SCM1612
Wi-Fi 6 and BLE 5 Low-Power SoC

CMSIS-FreeRTOS API Guide

Revision 0.1
Date 2024-3-26

Contact Information
Senscomm Semiconductor (www.senscomm.com)
Room 303, International Building, West 2 Suzhou Avenue,
SIP, Suzhou, China
For sales or technical support, please send email to
info@senscomm.com

Version History

Version	Date	Description
0.1	2024-3-26	Initial draft
1.0	2024-3-29	Basic errors fixed
1.1	2024-05-27	Updated the status of some APIs.

Table of Contents

1 Introduction

SCM1612 SDK incorporated CMSIS-FreeRTOS as an API which enables users to utilize various OS functions from their applications.
Although native FreeRTOS data types and functions may still be available for use, it is strongly recommended to use CMSIS-FreeRTOS API from application layer.

1.1 Overview

The SCM1612 SDK uses a CMSIS-FreeRTOS port:

- Header file
  - include/cmsis_os.h
- Definitions
  - include/hal/cmsis_os2.h

An application shall include <include/cmsis_os.h> instead of <include/hal/cmsis_os2.h>.

1.2 Build

There is nothing to do specifically to enable CMSIS-FreeRTOS API because it will always be enabled as a default.

2 Reference

CMSIS-FreeRTOS in scm1612 SDK is based on CMSIS-RTOS API v2 API which can be divided into following categories.

- Kernel Information and Control
- Thread Management
- Thread Flags
- Event Flags
- Generic Wait Functions
- Timer Management
- Mutex Management
- Semaphores
- Memory Pool (*)
- Message Queue

These categories will be described below in detail.

(*) Currently not supported.

2.1 Kernel Information and Control

The kernel Information and Control function group allows to:

- obtain information about the system and the underlying kernel.
- obtain version information about the CMSIS-RTOS API.
- initialize of the RTOS kernel for creating objects.
- start the RTOS kernel and thread switching.
- check the execution status of the RTOS kernel.

The kernel information and control functions cannot be called from Interrupt Service Routines.

2.1.1 Data Structures

- struct osVersion_t

  Identifies the underlying RTOS kernel and API version number.
  The version is represented in a combined decimal number in the format:

major.minor.rev: mmnnnrrrr

Data Fields		
uint32_t	api	API version (major.minor.rev: mmnnnrrrr dec).
uint32_t	kernel	Kernel version (major.minor.rev: mmnnnrrrr dec).

Use osKernelGetInfo to retrieve the version numbers.

2.1.2 Enumeration Types

- 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.
osKernelReserved	Prevents enum down-size compiler optimization. Reserved.

2.1.3 Functions

- osStatus_t osKernelInitialize(void)

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

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

- osStatus_t osKernelGetInfo(osVersion_t *version, char *id_buf, uint32_t id_size)

The function osKernelGetInfo retrieves the API and kernel version of the underlying RTOS kernel and a human readable identifier string for the kernel. It can be safely called before the RTOS is initialized or started (call to osKernelInitialize or osKernelStart).

Parameter		
version	out	pointer to buffer for retrieving version information.
Id_buf	out	pointer to buffer for retrieving kernel identification string.
Id_size	in	size of buffer for kernel identification string.

Return	
osOK	in case of success.
osError	if an unspecified error occurred.
osErrorISR	if called from an Interrupt Service Routine.

- osKernelState_t osKernelGetState(void)

  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).

Return	
osKernelError	if called from an Interrupt Service Routine.
the actual kernel state	otherwise.

- osStatus_t osKernelStart(void)

  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.

Return	
osOK	in case of success.
osError	if an unspecified error occurred.
osErrorISR	if called from an Interrupt Service Routine.

- int32_t osKernelLock(void)

  The function osKernelLock allows to lock all task switches. It returns the previous value of the lock state (1 if it was locked, 0 if it was unlocked), or a negative number representing an error code otherwise (refer to osStatus_t).

Return	
osError	if an unspecified error occurred.
osErrorISR	if called from an Interrupt Service Routine.
the previous value of the lock state	otherwise

- int32_t osKernelUnlock(void)

  The function osKernelUnlock resumes from osKernelLock. It returns the previous value of the lock state (1 if it was locked, 0 if it was unlocked), or a negative number representing an error code otherwise (refer to osStatus_t).

Return	
osError	if an unspecified error occurred.
osErrorISR	if called from an Interrupt Service Routine.
the previous value of the lock state	otherwise

- int32_t osKernelRestoreLock(int32_t lock)

  The function osKernelRestoreLock restores the previous lock state after osKernelLock or osKernelUnlock.
  The argument lock specifies the lock state as obtained by osKernelLock or osKernelUnlock.
  The function returns the new value of the lock state (1 if it was locked, 0 if it was unlocked), or a negative number representing an error code otherwise (refer to osStatus_t).

Return	
osError	if an unspecified error occurred.
osErrorISR	if called from an Interrupt Service Routine.
new lock state	otherwise

- uint32_t osKernelSuspend(void)

  Not supported.

- uint32_t osKernelResume(void)

  Not supported.

- uint32_t osKernelGetTickCount(void)

  The function osKernelGetTickCount returns the current RTOS kernel tick count.

- uint32_t osKernelGetTickFreq(void)

  The function osKernelGetTickFreq returns the frequency of the current RTOS kernel tick.

- uint32_t osKernelGetSysTimerCount(void)

  The function osKernelGetSysTimerCount returns the current RTOS kernel system timer as a 32-bit value. The value is a rolling 32-bit counter that is composed of the kernel system interrupt timer value and the counter that counts these interrupts (RTOS kernel ticks).

  This function allows the implementation of very short timeout checks below the RTOS tick granularity. Such checks might be required when checking for a busy status in a device or peripheral initialization routine.

- uint32_t osKernelGetSysTimerFreq(void)

  The function osKernelGetSysTimerFreq returns the frequency of the current RTOS kernel system timer.

2.2 Thread Management

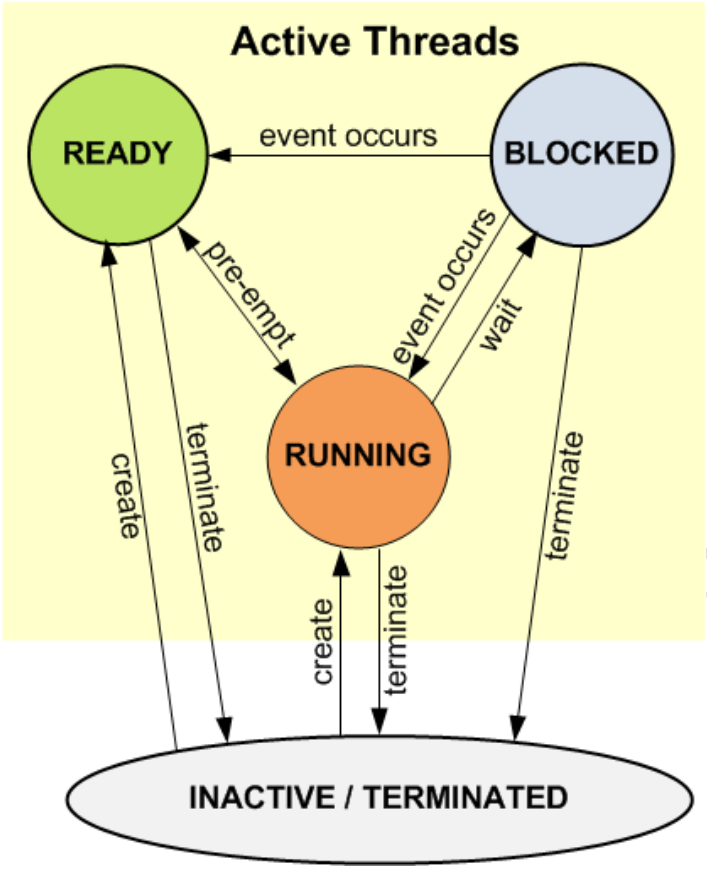
The Thread Management function group allows defining, creating, and controlling thread functions in the system.

