µC/OS-II

(The Real-Time Kernel)

Reference Manual

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

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

OS_ENTER_CRITICAL()
OS_EXIT_CRITICAL()

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

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

Arguments

none

Returned Values

none

Notes/Warnings

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

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

Example

```
void TaskX (void *pdata)
{
#if OS_CRITICAL_METHOD ==3
  OS_CPU_SR spu_sr;
#endif

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

OSFlagAccept()

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

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

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

Arguments

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

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

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

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

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

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

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

Returned Values

The state of the flags in the event flag group

Notes/Warnings

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

Example

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

OS FLAG_GRP *EngineStatus;

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

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

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

OSFlagCreate()

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

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

OSFlagCreate() is used to create an initialise an event flag group.

Arguments

flags	contains the initial value to store in the event flag group.	
err	is a pointer to a variable that is used to hold an error code. The error code can be one of the following:	
	OS_NO_ERR	if the call is successful and the event flag group has been created.
	OS_ERR_CREATE_ISR	if you attempt to create an event flag group from an ISR.
	OS_FLAG_GRP_DELETED	if no more event flag groups are available. You need to increase the value of OS_MAX_FLAGS in OS_CFG.H

Returned Values

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

Notes/Warnings

1. Event flag groups must be created by this function before they can be used by the other services.
2. This function does **not** block if the desired flags are not present.

Example

```
OS_FLAG_GRP * EngineStatus;

void main (void)
{
  INT8U err;

  .
  OSInit();     /* Initialize uC/OS-II */
  .
  .
  EngingeStatus = OSFlagCreate(0x00, &err);
  .
  .
  OSStart();    /* Start Multitasking  */
}
```

OSFlagDel()

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

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

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

Arguments

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

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

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

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

Returned Values

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

Notes/Warnings

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

Example

```
OS_FLAG_GRP *EngineStatusFlags;

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

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

OSFlagPend()

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

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

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

Arguments

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

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

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

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

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

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

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

Returned Values

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

Notes/Warnings

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

Example

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

OS FLAG_GRP *EngineStatus;

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

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

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

OSFlagPost()

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

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

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

Arguments

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

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

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

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

`OS_NO_ERR`	No error
`OS_ERR_PEND_ISR`	You try to call `OSFlagPend` from an ISR, which is not allowed.
`OS_FLAG_INVALID_PGRP`	You passed a `NULL` pointer.
`OS_ERR_EVENT_TYPE`	You are not pointing to an event flag group
`OS_FLAG_INVALID_OPT`	You specify an invalid option.

Returned Values

The new value of the event flags.

Notes/Warnings

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

Example

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

OS_FLAG_GRP *EngineStatusFlags;

void Task (void *pdata)
{
  INT8U     err;

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

OSFlagQuery()

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

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

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

Arguments

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

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

OS_NO_ERR	No error
OS_FLAG_INVALID_PGRP	You passed a NULL pointer.
OS_ERR_EVENT_TYPE	You are not pointing to an event flag group

Returned Values

The state of the flags in the event flag group.

Notes/Warnings

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

Example

```
OS_FLAG_GRP *EngineStatusFlags;

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

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

OSInit()

`void OSInit(void);`

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

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

Arguments

none

Returned Values

none

Notes/Warnings

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

Example

```
void main (void)
{
  .
  .
  OSInit();   /* initialize uC/OS-II */
  .
  .
  OSStart();  /* Start Multitaksing  */
}
```

OSIntEnter()

```
void OSIntEnter(void);
```

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

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

Arguments

none

Returned Values

none

Notes/Warnings

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

Example 1
(Motorola MC68332)

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

```
_OSISRx:
  MOVEM.L   A0-A6/D0-D7,-(A7)   ; Save the registers of the current task

  JSR       _OSIntEnter         ; Notify uC/OS-II of start of ISR
  .
  .
  JSR       _OSIntExit          ; Notify uC/OS-II of end of ISR
  MOVEM.L   (A7)+, A0-A6/D0-D7) ; Restore registers from stack
  RTE                           ; Return from interrupt
```

Example 2
(Motorola MC68332)

```
_OSISRy:
  MOVEM.L   A0-A6/D0-D7,-(A7)   ; Save the registers of the current task
  ADDQ.B    #1,_OSIntNesting    ; OSIntNesting++;
  .
  .
  SUBQ.B    #1,_OSIntNesting    ; OSIntNesting--;
  MOVEM.L   (A7)+, A0-A6/D0-D7) ; Restore registers from stack
  RTE                           ; Return from interrupt
```

OSIntExit()

```
void OSIntExit(void);
```

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

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

Arguments

none

Returned Values

none

Notes/Warnings

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

Example

(Motorola MC68332)

```
_OSISRx:
  MOVEM.L   A0-A6/D0-D7,-(A7)   ; Save the registers of the current task

  JSR       _OSIntEnter         ; Notify uC/OS-II of start of ISR
  .
  .
  JSR       _OSIntExit          ; Notify uC/OS-II of end of ISR
  MOVEM.L   (A7)+, A0-A6/D0-D7) ; Restore registers from stack
  RTE                           ; Return from interrupt
```

OSMboxAccept()

`void *OSMboxAccept(OS_EVENT *pevent);`

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

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

Arguments

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

Returned Values

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

Notes/Warnings

1. Mailboxes must be created before they are used.

Example

```
OS_EVENT *CommMbox;

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

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

OSMboxCreate()

`OS_EVENT *OSMboxCreate(void *msg);`

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

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

Arguments

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

Returned Values

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

Notes/Warnings

1. Mailboxes must be created before they are used.

Example

```
OS_EVENT *CommMbox;

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

OSMboxDel()

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

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

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

Arguments

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

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

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

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

Returned Values

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

Notes/Warnings

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

Example

```
OS_EVENT *DispMbox;

void Task (void *pdata)
{
  INT8U err;

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

OSMboxPend()

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

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

OSMboxPend() is used when a task expects to receive a message. The message is sent to the task either by an ISR or by another task. The message received is a pointer-sized variable, and its use is application specific. If a message is present in the mailbox when OSMboxPend() is called, the message is retrieved, the mailbox is emptied, and the retrieved message is returned to the caller. If no message is present in the mailbox, OSMboxPend() suspends the current task until either a message is received or a user-specified timeout expires. If a message is sent to the mailbox and multiple tasks are waiting for the message, µC/OS-II resumes the highest priority task waiting to run. A pended task that has been suspended with OSTaskSuspend() can receive a message. However, the task remains suspended until it is resumed by calling OSTaskResume().

Arguments

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

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

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

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

Returned Values

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

Notes/Warnings

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

Example

```
OS_EVENT *CommMbox;

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

  pdata = pdata;
  while (1)
  {
    .
    .
    msg = OSMboxPend(CommMbox, 10, &err);
    if (err == OS_NO_ERR)
    {
      /* Code for received message                    */
    }
    else
    {
      /* Code for message not received within timeout */
    }
  }
}
```

OSMboxPost()

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

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

`OSMboxPost()` sends a message to a task through a mailbox. A message is a pointer-sized variable and, its use is application specific. If a message is already in the mailbox, an error code is returned indicating that the mailbox is full. `OSMboxPost()` then immediately returns to its caller, and the message is not placed in the mailbox. If any task is waiting for a message at the mailbox, the highest priority task waiting receives the message. If the task waiting for the message has a higher priority than the task sending the message, the higher priority task is resumed, and the task sending the message is suspended. In other words, a context switch occurs.

Arguments

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

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

Returned Values

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

OS_NO_ERR	if the message is deposited in the mailbox.
OS_MBOX_FULL	if the mailbox already contains a message.
OS_ERR_EVENT_TYPE	if `pevent` is not pointing to a mailbox.
OS_ERR_PEVENT_NULL	if `pevent` is a `NULL` pointer.
OS_ERR_POST_NULL_PTR	if you are attempting to post a `NULL` pointer. By convention a `NULL` pointer is not supposed to point to anything.

