SCM1612
Wi-Fi 6 和 BLE 5 Low-Power SoC

CMSIS-FreeRTOS API 指南

文档版本 0.1
发布日期 2024-3-26

联系方式
速通半导体科技有限公司 (www.senscomm.com)
江苏省苏州市工业园区苏州大道西 2 号国际大厦 303 室
销售或技术支持，请发送电子邮件至
support@senscomm.com

免责声明和注意事项

本文档仅按"现状"提供。速通半导体有限公司保留在无需另行通知的情况下对其或本文档中包含的任何规格进行更正、改进和其他变更的权利。
与使用本文档中的信息有关的一切责任，包括侵犯任何专有权利的责任，均不予承认。此处不授予任何明示或暗示、通过禁止或其他方式对任何知识产权的许可。
本文档中的所有第三方信息均按"现状"提供，不对其真实性和准确性提供任何保证。
本文档中提及的所有商标、商号和注册商标均为其各自所有者的财产，特此确认。

版本历史

版本	日期	描述
0.1	2024-3-26	初始草案
1.0	2024-3-29	基本错误修正
1.1	2024-5-27	更新部分 api 的状态

目录

1 简介

SCM1612 SDK 集成了 CMSIS-FreeRTOS 作为一个 API，使用户能够在他们的应用程序中使用多种操作系统功能。虽然原生 FreeRTOS 的数据类型和函数仍然可以使用，但强烈推荐在应用层使用 CMSIS-FreeRTOS API。

1.1 概述

SCM1612 SDK 使用了 CMSIS-FreeRTOS 接口：

- 头文件
  - include/cmsis_os.h
- 定义
  - include/hal/cmsis_os2.h

应用程序应当引用 <include/cmsis_os.h> 而非 <include/hal/cmsis_os2.h>。

1.2 构建

无需特别设置即可启用 CMSIS-FreeRTOS API，因为它默认总是开启的。

2 参考

SCM1612 SDK 中的 CMSIS-FreeRTOS 基于 CMSIS-RTOS API v2 API ，分为以下几个类别：

- 内核信息和控制
- 线程管理
- 线程标志
- 事件标志
- 通用等待函数
- 定时器管理
- 互斥锁管理
- 信号量
- 内存池 (*)
- 消息队列

这些类别将在下文中详细介绍。

(*) 当前不支持。

2.1 内核信息和控制

内核信息和控制函数组允许进行以下操作：

- 获取有关系统和底层内核的信息。
- 获取 CMSIS-RTOS API 的版本信息。
- 初始化 RTOS 内核以创建对象。
- 启动 RTOS 内核和线程切换。
- 检查 RTOS 内核的执行状态。

内核信息和控制函数不能从中断服务例程 (ISR) 中调用。

2.1.1 数据结构

- struct osVersion_t

标识底层 RTOS 内核和 API 版本号。
版本以组合十进制数的格式表示：主版本号.次版本号.修订号: mmnnnrrrr

数据字段		
uint32_t	api	API 版本（主版本号.次版本号.修订号: mmnnnrrrr 十进制）。
uint32_t	kernel	内核版本（主版本号.次版本号.修订号: mmnnnrrrr 十进制）。

使用 osKernelGetInfo 来检索版本号。

2.1.2 枚举类型

- Enum osKernelState_t

通过 osKernelGetState 检索的内核状态。
如果 osKernelGetState 失败或者在 ISR 中调用，它将返回 osKernelError，否则返回内核状态。

枚举器	
osKernelInactive	非活动。 内核尚未准备好。需要成功执行 osKernelInitialize。
osKernelReady	准备就绪。 内核尚未运行。osKernelStart 将内核转移到运行状态。
osKernelRunning	运行中。 内核已初始化并运行。

osKernelLocked	锁定。 内核已使用 osKernelLock 锁定。函数 osKernelUnlock 或 osKernelRestoreLock 解锁它。
osKernelSuspended	挂起。 内核使用 osKernelSuspend 挂起。函数 osKernelResume 返回正常操作。
osKernelError	错误。 发生错误。
osKernelReserved	防止枚举下调编译器优化。 保留。

2.1.3 函数

- osStatus_t osKernelInitialize(void)

函数 osKernelInitialize 初始化 RTOS 内核。
在成功执行之前，只能调用函数 osKernelGetInfo 和 osKernelGetState。

返回值	
osOK	成功时。
osError	发生未指定的错误时。

osErrorISR	如果从 ISR 中调用。
osErrorNoMemory	如果无法为操作保留内存。

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

  函数 osKernelGetInfo 检索底层 RTOS 内核的 API 和内核版本以及内核的人类可读标识符字符串。在初始化或启动 RTOS（调用 osKernelInitialize 或 osKernelStart）之前可以安全地调用它。

参数		
version	输出参数	指向用于检索版本信息的缓冲区的指针。
Id_buf	输出参数	指向用于检索内核标识字符串的缓冲区的指针。
Id_size	输入参数	内核标识字符串的缓冲区大小。

返回值	
osOK	成功时。
osError	发生未指定的错误时。
osErrorISR	如果从 ISR 中调用。

- osKernelState_t osKernelGetState(void)

  函数 osKernelGetState 返回内核的当前状态，并且在初始化或启动 RTOS（调用 osKernelInitialize 或 osKernelStart）之前可以安全地调用。如果失败，它将返回 osKernelError，否则返回内核状态（参见 osKernelState_t 以获取内核状态列表）。

返回值	
osKernelError	如果从 ISR 中调用。
实际内核状态	其它场景。

- osStatus_t osKernelStart(void)

  函数 osKernelStart 启动 RTOS 内核并开始线程切换。在成功的情况下，它不会返回到其调用函数。在成功执行之前，只能调用函数 osKernelGetInfo、osKernelGetState 和对象创建函数（osXxxNew）。

在 osKernelStart 之前，至少应创建一个初始线程，请参见 osThreadNew。

返回值	
osOK	成功时。
osError	发生未指定的错误时。
osErrorISR	如果从 ISR 中调用。

- int32_t osKernelLock(void)

函数 osKernelLock 允许锁定所有任务切换。它返回锁定状态的前一个值（如果锁定，则为 1；如果未锁定，则为 0），或者否则返回表示错误代码的负数（参见 osStatus_t）。

返回值	
osError	发生未指定的错误时。
osErrorISR	如果从 ISR 中调用。
锁定状态的前一个值	其它场景。

- int32_t osKernelUnlock(void)

函数 osKernelUnlock 从 osKernelLock 恢复。它返回锁定状态的前一个值（如果锁定，则为 1；如果未锁定，则为 0），或者否则返回表示错误代码的负数（参见 osStatus_t）。

返回值	
osError	发生未指定的错误时。
osErrorISR	如果从 ISR 中调用。
锁定状态的前一个值	其它场景。

- int32_t osKernelRestoreLock(int32_t lock)

函数 osKernelRestoreLock 在 osKernelLock 或 osKernelUnlock 之后恢

复前一个锁定状态。 参数 lock 指定了由 osKernelLock 或 osKernelUnlock 获得的锁定状态。
函数返回锁定状态的新值（如果锁定，则为 1；如果未锁定，则为 0），或者否则返回表示错误代码的负数（参见 osStatus_t）。

返回值	
osError	osError 发生未指定的错误时。
osErrorISR	osErrorISR 如果从 ISR 中调用。
新的锁定状态	其它场景。

- uint32_t osKernelSuspend(void)

  不支持。

- uint32_t osKernelResume(void)

  不支持。

- uint32_t osKernelGetTickCount(void)

  函数 osKernelGetTickCount 返回当前 RTOS 内核的 tick 计数。

- uint32_t osKernelGetTickFreq(void)

  函数 osKernelGetTickFreq 返回当前 RTOS 内核 tick 的频率。

- uint32_t osKernelGetSysTimerCount(void)

  函数 osKernelGetSysTimerCount 返回当前 RTOS 内核系统计时器的值，为 32 位值。该值是由内核系统中断计时器值和计数这些中断的计数器（RTOS 内核 tick）组成的滚动 32 位计数器。

  该函数允许实现低于 RTOS tick 粒度的非常短的超时检查。在设备或外设初始化例程中检查忙状态时，可能需要进行此类检查。

- uint32_t osKernelGetSysTimerFreq(void)

  函数 osKernelGetSysTimerFreq 返回当前 RTOS 内核系统计时器的频率。

2.2 线程管理

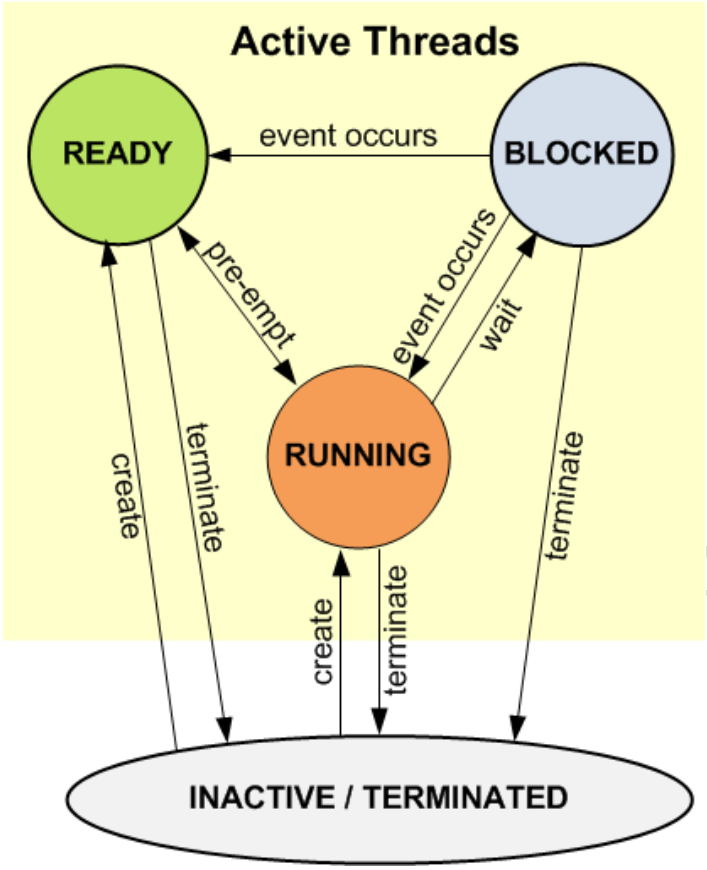
线程管理功能组允许在系统中定义、创建和控制线程功能。

线程可以处于以下状态：

- 运行中 (RUNNING)：当前运行的线程处于 RUNNING 状态。一次只能有一个线程处于此状态。
- 就绪 (READY)：准备运行的线程处于 READY 状态。一旦运行中的线程终止或被阻塞，具有最高优先级的下一个就绪线程成为运行中的线程。
- 阻塞 (BLOCKED)：被延迟、等待事件发生或被挂起的线程处于 BLOCKED 状态。
- 终止 (TERMINATED)： 调用 osThreadTerminate 时，线程被终止，资源尚未释放（适用于可连接的线程）。
- 非活动 (INACTIVE)： 未创建或已终止且所有资源已释放的线程处于 INACTIVE 状态。

