µC/OS-III™

The Real-Time Kernel

API Reference

Jean J. Labrosse

Micriµm Press

Weston, FL 33326

µC/OS-III User's Manual

uC-OS-III API Reference

- API - Event Flags
- API - Mutual Exclusion Semaphores
- API - Time Management
- API - Thread Local Storage
- API - Message Queues
- API - Semaphores
- API - Timers
- API - Task Semaphores
- API - Fixed-Size Memory Partitions
- API - Task Management
- API - Miscellaneous
- API - Task Message Queues
- API - Monitors

API - Event Flags

- OSFlagCreate()
- OSFlagDel()
- OSFlagPend()
- OSFlagPendAbort()
- OSFlagPendGetFlagsRdy()
- OSFlagPost()

OSFlagCreate

Description

Creates and initialize an event flag group. µC/OS-III allows the user to create an unlimited number of event flag groups (limited only by the amount of RAM in the system).

Files

os.h/os_flag.c

Prototype

```
void  OSFlagCreate (OS_FLAG_GRP  *p_grp,
                    CPU_CHAR     *p_name,
                    OS_FLAGS      flags,
                    OS_ERR       *p_err)
```

Arguments

`p_grp`

This is a pointer to an event flag group that must be allocated in the application. The user will need to declare a “global” variable as shown, and pass a pointer to this variable to `OSFlagCreate()`:

```
OS_FLAG_GRP MyEventFlag;
```

`p_name`

This is a pointer to an ASCII string used for the name of the event flag group. The name can be displayed by debuggers or by µC/Probe.

`flags`

`This` contains the initial value of the flags to store in the event flag group. Typically, you would set all flags to `0` events correspond to set bits and all `1`s if events correspond to cleared bits.

`p_err`

`This` 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 `OS_CFG_CALLED_FROM_ISR_CHK_EN` set to `DEF_ENABLED` in `os_cfg.h`: If attempting to create an event flag group from an ISR, w is not allowed.

`OS_ERR_ILLEGAL_CREATE_RUN_TIME`

If `OS_SAFETY_CRITICAL_IEC61508` is defined: you called this after calling `OSStart()` and thus you are no longer allowed to create additional kernel objects.

`OS_ERR_OBJ_PTR_NULL`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: If `p_grp` is a `NULL` pointer.

Returned Values

None

Required Configuration

`OS_CFG_FLAG_EN` must be enabled in `os_cfg.h`. Refer to µC/OS-III Configuration Manual.

Callers

Application.

Notes/Warnings

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

Example Usage

```
OS_FLAG_GRP  EngineStatus;


          void main (void)
          {
              OS_ERR  err;

              OSInit(&err);                    /* Initialize µC/OS-III
*/
              :
              :
              OSFlagCreate(&EngineStatus,
                           “Engine Status”,
                           (OS_FLAGS)0,
                           &err);              /* Create a flag grp containing the engine’s
status */
              /* Check “err” */
              :
              :
              OSStart();                       /* Start Multitasking
*/
```

Listing - OSFlagCreate() example usage

OSFlagDel

Description

Deletes an event flag group. This function should be used with care since multiple tasks may be relying on the presence of the event flag group. Generally, before deleting an event flag group, first delete all of the tasks that access the event flag group. Also, it is recommended that the user not delete kernel objects at run time.

Files

os.h/os_flag.c

Prototype

```
OS_OBJ_QTY  OSFlagDel (OS_FLAG_GRP  *p_grp,
                       OS_OPT        opt,
                       OS_ERR       *p_err);
```

Arguments

`p_grp`

is a pointer to the event flag group to delete.

`opt`

specifies whether the user wants to delete the event flag group only if there are no pending tasks (`OS_OPT_DEL_NO_PEND`), or whether the event flag group should always be deleted regardless of whether or not tasks are pending (`OS_OPT_DEL_ALWAYS`). In this case, all pending task are readied.

`p_err`

is a pointer to a variable 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 OS_CFG_CALLED_FROM_ISR_CHK_EN set to DEF_ENABLED in os_cfg.h: if the user attempts to delete an event flag group from an ISR.

OS_ERR_ILLEGAL_DEL_RUN_TIME

If OS_SAFETY_CRITICAL_IEC61508 is defined: you called this after calling OSStart() and thus you are no longer allowed to delete kernel objects.

OS_ERR_OBJ_PTR_NULL

If OS_CFG_ARG_CHK_EN is set to DEF_ENABLED in os_cfg.h: if p_grp is a NULL pointer.

OS_ERR_OBJ_TYPE

If OS_CFG_OBJ_TYPE_CHK_EN is set to DEF_ENABLED in os_cfg.h: if p_grp is not pointing to an event flag group.

OS_ERR_OPT_INVALID

If OS_CFG_ARG_CHK_EN is set to DEF_ENABLED in os_cfg.h: if the user does not specify one of the options mentioned in the opt argument.

OS_ERR_OS_NOT_RUNNING

If OS_CFG_INVALID_OS_CALLS_CHK_EN is set to DEF_ENABLED in os_cfg.h: if µC/OS-III is not running yet.

OS_ERR_TASK_WAITING

If one or more tasks are waiting on the event flag group and OS_OPT_DEL_NO_PEND is specified.

Returned Values

0 if no task was waiting on the event flag group, or an error occurs.

> 0 if one or more tasks waiting on the event flag group are now readied and informed

Required Configuration

`OS_CFG_FLAG_EN` and `OS_CFG_FLAG_DEL_EN` must be enabled in `os_cfg.h`. Refer to µC-OS-III Configuration Manual.

Callers

Application.

Notes/Warnings

1. You should use this call with care as other tasks might expect the presence of the event flag group.

Example Usage

```
OS_FLAG_GRP  EngineStatusFlags;


          void Task (void *p_arg)
          {
              OS_ERR      err;
              OS_OBJ_QTY  qty;

              (void)&p_arg;
              while (DEF_ON) {
                  :
                  :
                  qty = OSFlagDel(&EngineStatusFlags,
                                  OS_OPT_DEL_ALWAYS,
                                  &err);
                  /* Check “err” */
                  :
                  :
              }
          }
```

Listing - OSFlagDel() example usage

OSFlagPend

Description

Wait for a combination of conditions or events (i.e. bits) to be set (or cleared) in an event flag group. The 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, the calling task is blocked (optional) until the desired conditions or events are satisfied, the specified timeout expires, the event flag is deleted, or the pend is aborted by another task.

Files

os.h/os_flag.c

Prototype

```
OS_FLAGS  OSFlagPend (OS_FLAG_GRP  *p_grp,
                      OS_FLAGS      flags,
                      OS_TICK       timeout,
                      OS_OPT        opt,
                      CPU_TS       *p_ts,
                      OS_ERR       *p_err)
```

Arguments

`p_grp`

is a pointer to the event flag group.

`flags`

is a bit pattern indicating which bit(s) (i.e., flags) to check. The bits wanted are specified by setting the corresponding bits in flags. If the application wants to wait for bits 0 and 1 to be set, specify 0x03. The same applies if you'd want to wait for the same 2 bits to be cleared (you'd still specify which bits by passing `0x03`).

`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.

`opt`

specifies whether all bits are to be set/cleared or any of the bits are to be set/cleared. Here are the options:

`OS_OPT_PEND_FLAG_CLR_ALL`

If `OS_CFG_FLAG_MODE_CLR_EN` is set to `DEF_ENABLED` in `os_cfg.h`, check all bits in flags to be clear (0)

`OS_OPT_PEND_FLAG_CLR_ANY`

If `OS_CFG_FLAG_MODE_CLR_EN` is set to `DEF_ENABLED` in `os_cfg.h`, check any bit in flags to be clear (0)

`OS_OPT_PEND_FLAG_SET_ALL`

Check all bits in flags to be set (1)

`OS_OPT_PEND_FLAG_SET_ANY`

Check any bit in flags to be set (1)

The caller may also specify whether the flags are consumed by "adding" `OS_OPT_PEND_FLAG_CONSUME` to the `opt` argument. For example, to wait for any flag in a group and then clear the flags that satisfy the condition, you would set `opt` to:

`OS_OPT_PEND_FLAG_SET_ANY + OS_OPT_PEND_FLAG_CONSUME`

Finally, you can specify whether you want the caller to block if the flag(s) are available or not. You would then "add" the following options:

`OS_OPT_PEND_BLOCKING`

`OS_OPT_PEND_NON_BLOCKING`

Note that the `timeout` argument should be set to `0` when specifying `OS_OPT_PEND_NON_BLOCKING`, since the timeout value is irrelevant using this option. Having a non-zero value could simply confuse the reader of your code.

`p_ts`

is a pointer to a timestamp indicating when the flags were posted, the pend was aborted, or the event flag group was deleted. Passing a `NULL` pointer (i.e., `(CPU_TS *)0`) indicates that the caller does not desire the timestamp. In other words, passing a `NULL` pointer is valid, and indicates that the caller does not need the timestamp.

A timestamp is useful when the task desires to know when the event flag group was posted or how long it took for the task to resume after the event flag group was posted. In the latter case, the user must call `OS_TS_GET()` and compute the difference between the current value of the timestamp and `*p_ts`, as shown:

```
delta = OS_TS_GET() - *p_ts;
```

`p_err`

is a pointer to an error code and can be:

`OS_ERR_NONE`

No error.

`OS_ERR_OBJ_DEL`

If the event group was deleted.

`OS_ERR_OBJ_PTR_NULL`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if `p_grp` is a `NULL` pointer.

`OS_ERR_OBJ_TYPE`

If `OS_CFG_OBJ_TYPE_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: `p_grp` is not pointing to an event flag group.

`OS_ERR_OPT_INVALID`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: the caller specified an invalid option.

`OS_ERR_OS_NOT_RUNNING`

If `OS_CFG_INVALID_OS_CALLS_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if µC/OS-III is not running yet.

`OS_ERR_PEND_ABORT`

The wait on the flags was aborted by another task that called `OSFlagPendAbort()`.

`OS_ERR_PEND_ISR`

If `OS_CFG_CALLED_FROM_ISR_CHK_EN` set to `DEF_ENABLED` in `os_cfg.h`: An attempt was made to call `OSFlagPend()` from an ISR, which is not allowed.

`OS_ERR_PEND_WOULD_BLOCK`

If specifying non-blocking but the flags were not available and the call would block if the caller had specified `OS_OPT_PEND_BLOCKING`.

`OS_ERR_SCHED_LOCKED`

When calling this function while the scheduler was locked.

`OS_ERR_STATUS_INVALID`

If the pend status has an invalid value.

`OS_ERR_TIMEOUT`

The flags are not available within the specified amount of time.

Returned Values

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

Required Configuration

`OS_CFG_FLAG_EN` must be enabled, and optionally `OS_CFG_FLAG_MODE_CLR_EN`, in `os_cfg.h`. Refer to µC-OS-III Configuration Manual.

Callers

Application.

Notes/Warnings

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

Example Usage

```
#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)
          {
              OS_ERR    err;
              OS_FLAGS  value;
              CPU_TS    ts;


              (void)&p_arg;
              while (DEF_ON) {
                  value = OSFlagPend(&EngineStatus,
                                     ENGINE_OIL_PRES_OK   + ENGINE_OIL_TEMP_OK,
                                     OS_FLAG_WAIT_SET_ALL + OS_FLAG_CONSUME,
                                     10,
                                     OS_OPT_PEND_BLOCKING,
                                     &ts,
                                     &err);
                  /* Check “err” */
                  :
                  :
              }
          }
```

Listing - OSFlagPend() example usage

OSFlagPendAbort

Description

Aborts and readies any tasks currently waiting on an event flag group. This function would be used by another task to fault abort the wait on the event flag group, rather than to normally signal the event flag group via `OSFlagPost()`.

Files

os.h/os_flag.c

Prototype

```
OS_OBJ_QTY  OSFlagPendAbort (OS_SEM  *p_grp,
                             OS_OPT   opt,
                             OS_ERR  *p_err)
```

Arguments

`p_grp`

is a pointer to the event flag group for which pend(s) must be aborted.

`opt`

determines the type of abort performed.

`OS_OPT_PEND_ABORT_1`

Aborts the pend of only the highest priority task waiting on the event flag group.

`OS_OPT_PEND_ABORT_ALL`

Aborts the pend of all the tasks waiting on the event flag group.

`OS_OPT_POST_NO_SCHED`

Specifies that the scheduler should not be called even if the pend of a higher priority task is aborted. Scheduling will need to occur from another function.

You would use this option if the task calling `OSFlagPendAbort()` will perform additional pend aborts, rescheduling will take place at completion, and when multiple pend aborts are to take effect simultaneously.

`p_err`

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

`OS_ERR_NONE`

at least one task waiting on the event flag group was readied and informed of the aborted wait. The return value indicates the number of tasks where a wait on the event flag group was aborted.

`OS_ERR_OBJ_PTR_NULL`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if `p_grp` is a `NULL` pointer.

`OS_ERR_OBJ_TYPE`

If `OS_CFG_OBJ_TYPE_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if `p_grp` is not pointing to an event flag group.

`OS_ERR_OPT_INVALID`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if specifying an invalid option.

`OS_ERR_OS_NOT_RUNNING`

If `OS_CFG_INVALID_OS_CALLS_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if µ C/OS-III is not running yet.

`OS_ERR_PEND_ABORT_ISR`

If `OS_CFG_CALLED_FROM_ISR_CHK_EN` set to `DEF_ENABLED` in `os_cfg.h`: This function cannot be called from an ISR.

`OS_ERR_PEND_ABORT_NONE`

No task was aborted since no task was waiting.

Returned Value

`OSFlagPendAbort()` returns the number of tasks made ready-to-run by this function. Zero indicates that no tasks were pending on the event flag group and thus this function had no effect.

Required Configuration

`OS_CFG_FLAG_EN` and `OS_CFG_FLAG_PEND_ABORT_EN` must be enabled in `os_cfg.h`. Refer to µC-OS-III Configuration Manual.

Callers

Application.

Notes/Warnings

1. Event flag groups must be created before they are used.

Example Usage

```
OS_FLAG_GRP  EngineStatus;


          void Task (void *p_arg)
          {
              OS_ERR      err;
              OS_OBJ_QTY  nbr_tasks;


              (void)&p_arg;
              while (DEF_ON) {
                  :
                  :
                  nbr_tasks = OSFlagPendAbort(&EngineStatus,
                                              OS_OPT_PEND_ABORT_ALL,
                                              &err);
                  /* Check “err” */
                  :
                  :
              }
          }
```

Listing - OSFlagPendAbort() example usage

OSFlagPendGetFlagsRdy

Description

Returns the flags that caused the current task to be ready-to-run. This function allows the user to know “Who did it!”

Files

os.h/os_flag.c

Prototype

```
OS_FLAGS  OSFlagPendGetFlagsRdy (OS_ERR  *p_err)
```

Arguments

`p_err`

is a pointer to an error code and can be:

`OS_ERR_NONE`

No error.

`OS_ERR_OS_NOT_RUNNING`

If `OS_CFG_INVALID_OS_CALLS_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if µC/OS-III is not running yet.

`OS_ERR_PEND_ISR`

If `OS_CFG_CALLED_FROM_ISR_CHK_EN` set to `DEF_ENABLED` in `os_cfg.h`: When attempting to call this function from an ISR.

Returned Value

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

Required Configuration

`OS_CFG_FLAG_EN` must be enabled in `os_cfg.h`. Refer to µC-OS-III Configuration Manual.

Callers

Application.

Notes/Warnings

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

Example Usage

```
#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)
          {
              OS_ERR    err;
              OS_FLAGS  value;
              OS_FLAGS  flags_rdy;


              (void)&p_arg;
              while (DEF_ON) {
                  value     = OSFlagPend(&EngineStatus,
                                         ENGINE_OIL_PRES_OK   + ENGINE_OIL_TEMP_OK,
                                         OS_FLAG_WAIT_SET_ALL + OS_FLAG_CONSUME,
                                         10,
                                         &err);
                  /* Check “err” */
                  flags_rdy = OSFlagPendGetFlagsRdy(&err);
                  /* Check “err” */
                  :
                  :
             }
          }
```

Listing - OSFlagPendGetFlagsRdy() example usage

OSFlagPost

Description

Sets or clears event flag bits. The bits set or cleared are specified in a bit mask (i.e., the flags argument). `OSFlagPost()` readies each task that has its desired bits satisfied by this call. The caller can set or clear bits that are already set or cleared.

Files

os.h/os_flag.c

Prototype

```
OS_FLAGS  OSFlagPost (OS_FLAG_GRP  *p_grp,
                      OS_FLAGS      flags,
                      OS_OPT        opt,
                      OS_ERR       *p_err)
```

Arguments

`p_grp`

is a pointer to the event flag group.

`flags`

specifies which bits to be set or cleared. If `opt` is `OS_OPT_POST_FLAG_SET`, each bit that is set in `flags` will set the corresponding bit in the event flag group. For example to set bits 0, 4, and 5, you would set `flags` to `0x31` (note that bit 0 is the least significant bit). If opt is `OS_OPT_POST_FLAG_CLR`, each bit that is set in `flags` will clear the corresponding bit in the event flag group. For example to clear bits 0, 4, and 5, you would specify `flags` as `0x31` (again, bit 0 is the least significant bit).

`opt`

indicates whether the flags are set (`OS_OPT_POST_FLAG_SET`) or cleared (`OS_OPT_POST_FLAG_CLR`).

The caller may also “add” `OS_OPT_POST_NO_SCHED` so that µC/OS-III will not call the scheduler after the post.

`p_err`

is a pointer to an error code and can be:

`OS_ERR_NONE`

the call is successful.

`OS_ERR_OBJ_PTR_NULL`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if the caller passed a `NULL` pointer.

`OS_ERR_OBJ_TYPE`

If `OS_CFG_OBJ_TYPE_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: `p_grp` is not pointing to an event flag group.

`OS_ERR_OPT_INVALID`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if you specified an invalid option.

`OS_ERR_OS_NOT_RUNNING`

If `OS_CFG_INVALID_OS_CALLS_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if µC/OS-III is not running yet.

Returned Value

The new value of the event flags.

Required Configuration

`OS_CFG_FLAG_EN` must be enabled in `os_cfg.h`. Refer to µC-OS-III Configuration Manual.

Callers

Application and ISRs.

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 still deterministic.

3. Although the example below shows that we are posting from a task, `OSFlagPost()` can also be called from an ISR.

Example Usage

```
#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)
          {
              OS_ERR    err;
              OS_FLAGS  flags;


              (void)&p_arg;
              while (DEF_ON) {
                  :
                  :
                  flags = OSFlagPost(&EngineStatusFlags,
                                     ENGINE_START,
                                     OS_OPT_POST_FLAG_SET,
                                     &err);
                  /* Check 'err" */
                  :
                  :
              }
          }
```

Listing - OSFlagPost() example usage

API - Mutual Exclusion Semaphores

OSMutexCreate

Description

Create and initialize a mutex. A mutex is used to gain exclusive access to a resource.

Files

os.h/os_mutex.c

Prototype

```
void  OSMutexCreate (OS_MUTEX  *p_mutex,
                     CPU_CHAR  *p_name,
                     OS_ERR    *p_err)
```

Arguments

`p_mutex`

is a pointer to a mutex control block that must be allocated in the application. The user will need to declare a “global” variable as follows, and pass a pointer to this variable to `OSMutexCreate()`:

```
OS_MUTEX  MyMutex;
```

`p_name`

is a pointer to an ASCII string used to assign a name to the mutual exclusion semaphore. The name may be displayed by debuggers or μC/Probe.

`p_err`

is a pointer to a variable that is used to hold an error code:

`OS_ERR_NONE`

If the call is successful and the mutex has been created.

`OS_ERR_CREATE_ISR`

If `OS_CFG_CALLED_FROM_ISR_CHK_EN` set to `DEF_ENABLED` in `os_cfg.h`: if attempting to create a mutex from an ISR.

`OS_ERR_ILLEGAL_CREATE_RUN_TIME`

If `OS_SAFETY_CRITICAL_IEC61508` is defined: you called this after calling `OSStart()` and thus you are no longer allowed to create additional kernel objects.

`OS_ERR_OBJ_PTR_NULL`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if `p_mutex` is a `NULL` pointer.

Returned Value

None

Required Configuration

`OS_CFG_MUTEX_EN` must be enabled in `os_cfg.h`. Refer to µC-OS-III Configuration Manual.

Callers

Application.

Notes/Warnings

1. Mutexes must be created before they are used.

Example Usage

```
OS_MUTEX  DispMutex;


          void main (void)
          {
              OS_ERR  err;


              :
              OSInit(&err);                         /* Initialize µC/OS-III         */
              :
              :
              OSMutexCreate(&DispMutex,             /* Create Display Mutex         */
                            “Display Mutex”,
                            &err);
              /* Check “err” */

              :
              :
              OSStart(&err);                        /* Start Multitasking           */
          }
```

Listing - OSMutexCreate() example usage

OSMutexDel

Description

Deletes a mutex. This function makes all pending tasks ready to run and clears the mutex data. It also sets the owning task's priority to the highest priority in the owner's task mutex group or its base priority, whichever is higher. Generally speaking, before deleting a mutex, first delete all the tasks that access the mutex. However, as a guideline, do not delete kernel objects at run-time.

Files

os.h/os_mutex.c

Prototype

```
OS_OBJ_QTY  OSMutexDel (OS_MUTEX  *p_mutex,
                        OS_OPT     opt,
                        OS_ERR    *p_err)
```

Arguments

`p_mutex`

is a pointer to the mutex to delete.

`opt`

specifies whether to delete the mutex only if there are no pending tasks (`OS_OPT_DEL_NO_PEND`), or whether to always delete the mutex regardless of whether tasks are pending or not (`OS_OPT_DEL_ALWAYS`). In this case, all pending tasks are readied.

`p_err`

is a pointer to a variable that is used to hold an error code:

`OS_ERR_NONE`

If the call is successful and the mutex has been deleted.

`OS_ERR_DEL_ISR`

If `OS_CFG_CALLED_FROM_ISR_CHK_EN` set to `DEF_ENABLED` in `os_cfg.h`: if attempting to delete a mutex from an ISR.

`OS_ERR_ILLEGAL_DEL_RUN_TIME`

If `OS_SAFETY_CRITICAL_IEC61508` is defined: you are trying to delete the mutex after you called `OSStart()`.

`OS_ERR_OBJ_PTR_NULL`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if `p_mutex` is a `NULL` pointer.

`OS_ERR_OBJ_TYPE`

If `OS_CFG_OBJ_TYPE_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if `p_mutex` is not pointing to a mutex.

`OS_ERR_OPT_INVALID`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if a valid option is not specified.

`OS_ERR_OS_NOT_RUNNING`

If `OS_CFG_INVALID_OS_CALLS_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: If µC/OS-III is not running yet.

`OS_ERR_TASK_WAITING`

If one or more task are waiting on the mutex and `OS_OPT_DEL_NO_PEND` is specified.

Returned Value

The number of tasks that were waiting for the mutex. Zero either indicates an error or that no tasks were pending on the mutex.

Required Configuration

`OS_CFG_MUTEX_EN` and `OS_CFG_MUTEX_DEL_EN` must be enabled in `os_cfg.h`. Refer to µC-OS-III Configuration Manual.

Callers

Application.

Notes/Warnings

1. Use this call with care as other tasks may expect the presence of the mutex.

Example Usage

```
OS_MUTEX  DispMutex;


          void Task (void *p_arg)
          {
              OS_ERR      err;
              OS_OBJ_QTY  qty;


              (void)&p_arg;
              while (DEF_ON) {
                  :
                  :
                  qty = OSMutexDel(&DispMutex,
                                    OS_OPT_DEL_ALWAYS,
                                   &err);
                  /* Check "err" */
                  :
                  :
              }
          }
```

Listing - OSMutexDel() example usage

OSMutexPend

Description

Acquire a mutual exclusion semaphore. If a task calls `OSMutexPend()` and the mutex is available, `OSMutexPend()` gives the mutex to the caller and returns to its caller. Note that nothing is actually given to the caller except that if `p_err` 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 waits until the task that owns the mutex releases the mutex and therefore the resource, or until the specified timeout expires. If the mutex is signaled before the timeout expires, µC/OS-III resumes the highest-priority task that is waiting for the mutex. Note that if the mutex is owned by a lower-priority task, `OSMutexPend()` raises the priority of the task that owns the mutex to the same priority as the task requesting the mutex. The priority of the owning task will be set to the highest priority in the owning task's mutex group or its base priority, whichever is higher, when the owner releases the mutex (see `OSMutexPost()`). `OSMutexPend()` allows nesting. The same task can call `OSMutexPend()` multiple times. However, the same task must then call `OSMutexPost()` an equal number of times to release the mutex.

Files

os.h/os_mutex.c

Prototype

```
void  OSMutexPend (OS_MUTEX  *p_mutex,
                   OS_TICK    timeout,
                   OS_OPT     opt,
                   CPU_TS    *p_ts,
                   OS_ERR    *p_err)
```

Arguments

`p_mutex`

is a pointer to the mutex.

`timeout`

specifies a timeout value (in clock ticks) and is used to allow the task to resume execution if the mutex is not signaled (i.e., posted to) within the specified timeout. A timeout value of 0 indicates that the task wants to wait forever for the mutex. The timeout value is not synchronized with the clock tick. The timeout count is decremented on the next clock tick, which could potentially occur immediately.

`opt`

determines whether the user wants to block if the mutex is not available or not. This argument must be set to either:

`OS_OPT_PEND_BLOCKING`, or
`OS_OPT_PEND_NON_BLOCKING`

Note that the timeout argument should be set to 0 when specifying `OS_OPT_PEND_NON_BLOCKING` since the timeout value is irrelevant using this option.

`p_ts`

is a pointer to a timestamp indicating when the mutex was posted, the pend was aborted, or the mutex was deleted. If passing a `NULL` pointer (i.e., (`CPU_TS` *)0), the caller will not receive the timestamp. In other words, passing a `NULL` pointer is valid and indicates that the timestamp is not required.

A timestamp is useful when it is important for a task to know when the mutex was posted, or how long it took for the task to resume after the mutex was posted. In the latter case, the user must call `OS_TS_GET()` and compute the difference between the current value of the timestamp and `*p_ts`. In other words:

```
delta = OS_TS_GET() - *p_ts;
```

`p_err`

is a pointer to a variable that is used to hold an error code:

`OS_ERR_NONE`

If the call is successful and the mutex is available.

`OS_ERR_MUTEX_OWNER`

If the calling task already owns the mutex.

`OS_ERR_MUTEX_OVF`

The mutex nesting counter overflowed.

`OS_ERR_OBJ_DEL`

If the mutex was deleted.

`OS_ERR_OBJ_PTR_NULL`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if `p_mutex` is a `NULL` pointer.

`OS_ERR_OBJ_TYPE`

If `OS_CFG_OBJ_TYPE_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if the user did not pass a pointer to a mutex.

`OS_ERR_OPT_INVALID`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if a valid option is not specified.

`OS_ERR_OS_NOT_RUNNING`

If `OS_CFG_INVALID_OS_CALLS_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if µ C/OS-III is not running yet.

`OS_ERR_PEND_ABORT`

If `OS_CFG_MUTEX_PEND_ABORT_EN` is set to `DEF_ENABLED` in `os_cfg.h`: the pend was aborted by another task.

`OS_ERR_PEND_ISR`

If `OS_CFG_CALLED_FROM_ISR_CHK_EN` set to `DEF_ENABLED` in `os_cfg.h`: if attempting to acquire the mutex from an ISR.

`OS_ERR_PEND_WOULD_BLOCK`

If the mutex was not available and `OS_OPT_PEND_NON_BLOCKING` is specified.

`OS_ERR_SCHED_LOCKED`

If the scheduler is locked.

`OS_ERR_STATUS_INVALID`

If the pend status has an invalid value.

`OS_ERR_TIMEOUT`

If the mutex was not available within the specified timeout.

Returned Value

None

Required Configuration

`OS_CFG_MUTEX_EN` must be enabled in `os_cfg.h`. Refer to µC-OS-III Configuration Manual.

Callers

Application.

Notes/Warnings

1. Mutexes must be created before they are used.

Example Usage

```
OS_MUTEX  DispMutex;


          void  DispTask (void *p_arg)
          {
              OS_ERR  err;
              CPU_TS  ts;


              (void)&p_arg;
              while (DEF_ON) {
                  :
                  OSMutexPend(&DispMutex,
                               0,
                               OS_OPT_PEND_BLOCKING,
                              &ts,
                              &err);
                  /* Check “err” */
                  :
                  :
              }
          }
```

Listing - OSMutexCreate() example usage

OSMutexPendAbort

Description

Aborts and readies all the tasks currently pending on a mutex. This function should be used to fault-abort the wait on the mutex rather than to normally signal the mutex via `OSMutexPost()`.

Files

os.h/os_mutex.c

Prototype

```
OS_OBJ_QTY  OSMutexPendAbort (OS_MUTEX  *p_mutex,
                              OS_OPT     opt,
                              OS_ERR    *p_err)
```

Arguments

`p_mutex`

is a pointer to the mutex.

`opt`

specifies whether to abort only the highest-priority task waiting on the mutex or all tasks waiting on the mutex:

`OS_OPT_PEND_ABORT_1`

to abort only the highest-priority task waiting on the mutex.

`OS_OPT_PEND_ABORT_ALL`

to abort all tasks waiting on the mutex.

`OS_OPT_POST_NO_SCHED`

specifies that the scheduler should not be called even if the pend of a higher-priority task has been aborted. Scheduling will need to occur from another function.

The user would select this option if the task calling `OSMutexPendAbort()` will be doing additional pend aborts, rescheduling should not take place until all tasks are completed, and multiple pend aborts should take place simultaneously.

`p_err`

is a pointer to a variable that is used to hold an error code:

`OS_ERR_NONE`

If at least one task was aborted. Check the return value for the number of tasks aborted.

`OS_ERR_OBJ_PTR_NULL`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if `p_mutex` is a `NULL` pointer.

`OS_ERR_OBJ_TYPE`

If `OS_CFG_OBJ_TYPE_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if the caller does not pass a pointer to a mutex.

`OS_ERR_OPT_INVALID`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if the caller specified an invalid option.

`OS_ERR_OS_NOT_RUNNING`

If `OS_CFG_INVALID_OS_CALLS_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: If µC/OS-III is not running yet.

`OS_ERR_PEND_ABORT_ISR`

If `OS_CFG_CALLED_FROM_ISR_CHK_EN` set to `DEF_ENABLED` in `os_cfg.h`: if attempting to call this function from an ISR

`OS_ERR_PEND_ABORT_NONE`

If no tasks were aborted.

Returned Value

`OSMutexPendAbort()` returns the number of tasks made ready-to-run by this function. Zero either indicates an error or that no tasks were pending on the mutex.

Required Configuration

`OS_CFG_MUTEX_EN` and `OS_CFG_MUTEX_PEND_ABORT_EN` must be enabled in `os_cfg.h`. Refer to µC-OS-III Configuration Manual.

Callers

Application.

Notes/Warnings

1. Mutexes must be created before they are used.

Example Usage

```
OS_MUTEX  DispMutex;


          void  DispTask (void *p_arg)
          {
              OS_ERR      err;
              OS_OBJ_QTY  qty;


              (void)&p_arg;
              while (DEF_ON) {
                  :
                  :
                  qty = OSMutexPendAbort(&DispMutex,
                                          OS_OPT_PEND_ABORT_ALL,
                                         &err);
                  /* Check "err" */
                  :
                  :
              }
          }
```

Listing - OSMutexCreate() example usage

OSMutexPost

Description

A mutex is signaled (i.e., released) by calling `OSMutexPost()`. You should call this function only if you acquired the mutex by first calling `OSMutexPend()`. If the priority of the task that owns the mutex has been raised when a higher priority task attempted to acquire the mutex, the priority of the owning task will be set to the highest priority in the owning task's mutex group or its base priority, whichever is higher. 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 performed to run the readied task. If no task is waiting for the mutex, the mutex value is simply set to available.

Prototype

```
void  OSMutexPost (OS_MUTEX  *p_mutex,
                   OS_OPT     opt,
                   OS_ERR    *p_err);
```

Arguments

`p_mutex`

is a pointer to the mutex.

`opt`

determines the type of POST performed.

`OS_OPT_POST_NONE`

No special option selected.

