第3章 开源鸿蒙LiteOS基础

3.1 基础

3.1.1 嵌入式操作系统介绍

嵌入式操作系统是一种专门为嵌入式系统设计的操作系统，负责管理系统的软硬件资源任务调度和并行活动的控制。由于嵌入式设备中的资源受限，因此它们通常被设计得紧凑有效，具有可裁剪、可移植性。同时，嵌入式操作系统的个性化很强，其中的软件与硬件结合得非常紧密，一款嵌入式操作系统往往需要对不同的芯片架构（如 ARM Cortex-M、RISC-V、MIPS 等）进行适配。

嵌入式操作系统通常为实时操作系统（RTOS），主流的嵌入式操作系统有：μClinux、μC/OS-II、eCos、FreeRTOS、mbed OS、RTX、Vxworks、QNX、NuttX、zephyr、Alios Things、LiteOS、RT-Thread、SylixOS。

其中 FreeRTOS 与 μC/OS-II 因其成熟稳定、文档丰富、社区活跃而尤为知名，长期占据嵌入式开发领域的主导地位。然而，随着物联网技术的迅猛发展，各大厂家开始研发自己的嵌入式操作系统，以更好地满足特定应用场景的需求，提升产品竞争力。

于是，一批兼具技术创新与生态整合能力的操作系统应运而生，如由睿赛德科技负责的 RT-Thread、阿里的 AliOS Things、Linux 基金会支持的 Zephyr，以及华为推出的 LiteOS。

3.1.2 LiteOS 的起源

LiteOS 是华为面向物联网（IoT）领域打造的轻量级操作系统，于 2015 年 5 月在华为网络大会上发布。

LiteOS 从设计之初即以“轻”为核心：内核体积可低于 10KB，支持毫秒级快速启动，适用于资源极其受限的微控制器（MCU）设备，如传感器节点、智能穿戴设备和低功耗模组等。

此外，LiteOS 也有很好的中间件与组件的支持（如传感器框架和低功耗框架），内置了如 CoAP、LwM2M、MQTT 等轻量级物联网协议栈，并支持 NB-IoT、LoRa、BLE、Wi-Fi 等多种无线通信技术，极大地简化了设备的联网过程。

3.1.3 LiteOS 的发展

自发布以来，LiteOS 不断演进，逐步从一个企业级操作系统发展为开放、协同的物联网生态平台。

2016 年，华为宣布将 LiteOS 开源，并建立 LiteOS 开源社区，这一时期主要关注基础的内核功能和驱动程序，主要贡献集中在修复 bug、优化性能和增加新功能。

2017 年，华为提出“云-管-端”战略，进一步推动 LiteOS 在物联网生态中的布局，并将其应用于自有产品如 Watch 3 Pro 等可穿戴设备中。

2018 年，LiteOS 开始向行业级应用拓展，与多家芯片厂商（如兆易创新、乐鑫）及家

电企业(如美的、海尔)合作,推出定制化解决方案。同时,LiteOS 在智能家居、车联网等领域落地,支持设备间的互连互通,为日后成为 OpenHarmony 生态的重要组成部分奠定了基础。

2020 年 12 月,LiteOS 发布 V5.0 版本,带来架构升级与功能革新:引入模块化设计,支持灵活裁剪;推出轻量级 AI 框架,增强终端智能化能力;发布开发工具 LiteOS Studio,显著提升开发效率。这一阶段,LiteOS 已支持超过 30 种主流 MCU 平台,覆盖 NB-IoT、Wi-Fi、蓝牙等多种通信协议。

2021 年,LiteOS 进一步融合云计算与边缘计算能力,实现"端云协同"。例如,通过与华为云 IoT 平台深度整合,LiteOS 设备可直接接入云端进行数据分析与远程管理,广泛应用于智能抄表、智慧物流、工业监测等场景。

2022 年,LiteOS 原有代码仓开始逐步停止维护,逐步发展为 LiteOS-A 和 LiteOS-M 两种内核,并转移进 OpenHarmony 代码仓。其中 LiteOS-M 适用于 MCU 等各种资源极小的设备,LiteOS-A 适用于资源较丰富的嵌入式设备。

3.1.4 LiteOS 的现状

截至 2025 年,LiteOS 已全面融入 OpenHarmony 生态体系,成为其核心内核组件之一。目前,LiteOS 主要以 LiteOS-M 和 LiteOS-A 两种形态存在,覆盖从传感器到智能终端的全场景需求。

3.1.5 LiteOS 与 OpenHarmony 的关系

OpenHarmony 是华为捐献给开放原子开源基金会(OpenAtom Foundation)孵化及运营的开源项目。其本质是 HarmonyOS 的基础部分。OpenHarmony 将一个操作系统的基础进行拆分,然后让各个领域的开发者去完善、优化擅长的部分或者使用系统的各个部分。

OpenHarmony 的目标是打破传统嵌入式设备、手机、平板甚至计算机之间的壁垒,实现多端协同操作,按设备类型选择不同的内核来实现应用并可以将多个设备作为一个整体来使用。

LiteOS 最初是为物联网设备设计的操作系统,由内核及其上层框架构成,并提供了相应的 API 接口给应用程序开发者使用。为了实现多端协作操作的目标,OpenHarmony 包含了对传统嵌入式设备的支持,其中就包括 LiteOS 内核。

所以,我们可以理解为 LiteOS 为 OpenHarmony 轻量级内核的一部分,虽然 OpenHarmony 使用了 LiteOS 的核心部分,但不意味着它们之间一定存在包含关系。

3.1.6 星闪系列芯片与 LiteOS 的集成架构

在上海海思星闪系列芯片中,操作系统并非直接调用 LiteOS 接口,而是通过标准化的操作系统抽象层(OSAL),实现与 LiteOS 的高效集成。

OSAL 的设计核心在于解耦硬件平台与操作系统,为上层协议栈和应用提供统一、

可移植的系统服务接口。这种架构设计优势在于，可以实现高可移植性，通过 OSAL，星闪协议栈未来可以轻松适配不同的 RTOS 平台（如 FreeRTOS、RT-Thread 或未来其他系统），而无须修改核心通信逻辑，极大地提升了软件复用性。

3.2 任务

3.2.1 概述

在嵌入式实时操作系统中，任务（task）是程序执行的基本单元，也是系统进行调度和资源分配的核心对象。LiteOS 作为一款轻量级、高实时性的物联网操作系统，其任务管理机制的设计简洁高效，支持抢占式调度、优先级管理、任务间通信与同步等功能。

3.2.2 任务基本概念

在 LiteOS 中，任务是指一个独立运行的程序实体，拥有自己的运行上下文、栈空间和优先级。

从编程的角度来看，每个任务都以函数的形式定义。

当任务被创建后，系统为其分配独立的栈空间以保存局部变量、函数调用信息和寄存器状态，从而实现多任务并发执行。这种机制确保了任务在被中断或切换时能够完整保存执行状态，并在恢复运行时准确还原上下文，从而实现多个任务“并发”执行的效果。但实际上，单核处理器是通过快速切换来实现时间上的并行。

所有任务均由 LiteOS 内核统一管理，由任务调度器根据调度策略决定在任意时刻哪一个任务获得 CPU 资源并进入运行状态。LiteOS 采用抢占式调度机制，即当一个更高优先级的任务变为就绪状态时（如从阻塞中唤醒），任务调度器会立即中断当前正在运行的低优先级任务，将 CPU 控制权交给高优先级任务，从而保证关键任务能够及时响应，满足实时性要求。

对于优先级相同的任务，LiteOS 支持时间片轮转调度方式。每个任务被分配一个固定的时间片，当时间片耗尽后，系统会自动触发任务切换，调度同优先级队列中的下一个就绪任务运行。

LiteOS 支持 32 个任务优先级，编号范围为 0～31，其中 0 为最高优先级，31 为最低优先级。在设计应用时，应根据任务的实时性需求合理分配优先级。例如，对响应速度要求高的通信任务可设置较高优先级，而数据采集或日志记录类任务可设置较低优先级。

3.2.3 任务相关概念

为了深入理解 LiteOS 的任务机制，还需掌握以下几个关键概念。

1. 任务控制块

任务控制块（Task Control Block，TCB）是 LiteOS 内核用于管理任务的数据结构，每个任务对应一个唯一的 TCB。TCB 包含了任务上下文栈指针（stack pointer）、任务状

态、任务优先级、任务 ID、任务名、任务栈大小等信息。

2. 任务栈

每个任务都拥有独立的栈空间，用于存储函数调用过程中的局部变量、返回地址和寄存器压栈数据。栈的大小在创建任务时指定，需根据任务复杂度合理设置，过小可能导致栈溢出，过大则浪费内存资源。

3. 任务状态

如图 3.1 所示，在 LiteOS 中，每个任务在其生命周期内会经历不同的运行状态。任务状态的转换由内核调度器统一管理，反映了任务当前的执行情况和资源占用状态。LiteOS 将任务状态划分为以下四种基本类型。

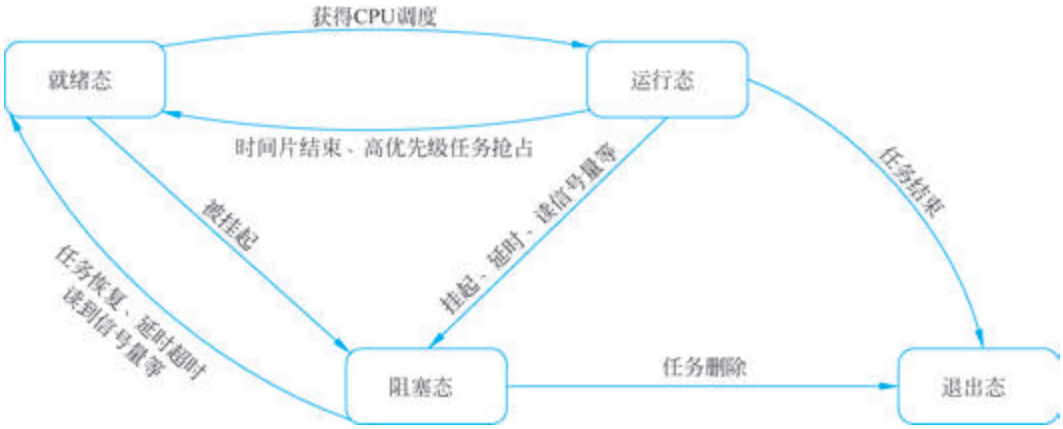

图 3.1 任务状态的转换示意图

(1) 就绪态(ready)：任务已经具备运行条件，所需的系统资源(如内存、外设)均已准备就绪，仅等待 CPU 调度。

此时任务被挂入就绪队列中，一旦被调度器选中，即可立即进入运行态。

(2) 运行态(running)：任务当前正在占用 CPU 并执行其代码逻辑。在单核处理器系统中，同一时刻仅有一个任务处于运行态。

当任务因时间片耗尽、主动让出 CPU 或被更高优先级任务抢占时，将退出运行态，转入就绪态或其他状态。

(3) 阻塞态(blocked)：任务因等待某一事件或资源而无法继续执行，不再参与调度。阻塞态是一个广义状态，包含多种具体情形：显式挂起(suspended)、延时等待(delay)、等待同步机制、等待通信事件。

处于阻塞态的任务不占用 CPU 资源，直到等待的条件满足或超时，才会重新进入就绪队列。

(4) 退出态(dead)：任务已执行完成或被删除，其任务控制块(TCB)和栈空间等待系统回收。该状态下的任务不再参与任何调度，资源将在系统空闲时由内核自动释放。

4. 任务优先级

任务优先级表示任务执行的优先顺序。任务优先级决定了在发生任务切换时即将要执行的任务，就绪队列中最高优先级的任务将得到执行。

3.2.4 任务运作机制

LiteOS 的任务运作机制主要包括任务的创建、调度和切换。

用户创建任务时，系统会初始化任务栈，预置上下文。此外，系统还会将“任务入口函数”地址放在相应位置。这样在任务第一次启动进入运行态时，将会执行“任务入口函数”。

通过调用接口，可以动态创建一个新任务，系统会分配 TCB 和任务栈内存，初始化 TCB 中的各项字段，将任务加入就绪队列，触发任务调度。若新任务优先级最高，则立即抢占运行。

LiteOS 使用优先级调度算法，内核维护一个任务就绪队列，调度器从中选择优先级最高的任务投入运行。调度时机包括任务主动让出 CPU、任务进入阻塞态、高优先级任务变为就绪态、时间片耗尽等。

任务切换是任务调度的结果。当任务进行切换时，系统会保存当前任务的上下文，更新当前任务的状态，选择下一个要运行的任务，恢复目标任务的上下文，跳转到目标任务继续执行。

3.2.5 任务使用案例

1. 相关 API 介绍

任务相关 API 列表如表 3.1～表 3.6 所示。

表 3.1 osal_kthread_create()函数接口

定义	osal_task * osal_kthread_create(osal_kthread_handler handler, void * data, const char * name, unsigned int stack_size);
功能	创建一个线程
参数	handler：线程处理函数 data：线程处理函数的参数 name：线程名称 stack_size：线程栈大小
返回值	osal_task *：成功(返回线程句柄) NULL：失败
依赖	kernel/osal/include/schedule/osal_task.h

表 3.2 osal_kthread_set_priority()函数接口

定义	int osal_kthread_set_priority(osal_task * task, unsigned int priority);
功能	设置线程优先级
参数	task：需要设置优先级的线程句柄 priority：优先级
返回值	OSAL_SUCCESS(0)：成功 OSAL_FAILURE(-1)：失败
依赖	kernel/osal/include/schedule/osal_task.h

表 3.3 osal_kthread_lock()函数接口

定义	void osal_kthread_lock(void);
功能	锁定任务调度(禁止任务切换)
参数	无
返回值	无
依赖	kernel/osal/include/schedule/osal_task.h