线程状态的变化如下：

- 使用函数 osThreadNew 创建线程。这会将线程置于就绪或运行中状态（取决于线程优先级）。
- CMSIS-RTOS 是抢占式的。具有最高优先级的活动线程成为运行中的线程，前提是它不等待任何事件。线程的初始优先级由 osThreadAttr_t 定义，但可以在执行过程中使用 osThreadSetPriority 函数更改。
- 运行中的线程在延迟、等待事件或被挂起时转入阻塞状态。
- 活动线程可以随时使用 osThreadTerminate 函数终止。线程也可以通过从线程函数返回来终止。被终止的线程处于非活动状态，通常不消耗任何动态内存资源。



2.2.1 数据结构

- struct osThreadAttr_t

数据字段		
const char *	name	线程的名称 指向线程对象的可读名称（在调试期间显示）的常量字符串的指针。 默认值：NULL 没有指定名称（调试器可能显示函数名称）。
uint32_t	attr_bits	属性位

		可以使用以下位掩码来设置选项： • osThreadDetached: 以分离模式创建线程（默认）。 • osThreadJoinable: 以可连接模式创建线程。
void *	cb_mem	控制块的内存 指向线程控制块对象的内存的指针。 默认值：NULL 使用自动动态分配线程控制块的内存。
uint32_t	cb_size	提供的控制块内存的大小 用 cb_mem 传递的内存块的大小（以字节为单位）。 默认值：0 表示没有使用 cb_mem 提供的内存。
void *	stack_mem	栈的内存 指向线程栈的内存位置的指针（64 位对齐）。 默认值：NULL 使用线程栈管理从固定大小的内存池分配栈
uint32_t	stack_size	栈的大小 指定的栈的大小（以字节为单位）。 默认值：0 表示没有使用 stack_mem 提供的内存。
osPriority_t	priority	初始线程优先级（默认：osPriorityNormal） 使用 osPriority_t 的值指定初始线程优先级。 默认值：osPriorityNormal。
TZ_ModuleId_t	tz_module	模块标识符。

		TrustZone 线程上下文管理标识符，用于为线程分配上下文内存。在非安全状态下运行的 RTOS 内核调用由头文件 TZ_context.h 定义的接口函数。对于根本不使用安全调用的线程，可以安全地设置为零。参见 TrustZone RTOS 上下文管理。 默认值：0 没有指定线程上下文。
uint32_t	reserved	保留（必须为 0） 保留以备将来使用。

2.2.2 类型定义

- void (*osThreadFunc_t)(void *argument)

  线程的入口函数。设置新线程（osThreadNew）将通过调用这个入口函数开始执行。可选的参数可用于将任意用户数据传递给线程，即用于识别线程或用于运行时参数。

参数		
argument	输入参数	在 osThreadNew 上设置的任意用户数据。

- osThreadId_t

  线程 ID 标识线程。

  由以下返回：

  - osThreadNew
  - osThreadGetId
  - osThreadEnumerate
  - osMutexGetOwner

2.2.3 枚举类型

- enum osThreadState_t

通过 osThreadGetState 检索的线程状态。如果 osThreadGetState 失败或从 ISR 调用，它将返回 osThreadError，否则返回线程状态。

枚举器	
osThreadInactive	非活动。 线程已创建但未被积极使用，或已被终止（用于静态控制块分配，当使用内存池时返回 osThreadError，因为控制块不再有效）
osThreadReady	就绪。 线程已准备好执行，但当前未运行。
osThreadRunning	运行中。 线程当前正在运行。
osThreadBlocked	阻塞。 线程当前被阻塞（延迟、等待事件或被挂起）。
osThreadTerminated	终止。 线程被终止，所有资源尚未释放（适用于可连接的线程）。
osThreadError	错误。 线程不存在（引发错误条件）且不能被调度。
osThreadReserved	

	防止枚举下调编译器优化。

- enum osPriority_t

  osPriority_t 值指定线程的优先级。默认线程优先级应为 osPriorityNormal。如果一个活动线程变为就绪，并且其优先级高于当前运行的线程，那么将立即发生线程切换。系统继续执行具有较高优先级的线程。

枚举器	
osPriorityNone	无优先级（未初始化）。
osPriorityIdle	保留给空闲线程。 这个最低优先级不应该用于任何其他线程。
osPriorityLow	优先级：低。
osPriorityLow1	优先级：低 + 1。
osPriorityLow2	优先级：低 + 2。
osPriorityLow3	优先级：低 + 3。
osPriorityLow4	优先级：低 + 4。
osPriorityLow5	优先级：低 + 5。
osPriorityLow6	优先级：低 + 6。
osPriorityLow7	

	优先级：低 + 7。
osPriorityBelowNormal	优先级：低于正常。
osPriorityBelowNormal1	优先级：低于正常 + 1。
osPriorityBelowNormal2	优先级：低于正常 + 2。
osPriorityBelowNormal3	优先级：低于正常 + 3。
osPriorityBelowNormal4	优先级：低于正常 + 4。
osPriorityBelowNormal5	优先级：低于正常 + 5。
osPriorityBelowNorma6	优先级：低于正常 + 6。
osPriorityBelowNorma7	优先级：低于正常 + 7。
osPriorityNormal	优先级：正常。
osPriorityNormal1	优先级：正常 + 1。
osPriorityNormal2	优先级：正常 + 2。
osPriorityNormal3	优先级：正常 + 3。
osPriorityNormal4	优先级：正常 + 4。

osPriorityNormal5	优先级：正常 + 5。
osPriorityNormal6	优先级：正常 + 6。
osPriorityNormal7	优先级：正常 + 7。
osPriorityAboveNormal	优先级：高于正常。
osPriorityAboveNormal1	优先级：高于正常 + 1。
osPriorityAboveNormal2	优先级：高于正常 + 2。
osPriorityAboveNormal3	优先级：高于正常 + 3。
osPriorityAboveNormal4	优先级：高于正常 + 4。
osPriorityAboveNormal5	优先级：高于正常 + 5。
osPriorityAboveNormal6	优先级：高于正常 + 6。
osPriorityAboveNormal7	优先级：高于正常 + 7。
osPriorityHigh	优先级：高。
osPriorityHigh1	优先级：高 + 1。
osPriorityHigh2	

	优先级：高 + 2。
osPriorityHigh3	优先级：高 + 3。
osPriorityHigh4	优先级：高 + 4。
osPriorityHigh5	优先级：高 + 5。
osPriorityHigh6	优先级：高 + 6。
osPriorityHigh7	优先级：高 + 7。
osPriorityRealtime	优先级：实时。
osPriorityRealtime1	优先级：实时 + 1。
osPriorityRealtime2	优先级：实时 + 2。
osPriorityRealtime3	优先级：实时 + 3。
osPriorityRealtime4	优先级：实时 + 4。
osPriorityRealtime5	优先级：实时 + 5。
osPriorityRealtime6	优先级：实时 + 6。
osPriorityRealtime7	优先级：实时 + 7。

osPriorityISR	保留给 ISR 延迟线程。 这个最高优先级可能被 RTOS 实现使用，但不得用于任何用户线程。
osPriorityError	系统无法确定优先级或优先级非法。
osPriorityReserved	防止枚举下调编译器优化。

2.2.4 函数

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

  函数 osThreadNew 通过将线程函数添加到活动线程列表并将其设置为 READY 状态来启动线程函数。使用参数指针 *argument 传递线程函数的参数。当创建的线程函数的优先级高于当前 RUNNING 线程时，创建的线程函数立即启动并成为新的 RUNNING 线程。使用参数指针 attr 定义线程属性。属性包括线程优先级、栈大小或内存分配的设置。

  在 RTOS 启动之前（调用 osKernelStart）可以安全地调用该函数，但在初始化之前（调用 osKernelInitialize）不能调用。

  函数 osThreadNew 返回指向线程对象标识符的指针，如果出错，则返回 NULL。

参数		
func	输入参数	线程函数。
argument	输入参数	传递给线程函数作为开始参数的指针。
attr	输入参数	线程属性；NULL：默认值。

返回值	
thread ID	成功时。
NULL	出错时。

- const char *osThreadGetName(osThreadId_t thread_id)

  函数 osThreadGetName 返回由参数 thread_id 标识的线程的名称字符串的指针，如果出错，则返回 NULL。

参数		
thread_id	输入参数	由 osThreadNew 或 osThreadGetId 获得的线程 ID。

返回值	
以 null 结尾的线程名称字符串	成功时。
NULL	出错时。

- osThreadId_t osThreadGetId(void)