`OS_OPT_POST_NO_SCHED`

Do not call the scheduler after the post, therefore the caller is resumed even if the mutex was posted and tasks of higher priority are waiting for the mutex.

Use this option if the task calling `OSMutexPost()` will be doing additional posts, if the user does not want to reschedule until all is complete, and multiple posts should take effect simultaneously.

`p_err`

is a pointer to a variable that is used to hold an error code:

`OS_ERR_NONE`

If the call is successful and the mutex is available.

`OS_ERR_MUTEX_NESTING`

If the owner of the mutex has the mutex nested and it has not fully un-nested.

`OS_ERR_MUTEX_NOT_OWNER`

If the caller is not the owner of the mutex and therefore is not allowed to release it.

`OS_ERR_OBJ_PTR_NULL`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if `p_mutex` is a `NULL` pointer.

`OS_ERR_OBJ_TYPE`

If `OS_CFG_OBJ_TYPE_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if not passing a pointer to a mutex.

`OS_ERR_OPT_INVALID`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if a valid option is not specified.

`OS_ERR_OS_NOT_RUNNING`

If `OS_CFG_INVALID_OS_CALLS_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if μ

C/OS-III is not running yet.

OS_ERR_POST_ISR

If OS_CFG_CALLED_FROM_ISR_CHK_EN set to DEF_ENABLED in os_cfg.h: if attempting to post the mutex from an ISR.

Returned Value

None

Required Configuration

OS_CFG_MUTEX_EN must be enabled in os_cfg.h. Refer to µC-OS-III Configuration Manual.

Callers

Application.

Notes/Warnings

1. Mutexes must be created before they are used.

Example Usage

```
OS_MUTEX  DispMutex;


          void  TaskX (void *p_arg)
          {
              OS_ERR  err;


              (void)&p_arg;
              while (DEF_ON) {
                  :
                  OSMutexPost(&DispMutex,
                               OS_OPT_POST_NONE,
                              &err);
                  /* Check “err” */
                  :
                  :
              }
          }
```

Listing - OSMutexCreate() example usage

API - Time Management

OSTimeDly

Description

Allows a task to delay itself for an integral number of clock ticks. The delay can either be relative (delay from current time), periodic (delay occurs at fixed intervals) or absolute (delay until we reach some time).

In relative mode, rescheduling always occurs when the number of clock ticks is greater than zero. A delay of `0` means that the task is not delayed, and `OSTimeDly()` returns immediately to the caller.

In periodic mode, you must specify a non-zero period otherwise the function returns immediately with an appropriate error code. The period is specified in “ticks”.

In absolute mode, rescheduling always occurs since all delay values are valid.

The actual delay time depends on the tick rate (see `OS_CFG_TICK_RATE_HZ` if `os_cfg_app.h`).

Files

os.h/os_time.c

Prototype

```
void  OSTimeDly (OS_TICK   dly,
                 OS_OPT    opt,
                 OS_ERR   *p_err)
```

Arguments

`dly`

> is the desired delay expressed in number of clock ticks. Depending on the value of the `opt` field, delays can be relative or absolute.
>
> A relative delay means that the delay is started from the “`current time + dly`”.

A periodic delay means the period (in number of ticks). µC/OS-III saves the current time + `dly` in `.TickCtrPrev` so the next time `OSTimeDly()` is called, we use `.TickDlyPrev + dly`.

An absolute delay means that the task will wake up when `OSTickCtr` reaches the value specified by `dly`.

`opt`

is used to indicate whether the delay is absolute or relative:

`OS_OPT_TIME_DLY`

Specifies a relative delay.

`OS_OPT_TIME_TIMEOUT`

Same as `OS_OPT_TIME_DLY`.

`OS_OPT_TIME_PERIODIC`

Specifies periodic mode.

`OS_OPT_TIME_MATCH`

Specifies that the task will wake up when `OSTickCtr` reaches the value specified by `dly`

`p_err`

is a pointer to a variable that will contain an error code returned by this function.

`OS_ERR_NONE`

If the call was successful, and the task has returned from the desired delay.

`OS_ERR_OPT_INVALID`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if a valid option is not specified.

`OS_ERR_OS_NOT_RUNNING`

If `OS_CFG_INVALID_OS_CALLS_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if µC/OS-III is not running yet.

`OS_ERR_SCHED_LOCKED`

If the scheduler is locked.

`OS_ERR_TIME_DLY_ISR`

If `OS_CFG_CALLED_FROM_ISR_CHK_EN` set to `DEF_ENABLED` in `os_cfg.h`: if calling this function from an ISR.

`OS_ERR_TIME_ZERO_DLY`

If specifying a delay of 0 when the option was set to `OS_OPT_TIME_DLY`. Note that a value of 0 is valid when setting the option to `OS_OPT_TIME_MATCH`.

Returned Value

None

Required Configuration

None

Callers

Application.

Notes/Warnings

None

Example Usage

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


            while (DEF_ON) {
              :
              :
              OSTimeDly(10,
                        OS_OPT_TIME_PERIODIC,
                        &err);
              /* Check “err” */
              :
              :
            }
          }
```

Listing - OSTimeDly() example usage

OSTimeDlyHMSM

Description

Allows a task to delay itself for a user-specified period that is specified in hours, minutes, seconds, and milliseconds. This format is more convenient and natural than simply specifying ticks as in `OSTimeDly()`. Rescheduling always occurs when at least one of the parameters is non-zero. The delay is relative from the time this function is called.

µC/OS-III allows the user to specify nearly any value when indicating that this function is not to be strict about the values being passed (`opt == OS_OPT_TIME_HMSM_NON_STRICT`). This is a useful feature, for example, to delay a task for thousands of milliseconds.

Files

os.h/os_time.c

Prototype

```
void  OSTimeDlyHMSM (CPU_INT16U   hours,
                     CPU_INT16U   minutes,
                     CPU_INT16U   seconds,
                     CPU_INT32U   milli,
                     OS_OPT       opt,
                     OS_ERR      *p_err)
```

Arguments

`hours`

is the number of hours the task is delayed. Depending on the `opt` value, the valid range is 0..99 (`OS_OPT_TIME_HMSM_STRICT`), or 0..999 (`OS_OPT_TIME_HMSM_NON_STRICT`). Please note that it *not* recommended to delay a task for many hours because feedback from the task will not be available for such a long period of time.

`minutes`

is the number of minutes the task is delayed. The valid range of values is 0 to 59 (`OS_OPT_TIME_HMSM_STRICT`), or 0..9,999 (`OS_OPT_TIME_HMSM_NON_STRICT`). Please note that it *not* recommended to delay a task for tens to hundreds of minutes because feedback from the task will not be available for such a long period of time.

`seconds`

is the number of seconds the task is delayed. The valid range of values is 0 to 59 (`OS_OPT_TIME_HMSM_STRICT`), or 0..65,535 (`OS_OPT_TIME_HMSM_NON_STRICT`).

`milli`

is the number of milliseconds the task is delayed. The valid range of values is 0 to 999 (`OS_OPT_TIME_HMSM_STRICT`), or 0..4,294,967,295 (`OS_OPT_TIME_HMSM_NON_STRICT`). 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 4 ms results in no delay because the delay is rounded to the nearest tick. Thus, a delay of 15 ms actually results in a delay of 20 ms.

`opt`

is the desired mode and can be either:

`OS_OPT_TIME_HMSM_STRICT`

(see above)

`OS_OPT_TIME_HMSM_NON_STRICT`

(see above)

`OS_OPT_TIME_DLY`

Specifies a relative delay.

`OS_OPT_TIME_TIMEOUT`

Same as `OS_OPT_TIME_DLY`.

`OS_OPT_TIME_PERIODIC`

Specifies periodic mode.

OS_OPT_TIME_MATCH

Specifies that the task will wake up when OSTickCtr reaches the value specified by hours, minutes, seconds and milli.

p_err

is a pointer to a variable that contains an error code returned by this function.

OS_ERR_NONE

If the call was successful and the task has returned from the desired delay.

OS_ERR_OPT_INVALID

If OS_CFG_ARG_CHK_EN is set to DEF_ENABLED in os_cfg.h: if a valid option is not specified.

OS_ERR_OS_NOT_RUNNING

If OS_CFG_INVALID_OS_CALLS_CHK_EN is set to DEF_ENABLED in os_cfg.h: if µC/OS-III is not running yet.

OS_ERR_SCHED_LOCKED

If the scheduler is locked.

OS_ERR_TIME_DLY_ISR

If OS_CFG_CALLED_FROM_ISR_CHK_EN set to DEF_ENABLED in os_cfg.h: if calling this function from an ISR.

OS_ERR_TIME_INVALID_HOURS

If OS_CFG_ARG_CHK_EN is set to DEF_ENABLED in os_cfg.h: if not specifying a valid value for hours.

`OS_ERR_TIME_INVALID_MINUTES`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if not specifying a valid value for minutes.

`OS_ERR_TIME_INVALID_SECONDS`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if not specifying a valid value for seconds.

`OS_ERR_TIME_INVALID_MILLISECONDS`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if not specifying a valid value for milliseconds.

`OS_ERR_TIME_ZERO_DLY`

If specifying a delay of 0 because all the time arguments are 0.

Returned Value

None

Required Configuration

`OS_CFG_TIME_DLY_HMSM_EN` must be enabled in `os_cfg.h`. Refer to µC-OS-III Configuration Manual.

Callers

Application.

Notes/Warnings

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

2. The total delay (in ticks) must not exceed the maximum acceptable value that an `OS_TICK` variable can hold. Typically `OS_TICK` is a 32-bit value.

Example Usage

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


              while (DEF_ON) {
                  :
                  :
                  OSTimeDlyHMSM(0,
                                0,
                                1,
                                0,
                                OS_OPT_TIME_HMSM_STRICT,
                                &err);              /* Delay task for 1 second */
                  /* Check “err” */
                  :
                  :
              }
          }
```

Listing - OSTimeDlyHMSM() example usage

OSTimeDlyResume

Description

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

Files

os.h/os_time.c

Prototype

```
void  OSTimeDlyResume (OS_TCB  *p_tcb,
                       OS_ERR  *p_err)
```

Arguments

`p_tcb`

is a pointer to the TCB of the task that is resuming. A `NULL` pointer is not valid since it would indicate that the user is attempting to resume the current task and that is not possible as the caller cannot possibly be delayed.

`p_err`

is a pointer to a variable that contains an error code returned by this function.

`OS_ERR_NONE`

If the call was successful and the task was resumed.

`OS_ERR_OS_NOT_RUNNING`

If `OS_CFG_INVALID_OS_CALLS_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if µC/OS-III is not running yet.

`OS_ERR_STATE_INVALID`

If the task is in an invalid state.

`OS_ERR_TASK_NOT_DLY`

If the task was not delayed or, you passed a NULL pointer for the TCB.

`OS_ERR_TASK_SUSPENDED`

If the task to resume is suspended and will remain suspended.

`OS_ERR_TCB_INVALID`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if 'p_tcb' is a NULL pointer

`OS_ERR_TIME_DLY_RESUME_ISR`

If `OS_CFG_CALLED_FROM_ISR_CHK_EN` set to `DEF_ENABLED` in `os_cfg.h`: if calling this function from an ISR.

Returned Value

None

Required Configuration

`OS_CFG_TIME_DLY_RESUME_EN` must be enabled in `os_cfg.h`. Refer to µC-OS-III Configuration Manual.

Callers

Application.

Notes/Warnings

1. Do not call this function to resume a task that is waiting for an event with timeout.

Example Usage

```
OS_TCB  AnotherTaskTCB;


          void TaskX (void *p_arg)
          {
              OS_ERR  err;


              while (DEF_ON) {
                  :
                  OSTimeDlyResume(&AnotherTaskTCB,
                                  &err);
                  /* Check "err" */
                  :
                  :
              }
          }
```

Listing - OSTimeDlyResume() example usage

OSTimeDynTick

Description

Notifies the kernel that a dynamic tick has occurred, and that time delays and timeouts need to be updated. This function must be called from the dynamic tick ISR.

Files

os.h/os_time.c

Prototype

```
void  OSTimeDynTick (OS_TICK  ticks)
```

Arguments

`ticks`

Passed by the dynamic tick ISR, this is the number of OS Ticks elapsed since the last ISR. Normally, this would be the number of ticks as desired by `BSP_OS_TickNextSet()`.

Returned Value

None

Required Configuration

`OS_CFG_DYN_TICK_EN` must be enabled in `os_cfg.h`. Refer to µC-OS-III Configuration Manual
.

Callers

Dynamic tick ISR.

Notes/Warnings

1. To use the Dynamic Tick feature, the OS Board Support Package (BSP) should implement the Dynamic Tick API defined in OS Board Support Package.

Example Usage

```
OS_TICK  BSP_OS_TicksToGo;    /* As set by BSP_OS_TickNextSet() */


         void  BSP_OS_DynTick_Handler (void)
         {
             /* Clear interrupt source */
             OSTimeDynTick(BSP_OS_TicksToGo);
             :
             :
         }
```

Listing - OSTimeTick() example usage

OSTimeGet

Description

Returns the current value of the system clock. Specifically, it returns a snapshot of the variable `OSTickCtr`. The system clock is a counter of type `OS_TICK` that counts the number of clock ticks since power was applied, or since `OSTickCtr` was last set by `OSTimeSet()`.

Files

os.h/os_time.c

Prototype

```
OS_TICK  OSTimeGet (OS_ERR  *p_err)
```

Arguments

`p_err`

is a pointer to a variable that contains an error code returned by this function.

`OS_ERR_NONE`

If the call was successful.

Returned Value

The current value of `OSTickCtr` (in number of ticks).

Required Configuration

None

Callers

Application and ISRs.

Notes/Warnings

None

Example Usage

```
void TaskX (void *p_arg)
          {
              OS_TICK  clk;
              OS_ERR   err;


              while (DEF_ON) {
                  :
                  :
                  clk = OSTimeGet(&err);  /* Get current value of system clock */
                  /* Check “err” */
                  :
                  :
              }
          }
```

Listing - OSMutexCreate() example usage

OSTimeSet

Description

Sets the system clock. The system clock (`OSTickCtr`) is a counter, which has a data type of `OS_TICK`, and it counts the number of clock ticks since power was applied, or since the system clock was last set.

Files

os.h/os_time.c

Prototype

```
void  OSTimeSet (OS_TICK   ticks,
                 OS_ERR   *p_err)
```

Arguments

`ticks`

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

`p_err`

is a pointer to a variable that will contain an error code returned by this function.

`OS_ERR_NONE`

If the call was successful.

Returned Value

None

Required Configuration

None

Callers

Application and ISRs.

Notes/Warnings

1. You should be careful when using this function because other tasks may depend on the current value of the tick counter (`OSTickCtr`). Specifically, a task may delay itself (see `OSTimeDly()` and specify to wake up when `OSTickCtr` reaches a specific value.

Example Usage

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


              while (DEF_ON) {
                  :
                  :
                  OSTimeSet(0,
                            &err);        /* Reset the system clock  */
                  /* Check “err” */
                  :
                  :
              }
          }
```

Listing - OSTimeSet() example usage

OSTimeTick

Description

Notifies the kernel that a tick has just occurred, and that time delays and timeouts need to be updated. This function must be called from the tick ISR.

Files

os.h/os_time.c

Prototype

```
void  OSTimeTick (void)
```

Arguments

None

Returned Value

None

Required Configuration

None

Callers

Tick ISR.

Notes/Warnings

None

Example Usage

```
void MyTickISR (void)
          {
              /* Clear interrupt source */
              OSTimeTick();
              :
              :
          }
```

Listing - OSTimeTick() example usage

OSTimeTickHook

Description

This function is called by `OSTimeTick()`, which is assumed to be called from an ISR. `OSTimeTickHook()` is called at the very beginning of `OSTimeTick()` to give priority to user or port-specific code when the tick interrupt occurs.

If the `#define OS_CFG_APP_HOOKS_EN` is set to `DEF_ENABLED` in `os_cfg.h`, `OSTimeTickHook()` will call `App_OS_TimeTickHook()`.

`OSTimeTickHook()` is part of the CPU port code and the function *must not* be called by the application code. `OSTimeTickHook()` is actually used by the µC/OS-III port developer.

Files

os.h/os_cpu_c.c

Prototype

```
void  OSTimeTickHook (void);
```

Arguments

None

Returned Value

None

Required Configuration

`OS_CFG_APP_HOOKS_EN` must be enabled in `os_cfg.h`. Refer to µC-OS-III Configuration Manual.

Callers

None

Notes/Warnings

1. *Do not* call this function from the application.

Example Usage

The code below calls an application-specific hook that the application programmer can define. The user can simply set the value of `OS_AppTimeTickHookPtr` to point to the desired hook function `OSTimeTickHook()` is called by `OSTimeTick()` which in turn calls `App_OS_TimeTickHook()` through the pointer `OS_AppTimeTickHookPtr`.

```
void  App_OS_TimeTickHook (void)                           /* os_app_hooks.c        */
          {
              /* Your code goes here! */
          }


          void App_OS_SetAllHooks (void)                             /* os_app_hooks.c
*/
          {
              CPU_SR_ALLOC();

              CPU_CRITICAL_ENTER();
              :
              OS_AppTimeTickHookPtr = App_OS_TimeTickHook;
              :
              CPU_CRITICAL_EXIT();
          }


          void  OSTimeTickHook (void)                                /* os_cpu_c.c
*/
          {
          #if OS_CFG_APP_HOOKS_EN > 0u
              if (OS_AppTimeTickHookPtr != (OS_APP_HOOK_VOID)0) {    /* Call application hook
*/
                  (*OS_AppTimeTickHookPtr)();
              }
          #endif
          }
```

Listing - App_OS_TimeTickHook() example usage

API - Thread Local Storage

- OS_TLS_GetID()
- OS_TLS_GetValue()
- OS_TLS_SetDestruct()
- OS_TLS_SetValue()

OS_TLS_GetID

Description

Called by the application to assign a TLS (thread-local storage) ID for a specific purpose. See Thread Safety of the Compiler’s Run-Time Library for details on TLS. TLS IDs are assigned dynamically as needed by the application. Once assigned, TLS IDs cannot be un-assigned.

Files

os.h/os_tls.c

Prototype

```
OS_TLS_ID  OS_TLS_GetID (OS_ERR  *p_err)
```

Arguments

`p_err`

is a pointer to a variable that contains an error code returned by this function. Possible values are:

`OS_ERR_NONE`

If the call was successful and the caller was returned a TLS ID.

`OS_ERR_TLS_NO_MORE_AVAIL`

If you called `OS_TLS_GetID()` more than `OS_CFG_TLS_TBL_SIZE` times.

Returned Value

The next available TLS ID or `OS_CFG_TLS_TBL_SIZE` if there are no more TLS IDs available.

Required Configuration

`OS_CFG_TLS_TBL_SIZE` must be greater than 0 in `os_cfg.h`. Refer to µC-OS-III Configuration Manual.

Callers

Application.

Notes/Warnings

None

Example Usage

```
OS_TLS_ID   MyTLS_ID;


          void  main (void)
          {
              OS_ERR   err;


              :
              OSInit(&err);
              :
              :
              MyTLS_ID = OS_TLS_GetID(&err);  /* Obtain the next available TLS ID       */
              /* Check “err” */
              :
              :
          }
```

Listing - OS_TLS_GetID() example usage

OS_TLS_GetValue

Description

Returns the current value of a task's TLS (thread-local storage) stored in the task's p_tcb->TLS_Tbl[id]. See Chapter 20, "Thread Safety of the Compiler's Run-Time Library" for details on TLS.

Files

os.h/os_tls.c

Prototype

```
OS_TLS  OS_TLS_GetValue (OS_TCB    *p_tcb,
                         OS_TLS_ID  id,
                         OS_ERR    *p_err);
```

Arguments

`p_tcb`

is a pointer to the `OS_TCB` of the task you wish to retrieve the TLS from. You will get a copy of the `p_tcb->TLS_Tbl[id]` entry and of course, the entry will not be changed.

`id`

is the TLS ID of the entry you desire.

`p_err`

is a pointer to a variable that contains an error code returned by this function. Possible values are:

`OS_ERR_NONE`

If the call was successful and the caller was returned the value.

`OS_ERR_OS_NOT_RUNNING`

If you called `OS_TLS_GetValue()` and the kernel has not started yet. However, it's acceptable to call this function prior to starting multitasking but in this case, you must specify a non-`NULL` pointer for `p_tcb`.

`OS_ERR_TLS_ID_INVALID`

If you called `OS_TLS_GetValue()` and specified a TLS ID that has not been assigned. See `OS_TLS_GetID()` about assigning TLS IDs.

`OS_ERR_TLS_NOT_EN`

If you called `OS_TLS_GetValue()` but the task was created with the option `OS_OPT_TASK_NO_TLS` indicating that the task does not need TLS support.

Returned Value

The value store in `p_tcb->TLS_Tbl[id]` or `NULL` if an error occurred.

Required Configuration

`OS_CFG_TLS_TBL_SIZE` must be greater than 0 in `os_cfg.h`. Refer to µC-OS-III Configuration Manual.

Callers

Application.

Notes/Warnings

1. You cannot call `OS_TLS_GetValue()` for a task until that task gets created.

Example Usage

```
OS_TLS_ID   MyTLS_ID;


          void MyTask (void *p_arg)
          {
              OS_ERR      err;
              OS_TLS      p_tls;


              :
              :
              while (DEF_TRUE) {
                  p_tls = OSTLS_GetValue((OS_TCB   *)0,
                                         (OS_TLS_ID)MyTLS_ID,
                                         (OS_ERR  *)&err);
                  /* Check “err” */
                  :
                  :
              }
          }
```

Listing - OS_TLS_GetValue() example usage

OS_TLS_SetDestruct

Description

Assigns a “destructor function” to a TLS (thread-local storage) ID. All destructor functions that have been set for the TLS IDs will be called when the task is deleted. Destructor functions are thus common to all tasks. Note that a destructor function must be declared as follows:

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

When the destructor function is called, it will be passed the address of the `OS_TCB` for the task being deleted, the TLS ID that is being destructed and the value of `p_tcb->TLS_Tbl[id]` which was set by `OS_TLS_SetValue()`.

Files

os.h/os_tls.c

Prototype

```
void  OS_TLS_SetDestruct (OS_TLS_ID             id,
                          OS_TLS_DESTRUCT_PTR   p_destruct,
                          OS_ERR               *p_err)
```

Arguments

`id`

is the TLS ID for which you want to set the destructor function for.

`p_destruct`

is a pointer to the destructor function you want to assign to the TLS ID.

`p_err`

is a pointer to a variable that contains an error code returned by this function. Possible values are:

`OS_ERR_NONE`

If the call was successful and the destructor function was assigned to the TLS ID value.

`OS_ERR_TLS_DESTRUCT_ASSIGNED`

If a destructor function has already been assigned. You can only assign a destructor function once for each TLS ID.

`OS_ERR_TLS_ID_INVALID`

If you specified a TLS ID that has not been assigned. See `OS_TLS_GetID()` about assigning TLS IDs.

Returned Value

None

Required Configuration

`OS_CFG_TLS_TBL_SIZE` must be greater than 0 in `os_cfg.h`. Refer to µC-OS-III Configuration Manual.

Callers

Application.

Notes/Warnings

1. You can only call `OS_TLS_SetDestruct()` once for each TLS ID.

2. Note that not all implementations of `os_tls.c` will have destructors for TLS IDs.

Example Usage

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

          OS_TLS_ID   MyTLS_ID;


          void  main (void)
          {
              OS_ERR   err;


              :
              OSInit(&err);
              :
              :
              MyTLS_ID = OS_TLS_GetID(&err);  /* Obtain the next available TLS ID       */
              OS_TSL_SetDestruct((OS_TLS_ID)MyTLS_ID,
                                 (OS_TLS_DESTRUCT_PTR)MyTLS_Destructor,
                                 (OS_ERR *)&err);
              /* Check “err” */
              :
              :
          }


          void  MyDestructFunction (OS_TCB    *p_tcb,
                                    OS_TLS_ID  id,
                                    OS_TLS     value)
          {
              /* Note that ‘value’ is typically a ‘void *’ that points to storage area for the
TLS */
          }
```

Listing - OS_TLS_SetDestruct() example usage

OS_TLS_SetValue

Description

Sets the value of a TLS (thread-local storage) entry in the specified task's `OS_TCB`. Specifically, this function assigns value to `p_tcb->TLS_Tbl[id]`. See Chapter 20, "Thread Safety of the Compiler's Run-Time Library" for details on TLS.

Files

os.h/os_tls.c

Prototype

```
void  OS_TLS_SetValue (OS_TCB     *p_tcb,
                       OS_TLS_ID   id,
                       OS_TLS      value,
                       OS_ERR     *p_err)
```

Arguments

`p_tcb`

is a pointer to the `OS_TCB` of the task you wish to assign the TLS value to. `value` will thus be assigned to `p_tcb->TLS_Tbl[id]`.

`id`

is the TLS ID of the entry you are setting.

`value`

is the value to store at `p_tcb->TLS_Tbl[id]`.

`p_err`

is a pointer to a variable that contains an error code returned by this function. Possible values are:

`OS_ERR_NONE`

If the call was successful and the caller was returned the value.

`OS_ERR_OS_NOT_RUNNING`

If you called `OS_TLS_SetValue()` and the kernel has not started yet. However, it's acceptable to call this function prior to starting multitasking but in this case, you must specify a non-`NULL` pointer for `p_tcb`.

`OS_ERR_TLS_ID_INVALID`

If you called `OS_TLS_GetValue()` and specified a TLS ID that has not been assigned. See `OS_TLS_GetID()` about assigning TLS IDs.

`OS_ERR_TLS_NOT_EN`

If you called `OS_TLS_SetValue()` but the task was created with the option `OS_OPT_TASK_NO_TLS` indicating that the task does not need TLS support.

Returned Value

None

Required Configuration

`OS_CFG_TLS_TBL_SIZE` must greater than 0 in `os_cfg.h`. Refer to µC-OS-III Configuration Manual.

Callers

Application.

Notes/Warnings

1. You cannot call `OS_TLS_SetValue()` for a task until that task gets created.

Example Usage

```
OS_TLS_ID   MyTLS_ID;


          void MyTask (void *p_arg)
          {
              OS_ERR      err;
              OS_TLS      p_tls;


              :
              :
              while (DEF_TRUE) {
                  p_tls = OSTLS_GetValue((OS_TCB   *)0,
                                         (OS_TLS_ID)MyTLS_ID,
                                         (OS_ERR  *)&err);
                  /* Check “err” */
                  :
                  :
              }
          }
```

Listing - OS_TLS_SetValue() example usage

API - Message Queues

- OSQCreate()
- OSQDel()
- OSQFlush()
- OSQPend()
- OSQPendAbort()
- OSQPost()

OSQCreate

Description

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.

Files

os.h/os_q.c

Prototype

```
void  OSQCreate (OS_Q        *p_q,
                 CPU_CHAR    *p_name,
                 OS_MSG_QTY   max_qty,
                 OS_ERR      *p_err)
```

Arguments

`p_q`

is a pointer to the message queue control block. It is assumed that storage for the message queue will be allocated in the application. The user will need to declare a “global” variable as follows, and pass a pointer to this variable to `OSQCreate()`:

```
OS_Q  MyMsgQ;
```

`p_name`

is a pointer to an ASCII string used to name the message queue. The name can be displayed by debuggers or µC/Probe.

`msg_qty`

indicates the maximum size of the message queue (must be non-zero). If the user intends to not limit the size of the queue, simply pass a very large number. Of course, if there are not enough `OS_MSGs` in the pool of `OS_MSGs`, the post call (i.e., `OSQPost()`) will simply fail and an error code will indicate that there are no more `OS_MSGs` to use.

`p_err`

is a pointer to a variable that is used to hold an error code:

`OS_ERR_NONE`

If the call is successful and the mutex has been created.

`OS_ERR_CREATE_ISR`

If `OS_CFG_CALLED_FROM_ISR_CHK_EN` set to `DEF_ENABLED` in `os_cfg.h`: if attempting to create the message queue from an ISR.

`OS_ERR_ILLEGAL_CREATE_RUN_TIME`

If `OS_SAFETY_CRITICAL_IEC61508` is defined: you called this after calling `OSStart()` and thus you are no longer allowed to create additional kernel objects.

`OS_ERR_OBJ_PTR_NULL`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if `p_q` is a `NULL` pointer.

`OS_ERR_Q_SIZE`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if the size specified is 0.

Returned Value

None

Required Configuration

`OS_CFG_Q_EN` must be enabled in `os_cfg.h`. Refer to µC-OS-III Configuration Manual.

Callers

Application.

Notes/Warnings

1. Queues must be created before they are used.

Example Usage

```
OS_Q   CommQ;


          void main (void)
          {
              OS_ERR   err;


              OSInit(&err);                           /* Initialize µC/OS-III  */
              :
              :
              OSQCreate(&CommQ,
                        “Comm Queue”,
                         10,
                        &err);                        /* Create COMM Q         */
              /* Check “err” */
              :
              :
              OSStart();                              /* Start Multitasking    */
          }
```

Listing - OSQCreate() example usage

OSQDel

Description

Deletes a message queue. This function should be used with care since multiple tasks may rely on the presence of the message queue. Generally speaking, before deleting a message queue, first delete all the tasks that can access the message queue. However, it is highly recommended that you do not delete kernel objects at run time.

Files

os.h/os_q.c

Prototype

```
OS_OBJ_QTY  OSQDel (OS_Q    *p_q,
                    OS_OPT   opt,
                    OS_ERR  *p_err)
```

Arguments

`p_q`

is a pointer to the message queue to delete.

`opt`

specifies whether to delete the queue only if there are no pending tasks (`OS_OPT_DEL_NO_PEND`), or always delete the queue regardless of whether tasks are pending or not (`OS_OPT_DEL_ALWAYS`). In this case, all pending task are readied.

`p_err`

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

`OS_ERR_NONE`

If the call is successful and the message queue has been deleted.

`OS_ERR_DEL_ISR`

If `OS_CFG_CALLED_FROM_ISR_CHK_EN` set to `DEF_ENABLED` in `os_cfg.h`: if the user attempts to delete the message queue from an ISR.

`OS_ERR_ILLEGAL_DEL_RUN_TIME`

If `OS_SAFETY_CRITICAL_IEC61508` is defined: you called this after calling `OSStart()` and thus you are no longer allowed to delete kernel objects.

`OS_ERR_OBJ_PTR_NULL`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if passing a `NULL` pointer for `p_q`.

`OS_ERR_OBJ_TYPE`

If `OS_CFG_OBJ_TYPE_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if `p_q` is not pointing to a queue.

`OS_ERR_OPT_INVALID`

If not specifying one of the two options mentioned in the opt argument.

`OS_ERR_OS_NOT_RUNNING`

If `OS_CFG_INVALID_OS_CALLS_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if µC/OS-III is not running yet.

`OS_ERR_TASK_WAITING`

If one or more tasks are waiting for messages at the message queue and it is specified to only delete if no task is pending.

Returned Value

The number of tasks that were waiting on the message queue and `0` if an error is detected or if no tasks were waiting.

Required Configuration

`OS_CFG_Q_EN` and `OS_CFG_Q_DEL_EN` must be enabled in `os_cfg.h`. Refer to µC-OS-III Configuration Manual.

Callers

Application.

Notes/Warnings

1. Message queues must be created before they can be used.
2. This function must be used with care. Tasks that would normally expect the presence of the queue *must* check the return code of `OSQPend()`.

Example Usage

```
OS_Q  DispQ;


          void Task (void *p_arg)
          {
              OS_ERR      err;
              OS_OBJ_QTY  qty;


              (void)&p_arg;
              while (DEF_ON) {
                  :
                  :
                  qty = OSQDel(&DispQ,
                                OS_OPT_DEL_ALWAYS,
                               &err);
                  /* Check "err" */
                  :
                  :
              }
          }
```

Listing - OSQDel() example usage

OSQFlush

Description

Empties the contents of the message queue and eliminates all 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. `OS_MSGs` from the queue are simply returned to the free pool of `OS_MSGs`.

Files

os.h/os_q.c

Prototype

```
OS_MSG_QTY  OSQFlush (OS_Q    *p_q,
                      OS_ERR  *p_err)
```

Arguments

`p_q`

is a pointer to the message queue.

