Micriμm, Inc.

μC/OS-II
and
Mutual Exclusion Semaphores

Application Note

AN-1002

Jean J. Labrosse
Jean.Labrosse@Micrium.com
www.Micrium.com

Summary
Mutual Exclusion Semaphores or simply *mutexes* are used by tasks to gain exclusive access to a resource. Mutexes are *binary semaphores* that have additional features beyond the normal semaphores mechanism provided by µC/OS-II. This application note describes the mutex series of services which were added to µC/OS-II V2.04.

Introduction
A mutex is used by your application code to reduce the priority inversion problem as described in the book: *MicroC/OS-II, The Real-Time kernel* (ISBN 0-87930-543-6), section 2.16, page 47. A priority inversion occurs when a low priority task owns a resource needed by a high priority task. In order to reduce priority inversion, the kernel can increase the priority of the low priority task to the priority of the higher priority task until the low priority task is done with the resource.

In order to implement mutexes, a real-time kernel needs to provide the ability to support multiple tasks at the same priority. Unfortunately, µC/OS-II doesn't allow to have multiple tasks at the same priority. However, there is a way around this problem. What if a priority just above the high priority task was *reserved* by the mutex to allow a low priority task to be raised in priority.

Let's use an example to illustrate how µC/OS-II mutexes work. Listing 1 shows three tasks that may need to access a common resource. To access the resource, each task must pend on the mutex `ResourceMutex`. Task #1 has the highest priority (10), task #2 has a medium priority (15) and task #3, the lowest (20). An unused priority just above the highest task priority (i.e. priority 9) will be reserved as the *Priority Inheritance Priority (PIP)*. As shown in `main()`, µC/OS-II is initialized L1(1) and a mutex is created by calling `OSMutexCreate()` L1(2). You should note that `OSMutexCreate()` is passed the PIP. The three tasks are then created L1(3) and µC/OS-II is started L1(4).

Suppose that this application has been running for a while and that, at some point, task #3 accesses the common resource first and thus acquires the mutex. Task #3 runs for a while and then gets preempted by task #1. Task #1 needs the resource and thus attempts to acquire the mutex (by calling `OSMutexPend()`). In this case, `OSMutexPend()` notices that a higher priority task needs the resource and thus raises the priority of task #3 to 9 which forces a context switch back to task #3. Task #3 will proceed and hopefully release the resource quickly. When done with the resource, task #3 will call `OSMutexPost()` to release the mutex. `OSMutexPost()` will notice that the mutex was *owned* by a lower priority task that got its priority raised and thus, will return task #3 to it's original priority. `OSMutexPost()` will notice that a higher priority task (i.e. task #1) needs access to the resource and will give the resource to task #1 and perform a context switch to task #1.

```
OS_EVENT *ResourceMutex;
OS_STK    TaskPrio10Stk[1000];
OS_STK    TaskPrio15Stk[1000];
OS_STK    TaskPrio20Stk[1000];


void main (void)
{
    INT8U err;


    OSInit();                                                          /* (1)     */
    /* ---------- Application Initialization ---------- */
    OSMutexCreate(9, &err);                                            /* (2)     */
    OSTaskCreate(TaskPrio10, (void *)0, &TaskPrio10Stk[999], 10);      /* (3)     */
    OSTaskCreate(TaskPrio15, (void *)0, &TaskPrio15Stk[999], 15);
    OSTaskCreate(TaskPrio20, (void *)0, &TaskPrio20Stk[999], 20);
    /* ---------- Application Initialization ---------- */
    OSStart();                                                         /* (4)     */
}


void TaskPrio10 (void *pdata)
{
    INT8U err;


    pdata = pdata;
    while (1) {
        /* --------- Application Code ---------- */
        OSMutexPend(ResourceMutex, 0, &err);
        /* ------- Access common resource ------ */
        OSMutexPost(ResourceMutex);
        /* --------- Application Code ---------- */
    }
}


void TaskPrio15 (void *pdata)
{
    INT8U err;


    pdata = pdata;
    while (1) {
        /* --------- Application Code ---------- */
        OSMutexPend(ResourceMutex, 0, &err);
        /* ------- Access common resource ------ */
        OSMutexPost(ResourceMutex);
        /* --------- Application Code ---------- */
    }
}


void TaskPrio20 (void *pdata)
{
    INT8U err;


    pdata = pdata;
    while (1) {
        /* --------- Application Code ---------- */
        OSMutexPend(ResourceMutex, 0, &err);
        /* ------- Access common resource ------ */
        OSMutexPost(ResourceMutex);
        /* --------- Application Code ---------- */
    }
}
```

Listing 1, Mutex utilization example

µC/OS-II's mutexes consist of three elements: a flag indicating whether the mutex is available (`0` or `1`), a priority to assign the task that owns the mutex in case a higher priority task attempts to gain access to the mutex, and a list of tasks waiting for the mutex. To enable µC/OS-II's mutex services, you must set the configuration constant `OS_MUTEX_EN` to `1` (see file `OS_CFG.H`).

A mutex needs to be created before it can be used. Creating a mutex is accomplished by calling `OSMutexCreate()` (see next section). The initial value of a mutex is always set to `1` indicating that the resource is available.

µC/OS-II provides six services to access mutexes: `OSMutexCreate()`, `OSMutexDel()`, `OSMutexPend()`, `OSMutexPost()`, `OSMutexAccept()` and `OSMutexQuery()`. Figure `1` shows a flow diagram to illustrate the relationship between tasks and a mutex. A mutex can only be accessed by tasks. Note that the symbology used to represent a mutex is a 'key'. The 'key' symbology shows that the mutex is used to access shared resources.

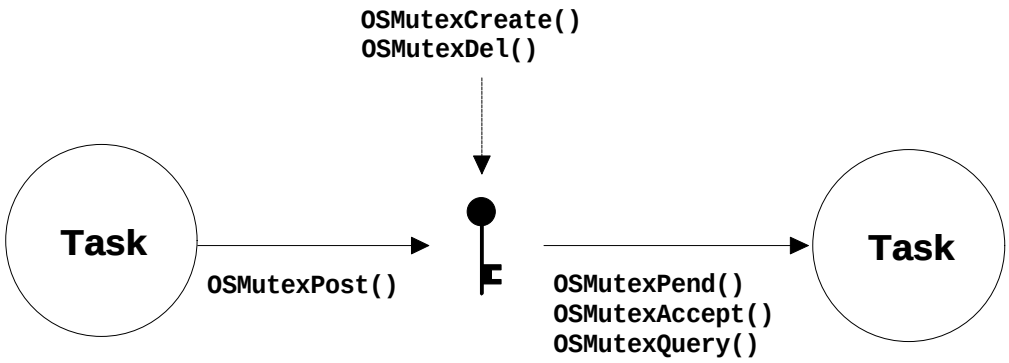

Figure 1, Relationship between tasks and a mutex.

