μC/OS-II V2.92.11

Thread Safety of the Compiler's Run-Time Library

As of V2.92.08, µC/OS-II provides built-in support for run-time library thread safety through the use of Task Local Storage (TLS) for storage of task-specific run-time library static data and mutual exclusion semaphores to protect accesses to shared resources.

The run-time environment consists of the run-time library, which contains the functions defined by the C and the C++ standards, and includes files that define the library interface (the system header files). Compilers provide complete libraries that are compliant with Standard C and C++. These libraries also supports floating-point numbers in IEEE 754 format and can be configured to include different levels of support for locale, file descriptors, multi-byte characters, etc. Most parts of the libraries are reentrant, but some functionality and parts are not reentrant because they require the use of static data. Different compilers provide different methods to add reentrancy to their libraries through an API defined by the tool chain supplier.

In a multi-threaded environment the C/C++ library has to handle all library objects with a global state differently. Either an object is a true global object, then any updates of its state has to be guarded by some locking mechanism to make sure that only one task can update it at any one time, or an object is local to each task, then the static variables containing the objects state must reside in a variable area local for the task. This area is commonly named thread local storage or, TLS.

The run-time library may also need to use multiple types of locks. For example, a lock could be necessary to ensure exclusive access to the file stream, another one to the heap, etc. It is thus common to protect the following functions through one or more semaphores:

- The heap through the usage of `malloc()`, `free()`, `realloc()`, and `calloc()`.
- The file system through the usage of `fopen()`, `fclose()`, `fdopen()`, `fflush()`, and
- `freopen()`.
- The signal system through the usage of `signal()`.
- The tempfile system through the usage of `tmpnam()`.
- Initialization of static function objects.

Thread-local storage is typically needed for the following library objects:

- Error functions through `errno` and `strerror`
- Locale functions through the usage of `localeconv()` and `setlocale()`
- Time functions through the usage of `asctime()`, `localtime()`, `gmtime()`, and
- `mktime()`
- Multibyte functions through the usage of `mbrlen()`, `mbrtowc()`, `mbsrtowc()`,
- `mbtowc()`, `wcrtomb()`, `wcsrtomb()`, and `wctomb()`
- Random functions through the usage of `rand()` and `srand()`
- Other functions through the usage of `atexit()` and `strtok()`
- C++ exception engine

Different compilers require different implementations and those implementation details are encapsulated into a single file called `os_tls.c`. There is thus one `os_tls.c` file associated with each compiler supported by Micrium and each implementation is placed in its own directory as follows:

```
\Micrium\Software\uCOS-III\TLS\<compiler manufacturer>\os_tls.c
```

Where 'compiler manufacturer' is the name of the compiler manufacturer or the code name for the compiler for which thread safety has been implemented. Refer to the code distribution to see if your compiler is supported.

ENABLING THREAD SAFETY

In order to enable thread safety, you need to do the following:

- Set `OS_TLS_TBL_SIZE` in `os_cfg.h` to a value greater than `1`. The actual value depends on the number of entries needed by the compiler used. In most cases you would set this to `5` but you should consult the `os_tls.c` that you plan to use for additional information.

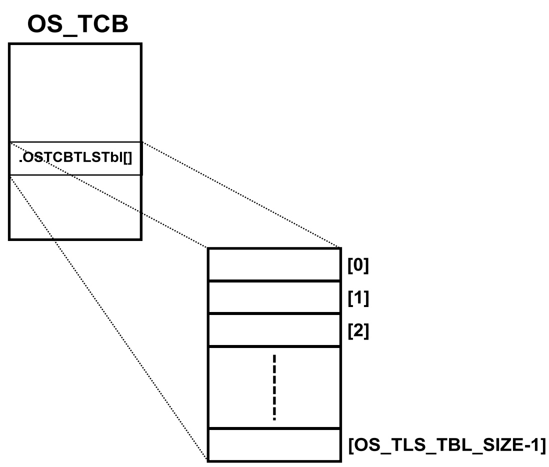
- Add to your build, the `os_tls.c` file that corresponds to the compiler you are using.

- Depending on the compiler and how TLS is allocated, you may also need to make sure that you have a heap. Consult your compiler documentation on how you can enable the heap and determine its size.

- Most likely, `os_tls.c` will make use of semaphores to guard access to shared resources (such as the heap or files) then you need to make sure `OS_SEM_EN` is set to `1` in `os_cfg.h`. Also, the run-time library may already define APIs to lock and unlock sections of code. The implementation of these functions should also be part of `os_tls.c`.

TASK SPECIFIC STORAGE

When `OS_TLS_TBL_SIZE` is set to `1` or greater, each task's `OS_TCB` will contain a new array called `.OSTCBTLSTbl[]` as shown below. Each array element is of type `OS_TLS` which is actually a pointer to `void`. This allows an `OS_TCB` to be extended so that it can have as many TLS areas as needed.



Each `OS_TCB` contains an array of `OS_TLS` when `OS_TLS_TBL_SIZE` is greater than `0` in `os_cfg.h`

The number of entries (i.e., the value to set `OS_TLS_TBL_SIZE` to) depends on the compiler being supported as well as whether TLS storage is needed for other purposes.

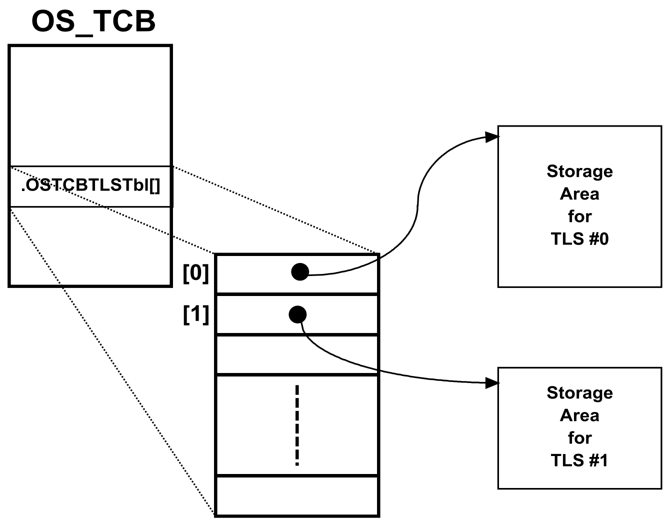
OS_TLS_GetID()

The index into `.OSTCBTLSTbl[]` is called the TLS ID and TLS IDs are assigned through an API function. In other words, TLS IDs are assigned dynamically as needed. Once a TLS ID is assigned for a specific purpose, it cannot be 'unassigned'. The function used to assign a TLS ID is called `OS_TLS_GetID()`.

OS_TLS_SetValue()

µC/OS-II sets the value of a `.OSTCBTLSTbl[]` entry by calling `OS_TLS_SetValue()`. Because TLS is specific to a given task then you will need to specify the address of the `OS_TCB` of the task, the

TLS ID that you want to set and the value to store into the table entry. Shown below is .OSTCBTLSTbl[] containing two entries (i.e., pointers) assigned by OS_TLS_SetValue().



OS_TLS_SetValue() assigns a pointer to a .OSTCBTLSTbl[] entry

OS_TLS_GetValue()

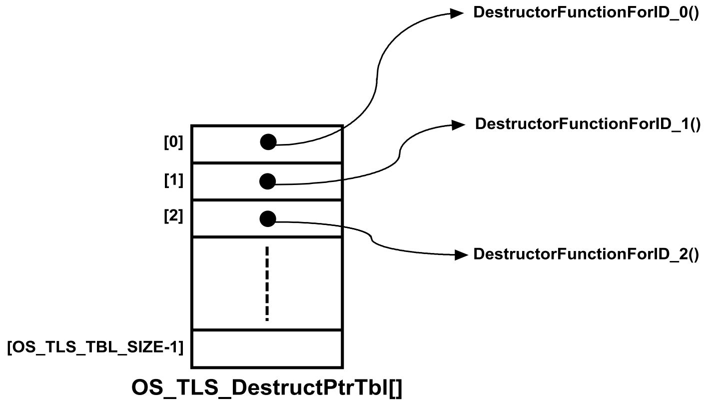
The value stored into a .OSTCBTLSTbl[] entry can be retrieved by calling OS_TLS_GetValue(). The address of the OS_TCB of the task you are interested has to be specified as part of the call as well as the desired TLS ID. OS_TLS_GetValue() returns the value stored in that task's .TLS_Tbl[] entry indexed by the TLS ID.

OS_TLS_SetDestruct()

Finally, each .OSTCBTLSTbl[] entry can have a 'destructor' associated with it. A destructor is a function that is called when the task is deleted. Destructors are common to all tasks. This means that if a destructor is assigned for a TLS ID, the same destructor will be called for all the tasks for that entry. Also, when a task is deleted, the destructor for all of the TLS IDs will be called – assuming, of course, that a destructor was assigned to the corresponding TLS ID. You set a destructor function by calling OS_TLS_SetDestruct() and specify the TLS ID associated with the destructor as well as a pointer to the function that will be called. Note that a destructor function must be declared as follows:

```
void MyDestructFunction (OS_TCB     *p_tcb,
                         OS_TLS_ID   id,
                         OS_TLS      value);
```

The drawing below shows the global destructor table. Note that not all implementations of os_tls.c will have destructors for the TLS.



Array of pointers to destructor functions (global to all tasks)

OS_TLS.C INTERNAL FUNCTIONS

There are four mandatory internal functions that needs to be implemented in `os_tls.c` if `OS_CFG_TLS_TBL_SIZE` is set to a non-zero value.

void OS_TLS_Init (void)

This function is called by `OSInit()` and in fact, is called after creating the kernel objects but before creating any of the internal µC/OS-III tasks. This means that `OS_TLS_Init()` is allowed to create event flags, semaphores, mutexes and message queues. `OS_TLS_Init()` would typically create mutexes to protect access to shared resources such as the heap or streams.

void OS_TLS_TaskCreate (OS_TCB *p_tcb)

This function is called by `OSTaskCreate()` allowing each task to allocate TLS storage as needed at task creation time. If a task needs to use a specific TLS ID, the TLS ID must have been previously assigned, most likely by the startup code in `main()` or in one of the first task that runs.

`OS_TLS_TaskCreate()` is called immediately after calling `OSTaskCreateHook()`.

You should note that you cannot call `OS_TLS_GetValue()` or `OS_TLS_SetValue()` for the specified task, unless the task has been created.

`OS_TLS_TaskCreate()` should check that TLS is a feature enabled for the task being created. This is done by examining the `OS_TCB`'s option field (i.e., `p_tcb->Opt`) as follows:

```
void OS_TLS_TaskCreate (OS_TCB *p_tcb)
{
    OS_TLS p_tls;


    if ((p_tcb->Opt & OS_OPT_TASK_NO_TLS) == OS_OPT_NONE) {
        p_tls                    = /* Allocate storage for TLS */
        p_tcb->TLS_Tbl[MyTLS_ID] = p_tls;
    }
}
```

void OS_TLS_TaskDel (OS_TCB *p_tcb)

This function is called by `OSTaskDel()` allowing each task to deallocate TLS storage that was allocated by `OS_TLS_TaskCreate()`. If the os_tls.c file implements destructor functions then `OS_TLS_Del()` should call all the destructors for the TLS IDs that have been assigned.

`OS_TLS_TaskDel()` is called by `OSTaskDel()`, immediately after calling `OSTaskDelHook()`.

The code below shows how `OS_TLS_TaskDel()` can be implemented.

```
void OS_TLS_TaskDel (OS_TCB *p_tcb)
{
    OS_TLS_ID            id;
    OS_TLS_DESTRUCT_PTR *p_tbl;


    for (id = 0; id < OS_TLS_NextAvailID; id++) {
        p_tbl = &OS_TLS_DestructPtrTbl[id];
        if (*p_tbl != (OS_TLS_DESTRUCT_PTR)0) {
           (*p_tbl)(p_tcb, id, p_tcb->TLS_Tbl[id]);
        }
    }
}
```

`OS_TLS_TaskDel()` should actually check that TLS was used by the task being deleted. This is done by examining the `OS_TCB`'s option field (i.e., `p_tcb->Opt`) as follows:

```
void OS_TLS_TaskDel (OS_TCB *p_tcb)
{
    OS_TLS_ID            id;
    OS_TLS_DESTRUCT_PTR *p_tbl;


    if ((p_tcb->Opt & OS_OPT_TASK_NO_TLS) == OS_OPT_NONE) {
        for (id = 0; id < OS_TLS_NextAvailID; id++) {
            p_tbl = &OS_TLS_DestructPtrTbl[id];
            if (*p_tbl != (OS_TLS_DESTRUCT_PTR)0) {
                (*p_tbl)(p_tcb, id, p_tcb->TLS_Tbl[id]);
            }
        }
    }
}
```

An alternate implementation is shown below where `OS_TLS_TaskDel()` needs to deallocate storage for the task is shown below.

void OS_TLS_TaskSw (void)

This function is called by `OSSched()` before invoking `OS_TASK_SW()` and also, by `OSIntExit()` before calling `OSIntCtxSw()`. When `OS_TLS_TaskSw()` is called, `OSTCBCurPtr` will point to the task being switched out and `OSTCBHighRdyPtr` will point to the task being switched in.

`OS_TLS_TaskSw()` allows you to change the "current TLS" during a context switch. For example, if a compiler uses a global pointer that points to the current TLS then, `OS_TLS_TaskSw()` could set this pointer to point to the new task's TLS.

`OS_TLS_TaskSw()` should check that TLS is desired for the task being switched in. This is done by examining the `OS_TCB`'s option field (i.e. `p_tcb->Opt`) as follows:

```
if ((p_tcb->Opt & OS_OPT_TASK_NO_TLS) == OS_OPT_NONE) {
    /* TLS option enabled for this task */
}
```

COMPILER-SPECIFIC LOCK APIs

As previously mentioned, some compilers may already have declared API functions that are called to ensure exclusive access to shared resources. For example, APIs such as `_mutex_lock_file_system()` and `_mutex_unlock_file_system()` could be required by the compiler to ensure exclusive access to the file system. `os_tls.c` might then implement these using µC/OS-III as shown below. Note that we also included the code to initialize the mutex in `OS_TLS_Init()`.

```
OS_EVENT *OS_TLS_FS_Sem;   /* Needed to ensure exclusive access to the FS */


void OS_TLS_Init (OS_ERR *p_err)
{
    OS_TLS_NextAvailID = 0u;
    OS_TLS_NewLibID = OS_TLS_GetID(p_err);
    if (*p_err != OS_ERR_NONE) {
        return;
    }
    OS_TLS_FS_Sem = OSSemCreate(1);
}


void _mutex_lock_file_system (void)
{
    INT8U  os_err;


    if (OSRunning == 0) {
        return;
    }
    OSSemPend((OS_EVENT *)OS_TLS_FS_Sem,
              (INT32U    )0u,
              (INT8U    *)&os_err);
}


void _mutex_unlock_file_system (void)
{
    INT8U err;


    if (OSRunning == 0) {
        return;
    }
    OSSemPost((OS_SEM *)OS_TLS_FS_Sem);
}
```

The compiler may require the implementation of many such API functions to ensure exclusive access to the heap, environment variables, etc. These would all be found in `os_tls.c`.

Reference Manual

This chapter provides a reference to µC/OS-II services. Each of the user-accessible kernel services is presented in alphabetical order. The following information is provided for each of the services:

- A brief description
- The function prototype
- The filename of the source code
- The `#define` constant needed to enable the code for the service
- A description of the arguments passed to the function
- A description of the returned value(s)
- Specific notes and warnings on using the service
- One or two examples of how to use the function

OS_ENTER_CRITICAL()
OS_EXIT_CRITICAL()

Chapter	*File*	*Called from*	*Code enabled by*
3	OS_CPU.H	Task or ISR	N/A

`OS_ENTER_CRITICAL()` and `OS_EXIT_CRITICAL()` are macros used to disable and enable, respectively, the processor's interrupts.

Arguments

none

Returned Values

none

Notes/Warnings

1. These macros must be used in pairs.
2. If `OS_CRITICAL_METHOD` is set to 3, your code is assumed to have allocated local storage for a variable of type `OS_CPU_SR`, which is called `cpu_sr`, as follows

```
#if OS_CRITICAL_METHOD == 3  /* Allocate storage for CPU status reg. */
    OS_CPU_SR  cpu_sr;
#endif
```

Example

```
void TaskX(void *p_arg)
{
#if OS_CRITICAL_METHOD == 3
      OS_CPU_SR   cpu_sr = 0;
 #endif


      for (;;) {
         .
         .
         OS_ENTER_CRITICAL();    /* Disable interrupts     */
         .                       /* Access critical code   */
         OS_EXIT_CRITICAL();     /* Enable  interrupts     */
         .
         .
      }
}
```

OSEventNameGet()

```
INT8U OSEventNameGet(OS_EVENT   *pevent,
                     INT8U     **pname,
                     INT8U      *perr);
```

Chapter	File	Called from	Code enabled by
New in V2.60	OS_CORE.C	Task	OS_EVENT_NAME_EN

`OSEventNameGet()` allows you to obtain the name that you assigned to a semaphore, a mutex, a mailbox or a message queue. 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 a pointer to the name of the semaphore, mutex, mailbox or queue.

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

- `OS_ERR_NONE` If `pname` now points to the name of the semaphore, mutex, mailbox or queue.
- `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`.
- `OS_ERR_NAME_GET_ISR` `You tried calling this function from an ISR.`

Returned Values

The size of the ASCII string 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

```
INT8U    *PrinterSemName;
OS_EVENT *PrinterSem;



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



    (void)p_arg;
    for (;;) {
        size = OSEventNameGet(PrinterSem, &PrinterSemName, &err);
        .
        .
    }
}
```

OSEventNameSet()

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

Chapter	File	Called from	Code enabled by
New in V2.60	OS_CORE.C	Task	OS_EVENT_NAME_EN

`OSEventNameSet()` allows you to assign a name to a semaphore, a mutex, a mailbox or a message queue. 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 the name of the semaphore, mutex, mailbox or queue.

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

`OS_ERR_NONE`	If the call was successfull.
`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`.
`OS_ERR_NAME_SET_ISR`	You called this function from an ISR.

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 or the event.

Example

```
OS_EVENT *PrinterSem;



void Task (void *p_arg)
{
    INT8U  err;



    (void)p_arg;
    for (;;) {
        OSEventNameSet(PrinterSem, "Printer #1", &err);
        .
        .
    }
}
```

OSEventPendMulti()

```
INT16U OSEventPendMulti(OS_EVENT **pevents_pend,
                        OS_EVENT **pevents_rdy,
                        void     **pmsgs_rdy,
                        INT16U     timeout,
                        INT8U     *perr);
```

Chapter	*File*	*Called from*	*Code enabled by*
	OS_CORE.C	Task only	OS_EVENT_MULTI_EN

OSEventPendMulti() is used when a task expects to wait on multiple events. If multiple events are ready when OSEventPendMulti() is called, then **all** available events and messages, if any, are returned as ready to the caller. If no events are ready, OSEventPendMulti() suspends the current task until either an event is ready or a user-specified timeout expires. If an event becomes ready and multiple tasks are waiting for the event, µC/OS-II resumes the highest priority task waiting to run.

A pended task that has been suspended with OSTaskSuspend() can still receive a message from a multi-pended mailbox or message queue or obtain a multi-pended semaphore. However, the task remains suspended until it is resumed by calling OSTaskResume().