`p_err`

is a pointer to a variable that will contain an error code returned by this function.

`OS_ERR_NONE`

If the message queue is flushed.

`OS_ERR_FLUSH_ISR`

If `OS_CFG_CALLED_FROM_ISR_CHK_EN` set to `DEF_ENABLED` in `os_cfg.h`: if calling this function from an ISR

`OS_ERR_OBJ_PTR_NULL`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if `p_q` is a `NULL` pointer.

`OS_ERR_OBJ_TYPE`

If `OS_CFG_OBJ_TYPE_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if you attempt to flush an object other than a message queue.

`OS_ERR_OS_NOT_RUNNING`

If `OS_CFG_INVALID_OS_CALLS_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if µC/OS-III is not running yet.

Returned Value

The number of `OS_MSG` entries freed from the message queue. Note that the `OS_MSG` entries are returned to the free pool of `OS_MSGs`.

Required Configuration

`OS_CFG_Q_EN` and `OS_CFG_Q_FLUSH_EN` must be enabled in `os_cfg.h`. Refer to µC-OS-III Configuration Manual.

Callers

Application.

Notes/Warnings

1. Queues must be created before they are used.
2. Use this function with great care. When flushing a queue, you lose the references to what the queue entries are pointing to, potentially causing 'memory leaks'. The data that the user is pointing to that is referenced by the queue entries should, most likely, be de-allocated (i.e., freed).

Example Usage

```
OS_Q  CommQ;


          void Task (void *p_arg)
          {
              OS_ERR      err;
              OS_MSG_QTY  entries;

              (void)&p_arg;
              while (DEF_ON) {
                  :
                  :
                  entries = OSQFlush(&CommQ,
                                     &err);
                  /* Check "err" */
                  :
                  :
              }
          }
```

Listing - OSQFlush() example usage

or, to flush a queue that contains entries, instead you can use `OSQPend()` and specify the `OS_OPT_PEND_NON_BLOCKING` option.

```
OS_Q  CommQ;


          void Task (void *p_arg)
          {
              OS_ERR       err;
              CPU_TS       ts;
              OS_MSG_SIZE  msg_size;


              (void)&p_arg;
              :
              do {
                 OSQPend(&CommQ,
                          0,
                          OS_OPT_PEND_NON_BLOCKING,
                         &msg_size,
                         &ts,
                         &err);
              } while (err != OS_ERR_PEND_WOULD_BLOCK);
              :
              :
          }
```

Listing - Queue flush using OSQPend() example

OSQPend

Description

Used when a task wants to receive messages from a message queue. The messages are sent to the task via the message queue either by an ISR, or by another task using the `OSQPost()` call. The messages received are pointer-sized variables, and their use is application specific. If at least one message is already present in the message queue when `OSQPend()` is called, the message is retrieved and returned to the caller.

If no message is present in the message queue and `OS_OPT_PEND_BLOCKING` is specified for the `opt` argument, `OSQPend()` suspends the current task until either a message is received, or a user-specified timeout expires. If a message is sent to the message queue and multiple tasks are waiting for such a message, µC/OS-III resumes the highest priority task that is waiting.

A pended task suspended with `OSTaskSuspend()` can receive a message. However, the task remains suspended until it is resumed by calling `OSTaskResume()`.

If no message is present in the queue and `OS_OPT_PEND_NON_BLOCKING` is specifed for the opt argument, `OSQPend()` returns to the caller with an appropriate error code, and returns a `NULL` pointer.

Files

os.h/os_q.c

Prototype

```
void  *OSQPend (OS_Q         *p_q,
                OS_TICK       timeout,
                OS_OPT        opt,
                OS_MSG_SIZE  *p_msg_size,
                CPU_TS       *p_ts,
                OS_ERR       *p_err)
```

Arguments

p_q

is a pointer to the queue from which the messages are received.

`timeout`

allows the task to resume execution if a message is not received from the message queue within the specified number of clock ticks. A `timeout` value of 0 indicates that the task is willing to wait forever for a 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.

`opt`

determines whether or not to block if a message is not available in the queue. This argument must be set to either:

`OS_OPT_PEND_BLOCKING`, or
`OS_OPT_PEND_NON_BLOCKING`

Note that the `timeout` argument should be set to `0` when specifying `OS_OPT_PEND_NON_BLOCKING`, since the timeout value is irrelevant using this option.

`p_msg_size`

is a pointer to a variable that will receive the size of the message (in number of bytes).

`p_ts`

is a pointer to a variable that will receive the timestamp of when the message was received. Passing a `NULL` pointer is valid, and indicates that the user does not need the timestamp.

A timestamp is useful when the user wants the task to know when the message queue was posted, or how long it took for the task to resume after the message queue was posted. In the latter case, you would call `OS_TS_GET()` and compute the difference between the current value of the timestamp and `*p_ts`. In other words:

```
delta = OS_TS_GET() - *p_ts;
```

`p_err`

is a pointer to a variable used to hold an error code.

`OS_ERR_NONE`

If a message is received.

`OS_ERR_OBJ_DEL`

If the message queue was deleted.

`OS_ERR_OBJ_PTR_NULL`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if `p_q` is a `NULL` pointer.

`OS_ERR_OBJ_TYPE`

If `OS_CFG_OBJ_TYPE_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if `p_q` is not pointing to a message queue.

`OS_ERR_OPT_INVALID`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if you specified invalid options.

`OS_ERR_OS_NOT_RUNNING`

If `OS_CFG_INVALID_OS_CALLS_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if µC/OS-III is not running yet.

`OS_ERR_PEND_ABORT`

If the pend was aborted because another task called `OSQPendAbort()`.

`OS_ERR_PEND_ISR`

If `OS_CFG_CALLED_FROM_ISR_CHK_EN` set to `DEF_ENABLED` in `os_cfg.h`: if the function is called from an ISR.

OS_ERR_PEND_WOULD_BLOCK

If this function is called with the opt argument set to OS_OPT_PEND_NON_BLOCKING, and no message is in the queue.

OS_ERR_PTR_INVALID

If OS_CFG_ARG_CHK_EN is set to DEF_ENABLED in os_cfg.h: if p_msg_size is a NULL pointer.

OS_ERR_SCHED_LOCKED

If calling this function when the scheduler is locked.

OS_ERR_TIMEOUT

If a message is not received within the specified timeout.

Returned Value

The message (i.e., a pointer) or a NULL pointer if no messages has been received. Note that it is possible for the actual message to be a NULL pointer, so you should check the returned error code instead of relying on the returned value.

Required Configuration

OS_CFG_Q_EN must be enabled in os_cfg.h. Refer to µC-OS-III Configuration Manual.

Callers

Application.

Notes/Warnings

1. Queues must be created before they are used.

Example Usage

```
OS_Q  CommQ;


          void CommTask (void *p_arg)
          {
              OS_ERR       err;
              void        *p_msg;
              OS_MSG_SIZE  msg_size;
              CPU_TS       ts;


              (void)&p_arg;
              while (DEF_ON) {
                  :
                  :
                  p_msg = OSQPend(&CommQ,
                                   100,
                                   OS_OPT_PEND_BLOCKING,
                                  &msg_size,
                                  &ts,
                                  &err);
                  /* Check “err” */
                  :
                  :
              }
          }
```

Listing - OSQPend() example usage

OSQPendAbort

Description

Aborts and readies any tasks currently waiting on a message queue. This function should be used to fault-abort the wait on the message queue, rather than to signal the message queue via `OSQPost()`.

Files

os.h/os_q.c

Prototype

```
OS_OBJ_QTY  OSQPendAbort (OS_Q    *p_q,
                          OS_OPT   opt,
                          OS_ERR  *p_err)
```

Arguments

`p_q`

is a pointer to the queue for which pend(s) need to be aborted.

`opt`

determines the type of abort to be performed.

`OS_OPT_PEND_ABORT_1`

Aborts the pend of only the highest-priority task waiting on the message queue.

`OS_OPT_PEND_ABORT_ALL`

Aborts the pend of all tasks waiting on the message queue.

`OS_OPT_POST_NO_SCHED`

specifies that the scheduler should not be called, even if the pend of a higher-priority task has been aborted. Scheduling will need to occur from another function.

You would use this option if the task calling `OSQPendAbort()` is doing additional pend aborts, rescheduling is not performed until completion, and multiple pend aborts are to take effect simultaneously.

`p_err`

is a pointer to a variable that holds an error code:

`OS_ERR_NONE`

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

`OS_ERR_OBJ_PTR_NULL`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if `p_q` is a `NULL` pointer.

`OS_ERR_OBJ_TYPE`

If `OS_CFG_OBJ_TYPE_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if `p_q` is not pointing to a message queue.

`OS_ERR_OPT_INVALID`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if a valid option is not specified.

`OS_ERR_OS_NOT_RUNNING`

If `OS_CFG_INVALID_OS_CALLS_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if µC/OS-III is not running yet.

`OS_ERR_PEND_ABORT_ISR`

If `OS_CFG_CALLED_FROM_ISR_CHK_EN` set to `DEF_ENABLED` in `os_cfg.h`: if called from an ISR

`OS_ERR_PEND_ABORT_NONE`

If no task was pending on the message queue

Returned Value

`OSQPendAbort()` returns the number of tasks made ready-to-run by this function. Zero indicates that no tasks were pending on the message queue, or an error.

Required Configuration

`OS_CFG_Q_EN` and `OS_CFG_Q_PEND_ABORT_EN` must be enabled in `os_cfg.h`. Refer to µC-OS-III Configuration Manual.

Callers

Application.

Notes/Warnings

1. Queues must be created before they are used.

Example Usage

```
OS_Q  CommQ;


          void CommTask(void *p_arg)
          {
              OS_ERR      err;
              OS_OBJ_QTY  nbr_tasks;


              (void)&p_arg;
              while (DEF_ON) {
                  :
                  :
                  nbr_tasks = OSQPendAbort(&CommQ,
                                            OS_OPT_PEND_ABORT_ALL,
                                           &err);
                  /* Check “err” */
                  :
                  :
              }
          }
```

Listing - OSQPendAbort() example usage

OSQPost

Description

Sends a message to a task through a message 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 message queue.

If any task is waiting for a message to be posted to the message 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 can be first-in first-out (`OS_OPT_POST_FIFO`), or last-in-first-out (`OS_OPT_POST_LIFO`) depending of the value specified in the `opt` argument.

If any task is waiting for a message at the message queue, `OSQPost()` allows the user to either post the message to the highest-priority task waiting at the queue (`opt` set to `OS_OPT_POST_FIFO` or `OS_OPT_POST_LIFO`), or to all tasks waiting at the message queue (`opt` is set to `OS_OPT_POST_ALL`). In either case, scheduling occurs unless `opt` is also set to `OS_OPT_POST_NO_SCHED`.

Files

os.h/os_q.c

Prototype

```
void  OSQPost (OS_Q         *p_q,
               void         *p_void,
               OS_MSG_SIZE   msg_size,
               OS_OPT        opt,
               OS_ERR       *p_err)
```

Arguments

p_q

is a pointer to the message queue being posted to.

`p_void`

is the actual message posted. `p_void` is a pointer-sized variable. Its meaning is application specific.

`msg_size`

specifies the size of the message (in number of bytes).

`opt`

determines the type of POST performed. The last two options may be added to either `OS_OPT_POST_FIFO` or `OS_OPT_POST_LIFO` to create different combinations:

`OS_OPT_POST_FIFO`

POST message to the end of the queue (FIFO), or send message to a single waiting task.

`OS_OPT_POST_LIFO`

POST message to the front of the queue (LIFO), or send message to a single waiting task

`OS_OPT_POST_ALL`

POST message to ALL tasks that are waiting on the queue. This option can be added to either `OS_OPT_POST_FIFO` or `OS_OPT_POST_LIFO`.

`OS_OPT_POST_NO_SCHED`

This option specifies to not call the scheduler after the post and therefore the caller is resumed, even if the message was posted to a message queue with tasks having a higher priority than the caller.

You would use this option if the task (or ISR) calling `OSQPost()` will do additional posts, in this case, the caller does not want to reschedule until finished, and, multiple posts are to take effect simultaneously.

`p_err`

is a pointer to a variable that will contain an error code returned by this function.

`OS_ERR_NONE`

If no tasks were waiting on the queue. In this case, the return value is also 0.

`OS_ERR_MSG_POOL_EMPTY`

If there are no more `OS_MSG` structures to use to store the message.

`OS_ERR_OBJ_PTR_NULL`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if `p_q` is a `NULL` pointer.

`OS_ERR_OBJ_TYPE`

If `OS_CFG_OBJ_TYPE_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if `p_q` is not pointing to a message queue.

`OS_ERR_OPT_INVALID`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if a valid option is not specified.

`OS_ERR_OS_NOT_RUNNING`

If `OS_CFG_INVALID_OS_CALLS_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if µ C/OS-III is not running yet.

`OS_ERR_Q_MAX`

If the queue is full and therefore cannot accept more messages.

Returned Value

None

Required Configuration

`OS_CFG_Q_EN` must be enabled in `os_cfg.h`. Refer to µC-OS-III Configuration Manual.

Callers

Application and ISRs.

Notes/Warnings

1. Queues must be created before they are used.

2. Possible combinations of options are:
   `OS_OPT_POST_FIFO`
   `OS_OPT_POST_LIFO`
   `OS_OPT_POST_FIFO + OS_OPT_POST_ALL`
   `OS_OPT_POST_LIFO + OS_OPT_POST_ALL`
   `OS_OPT_POST_FIFO + OS_OPT_POST_NO_SCHED`
   `OS_OPT_POST_LIFO + OS_OPT_POST_NO_SCHED`
   `OS_OPT_POST_FIFO + OS_OPT_POST_ALL + OS_OPT_POST_NO_SCHED`
   `OS_OPT_POST_LIFO + OS_OPT_POST_ALL + OS_OPT_POST_NO_SCHED`

3. Although the example below shows calling `OSQPost()` from a task, it can also be called from an ISR.

Example Usage

```
OS_Q         CommQ;
          CPU_INT08U   CommRxBuf[100];


          void CommTaskRx (void *p_arg)
          {
              OS_ERR  err;


              (void)&p_arg;
              while (DEF_ON) {
                  :
                  :
                  OSQPost(&CommQ,
                          &CommRxBuf[0],
                           sizeof(CommRxBuf),
                           OS_OPT_POST_OPT_FIFO + OS_OPT_POST_ALL + OS_OPT_POST_NO_SCHED,
                          &err);
                  /* Check “err” */
                  :
                  :
              }
          }
```

Listing - OSQPost() example usage

API - Semaphores

OSSemCreate

Description

Initializes a semaphore. Semaphores are 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. You would use a semaphore to signal the occurrence of an event to one or multiple tasks, and use mutexes to guard share resources. However, technically, semaphores allow for both.

Files

os.h/os_sem.c

Prototype

```
void  OSSemCreate (OS_SEM      *p_sem,
                   CPU_CHAR    *p_name,
                   OS_SEM_CTR   cnt,
                   OS_ERR      *p_err)
```

Arguments

`p_sem`

is a pointer to the semaphore control block. It is assumed that storage for the semaphore will be allocated in the application. In other words, you need to declare a “global” variable as follows, and pass a pointer to this variable to `OSSemCreate()`:

```
OS_SEM  MySem;
```

`p_name`

is a pointer to an ASCII string used to assign a name to the semaphore. The name can be displayed by debuggers or µC/Probe.

`cnt`

specifies the initial value of the semaphore.

If the semaphore is used for resource sharing, you would set the initial value of the semaphore to the number of identical resources guarded by the semaphore. If there is only one resource, the value should be set to `1` (this is called a binary semaphore). For multiple resources, set the value to the number of resources (this is called a counting semaphore).

If using a semaphore as a signaling mechanism, you should set the initial value to `0`.

`p_err`

is a pointer to a variable used to hold an error code:

`OS_ERR_NONE`

If the call is successful and the semaphore has been created.

`OS_ERR_CREATE_ISR`

If `OS_CFG_CALLED_FROM_ISR_CHK_EN` set to `DEF_ENABLED` in `os_cfg.h`: if you attempted to create a semaphore from an ISR.

`OS_ERR_ILLEGAL_CREATE_RUN_TIME`

If `OS_SAFETY_CRITICAL_IEC61508` is defined: you called this after calling `OSStart()` and thus you are no longer allowed to create additional kernel objects.

`OS_ERR_OBJ_PTR_NULL`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if `p_sem` is a `NULL` pointer.

Returned Value

None

Required Configuration

`OS_CFG_SEM_EN` must be enabled in `os_cfg.h`. Refer to µC-OS-III Configuration Manual.

Callers

Application.

Notes/Warnings

1. Semaphores must be created before they are used.

Example Usage

```
OS_SEM  SwSem;


          void main (void)
          {
              OS_ERR  err;


              :
              OSInit(&err);                 /* Initialize µC/OS-III             */
              :
              :
              OSSemCreate(&SwSem,           /* Create Switch Semaphore          */
                          “Switch Semaphore”,
                           0,
                          &err);
              /* Check “err” */
              :
              :
              OSStart(&err);                /* Start Multitasking              */
          }
```

Listing - OSSemCreate() example usage

OSSemDel

Description

Deletes a semaphore. This function should be used with care as multiple tasks may rely on the presence of the semaphore. Generally speaking, before deleting a semaphore, first delete all the tasks that access the semaphore. As a rule, it is highly recommended to not delete kernel objects at run time.

Deleting the semaphore will not de-allocate the object. In other words, storage for the variable will still remain at the same location unless the semaphore is allocated dynamically from the heap. The dynamic allocation of objects has its own set of problems. Specifically, it is not recommended for embedded systems to allocate (and de-allocate) objects from the heap given the high likelihood of fragmentation.

Files

os.h/os_sem.c

Prototype

```
OS_OBJ_QTY  OSSemDel (OS_SEM  *p_sem,
                      OS_OPT   opt,
                      OS_ERR  *p_err)
```

Arguments

`p_sem`

is a pointer to the semaphore.

`opt`

specifies one of two options: `OS_OPT_DEL_NO_PEND` or `OS_OPT_DEL_ALWAYS`.

`OS_OPT_DEL_NO_PEND` specifies to delete the semaphore only if no task is waiting on the semaphore. Because no task is “currently” waiting on the semaphore does not mean that a task will not attempt to wait for the semaphore later. How would such a task handle the situation waiting for a semaphore that was deleted? The application code will have to deal with this eventuality.

`OS_OPT_DEL_ALWAYS` specifies deleting the semaphore, regardless of whether tasks are waiting on the semaphore or not. If there are tasks waiting on the semaphore, these tasks will be made ready-to-run and informed (through an appropriate error code) that the reason the task is readied is that the semaphore it was waiting on was deleted. The same reasoning applies with the other option, how will the tasks handle the fact that the semaphore they want to wait for is no longer available?

`p_err`

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

`OS_ERR_NONE`

If the call is successful and the semaphore has been deleted.

`OS_ERR_DEL_ISR`

If `OS_CFG_CALLED_FROM_ISR_CHK_EN` set to `DEF_ENABLED` in `os_cfg.h`: if attempting to delete the semaphore from an ISR.

`OS_ERR_ILLEGAL_DEL_RUN_TIME`

If `OS_SAFETY_CRITICAL_IEC61508` is defined: you called this after calling `OSStart()` and thus you are no longer allowed to delete kernel objects.

`OS_ERR_OBJ_PTR_NULL`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if `p_sem` is a `NULL` pointer.

`OS_ERR_OBJ_TYPE`

If `OS_CFG_OBJ_TYPE_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if `p_sem` is not pointing to a semaphore.

`OS_ERR_OPT_INVALID`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if one of the two options mentioned in the `opt` argument is not specified.

`OS_ERR_OS_NOT_RUNNING`

If `OS_CFG_INVALID_OS_CALLS_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if µC/OS-III is not running yet.

`OS_ERR_TASK_WAITING`

If one or more tasks are waiting on the semaphore.

Returned Value

The number of tasks made ready-to-run by this function. Zero either indicates an error or that no tasks were pending on the semaphore.

Required Configuration

`OS_CFG_SEM_EN` and `OS_CFG_SEM_DEL_EN` must be enabled in `os_cfg.h`. Refer to µC-OS-III Configuration Manual.

Callers

Application.

Notes/Warnings

1. Use this call with care because other tasks might expect the presence of the semaphore.

Example Usage

```
OS_SEM  SwSem;


          void Task (void *p_arg)
          {
              OS_ERR      err;
              OS_OBJ_QTY  qty;


              (void)&p_arg;
              while (DEF_ON) {
                  :
                  :
                  qty = OSSemDel(&SwSem,
                                  OS_OPT_DEL_ALWAYS,
                                 &err);
                  /* Check “err” */
                  :
                  :
              }
          }
```

Listing - OSSemDel() example usage

OSSemPend

Description

Used when a task wants exclusive access to a resource, needs to synchronize its activities with an ISR or task, or is waiting until an event occurs.

When the semaphore is used for resource sharing, 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 the owner of the semaphore releases the semaphore by calling `OSSemPost()`, or the specified timeout expires. If the semaphore is signaled before the timeout expires, µC/OS-III resumes the highest-priority task waiting for the semaphore.

When the semaphore is used as a signaling mechanism, the calling task waits until a task or an ISR signals the semaphore by calling `OSSemPost()`, or the specified timeout expires.

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()`.

`OSSemPend()` also returns if the pend is aborted or, the semaphore is deleted.

Files

os.h/os_sem.c

Prototype

```
OS_SEM_CTR  OSSemPend (OS_SEM   *p_sem,
                       OS_TICK   timeout,
                       OS_OPT    opt,
                       CPU_TS   *p_ts,
                       OS_ERR   *p_err)
```

Arguments

`p_sem`

is a pointer to the semaphore.

`timeout`

allows the task to resume execution if a semaphore is not posted within the specified number of clock ticks. A timeout value of 0 indicates that the task waits forever for the semaphore. 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.

`opt`

specifies whether the call is to block if the semaphore is not available, or not block.

`OS_OPT_PEND_BLOCKING`

to block the caller until the semaphore is available or a timeout occurs.

`OS_OPT_PEND_NON_BLOCKING`

If the semaphore is not available, `OSSemPend()` will not block but return to the caller with an appropriate error code.

`p_ts`

is a pointer to a variable that will receive a timestamp of when the semaphore was posted, pend aborted, or deleted. Passing a `NULL` pointer is valid and indicates that a timestamp is not required.

A timestamp is useful when the task must know when the semaphore was posted or, how long it took for the task to resume after the semaphore was posted. In the latter case, call `OS_TS_GET()` and compute the difference between the current value of the timestamp and `*p_ts`. In other words:

```
delta = OS_TS_GET() - *p_ts;
```

`p_err`

is a pointer to a variable used to hold an error code:

`OS_ERR_NONE`

If the semaphore is available.

`OS_ERR_OBJ_DEL`

If the semaphore was deleted.

`OS_ERR_OBJ_PTR_NULL`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if `p_sem` is a `NULL` pointer.

`OS_ERR_OBJ_TYPE`

If `OS_CFG_OBJ_TYPE_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if `p_sem` is not pointing to a semaphore.

`OS_ERR_OPT_INVALID`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if `opt` is not `OS_OPT_PEND_NON_BLOCKING or OS_OPT_PEND_BLOCKING`.

`OS_ERR_OS_NOT_RUNNING`

If `OS_CFG_INVALID_OS_CALLS_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if µC/OS-III is not running yet.

`OS_ERR_PEND_ABORT`

`if` the pend was aborted

`OS_ERR_PEND_ISR`

If `OS_CFG_CALLED_FROM_ISR_CHK_EN` set to `DEF_ENABLED` in `os_cfg.h`: `if` this function is called from an ISR.

`OS_ERR_PEND_WOULD_BLOCK`

`if` this function is called as specified `OS_OPT_PEND_NON_BLOCKING`, and the semaphore was not available.

`OS_ERR_SCHED_LOCKED`

If calling this function when the scheduler is locked.

`OS_ERR_STATUS_INVALID`

If the pend status has an invalid value.

`OS_ERR_TIMEOUT`

If the semaphore is not signaled within the specified timeout.

Returned Value

The new value of the semaphore count.

Required Configuration

`OS_CFG_SEM_EN` must be enabled in `os_cfg.h`. Refer to µC-OS-III Configuration Manual.

Callers

Application.

Notes/Warnings

1. Semaphores must be created before they are used.

Example Usage

```
OS_SEM  SwSem;


          void DispTask (void *p_arg)
          {
              OS_ERR      err;
              CPU_TS      ts;
              OS_SEM_CTR  ctr;

              (void)&p_arg;
              while (DEF_ON) {
                  :
                  :
                  ctr = OSSemPend(&SwSem,
                                   0,
                                   OS_OPT_PEND_BLOCKING,
                                  &ts,
                                  &err);
                  /* Check “err” */
                  :
                  :
              }
          }
```

Listing - OSSemPend() example usage

OSSemPendAbort

Description

Aborts and readies any task 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()`.

Files

os.h/os_sem.c

Prototype

```
OS_OBJ_QTY  OSSemPendAbort (OS_SEM  *p_sem,
                            OS_OPT   opt,
                            OS_ERR  *p_err)
```

Arguments

`p_sem`

is a pointer to the semaphore for which pend(s) need to be aborted.

`opt`

determines the type of abort performed.

`OS_OPT_PEND_ABORT_1`

Aborts the pend of only the highest-priority task waiting on the semaphore.

`OS_OPT_PEND_ABORT_ALL`

Aborts the pend of all the tasks waiting on the semaphore.

`OS_OPT_POST_NO_SCHED`

Specifies that the scheduler should not be called, even if the pend of a higher-priority task has been aborted. Scheduling will need to occur from another function.

You would use this option if the task calling `OSSemPendAbort()` will be doing additional pend aborts, reschedule takes place when finished, and multiple pend aborts are to take effect simultaneously.

`p_err`

Is a pointer to a variable that holds an error code:

`OS_ERR_NONE`

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_OBJ_PTR_NULL`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: `if` `p_sem` is a `NULL` pointer.

`OS_ERR_OBJ_TYPE`

If `OS_CFG_OBJ_TYPE_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: `if` `p_sem` is not pointing to a semaphore.

`OS_ERR_OPT_INVALID`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: `if` an invalid option is specified.

`OS_ERR_OS_NOT_RUNNING`

If `OS_CFG_INVALID_OS_CALLS_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if µC/OS-III is not running yet.

`OS_ERR_PEND_ABORT_ISR`

If `OS_CFG_CALLED_FROM_ISR_CHK_EN` set to `DEF_ENABLED` in `os_cfg.h`: if you called this function from an ISR.

`OS_ERR_PEND_ABORT_NONE`

If no tasks were aborted because no task was waiting.

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 therefore, or an error.

Required Configuration

`OS_CFG_SEM_EN` and `OS_CFG_SEM_PEND_ABORT_EN` must be enabled in `os_cfg.h`. Refer to µC-OS-III Configuration Manual.

Callers

Application.

Notes/Warnings

1. Semaphores must be created before they are used.

Example Usage

```
OS_SEM  SwSem;


          void  CommTask(void  *p_arg)
          {
              OS_ERR      err;
              OS_OBJ_QTY  nbr_tasks;


              (void)&p_arg;
              while (DEF_ON) {
                  :
                  :
                  nbr_tasks = OSSemPendAbort(&SwSem,
                                              OS_OPT_PEND_ABORT_ALL,
                                             &err);
                  /* Check "err" */
                  :
                  :
              }
          }
```

Listing - OSSemCreate() example usage

OSSemPost

Description

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 that is ready-to-run.

Files

os.h/os_sem.c

Prototype

```
OS_SEM_CTR  OSSemPost (OS_SEM  *p_sem,
                       OS_OPT   opt,
                       OS_ERR  *p_err)
```

Arguments

`p_sem`

is a pointer to the semaphore.

`opt`

determines the type of post performed.

`OS_OPT_POST_1`

Post and ready only the highest-priority task waiting on the semaphore.

`OS_OPT_POST_ALL`

Post to all tasks waiting on the semaphore. You should only use this option if the semaphore is used as a signaling mechanism and never when the semaphore is used to guard a shared resource. It does not make sense to tell all tasks that are sharing a resource that they can all access the resource.

OS_OPT_POST_NO_SCHED

This option indicates that the caller does not want the scheduler to be called after the post. This option can be used in combination with one of the two previous options.

You should use this option if the task (or ISR) calling OSSemPost() will be doing additional posting and, the user does not want to reschedule until all done, and multiple posts are to take effect simultaneously.

p_err

is a pointer to a variable that holds an error code:

OS_ERR_NONE

If no tasks are waiting on the semaphore. In this case, the return value is also 0.

OS_ERR_OBJ_PTR_NULL

If OS_CFG_ARG_CHK_EN is set to DEF_ENABLED in os_cfg.h: if p_sem is a NULL pointer.

OS_ERR_OBJ_TYPE

If OS_CFG_OBJ_TYPE_CHK_EN is set to DEF_ENABLED in os_cfg.h: if p_sem is not pointing to a semaphore.

OS_ERR_OPT_INVALID

If OS_CFG_ARG_CHK_EN is set to DEF_ENABLED in os_cfg.h: if a valid option is not specified.

OS_ERR_OS_NOT_RUNNING

If OS_CFG_INVALID_OS_CALLS_CHK_EN is set to DEF_ENABLED in os_cfg.h: if µ

C/OS-III is not running yet.

`OS_ERR_SEM_OVF`

If the post would have caused the semaphore counter to overflow.

Returned Value

The current value of the semaphore count

Required Configuration

`OS_CFG_SEM_EN` must be enabled in `os_cfg.h`. Refer to µC-OS-III Configuration Manual.

Callers

Application and ISRs.

Notes/Warnings

1. Semaphores must be created before they are used.
2. You can also post to a semaphore from an ISR but the semaphore must be used as a signaling mechanism and not to protect a shared resource.

Example Usage

```
OS_SEM  SwSem;


          void  TaskX (void  *p_arg)
          {
              OS_ERR      err;
              OS_SEM_CTR  ctr;


              (void)&p_arg;
              while (DEF_ON) {
                  :
                  :
                  ctr = OSSemPost(&SwSem,
                                   OS_OPT_POST_1 + OS_OPT_POST_NO_SCHED,
                                  &err);
                  /* Check “err” */
                  :
                  :
              }
          }
```

Listing - OSSemPost() example usage

OSSemSet

Description

Changes the current value of the semaphore count. This function is normally selected 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, the count is only changed if there are no tasks waiting on the semaphore.

Files

os.h/os_sem.c

Prototype

```
void  OSSemSet (OS_SEM      *p_sem,
                OS_SEM_CTR   cnt,
                OS_ERR      *p_err)
```

Arguments

`p_sem`

is a pointer to the semaphore that is used as a signaling mechanism.

`cnt`

is the desired count that the semaphore should be set to.

`p_err`

is a pointer to a variable used to hold an error code:

`OS_ERR_NONE`

If the count was changed.

`OS_ERR_OBJ_PTR_NULL`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if `p_sem` is a `NULL` pointer.

`OS_ERR_OBJ_TYPE`

If `OS_CFG_OBJ_TYPE_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if `p_sem` is not pointing to a semaphore.

`OS_ERR_SET_ISR`

If `OS_CFG_CALLED_FROM_ISR_CHK_EN` set to `DEF_ENABLED` in `os_cfg.h`: if this function was called from an ISR.

`OS_ERR_TASK_WAITING`

If tasks are waiting on the semaphore, the count is not changed.

Returned Value

None

Required Configuration

`OS_CFG_SEM_EN` and `OS_CFG_SEM_SET_EN` must be enabled in `os_cfg.h`. Refer to µC-OS-III Configuration Manual.

Callers

Application.

Notes/Warnings

1. *Do not* use this function if the semaphore is used to protect a shared resource.

