A76xx Series Open SDK_RTOS 开发_应用指导

LTE 模组

芯讯通无线科技(上海)有限公司
上海市长宁区临虹路289号3号楼芯讯通总部大楼
电话：86-21-31575100
技术支持邮箱：support@simcom.com
官网：www.simcom.com

名称：	A76xx Series Open SDK_RTOS开发_应用指导
版本：	V1.00
类别：	应用文档
状态：	已发布

版权声明

芯讯通无线科技(上海)有限公司
上海市长宁区临虹路289号3号楼芯讯通总部大楼
电话：86-21-31575100
邮箱：simcom@simcom.com
官网：www.simcom.com

了解更多资料，请点击以下链接：
http://cn.simcom.com/download/list-230-cn.html

技术支持，请点击以下链接：
http://cn.simcom.com/ask/index-cn.html 或发送邮件至 support@simcom.com

Version History

Version	Date	Owner	What is new
V1.00	2022-11-11		第一版

About this Document

本文档适用于 A1803S open 系列、A1603 open 系列、A1606 open 系列。

目录

缩略语

RTOS	Real-time Operating System

1RTOS 介绍

实时操作系统（RTOS）是指当外界事件或数据产生时，能够接受并以足够快的速度予以处理，其处理的结果又能在规定的时间之内来控制生产过程或对处理系统做出快速响应，调度一切可利用的资源完成实时任务，并控制所有实时任务协调一致运行的操作系统。提供及时响应和高可靠性是其主要特点。

实时操作系统中的调度程序旨在提供可预测（通常描述为确定性）执行模式。嵌入式系统对此特别感兴趣，因为嵌入式系统通常具有真实的时间要求。实时要求是指定嵌入式系统的要求必须在严格定义的时间（截止日期）内响应某个事件。只有在操作行为的情况下才能保证满足实时要求系统的调度程序是可以预测的（因此是确定性的）。

1.1 任务（task）

任务是一个逻辑概念，指由一个软件完成的活动，或者是为实现某个目的的一系列操作。通常一个任务是一个程序的一次运行，一个任务包含一个或多个完成独立功能的子任务，这个独立的子任务是进程或者是线程。

任务的状态转换图如下。

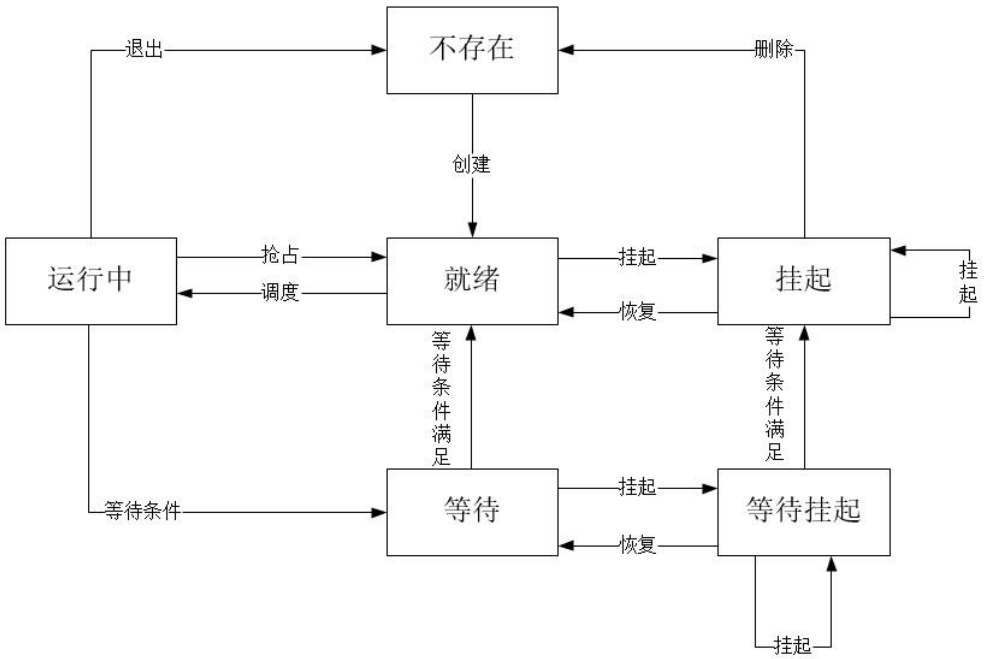

图 1 任务状态转换图

多个任务靠任务控制块组成了一个任务链表，每个任务都有独立的栈空间，而每个任务控制块使用双向链表链接在一起，而任务控制块的结构体某个成员变量指向任务堆栈，而任务堆栈包含了任务函数入口地址，如下图所示。

