# Micriμm, Inc.



# New Features and Services since μC/OS-II V2.00

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

# Introduction

This document describes all the features and services added to µC/OS-II since the introduction of the hard cover book *MicroC/OS-II, The Real-Time Kernel*, ISBN 0-87930-543-6. The software provided with the book was version 2.00 or V2.04. The version number of the change is shown when appropriate.

# New #define Constants and Macros

`OS_ARG_CHK_EN` `(OS_CFG.H, V2.04)`

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_CRITICAL_METHOD #3` `(OS_CPU.H, V2.04)`

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).

`OS_DEBUG_EN` `(OS_CFG.H, V2.60)`

This constant is used to enable ROM constants used for debugging using a kernel aware debugger. The constants are found in `OS_CORE.C`.

`OS_EVENT_NAME_SIZE` `(OS_CFG.H, V2.60)`

This constant determines the maximum number of characters that would be used to assign a name to either a semaphore, a mutex, a mailbox or a message queue. The name of these 'objects' would thus have to be smaller (in size) than this value. If `OS_EVENT_NAME_SIZE` is set to `0`, this feature is disabled. `OS_EVENT_NAME_SIZE` needs to accommodate a `NUL` terminated ASCII string.

`OS_FLAG_EN` `(OS_CFG.H, V2.51)`

This constant is used to specify whether you will enable (when 1) code generation for the event flags.

`OS_FLAG_NAME_SIZE` `(OS_CFG.H, V2.60)`

This constant determines the maximum number of characters that would be used to assign a name to an event flag group. The name of event flags would thus have to be smaller (in size) than this value. If `OS_FLAG_NAME_SIZE` is set to `0`, this feature is disabled. `OS_FLAG_NAME_SIZE` needs to accommodate a `NUL` terminated ASCII string.

OS_FLAG_WAIT_CLR_EN (OS_CFG.H, V2.51)

This constant is used to enable code generation (when 1) to allow to wait on cleared event flags.

OS_ISR_PROTO_EXT (OS_CPU.H, V2.02)

If you place this constant is OS_CPU.H, you can redefine the function prototypes for OSCtxSw() and OSTickISR(). In other words, if you add the following definition, YOU will have to declare the prototype for OSCtxSw() and OSTickISR().

```
#define OS_ISR_PROTO_EXT  1
```

OS_MAX_FLAGS (OS_CFG.H, V2.51)

This constant is used to determine how many event flags your application will support.

OS_MEM_NAME_SIZE (OS_CFG.H, V2.60)

This constant determines the maximum number of characters that would be used to assign a name to a memory partition. The name of memory partitions would thus have to be smaller (in size) than this value. If OS_MEM_NAME_SIZE is set to 0, this feature is disabled and no RAM is used in the OS_MEM for the memory partition. OS_MEM_NAME_SIZE needs to accommodate a NUL terminated ASCII string.

OS_MUTEX_EN (OS_CFG.H, V2.04)

This constant is used to specify whether you will enable (when 1) code generation for mutual exclusion semaphores.

OS_TASK_NAME_SIZE (OS_CFG.H, V2.60)

This constant determines the maximum number of characters that would be used to assign a name to a task. The name of tasks would thus have to be smaller (in size) than this value. If OS_TASK_NAME_SIZE is set to 0, this feature is disabled and no RAM is used in the OS_TCB for the task name. OS_TASK_NAME_SIZE needs to accommodate a NUL terminated ASCII string.

OS_TASK_PROFILE_EN (OS_CFG.H, V2.60)

This constant allows variables to be allocated in each task's OS_TCB that hold performance data about each task. Specifically, if OS_TASK_PROFILE_EN is set to 1, each task will have a variable to keep track of the number of context switches, the task execution time, the number of bytes used by the task and more.

OS_TASK_STAT_STK_CHK_EN (OS_CFG.H, V2.60)

This constant allows the statistic task to determine the actual stack usage of each active task. If OS_TASK_STAT_EN is set to 0 (the statistic task is not enabled), you can call OS_TaskStatStkChk() yourself from one of your tasks. . If OS_TASK_STAT_EN is set to 1, stack sizes will be determined every second.

OS_TASK_SW_HOOK_EN (OS_CFG.H, V2.60)

Normally, µC/OS-II requires that you have a context switch hook function called OSTaskSwHook(). When set to 0, this constant allows you to omit OSTaskSwHook() from your code. This configuration constant was added to reduce the amount of overhead during a context switch in applications that doesn't require the context switch hook. Of course, you will also need to remove the calls to OSTaskSwHook() from OSTaskStartHighRdy(), OSCtxSw() and OSIntCtxSw() in OS_CPU_A.ASM.

OS_TICK_STEP_EN (OS_CFG.H, V2.60)

µC/OS-View can now 'halt' µC/OS-II's tick processing and allow you to issue 'step' commands from µC/OS-View. In other words, µC/OS-View can prevent µC/OS-II from calling OSTimeTick() so that timeouts and time delays are no longer processed. However, though a keystroke from µC/OS-View, you can execute a single tick at a time. If OS_TIME_TICK_HOOK_EN (see below) is set to 1, OSTimeTickHook() is still executed at the regular tick rate in case you have time critical items to take care of in your application.

OS_TIME_TICK_HOOK_EN (OS_CFG.H, V2.60)

Normally, µC/OS-II requires the presence of a function called OSTimeTickHook() which is called at the very beginning of the tick ISR. When set to 0, this constant allows you to omit OSTimeTickHook() from your code. This configuration constant was added to reduce the amount of overhead during a tick ISR in applications that doesn't require this hook.

The following table summarizes some of the new #define constants in OS_CFG.H which were all added in since V2.00.

| **#define name in OS_CFG.H** | **... to enable the function:** |
|---|---|
| OS_DEBUG_EN | Enable debug constants in OS_CORE.C. If you are using a kernel aware debugger, you should enable this feature. |
| | |
| OS_EVENT_NAME_SIZE | OSEventNameGet()<br>OSEventNameSet()<br>And, to allow naming semaphores, mutexes, mailboxes and message queues. |
| | |
| OS_FLAG_ACCEPT_EN | OSFlagAccept() |
| OS_FLAG_DEL_EN | OSFlagDel() |
| OS_FLAG_NAME_SIZE | OSFlagNameGet()<br>OSFlagNameSet()<br>And, to allow naming event flag groups. |
| OS_FLAG_QUERY_EN | OSFlagQuery() |
| | |
| OS_MBOX_ACCEPT_EN | OSMboxAccept() |
| OS_MBOX_DEL_EN | OSMboxDel() |
| OS_MBOX_POST_EN | OSMboxPost() |
| OS_MBOX_POST_OPT_EN | OSMboxPostOpt() |
| OS_MBOX_QUERY_EN | OSMBoxQuery() |
| | |
| OS_MEM_NAME_SIZE | OSMemNameGet()<br>OSMemNameSet() |
| OS_MEM_QUERY_EN | OSMemQuery() |
| | |
| OS_MUTEX_ACCEPT_EN | OSMutexAccept() |
| OS_MUTEX_DEL_EN | OSMutexDel() |
| OS_MUTEX_QUERY_EN | OSMutexQuery() |

| | |
|---|---|
| OS_Q_ACCEPT_EN | OSQAccept() |
| OS_Q_DEL_EN | OSQDel() |
| OS_Q_FLUSH_EN | OSQFlush() |
| OS_Q_POST_EN | OSQPost() |
| OS_Q_POST_FRONT_EN | OSQPostFront() |
| OS_Q_POST_OPT_EN | OSQPostOpt() |
| OS_Q_QUERY_EN | OSQQuery() |
| | |
| OS_SEM_ACCEPT_EN | OSSemAccept() |
| OS_SEM_DEL_EN | OSSemDel() |
| OS_SEM_QUERY_EN | OSSemQuery() |
| | |
| OS_TASK_NAME_SIZE | OSTaskNameGet()<br>OSTaskNameSet()<br>And, to allow naming tasks. |
| OS_TASK_PROFILE_EN | To allocate variables in OS_TCB for performance monitoring of each task at run-time. |
| OS_TASK_QUERY_EN | OSTaskQuery() |
| OS_TASK_STAT_STK_CHK_EN | OS_TaskStatStkChk() |
| OS_TASK_SW_HOOK_EN | OSTaskSwHook() |
| | |
| OS_TICK_STEP_EN | To support the stepping feature of µC/OS-View. |
| | |
| OS_TIME_DLY_HMSM_EN | OSTimeDlyHMSM() |
| OS_TIME_DLY_RESUME_EN | OSTimeDlyResume() |
| OS_TIME_GET_SET_EN | OSTimeGet() and OSTimeSet() |
| OS_TIME_TICK_HOOK_EN | OSTimeTickHook() |
| | |
| OS_SCHED_LOCK_EN | OSSchedLock() and OSSchedUnlock() |

# New Data Types

`OS_CPU_SR` `(OS_CPU.H, V2.04)`

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.