Notes/Warnings

1. Mailboxes must be created before they are used.
2. You must never post a `NULL` pointer because this pointer indicates that the mailbox is empty.

Example

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

void Task (void *pdata)
{
  INT8U err;

  pdata = pdata;
  while (1)
  {
    .
    .
    err = OSMboxPost(CommMbox, (void *)&CommRxBuf[0]);
    .
    .
  }
}
```

OSMboxPostOpt()

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

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

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

Arguments

pevent	is a pointer to the mailbox into which the message is deposited. This pointer is returned to your application when the mailbox is created [see OSMboxCreate()].
msg	is the actual message sent to the task(s). msg is a pointer-sized variable and is application specific. You must never post a NULL pointer because this pointer indicates that the mailbox is empty.
opt	specifies whether you want to send the message to the highest priority task waiting at the mailbox (when opt is set to OS_POST_OPT_NONE) or to **all** tasks waiting at the mailbox (when opt is set to OS_POST_OPT_BROADCAST).

Returned Values

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

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

Notes/Warnings

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

Example

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

void Task (void *pdata)
{
  INT8U err;

  pdata = pdata;
  while (1)
  {
    .
    .
    err = OSMboxPostOpt(CommMbox, (void *)&CommRxBuf[0], OS_POST_OPT_BROADCAST);
    .
    .
  }
}
```

OSMboxQuery()

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

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

OSMboxQuery() obtains information about a message mailbox. Your application must allocate an OS_MBOX_DATA data structure, which is used to receive data from the event control block of the message mailbox. OSMboxQuery() allows you to determine whether any tasks are waiting for a message at the mailbox and how many tasks are waiting (by counting the number of 1s in the .OSEventTbl[] field). You can also examine the current contents of the mailbox. Note that the size of .OSEventTbl[] is established by the #define constant OS_EVENT_TBL_SIZE (see uCOS_II.H).

Arguments

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

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

```
void *OSMsg;              /* Copy of the message stored in the mailbox */
INT8U OSEventTbl[OS_EVENT_TBL_SIZE]; /* Copy of the mailbox wait list */
INT8U OSEventGrp;
```

Returned Values

OSMboxQuery() returns one of these error codes:

OS_NO_ERR	if the call is successful.
OS_ERR_PEVENT_NULL	if pevent is a NULL pointer.
OS_ERR_EVENT_TYPE	if pevent is not pointing to a message mailbox.

Notes/Warnings

1. Mailboxes must be created before they are used.

Example

```
OS_EVENT *CommMbox;

void Task (void *pdata)
{
  OS_MBOX_DATA mbox_data;
  INT8U        err;

  pdata = pdata;
  while (1)
  {
    .
    .
    err = OSMboxQuery(CommMbox, &mbox_data);
    if (err == OS_NO_ERR) {
      . /* Mailbox contains a message if mbox_data.OSMsg is not NULL */
    }
    .
    .
  }
}
```

OSMemCreate()

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

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

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

Arguments

`addr`	is the address of the start of a memory area that is used to create fixed-size memory blocks. Memory partitions can be created either using static array or `malloc()` during startup.
`nblks`	contains the number of memory blocks available from the specified partition. You must specify at least two memory blocks per partition.
`blksize`	specifies the size (in bytes) of each memory block within a partition. A memory block must be large enough to hold at least a pointer.
`err`	is a pointer to a variable that holds an error code. `OSMemCreate()` sets `*err` to:

`OS_NO_ERR`	if the memory partition is created successfully.
`OS_MEM_INVALID_ADDR`	if you are specifying an invalid address (i.e., `addr` is a `NULL` pointer).
`OS_MEM_INVALID_PART`	if a free memory partition is not available.
`OS_MEM_INVALID_BLKS`	if you don't specify at least two memory blocks per partition.
`OS_MEM_INVALID_SIZE`	if you don't specify a block size that can contain at least a pointer variable.

Returned Values

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

Notes/Warnings

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

Example

```
OS_MEM *CommMem;
INT8U   CommBuf[16][128];

void main (void)
{
  INT8U        err;

  OSInit();                   /* Initialize uC/OS-II      */
  .
  .
  CommMem = OSMemCreate(&CommBuf[0][0], 16, 128, &err);
  .
  .
  OSStart();                  /* Start Multitasking       */
}
```

OSMemGet()

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

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

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

Arguments

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

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

`OS_NO_ERR`	if a memory block is available and returned to your application.
`OS_MEM_NO_FREE_BLKS`	if the memory partition doesn't contain any more memory blocks to allocate.
`OS_MEM_INVALID_PMEM`	if `pmem` is a `NULL` pointer.

Returned Values

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

Notes/Warnings

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

Example

```
OS_MEM *CommMem;

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

  pdata = pdata;
  while (1)
  {
    msg = OSMemGet(CommMem, &err);
    if (msg != (INT8U *)0)
    {
      .              /* Memory block allocated, use it. */
      .
    }
    .
    .
  }
}
```

OSMemPut()

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

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

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

Arguments

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

pblk is a pointer to the memory block to be returned to the memory partition.a variable that holds an error code. OSMemGet() sets *err to:

Returned Values

OSMemPut() returns one of the following error codes:

OS_NO_ERR	if a memory block is available and returned to your application.
OS_MEM_FULL	if the memory partition can not accept more memory blocks. This code is surely an indication that something is wrong because you are returning more memory blocks than you obtained using OSMemGet().
OS_MEM_INVALID_PMEM	if pmem is a NULL pointer.
OS_MEM_INVALID_PBLK	if pblk is a NULL pointer.

Notes/Warnings

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

Example

```
OS_MEM *CommMem;
INT8U  *CommMsg;

void Task (void *pdata)
{
  INT8U  err;

  pdata = pdata;
  while (1)
  {
    err = OSMemPut(CommMem, (void *)CommMsg);
    if (err == OS_NO_ERR)
    {
      .              /* Memory block released */
      .
    }
    .
    .
  }
}
```

OSMemQuery()

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

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

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

Arguments

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

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

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

Returned Values

OSMemQuery() returns one of the following error codes:

OS_NO_ERR	if a memory block is available and returned to your application.
OS_MEM_INVALID_PMEM	if pmem is a NULL pointer.
OS_MEM_INVALID_PDATA	if pdata is a NULL pointer.

Notes/Warnings

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

Example

```
OS_MEM *CommMem;

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

  pdata = pdata;
  while (1)
  {
    .
    .
    err = OSMemQuery(CommMem, &mem_data);
    .
    .
  }
}
```

OSMutexAccept()

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

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

OSMutexAccept() allows you to check to see if a resource is available. Unlike OSMutexPend(), OSMutexAccept() does not suspend the calling task if the resource is not available. In other words, OSMutexAccept() is non-blocking.

Arguments

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

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

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

Returned Values

If the mutex is available, OSMutexAccept() returns 1. If the mutex is owned by another task, OSMutexAccept() returns 0.

Notes/Warnings

1. Mutexes must be created before they are used.
2. This function **must not** be called by an ISR.
3. If you acquire the mutex through OSMutexAccept(), you **must** call OSMutexPost() to release the mutex when you are done with the resource.

Example

```
OS_EVENT *DispMutex;

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

  pdata = pdata;
  while (1)
  {
    value = OSMutexAccept(DispMutex, &err);
    if (value == 1)
    {
      .                          /* Resource available, process */
      .
    } else {
      .                          /* Resource not available      */
      .
    }
    .
    .
  }
}
```

OSMutexCreate()

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

Chapter	***File***	***Called from***	***Code enabled by***
8	OS_MUTEX.C	Task or startup code	OS_MUTEX_EN

`OSMutexCreate()` is used to create and initialise a mutex. A mutex is used to gain exclusive access to a resource.

Arguments

`prio` is the priority inheritance priority (PIP) that is used when a high priority task attempts to acquire the mutex that is owned by a low priority task. In this case, the priority of the low priority task is *raised* to the PIP until the resource is released.