Creating a Mutex, OSMutexCreate()

The code to create a mutex is shown in listing 2. `OSMutexCreate()` starts by making sure it's not called from an ISR because that's not allowed L2(1).

`OSMutexCreate()` then verifies that the PIP is within valid ranged L2(2) based on what you determined the lowest priority is for your application as specified in `OS_CFG.H`.

`OSMutexCreate()` then checks to see there isn't already a task assigned to the PIP L2(3). A non-`NULL` pointer in `OSTCBPrioTbl[]` indicates for the Priority Inheritance Priority (PIP) is available.

If an entry is available, `OSMutexCreate()` *reserves* the priority by placing a non-`NULL` pointer in `OSTCBPrioTbl[prio]` L2(4).

`OSMutexCreate()` then attempts to obtain an ECB (Event Control Block) from the free list of ECBs L2(5) (see µC/OS-II book, figure 6-3, page 144).

The linked list of free ECBs is adjusted to point to the next free ECB L2(6).

If there was an ECB available, the ECB type is set to `OS_EVENT_TYPE_MUTEX` L2(7). Other µC/OS-II services will check this field to make sure that the ECB is of the proper type. This prevents you from calling `OSMutexPost()` on an ECB that was created for use as a message mailbox.

`OSMutexCreate()` then set the mutex value to 'available' and the PIP is stored L2(8).

It is worth noting that the `.OSEventCnt` field is used differently. Specifically, the upper 8 bits of `.OSEventCnt` are used to hold the PIP and the lower 8 bits are used to hold either the value of the mutex when the resource is available (`0xFF`) or, the priority of the task that 'owns' the mutex (a value between `0` and 62). This prevents having to add extra fields in an `OS_EVENT` structure and thus reduces the amount of RAM.

Because the mutex is being initialized, there are no tasks waiting for it L2(9).

The wait list is then initialized by calling `OSEventWaitListInit()` L2(10).

Finally, `OSMutexCreate()` returns a pointer to the ECB L2(11). This pointer MUST be used in subsequent calls to manipulate mutexes (`OSMutexPend()`, `OSMutexPost()`, `OSMutexAccept()`, `OSMutexDel()` and `OSMutexQuery()`). The pointer is basically used as the mutex's handle. If there were no more ECBs, `OSMutexCreate()` would have returned a `NULL` pointer.

```
OS_EVENT *OSMutexCreate (INT8U prio, INT8U *err)
{
#if OS_CRITICAL_METHOD == 3                                 /* Allocate storage for CPU status register      */
    OS_CPU_SR  cpu_sr;
#endif
    OS_EVENT *pevent;


    if (OSIntNesting > 0) {                                 /*  (1) See if called from ISR ...               */
        *err = OS_ERR_CREATE_ISR;                           /*      ... can't CREATE mutex from an ISR       */
        return ((OS_EVENT *)0);
    }
#if OS_ARG_CHK_EN
    if (prio >= OS_LOWEST_PRIO) {                           /*  (2) Validate PIP                             */
        *err = OS_PRIO_INVALID;
        return ((OS_EVENT *)0);
    }
#endif
    OS_ENTER_CRITICAL();
    if (OSTCBPrioTbl[prio] != (OS_TCB *)0) {                /*  (3) Mutex priority must not already exist    */
        *err = OS_PRIO_EXIST;                               /*      Task already exist at priority ...       */
        OS_EXIT_CRITICAL();                                 /*      ... inheritance priority                 */
        return ((OS_EVENT *)0);
    }
    OSTCBPrioTbl[prio] = (OS_TCB *)1;                       /*  (4) Reserve the table entry                  */
    pevent             = OSEventFreeList;                   /*  (5) Get next free event control block        */
    if (pevent == (OS_EVENT *)0) {                          /*      See if an ECB was available              */
        OSTCBPrioTbl[prio] = (OS_TCB *)0;                   /*      No, Release the table entry              */
        OS_EXIT_CRITICAL();
        *err               = OS_ERR_PEVENT_NULL;            /*      No more event control blocks             */
        return (pevent);
    }
    OSEventFreeList     = (OS_EVENT *)OSEventFreeList->OSEventPtr;   /* (6)  Adjust the free list           */
    OS_EXIT_CRITICAL();
    pevent->OSEventType = OS_EVENT_TYPE_MUTEX;              /*  (7) ECB is used as a MUTEX                   */
    pevent->OSEventCnt  = (prio << 8) | OS_MUTEX_AVAILABLE;/*  (8) Resource is available                    */
    pevent->OSEventPtr  = (void *)0;                        /*  (9) No task owning the mutex                 */
    OSEventWaitListInit(pevent);                            /* (10) Initialize the ECB                       */
    *err                = OS_NO_ERR;
    return (pevent);                                        /* (11)                                          */


    }
}
```

Listing 2, Creating a mutex.

Deleting a Mutex, OSMutexDel()

The code to delete a mutex is shown in listing 3. This is a dangerous function to use because multiple tasks could attempt to access a deleted mutex. You should always use this function with great care. Generally speaking, before you would delete a mutex, you would first delete all the tasks that access the mutex.

`OSMutexDel()` starts by making sure that this function is not called from an ISR because that's not allowed L3(1).

We then check the arguments passed to it L3(2) – `pevent` cannot be a `NULL` pointer and `pevent` needs to point to a mutex L3(3).

`OSMutexDel()` then determines whether there are any tasks waiting on the mutex. The flag `tasks_waiting` is set accordingly L3(4).

Based on the option (i.e. `opt`) specified in the call, `OSMutexDel()` will either delete the mutex only if no tasks are pending on the mutex (`opt == OS_DEL_NO_PEND`) or, delete the mutex even if tasks are waiting (`opt == OS_DEL_ALWAYS`).

