，而一个任务给多个任务发送信号量和消息相对比较少。前面所讲的信号量和消息队列均是单独的内核对象，是独立于任务存在的。这两章要讲述的任务信号量和任务消息队列是任务特有的属性，紧紧依赖于一个特定任务。

任务信号量和任务消息队列分别与多值信号量和消息队列非常相似，不同之处是，前者仅发布给一个特定任务，而后者可以发布给多个任务。因此，前者的操作相对比较简单，而且省时。如果任务信号量和任务消息队列可以满足设计需求，那么尽量不要使用普通多值信号量和消息队列。

本章先侧重介绍任务信号量，下一章介绍任务消息队列。

11.1 原理简述

任务信号量伴随任务存在，只要创建了任务，其任务信号量就是该任务的一个数据成员，任务信号量的数据成员被包含在任务控制块里。

11.1.1 OSTaskSemPost ()

OSTaskSemPost () 函数用于给一个任务发布任务信号量。OSTaskSemPost () 函数的信息如下表所示。

表 36 OSTaskSemPost ()

<table>
<tr><td>函数原型</td><td colspan="4">OS_SEM_CTR OSTaskSemPost (OS_TCB *p_tcb,
OS_OPT opt,
OS_ERR *p_err);</td></tr>
<tr><td>功能</td><td colspan="4">给一个任务发布任务信号量。</td></tr>
<tr><td rowspan="6">参数</td><td>p_tcb</td><td colspan="3">目标任务控制块。</td></tr>
<tr><td rowspan="2">opt</td><td>OS_OPT_POST_NONE</td><td colspan="2">没有选项。</td></tr>
<tr><td>OS_OPT_POST_NO_SCHED</td><td colspan="2">不进行任务调度。</td></tr>
<tr><td rowspan="3">p_err</td><td rowspan="3">返回错误类型</td><td>OS_ERR_NONE</td><td>没错误</td></tr>
<tr><td>OS_ERR_SEM_OVF</td><td>该发布将导致了信号量的计数值溢出。</td></tr>
<tr><td>OS_ERR_STATE_INVALID</td><td>任务状态非法。</td></tr>
<tr><td>返回值</td><td colspan="4">✧ 0，有错误。
✧ 其他值，任务信号量的当前计数值。</td></tr>
</table>

OSTaskSemPost () 函数的定义也位于“os_task.c”。

app.c* os_task.c

```
1497 OS_SEM_CTR  OSTaskSemPost (OS_TCB  *p_tcb,    //目标任务
1498                            OS_OPT   opt,      //选项
1499                            OS_ERR  *p_err)    //返回错误类型
1500 {
1501     OS_SEM_CTR  ctr;
1502     CPU_TS      ts;
1503
1504
1505
1506 #ifdef OS_SAFETY_CRITICAL                  //如果使能（默认禁用）了安全检测
1507     if (p_err == (OS_ERR *)0) {            //如果 p_err 为空
1508         OS_SAFETY_CRITICAL_EXCEPTION();    //执行安全检测异常函数
1509         return ((OS_SEM_CTR)0);            //返回0（有错误），停止执行
1510     }
1511 #endif
1512
1513 #if OS_CFG_ARG_CHK_EN > 0u                 //如果使能（默认使能）了参数检测功能
1514     switch (opt) {                         //根据选项分类处理
1515         case OS_OPT_POST_NONE:             //如果选项在预期之内
1516         case OS_OPT_POST_NO_SCHED:
1517              break;                        //跳出
1518
1519         default:                           //如果选项超出预期
1520             *p_err =  OS_ERR_OPT_INVALID;  //错误类型为“选项非法”
1521              return ((OS_SEM_CTR)0u);      //返回0（有错误），停止执行
1522     }
1523 #endif

1524
1525     ts = OS_TS_GET();                                   //获取时间戳
1526
1527 #if OS_CFG_ISR_POST_DEFERRED_EN > 0u                    //如果使能了中断延迟发布
1528     if (OSIntNestingCtr > (OS_NESTING_CTR)0) {          //如果该函数是在中断中被调用
1529         OS_IntQPost((OS_OBJ_TYPE)OS_OBJ_TYPE_TASK_SIGNAL,  //将该信号量发布到中断消息队列
1530                     (void      *)p_tcb,
1531                     (void      *)0,
1532                     (OS_MSG_SIZE)0,
1533                     (OS_FLAGS   )0,
1534                     (OS_OPT     )0,
1535                     (CPU_TS     )ts,
1536                     (OS_ERR    *)p_err);
1537         return ((OS_SEM_CTR)0);                         //返回0（尚未发布）
1538     }
1539 #endif
1540
1541     ctr = OS_TaskSemPost(p_tcb,                         //将信号量按照普通方式处理
1542                          opt,
1543                          ts,
1544                          p_err);
1545
1546     return (ctr);                                       //返回信号的当前计数值
1547 }
```

图 11-1 OSTaskSemPost () 函数

其实，不管是否使能了中断延迟发布，最终都是调用 OS_TaskSemPost() 函数进行发布信号量。只是使能了中断延迟发布的发布过程会比较曲折，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。OS_TaskSemPost() 函数的定义位于“os_sem.c”。

app.c* os_task.c

```
2342 OS_SEM_CTR  OS_TaskSemPost (OS_TCB  *p_tcb,    //目标任务
2343                             OS_OPT   opt,      //选项
2344                             CPU_TS   ts,       //时间戳
2345                             OS_ERR  *p_err)    //返回错误类型
2346 {
2347     OS_SEM_CTR  ctr;
2348     CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必需该宏，该宏声明和
2349                     //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
2350                     // SR（临界段关中断只需保存SR），开中断时将该值还原。
2351
2352     OS_CRITICAL_ENTER();                                    //进入临界段
2353     if (p_tcb == (OS_TCB *)0) {                             //如果 p_tcb 为空
2354         p_tcb = OSTCBCurPtr;                                //将任务信号量发给自己（任务）
2355     }
2356     p_tcb->TS = ts;                                         //记录信号量被发布的时间戳
2357    *p_err     = OS_ERR_NONE;                                //错误类型为“无错误”


2358     switch (p_tcb->TaskState) {                             //跟吴目标任务的任务状态分类处理
2359         case OS_TASK_STATE_RDY:                             //如果目标任务没有等待状态
2360         case OS_TASK_STATE_DLY:
2361         case OS_TASK_STATE_SUSPENDED:
2362         case OS_TASK_STATE_DLY_SUSPENDED:
2363              switch (sizeof(OS_SEM_CTR)) {                          //判断是否将导致该信
2364                  case 1u:                                           //号量计数值溢出，如
2365                       if (p_tcb->SemCtr == DEF_INT_08U_MAX_VAL) {   //果溢出，则开中断，
2366                           OS_CRITICAL_EXIT();                       //返回错误类型为“计
2367                          *p_err = OS_ERR_SEM_OVF;                   //数值溢出”，返回0
2368                           return ((OS_SEM_CTR)0);                   //（有错误），不继续
2369                       }                                             //执行。
2370                       break;
2371
2372                  case 2u:
2373                       if (p_tcb->SemCtr == DEF_INT_16U_MAX_VAL) {
2374                           OS_CRITICAL_EXIT();
2375                          *p_err = OS_ERR_SEM_OVF;
2376                           return ((OS_SEM_CTR)0);
2377                       }
2378                       break;
2379


2380                  case 4u:
2381                       if (p_tcb->SemCtr == DEF_INT_32U_MAX_VAL) {
2382                           OS_CRITICAL_EXIT();
2383                          *p_err = OS_ERR_SEM_OVF;
2384                           return ((OS_SEM_CTR)0);
2385                       }
2386                       break;
2387
2388                  default:
2389                       break;
2390              }
2391              p_tcb->SemCtr++;                                  //信号量计数值不溢出则加1
2392              ctr = p_tcb->SemCtr;                              //获取信号量的当前计数值
2393              OS_CRITICAL_EXIT();                               //退出临界段
2394              break;                                            //跳出
2395
```

```
2396        case OS_TASK_STATE_PEND:                                    //如果任务有等待状态
2397        case OS_TASK_STATE_PEND_TIMEOUT:
2398        case OS_TASK_STATE_PEND_SUSPENDED:
2399        case OS_TASK_STATE_PEND_TIMEOUT_SUSPENDED:
2400            if (p_tcb->PendOn == OS_TASK_PEND_ON_TASK_SEM) { //如果正等待任务信号量
2401                OS_Post((OS_PEND_OBJ *)0,                           //发布信号量给目标任务
2402                        (OS_TCB      *)p_tcb,
2403                        (void        *)0,
2404                        (OS_MSG_SIZE  )0u,
2405                        (CPU_TS       )ts);
2406                ctr = p_tcb->SemCtr;                                //获取信号量的当前计数值
2407                OS_CRITICAL_EXIT_NO_SCHED();                        //退出临界段（无调度）
2408                if ((opt & OS_OPT_POST_NO_SCHED) == (OS_OPT)0) { //如果选择了调度任务
2409                    OSSched();                                      //调度任务
2410                }

2411            } else {                                                //如果没等待任务信号量
2412                switch (sizeof(OS_SEM_CTR)) {                               //判断是否将导致
2413                    case 1u:                                                //该信号量计数值
2414                        if (p_tcb->SemCtr == DEF_INT_08U_MAX_VAL) {  //溢出，如果溢出，
2415                            OS_CRITICAL_EXIT();                             //则开中断，返回
2416                           *p_err = OS_ERR_SEM_OVF;                         //错误类型为“计
2417                            return ((OS_SEM_CTR)0);                         //数值溢出”，返
2418                        }                                                   //回0（有错误），
2419                        break;                                              //不继续执行。
2420
2421                    case 2u:
2422                        if (p_tcb->SemCtr == DEF_INT_16U_MAX_VAL) {
2423                            OS_CRITICAL_EXIT();
2424                           *p_err = OS_ERR_SEM_OVF;
2425                            return ((OS_SEM_CTR)0);
2426                        }
2427                        break;
2428
2429                    case 4u:
2430                        if (p_tcb->SemCtr == DEF_INT_32U_MAX_VAL) {
2431                            OS_CRITICAL_EXIT();
2432                           *p_err = OS_ERR_SEM_OVF;
2433                            return ((OS_SEM_CTR)0);
2434                        }
2435                        break;
2436
2437                    default:
2438                        break;
2439                }

2440                p_tcb->SemCtr++;                                    //信号量计数值不溢出则加1
2441                ctr = p_tcb->SemCtr;                                //获取信号量的当前计数值
2442                OS_CRITICAL_EXIT();                                 //退出临界段
2443            }
2444            break;                                                  //跳出
2445
2446        default:                                                    //如果任务状态超出预期
2447            OS_CRITICAL_EXIT();                                     //退出临界段
2448           *p_err = OS_ERR_STATE_INVALID;                           //错误类型为“状态非法”
2449            ctr   = (OS_SEM_CTR)0;                                  //清零 ctr
2450            break;                                                  //跳出
2451    }
2452    return (ctr);                                                   //返回信号量的当前计数值
2453}
```

图 11-2 OS_TaskSemPost() 函数

在 OS_SemPost() 函数中，又会调用 OS_Post() 函数发布内核对象。OS_Post() 函数是一个底层的发布函数，它不仅仅用来发布任务信号量，还可以发布多值信号量、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发布给目标任务。OS_Post() 函数的定义位于“os_core.c”。

os_sem.c | os.h | os_core.c | os_cfg.h

```
1844 void  OS_Post (OS_PEND_OBJ  *p_obj,      //内核对象类型指针
1845              OS_TCB       *p_tcb,      //任务控制块
1846              void         *p_void,     //消息
1847              OS_MSG_SIZE   msg_size,   //消息大小
1848              CPU_TS        ts)         //时间戳
1849 {
1850     switch (p_tcb->TaskState) {                                  //根据任务状态分类处理
1851         case OS_TASK_STATE_RDY:                                  //如果任务处于就绪状态
1852         case OS_TASK_STATE_DLY:                                  //如果任务处于延时状态
1853         case OS_TASK_STATE_SUSPENDED:                            //如果任务处于挂起状态
1854         case OS_TASK_STATE_DLY_SUSPENDED:                        //如果任务处于延时中被挂起状态
1855              break;                                              //不用处理，直接跳出

1856
1857         case OS_TASK_STATE_PEND:                                 //如果任务处于无期限等待状态
1858         case OS_TASK_STATE_PEND_TIMEOUT:                         //如果任务处于有期限等待状态
1859              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
1860                  OS_Post1(p_obj,                                 //标记哪个内核对象被发布
1861                           p_tcb,
1862                           p_void,
1863                           msg_size,
1864                           ts);
1865              } else {                                            //如果任务不是在等待多个信号量或消息队列
1866 #if (OS_MSG_EN > 0u)                                             //如果使能了任务队列或消息队列
1867                  p_tcb->MsgPtr  = p_void;                        //保存消息到等待任务
1868                  p_tcb->MsgSize = msg_size;
1869 #endif
1870                  p_tcb->TS      = ts;                            //保存时间戳到等待任务
1871              }
1872              if (p_obj != (OS_PEND_OBJ *)0) {                    //如果内核对象为空
1873                  OS_PendListRemove(p_tcb);                       //从等待列表移除该等待任务
1874 #if OS_CFG_DBG_EN > 0u                                           //如果使能了调试代码和变量
1875                  OS_PendDbgNameRemove(p_obj,                     //移除内核对象的调试名
1876                                       p_tcb);
1877 #endif
1878              }
1879              OS_TaskRdy(p_tcb);                                  //让该等待任务准备运行
1880              p_tcb->TaskState  = OS_TASK_STATE_RDY;              //任务状态改为就绪状态
1881              p_tcb->PendStatus = OS_STATUS_PEND_OK;              //清除等待状态
1882              p_tcb->PendOn     = OS_TASK_PEND_ON_NOTHING;        //标记不再等待
1883              break;

1884
1885         case OS_TASK_STATE_PEND_SUSPENDED:                       //如果任务在无期限等待中被挂起
1886         case OS_TASK_STATE_PEND_TIMEOUT_SUSPENDED:               //如果任务在有期限等待中被挂起
1887              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
1888                  OS_Post1(p_obj,                                 //标记哪个内核对象被发布
1889                           p_tcb,
1890                           p_void,
1891                           msg_size,
1892                           ts);
1893              } else {                                            //如果任务不在等待多个信号量或消息队列
1894 #if (OS_MSG_EN > 0u)                                             //如果使能了调试代码和变量
1895                  p_tcb->MsgPtr  = p_void;                        //保存消息到等待任务
1896                  p_tcb->MsgSize = msg_size;
1897 #endif
1898                  p_tcb->TS      = ts;                            //保存时间戳到等待任务
1899              }
1900              OS_TickListRemove(p_tcb);                           //从节拍列表移除该等待任务
1901              if (p_obj != (OS_PEND_OBJ *)0) {                    //如果内核对象为空
1902                  OS_PendListRemove(p_tcb);                       //从等待列表移除该等待任务
1903 #if OS_CFG_DBG_EN > 0u                                           //如果使能了调试代码和变量
1904                  OS_PendDbgNameRemove(p_obj,                     //移除内核对象的调试名
1905                                       p_tcb);
1906 #endif
1907              }
1908              p_tcb->TaskState  = OS_TASK_STATE_SUSPENDED;        //任务状态改为被挂起状态
1909              p_tcb->PendStatus = OS_STATUS_PEND_OK;              //清除等待状态
1910              p_tcb->PendOn     = OS_TASK_PEND_ON_NOTHING;        //标记不再等待
1911              break;
1912
1913         default:                                                 //如果任务状态超出预期
1914              break;                                              //直接跳出
1915     }
1916 }
```

图 11-3 OS_Post() 函数

11.1.2 OSTaskSemPend ()

与 OSSemPost () 多值信号量发布函数相对应，OSTaskSemPend () 函数用于等待任务信号量。

表 37 OSTaskSemPend ()

函数原型	OS_SEM_CTR OSTaskSemPend (OS_TICK timeout, OS_OPT opt, CPU_TS *p_ts, CPU_TS *p_ts, OS_ERR *p_err);			
功能	等待任务信号量。			
	timeout	等待超时时间（单位：时钟节拍），0 代表无期限等待。opt 为 OS_OPT_PEND_BLOCKING 时该参数才起作用。		
	opt	选项	OS_OPT_PEND_BLOCKING	如果不能立即获得信号量，就堵塞当前任务，继续等待信号量。
			OS_OPT_PEND_NON_BLOCKING	如果不能立即获得信号量，不堵塞当前任务，不继续等待信号量。
	p_ts	时间戳	用于存储信号量最后一次被发布的时间戳，或者等待被中止的时间戳，或者信号量被删除时的时间戳，具体返回哪个时间戳，要根据返回的 p_err 判断。该参数可以为 NULL，表示用户不需要获得时间戳。	
	p_err	返回错误类型	OS_ERR_NONE	没错误，成功获得信号量。
			OS_ERR_PEND_ABORT	等待被另一个任务中止。
			OS_ERR_PEND_ISR	在中断中被调用。
			OS_ERR_PEND_WOULD_BLOCK	缺乏堵塞。
			OS_ERR_SCHED_LOCKED	调度器被锁。
			OS_ERR_STATUS_INVALID	等待状态非法。
			OS_ERR_TIMEOUT	等待超时。
返回值	✧ 0，信号量的当前计数值为 0，或有错误。 ✧ 其他值，信号量的当前计数值。			
注意事项	✧ 不可以在中断中调用该函数。			

OSTaskSemPend () 函数的定义也位于“os_task.c”。

```
1262 OS_SEM_CTR  OSTaskSemPend (OS_TICK   timeout,   //等待超时时间
1263                            OS_OPT    opt,       //选项
1264                            CPU_TS   *p_ts,      //返回时间戳
1265                            OS_ERR   *p_err)     //返回错误类型
1266 {
1267     OS_SEM_CTR    ctr;
1268     CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必需该宏，该宏声明和
1269                     //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
1270                     // SR（临界段关中断只需保存SR），开中断时将该值还原。
1271
1272 #ifdef OS_SAFETY_CRITICAL                   //如果使能了安全检测
1273     if (p_err == (OS_ERR *)0) {             //如果错误类型实参为空
1274         OS_SAFETY_CRITICAL_EXCEPTION();     //执行安全检测异常函数
1275         return ((OS_SEM_CTR)0);             //返回0（有错误），停止执行
1276     }
1277 #endif
1278
1279 #if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u          //如果使能了中断中非法调用检测
1280     if (OSIntNestingCtr > (OS_NESTING_CTR)0) {  //如果该函数在中断中被调用
1281        *p_err = OS_ERR_PEND_ISR;                //返回错误类型为“在中断中等待”
1282         return ((OS_SEM_CTR)0);                 //返回0（有错误），停止执行
1283     }
1284 #endif

1285
1286 #if OS_CFG_ARG_CHK_EN > 0u                  //如果使能了参数检测
1287     switch (opt) {                          //根据选项分类处理
1288         case OS_OPT_PEND_BLOCKING:          //如果选项在预期内
1289         case OS_OPT_PEND_NON_BLOCKING:
1290              break;                         //直接跳出
1291
1292         default:                            //如果选项超出预期
1293             *p_err = OS_ERR_OPT_INVALID;    //错误类型为“选项非法”
1294              return ((OS_SEM_CTR)0);        //返回0（有错误），停止执行
1295     }
1296 #endif

1297
1298     if (p_ts != (CPU_TS *)0) {      //如果 p_ts 非空
1299        *p_ts  = (CPU_TS  )0;        //清零（初始化）p_ts
1300     }
1301
1302     CPU_CRITICAL_ENTER();                           //关中断
1303     if (OSTCBCurPtr->SemCtr > (OS_SEM_CTR)0) {      //如果任务信号量当前可用
1304         OSTCBCurPtr->SemCtr--;                      //信号量计数器减1
1305         ctr     = OSTCBCurPtr->SemCtr;              //获取信号量的当前计数值
1306         if (p_ts != (CPU_TS *)0) {                  //如果 p_ts 非空
1307            *p_ts  = OSTCBCurPtr->TS;                //返回信号量被发布的时间戳
1308         }
1309 #if OS_CFG_TASK_PROFILE_EN > 0u                     //如果使能了任务控制块的简况变量
1310         OSTCBCurPtr->SemPendTime = OS_TS_GET() - OSTCBCurPtr->TS;     //更新任务等待
1311         if (OSTCBCurPtr->SemPendTimeMax < OSTCBCurPtr->SemPendTime) { //任务信号量的
1312             OSTCBCurPtr->SemPendTimeMax = OSTCBCurPtr->SemPendTime;   //最长时间记录。
1313         }
1314 #endif
1315         CPU_CRITICAL_EXIT();                        //开中断
1316        *p_err = OS_ERR_NONE;                        //错误类型为“无错误”
1317         return (ctr);                               //返回信号量的当前计数值
1318     }
```

```
1319     /* 如果任务信号量当前不可用 */
1320     if ((opt & OS_OPT_PEND_NON_BLOCKING) != (OS_OPT)0) {  //如果选择了不阻塞任务
1321         CPU_CRITICAL_EXIT();                              //开中断
1322        *p_err = OS_ERR_PEND_WOULD_BLOCK;                  //错误类型为“缺乏阻塞”
1323         return ((OS_SEM_CTR)0);                           //返回0（有错误），停止执行
1324     } else {                                              //如果选择了阻塞任务
1325         if (OSSchedLockNestingCtr > (OS_NESTING_CTR)0) {  //如果调度器被锁
1326             CPU_CRITICAL_EXIT();                          //开中断
1327            *p_err = OS_ERR_SCHED_LOCKED;                  //错误类型为“调度器被锁”
1328             return ((OS_SEM_CTR)0);                       //返回0（有错误），停止执行
1329         }
1330     }
1331     /* 如果调度器未被锁 */
1332     OS_CRITICAL_ENTER_CPU_EXIT();                         //锁调度器，重开中断
1333     OS_Pend((OS_PEND_DATA *)0,                            //阻塞任务，等待信号量。
1334             (OS_PEND_OBJ  *)0,                            //不需插入等待列表。
1335             (OS_STATE      )OS_TASK_PEND_ON_TASK_SEM,
1336             (OS_TICK       )timeout);
1337     OS_CRITICAL_EXIT_NO_SCHED();                          //开调度器（无调度）
1338
1339     OSSched();                                            //调度任务

1340     /* 任务获得信号量后得以继续运行 */
1341     CPU_CRITICAL_ENTER();                                 //关中断
1342     switch (OSTCBCurPtr->PendStatus) {                    //根据任务的等待状态分类处理
1343         case OS_STATUS_PEND_OK:                           //如果任务成功获得信号量
1344              if (p_ts != (CPU_TS *)0) {                   //返回信号量被发布的时间戳
1345                 *p_ts                  =  OSTCBCurPtr->TS;
1346 #if OS_CFG_TASK_PROFILE_EN > 0u                           //更新最长等待时间记录
1347                 OSTCBCurPtr->SemPendTime = OS_TS_GET() - OSTCBCurPtr->TS;
1348                 if (OSTCBCurPtr->SemPendTimeMax < OSTCBCurPtr->SemPendTime) {
1349                     OSTCBCurPtr->SemPendTimeMax = OSTCBCurPtr->SemPendTime;
1350                 }
1351 #endif
1352              }
1353             *p_err = OS_ERR_NONE;                         //错误类型为“无错误”
1354              break;                                       //跳出
1355
1356         case OS_STATUS_PEND_ABORT:                        //如果等待被中止
1357              if (p_ts != (CPU_TS *)0) {                   //返回被终止时的时间戳
1358                 *p_ts  =  OSTCBCurPtr->TS;
1359              }
1360             *p_err = OS_ERR_PEND_ABORT;                   //错误类型为“等待被中止”
1361              break;                                       //跳出
1362

1363         case OS_STATUS_PEND_TIMEOUT:                      //如果等待超时
1364              if (p_ts != (CPU_TS *)0) {                   //返回时间戳为0
1365                 *p_ts  = (CPU_TS  )0;
1366              }
1367             *p_err = OS_ERR_TIMEOUT;                      //错误类型为“等待超时”
1368              break;                                       //跳出
1369
1370         default:                                          //如果等待状态超出预期
1371             *p_err = OS_ERR_STATUS_INVALID;               //错误类型为“状态非法”
1372              break;                                       //跳出
1373     }
1374     ctr = OSTCBCurPtr->SemCtr;                            //获取信号量的当前计数值
1375     CPU_CRITICAL_EXIT();                                  //开中断
1376     return (ctr);                                         //返回信号量的当前计数值
1377 }
```

图 11-4 OSTaskSemPend () 函数

当需要阻塞任务，等待任务信号量时，OSTaskSemPend () 函数会调用一个更加底层的等待函数来执行当前任务对多值信号量的等待，该函数就是 OS_Pend()。与 OS_Post() 函数一样，OS_Pend() 函数不仅仅用来等待任务信号量，还可以等待多值信号量、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_Pend()函数并没有把当前任务插入到等待列表。OS_Pend() 函数的定义位于“os_core.c”。

```
os_sem.c  os.h  os_core.c  os_cfg.h  os_type.h
873 void  OS_Pend (OS_PEND_DATA  *p_pend_data,   //待插入等待列表的元素
874               OS_PEND_OBJ   *p_obj,         //等待的内核对象
875               OS_STATE       pending_on,    //等待哪种对象内核
876               OS_TICK        timeout)       //等待期限
877 {
878     OS_PEND_LIST  *p_pend_list;
879
880
881
882     OSTCBCurPtr->PendOn     = pending_on;                    //资源不可用，开始等待
883     OSTCBCurPtr->PendStatus = OS_STATUS_PEND_OK;             //正常等待中
884
885     OS_TaskBlock(OSTCBCurPtr,                                //阻塞当前运行任务，
886                  timeout);                                   //如果 timeout 非0，把任务插入的节拍列表
887
888     if (p_obj != (OS_PEND_OBJ *)0) {                         //如果等待对象非空
889         p_pend_list           = &p_obj->PendList;            //获取对象的等待列表到 p_pend_list
890         p_pend_data->PendObjPtr = p_obj;                     //保存要等待的对象
891         OS_PendDataInit((OS_TCB      *)OSTCBCurPtr,          //初始化 p_pend_data（待插入等待列表）
892                         (OS_PEND_DATA *)p_pend_data,
893                         (OS_OBJ_QTY    )1);
894         OS_PendListInsertPrio(p_pend_list,                   //按优先级将 p_pend_data 插入到等待列表
895                               p_pend_data);
896     } else {                                                 //如果等待对象为空
897         OSTCBCurPtr->PendDataTblEntries = (OS_OBJ_QTY    )0; //清零当前任务的等待域数据
898         OSTCBCurPtr->PendDataTblPtr     = (OS_PEND_DATA *)0;
899     }
900 #if OS_CFG_DBG_EN > 0u                                       //如果使能了调试代码和变量
901     OS_PendDbgNameAdd(p_obj,                                 //更新信号量的 DbgNamePtr 元素为其等待
902                       OSTCBCurPtr);                          //列表中优先级最高的任务的名称。
903 #endif
904 }
```

图 11-5 OS_Pend() 函数

11.1.3 OSTaskSemPendAbort ()

OSTaskSemPendAbort() 函数用于中止一个任务对其任务信号量的等待。要使用 OSTaskSemPendAbort() 函数，还得事先使能 OS_CFG_TASK_SEM_PEND_ABORT_EN（位于“os_cfg.h”），如下图所示。

```
os_task.c  os_cfg.h
82                                          /* ------------------------ TASK MANAGEMENT ----
83 #define OS_CFG_STAT_TASK_EN             1u   //使能/禁用统计任务
84 #define OS_CFG_STAT_TASK_STK_CHK_EN     1u   //使能/禁用从统计任务检查堆栈
85
86 #define OS_CFG_TASK_CHANGE_PRIO_EN      1u   //使能/禁用函数 OSTaskChangePrio()
87 #define OS_CFG_TASK_DEL_EN              1u   //使能/禁用函数 OSTaskDel()
88 #define OS_CFG_TASK_Q_EN                1u   //使能/禁用函数 OSTaskQXXXX()
89 #define OS_CFG_TASK_Q_PEND_ABORT_EN     1u   //使能/禁用函数 OSTaskQPendAbort()
90 #define OS_CFG_TASK_PROFILE_EN          1u   //使能/禁用任务控制块的简况变量
91 #define OS_CFG_TASK_REG_TBL_SIZE        1u   //定义任务指定的寄存器数目
92 #define OS_CFG_TASK_SEM_PEND_ABORT_EN   1u   //使能/禁用函数 OSTaskSemPendAbort()
93 #define OS_CFG_TASK_SUSPEND_EN          1u   //使能/禁用函数 OSTaskSuspend() 和 OSTaskResume()
94
```

图 11-6

OSTaskSemPendAbort() 函数的信息如下表所示。

表 38 OSTaskSemPendAbort ()

函数原型	CPU_BOOLEAN OSTaskSemPendAbort (OS_TCB *p_tcb, OS_OPT opt, OS_ERR *p_err);	
功能	中止一个任务对其任务信号量的等待。	
参数	p_tcb	目标任务

	opt	选项	OS_OPT_POST_NONE	没有选项。
			OS_OPT_POST_NO_SCHED	不进行任务调度。
	p_err	返回错误类型	OS_ERR_NONE	没错误，成功中止。
			OS_ERR_PEND_ABORT_ISR	该函数在中断中被调用。
			OS_ERR_PEND_ABORT_NONE	目标任务并未在等待任务信号量。
			OS_ERR_PEND_ABORT_SELF	目标任务是自身。
返回值	✧ == DEF_FALSE>0，目标任务没在等待任务信号量，或者有错误。 ✧ == DEF_TRUE，目标任务确实在等待任务信号量，而且等待成功被中止。			
注意事项	✧ 不可以在中断中调用该函数。 ✧ 目标任务不可以是自身（当前运行任务）。			

OSTaskSemPendAbort() 函数的定义位于“os_task.c”。

```
1408 #if OS_CFG_TASK_SEM_PEND_ABORT_EN > 0u  //如果使能了 OSTaskSemPendAbort()
1409 CPU_BOOLEAN  OSTaskSemPendAbort (OS_TCB  *p_tcb, //目标任务
1410                                  OS_OPT   opt,    //选项
1411                                  OS_ERR  *p_err) //返回错误类型
1412 {
1413     CPU_TS         ts;
1414     CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必需该宏，该宏声明和
1415                     //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
1416                     // SR（临界段关中断只需保存SR），开中断时将该值还原。
1417
1418 #ifdef OS_SAFETY_CRITICAL                //如果使能了安全检测
1419     if (p_err == (OS_ERR *)0) {          //如果错误类型实参为空
1420         OS_SAFETY_CRITICAL_EXCEPTION(); //执行安全检测异常函数
1421         return (DEF_FALSE);             //返回（失败），停止执行
1422     }
1423 #endif
1424
1425 #if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u          //如果使能了中断中非法调用检测
1426     if (OSIntNestingCtr > (OS_NESTING_CTR)0) {  //如果该函数是在中断中被调用
1427        *p_err = OS_ERR_PEND_ABORT_ISR;          //错误类型为“在中断中创建对象”
1428         return (DEF_FALSE);                     //返回（失败），停止执行
1429     }
1430 #endif
1431
```

```
1432 #if OS_CFG_ARG_CHK_EN > 0u                  //如果使能了参数检测
1433     switch (opt) {                          //根据选项匪类处理
1434         case OS_OPT_POST_NONE:              //如果选项在预期内
1435         case OS_OPT_POST_NO_SCHED:
1436              break;                         //直接跳出
1437
1438         default:                            //如果选项超出预期
1439             *p_err =  OS_ERR_OPT_INVALID;   //错误类型为“选项非法”
1440              return (DEF_FALSE);            //返回（失败），停止执行
1441     }
1442 #endif
1443
```

```
1444        CPU_CRITICAL_ENTER();                                   //关中断
1445        if ((p_tcb == (OS_TCB *)0) ||                           //如果 p_tcb 为空，或者
1446            (p_tcb == OSTCBCurPtr)) {                           //p_tcb 指向当前运行任务。
1447            CPU_CRITICAL_EXIT();                                //开中断
1448           *p_err = OS_ERR_PEND_ABORT_SELF;                     //错误类型为“中止自身”
1449            return (DEF_FALSE);                                 //返回（失败），停止执行
1450        }


1451        /* 如果 p_tcb （目标任务） 不是当前运行任务（自身） */
1452        if (p_tcb->PendOn != OS_TASK_PEND_ON_TASK_SEM) { //如果目标任务没在等待任务信号量
1453            CPU_CRITICAL_EXIT();                                //开中断
1454           *p_err = OS_ERR_PEND_ABORT_NONE;                     //错误类型为“没在等待任务信号量”
1455            return (DEF_FALSE);                                 //返回（失败），停止执行
1456        }
1457        CPU_CRITICAL_EXIT();                                    //开中断
1458
1459        OS_CRITICAL_ENTER();                                    //进入临界段
1460        ts = OS_TS_GET();                                       //获取时间戳
1461        OS_PendAbort((OS_PEND_OBJ *)0,                          //中止目标任务对信号量的等待
1462                     p_tcb,
1463                     ts);
1464        OS_CRITICAL_EXIT_NO_SCHED();                            //退出临界段（无调度）
1465        if ((opt & OS_OPT_POST_NO_SCHED) == (OS_OPT)0) { //如果选择了任务调度
1466            OSSched();                                          //调度任务
1467        }
1468       *p_err = OS_ERR_NONE;                                    //错误类型为“无错误”
1469        return (DEF_TRUE);                                      //返回（中止成功）
1470    }
1471    #endif
```

