使用指南

通用 MCU RT_Thread 使用指南

简介

本文档主要介绍 RT_Thread 系统在国民技术通用 MCU 中的使用，适用于 N32G45x、N32G4FR、N32WB452、N32G43x、N32L40x、N32L43x 系列芯片。本文档以 N32G45x 为例介绍 RT_Thread 系统相关使用说明。

目录

国民技术股份有限公司 Nations Technologies Inc.
地址：深圳市南山区高新北区宝深路 109 号国民技术大厦
电话：+86-755-86309900 传真：+86-755-86169100
邮箱：info@nationstech.com 邮编：518057

1 RT_Thread

1.1 概述

RT-Thread，全称是 Real Time-Thread，顾名思义，它是一个嵌入式实时多线程操作系统，基本属性之一是支持多任务，允许多个任务同时运行并不意味着处理器在同一时刻真地执行了多个任务。事实上，一个处理器核心在某一时刻只能运行一个任务，由于每次对一个任务的执行时间很短、任务与任务之间通过任务调度器进行非常快速地切换（调度器根据优先级决定此刻该执行的任务），给人造成多个任务在一个时刻同时运行的错觉。在 RT-Thread 系统中，任务通过线程实现的，RT-Thread 中的线程调度器也就是以上提到的任务调度器。

RT-Thread 主要采用 C 语言编写，浅显易懂，方便移植。它把面向对象的设计方法应用到实时系统设计中，使得代码风格优雅、架构清晰、系统模块化并且可裁剪性非常好。针对资源受限的微控制器（MCU）系统，可通过方便易用的工具，裁剪出仅需要 3KB Flash、1.2KB RAM 内存资源的 NANO 版本（NANO 是 RT-Thread 官方于 2017 年 7 月份发布的一个极简版内核)；而对于资源丰富的物联网设备，RT-Thread 又能使用在线的软件包管理工具，配合系统配置工具实现直观快速的模块化裁剪，无缝地导入丰富的软件功能包，实现类似 Android 的图形界面及触摸滑动效果、智能语音交互效果等复杂功能。

相较于 Linux 操作系统，RT-Thread 体积小，成本低，功耗低、启动快速，除此以外 RT-Thread 还具有实时性高、占用资源小等特点，非常适用于各种资源受限（如成本、功耗限制等）的场合。虽然 32 位 MCU 是它的主要运行平台，实际上很多带有 MMU、基于 ARM9、ARM11 甚至 Cortex-A 系列级别 CPU 的应用处理器在特定应用场合也适合使用 RT-Thread。

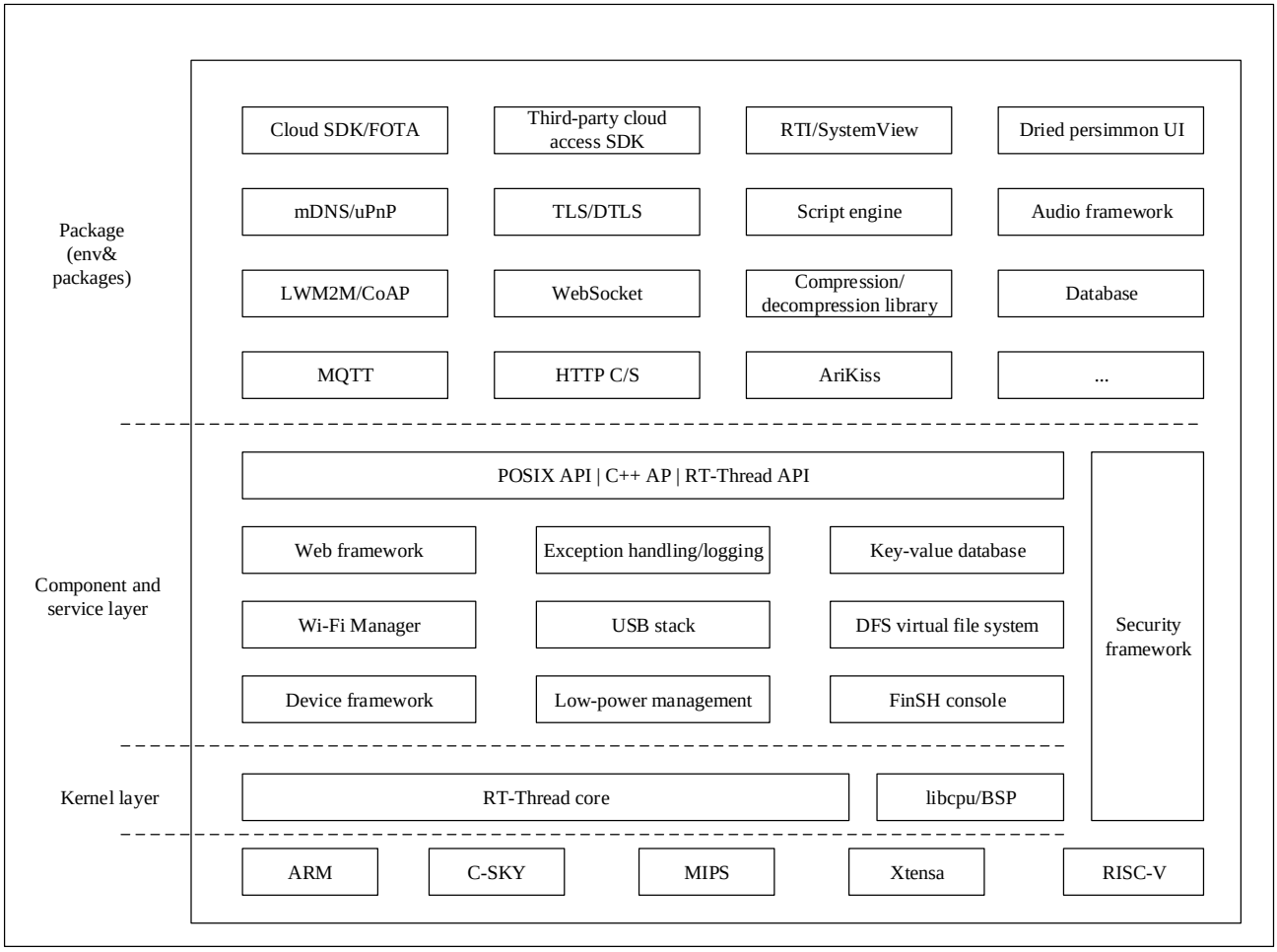
1.2 RT-Thread 架构

近年来，物联网（Internet Of Things，IoT）概念广为普及，物联网市场发展迅猛，嵌入式设备的联网已是大势所趋。终端联网使得软件复杂性大幅增加，传统的 RTOS 内核已经越来越难满足市场的需求，在这种情况下，物联网操作系统（IoT OS）的概念应运而生。物联网操作系统是指以操作系统内核（可以是 RTOS、Linux 等）为基础，包括如文件系统、图形库等较为完整的中间件组件，具备低功耗、安全、通信协议支持和云端连接能力的软件平台，RT-Thread 就是一个 IoT OS。