> When `opt` is set to `OS_DEL_NO_PEND` and there is no task waiting on the mutex L3(5), `OSMutexDel()` marks the ECB as unused L3(6) and the ECB is returned to the free list of ECBs L3(7). This will allow another mutex (or any other ECB based object) to be created. You will note that `OSMutexDel()` returns a `NULL` pointer L3(8) since, at this point, the mutex should no longer be accessed through the original pointer.
>
> When `opt` is set to `OS_DEL_ALWAYS` L3(9) then all tasks waiting on the mutex will be readied L3(10). Each task will *think* it has access to the mutex. Of course, that's a dangerous outcome since the whole point of having a mutex is to protect against multiple access of a resource. Once all pending tasks are readied, `OSMutexDel()` marks the ECB as unused L3(11) and the ECB is returned to the free list of ECBs L3(12). The scheduler is called only if there were tasks waiting on the mutex L3(13). You will note that `OSMutexDel()` returns a `NULL` pointer L3(14) since, at this point, the mutex should no longer be accessed through the original pointer.

```
OS_EVENT  *OSMutexDel (OS_EVENT *pevent, INT8U opt, INT8U *err)
{
#if OS_CRITICAL_METHOD == 3                                 /* Allocate storage for CPU status register      */
    OS_CPU_SR  cpu_sr;
#endif
    BOOLEAN    tasks_waiting;


    if (OSIntNesting > 0) {                                 /*  (1) See if called from ISR ...               */
        *err = OS_ERR_DEL_ISR;                              /*      ... can't DELETE from an ISR             */
        return (pevent);
    }
#if OS_ARG_CHK_EN
    if (pevent == (OS_EVENT *)0) {                          /*  (2) Validate 'pevent'                        */
        *err = OS_ERR_PEVENT_NULL;
        return (pevent);
    }
#endif
    OS_ENTER_CRITICAL();
#if OS_ARG_CHK_EN
    if (pevent->OSEventType != OS_EVENT_TYPE_MUTEX) {       /*  (3) Validate event block type               */
        OS_EXIT_CRITICAL();
        *err = OS_ERR_EVENT_TYPE;
        return (pevent);
    }
#endif
    if (pevent->OSEventGrp != 0x00) {                       /*  (4) See if any tasks waiting on mutex        */
        tasks_waiting = TRUE;                               /*      Yes                                      */
    } else {
        tasks_waiting = FALSE;                              /*      No                                       */
    }
    switch (opt) {
        case OS_DEL_NO_PEND:
             if (tasks_waiting == FALSE) {                  /*  (5) Delete mutex only if no task waiting     */
                 pevent->OSEventType = OS_EVENT_TYPE_UNUSED;/*  (6) Mark the ECB as unused                   */
                 pevent->OSEventPtr  = OSEventFreeList;     /*  (7) Return Event Control Block to free list  */
                 OSEventFreeList     = pevent;
                 OS_EXIT_CRITICAL();
                 *err = OS_NO_ERR;
                 return ((OS_EVENT *)0);                    /*  (8) Mutex has been deleted                   */
             } else {
                 OS_EXIT_CRITICAL();
                 *err = OS_ERR_TASK_WAITING;
                 return (pevent);
             }

        case OS_DEL_ALWAYS:                                 /*  (9) Always delete the mutex                 */
             while (pevent->OSEventGrp != 0x00) {           /* (10) Ready ALL tasks waiting for mutex        */
                 OSEventTaskRdy(pevent, (void *)0, OS_STAT_MUTEX);
             }
             pevent->OSEventType = OS_EVENT_TYPE_UNUSED;    /* (11) Mark the ECB as unused                   */
             pevent->OSEventPtr  = OSEventFreeList;         /* (12) Return Event Control Block to free list  */
             OSEventFreeList     = pevent;
             OS_EXIT_CRITICAL();
             if (tasks_waiting == TRUE) {                   /* (13) Reschedule only if task(s) were waiting  */
                 OSSched();                                 /*      Find highest priority task ready to run  */
             }
             *err = OS_NO_ERR;
             return ((OS_EVENT *)0);                        /* (14) Mutex has been deleted                   */

        default:
             OS_EXIT_CRITICAL();
             *err = OS_ERR_INVALID_OPT;
             return (pevent);
    }
}
```

Listing 3, Deleting a mutex.

Waiting on a Mutex, OSMutexPend()

The code to wait on a mutex is shown in listing 4.

Like all µC/OS-II pend calls, `OSMutexPend()` cannot be called from an ISR and thus, `OSMutexPend()` checks for this condition first L4(1).

Assuming that the configuration constant `OS_ARG_CHK_EN` is set to `1`, `OSMutexPend()` makes sure that the 'handle' `pevent` is not a `NULL` pointer L4(2) and that the ECB being pointed to has been created by `OSMutexCreate()` L4(3).

If the mutex is available, the lower 8 bits of `.OSEventCnt` is set to `0xFF` (i.e. `OS_MUTEX_AVAILABLE`) L4(4). If this is the case, `OSMutexPend()` will grant the mutex to the calling task and, `OSMutexPend()` will set the lower 8 bits of `.OSEventCnt` to the calling's task priority L4(5) and L4(6). `OSMutexPend()` then sets `.OSEventPtr` to point to the TCB of the calling task L4(7) and returns. At this point the caller can proceed with accessing the resource since the return error code is set to `OS_NO_ERR`. Obviously, if you want the mutex, this is the outcome you are looking for. This also happens to be the fastest (normal) path through `OSMutexPend()`.

If the mutex is owned by another task, the calling task needs to be put to sleep until another task relinquishes the mutex (see `OSMutexPost()`). `OSMutexPend()` allows you to specify a timeout value as one of its arguments (i.e. `timeout`). This feature is useful to avoid waiting indefinitely for the mutex. If the value passed is non-zero, then `OSMutexPend()` will suspend the task until the mutex is signaled or the specified timeout period expires. Note that a `timeout` value of `0` indicates that the task is willing to wait forever for the mutex to be signaled. Before the calling task is put to sleep, `OSMutexPend()` extracts the PIP of the mutex L4(8), the priority of the task that owns the mutex L4(9) and a pointer to the TCB of the task that owns the mutex L4(10). If the owner's priority is *lower* (a higher number) than the task that calls `OSMutexPend()` L4(11) then, the priority of the task that owns the mutex will be raised to the mutex's priority inheritance priority (PIP) L4(8). This will allow the owner to relinquish the mutex sooner.

`OSMutexPend()` then determines if the task that owns the mutex is ready-to-run L4(12). If it is, that task will be made no longer ready-to-run at the the owner's priority L4(13) and the flag `rdy` will be set indicating that the mutex owner was ready-to-run L4(14). If the task was not ready-to-run, `rdy` is set accordingly L4(15). The reason the flag is set is to determine whether we need make the task ready-to-run at the new, higher priority (i.e. at the PIP).

`OSMutexPend()` then computes TCB elements at the PIP L4(16). You should note that I could have saved this information in the `OS_EVENT` data structure when the mutex was created in order to save processing time. However, this would have meant additional RAM for each `OS_EVENT` instantiation. From this information and the sate of the `rdy` flag, we determine whether the mutex owner needs to be made ready-to-run at the PIP L4(17).

To put the calling task to sleep, `OSMutexPend()` sets the status flag in the task's TCB (Task Control Block) to indicate that the task is suspended waiting for a mutex L4(18). The timeout is also stored in the TCB L4(19) so that it can be decremented by `OSTimeTick()`. You should recall (see section 3.10, *Clock Tick*) that `OSTimeTick()` decrements each of the created task's `.OSTCBDly` field if it's non-zero. The actual work of putting the task to sleep is done by `OSEventTaskWait()` (see section 6.03, *Making a task wait for an event, OSEventTaskWait()*) L4(20).

