蹄牛操作系统 TINIUX 简明教程

-- TINIUX 官网提供 http://www.tiniux.org/

概述

蹄牛操作系统 TINIUX 是一个源代码开放的、易于移植的、面向深度嵌入式应用的微内核实时操作系统（RTOS），具有轻量级、低功耗、启动快、可裁剪、可分散加载等优点。主要应用领域为工业控制，智能传感器开发，智能终端，物联网，机器人等。

TINIUX 遵循 MIT 开源许可协议，可以免费在商业产品中使用，不需要公布应用源码，没有任何潜在商业风险。TINIUX 系统最新版本源代码及示例工程的发布网站为：http://www.tiniux.org/

在蹄牛操作系统TINIUX中，每个执行线程都被称为“任务”。对任务的操作是系统的核心，系统的一切处理都是围绕任务展开进行的。下面我们以任务为讲解的起点，逐步展开对TINIUX系统的介绍。

系统任务

任务函数

TINIUX系统的任务是由C语言函数实现的。任务函数原型具有固定的格式，其必须返回void，而且带有一个void指针参数。其函数定义原型如下所示：

```
void TaskFunction( void *pvParameters );
```

每个任务都是在自己权限范围内的一个小程序，该程序具有特定的入口地址。在TINIUX系统中，不允许任务从其实现函数中返回——它们绝不能有一条“return”语句，也不能执行到函数末尾。因此，一般情况下，任务函数会运行在一个无限循环中，不会退出。如果一个任务不再需要，须通过任务控制函数OSTaskDelete()显式地将其删除。

一个任务函数可以用来创建多个任务（实例）——创建出的任务均是独立的执行实例，拥有自己的栈空间和局部变量(栈变量)，即在任务函数内部定义的变量。

下面是一个典型的任务函数示例：

```
void TaskFunction( void *pvParameters )
{
```

```
    /* 在任务函数中可以像普通函数一样定义变量。用此任务函数创建的每个
     * 任务实例都有一个属于自己的iLocalPara变量。但如果被定义为
     * static类型，则所有的任务实例将会共享这个变量。 */
    int iLocalPara = 0;
    for( ;; )
    {
        /* 完成任务功能的代码将放在这里。 */
    }
    /* 若任务函数会前面循环退出，则任务实例的生命周期即将结束。任务必须
     * 在函数运行完之前显示的删除。传入OS_NULL参数表示删除的是当前任务 */
    OSTaskDelete( OS_NULL );
}
```

示例代码1、典型任务函数结构示例

任务状态

应用程序可以包含多个任务。如果运行应用程序的微控制器是单核的，那么在任意给定时间，实际上只会有一个任务被执行。这就意味着任务至少有两种状态，即运行状态和非运行状态（非运行状态又划分为若干个子状态，后面会逐个展开讲述）。当某个任务处于运行态时，处理器正在运行它的实例。当任务处于非运行态时，该任务进行休眠，它的所有状态都被保存到系统为其分配的栈空间上，以便调度器把该任务“切换”为运行态时，可以恢复执行。当任务恢复执行时，其将精确地从离开运行态时正准备执行的那一条指令开始执行。

任务从非运行态转移到运行态被称为“切入”。相反，任务从运行态转移到非运行态被称为“切出”。TINIUX的调度器是能让任务切入切出的唯一实体。

任务创建

任务的创建使用了TINIUX的API接口函数OSTaskCreate()。这是所有API函数中最复杂的一个。该函数原型定义如下所示：

```
OSTaskHandle_t OSTaskCreate(OSTaskFunction_t    pxTaskFunction,
                            void*               pvParameter,
                            const uOS16_t       usStackDepth,
                            uOSBase_t           uxPriority,
                            sOS8_t*             pcTaskName);
```