`OS_FLAGS` `(OS_CFG.H, V2.51)`

This data type determines how many bits an event flag group will have. You can thus `typedef` this data type to either `INT8U`, `INT16U` or `INT32U` to give event flags either 8, 16 or 32 bits, respectively.

# New Hook Functions

`void OSInitHookBegin(void)` `(OS_CPU.C, V2.04)`

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)` `(OS_CPU.C, V2.04)`

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)` `(OS_CPU.C, V2.04)`

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.

`void OSTaskIdleHook(void)` `(OS_CPU.C, V2.51)`

This function is called by `OSTaskIdle()`. This allows you to STOP the CPU and thus reduce power consumption while there is nothing to do.

# New Functions

This section describes the new functions (i.e. services) that YOUR application can call that were added or changed since V2.00.

# OSEventNameGet()

```
INT8U OSEventNameGet(OS_EVENT *pevent, char *pname, INT8U *err);
```

| Chapter | File | Called from | Code enabled by |
|---|---|---|---|
| New in V2.60 | OS_CORE.C | Task or ISR | OS_EVENT_NAME_SIZE |

OSEventNameGet() allows you to obtain the name that you assigned to a semaphore, a mutex, a mailbox or a message queue. The name is an ASCII string and the size of the name can contain up to OS_EVENT_NAME_SIZE characters (including the NUL termination). This function is typically used by a debugger to allow associating a name to a resource.

## Arguments

pevent is a pointer to the event control block. pevent can point either to a semaphore, a mutex, a mailbox or a queue. Where this function is concerned, the actual type is irrelevant. This pointer is returned to your application when the semaphore, mutex, mailbox or queue is created (see OSSemCreate(), OSMutexCreate(), OSMboxCreate() and OSQCreate()).

pname is a pointer to an ASCII string that will receive the name of the semaphore, mutex, mailbox or queue. The string must be able to hold at least OS_EVENT_NAME_SIZE characters (including the NUL character).

err a pointer to an error code and can be any of the following:

| | |
|---|---|
| OS_NO_ERR | If the name of the semaphore, mutex, mailbox or queue was copied to the array pointed to by pname. |
| OS_ERR_EVENT_TYPE | You are not pointing to either a semaphore, mutex, mailbox or message queue. |
| OS_ERR_PEVENT_NULL | You passed a NULL pointer for pevent. |

## Returned Values

The size of the ASCII string placed in the array pointed to by pname or 0 if an error is encountered.

## Notes/Warnings

1. The semaphore, mutex, mailbox or message queue must be created before you can use this function and obtain the name of the resource.

## Example

```
char      PrinterSemName[30];
OS_EVENT *PrinterSem;



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



    pdata = pdata;
    for (;;) {
        size = OSEventNameGet(PrinterSem, &PrinterSemName[0], &err);
        .
        .
    }
}
```

# OSEventNameSet()

```
void OSEventNameSet(OS_EVENT *pevent, char *pname, INT8U *err);
```

| Chapter | File | Called from | Code enabled by |
|---|---|---|---|
| New in V2.60 | OS_CORE.C | Task or ISR | OS_EVENT_NAME_SIZE |

`OSEventNameSet()` allows you to assign a name to a semaphore, a mutex, a mailbox or a message queue. The name is an ASCII string and the size of the name can contain up to `OS_EVENT_NAME_SIZE` characters (including the `NUL` termination). This function is typically used by a debugger to allow associating a name to a resource.

## Arguments

`pevent` is a pointer to the event control block that you want to name. `pevent` can point either to a semaphore, a mutex, a mailbox or a queue. Where this function is concerned, the actual type is irrelevant. This pointer is returned to your application when the semaphore, mutex, mailbox or queue is created (see `OSSemCreate()`, `OSMutexCreate()`, `OSMboxCreate()` and `OSQCreate()`).

`pname` is a pointer to an ASCII string that contains the name for the resource. The size of the string must be smaller than or equal to `OS_EVENT_NAME_SIZE` characters (including the `NUL` character).

`err` a pointer to an error code and can be any of the following:

| | |
|---|---|
| `OS_NO_ERR` | If the name of the semaphore, mutex, mailbox or queue was copied to the array pointed to by `pname`. |
| `OS_ERR_EVENT_TYPE` | You are not pointing to either a semaphore, mutex, mailbox or message queue. |
| `OS_ERR_PEVENT_NULL` | You passed a `NULL` pointer for `pevent`. |

## Returned Values

none

## Notes/Warnings

1. The semaphore, mutex, mailbox or message queue must be created before you can use this function and set the name of the resource.

## Example

```
OS_EVENT *PrinterSem;



void Task (void *pdata)
{
    INT8U  err;



    pdata = pdata;
    for (;;) {
        OSEventNameSet(PrinterSem, “Printer #1”, &err);
        .
        .
    }
}
```

# OSFlagAccept()

```
OS_FLAGS OSFlagAccept(OS_FLAG_GRP *pgrp, OS_FLAGS flags, INT8U wait_type,
INT8U *err);
```

| Chapter | File | Called from | Code enabled by |
|---|---|---|---|
| 9 | OS_FLAG.C | Task | OS_FLAG_EN && OS_FLAG_ACCEPT_EN |

`OSFlagAccept()` allows you to check the status of a combination of bits to be either set or cleared in an event flag group. Your application can check for **any** bit to be set/cleared or **all** bits to be set/cleared. This function behaves exactly as `OSFlagPend()` does, except that the caller does NOT block if the desired event flags are not present.

## Arguments

`pgrp` is a pointer to the event flag group. This pointer is returned to your application when the event flag group is created [see `OSFlagCreate()`].

`flags` is a bit pattern indicating which bit(s) (i.e., flags) you wish to check. The bits you want are specified by setting the corresponding bits in `flags`.

`wait_type` specifies whether you want **all** bits to be set/cleared or **any** of the bits to be set/cleared. You can specify the following arguments:

| | |
|---|---|
| `OS_FLAG_WAIT_CLR_ALL` | You check **all** bits in `flags` to be clear (0) |
| `OS_FLAG_WAIT_CLR_ANY` | You check **any** bit in `flags` to be clear (0) |
| `OS_FLAG_WAIT_SET_ALL` | You check **all** bits in `flags` to be set (1) |
| `OS_FLAG_WAIT_SET_ANY` | You check **any** bit in `flags` to be set (1) |

You can add `OS_FLAG_CONSUME` if you want the event flag(s) to be consumed by the call. For example, to wait for **any** flag in a group and then clear the flags that are present, set `wait_type` to

```
OS_FLAG_WAIT_SET_ANY + OS_FLAG_CONSUME
```

`err` a pointer to an error code and can be any of the following:

| | |
|---|---|
| `OS_NO_ERR` | No error |
| `OS_ERR_EVENT_TYPE` | You are not pointing to an event flag group |
| `OS_FLAG_ERR_WAIT_TYPE` | You didn't specify a proper `wait_type` argument. |
| `OS_FLAG_INVALID_PGRP` | You passed a `NULL` pointer instead of the event flag handle. |
| `OS_FLAG_ERR_NOT_RDY` | The desired flags for which you are waiting are not available. |

## Returned Values

The state of the flags in the event flag group.

## Notes/Warnings

1. The event flag group must be created before it is used.
2. This function does **not** block if the desired flags are not present.