Example Usage

```
OS_SEM  SwSem;


          void  Task (void  *p_arg)
          {
              OS_ERR  err;


              (void)&p_arg;
              while (DEF_ON) {
                  OSSemSet(&SwSem,      /* Reset the semaphore count */
                            0,
                           &err);
                  /* Check “err” */
                  :
                  :
              }
          }
```

Listing - OSSemSet() example usage

API - Timers

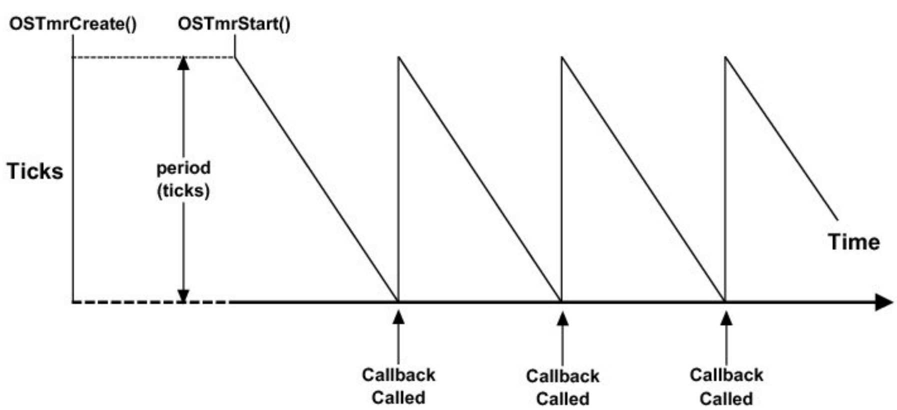
- OSTmrCreate()
- OSTmrDel()
- OSTmrRemainGet()
- OSTmrSet()
- OSTmrStart()
- OSTmrStateGet()
- OSTmrStop()

OSTmrCreate

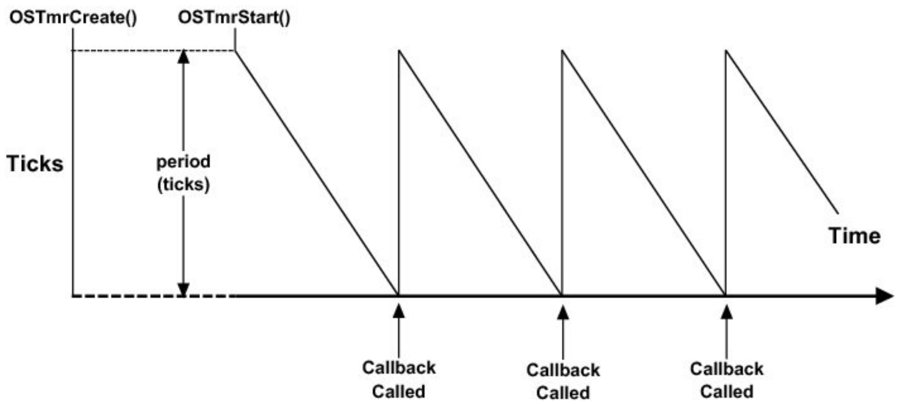
Description

`OSTmrCreate()` allows the user to create a software 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 is recommended to keep the callback function as short as possible.

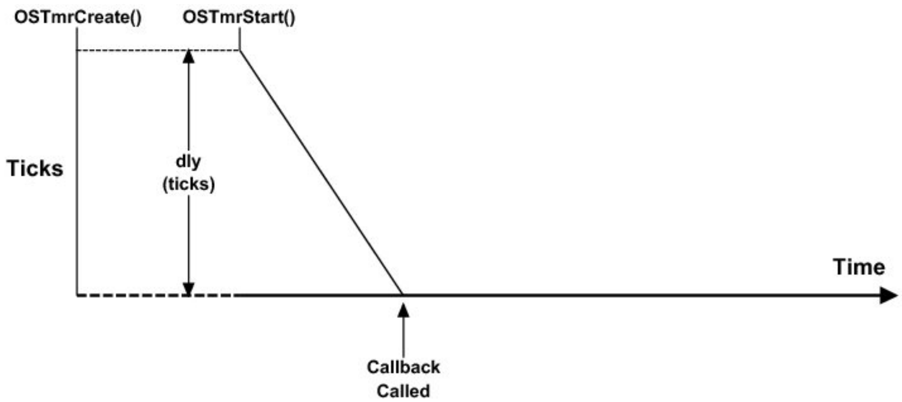
The timer is created in the "stop" mode and therefore the user *must* call `OSTmrStart()` to actually start the timer. If configuring the timer for ONE-SHOT mode, and the timer expires, you need to call `OSTmrStart()` to retrigger the timer, call `OSTmrDel()` to delete the timer if it is not necessary to retrigger it, or not use the timer anymore. Note: you can use the callback function to delete the timer if using the ONE-SHOT mode.



PERIODIC MODE (see "`opt`") – `dly` > 0, `period` > 0



PERIODIC MODE (see "`opt`") – "dly == 0, period > 0



ONE-SHOT MODE (see "`opt`") – `dly` > 0, `period` == 0

Files

os.h/os_tmr.c

Prototype

```
void  OSTmrCreate (OS_TMR               *p_tmr,
                   CPU_CHAR             *p_name,
                   OS_TICK               dly,
                   OS_TICK               period,
                   OS_OPT                opt,
                   OS_TMR_CALLBACK_PTR   p_callback,
                   void                 *p_callback_arg,
                   OS_ERR               *p_err)
```

Arguments

`p_tmr`

is a pointer to the timer-control block of the desired timer. It is assumed that storage for the timer will be allocated in the application. In other words, you should declare a "global" variable as follows, and pass a pointer to this variable to `OSTmrCreate()`:

```
OS_TMR  MyTmr;
```

`p_name`

is a pointer to an ASCII string (`NUL` terminated) used to assign a name to the timer. The name can be displayed by debuggers or µC/Probe.

`dly`

specifies the initial delay (specified in timer tick units) used by the timer (see drawing above). If the timer is configured for ONE-SHOT mode, this is the timeout used. If the timer is configured for PERIODIC mode, this is the timeout to wait before the timer enters periodic mode. The units of this time depends on how often the user will call `OSTmrSignal()` (see `OSTimeTick()`). If `OSTmrSignal()` is called every 1/10 of a second (i.e., `OS_CFG_TMR_TASK_RATE_HZ` set to 10), `dly` specifies the number of 1/10 of a second before the delay expires.

`period`

specifies the period repeated by the timer if configured for PERIODIC mode. You would set the "`period`" to 0 when using ONE-SHOT mode. The units of time depend on how often `OSTmrSignal()` is called. If `OSTmrSignal()` is called every 1/10 of a second (i.e., `OS_CFG_TMR_TASK_RATE_HZ` set to 10), the period specifies the number of 1/10 of a second before the timer repeats.

`opt`

is used to specify whether the timer is to be ONE-SHOT or PERIODIC:

`OS_OPT_TMR_ONE_SHOT` specifies ONE-SHOT mode
`OS_OPT_TMR_PERIODIC` specifies PERIODIC mode

`p_callback`

is a pointer to a function that will execute when the timer expires (ONE-SHOT mode), or every time the period expires (PERIODIC mode). A `NULL` pointer indicates that no action is to be performed upon timer expiration. The callback function must be declared as follows:

```
void MyCallback (OS_TMR *p_tmr, void *p_arg);
```

When called, the callback will be passed the pointer to the timer as well as an argument (`p_callback_arg`), which can be used to indicate to the callback what to do. Note that the user is allowed to call all of the timer related functions (i.e., `OSTmrCreate()`, `OSTmrDel()`, `OSTmrStateGet()`, `OSTmrRemainGet()`, `OSTmrStart()`, and `OSTmrStop()`) from the callback function.

Do not make blocking calls within callback functions.

`p_callback_arg`

is an argument passed to the callback function when the timer expires (ONE-SHOT mode), or every time the period expires (PERIODIC mode). The pointer is declared as a "`void *`" so it can point to any data.

`p_err`

is a pointer to a variable that contains an error code returned by this function.

`OS_ERR_NONE`

If the call was successful.

`OS_ERR_ILLEGAL_CREATE_RUN_TIME`

If `OS_SAFETY_CRITICAL_IEC61508` is defined: you called this after calling `OSStart()` and thus you are no longer allowed to create additional kernel objects.

`OS_ERR_OBJ_PTR_NULL`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if `p_tmr` is a `NULL` pointer.

`OS_ERR_OPT_INVALID`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if not specifying a valid option.

`OS_ERR_TMR_INVALID_CALLBACK`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if `p_callback` is a `NULL` pointer.

`OS_ERR_TMR_INVALID_DLY`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if specifying an invalid delay in ONE-SHOT mode. In other words, it is not allowed to delay for `0` in ONE-SHOT mode.

`OS_ERR_TMR_INVALID_PERIOD`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if specifying an invalid period in PERIODIC mode. It is not allowed to have a `0` period in PERIODIC.

`OS_ERR_TMR_ISR`

If `OS_CFG_CALLED_FROM_ISR_CHK_EN` set to `DEF_ENABLED` in `os_cfg.h`: if calling this function from an ISR.

Returned Values

None.

Required Configuration

`OS_CFG_TMR_EN` must be enabled in `os_cfg.h`. Refer to µC-OS-III Configuration Manual.

Callers

Application.

Notes/Warnings

1. *Do not* call this function from an ISR.
2. The timer is *not* started when it is created. To start the timer, simply call `OSTmrStart()`.
3. *Do not* make blocking calls within callback functions.
4. Keep callback functions as short as possible.

Example Usage

```
OS_TMR  CloseDoorTmr;


          void Task (void *p_arg)
          {
              OS_ERR   err;


              (void)&p_arg;
              while (DEF_ON) {
                  OSTmrCreate(&CloseDoorTmr,         /* p_tmr          */
                             “Door close”            /* p_name         */
                              10,                    /* dly            */
                              100,                   /* period         */
                              OS_OPT_TMR_PERIODIC,   /* opt            */
                              DoorCloseFnct,         /* p_callback     */
                              0,                     /* p_callback_arg */
                             &err);                  /* p_err          */
                  /* Check “err” */
                  :
                  :
              }
          }


          void  DoorCloseFnct (OS_TMR  *p_tmr,
                               void    *p_arg)
          {
              /* Close the door! */
          }
```

OSTmrDel

Description

`OSTmrDel()` allows the user 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 therefore stopped, it will simply be deleted.

It is up to the user to delete unused timers. If deleting a timer, you must not reference it again.

Files

os.h/os_tmr.c

Prototype

```
CPU_BOOLEAN  OSTmrDel (OS_TMR  *p_tmr,
                       OS_ERR  *p_err)
```

Arguments

`p_tmr`

is a pointer to the timer to be deleted.

`p_err`

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

`OS_ERR_NONE`

If the timer was deleted.

`OS_ERR_ILLEGAL_DEL_RUN_TIME`

If `OS_SAFETY_CRITICAL_IEC61508` is defined: you called this after calling `OSStart()` and thus you are no longer allowed to delete kernel objects.

`OS_ERR_OBJ_TYPE`

If OS_CFG_OBJ_TYPE_CHK_EN is set to DEF_ENABLED in os_cfg.h: if the user did not pass a pointer to a timer.

OS_ERR_OS_NOT_RUNNING

If OS_CFG_INVALID_OS_CALLS_CHK_EN is set to DEF_ENABLED in os_cfg.h: if µC/OS-III is not running yet.

OS_ERR_TMR_INACTIVE

p_tmr is pointing to an inactive timer. In other words, this error appears when pointing to a timer that has been deleted.

OS_ERR_TMR_INVALID

If OS_CFG_ARG_CHK_EN is set to DEF_ENABLED in os_cfg.h: if p_tmr is a NULL pointer.

OS_ERR_TMR_INVALID_STATE

If the timer is in an invalid state.

OS_ERR_TMR_ISR

If OS_CFG_CALLED_FROM_ISR_CHK_EN set to DEF_ENABLED in os_cfg.h: This function is called from an ISR, which is *not* allowed.

Returned Values

DEF_TRUE if the timer was deleted, DEF_FALSE if not or an error occurred.

Required Configuration

OS_CFG_TMR_EN and OS_CFG_TMR_DEL_EN must be enabled in os_cfg.h. Refer to µC-OS-III Configuration Manual.

Callers

Application.

Notes/Warnings

1. *Do not* call this function from an ISR.
2. When deleting a timer, *do not* reference it again unless you re-create the timer by calling `OSTmrCreate()`.

Example Usage

```
OS_TMR  CloseDoorTmr;


          void Task (void *p_arg)
          {
              OS_ERR       err;
              CPU_BOOLEAN  deleted;


              (void)&p_arg;
              while (DEF_ON) {
                  deleted = OSTmrDel(&CloseDoorTmr,
                                     &err);
                  /* Check “err” */
                  :
                  :
              }
          }
```

OSTmrRemainGet

Description

`OSTmrRemainGet()` allows the user to obtain the time remaining (before timeout) of the specified timer. The value returned depends on the rate (in Hz) at which the timer task is signaled (see `OS_CFG_TMR_TASK_RATE_HZ`). If `OS_CFG_TMR_TASK_RATE_HZ` is set to 10, 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 is 0.

Files

os.h/os_tmr.c

Prototype

```
OS_TICK  OSTmrRemainGet (OS_TMR  *p_tmr,
                         OS_ERR  *p_err)
```

Arguments

`p_tmr`

is a pointer to the timer the user is inquiring about.

`p_err`

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_OBJ_TYPE`

If `OS_CFG_OBJ_TYPE_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: ‘`p_tmr`” is not pointing to a timer.

`OS_ERR_OS_NOT_RUNNING`

If `OS_CFG_INVALID_OS_CALLS_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if µC/OS-III is not running yet.

`OS_ERR_TMR_INACTIVE`

`p_tmr` is pointing to an inactive timer. In other words, this error will appear when pointing to a timer that has been deleted.

`OS_ERR_TMR_INVALID`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if `p_tmr` is a `NULL` pointer.

`OS_ERR_TMR_INVALID_STATE`

If the timer is in an invalid state.

`OS_ERR_TMR_ISR`

If `OS_CFG_CALLED_FROM_ISR_CHK_EN` set to `DEF_ENABLED` in `os_cfg.h`: This function is called from an ISR, which is *not* allowed.

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_CFG_TMR_TASK_RATE_HZ`). If `OS_CFG_TMR_TASK_RATE_HZ` is set to 10 the value returned is the number of 1/10 of a second before the timer times out. If specifying an invalid timer, the returned value will be 0. If the timer expired, the returned value will be 0.

Required Configuration

`OS_CFG_TMR_EN` must be enabled in `os_cfg.h`. Refer to µC-OS-III Configuration Manual.

Callers

Application.

Notes/Warnings

1. *Do not* call this function from an ISR.

Example Usage

```
OS_TICK   TimeRemainToCloseDoor;
          OS_TMR    CloseDoorTmr;


          void Task (void *p_arg)
          {
              OS_ERR    err;


              (void)&p_arg;
              while (DEF_ON) {
                  TimeRemainToCloseDoor = OSTmrRemainGet(&CloseDoorTmr,
                                                         &err);
                  /* Check "err" */
                  :
                  :
              }
          }
```

OSTmrSet

Description

`OSTmrSet()` allows the user to change the period, delay and callback parameters of an already existing timer. For more information, see `OSTmrCreate()`.

Files

os.h/os_tmr.c

Prototype

```
void  OSTmrSet (OS_TMR                *p_tmr,
                OS_TICK                dly,
                OS_TICK                period,
                OS_TMR_CALLBACK_PTR    p_callback,
                void                  *p_callback_arg,
                OS_ERR                *p_err)
```

Arguments

`p_tmr`

is a pointer to the timer-control block of the desired timer. It is assumed that storage for the timer will be allocated in the application and that the timer has already been initialized by `OSTmrCreate()`.

`dly`

specifies the delay (specified in timer tick units) used by the timer. If the timer is configured for ONE-SHOT mode, this is the timeout used. If the timer is configured for PERIODIC mode, this is the timeout to wait before the timer enters periodic mode. The units of this time depends on how often the user will call `OSTmrSignal()` (see `OSTimeTick()`). If `OSTmrSignal()` is called every 1/10 of a second (i.e., `OS_CFG_TMR_TASK_RATE_HZ` set to 10), `dly` specifies the number of 1/10 of a second before the delay expires.

`period`

specifies the period repeated by the timer if configured for PERIODIC mode. You would set the “`period`” to 0 when using ONE-SHOT mode. The units of time depend on how often `OSTmrSignal()` is called. If `OSTmrSignal()` is called every 1/10 of a second (i.e., `OS_CFG_TMR_TASK_RATE_HZ` set to 10), the period specifies the number of 1/10 of a second before the timer repeats.

`p_callback`

is a pointer to a function that will execute when the timer expires (ONE-SHOT mode), or every time the period expires (PERIODIC mode). A `NULL` pointer indicates that no action is to be performed upon timer expiration. The callback function must be declared as follows:

```
void MyCallback (OS_TMR *p_tmr, void *p_arg);
```

When called, the callback will be passed the pointer to the timer as well as an argument (`p_callback_arg`), which can be used to indicate to the callback what to do. Note that the user is allowed to call all of the timer related functions (i.e., `OSTmrCreate()`, `OSTmrDel()` , `OSTmrStateGet()`,`OSTmrRemainGet()`, `OSTmrStart()`, and `OSTmrStop()`) from the callback function.

Do not make blocking calls within callback functions.

`p_callback_arg`

is an argument passed to the callback function when the timer expires (ONE-SHOT mode), or every time the period expires (PERIODIC mode). The pointer is declared as a “ `void *`” so it can point to any data.

`p_err`

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

`OS_ERR_NONE`

The timer was configured as expected.

`OS_ERR_OBJ_TYPE`

If `OS_CFG_OBJ_TYPE_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: ‘`p_tmr`” is not pointing to a timer.

`OS_ERR_OS_NOT_RUNNING`

If `OS_CFG_INVALID_OS_CALLS_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if µC/OS-III is not running yet.

`OS_ERR_TMR_INVALID`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if `p_tmr` is a `NULL` pointer or possess an invalid option.

`OS_ERR_TMR_INVALID_CALLBACK`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if `p_callback` is a `NULL` pointer.

`OS_ERR_TMR_INVALID_DLY`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if specifying an invalid delay in ONE-SHOT mode. In other words, it is not allowed to delay for `0` in ONE-SHOT mode.

`OS_ERR_TMR_INVALID_PERIOD`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if specifying an invalid period in PERIODIC mode. It is not allowed to have a `0` period in PERIODIC.

`OS_ERR_TMR_ISR`

If `OS_CFG_CALLED_FROM_ISR_CHK_EN` set to `DEF_ENABLED` in `os_cfg.h`: This function is called from an ISR, which is *not* allowed.

Returned Values

None

Required Configuration

`OS_CFG_TMR_EN` must be enabled in `os_cfg.h`. Refer to µC-OS-III Configuration Manual.

Callers

Application.

Notes/Warnings

1. *Do not* call this function from an ISR.

Example Usage

```
OS_TMR    CloseDoorTmr;


          void Task (void *p_arg)
          {
              OS_ERR    err;


              (void)&p_arg;
              while (DEF_ON) {
                  OSTmrSet(&CloseDoorTmr,          /* p_tmr          */
                            10,                    /* dly            */
                            100,                   /* period         */
                            DoorCloseFnct,         /* p_callback     */
                            0,                     /* p_callback_arg */
                           &err);                  /* p_err          */
                  /* Check “err” */
                  :
                  :
              }
          }


          void  DoorCloseFnct (OS_TMR  *p_tmr,
                               void    *p_arg)
          {
              /* Close the door! */
          }
```

OSTmrStart

Description

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

Files

os.h/os_tmr.c

Prototype

```
CPU_BOOLEAN  OSTmrStart (OS_TMR  *p_tmr,
                         OS_ERR  *p_err)
```

Arguments

`p_tmr`

is a pointer to the timer to start (or restart).

`p_err`

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

`OS_ERR_NONE`

If the timer was started.

`OS_ERR_OBJ_TYPE`

If `OS_CFG_OBJ_TYPE_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: ‘`p_tmr`” is not pointing to a timer.

`OS_ERR_OS_NOT_RUNNING`

If `OS_CFG_INVALID_OS_CALLS_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if µ

C/OS-III is not running yet.

OS_ERR_TMR_INACTIVE

p_tmr is pointing to an inactive timer. In other words, this error occurs if pointing to a timer that has been deleted or was not created.

OS_ERR_TMR_INVALID

If OS_CFG_ARG_CHK_EN is set to DEF_ENABLED in os_cfg.h: if p_tmr is a NULL pointer.

OS_ERR_TMR_INVALID_STATE

If the timer is in an invalid state.

OS_ERR_TMR_ISR

If OS_CFG_CALLED_FROM_ISR_CHK_EN set to DEF_ENABLED in os_cfg.h: This function was called from an ISR, which is *not* allowed.

Returned Values

DEF_TRUE

If the timer was started.

DEF_FALSE

If an error occurred.

Required Configuration

OS_CFG_TMR_EN must be enabled in os_cfg.h. Refer to µC-OS-III Configuration Manual.

Callers

Application.

Notes/Warnings

1. *Do not* call this function from an ISR.
2. The timer *must* have previously been created.

Example Usage

```
OS_TMR      CloseDoorTmr;


          void Task (void *p_arg)
          {
              OS_ERR       err;
              CPU_BOOLEAN  status;


              (void)&p_arg;
              while (DEF_ON) {
                  status = OSTmrStart(&CloseDoorTmr,
                                      &err);
                  /* Check “err” */
                  :
                  :
              }
          }
```

OSTmrStateGet

Description

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

`OS_TMR_STATE_UNUSED`

the timer has not been created

`OS_TMR_STATE_STOPPED`

the timer is created but has not yet 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

Files

os.h/os_tmr.c

Prototype

```
OS_STATE  OSTmrStateGet (OS_TMR  *p_tmr,
                         OS_ERR  *p_err)
```

Arguments

`p_tmr`

is a pointer to the timer that the user is inquiring about.

`p_err`

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

`OS_ERR_NONE`

If the function returned the state of the timer.

`OS_ERR_OBJ_TYPE`

If `OS_CFG_OBJ_TYPE_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: `p_tmr` is not pointing to a timer.

`OS_ERR_OS_NOT_RUNNING`

If `OS_CFG_INVALID_OS_CALLS_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if µC/OS-III is not running yet.

`OS_ERR_TMR_INVALID`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if `p_tmr` is a `NULL` pointer.

`OS_ERR_TMR_INVALID_STATE`

If the timer is in an invalid state.

`OS_ERR_TMR_ISR`

If `OS_CFG_CALLED_FROM_ISR_CHK_EN` set to `DEF_ENABLED` in `os_cfg.h`: This function was called from an ISR, which is *not* allowed.

Returned Values

The state of the timer (see description).

Required Configuration

`OS_CFG_TMR_EN` must be enabled in `os_cfg.h`. Refer to µC-OS-III Configuration Manual.

Callers

Application.

Notes/Warnings

1. *Do not* call this function from an ISR.

Example Usage

```
OS_STATE   CloseDoorTmrState;
          OS_TMR     CloseDoorTmr;


          void Task (void *p_arg)
          {
              OS_ERR  err;


              (void)&p_arg;
              while (DEF_ON) {
                  CloseDoorTmrState = OSTmrStateGet(&CloseDoorTmr,
                                                    &err);
                  /* Check "err" */
                  :
                  :
              }
          }
```

OSTmrStop

Description

`OSTmrStop()` allows the user to stop a timer. The user may execute the callback function of the timer when it is stopped, and pass this callback function a different argument than was specified when the timer was started. This allows the callback function to know that the timer was stopped since the callback argument can be set to indicate this (this is application specific). If the timer is already stopped, the callback function is not called.

Files

os.h/os_tmr.c

Prototype

```
CPU_BOOLEAN  OSTmrStop (OS_TMR  *p_tmr,
                        OS_OPT   opt,
                        void    *p_callback_arg,
                        OS_ERR  *p_err)
```

Arguments

`p_tmr`

is a pointer to the timer control block of the desired timer.

`opt`

is used to specify options:

`OS_OPT_TMR_NONE`

No option

`OS_OPT_TMR_CALLBACK`

Run the callback function with the argument specified when the timer was created.

OS_OPT_TMR_CALLBACK_ARG

Run the callback function, but use the argument passed in OSTmrStop() instead of the one specified when the task was created.

p_callback_arg

is a new argument to pass the callback functions (see options above).

p_err

is a pointer to a variable that contains an error code returned by this function.

OS_ERR_NONE

If the call was successful.

OS_ERR_OBJ_TYPE

If OS_CFG_OBJ_TYPE_CHK_EN is set to DEF_ENABLED in os_cfg.h: if p_tmr is not pointing to a timer object.

OS_ERR_OPT_INVALID

If OS_CFG_ARG_CHK_EN is set to DEF_ENABLED in os_cfg.h: if a valid option is not specified.

OS_ERR_OS_NOT_RUNNING

If OS_CFG_INVALID_OS_CALLS_CHK_EN is set to DEF_ENABLED in os_cfg.h: if µC/OS-III is not running yet.

OS_ERR_TMR_INACTIVE

If the timer cannot be stopped since it is inactive.

OS_ERR_TMR_INVALID

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if you passed a `NULL` pointer for the `p_tmr` argument.

`OS_ERR_TMR_INVALID_STATE`

If the timer is in an invalid state.

`OS_ERR_TMR_ISR`

If `OS_CFG_CALLED_FROM_ISR_CHK_EN` set to `DEF_ENABLED` in `os_cfg.h`: if calling this function from an ISR.

`OS_ERR_TMR_NO_CALLBACK`

If the timer lacks a callback function. This should have been specified when the timer was created.

`OS_ERR_TMR_STOPPED`

If the timer is currently stopped.

Returned Values

`DEF_TRUE`

If the timer was stopped (even if it was already stopped).

`DEF_FALSE`

If an error occurred.

Required Configuration

`OS_CFG_TMR_EN` must be enabled in `os_cfg.h`. Refer to µC-OS-III Configuration Manual.

Callers

Application.

Notes/Warnings

1. *Do not* call this function from an ISR.

2. The callback function is *not* called if the timer is already stopped.

Example Usage

```
OS_TMR  CloseDoorTmr;


          void Task (void *p_arg)
          {
              OS_ERR       err;
              CPU_BOOLEAN  is_stopped;


              (void)&p_arg;
              while (DEF_ON) {
                  is_stopped = OSTmrStop(&CloseDoorTmr,
                                          OS_TMR_OPT_CALLBACK,
                                         (void *)0,
                                         &err);
                  /* Check “err” */
                  :
                  :
              }
          }
```

API - Task Semaphores

OSTaskSemPend

Description

`OSTaskSemPend()` allows a task to wait for a signal to be sent by another task or ISR without going through an intermediate object such as a semaphore. If the task was previously signaled when `OSTaskSemPend()` is called then, the caller resumes.

If no signal was received by the task and `OS_OPT_PEND_BLOCKING` is specified for the `opt` argument, `OSTaskSemPend()` suspends the current task until either a signal is received, or a user-specified timeout expires. A pended task suspended with `OSTaskSuspend()` can receive signals. However, the task remains suspended until it is resumed by calling `OSTaskResume()`.

If no signals were sent to the task and `OS_OPT_PEND_NON_BLOCKING` was specified for the `opt` argument, `OSTaskSemPend()` returns to the caller with an appropriate error code and returns a signal count of `0`.

Files

os.h/os_task.c

Prototype

```
OS_SEM_CTR  OSTaskSemPend (OS_TICK   timeout,
                           OS_OPT    opt,
                           CPU_TS   *p_ts,
                           OS_ERR   *p_err)
```

Arguments

`timeout`

allows the task to resume execution if a signal is not received from a task or an ISR within the specified number of clock ticks. A timeout value of `0` indicates that the task wants to wait forever for a signal. 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.

`opt`

determines whether the user wants to block or not, if a signal was not sent to the task. Set this argument to either:

`OS_OPT_PEND_BLOCKING`, or

`OS_OPT_PEND_NON_BLOCKING`

Note that the `timeout` argument should be set to `0` when specifying `OS_OPT_PEND_NON_BLOCKING`, since the timeout value is irrelevant using this option.

`p_ts`

is a pointer to a timestamp indicating when the task's semaphore was posted, or the pend was aborted. Passing a `NULL` pointer is valid and indicates that the timestamp is not necessary.

A timestamp is useful when the task is to know when the semaphore was posted, or how long it took for the task to resume after the semaphore was posted. In the latter case, call `OS_TS_GET()` and compute the difference between the current value of the timestamp and `*p_ts`. In other words:

```
delta = OS_TS_GET() - *p_ts;
```

`p_err`

is a pointer to a variable used to hold an error code.

`OS_ERR_NONE`

If a signal is received.

`OS_ERR_OPT_INVALID`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if a valid option is not specified.

`OS_ERR_OS_NOT_RUNNING`

If `OS_CFG_INVALID_OS_CALLS_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if µC/OS-III is not running yet.

`OS_ERR_PEND_ABORT`

If the pend was aborted because another task called `OSTaskSemPendAbort()`.

`OS_ERR_PEND_ISR`

If `OS_CFG_CALLED_FROM_ISR_CHK_EN` set to `DEF_ENABLED` in `os_cfg.h`: if calling this function from an ISR.

`OS_ERR_PEND_WOULD_BLOCK`

If calling this function with the `opt` argument set to `OS_OPT_PEND_NON_BLOCKING`, and no signal was received.

`OS_ERR_SCHED_LOCKED`

If calling this function when the scheduler is locked and the user wanted the task to block.

`OS_ERR_STATUS_INVALID`

If the pend status has an invalid value.

`OS_ERR_TIMEOUT`

If a signal is not received within the specified timeout.

Returned Value

The current value of the signal counter after it has been decremented. In other words, the number of signals still remaining in the signal counter.

Required Configuration

Always enabled.

Callers

Application.

Notes/Warnings

1. Do not call `OSTaskSemPend()` from an ISR.

Example Usage

```
void CommTask(void *p_arg)
          {
              OS_ERR      err;
              OS_SEM_CTR  ctr;
              CPU_TS      ts;


              (void)&p_arg;
              while (DEF_ON) {
                  :
                  ctr = OSTaskSemPend(100,
                                      OS_OPT_PEND_BLOCKING,
                                      &ts,
                                      &err);
                  /* Check “err” */
                  :
                  :
              }
          }
```

OSTaskSemPendAbort

Description

`OSTaskSemPendAbort()` aborts and readies a task currently waiting on its built-in semaphore. This function should be used to fault-abort the wait on the task's semaphore, rather than to normally signal the task via `OSTaskSemPost()`.

Files

os.h/os_task.c

Prototype

```
CPU_BOOLEAN  OSTaskSemPendAbort (OS_TCB  *p_tcb,
                                 OS_OPT   opt,
                                 OS_ERR  *p_err)
```

Arguments

`p_tcb`

is a pointer to the task for which the pend must be aborted. Note that it does not make sense to pass a `NULL` pointer or the address of the calling task's TCB since, by definition, the calling task cannot be pending.

`opt`

provides options for this function.

`OS_OPT_POST_NONE`

no option specified, call the scheduler by default.

`OS_OPT_POST_NO_SCHED`

specifies that the scheduler should not be called even if the pend of a higher-priority task has been aborted. Scheduling will need to occur from another function.

Use this option if the task calling `OSTaskSemPendAbort()` will be doing additional pend aborts, rescheduling will not take place until finished, and multiple pend aborts are to take effect simultaneously.

`p_err`

is a pointer to a variable that holds an error code:

`OS_ERR_NONE`

the pend was aborted for the specified task.

`OS_ERR_OPT_INVALID`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if a valid option is not specified.

`OS_ERR_OS_NOT_RUNNING`

If `OS_CFG_INVALID_OS_CALLS_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if µ C/OS-III is not running yet.

`OS_ERR_PEND_ABORT_ISR`

If `OS_CFG_CALLED_FROM_ISR_CHK_EN` set to `DEF_ENABLED` in `os_cfg.h`: if called from an ISR.

`OS_ERR_PEND_ABORT_NONE`

If the task was not waiting for a signal.

`OS_ERR_PEND_ABORT_SELF`

If `p_tcb` is a `NULL` pointer or the TCB of the calling task is specified. The user is attempting to pend abort the calling task, which makes no sense since, by definition, the calling task is not pending.

Returned Value

`OSTaskSemPendAbort()` returns `DEF_TRUE` if the task was made ready-to-run by this function. `DEF_FALSE` indicates that the task was not pending, or an error occurred.

Required Configuration

`OS_CFG_TASK_SEM_PEND_ABORT_EN` must be enabled in `os_cfg.h`. Refer to µC-OS-III Configuration Manual.

Callers

Application.