Arguments

pevents_pend is a pointer to a null-terminated array of OS_EVENT pointers. These event pointers are returned to your application when the mailboxes, message queues, and semaphores are created [see OSMboxCreate(), OSQCreate(), and OSSemCreate()].

pevents_rdy is a pointer to an array to return the available OS_EVENT pointers. The size of the array must be greater than or equal to the size of the pevents_pend array, including the terminating NULL.

pmsgs_rdy is a pointer to an array to return messages from any multi-pended mailbox or message queue events. The size of the array must be greater than or equal to the size of the pevents_pend array, excluding the terminating NULL. Since NULL messages are valid messages, this array cannot be NULL-terminated. Instead, every available message is returned in the pmsgs_rdy array at the same index as the ready mailbox or message queue event is returned in the pevents_rdy array. All other pmsgs_rdy array indices are filled with NULL messages.

timeout allows the task to resume execution if no multi-pended event is ready within the specified number of clock ticks. A timeout value of 0 indicates that the task wants to wait forever for any of the multi-pended events. The maximum timeout is 65,535 clock ticks. The timeout value is not synchronized with the clock tick. The timeout count begins decrementing on the next clock tick, which could potentially occur immediately.

perr	is a pointer to a variable that holds an error code. OSEventPendMulti() sets *perr to one of the following:	
	OS_ERR_NONE	if any of the multi-pended events are ready; check the pevents_rdy array for which events are available.
	OS_ERR_TIMEOUT	if no multi-pended event is ready within the specified timeout.
	OS_ERR_PEND_ABORT	indicates that a multi-pended event was aborted; check the pevents_rdy array for which events were aborted.
	OS_ERR_EVENT_TYPE	if pevents_pend is not pointing to an array of valid mailbox, message queue, or semaphore events.
	OS_ERR_PEND_LOCKED	if you called this function when the scheduler is locked.
	OS_ERR_PEND_ISR	if you call this function from an ISR and µC/OS-II suspends it. In general, you should not call OSEventPendMulti() from an ISR, but µC/OS-II checks for this situation anyway.
	OS_ERR_PEVENT_NULL	if pevents_pend, pevents_rdy, or pmsgs_rdy is a NULL pointer.

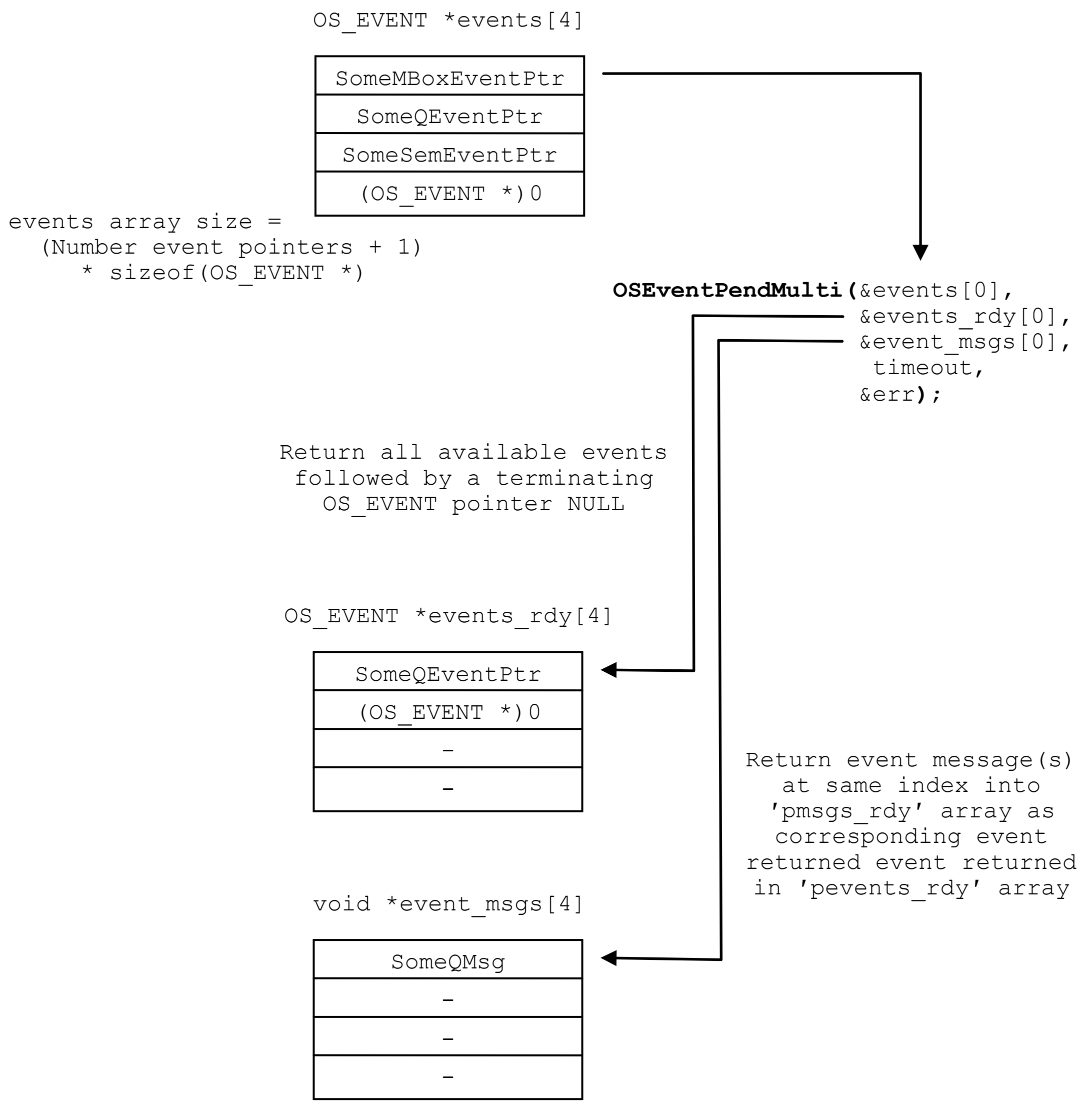
Returned Value

OSEventPendMulti() returns the number of multi-pended events that are ready or have been aborted, and *perr is set to OS_ERR_NONE or OS_ERR_PEND_ABORT, respectively. If no multi-pended event is ready within the specified timeout period or because of any error, then the pevents_rdy and pmsgs_rdy array are returned as NULL pointers, and *perr is set to OS_ERR_TIMEOUT or to the respective error.

Notes/Warnings

1. Mailbox, message queue, or semaphore events must be created before they are used.
2. You should not call OSEventPendMulti() from an ISR.
3. You cannot multi-pend on event flags and mutexes.

Example


OS_EVENT *events[4]
SomeMBoxEventPtr
SomeQEventPtr
SomeSemEventPtr
(OS_EVENT *)0
events array size =
(Number event pointers + 1)
* sizeof(OS_EVENT *)
OSEventPendMulti(&events[0],
&events_rdy[0],
&event_msgs[0],
timeout,
&err);
Return all available events
followed by a terminating
OS_EVENT pointer NULL
OS_EVENT *events_rdy[4]
SomeQEventPtr
(OS_EVENT *)0
-
-
Return event message(s)
at same index into
'pmsgs_rdy' array as
corresponding event
returned event returned
in 'pevents_rdy' array
void *event_msgs[4]
SomeQMsg
-
-
-

Example

```
void EventTask(void *p_arg)
{
    OS_EVENT  *events[4];
    OS_EVENT  *events_rdy[4];
    void      *event_msgs[4];
    INT16U     timeout;
    INT8U      err;


    (void)p_arg;
    for (;;) {
        .
        .
        events[0] = (OS_EVENT *)SomeMBoxEventPtr;
        events[1] = (OS_EVENT *)SomeQEventPtr;
        events[2] = (OS_EVENT *)SomeSemEventPtr;
        events[3] = (OS_EVENT *)0;
        events_nbr_rdy = OSEventsPendMulti(&events[0]
                                           &events_rdy[0],
                                           &event_msgs[0],
                                            timeout,
                                           &err);
        if (err == OS_ERR_NONE) {
            .
            .  /* Code for ready or aborted event(s)       */
            .
        } else {
            .
            .  /* Code for events not ready within timeout */
            .
        }
        .
        .
      }
}
```

OSFlagAccept()

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

Chapter	File	Called from	Code enabled by
9	OS_FLAG.C	Task and ISR	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`

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

`OS_ERR_NONE`	No error
`OS_ERR_EVENT_TYPE`	You are not pointing to an event flag group
`OS_ERR_FLAG_WAIT_TYPE`	You didn't specify a proper `wait_type` argument.
`OS_ERR_FLAG_INVALID_PGRP`	You passed a `NULL` pointer instead of the event flag handle.
`OS_ERR_FLAG_NOT_RDY`	The desired flags for which you are waiting are not available.

Returned Values

The flag(s) that cause the task to be ready or, 0 if either none of the flags are ready or an error occurred.

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.

IMPORTANT

The return value of `OSFlagAccept()` is different as of V2.70. In previous versions, `OSFlagAccept()` returned the current state of the flags and now, it returns the flag(s) that are ready, if any.

Example

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



OS_FLAG_GRP *EngineStatus;



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



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

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

OSFlagCreate()

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

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.

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

- `OS_ERR_NONE` 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_ERR_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       *perr);
```

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.

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

OS_ERR_NONE	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_ERR_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.
3. All tasks that were waiting for the event flag will be readied and returned an `OS_ERR_PEND_ABORT` if `OSFlagDel()` was called with `OS_DEL_ALWAYS`.

Example

```
OS_FLAG_GRP *EngineStatusFlags;



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



    (void)p_arg;
    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,
                    INT8U        **pname,
                    INT8U         *perr);
```

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

`OSFlagNameGet()` allows you to obtain the name that you assigned to an event flag group. 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 a pointer to the name of the event flag group.

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

OS_ERR_NONE	If the call was successfull.
OS_ERR_EVENT_TYPE	You are not pointing to a flag group.
OS_ERR_PNAME_NULL	You passed a `NULL` pointer for `pname`.
OS_ERR_INVALID_PGRP	You passed a `NULL` pointer for `pgrp`.

Returned Values

The size of the ASCII string 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

```
INT8U       *EngineStatusName;
OS_FLAG_GRP *EngineStatusFlags;



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



    (void)p_arg;
    for (;;) {
        size = OSFlagNameGet(EngineStatusFlags,
                             &EngineStatusName,
                             &err);
        .
        .
    }
}
```

OSFlagNameSet()

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

Chapter	File	Called from	Code enabled by
New in V2.60	OS_FLAG.C	Task	OS_FLAG_NAME_EN

`OSFlagNameSet()` allows you to assign a name to an event flag group. 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 of the event flag group.

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

`OS_ERR_NONE`	If the call was successfull.
`OS_ERR_EVENT_TYPE`	You are not pointing to an event flag group.
`OS_ERR_PNAME_NULL`	You passed a `NULL` pointer for `pname`.
`OS_ERR_INVALID_PGRP`	You passed a `NULL` pointer for `pgrp`.
`OS_ERR_NAME_SET_ISR`	You called this function from an ISR.

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 *p_arg)
{
    INT8U  err;



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

OSFlagPend()

```
OS_FLAGS OSFlagPend(OS_FLAG_GRP *pgrp,
                    OS_FLAGS     flags,
                    INT8U        wait_type,
                    INT32U       timeout,
                    INT8U       *perr);
```

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`

`timeout` allows the task to resume execution if the desired flag(s) is(are) not received from the event flag group within the specified number of clock ticks. A timeout value of 0 indicates that the task wants to wait forever for the flag(s). The timeout value is not synchronized with the clock tick. The timeout count begins decrementing on the next clock tick, which could potentially occur immediately.

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

`OS_ERR_NONE`	No error.
`OS_ERR_PEND_ISR`	You try to call `OSFlagPend` from an ISR, which is not allowed.
`OS_ERR_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_ERR_TIMEOUT`	The flags are not available within the specified amount of time.
`OS_ERR_FLAG_WAIT_TYPE`	You don't specify a proper `wait_type` argument.

OS_ERR_PEND_ABORT	If the wait on the event flag was aborted by a call to OSFlagPendAbort() or, by calling OSFlagDel() to delete the event flag group and this task was waiting on the event flag group.

Returned Values

The flag(s) that cause the task to be ready or, 0 if either none of the flags are ready or an error occurred.

Notes/Warnings

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

IMPORTANT
The return value of OSFlagPend() is different as of V2.70. In previous versions, OSFlagPend() returned the current state of the flags and now, it returns the flag(s) that are ready, if any.

Example

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



OS_FLAG_GRP *EngineStatus;



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



    (void)p_arg;
    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_ERR_NONE:
                 /* Desired flags are available                    */
                 break;

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

OSFlagPendGetFlagsRdy()

`OS_FLAGS OSFlagPendGetFlagsRdy(void)`

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

`OSFlagPendGetFlagsRdy()` 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 *p_arg)
{
    INT8U     err;
    OS_FLAGS  value;


    (void)p_arg;
    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_ERR_NONE:
                 /* Find out who made task ready                   */
                 flags = OSFlagPendGetFlagsRdy();
                 break;

            case OS_ERR_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       *perr);
```

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`).

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

`OS_ERR_NONE`	The call is successful.
`OS_ERR_FLAG_INVALID_PGRP`	You pass a `NULL` pointer.
`OS_ERR_EVENT_TYPE`	You are not pointing to an event flag group.
`OS_ERR_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 *p_arg)
{
    INT8U  err;



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

OSFlagQuery()

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

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()`].

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

OS_ERR_NONE	The call is successful.
OS_ERR_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 *p_arg)
{
    OS_FLAGS flags;
    INT8U    err;


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

OSInit()

`void OSInit(void);`

Chapter	*File*	*Called from*	*Code enabled by*
3	`OS_CORE.C`	Startup code only	N/A

`OSInit()` initializes µC/OS-II and must be called prior to calling `OSStart()`, which actually starts multitasking.

Arguments

none

Returned Values

none

Notes/Warnings

1. `OSInit()` must be called before `OSStart()`.

Example

```
void main (void)
{
      .
      .
      OSInit();       /* Initialize µC/OS-II */
      .
      .
      OSStart();      /* Start Multitasking  */
}
```

OSIntEnter()

`void OSIntEnter(void);`

Chapter	File	Called from	Code enabled by
3	OS_CORE.C	ISR only	N/A

`OSIntEnter()` notifies µC/OS-II that an ISR is being processed, which allows µC/OS-II to keep track of interrupt nesting. `OSIntEnter()` is used in conjunction with `OSIntExit()`.

Arguments

none

Returned Values

none

Notes/Warnings

1. This function must not be called by task-level code.
2. You can increment the interrupt-nesting counter (`OSIntNesting`) directly in your ISR to avoid the overhead of the function call/return. It's safe to increment `OSIntNesting` in your ISR because interrupts are assumed to be disabled when `OSIntNesting` needs to be incremented.
3. You are allowed to nest interrupts up to 255 levels deep.

Example 1

(Intel 80x86, real mode, large model)

Use `OSIntEnter()` for backward compatibility with µC/OS.

```
     ISRx PROC   FAR
          PUSHA                        ; Save interrupted task's context
          PUSH   ES
          PUSH   DS
;
          CALL   FAR PTR _OSIntEnter   ; Notify µC/OS-II of start of ISR
          .
          .
          POP    DS                    ; Restore processor registers
          POP    ES
          POPA
          IRET                         ; Return from interrupt
     ISRx ENDP
```

Example 2

(Intel 80x86, real mode, large model)

```
    ISRx    PROC   FAR
            PUSHA                          ; Save interrupted task's context
            PUSH   ES
            PUSH   DS
;
            MOV    AX, SEG(_OSIntNesting)  ; Reload DS
            MOV    DS, AX
;
            INC    BYTE PTR _OSIntNesting  ; Notify µC/OS-II of start of ISR
            .
            .
            .
            POP    DS                      ; Restore processor registers
            POP    ES
            POPA
            IRET                           ; Return from interrupt
    ISRx    ENDP
```

OSIntExit()

`void OSIntExit(void);`

Chapter	*File*	*Called from*	*Code enabled by*
3	OS_CORE.C	ISR only	N/A

`OSIntExit()` notifies µC/OS-II that an ISR is complete, which allows µC/OS-II to keep track of interrupt nesting. `OSIntExit()` is used in conjunction with `OSIntEnter()`. When the last nested interrupt completes, `OSIntExit()` determines if a higher priority task is ready to run, in which case, the interrupt returns to the higher priority task instead of the interrupted task.

Arguments

none

Returned Value

none

Notes/Warnings

1. This function must not be called by task-level code. Also, if you decided to increment `OSIntNesting`, you still need to call `OSIntExit()`.

Example

(Intel 80x86, real mode, large model)

```
ISRx    PROC    FAR
        PUSHA                      ; Save processor registers
        PUSH    ES
        PUSH    DS
        .
        .
        CALL    FAR PTR _OSIntExit ; Notify µC/OS-II of end of ISR
        POP     DS                 ; Restore processor registers
        POP     ES
        POPA
        IRET                       ; Return to interrupted task
ISRx    ENDP
```

OSMboxAccept()

`void *OSMboxAccept(OS_EVENT *pevent);`

Chapter	*File*	*Called from*	*Code enabled by*
10	`OS_MBOX.C`	Task or ISR	`OS_MBOX_EN && OS_MBOX_ACCEPT_EN`

`OSMboxAccept()` allows you to see if a message is available from the desired mailbox. Unlike `OSMboxPend()`, `OSMboxAccept()` does not suspend the calling task if a message is not available. In other words, `OSMboxAccept()` is non-blocking. If a message is available, the message is returned to your application, and the content of the mailbox is cleared. This call is typically used by ISRs because an ISR is not allowed to wait for a message at a mailbox.

Arguments

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

Returned Value

A pointer to the message if one is available; `NULL` if the mailbox does not contain a message.

Notes/Warnings

1. Mailboxes must be created before they are used.

Example

```
OS_EVENT *CommMbox;


void Task (void *p_arg)
{
    void *pmsg;


    (void)p_arg;
    for (;;) {
        pmsg = OSMboxAccept(CommMbox); /* Check mailbox for a message */
        if (pmsg != (void *)0) {
          .                            /* Message received, process   */
          .
        } else {
          .                            /* Message not received, do .. */
          .                            /* .. something else           */
        }
       .
       .
    }
}
```

OSMboxCreate()

`OS_EVENT *OSMboxCreate(void *pmsg);`

Chapter	*File*	*Called from*	*Code enabled by*
10	`OS_MBOX.C`	Task or startup code	`OS_MBOX_EN`

`OSMboxCreate()` creates and initializes a mailbox. A mailbox allows tasks or ISRs to send a pointer-sized variable (message) to one or more tasks.

Arguments

`pmsg` is used to initialize the contents of the mailbox. The mailbox is empty when `pmsg` is a `NULL` pointer. The mailbox initially contains a message when `pmsg` is non-`NULL`.

Returned Value

A pointer to the event control block allocated to the mailbox. If no event control block is available, `OSMboxCreate()` returns a `NULL` pointer.

Notes/Warnings

1. Mailboxes must be created before they are used.

Example

```
OS_EVENT *CommMbox;



void main (void)
{
        .
        .
        OSInit();                              /* Initialize µC/OS-II  */
        .
        .
        CommMbox = OSMboxCreate((void *)0);    /* Create COMM mailbox  */
        OSStart();                             /* Start Multitasking   */
}
```

OSMboxDel()

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

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.

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

OS_ERR_NONE	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 pevent is a NULL pointer.

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.
4. All tasks that were waiting for the mailbox will be readied and returned an OS_ERR_PEND_ABORT error code if OSMboxDel() was called with OS_DEL_ALWAYS option.