## Example

```
#define  ENGINE_OIL_PRES_OK   0x01
#define  ENGINE_OIL_TEMP_OK   0x02
#define  ENGINE_START         0x04



OS_FLAG_GRP *EngineStatus;



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



    pdata = pdata;
    for (;;) {
        value = OSFlagAccept(EngineStatus,
                             ENGINE_OIL_PRES_OK + ENGINE_OIL_TEMP_OK,
                             OS_FLAG_WAIT_SET_ALL,
                             &err);
        switch (err) {
            case OS_NO_ERR:
                 /* Desired flags are available */
                 break;

            case OS_FLAG_ERR_NOT_RDY:
                 /* The desired flags are NOT available */
                 break;
        }
        .
        .
    }
}
```

# OSFlagCreate()

```
OS_FLAG_GRP *OSFlagCreate(OS_FLAGS flags, INT8U *err);
```

| *Chapter* | *File* | *Called from* | *Code enabled by* |
|---|---|---|---|
| 9 | OS_FLAG.C | Task or startup code | OS_FLAG_EN |

`OSFlagCreate()` is used to create and initialize an event flag group.

## Arguments

`flags` contains the initial value to store in the event flag group.

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

| | |
|---|---|
| `OS_NO_ERR` | if the call is successful and the event flag group has been created. |
| `OS_ERR_CREATE_ISR` | if you attempt to create an event flag group from an ISR. |
| `OS_FLAG_GRP_DEPLETED` | if no more event flag groups are available. You need to increase the value of `OS_MAX_FLAGS` in `OS_CFG.H`. |

## Returned Values

A pointer to the event flag group if a free event flag group is available. If no event flag group is available, `OSFlagCreate()` returns a `NULL` pointer.

## Notes/Warnings

1. Event flag groups must be created by this function before they can be used by the other services.

## Example

```
OS_FLAG_GRP *EngineStatus;


void main (void)
{
    INT8U  err;


    .
    OSInit();         /* Initialize µC/OS-II */
    .
    .
                      /* Create a flag group containing the engine's status
*/
    EngineStatus = OSFlagCreate(0x00, &err);
    .
    .
    OSStart();        /* Start Multitasking  */
}
```

# OSFlagDel()

```
OS_FLAG_GRP *OSFlagDel(OS_FLAG_GRP *pgrp, INT8U opt, INT8U *err);
```

| *Chapter* | *File* | *Called from* | *Code enabled by* |
|---|---|---|---|
| 9 | OS_FLAG.C | Task | OS_FLAG_EN and OS_FLAG_DEL_EN |

OSFlagDel() is used to delete an event flag group. This function is dangerous to use because multiple tasks could be relying on the presence of the event flag group. You should always use this function with great care. Generally speaking, before you delete an event flag group, you must first delete all the tasks that access the event flag group.

## Arguments

pgrp is a pointer to the event flag group. This pointer is returned to your application when the event flag group is created [see OSFlagCreate()].

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

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

| | |
|---|---|
| OS_NO_ERR | if the call is successful and the event flag group has been deleted. |
| OS_ERR_DEL_ISR | if you attempt to delete an event flag group from an ISR. |
| OS_FLAG_INVALID_PGRP | if you pass a NULL pointer in pgrp. |
| OS_ERR_EVENT_TYPE | if pgrp is not pointing to an event flag group. |
| OS_ERR_INVALID_OPT | if you do not specify one of the two options mentioned in the opt argument. |
| OS_ERR_TASK_WAITING | if one or more task are waiting on the event flag group and you specify OS_DEL_NO_PEND. |

## Returned Values

A NULL pointer if the event flag group is deleted or pgrp if the event flag group is not deleted. In the latter case, you need to examine the error code to determine the reason for the error.

## Notes/Warnings

1. You should use this call with care because other tasks might expect the presence of the event flag group.
2. This call can potentially disable interrupts for a long time. The interrupt-disable time is directly proportional to the number of tasks waiting on the event flag group.

## Example

```
OS_FLAG_GRP *EngineStatusFlags;



void Task (void *pdata)
{
    INT8U        err;
    OS_FLAG_GRP *pgrp;



    pdata = pdata;
    while (1) {
        .
        .
        pgrp = OSFlagDel(EngineStatusFlags, OS_DEL_ALWAYS, &err);
        if (pgrp == (OS_FLAG_GRP *)0) {
            /* The event flag group was deleted */
        }
        .
        .
    }
}
```

# OSFlagNameGet()

```
INT8U OSFlagNameGet(OS_FLAG_GRP *pgrp, char *pname, INT8U *err);
```

| Chapter | File | Called from | Code enabled by |
|---|---|---|---|
| New in V2.60 | OS_FLAG.C | Task or ISR | OS_FLAG_NAME_SIZE |

`OSFlagNameGet()` allows you to obtain the name that you assigned to an event flag group. The name is an ASCII string and the size of the name can contain up to `OS_FLAG_NAME_SIZE` characters (including the `NUL` termination). This function is typically used by a debugger to allow associating a name to a resource.

## Arguments

`pgrp` is a pointer to the event flag group.

`pname` is a pointer to an ASCII string that will receive the name of the event flag group. The string must be able to hold at least `OS_FLAG_NAME_SIZE` characters (including the `NUL` character).

`err` a pointer to an error code and can be any of the following:

| | |
|---|---|
| `OS_NO_ERR` | If the name of the semaphore, mutex, mailbox or queue was copied to the array pointed to by `pname`. |
| `OS_ERR_EVENT_TYPE` | You are not pointing to either a semaphore, mutex, mailbox or message queue. |
| `OS_ERR_INVALID_PGRP` | You passed a `NULL` pointer for `pgrp`. |

## Returned Values

The size of the ASCII string placed in the array pointed to by `pname` or `0` if an error is encountered.

## Notes/Warnings

1. The event flag group must be created before you can use this function and obtain the name of the resource.

## Example

```
char         EngineStatusName[30];
OS_FLAG_GRP *EngineStatusFlags;



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



    pdata = pdata;
    for (;;) {
        size = OSFlagNameGet(EngineStatusFlags, &EngineStatusName[0],
&err);
        .
        .
    }
}
```

# OSFlagNameSet()

```
void OSFlagNameSet(OS_FLAG_GRP *pgrp, char *pname, INT8U *err);
```

| Chapter | File | Called from | Code enabled by |
|---|---|---|---|
| New in V2.60 | OS_FLAG.C | Task or ISR | OS_EVENT_NAME_SIZE |

OSFlagNameSet() allows you to assign a name to an event flag group. The name is an ASCII string and the size of the name can contain up to OS_FLAG_NAME_SIZE characters (including the NUL termination). This function is typically used by a debugger to allow associating a name to a resource.

## Arguments

pgrp is a pointer to the event flag group that you want to name. This pointer is returned to your application when the event flag group is created (see OSFlagCreate()).

pname is a pointer to an ASCII string that contains the name for the resource. The size of the string must be smaller than or equal to OS_EVENT_NAME_SIZE characters (including the NUL character).

err a pointer to an error code and can be any of the following:

| | |
|---|---|
| OS_NO_ERR | If the name of the event flag group was copied to the array pointed to by pname. |
| OS_ERR_EVENT_TYPE | You are not pointing to an event flag group. |
| OS_ERR_INVALID_PGRP | You passed a NULL pointer for pgrp. |

## Returned Values

none

## Notes/Warnings

1. The event flag group must be created before you can use this function to set the name of the resource.

## Example

```
OS_FLAG_GRP *EngineStatus;



void Task (void *pdata)
{
    INT8U  err;



    pdata = pdata;
    for (;;) {
        OSFlagNameSet(EngineStatus, “Engine Status Flags”, &err);
        .
        .
    }
}
```

# OSFlagPend()

```
OS_FLAGS OSFlagPend(OS_FLAG_GRP *pgrp,
                    OS_FLAGS     flags,
                    INT8U        wait_type,
                    INT16U       timeout,
                    INT8U       *err);
```

| *Chapter* | *File* | *Called from* | *Code enabled by* |
|---|---|---|---|
| 9 | OS_FLAG.C | Task only | OS_FLAG_EN |

`OSFlagPend()` is used to have a task wait for a combination of conditions (i.e., events or bits) to be set (or cleared) in an event flag group. You application can wait for **any** condition to be set or cleared or for **all** conditions to be set or cleared. If the events that the calling task desires are not available, then the calling task is blocked until the desired conditions are satisfied or the specified timeout expires.

## Arguments

`pgrp` is a pointer to the event flag group. This pointer is returned to your application when the event flag group is created [see `OSFlagCreate()`].

