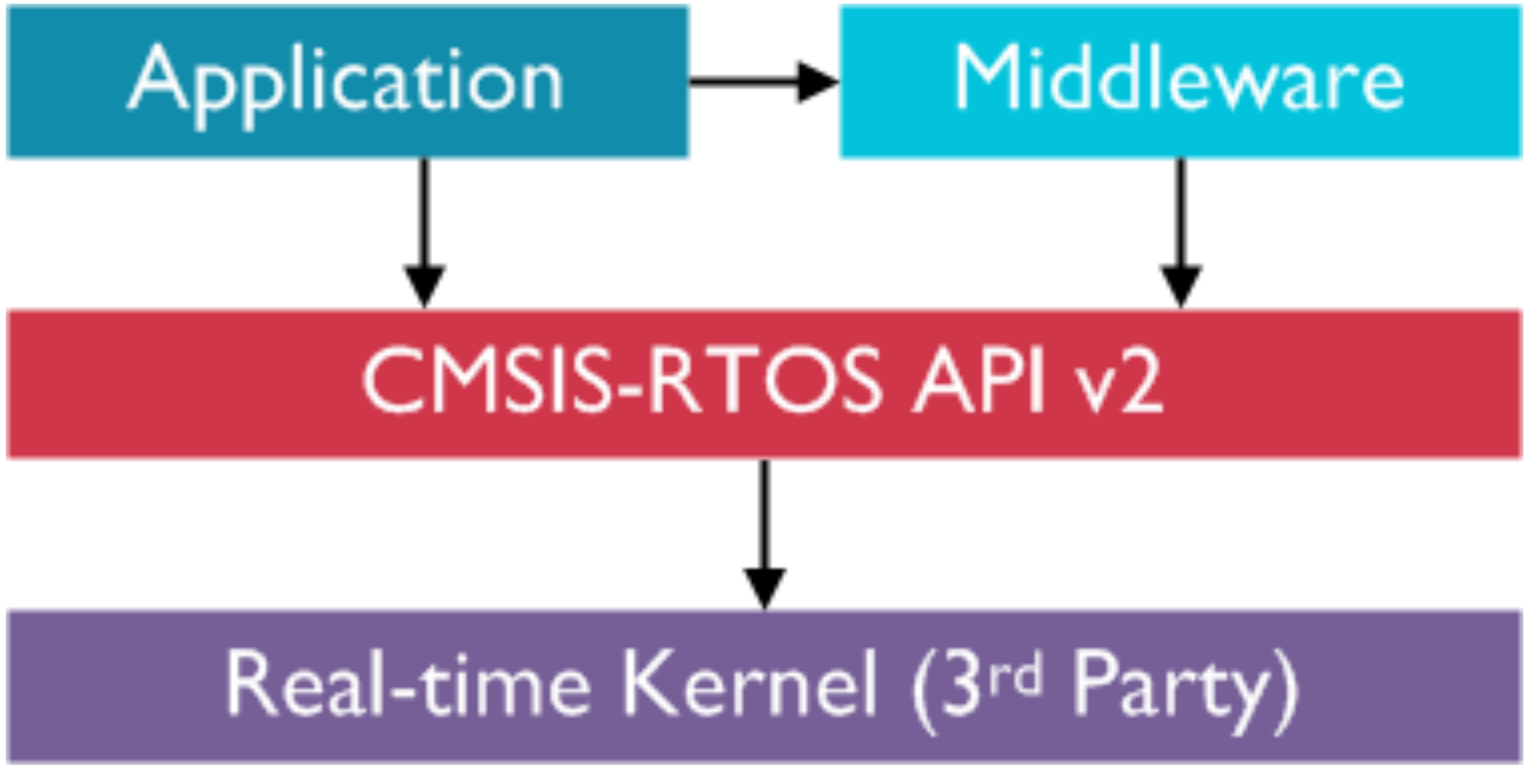
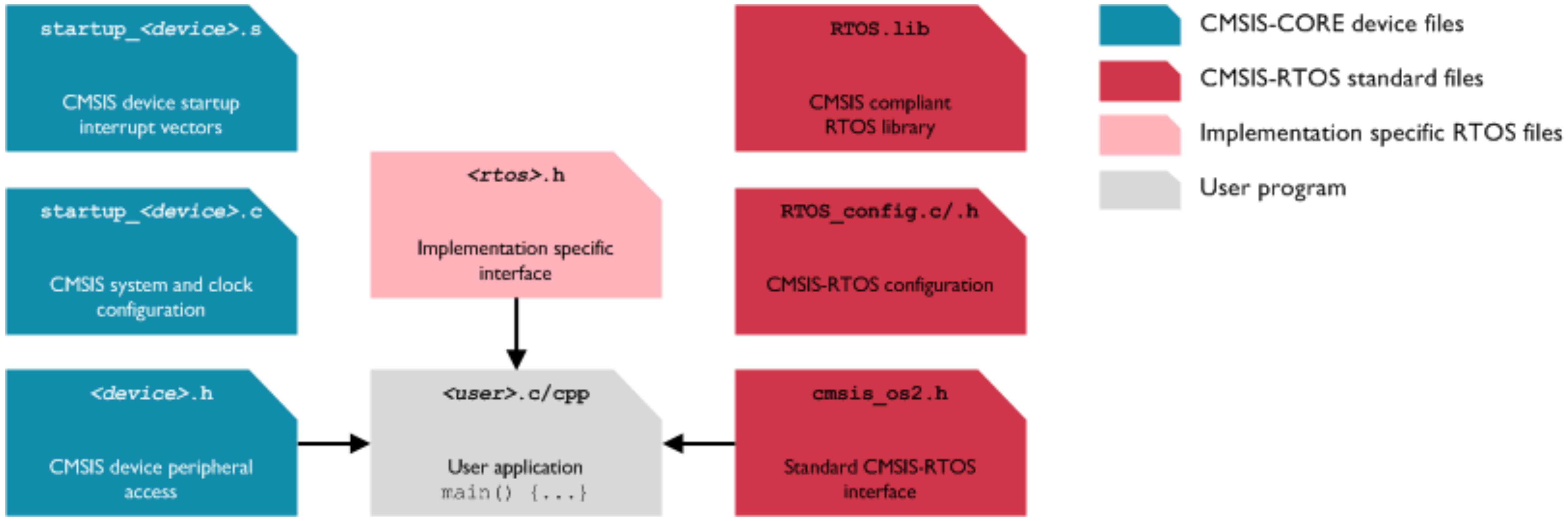
CMSIS-RTOS2

Dokumentacija:
https://www.keil.com/pack/doc/CMSIS/RTOS2/html/group__CMSIS__RTOS.html

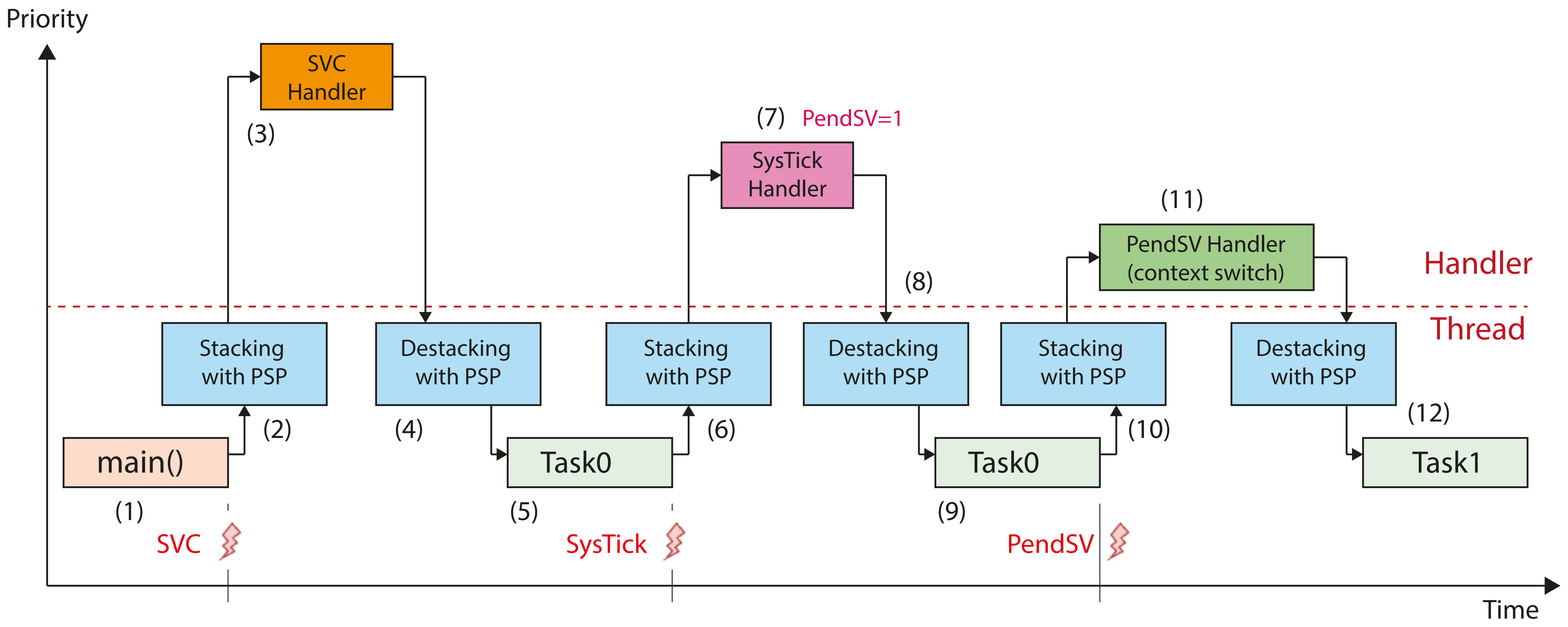

Application
Middleware
CMSIS-RTOS API v2
Real-time Kernel (3rd Party)


startup_<device>.s
CMSIS device startup interrupt vectors
startup_<device>.c
CMSIS system and clock configuration
<device>.h
CMSIS device peripheral access
<rtos>.h
Implementation specific interface
<user>.c/cpp
User application
main() {...}
RTOS.lib
CMSIS compliant RTOS library
RTOS_config.c/.h
CMSIS-RTOS configuration
cmsis_os2.h
Standard CMSIS-RTOS interface
CMSIS-CORE device files
CMSIS-RTOS standard files
Implementation specific RTOS files
User program

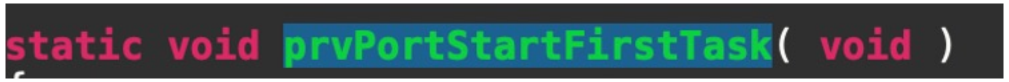
CMSIS-RTOS File Structure

Želite izvedeti več?

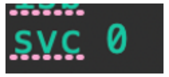
- https://interrupt.memfault.com/blog/cortex-m-rtos-context-switching
- https://mcuoneclipse.com/2016/08/28/arm-cortex-m-interrupts-and-freertos-part-3/


Priority
SVC Handler
(3)
(7) PendSV=1
SysTick Handler
(11)
PendSV Handler (context switch)
Handler
(8)
Thread
Stacking with PSP
Destacking with PSP
Stacking with PSP
Destacking with PSP
Stacking with PSP
Destacking with PSP
main()
(2)
(4)
Task0
(6)
Task0
(10)
(12)
Task1
(1)
SVC
(5)
SysTick
(9)
PendSV
Time

Zagon prvega opravila in razvrščevalnika

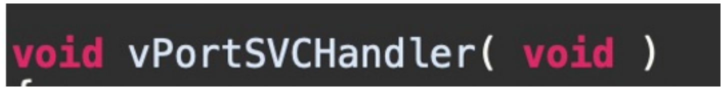


```
static void prvPortStartFirstTask( void )
```



```
svc 0
```

Z ukazom SVC sprožimo “SuperVisor Call” izjemo

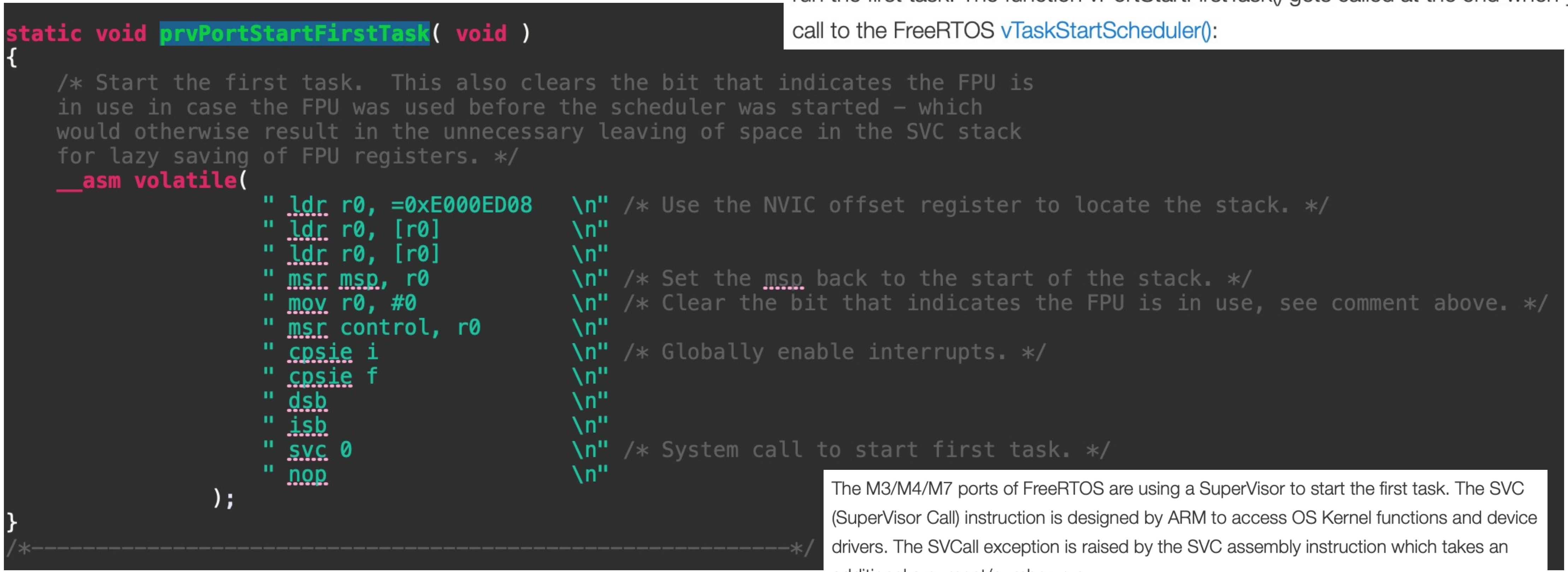


```
void vPortSVCHandler( void )
```

1. Nastavimo PSP prvega opravila iz strukture Task Control Block
2. Preberemo prvo opravilo s skladu (registre r4-r11 ter r14)
3. Omogoči prekinitve z najnižjo prioriteto (za PendSV)
4. Z ukazom bx r14 se z destackingom ”vrnemo” v prvo opravilo in čakamo na periodične prekinitve za task switching

Zagon prvega opravila

For M3/M4/M7, FreeRTOS uses the SVC in a single place: when it starts the scheduler to run the first task. The function vPortStartFirstTask() gets called at the end when you do a call to the FreeRTOS vTaskStartScheduler():

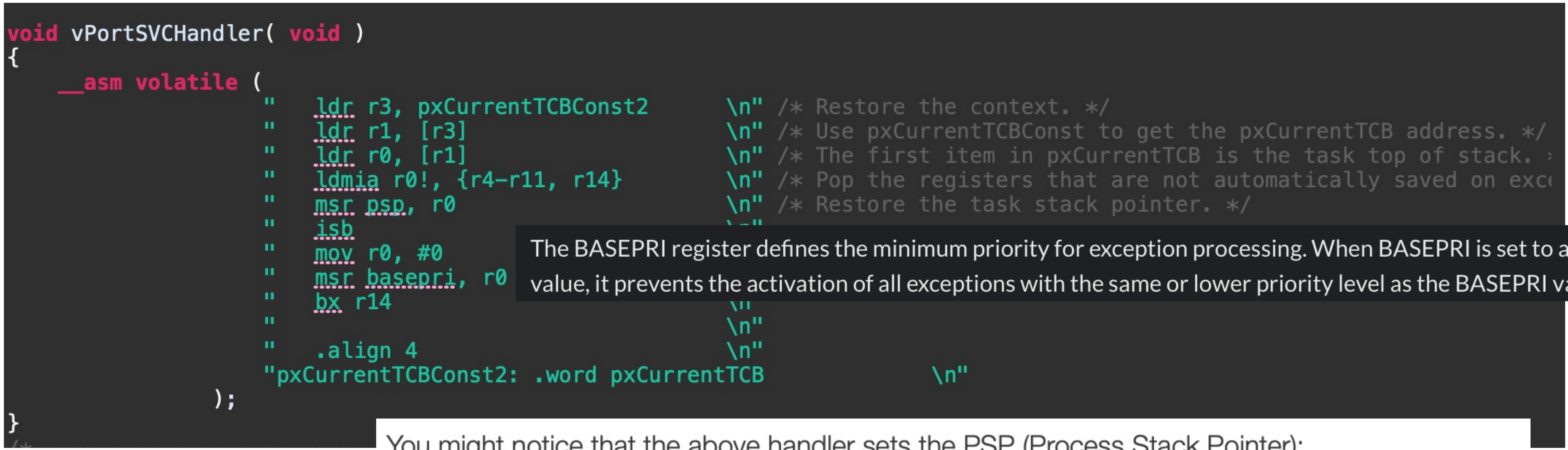


```
static void prvPortStartFirstTask( void )
{
    /* Start the first task.  This also clears the bit that indicates the FPU is
    in use in case the FPU was used before the scheduler was started - which
    would otherwise result in the unnecessary leaving of space in the SVC stack
    for lazy saving of FPU registers. */
    __asm volatile(
                    " ldr r0, =0xE000ED08   \n" /* Use the NVIC offset register to locate the stack. */
                    " ldr r0, [r0]          \n"
                    " ldr r0, [r0]          \n"
                    " msr msp, r0           \n" /* Set the msp back to the start of the stack. */
                    " mov r0, #0            \n" /* Clear the bit that indicates the FPU is in use, see comment above. */
                    " msr control, r0       \n"
                    " cpsie i               \n" /* Globally enable interrupts. */
                    " cpsie f               \n"
                    " dsb                   \n"
                    " isb                   \n"
                    " svc 0                 \n" /* System call to start first task. */
                    " nop                   \n"
                );
}
/*-----------------------------------------------------------*/
```

The M3/M4/M7 ports of FreeRTOS are using a SuperVisor to start the first task. The SVC (SuperVisor Call) instruction is designed by ARM to access OS Kernel functions and device drivers. The SVCall exception is raised by the SVC assembly instruction which takes an additional argument/number, e.g.

There are several different strategies but a common pattern an RTOS will follow when creating a new task is to initialize the task stack to look like it had been context switched out by the scheduler. Then to start the scheduler itself by triggering a SVC exception with the svc instruction.

SVC Handler:

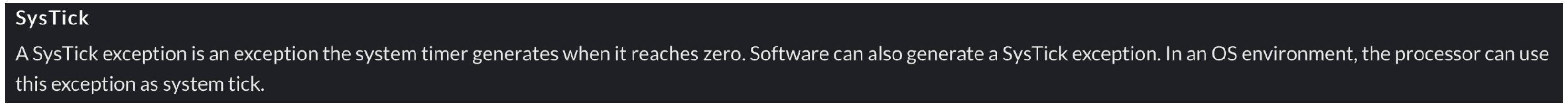


```
void vPortSVCHandler( void )
{
    __asm volatile (
                    "   ldr r3, pxCurrentTCBConst2      \n" /* Restore the context. */
                    "   ldr r1, [r3]                    \n" /* Use pxCurrentTCBConst to get the pxCurrentTCB address. */
                    "   ldr r0, [r1]                    \n" /* The first item in pxCurrentTCB is the task top of stack. *
                    "   ldmia r0!, {r4-r11, r14}        \n" /* Pop the registers that are not automatically saved on exc
                    "   msr psp, r0                     \n" /* Restore the task stack pointer. */
                    "   isb                             
                    "   mov r0, #0                      
                    "   msr basepri, r0                 
                    "   bx r14                          
                    "                                   \n"
                    "   .align 4                        \n"
                    "pxCurrentTCBConst2: .word pxCurrentTCB             \n"
                );
}
```

The BASEPRI register defines the minimum priority for exception processing. When BASEPRI is set to a nonzero value, it prevents the activation of all exceptions with the same or lower priority level as the BASEPRI value. See

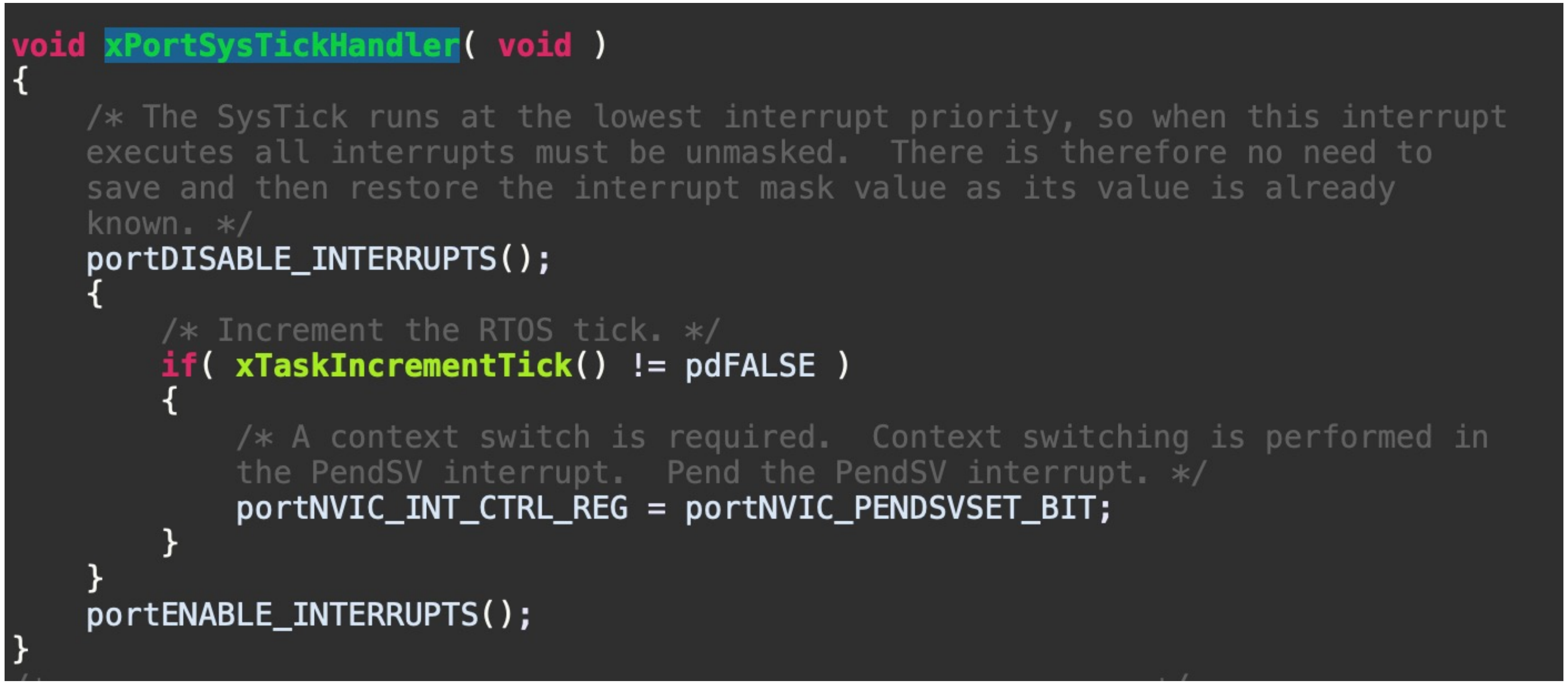
You might notice that the above handler sets the PSP (Process Stack Pointer):

```
"msr psp, r0 \n" /* Remember the new top of stack for the task.
```


SysTick

A SysTick exception is an exception the system timer generates when it reaches zero. Software can also generate a SysTick exception. In an OS environment, the processor can use this exception as system tick.