RT-Thread 与其他很多 RTOS 如 FreeRTOS、uC/OS 的主要区别之一是，它不仅仅是一个实时内核，还具备丰富的中间层组件，如图 1-1 所示。

国民技术股份有限公司 Nations Technologies Inc.
地址：深圳市南山区高新北区宝深路 109 号国民技术大厦
电话：+86-755-86309900 传真：+86-755-86169100
邮箱：info@nationstech.com 邮编：518057

图 1-1 RT_Thread 软件框架图



它具体包括以下部分:

- 内核层：RT-Thread 内核，是 RT-Thread 的核心部分，包括了内核系统中对象的实现，例如多线程及其调度、信号量、邮箱、消息队列、内存管理、定时器等；libcpu/BSP（芯片移植相关文件/板级支持包）与硬件密切相关，由外设驱动和 CPU 移植构成。
- 组件与服务层：组件是基于 RT-Thread 内核之上的上层软件，例如虚拟文件系统、FinSH 命令行界面、网络框架、设备框架等。采用模块化设计，做到组件内部高内聚，组件之间低耦合。
- RT-Thread 软件包：运行于 RT-Thread 物联网操作系统平台上，面向不同应用领域的通用软件组件，由描述信息、源代码或库文件组成。RT-Thread 提供了开放的软件包平台，这里存放了官方提供或开发者提供的软件包，该平台为开发者提供了众多可重用软件包的选择，这也是 RT-Thread 生态的重要组成部分。软件包生态对于一个操作系统的选择至关重要，因为这些软件包具有很强的可重用性，模块化程度很高，极大的方便应用开发者在最短时间内，打造出自己想要的系统。RT-Thread 已经支持的软件包数量已经达到 60+，如下举例：
  - 物联网相关的软件包：Paho MQTT、WebClient、mongoose、WebTerminal 等等
  - 脚本语言相关的软件包：目前支持 JerryScript、MicroPython
  - 多媒体相关的软件包：Openmv、mupdf

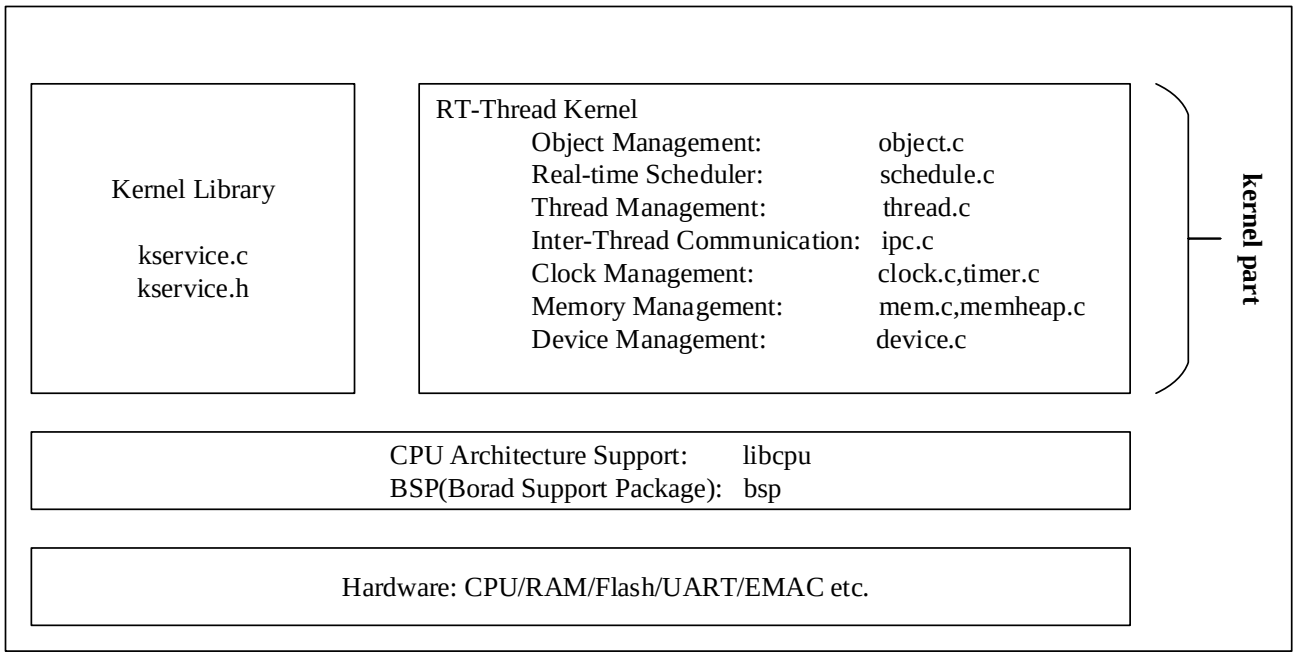
国民技术股份有限公司 Nations Technologies Inc.
地址：深圳市南山区高新北区宝深路 109 号国民技术大厦
电话：+86-755-86309900 传真：+86-755-86169100
邮箱：info@nationstech.com 邮编：518057

- 工具类软件包：CmBacktrace、EasyFlash、EasyLogger、SystemView
- 系统相关的软件包：RTGUI、Persimmon UI、lwext4、partition、SQLite 等等
- 外设库与驱动类软件包：RealTek RTL8710BN SDK

1.3 RT_Thread 内核

内核是操作系统最基础也是最重要的部分。图 1-2 为 RT-Thread 内核架构图，内核处于硬件层之上，内核部分包括内核库、实时内核实现。

图 1-2 RT_Thread 内核及底层结构



内核库是为了保证内核能够独立运行的一套小型的类似 C 库的函数实现子集。它提供了类似“strcpy”、“memcpy”、“printf”、“scanf”等函数的实现。RT-Thread 内核库仅提供内核用到的一小部分 C 库函数实现，为了避免与标准 C 库重名，在这些函数前都会添加上 rt_前缀。

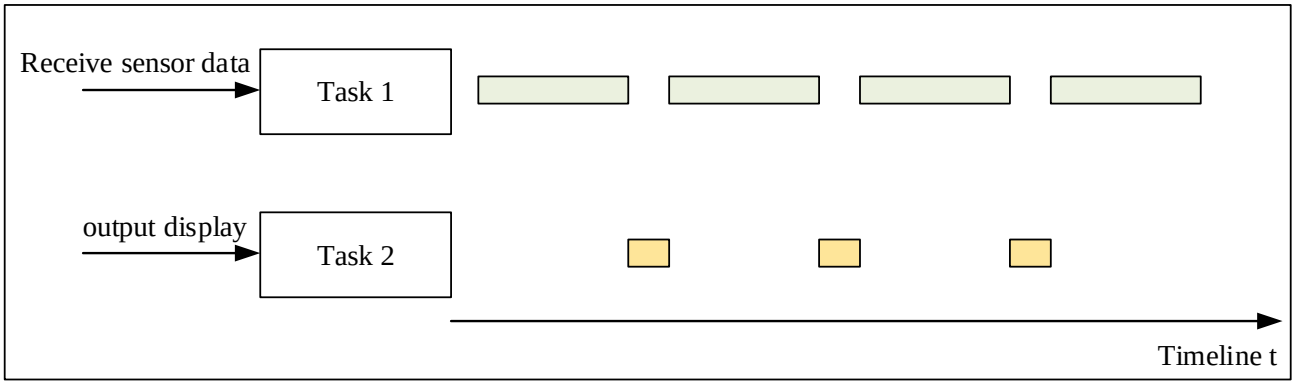
实时内核的实现包括：对象管理、线程管理及调度器、线程间通信管理、时钟管理及内存管理等等，内核最小的资源占用情况是 3KB ROM，1.2KB RAM。

