浅析μCOS/II v2.85 内核 OSFlagPend()和 OSFlagPost()函数工作原理

文章来源:http://gliethttp.cublog.cn[转载请声明出处]

```
//对于 flag--"事件组"的使用,可以用一个简单的例子做说明:
//  比如,我现在用迅雷下载一部 10 集的连续剧,我打算 10 集全部下载完成之后,
//才开始正式看,现在 3~10 集因为种子原因,先早下完了,现在第 1 集下到了 82%,
//第 2 集下到了 97%,因为我的计划是 10 集全部下完才开始看,而第 1 集和第 2 集
//由于网络,种子等等各种原因,迟迟不能下载完成,进而导致我的计划被悬停,不能进行,
//已下载的 8 集,也因为前 2 集没能下完,而白白等待---这就等同于 flag 事件组,
//1~10 集,每一集都是一个事件,因为我内定,10 个事件全部完成之后,才进入下一事件--"观看"
//所以及早完成自己事件的第 3~10 集,将主动把自己通过 flag 事件组函数 OSFlagPost()登记到事件组上,
//他们不关心,其他友邻事件完成否,只专注自己的事件是否完成,自己的事件一旦完成
//就登记到事件组上,最后 3~10 集,都把自己登记上去了,只剩下第 1 集和第 2 集,
//一旦某天的某个时刻,第 2 集下完了,那么第 2 集也把自己登记到事件组上,这样整个事件距离完成
//还剩下一个事件,就是第 1 集是否下载完成,只要第 1 集下载完成,那么我内定的"观看"计划
//开始启动,过了 3 分钟,由于网速提高,竟以 300k 的速度开始下载第 1 集,1 分钟之后,
//第 1 集也下载完成了,第 1 集立即调用 OSFlagPost 事件组函数,将自己登记到事件组上,
//ok,OSFlagPost()检测到所有事件已经完成,OSFlagPost()将是"我"自动进入下一事件---"观看"
//  还有一点就是关于 flag 事件组和 Sem,Mbox,Queue 的区别之处,flag 事件组不使用事件控制矩阵来
//管理被阻塞在事件上的 task 进程,flag 事件组使用 pgrp 的双向链表来挂接起所有 task,
//在 OSFlagPost()中将遍历这个链表,查找符合当前 flag 事件的 task,将该 task 从双向链表中摘下
//然后放入就绪控制矩阵中,之所以这样,是因为 flag 事件组不像 Sem,Mbox,Queue 那样具有二值性,
//即 Sem,Mbox,Queue,要么有,要么没有,flag 事件组,还要进一步判断,有的话,是什么程度的有.
//--------------------------------------------------------------------------
//1.OSFlagPend()函数
OS_FLAGS OSFlagPend(OS_FLAG_GRP *pgrp, OS_FLAGS flags, INT8U wait_type, INT16U timeout, INT8U *perr)
{
    OS_FLAG_NODE node;
    OS_FLAGS flags_rdy;
    INT8U result;
    INT8U pend_stat;
    BOOLEAN consume;
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR cpu_sr = 0;
#endif

#if OS_ARG_CHK_EN > 0
    if (perr == (INT8U *)0) {
        return ((OS_FLAGS)0);
    }
    if (pgrp == (OS_FLAG_GRP *)0) {
        *perr = OS_ERR_FLAG_INVALID_PGRP;
        return ((OS_FLAGS)0);
    }
#endif
    if (OSIntNesting > 0) {
```

```
//ISR 中,不能使用 OSFlagPend()
            *perr = OS_ERR_PEND_ISR;
            return ((OS_FLAGS)0);
        }
        if (OSLockNesting > 0) {
// μ COS/II v2.85 内核已经被强制锁住
            *perr = OS_ERR_PEND_LOCKED;
            return ((OS_FLAGS)0);
        }
        if (pgrp->OSFlagType != OS_EVENT_TYPE_ FLAG) {
//确保该 event 控制块是 flag 类型
            *perr = OS_ERR_EVENT_TYPE;
            return ((OS_FLAGS)0);
        }
        result = (INT8U)(wait_type & OS_FLAG_CONSUME);
        if (result != (INT8U)0) {
//收到指定事件们之后,复位 flag 事件组,将相应的事件标志清 0
            wait_type &= ~(INT8U)OS_FLAG_CONSU ME;
            consume = OS_TRUE;
        } else {
            consume = OS_FALSE;
        }
        OS_ENTER_CRITICAL();
        switch (wait_type) {
            case OS_FLAG_WAIT_SET_ALL:
                //2007-09-09 gliethttp
                //flag 事件组中所有事件都置位才唤醒
                 flags_rdy = (OS_FLAGS)(pgrp->OSFlagFlags & flags);
                 if (flags_rdy == flags) {
                //flag 事件组中指定的所有事件都已经登记了
                    if (consume == OS_TRUE) {
                //清除 flag 事件组中的相应事件标志位
                        pgrp->OSFlagFlags &= ~flags_rdy;
                    }
                    OSTCBCur->OSTCBFlagsRdy = flags_rdy;//返回成功的 flag 事件组值
                    OS_EXIT_CRITICAL();
                    *perr = OS_ERR_NONE;
                    return (flags_rdy);
                 } else {
//flag 事件组中指定的所有事件中,可能有 1 个还没有完成登记工作,所以本 task 悬停在 flag 事件控制矩阵中
                //2007-09-09 gliethttp
                //node 为该 task 在栈空间上分配的数据,因为本 task 任务需要悬停,所以
                //位于该 task 栈空间上的 node,不会被破坏,它和全局变量性质等同
                    OS_FlagBlock(pgrp, &node, flags, wait_type, timeout);
                    OS_EXIT_CRITICAL();
                 }
                 break;
            case OS_FLAG_WAIT_SET_ANY:
```

```
                //2007-09-09 gliethttp
                //flag 事件组中指定的事件们,只要有一个事件发生置位就唤醒
                  flags_rdy = (OS_FLAGS)(pgrp->OSFlagFlags & flags);
                  if (flags_rdy != (OS_FLAGS)0) {
                        if (consume == OS_TRUE) {
                //清除 flag 事件组中的相应事件标志位
                              pgrp->OSFlagFlags &= ~flags_rdy;
                        }
                        OSTCBCur->OSTCBFlagsRdy = flags_rdy;//返回成功的 flag 事件组值
                        OS_EXIT_CRITICAL();
                        *perr = OS_ERR_NONE;
                        return (flags_rdy);
                  } else {
//flag 事件组中指定的所有事件中,没有 1 个进行登记,所以悬停本 task 在 flag 事件控制矩阵中
                //2007-09-09 gliethttp
                //node 为该 task 在栈空间上分配的数据,因为本 task 任务需要悬停,所以
                //位于该 task 栈空间上的 node,不会被破坏,它和全局变量性质等同
                        OS_FlagBlock(pgrp, &node, flags, wait_type, timeout);
                        OS_EXIT_CRITICAL();
                  }
                  break;
#if OS_FLAG_WAIT_CLR_EN > 0
            case OS_FLAG_WAIT_CLR_ALL:
                //2007-09-09 gliethttp
                //flag 事件组中所有事件都清 0 才唤醒
                  flags_rdy = (OS_FLAGS)(~pgrp->OSFlagFlags & flags);
                  if (flags_rdy == flags) {
                //flag 事件组中指定的所有事件都已经把事件自己对应的位清 0
                        if (consume == OS_TRUE) {
                //还原 flag 事件组中的相应事件标志位
                              pgrp->OSFlagFlags |= flags_rdy;
                        }
                        OSTCBCur->OSTCBFlagsRdy = flags_rdy;//返回成功的 flag 事件组值
                        OS_EXIT_CRITICAL();
                        *perr = OS_ERR_NONE;
                        return (flags_rdy);
                  } else {
//flag 事件组中指定的所有事件中,可能有 1 个还没有完成清 0 工作,所以本 task 悬停在 flag 事件控制矩阵中
                //2007-09-09 gliethttp
                //node 为该 task 在栈空间上分配的数据,因为本 task 任务需要悬停,所以
                //位于该 task 栈空间上的 node,不会被破坏,它和全局变量性质等同
                        OS_FlagBlock(pgrp, &node, flags, wait_type, timeout);
                        OS_EXIT_CRITICAL();
                  }
                  break;
            case OS_FLAG_WAIT_CLR_ANY:
                //2007-09-09 gliethttp
                //flag 事件组中指定的事件们,只要有一个事件发生清 0 就唤醒
```

```
                flags_rdy = (OS_FLAGS)(~pgrp->OSFlagFlags & flags);
                if (flags_rdy != (OS_FLAGS)0) {
                    if (consume == OS_TRUE) {
                //还原 flag 事件组中的相应事件标志位
                        pgrp->OSFlagFlags |= flags_rdy;
                    }
                    OSTCBCur->OSTCBFlagsRdy = flags_rdy;//返回成功的 flag 事件组值
                    OS_EXIT_CRITICAL();
                    *perr = OS_ERR_NONE;
                    return (flags_rdy);
                } else {
//flag 事件组中指定的所有事件中,没有 1 个发生清 0 操作,所以悬停本 task 在 flag 事件控制矩阵中
                //2007-09-09 gliethttp
                //node 为该 task 在栈空间上分配的数据,因为本 task 任务需要悬停,所以
                //位于该 task 栈空间上的 node,不会被破坏,它和全局变量性质等同
                    OS_FlagBlock(pgrp, &node, flags, wait_type, timeout);
                    OS_EXIT_CRITICAL();
                }
                break;
#endif
            default:
                OS_EXIT_CRITICAL();
                flags_rdy = (OS_FLAGS)0;
                *perr = OS_ERR_FLAG_WAIT_TYPE;
                return (flags_rdy);
    }
//因为本 task 正在运行,所以本 task 现在的优先级最高,现在本 task 已经将自己从就绪控制矩阵--调度器(x,y)矩形阵列中
//把自己摘掉,所以调度函数 OS_Sched()一定会切换到另一个 task 中执行新 task 的代码[gliethttp]
    OS_Sched();//具体参见《浅析 μC/OS-II v2.85 内核调度函数》
//2007-09-09 gliethttp
//可能因为 OSFlagPend()中指定的 timeout 已经超时
//[由 OSTimeTick()函数把本 task 重新置入了就绪控制矩阵,具体参考《浅析 μC/OS-II v2.85 内核 OSTimeDly()函数工作原理》],
//又或者确实在应用程序的调用了 OSFlagPost(),最终使得 flag 事件组条件满足,
//以下代码将具体解析是有什么引起的:1.超时,2.收到正常信号
    OS_ENTER_CRITICAL();
    if (OSTCBCur->OSTCBStatPend != OS_STAT_PEND_OK) {
//是因为 timeout 超时,使得本 task 获得重新执行的机会
        pend_stat = OSTCBCur->OSTCBStatPend;
        OSTCBCur->OSTCBStatPend = OS_STAT_PEND_OK;
//OS_FlagUnlink()把分配在本 task 栈空间上的局部变量 node,从 pgrp 事件组双向链表上摘下来.
        OS_FlagUnlink(&node);
        OSTCBCur->OSTCBStat = OS_STAT_RDY;//本 task 正在运行,不悬停在任何事件控制矩阵上
        OS_EXIT_CRITICAL();
        flags_rdy = (OS_FLAGS)0;
        switch (pend_stat) {
            case OS_STAT_PEND_TO:
            default:
                *perr = OS_ERR_TIMEOUT;//因为超时,本 task 才被调度
```

```
                    break;
                case OS_STAT_PEND_ABORT:
                    *perr = OS_ERR_PEND_ABORT;// 人为取消
                    break;
            }
            return (flags_rdy);
        }
//由于每个事件都调用 OSFlagPost()登记了事件自己,所以条件满足,本 task 被正常唤醒
//已经将本 task 在 pgrp 事件组双向链表上摘下来,并且把本 task 放入了就绪控制矩阵中,
//否则本 task 也不会执行至此.[gliethttp]
        flags_rdy = OSTCBCur->OSTCBFlagsRdy;
        if (consume == OS_TRUE) {
            switch (wait_type) {
                case OS_FLAG_WAIT_SET_ALL:
                case OS_FLAG_WAIT_SET_ANY:
                //清除 flag 事件组中的相应事件标志位
                    pgrp->OSFlagFlags &= ~flags_rdy;
                    break;
#if OS_FLAG_WAIT_CLR_EN > 0
                case OS_FLAG_WAIT_CLR_ALL:
                case OS_FLAG_WAIT_CLR_ANY:
                //还原 flag 事件组中的相应事件标志位
                    pgrp->OSFlagFlags |= flags_rdy;
                    break;
#endif
                default:
                    OS_EXIT_CRITICAL();