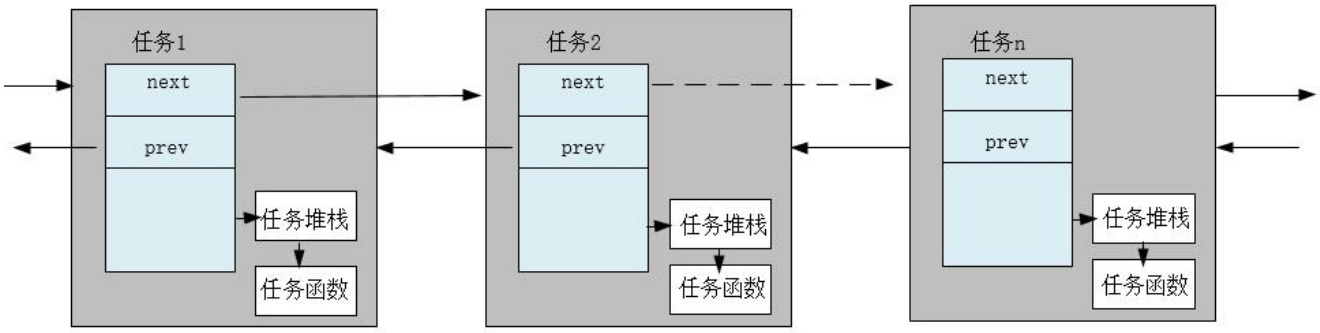

图 2 任务控制块链表

A76xx 平台提供的任务 API 接口如表 1 所示。

表 1 任务 API 接口

API 名称	说明
sAPI_TaskCreate	创建任务，此任务将自动启动
sAPI_TaskDelete	删除指定的任务
sAPI_TaskSuspend	暂停包括当前任务在内的特定任务
sAPI_TaskResume	恢复指定的任务
sAPI_TaskSleep	将当前正在执行的任务挂起指定时间
sAPI_TaskGetCurrentRef	获取当前正在执行的任务

1.2 消息队列（message queue）

消息队列可以在任务与任务间、中断和任务间传递消息，也可以进行数据传输。通过消息队列服务，任务或中断服务例程可以将一条或多条消息放入消息队列中。同样，一个或多个任务可以从消息队列中获得消息。

A76xx 平台提供的消息队列 API 接口如表 2 所示。

表 2 消息队列 API 接口

API 名称	说明
sAPI_MsgQCreate	创建消息队列
sAPI_MsgQDelete	请求删除指定的消息队列

sAPI_MsgQSend	请求将消息发送到指定的消息队列
sAPI_MsgQSendSuspend	在阻塞模式下将消息发送到指定的消息队列
sAPI_MsgQRecv	从指定的消息队列接收消息
sAPI_MsgQPoll	检查消息队列上的消息数

1.3 信号量（semaphore）

信号量也是为任务和任务、任务和中断之间通信做准备的，但是信号量一般用来进行资源管理和任务同步。因为信号量是一种共享资源，当它被创建之后，系统中所有任务和中断都能对信号量进行访问。同时也可以进行任务同步，即在一个任务(或中断)中告诉另一个任务它所等待的事件发生了，等到发生任务调度的时候，再切换到相应任务中，执行该事件发生的相关处理

A76xx 平台提供的信号量 API 接口如表 3 所示。

表 3 信号量 API 接口

API 名称	说明
sAPI_SemaphoreCreate	创建计数信号量
sAPI_SemaphoreDelete	删除计数信号量
sAPI_SemaphoreAcquire	递减指定的信号量
sAPI_SemaphoreRelease	将实例放入信号量，其计数增加一
sAPI_SemaphorePoll	获取当前信号量的计数

1.4 互斥锁（mutex）

互斥锁就像放在咖啡店篮子里的唯一一把钥匙。该钥匙可用于解锁共用的公共厕所。一个人在他们希望使用共享资源（洗手间）时拿走钥匙，并在完成后归还。当他们在洗手间时，其他人不得进入。其他人（类似于线程）必须等待密钥。互斥锁用来限制一次只能有一个线程访问某块代码。它会拦住访问代码或者资源的所有其他线程。这确保任何在临界区执行的代码是线程安全的，不会被其他线程破坏。

A76xx 平台提供的互斥锁 API 接口如表 4 所示。

表 4 互斥锁 API 接口

API 名称	说明
sAPI_MutexCreate	创建互斥锁
sAPI_MutexDelete	删除指定的互斥锁
sAPI_MutexLock	锁定指定的互斥锁
sAPI_MutexUnlock	释放之前获得的互斥锁

1.5 事件标志组（flag）

事件是一种实现任务间通信的机制，主要用于实现多任务间的同步，但事件通信只能是事件类型的通信，无数据传输。与信号量不同的是，它可以实现一对多，多对多的同步。即一个任务可以等待多个事件的发生：可以是任意一个事件发生时唤醒任务进行事件处理；也可以是几个事件都发生后才唤醒任务进行事件处理。同样，也可以是多个任务同步多个事件

A76xx 平台提供的事件标志组 API 接口如表 5 所示。

表 5 事件标志组 API 接口

API 名称	说明
sAPI_FlagCreate	创建标志组
sAPI_FlagDelete	删除标志组
sAPI_FlagSet	设置或清除事件组中的事件标志
sAPI_FlagWait	等待标志组上的指定操作完成

1.6 定时器（timer）

定时器是指从指定的时刻开始，经过一个指定时间，然后触发一个超时事件，用户可以自定义定时器的周期与频率。不妨想想我们生活当中的闹钟，我们可以设置闹钟每天什么时候响，还能设置响的次数，是响一次还是每天都响。