1.4 RT_Thread 线程管理

在日常生活中，我们要完成一个大任务，一般会将它分解成多个简单、容易解决的小问题，小问题逐个被解决，大问题也就随之解决了。在多线程操作系统中，也同样需要开发人员把一个复杂的应用分解成多个小的、可调度的、序列化的程序单元，当合理地划分任务并正确地执行时，这种设计能够让系统满足实时系统的性能及时间的要求，例如让嵌入式系统执行这样的任务，系统通过传感器采集数据，并通过显示屏将数据显示出来，在多线程实时系统中，可以将这个任务分解成两个子任务，如图 1-3 所示，一个子任务不间断地读取传感器数据，并将数据写到共享内存中，另外一个子任务周期性的从共享内存中读取数据，并将传感器数据输出到显示屏上。

国民技术股份有限公司 Nations Technologies Inc.
地址：深圳市南山区高新北区宝深路 109 号国民技术大厦
电话：+86-755-86309900 传真：+86-755-86169100
邮箱：info@nationstech.com 邮编：518057

图 1-3 传感器数据接收任务与显示任务的切换执行



在 RT-Thread 中，与上述子任务对应的程序实体就是线程，线程是实现任务的载体，它是 RT-Thread 中最基本的调度单位，它描述了一个任务执行的运行环境，也描述了这个任务所处的优先等级，重要的任务可设置相对较高的优先级，非重要的任务可以设置较低的优先级，不同的任务还可以设置相同的优先级，轮流运行。

当线程运行时，它会认为自己是以独占 CPU 的方式在运行，线程执行时的运行环境称为上下文，具体来说就是各个变量和数据，包括所有的寄存器变量、堆栈、内存信息等。

1.5 RT_Thread 时钟管理

RT-Thread 的时钟管理以时钟节拍为基础，时钟节拍指的是周期性硬件定时器两次中断间的间隔时间长度，这个周期性硬件定时器称之为系统时钟。时钟节拍（OS Tick）是 RT-Thread 操作系统中最小的时钟单位，系统节拍一般定义为 32 位无符号整数，提供给应用程序所有和时间有关的服务，如线程的延时、线程的时间片轮转调度以及定时器超时等，从系统启动开始计数的时钟节拍数称为系统时间。时钟节拍来源于定时器的周期性中断，一次中断表示一个 OS Tick。OS Tick 的长度可以根据 RT_TICK_PER_SECOND 的定义来调整，等于 1/RT_TICK_PER_SECOND 秒，精度越高的时钟将导致系统中定时器频繁检查。

1.6 RT_Thread 中断管理

RT-Thread 的中断管理功能主要是管理中断设备、中断服务例程、中断嵌套、中断栈的维护、线程切换时的现场保护和恢复等。

当 CPU 正在处理内部数据时，外界发生了紧急情况，要求 CPU 暂停当前的工作转去处理这个异步事件。处理完毕后，再回到原来被中断的地址，继续原来的工作，这样的过程称为中断。实现这一功能的系统称为中断系统，申请 CPU 中断的请求源称为中断源，当多个中断源同时向 CPU 请求中断时，就存在一个中断优先权的问题。通常根据中断源的优先级别，优先处理最紧急事件的中断请求源，即最先响应级别最高的中断请求。

当中断产生时，CPU 将按如下的顺序执行：

1) 保存当前处理机状态信息

2) 载入异常或中断处理函数到 PC 寄存器

3) 把控制权转交给处理函数并开始执行

国民技术股份有限公司 Nations Technologies Inc.
地址：深圳市南山区高新北区宝深路 109 号国民技术大厦
电话：+86-755-86309900 传真：+86-755-86169100
邮箱：info@nationstech.com 邮编：518057

4) 当处理函数执行完成时，恢复处理器状态信息

5) 从异常或中断中返回到前一个程序执行点

中断使得 CPU 可以在事件发生时才进行处理，而不必让 CPU 连续不断地查询是否有相应的事件发生。

1.7 RT_Thread 内存管理

静态内存池接口：内存池（Memory Pool）是一种内存分配方式，用于分配大量大小相同的小内存块。它可以极大地加快内存分配与释放的速度，且能尽量避免内存碎片化。当内存池为空时，可以阻止分配的线程（或者立即返回，或者等待一段时间返回，这由 timeout 参数确定）。当其他线程向此内存池释放内存块时，被阻塞的线程将被唤醒。

动态内存堆接口：动态内存管理是一个真实的堆内存管理模块，可以在当前资源满足的情况下，根据用户的需求分配任意大小的内存块。而当用户不需要再使用这些内存块时，又可以释放回堆中供其他应用分配使用。RT-Thread 系统为了满足不同的需求，提供了两套不同的动态内存管理算法，分别是小堆内存管理算法和 SLAB 内存管理算法。

- 小堆内存管理模块主要针对系统资源比较少，一般用于小于 2MB 内存空间的系统
- SLAB 内存管理模块则主要是在系统资源比较丰富时，提供了一种近似多内存池管理算法的快速算法

两种内存管理模块在系统运行时只能选择其中之一或者完全不使用动态堆内存管理器。这两种管理模块提供的 API 接口完全相同。

除上述之外，RT-Thread 还有一种针对多内存堆的管理机制，即 memheap 内存管理。memheap 方法适用于系统存在多个内存堆的情况，它可以将多个内存“粘贴”在一起，形成一个大的内存堆，用户使用起来会感到格外便捷。

国民技术股份有限公司 Nations Technologies Inc.
地址：深圳市南山区高新北区宝深路 109 号国民技术大厦
电话：+86-755-86309900 传真：+86-755-86169100
邮箱：info@nationstech.com 邮编：518057

2 RT_Thread 应用

2.1 线程创建示例

使用 RTOS 的实时应用程序可以构造为一组独立的线程。每个线程都在其自己的上下文中执行，而不会偶然依赖系统中的其他线程或 RTOS 调度程序本身。在任何时间点，应用程序中只有一个线程可以执行，而 RTOS 调度程序负责确定该线程应为哪个线程。