Notes/Warnings

1. Do not call `OSTaskSemPendAbort()` from an ISR.

Example Usage

```
OS_TCB  CommRxTaskTCB;


          void CommTask (void *p_arg)
          {
              OS_ERR      err;
              CPU_BOOLEAN  aborted;


              (void)&p_arg;
              while (DEF_ON) {
                  :
                  :
                  aborted = OSTaskSemPendAbort(&CommRxTaskTCB,
                                                OS_OPT_POST_NONE,
                                               &err);
                  /* Check “err” */
                  :
                  :
              }
          }
```

OSTaskSemPost

Description

`OSTaskSemPost()` sends a signal to a task through it's local semaphore.

If the task receiving the signal is actually waiting for a signal to be received, it will be made ready-to-run and, if the receiving task has a higher priority than the task sending the signal, the higher-priority task resumes, and the task sending the signal is suspended; that is, a context switch occurs. Note that scheduling only occurs if `opt` is set to `OS_OPT_POST_NONE`, because the `OS_OPT_POST_NO_SCHED` option does not cause the scheduler to be called.

Files

os.h/os_task.c

Prototype

```
OS_SEM_CTR  OSTaskSemPost (OS_TCB    *p_tcb,
                           OS_OPT     opt,
                           OS_ERR    *p_err)
```

Arguments

`p_tcb`

is a pointer to the TCB of the task being signaled. A `NULL` pointer indicates that the user is sending a signal to itself.

`opt`

provides options to the call.

`OS_OPT_POST_NONE`

No option, by default the scheduler will be called.

`OS_OPT_POST_NO_SCHED`

Do not call the scheduler after the post, therefore the caller is resumed.

You would use this option if the task (or ISR) calling `OSTaskSemPost()` will be doing additional posts, reschedule waits until all is done, and multiple posts are to take effect simultaneously.

`p_err`

is a pointer to a variable that will contain an error code returned by this function.

`OS_ERR_NONE`

If the call was successful and the signal was sent.

`OS_ERR_OPT_INVALID`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if a valid option is not specified.

`OS_ERR_OS_NOT_RUNNING`

If `OS_CFG_INVALID_OS_CALLS_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if µC/OS-III is not running yet.

`OS_ERR_SEM_OVF`

the post would have caused the semaphore counter to overflow.

`OS_ERR_STATE_INVALID`

If the task is in an invalid state.

Returned Value

The current value of the task's signal counter, or `0` if called from an ISR and `OS_CFG_ISR_POST_DEFERRED_EN` is set to `DEF_ENABLED`.

Required Configuration

Always enabled.

Callers

Application and ISRs.

Notes/Warnings

1. Do not call `OSTaskSemPost()` from an ISR.

Example Usage

```
OS_TCB       CommRxTaskTCB;


          void CommTaskRx (void *p_arg)
          {
              OS_ERR      err;
              OS_SEM_CTR  ctr;


              (void)&p_arg;
              while (DEF_ON) {
                  :
                  ctr = OSTaskSemPost(&CommRxTaskTCB,
                                       OS_OPT_POST_NONE,
                                      &err);
                  /* Check "err" */
                  :
                  :
              }
          }
```


OSTaskSemSet

Description

`OSTaskSemSet()` allows the user to set the value of the task's signal counter. You would set the signal counter of the calling task by passing a `NULL` pointer for `p_tcb`.

Files

os.h/os_task.c

Prototype

```
OS_SEM_CTR  OSTaskSemSet (OS_TCB      *p_tcb,
                          OS_SEM_CTR   cnt;
                          OS_ERR      *p_err)
```

Arguments

`p_tcb`

is a pointer to the task's `OS_TCB` to clear the signal counter. A `NULL` pointer indicates that the user wants to clear the caller's signal counter.

`cnt`

the desired value for the task semaphore counter.

`p_err`

is a pointer to a variable that will contain an error code returned by this function.

`OS_ERR_NONE`

If the call was successful and the signal counter was set.

`OS_ERR_SET_ISR`

If `OS_CFG_CALLED_FROM_ISR_CHK_EN` set to `DEF_ENABLED` in `os_cfg.h`: if calling this function from an ISR.

`OS_ERR_TASK_WAITING`

If tasks are waiting on the semaphore, the count is not changed.

Returned Value

The new value of the signal counter.

Required Configuration

Always enabled.

Callers

Application.

Notes/Warnings

1. Do not call `OSTaskSemSet()` from an ISR.

Example Usage

```
OS_TCB   TaskY;


          void TaskX (void *p_arg)
          {
              OS_ERR      err;
              OS_SEM_CTR  ctr;


              while (DEF_ON) {
                  :
                  :
                  ctr = OSTaskSemSet(&TaskY,
                                     0,
                                    &err);
                  /* Check “err” */
                  :
                  :
              }
          }
```

API - Fixed-Size Memory Partitions

- OSMemCreate()
- OSMemGet()
- OSMemPut()

OSMemCreate

Description

Creates and initializes a memory partition. A memory partition contains a user-specified number of fixed-size memory blocks. An application may obtain one of these memory blocks and, when completed, release the block back to the same partition where the block originated.

Files

os.h/os_mem.c

Prototype

```
void  OSMemCreate (OS_MEM       *p_mem,
                   CPU_CHAR     *p_name,
                   void         *p_addr,
                   OS_MEM_QTY    n_blks,
                   OS_MEM_SIZE   blk_size,
                   OS_ERR       *p_err)
```

Arguments

`p_mem`

is a pointer to a memory partition control block that must be allocated in the application. It is assumed that storage will be allocated for the memory control blocks in the application. In other words, the user will declare a “global” variable as follows, and pass a pointer to this variable to `OSMemCreate()`:

```
OS_MEM MyMemPartition;
```

`p_name`

is a pointer to an ASCII string to provide a name to the memory partition. The name can be displayed by debuggers or µC/Probe.

`p_addr`

is the address of the start of a memory area used to create fixed-size memory blocks. Memory partitions may be created using either static arrays or `malloc()` during startup. Note that the partition *must* align on a pointer boundary. Thus, if a pointer is 16-bits wide. 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, the partition must start on a memory location with an address that ends in 0, 4, 8 or C. The easiest way to ensure this is to create a static array as follows:

```
void *MyMemArray[N][M]
```

You should never deallocate memory blocks that were allocated from the heap to prevent fragmentation of your heap. It is quite acceptable to allocate memory blocks from the heap as long as the user does not deallocate them.

`n_blks`

contains the number of memory blocks available from the specified partition. You need to specify at least two memory blocks per partition.

`blk_size`

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. 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).

`p_err`

is a pointer to a variable that holds an error code:

`OS_ERR_NONE`

If the memory partition is created successfully

`OS_ERR_ILLEGAL_CREATE_RUN_TIME`

If `OS_SAFETY_CRITICAL_IEC61508` is defined: you called this after calling `OSStart()` and thus you are no longer allowed to create additional kernel objects.

`OS_ERR_MEM_CREATE_ISR`

If `OS_CFG_CALLED_FROM_ISR_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if you called `OSMemCreate()` from an ISR.

`OS_ERR_MEM_INVALID_BLKS`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if the user does not specify at least two memory blocks per partition

`OS_ERR_MEM_INVALID_P_ADDR`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if specifying an invalid address (i.e., `p_addr` is a `NULL` pointer) or the partition is not properly aligned.

`OS_ERR_MEM_INVALID_SIZE`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if the user does not specify a block size that can contain at least a pointer variable, and if it is not a multiple of a pointer-size variable.

Returned Value

None

Required Configuration

`OS_CFG_MEM_EN` must be enabled in `os_cfg.h`. Refer to µC-OS-III Configuration Manual.

Callers

Application.

Notes/Warnings

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

Example Usage

```
OS_MEM       CommMem;
          CPU_INT32U  *CommBuf[16][32];          /* 16 buffers of 32 words of 32 bits */


          void  main (void)
          {
              OS_ERR  err;


              OSInit(&err);                      /* Initialize µC/OS-III             */
              :
              :
              OSMemCreate(&CommMem,
                          “Comm Buffers”,
                          &CommBuf[0][0],
                          16,
                          32 * sizeof(CPU_INT32U),
                          &err);
              /* Check “err” */
              :
              :
              OSStart(&err);                     /* Start Multitasking              */
          }
```

Listing - OSMemCreate() example usage

OSMemGet

Description

Obtains a memory block from a memory partition. It is assumed that the application knows the size of each memory block obtained. Also, the application must return the memory block [using `OSMemPut()`] to the same memory partition when it no longer requires it. `OSMemGet()` may be called more than once until all memory blocks are allocated.

Files

os.h/os_mem.c

Prototype

```
void  *OSMemGet (OS_MEM  *p_mem,
                 OS_ERR  *p_err)
```

Arguments

`p_mem`

is a pointer to the desired memory partition control block.

`p_err`

is a pointer to a variable that holds an error code:

`OS_ERR_NONE`

If a memory block is available and returned to the application.

`OS_ERR_MEM_INVALID_P_MEM`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if `p_mem` is a `NULL` pointer.

`OS_ERR_MEM_NO_FREE_BLKS`

If the memory partition does not contain additional memory blocks to allocate.

OS_ERR_OBJ_TYPE

If OS_CFG_OBJ_TYPE_CHK_EN is set to DEF_ENABLED in os_cfg.h: if the user did not pass a pointer to a memory partition.

Returned Value

OSMemGet() returns a pointer to the allocated memory block if one is available. If a memory block is not available from the memory partition, OSMemGet() returns a NULL pointer. It is up to the application to “cast” the pointer to the proper data type since OSMemGet() returns a void *.

Required Configuration

OS_CFG_MEM_EN must be enabled in os_cfg.h. Refer to µC-OS-III Configuration Manual,

Callers

Application and ISRs.

Notes/Warnings

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

Example Usage

```
OS_MEM  CommMem;


          void Task (void *p_arg)
          {
              OS_ERR       err;
              CPU_INT32U  *p_msg;


              (void)&p_arg;
              while (DEF_ON) {
                  p_msg = (CPU_INT32U *)OSMemGet(&CommMem,
                                                 &err);
                  /* Check “err” */
                  :
                  :
              }
          }
```

Listing - OSMemGet() example usage

OSMemPut

Description

Returns a memory block back to a memory partition. It is assumed that the user will return the memory block to the same memory partition from which it was allocated.

Files

os.h/os_mem.c

Prototype

```
void  OSMemPut (OS_MEM  *p_mem,
                void    *p_blk,
                OS_ERR  *p_err)
```

Arguments

`p_mem`

is a pointer to the memory partition control block.

`p_blk`

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

`p_err`

is a pointer to a variable that holds an error code:

`OS_ERR_NONE`

If a memory block is available and returned to the application.

`OS_ERR_MEM_FULL`

If returning a memory block to an already full memory partition. This would indicate that the user freed more blocks that were allocated and potentially did not return some of the memory blocks to the proper memory partition.

`OS_ERR_MEM_INVALID_P_BLK`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if the user passed a `NULL` pointer for the memory block being returned to the memory partition.

`OS_ERR_MEM_INVALID_P_MEM`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if `p_mem` is a `NULL` pointer.

`OS_ERR_OBJ_TYPE`

If `OS_CFG_OBJ_TYPE_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if the user did not pass a pointer to a memory partition.

Returned Value

None

Required Configuration

`OS_CFG_MEM_EN` must be enabled in `os_cfg.h`. Refer to µC-OS-III Configuration Manual.

Callers

Application and ISRs.

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 Usage

```
OS_MEM       CommMem;
          CPU_INT32U  *CommMsg;


          void Task (void *p_arg)
          {
              OS_ERR err;


              (void)&p_arg;
              while (DEF_ON) {
                  OSMemPut(&CommMem,
                           (void *)CommMsg,
                           &err);
                  /* Check “err” */
                  :
                  :
              }
          }
```

Listing - OSMemPut() example usage

API - Task Management

- OSSchedRoundRobinCfg()
- OSSchedRoundRobinYield()
- OSTaskChangePrio()
- OSTaskCreate()
- OSTaskCreateHook()
- OSTaskDel()
- OSTaskDelHook()
- OSTaskRegGet()
- OSTaskRegGetID()
- OSTaskRegSet()
- OSTaskResume()
- OSTaskReturnHook()
- OSTaskStkChk()
- OSTaskStkInit()
- OSTaskSuspend()
- OSTaskSwHook()
- OSTaskTimeQuantaSet()

OSSchedRoundRobinCfg

Description

`OSSchedRoundRobinCfg()` is used to enable or disable round-robin scheduling.

Files

os.h/os_core.c

Prototype

```
void  OSSchedRoundRobinCfg (CPU_BOOLEAN   en,
                            OS_TICK       dflt_time_quanta,
                            OS_ERR       *p_err)
```

Arguments

`en`

when set to `DEF_ENABLED` enables round-robin scheduling, and when set to `DEF_DISABLED` disables it.

`dflt_time_quanta`

is the default time quanta given to a task. This value is used when a task is created and you specify a value of 0 for the time quanta. In other words, if the user did not specify a non-zero for the task's time quanta, this is the value that will be used. If passing 0 for this argument, µC/OS-III will assume a time quanta of 1/10 the tick rate. For example, if the tick rate is 1000 Hz and 0 is passed for `dflt_time_quanta` then, µC/OS-III will set the time quanta to 10 milliseconds.

`p_err`

is a pointer to a variable that is used to hold an error code:

`OS_ERR_NONE`

If the call is successful.

Returned Value

None

Required Configuration

`OS_CFG_SCHED_ROUND_ROBIN_EN` must be enabled in `os_cfg.h`. Refer to µC-OS-III Configuration Manual.

Callers

Application.

Notes/Warnings

None

Example Usage

```
void main (void)
          {
              OS_ERR  err;


              :
              OSInit(&err);                 /* Initialize µC/OS-III             */
              :
              :
              OSSchedRoundRobinCfg(DEF_ENABLED,
                                   10,
                                   &err);
              /* Check “err” */
              :
              :
              OSStart(&err);                /* Start Multitasking              */
          }
```

OSSchedRoundRobinYield

Description

`OSSchedRoundRobinYield()` is used to voluntarily give up a task's time slot, assuming that there are other tasks running at the same priority.

Files

os.h/os_core.c

Prototype

```
void  OSSchedRoundRobinYield (OS_ERR  *p_err)
```

Arguments

`p_err`

is a pointer to a variable used to hold an error code:

`OS_ERR_NONE`

If the call was successful.

`OS_ERR_ROUND_ROBIN_1`

If there is only one task at the current priority level that is ready-to-run.

`OS_ERR_ROUND_ROBIN_DISABLED`

If round-robin scheduling has not been enabled. See `OSSchedRoundRobinCfg()` to enable or disable.

`OS_ERR_SCHED_LOCKED`

If the scheduler is locked and µC/OS-III cannot switch tasks.

`OS_ERR_YIELD_ISR`

If `OS_CFG_CALLED_FROM_ISR_CHK_EN` set to `DEF_ENABLED` in `os_cfg.h`: if calling this function from an ISR.

Returned Value

None

Required Configuration

`OS_CFG_SCHED_ROUND_ROBIN_EN` must be enabled in `os_cfg.h`. Refer to µC-OS-III Configuration Manual.

Callers

Application.

Notes/Warnings

None

Example Usage

```
void Task (void *p_arg)
          {
              OS_ERR  err;


              (void)&p_arg;
              while (DEF_ON) {
                  :
                  :
                  OSSchedRoundRobinYield(&err); /* Give up the CPU to the next task at same
priority */
                  /* Check “err” */
                  :
                  :
              }
          }
```

OSTaskChangePrio

Description

When creating a task (see `OSTaskCreate()`), you specify the priority of the task being created. In most cases, it is not necessary to change the priority of the task at run time. However, it is sometimes useful to do so, and `OSTaskChangePrio()` allows this to take place.

If the task is ready-to-run, `OSTaskChangePrio()` simply changes the position of the task in µC/OS-III's ready list. If the task is waiting on an event, `OSTaskChangePrio()` will change the position of the task in the pend list of the corresponding object, so that the pend list remains sorted by priority.

Because µC/OS-III supports multiple tasks at the same priority, there are no restrictions on the priority that a task can have, except that task priority zero (0) is reserved by µC/OS-III, and priority `OS_PRIO_MAX-1` is used by the idle task.

Note that a task priority cannot be changed from an ISR.

Files

os.h/os_task.c

Prototype

```
void  OSTaskChangePrio (OS_TCB   *p_tcb,
                        OS_PRIO   prio_new,
                        OS_ERR   *p_err)
```

Arguments

`p_tcb`

is a pointer to the `OS_TCB` of the task for which the priority is being changed. If you pass a `NULL` pointer, the priority of the current task is changed.

`prio_new`

is the new task's priority. This value must never be set to `OS_CFG_PRIO_MAX-1`, or higher and you must not use priority 0 since they are reserved for µC/OS-III.

`p_err`

is a pointer to a variable that will receive an error code:

`OS_ERR_NONE`

If the task's priority is changed.

`OS_ERR_OS_NOT_RUNNING`

If `OS_CFG_INVALID_OS_CALLS_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if µC/OS-III is not running yet.

`OS_ERR_PRIO_INVALID`

If the priority of the task specified is invalid. By specifying a priority greater than or equal to `OS_PRIO_MAX-1`, or 0 or the same priority in use by another kernel task.

`OS_ERR_STATE_INVALID`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if the task is in an invalid state.

`OS_ERR_TASK_CHANGE_PRIO_ISR`

If `OS_CFG_CALLED_FROM_ISR_CHK_EN` set to `DEF_ENABLED` in `os_cfg.h`: if attempting to change the task's priority from an ISR.

Returned Value

None

Required Configuration

`OS_CFG_TASK_CHANGE_PRIO_EN` must be enabled in `os_cfg.h`. Refer to µC-OS-III Configuration Manual.

Callers

Application.

Notes/Warnings

1. The new priority must be available.

Example Usage

```
OS_TCB  MyTaskTCB;


          void TaskX (void *p_arg)
          {
              OS_ERR  err;


              while (DEF_ON) {
                  :
                  :
                  OSTaskChangePrio(&MyTaskTCB,     /* Change the priority of “MyTask” to 10 */
                                    10,
                                   &err);
                  /* Check “err” */
                  :
                  :
              }
          }
```

Listing - OSTaskChangePrio() example usage

OSTaskCreate

Description

Tasks must be created in order for µC/OS-III to recognize them as tasks. You create a task by calling `OSTaskCreate()` and by providing arguments specifying to µC/OS-III how the task will be managed. Tasks are always created in the ready-to-run state.

Tasks can be created either prior to the start of multitasking (i.e., before calling `OSStart()`), or by a running task. A task cannot be created by an ISR. A task must either be written as an infinite loop, or delete itself once completed. If the task code returns by mistake, µC/OS-III will terminate the task by calling `OSTaskDel((OS_TCB *)0, &err))`. At Micrium, we like the "`while (DEF_ON)`" to implement infinite loops because, by convention, we use a `while` loop when we don't know how many iterations a loop will do. This is the case of an infinite loop. We prefer to use `for` loops when we know how many iterations a loop will do.

Files

os.h/os_task.c

Prototype

```
void  OSTaskCreate (OS_TCB        *p_tcb,
                    CPU_CHAR      *p_name,
                    OS_TASK_PTR    p_task,
                    void          *p_arg,
                    OS_PRIO        prio,
                    CPU_STK       *p_stk_base,
                    CPU_STK_SIZE   stk_limit,
                    CPU_STK_SIZE   stk_size,
                    OS_MSG_QTY     q_size,
                    OS_TICK        time_quanta,
                    void          *p_ext,
                    OS_OPT         opt,
                    OS_ERR        *p_err)
```

Task as an infinite loop:

```
void MyTask (void *p_arg)
          {
              /* Local variables                                             */


              /* Do something with 'p_arg'                                   */
              /* Task initialization                                         */
              while (DEF_ON) {      /* Task body, as an infinite loop.           */

              }
          }
```

Listing - Task as an infinite loop

Run to completion task:

```
void MyTask (void *p_arg)
          {
              OS_ERR  err;
              /* Local variables                                             */


              /* Do something with 'p_arg'                                   */
              /* Task initialization                                         */
              /* Task body (do some work)                                    */
              OSTaskDel((OS_TCB *)0, &err);
              /* Check ’err” ... your code should never end up here!             */
          }
```

Listing - Run to completion task

Arguments

`p_tcb`

is a pointer to the task’s `OS_TCB` to use. It is assumed that storage for the TCB of the task will be allocated by the user code. You can declare a “global” variable as follows, and pass a pointer to this variable to `OSTaskCreate()`:

```
OS_TCB  MyTaskTCB;
```

`p_name`

is a pointer to an ASCII string (NUL terminated) to assign a name to the task. The name can be displayed by debuggers or by µC/Probe.

`p_task`

is a pointer to the task (i.e., the name of the function that defines the task).

`p_arg`

is a pointer to an optional data area which is used to pass parameters to the task when it is created. When µC/OS-III runs the task for the first time, the task will think that it was invoked, and passed the argument `p_arg`. For example, you could create a generic task that handles an asynchronous serial port. `p_arg` can be used to pass task information about the serial port it will manage: the port address, baud rate, number of bits, parity, and more. `p_arg` is the argument received by the task shown below.

```
void MyTask (void *p_arg)
          {
              while (DEF_ON) {
                  Task code;
              }
          }
```

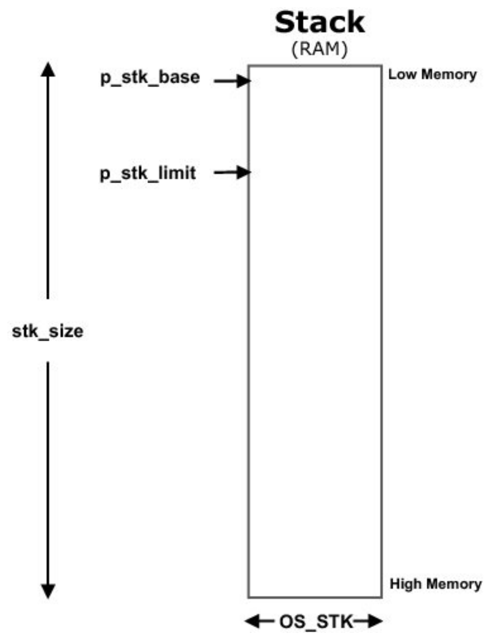
`prio`

is the task priority. The lower the number, the higher the priority (i.e., the importance) of the task. If `OS_CFG_ISR_POST_DEFERRED_EN` is set to `DEF_ENABLED` in `os_cfg.h`, the user cannot use priority `0`.

Task priority must also have a lower number than `OS_CFG_PRIO_MAX-1`. Priorities `0`, `1`, `OS_CFG_PRIO_MAX-2` and `OS_CFG_PRIO_MAX-1` are reserved. In other words, a task should have a priority between `2` and `OS_CFG_PRIO_MAX-3`, inclusively.

`p_stk_base`

is a pointer to the task's stack base address. The task's stack is used to store local variables, function parameters, return addresses, and possibly CPU registers during an interrupt.



The task stack must be declared as follows:

```
CPU_STK MyTaskStk[???];
```

The user would then pass `p_stk_base` the address of the first element of this array or, `&MyTaskStk[0]`. “???” represents the size of the stack.

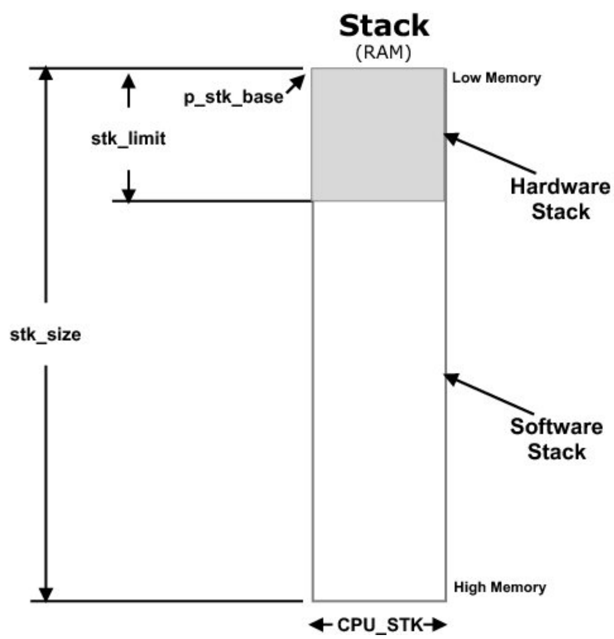
The size of this stack is determined by the task’s requirements and the anticipated interrupt nesting (unless the processor has a separate stack just for interrupts). Determining the size of the stack involves knowing how many bytes are required for storage of local variables for the task itself, all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

Note that you can allocate stack space for a task from the heap but, in this case, we don’t recommend to ever delete the task and free the stack space as this can cause the heap to fragment, which is not desirable in embedded systems.

`stk_limit`

is used to locate, within the task’s stack, a watermark limit that can be used to monitor and ensure that the stack does not overflow.

If the processor does not have hardware stack overflow detection, or this feature is not implemented in software by the port developer, this value may be used for other purposes. For example, some processors have two stacks, a hardware and a software stack. The hardware stack typically keeps track of function call nesting and the software stack is used to pass function arguments. `stk_limit` may be used to set the size of the hardware stack as shown below.



`stk_size`

specifies the size of the task's stack in number of elements. If `CPU_STK` is set to `CPU_INT08U` (see `os_type.h`), `stk_size` corresponds to the number of bytes available on the stack. If `CPU_STK` is set to `CPU_INT16U`, then `stk_size` contains the number of 16-bit entries available on the stack. Finally, if `CPU_STK` is set to `CPU_INT32U`, `stk_size` contains the number of 32-bit entries available on the stack.

`q_size`

A µC/OS-III task contains an optional internal message queue (if `OS_CFG_TASK_Q_EN` is set to `DEF_ENABLED` in `os_cfg.h`). This argument specifies the maximum number of messages that the task can receive through this message queue. The user may specify that the task is unable to receive messages by setting this argument to `0`.

`time_quanta`

the amount of time (in clock ticks) for the time quanta when round robin is enabled. If you specify `0`, then the default time quanta will be used which is the tick rate divided by `10`.

`p_ext`

is a pointer to a user-supplied memory location (typically a data structure) used as a TCB extension. For example, the user memory can hold the contents of floating-point registers during a context switch.

`opt`

contains task-specific options. Each option consists of one bit. The option is selected when the bit is set. The current version of µC/OS-III supports the following options:

`OS_OPT_TASK_NONE`

specifies that there are no options.

`OS_OPT_TASK_STK_CHK`

specifies whether stack checking is allowed for the task.

`OS_OPT_TASK_STK_CLR`

specifies whether the stack needs to be cleared.

`OS_OPT_TASK_SAVE_FP`

specifies whether floating-point registers are saved. This option is only valid if the processor has floating-point hardware and the processor-specific code saves the floating-point registers.

`OS_OPT_TASK_NO_TLS`

If the caller doesn't want or need TLS (Thread Local Storage) support for the task being created. If you do not include this option, TLS will be supported by default, assuming Micriµm supports TLS for the toolchain you are using. TLS support was added in V3.03.00.

`p_err`

is a pointer to a variable that will receive an error code:

`OS_ERR_NONE`

If the function is successful.

`OS_ERR_ILLEGAL_CREATE_RUN_TIME`

If `OS_SAFETY_CRITICAL_IEC61508` is defined: you called this after calling `OSStart()` and thus you are no longer allowed to create additional kernel objects.

`OS_ERR_PRIO_INVALID`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if `prio` is higher than the maximum value allowed (i.e., > `OS_PRIO_MAX-1`). Or, if your tried to use any in-use priority by the kernel, see the `prio` argument.

`OS_ERR_STAT_STK_SIZE_INVALID`

If the task's stack overflowed during initialization.

`OS_ERR_STK_INVALID`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if specifying a `NULL` pointer for `p_stk_base`.

`OS_ERR_STK_SIZE_INVALID`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if specifying a stack size smaller than what is currently specified by `OS_CFG_STK_SIZE_MIN` (see the `os_cfg.h`).

`OS_ERR_STK_LIMIT_INVALID`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if specifying a stack limit greater than or equal to the stack size.

`OS_ERR_TASK_CREATE_ISR`

If `OS_CFG_CALLED_FROM_ISR_CHK_EN` set to `DEF_ENABLED` in `os_cfg.h`: if attempting to create the task from an ISR.

`OS_ERR_TASK_INVALID`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if specifying a `NULL` pointer for `p_task`.

`OS_ERR_TCB_INVALID`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if specifying a `NULL` pointer for `p_tcb`.

Returned Value

None

Required Configuration

None

Callers

Application.

Notes/Warnings

1. The stack must be declared with the `CPU_STK` type.

2. A task must always invoke one of the services provided by µC/OS-III to wait for time to expire, suspend the task, or wait on an object (wait on a message queue, event flag, mutex, semaphore, a signal or a message to be sent directly to the task). This allows other tasks to gain control of the CPU.

3. You should not use task priorities `0`, `1`, `OS_CFG_PRIO_MAX-2` and `OS_CFG_PRIO_MAX-1` because they are reserved for use by µC/OS-III.

Example Usage

`OSTaskCreate()` can be called from `main()` (in C), or a previously created task.


```
OS_TCB  MyTaskTCB;                        /*  (1) Storage for task’s TCB                  */
          CPU_STK MyTaskStk[200];


          void  MyTask (void *p_arg)                /*  (3) The address of the task is its name 
*/
          {
              while (DEF_ON) {
                  /* Wait for an event */
                  /* My task body      */
              }
          }


          void SomeCode (void)
          {
              OS_ERR  err;
              :
              :
              OSTaskCreate (&MyTaskTCB,             /*  (1) Address of TCB assigned to the task 
*/
                            “My Task”,              /*  (2) Name you want to give the task        
*/
                             MyTask,                /*  (3) Address of the task itself            
*/
                            (void *)0,              /*  (4) “p_arg” is not used                   
*/
                             12,                    /*  (5) Priority you want to assign to the 
task  */
                            &MyTaskStk[0],          /*  (6) Base address of task’s stack          
*/
                             10,                    /*  (7) Watermark limit for stack growth      
*/
                            200,                    /*  (8) Stack size in number of CPU_STK 
elements */
                              5,                    /*  (9) Size of task message queue            
*/
                             10,                    /* (10) Time quanta (in number of ticks)      
*/
                            (void *)0,              /* (11) Extension pointer is not used         
*/
                            OS_OPT_TASK_STK_CHK + OS_OPT_TASK_STK_CLR, /* (12) Options            
*/
                            &err);                                     /* (13) Error code         
*/
              /* Check “err”                                              (14)                    
*/
              :
              :
          }
```

Listing - OSTaskCreate() example usage

(1) In order to create a task, you need to allocate storage for a TCB and pass a pointer to this TCB to `OSTaskCreate()`.

(2) You can assign an ASCII name to the task by passing a pointer to an ASCII string. The ASCII string may be allocated in code space (i.e., ROM), or data space (i.e., RAM). In

either case, it is assumed that the code can access that memory. The ASCII string must be `NUL` terminated.

(3) You pass the address of the task to `OSTaskCreate()`. In C, the address of a function is simply the name of that function.

(4) To provide additional data to `MyTask()`, you can pass a pointer to such data. In this case, `MyTask()` did not need such data and therefore, a `NULL` pointer is passed.

(5) The user must assign a priority to the task. The priority specifies the importance of this task with respect to other tasks. A low-priority value indicates a high priority. Priority `0` is the highest priority (reserved for an internal task) and a priority up to `OS_CFG_PRIO_MAX-3` can be specified (see `os_cfg.h`). Note that `OS_CFG_PRIO_MAX-1` is also reserved for an internal task, the idle task.

(6) The next argument specifies the “base address” of the task’s stack. In this case, it is simply the base address of the array `MyTaskStk[]`. Note that it is possible to simply specify the name of the array. We prefer to make it clear by writing `&MyTaskStk[0]`.

(7) This argument sets the watermark limit for stack growth. If the processor port does not use this field then you can set this value to `0`.

(8) µC/OS-III also needs to know the size of the stack for the task. This allows µC/OS-III to perform stack checking at run time. This argument represents the number of `CPU_STK` elements, not the number of bytes.

(9) µC/OS-III allows tasks or ISRs to send messages directly to a task. This argument specifies how many such messages can be received by this task.