A76xx 平台提供的定时器 API 接口如表 6 所示。

表 6 定时器 API 接口

API 名称	说明

sAPI_TimerCreate	创建软件定时器
sAPI_TimerStart	启动具有指定到期功能的软件计时器，并定时
sAPI_TimerStop	停止软件定时器
sAPI_TimerDelete	删除软件定时器
sAPI_TimerStatus	此函数要求在 sTimerStatus 结构中返回指定计时器的状态

2RTOS API

RTOS 头文件
#include "simcom_os.h"

2.1 数据类型

2.1.1 sTaskRef

定义:	typedef void *sTaskRef;
NOTE:	

2.1.2 sSemaRef

定义:	typedef void * sSemaRef;
NOTE:	

2.1.3 sMutexRef

定义:	typedef void * sMutexRef;
NOTE:	

2.1.4 sMsgQRef

定义:	typedef void * sMsgQRef;
NOTE:	

2.1.5 sTimerRef

定义:	typedef void * sTimerRef;
NOTE:	

2.1.6 sFlagRef

定义:	typedef void * sFlagRef;
NOTE:	

2.2 函数

2.2.1 sAPI_TaskCreate 创建任务

创建任务，此任务将自动启动。

接口:	SC_STATUS **sAPI_TaskCreate**(sTaskRef *taskRef, void* stackPtr, UINT32 stackSize, UINT8 priority, char *taskName, void(*taskStart)(void*), void* argv);
参数:	[in]**stackPtr**：指向堆栈低位地址的指针。 [in]**stackSize**：堆栈的最大大小。 [in]**priority**：建议任务的优先级在 80 到 150 之间(越小优先级越高)。 [in]**taskName**：指向任务的 8 个字符名称的指针。名称不必以空结尾。 [in]**taskStart**：任务的入口函数。 [in]**argv**：要传递到任务入口函数的参数。 [out]**taskRef**：OS 分配给任务的引用。
返回值:	SC_SUCCESS：成功完成服务。 SC_INVALID_REF：任务引用为空。 SC_INVALID_PTR：任务的入口函数指针为空。 SC_INVALID_MEMORY：指向堆栈内存的指针为空。 SC_INVALID_SIZE：堆栈大小不足。 SC_INVALID_PRIORITY：优先级无效。 SC_NO_TASKS：没有可用的任务引用。 SC_FAIL：操作系统特定错误。
NOTE:	

2.2.2 sAPI_TaskDelete 删除任务

此函数要求删除指定的任务。

接口:	SC_STATUS **sAPI_TaskDelete**(sTaskRef taskRef);
参数:	[in]**taskRef**：对任务的引用。
返回值:	SC_SUCCESS：成功完成服务。 SC_INVALID_REF：任务引用无效。 SC_FAIL：无法删除任务，因为它未处于挂起状态或由于其他 OS 特定错误。
NOTE:	

2.2.3 sAPI_TaskSuspend 暂停任务

此函数要求暂停包括当前任务在内的特定任务。

接口:	SC_STATUS **sAPI_TaskSuspend**(sTaskRef taskRef);
参数:	[in]**taskRef**：对任务的引用。
返回值:	SC_SUCCESS：成功完成服务。 SC_INVALID_REF：任务引用为空。 SC_FAIL：操作系统特定错误。
NOTE:	

2.2.4 sAPI_TaskResume 恢复任务

此函数要求恢复指定的任务。

接口:	SC_STATUS **sAPI_TaskResume**(sTaskRef taskRef);
参数:	[in] **taskRef**：对任务的引用。
返回值:	SC_SUCCESS：成功完成服务。 SC_INVALID_REF：任务引用无效。 SC_FAIL：操作系统特定错误。
NOTE:	

2.2.5 sAPI_TaskSleep 挂起任务

将当前正在执行的任务挂起指定时间。在我们的操作系统中，1 ticks = 5ms （跟芯片相关，和操作系统没有强相关）。

接口:	void **sAPI_TaskSleep**(UINT32 ticks);
参数:	[in]**ticks**：要休眠的操作系统时钟周期数。
返回值:	无
NOTE:	

2.2.6 sAPI_TaskGetCurrentRef 获取正在执行的任务

获取当前正在执行的任务。

接口:	SC_STATUS **sAPI_TaskGetCurrentRef**(sTaskRef *taskRef);
参数:	[out] **taskRef**：OS 分配给任务的引用。
返回值:	SC_SUCCESS：成功完成服务。 SC_INVALID_REF：任务引用无效。 SC_FAIL：操作系统特定错误。
NOTE:	

2.2.7 sAPI_MsgQCreate 创建消息队列

此函数要求创建消息队列。用于在消息队列上存储消息的所有内存都由操作系统分配。

接口:	SC_STATUS **sAPI_MsgQCreate**(sMsgQRef *msgQRef, char *queueName, UINT32 maxSize, UINT32 maxNumber, UINT32 waitingMode);
参数:	[in]**queueName**：队列的 8 个字符名称。名称不必以空结尾。 [in]**maxSize**：队列上消息的最大大小。sAPI_MsgQSend()用于进行错误检查。 [in]**maxNumber**：队列上的最大消息数。 [in]**waitingMode**：定义等待事件的调度：SC_FIFO，或 SC_PRIORITY。 [out]**msgQRef**：指向保存由操作系统分配的消息队列引用的位置。
返回值:	SC_SUCCESS：成功完成服务。 SC_INVALID_REF：队列引用无效。 SC_INVALID_MODE：等待模式无效。 SC_INVALID_SIZE：队列大小无效。 SC_NO_QUEUES：系统中没有可用队列。 SC_FAIL：操作系统特定错误。
NOTE:	