函数OSTaskCreate()的参数及返回值如下表所示：

参数名	描述
pxTaskFunction	任务是一个永不退出的C函数实例，通常是一个死循环。参数pxTaskFunction为指向任务的实现函数的指针(函数名)。
pvParameter	任务函数接受一个指向void的指针(void*)。pvParameter的值即是传递到任务函数实例中的值。
usStackDepth	当任务创建时，内核会分为每个任务分配属于其自己的唯一栈空间。 usStackDepth值用于告诉内核为它分配多大的栈空间。 这个值指定的是栈空间可以保存多少个字(word)，而不是多少个字节(byte)。比如说，如果是32位宽的栈空间，传入的usStackDepth值为100，则将会分配400 字节的栈空间(100 * 4bytes)。栈深度乘以栈宽度的结果千万不能超过一个size_t类型变量所能表达的最大值。 应用程序通过定义常量OSMINIMAL_STACK_SIZE来决定空闲任务任用的栈空间大小。在TINIUX为微控制器架构提供的Demo应用程序中，赋予此常量的值是对所有任务的最小建议值。 如果你的任务会使用大量栈空间，那么你应当赋予一个更大的值。 没有任何简单的方法可以决定一个任务到底需要多大的栈空间。计算出来虽然是可能的，但大多数用户会先简单地赋予一个自认为合理的值，然后利用TINIUX提供的特性来确保分配的空间既不欠缺也不浪费。
uxPriority	指定任务执行的优先级。优先级的取值范围可以从最低优先级0到OSHIGHEAST_PRIORITY-1。 OSHIGHEAST_PRIORITY是一个由用户定义的常量。优先级并没有上限(仅受限于优先级的数据类型和系统的有效内存空间)，但最好使用实际需要的最小数值以避免内存浪费。 系统会对参数uxPriority进行检测，如果其数值超过了OSHIGHEAST_PRIORITY-1，将会自动修正为最大有效优先级，即OSHIGHEAST_PRIORITY-1。
pcTaskName	具有描述性的任务名。这个参数不会被TINIUX系统使用。其仅仅用于辅助调试。 应用程序可以通过定义常量OSNAME_MAX_LEN来定义任务名的最大长度——包括'\0'结束符。如果传入的字符串长度超过了这个最大值，字符串将会自动被截断。 若用户不希望使用此参数，设置为OS_NULL即可。
返回值	有两个可能的返回值： 1. OSTaskHandle_t类型的任务句柄。通过该句柄变量，可以在系统调用接口函数中调整任务的相关参数，如改变任务优先级，或者删除任务。 2. OS_NULL。由于内存堆空间不足，TINIUX无法分配足够的空间来保存任务结构数据和任务栈，因此无法创建任务。

示例1：创建任务

本例演示了创建并启动两个任务的必要步骤。这两个任务只是周期性地打印输出字符串，采用原始的空循环方式来产生周期延迟。两者在创建时指定了相同的优先级，并且在实现上除输出的字符串外完全一样，下面是具体的实现代码：

```
static void Task1Function( void *pvParameters )
{
    const char *pcString = "Task 1 is running\r\n";
    volatile unsigned int i;
    const int   LOOP_COUNT = 10000;
    /* 和大多数任务一样，该任务处于一个死循环中。 */
    for( ;; )
    {
        /* 打印输出字符串。*/
        printf( pcString );
        /* 延迟，以产生一个周期 */
        for( i = 0; i < LOOP_COUNT; i++ )
        {
            /* 空循环是最原始的延迟实现方式。在循环中不做任何事情。 */
        }
        OSSchedule();
    }
}

static void Task2Function( void *pvParameters )
{
    const char *pcString = "Task 2 is running\r\n";
    volatile unsigned int i;
    const int   LOOP_COUNT = 10000;
    /* 和大多数任务一样，该任务处于一个死循环中。 */
    for( ;; )
    {
        /* 打印输出字符串。*/
        printf( pcString );
        /* 延迟，以产生一个周期 */
        for( i = 0; i < LOOP_COUNT; i++ )
        {
            /* 空循环是最原始的延迟实现方式。在循环中不做任何事情。 */
        }
        OSSchedule();
    }
}
```

```
int main( void )
{
    //Initialize the parameter in TINIUX
    OSInit();

    /* 创建第一个任务。较实用方式是在应用程序中保存函数OSTaskCreate()的返回值，
     * 以获取任务的句柄，并以句柄是否为OS_NULL检测任务创建成功与否。 */
    OSTaskCreate(  Task1Function,          /* 指向任务函数的指针 */
                   OS_NULL,                    /* 没有任务参数 */
                   OSMINIMAL_STACK_SIZE,  /* 栈深度 */
                   OSLOWEAST_PRIORITY+1,  /* 此任务优先级 */
                   "Task1");              /* 任务的文本名字 */
    /* 采用同样方法创建第二个任务 */
    OSTaskCreate(Task2Function, OS_NULL, OSMINIMAL_STACK_SIZE,
           OSLOWEAST_PRIORITY+1, "Task2");
    /* 启动系统，任务开始执行 */
    OSStart();
    /* 如果一切正常，main()函数不应该执行到这里。如果执行到这里，很可能是
     * 内存堆空间不足导致空闲任务无法创建。 */
    for( ;; );
}
```