Because the calling task is no longer ready-to-run, the scheduler is called to run the next highest priority task that is ready-to-run L4(21).

When the mutex is signaled (or the timeout period expires) and the task that called `OSMutexPend()` is again the highest priority task, `OSSched()` returns. `OSMutexPend()` then checks to see if the TCB's status flag is still set to indicate that the task is waiting for the mutex L4(22). If the task is still waiting for the mutex then it must not have been signaled by an `OSMutexPost()` call. Indeed, the task must have be readied by `OSTimeTick()` indicating that the timeout period has expired. In this case, the task is removed from the wait list for the mutex by calling `OSEventTO()` L4(23), and an error code is returned L4(24) to the task that called `OSMutexPend()` to indicate that a timeout occurred.

If the status flag in the task's TCB doesn't have the `OS_STAT_MUTEX` bit set then the mutex must have been signaled and the task that called `OSMutexPend()` can now conclude that it has the mutex. Note that the link to the ECB is removed L4(25).

```
void OSMutexPend (OS_EVENT *pevent, INT16U timeout, INT8U *err)
{
#if OS_CRITICAL_METHOD == 3                                 /* Allocate storage for CPU status register      */
    OS_CPU_SR  cpu_sr;
#endif
    INT8U      pip;                                         /* Priority Inheritance Priority (PIP)           */
    INT8U      mprio;                                       /* Mutex owner priority                          */
    BOOLEAN    rdy;                                         /* Flag indicating task was ready                */
    OS_TCB    *ptcb;


    if (OSIntNesting > 0) {                                 /*  (1) See if called from ISR ...               */
        *err = OS_ERR_PEND_ISR;                             /*      ... can't PEND from an ISR               */
        return;
    }
#if OS_ARG_CHK_EN
    if (pevent == (OS_EVENT *)0) {                          /*  (2) Validate 'pevent'                        */
        *err = OS_ERR_PEVENT_NULL;
        return;
    }
#endif
    OS_ENTER_CRITICAL();
#if OS_ARG_CHK_EN
    if (pevent->OSEventType != OS_EVENT_TYPE_MUTEX) {       /*  (3) Validate event block type                */
        OS_EXIT_CRITICAL();
        *err = OS_ERR_EVENT_TYPE;
        return;
    }
#endif
                                                            /*  (4) Is Mutex available?                      */
    if ((INT8U)(pevent->OSEventCnt & OS_MUTEX_KEEP_LOWER_8) == OS_MUTEX_AVAILABLE) {
        pevent->OSEventCnt &= OS_MUTEX_KEEP_UPPER_8;        /*  (5) Yes, Acquire the resource                */
        pevent->OSEventCnt |= OSTCBCur->OSTCBPrio;          /*  (6)      Save priority of owning task        */
        pevent->OSEventPtr  = (void *)OSTCBCur;             /*  (7)      Point to owning task's OS_TCB       */
        OS_EXIT_CRITICAL();
        *err  = OS_NO_ERR;
        return;
    }
```

```
    pip   = (INT8U)(pevent->OSEventCnt >> 8);                    /*  (8) No, Get PIP from mutex           */
    mprio = (INT8U)(pevent->OSEventCnt & OS_MUTEX_KEEP_LOWER_8); /*  (9)     Get priority of mutex owner  */
    ptcb  = (OS_TCB *)(pevent->OSEventPtr);                      /* (10)     Point to TCB of mutex owner  */

    if (ptcb->OSTCBPrio != pip && mprio > OSTCBCur->OSTCBPrio) {  /* (11)     Need to promote prio of owner?*/
        if ((OSRdyTbl[ptcb->OSTCBY] & ptcb->OSTCBBitX) != 0x00) { /* (12)     See if mutex owner is ready  */
                                                                  /* (13)     Yes, Remove owner from Rdy ...*/
                                                                  /*               ... list at current prio */
            if ((OSRdyTbl[ptcb->OSTCBY] &= ~ptcb->OSTCBBitX) == 0x00) {
                OSRdyGrp &= ~ptcb->OSTCBBitY;
            }
            rdy = TRUE;                                           /* (14)                                  */
        } else {
            rdy = FALSE;                                          /* (15) No                               */
        }
        ptcb->OSTCBPrio         = pip;                    /* (16) Change owner task prio to PIP          */
        ptcb->OSTCBY            = ptcb->OSTCBPrio >> 3;
        ptcb->OSTCBBitY         = OSMapTbl[ptcb->OSTCBY];
        ptcb->OSTCBX            = ptcb->OSTCBPrio & 0x07;
        ptcb->OSTCBBitX         = OSMapTbl[ptcb->OSTCBX];
        if (rdy == TRUE) {                                    /* (17) If task was ready at owner's priority ...*/
            OSRdyGrp               |= ptcb->OSTCBBitY;        /*      ... make it ready at new priority.       */
            OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
        }
        OSTCBPrioTbl[pip]       = (OS_TCB *)ptcb;
    }
    OSTCBCur->OSTCBStat |= OS_STAT_MUTEX;             /* (18) Mutex not available, pend current task       */
    OSTCBCur->OSTCBDly   = timeout;                   /* (19) Store timeout in current task's TCB          */
    OSEventTaskWait(pevent);                          /* (20) Suspend task until event or timeout occurs   */
    OS_EXIT_CRITICAL();
    OSSched();                                        /* (21) Find next highest priority task ready        */
    OS_ENTER_CRITICAL();
    if (OSTCBCur->OSTCBStat & OS_STAT_MUTEX) {        /* (22) Must have timed out if still waiting for event*/
        OSEventTO(pevent);                            /* (23)                                              */
        OS_EXIT_CRITICAL();
        *err = OS_TIMEOUT;                            /* (24) Indicate that we didn't get mutex within TO  */
        return;
    }
    OSTCBCur->OSTCBEventPtr = (OS_EVENT *)0;          /* (25)                                              */
    OS_EXIT_CRITICAL();
    *err = OS_NO_ERR;
}
```

Listing 4, Waiting for a mutex.

Signaling a Mutex, OSMutexPost()

The code to signal a mutex is shown in listing 5.

Mutual exclusion semaphores must only be used by tasks and thus, a check is performed to see if `OSMutexPost()` is called from an ISR L5(1).

Assuming that the configuration constant `OS_ARG_CHK_EN` is set to `1`, `OSMutexPost()` checks that the 'handle' `pevent` is not a `NULL` pointer L5(2) and that the ECB being pointed to has been created by `OSMutexCreate()` L5(3). Finally, `OSMutexPost()` makes sure that the task that is signaling the mutex actually owns the mutex. The owner's priority must either be set to the `pip` (`OSMutexPend()` could have raised the owner's priority) or the priority stored in the mutex itself L5(4).