2.2.8 sAPI_MsgQDelete 删除消息队列

此函数用于请求删除指定的消息队列。

接口:	SC_STATUS **sAPI_MsgQDelete**(sMsgQRef msgQRef);
参数:	[in]**msgQRef**：唯一标识消息队列的标识符。
返回值:	SC_SUCCESS：成功完成服务。 SC_INVALID_REF：任务引用为空。 SC_FAIL：操作系统特定错误。
NOTE:	

2.2.9 sAPI_MsgQSend 在非阻塞模式下发送消息

此函数要求在非阻塞模式下将消息发送到指定的消息队列。

接口:	SC_STATUS **sAPI_MsgQSend**(sMsgQRef msgQRef, SIM_MSG_T *msgPtr);
参数:	[in]**msgQRef**：唯一标识消息队列的标识符。 [in]**msgPtr**：数据的起始地址。
返回值:	SC_SUCCESS：成功完成服务。 SC_INVALID_REF：队列引用无效。 SC_INVALID_POINTER：消息指针为 NULL。 SC_QUEUE_FULL：表示消息队列已满。 SC_INVALID_SIZE：表示消息大小与队列支持的消息大小不兼容。可以发送到队列的消息的最大大小在 sAPI_MsgQCreate()中指定； SC_FAIL：操作系统特定错误。
NOTE:	

2.2.10 sAPI_MsgQSendSuspend 在阻塞模式下发送消息

此函数要求在阻塞模式下将消息发送到指定的消息队列。

接口:	SC_STATUS **sAPI_MsgQSendSuspend**(sMsgQRef msgQRef, SIM_MSG_T *msgPtr, UINT32 timeout);
参数:	[in]**msgQRef**：唯一标识消息队列的标识符。 [in]**msgPtr**：数据的起始地址。 [in]**timeout**：指定的 ticks。1 ticks = 5ms。如果 timeout = SC_SUSPEND，

	函数将阻塞，直到队列上有可用空间。
返回值:	SC_SUCCESS：成功完成服务。 SC_INVALID_REF：队列引用无效。 SC_INVALID_POINTER：消息指针为 NULL。 SC_QUEUE_FULL：表示消息队列已满。 SC_INVALID_SIZE：表示消息大小与队列支持的消息大小不兼容。可以发送到队列的消息的最大大小在 sAPI_MsgQCreate()中指定。 SC_FAIL：操作系统特定错误。
NOTE:	

2.2.11 sAPI_MsgQRecv 接收消息

此函数用于请求从指定的消息队列接收消息。如果队列为空，则调用的阻塞行为由“timeout”参数的值决定。

接口:	SC_STATUS **sAPI_MsgQRecv**(sMsgQRef msgQRef, SIM_MSG_T *recvMsg, UINT32 timeout);
参数:	[in]**msgQRef**：唯一标识消息队列的标识符。 [in]**timeout**：如果 timeout 设置为 SC_NO_SUSPEND，则此调用不会阻塞。如果超时设置为 SC_SUSPEND，则此调用将阻塞，直到队列上有消息可用。如果指定了介于 1 和 4、294、967、293 之间的超时值，则调用将被阻塞，直到消息可用或超时时间（以 OS 时钟 ticks 数计）过去。 [out]**recvMsg**：指向应用程序提供的缓冲区的指针，接收到的消息应复制到该缓冲区。
返回值:	SC_SUCCESS：成功完成服务。它表示消息已复制到接收任务的“recvMsg”缓冲区。 SC_INVALID_REF：队列引用无效。 SC_INVALID_POINTER：“recvMsg”指针为空。 SC_QUEUE_EMPTY：表示消息队列为空。 SC_TIMEOUT：暂停等待空队列上的消息时发生超时。 SC_INVALID_SIZE：表示实际消息大小大于“size”参数所指示的应用程序接收缓冲区的大小。 SC_FAIL：操作系统特定错误。
NOTE:	

2.2.12 sAPI_MsgQPoll 检查消息队列上的消息数

此函数用于检查消息队列上的消息数。

接口:	SC_STATUS **sAPI_MsgQPoll**(sMsgQRef msgQRef, UINT32* msgCount);

参数:	[in]**msgQRef**：唯一标识消息队列的标识符。 [out]**msgCount**：从该函数返回时，msgCount 包含队列上的消息。
返回值:	SC_SUCCESS：成功完成服务。 SC_INVALID_POINTER：msgCount 指针为 NULL。 SC_INVALID_REF：队列引用无效。
NOTE:	