Example

```
OS_EVENT *DispMbox;

void Task (void *p_arg)
{
    INT8U  err;

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

OSMboxPend()

```
void *OSMboxPend(OS_EVENT *pevent,
                 INT32U    timeout,
                 INT8U    *perr);
```

Chapter	*File*	*Called from*	*Code enabled by*
10	`OS_MBOX.C`	Task only	`OS_MBOX_EN`

`OSMboxPend()` is used when a task expects to receive a message. The message is sent to the task either by an ISR or by another task. The message received is a pointer-sized variable, and its use is application specific. If a message is present in the mailbox when `OSMboxPend()` is called, the message is retrieved, the mailbox is emptied, and the retrieved message is returned to the caller. If no message is present in the mailbox, `OSMboxPend()` suspends the current task until either a message is received or a user-specified timeout expires. If a message is sent to the mailbox and multiple tasks are waiting for the message, µC/OS-II resumes the highest priority task waiting to run. 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 mailbox from which the message is received. This pointer is returned to your application when the mailbox is created [see `OSMboxCreate()`].

`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 timeout value is not synchronized with the clock tick. The timeout count begins decrementing on the next clock tick, which could potentially occur immediately.

`perr` is a pointer to a variable that holds an error code. `OSMboxPend()` sets `*perr` to one of the following:

- `OS_ERR_NONE` if a message is received.
- `OS_ERR_TIMEOUT` if a message is not received within the specified timeout period.
- `OS_ERR_PEND_ABORT` indicates that the pend was aborted by another task or ISR by calling `OSMboxPendAbort()`.
- `OS_ERR_EVENT_TYPE` if `pevent` is not pointing to a mailbox.
- `OS_ERR_PEND_LOCKED` if you called this function when the scheduler is locked.
- `OS_ERR_PEND_ISR` if you call this function from an ISR and µC/OS-II suspends it. In general, you should not call `OSMboxPend()` from an ISR, but µC/OS-II checks for this situation anyway.
- `OS_ERR_PEVENT_NULL` if `pevent` is a `NULL` pointer.
- `OS_ERR_PEND_ABORT` If the wait on the mailbox was aborted by a call to `OSMboxPendAbort()` or, by calling `OSMboxDel()` to delete the mailbox and this task was waiting on the mailbox.

Returned Value

`OSMboxPend()` returns the message sent by either a task or an ISR, and `*perr` is set to `OS_ERR_NONE`. If a message is not received within the specified timeout period, the returned message is a `NULL` pointer, and `*perr` is set to `OS_ERR_TIMEOUT`.

Notes/Warnings

1. Mailboxes must be created before they are used.
2. You should not call `OSMboxPend()` from an ISR.

Example

```
OS_EVENT *CommMbox;



void CommTask(void *p_arg)
{
      INT8U  err;
      void   *pmsg;



      (void)p_arg;
      for (;;) {
         .
         .
         pmsg = OSMboxPend(CommMbox, 10, &err);
         if (err == OS_ERR_NONE) {
             .
             .  /* Code for received message                    */
             .
         } else {
             .
             .  /* Code for message not received within timeout */
             .
         }
         .
         .
      }
}
```

OSMboxPendAbort()

```
INT8U OSMboxPendAbort(OS_EVENT *pevent,
                      INT8U     opt,
                      INT8U    *perr);
```

New Function	*File*	*Called from*	*Code enabled by*
V2.84	OS_MBOX.C	Task only	OS_MBOX_EN && OS_MBOX_PEND_ABORT_EN

OSMboxPendAbort() aborts & readies any tasks currently waiting on a mailbox. This function should be used to fault-abort the wait on the mailbox, rather than to normally signal the mailbox via OSMboxPost() or OSMboxPostOpt().

Arguments

pevent is a pointer to the mailbox for which pend(s) need to be aborted. This pointer is returned to your application when the mailbox is created [see OSMboxCreate()].

opt determines what type of abort is performed.

OS_PEND_OPT_NONE Aborts the pend of only the highest priority task waiting on the mailbox.

OS_PEND_OPT_BROADCAST Aborts the pend of all the tasks waiting on the mailbox.

perr is a pointer to a variable that holds an error code. OSMboxPendAbort() sets *perr to one of the following:

OS_ERR_NONE if no tasks were waiting on the mailbox. In this case, the return value is also 0.

OS_ERR_PEND_ABORT at least one task waiting on the mailbox was readied and informed of the aborted wait. Check the return value for the number of tasks whose wait on the mailbox was aborted.

OS_ERR_EVENT_TYPE if pevent is not pointing to a mailbox.

OS_ERR_PEVENT_NULL if pevent is a NULL pointer.

Returned Value

OSMboxPendAbort() returns the number of tasks made ready to run by this function. Zero indicates that no tasks were pending on the mailbox and thus this function had no effect.

Notes/Warnings

1. Mailboxes must be created before they are used.

Example

```
OS_EVENT *CommMbox;



void CommTask(void *p_arg)
{
      INT8U  err;
      INT8U  nbr_tasks;



      (void)p_arg;
      for (;;) {
         .
         .
         nbr_tasks = OSMboxPendAbort(CommMbox, OS_PEND_OPT_BROADCAST, &err);
         if (err == OS_ERR_NONE) {
             .
             .  /* No tasks were waiting on the mailbox                 */
             .
         } else {
             .
             .  /* All pends of tasks waiting on mailbox were aborted … */
             .  /* … ‘nbr_tasks’ indicates how many were made ready.    */
             .
         }
         .
         .
      }
}
```

OSMboxPost()

```
INT8U OSMboxPost(OS_EVENT *pevent,
                 void     *pmsg);
```

Chapter	*File*	*Called from*	*Code enabled by*
10	`OS_MBOX.C`	Task or ISR	`OS_MBOX_EN && OS_MBOX_POST_EN`

`OSMboxPost()` sends 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. `OSMboxPost()` 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, the highest priority task waiting 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. In other words, a context switch occurs.

Arguments

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

`pmsg` is the actual message sent to the task. `pmsg` 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.

Returned Value

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

`OS_ERR_NONE`	if the message is deposited in the mailbox.
`OS_ERR_MBOX_FULL`	if the mailbox already contains a message.
`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 because this pointer indicates that the mailbox is empty.

Example

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


void CommTaskRx (void *p_arg)
{
      INT8U  err;


      (void)p_arg;
      for (;;) {
         .
         .
          err = OSMboxPost(CommMbox, (void *)&CommRxBuf[0]);
         .
         .
      }
}
```

OSMboxPostOpt()

```
INT8U OSMboxPostOpt(OS_EVENT *pevent,
                    void     *pmsg,
                    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()].

pmsg is the actual message sent to the task(s). pmsg 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).

When set to OS_POST_OPT_NO_SCHED, the scheduler will not be called to see if a higher priority task has been made ready to run.

Note that options are additive and thus, you can specify:

```
OS_POST_OPT_BROADCAST | OS_POST_OPT_NO_SCHED
```

Returned Value

Returns one of the following error codes:

OS_ERR_NONE if the call is successful and the message has been sent.

OS_ERR_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 *p_arg)
{
    INT8U  err;


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

OSMboxQuery()

```
INT8U OSMboxQuery(OS_EVENT     *pevent,
                  OS_MBOX_DATA *p_mbox_data);
```

Chapter	*File*	*Called from*	*Code enabled by*
10	`OS_MBOX.C`	Task or ISR	`OS_MBOX_EN && OS_MBOX_QUERY_EN`

`OSMboxQuery()` obtains information about a message mailbox. Your application must allocate an `OS_MBOX_DATA` data structure, which is used to receive data from the event control block of the message mailbox. `OSMboxQuery()` allows you to determine whether any tasks are waiting for a message at the mailbox and how many tasks are waiting (by counting the number of 1s in the `.OSEventTbl[]` field). You can also examine the current contents of the mailbox. 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 mailbox. This pointer is returned to your application when the mailbox is created [see `OSMboxCreate()`].

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

```
void  *OSMsg;              /* Copy of the message stored in the mailbox */
#if OS_LOWEST_PRIO <= 63
INT8U  OSEventTbl[OS_EVENT_TBL_SIZE]; /* Copy of the mailbox wait list  */
INT8U  OSEventGrp;
#else
INT16U OSEventTbl[OS_EVENT_TBL_SIZE]; /* Copy of the mailbox wait list  */
INT16U OSEventGrp;
#endif
```

Returned Value

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

`OS_ERR_NONE`	if the call is successful.
`OS_ERR_PEVENT_NULL`	if `pevent` is a `NULL` pointer.
`OS_ERR_EVENT_TYPE`	if you don't pass a pointer to a message mailbox.
`OS_ERR_PNAME_NULL`	You passed a `NULL` pointer for `p_mbox_data`.

Notes/Warnings

1. Message mailboxes must be created before they are used.

Example

```
OS_EVENT *CommMbox;


void Task (void *p_arg)
{
      OS_MBOXDATA mbox_data;
      INT8U       err;


      (void)p_arg;
      for (;;) {
         .
         .
         err = OSMboxQuery(CommMbox, &mbox_data);
         if (err == OS_ERR_NONE) {
           .   /* Mailbox contains a message if .. */
               /* .. mbox_data.OSMsg is not NULL   */
         }
         .
         .
      }
}
```

OSMemCreate()

```
OS_MEM *OSMemCreate(void   *addr,
                    INT32U  nblks,
                    INT32U  blksize,
                    INT8U  *perr);
```

Chapter	*File*	*Called from*	*Code enabled by*
12	`OS_MEM.C`	Task or startup code	`OS_MEM_EN`

`OSMemCreate()` creates and initializes a memory partition. A memory partition contains a user-specified number of fixed-size memory blocks. Your application can obtain one of these memory blocks and, when done, release the block back to the partition.

Arguments

`addr` is the address of the start of a memory area that is used to create fixed-size memory blocks. Memory partitions can be created either using static arrays or `malloc()` during startup. Note that the partition MUST align on a pointer boundary. Thus, if a pointer is 16 bits wide then the partition must start on a memory location with an address that ends with 0, 2, 4, 6, 8, etc. If a pointer is 32 bits wide then the partition must start on a memory location with and address that ends with 0, 4, 8 of C.

`nblks` contains the number of memory blocks available from the specified partition. You must specify at least two memory blocks per partition.

`blksize` specifies the size (in bytes) of each memory block within a partition. A memory block must be large enough to hold at least a pointer. Also, the size of a memory block must be a multiple of the size of a pointer. In other words, if a pointer is 32 bits wide then the block size must be 4, 8, 12, 16, 20, etc. bytes (i.e. a multiple of 4 bytes).

`perr` is a pointer to a variable that holds an error code. `OSMemCreate()` sets `*perr` to:

`OS_ERR_NONE` if the memory partition is created successfully

`OS_ERR_MEM_INVALID_ADDR` if you are specifying an invalid address (i.e., `addr` is a `NULL` pointer) or your partition is not properly aligned.

`OS_ERR_MEM_INVALID_PART` if a free memory partition is not available

`OS_ERR_MEM_INVALID_BLKS` if you don't specify at least two memory blocks per partition

`OS_ERR_MEM_INVALID_SIZE` if you don't specify a block size that can contain at least a pointer variable and if it's not a multiple of a pointer size variable.

Returned Value

`OSMemCreate()` returns a pointer to the created memory-partition control block if one is available. If no memory-partition control block is available, `OSMemCreate()` returns a `NULL` pointer.

Notes/Warnings

1. Memory partitions must be created before they are used.

Example

```
OS_MEM  *CommMem;
INT32U   CommBuf[16][32];


void main (void)
{
      INT8U err;


      OSInit();                        /* Initialize µC/OS-II           */
      .
      .
      CommMem = OSMemCreate(&CommBuf[0][0], 16, 32 * sizeof(INT32U), &err);
      .
      .
      OSStart();                       /* Start Multitasking            */
}
```

OSMemGet()

```
void *OSMemGet(OS_MEM *pmem,
               INT8U  *perr);
```

Chapter	*File*	*Called from*	*Code enabled by*
12	OS_MEM.C	Task or ISR	OS_MEM_EN

OSMemGet obtains a memory block from a memory partition. It is assumed that your application knows the size of each memory block obtained. Also, your application must return the memory block [using OSMemPut()] when it no longer needs it. You can call OSMemGet() more than once until all memory blocks are allocated.

Arguments

pmem is a pointer to the memory-partition control block that is returned to your application from the OSMemCreate() call.

perr is a pointer to a variable that holds an error code. OSMemGet() sets *perr to one of the following:

- OS_ERR_NONE if a memory block is available and returned to your application.
- OS_ERR_MEM_NO_FREE_BLKS if the memory partition doesn't contain any more memory blocks to allocate.
- OS_ERR_MEM_INVALID_PMEM if pmem is a NULL pointer.

Returned Value

OSMemGet() returns a pointer to the allocated memory block if one is available. If no memory block is available from the memory partition, OSMemGet() returns a NULL pointer.

Notes/Warnings

1. Memory partitions must be created before they are used.

Example

```
OS_MEM *CommMem;


void Task (void *p_arg)
{
      INT8U *pmsg;


      (void)p_arg;
      for (;;) {
         pmsg = OSMemGet(CommMem, &err);
         if (pmsg != (INT8U *)0) {
            .                      /* Memory block allocated, use it. */
            .
         }
         .
         .
      }
}
```

OSMemNameGet()

```
INT8U OSMemNameGet(OS_MEM   *pmem,
                   INT8U   **pname,
                   INT8U    *perr);
```

Chapter	File	Called from	Code enabled by
New in V2.60	OS_MEM.C	Task	OS_MEM_NAME_EN

`OSMemNameGet()` allows you to obtain the name that you assigned to a memory partition. 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 a pointer to an ASCII string that contains the name of the memory partition.

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

`OS_ERR_NONE`	If the call was successfull.
`OS_ERR_INVALID_PMEM`	You passed a `NULL` pointer for `pmem`.
`OS_ERR_PNAME_NULL`	You passed a `NULL` pointer for `pname`.
`OS_ERR_NAME_GET_ISR`	`You called this function from an ISR.`

Returned Values

The size of the ASCII string 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;
INT8U   *CommMemName;

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



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

OSMemNameSet()

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

Chapter	File	Called from	Code enabled by
New in V2.60	OS_MEM.C	Task	OS_MEM_NAME_EN

OSMemNameSet() allows you to assign a name to a memory partition. 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 memory partition.

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

OS_ERR_NONE	If the call was successfull.
OS_ERR_MEM_INVALID_PMEM	You passed a NULL pointer for pmem.
OS_ERR_PNAME_NULL	You passed a NULL pointer for pname.
OS_ERR_MEM_NAME_TOO_LONG	If the name is not able to fit in the specified storage.
OS_ERR_NAME_SET_ISR	You called this function from an ISR.

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 *p_arg)
{
    INT8U  err;



    (void)p_arg;
    for (;;) {
        OSMemNameSet(CommMem, "Comm. Buffer", &err);
        .
        .
    }
}
```

OSMemPut()

```
INT8U OSMemPut(OS_MEM *pmem,
               void   *pblk);
```

Chapter	*File*	*Called from*	*Code enabled by*
12	OS_MEM.C	Task or ISR	OS_MEM_EN

`OSMemPut()` returns a memory block to a memory partition. It is assumed that you return the memory block to the appropriate memory partition.

Arguments

`pmem` is a pointer to the memory-partition control block that is returned to your application from the `OSMemCreate()` call.

`pblk` is a pointer to the memory block to be returned to the memory partition.

Returned Value

`OSMemPut()` returns one of the following error codes:

`OS_ERR_NONE` if the memory block was returned to the memory partition.

`OS_ERR_MEM_FULL` if the memory partition can not accept more memory blocks. This code is surely an indication that something is wrong because you are returning more memory blocks than you obtained using `OSMemGet()`.

`OS_ERR_MEM_INVALID_PMEM` if `pmem` is a `NULL` pointer.

`OS_ERR_MEM_INVALID_PBLK` if `pblk` is a `NULL` pointer.

Notes/Warnings

1. Memory partitions must be created before they are used.
2. You must return a memory block to the proper memory partition.

Example

```
OS_MEM *CommMem;
INT8U  *CommMsg;



void Task (void *p_arg)
{
      INT8U err;



      (void)p_arg;
      for (;;) {
         err = OSMemPut(CommMem, (void *)CommMsg);
         if (err == OS_ERR_NONE) {
            .                        /* Memory block released        */
            .
         }
         .
         .
      }
}
```

OSMemQuery()

```
INT8U OSMemQuery(OS_MEM      *pmem,
                 OS_MEM_DATA *p_mem_data);
```

Chapter	File	Called from	Code enabled by
12	OS_MEM.C	Task or ISR	OS_MEM_EN && OS_MEM_QUERY_EN

OSMemQuery() obtains information about a memory partition. Basically, this function returns the same information found in the OS_MEM data structure but in a new data structure called OS_MEM_DATA. OS_MEM_DATA also contains an additional field that indicates the number of memory blocks in use.

Arguments

pmem — is a pointer to the memory-partition control block that is returned to your application from the OSMemCreate() call.

p_mem_data — is a pointer to a data structure of type OS_MEM_DATA, which contains the following fields

```
void   *OSAddr;     /* Points to beginning address of the memory partition   */
void   *OSFreeList; /* Points to beginning of the free list of memory blocks */
INT32U  OSBlkSize;  /* Size (in bytes) of each memory block                  */
INT32U  OSNBlks;    /* Total number of blocks in the partition               */
INT32U  OSNFree;    /* Number of memory blocks free                          */
INT32U  OSNUsed;    /* Number of memory blocks used                          */
```

Returned Value

OSMemQuery() returns one of the following error codes:

OS_ERR_NONE	if *p_mem_data was filled successfully.
OS_ERR_MEM_INVALID_PMEM	if pmem is a NULL pointer.
OS_ERR_MEM_INVALID_PDATA	if pdata is a NULL pointer.

Notes/Warnings

1. Memory partitions must be created before they are used.

Example

```
OS_MEM      *CommMem;



void Task (void *p_arg)
{
      INT8U         err;
      OS_MEM_DATA  mem_data;



      (void)p_arg;
      for (;;) {
         .
         .
         err = OSMemQuery(CommMem, &mem_data);
         .
         .
      }
}
```

OSMutexAccept()

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

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()`].

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

`OS_ERR_NONE`	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.
`OS_ERR_PCP_LOWER`	If the priority of the task that owns the Mutex is HIGHER (i.e. a lower number) than the PCP. This error indicates that you did not set the PCP higher (lower number) than ALL the tasks that compete for the Mutex. Unfortunately, this is something that could not be detected when the Mutex is created because we don't know what tasks will be using the Mutex.

Returned Value

If the mutex is available, `OSMutexAccept()` returns `OS_TRUE`. If the mutex is owned by another task, `OSMutexAccept()` returns `OS_FALSE`.

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 *p_arg)
{
    INT8U    err;
    BOOLEAN  test;



    (void)p_arg;
    for (;;) {
        test = OSMutexAccept(DispMutex, &err);
        if (test == OS_TRUE) {
            .                          /* Resource available, process */
            .
        } else {
            .                          /* Resource NOT available      */
            .
        }
        .
        .
    }
}
```

OSMutexCreate()

```
OS_EVENT *OSMutexCreate(INT8U  prio,
                        INT8U *perr);
```

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 ceiling priority (PCP) 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 PCP until the resource is released.

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

`OS_ERR_NONE`	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_ERR_PRIO_EXIST`	if a task at the specified priority ceiling priority already exists.
`OS_ERR_PEVENT_NULL`	if no more `OS_EVENT` structures are available.
`OS_ERR_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    *perr);
```

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.

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

- OS_ERR_NONE 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 pevent is a NULL pointer.