Systick Handler:



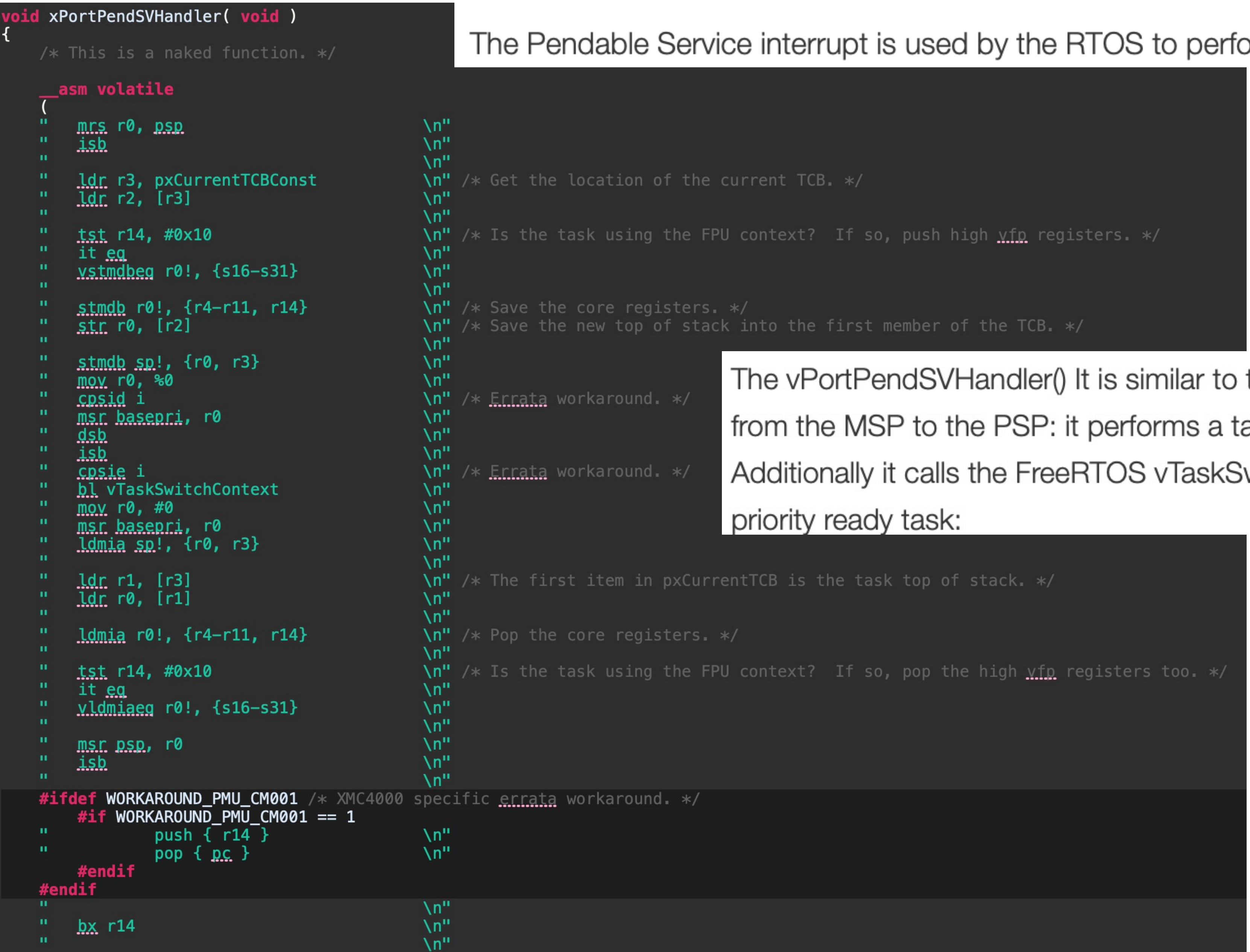
```
void xPortSysTickHandler( void )
{
    /* The SysTick runs at the lowest interrupt priority, so when this interrupt
    executes all interrupts must be unmasked.  There is therefore no need to
    save and then restore the interrupt mask value as its value is already
    known. */
    portDISABLE_INTERRUPTS();
    {
        /* Increment the RTOS tick. */
        if( xTaskIncrementTick() != pdFALSE )
        {
            /* A context switch is required.  Context switching is performed in
            the PendSV interrupt.  Pend the PendSV interrupt. */
            portNVIC_INT_CTRL_REG = portNVIC_PENDSVSET_BIT;
        }
    }
    portENABLE_INTERRUPTS();
}
```

The **FreeRTOS** scheduler works by utilizing the built in **SysTick** and **PendSV** interrupts. The **SysTick** is configured to fire periodically. Each time it fires, a check is performed to see if a context switch is required by calling `xTaskIncrementTick`:

PendSV is an interrupt-driven request for system-level service. In an OS environment, use PendSV for context switching when no other exception is active.

PendSV Handler:

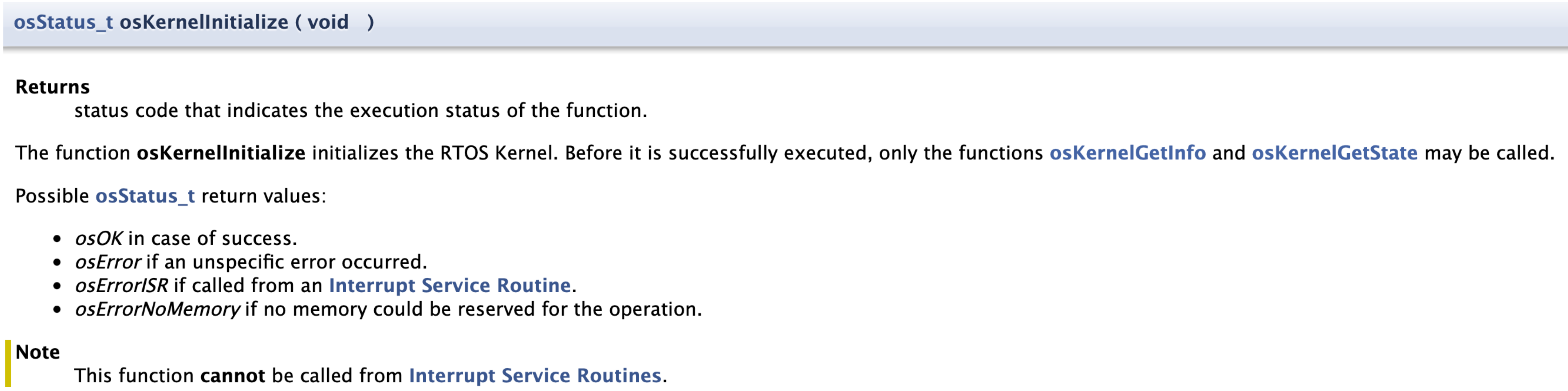
The Pendable Service interrupt is used by the RTOS to perform a context switch.



```
void xPortPendSVHandler( void )
{
    /* This is a naked function. */

    __asm volatile
    (
    "   mrs r0, psp                         \n"
    "   isb                                 \n"
    "                                       \n"
    "   ldr r3, pxCurrentTCBConst           \n" /* Get the location of the current TCB. */
    "   ldr r2, [r3]                        \n"
    "                                       \n"
    "   tst r14, #0x10                      \n" /* Is the task using the FPU context?  If so, push high vfp registers. */
    "   it eq                               \n"
    "   vstmdbeq r0!, {s16-s31}             \n"
    "                                       \n"
    "   stmdb r0!, {r4-r11, r14}            \n" /* Save the core registers. */
    "   str r0, [r2]                        \n" /* Save the new top of stack into the first member of the TCB. */
    "                                       \n"
    "   stmdb sp!, {r0, r3}                 \n"
    "   mov r0, %0                          \n"
    "   cpsid i                             \n" /* Errata workaround. */
    "   msr basepri, r0                     \n"
    "   dsb                                 \n"
    "   isb                                 \n"
    "   cpsie i                             \n" /* Errata workaround. */
    "   bl vTaskSwitchContext               \n"
    "   mov r0, #0                          \n"
    "   msr basepri, r0                     \n"
    "   ldmia sp!, {r0, r3}                 \n"
    "                                       \n"
    "   ldr r1, [r3]                        \n" /* The first item in pxCurrentTCB is the task top of stack. */
    "   ldr r0, [r1]                        \n"
    "                                       \n"
    "   ldmia r0!, {r4-r11, r14}            \n" /* Pop the core registers. */
    "                                       \n"
    "   tst r14, #0x10                      \n" /* Is the task using the FPU context?  If so, pop the high vfp registers too. */
    "   it eq                               \n"
    "   vldmiaeq r0!, {s16-s31}             \n"
    "                                       \n"
    "   msr psp, r0                         \n"
    "   isb                                 \n"
    "                                       \n"
    #ifdef WORKAROUND_PMU_CM001 /* XMC4000 specific errata workaround. */
        #if WORKAROUND_PMU_CM001 == 1
    "           push { r14 }                \n"
    "           pop { pc }                  \n"
        #endif
    #endif
    "                                       \n"
    "   bx r14                              \n"
    "                                       \n"
```

The vPortPendSVHandler() It is similar to the vPortSVCHandler(). But it does not switch from the MSP to the PSP: it performs a task context switch between different PSP values. Additionally it calls the FreeRTOS vTaskSwitchContext() which selects the highest (RTOS!) priority ready task:

Inicializacija jedra OS



osStatus_t osKernelInitialize (void)

Returns

status code that indicates the execution status of the function.

The function **osKernelInitialize** initializes the RTOS Kernel. Before it is successfully executed, only the functions **osKernelGetInfo** and **osKernelGetState** may be called.

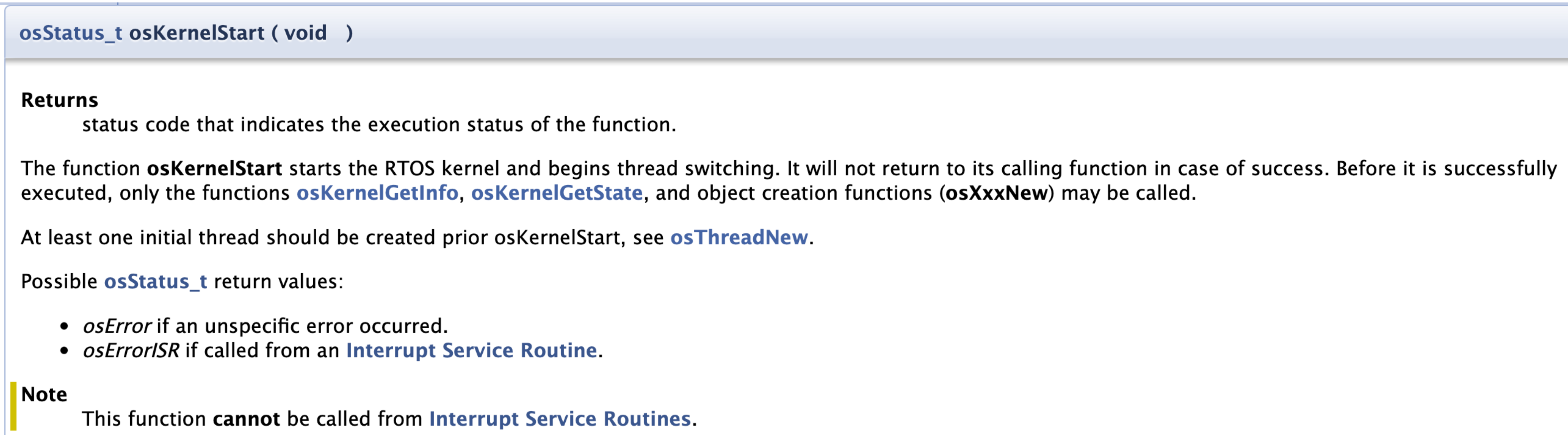
Possible **osStatus_t** return values:

- *osOK* in case of success.
- *osError* if an unspecific error occurred.
- *osErrorISR* if called from an **Interrupt Service Routine**.
- *osErrorNoMemory* if no memory could be reserved for the operation.

Note

This function **cannot** be called from **Interrupt Service Routines**.

Zagon razvrščevalnika



osStatus_t osKernelStart (void)

Returns

status code that indicates the execution status of the function.

The function **osKernelStart** starts the RTOS kernel and begins thread switching. It will not return to its calling function in case of success. Before it is successfully executed, only the functions osKernelGetInfo, osKernelGetState, and object creation functions (**osXxxNew**) may be called.

At least one initial thread should be created prior osKernelStart, see osThreadNew.

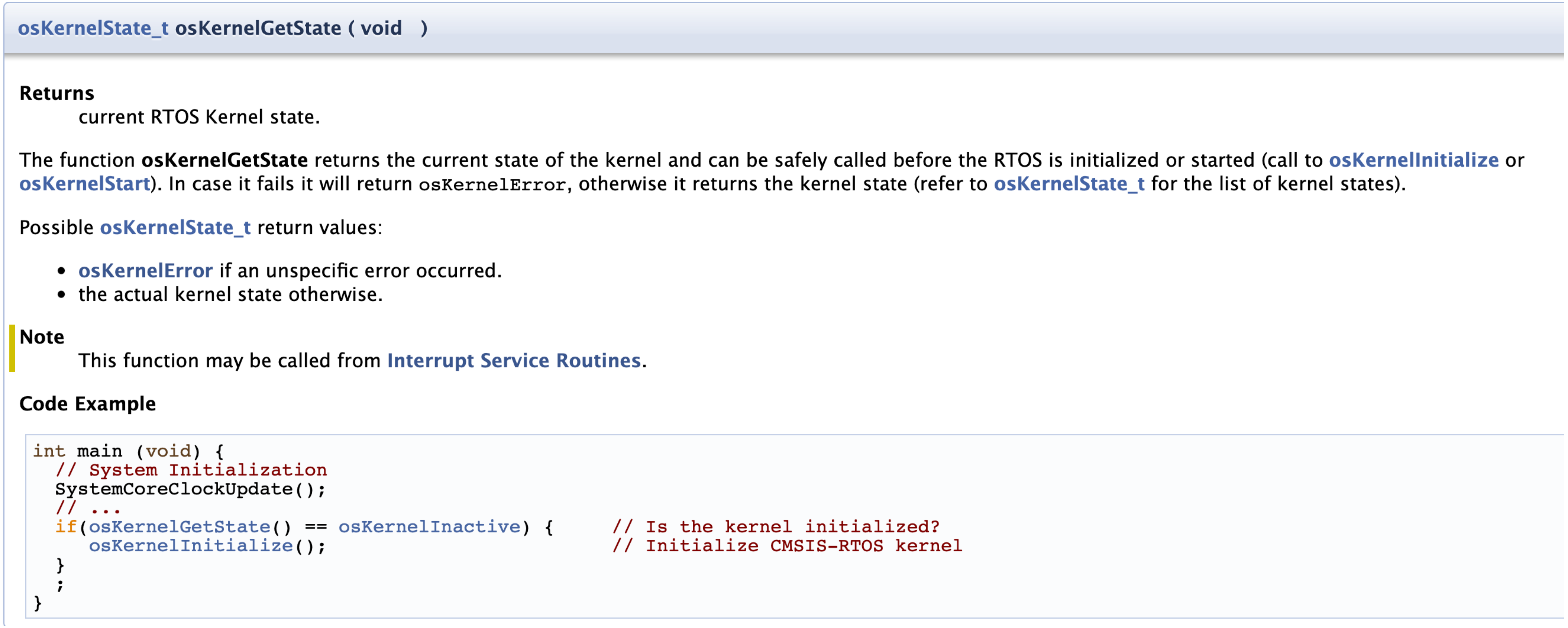
Possible osStatus_t return values:

- *osError* if an unspecific error occurred.
- *osErrorISR* if called from an Interrupt Service Routine.

Note

This function **cannot** be called from Interrupt Service Routines.

Stanje jedra



osKernelState_t osKernelGetState (void)

Returns

current RTOS Kernel state.

The function **osKernelGetState** returns the current state of the kernel and can be safely called before the RTOS is initialized or started (call to osKernelInitialize or osKernelStart). In case it fails it will return `osKernelError`, otherwise it returns the kernel state (refer to osKernelState_t for the list of kernel states).

Possible osKernelState_t return values:

- osKernelError if an unspecific error occurred.
- the actual kernel state otherwise.

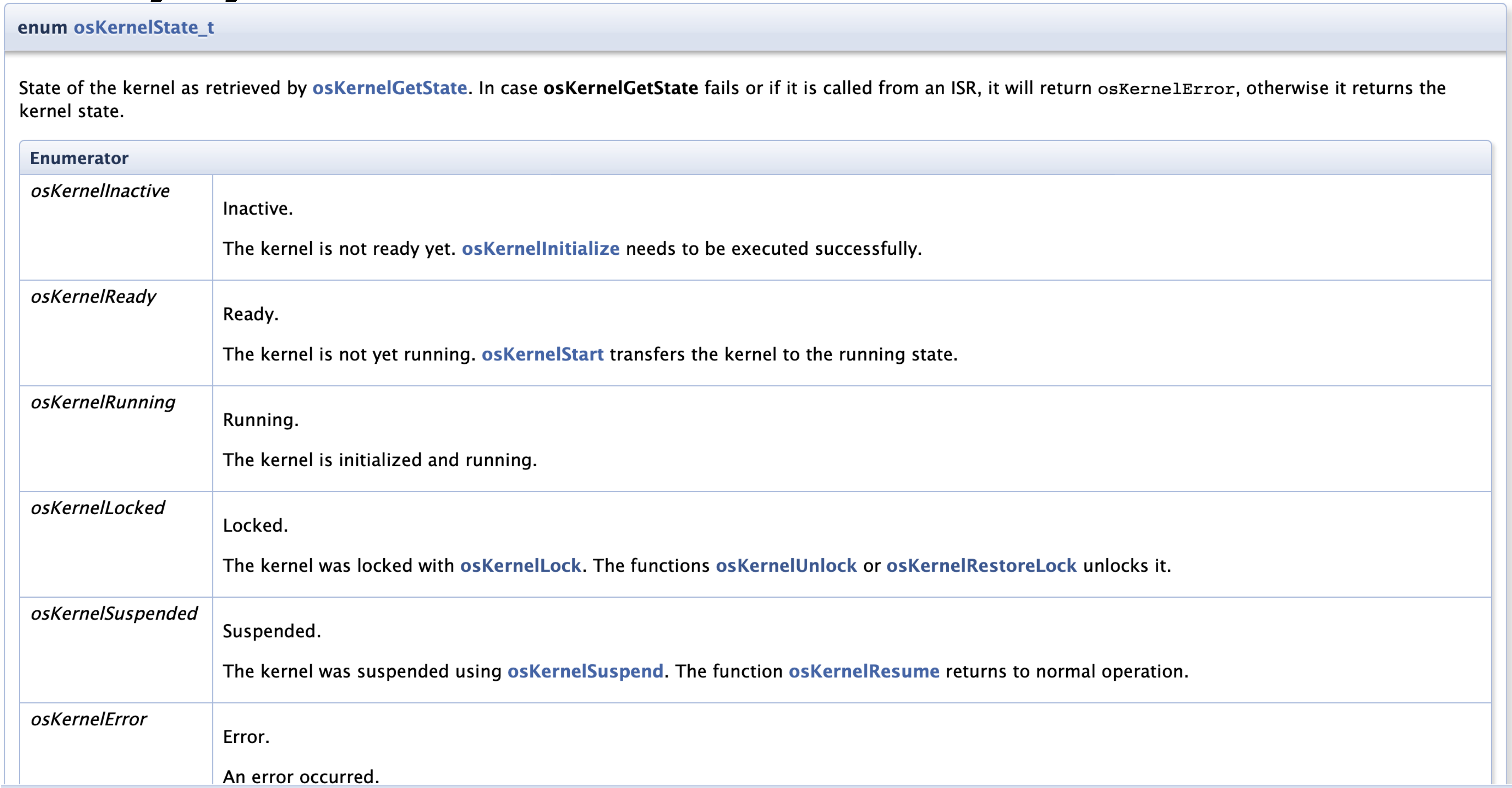
Note

This function may be called from **Interrupt Service Routines**.

Code Example

```
int main (void) {
  // System Initialization
  SystemCoreClockUpdate();
  // ...
  if(osKernelGetState() == osKernelInactive) {     // Is the kernel initialized?
     osKernelInitialize();                         // Initialize CMSIS-RTOS kernel
  }
  ;
}
```

Stanje jedra



enum osKernelState_t

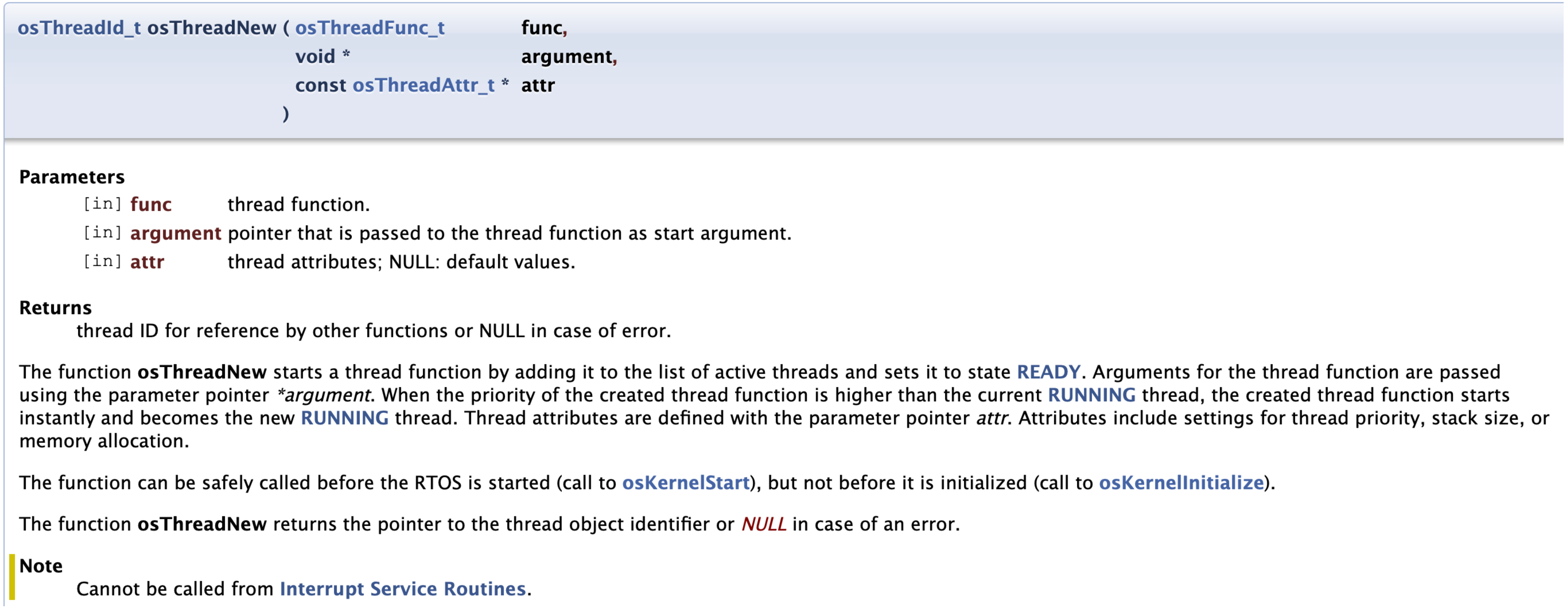
State of the kernel as retrieved by **osKernelGetState**. In case **osKernelGetState** fails or if it is called from an ISR, it will return `osKernelError`, otherwise it returns the kernel state.