2.2.13 sAPI_SemaphoreCreate 创建计数信号量

此函数要求创建计数信号量。

接口:	SC_STATUS **sAPI_SemaphoreCreate**(sSemaRef *semaRef, UINT32 initialCount, UINT32 waitingMode);
参数:	[in]**initialCount**：信号量的初始计数。 [in]**waitingMode**：SC_FIFO 或 SC_PRIORITY。“waitingMode”指定任务如何被添加到信号量的等待队列中。它们可以按先进先出的顺序添加（SC_FIFO）；或按优先级顺序（SC_priority）；最高优先级的等待任务位于队列的前面。 [out]**semaRef**：OS 分配给信号量的引用。
返回值:	SC_SUCCESS：成功完成服务。 SC_INVALID_REF：信号量引用无效。 SC_INVALID_MODE：模式无效。 SC_NO_SEMAPHORES：没有可用的信号量。
NOTE:	

2.2.14 sAPI_SemaphoreDelete 删除计数信号量

此函数要求删除计数信号量。

接口:	SC_STATUS **sAPI_ SemaphoreDelete**(sSemaRef semaRef);
参数:	[in]**semaRef**：OS 分配给信号量的引用。
返回值:	SC_SUCCESS：成功完成服务。 SC_INVALID_REF：信号量引用无效。 SC_FAIL：操作系统特定错误。
NOTE:	

2.2.15 sAPI_SemaphoreAcquire 递减指定的信号量

此函数要求递减指定的信号量。如果此调用之前的信号量计数为零，则无法立即满足服务。在这种情况下，阻塞行为由“timeout”参数指定。

接口:	SC_STATUS **sAPI_SemaphoreAcquire**(sSemaRef semaRef, UINT32 timeout);
参数:	[in]**semaRef**：OS 分配给信号量的引用。 [in]**timeout**：如果 timeout 设置为 SC_NO_SUSPEND，则此调用不会阻塞。如果超时设置为 SC_SUSPEND，则此调用将阻塞，直到队列上有消息可用。如果指定了介于 1 和 4、294、967、293 之间的超时值，则调用将被阻塞，直到消息可用或超时时间（以 OS 时钟 ticks 数计）过去。
返回值:	SC_SUCCESS：信号量已递减。 SC_INVALID_REF：信号量引用无效。 SC_UNAVAILABLE：信号量不可用。 SC_TIMEOUT：等待信号量计数大于零时发生超时。 SC_FAIL：操作系统特定错误。
NOTE:	

2.2.16 sAPI_SemaphoreRelease 释放计数信号量

如果有任何任务在等待信号量，那么第一个等待的任务就可以运行了。如果没有任务等待，则信号量的值将增加 1。

接口:	SC_STATUS **sAPI_SemaphoreRelease**(sSemaRef semaRef);
参数:	[in]**semaRef**：OS 分配给信号量的引用。
返回值:	SC_SUCCESS：成功完成服务。 SC_INVALID_REF：信号量引用无效。
NOTE:	

2.2.17 sAPI_SemaphorePoll 获取当前信号量的计数

获取当前信号量的计数。

接口:	SC_STATUS **sAPI_SemaphorePoll**(sSemaRef semaRef, UINT32 *count);
参数:	[in]**semaRef**：OS 分配给信号量的引用。 [out]**count**：信号量的当前值。
返回值:	SC_SUCCESS：成功完成服务。

	SC_INVALID_REF：信号量引用无效。
NOTE:	

2.2.18 sAPI_MutexCreate 创建互斥锁

此函数要求创建互斥锁。互斥锁使用优先级继承协议来限制优先级反转所花费的时间。

接口:	SC_STATUS **sAPI_MutexCreate**(sMutexRef *mutexRef, UINT32 waitingMode);
参数:	[in]**waitingMode**：SC_FIFO 或 SC_PRIORITY。 [out]**mutexRef**：OS 分配给互斥锁的引用。
返回值:	SC_SUCCESS：成功完成服务。 SC_INVALID_REF：无效的互斥锁引用。 SC_INVALID_MODE：模式无效。 SC_NO_MUTEXES：系统中没有可用的互斥锁。 SC_FAIL：互斥锁已被任务锁定。
NOTE:	

2.2.19 sAPI_MutexDelete 删除互斥锁

此函数要求删除指定的互斥锁。

接口:	SC_STATUS **sAPI_MutexDelete**(sMutexRef mutexRef);
参数:	[in]**mutexRef**：OS 分配给互斥锁的引用。
返回值:	SC_SUCCESS：成功完成服务。 SC_INVALID_REF：无效的互斥锁引用。 SC_FAIL：操作系统特定错误。
NOTE:	

2.2.20 sAPI_MutexLock 锁定互斥锁

此函数要求锁定指定的互斥锁。如果互斥锁在调用之前被另一个任务锁定，则服务无法立即得到满足。在这种情况下，阻塞行为由“timeout”参数指定。

接口:	SC_STATUS **sAPI_MutexLock**(sMutexRef mutexRef, UINT32 timeout);
参数:	[in]**semaRef**：OS 分配给信号量的引用。

	[in]**timeout**：如果 timeout 设置为 SC_NO_SUSPEND，则此调用不会阻塞。 如果超时设置为 SC_SUSPEND，则此调用将阻塞，直到队列上有消息可用。 如果指定了介于 1 和 4、294、967、293 之间的超时值，则调用将被阻塞，直到消息可用或超时时间（以 OS 时钟 ticks 数计）过去。
返回值:	SC_SUCCESS：已获取互斥锁。 SC_INVALID_REF：无效的互斥锁引用。 SC_UNAVAIABLE：互斥锁不可用。它被另一个任务锁定。 SC_TIMEOUT：等待时超时用于互斥锁。 SC_FAIL：调用任务已锁定互斥锁。
NOTE:	