图 11-7 OSTaskSemPendAbort() 函数

OSTaskSemPendAbort() 函数会调用一个更加底层的中止等待函数来执行当前任务对多值信号量的等待，该函数就是 OS_PendAbort()。OS_PendAbort() 函数不仅仅用来中止对任务信号量的等待，还可以中止对多值信号量、互斥信号量、消息队列、事件标志组或任务消息队列的等待。OS_PendAbort() 函数的定义位于“os_core.c”。

os_cfg.h | os_sem.c | os.h | os_core.c

```
927 void  OS_PendAbort (OS_PEND_OBJ  *p_obj,    //被等待对象的类型
928                     OS_TCB       *p_tcb,    //任务控制块指针
929                     CPU_TS        ts)       //等待被中止时的时间戳
930 {
931     switch (p_tcb->TaskState) {                                 //根据任务状态分类处理
932         case OS_TASK_STATE_RDY:                                 //如果任务是就绪状态
933         case OS_TASK_STATE_DLY:                                 //如果任务是延时状态
934         case OS_TASK_STATE_SUSPENDED:                           //如果任务是挂起状态
935         case OS_TASK_STATE_DLY_SUSPENDED:                       //如果任务是在延时中被挂起
936              break;                                             //这些情况均与等待无关，直接跳出
937


938         case OS_TASK_STATE_PEND:                                //如果任务是无期限等待状态
939         case OS_TASK_STATE_PEND_TIMEOUT:                        //如果任务是有期限等待状态
940              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
941                  OS_PendAbort1(p_obj,                           //强制解除任务对某一对象的等待
942                                p_tcb,
943                                ts);
944              }
945 #if (OS_MSG_EN > 0u)                                            //如果使能了任务队列或消息队列
946              p_tcb->MsgPtr     = (void      *)0;                //清除（复位）任务的消息域
947              p_tcb->MsgSize    = (OS_MSG_SIZE)0u;
948 #endif
949              p_tcb->TS         = ts;                            //保存等待被中止时的时间戳到任务控制块
950              if (p_obj != (OS_PEND_OBJ *)0) {                   //如果等待对象非空
951                  OS_PendListRemove(p_tcb);                      //将任务从所有等待列表中移除
952              }
953              OS_TaskRdy(p_tcb);                                 //让任务进准备运行
954              p_tcb->TaskState  = OS_TASK_STATE_RDY;             //修改任务状态为就绪状态
955              p_tcb->PendStatus = OS_STATUS_PEND_ABORT;          //标记任务的等待被中止
956              p_tcb->PendOn     = OS_TASK_PEND_ON_NOTHING;       //标记任务目前没有等待任何对象
957              break;                                             //跳出
958
```

```
959         case OS_TASK_STATE_PEND_SUSPENDED:                  //如果任务在无期限等待中被挂起
960         case OS_TASK_STATE_PEND_TIMEOUT_SUSPENDED:          //如果任务在有期限等待中被挂起
961             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
962                 OS_PendAbort1(p_obj,                        //强制解除任务对某一对象的等待
963                               p_tcb,
964                               ts);
965             }
966 #if (OS_MSG_EN > 0u)                                        //如果使能了任务队列或消息队列
967             p_tcb->MsgPtr     = (void      *)0;             //清除（复位）任务的消息域
968             p_tcb->MsgSize    = (OS_MSG_SIZE)0u;
969 #endif
970             p_tcb->TS         = ts;                         //保存等待被中止时的时间戳到任务控制块
971             if (p_obj != (OS_PEND_OBJ *)0) {                //如果等待对象非空
972                 OS_PendListRemove(p_tcb);                   //将任务从所有等待列表中移除
973             }
974             OS_TickListRemove(p_tcb);                       //让任务脱离节拍列表
975             p_tcb->TaskState  = OS_TASK_STATE_SUSPENDED;    //修改任务状态为挂起状态
976             p_tcb->PendStatus = OS_STATUS_PEND_ABORT;       //标记任务的等待被中止
977             p_tcb->PendOn     = OS_TASK_PEND_ON_NOTHING;    //标记任务目前没有等待任何对象
978             break;                                          //跳出
979
980         default:                                            //如果任务状态超出预期
981             break;                                          //不需处理，直接跳出
982     }
983 }
```

图 11-8 OS_PendAbort() 函数

11.2 实例演示

11.2.1 实例 1

本节实例创建两个应用任务，AppTaskPost()和 AppTaskPend()，任务 AppTaskPost()用于扫描按键 KEY1，当 KEY1 被单击时，就向任务 AppTaskPend()发布任务信号量。当任务 AppTaskPend()接收到任务信号量后就切换 LED1 的亮灭状态，并向串口调试助手打印给任务信号量从被发布到被接收的时间差。

该例程已经存放在配套资料的下图路径。

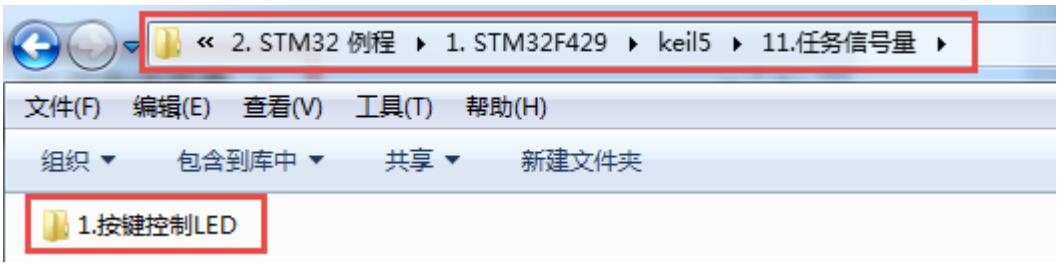

图 11-9 例程路径

本例程需用使用 LED、USART1 和按键，所以工程中需要添加其驱动文件和初始化。用户可参照前面相关实例，这里不再赘述。

任务函数 AppTaskPost() 的定义如下。任务中每隔 20ms 扫描一次 KEY1，如果 KEY1 被单击，就向任务 AppTaskPend()发布任务信号量。

```
202 *********************************************************************************************************
203 *                                             POST TASK
204 *********************************************************************************************************
205 */
206 static  void  AppTaskPost ( void * p_arg )
207 {
208   OS_ERR      err;
209 
210   uint8_t ucKey1Press = 0;     //记忆按键KEY1状态
211 
212 
213   (void)p_arg;
214 
215 
216   while (DEF_TRUE) {                                                    //任务体
217     if( Key_Scan ( macKEY1_GPIO_PORT, macKEY1_GPIO_PIN, 1, & ucKey1Press ) ) //如果KEY1被单击
218     {
219       /* 发布任务信号量 */
220       OSTaskSemPost((OS_TCB  *)&AppTaskPendTCB,                         //目标任务
221                     (OS_OPT   )OS_OPT_POST_NONE,                        //没选项要求
222                     (OS_ERR  *)&err);                                   //返回错误类型
223    }
224 
225     OSTimeDlyHMSM ( 0, 0, 0, 20, OS_OPT_TIME_DLY, & err );              //每20ms扫描一次
226 
227   }
228 
229 }
```

图 11-10 AppTaskPost () 任务函数

任务函数 AppTaskPend () 的定义如下。任务一直等待接收其任务信号量，当其任务信号量被发布时，就切换 LED1 的亮灭状态，并向串口调试助手打印信号量从被发布到被接收到的时间差。

```
233 *********************************************************************************************************
234 *                                             PEND TASK
235 *********************************************************************************************************
236 */
237 static  void  AppTaskPend ( void * p_arg )
238 {
239   OS_ERR          err;
240   CPU_TS          ts;
241   CPU_INT32U      cpu_clk_freq;
242   CPU_SR_ALLOC();
243 
244 
245   (void)p_arg;
246 
247 
248   cpu_clk_freq = BSP_CPU_ClkFreq();                   //获取CPU时钟，时间戳是以该时钟计数
249 
250 
```

```
251   while (DEF_TRUE) {                                   //任务体
252     /* 阻塞任务，直到KEY1被单击 */
253     OSTaskSemPend ((OS_TICK    )0,                     //无期限等待
254                    (OS_OPT     )OS_OPT_PEND_BLOCKING,  //如果信号量不可用就等待
255                    (CPU_TS    *)&ts,                   //获取信号量被发布的时间戳
256                    (OS_ERR    *)&err);                 //返回错误类型
257
258     ts = OS_TS_GET() - ts;                              //计算信号量从发布到接收的时间差
259
260     macLED1_TOGGLE ();                                  //切换LED1的亮灭状态
261
262     OS_CRITICAL_ENTER();                                //进入临界段，避免串口打印被打断
263
264     printf ( "\r\n接收到信号量与发布信号量的时间相差%dus\r\n",
265             ts / ( cpu_clk_freq / 1000000 ) );
266
267     OS_CRITICAL_EXIT();                                 //退出临界段
268
269   }
270
271 }
```

图 11-11 AppTaskPend () 任务函数

把 STM32 的 USART1 连接至电脑的串口调试助手，编译和下载程序到秉火 STM32 开发板，运行程序。当用户每单击一次 KEY1 时，就会看到 LED1 切换一次亮灭状态，并向串口调试助手打印信号量从被发布到被接收到的时间差。。

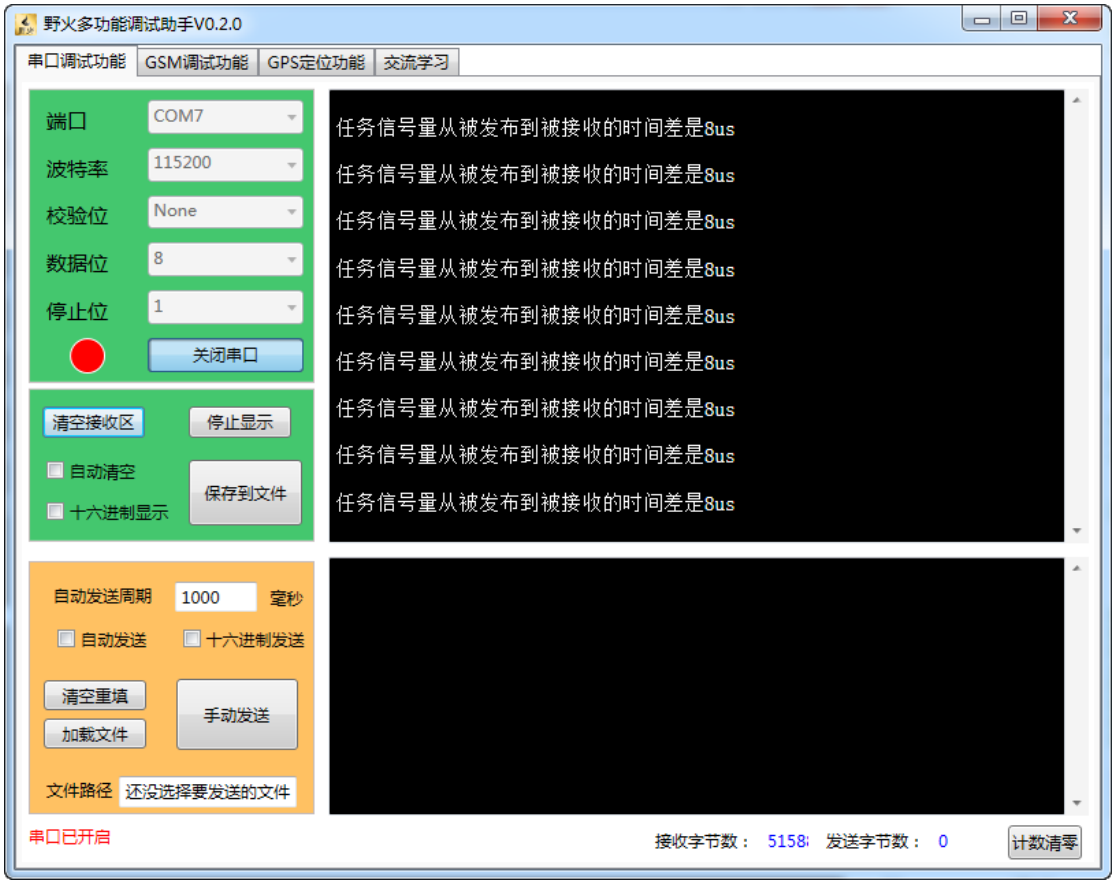

图 11-12 串口调试助手

11.3 章末总结

任务信号量跟多值信号量极其相似。本质区别在于，任务信号量时一个任务的特有属性，是某一个任务的信号量，其他任务均可以发布这个信号量，但只有该任务才能接收这个信号量。多值信号量却是一个独立的内核对象，任何任务均可以发布和接收多值信号量。因此，任务信号量不需像多值信号量那样单独创建，只要创建了任务，就同时创建了其任务信号量。

OSTaskSemPost () 函数用于用于给一个任务发布任务信号量，OSTaskSemPend () 函数用于等待（接收）任务信号量，OSTaskSemPendAbort() 函数用于中止一个任务对其任务信号量的等待。

第12章 任务消息队列

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，任务在则在，只有该任务才可以接收这个任务消息队列的消息，其他任务只能给这个任务消息队列发送消息，却不能接收。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，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。特此声明，本书所提“消息队列”，若无特别说明，均指前面的（普通）消息队列，非任务消息队列。

12.1 原理简述

如果想要使用任务消息队列，就必须事先使能任务消息队列。消息队列的使能位于“os_cfg.h”。

```
82                                                  /* ------------------------ TASK MANAGEMENT ----
83  #define OS_CFG_STAT_TASK_EN                1u   //使能/禁用统计任务
84  #define OS_CFG_STAT_TASK_STK_CHK_EN        1u   //使能/禁用从统计任务检查堆栈
85
86  #define OS_CFG_TASK_CHANGE_PRIO_EN         1u   //使能/禁用函数 OSTaskChangePrio()
87  #define OS_CFG_TASK_DEL_EN                 1u   //使能/禁用函数 OSTaskDel()
88  #define OS_CFG_TASK_Q_EN                   1u   //使能/禁用函数 OSTaskQXXXX()
89  #define OS_CFG_TASK_Q_PEND_ABORT_EN        1u   //使能/禁用函数 OSTaskQPendAbort()
90  #define OS_CFG_TASK_PROFILE_EN             1u   //使能/禁用任务控制块的简况变量
91  #define OS_CFG_TASK_REG_TBL_SIZE           1u   //定义任务指定的寄存器数目
92  #define OS_CFG_TASK_SEM_PEND_ABORT_EN      1u   //使能/禁用函数 OSTaskSemPendAbort()
93  #define OS_CFG_TASK_SUSPEND_EN             1u   //使能/禁用函数 OSTaskSuspend() 和 OSTaskResume()
94
```

图 12-1

特别声明，任务消息队列和（普通）消息队列公用一个消息池。一般任务消息队列或普通消息队列的最大消息容量不要超过消息池的消息数目。

12.1.1 OSTaskQPost ()

OSTaskQPost () 函数用于向任务消息队列发布一个消息。OSTaskQPost () 函数的信息如下表所示。

表 39 OSTaskQPost ()

函数原型	void OSTaskQPost (OS_TCB *p_tcb, void *p_void, OS_MSG_SIZE msg_size, OS_OPT opt, OS_ERR *p_err);	
功能	向任务消息队列发布一个消息。	
参	p_tcb	目标任务。如果该参数为 NULL，消息将发送给当前运行任务。

数	p_void,	消息指针。		
	msg_size	消息长度。		
	opt	选项	OS_OPT_POST_FIFO	把消息发布到队列的入口端。
			OS_OPT_POST_LIFO	把消息发布到队列的出口端。
			OS_OPT_POST_FIFO | OS_OPT_POST_NO_SCHED	把消息发布到队列的入口端；不进行任务调度，继续运行当前任务。
			OS_OPT_POST_LIFO | OS_OPT_POST_NO_SCHED	把消息发布到队列的出口端；不进行任务调度，继续运行当前任务。
	p_err	返回错误类型	OS_ERR_NONE	无错误，消息成功被发布了。
			OS_ERR_Q_MAX	任务消息队列已满。
			OS_ERR_MSG_POOL_EMPTY	消息池没可用消息。
返回值	无。			

OSTaskQPost () 函数的定义也位于“os_task.c

```
929 #if OS_CFG_TASK_Q_EN > 0u                    //如果使能了任务消息队列
930 void  OSTaskQPost (OS_TCB        *p_tcb,     //目标任务
931                    void          *p_void,    //消息内容地址
932                    OS_MSG_SIZE    msg_size,  //消息长度
933                    OS_OPT         opt,       //选项
934                    OS_ERR        *p_err)     //返回错误类型
935 {
936     CPU_TS   ts;
937
938
939
940 #ifdef OS_SAFETY_CRITICAL                    //如果使能（默认禁用）了安全检测
941     if (p_err == (OS_ERR *)0) {              //如果错误类型实参为空
942         OS_SAFETY_CRITICAL_EXCEPTION();      //执行安全检测异常函数
943         return;                              //返回，停止执行
944     }
945 #endif
946
```

```
947 #if OS_CFG_ARG_CHK_EN > 0u                    //如果使能了参数检测
948     switch (opt) {                           //根据选项分类处理
949         case OS_OPT_POST_FIFO:               //如果选项在预期内
950         case OS_OPT_POST_LIFO:
951         case OS_OPT_POST_FIFO | OS_OPT_POST_NO_SCHED:
952         case OS_OPT_POST_LIFO | OS_OPT_POST_NO_SCHED:
953              break;                          //直接跳出
954
955         default:                             //如果选项超出预期
956             *p_err = OS_ERR_OPT_INVALID;     //错误类型为“选项非法”
957              return;                         //返回，停止执行
958     }
959 #endif
960
961     ts = OS_TS_GET();                                    //获取时间戳
962
963 #if OS_CFG_ISR_POST_DEFERRED_EN > 0u                     //如果使能了中断延迟发布
964     if (OSIntNestingCtr > (OS_NESTING_CTR)0) {           //如果该函数在中断中被调用
965         OS_IntQPost((OS_OBJ_TYPE)OS_OBJ_TYPE_TASK_MSG,   //将消息先发布到中断消息队列
966                     (void      *)p_tcb,
967                     (void      *)p_void,
968                     (OS_MSG_SIZE)msg_size,
969                     (OS_FLAGS   )0,
970                     (OS_OPT     )opt,
971                     (CPU_TS     )ts,
972                     (OS_ERR    *)p_err);
973         return;                                          //返回
974     }
975 #endif
976
977     OS_TaskQPost(p_tcb,                                  //将消息直接发布
978                  p_void,
979                  msg_size,
980                  opt,
981                  ts,
982                  p_err);
983 }
984 #endif
```

图 12-2 OSTaskQPost () 函数

其实，不管是否使能了中断延迟发布，最终都是调用 OS_TaskQPost () 函数进行发布任务消息。只是使能了中断延迟发布的发布过程会比较曲折，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。OS_TaskQPost () 函数的定义位于“os_task.c”。

app_cfg.h | app.c | os_core.c | os_task.c

```
2126 #if OS_CFG_TASK_Q_EN > 0u                    //如果使能了任务消息队列
2127 void  OS_TaskQPost (OS_TCB       *p_tcb,     //目标任务
2128                     void         *p_void,    //消息内容地址
2129                     OS_MSG_SIZE   msg_size,  //消息长度
2130                     OS_OPT        opt,       //选项
2131                     CPU_TS        ts,        //时间戳
2132                     OS_ERR       *p_err)     //返回错误类型
2133 {
2134     CPU_SR_ALLOC();  //使用到临界段（在关/开中断时）时必需该宏，该宏声明和
2135                      //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
2136                      // SR（临界段关中断只需保存SR），开中断时将该值还原。
2137
```

```
2138     OS_CRITICAL_ENTER();                                        //进入临界段
2139     if (p_tcb == (OS_TCB *)0) {                                 //如果 p_tcb 为空
2140         p_tcb = OSTCBCurPtr;                                    //目标任务为自身
2141     }
2142    *p_err  = OS_ERR_NONE;                                       //错误类型为“无错误”
2143     switch (p_tcb->TaskState) {                                 //根据任务状态分类处理
2144         case OS_TASK_STATE_RDY:                                 //如果目标任务没等待状态
2145         case OS_TASK_STATE_DLY:
2146         case OS_TASK_STATE_SUSPENDED:
2147         case OS_TASK_STATE_DLY_SUSPENDED:
2148              OS_MsgQPut(&p_tcb->MsgQ,                           //把消息放入任务消息队列
2149                         p_void,
2150                         msg_size,
2151                         opt,
2152                         ts,
2153                         p_err);
2154              OS_CRITICAL_EXIT();                                //退出临界段
2155              break;                                             //跳出
2156
2157         case OS_TASK_STATE_PEND:                                //如果目标任务有等待状态
2158         case OS_TASK_STATE_PEND_TIMEOUT:
2159         case OS_TASK_STATE_PEND_SUSPENDED:
2160         case OS_TASK_STATE_PEND_TIMEOUT_SUSPENDED:
2161              if (p_tcb->PendOn == OS_TASK_PEND_ON_TASK_Q) {//如果等的是任务消息队列
2162                  OS_Post((OS_PEND_OBJ *)0,                      //把消息发布给目标任务
2163                          p_tcb,
2164                          p_void,
2165                          msg_size,
2166                          ts);
2167                  OS_CRITICAL_EXIT_NO_SCHED();                   //退出临界段（无调度）
2168                  if ((opt & OS_OPT_POST_NO_SCHED) == (OS_OPT)0u) { //如果要调度任务
2169                      OSSched();                                 //调度任务
2170                  }
2171              } else {                                           //如果没在等待任务消息队列
2172                  OS_MsgQPut(&p_tcb->MsgQ,                       //把消息放入任务消息队列
2173                             p_void,
2174                             msg_size,
2175                             opt,
2176                             ts,
2177                             p_err);
2178                  OS_CRITICAL_EXIT();                            //退出临界段
2179              }
2180              break;                                             //跳出
2181
2182         default:                                                //如果状态超出预期
2183              OS_CRITICAL_EXIT();                                //退出临界段
2184             *p_err = OS_ERR_STATE_INVALID;                      //错误类型为“状态非法”
2185              break;                                             //跳出
2186     }
2187 }
2188 #endif
```

图 12-3 OS_QPost () 函数

在 OS_TaskQPost () 函数中，会调用 OS_MsgQPut () 函数从消息池获取一个消息插入到消息队列。OS_MsgQPut () 函数的定义位于“os_msg.c”。

```
287 void  OS_MsgQPut (OS_MSG_Q      *p_msg_q,     //消息队列指针
288                  void          *p_void,      //消息指针
289                  OS_MSG_SIZE    msg_size,    //消息大小（单位：字节）
290                  OS_OPT         opt,         //选项
291                  CPU_TS         ts,          //消息被发布时的时间戳
292                  OS_ERR        *p_err)       //返回错误类型
293 {
294     OS_MSG  *p_msg;
295     OS_MSG  *p_msg_in;
296
297
298
299 #ifdef OS_SAFETY_CRITICAL                    //如果使能了安全检测
300     if (p_err == (OS_ERR *)0) {              //如果错误类型实参为空
301         OS_SAFETY_CRITICAL_EXCEPTION();      //执行安全检测异常函数
302         return;                              //返回，停止执行
303     }
304 #endif
```

```
305
306     if (p_msg_q->NbrEntries >= p_msg_q->NbrEntriesSize) { //如果消息队列已没有可用空间
307        *p_err = OS_ERR_Q_MAX;                             //错误类型为“队列已满”
308         return;                                           //返回，停止执行
309     }
310
311     if (OSMsgPool.NbrFree == (OS_MSG_QTY)0) {  //如果消息池没有可用消息
312        *p_err = OS_ERR_MSG_POOL_EMPTY;         //错误类型为“消息池没有消息”
313         return;                                //返回，停止执行
314     }
315     /* 从消息池获取一个消息（暂存于 p_msg ）*/
316     p_msg             = OSMsgPool.NextPtr;           //将消息控制块从消息池移除
317     OSMsgPool.NextPtr = p_msg->NextPtr;              //指向下一个消息（取走首个消息）
318     OSMsgPool.NbrFree--;                             //消息池可用消息数减1
319     OSMsgPool.NbrUsed++;                             //消息池被用消息数加1
320     if (OSMsgPool.NbrUsedMax < OSMsgPool.NbrUsed) { //更新消息被用最大数目的历史记录
321         OSMsgPool.NbrUsedMax = OSMsgPool.NbrUsed;
322     }
```

```
323     /* 将获取的消息插入到消息队列 */
324     if (p_msg_q->NbrEntries == (OS_MSG_QTY)0) {              //如果消息队列目前没有消息
325         p_msg_q->InPtr         = p_msg;                      //将其入队指针指向该消息
326         p_msg_q->OutPtr        = p_msg;                      //出队指针也指向该消息
327         p_msg_q->NbrEntries    = (OS_MSG_QTY)1;              //队列的消息数为1
328         p_msg->NextPtr         = (OS_MSG *)0;                //该消息的下一个消息为空
329     } else {                                                 //如果消息队列目前已有消息
330         if ((opt & OS_OPT_POST_LIFO) == OS_OPT_POST_FIFO) { //如果用FIFO方式插入队列，
331             p_msg_in           = p_msg_q->InPtr;             //将消息插入到入队端，入队
332             p_msg_in->NextPtr  = p_msg;                      //指针指向该消息。
333             p_msg_q->InPtr     = p_msg;
334             p_msg->NextPtr     = (OS_MSG *)0;
335         } else {                                             //如果用LIFO方式插入队列，
336             p_msg->NextPtr     = p_msg_q->OutPtr;            //将消息插入到出队端，出队
337             p_msg_q->OutPtr    = p_msg;                      //指针指向该消息。
338         }
339         p_msg_q->NbrEntries++;                               //消息队列的消息数目加1
340     }
341     if (p_msg_q->NbrEntriesMax < p_msg_q->NbrEntries) {      //更新改消息队列的最大消息
342         p_msg_q->NbrEntriesMax = p_msg_q->NbrEntries;        //数目的历史记录。
343     }
344     p_msg->MsgPtr  = p_void;                                 //给该消息填写消息内容
345     p_msg->MsgSize = msg_size;                               //给该消息填写消息大小
346     p_msg->MsgTS   = ts;                                     //填写发布该消息时的时间戳
347    *p_err          = OS_ERR_NONE;                            //错误类型为“无错误”
348 }
```

图 12-4 OS_MsgQPut () 函数

另外，OS_TaskQPost () 函数还调用了 OS_Post() 函数发布内核对象。OS_Post() 函数是一个底层的发布函数，它不仅仅用来发布任务消息队列，还可以发布多值信号量、互斥信号量、事件标志组、（普通）消息队列或任务信号量。OS_Post() 函数的定义位于“os_core.c”。

os_sem.c | os.h | os_core.c | os_cfg.h

```
1844 void  OS_Post (OS_PEND_OBJ  *p_obj,      //内核对象类型指针
1845                OS_TCB       *p_tcb,      //任务控制块
1846                void         *p_void,     //消息
1847                OS_MSG_SIZE   msg_size,   //消息大小
1848                CPU_TS        ts)         //时间戳
1849 {
1850     switch (p_tcb->TaskState) {                                  //根据任务状态分类处理
1851         case OS_TASK_STATE_RDY:                                  //如果任务处于就绪状态
1852         case OS_TASK_STATE_DLY:                                  //如果任务处于延时状态
1853         case OS_TASK_STATE_SUSPENDED:                            //如果任务处于挂起状态
1854         case OS_TASK_STATE_DLY_SUSPENDED:                        //如果任务处于延时中被挂起状态
1855              break;                                              //不用处理，直接跳出

1856
1857         case OS_TASK_STATE_PEND:                                 //如果任务处于无期限等待状态
1858         case OS_TASK_STATE_PEND_TIMEOUT:                         //如果任务处于有期限等待状态
1859              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
1860                  OS_Post1(p_obj,                                 //标记哪个内核对象被发布
1861                           p_tcb,
1862                           p_void,
1863                           msg_size,
1864                           ts);
1865              } else {                                            //如果任务不是在等待多个信号量或消息队列
1866 #if (OS_MSG_EN > 0u)                                             //如果使能了任务队列或消息队列
1867                  p_tcb->MsgPtr  = p_void;                        //保存消息到等待任务
1868                  p_tcb->MsgSize = msg_size;
1869 #endif
1870                  p_tcb->TS      = ts;                            //保存时间戳到等待任务
1871              }
1872              if (p_obj != (OS_PEND_OBJ *)0) {                    //如果内核对象为空
1873                  OS_PendListRemove(p_tcb);                       //从等待列表移除该等待任务
1874 #if OS_CFG_DBG_EN > 0u                                           //如果使能了调试代码和变量
1875                  OS_PendDbgNameRemove(p_obj,                     //移除内核对象的调试名
1876                                       p_tcb);
1877 #endif
1878              }
1879              OS_TaskRdy(p_tcb);                                  //让该等待任务准备运行
1880              p_tcb->TaskState  = OS_TASK_STATE_RDY;              //任务状态改为就绪状态
1881              p_tcb->PendStatus = OS_STATUS_PEND_OK;              //清除等待状态
1882              p_tcb->PendOn     = OS_TASK_PEND_ON_NOTHING;        //标记不再等待
1883              break;

1884
1885         case OS_TASK_STATE_PEND_SUSPENDED:                       //如果任务在无期限等待中被挂起
1886         case OS_TASK_STATE_PEND_TIMEOUT_SUSPENDED:               //如果任务在有期限等待中被挂起
1887              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
1888                  OS_Post1(p_obj,                                 //标记哪个内核对象被发布
1889                           p_tcb,
1890                           p_void,
1891                           msg_size,
1892                           ts);
1893              } else {                                            //如果任务不在等待多个信号量或消息队列
1894 #if (OS_MSG_EN > 0u)                                             //如果使能了调试代码和变量
1895                  p_tcb->MsgPtr  = p_void;                        //保存消息到等待任务
1896                  p_tcb->MsgSize = msg_size;
1897 #endif
1898                  p_tcb->TS      = ts;                            //保存时间戳到等待任务
1899              }
1900              OS_TickListRemove(p_tcb);                           //从节拍列表移除该等待任务
1901              if (p_obj != (OS_PEND_OBJ *)0) {                    //如果内核对象为空
1902                  OS_PendListRemove(p_tcb);                       //从等待列表移除该等待任务
1903 #if OS_CFG_DBG_EN > 0u                                           //如果使能了调试代码和变量
1904                  OS_PendDbgNameRemove(p_obj,                     //移除内核对象的调试名
1905                                       p_tcb);
1906 #endif
1907              }
1908              p_tcb->TaskState  = OS_TASK_STATE_SUSPENDED;        //任务状态改为被挂起状态
1909              p_tcb->PendStatus = OS_STATUS_PEND_OK;              //清除等待状态
1910              p_tcb->PendOn     = OS_TASK_PEND_ON_NOTHING;        //标记不再等待
1911              break;
1912
1913         default:                                                 //如果任务状态超出预期
1914              break;                                              //直接跳出
1915     }
1916 }
```

图 12-5 OS_Post() 函数

12.1.2 OSTaskQPend ()

与 OSTaskQPost () 任务消息队列发布消息函数相对应，OSTaskQPend () 函数用于等待获取任务消息队列的消息。

表 40 OSTaskQPend ()

函数原型	void *OSTaskQPend (OS_TICK timeout, OS_OPT opt, OS_MSG_SIZE *p_msg_size, CPU_TS *p_ts, OS_ERR *p_err);			
功能	等待任务消息队列的消息。			
	timeout	等待超时时间（单位：时钟节拍），0 代表无期限等待。opt 为 OS_OPT_PEND_BLOCKING 时该参数才起作用。		
	opt	选项	OS_OPT_PEND_BLOCKING	如果不能立即获得消息，就堵塞当前任务，继续等待消息。
			OS_OPT_PEND_NON_BLOCKING	如果不能立即获得消息，不堵塞当前任务，不继续等待消息。
	p_msg_size	消息长度。		
	p_ts	时间戳	用于存储任务消息队列最后一次被发布消息的时间戳。该参数可以为 NULL，表示用户不需要获得时间戳。	
	p_err	返回错误类型	OS_ERR_NONE	没错误，任务成功获得消息。
			OS_ERR_PTR_INVALID	p_msg_size 为空。
			OS_ERR_PEND_ABORT	等待被中止。
			OS_ERR_PEND_ISR	在中断中被调用。
			OS_ERR_PEND_WOULD_BLOCK	缺乏堵塞。
			OS_ERR_Q_EMPTY	任务消息队列里没有消息
			OS_ERR_SCHED_LOCKED	调度器被锁。
			OS_ERR_TIMEOUT	等待超时。
返回值	✧ != NULL，接收到的消息的指针（首地址）。 ✧ == NULL，接收到一个空消息，或者有错误。			
注意事项	✧ 不可以在中断中调用该函数。			