表 3.4 osal_kthread_unlock()函数接口

定义	void osal_kthread_unlock(void);
功能	解锁任务调度(允许任务切换)
参数	无
返回值	无
依赖	kernel/osal/include/schedule/osal_task.h

表 3.5 osal_kthread_suspend()函数接口

定义	void osal_kthread_suspend(osal_task *task);
功能	挂起指定任务(将任务从就绪队列中移除)
参数	task:需要挂起的线程句柄
返回值	无
依赖	kernel/osal/include/schedule/osal_task.h

表 3.6 osal_kthread_resume()函数接口

定义	void osal_kthread_resume(osal_task *task);
功能	恢复挂起的任务
参数	task:需要恢复的线程句柄
返回值	无
依赖	kernel/osal/include/schedule/osal_task.h

2. 任务完整案例

本案例将使用相关 API 创建两个不同优先级的任务,进行任务延时、任务锁定与解锁调度、挂起和恢复等操作。

1) 头文件

头文件需包含"securec.h"、"osal_task.h"、"osal_debug.h"、"app_init.h"、"soc_osal.h"、"stdint.h"和"tcxo.h",用于安全函数、任务管理、调试输出、应用初始化、操作系统抽象层、标准整数类型和时钟控制功能。

2）任务参数配置

定义测试线程使用的栈大小为 0x1000 字节，高优先级线程的优先级为 24，低优先级线程的优先级为 25（注意：数值越小则优先级越高）：

```
#define TEST_THREAD_STACK_SIZE 0x1000          //测试线程使用的栈大小,单位为字节
#define TEST_THREAD_PRIO_HI 24                 //高优先级线程的优先级
#define TEST_THREAD_PRIO_LOW 25                //低优先级线程的优先级
```

3）全局变量定义

设置高优先级任务句柄 task_handle_high 和低优先级任务句柄 task_handle_low，用于存储创建的线程对象，便于后续对线程进行操作。

4）线程函数定义

static void *test_thread_high(void)：高优先级线程入口函数，负责打印启动信息，延时 2s 后挂起自身，等待其他线程恢复。

static void *test_thread_low(void)：低优先级线程入口函数，负责打印启动信息，延时 2s 后恢复高优先级线程。

5）任务入口函数

（1）初始化任务环境：

```
osal_kthread_lock();          //锁定任务调度,防止在创建线程时发生上下文切换
```

（2）创建并配置低优先级线程：

```
task_handle_low = osal_kthread_create((osal_kthread_handler)test_thread_low,0,"test_
thread_low", TEST_THREAD_STACK_SIZE);          //创建低优先级线程
osal_kthread_set_priority(task_handle_low, TEST_THREAD_PRIO_LOW);     //设置线程优先级
                                                                      //为低优先级
```

（3）创建并配置高优先级线程：

```
task_handle_high = osal_kthread_create((osal_kthread_handler)test_thread_high,0,"test_
thread_high", TEST_THREAD_STACK_SIZE);         //创建高优先级线程
osal_kthread_set_priority(task_handle_high, TEST_THREAD_PRIO_HI);     //设置线程优先级
                                                                      //为高优先级
```

6）解锁任务调度

```
osal_kthread_unlock();        //解锁任务调度,允许线程切换
```

7）线程同步机制

高优先级线程通过 osal_kthread_suspend()挂起自身，等待低优先级线程通过 osal_kthread_resume()将其唤醒，实现线程间的同步控制。

8）应用启动注册

通过 app_run(TaskEntry)将 TaskEntry 函数注册为应用启动任务，系统启动时会自动执行该函数。

任务案例流程图如图 3.2 所示。

3. 案例运行现象

任务运行案例串口信息如图 3.3 所示。

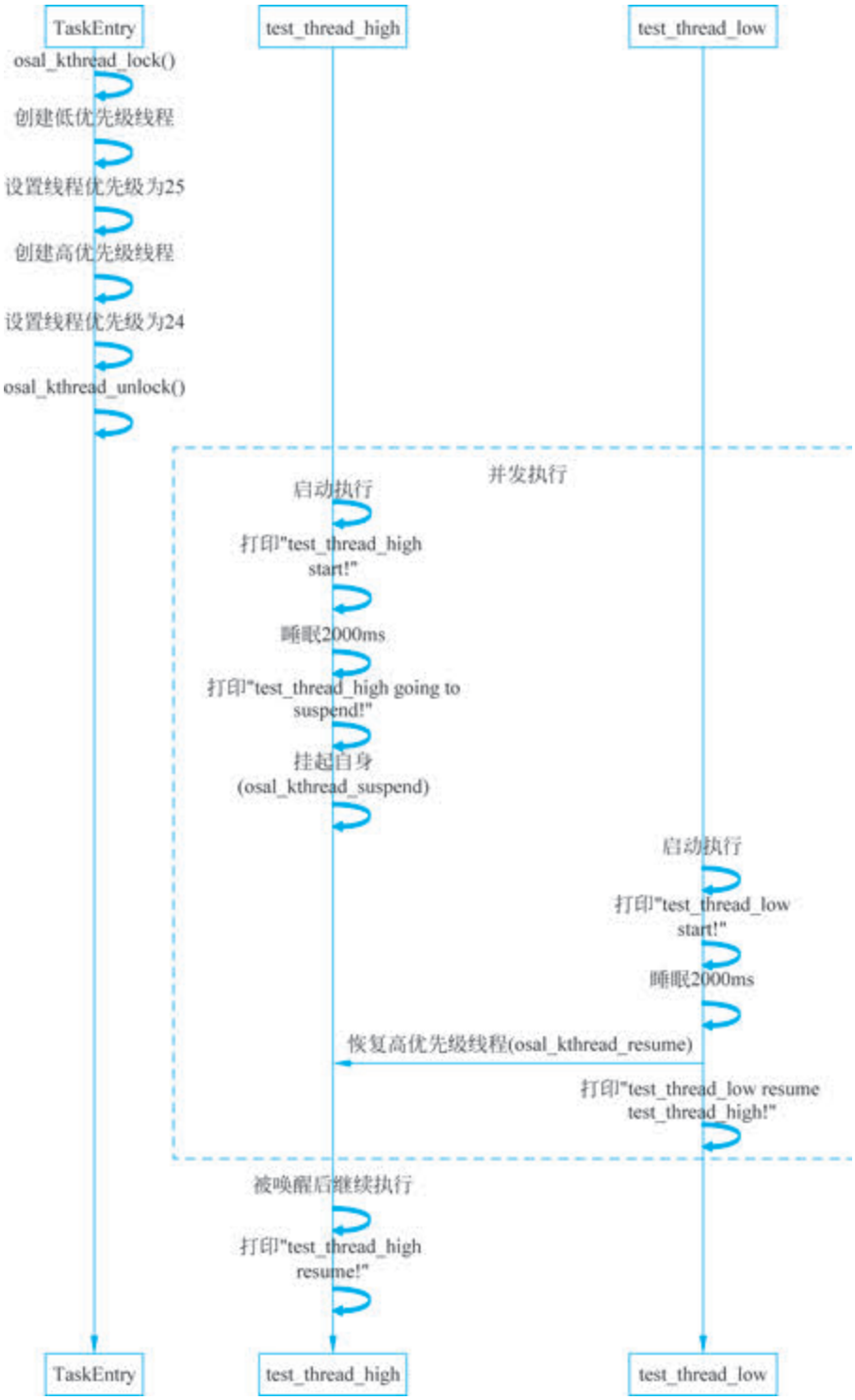
TaskEntry
test_thread_high
test_thread_low
osal_kthread_lock()
创建低优先级线程
设置线程优先级为25
创建高优先级线程
设置线程优先级为24
osal_kthread_unlock()
并发执行
启动执行
打印"test_thread_high start!"
睡眠2000ms
打印"test_thread_high going to suspend!"
挂起自身(osal_kthread_suspend)
启动执行
打印"test_thread_low start!"
睡眠2000ms
恢复高优先级线程(osal_kthread_resume)
打印"test_thread_low resume test_thread_high!"
被唤醒后继续执行
打印"test_thread_high resume!"
TaskEntry
test_thread_high
test_thread_low

图 3.2 任务案例流程图

```
cpu 0 entering scheduler
APP|btc open
test_thread_high start!
device_main_init: 0!
==hal_initialize_phy==226==
device_module_init:: succ!
cali_set_cali_mask:old[0x0] -> new[0x1fa2]

fe_rf_initialize
cali_offline_cali_entry enter
cali_set_cali_done_flag:old[0x0] -> new[0x1]

rf cali OK  time cost:22, ret:0
test_thread_low start!
test_thread_high going to suspend!
test_thread_high resume!
test_thread_low resume test_thread_high!
APP|[STS INFO] mem: used:76208, free:335156; log: drop/all[0/0], at_recv 0.
APP|[STS INFO] mem: used:76208, free:335156; log: drop/all[0/0], at_recv 0.
APP|[STS INFO] mem: used:76208, free:335156; log: drop/all[0/0], at_recv 0.
```

图 3.3　任务运行案例串口信息

3.3　队列

3.3.1　概述

在嵌入式实时操作系统中，队列(queue)是一种重要的任务间通信机制，用于在不同任务或中断服务程序之间传递消息或数据。

3.3.2　队列基本概念

队列是用于实现任务间异步通信的重要机制之一。它允许一个任务向另一个任务发送消息或数据，而无需双方直接同步执行。队列通常用于任务之间的数据交换，中断服务例程与任务之间的信息传递，实现生产者-消费者模型。

队列具有异步性。队列的发送方和接收方不需要同时处于运行态，接收方可以在发送方完成后的任意时间点读取消息。

队列具有缓冲能力，可以缓存一定数量的消息，避免因接收方暂时不可用而导致的数据丢失。

3.3.3　队列相关概念

为了深入理解 LiteOS 的队列机制，还需要掌握以下几个关键概念。

1. 消息结构

每个队列中的消息结构不是固定的，通常会使用结构体进行配置，可由开发者自行配置。

2. 队列控制块

每个队列都有一个队列控制块，用于存储队列的状态信息和管理队列中的消息。每个队列控制块都包含 11 个字段，分别是：队列指针，队列状态，创建队列时内存的分配方

式，队列长度，消息节点大小，队列 ID，消息头节点位置，消息尾节点位置，可读/写消息数，任务等待链表，内存块链表。

3. 队列的阻塞行为

对全部空闲队列进行读操作时，会引起任务挂起。

3.3.4 队列运作机制

1. 创建队列

当队列创建成功后，系统会返回一个唯一的队列 ID，并创建空间。

2. 写入消息（入队）

写队列时，不能对已满队列进行写操作，支持头部写入和尾部写入两种方式。尾部写入时，根据 Tail 找到起始空闲消息节点作为数据写入对象。头部节点写入时，将 Head 的前一个节点作为数据写入对象。

3. 读取消息（出队）

读队列时，首先要判断队列是否有消息需要读取，对全部空闲队列进行读操作会引起任务挂起，如果队列可以读取消息，则根据 Head 找到最先写入队列的消息节点进行读取。

队列读写操作示意图如图 3.4 所示。

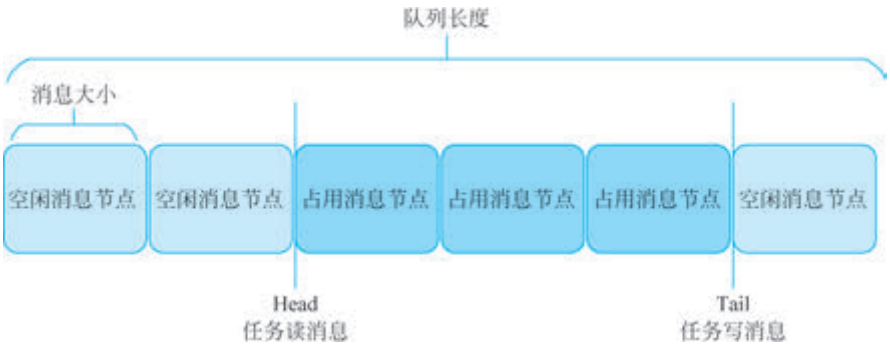

图 3.4 队列读写操作示意图

3.3.5 队列使用案例

1. 相关 API 介绍

队列相关 API 列表如表 3.7～表 3.9 所示。

表 3.7 osal_msg_queue_create()函数接口

定义	int osal_msg_queue_create(const char * name, unsigned short queue_len, unsigned long * queue_id, unsigned int flags, unsigned short max_msgsize);
功能	创建消息队列
参数	name：消息队列名称（保留参数，暂未使用） queue_len：队列长度（消息数量） queue_id：输出参数，返回创建成功的消息队列 ID

续表

参数	flags：队列模式(保留参数,暂未使用) max_msgsize：每个消息的最大字节数
返回值	OSAL_SUCCESS(0)：成功　OSAL_FAILURE(－1)：失败
依赖	kernel/osal/include/msgqueue/osal_msgqueue.h

表 3.8　osal_msg_queue_write_copy()函数接口

定义	int osal_msg_queue_write_copy(unsigned long queue_id, void * buffer_addr, unsigned int buffer_size, unsigned int timeout)；
功能	向消息队列写入数据(复制方式)
参数	queue_id：消息队列 ID buffer_addr：待写入数据的缓冲区地址 buffer_size：待写入数据的大小 timeout：超时时间(单位：Tick)
返回值	OSAL_SUCCESS(0)：成功　OSAL_FAILURE(－1)：失败
依赖	kernel/osal/include/msgqueue/osal_msgqueue.h

