UIP-Kernel 程序员手册

撰写人：UIP 团队

目　录

目　　录..1
一、引言..6
1.1 编写目的..6
1.2 系统理念..7
1.3 系统概述..8
1.3.1 任务管理..8
1.3.2 中断管理..8
1.3.3 事件管理..9
1.3.4 资源管理..9
1.3.5 警报..10
1.3.6 回调机制..10
二、UIP-Kernel 实时操作系统体系结构..11
2.1 处理等级..11
2.2 任务管理..13
2.2.1 任务的概念..13
2.2.1.1 基本任务..13
2.2.1.2 扩展任务..13
2.2.2 任务状态模型..13
2.2.2.1 扩展任务..13
2.2.2.2 基本任务..15
2.2.2.3 两种任务类型的比较..16
2.2.3 激活一个任务..16
2.2.4 任务切换机制..16
2.2.5 任务优先级..16
2.2.6 调度策略..17
2.2.6.1 完全抢占式调度..17
2.2.6.2 非抢占式调度..18
2.2.6.3 任务组..19
2.2.6.4 混和抢占式调度..19
2.2.6.5 调度策略的选择..19
2.2.7 任务的终止..20
2.3 中断处理..21
2.4 事件机制..23

2.5 资源管理......25
2.5.1 访问并占有资源时的处理方法......25
2.5.2 使用资源时的限制......25
2.5.3 作为资源的调度程序......26
2.5.4 同步机制的一般问题和解决方案......26
2.5.4.1 问题......26
2.5.4.2 解决方案......27
2.5.5 内部资源......30
2.6 警报......32
2.6.1 计数器......32
2.6.2 警报管理......32
2.6.3 报警回调函数......33
2.7 错误处理、跟踪与调试......34
2.7.1 回调程序......34
2.7.2 错误处理......35
2.7.3 系统启动......37
2.7.4 系统关闭......38
2.7.5 调试......38
三、系统服务接口......39
3.1 系统服务描述......39
3.1.1 系统对象的定义......39
3.1.2 约定......39
3.1.2.1 调用的类型......39
3.1.2.2 合法的调用......39
3.1.2.3 错误的特性......40
3.2 操作系统服务规范......42
3.2.1 常用数据类型......43
3.2.2 任务管理......44
3.2.2.1 数据类型......44
3.2.2.2 组成成员......44
3.2.2.2.1 DeclareTask......44
3.2.2.3 系统服务......44
3.2.2.3.1 ActiveTask......44
3.2.2.3.2 TerminateTask......45
3.2.2.3.3 ChainTask......45

3.2.2.3.4 Schedule 46
3.2.2.3.5 GetTaskID 47
3.2.2.3.6 GetTaskState 47
3.2.2.4 常量 48
3.2.2.5 命名规则 48
3.2.3 中断管理 49
3.2.3.1 数据类型 49
3.2.3.2 系统服务 49
3.2.3.2.1 EnableAllInterrupts 49
3.2.3.2.2 DisableAllInterrupts 49
3.2.3.2.3 ResumeAllInterrupts 50
3.2.3.2.4 SuspendAllInterrupts 50
3.2.3.2.5 ResumeOSInterrupts 51
3.2.3.2.6 SuspendOSInterrupts 52
3.2.3.3 命名规则 52
3.2.4 资源管理 53
3.2.4.1 数据类型 53
3.2.4.2 组成成员 53
3.2.4.2.1 DeclareResource 53
3.2.4.3 系统服务 53
3.2.4.3.1 GetResource 53
3.2.4.3.2 ReleaseResource 54
3.2.4.4 常量 54
3.2.5 事件控制 54
3.2.5.1 数据类型 54
3.2.5.2 组成成员 55
3.2.5.2.1 DeclareEvent 55
3.2.5.3 系统服务 55
3.2.5.3.1 SetEvent 55
3.2.5.3.2 ClearEvent 56
3.2.5.3.3 GetEvent 56
3.2.5.3.4 WaitEvent 56
3.2.6 消息 57
3.2.6.1 数据类型 57
3.2.6.2 系统服务 57

3.2.6.2.1 SendMessage 57
3.2.6.2.2 ReceiveMessage..58
3.2.6.2.3 GetMessageStatus 58
3.2.6.2.4 ReadFlag...59
3.2.6.2.5 ResetFlag 59
3.2.7 警报 60
3.2.7.1 数据类型 60
3.2.7.2 组成成员 60
3.2.7.2.1 DeclareAlarm..60
3.2.7.3 系统服务 60
3.2.7.3.1 GetAlarmBase..60
3.2.7.3.2 GetAlarm 61
3.2.7.3.3 SetRelAlarm 61
3.2.7.3.4 SetAbsAlarm..62
3.2.7.3.5 CancelAlarm 63
3.2.7.4 常量...64
3.2.7.5 命名规则 64
3.2.8 操作系统执行控制...65
3.2.8.1 数据类型 65
3.2.8.2 系统服务 65
3.2.8.2.1 GetActiveApplicationMode 65
3.2.8.2.2 StartOS 65
3.2.8.2.3 ShutdownOS 65
3.2.8.3 常量...66
3.2.9 回调程序...66
3.2.9.1 数据类型 66
3.2.9.2 系统服务 66
3.2.9.2.1 ErrorHook...66
3.2.9.2.2 PreTaskHook...67
3.2.9.2.3 PostTaskHook 67
3.2.9.2.4 StartupHook..67
3.2.9.2.5 ShutdownHook 67
3.2.9.3 常数...68
3.2.9.4 宏..68
四、软件应用..69

4.1 运行时上下文的配置 69
4.2 应用设计建议 70
4.2.1 资源管理 70
4.2.1.1 依照 LIFO 的资源占用 70
4.2.1.2 API 服务的调用等级 71
4.2.1.3 任务终止或中断结束时资源仍然被占据的情况 71
4.2.2 API 调用的位置 72
4.2.3 中断服务程序 72
4.2.3.1 不同类型中断的嵌套 72
4.2.3.2 中断层的直接操作 73
4.2.4 优先级和抢占 73
4.2.5 内部资源用法举例 73
4.2.6 传递给 shutdownOS 的参数 74
4.2.7 错误处理 74
4.2.8 错误和警告 75

一、引言

嵌入式实时操作系统 UIP-Kernel，属于统一/通用智能平台（Unified Intelligent Platform，简称 UIP）的一部分，也是后者的重要支撑构件。

1.1 编写目的

本文档主要介绍如何基于 UIP-Kernel 实时操作系统进行应用开发，包括简单介绍 UIP-Kernel 系统的主要功能、各个模块的应用接口等。

1.2 系统理念

UIP-Kernel 设计的首要目标是具有强实时性，因此 UIP-Kernel 操作系统提供先进的调度机制、调度算法与通信协议，保证系统的强实时性，并提供精简的函数集来支持事件驱动的控制系统。由于 UIP-Kernel 希望适用于任何类型的控制单元，因此它应该能够在多种硬件上支持对时间要求严格的应用，即，具备高度模块化和灵活的可配置性。对于时间要求严格的应用来说，动态地生成系统对象并不合适，应该在专门的系统对象生成阶段生成。为了尽可能减少对系统速度的影响，应尽量避免对系统内部错误的检测，UIP-Kernel 定义了具有扩展能力的错误检测框架，用于在测试阶段或者对实时性要求不高的场合进行系统错误检测。

UIP-Kernel 操作系统具有以下特点：

（1）标准化的接口

应用软件和 UIP-Kernel 之间的接口是由系统服务定义的，系统服务符合 ISO/类 ANSI-C 语法。UIP-Kernel 应用于不同类型的处理器时其接口是完全相同的。

（2）可剪裁

UIP-Kernel 具有系统的可配置性，能广泛用于不同的应用场合和硬件系统。

UIP-Kernel 操作系统只需要最小化的系统资源(RAM、ROM、CPU 时间)，因此可运行于 8 位处理器。

（3）错误检查

UIP-Kernel 利用 hook 机制提供两种不同级别的错误检查：（1）扩展状态，用于开发阶段；（2）标准状态，用于产品阶段。

扩展状态能进行深层次的正确性检查。由于执行了额外逻辑，该状态下系统的执行时间和内存空间使用量均多于标准状态。

（4）应用程序的可移植性

UIP-Kernel 的设计目标之一是支持应用软件的可移植性和可重用性。因此，应用软件与操作系统之间的接口是由标准系统服务定义的。使用标准化的系统服务简化了应用软件的维护和移植，降低了开发成本，实现了源代码级移植。

1.3 系统概述

UIP-Kernel 操作系统提供的功能模块包括：任务管理（启动任务、终止任务、终止的同时启动一个任务）；事件管理（设置事件、等待事件）；资源管理（获取资源、释放资源）；消息传递（发送消息、接收消息）；警报管理（设置警报、删除警报）；中断处理程序框架；Hook 机制。

1.3.1 任务管理

在 UIP-Kernel 中任务是最小的调度单位，系统不限制任务数量的多少，只和硬件平台的具体内存相关。任务分为基本任务和扩展任务两种。系统中定义了 256 个任务优先级（0—255），优先级 255 分配给了空闲任务。优先级数值越小，任务优先级越高。UIP-Kernel 还允许定义任务组来综合使用抢占式和非抢占式两种调度方式，对于优先级小于等于任务组中最高优先级的任务来说，组内任务类似不可抢占任务；对于优先级大于任务组中最高优先级的任务，组内任务类似可抢占任务。此外，UIP-Kernel 还支持混和抢占式调度，此时系统中既有可抢占式任务又有不可抢占式任务，调度策略根据运行任务的调度类型而定。如果正在运行的是不可抢占式任务，那么调度策略就为非抢占式调度；如果正在运行的是可抢占式任务，那么调度策略就为抢占式调度。

1.3.2 中断管理

UIP-Kernel 支持中断嵌套，嵌套层数从实时操作系统层面讲不受限制，只与芯片支持的中断优先级个数和分配的系统中断栈空间大小有关。中断响应时间和任务切换时间是微秒级的。

中断优先级的最大数取决于所用的控制器，同时也和具体的实现有关。中断的调度方法由硬件决定，与 UIP-Kernel 无关。也就是说，中断由硬件调度，任务由操作系统的调度程序调度。中断可以打断各种类型的任务（包括可抢占任务和不可抢占任务）。如果在一个中断服务程序中激活一个任务，那么要等到所有的中断程序结束后才会进行任务的调度。

在 UIP-Kernel 中处理中断的函数（中断服务程序 Interrupt Service Routine: ISR）可划分为两种类型：

1 型 ISR：这种 ISR 不使用操作系统服务。在 ISR 结束后，回到中断发生时的精确位置继续处理，即中断对任务的管理没有影响。这种中断的开销最小。

2 型 ISR：UIP-Kernel 操作系统提供了一个 ISR 的框架，为专用的用户程序准备运行时的环境。在系统生成时，用户程序就被分配给指定的中断。

在 ISR 内部，不会发生任务的重新调度。如果被中断的是可抢占式任务并且没有发生其他中断，那么在 2 型中断结束时，会发生任务的重新调度。

1.3.3 事件管理

事件是由 UIP-Kernel 操作系统管理的对象。它们不是独立的对象，而是分配给对应的扩展任务。每个扩展任务都有确定数目的事件。这个任务成为这些事件的“所有者”。一个独立的事件由它所属的任务及名字来标识。当扩展任务被激活时，操作系统清空其所有的事件。事件可以用来为其所属的扩展任务传递二进制信息。事件的含义是由具体应用所定义的，如定时器到时、资源可用或者收到消息等。

对事件可以有多种操作，这取决于该任务是事件的所有者还是其他任务（不一定是扩展任务）。所有的任务都可以为非挂起态的扩展任务设置事件，但只有事件的所有者可以清除事件，并等待事件的接收（也就是设置）。

事件的发生是扩展任务从等待态转为就绪态的条件。操作系统提供了事件的设置、清除、查询以及等待事件发生等服务。

任何任务和 2 型的 ISR 都可以为非挂起态的扩展任务设置事件，通过事件来通知扩展任务任何状态的变化。

在任何情况下，事件的接收者都是一个扩展任务。因此，一个中断服务程序或者一个基本任务是不会等待事件发生的。事件只能被其所属任务清除。

只要扩展任务等待的事件中有一个发生，它就由等待态释放为就绪态。如果一个正在运行的扩展任务试图等待某一事件而该事件已经发生，那么这个任务仍然保持运行态。

1.3.4 资源管理

资源管理的作用是协调不同优先级的任务对共享资源如管理实体（调度程序）、程序执行、内存或者硬件资源等的访问。

由于 UIP-Kernel 操作系统采用了优先级天花板协议，因此，不会出现任务或者中断企图访问已占有资源的情况，能防止优先级倒置与死锁。

如果资源的概念是用于协调任务以及中断，那么 UIP-Kernel 操作系统能够保证只有当一个中断服务程序执行时所需的全部资源都已经被释放后才开始执行。

UIP-Kernel 严格禁止对同一个资源的嵌套访问。在极少需要进行嵌套访问资源的情况下，推荐使用一个和第一个资源完全相同的新资源。

1.3.5 警报

UIP-Kernel 操作系统为处理重复发生的事件提供了服务，例如以固定间隔提供中断的定时器，或者在凸轮轴或机轴的角度匀速变化的时候产生中断的译码器，或者其他常规应用的特定触发器。

在处理这些事件时，UIP-Kernel 操作系统采用了“两级”的概念。重复发生的事件（源）由具体实现中的记数器来注册，基于记数器，操作系统软件为应用软件提供报警机制。

在 UIP-Kernel 操作系统中，一个硬件或软件定时器至少可以提供一个计数器，一个计数器可以关联多个警报。当计数器达到预先定义的计数值时，警报到达。当警报到达时，UIP-Kernel 操作系统会提供诸如激活任务、设置事件或者调用报警回调程序等服务。报警回调程序是由具体应用提供的一个函数。

报警有单次报警和循环报警两类。除此之外，操作系统还提供了取消报警和获取报警的当前状态等服务。

1.3.6 回调机制

UIP-Kernel 操作系统提供系统特殊的回调程序，允许在操作系统内部处理中调用用户自定义的动作。

二、UIP-Kernel 实时操作系统体系结构

2.1 处理等级

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作出快速响应，并控制所有实时任务协调一致运行的操作系统。实时操作系统有硬实时和软实时之分，硬实时要求在规定的时间内必须完成操作，这是在操作系统设计时保证的；软实时则只要按照任务的优先级，尽可能快地完成操作即可。UIP-Kernel 正是一个嵌入式硬实时操作系统，它为应用程序提供处理器之上的运行环境，可以控制多个实时进程的执行，使之看起来像是在并行运行一样。

