Micriµm, Inc.

New Services in µC/OS-II V2.04

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

New #define Constants

`OS_ARG_CHK_EN` `(OS_CFG.H)`

This constant is used to specify whether argument checking will be performed at the beginning of most of µC/OS-II services. You should always choose to turn this feature on (when set to `1`) unless you need to get the best performance possible out of µC/OS-II or, you need to reduce code size.

`OS_MBOX_DEL_EN` `(OS_CFG.H)`

This constant is used to specify whether you will enable (when set to `1`) code generation for the function `OSMboxDel()` which is used to delete a mailbox.

`OS_MUTEX_DEL_EN` `(OS_CFG.H)`

This constant is used to specify whether you will enable (when set to `1`) code generation for the function `OSMutexDel()` which is used to delete a mutual exclusion semaphore.

`OS_Q_DEL_EN` `(OS_CFG.H)`

This constant is used to specify whether you will enable (when set to `1`) code generation for the function `OSQDel()` which is used to delete a message queue.

`OS_SEM_DEL_EN` `(OS_CFG.H)`

This constant is used to specify whether you will enable (when set to `1`) code generation for the function `OSSemDel()` which is used to delete a semaphore.

`OS_CRITICAL_METHOD #3` `(OS_CPU.H)`

This constant specifies the method used to disable and enable interrupts during critical sections of code. Prior to V2.04, `OS_CRITICAL_METHOD` could be set to either `1` or `2`. In V2.04, I added a local variable (i.e. `cpu_sr`) in most function calls to save the processor status register which generally holds the state of the interrupt disable flag(s). You would then declare the two critical section macros as follows:

```
#define OS_ENTER_CRITICAL()  (cpu_sr = OSCPUSaveSR())
#define OS_EXIT_CRITICAL()   (OSCPURestoreSR(cpu_sr))
```

Note that the functions `OSCPUSaveSR()` and `OSCPURestoreSR()` would be written in assembly language and would typically be found in `OS_CPU_A.ASM` (or equivalent).

New Data Types

`OS_CPU_SR` (`OS_CPU.H`)

This data type is used to specify the size of the CPU status register which is used in conjunction with `OS_CRITICAL_METHOD` #3 (see above). For example, if the CPU status register is 16-bit wide then you would `typedef` accordingly.

New Functions

`void OSInitHookBegin(void)`

This function is called at the very beginning of `OSInit()` to allow for port specific initialization BEFORE µC/OS-II gets initialized.

`void OSInitHookEnd(void)`

This function is called at the end of `OSInit()` to allow for port specific initialization AFTER µC/OS-II gets initialized.

`void OSTCBInitHook(OS_TCB *ptcb)`

This function is called by `OSTCBInit()` during initialization of the TCB assigned to a newly created task. It allows port specific initialization of the TCB.

OSMboxDel()

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

File	Called from	Code enabled by
OS_MBOX.C	Task	OS_MBOX_EN and OS_MBOX_DEL_EN

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

Arguments

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

`opt` specifies whether you want to delete the mailbox only if there are no pending tasks (`OS_DEL_NO_PEND`) or whether you always want to delete the mailbox 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 mailbox was deleted.
`OS_ERR_DEL_ISR`	if you attempted to delete the mailbox from an ISR
`OS_ERR_EVENT_TYPE`	if `pevent` is not pointing to a mailbox.
`OS_ERR_INVALID_OPT`	if you didn't specify one of the two options mentioned above.
`OS_ERR_PEVENT_NULL`	if there are no more `OS_EVENT` structures available.

Returned Value

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

Notes/Warnings

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

Interrupts are disabled when pended tasks are readied. This means that interrupt latency depends on the number of tasks that were waiting on the mailbox.

Example

```
OS_EVENT *DispMbox;


void Task (void *pdata)
{
    INT8U  err;


    pdata = pdata;
    while (1) {
        .
        .
        DispMbox = OSMboxDel(DispMbox, OS_DEL_ALWAYS, &err);
        .
        .
    }
}
```

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;
                .
                .
            }
        }
        .
        .
    }
}
```

OSQDel()

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

File	Called from	Code enabled by
OS_Q.C	Task	OS_Q_EN and OS_Q_DEL_EN

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

Arguments

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

`opt` specifies whether you want to delete the queue only if there are no pending tasks (`OS_DEL_NO_PEND`) or whether you always want to delete the queue 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 queue was deleted.
`OS_ERR_DEL_ISR`	if you attempted to delete the queue from an ISR
`OS_ERR_EVENT_TYPE`	if `pevent` is not pointing to a queue.
`OS_ERR_INVALID_OPT`	if you didn't specify one of the two options mentioned above.
`OS_ERR_PEVENT_NULL`	if there are no more `OS_EVENT` structures available.

Returned Value

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

Notes/Warnings

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

Interrupts are disabled when pended tasks are readied. This means that interrupt latency depends on the number of tasks that were waiting on the queue.

Example

```
OS_EVENT *DispQ;


void Task (void *pdata)
{
    INT8U  err;


    pdata = pdata;
    while (1) {
        .
        .
        DispQ = OSQDel(DispQ, OS_DEL_ALWAYS, &err);
        .
        .
    }
}
```

OSSemDel()

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

File	Called from	Code enabled by
OS_SEM.C	Task	OS_SEM_EN and OS_SEM_DEL_EN

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

Arguments

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

`opt` specifies whether you want to delete the semaphore only if there are no pending tasks (`OS_DEL_NO_PEND`) or whether you always want to delete the semaphore 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 semaphore was deleted.
`OS_ERR_DEL_ISR`	if you attempted to delete the semaphore from an ISR
`OS_ERR_EVENT_TYPE`	if `pevent` is not pointing to a semaphore.
`OS_ERR_INVALID_OPT`	if you didn't specify one of the two options mentioned above.
`OS_ERR_PEVENT_NULL`	if there are no more `OS_EVENT` structures available.

Returned Value

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

Notes/Warnings

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

Interrupts are disabled when pended tasks are readied. This means that interrupt latency depends on the number of tasks that were waiting on the semaphore.

Example

```
OS_EVENT *DispSem;


void Task (void *pdata)
{
    INT8U  err;


    pdata = pdata;
    while (1) {
        .
        .
        DispSem = OSSemDel(DispSem, OS_DEL_ALWAYS, &err);
        .
        .
    }
}
```

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