UIP-Kernel 编程指南

撰写人：UIP 团队

目　　录

目　　录 1
一、序言 5
1.1 编写目的 5
1.2 文档结构 6
二、实时操作系统 UIP-Kernel 7
三、快速入门 8
3.1 准备环境 8
3.2 系统启动 9
3.3 系统关闭 10
3.4 范例 11
四、UIP-Kernel 简介 20
4.1 系统概述 20
4.1.1 任务管理 20
4.1.2 任务同步机制 20
4.1.3 中断管理 21
4.1.4 警报 21
4.1.5 消息管理 21
4.1.6 Hook 机制 21
4.2 支持的平台 22
五、任务管理 23
5.1 实时系统的需求 24
5.2 任务调度器 25
5.3 任务控制块 27
5.4 任务状态模型 29
5.4.1 扩展任务 29
5.4.2 基本任务 30
5.5 空闲任务 32
5.6 任务相关接口 33
5.6.1 数据类型 33
5.6.2 组成成员 33
5.6.2.1 DeclareTask 33
5.6.3 系统服务 33
5.6.3.1 ActiveTask 33
5.6.3.2 TerminateTask 34

5.6.3.3 ChainTask 35
5.6.3.4 Schedule 36
5.6.3.5 GetTaskID 36
5.6.3.6 GetTaskState 37
5.6.4 常量 37
5.6.5 命名规则 37
5.7 范例 39
六、中断 40
6.1 中断处理过程 40
6.2 中断服务程序 41
6.3 中断相关接口 42
6.3.1 数据类型 42
6.3.2 系统服务 42
6.3.2.1 EnableAllInterrupts 42
6.3.2.2 DisableAllInterrupts 42
6.3.2.3 ResumeAllInterrupts 43
6.3.2.4 SuspendAllInterrupts 44
6.3.2.5 ResumeOSInterrupts 44
6.3.2.6 SuspendOSInterrupts 45
6.3.3 命名规则 45
6.4 范例 47
七、事件 50
7.1 事件相关接口 52
7.1.1 数据类型 52
7.1.2 组成成员 52
7.1.2.1 DeclareEvent 52
7.1.3 系统服务 52
7.1.3.1 SetEvent 52
7.1.3.2 ClearEvent 53
7.1.3.3 GetEvent 53
7.1.3.4 WaitEvent 54
7.2 范例 55
八、资源管理 58
8.1 访问并占有资源时的处理方法 59
8.2 使用资源时的限制 60
8.3 作为资源的调度程序 61
8.4 资源控制块 62
8.5 资源相关接口 63
8.5.1 数据类型 63

8.5.2 组成成员 63
8.5.2.1 DeclareResource 63
8.5.3 系统服务 63
8.5.3.1 GetResource 63
8.5.3.2 ReleaseResource 64
8.6 范例 65
九、消息 67
9.1 消息控制块 67
9.2 消息相关接口 69
9.2.1 数据类型 69
9.2.2 系统服务 69
9.2.2.1 SendMessage 69
9.2.2.2 ReceiveMessage 70
9.2.2.3 GetMessageStatus 70
9.2.2.4 ReadFlag 70
9.2.2.5 ResetFlag 71
9.3 范例 72
十、警报 78
10.1 计数器 78
10.2 警报管理 79
10.3 报警回调函数 80
10.4 警报相关接口 81
10.4.1 数据类型 81
10.4.1.1 TickType 81
10.4.2 组成成员 82
10.4.2.1 DeclareAlarm 82
10.4.3 系统服务 82
10.4.3.1 GetAlarmBase 82
10.4.3.2 GetAlarm 82
10.4.3.3 SetRelAlarm 83
10.4.3.4 SetAbsAlarm 84
10.4.3.5 CancelAlarm 85
10.4.4 常量 85
10.4.5 命名规则 86
10.5 范例 87
十一、回调机制 88
11.1 回调程序 88
11.2 回调程序相关接口 89
11.2.1 数据类型 89

11.2.2 系统服务 89
11.2.2.1 ErrorHook 89
11.2.2.2 PreTaskHook 89
11.2.2.3 PostTaskHook 89
11.2.2.4 StartupHook.. 90
11.2.2.5 ShutdownHook 90
11.2.3 常数..90
11.2.4 宏..90
十二、应用程序设计建议 91
12.1 资源管理 91
12.1.1 依照 LIFO 的资源占用..91
12.1.2 API 服务的调用等级 92
12.1.3 任务终止或中断结束时资源仍然被占据的情况 92
12.2 API 调用的位置..94
12.3 中断服务程序..95
12.3.1 不同类型中断的嵌套 95
12.3.2 中断层的直接操作 95
12.4 优先级和抢占..96
12.5 内部资源用法举例 97
12.6 传递给 shutdownOS 的参数..98
12.7 错误处理 99
12.8 错误和警告..100

一、序言

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，包括详细介绍 UIP-Kernel 系统的主要功能、各个模块的应用接口以及如何使用应用接口进行编程的示例程序。

本文档面向使用 UIP-Kernel 操作系统进行编程的开发人员，并假定开发人员具备基本的 C 语言基础知识，如果具备基本的实时操作系统知识将能更好地理解文档中的一些基本概念。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，对于 UIP-Kernel 内部实现并不做过多、过细节地介绍。

1.2 文档结构

本文档是 UIP-Kernel 实时操作系统的编程指南，描述了：

- UIP-Kernel 快速入门，在无硬件平台的情况下，如何迅速了解 UIP-Kernel 实时操作系统；
- 详细介绍各个模块的结构，举例说明如何使用 UIP-Kernel 的应用接口进行编程，并介绍编程时的注意事项。

二、实时操作系统 UIP-Kernel

实时操作系统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。因而，提供及时响应和高可靠性是其主要特点。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要保证的；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 是一个支持多线程并发执行的硬实时操作系统，它支持在多个 MCU 和 DSP 处理器以及同一处理器的不同核上实现任务统一调度、资源统一管理、通信统一管理、中断统一管理、数据统一管理，从而达到功能满足、性能优化、维护性保障提升的目的。

UIP-Kernel 操作系统具有以下特点：

（1）标准化的接口

应用软件与操作系统之间的接口由系统服务定义。操作系统应用于不同类型的处理器时接口完全相同。系统服务符合 ISO/类 ANSI-C 语法。

（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 采用标准化接口（服务调用、数据类型和常量），可以实现源代码级移植。

三、快速入门

3.1 准备环境

在运行UIP-Kernel前，需要安装编译链接用的集成开发环境。UIP-Kernel是提供源码的实时操作系统，开发应用时，UIP-Kernel以源码库或者目标库的形式与应用程序代码一起共存于同一工程中，编译、链接后在目标板中调试、运行。其整个构建步骤如下所示：

（1） 安装芯片体系结构对应的集成开发环境

（2） 将UIP-Kernel实时操作系统源代码解压到指定文件夹下（例如：解压到E:\UIP-Kernel)

解压完成后的目录结构主要包括：app、bsp、uip-kernel等。

（3）打开根目录中对应的工程文件

（4）将开发的应用程序代码放在工程的app目录中，然后选择Build，可将UIP-Kernel与应用程序代码一起编译、链接，之后下载到目标板中调试、运行。

3.2 系统启动

在处理器复位后初始化将会实现，系统首先调用platform_init和platform_uart_init进行硬件平台的初始化，然后调用StartOS对操作系统和应用程序中使用的资源、消息、任务、警报等进行初始化，

操作系统初始化后（调度内核尚未执行前），StartOS 调用启动回调程序StartupHook，用户可以把所有与操作系统有关的初始化代码放于此函数中。为了组织 StartupHook 中的初始化代码，UIP-Kernel 提供 GetActiveApplicationMode函数。从启动回调函数返回后，UIP-Kernel 进行中断使能，并启动调度内核。之后，系统开始运行，执行应用层任务。具体步骤如图 1 所示。