Threads can be in the following states:

- RUNNING: The thread that is currently running is in the RUNNING state. Only one thread at a time can be in this state.
- READY: Threads which are ready to run are in the READY state. Once the RUNNING thread has terminated, or is BLOCKED, the next READY thread with the highest priority becomes the RUNNING thread.
- BLOCKED: Threads that are blocked either delayed, waiting for an event to occur or suspended are in the BLOCKED state.
- TERMINATED: When osThreadTerminate is called, threads are TERMINATED with resources not yet released (applies to joinable threads).
- INACTIVE: Threads that are not created or have been terminated with all resources released are in the INACTIVE state.

The thread states change as follows:

- A thread is created using the function osThreadNew. This puts the thread into the READY or RUNNING state (depending on the thread priority).
- CMSIS-RTOS is preemptive. The active thread with the highest priority becomes the RUNNING thread provided it does not wait for any event. The initial priority of a thread is defined with the osThreadAttr_t but may be changed during execution using the function osThreadSetPriority.
- The RUNNING thread transfers into the BLOCKED state when it is delayed, waiting for an event or suspended.
- Active threads can be terminated any time using the function osThreadTerminate. Threads can terminate also by just returning from the thread function. Threads that are terminated are in the INACTIVE state and typically do not consume any dynamic memory resources.



2.2.1 Data Structures

- struct osThreadAttr_t

Data Fields		
const char *	name	name of the thread Pointer to a constant string with a human readable name (displayed during debugging) of the thread object. Default: NULL no name specified (debugger may display function name instead).

uint32_t	attr_bits	attribute bits The following bit masks can be used to set options: • osThreadDetached: create thread in a detached mode (default). • osThreadJoinable: create thread in joinable mode.
void *	cb_mem	memory for control block Pointer to a memory for the thread control block object. Default: NULL to use Automatic Dynamic Allocation for the thread control block.
uint32_t	cb_size	size of provided memory for control block The size (in bytes) of memory block passed with cb_mem. Default: 0 as the default is no memory provided with cb_mem.
void *	stack_mem	memory for stack Pointer to a memory location for the thread stack (64-bit aligned). Default: NULL to allocate stack from a fixed-size memory pool using Thread Stack Management.
uint32_t	stack_size	size of stack The size (in bytes) of the stack specified by stack_mem. Default: 0 as the default is no memory provided with stack_mem.
osPriority_t	priority	initial thread priority (default: osPriorityNormal)

		Specifies the initial thread priority with a value from osPriority_t. Default: osPriorityNormal.
TZ_ModuleId_t	tz_module	TrustZone module identifier. TrustZone Thread Context Management Identifier to allocate context memory for threads. The RTOS kernel that runs in non-secure state calls the interface functions defined by the header file TZ_context.h. Can safely be set to zero for threads not using secure calls at all. See TrustZone RTOS Context Management. Default: 0 not thread context specified.
uint32_t	reserved	reserved (must be 0) Reserved for future use.

2.2.2 Typedefs

- void (*osThreadFunc_t)(void *argument)

  Entry function for threads. Setting up a new thread (osThreadNew) will start execution with a call into this entry function. The optional argument can be used to hand over arbitrary user data to the thread, i.e. to identify the thread or for runtime parameters.

Parameter		
argument	in	Arbitrary user data set on osThreadNew.

- osThreadId_t

  Thread ID identifies the thread.

  Returned by:

  - osThreadNew
  - osThreadGetId
  - osThreadEnumerate

- osMutexGetOwner

2.2.3 Enumeration Types

- enum osThreadState_t

State of a thread as retrieved by osThreadGetState. In case osThreadGetState fails or if it is called from an ISR, it will return osThreadError, otherwise it returns the thread state.

Enumerator	
osThreadInactive	Inactive. The thread is created but not actively used, or has been terminated (returned for static control block allocation, when memory pools are used osThreadError is returned as the control block is no longer valid)
osThreadReady	Ready. The thread is ready for execution but not currently running.
osThreadRunning	Running. The thread is currently running.
osThreadBlocked	Blocked. The thread is currently blocked (delayed, waiting for an event or suspended).
osThreadTerminated	Terminated.

	The thread is terminated and all its resources are not yet freed (applies to joinable threads).
osThreadError	Error. The thread does not exist (has raised an error condition) and cannot be scheduled.
osThreadReserved	Prevents enum down-size compiler optimization.

- enum osPriority_t

  The osPriority_t value specifies the priority for a thread. The default thread priority should be osPriorityNormal. If an active thread becomes ready that has a higher priority than the currently running thread then a thread switch occurs immediately. The system continues executing the thread with the higher priority.

Enumerator	
osPriorityNone	No priority (not initialized).
osPriorityIdle	Reserved for Idle thread. This lowest priority should not be used for any other thread.
osPriorityLow	Priority: low.
osPriorityLow1	Priority: low + 1.
osPriorityLow2	Priority: low + 2.
osPriorityLow3	Priority: low + 3.

osPriorityLow4	Priority: low + 4.
osPriorityLow5	Priority: low + 5.
osPriorityLow6	Priority: low + 6.
osPriorityLow7	Priority: low + 7.
osPriorityBelowNormal	Priority: below normal.
osPriorityBelowNormal1	Priority: below normal + 1.
osPriorityBelowNormal2	Priority: below normal + 2.
osPriorityBelowNormal3	Priority: below normal + 3.
osPriorityBelowNormal4	Priority: below normal + 4.
osPriorityBelowNormal5	Priority: below normal + 5.
osPriorityBelowNorma6	Priority: below normal + 6.
osPriorityBelowNorma7	Priority: below normal + 7.
osPriorityNormal	Priority: normal.
osPriorityNormal1	Priority: normal + 1.

osPriorityNormal2	Priority: normal + 2.
osPriorityNormal3	Priority: normal + 3.
osPriorityNormal4	Priority: normal + 4.
osPriorityNormal5	Priority: normal + 5.
osPriorityNormal6	Priority: normal + 6.
osPriorityNormal7	Priority: normal + 7.
osPriorityAboveNormal	Priority: above normal.
osPriorityAboveNormal1	Priority: above normal + 1.
osPriorityAboveNormal2	Priority: above normal + 2.
osPriorityAboveNormal3	Priority: above normal + 3.
osPriorityAboveNormal4	Priority: above normal + 4.
osPriorityAboveNormal5	Priority: above normal + 5.
osPriorityAboveNormal6	Priority: above normal + 6.
osPriorityAboveNormal7	Priority: above normal + 7.

osPriorityHigh	Priority: high.
osPriorityHigh1	Priority: high + 1.
osPriorityHigh2	Priority: high + 2.
osPriorityHigh3	Priority: high + 3.
osPriorityHigh4	Priority: high + 4.
osPriorityHigh5	Priority: high + 5.
osPriorityHigh6	Priority: high + 6.
osPriorityHigh7	Priority: high + 7.
osPriorityRealtime	Priority: realtime.
osPriorityRealtime1	Priority: realtime + 1.
osPriorityRealtime2	Priority: realtime + 2.
osPriorityRealtime3	Priority: realtime + 3.
osPriorityRealtime4	Priority: realtime + 4.
osPriorityRealtime5	Priority: realtime + 5.

osPriorityRealtime6	Priority: realtime + 6.
osPriorityRealtime7	Priority: realtime + 7.
osPriorityISR	Reserved for ISR deferred thread. This highest priority might be used by the RTOS implementation but must not be used for any user thread.
osPriorityError	System cannot determine priority or illegal priority.
osPriorityReserved	Prevents enum down-size compiler optimization.