                    *perr = OS_ERR_FLAG_WAIT_TYPE;
                    return ((OS_FLAGS)0);
            }
        }
        OS_EXIT_CRITICAL();
        *perr = OS_ERR_NONE;
        return (flags_rdy);
}
//--------------------------------------------------------------------
//2.OS_FlagBlock()函数
static void OS_FlagBlock(OS_FLAG_GRP *pgrp, OS_FLAG_NODE *pnode, OS_FLAGS flags, INT8U wait_type, INT16U timeout)
{
    OS_FLAG_NODE *pnode_next;
    INT8U y;
//pnode 指向本 task 在自己栈空间上分配的一个局部变量
//一个 node 描述一个 task
    OSTCBCur->OSTCBStat |= OS_STAT_FLAG;// 是 Flag 事件让本 task 进入悬停等待的
    OSTCBCur->OSTCBStatPend = OS_STAT_PEND_OK;//假定不是超时,为正常收到信号
//超时,如果 timeout=0,那么,本 task 将一直悬停,仅仅当收到事件触发信号后才重新进入调度队列
    OSTCBCur->OSTCBDly = timeout;
```

```
#if OS_TASK_DEL_EN > 0
    OSTCBCur->OSTCBFlagNode = pnode;
#endif
//一个 node 描述一个 task
    pnode->OSFlagNodeFlags = flags;//该 task 对应的 flag 事件组值
    pnode->OSFlagNodeWaitType = wait_type;//该 task 对应的等待类型
    pnode->OSFlagNodeTCB = (void *)OSTCBCur;//该 task 的 TCB 任务上下文指针
    pnode->OSFlagNodeNext = pgrp->OSFlagWaitList;//把该 node 挂到 pgrp->OSFlagWaitList 头部
    pnode->OSFlagNodePrev = (void *)0;//因为是头部,所以没有 prev
    pnode->OSFlagNodeFlagGrp = (void *)pgrp;//该 task 对应的 pgrp 管理组
    pnode_next = (OS_FLAG_NODE *)pgrp->OSFlagWaitList;
    if (pnode_next != (void *)0) {
    //在本 task 之前,已经有其他 task 悬停在 flag 事件组上了
        pnode_next->OSFlagNodePrev = pnode;
    }
    pgrp->OSFlagWaitList = (void *)pnode;//设置本 task 为链表头部
    //把本 task 从就绪控制矩阵中摘下[gliethttp]
    y = OSTCBCur->OSTCBY;
    OSRdyTbl[y] &= ~OSTCBCur->OSTCBBitX;
    if (OSRdyTbl[y] == 0x00) {
    //当前 y 行对应的 8 个或 16 个 task 都已经悬停,那么当前 y 行也清除.
        OSRdyGrp &= ~OSTCBCur->OSTCBBitY;
    }
}
//--------------------------------------------------------------------
//3.OS_FlagUnlink()函数
void OS_FlagUnlink (OS_FLAG_NODE *pnode)
{
#if OS_TASK_DEL_EN > 0
    OS_TCB *ptcb;
#endif
    OS_FLAG_GRP *pgrp;
    OS_FLAG_NODE *pnode_prev;
    OS_FLAG_NODE *pnode_next;
//把管理本 task 的 node 从 pgrp 双向链表中摘下来
    pnode_prev = (OS_FLAG_NODE *)pnode->OSFlagNodePrev;
    pnode_next = (OS_FLAG_NODE *)pnode->OSFlagNodeNext;
    if (pnode_prev == (OS_FLAG_NODE *)0) {
    //说明本 node 为双向链表头
        pgrp = (OS_FLAG_GRP *)pnode->OSFlagNodeFlagGrp;
        pgrp->OSFlagWaitList = (void *)pnode_next;//设置本 task 的下一个 task 作为链表头
        if (pnode_next != (OS_FLAG_NODE *)0) {
        //如果下一个 task 存在,那么将下一个 task 的 node 的 prev 设置成 0,
        //进而来表征下一个 task 是链表头
            pnode_next->OSFlagNodePrev = (OS_FLAG_NODE *)0;
        }
    } else {
    //说明本 node 为双向链表中普通一员
```

```
            pnode_prev->OSFlagNodeNext = pnode_next;//直接跳过本 node 的链接
            if (pnode_next != (OS_FLAG_NODE *)0) {
        //下一个 task 存在,那么完成双向链表的 prev 项
                pnode_next->OSFlagNodePrev = pnode_prev;
            }
        }
#if OS_TASK_DEL_EN > 0
        ptcb = (OS_TCB *)pnode->OSFlagNodeTCB;
        ptcb->OSTCBFlagNode = (OS_FLAG_NODE *)0;
#endif
}
//--------------------------------------------------------------------
//4.OSFlagPost()函数
OS_FLAGS OSFlagPost(OS_FLAG_GRP *pgrp, OS_FLAGS flags, INT8U opt, INT8U *perr)
{
        OS_FLAG_NODE *pnode;
        BOOLEAN sched;
        OS_FLAGS flags_cur;
        OS_FLAGS flags_rdy;
        BOOLEAN rdy;
#if OS_CRITICAL_METHOD == 3
        OS_CPU_SR cpu_sr = 0;
#endif

#if OS_ARG_CHK_EN > 0
        if (perr == (INT8U *)0) {
            return ((OS_FLAGS)0);
        }
        if (pgrp == (OS_FLAG_GRP *)0) {
            *perr = OS_ERR_FLAG_INVALID_PGRP;
            return ((OS_FLAGS)0);
        }
#endif
        if (pgrp->OSFlagType != OS_EVENT_TYPE_FLAG) {
            *perr = OS_ERR_EVENT_TYPE;
            return ((OS_FLAGS)0);
        }

        OS_ENTER_CRITICAL();
        //对 flag 事件组,进行位操作
        switch (opt) {
            case OS_FLAG_CLR:
                pgrp->OSFlagFlags &= ~flags;//清除 flag 标志组 pgrp->OSFlagFlags 中 flags 位为 1 的位
                break;
            case OS_FLAG_SET:
                pgrp->OSFlagFlags |= flags;//置位 flag 标志组 pgrp->OSFlagFlags 中 flags 位为 1 的位
                break;
            default://没有该操作,直接 error 返回
```

```
                OS_EXIT_CRITICAL();
                *perr = OS_ERR_FLAG_INVALID_OPT;
                return ((OS_FLAGS)0);
    }
    sched = OS_FALSE;
    pnode = (OS_FLAG_NODE *)pgrp->OSFlagWaitList;
//2007-09-10 gliethttp
//遍历悬停在 pgrp->OSFlagWaitList 双向链表上的所有 task,唤醒满足 flag 事件组条件者
    while (pnode != (OS_FLAG_NODE *)0) {// 双向链表中还有 task 没有运算
        switch (pnode->OSFlagNodeWaitType) {
//该 node 管理的 task 的等待类型
            case OS_FLAG_WAIT_SET_ALL:
                //flag 事件组中所有事件都置位才唤醒
                flags_rdy = (OS_FLAGS)(pgrp->OSFlagFlags & pnode->OSFlagNodeFlags);
                if (flags_rdy == pnode->OSFlagNodeFlags) {
                //flag 事件组中指定的所有事件都已经登记了,那么将本 task 的 node 从 flag 事件组 pgrp 双向链表中
                //摘下来,如果本 task 仅仅在等待 flag 事件组的发生,那么将本 task 添加到就绪控制矩阵中,
                //等待 os 的调度
                    rdy = OS_FlagTaskRdy(pnode, flags_rdy);
                    if (rdy == OS_TRUE) {
                        sched = OS_TRUE;// 本 task 被添加到了就绪控制矩阵中,为了 rtos 要求,需要调度
                    }
                }
                break;
            case OS_FLAG_WAIT_SET_ANY:
                //flag 事件组中指定的事件们,只要有一个事件发生置位就唤醒
                flags_rdy = (OS_FLAGS)(pgrp->OSFlagFlags & pnode->OSFlagNodeFlags);
                if (flags_rdy != (OS_FLAGS)0) {
                //flag 事件组中指定的所有事件都已经登记了,那么将本 task 的 node 从 flag 事件组 pgrp 双向链表中
                //摘下来,如果本 task 仅仅在等待 flag 事件组的发生,那么将本 task 添加到就绪控制矩阵中,
                //等待 os 的调度
                    rdy = OS_FlagTaskRdy(pnode, flags_rdy);
                    if (rdy == OS_TRUE) {
                        sched = OS_TRUE;// 本 task 被添加到了就绪控制矩阵中,为了 rtos 要求,需要调度
                    }
                }
                break;
#if OS_FLAG_WAIT_CLR_EN > 0
            case OS_FLAG_WAIT_CLR_ALL:
                //flag 事件组中所有事件都清 0 才唤醒
                flags_rdy = (OS_FLAGS)(~pgrp->OSFlagFlags & pnode->OSFlagNodeFlags);
                if (flags_rdy == pnode->OSFlagNodeFlags) {
                //flag 事件组中指定的所有事件都已经登记了,那么将本 task 的 node 从 flag 事件组 pgrp 双向链表中
                //摘下来,如果本 task 仅仅在等待 flag 事件组的发生,那么将本 task 添加到就绪控制矩阵中,
                //等待 os 的调度
                    rdy = OS_FlagTaskRdy(pnode, flags_rdy);
                    if (rdy == OS_TRUE) {
                        sched = OS_TRUE;// 本 task 被添加到了就绪控制矩阵中,为了 rtos 要求,需要调度
```

```
                    }
                }
                break;
            case OS_FLAG_WAIT_CLR_ANY:
                //flag 事件组中指定的事件们,只要有一个事件发生清 0 就唤醒
                flags_rdy = (OS_FLAGS)(~pgrp->OSFlagFlags & pnode->OSFlagNodeFlags);
                if (flags_rdy != (OS_FLAGS)0) {
                //flag 事件组中指定的所有事件都已经登记了,那么将本 task 的 node 从 flag 事件组 pgrp 双向链表中
                //摘下来,如果本 task 仅仅在等待 flag 事件组的发生,那么将本 task 添加到就绪控制矩阵中,
                //等待 os 的调度
                    rdy = OS_FlagTaskRdy(pnode, flags_rdy);
                    if (rdy == OS_TRUE) {
                        sched = OS_TRUE;// 本 task 被添加到了就绪控制矩阵中,为了 rtos 要求,需要调度
                    }
                }
                break;
#endif
            default:
                OS_EXIT_CRITICAL();
                *perr = OS_ERR_FLAG_WAIT_TYPE;
                return ((OS_FLAGS)0);
        }
        pnode = (OS_FLAG_NODE *)pnode->OSFlagNodeNext;//下一个悬停在该 flag 事件组上的 task
    }
    OS_EXIT_CRITICAL();
    if (sched == OS_TRUE) {
    //可能刚刚放到就绪控制矩阵上的被唤醒的 task-A 的优先级比调用 OSFlagPost()函数的进程 B 优先级高
    //所以需要调用 shedule 函数,
    //如果真的高,那么调用 OSFlagPost()函数的进程 B 就要被抢占,os 将会切换到新的 task 去执行[gliethttp]
    //如果没有调用 OSFlagPost()函数的进程 B 优先级高,那么 os 不会切换,仍然继续执行进程 B,OSFlagPost()正常返回
        OS_Sched();
    }
    OS_ENTER_CRITICAL();
//返回当前的 OSFlagFlags 数值,如果因为 OS_Sched()调度去执行了 A 进程,那么这里的 OSFlagFlags
//数值可能已经被 A 进程的 consume 属性复了位.[gliethttp]
    flags_cur = pgrp->OSFlagFlags;
    OS_EXIT_CRITICAL();
    *perr = OS_ERR_NONE;
    return (flags_cur);
}
//--------------------------------------------------------------------
//5.OS_FlagTaskRdy()函数
static BOOLEAN OS_FlagTaskRdy (OS_FLAG_NODE *pnode, OS_FLAGS flags_rdy)
{
    OS_TCB *ptcb;
    BOOLEAN sched;

    ptcb = (OS_TCB *)pnode->OSFlagNodeTCB;
```

```
        ptcb->OSTCBDly = 0;//复原为正常
        ptcb->OSTCBFlagsRdy = flags_rdy;
//本 task 悬停的 flag 事件组已经发生,清除 task 上下文控制块上的 OS_STAT_FLAG 位
        ptcb->OSTCBStat &= ~(INT8U)OS_STAT_FLAG;
        ptcb->OSTCBStatPend = OS_STAT_PEND_OK;// 正常收到信号
        if (ptcb->OSTCBStat == OS_STAT_RDY) {
//如果当前 task 只是等待该 flag 事件组,那么把该 task 放到就绪控制矩阵中,允许内核调度本 task
            OSRdyGrp |= ptcb->OSTCBBitY;
            OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
            sched = OS_TRUE;
        } else {
            sched = OS_FALSE;
        }
//OS_FlagUnlink()把分配在本 task 栈空间上的局部变量 node,从 pgrp 事件组双向链表上摘下来,
//进而清除 pgrp 事件组中无本 task 的相关链接
        OS_FlagUnlink(pnode);
        return (sched);
}
```