2.2.21 sAPI_MutexUnlock 解锁互斥锁

如果有任何任务在等待互斥锁，那么等待队列前面的任务就可以运行了。只有互斥锁所有者可以解锁互斥锁。

接口:	SC_STATUS **sAPI_MutexUnlock**(sMutexRef mutexRef);
参数:	[in]**semaRef**：OS 分配给信号量的引用。
返回值:	SC_SUCCESS：成功完成服务。 SC_INVALID_REF：无效的互斥锁引用。 SC_FAIL：调用任务不是互斥锁所有者。
NOTE:	

2.2.22 sAPI_FlagCreate 创建标志组

此函数要求创建标志组。

接口:	SC_STATUS **sAPI_FlagCreate**(sFlagRef *flagRef);
参数:	[out]**flagRef**：OS 分配给标志的引用。
返回值:	SC_SUCCESS：成功完成服务。 SC_INVALID_REF：指向“flagRef”的指针为 NULL。 SC_NO_FLAGS：系统中没有可用的标志。
NOTE:	

2.2.23 sAPI_FlagDlete 删除标志组

此函数请求删除标志组。

接口:	SC_STATUS **sAPI_FlagDlete**(sFlagRef flagRef);
参数:	[in]**flagRef**：OS 分配给标志的引用。
返回值:	SC_SUCCESS：成功完成服务。 SC_INVALID_REF：标志引用无效。 SC_FAIL：操作系统特定错误。
NOTE:	

2.2.24 sAPI_FlagSet 设置标志组

此函数对带有输入掩码的标志组执行逻辑 OR 或 AND。

接口:	SC_STATUS **sAPI_FlagSet**(sFlagRef flagRef, UINT32 mask, UINT32 operation);
参数:	[in]**flagRef**：OS 分配给标志的引用。 [in]**mask**：指定需要设置的位的掩码。某位位置的 1 将在标志中设置相同的位位置。 [in]**operation**：要对标志组执行的逻辑操作。SC_FLAG_AND 执行逻辑 AND，SC_FLAG_OR 执行逻辑 OR。
返回值:	SC_SUCCESS：成功完成服务。 SC_INVALID_REF：标志引用为 NULL。 SC_INVALID_MODE：操作无效。
NOTE:	

2.2.25 sAPI_FlagWait 等待标志组上的操作完成

此函数用于等待标志组上的指定操作完成。操作由“operation”输入参数定义。如果超时输入参数为 SC_NO_SUSPEND，则操作将立即完成，标志的当前值将在输出参数“flags”中返回。通过指定 SC_NO_SUSPEND，应用程序可以在不阻塞的情况下读取标志的当前值。

接口:	SC_STATUS **sAPI_FlagWait**(sFlagRef flagRef, UINT32 mask, UINT32 operation, UINT32* flags, UINT32 timeout);
参数:	[in]**flagRef**：OS 分配给标志的引用。 [in]**mask**：要等待的标志的掩码。 [in]**operation**：可以是以下操作之一： **SC_FLAG_AND**：等待输入掩码中的所有位 **SC_FLAG_AND_CLEAR**：等待设置输入掩码中的所有位，成功时清除所有事件标志 **SC_FLAG_OR**：等待输入掩码中的任何位被设置，不清除事件标志

	SC_FLAG_OR_CLEAR：等待输入掩码中的任何位完成设置，不清除事件标志 [in]**timeout**：如果 timeout 设置为 SC_NO_SUSPEND，操作将立即完成，标志的当前值将在输出参数“flags”中返回。如果超时设置为 SC_SUSPEND，此调用将被阻塞，直到满足“mask”和“operation”输入指定的条件。如果指定了介于 1 和 4、294、967、293 之间的超时值，则调用将被阻塞，直到满足等待条件，或者直到超时时间（以 OS 时钟 ticks 数计）过去。 [out]**flags**：所有标志的当前值。
返回值:	SC_SUCCESS：成功完成服务。 SC_INVALID_REF：标志引用为 NULL。 SC_INVALID_MODE：无效操作。 SC_INVALID_POINTER：指向“flags”的指针为 NULL。 SC_TIMEOUT：暂停等待 Wait 操作完成时发生超时。 SC_FLAG_NOT_PRESENT：使用 SC_NO_SUSPEND 时不符合标志组合。
NOTE:	

2.2.26 sAPI_Malloc 分配内存

从默认内存池中分配一个大小为字节的内存块，并返回一个指向块开头的指针。

接口:	void ***sAPI_Malloc**(UINT32 Size);
参数:	[in]**size**：要分配的字节数。
返回值:	指向分配的内存空间的第一个地址。如果返回为 NULL，说明分配内存失败。
NOTE:	

2.2.27 sAPI_Free 释放内存

将内存释放到默认内存池。

接口:	void **sAPI_Free**(void *p);
参数:	[in]**p**：需要释放的内存的第一个地址。
返回值:	无
NOTE:	