  函数 osThreadGetId 返回当前运行线程的线程对象 ID，如果出错，则返回 NULL。

返回值	
thread ID	成功时。
NULL	出错时。

- osThreadState_t osThreadGetState(osThreadId_t thread_id)

  函数 osThreadGetState 返回由参数 thread_id 标识的线程的状态。如果失败或从 ISR 中调用，它将返回 osThreadError，否则返回线程状态（参见 osThreadState_t 以获取线程状态列表）。

参数

thread_id	输入参数	由 osThreadNew 或 osThreadGetId 获得的线程 ID。

返回值	
当前线程状态	成功时。
osTheadError	出错时。

- osStatus_t osThreadSetPriority(osThreadId_t thread_id, osPriority_t priority)

函数 osThreadSetPriority 更改由参数 thread_id 指定的活动线程的优先级，将其更改为参数 priority 指定的优先级。

参数		
thread_id	输入参数	由 osThreadNew 或 osThreadGetId 获得的线程 ID。
priority	输入参数	线程函数的新优先级值。

Return	
osOK	指定线程的优先级已成功更改。
osErrorParameter	为 NULL 或无效或优先级不正确。
osErrorResource	线程处于无效状态。
osErrorISR	osThreadSetPriority 不能从中断服务例程中调用。

- osPriority_t osThreadGetPriority(osThreadId_t thread_id)

函数 osThreadGetPriority 返回由参数 thread_id 指定的活动线程的优先级。

参数		
thread_id	输入参数	由 osThreadNew 或 osThreadGetId 获得的线程 ID。

返回值	
priority	指定线程的优先级。

osPriorityError	无法确定优先级或优先级非法。当函数从中断服务例程中调用时也返回此值。

- osStatus_t osThreadYield(void)

函数 osThreadYield 将控制权传递给处于 READY 状态的具有相同优先级的下一个线程。如果没有处于 READY 状态的具有相同优先级的其他线程，则当前线程继续执行，不会发生线程切换。osThreadYield 不会将线程设置为 BLOCKED 状态。因此，即使有处于 READY 状态的线程，也不会调度具有较低优先级的线程。

当内核被锁定时调用此函数时，不会产生影响，参见 osKernelLock。

返回值	
osOK	控制权已成功传递给下一个线程。
osError	发生了未指定的错误。
osErrorISR	函数 osThreadYield 不能从中断服务例程中调用。

- osStatus_t osThreadSuspend(osThreadId_t thread_id)

函数 osThreadSuspend 暂停由参数 thread_id 标识的线程的执行。线程被置于 BLOCKED 状态 (osThreadBlocked)。暂停运行的线程将立即导致切换到另一个处于 READY 状态的线程。被暂停的线程在使用函数 osThreadResume 明确恢复之前不会执行。

已经处于 BLOCKED 状态的线程将从任何等待列表中移除，并在恢复时变为就绪。因此，不建议暂停已经被阻塞的线程。

当内核被锁定时，不得调用此函数来暂停运行的线程，即 osKernelLock。

参数		
thread_id	输入参数	由 osThreadNew 或 osThreadGetId 获得的线程 ID。

返回值	
osOK	线程已成功暂停。

osErrorParameter	thread_id 为 NULL 或无效。
osErrorResource	线程处于无效状态。
osErrorISR	函数 osThreadSuspend 不能从中断服务例程中调用。

- osStatus_t osThreadResume(osThreadId_t thread_id)

  函数 osThreadResume 将由参数 thread_id 标识的线程（必须处于 BLOCKED 状态）恢复到 READY 状态。如果恢复的线程的优先级高于运行中的线程，则立即发生上下文切换。

  线程变为就绪，无论阻塞线程的原因如何。因此，不建议恢复未被 osThreadSuspend 暂停的线程。

  会将线程置于 BLOCKED 状态的函数包括：osEventFlagsWait 和 osThreadFlagsWait, osDelay 和 osDelayUntil, osMutexAcquire 和 osSemaphoreAcquire, osMessageQueueGet, osMemoryPoolAlloc, osThreadJoin, osThreadSuspend。

  当内核被锁定（osKernelLock）时，可以调用此函数。在这种情况下，潜在的上下文切换将被延迟，直到内核解锁，即 osKernelUnlock 或 osKernelRestoreLock。

参数		
thread_id	**输入参数**	由 osThreadNew 或 osThreadGetId 获得的线程 ID。

Return	
osOK	线程已成功恢复。
osErrorParameter	thread_id 为 NULL 或无效。
osErrorResource	线程处于无效状态。
osErrorISR	函数 osThreadResume 不能从中断服务例程中调用。

- osStatus_t osThreadDetach(osThreadId_t thread_id)

不支持。

- osStatus_t osThreadJoin(osThreadId_t thread_id)

  不支持。

- __NO_RETURN void osThreadExit(void)

  函数 osThreadExit 终止调用线程。这允许线程与 osThreadJoin 同步。

- osStatus_t osThreadTerminate(osThreadId_t thread_id)

  函数 osThreadTerminate 从活动线程列表中移除由参数 thread_id 指定的线程。如果线程当前正在运行，线程将终止，并且执行将继续下一个 READY 线程。如果不存在这样的线程，函数将不会终止运行的线程，而是返回 osErrorResource。

  避免使用不存在或已被终止的线程 ID 调用该函数。
  osThreadTerminate 销毁非可连接线程并从系统中移除它们的 thread_id。后续访问 thread_id（例如 osThreadGetState）将返回 osThreadError。可连接线程将不会被销毁，并在与 osThreadJoin 结合使用后返回状态 osThreadTerminated。

参数		
thread_id	**输入参数**	由 osThreadNew 或 osThreadGetId 获得的线程 ID。

返回值	
osOK	指定线程已成功从活动线程列表中移除。
osErrorParameter	thread_id 为 NULL 或无效。
osErrorResource	线程处于无效状态或不存在其他 READY 线程。
osErrorISR	函数 osThreadTerminate 不能从中断服务例程中调用。

- uint32_t osThreadGetStackSize(osThreadId_t thread_id)

  不支持。

- uint32_t osThreadGetStackSpace(osThreadId_t thread_id)

  函数 osThreadGetStackSpace 返回由参数 thread_id 指定的线程的未使用栈空间的大小。需要启用执行期间的栈水印记录（参见线程配置）。如果出错，返回 0。

参数		
thread_id	输入参数	由 osThreadNew 或 osThreadGetId 获得的线程 ID。

返回值	
剩余栈大小（以字节为单位）	成功时。
0	出错时。

- uint32_t osThreadGetCount(void)

  函数 osThreadGetCount 返回活动线程的数量，如果出错，则返回 0。

返回值	
活动线程的数量	成功时。
0	出错时。

- uint32_t osThreadEnumerate(osThreadId_t *thread_array, uint32_t array_items)

  函数 osThreadEnumerate 返回枚举的线程数量，如果出错，则返回 0。

参数		
thread_array	输出参数	指向用于检索线程 ID 的数组的指针。

array_items	输入参数	用于检索线程 ID 的数组中的最大项目数。

返回值	
枚举的线程数量	成功时。
0	出错时。

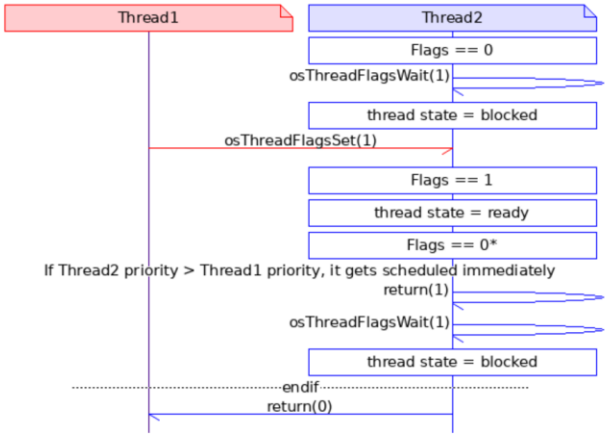
2.3 线程标志

线程标志是事件标志的更专业版本。参见事件标志。虽然事件标志可以用来全局信号通知多个线程，但线程标志只发送给一个特定的线程。每个线程实例都可以接收线程标志，无需额外分配线程标志对象。 线程标志管理函数不能从中断服务例程中调用，osThreadFlagsSet 除外。

使用示例

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 函数

- uint32_t osThreadFlagsSet(osThreadId_t thread_id, uint32_t flags)

osThreadFlagsSet 函数用于设置参数 thread_id 指定的线程的线程标志。线程将返回存储在线程控制块中的标志，如果最高位被设置，则返回错误代码（请参阅标志函数错误代码）。请查阅下面的使用示例，以了解返回值的计算方式。

等待设置标志的目标线程将从 BLOCKED 状态恢复。

参数		
thread_id	输入参数	通过 osThreadNew 或 osThreadGetId 获得的线程 ID。
flags	输入参数	指定应设置的线程标志。

返回值	
设置后的线程标志。	成功后。
osFlagsErrorUnknown	未指定的错误。
osFlagsErrorParameter	参数 thread_id 不是有效的线程，或者 flags 的最高位已设置。
osFlagsErrorResource	线程处于无效状态。

- uint32_t osThreadFlagsClear(uint32_t flags)

函数 osThreadFlagsClear 用于清除当前运行线程的指定标志。它返回清除前的标志，或者如果最高位被设置，则返回错误代码（请参考标志函数错误代码）。

参数		
flags	输入参数	指定需要清除标志的线程。

返回值	
返回清除前的线程标志。	成功后。

osFlagsErrorUnknown	未指定的错误，即不是从运行线程的上下文中调用。
osFlagsErrorParameter	参数 flags 的最高位已设置。
osFlagsErrorISR	函数 osThreadFlagsClear 不能从中断服务程序中调用。