(10) This argument specifies how much time (in number of ticks) this task will run on the CPU before µC/OS-III will force the CPU away from this task and run the next task at the same priority (if there are more than one task at the same priority that is ready-to-run).

(11) µC/OS-III allows the user to “extend” the capabilities of the TCB by allowing passing a pointer to some memory location that could contain additional information about the

task. For example, there may be a CPU that supports floating-point math and the user would likely need to save the floating-point registers during a context switch. This pointer could point to the storage area for these registers.

(12) When creating a task, options must be specified. Specifically, such options as, whether the stack of the task will be cleared (i.e., filled with 0x00) when the task is created (`OS_OPT_TASK_STK_CLR`), whether µC/OS-III will be allowed to check for stack usage (`OS_OPT_TASK_STK_CHK`), whether the CPU supports floating-point math, and whether the task will make use of the floating-point registers and therefore need to save and restore them during a context switch (`OS_OPT_TASK_SAVE_FP`). The options are additive.

(13) Most of µC/OS-III's services return an error code indicating the outcome of the call. The error code is always returned as a pointer to a variable of type `OS_ERR`. The user must allocate storage for this variable prior to calling `OSTaskCreate()`.

(14) It is highly recommended that the user examine the error code whenever calling a µC/OS-III function. If the call is successful, the error code will always be `OS_ERR_NONE`. If the call is not successful, the returned code will indicate the reason for the failure (see `p_err` and `OS_ERR_???` in `os.h`).

OSTaskCreateHook

Description

This function is called by `OSTaskCreate()` after initializing the `OS_TCB` fields and setting up the stack frame for the task, just before adding the task to the ready list. When `OSTaskCreateHook()` is called, all of the `OS_TCB` fields are assumed to be initialized.

`OSTaskCreateHook()` is part of the CPU port code and this function *must not* be called by the application code. `OSTaskCreateHook()` is actually used by the µC/OS-III port developer.

You can use this hook to initialize and store the contents of floating-point registers, MMU registers, or anything else that can be associated with a task. Typically, you would store this additional information in memory allocated by the application.

Files

os.h/os_cpu_c.c and os_app_hooks.c

Prototype

```
void  OSTaskCreateHook (OS_TCB  *p_tcb)
```

Arguments

`p_tcb`

is a pointer to the TCB of the task being created. Note that the `OS_TCB` has been validated by `OSTaskCreate()` and is guaranteed to not be a `NULL` pointer when `OSTaskCreateHook()` is called.

Returned Value

None

Required Configuration

OS_CFG_APP_HOOKS_EN must be enabled in os_cfg.h. Refer to µC-OS-III Configuration Manual.

Callers

OSTaskCreate().

Notes/Warnings

1. *Do not* call this function from the application.

Example Usage

The code below calls an application-specific hook that the application programmer can define. The user can simply set the value of OS_AppTaskCreateHookPtr to point to the desired hook function as shown in the example. OSTaskCreate() calls OSTaskCreateHook() which in turns calls App_OS_TaskCreateHook() through OS_AppTaskCreateHookPtr. As can be seen, when called, the application hook is passed the address of the OS_TCB of the newly created task.

```
void  App_OS_TaskCreateHook (OS_TCB *p_tcb)                 /* os_app_hooks.c         */
          {
              /* Your code goes here! */
          }


          void App_OS_SetAllHooks (void)                             /* os_app_hooks.c 
*/
          {
              CPU_SR_ALLOC();

              CPU_CRITICAL_ENTER();
              :
              OS_AppTaskCreateHookPtr = App_OS_TaskCreateHook;
              :
              CPU_CRITICAL_EXIT();
          }


          void  OSTaskCreateHook (OS_TCB *p_tcb)                     /* os_cpu_c.c 
*/
          {
          #if OS_CFG_APP_HOOKS_EN > 0u
              if (OS_AppTaskCreateHookPtr != (OS_APP_HOOK_TCB)0) {    /* Call application hook 
*/
                  (*OS_AppTaskCreateHookPtr)(p_tcb);
              }
          #endif
```

```
        }
```

OSTaskDel

Description

When a task is no longer needed, it can be deleted. Deleting a task does not mean that the code is removed, but that the task code is no longer managed by µC/OS-III. `OSTaskDel()` can be used when creating a task that will only run once. In this case, the task must not return but instead call `OSTaskDel((OS_TCB *)0, &err) which` specifies to µC/OS-III to delete the currently running task.

A task may also delete another task by specifying to `OSTaskDel()` the address of the `OS_TCB` of the task to delete.

Once a task is deleted, its `OS_TCB` and stack may be reused to create another task. This assumes that the task's stack requirement of the new task is satisfied by the stack size of the deleted task.

Even though µC/OS-III allows the user to delete tasks at run time, it is recommend that such actions be avoided. Why? Because a task can "own" resources that are shared with other tasks. Deleting the task that owns resource(s) without first relinquishing the resources could lead to strange behaviors and possible deadlocks.

Files

os.h/os_task.c

Prototype

```
void  OSTaskDel (OS_TCB  *p_tcb,
                 OS_ERR  *p_err)
```

Arguments

`p_tcb`

is a pointer to the TCB of the task to delete or, you can pass a `NULL` pointer to specify that the calling task delete itself. If deleting the calling task, the scheduler will be invoked so that the next highest-priority task is executed.

p_err

is a pointer to a variable that will receive an error code:

OS_ERR_NONE

'p_err' gets set to OS_ERR_NONE before OSSched() to allow the returned error code to be monitored (by another task) even for a task that is deleting itself. In this case, p_err *must* point to a global variable that can be accessed by that other task and, you should initialize that variable to OS_ERR_TASK_RUNNING prior to deleting the task.

OS_ERR_ILLEGAL_DEL_RUN_TIME

If OS_SAFETY_CRITICAL_IEC61508 is defined: you called this after calling OSStart() and thus you are no longer allowed to delete kernel objects.

OS_ERR_OS_NOT_RUNNING

If OS_CFG_INVALID_OS_CALLS_CHK_EN is set to DEF_ENABLED in os_cfg.h: if µ C/OS-III is not running yet.

OS_ERR_STATE_INVALID

If the task is in an invalid state.

OS_ERR_TASK_DEL_IDLE

If attempting to delete the idle task.

OS_ERR_TASK_DEL_INVALID

If attempting to delete the ISR Handler task while OS_CFG_ISR_POST_DEFERRED_EN is set to DEF_ENABLED.

OS_ERR_TASK_DEL_ISR

If `OS_CFG_CALLED_FROM_ISR_CHK_EN` set to `DEF_ENABLED` in `os_cfg.h`: if you called `OSTaskDel()` from an ISR.

Returned Value

None

Required Configuration

`OS_CFG_TASK_DEL_EN` must be enabled in `os_cfg.h`. Refer to µC-OS-III Configuration Manual
.

Callers

Application.

Notes/Warnings

1. `OSTaskDel()` verifies that the user is not attempting to delete the µC/OS-III idle task and the ISR handler task.

2. Be careful when deleting a task that owns resources.

Example Usage

```
OS_TCB  MyTaskTCB;


          void TaskX (void *p_arg)
          {
              OS_ERR  err;


              while (DEF_ON) {
                  :
                  :
                  OSTaskDel(&MyTaskTCB,
                            &err);
                  /* Check “err” */
                  :
                  :
              }
          }
```

Listing - OSTaskDel() example usage

OSTaskDelHook

Description

This function is called by `OSTaskDel()` after the task is removed from the ready list or any pend list.

You can use this hook to deallocate storage assigned to the task.

`OSTaskDelHook()` is part of the CPU port code and this function *must not* be called by the application code. `OSTaskDelHook()` is actually used by the µC/OS-III port developer.

Files

os.h/os_cpu_c.c and os_app_hooks.c

Prototype

```
void  OSTaskDelHook (OS_TCB  *p_tcb)
```

Arguments

`p_tcb`

> is a pointer to the TCB of the task being created. Note that the `OS_TCB` has been validated by `OSTaskDel()` and is guaranteed to not be a `NULL` pointer when `OSTaskDelHook()` is called.

Returned Value

None

Required Configuration

`OS_CFG_APP_HOOKS_EN` must be enabled in `os_cfg.h`. Refer to µC-OS-III Configuration Manual.

Callers

`OSTaskDel()`.

Notes/Warnings

1. *Do not* call this function from the application.

Example Usage

The code below calls an application-specific hook that the application programmer can define. The user can simply set the value of `OS_AppTaskDelHookPtr` to point to the desired hook function. `OSTaskDel()` calls `OSTaskDelHook()` which in turns calls `App_OS_TaskDelHook()` through `OS_AppTaskDelHookPtr`. As can be seen, when called, the application hook is passed the address of the `OS_TCB` of the task being deleted.

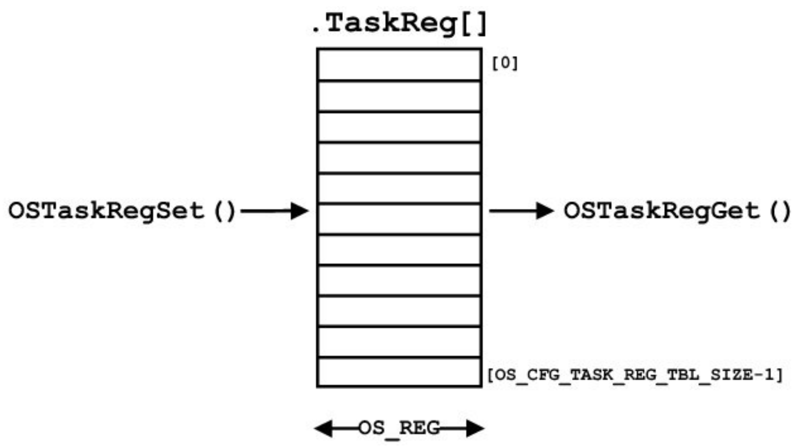
```
void  App_OS_TaskDelHook (OS_TCB *p_tcb)                  /* os_app_hooks.c       */
          {
              /* Your code goes here! */
          }


          void App_OS_SetAllHooks (void)                            /* os_app_hooks.c
*/
          {
              CPU_SR_ALLOC();

              CPU_CRITICAL_ENTER();
              :
              OS_AppTaskDelHookPtr = App_OS_TaskDelHook;
              :
              CPU_CRITICAL_EXIT();
          }



          void  OSTaskDelHook (OS_TCB *p_tcb)                        /* os_cpu_c.c
*/
          {
          #if OS_CFG_APP_HOOKS_EN > 0u
              if (OS_AppTaskDelHookPtr != (OS_APP_HOOK_TCB)0) {      /* Call application hook
*/
                  (*OS_AppTaskDelHookPtr)(p_tcb);
              }
          #endif
          }
```

OSTaskRegGet

Description

µC/OS-III 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_CFG_TASK_REG_TBL_SIZE` registers, and all registers have a data type of `OS_REG`. However, `OS_REG` can be declared at compile time (see `os_type.h`) to be nearly anything (8-, 16-, 32-, 64-bit signed or unsigned integer, or floating-point).

As shown below, 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_CFG_TASK_REG_TBL_SIZE-1`.



Files

os.h/os_task.c

Prototype

```
OS_REG  OSTaskRegGet (OS_TCB     *p_tcb,
                      OS_REG_ID   id,
                      OS_ERR     *p_err)
```

Arguments

`p_tcb`

is a pointer to the TCB of the task the user is receiving a task-register value from. A `NULL` pointer indicates that the user wants the value of a task register of the calling task.

`id`

is the identifier of the task register and valid values are from `0` to `OS_CFG_TASK_REG_TBL_SIZE-1`.

`p_err`

is a pointer to a variable that will contain an error code returned by this function.

`OS_ERR_NONE`

If the call was successful and the function returned the value of the desired task register.

`OS_ERR_REG_ID_INVALID`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if a valid task register identifier is not specified.

Returned Value

The current value of the task register.

Required Configuration

`OS_CFG_TASK_REG_TBL_SIZE` must be greater than 0 in `os_cfg.h`. Refer to µC-OS-III Configuration Manual.

Callers

Application.

Notes/Warnings

None

Example Usage

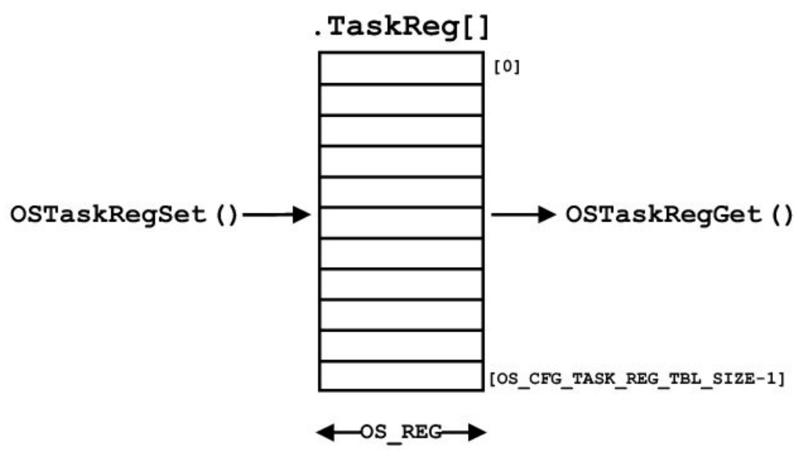
```
OS_TCB  MyTaskTCB;


          void TaskX (void *p_arg)
          {
              OS_ERR  err;
              OS_REG  reg;


              while (DEF_ON) {
                  :
                  reg = OSTaskRegGet(&MyTaskTCB,
                                      5,
                                     &err);
                  /* Check “err” */
                  :
                  :
              }
          }
```

OSTaskRegGetID

Description

`OSTaskRegGetID()` allows your application to assign task register IDs dynamically. In other words, instead of using `#define` constants to establish a task register number (or index) into the `.TaskReg[]` shown below, you should always use `OSTaskRegGetID()`, assign the ID to a variable and use this ID when calling `OSTaskRegGet()` or `OSTaskRegSet()`.

If successful, `OSTaskRegGetID()` will return an ID between `0` and `OS_CFG_TASK_REG_TBL_SIZE-1`.



Files

os.h/os_task.c

Prototype

```
OS_REG_ID  OSTaskRegGetID (OS_ERR  *p_err)
```

Arguments

`p_err`

is a pointer to a variable that will contain an error code returned by this function.

`OS_ERR_NONE`

If the call was successful and the function returned the next available task register ID (or index).

`OS_ERR_NO_MORE_ID_AVAIL`

If you already called `OSTaskRegGetID()` `OS_CFG_TASK_REG_TBL_SIZE` (see `os_cfg.h`) times and thus there are no more IDs available to be assigned.

Returned Value

The next available task register ID or `OS_CFG_TASK_REG_TBL_SIZE` if all the IDs have already been assigned.

Required Configuration

`OS_CFG_TASK_REG_TBL_SIZE` must be greater than 0 in `os_cfg.h`. Refer to µC-OS-III Configuration Manual.

Callers

Application.

Notes/Warnings

None

Example Usage

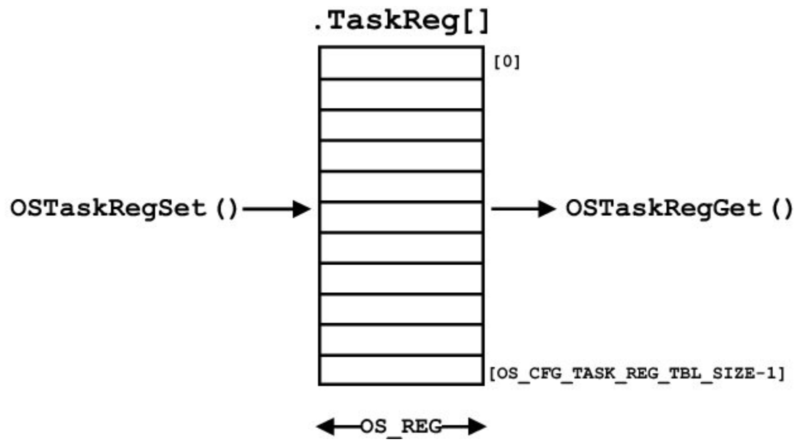
```
OS_REG_ID  MyTaskRegID;


          void main (void)
          {
              OS_ERR  err;

              :
              OSInit(&err);
              :
              MyTaskRegID = OSTaskRegGetID(&err);
              /* Check “err” */
              :
              :
              OSStart(&err);
          }
```

OSTaskRegSet

Description

µC/OS-III 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_CFG_TASK_REG_TBL_SIZE registers, and all registers have a data type of OS_REG. However, OS_REG can be declared at compile time to be nearly anything (8-, 16-, 32-, 64-bit signed or unsigned integer, or floating-point).

As shown below, 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_CFG_TASK_REG_TBL_SIZE-1.



Files

os.h/os_task.c

Prototype

```
void  OSTaskRegSet (OS_TCB     *p_tcb,
                    OS_REG_ID   id,
                    OS_REG      value,
                    OS_ERR     *p_err)
```

Arguments

`p_tcb`

is a pointer to the TCB of the task you are setting. A `NULL` pointer indicates that the user wants to set the value of a task register of the calling task.

`id`

is the identifier of the task register and valid values are from `0` to `OS_CFG_TASK_REG_TBL_SIZE-1`.

`value`

is the new value of the task register specified by `id`.

`p_err`

is a pointer to a variable that will contain an error code returned by this function.

`OS_ERR_NONE`

If the call was successful, and the function set the value of the desired task register.

`OS_ERR_REG_ID_INVALID`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if a valid task register identifier is not specified.

Returned Value

None

Required Configuration

`OS_CFG_TASK_REG_TBL_SIZE` must be greater than 0 in `os_cfg.h`. Refer to µC-OS-III Configuration Manual.

Callers

Application.

Notes/Warnings

None

Example Usage

```
OS_TCB  MyTaskTCB;


          void TaskX (void *p_arg)
          {
              OS_ERR  err;


              while (DEF_ON) {
                  :
                  OSTaskRegSet(&MyTaskTCB,
                                Â    5,
                                   23,
                                   &err);
                  /* Check “err” */
                  :
                  :
              }
          }
```

OSTaskResume

Description

Resumes a task suspended through the `OSTaskSuspend()` function. In fact, `OSTaskResume()` is the only function that can unsuspend a suspended task. Obviously, the suspended task can only be resumed by another task. If the suspended task is also waiting on another kernel object such as an event flag, semaphore, mutex, message queue etc., the suspension will simply be lifted (i.e., removed), but the task will continue waiting for the object.

The user can “nest” suspension of a task by calling `OSTaskSuspend()` and therefore must call `OSTaskResume()` an equivalent number of times to resume such a task. In other words, if suspending a task five times, it is necessary to unsuspend the same task five times to remove the suspension of the task.

Files

os.h/os_task.c

Prototype

```
void  OSTaskResume (OS_TCB  *p_tcb,
                    OS_ERR  *p_err)
```

Arguments

`p_tcb`

is a pointer to the TCB of the task that is resuming. A `NULL` pointer is not a valid value as one cannot resume the calling task because, by definition, the calling task is running and is not suspended.

`p_err`

is a pointer to a variable that will contain an error code returned by this function.

`OS_ERR_NONE`

If the call was successful and the desired task is resumed.

`OS_ERR_OS_NOT_RUNNING`

If `OS_CFG_INVALID_OS_CALLS_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if µC/OS-III is not running yet.

`OS_ERR_STATE_INVALID`

If the task is in an invalid state.

`OS_ERR_TASK_NOT_SUSPENDED`

If the task attempting to be resumed is not suspended.

`OS_ERR_TASK_RESUME_ISR`

If `OS_CFG_CALLED_FROM_ISR_CHK_EN` set to `DEF_ENABLED` in `os_cfg.h`: if calling this function from an ISR.

`OS_ERR_TASK_RESUME_SELF`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if passing a `NULL` pointer for `p_tcb` or, a pointer to the current TCB. It is not possible to resume the calling task since, if suspended, it cannot be executing.

Returned Value

None

Required Configuration

`OS_CFG_TASK_SUSPEND_EN` must be enabled in `os_cfg.h`. Refer to µC-OS-III Configuration Manual.

Callers

Application.

Notes/Warnings

None

Example Usage

```
OS_TCB   TaskY;


          void TaskX (void *p_arg)
          {
              OS_ERR err;


              while (DEF_ON) {
                  :
                  :
                  OSTaskResume(&TaskY,
                               &err);            /* Resume suspended task        */
                  /* Check “err” */
                  :
                  :
              }
          }
```

Listing - OSMutexCreate() example usage

OSTaskReturnHook

Description

This function is called by `OS_TaskReturn()`. `OS_TaskReturn()` is called if the user accidentally returns from the task code. In other words, the task should either be implemented as an infinite loop and never return, or the task must call `OSTaskDel((OS_TCB *)0, &err)` to delete itself to prevent it from exiting.

`OSTaskReturnHook()` is part of the CPU port code and this function *must not* be called by the application code. `OSTaskReturnHook()` is actually used by the µC/OS-III port developer.

Note that after calling `OSTaskReturnHook()`, `OS_TaskReturn()` will actually delete the task by calling:

```
OSTaskDel((OS_TCB *)0,
&err)
```

Files

os.h/os_cpu_c.c and os_app_hooks.c

Prototype

```
void  OSTaskReturnHook (OS_TCB  *p_tcb)
```

Arguments

`p_tcb`

is a pointer to the TCB of the task that is not behaving as expected. Note that the `OS_TCB` is validated by `OS_TaskReturn()`, and is guaranteed to not be a `NULL` pointer when `OSTaskReturnHook()` is called.

Returned Value

None

Required Configuration

OS_CFG_APP_HOOKS_EN must be enabled in os_cfg.h. Refer to µC-OS-III Configuration Manual.

Callers

OS_TaskReturn().

Notes/Warnings

1. *Do not* call this function from the application.

Example Usage

The code below calls an application-specific hook that the application programmer can define. For this, the user can simply set the value of OS_AppTaskReturnHookPtr to point to the desired hook function as shown in the example. If a task returns and forgets to call OSTaskDel((OS_TCB *)0, &err) then µC/OS-III will call OSTaskReturnHook() which in turns calls App_OS_TaskReturnHook() through OS_AppTaskReturnHookPtr. When called, the application hook is passed the address of the OS_TCB of the task returning.

```
void  App_OS_TaskReturnHook (OS_TCB  *p_tcb)                /* os_app_hooks.c         */
          {
              /* Your code goes here! */
          }


          void App_OS_SetAllHooks (void)                             /* os_app_hooks.c
*/
          {
              CPU_SR_ALLOC();

              CPU_CRITICAL_ENTER();
              :
              OS_AppTaskReturnHookPtr = App_OS_TaskReturnHook;
              :
              CPU_CRITICAL_EXIT();
          }


          void  OSTaskReturnHook (OS_TCB *p_tcb)                     /* os_cpu_c.c
*/
          {
          #if OS_CFG_APP_HOOKS_EN > 0u
              if (OS_AppTaskReturnHookPtr != (OS_APP_HOOK_TCB)0) {    /* Call application hook
*/
                  (*OS_AppTaskReturnHookPtr)(p_tcb);
```

```
    }
#endif
}
```

OSTaskStkChk

Description

`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 the `OS_TASK_OPT_STK_CHK` and `OS_TASK_OPT_STK_CLR` options.

Stack sizing is accomplished by walking from the bottom of the stack and counting the number of `0` entries on the stack until a non-zero value is found.

It is possible to not set the `OS_TASK_OPT_STK_CLR` when creating the task if the startup code clears all RAM, and tasks are not deleted (this reduces the execution time of `OSTaskCreate()`).

µC/OS-III's statistic task calls `OSTaskStkChk()` for each task created and stores the results in each task's `OS_TCB` so your application doesn't need to call this function if the statistic task is enabled.

Files

os.h/os_task.c

Prototype

```
void  OSTaskStkChk (OS_TCB        *p_tcb,
                    CPU_STK_SIZE  *p_free,
                    CPU_STK_SIZE  *p_used,
                    OS_ERR        *p_err)
```

Arguments

`p_tcb`

is a pointer to the TCB of the task where the stack is being checked. A `NULL` pointer indicates that the user is checking the calling task's stack.

`p_free`

is a pointer to a variable of type `CPU_STK_SIZE` and will contain the number of free `CPU_STK` elements on the stack of the task being inquired about.

`p_used`

is a pointer to a variable of type `CPU_STK_SIZE` and will contain the number of used `CPU_STK` elements on the stack of the task being inquired about.

`p_err`

is a pointer to a variable that will contain an error code returned by this function.

`OS_ERR_NONE`

If the call was successful.

`OS_ERR_PTR_INVALID`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if either `p_free` or `p_used` are `NULL` pointers.

`OS_ERR_TASK_NOT_EXIST`

If the stack pointer of the task is a `NULL` pointer.

`OS_ERR_TASK_OPT`

If `OS_OPT_TASK_STK_CHK` was not specififed when creating the task being checked.

`OS_ERR_TASK_STK_CHK_ISR`

If `OS_CFG_CALLED_FROM_ISR_CHK_EN` set to `DEF_ENABLED` in `os_cfg.h`: if calling this function from an ISR.

Returned Value

None

Required Configuration

OS_CFG_TASK_STAT_CHK_EN must be enabled in os_cfg.h. Refer to µC-OS-III Configuration Manual.

Callers

Application.

Notes/Warnings

1. Execution time of this task depends on the size of the task's stack.

2. The application can determine the total task stack space (in number of CPU_STK elements) by adding the value of *p_free and *p_used. This number should add up to the task's stack size which is stored in the .StkSize field of the OS_TCB of the task.

3. The #define CPU_CFG_STK_GROWTH must be declared (typically from os_cpu.h). When this #define is set to CPU_STK_GROWTH_LO_TO_HI, the stack grows from low memory to high memory. When this #define is set to CPU_STK_GROWTH_HI_TO_LO, the stack grows from high memory to low memory.

Example Usage

```
OS_TCB  MyTaskTCB;


          void Task (void *p_arg)
          {
              OS_ERR        err;
              CPU_STK_SIZE  n_free;
              CPU_STK_SIZE  n_used;


              (void)&p_arg;
              while (DEF_ON) {
                  :
                  :
                  OSTaskStkChk(&MyTaskTCB,
                               &n_free,
                               &n_used,
                               &err);
                  /* Check “err” */
                  :
                  :
              }
          }
```


OSTaskStkInit

Description

This function is called by `OSTaskCreate()` to setup the stack frame of the task being created. Typically, the stack frame will look as if an interrupt just occurred, and all CPU registers were pushed onto the task's stack. The stacking order of CPU registers is very CPU specific.

`OSTaskStkInit()` is part of the CPU port code and this function *must not* be called by the application code. `OSTaskStkInit()` is actually defined by the µC/OS-III port developer.

Files

os.h/os_cpu_c.c

Prototype

```
CPU_STK  *OSTaskStkInit (OS_TASK_PTR    p_task,
                         void          *p_arg,
                         CPU_STK       *p_stk_base,
                         CPU_STK       *p_stk_limit,
                         CPU_STK_SIZE   stk_size,
                         OS_OPT         opt)
```

Arguments

`p_task`

is the address of the task being created (see `MyTask()` below). Tasks must be declared as follows:

```
void   MyTask (void  *p_arg)
          {
              /* Do something with “p_arg” (optional) */
              while (DEF_ON) {
                  /* Wait for an event to occur */
                  /* Do some work               */
              }
          }
```

Or,

```
void   MyTask (void  *p_arg)
          {
              OS_ERR  err;


              /* Do something with “p_arg” (optional) */
              /* Do some work                         */
              OSTaskDel((OS_TCB *)0,
                        &err);
          }
```

`p_arg`

is the argument that the task will receive when the task first start (see code above).

`p_stk_base`

is the base address of the task’s stack. This is typically the lowest address of the area of storage reserved for the task stack. In other words, if declaring the task’s stack as follows:

```
CPU_STK  MyTaskStk[100];
```

`OSTaskCreate()` would pass `&OSMyTaskStk[0]` to `p_stk_base`.

`p_stk_limit`

is the address of the task’s stack limit watermark. This pointer is computed by `OSTaskCreate()` prior to calling `OSTaskStkInit()`.

`stk_size`

is the size of the task’s stack in number of `CPU_STK` elements. In the example above, the stack size is 100.

`opt`

is the options passed to `OSTaskCreate()` for the task being created.

Returned Value

The new top of stack after the task's stack is initialized. `OSTaskStkInit()` will place values on the task's stack and will return the new pointer for the stack pointer for the task. The value returned is very processor specific. For some processors, the returned value will point to the last value placed on the stack while, with other processors, the returned value will point at the next free stack entry.

Required Configuration

None

Callers

`OSTaskCreate()`.

Notes/Warnings

1. *Do not* call this function from the application.

Example Usage

The pseudo code below shows the typical steps performed by this function. Consult an existing µC/OS-III port for examples. Here it is assumed that the stack grows from high memory to low memory.

```
CPU_STK  *OSTaskStkInit (OS_TASK_PTR    p_task,
                                   void          *p_arg,
                                   CPU_STK       *p_stk_base,
                                   CPU_STK       *p_stk_limit,
                                   CPU_STK_SIZE   stk_size,
                                   OS_OPT         opt)
          {
              CPU_STK  *p_stk;


              p_stk    = &p_stk_base[stk_size – 1u];                            (1)
              *p_stk-- = Initialize the stack as if an interrupt just occurred; (2)
              return (p_stk);                                                   (3)
          }
```

(1) `p_stk` is set to the top-of-stack. It is assumed that the stack grows from high memory

locations to lower ones. If the stack of the CPU grew from low memory locations to higher ones, the user would simply set `p_stk` to point at the base. However, this also means that it would be necessary to initialize the stack frame in the opposite direction.

(2) The CPU registers are stored onto the stack using the same stacking order as used when an interrupt service routine (ISR) saves the registers at the beginning of the ISR. The value of the register contents on the stack is typically not important. However, there are some values that are critical. Specifically, you need to place the address of the task in the proper location on the stack frame and it may be important to load the value of the CPU register and possibly pass the value of `p_arg` in one of the CPU registers. Finally, if the task is to return by mistake, it is a good idea to place the address of `OS_TaskReturn()` in the proper location on the stack frame. This ensures that a faulty returning task is intercepted by µC/OS-III.

(3) Finally, your code will need to return the value of the stack pointer at the new top-of-stack frame. Some processors point to the last stored location, while others point to the next empty location. You should consult the processor documentation so that the return value points at the proper location.

Below is an example showing which arguments `OSTaskCreate()` passes to `OSTaskStkInit()`.

```
CPU_STK  MyTaskStk[100];
          OS_TCB   MyTaskTCB;


          void  MyTask (void *p_arg)
          {
              /* Do something with “parg” (optional) */
          }


          void  main (void)
          {
              OS_ERR  err;
              :
              :
              OSInit(&err);
              /* Check “err” */
              :
              :
              OSTaskCreate ((OS_TCB        *)&MyTaskTCB,
                            (CPU_CHAR      *)”My Task”,
                            (OS_TASK_PTR    )MyTask,                                    /*
“p_task”      of OSTaskStkInit() */
                            (void          *)0,                                         /*
“p_arg”       of OSTaskStkInit() */
                            (OS_PRIO        )prio,
                            (CPU_STK       *)&MyTaskStk[0],                             /*
“p_stk_base”  of OSTaskStkInit() */
                            (CPU_STK_SIZE   )10,                                        /*
“p_stk_limit” of OSTaskStkInit() */
                            (CPU_STK_SIZE   )100,                                       /*
“stk_size”    of OSTaskStkInit() */
```

```
                            (OS_MSG_QTY      )0,
                            (OS_TICK         )0,
                            (void           *)0,
                            (OS_OPT          )(OS_OPT_TASK_STK_CLR + OS_OPT_TASK_STK_CHK),   /*
“opt”         of OSTaskStkInit() */
                            (OS_ERR         *)&err);
              /* Check “err” */
              :
              :
              OSStart(&err);
              /* Check “err” */
              :
              :
          }
```

OSTaskSuspend

Description

`OSTaskSuspend()` suspends (or blocks) execution of a task unconditionally. The calling task may be suspended by specifying a `NULL` pointer for `p_tcb`, or simply by passing the address of its `OS_TCB`. In this case, another task needs to resume the suspended task. If the current task is suspended, rescheduling occurs, and µC/OS-III 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.