OSTaskQPend () 函数的定义也位于“os_task.c”。

```
671 #if OS_CFG_TASK_Q_EN > 0u                          //如果使能了任务消息队列
672 void  *OSTaskQPend (OS_TICK        timeout,      //等待期限（单位：时钟节拍）
673                     OS_OPT         opt,          //选项
674                     OS_MSG_SIZE   *p_msg_size,   //返回消息长度
675                     CPU_TS        *p_ts,         //返回时间戳
676                     OS_ERR        *p_err)        //返回错误类型
677 {
678     OS_MSG_Q     *p_msg_q;
679     void         *p_void;
680     CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必需该宏，该宏声明和
681                     //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
682                     // SR（临界段关中断只需保存SR），开中断时将该值还原。
683
684 #ifdef OS_SAFETY_CRITICAL                         //如果使能（默认禁用）了安全检测
685     if (p_err == (OS_ERR *)0) {                   //如果错误类型实参为空
686         OS_SAFETY_CRITICAL_EXCEPTION();           //执行安全检测异常函数
687         return ((void *)0);                       //返回0（有错误），停止执行
688     }
689 #endif
690
691 #if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u            //如果使能了中断中非法调用检测
692     if (OSIntNestingCtr > (OS_NESTING_CTR)0) {    //如果该函数在中断中被调用
693        *p_err = OS_ERR_PEND_ISR;                  //错误类型为“在中断中中止等待”
694         return ((void *)0);                       //返回0（有错误），停止执行
695     }
696 #endif

697
698 #if OS_CFG_ARG_CHK_EN > 0u                        //如果使能了参数检测
699     if (p_msg_size == (OS_MSG_SIZE *)0) {         //如果 p_msg_size 为空
700        *p_err = OS_ERR_PTR_INVALID;               //错误类型为“指针不可用”
701         return ((void *)0);                       //返回0（有错误），停止执行
702     }
703     switch (opt) {                                //根据选项分类处理
704         case OS_OPT_PEND_BLOCKING:                //如果选项在预期内
705         case OS_OPT_PEND_NON_BLOCKING:
706              break;                               //直接跳出
707
708         default:                                  //如果选项超出预期
709             *p_err = OS_ERR_OPT_INVALID;          //错误类型为“选项非法”
710              return ((void *)0);                  //返回0（有错误），停止执行
711     }
712 #endif

713
714     if (p_ts != (CPU_TS *)0) {    //如果 p_ts 非空
715        *p_ts  = (CPU_TS  )0;      //初始化（清零）p_ts，待用于返回时间戳
716     }
717
718     CPU_CRITICAL_ENTER();                                  //关中断
719     p_msg_q = &OSTCBCurPtr->MsgQ;                          //获取当前任务的消息队列
720     p_void  = OS_MsgQGet(p_msg_q,                          //从队列里获取一个消息
721                          p_msg_size,
722                          p_ts,
723                          p_err);
724     if (*p_err == OS_ERR_NONE) {                          //如果获取消息成功
725 #if OS_CFG_TASK_PROFILE_EN > 0u                          //如果使能了任务控制块的简况变量
726         if (p_ts != (CPU_TS *)0) {                                    //如果 p_ts
727             OSTCBCurPtr->MsgQPendTime = OS_TS_GET() - *p_ts;          //非空，更新
728             if (OSTCBCurPtr->MsgQPendTimeMax < OSTCBCurPtr->MsgQPendTime) {//等待任务消
729                 OSTCBCurPtr->MsgQPendTimeMax = OSTCBCurPtr->MsgQPendTime;  //息队列的最
730             }                                                         //长时间记录。
731         }
732 #endif
733         CPU_CRITICAL_EXIT();                          //开中断
734         return (p_void);                              //返回消息内容
735     }
```

```
736     /* 如果获取消息不成功（队列里没有消息） */
737     if ((opt & OS_OPT_PEND_NON_BLOCKING) != (OS_OPT)0) { //如果选择了不堵塞任务
738        *p_err = OS_ERR_PEND_WOULD_BLOCK;                 //错误类型为“缺乏阻塞”
739         CPU_CRITICAL_EXIT();                             //开中断
740         return ((void *)0);                              //返回0（有错误），停止执行
741     } else {                                             //如果选择了堵塞任务
742         if (OSSchedLockNestingCtr > (OS_NESTING_CTR)0) { //如果调度器被锁
743             CPU_CRITICAL_EXIT();                         //开中断
744            *p_err = OS_ERR_SCHED_LOCKED;                 //错误类型为“调度器被锁”
745             return ((void *)0);                          //返回0（有错误），停止执行
746         }
747     }

748     /* 如果调度器未被锁 */
749     OS_CRITICAL_ENTER_CPU_EXIT();                        //锁调度器，重开中断
750     OS_Pend((OS_PEND_DATA *)0,                           //阻塞当前任务，等待消息
751             (OS_PEND_OBJ  *)0,
752             (OS_STATE      )OS_TASK_PEND_ON_TASK_Q,
753             (OS_TICK       )timeout);
754     OS_CRITICAL_EXIT_NO_SCHED();                         //解锁调度器（无调度）
755
756     OSSched();                                           //调度任务

757     /* 当前任务（获得消息队列的消息）得以继续运行 */
758     CPU_CRITICAL_ENTER();                               //关中断
759     switch (OSTCBCurPtr->PendStatus) {                  //根据任务的等待状态分类处理
760         case OS_STATUS_PEND_OK:                         //如果任务已成功获得消息
761              p_void      = OSTCBCurPtr->MsgPtr;         //提取消息内容地址
762             *p_msg_size  = OSTCBCurPtr->MsgSize;        //提取消息长度
763              if (p_ts != (CPU_TS *)0) {                 //如果 p_ts 非空
764                 *p_ts  = OSTCBCurPtr->TS;               //获取任务等到消息时的时间戳
765 #if OS_CFG_TASK_PROFILE_EN > 0u                         //如果使能了任务控制块的简况变量
766                  OSTCBCurPtr->MsgQPendTime = OS_TS_GET() - OSTCBCurPtr->TS;     //更新等待
767                  if (OSTCBCurPtr->MsgQPendTimeMax < OSTCBCurPtr->MsgQPendTime) {//任务消息
768                      OSTCBCurPtr->MsgQPendTimeMax = OSTCBCurPtr->MsgQPendTime;  //队列的最
769                  }                                                              //长时间记
770 #endif                                                                         //录。
771              }
772             *p_err = OS_ERR_NONE;                       //错误类型为“无错误”
773              break;                                     //跳出
774

775         case OS_STATUS_PEND_ABORT:                      //如果等待被中止
776              p_void      = (void      *)0;               //返回消息内容为空
777             *p_msg_size = (OS_MSG_SIZE)0;               //返回消息大小为0
778              if (p_ts  != (CPU_TS *)0) {                //如果 p_ts 非空
779                 *p_ts   = (CPU_TS  )0;                  //清零 p_ts
780              }
781             *p_err       =  OS_ERR_PEND_ABORT;          //错误类型为“等待被中止”
782              break;                                     //跳出
783
784         case OS_STATUS_PEND_TIMEOUT:                    //如果等待超时，
785         default:                                        //或者任务状态超出预期。
786              p_void      = (void      *)0;               //返回消息内容为空
787             *p_msg_size = (OS_MSG_SIZE)0;               //返回消息大小为0
788              if (p_ts  != (CPU_TS *)0) {                //如果 p_ts 非空
789                 *p_ts   =  OSTCBCurPtr->TS;
790              }
791             *p_err       =  OS_ERR_TIMEOUT;             //错误类为“等待超时”
792              break;                                     //跳出
793     }
794     CPU_CRITICAL_EXIT();                                //开中断
795     return (p_void);                                    //返回消息内容地址
796 }
797 #endif
```

图 12-6 OSTaskQPend () 函数

在 OSTaskQPend () 函数中，会调用 OS_MsgQGet () 函数从任务消息队列获取一个消息。OS_MsgQGet () 函数的定义位于“os_msg.c”。

```
203 void  *OS_MsgQGet (OS_MSG_Q      *p_msg_q,      //消息队列
204                   OS_MSG_SIZE  *p_msg_size,   //返回消息大小
205                   CPU_TS       *p_ts,         //返回某些操作的时间戳
206                   OS_ERR       *p_err)        //返回错误类型
207 {
208     OS_MSG  *p_msg;
209     void    *p_void;
210
211
212
213 #ifdef OS_SAFETY_CRITICAL                     //如果使能（默认禁用）了安全检测
214     if (p_err == (OS_ERR *)0) {               //如果错误类型实参为空
215         OS_SAFETY_CRITICAL_EXCEPTION();       //执行安全检测异常函数
216         return ((void *)0);                   //返回空消息，停止执行
217     }
218 #endif
219
220     if (p_msg_q->NbrEntries == (OS_MSG_QTY)0) {  //如果消息队列没有消息
221        *p_msg_size = (OS_MSG_SIZE)0;             //返回消息长度为0
222         if (p_ts != (CPU_TS *)0) {               //如果 p_ts 非空
223            *p_ts  = (CPU_TS  )0;                 //清零 p_ts
224         }
225        *p_err = OS_ERR_Q_EMPTY;                  //错误类型为“队列没消息”
226         return ((void *)0);                      //返回空消息，停止执行
227     }
228     /* 如果消息队列有消息 */
229     p_msg           = p_msg_q->OutPtr;           //从队列的出口端提取消息
230     p_void          = p_msg->MsgPtr;             //提取消息内容
231    *p_msg_size      = p_msg->MsgSize;            //提取消息长度
232     if (p_ts != (CPU_TS *)0) {                  //如果 p_ts 非空
233        *p_ts  = p_msg->MsgTS;                   //获取消息被发布时的时间戳
234     }
235
236     p_msg_q->OutPtr = p_msg->NextPtr;           //修改队列的出队指针
237
238     if (p_msg_q->OutPtr == (OS_MSG *)0) {       //如果队列没有消息了
239         p_msg_q->InPtr      = (OS_MSG   *)0;    //清零出队指针
240         p_msg_q->NbrEntries = (OS_MSG_QTY)0;    //清零消息数
241     } else {                                    //如果队列还有消息
242         p_msg_q->NbrEntries--;                  //队列的消息数减1
243     }
244     /* 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 */
245     p_msg->NextPtr     = OSMsgPool.NextPtr;     //消息插回消息池
246     OSMsgPool.NextPtr = p_msg;
247     OSMsgPool.NbrFree++;                        //消息池的可用消息数加1
248     OSMsgPool.NbrUsed--;                        //消息池的已用消息数减1
249
250    *p_err              = OS_ERR_NONE;           //错误类型为“无错误”
251     return (p_void);                           //返回罅隙内容
252 }
```

图 12-7 OS_MsgQGet () 函数

OSTaskQPend () 函数会调用一个更加底层的等待函数来执行当前任务对消息队列的等待，该函数就是 OS_Pend()。与 OS_Post() 函数一样，OS_Pend() 函数不仅仅用来等待任务消息队列，还可以等待多值信号量、互斥信号量、事件标志组、（普通）消息队列或任务信号量。OS_Pend() 函数的定义位于“os_core.c”。

```
os_sem.c  os.h  os_core.c  os_cfg.h  os_type.h
873 void  OS_Pend (OS_PEND_DATA  *p_pend_data,   //待插入等待列表的元素
874               OS_PEND_OBJ   *p_obj,         //等待的内核对象
875               OS_STATE       pending_on,    //等待哪种对象内核
876               OS_TICK        timeout)       //等待期限
877 {
878     OS_PEND_LIST  *p_pend_list;
879
880
881
882     OSTCBCurPtr->PendOn     = pending_on;                    //资源不可用，开始等待
883     OSTCBCurPtr->PendStatus = OS_STATUS_PEND_OK;             //正常等待中
884
885     OS_TaskBlock(OSTCBCurPtr,                                //阻塞当前运行任务，
886                  timeout);                                   //如果 timeout 非0，把任务插入的节拍列表
887
888     if (p_obj != (OS_PEND_OBJ *)0) {                         //如果等待对象非空
889         p_pend_list           = &p_obj->PendList;            //获取对象的等待列表到 p_pend_list
890         p_pend_data->PendObjPtr = p_obj;                     //保存要等待的对象
891         OS_PendDataInit((OS_TCB      *)OSTCBCurPtr,          //初始化 p_pend_data（待插入等待列表）
892                         (OS_PEND_DATA *)p_pend_data,
893                         (OS_OBJ_QTY    )1);
894         OS_PendListInsertPrio(p_pend_list,                   //按优先级将 p_pend_data 插入到等待列表
895                               p_pend_data);
896     } else {                                                 //如果等待对象为空
897         OSTCBCurPtr->PendDataTblEntries = (OS_OBJ_QTY    )0; //清零当前任务的等待域数据
898         OSTCBCurPtr->PendDataTblPtr     = (OS_PEND_DATA *)0;
899     }
900 #if OS_CFG_DBG_EN > 0u                                       //如果使能了调试代码和变量
901     OS_PendDbgNameAdd(p_obj,                                 //更新信号量的 DbgNamePtr 元素为其等待
902                       OSTCBCurPtr);                          //列表中优先级最高的任务的名称。
903 #endif
904 }
```

图 12-8 OS_Pend() 函数

12.1.3 OSTaskQPendAbort ()

OSTaskQPendAbort () 函数用于中止任务对其任务消息队列的等待。要使用 OSTaskQPendAbort () 函数，除了要先使能前面的 OS_CFG_TASK_Q_EN 外，还得使能 OS_CFG_TASK_Q_PEND_ABORT_EN（位于“os_cfg.h”），如下图所示。

```
_cfg.h  app.c  os_core.c  os_task.c  os_cfg.h  os_msg.c
82                                              /* ------------------------ TASK MANAGEMENT -----
83 #define OS_CFG_STAT_TASK_EN             1u   //使能/禁用统计任务
84 #define OS_CFG_STAT_TASK_STK_CHK_EN     1u   //使能/禁用从统计任务检查堆栈
85
86 #define OS_CFG_TASK_CHANGE_PRIO_EN      1u   //使能/禁用函数 OSTaskChangePrio()
87 #define OS_CFG_TASK_DEL_EN              1u   //使能/禁用函数 OSTaskDel()
88 #define OS_CFG_TASK_Q_EN                1u   //使能/禁用函数 OSTaskQXXXX()
89 #define OS_CFG_TASK_Q_PEND_ABORT_EN     1u   //使能/禁用函数 OSTaskQPendAbort()
90 #define OS_CFG_TASK_PROFILE_EN          1u   //使能/禁用任务控制块的简况变量
91 #define OS_CFG_TASK_REG_TBL_SIZE        1u   //定义任务指定的寄存器数目
92 #define OS_CFG_TASK_SEM_PEND_ABORT_EN   1u   //使能/禁用函数 OSTaskSemPendAbort()
93 #define OS_CFG_TASK_SUSPEND_EN          1u   //使能/禁用函数 OSTaskSuspend() 和 OSTaskResume()
94
```

图 12-9

OSTaskQPendAbort () 函数的信息如下表所示。

表 41 OSTaskQPendAbort ()

函数原型	CPU_BOOLEAN OSTaskQPendAbort (OS_TCB *p_tcb, OS_OPT opt, OS_ERR *p_err);	
功能	中止一个任务对其消息队列的等待。	
参数	p_tcb	目标任务。该参数不能为 NULL 或当前运行任务。

	opt	选项。	OS_OPT_POST_NONE	没有选项要求。
			OS_OPT_POST_NO_SCHED	不进行任务调度。
	p_err	返回错误类型	OS_ERR_NONE	无错误，中止成功。
			OS_ERR_OPT_INVALID	选项非法。
			OS_ERR_PEND_ABORT_ISR	该函数在中断中被调用。
			OS_ERR_PEND_ABORT_NONE	目标任务没在等待任务消息队列。
			OS_ERR_PEND_ABORT_SELF	目标任务是自身（当前运行任务）。
返回值	✧ == DEF_FALSE，目标任务没在等待任务消息队列，或有错误。 ✧ == DEF_TRUE，目标任务是在等待任务消息队列，而且等待被中止。			
注意事项	✧ 不可以在中断中调用该函数。 ✧ p_tcb 不能为 NULL 或当前运行任务。			

OSTaskQPendAbort () 函数的定义位于“os_task.c”。

```
827 #if (OS_CFG_TASK_Q_EN > 0u) && (OS_CFG_TASK_Q_PEND_ABORT_EN > 0u)//如果使能了该函数
828 CPU_BOOLEAN  OSTaskQPendAbort (OS_TCB  *p_tcb,                   //目标任务
829                                OS_OPT   opt,                     //选项
830                                OS_ERR  *p_err)                   //返回错误类型
831 {
832     CPU_TS         ts;
833     CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必需该宏，该宏声明和
834                     //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
835                     // SR（临界段关中断只需保存SR），开中断时将该值还原。
836
837 #ifdef OS_SAFETY_CRITICAL                 //如果使能（默认禁用）了安全检测
838     if (p_err == (OS_ERR *)0) {           //如果错误类型实参为空
839         OS_SAFETY_CRITICAL_EXCEPTION(); //执行安全检测异常函数
840         return (DEF_FALSE);               //返回（有错误），停止执行
841     }
842 #endif
843
844 #if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u          //如果使能了中断中非法调用检测
845     if (OSIntNestingCtr > (OS_NESTING_CTR)0) { //如果该函数在中断中被调用
846        *p_err = OS_ERR_PEND_ABORT_ISR;          //错误类型为“在中断中中止等待”
847         return (DEF_FALSE);                     //返回（有错误），停止执行
848     }
849 #endif
850
```

```
#if OS_CFG_ARG_CHK_EN > 0u                          //如果使能了参数检测
    switch (opt) {                                  //根据选项分类处理
        case OS_OPT_POST_NONE:                      //如果选项在预期内
        case 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 (DEF_FALSE);                    //返回（有错误），停止执行
    }
#endif

    CPU_CRITICAL_ENTER();                                   //关中断
#if OS_CFG_ARG_CHK_EN > 0u                                  //如果使能了参数检测
    if ((p_tcb == (OS_TCB *)0) ||                           //如果目标任务为空，
        (p_tcb == OSTCBCurPtr)) {                           //或是自身。
        CPU_CRITICAL_EXIT();                                //开中断
       *p_err = OS_ERR_PEND_ABORT_SELF;                     //错误类型为“中止自身”
        return (DEF_FALSE);                                 //返回（有错误），停止执行
    }
#endif

    if (p_tcb->PendOn != OS_TASK_PEND_ON_TASK_Q) {          //如果任务没在等待任务消息队列
        CPU_CRITICAL_EXIT();                                //关中断
       *p_err = OS_ERR_PEND_ABORT_NONE;                     //错误类型为“中止对象有误”
        return (DEF_FALSE);                                 //返回（有错误），停止执行
    }
    /* 如果任务是在等待任务消息队列 */
    OS_CRITICAL_ENTER_CPU_EXIT();                           //锁调度器，重开中断
    ts = OS_TS_GET();                                       //获取时间戳作为中止时间
    OS_PendAbort((OS_PEND_OBJ *)0,                          //中止任务对其消息队列的等待
                 p_tcb,
                 ts);
    OS_CRITICAL_EXIT_NO_SCHED();                            //解锁调度器（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
    }
   *p_err = OS_ERR_NONE;                                    //错误类型为“无错误”
    return (DEF_TRUE);                                      //返回（中止成功）
}
#endif
```

图 12-10 OSTaskQPendAbort () 函数

OSTaskQPendAbort () 函数会调用一个更加底层的中止等待函数来执行当前任务对其任务消息队列的等待，该函数就是 OS_PendAbort()。OS_PendAbort() 函数不仅仅用来中止对任务消息队列的等待，还可以中止对多值信号量、互斥信号量、事件标志组、（普通）消息队列或任务信号量的等待。OS_PendAbort() 函数的定义位于“os_core.c”。

os_cfg.h | os_sem.c | os.h | os_core.c

```
void  OS_PendAbort (OS_PEND_OBJ  *p_obj,    //被等待对象的类型
                    OS_TCB       *p_tcb,    //任务控制块指针
                    CPU_TS        ts)       //等待被中止时的时间戳
{
    switch (p_tcb->TaskState) {                             //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;                                         //这些情况均与等待无关，直接跳出
```

```
938         case OS_TASK_STATE_PEND:                                    //如果任务是无期限等待状态
939         case OS_TASK_STATE_PEND_TIMEOUT:                            //如果任务是有期限等待状态
940             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
941                 OS_PendAbort1(p_obj,                                //强制解除任务对某一对象的等待
942                               p_tcb,
943                               ts);
944             }
945 #if (OS_MSG_EN > 0u)                                                //如果使能了任务队列或消息队列
946             p_tcb->MsgPtr     = (void      *)0;                     //清除（复位）任务的消息域
947             p_tcb->MsgSize    = (OS_MSG_SIZE)0u;
948 #endif
949             p_tcb->TS         = ts;                                 //保存等待被中止时的时间戳到任务控制块
950             if (p_obj != (OS_PEND_OBJ *)0) {                        //如果等待对象非空
951                 OS_PendListRemove(p_tcb);                           //将任务从所有等待列表中移除
952             }
953             OS_TaskRdy(p_tcb);                                      //让任务进准备运行
954             p_tcb->TaskState  = OS_TASK_STATE_RDY;                  //修改任务状态为就绪状态
955             p_tcb->PendStatus = OS_STATUS_PEND_ABORT;               //标记任务的等待被中止
956             p_tcb->PendOn     = OS_TASK_PEND_ON_NOTHING;            //标记任务目前没有等待任何对象
957             break;                                                  //跳出
958
959         case OS_TASK_STATE_PEND_SUSPENDED:                          //如果任务在无期限等待中被挂起
960         case OS_TASK_STATE_PEND_TIMEOUT_SUSPENDED:                  //如果任务在有期限等待中被挂起
961             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
962                 OS_PendAbort1(p_obj,                                //强制解除任务对某一对象的等待
963                               p_tcb,
964                               ts);
965             }
966 #if (OS_MSG_EN > 0u)                                                //如果使能了任务队列或消息队列
967             p_tcb->MsgPtr     = (void      *)0;                     //清除（复位）任务的消息域
968             p_tcb->MsgSize    = (OS_MSG_SIZE)0u;
969 #endif
970             p_tcb->TS         = ts;                                 //保存等待被中止时的时间戳到任务控制块
971             if (p_obj != (OS_PEND_OBJ *)0) {                        //如果等待对象非空
972                 OS_PendListRemove(p_tcb);                           //将任务从所有等待列表中移除
973             }
974             OS_TickListRemove(p_tcb);                               //让任务脱离节拍列表
975             p_tcb->TaskState  = OS_TASK_STATE_SUSPENDED;            //修改任务状态为挂起状态
976             p_tcb->PendStatus = OS_STATUS_PEND_ABORT;               //标记任务的等待被中止
977             p_tcb->PendOn     = OS_TASK_PEND_ON_NOTHING;            //标记任务目前没有等待任何对象
978             break;                                                  //跳出
979
980         default:                                                    //如果任务状态超出预期
981             break;                                                  //不需处理，直接跳出
982     }
983 }
```

图 12-11 OS_PendAbort() 函数

12.2 实例演示

12.2.1 实例 1

本实例与前面的（普通）消息队列的实例类似，也是创建两个应用任务，AppTaskPost ()和 AppTaskPend ()，任务 AppTaskPost ()发布消息给任务 AppTaskPend ()。不同的是，前面实例使用的是消息队列，而本章实例使用的是任务消息队列。

该例程已经存放在配套资料的下图路径。

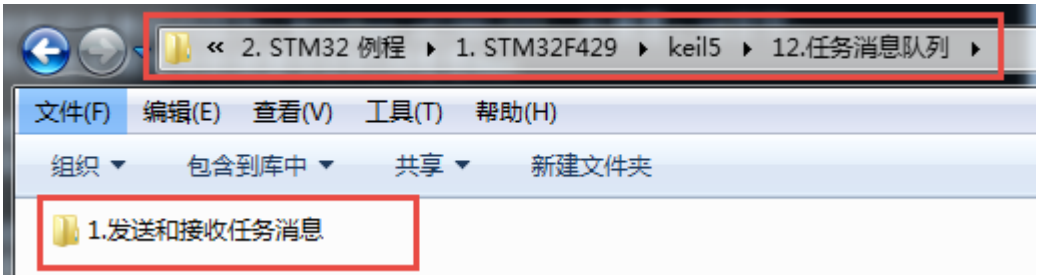

图 12-12 例程路径

本例程需用使用 LED1 和 USART1，所以工程中需要添加其驱动文件和初始化。用户可参照前面相关实例，这里不再赘述。

在起始任务 AppTaskStart() 中，在创建任务 AppTaskPend() 时，要为该任务初始化其任务消息队列的消息最大容量。

```
179
180     /* 创建 AppTaskPend 任务 */
181     OSTaskCreate((OS_TCB     *)&AppTaskPendTCB,                                //任务控制块地址
182                  (CPU_CHAR   *)"App Task Pend",                                //任务名称
183                  (OS_TASK_PTR ) AppTaskPend,                                   //任务函数
184                  (void       *) 0,                                             //传递给任务函数（形参p_arg）的实参
185                  (OS_PRIO     ) APP_TASK_PEND_PRIO,                            //任务的优先级
186                  (CPU_STK    *)&AppTaskPendStk[0],                             //任务堆栈的基地址
187                  (CPU_STK_SIZE) APP_TASK_PEND_STK_SIZE / 10,                   //任务堆栈空间剩下1/10时限制其增长
188                  (CPU_STK_SIZE) APP_TASK_PEND_STK_SIZE,                        //任务堆栈空间（单位：sizeof(CPU_STK)）
189                  (OS_MSG_QTY  ) 50u,                                           //任务可接收的最大消息数
190                  (OS_TICK     ) 0u,                                            //任务的时间片节拍数（0表默认值OSCfg_TickRate_Hz/10）
191                  (void       *) 0,                                             //任务扩展（0表不扩展）
192                  (OS_OPT      )(OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),    //任务选项
193                  (OS_ERR     *)&err);                                          //返回错误类型
194
```

图 12-13 初始化任务消息队列容量

任务函数 AppTaskPost () 的定义如下。任务每隔 1s 向任务 AppTaskPend()发送一个消息。

```
202 *********************************************************************************************************
203 *                                             POST TASK
204 *********************************************************************************************************
205 */
206 static  void  AppTaskPost ( void * p_arg )
207 {
208   OS_ERR      err;
209
210
211   (void)p_arg;
212
213
214   while (DEF_TRUE) {                                                  //任务体
215     /* 发布消息到任务 AppTaskPend */
216     OSTaskQPost ((OS_TCB       *)&AppTaskPendTCB,                     //目标任务的控制块
217                  (void         *)"Binghuo uC/OS-III",                 //消息内容
218                  (OS_MSG_SIZE   )sizeof ( "Binghuo uC/OS-III" ),      //消息长度
219                  (OS_OPT        )OS_OPT_POST_FIFO,                    //发布到任务消息队列的入口端
220                  (OS_ERR       *)&err);                               //返回错误类型
221
222     OSTimeDlyHMSM ( 0, 0, 1, 0, OS_OPT_TIME_DLY, & err );             //每20ms发送一次
223
224   }
225
226 }
```

图 12-14 AppTaskPost () 任务函数

任务函数 AppTaskPend () 的定义如下。任务接收到任务消息后，将切换 LED1 的亮灭状态，并将接收的消息内容和长度打印到串口调试助手，以及消息从被发布到被接收的时间差。

```c
/*
*********************************************************************************************************
*                                              PEND TASK
*********************************************************************************************************
*/
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, //没有消息就阻塞任务
                        (OS_MSG_SIZE    *)&msg_size,          //返回消息长度
                        (CPU_TS         *)&ts,                //返回消息被发布的时间戳
                        (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 ( "\r\n接收到的消息的内容为：%s，长度是：%d字节。",
             pMsg, msg_size );

    printf ( "\r\n任务信号量从被发布到被接收的时间差是%dus\r\n",
              ts / ( cpu_clk_freq / 1000000 ) );

    OS_CRITICAL_EXIT();                                   //退出临界段

  }

}
```

图 12-15 AppTaskPend () 任务函数

把 STM32 的 USART1 连接至电脑的串口调试助手，编译和下载程序到秉火 STM32 开发板，运行程序。用户可以看到串口调试助手上打印任务 AppTaskPend () 接收到的消息长度和内容。用户可以观察到每隔 1 秒 LED1 切换一次亮灭状态，同时串口调试助手上也会打印出接收到的消息内容和长度，以及消息从被发布到被接收的时间差。

图 12-16 串口调试助手

12.3 章末总结

任务消息队列跟消息队列极其相似。本质区别在于，任务信号量时一个任务的特有属性，是某一个任务专有的消息队列，其他任务均可以发布消息到这个任务消息队列，但只有该任务才能接收这个消息。消息队列却是一个独立的内核对象，任何任务均可以发布和接收消息队列的消息。因此，任务消息队列不需像消息队列那样单独创建，只要创建了任务，就同时创建了其任务消息队列。

OSTaskQPost () 函数用于向任务消息队列发布一个消息，OSTaskQPend () 函数用于等待任务消息队列的消息，OSTaskQPendAbort () 函数用于中止任务对其任务消息队列的等待。

第13章 内存管理

一个处理器，在不断地分配和释放内存的过程中，一整块连续的内存被分散为很多离散的小块内存，这些叫做内存碎片，内存碎片过多会导致内存的浪费。uC/OS 的内存管理机制就是为了尽量减少内存碎片。大致的思路是一次性取出一个较大的内存分区，把这个内存分区分成若干个内存块，然后将内存块逐个串成单向链表。每次要用到内存块就从内存分区中取出一块，用完就放回去。这跟消息队列的消息池的使用原理是一样的。

13.1 原理简述

如果想要使用内存管理机制，就必须事先使能内存管理。内存管理的使能位于“os_cfg.h”。

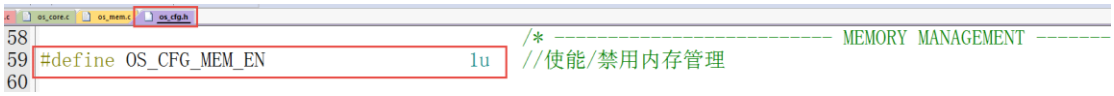

图 13-1

13.1.1 OSMemCreate ()

要使用 uC/OS 的内存管理必须先声明和创建内存管理对象，调用 OSMemCreate () 函数可以创建一个内存管理对象。注意，内存分区一经创建便不能删除，系统没有提供相应的删除函数。OSMemCreate () 函数的信息如下表所示。

表 42 OSMemCreate ()

函数原型	void OSMemCreate (OS_MEM *p_mem, CPU_CHAR *p_name, void *p_addr, OS_MEM_QTY n_blks, OS_MEM_SIZE blk_size, OS_ERR *p_err);			
功能	创建一个内存管理对象。			
参数	p_mem	内存管理对象。		
	p_name	命名内存管理对象。		
	p_addr	内存分区首地址。		
	n_blks	内存块数目，要求不小于 2。		
	blk_size	内存块空间字节数，不少于一个指针的字节数。（STM32 是的指针的字节数是 4）。		
	p_err	返回错误类型	OS_ERR_NONE	无错误，创建成功。
			OS_ERR_CREATE_ISR	在中断中调用该函数
			OS_ERR_ILLEGAL_CREATE_RUN_TIME	在 调 用

				OSSafetyCriticalStart() 函数后创建内核对象
			OS_ERR_MEM_INVALID_BLKS	内存块数目非法。
			OS_ERR_MEM_INVALID_P_ADDR	内存分区地址非法。
			OS_ERR_MEM_INVALID_SIZE	内存空间大小非法。
返回值	无。			
注意事项	✧ 创建前必须先为 p_mem 声明一个内存管理对象（OS_MEM）。 ✧ 不可以在中断中调用该函数。			

OSMemCreate () 函数的定义位于“os_mem.c”。

```
75  void  OSMemCreate (OS_MEM        *p_mem,     //内存分区控制块
76                     CPU_CHAR      *p_name,    //命名内存分区
77                     void          *p_addr,    //内存分区首地址
78                     OS_MEM_QTY     n_blks,    //内存块数目
79                     OS_MEM_SIZE    blk_size,  //内存块大小（单位：字节）
80                     OS_ERR        *p_err)     //返回错误类型
81 {
82 #if OS_CFG_ARG_CHK_EN > 0u
83     CPU_DATA       align_msk;
84 #endif
85     OS_MEM_QTY     i;
86     OS_MEM_QTY     loops;
87     CPU_INT08U    *p_blk;
88     void         **p_link;                  //二级指针，存放指针的指针
89     CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必需该宏，该宏声明和
90                     //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
91                     // SR（临界段关中断只需保存SR），开中断时将该值还原。
92
93 #ifdef OS_SAFETY_CRITICAL                  //如果使能了安全检测
94     if (p_err == (OS_ERR *)0) {            //如果错误类型实参为空
95         OS_SAFETY_CRITICAL_EXCEPTION();    //执行安全检测异常函数
96         return;                            //返回，停止执行
97     }
98 #endif
99
100 #ifdef OS_SAFETY_CRITICAL_IEC61508                //如果使能了安全关键
101     if (OSSafetyCriticalStartFlag == DEF_TRUE) { //如果在调用OSSafetyCriticalStart()后创建
102         *p_err = OS_ERR_ILLEGAL_CREATE_RUN_TIME;  //错误类型为“非法创建内核对象”
103         return;                                   //返回，停止执行
104     }
105 #endif
106
107 #if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u         //如果使能了中断中非法调用检测
108     if (OSIntNestingCtr > (OS_NESTING_CTR)0) { //如果该函数是在中断中被调用
109         *p_err = OS_ERR_MEM_CREATE_ISR;        //错误类型为“在中断中创建对象”
110         return;                                //返回，停止执行
111     }
112 #endif
113
```

```
114 #if OS_CFG_ARG_CHK_EN > 0u                          //如果使能了参数检测
115     if (p_addr == (void *)0) {                     //如果 p_addr 为空
116        *p_err    = OS_ERR_MEM_INVALID_P_ADDR;      //错误类型为“分区地址非法”
117         return;                                    //返回，停止执行
118     }
119     if (n_blks < (OS_MEM_QTY)2) {                  //如果分区的内存块数目少于2
120        *p_err = OS_ERR_MEM_INVALID_BLKS;           //错误类型为“内存块数目非法”
121         return;                                    //返回，停止执行
122     }
123     if (blk_size < sizeof(void *)) {               //如果内存块空间小于指针的
124        *p_err = OS_ERR_MEM_INVALID_SIZE;           //错误类型为“内存空间非法”
125         return;                                    //返回，停止执行
126     }
127     align_msk = sizeof(void *) - 1u;               //开始检查内存地址是否对齐
128     if (align_msk > 0u) {
129         if (((CPU_ADDR)p_addr & align_msk) != 0u){ //如果分区首地址没对齐
130            *p_err = OS_ERR_MEM_INVALID_P_ADDR;     //错误类型为“分区地址非法”
131             return;                                //返回，停止执行
132         }
133         if ((blk_size & align_msk) != 0u) {        //如果内存块地址没对齐
134            *p_err = OS_ERR_MEM_INVALID_SIZE;       //错误类型为“内存块大小非法”
135             return;                                //返回，停止执行
136         }
137     }
138 #endif
```

```
139     /* 将空闲内存块串联成一个单向链表 */
140     p_link = (void **)p_addr;                      //内存分区首地址转为二级指针
141     p_blk  = (CPU_INT08U *)p_addr;                 //首个内存块地址
142     loops  = n_blks - 1u;
143     for (i = 0u; i < loops; i++) {                 //将内存块逐个串成单向链表
144         p_blk +=  blk_size;                        //下一内存块地址
145        *p_link = (void  *)p_blk;                   //在当前内存块保存下一个内存块地址
146         p_link = (void **)(void *)p_blk;           //下一个内存块的地址转为二级指针
147     }
148    *p_link             = (void *)0;                //最后一个内存块指向空
149
150     OS_CRITICAL_ENTER();                           //进入临界段
151     p_mem->Type         = OS_OBJ_TYPE_MEM;         //设置对象的类型
152     p_mem->NamePtr      = p_name;                  //保存内存分区的命名
153     p_mem->AddrPtr      = p_addr;                  //存储内存分区的首地址
154     p_mem->FreeListPtr = p_addr;                   //初始化空闲内存块池的首地址
155     p_mem->NbrFree      = n_blks;                  //存储空闲内存块的数目
156     p_mem->NbrMax       = n_blks;                  //存储内存块的总数目
157     p_mem->BlkSize      = blk_size;                //存储内存块的空间大小
158
```

```
159 #if OS_CFG_DBG_EN > 0u                 //如果使能了调试代码和变量
160     OS_MemDbgListAdd(p_mem);           //将内存管理对象插入内存管理双向调试列表
161 #endif
162
163     OSMemQty++;                        //内存管理对象数目加1
164
165     OS_CRITICAL_EXIT_NO_SCHED();       //退出临界段（无调度）
166    *p_err = OS_ERR_NONE;               //错误类型为“无错误”
167 }
```