表 3.9　osal_msg_queue_read_copy()函数接口

定义	int osal_msg_queue_read_copy(unsigned long queue_id, void * buffer_addr, unsigned int * buffer_size, unsigned int timeout)；
功能	从消息队列读取数据(复制方式)
参数	queue_id：消息队列 ID buffer_addr：存储读取数据的缓冲区地址 buffer_size：输入时为缓冲区大小,输出时为实际读取的数据大小 timeout：超时时间(单位：Tick)
返回值	OSAL_SUCCESS(0)：成功　OSAL_FAILURE(－1)：失败
依赖	kernel/osal/include/msgqueue/osal_msgqueue.h

2. 任务完整案例

本案例将使用相关 API 创建两个不同优先级的任务,一个是生产者线程,另一个是消费者线程,生产者线程负责写消息,消费者线程负责读消息。

1) 头文件

头文件需包含"securec.h"、"osal_task.h"、"osal_msgqueue.h"、"osal_debug.h"、"app_init.h"、"soc_osal.h",用于安全函数、任务管理、消息队列管理、调试输出、应用初始化和操作系统抽象层。

2) 队列参数配置

定义测试线程使用的栈大小为 0x1000 字节,高优先级线程的优先级为 24,低优先级线程的优先级为 25(注意：数值越小优先级越高),消息队列长度为 10,最大消息大小为 test_msg_t：

```
#define TEST_THREAD_STACK_SIZE 0x1000          //测试线程使用的栈大小,单位为字节
#define TEST_THREAD_PRIO_HI 24                 //高优先级线程的优先级
#define TEST_THREAD_PRIO_LOW 25                //低优先级线程的优先级
#define MSG_QUEUE_LEN 10                       //消息队列长度
#define MAX_MSG_SIZE sizeof(test_msg_t)        //最大消息大小
```

3）全局变量定义

设置高优先级任务句柄 task_handle_high 和低优先级任务句柄 task_handle_low，用于存储创建的线程对象；定义 unsigned long 类型的全局队列 ID g_test_queue_id，用于标识创建的消息队列。

4）消息结构体定义

```
typedef struct {
    int id;                 //消息 ID
    char data[32];          //消息数据内容
} test_msg_t;               //测试消息结构体
```

5）线程函数定义

static void producer_task(void * param)：生产者线程函数，负责构造消息并将其发送到消息队列，每 1s 发送一次消息。

static void consumer_task(void * param)：消费者线程函数，负责从消息队列接收消息并处理，每 1.5s 处理一次消息。

6）任务入口函数

（1）初始化任务环境：

```
osal_kthread_lock();        //锁定任务调度，防止在创建线程时发生上下文切换
```

（2）创建消息队列：

```
osal_msg_queue_create("test_queue", MSG_QUEUE_LEN, &g_test_queue_id, 0, MAX_MSG_SIZE);
                            //创建消息队列
```

（3）创建并配置生产者线程（低优先级）：

```
task_handle_low = osal_kthread_create((osal_kthread_handler)producer_task, 0, "producer_
task", TEST_THREAD_STACK_SIZE);              //创建生产者线程
osal_kthread_set_priority(task_handle_low, TEST_THREAD_PRIO_LOW);      //设置线程优先级
                                                                       //为低优先级
```

（4）创建并配置消费者线程（高优先级）：

```
task_handle_high = osal_kthread_create((osal_kthread_handler)consumer_task, 0, "consumer_
task", TEST_THREAD_STACK_SIZE);              //创建消费者线程
osal_kthread_set_priority(task_handle_high, TEST_THREAD_PRIO_HI);      //设置线程优先级
                                                                       //为高优先级
```

（5）解锁任务调度：

```
osal_kthread_unlock();      //解锁任务调度，允许线程切换
```

7）消息队列操作

生产者使用 osal_msg_queue_write_copy()将消息写入队列，消费者使用 osal_msg_queue_read_copy()从队列读取消息，均使用 OSAL_MSGQ_WAIT_FOREVER 参数表示无限期等待。

8）应用启动注册

通过 app_run(TaskEntry)将 TaskEntry 函数注册为应用启动任务，系统启动时会自动执行该函数。

队列案例流程图如图 3.5 所示。

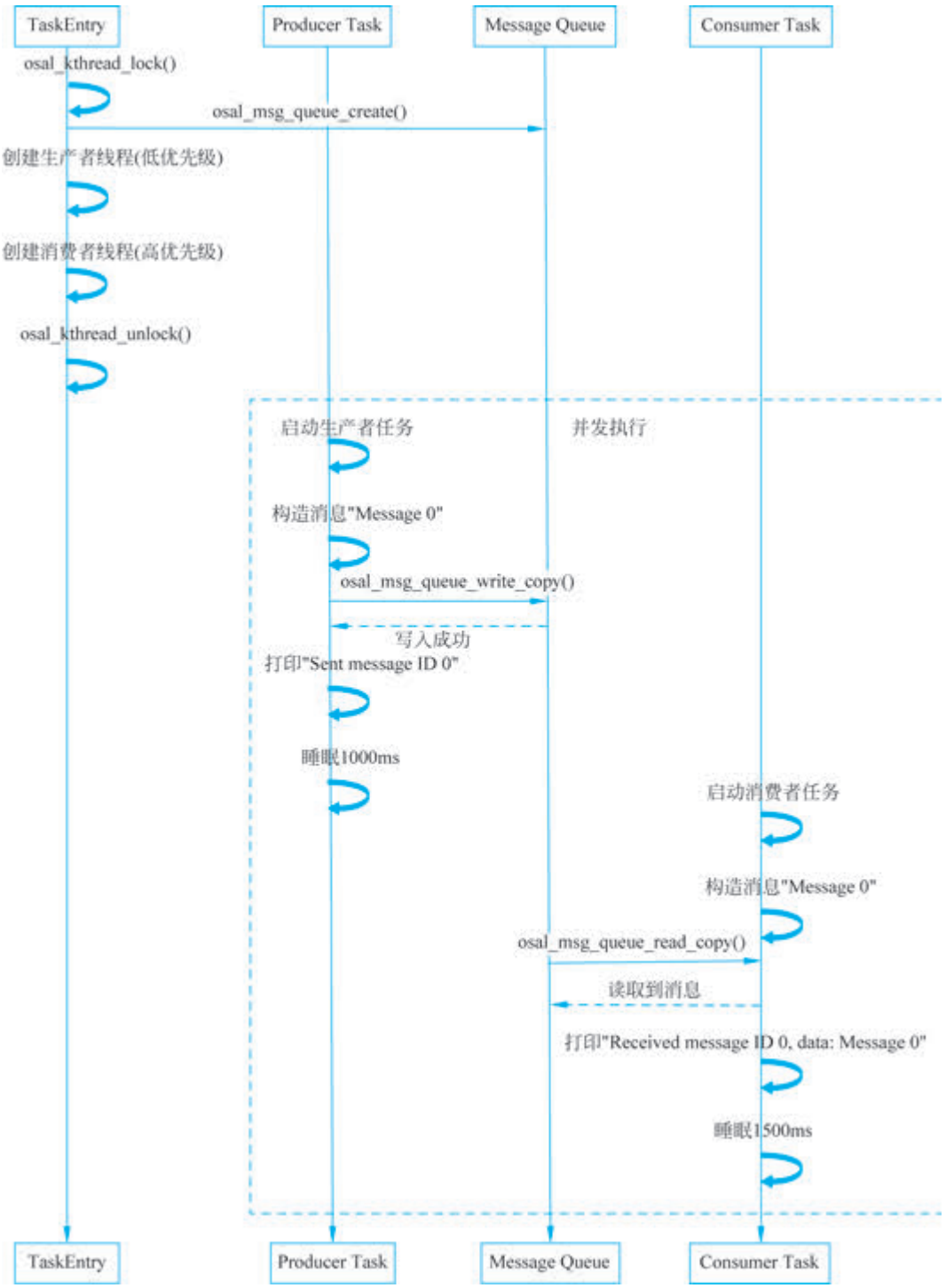

图 3.5　队列案例流程图

3. 案例运行现象

队列运行案例串口信息如图 3.6 所示。

```
Producer: Sent message ID 3
xo update temp:3,diff:0,xo:0x3083c
Producer: Sent message ID 4
Consumer: Received message ID 3, data: Message 3, msg_size: 36
Producer: Sent message ID 5
Consumer: Received message ID 4, data: Message 4, msg_size: 36
Producer: Sent message ID 6
Producer: Sent message ID 7
Consumer: Received message ID 5, data: Message 5, msg_size: 36
Producer: Sent message ID 8
Consumer: Received message ID 6, data: Message 6, msg_size: 36
Producer: Sent message ID 9
Producer: Sent message ID 10
Consumer: Received message ID 7, data: Message 7, msg_size: 36
APP|[SYS INFO] mem: used:84896, free:224308; log: drop/all[0/0], at_recv 0.
Producer: Sent message ID 11
Consumer: Received message ID 8, data: Message 8, msg_size: 36
Producer: Sent message ID 12
Producer: Sent message ID 13
```

图 3.6 队列运行案例串口信息

3.4 事件

3.4.1 概述

在嵌入式实时操作系统中，事件(event)是一种轻量级、高效的任务间同步与通信机制，常用于协调多个任务的执行顺序、通知特定状态的发生或触发某些动作。

与队列传递“数据”不同，事件主要用于传递“状态”或“信号”，具有开销小、响应快的特点，非常适合对实时性要求高的物联网应用场景。

3.4.2 事件基本概念

事件通常由一个或多个标志位组成，每个标志位可以表示一种特定的状态或条件。事件支持多种操作模式，如等待单个或多个事件、阻塞或非阻塞等待，可以通过逻辑运算符(如 AND、OR)组合多个事件标志位，实现复杂的同步需求。

事件与队列的区别如表 3.10 所示。

表 3.10 事件与队列的区别

特　性	队　列	事　件
用途	用于任务间的数据传输，支持队列消息传递	用于任务间的同步与状态通知，传递的是状态信息而非具体数据
复杂度	相对复杂，需要管理消息的存储与读取	简单轻量，主要处理标志位的操作
应用场景	数据流处理、生产者-消费者模型	任务同步、中断通知、状态监测

3.4.3 事件相关概念

为了深入理解 LiteOS 的事件机制，还需掌握以下几个关键概念。

1. 事件标志位

事件标志位是事件的核心组成部分，每个标志位可以表示一种特定的状态或条件。例如：

标志位 0：表示传感器 A 已准备好数据。

标志位 1：表示传感器 B 已准备好数据。

标志位 2：表示网络连接已建立。

2. 事件读取模式

(1) 所有事件：

逻辑与，基于接口传入的事件类型掩码 eventMask，只有这些事件都已经发生才能读取成功；否则该任务将阻塞等待或者返回错误码。

(2) 任一事件：

逻辑或，基于接口传入的事件类型掩码 eventMask；只要这些事件中有任一种事件发生就可以读取成功；否则该任务将阻塞等待或者返回错误码。

(3) 清除事件：

清除事件是一种附加读取模式，需要与所有事件或任一事件模式结合使用，在这种模式下，当设置的所有事件或任一事件模式读取成功后，会自动清除事件控制块中对应的事件类型位。

3.4.4 事件运作机制

任务调用 osal_event_read()时，可根据传入的事件掩码 eventMask 读取单个或多个事件，并通过模式参数选择等待所有事件(OSAL_WAITMODE_AND)或任一事件(OSAL_WAITMODE_OR)。

若设置 OSAL_WAITMODE_CLR 标志，读取成功后系统会自动清除已匹配的事件位，否则事件位保持不变，需后续显式调用 LOS_EventClear 进行清除。

任务或中断通过 osal_event_write()向指定事件写入一个或多个事件类型，该操作会立即更新事件掩码，并触发任务调度，唤醒正在等待对应事件的高优先级任务。

osal_event_clear()用于根据指定掩码将事件中的对应位清零，实现事件状态的主动重置。

事件唤醒任务示意图如图 3.7 所示。

3.4.5 事件使用案例

1. 相关 API 介绍

事件相关 API 列表如表 3.11～表 3.15 所示。

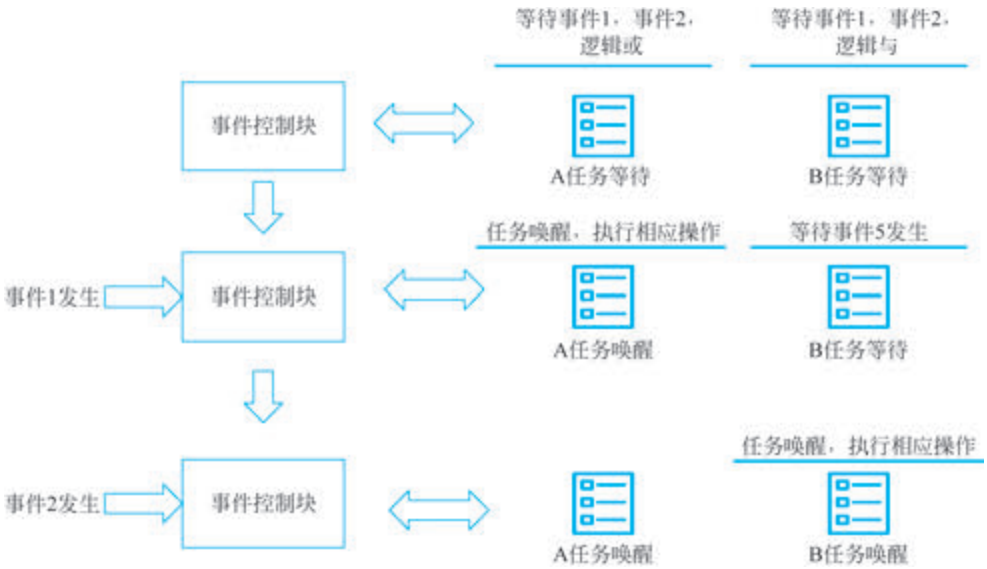

图 3.7　事件唤醒任务示意图

表 3.11　osal_event_init()函数接口

定义	int osal_event_init(osal_event * event_obj)；
功能	初始化事件控制块
参数	event_obj：指向需要初始化的事件控制块指针
返回值	OSAL_SUCCESS(0)：成功　OSAL_FAILURE(−1)：失败
依赖	kernel/osal/include/event/osal_event.h