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 *p_arg)
{
    INT8U  err;



    (void)p_arg;
    while (1) {
        .
        .
        DispMutex = OSMutexDel(DispMutex, OS_DEL_ALWAYS, &err);
        if (DispMutex == (OS_EVENT *)0) {
            /* Mutex has been deleted */
        }
        .
        .
    }
}
```

OSMutexPend()

```
void OSMutexPend(OS_EVENT *pevent,
                 INT32U    timeout,
                 INT8U    *perr);
```

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 perr is set to OS_ERR_NONE, 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, µC/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 PCP, 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 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.

perr — is a pointer to a variable that is used to hold an error code. OSMutexPend() sets *perr to one of the following:

OS_ERR_NONE	if the call is successful and the mutex is available.
OS_ERR_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_LOCKED	if you called this function when the scheduler is locked
OS_ERR_PEND_ABORT	if OSMutexPend() was aborted by another task
OS_ERR_PEND_ISR	if you attempt to acquire the mutex from an ISR.
OS_ERR_PCP_LOWER	If the priority of the task that owns the Mutex is HIGHER (i.e. a lower number) than the PCP. This error indicates that you did not set the PCP higher (lower number) than ALL the tasks that compete for the Mutex. Unfortunately, this is something that could not be detected when the Mutex is created because we don't know what tasks will be using the Mutex.

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 *p_arg)
{
    INT8U  err;



    (void)p_arg;
    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_ERR_NONE` 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.

`OS_ERR_PCP_LOWER` If the priority of the new task that owns the Mutex is HIGHER (i.e. a lower number) than the PCP. This error indicates that you did not set the PCP higher (lower number) than ALL the tasks that compete for the Mutex. Unfortunately, this is something that could not be detected when the Mutex is created because we don't know what tasks will be using 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 *p_arg)
{
    INT8U  err;



    (void)p_arg;
    for (;;) {
        .
        .
        err = OSMutexPost(DispMutex);
        switch (err) {
           case OS_ERR_NONE: /* 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 *p_mutex_data);
```

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 PCP, and determine whether the mutex is available (OS_TRUE) or not (OS_FALSE). 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()].

p_mutex_data — is a pointer to a data structure of type OS_MUTEX_DATA, which contains the following fields

```
INT8U   OSMutexPCP;    /* The PCP of the mutex                     */
INT8U   OSOwnerPrio;   /* The priority of the mutex owner          */
BOOLEAN OSValue;       /* The current mutex value                  */
                       /* OS_TRUE  means available                 */
                       /* OS_FALSE means unavailable               */
#if OS_LOWEST_PRIO <= 63
INT8U   OSEventGrp;    /* Copy of the mutex wait list              */
INT8U   OSEventTbl[OS_EVENT_TBL_SIZE];
#else
INT16U  OSEventGrp;    /* Copy of the mutex wait list              */
INT16U  OSEventTbl[OS_EVENT_TBL_SIZE];
#endif
```

Returned Value

OSMutexQuery() returns one of these error codes:

OS_ERR_NONE	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_PDATA_NULL	if p_mutex_data 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 *p_arg)
{
    OS_MUTEX_DATA mutex_data;
    INT8U         err;
    INT8U         highest;     /* Highest priority task waiting on mutex */
    INT8U         x;
    INT8U         y;



    (void)p_arg;
    for (;;) {
        .
        .
        err = OSMutexQuery(DispMutex, &mutex_data);
        if (err == OS_ERR_NONE) {
            /* Examine Mutex data */
                .
                .
            }
        }
        .
        .
    }
}
```

OSQAccept()

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

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()`].

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

- `OS_ERR_NONE` if the call is successful and a message is available in the desired message queue.
- `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_ERR_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 `perr` argument has been added to the call.

Example

```
OS_EVENT *CommQ;


void Task (void *p_arg)
{
      void *pmsg;


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

OSQCreate()

```
OS_EVENT *OSQCreate(void  **start,
                    INT8U   size);
```

Chapter	File	Called from	Code enabled by
11	OS_Q.C	Task or startup code	OS_Q_EN

`OSQCreate()` creates a message queue. A message queue allows tasks or ISRs to send pointer-sized variables (messages) to one or more tasks. The meaning of the messages sent are application specific.

Arguments

`start` is the base address of the message storage area. A message storage area is declared as an array of pointers to voids.

`size` is the size (in number of entries) of the message storage area.

Returned Value

`OSQCreate()` returns a pointer to the event control block allocated to the queue. If no event control block or no queue control block is available, `OSQCreate()` returns a `NULL` pointer.

Notes/Warnings

1. Queues must be created before they are used.

Example

```
OS_EVENT *CommQ;
void     *CommMsg[10];



void main (void)
{
       OSInit();                                 /* Initialize µC/OS-II   *
       .
       .
       CommQ = OSQCreate(&CommMsg[0], 10);       /* Create COMM Q         *
       .
       .
       OSStart();                                /* Start Multitasking    *
}
```

OSQDel()

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

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.

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

`OS_ERR_NONE`	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 `pevent` is a `NULL` pointer.

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.
3. All tasks that were waiting for the queue will be readied and returned an `OS_ERR_PEND_ABORT` error code if `OSQDel()` was called with `OS_DEL_ALWAYS` option.

Example

```
OS_EVENT *DispQ;



void Task (void *p_arg)
{
    INT8U  err;



    (void)p_arg;
    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_ERR_NONE`	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,
              INT32U    timeout,
              INT8U    *perr);
```

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 timeout value is not synchronized with the clock tick. The timeout count starts decrementing on the next clock tick, which could potentially occur immediately.

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

- `OS_ERR_NONE` if a message is received.
- `OS_ERR_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_ABORT` if `OSQPend()` was aborted by another task who called `OSQPendAbort()`.
- `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. µC/OS-II checks for this situation anyway.
- `OS_ERR_PEND_LOCKED` if you called this function when the scheduler is locked.
- `OS_ERR_PEND_ABORT` If the wait on the queue was aborted by a call to `OSQPendAbort()` or, by calling `OSQDel()` to delete the message queue and this task was waiting on the message queue.

Returned Value

`OSQPend()` returns a message sent by either a task or an ISR, and `*perr` is set to `OS_ERR_NONE`. If a timeout occurs, `OSQPend()` returns a `NULL` pointer and sets `*perr` to `OS_ERR_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 *p_arg)
{
      INT8U  err;
      void  *pmsg;



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

OSQPendAbort()

```
INT8U  OSQPendAbort(OS_EVENT *pevent,
                    INT8U     opt,
                    INT8U    *perr);
```

New Function	*File*	*Called from*	*Code enabled by*
V2.84	`OS_Q.C`	Task only	`OS_Q_EN` && `OS_Q_PEND_ABORT_EN`

`OSQPendAbort()` aborts & readies any tasks currently waiting on a queue. This function should be used to fault-abort the wait on the queue, rather than to normally signal the queue via `OSQPost()`, `OSQPostFront()` or `OSQPostOpt()`.

Arguments

`pevent` is a pointer to the queue for which pend(s) need to be aborted. This pointer is returned to your application when the queue is created [see `OSQCreate()`].

`opt` determines what type of abort is performed.

- `OS_PEND_OPT_NONE` Aborts the pend of only the highest priority task waiting on the queue.
- `OS_PEND_OPT_BROADCAST` Aborts the pend of all the tasks waiting on the queue.

`perr` is a pointer to a variable that holds an error code. `OSQPendAbort()` sets `*perr` to one of the following:

- `OS_ERR_NONE` if no tasks were waiting on the queue. In this case, the return value is also 0.
- `OS_ERR_PEND_ABORT` at least one task waiting on the queue was readied and informed of the aborted wait. Check the return value for the number of tasks whose wait on the queue was aborted.
- `OS_ERR_EVENT_TYPE` if `pevent` is not pointing to a queue.
- `OS_ERR_PEVENT_NULL` if `pevent` is a `NULL` pointer.

Returned Value

`OSQPendAbort()` returns the number of tasks made ready to run by this function. Zero indicates that no tasks were pending on the queue and thus this function had no effect.

Notes/Warnings

1. Queues must be created before they are used.

Example

```
OS_EVENT *CommQ;



void CommTask(void *p_arg)
{
      INT8U  err;
      INT8U  nbr_tasks;



      (void)p_arg;
      for (;;) {
         .
         .
         nbr_tasks = OSQPendAbort(CommQ, OS_PEND_OPT_BROADCAST, &err);
         if (err == OS_ERR_NONE) {
             .
             .  /* No tasks were waiting on the queue                 */
             .
         } else {
             .
             .  /* All pends of tasks waiting on queue were aborted … */
             .  /* … ‘nbr_tasks’ indicates how many were made ready.  */
             .
         }
         .
         .
      }
}
```

OSQPost()

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

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()`].

pmsg is the actual message sent to the task. `pmsg` 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_ERR_NONE`	if the message is deposited in the queue.
`OS_ERR_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 `perr` variable accordingly.

Example

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



void CommTaskRx (void *p_arg)
{
      INT8U  err;



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


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

OSQPostFront()

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

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()`].

`pmsg` is the actual message sent to the task. `pmsg` 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_ERR_NONE`	if the message is deposited in the queue.
`OS_ERR_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 `perr` variable accordingly.

Example

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



void CommTaskRx (void *p_arg)
{
      INT8U  err;



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


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

OSQPostOpt()

```
INT8U OSQPostOpt(OS_EVENT *pevent,
                 void     *pmsg,
                 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()`].

`pmsg` is the actual message sent to the task(s). `pmsg` is a pointer-sized variable, and what `pmsg` 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()`].

`OS_POST_OPT_NO_SCHED` Do not call the scheduler after the post.

Below is a list of some of 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 `pmsg` to **all** waiting tasks

`OS_POST_OPT_FRONT + OS_POST_OPT_BROADCAST`

is identical to `OSQPostFront()` except that broadcasts `pmsg` to **all** waiting tasks.

`OS_POST_OPT_FRONT + OS_POST_OPT_BROADCAST + OS_POST_OPT_NO_SCHED`

is identical to `OSQPostFront()` except that broadcasts `pmsg` to **all** waiting tasks and the scheduler will not be called

Returned Value

`OSQPostOpt` returns one of the following error codes:

`OS_ERR_NONE`	if the call is successful and the message has been sent.
`OS_ERR_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 *p_arg)
{
    INT8U  err;



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

OSQQuery()

```
INT8U OSQQuery(OS_EVENT  *pevent,
               OS_Q_DATA *p_q_data);
```

Chapter	*File*	*Called from*	*Code enabled by*
11	`OS_Q.C`	Task or ISR	`OS_Q_EN && OS_QUERY_EN`

`OSQQuery()` obtains information about a message queue. Your application must allocate an `OS_Q_DATA` data structure used to receive data from the event control block of the message queue. `OSQQuery()` allows you to determine whether any tasks are waiting for messages at the queue, how many tasks are waiting (by counting the number of 1s in the `.OSEventTbl[]` field), how many messages are in the queue, and what the message queue size is. `OSQQuery()` also obtains the next message that is returned if the queue is not empty. 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 message queue. This pointer is returned to your application when the queue is created [see `OSQCreate()`].

`p_q_data` is a pointer to a data structure of type `OS_Q_DATA`, which contains the following fields

```
void  *OSMsg;                  /* Next message if one available         */
INT16U OSNMsgs;                /* Number of messages in the queue       */
INT16U OSQSize;                /* Size of the message queue             */
#if OS_LOWEST_PRIO <= 63
INT8U  OSEventTbl[OS_EVENT_TBL_SIZE];      /* Message queue wait list   */
INT8U  OSEventGrp;
#else
INT16U OSEventTbl[OS_EVENT_TBL_SIZE];      /* Message queue wait list   */
INT16U  OSEventGrp;
#endif
```

Returned Value

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

`OS_ERR_NONE`	if the call is successful.
`OS_ERR_EVENT_TYPE`	if you don't pass a pointer to a message queue.
`OS_ERR_PEVENT_NULL`	if `pevent` is a `NULL` pointer.
`OS_ERR_PDATA_NULL`	if `p_q_data` is a `NULL` pointer.

Notes/Warnings

1. Message queues must be created before they are used.

Example

```
OS_EVENT *CommQ;



void Task (void *p_arg)
{
      OS_Q_DATA qdata;
      INT8U     err;



      (void)p_arg;
      for (;;) {
         .
         .
         err = OSQQuery(CommQ, &qdata);
         if (err == OS_ERR_NONE) {
           .   /* 'qdata' can be examined! */
         }
         .
         .
      }
}
```

`OSSafetyCriticalStart()`

`void OSSafetyCriticalStart(void);`

Chapter	*File*	*Called from*	*Code enabled by*
-	`OS_CORE.C`	Task or startup code	`OS_SAFETY_CRITICAL _IEC61508`

`OSSafetyCriticalStart()` indicates that all initialization has been completed and that kernel objects are no longer allowed to be created.

Arguments

none

Returned Value

none

Notes/Warnings

1. After calling `OSSafetyCriticalStart()`, your application must not make system calls that creates kernel objects, otherwise an exception is thrown.

Example

```
OS_STK  Task1Stk[1024];

void main (void)
{
      INT8U err;


      .
      OSInit();                 /* Initialize µC/OS-II            */
      .
      OSTaskCreate(Task1,
                   (void *)0,
                   &Task1Stk[1023],
                   25);
      .
      OSStart();                /* Start Multitasking             */
}



void Task1 (void *p_arg)
{
      (void)p_arg;              /* Prevent compiler warning       */

      OSTaskCreate(_);          /* Create the other tasks         */
      OSSemCreate(_);           /* Create semaphores              */
                                /* Create other kernel objects    */
      OSQCreate(_);             /* Create queues                  */

      OSSafetyCriticalStart()   /* Prevent kernel objects from... */
                                /* ... being created.             */

      for (;;) {
         .                      /* Task code                      */
         .
      }
}
```

OSSchedLock()

`void OSSchedLock(void);`

Chapter	*File*	*Called from*	*Code enabled by*
3	`OS_CORE.C`	Task or ISR	`OS_SCHED_LOCK_EN`

`OSSchedLock()` prevents task rescheduling until its counterpart, `OSSchedUnlock()`, is called. The task that calls `OSSchedLock()` keeps control of the CPU even though other higher priority tasks are ready to run. However, interrupts are still recognized and serviced (assuming interrupts are enabled). `OSSchedLock()` and `OSSchedUnlock()` must be used in pairs. µC/OS-II allows `OSSchedLock()` to be nested up to 255 levels deep. Scheduling is enabled when an equal number of `OSSchedUnlock()` calls have been made.

Arguments

none

Returned Value

none

Notes/Warnings

1. After calling `OSSchedLock()`, your application must not make system calls that suspend execution of the current task; that is, your application cannot call `OSTimeDly()`, `OSTimeDlyHMSM()`, `OSFlagPend()`, `OSSemPend()`, `OSMutexPend()`, `OSMboxPend()`, or `OSQPend()`. Because the scheduler is locked out, no other task is allowed to run, and your system will lock up.

Example

```
void TaskX (void *p_arg)
{
      (void)p_arg;
      for (;;) {
         .
         OSSchedLock();         /* Prevent other tasks to run         */
         .
         .                      /* Code protected from context switch */
         .
         OSSchedUnlock();       /* Enable other tasks to run          */
         .
      }
}
```

OSSchedUnlock()

`void OSSchedUnlock(void);`

Chapter	*File*	*Called from*	*Code enabled by*
3	OS_CORE.C	Task or ISR	OS_SCHED_LOCK_EN

`OSSchedUnlock()` re-enables task scheduling whenever it is paired with `OSSchedLock()`.

Arguments

none

Returned Value

none

Notes/Warnings

1. After calling `OSSchedLock()`, your application must not make system calls that suspend execution of the current task; that is, your application cannot call `OSTimeDly()`, `OSTimeDlyHMSM()`, `OSFlagPend()`, `OSSemPend()`, `OSMutexPend()`, `OSMboxPend()`, or `OSQPend()`. Because the scheduler is locked out, no other task is allowed to run, and your system will lock up.

Example

```
void TaskX (void *p_arg)
{
      (void)p_arg;
      for (;;) {
         .
         OSSchedLock();      /* Prevent other tasks to run          */
         .
         .                   /* Code protected from context switch */
         .
         OSSchedUnlock();    /* Enable other tasks to run           */
         .
      }
}
```

OSSemAccept()

`INT16U OSSemAccept(OS_EVENT *pevent);`

Chapter	*File*	*Called from*	*Code enabled by*
7	`OS_SEM.C`	Task or ISR	`OS_SEM_EN && OS_SEM_ACCEPT_EN`

`OSSemAccept()` checks to see if a resource is available or an event has occurred. Unlike `OSSemPend()`, `OSSemAccept()` does not suspend the calling task if the resource is not available. In other words, `OSSemAccept()` is non-blocking. Use `OSSemAccept()` from an ISR to obtain the semaphore.

Arguments

`pevent` is a pointer to the semaphore that guards the resource. This pointer is returned to your application when the semaphore is created [see `OSSemCreate()`].

Returned Value

When `OSSemAccept()` is called and the semaphore value is greater than 0, the semaphore value is decremented, and the value of the semaphore before the decrement is returned to your application. If the semaphore value is 0 when `OSSemAccept()` is called, the resource is not available, and 0 is returned to your application.

Notes/Warnings

1. Semaphores must be created before they are used.

Example

```
OS_EVENT *DispSem;


void Task (void *p_arg)
{
      INT16U value;



      (void)p_arg;
      for (;;) {
         value = OSSemAccept(DispSem);    /* Check resource availability */
         if (value > 0) {
            .                             /* Resource available, process */
            .
         }
         .
         .
      }
}
```

OSSemCreate()

`OS_EVENT *OSSemCreate(INT16U value);`

Chapter	*File*	*Called from*	*Code enabled by*
7	`OS_SEM.C`	Task or startup code	`OS_SEM_EN`

`OSSemCreate()` creates and initializes a semaphore. A semaphore

- allows a task to synchronize with either an ISR or a task (you initialize the semaphore to 0),
- gains exclusive access to a resource (you initialize the semaphore to a value greater than 0), and
- signals the occurrence of an event (you initialize the semaphore to 0).

Arguments

`value` is the initial value of the semaphore and can be between 0 and 65,535. A value of 0 indicates that a resource is not available or an event has not occurred.

Returned Value

`OSSemCreate()` returns a pointer to the event control block allocated to the semaphore. If no event control block is available, `OSSemCreate()` returns a `NULL` pointer.

Notes/Warnings

1. Semaphores must be created before they are used.

Example

```
OS_EVENT *DispSem;



void main (void)
{
      .
      .
      OSInit();                      /* Initialize µC/OS-II              */
      .
      .
      DispSem = OSSemCreate(1);      /* Create Display Semaphore         */
      .
      .
      OSStart();                     /* Start Multitasking               */
}
```

OSSemDel()

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

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.

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

OS_ERR_NONE 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 pevent is a NULL pointer.

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.
3. All tasks that were waiting for the semaphore will be readied and returned an OS_ERR_PEND_ABORT error code if OSSemDel() was called with OS_DEL_ALWAYS option.

Example

```
OS_EVENT *DispSem;



void Task (void *p_arg)
{
    INT8U  err;



    (void)p_arg;
    for (;;) {
        .
        .
        DispSem = OSSemDel(DispSem, OS_DEL_ALWAYS, &err);
        if (DispSem == (OS_EVENT *)0) {
            /* Semaphore has been deleted */
        }
        .
        .
    }
}
```

OSSemPend()

```
void OSSemPend(OS_EVENT *pevent,
               INT32U    timeout,
               INT8U    *perr);
```

Chapter	***File***	***Called from***	***Code enabled by***
7	`OS_SEM.C`	Task only	`OS_SEM_EN`

`OSSemPend()` is used when a task wants exclusive access to a resource, needs to synchronize its activities with an ISR or a task, or is waiting until an event occurs. If a task calls `OSSemPend()` and the value of the semaphore is greater than 0, `OSSemPend()` decrements the semaphore and returns to its caller. However, if the value of the semaphore is 0, `OSSemPend()` places the calling task in the waiting list for the semaphore. The task waits until a task or an ISR signals the semaphore or the specified timeout expires. If the semaphore is signaled before the timeout expires, µC/OS-II resumes the highest priority task waiting for the semaphore. A pended task that has been suspended with `OSTaskSuspend()` can obtain the semaphore. However, the task remains suspended until it is resumed by calling `OSTaskResume()`.

Arguments

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

`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 waits forever for the message. The timeout value is not synchronized with the clock tick. The timeout count begins decrementing on the next clock tick, which could potentially occur immediately.

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

`OS_ERR_NONE`	if the semaphore is available.
`OS_ERR_TIMEOUT`	if the semaphore is not signaled within the specified timeout.
`OS_ERR_EVENT_TYPE`	if `pevent` is not pointing to a semaphore.
`OS_ERR_PEND_ISR`	if you called this function from an ISR and µC/OS-II has to suspend it. You should not call `OSSemPend()` from an ISR. µC/OS-II checks for this situation.
`OS_ERR_PEND_LOCKED`	if you called this function when the scheduler is locked.
`OS_ERR_PEVENT_NULL`	if `pevent` is a `NULL` pointer.
`OS_ERR_PEND_ABORT`	If the wait on the semaphore was aborted by a call to `OSSemPendAbort()` or, by calling `OSSemDel()` to delete the semaphore and this task was waiting on the semaphore.

Returned Value

none

Notes/Warnings

1. Semaphores must be created before they are used.

Example

