

Ucos_II2.52 是一份非常完美的嵌入式开发系统，在学习 ARM 的基础上，嵌入 ucos 系统并增加自己的源码是一件不错的选择，目前在市面上已经有了大量的 ucos 嵌入案例，特别是在 arm 和 dsp 的应用当中，已经成为一种主流，虽然和其它的嵌入式系统相比，ucos 不是很完善，如没有内存分配、任务级别不多；但却是一个代码简短、条理清晰、实时性及安全性能很高的嵌入式操作系统。

Ucos_II2.52 对比 2.8 版的 256 个任务而言，任务数量相比过少，但却是目前应用量最大的一个版本，相对而言，能够满足我们的基本要求，而且增加了很多消息处理，特别是在优先级别方面，具有不可比拟的优势；我曾试图阅读 ecos 的源码，但还是失败了，还有挑战 linux0.01 版源码的想法，最终我不能不被屈服；对于 Ucos 而言，很多入门者是一个福音，因为它的代码非常的少，而且能够对应贝贝老师的书本直接参考，他的书本对结构方面讲解的极为 xian 详细。

在学习 Ucos 的整个过程中，E 文的理解是一个致命的打击，原因是我的 E 文水平很差，不过 Ucos 还是给了我尝试的动力，在作者的原基础上增加中文译码，也许是一件非常不错的选择，相信在中国和我这种水平的人多不胜数，中文的注解对源码而言，能够具有极高的理解价值，可以在极短的时间内，能够充分了解 ucos 的真正含义。

整个翻译过程历时 4 个月，每每在寒冬腊月坐在计算机前面，不断的查阅贝贝老师的书来对整个 Ucos 进行理解，对每个源码进行逐条翻译，也是一件非常需要勇气的事情，但 E 文的翻译过程中很多变量是不能完全理解的，所以在翻译过程中不乏错误译文很多，于此带来的错误还请读者纠正，相信克服种种困难一定会有所了解的。

对于经济窘迫的我来说，曾试图希望卖一点资料来养家糊口，但这种做法根本不现实，很多的读者可能和我一样，习惯了拿不收费的资料，并对变相收费有一种深恶痛绝的感觉；想了很多决定还是把它贡献出来，让更多的人来（更容易）了解 ucos，贡献自己的一点力量。

希望更多的人能加入这种高尚的学习氛围当中来，共同的来把一套完整的 U 系列源码译文早一日与我们分享，祝愿大家能够早日实现自己的梦想。

```
1 /*
2 ****
3 *                                     uC/OS-II实时控制内核
4 *                                     主要的包含文件
5 *
6 *
7 * 文   件: uCOS_II.C  包含主要uC/OS-II构造文件
8 * 作   者: Jean J. Labrosse
9 * 中文注解: 钟常慰  zhongcw @ 126.com  译注版本: 1.0 请尊重原版内容
10 ****
11 */
12
13 #define OS_GLOBALS                //定义全程变量 OS_GLOBALS
14 #include "includes.h"             //包含"includes.h"文件, 部分C语言头文件的汇总打包文件
15
16 /*****包含主要uC/OS-II文件*****/
17 //主要设定的地址, 不能有错, 可设定相对和绝对地址, 否则编译连接出错
18
19 #define OS_MASTER_FILE            //定义主要文件, 防止是从includes.h中跟随文件
20 #include "\software\ucos-ii\source\os_core.c" //包含内核结构管理文件
21 #include "\software\ucos-ii\source\os_flag.c" //包含时间标志组管理文件
22 #include "\software\ucos-ii\source\os_mbox.c" //包含消息邮箱管理文件
23 #include "\software\ucos-ii\source\os_mem.c"  //包含内存管理文件
24 #include "\software\ucos-ii\source\os_mutex.c" //包含互斥型信号管理文件
25 #include "\software\ucos-ii\source\os_q.c"    //包含消息队列管理文件
26 #include "\software\ucos-ii\source\os_sem.c"  //包含信号量管理文件
27 #include "\software\ucos-ii\source\os_task.c" //包含任务管理文件
28 #include "\software\ucos-ii\source\os_time.c" //包含时间管理文件
29
30 /*****结束*****/
31
```

```

1  /*
2  ****
3  *                                     uC/OS-II实时控制内核
4  *                                     主要的包含文件
5  * 文   件: uCOS_II.H           ucos内部函数参数设定
6  * 作   者: Jean J. Labrosse
7  * 中文注解: 钟常慰 zhongcw @ 126.com 译注版本: 1.0 请尊重原版内容
8  ****
9  */
10
11 /*
12 ****
13 *                                     混杂的设定
14 ****
15 */
16
17 #define OS_VERSION                251           // 定义uC/OS-II版本号
18
19 #ifdef OS_GLOBALS                 //如果 OS_GLOBALS 已被声明定义, 紧随代码将会被编译
20 #define OS_EXT                    //则定义 OS_EXT
21 #else
22 #define OS_EXT extern             //否则, 定义 OS_EXT 为 extern
23 #endif
24
25 #ifndef FALSE                     //是否未定义 FALSE
26 #define FALSE                     0             //如果是则定义 FALSE 为 0
27 #endif
28
29 #ifndef TRUE                      //是否未定义 TRUE
30 #define TRUE                      1             //如果是则定义 TRUE 为 1
31 #endif
32
33 #define OS_PRIO_SELF              0xFF          //定义 OS_PRIO_SELF 为 0xFF
34 #if OS_TASK_STAT_EN > 0
35 #define OS_N_SYS_TASKS           2             //任务体系号码
36 #else
37 #define OS_N_SYS_TASKS           1
38 #endif
39
40 #define OS_STAT_PRIO              (OS_LOWEST_PRIO - 1) //统计任务优先级
41 #define OS_IDLE_PRIO             (OS_LOWEST_PRIO)    //空闲任务优先级
42
43 #define OS_EVENT_TBL_SIZE ((OS_LOWEST_PRIO) / 8 + 1) //事件列表字节
44 #define OS_RDY_TBL_SIZE   ((OS_LOWEST_PRIO) / 8 + 1) //就绪列表字节
45
46 #define OS_TASK_IDLE_ID          65535          /* I.D. numbers for Idle and Stat tasks */
47 #define OS_TASK_STAT_ID          65534
48
49 #define OS_EVENT_EN               (((OS_Q_EN > 0) && (OS_MAX_QS > 0)) || (OS_MBOX_EN > 0) || (OS_SEM_EN > 0) || (OS_MUTEX_EN > 0))
50
51 /*$PAGE*/
52 /*
53 ****
54 *                                     任务状态字 TASK STATUS (字节定义在 OSTCBStat中)
55 ****
56 */
57 #define OS_STAT_RDY               0x00          // (将任务的状态字)处于完毕状态
58 #define OS_STAT_SEM               0x01          // (将任务的状态字)处于SEM状态
59 #define OS_STAT_MBOX              0x02          // (将任务的状态字)处于MBOX状态
60 #define OS_STAT_Q                 0x04          // (将任务的状态字)处于Q状态
61 #define OS_STAT_SUSPEND           0x08          // 表示任务被挂起
62 #define OS_STAT_MUTEX             0x10          // (将任务的状态字)处于MUTEX状态
63 #define OS_STAT_FLAG              0x20          // (将任务的状态字)处于FLAG状态
64
65 /*
66 ****
67 *                                     事件类型(OS_EVENT types)
68 ****
69 */
70 #define OS_EVENT_TYPE_UNUSED      0             // 定义事件类型的种类(无事件类型-----0)
71 #define OS_EVENT_TYPE_MBOX        1             // 定义事件类型的种类(邮箱为数字序列-----1)
72 #define OS_EVENT_TYPE_Q           2             // 定义事件类型的种类(消息队列为数字序列---2)
73 #define OS_EVENT_TYPE_SEM         3             // 定义事件类型的种类(信号量为数字序列-----3)
74 #define OS_EVENT_TYPE_MUTEX       4             // 定义事件类型的种类(互斥型信号量为数字序列4)
75 #define OS_EVENT_TYPE_FLAG        5             // 定义事件类型的种类(事件标志组为数字序列—5)

```

```

76
77 /*
78 ****
79 *                               事件标志 (EVENT FLAGS)
80 ****
81 */
82 #define OS_FLAG_WAIT_CLR_ALL    0        // 定义所有指定事件标志位清0 ----- 0
83 #define OS_FLAG_WAIT_CLR_AND    0        // 同上一样
84
85 #define OS_FLAG_WAIT_CLR_ANY    1        // 定义任意指定事件标志位清0 ----- 1
86 #define OS_FLAG_WAIT_CLR_OR     1        // 同上一样
87
88 #define OS_FLAG_WAIT_SET_ALL    2        // 定义所有指定事件标志位置1 ----- 2
89 #define OS_FLAG_WAIT_SET_AND    2        // 同上一样
90
91 #define OS_FLAG_WAIT_SET_ANY    3        // 定义任意指定事件标志位置1 ----- 3
92 #define OS_FLAG_WAIT_SET_OR     3        // 同上一样
93
94 // 如果需要在得到期望标志后, 恢复该事件标志, 加入此常量
95
96 #define OS_FLAG_CONSUME         0x80     // 定义常量OS_FLAG_CONSUME为0x80
97
98 #define OS_FLAG_CLR             0        // 定义 OS_FLAG_CLR 为清0
99 #define OS_FLAG_SET             1        // 定义 OS_FLAG_SET 为置1
100
101 /*
102 ****
103 *       设置字在'opt'中, 适用于 OS_SemDel(), OS_MboxDel(), OS_QDel() 和 OS_MutexDel() 函数
104 ****
105 */
106 #define OS_DEL_NO_PEND          0        // 可以选择只能在已经没有任何任务在等待该信号量时, 才能删除该信号量
107 #define OS_DEL_ALWAYS          1        // 不管有没有任务在等待该信号量, 立即删除该信号量
108
109 /*
110 ****
111 *                               OS???PostOpt() OPTIONS (设置)
112 *
113 * 这个设置适用用 OS_MboxPostOpt() 和 OS_QPostOpt() 两个函数.
114 ****
115 */
116 #define OS_POST_OPT_NONE        0x00     // 发送一个消息(或邮箱)给一个等待消息的任务
117 #define OS_POST_OPT_BROADCAST   0x01     // 发送消息给所有等待队列消息的任务*/
118 #define OS_POST_OPT_FRONT       0x02     // 以后进先出方式发消息(仿真OSQPostFront())
119
120 /*
121 ****
122 *                               任务设置 TASK OPTIONS (查看OSTaskCreateExt())
123 ****
124 */
125 #define OS_TASK_OPT_STK_CHK     0x0001   // 决定是否进行任务堆栈检查
126 #define OS_TASK_OPT_STK_CLR     0x0002   // 决定是否清空堆栈
127 #define OS_TASK_OPT_SAVE_FP     0x0004   // 决定是否保存浮点寄存器的数值。此项操作仅当处理器有浮点硬-
128                                           // 件时有效。保存操作由硬件相关的代码完成
129 /*
130 ****
131 *                               错误代码 ERROR CODES
132 ****
133 */
134 #define OS_NO_ERR               0        // 函数返回成功;
135
136 #define OS_ERR_EVENT_TYPE       1        // 不是指向事件(相关)类型的指针;
137 #define OS_ERR_PEND_ISR         2        // 在中断服务子程序中调用 OS各种信号类Accept() 函数.
138 #define OS_ERR_POST_NULL_PTR    3        // 用户发出空指针。根据规则, 这里不支持空指针;
139 #define OS_ERR_EVENT_NULL       4        // 'pevent' 是指空指针;
140 #define OS_ERR_POST_ISR         5        // 试图在中断服务子程序中调用OSMutexPost() 函数[释放一个mutex];
141 #define OS_ERR_QUERY_ISR        6        // 试图在中断子程序中调用OSMutexQuery() [得到mutex当前状态信息]
142 #define OS_ERR_INVALID_OPT      7        // 定义的opt参数无效;
143 #define OS_ERR_TASK_WAITING     8        // 有一个或一个以上的任务在等待消息队列中的消息;
144
145 #define OS_TIMEOUT              10       // 消息没有在指定的周期数内送到;
146 #define OS_TASK_NOT_EXIST       11       // 指定的任务不存;
147
148 #define OS_MBOX_FULL            20       // 消息邮箱已经包含了其他消息, 不空;
149
150 #define OS_Q_FULL               30       // 消息队列中已经存满;

```

```

151
152 #define OS_Prio_EXIST 40 // 优先级为PIP的任务已经存在;
153 #define OS_Prio_ERR 41 // 参数中的任务原先优先级不存在;
154 #define OS_Prio_INVALID 42 // 参数指定的优先级大于OS_LOWEST_Prio;
155
156 #define OS_SEM_OVF 50 // 信号量的值溢出;
157
158 #define OS_Task_DEL_ERR 60 // 指定要删除的任务不存在
159 #define OS_Task_DEL_IDLE 61 // 错误操作, 试图删除空闲任务(Idle task);
160 #define OS_Task_DEL_REQ 62 // 当前任务收到来自其他任务的删除请求;
161 #define OS_Task_DEL_ISR 63 // 错误操作, 试图在中断处理程序中删除任务;
162
163 #define OS_NO_MORE_TCB 70 // 系统中没有OS_TCB可以分配给任务了;
164
165 #define OS_TIME_NOT_DLY 80 // 要唤醒的任务不在延时状态;
166 #define OS_TIME_INVALID_MINUTES 81 // 参数错误, 分钟数大于59;
167 #define OS_TIME_INVALID_SECONDS 82 // 参数错误, 秒数大于59
168 #define OS_TIME_INVALID_MILLI 83 // 则返回参数错误, 毫秒数大于999;
169 #define OS_TIME_ZERO_DLY 84 // 四个参数全为0
170
171 #define OS_Task_SUSPEND_Prio 90 // 要挂起的任务不存在
172 #define OS_Task_SUSPEND_IDLE 91 // 试图挂起uC/OS-II中的空闲任务(Idle task)
173
174 #define OS_Task_RESUME_Prio 100 // 要唤醒的任务不存在;
175 #define OS_Task_NOT_SUSPENDED 101 // 要唤醒的任务不在挂起状态
176
177 #define OS_MEM_INVALID_PART 110 // 没有空闲的内存区;
178 #define OS_MEM_INVALID_BLKs 111 // 没有为每一个内存区建立至少2个内存块;
179 #define OS_MEM_INVALID_SIZE 112 // 内存块大小不足以容纳一个指针变量;
180 #define OS_MEM_NO_FREE_BLKs 113 // 内存区已经没有空间分配给内存块;
181 #define OS_MEM_FULL 114 // 内存区已经不能再接受更多释放的内存块。这种情况说明用户程序出
    现;
182 #define OS_MEM_INVALID_PBLK 115 //
183 #define OS_MEM_INVALID_PMem 116 // 'pmem' 是空指针;
184 #define OS_MEM_INVALID_PData 117 // pdata是空指针;
185 #define OS_MEM_INVALID_ADDR 118 // 非法地址, 即地址为空指针;
186
187 #define OS_ERR_NOT_MUTEX_OWNER 120 // 发出mutex的任务实际上并不占用mutex;
188
189 #define OS_Task_OPT_ERR 130 // 任务用OSTaskCreateExt()函数建立的时候没有指定
    OS_Task_OPT_STK_CHK-
190 // 一操作, 或者任务是用OSTaskCreate()函数建立的。
191
192 #define OS_ERR_DEL_ISR 140 // 试图在中断服务子程序中删除(消息、邮箱、信号量、消息对列、互
    斥型信号量)
193 #define OS_ERR_CREATE_ISR 141 // 试图在中断服务子程序中建立(事件标志组、互斥型信号量);
194
195 #define OS_FLAG_INVALID_PGRP 150 // pgrp是一个空指针;
196 #define OS_FLAG_ERR_WAIT_Type 151 // 'wait_type' 不是指定的参数之一;
197 #define OS_FLAG_ERR_NOT_RDY 152 // 指定的事件标志没有发生;
198 #define OS_FLAG_INVALID_OPT 153 // opt不是指定的参数之一;
199 #define OS_FLAG_GRP_DEPLETED 154 // 系统没有剩余的空闲事件标志组, 需要更改OS_CFG.H中的事件标志组
    数目配置
200
201 /*$PAGE*/
202 /*
203 ****
204 * 事件控制块(EVENT CONTROL BLOCK)
205 ****
206 */
207
208 #if (OS_EVENT_EN > 0) && (OS_MAX_EVENTS > 0)
209 typedef struct { // 定义一个时间控制块结构(OS_EVENT)
210     INT8U OSEventType; // 事件类型
211     INT8U OSEventGrp; // 等待任务所在的组
212     INT16U OSEventCnt; // 计数器(当事件是信号量时)
213     void *OSEventPtr; // 指向消息或者消息队列的指针
214     INT8U OSEventTbl[OS_EVENT_TBL_SIZE]; // 等待任务列表
215 } OS_EVENT;
216 #endif
217
218
219 /*
220 ****
221 * 事件标志控制块(EVENT FLAGS CONTROL BLOCK)
222 ****

```

```

223 */
224 //当版本为2.51 且 事件标志允许 且最大事件标志大于0时
225 #if (OS_VERSION >= 251) && (OS_FLAG_EN > 0) && (OS_MAX_FLAGS > 0)
226 typedef struct { // 定义一个OS_FLAG_GRP结构
227     INT8U      OSFlagType; // 用来检验指针的类型是否是指向事件标志组的指针
228     void        *OSFlagWaitList; // 包含了一个等待事件的任务列表
229     OS_FLAGS     OSFlagFlags; // 包含了一系列表明当前事件标志状态的位
230 } OS_FLAG_GRP; // 事件标志组
231
232
233
234 typedef struct { // 定义一个事件标志等待列表节点(OS_FLAG_NODE)结构
235     void        *OSFlagNodeNext; // 构建双向OS_FLAG_NODE数据结构链表的后一个链接
236     void        *OSFlagNodePrev; // 构建双向OS_FLAG_NODE数据结构链表的前一个链接
237     void        *OSFlagNodeTCB; // 指向某个等待事件标志组中的事件标志任务的控制块
238     void        *OSFlagNodeFlagGrp; // 是一个反向指向事件标志组的指针
239     OS_FLAGS     OSFlagNodeFlags; // 用来指明任务等待事件标志组中的哪些事件标志
240     INT8U        OSFlagNodeWaitType; // 指明等待事件标志组中的所有事件标志的发生(与、或)
241 // OS_FLAG_WAIT_AND 与
242 // OS_FLAG_WAIT_ALL 全部
243 // OS_FLAG_WAIT_OR 或
244 // OS_FLAG_WAIT_ANY 任一
245 } OS_FLAG_NODE;
246 #endif
247
248
249 /*
250 *****
251 *                设定一个消息队列的数据结构 (MESSAGE MAILBOX DATA)
252 *****
253 */
254
255 #if OS_MBOX_EN > 0
256 typedef struct { // 定义一个OS_MBOX_DATA结构
257     void        *OSMsg; // 如果消息队列中有消息, 它包含指针.OSQOut所指向的队列单元中
258 // 的内容。如果队列是空的, .OSMsg包含一个NULL指针
259     INT8U      OSEventTbl[OS_EVENT_TBL_SIZE]; // 消息队列的等待任务列表
260     INT8U      OSEventGrp; // 于OSEventTbl[]配合使用
261 } OS_MBOX_DATA;
262 #endif
263
264 /*
265 *****
266 *                设定一个内存的数据结构 (MEMORY PARTITION DATA STRUCTURES)
267 *****
268 */
269
270 #if (OS_MEM_EN > 0) && (OS_MAX_MEM_PART > 0)
271 typedef struct { // 使用内存控制块(memory control blocks)的数据结构来跟踪每一
272 // 个内存分区, 系统中的每个内存分区都有它自己的内存控制块。
273     void        *OSMemAddr; // 指向内存分区起始地址的指针
274     void        *OSMemFreeList; // 指向下一个空闲内存控制块或者下一个空闲的内存块的指针
275     INT32U      OSMemBlkSize; // 是内存分区中内存块的大小, 是用户建立该内存分区时指定的
276     INT32U      OSMemNBls; // 是内存分区中总的内存块数量, 也是用户建立该内存分区时指定的
277     INT32U      OSMemNFree; // 是内存分区中当前可以得空闲内存块数量
278 } OS_MEM;
279
280
281 typedef struct { // 定义一个内存数据结构(OS_MEM_DATA)
282     void        *OSAddr; // 指向内存区起始地址的指针
283     void        *OSFreeList; // 指向空闲内存块列表起始地址的指针
284     INT32U      OSBlkSize; // 每个内存块的大小
285     INT32U      OSNBls; // 该内存区的内存块总数
286     INT32U      OSNFree; // 空闲的内存块数目
287     INT32U      OSNUsed; // 使用的内存块数目
288 } OS_MEM_DATA;
289 #endif
290
291 /*$PAGE*/
292 /*
293 *****
294 *                互斥型信号量数据(MUTUAL EXCLUSION SEMAPHORE DATA)
295 *****
296 */
297
298 #if OS_MUTEX_EN > 0 // 允许(1)或者产生互斥型信号量相关代码

```

```

299 typedef struct { // 定义指向类型为(OS_MUTEX_DATA)的数据结构的指针
300     INT8U   OSEventTbl[OS_EVENT_TBL_SIZE]; // 容量大小由ucos_ii.H
301     INT8U   OSEventGrp; // 复制等待mutex的任务列表
302     INT8U   OSValue; // 当前mutex的值.1表示可以使用, 0表示不能使用
303     INT8U   OSOwnerPrio; // 占用mutex任务的优先级
304     INT8U   OSMutexPIP; // mutex的优先级继承优先级PIP
305 } OS_MUTEX_DATA;
306 #endif
307
308 /*
309 ****
310 * 消息队列数据 (MESSAGE QUEUE DATA)
311 ****
312 */
313 /*
314 队列控制块是一个用于维护消息队列信息的数据结构, 它包含了以下的一些域。这里, 仍然在各个变量前加入
315 * 一个[.]来表示它们是数据结构中的一个域。
316 * 1). OSQPtr: 在空闲队列控制块中链接所有的队列控制块。一旦建立了消息队列, 该域就不再有用了。
317 * 2). OSQStart: 是指向消息队列的指针数组的起始地址的指针。用户应用程序在使用消息队列之前必须先定义该数组
318 * 3). OSQEnd: 是指向消息队列结束单元的下一个地址的指针。该指针使得消息队列构成一个循环的缓冲区。
319 * 4). OSQIn: 是指向消息队列中插入下一条消息的位置的指针。当.OSQIn和.OSQEnd相等时,.OSQIn被调整指向
320 * 消息队列的起始单元。
321 * 5). OSQOut: 是指向消息队列中下一个取出消息的位置的指针。当.OSQOut和.OSQEnd相等时,.OSQOut被调整指
322 * 向消息队列的起始单元。
323 * 6). OSQSize: 是消息队列中总的单元数。该值是在建立消息队列时由用户应用程序决定的。在uC/OS-II中, 该值最
324 * 大可以是65,535。
325 * 7). OSQEntries: 是消息队列中当前的消息数量。当消息队列是空的时, 该值为0。当消息队列满了以后, 该值和
326 * .OSQSize值一样。在消息队列刚刚建立时, 该值为0。
327 */
328 #if OS_Q_EN > 0
329 typedef struct os_q { // 定义一个OS_Q队列控制块
330     struct os_q *OSQPtr; // 1)
331     void **OSQStart; // 2)
332     void **OSQEnd; // 3)
333     void **OSQIn; // 4)
334     void **OSQOut; // 5)
335     INT16U OSQSize; // 6)
336     INT16U OSQEntries; // 7)
337 } OS_Q;
338
339
340 typedef struct { // 定义一个消息队列数据(OS_Q_DATA)结构
341     void *OSMsg; // 如果消息队列中有消息, 它包含指针
342     INT16U OSNMsgs; // 是消息队列中的消息数
343     INT16U OSQSize; // 是消息队列的总的容量
344     INT8U OSEventTbl[OS_EVENT_TBL_SIZE];
345     INT8U OSEventGrp; //和OSEventTbl[]一起结合, 是消息队列的等待任务列表
346 } OS_Q_DATA;
347 #endif
348
349 /*
350 ****
351 * 信号量数据结构 (SEMAPHORE DATA)
352 ****
353 */
354
355 #if OS_SEM_EN > 0
356 typedef struct { // 定义一个信号量数据结构(OS_SEM_DATA)
357     INT16U OSCnt; // 定义信号量计数值
358     INT8U OSEventTbl[OS_EVENT_TBL_SIZE]; // 定义任务等待列表
359     INT8U OSEventGrp; // 定义等待事件的任务组
360 } OS_SEM_DATA;
361 #endif
362
363 /*
364 ****
365 * 任务堆栈数据 (TASK STACK DATA)
366 ****
367 */
368
369 #if OS_TASK_CREATE_EXT_EN > 0
370 typedef struct { // 定义一个堆栈数据结构(OS_STK_DATA)
371     INT32U OSFree; // 堆栈中未使用的字节数
372     INT32U OSUsed; // 堆栈中已使用的字节数
373 } OS_STK_DATA;

```



```

374 #endif
375
376 /*$PAGE*/
377 /*
378 ****
379 *                               任务控制块 (TASK CONTROL BLOCK)
380 ****
381 */
382
383 typedef struct os_tcb {
384     OS_STK      *OSTCBStkPtr;           //当前TCB的栈顶指针
385
386     #if OS_TASK_CREATE_EXT_EN > 0       //允许生成OSTaskCreateExt()函数
387         void      *OSTCBExtPtr;         //指向用户定义的任务控制块(扩展指针)
388         OS_STK      *OSTCBStkBottom;    //指向指向栈底的指针
389         INT32U      OSTCBStkSize;       //设定堆栈的容量
390         INT16U      OSTCBOpt;           //保存OS_TCB的选择项
391         INT16U      OSTCBId;            //否则使用旧的参数
392     #endif
393
394     struct os_tcb *OSTCBNext;           //定义指向TCB的双向链接的后链接
395     struct os_tcb *OSTCBPrev;           //定义指向TCB的双向链接的前链接
396
397     #if ((OS_Q_EN > 0) && (OS_MAX_QS > 0)) || (OS_MBOX_EN > 0) || (OS_SEM_EN > 0) || (OS_MUTEX_EN > 0)
398         //当以上各种事件允许时
399         OS_EVENT      *OSTCBEvtPtr;     //定义指向事件控制块的指针
400     #endif
401
402     #if ((OS_Q_EN > 0) && (OS_MAX_QS > 0)) || (OS_MBOX_EN > 0)
403         void      *OSTCBMsg;           //满足以上条件, 定义传递给任务的消息指针
404     #endif
405
406     #if (OS_VERSION >= 251) && (OS_FLAG_EN > 0) && (OS_MAX_FLAGS > 0)
407     #if OS_TASK_DEL_EN > 0
408         OS_FLAG_NODE *OSTCBFlagNode;    //定义事件标志节点的指针
409     #endif
410     OS_FLAGS      OSTCBFlagsRdy;        //定义运行准备完毕的任务控制块中的任务?
411     #endif
412
413     INT16U      OSTCBDly;               //定义允许任务等待时的最多节拍数
414     INT8U      OSTCBStat;               //定义任务的状态字
415     INT8U      OSTCBPrio;               //定义任务的优先级
416
417     INT8U      OSTCBX;                  //定义指向任务优先级的低3位, 即=priority&0x07
418     INT8U      OSTCBY;                  //定义指向任务优先级的高3位, 即=priority>>3
419     INT8U      OSTCBBitX;               //定义低3位就绪表对应值(0~7), 即=OSMapTbl[priority&0x07]
420     INT8U      OSTCBBitY;               //定义高3位就绪表对应值(0~7), 即=OSMapTbl[priority>>3]
421
422     #if OS_TASK_DEL_EN > 0               //允许生成 OSTaskDel() 函数代码函数
423         BOOLEAN      OSTCBDelReq;       //定义用于表示该任务是否须删除
424     #endif
425 } OS_TCB;
426
427 /*$PAGE*/
428 /*
429 ****
430 *                               全局变量 (GLOBAL VARIABLES)
431 ****
432 */
433
434 OS_EXT  INT32U      OSCtxSwCtr;           //上下文切换的次数(统计任务计数器)
435
436 #if (OS_EVENT_EN > 0) && (OS_MAX_EVENTS > 0) //如果有消息事件, 并且最大消息事件数>0
437 OS_EXT  OS_EVENT      *OSEventFreeList;   //空余事件管理列表指针
438 OS_EXT  OS_EVENT      OSEventTbl[OS_MAX_EVENTS]; //任务等待表首地址
439 #endif
440 //当满足版本大于2.51且事件标志允许且有最大事件标志
441 #if (OS_VERSION >= 251) && (OS_FLAG_EN > 0) && (OS_MAX_FLAGS > 0)
442 OS_EXT  OS_FLAG_GRP    OSFlagTbl[OS_MAX_FLAGS]; //定义一个事件标志列表
443 OS_EXT  OS_FLAG_GRP    *OSFlagFreeList;         //定义一个空闲的事件标志组
444 #endif
445
446 #if OS_TASK_STAT_EN > 0 //定义允许生产OS_TaskStat()函数
447 OS_EXT  INT8S          OSCPUUsage;           //定义CPU 使用率
448 OS_EXT  INT32U          OSIdleCtrMax;         //定义最大空闲计数值
449 OS_EXT  INT32U          OSIdleCtrRun;         //定义当前的空闲计数值

```



```

450 OS_EXT  BOOLEAN          OSStatRdy;                //定义统计任务就绪标志
451 OS_EXT  OS_STK           OStaskStatStk[OS_TASK_STAT_STK_SIZE]; //定义任务堆栈栈底指针
452 #endif
453
454 OS_EXT  INT8U            OSIntNesting;              //定义中断嵌套层数
455 OS_EXT  INT8U            OSIntExitY;                //用于函数OSInieExt( )
456
457 OS_EXT  INT8U            OSLockNesting;             //定义锁定嵌套计数器
458
459 OS_EXT  INT8U            OSPrioCur;                //定义正在运行的任务的优先级
460 OS_EXT  INT8U            OSPrioHighRdy;             //定义具有最高优先级级别的就绪任务的优先级
461
462 OS_EXT  INT8U            OSRdyGrp;                 //每i位对应OSRdyTbl[i]组有任务就绪0~7
463 OS_EXT  INT8U            OSRdyTbl[OS_RDY_TBL_SIZE]; //每i位对应OSRdyTbl[i*OSRdyGrp]的优先级任务
464
465 OS_EXT  BOOLEAN          OSRunning;                 //多任务已经开始=1, 任务处于不运行状态=0
466
467 OS_EXT  INT8U            OStaskCtr;                 //定义任务计数器
468
469 OS_EXT  INT32U           OSIdleCtr;                 //定义32位空闲任务的计数器
470
471 OS_EXT  OS_STK           OStaskIdleStk[OS_TASK_IDLE_STK_SIZE]; //分配空闲任务堆栈栈顶指针
472
473
474 OS_EXT  OS_TCB           *OSTCBCur;                 //定义指向正在运行任务控制块的指针
475 OS_EXT  OS_TCB           *OSTCBFreeList;            //定义空任务控制块指针
476 OS_EXT  OS_TCB           *OSTCBHighRdy;            //定义指向最高级优先级就绪任务控制块的指针
477 OS_EXT  OS_TCB           *OSTCBLst;                 //定义任务控制块列表首地址
478 OS_EXT  OS_TCB           *OSTCBPrioTbl[OS_LOWEST_PRIO + 1]; //定义任务控制块优先级表
479 OS_EXT  OS_TCB           OSTCBTbl[OS_MAX_TASKS + OS_N_SYS_TASKS]; //定义当前任务控制块列表
480
481                                     //条件编译: 若两个条件满足时, 产生以下代码
482                                     //OS_MEM_EN 允许 (1) 或者禁止 (0) 产生内存相关代码
483                                     //OS_MAX_MEM_PART 最多内存块数目
484 #if (OS_MEM_EN > 0) && (OS_MAX_MEM_PART > 0)
485 OS_EXT  OS_MEM           *OSMemFreeList;            //定义空余内存控制块(链接)
486 OS_EXT  OS_MEM           OSMemTbl[OS_MAX_MEM_PART]; //定义内存块最大地址
487 #endif
488
489 #if (OS_Q_EN > 0) && (OS_MAX_QS > 0)                //条件编译: OS_Q_EN 允许 (1) 产生消息队列相关代码
490                                     //条件编译: 应用中最多对列控制块的数目 > 0
491 OS_EXT  OS_Q             *OSQFreeList;              //定义空余队列控制链表的队列控制块
492 OS_EXT  OS_Q             OSQTbl[OS_MAX_QS];         //定义消息队列最大数
493 #endif
494
495 #if OS_TIME_GET_SET_EN > 0                          //允许生成OSTimeGet() 函数代码
496 OS_EXT  volatile INT32U  OSTime;                    //当前系统时钟数值
497 #endif
498
499 extern  INT8U  const      OSMatTbl[];                //该索引可得到优先级任务在. OSEventGrp中的位屏蔽码
500 extern  INT8U  const      OSUnMatTbl[];              //查找最高优先级任务号索引表
501
502 /*$PAGE*/
503 /*
504 ****
505 *                                     功能原型 (FUNCTION PROTOTYPES)
506 *                                     不受约束的函数 (Target Independent Functions)
507 ****
508 */
509
510 /*
511 ****
512 *                                     事件标志管理 (EVENT FLAGS MANAGEMENT)
513 *
514 * OSFlagAccept() 检查事件标志组函数(标志组的指针、事件标志位、等待事件标志位的方式、错误码指针)
515 * OSFlagCreate() 建立一个事件标志组(初值、错误码)
516 * OSFlagDel()   删除一个事件标志组(指针、条件值、错误码)
517 * OSFlagPend() 等待事件标志组的事件标志位(事件组指针、需要检查的标志位、等待事件标志位的方式、
518 *                                     允许等待的时钟节拍、出错代码的时钟节拍)
519 * OSFlagPost() 置位或清0事件标志组中的标志位(指针、标志位、条件值、错误码)
520 * OSFlagQuery() 查询事件标志组的当前事件标志状态(事件标志组的指针、错误代码的指针)
521 *
522 ****
523 */
524
525 #if (OS_VERSION >= 251) && (OS_FLAG_EN > 0) && (OS_MAX_FLAGS > 0)

```

```

526
527 #if OS_FLAG_ACCEPT_EN > 0
528 OS_FLAGS      OSFlagAccept(OS_FLAG_GRP *pgrp, OS_FLAGS flags, INT8U wait_type, INT8U *err);
529 #endif
530
531 OS_FLAG_GRP   *OSFlagCreate(OS_FLAGS flags, INT8U *err);
532
533 #if OS_FLAG_DEL_EN > 0
534 OS_FLAG_GRP   *OSFlagDel(OS_FLAG_GRP *pgrp, INT8U opt, INT8U *err);
535 #endif
536
537 OS_FLAGS      OSFlagPend(OS_FLAG_GRP *pgrp, OS_FLAGS flags, INT8U wait_type, INT16U timeout, INT8U *err);
538 OS_FLAGS      OSFlagPost(OS_FLAG_GRP *pgrp, OS_FLAGS flags, INT8U operation, INT8U *err);
539
540 #if OS_FLAG_QUERY_EN > 0
541 OS_FLAGS      OSFlagQuery(OS_FLAG_GRP *pgrp, INT8U *err);
542 #endif
543 #endif
544
545 /*
546 ****
547 *                               消息邮箱管理 (MESSAGE MAILBOX MANAGEMENT)
548 *
549 * OSMboxAccept () 查看消息邮箱(消息邮箱指针)
550 * OSMboxCreate () 建立并初始化一个消息邮箱(msg 参数不为空含内容)
551 * OSMboxDel ()   删除消息邮箱(消息邮箱指针、删除条件、出错代码指针)
552 * OSMboxPend ()  等待一个消息邮箱函数(消息邮箱指针、允许等待的时钟节拍、代码错误指针)
553 * OSMboxPost ()  发送消息函数(消息邮箱指针、即将实际发送给任务的消息)
554 * OSMboxPostOpt () 向邮箱发送一则消息(邮箱指针、消息、条件)
555 * OSMboxQuery () 查询一个邮箱的当前状态(信号量指针、状态数据结构指针)
556 ****
557 */
558
559 #if OS_MBOX_EN > 0
560
561 #if OS_MBOX_ACCEPT_EN > 0
562 void      *OSMboxAccept(OS_EVENT *pevent);
563 #endif
564
565 OS_EVENT   *OSMboxCreate(void *msg);
566
567 #if OS_MBOX_DEL_EN > 0
568 OS_EVENT   *OSMboxDel(OS_EVENT *pevent, INT8U opt, INT8U *err);
569 #endif
570
571 void      *OSMboxPend(OS_EVENT *pevent, INT16U timeout, INT8U *err);
572
573 #if OS_MBOX_POST_EN > 0
574 INT8U      OSMboxPost(OS_EVENT *pevent, void *msg);
575 #endif
576
577 #if OS_MBOX_POST_OPT_EN > 0
578 INT8U      OSMboxPostOpt(OS_EVENT *pevent, void *msg, INT8U opt);
579 #endif
580
581 #if OS_MBOX_QUERY_EN > 0
582 INT8U      OSMboxQuery(OS_EVENT *pevent, OS_MBOX_DATA *pdata);
583 #endif
584 #endif
585
586 /*
587 ****
588 *                               内存管理项 (MEMORY MANAGEMENT)
589 *
590 * OSMemCreate () 建立并初始化一块内存区(起始地址、需要的内存块数目、内存块大小、返回错误的指针)
591 * OSMemGet ()   从内存区分配一个内存块
592 * OSMemPut ()   释放一个内存块, 内存块必须释放回原先申请的内存区
593 * OSMemQuery () 得到内存区的信息
594 ****
595 */
596
597 #if (OS_MEM_EN > 0) && (OS_MAX_MEM_PART > 0)
598
599 OS_MEM      *OSMemCreate(void *addr, INT32U nblks, INT32U blksize, INT8U *err);
600 void      *OSMemGet(OS_MEM *pmem, INT8U *err);
601 INT8U      OSMemPut(OS_MEM *pmem, void *pblk);

```

```

602
603 #if OS_MEM_QUERY_EN > 0
604 INT8U      OSMemQuery(OS_MEM *pmem, OS_MEM_DATA *pdata);
605 #endif
606
607 #endif
608
609 /*
610 ****
611 *          互斥型信号量项管理 (MUTUAL EXCLUSION SEMAPHORE MANAGEMENT)
612 *
613 * OSMutexAccept ()  无等待地获取互斥型信号量[任务不挂起](信号量指针、错误代码)
614 * OSMutexCreate ()  建立并初始化一个互斥型信号量(优先级继承优先级(PIP)、出错代码指针)
615 * OSMutexDel ()     删除互斥型信号量(信号指针、删除条件、错误指针)
616 * OSMutexPend ()    等待一个互斥型信号量(指针、等待超时时限、出错代码指针)
617 * OSMutexPost ()    释放一个互斥型信号量(互斥型信号量指针)
618 * OSMutexQuery ()   查询一个互斥型信号量的当前状态(互斥型信号量指针、状态数据结构指针)
619 ****
620 */
621
622 #if OS_MUTEX_EN > 0
623
624 #if OS_MUTEX_ACCEPT_EN > 0
625 INT8U      OSMutexAccept(OS_EVENT *pevent, INT8U *err);
626 #endif
627
628 OS_EVENT    *OSMutexCreate(INT8U prio, INT8U *err);
629
630 #if OS_MUTEX_DEL_EN > 0
631 OS_EVENT    *OSMutexDel(OS_EVENT *pevent, INT8U opt, INT8U *err);
632 #endif
633
634 void         OSMutexPend(OS_EVENT *pevent, INT16U timeout, INT8U *err);
635 INT8U        OSMutexPost(OS_EVENT *pevent);
636
637 #if OS_MUTEX_QUERY_EN > 0
638 INT8U        OSMutexQuery(OS_EVENT *pevent, OS_MUTEX_DATA *pdata);
639 #endif
640
641 #endif
642
643 /*$PAGE*/
644 /*
645 ****
646 *          消息队列管理 (MESSAGE QUEUE MANAGEMENT)
647 *
648 * OSQAccept ()      检查消息队列中是否已经有需要的消息(消息队列的指针)
649 * OSQCreate ()      建立一个消息队列(消息内存区的基地址(指针数组)、消息内存区的大小)
650 * OSQDel ()         删除一个消息队列(消息队列指针、删除条件、错误指针)
651 * OSQFlush ()       清空消息队列(指向得到消息队列的指针)
652 * OSQPend ()        任务等待消息队列中的消息(消息队列指针、允许等待的时钟节拍、代码错误指针)
653 * OSQPost ()        向消息队列发送一则消息FIFO(消息队列指针、发送的消息)
654 * OSQPostFront ()   向消息队列发送一则消息LIFO(消息队列指针、发送的消息)
655 * OSQPostOpt ()     向消息队列发送一则消息LIFO(消息队列指针、发送的消息、发送条件)
656 * OSQQuery ()       查询一个消息队列的当前状态(信号量指针、状态数据结构指针)
657 ****
658 */
659
660 #if (OS_Q_EN > 0) && (OS_MAX_QS > 0)
661
662 #if OS_Q_ACCEPT_EN > 0
663 void         *OSQAccept(OS_EVENT *pevent);
664 #endif
665
666 OS_EVENT    *OSQCreate(void **start, INT16U size);
667
668 #if OS_Q_DEL_EN > 0
669 OS_EVENT    *OSQDel(OS_EVENT *pevent, INT8U opt, INT8U *err);
670 #endif
671
672 #if OS_Q_FLUSH_EN > 0
673 INT8U        OSQFlush(OS_EVENT *pevent);
674 #endif
675
676 void         *OSQPend(OS_EVENT *pevent, INT16U timeout, INT8U *err);
677

```

```

678 #if OS_Q_POST_EN > 0
679 INT8U      OSQPost(OS_EVENT *pevent, void *msg);
680 #endif
681
682 #if OS_Q_POST_FRONT_EN > 0
683 INT8U      OSQPostFront(OS_EVENT *pevent, void *msg);
684 #endif
685
686 #if OS_Q_POST_OPT_EN > 0
687 INT8U      OSQPostOpt(OS_EVENT *pevent, void *msg, INT8U opt);
688 #endif
689
690 #if OS_Q_QUERY_EN > 0
691 INT8U      OSQQuery(OS_EVENT *pevent, OS_Q_DATA *pdata);
692 #endif
693
694 #endif
695
696 /*$PAGE*/
697 /*
698 ****
699 *                                     信号量管理 (SEMAPHORE MANAGEMENT)
700 *
701 * OS_SemAccept() 无条件地等待请求一个信号量函数
702 * OS_SemCreate() 建立并初始化一个信号量(输入一个信号量值)
703 * OS_SemDel()   删除一个信号量(信号指针、删除条件、错误指针)
704 * OS_SemPend()  等待一个信号量函数(信号量指针、允许等待的时钟节拍、代码错误指针)
705 * OS_SemPost()  发出一个信号量函数(信号量指针)
706 * OS_SemQuery() 查询一个信号量的当前状态(信号量指针、状态数据结构指针)
707 ****
708 */
709 #if OS_SEM_EN > 0
710
711 #if OS_SEM_ACCEPT_EN > 0
712 INT16U      OSSemAccept(OS_EVENT *pevent);
713 #endif
714
715 OS_EVENT    *OSSemCreate(INT16U cnt);
716
717 #if OS_SEM_DEL_EN > 0
718 OS_EVENT    *OSSemDel(OS_EVENT *pevent, INT8U opt, INT8U *err);
719 #endif
720
721 void         OSSemPend(OS_EVENT *pevent, INT16U timeout, INT8U *err);
722 INT8U        OSSemPost(OS_EVENT *pevent);
723
724 #if OS_SEM_QUERY_EN > 0
725 INT8U        OSSemQuery(OS_EVENT *pevent, OS_SEM_DATA *pdata);
726 #endif
727
728 #endif
729
730 /*$PAGE*/
731 /*
732 ****
733 *                                     任务管理 (TASK MANAGEMENT)
734 *
735 * OSTaskChangePrio() 改变一个任务的优先级(任务旧的优先级、任务新的优先级)
736 * OSTaskCreate()     建立任务(任务代码指针、传递参数指针、分配任务堆栈栈顶指针、任务优先级)
737 * OSTaskCreateExt()  建立扩展任务(任务代码指针/传递参数指针/分配任务堆栈栈顶指针/分配任务优先级
738 *                    //(未来的)优先级标识(与优先级相同)/分配任务堆栈栈底指针/指定堆栈的容量(检验用)
739 *                    //指向用户附加的数据域的指针/建立任务设定选项)
740 * OSTaskDel()        删除任务(任务的优先级)
741 * OSTaskDelReq()     请求一个任务删除其它任务或自身?(任务的优先级)
742 * OSTaskResume()     唤醒一个用OSTaskSuspend()函数挂起的任务(任务的优先级)
743 * OSTaskStkChk()     检查任务堆栈状态(任务优先级、检验堆栈数据结构)
744 * OSTaskSuspend()    无条件挂起一个任务(任务优先级)
745 * OSTaskQuery()      获取任务信息(任务指针、保存数据结构指针)
746 ****
747 */
748 #if OS_TASK_CHANGE_PRIO_EN > 0
749 INT8U      OSTaskChangePrio(INT8U oldprio, INT8U newprio);
750 #endif
751
752 #if OS_TASK_CREATE_EN > 0
753 INT8U      OSTaskCreate(void (*task)(void *pd), void *pdata, OS_STK *ptos, INT8U prio);

```

```

754 #endif
755
756 #if OS_TASK_CREATE_EXT_EN > 0
757 INT8U      OSTaskCreateExt(void (*task)(void *pd),
758                             void *pdata,
759                             OS_STK *ptos,
760                             INT8U prio,
761                             INT16U id,
762                             OS_STK *pbos,
763                             INT32U stk_size,
764                             void *pext,
765                             INT16U opt);
766 #endif
767
768 #if OS_TASK_DEL_EN > 0
769 INT8U      OSTaskDel(INT8U prio);
770 INT8U      OSTaskDelReq(INT8U prio);
771 #endif
772
773 #if OS_TASK_SUSPEND_EN > 0
774 INT8U      OSTaskResume(INT8U prio);
775 INT8U      OSTaskSuspend(INT8U prio);
776 #endif
777
778 #if OS_TASK_CREATE_EXT_EN > 0
779 INT8U      OSTaskStkChk(INT8U prio, OS_STK_DATA *pdata);
780 #endif
781
782 #if OS_TASK_QUERY_EN > 0
783 INT8U      OSTaskQuery(INT8U prio, OS_TCB *pdata);
784 #endif
785
786 /*$PAGE*/
787 /*
788 ****
789 *                                     时钟管理项 (TIME MANAGEMENT)
790 *
791 * OSTimeDly ()      任务延时函数(时钟节拍数)
792 * OSTimeDlyHMSM ()  将一个任务延时若干时间(设定时、分、秒、毫秒)
793 * OSTimeDlyResume () 唤醒一个用OSTimeDly()或OSTimeDlyHMSM()函数的任务(优先级)
794 * OSTimeGet ()      获取当前系统时钟数值
795 * OSTimeSet ()      设置当前系统时钟数值
796 ****
797 */
798
799 void      OSTimeDly(INT16U ticks);
800
801 #if OS_TIME_DLY_HMSM_EN > 0
802 INT8U      OSTimeDlyHMSM(INT8U hours, INT8U minutes, INT8U seconds, INT16U milli);
803 #endif
804
805 #if OS_TIME_DLY_RESUME_EN > 0
806 INT8U      OSTimeDlyResume(INT8U prio);
807 #endif
808
809 #if OS_TIME_GET_SET_EN > 0
810 INT32U      OSTimeGet(void);
811 void      OSTimeSet(INT32U ticks);
812 #endif
813 18:31 2007-5-16
814 void      OSTimeTick(void);
815
816 /*****
817 *                                     混杂函数定义
818 *
819 * OSInit()          初始化UCOS-II函数
820 * OSIntEnter()      中断函数正在执行
821 * OSIntExit()       中断函数已经完成(脱离中断)
822 * OSSchedLock()     给调度器上锁
823 * OSSchedUnlock()   给调度器解锁
824 * OSStart()         启动多个任务
825 * OSStatInit()      统计任务初始化
826 * OSVersion()       获得版本号
827 *
828 *****/
829

```



```

905 *****
906 *                                     各类钩子程序函数定义项 (FUNCTION PROTOTYPES)
907 *                                     特别钩子函数原型 (Target Specific Functions)
908 *****
909 */
910
911 #if OS_VERSION >= 204                                //当版本大于2.04
912 void      OSInitHookBegin(void);
913 void      OSInitHookEnd(void);
914 #endif
915
916 void      OSIntCtxSw(void);
917
918 void      OSStartHighRdy(void);
919
920 void      OSTaskCreateHook(OS_TCB *ptcb);
921 void      OSTaskDelHook(OS_TCB *ptcb);
922
923 #if OS_VERSION >= 251
924 void      OSTaskIdleHook(void);
925 #endif
926
927 void      OSTaskStatHook(void);
928 OS_STK    *OSTaskStkInit(void (task)(void *pd), void *pdata, OS_STK *ptos, INT16U opt);
929 void      OSTaskSwHook(void);
930
931 #if OS_VERSION >= 204
932 void      OSTCBInitHook(OS_TCB *ptcb);
933 #endif
934
935 void      OSTimeTickHook(void);
936
937 /*
938 *****
939 *                                     函数原型      FUNCTION PROTOTYPES
940 *                                     特殊中断函数原型 (Compiler Specific ISR prototypes)
941 *****
942 */
943
944 #ifndef OS_ISR_PROTO_EXT
945 void      OSCtxSw(void);    //上下文切换函数
946 void      OSTickISR(void);  //
947 #endif
948
949 /*$PAGE*/
950 /*
951 *****
952 *                                     LOOK FOR MISSING #define CONSTANTS
953 *
954 * This section is used to generate ERROR messages at compile time if certain #define constants are
955 * MISSING in OS_CFG.H. This allows you to quickly determine the source of the error.
956 *
957 * You SHOULD NOT change this section UNLESS you would like to add more comments as to the source of the
958 * compile time error.
959 *****
960 */
961
962 /*****
963 * 事件标志管理条件编译
964 *****/
965
966 #ifndef OS_FLAG_EN
967 #error "OS_CFG.H, Missing OS_FLAG_EN: Enable (1) or Disable (0) code generation for Event Flags"
968 #else
969     #ifndef OS_MAX_FLAGS
970     #error "OS_CFG.H, Missing OS_MAX_FLAGS: Max. number of Event Flag Groups in your application"
971     #else
972         #if OS_MAX_FLAGS == 0
973         #error "OS_CFG.H, OS_MAX_FLAGS must be > 0"
974         #endif
975         #if OS_MAX_FLAGS > 255
976         #error "OS_CFG.H, OS_MAX_FLAGS must be <= 255"
977         #endif
978     #endif
979
980 #ifndef OS_FLAG_WAIT_CLR_EN

```

```

981 #error "OS_CFG.H, Missing OS_FLAG_WAIT_CLR_EN: Include code for Wait on Clear EVENT FLAGS"
982 #endif
983
984 #ifndef OS_FLAG_ACCEPT_EN
985 #error "OS_CFG.H, Missing OS_FLAG_ACCEPT_EN: Include code for OSFlagAccept()"
986 #endif
987
988 #ifndef OS_FLAG_DEL_EN
989 #error "OS_CFG.H, Missing OS_FLAG_DEL_EN: Include code for OSFlagDel()"
990 #endif
991
992 #ifndef OS_FLAG_QUERY_EN
993 #error "OS_CFG.H, Missing OS_FLAG_DEL_EN: Include code for OSFlagQuery()"
994 #endif
995 #endif
996
997 /*****
998 * 消息邮箱管理条件编译
999 *****/
1000
1001 #ifndef OS_MBOX_EN
1002 #error "OS_CFG.H, Missing OS_MBOX_EN: Enable (1) or Disable (0) code generation for MAILBOXES"
1003 #else
1004 #ifndef OS_MBOX_ACCEPT_EN
1005 #error "OS_CFG.H, Missing OS_MBOX_ACCEPT_EN: Include code for OSMboxAccept()"
1006 #endif
1007
1008 #ifndef OS_MBOX_DEL_EN
1009 #error "OS_CFG.H, Missing OS_MBOX_DEL_EN: Include code for OSMboxDel()"
1010 #endif
1011
1012 #ifndef OS_MBOX_POST_EN
1013 #error "OS_CFG.H, Missing OS_MBOX_POST_EN: Include code for OSMboxPost()"
1014 #endif
1015
1016 #ifndef OS_MBOX_POST_OPT_EN
1017 #error "OS_CFG.H, Missing OS_MBOX_POST_OPT_EN: Include code for OSMboxPostOpt()"
1018 #endif
1019
1020 #ifndef OS_MBOX_QUERY_EN
1021 #error "OS_CFG.H, Missing OS_MBOX_QUERY_EN: Include code for OSMboxQuery()"
1022 #endif
1023 #endif
1024
1025 /*****
1026 * 内存管理条件编译
1027 *****/
1028
1029 #ifndef OS_MEM_EN
1030 #error "OS_CFG.H, Missing OS_MEM_EN: Enable (1) or Disable (0) code generation for MEMORY MANAGER"
1031 #else
1032 #ifndef OS_MAX_MEM_PART
1033 #error "OS_CFG.H, Missing OS_MAX_MEM_PART: Max. number of memory partitions"
1034 #else
1035 #if OS_MAX_MEM_PART == 0
1036 #error "OS_CFG.H, OS_MAX_MEM_PART must be > 0"
1037 #endif
1038 #if OS_MAX_MEM_PART > 255
1039 #error "OS_CFG.H, OS_MAX_MEM_PART must be <= 255"
1040 #endif
1041 #endif
1042
1043 #ifndef OS_MEM_QUERY_EN
1044 #error "OS_CFG.H, Missing OS_MEM_QUERY_EN: Include code for OSMemQuery()"
1045 #endif
1046 #endif
1047
1048 /*****
1049 * 互斥型信号量管理条件编译
1050 *****/
1051
1052 #ifndef OS_MUTEX_EN
1053 #error "OS_CFG.H, Missing OS_MUTEX_EN: Enable (1) or Disable (0) code generation for MUTEX"
1054 #else
1055 #ifndef OS_MUTEX_ACCEPT_EN
1056 #error "OS_CFG.H, Missing OS_MUTEX_ACCEPT_EN: Include code for OSMutexAccept()"

```

```

1057     #endif
1058
1059     #ifndef OS_MUTEX_DEL_EN
1060     #error "OS_CFG.H, Missing OS_MUTEX_DEL_EN: Include code for OSMutexDel()"
1061     #endif
1062
1063     #ifndef OS_MUTEX_QUERY_EN
1064     #error "OS_CFG.H, Missing OS_MUTEX_QUERY_EN: Include code for OSMutexQuery()"
1065     #endif
1066 #endif
1067
1068 /*****
1069 * 消息队列条件编译
1070 *****/
1071
1072 #ifndef OS_Q_EN
1073 #error "OS_CFG.H, Missing OS_Q_EN: Enable (1) or Disable (0) code generation for QUEUES"
1074 #else
1075     #ifndef OS_MAX_QS
1076     #error "OS_CFG.H, Missing OS_MAX_QS: Max. number of queue control blocks"
1077     #else
1078         #if OS_MAX_QS == 0
1079         #error "OS_CFG.H, OS_MAX_QS must be > 0"
1080         #endif
1081         #if OS_MAX_QS > 255
1082         #error "OS_CFG.H, OS_MAX_QS must be <= 255"
1083         #endif
1084     #endif
1085
1086     #ifndef OS_Q_ACCEPT_EN
1087     #error "OS_CFG.H, Missing OS_Q_ACCEPT_EN: Include code for OSQAccept()"
1088     #endif
1089
1090     #ifndef OS_Q_DEL_EN
1091     #error "OS_CFG.H, Missing OS_Q_DEL_EN: Include code for OSQDel()"
1092     #endif
1093
1094     #ifndef OS_Q_FLUSH_EN
1095     #error "OS_CFG.H, Missing OS_Q_FLUSH_EN: Include code for OSQFlush()"
1096     #endif
1097
1098     #ifndef OS_Q_POST_EN
1099     #error "OS_CFG.H, Missing OS_Q_POST_EN: Include code for OSQPost()"
1100     #endif
1101
1102     #ifndef OS_Q_POST_FRONT_EN
1103     #error "OS_CFG.H, Missing OS_Q_POST_FRONT_EN: Include code for OSQPostFront()"
1104     #endif
1105
1106     #ifndef OS_Q_POST_OPT_EN
1107     #error "OS_CFG.H, Missing OS_Q_POST_OPT_EN: Include code for OSQPostOpt()"
1108     #endif
1109
1110     #ifndef OS_Q_QUERY_EN
1111     #error "OS_CFG.H, Missing OS_Q_QUERY_EN: Include code for OSQQuery()"
1112     #endif
1113 #endif
1114
1115 /*****
1116 * 信号量条件编译
1117 *****/
1118
1119 #ifndef OS_SEM_EN
1120 #error "OS_CFG.H, Missing OS_SEM_EN: Enable (1) or Disable (0) code generation for SEMAPHORES"
1121 #else
1122     #ifndef OS_SEM_ACCEPT_EN
1123     #error "OS_CFG.H, Missing OS_SEM_ACCEPT_EN: Include code for OSSemAccept()"
1124     #endif
1125
1126     #ifndef OS_SEM_DEL_EN
1127     #error "OS_CFG.H, Missing OS_SEM_DEL_EN: Include code for OSSemDel()"
1128     #endif
1129
1130     #ifndef OS_SEM_QUERY_EN
1131     #error "OS_CFG.H, Missing OS_SEM_QUERY_EN: Include code for OSSemQuery()"
1132     #endif