2.2.28 sAPI_TimerCreate 创建计时器

此函数用于分配计时器。分配的计时器的状态为非激活状态。sAPI_TimerStart();用于启动计时器。

接口:	SC_STATUS **sAPI_TimerCreate**(sTimerRef *timerRef);
参数:	[in]**timerRef**：存储由操作系统分配的计时器的引用的地址。
返回值:	SC_SUCCESS：成功完成服务。 SC_NO_TIMERS：系统中没有可用的计时器。 SC_INVALID_REF：输入参数“timerRef”是空指针。
NOTE:	

2.2.29 sAPI_TimerStart 启动计时器

此函数要求启动非激活计时器，并在计时器到期时执行回调函数。

接口:	SC_STATUS **sAPI_TimerStart**(sTimerRef timerRef, UINT32 initialTime, UINT32 rescheduleTime, void(*callBackRoutine)(UINT32), UINT32 timerArgc);
参数:	[in]**initialTime**：以 OS 时钟 ticks 为单位的初始到期时间。 [in]**rescheduleTime**：如果为 0，则禁用循环计时，计时器仅过期一次。如果不为 0，它表示循环定时器的周期，以 OS 时钟 ticks 为单位。1 tick = 5ms。 [in]**callBackRoutine**：指定每次计时器到期时要执行的应用程序例程。回调函数不得调用任何“blocking”操作系统调用。 [in]**timerArgc**：到期时传递给回调例程的参数。 [out]**timerRef**：OS 提供的计时器引用。
返回值:	SC_SUCCESS：成功完成服务。 SC_INVALID_REF：输入参数“timerRef”不是有效的计时器引用。 SC_INVALID_PTR：输入参数“callBackRoutine”是空指针 SC_FAIL：计时器仍处于激活状态。
NOTE:	请不要在计时器回调函数中执行阻塞操作。

2.2.30 sAPI_TimerStop 关闭计时器

此函数要求将活动计时器的状态更改为非激活。当计时器处于非激活状态时调用此函数无效。

接口:	SC_STATUS **sAPI_TimerStop**(sTimerRef timerRef);
参数:	[in]**timerRef**：OS 分配给计时器的引用。
返回值:	SC_SUCCESS：成功完成服务。

	SC_INVALID_REF：计时器引用无效。
NOTE:	

2.2.31 sAPI_TimerDelete 删除计时器

此函数要求删除指定的计时器。调用此函数时，计时器必须处于非激活状态。

接口:	SC_STATUS **sAPI_TimerDelete**(sTimerRef timerRef);
参数:	[in]**timerRef**：创建计时器时分配的计时器引用。
返回值:	SC_SUCCESS：成功完成服务。 SC_INVALID_REF：计时器引用无效。 SC_FAIL：计时器仍处于激活状态。
NOTE:	

2.2.32 sAPI_TimerGetStatus 获取计时器状态

此函数要求在 sTimerStatus 结构中返回指定计时器的状态。

接口:	SC_STATUS **sAPI_TimerGetStatus**(sTimerRef timerRef, sTimerStatus* timerStatus);
参数:	[in]**timerRef**：创建计时器时分配的计时器引用。 [out]**timerStatus**：指向要填充的结构的指针。
返回值:	SC_SUCCESS：成功完成服务。 SC_INVALID_REF：计时器引用无效。
NOTE:	

2.2.33 sAPI_GetTicks 获取上次系统启动以来的运行时间

此函数请求自上次系统启动以来的运行时间（以 OS 时钟周期为单位）。

接口:	UINT32 sAPI_GetTicks(void);
参数:	无。
返回值:	以 OS 时钟周期为单位的时间。
NOTE:	

2.2.34 sAPI_Gettimeofday 获取 Epoch 以来的秒数和微秒数

此函数可以获取时间，并给出自 Epoch（1970 年 1 月 1 日）以来的秒数和微秒数。

接口:	INT32 **sAPI_Gettimeofday**(sTimeval *tv,void* dummy);
参数:	[out]**tv**：给出自 Epoch 以来的秒数和微秒数。 [out]**dummy**：保留参数。
返回值:	成功返回 0，失败返回-1。
NOTE:	

2.2.35 sAPI_Time 获取 Epoch 以来的秒数

此函数返回自 Epoch 以来的秒数，1970-01-01 00:00:00 +0000 (UTC)。

接口:	unsigned long **sAPI_Time**(unsigned long *t);
参数:	[out]**t**：如果 t 不是 NULL，那么返回值也存储在 t 指向的内存中。
返回值:	返回自 Epoch 以来的时间值（以秒为单位）。
NOTE:	

2.2.36 sAPI_GetSystemInfo 获取 cpu 使用率和剩余堆空间大小

此函数输出 cpu 使用率和剩余堆空间大小。

接口:	Void **sAPI_GetSystemInfo**(unsignedint* cpuUsedRate, unsigned int *heapFreeSize)
参数:	[in]**cpuUsedRate**：用于保存 cpu 使用率。 [in]**heapFreeSize**：用于保存剩余堆大小。
返回值:	无。
NOTE:	

2.2.37 sAPI_ContextLock 锁定上下文

此函数用于锁定上下文（不能中断和抢占）。

接口:	SC_STATUS **sAPI_ContextLock**(void);
参数:	无。
返回值:	SC_SUCCESS：成功完成服务。 other：操作系统特定错误。
NOTE:	