```
OS_EVENT *DispSem;



void DispTask (void *p_arg)
{
      INT8U  err;



      (void)p_arg;
      for (;;) {
         .
         .
         OSSemPend(DispSem, 0, &err);
         .               /* The only way this task continues is if _ */
         .               /* _ the semaphore is signaled!             */
      }
}
```

OSSemPendAbort()

```
void *OSSemPendAbort(OS_EVENT *pevent,
                     INT8U     opt,
                     INT8U    *perr);
```

New Function	*File*	*Called from*	*Code enabled by*
V2.84	OS_SEM.C	Task only	OS_SEM_EN && OS_SEM_PEND_ABORT_EN

`OSSemPendAbort()` aborts & readies any tasks currently waiting on a semaphore. This function should be used to fault-abort the wait on the semaphore, rather than to normally signal the semaphore via `OSSemPost()`.

Arguments

`pevent` is a pointer to the semaphore for which pend(s) need to be aborted. This pointer is returned to your application when the semaphore is created [see `OSSemCreate()`].

`opt` determines what type of abort is performed.

- `OS_PEND_OPT_NONE` Aborts the pend of only the highest priority task waiting on the semaphore.
- `OS_PEND_OPT_BROADCAST` Aborts the pend of all the tasks waiting on the semaphore.

`perr` is a pointer to a variable that holds an error code. `OSSemPendAbort()` sets `*perr` to one of the following:

- `OS_ERR_NONE` if no tasks were waiting on the semaphore. In this case, the return value is also 0.
- `OS_ERR_PEND_ABORT` at least one task waiting on the semaphore was readied and informed of the aborted wait. Check the return value for the number of tasks whose wait on the semaphore was aborted.
- `OS_ERR_EVENT_TYPE` if `pevent` is not pointing to a semaphore.
- `OS_ERR_PEVENT_NULL` if `pevent` is a `NULL` pointer.

Returned Value

`OSSemPendAbort()` returns the number of tasks made ready to run by this function. Zero indicates that no tasks were pending on the semaphore and thus this function had no effect.

Notes/Warnings

1. Semaphores must be created before they are used.

Example

```
OS_EVENT *CommSem;



void CommTask(void *p_arg)
{
      INT8U  err;
      INT8U  nbr_tasks;



      (void)p_arg;
      for (;;) {
         .
         .
         nbr_tasks = OSSemPendAbort(CommSem, OS_PEND_OPT_BROADCAST, &err);
         if (err == OS_ERR_NONE) {
             .
             .  /* No tasks were waiting on the semaphore                 */
             .
         } else {
             .
             .  /* All pends of tasks waiting on semaphore were aborted … */
             .  /* … 'nbr_tasks' indicates how many were made ready.      */
             .
         }
         .
         .
      }
}
```

OSSemPost()

`INT8U OSSemPost(OS_EVENT *pevent);`

Chapter	*File*	*Called from*	*Code enabled by*
7	`OS_SEM.C`	Task or ISR	`OS_SEM_EN`

A semaphore is signaled by calling `OSSemPost()`. If the semaphore value is 0 or more, it is incremented, and `OSSemPost()` returns to its caller. If tasks are waiting for the semaphore to be signaled, `OSSemPost()` removes the highest priority task pending for the semaphore from the waiting list and makes this task ready to run. The scheduler is then called to determine if the awakened task is now the highest priority task ready to run.

Arguments

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

Returned Value

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

`OS_ERR_NONE`	if the semaphore is signaled successfully.
`OS_ERR_SEM_OVF`	if the semaphore count overflows.
`OS_ERR_EVENT_TYPE`	if `pevent` is not pointing to a semaphore.
`OS_ERR_PEVENT_NULL`	if `pevent` is a `NULL` pointer.

Notes/Warnings

1. Semaphores must be created before they are used.

Example

```
OS_EVENT *DispSem;



void TaskX (void *p_arg)
{
      INT8U  err;



      (void)p_arg;
      for (;;) {
         .
         .
         err = OSSemPost(DispSem);
         switch (err) {
             case OS_ERR_NONE:
                  /* Semaphore signaled       */
                  break;


             case OS_ERR_SEM_OVF:
                  /* Semaphore has overflowed */
                  break;
            .
            .
         }
         .
         .
      }
}
```

OSSemQuery()

```
INT8U OSSemQuery(OS_EVENT    *pevent,
                 OS_SEM_DATA *p_sem_data);
```

Chapter	*File*	*Called from*	*Code enabled by*
7	OS_SEM.C	Task or ISR	OS_SEM_EN && OS_SEM_QUERY_EN

OSSemQuery() obtains information about a semaphore. Your application must allocate an OS_SEM_DATA data structure used to receive data from the event control block of the semaphore. OSSemQuery() allows you to determine whether any tasks are waiting on the semaphore and how many tasks are waiting (by counting the number of 1s in the .OSEventTbl[] field) and obtains the semaphore count. 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 semaphore. This pointer is returned to your application when the semaphore is created [see OSSemCreate()].

P_sem_data is a pointer to a data structure of type OS_SEM_DATA, which contains the following fields

```
INT16U OSCnt;                            /* Current semaphore count     */
#if OS_LOWEST_PRIO <= 63
INT8U  OSEventTbl[OS_EVENT_TBL_SIZE];    /* Semaphore wait list         */
INT8U  OSEventGrp;
#else
INT16U OSEventTbl[OS_EVENT_TBL_SIZE];    /* Semaphore wait list         */
INT16U OSEventGrp;
#endif
```

Returned Value

OSSemQuery() returns one of these error codes:

OS_ERR_NONE	if the call is successful.
OS_ERR_EVENT_TYPE	if you don’t pass a pointer to a semaphore.
OS_ERR_PEVENT_NULL	if pevent is is a NULL pointer.
OS_ERR_PDATA_NULL	if p_sem_data is is a NULL pointer.

Notes/Warnings

1. Semaphores must be created before they are used.

Example

In this example, the contents of the semaphore is checked to determine the highest priority task waiting at the time the function call was made.

```
OS_EVENT *DispSem;



void Task (void *p_arg)
{
      OS_SEM_DATA sem_data;
      INT8U       err;
      INT8U       highest; /* Highest priority task waiting on sem. */
      INT8U       x;
      INT8U       y;



      (void)p_arg;
      for (;;) {
         .
         .
         err = OSSemQuery(DispSem, &sem_data);
         if (err == OS_ERR_NONE) {
            /* Examine sem_data */
                .
                .
         }
         .
         .
      }
}
```

OSSemSet()

```
void OSSemSet(OS_EVENT *pevent,
              INT16U    cnt,
              INT8U    *perr);
```

Chapter	*File*	*Called from*	*Code enabled by*
7	OS_SEM.C	Task or ISR	OS_SEM_EN && OS_SEM_SET_EN

OSSemSet() is used to change the current value of the semaphore count. This function would normally be used when a semaphore is used as a signaling mechanism. OSSemSet() can then be used to reset the count to any value. If the semaphore count is already 0 then, the count is only changed if there are no tasks waiting on the semaphore.

Arguments

pevent is a pointer to the semaphore that is used as a signaling mechanism. This pointer is returned to your application when the semaphore is created [see OSSemCreate()].

cnt is the desired count that you want the semaphore set to.

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

- OS_ERR_NONE if the count was changed or, not changed because there was one or more tasks waiting on the semaphore.
- OS_ERR_EVENT_TYPE if pevent is not pointing to a semaphore.
- OS_ERR_PEVENT_NULL if pevent is a NULL pointer.
- OS_ERR_TASK_WAITING if tasks are waiting on the semaphore.

Returned Value

None

Notes/Warnings

1. You should **NOT** use this function if the semaphore is used to protect a shared resource.

Example

```
OS_EVENT *SignalSem;


void Task (void *p_arg)
{
      INT8U err;



      (void)p_arg;
      for (;;) {
         OSSemSet(SignalSem, 0, &err);    /* Reset the semaphore count */
         .
         .
      }
}
```

OSStart()

`void OSStart(void);`

Chapter	*File*	*Called from*	*Code enabled by*
3	`OS_CORE.C`	Startup code only	N/A

`OSStart()` starts multitasking under µC/OS-II. This function is typically called from your startup code but after you call `OSInit()`.

Arguments

none

Returned Value

none

Notes/Warnings

1. `OSInit()` must be called prior to calling `OSStart()`. `OSStart()` should only be called once by your application code. If you do call `OSStart()` more than once, it does not do anything on the second and subsequent calls.

Example

```
void main (void)
{
    .                                   /* User Code            */
    .
    OSInit();                           /* Initialize µC/OS-II  */
    .                                   /* User Code            */
    .
    OSStart();                          /* Start Multitasking   */
    /* Any code here should NEVER be executed! */
}
```

OSStatInit()

```
void OSStatInit(void);
```

Chapter	*File*	*Called from*	*Code enabled by*
3	`OS_CORE.C`	Startup code only	`OS_TASK_STAT_EN && OS_TASK_CREATE_EXT_EN`

`OSStatInit()` determines the maximum value that a 32-bit counter can reach when no other task is executing. This function must be called when only one task is created in your application and when multitasking has started; that is, this function must be called from the first and, only, task created.

Arguments

none

Returned Value

none

Notes/Warnings

none

Example

```
void FirstAndOnlyTask (void *p_arg)
{
      .
      .
      OSStatInit();      /* Compute CPU capacity with no task running */
      .
      OSTaskCreate(_);   /* Create the other tasks                    */
      OSTaskCreate(_);
      .
      for (;;) {
         .
         .
      }
}
```

OSTaskChangePrio()

```
INT8U OSTaskChangePrio(INT8U oldprio,
                       INT8U newprio);
```

Chapter	*File*	*Called from*	*Code enabled by*
4	OS_TASK.C	Task only	OS_TASK_CHANGE_PRIO_EN

OSTaskChangePrio() changes the priority of a task.

Arguments

oldprio is the priority number of the task to change.

newprio is the new task's priority.

Returned Value

OSTaskChangePrio() returns one of the following error codes:

OS_ERR_NONE	if the task's priority is changed.
OS_ERR_PRIO_INVALID	if either the old priority or the new priority is equal to or exceeds OS_LOWEST_PRIO.
OS_ERR_PRIO_EXIST	if newprio already exists.
OS_ERR_PRIO	if no task with the specified old priority exists (i.e., the task specified by oldprio does not exist).
OS_ERR_TASK_NOT_EXITS	if the task is assigned to a Mutex PCP.

Notes/Warnings

1. The desired priority must not already have been assigned; otherwise, an error code is returned. Also, OSTaskChangePrio() verifies that the task to change exists.

Example

```
void TaskX (void *p_arg)
{
      INT8U  err;



      for (;;) {
         .
         .
         err = OSTaskChangePrio(10, 15);
         .
         .
      }
}
```

OSTaskCreate()

```
INT8U OSTaskCreate(void  (*task)(void *pd),
                   void   *pdata,
                   OS_STK *ptos,
                   INT8U   prio);
```

Chapter	*File*	*Called from*	*Code enabled by*
4	OS_TASK.C	Task or startup code	OS_TASK_CREATE_EN

OSTaskCreate() creates a task so it can be managed by µC/OS-II. Tasks can be created either prior to the start of multitasking or by a running task. A task cannot be created by an ISR. A task must be written as an infinite loop, as shown below, and must not return.

OSTaskCreate() is used for backward compatibility with µC/OS and when the added features of OSTaskCreateExt() are not needed.

Depending on how the stack frame is built, your task has interrupts either enabled or disabled. You need to check with the processor-specific code for details.

```
void Task (void *p_arg)
{
      .                    /* Do something with 'pdata'                */
      for (;;) {           /* Task body, always an infinite loop.      */
         .
         .
         /* Must call one of the following services:                   */
         /*    OSMboxPend()                                            */
         /*    OSFlagPend()                                            */
         /*    OSMutexPend()                                           */
         /*    OSQPend()                                               */
         /*    OSSemPend()                                             */
         /*    OSTimeDly()                                             */
         /*    OSTimeDlyHMSM()                                         */
         /*    OSTaskSuspend()     (Suspend self)                      */
         /*    OSTaskDel()         (Delete  self)                      */
         .
         .
      }
}
```

Arguments

`task` is a pointer to the task's code.

`pdata` is a pointer to an optional data area used to pass parameters to the task when it is created. Where the task is concerned, it thinks it is invoked and passes the argument `pdata`. `pdata` can be used to pass arguments to the task created. For example, you can create a generic task that handles an asynchronous serial port. `pdata` can be used to pass this task information about the serial port it has to manage: the port address, the baud rate, the number of bits, the parity, and more.

`ptos` is a pointer to the task's top-of-stack. The stack is used to store local variables, function parameters, return addresses, and CPU registers during an interrupt. The size of the stack is determined by the task's requirements and the anticipated interrupt nesting. Determining the size of the stack involves knowing how many bytes are required for storage of local variables for the task itself and all nested functions, as well as requirements for interrupts (accounting for nesting). If the configuration constant `OS_STK_GROWTH` is set to 1, the stack is assumed to grow downward (i.e., from high to low memory). `ptos` thus needs to point to the highest *valid* memory location on the stack. If `OS_STK_GROWTH` is set to 0, the stack is assumed to grow in the opposite direction (i.e., from low to high memory).

`prio` is the task priority. A unique priority number must be assigned to each task, and the lower the number, the higher the priority (i.e., the task importance).

Returned Value

`OSTaskCreate()` returns one of the following error codes:

`OS_ERR_NONE`	if the function is successful.
`OS_ERR_PRIO_EXIST`	if the requested priority already exists.
`OS_ERR_PRIO_INVALID`	if `prio` is higher than `OS_LOWEST_PRIO`.
`OS_ERR_NO_MORE_TCB`	if µC/OS-II doesn't have any more `OS_TCB`s to assign.
`OS_ERR_TASK_CREATE_ISR`	if you attempted to create the task from an ISR.

Notes/Warnings

1. The stack for the task must be declared with the `OS_STK` type.
2. A task must always invoke one of the services provided by µC/OS-II to wait for time to expire, suspend the task, or wait for an event to occur (wait on a mailbox, queue, or semaphore). This allows other tasks to gain control of the CPU.
3. You should not use task priorities `0`, `1`, `2`, `3`, `OS_LOWEST_PRIO-3`, `OS_LOWEST_PRIO-2`, `OS_LOWEST_PRIO-1`, and `OS_LOWEST_PRIO` because they are reserved for use by µC/OS-II.

Example 1

This example shows that the argument that `Task1()` receives is not used, so the pointer `pdata` is set to `NULL`. Note that I assume the stack grows from high to low memory because I pass the address of the highest valid memory location of the stack `Task1Stk[]`. If the stack grows in the opposite direction for the processor you are using, pass `&Task1Stk[0]` as the task's top-of-stack.

Assigning `pdata` to itself is used to prevent compilers from issuing a warning about the fact that `pdata` is not being used. In other words, if I had not added this line, some compilers would have complained about 'WARNING - variable `pdata` not used.'

```
OS_STK  Task1Stk[1024];


void main (void)
{
      INT8U err;


      .
      OSInit();                /* Initialize µC/OS-II             */
      .
      OSTaskCreate(Task1,
                   (void *)0,
                   &Task1Stk[1023],
                   25);
      .
      OSStart();               /* Start Multitasking              */
}



void Task1 (void *p_arg)
{
      (void)p_arg;              /* Prevent compiler warning      */
      for (;;) {
         .                      /* Task code                     */
         .
      }
}
```

Example 2

You can create a generic task that can be instantiated more than once. For example, a task that handles a serial port could be passed the address of a data structure that characterizes the specific port (i.e., port address and baud rate). Note that each task has its own stack space and its own (different) priority. In this example, I arbitrarily decided that COM1 is the most important port of the two.

```
OS_STK    *Comm1Stk[1024];
COMM_DATA  Comm1Data;           /* Data structure containing COMM port  */
                                /* Specific data for channel 1          */

OS_STK    *Comm2Stk[1024];
COMM_DATA  Comm2Data;           /* Data structure containing COMM port  */
                                /* Specific data for channel 2          */


void main (void)
{
   INT8U err;


   .
   OSInit();                    /* Initialize µC/OS-II                   */
   .
                                /* Create task to manage COM1           */
   OSTaskCreate(CommTask,
                (void *)&Comm1Data,
                &Comm1Stk[1023],
                25);
                                /* Create task to manage COM2           */
   OSTaskCreate(CommTask,
                (void *)&Comm2Data,
                &Comm2Stk[1023],
                26);
   .
   OSStart();                   /* Start Multitasking                   */
}


void CommTask (void *p_arg)     /* Generic communication task          */
{
      for (;;) {
         .                      /* Task code                           */
         .
      }
}
```

OSTaskCreateExt()

```
INT8U OSTaskCreateExt(void   (*task)(void *pd),
                      void    *pdata,
                      OS_STK  *ptos,
                      INT8U    prio,
                      INT16U   id,
                      OS_STK  *pbos,
                      INT32U   stk_size,
                      void    *pext,
                      INT16U   opt);
```

Chapter	***File***	***Called from***	***Code enabled by***
4	`OS_TASK.C`	Task or startup code	OS_TASK_CREATE_EXT_EN

`OSTaskCreateExt()` creates a task to be managed by µC/OS-II. This function serves the same purpose as `OSTaskCreate()`, except that it allows you to specify additional information about your task to µC/OS-II. Tasks can be created either prior to the start of multitasking or by a running task. A task cannot be created by an ISR. A task must be written as an infinite loop, as shown below, and must not return. Depending on how the stack frame is built, your task has interrupts either enabled or disabled. You need to check with the processor-specific code for details. Note that the first four arguments are exactly the same as the ones for `OSTaskCreate()`. This was done to simplify the migration to this new and more powerful function. It is highly recommended that you use `OSTaskCreateExt()` instead of the older `OSTaskCreate()` function because it's much more flexible.