```

```

1133 #endif
1134
1135 /*****
1136 *   任务管理条件编译
1137 *****/
1138
1139 #ifndef OS_MAX_TASKS
1140 #error "OS_CFG.H, Missing OS_MAX_TASKS: Max. number of tasks in your application"
1141 #else
1142     #if OS_MAX_TASKS == 0
1143         #error "OS_CFG.H, OS_MAX_TASKS must be >= 2"
1144     #endif
1145     #if OS_MAX_TASKS > 63
1146         #error "OS_CFG.H, OS_MAX_TASKS must be <= 63"
1147     #endif
1148 #endif
1149
1150 #ifndef OS_TASK_IDLE_STK_SIZE
1151 #error "OS_CFG.H, Missing OS_TASK_IDLE_STK_SIZE: Idle task stack size"
1152 #endif
1153
1154 #ifndef OS_TASK_STAT_EN
1155 #error "OS_CFG.H, Missing OS_TASK_STAT_EN: Enable (1) or Disable(0) the statistics task"
1156 #endif
1157
1158 #ifndef OS_TASK_STAT_STK_SIZE
1159 #error "OS_CFG.H, Missing OS_TASK_STAT_STK_SIZE: Statistics task stack size"
1160 #endif
1161
1162 #ifndef OS_TASK_CHANGE_PRIO_EN
1163 #error "OS_CFG.H, Missing OS_TASK_CHANGE_PRIO_EN: Include code for OSTaskChangePrio()"
1164 #endif
1165
1166 #ifndef OS_TASK_CREATE_EN
1167 #error "OS_CFG.H, Missing OS_TASK_CREATE_EN: Include code for OSTaskCreate()"
1168 #endif
1169
1170 #ifndef OS_TASK_CREATE_EXT_EN
1171 #error "OS_CFG.H, Missing OS_TASK_CREATE_EXT_EN: Include code for OSTaskCreateExt()"
1172 #endif
1173
1174 #ifndef OS_TASK_DEL_EN
1175 #error "OS_CFG.H, Missing OS_TASK_DEL_EN: Include code for OSTaskDel()"
1176 #endif
1177
1178 #ifndef OS_TASK_SUSPEND_EN
1179 #error "OS_CFG.H, Missing OS_TASK_SUSPEND_EN: Include code for OSTaskSuspend() and OSTaskResume()"
1180 #endif
1181
1182 #ifndef OS_TASK_QUERY_EN
1183 #error "OS_CFG.H, Missing OS_TASK_QUERY_EN: Include code for OSTaskQuery()"
1184 #endif
1185
1186 /*****
1187 *   时间管理条件编译
1188 *****/
1189
1190 #ifndef OS_TICKS_PER_SEC
1191 #error "OS_CFG.H, Missing OS_TICKS_PER_SEC: Sets the number of ticks in one second"
1192 #endif
1193
1194 #ifndef OS_TIME_DLY_HMSM_EN
1195 #error "OS_CFG.H, Missing OS_TIME_DLY_HMSM_EN: Include code for OStimeDlyHMSM()"
1196 #endif
1197
1198 #ifndef OS_TIME_DLY_RESUME_EN
1199 #error "OS_CFG.H, Missing OS_TIME_DLY_RESUME_EN: Include code for OStimeDlyResume()"
1200 #endif
1201
1202 #ifndef OS_TIME_GET_SET_EN
1203 #error "OS_CFG.H, Missing OS_TIME_GET_SET_EN: Include code for OStimeGet() and OStimeSet()"
1204 #endif
1205
1206 /*****
1207 *   混合管理条件编译
1208 *****/

```

```
1209
1210 #ifndef OS_MAX_EVENTS
1211 #error "OS_CFG.H, Missing OS_MAX_EVENTS: Max. number of event control blocks in your application"
1212 #else
1213     #if OS_MAX_EVENTS == 0
1214     #error "OS_CFG.H, OS_MAX_EVENTS must be > 0"
1215     #endif
1216     #if OS_MAX_EVENTS > 255
1217     #error "OS_CFG.H, OS_MAX_EVENTS must be <= 255"
1218     #endif
1219 #endif
1220
1221 #ifndef OS_LOWEST_PRIO
1222 #error "OS_CFG.H, Missing OS_LOWEST_PRIO: Defines the lowest priority that can be assigned"
1223 #endif
1224
1225 #ifndef OS_ARG_CHK_EN
1226 #error "OS_CFG.H, Missing OS_ARG_CHK_EN: Enable (1) or Disable (0) argument checking"
1227 #endif
1228
1229 #ifndef OS_CPU_HOOKS_EN
1230 #error "OS_CFG.H, Missing OS_CPU_HOOKS_EN: uC/OS-II hooks are found in the processor port files when 1"
1231 #endif
1232
1233 #ifndef OS_SCHED_LOCK_EN
1234 #error "OS_CFG.H, Missing OS_SCHED_LOCK_EN: Include code for OSSchedLock() and OSSchedUnlock()"
1235 #endif
1236
```

各变量初始化情况

变量	值	类型	变量的说明
OSPrioCur	0	INT8U	正在运行的任务的优先级
OSPrioHighRdy	0	INT8U	具有最高优先级级别的就绪任务的优先级
OSTCBStat		INT8U	任务的状态字
OSTCBPrio		INT8U	任务的优先级
OSTCBExtPtr		viod	指向用户定义的扩展，只在OSTaskCreateExt()中使用
OSTCB0pt		INT16U	把“选择项”传递给函数OSTaskCreateExt()
OSTCBstkPtr		OS_TSK	指向当前任务堆栈栈顶的指针
OSTCBstkBottom		OS_TSK	指向当前任务堆栈栈底的指针
OSTCBstkSize		INT32U	存有栈中可容纳的指针数目
OSTCBID		INT16U	用于存储任务的识别码
OSTCBNext		os_tcb	用于任务块双向链接表的后链接
OSTCBPrev		os_tcb	用于任务块双向链接表的前链接
OSTCBCur	NULL	OS_TCB*	指向正在运行任务控制块的指针
OSTCBFreeList			空任务控制块指针
OSTCBHighRdy	NULL	OS_TCB*	指向最高级优先级就绪任务控制块的指针
OSTCBEvtPtr		OS_EVENT	指向事件控制块的指针
OSTCBMsg		viod	指向传递给任务的消息指针
OSTCBFlagNode		OS_FLAG_NODE	指向事件标志节点的指针
OSTCBFlagDry		OS_FLAG	当任务等待事件标志组时，使任务进入就绪状态的事件标志
OSTCBDly		INT16U	允许任务等待时的最多节拍数
OSTCBDelReq		BOOLEAN	用于表示该任务是否须删除
OSTCBPrioTbl[]			任务控制块优先级表
OSTCBY		INT8U	指向任务优先级的高3位，即=priority>>3
OSTCBBitY		INT8U	高3位就绪表对应值(0~7)，即=OSMapTbl[priority>>3]
OSTCBX		INT8U	指向任务优先级的低3位，即=priority&0x07
OSTCBBitX		INT8U	低3位就绪表对应值(0~7)，即=OSMapTbl[priority&0x07]
OSRdyGrp	0~7	INT8U	每i位对应OSRdyTbl[i]组有任务就绪0~7
OSRdyTbl[i]	0~7	INT8U	每i位对应OSRdyTbl[i*OSRdyGrp]的优先级别任务
OSMapTbl[i]			就绪表；对应OSRdyGrp和OSRdyTbl[i]的值(0~7)
OSUnMapTbl[i]			最高优先级；对应OSRdyGrp和OSRdyTbl[i]的值(0~7)
OSTime	0L	INT32U	表示系统当前时间(节拍数)
OSIntNesting	0	INT32U	存放中断嵌套的层数(0~255)
OSLockNesting	0	INT8U	调用了OSSchedLock的嵌套数
OSCtxSwCtr	0	INT32U	上下文切换的次数(统计任务计数器)
OSTaskCtr	2	INT8U	已经建立的任务数
OSRunning	FALSE	BOOLEAN	OS-II是否正在运行的标志
OSCPUUsage	0	INT8S	存放CPU的利用率(%)的变数
OSIdleCtrMax	0L	INT32U	表示每秒空闲任务计数的最大值
OSIdleCtrRun	0L	INT32U	表示空闲任务计数器每秒的计数值
OSIdleCtr	0L	INT32U	空闲任务的计数器
OSStatRdy	FALSE	BOOLEAN	统计任务是否就绪的标志
OSIntExit	0	INT8U	用于函数OSInieExt()
OSEventType		INT8U	事件的类型
OSEventCnt		INT16U	信号量的计数器
OSEventPrt		Viod *	消息或消息队列的指针
OSEventGrp		INT8U	等待事件的任务组
OSEventTbl[]		INT8U	任务等待表，OSEventTbl[OS_EVENT_TBL_SIZE]
OS_EVENT_TYPE_SEM			表示事件信号量
OS_EVENT_TYPE_MUTEX			表示事件是互斥行信号量
OS_EVENT_TYPE_MBOX			表明事件是消息邮箱
OS_EVENT_TYPE_Q			表明事件是消息队列
OS_EVENT_TYPE_UNUSED			空事件控制块(未被使用的事件控制块)


```
1 /*
2 ****
3 *
4 *
5 *          uC/OS-II实时控制内核
6 *          主要的包含文件
7 * 文   件: INCLUDES.C      ucos包含文件
8 * 作   者: Jean J. Labrosse
9 * 中文注解: 钟常慰  zhongcw @ 126.com  译注版本: 1.0 请尊重原版内容
10 ****
11 */
12
13
14 #include    <stdio.h>          //包含<stdio.h>标准输入输出文件
15 #include    <string.h>         //包含<string.h>字符文件
16 #include    <ctype.h>          //包含<ctype.h>类型转换文件
17 #include    <stdlib.h>         //包含<stdlib.h>标准函数文件
18 #include    <conio.h>          //包含<conio.h>文件
19 #include    <dos.h>            //包含<dos.h>DOS文件
20 #include    <setjmp.h>         //包含<setjmp.h>文件
21
22 /****说明: 以下几个文件需确定所在详细地址, 可相对或绝对, 否则, 整个编译连接出错, 切记*****/
23
24 #include    "\software\ucos-ii\ix861\bc45\os_cpu.h" //包含"os_cpu.h"文件, 自定义处理器内部(寄存器)内容
25 #include    "os_cfg.h"          //包含"os_cfg.h"ucos的构造文件
26 #include    "\software\ucos-ii\source\ucos_ii.h"   //包含"ucos_ii.h"内部所有ucos所有的函数内参设定
27 #include    "\software\blocks\pc\bc45\pc.h"        //包含"pc.h"程序输出在显示器荧幕显示文件
28
29 /*****结束*****/
30
```

```

1  /*
2  ****
3  *
4  *                               uC/OS-II实时控制内核
5  *                               主要的包含文件
6  *
7  * 文    件: OS_CFG.H        ucos内核构造文件
8  * 作    者: Jean J. Labrosse
9  * 中文注解: 钟常慰  zhongcw @ 126.com  译注版本: 1.0  请尊重原版内容
10 ****
11 */
12
13 /*
14 ****
15 *                               uC/OS-II 的内核构造
16 ****
17 */
18
19 #define OS_MAX_EVENTS          2    /* 应用中最多事件控制块的数目... */
20                                     /* ... 必须大于 0 */
21 #define OS_MAX_FLAGS          5    /* 应用中最多事件标志组的数目... */
22                                     /* ... 必须大于 0 */
23 #define OS_MAX_MEM_PART       5    /* 最多内存块的数目... */
24                                     /* ... 必须大于 0 */
25 #define OS_MAX_QS             2    /* 应用中最多对列控制块的数目... */
26                                     /* ... 必须大于 0 */
27 #define OS_MAX_TASKS          11   /* 应用中最多任务数目... */
28                                     /* ... 必须大于等于2 */
29
30 #define OS_LOWEST_PRIO        12   /* 定义任务的最低优先级... */
31                                     /* ... 不得大于 63 ! */
32
33 #define OS_TASK_IDLE_STK_SIZE 512  /* 统计任务堆栈容量( # 按照OS_STK的宽度数目) */
34
35 #define OS_TASK_STAT_EN       1    /* 允许 (1) 或者禁止 (0) 统计任务 */
36 #define OS_TASK_STAT_STK_SIZE 512 /* 空闲任务堆栈容量 (#按照OS_STK的宽度数目) */
37
38 #define OS_ARG_CHK_EN         1    /* 允许 (1) 或者禁止 (0) 变量检查 */
39 #define OS_CPU_HOOKS_EN       1    /* 在处理器移植文件中允许使用 uC/OS-II 的接口函数 */
40
41
42                                     /* -----事件标志管理 ----- */
43 #define OS_FLAG_EN             1    /* 允许 (1) 或者禁止 (0) 产生事件标志相关代码 */
44 #define OS_FLAG_WAIT_CLR_EN   1    /* 允许生成 Wait on Clear 事件标志代码 */
45 #define OS_FLAG_ACCEPT_EN     1    /* 允许生成 OSFlagAccept() */
46 #define OS_FLAG_DEL_EN        1    /* 允许生成 OSFlagDel() */
47 #define OS_FLAG_QUERY_EN      1    /* 允许生成 OSFlagQuery() */
48
49
50                                     /* -----消息邮箱管理 ----- */
51 #define OS_MBOX_EN             1    /* 允许 (1) 或者禁止 (0) 产生消息邮箱相关代码 */
52 #define OS_MBOX_ACCEPT_EN     1    /* 允许生成 OSMboxAccept() */
53 #define OS_MBOX_DEL_EN        1    /* 允许生成 OSMboxDel() */
54 #define OS_MBOX_POST_EN       1    /* 允许生成 OSMboxPost() */
55 #define OS_MBOX_POST_OPT_EN   1    /* 允许生成 OSMboxPostOpt() */
56 #define OS_MBOX_QUERY_EN      1    /* 允许生成 OSMboxQuery() */
57
58
59                                     /* -----内存管理 ----- */
60 #define OS_MEM_EN              1    /* 允许 (1) 或者禁止 (0) 产生内存相关代码 */
61 #define OS_MEM_QUERY_EN       1    /* 允许生成 OSMemQuery() */
62
63
64                                     /* -----互斥型信号量管理 ----- */
65 #define OS_MUTEX_EN           1    /* 允许 (1) 或者禁止 (0) 产生互斥型信号量相关代码 */
66 #define OS_MUTEX_ACCEPT_EN    1    /* 允许生成 OSMutexAccept() */
67 #define OS_MUTEX_DEL_EN       1    /* 允许生成 OSMutexDel() */
68 #define OS_MUTEX_QUERY_EN     1    /* 允许生成 OSMutexQuery() */
69
70
71                                     /* -----消息队列号管理 ----- */
72 #define OS_Q_EN                1    /* 允许 (1) 或者禁止 (0) 产生消息队列相关代码 */
73 #define OS_Q_ACCEPT_EN         1    /* 允许生成 OSQAccept() */
74 #define OS_Q_DEL_EN            1    /* 允许生成 OSQDel() */
75 #define OS_Q_FLUSH_EN         1    /* 允许生成 OSQFlush() */
76 #define OS_Q_POST_EN          1    /* 允许生成 OSQPost() */

```

```

77 #define OS_Q_POST_FRONT_EN      1    /* 允许生成 OSQPostFront()          */
78 #define OS_Q_POST_OPT_EN        1    /* 允许生成 OSQPostOpt()            */
79 #define OS_Q_QUERY_EN           1    /* 允许生成 OSQQuery()              */
80
81
82 /* -----信号管理----- */
83 #define OS_SEM_EN                1    /* 允许 (1) 或者禁止 (0) 产生信号量相关代码 */
84 #define OS_SEM_ACCEPT_EN        1    /* 允许生成 OSSemAccept()            */
85 #define OS_SEM_DEL_EN           1    /* 允许生成 OSSemDel()               */
86 #define OS_SEM_QUERY_EN         1    /* 允许生成 OSSemQuery()             */
87
88
89 /* -----任务管理----- */
90 #define OS_TASK_CHANGE_PRIO_EN  1    /* 允许生成 OSTaskChangePrio() 函数代码 */
91 #define OS_TASK_CREATE_EN       1    /* 允许生成 OSTaskCreate() 函数代码 */
92 #define OS_TASK_CREATE_EXT_EN   1    /* 允许生成 OSTaskCreateExt() 函数代码 */
93 #define OS_TASK_DEL_EN          1    /* 允许生成 OSTaskDel() 函数代码 */
94 #define OS_TASK_SUSPEND_EN      1    /* 允许生成 OSTaskSuspend() and OSTaskResume() 函数代码 */
95 #define OS_TASK_QUERY_EN        1    /* 允许生成 OSTaskQuery() 函数代码 */
96
97
98 /* -----时间管理----- */
99 #define OS_TIME_DLY_HMSM_EN     1    /* 允许生成OSTimeDlyHMSM() 函数代码 */
100 #define OS_TIME_DLY_RESUME_EN   1    /* 允许生成OSTimeDlyResume() 函数代码 */
101 #define OS_TIME_GET_SET_EN      1    /* 允许生成 OSTimeGet() 和 OSTimeSet() 函数代码 */
102
103
104 /* -----混合管理----- */
105 #define OS_SCHED_LOCK_EN        1    /* 允许生成 OSSchedLock() 和 OSSchedUnlock() 代码 */
106
107
108 #define OS_TICKS_PER_SEC        200  /* 设置每秒的节拍数目 */
109
110
111 typedef INT16U                  OS_FLAGS; /* 事件标志的数据类型 (8位, 16位 或 32 位) */
112
113 /*****结束*****/
114

```

```

1  /*
2  ****
3  *                                     uC/OS-II实时控制内核
4  *                                     主要的包含文件
5  *                                     内核管理文件
6  *
7  * 文   件: OS_CORE.C   内核结构管理文件
8  * 作   者: Jean J. Labrosse
9  * 中文注解: 钟常慰 zhongcw @ 126.com 译注版本: 1.0 请尊重原版内容
10 ****
11 */
12
13 #ifndef OS_MASTER_FILE           //如果没有定义OS_MASTER_FILE主文件, 则
14 #define OS_GLOBALS               //定义全局变量 OS_GLOBALS
15 #include "includes.h"           //包含"includes.h"文件, 部分C语言头文件的汇总打包文件
16 #endif                           //定义结束
17
18 /*
19 ****
20 *                                     MAPPING TABLE TO MAP BIT POSITION TO BIT MASK
21 *
22 * 注意: 变址索引表是对应OSRbyTbl[i]的位值(0~7), 给定值符合OSMapTbl[]的数据(二进制)
23 *       Index into table is desired bit position, 0..7
24 *       Indexed value corresponds to bit mask
25 ****
26 */
27 //OSMapTbl[]: 就绪任务表; 对应OSRdy Grp和OSRbyTbl[i]的位值(0~7)
28
29 INT8U const OSMapTbl[] = {0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80};
30
31 /*
32 ****
33 *                                     最高优先级任务查找表(PRIORITY RESOLUTION TABLE)
34 *
35 * 注意: 变址索引表是位模式, 找出就绪态最高优先级别任务, 给定值应符合高优先级位值(0~7)
36 *       Index into table is bit pattern to resolve highest priority
37 *       Indexed value corresponds to highest priority bit position (i.e. 0..7)
38 ****
39 */
40 //OSUnMapTbl[]: 最高优先级任务查找表; 对应OSRdy Grp和OSRbyTbl[i]的位值(0~7)
41
42 INT8U const OSUnMapTbl[] = {
43     0, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
44     4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
45     5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
46     4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
47     6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
48     4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
49     5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
50     4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
51     7, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
52     4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
53     5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
54     4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
55     6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
56     4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
57     5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
58     4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0
59 };
60 /*$PAGE*/
61 /*
62 ****
63 *                                     初始值(INITIALIZATION)
64 *
65 * 描述: 初始化uC/OS-II。对这个函数的调用必须在调用OSStart()函数之前。
66 *       OSStart()函数真正开始运行多任务时
67 * 意见: 无
68 * 返回: 无
69 ****
70 */
71
72 void OSInit (void)               //初始化UCOS-II函数
73 {
74     INT16U    i;                 //定义一个16位变量i
75     INT8U     *prdytbl;          //定义一个就绪态最高级任务列表指针

```

```

76     OS_TCB      *ptcb1;           //定义任务控制块优先级表指针1
77     OS_TCB      *ptcb2;           //定义任务控制块优先级表指针2
78 #if (OS_EVENT_EN > 0) && (OS_MAX_EVENTS > 1) //如果有消息事件, 并且最大消息事件值>1
79     OS_EVENT     *pevent1;        //定义事件指针1
80     OS_EVENT     *pevent2;        //定义事件指针2
81 #endif
82
83
84 #if OS_VERSION >= 204              //如果版本大于2.04版
85     OSInitHookBegin();            //调初始化钩子函数, 可加入用户代码
86 #endif
87
88 #if OS_TIME_GET_SET_EN > 0         //允许生成 OSTimeGet() 和 OSTimeSet() 函数代码
89     OSTime         = 0L;           //清除32的系统时钟
90 #endif
91     OSIntNesting   = 0;            //清除中断嵌套计数器
92     OSLockNesting  = 0;            //清除上锁嵌套计数器
93     OSTaskCtr      = 0;            //清除任务计数器
94     OSRunning      = FALSE;        //任务处于不运行状态
95     OSIdleCtr      = 0L;           //清除32位空闲任务的计数器
96                                     //允许生成OSTaskCreate() 函数和OSTaskCreateExt() 函数
97 #if (OS_TASK_STAT_EN > 0) && (OS_TASK_CREATE_EXT_EN > 0)
98     OSIdleCtrRun   = 0L;           //空闲任务计数器每秒的计数值清0
99     OSIdleCtrMax   = 0L;           //每秒空闲任务计数的最大值清0
100    OSStatRdy      = FALSE;        //统计任务是否就绪的标志为空
101 #endif
102    OSCtxSwCtr      = 0;            //上下文切换的次数(统计任务计数器)清0
103    OSRdyGrp        = 0x00;        //清除OSRdyTbl[i]组对应的任务就绪列表
104    prdytbl         = &OSRdyTbl[0];
105    for (i = 0; i < OS_RDY_TBL_SIZE; i++) {
106        *prdytbl++ = 0x00;        //所有的就绪列表指针内容全部清0
107    }
108
109    OSPrioCur      = 0;            //正在运行的任务的优先级
110    OSPrioHighRdy   = 0;            //具有最高优先级级别的就绪任务的优先级
111    OSTCBHighRdy    = (OS_TCB *)0; //指向最高级优先级就绪任务控制块的指针清0
112    OSTCBCur        = (OS_TCB *)0; //指向正在运行任务控制块的指针清0
113    OSTCBLst        = (OS_TCB *)0; //任务控制块链接表的指针清0
114                                     //清除所有的优先级控制块优先级列表e
115    for (i = 0; i < (OS_LOWEST_PRIO + 1); i++) {
116        OSTCBPrioTbl[i] = (OS_TCB *)0;
117    }
118    ptcb1 = &OSTCBTbl[0];          //查找任务控制块列表(0)的对应地址
119    ptcb2 = &OSTCBTbl[1];          //查找任务控制块列表(1)的对应地址
120                                     //释放所有的任务控制块列表
121    for (i = 0; i < (OS_MAX_TASKS + OS_N_SYS_TASKS - 1); i++) {
122        ptcb1->OSTCBNext = ptcb2;
123        ptcb1++;
124        ptcb2++;
125    }
126    ptcb1->OSTCBNext = (OS_TCB *)0; //将最后的任务块双向链接表的后链接为0
127    OSTCBFreeList    = &OSTCBTbl[0]; //空任务控制块地址为当前任务控制块列表的首地址
128
129 #if (OS_EVENT_EN > 0) && (OS_MAX_EVENTS > 0) //如果有消息事件, 并且最大消息事件数>0
130     #if OS_MAX_EVENTS == 1                //如果最大消息事件数>1
131         //只能拥有单独的一个消息事件
132         OSEventFreeList = &OSEventTbl[0]; //空余事件管理列表=任务等待表首地址
133         OSEventFreeList->OSEventType = OS_EVENT_TYPE_UNUSED; //事件的类型=空闲
134         OSEventFreeList->OSEventPtr  = (OS_EVENT *)0;        //消息或消息队列的指针为空
135     #else
136         pevent1 = &OSEventTbl[0]; //查找任务等待表(0)对应首地址
137         pevent2 = &OSEventTbl[1]; //查找任务等待表(1)对应地址
138         //释放所有的任务等待表, 并将事件的类型=空闲
139         for (i = 0; i < (OS_MAX_EVENTS - 1); i++) {
140             pevent1->OSEventType = OS_EVENT_TYPE_UNUSED;
141             pevent1->OSEventPtr  = pevent2;
142             pevent1++;
143             pevent2++;
144         }
145         pevent1->OSEventType = OS_EVENT_TYPE_UNUSED; //首地址的事件的类型=空闲
146         pevent1->OSEventPtr  = (OS_EVENT *)0;        //首地址的消息或消息队列的指针为空
147         OSEventFreeList     = &OSEventTbl[0];        //空余事件管理列表=任务等待表首地址
148     #endif
149 #endif
150 //条件编译: UCOS版本>= 251 且 OS_FLAG_EN 允许产生事件标志程序代码 且 最大事件标志>0
151 #if (OS_VERSION >= 251) && (OS_FLAG_EN > 0) && (OS_MAX_FLAGS > 0)

```

```

152     OS_FlagInit(); //初始化事件标志结构
153 #endif
154 //条件编译: OS_Q_EN 允许 (1)产生消息队列相关代码 并且 应用中最多对列控制块的数目 > 0
155 #if (OS_Q_EN > 0) && (OS_MAX_QS > 0)
156     OS_QInit(); //初始化事件队列结构
157 #endif
158 //条件编译: 若两个条件满足时, 产生以下代码
159 //OS_MEM_EN允许 (1) 或者禁止 (0) 产生内存相关代码
160 //OS_MAX_MEM_PART 最多内存块的数目
161 #if (OS_MEM_EN > 0) && (OS_MAX_MEM_PART > 0)
162     OS_MemInit(); //初始化内存块结构
163 #endif
164
165 /* ----- 产生一个空闲的任务 (CREATION OF 'IDLE' TASK) ----- */
166 #if OS_TASK_CREATE_EXT_EN > 0 // 允许生成OSTaskCreateExt() 函数
167     #if OS_STK_GROWTH == 1 // 堆栈生长方向向下
168         // 建立扩展任务[...]
169         (void)OSTaskCreateExt(OS_TaskIdle, // 空闲任务
170             (void *)0, // 没有(传递参数指针)
171             &OSTaskIdleStk[OS_TASK_IDLE_STK_SIZE - 1], // 分配任务堆栈栈顶指针
172             OS_IDLE_PRIO, // 分配任务优先级
173             OS_TASK_IDLE_ID, // (未来的)优先级标识(与优先级相同)
174             &OSTaskIdleStk[0], // 分配任务堆栈栈底指针
175             OS_TASK_IDLE_STK_SIZE, // 指定堆栈的容量(检验用)
176             (void *)0, // 没有(指向用户附加的数据域的指针)
177             OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);
178     #else // 建立扩展任务[...] //堆栈生长方向向上
179         (void)OSTaskCreateExt(OS_TaskIdle, // 空闲任务
180             (void *)0, // 没有(传递参数指针)
181             &OSTaskIdleStk[0], // 分配任务堆栈栈底指针
182             OS_IDLE_PRIO, // 分配任务优先级
183             OS_TASK_IDLE_ID, // (未来的)优先级标识(与优先级相同)
184             &OSTaskIdleStk[OS_TASK_IDLE_STK_SIZE - 1], // 分配任务堆栈栈顶指针
185             OS_TASK_IDLE_STK_SIZE, // 指定堆栈的容量(检验用)
186             (void *)0, // 没有(指向用户附加的数据域的指针)
187             OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);
188     #endif
189 #else // 否则只能生成OSTaskCreate() 函数
190     #if OS_STK_GROWTH == 1 // 堆栈生长方向向下
191         // 建立任务[空闲任务、
192         (void)OSTaskCreate(OS_TaskIdle, // 没有(传递参数指针)
193             (void *)0, // 分配任务堆栈栈底指针
194             &OSTaskIdleStk[OS_TASK_IDLE_STK_SIZE - 1], // 分配任务堆栈栈顶指针
195             OS_IDLE_PRIO); // 分配任务优先级
196     #else // 否则堆栈生长方向向上
197         // 建立任务[空闲任务、
198         (void)OSTaskCreate(OS_TaskIdle, // 没有(传递参数指针)
199             (void *)0, // 分配任务堆栈栈底指针
200             &OSTaskIdleStk[0], // 分配任务堆栈栈顶指针
201             OS_IDLE_PRIO); // 分配任务优先级
202     #endif
203 #endif
204 #endif
205
206 /* ----- 产生一个统计任务 (CREATION OF 'STATISTIC' TASK) ----- */
207 #if OS_TASK_STAT_EN > 0
208     #if OS_TASK_CREATE_EXT_EN > 0 // 允许生成OSTaskCreateExt() 函数
209         #if OS_STK_GROWTH == 1 // 堆栈生长方向向下
210             // 建立扩展任务[...]
211             (void)OSTaskCreateExt(OS_TaskStat, // 产生一个统计任务
212                 (void *)0, // 没有(传递参数指针)
213                 &OSTaskStatStk[OS_TASK_STAT_STK_SIZE - 1], // 分配任务堆栈栈顶指针
214                 OS_STAT_PRIO, // 分配任务优先级
215                 OS_TASK_STAT_ID, // (未来的)优先级标识(与优先级相同)
216                 &OSTaskStatStk[0], // 分配任务堆栈栈底指针
217                 OS_TASK_STAT_STK_SIZE, // 指定堆栈的容量(检验用)
218                 (void *)0, // 没有(指向用户附加的数据域的指针)
219                 OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);
220         #else // 建立扩展任务[...] //堆栈生长方向向上
221             (void)OSTaskCreateExt(OS_TaskStat, // 产生一个统计任务
222                 (void *)0, // 没有(传递参数指针)
223                 &OSTaskStatStk[0], // 分配任务堆栈栈底指针
224                 OS_STAT_PRIO, // 分配任务优先级
225                 OS_TASK_STAT_ID, // (未来的)优先级标识(与优先级相同)
226                 &OSTaskStatStk[OS_TASK_STAT_STK_SIZE - 1], // 分配任务堆栈栈顶指针
227                 OS_TASK_STAT_STK_SIZE, // 指定堆栈的容量(检验用)
228                 (void *)0, // 没有(指向用户附加的数据域的指针)
229                 OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);
230         #endif
231     #else // 否则只能生成OSTaskCreate() 函数
232         #if OS_STK_GROWTH == 1 // 堆栈生长方向向下
233             // 建立任务[统计任务、
234             (void)OSTaskCreate(OS_TaskStat, // 没有(传递参数指针)
235                 (void *)0, // 分配任务堆栈栈底指针
236                 &OSTaskStatStk[OS_TASK_STAT_STK_SIZE - 1], // 分配任务堆栈栈顶指针
237                 OS_STAT_PRIO); // 分配任务优先级
238         #else // 否则堆栈生长方向向上
239             // 建立任务[统计任务、
240             (void)OSTaskCreate(OS_TaskStat, // 没有(传递参数指针)
241                 (void *)0, // 分配任务堆栈栈底指针
242                 &OSTaskStatStk[0], // 分配任务堆栈栈顶指针
243                 OS_STAT_PRIO); // 分配任务优先级
244         #endif
245     #endif
246 #endif

```



```

228         OS_TASK_STAT_STK_SIZE,           // 指定堆栈的容量(检验用
229         (void *)0,                       // 没有(指向用户附加的数据域的指针)
230         OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);
231     #endif
232     #else                                 // 否则只能生成OSTaskCreate() 函数
233     #if OS_STK_GROWTH == 1                // 堆栈生长方向向下
234         (void)OSTaskCreate(OS_TaskStat,   // 产生一个统计任务
235                             (void *)0,    // 没有(传递参数指针)
236                             &OSTaskStatStk[OS_TASK_STAT_STK_SIZE - 1], // 分配任务堆栈栈顶指针
237                             OS_STAT_PRIO); // 分配任务优先级
238     #else                                 // 否则堆栈生长方向向上
239         (void)OSTaskCreate(OS_TaskStat,   // 产生一个统计任务
240                             (void *)0,    // 没有(传递参数指针)
241                             &OSTaskStatStk[0], // 分配任务堆栈栈底指针
242                             OS_STAT_PRIO); // 分配任务优先级
243     #endif
244 #endif
245 #endif
246
247 #if OS_VERSION >= 204                    // 判断版本是否是大于或等于2.41版
248     OSInitHookEnd();                    // 调用OSInitHookEnd() 钩子程序
249 #endif
250 }
251 /*$PAGE*/
252 /*
253 ****
254 *                                     中断函数正在执行 ENTER ISR
255 *
256 * 描述: 通知uC/OS-II, 一个中断处理函数正在进行, 这有助于uC/OS-II掌握中断嵌套情况。
257 *       OSIntEnter ()和OSIntExit (函数联合使用), 调用者, 只能在中断程序中。
258 *
259 * 参数: 无
260 *
261 * 返回: 无
262 *
263 * 注意: 1) 在任务级不能调用该函数
264 *       2) 如果系统使用的处理器能够执行自动的独立执行读取一修改一写入的操作, 那么就可以直接递增
265 *          中断嵌套层数(OSIntNesting), 这样可以避免调用函数所带来的额外开销。在中断服务子程序中
266 *          给OSIntNesting加1是不会有问题的, 因为给OSIntNesting加1时, 中断是关闭的
267 *       3) 中断嵌套深度可达255
268 ****
269 */
270 void OSIntEnter (void)                  // 中断函数正在执行()
271 {
272     #if OS_CRITICAL_METHOD == 3        // 中断函数被设定为模式3
273         OS_CPU_SR cpu_sr;
274     #endif
275
276     OS_ENTER_CRITICAL();               // 关闭中断
277     if (OSIntNesting < 255) {          // 如果中断嵌套小于255
278         OSIntNesting++;                // 中断嵌套计数变量加1
279     }
280     OS_EXIT_CRITICAL();                 // 打开中断
281 }
282 /*$PAGE*/
283 /*
284 ****
285 ****
286 *                                     中断函数已经完成 EXIT ISR
287 *
288 * 描述: 通知uC/OS-II, 一个中断服务已经执行完成, 这有助于uC/OS-II掌握中断嵌套的情况。通常
289 *       OSIntExit ()和OSIntEnter ()联合使用。当最后一层嵌套的中断执行完毕时, 如果有更高优先级任
290 *       务准备就绪, 则uC/OS-II会调用任务调度函数。在这种情况下, 中断返回到更高优先级的任务, 而
291 *       不是被中断了的任务。调用者, 只能在中断程序中。
292 *
293 * 参数: 无
294 *
295 * 返回: 无
296 *
297 * 注意: 1) 在任务级不能调用该函数, 并且即使没有调用OSIntEnter() 函数, 而是使用直接递增
298 *       OSIntNesting的方法, 也必须调用OSIntExit()。
299 *       2) 给调度器上锁用于禁止任务调度 (查看 OSSchedLock() 函数)
300 ****
301 */
302
303 void OSIntExit (void)                  // 脱离中断函数

```

```

304 {
305 #if OS_CRITICAL_METHOD == 3           //中断函数被设定为模式3
306     OS_CPU_SR cpu_sr;
307 #endif
308
309
310     OS_ENTER_CRITICAL();               //关闭中断
311     if (OSIntNesting > 0) {           //如果中断嵌套大于0
312         OSIntNesting--;               //中断嵌套计数变量减1
313     }
314     //1) 中断嵌套层数计数器和锁定嵌套计数器(OSLockNesting)二者都必须是零
315     //2) OSRdyTbl[]所需的检索值Y是保存在全程变量OSIntExitY中
316     //3) 检查具有最高优先级级别的就绪任务的优先级是否是正在运行的任务的优先级
317     //4) 将任务控制块优先级表保存到指向最高级优先级就绪任务控制块的指针
318     //5) 上下文切换的次数(统计任务计数器)
319     //6) 做中断任务切换
320     if ((OSIntNesting == 0) && (OSLockNesting == 0)) { //1)
321         OSIntExitY = OSUnMapTbl[OSRdyGrp];           //2)
322         OSPrioHighRdy = (INT8U)((OSIntExitY << 3) + OSUnMapTbl[OSRdyTbl[OSIntExitY]]);
323         if (OSPriloHighRdy != OSPrioCur) {           //3)
324             OSTCBHighRdy = OSTCBPrioTbl[OSPriloHighRdy]; //4)
325             OSCtxSwCtr++;                               //5)
326             OSIntCtxSw();                               //6)
327         }
328     }
329     OS_EXIT_CRITICAL();           //打开中断
330 }
331 /*$PAGE*/
332 /*
333 ****
334 *                                     给调度器上锁 PREVENT SCHEDULING
335 *
336 * 描述: 本函数用于禁止任务调度, 直到任务完成后调用给调度器开锁函数OSSchedUnlock()为止。调用
337 *  OSSchedlock()的任务保持对CPU的控制权, 尽管有个优先级更高的任务进入了就绪态。然而, 此时
338 *  中断是可以被识别的, 中断服务也能得到(假设中断是开着的)。OSSchedlock()和OSSchedUnlock()
339 *  必须成对使用。变量OSLockNesting跟踪OSSchedLock()函数被调用的次数, 以允许嵌套的函数包含临
340 *  界段代码, 这段代码其它任务不得干预。uC/OS-II允许嵌套深度达255层。当OSLockNesting等于零时,
341 *  调度重新得到允许。函数OSSchedLock()和OSSchedUnlock()的使用要非常谨慎, 因为它们影响uC/OS-II
342 *  对任务的正常管理。
343 * 说明: 当OSLockNesting减到零的时候, OSSchedUnlock()调用OSSched。OSSchedUnlock()是被某任务调用的,
344 *  在调度器上锁的期间, 可能有什么事件发生了并使一个更高优先级的任务进入就绪态。
345 *
346 * 参数: 无
347 *
348 * 返回: 无
349 *
350 * 警告: 不得调用把当前任务挂起的程序
351 *
352 * 注意: 1) 调用OSSchedLock()以后, 用户的应用程序不得使用任何能将现行任务挂起的系统调用。也就说,
353 *  用户程序不得调用OSMboxPend()、OSQPend()、OSSemPend()、OSTaskSuspend(OS_PRIO_SELF)、
354 *  OSTimeDly()或OSTimeDlyHMSM(), 直到OSLockNesting回零为止。因为调度器上了锁, 用户就锁住
355 *  了系统, 任何其它任务都不能运行。
356 * 2) 当低优先级的任务要发消息给多任务的邮箱、消息队列、信号量时, 用户不希望高优先级的任
357 *  务在邮箱、队列和信号量没有得到消息之前就取得了CPU的控制权, 此时, 用户可以使用禁止
358 *  调度器函数。
359 ****
360 */
361
362 #if OS_SCHED_LOCK_EN > 0           //允许生产OSSchedLock()函数
363 void OSSchedLock (void)           //给调度器上锁函数
364 {
365 #if OS_CRITICAL_METHOD == 3           //中断函数被设定为模式3
366     OS_CPU_SR cpu_sr;
367 #endif
368
369
370     if (OSRunning == TRUE) {         //如果有多个任务在期待
371         OS_ENTER_CRITICAL();         //关闭中断
372         if (OSLockNesting < 255) {   //上锁嵌套是否大于255
373             OSLockNesting++;         //给上锁嵌套加1
374         }
375         OS_EXIT_CRITICAL();           //打开中断
376     }
377 }
378 #endif
379

```

```

380 /*$PAGE*/
381 /*
382 ****
383 *                                     给调度器解锁  ENABLE SCHEDULING
384 *
385 * 描述: 本函数用于解禁任务调度
386 *
387 * 参数: 无
388 *
389 * 返回: 无
390 *
391 * 注意: 1) OSSchedlock()和OSSchedUnlock()必须成对使用, 在使用OSSchedUnlock()函数之前必须使
392 *       用OSSchedLock()函数
393 ****
394 */
395
396 #if OS_SCHED_LOCK_EN > 0                //允许生产OSSchedUnlock()函数
397 void OSSchedUnlock (void)              //给调度器解锁函数
398 {
399     #if OS_CRITICAL_METHOD == 3        //中断函数被设定为模式3
400         OS_CPU_SR  cpu_sr;
401     #endif
402
403
404     if (OSRunning == TRUE) {           //如果有多个任务在期待
405         OS_ENTER_CRITICAL();           //关闭中断
406         if (OSLockNesting > 0) {        //上锁嵌套是否大于0
407             OSLockNesting--;            //给上锁嵌套减1
408             //如果函数不是在中断服务子程序中调用的, 且调度允许的,
409             if ((OSLockNesting == 0) && (OSIntNesting == 0)) {
410
411                 OS_EXIT_CRITICAL();     //打开中断
412                 OS_Sched();              //进入任务调度
413             } else {
414                 OS_EXIT_CRITICAL();     //打开中断
415             }
416         } else {
417             OS_EXIT_CRITICAL();         //打开中断
418         }
419     }
420 }
421 #endif
422
423 /*$PAGE*/
424 /*
425 ****
426 *                                     启动多个任务  START MULTITASKING
427 *
428 * 描述: 当调用OSSStart()时, OSSStart()从任务就绪表中找出那个用户建立的优先级最高任务的任务控制
429 *       块。然后, OSSStart()调用高优先级就绪任务启动函数OSSStartHighRdy(), (见汇编语言文件
430 *       OS_CPU_A.ASM), 这个文件与选择的微处理器有关。实质上, 函数OSSStartHighRdy()是将任务栈中
431 *       保存的值弹回到CPU寄存器中, 然后执行一条中断返回指令, 中断返回指令强制执行该任务代码。
432 *       高优先级就绪任务启动函数OSSStartHighRdy()。
433 *
434 * 参数: 无
435 *
436 * 返回: 无
437 *
438 * 注意: OSSStartHighRdy() 必须:
439 *       a) OSRunning为真, 指出多任务已经开始
440 *       b) 启动uC/OS-II之前, 至少必须建立一个应用任务
441 *       c) OSSStartHighRdy()将永远不返回到OSSStart()
442 ****
443 */
444
445 void OSSStart (void)                    //启动多个任务
446 {
447     INT8U y;
448     INT8U x;
449
450
451     if (OSRunning == FALSE) {           //OSRunning已设为“真”, 指出多任务已经开始
452         y = OSUnMapTbl[OSRdyGrp];      //查找最高优先级任务号码
453         x = OSUnMapTbl[OSRdyTbl[y]];
454         OSPrioHighRdy = (INT8U)((y << 3) + x); //找出就绪态最高级任务控制块
455         OSPrioCur = OSPrioHighRdy;

```

```

456 //OSPrIoCur和OSPrIoHighRdy存放的是用户应用任务的优先级
457 OSTCBHighRdy = OSTCBPrIoTbl[OSPrIoHighRdy];
458 OSTCBCur = OSTCBHighRdy;
459 OSStartHighRdy(); //调用高优先级就绪任务启动函数
460 }
461 }
462 /*$PAGE*/
463 /*
464 ****
465 * 统计任务初始化 STATISTICS INITIALIZATION
466 *
467 * 描述: 统计初始化函数OSStatInit() 决定在没有其它应用任务运行时, 空闲计数器(OSIdleCtr)的计数
468 * 有多快。这个任务每秒执行一次, 以确定所有应用程序中的任务消耗了多少CPU时间。当用户的
469 * 应用程序代码加入以后, 运行空闲任务的CPU时间就少了, OSIdleCtr就不会像原来什么任务都不
470 * 运行时那么多计数。要知道, OSIdleCtr的最大计数值是OSStatInit() 在初始化时保存在计数
471 * 器最大值OSIdleCtrMax中的。CPU利用率:
472 *
473 * 空闲计数值OSIdleCtr
474 * CPU 使用率Usage (%) = 100 * (1 - -----)
475 * 设定最大空闲计数值OSIdleCtrMax
476 *
477 * 参数: 无
478 *
479 * 返回: 无
480 ****
481 */
482
483 #if OS_TASK_STAT_EN > 0 //允许生产OSStatInit() 函数
484 void OSStatInit (void) //统计任务初始化
485 {
486 #if OS_CRITICAL_METHOD == 3 //中断函数被设定为模式3
487 OS_CPU_SR cpu_sr;
488 #endif
489
490 OSTimeDly(2); //调用延迟函数OSTimeDly() 将自身延时2个时钟节拍以停止自身的运行
491 //这是为了使OSStatInit() 与时钟节拍同步
492 OS_ENTER_CRITICAL(); //关闭中断
493 OSIdleCtr = 0L; //执行OSStartInit() 时, 空闲计数器OSIdleCtr被清零
494 OS_EXIT_CRITICAL(); //打开中断
495 OSTimeDly(OS_TICKS_PER_SEC); //将自身延时整整一秒
496 //((因为没有其它进入就绪态的任务, OSTaskIdle() 又获得了CPU的控制权)
497 OS_ENTER_CRITICAL(); //关闭中断
498 OSIdleCtrMax = OSIdleCtr; //空闲计数器将1秒钟内计数的值存入空闲计数器最大值OSIdleCtrMax中
499 OSStatRdy = TRUE; //将统计任务就绪标志OSStatRdy设为“真”, 以此来允许两个时钟节拍
500 //以后OSTaskStat() 开始计算CPU的利用率
501 OS_EXIT_CRITICAL(); //打开中断
502 }
503 #endif
504 /*$PAGE*/
505 /*
506 ****
507 * 时钟节拍函数 PROCESS SYSTEM TICK
508 *
509 *
510 * 描述: uC/OS需要用户提供周期性信号源, 用于实现时间延时和确认超时。节拍率应在每秒10次到100次
511 * 之间, 或者说10到100Hz。时钟节拍率越高, 系统的额外负荷就越重。时钟节拍的频率取决于
512 * 用户应用程序的精度。时钟节拍源可以是专门的硬件定时器, 也可以是来自50/60Hz交流电源的
513 * 信号
514 *
515 * 参数: 无
516 *
517 * 返回: 无
518 ****
519 */
520
521 void OSTimeTick (void) //时钟节拍函数
522 {
523 #if OS_CRITICAL_METHOD == 3 //中断函数被设定为模式3
524 OS_CPU_SR cpu_sr;
525 #endif
526 OS_TCB *ptcb; //定义任务控制块优先级表变量
527
528 OSTimeTickHook(); //调用用户·自定义函数(钩子函数)
529 #if OS_TIME_GET_SET_EN > 0 //允许生成OSTimeGet() 函数代码
530 OS_ENTER_CRITICAL(); //关闭中断
531 OSTime++; //累加从开机以来的时间, 用的是一个无符号32位变量

```

```

532     OS_EXIT_CRITICAL(); //打开中断
533 #endif
534     ptcB = OSTCBLIST; //保存任务控制块列表首地址
535
536     //从OSTCBLIST开始, 沿着OS_TCB链表做, 一直做到空闲任务
537     while (ptcB->OSTCBPrio != OS_IDLE_PRIO) {
538         OS_ENTER_CRITICAL(); //关闭中断
539         if (ptcB->OSTCBDly != 0) { //如果任务等待时的最多节拍数不为0
540             if (--ptcB->OSTCBDly == 0) { //如果任务等待时的最多节拍数为0
541
542                 //而确切被任务挂起的函数OSTaskSuspend()挂起的任务则不会进入就绪态
543                 //执行时间直接与应用程序中建立了多少个任务成正比
544                 if ((ptcB->OSTCBStat & OS_STAT_SUSPEND) == 0x00) {
545                     //当某任务的OS任务控制块中的时间延时项OSTCBDly减到了零, 这个任务就进入了就绪态
546                     OSRdyGrp |= ptcB->OSTCBBitY;
547                     OSRdyTbl[ptcB->OSTCBY] |= ptcB->OSTCBBitX;
548                 } else {
549                     ptcB->OSTCBDly = 1; //否则
550                 } //允许任务等待时的最多节拍数为1
551             }
552         }
553         ptcB = ptcB->OSTCBNext; //指向任务块双向链接表的后链接
554         OS_EXIT_CRITICAL(); //打开中断
555     }
556 }
557 /*$PAGE*/
558 /*
559 *****
560 * 获得版本号GET VERSION
561 *
562 * 描述: 这个函数是返回一个uC/OS-II的版本值. 这个返回值乘100是uC/OS-II的版本号. 也就是版本2.00
563 *       would be returned as 200.
564 *
565 * 参数: 无
566 *
567 * 返回: uC/OS-II的版本号除以100.
568 *****
569 */
570
571 INT16U OSVersion (void) //获得版本号
572 {
573     return (OS_VERSION); //返回版本值
574 }
575
576 /*$PAGE*/
577 /*
578 *****
579 * 虚拟函数 DUMMY FUNCTION
580 *
581 * 描述: 这个函数不做任务工作. 它是随便访问OSTaskDel()函数.
582 *
583 * 参数: 无
584 *
585 * 返回: 无
586 *****
587 */
588
589 #if OS_TASK_DEL_EN > 0 //允许生成 OS_Dummy() 函数
590 void OS_Dummy (void) //建立一个虚拟函数
591 { //不做任何工作
592 }
593 #endif
594
595 /*$PAGE*/
596 /*
597 *****
598 * 使一个任务进入就绪态 MAKE TASK READY TO RUN BASED ON EVENT OCCURRING
599 *
600 * 描述: 当发生了某个事件, 该事件等待任务列表中的最高优先级任务(HPT)要置于就绪态时, 该事件对应
601 *       的OSSemPost(), OSMBxPost(), OSQPost(), 和OSQPostFront() 函数调用OSEventTaskRdy() 实现
602 *       该操作. 换句话说, 该函数从等待任务队列中删除HPT任务, 并把该任务置于就绪态
603 *
604 * 参数: pEvent is a pointer to the event control block corresponding to the event.
605 *
606 *       msg is a pointer to a message. This pointer is used by message oriented services
607 *       such as MAILBOXes and QUEUES. The pointer is not used when called by other

```

```

608 *          service functions.
609 *
610 *          msk is a mask that is used to clear the status byte of the TCB. For example,
611 *          OSSemPost() will pass OS_STAT_SEM, OSMboxPost() will pass OS_STAT_MBOX etc.
612 *
613 * 返回: 无
614 *
615 * 注意: 这个函数是uC/OS-II内部函数, 你不可在应用程序调用它, 调用此函数也应当关闭中断
616 *****/
617 */
618 #if OS_EVENT_EN > 0                                     //各类消息事件是否允许
619 //使一个任务进入就绪态
620 INT8U OS_EventTaskRdy (OS_EVENT *pevent, void *msg, INT8U msk)
621 {
622     OS_TCB *ptcb;
623     INT8U x;
624     INT8U y;
625     INT8U bitx;
626     INT8U bity;
627     INT8U prio;
628
629     //1) 首先计算HPT任务在.OSEventTbl[]中的字节索引, 其结果是一个从0到OS_LOWEST_PRIO/8+1之间的数
630     //2) 并利用该索引得到该优先级任务在.OSEventGrp中的位屏蔽码
631     //3) 判断HPT任务在.OSEventTbl[]中相应位的位置
632     //4) 其结果是一个从0到OS_LOWEST_PRIO/8+1之间的数, 以及相应的位屏蔽码
633     //5) 根据以上结果, OSEventTaskRdy() 函数计算出HPT任务的优先级
634     //6) 然后就可以从等待任务列表中删除该任务了
635     y = OSUnMapTbl[pevent->OSEventGrp];           //1)
636     bity = OSMMapTbl[y];                           //2)
637     x = OSUnMapTbl[pevent->OSEventTbl[y]];         //3)
638     bitx = OSMMapTbl[x];                           //4)
639     prio = (INT8U)((y << 3) + x);                   //5)
640     if ((pevent->OSEventTbl[y] & ~bitx) == 0x00) { //6)
641         pevent->OSEventGrp &= ~bity;
642     }
643     //7) 任务的TCB中包含有需要改变的信息。知道了HPT任务的优先级, 就可得到指向该任务的TCB的指针
644     //8) 因为最高优先级任务运行条件已经得到满足, 必须停止OSTimeTick() 函数对.OSTCDBly域的递减操作,
645     // 所以OSEventTaskRdy() 直接将该域清0
646     //9) 因为该任务不再等待该事件的发生, 所以本函数将其任务控制块中指向事件控制块的指针指向NULL
647     //10) 如果OSEventTaskRdy() 是由OSMboxPost() 或者OSQPost() 调用的, 该函数还要将相应的消息传递给
648     // HPT, 放在它的任务控制块中
649     ptcb = OSTCBPrioTbl[prio];                     //7)
650     ptcb->OSTCDBly = 0;                             //8)
651     ptcb->OSTCBEventPtr = (OS_EVENT *)0;            //9)
652     #if ((OS_Q_EN > 0) && (OS_MAX_QS > 0)) || (OS_MBOX_EN > 0)
653     ptcb->OSTCBMsg = msg;                           //10)
654     #else
655     msg = msg;
656     #endif
657     //11) 当OSEventTaskRdy() 被调用时, 位屏蔽码msk作为参数传递给它。该参数是用于对任务控制块中的
658     // 位清零的位屏蔽码, 和所发生事件的类型相对应
659     //12) 根据.OSTCBStat判断该任务是否已处于就绪状态
660     //13) 如果是, 则将HPT插入到uC/OS-II的就绪任务列表中。注意, HPT任务得到该事件后不一定进入就绪
661     // 状态, 也许该任务已经由于其它原因挂起了
662     ptcb->OSTCBStat &= ~msk;                         //11)
663     if (ptcb->OSTCBStat == OS_STAT_RDY) {             //12)
664         OSRdyGrp |= bity;                             //13)
665         OSRdyTbl[y] |= bitx;                          //返回就绪态任务的优先级
666     }
667     return (prio);
668 }
669 #endif
670 /*$PAGE*/
671 /*
672 *****/
673 *          使一个任务进入等待某事件发生状态 MAKE TASK WAIT FOR EVENT TO OCCUR
674 *
675 * 描述: 当某个任务须等待一个事件的发生时, 信号量、互斥型信号量、邮箱以及消息队列会通过相应的
676 * PEND函数调用本函数, 使当前任务从就绪任务表中脱离就绪态, 并放到相应的事件控制块ECB的等
677 * 待任务表中
678 *
679 * 参数: pevent 分配给事件控制块的指针, 为等待某事件发生的任务
680 *
681 * 返回: 无
682 *
683 * 注意: 这个函数是uC/OS-II内部函数, 你不可在应用程序中调用它, 调用OS_EventT0() 也应当关闭中断

```



```

684 *****/
685 */
686 #if OS_EVENT_EN > 0                                //各类消息事件是否允许
687 void OS_EventTaskWait (OS_EVENT *pevent)            //使一个任务进入等待某事件发生状态 (ECB指针)
688 { //将指向事件控制块ECB的指针放到任务的任务控制块TCB中, 建立任务与事件控制块ECB之间的链接
689     OSTCBCur->OSTCBEventPtr = pevent;
690     //将任务从就绪任务表中删除
691     if ((OSRdyTbl[OSTCBCur->OSTCBy] & ~OSTCBCur->OSTCBBitX) == 0x00) {
692         OSRdyGrp &= ~OSTCBCur->OSTCBBitY;
693     }
694     //把该任务放到事件控制块ECB的等待事件的任务列表中
695     pevent->OSEventTbl[OSTCBCur->OSTCBy] |= OSTCBCur->OSTCBBitX;
696     pevent->OSEventGrp |= OSTCBCur->OSTCBBitY;
697 }
698 #endif
699 /*$PAGE*/
700 /*
701 *****/
702 *           由于超时而将任务置为就绪态 MAKE TASK READY TO RUN BASED ON EVENT TIMEOUT
703 *
704 * 描述: 如果在预先指定的等待时限内任务等待的事件没有发生, 那么本函数会因为等待超时而将任务的
705 *       状态置为就绪态。在这种情况下, 信号量、互斥型信号量、邮箱以及消息队列会通过PEND函数调用
706 *       用本函数, 以完成这项工作
707 *
708 * 参数: pevent 分配给事件控制块的指针, 为超时就绪态的任务
709 *
710 * 返回: 无
711 *
712 * 注意: 这个函数是uC/OS-II内部函数, 你不可在应用程序中调用它, 调用OS_EventT0()也应当关闭中断
713 *****/
714 */
715 #if OS_EVENT_EN > 0                                //消息事件是否 > 0
716 void OS_EventT0 (OS_EVENT *pevent)                 //由于超时而将任务置为就绪态 (ECB指针)
717 { //本函数必须从事件控制块ECB中等待任务列表中将该任务删除
718     if ((pevent->OSEventTbl[OSTCBCur->OSTCBy] & ~OSTCBCur->OSTCBBitX) == 0x00) {
719         pevent->OSEventGrp &= ~OSTCBCur->OSTCBBitY;
720     }
721     OSTCBCur->OSTCBStat = OS_STAT_RDY; //该任务被置为就绪态
722     OSTCBCur->OSTCBEventPtr = (OS_EVENT *)0; //从任务控制块TCB中将事件控制块ECB的指针删除
723 }
724 #endif
725 /*$PAGE*/
726 /*
727 *****/
728 *           事件控制块列表初始化 INITIALIZE EVENT CONTROL BLOCK'S WAIT LIST
729 *
730 * 描述: 当建立一个信号量、邮箱或者消息队列时, 相应的建立函数OSSemInit(), OSMboxCreate(), 或者
731 *       OSQCreate()通过调用OSEventWaitListInit()对事件控制块中的等待任务列表进行初始化。该函数
732 *       初始化一个空的等待任务列表, 其中没有任何任务。该函数的调用参数只有一个, 就是指向需要初
733 *       始化的事件控制块的指针pevent。
734 *
735 * 参数: pevent 传递一个指针给事件控制块, 该指针变量就是创建信号量、互斥型信号量、邮箱或消息队
736 *       列时分配的事件控制块的指针
737 *
738 * 返回: 无
739 *
740 * 注意: 这个函数是uC/OS-II内部函数, 你不可调用它。
741 *****/
742 */
743 #if ((OS_Q_EN > 0) && (OS_MAX_QS > 0)) || (OS_MBOX_EN > 0) || (OS_SEM_EN > 0) || (OS_MUTEX_EN > 0)
744 //当以上各种事件允许时
745 void OS_EventWaitListInit (OS_EVENT *pevent)        //事件控制块列表初始化(事件控制块指针)
746 {
747     INT8U *ptbl; //地址指针变量
748
749     pevent->OSEventGrp = 0x00; //等待任务所在的组为空
750     ptbl = &pevent->OSEventTbl[0]; //保存任务等待表首地址
751                                     //OSEventTbl[] (共8个字节OSMapTbl[prio&0x07])
752     //等待任务列表字节>0
753 #if OS_EVENT_TBL_SIZE > 0
754     *ptbl++ = 0x00; //OSEventTbl[0]清空
755 #endif
756
757     //等待任务列表字节>1
758 #if OS_EVENT_TBL_SIZE > 1
759     *ptbl++ = 0x00; //OSEventTbl[1]清空
760 #endif
761 #endif