`flags` is a bit pattern indicating which bit(s) (i.e., flags) you wish to check. The bits you want are specified by setting the corresponding bits in `flags`.

`wait_type` specifies whether you want **all** bits to be set/cleared or **any** of the bits to be set/cleared. You can specify the following arguments:

| | |
|---|---|
| `OS_FLAG_WAIT_CLR_ALL` | You check **all** bits in `flags` to be clear (0) |
| `OS_FLAG_WAIT_CLR_ANY` | You check **any** bit in `flags` to be clear (0) |
| `OS_FLAG_WAIT_SET_ALL` | You check **all** bits in `flags` to be set (1) |
| `OS_FLAG_WAIT_SET_ANY` | You check **any** bit in `flags` to be set (1) |

You can also specify whether the flags are consumed by adding `OS_FLAG_CONSUME` to the `wait_type`. For example, to wait for **any** flag in a group and then **clear** the flags that satisfy the condition, set `wait_type` to

```
OS_FLAG_WAIT_SET_ANY + OS_FLAG_CONSUME
```

`err` is a pointer to an error code and can be:

| | |
|---|---|
| `OS_NO_ERR` | No error. |
| `OS_ERR_PEND_ISR` | You try to call `OSFlagPend` from an ISR, which is not allowed. |
| `OS_FLAG_INVALID_PGRP` | You pass a `NULL` pointer instead of the event flag handle. |
| `OS_ERR_EVENT_TYPE` | You are not pointing to an event flag group. |
| `OS_TIMEOUT` | The flags are not available within the specified amount of time. |
| `OS_FLAG_ERR_WAIT_TYPE` | You don't specify a proper `wait_type` argument. |

## Returned Value

The value of the flags in the event flag group after they are consumed (if `OS_FLAG_CONSUME` is specified) or the state of the flags just before `OSFlagPend()` returns. `OSFlagPend()` returns 0 if a timeout occurs.

## Notes/Warnings

1. The event flag group must be created before it's used.

## Example

```
#define  ENGINE_OIL_PRES_OK   0x01
#define  ENGINE_OIL_TEMP_OK   0x02
#define  ENGINE_START         0x04



OS_FLAG_GRP *EngineStatus;



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



    pdata = pdata;
    for (;;) {
        value = OSFlagPend(EngineStatus,
                           ENGINE_OIL_PRES_OK   + ENGINE_OIL_TEMP_OK,
                           OS_FLAG_WAIT_SET_ALL + OS_FLAG_CONSUME,
                           10,
                           &err);
        switch (err) {
            case OS_NO_ERR:
              /* Desired flags are available */
                 break;

            case OS_TIMEOUT:
              /* The desired flags were NOT available before 10 ticks
occurred */
                 break;
        }
        .
        .
    }
}
```

# OSFlagPendGetRdyFlags()

```
OS_FLAGS OSFlagPendGetRdyFlags(void)
```

| *Chapter* | *File* | *Called from* | *Code enabled by* |
|---|---|---|---|
| Added in V2.60 | OS_FLAG.C | Task only | OS_FLAG_EN |

`OSFlagPendGetRdyFlags()` is used to obtain the flags that caused the current task to become ready to run. In other words, this function allows you to know "Who done It!"

## Arguments

None

## Returned Value

The value of the flags that caused the current task to become ready to run.

## Notes/Warnings

1. The event flag group must be created before it's used.

## Example

```
#define  ENGINE_OIL_PRES_OK   0x01
#define  ENGINE_OIL_TEMP_OK   0x02
#define  ENGINE_START         0x04

OS_FLAG_GRP *EngineStatus;

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



    pdata = pdata;
    for (;;) {
        value = OSFlagPend(EngineStatus,
                           ENGINE_OIL_PRES_OK   + ENGINE_OIL_TEMP_OK,
                           OS_FLAG_WAIT_SET_ALL + OS_FLAG_CONSUME,
                           10,
                           &err);
        switch (err) {
            case OS_NO_ERR:
                 flags = OSFlagPendGetRdyFlags();  /* Find out who made
task ready */
                 break;


            case OS_TIMEOUT:
                 /* The desired flags were NOT available before 10 ticks
occurred */
                 break;
        }
        .
        .
    }
}
```

# OSFlagPost()

```
OS_FLAGS OSFlagPost(OS_FLAG_GRP *pgrp, OS_FLAGS flags, INT8U opt, INT8U *err);
```

| *Chapter* | *File* | *Called from* | *Code enabled by* |
|---|---|---|---|
| 9 | OS_FLAG.C | Task or ISR | OS_FLAG_EN |

You set or clear event flag bits by calling `OSFlagPost()`. The bits set or cleared are specified in a *bit mask*. `OSFlagPost()` readies each task that has its desired bits satisfied by this call. You can set or clear bits that are already set or cleared.

## Arguments

`pgrp` is a pointer to the event flag group. This pointer is returned to your application when the event flag group is created [see `OSFlagCreate()`].

`flags` specifies which bits you want set or cleared. If `opt` is `OS_FLAG_SET`, each bit that is set in `flags` sets the corresponding bit in the event flag group. For example to set bits 0, 4, and 5, you set `flags` to `0x31` (note, bit 0 is the least significant bit). If `opt` is `OS_FLAG_CLR`, each bit that is set in flags will **clears** the corresponding bit in the event flag group. For example to clear bits 0, 4, and 5, you specify `flags` as `0x31` (note, bit 0 is the least significant bit).

`opt` indicates whether the flags are set (`OS_FLAG_SET`) or cleared (`OS_FLAG_CLR`).

`err` is a pointer to an error code and can be:

| | |
|---|---|
| `OS_NO_ERR` | The call is successful. |
| `OS_FLAG_INVALID_PGRP` | You pass a `NULL` pointer. |
| `OS_ERR_EVENT_TYPE` | You are not pointing to an event flag group. |
| `OS_FLAG_INVALID_OPT` | You specify an invalid option. |

## Returned Value

The new value of the event flags.

## Notes/Warnings

1. Event flag groups must be created before they are used.
2. The execution time of this function depends on the number of tasks waiting on the event flag group. However, the execution time is deterministic.
3. The amount of time interrupts are **disabled** also depends on the number of tasks waiting on the event flag group.

## Example

```
#define  ENGINE_OIL_PRES_OK   0x01
#define  ENGINE_OIL_TEMP_OK   0x02
#define  ENGINE_START         0x04



OS_FLAG_GRP  *EngineStatusFlags;



void  TaskX (void *pdata)
{
    INT8U  err;



    pdata = pdata;
    for (;;) {
        .
        .
        err = OSFlagPost(EngineStatusFlags, ENGINE_START, OS_FLAG_SET,
&err);
        .
        .
    }
}
```

# OSFlagQuery()

```
OS_FLAGS OSFlagQuery(OS_FLAG_GRP *pgrp, INT8U *err);
```

| *Chapter* | *File* | *Called from* | *Code enabled by* |
|---|---|---|---|
| 9 | OS_FLAG.C | Task or ISR | OS_FLAG_EN && OS_FLAG_QUERY_EN |

`OSFlagQuery()` is used to obtain the current value of the event flags in a group. At this time, this function does **not** return the list of tasks waiting for the event flag group.

## Arguments

`pgrp` is a pointer to the event flag group. This pointer is returned to your application when the event flag group is created [see `OSFlagCreate()`].

`err` is a pointer to an error code and can be:

| | |
|---|---|
| `OS_NO_ERR` | The call is successful. |
| `OS_FLAG_INVALID_PGRP` | You pass a `NULL` pointer. |
| `OS_ERR_EVENT_TYPE` | You are not pointing to an event flag groups. |

## Returned Value

The state of the flags in the event flag group.

## Notes/Warnings

1. The event flag group to query must be created.
2. You can call this function from an ISR.

## Example

```
OS_FLAG_GRP *EngineStatusFlags;


void Task (void *pdata)
{
    OS_FLAGS flags;
    INT8U    err;


    pdata = pdata;
    for (;;) {
        .
        .
        flags = OSFlagQuery(EngineStatusFlags, &err);
        .
        .
    }
}
```

# OSMemNameGet()

```
INT8U OSMemNameGet(OS_MEM *pmem, char *pname, INT8U *err);
```

| *Chapter* | *File* | *Called from* | *Code enabled by* |
|---|---|---|---|
| *New in V2.60* | *OS_MEM.C* | *Task or ISR* | *OS_MEM_NAME_SIZE* |

`OSMemNameGet()` allows you to obtain the name that you assigned to a memory partition. The name is an ASCII string and the size of the name can contain up to `OS_MEM_NAME_SIZE` characters (including the `NUL` termination). This function is typically used by a debugger to allow associating a name to a resource.