UIP-Kernel 操作系统为用户提供了一个说明详细的接口集。这些接口被竞争 CPU 的实体使用。这样的实体分为以下两类：

（1）操作系统管理的中断服务程序。

（2）任务（基本任务和扩展任务）。

处理等级被定义为某一范围的连续的数值，用来对任务或者中断服务程序进行处理。UIP-Kernel 定义了三个处理等级：

（1）中断等级。

（2）调度程序的逻辑等级。

（3）任务等级。

这三个处理等级的优先级满足以下规则：

（1）中断优先级高于任务。

（2）中断处理等级包括一个或多个中断优先级。

（3）中断服务程序的优先级是静态分配的。

（4）中断服务程序的优先级的分配取决于具体的实现和硬件体系结构。

（5）对于任务的优先级和资源的天花板优先级来说，优先级的值越小，其优先级越高。

（6）任务的优先级是由用户静态分配的。

在任务等级，根据用户为各种任务（非抢占式、完全抢占式、混和抢占式）

分配的优先级进行任务的调度。运行时上下文在任务执行一开始就被占用，在任务完成后被释放。

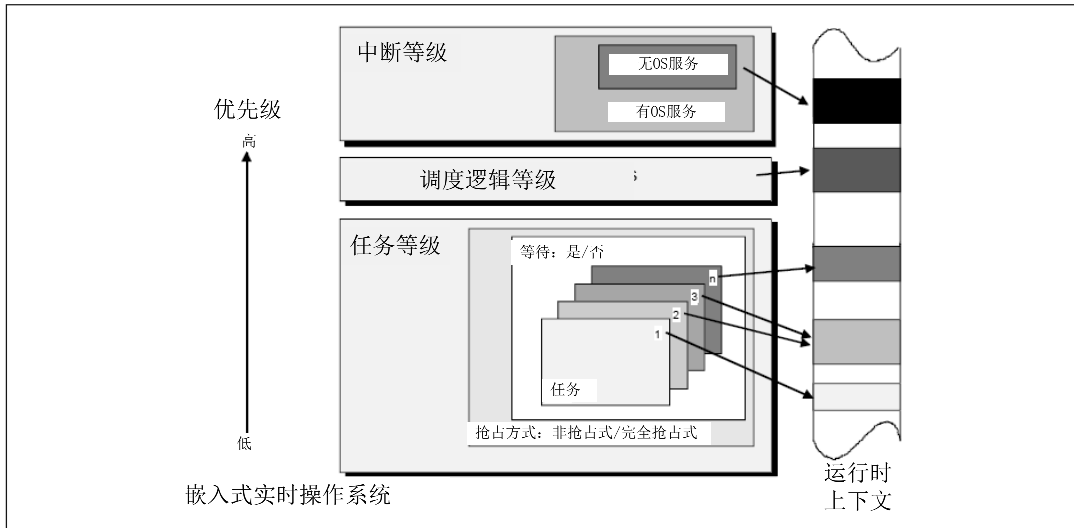

图 1 UIP-Kernel 操作系统的处理等级

处理等级被定义为某一范围的连续的数值，用来对任务或者中断服务程序进行处理。操作系统优先级到硬件优先级的映射由具体的实现来完成。

需要注意的是调度程序优先级的分配只是一个逻辑概念，不需直接使用优先级即可实现。另外，UIP-Kernel 并没有对任务的优先级与具体微处理器的硬件中断等级的关系做出任何限定。

2.2 任务管理

2.2.1 任务的概念

复杂的控制软件可以方便地根据他们的实时性要求划分为多个部分，这些部分可以通过不同任务来实现。任务是一个简单的程序，也称作一个线程，该程序可以认为 CPU 完全属于该程序占用，达到了虚拟化 CPU 的效果。每个任务被赋予一定的优先级，有它自己的一套 CPU 寄存器和自己的栈空间。

在多任务系统中，内核负责管理各个任务，或者说为每个任务分配 CPU 时间，并且负责任务之间的通信。

UIP-Kernel 操作系统提供了两种不同的任务的概念：基本任务和扩展任务。

2.2.1.1 基本任务

只有在如下几种情况下，基本任务才释放处理器的使用权：

- 任务终止
- UIP-Kernel 操作系统切换到更高优先级的任务
- 发生中断，处理器切换到中断服务程序（ISR）

2.2.1.2 扩展任务

扩展任务与基本任务的区别在于扩展任务可以调用操作系统服务 WaitEvent，使系统转入等待状态。处于等待状态时释放处理器，将处理器使用权交给低优先级的任务，但并不需要终止正在运行的扩展任务。

2.2.2 任务状态模型

由于处理器在任一时刻只能执行一个任务的一条指令，而同一时刻会有多个任务共同竞争处理器的使用权，因此任务不得不在不同的状态之间进行切换。当需要进行任务状态切换时由 UIP-Kernel 操作系统负责当前任务上下文的保存。

2.2.2.1 扩展任务

扩展任务有四种任务状态：

运行态 在运行态，任务掌握了 CPU 的使用权，其指令得以执行。在任一时刻只能有一个任务处于运行态，其它状态可以同时被多个任务采用。

就绪态 就绪态是任务进入运行态的必要条件，在此状态下任务已经准备好，只需等待分配处理器的使用权。多个任务处于此状态时，由调度机制

决定下一次执行哪个任务。

等待态 任务不能连续地执行，因为它至少需要等待某一事件的发生，此时任务处于等待态。

挂起态 处于挂起态的任务是非活动的，可以被再次激活。

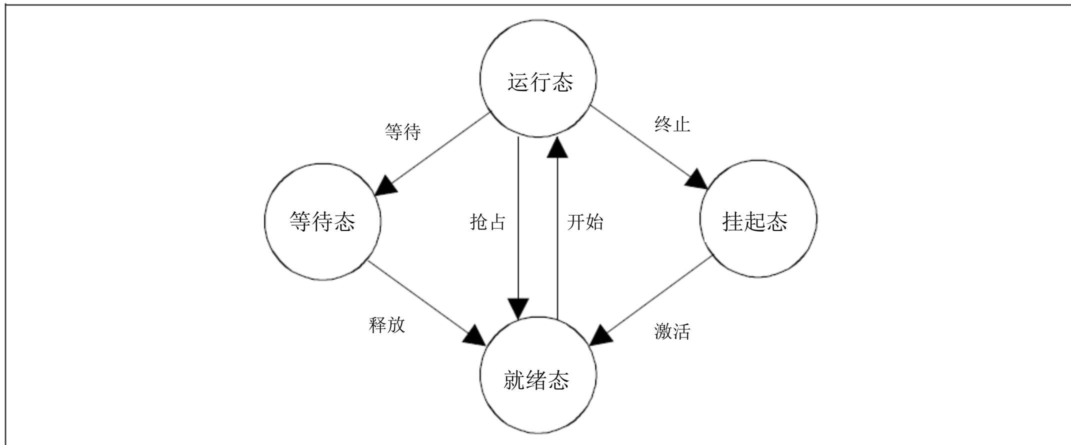

图 2 扩展任务状态模型

表 1 扩展任务状态转换

转换条件	**上一状态**	**新状态**	**描述**
激活	挂起	就绪	新的任务由系统服务置为就绪态，UIP-Kernel 操作系统保证任务的执行从其第一条指令开始。
开始	就绪	运行	由调度机制选择的就绪态任务得到执行。
等待	运行	等待	任务转为等待态是系统服务引起的，处于等待态的任务需要某一事件的触发才能继续执行。
释放	等待	就绪	处于等待态的任务等待的事件发生。
抢占	运行	就绪	调度机制决定开始另外的任务，处于运行态的任务转为就绪态。
终止	运行	挂起	运行态到挂起态的转换由系统服务引起。只有当任务自己结束自己时（self-terminatian）才会发生，这样可以降低操作系统的复杂度。

注意：任务不能从挂起态直接转为等待态，因为这种转换是多余的，并且会使任务调度变得更复杂。

2.2.2.2 基本任务

基本任务的状态模型与扩展任务几乎完全一样，不同之处在于基本任务没有等待状态。

运行态 在运行态，任务掌握了 CPU 的使用权，其指令得以执行。在任一时刻只能有一个任务处于此状态，其他状态可以同时被多个任务采用。

就绪态 就绪态是任务进入运行态的必要条件，在此状态下任务已经准备好，只需等待分配处理器的使用权。多个任务处于此状态时，由调度机制决定下一次执行哪个任务。

挂起态 处于挂起态的任务是非活动的，可以被再次激活。

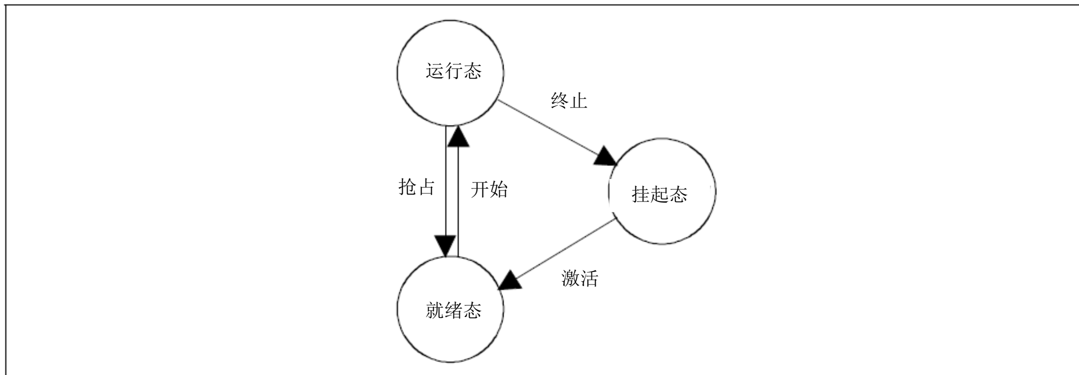

图 3 基本任务状态模型

表 2 基本任务状态转换

转换条件	上一状态	新状态	描述
激活	挂起	就绪	新的任务由系统服务置为就绪态，UIP-Kernel 操作系统保证任务的执行是从其第一条指令开始的。
开始	就绪	运行	由调度机制选择的就绪态任务得到执行
抢占	运行	就绪	调度机制决定开始另外的任务，处于运行态的任务转为就绪态
终止	运行	挂起	运行态到挂起态的转换由系统服务引起。只有当任务自己结束自己时（self-terminatian）才会发生，这样可以降低操作系统的复杂度。

2.2.2.3 两种任务类型的比较

基本任务没有等待状态，因此只包括任务开始和结束这两个同步时刻。如果应用含有内部的同步时刻，那么应该使用多个基本任务来实现。基本任务的一个优点是其运行时上下文对 RAM 的需求量适中，而扩展任务需要的系统资源更多。

扩展任务的一个优点是不管发生什么样的同步请求，都可以在单个任务中处理连贯的工作。当前没有可用于进一步处理的信息时，扩展任务就切换到等待状态。如果相应的事件发生或者所需的数据或者事件有更新时，任务离开等待状态。扩展任务的同步点也比基本任务多，但管理比基本任务复杂。

2.2.3 激活一个任务

任务的激活是由操作系统服务 ActivateTask 或者 ChainTask 实现的。激活后任务进入就绪态。

2.2.4 任务切换机制

与传统的顺序编程不同，多任务允许操作系统同时执行多个任务，因此 UIP-Kernel 实时操作系统要及时调度运行实时任务。调度是 UIP-Kernel 内核的主要职责之一，决定轮到哪个任务运行，并将正在运行任务的当前状态（CPU 寄存器中的全部内容）保存在任务自己的栈区，然后把下一个将要运行的任务的当前状态从该任务的栈中重新装入 CPU 的寄存器，并开始下一个任务的运行。

调度也可以被认为是一种可以被任务占有或者释放的资源。也就是说，任务可以占有调度权来避免任务切换，直到调度权被释放。

2.2.5 任务优先级

每个任务都有其优先级，任务越重要，赋予的优先级应越高。调度程序根据任务的优先级决定哪一个处于就绪态的任务可以转换为运行态。UIP-Kernel 支持 256 个任务优先级（0—255），优先级值越小，任务的优先级越高，255 被分配给空闲任务。优先级分为静态优先级和动态优先级。

（1） 静态优先级

应用程序执行过程中诸任务优先级不变，则称之为静态优先级。在静态优先级系统中，诸任务以及它们的时间约束在程序编译时是已知的。

（2）动态优先级

应用程序执行过程中，任务的优先级是可变的，则称之为动态优先级。

为了提高效率，UIP-Kernel 不支持动态优先级管理。因此，任务的优先级是静态定义的，即用户不能在程序执行时改变任务的优先级。但在特殊情况下，操作系统可以将任务的优先级进行提升。

2.2.6 调度策略

2.2.6.1 完全抢占式调度

完全抢占式调度是指根据操作系统预先设定的触发条件，当前处于运行态的任务可能被任意一条指令重新调度。在完全抢占式调度方式下，只要有高优先级的任务就绪，当前处于运行态的任务就被转为就绪态。此时任务的上下文会被保存，以便被抢占的任务能在被抢占的位置继续执行。

在完全抢占式调度方式下，反应时间与低优先级任务的运行时间无关。由于保存任务上下文而引起的不断增长的(RAM)内存需求量和为了保证任务间的同步而引起的复杂度的增加会带来一定的限制。由于从理论上讲每一个任务在任何位置都有可能被重新调度，因此多个任务共享的数据在使用时需要进行同步。

在图 4 中，低优先级的任务 2 并没有延迟高优先级任务 1 的调度。

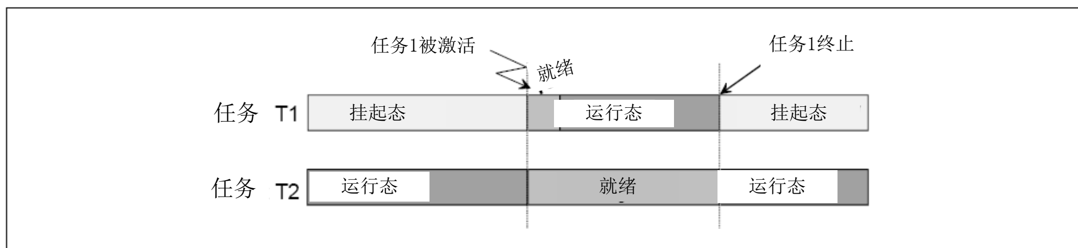

图 4 完全抢占式调度

在完全抢占式系统下，用户将可以一直期待抢占正在运行的任务的 CPU 使用权。即使任务的片断不能被抢占，也可以通过调用系统服务 GetResource 来暂时地阻断调度，以达到抢占的目的。

总结一下，在以下几种情况下，调度程序将被激活：

- 成功地终止一个任务（系统服务 TeminateTask）；
- 成功地终止一个任务，并且另一个任务被激活（系统服务 ChainTask）；
- 任务被来自任务级的系统服务激活（如 ActivateTask，或者在定义了任务

激活的情况下有消息通知、警告到达等)；

- 任务状态转换为等待态（只针对扩展任务)；
- 从任务级设置一个任务为等待态（例如系统服务 SetEvent，或者在定义了事件设置时有消息通知或者警告到达等)；
- 任务级的释放资源（系统服务 ReleaseResource)；
- 从中断级返回到任务级。

在中断服务程序的处理过程中，不会发生重新调度。

使用完全抢占式调度的应用软件不需要使用系统服务 Schedule，但是其他调度方式会用到该服务。为了使应用软件在不同的调度策略下可以移植，用户可以在自己认为正确分配 CPU 的使用权的位置强制地通过调用 Schedule 服务来进行调度。

2.2.6.2 非抢占式调度

只有使用明确定义的一组系统服务才能进行任务切换的调度方式被称之为非抢占式调度。非抢占式调度在任务实时性要求方面有些特殊的限制。尤其是一个正在运行的低优先级的任务如果正处于不可抢占的部分时，会将高优先级任务的开始延时到下一个重调度时刻。

在图 5 中，低优先级的任务 2 将高优先级的任务 1 延时到了下一个重调度时刻（在本例中是任务 2 的终止时刻)。

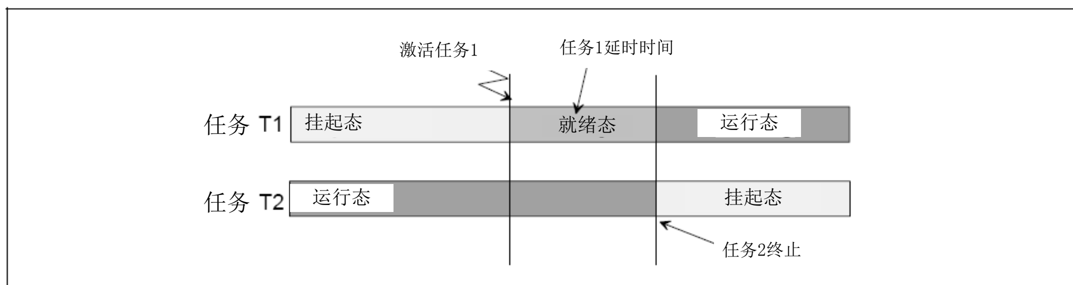

图 5 非抢占式任务调度

重新调度时刻

在非抢占式任务中，任务的重调度只发生在以下几种情况：

- 任务被系统服务 TerminateTask 成功终止
- 任务被成功终止，同时接下来的任务被系统服务 ChainTask 激活
- 直接调用系统服务 Schedule