图 13-2 OSMemCreate () 函数

如果使能了 OS_CFG_DBG_EN（位于“os_cfg.h”），创建内存管理对象时还会调用 OS_MemDbgListAdd () 函数将该内存管理对象插入到一个内存管理调试列表，是为方便调试所设。OS_MemDbgListAdd () 函数的定义位于“os_mem.c”。

```
#if OS_CFG_DBG_EN > 0u                                          //如果使能了调试代码和变量
void  OS_MemDbgListAdd (OS_MEM  *p_mem)                         //将内存管理对象插入到内存管理调试列表的最前端
{
    p_mem->DbgPrevPtr               = (OS_MEM *)0;              //将该对象作为列表的最前端
    if (OSMemDbgListPtr == (OS_MEM *)0) {                       //如果列表为空
        p_mem->DbgNextPtr           = (OS_MEM *)0;              //该队列的下一个元素也为空
    } else {                                                    //如果列表非空
        p_mem->DbgNextPtr           =  OSMemDbgListPtr;         //列表原来的首元素作为该队列的下一个元素
        OSMemDbgListPtr->DbgPrevPtr =  p_mem;                   //原来的首元素的前面变为了该队列
    }
    OSMemDbgListPtr                 =  p_mem;                   //该对象成为列表的新首元素
}
#endif
```

图 13-3 OS_MemDbgListAdd () 函数

13.1.2 OSMemGet ()

OSMemGet () 函数用于向内存管理对象获取一个空闲内存块。OSMemGet () 函数的信息如下表所示。

表 43 OSMemGet ()

函数原型	void *OSMemGet (OS_MEM *p_mem, OS_ERR *p_err);			
功能	向内存管理对象获取一个空闲内存块。			
参数	p_mem	内存管理对象。		
	p_err	返回错误类型	OS_ERR_NONE	无错误，获取成功。
			OS_ERR_MEM_INVALID_P_MEM	p_mem 为空。
			OS_ERR_MEM_NO_FREE_BLKS	没有空闲的内存块。
返回值	获取到的内存块			

OSMemGet () 函数的定义也位于“os_mem.c

```
190 void  *OSMemGet (OS_MEM  *p_mem, //内存管理对象
191                  OS_ERR  *p_err) //返回错误类型
192 {
193     void    *p_blk;
194     CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必需该宏，该宏声明和
195                     //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
196                     // SR（临界段关中断只需保存SR），开中断时将该值还原。
197
198 #ifdef OS_SAFETY_CRITICAL                   //如果使能了安全检测
199     if (p_err == (OS_ERR *)0) {             //如果错误类型实参为空
200         OS_SAFETY_CRITICAL_EXCEPTION();     //执行安全检测异常函数
201         return ((void *)0);                 //返回0（有错误），停止执行
202     }
203 #endif
204
205 #if OS_CFG_ARG_CHK_EN > 0u                  //如果使能了参数检测
206     if (p_mem == (OS_MEM *)0) {             //如果 p_mem 为空
207        *p_err  = OS_ERR_MEM_INVALID_P_MEM;  //错误类型为“内存分区非法”
208         return ((void *)0);                 //返回0（有错误），停止执行
209     }
210 #endif
211
```

```
212     CPU_CRITICAL_ENTER();                   //关中断
213     if (p_mem->NbrFree == (OS_MEM_QTY)0) {  //如果没有空闲的内存块
214         CPU_CRITICAL_EXIT();                //开中断
215        *p_err = OS_ERR_MEM_NO_FREE_BLKS;    //错误类型为“没有空闲内存块”
216         return ((void *)0);                 //返回0（有错误），停止执行
217     }
218     p_blk              = p_mem->FreeListPtr; //如果还有空闲内存块，就获取它
219     p_mem->FreeListPtr = *(void **)p_blk;    //调整空闲内存块指针
220     p_mem->NbrFree--;                        //空闲内存块数目减1
221     CPU_CRITICAL_EXIT();                     //开中断
222    *p_err = OS_ERR_NONE;                     //错误类型为“无错误”
223     return (p_blk);                          //返回获取到的内存块
224 }
```

图 13-4 OSMemGet () 函数

13.1.3 OSMemPut ()

OSMemPut () 函数用于把内存块退还回内存管理对象（内存分区）。

表 44 OSFlagPend ()

函数原型	void OSMemPut (OS_MEM *p_mem, void *p_blk, OS_ERR *p_err);			
功能	内存块退还回内存管理对象。			
参数	p_mem	内存管理对象。		
	p_blk	要退还的内存块（的首地址）。		
	p_err	返回错	OS_ERR_NONE	无错误，退还成功。

		误类型	OS_ERR_MEM_FULL	内存分区的空闲内存块已满。
			OS_ERR_MEM_INVALID_P_BLK	p_blk 为空。
			OS_ERR_MEM_INVALID_P_MEM	p_mem 为空。
返回值	无。			

OSMemPut () 函数的定义也位于“os_mem.c”。

```
247 void  OSMemPut (OS_MEM  *p_mem,    //内存管理对象
248                 void    *p_blk,    //要退回的内存块
249                 OS_ERR  *p_err)    //返回错误类型
250 {
251     CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必需该宏，该宏声明和
252                     //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
253                     // SR（临界段关中断只需保存SR），开中断时将该值还原。
254
255 #ifdef OS_SAFETY_CRITICAL                   //如果使能了安全检测
256     if (p_err == (OS_ERR *)0) {             //如果错误类型实参为空
257         OS_SAFETY_CRITICAL_EXCEPTION();     //执行安全检测异常函数
258         return;                             //返回，停止执行
259     }
260 #endif
261
262 #if OS_CFG_ARG_CHK_EN > 0u                  //如果使能了参数检测
263     if (p_mem == (OS_MEM *)0) {             //如果 p_mem 为空
264        *p_err  = OS_ERR_MEM_INVALID_P_MEM;  //错误类型为“内存分区非法”
265         return;                             //返回，停止执行
266     }
267     if (p_blk == (void *)0) {               //如果内存块为空
268        *p_err  = OS_ERR_MEM_INVALID_P_BLK;  //错误类型为"内存块非法"
269         return;                             //返回，停止执行
270     }
271 #endif
272
273     CPU_CRITICAL_ENTER();                   //关中断
274     if (p_mem->NbrFree >= p_mem->NbrMax) {  //如果所有的内存块已经退出分区
275         CPU_CRITICAL_EXIT();                //开中断
276        *p_err = OS_ERR_MEM_FULL;            //错误类型为“内存分区已满”
277         return;                             //返回，停止执行
278     }
279     *(void **)p_blk     = p_mem->FreeListPtr; //把内存块插入空闲内存块链表
280     p_mem->FreeListPtr = p_blk;             //内存块退回到链表的最前端
281     p_mem->NbrFree++;                       //空闲内存块数目加1
282     CPU_CRITICAL_EXIT();                    //开中断
283    *p_err              = OS_ERR_NONE;       //错误类型为“无错误”
284 }
```

图 13-5 OSMemPut () 函数

13.2 实例演示

13.2.1 实例 1

本实例改用前一章“任务消息队列”的实例 1。功能一样，只是消息的存取位置采用了内存分区。本实例创建两个应用任务，AppTaskPost ()和 AppTaskPend ()，任务 AppTaskPost () 发布消息给任务 AppTaskPend ()。

该例程已经存放在配套资料的下图路径。

图 13-6 例程路径

本例程需用使用 LED1 和 USART1，所以工程中需要添加其驱动文件和初始化。用户可参照前面相关实例，这里不再赘述。

在起始任务 AppTaskStart() 中，在创建任务 AppTaskPend() 时，要切记创建内存管理对象 mem。

app_cfg.h app.c

```
167
168
169     /* 创建内存管理对象 mem */
170     OSMemCreate ((OS_MEM      *)&mem,                //指向内存管理对象
171                  (CPU_CHAR    *)"Mem For Test",      //命名内存管理对象
172                  (void        *)ucArray,             //内存分区的首地址
173                  (OS_MEM_QTY   )3,                   //内存分区中内存块数目
174                  (OS_MEM_SIZE  )20,                  //内存块的字节数目
175                  (OS_ERR      *)&err);               //返回错误类型
176
177
```

图 13-7 创建内存管理对象

任务函数 AppTaskPost () 的定义如下。任务每隔 1s 向任务 AppTaskPend()发送一个消息，消息内容存放在内存分区的一个内存块里。

```c
/*********************************************************************************************************
*                                              POST TASK
*********************************************************************************************************/
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,                         //指向内存管理对象
                          (OS_ERR      *)&err);                        //返回错误类型

    sprintf ( p_mem_blk, "%d", ulCount ++ );                           //向内存块存取计数值

    /* 发布消息到任务 AppTaskPend */
    OSTaskQPost ((OS_TCB      *)&AppTaskPendTCB,                       //目标任务的控制块
                 (void        *)p_mem_blk,                             //消息内容的首地址
                 (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 );              //每20ms发送一次

  }
}
```

图 13-8 AppTaskPost () 任务函数

任务函数 AppTaskPend () 的定义如下。任务接收到任务消息后，将切换 LED1 的亮灭状态，并将接收的消息内容和长度打印到串口调试助手，以及消息从被发布到被接收的时间差。任务使用完消息（存放在内存块）后，将把该消息所占用的内存块退还回内存分区 mem，供循环使用。

```c
/*********************************************************************************************************
*                                              PEND TASK
*********************************************************************************************************/
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
```

```
272   while (DEF_TRUE) {                                                   //任务体
273     /* 阻塞任务，等待任务消息 */
274     pMsg = OSTaskQPend ((OS_TICK        )0,                           //无期限等待
275                         (OS_OPT         )OS_OPT_PEND_BLOCKING,        //没有消息就阻塞任务
276                         (OS_MSG_SIZE   *)&msg_size,                   //返回消息长度
277                         (CPU_TS        *)&ts,                         //返回消息被发布的时间戳
278                         (OS_ERR        *)&err);                       //返回错误类型
279
280     ts = OS_TS_GET() - ts;                                            //计算消息从发布到被接收的时间差
281
282     macLED1_TOGGLE ();                                                //切换LED1的亮灭状态
283
284     OS_CRITICAL_ENTER();                                              //进入临界段，避免串口打印被打断
285
286     printf ( "\r\n接收到的消息的内容为：%s，长度是：%d字节。",
287              pMsg, msg_size );
288
289     printf ( "\r\n任务信号量从被发布到被接收的时间差是%dus\r\n",
290               ts / ( cpu_clk_freq / 1000000 ) );
291
292     OS_CRITICAL_EXIT();                                               //退出临界段
293
294     /* 退还内存块 */
295     OSMemPut ((OS_MEM  *)&mem,                                        //指向内存管理对象
296               (void    *)pMsg,                                        //内存块的首地址
297               (OS_ERR  *)&err);                                       //返回错误类型
298
299   }
300
301 }
```

图 13-9 AppTaskPend () 任务函数

把 STM32 的 USART1 连接至电脑的串口调试助手，编译和下载程序到秉火 STM32 开发板，运行程序。用户可以看到串口调试助手上打印任务 AppTaskPend () 接收到的消息长度和内容。用户可以观察到每隔 1 秒 LED1 切换一次亮灭状态，同时串口调试助手上也会打印出接收到的消息内容和长度，以及消息从被发布到被接收的时间差。

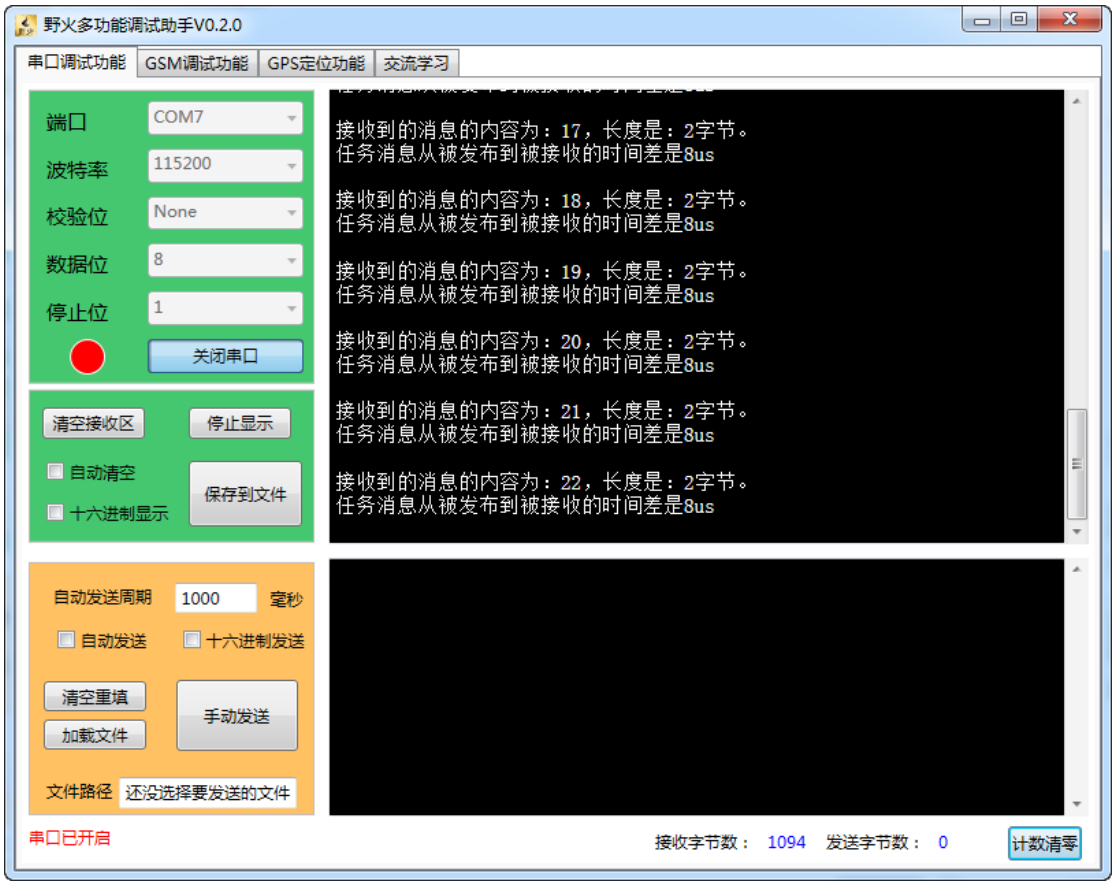

图 13-10 串口调试助手

13.3 章末总结

uC/OS 操作系统提供的内存管理，有利于减少处理器的内存碎片。内存分区是一个单向链表，当需要使用内存取数据时，就可以从链表中获取一个内存块，用完就退还回链表，这样就可以实现内存块的循环使用，避免了多次分配和释放内存而造成大量内存碎片。

使用内存管理之前必须先创建内存管理对象，创建内存管理对象使用 OSMemCreate () 函数。

OSMemGet () 函数用于向内存管理对象获取一个空闲内存块。OSMemPut () 函数用于把内存块退还回内存管理对象（内存分区）。

第14章 任务管理

在前面章节的诸多实例中，任务的使用屡见不鲜，但是本书还未曾对任务的机制进行过详细讲解，本章就在此着重讲解 uC/OS-III 的任务管理机制。

14.1 原理简述

如果想要使用内存管理机制，就必须事先使能内存管理。内存管理的使能位于“os_cfg.h”。

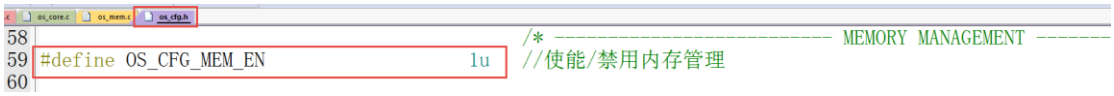

图 14-1

14.1.1 OSTaskCreate ()

要使用 uC/OS 的任务必须先声明任务控制块和创建任务，调用 OSTaskCreate () 函数可以创建一个任务。OSTaskCreate () 函数的信息如下表所示。

表 45 OSTaskCreate ()

函数原型	void OSTaskCreate (OS_TCB *p_tcb, CPU_CHAR *p_name, OS_TASK_PTR p_task, void *p_arg, OS_PRIO prio, CPU_STK *p_stk_base, CPU_STK_SIZE stk_limit, CPU_STK_SIZE stk_size, OS_MSG_QTY q_size, OS_TICK time_quanta, void *p_ext, OS_OPT opt, OS_ERR *p_err);	
功能	创建一个任务。	
参数	p_tcb	任务控制块指针。
	p_name	命名任务。
	p_task	任务函数。
	p_arg	传递给任务函数的参数。
	prio	任务优先级。uC/OS-III 允许任务拥有相同的优先级。
	p_stk_base	任务堆栈指针。

	stk_limit	任务堆栈的限制空间。		
	stk_size	任务堆栈总空间。		
	q_size	任务消息队列容量。只有使能了任务消息队列，该参数才有效。		
	time_quanta	时间片（单位：时钟节拍）。如果该参数设为 0，表示使用系统默认值（ OSCfg_TickRate_Hz / 10）。		
	p_ext	任务扩展内容的指针。		
	opt	选项	OS_OPT_TASK_NONE	没有选项要求。
			OS_OPT_TASK_STK_CHK	允许任务进行堆栈检测。
			OS_OPT_TASK_STK_CLR	堆栈全部进行清 0 初始化。
			OS_OPT_TASK_SAVE_FP	在上下文切换时保存浮点寄存器。STM32 芯片没有浮点寄存器，该项一般不用。
			OS_OPT_TASK_NO_TLS	屏蔽任务的 TLS 支持。
	p_err	返回错误类型	OS_ERR_NONE	无错误，创建成功。
			OS_ERR_ILLEGAL_CREATE_RUN_TIME	在调用 OSSafetyCriticalStart()函数后创建内核对象
			OS_ERR_NAME	p_name 为空。
			OS_ERR_PRIO_INVALID	prio>= S_CFG_PRIO_MAX-1，或者在 OS_CFG_ISR_POST_DEFERRED_EN 被置 1 时 prio=0。
			OS_ERR_STK_SIZE_INVALID	p_stk_base 为空。
			OS_ERR_STK_LIMIT_INVALID	stk_limit 大于 stk_size。
			OS_ERR_TASK_CREATE_ISR	在中断中创建任务。
			OS_ERR_TASK_INVALID	p_task 为空。
			OS_ERR_TCB_INVALID	p_tcb 为空。
返回值	无。			
注意事项	✧ 创建任务前必须声明任务的任务控制块 p_tcb、任务函数 p_task。 ✧ 不可以在中断中调用该函数。			

OSTaskCreate () 函数的定义位于“os_task.c”。

app.c | os_core.c | os_task.c | os_cfg_app.c | os_cfg.h | os_cpu_c.c | os.h | os_msg.c

```
249 void  OSTaskCreate (OS_TCB        *p_tcb,          //任务控制块指针
250                      CPU_CHAR      *p_name,         //命名任务
251                      OS_TASK_PTR    p_task,         //任务函数
252                      void          *p_arg,          //传递给任务函数的参数
253                      OS_PRIO        prio,           //任务优先级
254                      CPU_STK       *p_stk_base,     //任务堆栈基地址
255                      CPU_STK_SIZE   stk_limit,      //堆栈的剩余限制
256                      CPU_STK_SIZE   stk_size,       //堆栈大小
257                      OS_MSG_QTY     q_size,         //任务消息容量
258                      OS_TICK        time_quanta,    //时间片
259                      void          *p_ext,          //任务扩展
260                      OS_OPT         opt,            //选项
261                      OS_ERR        *p_err)          //返回错误类型
262 {
263     CPU_STK_SIZE   i;
264 #if OS_CFG_TASK_REG_TBL_SIZE > 0u
265     OS_REG_ID      reg_nbr;
266 #endif
267 #if defined(OS_CFG_TLS_TBL_SIZE) && (OS_CFG_TLS_TBL_SIZE > 0u)
268     OS_TLS_ID      id;
269 #endif
270
271     CPU_STK       *p_sp;
272     CPU_STK       *p_stk_limit;
273     CPU_SR_ALLOC();
274
275
276
277 #ifdef OS_SAFETY_CRITICAL                          //如果使能了安全检测
278     if (p_err == (OS_ERR *)0) {                    //如果错误类型实参为空
279         OS_SAFETY_CRITICAL_EXCEPTION();            //执行安全检测异常函数
280         return;                                    //返回，停止执行
281     }
282 #endif
```

```
283
284 #ifdef OS_SAFETY_CRITICAL_IEC61508                 //如果使能了安全关键
285     if (OSSafetyCriticalStartFlag == DEF_TRUE) {   //如果在调用OSSafetyCriticalStart()后创建
286        *p_err = OS_ERR_ILLEGAL_CREATE_RUN_TIME;    //错误类型为“非法创建内核对象”
287         return;                                    //返回，停止执行
288     }
289 #endif
290
291 #if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u             //如果使能了中断中非法调用检测
292     if (OSIntNestingCtr > (OS_NESTING_CTR)0) {     //如果该函数是在中断中被调用
293        *p_err = OS_ERR_TASK_CREATE_ISR;            //错误类型为“在中断中创建对象”
294         return;                                    //返回，停止执行
295     }
296 #endif
297
```

```
#if OS_CFG_ARG_CHK_EN > 0u                          //如果使能了参数检测
    if (p_tcb == (OS_TCB *)0) {                     //如果 p_tcb 为空
       *p_err = OS_ERR_TCB_INVALID;                 //错误类型为“任务控制块非法”
        return;                                     //返回，停止执行
    }
    if (p_task == (OS_TASK_PTR)0) {                 //如果 p_task 为空
       *p_err = OS_ERR_TASK_INVALID;                //错误类型为“任务函数非法”
        return;                                     //返回，停止执行
    }
    if (p_stk_base == (CPU_STK *)0) {               //如果 p_stk_base 为空
       *p_err = OS_ERR_STK_INVALID;                 //错误类型为“任务堆栈非法”
        return;                                     //返回，停止执行
    }
    if (stk_size < OSCfg_StkSizeMin) {                  //如果分配给任务的堆栈空间小于最小要求
       *p_err = OS_ERR_STK_SIZE_INVALID;                //错误类型为“任务堆栈空间非法”
        return;                                         //返回，停止执行
    }
    if (stk_limit >= stk_size) {                        //如果堆栈限制空间占尽了整个堆栈
       *p_err = OS_ERR_STK_LIMIT_INVALID;               //错误类型为“堆栈限制非法”
        return;                                         //返回，停止执行
    }
    if (prio >= OS_CFG_PRIO_MAX) {                      //如果任务优先级超出了限制范围
       *p_err = OS_ERR_PRIO_INVALID;                    //错误类型为i“优先级非法”
        return;                                         //返回，停止执行
    }
#endif

#if OS_CFG_ISR_POST_DEFERRED_EN > 0u                //如果使能了中断延迟发布
    if (prio == (OS_PRIO)0) {                       //如果优先级是0（中断延迟提交任务所需）
        if (p_tcb != &OSIntQTaskTCB) {              //如果 p_tcb 不是中断延迟提交任务
           *p_err = OS_ERR_PRIO_INVALID;            //错误类型为“优先级非法”
            return;                                 //返回，停止执行
        }
    }
#endif

    if (prio == (OS_CFG_PRIO_MAX - 1u)) {           //如果任务的优先级为最低（空闲任务所需）
        if (p_tcb != &OSIdleTaskTCB) {              //如果 p_tcb 不是空闲任务
           *p_err = OS_ERR_PRIO_INVALID;            //错误类型为“优先级非法”
            return;                                 //返回，停止执行
        }
    }


    OS_TaskInitTCB(p_tcb);                                  //初始化任务控制块 p_tcb

   *p_err = OS_ERR_NONE;                                    //错误类型为“无错误”

    if ((opt & OS_OPT_TASK_STK_CHK) != (OS_OPT)0) {         //如果选择了检测堆栈
        if ((opt & OS_OPT_TASK_STK_CLR) != (OS_OPT)0) {     //如果选择了清零堆栈
            p_sp = p_stk_base;                              //获取堆栈的基地址
            for (i = 0u; i < stk_size; i++) {               //将堆栈（数组）元素逐个清零
               *p_sp = (CPU_STK)0;
                p_sp++;
            }
        }
    }

    /* 初始化任务的堆栈结构 */
#if (CPU_CFG_STK_GROWTH == CPU_STK_GROWTH_HI_TO_LO)         //如果 CPU 的栈增长方向为从高到低
    p_stk_limit = p_stk_base + stk_limit;                   //堆栈限制空间靠堆栈的低地址端
#else                                                       //如果 CPU 的栈增长方向为从低到高
    p_stk_limit = p_stk_base + (stk_size - 1u) - stk_limit; //堆栈限制空间靠堆栈的高地址端
#endif
    /* 初始化任务堆栈 */
    p_sp = OSTaskStkInit(p_task,
                         p_arg,
                         p_stk_base,
                         p_stk_limit,
                         stk_size,
                         opt);
```

```
368     /* 初始化任务控制块 */
369     p_tcb->TaskEntryAddr = p_task;                          //保存任务地址
370     p_tcb->TaskEntryArg  = p_arg;                           //保存传递给任务函数的参数
371
372     p_tcb->NamePtr       = p_name;                          //保存任务名称
373
374     p_tcb->Prio          = prio;                            //保存任务遇险记
375
376     p_tcb->StkPtr        = p_sp;                            //保存堆栈的栈顶指针
377     p_tcb->StkLimitPtr   = p_stk_limit;                     //保存堆栈限制的临界地址
378
379     p_tcb->TimeQuanta    = time_quanta;                     //保存时间片
380 #if OS_CFG_SCHED_ROUND_ROBIN_EN > 0u                        //如果使能了时间片轮转调度
381     if (time_quanta == (OS_TICK)0) {                        //如果 time_quanta 为0
382         p_tcb->TimeQuantaCtr = OSSchedRoundRobinDfltTimeQuanta; //使用默认的时间片小小
383     } else {                                                //如果 time_quanta 非0
384         p_tcb->TimeQuantaCtr = time_quanta;                 //时间片为 time_quanta
385     }
386 #endif
387     p_tcb->ExtPtr        = p_ext;                           //保存任务扩展部分的指针
388     p_tcb->StkBasePtr    = p_stk_base;                      //保存堆栈的基地址
389     p_tcb->StkSize       = stk_size;                        //保存堆栈的大小
390     p_tcb->Opt           = opt;                             //保存选项
391
392 #if OS_CFG_TASK_REG_TBL_SIZE > 0u                           //如果允许使用任务寄存器
393     for (reg_nbr = 0u; reg_nbr < OS_CFG_TASK_REG_TBL_SIZE; reg_nbr++) { //将任务寄存器初始化为0
394         p_tcb->RegTbl[reg_nbr] = (OS_REG)0;
395     }
396 #endif
397
398 #if OS_CFG_TASK_Q_EN > 0u                                   //如果使能了任务消息队列
399     OS_MsgQInit(&p_tcb->MsgQ,                               //初始化任务消息队列
400                 q_size);
401 #else                                                       //如果禁用了任务消息队列
402     (void)&q_size;                                          //参数 q_size 无效
403 #endif
404
405     OSTaskCreateHook(p_tcb);                                //调用用户定义的钩子函数
406
407 #if defined(OS_CFG_TLS_TBL_SIZE) && (OS_CFG_TLS_TBL_SIZE > 0u)
408     for (id = 0u; id < OS_CFG_TLS_TBL_SIZE; id++) {
409         p_tcb->TLS_Tbl[id] = (OS_TLS)0;
410     }
411     OS_TLS_TaskCreate(p_tcb);                               /* Call TLS hook
412 #endif
413     /* 把任务插入就绪列表 */
414     OS_CRITICAL_ENTER();                                    //进入临界段
415     OS_PrioInsert(p_tcb->Prio);                             //置就绪优先级位映像表中相应优先级处于就绪状态
416     OS_RdyListInsertTail(p_tcb);                            //将新创建的任务插入就绪列表尾端
417
418 #if OS_CFG_DBG_EN > 0u                                      //如果使能了调试代码和变量
419     OS_TaskDbgListAdd(p_tcb);                               //把任务插入到任务调试双向列表
420 #endif
421
422     OSTaskQty++;                                                    //任务数目加1
423
424     if (OSRunning != OS_STATE_OS_RUNNING) {                         //如果系统尚未启动
425         OS_CRITICAL_EXIT();                                         //退出临界段
426         return;                                                     //返回，停止执行
427     }
428     /* 如果系统已经启动 */
429     OS_CRITICAL_EXIT_NO_SCHED();                                    //退出临界段（无调度）
430
431     OSSched();                                                      //进行任务调度
432 }
```

图 14-2 OSTaskCreate () 函数

如果使能了 OS_CFG_DBG_EN（位于“os_cfg.h”），创建任务时还会调用 OS_TaskDbgListAdd () 函数将该任务插入到一个任务调试双向列表，是为方便调试所设。OS_TaskDbgListAdd () 函数的定义位于“os_task.c”。

```
1861 #if OS_CFG_DBG_EN > 0u                                      //如果使能了调试代码和变量
1862 void  OS_TaskDbgListAdd (OS_TCB  *p_tcb)                    //将任务插入到任务调试列表的最前端
1863 {
1864     p_tcb->DbgPrevPtr               = (OS_TCB *)0;          //将该任务作为列表的最前端
1865     if (OSTaskDbgListPtr == (OS_TCB *)0) {                  //如果列表为空
1866         p_tcb->DbgNextPtr           = (OS_TCB *)0;          //该队列的下一个元素也为空
1867     } else {                                                //如果列表非空
1868         p_tcb->DbgNextPtr           =  OSTaskDbgListPtr;    //列表原来的首元素作为该任务的下一个元素
1869         OSTaskDbgListPtr->DbgPrevPtr =  p_tcb;              //原来的首元素的前面变为了该任务
1870     }
1871     OSTaskDbgListPtr                =  p_tcb;               //该任务成为列表的新首元素
1872 }
```

图 14-3 OS_TaskDbgListAdd () 函数

14.1.2 OSTaskSuspend ()

OSTaskSuspend () 函数用于挂起一个任务，令任务暂停运行。任务可以多次调用 OSTaskSuspend() 对任务进行挂起操作，即一个任务被挂起是可以嵌套的，但是想要将任务脱离挂起状态需要调用相应次数的 OSTaskResume() 函数。除空闲任务和延迟提交任务之外，任务可以挂起任何任务。OSTaskSuspend () 函数的信息如下表所示。

表 46 OSTaskSuspend ()

函数原型	void OSTaskSuspend (OS_TCB *p_tcb, OS_ERR *p_err);			
功能	挂起一个任务。			
参数	p_tcb	任务控制块指针，0 表自身。		
	p_err	返回错误类型	OS_ERR_NONE	无错误，挂起任务成功。
			OS_ERR_SCHED_LOCKED	挂起当前任务时调度器被锁。
			OS_ERR_TASK_SUSPEND_ISR	在禁用中断延迟发布的情况下，在中断中调用该函数。
			OS_ERR_TASK_SUSPEND_IDLE	挂起空闲任务。
			OS_ERR_TASK_SUSPEND_INT_HANDLER	在使能中断延迟发布的情况下，挂起中断延迟提交任务。
返回值	无。			
注意事项	✧ 不能挂起空闲任务。 ✧ 当 OS_CFG_ISR_POST_DEFERRED_EN = 0u 时，不能在中断中调用该函数。 ✧ 当 OS_CFG_ISR_POST_DEFERRED_EN > 0u 时，不能挂起中断延迟提交任务。			

OSTaskSuspend () 函数的定义也位于“os_task.c

```
1737 #if OS_CFG_TASK_SUSPEND_EN > 0u                          //如果使能了函数 OSTaskSuspend()
1738 void   OSTaskSuspend (OS_TCB  *p_tcb,                    //任务控制块指针
1739                       OS_ERR  *p_err)                    //返回错误类型
1740 {
1741 #ifdef OS_SAFETY_CRITICAL                                //如果使能了安全检测
1742     if (p_err == (OS_ERR *)0) {                           //如果 p_err 为空
1743         OS_SAFETY_CRITICAL_EXCEPTION();                   //执行安全检测异常函数
1744         return;                                           //返回，停止执行
1745     }
1746 #endif
1747
1748 #if (OS_CFG_ISR_POST_DEFERRED_EN   == 0u) && \
1749     (OS_CFG_CALLED_FROM_ISR_CHK_EN >  0u)                 //如果禁用了中断延迟发布和中断中非法调用检测
1750     if (OSIntNestingCtr > (OS_NESTING_CTR)0) {            //如果在中断中调用该函数
1751        *p_err = OS_ERR_TASK_SUSPEND_ISR;                  //错误类型为“在中断中挂起任务”
1752         return;                                           //返回，停止执行
1753     }
1754 #endif
1755
1756     if (p_tcb == &OSIdleTaskTCB) {                        //如果 p_tcb 是空闲任务
1757        *p_err = OS_ERR_TASK_SUSPEND_IDLE;                 //错误类型为“挂起空闲任务”
1758         return;                                           //返回，停止执行
1759     }


1760
1761 #if OS_CFG_ISR_POST_DEFERRED_EN > 0u                      //如果使能了中断延迟发布
1762     if (p_tcb == &OSIntQTaskTCB) {                        //如果 p_tcb 为中断延迟提交任务
1763        *p_err = OS_ERR_TASK_SUSPEND_INT_HANDLER;          //错误类型为“挂起中断延迟提交任务”
1764         return;                                           //返回，停止执行
1765     }
1766
1767     if (OSIntNestingCtr > (OS_NESTING_CTR)0) {            //如果在中断中调用该函数
1768         OS_IntQPost((OS_OBJ_TYPE)OS_OBJ_TYPE_TASK_SUSPEND,//把挂起任务命令发布到中断消息队列
1769                     (void      *)p_tcb,
1770                     (void      *)0,
1771                     (OS_MSG_SIZE)0,
1772                     (OS_FLAGS   )0,
1773                     (OS_OPT     )0,
1774                     (CPU_TS     )0,
1775                     (OS_ERR    *)p_err);
1776         return;                                           //返回，停止执行
1777     }
1778 #endif
1779     /* 如果禁用了中断延迟发布或不是在中断中调用该函数 */
1780     OS_TaskSuspend(p_tcb, p_err);                         //直接将任务 p_tcb 挂起
1781 }
1782 #endif
```