## Arguments

`pmem` is a pointer to the memory partition.

`pname` is a pointer to an ASCII string that will receive the name of the memory partition. The string must be able to hold at least `OS_MEM_NAME_SIZE` characters (including the `NUL` character).

`err` a pointer to an error code and can be any of the following:

| | |
|---|---|
| `OS_NO_ERR` | If the name of the semaphore, mutex, mailbox or queue was copied to the array pointed to by `pname`. |
| `OS_ERR_INVALID_PMEM` | You passed a `NULL` pointer for `pmem`. |

## Returned Values

The size of the ASCII string placed in the array pointed to by `pname` or `0` if an error is encountered.

## Notes/Warnings

1. The memory partition must be created before you can use this function and obtain the name of the resource.

## Example

```
OS_MEM  *CommMem;
char     CommMemName[OS_MEM_NAME_SIZE];


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



    pdata = pdata;
    for (;;) {
        size = OSMemNameGet(CommMem, & CommMemName [0], &err);
        .
        .
    }
}
```

# OSMemNameSet()

```
void OSMemNameSet(OS_MEM *pmem, char *pname, INT8U *err);
```

| *Chapter* | *File* | *Called from* | *Code enabled by* |
|---|---|---|---|
| *New in V2.60* | *OS_MEM.C* | *Task or ISR* | *OS_MEM_NAME_SIZE* |

`OSMemNameSet()` allows you to assign a name to a memory partition. The name is an ASCII string and the size of the name can contain up to `OS_MEM_NAME_SIZE` characters (including the `NUL` termination). This function is typically used by a debugger to allow associating a name to a resource.

## Arguments

`pmem` is a pointer to the memory partition that you want to name. This pointer is returned to your application when the memory partition is created (see `OSMemCreate()`).

`pname` is a pointer to an ASCII string that contains the name for the resource. The size of the string must be smaller than or equal to `OS_MEM_NAME_SIZE` characters (including the `NUL` character).

`err` a pointer to an error code and can be any of the following:

| | |
|---|---|
| `OS_NO_ERR` | If the name of the event flag group was copied to the array pointed to by `pname`. |
| `OS_ERR_INVALID_PMEM` | You passed a `NULL` pointer for `pmem`. |

## Returned Values

none

## Notes/Warnings

1. The memory partition must be created before you can use this function to set the name of the resource.

## Example

```
OS_MEM *CommMem;



void Task (void *pdata)
{
    INT8U  err;



    pdata = pdata;
    for (;;) {
        OSMemNameSet(CommMem, “Comm. Buffer”, &err);
        .
        .
    }
}
```

# OSMboxDel()

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

| *Chapter* | *File* | *Called from* | *Code enabled by* |
|---|---|---|---|
| 10 | OS_MBOX.C | Task | OS_MBOX_EN and OS_MBOX_DEL_EN |

`OSMboxDel()` is used to delete a message mailbox. This function is dangerous 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 delete a mailbox, you must first delete all the tasks that can 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 are readied.

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

| | |
|---|---|
| `OS_NO_ERR` | if the call is successful and the mailbox has been deleted. |
| `OS_ERR_DEL_ISR` | if you attempt to delete the mailbox from an ISR. |
| `OS_ERR_INVALID_OPT` | if you don't specify one of the two options mentioned in the `opt` argument. |
| `OS_ERR_TASK_WAITING` | One or more tasks is waiting on the mailbox. |
| `OS_ERR_EVENT_TYPE` | if `pevent` is not pointing to a mailbox. |
| `OS_ERR_PEVENT_NULL` | if no more `OS_EVENT` structures are available. |

## Returned Value

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

## Notes/Warnings

1. You should use this call with care because other tasks might expect the presence of the mailbox.
2. Interrupts are disabled when pended tasks are readied, which means that interrupt latency depends on the number of tasks that are waiting on the mailbox.
3. `OSMboxAccept()` callers do not know that the mailbox has been deleted.

## Example

```
OS_EVENT *DispMbox;


void Task (void *pdata)
{
    INT8U  err;


    pdata = pdata;
    while (1) {
        .
        .
        DispMbox = OSMboxDel(DispMbox, OS_DEL_ALWAYS, &err);
        if (DispMbox == (OS_EVENT *)0) {
            /* Mailbox has been deleted */
        }
        .
        .
    }
}
```

# OSMboxPostOpt()

```
INT8U OSMboxPostOpt(OS_EVENT *pevent, void *msg, INT8U opt);
```

| *Chapter* | *File* | *Called from* | *Code enabled by* |
|---|---|---|---|
| 10 | OS_MBOX.C | Task or ISR | OS_MBOX_EN and OS_MBOX_POST_OPT_EN |

OSMboxPostOpt() works just like OSMboxPost() except that it allows you to post a message to **multiple** tasks. In other words, OSMboxPostOpt() allows the message posted to be broadcast to **all** tasks waiting on the mailbox. OSMboxPostOpt() can actually replace OSMboxPost() because it can emulate OSMboxPost().

OSMboxPostOpt() is used to send a message to a task through a mailbox. A message is a pointer-sized variable, and its use is application specific. If a message is already in the mailbox, an error code is returned indicating that the mailbox is full. OSMboxPostOpt() then immediately returns to its caller, and the message is not placed in the mailbox. If any task is waiting for a message at the mailbox, OSMboxPostOpt() allows you either to post the message to the highest priority task waiting at the mailbox (opt set to OS_POST_OPT_NONE) or to all tasks waiting at the mailbox (opt is set to OS_POST_OPT_BROADCAST). In either case, scheduling occurs and, if any of the tasks that receives the message have a higher priority than the task that is posting the message, then the higher priority task is resumed, and the sending task is suspended. In other words, a context switch occurs.

## Arguments

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

msg is the actual message sent to the task(s). msg is a pointer-sized variable and is application specific. You must never post a NULL pointer because this pointer indicates that the mailbox is empty.

opt specifies whether you want to send the message to the highest priority task waiting at the mailbox (when opt is set to OS_POST_OPT_NONE) or to **all** tasks waiting at the mailbox (when opt is set to OS_POST_OPT_BROADCAST).

## Returned Value

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

| | |
|---|---|
| OS_NO_ERR | if the call is successful and the message has been sent. |
| OS_MBOX_FULL | if the mailbox already contains a message. You can only send **one** message at a time to a mailbox, and thus the message **must** be consumed before you are allowed to send another one. |
| OS_ERR_EVENT_TYPE | if pevent is not pointing to a mailbox. |
| OS_ERR_PEVENT_NULL | if pevent is a NULL pointer. |
| OS_ERR_POST_NULL_PTR | if you are attempting to post a NULL pointer. By convention, a NULL pointer is not supposed to point to anything. |

## Notes/Warnings

1. Mailboxes must be created before they are used.
2. You must **never** post a `NULL` pointer to a mailbox because this pointer indicates that the mailbox is empty.
3. If you need to use this function and want to reduce code space, you can disable code generation of `OSMboxPost()` because `OSMboxPostOpt()` can emulate `OSMboxPost()`.
4. The execution time of `OSMboxPostOpt()` depends on the number of tasks waiting on the mailbox if you set `opt` to `OS_POST_OPT_BROADCAST`.

## Example

```
OS_EVENT *CommMbox;
INT8U     CommRxBuf[100];



void CommRxTask (void *pdata)
{
    INT8U  err;



    pdata = pdata;
    for (;;) {
        .
        .
        err = OSMboxPostOpt(CommMbox, (void *)&CommRxBuf[0],
OS_POST_OPT_BROADCAST);
        .
        .
    }
}
```

# OSMutexAccept()

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

| *Chapter* | *File* | *Called from* | *Code enabled by* |
|---|---|---|---|
| 8 | 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. In other words, `OSMutexAccept()` is non-blocking.