2.2.4 Functions

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

  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.

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

Return	
thread ID	in case of success.
NULL	in case of error.

- const char *osThreadGetName(osThreadId_t thread_id)

The function osThreadGetName returns the pointer to the name string of the thread identified by parameter thread_id or NULL in case of an error.

Parameter		
thread_id	in	thread ID obtained by osThreadNew or osThreadGetId.

Return	
null-terminated thread name string	in case of success.
NULL	in case of error.

- osThreadId_t osThreadGetId(void)

The function osThreadGetId returns the thread object ID of the currently running thread or NULL in case of an error.

Return	
thread ID	in case of success.
NULL	in case of error.

- osThreadState_t osThreadGetState(osThreadId_t thread_id)

The function osThreadGetState returns the state of the thread identified by parameter thread_id. In case it fails or if it is called from an ISR, it will return osThreadError, otherwise it returns the thread state (refer to osThreadState_t for the list of thread states).

Parameter		
thread_id	in	thread ID obtained by osThreadNew or osThreadGetId.

Return	
current thread state	in case of success.
osTheadError	in case of error.

- osStatus_t osThreadSetPriority(osThreadId_t thread_id, osPriority_t priority)

  The function osThreadSetPriority changes the priority of an active thread specified by the parameter thread_id to the priority specified by the parameter priority.

Parameter		
thread_id	in	thread ID obtained by osThreadNew or osThreadGetId.
priority	in	new priority value for the thread function.

Return	
osOK	the priority of the specified thread has been changed successfully.
osErrorParameter	thread_id is NULL or invalid or priority is incorrect.
osErrorResource	the thread is in an invalid state.
osErrorISR	function osThreadSetPriority cannot be called from interrupt service routines.

- osPriority_t osThreadGetPriority(osThreadId_t thread_id)

  The function osThreadGetPriority returns the priority of an active thread specified by the parameter thread_id.

Parameter

thread_id	in	thread ID obtained by osThreadNew or osThreadGetId.

Return	
priority	the priority of the specified thread.
osPriorityError	priority cannot be determined or is illegal. It is also returned when the function is called from Interrupt Service Routines.

- osStatus_t osThreadYield(void)

  The function osThreadYield passes control to the next thread with the same priority that is in the READY state. If there is no other thread with the same priority in state READY, then the current thread continues execution and no thread switch occurs. osThreadYield does not set the thread to state BLOCKED. Thus no thread with a lower priority will be scheduled even if threads in state READY are available.

  This function has no impact when called when the kernel is locked, see osKernelLock.

Return	
osOK	control has been passed to the next thread successfully.
osError	an unspecified error has occurred.
osErrorISR	the function osThreadYield cannot be called from interrupt service routines.

- osStatus_t osThreadSuspend(osThreadId_t thread_id)

  The function osThreadSuspend suspends the execution of the thread identified by parameter thread_id. The thread is put into the BLOCKED state (osThreadBlocked). Suspending the running thread will cause a context switch to another thread in READY state immediately. The suspended thread is not executed until explicitly resumed with the function osThreadResume.

  Threads that are already BLOCKED are removed from any wait list and become ready when they are resumed. Thus it is not recommended to suspend an already blocked thread.

This function must not be called to suspend the running thread when the kernel is locked, i.e. osKernelLock.

Parameter		
thread_id	in	thread ID obtained by osThreadNew or osThreadGetId.

Return	
osOK	the thread has been suspended successfully.
osErrorParameter	thread_id is NULL or invalid.
osErrorResource	the thread is in an invalid state.
osErrorISR	the function osThreadSuspend cannot be called from interrupt service routines.

- osStatus_t osThreadResume(osThreadId_t thread_id)

The function osThreadResume puts the thread identified by parameter thread_id (which has to be in BLOCKED state) back to the READY state. If the resumed thread has a higher priority than the running thread a context switch occurs immediately.

The thread becomes ready regardless of the reason why the thread was blocked. Thus it is not recommended to resume a thread not suspended by osThreadSuspend.

Functions that will put a thread into BLOCKED state are: osEventFlagsWait and osThreadFlagsWait, osDelay and osDelayUntil, osMutexAcquire and osSemaphoreAcquire, osMessageQueueGet, osMemoryPoolAlloc, osThreadJoin, osThreadSuspend.

This function may be called when kernel is locked (osKernelLock). Under this circumstances a potential context switch is delayed until the kernel gets unlocked, i.e. osKernelUnlock or osKernelRestoreLock.

Parameter		
thread_id	in	thread ID obtained by osThreadNew or osThreadGetId.

Return

osOK	the thread has been resumed successfully.
osErrorParameter	thread_id is NULL or invalid.
osErrorResource	the thread is in an invalid state.
osErrorISR	the function osThreadResume cannot be called from interrupt service routines.

- osStatus_t osThreadDetach(osThreadId_t thread_id)

  Not supported.

- osStatus_t osThreadJoin(osThreadId_t thread_id)

  Not supported.

- __NO_RETURN void osThreadExit(void)

  The function osThreadExit terminates the calling thread. This allows the thread to be synchronized with osThreadJoin.

- osStatus_t osThreadTerminate(osThreadId_t thread_id)

  The function osThreadTerminate removes the thread specified by parameter thread_id from the list of active threads. If the thread is currently RUNNING, the thread terminates and the execution continues with the next READY thread. If no such thread exists, the function will not terminate the running thread, but return osErrorResource.

  Avoid calling the function with a thread_id that does not exist or has been terminated already.
  osThreadTerminate destroys non-joinable threads and removes their thread_id from the system. Subsequent access to the thread_id (for example osThreadGetState) will return an osThreadError. Joinable threads will not be destroyed and return the status osThreadTerminated until they are joined with osThreadJoin.

Parameter		
thread_id	in	thread ID obtained by osThreadNew or osThreadGetId.

Return	
osOK	the specified thread has been removed from the active thread list successfully.
osErrorParameter	thread_id is NULL or invalid.
osErrorResource	the thread is in an invalid state or no other READY thread exists..
osErrorISR	the function osThreadTerminate cannot be called from interrupt service routines.

- uint32_t osThreadGetStackSize(osThreadId_t thread_id)

  Not supported.

- uint32_t osThreadGetStackSpace(osThreadId_t thread_id)

  The function osThreadGetStackSpace returns the size of unused stack space for the thread specified by parameter thread_id. Stack watermark recording during execution needs to be enabled (refer to Thread Configuration). In case of an error, it returns 0.

Parameter		
thread_id	in	thread ID obtained by osThreadNew or osThreadGetId.

Return	
remaining stack size in bytes	in case of success.
0	In case of error.

- uint32_t osThreadGetCount(void)

  The function osThreadGetCount returns the number of active threads or 0 in case of an error.

Return	
number of active threads	in case of success.
0	In case of error.

- uint32_t osThreadEnumerate(osThreadId_t *thread_array, uint32_t array_items)

  The function osThreadEnumerate returns the number of enumerated threads or 0 in case of an error.