```
void Task (void *p_arg)
{
      .                    /* Do something with 'pdata'                 */
      for (;;) {           /* Task body, always an infinite loop.       */
         .
         .
         /* Must call one of the following services:                    */
         /*    OSMboxPend()                                             */
         /*    OSFlagPend()                                             */
         /*    OSMutexPend()                                            */
         /*    OSQPend()                                                */
         /*    OSSemPend()                                              */
         /*    OSTimeDly()                                              */
         /*    OSTimeDlyHMSM()                                          */
         /*    OSTaskSuspend()     (Suspend self)                       */
         /*    OSTaskDel()         (Delete  self)                       */
         .
         .
      }
}
```

Arguments

`task` is a pointer to the task's code.

`pdata` is a pointer to an optional data area, which is used to pass parameters to the task when it is created. Where the task is concerned, it thinks it is invoked and passes the argument `pdata`. `pdata` can be used to pass arguments to the task created. For example, you can create a generic task that handles an asynchronous serial port. `pdata` can be used to pass this task information about the serial port it has to manage: the port address, the baud rate, the number of bits, the parity, and more.

`ptos` is a pointer to the task's top-of-stack. The stack is used to store local variables, function parameters, return addresses, and CPU registers during an interrupt.

The size of this stack is determined by the task's requirements and the anticipated interrupt nesting. Determining the size of the stack involves knowing how many bytes are required for storage of local variables for the task itself and all nested functions, as well as requirements for interrupts (accounting for nesting).

If the configuration constant `OS_STK_GROWTH` is set to 1, the stack is assumed to grow downward (i.e., from high to low memory). `ptos` thus needs to point to the highest *valid* memory location on the stack. If `OS_STK_GROWTH` is set to 0, the stack is assumed to grow in the opposite direction (i.e., from low to high memory).

`prio` is the task priority. A unique priority number must be assigned to each task: the lower the number, the higher the priority (i.e., the importance) of the task.

`id` is the task's ID number. At this time, the ID is not currently used in any other function and has simply been added in `OSTaskCreateExt()` for future expansion. You should set `id` to the same value as the task's priority.

`pbos` is a pointer to the task's bottom-of-stack. If the configuration constant `OS_STK_GROWTH` is set to 1, the stack is assumed to grow downward (i.e., from high to low memory); thus, `pbos` must point to the lowest valid stack location. If `OS_STK_GROWTH` is set to 0, the stack is assumed to grow in the opposite direction (i.e., from low to high memory); thus, `pbos` must point to the highest valid stack location. `pbos` is used by the stack-checking function `OSTaskStkChk()`.

`stk_size` specifies the size of the task's stack in number of elements. If `OS_STK` is set to `INT8U`, then `stk_size` corresponds to the number of bytes available on the stack. If `OS_STK` is set to `INT16U`, then `stk_size` contains the number of 16-bit entries available on the stack. Finally, if `OS_STK` is set to `INT32U`, then `stk_size` contains the number of 32-bit entries available on the stack.

`pext` is a pointer to a user-supplied memory location (typically a data structure) used as a TCB extension. For example, this user memory can hold the contents of floating-point registers during a context switch, the time each task takes to execute, the number of times the task is switched in, and so on.

`opt` contains task-specific options. The lower 8 bits are reserved by µC/OS-II, but you can use the upper 8 bits for application-specific options. Each option consists of one or more bits. The option is selected when the bit(s) is set. The current version of µC/OS-II supports the following options:

`OS_TASK_OPT_NONE`	specifies that there are no options.
`OS_TASK_OPT_STK_CHK`	specifies whether stack checking is allowed for the task.
`OS_TASK_OPT_STK_CLR`	specifies whether the stack needs to be cleared.
`OS_TASK_OPT_SAVE_FP`	specifies whether floating-point registers are saved. This option is only valid if your processor has floating-point hardware and the processor-specific code saves the floating-point registers.

Refer to `uCOS_II.H` for other options.

Returned Value

`OSTaskCreateExt()` returns one of the following error codes:

`OS_ERR_NONE`	if the function is successful.
`OS_ERR_PRIO_EXIST`	if the requested priority already exists.
`OS_ERR_PRIO_INVALID`	if `prio` is higher than `OS_LOWEST_PRIO`.
`OS_ERR_NO_MORE_TCB`	if µC/OS-II doesn't have any more `OS_TCB`s to assign.
`OS_ERR_TASK_CREATE_ISR`	if you attempted to create the task from an ISR.

Notes/Warnings

1. The stack must be declared with the `OS_STK` type.
2. A task must always invoke one of the services provided by µC/OS-II to wait for time to expire, suspend the task, or wait an event to occur (wait on a mailbox, queue, or semaphore). This allows other tasks to gain control of the CPU.
3. You should not use task priorities `0`, `1`, `2`, `3`, `OS_LOWEST_PRIO-3`, `OS_LOWEST_PRIO-2`, `OS_LOWEST_PRIO-1`, and `OS_LOWEST_PRIO` because they are reserved for use by µC/OS-II.

Example 1

E1(1) The task control block is extended using a user-defined data structure called `OS_TASK_USER_DATA`, which in this case contains the name of the task as well as other fields.

E1(2) The task name is initialized with the standard library function `strcpy()`.

E1(4) Note that stack checking has been enabled for this task, so you are allowed to call `OSTaskStkChk()`.

E1(3) Also, assume here that the stack grows downward on the processor used (i.e., `OS_STK_GROWTH` is set to 1; `TOS` stands for top-of-stack and `BOS` stands for bottom-of-stack).

```
typedef struct {                    /*  User defined data structure */        (1)
    char    OSTaskName[20];
    INT16U  OSTaskCtr;
    INT16U  OSTaskExecTime;
    INT32U  OSTaskTotExecTime;
} OS_TASK_USER_DATA;

OS_STK           TaskStk[1024];
TASK_USER_DATA   TaskUserData;


void main (void)
{
      INT8U err;

      .
      OSInit();                             /* Initialize µC/OS-II     */
      .
      strcpy(TaskUserData.TaskName, "MyTaskName");  /* Name of task     */   (2)
      err = OSTaskCreateExt(Task,
               (void *)0,
               &TaskStk[1023],              /*  Stack grows down (TOS) */   (3)
               10,
               10,
               &TaskStk[0],                 /*  Stack grows down (BOS) */   (3)
               1024,
               (void *)&TaskUserData,       /* TCB Extension           */
               OS_TASK_OPT_STK_CHK);        /*  Stack checking enabled */   (4)
      .
      OSStart();                            /* Start Multitasking      */
}

void Task(void *p_arg)
{
      (void)p_arg;                          /* Avoid compiler warning  */
      for (;;) {
         .                                  /* Task code               */
         .
      }
}
```

Example 2

E2(1) We now create a task, but this time on a processor for which the stack grows upward. The Intel MCS-51 is an example of such a processor. In this case, `OS_STK_GROWTH` is set to 0.

E2(2) Note that stack checking has been enabled for this task so you are allowed to call `OSTaskStkChk()` (`TOS` stands for top-of-stack and `BOS` stands for bottom-of-stack).

```
OS_STK *TaskStk[1024];



void main (void)
{
      INT8U err;


      .
      OSInit();                               /* Initialize µC/OS-II     */
      .
      err = OSTaskCreateExt(Task,
               (void *)0,
               &TaskStk[0],                   /*  Stack grows up (TOS)   */   (1)
               10,
               10,
               &TaskStk[1023],                /*  Stack grows up (BOS)   */   (1)
               1024,
               (void *)0,
               OS_TASK_OPT_STK_CHK);          /*  Stack checking enabled */   (2)
      .
      OSStart();                              /* Start Multitasking      */
}



void Task (void *p_arg)
{
      (void)p_arg;                            /* Avoid compiler warning  */
      for (;;) {
         .                                    /* Task code               */
         .
      }
}
```

OSTaskDel()

`INT8U OSTaskDel(INT8U prio);`

Chapter	File	Called from	Code enabled by
4	OS_TASK.C	Task only	OS_TASK_DEL_EN

`OSTaskDel()` deletes a task by specifying the priority number of the task to delete. The calling task can be deleted by specifying its own priority number or `OS_PRIO_SELF` (if the task doesn't know its own priority number). The deleted task is returned to the dormant state. The deleted task can be re-created by calling either `OSTaskCreate()` or `OSTaskCreateExt()` to make the task active again.

Arguments

`prio` is the priority number of the task to delete. You can delete the calling task by passing `OS_PRIO_SELF`, in which case the next highest priority task is executed.

Returned Value

`OSTaskDel()` returns one of the following error codes:

`OS_ERR_NONE`	if the task doesn't delete itself.
`OS_ERR_TASK_IDLE`	if you try to delete the idle task, which is of course is not allowed.
`OS_ERR_TASK_DEL`	if the task to delete does not exist.
`OS_ERR_PRIO_INVALID`	if you specify a task priority higher than `OS_LOWEST_PRIO`.
`OS_ERR_TASK_DEL_ISR`	if you try to delete a task from an ISR.
`OS_ERR_TASK_DEL`	if the task is assigned to a Mutex.
`OS_ERR_TASK_NOT_EXIST`	if the task is assigned to a Mutex PCP.

Notes/Warnings

1. `OSTaskDel()` verifies that you are not attempting to delete the µC/OS-II idle task.
2. You must be careful when you delete a task that owns resources. Instead, consider using `OSTaskDelReq()` as a safer approach.

Example

```
void TaskX (void *p_arg)
{
      INT8U err;

      for (;;) {
         .
         .
         err = OSTaskDel(10);      /* Delete task with priority 10  */
         if (err == OS_ERR_NONE) {
            .                      /* Task was deleted              */
            .
         }
         .
         .
      }
}
```

OSTaskDelReq()

`INT8U OSTaskDelReq(INT8U prio);`

Chapter	*File*	*Called from*	*Code enabled by*
4	OS_TASK.C	Task only	OS_TASK_DEL_EN

OSTaskDelReq() requests that a task delete itself. Basically, use OSTaskDelReq() when you need to delete a task that can potentially own resources (e.g., the task might own a semaphore). In this case, you don't want to delete the task until the resource is released. The requesting task calls OSTaskDelReq() to indicate that the task needs to be deleted. Deletion of the task is, however, deferred to the task being deleted. In other words, the task is actually deleted when it regains control of the CPU. For example, suppose Task 10 needs to be deleted. The task wanting to delete this task (example Task 5) calls OSTaskDelReq(10). When Task 10 executes, it calls OSTaskDelReq(OS_PRIO_SELF) and monitors the return value. If the return value is OS_ERR_TASK_DEL_REQ, then Task 10 is asked to delete itself. At this point, Task 10 calls OSTaskDel(OS_PRIO_SELF). Task 5 knows whether Task 10 has been deleted by calling OSTaskDelReq(10) and checking the return code. If the return code is OS_ERR_TASK_NOT_EXIST, then Task 5 knows that Task 10 has been deleted. Task 5 might have to check periodically until OS_ERR_TASK_NOT_EXIST is returned.

Arguments

prio is the task's priority number of the task to delete. If you specify OS_PRIO_SELF, you are asking whether another task wants the current task to be deleted.

Returned Value

OSTaskDelReq() returns one of the following error codes:

OS_ERR_NONE if the task deletion has been registered.

OS_ERR_TASK_NOT_EXIST if the task does not exist. The requesting task can monitor this return code to see if the task is actually deleted.

OS_ERR_TASK_IDLE if you ask to delete the idle task (which is obviously not allowed).

OS_ERR_PRIO_INVALID if you specify a task priority higher than OS_LOWEST_PRIO or do not specify OS_PRIO_SELF.

OS_ERR_TASK_DEL if the task is assigned to a Mutex.

OS_ERR_TASK_DEL_REQ if a task (possibly another task) requests that the running task be deleted.

Notes/Warnings

1. OSTaskDelReq() verifies that you are not attempting to delete the µC/OS-II idle task.

Example

```
void TaskThatDeletes (void *p_arg)   /* My priority is 5                  */
{
      INT8U err;


      for (;;) {
         .
         .
         err = OSTaskDelReq(10);     /* Request task #10 to delete itself */
         if (err == OS_ERR_NONE) {
            while (err != OS_ERR_TASK_NOT_EXIST) {
               err = OSTaskDelReq(10);
               OSTimeDly(1);         /* Wait for task to be deleted       */
            }
            .                        /* Task #10 has been deleted         */
         }
         .
         .
      }
}



void TaskToBeDeleted (void *p_arg)   /* My priority is 10                 */
{
   .
   .
   (void)p_arg;
   for (;;) {
      OSTimeDly(1);
      if (OSTaskDelReq(OS_PRIO_SELF) == OS_ERR_TASK_DEL_REQ) {
         /* Release any owned resources;                                  */
         /* De-allocate any dynamic memory;                               */
         OSTaskDel(OS_PRIO_SELF);
      }
   }
}
```

OSTaskNameGet()

```
INT8U OSTaskNameGet(INT8U    prio,
                    INT8U  **pname,
                    INT8U   *perr);
```

Chapter	File	Called from	Code enabled by
New in V2.60	OS_TASK.C	Task	OS_TASK_NAME_EN

OSTaskNameGet() allows you to obtain the name that you assigned to a task. 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 a pointer to an ASCII string that point to the name of the task.

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

OS_ERR_NONE	If the call is successfull.
OS_ERR_TASK_NOT_EXIST	The task you specified was not created or has been deleted.
OS_ERR_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.
OS_ERR_PNAME_NULL	If you passed a NULL pointer for pname.
OS_ERR_NAME_GET_ISR	You called this function from an ISR.

Returned Values

The size of the ASCII string 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.

Example

```
INT8U   *EngineTaskName;


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



    (void)p_arg;
    for (;;) {
        size = OSTaskNameGet(OS_PRIO_SELF, &EngineTaskName, &err);
        .
        .
    }
}
```

OSTaskNameSet()

```
void OSTaskNameSet(INT8U  prio,
                   INT8U *pname,
                   INT8U *perr);
```

Chapter	File	Called from	Code enabled by
New in V2.60	OS_TASK.C	Task	OS_TASK_NAME_EN

OSTaskNameSet() allows you to assign a name to a task. 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 contains the name of the task.

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

OS_ERR_NONE	If the call was successfull.
OS_ERR_TASK_NOT_EXIST	The task you specified was not created or has been deleted.
OS_ERR_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.
OS_ERR_PNAME_NULL	If you passed a NULL pointer for pname.
OS_ERR_NAME_SET_ISR	You called this function from an ISR.

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 *p_arg)
{
    INT8U     err;



    (void)p_arg;
    for (;;) {
        OSTaskNameSet(OS_PRIO_SELF, “Engine Task”, &err);
        .
        .
    }
}
```

OSTaskRegGet()

```
INT32U OSTaskRegGet(INT8U  prio,
                    INT8U  id,
                    INT8U *perr);
```

Chapter	*File*	*Called from*	*Code enabled by*
N/A	`OS_TASK.C`	Task or ISRs	`OS_TASK_REG_TBL_SIZE > 0`

µC/OS-II allows the user to store task-specific values in task registers. Task registers are different than CPU registers and are used to save such information as "`errno`" which are common in software components. Task registers can also store task-related data to be associated with the task at run time such as I/O register settings, configuration values, etc. A task may have as many as `OS_TASK_REG_TBL_SIZE` registers, and all registers have a data type of `INT32U`. A task register is changed by calling `OSTaskRegSet()` and read by calling `OSTaskRegGet()`. The desired task register is specified as an argument to these functions and can take a value between `0` and `OS_TASK_REG_TBL_SIZE-1`.

Arguments

`prio` specifies the priority of the task to obtain the value of the desired task register.

`id` is the ID (or index) of the desired task register. Valid IDs are from `0` to `OS_TASK_REG_TBL_SIZE-1`.

`perr` is a pointer to an error return code and can have one of the following values:

- `OS_ERR_NONE` if the call is successful.
- `OS_ERR_PRIO_INVALID` if you specified an invalid task priority.
- `OS_ERR_ID_INVALID` if `id` is not within the range `0` to `OS_TASK_REG_TBL_SIZE-1`.

Returned Value

The current value of the specified task register.

Notes/Warnings

none

Example

The example below shows how you can obtain the value of task register #3 for the task that has a priority of 10.

```
void TaskX (void *p_arg)
{
      INT8U   err;
      INT32U  reg_value;



      for (;;) {
         .
         .
         reg_value = OSTaskRegGet(10,
                                   3,
                                  &err);
         if (err == OS_ERR_NONE) {
            .
            .
         }
         .
         .
      }
}
```

OSTaskRegGetID()

```
INT8U OSTaskRegGetID(INT8U *perr);
```

Chapter	*File*	*Called from*	*Code enabled by*
N/A	OS_TASK.C	Tasks	OS_TASK_REG_TBL_SIZE > 0

µC/OS-II allows the user to store task-specific values in task registers. Task registers are different than CPU registers and are used to save such information as "errno" which are common in software components. Task registers can also store task-related data to be associated with the task at run time such as I/O register settings, configuration values, etc. A task may have as many as OS_TASK_REG_TBL_SIZE registers, and all registers have a data type of INT32U. A task register is changed by calling OSTaskRegSet() and read by calling OSTaskRegGet(). The desired task register is specified as an argument to these functions and can take a value between 0 and OS_TASK_REG_TBL_SIZE-1.

This function allows you to dynamically assign task register IDs for specific purposes. However, one a task register ID has been allocated, you MUST use the same ID for all task. In other words, if you want to assign a task register ID to 'errno' then you would call OSTaskRegGetID() which would return the 'next' available ID (let's say you get the value 3). From there on, ALL tasks that need to access 'their' copy of 'errno' would need to specify this ID (i.e. 3). Of course, you need to assign the returned value of OSTaskRegGetID() to a variable (see example).

You would thus call OSTaskRegGetID() before starting multitasking to assign IDs to all your application's task register.

Arguments

perr is a pointer to an error return code and can have one of the following values:

OS_ERR_NONE	if the call is successful.
OS_ERR_NO_MORE_ID_AVAIL	if you called OSTaskRegGetID() more than OS_TASK_REG_TBL_SIZE times and thus, there are no more IDs available to assign.
OS_ERR_ID_INVALID	if id is not within the range 0 to OS_TASK_REG_TBL_SIZE-1.

Returned Value

The next available ID number if the call was successful.
OS_TASK_REG_TBL_SIZE upon error.

Notes/Warnings

none

Example

The example below allocates the next available task register ID to ‘`ErrNoID`’ which must be declared globally so that any task that requires access to its own copy of ‘`ErrNo`’ would be able to.

```
INT8U  ErrNoID;           /* Globally defined ID */

void TaskX (void *p_arg)
{
      INT8U   err;



      for (;;) {
         .
         .
         ErrNoID = OSTaskRegGetID(&err);
         if (err == OS_ERR_NONE) {
            /* ErrNoID contains the index into each task’s task … */
            /* … for the ‘ErrNo’ variable                         */
         }
         .
         .
      }
}
```

OSTaskRegSet()

```
void  OSTaskRegSet(INT8U  prio,
                   INT8U  id,
                   INT32U value;
                   INT8U *perr);
```

Chapter	*File*	*Called from*	*Code enabled by*
N/A	`OS_TASK.C`	Task or ISRs	`OS_TASK_REG_TBL_SIZE > 0`

µC/OS-II allows the user to store task-specific values in task registers. Task registers are different than CPU registers and are used to save such information as "`errno`" which are common in software components. Task registers can also store task-related data to be associated with the task at run time such as I/O register settings, configuration values, etc. A task may have as many as `OS_TASK_REG_TBL_SIZE` registers, and all registers have a data type of `INT32U`. A task register is changed by calling `OSTaskRegSet()` and read by calling `OSTaskRegGet()`. The desired task register is specified as an argument to these functions and can take a value between `0` and `OS_TASK_REG_TBL_SIZE-1`.

Arguments

`prio` specifies the priority of the task to obtain the value of the desired task register.

`id` is the ID (or index) of the desired task register. Valid IDs are from `0` to `OS_TASK_REG_TBL_SIZE-1`.

`value` is the desired value that you want to assign to the task register.

`perr` is a pointer to an error return code and can have one of the following values:

- `OS_ERR_NONE` if the call is successful.
- `OS_ERR_PRIO_INVALID` if you specified an invalid task priority.
- `OS_ERR_ID_INVALID` if `id` is not within the range `0` to `OS_TASK_REG_TBL_SIZE-1`.

Returned Value

none

Notes/Warnings

none

Example

The example below shows how you can change the value of task register #3 to 0x12345678 for the task that has a priority of 10.

```
void TaskX (void *p_arg)
{
      INT8U   err;



      for (;;) {
         .
         .
         OSTaskRegGet(10,
                       3,
                      0x12345678,
                      &err);
         if (err == OS_ERR_NONE) {
            .
            .
         }
         .
         .
      }
}
```

OSTaskResume()

`INT8U OSTaskResume(INT8U prio);`

Chapter	*File*	*Called from*	*Code enabled by*
4	`OS_TASK.C`	Task only	`OS_TASK_SUSPEND_EN`

`OSTaskResume()` resumes a task suspended through the `OSTaskSuspend()` function. In fact, `OSTaskResume()` is the only function that can unsuspend a suspended task.

Arguments

`prio` specifies the priority of the task to resume.