Enumerator	
osKernelInactive	Inactive. The kernel is not ready yet. **osKernelInitialize** needs to be executed successfully.
osKernelReady	Ready. The kernel is not yet running. **osKernelStart** transfers the kernel to the running state.
osKernelRunning	Running. The kernel is initialized and running.
osKernelLocked	Locked. The kernel was locked with **osKernelLock**. The functions **osKernelUnlock** or **osKernelRestoreLock** unlocks it.
osKernelSuspended	Suspended. The kernel was suspended using **osKernelSuspend**. The function **osKernelResume** returns to normal operation.
osKernelError	Error. An error occurred.

Zgled

```c
int main (void) {
  // System Initialization
  SystemCoreClockUpdate();
  // ...
  if(osKernelGetState() == osKernelInactive) {
    osKernelInitialize();
  }
  ; // ... Start Threads
  if (osKernelGetState() == osKernelReady) {        // If kernel is ready to run...
    osKernelStart();                                // ... start thread execution
    }

  while(1);                                         // only reached in case of error
}
```

Ustvarjanje niti



```
osThreadId_t osThreadNew ( osThreadFunc_t           func,
                           void *                   argument,
                           const osThreadAttr_t *   attr
                         )
```

Parameters

[in]	**func**	thread function.
[in]	**argument**	pointer that is passed to the thread function as start argument.
[in]	**attr**	thread attributes; NULL: default values.

Returns

thread ID for reference by other functions or NULL in case of error.

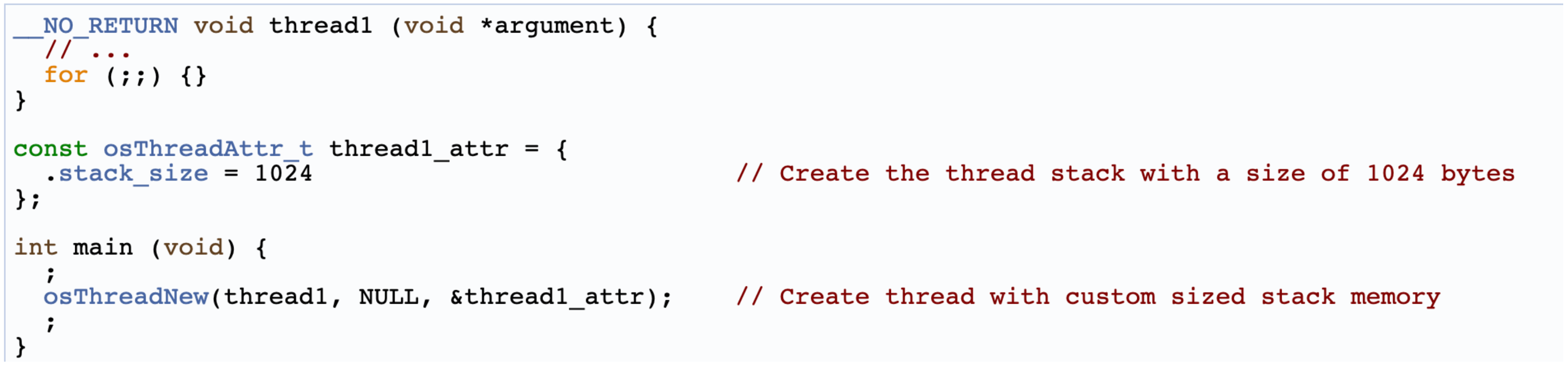
The function **osThreadNew** starts a thread function by adding it to the list of active threads and sets it to state **READY**. Arguments for the thread function are passed using the parameter pointer **argument*. When the priority of the created thread function is higher than the current **RUNNING** thread, the created thread function starts instantly and becomes the new **RUNNING** thread. Thread attributes are defined with the parameter pointer *attr*. Attributes include settings for thread priority, stack size, or memory allocation.

The function can be safely called before the RTOS is started (call to **osKernelStart**), but not before it is initialized (call to **osKernelInitialize**).

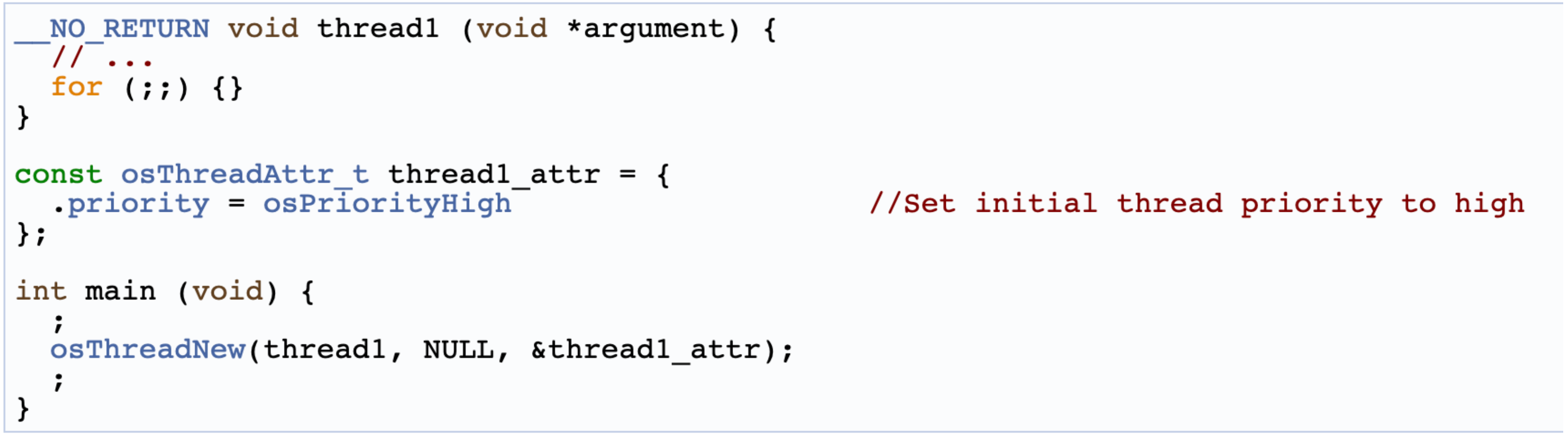
The function **osThreadNew** returns the pointer to the thread object identifier or *NULL* in case of an error.

Note

Cannot be called from **Interrupt Service Routines**.

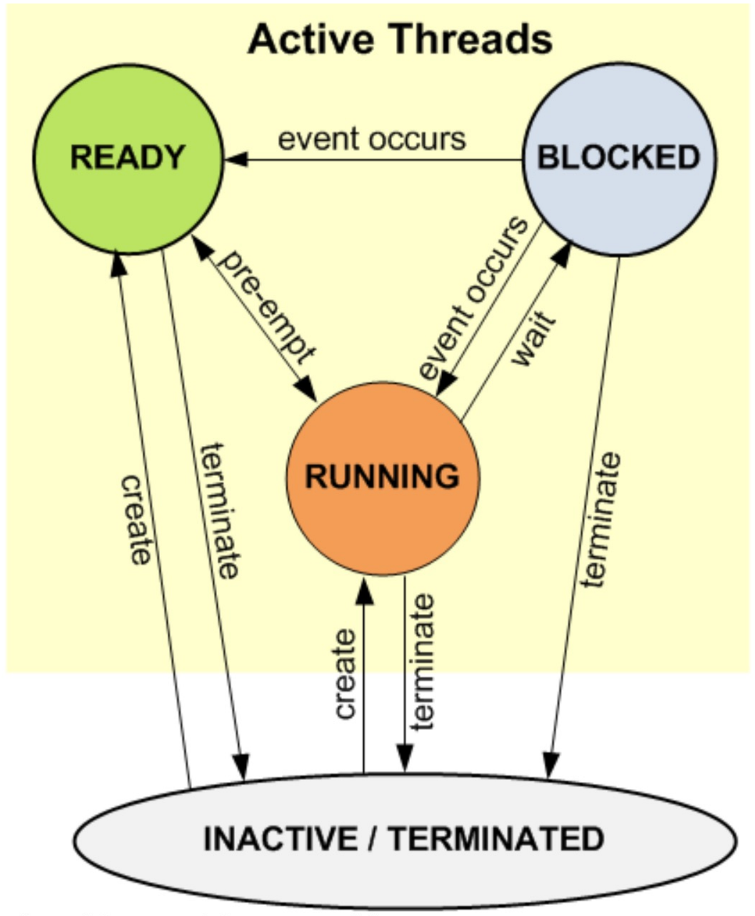


```c
__NO_RETURN void thread1 (void *argument) {
  // ...
  for (;;) {}
}

const osThreadAttr_t thread1_attr = {
  .stack_size = 1024                             // Create the thread stack with a size of 1024 bytes
};

int main (void) {
  ;
  osThreadNew(thread1, NULL, &thread1_attr);    // Create thread with custom sized stack memory
  ;
}
```

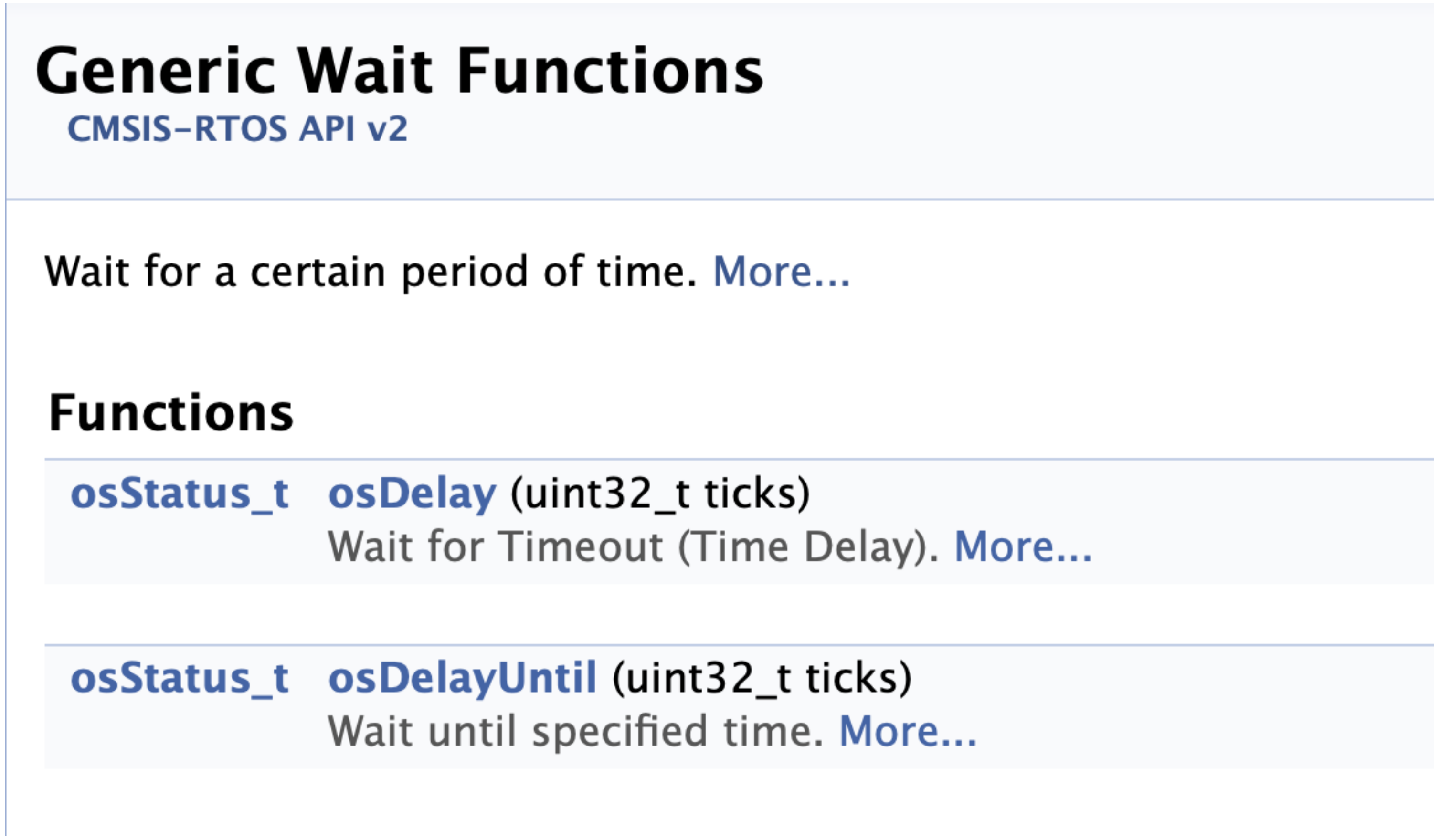


```c
__NO_RETURN void thread1 (void *argument) {
  // ...
  for (;;) {}
}

const osThreadAttr_t thread1_attr = {
  .priority = osPriorityHigh                     //Set initial thread priority to high
};

int main (void) {
  ;
  osThreadNew(thread1, NULL, &thread1_attr);
  ;
}
```

Niti


Active Threads
READY
BLOCKED
RUNNING
INACTIVE / TERMINATED
event occurs
pre-empt
event occurs
wait
create
terminate
create
terminate
terminate

Zakasnitve



Generic Wait Functions
CMSIS-RTOS API v2

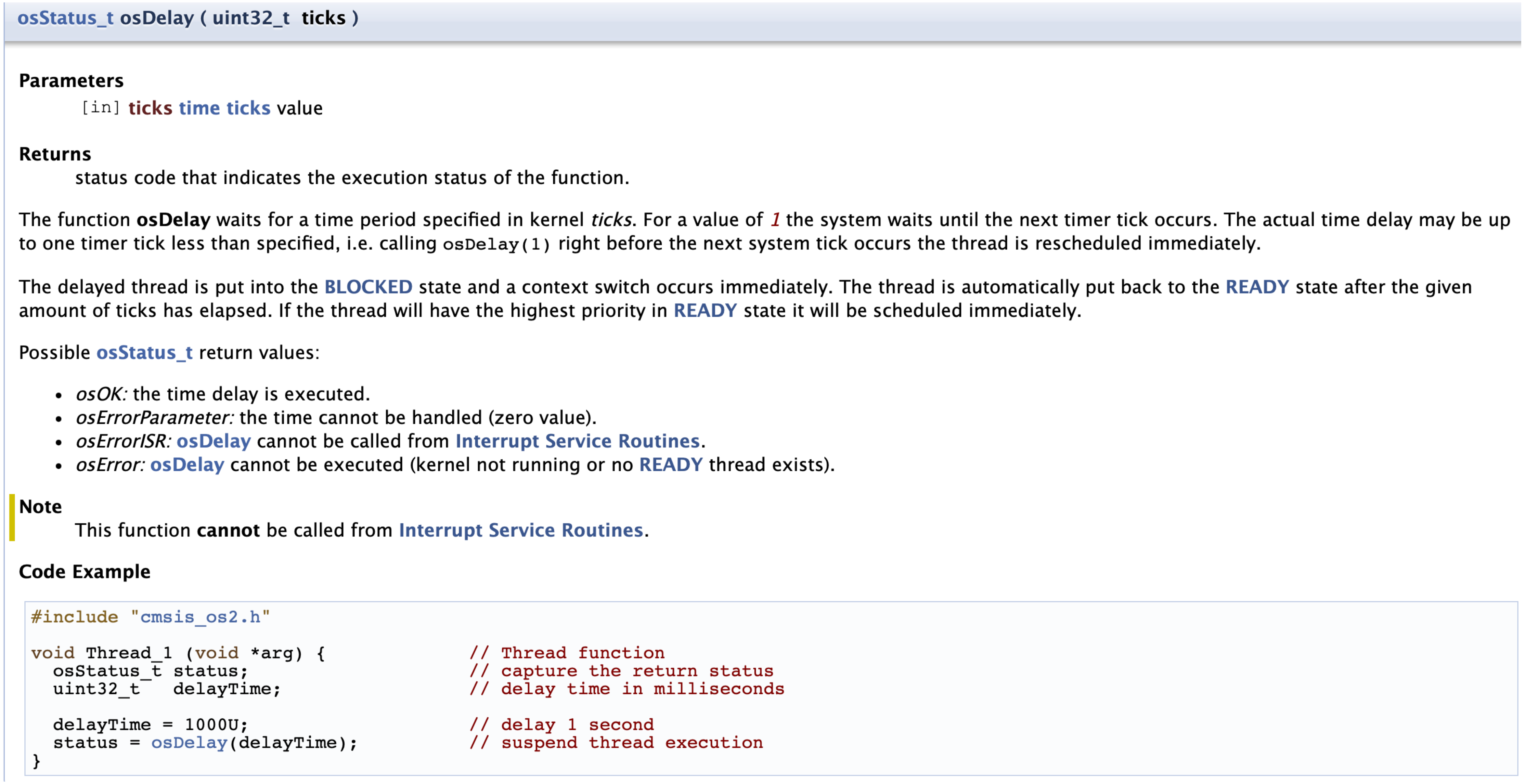
Wait for a certain period of time. More...

Functions

osStatus_t osDelay (uint32_t ticks)
Wait for Timeout (Time Delay). More...

osStatus_t osDelayUntil (uint32_t ticks)
Wait until specified time. More...

Zakasnitve



osStatus_t osDelay (uint32_t ticks)

Parameters

`[in]` **ticks** **time ticks** value

Returns

status code that indicates the execution status of the function.

The function **osDelay** waits for a time period specified in kernel *ticks*. For a value of *1* the system waits until the next timer tick occurs. The actual time delay may be up to one timer tick less than specified, i.e. calling `osDelay(1)` right before the next system tick occurs the thread is rescheduled immediately.

The delayed thread is put into the **BLOCKED** state and a context switch occurs immediately. The thread is automatically put back to the **READY** state after the given amount of ticks has elapsed. If the thread will have the highest priority in **READY** state it will be scheduled immediately.

Possible **osStatus_t** return values:

- *osOK:* the time delay is executed.
- *osErrorParameter:* the time cannot be handled (zero value).
- *osErrorISR:* **osDelay** cannot be called from **Interrupt Service Routines**.
- *osError:* **osDelay** cannot be executed (kernel not running or no **READY** thread exists).

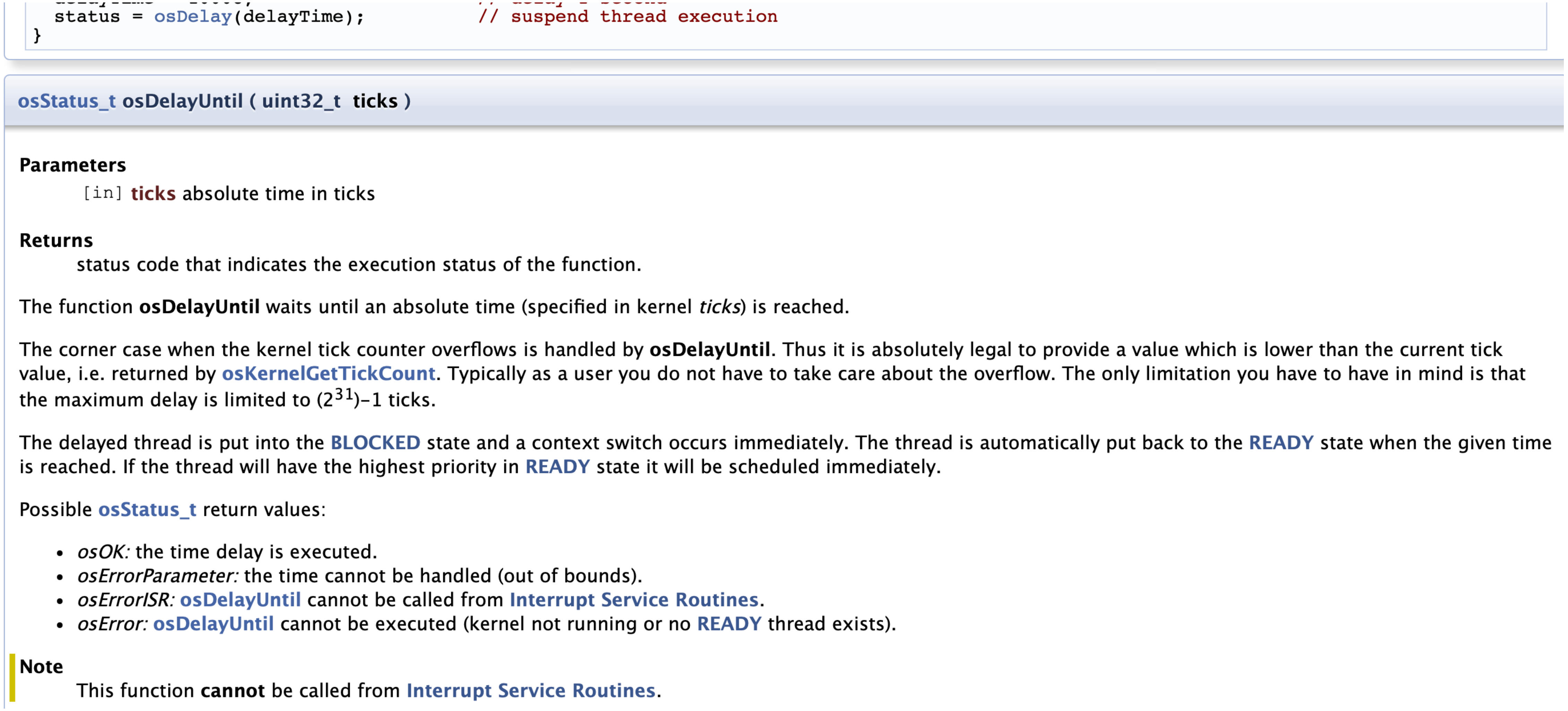
Note

This function **cannot** be called from **Interrupt Service Routines**.

Code Example

```
#include "cmsis_os2.h"

void Thread_1 (void *arg) {                // Thread function
  osStatus_t status;                       // capture the return status
  uint32_t   delayTime;                    // delay time in milliseconds

  delayTime = 1000U;                       // delay 1 second
  status = osDelay(delayTime);             // suspend thread execution
}
```

Zakasnitve



```
  status = osDelay(delayTime);          // suspend thread execution
}
```

osStatus_t osDelayUntil (uint32_t ticks)

Parameters

[in] **ticks** absolute time in ticks

Returns

status code that indicates the execution status of the function.

The function **osDelayUntil** waits until an absolute time (specified in kernel *ticks*) is reached.

The corner case when the kernel tick counter overflows is handled by **osDelayUntil**. Thus it is absolutely legal to provide a value which is lower than the current tick value, i.e. returned by osKernelGetTickCount. Typically as a user you do not have to take care about the overflow. The only limitation you have to have in mind is that the maximum delay is limited to $(2^{31})-1$ ticks.

The delayed thread is put into the BLOCKED state and a context switch occurs immediately. The thread is automatically put back to the READY state when the given time is reached. If the thread will have the highest priority in READY state it will be scheduled immediately.