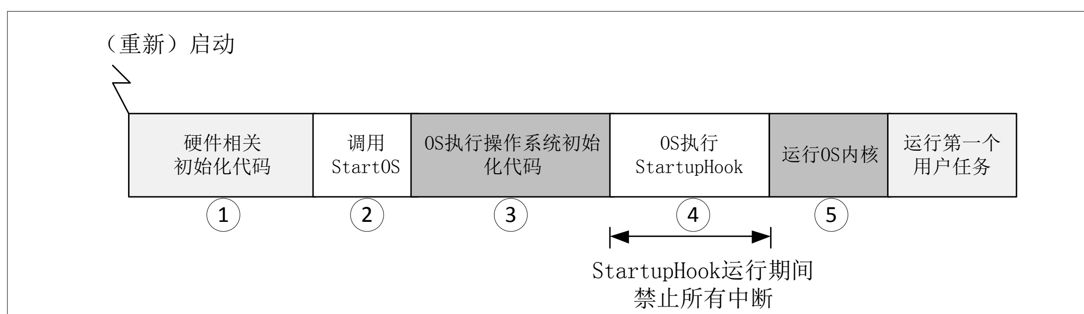

图 1 系统启动

图 1 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，不可移植部分代码结束；

②调用 StartOS 函数，应用模式作为该函数的一个参数。该函数启动操作系统；

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，禁止所有用户中断；

⑤操作系统使能用户中断、启动调度内核，启动当前应用软件定义的自动运行任务和警报，自动运行的任务启动早于自动运行的警报。

3.3 系统关闭

UIP-Kernel 定义了关闭操作系统的服务 API ShutdownOS。可由应用程序主动执行或发生致命错误时由操作系统调用。当 ShutdownOS 函数被调用时，UIP-Kernel 将调用 ShutdownHook 回调程序，然后关闭系统。

3.4 范例

范例1演示UIP-Kernel的程序框架。共包括两个文件demo.c（或timer_test.c），app_cfg.c。在 app_cfg.c 文件中，给出了系统中用到的警报、消息、资源、任务等的具体定义。在 demo.c 文件中定义了 uipAppInitialize(VOID)、任务的执行体、main()函数。demo.c 的程序代码如下所示：

```
#include "uipkernel_api.h"
VOID uipAppInitialize(VOID)
{
        uipTaskInitialize((uipTaskPtr)&AppTask[0], AppTaskNumber);
        uipAlarmInitialize((uipAlarmPtr)&AppAlarm[0], AppAlarmNumber);
        uipResInitialize((uipResPtr)&AppRes[0], AppResNumber);
        uipMsgInitialize((uipMsgPtr)&AppMsg[0], AppMsgNumber);
        uipCurrentTaskTCBPtr = &AppTask[TASK_idle_ID];
        uipCurrentTaskTCBPtr -> IsInited = 1;
        uipCurrentTaskPriority = TASK_50_READY_PRIO;
}

TASK(TASK_0)
{
     /*   任务TASK_0的执行体   */
     ……
}

TASK(TASK_1)
{
     /*   任务TASK_1的执行体   */
     ……
}
int main(void)
{
```

```
    int         mode;
    platform_init_flags    init_flags;
    platform_init_config init_config;
    mode = 0x01;
    memset(&init_flags, 0x01, sizeof(platform_init_flags));
    init_config.pllm = 0;
    platform_init(&init_flags, &init_config);
    /* Start RTOS */
    platform_uart_init();
    while (1)
    {
        StartOS(mode);
    }
    /* Return to exit (never happens) */
    return 1;
}
```

头文件“uipkernel_api.h”包括了系统提供的所有 API 的定义。

uipAppInitialize(VOID)函数对应用程序中用到的资源、消息、警报、任务等（在 app_cfg.c 中定义）进行初始化。

假设应用程序中定义了两个任务 TASK_0 和 TASK_1，则 TASK(TASK_0)用于定义任务 TASK_0 的执行体，TASK(TASK_0)用于定义任务 TASK_0 的执行体。

在 main()函数中，首先进行硬件平台初始化，然后调用 StartOS(mode)对内核和应用程序中用到的资源、消息、警报、任务等（在 app_cfg.c 中定义）进行初始化，最后调用 BSP 中的初始化函数对中断控制器、定时器等进行初始化，并开启异常处理、启动定时器，使能用户中断、启动调度内核、启动当前应用软件定义的自动运行任务和警报，开始整个应用软件的调度运行。

为了方便应用软件的设计，将应用软件中使用的所有资源、消息、警报、任务等定义在 cfg 文件中。cfg 文件可由系统自动生成，也可手工编写，假设应用程序中使用到 2 个警报、2 个资源、2 个消息、1 个事件、2 个任务（加上空闲任务，共 3 个），则对应的 app_cfg.c 的程序代码如下所示。

```
/*  AppAlarmBase 中存储计数器特性  */
AlarmBaseType AppAlarmBase[] =
{
    {
        65535,
        1,
        0,
    },
    {
        65535,
        1,
        0,
    }
};

/*  AppCount 是计数器定义，共定义了 2 个计数器  */
uipCounter AppCounter[] =
{
    {
        0,                      /* Counter's current value */
        0,
        &AppAlarmBase[0],
    },
    {
        0,                      /* Counter's current value */
        0,
        &AppAlarmBase[1],
    }
};

/*  AppAlarm 是系统中使用到的警报的定义，共定义了 2 个警报  */
uipAlarm   AppAlarm[] =
{
```

```
        {
            100,                        /* Alarm value */
            100,                        /* Cycle */
            APP_EVENT_TASK_0,           /* Event */
            &AppCounter[0],             /* Counter */
            0,                          /* CallBack */
            TASK_0_ID,                  /* Task ID */
            uip_ALARM_ON,               /* Alarm state */
            0,                          /* padding byte */
            0,                          /* padding byte */
        },
        {
            100,                        /* Alarm value */
            100,                        /* Cycle */
            APP_EVENT_TASK_1,           /* Event */
            &AppCounter[1],             /* Counter */
            0,                          /* CallBack */
            TASK_1_ID,                  /* Task ID */
            uip_ALARM_ON,               /* Alarm state */
            0,                          /* padding byte */
            0,                          /* padding byte */
        },
};

AlarmNumberType AppAlarmNumber = 2;   /*  系统中定义的警报个数：2    */

/*        系统中使用的资源的定义，共定义了 2 个资源            */
uipRes AppRes[] =
{
      {
        (struct uip_TSK_STRUCT *)0,
                          /* Pointer to TCB of task that is using this resource */
        (struct uip_RESOURCE_STRUCT *)0,    /* Link to next resource */
```

```
        0,
        /* Bit value in task priority group corresponding to task ready priority */
        RESOURCE_0_CEILING_PRIO,
                                    /*Resource's mutex ceiling priority */
        0,
      /* Index value into ready task table corresponding to task ready priority */
        0,
           /* Bit position in ready table corresponding to task ready priority */
        0,
                 /* Dispatched priority of task that is using this resource */
        0,
             /* Show the state of resource: 0 not being used; 1 being used */
        0,                                  /* Padding byte for ARM */
        0,                                  /* Padding byte for ARM */
        0,                                  /* Padding byte for ARM */
    },
     {
       (struct uip_TSK_STRUCT *)0,
       (struct uip_RESOURCE_STRUCT *)0,
       0,
       RESOURCE_1_CEILING_PRIO,
       0,
       0,
       0,
       0,
       0,
       0,
    },
};

ResNumberType AppResNumber = 2;     /*  系统中使用的资源个数：2   */

 /*   系统中使用的消息定义，共定义了 2 个消息   */
```

```
uip_CHAR AppMsgBuf_1[APP_MSG_LENGTH];
uip_CHAR AppMsgBuf_2[APP_MSG_LENGTH];
uipMsg AppMsg[] =
{
    {
        MSG_ID_0,                                   /* Message's id */
        0,
        /* Messages's current state(locked, unlocked, overflow or not etc.) */
        uip_MSG_QUEUED,
                        /* Messages's type(unqueued or queued message) */
        0,                              /* Flags to be set for notifying */
        (uip_UCHAR_PTR)&AppMsgBuf_1[0],
                                        /* Point to buffer start address */
        (uip_UCHAR_PTR)0,       /* Point to buffer address to be writen */
        ((uip_UCHAR_PTR)&AppMsgBuf_1[0]+sizeof(AppMsgBuf_1)),
                                        /* Point to buffer end address */
        (uip_UCHAR_PTR)0,       /* Point to buffer address to be read */
        APP_MSG_SIZE,                       /* Size of messages(bytes) */
        APP_EVENT_FOR_TASK,         /* Event to be set for notifying */
        (VOID *)0,          /* Callback routine called when sending message */
        TASK_4_ID,                          /* The ID of task to be nofified */
        uip_COM_NOTIFICATION_SETEVENT,      /* notification type */
        0,                                      /* Padding byte for ARM */
        0,                                      /* Padding byte for ARM */
    },
    {
        MSG_ID_1,
        0,
        uip_MSG_QUEUED,
        0,
        (uip_UCHAR_PTR)&AppMsgBuf_2[0],
        (uip_UCHAR_PTR)0,
        ((uip_UCHAR_PTR)&AppMsgBuf_2[0]+sizeof(AppMsgBuf_2)),
        (uip_UCHAR_PTR)0,
```

```
            APP_MSG_SIZE,
            APP_EVENT_FOR_TASK,
            (VOID *)0,
            TASK_6_ID,
            uip_COM_NOTIFICATION_SETEVENT,
            0,
            0,
        },
    };

MsgNumberType AppMsgNumber = 2;    /*  系统中定义的消息个数：2    */

/*  任务和栈定义  */
#pragma DATA_ALIGN(Task_0_Stack, 8);
#pragma DATA_ALIGN(Task_1_Stack, 8);

uip_UINT32 Task_0_Stack[APP_STACK_SIZE];
uip_UINT32 Task_1_Stack[APP_STACK_SIZE];

/*    声明 2 个任务    */
DeclareTask(TASK_0);
DeclareTask(TASK_1);

/*    任务结构体的定义    */
uipTask AppTask[] =
{
    {
        &Task_0_Stack[APP_STACK_SIZE-1],
                                        /* Task's stack starting address */
        TASK_0_ID,                                          /* Task's ID */
        READY_STATE | uip_TASK_EXTENDED,
                                            /* Taks's execution state */
        TASK_0_READY_PRIO,                      /* Task's ready priority */
```

```
#ifdef MixPreemption
        TASK_0_DISPATCH_PRIO,           /* Task's dispatched priority */
#else
        0,                                          /* padding byte */
#endif
        0,                 /* The occupaited resources list of this task*/
        TASK_0,                                 /* Task's entry point */
        (APP_STACK_SIZE-1),                     /* Task's stack size */
        0,                                /* Mask of waited events */
        0,                                /* Mask of setted events */
        0,
    /* Bit value in task priority group corresponding to task ready priority */
        0,
   /* Index value into ready task table corresponding to task ready priority */
        0,
         /* Bit position in ready table corresponding to task ready priority */
        0,                                /* Task activated number */
        0,                                          /* padding byte */
#ifdef MixPreemption
        0,
   /* Bit value in task priority group corresponding to task dispatch priority */
        0,
  /* Index value into ready task table corresponding to task dispatch priority */
        0,
        /* Bit position in ready table corresponding to task dispatch priority */
        0,                                          /* padding byte */
        0,                                          /* padding byte */
#endif
        0
    },
    {
        &Task_1_Stack[APP_STACK_SIZE-1],
        TASK_1_ID,
        READY_STATE | uip_TASK_EXTENDED,
```

```
        TASK_1_READY_PRIO,
#ifdef MixPreemption
        TASK_1_DISPATCH_PRIO,
#else
        0,
#endif
        0,
        TASK_1,
        (APP_STACK_SIZE-1),
        0,
        0,
        0,
        0,
        0,
        0,
        0,
#ifdef MixPreemption
        0,
        0,
        0,
        0,
        0,
#endif
        0
    },
};

TaskNumberType AppTaskNumber = 3;    /* 系统中定义的任务个数 */
```