图 14-4 OSTaskSuspend () 函数

其实，不管是否使能了中断延迟发布，最终都是调用 OS_TaskSuspend () 函数挂起任务。只是使能了中断延迟发布的挂起过程会比较曲折，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。OS_TaskSuspend () 函数的定义位于“os_task.c”。

```
2485 #if OS_CFG_TASK_SUSPEND_EN > 0u          //如果使能了函数 OSTaskSuspend()
2486 void   OS_TaskSuspend (OS_TCB  *p_tcb,    //任务控制块指针
2487                        OS_ERR  *p_err)    //返回错误类型
2488 {
2489     CPU_SR_ALLOC();  //使用到临界段（在关/开中断时）时必需该宏，该宏声明和
2490                      //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
2491                      // SR（临界段关中断只需保存SR），开中断时将该值还原。
2492
2493     CPU_CRITICAL_ENTER();                                //关中断
2494     if (p_tcb == (OS_TCB *)0) {                          //如果 p_tcb 为空
2495         p_tcb = OSTCBCurPtr;                             //挂起自身
2496     }
2497
2498     if (p_tcb == OSTCBCurPtr) {                          //如果是挂起自身
2499         if (OSSchedLockNestingCtr > (OS_NESTING_CTR)0) { //如果调度器被锁
2500             CPU_CRITICAL_EXIT();                         //开中断
2501            *p_err = OS_ERR_SCHED_LOCKED;                 //错误类型为“调度器被锁”
2502             return;                                      //返回，停止执行
2503         }
2504     }
2505
```

```
2506    *p_err = OS_ERR_NONE;                                    //错误类型为“无错误”
2507     switch (p_tcb->TaskState) {                             //根据 p_tcb 的任务状态分类处理
2508         case OS_TASK_STATE_RDY:                             //如果是就绪状态
2509              OS_CRITICAL_ENTER_CPU_EXIT();                  //锁调度器，重开中断
2510              p_tcb->TaskState  =  OS_TASK_STATE_SUSPENDED;   //任务状态改为“挂起状态”
2511              p_tcb->SuspendCtr = (OS_NESTING_CTR)1;         //挂起前套数为1
2512              OS_RdyListRemove(p_tcb);                       //将任务从就绪列表移除
2513              OS_CRITICAL_EXIT_NO_SCHED();                   //开调度器，不进行调度
2514              break;                                         //跳出
2515
2516         case OS_TASK_STATE_DLY:                             //如果是延时状态
2517              p_tcb->TaskState  = OS_TASK_STATE_DLY_SUSPENDED; //任务状态改为“延时中被挂起”
2518              p_tcb->SuspendCtr = (OS_NESTING_CTR)1;         //挂起前套数为1
2519              CPU_CRITICAL_EXIT();                           //开中断
2520              break;                                         //跳出
2521
2522         case OS_TASK_STATE_PEND:                            //如果是无期限等待状态
2523              p_tcb->TaskState  = OS_TASK_STATE_PEND_SUSPENDED;//任务状态改为“无期限等待中被挂起”
2524              p_tcb->SuspendCtr = (OS_NESTING_CTR)1;         //挂起前套数为1
2525              CPU_CRITICAL_EXIT();                           //开中断
2526              break;                                         //跳出
2527
2528         case OS_TASK_STATE_PEND_TIMEOUT:                    //如果是有期限等待状态
2529              p_tcb->TaskState  = OS_TASK_STATE_PEND_TIMEOUT_SUSPENDED; //改为“有期限等待中被挂起”
2530              p_tcb->SuspendCtr = (OS_NESTING_CTR)1;         //挂起前套数为1
2531              CPU_CRITICAL_EXIT();                           //开中断
2532              break;                                         //跳出
2533

2534         case OS_TASK_STATE_SUSPENDED:                       //如果状态中有挂起状态
2535         case OS_TASK_STATE_DLY_SUSPENDED:
2536         case OS_TASK_STATE_PEND_SUSPENDED:
2537         case OS_TASK_STATE_PEND_TIMEOUT_SUSPENDED:
2538              p_tcb->SuspendCtr++;                           //挂起嵌套数加1
2539              CPU_CRITICAL_EXIT();                           //开中断
2540              break;                                         //跳出
2541
2542         default:                                            //如果任务状态超出预期
2543              CPU_CRITICAL_EXIT();                           //开中断
2544             *p_err = OS_ERR_STATE_INVALID;                  //错误类型为“状态非法”
2545              return;                                        //返回，停止执行
2546     }
2547
2548     OSSched();                                             //调度任务
2549 }
2550 #endif
```

图 14-5 OS_TaskSuspend () 函数

14.1.3 OSTaskResume ()

与 OSTaskSuspend () 函数相对应，被挂起的任务如果要恢复被挂起前的任务状态，就必须调用 OSTaskResume () 函数解嵌该任务。如果解嵌后任务挂起前套数为 0，就可以恢复被挂起前的任务状态。

表 47 OSTaskResume ()

函数原型	void OSTaskResume (OS_TCB *p_tcb, OS_ERR *p_err);			
功能	解嵌一个被挂起的任务。			
参数	p_tcb	任务控制块指针。		
	p_err	返回错误类型	OS_ERR_NONE	无错误，解嵌成功。
			OS_ERR_STATE_INVALID	任务状态非法。

			OS_ERR_TASK_RESUME_ISR	在禁用中断延迟发布的情况下，在中断中调用该函数。
			OS_ERR_TASK_RESUME_SELF	解嵌自身。
			OS_ERR_TASK_NOT_SUSPENDED	任务 p_tcb 未被挂起。
返回值	无。			
注意事项	✧ 不能解嵌自身，即 p_tcb 不能为 0 或当前运行任务。 ✧ 当 OS_CFG_ISR_POST_DEFERRED_EN = 0u 时，不能在中断中调用该函数。			

OSTaskResume () 函数的定义也位于“os_task.c

```
1175 #if OS_CFG_TASK_SUSPEND_EN > 0u                 //如果使能了函数 OSTaskResume()
1176 void  OSTaskResume (OS_TCB  *p_tcb,             //任务控制块指针
1177                     OS_ERR  *p_err)             //返回错误类型
1178 {
1179     CPU_SR_ALLOC();  //使用到临界段（在关/开中断时）时必需该宏，该宏声明和
1180                      //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
1181                      // SR（临界段关中断只需保存SR），开中断时将该值还原。
1182
1183 #ifdef OS_SAFETY_CRITICAL                       //如果使能了安全检测
1184     if (p_err == (OS_ERR *)0) {                 //如果 p_err 为空
1185         OS_SAFETY_CRITICAL_EXCEPTION();         //执行安全检测异常函数
1186         return;                                 //返回，停止执行
1187     }
1188 #endif
1189
1190 #if (OS_CFG_ISR_POST_DEFERRED_EN    == 0u) && \
1191     (OS_CFG_CALLED_FROM_ISR_CHK_EN >  0u)       //如果禁用了中断延迟发布和中断中非法调用检测
1192     if (OSIntNestingCtr > (OS_NESTING_CTR)0) { //如果在中断中调用该函数
1193        *p_err = OS_ERR_TASK_RESUME_ISR;         //错误类型为“在中断中恢复任务”
1194         return;                                 //返回，停止执行
1195     }
1196 #endif
1197
1198

1199     CPU_CRITICAL_ENTER();                       //关中断
1200 #if OS_CFG_ARG_CHK_EN > 0u                      //如果使能了参数检测
1201     if ((p_tcb == (OS_TCB *)0) ||               //如果被恢复任务为空或是自身
1202         (p_tcb == OSTCBCurPtr)) {
1203         CPU_CRITICAL_EXIT();                    //开中断
1204        *p_err  = OS_ERR_TASK_RESUME_SELF;       //错误类型为“恢复自身”
1205         return;                                 //返回，停止执行
1206     }
1207 #endif
1208     CPU_CRITICAL_EXIT();                        //关中断
1209
1210 #if OS_CFG_ISR_POST_DEFERRED_EN > 0u                          //如果使能了中断延迟发布
1211     if (OSIntNestingCtr > (OS_NESTING_CTR)0) {                //如果该函数在中断中被调用
1212         OS_IntQPost((OS_OBJ_TYPE)OS_OBJ_TYPE_TASK_RESUME,     //把恢复任务命令发布到中断消息队列
1213                     (void      *)p_tcb,
1214                     (void      *)0,
1215                     (OS_MSG_SIZE)0,
1216                     (OS_FLAGS   )0,
1217                     (OS_OPT     )0,
1218                     (CPU_TS     )0,
1219                     (OS_ERR    *)p_err);
1220         return;                                               //返回，停止执行
1221     }
1222 #endif
1223     /* 如果禁用了中断延迟发布或不是在中断中调用该函数 */
1224     OS_TaskResume(p_tcb, p_err);                              //直接将任务 p_tcb 挂起
1225 }
1226 #endif
```

图 14-6 OSTaskResume () 函数

其实，不管是否使能了中断延迟发布，最终都是调用 OS_TaskResume () 函数恢复任务。只是使能了中断延迟发布的挂起过程会比较曲折，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。OS_TaskResume () 函数的定义位于“os_task.c”。

```
2214 #if OS_CFG_TASK_SUSPEND_EN > 0u            //如果使能了函数 OSTaskResume()
2215 void  OS_TaskResume (OS_TCB  *p_tcb,       //任务控制块指针
2216                      OS_ERR  *p_err)       //返回错误类型
2217 {
2218     CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必需该宏，该宏声明和
2219                     //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
2220                     // SR（临界段关中断只需保存SR），开中断时将该值还原。
2221     CPU_CRITICAL_ENTER();                                       //关中断
2222    *p_err  = OS_ERR_NONE;                                       //错误类型为“无错误”
2223     switch (p_tcb->TaskState) {                                 //根据 p_tcb 的任务状态分类处理
2224         case OS_TASK_STATE_RDY:                                 //如果状态中没有挂起状态
2225         case OS_TASK_STATE_DLY:
2226         case OS_TASK_STATE_PEND:
2227         case OS_TASK_STATE_PEND_TIMEOUT:
2228              CPU_CRITICAL_EXIT();                               //开中断
2229             *p_err = OS_ERR_TASK_NOT_SUSPENDED;                 //错误类型为“任务未被挂起”
2230              break;                                             //跳出
2231
2232         case OS_TASK_STATE_SUSPENDED:                           //如果是“挂起状态”
2233              OS_CRITICAL_ENTER_CPU_EXIT();                      //锁调度器，重开中断
2234              p_tcb->SuspendCtr--;                               //任务的挂起嵌套数减1
2235              if (p_tcb->SuspendCtr == (OS_NESTING_CTR)0) {      //如果挂起前套数为0
2236                  p_tcb->TaskState = OS_TASK_STATE_RDY;          //修改状态为“就绪状态”
2237                  OS_TaskRdy(p_tcb);                             //把 p_tcb 插入就绪列表
2238              }
2239              OS_CRITICAL_EXIT_NO_SCHED();                       //开调度器，不调度任务
2240              break;                                             //跳出
2241
2242         case OS_TASK_STATE_DLY_SUSPENDED:                       //如果是“延时中被挂起”
2243              p_tcb->SuspendCtr--;                               //任务的挂起嵌套数减1
2244              if (p_tcb->SuspendCtr == (OS_NESTING_CTR)0) {      //如果挂起前套数为0
2245                  p_tcb->TaskState = OS_TASK_STATE_DLY;          //修改状态为“延时状态”
2246              }
2247              CPU_CRITICAL_EXIT();                               //开中断
2248              break;                                             //跳出
2249
2250         case OS_TASK_STATE_PEND_SUSPENDED:                      //如果是“无期限等待中被挂起”
2251              p_tcb->SuspendCtr--;                               //任务的挂起嵌套数减1
2252              if (p_tcb->SuspendCtr == (OS_NESTING_CTR)0) {      //如果挂起前套数为0
2253                  p_tcb->TaskState = OS_TASK_STATE_PEND;         //修改状态为“无期限等待状态”
2254              }
2255              CPU_CRITICAL_EXIT();                               //开中断
2256              break;                                             //跳出
2257
2258         case OS_TASK_STATE_PEND_TIMEOUT_SUSPENDED:              //如果是“有期限等待中被挂起”
2259              p_tcb->SuspendCtr--;                               //任务的挂起嵌套数减1
2260              if (p_tcb->SuspendCtr == (OS_NESTING_CTR)0) {      //如果挂起前套数为0
2261                  p_tcb->TaskState = OS_TASK_STATE_PEND_TIMEOUT;//修改状态为“有期限等待状态”
2262              }
2263              CPU_CRITICAL_EXIT();                               //开中断
2264              break;                                             //跳出
2265
2266         default:                                                //如果 p_tcb 任务状态超出预期
2267              CPU_CRITICAL_EXIT();                               //开中断
2268             *p_err = OS_ERR_STATE_INVALID;                      //错误类型为“状态非法”
2269              return;                                            //跳出
2270     }
2271
2272     OSSched();                                                  //调度任务
2273 }
2274 #endif
```

图 14-7 OS_TaskResume () 函数

14.1.4 OSTaskChangePrio ()

在创建任务的时候，可以通过设置参数 prio 来设置任务的优先级。在创建完任务后，还可以通过 OSTaskChangePrio () 函数调整任务的优先级。要使用 OSTaskChangePrio () 函数，还得事先使能 OS_CFG_TASK_CHANGE_PRIO_EN（位于“os_cfg.h”），如下所示。

```
81
82                                                  /* ---------------------------- TASK MANAGEMENT ---
83  #define OS_CFG_STAT_TASK_EN                1u   //使能/禁用统计任务
84  #define OS_CFG_STAT_TASK_STK_CHK_EN        1u   //使能/禁用从统计任务检查堆栈
85
86  #define OS_CFG_TASK_CHANGE_PRIO_EN         1u   //使能/禁用函数 OSTaskChangePrio()
87  #define OS_CFG_TASK_DEL_EN                 1u   //使能/禁用函数 OSTaskDel()
88  #define OS_CFG_TASK_Q_EN                   1u   //使能/禁用函数 OSTaskQXXXX()
89  #define OS_CFG_TASK_Q_PEND_ABORT_EN        1u   //使能/禁用函数 OSTaskQPendAbort()
90  #define OS_CFG_TASK_PROFILE_EN             1u   //使能/禁用任务控制块的简况变量
91  #define OS_CFG_TASK_REG_TBL_SIZE           1u   //定义任务寄存器的数目
92  #define OS_CFG_TASK_SEM_PEND_ABORT_EN      1u   //使能/禁用函数 OSTaskSemPendAbort()
93  #define OS_CFG_TASK_SUSPEND_EN             1u   //使能/禁用函数 OSTaskSuspend() 和 OSTaskResume()
94
```

图 14-8

OSTaskChangePrio () 函数的信息如下所示。

表 48 OSTaskChangePrio ()

<table>
<tr><td>函数原型</td><td colspan="3">void OSTaskChangePrio (OS_TCB *p_tcb,
OS_PRIO prio_new,
OS_ERR *p_err);</td></tr>
<tr><td>功能</td><td colspan="3">改变一个任务的优先级。</td></tr>
<tr><td rowspan="6">参数</td><td>p_tcb</td><td colspan="2">目标任务的任务控制块指针，0 表自身。</td></tr>
<tr><td>prio_new</td><td colspan="2">新优先级。</td></tr>
<tr><td rowspan="4">p_err</td><td rowspan="4">返回错误类型</td><td>OS_ERR_NONE</td><td>无错误，调用成功。</td></tr>
<tr><td>OS_ERR_PRIO_INVALID</td><td>当使能了中断延迟发布时，prio_new = 0；或者，prio_new >= (OS_CFG_PRIO_MAX-1)。</td></tr>
<tr><td>OS_ERR_STATE_INVALID</td><td>目标任务的任务状态非法。</td></tr>
<tr><td>OS_ERR_TASK_CHANGE_PRIO_ISR</td><td>在中断中调用该函数。</td></tr>
<tr><td>返回值</td><td colspan="3">无。</td></tr>
<tr><td>注意事项</td><td colspan="3">✧ 当 OS_CFG_ISR_POST_DEFERRED_EN > 0u 时，prio_new 不能为 0；prio_new 不能 >= (OS_CFG_PRIO_MAX-1)。
✧ 不能在中断中调用该函数。</td></tr>
</table>

OSTaskChangePrio () 函数的定义也位于“os_task.c

```
#if OS_CFG_TASK_CHANGE_PRIO_EN > 0u                 //如果使能了函数 OSTaskChangePrio()
void  OSTaskChangePrio (OS_TCB   *p_tcb,            //目标任务的任务控制块指针
                        OS_PRIO   prio_new,         //新优先级
                        OS_ERR   *p_err)            //返回错误类型
{
    CPU_BOOLEAN   self;
    CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必需该宏，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                           //如果使能（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;                                     //返回，停止执行
    }
#endif

#if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u              //如果使能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_CHANGE_PRIO_ISR;        //错误类型为“在中断中改变优先级”
        return;                                     //返回，停止执行
    }
#endif


#if OS_CFG_ISR_POST_DEFERRED_EN > 0u                //如果使能了中断延迟发布
    if (prio_new == 0) {                            //如果 prio_new 为0
       *p_err = OS_ERR_PRIO_INVALID;                //错误类型为“优先级非法”
        return;                                     //返回，停止执行
    }
#endif

    if (prio_new >= (OS_CFG_PRIO_MAX - 1u)) {       //如果 prio_new 超出限制范围
       *p_err = OS_ERR_PRIO_INVALID;                //错误类型为“优先级非法”
        return;                                     //返回，停止执行
    }

    if (p_tcb == (OS_TCB *)0) {                     //如果 p_tcb 为空
        CPU_CRITICAL_ENTER();                       //关中断
        p_tcb = OSTCBCurPtr;                        //目标任务为当前任务（自身）
        CPU_CRITICAL_EXIT();                        //开中断
        self  = DEF_TRUE;                           //目标任务是自身
    } else {                                        //如果 p_tcb 非空
        self  = DEF_FALSE;                          //目标任务不是自身
    }


    OS_CRITICAL_ENTER();                            //进入临界段
    switch (p_tcb->TaskState) {                     //根据目标任务状态分类处理
        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
             p_tcb->Prio = prio_new;                //为任务设置新优先级
             OS_PrioInsert(p_tcb->Prio);            //在优先级表格中将该优先级位置1
             if (self == DEF_TRUE) {                //如果目标任务是自身
                 OS_RdyListInsertHead(p_tcb);       //将目标任务插至就绪列表头端
             } else {                               //如果目标任务不是自身
                 OS_RdyListInsertTail(p_tcb);       //将目标任务插至就绪列表尾端
             }
             break;                                 //跳出

        case OS_TASK_STATE_DLY:                     //如果是延时状态
        case OS_TASK_STATE_SUSPENDED:               //如果是挂起状态
        case OS_TASK_STATE_DLY_SUSPENDED:           //如果是延时中被挂起状态
             p_tcb->Prio = prio_new;                //直接修改任务的优先级
             break;                                 //跳出
```

```
125         case OS_TASK_STATE_PEND:                    //如果包含等待状态
126         case OS_TASK_STATE_PEND_TIMEOUT:
127         case OS_TASK_STATE_PEND_SUSPENDED:
128         case OS_TASK_STATE_PEND_TIMEOUT_SUSPENDED:
129              switch (p_tcb->PendOn) {               //根据任务等待的对象分类处理
130                  case OS_TASK_PEND_ON_TASK_Q:       //如果等待的是任务消息
131                  case OS_TASK_PEND_ON_TASK_SEM:     //如果等待的是任务信号量
132                  case OS_TASK_PEND_ON_FLAG:         //如果等待的是事件标志组
133                       p_tcb->Prio = prio_new;       //直接修改任务的优先级
134                       break;                        //跳出
135
136                  case OS_TASK_PEND_ON_MUTEX:        //如果等待的是互斥信号量
137                  case OS_TASK_PEND_ON_MULTI:        //如果等待的是多个内核对象
138                  case OS_TASK_PEND_ON_Q:            //如果等待的是消息队列
139                  case OS_TASK_PEND_ON_SEM:          //如果等待的是多值信号量
140                       OS_PendListChangePrio(p_tcb,     //改变任务的优先级和
141                                             prio_new);//在等待列表中的位置。
142                       break;                        //跳出
143
144                  default:                           //如果任务等待的对象超出预期
145                       break;                        //直接跳出
146              }
147              break;                                 //跳出
```

```
148
149         default:                                    //如果目标任务状态超出预期
150              OS_CRITICAL_EXIT();                    //退出临界段
151             *p_err = OS_ERR_STATE_INVALID;          //错误类型为“状态非法”
152              return;                                //返回，停止执行
153     }
154
155     OS_CRITICAL_EXIT_NO_SCHED();                    //退出临界段（无调度）
156
157     OSSched();                                      //调度任务
158
159    *p_err = OS_ERR_NONE;                            //错误类型为“无错误”
160 }
161 #endif
```

图 14-9 OSTaskChangePrio () 函数

14.1.5 OSTaskDel ()

当任务不再使用时，可以调用 OSTaskDel() 函数删除任务。要使用 OSTaskDel() 函数，还得事先使能 OS_CFG_TASK_SEM_PEND_ABORT_EN（位于“os_cfg.h”），如下所示。

```
81
82                                          /* ------------------------ TASK MANAGEMENT ---
83  #define OS_CFG_STAT_TASK_EN             1u   //使能/禁用统计任务
84  #define OS_CFG_STAT_TASK_STK_CHK_EN     1u   //使能/禁用从统计任务检查堆栈
85
86  #define OS_CFG_TASK_CHANGE_PRIO_EN      1u   //使能/禁用函数 OSTaskChangePrio()
87  #define OS_CFG_TASK_DEL_EN              1u   //使能/禁用函数 OSTaskDel()
88  #define OS_CFG_TASK_Q_EN                1u   //使能/禁用函数 OSTaskQXXXX()
89  #define OS_CFG_TASK_Q_PEND_ABORT_EN     1u   //使能/禁用函数 OSTaskQPendAbort()
90  #define OS_CFG_TASK_PROFILE_EN          1u   //使能/禁用任务控制块的简况变量
91  #define OS_CFG_TASK_REG_TBL_SIZE        1u   //定义任务寄存器的数目
92  #define OS_CFG_TASK_SEM_PEND_ABORT_EN   1u   //使能/禁用函数 OSTaskSemPendAbort()
93  #define OS_CFG_TASK_SUSPEND_EN          1u   //使能/禁用函数 OSTaskSuspend() 和 OSTaskResume()
94
```

图 14-10

OSTaskDel() 函数的信息如下所示。

表 49 OSTaskDel ()

函数原型	void OSTaskDel (OS_TCB *p_tcb, OS_ERR *p_err);			
功能	删除一个任务。			
参数	p_tcb	目标任务的任务控制块指针，0 表自身。		
	p_err	返回错误类型	OS_ERR_NONE	无错误，调用成功。
			OS_ERR_STATE_INVALID	任务状态非法。
			OS_ERR_TASK_DEL_IDLE	p_tcb 为空闲任务。
			OS_ERR_TASK_DEL_INVALID	当使能了中断延迟发布时，p_tcb 为中断延迟提交任务。
			OS_ERR_TASK_DEL_ISR	在中断中调用该函数。
返回值	无。			
注意事项	✧ 当 OS_CFG_ISR_POST_DEFERRED_EN > 0u 时，不能删除中断延迟提交任务。 ✧ 不能在中断中调用该函数。			

OSTaskDel () 函数的定义也位于“os_task.c

```
459 #if OS_CFG_TASK_DEL_EN > 0u                          //如果使能了函数 OSTaskDel()
460 void  OSTaskDel (OS_TCB  *p_tcb,                    //目标任务控制块指针
461                  OS_ERR  *p_err)                    //返回错误类型
462 {
463     CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必需该宏，该宏声明和
464                     //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
465                     // SR（临界段关中断只需保存SR），开中断时将该值还原。
466
467 #ifdef OS_SAFETY_CRITICAL                           //如果使能（默认禁用）了安全检测
468     if (p_err == (OS_ERR *)0) {                     //如果 p_err 为空
469         OS_SAFETY_CRITICAL_EXCEPTION();             //执行安全检测异常函数
470         return;                                     //返回，停止执行
471     }
472 #endif
473
474 #if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u              //如果使能了中断中非法调用检测
475     if (OSIntNestingCtr > (OS_NESTING_CTR)0) {      //如果该函数在中断中被调用
476        *p_err = OS_ERR_TASK_DEL_ISR;                //错误类型为“在中断中删除任务”
477         return;                                     //返回，停止执行
478     }
479 #endif
480
```

```
481     if (p_tcb == &OSIdleTaskTCB) {                  //如果目标任务是空闲任务
482         *p_err = OS_ERR_TASK_DEL_IDLE;              //错误类型为“删除空闲任务”
483          return;                                    //返回，停止执行
484     }
485
486 #if OS_CFG_ISR_POST_DEFERRED_EN > 0u                //如果使能了中断延迟发布
487     if (p_tcb == &OSIntQTaskTCB) {                  //如果目标任务是中断延迟提交任务
488         *p_err = OS_ERR_TASK_DEL_INVALID;           //错误类型为“非法删除任务”
489          return;                                    //返回，停止执行
490     }
491 #endif
492
493     if (p_tcb == (OS_TCB *)0) {                     //如果 p_tcb 为空
494         CPU_CRITICAL_ENTER();                       //关中断
495         p_tcb  = OSTCBCurPtr;                       //目标任务设为自身
496         CPU_CRITICAL_EXIT();                        //开中断
497     }
498
```

```
499     OS_CRITICAL_ENTER();                            //进入临界段
500     switch (p_tcb->TaskState) {                     //根据目标任务的任务状态分类处理
501         case OS_TASK_STATE_RDY:                     //如果是就绪状态
502              OS_RdyListRemove(p_tcb);               //将任务从就绪列表移除
503              break;                                 //跳出
504
505         case OS_TASK_STATE_SUSPENDED:               //如果是挂起状态
506              break;                                 //直接跳出
507
508         case OS_TASK_STATE_DLY:                     //如果包含延时状态
509         case OS_TASK_STATE_DLY_SUSPENDED:
510              OS_TickListRemove(p_tcb);              //将任务从节拍列表移除
511              break;                                 //跳出
512
```

```
513          case OS_TASK_STATE_PEND:                   //如果包含等待状态
514          case OS_TASK_STATE_PEND_SUSPENDED:
515          case OS_TASK_STATE_PEND_TIMEOUT:
516          case OS_TASK_STATE_PEND_TIMEOUT_SUSPENDED:
517               OS_TickListRemove(p_tcb);             //将任务从节拍列表移除
518               switch (p_tcb->PendOn) {              //根据任务的等待对象分类处理
519                   case OS_TASK_PEND_ON_NOTHING:     //如果没在等待内核对象
520                   case OS_TASK_PEND_ON_TASK_Q:      //如果等待的是任务消息队列
521                   case OS_TASK_PEND_ON_TASK_SEM:    //如果等待的是任务信号量
522                        break;                       //直接跳出
523
524                   case OS_TASK_PEND_ON_FLAG:        //如果等待的是事件标志组
525                   case OS_TASK_PEND_ON_MULTI:       //如果等待多个内核对象
526                   case OS_TASK_PEND_ON_MUTEX:       //如果等待的是互斥信号量
527                   case OS_TASK_PEND_ON_Q:           //如果等待的是消息队列
528                   case OS_TASK_PEND_ON_SEM:         //如果等待的是多值信号量
529                        OS_PendListRemove(p_tcb);    //将任务从等待列表移除
530                        break;                       //跳出
531
532                   default:                          //如果等待对象超出预期
533                        break;                       //直接跳出
534               }
535               break;                                //跳出
536
```

```
537         default:                                            //如果目标任务状态超出预期
538             OS_CRITICAL_EXIT();                             //退出临界段
539            *p_err = OS_ERR_STATE_INVALID;                   //错误类型为“状态非法”
540             return;                                         //返回，停止执行
541     }
542
543 #if OS_CFG_TASK_Q_EN > 0u                                   //如果使能了任务消息队列
544     (void)OS_MsgQFreeAll(&p_tcb->MsgQ);                     //释放任务的所有任务消息
545 #endif
546
547     OSTaskDelHook(p_tcb);                                   //调用用户自定义的钩子函数
548
549 #if defined(OS_CFG_TLS_TBL_SIZE) && (OS_CFG_TLS_TBL_SIZE > 0u)
550     OS_TLS_TaskDel(p_tcb);                                          /* Call TLS hook
551 #endif
552
553 #if OS_CFG_DBG_EN > 0u                                      //如果使能了调试代码和变量
554     OS_TaskDbgListRemove(p_tcb);                            //将任务从任务调试双向列表移除
555 #endif
556     OSTaskQty--;                                            //任务数目减1
557
558     OS_TaskInitTCB(p_tcb);                                  //初始化任务控制块
559     p_tcb->TaskState = (OS_STATE)OS_TASK_STATE_DEL;//标定任务已被删除
560
561     OS_CRITICAL_EXIT_NO_SCHED();                            //退出临界段（无调度）
562
563    *p_err = OS_ERR_NONE;                                    //错误类型为“无错误”
564
565     OSSched();                                              //调度任务
566 }
567 #endif
```

图 14-11 OSTaskDel () 函数

14.1.6 OSSchedRoundRobinCfg ()

当有任务使用相同的优先级的时候，一般需要使用时间片轮转调度。当具有相同优先级的多个任务就绪时，系统会根据分配给它们的时间片轮流调度各个任务运行。要使用时间片轮转调度功能，除了要先使能 OS_CFG_SCHED_ROUND_ROBIN_EN（位于“os_cfg.h”）外，还需调用 OSSchedRoundRobinCfg() 函数使能时间片轮转调度和配置相关指标。

OS_CFG_SCHED_ROUND_ROBIN_EN 位于“os_cfg.h”，如下图所示。

```
_task.c  app.c  os_cfg.h  os_core.c
32
33                                          /* ---------------------------- MISCELLANEOUS --
34  #define OS_CFG_APP_HOOKS_EN             1u   //使能/禁用钩子函数
35  #define OS_CFG_ARG_CHK_EN               1u   //使能/禁用参数检测
36  #define OS_CFG_CALLED_FROM_ISR_CHK_EN   1u   //使能/禁用检测中断中非法调用功能
37  #define OS_CFG_DBG_EN                   1u   //使能/禁用调试代码和变量
38  #define OS_CFG_ISR_POST_DEFERRED_EN     1u   //使能/禁用中断延迟发布
39  #define OS_CFG_OBJ_TYPE_CHK_EN          1u   //使能/禁用对象类型检测
40  #define OS_CFG_TS_EN                    1u   //使能/禁用时间戳
41
42  #define OS_CFG_PEND_MULTI_EN            1u   //使能/禁用等待多个内核对象
43
44  #define OS_CFG_PRIO_MAX                32u   //定义任务优先级的最大数目
45
46  #define OS_CFG_SCHED_LOCK_TIME_MEAS_EN  1u   //使能/禁用测量锁调度器时间功能
47  #define OS_CFG_SCHED_ROUND_ROBIN_EN     1u   //使能/禁用时间片轮转调度
48  #define OS_CFG_STK_SIZE_MIN            64u   //定义任务堆栈最小空间（单位：CPU_STK）
49
```

图 14-12

OSSchedRoundRobinCfg() 函数的信息如下表所示。

表 50 OS_TaskResume ()

函数原型	void OSSchedRoundRobinCfg (CPU_BOOLEAN en, OS_TICK dflt_time_quanta, OS_ERR *p_err);			
功能	配置时间片轮转调度。			
参数	en	使能/禁用事件片轮转调度	DEF_ENABLED	使能事件片轮转调度。
			DEF_DISABLED	禁用事件片轮转调度。
	dflt_time_quanta	设置默认时间片	>0	把 dflt_time_quanta 设为默认时间片值。
			=0	把系统默认值 OSCfg_TickRate_Hz / 10 设为默认时间片值。
	p_err	返回错误类型	OS_ERR_NONE	无错误，配置成功。
返回值	无。			
注意事项	✧ 当 OS_CFG_ISR_POST_DEFERRED_EN = 0u 时，不能在中断中调用该函数。			

OSSchedRoundRobinCfg() 函数的定义位于“os_core.c

```
571 #if OS_CFG_SCHED_ROUND_ROBIN_EN > 0u                          //如果使能了时间片轮转调度
572 void  OSSchedRoundRobinCfg (CPU_BOOLEAN   en,                 //使能/禁用时间片轮转调度
573                             OS_TICK      dflt_time_quanta,  //默认时间片,0表使用系统值
574                             OS_ERR      *p_err)             //返回错误类型
575 {
576     CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必需该宏，该宏声明和
577                     //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
578                     // SR（临界段关中断只需保存SR），开中断时将该值还原。
579
580 #ifdef OS_SAFETY_CRITICAL                                     //如果使能了安全检测
581     if (p_err == (OS_ERR *)0) {                               //如果 p_err 为空
582         OS_SAFETY_CRITICAL_EXCEPTION();                       //执行安全检测异常函数
583         return;                                               //返回，停止执行
584     }
585 #endif
586
587     CPU_CRITICAL_ENTER();                                      //关中断
588     if (en != DEF_ENABLED) {                                   //禁用时间片轮转调度
589         OSSchedRoundRobinEn = DEF_DISABLED;
590     } else {                                                   //使能时间片轮转调度
591         OSSchedRoundRobinEn = DEF_ENABLED;
592     }
593
594     if (dflt_time_quanta > (OS_TICK)0) {                       //用户自定义默认时间片
595         OSSchedRoundRobinDfltTimeQuanta = dflt_time_quanta;
596     } else {                                                   //使用系统值作为默认时间片
597         OSSchedRoundRobinDfltTimeQuanta = (OS_TICK)(OSCfg_TickRate_Hz / (OS_RATE_HZ)10);
598     }
599     CPU_CRITICAL_EXIT();                                       //开中断
600    *p_err = OS_ERR_NONE;                                       //错误类型为“无错误”
601 }
602 #endif
```

图 14-13 OSSchedRoundRobinCfg() 函数

14.1.7 OSSchedRoundRobinYield ()

一个任务也可以主动放弃时间片，当一个任务已经完成要执行的事情后，也可以主动放弃时间片，提前退出运行，让就绪列表（处于就绪状态的同一优先级任务）的下一个任务提前运行。但是，如果就绪列表中只有一个任务，该任务无法放弃时间片。一个任务放弃时间片使用 OSSchedRoundRobinYield () 函数，OSSchedRoundRobinYield () 函数的信息如下表所示。

表 51 OSSchedRoundRobinYield ()