```

```

760
761 #if OS_EVENT_TBL_SIZE > 2                //等待任务列表字节>2
762     *ptbl++ = 0x00;                      //OSEventTbl[3]清空
763 #endif
764
765 #if OS_EVENT_TBL_SIZE > 3                //等待任务列表字节>3
766     *ptbl++ = 0x00;                      //OSEventTbl[3]清空
767 #endif
768
769 #if OS_EVENT_TBL_SIZE > 4                //等待任务列表字节>4
770     *ptbl++ = 0x00;                      //OSEventTbl[4]清空
771 #endif
772
773 #if OS_EVENT_TBL_SIZE > 5                //等待任务列表字节>5
774     *ptbl++ = 0x00;                      //OSEventTbl[5]清空
775 #endif
776
777 #if OS_EVENT_TBL_SIZE > 6                //等待任务列表字节>6
778     *ptbl++ = 0x00;                      //OSEventTbl[6]清空
779 #endif
780
781 #if OS_EVENT_TBL_SIZE > 7                //等待任务列表字节>7
782     *ptbl = 0x00;                        //OSEventTbl[7]清空
783 #endif
784 }
785 #endif
786 /*$PAGE*/
787 /*
788 ****
789 *                                     任务调度
790 *
791 * 描述: uC/OS-II总是运行进入就绪态任务中优先级最高的那一个。确定哪个任务优先级最高,下面该哪
792 *       一个任务运行了的工作是由调度器(Scheduler)完成的。任务级的调度是由函数OSSched()完成的。
793 *       中断级的调度是由另一个函数OSIntExt()完成的eduling)。
794 *
795 * 参数: none
796 *
797 * 返回: none
798 *
799 * 注意: 1) 这是一个uC/OS-II内部函数,你不能在应用程序中使用它
800 *       2) 给调度器上锁用于禁止任务调度(查看 OSSchedLock()函数)
801 *
802 * 说明: 1) 任务切换很简单,由以下两步完成,将被挂起任务的微处理器寄存器推入堆栈,然后将较高优先
803 *       级的任务的寄存器值从栈中恢复到寄存器中。在uC/OS-II中,就绪任务的栈结构总是看起来跟刚
804 *       刚发生过中断一样,所有微处理器的寄存器都保存在栈中。换句话说,uC/OS-II运行就绪态的任
805 *       务所要做的一切,只是恢复所有的CPU寄存器并运行中断返回指令。为了做任务切换,运行
806 *       OS_TASK_SW(),人为模仿了一次中断。多数微处理器有软中断指令或者陷阱指令TRAP来实现上述操
807 *       作。中断服务子程序或陷阱处理(Trap hardler),也称作事故处理(exception handler),必须提
808 *       供中断向量给汇编语言函数OSCtxSw()。OSCtxSw()除了需要OS_TCBHighRdy指向即将被挂起的任务,
809 *       还需要让当前任务控制块OSTCBCur指向即将被挂起的任务,参见第8章,移植uC/OS-II,有关于
810 *       OSCtxSw()的更详尽的解释。
811 *       2) OSSched()的所有代码都属临界段代码。在寻找进入就绪态的优先级最高的任务过程中,为防止中
812 *       断服务子程序把一个或几个任务的就绪位置位,中断是被关掉的。为缩短切换时间,OSSched()全
813 *       部代码都可以用汇编语言写。为增加可读性,可移植性和将汇编语言代码最少化,OSSched()是用
814 *       C写的。
815 ****
816 */
817
818 void OS_Sched (void)                    //任务调度函数
819 {
820     #if OS_CRITICAL_METHOD == 3        //中断函数被设定为模式3
821         OS_CPU_SR cpu_sr;
822     #endif
823     INT8U y;                            //定义一个8位整数y
824
825     OS_ENTER_CRITICAL();                //关闭中断
826     //如果中断嵌套次数>0,且上锁(调度器)嵌套次数>0,函退出,不做任何调度
827     if ((OSIntNesting == 0) && (OSLockNesting == 0)) {
828         //如果函数不是在中断服务子程序中调用的,且调度允许的,则任务调度函数将找出进入就绪态的
829         //最高优先级任务,进入就绪态的任务在就绪表中OSRdyTbl[]中相应位置位。
830         y = OSUnMapTbl[OSRdyGrp];
831         OSPrioHighRdy = (INT8U)((y << 3) + OSUnMapTbl[OSRdyTbl[y]]);
832         //找到最高优先级任务后,函数检查这个优先级最高的任务是否是当前正在运行的任务,以避免不
833         //必要的任务调度,多花时间
834         if (OSPrioHighRdy != OSPrioCur) {

```

```

836 //为实现任务切换, OSTCBHighRdy必须指向优先级最高的那个任务控制块OS_TCB, 这是通过将
837 //以OSPrioHighRdy为下标的OSTCBPrioTbl[]数组中的那个元素赋给OSTCBHighRdy来实现的
838 OSTCBHighRdy = OSTCBPrioTbl[OSPrioHighRdy];
839 OSCtxSwCtr++; //统计计数器OSCtxSwCtr加1, 以跟踪任务切换次数

840 OS_TASK_SW(); //最后宏调用OS_TASK_SW()来完成实际上的任务切换
841 }
842 }
843 OS_EXIT_CRITICAL(); //打开中断
844 }
845 /*$PAGE*/
846 /*
847 ****
848 * 空闲任务 IDLE TASK
849 *
850 * 描述: 这个函数是uC/OS-II内部函数, uC/OS-II总要建立一个空闲任务, 这个任务在没有其它任务进入
851 * 就绪态时投入运行。这个空闲任务永远设为最低优先级, 即OS_LOWEST_PRIO。空闲任务不可能被应
852 * 用软件删除。
853 *
854 * 参数: 无
855 *
856 * 返回: 无
857 *
858 * 注意: 1) OSTaskIdleHook() 可以允许用户在函数中写入自己的代码, 可以借助OSTaskIdleHook(), 让
859 * CPU执行STOP指令, 从而进入低功耗模式, 当应用系统由电池供电时, 这种方式特别有用。
860 * 2) 这个函数永远处于就绪态, 所以不要在OSTaskIdleHook() 中调用可以使任务挂起的PEND函数、
861 * OSTimeDly???() 函数和OSTaskSuspend() 函数
862 ****
863 */
864
865 void OS_TaskIdle (void *pdata) //空闲任务函数(指向一个数据结构)
866 {
867 #if OS_CRITICAL_METHOD == 3 //中断函数被设定为模式3
868 OS_CPU_SR cpu_sr;
869 #endif
870
871
872 pdata = pdata; //参数等于本身, 防止一些编译器不能编译
873 for (;;) { //无限循环
874 OS_ENTER_CRITICAL(); //关闭中断
875 OSIdleCtr++; //32位空闲计数器加1, 提供给统计任务消耗CPU事件
876 OS_EXIT_CRITICAL(); //打开中断
877 OSTaskIdleHook(); //空闲任务钩子程序
878 }
879 }
880 /*$PAGE*/
881 /*
882 ****
883 * 统计任务 STATISTICS TASK
884 *
885 * 描述: uC/OS-II有一个提供运行时间统计的任务。这个任务叫做OSTaskStat(), 如果用户将系统定义常
886 * 数OS_TASK_STAT_EN(见文件OS_CFG.H)设为1, 这个任务就会建立。一旦得到了允许, OSTaskStat()
887 * 每秒钟运行一次(见文件OS_CORE.C), 计算当前的CPU利用率。换句话说, OSTaskStat()告诉用户
888 * 应用程序使用了多少CPU时间, 用百分比表示, 这个值放在一个有符号8位整数OSCPUUsage中, 精读
889 * 度是1个百分点。
890 * 如果用户应用程序打算使用统计任务, 用户必须在初始化时建立一个唯一的任务, 在这个任务中
891 * 调用OSStatInit()(见文件OS_CORE.C)。换句话说, 在调用系统启动函数OSStart()之前, 用户初
892 * 始代码必须先建立一个任务, 在这个任务中调用系统统计初始化函数OSStatInit(), 然后再建立
893 * 应用程序中的其它任务
894 *
895 *
896 * 
$$OSCPUUsage = 100 * (1 - \frac{OSIdleCtr}{OSIdleCtrMax})$$
 (units are in %)
897 *
898 *
899 * 参数: pdata 指向一个数据结构, 该结构用来在建立统计任务时向任务传递参数
900 *
901 * 返回: 无
902 *
903 * 注意: 1) uC/OS-II已经将空闲任务的优先级设为最低, 即OS_LOWEST_PRIO, 统计任务的优先级设为次
904 * 低, OS_LOWEST_PRIO-1。
905 * 2) 因为用户的应用程序必须先建立一个起始任务TaskStart()。在使用统计任务前, 用户必须首
906 * 先调用的是uC/OS-II中的系统初始化函数OSInit(),
907 * 3) 在创建统计任务之前, 为了保持系统达到稳定状态, 需要延迟5秒钟, 你必须至少延时2秒钟
908 * 以设定最大空闲计数值
909 * 3) We delay for 5 seconds in the beginning to allow the system to reach steady state
910 * and have all other tasks created before we do statistics. You MUST have at least

```

```

911 *          a delay of 2 seconds to allow for the system to establish the maximum value for
912 *          the idle counter.
913 *****/
914 */
915
916 #if OS_TASK_STAT_EN > 0          //允许生产OS_TaskStat() 函数
917 void OS_TaskStat (void *pdata)  //统计任务(指向一个数据结构)
918 {
919     #if OS_CRITICAL_METHOD == 3    //中断函数被设定为模式3
920         OS_CPU_SR cpu_sr;
921     #endif
922     INT32U run;                    //定义一个运算变量
923     INT8S usage;                  //定义一个使用变量
924
925
926     pdata = pdata;                //数据结构指针等于本身, 防止编译器不能编译
927     while (OSStatRdy == FALSE) {  //判断统计任务就绪标志OSStatRdy设为“假”
928         OSTimeDly(2 * OS_TICKS_PER_SEC); //延时2秒钟, 等待统计任务做好准备
929     }
930     for (;;) {                    //无限循环
931         OS_ENTER_CRITICAL();      //关闭中断
932         OSIdleCtrRun = OSIdleCtr; //获得当前的空闲计数值
933         run = OSIdleCtr;          //并保存到运算变量中, 以便运算
934         OSIdleCtr = 0L;           //空闲计数器OSIdleCtr被清零
935         OS_EXIT_CRITICAL();        //打开中断
936         if (OSIdleCtrMax > 0L) {    //如果最大空闲计数值 > 0
937             usage = (INT8S)(100L - 100L * run / OSIdleCtrMax); 求CPU利用率
938             if (usage >= 0) {       //不能是负的百分值
939                 OSCPUUsage = usage; //如果是正的值, 保存它
940             } else {
941                 OSCPUUsage = 0;     //如果是负值, 设为0
942             }
943         } else {                  //如果最大空闲计数值不 > 0
944             OSCPUUsage = 0;         //如果是负值, 设为0
945         }
946         OSTaskStatHook();          //调用钩子函数, 可添加自己的代码
947         OSTimeDly(OS_TICKS_PER_SEC); //调用延迟函数OSTimeDly()将自身延时1个时钟节拍以停止自身的运
行
948     }
949 }
950 #endif
951 /*$PAGE*/
952 /*
953 *****/
954 *          任务控制块初始化 INITIALIZE TCB
955 *
956 * 描述: 这个函数是uC/OS-II内部函数, 在建立任务时调用的初始化任务控制块OS_TCB函数, 含7个参数,
957 *      (查看 OSTaskCreate() 和 OSTaskCreateExt()).
958 *
959 * 参数: prio      任务的优先级
960 *
961 *      ptos      OSTaskInit() 建立栈结构以后, ptos是指向栈顶的指针, 且保存在OS_TCB的OSTCBStkPrt中
962 *
963 *      pbos      指向栈底的指针, 保存在OSTCBStkBottom变元中
964 *
965 *      id        任务标志符(0..65535), 保存在.OSTCBId中
966 *
967 *      stk_size   堆栈的容量, 保存在OS_TCB的OSTABStkSize中
968 *
969 *      pext       OS_TCB中的扩展指针, .OSTCBExtPtr的值
970 *
971 *      opt        OS_TCB的选择项, 保存在.OSTCBOpt中
972 *
973 * 返回: OS_NO_ERR      调用成功
974 *      OS_NO_MORE_TCB 没有更多的任务控制块被分配, 将无法创建新的任务
975 *
976 * 注意: 这个函数是uC/OS-II内部函数, 你不可以调用它。
977 *****/
978 */
979 //初始化任务控制块TCB(优先级指针、栈顶指针、栈底指针、任务标志符、堆栈容量、扩展指针、选择项)
980 INT8U OS_TCBInit (INT8U prio, OS_STK *ptos, OS_STK *pbos, INT16U id, INT32U stk_size,
981 void *pext, INT16U opt)
982 {
983     #if OS_CRITICAL_METHOD == 3    //中断函数被设定为模式3
984         OS_CPU_SR cpu_sr;
985     #endif

```

```

986     OS_TCB     *ptcb;                                //定义一个PCB变量
987
988
989     OS_ENTER_CRITICAL();                             //关闭中断
990     ptcb = OSTCBFreeList;                             //分配一个空任务控制块给ptcb
991     if (ptcb != (OS_TCB *)0) {                       //如果缓冲池有空余TCB, 这个TCB被初始化
992         OSTCBFreeList = ptcb->OSTCBNext;             //指向TCB的双向链接的后链接
993         OS_EXIT_CRITICAL();                             //打开中断
994         ptcb->OSTCBStkPtr = ptos;                     //指向当前TCB的栈顶指针(输入的数据)
995         ptcb->OSTCBPrio = (INT8U)prio;                 //保存当前TCB的优先级别(输入的数据)
996         ptcb->OSTCBStat = OS_STAT_RDY;                 //设定当前TCB的状态字(内容为(准备完毕))
997         ptcb->OSTCBDly = 0;                             //允许任务等待的最大字节节拍为0
998
999     #if OS_TASK_CREATE_EXT_EN > 0                     //允许生成OSTaskCreateExt()函数
1000         ptcb->OSTCBExtPtr = pext;                     //指向用户定义的任务控制块(扩展指针)
1001         ptcb->OSTCBStkSize = stk_size;                 //设定堆栈的容量
1002         ptcb->OSTCBStkBottom = pbos;                   //指向指向栈底的指针
1003         ptcb->OSTCBOpt = opt;                           //保存OS_TCB的选择项
1004         ptcb->OSTCBId = id;                             //保存任务标志符
1005     #else
1006         pext = pext;                                   //扩展指针
1007         stk_size = stk_size;                           //堆栈的容量
1008         pbos = pbos;                                   //栈底的指针
1009         opt = opt;                                     //选择项
1010         id = id;                                       //任务标志符
1011     #endif
1012
1013     #if OS_TASK_DEL_EN > 0                             //允许生成 OSTaskDel() 函数代码函数
1014         ptcb->OSTCBDelReq = OS_NO_ERR;                 //如果可以删除任务本身, 可以从每个
1015                                                     //OS_TCB中节省出一个布尔量
1016     #endif
1017                                                     //对一些参数提前运算, 为了节省CPU的操作事件
1018     ptcb->OSTCBY = prio >> 3;
1019     ptcb->OSTCBBitY = OSMaTbl[ptcb->OSTCBY];
1020     ptcb->OSTCBX = prio & 0x07;
1021     ptcb->OSTCBBitX = OSMaTbl[ptcb->OSTCBX];
1022
1023     #if OS_EVENT_EN > 0                               //如果不打算在应用程序中使用各类事件
1024         ptcb->OSTCBEvtPtr = (OS_EVENT *)0;             //OS_TCB中OSTCBEvtPtr就不会出现
1025     #endif                                             //针对的事件为信号量, 互斥型信号量、消息邮箱、消息队列
1026                                                     //当满足 版本大于2.51 且 事件标志允许 且 有最大事件标志 及 允许删除任务
1027     #if (OS_VERSION >= 251) && (OS_FLAG_EN > 0) && (OS_MAX_FLAGS > 0) && (OS_TASK_DEL_EN > 0)
1028         ptcb->OSTCBFlagNode = (OS_FLAG_NODE *)0;      //则向事件标志节点的指针被初始化为空指针
1029     #endif
1030
1031     #if (OS_MBOX_EN > 0) || ((OS_Q_EN > 0) && (OS_MAX_QS > 0))
1032         ptcb->OSTCBMsg = (void *)0;                   //满足以上条件, 指向传递给任务的消息指针为0空指针
1033     #endif
1034
1035     #if OS_VERSION >= 204                             //如果版本大于2.04
1036         OSTCBInitHook(ptcb);                           //允许使用OSTCBInitHook(ptcb)函数, 可对其加代码
1037     #endif //主要增加OS_TCB扩展, 浮点运算、MMU寄存器、与任务相关内容, 调用此程序时中断开着的
1038
1039     OSTaskCreateHook(ptcb);                           //调用用户建立任务钩子程序
1040     //该函数能够扩展[OSTaskCreate()或OSTaskCreateExt()函数]
1041     //当OS_CPU_HOOKS_EN为1时, OSTaskCreateHook()可以在OS_CPU.C中定义
1042     //若OS_CPU_HOOKS_EN为0时, 则可以在任何其它地方定义
1043     //调用此程序时中断开着的。
1044     OS_ENTER_CRITICAL();
1045     OSTCBPrioTbl[prio] = ptcb;
1046     ptcb->OSTCBNext = OSTCBList;                       //链接到任务控制块链接串
1047     ptcb->OSTCBPrev = (OS_TCB *)0;
1048     if (OSTCBList != (OS_TCB *)0) {
1049         OSTCBList->OSTCBPrev = ptcb;
1050     }
1051     OSTCBList = ptcb;                                   //让该任务进入就绪态
1052     OSRdyGrp |= ptcb->OSTCBBitY;
1053     OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
1054     OS_EXIT_CRITICAL();
1055     return (OS_NO_ERR);                                 //调用成功, 最后让此函数返回到调用函数[OSTaskCreate()或
1056 }                                                       //OSTaskCreateExt()函数], 返回值表示分配到任务控块, 并初始化了
1057 OS_EXIT_CRITICAL();                                   //打开中断
1058 return (OS_NO_MORE_TCB);                             //没有更多的任务控制块被分配, 将无法创建新的任务
1059 }
1060

```



```

1  /*
2  ****
3  *                                     uC/OS-II实时控制内核
4  *                                     主要的包含文件
5  *                                     --事件标志组管理--
6  * 文    件: OS_FLAG.C    事件标志组代码
7  * 作    者: Jean J. Labrosse
8  * 中文注解: 钟常慰  zhongcw @ 126.com  译注版本: 1.0  请尊重原版内容
9  ****
10 */
11
12 #ifndef OS_MASTER_FILE    //是否已定义OS_MASTER_FILE主文件
13 #include "INCLUDES.H"    //包含"includes.h"文件, 部分C语言头文件的汇总打包文件
14 #endif                    //定义结束
15
16 //条件编译: UCOS版本>= 251 且 OS_FLAG_EN 允许产生事件标志程序代码 且 最大事件标志>0
17 #if (OS_VERSION >= 251) && (OS_FLAG_EN > 0) && (OS_MAX_FLAGS > 0)
18 /*
19 ****
20 *                                     局部函数原型
21 ****
22 */
23
24 static void    OS_FlagBlock(OS_FLAG_GRP *pgrp, OS_FLAG_NODE *pnode,
25                             OS_FLAGS flags, INT8U wait_type, INT16U timeout);
26 static BOOLEAN OS_FlagTaskRdy(OS_FLAG_NODE *pnode, OS_FLAGS flags_rdy);
27
28 /*$PAGE*/
29 /*
30 ****
31 *                                     检查事件标志组
32 *
33 * 描述: 检查事件标志组中的事件标志位是置位还是清0。
34 *       应用程序可以检查任意一位是置位还是清0, 也可以检查所有位是置位还是清0。
35 *       此函数于OSFlagPend()不同在于, 如果需要的事件标志没有产生, 那么调用该函数的任务
36 *       并不挂起。
37 *
38 * 参数: pgrp    指向事件标志组的指针。建立事件标志组时(OSFlagCreate())得到该指针。
39 *
40 *       flags    指定需要检查的事件标志位。为1则检查对应位; 为0则忽略对应位。
41 *
42 *       wait_type 定义等待事件标志位的方式。可以分为以下4种:
43 *
44 *           OS_FLAG_WAIT_CLR_ALL    所有指定事件标志位清 (0);
45 *           OS_FLAG_WAIT_CLR_ANY    任意指定事件标志位清 (0);
46 *           OS_FLAG_WAIT_SET_ALL    所有指定事件标志位置 (1);
47 *           OS_FLAG_WAIT_SET_ANY    任意指定事件标志位置 (1)。
48 *
49 *       注意:如果需要在得到期望的事件标志后, 恢复该事件标志, 则可以在调用函数时
50 *       将该参数加上一个常量OS_FLAG_CONSUME。例如, 如果等待事件标志组中
51 *       任意指定事件标志位置位, 并且在任意事件标志位置位后清除该位, 则把参
52 *       数wait_type设置为:
53 *           OS_FLAG_WAIT_SET_ANY + OS_FLAG_CONSUME
54 *
55 *       err    指向包含错误码的变量的指针。返回的错误码可能为以下几种:
56 *           OS_NO_ERR        调用成功;
57 *           OS_ERR_EVENT_TYPE  pgrp指针不是指向事件标志组的指针;
58 *           OS_FLAG_ERR_WAIT_TYPE wait_type参数不是指定的4种方式之一;
59 *           OS_FLAG_INVALID_PGRP pgrp是一个空指针;
60 *           OS_FLAG_ERR_NOT_RDY 指定的事件标志没有发生。
61 *
62 * 返回: 返回事件标志组的事件标志状态
63 *
64 * 注意/警告: 1、必须先建立事件标志组, 然后使用;
65 *            2、如果指定的事件标志没有发生, 则调用任务并不挂起。
66 ****
67 */
68 #if OS_FLAG_ACCEPT_EN > 0    //允许生成 OSFlagAccept() 代码
69 // 检查事件标志组函数(标志组的指针、事件标志位、等待事件标志位的方式、错误码指针)
70 OS_FLAGS OSFlagAccept(OS_FLAG_GRP *pgrp, OS_FLAGS flags, INT8U wait_type, INT8U *err)
71 {
72 #if OS_CRITICAL_METHOD == 3    //中断函数被设定为模式3
73     OS_CPU_SR    cpu_sr;
74 #endif
75     OS_FLAGS    flags_cur;    //定义一个"取出当前位"保存值
76     OS_FLAGS    flags_rdy;    //定义一个"准备完毕"含量值

```

```

77     BOOLEAN        consume;                //定义一个“清除”事件标志位(保存值)
78
79
80 #if OS_ARG_CHK_EN > 0                      //所有参数在指定的范围之内
81     if (pgrp == (OS_FLAG_GRP *)0) {        //返回的事件标志组pgrp是一个空指针
82         *err = OS_FLAG_INVALID_PGRP;        //‘pgrp’不是指向事件标志组的指针
83         return ((OS_FLAGS)0);              //返回0
84     }
85     if (pgrp->OSFlagType != OS_EVENT_TYPE_FLAG) { //当数据指针类型不是事件标志组类型
86         *err = OS_ERR_EVENT_TYPE;          //‘pgrp’是空指针
87         return ((OS_FLAGS)0);              //返回0
88     }
89 #endif
90     if (wait_type & OS_FLAG_CONSUME) {      //保存这个事件标志位在局部变量中consume
91         wait_type &= ~OS_FLAG_CONSUME;    //wait_type保存这个反值
92         consume = TRUE;                    //“清除”事件标志位置1，需要对这个标志清0
93     } else {                                //否则
94         consume = FALSE;                   //“清除”事件标志位为0
95     }
96 /*$PAGE*/
97     *err = OS_NO_ERR;                      /* Assume NO error until proven otherwise. */
98     OS_ENTER_CRITICAL();                   //打开中断
99     switch (wait_type) {                   //判断等待事件标志位的方式
100         case OS_FLAG_WAIT_SET_ALL:        //1、如果所有指定事件标志位置1
101             flags_rdy = pgrp->OSFlagFlags & flags; //取事件标志组中由flags参数指定的事件标志位
102             if (flags_rdy == flags) {      //如果取出的位的状态恰好完全符合预期的状态
103                 if (consume == TRUE) {    //查看是否需要对这个标志清0
104                     pgrp->OSFlagFlags &= ~flags_rdy; //将任务需要等待的事件标志位置位(准备完毕)
105                 }
106             } else {
107                 *err = OS_FLAG_ERR_NOT_RDY; //指定的事件标志没有发生
108             }
109             flags_cur = pgrp->OSFlagFlags; //获取事件标志组的新的事件标志位值
110             OS_EXIT_CRITICAL();            //打开中断
111             break;                         //条件满足，则跳出此选择体
112
113         case OS_FLAG_WAIT_SET_ANY:         //2、任意指定事件标志位置1
114             flags_rdy = pgrp->OSFlagFlags & flags; //取事件标志组中由flags参数指定的事件标志位
115             if (flags_rdy != (OS_FLAGS)0) { //如果指定的事件标志位中任意一个已经置位，
116                 //则等待操作立刻结束，返回调用函数
117                 if (consume == TRUE) {    //查看是否需要对这个标志清0
118                     pgrp->OSFlagFlags &= ~flags_rdy; //将任务需要等待的事件标志位置位(准备完毕)
119                 }
120             } else {
121                 *err = OS_FLAG_ERR_NOT_RDY; //指定的事件标志没有发生
122             }
123             flags_cur = pgrp->OSFlagFlags; //获取事件标志组的新的事件标志位值
124             OS_EXIT_CRITICAL();            //打开中断
125             break;                         //条件满足，则跳出此选择体
126
127 #if OS_FLAG_WAIT_CLR_EN > 0                //允许生成 Wait on Clear 事件标志代码，其实使用前两种方式即可
128         case OS_FLAG_WAIT_CLR_ALL:        //3、所有指定事件标志位清0
129             flags_rdy = ~pgrp->OSFlagFlags & flags; //取事件标志组中由flags参数指定的事件标志位
130             if (flags_rdy == flags) {      //如果取出的位的状态恰好完全符合预期的状态
131                 if (consume == TRUE) {    //查看是否需要对这个标志清0
132                     pgrp->OSFlagFlags |= flags_rdy; //将任务需要等待的事件标志位清0(准备完毕)
133                 }
134             } else {
135                 *err = OS_FLAG_ERR_NOT_RDY; //指定的事件标志没有发生
136             }
137             flags_cur = pgrp->OSFlagFlags; //获取事件标志组的新的事件标志位值
138             OS_EXIT_CRITICAL();            //打开中断
139             break;                         //条件满足，则跳出此选择体
140
141         case OS_FLAG_WAIT_CLR_ANY:        //4、所有指定事件标志位清0
142             flags_rdy = ~pgrp->OSFlagFlags & flags; //取事件标志组中由flags参数指定的事件标志位
143             if (flags_rdy != (OS_FLAGS)0) { //如果指定的事件标志位中任意一个已经置位
144                 //则等待操作立刻结束，返回调用函数
145                 if (consume == TRUE) {    //查看是否需要对这个标志清0
146                     pgrp->OSFlagFlags |= flags_rdy; //将任务需要等待的事件标志位清0(准备完毕)
147                 }
148             } else {
149                 *err = OS_FLAG_ERR_NOT_RDY; //指定的事件标志没有发生
150             }
151             flags_cur = pgrp->OSFlagFlags; //获取事件标志组的新的事件标志位值
152             OS_EXIT_CRITICAL();            //打开中断

```



```

153         break;
154 #endif
155
156     default:                                //以上类型全部不是
157         OS_EXIT_CRITICAL();                  //打开中断
158         flags_cur = (OS_FLAGS)0;             //事件标志位值清0
159         *err      = OS_FLAG_ERR_WAIT_TYPE;   //wait_type参数不是指定的4种方式之一
160         break;                                //条件满足，则跳出此选择体
161     }
162     return (flags_cur);                       //返回新的事件标志位值(0)，并返回到调用函数
163 }
164 #endif
165
166 /*$PAGE*/
167 /*
168 ****
169 *                                     创建一个事件标志组
170 *
171 * 描述：创建并初始化一个事件标志组
172 *
173 * 参数：flags  事件标志组的事件标志初值
174 *
175 *      err      指向包含错误码的变量的指针。返回的错误码可能为以下几种：
176 *              OS_NO_ERR          成功创建事件标志组
177 *              OS_ERR_CREATE_ISR  从中断中调用OSFlagCreate()函数
178 *              OS_FLAG_GRP_DEPLETED 系统没有剩余的空闲事件标志组，需要更改OS_CFG.H中
179 *                                  的事件标志组数目配置
180 *
181 * 返回：如果成功创建事件标志组，则返回该事件标志组的指针；
182 *       若系统没有剩余的空闲事件标志组，则返回空指针。
183 *
184 * 注意/警告：在使用任何事件标志组功能之前，必须使用该函数创建事件标志组。
185 ****
186 */
187
188 OS_FLAG_GRP *OSFlagCreate (OS_FLAGS flags, INT8U *err)
189 {
190     //建立一个事件标志组(初值、错误码)
191     //中断函数被设定为模式3
192     #if OS_CRITICAL_METHOD == 3
193         OS_CPU_SR  cpu_sr;
194     #endif
195     OS_FLAG_GRP *pgrp;                                //定义一个事件标志变量
196
197     if (OSIntNesting > 0) {                            //中断嵌套数>0时，表示还有中断任务在运行
198         *err = OS_ERR_CREATE_ISR;                       //返回(从中断中调用OSFlagCreate()函数)不允许
199         return ((OS_FLAG_GRP *)0);                     //系统没有剩余的空闲事件标志组，则返回空指针
200     }
201     OS_ENTER_CRITICAL();                                //关闭中断
202     pgrp = OSFlagFreeList;                             //pgrp=从系统的空闲事件标志组链表中取一个空闲的事件标志组
203     //如果返回一个空指针，说明系统已经没有一个空闲的事件标志组可以分配
204     if (pgrp != (OS_FLAG_GRP *)0) {
205         //分配后，调整系统空闲事件标志组链表指针，os_max_flags中
206         OSFlagFreeList = (OS_FLAG_GRP *)OSFlagFreeList->OSFlagWaitList;
207         pgrp->OSFlagType = OS_EVENT_TYPE_FLAG;          //数据结构=事件标志组
208         pgrp->OSFlagFlags = flags;                      //事件标志初始化
209         pgrp->OSFlagWaitList = (void *)0;              //等待任务链接表指针初始化为NULL
210         OS_EXIT_CRITICAL();                             //打开中断
211         *err = OS_NO_ERR;                               //成功创建标志组
212     } else {
213         OS_EXIT_CRITICAL();                             //没有空余的事件标志组
214         *err = OS_FLAG_GRP_DEPLETED;                  //错误=没有空余的事件标志组
215     }
216     return (pgrp);                                     // 返回事件标志组指针
217 }
218
219 /*$PAGE*/
220 /*
221 ****
222 *                                     删除一个事件标志组
223 *
224 * 描述：用于删除一个事件标志组。因为多任务可能会试图继续使用已经删除了的事件标志组，故调
225 *       用本函数有风险，需小心。一般在删除事件标志组之前，应该首先删除与本事件有关任务。
226 *
227 * 参数：pgrp  指向事件标志组的指针。建立事件标志组时(OSFlagCreate())得到该指针。
228 *

```

```

229 *      opt      确定删除一个事件的条件值:
230 *      opt == OS_DEL_NO_PEND 指明是仅在没有任务等待事件标志组时删除该事件标志组
231 *      opt == OS_DEL_ALWAYS 指明不管是否有任务等待事件标志组都删除该事件标志组
232 *      如果是后者, 所有等待该事件标志组的任务都被置位就绪。
233 *
234 *      err 指向包含错误码的变量的指针。返回的错误码可能为以下几种之一:
235 *      OS_NO_ERR      成功删除该事件标志组;
236 *      OS_ERR_DEL_ISR  从中断中调用OSFlagDel()函数;
237 *      OS_FLAG_INVALID_PGRP pgrp是一个空指针;
238 *      OS_ERR_EVENT_TYPE pgrp不是指向事件标志组的指针;
239 *      OS_ERR_INVALID_OPT opt参数不是指定的值;
240 *      OS_ERR_TASK_WAITING 如果opt参数为OS_DEL_NO_PEND, 那么此时有任务等待
241 *      事件标志组
242 *
243 * 返回: 如果事件标志组被删除, 组则返回空指针;
244 *      如果没有删除, 则仍然返回指向该事件标志组的指针。
245 *      后一种情况需要检查出错代码, 找出事件标志的失败的原因
246 *
247 * 注意: 1) 需要小心, 可能有其它任务正在等待该事件标志组的事件标志
248 *      2) 该函数有可能长时间关闭中断, 其时间长短决定于标志组的任务个数
249 *****/
250 */
251
252 #if OS_FLAG_DEL_EN > 0                                //允许生成 OSFlagDel() 代码
253                                     //删除一个事件标志组(指针、条件值、错误值)
254 OS_FLAG_GRP *OSFlagDel (OS_FLAG_GRP *pgrp, INT8U opt, INT8U *err)
255 {
256     #if OS_CRITICAL_METHOD == 3                //中断函数被设定为模式3
257         OS_CPU_SR      cpu_sr;
258     #endif
259     BOOLEAN      tasks_waiting;                //任务等待条件
260     OS_FLAG_NODE *pnode;                       //定义标志节点
261
262     if (OSIntNesting > 0) {                    //中断嵌套数>0时, 表示还有中断任务在运行
263         *err = OS_ERR_DEL_ISR;                 //返回(从中断中调用OSFlagDel()函数)不允许
264         return (pgrp);                         //返回该事件标志组的指针
265     }
266     #if OS_ARG_CHK_EN > 0                    //所有参数在指定的范围之内
267     if (pgrp == (OS_FLAG_GRP *)0) {           //返回的事件标志组pgrp是一个空指针
268         *err = OS_FLAG_INVALID_PGRP;           //返回该事件标志组的指针
269         return (pgrp);
270     }
271     if (pgrp->OSFlagType != OS_EVENT_TYPE_FLAG) { //当数据指针类型不是事件标志组类型
272         *err = OS_ERR_EVENT_TYPE;             //pgrp不是指向事件标志组的指针
273         return (pgrp);                         //返回该事件标志组的指针
274     }
275     #endif
276     #endif
277     OS_ENTER_CRITICAL();                      //关闭中断
278     if (pgrp->OSFlagWaitList != (void *)0) {   //是否真的有任务在等待该标志组(不是0)
279         tasks_waiting = TRUE;                 //有任务在等待(真)
280     } else {                                  //否则
281         tasks_waiting = FALSE;                //没有任务在等待(假)
282     }
283     switch (opt) {                            // 1) 选择没有任务等待才删除事件标志组
284     case OS_DEL_NO_PEND:                      //当任务等待为假
285         if (tasks_waiting == FALSE) {         //事件标志组为空闲标志组
286             //标志组为空闲事件标志
287             //标志组放回到空闲事件标志链接表
288             //空余事件标志组=当前事件标志指针
289             pgrp->OSFlagType = OS_EVENT_TYPE_UNUSED;
290             pgrp->OSFlagWaitList = (void *)OSFlagFreeList;
291             OSFlagFreeList = pgrp;
292             OS_EXIT_CRITICAL();                //打开中断
293             *err = OS_NO_ERR;                 //成功删除事件标志组
294             return ((OS_FLAG_GRP *)0);        //成功删除事件标志组(返回0)
295         } else {                              //否则
296             OS_EXIT_CRITICAL();               //打开中断
297             *err = OS_ERR_TASK_WAITING;       //有任务在等待
298             return (pgrp);                   //返回该事件标志组的指针
299         }
300
301     case OS_DEL_ALWAYS:                       // 2) 多任务等待(全部置为就绪态)
302         pnode = pgrp->OSFlagWaitList;
303         while (pnode != (OS_FLAG_NODE *)0) {
304             OS_FlagTaskRdy(pnode, (OS_FLAGS)0);

```

```

305         pnode = pnode->OSFlagNodeNext;
306     }
307     //标志组为空闲事件标志
308     //标志组放回到空闲事件标志链接表
309     //空余事件标志组=当前事件标志指针
310     pgrp->OSFlagType = OS_EVENT_TYPE_UNUSED;
311     pgrp->OSFlagWaitList = (void *)OSFlagFreeList;
312     OSFlagFreeList = pgrp;
313     OS_EXIT_CRITICAL(); //打开中断
314     if (tasks_waiting == TRUE) { //当任务等待为假
315         OS_Sched(); //将最高就绪优先级状态任务运行(调度任务)
316     }
317     *err = OS_NO_ERR; //成功删除事件标志组
318     return ((OS_FLAG_GRP *)0); //成功删除事件标志组(返回0)
319
320     default: // 3) 以上两个态度都不是
321         OS_EXIT_CRITICAL(); //打开中断
322         *err = OS_ERR_INVALID_OPT; //返回以上两种状态都不是
323         return (pgrp); //返回该事件标志组的指针
324     }
325 }
326 #endif
327 /*$PAGE*/
328 /*
329 ****
330 * 等待事件标志组中的事件标志(WAIT ON AN EVENT FLAG GROUP)
331 *
332 * 描述: 任务等待事件标志组中的事件标志, 可以是多个事件标志的不同组合方式。可以等待任
333 * 意指定事件标志位置位或清0, 也可以是全部指定事件标志位置位或清0。如果任务等待
334 * 的事件标志位条件尚不满足, 则任务会被挂起, 直到指定的事件标志组合发生或指定的
335 * 等待时间超时。
336 *
337 * 参数: pgrp 指向事件标志组的指针。建立事件标志组时(OSFlagCreate())得到该指针。
338 *
339 * flags 指定需要检查的事件标志位。置为1, 则检查对应位; 置为0, 则忽略对应位。
340 *
341 * wait_type 定义等待事件标志位的方式。可以定为以下几种:
342 *
343 * OS_FLAG_WAIT_CLR_ALL 所有指定事件标志位清0 ;
344 * OS_FLAG_WAIT_SET_ALL 任意指定事件标志位置1 ;
345 * OS_FLAG_WAIT_CLR_ANY 所有指定事件标志位清0 ;
346 * OS_FLAG_WAIT_SET_ANY 任意指定事件标志位置1 ;
347 *
348 * 提示: 如果需要在得到期望的事件标志后恢复该事件标志, 则可以在调用该函数时, 将
349 * 该参数加上一个常量OS_FLAG_CONSUME。例如, 如果等待事件标志组中任意指定事
350 * 件标志位置位, 并且在任意事件标志位置位后清除该位, 则可以把参数wait_type
351 * 设置为: OS_FLAG_WAIT_SET_ANY + OS_FLAG_CONSUME
352 *
353 * timeout 以时钟节拍数目的等待超时时限。如果在这一时限得不到事件, 任务将恢复执行。
354 * timeout的值为0, 表示将无限期地等待事件。timeout的最大值是65535个时钟节
355 * 拍。timeout的值并不与时钟节拍同步, timeout计数器在下一个时钟节拍到来时
356 * 开始递减。在这里, 所谓下一个时钟节拍, 也就是立刻就到来了。
357 *
358 * err 指向错误代码的指针, 出错代码为以下值之一:
359 * OS_NO_ERR 成功调用;
360 * OS_ERR_PEND_ISR 从中断中调用该函数, 这是规则不允许的;
361 * OS_FLAG_INVALID_PGRP 'pgrp' 不是指向事件标志组的指针;
362 * OS_ERR_EVENT_TYPE 'pgrp' 是空指针
363 * OS_TIMEOUT 等待事件标志组的事件标志超时;
364 * OS_FLAG_ERR_WAIT_TYPE 'wait_type' 不是指定的参数之一。
365 *
366 * OS_FLAG_CONSUME 定义常量OS_FLAG_CONSUME为0x80
367 *
368 * 返回: 如果使用了OS_FLAG_CONSUME选项, 则返回清理后的事件标志组事件标志状态; 否则返
369 * 回OSFlagPend() 函数运行结束后的事件标志组事件标志状态; 如果发生了超时, 则返回0。
370 *
371 * 注意: 必须首先创建事件标志组, 再使用。
372 ****
373 */
374 //等待事件标志组的事件标志位(事件组指针、需要检查的标志位、等待事件标志位的方式、允许等待
375 的时钟节拍、出错代码的时钟节拍)
376 OS_FLAGS OSFlagPend (OS_FLAG_GRP *pgrp, OS_FLAGS flags, INT8U wait_type, INT16U timeout,
377 INT8U *err)
378 {
379 #if OS_CRITICAL_METHOD == 3 //中断函数被设定为模式3
380     OS_CPU_SR cpu_sr;

```

```

381 #endif
382 OS_FLAG_NODE node; //定义标志节点
383 OS_FLAGS flags_cur; //定义一个“取出当前位”保存值
384 OS_FLAGS flags_rdy; //定义一个“准备完毕”含量值
385 BOOLEAN consume; //定义一个“清除”事件标志位(保存值)
386
387
388 if (OSIntNesting > 0) { //中断嵌套数>0时，表示还有中断任务在运行
389     *err = OS_ERR_PEND_ISR; //从中断中调用该函数，这是规则不允许的
390     return ((OS_FLAGS)0); //返回0
391 }
392 #if OS_ARG_CHK_EN > 0 //所有参数在指定的范围之内
393 if (pgrp == (OS_FLAG_GRP *)0) { //返回的事件标志组pgrp是一个空指针
394     *err = OS_FLAG_INVALID_PGRP; //‘pgrp’不是指向事件标志组的指针
395     return ((OS_FLAGS)0); //返回0
396 }
397 if (pgrp->OSFlagType != OS_EVENT_TYPE_FLAG) { //当数据指针类型不是事件标志组类型
398     *err = OS_ERR_EVENT_TYPE; //‘pgrp’是空指针
399     return ((OS_FLAGS)0); //返回0
400 }
401 #endif
402 if (wait_type & OS_FLAG_CONSUME) { //保存这个事件标志位在局部变量中consume
403     wait_type &= ~OS_FLAG_CONSUME; //wait_type保存这个反值
404     consume = TRUE; //“清除”事件标志位置1，需要对这个标志清0
405 } else { //否则
406     consume = FALSE; //“清除”事件标志位为0
407 }
408 /*$PAGE*/
409 OS_ENTER_CRITICAL(); //打开中断
410 switch (wait_type) { //判断等待事件标志位的方式
411     case OS_FLAG_WAIT_SET_ALL: //1、如果所有指定事件标志位置1
412         flags_rdy = pgrp->OSFlagFlags & flags; //取事件标志组中由flags参数指定的事件标志位
413         if (flags_rdy == flags) { //如果取出的位的状态恰好完全符合预期的状态
414             if (consume == TRUE) { //查看是否需要对这个标志清0
415                 pgrp->OSFlagFlags &= ~flags_rdy; //将任务需要等待的事件标志位置位(准备完毕)
416             }
417             flags_cur = pgrp->OSFlagFlags; //获取事件标志组的新的事件标志位值
418             OS_EXIT_CRITICAL(); //打开中断
419             *err = OS_NO_ERR; //成功调用
420             return (flags_cur); //返回新的事件标志位值，并返回到调用函数
421         } else { //如果期望的事件标志位没有置位，任务将被挂起，
422             //直到事件标志位置位或者等待超时
423             OS_FlagBlock(pgrp, &node, flags, wait_type, timeout);
424             OS_EXIT_CRITICAL(); //打开中断
425         }
426         break; //条件满足，则跳出此选择体
427
428     case OS_FLAG_WAIT_SET_ANY: //2、任意指定事件标志位置1
429         flags_rdy = pgrp->OSFlagFlags & flags; //取事件标志组中由flags参数指定的事件标志位
430         if (flags_rdy != (OS_FLAGS)0) { //如果指定的事件标志位中任意一个已经置位，
431             //则等待操作立刻结束，返回调用函数
432             if (consume == TRUE) { //查看是否需要对这个标志清0
433                 pgrp->OSFlagFlags &= ~flags_rdy; //将任务需要等待的事件标志位置位(准备完毕)
434             }
435             flags_cur = pgrp->OSFlagFlags; //获取事件标志组的新的事件标志位值
436             OS_EXIT_CRITICAL(); //打开中断
437             *err = OS_NO_ERR; //成功调用
438             return (flags_cur); //返回新的事件标志位值，并返回到调用函数
439         } else { //如果期望的事件标志位没有置位，任务将被挂起，
440             //直到事件标志位置位或者等待超时
441             OS_FlagBlock(pgrp, &node, flags, wait_type, timeout);
442             OS_EXIT_CRITICAL(); //打开中断
443         }
444         break; //条件满足，则跳出此选择体
445
446     #if OS_FLAG_WAIT_CLR_EN > 0 //允许生成 Wait on Clear 事件标志代码，其实使用前两种方式即可
447     case OS_FLAG_WAIT_CLR_ALL: //3、所有指定事件标志位清0
448         flags_rdy = ~pgrp->OSFlagFlags & flags; //取事件标志组中由flags参数指定的事件标志位
449         if (flags_rdy == flags) { //如果取出的位的状态恰好完全符合预期的状态
450             if (consume == TRUE) { //查看是否需要对这个标志清0
451                 pgrp->OSFlagFlags |= flags_rdy; //将任务需要等待的事件标志位清0(准备完毕)
452             }
453             flags_cur = pgrp->OSFlagFlags; //获取事件标志组的新的事件标志位值
454             OS_EXIT_CRITICAL(); //打开中断
455             *err = OS_NO_ERR; //成功调用
456             return (flags_cur); //返回新的事件标志位值，并返回到调用函数

```

```

457     } else {                                     //如果期望的事件标志位没有置位，任务将被挂起，
458                                                     //直到事件标志位置位或者等待超时
459         OS_FlagBlock(pgrp, &node, flags, wait_type, timeout);
460         OS_EXIT_CRITICAL();                       //打开中断
461     }
462     break;                                       //条件满足，则跳出此选择体
463
464 case OS_FLAG_WAIT_CLR_ANY:                       //4、所有指定事件标志位清0
465     flags_rdy = ~pgrp->OSFlagFlags & flags;      //取事件标志组中由flags参数指定的事件标志位
466     if (flags_rdy != (OS_FLAGS)0) {              //如果指定的事件标志位中任意一个已经置位
467                                                     //则等待操作立刻结束，返回调用函数
468         if (consume == TRUE) {                   //查看是否需要对这个标志清0
469             pgrp->OSFlagFlags |= flags_rdy;      //将任务需要等待的事件标志位清0(准备完毕)
470         }
471         flags_cur = pgrp->OSFlagFlags;            //获取事件标志组的新的事件标志位值
472         OS_EXIT_CRITICAL();                       //打开中断
473         *err = OS_NO_ERR;                         //成功调用
474         return (flags_cur);                       //返回新的事件标志位值，并返回到调用函数
475     } else {                                     //如果期望的事件标志位没有置位，任务将被挂起，
476                                                     //直到事件标志位置位或者等待超时
477         OS_FlagBlock(pgrp, &node, flags, wait_type, timeout);
478         OS_EXIT_CRITICAL();                       //打开中断
479     }
480     break;                                       //条件满足，则跳出此选择体
481 #endif
482
483 default:                                         //以上类型全部不是
484     OS_EXIT_CRITICAL();                         //打开中断
485     flags_cur = (OS_FLAGS)0;                     //事件标志位值清0
486     *err = OS_FLAG_ERR_WAIT_TYPE;               // 'wait_type' 不是指定的参数之一
487     return (flags_cur);                         //返回新的事件标志位值(0)，并返回到调用函数
488 }
489 OS_Sched();                                     //调用高优先级就绪态任务运行
490 OS_ENTER_CRITICAL();                           //关闭中断
491 //当这个任务恢复运行时，首先检查是因为什么原因而恢复运行的。如果任务控制块的状态表明，该任
492 //务还在等待事件标志组的事件标志，那么这个任务一定是因为等待事件标志超时而恢复运行的
493 if (OSTCBCur->OSTCBStat & OS_STAT_FLAG) {       /* Have we timed-out? */
494     //在这种情况下，调用OS_FlagUnlink()函数，把这个OS_FLAG_NODE从事件标志组的等待任务链表中
495     //删除，并且返回一个出错代码，说明发生了等待超时。这段代码只是简单的将一个OS_FLAG_NODE
496     //从一个双向链表中删除
497     OS_FlagUnlink(&node);
498     OSTCBCur->OSTCBStat = OS_STAT_RDY;           //任务准备运行(准备完毕)
499     OS_EXIT_CRITICAL();                         //打开中断
500     flags_cur = (OS_FLAGS)0;                   //事件标志位值清0
501     *err = OS_TIMEOUT;                         //等待事件标志组的事件标志超时
502 } else {
503     //如果任务恢复运行不是因为等待超时，那么一定是任务等待的事件标志按照预期的方式产生了，此时，
504     //根据调用OSFlagPend()时传入的是否清除事件标志的参数，对事件标志进行相应的置位或清0操作
505     if (consume == TRUE) {                     //查看是否需要对这个标志清0
506         switch (wait_type) {                   //判断等待类型
507             case OS_FLAG_WAIT_SET_ALL:         //任意指定事件标志位置1
508             case OS_FLAG_WAIT_SET_ANY:         //所有指定事件标志位置1
509                                                     //将任务需要等待的事件标志位置位
510             pgrp->OSFlagFlags &= ~OSTCBCur->OSTCBFlagsRdy;
511             break;                               //条件满足，则跳出此选择体
512         }
513 #if OS_FLAG_WAIT_CLR_EN > 0                     //允许生成 Wait on Clear 事件标志代码
514             case OS_FLAG_WAIT_CLR_ALL:         //所有指定事件标志位清0
515             case OS_FLAG_WAIT_CLR_ANY:         //指定事件标志位清0
516                                                     //任意将任务需要等待的事件标志位清0
517             pgrp->OSFlagFlags |= OSTCBCur->OSTCBFlagsRdy;
518             break;                               //条件满足，则跳出此选择体
519 #endif
520         }
521     }
522     flags_cur = pgrp->OSFlagFlags;              //获取事件标志组的新的事件标志位值
523     OS_EXIT_CRITICAL();                         //打开中断
524     *err = OS_NO_ERR;                         //成功调用
525 }
526 return (flags_cur);                             //返回新的事件标志位值，并返回到调用函数
527 }
528 /**$PAGE*/
529 /**
530 *****
531 * 给出设定的事件标志位(POST EVENT FLAG BIT(S))
532 *

```



```

533 * 描述: 给出设定的事件标志位。指定的事件标志位可以设定为置位或清除。若OSFlagPost() 设
534 *      置的事件标志位正好满足某个等待使劲标志组的任务, 则OSFlagPost() 将该任务设为就绪。
535 *
536 * 参数: pgrp   指向事件标志组的指针。建立事件标志组时(OSFlagCreate())得到该指针。
537 *
538 *      flags   指定需要检查的事件标志位。如果opt参数位OS_FLAG_SET, 那么事件标志组中对应
539 *      的事件标志位置位。例如, 如果置位事件标志组的事件标志0、4和5, 则需要把
540 *      FLAGS参数设置位0x31(bit 0 是最低位)。若opt参数为OS_FLAG_CLR, 那么事件标志
541 *      组中对应的事件标志为被清0。
542 *
543 *      opt     表明是置位指定事件标志位(OS_FLAG_SET);
544 *      还是清0指定事件标志位(OS_FLAG_CLR)。
545 *
546 *      err     指向错误代码的指针, 出错代码为以下值之一:
547 *      OS_NO_ERR      成功调用
548 *      OS_FLAG_INVALID_PGRP  'pgrp' 指针为空指针
549 *      OS_ERR_EVENT_TYPE  'pgrp' 指针没有指向事件标志组结构;
550 *      OS_FLAG_INVALID_OPT opt不是指定的参数之一。
551 *
552 * 返回: 事件标志组的新的事件标志状态
553 *
554 * 警告: 1) 必须先创建事件标志组, 然后使用;
555 *      2) 这个函数的运行时间决定于等待事件标志组的任务的数目;
556 *      3) 关闭中断的时间也取决于等待事件标志组的任务的数目。
557 *****
558 */
559 //置位或清0事件标志组中的标志位(指针、标志位、条件值、错误码)
560 OS_FLAGS OSFlagPost (OS_FLAG_GRP *pgrp, OS_FLAGS flags, INT8U opt, INT8U *err)
561 {
562     #if OS_CRITICAL_METHOD == 3                //中断函数被设定为模式3
563         OS_CPU_SR      cpu_sr;
564     #endif
565     OS_FLAG_NODE *pnode;                      //定义标志节点指针
566     BOOLEAN        sched;                    //定义一个"最高就绪态运行"保存值
567     OS_FLAGS        flags_cur;                //定义一个"取出当前位"保存值
568     OS_FLAGS        flags_rdy;                //定义一个"准备完毕"含量值
569
570
571     #if OS_ARG_CHK_EN > 0                      //所有参数在指定的范围之内
572     if (pgrp == (OS_FLAG_GRP *)0) {           //返回的事件标志组pgrp是一个空指针
573         *err = OS_FLAG_INVALID_PGRP;          //返回的事件标志组pgrp是一个空指针
574         return ((OS_FLAGS)0);                 //返回0
575     }
576     if (pgrp->OSFlagType != OS_EVENT_TYPE_FLAG) { //当数据指针类型不是事件标志组类型
577         *err = OS_ERR_EVENT_TYPE;             // 'pgrp' 是空指针
578         return ((OS_FLAGS)0);                 //返回0
579     }
580     #endif
581     /*$PAGE*/
582     OS_ENTER_CRITICAL();                      //打开中断
583     switch (opt) {                            //判断设定的事件标志位条件
584     case OS_FLAG_CLR:                         //设定为清0指定事件标志位
585         pgrp->OSFlagFlags &= ~flags;          //清除事件标志
586         break;                                //条件满足, 则跳出此选择体
587
588     case OS_FLAG_SET:                         //设定为置位指定事件标志位
589         pgrp->OSFlagFlags |= flags;           //置位事件标志
590         break;                                //条件满足, 则跳出此选择体
591
592     default:                                  //如果两者都不是
593         OS_EXIT_CRITICAL();                  //打开中断
594         *err = OS_FLAG_INVALID_OPT;          //opt不是指定的参数之一
595         return ((OS_FLAGS)0);                //返回0
596     }
597     sched = FALSE;                           //假定对事件标志的操作不会导致一个更高优先级的任务进入就绪态
598     pnode = pgrp->OSFlagWaitList;             //保存事件标志组
599     while (pnode != (OS_FLAG_NODE *)0) {     //如果有任务在等待这个事件标志组,
600         switch (pnode->OSFlagNodeWaitType) {
601             //如果等待任务链表是空的, 本函数将获取当前事件标志组的事件标志状态, 并返回调用函数;
602             //若等待任务非空, 本函数将历遍所有的OS_FLAG_NODE, 以检查新设定的事件标志是否满足
603             //某个任务期待的条件。每个任务含以下四种情况:
604             case OS_FLAG_WAIT_SET_ALL:       //所有指定事件标志位置1
605                 flags_rdy = pgrp->OSFlagFlags & pnode->OSFlagNodeFlags;
606                 //如果一个任务等待的事件标志位条件得到满足, 那么这个任务将被标志为进入就绪态。
607                 if (flags_rdy == pnode->OSFlagNodeFlags) {
608                     //通过调用 OS_FlagTaskRdy() 来进入就绪态