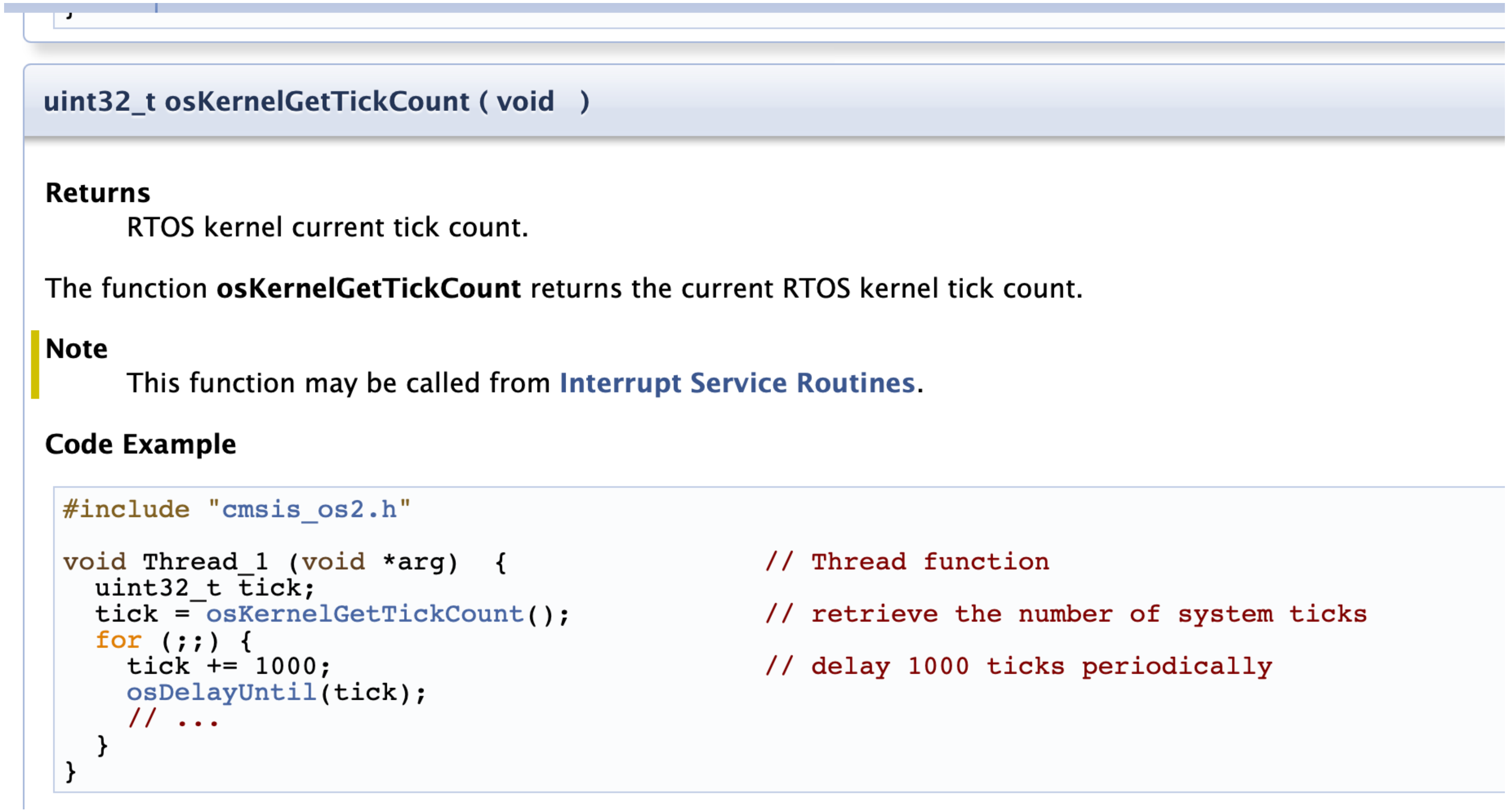
Possible osStatus_t return values:

- *osOK:* the time delay is executed.
- *osErrorParameter:* the time cannot be handled (out of bounds).
- *osErrorISR:* osDelayUntil cannot be called from Interrupt Service Routines.
- *osError:* osDelayUntil cannot be executed (kernel not running or no READY thread exists).

Note

This function **cannot** be called from Interrupt Service Routines.

Zakasnitve



uint32_t osKernelGetTickCount (void)

Returns
RTOS kernel current tick count.

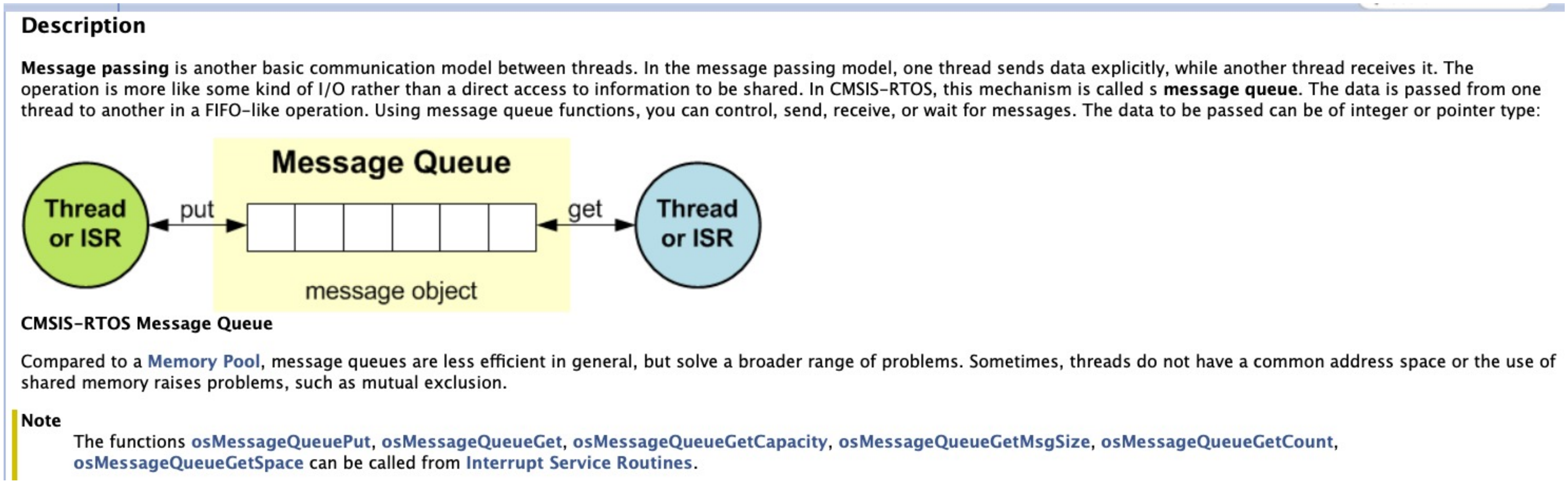
The function **osKernelGetTickCount** returns the current RTOS kernel tick count.

Note
This function may be called from **Interrupt Service Routines**.

Code Example

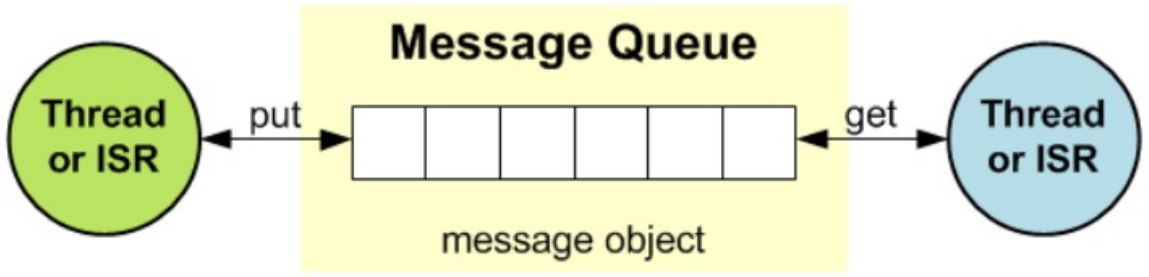
```
#include "cmsis_os2.h"

void Thread_1 (void *arg)  {                           // Thread function
  uint32_t tick;
  tick = osKernelGetTickCount();                       // retrieve the number of system ticks
  for (;;) {
    tick += 1000;                                      // delay 1000 ticks periodically
    osDelayUntil(tick);
    // ...
  }
}
```

Vrste



Description

Message passing is another basic communication model between threads. In the message passing model, one thread sends data explicitly, while another thread receives it. The operation is more like some kind of I/O rather than a direct access to information to be shared. In CMSIS-RTOS, this mechanism is called s **message queue**. The data is passed from one thread to another in a FIFO-like operation. Using message queue functions, you can control, send, receive, or wait for messages. The data to be passed can be of integer or pointer type:



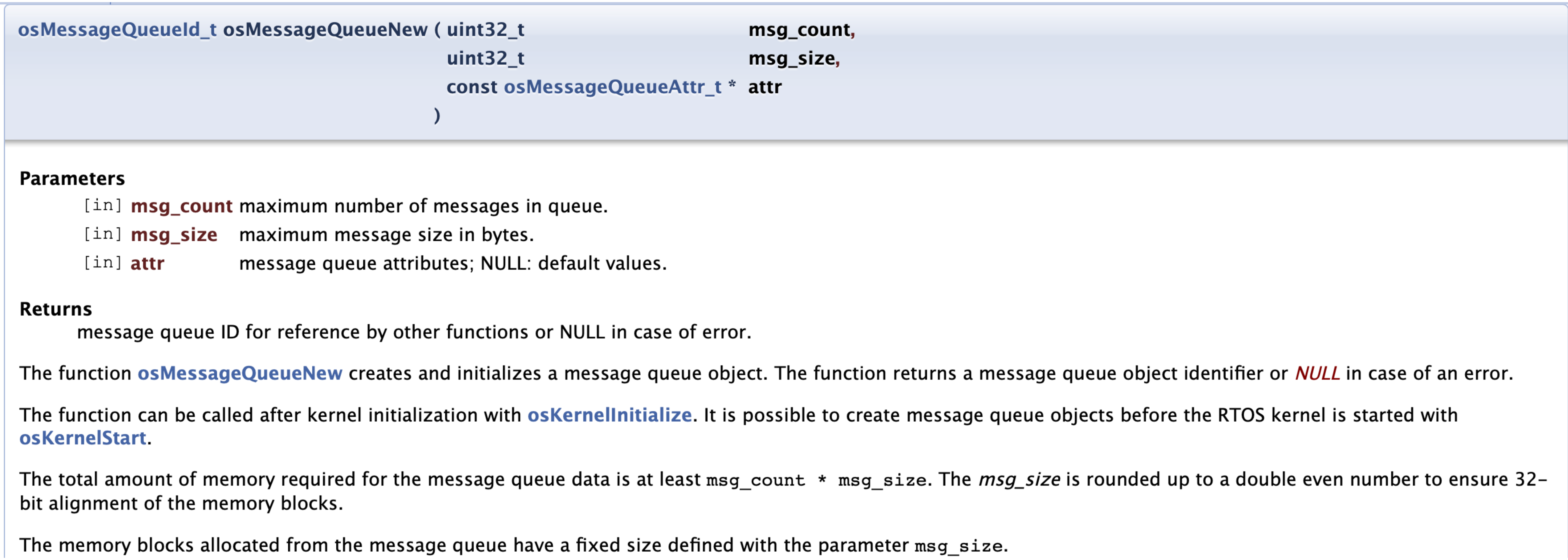
CMSIS-RTOS Message Queue

Compared to a Memory Pool, message queues are less efficient in general, but solve a broader range of problems. Sometimes, threads do not have a common address space or the use of shared memory raises problems, such as mutual exclusion.

Note

The functions osMessageQueuePut, osMessageQueueGet, osMessageQueueGetCapacity, osMessageQueueGetMsgSize, osMessageQueueGetCount, osMessageQueueGetSpace can be called from Interrupt Service Routines.

Vrste - ustvarjanje



```
osMessageQueueId_t osMessageQueueNew ( uint32_t                       msg_count,
                                       uint32_t                       msg_size,
                                       const osMessageQueueAttr_t *   attr
                                     )
```

Parameters

[in]	**msg_count**	maximum number of messages in queue.
[in]	**msg_size**	maximum message size in bytes.
[in]	**attr**	message queue attributes; NULL: default values.

Returns

message queue ID for reference by other functions or NULL in case of error.

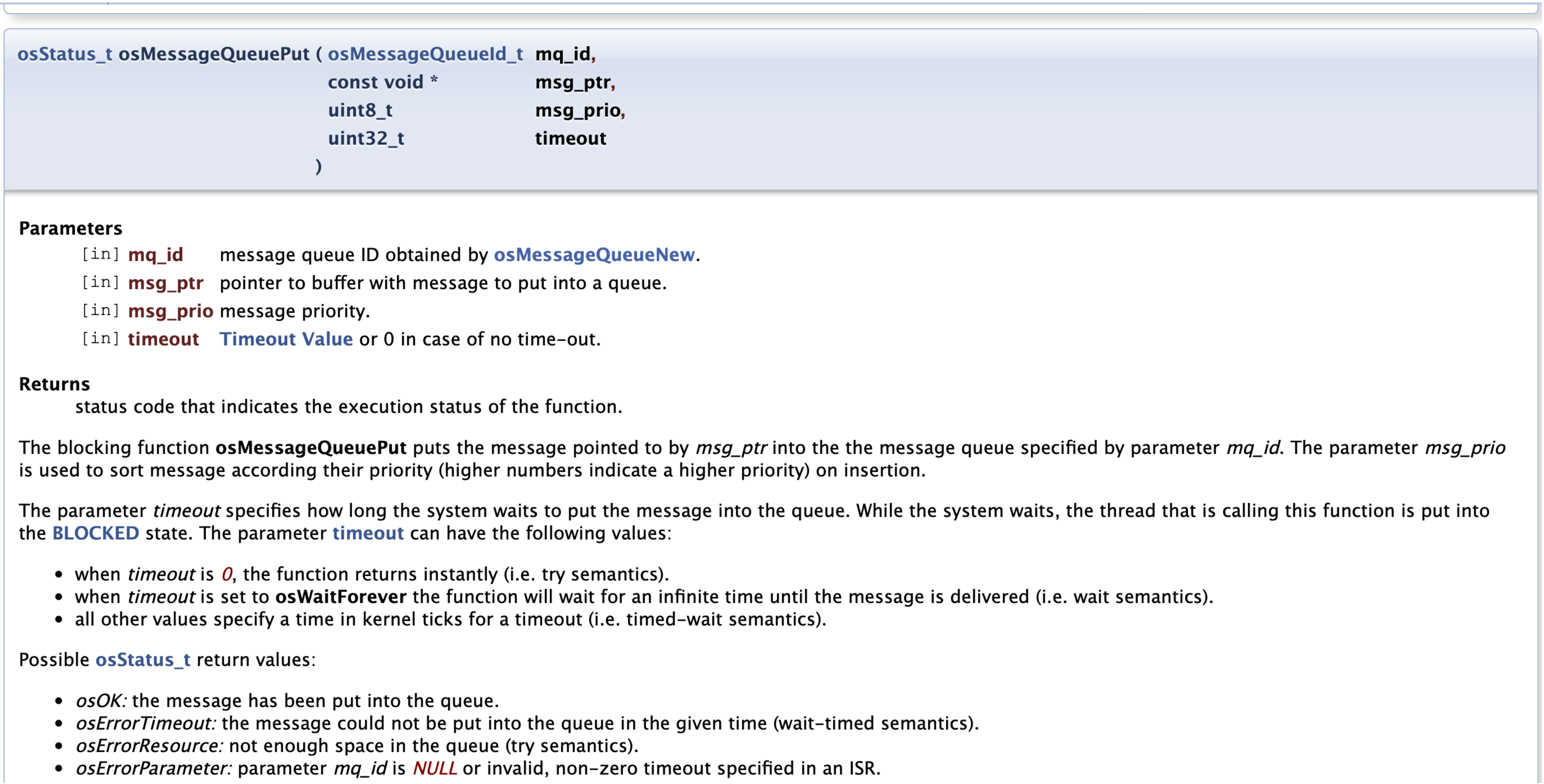
The function **osMessageQueueNew** creates and initializes a message queue object. The function returns a message queue object identifier or *NULL* in case of an error.

The function can be called after kernel initialization with **osKernelInitialize**. It is possible to create message queue objects before the RTOS kernel is started with **osKernelStart**.

The total amount of memory required for the message queue data is at least `msg_count * msg_size`. The *msg_size* is rounded up to a double even number to ensure 32-bit alignment of the memory blocks.

The memory blocks allocated from the message queue have a fixed size defined with the parameter `msg_size`.

Vrste - Put



```
osStatus_t osMessageQueuePut ( osMessageQueueId_t  mq_id,
                               const void *        msg_ptr,
                               uint8_t             msg_prio,
                               uint32_t            timeout
                             )
```

Parameters

[in]	**mq_id**	message queue ID obtained by osMessageQueueNew.
[in]	**msg_ptr**	pointer to buffer with message to put into a queue.
[in]	**msg_prio**	message priority.
[in]	**timeout**	Timeout Value or 0 in case of no time-out.

Returns

status code that indicates the execution status of the function.

The blocking function **osMessageQueuePut** puts the message pointed to by *msg_ptr* into the the message queue specified by parameter *mq_id*. The parameter *msg_prio* is used to sort message according their priority (higher numbers indicate a higher priority) on insertion.

The parameter *timeout* specifies how long the system waits to put the message into the queue. While the system waits, the thread that is calling this function is put into the BLOCKED state. The parameter timeout can have the following values:

- when *timeout* is *0*, the function returns instantly (i.e. try semantics).
- when *timeout* is set to **osWaitForever** the function will wait for an infinite time until the message is delivered (i.e. wait semantics).
- all other values specify a time in kernel ticks for a timeout (i.e. timed-wait semantics).

Possible osStatus_t return values:

- *osOK:* the message has been put into the queue.
- *osErrorTimeout:* the message could not be put into the queue in the given time (wait-timed semantics).
- *osErrorResource:* not enough space in the queue (try semantics).
- *osErrorParameter:* parameter *mq_id* is *NULL* or invalid, non-zero timeout specified in an ISR.

Vrste - Get

```
osStatus_t osMessageQueueGet ( osMessageQueueId_t  mq_id,
                               void *              msg_ptr,
                               uint8_t *           msg_prio,
                               uint32_t            timeout
                             )
```

Parameters

[in]	**mq_id**	message queue ID obtained by **osMessageQueueNew**.
[out]	**msg_ptr**	pointer to buffer for message to get from a queue.
[out]	**msg_prio**	pointer to buffer for message priority or NULL.
[in]	**timeout**	**Timeout Value** or 0 in case of no time-out.

Returns

status code that indicates the execution status of the function.

The function **osMessageQueueGet** retrieves a message from the message queue specified by the parameter *mq_id* and saves it to the buffer pointed to by the parameter *msg_ptr*. The message priority is stored to parameter *msg_prio* if not token{NULL}.

The parameter *timeout* specifies how long the system waits to retrieve the message from the queue. While the system waits, the thread that is calling this function is put into the **BLOCKED** state. The parameter **timeout** can have the following values:

- when *timeout* is *0*, the function returns instantly (i.e. try semantics).
- when *timeout* is set to **osWaitForever** the function will wait for an infinite time until the message is retrieved (i.e. wait semantics).
- all other values specify a time in kernel ticks for a timeout (i.e. timed-wait semantics).

Possible **osStatus_t** return values:

- *osOK:* the message has been retrieved from the queue.
- *osErrorTimeout:* the message could not be retrieved from the queue in the given time (timed-wait semantics).
- *osErrorResource:* nothing to get from the queue (try semantics).
- *osErrorParameter:* parameter *mq_id* is *NULL* or invalid, non-zero timeout specified in an ISR.

Vrste - Zgled

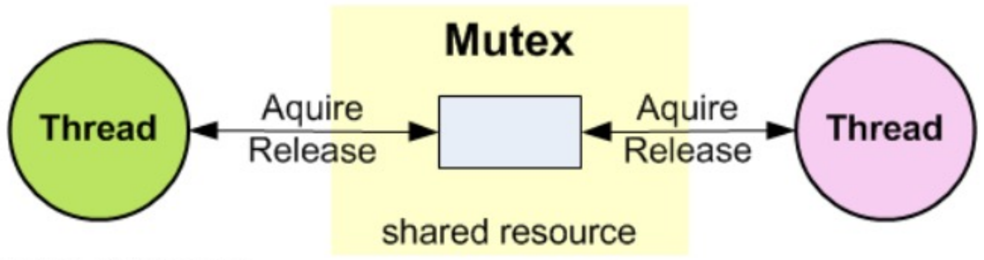
```c
#include "cmsis_os2.h"                          // CMSIS RTOS header file

/*----------------------------------------------------------------------------
 *      Message Queue creation & usage
 *---------------------------------------------------------------------------*/

#define MSGQUEUE_OBJECTS 16                     // number of Message Queue Objects

typedef struct {                                // object data type
  uint8_t Buf[32];
  uint8_t Idx;
} MSGQUEUE_OBJ_t;

osMessageQueueId_t mid_MsgQueue;                // message queue id

osThreadId_t tid_Thread_MsgQueue1;              // thread id 1
osThreadId_t tid_Thread_MsgQueue2;              // thread id 2

void Thread_MsgQueue1 (void *argument);         // thread function 1
void Thread_MsgQueue2 (void *argument);         // thread function 2

int Init_MsgQueue (void) {

  mid_MsgQueue = osMessageQueueNew(MSGQUEUE_OBJECTS, sizeof(MSGQUEUE_OBJ_t), NULL);
  if (mid_MsgQueue == NULL) {
    ; // Message Queue object not created, handle failure
  }

  tid_Thread_MsgQueue1 = osThreadNew(Thread_MsgQueue1, NULL, NULL);
  if (tid_Thread_MsgQueue1 == NULL) {
    return(-1);
  }
  tid_Thread_MsgQueue2 = osThreadNew(Thread_MsgQueue2, NULL, NULL);
  if (tid_Thread_MsgQueue2 == NULL) {
    return(-1);
  }

  return(0);
}
```

```c
void Thread_MsgQueue1 (void *argument) {
  MSGQUEUE_OBJ_t msg;

  while (1) {
    ; // Insert thread code here...
    msg.Buf[0] = 0x55U;                                         // do some work...
    msg.Idx    = 0U;
    osMessageQueuePut(mid_MsgQueue, &msg, 0U, 0U);
    osThreadYield();                                            // suspend thread
  }
}

void Thread_MsgQueue2 (void *argument) {
  MSGQUEUE_OBJ_t msg;
  osStatus_t status;

  while (1) {
    ; // Insert thread code here...
    status = osMessageQueueGet(mid_MsgQueue, &msg, NULL, 0U);   // wait for message
    if (status == osOK) {
      ; // process data
    }
  }
}
```

Ključavnice (Mutual Exclusion)

Description

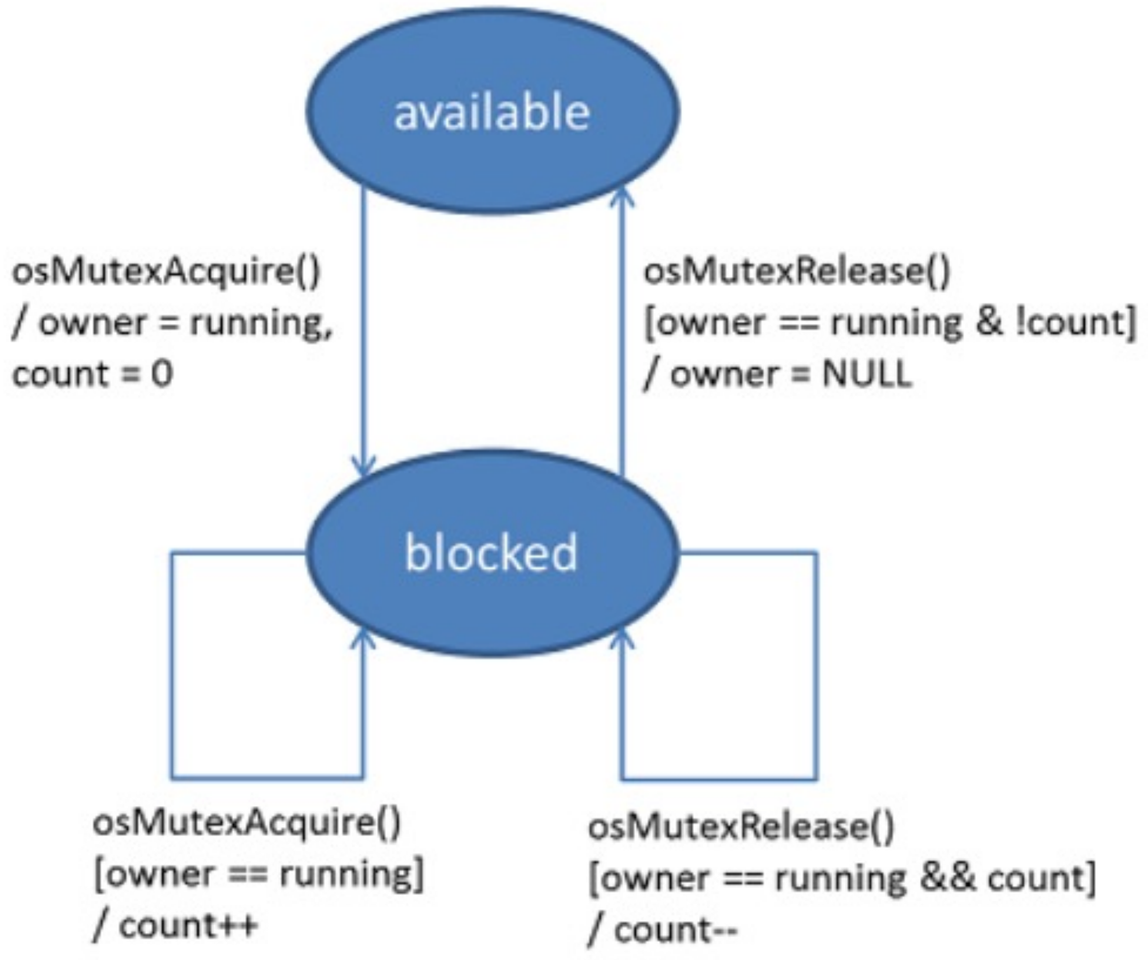
Mutual exclusion (widely known as **Mutex**) is used in various operating systems for resource management. Many resources in a microcontroller device can be used repeatedly, but only by one thread at a time (for example communication channels, memory, and files). Mutexes are used to protect access to a shared resource. A mutex is created and then passed between the threads (they can acquire and release the mutex).