`OSMutexPost()` then checks to see if the priority of the mutex owner had to be raised to the PIP L5(5) because a higher priority task attempted to access the mutex. In this case, the priority of the owner is reduced back to its original value. The original task priority was extracted from the lower 8 bits of `.OSEventCnt`. The calling task is removed from the ready list at the PIP and placed in the ready list at the task's original priority L5(6). Note that the TCB fields are recomputed for the original task priority.

Next, we check to see if any tasks are waiting on the mutex L5(7). There are tasks waiting when the `.OSEventGrp` field in the ECB contains a non-zero value. The highest priority task waiting for the mutex will be removed from the wait list by `OSEventTaskRdy()` (see section 6.02, *Making a task ready, OSEventTaskRdy()*) L5(8) and this task will be made ready-to-run. The priority of the new owner is saved in the mutex's ECB L5(9). `OSSched()` is then called to see if the task made ready is now the highest priority task ready-to-run L5(10). If it is, a context switch will result and the readied task will be executed. If the readied task is not the highest priority task then `OSSched()` will return and the task that called `OSMutexPost()` will continue execution. If there were no tasks waiting on the mutex, the lower 8 bits of `.OSEventCnt` would be set to `0xFF` L5(11) indicating that the mutex is immediately available.

```
INT8U OSMutexPost (OS_EVENT *pevent)
{
#if OS_CRITICAL_METHOD == 3                          /*      Allocate storage for CPU status register      */
    OS_CPU_SR  cpu_sr;
#endif
    INT8U      pip;                                  /*      Priority inheritance priority                 */
    INT8U      prio;


    if (OSIntNesting > 0) {                          /*  (1) See if called from ISR ...                    */
        return (OS_ERR_POST_ISR);
    }
#if OS_ARG_CHK_EN
    if (pevent == (OS_EVENT *)0) {                   /*  (2) Validate 'pevent'                             */
        return (OS_ERR_PEVENT_NULL);
    }
#endif
    OS_ENTER_CRITICAL();
    pip  = (INT8U)(pevent->OSEventCnt >> 8);         /*      Get priority inheritance priority of mutex    */
    prio = (INT8U)(pevent->OSEventCnt & OS_MUTEX_KEEP_LOWER_8);  /* Get owner's original priority          */
#if OS_ARG_CHK_EN
    if (pevent->OSEventType != OS_EVENT_TYPE_MUTEX) { /*  (3) Validate event block type                    */
        OS_EXIT_CRITICAL();
        return (OS_ERR_EVENT_TYPE);
    }
    if (OSTCBCur->OSTCBPrio != pip ||
        OSTCBCur->OSTCBPrio != prio) {               /*  (4) See if posting task owns the MUTEX            */
        OS_EXIT_CRITICAL();
        return (OS_ERR_NOT_MUTEX_OWNER);
    }
#endif
    if (OSTCBCur->OSTCBPrio == pip) {                /*  (5) Did we have to raise current task's priority? */
                                                     /*      Yes, Return to original priority              */
                                                     /*  (6)      Remove owner from ready list at 'pip'    */
        if ((OSRdyTbl[OSTCBCur->OSTCBY] &= ~OSTCBCur->OSTCBBitX) == 0) {
            OSRdyGrp &= ~OSTCBCur->OSTCBBitY;
        }
        OSTCBCur->OSTCBPrio         = prio;
        OSTCBCur->OSTCBY            = prio >> 3;
        OSTCBCur->OSTCBBitY         = OSMapTbl[OSTCBCur->OSTCBY];
        OSTCBCur->OSTCBX            = prio & 0x07;
        OSTCBCur->OSTCBBitX         = OSMapTbl[OSTCBCur->OSTCBX];
        OSRdyGrp                   |= OSTCBCur->OSTCBBitY;
        OSRdyTbl[OSTCBCur->OSTCBY] |= OSTCBCur->OSTCBBitX;
        OSTCBPrioTbl[prio]          = (OS_TCB *)OSTCBCur;
    }
    OSTCBPrioTbl[pip] = (OS_TCB *)1;                 /*      Reserve table entry                           */
    if (pevent->OSEventGrp) {                        /*  (7) Any task waiting for the mutex?               */
                                                     /*  (8) Yes, Make HPT waiting for mutex ready         */
        prio               = OSEventTaskRdy(pevent, (void *)0, OS_STAT_MUTEX);
        pevent->OSEventCnt &= 0xFF00;                /*  (9)      Save priority of mutex's new owner       */
        pevent->OSEventCnt |= prio;
        pevent->OSEventPtr  = OSTCBPrioTbl[prio];    /*           Link to mutex owner's OS_TCB             */
        OS_EXIT_CRITICAL();
        OSSched();                                   /* (10)      Find highest priority task ready to run  */
        return (OS_NO_ERR);
    }
    pevent->OSEventCnt |= 0x00FF;                    /* (11) No,  Mutex is now available                   */
    pevent->OSEventPtr  = (void *)0;
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);
}
```

Listing 5, Signaling a mutex.

Getting a Mutex without waiting, OSMutexAccept()

It is possible to obtain a mutex without putting a task to sleep if the mutex is not available. This is accomplished by calling `OSMutexAccept()` and the code for this function is shown in listing 6. As with the other calls, `OSMutexAccept()` start by ensuring that it's not called from and ISR and performs boundary checks L6(1).

`OSMutexAccept()` then checks to see if the mutex is available (the lower 8 bits of `.OSEventCnt` would be set to `0xFF`) L6(2). If the mutex is available, `OSMutexAccept()` would acquire the mutex by writing the priority of the mutex owner in the lower 8 bits of `.OSEventCnt` L6(3) and by linking the the owner's TCB L6(4).