由示例 1 可知，在开发应用程序时，至少需要两个文件：一个是 cfg 文件，用于对应用程序中使用的任务、警报、资源、消息等进行定义；另一个是应用文件，在该文件中定义应用程序中定义的任务的任务执行体、main 函数等。在具体应用软件开发时，程序员根据应用程序的要求编写程序代码。

四、UIP-Kernel 简介

UIP-Kernel 是一个实时操作系统，主要提供了任务管理及调度、中断管理、任务间通信等功能。包含若干汇编文件，其代码实现系统启动和任务上下文切换，其余均为 C 文件。

4.1 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。

4.1.1 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融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。

4.1.2 任务同步机制

UIP-Kernel的任务同步主要包括资源管理、事件机制。资源管理主要控制对共享（逻辑）资源或器件的访问，或者控制程序的执行流程，协调不同优先级任务或ISR对共享资源的访问，共享资源可以是管理实体（调度内核）、程序段、内存或者硬件资源等；事件机制用于任务同步的事件管理，是针对扩展任务的一种

同步方法，能将任务从等待态转出或转为等待态。

4.1.3 中断管理

支持中断嵌套，嵌套层数从 RTOS 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。中断服务程序可划分为两种类型：

1 型 ISR：这种 ISR 不使用操作系统 API。在 ISR 结束后，回到中断发生时的位置继续运行，中断对任务管理没有影响，这种中断处理方式的开销最小。

2 型 ISR：UIP-Kernel 提供一个 ISR 处理框架，为用户程序准备运行时环境。在系统生成时就把用户程序分配给指定的中断。

4.1.4 警报

UIP-Kernel 为处理重复发生事件提供了警报服务，警报可分为相对报警和绝对报警。当警报到达时，UIP-Kernel 会提供激活任务、设置事件或者调用报警回调程序等服务。

4.1.5 消息管理

UIP-Kernel 基于消息（message）进行通信，并允许使用长度为零的消息。

4.1.6 Hook 机制

UIP-Kernel 提供特殊的回调程序，允许在操作系统内部调用自定义动作。

4.2 支持的平台

UIP-Kernel 支持的硬件平台包括：各种 Cortex-M/A 系列，以及 ARM7/9 等 ARM 微处理器；TI/ADI 的 DSP 等。

五、任务管理

一个典型的简单应用程序会被设计成一个串行的系统运行：按照准确的指令步骤一次一个指令地运行。但是这种方法对于复杂一些的实时应用是不可行的，因为它们通常需要在固定的时间内“同时”处理多个输入输出，实时软件应用程序应该设计成一个并行的系统。并行设计需要开发人员把一个应用分解成一个个小的，可调度的，序列化的程序单元（任务）。当合理地划分任务，正确地并行执行时，这种设计能够让系统满足实时系统的性能及时间的要求。在这种管理模式中，任务提供函数执行的主体框架，操作系统提供任务同步和异步两种执行方式，调度程序负责组织任务的执行顺序。

5.1 实时系统的需求

在UIP-Kernel实时操作系统中，任务是最基本的调度单位，它描述了一个任务执行的上下文关系，也描述了这个任务所处的优先等级。系统总共支持256个优先级（0~255，数值越小的优先级越高，0为最高优先级，255分配给空闲任务使用），重要的任务能拥有相对较高的优先级，非重要的任务优先级可以放低。系统根据任务的优先级选择高优先级任务执行以保证系统的硬实时性。

5.2 任务调度器

调度是内核的主要职责之一，就是决定该轮到哪个任务运行了。**UIP-Kernel 中提供的任务调度器提供了完全抢占式调度、非抢占式调度和混和抢占式调度三种调度方式**。

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。这种调度方式使得反应时间与低优先级任务的运行时间无关，保证了系统的硬实时性。例如，在图 2 中，低优先级的任务 2 并没有延迟高优先级任务 1 的调度。

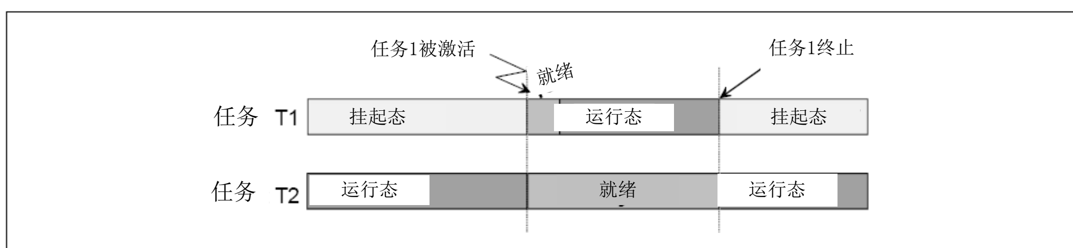

图 2 完全抢占式调度

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。例如，在图 3 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）。