以下是有关线程创建的示例。

led0_thread: this thread toggles LED0 every 500 ms
Create thread:
/* led0_thread definition */ rt_thread_init(&led0_thread, "led0", led0_thread_entry, RT_NULL, (rt_uint8_t*)&led0_stack[0], sizeof(led0_stack), 3, 5); /* Start led0_thread*/ rt_thread_startup(&led0_thread);

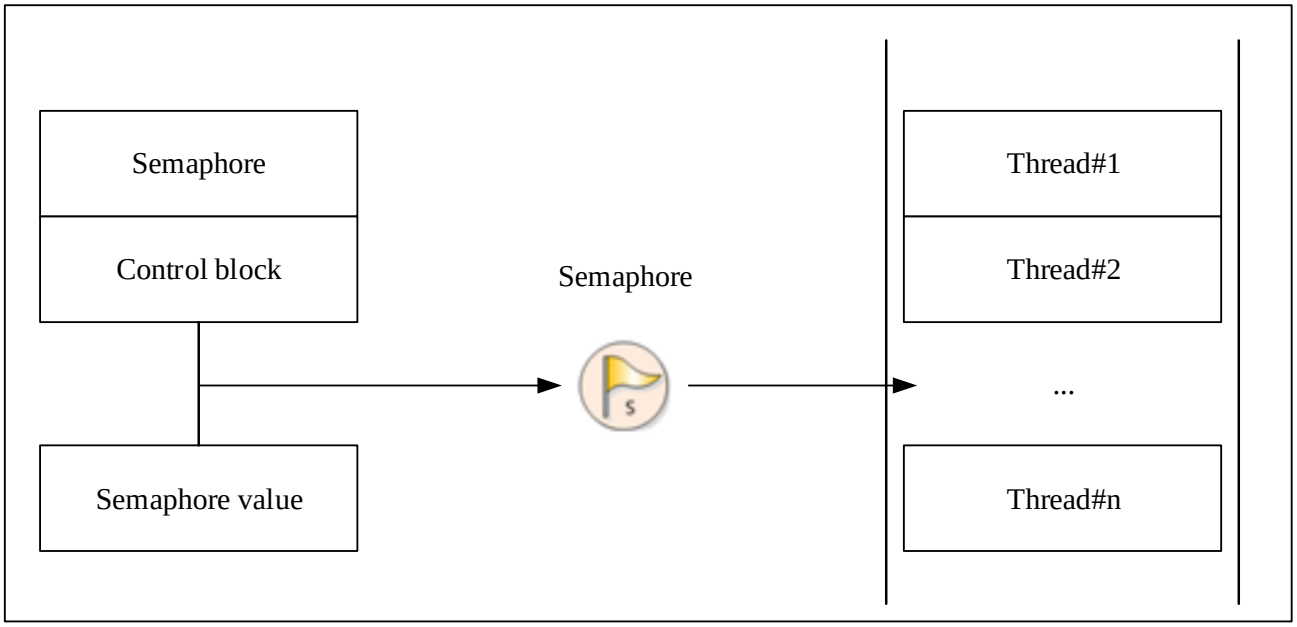
2.2 信号量示例

信号量是一种轻型的用于解决线程间同步问题的内核对象，线程可以获取或释放它，从而达到同步或互斥的目的。

信号量工作示意图如图 2-1 所示，每个信号量对象都有一个信号量值和一个线程等待队列，信号量的值对应了信号量对象的实例数目、资源数目，假如信号量值为 5，则表示共有 5 个信号量实例（资源）可以被使用，当信号量实例数目为零时，再申请该信号量的线程就会被挂起在该信号量的等待队列上，等待可用的信号量实例（资源）。

国民技术股份有限公司 Nations Technologies Inc.
地址：深圳市南山区高新北区宝深路 109 号国民技术大厦
电话：+86-755-86309900 传真：+86-755-86169100
邮箱：info@nationstech.com 邮编：518057

图 2-1 信号量工作示意图



Semaphore:

```
/* Create the binary semaphore */
rt_sem_init(&key_sem,
            "keysem",
            0,
            RT_IPC_FLAG_FIFO);

/* Get the semaphore*/
rt_sem_take(&key_sem,
            RT_WAITING_FOREVER);

/* Relase the semaphore*/
rt_sem_release(&key_sem);
```

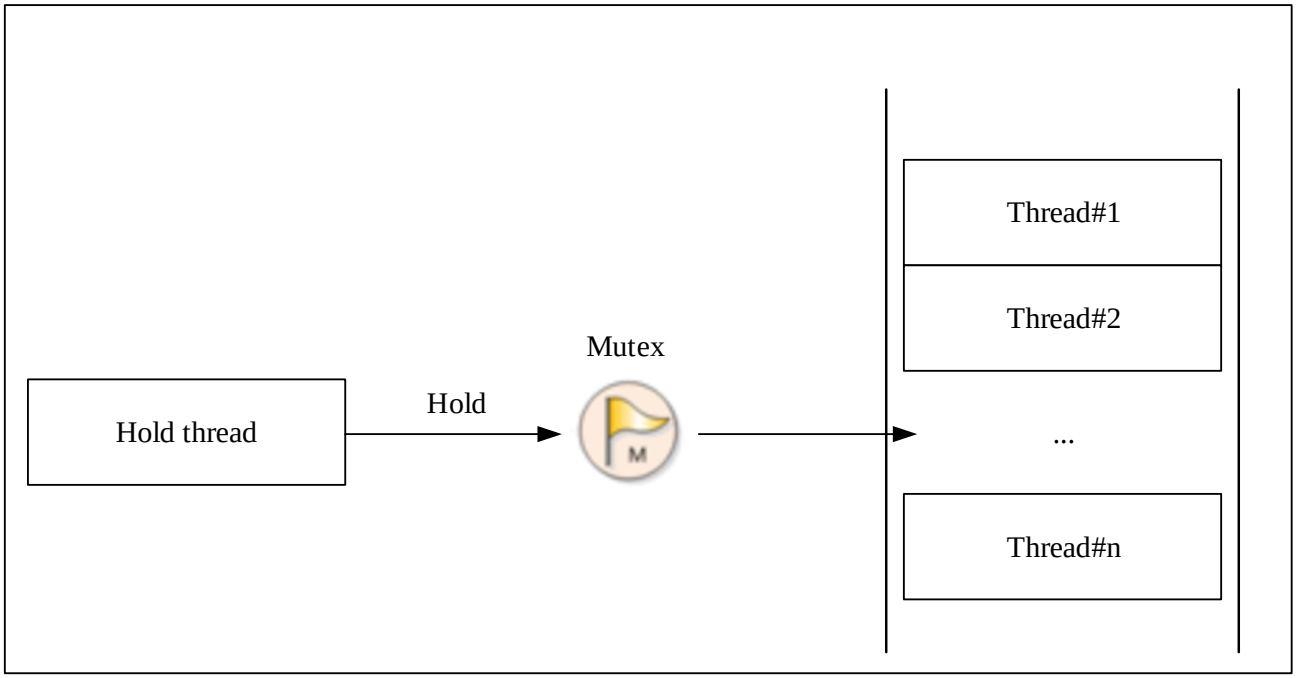
2.3 互斥量示例

互斥量和信号量不同的是：拥有互斥量的线程拥有互斥量的所有权，互斥量支持递归访问且能防止线程优先级翻转；并且互斥量只能由持有线程释放，而信号量则可以由任何线程释放。