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

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

Returned Values

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

Notes/Warnings

1. Mutexes must be created before they are used.
2. You **must** make sure that `prio` has a higher priority than **any** of the tasks that use the mutex to access the resource. For example, if three tasks of priority 20, 25, and 30 are going to use the mutex, then `prio` must be a number **lower** than 20. In addition, there **must not** already be a task created at the specified priority.

Example

```
OS_EVENT *DispMutex;

void main (void)
{
  INT8U err;

  .
  .
  OSInit();                          /* Initialize uC/OS-II  */
  .
  .
  DispMutex = OSMutexCreate(20, &err); /* Create Display Mutex */
  .
  .
  OSStart()                          /* Start Multitasking   */
}
```

OSMutexDel()

OS_EVENT *OSMutexDel(OS_EVENT *pevent, INT8U opt, INT8U *err);

Chapter	File	Called from	Code enabled by
8	OS_MUTEX.C	Task	OS_MUTEX_EN && OS_MUTEX_DEL_EN

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

Arguments

pevent — is a pointer to the mutex. This pointer is returned to your application when the mutex is created [see OSMutexCreate()].

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

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

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

Returned Values

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

Notes/Warnings

1. You should use this call with care because other tasks might expect the presence of the mutex.

Example

```
OS_EVENT *DispMutex;

void Task (void *pdata)
{
  INT8U err;

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

OSMutexPend()

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

Chapter	***File***	***Called from***	***Code enabled by***
8	`OS_MUTEX.C`	Task only	`OS_MUTEX_EN`

`OSMutexPend()` is used when a task desires to get exclusive access to a resource. If a task calls `OSMutexPend()` and the mutex is available, then `OSMutexPend()` gives the mutex to the caller and returns to its caller. Note that nothing is actually given to the caller except for the fact that if `err` is set to `OS_NO_ERR`, the caller can assume that it owns the mutex. However, if the mutex is already owned by another task, `OSMutexPend()` places the calling task in the wait list for the mutex. The task thus waits until the task that owns the mutex releases the mutex and thus the resource or until the specified timeout expires. If the mutex is signalled before the timeout expires, µC/OS-II resumes the highest priority task that is waiting for the mutex. Note that if the mutex is owned by a lower priority task, when `OSMutexPend()` raises the priority of the task that owns the mutex to the PIP, as specified when you created the mutex [see `OSMutexCreate()`].

Arguments

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

`timeout` is used to allow the task to resume execution if the mutex is not signalled (i.e., posted to) within the specified number of clock ticks. A timeout value of 0 indicates that the task desires to wait forever for the mutex. The maximum timeout is 65'535 clock ticks. The timeout value is not synchronized with the clock tick. The timeout count starts being decremented on the next clock tick, which could potentially occur immediately.

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

`OS_NO_ERR`	if the call is successful and the mutex is available.
`OS_TIMEOUT`	if the mutex is not available within the specified timeout.
`OS_ERR_EVENT_TYPE`	if `pevent` is not pointing to a mutex.
`OS_ERR_PEVENT_NULL`	if `pevent` is a `NULL` pointer.
`OS_ERR_PEND_ISR`	if you attempt to acquire the mutex from an ISR.

Returned Values

none

Notes/Warnings

1. Mutexes must be created before they are used.
2. You should **not** suspend the task that owns the mutex, have the mutex owner wait on any other µC/OS-II objects (i.e., semaphore, mailbox, or queue), and delay the task that owns the mutex. In other words, your code should hurry up and release the resource as quickly as possible.

Example

```
OS_EVENT *DispMutex;

void Task (void *pdata)
{
  INT8U err;

  pdata = pdata;
  while (1)
  {
    .
    .
    OSMutexPend(DispMutex, 0, &err);
    .                /* The only way this task continues is if */
    .                /* the mutex is available or signaled!    */
  }
}
```

OSMutexPost()

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

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

A mutex is signalled (i.e., released) by calling `OSMutexPost()`. You call this function only if you acquire the mutex by first calling either `OSMutexAccept()` or `OSMutexPend()`. If the priority of the task that owns the mutex has been raised when a higher priority task attempts to acquire the mutex, the original task priority of the task is restored. If one more tasks are waiting for the mutex, the mutex is given to the highest priority task ready to run, and if so, a context switch is done to run the readied task. If no task is waiting, for the mutex, the mutex value is simply set to available (`0xFF`).

Arguments

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

Returned Values

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

`OS_NO_ERR`	if the call is successful and the mutex is released.
`OS_ERR_EVENT_TYPE`	if `pevent` is not pointing to a mutex.
`OS_ERR_PEVENT_NULL`	if `pevent` is a `NULL` pointer.
`OS_ERR_POST_ISR`	if you attempt to call `OSMutexPost()` from an ISR.
`OS_ERR_NOT_MUTEX_OWNER`	if the task posting (i.e., signalling the mutex) doesn't actually own the mutex.

Notes/Warnings

1. Mutexes must be created before they are used.
2. You cannot call this function from an ISR.

Example

```
OS_EVENT *DispMutex;

void Task (void *pdata)
{
  INT8U err;

  pdata = pdata;
  while (1)
  {
    .
    .
    err = OSMutexPost(DispMutex);
    switch (err) {
      case OS_NO_ERR: /* Mutex signaled */
           .
           .
           break;

      case OS_ERR_EVENT_TYPE:
           .
           .
           break;

      case OS_ERR_PEVENT_NULL:
           .
           .
           break;

      case OS_ERR_POST_ISR:
           .
           .
           break;
    }
    .
    .
  }
}
```

OSMutexQuery()

`INT8U OSMutexQuery(OS_EVENT *pevent, OS_MUTEX_DATA *pdata);`

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

OSMutexQuery() is used to obtain run-time information about a mutex. Your application must allocate an OS_MUTEX_DATA data structure that is used to receive data from the event control block of the mutex. OSMutexQuery() allows you to determine whether any task is waiting on the mutex, how many tasks are waiting (by counting the number of 1s) in the .OSEventTbl[] field, obtain the PIP, and determine whether the mutex is available (1) or not (0). Note that the size of .OSEventTbl[] is established by the #define constant OS_EVENT_TBL_SIZE (see uCOS_II.H).

Arguments

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

pdata is a pointer to a data structure of type OS_MUTEX_DATA, which contains the following fields:

```
INT8U OSMutexPIP;  /* The PIP of the mutex                          */
INT8U OSOwnerPrio; /* The priority of the mutex owner               */
INT8U OSValue;     /* The current mutex value, 1 means available, */
                   /* 0 means unavailable                           */
INT8U OSEventGrp;  /* Copy of the mutex wait list                   */
INT8U OSEventTbl[OS_EVENT_TBL_SIZE];
```

Returned Values

OSMutexQuery() returns one of these error codes:

OS_NO_ERR	if the call is successful.
OS_ERR_EVENT_TYPE	if pevent is not pointing to a mutex.
OS_ERR_PEVENT_NULL	if pevent is a NULL pointer.
OS_ERR_QUERY_ISR	if you attempt to call OSMutexQuery() from an ISR.

Notes/Warnings

1. Mutexes must be created before they are used.
2. You cannot call this function from an ISR.

Example

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

```
OS_EVENT *DispMutex;

void Task (void *pdata)
{
  OS_MUTEX_DATA mutex_data;
  INT8U         err;
  INT8U         highest;   /* highest priority task waiting on mutex */
  INT8U         x;
  INT8U         y;

  pdata = pdata;
  while (1)
  {
    .
    .
    err = OSMutexQuery(DispMutex, &mutex_data);
    if (err == OS_NO_ERR) {
      if (mutex_data.OSEventGrp != 0x00) {
        y       = OSUnMapTbl[mutex_data.OSEventGrp];
        x       = OSUnMapTbl[mutex_data.OsEventTbl[y]];
        highest = (y << 3) + x;
        .
        .
      }
    }
    .
    .
  }
}
```

OSQAccept()

`void *OSQAccept(OS_EVENT *pevent);`

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

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

Arguments

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

Returned Values

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

Notes/Warnings

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

Example