The code that called `OSMutexAccept()` will need to examine the returned value. A returned value of `0` indicates that the mutex was not available while a return value of `1` indicates that the mutex was available and the caller can access the resource.

```
INT8U OSMutexAccept (OS_EVENT *pevent, INT8U *err)
{
#if OS_CRITICAL_METHOD == 3                          /*     Allocate storage for CPU status register     */
    OS_CPU_SR  cpu_sr;
#endif


    if (OSIntNesting > 0) {                          /* (1) Make sure it's not called from an ISR        */
        *err = OS_ERR_PEND_ISR;
        return (0);
    }
#if OS_ARG_CHK_EN
    if (pevent == (OS_EVENT *)0) {                   /*     Validate 'pevent'                            */
        *err = OS_ERR_PEVENT_NULL;
        return (0);
    }
#endif
    OS_ENTER_CRITICAL();
#if OS_ARG_CHK_EN
    if (pevent->OSEventType != OS_EVENT_TYPE_MUTEX) {  /*     Validate event block type                    */
        OS_EXIT_CRITICAL();
        *err = OS_ERR_EVENT_TYPE;
        return (0);
    }
#endif
    OS_ENTER_CRITICAL();
                                                     /* (2) Get value (0 or 1) of Mutex                  */
    if ((pevent->OSEventCnt & OS_MUTEX_KEEP_LOWER_8) == OS_MUTEX_AVAILABLE) {
        pevent->OSEventCnt &= OS_MUTEX_KEEP_UPPER_8;   /* (3)      Mask off LSByte (Acquire Mutex)         */
        pevent->OSEventCnt |= OSTCBCur->OSTCBPrio;     /*          Save current task priority in LSByte    */
        pevent->OSEventPtr  = (void *)OSTCBCur;        /* (4)      Link TCB of task owning Mutex           */
        OS_EXIT_CRITICAL();
        *err = OS_NO_ERR;
        return (1);
    }
    OS_EXIT_CRITICAL();
    *err = OS_NO_ERR;
    return (0);
}
```

Listing 6, Getting a mutex without waiting.

Obtaining the status of a mutex, OSMutexQuery()

`OSMutexQuery()` allows your application to take a 'snapshot' of an ECB that is used as a mutex. The code for this function is shown in listing 7.

`OSMutexQuery()` is passed two arguments: `pevent` contains a pointer to the mutex which is returned by `OSMutexCreate()` when the mutex is created and, `pdata` which is a pointer to a data structure (`OS_MUTEX_DATA`, see `uCOS_II.H`) that will hold information about the mutex. Your application will thus need to allocate a variable of type `OS_MUTEX_DATA` that will be used to receive the information about the desired mutex. I decided to use a new data structure because the caller should only be concerned with mutex specific data as opposed to the more generic `OS_EVENT` data structure. `OS_MUTEX_DATA` contains the mutex PIP (Priority Inheritance Priority) (`.OSMutexPIP`), the priority of the task owning the mutex (`.OSMutexPrio`) and the value of the mutex (`.OSMutexValue`) which is set to `1` when the mutex is available and `0` if it's not. Note that `.OSMutexPrio` contains `0xFF` if no task owns the mutex. Finally, `OS_MUTEX_DATA` contains the list of tasks waiting on the mutex (`.OSEventTbl[]` and `.OSEventGrp`).

As with all mutex calls, `OSMutexQuery()` determines whether the call is made from an ISR L7(1). If the configuration constant `OS_ARG_CHK_EN` is set to `1`, `OSMutexQuery()` checks that the 'handle' `pevent` is not a `NULL` pointer L7(2) and that the ECB being pointed to has been created by `OSMutexCreate()` L7(3). `OSMutexQuery()` then loads the `OS_MUTEX_DATA` structure with the appropriate fields. First, we extract the Priority Inheritance Priority (PIP) from the upper 8 bits of the `.OSEventCnt` field of the mutex L7(4). Next, we obtain the mutex value from the lower 8 bits of the `.OSEventCnt` field of the mutex. If the mutex is available (i.e. lower 8 bits set to `0xFF`) then the mutex value is assumed to be `1` L7(5). Otherwise, the mutex value is `0` (i.e. unavailable because it's owned by a task) L7(6). Finally, the mutex wait list is copied into the appropriate fields in `OS_MUTEX_DATA` L7(7).

```
INT8U OSMutexQuery (OS_EVENT *pevent, OS_MUTEX_DATA *pdata)
{
#if OS_CRITICAL_METHOD == 3                                /* Allocate storage for CPU status register    */
    OS_CPU_SR  cpu_sr;
#endif
    INT8U     *psrc;
    INT8U     *pdest;


    if (OSIntNesting > 0) {                                /* (1) See if called from ISR ...              */
        return (OS_ERR_QUERY_ISR);
    }
#if OS_ARG_CHK_EN
    if (pevent == (OS_EVENT *)0) {                         /* (2) Validate 'pevent'                       */
        return (OS_ERR_PEVENT_NULL);
    }
#endif
    OS_ENTER_CRITICAL();
#if OS_ARG_CHK_EN
    if (pevent->OSEventType != OS_EVENT_TYPE_MUTEX) {      /* (3) Validate event block type               */
        OS_EXIT_CRITICAL();
        return (OS_ERR_EVENT_TYPE);
    }
#endif
    pdata->OSMutexPIP  = (INT8U)(pevent->OSEventCnt >> 8); /* (4)                                         */
    pdata->OSOwnerPrio = (INT8U)(pevent->OSEventCnt & 0x00FF);
    if (pdata->OSOwnerPrio == 0xFF) {
        pdata->OSValue = 1;                                /* (5)                                         */
    } else {
        pdata->OSValue = 0;                                /* (6)                                         */
    }
    pdata->OSEventGrp  = pevent->OSEventGrp;               /* (7) Copy wait list                          */
    psrc               = &pevent->OSEventTbl[0];
    pdest              = &pdata->OSEventTbl[0];
#if OS_EVENT_TBL_SIZE > 0
    *pdest++           = *psrc++;
#endif

#if OS_EVENT_TBL_SIZE > 1
    *pdest++           = *psrc++;
#endif

#if OS_EVENT_TBL_SIZE > 2
    *pdest++           = *psrc++;
#endif

#if OS_EVENT_TBL_SIZE > 3
    *pdest++           = *psrc++;
#endif

#if OS_EVENT_TBL_SIZE > 4
    *pdest++           = *psrc++;
#endif

#if OS_EVENT_TBL_SIZE > 5
    *pdest++           = *psrc++;
#endif

#if OS_EVENT_TBL_SIZE > 6
    *pdest++           = *psrc++;
#endif

#if OS_EVENT_TBL_SIZE > 7
    *pdest             = *psrc;
#endif
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);
}
```

Listing 7, Obtaining the status of a mutex.

OSMutexAccept()

INT8U OSMutexAccept(OS_EVENT *pevent, INT8U *err);

File	Called from	Code enabled by
OS_MUTEX.C	Task	OS_MUTEX_EN

`OSMutexAccept()` allows you to check to see if a resource is available. Unlike `OSMutexPend()`, `OSMutexAccept()` does not suspend the calling task if the resource is not available.

Arguments

`pevent` is a pointer to the mutex that guards the resource. This pointer is returned to your application when the mutex is created (see `OSMutexCreate()`).

`err` is a pointer to a variable used to hold an error code. `OSMutexAccept()` sets `*err` to one of the following:

`OS_NO_ERR`	if the call was successful.
`OS_ERR_PEVENT_NULL`	if `pevent` is a `NULL` pointer.
`OS_ERR_EVENT_TYPE`	if `pevent` is not pointing to a mutex.
`OS_ERR_PEND_ISR`	if you called `OSMutexAccept()` from an ISR.