```

```

609         if (OS_FlagTaskRdy(pnode, flags_rdy) == TRUE) {
610             sched = TRUE; //这里任务等待的事件标志已经满足，任务进入就绪态，
611             //所以需要立即进入任务调度。但是并不是每检查一个OS_FLAG_NODE，就进行
612             //进行一次任务调度，而是在历遍完全部等待任务后，进行一次总的调度，所
613             //以这里将是否需要进行调度的信息保留在一个局部布尔变量sched中。
614         }
615     }
616     break; //条件满足，则跳出此选择体
617
618     case OS_FLAG_WAIT_SET_ANY: //任意指定事件标志位置1
619         flags_rdy = pgrp->OSFlagFlags & pnode->OSFlagNodeFlags;
620         if (flags_rdy != (OS_FLAGS)0) {
621             if (OS_FlagTaskRdy(pnode, flags_rdy) == TRUE) {
622                 sched = TRUE;
623             }
624         }
625         break;
626
627 #if OS_FLAG_WAIT_CLR_EN > 0 //允许生成 Wait on Clear 事件标志代码，其实使用前两种方式即可
628     case OS_FLAG_WAIT_CLR_ALL: //所有指定事件标志位清0
629         flags_rdy = ~pgrp->OSFlagFlags & pnode->OSFlagNodeFlags;
630         if (flags_rdy == pnode->OSFlagNodeFlags) {
631             if (OS_FlagTaskRdy(pnode, flags_rdy) == TRUE) {
632                 sched = TRUE;
633             }
634         }
635         break;
636
637     case OS_FLAG_WAIT_CLR_ANY: //任意指定事件标志位清0
638         flags_rdy = ~pgrp->OSFlagFlags & pnode->OSFlagNodeFlags;
639         if (flags_rdy != (OS_FLAGS)0) {
640             if (OS_FlagTaskRdy(pnode, flags_rdy) == TRUE) {
641                 sched = TRUE;
642             }
643         }
644         break;
645 #endif
646 }
647
648 pnode = pnode->OSFlagNodeNext; //通过双向链表得到下一个OS_FLAG_NODE的指针
649 }
650 //当历遍等待任务列表时，中断是关闭的，这意味着OSFlagPost()函数有可能导致中断的长时
651 //间关闭，特别是在OSFlagPost()函数可能引起多任务进入就绪态时，在这种情况下，该函数
652 //的执行时间仍然是有限的，而且是可能预先确定的
653 OS_EXIT_CRITICAL(); //打开中断
654 if (sched == TRUE) { //如果发现更高优先级任务由于对事件标志的操作而进入就绪态TRUE
655     OS_Sched(); //历遍完等待任务链表后，判断是否需要进入任务调度，这可能导致一个
656                 //刚刚接收到预期的事件标志而进入就绪态的更高优先级的任务开始运行。
657 }
658 OS_ENTER_CRITICAL(); //关闭中断
659 flags_cur = pgrp->OSFlagFlags; //获取事件标志组的新的事件标志位值
660 OS_EXIT_CRITICAL(); //打开中断
661 *err = OS_NO_ERR; //成功调用
662 return (flags_cur); //返回当前事件标志组的事件标志状态
663 }
664 /*$PAGE*/
665 /*
666 *****
667 * 查询事件标志组的当前事件标志状态(QUERY EVENT FLAG)
668 *
669 * 描述： 查询事件标志组的当前事件标志状态。在现在的版本中，该函数还不能返回等待该事件
670 * 标志组的任务列表
671 *
672 * 参数： pgrp 指向事件标志组的指针。建立事件标志组时(OSFlagCreate())得到该指针。
673 *
674 * err 指向错误代码的指针，出错代码为以下值之一：
675 * OS_NO_ERR 成功调用
676 * OS_FLAG_INVALID_PGRP 'pgrp' 指针为空指针
677 * OS_ERR_EVENT_TYPE 'pgrp' 指针没有指向事件标志组结构。
678 *
679 * 返回： 事件标志组的新的事件标志状态
680 *
681 * 警告： 1) 必须先创建事件标志组，然后使用；
682 * 2) 可以从中断中调用该函数。
683 *
684 * Called From: Task or ISR

```



```

685 *****
686 */
687
688 #if OS_FLAG_QUERY_EN > 0                //允许生成 OSFlagQuery()
689     //查询事件标志组的当前事件标志状态(事件标志组的指针、错误代码的指针)
690 OS_FLAGS OSFlagQuery (OS_FLAG_GRP *pgrp, INT8U *err)
691 {
692     #if OS_CRITICAL_METHOD == 3          //中断函数被设定为模式3
693         OS_CPU_SR cpu_sr;
694     #endif
695     OS_FLAGS flags;                      //定义一个"当前位"状态
696
697
698     #if OS_ARG_CHK_EN > 0                //所有参数在指定的范围之内
699         if (pgrp == (OS_FLAG_GRP *)0) { //返回的事件标志组pgrp是一个空指针
700             *err = OS_FLAG_INVALID_PGRP; //返回的事件标志组pgrp是一个空指针
701             return ((OS_FLAGS)0);        //返回0
702         }
703         if (pgrp->OSFlagType != OS_EVENT_TYPE_FLAG) { //当数据指针类型不是事件标志组类型
704             *err = OS_ERR_EVENT_TYPE;           //'pgrp' 是空指针
705             return ((OS_FLAGS)0);               //返回0
706         }
707     #endif
708     OS_ENTER_CRITICAL();                  //打开中断
709     flags = pgrp->OSFlagFlags;             //获取事件标志组的当前的事件标志位值
710     OS_EXIT_CRITICAL();                   //关闭中断
711     *err = OS_NO_ERR;                     //成功调用
712     return (flags);                       //返回当前事件标志状态
713 }
714 #endif
715
716 /*$PAGE*/
717 /*
718 *****
719 *                SUSPEND TASK UNTIL EVENT FLAG(s) RECEIVED OR TIMEOUT OCCURS
720 *
721 * 描述: 这个函数是uC/OS-II的内部函数, 如果期望的事件标志位没有置位, 任务将被挂起, 直到事件
722 *        标志位置位或者等待超时。
723 *        此程序完成操作并将调用OSFlagBlock() 的任务添加到事件标志组的等待列表中
724 *
725 * 参数: pgrp    指向事件标志组的指针。建立事件标志组时(OSFlagCreate())得到该指针。
726 *
727 *        pnode   is a pointer to a structure which contains data about the task waiting for
728 *                event flag bit(s) to be set.
729 *
730 *        flags   指定需要检查的事件标志位。如果opt参数位OS_FLAG_SET, 那么事件标志组中对
731 *                应的事件标志位置位。例如, 如果置位事件标志组的事件标志0、4和5, 则需要把
732 *                FLAGS参数设置位0x31(bit 0 是最低位)。若opt参数为OS_FLAG_CLR, 那么事件标
733 *                志组中对应的事件标志为被清0。
734 *
735 *
736 *        wait_type 定义等待事件标志位的方式。可以分为以下4种:
737 *
738 *                OS_FLAG_WAIT_CLR_ALL    所有指定事件标志位清 (0);
739 *                OS_FLAG_WAIT_CLR_ANY    任意指定事件标志位清 (0);
740 *                OS_FLAG_WAIT_SET_ALL    所有指定事件标志位置 (1);
741 *                OS_FLAG_WAIT_SET_ANY    任意指定事件标志位置 (1)。
742 *
743 *        timeout   以时钟节拍数目的等待超时时限。如果在这一时限得不到事件, 任务将恢复执行。
744 *                timeout的值为0, 表示将无限期地等待事件。timeout的最大值是65535个时钟节
745 *                拍。timeout的值并不与时钟节拍同步, timeout计数器在下一个时钟节拍到来时
746 *                开始递减。在这里, 所谓下一个时钟节拍, 也就是立刻就到来了。
747 *
748 * 返回: 无
749 *
750 * 程序在: OS_FLAG.C中OSFlagPend()
751 *
752 * 注意: 这个程序是uC/OS-II内部的, 请不要调用它。
753 *****
754
755 static void OS_FlagBlock (OS_FLAG_GRP *pgrp, OS_FLAG_NODE *pnode, OS_FLAGS flags,
756 INT8U wait_type, INT16U timeout)
757 {
758     OS_FLAG_NODE *pnode_next;           //定义标志下一个节点(变量)
759
760

```

```

761     OSTCBCur->OSTCBStat      |= OS_STAT_FLAG;          //(将任务的状态字)处于FLAG状态0x20
762     OSTCBCur->OSTCBDly       = timeout;                /* Store timeout in task's TCB */
763 #if OS_TASK_DEL_EN > 0
764     OSTCBCur->OSTCBFlagNode   = pnode;                  //把标志节点保存到TCB中
765 #endif
766     pnode->OSFlagNodeFlags    = flags;                  //保存任务等待事件标志组的指定事件标志位
767     pnode->OSFlagNodeWaitType = wait_type;              //保存任务等待事件标志组的等待类型的信息
768     //把任务控制块指针保存到标志节点的任务控制块链接中
769     pnode->OSFlagNodeTCB      = (void *)OSTCBCur;
770     //事件标志组对应的所有标志节点都链接在一起, 保存在事件标志组的等待任务链表中
771     pnode->OSFlagNodeNext     = pgrp->OSFlagWaitList;
772     pnode->OSFlagNodePrev     = (void *)0;              //新增的标志节点被添加到双向链表的开始端
773     pnode->OSFlagNodeFlagGrp  = (void *)pgrp;
774     //事件标志组的指针被反向链接到标志节点的事件标志组指针中, 当删除一个任务时, 需要根据这个链接
775     //把被删除的任务从对应的事件标志组的等待任务列表删除。
776     pnode_next               = pgrp->OSFlagWaitList;
777     if (pnode_next != (void *)0) {                      //把前一个标志节点指针链接到新添加的标志节点
778         pnode_next->OSFlagNodePrev = pnode;              /* No, link in doubly linked list */
779     }
780     pgrp->OSFlagWaitList = (void *)pnode;
781     //等待任务列表的起始指针被变更为新添加的标志节点, 调用任务也不再处于就绪态
782     if ((OSRdyTbl[OSTCBCur->OSTCBY] &~ OSTCBCur->OSTCBBitX) == 0) {
783         OSRdyGrp &~ OSTCBCur->OSTCBBitY;
784     }
785 }
786
787 /*$PAGE*/
788 /*
789 ****
790 *                                     INITIALIZE THE EVENT FLAG MODULE
791 *
792 * 描述: This function is called by uC/OS-II to initialize the event flag module. Your application
793 *       MUST NOT call this function. In other words, this function is internal to uC/OS-II.
794 *
795 * 参数: 无
796 *
797 * 返回: 无
798 *
799 * 警告: You MUST NOT call this function from your code. This is an INTERNAL function to uC/OS-II.
800 ****
801 */
802
803 void OS_FlagInit (void)
804 {
805 #if OS_MAX_FLAGS == 1
806     OSFlagFreeList          = (OS_FLAG_GRP *)&OSFlagTbl[0];
807     OSFlagFreeList->OSFlagType = OS_EVENT_TYPE_UNUSED;
808     OSFlagFreeList->OSFlagWaitList = (void *)0;
809 #endif
810
811 #if OS_MAX_FLAGS >= 2
812     INT8U i;
813     OS_FLAG_GRP *pgrp1;
814     OS_FLAG_GRP *pgrp2;
815
816     pgrp1 = &OSFlagTbl[0];
817     pgrp2 = &OSFlagTbl[1];
818     for (i = 0; i < (OS_MAX_FLAGS - 1); i++) {          /* Init. list of free EVENT FLAGS */
819         pgrp1->OSFlagType = OS_EVENT_TYPE_UNUSED;
820         pgrp1->OSFlagWaitList = (void *)pgrp2;
821         pgrp1++;
822         pgrp2++;
823     }
824     pgrp1->OSFlagWaitList = (void *)0;
825     OSFlagFreeList       = (OS_FLAG_GRP *)&OSFlagTbl[0];
826 #endif
827 }
828
829 /*$PAGE*/
830 /*
831 ****
832 *                                     使等待事件标志的任务进入就绪态 MAKE TASK READY-TO-RUN, EVENT(s) OCCURRED
833 *
834 * 描述: 这个函数是 uC/OS-II内部函数。
835 *       该处理在uC/OS-II中是一个标准过程, 这个惟一的不同在于, 事件标志组中当一个任务等待的

```

```

837 *      事件标志发生后, 为该任务建立的OS_FLAG_NODE数据结构就没有用处了; 所以这里把这个任务
838 *      的OS_FLAG_NODE数据结构从等待任务链表中删除掉, 同时还会把这个OS_FLAG_NODE数据结构指
839 *      针, 从该任务的事件控制块中删除掉。
840 *
841 * 参数: pnode   标志节点is a pointer to a structure which contains data about the task
842 *           waiting forevent flag bit(s) to be set.
843 *
844 *      flags_rdy  contains the bit pattern of the event flags that cause the task to become
845 *           ready-to-run.
846 *           //定义一个"准备完毕"含量值
847 *
848 * 返回: 无
849 *
850 * 访问: 本函数在OS_FLAG.C的OSFlagsPost() 中
851 *
852 * 注意: 1) 即使任务等待的事件标志都发生了, 任务已经从事件标志组的等待任务链表中被删除了, 但
853 *        是这个任务可能由于其它的原因而不能进入就绪态;
854 *        2) 这个函数是uC/OS-II内部函数, 你应用的时候不要调用它。
855 *****
856 */
857
858 static BOOLEAN OS_FlagTaskRdy (OS_FLAG_NODE *pnode, OS_FLAGS flags_rdy)
859 {
860     OS_TCB    *ptcb;
861     BOOLEAN    sched;
862
863
864     ptcb       = (OS_TCB *)pnode->OSFlagNodeTCB;  //
865     ptcb->OSTCBDly = 0;
866     ptcb->OSTCBFlagsRdy = flags_rdy;
867     ptcb->OSTCBStat  &= ~OS_STAT_FLAG;
868     if (ptcb->OSTCBStat == OS_STAT_RDY) {          //
869         OSRdyGrp    |= ptcb->OSTCBBitY;
870         OSRdyTbl[ptcb->OSTCBBY] |= ptcb->OSTCBBitX;
871         sched        = TRUE;
872     } else {
873         sched        = FALSE;
874     }
875     OS_FlagUnlink(pnode);
876     return (sched);
877 }
878
879 /*$PAGE*/
880 /*
881 *****
882 *      从任务等待链表中删除 UNLINK EVENT FLAG NODE FROM WAITING LIST
883 *
884 * 描述: 这个函数是uC/OS-II内部函数, 从任务等待链表中删除。
885 *
886 * 参数: pnode   is a pointer to a structure which contains data about the task waiting for
887 *           event flag bit(s) to be set.
888 *
889 * 返回: 无
890 *
891 * 访问: OS_FlagTaskRdy() OS_FLAG.C
892 *        OSFlagPend()    OS_FLAG.C
893 *        OSTaskDel()     OS_TASK.C
894 *
895 * 注意: 1) This function assumes that interrupts are disabled.
896 *        2) 这个函数是uC/OS-II内部函数, 你应用的时候不要调用它。
897 *****
898 */
899 //在这种情况下, 调用OS_FlagUnlink()函数, 把这个OS_FLAG_NODE从事件标志组的等待任务链表中
900 //删除, 并且返回一个出错代码, 说明发生了等待超时。这段代码只是简单的将一个OS_FLAG_NODE
901 //从一个双向链表中删除
902 void OS_FlagUnlink (OS_FLAG_NODE *pnode)
903 {
904     OS_TCB    *ptcb;
905     OS_FLAG_GRP *pgrp;
906     OS_FLAG_NODE *pnode_prev;
907     OS_FLAG_NODE *pnode_next;
908
909
910     pnode_prev = pnode->OSFlagNodePrev; //定义指向链表中的后一个数据结构(链表的一个节点)
911     pnode_next = pnode->OSFlagNodeNext; //定义指向链表中的前一个数据结构(链表的一个节点)
912     if (pnode_prev == (OS_FLAG_NODE *)0) { //检查指向前一个节点的指针为NULL

```

```

913 //事件标志组的指针被反向链接到标志节点的事件标志组指针中，当删除一个任务时，需要根据
914 //这个链接把被删除的任务从对应的事件标志组的等待任务列表删除
915 pgrp = pnode->OSFlagNodeFlagGrp;
916 pgrp->OSFlagWaitList = (void *)pnode_next;
917 //如果即将被删除的节点是链表第一个节点，那么在该节点删除后，链表的指针表头指针应该指
918 //向下一个节点。
919 if (pnode_next != (OS_FLAG_NODE *)0) {
920     //如果被删除的节点确实是链接的第1个节点，那么紧接着的下一个节点的“后一个节点”，
921     //这个新的头节点的“前一个节点指针”将被更新为NULL。
922     pnode_next->OSFlagNodePrev = (OS_FLAG_NODE *)0;
923 }
924 } else {
925     //如果被删除的节点不是链表中的第1个节点，那么这个节点的前一个节点的“后一个节点指针”将
926     //指向即将被删除的节点的后一个节点
927     pnode_prev->OSFlagNodeNext = pnode_next;
928     if (pnode_next != (OS_FLAG_NODE *)0) {
929         pnode_next->OSFlagNodePrev = pnode_prev;
930         //同样，这个被删除的节点的后一个节点的“前一个节点指针”也要被更新为该即将被删除节点
931         //的前一个节点。
932     }
933 }
934 ptcb = (OS_TCB *)pnode->OSFlagNodeTCB;
935 #if OS_TASK_DEL_EN > 0
936 ptcb->OSTCBFlagNode = (void *)0; //在所有的4种情况下，都会把任务控制块中的OSTCBFlagNode
937 //指针重新赋值为NULL。因为任务得到了预期的事件标志，则OSFlagPend()
938 //函数也将退出，建立OS_FLAG_NODE数据结构也将不复存在。
939 #endif
940 }
941 #endif
942

```

```

1  /*
2  ****
3  *                                     uC/OS-II实时控制内核
4  *                                     互斥型信号量项管理
5  *
6  * 文    件: OS_MUTEX.C  包含主要互斥型信号量代码
7  * 作    者: Jean J. Labrosse
8  * 中文注解: 钟常慰  zhongcw @ 126.com  译注版本: 1.0  请尊重原版内容
9  ****
10 */
11
12 #ifndef OS_MASTER_FILE //是否已定义OS_MASTER_FILE主文件
13 #include "includes.h" //包含"includes.h"文件, 部分C语言头文件的汇总打包文件
14 #endif //定义结束
15
16 /*
17 ****
18 *                                     局部变量 (LOCAL CONSTANTS)
19 ****
20 */
21
22 #define OS_MUTEX_KEEP_LOWER_8  0x00FF //设定互斥型信号量低8位有效(相与)
23 #define OS_MUTEX_KEEP_UPPER_8  0xFF00 //设定互斥型信号量高8位有效(相与)
24
25 #define OS_MUTEX_AVAILABLE     0x00FF //置MUTEX的值为有效, 同时保存PIP值, 高8位有效(相或)
26
27
28 #if OS_MUTEX_EN > 0 //条件编译: 当OS_SEM_EN允许产生信号量程序代码
29 /*
30 ****
31 *                                     无等待地获取互斥型信号量(ACCEPT MUTUAL EXCLUSION SEMAPHORE)
32 *
33 * 描述: 检查互斥型信号量, 以判断某资源是否可以使用, 与 OSMutexPend()不同的是, 若资源不能使用,
34 *       则调用 OSMutexAccept()函数的任务并不被挂起, OSMutexAccept()仅查询状态。
35 *
36 * 参数: pevent  指向管理某资源的互斥型信号量。程序在建立mutex时, 得到该指针(参见 OSMutexCreate())
37 *
38 *       err      指向出错代码的指针, 为以下值之一:
39 *               OS_NO_ERR      调用成功;
40 *               OS_ERR_EVENT_TYPE 'pevent'不是指向mutex类型的指针;
41 *               OS_ERR_PEVENT_NULL 'pevent'是空指针;
42 *               OS_ERR_PEND_ISR   在中断服务子程序中调用 OSMutexAccept()。
43 *
44 * 返回: == 1  如果mutex有效, OSMutexAccept()函数返回1;
45 *       == 0  如果mutex被其他任务占用, OSMutexAccept()则返回0。
46 *
47 * 警告: 1、必须先建立mutex, 然后才能使用;
48 *       2、在中断服务子程序中不能调用 OSMutexAccept()函数;
49 *       3、如使用 OSMutexAccept()获取mutex的状态, 那么使用完共享资源后, 必须调用 OSMutexPost()
50 *          函数释放mutex
51 ****
52 */
53
54 #if OS_MUTEX_ACCEPT_EN > 0 //允许生成 OSSemAccept()函数
55 INT8U OSMutexAccept (OS_EVENT *pevent, INT8U *err) //无等待地获取互斥型信号量[任务不挂起](信号量指针、错误代码)
56 {
57     #if OS_CRITICAL_METHOD == 3 //中断函数被设定为模式3
58         OS_CPU_SR cpu_sr;
59     #endif
60
61     if (OSIntNesting > 0) { //当前中断嵌套 > 0时, 表示还有中断程序运行
62         *err = OS_ERR_PEND_ISR; //在中断服务子程序中调用 OSMutexAccept()
63         return (0); //返回Null
64     }
65
66     #if OS_ARG_CHK_EN > 0 //所有参数必须在指定的参数内
67     if (pevent == (OS_EVENT *)0) { //当互斥型信号量的指针为空(Null)
68         *err = OS_ERR_PEVENT_NULL; // 'pevent'是空指针
69         return (0); //返回Null
70     }
71
72     if (pevent->OSEventType != OS_EVENT_TYPE_MUTEX) { //当事件类型不是一个互斥型信号量
73         *err = OS_ERR_EVENT_TYPE; // 'pevent'不是指向mutex类型的指针
74         return (0); //返回Null
75     }
76
77     #endif
78
79     OS_ENTER_CRITICAL(); //关闭中断
80
81     //获得Mutex的值(0或1), OSEventCnt相与0x00ff后判断OSEventCnt低8为0xff

```

```

77                                     //如果Mutex(高8位PIP)有效
78                                     //将PIP保存到OSEventCnt的高8位(相与0xff00)
79                                     //把该任务的优先级写到OSEventCnt的低8位(相或OSTCBPrio)
80                                     //将Mutex的事件控制块ECB链接到该任务的任务控制块
81     if ((pevent->OSEventCnt & OS_MUTEX_KEEP_LOWER_8) == OS_MUTEX_AVAILABLE) {
82         pevent->OSEventCnt &= OS_MUTEX_KEEP_UPPER_8;
83         pevent->OSEventCnt |= OSTCBCur->OSTCBPrio;
84         pevent->OSEventPtr = (void *)OSTCBCur;
85         OS_EXIT_CRITICAL();                                     //打开中断
86         *err = OS_NO_ERR;                                       //调用成功
87         return (1);                                             //返回1, 表明mutex已经得到, 可以使用相应的共享资源
88     }
89     OS_EXIT_CRITICAL();                                     //打开中断
90     *err = OS_NO_ERR;                                       //调用成功
91     return (0);                                             //返回Null, 表明mutex无效, 不能使用
92 }
93 #endif
94
95 /*$PAGE*/
96 /*
97 ****
98 *                                     建立和初始化互斥型信号量(CREATE A MUTUAL EXCLUSION SEMAPHORE)
99 *
100 * 描述: 互斥型信号量mutual的建立和初始化. 在与共享资源打交道时, 使用mutex可以保证满足互斥条件.
101 *
102 * 参数: prio    优先级继承优先级(PIP). 当一个高优先级的任务想要得到某mutex, 而此时这个mutex却被
103 *              一个低优先级的任务占用时, 低优先级任务的优先级可以提升到PIP, 知道其释放共享资源.
104 *
105 *              err    指向出错代码的指针, 为以下值之一:
106 *                  OS_NO_ERR        调用成功mutex已被成功的建立;
107 *                  OS_ERR_CREATE_ISR 试图在中断服务子程序中建立mutex;
108 *                  OS_PRIO_EXIST    优先级为PIP的任务已经存在;
109 *                  OS_ERR_PEVENT_NULL 已经没有OS_EVENT结构可以使用的了;
110 *                  OS_PRIO_INVALID   定义的优先级非法, 其值大于OS_LOWEST_PRIO.
111 *
112 * 返回: 返回一个指针, 该指针指向分配给mutex的事件控制块. 如果得不到事件控制块, 则返回一个空指针.
113 *
114 * 注意: 1) 必须先建立mutex, 然后才能使用;
115 *        2) 必须确保优先级继承优先级. 即prio高于可能与相应共享资源打交道的任务中优先级最高的任
116 *            务的优先级. 例如有3个优先级分别为20, 25, 30的任务会使用mutex, 那么prio的值必须小于
117 *            20; 并且, 已经建立了任务没有占用这个优先级.
118 ****
119 */
120 //建立并初始化一个互斥型信号量(优先级继承优先级(PIP)、出错代码指针)
121 OS_EVENT *OSMutexCreate (INT8U prio, INT8U *err)
122 {
123     #if OS_CRITICAL_METHOD == 3                                     //中断函数被设定为模式3
124         OS_CPU_SR cpu_sr;
125     #endif
126     OS_EVENT *pevent;                                             //定义一个互斥型信号量变量
127
128
129     if (OSIntNesting > 0) {                                       //中断嵌套数>0时, 表示还有中断任务在运行
130         *err = OS_ERR_CREATE_ISR;                                 //试图在中断服务子程序中建立mutex
131         return ((OS_EVENT *)0);                                  //返回0
132     }
133     #if OS_ARG_CHK_EN > 0                                         //所有参数在指定的范围之内
134         if (prio >= OS_LOWEST_PRIO) {                            //当任务优先级大于等于最大优先级
135             *err = OS_PRIO_INVALID;                               //定义的优先级非法, 其值大于OS_LOWEST_PRIO
136             return ((OS_EVENT *)0);                              //返回0
137         }
138     #endif
139     OS_ENTER_CRITICAL();                                           //关闭中断
140     if (OSTCBPrioTbl[prio] != (OS_TCB *)0) {                     //确认优先级别未占用, 即就绪状态不为0
141         OS_EXIT_CRITICAL();                                       //打开中断
142         *err = OS_PRIO_EXIST;                                     //优先级为PIP的任务已经存在
143         return ((OS_EVENT *)0);                                  //返回0
144     }
145     OSTCBPrioTbl[prio] = (OS_TCB *)1;                             //否则优先级别已用, 即就绪状态为1
146     pevent = OSEventFreeList;                                     // 试从空余事件控制列表中得到一个控制块ECB
147     if (pevent == (OS_EVENT *)0) {                                //当控制块=0时
148         OSTCBPrioTbl[prio] = (OS_TCB *)0;                       //将优先级别就绪态清0
149         OS_EXIT_CRITICAL();                                       //打开中断
150         *err = OS_ERR_PEVENT_NULL;                               //错误为(已经没有OS_EVENT结构可以使用的了)
151         return (pevent);                                          //返回pevent指针
152     }
153     //空余事件控制列表指向下一个空余事件控制块(指针)

```



```

153     OSEventFreeList = (OS_EVENT *)OSEventFreeList->OSEventPtr;
154     OS_EXIT_CRITICAL(); //打开中断
155     pevent->OSEventType = OS_EVENT_TYPE_MUTEX; //事件类型=MUTEX类型
156     pevent->OSEventCnt = (prio << 8) | OS_MUTEX_AVAILABLE; //置MUTEX的值为有效, 同时保存PIP值
157     pevent->OSEventPtr = (void *)0; //指向消息指针为0, 没有等待这个MUTEX的任务

158     OS_EventWaitListInit(pevent); //调用初始化等待任务列表
159     *err = OS_NO_ERR; //调用成功mutex已被成功的建立
160     return (pevent); //返回对应的OS_EventWaitListInit值
161 }
162
163 /*$PAGE*/
164 /*
165 ****
166 * 删除互斥型信号量 (DELETE A MUTEX)
167 *
168 * 描述: 删除一个mutex。使用这个函数有风险, 因为多任务中其他任务可能还想用这个实际上已经被删除
169 * 了的mutex。使用这个函数时必须十分小心, 一般地说, 要删除一个mutex, 首先应删除可能会用到
170 * 这个mutex的所有任务。
171 *
172 * 参数: pevent 指向mutex的指针。应用程序建立mutex时得到该指针(参见OSMutexCreate())
173 *
174 * opt 该参数定义删除mutex的条件。:
175 *      opt == OS_DEL_NO_PEND 只能在已经没有任何任务在等待该mutex时, 才能删除;
176 *      opt == OS_DEL_ALWAYS 不管有没有任务在等待这个mutex, 立刻删除mutex。
177 *      -->在第二种情况下, 所有等待mutex的任务都立即进入就绪态。
178 *
179 * err 指向出错代码的指针, 为以下值之一:
180 *      OS_NO_ERR 调用成功, mutex删除成功;
181 *      OS_ERR_DEL_ISR 试图在中断服务子程序中删除mutex。
182 *      OS_ERR_INVALID_OPT 定义的opt参数无效, 不是上面提到的2个参数之一;
183 *      OS_ERR_TASK_WAITING 定义了OS_DEL_NO_PEND, 而有一个或一个以上的任务在等这个mutex。
184 *      OS_ERR_EVENT_TYPE 'pevent'不是指向mutex的指针;
185 *      OS_ERR_PEVENT_NULL 已经没有可以使用的OS_EVENT数据结构了。
186 *
187 * 返回: pevent 如果mutex已经删除, 则返回空指针; 如果mutex没能删除, 则返回pevent。
188 *      在后一种情况下, 程序应检查出错代码, 以查出原因。
189 *
190 * 注意: 1) 使用这个函数时必须十分小心, 因为其他任务可能会用到mutex。
191 *      这个的所有任务。
192 ****
193 */
194
195 #if OS_MUTEX_DEL_EN //允许生成 MutexDel() 代码
196 OS_EVENT *OSMutexDel (OS_EVENT *pevent, INT8U opt, INT8U *err)
197 { //删除互斥型信号量(信号指针、删除条件、错误指针)
198     #if OS_CRITICAL_METHOD == 3 //中断函数被设定为模式3
199         OS_CPU_SR cpu_sr;
200     #endif
201     BOOLEAN tasks_waiting; //定义布尔量, 任务等待条件
202     INT8U pip; //定义优先级继承优先级
203
204     if (OSIntNesting > 0) { //中断嵌套数 > 0时, 表示还有中断任务在运行
205         *err = OS_ERR_DEL_ISR; //试图在中断服务子程序中删除mutex
206         return (pevent); //返回pevent指针
207     }
208     #if OS_ARG_CHK_EN > 0
209     if (pevent == (OS_EVENT *)0) { //所有参数在指定的范围之内
210         *err = OS_ERR_PEVENT_NULL; //已经没有可以使用的OS_EVENT数据结构了
211         return ((OS_EVENT *)0); //返回空值0
212     }
213     if (pevent->OSEventType != OS_EVENT_TYPE_MUTEX) { //当事件标志不是mutex类型时
214         *err = OS_ERR_EVENT_TYPE; // 'pevent'不是指向mutex的指针;
215         return (pevent); //返回pevent指针
216     }
217     #endif
218     OS_ENTER_CRITICAL(); //关闭中断
219     if (pevent->OSEventGrp != 0x00) { //当事件就绪表中有任务在等待该mutex
220         tasks_waiting = TRUE; //任务等待标志为(真)
221     } else {
222         tasks_waiting = FALSE; //否则, 该任务等待标志为(假)
223     }
224     switch (opt) { //opt设定选项, 删除条件
225     case OS_DEL_NO_PEND: // 1) 只能在已经没有任何任务在等待该mutex时, 才能删除
226         if (tasks_waiting == FALSE) { //没有任务在等待这个mutex

```



```

228         pip                = (INT8U) (pevent->OSEventCnt >> 8); //优先级继承优先级
229         OSTCBPrioTbl[pip]   = (OS_TCB *)0; //任务控制块优先级表pip为空
230         pevent->OSEventType = OS_EVENT_TYPE_UNUSED; //事件类型=空闲状态
231         pevent->OSEventPtr  = OSEventFreeList; //指向消息的指针=当前空余事件指针
232         OSEventFreeList    = pevent; //空余事件列表等于被删除的指针
233         OS_EXIT_CRITICAL(); //打开中断
234         *err = OS_NO_ERR; //调用成功, mutex删除成功
235         return ((OS_EVENT *)0); //返回空指针0
236     } else {
237         OS_EXIT_CRITICAL(); //打开中断
238         *err = OS_ERR_TASK_WAITING; //有一个或一个以上的任务在等这个mutex.
239         return (pevent); //返回pevent指针
240     }
241
242     case OS_DEL_ALWAYS: // 2) 多任务等待, 尽管有任务在等待, 还是要删除
243         while (pevent->OSEventGrp != 0x00) { //等待标志≠0, 还是要删除
244             //OS_EventTaskRdy() 函数将最高级优先级任务从等待列表中删除
245             OS_EventTaskRdy(pevent, (void *)0, OS_STAT_MUTEX); //使一个任务进入就绪态
246         }
247         pip                = (INT8U) (pevent->OSEventCnt >> 8); //优先级继承优先级
248         OSTCBPrioTbl[pip]   = (OS_TCB *)0; //任务控制块优先级表pip为空
249         pevent->OSEventType = OS_EVENT_TYPE_UNUSED; //事件类型=空闲状态
250         pevent->OSEventPtr  = OSEventFreeList; //指向消息的指针=当前空余事件指针
251         OSEventFreeList    = pevent; //空余事件列表等于被删除的指针
252         OS_EXIT_CRITICAL();
253         if (tasks_waiting == TRUE) { //有任务在等待这个mutex
254             OS_Sched(); //调用调度函数, 最高优先级任务运行
255         }
256         *err = OS_NO_ERR; //调用成功, mutex删除成功
257         return ((OS_EVENT *)0); //返回空指针0
258
259     default: // 3) 两个条件都不是
260         OS_EXIT_CRITICAL(); //打开中断
261         *err = OS_ERR_INVALID_OPT; //指定的opt无效不是指定的OS_DEL_NO_PEND和OS_DEL_ALWAYS
262         return (pevent); //返回pevent指针
263     }
264 }
265 #endif
266
267 /*$PAGE*/
268 /*
269 *****
270 * 等待一个互斥型信号量(挂起) (PEND ON MUTUAL EXCLUSION SEMAPHORE)
271 *
272 * 描述: 当任务需要独占共享资源时, 应使用OSMutexPend()函数. 如果任务在调用本函数时共享资源可
273 * 以使用, 则OSMutexPend()函数返回, 调用OSMutexPend()函数的任务得到了mutex.
274 *
275 * 注意: OSMutexPend()实际上并没有“给”调用本函数的任务什么值, 只不过参数err的值被置为
276 * OS_NO_ERR, 调用本函数的任务好像得到了mutex并继续运行.
277 * ---> 然而, 如果mutex已经被别的任务占用了, 那么OSMutexPend()函数就将调用该函数的任务放入
278 * 等待mutex的任务列表中, 这个任务于是进入了等待状态, 直到占有mutex的任务释放了mutex以
279 * 及共享资源, 或者直到定义的等待时限超时. 如果在等待时限内mutex得以释放, 那么ucos_ii恢
280 * 复运行等待mutex的任务中优先级最高的任务.
281 * 注意: 如果mutex被优先级较低的任务占用了, 那么OSMutexPend()会将占用mutex的任务的优先级提升
282 * 到优先级继承优先级PIP. PIP是在mutex建立时定义的(参见OSMutexCreate())
283 *
284 * 参数: pevent 指向mutex的指针. 应用程序在建立mutex时得到该指针的(参见OSMutexCreate())
285 *
286 * timeout 以时钟节拍数目的等待超时时限. 如果在这一时限得不到mutex, 任务将恢复执行.
287 * timeout的值为0, 表示将无限期地等待mutex. timeout的最大值是65535个时钟节
288 * 拍. timeout的值并不与时钟节拍同步, timeout计数器在下一个时钟节拍到来时
289 * 开始递减. 在这里, 所谓下一个时钟节拍, 也就是立刻就到来了.
290 *
291 * err 指向出错代码的指针, 为以下值之一:
292 * OS_NO_ERR 调用成功, mutex可以使用;
293 * OS_TIMEOUT 在定义的时间限内得不到mutex;
294 * OS_ERR_EVENT_TYPE 用户没能向OSMutexPend()传递指向mutex的指针;
295 * OS_ERR_PEVENT_NULL 'pevent' 是空指针
296 * OS_ERR_PEND_ISR 试图在中断服务子程序中获得mutex.
297 *
298 * 返回: 无
299 *
300 * 注意: 1) 必须先建立mutex, 然后才能使用;
301 * 2) 不要将占用mutex的任务挂起, 也不要让占有mutex的任务等待ucos_ii提供的信号量、邮箱及消
302 * 息队列等, 不要将占用mutex的任务延迟. 换言之, 用户代码应该抓紧时间, 尽量快地释放共享资源.
303 *****

```

```

304 */
305 void OSMutexPend (OS_EVENT *pevent, INT16U timeout, INT8U *err)
306 {
307     #if OS_CRITICAL_METHOD == 3
308         OS_CPU_SR cpu_sr;
309     #endif
310     INT8U pip; //定义mutex中的PIP
311     INT8U mprio; //定义mutex的优先级
312     BOOLEAN rdy; //布尔量rdy
313     OS_TCB *ptcb; //定义mutex的任务控制块指针
314
315     if (OSIntNesting > 0) { //中断嵌套数>0时, 表示还有中断任务在运行
316         *err = OS_ERR_PEND_ISR; //试图在中断服务子程序中获得mutex
317         return; //返回
318     }
319
320     #if OS_ARG_CHK_EN > 0
321     if (pevent == (OS_EVENT *)0) { //所有参数在指定的范围之内
322         *err = OS_ERR_PEVENT_NULL; //pevent=0
323         return; //返回
324     }
325     if (pevent->OSEventType != OS_EVENT_TYPE_MUTEX) { //当事件类型不否是mutex类型
326         *err = OS_ERR_EVENT_TYPE; //用户没能向OSMutexPend() 传递指向mutex的指针
327         return; //返回
328     }
329     #endif
330     OS_ENTER_CRITICAL(); //关闭中断
331     //OSEventCnt: 高8位是PIP值, 低8位是无任占用务时为0xFF值, 有任务占用时为任务优先级
332     //如果OSEventCnt低8位=0xFF
333     if ((INT8U)(pevent->OSEventCnt & OS_MUTEX_KEEP_LOWER_8) == OS_MUTEX_AVAILABLE) {
334         pevent->OSEventCnt &= OS_MUTEX_KEEP_UPPER_8; //计数器=低8位
335         pevent->OSEventCnt |= OSTCBCur->OSTCBPrio; //计数器低8位=调用该函数任务优先级
336         pevent->OSEventPtr = (void *)OSTCBCur; //指针指向调用该函数任务控制块TCB
337         OS_EXIT_CRITICAL(); //打开中断
338         *err = OS_NO_ERR; //调用成功, mutex可以使用
339         return; //返回
340     }
341     pip = (INT8U)(pevent->OSEventCnt >> 8); //提取mutex中的PIP
342     mprio = (INT8U)(pevent->OSEventCnt & OS_MUTEX_KEEP_LOWER_8); //提取mutex的优先级
343     ptcb = (OS_TCB *) (pevent->OSEventPtr); //占用mutex的任务控制块指针
344     //当前任务优先级不等于占用mutex优先级并且占用mutex的优先级 > 当前运行的任务优先级
345     if (ptcb->OSTCBPrio != pip && mprio > OSTCBCur->OSTCBPrio) {
346         //确认占用mutex的任务是否进入就绪态
347         if ((OSRdyTbl[ptcb->OSTCBY] & ptcb->OSTCBBitX) != 0x00) {
348             //如果该任务处于就绪态, 那么这个任务已不是处在它原来优先级上的就绪态,
349             if ((OSRdyTbl[ptcb->OSTCBY] & ~ptcb->OSTCBBitX) == 0x00) {
350                 OSRdyGrp &= ~ptcb->OSTCBBitY;
351             }
352             rdy = TRUE; //置rdy标志, 可以运行, 占用Mutex的任务进入就绪状态
353         } else {
354             rdy = FALSE; //否则, 清rdy标志
355         }
356         ptcb->OSTCBPrio = pip; //当前任务控制块优先级=提取Mutex的PIP
357         ptcb->OSTCBY = ptcb->OSTCBPrio >> 3; //取高3位优先级的值
358         ptcb->OSTCBBitY = OSMaTbl[ptcb->OSTCBY]; //对应的高3位OSMaTbl值
359         ptcb->OSTCBX = ptcb->OSTCBPrio & 0x07; //取低3位优先级的值
360         ptcb->OSTCBBitX = OSMaTbl[ptcb->OSTCBX]; //对应的低3位OSMaTbl值
361         if (rdy == TRUE) { //当rdy=1时,
362             OSRdyGrp |= ptcb->OSTCBBitY; //保存任务就绪标准0-7到OSRdyGrp
363             OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX; //保存任务优先级0-7到OSRdyTbl[]
364         }
365         OSTCBPrioTbl[pip] = (OS_TCB *)ptcb; //确认占用mutex的任务是否PIP优先级进入就绪态
366     }
367     OSTCBCur->OSTCBStat |= OS_STAT_MUTEX; //让任务控制块中的状态标志置位, 标明任务等待mutex而挂起
368     OSTCBCur->OSTCBDly = timeout; //等待超时参数也保存在任务控制块中
369     OS_EventTaskWait(pevent); //让任务进入休眠状态
370     OS_EXIT_CRITICAL(); //打开中断
371     OS_Sched(); //任务调度
372     OS_ENTER_CRITICAL(); //关闭中断
373     if (OSTCBCur->OSTCBStat & OS_STAT_MUTEX) { //检查任务控制块状态
374         OS_EventTO(pevent); //
375         OS_EXIT_CRITICAL(); //打开中断
376         *err = OS_TIMEOUT; //以时钟节拍数目的等待超时时限
377         return; //返回Null
378     }
379     OSTCBCur->OSTCBEvtPtr = (OS_EVENT *)0; //指向信号指针=0

```

```

380     OS_EXIT_CRITICAL(); //打开中断
381     *err = OS_NO_ERR; //返回调用成功, mutex可以使用
382 }
383 /*$PAGE*/
384 /*
385 *****
386 *          释放一个互斥型信号量(POST TO A MUTUAL EXCLUSION SEMAPHORE)
387 *
388 * 描述: 调用OSMutexPost()可以发出mutex。只是当用户程序已调用OSMutexAccept()或OSMutexPend()请
389 *        求得mutex时, OSMutexPost()函数才起作用。当优先级较高的任务试图得到mutex时, 如果占用
390 *        mutex的任务的优先级已经被升高, 那么OSMutexPost()函数使优先级升高了的任务恢复原来的优
391 *        先级。如果有一个以上的任务在等待这个mutex, 那么等待mutex的任务中优先级最高的任务将得
392 *        到mutex。然后本函数会调用调度函数, 看被唤醒的任务是不是进入就绪态任务中优先级最高的
393 *        任务。如果是, 则做任务切换, 让这个任务运行。如果没有等待mutex的任务, 那么本函数只不过
394 *        是将mutex的值设为0xFF, 表示mutex可以使用。
395 *
396 * 参数: pevent 指向mutex的指针。应用程序在建立mutex时得到该指针的(参见OSMutexCreate())
397 *
398 * 返回: OS_NO_ERR          调用成功, mutex被释放;
399 *        OS_ERR_EVENT_TYPE OSMutexPost()传递的不是指向mutex的指针;
400 *        OS_ERR_PEVENT_NULL 'pevent'是空指针;
401 *        OS_ERR_POST_ISR    试图在中断服务子程序中调用OSMutexPost()函数;
402 *        OS_ERR_NOT_MUTEX_OWNER 发出mutex的任务实际上并不占用mutex。
403 *
404 * 注意: 1) 必须先建立mutex, 然后才能使用;
405 *        2) 在中断服务子程序中不能调用OSMutexPost()函数
406 *****
407 */
408
409 INT8U OSMutexPost (OS_EVENT *pevent) //释放一个互斥型信号量(互斥型信号量指针)
410 {
411     #if OS_CRITICAL_METHOD == 3 //中断函数被设定为模式3
412         OS_CPU_SR cpu_sr;
413     #endif
414     INT8U pip; //定义mutex中的PIP
415     INT8U prio; //定义当前mutex的事件优先级
416
417     if (OSIntNesting > 0) { //中断嵌套数>0时, 表示还有中断任务在运行
418         return (OS_ERR_POST_ISR); //返回(试图在中断服务子程序中调用OSMutexPost()函数)
419     }
420
421     #if OS_ARG_CHK_EN > 0 //所有参数在指定的范围之内
422     if (pevent == (OS_EVENT *)0) { //pevent=0
423         return (OS_ERR_PEVENT_NULL); // 'pevent'是空指针
424     }
425     if (pevent->OSEventType != OS_EVENT_TYPE_MUTEX) { //当事件类型是否是mutex类型
426         return (OS_ERR_EVENT_TYPE); //OSMutexPost()传递的不是指向mutex的指针
427     }
428     #endif
429     OS_ENTER_CRITICAL(); //关闭中断
430     //OSEventCnt: 高8位是PIP值, 低8位是无任务占用时为0xFF值, 有任务占用时为任务优先级
431     pip = (INT8U)(pevent->OSEventCnt >> 8); //提取mutex的PIP
432     prio = (INT8U)(pevent->OSEventCnt & OS_MUTEX_KEEP_LOWER_8); //拾取mutex的优先级
433     //OSMutexPost()确认, 释放mutex的任务确实占用mutex的任务, 占用mutex的任务的优先级:
434     //或者是被提升到PIP(OSMutexpend()函数已经将该任务的优先级升高);
435     //或仍然是保持在mutex之中的优先级。
436     if (OSTCBCur->OSTCBPrio != pip && //任务的优先级是否=当前任务mutex的PIP, 并且
437         OSTCBCur->OSTCBPrio != prio) { //任务的优先级是否=当前mutex事件优先级
438         OS_EXIT_CRITICAL(); //打开中断
439         return (OS_ERR_NOT_MUTEX_OWNER); //发出mutex的任务实际上并不占用mutex
440     }
441     //查看占用mutex的任务优先级是否已经上升到了PIP, 因为有个高优先级的任务也需要这个mutex。
442     //在这种情况下, 占用mutex的任务优先级降到原来的优先级(从OSEventCnt低8为得到)
443     if (OSTCBCur->OSTCBPrio == pip) {
444         //将调用本函数的任务从任务就绪表中pip位置上删除, 放回到任务就绪表原来的优先级位置上
445         if ((OSRdyTbl[OSTCBCur->OSTCBY] & ~OSTCBCur->OSTCBBitX) == 0) {
446             OSRdyGrp &= ~OSTCBCur->OSTCBBitY;
447         }
448         OSTCBCur->OSTCBPrio = prio; //当前任务块优先级=当前mutex的优先级
449         OSTCBCur->OSTCBY = prio >> 3; //取高3位优先级的值
450         OSTCBCur->OSTCBBitY = OSMMapTbl[OSTCBCur->OSTCBY]; //对应的高3位OSMapTbl[]表值
451         OSTCBCur->OSTCBX = prio & 0x07; //取低3位优先级的值
452         OSTCBCur->OSTCBBitX = OSMMapTbl[OSTCBCur->OSTCBX]; //对应低3位OSMapTbl[]表值
453         OSRdyGrp |= OSTCBCur->OSTCBBitY; //保存任务就绪标准0-7到OSRdyGrp
454         OSRdyTbl[OSTCBCur->OSTCBY] |= OSTCBCur->OSTCBBitX; //保存任务优先级0-7到OSRdyTbl[]
455         //任务控制块优先级表=指向正在运行任务控制块的指针

```

```

456     OSTCBPrioTbl[prio] = (OS_TCB *)OSTCBCur;
457 }
458 OSTCBPrioTbl[pip] = (OS_TCB *)1; //确认占用mutex的任务是否PIP优先级进入就绪态
459 if (pevent->OSEventGrp != 0x00) { //查看是否有正在等待mutex的任务, 不为0表示有
460     //将最高级的任务从等待mutex的任务列表中删除(OS_EventTaskRdy() 使一个任务进入就绪态)
461     prio = OS_EventTaskRdy(pevent, (void *)0, OS_STAT_MUTEX);
462     //OSEventCnt: 高8位是PIP值, 低8位是无任务占用时为0xFF值, 有任务占用时为任务优先级
463     pevent->OSEventCnt &= OS_MUTEX_KEEP_UPPER_8; //新占用mutex的任务保存高8位(PIP)
464     pevent->OSEventCnt |= prio; //保存优先级
465     pevent->OSEventPtr = OSTCBPrioTbl[prio]; //mutex指针保存新任务控制块优先级
466     OS_EXIT_CRITICAL(); //打开中断
467     OS_Sched(); //进入调度任务, 使就绪态优先级最高任务运行
468     return (OS_NO_ERR); //返回调用成功, mutex被释放
469 }
470 //如果没有等待mutex的任务, 则OSEventCnt的低8位为0xFF, 表明mutex有效, 立即可以
471 pevent->OSEventCnt |= OS_MUTEX_AVAILABLE;
472 pevent->OSEventPtr = (void *)0; //mutex的指针在=0
473 OS_EXIT_CRITICAL(); //打开中断
474 return (OS_NO_ERR); //返回调用成功, mutex被释放
475 }
476 /*$PAGE*/
477 /*
478 ****得到mutex当前状态信息(QUERY A MUTUAL EXCLUSION SEMAPHORE)
479 *
480 * 描述: 得到mutex当前状态信息。应用程序必须给OS_MUTEX_DATA数据结构分配存储空间, 这个数据结构用
481 * 于接受来自mutex的事件控制块的数据。通过调用OSMutexQuery() 函数, 得知
482 * mutex。计算在.OSEventTbl[]中有几个任务在等待mutex(计算有几个1), 得到PIP的值, 以及确认
483 * mutex是否可以使用(是1还是0)。
484 *
485 * 参数: pevent 指向管理某资源的互斥型信号量。程序在建立mutex时, 得到该指针(参见OSMutexCreate())
486 *
487 * pdata 指向类型为OS_MUTEX_DATA的数据结构的指针。这个数据结构包括以下域:
488 *      INT8U OSMutexPIP; // mutex的优先级继承优先级PIP;
489 *      INT8U OSOwnerPrio; // 占用mutex任务的优先级
490 *      INT8U OSValue; // 当前mutex的值。1表示可以使用, 0表示不能使用;
491 *      INT8U OSEventGrp; // 复制等待mutex的任务列表。
492 *      INT8U OSEventTbl[OS_EVENT_TBL_SIZE] //容量大小由ucos_ii.H
493 *
494 * 返回: OS_NO_ERR 调用成功;
495 *      OS_ERR_QUERY_ISR 试图在中断子程序中调用OSMutexQuery();
496 *      OS_ERR_PEVENT_NULL 'pevent' 是空指针;
497 *      OS_ERR_EVENT_TYPE OSMutexQuery() 不是指向mutex的指针。
498 *
499 * 注意: 1) 必须先建立mutex, 然后才能使用;
500 *      2) 在中断服务子程序中不能调用OSMutexPost() 函数。
501 *
502 ****
503 */
504
505 #if OS_MUTEX_QUERY_EN > 0 //允许生成 OSMutexQuery() 代码
506 INT8U OSMutexQuery (OS_EVENT *pevent, OS_MUTEX_DATA *pdata)
507 { //查询一个互斥型信号量的当前状态(互斥型信号量指针、状态数据结构指针)
508 #if OS_CRITICAL_METHOD == 3 //中断函数被设定为模式3
509     OS_CPU_SR cpu_sr;
510 #endif
511     INT8U *psrc; //定义8位pevent->OSEventTbl[0]的地址指针
512     INT8U *pdest; //定义8位pdata->OSEventTbl[0]的地址指针
513
514     if (OSIntNesting > 0) { //中断嵌套数>0时, 表示还有中断任务在运行
515         return (OS_ERR_QUERY_ISR); //错误等于(试图在中断子程序中调用OSMutexQuery())
516     }
517 #if OS_ARG_CHK_EN > 0 //所有参数在指定的范围之内
518     if (pevent == (OS_EVENT *)0) { //当互斥型信号量指针为NULL, 即0(空)
519         return (OS_ERR_PEVENT_NULL); //event是空指针
520     }
521     if (pevent->OSEventType != OS_EVENT_TYPE_MUTEX) { //当事件类型是否是互斥型信号量类型
522         return (OS_ERR_EVENT_TYPE); //OSMutexQuery() 不是指向互斥型信号量的指针
523     }
524 #endif
525 #endif
526     OS_ENTER_CRITICAL(); //关闭中断
527 //将事件(互斥型信号量)结构中的等待任务列表复制到pdata数据结构中, 计算在.OSEventTbl[]中有几个任务
528 //在等待mutex(计算有几个1), 得到PIP的值, 以及确认mutex是否可以使用(是1还是0)
529     pdata->OSMutexPIP = (INT8U) (pevent->OSEventCnt >> 8);
530     pdata->OSOwnerPrio = (INT8U) (pevent->OSEventCnt & OS_MUTEX_KEEP_LOWER_8);
531     //保存mutex的优先级继承优先级PIP

```

```

532                                     //保存占用mutex任务的优先级
533     if (pdata->OSOwnerPrio == 0xFF) { //如果占用mutex任务的优先级为255时
534         pdata->OSValue = 1;           //当前mutex的值。1表示可以使用。
535     } else {                           //否则
536         pdata->OSValue = 0;           //当前mutex的值。0表示不能使用。
537     }                                 //将事件(互斥型信号量)结构中的等待任务列表复制到pdata数
                                     据结构中
538     pdata->OSEventGrp = pevent->OSEventGrp; //等待事件的任务组中的内容传送到状态数据结构中
539     psrc              = &pevent->OSEventTbl[0]; //保存pevent->OSEventTbl[0]对应的地址
540     pdest             = &pdata->OSEventTbl[0]; //保存pdata->OSEventTbl[0]对应的地址
541     #if OS_EVENT_TBL_SIZE > 0             //当事件就绪对应表中的对应值>0时
542         *pdest++ = *psrc++;               //地址指针下移一个类型地址, 获取互斥型信号量的值
543     #endif
544
545     #if OS_EVENT_TBL_SIZE > 1             //事件就绪对应表中的对应值>1时
546         *pdest++ = *psrc++;               //地址指针继续下移一个类型地址, 获取互斥型信号量的值
547     #endif
548
549     #if OS_EVENT_TBL_SIZE > 2             //事件就绪对应表中的对应值>2时
550         *pdest++ = *psrc++;               //地址指针继续下移一个类型地址, 获取互斥型信号量的值
551     #endif
552
553     #if OS_EVENT_TBL_SIZE > 3             //事件就绪对应表中的对应值>3时
554         *pdest++ = *psrc++;               //地址指针继续下移一个类型地址, 获取互斥型信号量的值
555     #endif
556
557     #if OS_EVENT_TBL_SIZE > 4             //事件就绪对应表中的对应值>4时
558         *pdest++ = *psrc++;               //地址指针继续下移一个类型地址, 获取互斥型信号量的值
559     #endif
560
561     #if OS_EVENT_TBL_SIZE > 5             //事件就绪对应表中的对应值>5时
562         *pdest++ = *psrc++;               //地址指针继续下移一个类型地址, 获取互斥型信号量的值
563     #endif
564
565     #if OS_EVENT_TBL_SIZE > 6             //事件就绪对应表中的对应值>6时
566         *pdest++ = *psrc++;               //地址指针继续下移一个类型地址, 获取互斥型信号量的值
567     #endif
568
569     #if OS_EVENT_TBL_SIZE > 7             //事件就绪对应表中的对应值>7时
570         *pdest = *psrc;                   //获取最后地址的互斥型信号量的值
571     #endif
572     OS_EXIT_CRITICAL();                  //打开中断
573     return (OS_NO_ERR);                  //返回成功运行
574 }
575 #endif //OS_SEM_QUERY_EN函数结束
576 #endif //OS_MUTEX_EN文件结束
577

```



```

1  /*
2  ****
3  *                                     uC/OS-II实时控制内核
4  *                                     消息邮箱管理
5  * 文 件 : OS_MBOX.C      消息邮件管理代码
6  * 作 者 : Jean J. Labrosse
7  * 中文注解: 钟常慰  zhongcw @ 126.com  译注版本: 1.0 请尊重原版内容
8  ****
9  */
10
11 #ifndef OS_MASTER_FILE                //是否已经定义OS_MASTER_FILE
12 #include "includes.h"                //包含"includes.h"文件
13 #endif                                //结束定义
14
15 #if OS_MBOX_EN > 0                    //条件编译允许(1)产生消息邮箱相关代码
16 /*
17 ****
18 *                                     查看指定的消息邮箱是否有需要的消息(ACCEPT MESSAGE FROM MAILBOX)
19 *
20 * 描述: OSMboxAccept() 函数查看指定的消息邮箱是否有需要的消息。不同于OSMboxPend() 函数, 如果没有需要的消息,
21 *       OSMboxAccept() 函数并不挂起任务。如果消息已经到达, 该消息被传递到用户任务并且从消息邮箱中清除。通
22 *       常中断调用该函数, 因为中断不允许挂起等待消息。
23 *
24 * 意见: pevent 是指向需要查看的消息邮箱的指针。当建立消息邮箱时, 该指针返回到用户程序。
25 *       (参考OSMboxCreate() 函数)。
26 *
27 * 返回: 如果消息已经到达, 返回指向该消息的指针; 如果消息邮箱没有消息, 返回空指针。
28 *
29 * 注意: 必须先建立消息邮箱, 然后使用。
30 ****
31 */
32
33 #if OS_MBOX_ACCEPT_EN > 0              //允许(1)生成 OSMboxAccept() 代码
34 void *OSMboxAccept (OS_EVENT *pevent) //查看消息邮箱(消息邮箱指针)
35 {
36     #if OS_CRITICAL_METHOD == 3        //中断函数被设定为模式3
37         OS_CPU_SR cpu_sr;
38     #endif
39     void *msg;                          //定义消息邮箱内容的指针
40
41
42     #if OS_ARG_CHK_EN > 0              //所有参数必须在指定的参数内
43         if (pevent == (OS_EVENT *)0) { //当消息邮箱指针为NULL时, 返回0, 空指针
44             return ((void *)0);
45         }
46         if (pevent->OSEventType != OS_EVENT_TYPE_MBOX) { //当事件类型≠消息邮箱类型
47             return ((void *)0);          //返回空指针
48         }
49     #endif
50     OS_ENTER_CRITICAL();                //关闭中断
51     msg = pevent->OSEventPtr;           //取消息邮箱中的内容
52     pevent->OSEventPtr = (void *)0;     //将消息邮箱的内容清0
53     OS_EXIT_CRITICAL();                 //打开中断
54     return (msg);                       //返回消息, 如果为空, 说明没有消息; 不为空, 说明有内容
55 }
56 #endif
57 /*$PAGE*/
58 /*
59 ****
60 *                                     建立并初始化一个消息邮箱(CREATE A MESSAGE MAILBOX)
61 *
62 * 描述: 建立并初始化一个消息邮箱。消息邮箱允许任务或中断向其他一个或几个任务发送消息。
63 *
64 * 参数: msg 参数用来初始化建立的消息邮箱。如果该指针不为空, 建立的消息邮箱将含有消息。
65 *
66 * 返回: 指向分配给所建立的消息邮箱的事件控制块的指针。如果没有可用的事件控制块, 返回空指针。
67 *
68 * 注意: 必须先建立消息邮箱, 然后使用。
69 ****
70 */
71
72 OS_EVENT *OSMboxCreate (void *msg)    //建立并初始化一个消息邮箱(msg 参数不为空含内容)
73 {
74     #if OS_CRITICAL_METHOD == 3        //中断函数被设定为模式3
75         OS_CPU_SR cpu_sr;
76     #endif