```
OS_EVENT *CommQ;

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

  pdata = pdata;
  while (1)
  {
    msg = OSQAccept(CommQ);   /* Check queue for a message */
    if (msg != (void *)0) {
      .                       /* Message received, process */
      .
    } else {
      .                       /* Message not received, do  */
      .                       /* something else            */
    }
    .
    .
  }
}
```

OSQCreate()

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

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

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

Arguments

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

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

Returned Values

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

Notes/Warnings

1. Queues must be created before they are used.

Example

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

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

OSQDel()

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

Chapter	*File*	*Called from*	*Code enabled by*
11	OS_Q.C	Task	OS_Q_EN && OS_Q_DEL_EN

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

Arguments

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

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

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

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

Returned Values

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

Notes/Warnings

1. You should use this call with care because other tasks might expect the presence of the queue.
2. Interrupts are disabled when pended tasks are readied, which means that interrupt latency depends on the number of tasks that are waiting on the queue.

Example

```
OS_EVENT *DispQ;

void Task (void *pdata)
{
  INT8U err;

  pdata = pdata;
  while (1) {
    .
    .
    DispQ = OSQDel(DispQ, OS_DEL_ALWAYS, &err);
    if (DispQ == (OS_EVENT *)0) {
      /* Queue has been deleted */
    }
    .
    .
  }
}
```

OSQFlush()

`INT8U *OSQFlush(OS_EVENT *pevent);`

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

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

Arguments

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

Returned Values

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

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

Notes/Warnings

1. Queues must be created before they are used.

Example

```
OS_EVENT *CommQ;

void main (void)
{
  INT8U err;

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

OSQPend()

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

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

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

Arguments

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

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

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

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

Returned Values

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

Notes/Warnings

1. Queues must be created before they are used.
2. You should not call OSQPend() from an ISR.

Example

```
OS_EVENT *CommQ;

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

  pdata = pdata;
  while (1)
  {
    .
    .
    msg = OSQPend(CommQ, 100, &err);
    if (err == OS_NO_ERR) {
      .               /* Message received within 100 ticks!        */
      .
    } else {
      .               /* Message not received, must have timed out */
      .
    }
    .
    .
  }
}
```

OSQPost()

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

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

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

Arguments

pevent	is a pointer to the queue into which the message is deposited. This pointer is returned to your application when the queue is created [see OSQCreate()].
msg	is the actual message sent to the task. msg is a pointer-sized variable and is application specific. You must never post a NULL pointer.

Returned Values

OSQPost() returns one of these error codes:

OS_NO_ERR	if the message is deposited in the queue.
OS_Q_FULL	if the queue is already full.
OS_ERR_EVENT_TYPE	if pevent is not pointing to a message queue.
OS_ERR_PEVENT_NULL	if pevent is a NULL pointer.
OS_ERR_POST_NULL_PTR	if you are posting a NULL pointer. By convention, a NULL pointer is not supposed to point to anything valid.

Notes/Warnings

1. Queues must be created before they are used.
2. You must never post a NULL pointer.

Example

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

void Task (void *pdata)
{
  INT8U err;

  pdata = pdata;
  while (1)
  {
    .
    .
    err = OSQPost(CommQ, (void *)&CommRxBuf[0]);
    switch (err){
      case OS_NO_ERR:
           /* Message was deposited into queue */
           break;

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

OSQPostFront()

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

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

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

Arguments

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

`msg` is the actual message sent to the task. `msg` is a pointer-sized variable and is application specific. You must never post a `NULL` pointer.

Returned Values

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

`OS_NO_ERR`	if the message is deposited in the queue.
`OS_Q_FULL`	if the queue is already full.
`OS_ERR_EVENT_TYPE`	if `pevent` is not pointing to a message queue.
`OS_ERR_PEVENT_NULL`	if `pevent` is a `NULL` pointer.
`OS_ERR_POST_NULL_PTR`	if you are posting a `NULL` pointer. By convention, a `NULL` pointer is not supposed to point to anything valid.

Notes/Warnings

1. Queues must be created before they are used.
2. You must never post a `NULL` pointer.

Example

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

void Task (void *pdata)
{
  INT8U err;

  pdata = pdata;
  while (1)
  {
    .
    .
    err = OSQPostFront(CommQ, (void *)&CommRxBuf[0]);
    switch (err){
      case OS_NO_ERR:
           /* Message was deposited into queue */
           break;

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

OSQPostOpt()

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

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

OSQPostOpt() is used to send a message to a task through a queue. A message is a pointer-sized variable, and its use is application specific. If the message queue is full, an error code is returned indicating that the queue is full. OSQPostOpt() then immediately returns to its caller, and the message is not placed in the queue. If any task is waiting for a message at the queue, OSQPostOpt() allows you to either post the message to the highest priority task waiting at the queue (opt set to OS_POST_OPT_NONE) or to all tasks waiting at the queue (opt is set to OS_POST_OPT_BROADCAST). In either case, scheduling occurs, and, if any of the tasks that receive the message have a higher priority than the task is posting the message, then the higher priority task is resumed, and the sending task is suspended. In other words, a context switch occurs.
OSQPostOpt() emulates both OSQPost () and OSQPostFront() and allows you to post a message to **multiple** tasks. In other words, it allows the message posted to be broadcasted to **all** tasks waiting on the queue. OSQPostOpt() can actually replace OSQPost() and OSQPostFront() because you specify the mode of operation via an option argument, opt. Doing this allows you to reduce the amount of code space needed by µC/OS-II.

Arguments

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

msg — is the actual message sent to the task(s). msg is a pointer-sized variable and is application specific. You must never post a NULL pointer.

opt — determines the type of POST performed:

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

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

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

Returned Values

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

OS_NO_ERR	if the call is successful and the message has been sent.
OS_Q_FULL	if the queue is already full.
OS_ERR_EVENT_TYPE	if pevent is not pointing to a message queue.
OS_ERR_PEVENT_NULL	if pevent is a NULL pointer.
OS_ERR_POST_NULL_PTR	if you are attempting to post a NULL pointer.

Notes/Warnings

1. Queues must be created before they are used.
2. You must **never** post a `NULL` pointer to a queue.
3. If you need to use this function and want to reduce code space, you can disable code generation of `OSQPost()` (set `OS_Q_POST_EN` to 0 in `OS_CFG.H`) and `OSQPostFront()` (set `OS_Q_POST_FRONT_EN` to 0 in `OS_CFG.H`) because `OSQPostOpt()` can emulate these two functions.
4. The execution time of `OSQPostOpt()` depends on the number of tasks waiting on the queue if you set `opt` to `OS_POST_OPT_BROADCAST`.