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

## Returned Value

If the mutex is available, `OSMutexAccept()` returns 1. If the mutex is 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);
```

| *Chapter* | *File* | *Called from* | *Code enabled by* |
|---|---|---|---|
| 8 | 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 is 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 is *raised* to the PIP until the resource is released.

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

| | |
|---|---|
| `OS_NO_ERR` | if the call is successful and the mutex has been created. |
| `OS_ERR_CREATE_ISR` | if you attempt to create a mutex from an ISR. |
| `OS_PRIO_EXIST` | if a task at the specified priority inheritance priority already exists. |
| `OS_ERR_PEVENT_NULL` | if no more `OS_EVENT` structures are available. |
| `OS_PRIO_INVALID` | if you specify a priority with a higher number than `OS_LOWEST_PRIO`. |

## Returned Value

A pointer to the event control block allocated to the mutex. If no event control block is available, `OSMutexCreate()` returns 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 use the mutex to access the resource. For example, if three 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);
```

| *Chapter* | *File* | *Called from* | *Code enabled by* |
|---|---|---|---|
| 8 | OS_MUTEX.C | Task | OS_MUTEX_EN and OS_MUTEX_DEL_EN |

`OSMutexDel()` is used to delete a mutex. This function is dangerous 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 delete a mutex, you must first delete all the tasks that can 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 are readied.

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

| | |
|---|---|
| `OS_NO_ERR` | if the call is successful and the mutex has been deleted. |
| `OS_ERR_DEL_ISR` | if you attempt to delete a mutex from an ISR. |
| `OS_ERR_INVALID_OPT` | if you don't specify one of the two options mentioned in the `opt` argument. |
| `OS_ERR_TASK_WAITING` | if one or more task are waiting on the mutex and you specify `OS_DEL_NO_PEND`. |
| `OS_ERR_EVENT_TYPE` | if `pevent` is not pointing to a mutex. |
| `OS_ERR_PEVENT_NULL` | if no more `OS_EVENT` structures are available. |

## Returned Value

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

## Notes/Warnings

1. You should use this call with care because other tasks might 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);
        if (DispMutex == (OS_EVENT *)0) {
            /* Mutex has been deleted */
        }
        .
        .
    }
}
```

# OSMutexPend()

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

| *Chapter* | *File* | *Called from* | *Code enabled by* |
|---|---|---|---|
| 8 | 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()` gives the mutex to the caller and returns to its caller. Note that nothing is actually given to the caller except for the fact that if `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()` places the calling task in the wait list for the mutex. The task thus waits until the task that owns the mutex releases the mutex and thus the resource or until the specified timeout expires. If the mutex is signaled before the timeout expires, _C/OS-II resumes the highest priority task that is waiting for the mutex. Note that if the mutex is owned by a lower priority task, then `OSMutexPend()` raises the priority of the task that owns the mutex to the 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 65,535 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 that is used to hold an error code. `OSMutexPend()` sets `*err` to one of the following:

| | |
|---|---|
| `OS_NO_ERR` | if the call is successful and the mutex is available. |
| `OS_TIMEOUT` | if the mutex is not available within the specified timeout. |
| `OS_ERR_EVENT_TYPE` | if you don'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 attempt to acquire the mutex from an ISR. |

## Returned Value

none

## Notes/Warnings

1. Mutexes must be created before they are used.
2. You should **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 delay the task that owns the mutex. In other words, your code should hurry up and release the resource as quickly 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);
```

| *Chapter* | *File* | *Called from* | *Code enabled by* |
|---|---|---|---|
| 8 | OS_MUTEX.C | Task | OS_MUTEX_EN |

A mutex is signaled (i.e., released) by calling `OSMutexPost()`. You call this function only if you acquire the mutex by first calling either `OSMutexAccept()` or `OSMutexPend()`. If the priority of the task that owns the mutex has been raised when a higher priority task attempts to acquire the mutex, the original task priority of the task is 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 is 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 is successful and the mutex is released. |
| `OS_ERR_EVENT_TYPE` | if you don'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 attempt to call `OSMutexPost()` from an ISR. |
| `OS_ERR_NOT_MUTEX_OWNER` | if the task posting (i.e., signaling the mutex) doesn't actually own 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);
```

| *Chapter* | *File* | *Called from* | *Code enabled by* |
|---|---|---|---|
| 8 | OS_MUTEX.C | Task | OS_MUTEX_EN && OS_MUTEX_QUERY_EN |

OSMutexQuery() is used to obtain run-time information about a mutex. Your application must allocate an OS_MUTEX_DATA data structure that is 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 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 is successful. |
| OS_ERR_EVENT_TYPE | if you don'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 attempt 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;
                .
                .
            }
        }
        .
        .
    }
}
```

# OSQAccept()

```
void *OSQAccept(OS_EVENT *pevent, INT8U *err);
```

| *Chapter* | *File* | *Called from* | *Code enabled by* |
|---|---|---|---|
| 11 | OS_Q.C | Task or ISR | OS_Q_EN |

`OSQAccept()` checks to see if a message is available in the desired message queue. Unlike `OSQPend()`, `OSQAccept()` does not suspend the calling task if a message is not available. In other words, `OSQAccept()` is non-blocking. If a message is available, it is extracted from the queue and returned to your application. This call is typically used by ISRs because an ISR is not allowed to wait for messages at a queue.

## Arguments

`pevent` is a pointer to the message queue from which the message is received. This pointer is returned to your application when the message queue is created [see `OSQCreate()`].

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

| | |
|---|---|
| `OS_NO_ERR` | if the call is successful and the mutex is available. |
| `OS_ERR_EVENT_TYPE` | if you don't pass a pointer to a queue to `OSQAccept()`. |
| `OS_ERR_PEVENT_NULL` | if `pevent` is a `NULL` pointer. |
| `OS_Q_EMPTY` | if the queue doesn't contain any messages. |

## Returned Value

A pointer to the message if one is available; `NULL` if the message queue does not contain a message or the message received is a `NULL` pointer. If a message was available in the queue, it will be removed before `OSQAccept()` returns.

## Notes/Warnings

1. Message queues must be created before they are used.
2. The API (Application Programming Interface) has changed for this function in V2.60 becausee you can now post `NULL` pointers to queues. Specifically, the `err` argument has been added to the call.

## Example

```
OS_EVENT *CommQ;

void Task (void *pdata)
{
      void *msg;

      pdata = pdata;
      for (;;) {
         msg = OSQAccept(CommQ);      /* Check queue for a message   */
         if (msg != (void *)0) {
            .                         /* Message received, process   */
            .
         } else {
            .                         /* Message not received, do .. */
            .                         /* .. something else           */
         }
         .
         .
      }
}
```

# OSQDel()

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

| *Chapter* | *File* | *Called from* | *Code enabled by* |
|---|---|---|---|
| 11 | OS_Q.C | Task | OS_Q_EN and OS_Q_DEL_EN |

OSQDel() is used to delete a message queue. This function is dangerous 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 delete a queue, you must first delete all the tasks that can 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 are readied.

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

| | |
|---|---|
| OS_NO_ERR | if the call is successful and the queue has been deleted. |
| OS_ERR_DEL_ISR | if you attempt to delete the queue from an ISR. |
| OS_ERR_INVALID_OPT | if you don't specify one of the two options mentioned in the opt argument. |
| OS_ERR_TASK_WAITING | if one or more tasks are waiting for messages at the message queue. |
| OS_ERR_EVENT_TYPE | if pevent is not pointing to a queue. |
| OS_ERR_PEVENT_NULL | if no more OS_EVENT structures are available. |

## Returned Value

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

## Notes/Warnings

1. You should use this call with care because other tasks might expect the presence of the queue.
2. Interrupts are disabled when pended tasks are readied, which means that interrupt latency depends on the number of tasks that are 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);
        if (DispQ == (OS_EVENT *)0) {
            /* Queue has been deleted */
        }
        .
        .
    }
}
```

# OSQFlush()

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

| *Chapter* | *File* | *Called from* | *Code enabled by* |
|---|---|---|---|
| 11 | OS_Q.C | Task or ISR | OS_Q_EN && OS_Q_FLUSH_EN |

`OSQFlush()` empties the contents of the message queue and eliminates all the messages sent to the queue. This function takes the same amount of time to execute regardless of whether tasks are waiting on the queue (and thus no messages are present) or the queue contains one or more messages.

## Arguments

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

## Returned Value

`OSQFlush()` returns one of the following codes:

| | |
|---|---|
| `OS_NO_ERR` | if the message queue is flushed. |
| `OS_ERR_EVENT_TYPE` | if you attempt to flush an object other than a message queue. |
| `OS_ERR_PEVENT_NULL` | if `pevent` is a `NULL` pointer. |

## Notes/Warnings

1. Queues must be created before they are used.
2. You should use this function with great care because, when to flush the queue, you LOOSE the references to what the queue entries are pointing to and thus, you could cause 'memory leaks'. In other words, the data you are pointing to that's being referenced by the queue entries should, most likely, need to be de-allocated (i.e. freed). To flush a queue that contains entries, you should instead repeateadly use `OSQAccept()`.

## Example