函数原型	OSSchedRoundRobinYield (OS_ERR *p_err);			
功能	放弃时间片。			
参数	p_err	返回错误类型	OS_ERR_NONE	无错误，调用成功。
			OS_ERR_ROUND_ROBIN_1	当前优先级的就绪列表中只有一个任务（当前任务）。
			OS_ERR_ROUND_ROBIN_DISABLED	未使能时间片轮转调度。
			OS_ERR_SCHED_LOCKED	调度器被锁。
			OS_ERR_YIELD_ISR	在中断中调用该函数。

返回值	无。
注意事项	✧ 不能在中断中调用该函数。

OSSchedRoundRobinYield () 函数的定义位于“os_core.c

```
625 #if OS_CFG_SCHED_ROUND_ROBIN_EN > 0u                 //如果使能了时间片轮转调度
626 void  OSSchedRoundRobinYield (OS_ERR  *p_err)       //返回错误类型
627 {
628     OS_RDY_LIST  *p_rdy_list;
629     OS_TCB       *p_tcb;
630     CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必需该宏，该宏声明和
631                     //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
632                     // SR（临界段关中断只需保存SR），开中断时将该值还原。
633
634 #ifdef OS_SAFETY_CRITICAL                           //如果使能了安全检测
635     if (p_err == (OS_ERR *)0) {                     //如果 p_err 为空
636         OS_SAFETY_CRITICAL_EXCEPTION();             //执行安全检测异常函数
637         return;                                     //返回，停止执行
638     }
639 #endif
640
641 #if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u              //如果使能了中断中非法调用检测
642     if (OSIntNestingCtr > (OS_NESTING_CTR)0) {      //如果该函数是在中断中被调用
643         *p_err = OS_ERR_YIELD_ISR;                  //错误类型为“在中断中放弃时间片”
644         return;                                     //返回，停止执行
645     }
646 #endif
647
```

```
648     if (OSSchedLockNestingCtr > (OS_NESTING_CTR)0) { //如果调度器被锁
649         *p_err = OS_ERR_SCHED_LOCKED;               //错误类型为“调度器被锁”
650         return;                                     //返回，停止执行
651     }
652
653     if (OSSchedRoundRobinEn != DEF_TRUE) {          //如果禁用了时间轮转调度
654         *p_err = OS_ERR_ROUND_ROBIN_DISABLED;       //错误类型为“未使能时间片轮转调度”
655         return;                                     //返回，停止执行
656     }
657
658     CPU_CRITICAL_ENTER();                           //关中断
659     p_rdy_list = &OSRdyList[OSPrioCur];             //获取当前优先级的就绪列表
660     if (p_rdy_list->NbrEntries < (OS_OBJ_QTY)2) {   //如果就绪列表只有1个任务
661         CPU_CRITICAL_EXIT();                        //开中断
662         *p_err = OS_ERR_ROUND_ROBIN_1;              //错误类型为“就绪列表只有1个任务”
663         return;                                     //返回，停止执行
664     }
```

```
665     /* 如果具有该优先级的任务不止1个，停止运行当前任务 */
666     OS_RdyListMoveHeadToTail(p_rdy_list);               //把当前任务移到就绪列表的尾端
667     p_tcb = p_rdy_list->HeadPtr;                        //获取就绪列表头端任务（准备运行）
668     if (p_tcb->TimeQuanta == (OS_TICK)0) {              //使用默认的时间片
669         p_tcb->TimeQuantaCtr = OSSchedRoundRobinDfltTimeQuanta;
670     } else {                                            //使用自定义的时间片
671         p_tcb->TimeQuantaCtr = p_tcb->TimeQuanta;
672     }
673
674     CPU_CRITICAL_EXIT();                                //开中断
675
676     OSSched();                                          //调度任务（头端任务运行）
677    *p_err = OS_ERR_NONE;                                //错误类型为“无错误”
678 }
679 #endif
```

图 14-14 OSSchedRoundRobinYield () 函数

14.1.8 OSTaskTimeQuantaSet ()

在创建任务的时候，可以通过设置参数 time_quanta 来设置任务的时间片。在创建完任务后，还可以通过 OSTaskTimeQuantaSet () 函数调整任务的时间片。OSTaskTimeQuantaSet () 函数的信息如下所示。

表 52 OSTaskTimeQuantaSet ()

<table>
<tr><td>函数原型</td><td colspan="4">void OSTaskTimeQuantaSet (OS_TCB *p_tcb,
OS_TICK time_quanta,
OS_ERR *p_err);</td></tr>
<tr><td>功能</td><td colspan="4">设置一个任务的时间片。</td></tr>
<tr><td rowspan="4">参数</td><td>p_tcb</td><td colspan="3">目标任务的任务控制块指针，0 表自身。</td></tr>
<tr><td rowspan="2">time_quanta</td><td rowspan="2">时间片（单位：时钟节拍）</td><td>>0</td><td>把 time_quanta 设为任务的时间片。</td></tr>
<tr><td>=0</td><td>把时间片默认值设为任务的时间片。</td></tr>
<tr><td>p_err</td><td rowspan="2">返回错误类型</td><td>OS_ERR_NONE</td><td>无错误，调用成功。</td></tr>
<tr><td>OS_ERR_SET_ISR</td><td>在中断中调用该函数。</td></tr>
<tr><td>返回值</td><td colspan="4">无。</td></tr>
<tr><td>注意事项</td><td colspan="4">✧ 不能在中断中调用该函数。</td></tr>
</table>

OSTaskTimeQuantaSet () 函数的定义位于“os_task.c”。

os_task.c | app.c* | os_cfg.h | os_core.c | os.h

```
1805 #if OS_CFG_SCHED_ROUND_ROBIN_EN > 0u                    //如果使能了时间片轮转调度
1806 void  OSTaskTimeQuantaSet (OS_TCB   *p_tcb,             //任务控制块指针
1807                           OS_TICK   time_quanta,        //设置时间片
1808                           OS_ERR   *p_err)              //返回错误类型
1809 {
1810     CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必需该宏，该宏声明和
1811                     //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
1812                     // SR（临界段关中断只需保存SR），开中断时将该值还原。
1813
1814 #ifdef OS_SAFETY_CRITICAL                   //如果使能（默认禁用）了安全检测
1815     if (p_err == (OS_ERR *)0) {             //如果 p_err 为空
1816         OS_SAFETY_CRITICAL_EXCEPTION();     //执行安全检测异常函数
1817         return;                             //返回，停止执行
1818     }
1819 #endif
1820
1821 #if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u          //如果使能了中断中非法调用检测
1822     if (OSIntNestingCtr > (OS_NESTING_CTR)0) { //如果该函数在中断中被调用
1823        *p_err = OS_ERR_SET_ISR;                 //错误类型为“在中断中设置”
1824         return;                                 //返回，停止执行
1825     }
1826 #endif
1827
```

```
1828     CPU_CRITICAL_ENTER();                       //关中断
1829     if (p_tcb == (OS_TCB *)0) {                 //如果 p_tcb 为空
1830         p_tcb = OSTCBCurPtr;                    //目标任务为当前运行任务
1831     }
1832
1833     if (time_quanta == 0u) {                    //如果 time_quanta 为0
1834         p_tcb->TimeQuanta    = OSSchedRoundRobinDfltTimeQuanta; //设置为默认时间片
1835     } else {                                    //如果 time_quanta 非0
1836         p_tcb->TimeQuanta    = time_quanta;     //把 time_quanta 设为任务的时间片
1837     }
1838     if (p_tcb->TimeQuanta > p_tcb->TimeQuantaCtr) { //如果任务的剩余时间片小于新设置值
1839         p_tcb->TimeQuantaCtr = p_tcb->TimeQuanta;   //把剩余时间片调整为新设置值
1840     }
1841     CPU_CRITICAL_EXIT();                        //开中断
1842    *p_err = OS_ERR_NONE;                        //错误类型为“无错误”
1843 }
1844 #endif
```

图 14-15 OSTaskTimeQuantaSet () 函数

14.1.9 OSTaskRegSet ()

UC/OS-III 允许任务拥有给用户自己使用的任务寄存器。所谓任务寄存器，其实就是在任务的任务控制块里增加一个无符号 32 位整型的数组，用来给用户存放一些任务相关的数据。在创建任务时，会将任务寄存器全部置 0。要使用任务寄存器，还得事先通过 OS_CFG_TASK_REG_TBL_SIZE（位于“os_cfg.h”）定义任务的任务寄存器数目，如下所示。

```
82                                                  /* ---------------------------- TASK MANAGEMENT ----
83  #define OS_CFG_STAT_TASK_EN                1u   //使能/禁用统计任务
84  #define OS_CFG_STAT_TASK_STK_CHK_EN        1u   //使能/禁用从统计任务检查堆栈
85
86  #define OS_CFG_TASK_CHANGE_PRIO_EN         1u   //使能/禁用函数 OSTaskChangePrio()
87  #define OS_CFG_TASK_DEL_EN                 1u   //使能/禁用函数 OSTaskDel()
88  #define OS_CFG_TASK_Q_EN                   1u   //使能/禁用函数 OSTaskQXXXX()
89  #define OS_CFG_TASK_Q_PEND_ABORT_EN        1u   //使能/禁用函数 OSTaskQPendAbort()
90  #define OS_CFG_TASK_PROFILE_EN             1u   //使能/禁用任务控制块的简况变量
91  #define OS_CFG_TASK_REG_TBL_SIZE           1u   //定义任务寄存器的数目
92  #define OS_CFG_TASK_SEM_PEND_ABORT_EN      1u   //使能/禁用函数 OSTaskSemPendAbort()
93  #define OS_CFG_TASK_SUSPEND_EN             1u   //使能/禁用函数 OSTaskSuspend() 和 OSTaskResume()
94
```

图 14-16

通过 OSTaskRegSet () 函数可以设置一个任务的任务寄存器的值。OSTaskRegSet () 函数的信息如下所示。

表 53 OSTaskRegSet ()

函数原型	void OSTaskRegSet (OS_TCB *p_tcb, OS_REG_ID id, OS_REG value, OS_ERR *p_err);			
功能	设置一个任务的某个任务寄存器的值。			
参数	p_tcb	目标任务的任务控制块指针，0 表自身。		
	id	任务寄存器的 id，取值范围为[0，OS_CFG_TASK_REG_TBL_SIZE – 1]。		
	value	设置任务寄存器的内容。		
	p_err	返回错误类型	OS_ERR_NONE	无错误，调用成功。
			OS_ERR_REG_ID_INVALID	Id 不在[0，OS_CFG_TASK_REG_TBL_SIZE – 1]内。
返回值	无。			
注意事项	✧ id 必须不超出[0，OS_CFG_TASK_REG_TBL_SIZE – 1]。			

OSTaskRegSet () 函数的定义位于“os_task.c”。

```
1119 #if OS_CFG_TASK_REG_TBL_SIZE > 0u          //如果为任务定义了任务寄存器
1120 void  OSTaskRegSet (OS_TCB     *p_tcb,    //目标任务控制块的指针
1121                     OS_REG_ID   id,        //任务寄存器的id（数组下标）
1122                     OS_REG      value,     //设置任务寄存器的内容
1123                     OS_ERR     *p_err)     //返回错误类型
1124 {
1125     CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必需该宏，该宏声明和
1126                     //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
1127                     // SR（临界段关中断只需保存SR），开中断时将该值还原。
1128
1129 #ifdef OS_SAFETY_CRITICAL                   //如果使能了安全检测
1130     if (p_err == (OS_ERR *)0) {             //如果 p_err 为空
1131         OS_SAFETY_CRITICAL_EXCEPTION();     //执行安全检测异常函数
1132         return;                             //返回，停止执行
1133     }
1134 #endif
1135
1136 #if OS_CFG_ARG_CHK_EN > 0u                  //如果使能了参数检测
1137     if (id >= OS_CFG_TASK_REG_TBL_SIZE) {   //如果 id 超出范围
1138        *p_err = OS_ERR_REG_ID_INVALID;      //错误类型为“id 非法”
1139         return;                             //返回，停止执行
1140     }
1141 #endif
1142
1143     CPU_CRITICAL_ENTER();                   //关中断
1144     if (p_tcb == (OS_TCB *)0) {             //如果 p_tcb 为空
1145         p_tcb = OSTCBCurPtr;                //目标任务为当前运行任务（自身）
1146     }
1147     p_tcb->RegTbl[id] = value;              //设置任务寄存器的内容
1148     CPU_CRITICAL_EXIT();                    //开中断
1149    *p_err             = OS_ERR_NONE;       //错误类型为“无错误”
1150 }
1151 #endif
```

图 14-17 OSTaskRegSet () 函数

14.1.10 OSTaskRegSet ()

UC/OS-III 允许任务拥有给用户自己使用的任务寄存器。所谓任务寄存器，其实就是在任务的任务控制块里增加一个无符号 32 位整型的数组，用来给用户存放一些任务相关的数据。要使用任务寄存器，还得事先通过 OS_CFG_TASK_REG_TBL_SIZE（位于“os_cfg.h”）定义任务的任务寄存器数目，如下所示。

```
82                                      /* ------------------------- TASK MANAGEMENT ----
83 #define OS_CFG_STAT_TASK_EN             1u   //使能/禁用统计任务
84 #define OS_CFG_STAT_TASK_STK_CHK_EN     1u   //使能/禁用从统计任务检查堆栈
85
86 #define OS_CFG_TASK_CHANGE_PRIO_EN      1u   //使能/禁用函数 OSTaskChangePrio()
87 #define OS_CFG_TASK_DEL_EN              1u   //使能/禁用函数 OSTaskDel()
88 #define OS_CFG_TASK_Q_EN                1u   //使能/禁用函数 OSTaskQXXXX()
89 #define OS_CFG_TASK_Q_PEND_ABORT_EN     1u   //使能/禁用函数 OSTaskQPendAbort()
90 #define OS_CFG_TASK_PROFILE_EN          1u   //使能/禁用任务控制块的简况变量
91 #define OS_CFG_TASK_REG_TBL_SIZE        1u   //定义任务寄存器的数目
92 #define OS_CFG_TASK_SEM_PEND_ABORT_EN   1u   //使能/禁用函数 OSTaskSemPendAbort()
93 #define OS_CFG_TASK_SUSPEND_EN          1u   //使能/禁用函数 OSTaskSuspend() 和 OSTaskResume()
94
```

图 14-18

通过 OSTaskRegGet () 函数可以获取一个任务的任务寄存器的值。OSTaskRegSet () 函数

的信息如下所示。

表 54 OSTaskRegSet ()

函数原型	OS_REG OSTaskRegGet (OS_TCB *p_tcb, OS_REG_ID id, OS_ERR *p_err);			
功能	获取一个任务的某个任务寄存器的值。			
参数	p_tcb	目标任务的任务控制块指针，0 表自身。		
	id	任务寄存器的 id，取值范围为[0，OS_CFG_TASK_REG_TBL_SIZE – 1]。		
	p_err	返回错误类型	OS_ERR_NONE	无错误，调用成功。
			OS_ERR_REG_ID_INVALID	Id 不在[0，OS_CFG_TASK_REG_TBL_SIZE – 1]内。
返回值	✧ 0，获取的任务寄存器的值为 0（p_err == OS_ERR_NONE），或有错误（p_err != OS_ERR_NONE）。 ✧ 其他，获取的任务寄存器的值。			
注意事项	✧ id 必须不超出[0，OS_CFG_TASK_REG_TBL_SIZE – 1]。			

OSTaskRegGet () 函数的定义位于“os_task.c”。

app.c | os_stat.c | os_task.c | os_cfg.h | os.h | os_type.h

```
1010 #if OS_CFG_TASK_REG_TBL_SIZE > 0u                  //如果为任务定义了任务寄存器
1011 OS_REG  OSTaskRegGet (OS_TCB     *p_tcb,          //目标任务控制块的指针
1012                       OS_REG_ID   id,             //任务寄存器的id（数组下标）
1013                       OS_ERR     *p_err)          //返回错误类型
1014 {
1015     OS_REG     value;
1016     CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必需该宏，该宏声明和
1017                     //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
1018                     // SR（临界段关中断只需保存SR），开中断时将该值还原。
1019
1020 #ifdef OS_SAFETY_CRITICAL                          //如果使能了安全检测
1021     if (p_err == (OS_ERR *)0) {                    //如果 p_err 为空
1022         OS_SAFETY_CRITICAL_EXCEPTION();            //执行安全检测异常函数
1023         return ((OS_REG)0);                        //返回0（有错误），停止执行
1024     }
1025 #endif
1026
1027 #if OS_CFG_ARG_CHK_EN > 0u                         //如果使能了参数检测
1028     if (id >= OS_CFG_TASK_REG_TBL_SIZE) {          //如果 id 超出范围
1029        *p_err = OS_ERR_REG_ID_INVALID;             //错误类型为“id 非法”
1030         return ((OS_REG)0);                        //返回0（有错误），停止执行
1031     }
1032 #endif
```

```
1033
1034      CPU_CRITICAL_ENTER();                    //关中断
1035      if (p_tcb == (OS_TCB *)0) {              //如果 p_tcb 为空
1036          p_tcb = OSTCBCurPtr;                 //目标任务为当前运行任务（自身）
1037      }
1038      value = p_tcb->RegTbl[id];               //获取任务寄存器的内容
1039      CPU_CRITICAL_EXIT();                     //开中断
1040     *p_err = OS_ERR_NONE;                     //错误类型为“无错误”
1041      return ((OS_REG)value);                  //返回任务寄存器的内容
1042  }
1043  #endif
```

图 14-19 OSTaskRegGet () 函数

14.2 实例演示

14.2.1 实例 1

任务的创建和使用在前面实例中均有出现，本节实例着重演示任务的挂起和恢复，另外还有其他函数的使用。本节实例创建 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 灯的亮灭状态。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。

该例程已经存放在配套资料的下图路径。

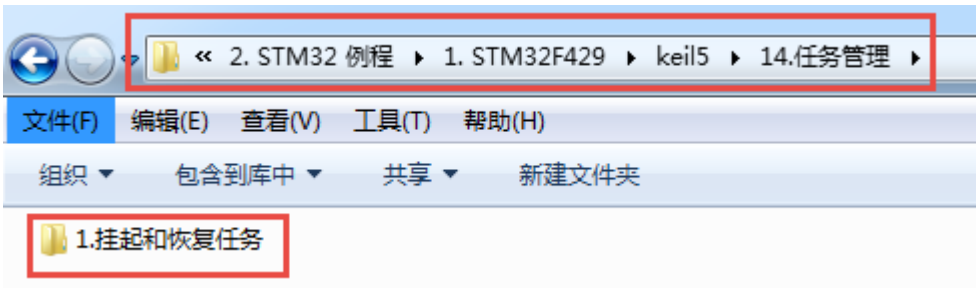

图 14-20 例程路径

本例程需用使用 LED，所以工程中需要添加其驱动文件和初始化。用户可参照前面相关实例，这里不再赘述。

在起始任务 AppTaskStart() 中，在创建应用任务前，配置了时间片轮转调度的指标。

```
170
171
172     /* 配置时间片轮转调度 */
173     OSSchedRoundRobinCfg((CPU_BOOLEAN   )DEF_ENABLED,          //使能时间片轮转调度
174                          (OS_TICK       )0,                    //把 OSCfg_TickRate_Hz / 10 设为默认时间片值
175                          (OS_ERR        *)&err );              //返回错误类型
176
```

图 14-21 配置时间片轮转调度

LED1 任务的任务函数 AppTaskLed1 () 的定义如下。LED1 切换 10 次后就帮助恢复 LED2 和 LED3 任务。任务中使用任务寄存器来计数 LED1 的切换次数，LED2 和 LED3 任务也是如此。

```
app_cfg.h | app.c | os_task.c | os_core.c
231 *********************************************************************************************************
232 *                                                LED1 TASK
233 *********************************************************************************************************
234 */
235
236 static  void  AppTaskLed1 ( void * p_arg )
237 {
238   OS_ERR      err;
239   OS_REG      value;
240
241
242   (void)p_arg;
243
244
245   while (DEF_TRUE) {                                    //任务体，通常写成一个死循环
246     macLED1_TOGGLE ();                                  //切换 LED1 的亮灭状态
247
248     value = OSTaskRegGet ( 0, 0, & err );               //获取自身任务寄存器值
249
250     if ( value < 10 )                                   //如果任务寄存器值<10
251     {
252       OSTaskRegSet ( 0, 0, ++ value, & err );           //继续累加任务寄存器值
253     }
254     else                                                //如果累加到10
255     {
256       OSTaskRegSet ( 0, 0, 0, & err );                  //将任务寄存器值归0
257
258       OSTaskResume ( & AppTaskLed2TCB, & err );         //恢复 LED2 任务
259
260       OSTaskResume ( & AppTaskLed3TCB, & err );         //恢复 LED3 任务
261
262     }
263
264     OSTimeDly ( 1000, OS_OPT_TIME_DLY, & err );         //相对性延时1000个时钟节拍（1s）
265
266   }
```

图 14-22 AppTaskLed1 () 任务函数

LED2 任务的任务函数 AppTaskLed2 () 的定义如下。LED2 切换 5 次后就挂起自身，停止运行。

```
app_cfg.h | app.c | os_task.c | os_core.c
273 *********************************************************************************************************
274 *                                                LED2 TASK
275 *********************************************************************************************************
276 */
277
278 static  void  AppTaskLed2 ( void * p_arg )
279 {
280   OS_ERR      err;
281   OS_REG      value;
282
283
284   (void)p_arg;
285
286
```

```
287   while (DEF_TRUE) {                                   //任务体，通常写成一个死循环
288     macLED2_TOGGLE ();                                 //切换 LED2 的亮灭状态
289
290     value = OSTaskRegGet ( 0, 0, & err );              //获取自身任务寄存器值
291
292     if ( value < 5 )                                   //如果任务寄存器值<5
293     {
294       OSTaskRegSet ( 0, 0, ++ value, & err );          //继续累加任务寄存器值
295     }
296     else                                               //如果累加到5
297     {
298       OSTaskRegSet ( 0, 0, 0, & err );                 //将任务寄存器值归0
299
300       OSTaskSuspend ( 0, & err );                      //挂起自身
301
302     }
303
304     OSTimeDly ( 1000, OS_OPT_TIME_DLY, & err );        //相对性延时1000个时钟节拍（1s）
305
306   }
```

图 14-23 AppTaskLed2 () 任务函数

LED3 任务的任务函数 AppTaskLed3 () 的定义如下。LED3 和 LED2 任务是一样的，LED3 切换 5 次后就挂起自身，停止运行。

```
313 *********************************************************************************************
314 *                                              LED3 TASK
315 *********************************************************************************************
316 */
317
318 static  void  AppTaskLed3 ( void * p_arg )
319 {
320   OS_ERR      err;
321   OS_REG      value;
322
323
324   (void)p_arg;
325
326
```

```
327   while (DEF_TRUE) {                                   //任务体，通常写成一个死循环
328     macLED3_TOGGLE ();                                 //切换 LED3 的亮灭状态
329
330     value = OSTaskRegGet ( 0, 0, & err );              //获取自身任务寄存器值
331
332     if ( value < 5 )                                   //如果任务寄存器值<5
333     {
334       OSTaskRegSet ( 0, 0, ++ value, & err );          //继续累加任务寄存器值
335     }
336     else                                               //如果累加到5
337     {
338       OSTaskRegSet ( 0, 0, 0, & err );                 //将任务寄存器值归零
339
340       OSTaskSuspend ( 0, & err );                      //挂起自身
341
342     }
343
344     OSTimeDly ( 1000, OS_OPT_TIME_DLY, & err );        //相对性延时1000个时钟节拍（1s）
345
346   }
347
348
349 }
```

图 14-24 AppTaskLed3 () 任务函数

编译和下载程序到秉火 STM32 开发板，运行程序。用户可以看到，一开始 LED1、LED2 和 LED3 均每隔 1s 切换一次亮灭状态，当 LED2 和 LED3 均切换 5 次时，就停止切换。而 LED1

继续切换，待 LED1 切换 10 次时，LED2 和 LED3 又得以继续切换。依此循环。

14.3 章末总结

任务管理是 uC/OS 操作系统的核心技术。uC/OS-III 允许两个任务拥有相同的优先级，而且可以使用时间片轮转调度同优先级任务运行。

使用 OSTaskCreate () 函数就可以创建一个任务。OSTaskSuspend () 函数用于挂起一个任务，使用该函数可以多次挂起一个任务，也就是嵌套挂起。但是嵌套多少次，就必须使用相同次数 OSTaskResume () 函数来解嵌该任务，使该任务恢复被挂起前的状态。创建任务时，可以通过设置有关参数给定任务的优先级，在创建完任务后，还可以使用 OSTaskChangePrio () 函数修改任务的优先级。当一个任务不再使用时，可以使用 OSTaskDel() 函数将其删除。

当应用程序中使用到相同优先级的任务时，建议使用时间片轮转调度它们。要使用时间片轮转调度功能，需要调用 OSSchedRoundRobinCfg() 函数使能时间片轮转调度和配置默认时间片大小。在创建任务时，可以通过设置有关参数给定任务的时间片，在创建完任务后，还可以使用 OSTaskTimeQuantaSet () 函数修改任务的时间片。当一个任务提前完成要处理的事情时，也可以调用 OSSchedRoundRobinYield () 函数主动放弃剩余的时间片，让下一个就绪的同优先级任务提前运行，但前提是该优先级就绪列表中不止一个任务。

第15章 中断管理

在 uC/OS 系统中，中断相当于一个优先级最高的任务。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，带退出中断后再做详细处理。在使用 uC/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，处理任务再做详细处理。

15.1 原理简述

在使用 uC/OS 系统时，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，处理任务再做纤细处理。在使用 uC/OS 系统时，中断的处理一般是先在中断服务函数中通过发布信号量、消息或事件标志组等内核对象来标志中断的发生，等退出中断后再由相关处理任务详细处理中断。根据这些内核对象的发布大致可以分为两种情况，一种是在中断中直接发布，另一种是退出中断后再发布，也就是中断延迟发布。通过宏 OS_CFG_ISR_POST_DEFERRED_EN（位于“os_cfg.h”）可以使能或禁用中断延迟发布，如下所示。

```
33                                                  /* ---------------------------- MISCELLANEOUS ---
34  #define  OS_CFG_APP_HOOKS_EN                1u  //使能/禁用钩子函数
35  #define  OS_CFG_ARG_CHK_EN                  1u  //使能/禁用参数检测
36  #define  OS_CFG_CALLED_FROM_ISR_CHK_EN      1u  //使能/禁用检测中断中非法调用功能
37  #define  OS_CFG_DBG_EN                      1u  //使能/禁用调试代码和变量
38  #define  OS_CFG_ISR_POST_DEFERRED_EN        1u  //使能/禁用中断延迟发布
39  #define  OS_CFG_OBJ_TYPE_CHK_EN             1u  //使能/禁用对象类型检测
40  #define  OS_CFG_TS_EN                       1u  //使能/禁用时间戳
41
```

图 15-1 使能或禁用中断延迟发布

使能中断延时发布，可以将中断级任务转换成任务级任务，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。在前面提到的 OSTimeTick()、OSSemPost()、OSQPost()、OSFlagPost()、OSTaskSemPost()、OSTaskQPost()、OSTaskSuspend()和 OSTaskResume() 这些函数，在使能中断延时发布后，如果在中断中调用这些函数，不会直接执行相关功能操作，而是先使用 OS_IntQPost() 函数（位于“os_int.c”）把这些事件发布到中断队列记录起来，待退出中断后，就会调度具有最高优先级的中断延迟提交任务 OS_IntQTask() 函数（位于“os_int.c”）真正处理这些事件。

OS_IntQPost() 函数的定义位于“os_int.c”。

```
 91 void  OS_IntQPost (OS_OBJ_TYPE    type,          //内核对象类型
 92                    void          *p_obj,         //被发布的内核对象
 93                    void          *p_void,        //消息队列或任务消息
 94                    OS_MSG_SIZE    msg_size,      //消息的数目
 95                    OS_FLAGS       flags,         //事件标志组
 96                    OS_OPT         opt,           //发布内核对象时的选项
 97                    CPU_TS         ts,            //发布内核对象时的时间戳
 98                    OS_ERR        *p_err)         //返回错误类型
 99 {
100     CPU_SR_ALLOC();  //使用到临界段（在关/开中断时）时必需该宏，该宏声明和定义一个局部变
101                      //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）
102                      //，开中断时将该值还原。
103
104 #ifdef OS_SAFETY_CRITICAL                  //如果使能（默认禁用）了安全检测
105     if (p_err == (OS_ERR *)0) {            //如果错误类型实参为空
106         OS_SAFETY_CRITICAL_EXCEPTION(); //执行安全检测异常函数
107         return;                            //返回，不继续执行
108     }
109 #endif
110
```

```
111     CPU_CRITICAL_ENTER();                                       //关中断
112     if (OSIntQNbrEntries < OSCfg_IntQSize) {                    //如果中断队列未占满
113         OSIntQNbrEntries++;
114
115         if (OSIntQNbrEntriesMax < OSIntQNbrEntries) {           //更新中断队列的最大使用数目的历史记录
116             OSIntQNbrEntriesMax = OSIntQNbrEntries;
117         }
118         /* 将要重新提交的内核对象的信息放入到中断队列入口的信息记录块 */
119         OSIntQInPtr->Type       = type;                         //保存内核对象类型
120         OSIntQInPtr->ObjPtr     = p_obj;                        //保存被发布的内核对象
121         OSIntQInPtr->MsgPtr     = p_void;                       //保存消息内容指针
122         OSIntQInPtr->MsgSize    = msg_size;                     //保存消息大小
123         OSIntQInPtr->Flags      = flags;                        //保存事件标志组
124         OSIntQInPtr->Opt        = opt;                          //保存选项
125         OSIntQInPtr->TS         = ts;                           //保存对象被发布的时间错
126
127         OSIntQInPtr             =  OSIntQInPtr->NextPtr;        //指向下一个带处理成员
128         /* 让中断队列管理任务 OSIntQTask 就绪 */
129         OSRdyList[0].NbrEntries = (OS_OBJ_QTY)1;                //更新就绪列表上的优先级0的任务数为1个
130         OSRdyList[0].HeadPtr    = &OSIntQTaskTCB;               //该就绪列表的头指针指向 OSIntQTask 任务
131         OSRdyList[0].TailPtr    = &OSIntQTaskTCB;               //该就绪列表的尾指针指向 OSIntQTask 任务
132         OS_PrioInsert(0u);                                      //在优先级列表中增加优先级0
133         if (OSPrioCur != 0) {                                   //如果当前运行的不是 OSIntQTask 任务
134             OSPrioSaved         = OSPrioCur;                    //保存当前任务的优先级
135         }
136
137        *p_err                   = OS_ERR_NONE;                  //返回错误类型为“无错误”
```

```
138     } else {                                                    //如果中断队列已占满
139         OSIntQOvfCtr++;                                         //中断队列溢出数目加1
140        *p_err                   = OS_ERR_INT_Q_FULL;            //返回错误类型为“无错误”
141     }
142     CPU_CRITICAL_EXIT();                                        //开中断
143 }
```

图 15-2 OS_IntQPost () 函数

OS_IntQTask () 函数的定义位于“os_int.c”。

```
269 void  OS_IntQTask (void  *p_arg)
270 {
271     CPU_BOOLEAN  done;
272     CPU_TS       ts_start;
273     CPU_TS       ts_end;
274     CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必需该宏，该宏声明和
275                     //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
276                     // SR（临界段关中断只需保存SR），开中断时将该值还原。
277
278     p_arg = p_arg;
```

```
279     while (DEF_ON) {                                            //进入死循环
280         done = DEF_FALSE;
281         while (done == DEF_FALSE) {
282             CPU_CRITICAL_ENTER();                               //关中断
283             if (OSIntQNbrEntries == (OS_OBJ_QTY)0u) {           //如果中断队列里的内核对象发布完毕
284                 OSRdyList[0].NbrEntries = (OS_OBJ_QTY)0u;       //从就绪列表移除中断队列管理任务 OS_IntQTask
285                 OSRdyList[0].HeadPtr    = (OS_TCB    *)0;
286                 OSRdyList[0].TailPtr    = (OS_TCB    *)0;
287                 OS_PrioRemove(0u);                              //从优先级表格移除优先级0
288                 CPU_CRITICAL_EXIT();                            //开中断
289                 OSSched();                                      //任务调度
290                 done = DEF_TRUE;                                //退出循环
291             } else {                                            //如果中断队列里还有内核对象
292                 CPU_CRITICAL_EXIT();                            //开中断
293                 ts_start = OS_TS_GET();                         //获取时间戳
294                 OS_IntQRePost();                                //发布中断队列里的内核对象
295                 ts_end   = OS_TS_GET() - ts_start;              //计算该次发布时间
296                 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
297                     OSIntQTaskTimeMax = ts_end;
298                 }
299                 CPU_CRITICAL_ENTER();                           //关中断
300                 OSIntQOutPtr = OSIntQOutPtr->NextPtr;           //中断队列出口转至下一个
301                 OSIntQNbrEntries--;                             //中断队列的成员数目减1
302                 CPU_CRITICAL_EXIT();                            //开中断
303             }
304         }
305     }
306 }
```