Example

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

void Task (void *pdata)
{
  INT8U err;

  pdata = pdata;
  while (1)
  {
    .
    .
    err = OSQPostOpt(CommQ, (void *)&CommRxBuf[0], OS_POST_OPT_BROADCAST);
    .
    .
  }
}
```

OSQQuery()

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

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

OSQQuery() obtains information about a message queue. Your application must allocate an OS_Q_DATA data structure used to receive data from the event control block of the message queue. OSQQuery() allows you to determine whether any tasks are waiting for messages at the queue, how many tasks are waitint (by counting the number of 1s in the .OSEventTbl[] field), how many messages are in the queue, and whar the message queue size is. OSQQuery() also obtains the next message that is returned if the queue is not empty. Note that the size of .OSEventTbl[] is established by the #define constant OS_EVENT_TBL_SIZE (see uCOS_II.H).

Arguments

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

pdata — is a pointer to a data structure of type OS_Q_DATA , which contains the following fields:

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

opt — determines the type of POST performed:

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

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

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

Returned Values

OSQQuery() returns one of these error codes:

OS_NO_ERR	if the call is successful.
OS_ERR_EVENT_TYPE	if pevent is not pointing to a message queue.
OS_ERR_PEVENT_NULL	if pevent is a NULL pointer.

Notes/Warnings

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

Example

```
OS_EVENT *CommQ;

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

  pdata = pdata;
  while (1)
  {
    .
    .
    err = OSQQuery(CommQ, &qdata);
    if (err == OS_NO_ERR) {
      . /* 'qdata' can be examined! */
    }
    .
    .
  }
}
```

OSSchedLock()

`void OSSchedLock(void);`

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

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

Arguments

none

Returned Values

none

Notes/Warnings

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

Example

```
void Task (void *pdata)
{
  pdata = pdata;
  while (1)
  {
    .
    OSSchedLock();    /* Prevent other tasks to run         */
    .
    .                 /* Code protected from context switch */
    .
    OSSchedUnlock();  /* Enable other tasks to run          */
    .
  }
}
```

OSSchedUnlock()

```
void OSSchedUnlock(void);
```

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

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

Arguments

none

Returned Values

none

Notes/Warnings

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

Example

```
void Task (void *pdata)
{
  pdata = pdata;
  while (1)
  {
    .
    OSSchedLock();    /* Prevent other tasks to run         */
    .
    .                 /* Code protected from context switch */
    .
    OSSchedUnlock();  /* Enable other tasks to run          */
    .
  }
}
```

OSSemAccept()

`INT16U OSSemAccept(OS_EVENT *pevent);`

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

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

Arguments

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

Returned Values

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

Notes/Warnings

1. Semaphores must be created before they are used.

Example

```
OS_EVENT *DispSem;

void Task (void *pdata)
{
  INT16U value;

  pdata = pdata;
  while (1)
  {
    value = OSSemAccept(DispSem); /* Check resource availability */
    if (value > 0)
    {
      .
      .
    }
    .
    .
  }
}
```

OSSemCreate()

`OS_EVENT *OSSemCreate(INT16U value);`

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

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

Arguments

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

Returned Values

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

Notes/Warnings

1. Semaphores must be created before they are used.

Example

```
OS_EVENT *DispSem;

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

OSSemDel()

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

Chapter	File	Called from	Code enabled by
7	OS_SEM.C	Task	OS_SEM_EN && OS_SEM_DEL_EN

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

Arguments

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

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

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

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

Returned Values

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

Notes/Warnings

1. You should use this call with care because other tasks might expect the presence of the semaphore.
2. Interrupts are disabled when pended tasks are readied, which means that interrupts latency depends on the number of tasks that are waiting on the semaphore.

Example

```
OS_EVENT *DispSem;

void Task (void *pdata) {
  INT8U err;

  pdata = pdata;
  while (1) {
    .
    .
    DispSem = OSSemDel(DispSem, OS_DEL_ALWAYS, &err);
    if (DispSem == (OS_EVENT *)0)     {
      /* Semapore has been deleted */
    }
    .
    .
  }
}
```

OSSemPend()

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

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

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

Arguments

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

`timeout` allows the task to resume execution if a semaphore is not signaled within the specified number of clock ticks. A timeout value of 0 indicates that the task waits forever for the semaphore. The maximum timeout is 65'535 clock ticks. The timeout value is not synchronized with the clock tick. The timeout count begins decrementing on the next clock tick, which could potentially occur immediately.

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

OS_NO_ERR	if the semaphore is available.
OS_TIMEOUT	if the semaphore is not signalled within the specified timeout.
OS_ERR_EVENT_TYPE	if pevent is not pointing to a semaphore.
OS_ERR_PEND_ISR	if you called this function from an ISR and µC/OS-II has to suspend it. You should not call OSSemPend() from an ISR. µC/OS-II checks for this situation.
OS_ERR_PEVENT_NULL	if pevent is a NULL pointer.

Returned Values

none

Notes/Warnings

1. Semaphores must be created before they are used.

Example

```
OS_EVENT *DispSem;

void Task (void *pdata)
{
  INT8U err;

  pdata = pdata;
  while (1)
  {
    .
    .
    OSSemPend(DispSem, 0, &err);
    .   /* The only way this task continues is if... */
    .   /* ... the semaphore is signaled!            */
  }
}
```

OSSemPost()

`INT8U OSSemPost(OS_EVENT *pevent);`

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

A semaphore is signalled 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 signalled, `OSSemPost()` removes the highest priority task pending for the semaphore from the waiting list and makes this task ready to run. The scheduler is then called to determine if the awakened task is now the highest priority task ready to run.

Arguments

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

Returned Values

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

`OS_NO_ERR`	if the semaphore is signalled successfully.
`OS_SEM_OVF`	if the semaphore count overflows.
`OS_ERR_EVENT_TYPE`	if `pevent` is not pointing to a semaphore.
`OS_ERR_PEVENT_NULL`	if `pevent` is a `NULL` pointer.

Notes/Warnings

1. Semaphores must be created before they are used.

Example

```
OS_EVENT *DispSem;

void Task (void *pdata)
{
  INT8U err;

  pdata = pdata;
  while (1)
  {
    .
    .
    err = OSSemPost(DispSem);
    switch (err)
    {
      case OS_NO_ERR:
           /* semaphore signalled */
           break;

      case OS_SEM_OVF:
           /* semaphore has overflowed */
           break;
      .
      .
    }
    .
    .
  }
}
```

OSSemQuery()

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

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

OSSemQuery() obtains information about a semaphore. Your application must allocate an OS_SEM_DATA data structure used to receive data from the event control block of the semaphore. OSSemQuery() allows you to determine whether any tasks are waiting on the semaphore and how many tasks are waiting (by counting the number of 1s in the .OSEventTbl[] field) and obtains the semaphore count. Note that the size of .OSEventTbl[] is established by the #define constant OS_EVENT_TBL_SIZE (see uCOS_II.H).

Arguments

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

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

```
INT16U OSCnt;                          /* Current semaphore count */
INT8U  OSEventTbl[OS_EVENT_TBL_SIZE]; /* Semaphore wait list     */
INT8U  OSEventGrp;
```

Returned Values

OSSemQuery() returns one of these error codes:

OS_NO_ERR	if the call is successful.
OS_ERR_EVENT_TYPE	if pevent is not pointing to a semaphore.
OS_ERR_PEVENT_NULL	if pevent is a NULL pointer.

Notes/Warnings

1. Semaphores must be created before they are used.

Example

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

```
OS_EVENT *DispSem;

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

  pdata = pdata;
  while (1)
  {
    .
    .
    err = OSSemQuery(DispSem, &sem_data);
    if (err)
    {
      if (sem_data.OSEventGrp != 0x00)
      {
        y       = OSUnMapTbl[sem_data.OSEventGrp];
        x       = OSUnMapTbl[sem_data.OSEventTbl[y]];
        highest = (y << 3) + x;
        .
        .
      }
    }
    .
    .
  }
}
```

OSStart()

```
void OSStart(void);
```

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

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

Arguments

none

Returned Values

none

Notes/Warnings

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

Example

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

OSStatInit()

`void OSStatInit(void);`

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

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

Arguments

none

Returned Values

none

Notes/Warnings

none

Example

```
void FirstAndOnlyTask (void *pdata)
{
  .
  .
  OSStatInit();       /* Compute CPU capacity with no task running */
  .
  OSTaskCreate(...);  /* Create the other tasks                    */
  OSTaskCreate(...);
  .
  while (1)
  {
    .
    .
  }
}
```

OSTaskChangePrio()

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

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

`OSTaskChangePrio()` changes the priority of a task.

Arguments

`oldprio` is the priority number of the task to change.
`newprio` is the priority number of the task to change.

Returned Values

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

`OS_NO_ERR`	if the task's priority is changed.
`OS_PRIO_INVALID`	if either the old priority or the new priority is equal to or exceeds `OS_LOWEST_PRIO`.
`OS_PRIO_EXIST`	if `newprio` already exists.
`OS_PRIO_ERR`	if no task with the specified old priority (i.e., the task specified by `oldprio` does not exist).