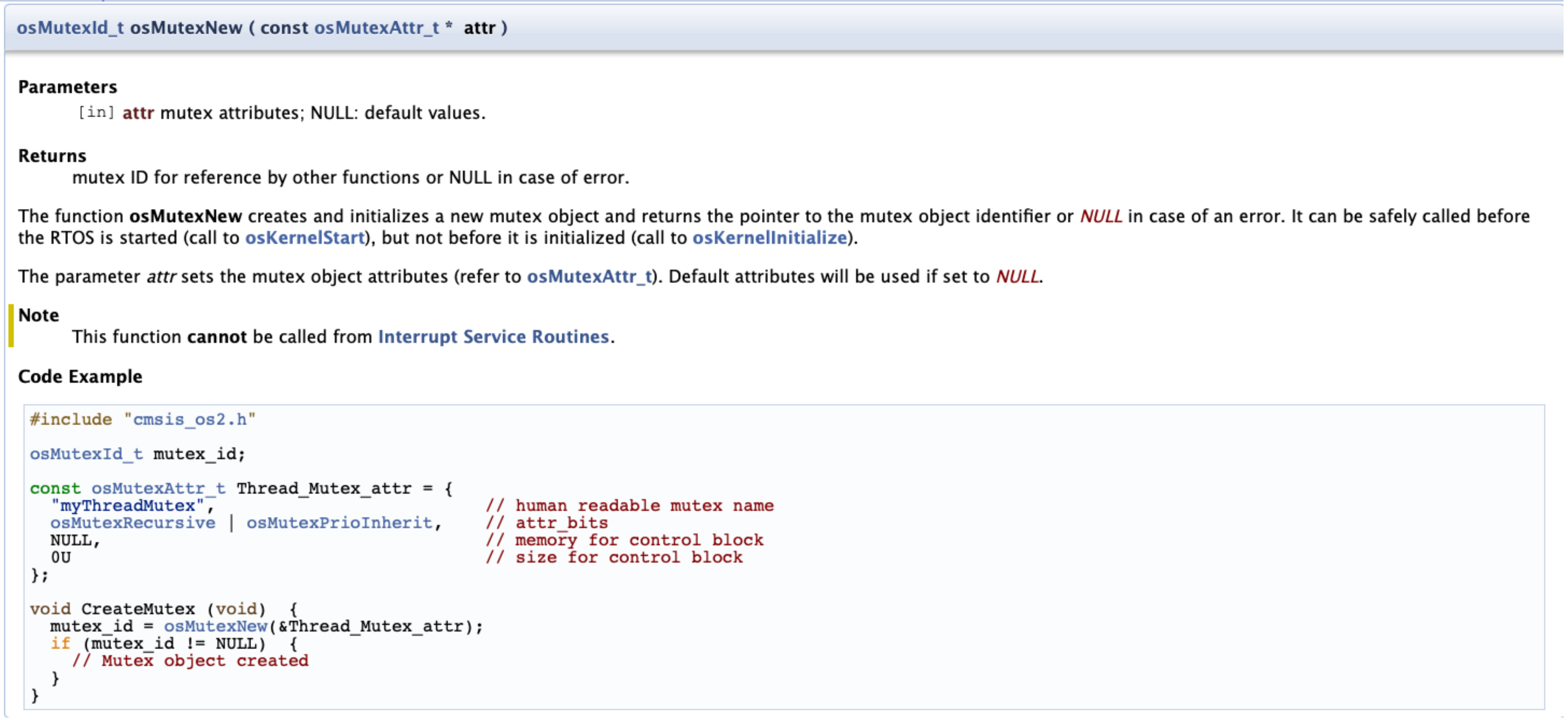
CMSIS-RTOS Mutex

A mutex is a special version of a **semaphore**. Like the semaphore, it is a container for tokens. But instead of being able to have multiple tokens, a mutex can only carry one (representing the resource). Thus, a mutex token is binary and bounded, i.e. it is either *available*, or *blocked* by a owning thread. The advantage of a mutex is that it introduces thread ownership. When a thread acquires a mutex and becomes its owner, subsequent mutex acquires from that thread will succeed immediately without any latency (if **osMutexRecursive** is specified). Thus, mutex acquires/releases can be nested.

Ključavnice (Mutual Exclusion)



Ključavnice (Mutual Exclusion)



osMutexId_t osMutexNew (const osMutexAttr_t * attr)

Parameters

[in] **attr** mutex attributes; NULL: default values.

Returns

mutex ID for reference by other functions or NULL in case of error.

The function **osMutexNew** creates and initializes a new mutex object and returns the pointer to the mutex object identifier or *NULL* in case of an error. It can be safely called before the RTOS is started (call to **osKernelStart**), but not before it is initialized (call to **osKernelInitialize**).

The parameter *attr* sets the mutex object attributes (refer to **osMutexAttr_t**). Default attributes will be used if set to *NULL*.

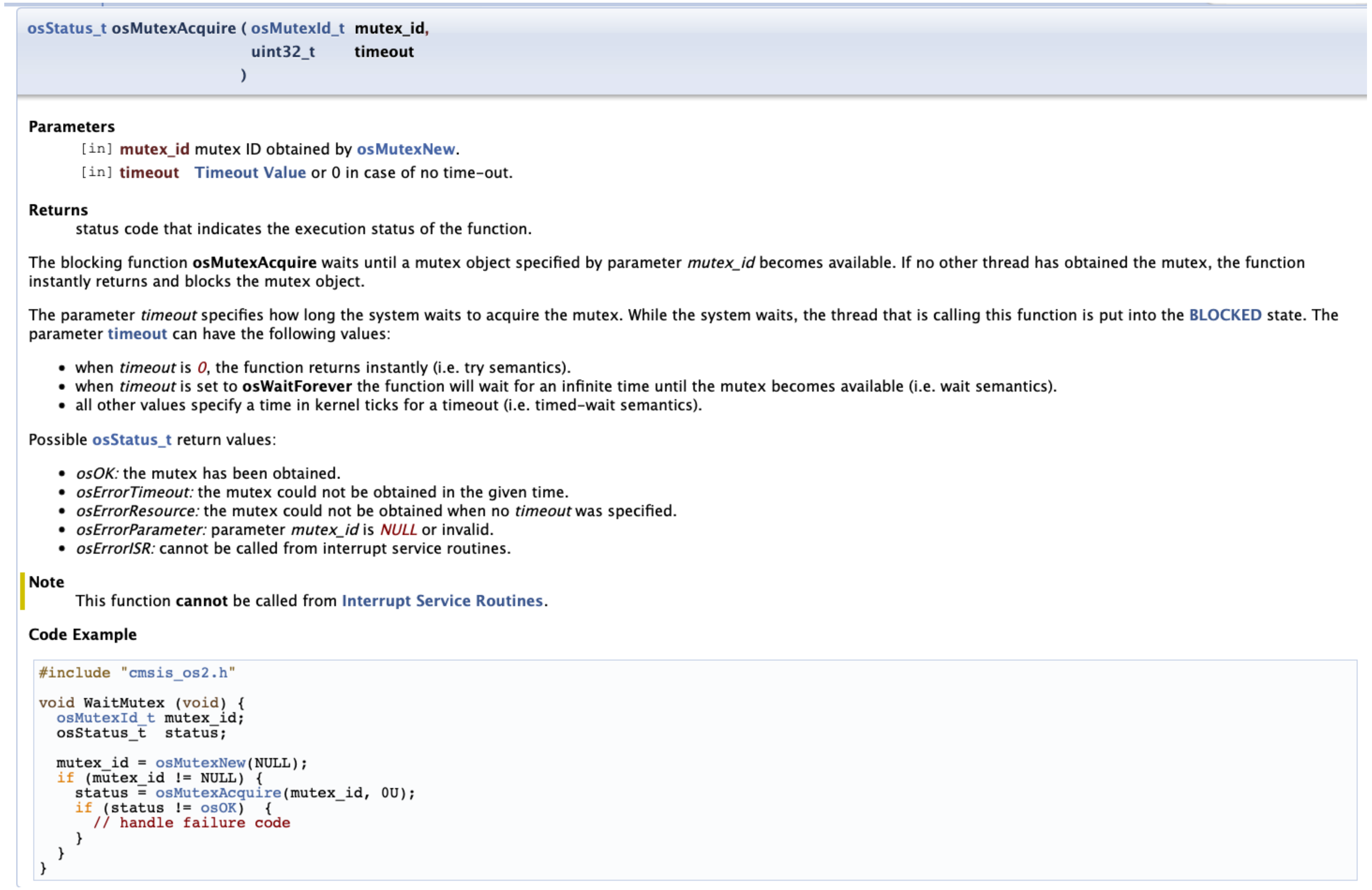
Note

This function **cannot** be called from **Interrupt Service Routines**.

Code Example

```
#include "cmsis_os2.h"

osMutexId_t mutex_id;

const osMutexAttr_t Thread_Mutex_attr = {
  "myThreadMutex",                           // human readable mutex name
  osMutexRecursive | osMutexPrioInherit,     // attr_bits
  NULL,                                      // memory for control block
  0U                                         // size for control block
};

void CreateMutex (void)  {
  mutex_id = osMutexNew(&Thread_Mutex_attr);
  if (mutex_id != NULL)  {
    // Mutex object created
  }
}
```

Ključavnice (Mutual Exclusion)



```
osStatus_t osMutexAcquire ( osMutexId_t  mutex_id,
                            uint32_t     timeout
                          )
```

Parameters

[in] **mutex_id** mutex ID obtained by osMutexNew.
[in] **timeout** Timeout Value or 0 in case of no time-out.

Returns

status code that indicates the execution status of the function.

The blocking function **osMutexAcquire** waits until a mutex object specified by parameter *mutex_id* becomes available. If no other thread has obtained the mutex, the function instantly returns and blocks the mutex object.

The parameter *timeout* specifies how long the system waits to acquire the mutex. While the system waits, the thread that is calling this function is put into the BLOCKED state. The parameter timeout can have the following values:

- when *timeout* is *0*, the function returns instantly (i.e. try semantics).
- when *timeout* is set to **osWaitForever** the function will wait for an infinite time until the mutex becomes available (i.e. wait semantics).
- all other values specify a time in kernel ticks for a timeout (i.e. timed-wait semantics).

Possible osStatus_t return values:

- *osOK:* the mutex has been obtained.
- *osErrorTimeout:* the mutex could not be obtained in the given time.
- *osErrorResource:* the mutex could not be obtained when no *timeout* was specified.
- *osErrorParameter:* parameter *mutex_id* is *NULL* or invalid.
- *osErrorISR:* cannot be called from interrupt service routines.

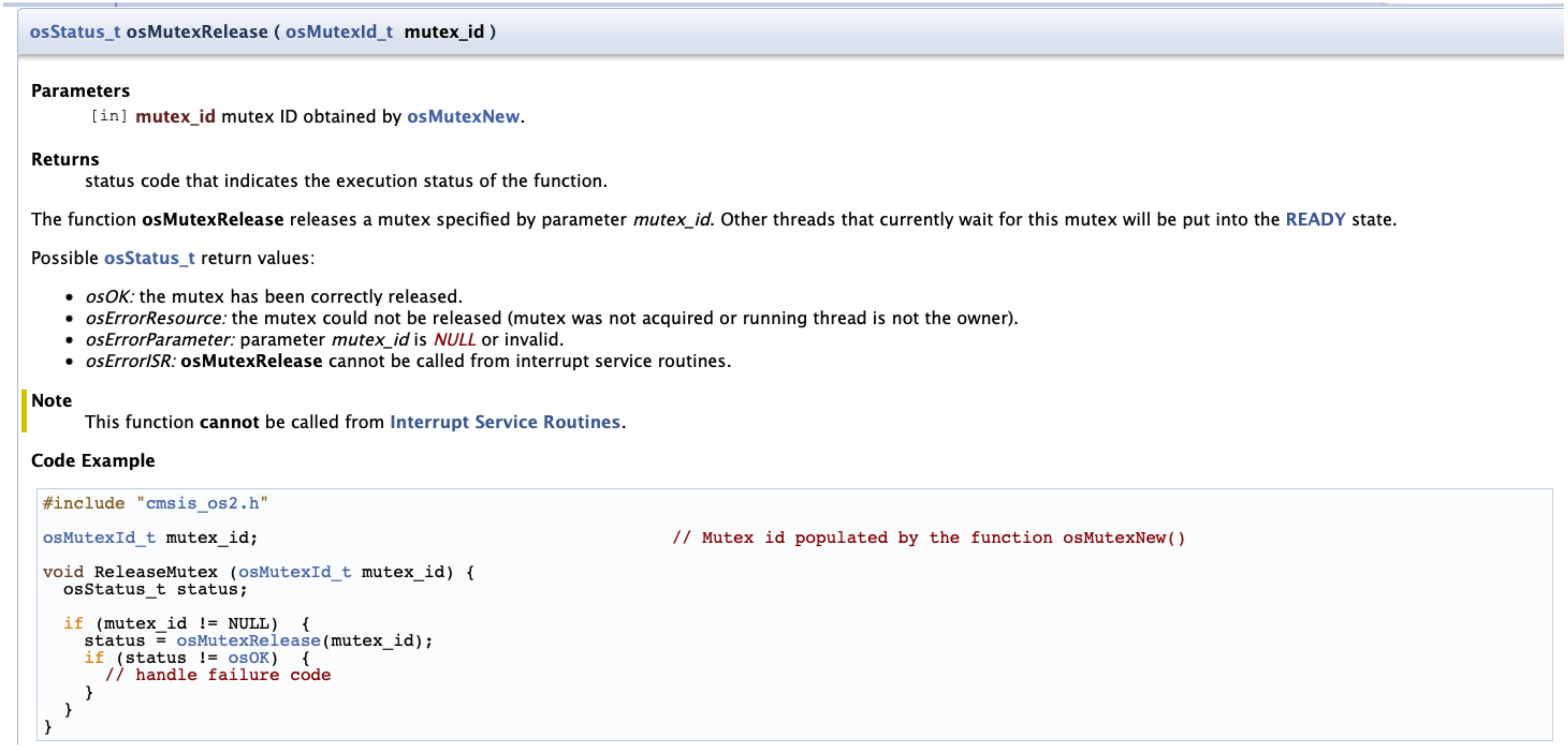
Note

This function **cannot** be called from Interrupt Service Routines.

Code Example

```
#include "cmsis_os2.h"

void WaitMutex (void) {
  osMutexId_t mutex_id;
  osStatus_t  status;

  mutex_id = osMutexNew(NULL);
  if (mutex_id != NULL) {
    status = osMutexAcquire(mutex_id, 0U);
    if (status != osOK)  {
      // handle failure code
    }
  }
}
```

Ključavnice (Mutual Exclusion)



osStatus_t osMutexRelease (osMutexId_t mutex_id)

Parameters

`[in]` **mutex_id** mutex ID obtained by **osMutexNew**.

Returns

status code that indicates the execution status of the function.

The function **osMutexRelease** releases a mutex specified by parameter *mutex_id*. Other threads that currently wait for this mutex will be put into the **READY** state.

Possible **osStatus_t** return values:

- *osOK:* the mutex has been correctly released.
- *osErrorResource:* the mutex could not be released (mutex was not acquired or running thread is not the owner).
- *osErrorParameter:* parameter *mutex_id* is *NULL* or invalid.
- *osErrorISR:* **osMutexRelease** cannot be called from interrupt service routines.

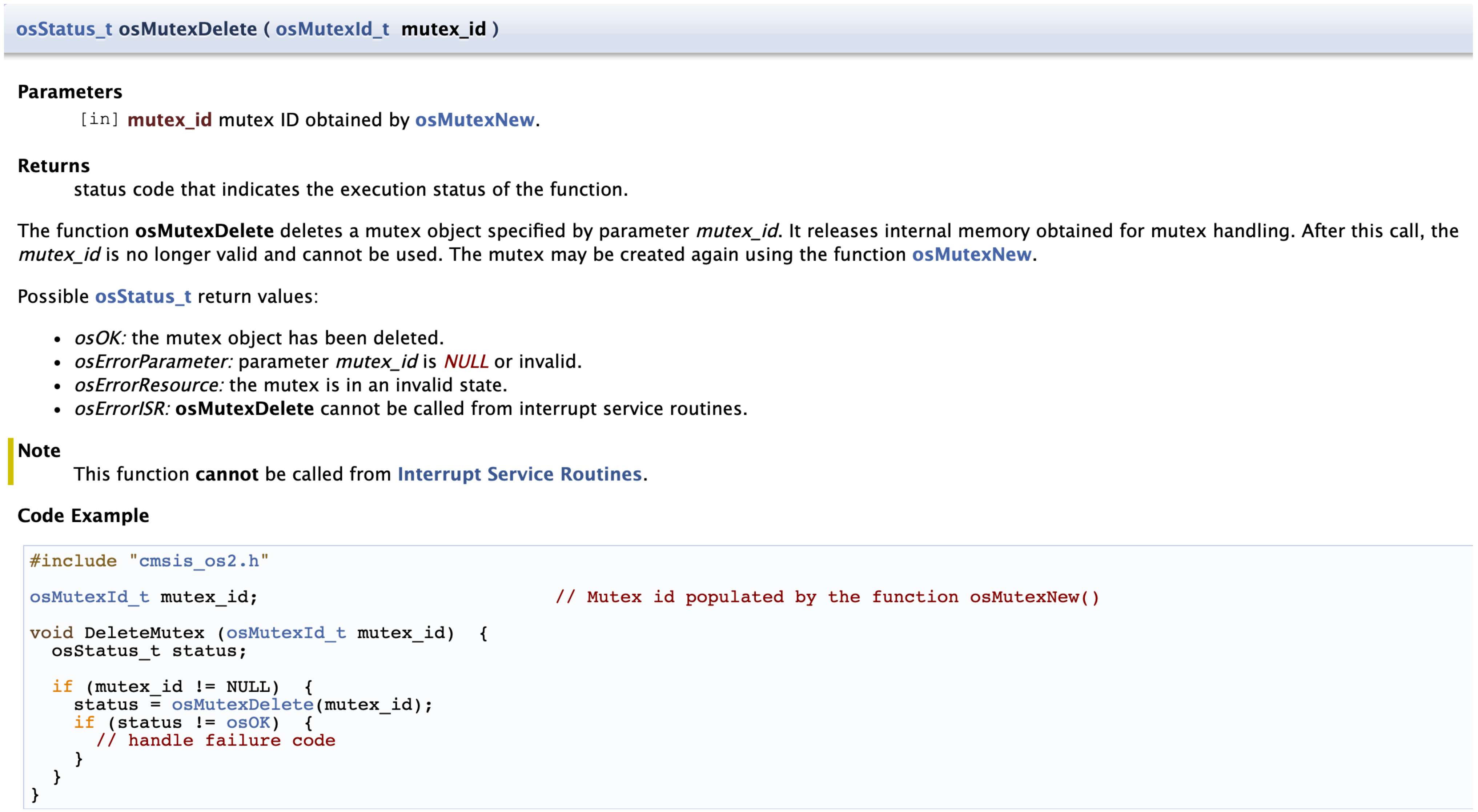
Note

This function **cannot** be called from **Interrupt Service Routines**.

Code Example

```
#include "cmsis_os2.h"

osMutexId_t mutex_id;                                    // Mutex id populated by the function osMutexNew()

void ReleaseMutex (osMutexId_t mutex_id) {
  osStatus_t status;

  if (mutex_id != NULL)  {
    status = osMutexRelease(mutex_id);
    if (status != osOK)  {
      // handle failure code
    }
  }
}
```

Ključavnice (Mutual Exclusion)



osStatus_t osMutexDelete (osMutexId_t mutex_id)

Parameters

[in] **mutex_id** mutex ID obtained by **osMutexNew**.

Returns

status code that indicates the execution status of the function.

The function **osMutexDelete** deletes a mutex object specified by parameter *mutex_id*. It releases internal memory obtained for mutex handling. After this call, the *mutex_id* is no longer valid and cannot be used. The mutex may be created again using the function **osMutexNew**.

Possible **osStatus_t** return values:

- *osOK:* the mutex object has been deleted.
- *osErrorParameter:* parameter *mutex_id* is *NULL* or invalid.
- *osErrorResource:* the mutex is in an invalid state.
- *osErrorISR:* **osMutexDelete** cannot be called from interrupt service routines.