Returned Value

`OSTaskResume()` returns one of the these error codes:

`OS_ERR_NONE`	if the call is successful.
`OS_ERR_TASK_RESUME_PRIO`	if the task you are attempting to resume does not exist.
`OS_ERR_TASK_NOT_SUSPENDED`	if the task to resume has not been suspended.
`OS_ERR_PRIO_INVALID`	if `prio` is higher or equal to `OS_LOWEST_PRIO`.
`OS_ERR_TASK_NOT_EXIST`	if the task is assigned to a Mutex PCP .

Notes/Warnings

none

Example

```
void TaskX (void *p_arg)
{
      INT8U err;



      for (;;) {
         .
         .
         err = OSTaskResume(10);         /* Resume task with priority 10 */
         if (err == OS_ERR_NONE) {
            .                            /* Task was resumed             */
            .
         }
         .
         .
      }
}
```

OSTaskStkChk()

```
INT8U OSTaskStkChk(INT8U        prio,
                   OS_STK_DATA *p_stk_data);
```

Chapter	*File*	*Called from*	*Code enabled by*
4	`OS_TASK.C`	Task code	`OS_TASK_CREATE_EXT`

`OSTaskStkChk()` determines a task's stack statistics. Specifically, it computes the amount of free stack space, as well as the amount of stack space used by the specified task. This function requires that the task be created with `OSTaskCreateExt()` and that you specify `OS_TASK_OPT_STK_CHK` in the `opt` argument.

Stack sizing is done by walking from the bottom of the stack and counting the number of 0 entries on the stack until a nonzero value is found. Of course, this assumes that the stack is cleared when the task is created. For that purpose, you need to set `OS_TASK_OPT_STK_CLR` to 1 as an option when you create the task. You could set `OS_TASK_OPT_STK_CLR` to 0 if your startup code clears all RAM and you never delete your tasks. This reduces the execution time of `OSTaskCreateExt()`.

Arguments

`prio` is the priority of the task about which you want to obtain stack information. You can check the stack of the calling task by passing `OS_PRIO_SELF`.

`P_stk_data` is a pointer to a variable of type `OS_STK_DATA`, which contains the following fields:

```
    INT32U OSFree;        /* Number of free entries on the stack         */
    INT32U OSUsed;        /* Number of entries used on the stack         */
```

Returned Value

`OSTaskStkChk()` returns one of the these error codes:

`OS_ERR_NONE`	if you specify valid arguments and the call is successful.
`OS_ERR_PRIO_INVALID`	if you specify a task priority higher than `OS_LOWEST_PRIO` or you don't specify `OS_PRIO_SELF`.
`OS_ERR_TASK_NOT_EXIST`	if the specified task does not exist.
`OS_ERR_TASK_OPT_ERR`	if you do not specify `OS_TASK_OPT_STK_CHK` when the task was created by `OSTaskCreateExt()` or if you create the task by using `OSTaskCreate()`.
`OS_ERR_PDATA_NULL`	if `p_stk_data` is a `NULL` pointer.

Notes/Warnings

1. Execution time of this function depends on the size of the task's stack and is thus nondeterministic.
2. Your application can determine the total task stack space (in number of entries) by adding the two fields `.OSFree` and `.OSUsed` of the `OS_STK_DATA` data structure.
3. Technically, this function can be called by an ISR, but because of the possibly long execution time, it is not advisable.

Example

```
void Task (void *p_arg)
{
      OS_STK_DATA stk_data;
      INT32U      stk_size;



      (void)p_arg;
      for (;;) {
         .
         .
         err = OSTaskStkChk(10, &stk_data);
         if (err == OS_ERR_NONE) {
            stk_size = stk_data.OSFree + stk_data.OSUsed;
         }
         .
         .
      }
}
```

OSTaskSuspend()

`INT8U OSTaskSuspend(INT8U prio);`

Chapter	*File*	*Called from*	*Code enabled by*
4	OS_TASK.C	Task only	OS_TASK_SUSPEND_EN

`OSTaskSuspend()` suspends (or blocks) execution of a task unconditionally. The calling task can be suspended by specifying its own priority number or `OS_PRIO_SELF` if the task doesn't know its own priority number. In this case, another task needs to resume the suspended task. If the current task is suspended, rescheduling occurs, and µC/OS-II runs the next highest priority task ready to run. The only way to resume a suspended task is to call `OSTaskResume()`.

Task suspension is additive, which means that if the task being suspended is delayed until *n* ticks expire, the task is resumed only when both the time expires and the suspension is removed. Also, if the suspended task is waiting for a semaphore and the semaphore is signaled, the task is removed from the semaphore-wait list (if it is the highest priority task waiting for the semaphore), but execution is not resumed until the suspension is removed.

Arguments

`prio` specifies the priority of the task to suspend. You can suspend the calling task by passing `OS_PRIO_SELF`, in which case, the next highest priority task is executed.

Returned Value

`OSTaskSuspend()` returns one of the these error codes:

`OS_ERR_NONE`	if the call is successful.
`OS_ERR_TASK_SUSPEND_IDLE`	if you attempt to suspend the _C/OS-II idle task, which is not allowed.
`OS_ERR_PRIO_INVALID`	if you specify a priority higher than the maximum allowed (i.e., you specify a priority of `OS_LOWEST_PRIO` or more) or you don't specify `OS_PRIO_SELF`.
`OS_ERR_TASK_SUSPEND_PRIO`	if the task you are attempting to suspend does not exist.
`OS_ERR_TASK_NOT_EXITS`	if the task is assigned to a Mutex PCP.

Notes/Warnings

1. `OSTaskSuspend()` and `OSTaskResume()` must be used in pairs.
2. A suspended task can only be resumed by `OSTaskResume()`.

Example

```
void TaskX (void *p_arg)
{
      INT8U err;



      (void)p_arg;
      for (;;) {
         .
         .
         err = OSTaskSuspend(OS_PRIO_SELF);  /* Suspend current task    */
         .                 /* Execution continues when ANOTHER task ..   */
         .                 /* .. explicitly resumes this task.           */
         .
      }
}
```

OSTaskQuery()

```
INT8U OSTaskQuery(INT8U   prio,
                  OS_TCB *p_task_data);
```

Chapter	File	Called from	Code enabled by
4	OS_TASK.C	Task or ISR	OS_TASK_QUERY_EN

`OSTaskQuery()` obtains information about a task. Your application must allocate an `OS_TCB` data structure to receive a snapshot of the desired task's control block. Your copy contains *every* field in the `OS_TCB` structure. You should be careful when accessing the contents of the `OS_TCB` structure, especially `OSTCBNext` and `OSTCBPrev`, because they point to the next and previous `OS_TCB`s in the chain of created tasks, respectively. You could use this function to provide a debugger kernel awareness.

Arguments

`prio` is the priority of the task from which you wish to obtain data. You can obtain information about the calling task by specifying `OS_PRIO_SELF`.

`p_task_data` is a pointer to a structure of type `OS_TCB`, which contains a copy of the task's control block.

Returned Value

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

`OS_ERR_NONE`	if the call is successful.
`OS_ERR_PRIO_INVALID`	if you specify a priority higher than `OS_LOWEST_PRIO`.
`OS_ERR_PRIO`	if you try to obtain information from an invalid task.
`OS_ERR_TASK_NOT_EXIST`	if the task is assigned to a Mutex PCP.
`OS_ERR_PDATA_NULL`	if `p_task_data` is a `NULL` pointer.

Notes/Warnings

1. The fields in the task control block depend on the following configuration options (see `OS_CFG.H`):
   - `OS_TASK_CREATE_EN`
   - `OS_Q_EN`
   - `OS_FLAG_EN`
   - `OS_MBOX_EN`
   - `OS_SEM_EN`
   - `OS_TASK_DEL_EN`

Example

```
void Task (void *p_arg)
{
      OS_TCB  task_data;
      INT8U   err;
      void   *pext;
      INT8U   status;


      (void)p_arg;
      for (;;) {
         .
         .
         err = OSTaskQuery(OS_PRIO_SELF, &task_data);
         if (err == OS_ERR_NONE) {
            pext   = task_data.OSTCBExtPtr; /* Get TCB extension pointer   */
            status = task_data.OSTCBStat;   /* Get task status             */
            .
            .
         }
         .
         .
      }
}
```

OSTimeDly()

`void OSTimeDly(INT32U ticks);`

Chapter	*File*	*Called from*	*Code enabled by*
5	`OS_TIME.C`	Task only	N/A

`OSTimeDly()` allows a task to delay itself for an integral number of clock ticks. Rescheduling always occurs when the number of clock ticks is greater than zero. Valid delays range from one to 2^{32}-1 ticks. A delay of 0 means that the task is not delayed, and `OSTimeDly()` returns immediately to the caller. The actual delay time depends on the tick rate (see `OS_TICKS_PER_SEC` in the configuration file `OS_CFG.H`).

Arguments

`ticks` is the number of clock ticks to delay the current task.

Returned Value

none

Notes/Warnings

1. Note that calling this function with a value of 0 results in no delay, and the function returns immediately to the caller.
2. To ensure that a task delays for the specified number of ticks, you should consider using a delay value that is one tick higher. For example, to delay a task for at least 10 ticks, you should specify a value of 11.

Example

```
void TaskX (void *p_arg)
{
      for (;;) {
         .
         .
         OSTimeDly(10);                /* Delay task for 10 clock ticks */
         .
         .
      }
}
```

OSTimeDlyHMSM()

```
void OSTimeDlyHMSM (INT8U  hours,
                    INT8U  minutes,
                    INT8U  seconds,
                    INT16U ms);
```

Chapter	***File***	***Called from***	***Code enabled by***
5	`OS_TIME.C`	Task only	N/A

`OSTimeDlyHMSM()` allows a task to delay itself for a user-specified amount of time specified in hours, minutes, seconds, and milliseconds. This format is more convenient and natural than ticks. Rescheduling always occurs when at least one of the parameters is nonzero.

Arguments

`hours` is the number of hours the task is delayed. The valid range of values is 0 to 255.

`minutes` is the number of minutes the task is delayed. The valid range of values is 0 to 59.

`seconds` is the number of seconds the task is delayed. The valid range of values is 0 to 59.

`ms` is the number of milliseconds the task is delayed. The valid range of values is 0 to 999. Note that the resolution of this argument is in multiples of the tick rate. For instance, if the tick rate is set to 100Hz, a delay of 4ms results in no delay. The delay is rounded to the nearest tick. Thus, a delay of 15ms actually results in a delay of 20ms.

Returned Value

`OSTimeDlyHMSM()` returns one of the these error codes:

`OS_ERR_NONE` if you specify valid arguments and the call is successful.

`OS_ERR_TIME_INVALID_MINUTES` if the minutes argument is greater than 59.

`OS_ERR_TIME_INVALID_SECONDS` if the seconds argument is greater than 59.

`OS_ERR_TIME_INVALID_MS` if the milliseconds argument is greater than 999.

`OS_ERR_TIME_ZERO_DLY` if all four arguments are 0.

`OS_ERR_TIME_DLY_ISR` if you called this function from an ISR.

Notes/Warnings

1. Note that `OSTimeDlyHMSM(0,0,0,0)` (i.e., hours, minutes, seconds, milliseconds) results in no delay, and the function returns to the caller.

Example

```
void TaskX (void *p_arg)
{
      for (;;) {
         .
         .
         OSTimeDlyHMSM(0, 0, 1, 0);  /* Delay task for 1 second */
         .
         .
      }
}
```

OSTimeDlyResume()

`INT8U OSTimeDlyResume(INT8U prio);`

Chapter	*File*	*Called from*	*Code enabled by*
5	`OS_TIME.C`	Task only	N/A

`OSTimeDlyResume()` resumes a task that has been delayed through a call to either `OSTimeDly()` or `OSTimeDlyHMSM()`.

Arguments

`prio` specifies the priority of the task to resume.

Returned Value

`OSTimeDlyResume()` returns one of the these error codes:

`OS_ERR_NONE`	if the call is successful.
`OS_ERR_PRIO_INVALID`	if you specify a task priority greater than `OS_LOWEST_PRIO`.
`OS_ERR_TIME_NOT_DLY`	if the task is not waiting for time to expire.
`OS_ERR_TASK_NOT_EXIST`	if the task has not been created or has been assigned to a Mutex PCP.

Notes/Warnings

1. Note that you must not call this function to resume a task that is waiting for an event with timeout. This situation makes the task look like a timeout occurred (unless you desire this effect).

Example

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

   pdata = pdata;
   for (;;) {
      .
      err = OSTimeDlyResume(10); /* Resume task with priority 10 */
      if (err == OS_ERR_NONE) {
         .                       /* Task was resumed             */
         .
      }
      .
   }
}
```

OSTimeGet()

`INT32U OSTimeGet(void);`

Chapter	*File*	*Called from*	*Code enabled by*
5	`OS_TIME.C`	Task or ISR	N/A

`OSTimeGet()` obtains the current value of the system clock. The system clock is a 32-bit counter that counts the number of clock ticks since power was applied or from a value set by `OSTimeSet()`.

Arguments

none

Returned Value

The current system clock value (in number of ticks).

Notes/Warnings

none

Example

```
void TaskX (void *p_arg)
{
      INT32U clk;



      for (;;) {
         .
         .
         clk = OSTimeGet();  /* Get current value of system clock */
         .
         .
      }
}
```

OSTimeSet()

`void OSTimeSet(INT32U ticks);`

Chapter	***File***	***Called from***	***Code enabled by***
5	`OS_TIME.C`	Task or ISR	N/A

`OSTimeSet()` sets the system clock. The system clock is a 32-bit counter that counts the number of clock ticks since power was applied or since the system clock was last set.

Arguments

`ticks` is the desired value for the system clock, in ticks.

Returned Value

none

Notes/Warnings

none

Example

```
void TaskX (void *p_arg)
{
      for (;;) {
         .
         .
         OSTimeSet(0L);    /* Reset the system clock  */
         .
         .
      }
}
```

OSTimeTick()

`void OSTimeTick(void);`

Chapter	*File*	*Called from*	*Code enabled by*
5	`OS_TIME.C`	Task or ISR	N/A

`OSTimeTick()` processes a clock tick. µC/OS-II checks all tasks to see if they are either waiting for time to expire [because they called `OSTimeDly()` or `OSTimeDlyHMSM()`] or waiting for events to occur until they timeout.

Arguments

none

Returned Value

none

Notes/Warnings

1. The execution time of `OSTimeTick()` is directly proportional to the number of tasks created in an application. `OSTimeTick()` can be called by either an ISR or a task. If called by a task, the task priority should be very high (i.e., have a low priority number) because this function is responsible for updating delays and timeouts.

Example

(Intel 80x86, real mode, large model)

```
_OSTickISR PROC FAR
               PUSHA                                ; Save processor context
               PUSH ES
               PUSH DS
;
               MOV    AX, SEG(_OSIntNesting)        ; Reload DS
               MOV    DS, AX
               INC    BYTE PTR DS:_OSIntNesting     ; Notify µC/OS-II of ISR
;
               CMP    BYTE PTR DS:_OSIntNesting, 1  ; if (OSIntNesting == 1)
               JNE    SHORT _OSTickISR1
               MOV    AX, SEG(_OSTCBCur)            ;     Reload DS
               MOV    DS, AX
               LES    BX, DWORD PTR DS:_OSTCBCur    ;     OSTCBCur->OSTCBStkPtr = SS:SP
               MOV    ES:[BX+2], SS                 ;
               MOV    ES:[BX+0], SP                 ;
               CALL  FAR  PTR  _OSTimeTick          ; Process clock tick
               .                                    ; User Code to clear interrupt
               .
               CALL FAR PTR _OSIntExit              ; Notify µC/OS-II of end of ISR
               POP  DS                              ; Restore processor registers
               POP  ES
               POPA
;
               IRET                                ; Return to interrupted task
_OSTickISR     ENDP
```

OSTmrCreate()

```
OS_TMR *OSTmrCreate(INT32U           dly,
                    INT32U           period,
                    INT8U            opt,
                    OS_TMR_CALLBACK  callback,
                    void            *callback_arg,
                    INT8U           *pname,
                    INT8U           *perr);
```

Chapter	File	Called from	Code enabled by
New in V2.83	OS_TMR.C	Task	OS_TMR_EN

OSTmrCreate() allows you to create a timer. The timer can be configured to run continuously (opt set to OS_TMR_OPT_PERIODIC) or only once (opt set to OS_TMR_OPT_ONE_SHOT). When the timer counts down to 0 (from the value specified in period), an optional 'callback' function can be executed. The callback can be used to signal a task that the timer expired or, perform any other function. However, it's recommended that you keep the callback function as short as possible.

You **MUST** call OSTmrStart() to actually start the timer. If you configured the timer for one shot mode and the timer expired, you need to call OSTmrStart() to retrigger the timer or OSTmrDel() to delete the timer if you don't plan on retriggering it and or not use the timer anymore. Note that you can use the callback function to delete the timer if you use the one shot mode.

Arguments

dly specifies an initial delay used by the timer (see drawing below).

In ONE-SHOT mode, this is the time of the one-shot.

If in PERIODIC mode, this is the initial delay before the timer enters periodic mode.

The units of this time depends on how often you call OSTmrSignal(). In other words, if OSTmrSignal() is called every 1/10 of a second (i.e. OS_TMR_CFG_TICKS_PER_SEC set to 10) then, dly specifies the number of 1/10 of a second before the delay expires. Note that the timer is **NOT** started when it is created.

period specifies the amount of time it will take before the timer expires. You should set the 'period' to 0 when you use one-shot mode. The units of this time depends on how often you call OSTmrSignal(). In other words, if OSTmrSignal() is called every 1/10 of a second (i.e. OS_TMR_CFG_TICKS_PER_SEC set to 10) then, period specifies the number of 1/10 of a second before the timer times out.

opt OS_TMR_OPT_PERIODIC:
specifies whether you want to timer to automatically reload itself.

OS_TMR_OPT_ONE_SHOT:
specifies to stop the timer when it times out.

Note that you **MUST** select one of these two options.

`callback` specifies the address of a function (optional) that you want to execute when the timer expires or, is terminated before it expires (i.e. by calling `OSTmrStop()`). The callback function must be declared as follows:

`void MyCallback (void *ptmr, void *callback_arg);`

When the timer expires, this function will be called and passed the timer 'handle' of the expiring timer as well as the argument specified by `callback_arg`.

You should note that you don't have to specify a callback and, in this case, simply pass a `NULL` pointer.

`callback_arg` Is the argument passed to the callback function when the timer expires or is terminated. `callback_arg` can be a `NULL` pointer if the callback function doesn't require arguments.

`pname` Is a pointer to an ASCII string that allows you to give a name to your timer. You can retrieve this name by calling `OSTmrNameGet()`.

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

`OS_ERR_NONE` If the timer was created successfully.

`OS_ERR_TMR_INVALID_DLY` You specified a delay of 0 when in ONE SHOT mode.

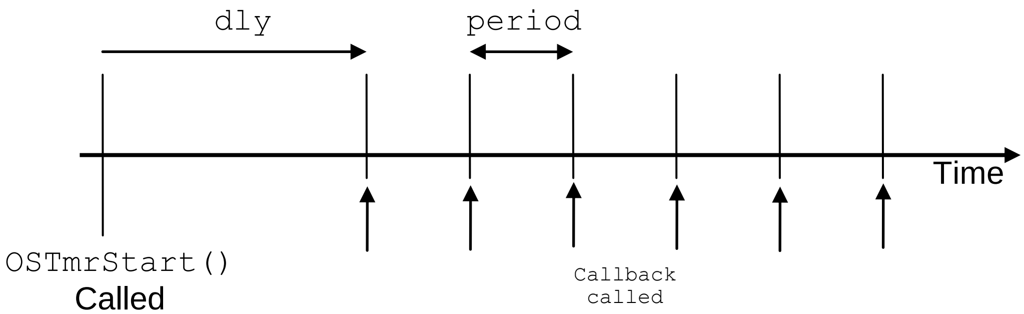
`OS_ERR_TMR_INVALID_PERIOD` You specified a period of 0 when in PERIODIC mode.

`OS_ERR_TMR_INVALID_OPT` If you did not specify either `OS_TMR_OPT_PERIODIC` or `OS_TMR_OPT_ONE_SHOT`.

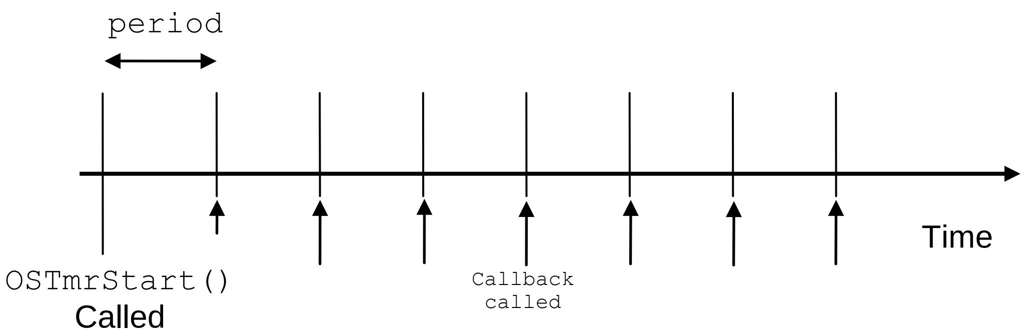
`OS_ERR_TMR_ISR` If you called this function from an ISR, which you are not allowed to do.

`OS_ERR_TMR_NON_AVAIL` You get this error when you cannot start a timer because all timer elements (i.e. objects) have already been allocated.

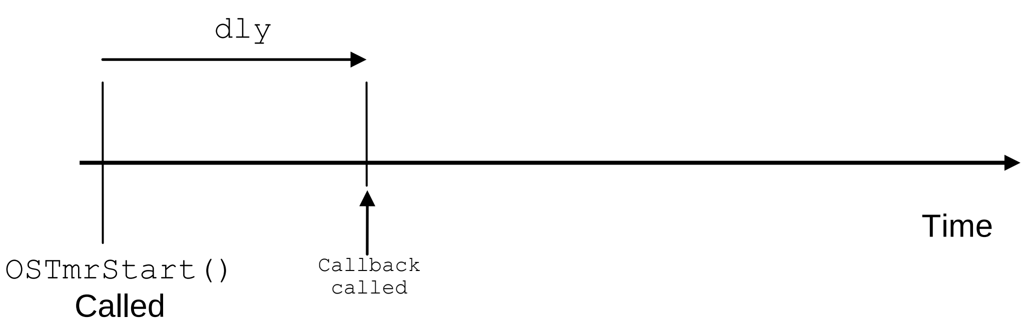
PERIODIC MODE (see 'opt') – `dly > 0`



PERIODIC MODE (see 'opt') – `dly == 0`



ONE-SHOT MODE (see 'opt') – `dly` MUST be non-zero



Returned Values

A pointer to an `OS_TMR` object that you **MUST** use to reference the timer that you just created. A `NULL` pointer is returned if the timer was not created because of errors (see returned error codes).

Notes/Warnings

1. You should examine the return value to make sure what you get from this function is valid.
2. You **MUST NOT** call this function from an ISR.
3. Note that the timer is **NOT** started when it is created. To start the timer, you **MUST** call `OSTmrStart()`.

Example