Returned Value

If the mutex was available, `OSMutexAccept()` returns `1`. If the mutex was owned by another task, `OSMutexAccept()` returns `0`.

Notes/Warnings

1) Mutexes must be created before they are used.
2) This function MUST NOT be called by an ISR.
3) If you acquire the mutex through `OSMutexAccept()`, you MUST call `OSMutexPost()` to release the mutex when you are done with the resource.

Example

```
OS_EVENT *DispMutex;


void Task (void *pdata)
{
    INT8U  err;
    INT8U  value;


    pdata = pdata;
    for (;;) {
        value = OSMutexAccept(DispMutex, &err);
        if (value == 1) {
            .                           /* Resource available, process */
            .
        } else {
            .                           /* Resource NOT available      */
            .
        }
        .
        .
    }
}
```

OSMutexCreate()

OS_EVENT *OSMutexCreate(INT8U prio, INT8U *err);

File	Called from	Code enabled by
OS_MUTEX.C	Task or startup code	OS_MUTEX_EN

`OSMutexCreate()` is used to create and initialize a mutex. A mutex is used to gain exclusive access to a resource.

Arguments

`prio` is the Priority Inheritance Priority (PIP) that will be used when a high priority task attempts to acquire the mutex that is owned by a low priority task. In this case, the priority of the low priority task will be *raised* to the PIP until the resource is released.

`err` is a pointer to a variable which will be used to hold an error code. The error code can be one of the following:

`OS_NO_ERR`	if the call was successful and the mutex was created.
`OS_PRIO_EXIST`	if a task at the specified priority inheritance priority already exist.
`OS_PRIO_INVALID`	if you specified a priority with a higher number than `OS_LOWEST_PRIO`.
`OS_ERR_PEVENT_NULL`	if there are no more `OS_EVENT` structures available.
`OS_ERR_CREATE_ISR`	if you attempted to create a mutex from an ISR.

Returned Value

A pointer to the event control block allocated to the mutex. If no event control block is available, `OSMutexCreate()` will return a `NULL` pointer.

Notes/Warnings

1) Mutexes must be created before they are used.
2) You MUST make sure that `prio` has a higher priority than ANY of the tasks that WILL be using the mutex to access the resource. For example, if 3 tasks of priority 20, 25 and 30 are going to use the mutex then, `prio` must be a number LOWER than 20. In addition, there MUST NOT already be a task created at the specified priority.

Example

```
OS_EVENT *DispMutex;


void main (void)
{
    INT8U  err;

    .
    .
    OSInit();                             /* Initialize µC/OS-II          */
    .
    .
    DispMutex = OSMutexCreate(20, &err);  /* Create Display Mutex         */
    .
    .
    OSStart();                            /* Start Multitasking           */
}
```

OSMutexDel()

```
OS_EVENT *OSMutexDel(OS_EVENT *pevent, INT8U opt, INT8U *err);
```

File	Called from	Code enabled by
OS_MUTEX.C	Task	OS_MUTEX_EN and OS_MUTEX_DEL_EN

`OSMutexDel()` is used to delete a mutex. This is a dangerous function to use because multiple tasks could attempt to access a deleted mutex. You should always use this function with great care. Generally speaking, before you would delete a mutex, you would first delete all the tasks that access the mutex.

Arguments

`pevent` is a pointer to the mutex. This pointer is returned to your application when the mutex is created (see `OSMutexCreate()`).

`opt` specifies whether you want to delete the mutex only if there are no pending tasks (`OS_DEL_NO_PEND`) or whether you always want to delete the mutex regardless of whether tasks are pending or not (`OS_DEL_ALWAYS`). In this case, all pending task will be readied.

`err` is a pointer to a variable which will be used to hold an error code. The error code can be one of the following:

`OS_NO_ERR`	if the call was successful and the mutex was deleted.
`OS_ERR_DEL_ISR`	if you attempted to delete a mutex from an ISR.
`OS_ERR_EVENT_TYPE`	if `pevent` is not pointing to a mutex.
`OS_ERR_INVALID_OPT`	if you didn't specify one of the two options mentioned above.
`OS_ERR_TASK_WAITING`	if one or more task were waiting on the mutex and and you specified `OS_DEL_NO_PEND`.
`OS_ERR_PEVENT_NULL`	if there are no more `OS_EVENT` structures available.

Returned Value

A `NULL` pointer if the mutex is deleted or `pevent` if the mutex was not deleted. In the latter case, you would need to examine the error code to determine the reason.

Notes/Warnings

1) You should use this call with care because other tasks may expect the presence of the mutex.

Example

```
OS_EVENT *DispMutex;


void Task (void *pdata)
{
    INT8U  err;


    pdata = pdata;
    while (1) {
        .
        .
        DispMutex = OSMutexDel(DispMutex, OS_DEL_ALWAYS, &err);
        .
        .
    }
}
```

OSMutexPend()

```
void OSMutexPend(OS_EVENT *pevent, INT16U timeout, INT8U *err);
```

File	Called from	Code enabled by
OS_MUTEX.C	Task only	OS_MUTEX_EN

`OSMutexPend()` is used when a task desires to get exclusive access to a resource. If a task calls `OSMutexPend()` and the mutex is available, then `OSMutexPend()` will *give* the mutex to the caller and return to its caller. Note that nothing is actually given to the caller except for the fact that if the `err` is set to `OS_NO_ERR`, the caller can assume that it owns the mutex. However, if the mutex is already owned by another task, `OSMutexPend()` will place the calling task in the wait list for the mutex. The task will thus wait until the task that owns the mutex releases the mutex and thus the resource or, the specified timeout expires. If the mutex is signaled before the timeout expires, µC/OS-II will resume the highest priority task that is waiting for the mutex. Note that if the mutex is owned by a lower priority task then `OSMutexPend()` will raise the priority of the task that owns the mutext to the Priority Inheritance Priority (PIP) as specified when you created the mutex (see `OSMutexCreate()`).

Arguments

`pevent` is a pointer to the mutex. This pointer is returned to your application when the mutex is created (see `OSMutexCreate()`).

`timeout` is used to allow the task to resume execution if the mutex is not signaled (i.e. posted to) within the specified number of clock ticks. A `timeout` value of `0` indicates that the task desires to wait forever for the mutex. The maximum `timeout` is `65535` clock ticks. The timeout value is not synchronized with the clock tick. The timeout count starts being decremented on the next clock tick which could potentially occur immediately.

