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_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_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_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.

The following table summarizes some of the new #define constants in OS_CFG.H which were all added in V2.51.

#define name in OS_CFG.H	**... to enable the function:**
OS_FLAG_ACCEPT_EN	OSFlagAccept()
OS_FLAG_DEL_EN	OSFlagDel()
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_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_QUERY_EN	OSTaskQuery()
OS_TIME_DLY_HMSM_EN	OSTimeDlyHMSM()
OS_TIME_DLY_RESUME_EN	OSTimeDlyResume()
OS_TIME_GET_SET_EN	OSTimeGet() and OSTimeSet()
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.

OSFlagAccept()

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

File	Called from	Code enabled by	Version
OS_FLAG.C	Task	OS_FLAG_EN && OS_FLAG_ACCEPT_EN	V2.51

`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()` except that the caller will 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 argument:

`OS_FLAG_WAIT_CLR_ALL`	You will check ALL bits in 'flags' to be clear (0)
`OS_FLAG_WAIT_CLR_ANY`	You will check ANY bit in 'flags' to be clear (0)
`OS_FLAG_WAIT_SET_ALL`	You will check ALL bits in 'flags' to be set (1)
`OS_FLAG_WAIT_SET_ANY`	You will 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:

`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 you are waiting for are not available.

Returned Value

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);`

File	Called from	Code enabled by	Version
OS_FLAG.C	Task or startup code	OS_FLAG_EN	V2.51

`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 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 event flag group was created.
`OS_ERR_CREATE_ISR`	if you attempted to create an event flag group from an ISR.
`OS_FLAG_GRP_DEPLETED`	if there are no more event flag groups available. You will need to increase the value of `OS_MAX_FLAGS` in `OS_CFG.H`.

Returned Value

A pointer to the event flag group if a free one is available. If no event flag group is available, `OSFlagCreate()` will return 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                                   */
    .
    .
    EngineStatus = OSFlagCreate(0x00, &err);  /* Create an event flag group containing the engine’s status   */
    .
    .
    OSStart();                           /* Start Multitasking                                    */
}
```

OSFlagDel()

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

File	Called from	Code enabled by	Version
OS_FLAG.C	Task	OS_FLAG_EN and OS_FLAG_DEL_EN	V2.51

`OSFlagDel()` is used to delete an event flag group. This is a dangerous function 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 would delete an event flag group, you would 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 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 event flag group was deleted.
`OS_ERR_DEL_ISR`	if you attempted to delete an event flag group from an ISR.
`OS_ERR_EVENT_TYPE`	if `pgrp` is not pointing to an event flag group.
`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 event flag group and and you specified `OS_DEL_NO_PEND`.
`OS_FLAG_INVALID_PGRP`	if you passed a `NULL` pointer in `pgrp`.

Returned Value

A `NULL` pointer if the event flag group is deleted or `pgrp` if the event flag group 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 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 */
        }
        .
        .
    }
}
```

OSFlagPend()

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

File	Called from	Code enabled by	Version
OS_FLAG.C	Task only	OS_FLAG_EN	V2.51

`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, ALL conditions to be either set or cleared. If the events that the calling task desires are not available then, the calling task will be 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 argument:

`OS_FLAG_WAIT_CLR_ALL`	You will check ALL bits in 'flags' to be clear (0)
`OS_FLAG_WAIT_CLR_ANY`	You will check ANY bit in 'flags' to be clear (0)
`OS_FLAG_WAIT_SET_ALL`	You will check ALL bits in 'flags' to be set (1)
`OS_FLAG_WAIT_SET_ANY`	You will check ANY bit in 'flags' to be set (1)

You can also specify whether the flags will be '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` a pointer to an error code and can be:

`OS_NO_ERR`	No error
`OS_ERR_PEND_ISR`	You tried to call `OSFlagPend` from an ISR which is not allowed.
`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 you are waiting for are not available.
`OS_TIMEOUT`	The flags were not available within the specified amount of time.