```

```

77 OS_EVENT *pevent; //定义一个指向事件控制块的指针
78
79
80 if (OSIntNesting > 0) { //中断嵌套数>0时，表示还有中断任务在运行
81     return ((OS_EVENT *)0); //返回0
82 }
83 OS_ENTER_CRITICAL(); //关闭中断
84 pevent = OSEventFreeList; //pevent=空余事件管理列表
85 if (OSEventFreeList != (OS_EVENT *)0) { //如果有空余事件管理块
86     OSEventFreeList = (OS_EVENT *)OSEventFreeList->OSEventPtr;
87 } //空余事件控制链表指向下一个空余事件控制块
88 OS_EXIT_CRITICAL(); //打开中断
89 if (pevent != (OS_EVENT *)0) { //如果有空余的事件控制块可用
90     pevent->OSEventType = OS_EVENT_TYPE_MBOX; //则这个类型=消息邮箱类型
91     pevent->OSEventPtr = msg; //将初始值存入事件管理块ECB中
92     OS_EventWaitListInit(pevent); //初始化一个事件控制块
93 }
94 return (pevent); //返回该详细邮箱(事件)的指针，即邮箱句柄
95 }
96 /*$PAGE*/
97 /*
98 ****
99 * 删除消息邮箱 (DELETE A MAIBOX)
100 *
101 * 描述：删除消息邮箱。因为多任务可能会试图继续使用已经删除了的邮箱，故调用本函数有风险。使用本函数
102 * 须特别小心。一般的说，删除邮箱之前，应该首先删除与本邮箱有关的任务。
103 *
104 * 描述：pevent 指向邮箱得指针。该指针是在邮箱建立时，返回给用户应用程序的指针。(参考OSMboxCreate())
105 *
106 * opt 该选项定义邮箱的删除条件：
107 * opt == OS_DEL_NO_PEND 可以选择只能在已经没有任何任务在等待该邮箱的消息时，才能删除邮箱；
108 * opt == OS_DEL_ALWAYS 不管有没有任务在等待邮箱的消息，立即删除邮箱。
109 * 一>第2种情况下，所有等待邮箱消息的任务都立即进入就绪态。
110 *
111 * err 指向错误代码的指针，返回出错代码可以是以下几种之一：
112 * OS_NO_ERR 调用成功，邮箱已经删除；
113 * OS_ERR_DEL_ISR 试图在中断服务子程序中删除邮箱；
114 * OS_ERR_INVALID_OPT 无效的opt参数，用户没有将opt定义为上述2种情况之一；
115 * OS_ERR_TASK_WAITING 一个或更多的任务在等待邮箱的消息；
116 * OS_ERR_EVENT_TYPE pevent不是指向邮箱的指针；
117 * OS_ERR_PEVENT_NULL 已经没有OS_EVENT数据结构可以使用。
118 *
119 * 返回：pevent 返回空指针NULL，表示邮箱已被删除，返回pevent，表示邮箱没有删除，在这种情况下，应该进一步
120 * 查看出错代码，找到出错原因。
121 *
122 * 注意：1) 使用这个函数调用时，须特别小心。因为其他任务可能还要用这个邮箱。
123 * 2) 当挂起的任务进入就绪态时，中断是关闭的，这就意味着中断延迟时间与在等待邮箱的消息的任务数有关。
124 * 3) 调用OSMboxAccept()函数也不可能知道邮箱是否已经被删除了。
125 ****
126 */
127
128 #if OS_MBOX_DEL_EN > 0 //允许(1)生成 OSMboxDel()代码
129 //删除消息邮箱(消息邮箱指针、删除条件、出错代码指针)
130 OS_EVENT *OSMboxDel (OS_EVENT *pevent, INT8U opt, INT8U *err)
131 {
132     #if OS_CRITICAL_METHOD == 3 //中断函数被设定为模式3
133         OS_CPU_SR cpu_sr;
134     #endif
135     BOOLEAN tasks_waiting; //定义布尔量，任务等待条件
136
137
138     if (OSIntNesting > 0) { //中断嵌套数>0时，表示还有中断任务在运行
139         *err = OS_ERR_DEL_ISR; //错误等于(试图在中断程序中删除一个信号量事件)
140         return (pevent); //返回消息邮箱指针
141     }
142     #if OS_ARG_CHK_EN > 0 //所有参数在指定的范围之内
143     if (pevent == (OS_EVENT *)0) { //当消息邮箱指针为NULL，即0(空)
144         *err = OS_ERR_PEVENT_NULL; //错误等于(已经没有可用的OS_EVENT数据结构了)
145         return (pevent); //返回消息邮箱指针
146     }
147     if (pevent->OSEventType != OS_EVENT_TYPE_MBOX) { //当事件类型不否是消息邮箱类型
148         *err = OS_ERR_EVENT_TYPE; //pevent指针不是指向消息邮箱
149         return (pevent); //返回消息邮箱指针
150     }
151 #endif

```



```

152     OS_ENTER_CRITICAL(); //关闭中断
153     if (pevent->OSEventGrp != 0x00) { //事件等待标志, 索引值≠0, 有任务在等待
154         tasks_waiting = TRUE; //有任务在等待=1(TRUE真)
155     } else {
156         tasks_waiting = FALSE; //否则, 没有任务在等待=0, (FALSE假)
157     }
158     switch (opt) { //条件选择
159     case OS_DEL_NO_PEND: //1) 没有任务在等待该消息邮箱
160         if (tasks_waiting == FALSE) { //如果没有事件在等待
161             pevent->OSEventType = OS_EVENT_TYPE_UNUSED; //事件类型=空闲
162             pevent->OSEventPtr = OSEventFreeList; //信号量对应的指针=空余块链接表
163             OSEventFreeList = pevent; //空余块链接表=当前事件指针
164             OS_EXIT_CRITICAL(); //关闭中断
165             *err = OS_NO_ERR; //错误等于(成功删除)
166             return ((OS_EVENT *)0); //返回0
167         } else { //否则, 有任务在等待
168             OS_EXIT_CRITICAL(); //打开中断
169             *err = OS_ERR_TASK_WAITING; //错误等于(有一个或一个以上的任务在等待消息邮箱)
170             return (pevent); //返回消息邮箱指针
171         }
172     case OS_DEL_ALWAYS: //2) 多任务等待, 尽管有任务在等待, 还是要删除
173         while (pevent->OSEventGrp != 0x00) { //等待标志≠0, 还是要删除
174             //OS_EventTaskRdy() 函数将最高级优先级任务从等待列表中删除
175             OS_EventTaskRdy(pevent, (void *)0, OS_STAT_MBOX); //使一个任务进入就绪态
176         }
177         pevent->OSEventType = OS_EVENT_TYPE_UNUSED; //事件类型=空闲
178         pevent->OSEventPtr = OSEventFreeList; //消息邮箱对应的指针=空余块链接表
179         OSEventFreeList = pevent; //空余块链接表=当前事件指针
180         OS_EXIT_CRITICAL(); //关闭中断
181         if (tasks_waiting == TRUE) { //当任务等待=1, 真
182             OS_Sched(); //任务调度, 最高优先级进入运行状态
183         }
184         *err = OS_NO_ERR; //错误等于(成功删除)
185         return ((OS_EVENT *)0); //返回0
186     default: // 3) 当以上两种情况都不是
187         OS_EXIT_CRITICAL(); //关闭中断
188         *err = OS_ERR_INVALID_OPT; //错误等于(没有将opt参数定义为2种合法的参数之一)
189         return (pevent); //返回信号量指针
190     }
191 }
192 #endif
193
194 /**$PAGE*/
195 /**
196 *****
197 任务等待消息(PEND ON MAILBOX FOR A MESSAGE)
198 *****
199 *
200 * 描述: 用于任务等待消息。消息通过中断或另外的任务发送给需要的任务。
201 *
202 * 参数: pevent 是指向即将接受消息的消息邮箱的指针。该指针的值在建立该消息邮箱时可以得到。
203 *
204 * timeout 允许一个任务在经过了指定数目的时钟节拍后还没有得到需要的消息时恢复运行。如果该值为零表示任务将持续的等待消息。最大的等待时间为65, 535个时钟节拍。这个时间长度并不是非常严格的, 可能存在一个时钟节拍的误差, 因为只有在一个时钟节拍结束后才会减少定义的等待超时时钟节拍。
205 *
206 * err 是指向包含错误码的变量的指针。OSMboxPend() 函数返回的错误码可能为下述几种:
207 *
208 * OS_NO_ERR 消息被正确的接受;
209 * OS_TIMEOUT 消息没有在指定的周期数内送到;
210 * OS_ERR_EVENT_TYPE pevent 不是指向消息邮箱的指针;
211 * OS_ERR_PEND_ISR 从中断调用该函数。虽然规定了不允许从中断调用该函数, 但uC/OS-ii仍然包含了检测这种情况的功能;
212 * OS_ERR_PEVENT_NULL 'pevent' 是空指针。
213 *
214 * 返回: 返回接受的消息并将 *err置为OS_NO_ERR。如果没有在指定数目的时钟节拍内接收到需要的消息, OSMboxPend() 函数返回空指针并且将 *err设置为OS_TIMEOUT。
215 *
216 * 注意: 必须先建立消息邮箱, 然后使用。
217 * 不允许从中断调用该函数。
218 *****
219 */
220 //等待一个消息邮箱函数(消息邮箱指针、允许等待的时钟节拍、代码错误指针)
221 void *OSMboxPend (OS_EVENT *pevent, INT16U timeout, INT8U *err)
222 {

```

```

228 #if OS_CRITICAL_METHOD == 3           //中断函数被设定为模式3
229     OS_CPU_SR  cpu_sr;
230 #endif
231     void        *msg;                   //定义消息邮箱内容的指针
232
233
234     if (OSIntNesting > 0) {              //中断嵌套数>0时, 表示还有中断任务在运行
235         *err = OS_ERR_PEND_ISR;          //错误等于(试图在中断程序中等待一个消息邮箱事件)
236         return ((void *)0);              //返回
237     }
238 #if OS_ARG_CHK_EN > 0                   //所有参数在指定的范围之内
239     if (pevent == (OS_EVENT *)0) {       //当邮箱指针为NULL, 即0(空)
240         *err = OS_ERR_PEVENT_NULL;       //pevent是空指针
241         return ((void *)0);              //返回
242     }
243     if (pevent->OSEventType != OS_EVENT_TYPE_MBOX) { //当事件类型不否是消息邮箱类型
244         *err = OS_ERR_EVENT_TYPE;        //pevent指针不是指向消息邮箱
245         return ((void *)0);              //返回
246     }
247 #endif
248     OS_ENTER_CRITICAL();                 //关闭中断
249     msg = pevent->OSEventPtr;             //试取消息邮箱内容
250     if (msg != (void *)0) {               //检查消息邮箱是否为空
251         pevent->OSEventPtr = (void *)0;    //将0存入消息邮箱中
252         OS_EXIT_CRITICAL();               //打开中断
253         *err = OS_NO_ERR;                 //返回成功调用, 取出消息
254         return (msg);                     //返回接收消息
255     }                                     //如果消息邮箱为空, 则进入等待中
256     OSTCBCur->OSTCBStat |= OS_STAT_MBOX; //将任务状态置1, 进入睡眠状态, 只能通过消息邮箱唤醒
257     OSTCBCur->OSTCBDly = timeout;         //最长等待时间=timeout, 递减式
258     OS_EventTaskWait(pevent);             //使任务进入等待时间唤醒状态
259     OS_EXIT_CRITICAL();                   //打开中断
260     OS_Sched();                           //进入调度任务, 使就绪态优先级最高任务运行
261     OS_ENTER_CRITICAL();                 //关闭中断
262     msg = OSTCBCur->OSTCBMsg;             //接收消息=指向任务消息的指针
263     if (msg != (void *)0) {               //接收消息邮箱是否为空
264         OSTCBCur->OSTCBMsg = (void *)0;    //传递给消息的指针=空
265         OSTCBCur->OSTCBStat = OS_STAT_RDY; //表示任务处于就绪状态
266         OSTCBCur->OSTCBEventPtr = (OS_EVENT *)0; //指向事件控制块的指针=0
267         OS_EXIT_CRITICAL();               //打开中断
268         *err = OS_NO_ERR;                 //成功等待该消息邮箱
269         return (msg);                     //返回接收消息
270     }
271     OS_EventTO(pevent);                   //如果没有获得消息, 由于等待超时而返回
272     OS_EXIT_CRITICAL();                   //打开中断
273     *err = OS_TIMEOUT;                     //消息没有在指定的时间送到
274     return ((void *)0);                   //返回0
275 }
276 /*$PAGE*/
277 /*
278 ****
279 *           通过消息邮箱向任务发送消息(POST MESSAGE TO A MAILBOX)
280 *
281 * 描述: 通过消息邮箱向任务发送消息。消息是一个指针长度的变量, 在不同的程序中消息的使用也可能不同。如果
282 * 消息邮箱中已经存在消息, 返回错误码说明消息邮箱已满。OSMboxPost()函数立即返回调用者, 消息也没有
283 * 有能够发到消息邮箱。如果有任何任务在等待消息邮箱的消息, 最高优先级的任务将得到这个消息。如果等
284 * 待消息的任务优先级比发送消息的任务优先级高, 那么高优先级的任务将得到消息而恢复执行, 也就是说,
285 * 发生了一次任务切换。
286 *
287 * 参数: pevent 是指向即将接受消息的消息邮箱的指针。该指针的值在建立该消息邮箱时可以得到。
288 *       参考OSMboxCreate()函数
289 *
290 *       msg     是即将实际发送给任务的消息。消息是一个指针长度的变量, 在不同的程序中消息的使用也可能不
291 *       同。不允许传递一个空指针, 因为这意味着消息邮箱为空。
292 *
293 * 返回: OS_NO_ERR      消息成功的放到消息邮箱中;
294 *       OS_MBOX_FULL   消息邮箱已经包含了其他消息, 不空;
295 *       OS_ERR_EVENT_TYPE 'pevent' 不是指向消息邮箱的指针;
296 *       OS_ERR_PEVENT_NULL 'pevent' 是空指针。
297 *       OS_ERR_POST_NULL_PTR 用户试图发出空指针。根据规则, 在这里不支持空指针。
298 *
299 * 注意: 必须先建立消息邮箱, 然后使用。
300 *       不允许传递一个空指针, 因为这意味着消息邮箱为空。
301 ****
302 */
303

```

```

304 #if OS_MBOX_POST_EN > 0 //允许(1)生成 OSMboxPost() 代码
305 INT8U OSMboxPost (OS_EVENT *pevent, void *msg) //发送消息函数(消息邮箱指针、即将实际发送给任务的消息)
306 {
307     #if OS_CRITICAL_METHOD == 3 //中断函数被设定为模式3
308         OS_CPU_SR cpu_sr;
309     #endif
310
311
312     #if OS_ARG_CHK_EN > 0 //所有参数在指定的范围之内
313         if (pevent == (OS_EVENT *)0) { //当邮箱指针为NULL, 即0(空)
314             return (OS_ERR_PEVENT_NULL); //返回(pevent是空指针)
315         }
316         if (msg == (void *)0) { //检查消息是否为空
317             return (OS_ERR_POST_NULL_PTR); //返回(用户试图发出空指针)不支持空指针
318         }
319         if (pevent->OSEventType != OS_EVENT_TYPE_MBOX) { //事件发送标志是否是邮箱标志
320             return (OS_ERR_EVENT_TYPE); // 'pevent' 不是指向消息邮箱的指针
321         }
322     #endif
323     OS_ENTER_CRITICAL(); //关闭中断
324     if (pevent->OSEventGrp != 0x00) { //是否有任务在等待该邮箱, 索引值≠0
325         //OS_EventTaskRdy() 函数将最高级优先级任务从等待列表中删除
326         OS_EventTaskRdy(pevent, msg, OS_STAT_MBOX);
327         OS_EXIT_CRITICAL(); //关闭中断
328         OS_Sched(); //如果该任务不是最高优先级, 进入(任务调度)
329         return (OS_NO_ERR); //消息成功的放到消息邮箱中;
330     }
331     if (pevent->OSEventPtr != (void *)0) { //事件邮箱指针=空
332         OS_EXIT_CRITICAL(); //打开中断
333         return (OS_MBOX_FULL); //消息邮箱中已经包含了其它的消息(邮箱已满)
334     }
335     pevent->OSEventPtr = msg; //指向消息的指针保存到邮箱中
336     OS_EXIT_CRITICAL(); //打开中断
337     return (OS_NO_ERR); //消息成功的放到消息邮箱中
338 }
339 #endif
340
341 /*$PAGE*/
342 /*
343 ****
344 * 通过邮箱向(多)任务发送消息 (POST MESSAGE TO A MAILBOX)
345 *
346 * 描述: OSMboxPostOpt() 与 OSMboxPost() 相同, 只是允许用户程序发消息给多个任务。也就是允许将消息广播给
347 * 所有的等待邮箱消息的任务。OSMboxPostOpt() 实际上取代 OSMboxPost(), 因为它可仿真 OSMboxPost()。
348 *
349 * 通过邮箱向任务发送消息。消息是一个指针长度的变量, 在不同的程序中消息的使用也可能不同。如果
350 * 消息邮箱中已经存在消息, 返回错误码说明消息邮箱已满。OSMboxPostOpt() 函数立即返回调用者, 消息
351 * 也没有能够发到消息邮箱。如果有任何任务在等待消息邮箱的消息, 那么 OSMboxPostOpt() 允许选择2种
352 * 情况之一: 1、定义消息只发送给等待邮箱消息得任务中优先级最高得任务, 2、让所有等待邮箱消息得,
353 * 任务都得到消息无论在那种条件下, 如果得到消息的任务优先级比发送消息的任务优先级高, 那么得到
354 * 消息的最高优先级的任务将恢复执行, 发消息的任务将被挂起。也就是发生一次任务切换。
355 *
356 * 参数: pevent 是指向即将发送消息的消息邮箱的指针。该指针的值在建立该消息邮箱时可以得到。
357 * 参考 OSMboxCreate() 函数。
358 *
359 * msg 是即将实际发送给任务的消息。消息是一个指针长度的变量, 在不同的程序中消息的使用也可能
360 * 不同。不允许传递一个空指针, 因为这意味着消息邮箱为空。
361 *
362 * opt 该选项定义邮箱的发送条件:
363 * OS_POST_OPT_NONE 定义消息只发送给等待邮箱消息得任务中优先级最高得任务;
364 * OS_POST_OPT_BROADCAST 让所有等待邮箱消息得任务都得到消息。
365 *
366 * 返回: OS_NO_ERR 消息成功的放到消息邮箱中;
367 * OS_MBOX_FULL 消息邮箱已经包含了其他消息, 已满;
368 * OS_ERR_EVENT_TYPE 'pevent' 不是指向消息邮箱的指针;
369 * S_ERR_PEVENT_NULL 'pevent' 是空指针
370 * OS_ERR_POST_NULL_PTR 用户试图发出空指针。根据规则, 在这里不支持空指针。
371 *
372 * 警告: 1) 必须先建立消息邮箱, 然后再使用;
373 * 2) 不允许向邮箱发送空指针, 因为这意味着消息邮箱为空;
374 * 3) 若想使用本函数, 又希望压缩代码长度, 则可以将 OSMboxPost() 函数得开关量关掉;
375 * 因为 OSMboxPostOpt() 可以仿真 OSMboxPost()。
376 * 4) OSMboxPostOpt() 在广播方式下, 即已将 opt 置为 OS_POST_OPT_BROADCAST, 函数的执行时间取决与等待
377 * 邮箱消息的任务的数目。
378 ****
379 */

```

```

380
381 #if OS_MBOX_POST_OPT_EN > 0 //允许(1)生成 OSMboxPost() 代码
382 INT8U OSMboxPostOpt (OS_EVENT *pevent, void *msg, INT8U opt)
383 { //向邮箱发送一则消息(邮箱指针、消息、条件)
384 #if OS_CRITICAL_METHOD == 3 //中断函数被设定为模式3
385     OS_CPU_SR cpu_sr;
386 #endif
387
388
389 #if OS_ARG_CHK_EN > 0 //所有参数在指定的范围之内
390     if (pevent == (OS_EVENT *)0) { //当邮箱指针为NULL, 即0(空)
391         return (OS_ERR_PEVENT_NULL); //返回(pevent是空指针)
392     }
393     if (msg == (void *)0) { //检查消息是否为空指针
394         return (OS_ERR_POST_NULL_PTR); //返回(用户试图发出空指针), 不支持空指针
395     }
396     if (pevent->OSEventType != OS_EVENT_TYPE_MBOX) { //事件发送标志是否是邮箱标志
397         return (OS_ERR_EVENT_TYPE); // 'pevent' 不是指向消息邮箱的指针
398     }
399 #endif
400     OS_ENTER_CRITICAL(); //关闭中断
401     if (pevent->OSEventGrp != 0x00) { //是否有任务在等待该邮箱, 索引值≠0
402         //如果opt必须为OS_POST_OPT_BROADCAST, 所有的任务都得到该消息
403         if ((opt & OS_POST_OPT_BROADCAST) != 0x00) { //如果opt必须为OS_POST_OPT_BROADCAST
404             while (pevent->OSEventGrp != 0x00) { //如果是, 所有的任务都得到该消息
405                 OS_EventTaskRdy(pevent, msg, OS_STAT_MBOX);
406             }
407         } else { //如果没有请求广播, 那么只有最高任务进入就绪态, 准备运行
408             // OS_EventTaskRdy() 函数将最高级优先级任务从等待列表中删除
409             OS_EventTaskRdy(pevent, msg, OS_STAT_MBOX);
410         }
411         OS_EXIT_CRITICAL(); //打开中断
412         OS_Sched(); //如果该任务不是最高优先级, 进入(任务调度)
413         return (OS_NO_ERR); //消息成功的放到消息邮箱中
414     }
415     if (pevent->OSEventPtr != (void *)0) { //是否有任务在等待该邮箱, 索引值≠0
416         OS_EXIT_CRITICAL(); //打开中断
417         return (OS_MBOX_FULL); //消息邮箱已经包含了其他消息, 已满
418     }
419     pevent->OSEventPtr = msg; //将消息的指针保存到邮箱中
420     OS_EXIT_CRITICAL(); //打开中断
421     return (OS_NO_ERR); //消息成功的放到消息邮箱中
422 }
423 #endif
424
425 /*$PAGE*/
426 /*
427 ****
428 * 取得消息邮箱的信息(QUERY A MESSAGE MAILBOX)
429 *
430 * 描述: 用来取得消息邮箱的信息。用户程序必须分配一个OS_MBOX_DATA的数据结构, 该结构用来从消息邮箱的事件
431 * 控制块接受数据。通过调用OSMboxQuery()函数可以知道任务是否在等待消息以及有多少个任务在等待消息,
432 * 还可以检查消息邮箱现在的消息。
433 *
434 * 参数: pevent 是指向即将接受消息的消息邮箱的指针。该指针的值在建立该消息邮箱时可以得到。
435 *
436 * pdata 是指向OS_MBOX_DATA数据结构指针, 该数据结构包含下述成员:
437 * Void *OSMsg; /* 消息邮箱中消息的复制 */
438 * INT8U OSEventTbl[OS_EVENT_TBL_SIZE]; /* 消息邮箱等待队列的复制 */
439 * INT8U OSEventGrp;
440 *
441 * 返回: OS_NO_ERR 调用成功
442 * OS_ERR_EVENT_TYPE 'pevent' 不是指向消息邮箱的指针。
443 * OS_ERR_PEVENT_NULL 'pevent' 是空指针。
444 * 注意: 必须先建立消息邮箱, 然后使用
445 ****
446 */
447
448 #if OS_MBOX_QUERY_EN > 0 //允许(1)生成 OSMboxPost() 代码
449 INT8U OSMboxQuery (OS_EVENT *pevent, OS_MBOX_DATA *pdata)
450 { //查询一个邮箱的当前状态(信号量指针、状态数据结构指针)
451 #if OS_CRITICAL_METHOD == 3 //中断函数被设定为模式3
452     OS_CPU_SR cpu_sr;
453 #endif
454     INT8U *psrc; //定义8位pevent->OSEventTbl[0]的地址指针

```

```

455     INT8U      *pdest;                                //定义8位pdata->OSEventTbl[0]的地址指针
456
457
458 #if OS_ARG_CHK_EN > 0                                //所有参数在指定的范围之内
459     if (pevent == (OS_EVENT *)0) {                    //当邮箱指针为NULL, 即0(空)
460         return (OS_ERR_PEVENT_NULL);                  //返回(pevent是空指针)
461     }
462     if (pevent->OSEventType != OS_EVENT_TYPE_MBOX) {   //当事件类型不是邮箱类型
463         return (OS_ERR_EVENT_TYPE);                   //'pevent'不是指向消息邮箱的指针
464     }
465 #endif
466     OS_ENTER_CRITICAL();                               //关闭中断
467     //将事件(邮箱)结构中的等待任务列表复制到pdata数据结构中
468     pdata->OSEventGrp = pevent->OSEventGrp;           //等待事件的任务组中的内容传送到状态数据结构中
469     psrc              = &pevent->OSEventTbl[0];       //保存pevent->OSEventTbl[0]对应的地址
470     pdest             = &pdata->OSEventTbl[0];       //保存pdata->OSEventTbl[0]对应的地址
471
472 #if OS_EVENT_TBL_SIZE > 0                             //当事件就绪对应表中的对应值>0时
473     *pdest++          = *psrc++;                      //地址指针下移一个类型地址, 获取消息邮箱的值
474 #endif
475
476 #if OS_EVENT_TBL_SIZE > 1                             //事件就绪对应表中的对应值>1时
477     *pdest++          = *psrc++;                      //地址指针继续下移一个类型地址, 获取消息邮箱的值
478 #endif
479
480 #if OS_EVENT_TBL_SIZE > 2                             //事件就绪对应表中的对应值>1时
481     *pdest++          = *psrc++;                      //地址指针继续下移一个类型地址, 获取消息邮箱的值
482 #endif
483
484 #if OS_EVENT_TBL_SIZE > 3                             //事件就绪对应表中的对应值>1时
485     *pdest++          = *psrc++;                      //地址指针继续下移一个类型地址, 获取消息邮箱的值
486 #endif
487
488 #if OS_EVENT_TBL_SIZE > 4                             //事件就绪对应表中的对应值>1时
489     *pdest++          = *psrc++;                      //地址指针继续下移一个类型地址, 获取消息邮箱的值
490 #endif
491
492 #if OS_EVENT_TBL_SIZE > 5                             //事件就绪对应表中的对应值>1时
493     *pdest++          = *psrc++;                      //地址指针继续下移一个类型地址, 获取消息邮箱的值
494 #endif
495
496 #if OS_EVENT_TBL_SIZE > 6                             //事件就绪对应表中的对应值>1时
497     *pdest++          = *psrc++;                      //地址指针继续下移一个类型地址, 获取消息邮箱的值
498 #endif
499
500 #if OS_EVENT_TBL_SIZE > 7                             //事件就绪对应表中的对应值>7时
501     *pdest            = *psrc;                        //获取最后地址的信号量的值
502 #endif
503     pdata->OSMsg = pevent->OSEventPtr;                 //将邮箱中的当前消息从事件数据结构复制到OS_MBOX_DATA数据
504     OS_EXIT_CRITICAL();                               //打开中断
505     return (OS_NO_ERR);                               //返回成功运行
506 }
507 #endif
508 #endif
509 // OS_MBOX_QUERY_EN 函数结束
510 // OS_MBOX_EN文件结束

```



```

1  /*
2  ****
3  *                                     uC/OS-II 实时控制内核
4  *                                     主要的包含文件
5  *                                     --消息队列管理项--
6  *
7  * 文    件: OS_q.C  消息队列管理代码
8  * 作    者: Jean J. Labrosse
9  * 中文注解: 钟常慰  zhongcw @ 126.com  译注版本: 1.0  请尊重原版内容
10 ****
11 */
12
13 #ifndef OS_MASTER_FILE  //是否已定义OS_MASTER_FILE主文件
14 #include "includes.h"  //包含"includes.h"文件, 部分C语言头文件的汇总打包文件
15 #endif  //定义结束
16
17 #if (OS_Q_EN > 0) && (OS_MAX_QS > 0)  //条件编译: OS_Q_EN 允许 (1)产生消息队列相关代码
18                                     //条件编译: 应用中最多对列控制块的数目 > 0
19 /*
20 ****
21 *                                     检查消息队列中是否已经有需要的消息(ACCEPT MESSAGE FROM QUEUE)
22 *
23 * 描述: 检查消息队列中是否已经有需要的消息. 不同于OSQPend()函数, 如果没有需要的消息, OSQAccept()
24 *       函数并不挂起任务. 如果消息已经到达, 该消息被传递到用户任务. 通常中断调用该函数, 因为中
25 *       断不允许挂起等待消息.
26 *
27 * 参数: pevent 是指向需要查看的消息队列的指针. 当建立消息队列时, 该指针返回到用户程序.
28 *       (参考OSMboxCreate()函数).
29 *
30 * 返回: 如果消息已经到达, 返回指向该消息的指针; 如果消息队列没有消息, 返回空指针.
31 *
32 * 注意: 必须先建立消息队列, 然后使用.
33 ****
34 */
35
36 #if OS_Q_ACCEPT_EN > 0  //条件编译: 允许生成 OSQAccept() 代码
37 void *OSQAccept (OS_EVENT *pevent)  //检查消息队列中是否已经有需要的消息(消息队列的指针)
38 {
39     #if OS_CRITICAL_METHOD == 3  //中断函数被设定为模式3
40         OS_CPU_SR  cpu_sr;
41     #endif
42     void *msg;  //定义消息队列指针(输出缓冲区)
43     OS_Q *pq;  //定义消息队列事件
44
45
46     #if OS_ARG_CHK_EN > 0  //所有参数必须在指定的参数内
47         if (pevent == (OS_EVENT *)0) {  //当消息队列指针为NULL时, 返回0, 空指针
48             return ((void *)0);
49         }
50         if (pevent->OSEventType != OS_EVENT_TYPE_Q)  //当事件类型≠消息队列类型
51             return ((void *)0);  //返回0
52     }
53 #endif
54     OS_ENTER_CRITICAL();  //关闭中断
55     pq = (OS_Q *)pevent->OSEventPtr;  //队列指针=当前事件指针
56     if (pq->OSQEntries != 0) {  //当消息队列消息数
57         msg = *pq->OSQOut++;  //输出消息内容到缓冲区
58         pq->OSQEntries--;  //消息数减1
59         if (pq->OSQOut == pq->OSQEnd) {  //当输出指针=结束指针
60             pq->OSQOut = pq->OSQStart;  //输出指针跳转到起始指针
61         }
62     } else {  //否则
63         msg = (void *)0;  //将定义消息队列指针(输出缓冲区)清空
64     }
65     OS_EXIT_CRITICAL();  //打开中断
66     return (msg);  //返回(消息=为空没有消息; 消息=不为空, 有消息)
67 }
68 #endif
69 /*$PAGE*/
70 /*
71 ****
72 *                                     建立一个消息队列(CREATE A MESSAGE QUEUE)
73 *
74 * 描述: 建立一个消息队列. 任务或中断可以通过消息队列向其他一个或多个任务发送消息. 消息的含义是
75 *       和具体的应用密切相关的.
76 *

```



```

77 * 参数: start 是消息内存区的基地址, 消息内存区是一个指针数组。
78 *      size 是消息内存区的大小。
79 *
80 * 返回: OSQCreate() 函数返回一个指向消息队列事件控制块的指针;
81 *      如果没有空余的事件空闲块, OSQCreate() 函数返回空指针。
82 *
83 * 注意 必须先建立消息队列, 然后使用
84 ****
85 */
86 //建立一个消息队列(消息内存区的基地址(指针数组)、消息内存区的大小)
87 OS_EVENT *OSQCreate (void **start, INT16U size)
88 {
89     #if OS_CRITICAL_METHOD == 3                //中断函数被设定为模式3
90         OS_CPU_SR cpu_sr;
91     #endif
92     OS_EVENT *pevent;                          //定义一个指向事件控制快的指针
93     OS_Q *pq;                                  //定义一个队列控制模块指针(事件)
94
95     if (OSIntNesting > 0) {                    //中断嵌套数>0时, 表示还有中断任务在运行
96         return ((OS_EVENT *)0);                //返回0;
97     }
98     OS_ENTER_CRITICAL();                       //关闭中断
99     pevent = OSEventFreeList;                  //pevent=空余事件管理列表
100     if (OSEventFreeList != (OS_EVENT *)0) {    //如果有空余事件管理块
101         OSEventFreeList = (OS_EVENT *)OSEventFreeList->OSEventPtr;
102     }                                           //空余事件控制链表指向下一个空余事件控制块
103     OS_EXIT_CRITICAL();                         //打开中断
104     if (pevent != (OS_EVENT *)0) {             //如果有事件控制块ECB可用
105         OS_ENTER_CRITICAL();                   //关闭中断
106         pq = OSQFreeList;                      //队列指针指向空余队列控制链表的队列控制块
107         if (OSQFreeList != (OS_Q *)0) {        //当是空余列表时(即链接表中还有空余)
108             OSQFreeList = OSQFreeList->OSQPtr; //控制链接表指向下一个空余控制块
109         }
110         OS_EXIT_CRITICAL();                     //打开中断
111         if (pq != (OS_Q *)0) {                 //消息队列初始化
112             pq->OSQStart = start;               //指向消息队列的起始指针
113             pq->OSQEnd = &start[size];         //指向消息队列结束下一个单元地址
114             pq->OSQIn = start;                  //指向消息队列下一个要插入的消息指针
115             pq->OSQOut = start;                 //指向消息队列下一个取出消息指针
116             pq->OSQSize = size;                 //消息队列可容纳的最多消息数
117             pq->OSQEntries = 0;                 //消息队列中的消息数
118             pevent->OSEventType = OS_EVENT_TYPE_Q; //事件类型为消息队列
119             pevent->OSEventPtr = pq;            //OSEventcnt只用于信号量, 不用置0
120             OS_EventWaitListInit(pevent);       //初始化一个事件控制块
121         } else {
122             OS_ENTER_CRITICAL();               //关闭中断
123             pevent->OSEventPtr = (void *)OSEventFreeList; //事件控制块ECB返回
124             OSEventFreeList = pevent;          //空余块链接表=当前事件指针
125             OS_EXIT_CRITICAL();                 //打开中断
126             pevent = (OS_EVENT *)0;           //事件指针=0
127         }
128     }
129     return (pevent);                          //消息队列建立成功, 返回一个消息队列得指针, 并成为该消息句柄
130 }
131 */
132 /*$PAGE*/
133 /*
134 ****
135 *      删除消息队列(DELETE A MESSAGE QUEUE)
136 *
137 * 描述: 删除消息队列。使用这个函数有风险, 因为多任务中的其他任务可能还想用这个消息队列。使用这个函数要特别小心。一般的说, 应先删除可能会用到这个消息队列的所以任务, 再调用本函数。
138 *
139 *
140 *
141 * 参数: pevent 是指向消息队列的指针。该指针的值在建立该队列时可以得到。(参考OSQCreate() 函数)
142 *
143 *      opt 该选项定义消息队列删除条件:
144 *          opt==OS_DEL_NO_PEND 可以选择在没有任何任务在等待该消息队列时, 才删除该消息队列;
145 *          opt==OS_DEL_ALWAYS 不管有没有任务在等待该消息队列的消息, 立刻删除该消息队列。
146 *                              后一种情况下, 所有等待该消息的任务都立刻进入就绪态
147 *
148 *      err 指向错误代码的指针, 出错代码为以下之一:
149 *          OS_NO_ERR 调用成功, 消息队列已被删除;
150 *          OS_ERR_DEL_ISR 试图在中断服务子程序中删除消息队列;
151 *          OS_ERR_INVALID_OPT 'opt' 参数没有在以上2个参数值之一;
152 *          OS_ERR_TASK_WAITING 有一个或一个以上的任务在等待消息队列中的消息;

```

```

153 *          OS_ERR_EVENT_TYPE      'pevent' 不是指向消息队列的指针;
154 *          OS_ERR_PEVENT_NULL     已经没有OS_EVENT(事件)数据结构可以使用了.
155 *
156 * 返回: pevent 如果消息队列删除成功, 则返回空指针;
157 *          如果消息队列没有被删除, 则返回pevent. 在后一种情况查看出错代码, 找出原因.
158 *
159 * 注意: 1) 调用本函数需十分小心, 因为多任务中的其他任务可能还想用这个消息队列;
160 *        2) 当挂起任务进入就绪态时, 中断是关闭的, 这就意味着中断延迟时间取决于等待消息队列的任务数目
161 *        ****
162 */
163
164 #if OS_Q_DEL_EN > 0                                //允许生成 OSSemDel() 代码
165 OS_EVENT *OSQDel (OS_EVENT *pevent, INT8U opt, INT8U *err)
166 {                                                    //删除一个消息队列(消息队列指针、删除条件、错误指针)
167     #if OS_CRITICAL_METHOD == 3                    //中断函数被设定为模式3
168         OS_CPU_SR cpu_sr;
169     #endif
170     BOOLEAN      tasks_waiting;                    //定义布尔量, 任务等待条件
171     OS_Q          *pq;                             //定义一个队列控制模块指针(事件)
172
173     if (OSIntNesting > 0) {                        //中断嵌套数>0时, 表示还有中断任务在运行
174         *err = OS_ERR_DEL_ISR;                    //错误等于(试图在中断程序中删除一个消息队列)
175         return ((OS_EVENT *)0);                  //返回0
176     }
177     #if OS_ARG_CHK_EN > 0                          //所有参数在指定的范围之内
178     if (pevent == (OS_EVENT *)0) {                //当信号量指针为NULL, 即0(空)
179         *err = OS_ERR_PEVENT_NULL;                //已经没有可用的OS_EVENT数据结构可以使用了
180         return (pevent);                          //返回指针
181     }
182     if (pevent->OSEventType != OS_EVENT_TYPE_Q) { //事件类型是否时消息队列
183         *err = OS_ERR_EVENT_TYPE;                 //'pevent' 不是指向消息队列的指针
184         return (pevent);                          //返回指针
185     }
186     #endif
187     OS_ENTER_CRITICAL();                          //关闭中断
188     if (pevent->OSEventGrp != 0x00) {               //事件等待标志(索引值是否有任务在等待)
189         tasks_waiting = TRUE;                      //有则任务等待标志=1
190     } else {                                       //否则
191         tasks_waiting = FALSE;                    //没有则任务等待标志=0
192     }
193     switch (opt) {                                //条件
194     case OS_DEL_NO_PEND:                          // 1) 选择没有任务在等待该消息队列
195         if (tasks_waiting == FALSE) {              //没有任务在等待
196             pq = pevent->OSEventPtr;               //队列指针=当前事件指针
197             pq->OSQPtr = OSQFreeList;              //队列空余指针=当前空闲队列链接表
198             OSQFreeList = pq;                     //空闲队列链接表=当前队列指针
199             pevent->OSEventType = OS_EVENT_TYPE_UNUSED; //事件类型=空闲
200             pevent->OSEventPtr = OSEventFreeList;  //队列对应得事件指针=空余块链接表
201             OSEventFreeList = pevent;              //空余块链接表=当前事件指针
202             OS_EXIT_CRITICAL();                    //打开中断
203             *err = OS_NO_ERR;                      //调用成功, 消息队列已被删除
204             return ((OS_EVENT *)0);                //返回0
205         } else {                                  //否则
206             OS_EXIT_CRITICAL();                    //打开中断
207             *err = OS_ERR_TASK_WAITING;            //有一个或一个以上的任务在等待消息队列中的消息
208             return (pevent);                      //返回消息队列指针
209         }
210     }
211     case OS_DEL_ALWAYS:                          // 2) 尽管有(多)任务在等待, 还是要删除
212         while (pevent->OSEventGrp != 0x00) {       //事件等待标志≠0, 还是要删除
213             OS_EventTaskRdy (pevent, (void *)0, OS_STAT_Q);
214             //使一个任务进入到就绪状态
215             pq = pevent->OSEventPtr;               //队列指针=当前事件指针
216             pq->OSQPtr = OSQFreeList;              //队列空余指针=当前空闲队列链接表
217             OSQFreeList = pq;                     //空闲队列链接表=当前队列指针
218             pevent->OSEventType = OS_EVENT_TYPE_UNUSED; //事件类型=空闲
219             pevent->OSEventPtr = OSEventFreeList;  //队列对应得事件指针=空余块链接表
220             OSEventFreeList = pevent;              //空余块链接表=当前事件指针
221             OS_EXIT_CRITICAL();                    //打开中断
222             if (tasks_waiting == TRUE) {            //当有任务在等待(1)
223                 OS_Sched();                        //任务调度函数, 最高任务优先级进入就绪态
224             }
225             *err = OS_NO_ERR;                      //调用成功, 消息队列已被删除
226             return ((OS_EVENT *)0);                //返回0
227         }
228

```

```

229     default:                                     // 3) 以上两者都不是
230         OS_EXIT_CRITICAL();                       // 打开中断
231         *err = OS_ERR_INVALID_OPT;                 // 'opt' 参数没有在上面2个参数值之一
232         return (pevent);                           // 返回指针
233     }
234 }
235 #endif
236
237 /*$PAGE*/
238 /*
239 ****
240 *      清空消息队列并且忽略发往队列的所有消息 (FLUSH QUEUE)
241 *
242 * 描述: 清空消息队列并且忽略发往队列的所有消息。
243 *      不管队列中是否有消息, 这个函数的执行时间都是相同的。
244 *
245 * 参数: 无
246 *
247 * 返回: OS_NO_ERR      消息队列被成功清空
248 *      OS_ERR_EVENT_TYPE 试图清除不是消息队列的对象
249 *      OS_ERR_PEVENT_NULL 'pevent' 是空指针
250 *
251 * 注意: 必须先建立消息队列, 然后使用
252 ****
253 */
254
255 #if OS_Q_FLUSH_EN > 0                             // 允许生成 OSQFlush() 代码
256 INT8U OSQFlush (OS_EVENT *pevent)                 // 清空消息队列 (指向得到消息队列的指针)
257 {
258     #if OS_CRITICAL_METHOD == 3                     // 中断函数被设定为模式3
259         OS_CPU_SR cpu_sr;
260     #endif
261     OS_Q *pq;                                       // 定义一个队列事件
262
263     #if OS_ARG_CHK_EN > 0                           // 所有参数在指定的范围之内
264         if (pevent == (OS_EVENT *)0) {             // 当信号量指针为NULL, 即0 (空)
265             return (OS_ERR_PEVENT_NULL);           // pevent 是空指针
266         }
267         if (pevent->OSEventType != OS_EVENT_TYPE_Q) { // 当事件类型不是消息队列
268             return (OS_ERR_EVENT_TYPE);           // 试图清除不是消息队列的对象
269         }
270     }
271     #endif
272     OS_ENTER_CRITICAL();                             // 关闭中断
273     pq = (OS_Q *)pevent->OSEventPtr;                // 队列指针 = 当前事件指针
274     pq->OSQIn = pq->OSQStart;                         // 插入指针 = 起始指针
275     pq->OSQOut = pq->OSQStart;                        // 输出指针 = 起始指针
276     pq->OSQEntries = 0;                               // 消息队列数目 = 0
277     OS_EXIT_CRITICAL();                               // 打开中断
278     return (OS_NO_ERR);                               // 返回 (消息队列被成功清空)
279 }
280 #endif
281
282 /*$PAGE*/
283 /*
284 ****
285 *      任务等待消息队列中的消息 (PEND ON A QUEUE FOR A MESSAGE)
286 *
287 * 描述: 用于任务等待消息。消息通过中断或另外的任务发送给需要的任务。
288 *      消息是一个以指针定义的变量, 在不同的程序中消息的使用也可能不同。如果调用OSQPend() 函数时
289 *      队列中已经存在需要的消息, 那么该消息被返回给OSQPend() 函数的调用者, 队列中清除该消息。如果
290 *      调用OSQPend() 函数时队列中没有需要的消息, OSQPend() 函数挂起当前任务直到得到需要的消息或
291 *      超出定义的超时时间。如果同时有多个任务等待同一个消息, uC/OS-ii 默认最高优先级的任务取得消
292 *      息并且任务恢复执行。一个由OSTaskSuspend() 函数挂起的任务也可以接受消息, 但这个任务将一直
293 *      保持挂起状态直到通过调用OSTaskResume() 函数恢复任务的运行。
294 *
295 * 参数: pevent 是指向即将接受消息的队列的指针。
296 *      该指针的值在建立该队列时可以得到。(参考OSMboxCreate() 函数)
297 *
298 *      timeout 允许一个任务在经过了指定数目的时钟节拍后还没有得到需要的消息时恢复运行状态。
299 *      如果该值为零表示任务将持续的等待消息。最大的等待时间为65535个时钟节拍。这个时
300 *      间长度并不是非常严格的, 可能存在一个时钟节拍的误差, 因为只有在一个时钟节拍结
301 *      束后才会减少定义的等待超时时钟节拍。
302 *
303 *      err 是指向包含错误码的变量的指针。OSQPend() 函数返回的错误码可能为下述几种:
304 *      OS_NO_ERR 消息被正确的接受;

```

```

305 *          OS_TIMEOUT          消息没有在指定的周期数内送到;
306 *          OS_ERR_EVENT_TYPE    'pevent' 不是指向消息队列的指针;
307 *          OS_ERR_PEVENT_NULL   'pevent' 是空指针;
308 *          OS_ERR_PEND_ISR      从中断调用该函数。虽然规定了不允许从中断调用该函数, 但
309 *                               uc/OS-ii 仍然包含了检测这种情况的功能
310 *
311 * 返回: OSQPend() 函数返回接受的消息并将 *err置为OS_NO_ERR。
312 *       如果没有在指定数目的时钟节拍内接收到需要的消息, OSQPend() 函数返回空指针并且将 *err
313 *       设置为OS_TIMEOUT。
314 *
315 * 注意: 1、必须先建立消息邮箱, 然后使用;
316 *       2、不允许从中断调用该函数。
317 *****/
318 */
319 //任务等待消息队列中的消息(消息队列指针、允许等待的时钟节拍、代码错误指针)
320 void *OSQPend (OS_EVENT *pevent, INT16U timeout, INT8U *err)
321 {
322     #if OS_CRITICAL_METHOD == 3                //中断函数被设定为模式3
323         OS_CPU_SR  cpu_sr;
324     #endif
325     void          *msg;                        //定义消息队列的指针、取出的暂存指针
326     OS_Q          *pq;                        //定义一个队列事件
327
328
329     if (OSIntNesting > 0) {                    //中断嵌套数>0时, 表示还有中断任务在运行
330         *err = OS_ERR_PEND_ISR;                //试图从中断调用该函数
331         return ((void *)0);                    //返回空(0)
332     }
333     #if OS_ARG_CHK_EN > 0                      //所有参数在指定的范围之内
334     if (pevent == (OS_EVENT *)0) {            //当信号量指针为NULL, 即0(空)
335         *err = OS_ERR_PEVENT_NULL;            //pevent是空指针
336         return ((void *)0);                    //返回
337     }
338     if (pevent->OSEventType != OS_EVENT_TYPE_Q) { //当事件类型不否是消息队列类型
339         *err = OS_ERR_EVENT_TYPE;              //'pevent' 不是指向消息队列的指针
340         return ((void *)0);                    //返回空(0)
341     }
342     #endif
343     OS_ENTER_CRITICAL();                      //关闭中断
344     pq = (OS_Q *)pevent->OSEventPtr;          //队列指针=当前事件指针
345     if (pq->OSQEntries != 0) {                 //当前消息队列中消息数 > 0, 有消息
346         msg = *pq->OSQOut++;                   //OSQOut将对应的地址的消息复制到msg
347         pq->OSQEntries--;                       //当前队列消息数减1
348         if (pq->OSQOut == pq->OSQEnd) {         //当取出指针=最高消息队列单元时
349             pq->OSQOut = pq->OSQStart;          //取出指针跳转到起始单元
350         }
351         OS_EXIT_CRITICAL();                    //打开中断
352         *err = OS_NO_ERR;                      //消息被正确的接受
353         return (msg);                          //返回消息暂存(数据)指针
354     } // 无消息
355     OSTCBCur->OSTCBStat |= OS_STAT_Q;          //将事件进入睡眠状态, 由消息队列唤醒
356     OSTCBCur->OSTCBDly = timeout;              //等待时间置入任务控制中
357     OS_EventTaskWait(pevent);                 //使任务进入等待消息队列状态
358     OS_EXIT_CRITICAL();                       //打开中断
359     OS_Sched();                               //任务调度函数, 调用一个就绪的高优先级任务运行
360     OS_ENTER_CRITICAL();                      //关闭中断
361     msg = OSTCBCur->OSTCBMsg;                  //接收消息=指向当前任务的消息指针
362     if (msg != (void *)0) {                   //检查消息是否为空
363         OSTCBCur->OSTCBMsg = (void *)0;        //传递给消息的指针为空
364         OSTCBCur->OSTCBStat = OS_STAT_RDY;     //表示任务处于就绪状态
365         OSTCBCur->OSTCBEvtPtr = (OS_EVENT *)0; //指向事件控制块的指针=0
366         OS_EXIT_CRITICAL();                    //打开中断
367         *err = OS_NO_ERR;                      //成功等待消息队列
368         return (msg);                          //返回消息暂存(数据)指针
369     }
370     OS_EventTO(pevent);                       //如果没有获得消息, 由于等待起始时间
371     OS_EXIT_CRITICAL();                       //打开中断
372     *err = OS_TIMEOUT;                        //消息没有在指定的时间送到
373     return ((void *)0);                       //返回0
374 }
375 **/$PAGE*/
376 */
377 *****/
378 *          向消息队列发送一则消息(POST MESSAGE TO A QUEUE)
379 *
380 * 描述: 通过消息队列向任务发送消息. 消息是一个指针长度的变量, 在不同的程序中消息的使用也可能不同.

```



```

381 *      如果队列中已经存满消息, 返回错误码. OSQPost() 函数立即返回调用者, 消息也没有能够发到队列.
382 *      如果有任何任务在等待队列中的消息, 最高优先级的任务将得到这个消息. 如果等待消息的任务优先
383 *      级比发送消息的任务优先级高, 那么高优先级的任务将得到消息而恢复执行, 也就是说, 发生了一次
384 *      任务切换. 消息队列是先入先出(FIFO)机制的, 先进入队列的消息先被传递给任务.
385 *
386 * 参数: pevent 是指向即将接受消息的消息队列的指针. 该指针的值在建立该队列时可以得到.
387 *      (参考OSQCreate() 函数)
388 *
389 *      msg      是即将实际发送给任务的消息. 消息是一个指针长度的变量, 在不同的程序中消息的使用也
390 *      可能不同. 不允许传递一个空指针.
391 *
392 * 返回:
393 *      OS_NO_ERR      消息成功的放到消息队列中;
394 *      OS_Q_FULL      消息队列中已经存满;
395 *      OS_ERR_EVENT_TYPE  'pevent' 不是指向消息队列的指针;
396 *      OS_ERR_PEVENT_NULL 'pevent' 是空指针;
397 *      OS_ERR_POST_NULL_PTR 用户发出空指针. 根据规则, 这里不支持空指针.
398 *
399 * 注意: 1、必须先建立消息队列, 然后使用;
400 *      2、不允许从中断调用该函数.
401 *****
402 */
403
404 #if OS_Q_POST_EN > 0                                //允许生成 OSQPost() 代码
405 INT8U OSQPost (OS_EVENT *pevent, void *msg)
406 {
407     //向消息队列发送一则消息FIFO(消息队列指针、发送的消息)
408     #if OS_CRITICAL_METHOD == 3                    //中断函数被设定为模式3
409         OS_CPU_SR cpu_sr;
410     #endif
411     OS_Q *pq;                                     //定义一个队列事件
412
413     #if OS_ARG_CHK_EN > 0                            //所有参数在指定的范围之内
414         if (pevent == (OS_EVENT *)0) {              //当消息队列指针为NULL, 即0(空)
415             return (OS_ERR_PEVENT_NULL);            //pevent是空指针
416         }
417         if (msg == (void *)0) {                      //检查消息队列是否为空, 用户试发出空消息
418             return (OS_ERR_POST_NULL_PTR);          //用户发出空指针. 根据规则, 这里不支持空指针.
419         }
420         if (pevent->OSEventType != OS_EVENT_TYPE_Q) { //事件类型是否为消息队列
421             return (OS_ERR_EVENT_TYPE);            //'pevent' 不是指向消息队列的指针
422         }
423     #endif
424     OS_ENTER_CRITICAL();                            //关闭中断
425     if (pevent->OSEventGrp != 0x00) {                //是否有任务在等待该消息队列, 索引值≠0
426         OS_EventTaskRdy(pevent, msg, OS_STAT_Q);    //使最高优先级任务进入就绪态
427         OS_EXIT_CRITICAL();                          //打开中断
428         OS_Sched();                                  //任务调度函数, 调用一个就绪的高优先级任务运行
429         return (OS_NO_ERR);                          //消息成功的放到消息队列中
430     }
431     pq = (OS_Q *)pevent->OSEventPtr;                //消息队列指针=当前事件指针
432     if (pq->OSQEntries >= pq->OSQSize) {              //消息队列当前消息数>=消息中可容纳的消息数
433         OS_EXIT_CRITICAL();                          //打开中断
434         return (OS_Q_FULL);                          //返回消息队列已满
435     }
436     *pq->OSQIn++ = msg;                              //插入当前的消息(内容), 地址为指针加1
437     pq->OSQEntries++;                                //消息队列数加1
438     if (pq->OSQIn == pq->OSQEnd) {                   //当插入的消息指针=最后(结束)的指针
439         pq->OSQIn = pq->OSQStart;                    //插入指针跳到起始处指针
440     }
441     OS_EXIT_CRITICAL();                              //打开中断
442     return (OS_NO_ERR);                              //消息成功的放到消息队列中
443 }
444 #endif
445 /*$PAGE*/
446 /*
447 *****
448 *      通过消息队列向任务发送消息(POST MESSAGE TO THE FRONT OF A QUEUE)
449 *
450 * 描述: 通过消息队列向任务发送消息. OSQPostFront() 函数和OSQPost() 函数非常相似, 不同之处在于
451 *      OSQPostFront() 函数将发送的消息插到消息队列的最前端. 也就是说, OSQPostFront() 函数使得
452 *      消息队列按照后入先出(LIFO)的方式工作, 而不是先入先出(FIFO). 消息是一个指针长度的变
453 *      量, 在不同的程序中消息的使用也可能不同. 如果队列中已经存满消息, 返回错误码. OSQPost()
454 *      函数立即返回调用者, 消息也没能发到队列. 如果有任何任务在等待队列中的消息, 最高优先级的
455 *      任务将得到这个消息. 如果等待消息的任务优先级比发送消息的任务优先级高, 那么高优先级
456 *      的任务将得到消息而恢复执行, 也就是说, 发生了一次任务切换.