```
OS_TMR  *CloseDoorTmr;



void Task (void *p_arg)
{
    INT8U     err;


    (void)p_arg;
    for (;;) {
        CloseDoorTmr = OSTmrCreate( 10,
                                   100,
                                   OS_TMR_OPT_PERIODIC,
                                   DoorCloseFnct,
                                   (void *)0,
                                   "Door Close",
                                   &err);
        if (err == OS_ERR_NONE) {
            /* Timer was created but NOT started */
        }
    }
}
```

OSTmrDel()

```
BOOLEAN OSTmrDel(OS_TMR     *ptmr,
                 INT8U      *perr);
```

Chapter	File	Called from	Code enabled by
New in V2.83	OS_TMR.C	Task	OS_TMR_EN

`OSTmrDel()` allows you to delete a timer. If a timer was running, it will be stopped and then deleted. If the timer has already timed out and is thus stopped, it will simply be deleted.

It is up to you to delete unused timers. If you delete a timer you **MUST NOT** reference it anymore.

Arguments

`ptmr` is a pointer to the timer that you want to delete. This pointer is returned to you when the timer is created (see `OSTmrCreate()`).

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

`OS_ERR_NONE`	If the timer was deleted successfully.
`OS_ERR_TMR_INVALID`	If you passed a `NULL` pointer for the `ptmr` argument.
`OS_ERR_TMR_INVALID_TYPE`	'`ptmr`' is not pointing to a timer.
`OS_ERR_TMR_ISR`	You called this function from an ISR which is NOT allowed.
`OS_ERR_TMR_INACTIVE`	`ptmr` is pointing to an inactive timer. In other words, you would get this error if you are pointing to a timer that has been deleted or was not created.
`OS_ERR_TMR_INVALID_STATE`	The timer is in an invalid state.

Returned Values

`OS_TRUE` if the timer was deleted
`OS_FALSE` if an error occurred.

Notes/Warnings

1. You should examine the return value to make sure what you get from this function is valid.
2. You **MUST NOT** call this function from an ISR.
3. If you delete a timer you **MUST NOT** reference it anymore.

Example

```
OS_TMR  *CloseDoorTmr;



void Task (void *p_arg)
{
    INT8U     err;


    (void)p_arg;
    for (;;) {
        CloseDoorTmr = OSTmrDel(CloseDoorTmr,
                                &err);
        if (err == OS_ERR_NONE) {
            /* Timer was deleted ... DO NOT reference it anymore! */
        }
    }
}
```

OSTmrNameGet()

```
INT8U  OSTmrNameGet(OS_TMR    *ptmr,
                    INT8U    **pdest,
                    INT8U     *perr);
```

Chapter	File	Called from	Code enabled by
New in V2.81	OS_TMR.C	Task	OS_TMR_EN && OS_TMR_CFG_NAME_EN

OSTmrNameGet() allows you to retrieve the name associated with the specified timer. OSTmrNameGet() places the name of the timer in an array of characters which must be as big as OS_TMR_CFG_NAME_SIZE (see OS_CFG.H).

Arguments

ptmr — is a pointer to the timer that you are inquiring about. This pointer is returned to you when the timer is created (see OSTmrCreate()).

pdest — is a pointer to a pointer to the name of the timer.

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

OS_ERR_NONE	If the name of the task was copied to the array pointed to by pname.
OS_ERR_TMR_INVALID_DEST	If you specified a NULL pointer for pdest.
OS_ERR_TMR_INVALID	If you passed a NULL pointer for the ptmr argument.
OS_ERR_TMR_INVALID_TYPE	'ptmr' is not pointing to a timer.
OS_ERR_NAME_GET_ISR	You called this function from an ISR which is **NOT** allowed.
OS_ERR_TMR_INACTIVE	ptmr is pointing to an inactive timer. In other words, you would get this error if you are pointing to a timer that has been deleted or was not created.
OS_ERR_TMR_INVALID_STATE	The timer is in an invalid state.

Returned Values

The length of the timer name (in number of characters).

Notes/Warnings

1. You should examine the return value of this function.
2. You **MUST NOT** call this function from an ISR.

Example

```
INT8U   *CloseDoorTmrName;
OS_TMR  *CloseDoorTmr;

void Task (void *p_arg)
{
    INT8U     err;


    (void)p_arg;
    for (;;) {
        OSTmrNameGet(CloseDoorTmr, &CloseDoorTmrName, &err);
        if (err == OS_ERR_NONE) {
            /* CloseDoorTmrName points to the name of the timer */
        }
    }
}
```

OSTmrRemainGet()

```
INT32U OSTmrRemainGet(OS_TMR *ptmr,
                      INT8U  *perr);
```

Chapter	File	Called from	Code enabled by
New in V2.81	OS_TMR.C	Task	OS_TMR_EN

OSTmrRemainGet() allows you to obtain the time remaining (before it times out) of the specified timer. The value returned depends on the rate (in Hz) at which the timer task is signaled (see OS_TMR_CFG_TICKS_PER_SEC in OS_CFG.H). In other words, if OS_TMR_CFG_TICKS_PER_SEC is set to 10 then the value returned is the number of 1/10 of a second before the timer times out. If the timer has timed out, the value returned will be 0.

Arguments

ptmr is a pointer to the timer that you are inquiring about. This pointer is returned to you when the timer is created (see OSTmrCreate()).

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

OS_ERR_NONE	If the function returned the time remaining for the timer.
OS_ERR_TMR_INVALID	If you passed a NULL pointer for the ptmr argument.
OS_ERR_TMR_INVALID_TYPE	'ptmr' is not pointing to a timer.
OS_ERR_TMR_ISR	You called this function from an ISR which is NOT allowed.
OS_ERR_TMR_INACTIVE	ptmr is pointing to an inactive timer. In other words, you would get this error if you are pointing to a timer that has been deleted or was not created.
OS_ERR_TMR_INVALID_STATE	The timer is in an invalid state.

Returned Values

The time remaining for the timer. The value returned depends on the rate (in Hz) at which the timer task is signaled (see OS_TMR_CFG_TICKS_PER_SEC in OS_CFG.H). In other words, if OS_TMR_CFG_TICKS_PER_SEC is set to 10 then the value returned is the number of 1/10 of a second before the timer times out. If you specified an invalid timer, the returned value will be 0. If the timer has already expired then the returned value will be 0.

Notes/Warnings

1. You should examine the return value to make sure what you get from this function is valid.
2. You **MUST NOT** call this function from an ISR.

Example

```
INT32U   TimeRemainToCloseDoor;
OS_TMR  *CloseDoorTmr;

void Task (void *p_arg)
{
    INT8U     err;


    (void)p_arg;
    for (;;) {
        TimeRemainToCloseDoor = OSTmrRemainGet(CloseDoorTmr, &err);
        if (err == OS_ERR_NONE) {
            /* Call was successful */
        }
    }
}
```

OSTmrSignal()

`INT8U OSTmrSignal(void);`

Chapter	File	Called from	Code enabled by
New in V2.81	`OS_TMR.C`	Task or ISR	`OS_TMR_EN`

`OSTmrSignal()` is called either by a task or an ISR to indicate that it's time to update the timers. Typically, `OSTmrSignal()` would be called by `OSTimeTickHook()` at a multiple of the tick rate. In other words, if `OS_TICKS_PER_SEC` is set to `1000` in `OS_CFG.H` then you should call `OSTmrSignal()` every 10th or 100th tick interrupt (100 Hz or 10 Hz, respectively). You should typically call `OSTmrSignal()` every 1/10 of a second. The higher the timer rate, of course, the more overhead timer management will impose on your system. Generally, we recommend 10 Hz (1/10 of a second).

You 'could' call `OSTmrSignal()` from the µC/OS-II tick ISR hook function (see example below). If the tick rate occurs at 1000 Hz then you can simply call `OSTmrSignal()` every 100th tick. Of course, you would have to implement a simple counter to do this.

Arguments

None.

Returned Values

`OSTmrSignal()` uses semaphores to implement the signaling mechanism. Because of that, `OSTmrSignal()` can return one of the following errors. However, it's very unlikely you will get anything else but `OS_ERR_NONE`.

`OS_ERR_NONE`	The call was successful and the timer task was signaled.
`OS_ERR_SEM_OVF`	If `OSTmrSignal()` was called more often than `OSTmr_Task()` can handle the timers. This would indicate that your system is heavily loaded.
`OS_ERR_EVENT_TYPE`	Unlikely you would get this error because the semaphore used for signaling is created by µC/OS-II.
`OS_ERR_PEVENT_NULL`	Again, unlikely you would ever get this error because the semaphore used for signaling is created by µC/OS-II.

Notes/Warnings

None.

Example

```
#if OS_TMR_EN > 0
static  INT16U  OSTmrTickCtr = 0;
#endif



void OSTimeTickHook (void)
{
#if OS_TMR_EN > 0
    OSTmrTickCtr++;
    if (OSTmrTickCtr >= (OS_TICKS_PER_SEC / OS_TMR_CFG_TICKS_PER_SEC)) {
        OSTmrTickCtr = 0;
        OSTmrSignal();
    }
#endif
}
```

OSTmrStart()

```
BOOLEAN  OSTmrStart(OS_TMR    *ptmr,
                    INT8U     *perr);
```

Chapter	File	Called from	Code enabled by
New in V2.81	OS_TMR.C	Task	OS_TMR_EN

`OSTmrStart()` allows you to start (or restart) the countdown process of a timer. The timer to start **MUST** have previously been created.

Arguments

`ptmr` is a pointer to the timer that you want to start (or restart). This pointer is returned to you when the timer is created (see `OSTmrCreate()`).

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

`OS_ERR_NONE`	If the timer was started.
`OS_ERR_TMR_INVALID`	If you passed a `NULL` pointer for the `ptmr` argument.
`OS_ERR_TMR_INVALID_TYPE`	'`ptmr`' is not pointing to a timer.
`OS_ERR_TMR_ISR`	You called this function from an ISR which is NOT allowed.
`OS_ERR_TMR_INACTIVE`	`ptmr` is pointing to an inactive timer. In other words, you would get this error if you are pointing to a timer that has been deleted or was not created.
`OS_ERR_TMR_INVALID_STATE`	The timer is in an invalid state.

Returned Values

`OS_TRUE` if the timer was started
`OS_FALSE` if an error occurred.

Notes/Warnings

1. You should examine the return value to make sure what you get from this function is valid.
2. You **MUST NOT** call this function from an ISR.
3. The timer to start **MUST** have previously been created.

Example

```
OS_TMR  *CloseDoorTmr;
BOOLEAN  status;



void Task (void *p_arg)
{
    INT8U     err;


    (void)p_arg;
    for (;;) {
        status = OSTmrStart(CloseDoorTmr,
                            &err);
        if (err == OS_ERR_NONE) {
            /* Timer was started */
        }
    }
}
```

OSTmrStateGet()

```
INT8U OSTmrStateGet(OS_TMR *ptmr,
                    INT8U  *perr);
```

Chapter	File	Called from	Code enabled by
New in V2.83	OS_TMR.C	Task	OS_TMR_EN

`OSTmrStateGet()` allows you to obtain the current state of a timer. A timer can be in one of 4 states:

`OS_TMR_STATE_UNUSED`	The timer has not been created
`OS_TMR_STATE_STOPPED`	The timer has been created but has not been started or has been stopped.
`OS_TMR_STATE_COMPLETED`	The timer is in ONE-SHOT mode and has completed its delay.
`OS_TMR_STATE_RUNNING`	The timer is currently running

Arguments

`ptmr` is a pointer to the timer that you are inquiring about. This pointer is returned to you when the timer is created (see `OSTmrCreate()`).

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

`OS_ERR_NONE`	If the function returned the time remaining for the timer.
`OS_ERR_TMR_INVALID`	If you passed a `NULL` pointer for the `ptmr` argument.
`OS_ERR_TMR_INVALID_TYPE`	'`ptmr`' is not pointing to a timer.
`OS_ERR_TMR_ISR`	You called this function from an ISR which is NOT allowed.
`OS_ERR_TMR_INACTIVE`	`ptmr` is pointing to an inactive timer. In other words, you would get this error if you are pointing to a timer that has been deleted or was not created.
`OS_ERR_TMR_INVALID_STATE`	The timer is in an invalid state.

Returned Values

The state of the timer (see description).

Notes/Warnings

1. You should examine the return value to make sure what you get from this function is valid.
2. You **MUST NOT** call this function from an ISR.

Example

```
INT8U     CloseDoorTmrState;
OS_TMR   *CloseDoorTmr;

void Task (void *p_arg)
{
    INT8U     err;


    (void)p_arg;
    for (;;) {
        CloseDoorTmrState = OSTmrStateGet(CloseDoorTmr, &err);
        if (err == OS_ERR_NONE) {
            /* Call was successful */
        }
    }
}
```

OSTmrStop()

```
BOOLEAN  OSTmrStop(OS_TMR  *ptmr,
                   INT8U    opt,
                   void    *callback_arg,
                   INT8U   *perr);
```

Chapter	File	Called from	Code enabled by
New in V2.81	`OS_TMR.C`	Task	`OS_TMR_EN`

`OSTmrStop()` allows you to stop (i.e. abort) a timer. You can execute the callback function of the timer when it's stopped and pass this callback function a different argument than what was specified when the timer was started. This allows your callback function to *know* that the timer was stopped because the callback argument can be made to indicate this (this, of course, is application specific). If the timer is already stopped, the callback function is not called.

Arguments

`ptmr` Is a pointer to the timer you want to stop. This 'handle' was returned to your application when you called `OSTmrStart()` and uniquely identifies the timer.

`opt` specifies whether you want the timer to:

1) `OS_TMR_OPT_NONE`:
   Do NOT call the callback function.
2) `OS_TMR_OPT_CALLBACK`:
   Call the callback function and pass it the callback argument specified when you started the timer (see `OSTmrCreate()`).
3) `OS_TMR_OPT_CALLBACK_ARG`:
   Call the callback function BUT pass it the callback argument specified in the `OSTmrStop()` function INSTEAD of the one defined in `OSTmrCreate()`.

`callback_arg` If you set `opt` to `OS_TMR_OPT_CALLBACK_ARG` then this is the argument passed to the callback function when it's executed.

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

`OS_ERR_NONE`	If the timer was started.
`OS_ERR_TMR_INVALID`	If you passed a `NULL` pointer for the `ptmr` argument.
`OS_ERR_TMR_INVALID_TYPE`	'`ptmr`' is not pointing to a timer.
`OS_ERR_TMR_ISR`	You called this function from an ISR which is NOT allowed.
`OS_ERR_TMR_INVALID_OPT`	You specified an invalid option for '`opt`'.
`OS_ERR_TMR_STOPPED`	The timer was already stopped. However, this is NOT considered an actual error since it's OK to attempt to stop a timer that is already stopped.
`OS_ERR_TMR_INACTIVE`	`ptmr` is pointing to an inactive timer. In other words, you would get this error if you are pointing to a timer that has been deleted or was not created.
`OS_ERR_TMR_INVALID_STATE`	The timer is in an invalid state.
`OS_ERR_TMR_NO_CALLBACK`	If you wanted the callback to be called but no callback has been specified for this timer.

Returned Values

`OS_TRUE`	if the timer was stopped (even if it was already stopped).
`OS_FALSE`	if an error occurred.

Notes/Warnings

1. You should examine the return value to make sure what you get from this function is valid.
2. You **MUST NOT** call this function from an ISR.
3. The callback function is **NOT** called if the timer is already stopped.

Example

```
OS_TMR  *CloseDoorTmr;



void Task (void *p_arg)
{
    INT8U     err;


    (void)p_arg;
    for (;;) {
        OSTmrStop(CloseDoorTmr,
                  OS_TMR_OPT_CALLBACK,
                  (void *)0,
                  &err);
        if (err == OS_ERR_NONE || err == OS_ERR_TMR_STOPPED) {
            /* Timer was stopped ...                             */
            /* ... callback was called only if timer was running */
        }
    }
}
```

OSVersion()

`INT16U OSVersion(void);`

Chapter	File	Called from	Code enabled by
3	OS_CORE.C	Task or ISR	N/A

`OSVersion()` obtains the current version of µC/OS-II.

Arguments

none

Returned Value

The version is returned as *x.yy* multiplied by 10000. For example, v2.92.07 is returned as 29207.

Notes/Warnings

none

Example

```
void TaskX (void *p_arg)
{
      INT16U os_version;



      for (;;) {
         .
         .
         os_version = OSVersion();  /* Obtain µC/OS-II's version   */
         .
         .
      }
}
```