浅析μCOS/II v2.85 内核 OSMboxPend()和 OSMboxPost()函数工作原理

文章来源:http://gliethttp.cublog.cn[转载请声明出处]

```
//-------------------------------------------------------------------------
//1.OSMboxPend()函数
void *OSMboxPend (OS_EVENT *pevent, INT16U timeout, INT8U *perr)
{
        void *pmsg;
        INT8U pend_stat;
#if OS_CRITICAL_METHOD == 3
        OS_CPU_SR cpu_sr = 0;
#endif

#if OS_ARG_CHK_EN > 0
        if (perr == (INT8U *)0) {
            return ((void *)0);
        }
        if (pevent == (OS_EVENT *)0) {
            *perr = OS_ERR_PEVENT_NULL;
            return ((void *)0);
        }
#endif
        if (pevent->OSEventType != OS_EVENT_TYPE_MBOX) {
//确保该 event 控制块是 Mbox 类型
            *perr = OS_ERR_EVENT_TYPE;
            return ((void *)0);
        }
```

```
    if (OSIntNesting > 0) {
//ISR 中,不能使用 OSMboxPend()
        *perr = OS_ERR_PEND_ISR;
        return ((void *)0);
    }
    if (OSLockNesting > 0) {
// μ COS/II v2.85 内核已经被强制锁住
        *perr = OS_ERR_PEND_LOCKED;
        return ((void *)0);
    }
//非法的统统不是,信号正常,所以有必要进一步处理
    OS_ENTER_CRITICAL();
    pmsg = pevent->OSEventPtr;
    if (pmsg != (void *)0) {
//pevent->OSEventPtr 存放对应的消息指针,如果为 0,说明还没有消息[gliethttp]
//程序的其他地方已经触发了事件
//所以该 task 无需悬停,直接获得事件的使用权
        pevent->OSEventPtr = (void *)0;
        OS_EXIT_CRITICAL();
        *perr = OS_ERR_NONE;
        return (pmsg);
    }
//当前还没有任何事件发生,所以本 task 需要悬停,让出 cpu[gliethttp]
    OSTCBCur->OSTCBStat |= OS_STAT_MBOX;// 是 Mbox 事件让本 task 进入悬停等待的
    OSTCBCur->OSTCBStatPend = OS_STAT_PEND_OK;// 假定不是超时,为正常收到信号
//超时,如果 timeout=0,那么,本 task 将一直悬停,仅仅当收到事件触发信号后才重新进入调度队列
    OSTCBCur->OSTCBDly = timeout;
//OS_EventTaskWait()函数实现的功能:
//把本 task 从就绪控制矩阵中摘下,放到 pevent 事件专有的进程事件控制矩阵表中.
    OS_EventTaskWait(pevent);
    OS_EXIT_CRITICAL();
//因为本 task 正在运行,所以本 task 现在的优先级最高,现在本 task 已经将自己从就绪控制矩阵--调度器(x,y)矩形阵列中
//把自己摘掉,所以调度函数 OS_Sched()一定会切换到另一个 task 中执行新 task 的代码[gliethttp]
    OS_Sched();//具体参见《浅析 μ C/OS-II v2.85 内核调度函数》
    OS_ENTER_CRITICAL();
//2007-09-09 gliethttp
//可能因为 OSMboxPend()中指定的 timeout 已经超时
//[由 OSTimeTick()函数把本 task 重新置入了就绪队列,具体参考《浅析 μ C/OS-II v2.85 内核 OSTimeDly()函数工作原理》],
//又或者确实在应用程序的某个地方调用了 OSMboxPost(),以下代码将具体解析是有什么引起的:1.超时,2.收到正常信号
    if (OSTCBCur->OSTCBStatPend != OS_STAT_PEND_OK) {
//是因为 timeout 超时,使得本 task 获得重新执行的机会
        pend_stat = OSTCBCur->OSTCBStatPend;
//清除 event 事件块上本 task 的标志
        OS_EventTOAbort(pevent);
        OS_EXIT_CRITICAL();
        switch (pend_stat) {
            case OS_STAT_PEND_TO:
            default:
```

```
                    *perr = OS_ERR_TIMEOUT;
                    break;
                case OS_STAT_PEND_ABORT:
                    *perr = OS_ERR_PEND_ABORT;
                    break;
            }
            return ((void *)0);
        }
//由 OSMboxPost()抛出正常事件,唤醒了本 task,因为在 OSMboxPost()时,
//已经将本 task 在 event 事件控制矩阵上的对应位清除掉,并且把本 task 放入了就绪控制矩阵中,
//否则本 task 也不会执行至此.
        pmsg = OSTCBCur->OSTCBMsg;//由 OSMboxPost()传递来的消息指针
        OSTCBCur->OSTCBMsg = (void *)0;//清空消息指针
//状态 ok,等待 os 调度登记到就绪控制矩阵中的自己
        OSTCBCur->OSTCBStat = OS_STAT_RDY;
        OSTCBCur->OSTCBEventPtr = (OS_EVENT *)0;//现在本 task 不悬停在任何 event 事件上
        OS_EXIT_CRITICAL();
        *perr = OS_ERR_NONE;
        return (pmsg);
}
//--------------------------------------------------------------------
//2.OS_EventTaskWait()函数
void OS_EventTaskWait (OS_EVENT *pevent)
{
        INT8U y;
//2007-09-09 gliethttp
//pevent 为此次 task 挂起的 EventPtr 单元
        OSTCBCur->OSTCBEventPtr = pevent;
//清除调度器中该 task 对应的标志位
        y = OSTCBCur->OSTCBY;
        OSRdyTbl[y] &= ~OSTCBCur->OSTCBBitX;
        if (OSRdyTbl[y] == 0) {
//当前 y 行对应的 8 个或 16 个 task 都已经悬停,那么当前 y 行也清除.
            OSRdyGrp &= ~OSTCBCur->OSTCBBitY;
        }
//2007-09-09 gliethttp
//将该 task 的 prio 添加到 pevent 事件控制矩阵中,这个矩阵的功能和 OSRdyGrp、OSRdyTbl 就绪控制矩阵没有区别
//都是用来就算出已经就绪 tasks 中的优先级最高的那个
        pevent->OSEventTbl[OSTCBCur->OSTCBY] |= OSTCBCur->OSTCBBitX;
        pevent->OSEventGrp |= OSTCBCur->OSTCBBitY;
}
//--------------------------------------------------------------------
//3.OS_EventTOAbort()函数
void OS_EventTOAbort (OS_EVENT *pevent)
{
        INT8U y;
//清除 event 事件控制矩阵上本 task 的标志,因为 OSTimeTick()函数未清除该单元
//它仅仅把本 task 放入就绪控制矩阵,使得本 task 重新获得被 OS 调度的机会而已
```

```
//具体的清除工作还要自己完成
//具体参考《浅析 μ C/OS-II v2.85 内核 OSTimeDly()函数工作原理》
    y = OSTCBCur->OSTCBY;
    pevent->OSEventTbl[y] &= ~OSTCBCur->OSTCBBitX;
    if (pevent->OSEventTbl[y] == 0x00) {
        pevent->OSEventGrp &= ~OSTCBCur->OSTCBBitY;
    }
    OSTCBCur->OSTCBStatPend = OS_STAT_PEND_OK;
    OSTCBCur->OSTCBStat = OS_STAT_RDY;
    OSTCBCur->OSTCBEventPtr = (OS_EVENT *)0;//现在本 task 不悬停在任何 event 事件上
}
//--------------------------------------------------------------------
//3.OSMboxPost()函数
INT8U OSMboxPost (OS_EVENT *pevent, void *pmsg)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR cpu_sr = 0;//方式 3 将把 cpsr 状态寄存器推入临时堆栈 cpu_sr 中,可以安全返回之前的中断状态
#endif

#if OS_ARG_CHK_EN > 0
    if (pevent == (OS_EVENT *)0) {
        return (OS_ERR_PEVENT_NULL);
    }
    if (pmsg == (void *)0) {
        return (OS_ERR_POST_NULL_PTR);
    }
#endif
    if (pevent->OSEventType != OS_EVENT_TYPE_MBOX) {
        return (OS_ERR_EVENT_TYPE);
    }
    OS_ENTER_CRITICAL();
    if (pevent->OSEventGrp != 0) {
    //2007-09-09 gliethttp
    //OS_EventTaskRdy()函数将摘掉等待在 pevent 事件控制矩阵上的 task 中优先级最高的 task
    //如果该 task 仅仅等待该 pevent 事件,那么将该 task 添加到就绪控制矩阵中
    //OSRdyGrp |= bity;
    //OSRdyTbl[y] |= bitx;这样调度程序就会根据情况调度 OS_Sched()该 task 了
        (void)OS_EventTaskRdy(pevent, pmsg, OS_STAT_MBOX, OS_STAT_PE ND_OK);
        OS_EXIT_CRITICAL();
    //可能刚刚放到就绪控制矩阵上的被唤醒的 task 的优先级比调用 OSMboxPost()函数的进程 B 优先级高
    //所以需要调用 shedule 函数,
    //如果真的高,那么调用 OSMboxPost()函数的进程 B 就要被抢占,os 将会切换到新的 task 去执行[gliethttp]
    //如果没有调用 OSMboxPost()函数的进程 B 优先级高,那么 os 不会切换,仍然继续执行进程 B,OSMboxPost()正常返回
        OS_Sched();
        return (OS_ERR_NONE);
    }
    //没有任何一个 task 悬停在本 event 事件控制矩阵上[gliethttp]
    if (pevent->OSEventPtr != (void *)0) {
```

```
            OS_EXIT_CRITICAL();
            return (OS_ERR_MBOX_FULL);// 邮箱已经满了
        }
        pevent->OSEventPtr = pmsg;//把该消息指针推到 pevent->OSEventPtr 中
        OS_EXIT_CRITICAL();
        return (OS_ERR_NONE);
}
//--------------------------------------------------------------------
//5.OS_EventTaskRdy()函数
INT8U OS_EventTaskRdy (OS_EVENT *pevent, void *pmsg, INT8U msk, INT8U pend_stat)
{
        OS_TCB *ptcb;
        INT8U x;
        INT8U y;
        INT8U prio;
#if OS_LOWEST_PRIO <= 63
        INT8U bitx;
        INT8U bity;
#else
        INT16U bitx;
        INT16U bity;
        INT16U *ptbl;
#endif

#if OS_LOWEST_PRIO <= 63
//小于 64 个 task 时,快速计算
//最有优先权的 task 位于事件控制矩阵中的第 y 行的第 x 列
        y = OSUnMapTbl[pevent->OSEventGrp];
        bity = (INT8U)(1 << y);
        x = OSUnMapTbl[pevent->OSEventTbl[y]];
        bitx = (INT8U)(1 << x);
        prio = (INT8U)((y << 3) + x);
#else
//对于 256 个 task
//最有优先权的 task 位于事件控制矩阵中的第 y 行的第 x 列
//以下的操作原理具体参见《浅析μC/OS-II v2.85 内核调度函数》
        if ((pevent->OSEventGrp & 0xFF) != 0) {
            y = OSUnMapTbl[pevent->OSEventGrp & 0xFF];
        } else {
            y = OSUnMapTbl[(pevent->OSEventGrp >> 8) & 0xFF] + 8;
        }
        bity = (INT16U)(1 << y);
        ptbl = &pevent->OSEventTbl[y];
        if ((*ptbl & 0xFF) != 0) {
            x = OSUnMapTbl[*ptbl & 0xFF];
        } else {
            x = OSUnMapTbl[(*ptbl >> 8) & 0xFF] + 8;
        }
```

```
    bitx = (INT16U)(1 << x);
    prio = (INT8U)((y << 4) + x);//该 task 对应的 prio 优先级值
//ok,等待在 event 事件上的所有 task 中,只有在事件控制矩阵中的第 y 行的第 x 列 task
//优先级最高、最值的成为此次事件的唤醒对象[gliethttp]
#endif
//清除此 task 在 event 事件控制矩阵中的 bit 位
    pevent->OSEventTbl[y] &= ~bitx;
    if (pevent->OSEventTbl[y] == 0) {
        pevent->OSEventGrp &= ~bity;
    }
//通过 prio 优先级找到该 prio 唯一对应的 task 对应的 ptcb 进程上下文控制块
    ptcb = OSTCBPrioTbl[prio];
    ptcb->OSTCBDly = 0;//复原为正常
    ptcb->OSTCBEventPtr = (OS_EVENT *)0;//现在本 task 不悬停在任何 event 事件上
#if ((OS_Q_EN > 0) && (OS_MAX_QS > 0)) || (OS_MBOX_EN > 0)
    ptcb->OSTCBMsg = pmsg;//传递消息指针
#else
    pmsg = pmsg;
#endif
    ptcb->OSTCBStatPend = pend_stat;//悬停状态值
    ptcb->OSTCBStat &= ~msk;//该 msk 事件已经发生,清除 task 上下文控制块上的 msk 位,如:OS_STAT_MBOX
    if (ptcb->OSTCBStat == OS_STAT_RDY) {
    //如果当前 task 只是等待该事件,那么把该 task 放到就绪控制矩阵中,允许内核调度本 task
        OSRdyGrp |= bity;
        OSRdyTbl[y] |= bitx;
    }
    return (prio);//返回本 task 对应的优先级值
}
//
```