图 15-3 OS_IntQTask () 函数

在执行任务 OS_IntQTask () 时真正起到提交作用的是 OS_IntQRePost() 函数，OS_IntQRePost() 函数的定义也位于“os_int.c”。

```
159 void  OS_IntQRePost (void)
160 {
161     CPU_TS  ts;
162     OS_ERR  err;
163
164
165     switch (OSIntQOutPtr->Type) {    //根据内核对象类型分类处理
166         case OS_OBJ_TYPE_FLAG:       //如果对象类型是事件标志
167 #if OS_CFG_FLAG_EN > 0u              //如果使能了事件标志，则发布事件标志
168             (void)OS_FlagPost((OS_FLAG_GRP *) OSIntQOutPtr->ObjPtr,
169                               (OS_FLAGS     ) OSIntQOutPtr->Flags,
170                               (OS_OPT       ) OSIntQOutPtr->Opt,
171                               (CPU_TS       ) OSIntQOutPtr->TS,
172                               (OS_ERR      *)&err);
173 #endif
174             break;                   //跳出
175
```

```
176         case OS_OBJ_TYPE_Q:                     //如果对象类型是消息队列
177#if OS_CFG_Q_EN > 0u                             //如果使能了消息队列，则发布消息队列
178             OS_QPost((OS_Q       *) OSIntQOutPtr->ObjPtr,
179                      (void       *) OSIntQOutPtr->MsgPtr,
180                      (OS_MSG_SIZE) OSIntQOutPtr->MsgSize,
181                      (OS_OPT      ) OSIntQOutPtr->Opt,
182                      (CPU_TS      ) OSIntQOutPtr->TS,
183                      (OS_ERR     *)&err);
184#endif
185             break;                                 //跳出
186
187         case OS_OBJ_TYPE_SEM:                   //如果对象类型是多值信号量
188#if OS_CFG_SEM_EN > 0u                              //如果使能了多值信号量，则发布多值信号量
189             (void)OS_SemPost((OS_SEM *) OSIntQOutPtr->ObjPtr,
190                              (OS_OPT  ) OSIntQOutPtr->Opt,
191                              (CPU_TS  ) OSIntQOutPtr->TS,
192                              (OS_ERR *)&err);
193#endif
194             break;                                 //跳出
195

196         case OS_OBJ_TYPE_TASK_MSG:  //如果对象类型是任务消息
197#if OS_CFG_TASK_Q_EN > 0u                           //如果使能了任务消息，则发布任务消息
198             OS_TaskQPost((OS_TCB     *) OSIntQOutPtr->ObjPtr,
199                          (void       *) OSIntQOutPtr->MsgPtr,
200                          (OS_MSG_SIZE) OSIntQOutPtr->MsgSize,
201                          (OS_OPT      ) OSIntQOutPtr->Opt,
202                          (CPU_TS      ) OSIntQOutPtr->TS,
203                          (OS_ERR     *)&err);
204#endif
205             break;                                 //跳出
206
207         case OS_OBJ_TYPE_TASK_RESUME://如果对象类型是恢复任务
208#if OS_CFG_TASK_SUSPEND_EN > 0u                     //如果使能了函数OSTaskResume()，恢复该任务
209             (void)OS_TaskResume((OS_TCB *) OSIntQOutPtr->ObjPtr,
210                                 (OS_ERR *)&err);
211#endif
212             break;                                 //跳出
213

214         case OS_OBJ_TYPE_TASK_SIGNAL://如果对象类型是任务信号量
215             (void)OS_TaskSemPost((OS_TCB *) OSIntQOutPtr->ObjPtr,//发布任务信号量
216                                  (OS_OPT  ) OSIntQOutPtr->Opt,
217                                  (CPU_TS  ) OSIntQOutPtr->TS,
218                                  (OS_ERR *)&err);
219             break;                                 //跳出
220
221         case OS_OBJ_TYPE_TASK_SUSPEND://如果对象类型是挂起任务
222#if OS_CFG_TASK_SUSPEND_EN > 0u                    //如果使能了函数 OSTaskSuspend()，挂起该任务
223             (void)OS_TaskSuspend((OS_TCB *) OSIntQOutPtr->ObjPtr,
224                                  (OS_ERR *)&err);
225#endif
226             break;                                 //跳出
227
```

```
228         case OS_OBJ_TYPE_TICK:                  //如果对象类型是时钟节拍
229#if OS_CFG_SCHED_ROUND_ROBIN_EN > 0u //如果使能了时间片轮转调度，
230             OS_SchedRoundRobin(&OSRdyList[OSPrioSaved]); //轮转调度进中断前优先级任务
231 #endif
232
233             (void)OS_TaskSemPost((OS_TCB *)&OSTickTaskTCB, //发送信号量给时钟节拍任务
234                                  (OS_OPT  ) OS_OPT_POST_NONE,
235                                  (CPU_TS  ) OSIntQOutPtr->TS,
236                                  (OS_ERR *)&err);
237#if OS_CFG_TMR_EN > 0u                  //如果使能了软件定时器，发送信号量给定时器任务
238             OSTmrUpdateCtr--;
239             if (OSTmrUpdateCtr == (OS_CTR)0u) {
240                 OSTmrUpdateCtr = OSTmrUpdateCnt;
241                 ts             = OS_TS_GET();
242                 (void)OS_TaskSemPost((OS_TCB *)&OSTmrTaskTCB,
243                                      (OS_OPT  ) OS_OPT_POST_NONE,
244                                      (CPU_TS  ) ts,
245                                      (OS_ERR *)&err);
246             }
247 #endif
248             break;                      //跳出
249
250         default:                                //如果内核对象类型超出预期
251             break;                              //直接跳出
252     }
253 }
```

图 15-4 OS_IntQRePost () 函数

15.1.1 OSIntEnter ()

任务在进入中断服务函数时需要首先调用 OSIntEnter () 函数标记进入中断，方便中断嵌套管理。OSIntEnter () 函数的信息如下表所示。

表 55 OSIntEnter ()

函数原型	void OSIntEnter (void);
功能	标记进入中断服务函数。
参数	无。
返回值	无。

OSIntEnter () 函数的定义位于“os_core.c”。

```
266 void  OSIntEnter (void)
267 {
268     if (OSRunning != OS_STATE_OS_RUNNING) {          //如果操作系统还没运行
269         return;                                      //返回，停止执行
270     }
271     /* 如果操作系统已经运行 */
272     if (OSIntNestingCtr >= (OS_NESTING_CTR)250u) {   //如果中断嵌套已经达到250层
273         return;                                      //返回，停止执行
274     }
275     /* 如果中断嵌套尚未达到250层 */
276     OSIntNestingCtr++;                               //中断嵌套数目加1
277 }
```

图 15-5 OSIntEnter () 函数

15.1.2 OSIntExit ()

与 OSIntEnter () 函数相对应，任务在退出中断服务函数时需要首调用 OSIntExit () 函数标记退出中断，方便中断嵌套管理。OSIntExit () 函数的信息如下表所示。

表 56 OSIntExit ()

函数原型	void OSIntExit (void);
功能	标记退出中断服务函数。
参数	无。
返回值	无。

OSIntExit () 函数的定义位于“os_core.c”。

```
300 void  OSIntExit (void)
301 {
302     CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必需该宏，该宏声明和
303                     //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
304                     // SR（临界段关中断只需保存SR），开中断时将该值还原。
305
306     if (OSRunning != OS_STATE_OS_RUNNING) {          //如果任务尚未运行
307         return;                                      //返回，停止执行
308     }
309
310     CPU_INT_DIS();                                   //关中断
311     if (OSIntNestingCtr == (OS_NESTING_CTR)0) {      //如果该函数不是在中断内
312         CPU_INT_EN();                                //开中断
313         return;                                      //返回，停止执行
314     }
315     OSIntNestingCtr--;                               //如果该函数在中断内，中断嵌套数减1
316     if (OSIntNestingCtr > (OS_NESTING_CTR)0) {       //如果中断仍被嵌套
317         CPU_INT_EN();                                //开中断，执行上一层中断服务
318         return;                                      //返回，停止执行
319     }
```

```
320     /* 如果中断已经完全解嵌（没有上一层中断） */
321     if (OSSchedLockNestingCtr > (OS_NESTING_CTR)0) {   //如果调度器被锁
322         CPU_INT_EN();                                  //开中断
323         return;                                        //返回，停止执行
324     }
325     /* 如果调度器未被锁 */
326     OSPrioHighRdy   = OS_PrioGetHighest();             //获取就绪的最高优先级
327     OSTCBHighRdyPtr = OSRdyList[OSPrioHighRdy].HeadPtr;//提取该优先级就绪列表的头个任务
328     if (OSTCBHighRdyPtr == OSTCBCurPtr) {              //如果该任务是进中断前运行的任务
329         CPU_INT_EN();                                  //开中断
330         return;                                        //直接返回便会继续执行该任务
331     }
332     /* 如果该任务不是进中断前运行的任务 */
333 #if OS_CFG_TASK_PROFILE_EN > 0u                        //如果使能了任务控制块的简况变量
334     OSTCBHighRdyPtr->CtxSwCtr++;                       //该任务被切换的次数加1
335 #endif
336     OSTaskCtxSwCtr++;                                  //总的任务切换次数加1
337
338 #if defined(OS_CFG_TLS_TBL_SIZE) && (OS_CFG_TLS_TBL_SIZE > 0u)
339     OS_TLS_TaskSw();
340 #endif
341
342     OSIntCtxSw();                                      //中断级任务调度，调度该任务运行
343     CPU_INT_EN();                                      //开中断
344 }
```

图 15-6 OSIntExit () 函数

15.1.3 CPU_IntDisMeasMaxGet ()

关中断时间是嵌入式程序设计一个很重要的参数，uC/OS 系统也提供了测量关中断时间的机制。要使用测量关中断时间机制，必须事先使能该机制（位于“cpu_cfg.h”），如下所示。

```
134
135 #if 1          // Modified by fire （原是 0）              //使能/禁用关中断时间测量
136 #define  CPU_CFG_INT_DIS_MEAS_EN
137 #endif
138
```

图 15-7

要测量关中断时间，除了要使能测量关中断时间功能外，还须在程序初始化时调用 CPU_Init() 函数，该函数里面包括用于初始化测量关中断时间的 CPU_IntDisMeasInit() 函数。CPU_Init() 函数一般在起始任务的初始化部分调用，如下所示。

```
147 static  void  AppTaskStart (void *p_arg)
148 {
149     CPU_INT32U  cpu_clk_freq;
150     CPU_INT32U  cnts;
151     OS_ERR      err;
152
153
154     (void)p_arg;
155
156     BSP_Init();         //板级初始化
157     CPU_Init();         //初始化 CPU 组件（时间戳、关中断时间测量和主机名）
158
```

图 15-8

使用 CPU_IntDisMeasMaxGet () 函数可以获取整个程序目前的最大关中断时间，该函数

的信息如下所示。

表 57 CPU_IntDisMeasMaxGet ()

函数原型	CPU_TS_TMR CPU_IntDisMeasMaxGet (void);
功能	获取整个程序目前的最大关中断时间。
参数	无。
返回值	整个程序目前的最大关中断时间（时间戳时间）。
注意事项	✧ 返回值是一个以 CPU 时钟运行的计数值，通过 BSP_CPU_ClkFreq() 函数可以获取 CPU 时钟频率。

CPU_IntDisMeasMaxGet () 函数的定义也位于“cpu_core.c

```
797 #ifdef  CPU_CFG_INT_DIS_MEAS_EN                          //如果使能了关中断时间测量
798 CPU_TS_TMR  CPU_IntDisMeasMaxGet (void)                 //获取整个程序目前最大的关中断时间
799 {
800     CPU_TS_TMR  time_tot_cnts;
801     CPU_TS_TMR  time_max_cnts;
802     CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必需该宏，该宏声明和
803                     //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
804                     // SR（临界段关中断只需保存SR），开中断时将该值还原。
805     CPU_INT_DIS();                                       //关中断
806     time_tot_cnts = CPU_IntDisMeasMax_cnts;              //获取尚未处理的最大关中断时间
807     CPU_INT_EN();                                        //开中断
808     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
809
810     return (time_max_cnts);                              //返回目前最大关中断时间
811 }
812 #endif
```

图 15-9 CPU_IntDisMeasMaxGet () 函数

CPU_IntDisMeasMaxGet () 函数中调用了 CPU_IntDisMeasMaxCalc() 函数将测量的最大关中断减去测量时间，得到更加纯净的最大关中断时，CPU_IntDisMeasMaxCalc() 函数的定义也位于“cpu_core.c”。

```
2345 #ifdef  CPU_CFG_INT_DIS_MEAS_EN                    //如果使能了关中断时间测量
2346 static  CPU_TS_TMR  CPU_IntDisMeasMaxCalc (CPU_TS_TMR  time_tot_cnts)
2347 {
2348     CPU_TS_TMR  time_max_cnts;
2349
2350     /* 将尚未处理的最大关中断时间（包括测量时间）减去测量时间 */
2351     time_max_cnts = time_tot_cnts;
2352     if (time_max_cnts >  CPU_IntDisMeasOvrhd_cnts) {
2353         time_max_cnts -= CPU_IntDisMeasOvrhd_cnts;
2354     } else {
2355         time_max_cnts  = 0u;
2356     }
2357
2358     return (time_max_cnts);
2359 }
2360 #endif
```

图 15-10 CPU_IntDisMeasMaxCalc() 函数

15.1.4 CPU_IntDisMeasMaxCurReset ()

uC/OS 除了提供测量整个程序的最大关中断时间的用 CPU_IntDisMeasMaxGet () 函数外，还提供了测量某程序段运行过程中的最大关中断时间的功能函数。要使用该功能，同样需要进行跟 CPU_IntDisMeasMaxGet () 函数一样的初始化。使用 CPU_IntDisMeasMaxCurReset () 和 CPU_IntDisMeasMaxCurGet() 两个函数可以实现测量某程序段运行过程中的最大关中断时间。

CPU_IntDisMeasMaxCurReset () 函数为开始测量程序段的最大关中断进行初始化工作，CPU_IntDisMeasMaxCurReset () 函数的信息如下所示。

表 58 CPU_IntDisMeasMaxCurReset ()

函数原型	CPU_TS_TMR　CPU_IntDisMeasMaxCurReset (void);
功能	开始测量一个程序段的最大关中断时间。
参数	无。
返回值	上一次测量的程序段最大关中断时间（时间戳时间）。
注意事项	✧ 返回值是一个以 CPU 时钟运行的计数值，通过 BSP_CPU_ClkFreq() 函数可以获取 CPU 时钟频率。

CPU_IntDisMeasMaxCurReset () 函数的定义位于“cpu_core.c

```
712 #ifdef  CPU_CFG_INT_DIS_MEAS_EN                 //如果使能了关中断时间测量
713 CPU_TS_TMR  CPU_IntDisMeasMaxCurReset (void)    //初始化（复位）测量程序段的最大关中断时间
714 {
715     CPU_TS_TMR  time_max_cnts;
716     CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必需该宏，该宏声明和
717                     //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
718                     // SR（临界段关中断只需保存SR），开中断时将该值还原。
719     time_max_cnts = CPU_IntDisMeasMaxCurGet(); //获取复位前的程序段最大关中断时间
720     CPU_INT_DIS();                              //关中断
721     CPU_IntDisMeasMaxCur_cnts = 0u;             //清零程序段的最大关中断时间
722     CPU_INT_EN();                               //开中断
723
724     return (time_max_cnts);                     //返回复位前的程序段最大关中断时间
725 }
726 #endif
```

图 15-11 CPU_IntDisMeasMaxCurReset () 函数

15.1.5 CPU_IntDisMeasMaxCurGet ()

与 CPU_IntDisMeasMaxCurReset () 函数相对应，CPU_IntDisMeasMaxCurGet() 函数是配合其完成测量程序段的最大关中断时间。CPU_IntDisMeasMaxCurGet() 函数用于结束程序段的最大关中断时间的测量，并返回测量的时间。

CPU_IntDisMeasMaxCurGet() 函数的信息如下所示。

表 59 CPU_IntDisMeasMaxCurGet ()

函数原型	CPU_TS_TMR　CPU_IntDisMeasMaxCurGet (void)
功能	结束一个程序段的最大关中断时间的测量。
参数	无。
返回值	测量的程序段最大关中断时间（时间戳时间）。
注意事项	✧ 返回值是一个以 CPU 时钟运行的计数值，通过 BSP_CPU_ClkFreq() 函数可以获取 CPU 时钟频率。

CPU_IntDisMeasMaxCurGet () 函数的定义位于“cpu_core.c

```
754 #ifdef  CPU_CFG_INT_DIS_MEAS_EN            //如果使能了关中断时间测量
755 CPU_TS_TMR  CPU_IntDisMeasMaxCurGet (void) //获取测量的程序段的最大关中断时间
756 {
757     CPU_TS_TMR  time_tot_cnts;
758     CPU_TS_TMR  time_max_cnts;
759     CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必需该宏，该宏声明和
760                     //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
761                     // SR（临界段关中断只需保存SR），开中断时将该值还原。
762     CPU_INT_DIS();                                  //关中断
763     time_tot_cnts = CPU_IntDisMeasMaxCur_cnts;      //获取未处理的程序段最大关中断时间
764     CPU_INT_EN();                                   //开中断
765     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
766
767     return (time_max_cnts);                         //返回程序段的最大关中断时间
768 }
769 #endif
```

图 15-12 CPU_IntDisMeasMaxCurGet () 函数

CPU_IntDisMeasMaxCurGet () 函数跟 CPU_IntDisMeasMaxGet () 函数一样调用了 CPU_IntDisMeasMaxCalc() 函数将测量的最大关中断减去测量时间，得到更加纯净的最大关中断时，CPU_IntDisMeasMaxCalc() 函数的定义位于“cpu_core.c”。

```
2345 #ifdef  CPU_CFG_INT_DIS_MEAS_EN            //如果使能了关中断时间测量
2346 static  CPU_TS_TMR  CPU_IntDisMeasMaxCalc (CPU_TS_TMR  time_tot_cnts)
2347 {
2348     CPU_TS_TMR  time_max_cnts;
2349
2350     /* 将尚未处理的最大关中断时间（包括测量时间）减去测量时间 */
2351     time_max_cnts = time_tot_cnts;
2352     if (time_max_cnts >  CPU_IntDisMeasOvrhd_cnts) {
2353         time_max_cnts -= CPU_IntDisMeasOvrhd_cnts;
2354     } else {
2355         time_max_cnts  = 0u;
2356     }
2357
2358     return (time_max_cnts);
2359 }
2360 #endif
```

图 15-13 CPU_IntDisMeasMaxCalc() 函数

15.2 实例演示

15.2.1 实例 1

本实例实现串口 USART1 的接收中断和按键 KEY1 的 EXIT 外部中断。当 USART1 在中断中接收到数据时发给串口任务 AppTaskUsart() 打印出来，实现串口回显。当按键 KEY1 触发 EXIT 中断时，把中断信号发给按键任务 AppTaskKey()，该任务获取并打印最大关中断时间。

该例程已经存放在配套资料的下图路径。

图 15-14 例程路径

本例程需用使用 USART 和 EXIT，所以工程中需要添加其驱动文件和初始化。用户可参照前面相关实例，这里不再赘述。

在“stm32f4xx_it.c”文件定义了串口接收中断和 KEY1 中断。串口接收中断服务函数的定义如下。中断发生后，该中断服务函数把接收数据通过任务消息发送给任务 AppTaskUsart()打印，消息存放在内存分区 mem。

```
app.c  stm32f4xx_it.c
/**
  * @brief  USART 中断服务函数
  * @param  无
  * @retval 无
  */
void macUSART_INT_FUN(void)
{
  OS_ERR    err;

  char *    p_mem_blk;


  OSIntEnter();                                              //进入中断

  /* 从内存分区 mem 获取一个内存块 */
  p_mem_blk = OSMemGet((OS_MEM      *)&mem,
                       (OS_ERR      *)&err);

  if ( USART_GetITStatus ( macUSARTx, USART_IT_RXNE ) != RESET )
  {
    * p_mem_blk = USART_ReceiveData ( macUSARTx );           //获取接收到的数据

    /* 发布任务消息到任务 AppTaskUsart */
    OSTaskQPost ((OS_TCB      *)&AppTaskUsartTCB,            //目标任务的控制块
                 (void        *)p_mem_blk,                   //消息内容的首地址
                 (OS_MSG_SIZE  )1,                           //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

  }

  OSIntExit();                                               //退出中断

}
```

图 15-15 USART1 接收中断服务函数

KEY1 中断服务函数的定义如下。中断发生后，该中断服务函数通过任务信号量通知任务 AppTaskKey()，任务 AppTaskKey() 就打印最大关中断时间。

app.c | stm32f4xx_it.c

```
209
210 /**
211   * @brief  EXTI 中断服务函数
212   * @param  无
213   * @retval 无
214   */
215 void macEXTI_INT_FUNCTION (void)
216 {
217   OS_ERR    err;
218
219
220   OSIntEnter();                                          //进入中断
221
222   if(EXTI_GetITStatus(macEXTI_LINE) != RESET)   //确保是否产生了EXTI Line中断
223   {
224     /* 发送任务信号量到任务 AppTaskKey */
225     OSTaskSemPost((OS_TCB  *)&AppTaskKeyTCB,    //目标任务
226                   (OS_OPT   )OS_OPT_POST_NONE,  //没选项要求
227                   (OS_ERR  *)&err);             //返回错误类型
228
229     EXTI_ClearITPendingBit(macEXTI_LINE);       //清除中断标志位
230   }
231
232   OSIntExit();                                           //退出中断
233
234 }
235
```

图 15-16 KEY1 中断服务函数

串口任务 AppTaskUsart () 的定义如下。该任务接收到 USART1 接收中断发送过来的任务消息后，就打印任务消息的内容，实现串口回显。

cpu_cfg.h | app_cfg.h | app.c | os_cfg.h | os_stat.c | os_cfg_app.c | os_cfg_app.h | os_task.c | os_type.h | os.h | os_core.c | os_time.c | cpu_core.c | bsp.c | stm32f10x

```
223 ***********************************************************************************************
224 *                                          USART TASK
225 ***********************************************************************************************
226 */
227 static  void  AppTaskUsart ( void * p_arg )
228 {
229   OS_ERR         err;
230   OS_MSG_SIZE    msg_size;
231   CPU_SR_ALLOC();
232
233   char * pMsg;
234
235
236   (void)p_arg;
237
238
239   while (DEF_TRUE) {                                           //任务体
240     /* 阻塞任务，等待任务消息 */
241     pMsg = OSTaskQPend ((OS_TICK        )0,                    //无期限等待
242                         (OS_OPT         )OS_OPT_PEND_BLOCKING, //没有消息就阻塞任务
243                         (OS_MSG_SIZE   *)&msg_size,            //返回消息长度
244                         (CPU_TS        *)0,                    //返回消息被发布的时间戳
245                         (OS_ERR        *)&err);                //返回错误类型
246
247     OS_CRITICAL_ENTER();                                       //进入临界段，避免串口打印被打断
248
249     printf ( "%c", * pMsg );                                   //打印消息内容
250
251     OS_CRITICAL_EXIT();                                        //退出临界段
252
253     /* 退还内存块 */
254     OSMemPut ((OS_MEM  *)&mem,                                 //指向内存管理对象
255               (void    *)pMsg,                                 //内存块的首地址
256               (OS_ERR  *)&err);                                //返回错误类型
257
258   }
259
260 }
```

图 15-17 AppTaskUsart () 任务函数

按键任务 AppTaskKey() 的定义如下。该任务接收到任务信号量后，就获取和打印目前最大关中断时间。

bsp_exti.c | stm32f10x_it.c | app_cfg.h | app.c

```
/*
*********************************************************************************************************
*                                              KEY TASK
*********************************************************************************************************
*/
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被单击 */
    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    *)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 ( "\r\n最大中断时间是%dus\r\n",
             ts_int / ( cpu_clk_freq / 1000000 ) );

    OS_CRITICAL_EXIT();                                 //退出临界段

  }
}
```

图 15-18 AppTaskKey () 任务函数

把 STM32 的 USART1 连接至电脑的串口调试助手，编译和下载程序到秉火 STM32 开发板，运行程序。用户在串口调试助手的发送区发送数据（<=70 字节），接收区会立即返回该数据。按下 KEY1，串口会打印目前统计的最大关中断时间。

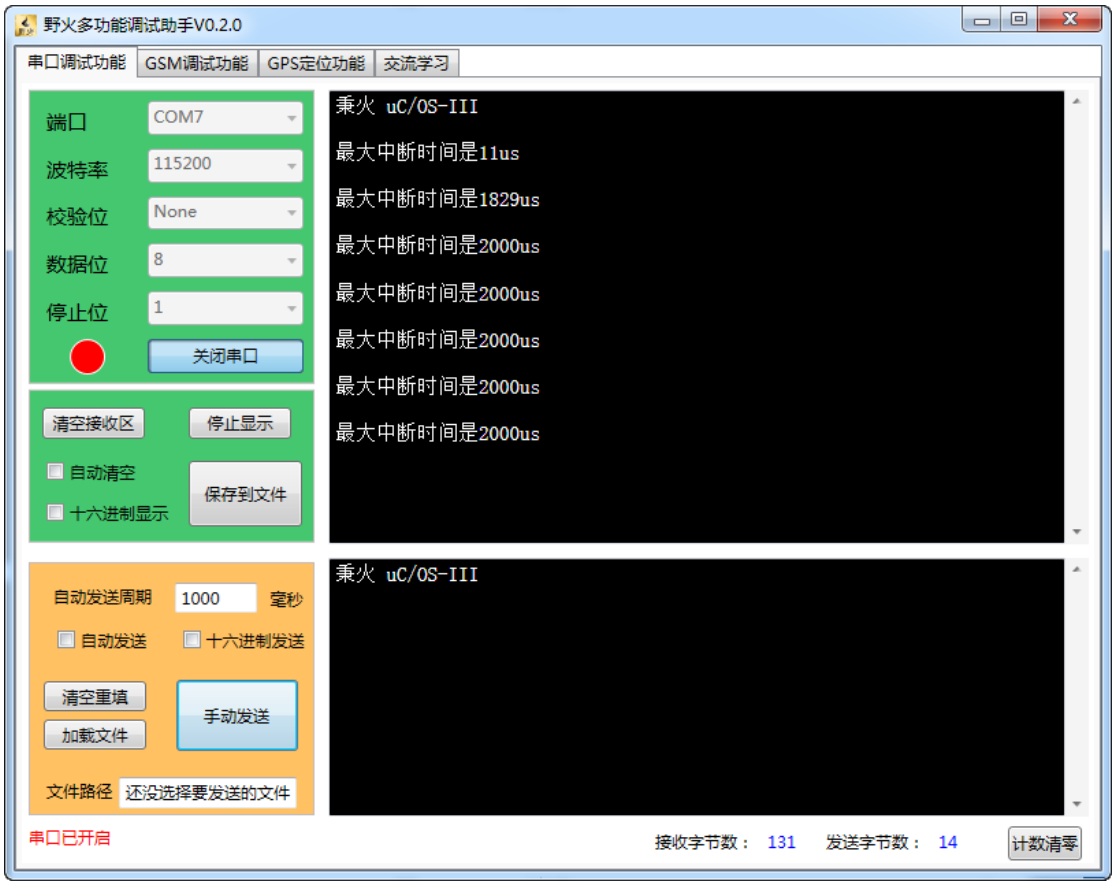

图 15-19 串口调试助手

上面调试是在没有使能中断延迟发布的情况进行的，下面使能中断延迟发布，即使能 OS_CFG_ISR_POST_DEFERRED_EN（位于“os_cfg.h”），如下图所示，重新调试程序。

```
33                                                  /* ------------------------------ MISCELLANEOUS ---
34  #define  OS_CFG_APP_HOOKS_EN                1u  //使能/禁用钩子函数
35  #define  OS_CFG_ARG_CHK_EN                  1u  //使能/禁用参数检测
36  #define  OS_CFG_CALLED_FROM_ISR_CHK_EN      1u  //使能/禁用检测中断中非法调用功能
37  #define  OS_CFG_DBG_EN                      1u  //使能/禁用调试代码和变量
38  #define  OS_CFG_ISR_POST_DEFERRED_EN        1u  //使能/禁用中断延迟发布
39  #define  OS_CFG_OBJ_TYPE_CHK_EN             1u  //使能/禁用对象类型检测
40  #define  OS_CFG_TS_EN                       1u  //使能/禁用时间戳
41
```

图 15-20 使能中断延迟发布

编译和下载程序到秉火 STM32 开发板，运行程序。操作步骤不变，可以发现最大关中断时间几乎为 0，大大的减小了。这就是因为，在使能中断延迟发布的情况下，很多中断级任务转换成了任务级任务，大大减小了关中断时间。减小了关中断时间有利于提高系统的实时性，所以建议用户开发时尽量使能中断延迟发布。

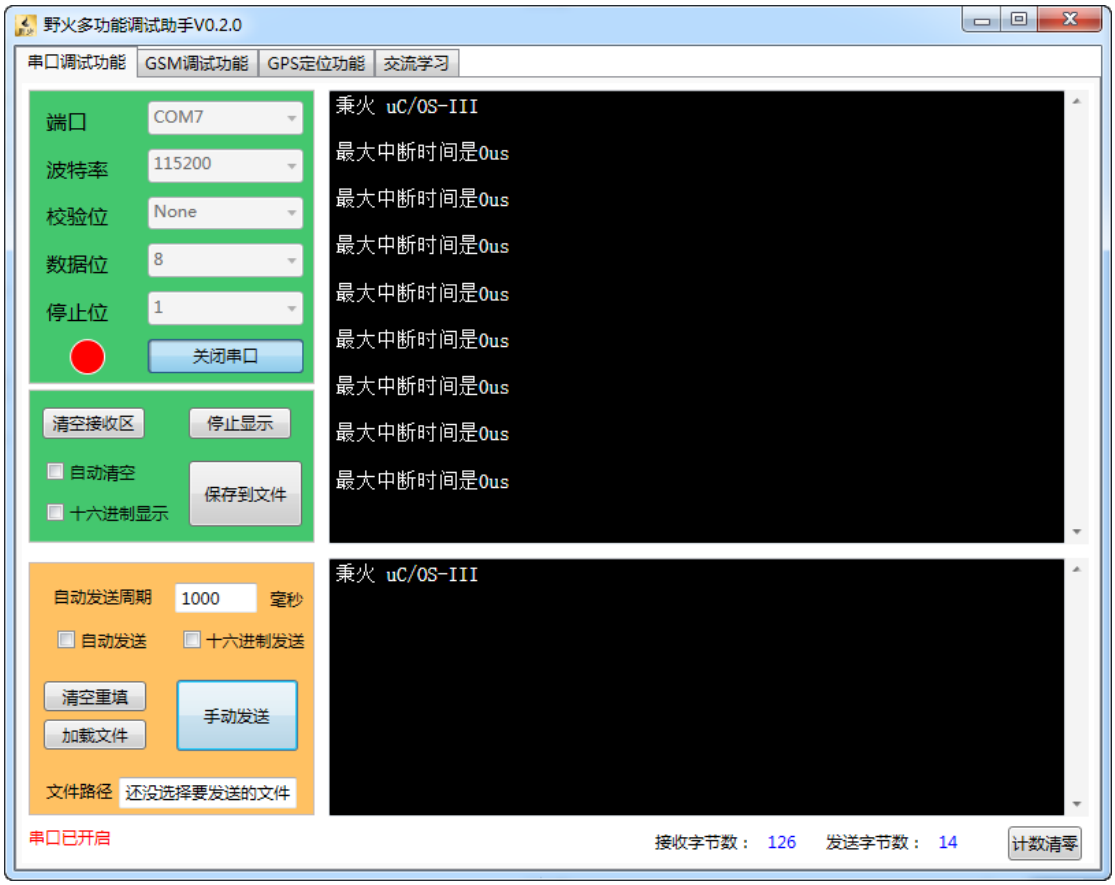

图 15-21 串口调试助手

15.3 章末总结

uC/OS 操作系统的中断管理主要分为禁用和使能中断延迟发布两种模式，“延迟”主要体现在时基中断的处理，在中断中发布内核对象，在中断中挂起任务或回复任务时的处理方式不同。在禁用中断延迟发布时，会立即执行这些操作。而在使能中断延迟发布时，会在退出中断后立即执行这些操作，把中断级任务转换成任务级任务，这可以大大减小系统的关中断时间，有利于提高系统的实时性。因此，建议用户在使用 uC/OS 操作系统时使能中断延迟发布。

出于 uC/OS 操作系统中断管理的要求，在进入中断服务函数时必须调用 OSIntEnter () 函数标记进入一个中断，在退出中断服务时也要相应调用 OSIntExit () 函数标记退出一个中断。

使用 CPU_IntDisMeasMaxGet () 函数可以获取整个程序目前的最大关中断时间。若要获得某段程序运行过程中的最大关中断时间，在该程序段前调用 CPU_IntDisMeasMaxCurReset () 函数为开始测量程序段的最大关中断进行初始化，在程序段结尾处调用 CPU_IntDisMeasMaxCurGet() 函数皆可以结束测量和返回程序段的最大关中断时间。

第16章 统计信息

uC/OS 系统提供了很多系统运行时的统计信息，方便用户了解系统的运行情况。

16.1 原理简述

16.1.1统计任务

uC/OS 系统为用户提供了一个统计任务，用于统计 CPU 使用率和任务堆栈的用量。用户在使用统计任务前，必须先使能它，统计任务的使能位于“os_cfg.h”。

```
82                                                  /* ---------------------------- TASK MANAGEMENT -----
83  #define  OS_CFG_STAT_TASK_EN                1u  //使能/禁用统计任务
84  #define  OS_CFG_STAT_TASK_STK_CHK_EN        1u  //使能/禁用在统计任务里检测任务堆栈
85
86  #define  OS_CFG_TASK_CHANGE_PRIO_EN         1u  //使能/禁用函数 OSTaskChangePrio()
87  #define  OS_CFG_TASK_DEL_EN                 1u  //使能/禁用函数 OSTaskDel()
88  #define  OS_CFG_TASK_Q_EN                   1u  //使能/禁用函数 OSTaskQXXXX()
89  #define  OS_CFG_TASK_Q_PEND_ABORT_EN        1u  //使能/禁用函数 OSTaskQPendAbort()
90  #define  OS_CFG_TASK_PROFILE_EN             1u  //使能/禁用任务控制块的简况变量
91  #define  OS_CFG_TASK_REG_TBL_SIZE           1u  //定义任务寄存器的数目
92  #define  OS_CFG_TASK_SEM_PEND_ABORT_EN      1u  //使能/禁用函数 OSTaskSemPendAbort()
93  #define  OS_CFG_TASK_SUSPEND_EN             1u  //使能/禁用函数 OSTaskSuspend() 和 OSTaskResume()
94
```

图 16-1 使能统计任务

如果要在统计任务中检测任务堆栈，还得额外使能 OS_CFG_STAT_TASK_STK_CHK_EN（位于“os_cfg.h”），如下图所示。对于需要进行任务堆栈检测的任务，在其被 OSTaskCreate()函数创建时，选项参数 opt 还需包含 OS_OPT_TASK_STK_CHK。

```
82                                                  /* ---------------------------- TASK MANAGEMENT -----
83  #define  OS_CFG_STAT_TASK_EN                1u  //使能/禁用统计任务
84  #define  OS_CFG_STAT_TASK_STK_CHK_EN        1u  //使能/禁用在统计任务里检测任务堆栈
85
86  #define  OS_CFG_TASK_CHANGE_PRIO_EN         1u  //使能/禁用函数 OSTaskChangePrio()
87  #define  OS_CFG_TASK_DEL_EN                 1u  //使能/禁用函数 OSTaskDel()
88  #define  OS_CFG_TASK_Q_EN                   1u  //使能/禁用函数 OSTaskQXXXX()
89  #define  OS_CFG_TASK_Q_PEND_ABORT_EN        1u  //使能/禁用函数 OSTaskQPendAbort()
90  #define  OS_CFG_TASK_PROFILE_EN             1u  //使能/禁用任务控制块的简况变量
91  #define  OS_CFG_TASK_REG_TBL_SIZE           1u  //定义任务寄存器的数目
92  #define  OS_CFG_TASK_SEM_PEND_ABORT_EN      1u  //使能/禁用函数 OSTaskSemPendAbort()
93  #define  OS_CFG_TASK_SUSPEND_EN             1u  //使能/禁用函数 OSTaskSuspend() 和 OSTaskResume()
94
```