- 调用系统服务 WaitEvent 将任务转为等待态

在非抢占式系统实现时应指定只有在最高的任务编程层次（而不是任务子函数）才能调用导致任务重调度的系统服务。

2.2.6.3 任务组

操作系统允许定义任务组来综合抢占式和非抢占式两种调度方式。对于优先级小于等于任务组中最高优先级的任务来说，组内任务类似不可抢占任务。对于优先级大于任务组中最高优先级的任务，组内任务类似可抢占任务。

不可抢占任务是内部资源最常见的一种应用，是具有分配了最高优先级的特殊内部资源的任务。

2.2.6.4 混和抢占式调度

如果在同一个系统中既有可抢占式任务又有不可抢占式任务，那么产生的调度策略就叫“混和抢占式调度”。在这种情况下，调度策略就根据正在运行的任务的调度类型而定。如果正在运行的是不可抢占式任务，那么调度策略就为非抢占式调度；如果正在运行的是可抢占式任务，那么调度策略就为抢占式调度。

如果是完全可抢占式操作系统，那么在如下几种情况下，不可抢占任务的定义有意义：

- 任务的执行时间与任务的切换时间为同一数量级
- 为了给任务上下文的保存提供空间，需要经济地使用 RAM
- 任务不会被抢占

很多应用中需要长时间并行执行的任务通常很少，对于这样的任务，使用完全抢占式操作系统会很方便，更多的是有确定的执行时间的短期任务，这时使用非抢占式调度会更有效。这种配置下，混和抢占式调度就发展成一种可选之举。

2.2.6.5 调度策略的选择

软件开发者或者系统集成者通过配置任务的优先级和指定作为任务属性的调度类型来决定任务的执行次序。

任务类型（基本任务还是扩展任务）与任务的调度类型（可抢占还是不可抢占）是相互独立的。因此一个完全抢占式系统可能包含基本任务，一个非抢占式系统也可能包含扩展任务。

如果一个系统服务正在运行，那么任务的抢占和上下文切换会被延时到服务结束。

2.2.7 任务的终止

在 UIP-Kernel 操作系统中，任务只能自己终止自己。

UIP-Kernel 操作系统提供了 ChainTask 服务以确保正在运行的任务终止后能立即激活新的任务。该函数会把新激活的任务放入优先级队列的最后。

每个任务都应该在代码的最后一句终止自己。结束任务时不调用 TerminationTask 或者 ChainTask 严格来讲是不允许的，会导致不可预料的结果。

2.3 中断处理

所谓中断，是指在 CPU 执行程序的过程中，当出现某种情况，由服务对象向 CPU 发出中断请求信号，要求 CPU 暂时中断当前程序的执行，而转去执行相应的处理程序，待处理程序执行完毕后，再返回来继续执行原来被打断的程序。也就是说，中断是通过硬件来改变 CPU 程序运行方向的一种技术，它既和硬件有关，也和软件有关。

从中断的定义我们可以看出中断应具备中断源、中断响应和中断返回这三个要素。中断源发出中断请求，系统对中断请求进行响应，当中断响应完成后应进行中断返回，返回被中断的地方继续执行原来被中断的程序。其处理过程如图 6 所示：

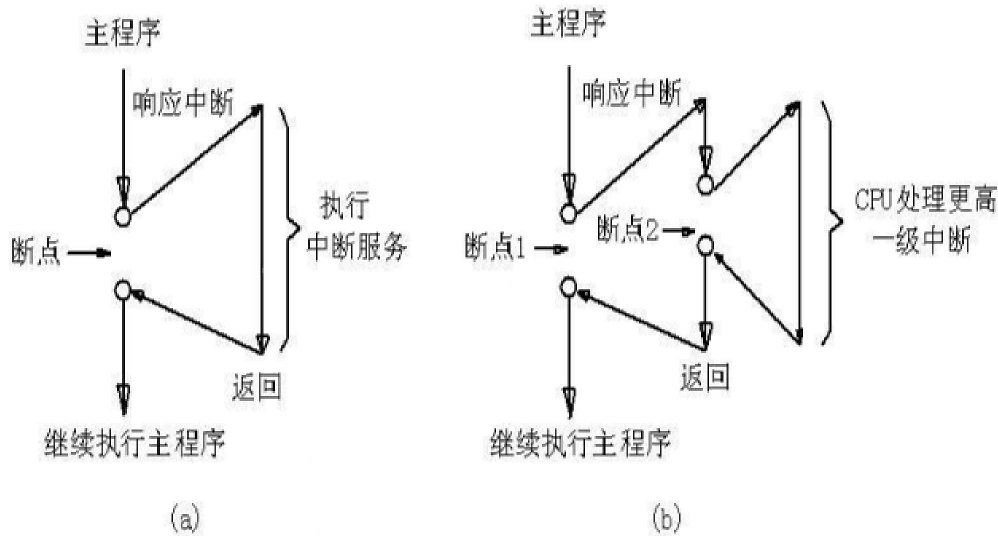

图 6 中断处理过程

在 UIP-Kernel 实时操作系统中，处理中断的函数（中断服务程序 Interrupt Service Routine: ISR）可划分为两种类型（图 7 所示）：

1 型 ISR：这种 ISR 不使用操作系统服务。在 ISR 结束后，回到中断发生时的精确位置继续处理，即中断对任务的管理没有影响。这种中断的开销最小。

2 型 ISR：UIP-Kernel 操作系统提供了一个 ISR 的框架，为专用的用户程序准备运行时的环境。在系统生成时，用户程序就被分配给指定的中断。

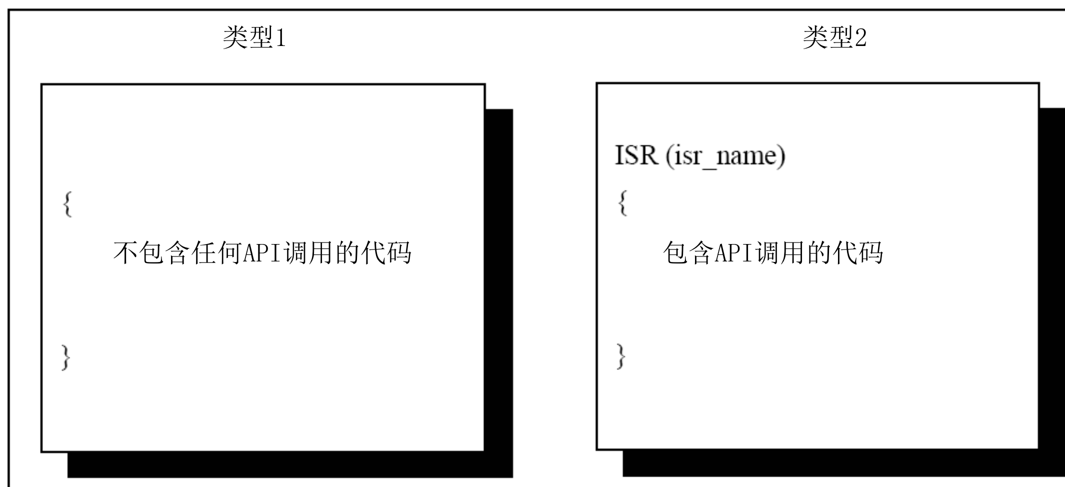

图 7 UIP-Kernel 操作系统的两种 ISR 类型

在 ISR 内部，不会发生任务的重新调度。如果被中断的是可抢占式任务并且没有发生其他中断，那么在 2 型中断结束时，会发生任务的重新调度。

在实现时须保证任务按照 UIP-Kernel 的调度时刻执行，为此，需要针对各个类型的 ISR 的中断优先级添加一些限制，并/或在配置时进行检查。

中断优先级的最大数取决于所用的控制器，同时也和具体的实现有关。中断的调度方法由硬件决定，与 UIP-Kernel 无关。也就是说，中断由硬件调度，任务由操作系统的调度程序调度。中断可以打断各种类型的任务（包括可抢占任务和不可抢占任务）。如果在一个中断服务程序中激活一个任务，那么要等到所有的中断程序结束后才会进行任务的调度。

快速禁止/使能 API 函数

UIP-Kernel 提供了快速禁止所有中断以及快速禁止所有 2 型中断的 API 函数。典型的应用是用来保护比较短的重要代码，在这些受保护的重要代码中，不允许从中断返回，也就是说，有一个"suspend/disable"就必须有一个与之相配对的"resume/enable"。唯一能在 Suspend- 和 Resume- 配对中调用的操作系统服务是 SuspendOSInterrupts / ResumeOSInterrupts 配对或者 SuspendAllInterrupts / ResumeAllInterrupts 配对。

2.4 事件机制

事件是由 UIP-Kernel 实时操作系统管理的对象。它们不是独立的对象，而是分配给对应的扩展任务。每个扩展任务都有确定数目的事件，这个任务成为这些事件的“所有者”。一个独立的事件由它所属的任务及名字来标识。当扩展任务被激活时，操作系统清空其所有的事件。事件可以用来为其所属的扩展任务传递二进制信息。事件的含义是由具体应用所定义的，如定时器到时、资源可用或者收到消息等。

对事件可以有多种操作，这取决于该任务是事件的所有者还是其他任务（不一定是扩展任务）。所有的任务都可以为非挂起态的扩展任务设置事件，但只有事件的所有者可以清除事件，并等待事件的接收（也就是设置）。

事件的发生是扩展任务从等待态转为就绪态的条件。操作系统提供了事件的设置、清除、查询以及等待事件发生的服务。

任何任务和 2 型的 ISR 都可以为非挂起态的扩展任务设置事件，通过事件来通知扩展任务任何状态的变化。

在任何情况下，事件的接收者都是一个扩展任务。因此，一个中断服务程序或者一个基本任务是不会等待事件发生的。事件只能被其所属任务清除。

只要扩展任务等待的事件中有一个发生，它就由等待态释放为就绪态。如果一个正在运行的扩展任务试图等待某一事件而该事件已经发生，那么这个任务仍然保持运行态。

图 8 解释了在完全抢占式调度下如何通过设置事件实现扩展任务间的同步，其中，扩展任务 1 优先级较高。

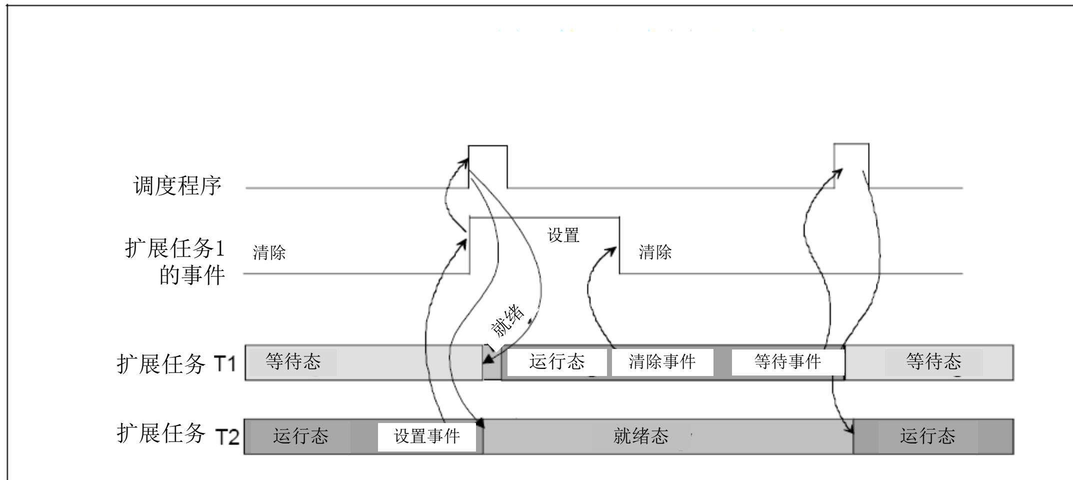

图 8 可抢占扩展任务的同步

图 8 说明了设置一个事件对处理过程的影响：任务 1 等待事件的发生，任务 2 为任务 1 设置了该事件，调度程序被激活。接着，任务 1 由等待态转为就绪态。由于任务 1 的优先级比较高，这就导致了任务的切换，任务 2 被任务 1 抢占。任务 1 清除事件。之后任务 1 就开始重新等待这个事件，调度程序继续任务 2 的执行。

如果是非抢占式调度，那么事件设置后调度不会立即开始（见图 9，其中扩展任务 1 优先级较高）。

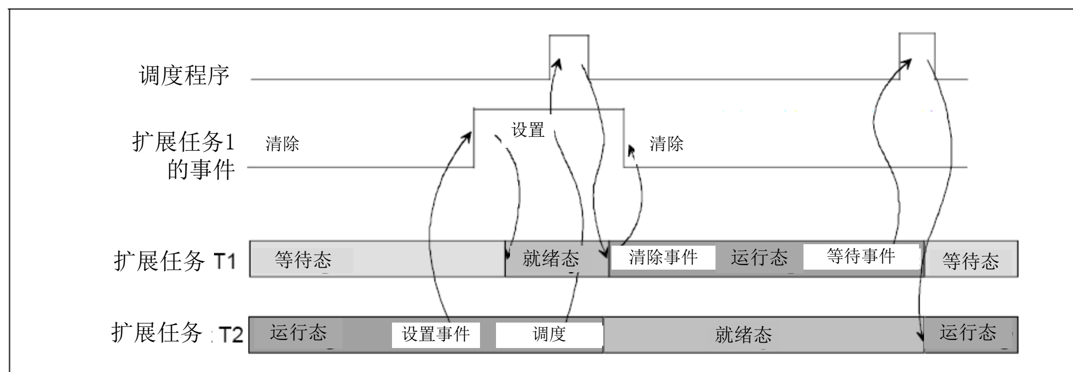

图 9 非抢占式扩展任务的同步

2.5 资源管理

资源管理的作用是协调不同优先级的任务对共享资源如管理实体（调度程序）、程序执行、内存或者硬件资源等的访问。

资源管理可以保证：

- 两个任务不能同时占有同一个资源
- 不会发生优先级反转
- 使用这些资源不会造成死锁
- 访问资源不会造成任务进入等待态

如果资源管理扩展到了中断级，那么它还能保证：

- 两个任务或中断程序不会同时占有同一个资源

资源管理函数用于以下几种情况：

- 抢占式任务
- 非抢占式任务，如果用户打算将应用代码执行于其他调度方式下
- 任务与中断服务程序中存在资源的共享
- 中断服务程序之间存在资源的共享

如果用户不仅要在任务切换时对共享资源进行保护，还要在发生中断时进行保护，那么需要使用操作系统服务来进行中断的禁止和使能，这两个操作不会导致任务重新调度。

2.5.1 访问并占有资源时的处理方法

由于 UIP-Kernel 操作系统采用了优先级天花板协议，因此，不会出现任务或者中断企图访问已占有资源的情况。

如果资源的概念是用于协调任务以及中断，那么 UIP-Kernel 操作系统能够保证只有当一个中断服务程序执行时所需的全部资源都已经被释放后才开始执行。

UIP-Kernel 严格禁止对同一个资源的嵌套访问。在极少需要进行嵌套访问资源的情况下，推荐使用一个和第一个资源完全相同的新资源。

2.5.2 使用资源时的限制

资源被占有时不能调用 TerminateTask、ChainTask、Schedule 和 WaitEvent。

当资源被占用时不能结束一个中断服务程序。在一个任务占用多个资源的情

况下，用户必须遵照 LIFO 原则来申请和释放资源。

2.5.3 作为资源的调度程序

任务可以通过锁定调度程序来保护自己不被其他任务抢占。调度程序此时就是一种所有任务都可以访问的资源。因此，系统会自动产生一个预命名为 RES_SCHEDULER 的资源。

中断的接收和处理与 RES_SCHEDULER 资源的状态无关，但是可以阻止任务的重新调度。

2.5.4 同步机制的一般问题和解决方案

2.5.4.1 问题

（1）优先级反转

常见同步机制（如使用信号量）的典型问题与优先级反转有关。

优先级反转指高优先级任务需要等待低优先级任务释放资源，而低优先级任务又正在等待中等优先级任务的现象。此时高优先级任务和中等优先级任务之间没有任何共享资源但执行顺序却发生了倒置，这种情况称为优先级反转，而高优先级任务因为等待低优先级任务释放资源而阻塞的情况则不称为优先级反转。