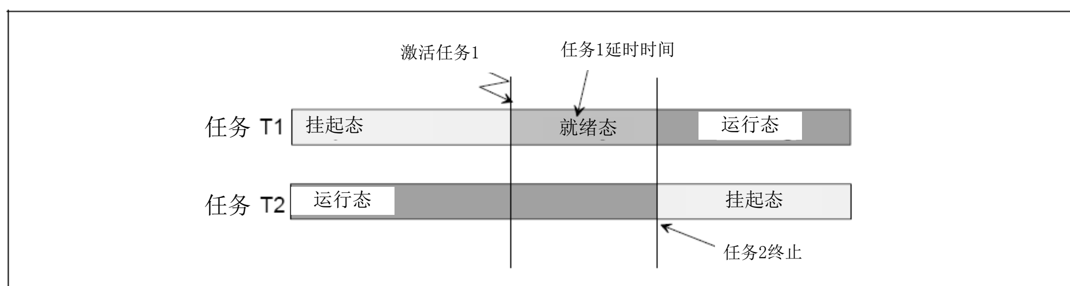

图 3 非抢占式任务调度

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

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。

5.3 任务控制块

任务控制块是操作系统用于控制任务的一个数据结构，它会存放任务的一些信息，例如优先级，名称等，也包含任务与任务之间的链表结构，任务等待事件集合等。在UIP-Kernel实时操作系统中，任务控制块由结构体uipTask（如下代码中所示）表示，另外一种写法是uipTaskPtr，在C实现上是指向任务控制块的指针。

```
typedef struct uip_TSK_STRUCT
{
    uip_UINT32_PTR uipStackPtr;                 /*Task's stack starting or end address */
    uip_UCHAR       uipTaskID;                  /* Task's ID */
    uip_UCHAR       uipTaskState;               /* Taks's execution state */
    uip_UCHAR       uipReadyPrio;               /* Task's ready priority */
#ifdef MixPreemption
    uip_UCHAR       uipDispatchPrio;            /* Task's dispatched priority */
#else
    uip_UCHAR       padding_1;                  /* Padding byte for ARM */
#endif
    struct uip_RESOURCE_STRUCT    *uipResources;
                                    /* The occupaited resources list of this task*/
    VOID            (*uipTaskEntry)(VOID);      /* Task's entry point */
    uip_UINT32      uipStackSize;               /* Task's stack size */
    EventMaskType   uipWaitEvent;               /* Mask of waited events */
    EventMaskType   uipSetEvent;                /* Mask of setted events */
    uip_UINT32      uipRdyPrioGrpBit;
                    /* Bit value in task priority group corresponding to task ready priority */
    uip_UCHAR       uipRdyPrioGrpArrayIndex;
                    /* Index value into ready task table corresponding to task ready priority */
    uip_UCHAR       uipRdyPrioGrpArrayBit;
                    /* Bit position in ready table corresponding to task ready priority */
    uip_UCHAR       uipActivatedNumber;         /* Task activated number */
```

```
    uip_UCHAR           padding_2;                    /* Padding byte for ARM */
#ifdef MixPreemption
    uip_UINT32          uipDispPrioGrpBit;
                        /* Bit value in task priority group corresponding to task dispatch priority */
    uip_UCHAR           uipDispPrioGrpArrayIndex;
                        /* Index value into ready task table corresponding to task dispatch priority */
    uip_UCHAR           uipDispPrioGrpArrayBit;
                        /* Bit position in ready table  corresponding to task dispatch priority */
    uip_UCHAR           padding_3;                    /* Padding byte for ARM */
    uip_UCHAR           padding_4;                    /* Padding byte for ARM */
#endif
} uipTask;
typedef     uipTask*        ArtTaskPtr;
```

5.4 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。由于UIP-Kernel操作系统提供了两种不同的任务：基本任务和扩展任务，两者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。

5.4.1 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。

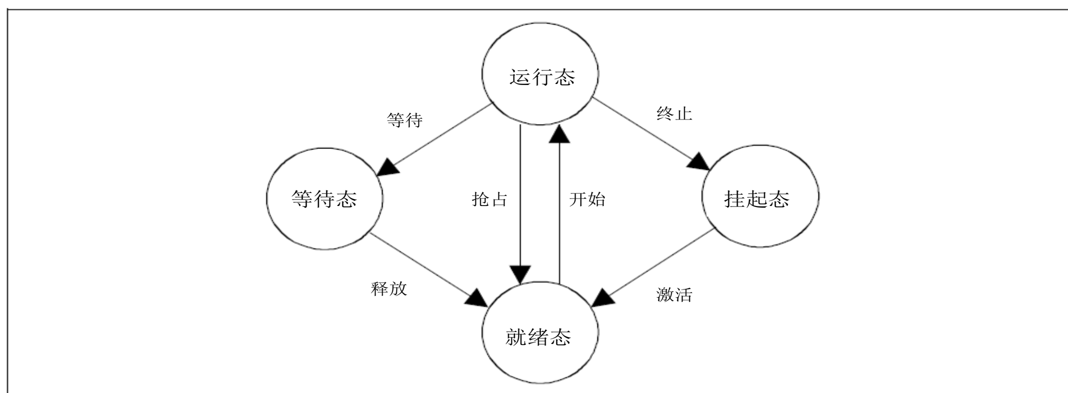

图 4 扩展任务状态模型

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的。

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。

任务的终止是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。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。

5.4.2 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。

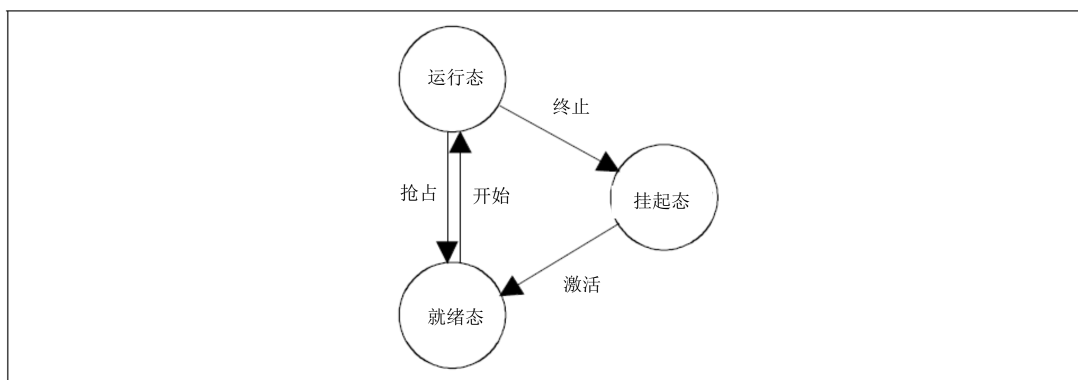

图 5 基本任务状态模型

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。

5.5 空闲任务

空闲任务是系统中一个比较特殊的任务，它具备最低的优先级，当系统中无其它任务可运行时，调度内核将调度执行空闲任务。空闲任务通常是一个死循环，永远不被挂起。

5.6 任务相关接口

5.6.1 数据类型

（1）TaskType

该数据类型标识了一个任务，具体定义：

typedef uip_INT32 TaskType；

（2）TaskRefType

该数据类型指向一个可变的 TaskType，具体定义：

Typedef uip_INT32_PTR TaskRefType;

（3）TaskStateType

该数据类型标识了一个任务的状态，具体定义

Typedef uip_UCHAR TaskStateType;

（4）TaskStateRefType

该数据类型指向一个可变的TaskStateType数据类型，具体定义

typedef uip_UCHAR_PTR TaskStateRefType;

5.6.2 组成成员

5.6.2.1 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： -

5.6.3 系统服务

5.6.3.1 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

5.6.3.2 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

5.6.3.3 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。

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

5.6.3.4 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

5.6.3.5 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

5.6.3.6 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

5.6.4 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

5.6.5 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。

5.7 范例

在本示例程序中，定义了三个任务。task_0，task_1和task_2。一开始task_0处于运行状态，而task_1和task_2都处于挂起状态。Task_0通过调用ActiveTask (GetTaskID(task_1))激活task_1，并调用TerminateTask()使自己处于挂起状态。Task_1通过调用ChainTask(GetTaskID(task_2))使自己处于悬挂状态，并激活任务task_2。

```
TASK(Task_0)
{
    StatusType status1;
    TaskRefType id1;
    ……
    ActiveTask (GetTaskID(task_1) );
    ……
    TerminateTask();
}

TASK(Task_1)
{
    ……
    ChainTask(GetTaskID(task_2));
}

TASK(Task_2)
{
    ……
    TerminateTask();
}
```

六、中断

中断是一种硬件机制，用于通知 CPU“有个异步事件发生了”。中断一旦被识别，CPU 保存部分（或全部）现场，即部分或全部寄存器的值，跳转到专门的子程序（称作中断服务子程序 ISR），进行中断处理。

6.1 中断处理过程

当中断产生时，处理机将按如下的顺序执行：

- 保存当前处理机状态信息
- 载入异常或中断处理函数到 PC 寄存器
- 把控制权转交给处理函数并开始执行
- 当处理函数执行完成时，恢复处理器状态信息
- 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予以处理，而不必让微处理器连续不断地查询是否有相应事件发生。通过两条特殊指令：关中断和开中断可以让处理器不响应或响应中断。在执行中断服务例程过程中，如果有更高优先级别的中断源触发中断，由于当前处于中断处理上下文环境中，根据不同的处理器构架可能有不同的处理方式：新的中断等待挂起直到当前中断处理离开或打断当前中断处理过程，让处理器响应这个更高优先级的中断源。后面这种情况，一般称之为中断嵌套。在硬实时环境中，前一种情况是不允许发生的，关闭中断响应的时间应尽量的短。在软件处理上，UIP-Kernel 允许中断嵌套，即在一个中断服务例程期间，处理器可以响应另外一个更重要的中断。