示例代码2、创建任务并启动系统

本例的运行输出如下：

Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
……

从中看到两个任务似乎在同时运行，实际上这两个任务运行在同一个处理器上，所以不可能会同时运行。事实上这两个任务都迅速地进入与退出运行态。由于这两个任务运行在同一个处理器上，所以会平等共享处理器时间。真实的执行流程如下图1所示：

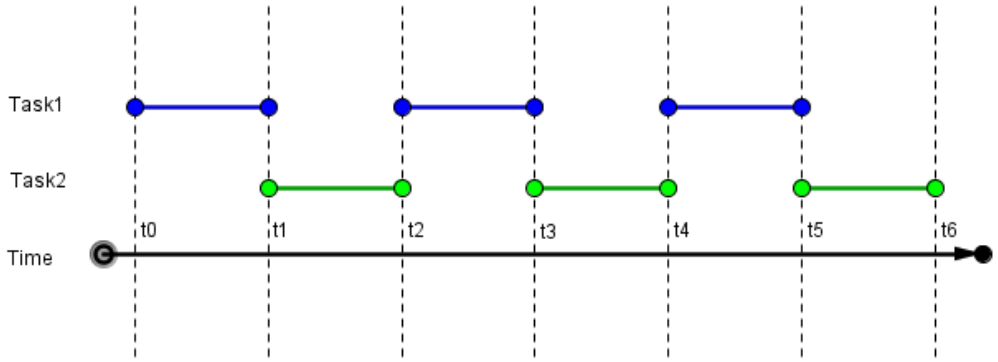

图1、示例1运行流程

上图1中底部的箭头表示系统运行的时间轴。彩色的线段表示在每个时间点上正在运行的任务，t0时刻系统的调度器选中任务1执行，t0与t1之间运行的是任务1，t1与t2之间运行的是任务2。

在任何时刻只可能有一个任务处于运行态。当一个任务进入运行态后(切入)，另一个任务就会进入非运行态(切出)。

在例**1**中，main()函数在系统启动之前先完成两个任务的创建。当然也可以从一个任务中创建另一个任务。我们可以先在main()中创建任务1，然后在任务1中创建任务2。调整之后的代码如下所示。这样，在系统启动之前，任务2还没有被创建，但是整个程序运行的输出结果还是相同的。

```
static void Task1Function( void *pvParameters )
{
    const char *pcString = "Task 1 is running\r\n";
    volatile unsigned int i;
    const int   LOOP_COUNT = 10000;

    /* 在此处创建第二个任务 */
    OSTaskCreate(Task2Function, OS_NULL, OSMINIMAL_STACK_SIZE,
            OSLOWEAST_PRIORITY+1, "Task2");

    /* 和大多数任务一样，该任务处于一个死循环中。 */
    for( ;; )
    {
        /* 打印输出字符串. */
        printf( pcString );
        /* 延迟，以产生一个周期 */
        for( i = 0; i < LOOP_COUNT; i++ )
        {
            /* 空循环是最原始的延迟实现方式。在循环中不做任何事情。 */
        }
        OSSchedule();
```

```
    }
}
```

示例代码3、在一个任务中创建另一个任务

示例2. 使用任务参数

在例**1**中创建的两个任务几乎完全相同，唯一的区别就是打印输出的字符串。为了避免同一功能的重复定义，可以通过采用一个任务函数创建两个实例的方式实现。这时任务函数的参数可以用来传递各自打印输出的字符串。