图 10 说明了两个任务对信号量的一般访问的次序（完全抢占式任务，任务 1 为高优先级任务）。任务 T4 优先级较低，占有信号量 S1。T1 抢占 T4，请求同一个信号量。因为信号量 S1 已经被占有，T1 进入等待状态。这时低优先级的任务 T4 被中断，被优先级介于 T1 和 T4 之间的其他任务抢占。T1 只能等到所有低优先级任务都终止、信号量 S1 被重新释放之后才能得以执行。虽然 T2 和 T3 没有使用信号量 S1，它们的运行也延时了任务 1。

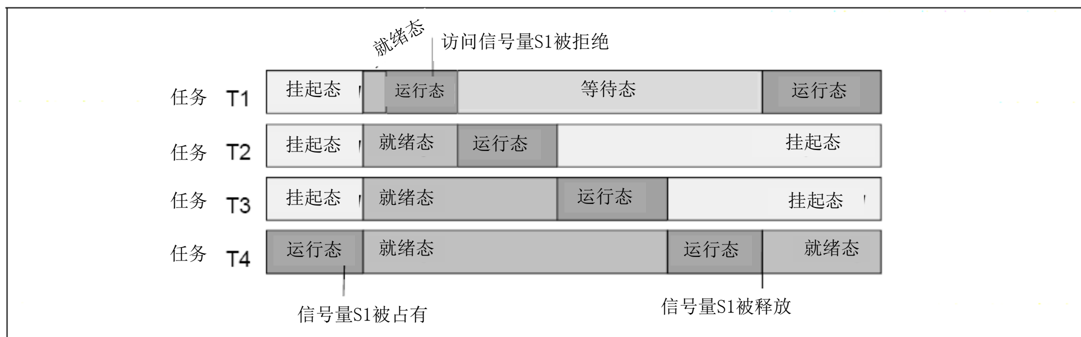

图 10 使用信号量时的优先级反转

（2）死锁

常见同步机制（如使用信号量）的另一个典型问题是死锁问题。所谓死锁，是指两个或两个以上的任务在执行过程中，因争夺资源而造成的一种互相等待的现象。在这种情况下，死锁意味着任务因为无限等待被锁死的资源而不会有执行的可能。

下面的例子就会造成死锁（见图 11）：

任务 T1 占有信号量 S1，然后因为需要等待一个事件等原因而停止执行。因此，低优先级的任务 T2 就转为运行态。它占有了信号量 S2。如果 T1 再次就绪，试图占有信号量 S2，它就再次进入等待态。如果此时 T2 试图占有信号量 S1，就会导致死锁。

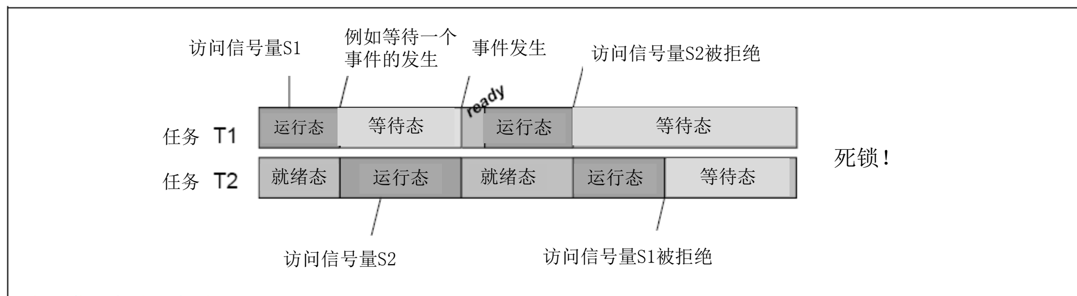

图 11 使用信号量时的死锁

2.5.4.2 解决方案

避免优先级反转的两种常见方法是：优先级继承协议（Priority Inheritance Protocol, PIP）和优先级上限协议（Priority Ceiling Protocol, PCP），其中优先级上限协议又被称为优先级天花板协议。UIP-Kernel 采用了 UIP-Kernel 优先级天花板协议来防止优先级反转的发生。

（1）UIP-Kernel 优先级天花板协议

为了避免优先级反转和死锁问题，UIP-Kernel 操作系统做出了如下要求；

- 在系统生成时，每个资源的天花板优先级都是静态指定好的。天花板优先级应大于等于所有会访问该资源的任务优先级的最大值或者与该资源相关的其他资源的优先级。同时，天花板优先级还应该小于所有不访问该资源的任务的最低优先级，这个最低优先级大于等于所有访问该资源的任务的最高优先级。
- 如果任务需要某一资源，但其优先级小于资源的天花板优先级，那么任

务的优先级就会被提升到该资源的天花板优先级。

- 如果任务释放了资源，那么该任务的优先级就会被动态地重新设置回获取资源前的优先级。

优先级天花板策略可能对优先级小于等于资源优先级的任务产生一定的时延。时延的极限值是资源被任何低优先级任务占有时间的最大值。

当任务由于优先级低于正在运行的任务而未进入运行态时，其他任务可能会与该任务占有相同的资源。如果被任务占有的资源被释放，那么可能会使得占有该资源的其他任务可以进入运行态。对抢占式任务来说这就是一个重新调度时刻。

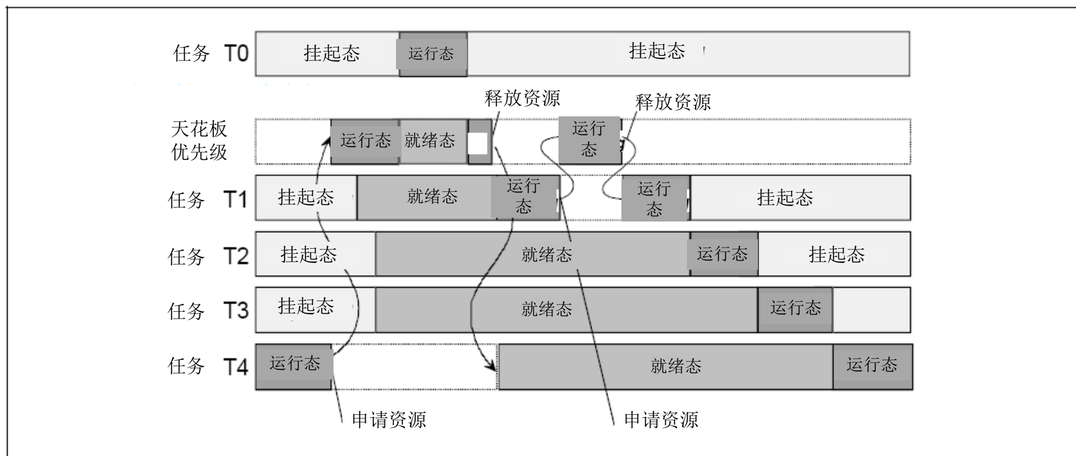

图 12 可抢占任务间带有优先级天花板的资源的分配

图 12 举例说明了优先级天花板的工作机制。任务 T0 优先级最高，T4 优先级最低。T1 和 T4 试图访问同一个资源。图 12 清楚地表明不会发生优先级的反转。高优先级任务 T1 等待的时间小于任务 T4 占有资源的时间。

（2）UIP-Kernel 优先级天花板协议的扩展（中断等级）

为了确定在中断中使用的资源的天花板优先级，需要给中断指定一个比所有任务优先级都高的虚拟优先级。

- 在系统生成时，每个资源的天花板优先级都是静态指定好的。天花板优先级应大于等于所有会访问该资源的任务优先级的最大值或者与该资源相关的其他资源的优先级。同时，天花板优先级还应该小于所有不访问该资源的任务的最低优先级，这个最低优先级大于等于所有访问该资源的任务的最高优先级。

- 如果任务需要某一资源，但其优先级小于资源的天花板优先级，那么任务的优先级就会被提升到该资源的天花板优先级。
- 如果任务释放了资源，那么该任务的优先级就会被动态地重新设置回获取资源前的优先级。

当任务由于优先级低于正在运行的任务而未进入运行态时，其他任务可能会与该任务占有相同的资源。如果被任务占有的资源被释放，那么可能会使得占有该资源的其他任务可以进入运行态。如果任务新的优先级不是中断的虚拟优先级，那么对抢占式任务来说这就是一个重新调度时刻。

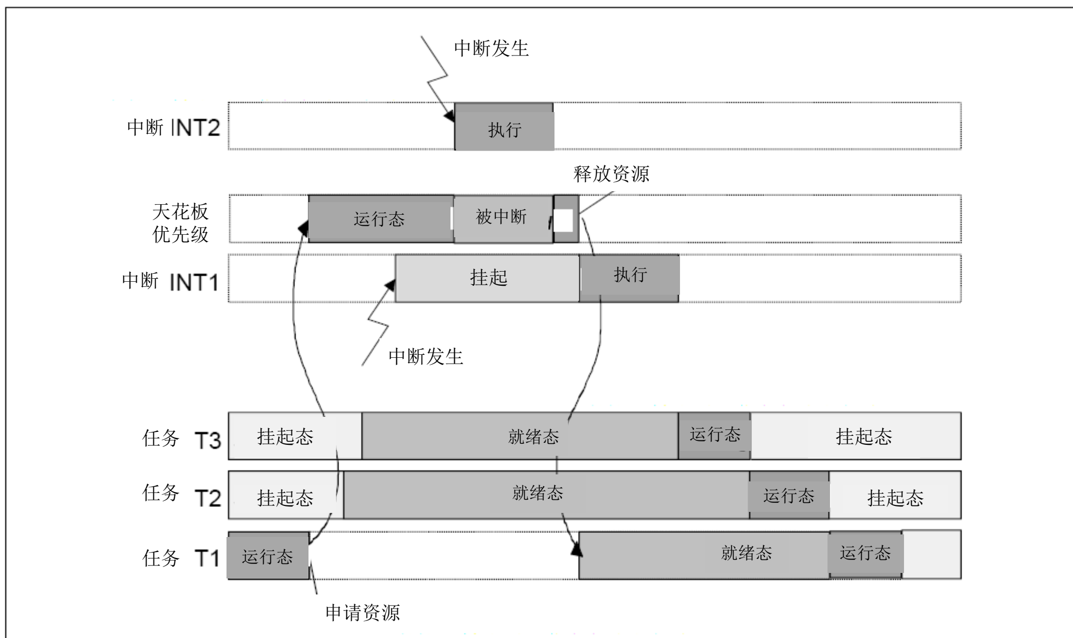

图 13 抢占式任务与中断服务程序间的带有天花板优先级的资源分配

图 13 所示的例子描述了下面的过程：

可抢占任务 T1 正在运行，并申请一个与中断服务程序 INT1 共享的资源。任务 T1 激活了优先级更高的 T2 和 T3。由于 UIP-Kernel 的优先级天花板协议，任务 T1 仍然继续运行。这时中断 INT1 发生。由于 UIP-Kernel 的优先级天花板协议，任务 T2 仍然继续运行，中断 INT1 被挂起。这时中断 INT2 发生。中断服务程序 INT2 中断了任务 T1，并开始执行。INT2 结束后，任务 T1 继续执行。之后 T1 释放了上述资源，中断服务程序 INT1 开始执行，任务 T1 被中断。INT1 执行完后，任务 T3 开始运行。T3 终止后 T2 开始运行。T2 终止后，任务 T1 继续执行。

图 14 则描述了如下的过程：

可抢占任务 T1 正在运行，此时中断 INT1 发生。任务 T1 被中断，中断服务程序 INT1 开始执行。中断服务程序 INT1 请求一个与 INT2 共享的资源，这时更高级别的中断 INT2 发生。由于 UIP-Kernel 的优先级天花板协议，INT1 仍然执行，INT2 被挂起。此时中断 ITN3 发生。由于 INT3 的优先级比 INT1 高，所以 INT3 中断了 INT1，开始执行。INT3 激活了任务 T2。INT3 结束后，INT1 继续执行。INT1 释放了上述资源后，由于 INT2 的优先级比 INT1 高，INT2 开始执行。INT2 结束后 INT1 继续执行。INT1 结束后，由于任务 T2 优先级比任务 T1 高，T2 开始运行，此时 T1 处于就绪态。当 T2 终止后 T1 继续执行。

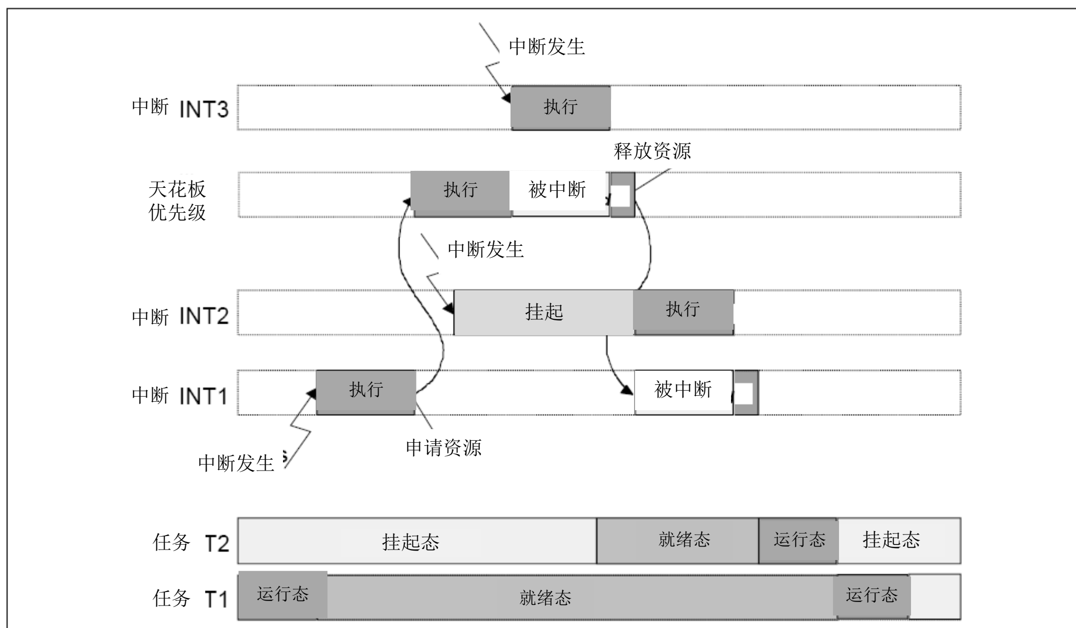

图 14 中断服务程序间带有优先级天花板的资源分配

2.5.5 内部资源

内部资源指用户不可见的资源。因此系统函数 GetResource 和 ReleaseResource 对内部资源无效。它们严格地由一组明确定义的系统函数来管理。除此之外，内部资源与标准的资源完全相同（如优先级天花板策略等）。

任务对内部资源的使用也有限制。在系统生成时，一个任务最多只能分配一个内部资源。如果内部资源被分配给了一个任务，那么：

- 当任务进入运行态时自动占有该资源，除非它已经占有。因此，任务的优先级会自动地改变为资源的天花板优先级。
- 在实现时可能对此进行优化，如当需要重调度时，只在系统服务 Schedule

内释放或获取内部资源。

不可抢占式任务是特殊的任务组，其内部资源的优先级与 RES_SCHEDULER 相同。内部资源可以用在任何场合，当有必要避免任务组内不需要的重调度时，系统可以定义多个任务组（也就是说多个内部资源）。

对系统调用的一些限制（不能在资源被占有时调用）不适用于内部资源，因为对内部资源的处理是在这些调用内部进行的。但是，在内部资源被释放之前必须遵循“LIFO 原则”释放所有标准的资源。

多个被分配了相同内部资源的任务涵盖了一个优先级范围。在这个优先级范围内，可能存在不使用这个内部资源的任务。由具体应用来决定是否需要这样。

2.6 警报

UIP-Kernel 操作系统为处理重复发生的事件提供了服务，例如以固定间隔提供中断的定时器，或者在凸轮轴或机轴的角度匀速变化的时候产生中断的译码器，或者其他常规应用的特定触发器。