2.2.38 sAPI_ContextUnlock 还原上下文

此函数用于还原上下文。

接口:	SC_STATUS **sAPI_ContextUnlock**(void);
参数:	无。
返回值:	SC_SUCCESS：成功完成服务。 other：操作系统特定错误。
NOTE:	

2.2.39 sAPI_GettimeofdaySyncRtc 获取时间

此函数可以获取时间，并给出自 Epoch 以来的秒数和微秒数。RTC 时间改变后，获得的时间将同步改变。

接口:	INT32 **sAPI_GettimeofdaySyncRtc**(sTimeval *tv,void* dummy);
参数:	[out]**tv**：给出 Epoch 以来的秒数和微秒数。 [out]**dummy**：保留参数。
返回值:	成功返回 0，失败返回-1。
NOTE:	

2.2.40 sAPI_DelayUs 任务挂起

将当前正在执行的任务挂起指定时间（执行此函数有概率导致系统死机，请谨慎使用）。

接口:	void **sAPI_DelayUs**(unsigned int us);
参数:	[in]**us**：延迟时间。
返回值:	无。

NOTE:

3 变量定义

3.1 SC_STATUS

```
typedef enum
{
    SC_SUCCESS = 0,              /* 0x0 –没有错误                          */
    SC_FAIL,                     /* 0x1 -操作失败代码                      */
    SC_TIMEOUT,                  /* 0x2 -等待资源超时                      */
    SC_NO_RESOURCES,             /* 0x3 -内部操作系统资源已过期            */
    SC_INVALID_POINTER,          /* 0x4 -0 或指针值超出范围                */
    SC_INVALID_REF,              /* 0x5 -无效的引用                       */
    SC_INVALID_DELETE,           /* 0x6 -删除未终止的任务                  */
    SC_INVALID_PTR,              /* 0x7 -无效的内存指针                    */
    SC_INVALID_MEMORY,           /* 0x8 -无效的内存指针                    */
    SC_INVALID_SIZE,             /* 0x9 -大小参数超出范围                 */
    SC_INVALID_MODE,             /* 0xA, 10 -无效模式                      */
    SC_INVALID_PRIORITY,         /* 0xB, 11 -超出范围的任务优先级          */
    SC_UNAVAILABLE,              /* 0xC, 12 -请求的服务不可用或正在使用    */
    SC_POOL_EMPTY,               /* 0xD, 13 -资源池中没有资源              */
    SC_QUEUE_FULL,               /* 0xE, 14 -尝试发送到完整消息队列        */
    SC_QUEUE_EMPTY,              /* 0xF, 15 -队列上没有消息                */
    SC_NO_MEMORY,                /* 0x10, 16 -没有剩余内存                 */
    SC_DELETED,                  /* 0x11, 17 -服务已删除                   */
    SC_SEM_DELETED,              /* 0x12, 18 -信号量已删除                 */
    SC_MUTEX_DELETED,            /* 0x13, 19 –互斥锁已删除                 */
    SC_MSGQ_DELETED,             /* 0x14, 20 –消息队列已删除               */
    SC_MBOX_DELETED,             /* 0x15, 21 -邮箱队列                     */
    SC_FLAG_DELETED,             /* 0x16, 22 -标志已删除                   */
    SC_INVALID_VECTOR,           /* 0x17, 23 -中断向量无效                 */
    SC_NO_TASKS,                 /* 0x18, 24 -超过了系统中的最大任务数     */
    SC_NO_FLAGS,                 /* 0x19, 25 -超过系统中标志的最大数量     */
    SC_NO_SEMAPHORES,            /* 0x1A, 26 -超过系统中信号量的最大数量   */
    SC_NO_MUTEXES,               /* 0x1B, 27 -超过了系统中互斥锁的最大数量 */
    SC_NO_QUEUES,                /* 0x1C, 28 -超过系统中消息队列的最大数量 */
    SC_NO_MBOXES,                /* 0x1D, 29 -超过了系统中邮箱队列的最大数量 */
    SC_NO_TIMERS,                /* 0x1E, 30 -超过系统中计时器的最大数量   */
```

```
    SC_NO_MEM_POOLS,        /* 0x1F, 31 -超过系统中 mem 池的最大数量      */
    SC_NO_INTERRUPTS,       /* 0x20, 32 -超过了系统中 isr 的最大数量      */
    SC_FLAG_NOT_PRESENT,    /* 0x21, 33 -请求的标志组合不存在             */
    SC_UNSUPPORTED,         /* 0x22, 34 -操作系统不支持服务               */
    SC_NO_MEM_CELLS,        /* 0x23, 35 -没有全局存储单元                 */
    SC_DUPLICATE_NAME,      /* 0x24, 36 -重复的全局存储单元名称           */
    SC_INVALID_PARM         /* 0x25, 37 -无效参数                         */
} SC_STATUS;
```

4 注意事项

4.1 任务优先级

数值越小任务的优先级越高，建议任务的优先级在 80 到 150 之间。

4.2 系统时间精度

在我们的操作系统中，1 ticks = 5ms （跟芯片相关，和操作系统没有强相关）。

4.3 定时器回调函数

请不要在计时器回调函数中执行阻塞操作，比如 sAPI_DelayUs，执行此操作有概率导致系统死机。