Parameter		
thread_array	out	pointer to array for retrieving thread IDs.
array_items	in	maximum number of items in array for retrieving thread IDs.

Return	
number of enumerated threads	in case of success.
0	In case of error.

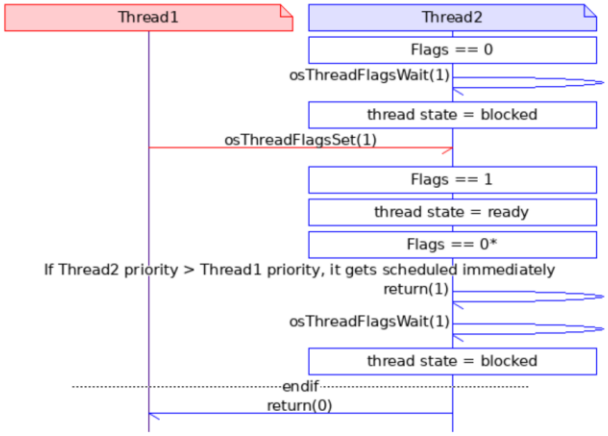
2.3 Thread Flags

Thread Flags are a more specialized version of the Event Flags. See Event Flags. While Event Flags can be used to globally signal a number of threads, thread flags are only sent to a single specific thread. Every thread instance can receive thread flags without any additional allocation of a thread flags object. Thread flag management functions cannot be called from Interrupt Service Routines, except for osThreadFlagsSet.

Usage Examples

The following diagram assumes that in the control block of Thread1 the flag 1 is already set. Thread2 now sets flag 2 and Thread1 returns the updated value immediately:

Depending on thread scheduling, the flag status can be modified before returning:



Note
* In this case **osThreadFlagsWait** auto-clears the flag.

2.3.1 Functions

- uint32_t osThreadFlagsSet(osThreadId_t thread_id, uint32_t flags)

The function osThreadFlagsSet sets the thread flags for a thread specified by parameter thread_id. The thread returns the flags stored in the thread control block, or an error code if highest bit is set (refer to Flags Functions Error Codes). Refer to Usage Examples below to understand how the return value is computed.

The target thread waiting for a flag to be set will resume from BLOCKED state.

Parameter		
thread_id	in	thread ID obtained by osThreadNew or osThreadGetId.
flags	in	specifies the flags of the thread that shall be set.

Return	
thread flags after setting	in case of success.
osFlagsErrorUnknown	unspecified error.
osFlagsErrorParameter	parameter thread_id is not a valid thread or flags has highest bit set.
osFlagsErrorResource	the thread is in invalid state.

- uint32_t osThreadFlagsClear(uint32_t flags)

  The function osThreadFlagsClear clears the specified flags for the currently running thread. It returns the flags before clearing, or an error code if highest bit is set (refer to Flags Functions Error Codes).

Parameter		
flags	in	specifies the flags of the thread that shall be cleared.

Return	
thread flags before clearing	in case of success.
osFlagsErrorUnknown	unspecified error, i.e. not called from a running threads context.
osFlagsErrorParameter	parameter flags has highest bit set.
osFlagsErrorISR	the function osThreadFlagsClear cannot be called from interrupt service routines.

- uint32_t osThreadFlagsGet(void)

  The function osThreadFlagsGet returns the flags currently set for the currently running thread. If called without a active and currently running thread osThreadFlagsGet return zero.

Return	
current thread flags	

- uint32_t osThreadFlagsWait(uint32_t flags, uint32_t options, uint32_t timeout)

The function osThreadFlagsWait suspends the execution of the currently RUNNING thread until any or all of the thread flags specified with the parameter flags are set. When these thread flags are already set, the function returns instantly. Otherwise the thread is put into the state BLOCKED.

The parameter options specifies the wait condition:

Option	
osFlagsWaitAny	Wait for any flag (default).
osFlagsWaitAll	Wait for all flags.
osFlagsNoClear	Do not clear flags which have been specified to wait for.

If osFlagsNoClear is set in the options osThreadFlagsClear can be used to clear flags manually. Otherwise osThreadFlagsWait automatically clears the flags waited for.

The parameter timeout represents a number of timer ticks and is an upper bound. The exact time delay depends on the actual time elapsed since the last timer tick.

Parameter		
flags	in	specifies the flags to wait for.
options	in	specifies flags options (osFlagsXxxx).
timeout	in	Timeout Value or 0 in case of no time-out.

Return	
thread flags before clearing	in case of success.
osFlagsErrorUnknown	unspecified error, i.e. not called from a running threads context.
osFlagsErrorParameter	parameter flags has highest bit set.
osFlagsErrorTimeout	awaited flags have not been set in the given time.
osFlagsErrorResource	awaited flags have not been set when no timeout was specified.

2.4 Event Flags

The event flags management functions in CMSIS-RTOS allow you to control or wait for event flags. Each signal has up to 31 event flags.

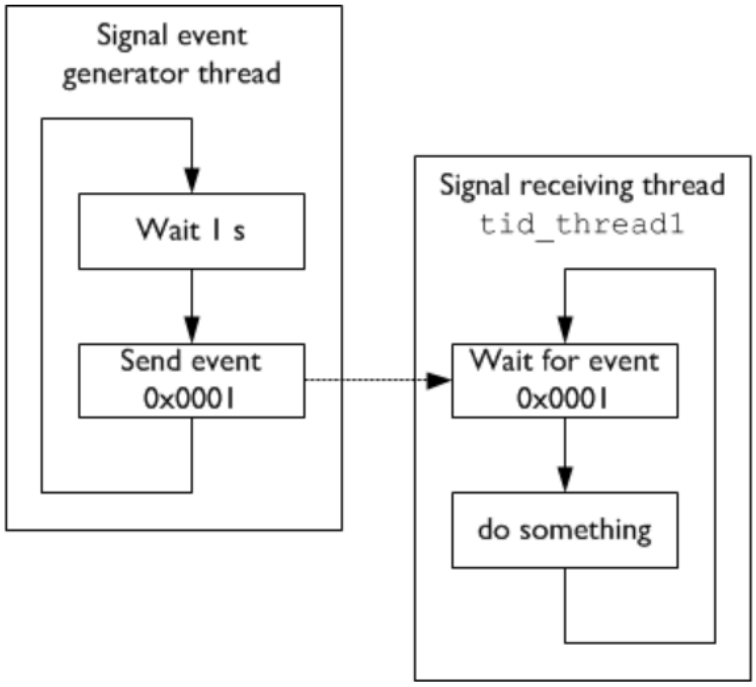
A thread

- can wait for event flags to be set (using osEventFlagsWait). Using this function, it enters the BLOCKED state.
- may set one or more flags in any other given thread (using osEventFlagsSet).
- may clear its own signals or the signals of other threads (using osEventFlagsClear).

When a thread wakes up and resumes execution, its signal flags are automatically cleared (unless event flags option osFlagsNoClear is specified).

The functions osEventFlagsSet, osEventFlagsClear, osEventFlagsGet, and osEventFlagsWait can be called from Interrupt Service Routines.

Working with Events:



2.4.1 Data Structures

- struct osEventFlagsAttr_t

Data Fields		
const char *	name	name of the event flags Pointer to a constant string with a human readable name (displayed during debugging) of the event flag object. Default: NULL no name specified.
uint32_t	attr_bits	attribute bits Reserved for future use (must be set to '0' for future compatibility).
void *	cb_mem	memory for control block Pointer to a memory for the event flag control block object. Default: NULL to use Automatic Dynamic Allocation for the event flag control block.
uint32_t	cb_size	size of provided memory for control block The size (in bytes) of memory block passed with cb_mem. Default: 0 as the default is no memory provided with cb_mem.