浅析μCOSII v2.85 内核 OSMutexPend()和 OSMutexPost()函数工作原理

文章来源:http://gliethttp.cublog.cn[转载请声明出处]

```
//优先级翻转问题发生在>=3 个进程同时访问一个共享的情况下,
//所以至少有 3 个进程才有可能发生优先级翻转,即 A>B>C 时,B 才可能将 A 翻转.
//μCOS/II v2.85 内核采用"变相置顶的方式"来解决优先级翻转问题,

//"优先级继承方式"是指一个较高优先级的任务申请某信号量,但此信号量已被一个较低优先级的任务占有,这时,抬升较低优先级任务的优先级到较高优先级任务的优先级,这是一个自动过程.
//"置顶方式"是指一旦有进程访问互斥资源,立即把该进程的优先级提升到置顶值.

//"变相置顶的方式"是指我们根据工程应用任务需要,自己内定一个最高优先级,结合优先级继承方式,根据情况来判断当前 task 进程优先级是否需要抬升到置顶值.
//这个最高优先级可能不是 0,比如可能是 5(5 空闲,不能用于创建其他任务),原因是 0~4,之间的任务不会访问互斥空间,
//仅仅优先级>=6 的进程才会访问互斥空间,这时 5 就是所谓的 A,假如优先级为 10 的进程要访问互斥空间,
//这时因为没有任何进程访问互斥空间,所以 10 作为 pevent->OSEventCnt 的低 8 位值,被保存,之后持有
//互斥空间的操作权利,此时优先级为 12 的进程也要访问互斥空间,因为 12 处于 C 的角色,所以不会发生优先级翻转,
```

```
//12 将直接被悬停在 Mutexevent 事件控制矩阵上,之后 8 打算访问互斥资源空间,
//那么,因为 8 处于 B 的角色,这时 10 需要提升自己到这个所谓的"顶"--优先级 5,
//这样,以后所有访问互斥资源的进程都因为优先级小于 5 而直接悬停在 Mutexevent 事件控制矩阵上,
//不会出现因为 6、7、8 悬停在 Mutexevent 事件控制矩阵上,而此时 9 因为获得 cpu 执行权,
//而抢占了 10 的执行,进而发生优先级翻转现象.[gliethttp]
//但是如果真的 0~4 中的某个进程不守规矩,贸然访问了共享资源,会发生什么呢,让我们来看看:
//如 2 要访问资源,那么 2 一定是直接悬停在事件控制矩阵上,直到已经提升优先级或未提升优先级的占用互斥资源空间
//的进程退出,重新计算悬停在事件控制矩阵上的优先级最高的任务的时候 ,2 将会被加入到就绪控制矩阵中,等待 cpu
//调度,进而占用互斥资源,这好像也没有问题,那么继续进行假设,如果 4 优先级在操作互斥资源,此时 2 打算操作,
//那么 2 需要等待,被添加到了 mutex 事件控制矩阵中,这时就那么巧,3 又获得了 cpu 的使用权,那么优先级 4 将被 3 抢占
//所以出现了优先级翻转现象--3 把 2 给翻转了,所以对于优先级高于内定置顶值 pip 的进程访问互斥资源时,
//并不能受到 mutex 互斥保护机制的保护,所以对于这些进程,他们可能会发生优先级翻转,也可能不会.
//因此,这使我们更坚定一点,不要存在侥幸心里,踏踏实实的把 pip 设置成真正的"置顶"值.[gliethttp]
//----------------------------------------------------------------------
//1.OSMutexPend()函数
void OSMutexPend (OS_EVENT *pevent, INT16U timeout, INT8U *perr)
{
    INT8U pip;
    INT8U mprio;
    BOOLEAN rdy;
    OS_TCB *ptcb;
    OS_EVENT *pevent2;
    INT8U y;
    INT8U pend_stat;
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR cpu_sr = 0;
#endif


#if OS_ARG_CHK_EN > 0
    if (perr == (INT8U *)0) {
        return;
    }
    if (pevent == (OS_EVENT *)0) {
        *perr = OS_ERR_PEVENT_NULL;
        return;
    }
#endif
    if (pevent->OSEventType != OS_EVENT_TYPE_MUTEX) {
//确保该 event 控制块是 Mutex 类型
        *perr = OS_ERR_EVENT_TYPE;
        return;
    }
    if (OSIntNesting > 0) {
//ISR 中,不能使用 OSMboxPend()
        *perr = OS_ERR_PEND_ISR;
        return;
    }
```

```
    if (OSLockNesting > 0) {
// μ COS/II v2.85 内核已经被强制锁住
        *perr = OS_ERR_PEND_LOCKED;
        return;
    }
//非法的统统不是,信号正常,所以有必要进一步处理
    OS_ENTER_CRITICAL();
//在 OSMutexCreate()中
//#define OS_MUTEX_AVAILABLE (INT16U)0x00FFu
//pevent->OSEventCnt = (INT16U)((INT16U)prio << 8) | OS_MUTEX_AVAILABLE;
//prio 为一个空闲的优先级号,没有 task 使用该 prio[gliethttp]
    pip = (INT8U)(pevent->OSEventCnt >> 8);//变相置顶值 pip
    if ((INT8U)(pevent->OSEventCnt & OS_MUTEX_KEEP_LOWER_8) == OS_MUTEX_AVAILABLE) {
//当前没有其他进程使用 OSMutexPend()占有共享资源,
//所以 task 将要成为当前唯一占用共享资源的进程,pevent->OSEventCnt 的低 8 位存放
//当前正在使用共享资源 task 的 prio 优先级值,
//pip=pevent->OSEventCnt 的高 8 位存放
//"变相置顶"--所谓的最高优先级值 pip(访问互斥资源的所有进程的优先级都比该 pip 低)
//通过提升到这个所谓的顶,来解决优先级翻转问题[gliethttp]
        pevent->OSEventCnt &= OS_MUTEX_KEEP_UPPER_8;
        pevent->OSEventCnt |= OSTCBCur->OSTCBPrio;
        pevent->OSEventPtr = (void *)OSTCBCur;
        if (OSTCBCur->OSTCBPrio <= pip) {
//因为 pip 是置顶优先级,所以 pip 必须是所有能够访问互斥资源的进程中优先级最高的[gliethttp]
//如果低,那么这时 mutex 互斥机制失效,mutex 不起作用了
            OS_EXIT_CRITICAL();
            *perr = OS_ERR_PIP_LOWER;
        } else {
            OS_EXIT_CRITICAL();
            *perr = OS_ERR_NONE;
        }
        return;
    }
//2007-09-09 gliethttp
//说明已经有一个 task 占用了共享资源,持有共享互斥锁 Mutex 了,如下做进一步处理
    mprio = (INT8U)(pevent->OSEventCnt & OS_MUTEX_KEEP_LOWER_8);
    ptcb = (OS_TCB *)(pevent->OSEventPtr);
//2007-09-09 gliethttp
//[注:未提升优先级之前 ptcb->OSTCBPrio 等于 mprio,提升之后 ptcb->OSTCBPrio 等于 pip(置顶优先级)]
//因为 μ COS/II v2.85 内核采用"变相置顶的方式"来解决优先级翻转问题,所以 pip 就是 A;
//所以在工程应用上有这样一个限制,能够访问互斥空间的所有进程优先级都要低于这个 pip,
//优先级高于 pip 的进程不能访问,互斥空间,所以这也就要求,我们指定的 pip 的优先级必须保证,
//大于所有能访问互斥空间的进程,[gilethttp]
//只有低于 pip 优先级的进程才能够受"资源互斥机制"的保护【这就是 μ COS/II v2.85 内核优先级翻转的局限之处】
//但是,这在实际的工程应用中是可以接受的.
//优先级 A>B>C 时,B 可能引起优先级翻转,所以下面检测 OSTCBCur->OSTCBPrio 是否处于 B 的角色
//如果是,那么需要提升 C 的优先级,否则不会发生优先级翻转[gliethttp]
    if (ptcb->OSTCBPrio > pip) {
```

```
//说明 ptcb->OSTCBPrio 还没有提升到所谓的最高优先级 pip,
//否则 ptcb->OSTCBPrio 将等于 pip[gliethttp]
//如果 mprio < OSTCBCur->OSTCBPrio 说明,当前持有 mutex 资源的 task 处于 B 的角色
        if (mprio > OSTCBCur->OSTCBPrio) {
//这时 OSTCBCur->OSTCBPrio 处于 B 的角色,那么可能会发生优先级翻转,
//因为可能,所以程序肯定会照着最坏的情况考虑,使用"变相置顶的方式",提升当前持有互斥资源的 task 的优先级
//到达这个所谓的"顶"--pip.
            y = ptcb->OSTCBY;
            if ((OSRdyTbl[y] & ptcb->OSTCBBitX) != 0) {
//如果 A 已经在就绪控制矩阵中,那么把 A 从就绪控制矩阵中摘下[gliethttp]
//具体参考《浅析 μ COS/II v2.85 内核 OSMboxPend()和 OSMboxPost()函数工作原理》
                OSRdyTbl[y] &= ~ptcb->OSTCBBitX;
                if (OSRdyTbl[y] == 0) {
                    OSRdyGrp &= ~ptcb->OSTCBBitY;
                }
                rdy = OS_TRUE;//task 处于就绪状态
            } else {
                pevent2 = ptcb->OSTCBEventPtr;
                if (pevent2 != (OS_EVENT *)0) {
                    if ((pevent2->OSEventTbl[ptcb->OSTCBY] &= ~ptcb->OSTCBBitX) == 0) {
//如果 A 已经在事件控制矩阵中,那么把 A 从事件控制矩阵中摘下[gliethttp]
//具体参考《浅析 μ COS/II v2.85 内核 OSMboxPend()和 OSMboxPost()函数工作原理》
                        pevent2->OSEventGrp &= ~ptcb->OSTCBBitY;
                    }
                }
                rdy = OS_FALSE;//task 处于事件等待状态
            }
            ptcb->OSTCBPrio = pip;//将 task 优先级置顶到 pip
#if OS_LOWEST_PRIO <= 63
//对于 64 个 tasks 配置
//重新就绪控制矩阵和事件控制矩阵中使用到(x,y)值
            ptcb->OSTCBY = (INT8U)( ptcb->OSTCBPrio >> 3);
            ptcb->OSTCBX = (INT8U)( ptcb->OSTCBPrio & 0x07);
            ptcb->OSTCBBitY = (INT8U)(1 << ptcb->OSTCBY);
            ptcb->OSTCBBitX = (INT8U)(1 << ptcb->OSTCBX);
#else
//对于 255 个 tasks 配置
//重新计算就绪控制矩阵和事件控制矩阵中使用到(x,y)值
            ptcb->OSTCBY = (INT8U)((ptcb->OSTCBPrio >> 4) & 0xFF);
            ptcb->OSTCBX = (INT8U)( ptcb->OSTCBPrio & 0x0F);
            ptcb->OSTCBBitY = (INT16U)(1 << ptcb->OSTCBY);
            ptcb->OSTCBBitX = (INT16U)(1 << ptcb->OSTCBX);
#endif
            if (rdy == OS_TRUE) {
//提升优先级之前该 task 处于就绪控制矩阵中,等待 cpu 调度
                OSRdyGrp |= ptcb->OSTCBBitY;
                OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
            } else {
```

```
//提升优先级之前该 task 处于事件控制矩阵中,等待事件的到来
//需要说明一点,一个 task 只有两种状态,
//1.要么在就绪控制矩阵中等待被 cpu 调度
//2.要么阻塞在事件控制矩阵中[gliethttp]
                    pevent2 = ptcb->OSTCBEventPtr;
                    if (pevent2 != (OS_EVENT *)0) {
                        pevent2->OSEventGrp |= ptcb->OSTCBBitY;
                        pevent2->OSEventTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
                    }
                }
                OSTCBPrioTbl[pip] = ptcb;//添加新 pip 优先级下的 TCB
            }
        }
        OSTCBCur->OSTCBStat |= OS_STAT_MUTEX;//是 Mutex 事件让本 task 进入悬停等待的
        OSTCBCur->OSTCBStatPend = OS_STAT_PEND_OK;//假定不是超时,为正常收到信号
//超时,如果 timeout=0,那么,本 task 将一直悬停,仅仅当收到事件触发信号后才重新进入调度队列
        OSTCBCur->OSTCBDly = timeout;
//OS_EventTaskWait()函数实现的功能:
//把本 task 从就绪控制矩阵中摘下,放到 pevent 事件专有的进程事件控制矩阵表中.
        OS_EventTaskWait(pevent);
        OS_EXIT_CRITICAL();
//因为本 task 正在运行,所以本 task 现在的优先级最高的,现在本 task 已经将自己从就绪控制矩阵--调度器(x,y)矩形阵列中
//把自己摘掉,所以调度函数 OS_Sched()一定会切换到另一个 task 中执行新 task 的代码[gliethttp]
        OS_Sched();//具体参见《浅析 μC/OS-II v2.85 内核调度函数》
        OS_ENTER_CRITICAL();
//2007-09-09 gliethttp
//可能因为 OSMutexPend()中指定的 timeout 已经超时
//[由 OSTimeTick()函数把本 task 重新置入了就绪队列,具体参考《浅析 μC/OS-II v2.85 内核 OSTimeDly()函数工作原理》],
//又或者确实在应用程序的某个地方调用了 OSMutexPost(),以下代码将具体解析是有什么引起的:1.超时,2.收到正常信号
        if (OSTCBCur->OSTCBStatPend != OS_STAT_PEND_OK) {
//是因为 timeout 超时,使得本 task 获得重新执行的机会
            pend_stat = OSTCBCur->OSTCBStatPend;
//清除 event 事件块上本 task 的标志
            OS_EventTOAbort(pevent);
            OS_EXIT_CRITICAL();
            switch (pend_stat) {
                case OS_STAT_PEND_TO:
                default:
                    *perr = OS_ERR_TIMEOUT;
                    break;
                case OS_STAT_PEND_ABORT:
                    *perr = OS_ERR_PEND_ABORT;
                    break;
            }
            return;
        }
//由 OSMutexPost()抛出正常事件,唤醒了本 task,因为在 OSMutexPost()时,
//已经将本 task 在 event 事件控制矩阵上的对应位清除掉,并且把本 task 放入了就绪控制矩阵中,
```

```
//否则本 task 也不会执行至此.
    OSTCBCur->OSTCBEventPtr = (OS_EVENT *)0;
    OS_EXIT_CRITICAL();
    *perr = OS_ERR_NONE;
}
//--------------------------------------------------------------------
//2.OS_EventTaskWait()函数
void OS_EventTaskWait (OS_EVENT *pevent)
{
    INT8U y;
//2007-09-09 gliethttp
//pevent 为此次 task 挂起的 EventPtr 单元
    OSTCBCur->OSTCBEventPtr = pevent;
//清除调度器中该 task 对应的标志位
    y = OSTCBCur->OSTCBY;
    OSRdyTbl[y] &= ~OSTCBCur->OSTCBBitX;
    if (OSRdyTbl[y] == 0) {
//当前 y 行对应的 8 个或 16 个 task 都已经悬停,那么当前 y 行也清除.
        OSRdyGrp &= ~OSTCBCur->OSTCBBitY;
    }
//2007-09-09 gliethttp
//将该 task 的 prio 添加到 pevent 事件控制矩阵中,这个矩阵的功能和 OSRdyGrp、OSRdyTbl 就绪控制矩阵没有区别
//都是用来计算出已经就绪 tasks 中的优先级最高的那个
    pevent->OSEventTbl[OSTCBCur->OSTCBY] |= OSTCBCur->OSTCBBitX;
    pevent->OSEventGrp |= OSTCBCur->OSTCBBitY;
}
//--------------------------------------------------------------------
//3.OS_EventTOAbort()函数
void OS_EventTOAbort (OS_EVENT *pevent)
{
    INT8U y;
//清除 event 事件控制矩阵上本 task 的标志,因为 OSTimeTick()函数未清除该单元
//它仅仅把本 task 放入就绪控制矩阵,使得本 task 重新获得被 OS 调度的机会而已
//具体的清除工作还要自己完成
//具体参考《浅析μC/OS-II v2.85 内核 OSTimeDly()函数工作原理》
    y = OSTCBCur->OSTCBY;
    pevent->OSEventTbl[y] &= ~OSTCBCur->OSTCBBitX;
    if (pevent->OSEventTbl[y] == 0x00) {
        pevent->OSEventGrp &= ~OSTCBCur->OSTCBBitY;
    }
    OSTCBCur->OSTCBStatPend = OS_STAT_PEND_OK;
    OSTCBCur->OSTCBStat = OS_STAT_RDY;
    OSTCBCur->OSTCBEventPtr = (OS_EVENT *)0;//现在本 task 不悬停在任何 event 事件上
}
//--------------------------------------------------------------------
//4.OSMutexPost()函数
INT8U OSMutexPost (OS_EVENT *pevent)
{
```

```
    INT8U pip;
    INT8U prio;
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR cpu_sr = 0;//方式 3 将把 cpsr 状态寄存器推入临时堆栈 cpu_sr 中,可以安全返回之前的中断状态
#endif
    if (OSIntNesting > 0) {
        return (OS_ERR_POST_ISR);
    }
#if OS_ARG_CHK_EN > 0
    if (pevent == (OS_EVENT *)0) {
        return (OS_ERR_PEVENT_NULL);
    }
#endif
    if (pevent->OSEventType != OS_EVENT_TYPE_MUTEX) {
        return (OS_ERR_EVENT_TYPE);
    }
    OS_ENTER_CRITICAL();
    pip = (INT8U)(pevent->OSEventCnt >> 8);//变相置顶值 pip
    prio = (INT8U)(pevent->OSEventCnt & OS_MUTEX_KEEP_LOWER_8);// 持有 mutex 资源的 task 原始优先级
    if (OSTCBCur != (OS_TCB *)pevent->OSEventPtr) {
//因为解决了局部优先级翻转问题,所以 OSTCBCur 肯定要等于 pevent->OSEventPtr
//否则发生了不知名的异常
        OS_EXIT_CRITICAL();
        return (OS_ERR_NOT_MUTEX_OWNER);
    }
    if (OSTCBCur->OSTCBPrio == pip) {
//task 被提升了优先级到 pip 置顶值,也就是一个比该 task 优先级高
//比 pip 低的进程需要访问互斥资源,即:存在 B 角色进程,
//那么使用 OSMutex_RdyAtPrio()把本 task 从就绪控制矩阵中摘下来
//同时将自己还原到 prio 优先级
        OSMutex_RdyAtPrio(OSTCBCur, prio);
    }
    OSTCBPrioTbl[pip] = OS_TCB_RESERVED;
    if (pevent->OSEventGrp != 0) {
    //2007-09-09 gliethttp
    //OS_EventTaskRdy()函数将摘掉等待在 pevent 事件控制矩阵上的 task 中优先级最高的 task
    //如果该 task 仅仅等待该 pevent 事件,那么将该 task 添加到就绪控制矩阵中
    //OSRdyGrp |= bity;
    //OSRdyTbl[y] |= bitx;这样调度程序就会根据情况调度 OS_Sched()该 task 了
        prio = OS_EventTaskRdy(pevent, (void *)0, OS_STAT_MUTEX, OS_STAT_PEND_OK);
    //保持处于高 8 位的 pip 置顶优先级值
    //同时清除低 8 位数据
        pevent->OSEventCnt &= OS_MUTEX_KEEP_UPPER_8;
        pevent->OSEventCnt |= prio;//获得执行权的 task 充当优先级翻转计算中的 C 角色
        pevent->OSEventPtr = OSTCBPrioTbl[prio];//task 的控制块传给 OSEventPtr 指针,供优先级翻转计算使用
        if (prio <= pip) {
//2007-09-09 gliethttp
//prio 比 pip 小,那么说明比置顶值 pip 优先级还要高的进程竟然访问了共享资源,
```

```
//那么这时可能会出现优先级翻转,因为这时 mutex 机制已经不起作用,
//所以应该保证"变相置顶的方式"初始化时,自己内定的最高优先级 pip 务必大于所有能访问互斥资源的进程优先级
[gliethttp]
                OS_EXIT_CRITICAL();
                OS_Sched();//具体参见《浅析 μ C/OS-II v2.85 内核调度函数》
                return (OS_ERR_PIP_LOWER);
            } else {
                OS_EXIT_CRITICAL();
                OS_Sched();//具体参见《浅析 μ C/OS-II v2.85 内核调度函数》
                return (OS_ERR_NONE);
            }
        }
        //没有任何一个 task 悬停在本 event 事件控制矩阵上[gliethttp]
        pevent->OSEventCnt |= OS_MUTEX_AVAILABLE;//还原为初始值
        pevent->OSEventPtr = (void *)0;//现在本 task 不悬停在任何 event 事件上
        OS_EXIT_CRITICAL();
        return (OS_ERR_NONE);
}
//--------------------------------------------------------------------------
//5.OS_EventTaskRdy()函数
INT8U OS_EventTaskRdy (OS_EVENT *pevent, void *pmsg, INT8U msk, INT8U pend_stat)
{
        OS_TCB *ptcb;
        INT8U x;
        INT8U y;
        INT8U prio;
#if OS_LOWEST_PRIO <= 63
        INT8U bitx;
        INT8U bity;
#else
        INT16U bitx;
        INT16U bity;
        INT16U *ptbl;
#endif

#if OS_LOWEST_PRIO <= 63
//小于 64 个 task 时,快速计算
//最有优先权的 task 位于事件控制矩阵中的第 y 行的第 x 列
        y = OSUnMapTbl[pevent->OSEventGrp];
        bity = (INT8U)(1 << y);
        x = OSUnMapTbl[pevent->OSEventTbl[y]];
        bitx = (INT8U)(1 << x);
        prio = (INT8U)((y << 3) + x);
#else
//对于 256 个 task
//最有优先权的 task 位于事件控制矩阵中的第 y 行的第 x 列
//以下的操作原理具体参见《浅析 μ C/OS-II v2.85 内核调度函数》
        if ((pevent->OSEventGrp & 0xFF) != 0) {
```

```
        y = OSUnMapTbl[pevent->OSEventGrp & 0xFF];
    } else {
        y = OSUnMapTbl[(pevent->OSEventGrp >> 8) & 0xFF] + 8;
    }
    bity = (INT16U)(1 << y);
    ptbl = &pevent->OSEventTbl[y];
    if ((*ptbl & 0xFF) != 0) {
        x = OSUnMapTbl[*ptbl & 0xFF];
    } else {
        x = OSUnMapTbl[(*ptbl >> 8) & 0xFF] + 8;
    }
    bitx = (INT16U)(1 << x);
    prio = (INT8U)((y << 4) + x);//该 task 对应的 prio 优先级值
//ok,等待在 event 事件上的所有 task 中,只有在事件控制矩阵中的第 y 行的第 x 列 task
//优先级最高、最值的成为此次事件的唤醒对象[gliethttp]
#endif
//清除此 task 在 event 事件控制矩阵中的 bit 位
    pevent->OSEventTbl[y] &= ~bitx;
    if (pevent->OSEventTbl[y] == 0) {
        pevent->OSEventGrp &= ~bity;
    }
//通过 prio 优先级找到该 prio 唯一对应的 task 对应的 ptcb 进程上下文控制块
    ptcb = OSTCBPrioTbl[prio];
    ptcb->OSTCBDly = 0;//复原为正常
    ptcb->OSTCBEventPtr = (OS_EVENT *)0;//现在本 task 不悬停在任何 event 事件上
#if ((OS_Q_EN > 0) && (OS_MAX_QS > 0)) || (OS_MBOX_EN > 0)
    ptcb->OSTCBMsg = pmsg;//传递消息指针
#else
    pmsg = pmsg;
#endif
    ptcb->OSTCBStatPend = pend_stat;//悬停状态值
    ptcb->OSTCBStat &= ~msk;//该 msk 事件已经发生,清除 task 上下文控制块上的 msk 位,如:OS_STAT_MUTEX
    if (ptcb->OSTCBStat == OS_STAT_RDY) {
    //如果当前 task 只是等待该事件,那么把该 task 放到就绪控制矩阵中,允许内核调度本 task
        OSRdyGrp |= bity;
        OSRdyTbl[y] |= bitx;
    }
    return (prio);//返回本 task 对应的优先级值
}
```