在处理这些事件时，UIP-Kernel 操作系统采用了“两级”的概念。重复发生的事件（源）由具体实现中的记数器来注册。基于记数器，操作系统软件为应用软件提供报警机制。

2.6.1 计数器

计数器由一个计数值（以“tick”为单位）以及一些特殊的常数来表示。

UIP-Kernel 操作系统没有提供标准的 API 函数直接对计数器进行操作。

UIP-Kernel 操作系统负责当计数器计数值满时对警报进行必要的操作和管理。

在 UIP-Kernel 操作系统中，一个硬件或软件定时器至少可以提供一个计数器。

2.6.2 警报管理

当警报到达时，UIP-Kernel 操作系统会提供诸如激活任务、设置事件或者调用报警回调程序等服务。报警回调程序是由具体应用提供的一个函数。

当计数器达到预先定义的计数值时，警报到达。这个计数器值可能是实际计数值的相对值（相对报警）或者一个绝对值（绝对报警）。例如，当达到某一特定的角度或收到一个消息时，计时器都会产生中断，从而警报到达。

报警有单次报警和循环报警两类。除此之外，操作系统还提供了取消报警和获取报警的当前状态的服务。

一个计数器可以关联多个警报。

警报在系统生成时静态的分配给：

- 计数器
- 任务或报警回调程序

根据不同的配置，当警报到达时，调用上述报警回调程序、或者激活上述任务或者设置该任务的某一事件。报警回调程序在 2 型中断禁止后才运行。警报到

达时的任务激活和事件设置与普通的激活和设置相同。

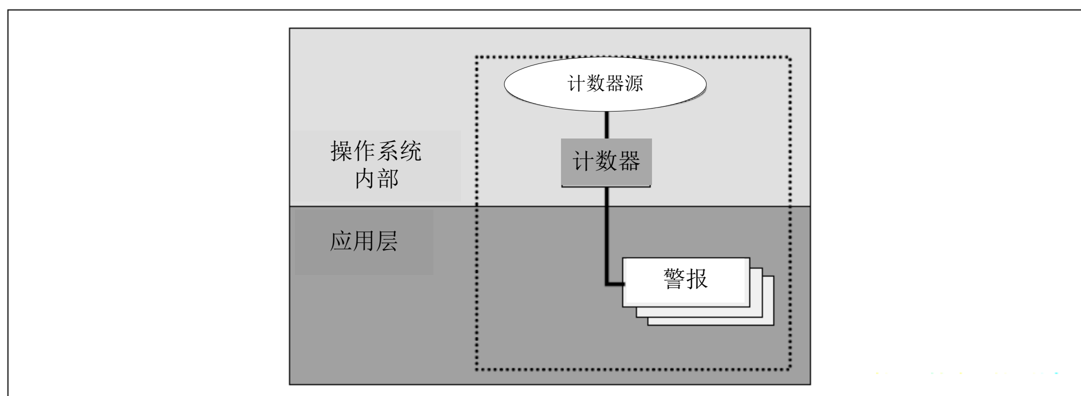

图 15 警报管理的分层模型

计数器和警报都是静态定义的。同时警报与计数器之间如何对应、警报到达时如何反应也是静态定义的。

警报到达时的计数器值以及循环报警的周期则是动态参数。

2.6.3 报警回调函数

报警回调函数可以既没有参数也没有返回值。

报警回调函数格式：

```
ALARMCALLBACK(AlarmCallbackRoutineName);
```

报警回调函数的例子：

```
ALARMCALLBACK(BrakePedalStroke)
{
/* do application processing */
}
```

报警回调函数的处理等级与调度程序或者 ISR 使用等级相同，由具体实现来决定。

2.7 错误处理、跟踪与调试

2.7.1 回调程序

UIP-Kernel 操作系统提供系统特殊的回调程序，允许在操作系统内部处理中调用用户自定义的动作。

回调程序：

- 在特殊的上下文中由操作系统调用，其上下文由操作系统的具体实现决定
- 比任何任务的优先级都高
- 不会被 2 型中断打断
- 是操作系统的一部分
- 由用户定义的函数来实现
- 函数接口是标准的，但函数实现（环境以及程序本身的行为）并不是标准化的，因此，通常回调程序不可移植
- 只允许使用一部分 API 函数

在操作系统中，回调程序可能用在：

- 系统启动

对应的回调程序（StartHook）在操作系统启动之后、调度程序启动之前调用。

- 系统关闭

对应的回调程序（ShutdownHook）在应用程序请求关闭系统或者系统出现严重错误时调用。

- 跟踪、应用中的调试或者用户定义的扩展的上下文切换
- 错误处理

UIP-Kernel 的每个实现都对回调程序的规范给出了描述。

如果应用软件在回调程序中调用了非法的 API 函数，那么可能会出现未知的结果。出现错误时，UIP-Kernel 返回一个在实现中定义过的错误代码。

大部分的操作系统服务都不能应用在回调函数中，这样的限制对降低系统的复杂度是很有必要的。

2.7.2 错误处理

概述

错误服务是为了用于对 UIP-Kernel 操作系统中的暂时性或不变性的错误进行处理而提供的。它的基本框架是预先定义好的，但是必须由用户来完成。有错误分散处理和集中处理两种有效的方法供用户选择。

两种不同的错误的区别在于：

- **应用错误**

操作系统不能正确执行请求的服务，但认为内部数据是正确的。

在这种情况下，调用集中错误处理。同时，操作系统通过分散处理的状态信息将错误返回。用户根据发生错误的类型决定如何处理。

- **致命错误**

操作系统任务内部数据出现错误。

在这种情况下，操作系统调用集中的系统关闭。

所有这些错误服务都分配了一个参数，用来表示错误类型。

UIP-Kernel 操作系统提供了两个级别的错误检查：标准状态和扩展状态。在标准状态的版本下，如果一个任务被激活，可能会返回“E_OK”或者“过多的任务激活”。在扩展状态的版本下，还可能会返回“任务合法”或者“任务仍然占有资源”等。目标软件在执行时不会返回这些扩展的返回值。也就是说操作系统的运行时版本不截获这些错误。

UIP-Kernel API 函数的返回值优先级要高于其输出参数。如果一个 API 函数返回错误，那么其输出参数是不确定的值。

错误回调程序

如果一个系统服务返回的 StatusType 值不等于 E_OK，错误回调程序（ErrorHook）就会被调用。ErrorHook 函数内部调用的系统服务将不再会调用 ErrorHook 错误回调程序（也就是说，不会发生错误回调的递归）。从 ErrorHook 中调用系统服务可能发生的任何错误都只能通过返回值来发现。

当任务被激活或者事件被设置时如果发生错误。例如发生告警或者消息到达，也会调用 ErrorHook。

错误管理

为了在 ErrorHook 中更有效地进行错误管理，用户可以获取附加信息。下图总结了错误处理的逻辑构架。

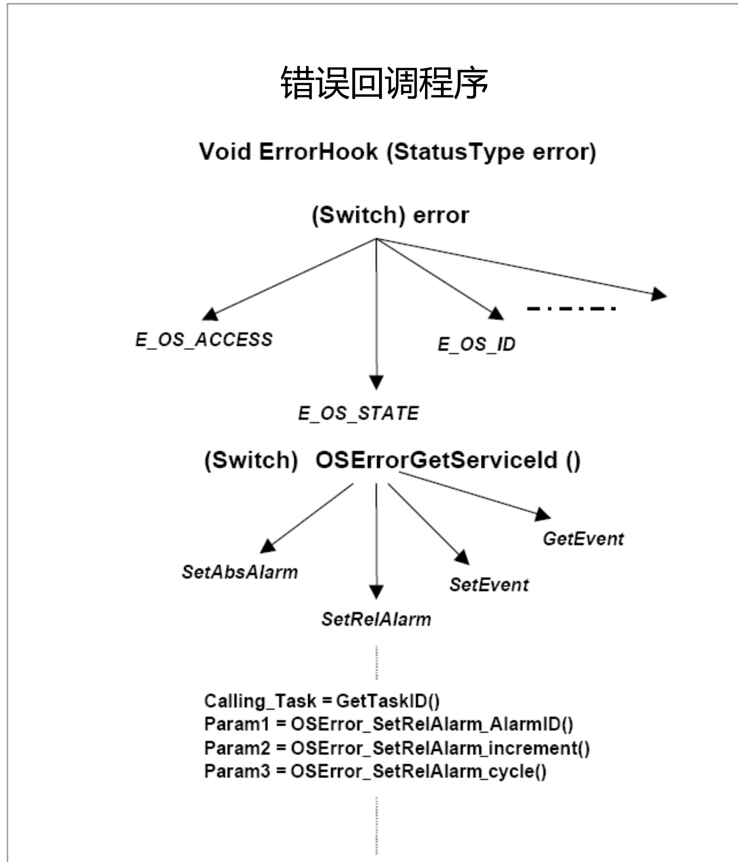

图 16 集中错误处理示例（扩展状态）

宏 OSErrorGetServiceId()提供发生错误的服务的标识，这个标识的类型是 OSServiceIdType，其值可能是 OSServiceId_xxxx，这里，xxxx 代表系统服务的名称。OSErrorGetServiceId 的实现是强制的。如果提供了调用 ErrorHook 的系统服务的参数，那么访问该参数的宏的命名方式为：OSError_Name1_Name2，其中：

- Name1：系统服务的名字
- Name2：该参数在 UIP-Kernel 操作系统规范中的正式名称

例如，获取 SetRelAlarm 各参数的宏分别为：

- OSError_SetRelAlarm_AlarmID()
- OSError_SetRelAlarm_increment()
- OSError_SetRelAlarm_cycle()

如果系统服务的第一个参数是一个对象标识，那么获取该参数的宏是强制的。

2.7.3 系统启动

处理器复位后的初始化工作是由具体实现来完成的，但是 UIP-Kernel 操作系统也提供标准化的初始化方式的支持。

UIP-Kernel 操作系统不强制应用软件定义特殊的在操作系统初始化后启动的任务，但它允许用户在系统生成时定义自动运行的任务或者自动运行的警报。

CPU 复位后会执行与硬件相关的应用软件（无操作系统）。当检测到应用模式时，不可移植的代码部分就结束了。应用的可移植代码部分就开始于调用启动操作系统的函数（StartOS），应用模式作为它的一个参数。操作系统初始化后（调度程序尚未运行），StartOS 调用启动回调程序 StartupHook，用户可以把他所有与操作系统有关的初始化代码都放在这个函数中。为了根据已经启动的应用模式组织 StartupHook 中的初始化代码，系统提供了名为 GetActiveApplicationMode 的函数。从启动回调函数返回后，操作系统进行中断的使能，并启动调度程序。之后，系统就开始运行，执行用户任务。具体步骤如图 17 所示。

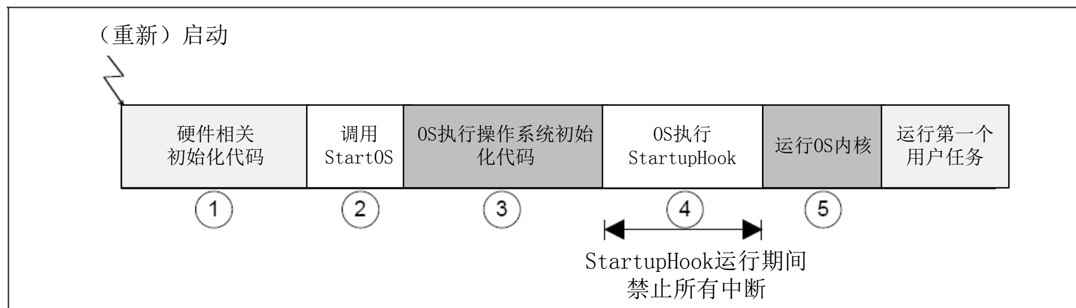

图 17 系统启动

图 17 的各部分介绍如下：

（1） 重新启动后，用户可以执行（不可移植的）硬件相关的代码。类型 2 中断直到第 5 个阶段才允许运行。当检测到用户模式时，不可移植的部分结束。

（2） 调用 StartOS 函数，应用模式作为该函数的一个参数。该调用启动操作系统。

（3） 操作系统运行内部启动代码。

（4） 调用回调程序 StartupHook，其中包含了用户的初始化代码。在此回调程序执行期间，禁止所有的用户中断。

（5） 操作系统使能用户中断、启动调度程序，启动当前应用软件定义的自动运行的任务和警报。优先级相同的自动运行任务激活的次序是不确定的。自动运行的任务启动早于自动运行的警报。

2.7.4 系统关闭

UIP-Kernel 操作系统定义了关闭操作系统的服务 ShutdownOS，该服务可以由应用程序申请或者发生致命错误时由操作系统调用。

当 ShutdownOS 函数被调用时，系统会调用 ShutdownHook 回调程序然后关闭系统。通常用户可以在 ShutdownHook 中自由定义任何的系统行为，如不从该程序中返回。

2.7.5 调试

在任务上下文切换时会调用 PreTaskHook 和 PostTaskHook 两个回调程序，可能用于调试或者时间测量（包括上下文切换时间）。因此每次当老的任务离开运行态之前都要直接调用 PostTaskHook；每次当新任务进入运行状态之后都要直接调用 PreTaskHook。由于任务仍处于运行态或者已经进入运行态，GetTaskId 不会返回 INVALID_TASK。

当一个任务正在运行时 ShutdownOS 被调用，那么 ShutdownOS 可能调用 PostTaskHook 也可能不调用。如果 PostTaskHook 在 ShutdownHook 之前或者之后调用，那么它的行为是未知的。

三、系统服务接口

3.1 系统服务描述

3.1.1 系统对象的定义

在 UIP-Kernel 操作系统内部，所有的系统对象都必须由用户静态地定义。操作系统供应商提供操作系统对象的定义。对象的实际创建（唯一的名称和特定的特征）是在系统生成阶段。应用软件源代码中的声明是对这些操作系统对象的外部引用。没有系统服务可以动态地创建系统对象。使用一个系统对象时通过声明可以找到其创建地址。在系统服务中名称是系统对象的标识，通常这些名称的作用域类似于 C 语言中的外部变量。

3.1.2 约定

3.1.2.1 调用的类型

系统服务的接口是符合 ISO/ANSI-C 的，其实现通常是一个函数的调用，但根据具体实现的需求也可能不同，例如可能是 C 预处理器的宏。用户不能定义自己的实现方式。

3.1.2.2 合法的调用

系统服务由任务、中断服务程序、回调程序和报警回调函数调用。根据系统服务的不同，在使用时会有些限制。

表 3 列出了所有的系统服务以及它们在何种情况下可以被调用（√）。

表 3 API 服务限制

服务名称	任务	1 型 ISR	2 型 ISR	ErrorHook	PreTaskHook	PostTaskHok	StartupHook	ShutdownHook	alarm-callback
ActiveTask	√		√						
TerminateTask	√								
ChainTask	√								
Schedule	√								
GetTaskID	√		√	√	√	√			

DisableAllInterrupts	√	√	√						
EnableAllInterrupts	√	√	√						
SuspendAllInterrupts	√	√	√	√	√	√			√
ResumeAllInterrupts	√	√	√	√	√	√			√
SuspendOSInterrupts	√	√	√						
ResumeOSInterrputs	√	√	√						
GetResource	√		√						
ReleaseResource	√		√						
SetEvent	√		√						
ClearEvent	√								
GetEvent	√		√	√	√	√			
WaitEvent	√								
GetAlarmBase	√		√	√	√	√			
GetAlarm	√		√	√	√	√			
SetRelAlarm	√		√						
CancelAlarm	√		√						
GetActiveApplicationMod	√		√	√	√	√	√	√	
StartOS									
ShutdownOS	√		√	√			√		