```
OS_EVENT *CommQ;



void main (void)
{
       INT8U err;


       OSInit();                            /* Initialize µC/OS-II   */
       .
       .
       err = OSQFlush(CommQ);
       .
       .
       OSStart();                           /* Start Multitasking    */
}
```

# OSQPend()

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

| *Chapter* | *File* | *Called from* | *Code enabled by* |
|---|---|---|---|
| 11 | OS_Q.C | Task only | OS_Q_EN |

OSQPend() is used when a task wants to receive messages from a queue. The messages are sent to the task either by an ISR or by another task. The messages received are pointer-sized variables, and their use is application specific. If at least one message is present at the queue when OSQPend() is called, the message is retrieved and returned to the caller. If no message is present at the queue, OSQPend() suspends the current task until either a message is received or a user-specified timeout expires. If a message is sent to the queue and multiple tasks are waiting for such a message, then µC/OS-II resumes the highest priority task that is waiting. A pended task that has been suspended with OSTaskSuspend() can receive a message. However, the task remains suspended until it is resumed by calling OSTaskResume().

## Arguments

pevent is a pointer to the queue from which the messages are received. This pointer is returned to your application when the queue is created [see OSQCreate()].

timeout allows the task to resume execution if a message is not received from the mailbox within the specified number of clock ticks. A timeout value of 0 indicates that the task wants to wait forever for the message. The maximum timeout is 65,535 clock ticks. The timeout value is not synchronized with the clock tick. The timeout count starts decrementing on the next clock tick, which could potentially occur immediately.

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

| | |
|---|---|
| OS_NO_ERR | if a message is received. |
| OS_TIMEOUT | if a message is not received within the specified timeout. |
| OS_ERR_EVENT_TYPE | if pevent is not pointing to a message queue. |
| OS_ERR_PEVENT_NULL | if pevent is a NULL pointer. |
| OS_ERR_PEND_ISR | if you call this function from an ISR and µC/OS-II has to suspend it. In general, you should not call OSQPend() from an ISR. _C/OS-II checks for this situation anyway. |

## Returned Value

OSQPend() returns a message sent by either a task or an ISR, and *err is set to OS_NO_ERR. If a timeout occurs, OSQPend() returns a NULL pointer and sets *err to OS_TIMEOUT.

## Notes/Warnings

1. Queues must be created before they are used.
2. You should not call OSQPend() from an ISR.
3. OSQPend() was changed in V2.60 to allow it to receive NULL pointer messages.

## Example

```
OS_EVENT *CommQ;



void CommTask(void *data)
{
      INT8U  err;
      void  *msg;



      pdata = pdata;
      for (;;) {
         .
         .
         msg = OSQPend(CommQ, 100, &err);
         if (err == OS_NO_ERR) {
            .
            .         /* Message received within 100 ticks!         */
            .
         } else {
            .
            .         /* Message not received, must have timed out  */
            .
         }
         .
         .
      }
}
```

# OSQPost()

```
INT8U OSQPost(OS_EVENT *pevent, void *msg);
```

| *Chapter* | *File* | *Called from* | *Code enabled by* |
|---|---|---|---|
| 11 | OS_Q.C | Task or ISR | OS_Q_EN && OS_Q_POST_EN |

`OSQPost()` sends a message to a task through a queue. A message is a pointer-sized variable, and its use is application specific. If the message queue is full, an error code is returned to the caller. In this case, `OSQPost()` immediately returns to its caller, and the message is not placed in the queue. If any task is waiting for a message at the queue, the highest priority task receives the message. If the task waiting for the message has a higher priority than the task sending the message, the higher priority task resumes, and the task sending the message is suspended; that is, a context switch occurs. Message queues are first-in first-out (FIFO), which means that the first message sent is the first message received.

## Arguments

pevent is a pointer to the queue into which the message is deposited. This pointer is returned to your application when the queue is created [see `OSQCreate()`].

msg is the actual message sent to the task. `msg` is a pointer-sized variable and is application specific. As of V2.60, you are allowed to post a `NULL` pointer.

## Returned Value

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

| | |
|---|---|
| `OS_NO_ERR` | if the message is deposited in the queue. |
| `OS_Q_FULL` | if the queue is already full. |
| `OS_ERR_EVENT_TYPE` | if `pevent` is not pointing to a message queue. |
| `OS_ERR_PEVENT_NULL` | if `pevent` is a `NULL` pointer. |

## Notes/Warnings

1. Queues must be created before they are used.
2. As of V2.60, you are now allowed to post a `NULL` pointer. It is up to you're application to check the `err` variable accordingly.

## Example

```
OS_EVENT *CommQ;
INT8U     CommRxBuf[100];



void CommTaskRx (void *pdata)
{
      INT8U  err;



      pdata = pdata;
      for (;;) {
         .
         .
         err = OSQPost(CommQ, (void *)&CommRxBuf[0]);
         switch (err) {
             case OS_NO_ERR:
                  /* Message was deposited into queue    */
                  break;


             case OS_Q_FULL:
                  /* Queue is full                       */
                  Break;
            .
         }
         .
         .
      }
}
```

# OSQPostFront()

```
INT8U OSQPostFront(OS_EVENT *pevent, void *msg);
```

| *Chapter* | *File* | *Called from* | *Code enabled by* |
|---|---|---|---|
| 11 | OS_Q.C | Task or ISR | OS_Q_EN && OS_Q_POST_FRONT_EN |

`OSQPostFront()` sends a message to a task through a queue. `OSQPostFront()` behaves very much like `OSQPost()`, except that the message is inserted at the front of the queue. This means that `OSQPostFront()` makes the message queue behave like a last-in first-out (LIFO) queue instead of a first-in first-out (FIFO) queue. The message is a pointer-sized variable, and its use is application specific. If the message queue is full, an error code is returned to the caller. `OSQPostFront()` immediately returns to its caller, and the message is not placed in the queue. If any tasks are waiting for a message at the queue, the highest priority task receives the message. If the task waiting for the message has a higher priority than the task sending the message, the higher priority task is resumed, and the task sending the message is suspended; that is, a context switch occurs.

## Arguments

`pevent` is a pointer to the queue into which the message is deposited. This pointer is returned to your application when the queue is created [see `OSQCreate()`].

`msg` is the actual message sent to the task. `msg` is a pointer-sized variable and is application specific. As of V2.60, you are allowed to post a `NULL` pointer.

## Returned Value

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

| | |
|---|---|
| `OS_NO_ERR` | if the message is deposited in the queue. |
| `OS_Q_FULL` | if the queue is already full. |
| `OS_ERR_EVENT_TYPE` | if `pevent` is not pointing to a message queue. |
| `OS_ERR_PEVENT_NULL` | if `pevent` is a `NULL` pointer. |

## Notes/Warnings

1. Queues must be created before they are used.
2. As of V2.60, you are now allowed to post a `NULL` pointer. It is up to you're application to check the `err` variable accordingly.

## Example

```
OS_EVENT *CommQ;
INT8U     CommRxBuf[100];



void CommTaskRx (void *pdata)
{
      INT8U  err;



      pdata = pdata;
      for (;;) {
         .
         .
         err = OSQPostFront(CommQ, (void *)&CommRxBuf[0]);
         switch (err) {
             case OS_NO_ERR:
                  /* Message was deposited into queue    */
                  break;


             case OS_Q_FULL:
                  /* Queue is full                       */
                  break;
            .
         }
         .
         .
      }
}
```

# OSQPostOpt()