浅析μCOSII v2.85 内核 OSQPend()和 OSQPost()函数工作原理

```
//--------------------------------------------------------------------------
//1.OSQPend()函数
void *OSQPend (OS_EVENT *pevent, INT16U timeout, INT8U *perr)
{
    void *pmsg;
    OS_Q *pq;
    INT8U pend_stat;
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR cpu_sr = 0;
#endif

#if OS_ARG_CHK_EN > 0
    if (perr == (INT8U *)0) {
        return ((void *)0);
    }
    if (pevent == (OS_EVENT *)0) {
        *perr = OS_ERR_PEVENT_NULL;
        return ((void *)0);
    }
    if (pevent->OSEventType != OS_EVENT_TYPE_Q) {
//确保该 event 控制块是 Q 类型
        *perr = OS_ERR_EVENT_TYPE;
        return ((void *)0);
    }
#endif
    if (OSIntNesting > 0) {
//ISR 中,不能使用 OSQPend()
        *perr = OS_ERR_PEND_ISR;
        return ((void *)0);
    }
    if (OSLockNesting > 0) {
// μ COS/II v2.85 内核已经被强制锁住
        *perr = OS_ERR_PEND_LOCKED;
        return ((void *)0);
    }
//非法的统统不是,信号正常,所以有必要进一步处理
    OS_ENTER_CRITICAL();
    pq = (OS_Q *)pevent->OSEventPtr;
    if (pq->OSQEntries > 0) {
//程序的其他地方已经触发了事件
//所以该 task 无需悬停,直接获得事件的使用权
        pmsg = *pq->OSQOut++;//消息采用先进先出方式
        pq->OSQEntries--;//个数-1
        if (pq->OSQOut == pq->OSQEnd) {
//在 OSQCreate(void **start, INT16U size)中,做了如下初始化:
//pq->OSQStart       = start;
//pq->OSQEnd         = &start[size];
```

```
//pq->OSQIn            = start;
//pq->OSQOut           = start;
//pq->OSQSize          = size;
//pq->OSQEntries     = 0;
                pq->OSQOut = pq->OSQStart;
            }
            OS_EXIT_CRITICAL();
            *perr = OS_ERR_NONE;
            return (pmsg);
        }
//当前还没有任何事件发生,所以本 task 需要悬停,让出 cpu[gliethttp]
        OSTCBCur->OSTCBStat |= OS_STAT_Q;//是 Q 事件让本 task 进入悬停等待的
        OSTCBCur->OSTCBStatPend = OS_STAT_PEND_OK;//假定不是超时,为正常收到信号
//超时,如果 timeout=0,那么,本 task 将一直悬停,仅仅当收到事件触发信号后才重新进入调度队列
        OSTCBCur->OSTCBDly = timeout;
//OS_EventTaskWait()函数实现的功能:
//把本 task 从就绪控制矩阵中摘下,放到 pevent 事件专有的进程事件控制矩阵表中.
        OS_EventTaskWait(pevent);
        OS_EXIT_CRITICAL();
//因为本 task 正在运行,所以本 task 现在的优先级最高,现在本 task 已经将自己从就绪控制矩阵--调度器(x,y)矩形阵列中
//把自己摘掉,所以调度函数 OS_Sched()一定会切换到另一个 task 中执行新 task 的代码[gliethttp]
        OS_Sched();//具体参见《浅析 μC/OS-II v2.85 内核调度函数》
//2007-09-09 gliethttp
//可能因为 OSQPend()中指定的 timeout 已经超时
//[由 OSTimeTick()函数把本 task 重新置入了就绪队列,具体参考《浅析 μC/OS-II v2.85 内核 OSTimeDly()函数工作原理》],
//又或者确实在应用程序的某个地方调用了 OSQPost(),以下代码将具体解析是有什么引起的:1.超时,2.收到正常信号
        OS_ENTER_CRITICAL();
        if (OSTCBCur->OSTCBStatPend != OS_STAT_PEND_OK) {
//是因为 timeout 超时,使得本 task 获得重新执行的机会
            pend_stat = OSTCBCur->OSTCBStatPend;
//清除 event 事件控制矩阵上本 task 的标志
            OS_EventTOAbort(pevent);
            OS_EXIT_CRITICAL();
            switch (pend_stat) {
                case OS_STAT_PEND_TO:
                default:
                    *perr = OS_ERR_TIMEOUT;
                    break;
                case OS_STAT_PEND_ABORT:
                    *perr = OS_ERR_PEND_ABORT;
                    break;
            }
            return ((void *)0);
        }
//由 OSQPost()抛出正常事件,唤醒了本 task,因为在 OSQPost()时,
//已经将本 task 在 event 事件控制矩阵上的对应位清除掉,并且把本 task 放入了就绪控制矩阵中,
//否则本 task 也不会执行至此.
        pmsg = OSTCBCur->OSTCBMsg;//由 OSQPost()传递来的消息指针
```

```
    OSTCBCur->OSTCBMsg = (void *)0;//清空消息指针
//状态 ok,等待 os 调度登记到就绪控制矩阵中的自己
    OSTCBCur->OSTCBStat = OS_STAT_RDY;
    OSTCBCur->OSTCBEventPtr = (OS_EVENT *)0;//现在本 task 不悬停在任何 event 事件上
    OS_EXIT_CRITICAL();
    *perr = OS_ERR_NONE;
    return (pmsg);//返回 OSQPend()到的消息指针
}
//--------------------------------------------------------------------
//2.OS_EventTaskWait()函数
void OS_EventTaskWait (OS_EVENT *pevent)
{
    INT8U y;
//2007-09-09 gliethttp
//pevent 为此次 task 挂起的 EventPtr 单元
    OSTCBCur->OSTCBEventPtr = pevent;
//清除调度器就绪控制矩阵中该 task 对应的标志位
    y = OSTCBCur->OSTCBY;
    OSRdyTbl[y] &= ~OSTCBCur->OSTCBBitX;
    if (OSRdyTbl[y] == 0) {
//当前 y 行对应的 8 个或 16 个 task 都已经悬停,那么当前 y 行也清除.
        OSRdyGrp &= ~OSTCBCur->OSTCBBitY;
    }
//2007-09-09 gliethttp
//将该 task 的 prio 添加到 pevent 事件控制矩阵中,这个矩阵的功能和 OSRdyGrp、OSRdyTbl 就绪控制矩阵没有区别
//都是用来就算出已经就绪 tasks 中的优先级最高的那个
    pevent->OSEventTbl[OSTCBCur->OSTCBY] |= OSTCBCur->OSTCBBitX;
    pevent->OSEventGrp |= OSTCBCur->OSTCBBitY;
}
//--------------------------------------------------------------------
//3.OS_EventTOAbort()函数
void OS_EventTOAbort (OS_EVENT *pevent)
{
    INT8U y;
//清除 event 事件控制矩阵上本 task 的标志,因为 OSTimeTick()函数未清除该单元
//它仅仅把本 task 放入就绪控制矩阵,使得本 task 重新获得被 OS 调度的机会而已
//具体的清除工作还要自己完成
//具体参考《浅析 μ C/OS-II v2.85 内核 OSTimeDly()函数工作原理》
    y = OSTCBCur->OSTCBY;
    pevent->OSEventTbl[y] &= ~OSTCBCur->OSTCBBitX;
    if (pevent->OSEventTbl[y] == 0x00) {
        pevent->OSEventGrp &= ~OSTCBCur->OSTCBBitY;
    }
    OSTCBCur->OSTCBStatPend = OS_STAT_PEND_OK;
    OSTCBCur->OSTCBStat = OS_STAT_RDY;
    OSTCBCur->OSTCBEventPtr = (OS_EVENT *)0;//现在本 task 不悬停在任何 event 事件上
}
//--------------------------------------------------------------------
```

```
//4.OSQPost()函数
INT8U OSQPost (OS_EVENT *pevent, void *pmsg)
{
    OS_Q *pq;
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR cpu_sr = 0;//方式 3 将把 cpsr 状态寄存器推入临时堆栈 cpu_sr 中,可以安全返回之前的中断状态
#endif

#if OS_ARG_CHK_EN > 0
    if (pevent == (OS_EVENT *)0) {
        return (OS_ERR_PEVENT_NULL);
    }
#endif
    if (pevent->OSEventType != OS_EVENT_TYPE_Q) {
        return (OS_ERR_EVENT_TYPE);
    }
    OS_ENTER_CRITICAL();
    if (pevent->OSEventGrp != 0) {
    //2007-09-09 gliethttp
    //OS_EventTaskRdy()函数将摘掉等待在 pevent 事件控制矩阵上的 task 中优先级最高的 task
    //如果该 task 仅仅等待该 pevent 事件,那么将该 task 添加到就绪控制矩阵中
    //OSRdyGrp |= bity;
    //OSRdyTbl[y] |= bitx;这样调度程序就会根据情况调度 OS_Sched()该 task 了
        (void)OS_EventTaskRdy(pevent, pmsg, OS_STAT_Q, OS_STAT_PEND_OK);
        OS_EXIT_CRITICAL();
    //可能刚刚放到就绪控制矩阵上的被唤醒的 task 的优先级比调用 OSQPost()函数的进程 B 优先级高
    //所以需要调用 shedule 函数,
    //如果真的高,那么调用 OSQPost()函数的进程 B 就要被抢占,os 将会切换到新的 task 去执行[gliethttp]
    //如果没有调用 OSQPost()函数的进程 B 优先级高,那么 os 不会切换,仍然继续执行进程 B,OSQPost()正常返回
        OS_Sched();
        return (OS_ERR_NONE);
    }
    //没有任何一个 task 悬停在本 event 事件控制矩阵上,
    //那么将此消息入队,进而堆积消息,用来缓冲消息数据[gliethttp]
//在 OSQCreate(void **start, INT16U size)中,做了如下初始化:
//pq->OSQStart      = start;
//pq->OSQEnd        = &start[size];
//pq->OSQIn         = start;
//pq->OSQOut        = start;
//pq->OSQSize       = size;
//pq->OSQEntries    = 0;
    pq = (OS_Q *)pevent->OSEventPtr;
    if (pq->OSQEntries >= pq->OSQSize) {
        OS_EXIT_CRITICAL();//Queue 满了
        return (OS_ERR_Q_FULL);
    }
    *pq->OSQIn++ = pmsg;//将 pmsg 推入消息队列
    pq->OSQEntries++;
```

```
    if (pq->OSQIn == pq->OSQEnd) {
        pq->OSQIn = pq->OSQStart;
    }
    OS_EXIT_CRITICAL();
    return (OS_ERR_NONE);
}
//--------------------------------------------------------------------
//5.OS_EventTaskRdy()函数
INT8U OS_EventTaskRdy (OS_EVENT *pevent, void *pmsg, INT8U msk, INT8U pend_stat)
{
    OS_TCB *ptcb;
    INT8U x;
    INT8U y;
    INT8U prio;
#if OS_LOWEST_PRIO <= 63
    INT8U bitx;
    INT8U bity;
#else
    INT16U bitx;
    INT16U bity;
    INT16U *ptbl;
#endif

#if OS_LOWEST_PRIO <= 63
//小于 64 个 task 时,快速计算
//最有优先权的 task 位于事件控制矩阵中的第 y 行的第 x 列
    y = OSUnMapTbl[pevent->OSEventGrp];
    bity = (INT8U)(1 << y);
    x = OSUnMapTbl[pevent->OSEventTbl[y]];
    bitx = (INT8U)(1 << x);
    prio = (INT8U)((y << 3) + x);
#else
//对于 256 个 task
//最有优先权的 task 位于事件控制矩阵中的第 y 行的第 x 列
//以下的操作原理具体参见《浅析μC/OS-II v2.85 内核调度函数》
    if ((pevent->OSEventGrp & 0xFF) != 0) {
        y = OSUnMapTbl[pevent->OSEventGrp & 0xFF];
    } else {
        y = OSUnMapTbl[(pevent->OSEventGrp >> 8) & 0xFF] + 8;
    }
    bity = (INT16U)(1 << y);
    ptbl = &pevent->OSEventTbl[y];
    if ((*ptbl & 0xFF) != 0) {
        x = OSUnMapTbl[*ptbl & 0xFF];
    } else {
        x = OSUnMapTbl[(*ptbl >> 8) & 0xFF] + 8;
    }
    bitx = (INT16U)(1 << x);
```

```
    prio = (INT8U)((y << 4) + x);//该 task 对应的 prio 优先级值
//ok,等待在 event 事件控制矩阵上的所有 task 中,只有在事件控制矩阵中的第 y 行的第 x 列 task
//优先级最高、最值的成为此次事件的唤醒对象[gliethttp]
#endif
//清除此 task 在 event 事件控制矩阵中的 bit 位
    pevent->OSEventTbl[y] &= ~bitx;
    if (pevent->OSEventTbl[y] == 0) {
        pevent->OSEventGrp &= ~bity;
    }
//通过 prio 优先级找到该 prio 唯一对应的 task 对应的 ptcb 进程上下文控制块
    ptcb = OSTCBPrioTbl[prio];
    ptcb->OSTCBDly = 0;//复原为正常
    ptcb->OSTCBEventPtr = (OS_EVENT *)0;//现在本 task 不悬停在任何 event 事件上
#if ((OS_Q_EN > 0) && (OS_MAX_QS > 0)) || (OS_MBOX_EN > 0)
    ptcb->OSTCBMsg = pmsg;//传递消息指针
#else
    pmsg = pmsg;
#endif
    ptcb->OSTCBStatPend = pend_stat;//悬停状态值
    ptcb->OSTCBStat &= ~msk;//该 msk 事件已经发生,清除 task 上下文控制块上的 msk 位,如:OS_STAT_Q
    if (ptcb->OSTCBStat == OS_STAT_RDY) {
    //如果当前 task 只是等待该事件,那么把该 task 放到就绪控制矩阵中,允许内核调度本 task
        OSRdyGrp |= bity;
        OSRdyTbl[y] |= bitx;
    }
    return (prio);//返回本 task 对应的优先级值
}
```