6.2 中断服务程序

在本系统中，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：

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。

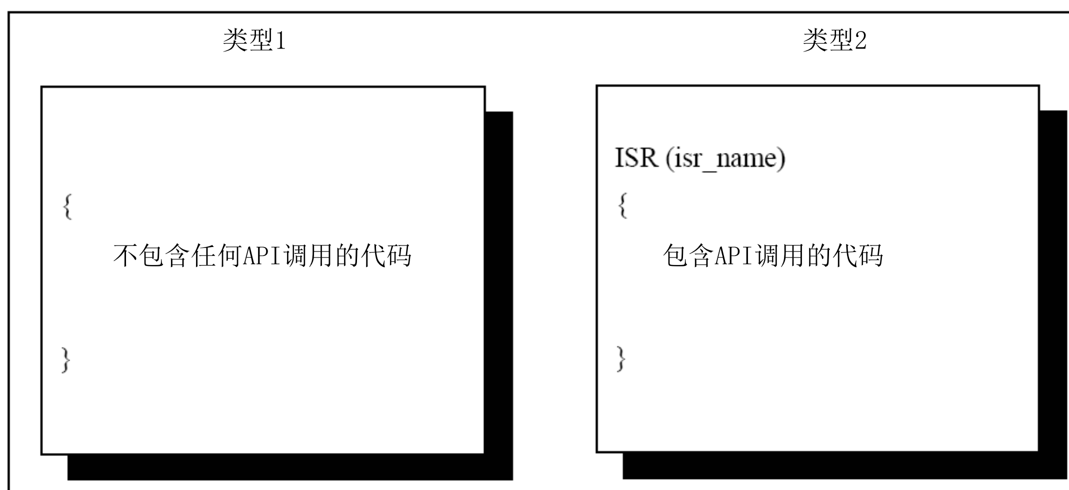

图 6 UIP-Kernel 操作系统的两种 ISR 类型

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。

6.3 中断相关接口

6.3.1 数据类型

没有为UIP-Kernel中断管理定义特殊数据类型。

6.3.2 系统服务

6.3.2.1 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

6.3.2.2 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

6.3.2.3 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

6.3.2.4 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

6.3.2.5 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

6.3.2.6 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

6.3.3 命名规则

实现时，2 型中断的服务程序定义形式如下：

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

关键字 ISR 在系统生成时生成，用于明确地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。

6.4 范例

在该示例程序中，定义了三个任务：Task_0、Task_1和Task_2。

Task_0通过调用DisableAllInterrupts()和EnableAllInterrupts()来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。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。其中DisableAllInterrupts()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，当调用EnableAllIterrupts时恢复保存的状态。

Task_1通过调用SuspendAllInterrupts ()和ResumeAllInterrupts ()来完成一段重要的代码，在这个关键部分中不能调用除了SuspendOSInterrupts / ResumeOSInterrupts和SuspendAllInterrupts / ResumeAllInterrupts两对函数外的任何API服务。其中SuspendAllInterrupts()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，ResumeAllInterrupts ()服务恢复SuspendAllInterrupts保存的所有中断的状态。

Task_2通过调用SuspendOSInterrupts()和ResumeOSInterrupts()来完成一段重要的代码，在这个关键部分中不能调用除了SuspendOSInterrupts / ResumeOSInterrupts和SuspendAllInterrupts / ResumeAllInterrupts两对函数外的任何API服务。其中SuspendOSInterrupts ()服务保存2型中断的识别状态，并关闭对这些中断的识别，ResumeOSInterrupts()服务恢复由SuspendOSInterrupts保存的中断识别状态。

```
#include "uipkernel_api.h"
/*-----------------------------------------------------------------------*/
/* UIP-Kernel事件和任务定义                                               */
/*-----------------------------------------------------------------------*/
DeclareTask(Task_0);            // 声明任务Task_0
DeclareTask(Task_1);            // 声明任务Task_1
DeclareTask(Task_2);            // 声明任务Task_2
/*-----------------------------------------------------------------------*/
```

```
/*  定义任务Task_0
   任务名字：Task_0
   任务类型：扩展任务
   调度方式：抢占式调度                                   */
/*-----------------------------------------------------------------------*/
TASK(Task_0)
{
     ……
     DisableAllInterrupts( );
     /*   关键代码，在这个关键部分中不能调用任何API服务   */
     ……
     EnableAllInterrupts ( );
     ……
}

/*-----------------------------------------------------------------------*/
/*  定义任务Task_1
   任务名字：Task_1
   任务类型：扩展任务
   调度方式：抢占式调度                                   */
/*-----------------------------------------------------------------------*/
TASK(Task_1)
{
     ……
     SuspendAllInterrupts( );
     /*   关键代码，在这个关键部分中不能调用任何API服务   */
     ……
     ResumeAllInterrupts ( );
     ……
}
```

```
/*------------------------------------------------------------------------*/
/*  定义任务Task_2
    任务名字：Task_2
    任务类型：扩展任务
    调度方式：抢占式调度                                                   */
/*------------------------------------------------------------------------*/
TASK(Task_2)
{
    ……
    SuspendOSInterrupts( );
    /*    关键代码，在这个关键部分中不能调用任何API服务    */
    ……
    ResumeOSInterrupts( );
    ……
}
```

七、事件

事件是由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 7 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。

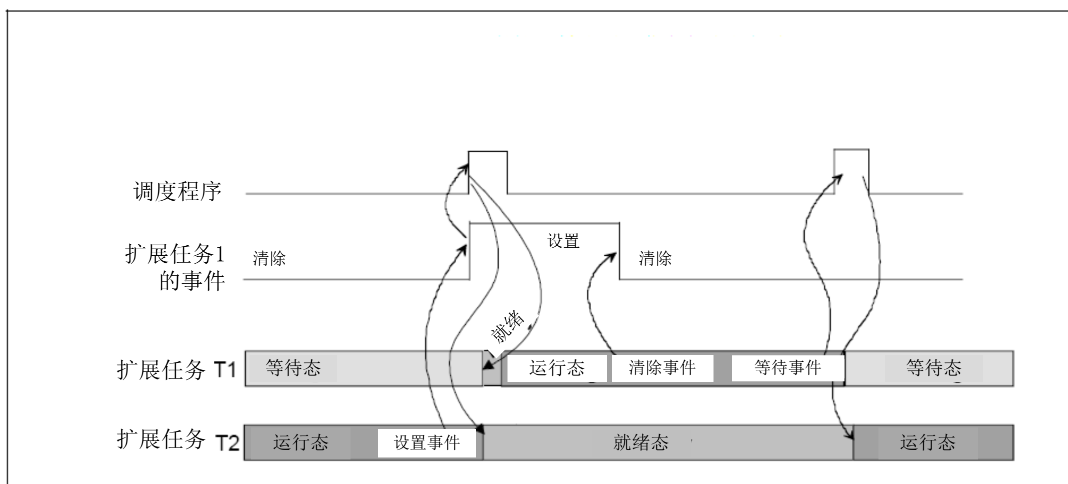

图 7 可抢占扩展任务的同步

图 7 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，调度内核被激活。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 开始重新等待该事件，调度内核继续任务 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 8，其中扩展任务 1 优先级较高）。

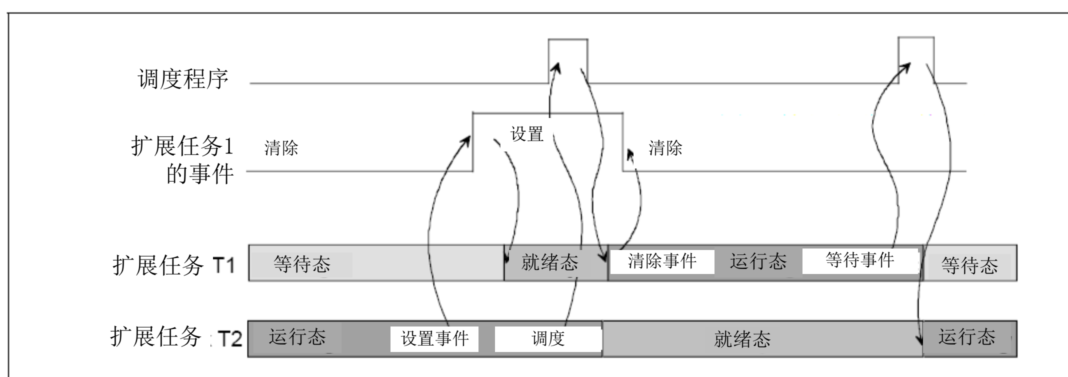

图8 非抢占式扩展任务的同步

7.1 事件相关接口

7.1.1 数据类型

7.1.1.1 EventMaskType

事件屏蔽数据类型，具体定义：

typedef uip_UINT EventMaskType;