表 3.12　osal_event_write()函数接口

定义	int osal_event_write(osal_event * event_obj，unsigned int mask)；
功能	写入事件(触发事件)
参数	event_obj：指向事件控制块的指针 mask：事件掩码，指定要触发的事件位
返回值	OSAL_SUCCESS(0)：成功　OSAL_FAILURE(−1)：失败
依赖	kernel/osal/include/event/osal_event.h

表 3.13　osal_event_read()函数接口

定义	int osal_event_read(osal_event * event_obj，unsigned int mask，unsigned int timeout_ms，unsigned int mode)；
功能	读取事件(等待事件)
参数	event_obj：指向事件控制块的指针 mask：期望发生的事件掩码 timeout_ms：超时时间(单位：毫秒) mode：事件读取模式(如 OSAL_WAITMODE_AND、OSAL_WAITMODE_OR、OSAL_WAITMODE_CLR)
返回值	OSAL_SUCCESS(0)：成功　OSAL_FAILURE(−1)：失败
依赖	kernel/osal/include/event/osal_event.h

表 3.14　osal_event_clear()函数接口

定义	int osal_event_clear(osal_event * event_obj, unsigned int mask);
功能	清除事件
参数	event_obj：指向事件控制块的指针 mask：需要清除的事件掩码
返回值	OSAL_SUCCESS(0)：成功　OSAL_FAILURE(−1)：失败
依赖	kernel/osal/include/event/osal_event.h

表 3.15　osal_event_destroy()函数接口

定义	int osal_event_destroy(osal_event * event_obj);
功能	销毁事件控制块
参数	event_obj：指向需要销毁的事件控制块指针
返回值	OSAL_SUCCESS(0)：成功　OSAL_FAILURE(−1)：失败
依赖	kernel/osal/include/event/osal_event.h

2. 任务完整案例

本案例将使用相关 API 创建两个不同优先级的任务，通过事件进行任务之间的同步。

1）头文件

头文件需包含"securec.h"、"osal_task.h"、"osal_event.h"、"osal_debug.h"、"app_init.h"和"soc_osal.h"，用于安全函数、任务管理、事件管理、调试输出、应用初始化、操作系统抽象层。

2）任务参数配置

定义测试线程使用的栈大小为 0x1000 字节，高优先级线程的优先级为 24，低优先级线程的优先级为 25(注意：数值越小则优先级越高)：

```
#define TEST_THREAD_STACK_SIZE 0x1000          //测试线程使用的栈大小,单位为字节
#define TEST_THREAD_PRIO_HI 24                 //高优先级线程的优先级
#define TEST_THREAD_PRIO_LOW 25                //低优先级线程的优先级
#define EVENT_1 0x01                           //事件 1 标识
#define EVENT_2 0x02                           //事件 2 标识
```

3）全局变量定义

设置高优先级任务句柄 task_handle_high 和低优先级任务句柄 task_handle_low，用于存储创建的线程对象；定义 osal_event 类型的全局事件对象 g_event，用于线程间的同步。

4）线程函数定义

static void * test_thread_high(void)：高优先级线程入口函数，负责写入 EVENT_1 事件，然后等待 EVENT_2 事件，在接收到 EVENT_2 后再次写入 EVENT_1 事件。

static void * test_thread_low(void)：低优先级线程入口函数，负责循环等待 EVENT_1 或 EVENT_2 事件，接收到事件后进行处理，并继续等待 EVENT_1 事件，实现持续的事件处理循环。

5）任务入口函数

（1）初始化任务环境：

```
osal_kthread_lock();                    //锁定任务调度,防止在创建线程时发生上下文切换
osal_event_init(&g_event);              //初始化事件对象
```

（2）创建并配置低优先级线程：

```
task_handle_low = osal_kthread_create((osal_kthread_handler)test_thread_low,0,"test_
thread_low", TEST_THREAD_STACK_SIZE);      //创建低优先级线程
osal_kthread_set_priority(task_handle_low, TEST_THREAD_PRIO_LOW);      //设置线程优先级
                                                                       //为低优先级
```

（3）创建并配置高优先级线程：

```
task_handle_high = osal_kthread_create((osal_kthread_handler)test_thread_high,0,"test_
thread_high", TEST_THREAD_STACK_SIZE);      //创建高优先级线程
osal_kthread_set_priority(task_handle_high, TEST_THREAD_PRIO_HI);      //设置线程优先级
                                                                       //为高优先级
```

（4）解锁任务调度：

```
osal_kthread_unlock();              //解锁任务调度,允许线程切换
```

6）事件处理机制

通过 osal_event_write()写入事件，通过 osal_event_read()等待并读取事件，使用 OSAL_WAIT_FOREVER 表示无限期等待，OSAL_WAITMODE_OR 表示等待任一事件，OSAL_WAITMODE_AND 表示等待所有指定事件，OSAL_WAITMODE_CLR 表示读取后自动清除事件标志。

7）应用启动注册

通过 app_run(TaskEntry)将 TaskEntry 函数注册为应用启动任务，系统启动时会自动执行该函数。

事件案例流程图如图 3.8 所示。

3. 案例运行现象

程序运行时，首先初始化事件对象，然后创建高优先级线程和低优先级线程，高优先级线程首先进入，写入事件 1，并等待事件 2。然后，低优先级线程进入，等待事件 1 或事件 2，由于已经在高优先级线程写入事件 1，所以读取事件成功，随后写入事件 2。高优先级线程读取事件 2 后，写入事件 1，触发任务调度。

低优先级线程读取事件 1 后，写入事件 2，等待读取事件 1，读取事件 1 成功后，结束整个线程。事件运行案例串口信息如图 3.9 所示。

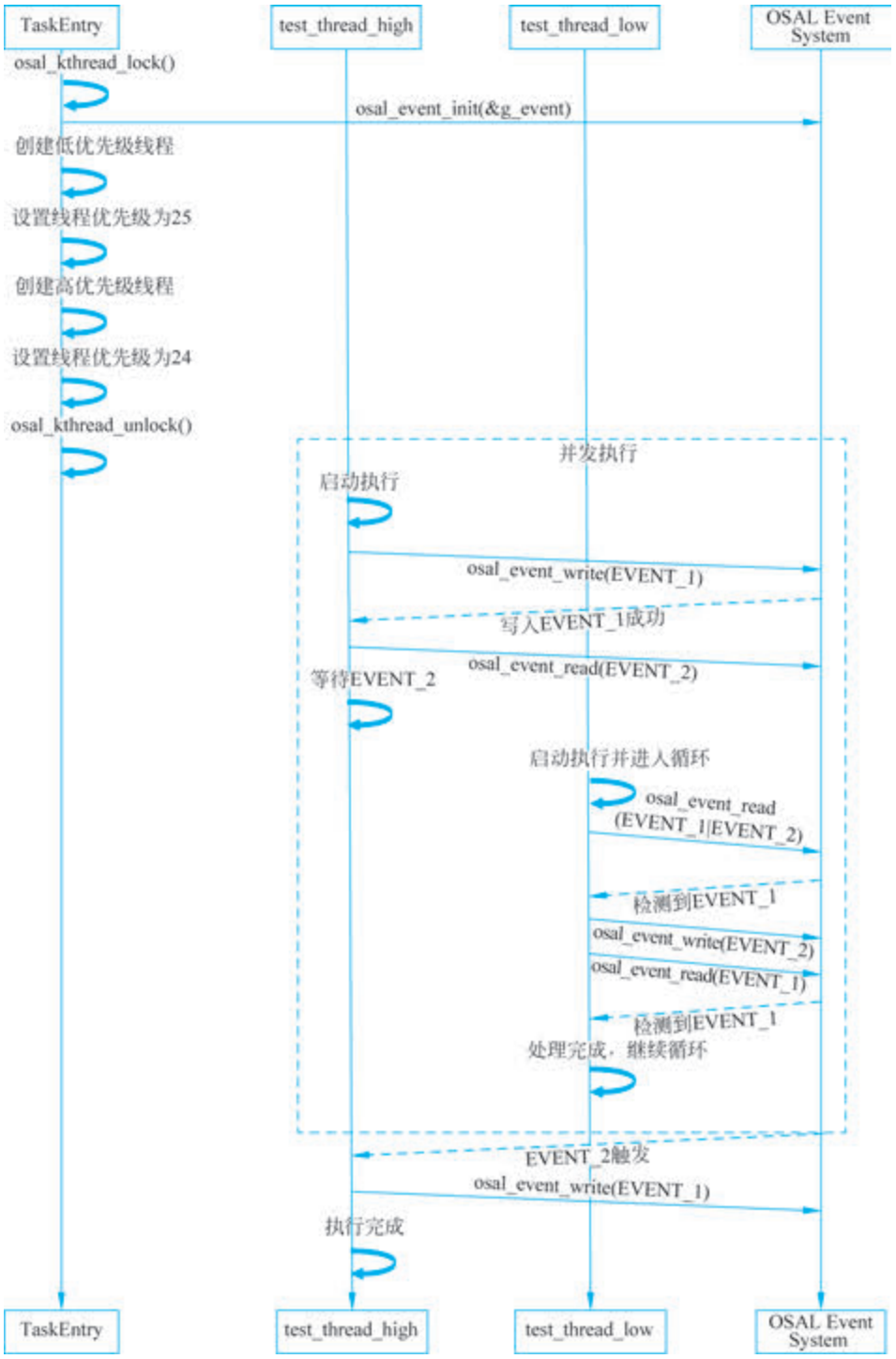
TaskEntry
test_thread_high
test_thread_low
OSAL Event System
osal_kthread_lock()
osal_event_init(&g_event)
创建低优先级线程
设置线程优先级为25
创建高优先级线程
设置线程优先级为24
osal_kthread_unlock()
并发执行
启动执行
osal_event_write(EVENT_1)
写入EVENT_1成功
osal_event_read(EVENT_2)
等待EVENT_2
启动执行并进入循环
osal_event_read
(EVENT_1|EVENT_2)
检测到EVENT_1
osal_event_write(EVENT_2)
osal_event_read(EVENT_1)
检测到EVENT_1
处理完成，继续循环
EVENT_2触发
osal_event_write(EVENT_1)
执行完成
TaskEntry
test_thread_high
test_thread_low
OSAL Event System

图 3.8　事件案例流程图

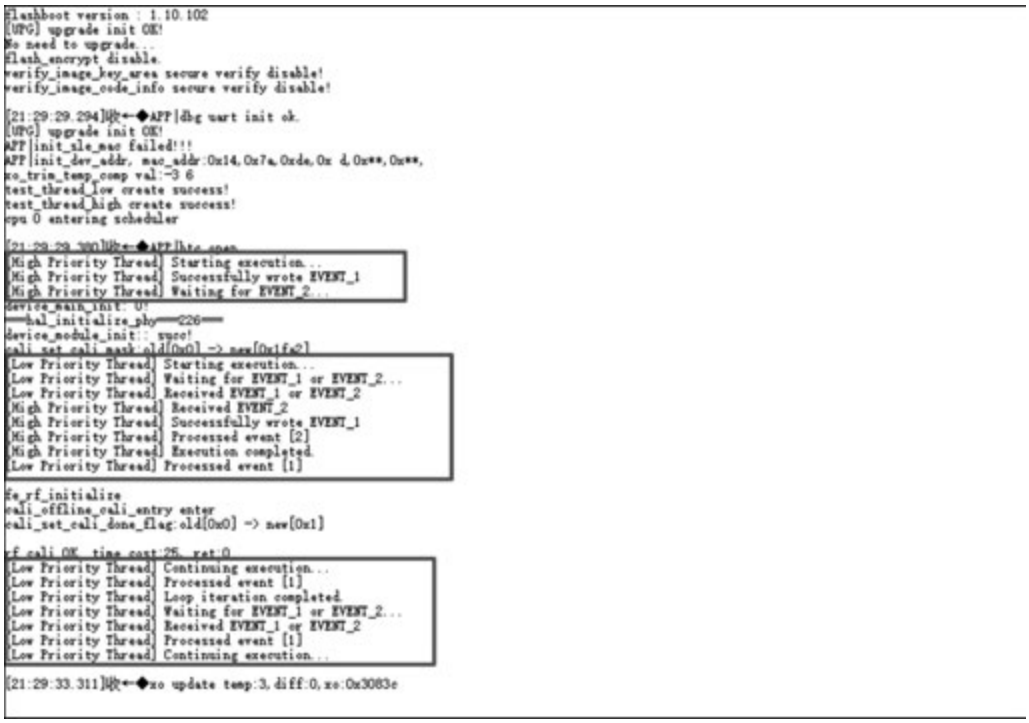

图 3.9　事件运行案例串口信息

3.5 互斥锁

3.5.1 概述

在嵌入式实时操作系统中，互斥锁（mutex）常用于协调多个任务对共享资源的访问，以确保数据的一致性和完整性。

与队列传递“数据”不同，互斥锁主要用于传递“独占权”，具有开销小、响应快的特点，非常适合对实时性要求高的物联网应用场景。

3.5.2 互斥锁基本概念

互斥锁可用于保护共享资源免受并发访问导致的数据不一致问题。

互斥锁提供了对临界区的独占访问权限，确保同一时刻只有一个任务能够进入该区域执行代码。

临界区是指那些需要独占访问共享资源的代码段，任务在访问临界区之前必须先获取互斥锁，在完成操作后释放互斥锁，以便其他等待的任务可以继续执行。

互斥锁支持多种操作模式，如阻塞或非阻塞等待，可以通过递归锁功能实现同一个任务多次获取同一把互斥锁而不发生死锁。

互斥锁与队列的区别如表 3.16 所示。

表 3.16　互斥锁与队列的区别

特　性	队　列	互　斥　锁
用途	用于任务间的数据传输，支持队列消息传递	用于任务间的同步与独占访问控制，保护共享资源
复杂度	相对复杂，需要管理消息的存储与读取	简单轻量，主要处理独占权的操作
应用场景	数据流处理、生产者-消费者模型	共享资源保护、硬件设备访问控制