浅析μCOS/II v2.85 内核 OSSemPend()和 OSSemPost()函数工作原理

文章来源:http://gliethttp.cublog.cn[转载请声明出处]

```
//-------------------------------------------------------------------------
//1.OSSemPend()函数
void OSSemPend (OS_EVENT *pevent, INT16U timeout, INT8U *perr)
{
    INT8U pend_stat;
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR cpu_sr = 0;
#endif

#if OS_ARG_CHK_EN > 0
    if (perr == (INT8U *)0) {
        return;
    }
    if (pevent == (OS_EVENT *)0) {
        *perr = OS_ERR_PEVENT_NULL;
```

```
            return;
        }
#endif
        if (pevent->OSEventType != OS_EVENT_TYPE_SEM) {
//确保该 event 控制块是 Sem 类型
            *perr = OS_ERR_EVENT_TYPE;
            return;
        }
        if (OSIntNesting > 0) {
//ISR 中,不能使用 OSSemPend()
            *perr = OS_ERR_PEND_ISR;
            return;
        }
        if (OSLockNesting > 0) {
// μ COS/II v2.85 内核已经被强制锁住
            *perr = OS_ERR_PEND_LOCKED;
            return;
        }
//非法的统统不是,信号正常,所以有必要进一步处理
        OS_ENTER_CRITICAL();
        if (pevent->OSEventCnt > 0) {
//程序的其他地方已经触发了事件,异或在初始化时设定了 n,如:OSSemCreate(2);
//所以该 task 无需悬停,直接获得事件的使用权
            pevent->OSEventCnt--;
            OS_EXIT_CRITICAL();
            *perr = OS_ERR_NONE;
            return;
        }
//当前还没有任何事件发生,所以本 task 需要悬停,让出 cpu[gliethttp]
        OSTCBCur->OSTCBStat |= OS_STAT_SEM;// 是 sem 事件让本 task 进入悬停等待的
        OSTCBCur->OSTCBStatPend = OS_STAT_PEND_OK;// 假定不是超时,为正常收到信号
//超时,如果 timeout=0,那么,本 task 将一直悬停,仅仅当收到事件触发信号后才重新进入调度队列
        OSTCBCur->OSTCBDly = timeout;
//OS_EventTaskWait()函数实现的功能:
//把本 task 从就绪控制矩阵中摘下,放到 pevent 事件专有的进程事件控制矩阵表中.
        OS_EventTaskWait(pevent);
        OS_EXIT_CRITICAL();
//因为本 task 正在运行,所以本 task 现在的优先级最高,现在本 task 已经将自己从就绪控制矩阵--调度器(x,y)矩形阵列中
//把自己摘掉,所以调度函数 OS_Sched()一定会切换到另一个 task 中执行新 task 的代码[gliethttp]
        OS_Sched();// 具体参见《浅析 μ C/OS-II v2.85 内核调度函数》
//2007-09-09 gliethttp
//可能因为 OSSemPend()中指定的 timeout 已经超时
//[由 OSTimeTick()函数把本 task 重新置入了就绪控制矩阵,具体参考《浅析 μ C/OS-II v2.85 内核 OSTimeDly()函数工作原理》],
//又或者确实在应用程序的某个地方调用了 OSSemPost(),以下代码将具体解析是有什么引起的:1.超时,2.收到正常信号
        OS_ENTER_CRITICAL();
        if (OSTCBCur->OSTCBStatPend != OS_STAT_PEND_OK) {
//是因为 timeout 超时,使得本 task 获得重新执行的机会
            pend_stat = OSTCBCur->OSTCBStatPend;
```

```
//清除 event 事件块上本 task 的标志
            OS_EventTOAbort(pevent);
            OS_EXIT_CRITICAL();
            switch (pend_stat) {
                case OS_STAT_PEND_TO:
                default:
                    *perr = OS_ERR_TIMEOUT;
                    break;
                case OS_STAT_PEND_ABORT:
                    *perr = OS_ERR_PEND_ABORT;
                    break;
            }
            return;
        }
//由 OSSemPost()抛出正常事件,唤醒了本 task,因为在 OSSemPost()时,
//已经将本 task 在 event 事件控制矩阵上的对应位清除掉,并且把本 task 放入了就绪控制矩阵中,
//否则本 task 也不会执行至此.
        OSTCBCur->OSTCBEventPtr = (OS_EVENT *)0;//现在本 task 不悬停在任何 event 事件上
        OS_EXIT_CRITICAL();
        *perr = OS_ERR_NONE;
}
//--------------------------------------------------------------------------
//2.OS_EventTaskWait()函数
void OS_EventTaskWait (OS_EVENT *pevent)
{
        INT8U y;
//2007-09-09 gliethttp
//pevent 为此次 task 挂起的 EventPtr 单元
        OSTCBCur->OSTCBEventPtr = pevent;
//清除调度器中该 task 对应的标志位
        y = OSTCBCur->OSTCBY;
        OSRdyTbl[y] &= ~OSTCBCur->OSTCBBitX;
        if (OSRdyTbl[y] == 0) {
//当前 y 行对应的 8 个或 16 个 task 都已经悬停,那么当前 y 行也清除.
            OSRdyGrp &= ~OSTCBCur->OSTCBBitY;
        }
//2007-09-09 gliethttp
//将该 task 的 prio 添加到 pevent 事件控制矩阵中,这个矩阵的功能和 OSRdyGrp、OSRdyTbl 没有区别
//都是用来计算出已经就绪 tasks 中的优先级最高的那个
        pevent->OSEventTbl[OSTCBCur->OSTCBY] |= OSTCBCur->OSTCBBitX;
        pevent->OSEventGrp |= OSTCBCur->OSTCBBitY;
}
//--------------------------------------------------------------------------
//3.OS_EventTOAbort()函数
void OS_EventTOAbort (OS_EVENT *pevent)
{
        INT8U y;
//清除 event 事件控制矩阵上本 task 的标志,因为 OSTimeTick()函数未清除该单元
```

```
//它仅仅把本 task 放入就绪控制矩阵中,使得本 task 重新获得被 OS 调度的机会而已
//具体的清除工作还要自己完成
//具体参考《浅析 μC/OS-II v2.85 内核 OSTimeDly()函数工作原理》
    y = OSTCBCur->OSTCBY;
    pevent->OSEventTbl[y] &= ~OSTCBCur->OSTCBBitX;
    if (pevent->OSEventTbl[y] == 0x00) {
        pevent->OSEventGrp &= ~OSTCBCur->OSTCBBitY;
    }
    OSTCBCur->OSTCBStatPend = OS_STAT_PEND_OK;
    OSTCBCur->OSTCBStat = OS_STAT_RDY;
    OSTCBCur->OSTCBEventPtr = (OS_EVENT *)0;//现在本 task 不悬停在任何 event 事件上
}
//--------------------------------------------------------------------
//4.OSSemPost()函数
INT8U OSSemPost (OS_EVENT *pevent)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR cpu_sr = 0;//方式 3 将把 cpsr 状态寄存器推入临时堆栈 cpu_sr 中,可以安全返回之前的中断状态
#endif

#if OS_ARG_CHK_EN > 0
    if (pevent == (OS_EVENT *)0) {
        return (OS_ERR_PEVENT_NULL);
    }
#endif
    if (pevent->OSEventType != OS_EVENT_TYPE_SEM) {
        return (OS_ERR_EVENT_TYPE);
    }
    OS_ENTER_CRITICAL();
    if (pevent->OSEventGrp != 0) {
    //2007-09-09 gliethttp
    //OS_EventTaskRdy()函数将摘掉等待在 pevent 事件控制矩阵上的 task 中优先级最高的 task
    //如果该 task 仅仅等待该 pevent 事件,那么将该 task 添加到就绪控制矩阵中
    //OSRdyGrp |= bity;
    //OSRdyTbl[y] |= bitx;这样调度程序就会根据情况调度 OS_Sched()该 task 了
        (void)OS_EventTaskRdy(pevent, (void *)0, OS_STAT_SEM, OS_STAT_PEND_OK);
        OS_EXIT_CRITICAL();
    //可能刚刚放到就绪控制矩阵上的被唤醒的 task 的优先级比调用 OSSemPost()函数的进程 B 优先级高
    //所以需要调用 shedule 函数,
    //如果真的高,那么调用 OSSemPost()函数的进程 B 就要被抢占,os 将会切换到新的 task 去执行[gliethttp]
    //如果没有调用 OSSemPost()函数的进程 B 优先级高,那么 os 不会切换,仍然继续执行进程 B,OSSemPost()正常返回
        OS_Sched();
        return (OS_ERR_NONE);
    }
    //没有任何一个 task 悬停在本 event 事件控制矩阵上,
    //那么单纯的对 pevent->OSEventCnt++加操作.
    if (pevent->OSEventCnt < 65535u) {
        pevent->OSEventCnt++;
```

```
            OS_EXIT_CRITICAL();
            return (OS_ERR_NONE);
        }
        //已经堆积了 0xffff 次,溢出
        OS_EXIT_CRITICAL();
        return (OS_ERR_SEM_OVF);
}
//--------------------------------------------------------------------
//5.OS_EventTaskRdy()函数
INT8U OS_EventTaskRdy (OS_EVENT *pevent, void *pmsg, INT8U msk, INT8U pend_stat)
{
        OS_TCB *ptcb;
        INT8U x;
        INT8U y;
        INT8U prio;
#if OS_LOWEST_PRIO <= 63
        INT8U bitx;
        INT8U bity;
#else
        INT16U bitx;
        INT16U bity;
        INT16U *ptbl;
#endif

#if OS_LOWEST_PRIO <= 63
//小于 64 个 task 时,快速计算
//最有优先权的 task 位于事件控制矩阵中的第 y 行的第 x 列
        y = OSUnMapTbl[pevent->OSEventGrp];
        bity = (INT8U)(1 << y);
        x = OSUnMapTbl[pevent->OSEventTbl[y]];
        bitx = (INT8U)(1 << x);
        prio = (INT8U)((y << 3) + x);
#else
//对于 256 个 task
//最有优先权的 task 位于事件控制矩阵中的第 y 行的第 x 列
//以下的操作原理具体参见《浅析μC/OS-II v2.85 内核调度函数》
        if ((pevent->OSEventGrp & 0xFF) != 0) {
            y = OSUnMapTbl[pevent->OSEventGrp & 0xFF];
        } else {
            y = OSUnMapTbl[(pevent->OSEventGrp >> 8) & 0xFF] + 8;
        }
        bity = (INT16U)(1 << y);
        ptbl = &pevent->OSEventTbl[y];
        if ((*ptbl & 0xFF) != 0) {
            x = OSUnMapTbl[*ptbl & 0xFF];
        } else {
            x = OSUnMapTbl[(*ptbl >> 8) & 0xFF] + 8;
        }
```

```
    bitx = (INT16U)(1 << x);
    prio = (INT8U)((y << 4) + x);//该 task 对应的 prio 优先级值
//ok,等待在 event 事件上的所有 task 中,只有在事件控制矩阵中的第 y 行的第 x 列 task
//优先级最高、最值的成为此次事件的唤醒对象[gliethttp]
#endif
//清除此 task 在 event 事件控制矩阵中的 bit 位
    pevent->OSEventTbl[y] &= ~bitx;
    if (pevent->OSEventTbl[y] == 0) {
        pevent->OSEventGrp &= ~bity;
    }
//通过 prio 优先级找到该 prio 唯一对应的 task 对应的 ptcb 进程上下文控制块
    ptcb = OSTCBPrioTbl[prio];
    ptcb->OSTCBDly = 0;//复原为正常
    ptcb->OSTCBEventPtr = (OS_EVENT *)0;//现在本 task 不悬停在任何 event 事件上
#if ((OS_Q_EN > 0) && (OS_MAX_QS > 0)) || (OS_MBOX_EN > 0)
    ptcb->OSTCBMsg = pmsg;//传递消息指针
#else
    pmsg = pmsg;
#endif
    ptcb->OSTCBStatPend = pend_stat;//悬停状态值
    ptcb->OSTCBStat &= ~msk;//该 msk 事件已经发生,清除 task 上下文控制块上的 msk 位,如:OS_STAT_SEM
    if (ptcb->OSTCBStat == OS_STAT_RDY) {
    //如果当前 task 只是等待该事件,那么把该 task 放到就绪控制矩阵中,允许内核调度本 task
        OSRdyGrp |= bity;
        OSRdyTbl[y] |= bitx;
    }
    return (prio);//返回本 task 对应的优先级值
}
```