7.1.1.1 EventMaskRefType

EventMaskRefType指向一个事件屏蔽，其具体类型定义为：

Typedef EventMaskType * EventMaskRefType;

7.1.2 组成成员

7.1.2.1 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： -

7.1.3 系统服务

7.1.3.1 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

7.1.3.2 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

7.1.3.3 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

7.1.3.4 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

7.2 范例

在该示例程序中，定义了一个事件BarrierEvent和两个任务：LowTask和HighTask，任务LowTask的优先级比任务HighTask的优先级低。最开始，任务HighTask处于运行状态，当执行到WaitEvent(BarrierEvent)语句时，任务HighTask处于等待状态，系统调度任务LowTask运行，当运行到SetEvent(HighTask, BarrierEvent)语句时，使得任务HighTask成为就绪状态，并进行任务的重新调度使得高优先级的任务HighTask重新获得CPU使用权。任务HighTask通过调用ClearEvent(BarrierEvent) 使得自己等待的相应事件（BarrierEvent）被清除。

```
#include "uipkernel_api.h"

/*-----------------------------------------------------------------------*/
/* UIP-Kernel事件和任务定义                                              */
/*-----------------------------------------------------------------------*/
DeclareEvent(BarrierEvent);        //声明一个事件BarrierEvent
DeclareTask(LowTask);              //声明任务LowTask
DeclareTask(HighTask);             //声明任务HighTask

/*-----------------------------------------------------------------------*/
int digits;

/*-----------------------------------------------------------------------*/
void user_1ms_isr_type2(){}

/*-----------------------------------------------------------------------*/
/*  定义任务LowTask，
   任务名字：LowTask
   任务优先级：3
   任务类型：扩展任务
   调度方式：抢占式调度                                      */
```

```
/*-----------------------------------------------------------------------*/
TASK(LowTask)
{
    int hcount, lcount;
    while(1)
   {
        for (hcount = 0; hcount < 10; hcount++)
       {
            for (lcount = 0; lcount < 3200; lcount++) ;
            digits--;
        }
        SetEvent(HighTask, BarrierEvent);
   }
   TerminateTask(); /* or ChainTask() */
}

/*-----------------------------------------------------------------------*/
/*  定义任务HighTask
   任务名字：HighTask
   任务优先级：2
   任务类型：EXTENDED TASK
   调度方式：抢占式调度                                             */
/*-----------------------------------------------------------------------*/
TASK(HighTask)
{
   int hcount, lcount;
   while(1)
  {
       for (hcount = 0; hcount < 10; hcount++)
      {
```

```
            for (lcount = 0; lcount < 3200; lcount++) ;
                digits++;
            }
          WaitEvent(BarrierEvent)
           ClearEvent(BarrierEvent);
           digits=0;
       }
     TerminateTask(); /* or ChainTask() */
}
```

八、资源管理

任何为任务所占有的实体都可称为资源。资源可以是输入/输出设备，例如打印机、键盘及显示器；也可以是一个变量、一个结构或一个数组等。可以被一个以上任务使用的资源叫共享资源。为了防止数据被破坏，每个任务在使用共享资源时，必须独占该资源，叫做互斥。资源管理的作用是协调不同优先级任务对共享资源（例如管理实体（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。

8.1 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 能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。

8.2 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。

在一个任务内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。

8.3 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。

8.4 资源控制块

资源控制块结构的定义如下列代码所示：

```
typedef struct uip_RESOURCE_STRUCT
{
    struct uip_TSK_STRUCT     *uipTaskPtr;
                              /* Pointer to TCB of task that is using this resource */
    struct uip_RESOURCE_STRUCT    *uipNextRes;        /* Link to next resource */
    uip_UINT32    uipResCeilingGrpBit;
                 /*Bit value in task priority group corresponding to task ready priority */
    uip_UCHAR    uipCeilingPrio;              /* Resource's mutex ceiling priority */
    uip_UCHAR    uipResCeilingGrpArrayIndex;
               /* Index value into ready task table corresponding to task ready priority */
    uip_UCHAR    uipResCeilingGrpArrayBit;
                 /* Bit position in ready table corresponding to task ready priority */
    uip_UCHAR    uipTaskPrio;
                       /* Dispatched priority of task that is using this resource */
    uip_UCHAR    uipLock;
                     /* Show the state of resource: 0 not being used; 1 being used */
    uip_UCHAR      padding_1;                 /* Padding byte for ARM */
    uip_UCHAR      padding_2;                 /* Padding byte for ARM */
    uip_UCHAR      padding_3;                 /* Padding byte for ARM */
} uipRes;

Typedef uipRes ResourceType;
```

8.5 资源相关接口

8.5.1 数据类型

（1）ResourceType

该数据类型标识了一个资源，具体定义：

Typedef uip_UINT ResourceType;

8.5.2 组成成员

8.5.2.1 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： -

8.5.3 系统服务

8.5.3.1 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 的调用放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

8.5.3.2 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

8.6 范例

在该示例程序中，定义了一个资源resource1、一个事件Event1和两个任务：LowTask和HighTask，任务LowTask的优先级比任务HighTask的优先级低。最开始，任务HighTask处于运行状态，当执行到WaitEvent(event1)语句时，任务HighTask处于等待状态，系统调度任务LowTask运行，任务LowTask通过调用语句GetResource(resource1)获得资源resource1，然后任务LowTask通过调用语句SetEvent(HighTask, event1)使得任务HighTask成为就绪状态，并进行任务的重新调度使得高优先级的任务HighTask重新获得CPU使用权。

当任务HighTask运行到语句GetResource(resource1)时，由于资源resource1被任务LowTask占有，此时发生任务的重新调度使得低优先级的任务LowTask重新获得CPU使用权，任务LowTask通过调用语句ReleaseResource(resource1)释放资源resource1后，任务HighTask才能获得资源resource1运行。其具体代码如下所示。

```
#include "uipkernel_api.h"
/*  计数器、资源、事件和任务的声明  */
DeclareResource(resource1);          //声明一个资源：resource1
DeclareEvent(event1);                //声明一个事件：event1
DeclareTask(LowTask);                //声明一个任务：LowTask
DeclareTask(HighTask);               //声明一个任务：HighTask

#define COUNT 5000000
int digits;
int lowtaskcount;
int hightaskcount;

TASK(LowTask)
{
     int rcount;
    for (rcount = 0; rcount < COUNT; rcount++);
    GetResource(resource1);
```

```
    for(rcount = 0; rcount < COUNT; rcount++) digits++;
    SetEvent(HighTask, event1);
    for(rcount=0; rcount < COUNT; rcount++) digits--;
    ReleaseResource(resource1);
    TerminateTask();
}

TASK(HighTask)
{
    int rcount;
    for(rcount=0; rcount < COUNT; rcount++) digits++;
    for(rcount=0; rcount < COUNT; rcount++) digits++;
    WaitEvent(event1);
    ClearEvent(event1);
    for(rcount=0; rcount < COUNT; rcount++) digits++;
    GetResource(resource1);
    for (rcount = 0; rcount < COUNT; rcount++);
    ReleaseResource(resource1);
    TerminateTask();
}
```

九、消息

有时需要任务间或中断服务与任务间的通信，这种消息传递称为任务间的通信。任务间消息的传递有两个途径：通过全局变量，或发消息给另一个任务。UIP-Kernel 通过消息实现任务间的通信。

9.1 消息控制块

消息控制块是操作系统用于控制消息的一个数据结构，它会存放消息的一些信息，例如消息ID，消息类型等。在UIP-Kernel实时操作系统中，消息控制块由结构体uipMsg（如下代码中所示）表示，另外一种写法是uipMsgPtr，在C的实现上是指向消息控制块的指针。

```
typedef struct uip_MSG_STRUCT
{
    uip_UCHAR           uipMsgId;                   /* Message's id */
    uip_UCHAR           uipMsgStatus;
                    /* Messages's current state(locked, unlocked, overflow or not etc.) */
    uip_UCHAR           uipMsgType;
                            /* Messages's type(unqueued or queued message) */
    uip_UCHAR           uipMsgFlag;                 /* Flags to be set for notifying */
    uip_UCHAR_PTR       uipMsgPtr;                  /* Point to buffer start address */
    uip_UCHAR_PTR       uipQueueWriteHead;  /* Point to buffer address to be writen */
    uip_UCHAR_PTR       uipQueueEnd;                /* Point to buffer end address */
    uip_UCHAR_PTR       uipQueueReadHead;   /* Point to buffer address to be read */
    uip_UINT            uipMsgSize;                 /* Size of messages(bytes) */
    uip_UINT            uipMsgEvent;                /* Event to be set for notifying */
    VOID                (*uipMsgCallBack)(VOID);
                            /* Callback routine called when sending message */
    uip_UCHAR           uipMsgNotifyTaskID;         /* The ID of task to be nofified */
    uip_UCHAR           uipMsgNotifyType;           /* notification type */
```

```
    uip_UCHAR           padding_2;              /* Padding byte for ARM */
    uip_UCHAR           padding_3;              /* Padding byte for ARM */
} uipMsg;

    typedef uipMsg *                    uipMsgPtr;
```