3.5.3　互斥锁相关概念

为了深入理解 LiteOS 的互斥锁机制，还需掌握以下几个关键概念。

1. 互斥锁状态

互斥锁的状态通常分为两种：未使用状态，即互斥锁尚未被使用，可以获取；正在使用状态，即互斥锁已被创建并正在使用中。

2. 互斥锁操作模式

1）获取互斥锁

任务在访问临界区之前必须先获取互斥锁。如果当前没有其他任务持有该互斥锁，则调用立即成功；否则，调用任务会被挂起，直到获得互斥锁为止。

2）释放互斥锁

任务在完成对临界区的操作后必须释放互斥锁，以便其他等待的任务可以继续执行。

3）递归锁支持

允许同一个任务多次获取同一把互斥锁而不发生死锁。

3. 优先级继承

当高优先级任务被低优先级任务持有的互斥锁阻塞时，系统会提升低优先级任务的优先级以减少优先级反转现象，这种机制称为优先级继承。

3.5.4　互斥锁运作机制

在多任务环境下，当多个任务需要访问同一个非共享的临界区资源时，互斥锁通过“上锁-解锁”机制来确保资源的独占访问。

初始状态下的互斥锁处于解锁状态，允许任务获取。一旦某个任务成功获取（加锁）互斥锁并进入临界区，该锁即进入加锁状态，其他试图获取该锁的任务将被阻塞并挂起，无法进入临界区。只有当持有互斥锁的任务完成对临界资源的操作并主动释放互斥锁（解锁）后，系统才会唤醒等待队列中的一个任务（通常为优先级最高者），使其获取互斥锁并进入临界区。

互斥锁运作示意图如图 3.10 所示。

3.5.5　互斥锁使用案例

1. 相关 API 介绍

互斥锁相关 API 列表如表 3.17～表 3.19 所示。

图 3.10　互斥锁运作示意图

表 3.17　osal_mutex_init()函数接口

定义	int osal_mutex_init(osal_mutex * mutex);
功能	初始化互斥锁
参数	mutex：指向需要初始化的互斥锁结构体指针
返回值	OSAL_SUCCESS(0)：成功　OSAL_FAILURE(−1)：失败
依赖	kernel/osal/include/lock/osal_mutex.h

表 3.18　osal_mutex_lock_timeout()函数接口

定义	int osal_mutex_lock_timeout(osal_mutex * mutex, unsigned int timeout);
功能	获取互斥锁(带超时机制)
参数	mutex：指向互斥锁结构体指针 timeout：等待超时时间(单位：Tick)
返回值	OSAL_SUCCESS(0)：成功　OSAL_FAILURE(−1)：失败
依赖	kernel/osal/include/lock/osal_mutex.h

表 3.19　osal_mutex_unlock()函数接口

定义	void osal_mutex_unlock(osal_mutex * mutex);
功能	释放互斥锁
参数	mutex：指向互斥锁结构体指针
返回值	无
依赖	kernel/osal/include/lock/osal_mutex.h

2. 任务完整案例

本案例将使用相关 API 创建两个不同优先级的任务，使用互斥锁对临界区资源进行访问。

1) 头文件

头文件需包含"securec.h"、"osal_task.h"、"osal_event.h"、"osal_debug.h"、"app_init.h"、"soc_osal.h"、"osal_mutex.h"和"watchdog.h"，用于实现安全函数、任务管理、事件管理、调试输出、应用初始化、操作系统抽象层、互斥锁管理和看门狗功能。

2) 任务参数配置

定义测试线程使用的栈大小为 0x1000 字节，高优先级线程的优先级为 24，低优先级

线程的优先级为 25(注意：数值越小则优先级越高)：

```
#define TEST_THREAD_STACK_SIZE 0x1000          //测试线程使用的栈大小,单位为字节
#define TEST_THREAD_PRIO_HI 24                 //高优先级线程的优先级
#define TEST_THREAD_PRIO_LOW 25                //低优先级线程的优先级
```

3) 全局变量定义

设置高优先级任务句柄 task_handle_high 和低优先级任务句柄 task_handle_low,用于存储创建的线程对象；定义 osal_mutex 类型的全局互斥锁对象 g_mutex,用于线程间同步。

4) 线程函数定义

static void * test_thread_high(void)：高优先级线程入口函数,负责获取互斥锁,执行临界区代码(延时 1s),然后释放互斥锁。

static void * test_thread_low(void)：低优先级线程入口函数,负责获取互斥锁,执行临界区代码(延时 1s),然后释放互斥锁。

5) 任务入口函数

(1) 初始化互斥锁：

```
osal_mutex_init(&g_mutex);          //初始化互斥锁对象
if (ret != OSAL_SUCCESS) {
    osal_printk("osal_mutex_init failed, %02x\r\n", ret);
}
```

(2) 创建并配置低优先级线程：

```
task_handle_low = osal_kthread_create((osal_kthread_handler)test_thread_low, 0, "test_
thread_low", TEST_THREAD_STACK_SIZE);     //创建低优先级线程
osal_kthread_set_priority(task_handle_low, TEST_THREAD_PRIO_LOW);     //设置线程优先级
                                                                      //为低优先级
osal_printk("test_thread_low create success!\r\n");
```

(3) 创建并配置高优先级线程：

```
task_handle_high = osal_kthread_create((osal_kthread_handler)test_thread_high, 0, "test_
thread_high", TEST_THREAD_STACK_SIZE);     //创建高优先级线程
osal_kthread_set_priority(task_handle_high, TEST_THREAD_PRIO_HI);     //设置线程优先级
                                                                      //为高优先级
osal_printk("test_thread_high create success!\r\n");
```

6) 互斥锁操作

通过 osal_mutex_lock_timeout()获取互斥锁,使用 OSAL_WAIT_FOREVER 参数表示无限期等待；通过 osal_mutex_unlock()释放互斥锁。

7) 线程同步机制

两个线程通过互斥锁 g_mutex 实现对临界区资源的互斥访问。当一个线程获取互斥锁后,另一个线程将阻塞等待,直到第一个线程释放互斥锁为止。

8) 应用启动注册

通过 app_run(TaskEntry)将 TaskEntry 函数注册为应用启动任务,系统启动时会自动执行该函数。

互斥锁案例流程图如图 3.11 所示。

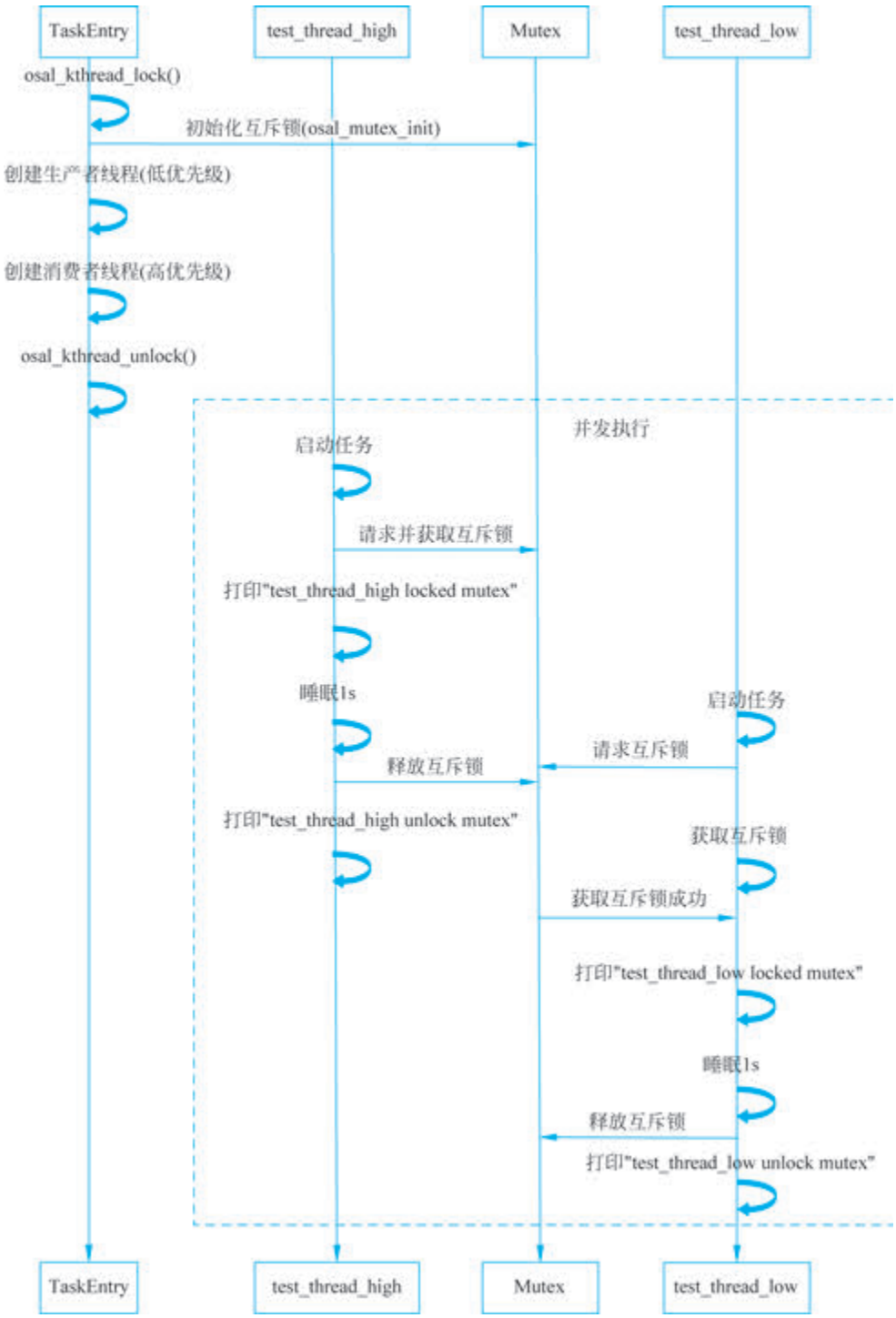

图 3.11　互斥锁案例流程图

3. 案例运行现象

互斥锁运行案例串口信息如图 3.12 所示。

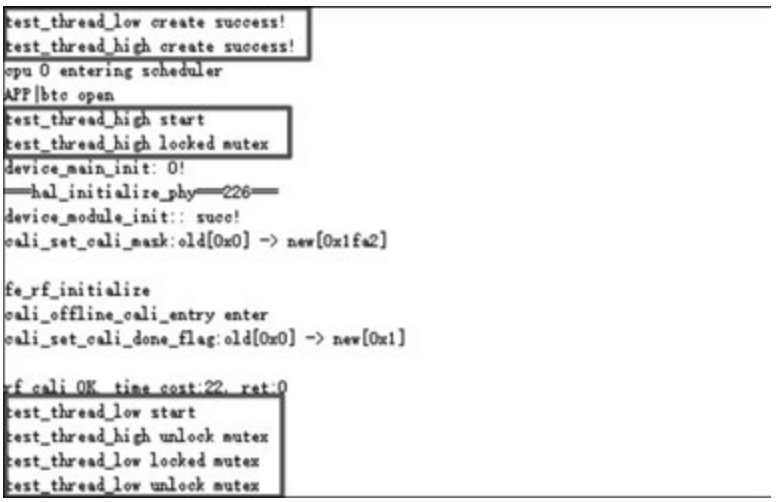

图 3.12　互斥锁运行案例串口信息

3.6　信号量

3.6.1　概述

在嵌入式实时操作系统中，信号量用于对共享资源访问的控制，是重要的同步机制之一。它通过维护一个计数器来管理多个任务对有限资源的访问，可以有效避免资源竞争和死锁。

3.6.2　信号量基本概念

信号量由 Dijkstra 提出，它通过一个计数器来表示资源的数量或状态，允许任务在访问资源之前进行等待，并在完成后释放资源。这个计数器与一般的计数器不同，除初始化之外，仅能通过两个标准的原子操作进行，这两个原子操作分别为 wait()与 signal()，也称为 P、V 操作。其中操作 wait()称为 P 操作(荷兰语 proberen，测试)，操作 signal()称为 V 操作(荷兰语 verhogen，增加)。

在 LiteOS 中，信号量分为两种类型：二值信号量和计数信号量。二值信号量类似于互斥锁，但没有所有权概念，通常用于任务间的同步，而不是资源保护；计数信号量可用于管理多个相同类型的资源，当计数大于 0 时，表示有可用资源；当计数等于 0 时，表示资源不可用，任务需等待。计数信号量适用于需要限制同时访问资源数量的场景。

信号量通常用于任务间的同步、资源管理、事件通知等。信号量与互斥锁的区别如表 3.20 所示。

表 3.20 信号量与互斥锁的区别

特　性	信　号　量	互　斥　锁
用途	主要用于任务间的同步和资源管理	主要用于保护临界区，防止数据竞争
所有权	没有所有权概念	具有所有权概念，只能由持有者解锁

3.6.3 信号量相关概念

为了深入理解 LiteOS 的信号量机制，还需掌握以下几个关键概念。

1. 信号量控制块

信号量控制块用于管理信号的数据结构，包含信号量的状态、类型、计数值等信息。每个信号量都有一个与之对应的控制块，用于存储和管理信号量的各种信息。

2. 信号量的申请模式

(1) 无阻塞模式：在申请信号量时，传入的等待时间为 0。若当前信号量计数值不为 0，则申请成功，反之失败。

(2) 永久阻塞模式：在申请信号量时，传入的等待时间为 0xFFFFFFFF。若当前信号量计数值不为 0，则申请成功，否则该任务进入阻塞态，系统切换到就绪任务中优先级最高者继续执行。任务进入阻塞态后，直到有其他任务释放该信号量，阻塞任务才会重新得以执行。