- uint32_t osThreadFlagsGet(void)

  函数 osThreadFlagsGet 返回当前运行线程当前设置的标志。如果在没有活动且当前运行线程的情况下调用 osThreadFlagsGet，则返回零。

返回值	
当前线程的标志	

- uint32_t osThreadFlagsWait(uint32_t flags, uint32_t options, uint32_t timeout)
  函数 osThreadFlagsWait 会暂停当前正在运行的线程，直到参数 flags 中指定的任一或全部线程标志被设置。如果这些线程标志已经被设置，该函数将立即返回；否则，线程将进入阻塞状态。

  参数 options 用于指定等待条件：

选项	
osFlagsWaitAny	等待任意一个标志（默认）。
osFlagsWaitAll	等待所有标志。
osFlagsNoClear	不清除已指定等待的标志。

  如果 options 中设置了 osFlagsNoClear，可以使用 osThreadFlagsClear 函数手动清除标志。否则，osThreadFlagsWait 函数将自动清除已等待的标志。

  参数 timeout
  表示计时器滴答数，是一个上限值。实际的延迟时间取决于自上次计时器滴答以来经过的时间。

参数		
flags	输入参数	指定要等待的标志。
options	输入参数	指定标志选项（osFlagsXxxx）。
timeout	输入参数	超时值；如果没有超时则为 0。

返回值	
清除线程标志之前的标志。	成功后。
osFlagsErrorUnknown	未指定的错误，即不是从运行线程的上下文中调用。
osFlagsErrorParameter	参数 flags 的最高位已设置。
osFlagsErrorTimeout	在给定时间内未设置待定的标志。
osFlagsErrorResource	在未指定超时的情况下未设置待定的标志。

2.4 事件标志

CMSIS-RTOS 中的事件标志管理功能允许您对事件标志进行控制或等待操作。每个线程最多可以设置 31 个事件标志。

一个线程具有以下操作事件标志的能力:

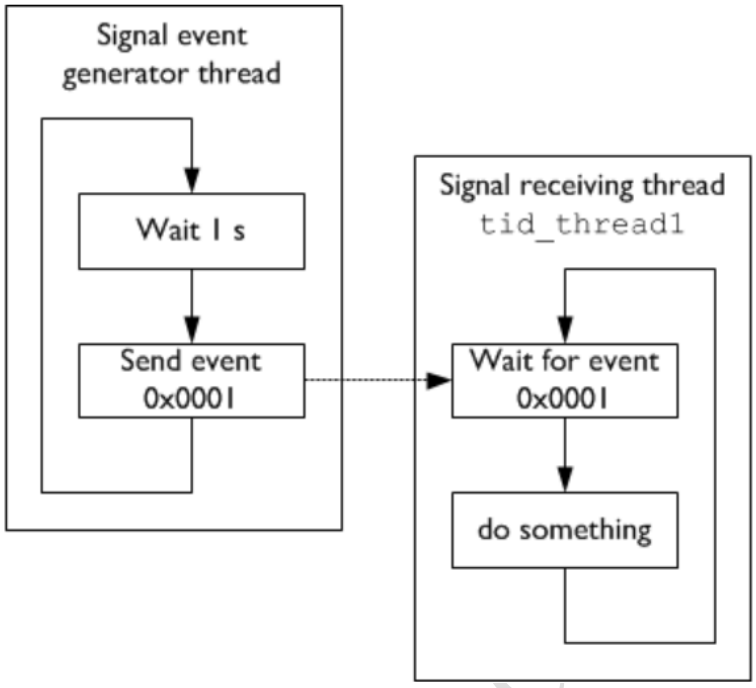
- 等待特定的事件标志被设置(使用 osEventFlagsWait 函数),在此期间线程进入阻塞(BLOCKED)状态。
- 在任意其他线程中设置一个或多个事件标志(使用 osEventFlagsSet 函数)。
- 清除自身或其他线程的事件标志(使用 osEventFlagsClear 函数)。

当一个阻塞的线程被唤醒恢复执行时,其拥有的事件标志将自动清除(除非指定了 osFlagsNoClear 事件标志选项来保留标志状态)。

osEventFlagsSet、osEventFlagsClear、osEventFlagsGet 和 osEventFlagsWait

等函数可在中断服务例程中调用。

基于事件标志的线程同步操作:



2.4.1 数据结构

- struct osEventFlagsAttr_t

数据字段		
const char *	name	事件标志对象的名称字符串指针 指向一个常量字符串,用于在调试时显示事件标志对象的可读名称。 默认值:NULL,表示未指定对象名称。

uint32_t	attr_bits	属性位 保留供将来使用,必须设置为 0 以保证向后兼容。
void *	cb_mem	事件标志控制块内存指针 指向用于存放事件标志控制块对象的内存区域。 默认值:NULL,表示使用内核自动动态分配内存。
uint32_t	cb_size	控制块内存区域大小(字节) 通过 cb_mem 参数传入的内存区域大小(以字节为单位)。 默认值:0,表示未提供 cb_mem 内存区域。

2.4.2 类型定义

- osEventFlagsId_t

  事件标志 ID 类型,用于标识一个事件标志对象。

2.4.3 函数

- osEventFlagsId_t osEventFlagsNew(const osEventFlagsAttr_t *attr)

  该函数用于创建一个新的事件标志对象，该对象用于在线程间传递事件，并返回指向该事件标志对象标识符的指针，或在发生错误时返回 NULL。在 RTOS 启动之前（调用 osKernelStart）可以安全地调用此函数，但不能在初始化之前（调用 osKernelInitialize）调用。

参数 attr 用于设置事件标志的属性（参见 osEventFlagsAttr_t）。如果设置为 NULL，则使用默认属性，即使用内核内存分配用于事件控制块。

参数		
attr	输入	事件标志属性；NULL 表示使用默认值。

返回值	
事件标志 ID。	成功后。
NULL	出错时。

- uint32_t osEventFlagsSet(osEventFlagsId_t ef_id, uint32_t flags)

  该函数用于设置由参数 flags 指定的事件标志，这些标志位于由参数 ef_id 指定的事件标志对象中。

  处于阻塞状态的具有最高优先级的线程将被通知恢复执行。函数返回存储在事件控制块中的事件标志或错误代码（最高位设置，参见标志函数错误代码）。当给 osEventFlagsWait 调用提供选项 osFlagsNoClear 时，可能会按优先级顺序唤醒更多线程。

参数		
ef_id	输入	通过 osEventFlagsNew 获取的事件标志 ID。
flags	输入	指定应设置的标志。

返回值	
设置后的事件标志	成功后。
osFlagsErrorUnknown	未指定错误。
osFlagsErrorParameter	参数 ef_id 未识别为有效的事件标志对象，或 flags 的最高位设置。
osFlagsErrorResource	事件标志对象处于无效状态。

- uint32_t osEventFlagsClear(osEventFlagsId_t ef_id, uint32_t flags)

该函数用于清除由参数 flags 指定的事件标志，这些标志位于由参数 ef_id 指定的事件标志对象中。函数返回清除前的事件标志或错误代码。

参数		
ef_id	输入	通过 osEventFlagsNew 获取的事件标志 ID。
flags	输入	指定应清除的标志。

返回值	
清除前的事件标志	成功时。
osFlagsErrorUnknown	未指定错误。
osFlagsErrorParameter	参数 ef_id 未识别为有效的事件标志对象，或 flags 的最高位设置。
osFlagsErrorResource	事件标志对象处于无效状态。

- uint32_t osEventFlagsGet(osEventFlagsId_t ef_id)

  该函数返回由参数 ef_id 指定的事件标志对象中当前设置的事件标志，或在出现错误时返回 0。

参数		
ef_id	输入	通过 osEventFlagsNew 获取的事件标志 ID。

返回值	
当前事件标志	成功后。
0	错误时。

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

  该函数暂停当前运行中的线程，直到由参数 flags 指定的任意或全部事件标志在由参数 ef_id 指定的事件对象中被设置。如果这些事件标志已经被设置，函数会立即返回。否则，线程会进入阻塞状态。

options 参数指定等待条件：

选项值:	
osFlagsWaitAny	等待任意标志（默认）。
osFlagsWaitAll	等待所有标志。
osFlagsNoClear	不清除已指定等待的标志。

如果在 options 中设置了 osFlagsNoClear，可以使用 osEventFlagsClear 手动清除标志。

timeout 参数指定系统等待事件标志的时间。在等待期间，调用此函数的线程会进入阻塞状态。timeout 参数可以有以下值：

- 当 timeout 为 0 时，函数立即返回（即尝试行为）。
- 当 timeout 设置为 osWaitForever 时，函数将无限期等待直到事件标志变为可用（即等待行为）。
- 所有其他值指定超时的内核滴答数（即定时等待行为）。

参数:		
ef_id	输入	通过 osEventFlagsNew 获取的事件标志 ID。
flags	输入	指定要等待的标志。
options	输入	指定标志选项（osFlagsXxxx）。
timeout	输入	超时值，或在无超时的情况下为 0。

返回值	
清除前的事件标志	成功时。
osFlagsErrorUnknown	未指定错误。
osFlagsErrorTimeout	在给定时间内未设置所等待的标志。
osFlagsErrorParameter	参数 ef_id 未识别为有效的事件标志对象，或 flags 的最高位设置。
osFlagsErrorResource	在未指定超时时未设置所等待的标志。

- osStatus_t osEventFlagsDelete(osEventFlagsId_t ef_id)

该函数删除由参数 ef_id 指定的事件标志对象，并释放用于事件标志处理的内部内存。此调用后，ef_id 不再有效且不能使用。这可能导致等待此事件对象标志的线程饥饿。ef_id 可以使用函数 osEventFlagsNew 再次创建。