Notes/Warnings

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

Example

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

  while (1)
  {
    .
    .
    err = OSTaskChangePrio(10,15);
    .
    .
  }
}
```

OSTaskCreate()

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

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

`OSTaskCreate()` creates a task so it can be managed by µC/OS-II. Tasks can be created either prior to the start of multitasking or by a running task. A task cannot be created by an ISR. A task must be written as an infinite loop, as shown below, and must not return.
`OSTaskCreate()` is used for backward compatibility with µC/OS and when the added features of `OSTaskCreateExt()` are not needed.
Depending on how the stack frame is built, your task has interrupts either enabled or disabled. You need to check with the processor-specific code for details.

```
void Task (void *pdata)
{
  .              /* Do something with 'pdata'           */
  while (1)      /* Task body, always an infinite loop. */
  {
    .
    .
    /* Must call one of the following services:        */
    /*  OSMboxPend()                                   */
    /*  OSFlagPend()                                   */
    /*  OSMutexPend()                                  */
    /*  OSQPend()                                      */
    /*  OSSemPend()                                    */
    /*  OSTimeDly()                                    */
    /*  OSTimeDlyHMSM()                                */
    /*  OSTaskSuspend()  (Suspend self)                */
    /*  OSTaskDel()      (Delete self)                 */
    .
    .
  }
}
```

Arguments

`task`	is a pointer to the task's code.
`pdata`	is a pointer to an optional data area used to pass parameters to the task when it is created. Where the task is concerned, it thinks it is invoked and passes the argument `pdata`. `pdata` can be used to pass arguments to the task created. For example, you can create y generic task that handles an asynchronous serial port. `pdata` can be used to pass this task information about the serial port it has to manage: the port address, the baud rate, the number of bits, the parity, and more.
`ptos`	is a pointer to the task's top-of-stack. The stack is used to store local variables, function parameters, return address, and CPU register during an interrupt. The size of the stack is determined by the task's requirements and the anticipated interrupt nesting. Determining the size of the stack involves knowing how many bytes are required for storage of local variables for the task itself and all nested functions, as well as requirements for interrupts (accounting for nesting). If the configuration constant `OS_STK_GROWTH` is set to 1, the stack is assumed to grow downward (i.e., from high to low memory). `ptos` thus needs to point to the highest *valid* memory location on the stack. If `OS_STK_GROWTH` is set to 0, the stack is assumed to grow in the opposite direction (i.e., from low to high memory).
`prio`	is the task priority. A unique priority number must be assigned to each task and the lower the number, the higher the priority (i.e., the task importance).

Returned Values

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

`OS_NO_ERR`	if the function is successful.
`OS_PRIO_INVALID`	if `prio` is higher than `OS_LOWEST_PRIO`.
`OS_PRIO_EXIST`	if the requested priority already exists.
`OS_NO_MORE_TCB`	if µC/OS-II doesn't have any more `OS_TCB`s to assign.

Notes/Warnings

1. The stack for the task must be declared with the `OS_STK` type.
2. A task must always invoke one of the services provided by µC/OS-II to wait for time to expire, suspend the task, or wait for an event to occur (wait on a mailbox, queue, or semaphore). This allows other tasks to gain control of the CPU.
3. You should not use task priorities `0`, `1`, `2`, `3`, `OS_LOWEST_PRIO-3`, `OS_LOWEST_PRIO-2`, `OS_LOWEST_PRIO-1`, and `OS_LOWEST_PRIO` because they are reserved for use by µC/OS-II. This leaves you with up to 56 application tasks.

Example 1

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

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

```
OS_STK Task1Stk[1024];

void main (void)
{
  INT8U err;

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

void Task1 (void *pdata)
{
  pdata = pdata;        /* prevent compiler warning */

  while (1)
  {
    .                   /* Task code                */
    .
  }
}
```

Example 2

You can create a generic task that can be instantiated more than once. For example, a task that handles a serial port could be passed the address of a data structure that characterizes the specific port (i.e., port address and baud rate). Note that each task has its own stack space and its own (different) priority. In this example, I arbitrarily decided that COM 1 is the most important port of the two.

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

OS_STK    *Comm2Stk[1024];
COMM_DATA Comm2Data;          /* Data structure containing COMM port   */
                              /* specific data for channel 1           */

void main (void)
{
  INT8U err;

  .
  OSInit();                   /* initialize uC/OS-II                   */
  .
                              /* Create task to manage COM 1           */
  OSTaskCreate(CommTask,
               (void *)&Comm1Data,
               &Comm1Stk[1023],
               25);
                              /* Create task to manage COM 2           */
  OSTaskCreate(CommTask,
               (void *)&Comm1Data,
               &Comm2Stk[1023],
               26);
  .
  OSStart();                  /* Start multitasking                    */
}

void CommTask (void *pdata) /* Generic communication task              */
{
  pdata = pdata;

  while (1)
  {
    .                         /* Task code                             */
    .
  }
}
```

OSTaskCreateExt()

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

Chapter	File	Called from	Code enabled by
4	OS_TASK.C	Task or startup code	N/A

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

```
void Task (void *pdata)
{
  .             /* Do something with 'pdata'           */
  while (1)     /* Task body, always an infinite loop. */
  {
    .
    .
    /* Must call one of the following services:        */
    /*  OSMboxPend()                                   */
    /*  OSFlagPend()                                   */
    /*  OSMutexPend()                                  */
    /*  OSQPend()                                      */
    /*  OSSemPend()                                    */
    /*  OSTimeDly()                                    */
    /*  OSTimeDlyHMSM()                                */
    /*  OSTaskSuspend()  (Suspend self)                */
    /*  OSTaskDel()      (Delete self)                 */
    .
    .
  }
}
```

Arguments

`task`	is a pointer to the task's code.
`pdata`	is a pointer to an optional data area used to pass parameters to the task when it is created. Where the task is concerned, it thinks it is invoked and passes the argument `pdata`. `pdata` can be used to pass arguments to the task created. For example, you can create y generic task that handles an asynchronous serial port. `pdata` can be used to pass this task information about the serial port it has to manage: the port address, the baud rate, the number of bits, the parity, and more.
`ptos`	is a pointer to the task's top-of-stack. The stack is used to store local variables, function parameters, return address, and CPU register during an interrupt. The size of the stack is determined by the task's requirements and the anticipated interrupt nesting. Determining the size of the stack involves knowing how many bytes are required for storage of local variables for the task itself and all nested functions, as well as requirements for interrupts (accounting for nesting). If the configuration constant `OS_STK_GROWTH` is set to 1, the stack is assumed to grow downward (i.e., from high to low memory). `ptos` thus needs to point to the highest *valid* memory location on the stack. If `OS_STK_GROWTH` is set to 0, the stack is assumed to grow in the opposite direction (i.e., from low to high memory).
`prio`	is the task priority. A unique priority number must be assigned to each task and the lower the number, the higher the priority (i.e., the task importance).
`id`	is the task's ID number. At this time, the ID is not currently used in any other function and has simply been added in `OSTaskCreateExt()` for future expansion. You should set `id` to the same value as the task's priority.
`pbos`	is a pointer to the task's bottom-of-stack. If the configuration constant `OS_STK_GROWTH` is set to 1, the stack is assumed to grow downward (i.e., from high to low memory); thus, `pbos` must point to the lowest valid stack location. If `OS_STK_GROWTH` is set to 0, the stack is assumed to grow in the opposite direction (i.e., from low to high memory); thus, `pbos` must point to the highest valid stack location. `pbos` is used by the stack-checking function `OSTaskStkChk()`.
`stk_size`	specifies the size of the task's stack in number of elements. If `OS_STK` is set to `INT8U`, then `stk_size` corresponds to the number of bytes available on the stack. If `OS_STK` is set to `INT16U`, then `stk_size` contains the number uf 16-bit entries available on the stack. Finally, if `OS_STK` is set to `INT32U`, then `stk_size` contains the number of 32-bit entries available on the stack.
`pext`	is a pointer to a user-supplied memory location (typically a data structure) used as a TCB extension. For example, this user memory can hold the contents of floating-point registers during a context switch, the time each task takes to execute, the number of times the task is switched in, an so on.
`opt`	contains task-specific options. The lower 8 bits are reserved by µC/OS-II, but you can use the upper 8 bits for application-specific options. Each option consists of one or more bits. The option is selected when the bit(s) is set. The current version of µC/OS-II supports the following options:

`OS_TASK_OPT_STK_CHK`	specifies whether stack checking is allowed for the task.
`OS_TASK_OPT_STK_CLR`	specifies whether stack needs to be cleared.
`OS_TASK_OPT_SAVE_FP`	specifies whether floating-point registers are saved. This option is only valid if your processor has floating-point hardware and the processor-specific code saves the floating-point registers.