(3) 定时阻塞模式：在申请信号量时，传入的等待时间在 0～0xFFFFFFFF 之间。若当前信号量计数值不为 0，则申请成功，否则该任务进入阻塞态，系统切换到就绪任务中优先级最高者继续执行。任务进入阻塞态后，如果超时前有其他任务释放该信号量，则该任务可成功获取信号量并继续执行；若超时前未获取信号量，则接口返回超时错误码。

3.6.4 信号量运作机制

信号量允许多个任务在同一时刻访问共享资源，但是会限制同一时刻访问该资源的最大任务数目。当访问资源的任务达到最大数量时，会阻塞后续其他试图获取该资源的任务，直到有任务释放该信号量。

信号量在初始化时为配置数量的信号量分配内存，并将其初始化为未使用状态，统一管理于未使用链表中。

创建信号量时，从该链表中获取一个信号量并设置初始计数值。

当任务申请信号量时，若其计数值大于 0，则自动减 1 并立即获得资源；若计数值为 0，则任务被阻塞并插入该信号量的等待队列尾部，进入睡眠状态，可设定超时等待。

当其他任务释放信号量时，若无任务等待，则计数值加 1；若有任务等待，则唤醒队列头部的第一个任务，使其获取信号量并恢复运行。

删除信号量时，将其状态设置为未使用并回收至未使用链表。

信号量运作示意图如图 3.13 所示。

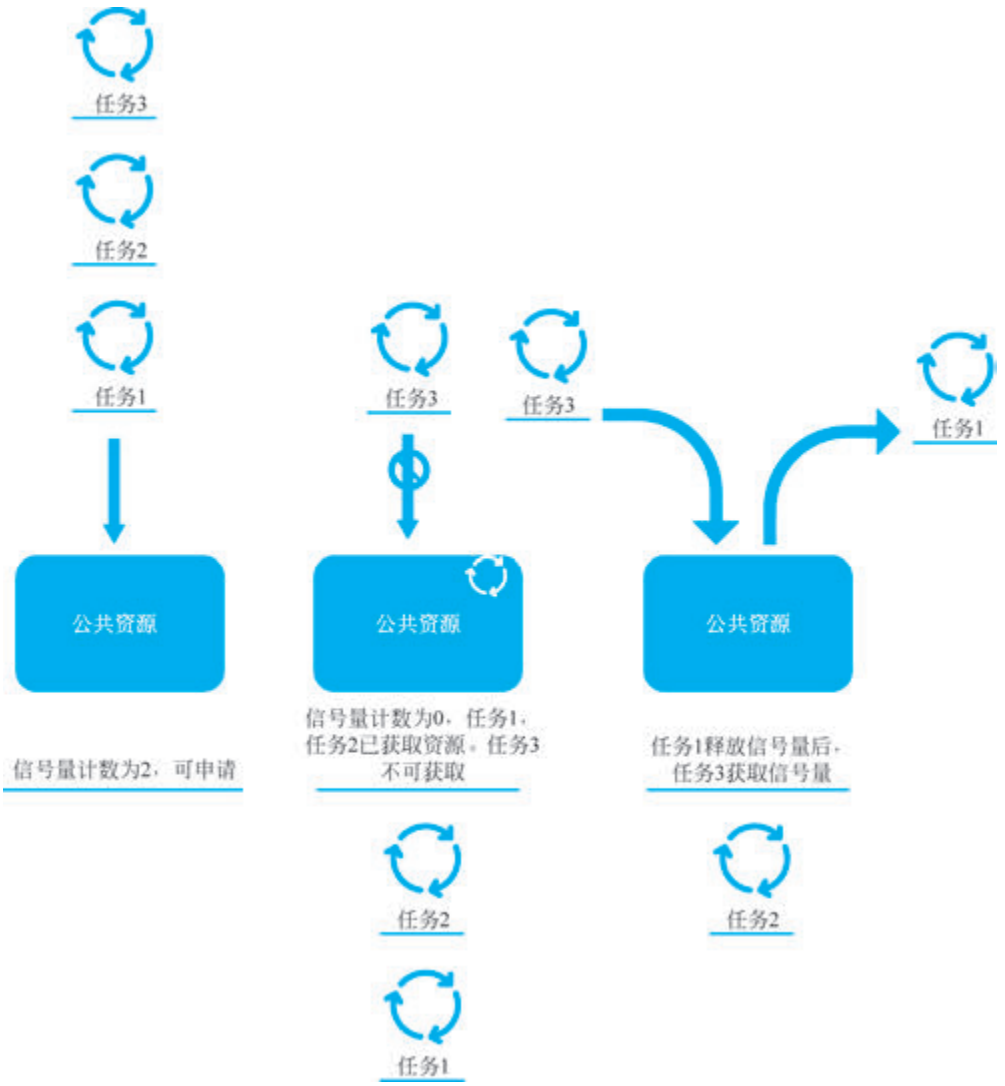

图 3.13　信号量运作示意图

3.6.5　信号量使用案例

1. 相关 API 介绍

信号量相关 API 列表如表 3.21～表 3.23 所示。

表 3.21　osal_sem_init()函数接口

定义	int osal_sem_init(osal_semaphore * sem, int val);
功能	初始化信号量
参数	sem：指向需要初始化的信号量结构体指针 val：信号量的初始值

续表

返回值	OSAL_SUCCESS(0)：成功　OSAL_FAILURE(－1)：失败
依赖	kernel/osal/include/semaphore/osal_semaphore.h

表 3.22　osal_sem_down_timeout()函数接口

定义	int osal_sem_down_timeout(osal_semaphore *sem, unsigned int timeout);
功能	获取信号量(带超时机制)
参数	sem：指向信号量结构体指针 timeout：等待超时时间(单位：ms)
返回值	OSAL_SUCCESS(0)：成功　OSAL_FAILURE(－1)：失败
依赖	kernel/osal/include/semaphore/osal_semaphore.h

表 3.23　osal_sem_up()函数接口

定义	void osal_sem_up(osal_semaphore *sem);
功能	释放信号量
参数	sem：指向信号量结构体指针
返回值	无
依赖	kernel/osal/include/semaphore/osal_semaphore.h

2. 任务完整案例

本案例将使用相关 API 创建两个不同优先级的任务，使用信号量对临界资源进行访问。

1) 头文件

头文件需包含"securec.h"、"osal_task.h"、"osal_event.h"、"osal_debug.h"、"app_init.h"、"soc_osal.h"和"osal_semaphore.h"，用于安全函数、任务管理、事件管理、调试输出、应用初始化、操作系统抽象层和信号量管理功能。

2) 任务参数配置

定义测试线程使用的栈大小为 0x1000 字节，高优先级线程的优先级为 24，低优先级线程的优先级为 25(注意：数值越小则优先级越高)：

```
#define TEST_THREAD_STACK_SIZE 0x1000          //测试线程使用的栈大小,单位为字节
#define TEST_THREAD_PRIO_HI 24                 //高优先级线程的优先级
#define TEST_THREAD_PRIO_LOW 25                //低优先级线程的优先级
```

3) 全局变量定义

设置高优先级任务句柄 task_handle_high 和低优先级任务句柄 task_handle_low，用于存储创建的线程对象；定义 osal_semaphore 类型的全局信号量对象 sem，用于线程间同步。

4) 线程函数定义

static void *test_thread_high(void)：高优先级线程入口函数，首先尝试在 20ms 超

时时间内获取信号量，如果失败则无限期等待获取信号量。

static void * test_thread_low(void)：低优先级线程入口函数，无限期等待获取信号量，获取成功后延时 20ms，然后释放信号量。

5）任务入口函数

（1）初始化信号量：

```
osal_sem_init(&sem, 0);          //初始化信号量,初始值为 0
```

（2）锁定任务调度：

```
osal_kthread_lock();             //锁定任务调度,防止在创建线程时发生上下文切换
```

（3）创建并配置低优先级线程：

```
task_handle_low = osal_kthread_create((osal_kthread_handler)test_thread_low, 0, "test_
thread_low", TEST_THREAD_STACK_SIZE);        //创建低优先级线程
osal_kthread_set_priority(task_handle_low, TEST_THREAD_PRIO_LOW);      //设置线程优先级
                                                                       //为低优先级
osal_printk("test_thread_low create success!\r\n");
```

（4）创建并配置高优先级线程：

```
task_handle_high = osal_kthread_create((osal_kthread_handler)test_thread_high, 0, "test_
thread_high", TEST_THREAD_STACK_SIZE);      //创建高优先级线程
osal_kthread_set_priority(task_handle_high, TEST_THREAD_PRIO_HI);      //设置线程优先级
                                                                       //为高优先级
osal_printk("test_thread_high create success!\r\n");
```

（5）解锁任务调度：

```
osal_kthread_unlock();          //解锁任务调度,允许线程切换
```

6）信号量操作

通过 osal_sem_down_timeout()获取信号量，可以设置超时时间或使用 OSAL_SEM_WAIT _FOREVER 无限期等待；通过 osal_sem_up()释放信号量。

7）线程同步机制

两个线程通过信号量 sem 实现同步。高优先级线程先尝试获取信号量，由于初始值为 0 会失败，然后进入无限期等待状态；低优先级线程获取信号量也会阻塞，但 20ms 后释放信号量，此时高优先级线程会被唤醒并获取信号量。

8）应用启动注册

通过 app_run(TaskEntry)将 TaskEntry 函数注册为应用启动任务，系统启动时会自动执行该函数。

信号量案例流程图如图 3.14 所示。

3. 案例运行现象

信号量运行案例串口信息如图 3.15 所示。

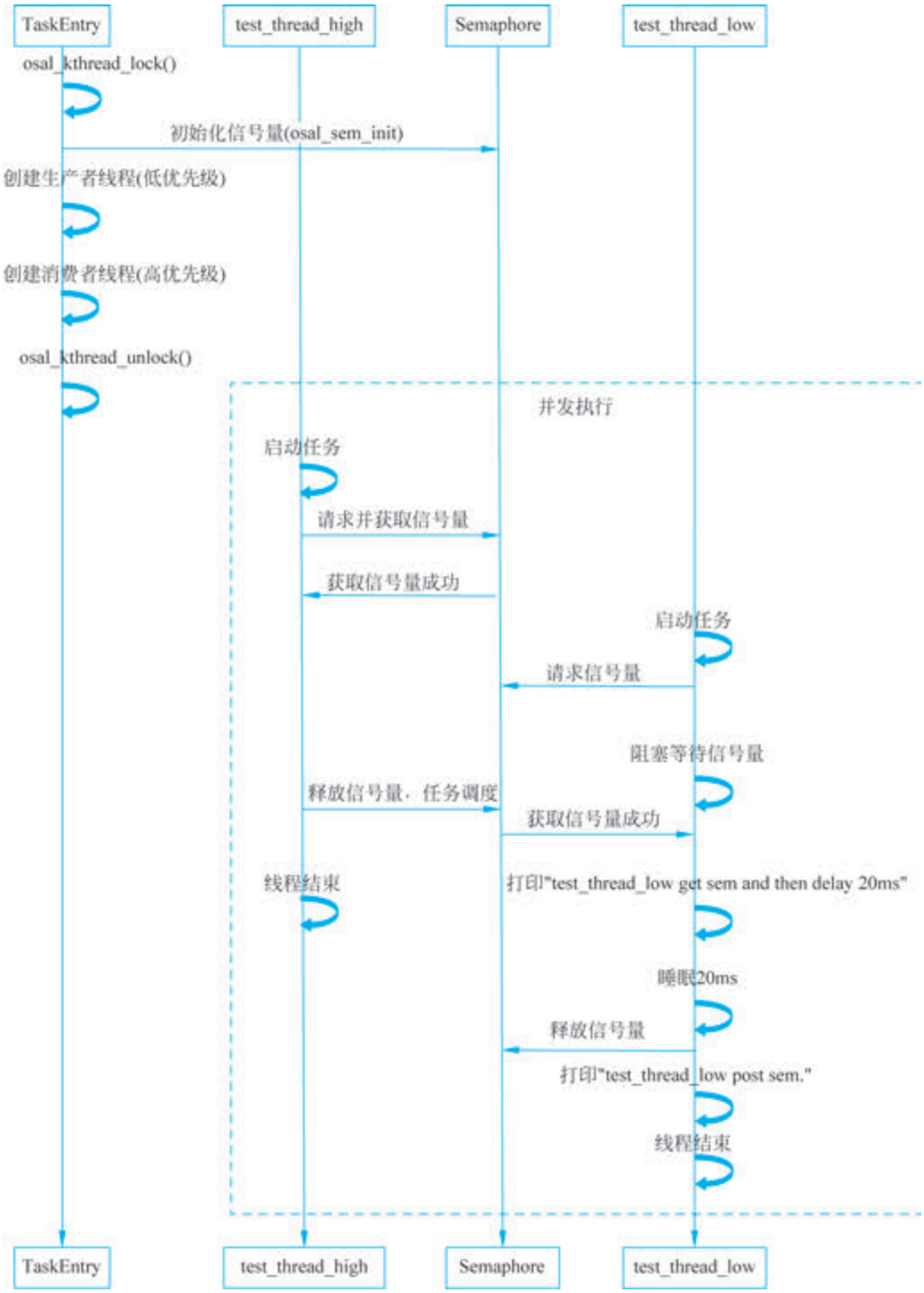

图 3.14　信号量案例流程图

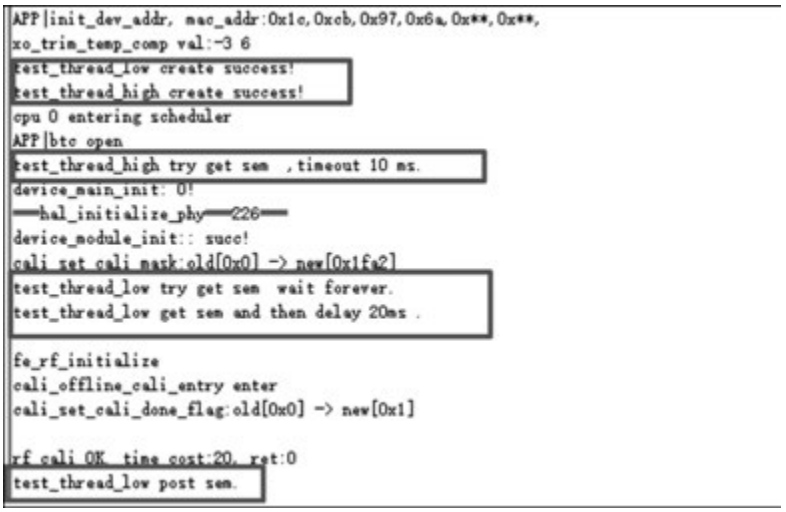

图 3.15　信号量运行案例串口信息

3.7　软件定时器

3.7.1　概述

在嵌入式实时操作系统中，软件定时器（software timer）是一种非常重要的功能组件，用于完成延时、周期性任务触发、超时控制等时间相关操作。

3.7.2　软件定时器基本概念

硬件定时器由硬件直接支持，通常具有较高的精度和稳定性，但数量有限，受限于硬件设计。与硬件定时器不同，软件定时器由操作系统内核管理，基于系统节拍（tick）进行计时，灵活性高，数量可根据需求配置。