参数		
ef_id	输入	通过 osEventFlagsNew 获取的事件标志 ID。

返回值	
osOK	指定的事件标志对象已被删除。
osErrorISR	osEventFlagsDelete 不能从中断服务例程调用。
osFlagsErrorParameter	参数 ef_id 为 NULL 或无效。
osFlagsErrorResource	事件标志对象处于无效状态。

- const char * osEventFlagsGetName(osEventFlagsId_t ef_id)

该函数返回由参数 ef_id 标识的事件标志对象的名称字符串的指针，或在出现错误时返回 NULL。

参数		
ef_id	输入	通过 osEventFlagsNew 获取的事件标志 ID。

返回值	
返回名称的以 null 结尾的字符串	成功时。
NULL	错误时。

该 API 目前不支持。

2.5 通用等待功能

通用等待功能提供了时间延迟的手段。

2.5.1 函数

- osStatus_t osDelay(uint32_t ticks)

  该函数等待由内核滴答数指定的时间段。对于值为 1 的情况，系统将等待直到下一个定时器滴答发生。实际的时间延迟可能比指定的少一个定时器滴答，即如果在下一个系统滴答发生之前立即调用 osDelay(1)，线程将立即被重新调度。

  被延迟的线程将进入阻塞状态，并立即发生上下文切换。在给定的滴答数经过后，线程自动回到就绪状态。如果线程在就绪状态下具有最高优先级，它将立即被调度。

参数		
ticks	输入	时间滴答值。

返回值	
osOK	时间延迟已执行。
osErrorParameter	时间不能被处理（零值）。
osErrorISR	osDelay 不能从中断服务例程调用。
osError	osDelay 不能执行（内核未运行或不存在就绪线程）。

- osStatus_t osDelayUntil(uint32_t ticks)

  该函数等待直到达到绝对时间（以内核滴答数指定）。

  当内核滴答计数器溢出时，osDelayUntil 会处理这种边界情况。因此，提供一个低于当前滴答值的值是完全合法的，例如由 osKernelGetTickCount 返回的值。通常作为用户，你不必关心溢出。你唯一需要记住的限制是最大延迟限制为 (2^31)-1 滴答。

被延迟的线程将进入阻塞状态，并立即发生上下文切换。在达到给定时间时，线程自动回到就绪状态。如果线程在就绪状态下具有最高优先级，它将立即被调度。

参数		
ticks	输入	绝对时间滴答数。

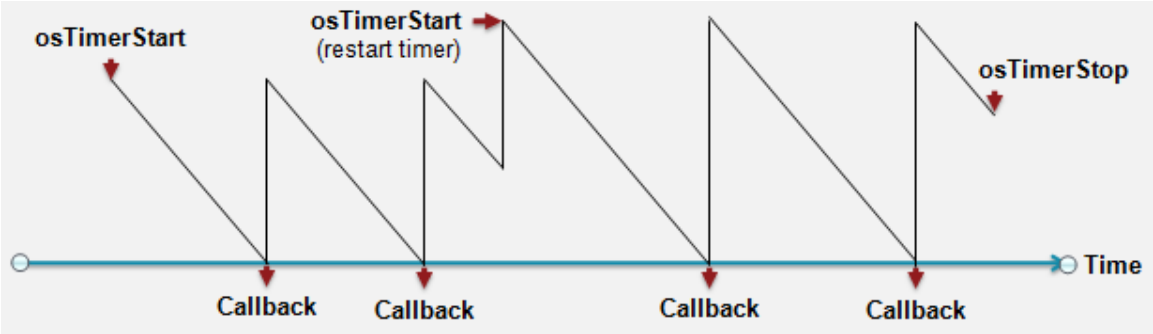
返回值	
osOK	时间延迟已执行。
osErrorParameter	时间不能被处理（超出范围）。
osErrorISR	osDelayUntil 不能从中断服务例程调用。
osError	osDelayUntil 不能执行（内核未运行或不存在就绪线程）。

2.6 定时器管理

除了通用等待功能外，CMSIS-RTOS 还支持虚拟定时器对象。这些定时器对象可以触发函数的执行（而不是线程）。当定时器到期时，将执行回调函数以运行与定时器相关的代码。每个定时器都可以配置为一次性定时器或周期性定时器。周期性定时器将重复其操作，直到被删除或停止。所有定时器都可以被启动、重启或停止。

定时器管理功能不能从中断服务例程中调用。

下图展示了周期性定时器的行为。对于一次性定时器，回调函数执行后定时器将停止。



使用 CMSIS-RTOS 软件定时器需要执行以下步骤:

- 定义定时器对象:

```
osTimerId_t one_shot_id, periodic_id;
```

- 定义定时器回调函数:

```
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
}
```

- 实例化并启动定时器:

```
// 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 数据结构

- struct osTimerAttr_t

数据字段		
const char *	name	定时器的名称 一个指向具有人类可读名称的常量字符串的指针（在调试期间显示）。 Default: NULL no name specified.
uint32_t	attr_bits	属性位 保留供将来使用（为了将来兼容性必须设置为 '0'）。
void *	cb_mem	控制块内存 指向定时器控制块对象的内存的指针。 默认值：NULL 表示使用自动动态分配的方式为定时器控制块分配内存。
uint32_t	cb_size	提供的控制块内存大小 通过 `cb_mem` 传递的内存块的大小（以字节为单位）。 默认值：0 表示默认情况下没有为控制块（`cb_mem`）提供内存。

2.6.2 类型定义

- osTimerId_t

  定时器 ID 类型,用于标识一个定时器对象。

- void(* osTimerFunc_t)(void *argument)

  定时器回调函数，每次定时器到期时都会调用。回调可能在专用定时器线程或中断上下文中执行，因此建议仅在定时器回调中使用 ISR 可调用函数。

参数		
argument	输入	传递给 osTimerNew 的参数。

2.6.3 枚举类型

- enum osTimerType_t

  指定 osTimerNew 中创建的定时器类型,包括一次性(osTimerOnce)和周期(osTimerPeriodic)两种。

枚举器	
osTimerOnce	一次性定时器。 定时器一旦到期就不会自动重新启动，可以根据需要使用 `osTimerStart` 手动重新启动。
osTimerPeriodic	重复定时器。 定时器会自动重复，并在运行时连续触发回调。

2.6.4 函数

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

  函数 osTimerNew 用于创建一个一次性或周期性定时器，并将其与一个带参数的回调函数关联。定时器在使用 osTimerStart 函数启动之前处于停止状态。该函数可以在 RTOS 启动之前（调用 osKernelStart）安全地调用，但不能在它初始化之前（调用 osKernelInitialize）调用。

  函数 osTimerNew 在成功时返回指向定时器对象标识符的指针，如果出现错误，则返回 NULL。

参数		
func	输入	指向回调函数的函数指针。
type	输入	osTimerOnce 表示一次性定时器， osTimerPeriodic 表示周期性定时器。
argument	输入	传递给定时器回调函数的参数。
attr	输入	定时器属性；如果为 NULL，则使用默认值。

返回值	
定时器 ID	成功时。
NULL	出错时。

- const char *osTimerGetName(osTimerId_t timer_id)

  该函数返回由参数 timer_id 标识的定时器的名称字符串的指针，或在出现错误时返回 NULL。

Parameter		
timer_id	输入	通过 osTimerNew 获取的定时器 ID。

返回值	
返回定时器名称的以 null 结尾的字符串	成功时。

NULL	出错时。

- osStatus_t osTimerStart(osTimerId_t timer_id, uint32_t ticks)

  该函数启动或重启由参数 timer_id 指定的定时器。参数 ticks 指定定时器的时间滴答值。

参数		
timer_id	输入	通过 osTimerNew 获取的定时器 ID。
ticks	输入	定时器的时间滴答值。

返回值	
osOK	指定的定时器已启动或重启。
osErrorISR	osTimerStart 不能从中断服务例程调用。
osErrorParameter	参数 timer_id 为 NULL 或无效，或者 ticks 不正确。
osErrorResource	定时器处于无效状态。
osErrorNeedSched	当前线程将被抢占。

- osStatus_t osTimerStop(osTimerId_t timer_id)

  该函数停止由参数 timer_id 指定的定时器。

参数		
timer_id	输入	通过 osTimerNew 获取的定时器 ID。

返回值	
osOK	指定的定时器已停止。
osErrorISR	osTimerStop 不能从中断服务例程调用。
osErrorParameter	参数 timer_id 为 NULL 或无效。
osErrorResource	定时器未运行（只能停止正在运行的定时器）。
osErrorNeedSched	当前线程将被抢占。

- uint32_t osTimerIsRunning(osTimerId_t timer_id)

  该函数检查由参数 timer_id 指定的定时器是否正在运行。如果定时器正在运行，则返回 1；如果定时器已停止或发生错误，则返回 0。

参数		
timer_id	输入	通过 osTimerNew 获取的定时器 ID。

返回值	
0	定时器已停止。
1	定时器正在运行。

- osStatus_t osTimerDelete(osTimerId_t timer_id)

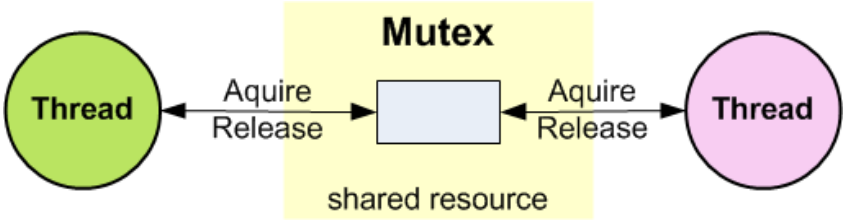
  该函数删除由参数 timer_id 指定的定时器。

参数		
timer_id	输入	通过 osTimerNew 获取的定时器 ID。

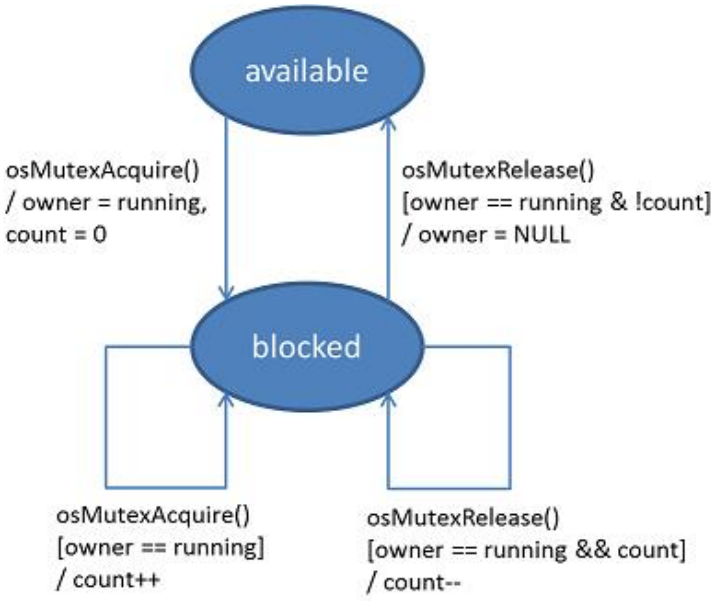
返回值	
osOK	指定的定时器已被删除。
osErrorISR	osTimerDelete 不能从中断服务例程调用。
osErrorParameter	参数 timer_id 为 NULL 或无效。
osErrorResource	定时器处于无效状态。

2.7 互斥锁管理

互斥锁（通常称为 Mutex）在各种操作系统中用于资源管理。在微控制器设备中，许多资源可以被反复使用，但在同一时间只能由一个线程使用（例如通信通道、内存和文件）。互斥锁用于保护对共享资源的访问。一个互斥锁被创建后，可以在不同的线程之间传递（它们可以获取和释放互斥锁）。



互斥锁是信号量的一种特殊形式。与信号量相似，它是一个令牌的容器。但与可以拥有多个令牌的信号量不同，互斥锁只能携带一个令牌（代表资源）。因此，互斥锁令牌是二元的且有界的，即它要么是可用的，要么被拥有线程阻塞。互斥锁的优势在于它引入了线程所有权的概念。当一个线程获取互斥锁并成为其所有者时，来自该线程的后续互斥锁获取将立即成功，无需任何延迟（如果指定了 osMutexRecursive）。因此，互斥锁的获取和释放可以嵌套。



与二进制信号量不同，后者可以从中断服务例程 (ISR) 中释放，互斥锁管理函数不能在中断服务例程 (ISR) 中调用。

2.7.1 数据结构

- struct osMutexAttr_t

数据字段

const char *	name	互斥锁的名称 指向一个常量字符串,用于在调试时显示互斥量对象的可读名称。 默认值: NULL,表示未指定对象名称。
uint32_t	attr_bits	属性位 可以使用以下位掩码来设置选项： osMutexRecursive: 线程可以多次获取互斥锁而不会锁定自己。 osMutexPrioInherit(*):拥有线程继承（更高优先级的）等待线程的优先级。 osMutexRobust(*):当拥有线程终止时，互斥锁会自动释放。 使用逻辑“或”运算来选择多个选项，例如： osMutexRecursive | osMutexPrioInherit; 默认值: 0,表示: 非递归互斥量:线程不能多次获取同一互斥量。 非优先级继承:所有者线程优先级不会改变。 互斥量不自动释放:所有者线程终止时,互斥量对象不会自动释放。 (*):不支持该选项

void *	cb_mem	控制块内存指针 指向用于存放互斥量控制块对象的内存区域。 默认值: NULL,表示使用自动动态内存分配方式分配控制块内存。
uint32_t	cb_size	控制块内存大小 通过 cb_mem 参数传入的内存区域大小(字节)。 默认值: 0,表示未提供 cb_mem 内存区域。

2.7.2 宏定义

- #define osMutexRecursive 0x00000001U

  osMutexAttr_t 中的递归标志。

  允许同一个线程多次获取互斥锁而不会造成死锁。每次互斥锁被获取时，锁计数会增加。互斥锁必须被释放相同的次数直到锁计数归零，此时互斥锁实际上被释放，其他线程可以获取它。

  递归锁的最大数量取决于实现，即用于锁计数的数据类型的大小。如果递归锁的最大数量耗尽，互斥锁的获取可能会失败。

- #define osMutexPrioInherit 0x00000002U

  osMutexAttr_t 中的优先级继承标志。

  使用优先级继承协议的互斥锁会将等待线程的优先级传递给当前的互斥锁所有者，如果所有者线程的优先级较低。这确保低优先级线程不会阻塞高优先级线程。

  否则，低优先级线程可能持有互斥锁但不会被执行，因为另一个中优先级线程正在运行。没有优先级继承，高优先级线程等待互斥锁会被中优先级线程阻塞，这种情况被称为优先级反转。

目前此标志不受支持。

- #define osMutexRobust 0x00000008U

  osMutexAttr_t 中的健壮标志。

  健壮的互斥锁在所有者线程被终止时（通过 osThreadExit 或 osThreadTerminate）会自动释放。非健壮的互斥锁不会自动释放，用户必须手动确保互斥锁的释放。

  目前此标志不受支持。

2.7.3 类型定义

- osMutexId_t

  互斥锁 ID 用于标识互斥锁。

2.7.4 函数

- osMutexId_t osMutexNew(const osMutexAttr_t *attr)

  该函数创建并初始化一个新的互斥锁对象，并返回指向互斥锁对象标识符的指针，或在出现错误时返回 NULL。在 RTOS 启动之前（调用 osKernelStart）可以安全地调用该函数，但在初始化之前（调用 osKernelInitialize）不能调用。

  参数 attr 设置互斥锁对象的属性（参考 osMutexAttr_t）。如果设置为 NULL，则使用默认属性。

参数		
attr	输入	互斥锁属性；如果为 NULL，则使用默认值。

返回值

互斥锁 ID	成功时。
NULL	出错时。

- const char *osMutexGetName(osMutexId_t mutex_id)

  该函数返回由参数 mutex_id 标识的互斥锁的名称字符串的指针，或在出现错误时返回 NULL。

参数		
mutex_id	输入	通过 osMutexNew 获取的互斥锁 ID。

返回值	
返回互斥锁名称的以 null 结尾的字符串	成功时。
NULL	出错时。

  该 API 目前不支持。