互斥量的状态只有两种，开锁或闭锁（两种状态值）。当有线程持有它时，互斥量处于闭锁状态，由这个线程获得它的所有权。相反，当这个线程释放它时，将对互斥量进行开锁，失去它的所有权。当一个线程持有互斥量时，其他线程将不能够对它进行开锁或持有它，持有该互斥量的线程也能够再次获得这个锁而不被挂起，如图 2-2 所示。这个特性与一般的二值信号量有很大的不同：在信号量中，因为已经不存在实例，线程递归持有会发生主动挂起（最终形成死锁）。

国民技术股份有限公司 Nations Technologies Inc.
地址：深圳市南山区高新北区宝深路 109 号国民技术大厦
电话：+86-755-86309900 传真：+86-755-86169100
邮箱：info@nationstech.com 邮编：518057

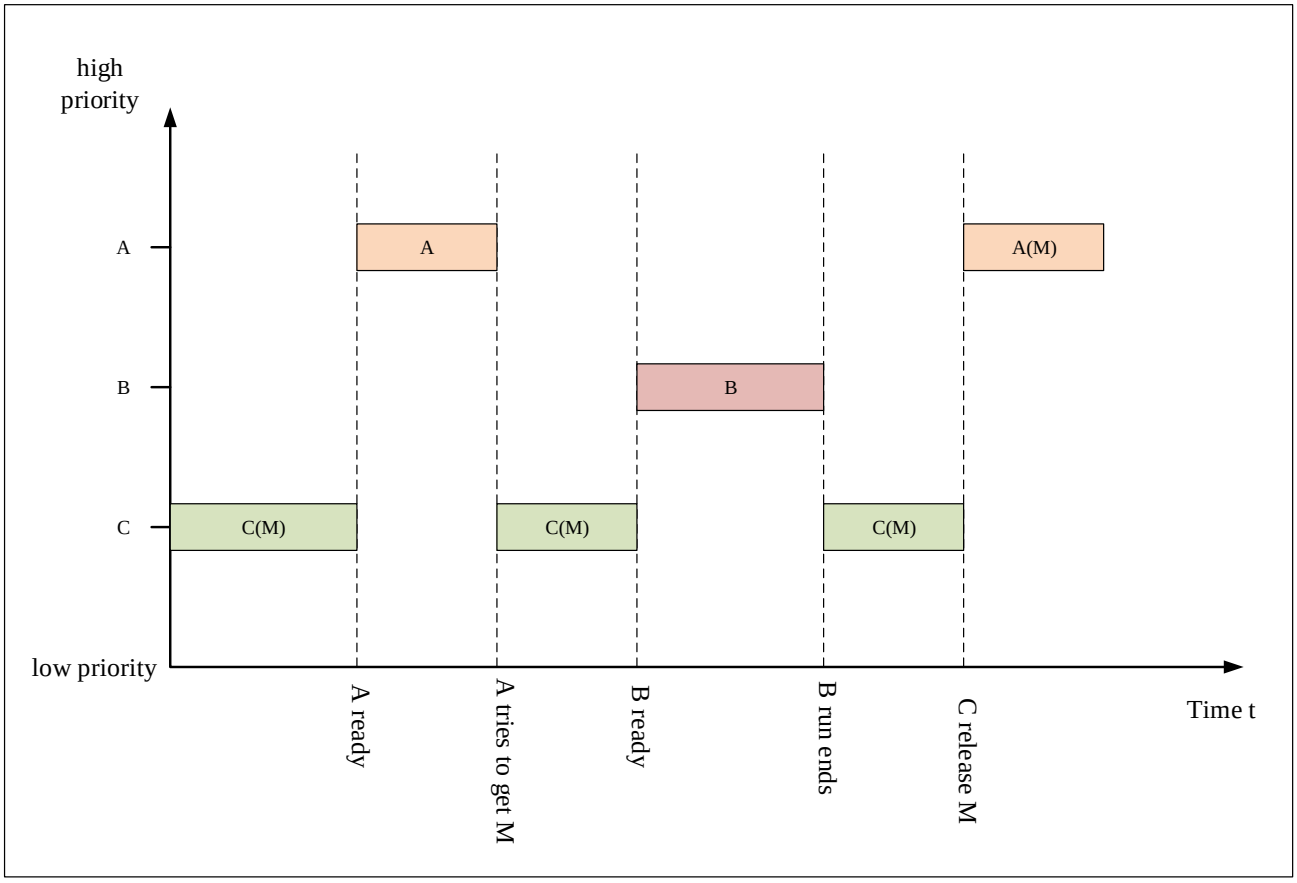
图 2-2 互斥量工作示意图



使用信号量会导致的另一个潜在问题是线程优先级翻转问题。所谓优先级翻转，即当一个高优先级线程试图通过信号量机制访问共享资源时，如果该信号量已被一低优先级线程持有，而这个低优先级线程在运行过程中可能又被其它一些中等优先级的线程抢占，因此造成高优先级线程被许多具有较低优先级的线程阻塞，实时性难以得到保证。如图 2-3 所示：有优先级为 A、B 和 C 的三个线程，优先级 A>B>C。线程 A，B 处于挂起状态，等待某一事件触发，线程 C 正在运行，此时线程 C 开始使用某一共享资源 M。在使用过程中，线程 A 等待的事件到来，线程 A 转为就绪态，因为它比线程 C 优先级高，所以立即执行。但是当线程 A 要使用共享资源 M 时，由于其正在被线程 C 使用，因此线程 A 被挂起切换到线程 C 运行。如果此时线程 B 等待的事件到来，则线程 B 转为就绪态。由于线程 B 的优先级比线程 C 高，因此线程 B 开始运行，直到其运行完毕，线程 C 才开始运行。只有当线程 C 释放共享资源 M 后，线程 A 才得以执行。在这种情况下，优先级发生了翻转：线程 B 先于线程 A 运行。这样便不能保证高优先级线程的响应时间。

国民技术股份有限公司 Nations Technologies Inc.
地址：深圳市南山区高新北区宝深路 109 号国民技术大厦
电话：+86-755-86309900 传真：+86-755-86169100
邮箱：info@nationstech.com 邮编：518057