软件定时器适用于需要大量定时器的应用场景，如周期性数据采集、超时检测等。需要注意的是，软件定时器不是无限的，按需使用。

由于硬件定时器的数量往往不足以满足实际应用的需求，LiteOS 提供了软件定时器功能，以满足更多的定时器需求。

3.7.3　软件定时器运作机制

软件定时器是 LiteOS 提供的一种重要的系统资源，其底层实现依赖于系统的一个专用队列和一个高优先级任务，用于统一管理所有定时器的触发与回调执行。

在系统内核初始化阶段，LiteOS 会预先分配一块连续内存，用于存储所有软件定时器的控制结构，确保定时器模块的高效运行。

软件定时器的触发遵循先进先出（FIFO）的队列规则，但其内部的排序依据是到期时间。系统将所有激活的定时器按照其到期 tick 时间从小到大组织成一个全局计时

链表(或称为计时队列),这意味着:定时时间较短的定时器总是排在队列头部,优先被检测和处理,定时时间较长的定时器位于队列尾部,确保时间精度与触发顺序的合理性。

软件定时器以系统 tick 为基本计时单位。当用户创建并启动一个软件定时器时,LiteOS 会获取当前系统 tick 计数,结合设定的定时时长,计算出该定时器的到期 tick 时间,将该定时器的控制块插入全局计时链表中,按到期时间排序。

当系统发生 tick 中断时,系统会在 tick 中断处理函数中遍历全局计时链表,检查是否有定时器已到达或超过其到期时间,若发现超时定时器,则将其记录到待处理列表中,但不会立即执行回调函数(避免在中断上下文中执行耗时操作)。tick 中断处理完成后,系统会唤醒一个专用的软件定时器任务,该任务在 LiteOS 中具有最高优先级,确保定时事件能被快速响应,在该任务上下文中,依次调用所有已记录超时定时器的用户注册回调函数。

3.7.4 软件定时器使用案例

1. 相关 API 介绍

软件定时器相关 API 列表如表 3.24～表 3.29 所示。

表 3.24 osal_timer_init()函数接口

定义	int osal_timer_init(osal_timer * timer);
功能	初始化定时器
参数	timer:指向需要初始化的定时器结构体指针(需预先设置 handler、data 和 interval)
返回值	OSAL_SUCCESS(0):成功 OSAL_FAILURE(-1):失败
依赖	kernel/osal/include/time/osal_timer.h

表 3.25 osal_timer_start()函数接口

定义	int osal_timer_start(osal_timer * timer);
功能	启动定时器
参数	timer:指向需要启动的定时器结构体指针
返回值	OSAL_SUCCESS(0):成功 OSAL_FAILURE(-1):失败
依赖	kernel/osal/include/time/osal_timer.h

表 3.26 osal_timer_mod()函数接口

定义	int osal_timer_mod(osal_timer * timer, unsigned int interval);
功能	修改定时器超时时间(如果定时器未激活,则会激活)
参数	timer:指向需要修改的定时器结构体指针 interval:新的超时时间(单位:ms)

续表

返回值	OSAL_SUCCESS(0)：成功　OSAL_FAILURE(－1)：失败
依赖	kernel/osal/include/time/osal_timer.h

表 3.27　osal_timer_stop()函数接口

定义	int osal_timer_stop(osal_timer * timer)；
功能	停止定时器
参数	timer：指向需要停止的定时器结构体指针
返回值	OSAL_SUCCESS(0)：成功　OSAL_FAILURE(－1)：失败
依赖	kernel/osal/include/time/osal_timer.h

表 3.28　osal_timer_destroy()函数接口

定义	int osal_timer_destroy(osal_timer * timer)；
功能	销毁定时器
参数	timer：指向需要停止的定时器结构体指针
返回值	OSAL_SUCCESS(0)：成功　OSAL_FAILURE(－1)：失败
依赖	kernel/osal/include/time/osal_timer.h

表 3.29　osal_gettimeofday()函数接口

定义	void osal_gettimeofday(osal_timeval * tv)；
功能	获取当前系统内核时间
参数	tv：输出参数，存储获取到的系统时间
返回值	无
依赖	kernel/osal/include/time/osal_timer.h

2. 任务完整案例

本案例将使用相关 API 创建两个不同优先级的任务，使用信号量对临界资源进行访问。

1）头文件

头文件需包含"securec.h"、"osal_task.h"、"osal_debug.h"、"app_init.h"、"soc_osal.h"、"timer.h"、"chip_core_irq.h"和"osal_timer.h"，用于安全函数、任务管理、调试输出、应用初始化、操作系统抽象层、定时器硬件接口、中断控制和操作系统定时器功能。

2）定时器参数配置

定义测试线程使用的栈大小为 0x1000 字节，线程优先级为 24：

```
#define TEST_THREAD_STACK_SIZE 0x1000          //测试线程使用的栈大小,单位为字节
```

```
#define TEST_THREAD_PRIO 24                    //线程优先级
```

3）全局变量定义

定义 osal_timer 类型的全局定时器对象 timer1，用于存储创建的定时器实例。

4）定时器操作函数定义

int timer_create(osal_timer * timer, void (* handler)(unsigned long), unsigned long data, unsigned int interval)：创建并启动定时器，初始化定时器参数并启动定时器。

int timer_start(osal_timer * timer,unsigned int interval)：重新启动定时器，修改定时器间隔时间。

int timer_stop(osal_timer * timer)：停止定时器运行。

int timer_destroy(osal_timer * timer)：销毁定时器资源。

5）定时器回调函数

static void timer_callback(unsigned long data)：定时器超时回调函数，打印定时器触发信息，并重新设置定时器间隔为 2000ms。

6）线程函数定义

static void * test_thread(void)：测试线程入口函数，负责创建定时器、停止定时器，然后修改定时器间隔。

7）任务入口函数

（1）锁定任务调度：

```
osal_kthread_lock();        //锁定任务调度,防止在创建线程时发生上下文切换
```

（2）创建测试线程：

```
task_handle = osal_kthread_create((osal_kthread_handler)test_thread, 0, "test_thread",
TEST_THREAD_STACK_SIZE);      //创建测试线程
osal_kthread_set_priority(task_handle, TEST_THREAD_PRIO - 1);      //设置线程优先级
```

（3）解锁任务调度：

```
osal_kthread_unlock();          //解锁任务调度,允许线程切换
```

8）定时器操作流程

使用 osal_timer_start()创建并启动定时器，设置初始间隔为 1000ms；使用 osal_timer_stop()停止定时器运行；使用 osal_timer_mod()修改定时器间隔为 2000ms。

9）应用启动注册

通过 app_run(TaskEntry)将 TaskEntry 函数注册为应用启动任务，系统启动时会自动执行该函数。

软件定时器案例流程图如图 3.16 所示。

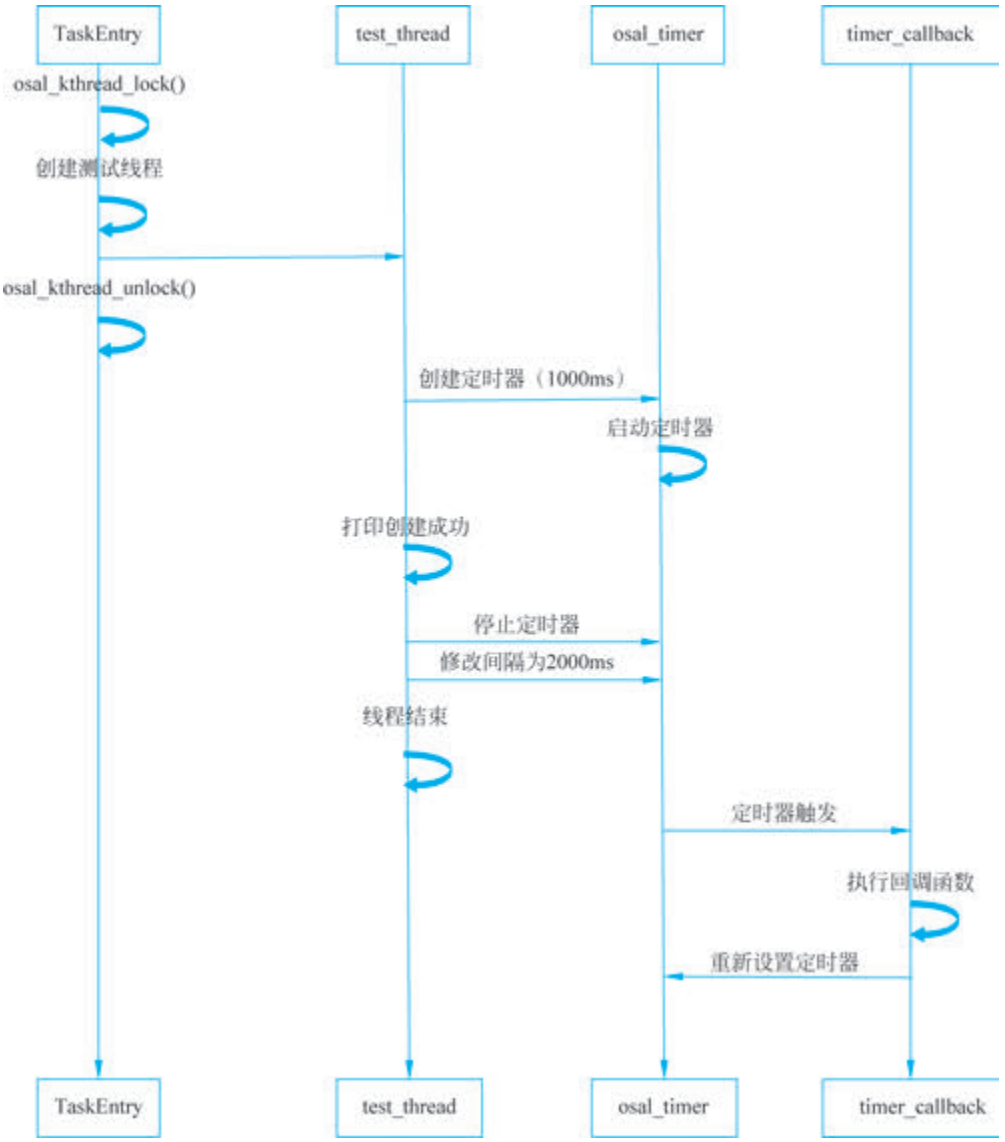

图 3.16　软件定时器案例流程图

3.8　XTS 认证

3.8.1　XTS 认证简介

开源鸿蒙 XTS(X Test Suite)认证是 OpenHarmony 生态的“兼容性护照”。它通过 acts/dcts 等自动化测试套件，对设备或应用的系统功能、性能、安全、接口一致性进行 4 级用例、3 种粒度的严格验证，确保同一套代码在上海海思、瑞芯微等多芯片平台上都能稳定运行、统一体验。通过 XTS 认证，即表明该产品已获得官方认可，可无缝接入开源

鸿蒙生态，实现跨品类互联互通，成为分布式场景规模化落地的先决条件。

开源鸿蒙兼容性测评服务主要支持轻量系统（参考内存≥128KB）、小型系统（参考内存≥1MB）和标准系统（参考内存≥128MB），支持的版本有 v4.1Release（正式发布版）、v5.1Release 等。

本节的 XTS 认证是基于轻量系统开发板进行的 v5.1Release 版本的认证。

3.8.2 XTS 认证的基本流程

1. 概述

企业执行测评的步骤如下。

（1）申请 OpenHarmony 兼容性测评的企业（以下简称“申请方”）在开放原子开源基金会网站申请企业账号。

（2）申请方从 Gitee 平台获取代码进行适配开发；从 OpenHarmony 官网兼容性 XTS 专区获取兼容性测试套件并在本地进行测试，测试完成后，申请方可获取测试报告；从 OpenHarmony 官网兼容性 PCS 专区获取 PCS 自检表并填写 PCS 自检表；如需申请失败项豁免，请前往 OpenHarmony 兼容性平台进行豁免申请，获取豁免结果；兼容性测试与 PCS 自检也可委托兼容性工作组授权的兼容性测评合作中心进行。