浅析μC/OS-II v2.85 内核 OSTimeDly()函数工作原理

文章来源:http://gliethttp.cublog.cn[转载请声明出处]

```
//---------------------------------------------------------------------------
//1.OSTimeDly()函数
void OSTimeDly (INT16U ticks)
{
    INT8U y;
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR cpu_sr = 0;
#endif
    if (OSIntNesting > 0) {
        return;//在中断处理函数中调用了 OSTimeDly(),那么直接退出
    }
    if (ticks > 0) {
        OS_ENTER_CRITICAL();
//调用 OSTimeDly()的进程自己把自己从就绪控制矩阵中拿下来,
```

```
//即:去掉调度器(x,y)矩形阵列(OSRdyTbl,OSRdyGrp)中该 task 对应的 bit 位,使得调度器不考虑
//该 task 的调度
            y = OSTCBCur->OSTCBY;
            OSRdyTbl[y] &= ~OSTCBCur->OSTCBBitX;
            if (OSRdyTbl[y] == 0) {
                OSRdyGrp &= ~OSTCBCur->OSTCBBitY;
            }
//延时 ticks 值,放入 OSTCBDly 单元,在 os 时钟滴答处理函数 OSTimeTick()中,会处理该单元[gliethttp]
            OSTCBCur->OSTCBDly = ticks;
            OS_EXIT_CRITICAL();
//因为本 task 正在运行,所以本 task 现在的优先级最高,现在本 task 已经将自己从就绪控制矩阵中--调度器(x,y)矩形阵列
//把自己摘掉,所以调度函数 OS_Sched()一定会切换到另一个 task 中执行新 task 的代码[gliethttp]
            OS_Sched();//具体参见《浅析 μC/OS-II v2.85 内核调度函数》
        }//ticks==0,那么什么也不做
}
//--------------------------------------------------------------------
//2.OSTimeTick()--在定时中断里引用的系统滴答函数
void OSTimeTick (void)
{
    OS_TCB *ptcb;
#if OS_TICK_STEP_EN > 0
    BOOLEAN step;
#endif
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR cpu_sr = 0;//该 3 方式将使中断状态寄存器放入堆栈中

#endif

#if OS_TIME_TICK_HOOK_EN > 0
    OSTimeTickHook();
#endif
#if OS_TIME_GET_SET_EN > 0
    OS_ENTER_CRITICAL();
    OSTime++;
    OS_EXIT_CRITICAL();
#endif
    if (OSRunning == OS_TRUE) {
#if OS_TICK_STEP_EN > 0
//控制内核的 tick
        switch (OSTickStepState) {
            case OS_TICK_STEP_DIS:
                step = OS_TRUE;
                break;
            case OS_TICK_STEP_WAIT:
                step = OS_FALSE;
                break;
            case OS_TICK_STEP_ONCE:
//本次 tick 将将影响到 task 的 OSTCBDly 域
```

```
//但以后的 tick 将一直被屏蔽,不会影响到 OSTCBDly 域
//直到外部将 OSTickStepState 改变为止[gliethttp]
                    step = OS_TRUE;
                    OSTickStepState = OS_TICK_STEP_WAIT;
                    break;
                default:
                    step = OS_TRUE;// 本次 tick 将影响到 task 的 OSTCBDly 域
                    OSTickStepState = OS_TICK_STEP_DIS;
                    break;
            }
            if (step == OS_FALSE) {
                return;
            }
#endif
            ptcb = OSTCBList;
//2007-09-08 gliethttp
//OSTCBList 是一个按进程创建的先后顺序链接成的 task 单向链表,最后创建的 task 在最前面,最先创建的
//task 在单向链表的尾端,
//所以 OS_TaskIdle 空闲进程在链表的最后,因为它最先创建
            while (ptcb->OSTCBPrio != OS_TASK_IDLE_PRIO) {
                OS_ENTER_CRITICAL();
                if (ptcb->OSTCBDly != 0) {
                    if (--ptcb->OSTCBDly == 0) {
//该 task 的延时时间已到,解析此次延时是 OSTimeDly()引起的,还是 OSQPend()之类超时引起的[gliethttp]
                        if ((ptcb->OSTCBStat & OS_STAT_PEND_ANY) != OS_STAT_RDY) {
                            //2007-09-08 gliethttp
                            //如:由 OSSemPend (pevent,timeout,perr);定义的 timeout 已经到了,对应 task 需要运行了
                            //超时时间到,所以不论当前进程是在做什么,只要时间一到
                            //该 task 就可以运行了,所以清除所有事件标志,之后状态标示为 OS_STAT_PEND_TO(超时)
                            ptcb->OSTCBStat &= ~(INT8U)OS_STAT_PEND_ANY;
                            ptcb->OSTCBStatPend = OS_STAT_PEND_TO;//超时异常
                        } else {
                            //2007-09-08 gliethttp
                            //说明该 task 调用的是 OSTimeDly()
                            ptcb->OSTCBStatPend = OS_STAT_PEND_OK;// 正常结束
                        }
                        if ((ptcb->OSTCBStat & OS_STAT_SUSPEND) == OS_STAT_RDY) {
                            //2007-09-08 gliethttp
                            //如果该 task 没有 suspend,那么把当前就绪的 task 加入到运行调度器的就绪控制矩阵中
                            //等待被调度
                            OSRdyGrp |= ptcb->OSTCBBitY;
                            OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
                        }
                    }
                }
                ptcb = ptcb->OSTCBNext;//继续运算下一个 task 的 OSTCBDly 时间域
                OS_EXIT_CRITICAL();
            }
```

```
    }
}
```

浅析μC/OS-II v2.85 内核调度函数

文章发表于：2007-09-17 02:33

浅析μC/OS-II v2.85 内核 OSTimeDly()函数工作原理
http://gliethttp.cublog.cn

```
//-------------------------------------------------------------------------
//1.OSTimeDly()函数
void OSTimeDly (INT16U ticks)
{
    INT8U y;
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR cpu_sr = 0;
#endif
    if (OSIntNesting > 0) {
        return;//在中断处理函数中调用了 OSTimeDly(),那么直接退出
    }
    if (ticks > 0) {
        OS_ENTER_CRITICAL();
//调用 OSTimeDly()的进程自己把自己从就绪控制矩阵中拿下来,
//即:去掉调度器(x,y)矩形阵列(OSRdyTbl,OSRdyGrp)中该 task 对应的 bit 位,使得调度器不考虑
//该 task 的调度
        y = OSTCBCur->OSTCBY;
        OSRdyTbl[y] &= ~OSTCBCur->OSTCBBitX;
        if (OSRdyTbl[y] == 0) {
            OSRdyGrp &= ~OSTCBCur->OSTCBBitY;
        }
//延时 ticks 值,放入 OSTCBDly 单元,在 os 时钟滴答处理函数 OSTimeTick()中,会处理该单元[gliethttp]
        OSTCBCur->OSTCBDly = ticks;
        OS_EXIT_CRITICAL();
//因为本 task 正在运行,所以本 task 现在的优先级最高,现在本 task 已经将自己从就绪控制矩阵中--调度器(x,y)矩形阵列
//把自己摘掉,所以调度函数 OS_Sched()一定会切换到另一个 task 中执行新 task 的代码[gliethttp]
        OS_Sched();//具体参见《浅析μC/OS-II v2.85 内核调度函数》
    }//ticks==0,那么什么也不做
}
//-------------------------------------------------------------------------
//2.OSTimeTick()--在定时中断里引用的系统滴答函数
void OSTimeTick (void)
{
    OS_TCB *ptcb;
#if OS_TICK_STEP_EN > 0
    BOOLEAN step;
#endif
#if OS_CRITICAL_METHOD == 3
```

```
    OS_CPU_SR cpu_sr = 0;// 该 3 方式将使中断状态寄存器放入堆栈中

#endif

#if OS_TIME_TICK_HOOK_EN > 0
    OSTimeTickHook();
#endif
#if OS_TIME_GET_SET_EN > 0
    OS_ENTER_CRITICAL();
    OSTime++;
    OS_EXIT_CRITICAL();
#endif
    if (OSRunning == OS_TRUE) {
#if OS_TICK_STEP_EN > 0
//控制内核的 tick
        switch (OSTickStepState) {
            case OS_TICK_STEP_DIS:
                step = OS_TRUE;
                break;
            case OS_TICK_STEP_WAIT:
                step = OS_FALSE;
                break;
            case OS_TICK_STEP_ONCE:
//本次 tick 将将影响到 task 的 OSTCBDly 域
//但以后的 tick 将一直被屏蔽,不会影响到 OSTCBDly 域
//直到外部将 OSTickStepState 改变为止[gliethttp]
                step = OS_TRUE;
                OSTickStepState = OS_TICK_STEP_WAIT;
                break;
            default:
                step = OS_TRUE;// 本次 tick 将影响到 task 的 OSTCBDly 域
                OSTickStepState = OS_TICK_STEP_DIS;
                break;
        }
        if (step == OS_FALSE) {
            return;
        }
#endif
        ptcb = OSTCBList;
//2007-09-08 gliethttp
//OSTCBList 是一个按进程创建的先后顺序链接成的 task 单向链表,最后创建的 task 在最前面,最先创建的
//task 在单向链表的尾端,
//所以 OS_TaskIdle 空闲进程在链表的最后,因为它最先创建
        while (ptcb->OSTCBPrio != OS_TASK_IDLE_PRIO) {
            OS_ENTER_CRITICAL();
            if (ptcb->OSTCBDly != 0) {
                if (--ptcb->OSTCBDly == 0) {
//该 task 的延时时间已到,解析此次延时是 OSTimeDly()引起的,还是 OSQPend()之类超时引起的[gliethttp]
```

```
                    if ((ptcb->OSTCBStat & OS_STAT_PEND_ANY) != OS_STAT_RDY) {
                        //2007-09-08 gliethttp
                        //如:由 OSSemPend (pevent,timeout,perr);定义的 timeout 已经到了,对应 task 需要运行了
                        //超时时间到,所以不论当前进程是在做什么,只要时间一到
                        //该 task 就可以运行了,所以清除所有事件标志,之后状态标示为 OS_STAT_PEND_TO(超时)
                        ptcb->OSTCBStat &= ~(INT8U)OS_STAT_PEND_ANY;
                        ptcb->OSTCBStatPend = OS_STAT_PEND_TO;//超时异常
                    } else {
                        //2007-09-08 gliethttp
                        //说明该 task 调用的是 OSTimeDly()
                        ptcb->OSTCBStatPend = OS_STAT_PEND_OK;//正常结束
                    }
                    if ((ptcb->OSTCBStat & OS_STAT_SUSPEND) == OS_STAT_RDY) {
                        //2007-09-08 gliethttp
                        //如果该 task 没有 suspend,那么把当前就绪的 task 加入到运行调度器的就绪控制矩阵中
                        //等待被调度
                        OSRdyGrp |= ptcb->OSTCBBitY;
                        OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
                    }
                }
            }
            ptcb = ptcb->OSTCBNext;//继续运算下一个 task 的 OSTCBDly 时间域
            OS_EXIT_CRITICAL();
        }
    }
}
```

使用 uC/OS-II 操作系统的短信息电话机

[日期：2004-12-10] 来源：单片机及嵌入式系统应用　作者：何昭辉 谢吉华

摘要：将μC/OS-II 实时嵌入式操作系统移植到 EPSON 八位单片机上来开发短信息电话机。此电话机除普通电话的通用功能外，还增加了短消息收/发功能、信息浏览与查阅功能、信息点播与信息订阅功能等。

关键词：短信息电话机 实时操作系统 μC/OS-II

1 背 景

后 PC 时代的到来，使人们开始越来越多地接触到一个新的概念——嵌入式产品。手机、PDA（如商务通）等均属于手持式嵌入式产品，VCD 机、机顶盒等也属于嵌入式产品，而车载 GPS 系统、数控机床、网络冰箱等同样都属于嵌入式产品。信息电器具有信息服务功能，如网络浏览、视频点播、文字处理、电子邮件、个人数字助理等。后 PC 时代，家用电器将向数字化和网络化发展，计算机都将嵌入到电视机、冰箱、微波炉、电话等中，并通过家庭控制中心与 Internet 连接，转变为智能网络家电；还可以实现远程医疗、远程教育等。据估计未来十年间，信息家电（网络冰箱、机顶盒、家庭网关、数字机顶盒等）总量可达到 1~1.5 亿台，总市场需求将达到 900~1500 亿元。

Short Messages Phone(短信息电话机)是一种操作简便、成本低廉的固定电话信息终端产品。它集成了网络和智能电话的优点，是一种基于信息网络的，具有交互功能的信息终端产品，可以通过固定电话网络与信息中心进行数据双向传送。

2 系统功能及其实现方案

2.1 功能特点

短信息电话机除了实现普通电话机的通用功能外，还增加了短消息收/发、信息浏览与查阅、信息点播、信息订阅、电话簿、来电号码显示与存储、通话记录、来电转移、呼叫等待、呼叫限制、对屏幕的灰度进行设置、对回执进行设置以及时钟和日历功能、闹钟功能、特色铃声功能、记事本功能等。

短信息电话机还可以和信息中心在线交互，查看信息中心提供的信息（例如天气预报、电视预告、电影预告、股价查询、话费查询、区号邮编查询、新闻栏目、彩票信息等等），还可以订阅一些自己喜欢的信息。在线浏览的信息取决于信息中心提供的信息。

短信息电话机就像是一个固定的手机。手机的功能几乎它都具备。

2.2 硬件方案

（1）MCU 的选择

MCU 是整个方案的核心，由它来处理 CPE（客户端设备）与 IIS（集成信息系统），控制整个电话机的各个功能部件。由于人机界面对速度的要求不是很高，数据通信速度也相对较低，一般的 8 位 MCU 就能够满足方案的要求。我们选择 EPSON 的 EOC88 系列的 E0C88104 芯片。

（2）数据信号的解/编码

① FSK 信号的解码。FSK（频移键控）的解码选用台湾华邦公司的 W91030。它是用同步串口与 MCU 连接的，不仅能提供 FSK 的解码，而且还能提供 CAS 信号的检测（CAS 信号是 IIS 与 CPE 连接时下传的一个很重要的握手信号）和振铃检测，是一款很实用的芯片，但它的成本较高。如果用分离器件，也能做到 FSK 解码，但这样做没有经过验证，风险较大。

② DTMF 信号的解码。选用的是 9170，这一款芯片是很通用的。

③ DTMF 信号的编码。选用的是台湾华邦公司的 W91082，同样是以同步串口与 MCU 相连接。

④ FSK 的编码。FSK 信号的编码不是必需的，考虑到 FSK 传送的效率要比 DTMF 高 20~28 倍，当然最好选用 FSK 上传方式。由于相应的 FSK 的编码器件价位很高，本方案采用了低成本的软件方式作为后备模块。

（3）语音电路

目前选用的是 Philips 公司的 TEA1098。该器件集成了普通话音放大、消侧音及免提功能，实用方便，但价位有点高。考虑到我们初次接触电话电路，而且这部分也不是关键技术，为了缩短研发周期，选择这款芯片是比较合适的。

2.3 软件方案

本系统选用 μ C/OS-II 操作系统，将其移植到所选的 MCU 上。在 main 中建立一个起始任务

StartTask。

```
int main (void){
```

```
OSInit();

OSTaskCreate((void*)StartTask,(void*)0X00,(OS_STK ) &StartTaskStk[START_TASK_STK_SIZE -1],0);

OSStart();

return 0;

}
```

起始任务中，首先建立一系列的信号量和邮箱：

```
Sem_Int_Ring = OSSemCreate(0) ;唤醒振铃任务

Sem_Any_SMTask = OSSemCreate(0) ;唤醒短消息任务

Sem_Int_Keyboard = OSSemCreate(0) ;唤醒键盘任务

Sem_Int_ClockTask = OSSemCreate(0) ;唤醒时钟中断

Mbox_Any_UI= OSMboxCreate((void *)0);唤醒 UI 任务

Mbox_Int_Hook= OSMboxCreate((void *)0);唤醒摘挂机任

;务
```

然后，用 OSTaskCreate()函数建立 6 个任务。任务名分别为：HandsetTask（摘/挂机任务）、RingTask（响铃任务）、KeyTask（键盘任务）、SmTask（短消息任务）、UITask（用户界面（UI）任务）、ClockTask（时钟任务）。最后，在起始任务中将它本身删除掉。

（1）摘/挂机任务

当拿起听筒或放下听筒时，就产生中断。在中断中，调用 OSMboxPost(Mbox_Int_Hook，Msg_Int_Hook)来唤醒摘/挂机任务，同时清除中断标志。Msg_Int_Hook[0]＝0x00 为摘机，Msg_Int_Hook[0]＝0x01 为挂机。