3.1.2.3 错误的特性

为了保证系统的高效快速，UIP-Kernel 操作系统并不检测所有的错误。如果应用软件错误地使用了操作系统服务，可能会导致不确定的结果。

大部分的系统服务会返回一个状态给用户。如果可以不受任何限制地执行该服务，那么返回值为 E_OK。如果系统识别到影响服务执行的异常，返回不同值。

返回非 E_OK 的状态值可能是警报等信息而不被认为发生了错误。一个例子就是系统服务 CancelAlarm，如果返回非 E_OK 标明要取消的警报已经到达，因此用户程序就被告知可能发生了一个不需要的任务激活。检测“警告”是系统服务的一部分。

如果可以在运行时之前排除错误，那么运行时版本就可以省去对这些错误的检查。如果返回值只可能是 E_OK，那么实现时可以没有返回值。

在每个系统服务单个的描述中列出了它所有返回值。在“标准”状态和“扩

展”状态下，返回值不同。“标准”版本遵循上述已调试应用系统的要求。根据上述描述，标准模式下返回非 E_OK 表示“警告”。“扩展”状态一般用于测试尚未完全调试的应用，与标准版本相比，扩展版本还包含了扩展错误的检查。

操作系统内错误检查的次序是不确定的。一旦多个错误同时发生，由具体实现决定返回给应用哪个状态。

如果定义了 ErrorHook，那么在发生应用错误的情况下，操作系统会调用该回调程序。ErrorHook 的目的就是使能集中处理状态信息。

只有当返回非 E_OK 时才调用 ErrorHook。

为了实现对 ErrorHook 的管理，需要区分标准模式、标准状态管理和扩展模式、扩展模式管理。

系统将附加信息传递给 ErrorHook 回调例程。考虑到性能以及堆栈消耗，用一个全局的结构体来表示最后一条错误的完整信息，执行时根据给定的服务和具体实现的限制填充这个全局数据结构。为了实现的效率和适应性起见，这个错误管理数据结构的格式并不是预先定义好的。但是，为了实现 ErrorHook 代码的可移植性，定义了标准化的宏用以访问不同的参数。

在发生致命错误的时候，系统服务不会返回任何值给应用软件，但会调用 ShutdownOS。举例来说，某个系统服务出现了一个未被检测到的错误参数，导致系统矛盾。这时传递给 ShutdounOS 的是一个由具体实现定义的系统错误代码。系统错误代码占用了一个范围内的数，不会与操作系统服务的状态值冲突。

函数 ShutdownOS 的行为是与具体实现有关的，可能停止应用或者报告异常。应用软件自己也可以访问 ShutdownOS 来关闭操作系统。

处理无归属的错误（如“非法指令”）时，也推荐调用 ShutdownOS。因为需要硬件支持，所以这不是强制的。

3.2 操作系统服务规范

描述的结构

操作系统服务被划分为不同的逻辑组，如任务管理、中断管理等，并对其所有的服务进行了描述。

每个逻辑组的描述都是以数据类型的定义开始的，接下来是组成成员及系统服务，最后是常量及一些附加的约定。

组成成员

对组成成员的描述包括以下几项：

语法：　　类似 C 语言语法的接口

参数（入口）：　所有入口参数的列表

描述：　　对组成成员的解释

特性：　　对使用时的限制的解释

服务描述

对服务的描述包括以下几项：

语法：　　类似 C 语言语法的接口

参数（入口）；　所有入口参数的列表

参数（出口）：　所有出口参数的列表

描述：　　对操作系统服务功能的描述

特性：　　对操作系统服务使用时的限制的解释

状态：　　所有可能返回值的列表

标准：　操作系统标准版本下所有返回值的列表。特殊情况：服务不返回任何值

扩展：　操作系统扩展版本下所有附加返回值的列表

操作系统服务规范中数据类型采用如下的命名方式：

…Type：描述单个数据的值（包括指针）

…RefType：描述指向…Type 的指针（用于引用调用）

3.2.1 常用数据类型

StausType

这个数据结构用于 API 服务提供的所有状态信息。

命名规则：所有 API 服务的错误状态都以 E_开头，操作系统保留的状态以 E_OS_开头。正常返回的是 E_OK，其值为 0。除此之外还定义了下列错误值：

API 服务的所有错误值：

- E_OS_ACCESS = 1,
- E_OS_CALLEVEL = 2,
- E_OS_ID = 3,
- E_OS_LIMT = 4,
- E_OS_NOFUNC = 5,
- E_OS_RESOURCE = 6,
- E_OS_STATE = 7,
- E_OS_VALUE = 8

如果返回值只可能是 E_OK，那么在具体实现时可以选择不返回状态。这在单个服务的描述中没有专门列出。

操作系统内部错误：

这些错误都是与具体实现有关的，属于不可移植的部分。这些错误的名字与上述 API 服务的错误属于同一个命名空间（name-space）。实现时必须保证取值范围不能重叠。

为了体现使用中的不同，内部错误的名字是以 E_OS_SYS_开头的。

例如：

- E_OS_SYS_STACK
- E_OS_SYS_PARITY
- ….

UIP-Kernel 操作系统内部错误的名字、取值范围与其他 UIP-Kernel 服务（如通信与网管）的名字、取值范围以及 API 的错误取值范围不重叠。

3.2.2 任务管理

3.2.2.1 数据类型

TaskType

该数据类型标识了一个任务

TaskRefType

该数据类型指向一个可变的 TaskType

TaskStateType

该数据类型标识了一个任务的状态

TaskStateRefType

该数据类型指向一个可变的 TaskStateType 数据类型

3.2.2.2 组成成员

3.2.2.2.1 DeclareTask

语法： DeclareTask(<TaskIdentifier>)

参数（入口）：

TaskIdentifier 任务标识

描述： DeclareTask 是对任务的外部声明。该服务的作用和使用方法与声明外部变量相似

特性： -

3.2.2.3 系统服务

3.2.2.3.1 ActiveTask

语法： StatusType ActiveTask (TaskType <TaskID>)

参数（入口）：

TaskID 要激活的任务

参数（出口）： 无

描述： 任务<TaskID>将从挂起态转为就绪态。操作系统保证从任务代码的第一条语句开始执行。

特性： 该服务可以由中断级别调用，也可以由任务级别调用。

调用 ActiveTask 后如何进行任务的重新调度取决于调用该服务的位置（ISR、不可抢占任务还是可抢占任务）

如果返回 E_OS_LIMIT，忽略此次激活。

如果一个扩展任务从挂起态转为就绪态，那么它的所有事件将被清空。

状态：

标准：无错误，E_OK

对<TaskID>有多重激活，E_OS_LIMIT

扩展：任务<TaskID>非法，E_OS_ID

3.2.2.3.2 TerminateTask

语法： StatusType TerminateTask (void)

参数（入口）： 无

参数（出口）： 无

描述： 该服务终止正在调用的任务，任务由运行态转为挂起态。

特性： 分配给正在调用任务的内部资源被自动释放。其他由该任务占用的资源应该在调用 TerminateTask 之前完成释放。如果在标准状态下仍有资源被该任务占用，会出现不确定的结果。

如果调用成功，TerminateTask 不会返回到调用它的级别，因此也不会有状态值

如果使用的是扩展版本，那么当发生错误时服务返回，并提供一个与应用有关的状态值。

如果服务 TerminateTask 调用成功，会强制进行任务的重新调度。

不调用 TerminateTask 或者 ChainTask 而结束一个任务是严格禁止的，这样可能会使系统处于不确定的状态。

状态：

标准：不返回到调用程序

扩展：如果仍有资源被调用任务占有，E_OS_RESOURCE

由中断程序调用，E_OS_CALLEVEL

3.2.2.3.3 ChainTask

语法： StatusType ChainTask (TaskType <TaskID>)

参数（入口）：

TaskID 等待激活的下一个任务

参数（出口）： 无

描述： 该服务终止正在调用的任务，终止后激活下一个任务<TaskID>。使用这个服务可以保证正在运行的任务终止后可以最快的速度开始启动下一个任务。

特性： 如果下一个任务与当前任务相同，也不会导致重复请求。任务不会进入挂起态，但会立刻再次进入就绪态。

即使下一个任务与当前任务相同，分配给当前任务的内部资源也会自动释放。由当前任务占有的其他资源应该在调用 ChainTask 之前完成释放。如果在标准状态下仍有资源被该任务占用，会出现不确定的结果。

如果调用成功，ChainTask 不会返回到调用级别，因此也不会有状态值。

如果发生错误，服务返回到调用任务，并提供一个与应用相关的状态值。

如果 ChainTask 被正确调用，会强制系统进行任务的重新调度。

不调用 TerminateTask 或者 ChainTask 而结束一个任务是严格禁止的，这样可能会使系统处于不确定的状态。

如果返回 E_OS_LIMIT，忽略此次激活。

如果一个扩展任务从挂起态转为就绪态，那么它的所有事件将被清空。

状态：

标准： 不返回到调用程序

对<TaskID>有多重激活，E_OS_LIMIT

扩展： 如果<TaskID>非法，E_OS_ID

如果仍有资源被调用任务占有，E_OS_RESOURCE

由中断程序调用，E_OS_CALLEVEL

3.2.2.3.4 Schedule

语法： StatusType Schedule (void)

参数（入口）： 无

参数（出口）： 无

描述： 如果高优先级任务就绪，那么释放当前任务的内部资源，转入就绪态，保存其上下文，开始执行高优先级任务。否则当前任务继续执行。

特性： 该服务只针对在系统生成时分配了内部资源的任务。对于这些任务，Schedule 将处理器分配给优先级大于等于内部资源天花板优先级或者优先级高于调用任务的任务。当任务从 Schedule 返回时，内部资源被重新占有。

该服务对没有分配内部资源的任务（可抢占式任务）没有影响。

状态：

标准： 无错误，E_OK

扩展： 由中断程序调用，E_OS_CALLEVEL

调用任务仍占有资源，E_OS_RESOURCE

3.2.2.3.5 GetTaskID

语法： StatusType GetTaskID (TaskRefType <TaskID>)

参数（入口）： 无

参数（出口）：

TaskID 指向当前正在运行的任务

描述： GetTaskID 返回正在运行的任务的 TaskID

特性： 允许在任务、中断服务程序以及多个回调程序中使用

该服务设计用于库函数以及回调程序。

如果 <TaskID> 没有值（没有任务正在运行），服务返回 INVALID_TASK 作为 TaskType。

状态：

标准： 无错误，E_OK

扩展： 无错误，E_OK

3.2.2.3.6 GetTaskState

语法： StatusType GetTaskState (TaskType <TaskID>, TaskStateRefType <State>)

参数（入口）：

TaskID 任务

参数（出口）：

State 指向任务<TaskID>的状态

描述： 返回指定任务在调用 GetTaskState 时的状态（运行态、就绪态、等待态、挂起态）

特性： 该服务可由中断服务程序、任务以及一些回调程序调用。

如在完全抢占式系统的任务中调用该服务，那么在取状态值时调用结果可能已经是错误的。

当该服务由一个被多次激活的任务调用时，只要有任务的实例在运行，状态值都为运行态。

状态：

标准： 无错误，E_OK

扩展： 任务<TaskID>非法，E_OS_ID

3.2.2.4 常量

RUNNING TaskStateType 类型的常量，表示任务运行态

WAITING TaskStateType 类型的常量，表示任务等待态

READY TaskStateType 类型的常量，表示任务就绪态

SUSPEND TaskStateType 类型的常量，表示任务挂起态

INVALID_TASK TaskType 类型的常量，表示一个未定义的任务

3.2.2.5 命名规则

操作系统应该根据作为任务标识的任务名称分配相应的任务入口地址。通过入口地址，操作系统才能调用任务。

在应用中，任务的定义形式如下：

```
TASK ( TaskName )
{
}
```

宏 TASK 可以使用“任务标识”和“任务函数的名称”。

任务标识在系统生成阶段由 TaskName 产生。

3.2.3 中断管理

3.2.3.1 数据类型

没有为 UIP-Kernel 中断管理定义的特殊数据类型。

3.2.3.2 系统服务

3.2.3.2.1 EnableAllInterrupts

语法： void EnableAllInterrupts (void)

参数（入口）： 无

参数（出口）： 无

描述： 该服务恢复 DisableAllInterrupts 保存的状态

特性： 该服务可以由 1 型中断、2 型中断和任务调用，但不能由回调程序调用。

该服务与 DisableAllInterrupts 配对，在调用该服务之前必须已经调用 DisableAllInterrupts，其目的是完成一段重要的代码，在这个关键部分中不能调用任何 API 服务。

实现时应将该服务适配到目标硬件系统以提供最小的开销。通常，该服务可以使能对 CPU 中断的识别。

状态：

标准： 无

扩展： 无

3.2.3.2.2 DisableAllInterrupts

语法： void DisableAllInterrputs (void)

参数（入口）： 无

参数（出口）： 无

描述： 该服务禁止所有硬件上可禁止的中断。禁止之前的状态会被保存，用于调用 EnableAllIterrupts 的状态的恢复。

特性： 该服务可以由 1 型中断、2 型中断和任务调用，但不能由回调程序调用。

该服务用于开始一段重要的代码，这部分代码通过调用 EnableAllInterrupts 来完成。在这段重要的代码中不能调用任何

API 服务。

实现时应将该服务适配到目标硬件系统以提供最小的开销。通常，该服务关闭了对 CPU 中断的识别。如果重要代码如库中需要嵌套，应该使用 SuspendOSInterrupts/ResumeOSInterrupts 或者 SuspendAllInterrupts/ResumeAllInterrupts。

状态：

标准： 无

扩展： 无

3.2.3.2.3 ResumeAllInterrupts

语法： void ResumeAllIterrupts (void)

参数（入口）： 无

参数（出口）： 无

描述： 该服务恢复 SuspendAllInterrupts 保存的所有中断的状态。

特性： 该服务可以由 1 型中断、2 型中断和任务调用，但不能由回调程序调用。

该服务与 SuspendAllInterrupts 配对使用，在调用之前必须已经调用 SuspendAllInterrupts，其目的是完成一段重要的代码。在这段重要代码中，除了 SuspendOSInterrupts/ResumeOSInterrupts 和 SuspendAllInterrupts/ResumeAllInterrupts 两对函数外不能调用其他任何 API 服务。

实现时应将该服务适配到目标硬件系统以提供最小的开销。

SuspendAllInterrupts/ResumeAllInterrupts 可以嵌套使用。在嵌套使用 SuspendAllInterrupts 和 ResumeAllInterrupts 时，第一次调用 SuspendAllInterrupts 时保存的中断识别状态由最后一个 ResumeAllInterrupts 恢复。

状态：

标准： 无

扩展： 无

3.2.3.2.4 SuspendAllInterrupts

语法： void SuspendAllInterrupts (void)

参数（入口）：　无

参数（出口）：　无

描述：　该服务保存所有中断的识别状态，并禁止所有硬件可禁止的中断。

特性：　该服务可以由 1 型中断、2 型中断和任务调用，但不能由回调程序调用。

该服务用于保护一段重要的代码不被任何中断打断。这段代码结束时应调用 ResumeAllInterrupts 服务。在这段重要代码中，除了 SuspendOSInterrupts/ResumeOSInterrupts 和 SuspendAllInterrupts/ResumeAllInterrupts 两对函数外不能调用其他任何 API 服务。

实现时应将该服务适配到目标硬件系统以提供最小的开销。

状态：

标准：　无

扩展：　无

3.2.3.2.5 ResumeOSInterrupts

语法：　void ResumeOSInterrupts (void)

参数（入口）：　无

参数（出口）：　无

描述：　该服务恢复由 SuspendOSInterrupts 保存的中断识别状态

特性：　该服务可以由 1 型中断、2 型中断和任务调用，但不能由回调程序调用。

该服务与 SuspendOSInterrupts 配对使用，在调用之前必须已经调用 SuspendOSInterrupts，其目的是完成一段重要的代码。在这段重要代码中，除了 SuspendOSInterrupts/ResumeOSInterrupts 和 SuspendAllInterrupts/ResumeAllInterrupts 两对函数外不能调用其他任何 API 服务。

实现时应将该服务适配到目标硬件系统以提供最小的开销。

SuspendOSInterrupts/ ResumeOSInterrupts 可以嵌套使用。在嵌套使用 SuspendOSInterrupts 和 ResumeOSInterrupts 时，第一次调用