9.2 消息相关接口

9.2.1 数据类型

9.2.1.1 MsgType

消息类型，其具体类型定义为：

Typedef uip_UCHAR MsgType;

9.2.2 系统服务

9.2.2.1 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 uipMsgType 的值执行不同的操作。若 uipMsgNotifyType = uip_COM_NOTIFICATION_CALLBACK，则在发送该消息时，自动执行(*uipMsgCallBack)(VOID)指定的回调函数；

若 uipMsgNotifyType = uip_COM_NOTIFICATION_FLAG，则在发送消息时将 uipMsgFlag 的值设为 1；

若 uipMsgNotifyType = uip_COM_NOTIFICATION_SETEVENT，则调用 SetEvent（uipMsgNotifyTaskID，uipMsgEvent))为任务 uipMsgNotifyTaskID 设置事件 uipMsgEvent；

若 uipMsgNotifyType = uip_COM_NOTIFICATION_ACTIVATETASK，则调用 ActivateTask(uipMsgNotifyTaskID)激活任务 uipMsgNotifyTaskID。

特性： —

状态：

无错误，E_OK

消息<MsgID>非法，E_OS_ID

消息< uipMsgStatus>= uip_MSG_LOCKED, uip_COM_LOCKED

9.2.2.2 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< uipMsgStatus>= uip_MSG_LOCKED，uip_COM_LOCKED

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

消息缓冲区溢出，uip_COM_OVERFLOW

9.2.2.3 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< uipMsgStatus>= uip_MSG_LOCKED，返回 uip_COM_LOCKED

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

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

描述： 返回< MsgID >指定的消息的状态。

特性： —

9.2.2.4 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 uipMsgFlag

描述：　返回< MsgID >指定的消息的标志。

特性：　—

状态：　无

9.2.2.5 ResetFlag

语法：　　　　StatusType uipResetFlag(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 uipMsgFlag 重置为 0。

特性：　—

状态：

无错误，E_OK

9.3 范例

在该示例程序中，定义了一个消息(MSG_ID_0)、两个任务（TASK_0和TASK_1）。任务TASK_0通过调用SendMessage(MSG_ID_0, &appBuf[0])给任务TASK_1发送消息，由于消息MSG_ID_0的uipMsgNotifyType数据项的值设为uip_COM_NOTIFICATION_SETEVENT，uipMsgNotifyTaskID数据项的值设为TASK_1_ID，uipMsgEvent数据项的值设为APP_EVENT_FOR_TASK，则执行SendMessage函数时，会自动调用SetEvent(TASK_1_ID, APP_EVENT_FOR_TASK)设置任务TASK_1的APP_EVENT_FOR_TASK事件，然后任务TASK_0通过调用WaitEvent(APP_EVENT_0)使自己处于等待状态。

此时发生任务的重新调度，任务TASK_1获得操作系统执行权。由于任务TASK_0调用SendMessage函数时已设置任务TASK_1的APP_EVENT_FOR_TASK事件，则任务TASK_1调用WaitEvent(APP_EVENT_FOR_TASK)时返回E_OK，之后任务TASK_1调用ClearEvent(APP_EVENT_FOR_TASK)清除事件APP_EVENT_FOR_TASK，并调用ReceiveMessage(MSG_ID_0, &appBuf[0])接收消息MSG_ID_0，最后任务TASK_1通过调用SetEvent(TASK_0_ID,APP_EVENT_0)设置任务TASK_0的APP_EVENT_0事件。系统调度任务TASK_0运行，如此反复进行。

为了实现以上功能，编写三个文件：demo9.h、app_cfg9.c和demo9.c。demo9.h中进行一些变量和宏的定义，app_cfg9.c主要对系统中使用的消息、事件和任务进行定义，demo9.c中定义每个任务的任务执行体。具体代码如下所示。

```
/**********************          demo9.h          *****************/
#define      APP_EVENT_0                     0x0001
#define      APP_EVENT_FOR_TASK              0x00400000
#define      MSG_ID_0                        0
#define      TASK_0_ID                       0
#define      TASK_1_ID                       1
#define      TASK_0_READY_PRIO               32
#define      TASK_1_READY_PRIO               20
#define      TASK_0_DISPATCH_PRIO            22
#define      TASK_1_DISPATCH_PRIO            10
```

```
#define        APP_STACK_SIZE                         1000
#define        APP_MSG_SIZE                           14
#define        APP_MSG_LENGTH                         (14*10)

/*********************        app_cfg9.c        *****************/
/*********************        消息定义          ****************/
uip_CHAR AppMsgBuf_1[APP_MSG_LENGTH];
uipMsg AppMsg[] =
{
    {
        MSG_ID_0,                                          /* Message's id */
        0,          /* Messages's current state(locked, unlocked, overflow or not etc.) */
        uip_MSG_QUEUED,          /* Messages's type(unqueued or queued message) */
        0,                                        /* Flags to be set for notifying */
        (uip_UCHAR_PTR)&AppMsgBuf_1[0],           /* Point to buffer start address */
        (uip_UCHAR_PTR)0,                  /* Point to buffer address to be writen */
        ((uip_UCHAR_PTR)&AppMsgBuf_1[0]+sizeof(AppMsgBuf_1)),
                                                  /* Point to buffer end address */
        (uip_UCHAR_PTR)0,                  /* Point to buffer address to be read */
        APP_MSG_SIZE,                                /* Size of messages(bytes) */
        APP_EVENT_FOR_TASK,                      /* Event to be set for notifying */
        (VOID *)0,               /* Callback routine called when sending message */
        TASK_1_ID,                                  /* The ID of task to be nofified */
        uip_COM_NOTIFICATION_SETEVENT,                      /* notification type */
        0,                                             /* Padding byte for ARM */
        0,                                             /* Padding byte for ARM */
    },
};
MsgNumberType AppMsgNumber = 1;
```

```
/**********************          任务定义          ****************/
#pragma DATA_ALIGN(Task_0_Stack, 8);
#pragma DATA_ALIGN(Task_1_Stack, 8);
uip_UINT32 Task_0_Stack[APP_STACK_SIZE];
uip_UINT32 Task_1_Stack[APP_STACK_SIZE];
DeclareTask(TASK_0);
DeclareTask(TASK_1);
uipTask AppTask[] =
{
    {
        &Task_0_Stack[APP_STACK_SIZE-1],         /* Task's stack starting address */
        TASK_0_ID,                               /* Task's ID */
        READY_STATE | uip_TASK_EXTENDED,         /* Taks's execution state */
        TASK_0_READY_PRIO,                       /* Task's ready priority */
#ifdef MixPreemption
        TASK_0_DISPATCH_PRIO,                    /* Task's dispatched priority */
#else
        0,                                       /* padding byte */
#endif
        0,                                       /* The occupaited resources list of this task*/
        TASK_0,                                  /* Task's entry point */
        (APP_STACK_SIZE-1),                      /* Task's stack size */
        0,                                       /* Mask of waited events */
        0,                                       /* Mask of setted events */
        0,                                       /*Bit value in task priority group */
        0,                                       /* Index value into ready task table*/
        0,                                       /* Bit position in ready table*/
        0,                                       /* Task activated number */
        0,                                       /* padding byte */
#ifdef MixPreemption
```

```
        0,                                  /* Bit value in task priority group*/
        0,                                  /* Index value into ready task table*/
        0,                                  /* Bit position in ready table*/
        0,                                  /* padding byte */
        0,                                  /* padding byte */
#endif
        0
    },

    {
        &Task_1_Stack[APP_STACK_SIZE-1],
                                            /* Task's stack starting address */
        TASK_1_ID,                          /* Task's ID */
        READY_STATE | uip_TASK_EXTENDED,    /* Taks's execution state */
        TASK_1_READY_PRIO,                  /* Task's ready priority */
#ifdef MixPreemption
        TASK_1_DISPATCH_PRIO,               /* Task's dispatched priority */
#else
        0,                                  /* padding byte */
#endif
        0,                                  /* The occupaited resources list of this task*/
        TASK_1,                             /* Task's entry point */
        (APP_STACK_SIZE-1),                 /* Task's stack size */
        0,                                  /* Mask of waited events */
        0,                                  /* Mask of setted events */
        0,                                  /* Bit value in task priority group */
        0,                                  /* Index value into ready task table */
        0,                                  /* Bit position in ready table */
        0,                                  /* Task activated number */
        0,                                  /* padding byte */
```

```
#ifdef MixPreemption
        0,                              /* Bit value in task priority group */
        0,                              /* Index value into ready task table */
        0,                              /* Bit position in ready table*/
        0,                              /* padding byte */
        0,                              /* padding byte */
#endif
        0
        },
    };
    TaskNumberType AppTaskNumber = 3;//空闲任务，TASK_0，TASK_1

/**********************         demo9.c         *****************/
TASK(TASK_0)
{
    uip_UCHAR appBuf[APP_MSG_SIZE];
    ……
    while(1)
    {
        ……
        SendMessage(MSG_ID_0, &appBuf[0]);
        WaitEvent(APP_EVENT_0);
        ClearEvent(APP_EVENT_0);
        ……
    }
}

TASK(TASK_1)
{
    ……
```

```
    while(1)
    {
        uip_UCHAR appBuf[APP_MSG_SIZE];
        ……
        if(WaitEvent(APP_EVENT_FOR_TASK) == E_OK)
        {
            ClearEvent(APP_EVENT_FOR_TASK);
            ReceiveMessage(MSG_ID_0, &appBuf[0]);
            ……
            SetEvent(TASK_0_ID,APP_EVENT_0);
        }
    }
}
```