摘/挂机任务不断调用 OSMboxPend(Mbox _Int_Hook，0，&err)来获得信箱。获得信箱后，根据 Msg_Int_Hook[0]中的值，就知道是摘机还是挂机，然后调用 OSMboxPost(Mbox_Any_UI，Msg_Any_UI)来唤醒 UI 任务。Msg_Any_UI [0]＝任务号 1，Msg_Any_UI [1]＝0x00 为摘机，Msg_Any_UI [1]＝0x01 为挂机。

在挂机的时候，如果先前是在响铃的时候摘机的，那么 UI 任务把它当做已接来电处理；如果不是在响铃的时候摘机的，那么在挂机的时候把它当做已拨电话处理。

（2）振铃任务

当铃声来到时，就产生中断。在这个中断中，调用 OSSemPost(Sem_Int_Ring)来唤醒振铃任务。

响铃任务调用 OSSemPend(Sem_Int_Ring，0，&err)来获得信号量。获得信号量后，就把下传的号码接收下来；根据号

码就可以知道是短信息，还是普通电话。

① 当是短信息时，就调用 OSSemPost(Sem_Any _SMTask)来唤醒短消息任务；

② 如果是电话，就响铃，同时调用 OSMboxPost(Mbox_Any_UI，Msg_Any_UI)来唤醒 UI 任务。这时 Msg_Any_UI[0]＝任务号 2，从 Msg_Any_UI[1]开始存的是来电时间和电话号码，然后调用 OSSemPend(Sem_Int_Ring，RING_RECV_TIMEOUT，&err)，来延时响铃一段时间。

如果在这段时间内没有接电话，那么必然会超时，于是就清除铃声中断，同时调用 OSMboxPost(Mbox_Any_UI，Msg_Any_UI)来唤醒 UI 任务，这时 Msg_Any_UI[0]＝任务号 2，Msg_Any_UI[1]＝0xff 表示铃声结束。UI 任务把它当作未接来电处理。

如果在这段时间内接电话，就必然唤醒摘/挂机任务。如果从响铃到对摘/挂机任务处理完毕所需要的总时间还没有到 RING_RECV_TIMEOUT，那么，当时间到了，UI 任务也会收到铃声结束的消息；但此时，UI 任务不对它做任何处理。

响铃任务如何区别是正常的电话还是信息中心的来电信息呢?其实很简单。信息中心也是一个终端，是由一个特定的号码来确定的。就像用不同的电话号码来区别不同的用户一样。这样响铃任务就可以根据电话线上下传的号码，来确定是哪种情况了。

（3）键盘任务

有按键按下时，产生一中断，在中断服务子程序中调用 OSSemPost(Sem_Int_Keyboard)来唤醒键盘任务，同时清除中断标志。键盘任务调用 OSSemPend(Sem_Int_Keyboard,0,&err)来获得信号量。获得信号量后，键盘任务将调用 OSMboxPost(Mbox_Any_UI，Msg_Any_UI)唤醒 UI 任务。

Msg_Any_UI[0]为任务号 3， Msg_Any_UI[1]为键值。

（4）短消息任务

该任务调用 OSSemPend(Sem_Any_SMTask,0,&err)获得信号量。当获得信号量后，通过标志位判别是准备上传数据还是下传数据。

① 下传数据时，将下传的数据放在 recv_buf[]中，然后调用 OSMboxPost(Mbox_Any_UI，Msg_Any_UI)唤醒 UI 任务。Msg_Any_UI [0]＝任务号 4，从 Msg_Any_UI [1]开始存的是从 recv_buf[]中筛选过的数据。这是在连上后从信息中心下载信息的情况，其中 Msg_Any_UI [1]存的是信息的种类号，种类号都是大于 0x80H 小于 0xFFH 的；没有连上信息中心或者是连上之后完成任务。这时 recv_buf[1]存的就是 0xff；如果刚刚连上，就在 recv_buf[1]存 0x01。

② 如果用户需要发送短消息任务，先将信息存在 Msg_send_buf[]中，然后 UI 任务调用 OSSemPost(Sem_Any_SMTask)，将该任务激活。该任务提取存在 Msg_send_buf[]中的信息后，包装后保存在 send_buf[]，然后将其发出去。

（5）用户界面任务

UI 任务一直在等待消息。当它被激活时，根据存在 Msg_Any_UI[255]中的不同数据进行不同的处理。详述见“3.用户界面任务详述”。

（6）时钟任务

单片机每 10ms 产生一个中断，在这个中断服务子程序中，对 clock_count 减 1。当减为 0 时，就调用

OSSemPost(Sem_Int_ClockTask)唤醒时钟任务；同时，clock_count 重新回到 100。这样每 1s，就会调用一次 OSSemPost(Sem_Int_ClockTask)。时钟任务调用 OSSemPend(Sem_Int_ClockTask,0,&err)获得信号量。当获得信号量时，就更新时间。如果在 23：59：59 的时候更新时间，那就要同时更新日期和星期。如果此时 UI 任务处于待机状态，时钟任务就调用 OSMboxPost(Mbox_Any_UI,Msg_Any_UI)来唤醒 UI 任务。Msg_Any_UI[0]为任务号 0x06。时钟任务只是负责更新时间，显示时间是由 UI 任务在待机的时候来完成的。

3 用户界面任务详述

在所有这些任务中，用户界面任务做的工作相对比较烦琐。首先，根据放在 Msg_Any_UI[255]中的第一个字节，判断是什么任务激活了本任务。

① 如果是摘/挂机任务激活的，再判断第二个字节的值，根据值的不同知道是挂机还是摘机。如果是挂机，就在液晶上显示待机画面。如果是摘机，就在液晶上显示打电话图标和有关的字样，并等待键盘任务送来的数字键，将其显示出来。(只有主动向外打电话才显示数字键，在接电话按数字键转接时不显示数字键。)

② 如果是振铃任务激活的，就将 Msg_Any_UI[255]中接下来的来电号码、时间和来电图标显示在液晶上。

③ 如果是键盘任务激活的，就根据 3 号任务放在 Msg_Any_UI[255]中的键值和当前的工作模式做不同的处理。

如果是在摘机模式下，就只是响应数字键和#*键，并每按一个键就通过 5 号任务号码发出去。

如果是在挂机模式下，那么，就可以响应数字键和有关操作菜单的键（包括进入各个子菜单的快捷键、上下键、左右键、返回上级菜单键、快速返回待机画面的键、确定键、删除键、拼音输入法/字母输入法/数字输入法/短语输入法切换键）。各个菜单是不同的状态，通过键值和原来的状态可以知道现在该进入什么状态。然后根据新得到的状态，进行相应的液晶刷新和完成相应的功能。比如添加、删除、查找电话簿；添加、删除、查找通话记录；编辑短消息并发送出去，删除草稿箱里的短消息；写记事本，设置闹钟、时间、日期，设置信箱的密码，设置信息中心的号码等。

如果是在与信息中心的连接模式下，那么就可以响应上下键，确定键（进入下级菜单），返回上一级菜单，通过选择返回这个选项来实现。

④如果是短消息任务激活的，就根据 Msg_ Any_UI[255]中的命令码，来确定该信息的模块种类是信息下载管理模块、信息上传管理模块、信息询问管理模块，或者是屏幕信息输出管理模块，然后，再根据 Msg_Any_UI[255]中接下来的信息做相应的处理。

⑤ 如果是时钟任务激活的，UI 任务就刷新界面上的时间、日期和星期这些信息。

4 调试环境与调试成果

此程序用 EPSON 公司的 S1C88 C-Compiler 编译器编译，用 EPSON 公司的 ICE88UR 的 E0C88 系列在线仿真器进行仿真。在仿真器下仿真完毕后，再脱机运行。试验结果令人满意，达到了预期的效果。

关于 uC/OS-II 中优先级翻转问题

作者：秦绍华 陈涤　来源：山东大学　更新日期：2005-03-01
简述：就 uC/OS-II 中的优先级翻转问题与大家一同进行探讨
原文发表于 单片机与嵌入式系统应用

1 uC/OS-II 的运行机制

在嵌入式系统的应用中，实时性是一个重要的指标，而优先级翻转是影响系统实时性的重要问题。本文着重分析优先级翻转问题的产生和影响，以及在 uC/OS-II 中的解决方案。

uC/OS-II 采用基于固定优先级的占先式调度方式，是一个实时、多任务的操作系统。系统中的每个任务具有一个任务控制快 OS_TCB，任务控制块记录任务执行的环境，包括任务的优先级，任务的堆栈指针，任务的相关事件控制块指针等。内核将系统中处于就绪态的任务在就绪表(ready list)进行标注，通过就绪表中的两个变量 OSRdyGrp 和 OSRdyTbl[]可快速查找系统中就绪的任务。在 uC/OS-II 中每个任务有唯一的优先级，因此任务的优先级也是任务的唯一编号(ID)，可以作为任务的唯一标识。内核可用控制块优先级表 OSTCBPrioTbl[]由任务的优先级查到任务控制块的地址。uC/OS-II 主要就是利用任务控制快 OS_TCB、就绪表(ready list)和控制块优先级表 OSTCBPrioTbl[]来进行任务调度的。任务调度程序 OSSched()首先由就绪表(ready list)中找到当前系统中处于就绪态的优先级最高的任务，然后根据其优先级由控制块优先级表 OSTCBPrioTbl[]取得相应任务控制块的地址，由 OS_TASK_SW()程序进行运行环境的切换。将当前运行环境切换成该任务的运行环境，则该任务由就绪态转为运行态。

当这个任务运行完毕或因其它原因挂起时，任务调度程序 OSSched()再次到就绪表(ready list)中寻找当前系统中处于就绪态中优先级最高的任务，转而执行该任务，如此完成任务调度。若在任务运行时发生中断，则转向执行中断程序，执行完毕后不是简单的返回中断调用处，而是由 OSIntExit()程序进行任务调度，执行当前系统中优先级最高的就绪态任务。当系统中所有任务都执行完毕时，任务调度程序 OSSched()就不断执行优先级最低的空闲任务 OSTaskIdle()，等待用户程序的运行。

2 uC/OS-II 中的优先级翻转问题

在 uC/OS-II 中，多个任务按照优先级高低由内核调度执行，而且任务调度所花的时间是常数，与应用程序中建立的任务数无关。对于占先式内核，任务的响应时间是确定的，而且是最优化的，占先式内核保证最高优先级的任务最先执行。

任务的响应时间=寻找最高优先级任务的时间+任务切换时间

在 uC/OS-II 中寻找进入就绪态的最高优先级任务是通过查就绪表实现的，这减少了所需时间。

```
y=OSUnMapTbl[OSRdyGrp];
x=OSUnMapTbl [OSRdyTbl[y]];
prio=(y<<3)+x;
```

任务切换是通过调用汇编函数 OS_TASK_SW()来实现的，主要完成两个任务运行环境的保存和恢复。因此用户可以通过安排任务的优先级，保证系统的实时性。当涉及到共享资源的互斥访问时，多任务实时操作系统常常会出现优先级翻转问题(priority inversion)，不能保证高优先级任务的响应时间，影响系统的实时性，uC/OS-II 中也存在同样问题。所谓优先级翻转问题(priority inversion)即当一个高优先级任务通过信号量机制访问共享资源时，该信号量已被一低优先级任务占有，而这个低优先级任务在访问共享资源时可能又被其它一些中等优先级的任务抢 先，因此造成高优先级任务被许多具有较低优先级的任务阻塞，实时性难以得到保证。

例如：有优先级为 A、B 和 C 的三个任务，优先级 A>B>C，任务 A，B 处于挂起状态，等待某一事件的发生，任务 C 正在运行，此时任务 C 开始使用某一共享资源 S。在使用中，任务 A 等待的事件到来，任务 A 转为就绪态，因为它比任务 C 优先级高，所以立即执行。当任务 A 要使用共享资源 S 时，由于其正在被任务 C 使用，因此任务 A 被挂起，任务 C 开始运行。如果此时任务 B 等待的事件到来，则任务 B 转为就绪态。由于任务 B 的优先级比任务 C 高，因此任务 B 开始运行，直到其运行完毕，任务 C 才开始运行。直到任务 C 释放共享资源 S 后，任务 A 才得以执行。在这种情况下，优先

级发生了翻转，任务 B 先于任务 A 运行。这样便不能保证高优先级任务的响应时间，解决优先级翻转问题有优先级天花板(priority ceiling)和优先级继承(priority inheritance)两种办法。

优先级天花板是当任务申请某资源时，把该任务的优先级提升到可访问这个资源的所有任务中的最高优先级，这个优先级称为该资源的优先级天花板。这种方法简单易行，不必进行复杂的判断，不管任务是否阻塞了高优先级任务的运行，只要任务访问共享资源都会提升任务的优先级。在 uC/OS-II 中，可以通过 OSTaskChangePrio()改变任务的优先级，但是改变任务的优先级是很花时间的。如果不发生优先级翻转而提升了任务的优先级，释放资源后又改回原优先级，则无形中浪费了许多 CPU 时间，也影响了系统的实时性。

优先级继承是当任务 A 申请共享资源 S 时，如果 S 正在被任务 C 使用，通过比较任务 C 与自身的优先级，如发现任务 C 的优先级小于自身的优先级，则将任务 C 的优先级提升到自身的优先级，任务 C 释放资源 S 后，再恢复任务 C 的原优先级。这种方法只在占有资源的低优先级任务阻塞了高优先级任务时才动态的改变任务的优先级，如果过程较复杂，则需要进行判断。uC/OS-II 不支持优先级继承，而且其以任务的优先级作为任务标识，每个优先级只能有一个任务，因此，不适宜在应用程序中使用优先级继承。

3 uC/OS-II 中优先级翻转问题的解决

在 uC/OS-II 中，为解决优先级翻转影响任务实时性的问题，可以借鉴优先级继承的方法对优先级天花板方法进行改进。对 uC/OS-II 的使用，共享资源任务的优先级不是全部提升，而是先判断再决定是否提升。即当有任务 A 申请共享资源 S 时，首先判断是否有别的的任务正在占用资源 S，若无，则任务 A 继续执行，若有，假设为任务 B 正在使用该资源，则判断任务 B 的优先级是否低于任务 A，若高于任务 A，则任务 A 挂起，等待任务 B 释放该资源，如果任务 B 的优先级低于任务 A，则提升任务 B 的优先级到该资源的优先级天花板，当任务 B 释放资源后，再恢复到原优先级。在 uC/OS-II 中，每个共享资源都可看作一个事件，每个事件都有相应的事件控制块 ECB。在 ECB 中包含一个等待本事件的等待任务列表，该列表包括 OSEventTbl[]和 OSEventGrp 两个域，通过对等待任务列表的判断可以很容易的确定是否有多个任务在等待该资源，同时也可判断任务的优先级与当前任务优先级的高低，从而决定是否需要用 OSTaskChangePio()来改变任务的优先级。

这样，仅在优先级有可能发生翻转的情况下才改变任务的优先级，而且利用事件的等待任务列表进行判断，比用 OSTaskChangePio()来改变任务的优先级速度快，并占用较少的 CPU 时间，有利于系统实时性的提高。

总之，优先级翻转问题是多任务实时操作系统普遍存在的问题，这个问题也存在于 uC/OS-II 中。通过在应用程序中进行简单的判断，在可能出现优先级翻转的情况下动态的改变任务的优先级，可以有效地避免任务的优先级翻转，保证高优先级任务的执行，提高了系统的实时性。