`err` is a pointer to a variable which will be used to hold an error code. `OSMutexPend()` sets `*err` to either:

`OS_NO_ERR`	if the call was successful and the mutex was available.
`OS_TIMEOUT`	if the mutex was not available within the specified timeout.
`OS_ERR_EVENT_TYPE`	if you didn't pass a pointer to a mutex to `OSMutexPend()`.
`OS_ERR_PEVENT_NULL`	if `pevent` is a `NULL` pointer.
`OS_ERR_PEND_ISR`	if you attempted to acquire the mutex from an ISR.

Returned Value

NONE

Notes/Warnings

1) Mutexes must be created before they are used.
2) You shoud NOT suspend the task that owns the mutex, have the mutex owner *wait* on any other µC/OS-II objects (i.e. semaphore, mailbox or queue) and, you should NOT delay the task that owns the mutex. In other words, your code should *hurry up* and release the resource as soon as possible.

Example

```
OS_EVENT *DispMutex;


void  DispTask (void *pdata)
{
    INT8U  err;


    pdata = pdata;
    for (;;) {
        .
        .
        OSMutexPend(DispMutex, 0, &err);
        .                              /* The only way this task continues is if … */
        .                              /* … the mutex is available or signaled!    */
    }
}
```

OSMutexPost()

```
INT8U OSMutexPost(OS_EVENT *pevent);
```

File	Called from	Code enabled by
OS_MUTEX.C	Task	OS_MUTEX_EN

A mutex is signaled (i.e. released) by calling `OSMutexPost()`. You would call this function only if you acquired the mutex either by first calling `OSMutexAccept()` or `OSMutexPend()`. If the priority of the task that owns the mutex has been raised when a higher priority task attempted to acquire the mutex then the original task priority of the task will be restored. If one or more tasks are waiting for the mutex, the mutex is given to the highest priority task waiting on the mutex. The scheduler is then called to determine if the awakened task is now the highest priority task ready to run and if so, a context switch will be done to run the readied task. If no task is waiting for the mutex, the mutex value is simply set to *available* (`0xFF`).

Arguments

`pevent` is a pointer to the mutex. This pointer is returned to your application when the mutex is created (see `OSMutexCreate()`).

Returned Value

`OSMutexPost()` returns one of these error codes:

`OS_NO_ERR`	if the call was successful and the mutex released.
`OS_ERR_EVENT_TYPE`	if you didn't pass a pointer to a mutex to `OSMutexPost()`.
`OS_ERR_PEVENT_NULL`	if `pevent` is a `NULL` pointer.
`OS_ERR_POST_ISR`	if you attempted to call `OSMutexPost()` from an ISR.
`OS_ERR_NOT_MUTEX_OWNER`	if the task posting (i.e. signaling the mutex) doesn't actually owns the mutex.

Notes/Warnings

1) Mutexes must be created before they are used.
2) You cannot call this function from an ISR.

Example

```
OS_EVENT  *DispMutex;


void  TaskX (void *pdata)
{
    INT8U  err;


    pdata = pdata;
    for (;;) {
        .
        .
        err = OSMutexPost(DispMutex);
        switch (err) {
           case OS_NO_ERR: /* Mutex signaled       */
                .
                .
                break;

           case OS_ERR_EVENT_TYPE:
                .
                .
                break;

           case OS_ERR_PEVENT_NULL:
                .
                .
                break;

           case OS_ERR_POST_ISR:
                .
                .
                break;

        }
        .
        .
    }
}
```

OSMutexQuery()

INT8U OSMutexQuery(OS_EVENT *pevent, OS_MUTEX_DATA *pdata);

File	Called from	Code enabled by
OS_MUTEX.C	Task	OS_MUTEX_EN

`OSMutexQuery()` is used to obtain run-time information about a mutex. Your application must allocate an `OS_MUTEX_DATA` data structure which will be used to receive data from the event control block of the mutex. `OSMutexQuery()` allows you to determine whether any task is waiting on the mutex, how many tasks are waiting (by counting the number of 1s in the `.OSEventTbl[]` field, obtain the Priority Inheritance Priority (PIP) and determine whether the mutex is available (`1`) or not (`0`). Note that the size of `.OSEventTbl[]` is established by the `#define` constant `OS_EVENT_TBL_SIZE` (see `uCOS_II.H`).

Arguments

`pevent` is a pointer to the mutex. This pointer is returned to your application when the mutex is created (see `OSMutexCreate()`).

`pdata` is a pointer to a data structure of type OS_MUTEX_DATA, which contains the following fields:

```
INT8U  OSMutexPIP;                 /* The PIP of the mutex                                          */
INT8U  OSOwnerPrio;                /* The priority of the mutex owner                               */
INT8U  OSValue;                    /* The current mutex value, 1 means available, 0 means unavailable */
INT8U  OSEventGrp;                 /* Copy of the mutex wait list                                   */
INT8U  OSEventTbl[OS_EVENT_TBL_SIZE];
```

Returned Value

`OSMutexQuery()` returns one of these error codes:

`OS_NO_ERR`	if the call was successful.
`OS_ERR_EVENT_TYPE`	if you didn't pass a pointer to a mutex to `OSMutexQuery()`.
`OS_ERR_PEVENT_NULL`	if `pevent` is a `NULL` pointer.
`OS_ERR_QUERY_ISR`	if you attempted to call `OSMutexQuery()` from an ISR.

Notes/Warnings

1) Mutexes must be created before they are used.
2) You cannot call this function from an ISR.

Example

In this example, we check the contents of the mutex to determine the highest priority task that is waiting for it.

```
OS_EVENT *DispMutex;


void Task (void *pdata)
{
    OS_MUTEX_DATA mutex_data;
    INT8U         err;
    INT8U         highest;       /* Highest priority task waiting on mutex */
    INT8U         x;
    INT8U         y;


    pdata = pdata;
    for (;;) {
        .
        .
        err = OSMutexQuery(DispMutex, &mutex_data);
        if (err == OS_NO_ERR) {
            if (mutex_data.OSEventGrp != 0x00) {
                y       = OSUnMapTbl[mutex_data.OSEventGrp];
                x       = OSUnMapTbl[mutex_data.OSEventTbl[y]];
                highest = (y << 3) + x;
                .
                .
            }
        }
        .
        .
    }
}
```

References

µC/OS-II, The Real-Time Kernel
Jean J. Labrosse
R&D Technical Books, 1998
ISBN 0-87930-543-6

Contacts

Micriμm, Inc.
949 Crestview Circle
Weston, FL 33327
954-217-2036
954-217-2037 (FAX)
e-mail: Jean.Labrosse@Micrium.com
WEB: www.Micrium.com

R&D Books, Inc.
1601 W. 23rd St., Suite 200
Lawrence, KS 66046-9950
(785) 841-1631
(785) 841-2624 (FAX)
WEB: http://www.rdbooks.com
e-mail: rdorders@rdbooks.com