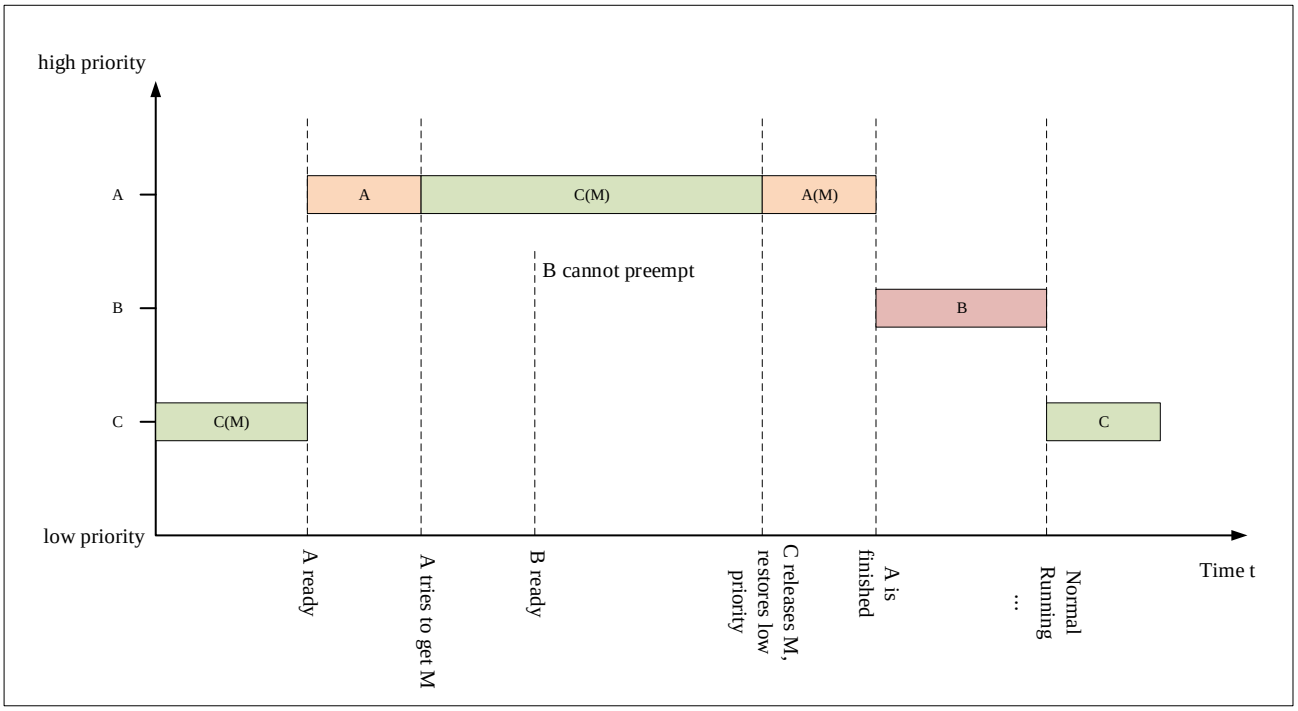
图 2-3 优先级反转（M 为信号量）



在 RT-Thread 操作系统中，互斥量可以解决优先级翻转问题，实现的是优先级继承算法。优先级继承是通过在线程 A 尝试获取共享资源而被挂起的期间内，将线程 C 的优先级提升到线程 A 的优先级别，从而解决优先级翻转引起的问题。这样能够防止 C（间接地防止 A）被 B 抢占，如图 2-4 所示。优先级继承是指，提高某个占有某种资源的低优先级线程的优先级，使之与所有等待该资源的线程中优先级最高的那个线程的优先级相等，然后执行，而当这个低优先级线程释放该资源时，优先级重新回到初始设定。因此，继承优先级的线程避免了系统资源被任何中间优先级的线程抢占。

国民技术股份有限公司 Nations Technologies Inc.
地址：深圳市南山区高新北区宝深路 109 号国民技术大厦
电话：+86-755-86309900 传真：+86-755-86169100
邮箱：info@nationstech.com 邮编：518057

图 2-4 优先级继承（M 为互斥量）



注意：在获得互斥量后，请尽快释放互斥量，并且在持有互斥量的过程中，不得再行更改持有互斥量线程的优先级。

Mutex

```
/* Create the mutex */
rt_mutex_init(&static_mutex,
              "smutex",
              RT_IPC_FLAG_FIFO);

/* Get the mutex */
rt_mutex_take(&static_mutex,
              10);

/* Relase the mutex */
rt_mutex_detach(&static_mutex);
```

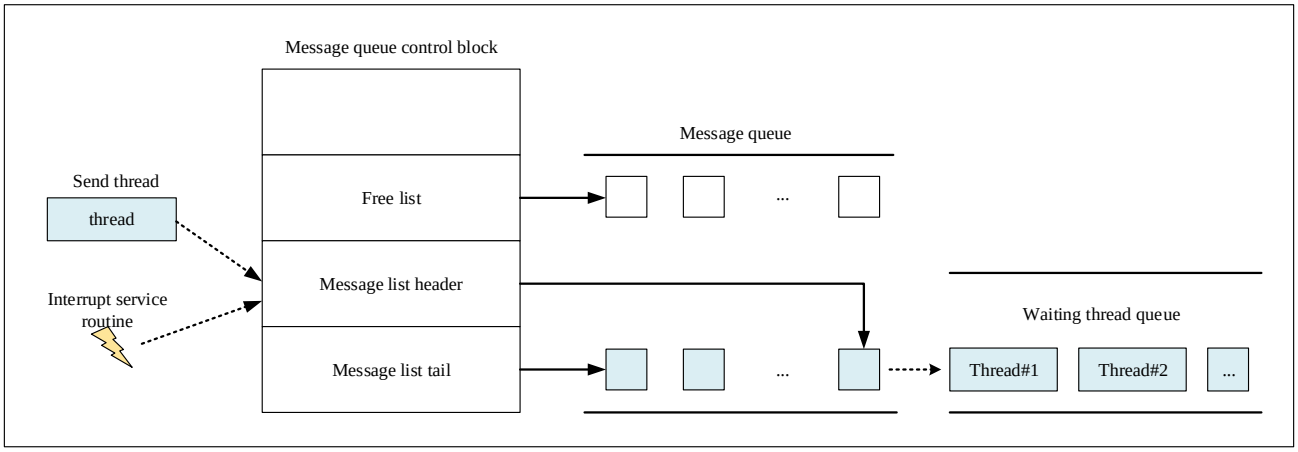
2.4 消息队列示例

消息队列能够接收来自线程或中断服务例程中不固定长度的消息，并把消息缓存在自己的内存空间中。其他线程也能够从消息队列中读取相应的消息，而当消息队列是空的时候，可以挂起读取线程。当有新的消息到达时，挂起的线程将被唤醒以接收并处理消息。消息队列是一种异步的通信方式。

如图 2-5 所示，线程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，一个或多个线程也可以从消息队列中获得消息。当有多个消息发送到消息队列时，通常将先进入消息队列的消息先传给线程，也就是说，线程先得到的是最先进入消息队列的消息，即先进先出原则(FIFO)。

国民技术股份有限公司 Nations Technologies Inc.
地址：深圳市南山区高新北区宝深路 109 号国民技术大厦
电话：+86-755-86309900 传真：+86-755-86169100
邮箱：info@nationstech.com 邮编：518057