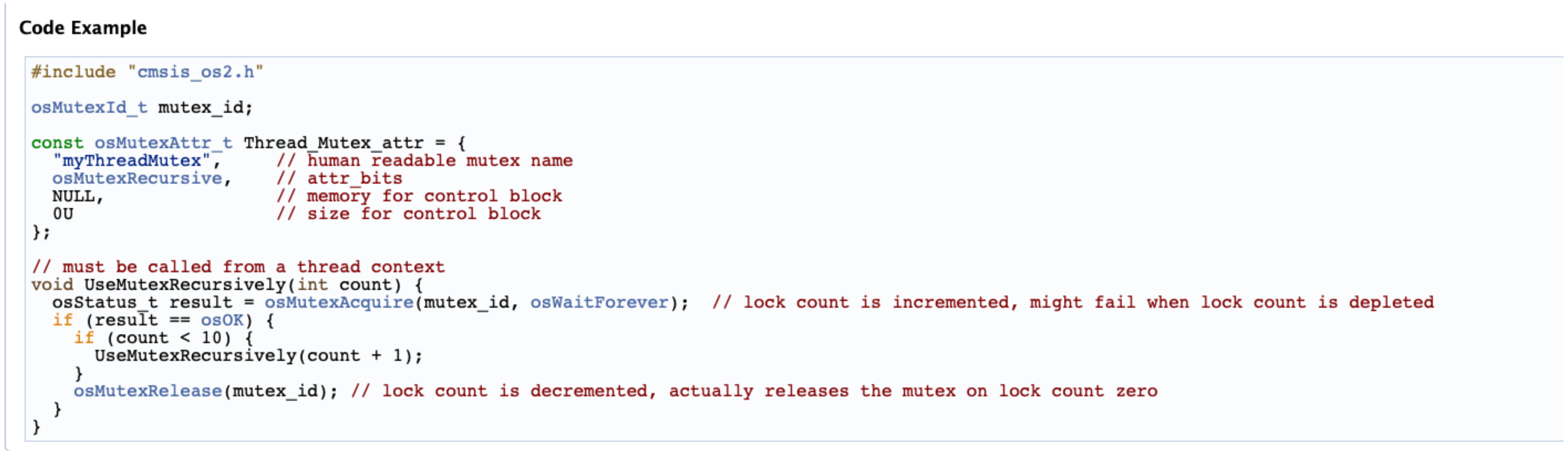
Note

This function **cannot** be called from **Interrupt Service Routines**.

Code Example

```
#include "cmsis_os2.h"

osMutexId_t mutex_id;                               // Mutex id populated by the function osMutexNew()

void DeleteMutex (osMutexId_t mutex_id)  {
  osStatus_t status;

  if (mutex_id != NULL)  {
    status = osMutexDelete(mutex_id);
    if (status != osOK)  {
      // handle failure code
    }
  }
}
```

Ključavnice (Mutual Exclusion)

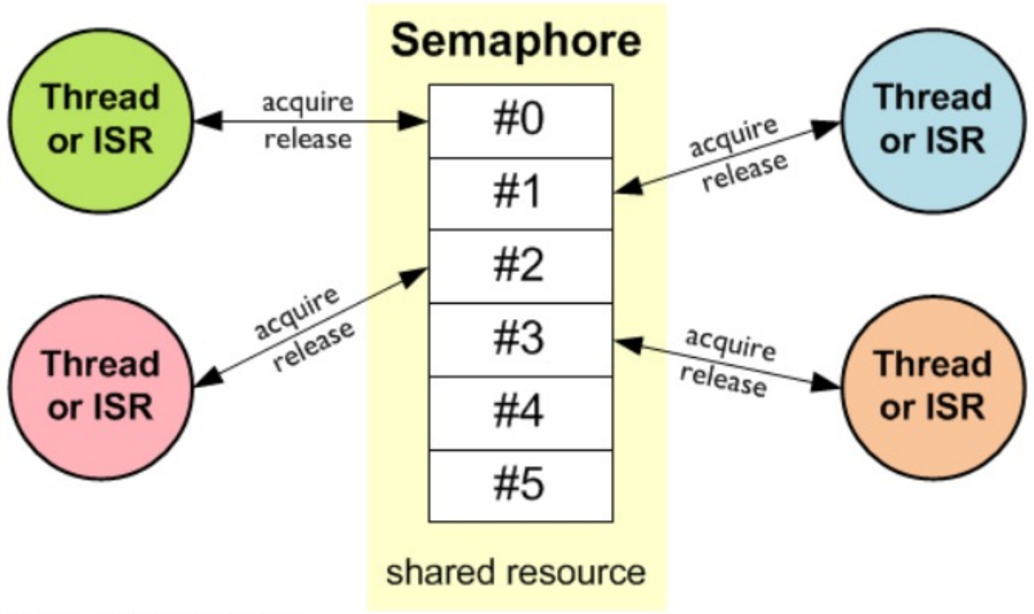


Code Example

```c
#include "cmsis_os2.h"

osMutexId_t mutex_id;

const osMutexAttr_t Thread_Mutex_attr = {
  "myThreadMutex",      // human readable mutex name
  osMutexRecursive,     // attr_bits
  NULL,                 // memory for control block
  0U                    // size for control block
};

// must be called from a thread context
void UseMutexRecursively(int count) {
  osStatus_t result = osMutexAcquire(mutex_id, osWaitForever);  // lock count is incremented, might fail when lock count is depleted
  if (result == osOK) {
    if (count < 10) {
      UseMutexRecursively(count + 1);
    }
    osMutexRelease(mutex_id); // lock count is decremented, actually releases the mutex on lock count zero
  }
}
```

Semaforji

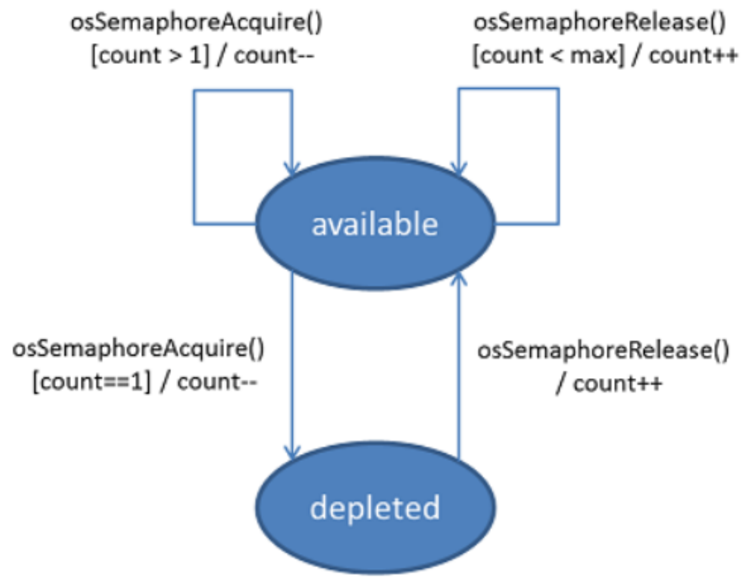
Semaphores are used to manage and protect access to shared resources. Semaphores are very similar to Mutexes. Whereas a Mutex permits just one thread to access a shared resource at a time, a semaphore can be used to permit a fixed number of threads/ISRs to access a pool of shared resources. Using semaphores, access to a group of identical peripherals can be managed (for example multiple DMA channels).



CMSIS-RTOS Semaphore

Semaforji

A semaphore object should be initialized to the maximum number of available tokens. This number of available resources is specified as parameter of the **osSemaphoreNew** function. Each time a semaphore token is obtained with **osSemaphoreAcquire** (in *available* state), the semaphore count is decremented. When the semaphore count is 0 (i.e. *depleted* state), no more semaphore tokens can be obtained. The thread/ISR that tries to obtain the semaphore token needs to wait until the next token is free. Semaphores are released with **osSemaphoreRelease** incrementing the semaphore count.


osSemaphoreAcquire() [count > 1] / count--
osSemaphoreRelease() [count < max] / count++
available
osSemaphoreAcquire() [count==1] / count--
osSemaphoreRelease() / count++
depleted

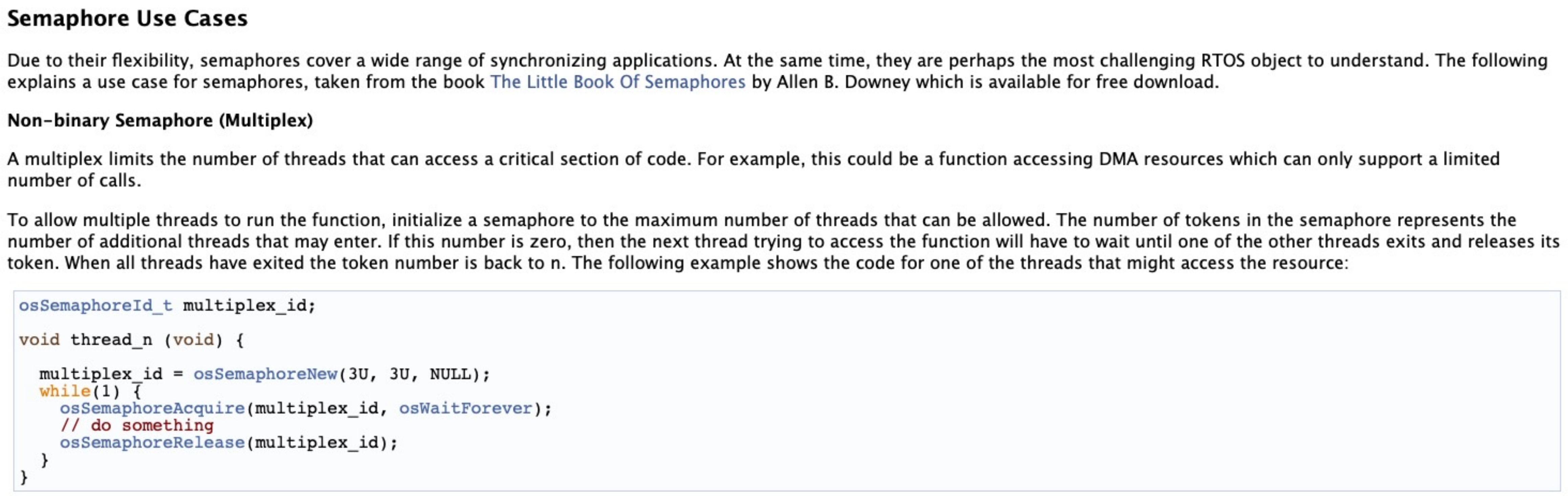
CMSIS-RTOS Semaphore States

Note

The functions **osSemaphoreAcquire**, **osSemaphoreGetCount**, and **osSemaphoreRelease** can be called from **Interrupt Service Routines**.

Refer to **Semaphore Configuration** for RTX5 configuration options.

Semaforji



Semaphore Use Cases

Due to their flexibility, semaphores cover a wide range of synchronizing applications. At the same time, they are perhaps the most challenging RTOS object to understand. The following explains a use case for semaphores, taken from the book The Little Book Of Semaphores by Allen B. Downey which is available for free download.

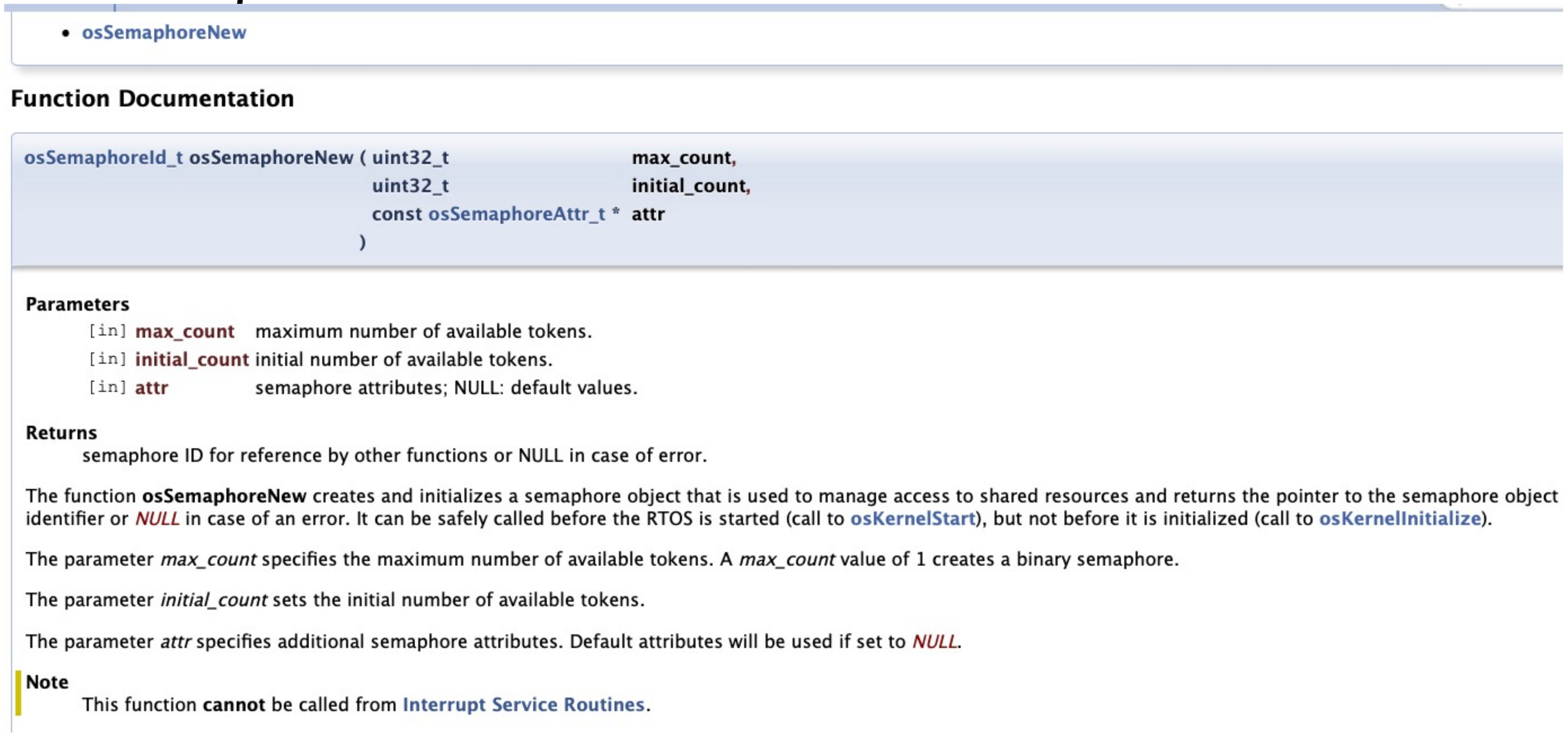
Non-binary Semaphore (Multiplex)

A multiplex limits the number of threads that can access a critical section of code. For example, this could be a function accessing DMA resources which can only support a limited number of calls.

To allow multiple threads to run the function, initialize a semaphore to the maximum number of threads that can be allowed. The number of tokens in the semaphore represents the number of additional threads that may enter. If this number is zero, then the next thread trying to access the function will have to wait until one of the other threads exits and releases its token. When all threads have exited the token number is back to n. The following example shows the code for one of the threads that might access the resource:

```
osSemaphoreId_t multiplex_id;

void thread_n (void) {

  multiplex_id = osSemaphoreNew(3U, 3U, NULL);
  while(1) {
    osSemaphoreAcquire(multiplex_id, osWaitForever);
    // do something
    osSemaphoreRelease(multiplex_id);
  }
}
```

Semaforji



- osSemaphoreNew

Function Documentation

```
osSemaphoreId_t osSemaphoreNew ( uint32_t                 max_count,
                                 uint32_t                 initial_count,
                                 const osSemaphoreAttr_t * attr
                               )
```

Parameters

[in]	**max_count**	maximum number of available tokens.
[in]	**initial_count**	initial number of available tokens.
[in]	**attr**	semaphore attributes; NULL: default values.

Returns

semaphore ID for reference by other functions or NULL in case of error.

The function **osSemaphoreNew** creates and initializes a semaphore object that is used to manage access to shared resources and returns the pointer to the semaphore object identifier or *NULL* in case of an error. It can be safely called before the RTOS is started (call to osKernelStart), but not before it is initialized (call to osKernelInitialize).

The parameter *max_count* specifies the maximum number of available tokens. A *max_count* value of 1 creates a binary semaphore.

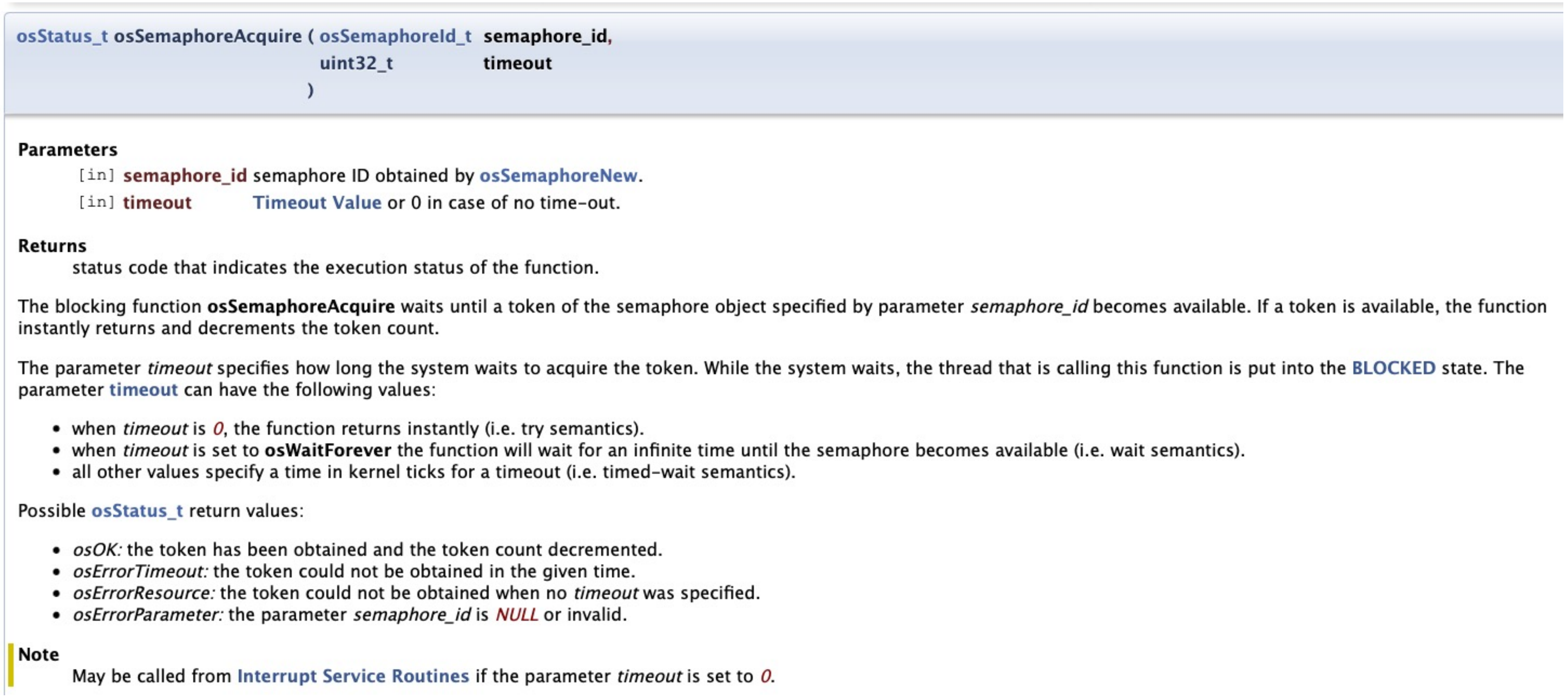
The parameter *initial_count* sets the initial number of available tokens.

The parameter *attr* specifies additional semaphore attributes. Default attributes will be used if set to *NULL*.

Note

This function **cannot** be called from Interrupt Service Routines.

Semaforji



```
osStatus_t osSemaphoreAcquire ( osSemaphoreId_t  semaphore_id,
                                uint32_t         timeout
                              )
```

Parameters

[in]	**semaphore_id**	semaphore ID obtained by **osSemaphoreNew**.
[in]	**timeout**	**Timeout Value** or 0 in case of no time-out.

Returns

status code that indicates the execution status of the function.

The blocking function **osSemaphoreAcquire** waits until a token of the semaphore object specified by parameter *semaphore_id* becomes available. If a token is available, the function instantly returns and decrements the token count.

The parameter *timeout* specifies how long the system waits to acquire the token. While the system waits, the thread that is calling this function is put into the **BLOCKED** state. The parameter **timeout** can have the following values:

- when *timeout* is *0*, the function returns instantly (i.e. try semantics).
- when *timeout* is set to **osWaitForever** the function will wait for an infinite time until the semaphore becomes available (i.e. wait semantics).
- all other values specify a time in kernel ticks for a timeout (i.e. timed-wait semantics).

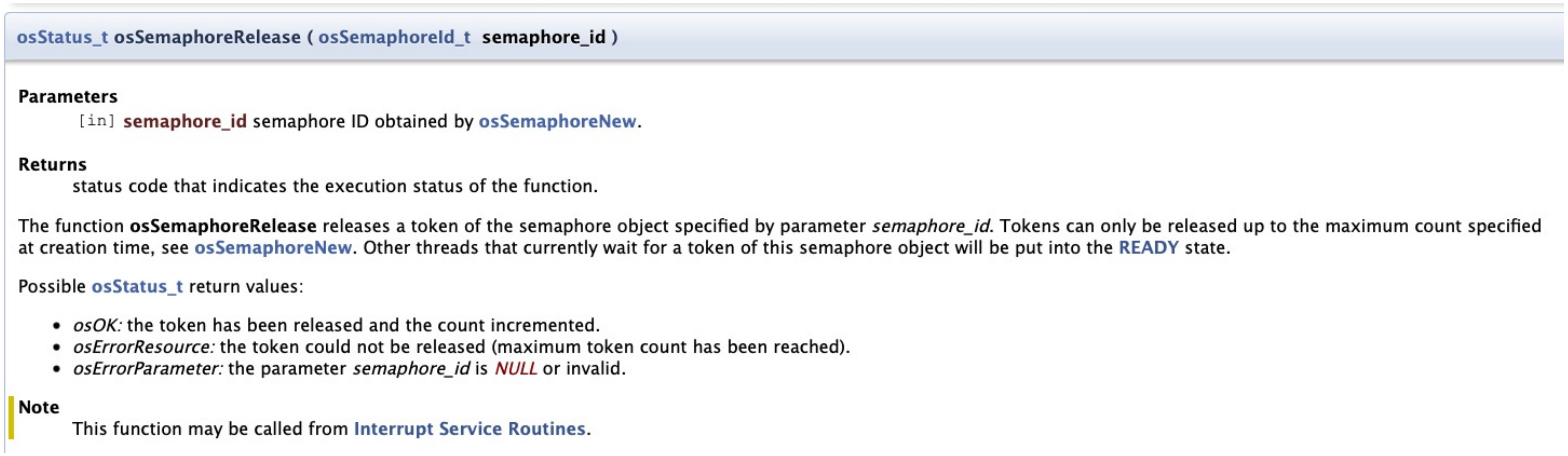
Possible **osStatus_t** return values:

- *osOK:* the token has been obtained and the token count decremented.
- *osErrorTimeout:* the token could not be obtained in the given time.
- *osErrorResource:* the token could not be obtained when no *timeout* was specified.
- *osErrorParameter:* the parameter *semaphore_id* is *NULL* or invalid.

Note

May be called from **Interrupt Service Routines** if the parameter *timeout* is set to *0*.

Semaforji



osStatus_t osSemaphoreRelease (osSemaphoreId_t semaphore_id)

Parameters

[in] **semaphore_id** semaphore ID obtained by **osSemaphoreNew**.

Returns

status code that indicates the execution status of the function.

The function **osSemaphoreRelease** releases a token of the semaphore object specified by parameter *semaphore_id*. Tokens can only be released up to the maximum count specified at creation time, see **osSemaphoreNew**. Other threads that currently wait for a token of this semaphore object will be put into the **READY** state.