十、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。

10.1 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 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中，一个硬件或软件定时器至少可以提供一个计数器。

10.2 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地分配给：

- 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。

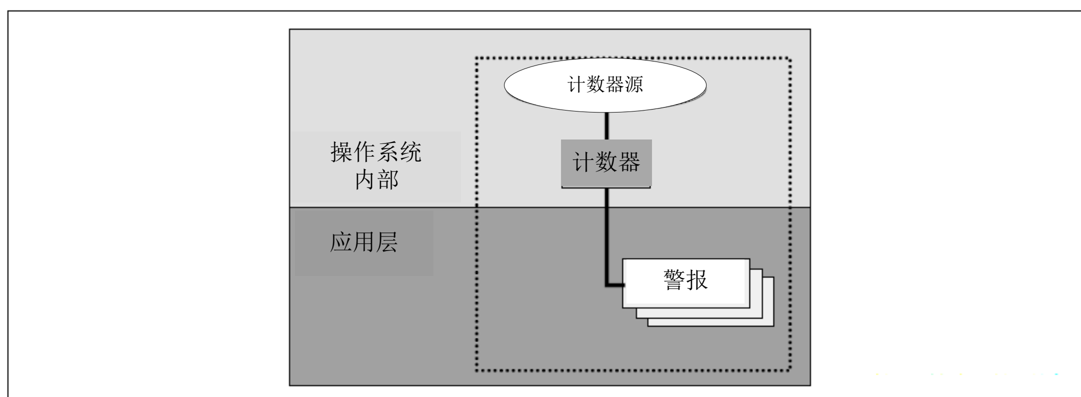

图 9 警报管理的分层模型

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。

10.3 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，其格式为：

```
ALARMCALLBACK(AlarmCallbackRoutineName);
```

例子：

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

报警回调函数处理等级与调度程序或者ISR使用等级相同，由具体实现决定。

10.4 警报相关接口

10.4.1 数据类型

10.4.1.1 TickType

该数据结构代表 tick 计数值

10.4.1.2 TickRefType

指向 TickType 的数据结构

10.4.1.3 AlarmBaseType

用于存储计数器特性的结构体，其结构定义如下列代码所示：

```
typedef struct uip_ALARMBASE_STRUCT
{
    TickType            maxAllowedValue;
    TickType            ticksPerBase;
    TickType            minCycle;
} AlarmBaseType;
typedef AlarmBaseType *                  AlarmBaseRefType;
```

该结构体中各数据项的说明如下：

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）。

10.4.1.4 AlarmBaseRefType

指向 AlarmBaseTpe 的数据结构，其代码定义如下所示：

```
typedef AlarmBaseType *                  AlarmBaseRefType;
```

10.4.1.5 AlarmType

该数据结构代表一个警报对象。

10.4.2 组成成员

10.4.2.1 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： -

10.4.3 系统服务

10.4.3.1 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

10.4.3.2 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

10.4.3.3 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

10.4.3.4 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

10.4.3.5 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

10.4.4 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：

（1） OSMAXALLOWEDVALUE_x

计数器 x 允许的最大 tick 值。

（2） OSTICKSPERBASE_x

计数器 x 达到一个特定的单元时的 tick 数

（3） 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：

（1） OSMAXALLOWEDVALUE

系统计数器允许的最大 tick 值

（2） OSTICKSPERBASE

系统计数器达到特定计数单位时的 tick 数

（3） 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：

（1）OSTICKDURATION

系统计数器一个tick的长度，以纳秒为单位

10.4.5 命名规则

在应用中，报警回调函数的定义形式如下：

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

10.5 范例

在该示例程序中，定义了一个计数器SysTimerCnt、一个警报Alarm1和两个任务：Task_Alarm和Task_Background，在任务Task_Background中定义相对警报Alarm1，每当经过了1000个tick后，任务Task_Alarm被激活。任务Task_Background从不终止。其具体代码如下所示。

```
#include "uipkernel_api.h"
DeclareAlarm(Alarm1);                //声明一个警报Alarm1
DeclareCounter(SysTimerCnt);         //声明一个计数器SysTimerCnt

/*  定义任务Task_Alarm    */
TASK(Task_Alarm)
{
    ……

    TerminateTask();
}

/*   定义一个后台任务Task_Background，该任务从不终止      */
TASK(Task_Background)
{
    // Alarm1(Task_Alarm) is activated with 500msec period after 1000msec
    SetRelAlarm(Alarm1, 1000, 500);
 /*定义一个相对警报Alarm1。当经过了1000个tick后，任务Task_Alarm被激活*/
    while(1);
}
```

十一、回调机制

11.1 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 的每个实现都必须对回调程序的规范给出描述。

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，应该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。

11.2 回调程序相关接口

11.2.1 数据类型

OSServiceIdType

该数据类型代表系统服务的标识

11.2.2 系统服务

11.2.2.1 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。

11.2.2.2 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。

11.2.2.3 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。

11.2.2.4 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。

11.2.2.5 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。

11.2.3 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。

11.2.4 宏

OSErrorGetServiceId：提供发生错误服务的标识。服务标识是OSServiceIdType 类型的。可能的取值是 OSServiceId_xx，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。

十二、应用程序设计建议

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。

12.1 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。

12.1.1 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_l。

```
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。

12.1.2 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。

12.1.3 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。

12.2 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。

12.3 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。

12.3.1 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。

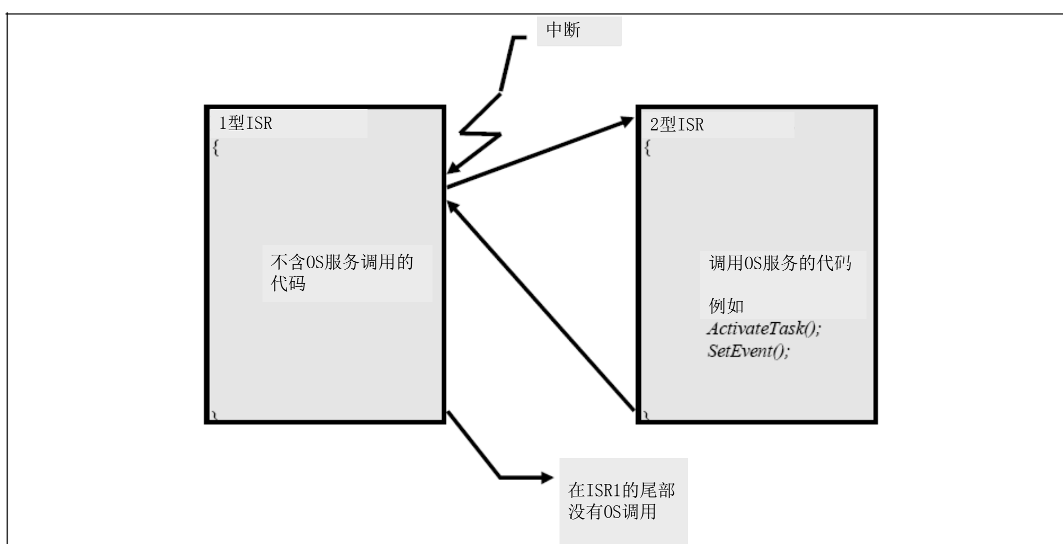

图 10 嵌套的中断

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。

12.3.2 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。

12.4 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。

12.5 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，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。 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。

12.6 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。

12.7 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。

12.8 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。