- osStatus_t osMutexAcquire(osMutexId_t mutex_id, uint32_t timeout)

  该阻塞函数等待由参数 mutex_id 指定的互斥锁对象变为可用。如果没有其他线程获取了互斥锁，函数立即返回并阻塞互斥锁对象。

  参数 timeout 指定系统等待获取互斥锁的时间。在等待期间，调用此函数的线程会进入阻塞状态。参数 timeout 可以有以下值：

  - 当 timeout 为 0 时，函数立即返回（即尝试行为）。
  - 当 timeout 设置为 osWaitForever 时，函数将无限期等待直到互斥锁变为可用（即等待行为）。
  - 所有其他值指定超时的内核滴答数（即定时等待行为）。

参数		
mutex_id	输入	通过 osMutexNew 获取的互斥锁 ID。
timeout	输入	超时值，或在无超时的情况下为 0。

返回值	
osOK	互斥锁已被获取。
osErrorTimeout	在给定时间内无法获取互斥锁。
osErrorISR	不能从中断服务例程调用。
osErrorParameter	参数 mutex_id 为 NULL 或无效。
osErrorResource	在未指定超时时无法获取互斥锁。

- osStatus_t osMutexRelease(osMutexId_t mutex_id)

  该函数释放由参数 mutex_id 指定的互斥锁。当前等待此互斥锁的其他线程将被置于就绪状态。

参数		
mutex_id	输入	通过 osMutexNew 获取的互斥锁 ID。

返回值	
osOK	互斥锁已正确释放。
osErrorISR	不能从中断服务例程调用。
osErrorParameter	参数 mutex_id 为 NULL 或无效。
osErrorResource	无法释放互斥锁（互斥锁未被获取或运行线程不是所有者）。

- osThreadId_t osMutexGetOwner(osMutexId_t mutex_id)

  该函数返回获取由参数 mutex_id 指定的互斥锁的线程的线程 ID。在出错的情况下或如果互斥锁没有被任何线程阻塞，它返回 NULL。

参数		
mutex_id	输入	通过 osMutexNew 获取的互斥锁 ID。

返回值	
返回所有者线程的线程 ID	成功时。
NULL	如果互斥锁未被获取。

- osStatus_t osMutexDelete(osMutexId_t mutex_id)

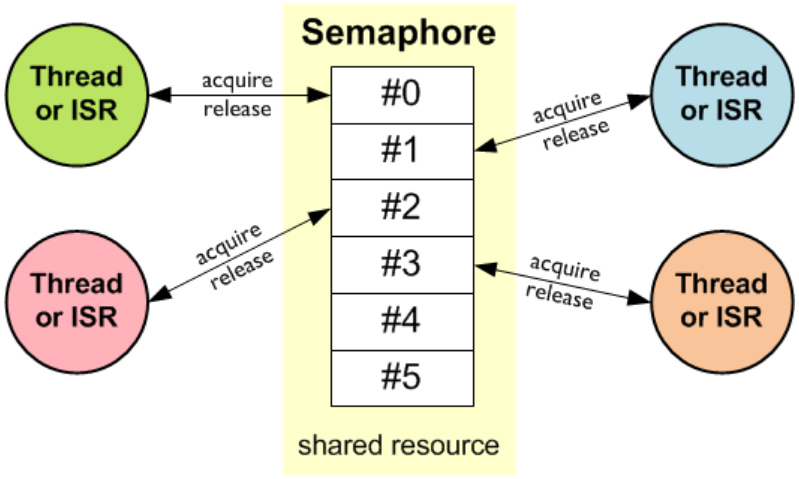
  该函数删除由参数 mutex_id 指定的互斥锁对象。它释放用于互斥锁处理的内部内存。此调用后，mutex_id 不再有效且不能使用。可以使用函数 osMutexNew 再次创建互斥锁。

参数		
mutex_id	输入	通过 osMutexNew 获取的互斥锁 ID。

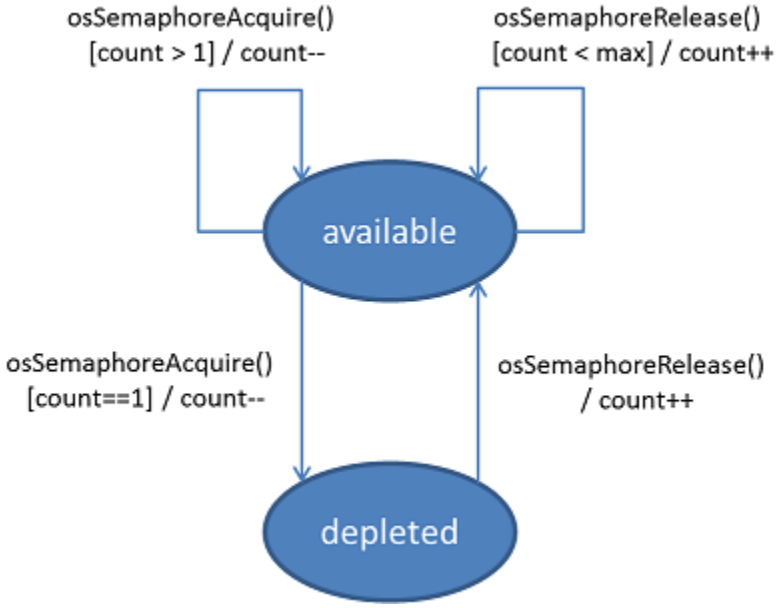
返回值	
osOK	互斥锁对象已被删除。
osErrorISR	不能从中断服务例程调用。
osErrorParameter	参数 mutex_id 为 NULL 或无效。

2.8 信号量

信号量用于管理和保护对共享资源的访问。信号量与互斥锁非常相似。不同的是，互斥锁一次只允许一个线程访问共享资源，而信号量可以用来允许一定数量的线程或中断服务例程 (ISR) 访问一组共享资源。通过使用信号量，可以有效地管理对一组相同外设的访问（例如多个 DMA 通道）。



信号量对象应该初始化为可用令牌的最大数量。这个可用资源的数量是作为 osSemaphoreNew 函数的参数指定的。每次通过 osSemaphoreAcquire 获取一个信号量令牌（处于可用状态），信号量的计数就会减少。
当信号量的计数为 0（即信号量耗尽状态）时，就无法再获取更多的信号量令牌。尝试获取信号量令牌的线程或中断服务例程 (ISR) 需要等待，直到下一个令牌释放。通过 osSemaphoreRelease 释放信号量会增加信号量的计数。



函数 osSemaphoreAcquire、osSemaphoreGetCount 和 osSemaphoreRelease 可以在中断服务例程中调用。

2.8.1 数据结构

- struct osSemaphoreAttr_t

数据字段		
const char *	name	信号量的名称 一个指向具有易读名称的常量字符串的指针（在调试期间显示）。 默认情况下，此指针为 NULL，表示未为信号量对象分配名称。
uint32_t	attr_bits	属性位 保留供将来使用（为了将来兼容性必须设置为 '0'）。
void *	cb_mem	控制块内存指针 指向信号量控制块对象的内存的指针。 默认值：NULL，表示使用自动动态分配的方式为信号量控制块分配内存。
uint32_t	cb_size	提供的控制块内存大小，通过 `cb_mem` 传递的内存块的大小（以字节为单位）。 默认值：0，表示默认情况下没有为控制块（`cb_mem`）提供内存。

2.8.2 类型定义

- osSemaphoreId_t

信号量 ID 用于标识信号量。

2.8.3 函数

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

  该函数创建并初始化一个用于管理对共享资源的访问的信号量对象，并返回指向信号量对象标识符的指针，或在出现错误时返回 NULL。在 RTOS 启动之前（调用 osKernelStart）可以安全地调用该函数，但在初始化之前（调用 osKernelInitialize）不能调用。

  参数 max_count 指定可用令牌的最大数量。max_count 值为 1 创建一个二进制信号量。

  参数 initial_count 设置可用令牌的初始数量。

  参数 attr 指定额外的信号量属性。如果设置为 NULL，则使用默认属性。

参数		
max_count	输入	可用令牌的最大数量。
initial_count	输入	可用令牌的初始数量。
attr	输入	信号量属性；如果为 NULL，则使用默认值。

返回值	
semaphore ID	成功时。
NULL	出错时。

- const char * osSemaphoreGetName(osSemaphoreId_t semaphore_id)

  该函数返回由参数 semaphore_id 标识的信号量的名称字符串的指针，或在出现错误时返回 NULL。

参数		
semaphore_id	输入	通过 osSemaphoreNew 获取的信号量 ID。

返回值	
name as null-terminated string	成功时。

NULL	出错时。

该 API 目前不支持。

- osStatus_t osSemaphoreAcquire(osSemaphoreId_t semaphore_id, uint32_t timeout)

该阻塞函数等待由参数 semaphore_id 指定的信号量对象的一个令牌变为可用。如果令牌可用，函数立即返回并递减令牌计数。

参数 timeout 指定系统等待获取令牌的时间。在等待期间，调用此函数的线程会进入阻塞状态。参数 timeout 可以有以下值：

- 当 timeout 为 0 时，函数立即返回（即尝试行为）。
- 当 timeout 设置为 osWaitForever 时，函数将无限期等待直到信号量变为可用（即等待行为）。
- 所有其他值指定超时的内核滴答数（即定时等待行为）。

如果参数 timeout 设置为 0，则可以在中断服务例程中调用。

参数		
semaphore_id	输入	通过 osSemaphoreNew 获取的信号量 ID。
timeout	输入	超时值，或在无超时的情况下为 0。

返回值	
osOK	令牌已被获取并且令牌计数递减。
osErrorTimeout	在给定时间内无法获取令牌。
osErrorParameter	参数 semaphore_id 为 NULL 或无效。
osErrorResource	在未指定超时时无法获取令牌。

- osStatus_t osSemaphoreRelease (osSemaphoreId_t semaphore_id)

该函数释放由参数 semaphore_id 指定的信号量对象的一个令牌。令牌只能被释放到创建时指定的最大计数，参见 osSemaphoreNew。当前等待此信号量对象的令牌的其他线程将被置于就绪状态。

参数

semaphore_id	输入	通过 osSemaphoreNew 获取的信号量 ID。

返回值	
osOK	令牌已被释放并且计数递增。
osErrorParameter	参数 semaphore_id 为 NULL 或无效。
osErrorResource	无法释放令牌（已达到最大令牌计数）。
osErrorNeedSched	当前线程将被抢占。

- uint32_t osSemaphoreGetCount(osSemaphoreId_t semaphore_id)

  该函数返回由参数 semaphore_id 指定的信号量对象的可用令牌数量。在出错的情况下返回 0。

参数		
semaphore_id	输入	通过 osSemaphoreNew 获取的信号量 ID。

返回值	
number of tokens available.	成功时。
0	出错时。

- osStatus_t osSemaphoreDelete(osSemaphoreId_t semaphore_id)

  该函数删除由参数 semaphore_id 指定的信号量对象。它释放用于信号量处理的内部内存。此调用后，semaphore_id 不再有效且不能使用。可以使用函数 osSemaphoreNew 再次创建信号量。

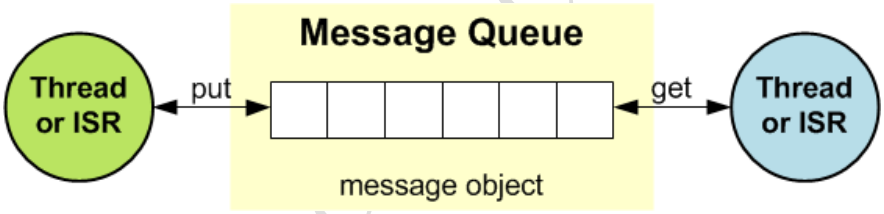
参数		
semaphore_id	输入	通过 osSemaphoreNew 获取的信号量 ID。

返回值

osOK	信号量对象已被删除。
osErrorParameter	参数 semaphore_id 为 NULL 或无效。
osErrorISR	不能从中断服务例程调用。

2.9 消息队列

消息传递是线程之间另一种基本的通信方式。在消息传递模型中，一个线程显式地发送数据，而另一个线程接收它。这种操作更像是某种输入/输出（I/O）操作，而不是直接访问要共享的信息。在 CMSIS-RTOS 中，这种机制被称为消息队列。数据以类似先进先出（FIFO）的方式从一个线程传递到另一个线程。使用消息队列函数，你可以控制、发送、接收或等待消息。要传递的数据可以是整数或指针类型：



函数 osMessageQueuePut、osMessageQueueGet、osMessageQueueGetCapacity、osMessageQueueGetMsgSize、osMessageQueueGetCount、osMessageQueueGetSpace 可以在中断服务例程中调用。

2.9.1 数据结构

- struct osMessageQueueAttr_t

数据字段

const char *	name	消息队列的名称 一个指向具有易读名称的常量字符串的指针（在调试期间显示）。 默认情况下，此指针为 NULL，表示未为消息队列对象分配名称。 .
uint32_t	attr_bits	属性位 保留供将来使用（为了将来兼容性必须设置为 '0'）。
void *	cb_mem	控制块内存指针 指向消息队列控制块对象的内存的指针。 默认值：NULL，表示使用自动动态分配的方式为消息队列控制块分配内存。