图 16-2 使能检测堆栈

统计任务的任务函数为 OS_StatTask()，其定义位于“os_stat.c”。在任务中计算了整体 CPU 使用率，各个任务的 CPU 使用率和各个任务的堆栈用量。CPU 使用率和其最大记录分别保存于全局变量 OSStatTaskCPUUsage 和 OSStatTaskCPUUsageMax，一个任务的 CPU 使用率和其最大记录分别保存于其任务控制块的 CPUUsage 和 CPUUsageMax 成员，一个任务的任务堆栈的空闲大小和已用大小分别保存于其任务控制块的 StkFree 和 StkUsed 成员。需要注意，OSStatTaskCPUUsage、 OSStatTaskCPUUsageMax、CPUUsage 和 CPUUsageMax 均被放大了 10000 倍，所以这几个值缩小 10000 倍后才是真实值。

cpu_cfg.h | app_cfg.h | app.c | os_cfg.h | os_stat.c | os_cfg_app.c | os_cfg_app.h | os_task.c | os_type.h | os.h | os_core.c | os_time.c | cpu_core.c

```
281 void  OS_StatTask (void  *p_arg)                                  //统计任务函数
282 {
283 #if OS_CFG_DBG_EN > 0u
284 #if OS_CFG_TASK_PROFILE_EN > 0u
285     OS_CPU_USAGE usage;
286     OS_CYCLES    cycles_total;
287     OS_CYCLES    cycles_div;
288     OS_CYCLES    cycles_mult;
289     OS_CYCLES    cycles_max;
290 #endif
291     OS_TCB      *p_tcb;
292 #endif
293     OS_TICK      ctr_max;
294     OS_TICK      ctr_mult;
295     OS_TICK      ctr_div;
296     OS_ERR       err;
297     OS_TICK      dly;
298     CPU_TS       ts_start;
299     CPU_TS       ts_end;
300     CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必需该宏，该宏声明和
301                     //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
302                     // SR（临界段关中断只需保存SR），开中断时将该值还原。
303
304     p_arg = p_arg;                                                 //没意义，仅为预防编译器警告

305     while (OSStatTaskRdy != DEF_TRUE) {                            //如果统计任务没被允许运行
306         OSTimeDly(2u * OSCfg_StatTaskRate_Hz,                      //一直延时
307                   OS_OPT_TIME_DLY,
308                   &err);
309     }
310     OSStatReset(&err);                                             //如果统计任务已被就绪，复位统计，继续执行
311     /* 根据设置的宏计算统计任务的执行节拍数 */
312     dly = (OS_TICK)0;
313     if (OSCfg_TickRate_Hz > OSCfg_StatTaskRate_Hz) {
314         dly = (OS_TICK)(OSCfg_TickRate_Hz / OSCfg_StatTaskRate_Hz);
315     }
316     if (dly == (OS_TICK)0) {
317         dly =  (OS_TICK)(OSCfg_TickRate_Hz / (OS_RATE_HZ)10);
318     }
319

320     while (DEF_ON) {                                               //进入任务体
321         ts_start         = OS_TS_GET();                            //获取时间戳
322 #ifdef  CPU_CFG_INT_DIS_MEAS_EN                                    //如果要测量关中断时间
323         OSIntDisTimeMax = CPU_IntDisMeasMaxGet();                  //获取最大的关中断时间
324 #endif
325
326         CPU_CRITICAL_ENTER();                                      //关中断
327         OSStatTaskCtrRun    = OSStatTaskCtr;                       //获取上一次延时的空闲计数值
328         OSStatTaskCtr       = (OS_TICK)0;                          //进行下一次延时的空闲计数
329         CPU_CRITICAL_EXIT();                                       //开中断

330         /* 计算CPU使用率 */
331         if (OSStatTaskCtrMax > OSStatTaskCtrRun) {                 //如果空闲计数值小于最大空闲计数值
332             if (OSStatTaskCtrMax < 400000u) {                      //这些分类是为了避免计算CPU使用率过程中
333                 ctr_mult = 10000u;                                 //产生溢出，  就是避免相乘时超出32位寄存器。
334                 ctr_div  =     1u;
335             } else if (OSStatTaskCtrMax <   4000000u) {
336                 ctr_mult =  1000u;
337                 ctr_div  =    10u;
338             } else if (OSStatTaskCtrMax <  40000000u) {
339                 ctr_mult =   100u;
340                 ctr_div  =   100u;
341             } else if (OSStatTaskCtrMax < 400000000u) {
342                 ctr_mult =    10u;
343                 ctr_div  =  1000u;
344             } else {
345                 ctr_mult =     1u;
346                 ctr_div  = 10000u;
347             }
348             ctr_max            = OSStatTaskCtrMax / ctr_div;
349             OSStatTaskCPUUsage = (OS_CPU_USAGE)((OS_TICK)10000u - ctr_mult * OSStatTaskCtrRun / ctr_max);
350             if (OSStatTaskCPUUsageMax < OSStatTaskCPUUsage) { //更新CPU使用率的最大历史记录
351                 OSStatTaskCPUUsageMax = OSStatTaskCPUUsage;
352             }
353         } else {                                                   //如果空闲计数值大于或等于最大空闲计数值
354             OSStatTaskCPUUsage = (OS_CPU_USAGE)10000u;             //那么CPU使用率为0
355         }
356
357         OSStatTaskHook();                                          //用户自定义的钩子函数
358
```

```
359          /* 下面计算各个任务的CPU使用率，原理跟计算整体CPU使用率相似 */
360 #if OS_CFG_DBG_EN > 0u                                        //如果使能了调试代码和变量
361 #if OS_CFG_TASK_PROFILE_EN > 0u                               //如果使能了任务控制块的简况变量
362         cycles_total = (OS_CYCLES)0;
363
364         CPU_CRITICAL_ENTER();                                 //关中断
365         p_tcb = OSTaskDbgListPtr;                             //获取任务双向调试列表的首个任务
366         CPU_CRITICAL_EXIT();                                  //开中断
367         while (p_tcb != (OS_TCB *)0) {                        //如果该任务非空
368             OS_CRITICAL_ENTER();                              //进入临界段
369             p_tcb->CyclesTotalPrev =  p_tcb->CyclesTotal;     //保存任务的运行周期
370             p_tcb->CyclesTotal     = (OS_CYCLES)0;            //复位运行周期，为下次运行做准备
371             OS_CRITICAL_EXIT();                               //退出临界段
372
373             cycles_total          += p_tcb->CyclesTotalPrev;//所有任务运行周期的总和
374
375             CPU_CRITICAL_ENTER();                             //关中断
376             p_tcb                  = p_tcb->DbgNextPtr;       //获取列表的下一个任务，进行下一次循环
377             CPU_CRITICAL_EXIT();                              //开中断
378         }
379 #endif
380
381         /* 使用算法计算各个任务的CPU使用率和任务堆栈用量 */
382 #if OS_CFG_TASK_PROFILE_EN > 0u                               //如果使能了任务控制块的简况变量
383
384         if (cycles_total > (OS_CYCLES)0u) {                    //如果有任务占用过CPU
385             if (cycles_total < 400000u) {                     //这些分类是为了避免计算CPU使用率过程中
386                 cycles_mult = 10000u;                         //产生溢出，  就是避免相乘时超出32位寄存器。
387                 cycles_div  =     1u;
388             } else if (cycles_total <   4000000u) {
389                 cycles_mult =  1000u;
390                 cycles_div  =    10u;
391             } else if (cycles_total <  40000000u) {
392                 cycles_mult =   100u;
393                 cycles_div  =   100u;
394             } else if (cycles_total < 400000000u) {
395                 cycles_mult =    10u;
396                 cycles_div  =  1000u;
397             } else {
398                 cycles_mult =     1u;
399                 cycles_div  = 10000u;
400             }
401             cycles_max  = cycles_total / cycles_div;
402         } else {                                              //如果没有任务占用过CPU
403             cycles_mult = 0u;
404             cycles_max  = 1u;
405         }
406 #endif

407         CPU_CRITICAL_ENTER();                                 //关中断
408         p_tcb = OSTaskDbgListPtr;                             //获取任务双向调试列表的首个任务
409         CPU_CRITICAL_EXIT();                                  //开中断
410         while (p_tcb != (OS_TCB *)0) {                        //如果该任务非空
411 #if OS_CFG_TASK_PROFILE_EN > 0u                               //如果使能了任务控制块的简况变量
412             usage = (OS_CPU_USAGE)(cycles_mult * p_tcb->CyclesTotalPrev / cycles_max); //计算任务的CPU使用率
413             if (usage > 10000u) {                             //任务的CPU使用率为100%
414                 usage = 10000u;
415             }
416             p_tcb->CPUUsage = usage;                          //保存任务的CPU使用率
417             if (p_tcb->CPUUsageMax < usage) {                 //更新任务的最大CPU使用率的历史记录
418                 p_tcb->CPUUsageMax = usage;
419             }
420 #endif
421         /* 堆栈检测 */
422 #if OS_CFG_STAT_TASK_STK_CHK_EN > 0u                          //如果使能了任务堆栈检测
423             OSTaskStkChk( p_tcb,                              //计算被激活任务的堆栈用量
424                           &p_tcb->StkFree,
425                           &p_tcb->StkUsed,
426                           &err);
427 #endif
428
429             CPU_CRITICAL_ENTER();                             //关中断
430             p_tcb = p_tcb->DbgNextPtr;                        //获取列表的下一个任务，进行下一次循环
431             CPU_CRITICAL_EXIT();                              //开中断
432         }
433 #endif
```

```
434
435         if (OSStatResetFlag == DEF_TRUE) {                  //如果需要复位统计
436             OSStatResetFlag  = DEF_FALSE;
437             OSStatReset(&err);                              //复位统计
438         }
439
440         ts_end = OS_TS_GET() - ts_start;                    //计算统计任务的执行时间
441         if (OSStatTaskTimeMax < ts_end) {                   //更新统计任务的最大执行时间的历史记录
442             OSStatTaskTimeMax = ts_end;
443         }
444
445         OSTimeDly(dly,                                      //按照先前计算的执行节拍数延时
446                   OS_OPT_TIME_DLY,
447                   &err);
448     }
449 }
```

图 16-3 统计任务 OS_StatTask()

在统计任务 OS_StatTask()中，如果使能了检测堆栈，就调用 OSTaskStkChk () 函数计算任务的任务堆栈的空闲大小和已用大小，单位为 CPU_STK。OSTaskStkChk () 函数的定义位于“os_task.c”。

```
1629 #if OS_CFG_STAT_TASK_STK_CHK_EN > 0u                      //如果使能了任务堆栈检测
1630 void  OSTaskStkChk (OS_TCB        *p_tcb,                //目标任务控制块的指针
1631                     CPU_STK_SIZE  *p_free,               //返回空闲堆栈大小
1632                     CPU_STK_SIZE  *p_used,               //返回已用堆栈大小
1633                     OS_ERR        *p_err)                //返回错误类型
1634 {
1635     CPU_STK_SIZE  free_stk;
1636     CPU_STK      *p_stk;
1637     CPU_SR_ALLOC(); //使用到临界段（在关/开中断时）时必需该宏，该宏声明和
1638                     //定义一个局部变量，用于保存关中断前的 CPU 状态寄存器
1639                     // SR（临界段关中断只需保存SR），开中断时将该值还原。
1640
1641 #ifdef OS_SAFETY_CRITICAL                                 //如果使能了安全检测
1642     if (p_err == (OS_ERR *)0) {                           //如果 p_err 为空
1643         OS_SAFETY_CRITICAL_EXCEPTION();                   //执行安全检测异常函数
1644         return;                                           //返回，停止执行
1645     }
1646 #endif
1647
1648 #if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u                    //如果使能了中断中非法调用检测
1649     if (OSIntNestingCtr > (OS_NESTING_CTR)0) {            //如果该函数是在中断中被调用
1650        *p_err = OS_ERR_TASK_STK_CHK_ISR;                  //错误类型为“在中断中检测堆栈”
1651         return;                                           //返回，停止执行
1652     }
1653 #endif
1654
```

```
1655 #if OS_CFG_ARG_CHK_EN > 0u                          //如果使能了参数检测
1656     if (p_free == (CPU_STK_SIZE*)0) {               //如果 p_free 为空
1657         *p_err  = OS_ERR_PTR_INVALID;               //错误类型为“指针非法”
1658          return;                                    //返回，停止执行
1659     }
1660
1661     if (p_used == (CPU_STK_SIZE*)0) {               //如果 p_used 为空
1662         *p_err  = OS_ERR_PTR_INVALID;               //错误类型为“指针非法”
1663          return;                                    //返回，停止执行
1664     }
1665 #endif
1666
1667     CPU_CRITICAL_ENTER();                                   //关中断
1668     if (p_tcb == (OS_TCB *)0) {                             //如果 p_tcb 为空
1669         p_tcb = OSTCBCurPtr;                                //目标任务为当前运行任务（自身）
1670     }
1671
1672     if (p_tcb->StkPtr == (CPU_STK*)0) {                     //如果目标任务的堆栈为空
1673         CPU_CRITICAL_EXIT();                                //开中断
1674         *p_free = (CPU_STK_SIZE)0;                          //清零 p_free
1675         *p_used = (CPU_STK_SIZE)0;                          //清零 p_used
1676         *p_err  =  OS_ERR_TASK_NOT_EXIST;                   //错误类型为“任务不存在”
1677          return;                                            //返回，停止执行
1678     }
1679     /* 如果目标任务的堆栈非空 */
1680     if ((p_tcb->Opt & OS_OPT_TASK_STK_CHK) == (OS_OPT)0) {  //如果目标任务没选择检测堆栈
1681         CPU_CRITICAL_EXIT();                                //开中断
1682         *p_free = (CPU_STK_SIZE)0;                          //清零 p_free
1683         *p_used = (CPU_STK_SIZE)0;                          //清零 p_used
1684         *p_err  =  OS_ERR_TASK_OPT;                         //错误类型为“任务选项有误”
1685          return;                                            //返回，停止执行
1686     }
1687     CPU_CRITICAL_EXIT();                                    //如果任务选择了检测堆栈，开中断
1688     /* 开始计算目标任务的堆栈的空闲数目和已用数目 */
1689     free_stk  = 0u;                                         //初始化计算堆栈工作
1690 #if CPU_CFG_STK_GROWTH == CPU_STK_GROWTH_HI_TO_LO           //如果CPU的堆栈是从高向低增长
1691     p_stk = p_tcb->StkBasePtr;                              //从目标任务堆栈最低地址开始计算
1692     while (*p_stk == (CPU_STK)0) {                          //计算值为0的堆栈数目
1693         p_stk++;
1694         free_stk++;
1695     }
1696 #else                                                       //如果CPU的堆栈是从低向高增长
1697     p_stk = p_tcb->StkBasePtr + p_tcb->StkSize - 1u;        //从目标任务堆栈最高地址开始计算
1698     while (*p_stk == (CPU_STK)0) {                          //计算值为0的堆栈数目
1699         free_stk++;
1700         p_stk--;
1701     }
1702 #endif
1703     *p_free = free_stk;                                     //返回目标任务堆栈的空闲数目
1704     *p_used = (p_tcb->StkSize - free_stk);                  //返回目标任务堆栈的已用数目
1705     *p_err  = OS_ERR_NONE;                                  //错误类型为“无错误”
1706 }
1707 #endif
```

图 16-4 OSTaskStkChk () 函数

16.1.2 CPU 主频

使用 BSP_CPU_ClkFreq() 函数可以返回 CPU 的工作频率（单位：Hz），时间戳就是以这个频率计数的，用户在将时间戳转换成国际时间单位时，需要除以这个频率，在前面的诸多例程中，已经演示过这些功能的使用。BSP_CPU_ClkFreq() 函数的信息如下表所示。

表 60 BSP_CPU_ClkFreq ()

函数	CPU_INT32U BSP_CPU_ClkFreq (void);

原型	
功能	获取 CPU 主频。
参数	无。
返回值	CPU 的主频（单位：HZ），即 HCLK 时钟频率。

BSP_CPU_ClkFreq() 函数的定义位于“bsp.c”。

```
188 CPU_INT32U  BSP_CPU_ClkFreq (void)                    //获取CPU主频
189 {
190     RCC_ClocksTypeDef  rcc_clocks;
191
192
193     RCC_GetClocksFreq(&rcc_clocks);                   //获取芯片的各种时钟频率
194
195     return ((CPU_INT32U)rcc_clocks.HCLK_Frequency); //返回 HCLK 时钟频率
196 }
```

图 16-5 BSP_CPU_ClkFreq() 函数

16.1.3 uC/OS 版本号

使用 OSVersion () 函数可以获取 uC/OS 系统的版本号，版本号格式为“Vx.yy.zz”，返回的版本号是一个被去掉“.”符号的整数，例如版本号为“V3.01.02”就返回 30102。OSVersion () 函数的信息如下表所示。

表 61 OSVersion ()

函数原型	CPU_INT16U OSVersion (OS_ERR *p_err);			
功能	获取 CPU 主频。			
参数	p_err	返回错误类型	OS_ERR_NONE	无错误。
返回值	✧ 0，有错误，获取 uC/OS 系统版本号失败。 ✧ 其他值，uC/OS 系统版本号。			
注意事项	✧ 返回的版本号是一个被去掉“.”符号的整数，用户需要自行处理获取真实的版本号。			

OSVersion () 函数的定义位于“os_core.c”。

```
745 CPU_INT16U  OSVersion (OS_ERR  *p_err)      //获取uC/OS系统的版本号
746 {
747 #ifdef OS_SAFETY_CRITICAL                   //如果使能（默认禁用）了安全检测
748     if (p_err == (OS_ERR *)0) {              //如果 p_err 为空
749         OS_SAFETY_CRITICAL_EXCEPTION();      //执行安全检测异常函数
750         return ((CPU_INT16U)0u);             //返回0（有错误），停止执行
751     }
752 #endif
753    /* 如果 p_err 非空 */
754    *p_err = OS_ERR_NONE;                     //错误类型为“无错误”
755     return (OS_VERSION);                     //返回版本号
756 }
```

图 16-6 OSVersion () 函数

其中 OS_VERSION 其实就是一个宏常量，是一个定义版本号的宏，其定义位于“os.h”。

```
39 /*
40 *********************************************************************************************************
41 *                                          uC/OS-III VERSION NUMBER
42 *********************************************************************************************************
43 */
44
45 #define  OS_VERSION  30301u                  /* Version of uC/OS-III (Vx.yy.zz mult. by 10000)          */
46
```

图 16-7 OS_VERSION 的定义

16.1.4 其他统计信息

除了前面讲述的统计信息外，uC/OS 系统还为我们提供了很多统计信息。前一章讲述的最大关中断时间就是个很重要参数。此外，全局变量 OSTaskCtxSwCtr 记录了任务切换总次数，OSTaskQty 记录了被创建任务的数目，OSSchedLockTimeMax（需使能 OS_CFG_SCHED_LOCK_TIME_MEAS_EN，位于“os_cfg.h”）记录了调度器被锁的最大时间，OSIntQNbrEntriesMax 记录了中断队列成员被使用的最大数目，OSFlagQty 记录了事件标志组对象的数目，OSMemQty 记录了内存管理（分区）对象的数目，OSMutexQty 记录了互斥信号量对象的数目，OSQQty 记录了消息队列对象的数目，OSSemQty 记录了多值信号量对象的数目，等等。

16.2 实例演示

16.2.1 实例 1

本实例沿用上一章“中断管理”的实例 1，除打印最大关中断时间外，还打印本章讲述的诸多统计信息。另外用户可以对比，在没有使用串口回显和使用串口回显后串口任务的 CPU 最大使用率的区别。

该例程已经存放在配套资料的下图路径。

图 16-8 例程路径

本例程需用使用 USART 和 EXIT，所以工程中需要添加其驱动文件和初始化。用户可参照前面相关实例，这里不再赘述。

在“stm32f4xx_it.c”文件定义了串口接收中断和 KEY1 中断。串口接收中断服务函数的定义如下。中断发生后，该中断服务函数把接收数据通过任务消息发送给任务 AppTaskUsart()打印，消息存放在内存分区 mem。

```
app.c  stm32f4xx_it.c
174 /**
175   * @brief  USART 中断服务函数
176   * @param  无
177   * @retval 无
178   */
179 void macUSART_INT_FUN(void)
180 {
181   OS_ERR    err;
182
183   char *    p_mem_blk;
184
185
186   OSIntEnter();                                          //进入中断
187
188   /* 从内存分区 mem 获取一个内存块 */
189   p_mem_blk = OSMemGet((OS_MEM      *)&mem,
190                        (OS_ERR      *)&err);
191
192   if ( USART_GetITStatus ( macUSARTx, USART_IT_RXNE ) != RESET )
193   {
194     * p_mem_blk = USART_ReceiveData ( macUSARTx );       //获取接收到的数据
195
196     /* 发布任务消息到任务 AppTaskUsart */
197     OSTaskQPost ((OS_TCB      *)&AppTaskUsartTCB,        //目标任务的控制块
198                  (void        *)p_mem_blk,               //消息内容的首地址
199                  (OS_MSG_SIZE  )1,                       //消息长度
200                  (OS_OPT       )OS_OPT_POST_FIFO,        //发布到任务消息队列的入口端
201                  (OS_ERR      *)&err);                   //返回错误类型
202
203   }
204
205   OSIntExit();                                           //退出中断
206
207 }
208
```

图 16-9 USART1 接收中断服务函数

KEY1 中断服务函数的定义如下。中断发生后，该中断服务函数通过任务信号量通知任务 AppTaskKey()，任务 AppTaskKey() 就打印统计信息。

```c
/**
  * @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_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

}
```

图 16-10 KEY1 中断服务函数

串口任务 AppTaskUsart () 的定义如下。该任务接收到 USART1 接收中断发送过来的任务消息后，就打印任务消息的内容，实现串口回显。

```c
/*
*********************************************************************************************************
*                                            USART TASK
*********************************************************************************************************
*/
static  void  AppTaskUsart ( void * p_arg )
{
  OS_ERR         err;
  OS_MSG_SIZE    msg_size;
  CPU_SR_ALLOC();

  char * pMsg;


  (void)p_arg;


  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        *)0,                             //返回消息被发布的时间戳
                        (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 ( "%c", * pMsg );                                            //打印消息内容

    OS_CRITICAL_EXIT();                                                 //退出临界段

    /* 退还内存块 */
    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

  }

}
```

图 16-11 AppTaskUsart () 任务函数

任务 AppTaskKey() 的定义如下。该任务接收到任务信号量后，就获取和打印统计信息。

cpu_cfg.h | app_cfg.h | app.c | os_cfg.h | os_stat.c | os_cfg_app.c | os_cfg_app.h | os_task.c | os_type.h | os.h | os_core.c | os_time.c | cpu_core.c | bsp.c | stm32f10x_rcc.c

```
/*
*********************************************************************************************************
*                                              KEY TASK
*********************************************************************************************************
*/
static  void  AppTaskKey ( void * p_arg )
{
  OS_ERR          err;
  CPU_TS_TMR      ts_int;
  CPU_INT16U      version;
  CPU_INT32U      cpu_clk_freq;
  CPU_SR_ALLOC();


  (void)p_arg;

  version = OSVersion(&err);                          //获取uC/OS版本号

  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被单击 */
    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     *)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 ( "\r\nuC/OS版本号：V%d.%02d.%02d\r\n",
             version / 10000, version % 10000 / 100, version % 100 );

    printf ( "CPU主频：%d MHz\r\n", cpu_clk_freq / 1000000 );

    printf ( "最大中断时间：%d us\r\n",
            ts_int / ( cpu_clk_freq / 1000000 ) );

    printf ( "最大锁调度器时间：%d us\r\n",
             OSSchedLockTimeMax / ( cpu_clk_freq / 1000000 ) );

    printf ( "任务切换总次数：%d\r\n", OSTaskCtxSwCtr );

```

```
308         printf ( "CPU使用率：%d.%d%%\r\n",
309                  OSStatTaskCPUUsage / 100, OSStatTaskCPUUsage % 100 );
310
311         printf ( "CPU最大使用率：%d.%d%%\r\n",
312                  OSStatTaskCPUUsageMax / 100, OSStatTaskCPUUsageMax % 100 );
313
314         printf ( "串口任务的CPU使用率：%d.%d%%\r\n",
315                  AppTaskUsartTCB.CPUUsage / 100, AppTaskUsartTCB.CPUUsage % 100 );
316
317         printf ( "串口任务的CPU最大使用率：%d.%d%%\r\n",
318                  AppTaskUsartTCB.CPUUsageMax / 100, AppTaskUsartTCB.CPUUsageMax % 100 );
319
320         printf ( "按键任务的CPU使用率：%d.%d%%\r\n",
321                  AppTaskKeyTCB.CPUUsage / 100, AppTaskKeyTCB.CPUUsage % 100 );
322
323         printf ( "按键任务的CPU最大使用率：%d.%d%%     \r\n",
324                  AppTaskKeyTCB.CPUUsageMax / 100, AppTaskKeyTCB.CPUUsageMax % 100 );
325
326         printf ( "串口任务的已用和空闲堆栈大小分别为：%d,%d\r\n",
327                  AppTaskUsartTCB.StkUsed, AppTaskUsartTCB.StkFree );
328
329         printf ( "按键任务的已用和空闲堆栈大小分别为：%d,%d\r\n",
330                  AppTaskKeyTCB.StkUsed, AppTaskKeyTCB.StkFree );
331
332         OS_CRITICAL_EXIT();                                 //退出临界段
333
334     }
335
336 }
```

图 16-12 AppTaskKey () 任务函数

把 STM32 的 USART1 连接至电脑的串口调试助手，编译和下载程序到秉火 STM32 开发板，运行程序。用户先不要使用串口调试助手向 STM32 发送数据，按下 KEY1，打印统计信息。

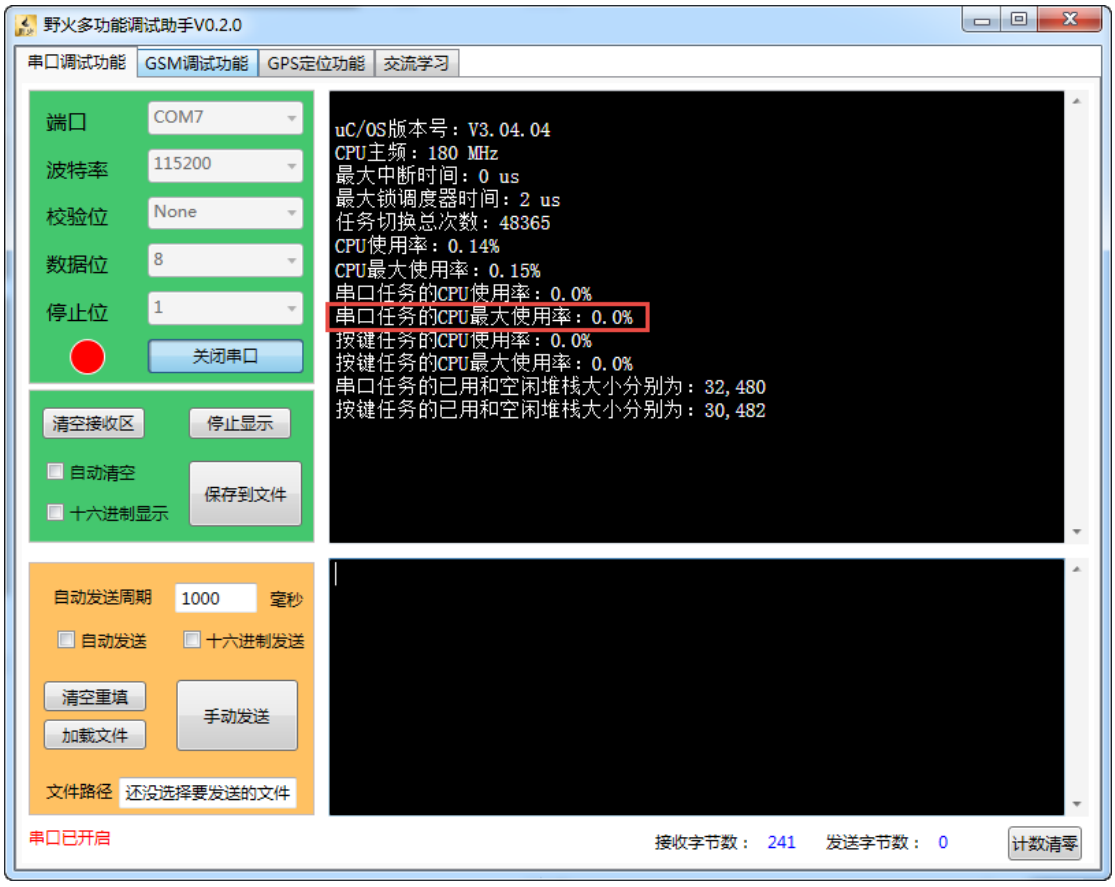

图 16-13 串口调试助手

其中可以看到，串口任务的 CPU 最大使用率为 0.0%，也就是说串口任务没运行过，这是符合实际情况的，因为当前还没使用过串口回显功能，串口任务根本没工作过。使用串口调试助手向 STM32 发送数据，再打印统计信息，如下，可以发现，串口任务的 CPU 最大使用率变成 0.59%，这也是符合实际情况，因为刚刚向 STM32 发送数据时就触发了串口任务的运行。

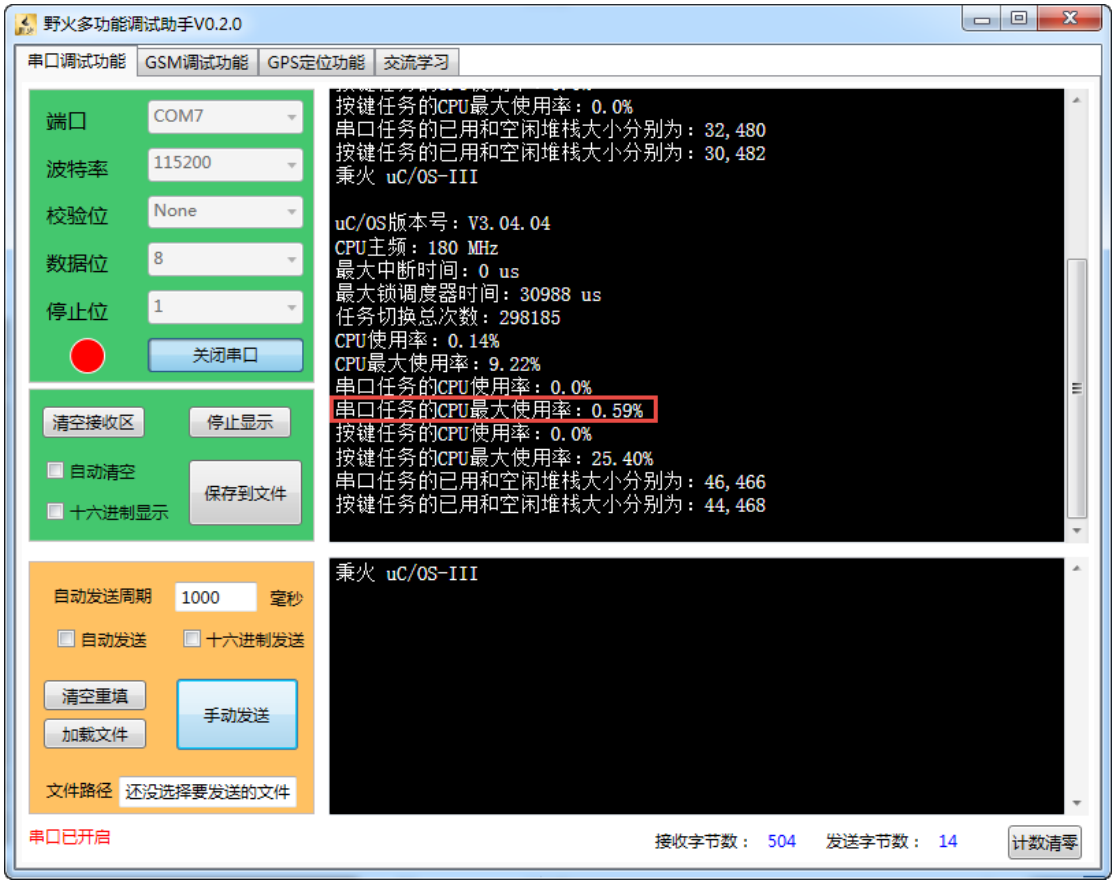

图 16-14 串口调试助手

16.3 章末总结

uC/OS 操作系统为用户提供了很多统计信息，方便用户了解系统的运行情况。其中系统内部的统计任务为用户提供了所有应用任务的 CPU 使用率 OSStatTaskCPUUsage 和最大使用率 OSStatTaskCPUUsageMax，每个任务的 CPU 使用率 CPUUsage 和最大使用率 CPUUsageMax，每个任务的任务堆栈的空闲大小 StkFree 和已用大小 StkUsed。某个任务的统计信息一般存放于其任务控制块的相应成员里，整体的统计信息一般存放于全局变量。

通过 BSP_CPU_ClkFreq() 函数可以获取 CPU 的工作主频，时间戳就是以该频率作为时钟的一个计数值。

通过 OSVersion () 函数可以获取 uC/OS 系统的版本号，获取到版本号是个去掉“.”符号的整型值，还需做相应处理将其转换为标准的版本号。

此外，通过 CPU_IntDisMeasMaxGet ()函数可以获取最大关中断时间。全局变量 OSSchedLockTimeMax 记录了调度器被锁的最大时间，全局变量 OSTaskCtxSwCtr 记录了任务切换总次数，OSTaskQty 记录了被创建任务的数目， OSIntQNbrEntriesMax 记录了中断队列成员被使用的最大数目，OSFlagQty 记录了事件标志组对象的数目，OSMemQty 记录了内存管理（分区）对象的数目，OSMutexQty 记录了互斥信号量对象的数目，OSQQty 记录了消息队列对象的数目，OSSemQty 记录了多值信号量对象的数目，等等。