The user can “nest” suspension of a task by calling `OSTaskSuspend()` and therefore it is important to call `OSTaskResume()` an equivalent number of times to resume the task. If suspending a task five times, it is necessary to unsuspend the same task five times to remove the suspension of the task.

Files

os.h/os_task.c

Prototype

```
void   OSTaskSuspend (OS_TCB  *p_tcb,
                      OS_ERR  *p_err)
```

Arguments

`p_tcb`

is a pointer to the TCB of the task the user is suspending. A `NULL` pointer indicates suspension of the calling task.

`p_err`

is a pointer to a variable that will contain an error code returned by this function.

`OS_ERR_NONE`

If the call was successful and the desired task was suspended.

`OS_ERR_OS_NOT_RUNNING`

If `OS_CFG_INVALID_OS_CALLS_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if µC/OS-III is not running yet.

`OS_ERR_SCHED_LOCKED`

If attempting to suspend a task while the scheduler is locked.

`OS_ERR_STATE_INVALID`

If attempting to suspend a task that is in an invalid state.

`OS_ERR_TASK_SUSPEND_CTR_OVF`

If the nesting counter overflowed.

`OS_ERR_TASK_SUSPEND_ISR`

If `OS_CFG_CALLED_FROM_ISR_CHK_EN` set to `DEF_ENABLED` in `os_cfg.h`: if the function is called from an ISR.

`OS_ERR_TASK_SUSPEND_IDLE`

If attempting to suspend the idle task. This is not allowed since the idle task must always exist.

`OS_ERR_TASK_SUSPEND_INT_HANDLER`

If `OS_CFG_ISR_POST_DEFERRED_EN` is set to `DEF_ENABLED` in `os_cfg.h`: If attempting to suspend the ISR handler task. This is not allowed since the ISR handler task is a µC/OS-III internal task.

Returned Value

None

Required Configuration

`OS_CFG_TASK_SUSPEND_EN` must be enabled in `os_cfg.h`. Refer to µC-OS-III Configuration Manual.

Callers

Application.

Notes/Warnings

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

Example Usage

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

              (void)&p_arg;
              while (DEF_ON) {
                  :
                  :
                  OSTaskSuspend((OS_TCB *)0,
                                &err);       /* Suspend current task                       */
                  /* Check “err” */
                  :
                  :
              }
          }
```

Listing - OSTaskSuspend() example usage

OSTaskSwHook

Description

`OSTaskSwHook()` is always called by either `OSCtxSw()` or `OSIntCtxSw()` (see `os_cpu_a.asm`), just after saving the CPU registers onto the task being switched out. This hook function allows the port developer to perform additional operations (if needed) when µC/OS-III performs a context switch.

Before calling `OSTaskSwHook()`, `OSTCBCurPtr` points at the `OS_TCB` of the task being switched out, and `OSTCBHighRdyPtr` points at the `OS_TCB` of the new task being switched in.

The code shown in the example below should be included in all implementations of `OSTaskSwHook()`, and is used for performance measurements. This code is written in C for portability.

Files

os.h/os_cpu_c.c and os_app_hooks.c

Prototype

```
void  OSTaskSwHook (void)
```

Arguments

None

Returned Values

None

Required Configuration

`OS_CFG_APP_HOOKS_EN` must be enabled in `os_cfg.h`. Refer to µC-OS-III Configuration Manual.

Callers

`OSCtxSw()` and `OSIntCtxSw()`.

Notes/Warnings

None

Example Usage

The code below calls an application specific hook that the application programmer can define. The user can simply set the value of `OS_AppTaskSwHookPtr` to point to the desired hook function. When µC/OS-III performs a context switch, it calls `OSTaskSwHook()` which in turn calls `App_OS_TaskSwHook()` through `OS_AppTaskSwHookPtr`.

```
void  App_OS_TaskSwHook (void)                              /* os_app_hooks.c         */
          {
              /* Your code goes here! */
          }


          void App_OS_SetAllHooks (void)                              /* os_app_hooks.c
*/
          {
              CPU_SR_ALLOC();

              CPU_CRITICAL_ENTER();
              :
              OS_AppTaskSwHookPtr = App_OS_TaskSwHook;
              :
              CPU_CRITICAL_EXIT();
          }

          void  OSTaskSwHook (void)                                   /* os_cpu_c.c
*/
          {
          #if OS_CFG_TASK_PROFILE_EN > 0u
              CPU_TS     ts;
          #endif
          #ifdef  CPU_CFG_TIME_MEAS_INT_DIS_EN
              CPU_TS     int_dis_time;
          #endif


          #if OS_CFG_APP_HOOKS_EN > 0u
              if (OS_AppTaskSwHookPtr != (OS_APP_HOOK_VOID)0) {
                  (*OS_AppTaskSwHookPtr)();
              }
          #endif

          #if OS_CFG_TASK_PROFILE_EN > 0u
              ts = OS_TS_GET();
              if (OSTCBCurPtr != OSTCBHighRdyPtr) {
```

```
            OSTCBCurPtr->CyclesDelta  = ts - OSTCBCurPtr->CyclesStart;
            OSTCBCurPtr->CyclesTotal += OSTCBCurPtr->CyclesDelta;
        }
        OSTCBHighRdyPtr->CyclesStart = ts;
    #endif

    #ifdef  CPU_CFG_INT_DIS_MEAS_EN
        int_dis_time = CPU_IntDisMeasMaxCurReset();
        if (int_dis_time > OSTCBCurPtr->IntDisTimeMax) {
            OSTCBCurPtr->IntDisTimeMax = int_dis_time;
        }
    #endif

    #if OS_CFG_SCHED_LOCK_TIME_MEAS_EN > 0u
        if (OSTCBCurPtr->SchedLockTimeMax < OSSchedLockTimeMaxCur) {
            OSTCBCurPtr->SchedLockTimeMax = OSSchedLockTimeMaxCur;
        }
        OSSchedLockTimeMaxCur         = (CPU_TS)0;
    #endif

    #if (OS_CFG_TASK_STK_REDZONE_EN == DEF_ENABLED)
        stk_status = OSTaskStkRedzoneChk(DEF_NULL);
        if (stk_status != DEF_OK) {
            OSRedzoneHitHook(OSTCBCurPtr);
        }
    #endif
    }
```

OSTaskTimeQuantaSet

Description

`OSTaskTimeQuantaSet()` is used to change the amount of time a task is given when time slicing multiple tasks running at the same priority.

Files

os.h/os_task.c

Prototype

```
void  OSTaskTimeQuantaSet (OS_TCB   *p_tcb,
                           OS_TICK   time_quanta,
                           OS_ERR   *p_err)
```

Arguments

`p_tcb`

is a pointer to the TCB of the task for which the time quanta is being set. A `NULL` pointer indicates that the user is changing the time quanta for the calling task.

`time_quanta`

specifies the amount of time (in ticks) that the task will run when µC/OS-III is time slicing between tasks at the same priority. Specifying 0 indicates that the default time as specified will be used when calling the function `OSSchedRoundRobinCfg()`, or `OS_CFG_TICK_RATE_HZ / 10` if you never called `OSSchedRoundRobinCfg()`.

You should not specify a “large” value for this argument as this means that the task will execute for that amount of time when multiple tasks are ready-to-run at the same priority. The concept of time slicing is to allow other equal-priority tasks a chance to run. Typical time quanta periods should be approximately 10 ms. A too small value results in more overhead because of the additional context switches.

`p_err`

is a pointer to a variable that will contain an error code returned by this function.

OS_ERR_NONE

If the call was successful and the time quanta for the task was changed.

OS_ERR_SET_ISR

If OS_CFG_CALLED_FROM_ISR_CHK_EN set to DEF_ENABLED in os_cfg.h: if calling this function from an ISR.

Returned Value

None

Required Configuration

OS_CFG_SCHED_ROUND_ROBIN_EN must be enabled in os_cfg.h. Refer to µC-OS-III Configuration Manual.

Callers

Application.

Notes/Warnings

1. Do not specify a large value for time_quanta.

Example Usage

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


              while (DEF_ON) {
                  :
                  :
                  OSTaskTimeQuantaSet((OS_TCB *)0,
                                      OS_CFG_TICK_RATE_HZ / 4;
                                      &err);
                  /* Check “err” */
                  :
                  :
              }
          }
```

API - Miscellaneous

- BSP_OS_TickISR()
- OSCtxSw()
- OSIdleTaskHook()
- OSInit()
- OSInitHook()
- OSIntCtxSw()
- OSIntEnter()
- OSIntExit()
- OSPendMulti() - DEPRECATED
- OSRedzoneHitHook()
- OSSched()
- OSSchedLock()
- OSSchedUnlock()
- OSStart()
- OSStartHighRdy()
- OSStatReset()
- OSStatTaskCPUUsageInit()
- OSStatTaskHook()

- OSVersion()

OS_BSP_TickISR

Description

`BSP_OS_TickISR()` is the interrupt service routine (ISR) associated with the tick interrupt and this function can be written either in assembly language or C depending on the toolchain being used (see also Chapter 9, “Interrupt Management”).

Files

os.h/bsp_os_a.asm or bsp_os.c

Prototype

```
void  BSP_OS_TickISR (void)
```

Arguments

None

Returned Values

None

Required Configuration

None

Callers

Tick interrupt.

Notes/Warnings

None

Example Usage

The code below indicates how to write `BSP_OS_TickISR()` if all interrupts vector to a common location, and the interrupt handler simply calls `BSP_OS_TickISR()`. As indicated, this code can be written completely in C and can be placed in `bsp_os.c` of the board support package (BSP) and be reused by applications using the same BSP.

```
void  BSP_OS_TickISR (void)
          {
              Clear the tick interrupt;
              OSTimeTick();
          }
```

The pseudo code below shows how to write `BSP_OS_TickISR()` if each interrupt directly vectors to its own interrupt handler. The code, in this case, would be written in assembly language and placed either in `bsp_os_a.asm` of the board support package.

```
void  BSP_OS_TickISR (void)
          {
              Save all the CPU registers onto the current task’s stack;
              if (OSIntNestingCtr == 0) {
                  OSTCBCurPtr->StkPtr = SP;
              }
              OSIntEnter();
              Clear the tick interrupt;
              OSTimeTick();
              OSIntExit();
              Restore the CPU registers from the stack;
              Return from interrupt;
          }
```

OSCtxSw 1

Description

`OSCtxSw()` is called from the macro `OS_TASK_SW()`, which in turn is called from `OSSched()` to perform a task-level context switch. Interrupts are disabled when `OSCtxSw()` is called.

Prior to calling `OSCtxSw()`, `OSTCBCurPtr` to point at the `OS_TCB` of the task that is being switched out, and `OSSched()` sets `OSTCBHighRdyPtr` to point at the `OS_TCB` of the task being switched in.

Files

os.h/os_cpu_a.asm

Prototype

```
void  OSCtxSw (void)
```

Arguments

None

Returned Values

None

Required Configuration

None

Callers

`OSSched()`.

Notes/Warnings

None

Example Usage

The pseudocode for `OSCtxSw()` follows:

```
void  OSCtxSw (void)
          {
              Save all CPU registers;                          (1)
              OSTCBCurPtr->StkPtr = SP;                        (2)
              OSTaskSwHook();                                  (3)
              OSPrioCur           = OSPrioHighRdy;             (4)
              OSTCBCurPtr         = OSTCBHighRdyPtr;           (5)
              SP                  = OSTCBHighRdyPtr->StkPtr;   (6)
              Restore all CPU registers;                       (7)
              Return from interrupt;                           (8)
          }
```

(1) `OSCtxSw()` must save all of the CPU registers onto the current task's stack. `OSCtxSw()` is called from the context of the task being switched out. Therefore, the CPU stack pointer is pointing to the proper stack. The user must save all of the registers in the same order as if an ISR started and all the CPU registers were saved on the stack. The stacking order should therefore match that of `OSTaskStkInit()`.

(2) The current task's stack pointer is then saved into the current task's `OS_TCB`.

(3) Next, `OSCtxSw()` must call `OSTaskSwHook()`.

(4) `OSPrioHighRdy` is copied to `OSPrioCur`.

(5) `OSTCBHighRdyPtr` is copied to `OSTCBCurPtr` since the current task is now the task being switched in.

(6) The stack pointer of the new task is restored from the `OS_TCB` of the new task.

(7) All the CPU registers from the new task's stack are restored.

(8) Finally, `OSCtxSw()` must execute a return from interrupt instruction.

OSIdleTaskHook

Description

This function is called by `OS_IdleTask()`.

`OSIdleTaskHook()` is part of the CPU port code and this function *must not* be called by the application code. `OSIdleTaskHook()` is used by the µC/OS-III port developer.

`OSIdleTaskHook()` runs in the context of the idle task and thus it is important to make sure there is sufficient stack space in the idle task. `OSIdleTaskHook()` *must not* make any `OS???Pend()` calls, call `OSTaskSuspend()` or `OSTimeDly???()`. In other words, this function must never be allowed to make a blocking call.

Files

os.h/os_cpu_c.c and os_app_hooks.c

Prototype

```
void  OSIdleTaskHook (void)
```

Arguments

None

Returned Value

None

Required Configuration

`OS_CFG_APP_HOOKS_EN` must be enabled in `os_cfg.h`. Refer to µC-OS-III Configuration Manual.

Callers

`OS_IdleTask()`.

Notes/Warnings

1. Never make blocking calls from `OSIdleTaskHook()`.

2. *Do not* call this function from you application.

Example Usage

The code below calls an application-specific hook that the application programmer can define. The user can simply set the value of `OS_AppIdleTaskHookPtr` to point to the desired hook function which in this case is assumed to be defined in `os_app_hooks.c`. The idle task calls `OSIdleTaskHook()` which in turns calls `App_OS_IdleTaskHook()` through `OS_AppIdleTaskHookPtr`.

This feature is very useful when there is a processor that can enter low-power mode. When µC/OS-III has no other task to run, the processor can be put to sleep waiting for an interrupt to wake it up.

```
void  App_OS_IdleTaskHook (void)                           /* See os_app_hooks.c    */
          {
              /* Your code goes here! */
              /* Put the CPU in low power mode (optional) */
          }


          void App_OS_SetAllHooks (void)                           /* os_app_hooks.c
*/
          {
              CPU_SR_ALLOC();

              CPU_CRITICAL_ENTER();
              :
              OS_AppIdleTaskHookPtr = App_OS_IdleTaskHook;
              :
              CPU_CRITICAL_EXIT();
          }


          void  OSIdleTaskHook (void)                              /* See os_cpu_c.c
*/
          {
          #if OS_CFG_APP_HOOKS_EN > 0u
              if (OS_AppIdleTaskHookPtr != (OS_APP_HOOK_VOID)0) {    /* Call application hook
*/
```

```
            (*OS_AppIdleTaskHookPtr)();
        }
    #endif
    }
```

OSInit

Description

Initializes µC/OS-III and it must be called prior to calling any other µC/OS-III function. Including `OSStart()` which will start multitasking. `OSInit()` returns as soon as an error is detected.

Files

os.h/os_core.c

Prototype

```
void  OSInit (OS_CFG    *p_cfg,
              MEM_SEG   *p_mem_seg,
              RTOS_ERR  *p_err)
```

Arguments

`p_cfg`

is a pointer to a `OS_CFG` structure. See Note #3 below for a description of `OS_CFG`. To support the old `os_cfg_app.h` scheme, `p_cfg` can be `DEF_NULL` if `OS_CFG_COMPAT_INIT_EN` is equal to `DEF_ENABLED`.

`p_mem_seg`

is a pointer to a `MEM_SEG` structure. This segment will be used to allocate the kernel objects. Can be `DEF_NULL` to allocate objects on the heap.

`p_err`

is a pointer to an error code. Some of the error codes below are issued only if the associated feature is enabled.

`RTOS_ERR_NONE`

initialization was successful.

```
*NEED TO COMPLETE*
```

Returned Values

None

Required Configuration

None

Callers

Application.

Notes/Warnings

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

2. `OSInit()` returns as soon as it detects an error in any of the sub-functions it calls. For example, if `OSInit()` encounters a problem initializing the task manager, an appropriate error code will be returned and `OSInit()` will not go any further. It is therefore important that the user checks the error code before starting multitasking.

3. The listing below shows the `OS_CFG` and `OS_TASK_CFG` structures. The `OS_APP_CFG_???_DFLT` defines in `os.h` contain the default values for every `OS_CFG` field.

```
typedef  struct  os_cfg             OS_CFG;
typedef  struct  os_task_cfg        OS_TASK_CFG;

struct  os_task_cfg {                                      /* Kernel Task 
Configuration.                               */
    OS_PRIO             Prio;                              (1)
    OS_RATE_HZ          RateHz;                            (2)
    CPU_STK_SIZE        StkSize;                           (3)
};

struct  os_cfg {                                           /* Runtime Configuration.
*/
    CPU_STK_SIZE        ISRStkSize;                        (4)
#if (OS_MSG_EN == DEF_ENABLED)
    OS_MSG_SIZE         MsgPoolSize;                       (5)
#endif
#if ((OS_CFG_TASK_IDLE_EN == DEF_ENABLED) || (OS_CFG_STAT_TASK_EN == DEF_ENABLED) || \
     (OS_CFG_TASK_TICK_EN == DEF_ENABLED) || (OS_CFG_TMR_EN == DEF_ENABLED))
    CPU_STK_SIZE        TaskStkLimit;                      (6)
#endif
#if (OS_CFG_TASK_IDLE_EN == DEF_ENABLED)
    CPU_STK_SIZE        IdleTaskStkSize;                   (7)
#endif
#if (OS_CFG_STAT_TASK_EN == DEF_ENABLED)
    OS_TASK_CFG         StatTaskCfg;                       (8)
#endif
#if (OS_CFG_TASK_TICK_EN == DEF_ENABLED)
    OS_TASK_CFG         TickTaskCfg;                       (9)
#endif
#if (OS_CFG_TMR_EN == DEF_ENABLED)
    OS_TASK_CFG         TmrTaskCfg;                        (10)
#endif
};
```

Listing - OS_CFG

(1) Kernel tasks have a configurable priority. The priority must be between `0` and `OS_CFG_PRIO_MAX-2` inclusively.

(2) Kernel tasks have a configurable execution rate. The rate is expressed in hertz.

(3) Kernel tasks have a configurable stack size. The stack size is expressed in `CPU_STK` elements.

(4) If the MCU supports it, a separate stack can be used for the ISRs. This field contains the ISR stack size in `CPU_STK` elements.

(5) If a message pool is needed (either Task Queues or Message Queues are enabled), this field sets the number available messages.

(6) The stack limit field is expressed as a percentage. It is used to indicate that a task's stack has `TaskStkLimit` percent left of unused space.

(7) If the Idle task is enabled, this field hold the Idle task's stack size in `CPU_STK` elements.

(8) If the Stat task is enabled, this field contains the Stat task's configuration. See descriptions **(1)** through **(3)**.

(9) If the Tick task is enabled, this field contains the Tick task's configuration. See descriptions **(1)** through **(3)**.

(10) If Timers are enabled, this field contains the Timer Management task's configuration. See descriptions **(1)** through **(3)**.

Example Usage

```
void main (void)
          {
              RTOS_ERR  err;
              OS_CFG    os_cfg;


              os_cfg.ISRStkSize          = OS_APP_CFG_ISR_STK_SIZE_DFLT;
#if (OS_MSG_EN == DEF_ENABLED)
              os_cfg.MsgPoolSize         = OS_APP_CFG_MSG_POOL_SIZE_DFLT;
#endif
#if ((OS_CFG_TASK_IDLE_EN == DEF_ENABLED) || (OS_CFG_STAT_TASK_EN == DEF_ENABLED) || \
     (OS_CFG_TASK_TICK_EN == DEF_ENABLED) || (OS_CFG_TMR_EN == DEF_ENABLED))
              os_cfg.TaskStkLimit        = OS_APP_CFG_TASK_STK_LIMIT_PCT_EMPTY_DFLT;
#endif
#if (OS_CFG_TASK_IDLE_EN == DEF_ENABLED)
              os_cfg.IdleTaskStkSize     = OS_APP_CFG_IDLE_TASK_STK_SIZE_DFLT;
#endif
#if (OS_CFG_STAT_TASK_EN == DEF_ENABLED)
              os_cfg.StatTaskCfg.Prio    = OS_APP_CFG_STAT_TASK_PRIO_DFLT;
              os_cfg.StatTaskCfg.RateHz  = OS_APP_CFG_STAT_TASK_RATE_HZ_DFLT;
              os_cfg.StatTaskCfg.StkSize = OS_APP_CFG_STAT_TASK_STK_SIZE_DFLT;
#endif
#if (OS_CFG_TASK_TICK_EN == DEF_ENABLED)
              os_cfg.TickTaskCfg.Prio    = OS_APP_CFG_TICK_TASK_PRIO_DFLT;
              os_cfg.TickTaskCfg.RateHz  = OS_APP_CFG_TICK_RATE_HZ_DFLT;
              os_cfg.TickTaskCfg.StkSize = OS_APP_CFG_TICK_TASK_STK_SIZE_DFLT;
#endif
#if (OS_CFG_TMR_EN == DEF_ENABLED)
              os_cfg.TmrTaskCfg.Prio     = OS_APP_CFG_TMR_TASK_PRIO_DFLT;
              os_cfg.TmrTaskCfg.RateHz   = OS_APP_CFG_TMR_TASK_RATE_HZ_DFLT;
              os_cfg.TmrTaskCfg.StkSize  = OS_APP_CFG_TMR_TASK_STK_SIZE_DFLT;
#endif
                 :
                                              /* Initialize µC/OS-III              */
              OSInit(&os_cfg,                 /* Use default values                */
                     (MEM_SEG *)DEF_NULL,     /* Allocate objects on the heap      */
                     &err);

              /* Check “err” */
              :
              :
              OSStart(&err);                  /* Start Multitasking                */
              /* Check “err” */               /* Code not supposed to end up here! */
              :
              :
          }
```

Listing - OSInit() example usage

OSInitHook

Description

`OSInitHook()` is a function that is called by µC/OS-III's initialization code, `OSInit()`. `OSInitHook()` is typically implemented by the port implementer for the processor used. This hook allows the port to be extended to do such tasks as setup exception stacks, floating-point registers, and more. `OSInitHook()` is called at the beginning of `OSInit()`, before any µC/OS-III task and data structure have been initialized.

Files

os.h/os_cpu_c.c

Prototype

```
void  OSInitHook (void)
```

Arguments

None

Returned Values

None

Required Configuration

None

Callers

`OSInit()`.

Notes/Warnings

None

Example Usage

```
void  OSInitHook (void)                                /* See os_cpu_c.c         */
          {
              /* Perform any initialization code necessary by the port */
          }
```

OSIntCtxSw 1

Description

`OSIntCtxSw()` is called from `OSIntExit()` to perform a context switch when all nested interrupts have returned.

Interrupts are disabled when `OSIntCtxSw()` is called.

`OSTCBCurPtr` points at the `OS_TCB` of the task that is switched out when `OSIntCtxSw()` is called and `OSIntExit()` sets `OSTCBHighRdyPtr` to point at the `OS_TCB` of the task that is switched in.

Files

os.h/os_cpu_a.asm

Prototype

```
void  OSIntCtxSw (void)
```

Arguments

None

Returned Values

None

Required Configuration

None

Callers

`OSIntExit()`.

Notes/Warnings

None

Example Usage

The pseudocode for `OSIntCtxSw()` is shown below. Notice that the code does only half of what `OSCtxSw()` did. The reason is that `OSIntCtxSw()` is called from an ISR and it is assumed that all of the CPU registers of the interrupted task were saved at the beginning of the ISR. `OSIntCtxSw()` therefore must only restore the context of the new, high-priority task.

```
void  OSIntCtxSw (void)
          {
              OSTaskSwHook();                                 (1)
              OSPrioCur           = OSPrioHighRdy;            (2)
              OSTCBCurPtr         = OSTCBHighRdyPtr;          (3)
              SP                  = OSTCBHighRdyPtr->StkPtr;  (4)
              Restore all CPU registers;                      (5)
              Return from interrupt;                          (6)
          }
```

(1) `OSIntCtxSw()` must call `OSTaskSwHook()`.

(2) `OSPrioHighRdy` needs to be copied to `OSPrioCur`.

(3) `OSTCBHighRdyPtr` needs to be copied to `OSTCBCurPtr` because the current task will now be the new task.

(4) The stack pointer of the new task is restored from the `OS_TCB` of the new task.

(5) All the CPU registers need to be restored from the new task's stack.

(6) A return from interrupt instruction must be executed.

OSIntEnter

Description

`OSIntEnter()` notifies μC/OS-III that an ISR is being processed. This allows μC/OS-III to keep track of interrupt nesting. `OSIntEnter()` is used in conjunction with `OSIntExit()`. This function is generally called at the beginning of ISRs. Note that on some CPU architectures, it must be written in assembly language (shown below in pseudo code):

```
MyISR:
              Save CPU registers;
              OSIntEnter();                /* Or, OSIntNestingCtr++ */
                  :
              Process ISR;
                  :
              OSIntExit();
              Restore CPU registers;
              Return from interrupt;
```

Files

os.h/os_core.c

Prototype

```
void  OSIntEnter (void)
```

Arguments

None

Returned Values

None

Required Configuration

None

Callers

ISRs only.

Notes/Warnings

1. This function must not be called by task-level code.

2. You can also increment the interrupt-nesting counter (`OSIntNestingCtr`) directly in the ISR to avoid the overhead of the function call/return. It is safe to increment `OSIntNestingCtr` in the ISR since interrupts are assumed to be disabled when `OSIntNestingCtr` is incremented. However, that is not true for all CPU architectures. You need to make sure that interrupts are disabled in the ISR before directly incrementing `OSIntNestingCtr`.

3. It is possible to nest interrupts up to 250 levels deep.

OSIntExit

Description

`OSIntExit()` notifies µC/OS-III that an ISR is complete. This allows µC/OS-III 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. If so, the interrupt returns to the higher priority task instead of the interrupted task.

This function is typically called at the end of ISRs as follows, and on some CPU architectures, it must be written in assembly language (shown below in pseudo code):

```
MyISR:
              Save CPU registers;
              OSIntEnter();
                  :
              Process ISR;
                  :
              OSIntExit();
              Restore CPU registers;
              Return from interrupt;
```

Files

os.h/os_core.c

Prototype

```
void  OSIntExit (void)
```

Arguments

None

Returned Value

None

Required Configuration

None

Callers

ISRs only.

Notes/Warnings

1. This function must not be called by task-level code. Also, if you decide to directly increment `OSIntNestingCtr`, instead of calling `OSIntEnter()`, you must still call `OSIntExit()`.

OSPendMulti

OSRedzoneHitHook

Description

`OSRedzoneHitHook()` is a function called by μC/OS-III's Task Switching Hook, `OSTaskSwHook()`. The hook is called when μC/OS-III determines that the task to be switched out has overflowed its stack. The hook can then cleanly exit the application, report an error, try to fix the stack or simply call the software based exception, `CPU_SW_EXCEPTION()`. The hook, if defined, must ultimately call `CPU_SW_EXCEPTION()` or stop μC/OS-III from executing a corrupted task.

Files

os.h/os_cpu_c.c and os_app_hooks.c

Prototype

```
void  OSRedzoneHitHook (OS_TCB  *p_tcb)
```

Arguments

`p_tcb`

is a pointer to the TCB of the offending task. Note that `p_tcb` can be `NULL`. In that case, the ISR stack is corrupted.

Returned Values

None

Required Configuration

`OS_CFG_APP_HOOKS_EN` must be enabled in `os_cfg.h`. Refer to μC-OS-III Configuration Manual.

Callers

`OSRedzoneHitHook()` through `OSIntExit()` and `OSTaskSwHook()`.

Notes/Warnings

1. If the application hook is defined, it must stop µC/OS-III from executing a corrupted task by, for example, calling the software based exception, `CPU_SW_EXCEPTION()`.

Example Usage

The code below calls an application-specific hook that an application programmer can define. For this, the user can simply set the value of `OS_AppRedzoneHitHookPtr` to point to the desired hook function (see `App_OS_SetAllHooks()` in `os_app_hooks.c`).

In the example below, `OSRedzoneHitHook()` calls `App_OS_RedzoneHitHook()` if the pointer `OS_AppRedzoneHitHookPtr` is set to that function.

```
void  App_OS_RedzoneHitHook (OS_TCB  *p_tcb)                  /* os_app_hooks.c       */
          {
              (void)&p_tcb;
              CPU_SW_EXCEPTION(;);
          }


          void  App_OS_SetAllHooks (void)                            /* os_app_hooks.c
*/
          {
              CPU_SR_ALLOC();

              CPU_CRITICAL_ENTER();
              :
              OS_AppRedzoneHitHookPtr = App_OS_RedzoneHitHook;
              :
              CPU_CRITICAL_EXIT();
          }


          void  OSRedzoneHitHook (OS_TCB  *p_tcb)                    /* os_cpu_c.c
*/
          {
          #if OS_CFG_APP_HOOKS_EN > 0u
              if (OS_AppRedzoneHitHookPtr != (OS_APP_HOOK_TCB)0) {
                  (*OS_AppRedzoneHitHookPtr)(p_tcb);
              }
          #endif
              (void)p_tcb;
              CPU_SW_EXCEPTION(;);
          }
```

OSSafetyCriticalStart

OSSched

Description

`OSSched()` allows a task to call the scheduler. You would use this function after doing a series of "posts" where you specified `OS_OPT_POST_NO_SCHED` as a post option.

`OSSched()` can only be called by task-level code. Also, if the scheduler is locked (i.e., `OSSchedLock()` was previously called), then `OSSched()` will have no effect.

If a higher-priority task than the calling task is ready-to-run, `OSSched()` will context switch to that task.

Files

os.h/os_core.c

Prototype

```
void  OSSched (void)
```

Arguments

None

Returned Value

None

Required Configuration

None

Callers

Application.

Notes/Warnings

None

Example Usage

```
void TaskX (void *p_arg)
          {
              (void)&p_arg;
              while (DEF_ON) {
                  :
                  OS??Post(…);         /* Posts with OS_OPT_POST_NO_SCHED option         */
                  /* Check “err” */
                  :
                  :
                  OS??Post(…);
                  /* Check “err” */
                  :
                  :
                  OS??Post(…);
                  /* Check “err” */
                  :
                  :
                  OSSched();           /* Run the scheduler                              */
                  :
                  :
              }
          }
```

OSSchedLock

Description

`OSSchedLock()` prevents task rescheduling until its counterpart, `OSSchedUnlock()`, is called. The task that calls `OSSchedLock()` retains 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-III allows `OSSchedLock()` to be nested up to 250 levels deep. Scheduling is enabled when an equal number of `OSSchedUnlock()` calls have been made.

Files

os.h/os_core.c

Prototype

```
void  OSSchedLock (OS_ERR  *p_err)
```

Arguments

`p_err`

is a pointer to a variable that will contain an error code returned by this function.

`OS_ERR_NONE`

If the scheduler is locked.

`OS_ERR_LOCK_NESTING_OVF`

If the user attempted to nest the locking more than 250 times.

`OS_ERR_OS_NOT_RUNNING`

If the function is called before calling `OSStart()`.

`OS_ERR_SCHED_LOCK_ISR`

If `OS_CFG_CALLED_FROM_ISR_CHK_EN` set to `DEF_ENABLED` in `os_cfg.h`: if you attempted to call `OSSchedLock()` from an ISR.

Returned Value

None

Required Configuration

None

Callers

Application.