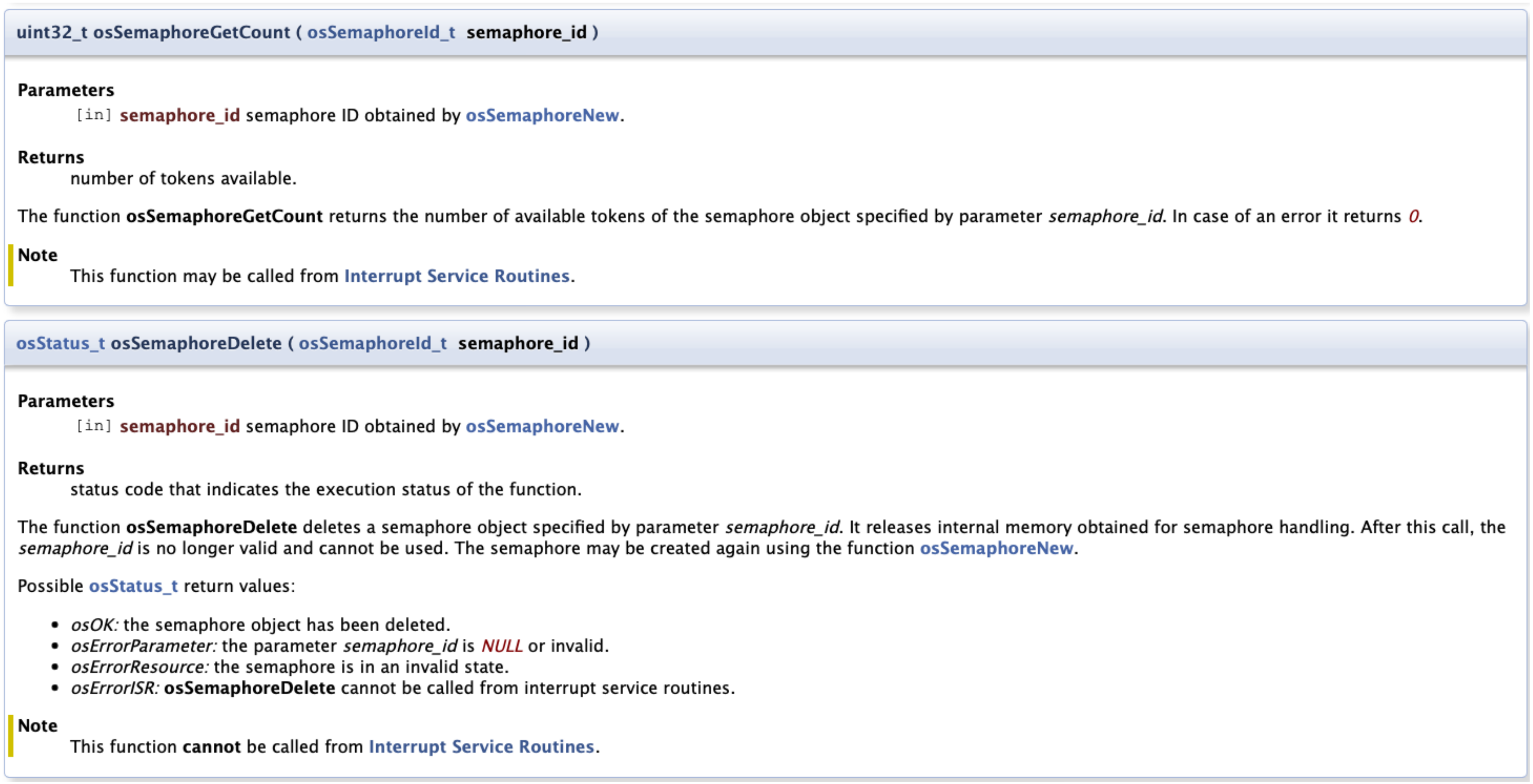
Possible **osStatus_t** return values:

- *osOK:* the token has been released and the count incremented.
- *osErrorResource:* the token could not be released (maximum token count has been reached).
- *osErrorParameter:* the parameter *semaphore_id* is *NULL* or invalid.

Note

This function may be called from **Interrupt Service Routines**.

Semaforji



uint32_t osSemaphoreGetCount (osSemaphoreId_t semaphore_id)

Parameters

`[in]` **semaphore_id** semaphore ID obtained by **osSemaphoreNew**.

Returns

number of tokens available.

The function **osSemaphoreGetCount** returns the number of available tokens of the semaphore object specified by parameter *semaphore_id*. In case of an error it returns *0*.

Note

This function may be called from **Interrupt Service Routines**.

osStatus_t osSemaphoreDelete (osSemaphoreId_t semaphore_id)

Parameters

`[in]` **semaphore_id** semaphore ID obtained by **osSemaphoreNew**.

Returns

status code that indicates the execution status of the function.

The function **osSemaphoreDelete** deletes a semaphore object specified by parameter *semaphore_id*. It releases internal memory obtained for semaphore handling. After this call, the *semaphore_id* is no longer valid and cannot be used. The semaphore may be created again using the function **osSemaphoreNew**.

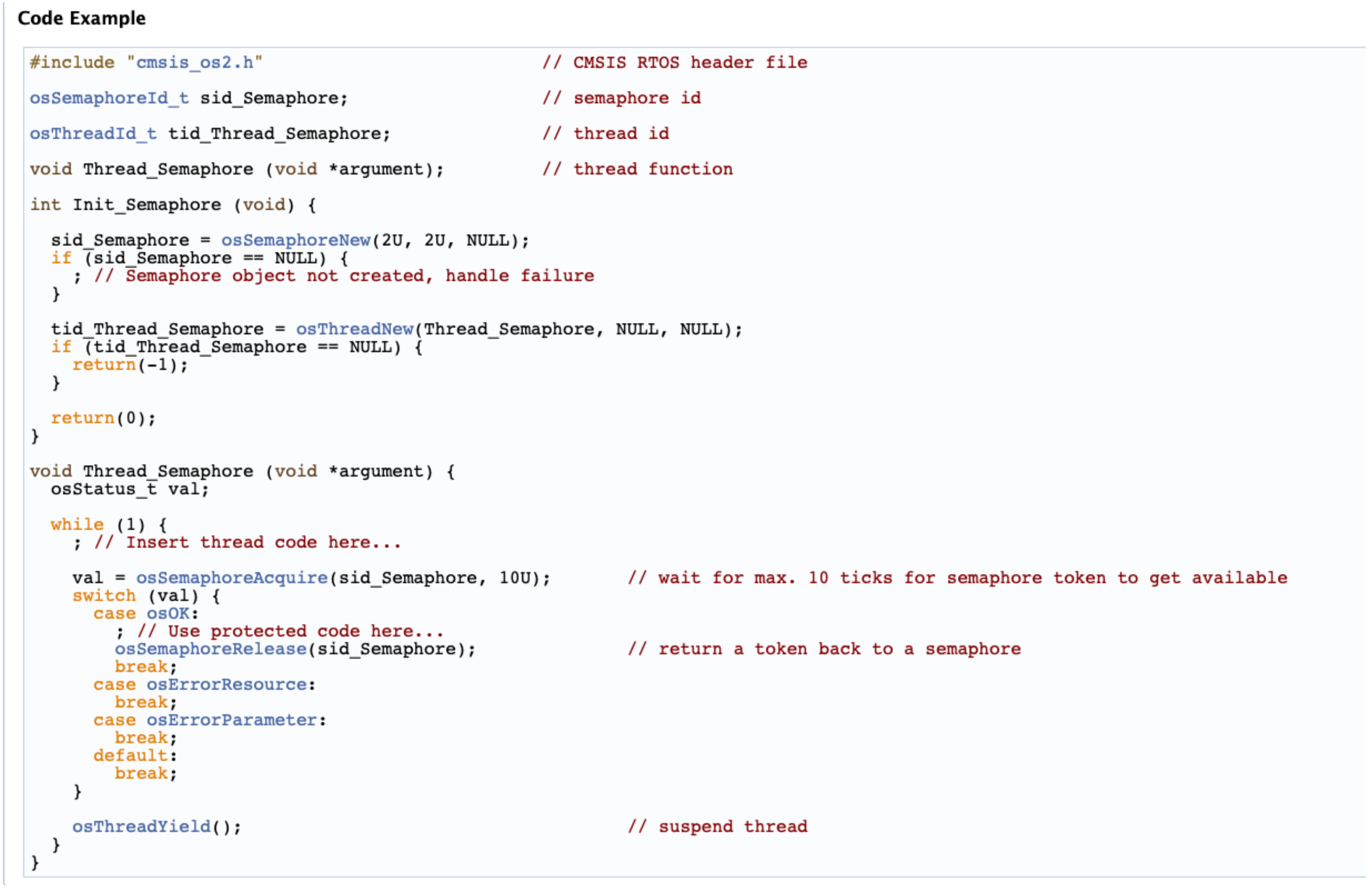
Possible **osStatus_t** return values:

- *osOK:* the semaphore object has been deleted.
- *osErrorParameter:* the parameter *semaphore_id* is *NULL* or invalid.
- *osErrorResource:* the semaphore is in an invalid state.
- *osErrorISR:* **osSemaphoreDelete** cannot be called from interrupt service routines.

Note

This function **cannot** be called from **Interrupt Service Routines**.

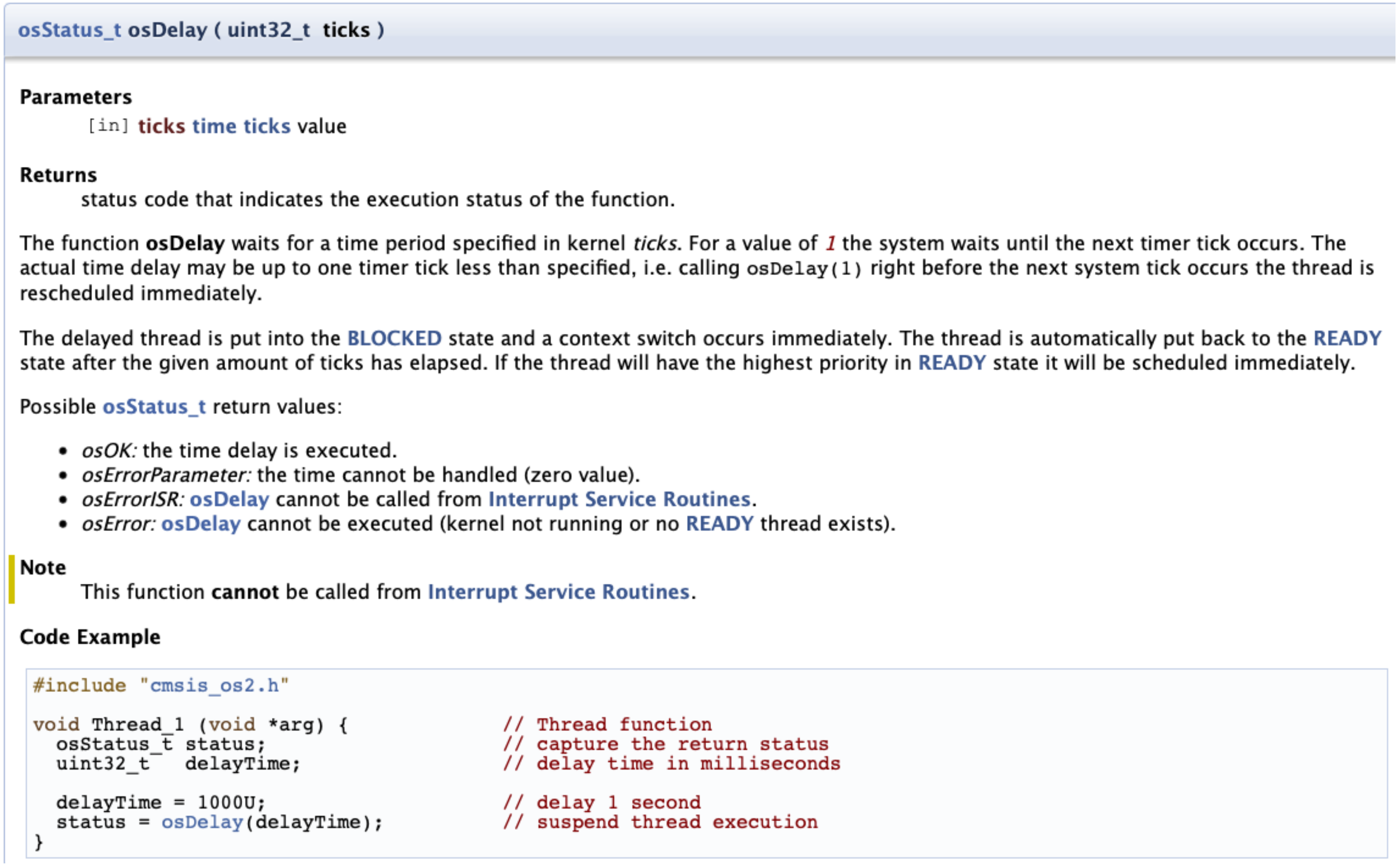
Semaforji



Code Example

```c
#include "cmsis_os2.h"                          // CMSIS RTOS header file

osSemaphoreId_t sid_Semaphore;                  // semaphore id

osThreadId_t tid_Thread_Semaphore;              // thread id

void Thread_Semaphore (void *argument);         // thread function

int Init_Semaphore (void) {

  sid_Semaphore = osSemaphoreNew(2U, 2U, NULL);
  if (sid_Semaphore == NULL) {
    ; // Semaphore object not created, handle failure
  }

  tid_Thread_Semaphore = osThreadNew(Thread_Semaphore, NULL, NULL);
  if (tid_Thread_Semaphore == NULL) {
    return(-1);
  }

  return(0);
}

void Thread_Semaphore (void *argument) {
  osStatus_t val;

  while (1) {
    ; // Insert thread code here...

    val = osSemaphoreAcquire(sid_Semaphore, 10U);       // wait for max. 10 ticks for semaphore token to get available
    switch (val) {
      case osOK:
        ; // Use protected code here...
        osSemaphoreRelease(sid_Semaphore);              // return a token back to a semaphore
        break;
      case osErrorResource:
        break;
      case osErrorParameter:
        break;
      default:
        break;
    }

    osThreadYield();                                    // suspend thread
  }
}
```

Zakasnitve



osStatus_t osDelay (uint32_t ticks)

Parameters

[in] **ticks** **time ticks** value

Returns

status code that indicates the execution status of the function.

The function **osDelay** waits for a time period specified in kernel *ticks*. For a value of *1* the system waits until the next timer tick occurs. The actual time delay may be up to one timer tick less than specified, i.e. calling `osDelay(1)` right before the next system tick occurs the thread is rescheduled immediately.

The delayed thread is put into the **BLOCKED** state and a context switch occurs immediately. The thread is automatically put back to the **READY** state after the given amount of ticks has elapsed. If the thread will have the highest priority in **READY** state it will be scheduled immediately.

Possible **osStatus_t** return values:

- *osOK:* the time delay is executed.
- *osErrorParameter:* the time cannot be handled (zero value).
- *osErrorISR:* **osDelay** cannot be called from **Interrupt Service Routines**.
- *osError:* **osDelay** cannot be executed (kernel not running or no **READY** thread exists).

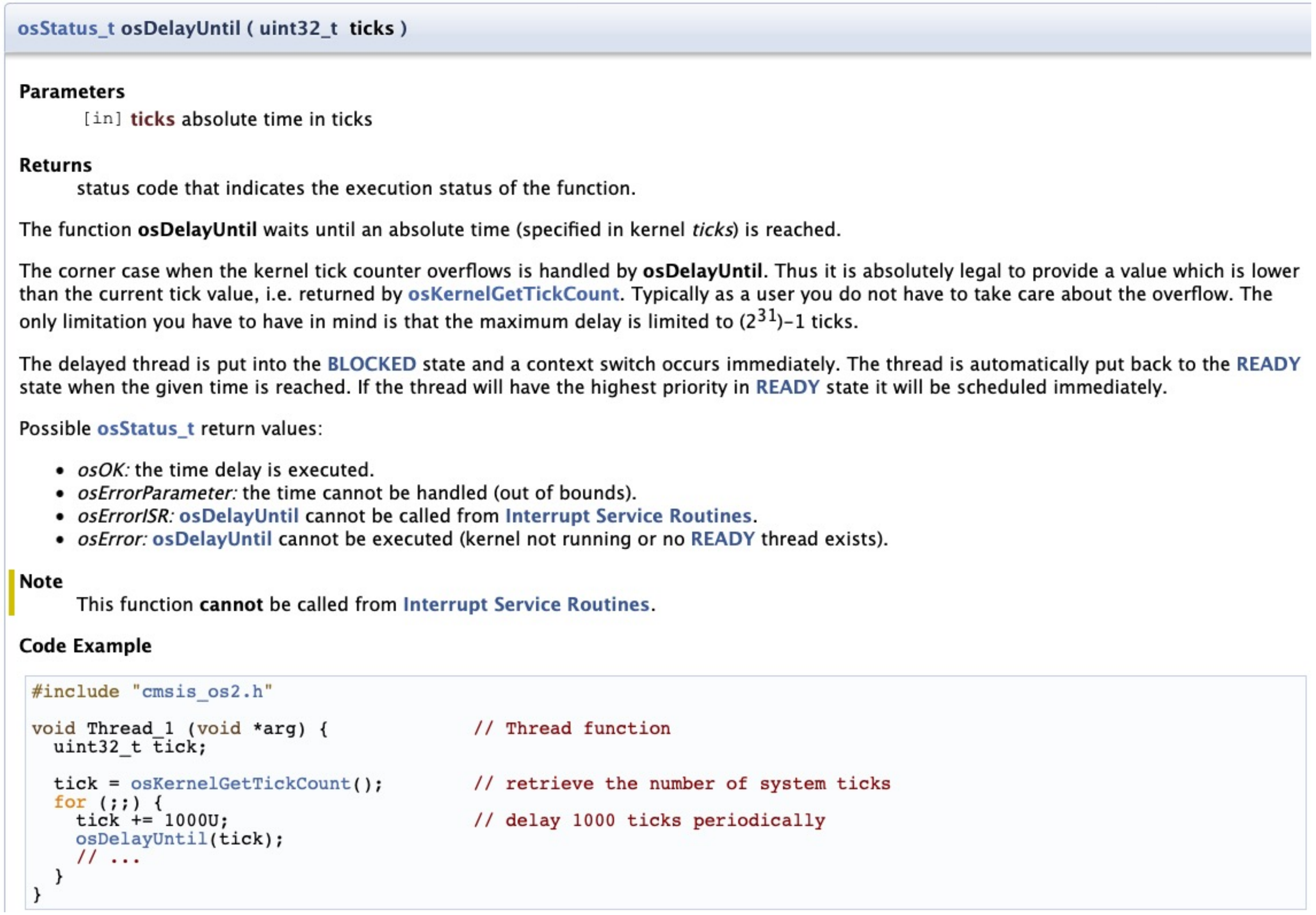
Note

This function **cannot** be called from **Interrupt Service Routines**.

Code Example

```
#include "cmsis_os2.h"

void Thread_1 (void *arg) {                // Thread function
  osStatus_t status;                       // capture the return status
  uint32_t   delayTime;                    // delay time in milliseconds

  delayTime = 1000U;                       // delay 1 second
  status = osDelay(delayTime);             // suspend thread execution
}
```

Zakasnitve



osStatus_t osDelayUntil (uint32_t ticks)

Parameters

[in] **ticks** absolute time in ticks

Returns

status code that indicates the execution status of the function.

The function **osDelayUntil** waits until an absolute time (specified in kernel *ticks*) is reached.

The corner case when the kernel tick counter overflows is handled by **osDelayUntil**. Thus it is absolutely legal to provide a value which is lower than the current tick value, i.e. returned by osKernelGetTickCount. Typically as a user you do not have to take care about the overflow. The only limitation you have to have in mind is that the maximum delay is limited to $(2^{31})-1$ ticks.

The delayed thread is put into the BLOCKED state and a context switch occurs immediately. The thread is automatically put back to the READY state when the given time is reached. If the thread will have the highest priority in READY state it will be scheduled immediately.

Possible osStatus_t return values:

- *osOK:* the time delay is executed.
- *osErrorParameter:* the time cannot be handled (out of bounds).
- *osErrorISR:* osDelayUntil cannot be called from Interrupt Service Routines.
- *osError:* osDelayUntil cannot be executed (kernel not running or no READY thread exists).

Note

This function **cannot** be called from Interrupt Service Routines.

Code Example

```
#include "cmsis_os2.h"

void Thread_1 (void *arg) {                // Thread function
  uint32_t tick;

  tick = osKernelGetTickCount();           // retrieve the number of system ticks
  for (;;) {
    tick += 1000U;                         // delay 1000 ticks periodically
    osDelayUntil(tick);
    // ...
  }
}
```

Virtualni časovniki

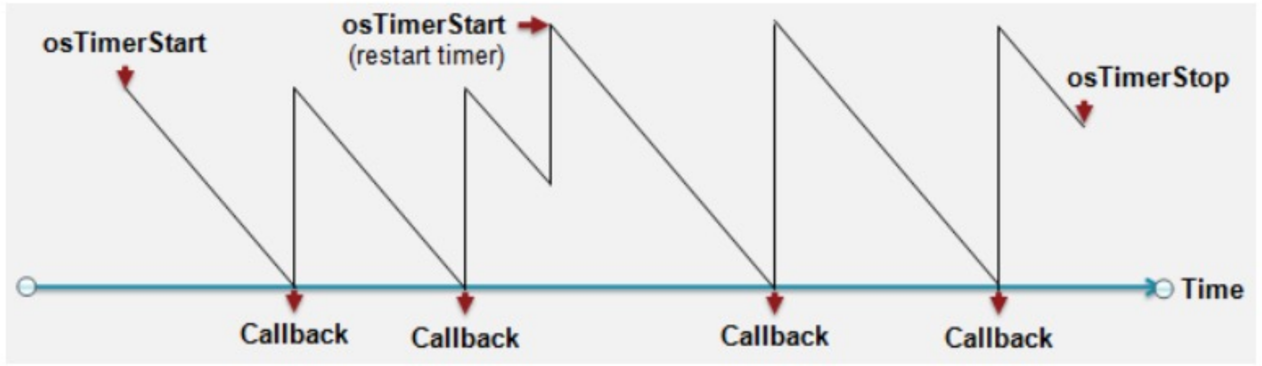
In addition to the Generic Wait Functions CMSIS-RTOS also supports virtual timer objects. These timer objects can trigger the execution of a function (not threads). When a timer expires, a callback function is executed to run associated code with the timer. Each timer can be configured as a one-shot or a periodic timer. A periodic timer repeats its operation until it is deleted or stopped. All timers can be started, restarted, or stopped.

Note

RTX handles Timers in the thread **osRtxTimerThread**. Callback functions run under control of this thread and may use other CMSIS-RTOS API calls. The **osRtxTimerThread** is configured in Timer Configuration.

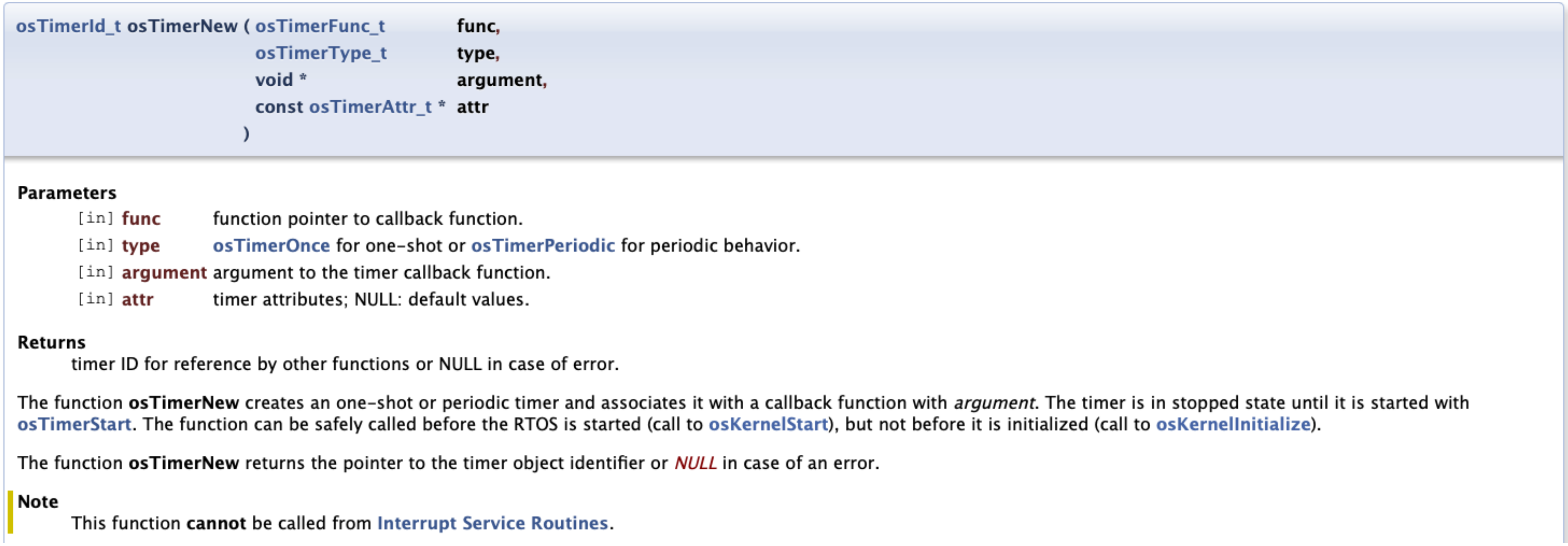
Timer management functions cannot be called from Interrupt Service Routines.