示例代码**4**中的任务函数(TaskFunction)将代替了示例代码**2**中的两个任务函数(Task1Function与Task2Function)。这个函数的任务参数被强制转化为char*以得到任务需要打印输出的字符串。

```
static void TaskFunction( void *pvParameters )
{
    const char *pcString = ( char* )pvParameters;
    volatile unsigned int i;
    const int   LOOP_COUNT = 10000;
    /* 和大多数任务一样，该任务处于一个死循环中。 */
    for( ;; )
    {
        /* 打印输出字符串. */
        printf( pcString );
        /* 延迟，以产生一个周期 */
        for( i = 0; i < LOOP_COUNT; i++ )
        {
            /* 空循环是最原始的延迟实现方式。在循环中不做任何事情。 */
        }
        OSSchedule();
    }
}
```

示例代码4、创建多个任务实例的任务函数

尽管现在只有一个任务函数代码(TaskFunction)，但是可以创建多个任务实例。每个任务实例都可以在TINIUX调度器的控制下单独运行。

传递给系统API接口函数OSTaskCreate()的参数pvPrameters用于传入字符串指针。如示例代码**5**所示。

```
/* 定义将要通过任务参数传递的字符串。定义为const，且不是在栈空间上，以保证任务执行时也有效。 */
static const char *pcTextForTask1 = "Task 1 is running\r\n";
static const char *pcTextForTask2 = "Task 2 is running\t\n";
```

```
int main( void )
{
    //Initialize the parameter in TINIUX
    OSInit();

    /* 创建第一个任务。较实用方式是在应用程序中保存函数OSTaskCreate()的返回值，
     * 以获取任务的句柄，并以句柄是否为OS_NULL检测任务创建成功与否。 */
    OSTaskCreate(  TaskFunction,          /* 指向任务函数的指针 */
                   (void*)pcTextForTask1, /* 任务函数参数 */
                   OSMINIMAL_STACK_SIZE,  /* 栈深度 */
                   OSLOWEAST_PRIORITY+1,  /* 此任务优先级 */
                   "Task1");              /* 任务的文本名字 */
    /* 采用同样方法创建第二个任务 */
    OSTaskCreate(TaskFunction, (void*)pcTextForTask2, OSMINIMAL_STACK_SIZE,
           OSLOWEAST_PRIORITY+1, "Task2");
    /* 启动系统，任务开始执行 */
    OSStart();
    /* 如果一切正常，main()函数不应该执行到这里。如果执行到这里，很可能是
     * 内存堆空间不足导致空闲任务无法创建。 */
    for( ;; );
}
```

示例代码5、例2中创建多任务的main()函数

例**2**的运行输出结果与例**1**完全一样。

任务优先级

系统API函数OSTaskCreate()的参数uxPriority为创建的任务赋予了一个初始优先级。任务优先级可以通过系统API接口函数OSTaskSetPriority()进行修改。

优先级的最大值通过宏定义OSHIGHEAST_PRIORITY进行配置，该宏定义可以在工程配置文件“OSPreset.h”中修改。TINIUX系统并没有限定这个常量的最大值，但这个值越大，则内核花销的内存空间就越多，因此建议将此常量设为够用的最小值，系统建议此配置变量不超过32。

对于如何为任务指定优先级，TINIUX并没有强加任何限制。多个任务也可以共享同一个优先级——以保证最大设计弹性。当然，如果需要的话，也可以为每个任务指定唯一的优先级，但这不是强制要求的。用户可以根据任务的重要程度为任务灵活分配优先级，重要性越大的任务，为其分配的优先级应当越高。以汽车控制系统为例，响应刹车指令的任务优先级要高于响应用户听歌/娱乐等指令的优先级。（当然，实时性要求更高的事件，则应当配置硬件中断服务函数进行响应。）

在TINIUX系统中，优先级数值越小表示任务的优先级越低，优先级0表示最低优先级，最低优先级在系统中用宏定义OSLOWEAST_PRIORITY表示。用户任务可使用的有效范围为OSLOWEAST_PRIORITY到OSHIGHEAST_PRIORITY-1。