Notes/Warnings

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

Example Usage

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


              (void)&p_arg;
              while (DEF_ON) {
                  :
                  OSSchedLock(&err);     /* Prevent other tasks to run         */
                  /* Check “err” */
                  :
                  :                      /* Code protected from context switch */

                  :
                  OSSchedUnlock(&err);   /* Enable other tasks to run          */
                  /* Check “err” */
                  :
                  :
              }
          }
```


OSSchedUnlock

Description

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

Files

os.h/os_core.c

Prototype

```
void  OSSchedUnlock (OS_ERR  *p_err)
```

Arguments

`p_err`

is a pointer to a variable that will contain an error code returned by this function.

`OS_ERR_NONE`

the call is successful and the scheduler is no longer locked.

`OS_ERR_OS_NOT_RUNNING`

If calling this function before calling `OSStart()`.

`OS_ERR_SCHED_LOCKED`

If the scheduler is still locked. This would indicate that scheduler lock has not fully unnested.

`OS_ERR_SCHED_NOT_LOCKED`

If the user did not call `OSSchedLock()`.

`OS_ERR_SCHED_UNLOCK_ISR`

If `OS_CFG_CALLED_FROM_ISR_CHK_EN` set to `DEF_ENABLED` in `os_cfg.h`: if you attempted to unlock scheduler from an ISR.

Returned Value

None

Required Configuration

None

Callers

Application.

Notes/Warnings

None

Example Usage

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


              (void)&p_arg;
              while (DEF_ON) {
                  :
                  OSSchedLock(&err);     /* Prevent other tasks to run         */
                  /* Check “err” */
                  :
                  :                      /* Code protected from context switch */

                  :
                  OSSchedUnlock(&err);   /* Enable other tasks to run          */
                  /* Check “err” */
                  :
                  :
              }
          }
```

OSStart

Description

Starts multitasking under µC/OS-III. This function is typically called from startup code after calling `OSInit()` and creating at least one application task. `OSStart()` will not return to the caller. Once µC/OS-III is running, calling `OSStart()` again will have no effect.

Files

os.h/os_core.c

Prototype

```
void  OSStart (OS_ERR  *p_err)
```

Arguments

`p_err`

is a pointer to a variable used to hold an error code:

`OS_ERR_FATAL_RETURN`

If we ever return to this function.

`OS_ERR_OS_NOT_INIT`

µC/OS-III not initialized.

`OS_ERR_OS_NO_APP_TASK`

No application task created.

`OS_ERR_OS_RUNNING`

If the kernel is already running. In other words, if this function has already been called.

Returned Value

None

Required Configuration

None

Callers

Application.

Notes/Warnings

1. `OSInit()` must be called prior to calling `OSStart()`. `OSStart()` should only be called once by the application code. However, if you called `OSStart()` more than once, nothing happens on the second and subsequent calls.

Example Usage

```
void main (void)
          {
              OS_ERR  err;

                                                  /* User Code            */
              :
              OSInit(&err);                       /* Initialize µC/OS-III */
              /* Check “err” */
              :                                   /* User Code            */
              :
              OSStart(&err);                      /* Start Multitasking   */
              /* Any code here should NEVER be executed! */
          }
```

Listing - OSStart() example usage

OSStartHighRdy

Description

`OSStartHighRdy()` is responsible for starting the highest-priority task that was created prior to calling `OSStart()`. `OSStartHighRdy()` is a µC/OS-III port function that is generally written in assembly language.

Files

os.h/os_cpu_a.asm

Prototype

```
void  OSStartHighRdy (void)
```

Arguments

None

Returned Values

None

Required Configuration

None

Callers

`OSStart()`.

Notes/Warnings

None

Example Usage

The pseudocode for `OSStartHighRdy()` is shown below.

```
OSStartHighRdy:
              OSTaskSwHook();                              (1)
              SP = OSTCBHighRdyPtr->StkPtr;                (2)
              Pop CPU registers off the task's stack;      (3)
              Return from interrupt;                       (4)
```

(1) `OSStartHighRdy()` must call `OSTaskSwHook()`.

When called, `OSTCBCurPtr` and `OSTCBHighRdyPtr` both point to the `OS_TCB` of the highest-priority task created.

`OSTaskSwHook()` should check that `OSTCBCurPtr` is not equal to `OSTCBHighRdyPtr` as this is the first time `OSTaskSwHook()` is called and there is not a task being switched out.

(2) The CPU stack pointer register is loaded with the top-of-stack (TOS) of the task being started. The TOS is found in the `.StkPtr` field of the `OS_TCB`. For convenience, the `.StkPtr` field is the very first field of the `OS_TCB` data structure. This makes it easily accessible from assembly language.

(3) The registers are popped from the task's stack frame. Recall that the registers should have been placed on the stack frame in the same order as if they were pushed at the beginning of an interrupt service routine.

(4) You must execute a return from interrupt. This starts the task as if it was resumed when returning from a real interrupt.

OSStatReset

Description

`OSStatReset()` is used to reset statistical variables maintained by µC/OS-III. Specifically, the per-task maximum interrupt disable time, maximum scheduler lock time, maximum amount of time a message takes to reach a task queue, the maximum amount of time it takes a signal to reach a task and more.

Files

os.h/os_stat.c

Prototype

```
void  OSStatReset (OS_ERR  *p_err)
```

Arguments

`p_err`

is a pointer to a variable used to hold an error code:

`OS_ERR_NONE`

The call was successful.

Returned Value

None

Required Configuration

`OS_CFG_STAT_TASK_EN` must be enabled in `os_cfg.h`. Refer to µC-OS-III Configuration Manual.

Callers

Application.

Notes/Warnings

None

Example Usage

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

              (void)&p_arg;
              while (DEF_ON) {
                  :
                  :
                  if (statistics reset switch is pressed) {
                      OSStatReset(&err);
                      /* Check “err” */
                      :
                      :
                  }
                  :
                  :
              }
          }
```

OSStatTaskCPUUsageInit

Description

`OSStatTaskCPUUsageInit()` 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 the application and when multitasking has started. This function must be called from the first and only task created by the application.

Files

os.h/os_stat.c

Prototype

```
void  OSStatTaskCPUUsageInit (OS_ERR  *p_err)
```

Arguments

`p_err`

is a pointer to a variable used to hold an error code:

`OS_ERR_NONE`

Counter maximum value obtained.

`OS_ERR_OS_NOT_RUNNING`

If `OS_CFG_INVALID_OS_CALLS_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if µC/OS-III is not running yet.

Returned Value

None

Required Configuration

OS_CFG_TASK_STAT_EN must be enabled in os_cfg.h. Refer to µC-OS-III Configuration Manual.

Callers

Application.

Notes/Warnings

None

Example Usage

```
void FirstAndOnlyTask (void *p_arg)
          {
              OS_ERR  err;
              :
              :
          #if OS_CFG_TASK_STAT_EN == DEF_ENABLED
              OSStatTaskCPUUsageInit(&err); /* Compute CPU capacity with no task running */
          #endif
              :
              OSTaskCreate(_);              /* Create the other tasks                     */
              OSTaskCreate(_);
              :
              while (DEF_ON) {
                  :
                  :
              }
          }
```


OSStatTaskHook

Description

`OSStatTaskHook()` is a function called by µC/OS-III's statistic task, `OSStatTask()`. `OSStatTaskHook()` is generally implemented by the port implementer for the processor used. This hook allows the port to perform additional statistics.

Files

os.h/os_cpu_c.c and os_app_hooks.c

Prototype

```
void  OSStatTaskHook (void)
```

Arguments

None

Returned Values

None

Required Configuration

`OS_CFG_APP_HOOKS_EN` must be enabled in `os_cfg.h`. Refer to µC-OS-III Configuration Manual.

Callers

`OSStatTask()`.

Notes/Warnings

None

Example Usage

The code below calls an application-specific hook that an application programmer can define. For this, the user can simply set the value of `OS_AppStatTaskHookPtr` to point to the desired hook function (see `App_OS_SetAllHooks()` in `os_app_hooks.c`).

In the example below, `OSStatTaskHook()` calls `App_OS_StatTaskHook()` if the pointer `OS_AppStatTaskHookPtr` is set to that function.

```
void  App_OS_StatTaskHook (void)                           /* os_app_hooks.c        */
          {
              /* Your code goes here! */
          }


          void App_OS_SetAllHooks (void)                             /* os_app_hooks.c
*/
          {
              CPU_SR_ALLOC();

              CPU_CRITICAL_ENTER();
              :
              OS_AppStatTaskHookPtr = App_OS_StatTaskHook;
              :
              CPU_CRITICAL_EXIT();
          }


          void  OSStatTaskHook (void)                                /* os_cpu_c.c
*/
          {
          #if OS_CFG_APP_HOOKS_EN > 0u
              if (OS_AppStatTaskHookPtr != (OS_APP_HOOK_VOID)0) {    /* Call application hook
*/
                  (*OS_AppStatTaskHookPtr)();
              }
          #endif
          }
```


OSVersion

Description

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

Files

os.h/os_core.c

Prototype

```
CPU_INT16U  OSVersion (OS_ERR  *p_err)
```

Arguments

`p_err`

is a pointer to a variable that contains an error code returned by this function. Currently, `OSVersion()` always returns:

`OS_ERR_NONE`

The call succeeded.

Returned Value

The version is returned as `x.yy.zz` multiplied by `10,000`. For example, V3.00.00 is returned as `30000`.

Required Configuration

None

Callers

Application and ISRs.

Notes/Warnings

None

Example Usage

```
void TaskX (void *p_arg)
          {
              CPU_INT16U  os_version;
              OS_ERR      err;


              while (DEF_ON) {
                  :
                  :
                  os_version = OSVersion(&err);  /* Obtain μC/OS-III's version       */
              }
          }
```

API - Task Message Queues

- OSTaskQFlush()
- OSTaskQPend()
- OSTaskQPendAbort()
- OSTaskQPost()

OSTaskQFlush

Description

`OSTaskQFlush()` empties the contents of the task message queue and eliminates all messages sent to the queue. `OS_MSGs` from the queue are simply returned to the free pool of `OS_MSGs`.

Files

os.h/os_task.c

Prototype

```
OS_MSG_QTY  OSTaskQFlush (OS_TCB  *p_tcb,
                          OS_ERR  *p_err)
```

Arguments

`p_tcb`

is a pointer to the TCB of the task that contains the queue to flush. Specifying a `NULL` pointer tells `OSTaskQFlush()` to flush the queue of the calling task's built-in message queue.

`p_err`

is a pointer to a variable that will contain an error code returned by this function.

`OS_ERR_NONE`

If the message queue is flushed.

`OS_ERR_FLUSH_ISR`

If `OS_CFG_CALLED_FROM_ISR_CHK_EN` set to `DEF_ENABLED` in `os_cfg.h`: if calling this function from an ISR

`OS_ERR_OS_NOT_RUNNING`

If `OS_CFG_INVALID_OS_CALLS_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if µC/OS-III is not running yet.

Returned Value

The number of `OS_MSG` entries freed from the message queue. Note that the `OS_MSG` entries are returned to the free pool of `OS_MSGs`.

Required Configuration

`OS_CFG_TASK_Q_EN` must be enabled in `os_cfg.h`. Refer to µC-OS-III Configuration Manual.

Callers

Application.

Notes/Warnings

1. Use this function with great care. When flushing a queue, you lose the references to what the queue entries are pointing to, potentially causing 'memory leaks'. The data that the user is pointing to that is referenced by the queue entries should, most likely, be de-allocated (i.e., freed).

Example Usage

```
void Task (void *p_arg)
          {
              OS_ERR      err;
              OS_MSG_QTY  entries;

              (void)&p_arg;
              while (DEF_ON) {
                  :
                  :
                  entries = OSTaskQFlush((OS_TCB *)0,
                                         &err);
                  /* Check “err” */
                  :
                  :
              }
          }
```

or, to flush a queue that contains entries, instead you can use `OSTaskQPend()` and specify the `OS_OPT_PEND_NON_BLOCKING` option.

```
void Task (void *p_arg)
          {
              OS_ERR       err;
              CPU_TS       ts;
              OS_MSG_SIZE  msg_size;


              (void)&p_arg;
              :
              do {
                  OSTaskQPend(0,
                              OS_OPT_PEND_NON_BLOCKING,
                             &msg_size,
                             &ts,
                             &err);
              } while (err != OS_ERR_PEND_WOULD_BLOCK);
              :
              :
          }
```

OSTaskQPend

Description

`OSTaskQPend()` allows a task to receive messages directly from an ISR or another task, without going through an intermediate message queue. In fact, each task has a built-in message queue if the configuration constant `OS_CFG_TASK_Q_EN` is set to `DEF_ENABLED`. The messages received are pointer-sized variables, and their use is application specific. If at least one message is already present in the message queue when `OSTaskQPend()` is called, the message is retrieved and returned to the caller.

If no message is present in the task's message queue and `OS_OPT_PEND_BLOCKING` is specified for the `opt` argument, `OSTaskQPend()` suspends the current task (assuming the scheduler is not locked) until either a message is received, or a user-specified timeout expires. A pended task that is suspended with `OSTaskSuspend()` can receive messages. However, the task remains suspended until it is resumed by calling `OSTaskResume()`.

If no message is present in the task's message queue and `OS_OPT_PEND_NON_BLOCKING` is specified for the `opt` argument, `OSTaskQPend()` returns to the caller with an appropriate error code and returns a `NULL` pointer.

Files

os.h/os_task.c

Prototype

```
void  *OSTaskQPend (OS_TICK       timeout,
                    OS_OPT        opt,
                    OS_MSG_SIZE  *p_msg_size,
                    CPU_TS       *p_ts,
                    OS_ERR       *p_err)
```

Arguments

`timeout`

allows the task to resume execution if a message is not received from a task or an ISR within the specified number of clock ticks. A timeout value of 0 indicates that the task wants to wait forever for a 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.

`opt`

determines whether or not the user wants to block if a message is not available in the task's queue. This argument must be set to either:

`OS_OPT_PEND_BLOCKING`, or

`OS_OPT_PEND_NON_BLOCKING`

Note that the timeout argument should be set to 0 when `OS_OPT_PEND_NON_BLOCKING` is specified, since the timeout value is irrelevant using this option.

`p_msg_size`

is a pointer to a variable that will receive the size of the message.

`p_ts`

is a pointer to a timestamp indicating when the task's queue was posted, or the pend aborted. Passing a `NULL` pointer is valid and indicates that the timestamp is not necessary.

A timestamp is useful when the task must know when the task message queue was posted, or how long it took for the task to resume after the task message queue was posted. In the latter case, call `OS_TS_GET()` and compute the difference between the current value of the timestamp and `*p_ts`. In other words:

```
delta = OS_TS_GET() - *p_ts;
```

`p_err`

is a pointer to a variable used to hold an error code.

`OS_ERR_NONE`

If a message is received.

`OS_ERR_OPT_INVALID`

If `OS_CFG_ARG_CHK_EN` set to `DEF_ENABLED` in `os_cfg.h`: if you specified an invalid option.

`OS_ERR_OS_NOT_RUNNING`

If `OS_CFG_INVALID_OS_CALLS_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if µ C/OS-III is not running yet.

`OS_ERR_PEND_ABORT`

If the pend was aborted because another task called `OSTaskQPendAbort()`.

`OS_ERR_PEND_ISR`

If `OS_CFG_CALLED_FROM_ISR_CHK_EN` set to `DEF_ENABLED` in `os_cfg.h`: if calling this function from an ISR.

`OS_ERR_PEND_WOULD_BLOCK`

If calling this function with the opt argument set to `OS_OPT_PEND_NON_BLOCKING` and no message is in the task's message queue.

`OS_ERR_PTR_INVALID`

If `OS_CFG_ARG_CHK_EN` set to `DEF_ENABLED` in `os_cfg.h`: if `p_msg_size` is a `NULL` pointer.

`OS_ERR_SCHED_LOCKED`

If calling this function when the scheduler is locked and the user wanted to block.

`OS_ERR_TIMEOUT`

If a message is not received within the specified timeout.

Returned Value

The message if no error or a NULL pointer upon error. You should examine the error code since it is possible to send NULL pointer messages. In other words, a NULL pointer does not mean an error occurred. *p_err must be examined to determine the reason for the error.

Required Configuration

OS_CFG_TASK_Q_EN must be enabled in os_cfg.h. Refer to µC-OS-III Configuration Manual.

Callers

Application.

Notes/Warnings

1. Do not call OSTaskQPend() from an ISR.

Example Usage

```
void  CommTask (void *p_arg)
          {
              OS_ERR        err;
              void         *p_msg;
              OS_MSG_SIZE   msg_size;
              CPU_TS        ts;


              (void)&p_arg;
              while (DEF_ON) {
                  :
                  :
                  p_msg = OSTaskQPend(100,
                                      OS_OPT_PEND_BLOCKING,
                                      &msg_size,
                                      &ts,
                                      &err);
                  /* Check "err" */
                  :
                  :
              }
          }
```


OSTaskQPendAbort

Description

`OSTaskQPendAbort()` aborts and readies a task currently waiting on its built-in message queue. This function should be used to fault-abort the wait on the task's message queue, rather than to normally signal the message queue via `OSTaskQPost()`.

Files

os.h/os_task.c

Prototype

```
CPU_BOOLEAN  OSTaskQPendAbort (OS_TCB  *p_tcb,
                               OS_OPT   opt,
                               OS_ERR  *p_err)
```

Arguments

`p_tcb`

is a pointer to the task for which the pend needs to be aborted. Note that it doesn't make sense to pass a `NULL` pointer or the address of the calling task's TCB since, by definition, the calling task cannot be pending.

`opt`

provides options for this function.

`OS_OPT_POST_NONE`

No option specified.

`OS_OPT_POST_NO_SCHED`

specifies that the scheduler should not be called even if the pend of a higher priority task has been aborted. Scheduling will need to occur from another function.

Use this option if the task calling `OSTaskQPendAbort()` will do additional pend aborts, rescheduling will take place when completed, and multiple pend aborts should take effect simultaneously.

`p_err`

is a pointer to a variable that holds an error code:

`OS_ERR_NONE`

the task was readied by another task and it was informed of the aborted wait.

`OS_ERR_OPT_INVALID`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if a valid option is not specified.

`OS_ERR_OS_NOT_RUNNING`

If `OS_CFG_INVALID_OS_CALLS_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if µ C/OS-III is not running yet.

`OS_ERR_PEND_ABORT_ISR`

If `OS_CFG_CALLED_FROM_ISR_CHK_EN` set to `DEF_ENABLED` in `os_cfg.h`: if called from an ISR

`OS_ERR_PEND_ABORT_NONE`

If the task was not pending on the task's message queue.

`OS_ERR_PEND_ABORT_SELF`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if `p_tcb` is a `NULL` pointer. The user is attempting to pend abort the calling task which makes no sense as the caller, by definition, is not pending.

Returned Value

`OSTaskQPendAbort()` returns `DEF_TRUE` if the task was made ready-to-run by this function. `DEF_FALSE` indicates that the task was not pending, or an error occurred.

Required Configuration

`OS_CFG_TASK_Q_EN` and `OS_CFG_TASK_Q_PEND_ABORT_EN` must be enabled in `os_cfg.h`. Refer to µC-OS-III Configuration Manual.

Callers

Application.

Notes/Warnings

None

Example Usage

```
OS_TCB  CommRxTaskTCB;


          void CommTask (void *p_arg)
          {
              OS_ERR       err;
              CPU_BOOLEAN  aborted;


              (void)&p_arg;
              while (DEF_ON) {
                  :
                  :
                  aborted = OSTaskQPendAbort(&CommRxTaskTCB,
                                             OS_OPT_POST_NONE,
                                            &err);
                  /* Check "err" */
                  :
                  :
              }
          }
```

OSTaskQPost

Description

`OSTaskQPost()` sends a message to a task through its local message queue. A message is a pointer-sized variable, and its use is application specific. If the task's message queue is full, an error code is returned to the caller. In this case, `OSTaskQPost()` immediately returns to its caller, and the message is not placed in the message queue.

If the task receiving the message is waiting for a message to arrive, it will be made ready-to-run. If the receiving task 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. A message can be posted as first-in first-out (FIFO), or last-in-first-out (LIFO), depending on the value specified in the `opt` argument. In either case, scheduling occurs unless `opt` is set to `OS_OPT_POST_NO_SCHED`.

Files

os.h/os_task.c

Prototype

```
void  OSTaskQPost (OS_TCB      *p_tcb,
                   void        *p_void,
                   OS_MSG_SIZE  msg_size,
                   OS_OPT       opt,
                   OS_ERR      *p_err)
```

Arguments

`p_tcb`

is a pointer to the TCB of the task. Note that it is possible to post a message to the calling task (i.e., self) by specifying a `NULL` pointer, or the address of its TCB.

`p_void`

is the actual message sent to the task. `p_void` is a pointer-sized variable and its meaning is application specific.

`msg_size`

specifies the size of the message posted (in number of bytes).

`opt`

determines the type of POST performed. Of course, it does not make sense to post LIFO and FIFO simultaneously, so these options are exclusive:

`OS_OPT_POST_FIFO`

POST message to task and place at the end of the queue if the task is not waiting for messages.

`OS_OPT_POST_LIFO`

POST message to task and place at the front of the queue if the task is not waiting for messages.

`OS_OPT_POST_NO_SCHED`

This option prevents calling the scheduler after the post and therefore the caller is resumed.

You should use this option if the task (or ISR) calling `OSTaskQPost()` will be doing additional posts, the user does not want to reschedule until all done, and multiple posts are to take effect simultaneously.

`p_err`

is a pointer to a variable that will contain an error code returned by this function.

`OS_ERR_NONE`

If the call was successful and the message was posted to the task's message queue.

`OS_ERR_MSG_POOL_EMPTY`

If running out of `OS_MSG` to hold the message being posted.

`OS_ERR_OPT_INVALID`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if a valid option is not specified.

`OS_ERR_OS_NOT_RUNNING`

If `OS_CFG_INVALID_OS_CALLS_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if µC/OS-III is not running yet.

`OS_ERR_Q_MAX`

If the task's message queue is full and cannot accept more messages.

`OS_ERR_STATE_INVALID`

If the task is in an invalid state.

Returned Value

None

Required Configuration

`OS_CFG_TASK_Q_EN` must be enabled in `os_cfg.h`. Refer to µC-OS-III Configuration Manual.

Callers

Application and ISRs.

Notes/Warnings

None

Example Usage

```
OS_TCB       CommRxTaskTCB;
CPU_INT08U   CommRxBuf[100];


void CommTaskRx (void *p_arg)
{
    OS_ERR  err;


    (void)&p_arg;
    while (DEF_ON) {
        :
        OSTaskQPost(&CommRxTaskTCB,
                    (void *)&CommRxBuf[0],
                    sizeof(CommRxBuf),
                    OS_OPT_POST_FIFO,
                    &err);
        /* Check “err” */
        :
        :
    }
}
```

API - Monitors

- OSMonCreate()
- OSMonDel()
- OSMonOp()

OSMonCreate

Description

Creates a monitor.

Files

os.h/os_mon.c

Prototype

```
void  OSMonCreate (OS_MON    *p_mon,
                   CPU_CHAR  *p_name,
                   void      *p_mon_data,
                   OS_ERR    *p_err)
```

Arguments

`p_mon`

is a pointer to the monitor to initialize. Your application is responsible for allocating storage for the monitor.

`p_name`

is a pointer to an ASCII string to the name you would like to give the monitor.

`p_mon_data`

is a pointer to the monitor global data.

`p_err`

is a pointer to a variable used to hold an error code:

`OS_ERR_NONE`

If the call is successful and the monitor has been created.

`OS_ERR_CREATE_ISR`

If `OS_CFG_CALLED_FROM_ISR_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: you attempted to create a monitor from an ISR.

`OS_ERR_ILLEGAL_CREATE_RUN_TIME`

If `OS_SAFETY_CRITICAL_IEC61508` is defined: you called this after calling `OSStart()` and thus you are no longer allowed to create additional kernel objects.

`OS_ERR_OBJ_PTR_NULL`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if `p_mon` is a `NULL` pointer.

Returned Value

None

Required Configuration

`OS_CFG_MON_EN` must be enabled in `os_cfg.h`. Refer to µC-OS-III Configuration Manual.

Callers

Application.

Notes/Warnings

1. Monitors must be created before they are used.

Example Usage

```
OS_MON      AppMon;
          CPU_INT32U  AppState;


          void main (void)
          {
              OS_ERR  err;


              :
              OSInit(&err);                 /* Initialize µC/OS-III                 */
              :
              :
                                            /* Create Monitor.                      */
              OSMonCreate(&AppMon,          /*   Pointer to user allocated monitor. */
                          "App Monitor",    /*   Name used for debugging.           */
                          (void *)&AppState,/*   Global monitor data: the State.    */
                          &err);
              /* Check “err” */
              :
              :
              OSStart(&err);                /* Start Multitasking                   */
          }
```

Listing - OSMonCreate() example usage

OSMonDel

Description

Deletes a monitor.

Files

os.h/os_mon.c

Prototype

```
OS_OBJ_QTY  OSMonDel (OS_MON  *p_mon,
                      OS_OPT   opt,
                      OS_ERR  *p_err)
```

Arguments

`p_mon`

is a pointer to the monitor to delete.

`opt`

specifies whether the user wants to delete the monitor only if there are no pending tasks (`OS_OPT_DEL_NO_PEND`), or whether the monitor should always be deleted regardless of whether or not tasks are pending (`OS_OPT_DEL_ALWAYS`). In this case, all pending task are readied.

`p_err`

is a pointer to a variable used to hold an error code:

`OS_ERR_NONE`

If the call is successful and the monitor has been deleted.

`OS_ERR_DEL_ISR`

If OS_CFG_CALLED_FROM_ISR_CHK_EN set to DEF_ENABLED in os_cfg.h: if attempting to delete the monitor from an ISR.

OS_ERR_ILLEGAL_DEL_RUN_TIME

If OS_SAFETY_CRITICAL_IEC61508 is defined: you called this after calling OSStart() and thus you are no longer allowed to delete kernel objects.

OS_ERR_OBJ_PTR_NULL

If OS_CFG_ARG_CHK_EN is set to DEF_ENABLED in os_cfg.h: if p_mon is a NULL pointer.

OS_ERR_OBJ_TYPE

If OS_CFG_OBJ_TYPE_CHK_EN is set to DEF_ENABLED in os_cfg.h: if p_mon is not pointing to a semaphore.

OS_ERR_OPT_INVALID

If OS_CFG_ARG_CHK_EN is set to DEF_ENABLED in os_cfg.h: if one of the two options mentioned in the opt argument is not specified.

OS_ERR_OS_NOT_RUNNING

If OS_CFG_INVALID_OS_CALLS_CHK_EN is set to DEF_ENABLED in os_cfg.h: if µC/OS-III is not running yet.

OS_ERR_TASK_WAITING

If one or more tasks are waiting on the monitor and the OS_OPT_DEL_NO_PEND option was specified.

Returned Value

None

Required Configuration

`OS_CFG_MON_EN` and `OS_CFG_MON_DEL_EN` must be enabled in `os_cfg.h`. Refer to µC-OS-III Configuration Manual.

Callers

Application.

Notes/Warnings

1. Use this call with care because other tasks might expect the presence of the monitor.

Example Usage

```
OS_MON  AppMon;


void Task (void *p_arg)
{
    OS_ERR      err;
    OS_OBJ_QTY  qty;


    (void)&p_arg;
    while (DEF_ON) {
        :
        :
        qty = OSMonDel(&AppMon,
                        OS_OPT_DEL_ALWAYS,
                       &err);
        /* Check “err” */
        :
        :
    }
}
```

Listing - OSMonDel() example usage

OSMonOp

Description

Operates a monitor. Depending on the callbacks used, the calling task can block and the pending tasks could wake up.

Files

os.h/os_mon.c

Prototype

```
void  OSMonOp (OS_MON                *p_mon,
               OS_TICK                timeout,
               void                  *p_arg,
               OS_MON_ON_ENTER_PTR    p_on_enter,
               OS_MON_ON_EVAL_PTR     p_on_eval,
               OS_OPT                 opt,
               OS_ERR                *p_err)
```

Arguments

`p_mon`

is a pointer to the monitor.

`timeout`

optional timeout to be applied if the monitor blocks (pend).

`p_arg`

argument passed to the optional enter callback and to all registered evaluation callbacks.

`p_on_enter`

pointer to the enter callback. If not null, this function will be called before calling all registered evaluation callbacks. The callback function must be declared as follows:

```
OS_MON_RES MyEnterCallback (OS_MON  *p_mon, void  *p_data);
```

When the enter callback is called, the `p_mon` argument points to the monitor object and the `p_data` argument points to the monitor's argument. In this case its this function's `p_arg`.

`p_on_eval`

pointer to the evaluation callback. If not null, this callback will be called everytime there is another operation performed on the monitor. The callback function must be declared as follows:

```
OS_MON_RES MyEvalCallback (OS_MON  *p_mon, void  *p_eval_data, void
 *p_arg);
```

When the evaluation callback is called, the `p_mon` argument points to the monitor object and the `p_arg` arguments points to the `p_arg` parameter passed to the `OSMonOp()` call that caused this evaluation callback to be executed. The `p_eval_data` argument points to the `p_arg` parameter passed to the `OSMonOp()` call that blocked the task registered with this callback,

opt

the only available option is `OS_OPT_POST_NO_SCHED` to prevent the scheduler from being called if any of the pending tasks woke up after evaluation.

`p_err`

is a pointer to a variable used to hold an error code:

`OS_ERR_NONE`

The call was successful.

`OS_ERR_OBJ_DEL`

The monitor was deleted while pending.

`OS_ERR_OBJ_PTR_NULL`

If `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if `p_mon` is a `NULL` pointer.

`OS_ERR_OS_NOT_RUNNING`

If `OS_CFG_INVALID_OS_CALLS_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`: if µC/OS-III is not running yet.

`OS_ERR_PEND_ABORT`

If the operation was aborted by another task.

`OS_ERR_STATUS_INVALID`

If the pend status has an invalid value.

`OS_ERR_TIMEOUT`

The monitor did not evaluate successfully within the specified timeout.

Returned Value

None

Required Configuration

`OS_CFG_MON_EN` must be enabled in `os_cfg.h`. Refer to µC-OS-III Configuration Manual.

Callers

Application.

Notes/Warnings

None

Example Usage

```
OS_MON  App_Mon;
              :
              :
                                                                /* Application States.
*/
#define  APP_MON_PEND_ON_STATE                      0xC0FFEEu
              :
              :
OS_MON_RES  App_MonPend (OS_MON  *p_mon,
                         void    *p_eval_data,
                         void    *p_arg)
{
    CPU_INT32U  *p_state;
    OS_MON_RES   res;


                                                                /* Prevent compiler warning.
*/
    (void)(p_arg);
                                                                /* Get State variable.
*/
    p_state = (CPU_INT32U *)p_mon->MonDataPtr;
                                                                /* If new state matches, wake
up the pending task.      */
    if (*p_state == (CPU_INT32U)p_eval_data) {
        res = OS_MON_RES_ALLOW;
    } else {
        res = OS_MON_RES_BLOCK;
    }

    return (res);
}

OS_MON_RES  App_MonPost (OS_MON  *p_mon,
                         void    *p_data)
{
    CPU_INT32U *p_state;


                                                                /* Get State variable.
*/
    p_state = (CPU_INT32U *)p_mon->MonDataPtr;
                                                                /* Change state.
*/
    *p_state = (CPU_INT32U)p_data;

    return (OS_MON_RES_ALLOW);
}
              :
              :
void  App_SomeTask (void  *p_arg)
{
    RTOS_ERR  err;
              :
              :
                                                                /* Prevent compiler warning.
*/
    (void)(p_arg);
                                                                /* Wait for State.
*/
    OSMonOp(&App_Mon,                                           /*   Pointer to user allocated
monitor.                 */
             500,                                               /*   Timeout: 500 OS Ticks.
*/
            (void *)APP_MON_PEND_ON_STATE,                      /*   Wait for this state.
*/
             DEF_NULL,                                          /*   Function used on entering
```

```
the monitor.             */
            &App_MonPend,                                  /*   Function used for
evaluation.                      */
             0,                                            /*   Option: none.
*/
            &err);
    /* Check 'err'. */
              :
              :
}

void  App_SomeOtherTask (void  *p_arg)
{
    OS_ERR  err;
              :
              :
                                                           /* Prevent compiler warning.
*/
    (void)(p_arg);
                                                           /* Change State.
*/
    OSMonOp(&App_Mon,                                      /*   Pointer to user allocated
monitor.                 */
             0,                                            /*   Timeout: none, this is a
post operation.           */
            (void *)APP_MON_PEND_ON_STATE,                 /*   Change state to new value.
*/
            &App_MonPost,                                  /*   Function used on entering
the monitor.             */
             DEF_NULL,                                     /*   Function used for
evaluation.                      */
             0,                                            /*   Option: none.
*/
            &err);
    /* Check 'err'. */
              :
              :
}
```

Listing - OSMonOp() example usage