The figure below shows the behavior of a periodic timer. For one-shot timers, the timer stops after execution of the callback function.



Behavior of a Periodic Timer

Virtualni časovniki



```
osTimerId_t osTimerNew ( osTimerFunc_t           func,
                         osTimerType_t           type,
                         void *                  argument,
                         const osTimerAttr_t *   attr
                       )
```

Parameters

[in]	**func**	function pointer to callback function.
[in]	**type**	osTimerOnce for one-shot or osTimerPeriodic for periodic behavior.
[in]	**argument**	argument to the timer callback function.
[in]	**attr**	timer attributes; NULL: default values.

Returns

timer ID for reference by other functions or NULL in case of error.

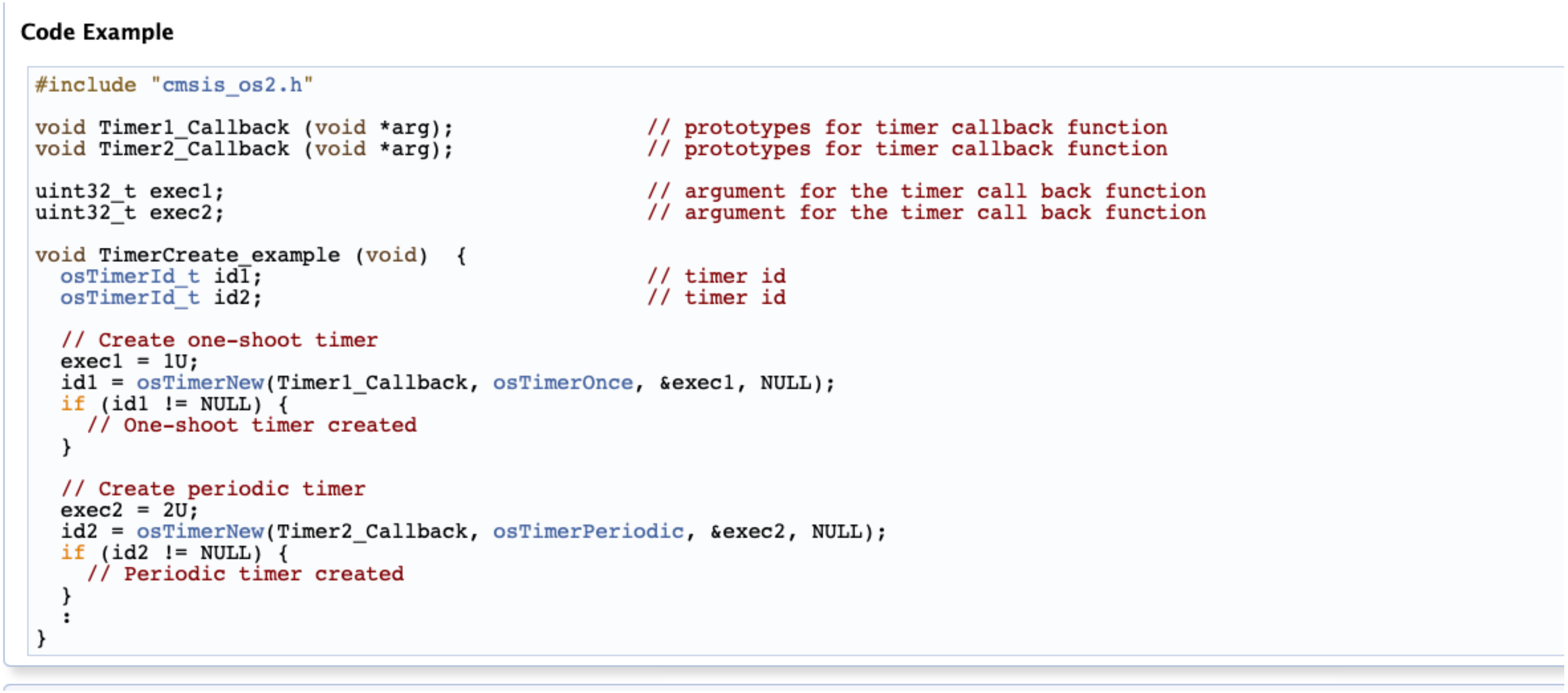
The function **osTimerNew** creates an one-shot or periodic timer and associates it with a callback function with *argument*. The timer is in stopped state until it is started with osTimerStart. The function can be safely called before the RTOS is started (call to osKernelStart), but not before it is initialized (call to osKernelInitialize).

The function **osTimerNew** returns the pointer to the timer object identifier or *NULL* in case of an error.

Note

This function **cannot** be called from Interrupt Service Routines.

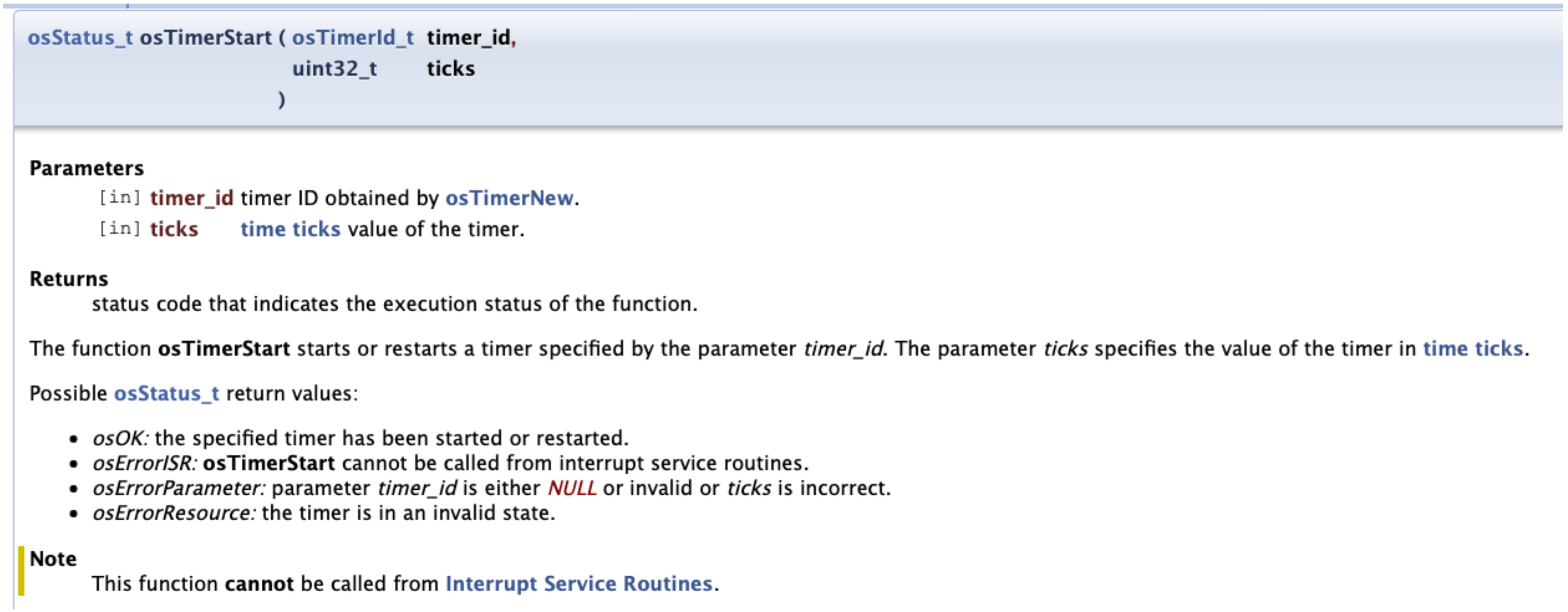
Virtualni časovniki



Code Example

```c
#include "cmsis_os2.h"

void Timer1_Callback (void *arg);                        // prototypes for timer callback function
void Timer2_Callback (void *arg);                        // prototypes for timer callback function

uint32_t exec1;                                          // argument for the timer call back function
uint32_t exec2;                                          // argument for the timer call back function

void TimerCreate_example (void)  {
  osTimerId_t id1;                                       // timer id
  osTimerId_t id2;                                       // timer id

  // Create one-shoot timer
  exec1 = 1U;
  id1 = osTimerNew(Timer1_Callback, osTimerOnce, &exec1, NULL);
  if (id1 != NULL) {
    // One-shoot timer created
  }

  // Create periodic timer
  exec2 = 2U;
  id2 = osTimerNew(Timer2_Callback, osTimerPeriodic, &exec2, NULL);
  if (id2 != NULL) {
    // Periodic timer created
  }
  :
}
```

Virtualni časovniki



```
osStatus_t osTimerStart ( osTimerId_t  timer_id,
                          uint32_t     ticks
                        )
```

Parameters

[in] **timer_id** timer ID obtained by **osTimerNew**.
[in] **ticks** **time ticks** value of the timer.

Returns

status code that indicates the execution status of the function.

The function **osTimerStart** starts or restarts a timer specified by the parameter *timer_id*. The parameter *ticks* specifies the value of the timer in **time ticks**.

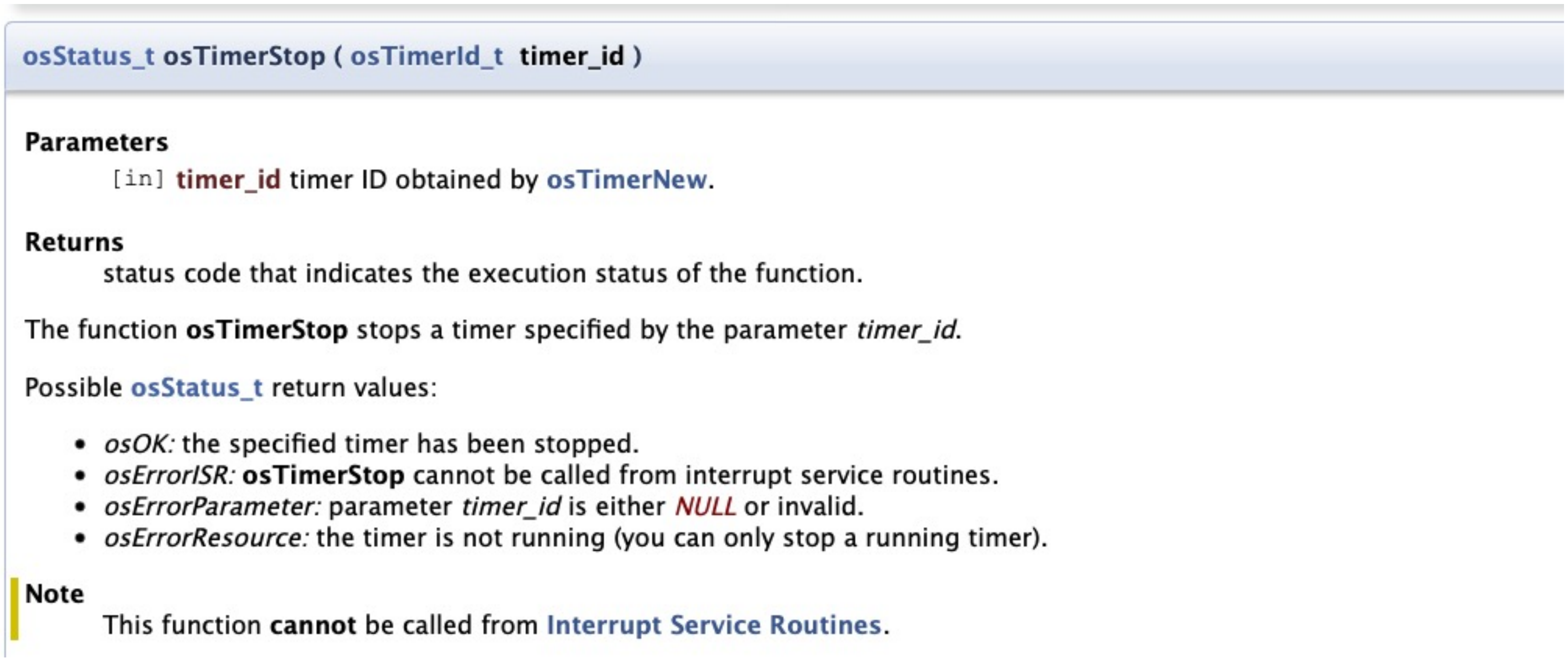
Possible **osStatus_t** return values:

- *osOK:* the specified timer has been started or restarted.
- *osErrorISR:* **osTimerStart** cannot be called from interrupt service routines.
- *osErrorParameter:* parameter *timer_id* is either *NULL* or invalid or *ticks* is incorrect.
- *osErrorResource:* the timer is in an invalid state.

Note

This function **cannot** be called from **Interrupt Service Routines**.

Virtualni časovniki



osStatus_t osTimerStop (osTimerId_t timer_id)

Parameters

`[in]` **timer_id** timer ID obtained by **osTimerNew**.

Returns

status code that indicates the execution status of the function.

The function **osTimerStop** stops a timer specified by the parameter *timer_id*.

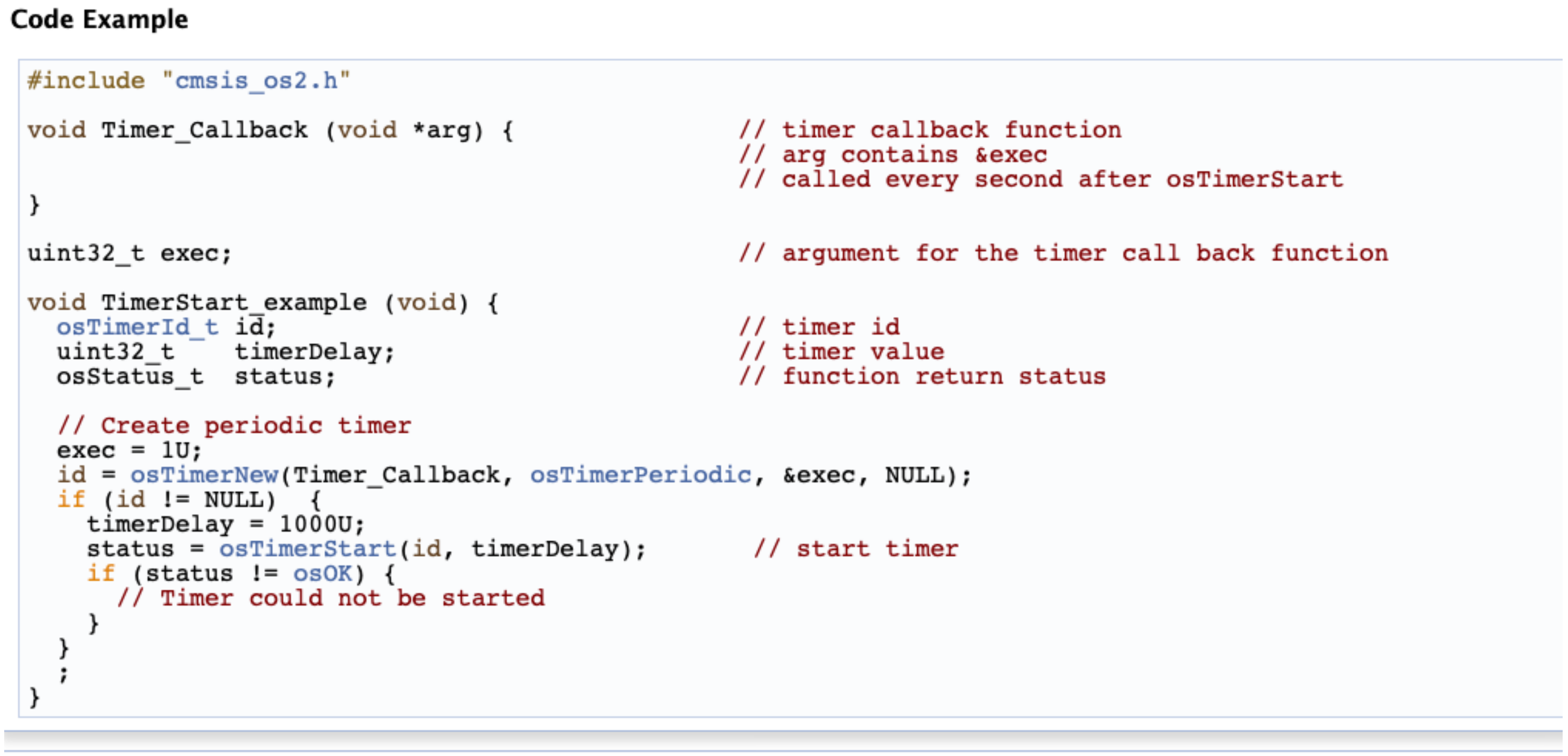
Possible **osStatus_t** return values:

- *osOK:* the specified timer has been stopped.
- *osErrorISR:* **osTimerStop** cannot be called from interrupt service routines.
- *osErrorParameter:* parameter *timer_id* is either *NULL* or invalid.
- *osErrorResource:* the timer is not running (you can only stop a running timer).

Note

This function **cannot** be called from **Interrupt Service Routines**.

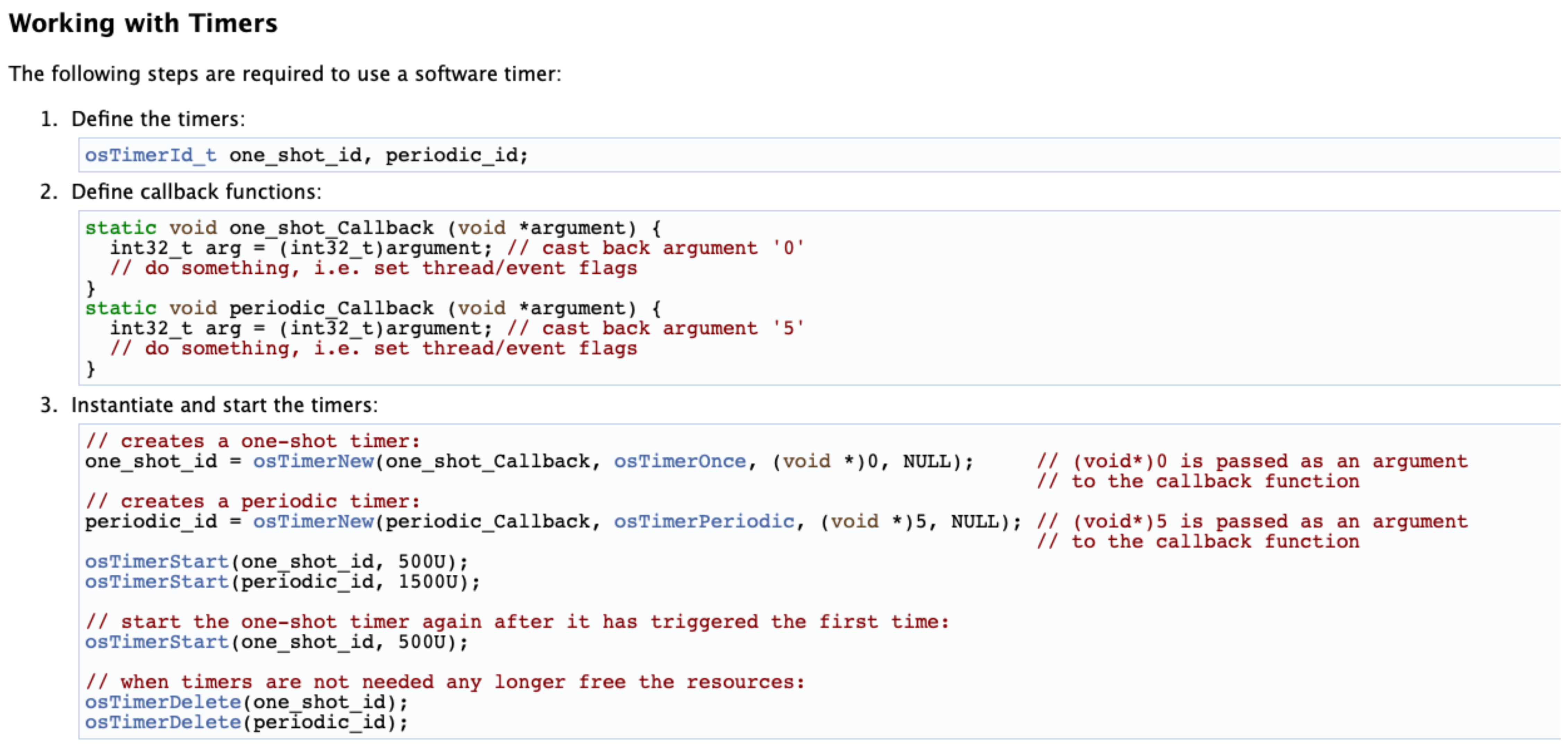
Virtualni časovniki



Code Example

```c
#include "cmsis_os2.h"

void Timer_Callback (void *arg) {               // timer callback function
                                                // arg contains &exec
                                                // called every second after osTimerStart
}

uint32_t exec;                                  // argument for the timer call back function

void TimerStart_example (void) {
  osTimerId_t id;                               // timer id
  uint32_t    timerDelay;                       // timer value
  osStatus_t  status;                           // function return status

  // Create periodic timer
  exec = 1U;
  id = osTimerNew(Timer_Callback, osTimerPeriodic, &exec, NULL);
  if (id != NULL)  {
    timerDelay = 1000U;
    status = osTimerStart(id, timerDelay);       // start timer
    if (status != osOK) {
      // Timer could not be started
    }
  }
  ;
}
```

Virtualni časovniki



Working with Timers

The following steps are required to use a software timer:

1. Define the timers:

```
osTimerId_t one_shot_id, periodic_id;
```

2. Define callback functions:

```
static void one_shot_Callback (void *argument) {
  int32_t arg = (int32_t)argument; // cast back argument '0'
  // do something, i.e. set thread/event flags
}
static void periodic_Callback (void *argument) {
  int32_t arg = (int32_t)argument; // cast back argument '5'
  // do something, i.e. set thread/event flags
}
```

3. Instantiate and start the timers:

```
// creates a one-shot timer:
one_shot_id = osTimerNew(one_shot_Callback, osTimerOnce, (void *)0, NULL);     // (void*)0 is passed as an argument
                                                                               // to the callback function
// creates a periodic timer:
periodic_id = osTimerNew(periodic_Callback, osTimerPeriodic, (void *)5, NULL); // (void*)5 is passed as an argument
                                                                               // to the callback function

osTimerStart(one_shot_id, 500U);
osTimerStart(periodic_id, 1500U);

// start the one-shot timer again after it has triggered the first time:
osTimerStart(one_shot_id, 500U);

// when timers are not needed any longer free the resources:
osTimerDelete(one_shot_id);
osTimerDelete(periodic_id);
```