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;
        }
        .
        .
    }
}
```

OSFlagPost()

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

File	Called from	Code enabled by	Version
OS_FLAG.C	Task or ISR	OS_FLAG_EN	V2.51

You set or clear event flag bits by calling `OSFlagPost()`. The bits set or cleared are specified in a 'bit mask'. `OSFlagPost()` will ready each task that has it's 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` (see below) is `OS_FLAG_SET`, each bit that is set in `'flags'` will set the corresponding bit in the event flag group. e.g. to set bits 0, 4 and 5 you would set `flags` to 0x31 (note, bit 0 is least significant bit). If `opt` (see below) is `OS_FLAG_CLR`, each bit that is set in `flags` will CLEAR the corresponding bit in the event flag group. e.g. to clear bits 0, 4 and 5 you would specify `'flags'` as 0x31 (note, bit 0 is least significant bit).

`opt` indicates whether the flags will be 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 was successfull
`OS_FLAG_INVALID_PGRP`	You passed a `NULL` pointer
`OS_ERR_EVENT_TYPE`	You are not pointing to an event flag group
`OS_FLAG_INVALID_OPT`	You specified 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);`

File	Called from	Code enabled by	Version
OS_FLAG.C	Task or ISR	OS_FLAG_EN && OS_FLAG_QUERY_EN	V2.51

`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 was successfull
`OS_FLAG_INVALID_PGRP`	You passed a `NULL` pointer
`OS_ERR_EVENT_TYPE`	You are not pointing to an event flag group

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

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

```
OS_FLAG_GRP *EngineStatusFlags;


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


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

OSMboxDel()

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

File	Called from	Code enabled by	Version
OS_MBOX.C	Task	OS_MBOX_EN and OS_MBOX_DEL_EN	V2.04

`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);
        .
        .
    }
}
```

OSMboxPostOpt()

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

File	Called from	Code enabled by	Version
OS_MBOX.C	Task or ISR	OS_MBOX_EN and OS_MBOX_POST_OPT_EN	V2.51

`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 to either 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 will occur and if any of the task that receives the message has a higher priority than the task that is posting the message then, the higher priority task will be resumed and the sending task will be suspended. In other words, a context switch will occur.

`OSMboxPostOpt()` works just like `OSMboxPost()` except that it 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 mailbox. `OSMboxPostOpt()` can actually replace `OSMboxPost()` because it can emulate `OSMboxPost()`.

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 what msg points to is application specific. You must never post a NULL pointer because this 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 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 message was 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.

Notes/Warnings

Mailboxes must be created before they are used.

You must NEVER post a `NULL` pointer to a mailbox because this indicates that the mailbox is empty.

If you need to use this function and want to reduce code space, you may disable code generation of `OSMboxPost()` since `OSMboxPostOpt()` can emulate `OSMboxPost()`.

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);
```

File	Called from	Code enabled by	Version
OS_MUTEX.C	Task	OS_MUTEX_EN	V2.04

`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	Version
OS_MUTEX.C	Task or startup code	OS_MUTEX_EN	V2.04

`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	Version
OS_MUTEX.C	Task	OS_MUTEX_EN and OS_MUTEX_DEL_EN	V2.04

`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	Version
OS_MUTEX.C	Task only	OS_MUTEX_EN	V2.04

`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	Version
OS_MUTEX.C	Task	OS_MUTEX_EN	V2.04

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	Version
OS_MUTEX.C	Task	OS_MUTEX_EN && OS_MUTEX_QUERY_EN	V2.04

`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	Version
OS_Q.C	Task	OS_Q_EN and OS_Q_DEL_EN	V2.04

`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);
        .
        .
    }
}
```

OSQPostOpt()

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

File	Called from	Code enabled by	Version
OS_Q.C	Task or ISR	OS_Q_EN and OS_Q_POST_OPT_EN	V2.51

`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 will occur and if any of the task that receives the message has a higher priority than the task that is posting the message then, the higher priority task will be resumed and the sending task will be suspended. In other words, a context switch will occur.

`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 thus actually replace `OSQPost()` and `OSQPostFront()` because you specify the mode of operation via an option argument, `opt`.

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. You must never 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 that are 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:

1) `OS_POST_OPT_NONE` identical to `OSQPost()`

2) `OS_POST_OPT_FRONT` identical to `OSQPostFront()`

3) `OS_POST_OPT_BROADCAST` identical to `OSQPost()` but will broadcast `msg` to ALL waiting tasks

4) `OS_POST_OPT_FRONT + OS_POST_OPT_BROADCAST` is identical to `OSQPostFront()` except that will broadcast `msg` to ALL waiting tasks.

Returned Value

`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 message was 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.
`OS_ERR_POST_NULL_PTR`	if you are attempting to post a `NULL` pointer.

Notes/Warnings

Queues must be created before they are used.

You must NEVER post a `NULL` pointer to a queue.

If you need to use this function and want to reduce code space, you may disable code generation of `OSQPost()` and `OSQPostFront()` since `OSQPostOpt()` can emulate `OSQPost()` and `OSQPostFront()`.

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);`

File	Called from	Code enabled by	Version
OS_SEM.C	Task	OS_SEM_EN and OS_SEM_DEL_EN	V2.04

`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