2.4.2 Typedefs

- osEventFlagsId_t

  Event Flags ID identifies the event flags.

2.4.3 Functions

- osEventFlagsId_t osEventFlagsNew(const osEventFlagsAttr_t *attr)

  The function osEventFlagsNew creates a new event flags object that is used to send events across threads and returns the pointer to the event flags 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 event flags attributes (refer to osEventFlagsAttr_t). Default attributes will be used if set to NULL, i.e. kernel memory allocation is used for the event control block.

Parameter		
attr	in	event flags attributes; NULL: default values.

Return	
event flags ID	in case of success.
NULL	in case of error.

- uint32_t osEventFlagsSet(osEventFlagsId_t ef_id, uint32_t flags)

  The function osEventFlagsSet sets the event flags specified by the parameter flags in an event flags object specified by parameter ef_id.

  The threads with highest priority waiting for the flag(s) set will be notified to resume from BLOCKED state. The function returns the event flags stored in the event control block or an error code (highest bit is set, refer to Flags Functions Error Codes). Further threads may be wakened in priority order when the option osFlagsNoClear is given to the osEventFlagsWait call.

Parameter		
ef_id	in	event flags ID obtained by osEventFlagsNew.
flags	in	specifies the flags that shall be set.

Return	
event flags after setting	in case of success.
osFlagsErrorUnknown	unspecified error.
osFlagsErrorParameter	parameter ef_id does not identify a valid event flags object or flags has highest bit set.
osFlagsErrorResource	the event flags object is in an invalid state.

- uint32_t osEventFlagsClear(osEventFlagsId_t ef_id, uint32_t flags)

  The function osEventFlagsClear clears the event flags specified by the parameter flags in an event flags object specified by parameter ef_id. The function returns the event flags before clearing or an error code.

Parameter		
ef_id	in	event flags ID obtained by osEventFlagsNew.
flags	in	specifies the flags that shall be cleared.

Return	
event flags before clearing	in case of success.
osFlagsErrorUnknown	unspecified error.
osFlagsErrorParameter	parameter ef_id does not identify a valid event flags object or flags has highest bit set.
osFlagsErrorResource	the event flags object is in an invalid state.

- uint32_t osEventFlagsGet(osEventFlagsId_t ef_id)

  The function osEventFlagsGet returns the event flags currently set in an event flags object specified by parameter ef_id or 0 in case of an error.

Parameter		
ef_id	in	event flags ID obtained by osEventFlagsNew.

Return	
current event flags.	in case of success.
0	in case of error.

- uint32_t osEventFlagsWait(osEventFlagsId_t ef_id, uint32_t flags, uint32_t options, uint32_t timeout)

  The function osEventFlagsWait suspends the execution of the currently RUNNING thread until any or all event flags specified by the parameter flags in the event object specified by parameter ef_id are set. When these event flags are already set, the function returns instantly. Otherwise, the thread is put into the state BLOCKED.

  The options parameter specifies the wait condition:

Option	
osFlagsWaitAny	Wait for any flag (default).
osFlagsWaitAll	Wait for all flags.
osFlagsNoClear	Do not clear flags which have been specified to wait for.

  If osFlagsNoClear is set in the options osEventFlagsClear can be used to clear flags manually.

  The parameter timeout specifies how long the system waits for event flags. 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 event flags become available (i.e. wait semantics).
  - all other values specify a time in kernel ticks for a timeout (i.e. timed-wait semantics).

Parameter		
ef_id	in	event flags ID obtained by osEventFlagsNew.
flags	in	specifies the flags to wait for.
options	in	specifies flags options (osFlagsXxxx).
timeout	in	timeout value or 0 in case of no time-out.

Return	
event flags before clearing	in case of success.
osFlagsErrorUnknown	unspecified error.
osFlagsErrorTimeout	awaited flags have not been set in the given time.
osFlagsErrorParameter	parameter ef_id does not identify a valid event flags object or flags has highest bit set.
osFlagsErrorResource	awaited flags have not been set when no timeout was specified.

- osStatus_t osEventFlagsDelete(osEventFlagsId_t ef_id)

  The function osEventFlagsDelete deletes the event flags object specified by parameter ef_id and releases the internal memory obtained for the event flags handling. After this call, the ef_id is no longer valid and cannot be used. This can cause starvation of threads that are waiting for flags of this event object. The ef_id may be created again using the function osEventFlagsNew.

Parameter		
ef_id	in	event flags ID obtained by osEventFlagsNew.

Return	
osOK	the specified event flags object has been deleted.
osErrorISR	osEventFlagsDelete cannot be called from interrupt service routines.
osFlagsErrorParameter	parameter ef_id is NULL or invalid.
osFlagsErrorResource	the event flags object is in an invalid state.

- const char * osEventFlagsGetName(osEventFlagsId_t ef_id)

  The function osEventFlagsGetName returns the pointer to the name string of the event flags object identified by parameter ef_id or NULL in case of an error.

Parameter		
ef_id	in	event flags ID obtained by osEventFlagsNew.

Return	
name as null-terminated string	In case of success.
NULL	in case of error.

This API is currently not supported.

2.5 Generic Wait Functions

The generic wait functions provide means for a time delay.

2.5.1 Functions

- osStatus_t osDelay(uint32_t ticks)

  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.

Parameter		
ticks	in	time ticks value

Return	
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).

- osStatus_t osDelayUntil(uint32_t ticks)

  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.

Parameter		
ticks	in	absolute time in ticks

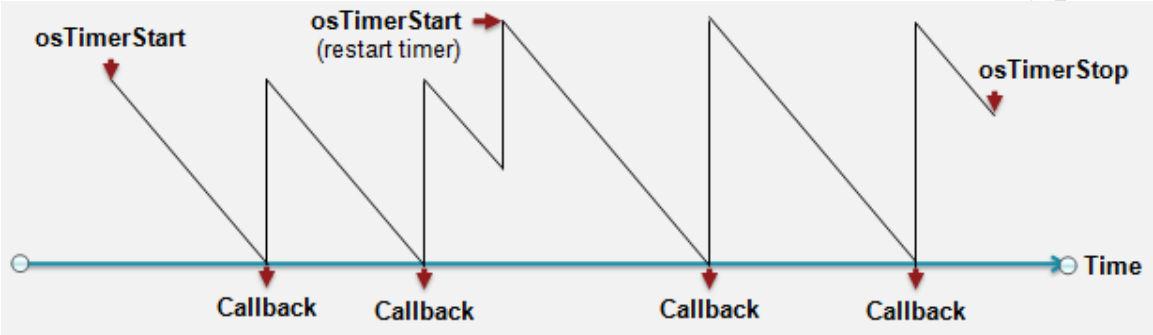
Return	
osOK	the time delay is executed.
osErrorParameter	the time cannot be handled (out of bounds).
osErrorISR	osDelayUntil cannot be called from Interrupt Service Routines.
osError	osDelayUntil cannot be executed (kernel not running or no READY thread exists).

2.6 Timer Management

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.

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.