```

```

457 *
458 * 参数: pevent 是指向即将接受消息的消息队列的指针。
459 *           该指针的值在建立该队列时可以得到。(参考OSQCreate()函数)。
460 *           msg 是即将实际发送给任务的消息。消息是一个指针长度的变量,在不同的程序中消息的使
461 *           用也可能不同。不允许传递一个空指针。
462 *
463 * 返回: OS_NO_ERR 消息成功的放到消息队列中;
464 *        OS_Q_FULL 消息队列已满;
465 *        OS_ERR_EVENT_TYPE 'pevent'不是指向消息队列的指针;
466 *        OS_ERR_PEVENT_NULL 'pevent'是指空指针;
467 *        OS_ERR_POST_NULL_PTR 用户发出空指针。根据规则,这里不支持空指针。
468 *
469 * 注意: 1、必须先建立消息队列,然后使用。
470 *        2、不允许传递一个空指针
471 *****/
472 */
473
474 #if OS_Q_POST_FRONT_EN > 0 //允许生成 OSQPost() 代码
475 INT8U OSQPostFront (OS_EVENT *pevent, void *msg)
476 { //向消息队列发送一则消息LIFO(消息队列指针、发送的消息)
477     #if OS_CRITICAL_METHOD == 3 //中断函数被设定为模式3
478         OS_CPU_SR cpu_sr;
479     #endif
480     OS_Q *pq; //定义一个队列事件
481
482     #if OS_ARG_CHK_EN > 0 //所有参数在指定的范围之内
483         if (pevent == (OS_EVENT *)0) { //当消息队列指针为NULL,即0(空)
484             return (OS_ERR_PEVENT_NULL); //pevent是空指针
485         }
486         if (msg == (void *)0) { //检查消息队列是否为空,用户试发出空消息
487             return (OS_ERR_POST_NULL_PTR); //用户发出空指针。根据规则,这里不支持空指针。
488         }
489         if (pevent->OSEventType != OS_EVENT_TYPE_Q) { //事件类型是否为消息队列
490             return (OS_ERR_EVENT_TYPE); // 'pevent'不是指向消息队列的指针
491         }
492     #endif
493     OS_ENTER_CRITICAL(); //关闭中断
494     if (pevent->OSEventGrp != 0x00) { //是否有任务在等待该消息队列,索引值≠0
495         OS_EventTaskRdy(pevent, msg, OS_STAT_Q); //使最高优先级任务进入就绪态
496         OS_EXIT_CRITICAL(); //打开中断
497         OS_Sched(); //任务调度函数,调用一个就绪的高优先级任务运行
498         return (OS_NO_ERR); //消息成功的放到消息队列中
499     }
500     pq = (OS_Q *)pevent->OSEventPtr; //消息队列指针=当前事件指针
501     if (pq->OSQEntries >= pq->OSQSize) { //消息队列当前消息数>=消息中可容纳的消息数
502         OS_EXIT_CRITICAL(); //打开中断
503         return (OS_Q_FULL); //返回消息队列已满
504     }
505     if (pq->OSQOut == pq->OSQStart) { //当插入指针=指针的起始地址(指针)
506         pq->OSQOut = pq->OSQEnd; //插入指针跳转到最后地址(指针)
507     }
508     pq->OSQOut--; //插入指针减1
509     *pq->OSQOut = msg; //插入当前消息(内容)
510     pq->OSQEntries++; //消息数加1
511     OS_EXIT_CRITICAL(); //打开中断
512     return (OS_NO_ERR); //消息成功的放到消息队列中
513 }
514 #endif
515 /**$PAGE*/
516 /**
517 *****/
518 * 消息队列向任务发消息(POST MESSAGE TO A QUEUE)
519 *
520 * 描述: 通过消息队列向任务发消息。消息是一个以指针表示的某种数据类型的变量,在不同的程序中消
521 * 息的使用也可能不同。如果消息队列中的消息已满,则返回出错代码,说明消息队列已满。
522 * OSQPostOpt() 函数立即返回调用者,消息也没有能够发到消息队列,如果有任何任务在等待消息
523 * 队列中的消息,那么 OSQPostOpt() 允许选择以下2种情况之一:
524 * 1、让最高优先级的任务得到这则消息(opt置为OS_POST_OPT_NONE);
525 * 2、或者让所有等待队列消息的任务都得到消息(opt置为OS_POST_OPT_BROADCAST)
526 * ->无论在哪种情况下,如果得到消息的任务优先级比发送消息的任务优先级高,那么得到消息
527 * 的最高优先级的任务恢复执行,发消息的任务将被挂起。也就是执行一次任务切换。
528 *
529 * OSQPostOpt() 仿真OSQPost()和OSQPostFront()这2个函数,并允许程序发消息给多个任务。换句
530 * 话说。OSQPostOpt() 允许将消息广播给所有的等待队列消息的任务。OSQPostOpt()实际上可以取
531 * 代OSQPost(),因为可以通过设定opt参数定义队列消息的接收方式,这样做可以减少ucos_ii占用的
532 * 的代码空间。

```



```

533 *
534 * 参数: pevent 是指向即将接收消息的消息队列的指针。该指针的值在建立该消息邮箱时可以得到。
535 *          (参考OSQCreate()函数)。
536 *
537 *      msg      即将发送给任务的消息。消息是一个指向某种变量类型的指针,在不同的应用程序中,
538 *              消息的类型是用户定义的。不允许传递一个空指针。
539 *
540 *      opt      决定消息发送方式的选项:
541 *              OS_POST_OPT_NONE      发送一个消息给一个等待消息的任务(等同于OSQPost())
542 *              OS_POST_OPT_BROADCAST  发送消息给所有等待队列消息的任务
543 *              OS_POST_OPT_FRONT      以后进先出方式发消息(仿真OSQPostFront())
544 *
545 *          以下是所有标志可能的组合:
546 *
547 *              1) OS_POST_OPT_NONE      等同于OSQPost()
548 *              2) OS_POST_OPT_FRONT      等同于OSQPostFront()
549 *              3) OS_POST_OPT_BROADCAST  等同于OSQPost(),但广播给所有等待队列消息的任务
550 *              4) OS_POST_OPT_FRONT + OS_POST_OPT_BROADCAST is identical to
551 *                  ->等同于OSQPostFront()不同的是,它将消息广播给所有等待队列消息的任务
552 *
553 * 返回: OS_NO_ERR      调用成功,消息已经发出;
554 *       OS_Q_FULL      消息队列已满,不能再接收新消息;
555 *       OS_ERR_EVENT_TYPE  'pevent'指向的数据类型错;
556 *       OS_ERR_PEVENT_NULL 'pevent'是空指针;
557 *       OS_ERR_POST_NULL_PTR 用户程序试图发出空指针。
558 *
559 * 警告: 1、必须先建立消息队列,然后使用;
560 *       2、不允许传递一个空指针;
561 *       3、如故想使用本函数,又希望压缩代码长度,则可以将OSQPost()函数的开关量关闭(置
562 *          OS_CFG.H文件中的OS_Q_POST_EN为0),并将OSQPostFront()的开关量关闭(置OS_CFG.H文件
563 *          中的OS_Q_POST_FRONT_EN为0),因为OSQPostOpt()可以仿真这2个函数;
564 *       4、OSQPostOpt()在广播方式下,即将opt置为OS_POST_OPT_BROADCAST,函数的执行时间取
565 *          决于等待队列消息的任务的数目
566 *****
567 */
568
569 #if OS_Q_POST_OPT_EN > 0                      //允许生成 OSQPostOpt()代码
570 INT8U OSQPostOpt (OS_EVENT *pevent, void *msg, INT8U opt)
571 {
572     //向消息队列发送一则消息LIFO(消息队列指针、发送的消息、发送条件)
573     #if OS_CRITICAL_METHOD == 3              //中断函数被设定为模式3
574         OS_CPU_SR cpu_sr;
575     #endif
576     OS_Q *pq;                               //定义一个队列事件
577
578     #if OS_ARG_CHK_EN > 0                    //所有参数在指定的范围之内
579         if (pevent == (OS_EVENT *)0) {      //当消息队列指针为NULL,即0(空)
580             return (OS_ERR_PEVENT_NULL);    //pevent是空指针
581         }
582         if (msg == (void *)0) {              //检查消息队列是否为空,用户试发出空消息
583             return (OS_ERR_POST_NULL_PTR);  //用户发出空指针。根据规则,这里不支持空指针
584         }
585         if (pevent->OSEventType != OS_EVENT_TYPE_Q) { //事件类型是否为消息队列
586             return (OS_ERR_EVENT_TYPE);    //'pevent'不是指向消息队列的指针
587         }
588     #endif
589     OS_ENTER_CRITICAL();                     //关闭中断
590     if (pevent->OSEventGrp != 0x00) {        //是否有任务在等待该消息队列,索引值≠0
591         if ((opt & OS_POST_OPT_BROADCAST) != 0x00) { // 1) 发送消息给所有等待队列消息的任务
592             while (pevent->OSEventGrp != 0x00) {    //如果有任务在等待该消息队列
593                 OS_EventTaskRdy(pevent, msg, OS_STAT_Q); //发送消息给所有等待队列消息的任务
594             }
595         } else {                               // 2) 否则
596             OS_EventTaskRdy(pevent, msg, OS_STAT_Q); //如果没有广播请求,最优先进入请求
597         }
598         OS_EXIT_CRITICAL();                     //打开中断
599         OS_Sched();                             //任务调度函数,调用一个就绪的高优先级任务运行
600         return (OS_NO_ERR);                     //消息成功的放到消息队列中
601     }
602     pq = (OS_Q *)pevent->OSEventPtr;          //消息队列指针=当前事件指针
603     if (pq->OSQEntries >= pq->OSQSize) {      //消息队列当前消息数>=消息中可容纳的消息数
604         OS_EXIT_CRITICAL();                     //打开中断
605         return (OS_Q_FULL);                     //返回消息队列已满
606     }
607     if ((opt & OS_POST_OPT_FRONT) != 0x00) { // 1) 如果选择后进先出
608         if (pq->OSQOut == pq->OSQStart) {      //当插入指针=指针的起始地址(指针)

```

```

609         pq->OSQOut = pq->OSQEnd;           //插入指针跳转到最后地址(指针)
610     }
611     pq->OSQOut--;
612     *pq->OSQOut = msg;                      //插入当前消息(内容)
613 } else {                                   // 2) 否则, 选择先进先出
614     *pq->OSQIn++ = msg;                    //插入当前消息(内容)
615     if (pq->OSQIn == pq->OSQEnd) {          //当插入指针=指针的起始地址(指针)
616         pq->OSQIn = pq->OSQStart;          //插入指针跳转到最后地址(指针)
617     }
618 }
619 pq->OSQEntries++;                          //消息数加1
620 OS_EXIT_CRITICAL();                       //消息成功的放到消息队列中
621 return (OS_NO_ERR);                       //消息成功的放到消息队列中
622 }
623 #endif
624 /*$PAGE*/
625 /*
626 ****
627 * 取得消息队列的信息(QUERY A MESSAGE QUEUE)
628 *
629 * 描述: 取得消息队列的信息。用户程序必须建立一个OS_Q_DATA的数据结构, 该结构用来保存从消息队
630 * 列的事件控制块得到的数据。通过调用OSQQuery()函数可以知道任务是否在等待消息、有多少个
631 * 任务在等待消息、队列中有多少消息以及消息队列可以容纳的消息数。OSQQuery()函数还可以得
632 * 到即将被传递给任务的消息的信息。
633 *
634 * 参数: pevent 是指向即将接受消息的消息队列的指针。该指针的值在建立该消息邮箱时可以得到。
635 * (参考OSQCreate()函数)。
636 *
637 * pdata 是指向OS_Q_DATA数据结构的指针, 该数据结构包含下述成员:
638 * Void *OSMsg; // 下一个可用的消息
639 * INT16U OSNmsgs; // 队列中的消息数目
640 * INT16U OSQSize; // 消息队列的大小
641 * INT8U OSEventTbl[OS_EVENT_TBL_SIZE]; // 消息队列的等待队列
642 * INT8U OSEventGrp;
643 *
644 * 返回: OS_NO_ERR 调用成功;
645 * OS_ERR_EVENT_TYPE pevent不是指向消息队列的指针;
646 * OS_ERR_PEVENT_NULL pevent是空指针。
647 ****
648 */
649
650 #if OS_Q_QUERY_EN > 0 //允许生成 OSQQuery()代码
651 INT8U OSQQuery (OS_EVENT *pevent, OS_Q_DATA *pdata)
652 { //查询一个消息队列的当前状态(信号量指针、状态数据结构指针)
653 #if OS_CRITICAL_METHOD == 3 //中断函数被设定为模式3
654     OS_CPU_SR cpu_sr;
655 #endif
656     OS_Q *pq; //定义一个队列事件指针
657     INT8U *psrc; //定义8位pevent->OSEventTbl[0]的地址指针
658     INT8U *pdest; //定义8位pdata->OSEventTbl[0]的地址指针
659
660
661 #if OS_ARG_CHK_EN > 0 //所有参数在指定的范围之内
662     if (pevent == (OS_EVENT *)0) { //当消息队列指针为NULL, 即0(空)
663         return (OS_ERR_PEVENT_NULL); //pevent是空指针
664     }
665     if (pevent->OSEventType != OS_EVENT_TYPE_Q) { //当事件类型不是消息队列类型
666         return (OS_ERR_EVENT_TYPE); //pevent指针不是指向消息队列
667     }
668 #endif
669     OS_ENTER_CRITICAL(); //关闭中断
670     //将事件(消息队列)结构中的等待任务列表复制到pdata数据结构中
671     pdata->OSEventGrp = pevent->OSEventGrp; //等待事件的任务组中的内容传送到状态数据结构中
672     psrc = &pevent->OSEventTbl[0]; //保存pevent->OSEventTbl[0]对应的地址
673     pdest = &pdata->OSEventTbl[0]; //保存pdata->OSEventTbl[0]对应的地址
674 #if OS_EVENT_TBL_SIZE > 0 //当事件就绪对应表中的对应值>0时
675     *pdest++ = *psrc++; //地址指针下移一个类型地址, 获取信号量的值
676 #endif
677
678 #if OS_EVENT_TBL_SIZE > 1 //事件就绪对应表中的对应值>1时
679     *pdest++ = *psrc++; //地址指针继续下移一个类型地址, 获取信号量的值
680 #endif
681
682 #if OS_EVENT_TBL_SIZE > 2 //事件就绪对应表中的对应值>2时
683     *pdest++ = *psrc++; //地址指针继续下移一个类型地址, 获取信号量的值
684 #endif

```

```

685
686 #if OS_EVENT_TBL_SIZE > 3                //事件就绪对应表中的对应值>3时
687     *pdest++ = *psrc++;                  //地址指针继续下移一个类型地址，获取信号量的值
688 #endif
689
690 #if OS_EVENT_TBL_SIZE > 4                //事件就绪对应表中的对应值>4时
691     *pdest++ = *psrc++;                  //地址指针继续下移一个类型地址，获取信号量的值
692 #endif
693
694 #if OS_EVENT_TBL_SIZE > 5                //事件就绪对应表中的对应值>5时
695     *pdest++ = *psrc++;                  //地址指针继续下移一个类型地址，获取信号量的值
696 #endif
697
698 #if OS_EVENT_TBL_SIZE > 6                //事件就绪对应表中的对应值>6时
699     *pdest++ = *psrc++;                  //地址指针继续下移一个类型地址，获取信号量的值
700 #endif
701
702 #if OS_EVENT_TBL_SIZE > 7                //事件就绪对应表中的对应值>7时
703     *pdest = *psrc;                      //获取最后地址的信号量的值
704 #endif
705     pq = (OS_Q *)pevent->OSEventPtr;     //将队列事件指针保存到pq 中
706     if (pq->OSQEntries > 0) {             //如果消息队列指针中有消息
707         pdata->OSMsg = pq->OSQOut;        //将最早进入队列得消息复制到数据结构的OSMsg中
708     } else {
709         pdata->OSMsg = (void *)0;         //如果队列中没有消息(包含一个空指针)
710     }
711     pdata->OSNmsgs = pq->OSQEntries;      //消息队列中的消息数放置在数据结构的(OSNmsgs)中
712     pdata->OSQSize = pq->OSQSize;         //消息队列中的消息队列容量放置在数据结构得(OSQSize)中
713     OS_EXIT_CRITICAL();                  //打开中断
714     return (OS_NO_ERR);                  //返回调用成功
715 }
716 #endif                                  //结束OSQQuery () 函数
717
718 /*$PAGE*/
719 /*
720 ****
721 *
722 *
723 * 描述：初始化Q。
724 *
725 * 参数：无
726 *
727 * 返回：无
728 *
729 * 注意：
730 ****
731 */
732
733 void OS_QInit (void)
734 {
735     #if OS_MAX_QS == 1
736         OSQFreeList = &OSQTbl[0];        /* Only ONE queue! */
737         OSQFreeList->OSQPtr = (OS_Q *)0;
738     #endif
739
740     #if OS_MAX_QS >= 2
741         INT16U i;
742         OS_Q *pq1;
743         OS_Q *pq2;
744
745
746         pq1 = &OSQTbl[0];
747         pq2 = &OSQTbl[1];
748         for (i = 0; i < (OS_MAX_QS - 1); i++) { /* Init. list of free QUEUE control blocks */
749             pq1->OSQPtr = pq2;
750             pq1++;
751             pq2++;
752         }
753         pq1->OSQPtr = (OS_Q *)0;
754         OSQFreeList = &OSQTbl[0];
755     #endif
756 }
757 #endif                                  /* OS_Q_EN */
758

```

```

1  /*
2  ****
3  *                                     uC/OS-II实时控制内核
4  *                                     主要的包含文件
5  *                                     --内存管理项代码--
6  *
7  * 文    件: OS_MEM.C  内存管理项代码
8  * 作    者: Jean J. Labrosse
9  * 中文注解: 钟常慰 zhongcw @ 126.com  译注版本: 1.0 请尊重原版内容
10 ****
11 */
12
13 #ifndef OS_MASTER_FILE                //是否已定义OS_MASTER_FILE主文件
14 #include "includes.h"                //包含"includes.h"文件, 部分C语言头文件的汇总打包文件
15 #endif                                //定义结束
16
17 #if (OS_MEM_EN > 0) && (OS_MAX_MEM_PART > 0) //若两个条件满足时, 产生以下代码
18                                             //OS_MEM_EN允许 (1) 或者禁止 (0) 产生内存相关代码
19                                             //OS_MAX_MEM_PART 最多内存块的数目
20 ****
21 *                                     建立并初始化一块内存区 (CREATE A MEMORY PARTITION)
22 *
23 * 描述: 建立并初始化一块内存区。一块内存区包含指定数目的大小确定的内存块。程序可以包含这些内存
24 *       块并在用完后释放回内存区。
25 *
26 * 参数: addr      建立的内存区的起始地址。内存区可以使用静态数组或在初始化时使用malloc()函数建立。
27 *
28 *       nblks      需要的内存块的数目。每一个内存区最少需要定义两个内存块。
29 *
30 *       blksize    每个内存块的大小, 最少应该能够容纳一个指针。
31 *
32 *       err        是指向包含错误码的变量的指针。OSMemCreate()函数返回的错误码可能为下述几种:
33 *               OS_NO_ERR      成功建立内存区;
34 *               OS_MEM_INVALID_ADDR 非法地址, 即地址为空指针;
35 *               OS_MEM_INVALID_PART 没有空闲的内存区;
36 *               OS_MEM_INVALID_BLKS 没有为每一个内存区建立至少2个内存块;
37 *               OS_MEM_INVALID_SIZE 内存块大小不足以容纳一个指针变量。
38 * 返回: 返回指向内存区控制块的指针。如果没有剩余内存区, OSMemCreate()函数返回空指针。
39 *
40 * 注意: 必须首先建立内存区, 然后使用
41 ****
42 */
43 //建立并初始化一块内存区(起始地址、需要的内存块数目、每块内存块大小、返回错误的指针)
44 OS_MEM *OSMemCreate (void *addr, INT32U nblks, INT32U blksize, INT8U *err)
45 {
46     #if OS_CRITICAL_METHOD == 3                //中断函数被设定为模式3
47         OS_CPU_SR cpu_sr;
48     #endif
49     OS_MEM *pmem;                            //内存控制块指针
50     INT8U *pblk;                             //每块内存块的起始地址
51     void **plink;                             //链接起始地址
52     INT32U i;                                //内存包含的内存区数量
53
54
55     #if OS_ARG_CHK_EN > 0                    //所有参数在指定的范围之内
56         if (addr == (void *)0) {              //当内存起始地址为0时
57             *err = OS_MEM_INVALID_ADDR;        //错误显示为(非法地址, 即地址为空指针, 无效)
58             return ((OS_MEM *)0);
59         }
60         if (nblks < 2) {                      //每个内存分区至少有两个内存块
61             *err = OS_MEM_INVALID_BLKS;        //否则显示(没有为每一个内存区建立至少2个内存块)
62             return ((OS_MEM *)0);
63         }
64         if (blksize < sizeof(void *)) {        //每个内存块至少容得一个指针(链接指针)
65             *err = OS_MEM_INVALID_SIZE;        //否则显示(内存块大小不足以容纳一个指针变量)
66             return ((OS_MEM *)0);
67         }
68     #endif
69     OS_ENTER_CRITICAL();                    //关闭中断
70     pmem = OSMemFreeList;                  //内存控制块指针=空余内存控制块(链接)
71     if (OSMemFreeList != (OS_MEM *)0) {      //当内存链接控制块≠0, 即有空余控制块
72         OSMemFreeList = (OS_MEM *)OSMemFreeList->OSMemFreeList; //指向下一个空余链接控制块
73     }
74     OS_EXIT_CRITICAL();                    //打开中断
75     if (pmem == (OS_MEM *)0) {              //判断是否有空余内存控制块(为1有)
76         *err = OS_MEM_INVALID_PART;          //没有空闲的内存区

```

```

77     return ((OS_MEM *)0);          //返回Null, 建立内存失败
78 }
79 plink = (void **)addr;              //链接起始地址=内存分区起始地址
80 pblk = (INT8U *)addr + blksize;    //每块内存的起始地址=内存分区起始地址+每块内存块大小
81 for (i = 0; i < (nblks - 1); i++) { //循环体(需要的内存块数目(次数))?
82     *plink = (void *)pblk;          //链接起始地址(内容)=每块内存块的起始地址(内容)?
83     plink = (void **)pblk;          //链接起始地址=内存块的起始地址指针(内容)?
84     pblk = pblk + blksize;          //内存块的起始地址=自己+每块内存块大小?
85 }
86 *plink = (void *)0;                //链接起始地址=0
87 OS_ENTER_CRITICAL();               //关闭中断
88 pmem->OSMemAddr = addr;             //内存区指针=分区起始地址
89 pmem->OSMemFreeList = addr;         //下一空余控制块=分区起始地址
90 pmem->OSMemNFree = nblks;           //分区中内存块大小=需要的内存块数目
91 pmem->OSMemNBlks = nblks;           //总的内存块数量=需要的内存块数目
92 pmem->OSMemBlkSize = blksize;       //空余内存块数量=每块内存块大小
93 OS_EXIT_CRITICAL();                //打开中断
94 *err = OS_NO_ERR;                  //成功建立内存区
95 return (pmem);                     //返回(内存控制块指针)
96 }
97 /*$PAGE*/
98 /*
99 ****
100 *          从内存区分配一个内存块(GET A MEMORY BLOCK)
101 *
102 * 描述: 用于从内存区分配一个内存块。用户程序必须知道所建立的内存块的大小, 同时用户程序必须在使用
103 *        完内存块后释放内存块。使用OSMemGet()函数释放内存块。可以多次调用OSMemGet()函数。
104 *
105 * 参数: pmem    是指向内存区控制块的指针, 可以从OSMemCreate()函数返回得到。
106 *
107 *        err     是指向包含错误码的变量的指针。OSMemGet()函数返回的错误码可能为下述几种:
108 *              OS_NO_ERR          成功得到一个内存块;
109 *              OS_MEM_NO_FREE_BLKS 内存区已经没有空间分配给内存块;
110 *              OS_MEM_INVALID_PMEM 'pmem' 是空指针。
111 *
112 * 返回: 返回指向内存区块的指针。如果没有空间分配给内存块, OSMemGet()函数返回空指针。
113 *
114 * 注意: 必须首先建立内存区, 然后使用
115 ****
116 */
117
118 void *OSMemGet (OS_MEM *pmem, INT8U *err) //从内存区分配一个内存块(内存区控制块的指针、错误指针)
119 {
120     #if OS_CRITICAL_METHOD == 3          //中断函数被设定为模式3
121         OS_CPU_SR cpu_sr;
122     #endif
123     void *pblk;                          //内存块的起始地址
124
125     #if OS_ARG_CHK_EN > 0                //所有的参数都是在指定的范围内
126         if (pmem == (OS_MEM *)0) {      //指向内存区控制块的指针不能为空指针
127             *err = OS_MEM_INVALID_PMEM; // 'pmem' 是空指针
128             return ((OS_MEM *)0);        //返回Null
129         }
130     #endif
131     OS_ENTER_CRITICAL();                 //关闭中断
132     if (pmem->OSMemNFree > 0) {           //检查内存分区中是否有空余的内存块(必须大于0)
133         pblk = pmem->OSMemFreeList;      //刷新空余内存块链接表
134         pmem->OSMemFreeList = *(void **)pblk; //将链接表头指针后移1个元素
135         pmem->OSMemNFree--;               //将空余内存块数减1
136         OS_EXIT_CRITICAL();              //打开中断
137         *err = OS_NO_ERR;                 //成功得到一个内存块
138         return (pblk);                    //返回(内存块的起始地址)
139     }
140     OS_EXIT_CRITICAL();                  //打开中断
141     *err = OS_MEM_NO_FREE_BLKS;          //内存区已经没有空间分配给内存块
142     return ((void *)0);                  //返回(没有空间分配给内存块)
143 }
144 /*$PAGE*/
145 /*
146 ****
147 *          释放一个内存块 (RELEASE A MEMORY BLOCK)
148 *
149 * 描述: 释放一个内存块, 内存块必须释放回原先申请的内存区。
150 *
151 * 参数: pmem    是指向内存区控制块的指针, 可以从OSMemCreate()函数 返回得到。
152 *

```



```

153 *      pblk    是指向将被释放的内存块的指针。
154 *
155 *  返回: OS_NO_ERR          成功释放内存块;
156 *      OS_MEM_FULL        内存区已经不能再接受更多释放的内存块。这种情况说明用户程序出现
157 *                          了错误, 释放了多于用OSMemGet() 函数得到的内存块
158 *      OS_MEM_INVALID_PMEM 'pmem' 是空指针;
159 *      OS_MEM_INVALID_PBLK //指向将被释放的内存块的指针不能为空指针
160 *
161 *  注意: 1) 必须首先建立内存区, 然后使用;
162 *        2) 内存块必须释放回原先申请的内存区。
163 *****
164 */
165
166 INT8U OSMemPut (OS_MEM *pmem, void *pblk)
167 {
168     //释放一个内存块(内存区控制块的指针、被释放的内存块的指针)
169     //中断函数被设定为模式3
170     #if OS_CRITICAL_METHOD == 3
171         OS_CPU_SR cpu_sr;
172     #endif
173
174     //所有的参数都是在指定的范围内
175     if (pmem == (OS_MEM *)0) {
176         //指向内存区控制块的指针不能为空指针
177         return (OS_MEM_INVALID_PMEM);
178     }
179     // 'pmem' 是空指针
180
181     if (pblk == (void *)0) {
182         //指向将被释放的内存块的指针不能为空指针
183         return (OS_MEM_INVALID_PBLK);
184     }
185     // 'pblk' 是空指针
186
187     #endif
188     OS_ENTER_CRITICAL();
189     if (pmem->OSMemNFree >= pmem->OSMemNBKs) {
190         //分区中内存块大小 >= 总的内存块数量(已满)
191         OS_EXIT_CRITICAL();
192         //打开中断
193         return (OS_MEM_FULL);
194     }
195     //内存区已经不能再接受更多释放的内存块
196
197     //如果未满, 将释放的内存块插入到该分区的空余内存块链接表中
198     *(void **)pblk = pmem->OSMemFreeList;
199     pmem->OSMemFreeList = pblk;
200     //将链接表头指针指向被释放的内存块的指针
201     pmem->OSMemNFree++;
202     //将分区中空余的内存块总数加1
203     OS_EXIT_CRITICAL();
204     return (OS_NO_ERR);
205     //成功释放内存块
206 }
207
208 /*$PAGE*/
209 /*
210 *****
211 *      得到内存区的信息(QUERY MEMORY PARTITION)
212 *
213 *  描述: 得到内存区的信息。该函数返回OS_MEM结构包含的信息, 但使用了一个新的OS_MEM_DATA的数据结
214 *        构。OS_MEM_DATA数据结构还包含了正被使用的内存块数目的域。
215 *
216 *  参数: pmem    是指向内存区控制块的指针, 可以从OSMemCreate() 函数 返回得到。
217 *
218 *      pdata    是指向OS_MEM_DATA数据结构的指针, 该数据结构包含了以下的域:
219 *      Void      OSAddr;          //指向内存区起始地址的指针
220 *      Void      OSFreeList;      //指向空闲内存块列表起始地址的指针
221 *      INT32U    OSBlkSize;       //每个内存块的大小
222 *      INT32U    OSNBKs;         //该内存区的内存块总数
223 *      INT32U    OSNFree;        //空闲的内存块数目
224 *      INT32U    OSNUsed;        //使用的内存块数目
225 *
226 *  返回: OS_NO_ERR          存储块有效, 返回用户应用程序;
227 *      OS_MEM_INVALID_PMEM  'pmem' 是空指针;
228 *      OS_MEM_INVALID_PDATA  pdata是空指针。
229 *
230 *  注意: 1) 必须首先建立内存区, 然后使用;
231 *****
232 */
233
234 #if OS_MEM_QUERY_EN > 0
235     //允许生成 OSMemQuery() 函数
236     INT8U OSMemQuery (OS_MEM *pmem, OS_MEM_DATA *pdata)
237     {
238         //查询内存区的信息(内存区控制块的指针、保存数据的指针)
239         //中断函数被设定为模式3
240         #if OS_CRITICAL_METHOD == 3
241             OS_CPU_SR cpu_sr;
242         #endif
243
244         //所有的参数都是在指定的范围内
245         if (pmem == (OS_MEM *)0) {
246             //指向内存区控制块的指针不能为空指针

```


[illegible]

```

296 *****结束*****
297

```

```

1  /*
2  ****
3  *                                     uC/OS-II实时控制内核
4  *                                     主要的包含文件
5  *                                     信号量程序函数
6  *
7  * 文    件: OS_SEM.C      信号量程序函数
8  * 作    者: Jean J. Labrosse
9  * 中文注解: 钟常慰 zhongcw @ 126.com 译注版本: 1.0 请尊重原版内容
10 ****
11 */
12
13 #ifndef OS_MASTER_FILE //是否已定义OS_MASTER_FILE主文件
14 #include "includes.h" //包含"includes.h"文件, 部分C语言头文件的汇总打包文件
15 #endif //定义结束
16
17 #if OS_SEM_EN > 0 //条件编译: 当OS_SEM_EN允许产生信号量程序代码
18 /*
19 ****
20 *                                     无条件地等待请求一个信号量
21 *
22 * 描述: 该函数是查看资源是否使用或事件是否发生。中断调用该函数查询信号量。
23 *       不同于OSSemPend()函数, 如果资源不可使用, OSSemAccept()函数并不挂起任务。
24 *
25 * 参数: pevent 指向需要保护地共享资源地信号量。当建立信号量时, 用户得到该指针
26 *
27 * 返回: 当调用OSSemAccept()函数时;
28 *       共享资源信号量的值 > 0, 则说明共享资源可以使用, 这个值被返回调用者, 信号量的值减1;
29 *       共享资源信号量的值 = 0, 则说明资源不能使用, 返回0。
30 ****
31 */
32
33 #if OS_SEM_ACCEPT_EN > 0 //允许生成 OSSemAccept()函数
34 INT16U OSSemAccept (OS_EVENT *pevent) //无条件地等待请求一个信号量函数
35 {
36     #if OS_CRITICAL_METHOD == 3 //中断函数被设定为模式3
37         OS_CPU_SR cpu_sr;
38     #endif
39     INT16U cnt; //信号量的内容暂时存储变量
40
41
42     #if OS_ARG_CHK_EN > 0 //所有参数必须在指定的参数内
43         if (pevent == (OS_EVENT *)0) { //当信号量指针为NULL时, 返回0, 空指针
44             return (0);
45         }
46         if (pevent->OSEventType != OS_EVENT_TYPE_SEM) { //当事件类型≠信号量类型
47             return (0); //返回0
48         }
49     #endif
50     OS_ENTER_CRITICAL(); //关闭中断
51     cnt = pevent->OSEventCnt; //取信号值
52     if (cnt > 0) { //当信号值>0时, 该值有效
53         pevent->OSEventCnt--; //信号量减1
54     }
55     OS_EXIT_CRITICAL(); //打开中断
56     return (cnt); //返回信号值
57 }
58 #endif
59
60 /*$PAGE*/
61 /*
62 ****
63 *                                     建立一个信号量
64 *
65 * 描述: 建立并初始化一个信号量。信号量的作用为:
66 *       1、允许一个任务与其它任务或中断同步;
67 *       2、取得共享资源的使用权;
68 *       3、标志事件的发生
69 *
70 * 参数: cnt 建立信号量的初始值, 可以为0 ~ 65 535的任何值
71 *
72 * 注意: 必须先建立信号量, 然后才能使用
73 *
74 * 返回: != (void *)0 返回指向分配给所建立的消息邮箱的事件控制块指针;
75 *       == (void *)0 如果没有可用的事件控制块, 返回空指针
76 ****

```

```

76 */
77
78 OS_EVENT *OSSemCreate (INT16U cnt)           //建立并初始化一个信号量(输入一个信号量值)
79 {
80 #if OS_CRITICAL_METHOD == 3                 //中断函数被设定为模式3
81     OS_CPU_SR cpu_sr;
82 #endif
83     OS_EVENT *pevent;                       //建立信号量的初始值, 可以在0至65535之间
84
85
86     if (OSIntNesting > 0) {                 //中断嵌套数>0时, 表示还有中断任务在运行
87         return ((OS_EVENT *)0);           //返回0;
88     }
89     OS_ENTER_CRITICAL();                    //关闭中断
90     pevent = OSEventFreeList;               //pevent=空余事件管理列表
91     if (OSEventFreeList != (OS_EVENT *)0) { //如果有空余事件管理块
92         OSEventFreeList = (OS_EVENT *)OSEventFreeList->OSEventPtr;
93     }                                       //空余事件控制链表指向下一个空余事件控制块
94     OS_EXIT_CRITICAL();                     //打开中断
95     if (pevent != (OS_EVENT *)0) {         //如果有事件控制块ECB可用
96         pevent->OSEventType = OS_EVENT_TYPE_SEM; //事件类型=信号类型
97         pevent->OSEventCnt = cnt;           //将信号量值存入事件管理块中(信号量的计数器)
98         OS_EventWaitListInit(pevent);      //初始化一个事件控制块
99     }
100     return (pevent);                       //返回指针
101 }
102
103 /*$PAGE*/
104 /*
105 ****
106 *                                     删除一个信号量
107 * 描述: 用于删除一个信号量。
108 *       使用本函数有风险, 因为多任务中的其它任务可能还想使用这个信号量, 必须特别小心。
109 *       一般而言, 在删除信号量之前, 应该先删除所有可能会用到的这个信号量的任务。
110 *
111 * 参数: pevent  指向信号量指针。该指针的值在建立该信号量时得到。(参见OSSemCreate ()函数)
112 *
113 *       opt      该选项定义信号量的删除条件。可以选择只能在已经没有任何任务在等待该信号量时, 才能删除该信号量(OS_DEL_NO_PEND); 或者, 不管有没有任务在等待该信号量, 立即删除该信号量(OS_DEL_ALWAYS), 在这种情况下, 所有等待该信号量的任务都立即进入就绪态
114 *
115 *
116 *
117 *       err      指向包含错误码的变量的指针。返回的错误码可能为以下几种:
118 *               OS_NO_ERR          调用成功, 信号量已被删除;
119 *               OS_ERR_DEL_ISR     试图在中断服务子程序中删除信号量;
120 *               OS_ERR_INVALID_OPT 没有将opt参数定义为2种合法的参数之一;
121 *               OS_ERR_TASK_WAITING 有一个或一个以上的任务在等待信号量;
122 *               OS_ERR_EVENT_TYPE  pevent不是指向信号量的指针;
123 *               OS_ERR_PEVENT_NULL 已经没有可用的OS_EVENT数据结构了。
124 *
125 * 返回: pevent  如果信号量已被删除, 返回空指针;
126 *             若信号量没有删除, 则返回pevent(信号量指针), 可查看出错代码。
127 *
128 * 注意: 1) 使用此函数必须特别小心, 因为多任务中的其它任务可能还想使用这个信号量;
129 *       2) 当挂起的任务进入就绪态时, 中断是关闭的, 这意味着中断延迟时间与等待信号量的任务数有关。
130 ****
131 */
132
133 #if OS_SEM_DEL_EN > 0                       //允许生成 OSSemDel() 代码
134 OS_EVENT *OSSemDel (OS_EVENT *pevent, INT8U opt, INT8U *err)
135 {
136 #if OS_CRITICAL_METHOD == 3                 //中断函数被设定为模式3
137     OS_CPU_SR cpu_sr;
138 #endif
139     BOOLEAN tasks_waiting;                 //定义布尔量, 任务等待条件
140
141
142     if (OSIntNesting > 0) {                 //中断嵌套数>0时, 表示还有中断任务在运行
143         *err = OS_ERR_DEL_ISR;             //错误等于(试图在中断程序中删除一个信号量事件)
144         return (pevent);                   //返回信号量指针
145     }
146 #if OS_ARG_CHK_EN > 0                       //所有参数在指定的范围之内
147     if (pevent == (OS_EVENT *)0) {         //当信号量指针为NULL, 即0(空)
148         *err = OS_ERR_PEVENT_NULL;        //错误等于(已经没有可用的OS_EVENT数据结构了)
149         return (pevent);                   //返回信号量指针
150     }
151     if (pevent->OSEventType != OS_EVENT_TYPE_SEM) { //当事件类型不否是信号量类型

```

```

152     *err = OS_ERR_EVENT_TYPE;           //pevent指针不是指向信号量
153     return (pevent);                   //返回信号量指针
154 }
155 #endif
156 OS_ENTER_CRITICAL();                  //关闭中断
157 if (pevent->OSEventGrp != 0x00) {      //事件等待标志, 索引值≠0, 有任务在等待
158     tasks_waiting = TRUE;              //有任务在等待=1(TRUE真)
159 } else {
160     tasks_waiting = FALSE;              //否则, 没有任务在等待=0, (FALSE假)
161 }
162 switch (opt) {                          //条件选择
163     case OS_DEL_NO_PEND:                 // 1) 没有任务在等待该信号量
164         if (tasks_waiting == FALSE) {    // 如果没有事件在等待
165             pevent->OSEventType = OS_EVENT_TYPE_UNUSED; //事件类型=空闲
166             pevent->OSEventPtr = OSEventFreeList; //信号量对应的指针=空余块链接表
167             OSEventFreeList = pevent; //空余块链接表=当前事件指针
168             OS_EXIT_CRITICAL();           //打开中断
169             *err = OS_NO_ERR;             //错误等于(成功删除)
170             return ((OS_EVENT *)0);       //返回0
171         } else {                          //否则, 有任务在等待
172             OS_EXIT_CRITICAL();           //打开中断
173             *err = OS_ERR_TASK_WAITING;   //错误等于(有一个或一个以上的任务在等待信号量)
174             return (pevent);              //返回信号量指针
175         }
176
177     case OS_DEL_ALWAYS:                  // 2) 多任务等待, 尽管有任务在等待, 还是要删除
178         while (pevent->OSEventGrp != 0x00) { //等待标志≠0, 还是要删除
179             OS_EventTaskRdy(pevent, (void *)0, OS_STAT_SEM);
180             //使一个任务进入就绪态
181             pevent->OSEventType = OS_EVENT_TYPE_UNUSED; //事件类型=空闲
182             pevent->OSEventPtr = OSEventFreeList; //信号量对应的指针=空余块链接表
183             OSEventFreeList = pevent; //空余块链接表=当前事件指针
184             OS_EXIT_CRITICAL();           //打开中断
185             if (tasks_waiting == TRUE) { //当任务等待=1, 真
186                 OS_Sched();              //任务调度, 最高优先级进入运行状态
187             }
188             *err = OS_NO_ERR;             //错误等于(成功删除)
189             return ((OS_EVENT *)0);       //返回0
190
191         default:                          // 3) 当以上两种情况都不是
192             OS_EXIT_CRITICAL();           //打开中断
193             *err = OS_ERR_INVALID_OPT;    //错误等于(没有将opt参数定义为2种合法的参数之一)
194             return (pevent);              //返回信号量指针
195     }
196 }
197 #endif
198
199 /*$PAGE*/
200 /*
201 ****
202 *                                     等待一个信号量
203 * 描述: 等待一个信号量。
204 * 任务试图取得共享资源使用权、任务需要与其它任务或中断同步及任务需要等待特定事件的发生的场合。
205 * 若任务调用该函数, 且信号量的值>0, 那么OSSemPend()递减该值并返回该值;
206 * 若任务调用该函数, 且信号量的值=0, 那么OSSemPend()函数将任务加入该信号量的等待列表中。
207 *
208 * 参数: pevent    指向信号量指针。该指针的值在建立该信号量时得到。(参见OSSemCreate ()函数)
209 *
210 *  timeout    允许任务在经过指定数目的时钟节拍后还没有得到需要的信号量时; 恢复运行状态。如果
211 *             该值为0。则表示任务将持续地等待信号量。最长等待时间为65 535个时钟节拍。这个时
212 *             间长度并不是严格的, 可能存在一个时间节拍的误差, 因为自由一个时钟节拍结束后, 才
213 *             会给定义的等待超时时钟节拍减1。
214 *  err        指向包含错误码的变量的指针。返回的错误码可能为以下几种;
215 *
216 *             OS_NO_ERR        成功, 信号量是可用的;
217 *             OS_TIMEOUT       信号量没有在指定的周期数内置位;
218 *             OS_ERR_EVENT_TYPE pevent不是指向信号量的指针;
219 *             OS_ERR_PEND_ISR   在中断中调用该函数。虽然规定了不允许在中断中调用该函数, 但
220 *             ucos仍然包含了检测这种情况的功能;
221 *             OS_ERR_PEVENT_NULL pevent是空指针。
222 *  返回: 无
223 *  注意: 必须先建立信号量, 然后才能使用。
224 ****
225 */
226 //等待一个信号量函数(信号量指针、允许等待的时钟节拍、代码错误指针)
227 void OSSemPend (OS_EVENT *pevent, INT16U timeout, INT8U *err)

```

```

228 {
229 #if OS_CRITICAL_METHOD == 3                //中断函数被设定为模式3
230     OS_CPU_SR  cpu_sr;
231 #endif
232
233
234     if (OSIntNesting > 0) {                //中断嵌套数>0时，表示还有中断任务在运行
235         *err = OS_ERR_PEND_ISR;            //错误等于(试图在中断程序中等待一个信号量事件)
236         return;                            //返回
237     }
238 #if OS_ARG_CHK_EN > 0                    //所有参数在指定的范围之内
239     if (pevent == (OS_EVENT *)0) {        //当信号量指针为NULL，即0(空)
240         *err = OS_ERR_PEVENT_NULL;        //pevent是空指针
241         return;                            //返回
242     }
243     if (pevent->OSEventType != OS_EVENT_TYPE_SEM) { //当事件类型不否是信号量类型
244         *err = OS_ERR_EVENT_TYPE;        //pevent指针不是指向信号量
245         return;                            //返回
246     }
247 #endif
248     OS_ENTER_CRITICAL();                  //关闭中断
249     if (pevent->OSEventCnt > 0) {          //当信号量计数器>0时，
250         pevent->OSEventCnt--;             //信号量计数器减1
251         OS_EXIT_CRITICAL();               //打开中断
252         *err = OS_NO_ERR;                 //错误等于(成功，信号量是可用的)
253         return;                           //返回
254     }
255
256     OSTCBCur->OSTCBStat |= OS_STAT_SEM; //将任务状态置1，进入睡眠状态，只能通过信号量唤醒
257     OSTCBCur->OSTCBDly = timeout;        //最长等待时间=timeout，递减式
258     OS_EventTaskWait(pevent);            //使任务进入等待时间唤醒状态
259     OS_EXIT_CRITICAL();                  //打开中断
260     OS_Sched();                          //进入调度任务，使就绪态优先级最高任务运行
261     OS_ENTER_CRITICAL();                  //关闭中断
262     if (OSTCBCur->OSTCBStat & OS_STAT_SEM) { //检查任务状态是否还是在睡眠状态，即信号量没有唤醒
263         OS_EventTO(pevent);               //如果没有等到信号量，由等待事件返回
264         OS_EXIT_CRITICAL();               //打开中断
265         *err = OS_TIMEOUT;                //错误等于(信号量没有在指定的周期数内置位)
266         return;                           //返回
267     }
268     OSTCBCur->OSTCBEventPtr = (OS_EVENT *)0; //将信号量ECB的指针从该任务控制块中删除
269     OS_EXIT_CRITICAL();                  //打开中断
270     *err = OS_NO_ERR;                    //错误等于(成功，信号量是可用的)
271 }
272 /*$PAGE*/
273 /*
274 ****
275 *                                     发出一个信号量
276 *
277 * 描述：置位指定的信号量。如果指定的信号量是0或大于0，该函数则递增该信号量并返回；
278 *       如果有任何任务在等待信号量，那么最高优先级任务将得到该信号量并进入就绪态；
279 *       如果被唤醒的任务就是最高优先级的就绪态任务，则任务调度函数将进入任务调度。
280 *
281 * 参数：pevent    指向信号量指针。该指针的值在建立该信号量时得到。(参见OSSemCreate ()函数)
282 *
283 * 返回：OS_NO_ERR      信号量成功的置位；
284 *       OS_SEM_OVF     信号量的值溢出；
285 *       OS_ERR_EVENT_TYPE pevent不是指向信号量的指针；
286 *       OS_ERR_PEVENT_NULL pevent是空指针。
287 ****
288 */
289
290 INT8U OSEmPost (OS_EVENT *pevent)        //发出一个信号量函数(信号量指针)
291 {
292 #if OS_CRITICAL_METHOD == 3                //中断函数被设定为模式3
293     OS_CPU_SR  cpu_sr;
294 #endif
295
296
297 #if OS_ARG_CHK_EN > 0                    //所有参数在指定的范围之内
298     if (pevent == (OS_EVENT *)0) {        //当信号量指针为NULL，即0(空)
299         return (OS_ERR_PEVENT_NULL);      //pevent是空指针
300     }
301     if (pevent->OSEventType != OS_EVENT_TYPE_SEM) { //当事件类型不否是信号量类型
302         return (OS_ERR_EVENT_TYPE);      //pevent指针不是指向信号量
303     }

```



```

304 #endif
305     OS_ENTER_CRITICAL(); //关闭中断
306     if (pevent->OSEventGrp != 0x00) { //有任务在等待信号量，等待事件的任务组=0
307         OS_EventTaskRdy(pevent, (void *)0, OS_STAT_SEM); //使任务进入就绪态
308         OS_EXIT_CRITICAL(); //打开中断
309         OS_Sched(); //进入调度任务，使就绪态优先级最高任务运行
310         return (OS_NO_ERR); //返回(信号量成功的置位)
311     }
312     if (pevent->OSEventCnt < 65535) { //当信号量值< 65535时，
313         pevent->OSEventCnt++; //信号量计数加1
314         OS_EXIT_CRITICAL(); //打开中断
315         return (OS_NO_ERR); //返回(信号量成功的置位)
316     }
317     OS_EXIT_CRITICAL(); //打开中断
318     return (OS_SEM_OVF); //返回(信号量的值溢出)
319 }
320 /*
321 ****
322 * 查询一个信号量的当前状态
323 *
324 * 描述：用于获取某个信号量的信息。在使用该函数之前，应用程序先要建立OS_SEM_DATA的数据结构，用来保存
325 * 从信号量的事件控制中取得的数据。使用该函数可以得知，是否有以及多少任务目前位于信号量的任务
326 * 等待对列中(查询OSEventTbl()域中的数目)，并还可以获取信号量的值。
327 *
328 * 参数：pevent 指向信号量指针。该指针的值在建立该信号量时得到。(参见OSSemCreate()函数)
329 *
330 * pdata 一个指向数据结构OS_SEM_DATA的指针。
331 *
332 * 返回：OS_NO_ERR 用成功；
333 * OS_ERR_EVENT_TYPE pevent不是指向信号量的指针；
334 * OS_ERR_PEVENT_NULL pevent是空指针。
335 ****
336 */
337
338 #if OS_SEM_QUERY_EN > 0 //允许生成 OS_SemQuery()代码
339 INT8U OS_SemQuery (OS_EVENT *pevent, OS_SEM_DATA *pdata)
340 { //查询一个信号量的当前状态(信号量指针、状态数据结构指针)
341     #if OS_CRITICAL_METHOD == 3 //中断函数被设定为模式3
342         OS_CPU_SR cpu_sr;
343     #endif
344     INT8U *psrc; //定义8位pevent->OSEventTbl[0]的地址指针
345     INT8U *pdest; //定义8位pdata->OSEventTbl[0]的地址指针
346
347
348     #if OS_ARG_CHK_EN > 0 //所有参数在指定的范围之内
349         if (pevent == (OS_EVENT *)0) { //当信号量指针为NULL，即0(空)
350             return (OS_ERR_PEVENT_NULL); //pevent是空指针
351         }
352         if (pevent->OSEventType != OS_EVENT_TYPE_SEM) { //当事件类型不是信号量类型
353             return (OS_ERR_EVENT_TYPE); //pevent指针不是指向信号量
354         }
355     #endif
356     OS_ENTER_CRITICAL(); //关闭中断
357     //将事件(信号量)结构中的等待任务列表复制到pdata数据结构中
358     pdata->OSEventGrp = pevent->OSEventGrp; //等待事件的任务组中的内容传送到状态数据结构中
359     psrc = &pevent->OSEventTbl[0]; //保存pevent->OSEventTbl[0]对应的地址
360     pdest = &pdata->OSEventTbl[0]; //保存pdata->OSEventTbl[0]对应的地址
361     #if OS_EVENT_TBL_SIZE > 0 //当事件就绪对应表中的对应值>0时
362         *pdest++ = *psrc++; //地址指针下移一个类型地址，获取信号量的值
363     #endif
364
365     #if OS_EVENT_TBL_SIZE > 1 //事件就绪对应表中的对应值>1时
366         *pdest++ = *psrc++; //地址指针继续下移一个类型地址，获取信号量的值
367     #endif
368
369     #if OS_EVENT_TBL_SIZE > 2 //事件就绪对应表中的对应值>2时
370         *pdest++ = *psrc++; //地址指针继续下移一个类型地址，获取信号量的值
371     #endif
372
373     #if OS_EVENT_TBL_SIZE > 3 //事件就绪对应表中的对应值>3时
374         *pdest++ = *psrc++; //地址指针继续下移一个类型地址，获取信号量的值
375     #endif
376
377     #if OS_EVENT_TBL_SIZE > 4 //事件就绪对应表中的对应值>4时
378         *pdest++ = *psrc++; //地址指针继续下移一个类型地址，获取信号量的值
379     #endif

```



```
380
381 #if OS_EVENT_TBL_SIZE > 5           //事件就绪对应表中的对应值>5时
382     *pdest++                        = *psrc++;      //地址指针继续下移一个类型地址，获取信号量的值
383 #endif
384
385 #if OS_EVENT_TBL_SIZE > 6           //事件就绪对应表中的对应值>6时
386     *pdest++                        = *psrc++;      //地址指针继续下移一个类型地址，获取信号量的值
387 #endif
388
389 #if OS_EVENT_TBL_SIZE > 7           //事件就绪对应表中的对应值>7时
390     *pdest                          = *psrc;        //获取最后地址的信号量的值
391 #endif
392     pdata->OSCnt                    = pevent->OSEventCnt; //数据计数=当前信号事件对应的计数值(?任务数)
393     OS_EXIT_CRITICAL();              //打开中断
394     return (OS_NO_ERR);              //返回成功运行
395 }
396 #endif                               //OS_SEM_QUERY_EN函数结束
397 #endif                               //OS_SEM_EN文件结束
398
```

```

1  /*
2  ****
3  *                                     uC/OS-II实时控制内核
4  *                                     主要的包含文件
5  *                                     任务管理
6  *
7  * 文    件: OS_TASK.C      任务管理代码
8  * 作    者: Jean J. Labrosse
9  * 中文注解: 钟常慰 zhongcw @ 126.com 译注版本: 1.0 请尊重原版内容
10 ****
11 */
12
13 #ifndef OS_MASTER_FILE                //是否已经定义OS_MASTER_FILE
14 #include "includes.h"                //包含"includes.h"文件
15 #endif                                //结束定义
16
17 /*
18 ****
19 *                                     改变一个任务的优先级 (CHANGE PRIORITY OF A TASK)
20 *
21 * 描述: 改变一个任务的优先级。
22 *
23 * 参数: oldp      是任务原先的优先级。
24 *
25 *        newp      是任务的新优先级。
26 *
27 * 返回: OS_NO_ERR      任务优先级成功改变。
28 *        OS_PRIO_INVALID 参数中的任务原先优先级或新优先级大于或等于OS_LOWEST_PRIO。
29 *        (i.e. >= OS_LOWEST_PRIO)
30 *        OS_PRIO_EXIST   优先级为PIP的任务已经存在;
31 *        OS_PRIO_ERR      参数中的任务原先优先级不存在。
32 *
33 * 注意: 参数中的新优先级必须是没有使用过的, 否则会返回错误码. 在OSTaskChangePrio()中还会先
34 *        判断要改变优先级的任务是否存在。
35 *
36 ****
37 */
38
39 #if OS_TASK_CHANGE_PRIO_EN > 0        //允许生成OSTaskChangePrio()函数
40 INT8U OSTaskChangePrio (INT8U oldprio, INT8U newprio)
41 {
42     #if OS_CRITICAL_METHOD == 3        //允许生成OSTaskCreate()函数
43         OS_CPU_SR      cpu_sr;
44     #endif
45
46     #if OS_EVENT_EN > 0                //消息事件是否 > 0
47         OS_EVENT      *pevent;        //定义事件指针
48     #endif
49
50     OS_TCB      *ptcb;                //定义消息事件的任务控制块指针
51     INT8U      x;                    //优先级低3位值
52     INT8U      y;                    //优先级高3位值
53     INT8U      bitx;                //优先级低3位值计算对应值
54     INT8U      bity;                //优先级高3位值计算索引值
55
56
57
58     #if OS_ARG_CHK_EN > 0                //所有参数必须在指定的参数内
59         //当旧任务 >= 最低优先级 并且 旧任务不是本身 并且新任务>= 最低优先级
60         if ((oldprio >= OS_LOWEST_PRIO && oldprio != OS_PRIO_SELF) ||
61             newprio >= OS_LOWEST_PRIO) {
62             return (OS_PRIO_INVALID);    //参数中的任务原先优先级或新优先级大于或等于OS_LOWEST_PRIO
63         }
64     #endif
65     OS_ENTER_CRITICAL();                //关闭中断
66     if (OSTCBPrioTbl[newprio] != (OS_TCB *)0) { //确认新任务优先级未被使用, 即就绪态为0
67         OS_EXIT_CRITICAL();            //打开中断
68         return (OS_PRIO_EXIST);        //返回新任务(优先级为PIP的任务已经存在)
69     } else {
70         OSTCBPrioTbl[newprio] = (OS_TCB *)1; //新任务优先级未被使用, 保留它(为1)
71         OS_EXIT_CRITICAL();            //打开中断
72         //预先计算新任务优先级任务控制块OS_TCB的某些值
73         y = newprio >> 3;            //保留优先级高3位(3-5位)
74         bity = OSMAP_TBL[y];        //计算索引值
75         x = newprio & 0x07;        //保存优先级低3位(0-2位)
76         bitx = OSMAP_TBL[x];        //计算对应值

```

```

77     OS_ENTER_CRITICAL(); //关闭中断
78     if (oldprio == OS_PRIO_SELF) { //要改变的是否使旧任务本身
79         oldprio = OSTCBCur->OSTCBPrio; //如果是(正在运行的优先级(旧任务本身的优先级))
80     }
81     if ((ptcb = OSTCBPrioTbl[oldprio]) != (OS_TCB *)0) { //变更的旧任务必须存在(1即就绪)
82         OSTCBPrioTbl[oldprio] = (OS_TCB *)0; //旧任务就绪态去除它(为0)
83         if ((OSRdyTbl[ptcb->OSTCBY] & ptcb->OSTCBBitX) != 0x00) {
84             //如果该任务处于就绪态,那么必须在当前的优先级下,从就绪表中移除该任务,然后在新
85             //的优先级下,将该任务插入到就绪表中。
86             if ((OSRdyTbl[ptcb->OSTCBY] & ~ptcb->OSTCBBitX) == 0x00) {
87                 OSRdyGrp &= ~ptcb->OSTCBBitY;
88             }
89             OSRdyGrp |= bity; //利用预先计算值将任务插入到就绪表中
90             OSRdyTbl[y] |= bitx;
91 #if OS_EVENT_EN > 0 //消息事件是否 > 0
92         } else { //任务未就绪,拾取任务事件指针
93             if ((pevent = ptcb->OSTCBEventPtr) != (OS_EVENT *)0) { //任务事件表为1(有消息)
94                 //如果任务正在等待某一事件的发生,该函数必须将任务从事件控制块的等待列表中删除
95                 //并在新的优先级下将事件插入到等待队列中。任务也可能正在等待延时时间到,或是被
96                 //挂起。
97                 if ((pevent->OSEventTbl[ptcb->OSTCBY] & ~ptcb->OSTCBBitX) == 0) {
98                     pevent->OSEventGrp &= ~ptcb->OSTCBBitY;
99                 }
100                 pevent->OSEventGrp |= bity; //将任务插入到等待列表中
101                 pevent->OSEventTbl[y] |= bitx;
102             }
103 #endif
104         }
105         OSTCBPrioTbl[newprio] = ptcb; //将任务的OS_TCB的指针存到新任务OSTCBPrioTbl[]
106         ptcb->OSTCBPrio = newprio; //设定新的任务优先级,并保存原有计算值
107         ptcb->OSTCBY = y; //高3位计算值
108         ptcb->OSTCBX = x; //低3位计算值
109         ptcb->OSTCBBitY = bity;
110         ptcb->OSTCBBitX = bitx;
111         OS_EXIT_CRITICAL(); //打开中断
112         OS_Sched(); //任务调度,最高任务优先级运行
113         return (OS_NO_ERR); //任务优先级成功改变
114     } else { //否则
115         OSTCBPrioTbl[newprio] = (OS_TCB *)0; //新任务就绪态去除它(为0不存在),
116         OS_EXIT_CRITICAL(); //打开中断
117         return (OS_PRIO_ERR); //返回(参数中的任务原先优先级不存在)
118     }
119 }
120 }
121 #endif
122 /*$PAGE*/
123 /*
124 ****
125 *
126 * 建立一个新任务(CREATE A TASK)
127 * 描述: 建立一个新任务。任务的建立可以在多任务环境启动之前,也可以在正在运行的任务中建立。中断
128 * 处理程序中不能建立任务。一个任务必须为无限循环结构(如下所示),且不能有返回点。
129 * OSTaskCreate()是为与先前的μC/OS版本保持兼容,新增的特性在OSTaskCreateExt()函数中。
130 * 无论用户程序中是否产生中断,在初始化任务堆栈时,堆栈的结构必须与CPU中断后寄存器入栈的
131 * 顺序结构相同。详细说明请参考所用处理器的手册。
132 *
133 * 参数: task 是指向任务代码的指针。
134 *
135 * pdata 指向一个数据结构,该结构用来在建立任务时向任务传递参数。下例中说明uC/OS中的任
136 * 务结构以及如何传递参数pdata:
137 * void Task (void *pdata)
138 * {
139 *     ... // 对参数'pdata'进行操作
140 *     for (;;) { // 任务函数体。
141 *         ...
142 *         ...
143 *         // 在任务体中必须调用如下函数之一:
144 *         // OSMboxPend() 用于任务等待消息,消息通过中断或另外的任务发送给需要的任务
145 *         // OSFlgPend() 用于任务等待事件标志中的事件标志
146 *         // OSMutexPend() 任务需要独占资源
147 *         // OSQPend() 用于任务等待消息
148 *         // OSSemPend() 用于任务试图取得共享资源的使用权,任务需要与其它任务或中断
149 *         // 同步及任务需要等待特定事件的发生场合
150 *         // OSTimeDly() 任务延时若干时钟节拍
151 *         // OSTimeDlyHMSM() 任务延时若干时间
152 *         // OSTaskSuspend() 挂起任务本身