Refer to `uCOS_II.H` for other options.

Returned Values

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

`OS_NO_ERR`	if the function is successful.
`OS_PRIO_INVALID`	if `prio` is higher than `OS_LOWEST_PRIO`.
`OS_PRIO_EXIST`	if the requested priority already exists.
`OS_NO_MORE_TCB`	if µC/OS-II doesn't have any more `OS_TCB`s to assign.

Notes/Warnings

4. The stack for the task must be declared with the `OS_STK` type.
5. A task must always invoke one of the services provided by µC/OS-II to wait for time to expire, suspend the task, or wait for an event to occur (wait on a mailbox, queue, or semaphore). This allows other tasks to gain control of the CPU.
6. You should not use task priorities `0`, `1`, `2`, `3`, `OS_LOWEST_PRIO-3`, `OS_LOWEST_PRIO-2`, `OS_LOWEST_PRIO-1`, and `OS_LOWEST_PRIO` because they are reserved for use by µC/OS-II. This leaves you with up to 56 application tasks.

Example 1

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

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

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

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

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

OS_STK            TaskStk[1024];
OS_TASK_USER_DATA TaskUserData;

void main (void){
  INT8U err;

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

void Task (void *pdata){
  pdata = pdata;                    /* Avoid compiler warning         */

  while (1){
    .                               /* Task code                      */
    .
  }
}
```

Example 2

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

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

```
OS_STK *TaskStk[1024];

void main (void)
{
  INT8U err;

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

void Task (void *pdata)
{
  pdata = pdata;                   /* Avoid compiler warning        */

  while (1)
  {
    .                              /* Task code                     */
    .
  }
}
```

OSTaskDel()

`INT8U OSTaskDel(INT8U prio);`

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

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

Arguments

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

Returned Values

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

`OS_NO_ERR`	if the task doesn't delete itself.
`OS_TASK_DEL_IDLE`	if you try to delete the idle task, which is of course is not allowed.
`OS_TASK_DEL_ERR`	if the task to delete does not exist.
`OS_PRIO_INVALID`	if you specify a task priority higher than `OS_LOWEST_PRIO`.
`OS_TASK_DEL_ISR`	if you try to delete a task from an ISR.

Notes/Warnings

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

Example

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

  pdata = pdata;            /* Avoid compiler warning       */
  while (1)
  {
    .
    .
    err = OSTaskDel(10);    /* Delete task with priority 10 */
    if (err == OS_NO_ERR)
    {
      .                     /* Task was deleted             */
      .
    }
    .
    .
  }
}
```

OSTaskDelReq()

```
INT8U OSTaskDelReq(INT8U prio);
```

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

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

Arguments

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

Returned Values

OSTaskDelReq() returns one of these error codes:

OS_NO_ERR	if the task deletion has been registered.
OS_TASK_NOT_EXIST	if the task does not exist. The requesting task can monitor this return code to see if the task is actually deleted.
OS_TASK_DEL_IDLE	if you ask to delete the idle task (which is not allowed).
OS_PRIO_INVALID	if you specify a task priority higher than OS_LOWEST_PRIO or do not specify OS_PRIO_SELF.
OS_TASK_DEL_REQ	if a task (possibly anther task) requests that the running task be deleted.

Notes/Warnings

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

Example

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

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

void TaskToBeDeleted (void *pdata)  /* My priority is 10                */
{
  .
  .
  pdata = pdata;            /* Avoid compiler warning       */
  while (1)
  {
    OSTimeDly(1);
    if (OSTaskDelReq(OS_PRIO_SELF) == OS_TASK_DEL_REQ)
    {
      /* Release any owned resources;                                    */
      /* De-allocate any dynamic memory;                                 */
      OSTaskDel(OS_PRIO_SELF);
    }
  }
}
```

OSTaskQuery()

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

Chapter	File	Called from	Code enabled by
4	OS_TASK.C	Task or ISR	N/A

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

Arguments

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

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

Returned Values

OSTaskQuery() returns one of these error codes:

OS_NO_ERR	if the call is successful.
OS_PRIO_ERR	if you try to obtain information from an invalid task.
OS_PRIO_INVALID	if you specify priority higher than OS_LOWEST_PRIO.

Notes/Warnings

1. The fields in the task control block depend on the following configuration options (see OS_CFG.H):

- OS_TASK_CREATE_EN
- OS_Q_EN
- OS_FLAG_EN
- OS_MBOX_EN
- OS_SEM_EN
- OS_TASK_DEL_EN

Example

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

  pdata = pdata;
  while (1) {
    .
    .
    err = OSTaskQuery(OS_PRIO_SELF, &task_data);
    if (err == OS_NO_ERR) {
      pext   = task_data.OSTCBExtPtr; /* Get TCB extension pointer */
      status = task_data.OSTCBStat;   /* Get task status           */
      .
      .
    }
    .
    .
  }
}
```

OSTaskResume()

`INT8U OSTaskResume(INT8U prio);`

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

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

Arguments

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

Returned Values

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

OS_NO_ERR	if the call is successful.
OS_TASK_RESUME_PRIO	if the task you are attempting to resume does not exist.
OS_TASK_NOT_SUSPENDED	if the task to resume has not been suspended.
OS_PRIO_INVALID	if `prio` is higher or equal to `OS_LOWEST_PRIO`.

Notes/Warnings

none

Example

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

  pdata = pdata;
  while (1)
  {
    .
    .
    err = OSTaskResume(10);     /* Resume task with priority 10 */
    if (err == OS_NO_ERR)
    {
      .                         /* Task was resumed             */
      .
    }
    .
    .
  }
}
```

OSTaskStkChk()

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

Chapter	File	Called from	Code enabled by
4	OS_TASK.C	Task code	OS_TASK_CREATE_EXT_EN

`OSTaskStkChk()` determines a task's stack statistics. Specifically, it computes the amount of free stack space, as well as the amount of stack space used by the specified task. This function requires that the task be created with `OSTaskCreateExt()` and that you specify `OS_TASK_OPT_STK_CHK` in the `opt` argument.
Stack sizing is done by walking from the bottom of the stack an counting the number of 0 entries on the stack until a nonzero value is found. Of course, this assumes that the stack is cleared when the task is created. For that purpose, you need to set `OS_TASK_OPT_STK_CLR` to 1 as an option when you create the task. You should set `OS_TASK_OPT_STK_CLR` to 0 if your startup code clears all RAM and you never delete your tasks. This reduces the execution time of `OSTaskCreateExt()`.

Arguments

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

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