The following steps are required to use a software timer:

- Define the timers:

```
osTimerId_t one_shot_id, periodic_id;
```

- 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
}
```

- 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);
```

2.6.1 Data Structures

- struct osTimerAttr_t

Data Fields		
const char *	name	name of the timer Pointer to a constant string with a human readable name (displayed during debugging) of the timer object. Default: NULL no name specified.
uint32_t	attr_bits	attribute bits Reserved for future use (must be set to '0' for future compatibility).
void *	cb_mem	memory for control block Pointer to a memory for the timer control block object. Default: NULL to use Automatic Dynamic Allocation for the timer control block.
uint32_t	cb_size	size of provided memory for control block

		The size (in bytes) of memory block passed with cb_mem. Default: 0 as the default is no memory provided with cb_mem.

2.6.2 Typedefs

- osTimerId_t

  Timer ID identifies the timer.
  Instances of this type hold a reference to a timer object.

- void(* osTimerFunc_t)(void *argument)

  The timer callback function is called every time the timer elapses.

  The callback might be executed either in a dedicated timer thread or in interrupt context. Thus it is recommended to only use ISR callable functions from the timer callback.

Parameter		
argument	in	The argument provided to osTimerNew.

2.6.3 Enumeration Types

- enum osTimerType_t

  The osTimerType_t specifies the a repeating (periodic) or one-shot timer for the function osTimerNew.

Enumerator	
osTimerOnce	One-shot timer. The timer is not automatically restarted once it has elapsed. It can be restarted manually using osTimerStart as needed.

osTimerPeriodic	Repeating timer. The timer repeats automatically and triggers the callback continuously while running, see osTimerStart and osTimerStop.

2.6.4 Functions

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

  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.

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

Return	
timer ID	in case of success.
NULL	in case of error.

- const char *osTimerGetName(osTimerId_t timer_id)

  The function osTimerGetName returns the pointer to the name string of the timer identified by parameter timer_id or NULL in case of an error.

Parameter		
timer_id	in	timer ID obtained by osTimerNew.

Return	
name as null-terminated string	in case of success.
NULL	in case of error.

- osStatus_t osTimerStart(osTimerId_t timer_id, uint32_t ticks)

  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.

Parameter		
timer_id	in	timer ID obtained by osTimerNew.
ticks	in	time ticks value of the timer.

Return	
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.
osErrorNeedSched	the current thread will be preempted.

- osStatus_t osTimerStop(osTimerId_t timer_id)

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

Parameter		
timer_id	in	timer ID obtained by osTimerNew.

Return

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).
osErrorNeedSched	the current thread will be preempted.

- uint32_t osTimerIsRunning(osTimerId_t timer_id)

  The function osTimerIsRunning checks whether a timer specified by parameter timer_id is running. It returns 1 if the timer is running and 0 if the timer is stopped or an error occurred.

Parameter		
timer_id	in	timer ID obtained by osTimerNew.

Return	
0	the timer is stopped.
1	the timer is running.

- osStatus_t osTimerDelete(osTimerId_t timer_id)

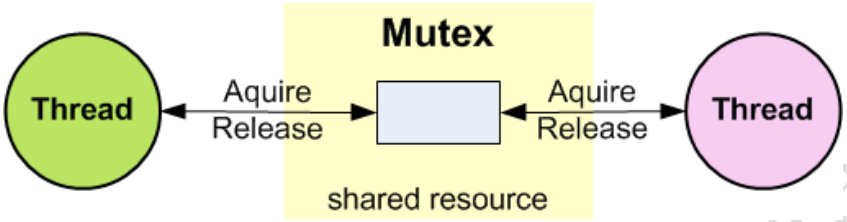
  The function osTimerDelete deletes the timer specified by parameter timer_id.

Parameter		
timer_id	in	timer ID obtained by osTimerNew.

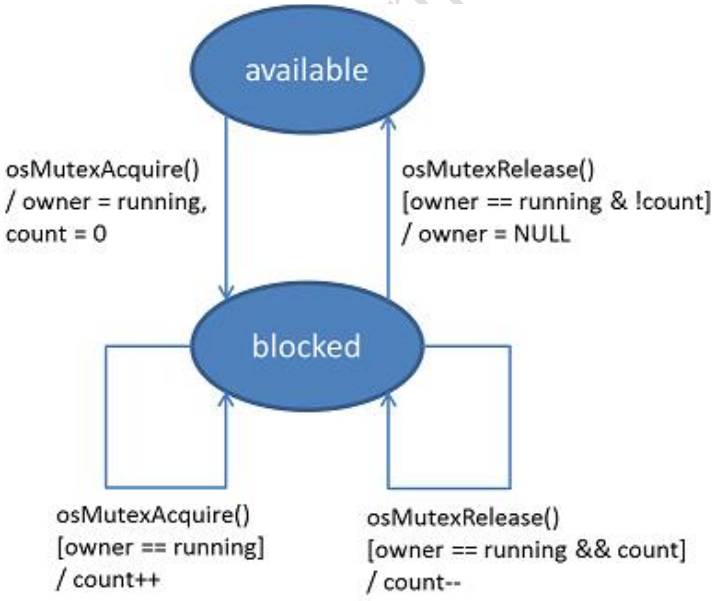
Return	
osOK	the specified timer has been deleted.
osErrorISR	osTimerDelete cannot be called from interrupt service routines.
osErrorParameter	parameter timer_id is either NULL or invalid.
osErrorResource	the timer is in an invalid state.

2.7 Mutex Management

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).



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.



Mutex management functions cannot be called from Interrupt Service Routines (ISR), unlike a binary semaphore that can be released from an ISR.

2.7.1 Data Structures

- struct osMutexAttr_t

Data Fields		
const char *	name	name of the mutex Pointer to a constant string with a human readable name (displayed during debugging) of the mutex object. Default: NULL no name specified.
uint32_t	attr_bits	attribute bits The following bit masks can be used to set options: osMutexRecursive: a thread can consume the mutex multiple times without locking itself. osMutexPrioInherit(*): the owner thread inherits the priority of a (higher priority) waiting thread. osMutexRobust(*): the mutex is automatically released when owner thread is terminated. Use logical 'OR' operation to select multiple options, for example: osMutexRecursive | osMutexPrioInherit; Default: 0 which specifies:

		non recursive mutex: a thread cannot consume the mutex multiple times. non priority raising: the priority of an owning thread is not changed. mutex is not automatically release: the mutex object must be always is automatically released when owner thread is terminated. (*): Not supported
void *	cb_mem	memory for control block Pointer to a memory for the mutex control block object. Default: NULL to use Automatic Dynamic Allocation for the timer control block.
uint32_t	cb_size	size of provided memory for control block The size (in bytes) of memory block passed with cb_mem. Default: 0 as the default is no memory provided with cb_mem.

2.7.2 Macro Definitions

- #define osMutexRecursive 0x00000001U

  Recursive flag in osMutexAttr_t.

  The same thread can consume a mutex multiple times without locking itself. Each time the owning thread acquires the mutex the lock count is incremented. The mutex must be released multiple times as well until the lock count reaches zero. At reaching zero the mutex is actually released and can be acquired by other threads.

The maximum amount of recursive locks possible is implementation specific, i.e. the type size used for the lock count. If the maximum amount of recursive locks is depleted mutex acquire might fail.

- #define osMutexPrioInherit 0x00000002U

  Priority inheritance flag in osMutexAttr_t.

  A mutex using priority inheritance protocol transfers a waiting threads priority to the current mutex owner if the owners thread priority is lower. This assures that a low priority thread does not block a high priority thread.

  Otherwise a low priority thread might hold a mutex but is not granted execution time due to another mid priority thread. Without priority inheritance the high priority thread waiting for the mutex would be blocked by the mid priority thread, called priority inversion.