```

```

153 *      //      OSTaskDel()      删除任务本身
154 *      ...
155 *      ...
156 *      }
157 *      ptos  为指向任务堆栈栈顶的指针。任务堆栈用来保存局部变量, 函数参数, 返回地址以及任务被
158 *            中断时的CPU寄存器内容。任务堆栈的大小决定于任务的需要及预计的中断嵌套层数。计算
159 *            堆栈的大小, 需要知道任务的局部变量所占的空间, 可能产生嵌套调用的函数, 及中断嵌套
160 *            所需空间。如果初始化常量OS_STK_GROWTH设为1, 堆栈被设为从内存高地址向低地址增长,
161 *            此时ptos应该指向任务堆栈空间的最高地址。反之, 如果OS_STK_GROWTH设为0, 堆栈将从内存
162 *            的低地址向高地址增长。
163 *
164 *      prio  为任务的优先级。每个任务必须有一个唯一的优先级作为标识。数字越小, 优先级越高。
165 *
166 * 返回: OS_NO_ERR      函数调用成功;
167 *       OS_PRIO_EXIT   具有该优先级的任务已经存在;
168 *       OS_PRIO_INVALID 参数指定的优先级大于OS_LOWEST_PRIO; (i.e. >= OS_LOWEST_PRIO)
169 *       OS_NO_MORE_TCB 系统中没有OS_TCB可以分配给任务了。
170 *
171 * 注意: 1、任务堆栈必须声明为OS_STK类型。
172 *       2、在任务中必须调用uC/OS提供的下述过程之一: 延时等待、任务挂起、等待事件发生(等待信
173 *       号量, 消息邮箱、消息队列), 以使其他任务得到CPU。
174 *       3、用户程序中不能使用优先级0, 1, 2, 3, 以及OS_LOWEST_PRIO-3, OS_LOWEST_PRIO-2,
175 *       OS_LOWEST_PRIO-1, OS_LOWEST_PRIO。这些优先级uC/OS系统保留, 其余的56个优先级提供给
176 *       应用程序。
177 *****
178 */
179
180 #if OS_TASK_CREATE_EN > 0      //允许生成OSTaskCreate()函数
181 INT8U OSTaskCreate(void (*task)(void *pd), void *pdata, OS_STK *ptos, INT8U prio)
182 {      //建立任务(任务代码指针、传递参数指针、分配任务堆栈栈顶指针、任务优先级)
183 #if OS_CRITICAL_METHOD == 3      //中断函数被设定为模式3
184     OS_CPU_SR cpu_sr;
185 #endif
186     OS_STK *psp;      //初始化任务堆栈指针变量, 返回新的栈顶指针
187     INT8U err;      //定义(获得并定义初始化任务控制块)是否成功
188
189     #if OS_ARG_CHK_EN > 0      //所有参数必须在指定的参数内
190         if (prio > OS_LOWEST_PRIO) {      //检查任务优先级是否合法
191             return (OS_PRIO_INVALID);      //参数指定的优先级大于OS_LOWEST_PRIO
192         }
193     #endif
194     OS_ENTER_CRITICAL();      //关闭中断
195     if (OSTCBPrioTbl[prio] == (OS_TCB *)0) {      //确认优先级未被使用, 即就绪态为0
196         OSTCBPrioTbl[prio] = (OS_TCB *)1;      //保留这个优先级, 将就绪态设为0
197
198         OS_EXIT_CRITICAL();      //打开中断
199         psp = (OS_STK *)OSTaskStkInit(task, pdata, ptos, 0);      //初始化任务堆栈
200         err = OS_TCBInit(prio, psp, (OS_STK *)0, 0, 0, (void *)0, 0);      //获得并初始化任务控制块
201         if (err == OS_NO_ERR) {      //任务控制初始化成功
202             OS_ENTER_CRITICAL();      //关闭中断
203             OSTaskCtr++;      //任务计数器加1
204             OS_EXIT_CRITICAL();      //打开中断
205             if (OSRunning == TRUE) {      //检查是否有(某个)任务在运行
206                 OS_Sched();      //任务调度, 最高任务优先级运行
207             }
208         } else {      //否则, 任务初始化失败
209             OS_ENTER_CRITICAL();      //关闭中断
210             OSTCBPrioTbl[prio] = (OS_TCB *)0;      //放弃任务, 设此任务就绪态为0
211             OS_EXIT_CRITICAL();      //打开中断
212         }
213         return (err);      //返回(获得并定义初始化任务控制块是否成功)
214     }
215     OS_EXIT_CRITICAL();      //打开中断
216     return (OS_PRIO_EXIST);      //返回(具有该优先级的任务已经存在)
217 }
218 #endif
219 /*$PAGE*/
220 /*
221 *****
222 *      CREATE A TASK (Extended Version)
223 *
224 * 描述: 建立一个新任务。与OSTaskCreate()不同的是, OSTaskCreateExt()允许用户设置更多的细节
225 *       内容。任务的建立可以在多任务环境启动之前, 也可以在正在运行的任务中建立, 但中断处理
226 *       程序中不能建立新任务。一个任务必须为无限循环结构(如下所示), 且不能有返回点。
227 *
228 * 参数: task      是指向任务代码的指针。

```

```

229 *
230 *      pdata      Pdata指针指向一个数据结构,该结构用来在建立任务时向任务传递参数。下例中说明uC/OS中的任务代码结构以及如何传递参数pdata: (如果在程序中不使用参数pdata,
231 *
232 *                  为了避免在编译中出现“参数未使用”的警告信息,可以写一句pdata= pdata; )
233 *                  void Task (void *pdata)
234 *                  {
235 *                      ...                               //对参数pdata进行操作,例如pdata= pdata
236 *                      for (;;) {                         // 任务函数体,总是为无限循环结构
237 *                          ...
238 *                          ...
239 *                      // 任务中必须调用如下的函数:
240 *                      //      OSMboxPend()      用于任务等待消息,消息通过中断或另外的任务发送给需要的任务
241 *                      //      OSFlgPend()       用于任务等待事件标志中的事件标志
242 *                      //      OSMutexPend()     任务需要独占资源
243 *                      //      OSQPend()        用于任务等待消息
244 *                      //      OSSemPend()      用于任务试图取得共享资源的使用权,任务需要与其它任务或中断
245 *                      //                      同步及任务需要等待特定事件的发生场合
246 *                      //      OSTimeDly()      任务延时若干时钟节拍
247 *                      //      OSTimeDlyHMSM()  任务延时若干时间
248 *                      //      OSTaskSuspend()   挂起任务本身
249 *                      //      OSTaskDel()      删除任务本身
250 *                      ...
251 *                      ...
252 *                      }
253 *
254 *      ptos      为指向任务堆栈栈顶的指针。任务堆栈用来保存局部变量,函数参数,返回地址以及中
255 *
256 *                  断时的CPU寄存器内容。任务堆栈的大小决定于任务的需要及预计的中断嵌套层数。计
257 *
258 *                  算堆栈的大小,需要知道任务的局部变量所占的空间,可能产生嵌套调用的函数,及
259 *
260 *                  中断嵌套所需空间。如果初始化常量OS_STK_GROWTH设为1,堆栈被设为向低端增长
261 *
262 *                  (从内存高地址向低地址增长)。此时ptos应该指向任务堆栈空间的最高地址。反之,
263 *
264 *                  如果OS_STK_GROWTH设为0,堆栈将从低地址向高地址增长。
265 *
266 *      prio      任务的优先级。每个任务必须有一个唯一的优先级作为标识。数字越小,优先级越高。
267 *
268 *      id        是任务的标识,目前这个参数没有实际的用途,但保留在OSTaskCreateExt()中供今后
269 *
270 *                  扩展,应用程序中可设置id与优先级相同。(0..65535)
271 *
272 *      pbot      为指向堆栈底端的指针。如果初始化常量OS_STK_GROWTH设为1,堆栈被设为从内存高
273 *
274 *                  地址向低地址增长。此时pbot应该指向任务堆栈空间的最低地址。反之,如果
275 *
276 *                  OS_STK_GROWTH设为0,堆栈将从低地址向高地址增长。pbot应该指向堆栈空间的最高
277 *
278 *                  地址。参数pbot用于堆栈检测函数OSTaskStkChk()。
279 *
280 *      stk_size  指定任务堆栈的大小。其单位由OS_STK定义:当OS_STK的类型定义为INT8U、INT16U、
281 *
282 *                  INT32U的时候,stk_size的单位为分别为字节(8位)、字(16位)和双字(32位)。
283 *
284 *      pext      是一个用户定义数据结构的指针,可作为TCB的扩展。例如,当任务切换时,用户定义
285 *
286 *                  的数据结构中可存放浮点寄存器的数值,任务运行时间,任务切入次数等等信息。
287 *
288 *      opt       存放与任务相关的操作信息。opt的低8位由uC/OS保留,用户不能使用。用户可以使用
289 *
290 *                  opt的高8位。每一种操作由opt中的一位或几位指定,当相应的位被置位时,表示选择
291 *
292 *                  某种操作。当前的μC/OS版本支持下列操作:
293 *
294 *                  OS_TASK_OPT_STK_CHK: 决定是否进行任务堆栈检查;
295 *
296 *                  OS_TASK_OPT_STK_CLR: 决定是否清空堆栈;
297 *
298 *                  OS_TASK_OPT_SAVE_FP: 决定是否保存浮点寄存器的数值。此项操作仅当处理器有浮
299 *
300 *                      点硬件时有效。保存操作由硬件相关的代码完成。
301 *
302 *      返回: OS_NO_ERR: 函数调用成功;
303 *
304 *      OS_PRIO_EXIST: 具有该优先级的任务已经存在;
305 *
306 *      OS_PRIO_INVALID: 参数指定的优先级大于OS_LOWEST_PRIO;
307 *
308 *      OS_NO_MORE_TCB: 系统中没有OS_TCB可以分配给任务了。
309 *
310 *      注意: 1、任务堆栈必须声明为OS_STK类型;
311 *
312 *            2、在任务中必须进行uC/OS提供的下述过程之一: 延时等待、任务挂起、等待事件发生(等待
313 *
314 *                信号量,消息邮箱、消息队列),以使其他任务得到CPU;
315 *
316 *            3、用户程序中不能使用优先级0,1,2,3,以及OS_LOWEST_PRIO-3, OS_LOWEST_PRIO-2,
317 *
318 *                OS_LOWEST_PRIO-1, OS_LOWEST_PRIO。这些优先级μC/OS系统保留,其余56个优先级提供给
319 *
320 *                应用程序。
321 *
322 * *****
323 */
324 /*$PAGE*/
325
326 #if OS_TASK_CREATE_EXT_EN > 0                               //允许生成OSTaskCreateExt()函数
327 INT8U OSTaskCreateExt (void (*task)(void *pd),             //建立扩展任务(任务代码指针
328                        void *pdata,                         //传递参数指针
329                        OS_STK *ptos,                       //分配任务堆栈栈顶指针
330                        INT8U prio,                         //分配任务优先级

```



```

305         INT16U    id,           // (未来的) 优先级标识 (与优先级相同)
306         OS_STK    *pbos,        // 分配任务堆栈栈底指针
307         INT32U    stk_size,     // 指定堆栈的容量 (检验用)
308         void      *pext,        // 指向用户附加的数据域的指针
309         INT16U    opt)          // 建立任务设定选项)
310 {
311     #if OS_CRITICAL_METHOD == 3                // 中断函数被设定为模式3
312         OS_CPU_SR  cpu_sr;
313     #endif
314     OS_STK    *psp;                // 初始化任务堆栈指针变量, 返回新的栈顶指针
315     INT8U     err;                // 定义 (获得并定义初始化任务控制块) 是否成功
316
317     #if OS_ARG_CHK_EN > 0                // 所有参数必须在指定的参数内
318         if (prio > OS_LOWEST_PRIO) {        // 检查任务优先级是否合法
319             return (OS_PRIO_INVALID);        // 参数指定的优先级大于OS_LOWEST_PRIO
320         }
321     #endif
322     OS_ENTER_CRITICAL();                // 关闭中断
323     if (OSTCBPrioTbl[prio] == (OS_TCB *)0) { // 确认优先级未被使用, 即就绪态为0
324         OSTCBPrioTbl[prio] = (OS_TCB *)1;    // 保留这个优先级, 将就绪态设为0
325
326         OS_EXIT_CRITICAL();                // 打开中断
327     }
328     // 以下两为1堆栈才能清0
329     if ((opt & OS_TASK_OPT_STK_CHK) != 0x0000) || // 检验任务堆栈, CHK=1
330         ((opt & OS_TASK_OPT_STK_CLR) != 0x0000)) { // 任务建立时是否清0, CLR=1
331         #if OS_STK_GROWTH == 1                // 堆栈生长方向
332             (void)memset(pbos, 0, stk_size * sizeof(OS_STK)); // 从下向上递增
333         #else
334             (void)memset(ptos, 0, stk_size * sizeof(OS_STK)); // 从上向下递减
335         #endif
336     }
337     psp = (OS_STK *)OSTaskStkInit(task, pdata, ptos, opt); // 初始化任务堆栈
338     err = OS_TCBInit(prio, psp, pbos, id, stk_size, pext, opt); // 获得并初始化任务控制块
339     if (err == OS_NO_ERR) {                    // 任务控制初始化成功
340         OS_ENTER_CRITICAL();                // 关闭中断
341         OSTaskCtr++;                        // 任务计数器加1
342         OS_EXIT_CRITICAL();                // 打开中断
343         if (OSRunning == TRUE) {            // 检查是否有 (某个) 任务在运行
344             OS_Sched();                    // 任务调度, 最高任务优先级运行
345         }
346     } else {                                // 否则, 任务初始化失败
347         OS_ENTER_CRITICAL();                // 关闭中断
348         OSTCBPrioTbl[prio] = (OS_TCB *)0;    // 放弃任务, 设此任务就绪态为0
349         OS_EXIT_CRITICAL();                // 打开中断
350     }
351     return (err);                            // 返回 (获得并定义初始化任务控制块是否成功)
352 }
353 OS_EXIT_CRITICAL();                // 打开中断
354 return (OS_PRIO_EXIST);            // 具有该优先级的任务已经存在
355 }
356 #endif
357 /*$PAGE*/
358 /*
359 *****
360 *                                     删除一个指定优先级的任务 (DELETE A TASK)
361 *
362 * 描述: 删除一个指定优先级的任务。任务可以传递自己的优先级给OSTaskDel(), 从而删除自身。如果任
363 * 务不知道自己的优先级, 还可以传递参数OS_PRIO_SELF。被删除的任务将回到休眠状态。任务被删
364 * 除后可以用函数OSTaskCreate() 或OSTaskCreateExt() 重新建立。
365 *
366 * 参数: prio    为指定要删除任务的优先级, 也可以用参数OS_PRIO_SELF代替, 此时, 下一个优先级最高
367 * 的就绪任务将开始运行。
368 *
369 * 返回: OS_NO_ERR: 函数调用成功;
370 *       OS_TASK_DEL_IDLE: 错误操作, 试图删除空闲任务 (Idle task);
371 *       OS_TASK_DEL_ERR: 错误操作, 指定要删除的任务不存在;
372 *       OS_PRIO_INVALID: 参数指定的优先级大于OS_LOWEST_PRIO;
373 *       OS_TASK_DEL_ISR: 错误操作, 试图在中断处理程序中删除任务。
374 *
375 *
376 * 注意: 1、OSTaskDel() 将判断用户是否试图删除uC/OS中的空闲任务 (Idle task);
377 *       2、在删除占用系统资源的任务时要小心, 此时, 为安全起见可以用另一个函数OSTaskDelReq()。
378 *
379 *****
380 */

```

```

381 /*$PAGE*/
382 #if OS_TASK_DEL_EN > 0 //允许生成 OSTaskDel() 函数
383 INT8U OSTaskDel (INT8U prio) //删除任务(任务的优先级)
384 {
385     #if OS_CRITICAL_METHOD == 3 //中断函数被设定为模式3
386         OS_CPU_SR cpu_sr;
387     #endif
388
389     #if OS_EVENT_EN > 0 //消息事件是否 > 0
390         OS_EVENT *pevent; //定义事件指针
391     #endif
392     //版本是否 > 2.51 并且 允许产生事件标志相关代码 并且应用中最多事件标志组的数目 > 0
393     #if (OS_VERSION >= 251) && (OS_FLAG_EN > 0) && (OS_MAX_FLAGS > 0)
394         OS_FLAG_NODE *pnode; //定义标志节点
395     #endif
396     OS_TCB *ptcb; //定义消息事件的任务控制块指针
397
398
399
400     if (OSIntNesting > 0) { //当前中断嵌套 > 0时, 表示还有中断程序运行
401         return (OS_TASK_DEL_ISR); //错误操作, 试图在中断处理程序中删除任务
402     }
403     #if OS_ARG_CHK_EN > 0 //所有参数必须在指定的参数内
404     if (prio == OS_IDLE_PRIO) { //检查任务优先级是否是空闲任务
405         return (OS_TASK_DEL_IDLE); //错误操作, 试图删除空闲任务 (Idle task)
406     } //当任务 >= 最低优先级 并且 任务不是本身
407     if (prio >= OS_LOWEST_PRIO && prio != OS_PRIO_SELF) { //检查任务是否是合法的
408         return (OS_PRIO_INVALID); //参数指定的优先级大于OS_LOWEST_PRIO
409     }
410     #endif
411     OS_ENTER_CRITICAL(); //关闭中断
412     if (prio == OS_PRIO_SELF) { //检查的删除者是否是任务(优先级)本身
413         prio = OSTCBBCur->OSTCBPrio; //正在运行的优先级(任务本身的优先级)
414     } //删除的任务必须存在
415     if ((ptcb = OSTCBPrioTbl[prio]) != (OS_TCB *)0) { //调用这个任务的优先级的就绪值
416         if ((OSRdyTbl[ptcb->OSTCBY] & ~ptcb->OSTCBBitX) == 0x00) { //当就绪=1(即就绪状态)
417             OSRdyGrp &= ~ptcb->OSTCBBitY; //该任务处于就绪值, 从就绪表中去除
418         }
419     }
420     #if OS_EVENT_EN > 0 //消息事件是否 > 0
421     pevent = ptcb->OSTCBEventPtr; //拾取该任务的事件控制指针
422     if (pevent != (OS_EVENT *)0) {
423         //指向事件控制块的指针是否为Null
424         if ((pevent->OSEventTbl[ptcb->OSTCBY] & ~ptcb->OSTCBBitX) == 0) {
425             //事件(消息)在等待表中将自己所处的表中删除
426             pevent->OSEventGrp &= ~ptcb->OSTCBBitY;
427         }
428     }
429     #endif
430     //版本是否 > 2.51 并且 允许产生事件标志相关代码 并且应用中最多事件标志组的数目 > 0
431     #if (OS_VERSION >= 251) && (OS_FLAG_EN > 0) && (OS_MAX_FLAGS > 0)
432     pnode = ptcb->OSTCBFlagNode; //标志节点 = 指向事件标志节点的指针
433     if (pnode != (OS_FLAG_NODE *)0) { //如果任务处于事件标志的等待表中
434         OS_FlagUnlink(pnode); //从此表中删除
435     }
436     #endif
437     ptcb->OSTCBDly = 0; //将任务时钟节拍清0, 确保重新开中断, 中断程序不使该任务就绪
438     ptcb->OSTCBStat = OS_STAT_RDY; //将任务的状态字处于完毕状态
439     if (OSLockNesting < 255) { //如果锁定嵌套计数器 < 255,
440         OSLockNesting++; //锁定嵌套计数器加1, 使这个任务处于休眠状态
441     }
442     OS_EXIT_CRITICAL(); //打开中断
443     OS_Dummy(); //增加一个空指令
444     OS_ENTER_CRITICAL(); //关闭中断
445     if (OSLockNesting > 0) { //((可以继续执行删除任务的操作了)
446         OSLockNesting--; //重新关闭中断后, 可以通过锁定嵌套计数器减1, 重新允许任务调度
447     }
448     OSTaskDelHook(ptcb); //可在钩子程序中加入自定程序
449     OSTaskCtr--; //任务计数器减1, ucos管理的任务减少一个
450     OSTCBPrioTbl[prio] = (OS_TCB *)0; //将被删除的任务控制块OS_TCB指针=NULL,
451     //从任务优先级中把OS_TCB删除
452     if (ptcb->OSTCBPrev == (OS_TCB *)0) { //任务块双向链接表的前链接是否Null
453         ptcb->OSTCBNext->OSTCBPrev = (OS_TCB *)0; //删除该任务的任务控制块OS_TCB
454         OSTCBLst = ptcb->OSTCBNext; //链接表指向后一个链接
455     } else { //否则
456         ptcb->OSTCBPrev->OSTCBNext = ptcb->OSTCBNext; // ?
457         ptcb->OSTCBNext->OSTCBPrev = ptcb->OSTCBPrev; // ?

```

```

457     }                                     //OSTCBPrev:用于任务块双向链接表的前链接
458                                     //OSTCBNext:用于任务块双向链接表的后链接
459                                     //OSTCBFreeList:空任务控制块指针
460     ptcb->OSTCBNext = OSTCBFreeList;     //被删除的任务控制块OS_TCB被退回到空闲OS_TCB表中
461     OSTCBFreeList = ptcb;               //以供建立其它任务使用
462     OS_EXIT_CRITICAL();                 //打开中断
463     OS_Sched();                         //任务调度,最高任务优先级运行
464     return (OS_NO_ERR);                 //函数调用成功
465 }
466 OS_EXIT_CRITICAL();                   //打开中断
467 return (OS_TASK_DEL_ERR);             //错误操作,指定要删除的任务不存在
468 }
469 #endif
470 /*$PAGE*/
471 /*
472 ****
473 *      请求一个任务删除其它任务或自身(REQUEST THAT A TASK DELETE ITSELF)
474 *
475 * 描述: 请求一个任务删除自身。通常OSTaskDelReq()用于删除一个占有系统资源的任务(例如任务建立了信号量)对于此类任务,在删除任务之前应当先释放任务占用的系统资源。
476 *      具体的做法是:在需要被删除的任务中调用OSTaskDelReq()检测是否有其他任务的删除请求,如果有,则释放自身占用的资源,然后调用OSTaskDel()删除自身。例如,假设任务5要删除任务10,而任务10占有系统资源,此时任务5不能直接调用OSTaskDel(10)删除任务10,而应该调用OSTaskDelReq(10)向任务10发送删除请求。在任务10中调用OSTaskDelReq(OS_PRIO_SELF),并检测返回值。如果返回OS_TASK_DEL_REQ,则表明有来自其他任务的删除请求,此时任务10应该先释放资源,然后调用OSTaskDel(OS_PRIO_SELF)删除自己。任务5可以循环调用OSTaskDelReq(10)并检测返回值,如果返回OS_TASK_NOT_EXIST,表明任务10已经成功删除。
477 *      void Task(void *data)
478 *      {
479 *          ...
480 *          ...
481 *          while (1) {
482 *              OSTimeDly(1);
483 *              if (OSTaskDelReq(OS_PRIO_SELF) == OS_TASK_DEL_REQ)
484 *              {
485 *                  释放任务占用的系统资源;
486 *                  释放动态分配的内存;
487 *                  OSTaskDel(OS_PRIO_SELF);
488 *              }
489 *          }
490 *      }
491 * 参数: prio 为要求删除任务的优先级。如果参数为OS_PRIO_SELF,则表示调用函数的任务正在查询是否有来自其他任务的删除请求。
492 * 返回: OS_NO_ERR: 删除请求已经被任务记录;
493 *       OS_TASK_NOT_EXIST: 指定的任务不存在,发送删除请求的任务可以等待此返回值,看删除是否成功;
494 *       OS_TASK_DEL_IDLE: 错误操作,试图删除空闲任务(Idle task);
495 *       OS_PRIO_INVALID: 参数指定的优先级大于OS_LOWEST_PRIO或没有设定OS_PRIO_SELF的值;
496 *       OS_TASK_DEL_REQ: 当前任务收到来自其他任务的删除请求;
497 * 注意: OSTaskDelReq()将判断用户是否试图删除uC/OS中的空闲任务(Idle task)。
498 ****
499 */
500 /*$PAGE*/
501 #if OS_TASK_DEL_EN > 0                 //允许生成 OSTaskDel() 函数
502 INT8U OSTaskDelReq (INT8U prio)       //请求一个任务删除其它任务或自身?(任务的优先级)
503 {
504     #if OS_CRITICAL_METHOD == 3       //中断函数被设定为模式3
505         OS_CPU_SR cpu_sr;
506     #endif
507     BOOLEAN stat;                     //定义(布尔)任务标志返回值
508     INT8U err;                        //定义函数成功删除或其它任务正在申请删除
509     OS_TCB *ptcb;                     //定义消息事件的任务控制块指针
510
511     #if OS_ARG_CHK_EN > 0              //所有参数必须在指定的参数内
512     if (prio == OS_IDLE_PRIO) {       //检查任务优先级是否是空闲任务
513         return (OS_TASK_DEL_IDLE);    //错误操作,试图删除空闲任务(Idle task)
514     }                                 //当任务 >= 最低优先级 并且 任务不是本身
515     if (prio >= OS_LOWEST_PRIO && prio != OS_PRIO_SELF) { //检查任务是否是合法的
516         return (OS_PRIO_INVALID);    //参数指定的优先级大于OS_LOWEST_PRIO
517     }
518     #endif
519     #if OS_CRITICAL_METHOD == 3
520     if (prio == OS_PRIO_SELF) {       //如果删除者是任务本身
521         OS_ENTER_CRITICAL();          //关闭中断
522         stat = OSTCBBur->OSTCBBurDelReq; //stat: 存储在任务控制块中的标志值
523         OS_EXIT_CRITICAL();           //打开中断

```

```

533     return (stat);                //返回任务标志值
534 }
535 OS_ENTER_CRITICAL();              //关闭中断
536 if ((ptcb = OSTCBPrioTbl[prio]) != (OS_TCB *)0) { //调用这个任务的优先级的就绪值
537     ptcb->OSTCBDelReq = OS_TASK_DEL_REQ;          //当前任务收到来自其他任务的删除请求
538     err               = OS_NO_ERR;                //删除请求已经被任务记录
539 } else {                                          //否则
540     err               = OS_TASK_NOT_EXIST;        //该任务可能已经删除了
541 }
542 //指定的任务不存在，发送删除请求的任务可以等待此返回值，看删除是否成功
543
544 OS_EXIT_CRITICAL();                //关闭中断
545 return (err);                      //返回删除情况标志
546 }
547 #endif
548 /*$PAGE*/
549 /*
550 ****
551 *           唤醒一个用OSTaskSuspend() 函数挂起的任务 (RESUME A SUSPENDED TASK)
552 *
553 * 描述：唤醒一个用OSTaskSuspend() 函数挂起的任务。OSTaskResume() 也是唯一能“解挂”挂起任务的函数。
554 *
555 * 参数：prio    指定要唤醒任务的优先级。
556 *
557 * 返回：OS_NO_ERR：    函数调用成功；
558 *       OS_TASK_RESUME_PRIO： 要唤醒的任务不存在；
559 *       OS_TASK_NOT_SUSPENDED： 要唤醒的任务不在挂起状态；
560 *       OS_PRIO_INVALID：    参数指定的优先级大于或等于OS_LOWEST_PRIO。
561 ****
562 */
563
564 #if OS_TASK_SUSPEND_EN > 0          //允许生成 OSTaskDel() 函数
565 INT8U OSTaskResume (INT8U prio)     //唤醒一个用OSTaskSuspend() 函数挂起的任务 (任务的优先级)
566 {
567     #if OS_CRITICAL_METHOD == 3    //中断函数被设定为模式3
568         OS_CPU_SR cpu_sr;
569     #endif
570     OS_TCB *ptcb;                  //定义消息事件的任务控制块指针
571
572
573     #if OS_ARG_CHK_EN > 0          //所有参数必须在指定的参数内
574         if (prio >= OS_LOWEST_PRIO) { //当任务 >= 最低优先级
575             return (OS_PRIO_INVALID); //返回(要唤醒的任务不存在)
576         }
577     #endif
578     OS_ENTER_CRITICAL();           //关闭中断
579     if ((ptcb = OSTCBPrioTbl[prio]) == (OS_TCB *)0) { //调用这个任务的优先级的就绪值
580         OS_EXIT_CRITICAL();         //打开中断
581         return (OS_TASK_RESUME_PRIO); //返回(要唤醒的任务不存在)
582     } //任务控制块状态字相与OS_STAT_SUSPEND为1，任务存在
583     if ((ptcb->OSTCBStat & OS_STAT_SUSPEND) != 0x00) { //要恢复的任务必须是存在的
584         //通过清除OSTCBStat域中的OS_STAT_SUSPEND未而取消挂起
585         //要使任务处于就绪状态，OSTCBDly必须是0，这是因为在OSTCBStat中，没有任何标志表明任务正在
586         //等待延时事件到
587         if (((ptcb->OSTCBStat & ~OS_STAT_SUSPEND) == OS_STAT_RDY) &&
588             (ptcb->OSTCBDly == 0)) {
589             //只有以上两个条件满足，任务才能处于就绪态
590             OSRdyGrp |= ptcb->OSTCBBitY; //保存任务就绪标准0-7到OSRdyGrp
591             OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX; //保存任务优先级别0-7到OSRdyTbl[]
592             OS_EXIT_CRITICAL(); //打开中断
593             OS_Sched();         //任务调度，最高任务优先级运行
594         } else {                //否则
595             OS_EXIT_CRITICAL(); //打开中断
596         }
597         return (OS_NO_ERR);      //返回(函数调用成功)
598     }
599     OS_EXIT_CRITICAL();         //打开中断
600     return (OS_TASK_NOT_SUSPENDED); //返回(要唤醒的任务不在挂起状态)
601 }
602 #endif
603 /*$PAGE*/
604 /*
605 ****
606 *           检查任务堆栈状态，计算指定任务堆栈中的未用空间和已用空间 (STACK CHECKING)
607 *
608 * 描述：检查任务堆栈状态，计算指定任务堆栈中的未用空间和已用空间。使用OSTaskStkChk() 函数要求

```



```

609 *      所检查的任务是被OSTaskCreateExt()函数建立的,且opt参数中OS_TASK_OPT_STK_CHK操作项打开。
610 *      //计算堆栈未用空间的方法是从堆栈底端向顶端逐个字节比较,检查堆栈中0的个数,直到一个非0的
611 *      数值出现.这种方法的前提是堆栈建立时已经全部清零.要实现清零操作,需要在任务建立初始化
612 *      堆栈时设置OS_TASK_OPT_STK_CLR为1.如果应用程序在初始化时已经将全部RAM清零,且不进行任
613 *      务删除操作,也可以设置OS_TASK_OPT_STK_CLR为0,这将加快OSTaskCreateExt()函数的执行速度。
614 *
615 * 参数: prio 为指定要获取堆栈信息的任务优先级,也可以指定参数OS_PRIO_SELF,获取调用任务本身
616 *      信息。
617 *
618 *      pdata 指向一个类型为OS_STK_DATA的数据结构,其中包含如下信息:
619 *      INT32U OSFree;          // 堆栈中未使用的字节数
620 *      INT32U OSUsed;          // 堆栈中已使用的字节数
621 *
622 * 返回: OS_NO_ERR:          函数调用成功;
623 *      OS_PRIO_INVALID:    参数指定的优先级大于OS_LOWEST_PRIO,或未指定OS_PRIO_SELF;
624 *      OS_TASK_NOT_EXIST: 指定的任务不存在;
625 *      OS_TASK_OPT_ERR:    任务用OSTaskCreateExt()函数建立的时候没有指定OS_TASK_OPT_STK_CHK
626 *      操作,或者任务是用OSTaskCreate()函数建立的。
627 *
628 * 注意: 1、函数的执行时间是由任务堆栈的大小决定的,事先不可预料;
629 *      2、在应用程序中可以把OS_STK_DATA结构中的数据项OSFree和OSUsed相加,可得到堆栈的大小;
630 *      3、虽然原则上该函数可以在中断程序中调用,但由于该函数可能执行很长时间,所以实际中不提倡这种做法。
631 *
632 *****
633 */
634 #if OS_TASK_CREATE_EXT_EN > 0          //允许生成 OSTaskStkChk() 函数
635 INT8U OSTaskStkChk (INT8U prio, OS_STK_DATA *pdata)
636 {
637     #if OS_CRITICAL_METHOD == 3          //中断函数被设定为模式3
638         OS_CPU_SR cpu_sr;
639     #endif
640     OS_TCB *ptcb;                        //定义消息事件的任务控制块指针
641     OS_STK *pchk;                        //定义指向当前任务堆栈栈底的指针
642     INT32U free;                         //定义任务堆栈实际空闲字节数
643     INT32U size;                         //定义存有栈中可容纳的指针数目
644
645
646     #if OS_ARG_CHK_EN > 0                //所有参数必须在指定的参数内
647         //当任务 >= 最低优先级 并且 任务不是本身
648         if (prio > OS_LOWEST_PRIO && prio != OS_PRIO_SELF) { //检查任务是否是合法的
649             return (OS_PRIO_INVALID); //参数指定的优先级大于OS_LOWEST_PRIO,或未指定OS_PRIO_SELF
650         }
651     #endif
652     pdata->OSFree = 0;                   //未使用的字节清0
653     pdata->OSUsed = 0;                   //已使用的字节清0
654     OS_ENTER_CRITICAL();                 //关闭中断
655     if (prio == OS_PRIO_SELF) {          //如果检查者是任务本身
656         prio = OSTCBCur->OSTCBPrio;     //指向正在运行(优先级)任务控制块的指针
657     }
658     ptcb = OSTCBPrioTbl[prio];           //调用这个优先级别任务的就绪值
659     if (ptcb == (OS_TCB *)0) {           //当检验OSTCBPrioTbl[]里是非0值,确定任务存在
660         OS_EXIT_CRITICAL();              //打开中断
661         return (OS_TASK_NOT_EXIST);      //指定的任务不存在
662     }
663     //要执行堆栈检验,必须用OSTaskCreateExt()函数建立任务,并且已经传递了选项OS_TASK_OPT_STK_CHK,
664     //而对OSTaskCreate()函数建立的任务,那么会对OPT(传递给OS_TCBInit)为0,而使检验失败
665     if ((ptcb->OSTCBOpt & OS_TASK_OPT_STK_CHK) == 0) { //如果检验OPT=0
666         OS_EXIT_CRITICAL();              //打开中断
667         return (OS_TASK_OPT_ERR);        //返回检验失败
668     }
669     free = 0;                            //从堆栈栈底统计堆栈空闲空间,直到遇到一个存储非0值
670     size = ptcb->OSTCBStkSize;            //堆栈中未使用的字节数free = 0
671     pchk = ptcb->OSTCBStkBottom;          //size = 存有栈中可容纳的指针数目
672     OS_EXIT_CRITICAL();                  //pchk = 指向当前任务堆栈栈底的指针
673     #if OS_STK_GROWTH == 1                //打开中断
674         while (*pchk++ == (OS_STK)0) {   //查看堆栈增长方向,为1递增
675             free++;                       //当前任务堆栈栈顶的指针加1(内容)是否等于0
676         }                                //定义堆栈中未使用的字节数加1
677     #else
678         while (*pchk-- == (OS_STK)0) {   //否则,向下递减
679             free++;                       //当前任务堆栈栈底的指针减1(内容)是否等于0
680         }                                //定义堆栈中未使用的字节数加1
681     #endif
682     pdata->OSFree = free * sizeof(OS_STK); //任务堆栈实际空闲字节数
683     pdata->OSUsed = (size - free) * sizeof(OS_STK); //任务堆栈已被占用的字节数
684     return (OS_NO_ERR);                  //函数调用成功

```



```

685 }
686 #endif
687 /*$PAGE*/
688 /*
689 ****
690 * 无条件挂起一个任务 (SUSPEND A TASK)
691 *
692 * 描述: 无条件挂起一个任务. 调用此函数的任务也可以传递参数OS_PRI0_SELF, 挂起调用任务本身.
693 * 当前任务挂起后, 只有其他任务才能唤醒. 任务挂起后, 系统会重新进行任务调度, 运行下一个
694 * 优先级最高的就绪任务. 唤醒挂起任务需要调用函数OSTaskResume().
695 * //任务的挂起是可以叠加到其他操作上的. 例如, 任务被挂起时正在进行延时操作, 那么任务的
696 * 唤醒就需要两个条件: 延时的结束以及其他任务的唤醒操作. 又如, 任务被挂起时正在等待信
697 * 号量, 当任务从信号量的等待对列中清除后也不能立即运行, 而必须等到唤醒操作后.
698 *
699 * 参数: prio 为指定要获取挂起的任务优先级, 也可以指定参数OS_PRI0_SELF, 挂起任务本身. 此时,
700 * 下一个优先级最高的就绪任务将运行.
701 *
702 * 返回: OS_NO_ERR: 函数调用成功;
703 * OS_TASK_SUSPEND_IDLE: 试图挂起uC/OS-II中的空闲任务 (Idle task). 此为非法操作;
704 * OS_PRI0_INVALID: 参数指定的优先级大于OS_LOWEST_PRI0或没有设定OS_PRI0_SELF的值;
705 * OS_TASK_SUSPEND_PRI0: 要挂起的任务不存在.
706 *
707 * 注意: 1、在程序中OSTaskSuspend() 和OSTaskResume() 应该成对使用;
708 * 2、用OSTaskSuspend() 挂起的任务只能用OSTaskResume() 唤醒.
709 ****
710 */
711
712 #if OS_TASK_SUSPEND_EN > 0 //允许生成 OSTaskSuspend() 函数
713 INT8U OSTaskSuspend (INT8U prio) //无条件挂起一个任务 (任务优先级)
714 {
715     #if OS_CRITICAL_METHOD == 3 //中断函数被设定为模式3
716         OS_CPU_SR cpu_sr;
717     #endif
718     BOOLEAN self; //定义布尔(=1为任务本身)
719     OS_TCB *ptcb; //定义消息事件的任务控制块指针
720
721     #if OS_ARG_CHK_EN > 0 //所有参数必须在指定的参数内
722         if (prio == OS_IDLE_PRI0) { //检查任务的优先级是否是空闲任务
723             return (OS_TASK_SUSPEND_IDLE); //试图挂起uC/OS-II中的空闲任务 (Idle task). 此为非法操作
724         } //当任务 >= 最低优先级 并且 任务不是本身
725         if (prio >= OS_LOWEST_PRI0 && prio != OS_PRI0_SELF) { //检查任务是否是合法的
726             return (OS_PRI0_INVALID); //参数指定的优先级大于OS_LOWEST_PRI0或没有设定OS_PRI0_SELF的值
727         }
728     #endif
729     #endif
730     OS_ENTER_CRITICAL(); //关闭中断
731     if (prio == OS_PRI0_SELF) { //如果删除者是任务本身
732         prio = OSTCBCur->OSTCBPrio; //从当前任务的任务控制块OS_TCB中获取当前任务的优先级
733         self = TRUE; //是任务本身(为1)
734     } else if (prio == OSTCBCur->OSTCBPrio) { //也可以通过指定优先级, 挂起调用本函数的任务
735         self = TRUE; //是任务本身(为1)
736     } else { //否则
737         self = FALSE; //要删除的不是任务本身
738     }
739     if ((ptcb = OSTCBPrioTbl[prio]) == (OS_TCB *)0) { //检验要挂起的任务是否存在(为1任务存在)
740         OS_EXIT_CRITICAL(); //打开中断
741         return (OS_TASK_SUSPEND_PRI0); //返回(要挂起的任务不存在)
742     }
743     if ((OSRdyTbl[ptcb->OSTCBY] & ~ptcb->OSTCBBitX) == 0x00) { //如果该任务存在
744         OSRdyGrp &= ~ptcb->OSTCBBitY; //从就绪表中清除它(等待事件的话也删除)
745     }
746     ptcb->OSTCBStat |= OS_STAT_SUSPEND; //在任务的状态字中标明' SUSPENDED', 表示任务被挂起
747     OS_EXIT_CRITICAL(); //打开中断
748     if (self == TRUE) { //如果是任务本身(为1)
749         OS_Sched(); //任务调度, 最高任务优先级运行
750         //说明: self == TRUE, 删除本身, 需要进入任务调度执行新的任务
751         //self = FALSE, 删除的是其它任务, 无需进入调度, 可继续执行本任务
752     }
753     return (OS_NO_ERR); //返回(函数调用成功)
754 }
755 #endif
756 /*$PAGE*/
757 /*
758 ****
759 * 获取任务信息, 函数返回任务TCB的一个完整的拷贝 (QUERY A TASK)
760 *

```

```

761 * 描述: 获取任务信息, 函数返回任务TCB的一个完整的拷贝. 应用程序必须建立一个OS_TCB类型的数据结
762 *       构容纳返回的数据. 需要提醒用户的是, 在对任务OS_TCB对象中的数据操作时要小心, 尤其是数据
763 *       项OSTCBNext和OSTCBPrev. 它们分别指向TCB链表中的后一项和前一项.
764 *
765 * 参数: prio 为指定要获取TCB内容的任务优先级, 也可以指定参数OS_PRIO_SELF, 获取调用任务的信息.
766 *       pdata指向一个OS_TCB类型的数据结构, 容纳返回的任务TCB的一个拷贝.
767 *
768 * 返回: OS_NO_ERR:      函数调用成功;
769 *       OS_PRIO_ERR:    参数指定的任务非法;
770 *       OS_PRIO_INVALID: 参数指定的优先级大于OS_LOWEST_PRIO.
771 * 注意: 任务控制块 (TCB) 中所包含的数据成员取决于下述开关量在初始化时的设定 (参见OS_CFG.H)
772 *       OS_TASK_CREATE_EN
773 *       OS_Q_EN
774 *       OS_MBOX_EN
775 *       OS_SEM_EN
776 *       OS_TASK_DEL_EN
777 *****
778 */
779
780 #if OS_TASK_QUERY_EN > 0                                //允许(1)生成OSTaskQuery() 代码
781 INT8U OSTaskQuery (INT8U prio, OS_TCB *pdata)           //获取任务信息(任务指针、保存数据结构指针)
782 {
783     #if OS_CRITICAL_METHOD == 3                          //中断函数被设定为模式3
784         OS_CPU_SR cpu_sr;
785     #endif
786     OS_TCB *ptcb;                                         //定义消息事件的任务控制块指针
787
788     #if OS_ARG_CHK_EN > 0
789         //当任务 >= 最低优先级 并且 任务不是本身
790         if (prio > OS_LOWEST_PRIO && prio != OS_PRIO_SELF) { //检查任务是否是合法的
791             return (OS_PRIO_INVALID); //参数指定的优先级大于OS_LOWEST_PRIO
792         }
793     #endif
794     OS_ENTER_CRITICAL(); //关闭中断
795     if (prio == OS_PRIO_SELF) { //检查的删除者是否是任务(优先级)本身
796         prio = OSTCBCur->OSTCBPrio; //正在运行的优先级(任务本身的优先级)
797     } //获取(信息)任务必须存在
798     if ((ptcb = OSTCBPrioTbl[prio]) == (OS_TCB *)0) { //调用这个任务的优先级的就绪值
799         OS_EXIT_CRITICAL(); //打开中断
800         return (OS_PRIO_ERR); //返回(参数指定的任务非法)
801     }
802     memcpy(pdata, ptcb, sizeof(OS_TCB)); //所有的域(通过赋值语句)一次性复制
803     OS_EXIT_CRITICAL(); //打开中断
804     return (OS_NO_ERR); //返回获取信息成功(函数调用成功)
805 }
806 #endif
807
808

```

```

1  /*
2  ****
3  *                                     uC/OS-II实时控制内核
4  *                                     主要的包含文件
5  *                                     时钟管理项
6  * 文    件: OS_TIME.C          时钟管理代码
7  * 作    者: Jean J. Labrosse
8  * 中文注解: 钟常慰  zhongcw @ 126.com  译注版本: 1.0  请尊重原版内容
9  ****
10 */
11
12 #ifndef OS_MASTER_FILE          //是否已定义OS_MASTER_FILE主文件
13 #include "includes.h"          //包含"includes.h"文件, 部分C语言头文件的汇总打包文件
14 #endif                          //定义结束
15
16 /*
17 ****
18 *                                     将一个任务延时若干个时钟节拍(DELAY TASK 'n' TICKS (n from 0 to 65535))
19 *
20 * 描述: 将一个任务延时若干个时钟节拍。如果延时时间大于0, 系统将立即进行任务调度。延时时间的长度
21 *       可从0到65535个时钟节拍。延时时间0表示不进行延时, 函数将立即返回调用者。延时的具体时间依
22 *       赖于系统每秒钟有多少时钟节拍(由文件SO_CFG.H中的常量OS_TICKS_PER_SEC设定)。
23 *
24 * 附加: 调用该函数会使uC/OS-ii进行一次任务调度, 并且执行下一个优先级最高的就绪态任务。任务调用
25 *       OSTimeDly()后, 一旦规定的时间期满或者有其它的任务通过调用OSTimeDlyResume()取消了延时,
26 *       它就会马上进入就绪状态。注意, 只有当该任务在所有就绪任务中具有最高的优先级时, 它才会立即
27 *       运行。
28 * 参数: ticks  为要延时的时钟节拍数。(一个1 到65535之间的数)
29 *
30 * 返回: 无
31 * 注意: 注意到延时时间0表示不进行延时操作, 而立即返回调用者。为了确保设定的延时时间, 建议用户设定
32 *       的时钟节拍数加1。例如, 希望延时10个时钟节拍, 可设定参数为11。
33 ****
34 */
35 void OSTimeDly (INT16U ticks)    //任务延时函数(时钟节拍数)
36 {
37     #if OS_CRITICAL_METHOD == 3    //中断函数被设定为模式3
38         OS_CPU_SR  cpu_sr;
39     #endif
40
41
42     if (ticks > 0) {              //如果延时设定为0值, 表示不想对任务延时, 返回调用任务
43         OS_ENTER_CRITICAL();      //关闭中断
44         if ((OSRdyTbl[OSTCBCur->OSTCBY] &= ~OSTCBCur->OSTCBBitX) == 0) {
45             OSRdyGrp &= ~OSTCBCur->OSTCBBitY;
46         }
47
48         OSTCBCur->OSTCBDly = ticks; //接着, 这个延时节拍数会被保存在当前任务的OS_TCB中
49         OS_EXIT_CRITICAL();         //打开中断
50         OS_Sched();                //既然任务已经不再处于就绪状态, (任务调度)
51                                     //任务调度程序会执行下一个优先级最高的就绪任务。
52     }
53 }
54 /*$PAGE*/
55 /*
56 ****
57 *                                     将一个任务延时若干时间(DELAY TASK FOR SPECIFIED TIME)
58 *
59 * 描述: 将一个任务延时若干时间。延时的单位是小时、分、秒、毫秒。所以使用OSTimeDlyHMSM()比OSTimeDly()
60 *       更方便。调用OSTimeDlyHMSM()后, 如果延时时间不为0, 系统将立即进行任务调度。
61 *
62 * 参数: hours    为延时小时数, 范围从0-255. (max. is 255)
63 *       minutes  为延时分分钟数, 范围从0-59. (max. 59)
64 *       seconds   为延时秒数, 范围从0-59. (max. 59)
65 *       milli     为延时毫秒数, 范围从0-999. (max. 999)
66 *       需要说明的是, 延时操作函数都是以时钟节拍为单位的。实际的延时时间是时钟节拍的整数倍。例如系统
67 *       每次时钟节拍间隔是10ms, 如果设定延时为5ms, 将不产生任何延时操作, 而设定延时15ms, 实际的延时是
68 *       两个时钟节拍, 也就是20ms。
69 *
70 * 附加: 调用OSTimeDlyHMSM()函数也会使uC/OS-ii进行一次任务调度, 并且执行下一个优先级最高的就绪态任务。
71 *       任务调用OSTimeDlyHMSM()后, 一旦规定的时间期满或者有其它的任务通过调用OSTimeDlyResume()取消了延
72 *       时(恢复延时的任务OSTimeDlyResume()), 它就会马上处于就绪态。同样, 只有当该任务在所有就绪态任务
73 *       中具有最高的优先级时, 它才会立即运行。
74 *
75 * 返回: OS_NO_ERR          函数调用成功;
76 *       OS_TIME_INVALID_MINUTES  参数错误, 分钟数大于59;

```

```

77 *      OS_TIME_INVALID_SECONDS  参数错误,秒数大于59;
78 *      OS_TIME_INVALID_MS       参数错误,毫秒数大于999;
79 *      OS_TIME_ZERO_DLY         四个参数全为0.
80 *
81 * 注意: OSTimeDlyHMSM(0, 0, 0, 0)表示不进行延时操作,而立即返回调用者.另外,如果延时总时间超过65535个
82 * 时钟节拍,将不能用OSTimeDlyResume()函数终止延时并唤醒任务。
83 *****
84 */
85
86 #if OS_TIME_DLY_HMSM_EN > 0          //允许生成OSTimeDlyHMSM()函数代码
87 INT8U OSTimeDlyHMSM (INT8U hours, INT8U minutes, INT8U seconds, INT16U milli)
88 {
89     INT32U ticks;                    //将一个任务延时若干时间(设定时、分、秒、毫秒)
90     INT16U loops;                    //定义节拍数
91                                     //定义循环次数
92                                     //当设定值大于0值(是否时有效值)
93     if (hours > 0 || minutes > 0 || seconds > 0 || milli > 0) {
94         if (minutes > 59) {          //如果分钟>59,则返回参数错误,分钟数大于59;
95             return (OS_TIME_INVALID_MINUTES);
96         }
97         if (seconds > 59) {          //如果秒>59,则返回参数错误,秒数大于59;
98             return (OS_TIME_INVALID_SECONDS);
99         }
100        if (milli > 999) {            //如果毫秒>999,则返回参数错误,毫秒数大于999;
101            return (OS_TIME_INVALID_MILLI);
102        }
103    }
104    /*
105    * 因为uC/OS-ii只知道节拍,所以节拍总数是从指定的时间中计算出来的。很明显,程序清单中的程序并不是十分
106    * 有效的。笔者只是用这种方法告诉大家一个公式,这样用户就可以知道怎样计算总的节拍数了。真正有意义的只
107    * 是OS_TICKS_PER_SEC。下段程序决定了最接近需要延迟的时间的时钟节拍总数。500/OS_TICKS_PER_SECOND的值
108    * 基本上与0.5个节拍对应的毫秒数相同。例如,若将时钟频率(OS_TICKS_PER_SEC)设置成100Hz(10ms),4ms的延时
109    * 不会产生任何延时!而5ms的延时就等于延时10ms。
110    */
111    //计算程序输入的总的节拍数
112    ticks = ((INT32U)hours * 3600L + (INT32U)minutes * 60L + (INT32U)seconds) * OS_TICKS_PER_SEC
113           + OS_TICKS_PER_SEC * ((INT32U)milli + 500L / OS_TICKS_PER_SEC) / 1000L;
114    /*
115    * uC/OS-ii支持的延时最长为65,535个节拍。要想支持更长时间的延时,OSTimeDlyHMSM()确定了用户想延时多少
116    * 次超过65,535个节拍的数目和剩下的节拍数。例如,若OS_TICKS_PER_SEC的值为100,用户想延时15分钟,则
117    * OSTimeDlyHMSM()会延时15x60x100=90,000个时钟。这个延时会被分割成两次32,768个节拍的延时(因为用户只能
118    * 延时65,535个节拍而不是65536个节拍)和一次24,464个节拍的延时。在这种情况下,OSTimeDlyHMSM()首先考虑
119    * 剩下的节拍,然后是超过65,535的节拍数(即两个32,768个节拍延时)。
120    */
121    loops = (INT16U)(ticks / 65536L); //计算得商得倍数(多少个65536 时钟节拍)
122    ticks = ticks % 65536L;           //计算得余数
123    OSTimeDly((INT16U)ticks);         //先作余数清除
124    while (loops > 0) {                //如果节拍数超过65536 个时钟节拍
125        OSTimeDly(32768);              //执行两次延时,实现共65536 个时钟节拍
126        OSTimeDly(32768);
127        loops--;                       //继续减1,直到为0
128    }
129    return (OS_NO_ERR);                //返回(函数调用成功)
130 }
131 #endif
132 /*$PAGE*/
133 /*
134 *****
135 *      唤醒一个用OSTimeDly()或OSTimeDlyHMSM()函数延时的任务(RESUME A DELAYED TASK)
136 *
137 * 描述: 唤醒一个用OSTimeDly()或OSTimeDlyHMSM()函数延时的任务
138 * OSTimeDlyResume()函数不能唤醒一个用OSTimeDlyHMSM()延时,且延时时间总计超过65535个时钟节拍的
139 * 任务。例如,如果系统时钟为100Hz,OSTimeDlyResume()不能唤醒延时OSTimeDlyHMSM(0, 10, 55, 350)
140 * 或更长时间的任务。
141 * (OSTimeDlyHMSM(0, 10, 55, 350)共延时[10 minutes * 60 + (55+0.35)seconds] * 100 = 65,535次时
142 * 钟节拍---译者注)
143 *
144 * 参数: prio 为指定要唤醒任务的优先级
145 *
146 * 返回: OS_NO_ERR      函数调用成功
147 *       OS_PRIO_INVALID 参数指定的优先级大于OS_LOWEST_PRIO
148 *       OS_TIME_NOT_DLY 要唤醒的任务不在延时状态
149 *       OS_TASK_NOT_EXIST 指定的任务不存在
150 *
151 * 注意: 用户不应该用OSTimeDlyResume()去唤醒一个设置了等待超时操作,并且正在等待事件发生的任务。操作的
152 * 结果是使该任务结束等待,除非的确希望这么做。

```

```

153 *****
154 *      uC/OS-ii允许用户结束延时正处于延时期的任务。延时的任务可以不等待延时期满，而是通过其它任务取
155 *      消延时来使自己处于就绪态。这可以通过调用OSTimeDlyResume()和指定要恢复的任务的优先级来完成。
156 *      实际上，OSTimeDlyResume()也可以唤醒正在等待事件(参看任务间的通讯和同步)的任务，虽然这一点并
157 *      没有提到过。在这种情况下，等待事件发生的任务会考虑是否终止等待事件。
158 *      OSTimeDlyResume()的代码如程序，它首先要确保指定的任务优先级有效。接着，OSTimeDlyResume()要确
159 *      认要结束延时的任务是确实存在的。如果任务存在，OSTimeDlyResume()会检验任务是否在等待延时期满。
160 *      只要OS_TCB域中的OSTCBDly包含非0值就表明任务正在等待延时期满，因为任务调用了OSTimeDly()，
161 *      OSTimeDlyHMSM()或其它在第六章中所描述的PEND函数。然后延时就可以通过强制命令OSTCBDly为0来取消
162 *      。延时的任务有可能已被挂起了，这样的话，任务只有在没有被挂起的情况下才能处于就绪状态。当上面
163 *      的条件都满足后，任务就会被放在就绪表中。这时，OSTimeDlyResume()会调用任务调度程序来看被恢复
164 *      的任务是否拥有比当前任务更高的优先级。这会导致任务的切换。
165 *****
166 */
167
168 #if OS_TIME_DLY_RESUME_EN > 0                //允许生成OSTimeDlyResume() 函数代码
169 INT8U OSTimeDlyResume (INT8U prio)           //唤醒一个用OSTimeDly()或OSTimeDlyHMSM()函数的任务(优先级)
170 {
171     #if OS_CRITICAL_METHOD == 3              //中断函数被设定为模式3
172         OS_CPU_SR  cpu_sr;
173     #endif
174     OS_TCB  *ptcb;                            //定义任务控制块优先级表变量
175
176
177     if (prio >= OS_LOWEST_PRIO) {            //当任务指针大于等于最低(大)优先级时，确保优先级有效
178         return (OS_PRIO_INVALID);            //返回(参数指定的优先级大于OS_LOWEST_PRIO)
179     }
180     OS_ENTER_CRITICAL();                      //关闭中断
181     ptcb = (OS_TCB *)OSTCBPrioTbl[prio];     //ptcb = 任务控制块优先级表当前优先级
182     if (ptcb != (OS_TCB *)0) {                //确保要结束的延时的任务是确实存在的
183         if (ptcb->OSTCBDly != 0) {            //如果任务存在，程序会检验任务是否在等待延时期满，只要任务
184                                             //控制块的.OSTCBDly域非0值，就表明任务正在等待延时期满，因
185                                             //为任务调用了OSTimeDly()、OSTimeDlyHMSM()或其它的pend函数
186             ptcb->OSTCBDly = 0;                //通过使 .OSTCBDly为0而取消延时
187             if ((ptcb->OSTCBStat & OS_STAT_SUSPEND) == 0x00) {
188                 //延时的任务有可能已被挂起，然而任务在没有被挂起的情况下，
189                 //才能处于就绪态
190                 OSRdyGrp |= ptcb->OSTCBBitY;
191                 OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
192                 //当上面的条件都满足时，任务就会被放在就绪表中
193                 OS_EXIT_CRITICAL();            //打开中断
194                 OS_Sched();                    //任务调度程序会执行下一个优先级最高的就绪任务(任务调度)
195             } else {                          //不然
196                 OS_EXIT_CRITICAL();            //打开中断
197             }
198             return (OS_NO_ERR);                //函数调用成功
199         } else {                              //否则
200             OS_EXIT_CRITICAL();                //打开中断
201             return (OS_TIME_NOT_DLY);          //返回(要唤醒的任务不在延时状态)
202         }
203     }
204     OS_EXIT_CRITICAL();                        //打开中断
205     return (OS_TASK_NOT_EXIST);                //返回(指定的任务不存在)
206 }
207 #endif
208 /*$PAGE*/
209 /*
210 *****
211 *      获取当前系统时钟数值(GET CURRENT SYSTEM TIME)
212 *
213 *      描述：获取当前系统时钟数值。系统时钟是一个32位的计数器，记录系统上电后或时钟重新设置后的时钟计数。
214 *
215 *      附加：无论时钟节拍何时发生，uC/OS-ii都会将一个32位的计数器加1。这个计数器在用户调用OSStart()初始
216 *      化多任务和4,294,967,295个节拍执行完一遍的时候从0开始计数。在时钟节拍的频率等于100Hz的时候，
217 *      这个32位的计数器每隔497天就重新开始计数。用户可以通过调用OSTimeGet()来获得该计数器的当前值。
218 *      也可以通过调用OSTimeSet()来改变该计数器的值。OSTimeGet()和OSTimeSet()两个函数的代码如程序。
219 *      注意，在访问OSTime的时候中断是关掉的。这是因为在大多数8位处理器上增加和拷贝一个32位的数都需要
220 *      要数条指令，这些指令一般都需要一次执行完毕，而不能被中断等因素打断
221 *
222 *      参数：无
223 *
224 *      返回：当前时钟计数(时钟节拍数)
225 *****
226 */
227
228 #if OS_TIME_GET_SET_EN > 0                    //允许生成OSTimeGet() 函数代码