SuspendOSInterrupts 时保存的中断识别状态由最后一个 ResumeOSInterrupts 恢复。

状态：

标准：无

扩展：无

3.2.3.2.6 SuspendOSInterrupts

语法： void SuspendOSInterrupts (void)

参数（入口）： 无

参数（出口）： 无

描述： 该服务保存 2 型中断的识别状态，并关闭对这些中断的识别

特性： 该服务可以由 1 型中断、2 型中断和任务调用，但不能由回调程序调用。

该服务用于保护一段重要的代码不被任何中断打断。这段代码结束时应调用 ResumeOSInterrupts 服务。在这段重要代码中，除了 SuspendOSInterrupts/ResumeOSInterrupts 和 SuspendAllInterrupts/ResumeAllInterrupts 两对函数外不能调用其他任何 API 服务。

实现时应将该服务适配到目标硬件系统以提供最小的开销。

该服务设计只是用来禁止 2 型中断。但是如果这样效率不高，可能会禁止更多的中断。

状态：

标准：无

扩展：无

3.2.3.3 命名规则

在实现时，2 型中断的中断服务程序的定义形式如下：

```
ISR ( FuncName )
{
}
```

关键字 ISR 在系统生成时生成，以在源代码中明确的区分函数和中断服务程序。

1 型中断的中断服务程序没有特定的命名规则，由具体实现来定义。

3.2.4 资源管理

3.2.4.1 数据类型

ResourceType

资源数据结构

3.2.4.2 组成成员

3.2.4.2.1 DeclareResource

语法： DeclareResource (< ResourceIdentifier>)

参数（入口）：

ResourceIdentifier 资源标识符

描述： DeclareResource 是资源的外部声明，其功能和使用方法与变量的外部声明类似。

特性： -

3.2.4.3 系统服务

3.2.4.3.1 GetResource

语法： StatusType GetResource (ResourceType <ResID>)

参数（入口）：

ResID 资源 ID

参数（出口）： 无

描述： 调用该服务是为了进入一段重要的代码，这段代码需要使用 <ResID> 指向的资源。离开重要代码段时通常要调用 ReleaseResource。

特性： 资源管理的 UIP-Kernel 优先级天花板协议。

只有当内层关键代码的执行完全在外层关键代码内部时，才允许进行资源的嵌套占有。同一个资源的嵌套使用也是禁止的。

推荐将一对 GetResource 和 ReleaseResource 的调用在同一个函数中进行。

在重要代码中，不能使用会导致不可抢占系统任务重新调度的服务（如 TerminateTask、ChainTask、Schedule 和 WaitEvent）。另外，

应该在中断服务程序结束之前离开重要代码段。

重要代码段通常都比较短。

该服务可由 ISR 和任务调用。

状态：

标准： 无错误，E_OK

扩展： 资源<ResID>非法，E_OS_ID

试图获取一个已被任务或 ISR 占用的资源，或者调用任务或中断程序静态分配的优先级高于计算出来的天花板优先级，E_OS_ACESS

3.2.4.3.2 ReleaseResource

语法： StatusType ReleaseResource(ResourceType <ResID>)

参数（入口）：

ResID 资源 ID

参数（出口）： 无

描述： ReleaseResource 与 GetResource 是配对使用的，用于离开一段分配了资源<ResID>的重要代码。

特性： 资源嵌套的条件见 GetResource 的特性。

状态：

标准： 无错误，E_OK

扩展： 资源<ResID>非法，E_OS_ID

试图释放一个没有被任何任务或 ISR 占用的资源，或者一个应该在之前释放的资源，E_OS_NOFUNC

试图释放一个天花板优先级低于调用任务或中断程序的静态优先级的资源，E_OS_ACCESS

3.2.4.4 常量

RES_SCHEDULER ResourceType 类型的常量

3.2.5 事件控制

3.2.5.1 数据类型

EventMaskType

事件屏蔽数据类型

EventMaskRefType

指向一个事件屏蔽

3.2.5.2 组成成员

3.2.5.2.1 DeclareEvent

语法： DeclareEvent (<EventIdentifier>)

参数（入口）：

EventIdentifier 事件标识

描述： DeclareEvent 用作事件的外部声明，其功能和作用与变量的外部声明类似。

特性： -

3.2.5.3 系统服务

3.2.5.3.1 SetEvent

语法： StatusType SetEvent (TaskType <TaskID> EventMaskType <Mask>)

参数（入口）：

TaskID 要设置一个或多个事件的任务。

Mask 要设置的事件的屏蔽值

参数（出口）： 无

描述： 该服务可由中断服务程序或任务调用，但不能由回调程序调用。根据事件屏蔽值<Mask>来设置任务<TaskID>的事件。如果任务正在等待<Mask>指定的至少一个事件，那么调用 SetEvent 将任务转为就绪态。

特性： 没有在事件屏蔽中设置的事件保持不变。

状态：

标准： 无错误，E_OK

扩展： 任务<TaskID>非法，E_OS_ID

指向的任务不是扩展任务，E_OS_ACCESS

事件不能被设置，因为任务处于挂起态，E_OS_STATE

3.2.5.3.2 ClearEvent

语法： StatusType ClearEvent(EventMaskType <Mask>)

参数（入口）：

Mask 要被清除的事件的屏蔽值

参数（出口）： 无

描述： 根据事件屏蔽<Mask>的值，扩展任务的相应事件被清除

特性： ClearEvent 只应用于拥有事件的扩展任务

状态：

标准： 无错误，E_OK

扩展： 调用者不是扩展任务，E_OS_ACCESS

由中断程序调用，E_OS_CALLEVEL

3.2.5.3.3 GetEvent

语法： StatusType GetEvent (TaskType <TaskID> EventMaskType <Mask>)

参数（入口）

TaskID 需要返回其事件屏蔽值的任务 ID

参数（出口）

Event：指向返回数据的内存区域

描述： 该服务返回任务<TaskID>所有事件位的当前状态，而不是任务等待的事件。

该服务可由中断服务程序、任务调用，但不能由回调程序调用。

任务<TaskID>任务屏蔽的当前状态被拷贝到<Event>。

特性： 指向的任务应为扩展任务

状态：

标准： 无错误，E_OK

扩展： <TaskID>指向的任务不是扩展任务，E_OS_ACCESS

<TaskID>指向的任务状态为挂起态，E_OS_STATE

3.2.5.3.4 WaitEvent

语法： StatusType WaitEvent (EventMaskType <Mask>)

参数（入口）：

Mask　等待的事件的屏蔽值

参数（出口）：　无

描述：　调用任务的状态被设为等待态，除非<Mask>中指定的事件至少有一个已经发生。

特性：　如果等待的条件发生，对该服务的调用会强制进行任务的重新调度。如果发生重新调度，那么当任务处于等待态时任务的内部资源会被释放。

状态：

标准：　无错误，E_OK

扩展：　调用任务不是扩展任务，E_OS_ACCESS

调用任务仍然占有资源，E_OS_RESOURCE

由中断程序调用，E_OS_CALLEVEL

3.2.6 消息

3.2.6.1 数据类型

MsgType

消息数据类型

3.2.6.2 系统服务

3.2.6.2.1 SendMessage

语法：　SendMessage(MsgType MsgID, AppDataRef DataRef)

参数（入口）：

MsgID　要发送的消息的 ID。

DataRef　要发送的消息的存放地址

参数（出口）：　无

描述：　根据 artMsgType 的值执行不同的操作。若 artMsgNotifyType = ART_COM_NOTIFICATION_CALLBACK，则在发送该消息时，自动执行(*artMsgCallBack)(VOID)指定的回调函数；

若 artMsgNotifyType = ART_COM_NOTIFICATION_FLAG，则在发送消息时将 artMsgFlag 的值设为 1；

若 artMsgNotifyType = ART_COM_NOTIFICATION_SETEVENT，则调用 SetEvent (artMsgNotifyTaskID, artMsgEvent))为任务 artMsgNotifyTaskID 设置事件 artMsgEvent；

若 artMsgNotifyType = ART_COM_NOTIFICATION_ACTIVATETASK，则调用 ActivateTask(artMsgNotifyTaskID)激活任务 artMsgNotifyTaskID。

特性：　　—

状态：

无错误，E_OK

消息<MsgID>非法，E_OS_ID

消息< artMsgStatus>= ART_MSG_LOCKED, ART_COM_LOCKED

3.2.6.2.2 ReceiveMessage

语法：　　ReceiveMessage(MsgType MsgID, AppDataRef DataRef)

参数（入口）：

MsgID　　要接收的消息的 ID。

DataRef　　接收消息的存放地址

参数（出口）：　无

描述：　接收< MsgID >指定的消息，并放入 DataRef 指定地址处。

特性：　　—

状态：

无错误，E_OK

消息<MsgID>非法，E_OS_ID

消息< artMsgStatus>= ART_MSG_LOCKED，ART_COM_LOCKED

若为 queued message，且消息缓冲区无数据，ART_COM_BUFFER_EMPTY

消息缓冲区溢出，ART_COM_OVERFLOW

3.2.6.2.3 GetMessageStatus

语法：　　StatusType GetMessageStatus(MsgType MsgID)

参数（入口）：

MsgID　　要获取状态的消息的 ID。

参数（出口）：

StatusType < MsgID >指定的消息的状态

无错误，返回 E_OK

消息<MsgID>非法，返回 E_OS_ID

消息< artMsgStatus>= ART_MSG_LOCKED，返回 ART_COM_LOCKED

若为 queued message，且消息缓冲区无数据，返回 ART_COM_BUFFER_EMPTY

若为 queued message，且消息缓冲区溢出，返回 ART_COM_OVERFLOW

描述： 返回< MsgID >指定的消息的状态。

特性： —

3.2.6.2.4 ReadFlag

语法： FlagType ReadFlag(MsgType MsgID)

参数（入口）：

MsgID 要获取标志的消息的 ID。

参数（出口）：

FlagType < MsgID >指定的消息的标志 artMsgFlag

描述： 返回< MsgID >指定的消息的标志。

特性： —

状态： 无

3.2.6.2.5 ResetFlag

语法： StatusType ARTResetFlag(MsgType MsgID)

参数（入口）：

MsgID 要重置标志的消息的 ID。

参数（出口）：

无

描述： 将< MsgID >指定的消息的 artMsgFlag 重置为 0。

特性： —

状态：

无错误，E_OK

3.2.7 警报

3.2.7.1 数据类型

TickType

该数据结构代表 tick 计数值

TickRefType

指向 TickType 的数据结构

AlarmBaseType

用于存储计数器特性的结构体，结构体的成员包括：

Maxallowedvzlue	tick 计数允许的最大值
Ticksperbase	到达某个重要的计数点的 tick 计数值
Mincycle	SetRelAlarm/SetAbsAlarm 的周期参数所允许的最小值（只适用于扩展状态的系统）。

结构体所有成员都是 TickType 类型的。

AlarmBaseRefType

指向 AlarmBaseTpe 的数据结构

AlarmType

该数据结构代表一个警报对象。

3.2.7.2 组成成员

3.2.7.2.1 DeclareAlarm

语法： DeclareAlarm(<AlarmIdentifier>)

参数（入口）：

AlarmIdentifier 警报的标识

描述： DeclareAlarm 是对一个警报的外部声明。

特性： -

3.2.7.3 系统服务

3.2.7.3.1 GetAlarmBase

语法： StatusType GetAlarmBase(AlarmType <AlarmID>,

AlarmBaseRefType <Info>)

参数（入口）：

AlarmID 警报 ID

参数（出口）：

Info： 计数器的基本参数数据结构

描述： 该服务读取指定计数器的基本特性。返回值<Info>指向存储了 AlarmBaseType 类型信息的结构体。

特性： 可由任务、ISR 以及多个回调函数中调用。

状态：

标准： 无错误，E_OK

扩展： <AlarmID>非法，E_OS_ID

3.2.7.3.2 GetAlarm

语法： StatusType GetAlarm (AlarmType <AlarmID>, TickRefType <Tick>)

参数（入口）：

AlarmID 警报 ID

参数（出口）：

Tick 警报<AlarmID>到期前的相对 tick 值。

描述： 该服务返回警报<AlarmID>到期前的相对 tick 值。

特性： 由应用来决定 CancelAlarm 等函数是否仍然有效。

如果<AlarmID>未使用，那么返回的<Tick>是不确定的值。

可由任务、ISR 和多个回调程序调用。

状态：

标准： 无错误，E_OK

警报<AlarmID>未使用，E_OS_NOFUNC

扩展： <AlarmID>非法，E_OS_ID

3.2.7.3.3 SetRelAlarm

语法： StatusType SetRelAlarm (AlarmType <AlarmID>, TickType <increment>, TickType <cycle>)

参数（入口）：

AlarmID　警报 ID

Increment　相对 tick 值

Cycle　循环报警时的周期值。单次报警时，该值应为 0。

参数（出口）　无

描述：　该服务占用<AlarmID>指定的警报。当经过了<increment>个 tick 后，分配给<AlarmID>的任务被激活或者事件被设置（只对扩展任务），或者警报回调函数被调用。

特性：　<increment>为 0 时如何处理由具体实现决定。

如果<increment>的相对值非常小，那么该系统服务返回用户之前警报就可能会到期，相应的任务就会变为就绪态或者就会调用报警回调程序。

如果<cycle>不等于 0，那么警报到<cycle>的值后立即重新开始。

如果要改变正在使用的警报的值，必须先取消警报。

如果警报已经在使用，那么该调用会被忽略，并返回错误代码 E_OS_STATE。

可由任务、ISR 调用，但不能由回调程序调用。

状态：

标准：　无错误，E_OK

警报<AlarmID>已经在使用，E_OS_STATE

扩展：　警报< AlarmID>非法，E_OS_ID

<increment>的值超出允许的极限（小于 0 或者大于 maxallowedvalue），E_OS_VALUE

<cycle>值不等于 0 且超出允许的计数极限（小于 mincycle 或者大于 maxallowedvalue），E_OS_VALUE

3.2.7.3.4 SetAbsAlarm

语法：　StatusType SetAbsAlarm (AlarmType <AlarmID>, TickType <start>, TickType <cycle>)

参数（入口）：

AlarmID　警报 ID

Start　绝对 tick 数

Cycle　　循环报警时的周期值。如果是单次报警，该值为 0。

参数（出口）：　无

描述：　该服务占用<AlarmID>指定的警报。当到达<start>个 tick 时，分配给警报<AlarmID>的任务被激活、事件被设置（只对扩展任务）或者警报回调函数被调用。

特性：　如果绝对值<start>非常接近计数器的当前值，那么在服务返回用户之前任务就可能变为就绪态或者就可能调用警报回调函数。

如果绝对值<start>在调用之前已经达到，那么只有当绝对值再次为<start>时警报才会到期，如等到下次计数器反转后。

如果<cycle>不等于 0，那么一旦达到相对值<cycle>后立即重新开始。

<AlarmID>不应已经被使用。

如果要改变正在使用的警报的值，必须先取消警报。

如果警报已经在使用，那么该调用会被忽略，并返回错误代码 E_OS_STATE。

可由任务、ISR 调用，但不能由回调程序调用。

状态：

标准：　无错误，E_OK

警报<AlarmID>已经在使用，E_OS_STATE

扩展：　警报< AlarmID>非法，E_OS_ID

<start> 的值超出允许的计数极限（小于 0 或者大于 maxallowedvalue），E_OS_VALUE

<cycle>值不等于 0 且超出允许的计数极限（小于 mincycle 或者大于 maxallowedvalue），E_OS_VALUE

3.2.7.3.5 CancelAlarm

语法：　StatusType CancelAlarm (AlarmType <AlarmID>)