（3）申请方首次申请测试报告评审时，应签署 OpenHarmony 兼容性平台所示《OpenHarmony 兼容性协议》及《OpenHarmony 兼容性平台隐私声明》；申请方上传测试报告、PCS 自检表和镜像到 OpenHarmony 兼容性平台，申请方还应在上传测试报告的同时向 OpenHarmony 兼容性工作组寄送产品样品。

（4）OpenHarmony 兼容性工作组收到申请方上传的测试报告和产品样品后进行测评，并给出测评结果。若测评通过，则进入步骤（5）；若测评不通过，OpenHarmony 兼容性工作组将通知申请方进行整改。

（5）测评通过后，OpenHarmony 兼容性工作组将按需启动复测流程。若未被选中复测，则申请方通过本次 OpenHarmony 兼容性测评。若被选中复测，则复测所用的兼容性测试套件包将由 OpenHarmony 兼容性工作组上传至平台，申请方在 OpenHarmony 兼容性平台上下载前述复测套件包并在本地执行，生成复测报告后上传到 OpenHarmony 兼容性平台。

（6）OpenHarmony 兼容性工作组对申请方复测报告进行评审，若复测评审通过，则本次 OpenHarmony 兼容性测评通过；若复测评审不通过，OpenHarmony 兼容性工作组将通知申请方整改。

（7）OpenHarmony 兼容性测评通过后，开放原子开源基金会将发放证书，在 OpenHarmony 官网进行展示，并授权申请方在其设备类 OpenHarmony 兼容产品及其包装、营销材料上使用 OpenHarmony 兼容性标识。

备注：如需反馈问题请通过 OpenHarmony 兼容性平台提交问题详情。

执行测评流程的步骤（1），注册企业账号后，步骤（2）、步骤（3）需要在兼容性平台进行。兼容性平台的企业账号页面如图 3.17 所示。

图 3.17　兼容性平台的企业账号页面

XTS 认证流程如图 3.18 所示。

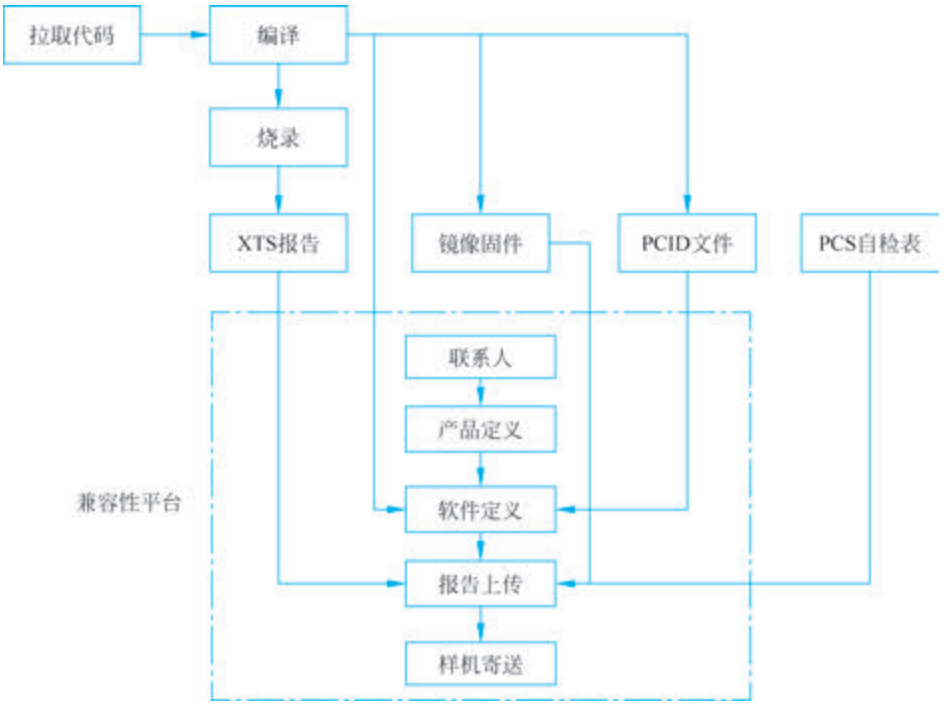

图 3.18　XTS 认证流程图

2. 源码编译与烧录

1）拉取源码并配置环境

在新终端中输入，执行如下命令安装码云 repo 工具，这些命令中的安装路径以"～/bin"为例，请用户自行创建所需目录。

（1）按顺序执行：

```
mkdir ～/bin
```

```
curl https://gitee.com/oschina/repo/raw/fork_flow/repo-py3 -o ~/bin/repo
chmod a+x ~/bin/repo
pip3 install -i https://repo.huaweicloud.com/repository/pypi/simple requests
```

(2) 安装 git-lfs：

```
sudo apt-get install git-lfs
```

(3) 将 repo 添加到环境变量：

```
echo 'export PATH=~/bin:$PATH'>> ~/.bashrc
source ~/.bashrc
```

(4) 通过 repo + https 下载，需要自己建立文件夹目录，否则会全部在根目录：

```
mkdir ws63_ohos
cd ws63_ohos
repo init -u https://gitee.com/HiSpark/manifest.git -b OpenHarmony-v4.1.0-Release -m
default-ws63-xts.xml --no-repo-verify
repo sync -c
repo forall -c 'git lfs pull
```

(5) 下载编译工具链：

```
./build/prebuilts_download.sh
```

(6) 安装 hb：

```
python -m pip install build/hb
```

(7) 添加环境变量：

```
PATH=/root/ws63_ohos/device/soc/hisilicon/ws63v100/sdk/tools/bin/compiler/riscv/cc_
riscv32_musl_100/cc_riscv32_musl/bin:$PATH
source ~/.bashrc
```

2) XTS 代码编译

(1) 在根目录下执行以下指令，选择编译选项：

```
hb set
```

(2) 选择轻量系统编译：

```
mini
```

(3) 选择编译产品类型：

```
nearlink_dk_3863_xts
```

(4) 选择完成后，输入以下指令执行编译：

```
hb build -f -b debug --gn-args build_xts=true
```

(5) 等待编译结束即可。

3) 烧录并获得 XTS 报告

(1) 下载 BurnTool 烧录工具并解压。

(2) 打开烧录工具，选择对应的串口，打开烧录工具，点开 Option 选项，选择对应的目标即可。

（3）选择烧录文件，示例路径为（选 ws63-liteos-app-all.fwpkg）“{代码根目录}/out/nearlink_dk_3863/nearlink_dk_3863_xts/ws63-liteos-xts/ws63-liteos-xts_all.fwpkg”。

（4）切换到 APP 模式，使用 AT 发送“AT＋FTM＝0”指令即可。可以看到如图 3.19 所示的运行结果，当开始执行自动兼容性测试时代表进入测试模式，等待结束后将输出日志保存即可（下一部分的 XTS 报告提交将会使用）。兼容性测评用例运行过程信息如图 3.19 所示。

```
01-01 00:00:00.440 0 108 I 1/SAMGR: Init service:0x3f46c0

01-01 00:00:00.440 0 108 I 1/SAMGR: Init service:0x3f5360

01-01 00:00:00.440 0 108 I 1/SAMGR: Init service:0x3f5544

01-01 00:00:00.440 0 108 I 1/SAMGR: Init service:0x3f56a8


[00:28:05.186]收←◆Run test suite 1 times
[TestStartBootstrapSamgr001:0]../../../test/xts/acts/startup_lite/bootstrap_hal/src/samgr_api_test.c:342:TestStartBootstrapSa
mgr001:PASS
g_coreInit0: 01: 0default: 03: 04: 05:
0../../../test/xts/acts/startup_lite/bootstrap_hal/src/samgr_api_test.c:353:TestStartBootstrapSamgr002:PASS
g_sysServiceInit0: 01: 0default: 03: 04: 05:
0../../../test/xts/acts/startup_lite/bootstrap_hal/src/samgr_api_test.c:365:TestStartBootstrapSamgr003:PASS
g_sysFeatureInit0: 01: 0default: 03: 04: 05:
0../../../test/xts/acts/startup_lite/bootstrap_hal/src/samgr_api_test.c:377:TestStartBootstrapSamgr004:PASS
g_sysExServiceInit0: 01: 0default: 03: 04: 05: 0g_appServiceInit0: 01: 0default: 03: 04: 05:
0../../../test/xts/acts/startup_lite/bootstrap_hal/src/samgr_api_test.c:389:TestStartBootstrapSamgr005:PASS
g_sysExFeatureInit0: 01: 0default: 03: 04: 05: 0g_appFeatureInit0: 01: 0default: 03: 04: 05:
0../../../test/xts/acts/startup_lite/bootstrap_hal/src/samgr_api_test.c:403:TestStartBootstrapSamgr006:PASS
g_sysRun0: 01: 0default: 03: 04: 05:
0../../../test/xts/acts/startup_lite/bootstrap_hal/src/samgr_api_test.c:417:TestStartBootstrapSamgr007:PASS
+-----------------------------------------+


7 Tests 0 Failures 0 Ignored
OK
Start to run test suite:Broadcast02TestSuite
```

图 3.19　兼容性测评用例运行过程

3. 兼容性平台申请流程

1）第一部分：联系人

联系人填写界面如图 3.20 所示，该部分按要求填写即可。

图 3.20　联系人填写界面

2）第二部分：产品定义

产品定义填写界面如图 3.21 所示。

图 3.21　产品定义填写界面

（1）测评类型：请在“模组/开发板”、“商用设备”和“发行版”三者中勾选其一；若勾选商用设备，需补充设备名称、品牌等字段，并在后面的“软件定义”中完成授权验证。以下流程均以商用设备为例。

（2）操作系统类型：按硬件性能在“标准系统”、“小型系统”和“轻量系统”三者中勾选其一；标准系统在 XTS 测试阶段用例更多。以下流程均以标准系统为例。

（3）操作系统版本号：填写本次构建的 OpenHarmony 版本号。

（4）设备类型：在多级下拉列表中选最贴合的一项。

（5）设备名称（传播名）/设备型号/品牌/品牌英文名：按厂商资料如实填写，其中“中文名称”与“型号”将出现在证书中。

（6）硬件设备版本号/模组型号/芯片型号：按实际物料填写。

（7）基本信息描述：简述设备核心用途。

（8）外观图：按规格上传纯白背景、指定尺寸的图片。

（9）公示选项：按需选择“发证即公示”、“指定最早公示日期”或“不公示”。

3）第三部分：软件定义

软件定义填写界面如图 3.22 所示。

（1）软件版本号：按厂商提供的实际版本填写。

（2）安全补丁标签：完成对应 XTS 测试后，可填写该补丁日期。

（3）版本 id：按需填写即可。

（4）版本 Hash：保持默认即可。

（5）PCID：在编译产物的 out 目录下提取 PCID.sc 文件并上传。

4）第四部分：报告上传

报告上传填写界面如图 3.23 所示。报告上传主要分为三个文件：镜像文件、XTS 报告和 PCS 自检表。

图 3.22　软件定义填写界面

图 3.23　报告上传填写界面

（1）镜像文件：将镜像文件即编译出的固件包，压缩在兼容性平台的“企业空间”界面上传后，在此处下拉框选择即可。

（2）XTS 报告：主要为烧录进 XTS 固件的开发板/设备的打印 log。

（3）PCS 自检表：下载对应的 OH 系统版本、对应系统类型的 PCS 自检表，按照 PCS 规范文件进行填写后上传。

5）第五部分：样机寄送

样机寄送填写界面如图 3.24 所示。

（1）快递单号 1：填写测评编号（每项申请对应一个）即可。

（2）添加调测设备：完成前文的“授权验证”工作后，选择相应的设备。

（3）快递物品清单：填写邮寄的设备等，若设备较大，可仅邮寄开发套件与屏幕（若带屏）。注意：“模组/开发板”、“商用设备”和“发行版”所需调测设备数量不同。

图 3.24　样机寄送填写界面

3.8.3　注意事项

（1）如果出现用例失败，建议先分析是否为用例本身问题，部分分支可能未同步完全，建议更新代码再进一步查看。

（2）注意第一次要切换为 APP 模式，即使用 AT 发送“AT+FTM=0”指令。

（3）在“兼容性平台申请流程”中的软件定义部分，注意版本 id 的填写，版本 id 由共计 10 个字段拼接而成，即“devicetype/manufacture/brand/productSeries/OSFullName/productModel/SoftwareModel/OHOS _ SDK _ API _ VERSION/incrementalVersion/buildType”。具体填写内容如下。

① devicetype：根据实际设备选择，可选“default”“phone”“tablet”“tv”等选项，可查看开源鸿蒙 OS 应用开发 3.1 部分。

② manufacture：填写厂商英文名。注意：需要与兼容性平台-用户管理-账户管理中的“企业简称(英文)”保持一致。

③ brand：填写设备品牌。注意：需要与兼容性测评-软件定义中的“品牌英文名”保持一致。

④ productSeries：填写产品系列名称，无特殊要求。

⑤ OSFullName：使用 USB 连接 OH 开发板和 PC，在 hdc shell 输入 begetctl dump api 指令，查看 OSFullName 的值即可。例如：OpenHarmony-4.0.10.13。

⑥ productModel：填写设备品牌。注意：需要与兼容性测评-软件定义中的“品牌英文名”保持一致。

⑦ SoftwareModel：用户可见的软件版本号，无特殊要求。

⑧ OHOS_SDK_API_VERSION：系统软件 API Version，根据 OpenHarmony 系统的版本查询对应的 API 版本。例如：OpenHarmonyV4.0 对应 API 10，则此处填写“10”。

⑨ incrementalVersion：差异版本号，无特殊要求。

⑩ buildType：构建类型。可选 debug、release、log、nolog 等。