```



```

229 INT32U OSTimeGet (void)           //获取当前系统时钟数值
230 {
231     #if OS_CRITICAL_METHOD == 3    //中断函数被设定为模式3
232         OS_CPU_SR cpu_sr;
233     #endif
234     INT32U ticks;                  //定义节拍数
235
236
237     OS_ENTER_CRITICAL();           //关闭中断
238     ticks = OSTime;                //获取当前系统时钟数值
239     OS_EXIT_CRITICAL();            //打开中断
240     return (ticks);                //返回系统时钟数值
241 }
242 #endif
243
244 /*
245 *****
246 *                                     设置当前系统时钟数值(SET SYSTEM CLOCK)
247 *
248 * 描述：设置当前系统时钟数值。系统时钟是一个32位的计数器，记录系统上电后或时钟重新设置后的时钟计数。
249 *
250 * 参数：ticks 要设置的时钟数，单位是时钟节拍数。
251 *
252 * 返回：无
253 *****
254 */
255
256 #if OS_TIME_GET_SET_EN > 0         //允许生成OSTimeSet() 函数代码
257 void OSTimeSet (INT32U ticks)      //设置当前系统时钟数值
258 {
259     #if OS_CRITICAL_METHOD == 3    //中断函数被设定为模式3
260         OS_CPU_SR cpu_sr;
261     #endif
262
263
264     OS_ENTER_CRITICAL();           //关闭中断
265     OSTime = ticks;                //设置当前系统时钟数值为多少个节拍数
266     OS_EXIT_CRITICAL();            //打开中断
267 }
268 #endif
269

```

```

1  /*
2  ****
3  *                                     PC SUPPORT FUNCTIONS
4  *
5  *                                     (c) Copyright 1992-1999, Jean J. Labrosse, Weston, FL
6  *                                     All Rights Reserved
7  *
8  * File : PC.C
9  * By   : Jean J. Labrosse
10 ****
11 */
12
13 #include "includes.h"
14
15 /*
16 ****
17 *                                     CONSTANTS
18 ****
19 */
20 #define DISP_BASE          0xB800      /* Base segment of display (0xB800=VGA, 0xB000=Mono) */
21 #define DISP_MAX_X         80          /* Maximum number of columns */
22 #define DISP_MAX_Y         25          /* Maximum number of rows */
23
24 #define TICK_TO_8254_CWR    0x43       /* 8254 PIT Control Word Register address. */
25 #define TICK_TO_8254_CTR0   0x40       /* 8254 PIT Timer 0 Register address. */
26 #define TICK_TO_8254_CTR1   0x41       /* 8254 PIT Timer 1 Register address. */
27 #define TICK_TO_8254_CTR2   0x42       /* 8254 PIT Timer 2 Register address. */
28
29 #define TICK_TO_8254_CTR0_MODE3 0x36    /* 8254 PIT Binary Mode 3 for Counter 0 control word. */
30 #define TICK_TO_8254_CTR2_MODE0 0xB0    /* 8254 PIT Binary Mode 0 for Counter 2 control word. */
31 #define TICK_TO_8254_CTR2_LATCH 0x80    /* 8254 PIT Latch command control word */
32
33 #define VECT_TICK           0x08       /* Vector number for 82C54 timer tick */
34 #define VECT_DOS_CHAIN      0x81       /* Vector number used to chain DOS */
35
36 /*
37 ****
38 *                                     LOCAL GLOBAL VARIABLES
39 ****
40 */
41
42 static INT16U   PC_ElapsedOverhead;
43 static jmp_buf  PC_JumpBuf;
44 static BOOLEAN  PC_ExitFlag;
45 void            (*PC_TickISR)(void);
46
47 /*$PAGE*/
48 /*
49 ****
50 *                                     DISPLAY A SINGLE CHARACTER AT 'X' & 'Y' COORDINATE
51 *
52 * Description : This function writes a single character anywhere on the PC's screen. This function
53 *               writes directly to video RAM instead of using the BIOS for speed reasons. It assumed
54 *               that the video adapter is VGA compatible. Video RAM starts at absolute address
55 *               0x000B8000. Each character on the screen is composed of two bytes: the ASCII character
56 *               to appear on the screen followed by a video attribute. An attribute of 0x07 displays
57 *               the character in WHITE with a black background.
58 *
59 * Arguments   : x      corresponds to the desired column on the screen. Valid columns numbers are from
60 *                   0 to 79. Column 0 corresponds to the leftmost column.
61 *               y      corresponds to the desired row on the screen. Valid row numbers are from 0 to 24.
62 *                   Line 0 corresponds to the topmost row.
63 *               c      Is the ASCII character to display. You can also specify a character with a
64 *                   numeric value higher than 128. In this case, special character based graphics
65 *                   will be displayed.
66 *               color   specifies the foreground/background color to use (see PC.H for available choices)
67 *                   and whether the character will blink or not.
68 *
69 * Returns      : None
70 ****
71 */
72 void PC_Dispatch (INT8U x, INT8U y, INT8U c, INT8U color)
73 {
74     INT8U far *pscr;
75     INT16U   offset;
76

```

```

77
78     offset = (INT16U)y * DISP_MAX_X * 2 + (INT16U)x * 2; /* Calculate position on the screen */
79     pscr = (INT8U far *)MK_FP(DISP_BASE, offset);
80     *pscr++ = c; /* Put character in video RAM */
81     *pscr = color; /* Put video attribute in video RAM */
82 }
83 /*$PAGE*/
84 /*
85 ****
86 *
87 *
88 * Description : This function clears one of the 80 columns on the PC's screen by directly accessing video
89 *               RAM instead of using the BIOS. It assumed that the video adapter is VGA compatible.
90 *               Video RAM starts at absolute address 0x000B8000. Each character on the screen is
91 *               composed of two bytes: the ASCII character to appear on the screen followed by a video
92 *               attribute. An attribute of 0x07 displays the character in WHITE with a black background.
93 *
94 * Arguments   : x           corresponds to the desired column to clear. Valid column numbers are from
95 *                   0 to 79. Column 0 corresponds to the leftmost column.
96 *
97 *               color       specifies the foreground/background color combination to use
98 *                   (see PC.H for available choices)
99 *
100 * Returns    : None
101 ****
102 */
103 void PC_DispcClrCol (INT8U x, INT8U color)
104 {
105     INT8U far *pscr;
106     INT8U i;
107
108
109     pscr = (INT8U far *)MK_FP(DISP_BASE, (INT16U)x * 2);
110     for (i = 0; i < DISP_MAX_Y; i++) {
111         *pscr++ = ' '; /* Put ' ' character in video RAM */
112         *pscr = color; /* Put video attribute in video RAM */
113         pscr = pscr + DISP_MAX_X * 2; /* Position on next row */
114     }
115 }
116 /*$PAGE*/
117 /*
118 ****
119 *
120 *
121 * Description : This function clears one of the 25 lines on the PC's screen by directly accessing video
122 *               RAM instead of using the BIOS. It assumed that the video adapter is VGA compatible.
123 *               Video RAM starts at absolute address 0x000B8000. Each character on the screen is
124 *               composed of two bytes: the ASCII character to appear on the screen followed by a video
125 *               attribute. An attribute of 0x07 displays the character in WHITE with a black background.
126 *
127 * Arguments   : y           corresponds to the desired row to clear. Valid row numbers are from
128 *                   0 to 24. Row 0 corresponds to the topmost line.
129 *
130 *               color       specifies the foreground/background color combination to use
131 *                   (see PC.H for available choices)
132 *
133 * Returns    : None
134 ****
135 */
136 void PC_DispcClrRow (INT8U y, INT8U color)
137 {
138     INT8U far *pscr;
139     INT8U i;
140
141
142     pscr = (INT8U far *)MK_FP(DISP_BASE, (INT16U)y * DISP_MAX_X * 2);
143     for (i = 0; i < DISP_MAX_X; i++) {
144         *pscr++ = ' '; /* Put ' ' character in video RAM */
145         *pscr++ = color; /* Put video attribute in video RAM */
146     }
147 }
148 /*$PAGE*/
149 /*
150 ****
151 *
152 *

```

```

153 * Description : This function clears the PC's screen by directly accessing video RAM instead of using
154 *               the BIOS. It assumed that the video adapter is VGA compatible. Video RAM starts at
155 *               absolute address 0x000B8000. Each character on the screen is composed of two bytes:
156 *               the ASCII character to appear on the screen followed by a video attribute. An attribute
157 *               of 0x07 displays the character in WHITE with a black background.
158 *
159 * Arguments   : color    specifies the foreground/background color combination to use
160 *               (see PC.H for available choices)
161 *
162 * Returns      : None
163 *****
164 */
165 void PC_DisClnScr (INT8U color)
166 {
167     INT8U far *pscr;
168     INT16U    i;
169
170
171     pscr = (INT8U far *)MK_FP(DISP_BASE, 0x0000);
172     for (i = 0; i < (DISP_MAX_X * DISP_MAX_Y); i++) { /* PC display has 80 columns and 25 lines */
173         *pscr++ = ' '; /* Put ' ' character in video RAM */
174         *pscr++ = color; /* Put video attribute in video RAM */
175     }
176 }
177 /*$PAGE*/
178 /*
179 *****
180 *                               DISPLAY A STRING AT 'X' & 'Y' COORDINATE
181 *
182 * Description : This function writes an ASCII string anywhere on the PC's screen. This function writes
183 *               directly to video RAM instead of using the BIOS for speed reasons. It assumed that the
184 *               video adapter is VGA compatible. Video RAM starts at absolute address 0x000B8000. Each
185 *               character on the screen is composed of two bytes: the ASCII character to appear on the
186 *               screen followed by a video attribute. An attribute of 0x07 displays the character in
187 *               WHITE with a black background.
188 *
189 * Arguments   : x        corresponds to the desired column on the screen. Valid columns numbers are from
190 *               0 to 79. Column 0 corresponds to the leftmost column.
191 *               y        corresponds to the desired row on the screen. Valid row numbers are from 0 to 24.
192 *               Line 0 corresponds to the topmost row.
193 *               s        Is the ASCII string to display. You can also specify a string containing
194 *               characters with numeric values higher than 128. In this case, special character
195 *               based graphics will be displayed.
196 *               color    specifies the foreground/background color to use (see PC.H for available choices)
197 *               and whether the characters will blink or not.
198 *
199 * Returns      : None
200 *****
201 */
202 void PC_DisPStr (INT8U x, INT8U y, INT8U *s, INT8U color)
203 {
204     INT8U far *pscr;
205     INT16U    offset;
206
207
208     offset = (INT16U)y * DISP_MAX_X * 2 + (INT16U)x * 2; /* Calculate position of 1st character */
209     pscr = (INT8U far *)MK_FP(DISP_BASE, offset);
210     while (*s) {
211         *pscr++ = *s++; /* Put character in video RAM */
212         *pscr++ = color; /* Put video attribute in video RAM */
213     }
214 }
215 /*$PAGE*/
216 /*
217 *****
218 *                               RETURN TO DOS
219 *
220 * Description : This functions returns control back to DOS by doing a 'long jump' back to the saved
221 *               location stored in 'PC_JumpBuf'. The saved location was established by the function
222 *               'PC_DOSSaveReturn()'. After execution of the long jump, execution will resume at the
223 *               line following the 'set jump' back in 'PC_DOSSaveReturn()'. Setting the flag
224 *               'PC_ExitFlag' to TRUE ensures that the 'if' statement in 'PC_DOSSaveReturn()' executes.
225 *
226 * Arguments   : None
227 *
228 * Returns      : None

```

```

229 *****
230 */
231 void PC_DOSReturn (void)
232 {
233     PC_ExitFlag = TRUE;                /* Indicate we are returning to DOS */
234     longjmp(PC_JumpBuf, 1);            /* Jump back to saved environment */
235 }
236 /*$PAGE*/
237 /*
238 *****
239 *                               SAVE DOS RETURN LOCATION
240 *
241 * Description : This function saves the location of where we are in DOS so that it can be recovered.
242 *               This allows us to abort multitasking under uC/OS-II and return back to DOS as if we had
243 *               never left. When this function is called by 'main()', it sets 'PC_ExitFlag' to FALSE
244 *               so that we don't take the 'if' branch. Instead, the CPU registers are saved in the
245 *               long jump buffer 'PC_JumpBuf' and we simply return to the caller. If a 'long jump' is
246 *               performed using the jump buffer then, execution would resume at the 'if' statement and
247 *               this time, if 'PC_ExitFlag' is set to TRUE then we would execute the 'if' statements and
248 *               restore the DOS environment.
249 *
250 * Arguments   : None
251 *
252 * Returns     : None
253 *****
254 */
255 void PC_DOSSaveReturn (void)
256 {
257     #if OS_CRITICAL_METHOD == 3                /* Allocate storage for CPU status register */
258         OS_CPU_SR cpu_sr;
259     #endif
260
261
262     PC_ExitFlag = FALSE;                /* Indicate that we are not exiting yet! */
263     OSTickDOSctr = 1;                    /* Initialize the DOS tick counter */
264     PC_TickISR = PC_VectGet(VECT_TICK);    /* Get MS-DOS's tick vector */
265
266     PC_VectSet(VECT_DOS_CHAIN, PC_TickISR); /* Store MS-DOS's tick to chain */
267
268     setjmp(PC_JumpBuf);                  /* Capture where we are in DOS */
269     if (PC_ExitFlag == TRUE) {            /* See if we are exiting back to DOS */
270         OS_ENTER_CRITICAL();
271         PC_SetTickRate(18);                /* Restore tick rate to 18.2 Hz */
272         OS_EXIT_CRITICAL();
273         PC_VectSet(VECT_TICK, PC_TickISR); /* Restore DOS's tick vector */
274         PC_DispClrScr(DISP_FGND_WHITE + DISP_BGND_BLACK); /* Clear the display */
275         exit(0);                          /* Return to DOS */
276     }
277 }
278 /*$PAGE*/
279 /*
280 *****
281 *                               ELAPSED TIME INITIALIZATION
282 *
283 * Description : This function initialize the elapsed time module by determining how long the START and
284 *               STOP functions take to execute. In other words, this function calibrates this module
285 *               to account for the processing time of the START and STOP functions.
286 *
287 * Arguments   : None.
288 *
289 * Returns     : None.
290 *****
291 */
292 void PC_ElapsedInit(void)
293 {
294     PC_ElapsedOverhead = 0;
295     PC_ElapsedStart();
296     PC_ElapsedOverhead = PC_ElapsedStop();
297 }
298 /*$PAGE*/
299 /*
300 *****
301 *                               INITIALIZE PC'S TIMER #2
302 *
303 * Description : This function initialize the PC's Timer #2 to be used to measure the time between events.
304 *               Timer #2 will be running when the function returns.

```



```

305 *
306 * Arguments   : None.
307 *
308 * Returns     : None.
309 ****
310 */
311 void PC_ElapsedStart(void)
312 {
313     #if OS_CRITICAL_METHOD == 3                /* Allocate storage for CPU status register */
314         OS_CPU_SR  cpu_sr;
315     #endif
316     INT8U    data;
317
318     OS_ENTER_CRITICAL();
319     data = (INT8U)inp(0x61);                    /* Disable timer #2 */
320     data &= 0xFE;
321     outp(0x61, data);
322     outp(TICK_TO_8254_CWR, TICK_TO_8254_CTR2_MODE0); /* Program timer #2 for Mode 0 */
323     outp(TICK_TO_8254_CTR2, 0xFF);
324     outp(TICK_TO_8254_CTR2, 0xFF);
325     data |= 0x01;                                /* Start the timer */
326     outp(0x61, data);
327     OS_EXIT_CRITICAL();
328 }
329
330 /*$PAGE*/
331 /*
332 ****
333                                     STOP THE PC'S TIMER #2 AND GET ELAPSED TIME
334 *
335 * Description : This function stops the PC's Timer #2, obtains the elapsed counts from when it was
336 *                started and converts the elapsed counts to micro-seconds.
337 *
338 * Arguments   : None.
339 *
340 * Returns     : The number of micro-seconds since the timer was last started.
341 *
342 * Notes       : - The returned time accounts for the processing time of the START and STOP functions.
343 *                - 54926 represents 54926S-16 or, 0.838097 which is used to convert timer counts to
344 *                micro-seconds. The clock source for the PC's timer #2 is 1.19318 MHz (or 0.838097 uS)
345 ****
346 */
347 INT16U PC_ElapsedStop(void)
348 {
349     #if OS_CRITICAL_METHOD == 3                /* Allocate storage for CPU status register */
350         OS_CPU_SR  cpu_sr;
351     #endif
352     INT8U    data;
353     INT8U    low;
354     INT8U    high;
355     INT16U    cnts;
356
357     OS_ENTER_CRITICAL();
358     data = (INT8U)inp(0x61);                    /* Disable the timer */
359     data &= 0xFE;
360     outp(0x61, data);
361     outp(TICK_TO_8254_CWR, TICK_TO_8254_CTR2_LATCH); /* Latch the timer value */
362     low = inp(TICK_TO_8254_CTR2);
363     high = inp(TICK_TO_8254_CTR2);
364     cnts = (INT16U)0xFFFF - (((INT16U)high << 8) + (INT16U)low); /* Compute time it took for operation */
365     OS_EXIT_CRITICAL();
366     return ((INT16U)((ULONG)cnts * 54926L >> 16) - PC_ElapsedOverhead);
367 }
368
369 /*$PAGE*/
370 /*
371 ****
372                                     GET THE CURRENT DATE AND TIME
373 *
374 * Description: This function obtains the current date and time from the PC.
375 *
376 * Arguments   : s      is a pointer to where the ASCII string of the current date and time will be stored.
377 *                You must allocate at least 21 bytes (includes the NUL) of storage in the return
378 *                string. The date and time will be formatted as follows:
379 *
380 *                "YYYY-MM-DD HH:MM:SS"

```

```

381 *
382 * Returns      : none
383 ****
384 */
385 void PC_GetDateTime (char *s)
386 {
387     struct time now;
388     struct date today;
389
390
391     gettime(&now);
392     getdate(&today);
393     sprintf(s, "%04d-%02d-%02d %02d:%02d:%02d",
394             today.da_year,
395             today.da_mon,
396             today.da_day,
397             now.ti_hour,
398             now.ti_min,
399             now.ti_sec);
400 }
401 /*$PAGE*/
402 /*
403 ****
404 *                                     CHECK AND GET KEYBOARD KEY
405 *
406 * Description: This function checks to see if a key has been pressed at the keyboard and returns TRUE if
407 *              so. Also, if a key is pressed, the key is read and copied where the argument is pointing
408 *              to.
409 *
410 * Arguments   : c      is a pointer to where the read key will be stored.
411 *
412 * Returns     : TRUE   if a key was pressed
413 *              FALSE  otherwise
414 ****
415 */
416 BOOLEAN PC_GetKey (INT16S *c)
417 {
418     if (kbhit()) {                                /* See if a key has been pressed */
419         *c = (INT16S) getch();                    /* Get key pressed */
420         return (TRUE);
421     } else {
422         *c = 0x00;                                /* No key pressed */
423         return (FALSE);
424     }
425 }
426 /*$PAGE*/
427 /*
428 ****
429 *                                     SET THE PC'S TICK FREQUENCY
430 *
431 * Description: This function is called to change the tick rate of a PC.
432 *
433 * Arguments   : freq      is the desired frequency of the ticker (in Hz)
434 *
435 * Returns     : none
436 *
437 * Notes       : 1) The magic number 2386360 is actually twice the input frequency of the 8254 chip which
438 *                is always 1.193180 MHz.
439 *              2) The equation computes the counts needed to load into the 8254. The strange equation
440 *                is actually used to round the number using integer arithmetic. This is equivalent to
441 *                the floating point equation:
442 *
443 *                1193180.0 Hz
444 *                count = ----- + 0.5
445 *                freq
446 ****
447 */
448 void PC_SetTickRate (INT16U freq)
449 {
450     #if OS_CRITICAL_METHOD == 3                    /* Allocate storage for CPU status register */
451         OS_CPU_SR cpu_sr;
452     #endif
453     INT16U count;
454
455
456     if (freq == 18) {                                /* See if we need to restore the DOS frequency */

```

```

457     count = 0;
458 } else if (freq > 0) {
459     /* Compute 8254 counts for desired frequency and ... */
460     /* ... round to nearest count */
461     count = (INT16U) (((INT32U) 2386360L / freq + 1) >> 1);
462 } else {
463     count = 0;
464 }
465 OS_ENTER_CRITICAL();
466 outp(TICK_TO_8254_CWR, TICK_TO_8254_CTR0_MODE3); /* Load the 8254 with desired frequency */
467 outp(TICK_TO_8254_CTR0, count & 0xFF); /* Low byte */
468 outp(TICK_TO_8254_CTR0, (count >> 8) & 0xFF); /* High byte */
469 OS_EXIT_CRITICAL();
470 }
471 /*$PAGE*/
472 /*
473 ****
474 *                                OBTAIN INTERRUPT VECTOR
475 *
476 * Description: This function reads the pointer stored at the specified vector.
477 *
478 * Arguments : vect is the desired interrupt vector number, a number between 0 and 255.
479 *
480 * Returns : The address of the Interrupt handler stored at the desired vector location.
481 ****
482 */
483 void *PC_VectGet (INT8U vect)
484 {
485     #if OS_CRITICAL_METHOD == 3 /* Allocate storage for CPU status register */
486         OS_CPU_SR cpu_sr;
487     #endif
488     INT16U *pvect;
489     INT16U off;
490     INT16U seg;
491
492
493     pvect = (INT16U *) MK_FP(0x0000, vect * 4); /* Point into IVT at desired vector location */
494     OS_ENTER_CRITICAL();
495     off = *pvect++; /* Obtain the vector's OFFSET */
496     seg = *pvect; /* Obtain the vector's SEGMENT */
497     OS_EXIT_CRITICAL();
498     return (MK_FP(seg, off));
499 }
500
501 /*
502 ****
503 *                                INSTALL INTERRUPT VECTOR
504 *
505 * Description: This function sets an interrupt vector in the interrupt vector table.
506 *
507 * Arguments : vect is the desired interrupt vector number, a number between 0 and 255.
508 *            isr is a pointer to a function to execute when the interrupt or exception occurs.
509 *
510 * Returns : none
511 ****
512 */
513 void PC_VectSet (INT8U vect, void (*isr)(void))
514 {
515     #if OS_CRITICAL_METHOD == 3 /* Allocate storage for CPU status register */
516         OS_CPU_SR cpu_sr;
517     #endif
518     INT16U *pvect;
519
520
521     pvect = (INT16U *) MK_FP(0x0000, vect * 4); /* Point into IVT at desired vector location */
522     OS_ENTER_CRITICAL();
523     *pvect++ = (INT16U) FP_OFF(isr); /* Store ISR offset */
524     *pvect = (INT16U) FP_SEG(isr); /* Store ISR segment */
525     OS_EXIT_CRITICAL();
526 }
527

```

```

1  /*
2  ****
3  *                                     PC支持功能(PC SUPPORT FUNCTIONS)
4  *
5  * 文    件: PC.H    PC显示字符设定值及函数原型
6  * 作    者: Jean J. Labrosse
7  * 中文注解: zhongcw zhongcw @ 126.com 译注版本: 1.0 请尊重原版内容
8  ****
9  */
10
11 /*
12 ****
13 *                                     VGA设定不同的颜色属性
14 *
15 * 描述:          PC.C提供在VGA显示器上显示ASCII字符和一些特殊字符, FGND:前景色  BBND:背景色  DISP:闪烁
16 *
17 *          函数原型参考: PC_DisChar(0, 0, 'A', DISP_FGND_WHITE + DISP_BGND_BLUE + DISP_BLINK);
18 ****
19 */
20 #define DISP_FGND_BLACK          0x00          //设定字符色彩为黑色, 值为0x00
21 #define DISP_FGND_BLUE           0x01          //设定字符色彩为蓝色, 值为0x01
22 #define DISP_FGND_GREEN          0x02          //设定字符色彩为绿色, 值为0x02
23 #define DISP_FGND_CYAN           0x03          //设定字符色彩为青色, 值为0x03
24 #define DISP_FGND_RED            0x04          //设定字符色彩为红色, 值为0x04
25 #define DISP_FGND_PURPLE         0x05          //设定字符色彩为紫色, 值为0x05
26 #define DISP_FGND_BROWN         0x06          //设定字符色彩为褐色, 值为0x06
27 #define DISP_FGND_LIGHT_GRAY     0x07          //设定字符色彩为白底灰色, 值为0x07
28 #define DISP_FGND_DARK_GRAY      0x08          //设定字符色彩为黑底灰色, 值为0x08
29 #define DISP_FGND_LIGHT_BLUE     0x09          //设定字符色彩为白底蓝色, 值为0x09
30 #define DISP_FGND_LIGHT_GREEN    0x0A          //设定字符色彩为白底绿色, 值为0x0A
31 #define DISP_FGND_LIGHT_CYAN     0x0B          //设定字符色彩为白底青色, 值为0x0B
32 #define DISP_FGND_LIGHT_RED      0x0C          //设定字符色彩为白底红色, 值为0x0C
33 #define DISP_FGND_LIGHT_PURPLE   0x0D          //设定字符色彩为白底紫色, 值为0x0D
34 #define DISP_FGND_YELLOW         0x0E          //设定字符色彩为黄色, 值为0x0E
35 #define DISP_FGND_WHITE          0x0F          //设定字符色彩为白色, 值为0x0F
36
37 #define DISP_BGND_BLACK          0x00          //设定背景色为黑色, 值为0x70
38 #define DISP_BGND_BLUE           0x10          //设定背景色为蓝色, 值为0x70
39 #define DISP_BGND_GREEN          0x20          //设定背景色为绿色, 值为0x70
40 #define DISP_BGND_CYAN           0x30          //设定背景色为青色, 值为0x70
41 #define DISP_BGND_RED            0x40          //设定背景色为红色, 值为0x70
42 #define DISP_BGND_PURPLE         0x50          //设定背景色为紫色, 值为0x70
43 #define DISP_BGND_BROWN         0x60          //设定背景色为褐色, 值为0x70
44 #define DISP_BGND_LIGHT_GRAY     0x70          //设定背景色为灰白色, 值为0x70
45
46 #define DISP_BLINK               0x80          //设置闪烁, 值为0x80
47
48 /*
49 ****
50 *                                     PC显示各功能函数原型
51 ****
52 */
53
54 void PC_DisChar(INT8U x, INT8U y, INT8U c, INT8U color); //在荧幕任何位置显示一个ASCII字符
55 void PC_DisClrCol(INT8U x, INT8U bgnd_color); //在荧幕上清一行
56 void PC_DisClrRow(INT8U y, INT8U bgnd_color); //在荧幕上清一列
57 void PC_DisClrScr(INT8U bgnd_color); //清屏
58 void PC_DisPStr(INT8U x, INT8U y, INT8U *s, INT8U color); //在荧幕任何位置显示一个ASCII字符串
59
60 void PC_DOSReturn(void); //保存和恢复DOS的环境设置
61 void PC_DOSSaveReturn(void); //允许程序在正式开始多任务前, 保存重要的寄存器值, 保证程序正常返回DOS
62
63 void PC_ElapsedInit(void); //计算PC_ElapsedStart(void)和PC_ElapsedStop(void)本身运行所需时间。
64
65 /*在执行PC_ElapsedStart(void)和PC_ElapsedStop(void)之前一定先调用PC_ElapsedInit(void)函数*/
66
67 void PC_ElapsedStart(void); //程序启动
68 INT16U PC_ElapsedStop(void); //程序停止
69
70 /*从PC的实时时钟芯片中获取当前的日期和时间, 将这些信息长度不少于21个ASCII字符串返回*/
71 void PC_GetDateTime(char *s);
72 BOOLEAN PC_GetKey(INT16S *c); //检查键盘是否被按下, 如果有, 返回按键值
73 /*将PC的时钟节拍从标准值18.20648Hz改变成更快, 一个200Hz的时钟节拍是18.20648Hz的11倍*/
74 void PC_SetTickRate(INT16U freq);
75
76 void *PC_VectGet(INT8U vect); //获取由中断向量值vect指定的中断服务子程序的地址

```

```
77 void    PC_VectSet(INT8U vect, void (*isr)(void)); //设定中断向量表的内容
78
```


如何在 BC4.5 下成功编译 uc0s 系统

----- 钟常慰

鉴于很多人编译 uc0s 都遇到很大的问题，我在实际的实践过程中也是如此，在编译成功的喜悦之际，也希望把它写下来，让其他的人少走更多的弯路，决定写下此文，希望大家能够早日对 uc0s 有一个全面的了解。

请在相关的网址上下载 BC4.5 的安装文件，我的是 41.6M 的安装文件（压缩后 41M），成功的编译了 uc0s 文件，由于网站的 BC 软件五花八门，所以很多人都有失败过，建议找和我一样的安装文件包，其中包含以下文件，大家可以放大查看一下。图一

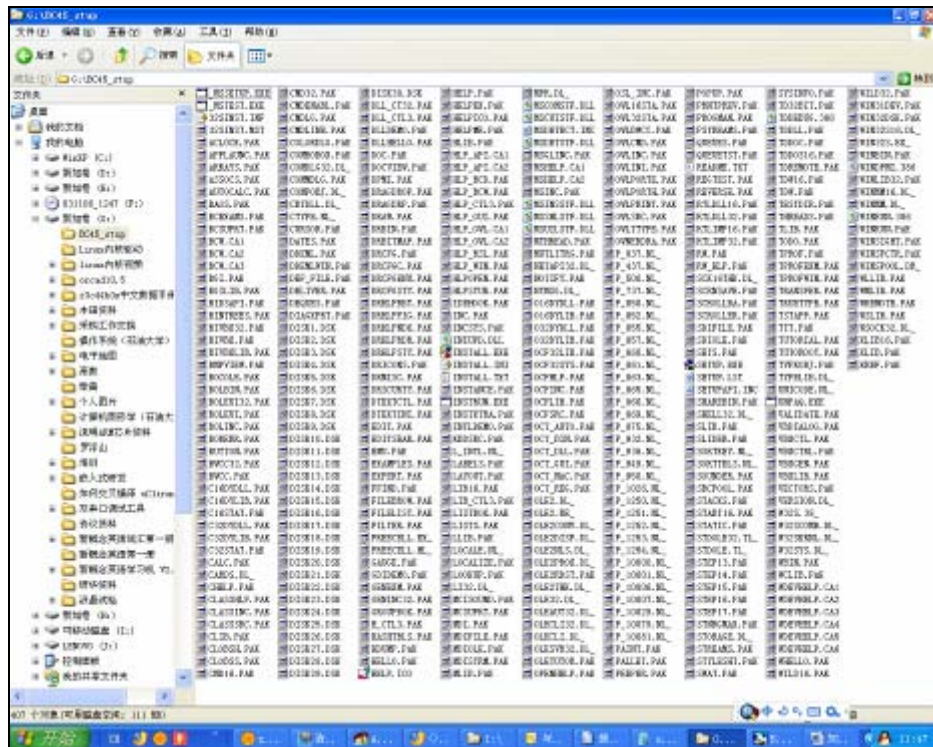


图 一

其中有一个叫着 INSTALL. EXE 的文件，点击后进入安装画面，图二

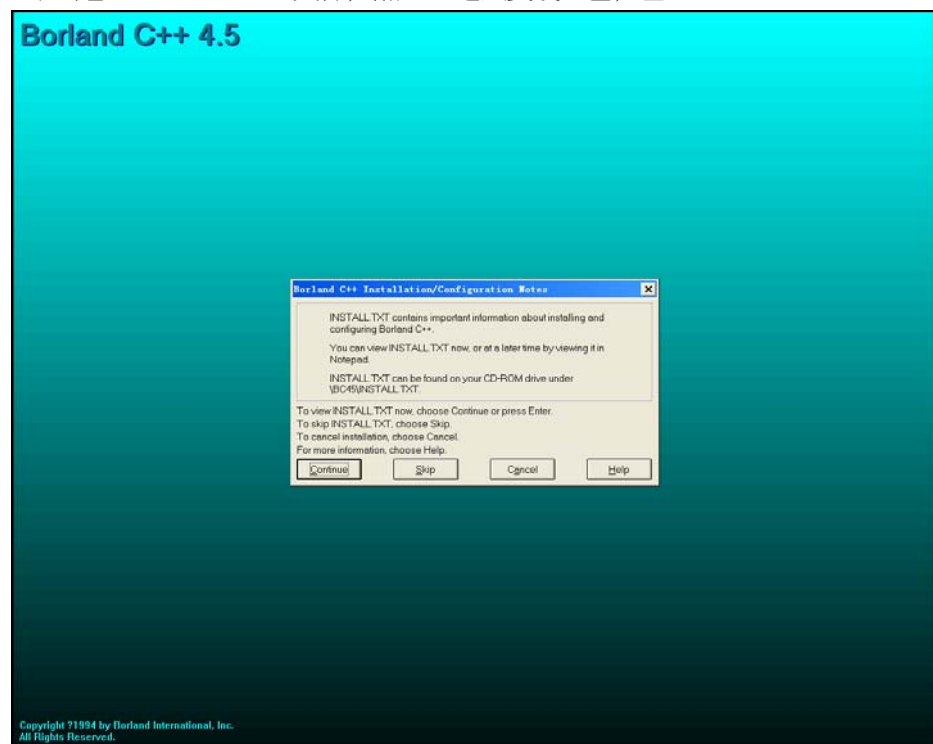


图 二

点击 skip 按钮进入安装，可以选中你需要的盘，我选的是 D 盘，所有的文件是在 D 盘下的 bc45 中，安装完成后，查看桌面的开始菜单，选择 Borland C++，打开软件，图三

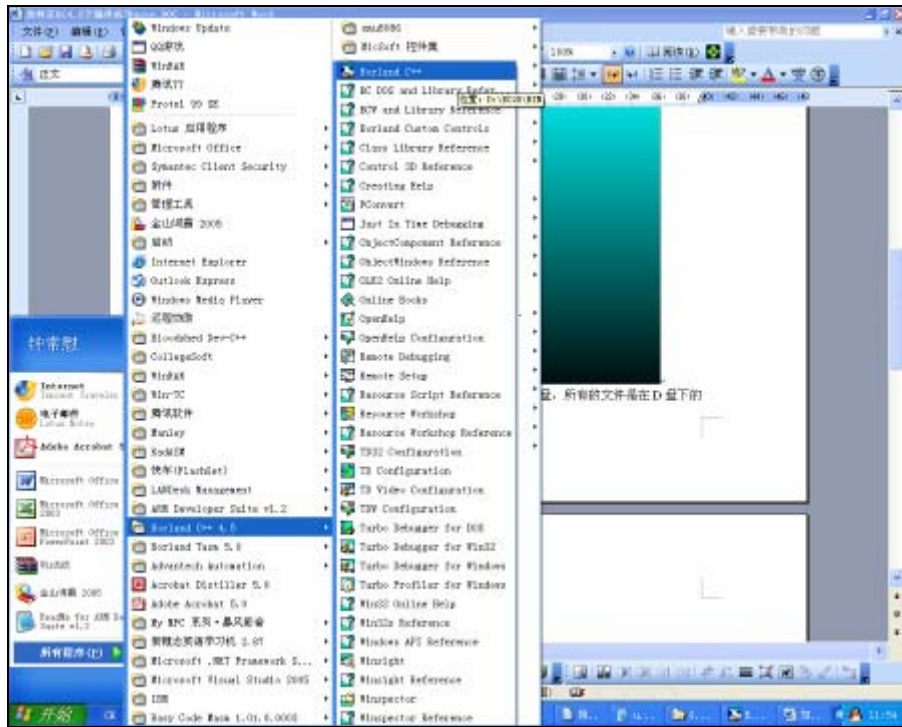


图 三

请看软件：图四

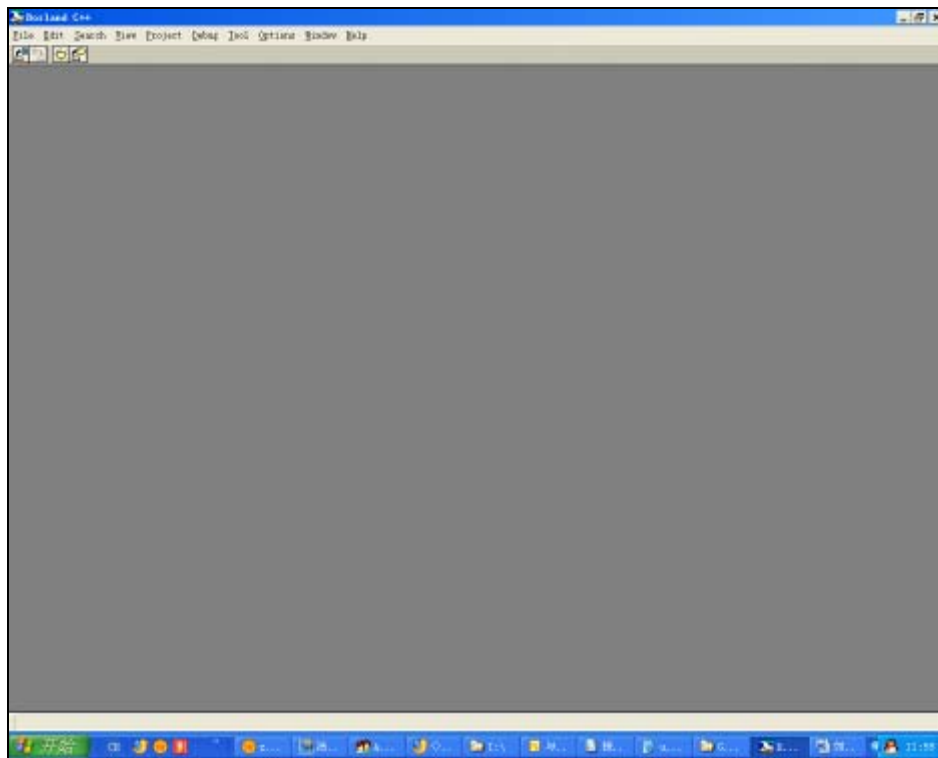


图 四

请打开你的 BC 安装文件夹，我的是在 D 盘下，图五

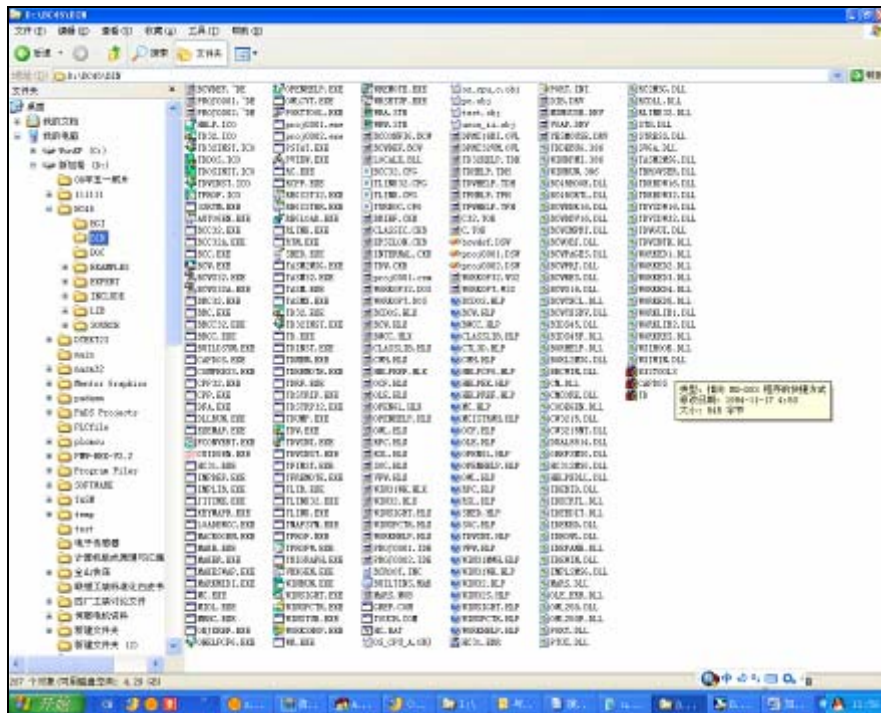


图 五

一定要查看你的 BIN 文件夹下是否包含了以下的非常重要编译文件，如 C/C++ 和 ASM 的编译器和链接器等，请看红色包含栏，这些非常重要。图六

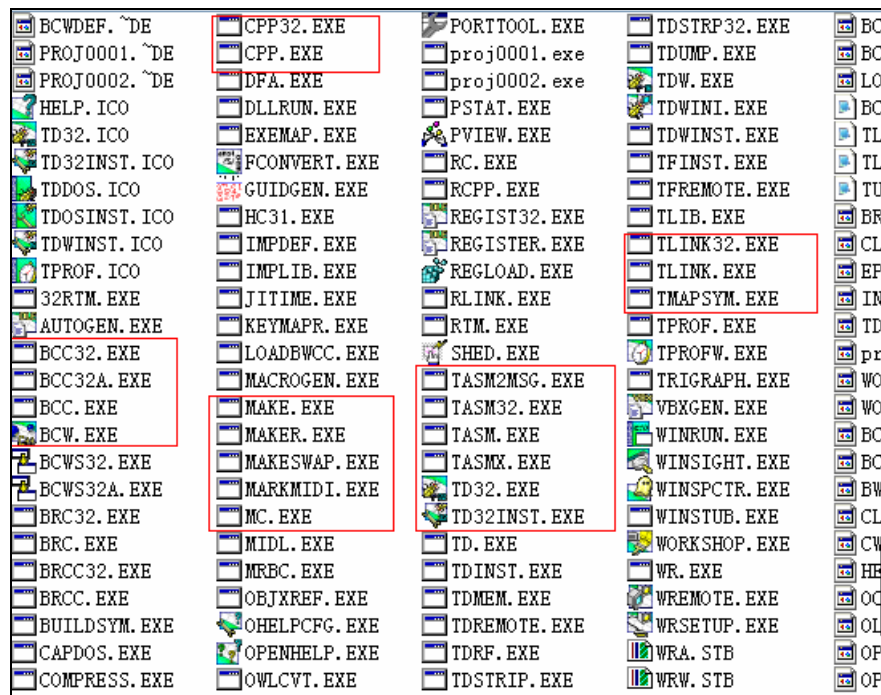


图 六

把邵贝贝翻译的这本学 uCOS 必备的《嵌入式实时操作系统 uC/OS-II》光盘里的 SOFTWARE 放到 D 盘下，参见图七图八

把 D:\SOFTWARE\uCOS-II\EX1_x86L\BC45\SOURCE\INCLUDES.H 复制到 c:\bc45\include 目录下。

然后将其中的相对路径。参见图八

#include "\software\ucos-ii\ix86l\bc45\os_cpu.h"

#include "os_cfg.h"

#include "\software\ucos-ii\source\ucos_ii.h"

#include "\software\blocks\pc\bc45\pc.h"

修改为绝对路径（我用的是 D 盘）

#include "D:\software\ucos-ii\ix86l\bc45\os_cpu.h"

#include "D:\SOFTWARE\uCOS-II\EX1_x86L\BC45\SOURCE\os_cfg.h"

```
#include "D:\software\ucos-ii\source\ucos_ii.h"
#include "D:\software\blocks\pc\bc45\pc.h"
```

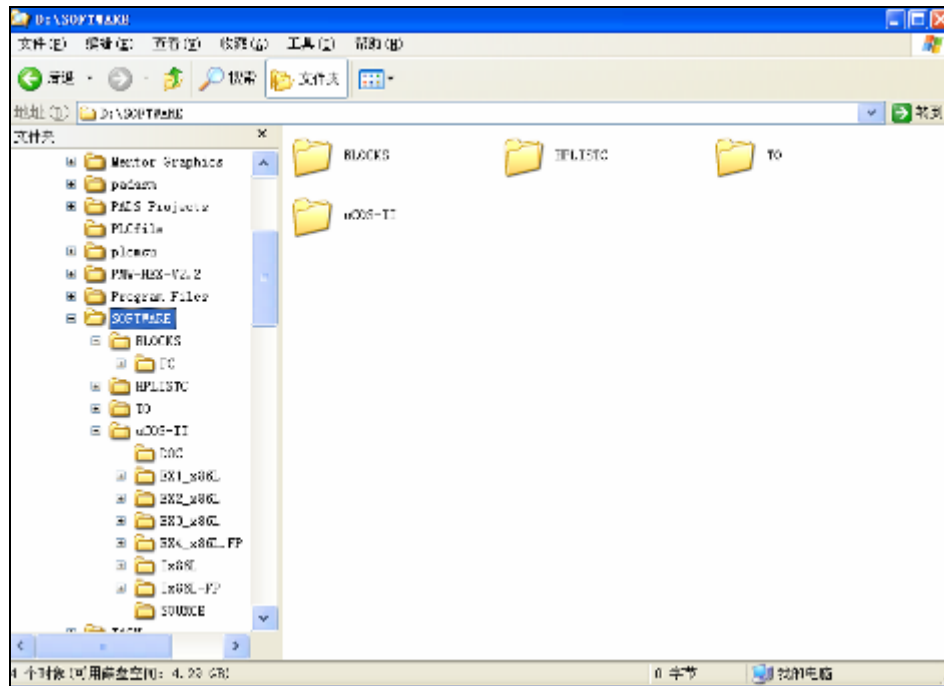


图 七

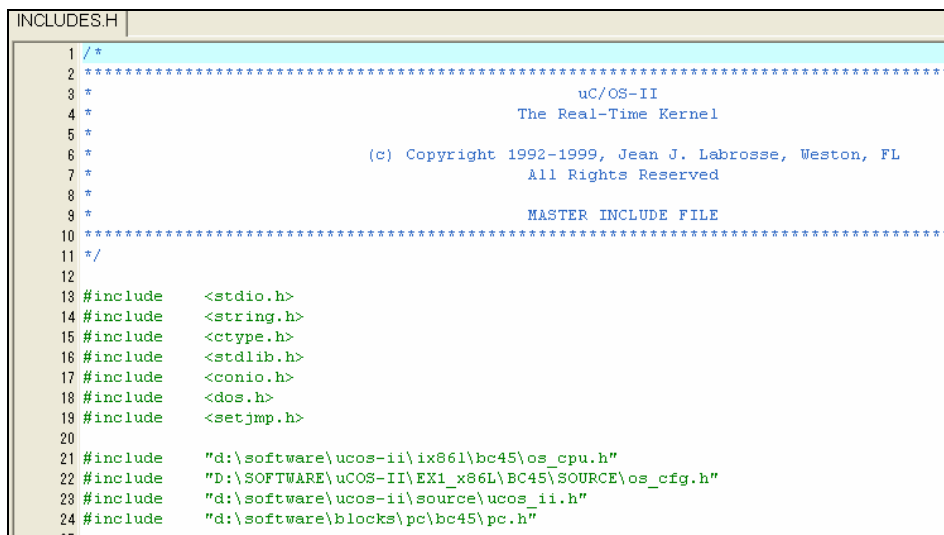


图 八

打开软件后 Borland C++4.5 软件后，打开 project 菜单，点击 new project...，出现对话框，里面有默认的建立文件 prj0001 文件，你可以在工程名称中输入你的名字建立一个文件夹，选择自己的盘，在 target type 下选择 Application[.exe]，在 platform 中选择 DOS (Standard)，点击 OK，出现图十



图 九

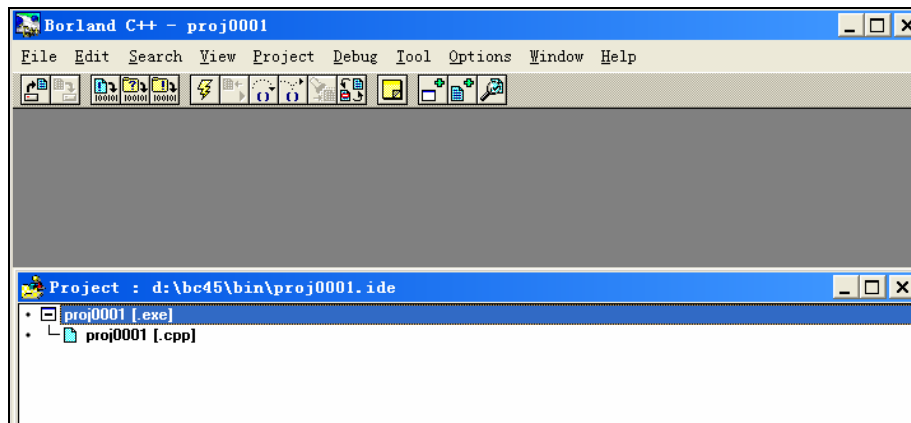


图 十

将 proj001.cpp 文件删除，在选择 proj.exe 文件后点击右键出现菜单栏，选择 add note，以此添加以下文件参见图十一

D:\SOFTWARE\BLOCKS\PC\BC45\PC.C

D:\SOFTWARE\uCOS-II\EX1_x86L\BC45\SOURCE\TEST.C

D:\SOFTWARE\uCOS-II\Ix86L\BC45\OS_CPU.C.C

D:\SOFTWARE\uCOS-II\Ix86L\BC45\OS_CPU_A.ASM

D:\SOFTWARE\uCOS-II\SOURCE\uCOS_II.C

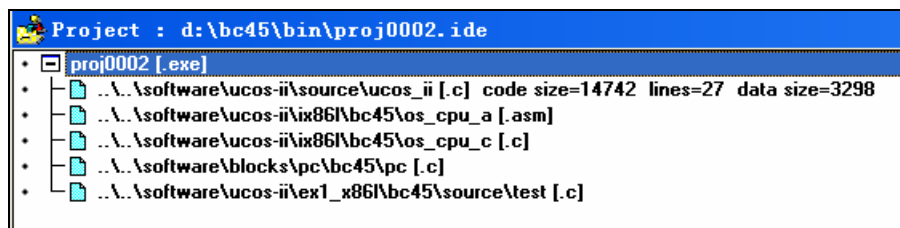


图 十一

如是这样添加将会编译错误，参见图十二

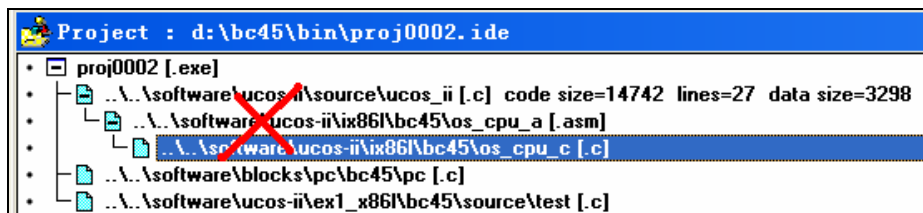
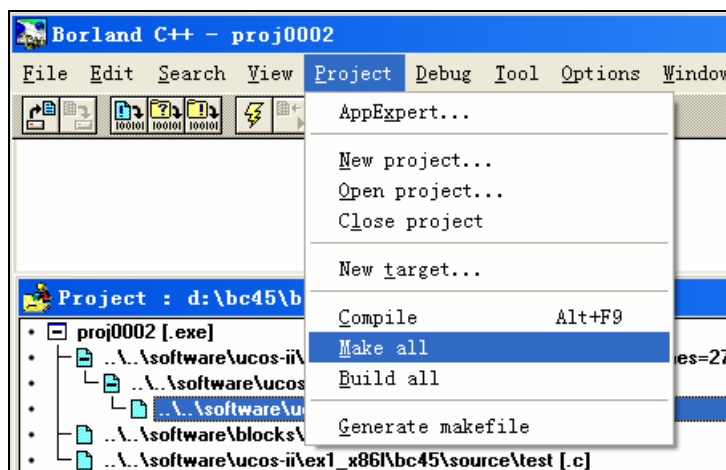


图 十二

点击 Make all 进行编译，参见图十三



图十三

如果成功，将出现一下对话框，参见图十四

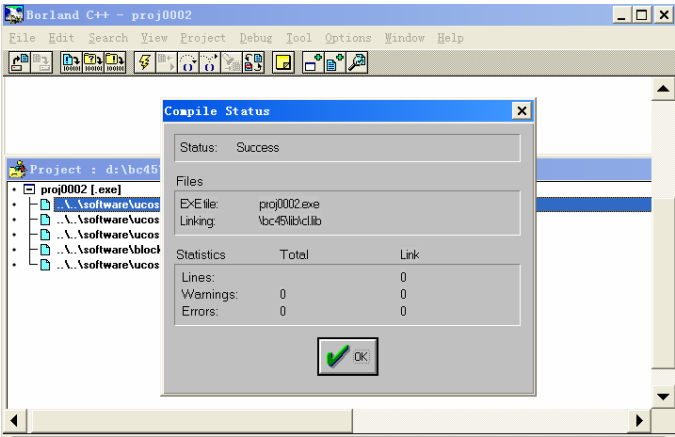


图 十四

点击 Debug 菜单下的 Run，参见图十五，出现以下画面，到此，你的编译成功了，参见图十六

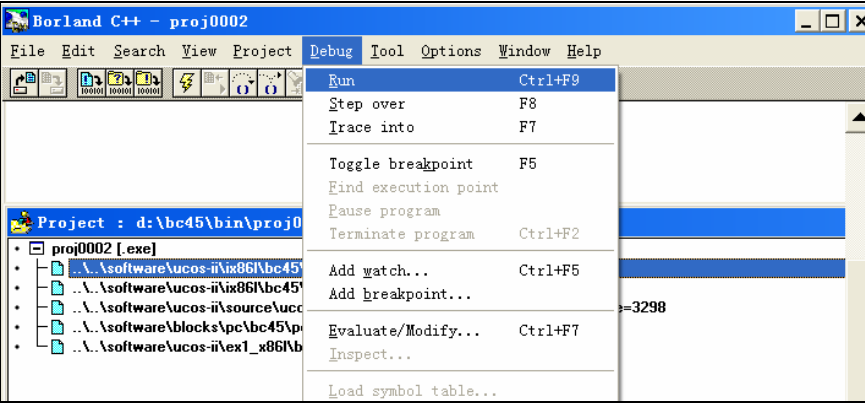


图 十五

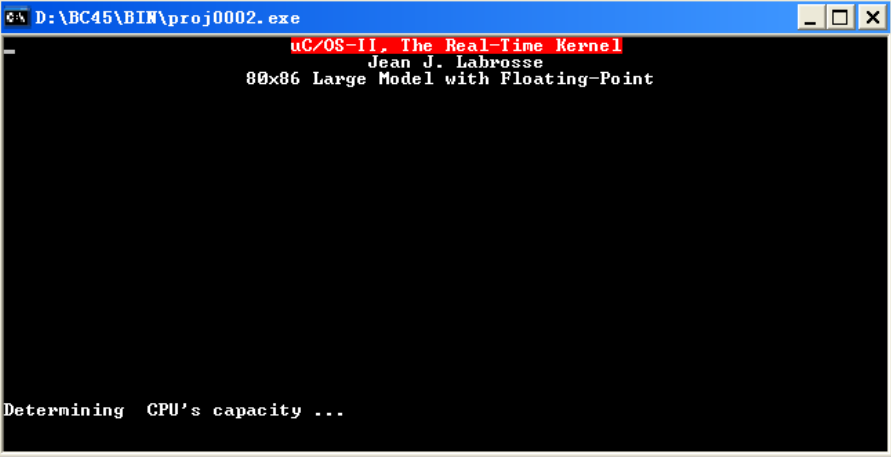


图 十六