图 2-5 消息队列工作示意图



RT-Thread 操作系统的消息队列对象由多个元素组成，当消息队列被创建时，它就被分配了消息队列控制块：消息队列名称、内存缓冲区、消息大小以及队列长度等。同时每个消息队列对象中包含着多个消息框，每个消息框可以存放一条消息；消息队列中的第一个和最后一个消息框被分别称为消息链表头和消息链表尾，对应于消息队列控制块中的 msg_queue_head 和 msg_queue_tail；有些消息框可能是空的，它们通过 msg_queue_free 形成一个空闲消息框链表。所有消息队列中的消息框总数即是消息队列的长度，这个长度可在消息队列创建时指定。

Message queue

```
/* Create the message queue*/
rt_mq_init(&mq,
           "mqt",
            &msg_pool[0],
            128- sizeof(void*),
            sizeof(msg_pool),
            RT_IPC_FLAG_FIFO);

/* Send the message queue*/
rt_mq_send(&mq,
            &key_info[0],
            sizeof(key_info));

/* Receive the message queue*/
rt_mq_recv(&mq,
            &buf[0],
            sizeof(buf),
            RT_WAITING_FOREVER);
```

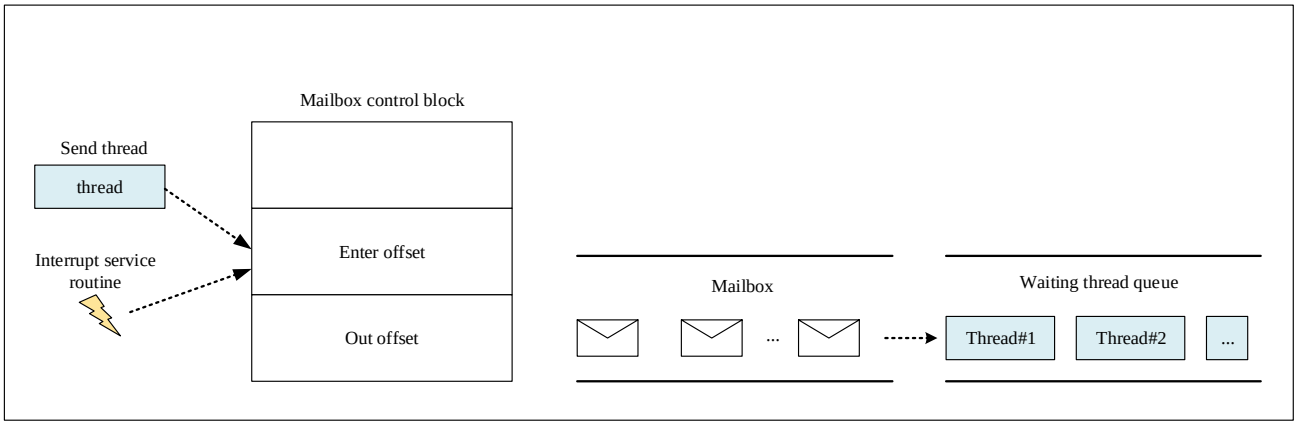
2.5 邮箱示例

RT-Thread 操作系统的邮箱用于线程间通信，特点是开销比较低，效率较高。邮箱中的每一封邮件只能容纳固定的 4 字节内容（针对 32 位处理系统，指针的大小即为 4 个字节，所以一封邮件恰好能够容纳一个指

国民技术股份有限公司 Nations Technologies Inc.
地址：深圳市南山区高新北区宝深路 109 号国民技术大厦
电话：+86-755-86309900 传真：+86-755-86169100
邮箱：info@nationstech.com 邮编：518057

针）。典型的邮箱也称作交换消息，如图 2-6 所示，线程或中断服务例程把一封 4 字节长度的邮件发送到邮箱中，而一个或多个线程可以从邮箱中接收这些邮件并进行处理。

图 2-6 邮箱工作示意图



Mailbox

```
/* Create the mailbox*/
rt_mb_init(&mb,
           "mbt",
           &mb_pool[0],
           sizeof(mb_pool)/4,
           RT_IPC_FLAG_FIFO);

/* Send the mailbox*/
rt_mb_send(&mb,
           (rt_uint32_t )&key_info[0]);

/* Receive the mailbox*/
rt_mb_recv(&mb,
           (rt_uint32_t*)&str,
           RT_WAITING_FOREVER);
```

2.6 事件示例

事件集主要用于线程间的同步，与信号量不同，它的特点是可以实现一对多，多对多的同步。即一个线程与多个事件的关系可设置为：其中任意一个事件唤醒线程，或几个事件都到达后才唤醒线程进行后续的处理；同样，事件也可以是多个线程同步多个事件。这种多个事件的集合可以用一个 32 位无符号整型变量来表示，变量的每一位代表一个事件，线程通过“逻辑与”或“逻辑或”将一个或多个事件关联起来，形成事件组合。事件的“逻辑或”也称为是独立型同步，指的是线程与任何事件之一发生同步；事件“逻辑与”也称为是关联型同步，指的是线程与若干事件都发生同步。

RT-Thread 定义的事件集有以下特点：

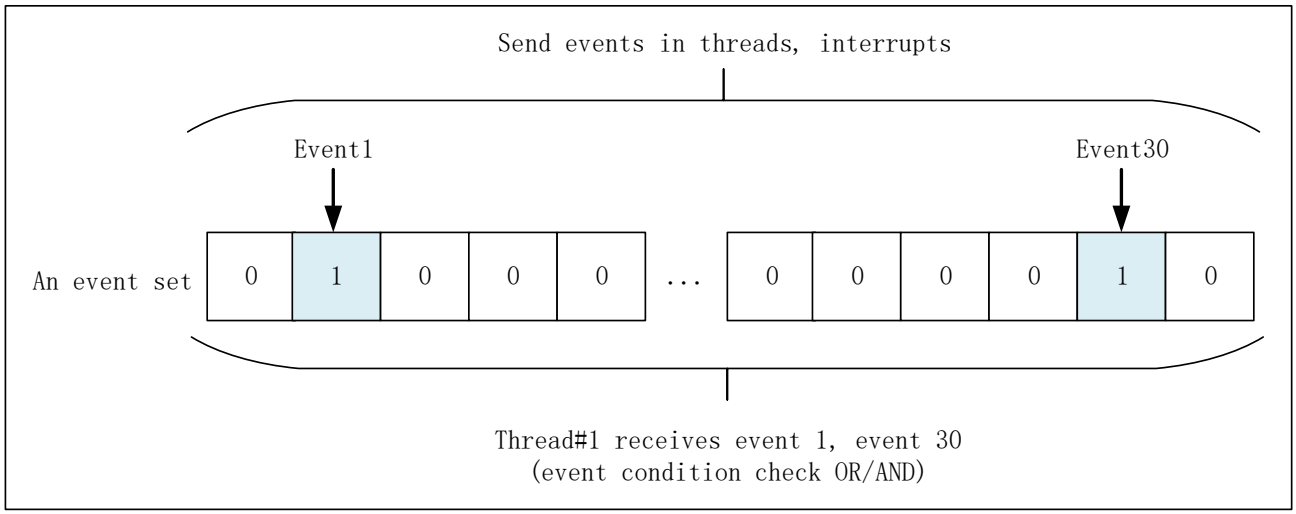
1) 事件只与线程相关，事件间相互独立：每个线程可拥有 32 个事件标志，采用一个 32 bit 无符号整型数进行记录，每一个 bit 代表一个事件