参数（入口）：

AlarmID　　警报 ID

参数（出口）：　无

描述：　该服务取消警报<AlarmID>。

特性：　　　　可由任务、ISR 调用，但不能由回调程序调用。

状态：

　　标准：无错误，E_OK

　　　　　如果警报<AlarmID>未使用，E_OS_NOFUNC

　　扩展：警报<AlarmID>非法，E_OS_ID

3.2.7.4 常量

对于所有的计数器，GetAlarmBase 的返回值也可以通过下列常量获得：

OSMAXALLOWEDVALUE_x　　计数器 x 允许的最大 tick 值。

OSTICKSPERBASE_x　　计数器 x 达到一个特定的单元时的 tick 数

OSMINCYCLE_x　　计数器 x 循环报警时允许的最小 tick 数

因此，如果知道计数器的名字，那么没有必要调用 GetAlarmBase。

通常会至少有一个计数器是时间计数器（系统计数器）。该计数器的参数可以通过下列常量获得：

OSMAXALLOWEDVALUE　　系统计数器允许的最大 tick 值

OSTICKSPERBASE　　系统计数器达到特定计数单位时的 tick 数

OSMINCYCLE　　系统计数器循环报警时允许的最小 tick 数

另外还有下面的常量：

OSTICKDURATION　　系统计数器一个 tick 的长度，以纳秒为单位

3.2.7.5 命名规则

在应用中，报警回调函数的定义形式如下：

```
ALARMCALLBACK(AlarmCallBackName )
{
}
```

3.2.8 操作系统执行控制

3.2.8.1 数据类型

AppModeType

该数据类型代表应用模式。

3.2.8.2 系统服务

3.2.8.2.1 GetActiveApplicationMode

语法：　AppModeType GetActiveApplicationMode (void)

描述：　该服务返回当前的应用模式，可用来写与模式有关的代码。

特性：　可由任务、ISR 和所有的回调程序调用。

3.2.8.2.2 StartOS

语法：　void StartOS (AppModeType < Mode >)

参数（入口）：

　Mode　应用模式

参数（出口）：　无

描述：　用户调用该服务可以以指定的模式启动操作系统。

特性：　用于操作系统外部，因此在实现时会有特殊限制。系统启动。调用该服务不需要返回。

3.2.8.2.3 ShutdownOS

语法：　void ShutdownOS (StatusType < Error >)

参数（入口）：

　Error　发生的错误

参数（出口）：　无

描述：　用户可以调用该服务来关闭整个系统（如紧急关闭）。当出现未定义的内部状态或者不需要再允许时，操作系统也可以在内部调用该服务。

如果定义了 ShutdownHook 回调程序，那么在关闭操作系统前先调用 ShutdownHook（Error 作为参数）。

如果 ShutdownHook 会返回，那么 ShutdownOS 的进一步行为由具体实现决定。

<Error>应该是 UIP-Kernel 支持的合法的错误代码。

特性： 调用此服务后操作系统关闭。

可由任务、ISR 以及 ErrorHook 和 StartupHook 调用，也可由操作系统内部调用。

如果是操作系统调用 ShutdownOS，那么入口参数值不可能为 E_OK。

3.2.8.3 常量

OSDEFAULTAPPMODE 默认的应用模式，通常是 StartOS 的合法参数。

3.2.9 回调程序

3.2.9.1 数据类型

OSServiceIdType

该数据类型代表系统服务的标识

3.2.9.2 系统服务

3.2.9.2.1 ErrorHook

语法： void ErrorHook (StatusType <Error>)

参数（入口）：

Error 发生的错误

参数（出口）： 无

描述： 当系统服务返回的状态值不等于 E_OK 时，操作系统会在服务的最后调用该回调程序。它在返回任务级之前调用。

当任务激活或者事件设置过程中如果警报到期或者检测到有错误发生也会调用该回调程序。

当 ErrorHook 调用的系统服务返回的状态值不为 E_OK 时，不会

再调用 ErrorHook。ErrorHook 调用的系统服务的错误只能通过检查其状态值来发现。

特性：　　　　见相关回调程序的综述。

3.2.9.2.2 PreTaskHook

语法：　　　　void PreTaskHook (void)

参数（入口）：无

参数（出口）：无

描述：　　　　该服务在操作系统开始执行一个新的任务之前、任务状态转换为运行态之后调用。

特性：　　　　见相关回调程序的综述。

3.2.9.2.3 PostTaskHook

语法：　　　　void PostTaskHook (void)

参数（入口）：　无

参数（出口）：　无

描述：　　　　该服务在操作系统执行当前任务之后、离开运行态之后调用。

特性：　　　　见相关回调程序的综述。

3.2.9.2.4 StartupHook

语法：　　　　void StartupHook (void)

参数（入口）：　无

参数（出口）：　无

描述：　　　　该服务在操作系统初始化之后、调度程序运行前由操作系统调用。此时，应用程序可以进行设备驱动初始化等操作。

特性：　　　　见相关回调程序的综述。

3.2.9.2.5 ShutdownHook

语法：　　　　void ShutdownHook (StatusType < Error >)

参数（入口）：

　　Error　　发生的错误

参数（出口）：　无

描述：　　　　该回调程序当操作系统服务 ShutdownOS 被调用的情况下由操作系统调用，在关闭操作系统时调用。

特性：　　　　ShutdownHook 是用于发生未定义的系统关闭的回调程序。

3.2.9.3 常数

OSServiceId_xx 系统服务 xx 的唯一标识，例如：OSServiceId_ActiveTask，OSServiceId_xx 是 OSServiceIdType 类型的。

3.2.9.4 宏

OSErrorGetServiceId 提供发生错误的服务的标识。服务标识是 OSServiceIdType 类型的。可能的取值是 OSServiceId_xx，其中 xx 是系统服务的名称。

OSError_x1_x2　　在 ErrorHook 内获取调用 ErrorHook 的系统服务参数的宏，其中，x1 是系统服务的名称，x2 是参数的名称。

四、软件应用

4.1 运行时上下文的配置

每个任务都有一个运行时上下文，也就是指任务的所有内存资源，在执行开始时被占据，一旦任务结束就将被释放。通常情况下运行时上下文由寄存器、任务控制模块和一定数量可操作的堆栈空间组成。

根据任务设计的不同（类型和是否抢占）和调度机制的不同（不可抢占、混和抢占和可抢占)，运行时上下文的大小也会不同。相互不会抢占的任务可以在同一个运行时上下文中运行，以达到对可用 RAM 空间的高效利用。

4.2 应用设计建议

本章的目的是针对在进行 UIP-Kernel 操作系统的应用设计时可能出现的一些问题提供附加的信息。本规范并不能涉及系统设计的所有问题。其他的设计方法来自于当前 DSP 应用设计经验。

4.2.1 资源管理

本节提到的这些内容，目的在于保证对所有资源的适当操作。

4.2.1.1 依照 LIFO 的资源占用

每次对资源的访问都可封装对服务 GetResource 和 ReleaseResource 的调用。资源的释放顺序应与其占有顺序相反。以下代码的顺序是错误的，因为函数 foo 不可以释放 res_1。

```
TASK(incorrect)
{
    GetResource( res_1 );
    /* some code accessing resource res_1 */
    ...
    foo();
    ...
    ReleaseResource( res_2 );
}
void foo()
{
    GetResource( res_2 );
    /* code accessing resource res_2 */
    ...
    ReleaseResource( res_1 );
}
```

嵌套的资源占有是允许的。资源占据必须严格依照 LIFO（堆栈法则）严格进行。如果如上所示的要访问资源的代码被优先级更高（高于资源的天花板优先级）的任务抢占，那么在这个任务中申请其他的资源就导致了嵌套资源占有，这是符合 LIFO 的。

4.2.1.2 API 服务的调用等级

UIP-Kernel API 服务 GetResource 和 ReleaseResource 应从同一函数调用等级调用。假如函数 foo 按照 LIFO 是正确的，进行如下的资源占有：

```
void foo( void )
{
    ReleaseResource( res_1 );
    GetResource( res_2 );
    /* some code accessing resource res_2 */
    ...
    ReleaseResource( res_2 );
}
```

仍可能会出现问题，因为 ReleaseResource(res_1)与 GetResource(res_1)是在不同的等级上被调用的。在实现时从不同的等级调用 API 服务可能会引发一些问题。

4.2.1.3 任务终止或中断结束时资源仍然被占据的情况

对资源的访问应该直接被封装成对 GetResource 和 ReleaseResource 的调用。否则可能会出现终止任务时忘记释放资源的情况。

```
GetResource( res_1 );
...
switch ( condition )
{
    case CASE_1 :
        do_something1();
        ReleaseResource( res_1 );
        break;
    case CASE_2 : /* !!! WRONG: no release of resource here !!! */
        do_something2();
        break;
    default:
        do_something3();
        ReleaseResource( res_1 );
    }
...
```

在操作系统的标准状态下的任务终止时，或者在标准或扩展状态下的中断结束时，如果没有完全释放被占据的资源，由此导致的系统行为规范中并未详细说明。根据操作系统的具体实现，资源可能会被锁死，因为操作系统拒绝所有对该资源的进一步访问。

4.2.2 API 调用的位置

API 服务 TerminateTask 和 ChainTask 对于操作系统至关重要。这两个服务都是用来终止正在运行的任务。从任务的子程序层调用这些服务，操作系统负责在终止任务时对堆栈进行适当的处理。一种处理方法是在运行态任务的入口保存堆栈指针的位置，在任务终止时恢复其值。

4.2.3 中断服务程序

在使用 1 型和 2 型的 ISR 时，用户应注意一些可能出错的情况。

4.2.3.1 不同类型中断的嵌套

因为所有中断的优先级都比任务高，所以在系统回到任务层之前对中断的处理应该终止。如果一个 2 型 ISR 中断了一个 1 型 ISR，系统会在 ISR2 结束后再继续处理 ISR1。如果在中断层 ISR2 中进行了任务的激活或者事件的设置，那么为了进行任务的重调度，操作系统不会在 ISR1 结束后被唤醒。

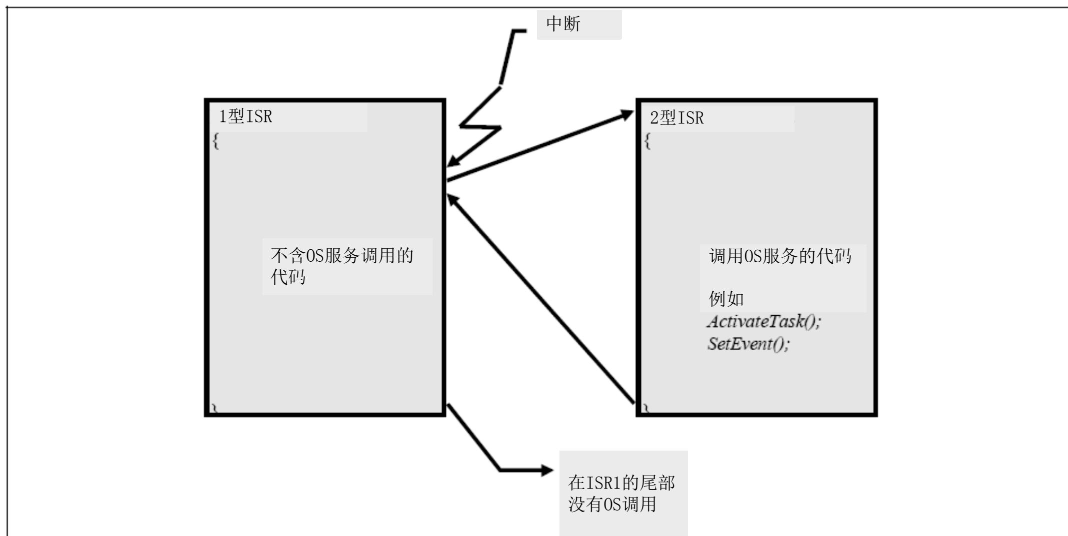

图 18　嵌套的中断

由于 1 型 ISR 并不在操作系统的控制下运行，所以操作系统不可能在 ISR 结束时进行任务的重新调度。这样，在 ISR2 中任何操作系统调用引起的活动都会

被延时到下一个重新调度时刻。

针对上面讨论的问题，每一个系统都应该设定规则以避免这些问题出现。一些特别的实现可以避免这些问题，或者应用本身的特性使得根本不会发生这些问题（如非抢占系统）。因此这些规则在实现和应用中均应受到重视。

然而为了获得最大的应用可移植性，有一个简单而有效的方法：所有 1 型中断的硬件优先级都大于等于 2 型中断。

4.2.3.2 中断层的直接操作

中断层的直接操作是不可移植的，受到实现的限制。

4.2.4 优先级和抢占

系统会依据任务的优先级调度任务。一个任务可以声明为可抢占的或者不可抢占的。应用程序应该连续地对待这两种任务属性，以避免系统运行时发生冲突。需要注意低优先级的不可抢占任务会延时高优先级的任务。

通常情况下任务是否可抢占在系统设计时指定，任务优先级在系统生成时配置。因为大的软件工程会有很多人参与，所以开发过程需要精确的协作。为了实现运行时系统行为状态良好，这种协作是十分关键的。

4.2.5 内部资源用法举例

除了不可抢占式任务以外，内部资源还能在很多情况下被使用。

一般情况下，内部资源可以用来保护一组任务，使它们免受同组其他任务的抢占，除非这个任务组中正在运行的任务明确地调用 WaitEvent 和 TerminateTask/ChainTask 函数进行任务的重新调度。举例来说，如果一个组中的所有任务只在第一个任务处理等级调用这些函数，那么这些任务堆栈的使用会得到极大的优化。

除了不可抢占式任务外还有一个例子，这种情况有时被称为“协作任务”，在这种情况下，多个最低优先级的任务共享相同的内部资源，可以被高优先级的任务自由抢占，但相互之间不能抢占。 对这个例子进行扩展——将系统中最低优先级的任务作为背景任务，这样这个任务就可以被所有的任务抢占。

不可抢占任务和协作任务的概念可以通过在一个配置中使用两个不同的内部资源的方法容易地在一个系统中结合到一起。

没有分配内部资源的任务是可抢占的。

4.2.6 传递给 shutdownOS 的参数

传递给 shutdownOS 的参数同样也会传递到 ShutdownHook 。如果操作系统调用 ShutdownHook 函数，那么传递的参数是与实现相关的错误值。如果用户调用 ShutdownOS 函数，那么应该使用已有的 UIP-Kernel 操作系统错误值。

. 强烈推荐使用实现文档中给出的错误值。如果没有特别为 ShutdownOS 定义错误值，还可以使用 E_OK，用以区别是操作系统调用 ShutdownOS 函数还是应用程序调用。

4.2.7 错误处理

应用程序软件中的错误一般是由以下原因造成的：

- 操作系统处理中的错误，例如，对操作系统错误的配置/初始化/定义尺寸，或者违反了操作系统服务的规定。
- 软件设计中的错误，例如，任务优先级选择不当、重要代码没有保护、计时错误、任务低效率设计。

实现的测试

断点、跟踪、时间戳可以被单独地集成到应用软件中。

例如用户可以设置时间戳，以便在调用系统服务之前在以下几个位置追踪程序执行：

- 当激活或终止任务时
- 扩展任务进行事件设置和事件清除时
- 明确的调度时刻
- 在 ISR 开始或结束时
- 当占有和释放资源时或者在临界状态时

时间监视

操作系统的时间监视特性并不是必须的，时间监视的作用是保证一旦超过定义的最大时长就激活每个任务或者比如激活最低优先级的任务。

用户可选择使用回调程序或者设置一个看门狗对操作系统状态进行监视。

组成成员

在 UIP-Kernel 操作系统中，组成成员（如 DeclareTask）是创建应用程序中的系统对象的一种方法。像外部声明一样，组成成员位于源文件的头部。在具体实现时，它们也可以用作宏。

4.2.8 错误和警告

大多数系统服务的错误值都指向应用程序错误。然而在如下的特殊情况下，错误值表示一些在正常操作过程中会出现的警告：

- ActivateTask, ChainTask E_OS_LIMIT (标准的)
- GetAlarm E_OS_NOFUNC (标准的)
- SetAbsAlarm, SetRelAlarm E_OS_STATE (标准的)
- CancelAlarm E_OS_NOFUNC (标准的)

特别是使用 ErrorHook 实现一个集中错误处理时，需要考虑上述情况。