```
INT8U OSQPostOpt(OS_EVENT *pevent, void *msg, INT8U opt);
```

| *Chapter* | *File* | *Called from* | *Code enabled by* |
|---|---|---|---|
| 11 | OS_Q.C | Task or ISR | OS_Q_EN && OS_Q_POST_OPT_EN |

OSQPostOpt() is used to send a message to a task through a queue. A message is a pointer-sized variable, and its use is application specific. If the message queue is full, an error code is returned indicating that the queue is full. OSQPostOpt() then immediately returns to its caller, and the message is not placed in the queue. If any task is waiting for a message at the queue, OSQPostOpt() allows you to either post the message to the highest priority task waiting at the queue (opt set to OS_POST_OPT_NONE) or to all tasks waiting at the queue (opt is set to OS_POST_OPT_BROADCAST). In either case, scheduling occurs, and, if any of the tasks that receive the message have a higher priority than the task that is posting the message, then the higher priority task is resumed, and the sending task is suspended. In other words, a context switch occurs.

OSQPostOpt() emulates both OSQPost() and OSQPostFront() and also allows you to post a message to **multiple** tasks. In other words, it allows the message posted to be broadcast to **all** tasks waiting on the queue. OSQPostOpt() can actually replace OSQPost() and OSQPostFront() because you specify the mode of operation via an option argument, opt. Doing this allows you to reduce the amount of code space needed by µC/OS-II.

## Arguments

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

msg is the actual message sent to the task(s). msg is a pointer-sized variable, and what msg points to is application specific. As of V2.60, you are now allowed to post a NULL pointer.

opt determines the type of POST performed:

| | |
|---|---|
| OS_POST_OPT_NONE | POST to a single waiting task [identical to OSQPost()]. |
| OS_POST_OPT_BROADCAST | POST to **all** tasks waiting on the queue. |
| OS_POST_OPT_FRONT | POST as LIFO [simulates OSQPostFront()]. |

Below is a list of **all** the possible combination of these flags:

| | |
|---|---|
| OS_POST_OPT_NONE | is identical to OSQPost() |
| OS_POST_OPT_FRONT | is identical to OSQPostFront() |
| OS_POST_OPT_BROADCAST | is identical to OSQPost() but broadcasts msg to **all** waiting tasks |
| OS_POST_OPT_FRONT + OS_POST_OPT_BROADCAST | is identical to OSQPostFront() except that broadcasts msg to **all** waiting tasks. |

## Returned Value

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

| | |
|---|---|
| OS_NO_ERR | if the call is successful and the message has been sent. |
| OS_Q_FULL | if the queue can no longer accept messages because it is full. |
| OS_ERR_EVENT_TYPE | if pevent is not pointing to a mailbox. |
| OS_ERR_PEVENT_NULL | if pevent is a NULL pointer. |

## Notes/Warnings

1. Queues must be created before they are used.
2. If you need to use this function and want to reduce code space, you can disable code generation of `OSQPost()` (set `OS_Q_POST_EN` to 0 in `OS_CFG.H`) and `OSQPostFront()` (set `OS_Q_POST_FRONT_EN` to 0 in `OS_CFG.H`) because `OSQPostOpt()` can emulate these two functions.
3. The execution time of `OSQPostOpt()` depends on the number of tasks waiting on the queue if you set `opt` to `OS_POST_OPT_BROADCAST`.

## Example

```
OS_EVENT *CommQ;
INT8U     CommRxBuf[100];



void CommRxTask (void *pdata)
{
    INT8U  err;



    pdata = pdata;
    for (;;) {
        .
        .
        err = OSQPostOpt(CommQ, (void *)&CommRxBuf[0],
OS_POST_OPT_BROADCAST);
        .
        .
    }
}
```

# OSSemDel()

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

| *Chapter* | *File* | *Called from* | *Code enabled by* |
|---|---|---|---|
| 7 | OS_SEM.C | Task | OS_SEM_EN and OS_SEM_DEL_EN |

`OSSemDel()` is used to delete a semaphore. This function is dangerous 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 delete a semaphore, you must first delete all the tasks that can 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 are readied.

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

| | |
|---|---|
| `OS_NO_ERR` | if the call is successful and the semaphore has been deleted. |
| `OS_ERR_DEL_ISR` | if you attempt to delete the semaphore from an ISR. |
| `OS_ERR_INVALID_OPT` | if you don't specify one of the two options mentioned in the `opt` argument. |
| `OS_ERR_TASK_WAITING` | if one or more tasks are waiting on the semaphore. |
| `OS_ERR_EVENT_TYPE` | if `pevent` is not pointing to a semaphore. |
| `OS_ERR_PEVENT_NULL` | if no more `OS_EVENT` structures are available. |

## Returned Value

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

## Notes/Warnings

1. You should use this call with care because other tasks might expect the presence of the semaphore.
2. Interrupts are disabled when pended tasks are readied, which means that interrupt latency depends on the number of tasks that are 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);
        if (DispSem == (OS_EVENT *)0) {
            /* Semaphore has been deleted */
        }
        .
        .
    }
}
```

# OSTaskNameGet()

```
INT8U OSTaskNameGet(INT8U prio, char *pname, INT8U *err);
```

| Chapter | File | Called from | Code enabled by |
|---|---|---|---|
| New in V2.60 | OS_TASK.C | Task or ISR | OS_TASK_NAME_SIZE |

`OSTaskNameGet()` allows you to obtain the name that you assigned to a task. The name is an ASCII string and the size of the name can contain up to `OS_TASK_NAME_SIZE` characters (including the `NUL` termination). This function is typically used by a debugger to allow associating a name to a task.

## Arguments

`prio` is the priority of the task from which you would like to obtain the name from. If you specify `OS_PRIO_SELF`, you would obtain the name of the current task.

`pname` is a pointer to an ASCII string that will receive the name of the task. The string must be able to hold at least `OS_TASK_NAME_SIZE` characters (including the `NUL` character).

`err` a pointer to an error code and can be any of the following:

| | |
|---|---|
| `OS_NO_ERR` | If the name of the task was copied to the array pointed to by `pname`. |
| `OS_TASK_NOT_EXIST` | The task you specified was not created or has been deleted. |
| `OS_PRIO_INVALID` | If you specified an invalid priority - a priority higher than the idle task (`OS_LOWEST_PRIO`) or you didn't specify `OS_PRIO_SELF`. |

## Returned Values

The size of the ASCII string placed in the array pointed to by `pname` or `0` if an error is encountered.

## Notes/Warnings

1. The task must be created before you can use this function and obtain the name of the task.
2. You must ensure that you have sufficient storage in the destination string to hold the name of the task.

## Example

```
char   EngineTaskName[30];


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



    pdata = pdata;
    for (;;) {
        size = OSTaskNameGet(OS_PRIO_SELF, &EngineTaskName[0], &err);
        .
        .
    }
}
```

# OSTaskNameSet()

```
void OSTaskNameSet(INT8U prio, char *pname, INT8U *err);
```

| Chapter | File | Called from | Code enabled by |
|---|---|---|---|
| New in V2.60 | OS_TASK.C | Task or ISR | OS_TASK_NAME_SIZE |

OSTaskNameSet() allows you to assign a name to a task. The name is an ASCII string and the size of the name can contain up to OS_TASK_NAME_SIZE characters (including the NUL termination). This function is typically used by a debugger to allow associating a name to a task.

## Arguments

prio is the priority of the task that you want to name. If you specify OS_PRIO_SELF, you would set the name of the current task.

pname is a pointer to an ASCII string that hold the name of the task. The string must be smaller than or equal to OS_TASK_NAME_SIZE characters (including the NUL character).

err a pointer to an error code and can be any of the following:

| | |
|---|---|
| OS_NO_ERR | If the name of the task was set. |
| OS_TASK_NOT_EXIST | The task you specified was not created or has been deleted. |
| OS_PRIO_INVALID | If you specified an invalid priority - a priority higher than the idle task (OS_LOWEST_PRIO) or you didn't specify OS_PRIO_SELF. |

## Returned Values

None.

## Notes/Warnings

1. The task must be created before you can use this function to set the name of the task.

## Example

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



    pdata = pdata;
    for (;;) {
        OSTaskNameSet(OS_PRIO_SELF, "Engine Task", &err);
        .
        .
    }
}
```

# References

***µC/OS-II, The Real-Time Kerne, 2nd EditionI***
Jean J. Labrosse
CMP Books, 2002
ISBN 1-57820-103-9

# 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

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