  This flag is currently not supported.

- #define osMutexRobust 0x00000008U

  Robust flag in osMutexAttr_t.

  Robust mutexes are automatically released if the owning thread is terminated (either by osThreadExit or osThreadTerminate). Non-robust mutexes are not released and the user must assure mutex release manually.

  This flag is currently not supported.

2.7.3 Typedefs

- osMutexId_t

  Mutex ID identifies the mutex.

2.7.4 Functions

- osMutexId_t osMutexNew(const osMutexAttr_t *attr)

  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.

Parameter		
attr	in	mutex attributes; NULL: default values.

Return	
mutex ID	in case of success.
NULL	in case of error.

- const char *osMutexGetName(osMutexId_t mutex_id)

  The function osMutexGetName returns the pointer to the name string of the mutex identified by parameter mutex_id or NULL in case of an error.

Parameter		
mutex_id	in	mutex ID obtained by osMutexNew.

Return	
name as null-terminated string	in case of success.
NULL	in case of error.

  This API is currently not supported.

- osStatus_t osMutexAcquire(osMutexId_t mutex_id, uint32_t timeout)

  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).

Parameter		
mutex_id	in	mutex ID obtained by osMutexNew.
timeout	in	timeout value or 0 in case of no time-out.

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

- osStatus_t osMutexRelease(osMutexId_t mutex_id)

  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.

Parameter		
mutex_id	in	mutex ID obtained by osMutexNew.

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

- osThreadId_t osMutexGetOwner(osMutexId_t mutex_id)

  The function osMutexGetOwner returns the thread ID of the thread that acquired a mutex specified by parameter mutex_id. In case of an error or if the mutex is not blocked by any thread, it returns NULL.

Parameter		
mutex_id	in	mutex ID obtained by osMutexNew.

Return	
thread ID of owner thread	in case of success.
NULL	if mutex was not acquired.

- osStatus_t osMutexDelete(osMutexId_t mutex_id)

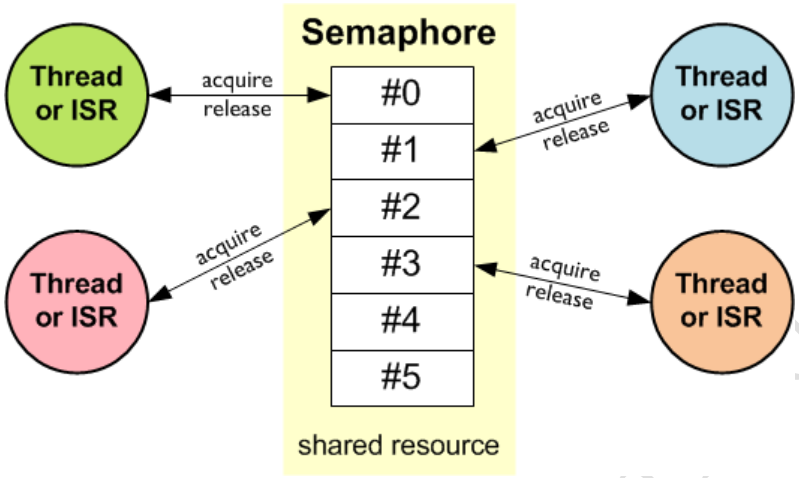
  The function osMutexDelete deletes a mutex object specified by parameter mutex_id. It releases internal memory obtained for mutex handling. After this call, the mutex_id is no longer valid and cannot be used. The mutex may be created again using the function osMutexNew.

Parameter		
mutex_id	in	mutex ID obtained by osMutexNew.

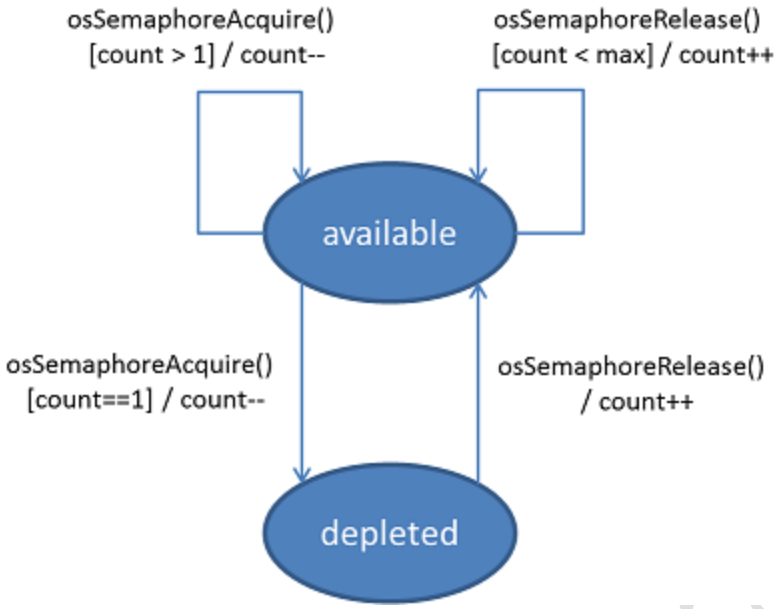
Return	
osOK	the mutex object has been deleted.
osErrorISR	cannot be called from interrupt service routines.
osErrorParameter	parameter mutex_id is either NULL or invalid.

2.8 Semaphores

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).



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.



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

2.8.1 Data Structures

- struct osSemaphoreAttr_t

Data Fields		
const char *	name	name of the semaphore Pointer to a constant string with a human readable name (displayed during debugging) of the semaphore object. Default: NULL no name specified.
uint32_t	attr_bits	attribute bits Reserved for future use (must be set to '0' for future compatibility).
void *	cb_mem	memory for control block

		Pointer to a memory for the semaphore control block object. Default: NULL to use Automatic Dynamic Allocation for the semaphore control block.
uint32_t	cb_size	size of provided memory for control block The size (in bytes) of memory block passed with cb_mem. Default: 0 as the default is no memory provided with cb_mem.

2.8.2 Typedefs

- osSemaphoreId_t

  Semaphore ID identifies the semaphore.

2.8.3 Functions

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

  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.

Parameter

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

Return	
semaphore ID	in case of success.
NULL	in case of error.

- const char * osSemaphoreGetName(osSemaphoreId_t semaphore_id)

  The function osSemaphoreGetName returns the pointer to the name string of the semaphore identified by parameter semaphore_id or NULL in case of an error.

Parameter		
semaphore_id	in	semaphore ID obtained by osSemaphoreNew.

Return	
name as null-terminated string	in case of success.
NULL	in case of error.

  This API is currently not supported.

- osStatus_t osSemaphoreAcquire(osSemaphoreId_t semaphore_id, uint32_t timeout)

  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).

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

Parameter		
semaphore_id	in	semaphore ID obtained by osSemaphoreNew.
timeout	in	timeout value or 0 in case of no time-out.

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

- osStatus_t osSemaphoreRelease(osSemaphoreId_t semaphore_id)

  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.

Parameter		
semaphore_id	in	semaphore ID obtained by osSemaphoreNew.

Return	
osOK	the token has been released and the count incremented.
osErrorParameter	the parameter semaphore_id is NULL or invalid.
osErrorResource	the token could not be released (maximum token count has been reached).
osErrorNeedSched	the current thread will be preempted.

- uint32_t osSemaphoreGetCount(osSemaphoreId_t semaphore_id)

  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.

Parameter		
semaphore_id	in	semaphore ID obtained by osSemaphoreNew.

Return	
number of tokens available.	in case of success.
0	in case of error.