```
INT32U OSFree;    /* Number of bytes free on the stack */
INT32U OSUSed;    /* Number of bytes used on the stack */
```

Returned Values

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

`OS_NO_ERR`	if you specify valid arguments and the call is successful.
`OS_PRIO_INVALID`	if you specify a task priority higher than `OS_LOWEST_PRIO` or you don't specify `OS_PRIO_SELF`.
`OS_TASK_NOT_EXIST`	if the specified task does not exist.
`OS_TASK_OPT_ERR`	if you do not specify `OS_TASK_OPT_STK_CHK` when the task was created by `OSTaskCreateExt()` or if you create the task by using `OSTaskCreate()`.

Notes/Warnings

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

Example

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

  pdata = pdata;
  while (1)
  {
    .
    .
    err = OSTaskStkChk(10, &stk_data);  /* Resume task with priority 10 */
    if (err == OS_NO_ERR)
    {
      stk_size = stk_data.OSFree + stk_data.OSUsed;
    }
    .
    .
  }
}
```

OSTaskSuspend()

`INT8U OSTaskSuspend(INT8U prio);`

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

`OSTaskSuspend()` suspends (or blocks) execution of a task unconditionally. The calling task can be suspended by specifying its own priority number or `OS_PRIO_SELF` if the task doesn't know its own priority number. In this case, another task needs to resume the suspended task. If the current task is suspended, rescheduling occurs, and µC/OS-II runs the next highest priority task ready to run. The only way to resume a suspended task is to call `OSTaskResume()`.
Task suspension is additive, which means that if the task being suspended is delayed until *n* ticks expire, the task is resumed only when both the time expires and the suspension is removed. Also, if the suspended task is waiting for a semaphore and the semaphore is signalled, 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 suspension is removed.

Arguments

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

Returned Values

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

`OS_NO_ERR`	if the call is successful.
`OS_TASK_SUSPEND_IDLE`	if you attempt to suspend the µC/OS-II idle task, which is not allowed.
`OS_PRIO_INVALID`	if you specify a task priority higher than `OS_LOWEST_PRIO` or you don't specify `OS_PRIO_SELF`.
`OS_TASK_SUSPEND_PRIO`	if the task you are attempting to suspend does not exist.

Notes/Warnings

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

Example

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

  pdata = pdata;
  while (1)
  {
    .
    .
    err = OSTaskSuspend(OS_PRIO_SELF);  /* suspend current task */
    .                /* Execution continues when ANOTHER task.. */
    .                /* .. explicitly resumes this task         */
  }
}
```

OSTimeDly()

`void OSTimeDly(INT16U ticks);`

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

`OSTimeDly()` allows a task to delay itself for an integral number of clock ticks. Rescheduling always occurs when the number of clock ticks is greater than zero. Valid delays range from one to 65'535 ticks. A delay of 0 means that the task is not delayed, and `OSTimeDly()` returns immediately to the caller. The actual delay time depends on the tick rate (see `OS_TICKS_PER_SEC` in the configuration file `OS_CFG.H`).

Arguments

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

Returned Values

none

Notes/Warnings

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

Example

```
void Task (void *pdata)
{

  pdata = pdata;
  while (1)
  {
    .
    .
    OSTimeDly(10);  /* delay task for 10 clock ticks */
    .
    .
  }
}
```

OSTimeDlyHMSM()

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

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

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

Arguments

`hours`	is the number of hours the task is delayed. The valid range of values is 0 to 255.
`minutes`	is the number of minutes the task is delayed. The valid range of values is 0 to 59.
`seconds`	is the number of seconds the task is delayed. The valid range of values is 0 to 59.
`milli`	is the number of milliseconds the task is delayed. The valid range of values is 0 to 999. Note that the resolution of this argument is in multiples of the tick rate. For instance, if the tick rate is set to 100Hz, a delay of 4ms results in no delay. The delay is rounded to the nearest tick. Thus, a delay of 15ms actually results in a delay of 20ms.

Returned Values

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

`OS_NO_ERR`	if you specify valid arguments and the call is successful.
`OS_TIME_INVALID_MINUTES`	if the minutes argument is greater than 59 .
`OS_TIME_INVALID_SECONDS`	if the seconds argument is greater than 59.
`OS_TIME_INVALID_MILLI`	if the milliseconds argument is greater than 999.
`OS_TIME_ZERO_DLY`	if all four arguments are 0.

Notes/Warnings

1. Note that `OSTimeDlyHMSM(0,0,0,0)` (i.e., hours minutes, seconds, milliseconds) results in no delay, and the function returns to the caller. Also, if the total delay time is longer than 65'535 clock ticks, you cannot abort the delay and resume the task by calling `OSTimeDlyResume()`.

Example

```
void Task (void *pdata)
{

  pdata = pdata;
  while (1)
  {
    .
    .
    OSTimeDlyHMSM(0,0,1,0);  /* delay task for 1 second */
    .
    .
  }
}
```

OSTimeDlyResume()

`INT8U OSTimeDlyResume(INT8U prio);`

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

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

Arguments

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

Returned Values

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

`OS_NO_ERR`	if the call is successful.
`OS_PRIO_INVALID`	if you specify a task priority greater than `OS_LOWEST_PRIO`.
`OS_TIME_NOT_DLY`	if the task is not waiting for time to expire.
`OS_TASK_NOT_EXIST`	if the task has not been created.

Notes/Warnings

1. Note that you must not call this function to resume a task that is waiting for an event with timeout. This situation makes the task look like a timeout occurred (unless you desire this effect).
2. You cannot resume a task that has called `OSTimeDlyHMSM()` with a combined time that exceeds 65'535 clock ticks. In other words, if the clock tick runs at 100Hz, you cannot resume a delayed task that called `OSTimeDlyHMSM(0, 10, 55, 350)` or higher.
   `(10 minutes * 60 + (55 + 0.35)seconds) * 100 ticks/second`

Example

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

  pdata = pdata;
  while (1)
  {
    .
    .
    err = OSTimeDlyResume(10);  /* resume task with priority 10 */
    if (err == OS_NO_ERR)
    {
      .                         /* task was resumed             */
      .
    }
  }
}
```

OSTimeGet()

`INT32U OSTimeGet(void);`

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

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

Arguments

none

Returned Values

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

Notes/Warnings

none

Example

```
void Task (void *pdata)
{
  INT16U clk;

  pdata = pdata;
  while (1)
  {
    .
    .
    clk = OSTimeGet(); /* get current value of system clock */
    .
    .
  }
}
```

OSTimeSet()

```
void OSTimeSet(INT32U ticks);
```

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

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

Arguments

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

Returned Values

none

Notes/Warnings

none

Example

```
void Task (void *pdata)
{
  pdata = pdata;
  while (1)
  {
    .
    .
    OSTimeSet(0); /* reset the system clock */
    .
    .
  }
}
```

OSTimeTick()

```
void OSTimeTick(void);
```

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

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

Arguments

none

Returned Values

none

Notes/Warnings

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

Example

(Motorola MC68332)

```
_OSTickISR:

    ADDQ.B     #1,_OSIntNesting        ; OSIntNesting++;

    MOVEM.L    A0-A6/D0-D7,-(A7)       ; Save the registers of the current task

    MOVEQ      #1,D0
    CMP.B      _OSIntNesting,D0        ; if(OSIntNesting == 1){                 */

    BNE        _OSTickISR1

    MOVE.L     (_OSTCBCur),A1          ;     OSTCBCur->OSTCBStkPtr = SP;
    MOVE.L     A7,(A1)                 ; }

    JSR        _OSTimeTick             ; Call uC/OS-II's tick updating function

    ORI        #$0700,SR               ; Disable interrupts

    SUBQ.B     #1,_OSIntNesting        ; OSIntNesting--;

    MOVEM.L    (A7)+,A0-A6/D0-D7       ; Restore the CPU registers

    RTE                                ; Return to task or nested ISR
```

OSVersion()

`INT16U OSVersion(void);`

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

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

Arguments

none

Returned Values

The version is returned as *x.yy* multiplied by 100. For example, v2.52 is returned as 252.

Notes/Warnings

none

Example

(Motorola MC68332)

```
void Task (void *pdata)
{
  INT16U os_version

  pdata = pdata;
  while (1)
  {
    .
    .
    os_version = OSVersion(); /* obtain uC/OS-II's version */
    .
    .
  }
}
```