国民技术股份有限公司 Nations Technologies Inc.
地址：深圳市南山区高新北区宝深路 109 号国民技术大厦
电话：+86-755-86309900 传真：+86-755-86169100
邮箱：info@nationstech.com 邮编：518057

2) 事件仅用于同步，不提供数据传输功能

3) 事件无排队性，即多次向线程发送同一事件(如果线程还未来得及读走)，其效果等同于只发送一次。在 RT-Thread 中，每个线程都拥有一个事件信息标记，它有三个属性，分别是 RT_EVENT_FLAG_AND(逻辑与)，RT_EVENT_FLAG_OR(逻辑或）以及 RT_EVENT_FLAG_CLEAR(清除标记）。当线程等待事件同步时，可以通过 32 个事件标志和这个事件信息标记来判断当前接收的事件是否满足同步条件

图 2-7 事件工作示意图



如图 2-7 所示，线程#1 的事件标志中第 1 位和第 30 位被置位，如果事件信息标记位设为逻辑与，则表示线程#1 只有在事件 1 和事件 30 都发生以后才会被触发唤醒，如果事件信息标记位设为逻辑或，则事件 1 或事件 30 中的任意一个发生都会触发唤醒线程#1。如果信息标记同时设置了清除标记位，则当线程#1 唤醒后将主动把事件 1 和事件 30 清为零，否则事件标志将依然存在（即置 1）。

Event

```
/* Create the event*/
rt_event_init( &event,
               "event",
               RT_IPC_FLAG_FIFO);

/* Send the event */
rt_event_send( &event,
               (1 << 0)) ;

/* Receive the event s*/
rt_event_recv( &event,
              ((1 << 0) | (1 << 1)),
               RT_EVENT_FLAG_AND | RT_EVENT_FLAG_CLEAR,
               10,
               &evt);
```


国民技术股份有限公司 Nations Technologies Inc.
地址：深圳市南山区高新北区宝深路 109 号国民技术大厦
电话：+86-755-86309900 传真：+86-755-86169100
邮箱：info@nationstech.com 邮编：518057

3 补充说明

在嵌入式领域有多种不同 CPU 架构，例如 Cortex-M、ARM920T、MIPS32、RISC-V 等等。为了使 RT-Thread 能够在不同 CPU 架构的芯片上运行，RT-Thread 提供了一个 libcpu 抽象层来适配不同的 CPU 架构。libcpu 层向上对内核提供统一的接口，包括全局中断的开关，线程栈的初始化，上下文切换等。

RT-Thread 的 libcpu 抽象层向下提供了一套统一的 CPU 架构移植接口，这部分接口包含了全局中断开关函数、线程上下文切换函数、时钟节拍的配置和中断函数、Cache 等等内容。下表是 CPU 架构移植需要实现的接口和变量。

表 3-1 libcpu 移植相关 API

函数和变量	描述
rt_base_t rt_hw_interrupt_disable(void);	关闭全局中断
void rt_hw_interrupt_enable(rt_base_t level);	打开全局中断
rt_uint8_t *rt_hw_stack_init(void *tentry, void *parameter, rt_uint8_t *stack_addr, void *texit);	线程栈的初始化，内核在线程创建和线程初始化里面会调用这个函数
void rt_hw_context_switch_to(rt_uint32 to);	没有来源线程的上下文切换，在调度器启动第一个线程的时候调用，以及在 signal 里面会调用
void rt_hw_context_switch(rt_uint32 from,rt_uint32 to);	从 from 线程切换到 to 线程，用于线程和线程之间的切换
void rt_hw_context_switch_interrupt(rt_uint32 from, rt_uint32 to);	从 from 线程切换到 to 线程，用于中断里面进行切换的时候使用
rt_uint32_t rt_thread_switch_interrupt_flag;	表示需要在中断里进行切换的标志
rt_uint32_t rt_interrupt_from_thread, rt_interrupt_to_thread;	在线程进行上下文切换时候，用来保存 from 和 to 线程

国民技术股份有限公司 Nations Technologies Inc.
地址：深圳市南山区高新北区宝深路 109 号国民技术大厦
电话：+86-755-86309900 传真：+86-755-86169100
邮箱：info@nationstech.com 邮编：518057

4 版本历史

日期	版本	修改
2021.01.08	V1.0	初始版本

国民技术股份有限公司 Nations Technologies Inc.
地址：深圳市南山区高新北区宝深路 109 号国民技术大厦
电话：+86-755-86309900 传真：+86-755-86169100
邮箱：info@nationstech.com 邮编：518057

5 声明

国民技术股份有限公司（下称“国民技术”）对此文档拥有专属产权。依据中华人民共和国的法律、条约以及世界其他法域相适用的管辖，此文档及其中描述的国民技术产品（下称“产品”）为公司所有。

国民技术在此并未授予专利权、著作权、商标权或其他任何知识产权许可。所提到或引用的第三方名称或品牌（如有）仅用作区别之目的。

国民技术保留随时变更、订正、增强、修改和改良此文档的权利，恕不另行通知。请使用人在下单购买前联系国民技术获取此文档的最新版本。

国民技术竭力提供准确可信的资讯，但即便如此，并不推定国民技术对此文档准确性和可靠性承担责任。

使用此文档信息以及生成产品时，使用者应当进行合理的设计、编程并测试其功能性和安全性，国民技术不对任何因使用此文档或本产品而产生的任何直接、间接、意外、特殊、惩罚性或衍生性损害结果承担责任。

国民技术对于产品在系统或设备中的应用效果没有任何故意或保证，如有任何应用在其发生操作不当或故障情况下，有可能致使人员伤亡、人身伤害或严重财产损失，则此类应用被视为“不安全使用”。

不安全使用包括但不限于：外科手术设备、原子能控制仪器、飞机或宇宙飞船仪器、所有类型的安全装置以及其他旨在支持或维持生命的应用。

所有不安全使用的风险应由使用人承担，同时使用人应使国民技术免于因为这类不安全使用而导致被诉、支付费用、发生损害或承担责任时的赔偿。

对于此文档和产品的任何明示、默示之保证，包括但不限于适销性、特定用途适用性和不侵权的保证责任，国民技术可在法律允许范围内进行免责。

未经明确许可，任何人不得以任何理由对此文档的全部或部分进行使用、复制、修改、抄录和传播。

国民技术股份有限公司 Nations Technologies Inc.
地址：深圳市南山区高新北区宝深路 109 号国民技术大厦
电话：+86-755-86309900 传真：+86-755-86169100
邮箱：info@nationstech.com 邮编：518057