- osStatus_t osSemaphoreDelete(osSemaphoreId_t semaphore_id)

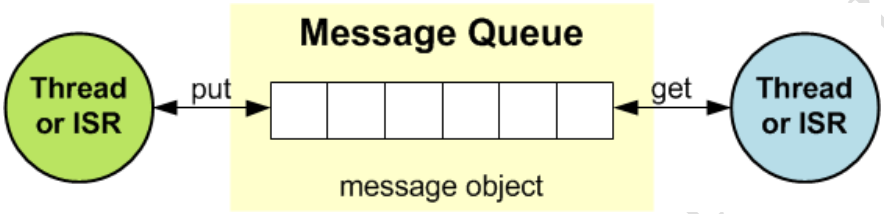
  The function osSemaphoreDelete deletes a semaphore object specified by parameter semaphore_id. It releases internal memory obtained for semaphore handling. After this call, the semaphore_id is no longer valid and cannot be used. The semaphore may be created again using the function osSemaphoreNew.

Parameter		
semaphore_id	in	semaphore ID obtained by osSemaphoreNew.

Return	
osOK	the semaphore object has been deleted.
osErrorParameter	the parameter semaphore_id is NULL or invalid.
osErrorISR	cannot be called from interrupt service routines.

2.9 Message Queue

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 as 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:



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

2.9.1 Data Structures

- struct osMessageQueueAttr_t

Data Fields		
const char *	name	name of the message queue Pointer to a constant string with a human readable name (displayed during debugging) of the message queue object. Default: NULL no name specified.
uint32_t	attr_bits	attribute bits

		Reserved for future use (must be set to '0' for future compatibility).
void *	cb_mem	memory for control block Pointer to a memory for the message queue control block object. Default: NULL to use Automatic Dynamic Allocation for the message queue control block.
uint32_t	cb_size	size of provided memory for control block The size (in bytes) of memory block passed with cb_mem. Default: 0 as the default is no memory provided with cb_mem.
void *	mq_mem	memory for data storage Pointer to a memory for the message queue data. Default: NULL to use Automatic Dynamic Allocation for the memory pool data.
uint32_t	mq_size	size of provided memory for data storage The size (in bytes) of memory block passed with mq_mem. The minimum memory block size is msg_count * msg_size (parameters of the osMessageQueueNew function). The msg_size is rounded up to a double even number to ensure 32-bit alignment of the memory blocks. Default: 0 as the default is no memory provided with mq_mem.

2.9.2 Functions

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

  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.

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

Return	
message queue ID	in case of success.
NULL	in case of error.

- const char *osMessageQueueGetName (osMessageQueueId_t mq_id)

  The function osMessageQueueGetName returns the pointer to the name string of the message queue identified by parameter mq_id or NULL in case of an error.

Parameter		
mq_id	in	message queue ID obtained by osMessageQueueNew.

Return	
name as null-terminated string	in case of success.
NULL	in case of error.

This API is currently not supported.

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

  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).

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

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

Return	
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.
osErrorNeedSched	the current thread will be preempted.

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

  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).

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

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

Return	
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.

- uint32_t osMessageQueueGetCapacity(osMessageQueueId_t mq_id)

  The function osMessageQueueGetCapacity returns the maximum number of messages in the message queue object specified by parameter mq_id or 0 in case of an error.

Parameter		
mq_id	in	message queue ID obtained by osMessageQueueNew.

Return	
maximum number of messages	in case of success.
0	in case of error.

- uint32_t osMessageQueueGetMsgSize(osMessageQueueId_t mq_id)

  The function osMessageQueueGetMsgSize returns the maximum message size in bytes for the message queue object specified by parameter mq_id or 0 in case of an error.

Parameter		
mq_id	in	message queue ID obtained by osMessageQueueNew.

Return	
maximum message size in bytes	in case of success.
0	in case of error.

- uint32_t osMessageQueueGetCount(osMessageQueueId_t mq_id)

The function osMessageQueueGetCount returns the number of queued messages in the message queue object specified by parameter mq_id or 0 in case of an error.

Parameter		
mq_id	in	message queue ID obtained by osMessageQueueNew.

Return	
number of queued messages	in case of success.
0	in case of error.

- uint32_t osMessageQueueGetSpace(osMessageQueueId_t mq_id)

  The function osMessageQueueGetSpace returns the number available slots for messages in the message queue object specified by parameter mq_id or 0 in case of an error.

Parameter		
mq_id	in	message queue ID obtained by osMessageQueueNew.

Return	
number of available slots for messages	in case of success.
0	in case of error.

- osStatus_t osMessageQueueReset(osMessageQueueId_t mq_id)

  The function osMessageQueueReset resets the message queue specified by the parameter mq_id.

Parameter		
mq_id	in	message queue ID obtained by osMessageQueueNew.

Return

osOK	the message queue has been reset.
osErrorParameter	parameter mq_id is NULL or invalid.
osErrorResource	the message queue is in an invalid state.
osErrorISR	osMessageQueueReset cannot be called from interrupt service routines.

- osStatus_t osMessageQueueDelete(osMessageQueueId_t mq_id)

  The function osMessageQueueDelete deletes a message queue object specified by parameter mq_id. It releases internal memory obtained for message queue handling. After this call, the mq_id is no longer valid and cannot be used. The message queue may be created again using the function osMessageQueueNew.

Parameter		
mq_id	in	message queue ID obtained by osMessageQueueNew.

Return	
osOK	the message queue object has been deleted.
osErrorParameter	parameter mq_id is NULL or invalid.
osErrorISR	osMessageQueueDelete cannot be called from interrupt service routines.

2.10 Definitions

The following constants and enumerations are used by many CMSIS-RTOS function calls.

2.10.1 Macro Definitions

- #define osWaitForever 0xFFFFFFFFU

  A special timeout value that informs the RTOS to wait infinite until a resource becomes available. It applies to the following functions:

  - osDelay : Wait for Timeout (Time Delay).
  - osThreadFlagsWait : Wait for one or more Thread Flags of the current running thread to become signaled.
  - osEventFlagsWait : Wait for one or more Event Flags to become signaled.
  - osMutexAcquire : Acquire a Mutex or timeout if it is locked.
  - osSemaphoreAcquire : Acquire a Semaphore token or timeout if no tokens are available.
  - osMessageQueuePut : Put a Message into a Queue or timeout if Queue is full.
  - osMessageQueueGet : Get a Message from a Queue or timeout if Queue is empty.

- #define osFlagsWaitAny 0x00000000U

Reference:

- osEventFlagsWait
- osThreadFlagsWait

- #define osFlagsWaitAll 0x00000001U

  Reference:

  - osEventFlagsWait
  - osThreadFlagsWait

- #define osFlagsNoClear 0x00000002U

  Reference:

  - osEventFlagsWait
  - osThreadFlagsWait

2.10.2 Enumeration Types

- enum osStatus_t

  The osStatus_t enumeration defines the event status and error codes that are returned by many CMSIS-RTOS functions.

Enumerator	
osOK	Operation completed successfully.
osError	Unspecified RTOS error: run-time error but no other error message fits.
osErrorTimeout	Operation not completed within the timeout period.
osErrorResource	Resource not available.

osErrorParameter	Parameter error.
osErrorNoMemory	System is out of memory: it was impossible to allocate or reserve memory for the operation.
osErrorISR	Not allowed in ISR context: the function cannot be called from interrupt service routines.
osErrorNeedSched	Need reschedule because a higher priority task can be woken up.
osStatusReserved	Prevents enum down-size compiler optimization.