uint32_t	cb_size	提供的控制块内存大小，通过 `cb_mem` 传递的内存块的大小（以字节为单位）。 默认值：0，表示默认情况下没有为控制块（`cb_mem`）提供内存。
void *	mq_mem	数据存储内存，指向消息队列数据的内存的指针 默认值：NULL，表示使用自动动态分配的方式为内存池数据分配内存。
uint32_t	mq_size	提供的数据存储内存大小，通过 `mq_mem` 传递的内存块的大小（以字节为单位）。最小内存块大小为 osMessageQueueNew 函数的参数 `msg_count * msg_size`。`msg_size` 被四舍五

		入到一个双倍偶数以确保内存块的 32 位对齐。 默认值：0，表示默认情况下没有为数据存储（`mq_mem`）提供内存。

2.9.2 函数

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

  该函数创建并初始化一个消息队列对象，并返回消息队列对象标识符，或在出现错误时返回 NULL。

  函数可以在内核初始化后通过 osKernelInitialize 调用。在 RTOS 内核启动之前通过 osKernelStart 可以创建消息队列对象。

  消息队列数据所需的总内存至少为 `msg_count * msg_size`。`msg_size` 被四舍五入到一个双倍偶数以确保内存块的 32 位对齐。

  从消息队列分配的内存块具有用参数 `msg_size` 定义的固定大小。

参数		
msg_count	输入	队列中的最大消息数。
msg_size	输入	字节中的最大消息大小。
attr	输入	消息队列属性；如果为 NULL，则使用默认值。

返回值	
message queue ID	成功时。
NULL	出错时。

- const char *osMessageQueueGetName (osMessageQueueId_t mq_id)

该函数返回由参数 `mq_id` 标识的消息队列的名称字符串的指针，或在出现错误时返回 NULL。

参数		
mq_id	输入	通过 osMessageQueueNew 获取的消息队列 ID。

返回值	
name as null-terminated string	成功时。
NULL	出错时。

该 API 目前不支持。

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

  阻塞函数 osMessageQueuePut 将由 msg_ptr 指向的消息放入参数 mq_id 指定的消息队列中。参数 msg_prio 用于按照消息的优先级（数字越大表示优先级越高）进行排序。

  参数 timeout 指定系统等待将消息放入队列的时间。在系统等待期间，调用此函数的线程将进入阻塞状态。参数 timeout 可以有以下值：

  - 当 timeout 为 0 时，函数立即返回（即尝试行为）。
  - 当 timeout 设置为 osWaitForever 时，函数将无限期等待，直到消息被传递（即等待行为）。
  - 所有其他值指定了一个以内核滴答为单位的超时时间（即定时等待行为）。

  如果参数 timeout 设置为 0，则可以从中断服务程序中调用。

参数		
mq_id	输入	通过 osMessageQueueNew 获得的消息队列 ID。
msg_ptr	输入	指向要放入队列的消息的缓冲区的指针。

msg_prio	输入	消息优先级。
timeout	输入	超时值，或者在没有超时的情况下为 0。

返回值	
osOK	消息已放入队列。
osErrorTimeout	在给定时间内无法将消息放入队列（等待定时行为）。
osErrorResource	队列中没有足够的空间（尝试行为）。
osErrorParameter	参数 mq_id 为 NULL 或无效，中断服务程序中指定了非零超时时间。
osErrorNeedSched	当前线程将被抢占。

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

  函数 osMessageQueueGet 从参数 mq_id 指定的消息队列中检索消息，并将其保存到参数 msg_ptr 指向的缓冲区中。如果非 NULL，则将消息优先级存储到参数 msg_prio 中。

  参数 timeout 指定系统等待从队列中检索消息的时间。在系统等待期间，调用此函数的线程将进入阻塞状态。参数 timeout 可以有以下值：

  - 当 timeout 为 0 时，函数立即返回（即尝试行为）。
  - 当 timeout 设置为 osWaitForever 时，函数将无限期等待，直到消息被检索（即等待行为）。
  - 所有其他值指定了一个以内核滴答为单位的超时时间（即定时等待行为）。

  如果参数 timeout 设置为 0，则可以从中断服务程序中调用。

参数		
mq_id	输入	通过 osMessageQueueNew 获得的消息队列 ID。
msg_ptr	输出	用于从队列中获取消息的缓冲区的指针。
msg_prio	输出	用于消息优先级的缓冲区的指针，如果为 NULL 则不保存。

timeout	输入	超时值，或者在没有超时的情况下为 0。

返回值	
osOK	消息已从队列中检索。
osErrorTimeout	在给定时间内无法从队列中检索消息（定时等待行为）。
osErrorResource	队列中没有要获取的内容（尝试行为）。
osErrorParameter	参数 mq_id 为 NULL 或无效，中断服务程序中指定了非零超时时间。

- uint32_t osMessageQueueGetCapacity(osMessageQueueId_t mq_id)

  函数 osMessageQueueGetCapacity 返回参数 mq_id 指定的消息队列对象中的最大消息数，或者在出错时返回 0。

参数		
mq_id	输入	通过 osMessageQueueNew 获得的消息队列 ID。

返回值	
maximum number of messages	成功时。
0	出错时。

- uint32_t osMessageQueueGetMsgSize(osMessageQueueId_t mq_id)

  函数 osMessageQueueGetMsgSize 返回参数 mq_id 指定的消息队列对象的最大消息大小（以字节为单位），或者在出错时返回 0。

参数		
mq_id	输入	通过 osMessageQueueNew 获得的消息队列 ID。

返回值

maximum message size in bytes	成功时。
0	出错时。

- uint32_t osMessageQueueGetCount(osMessageQueueId_t mq_id)

  函数 osMessageQueueGetCount 返回参数 mq_id 指定的消息队列对象中排队的消息数，或者在出错时返回 0。

参数		
mq_id	输入	通过 osMessageQueueNew 获得的消息队列 ID。

返回值	
number of queued messages	成功时。
0	出错时。

- uint32_t osMessageQueueGetSpace(osMessageQueueId_t mq_id)

  函数 osMessageQueueGetSpace 返回参数 mq_id 指定的消息队列对象中可用于消息的槽位数，或者在出错时返回 0。

参数		
mq_id	输入	通过 osMessageQueueNew 获得的消息队列 ID。

返回值	
number of available slots for messages	成功时。
0	出错时。

- osStatus_t osMessageQueueReset(osMessageQueueId_t mq_id)

  函数 osMessageQueueReset 重置参数 mq_id 指定的消息队列。

参数		
mq_id	输入	通过 osMessageQueueNew 获得的消息队列 ID。

返回值	
osOK	消息队列已重置。
osErrorParameter	参数 mq_id 为 NULL 或无效。
osErrorResource	消息队列处于无效状态。
osErrorISR	不允许从中断服务程序中调用 osMessageQueueReset。

- osStatus_t osMessageQueueDelete(osMessageQueueId_t mq_id)

  函数 osMessageQueueDelete 删除参数 mq_id 指定的消息队列对象。它释放了为消息队列处理而获得的内部内存。调用此函数后，mq_id 不再有效，不能再使用。可以使用函数 osMessageQueueNew 再次创建消息队列。

参数		
mq_id	输入	通过 osMessageQueueNew 获得的消息队列 ID。

返回值	
osOK	消息队列对象已删除。
osErrorParameter	参数 mq_id 为 NULL 或无效。
osErrorISR	不允许从中断服务程序中调用 osMessageQueueDelete。

2.10 定义

以下常量和枚举用于许多 CMSIS-RTOS 函数调用。

2.10.1 宏定义

- #define osWaitForever 0xFFFFFFFFU

  一个特殊的超时值，通知 RTOS 无限等待直到资源可用。适用于以下函数：

  - osDelay：等待超时（时间延迟）。
  - osThreadFlagsWait：等待当前运行线程的一个或多个线程标志变为信号状态。
  - osEventFlagsWait：等待一个或多个事件标志变为信号状态。
  - osMutexAcquire：获取互斥锁或超时（如果锁已被锁定）。
  - osSemaphoreAcquire：获取信号量令牌或超时（如果没有可用的令牌）。

- osMessageQueuePut : 将消息放入队列或超时（如果队列已满）。
- osMessageQueueGet : 从队列获取消息或超时（如果队列为空）。

- #define osFlagsWaitAny 0x00000000U

  参考：

  - osEventFlagsWait
  - osThreadFlagsWait

- #define osFlagsWaitAll 0x00000001U

  参考：

  - osEventFlagsWait
  - osThreadFlagsWait

- #define osFlagsNoClear 0x00000002U

  参考：

  - osEventFlagsWait
  - osThreadFlagsWait

2.10.2 枚举类型

- enum osStatus_t

  osStatus_t 枚举定义了许多 CMSIS-RTOS 函数返回的事件状态和错误代码。

枚举值	
osOK	操作成功完成。
osError	未指定的 RTOS 错误：运行时错误但没有其他错误消息适用。

osErrorTimeout	操作未在超时期间完成。
osErrorResource	资源不可用。
osErrorParameter	参数错误。
osErrorNoMemory	系统内存不足：无法为操作分配或保留内存
osErrorISR	不允许在 ISR 上下文中：不能从中断服务例程中调用该函数。
osErrorNeedSched	需要重新调度，因为可以唤醒优先级更高的任务。
osStatusReserved	防止枚举向下大小编译器优化。