系统调度器会在所有可运行的任务中选择最高优先级的任务，并使其进入运行态。如果被选中的优先级上具有多个任务，系统调度器会选择等待时间最长的任务首先切入执行。

需要明确指出的是，TINIUX系统是抢占式实时操作系统，被系统调度器切入的任务会一直运行下去，除非该任务主动让出执行权限（如等待某一个同步事件），或者被更高优先级的任务抢断执行；这种行为方式在上面的例子中可以明显看出来。两个测试任务被创建在同一个优先级上，任务运行过程中均通过调用接口函数OSSchedule()，主动进行系统调度切换。所以每个任务都有了执行机会。

在运行的过程中，调度器需要定期的对系统中任务状态进行检测。TINIUX系统为此设置了一个周期性的中断，称为心跳（Tick，亦称为时钟滴答，本文中一律称为时钟心跳）中断。时间片的长度由心跳中断的频率（心跳频率）决定，心跳频率通过宏定义OSTICK_RATE_HZ进行配置，该常量可以在工程配置文件“OSPreset.h”中修改。如果OSTICK_RATE_HZ设为1000(HZ)，则时间片长度为1ms。可以将图1进行扩展，将调度器自身的执行时间在整个执行流程中体现出来。请参见图2。

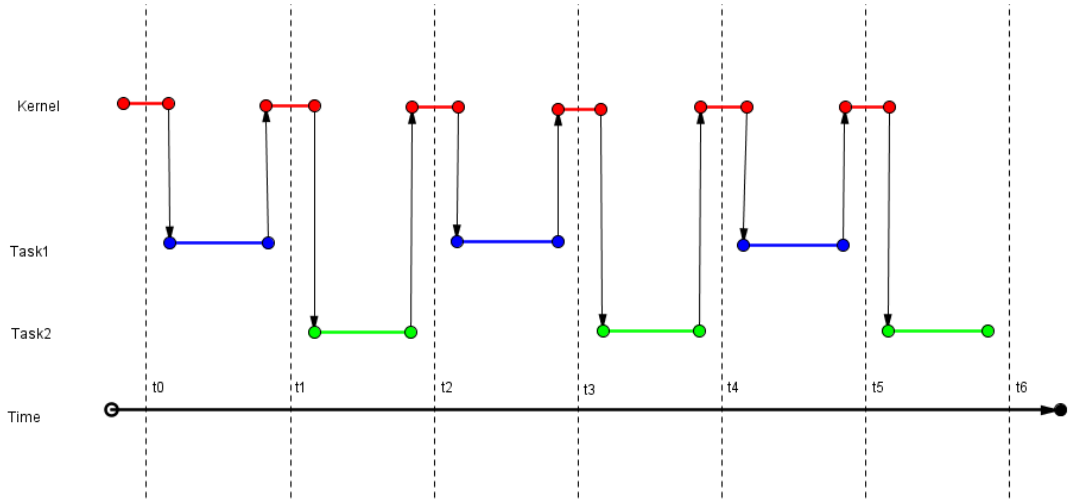

图2对执行流程进行扩展以显示心跳中断的执行

图**2**中红色的线段代表内核本身在运行，值得注意的是，图中为了表达直观，将内核运行时间拉长，实际执行过程中，内核调度任务时只需数条指令即可完成。内核到任务之间的黑色箭头表示任务到内核，内核再到另一个任务的执行顺序。

示例3. 优先级实验

调度器总是在可运行的任务中进行选择，选取具有最高优级的任务，并使其进入运行态。

到目前为止的示例程序中，两个任务都创建在相同的优先级上。所以这两个任务轮番进入和退出运行态。本例将改变例**2**其中一个任务的优先级，看一下倒底会发生什么。现在第一个任务创建在优先级OSLOWEAST_PRIORITY+1上，而另一个任务创建在优先级OSLOWEAST_PRIORITY+2上。创建这两个任务的代码参见示例代码**6**。这两个任务的实现函数没有任何改动，还是通过空循环产生延迟来周期性打印输出字符串。

```
/* 定义将要通过任务参数传递的字符串。定义为const，且不是在栈空间上，以保证任务执行时也有效。 */
static const char *pcTextForTask1 = "Task 1 is running\r\n";
static const char *pcTextForTask2 = "Task 2 is running\t\n";

int main( void )
{
    //Initialize the parameter in TINIUX
    OSInit();

    /* 创建第一个任务。较实用方式是在应用程序中保存函数OSTaskCreate()的返回值，
     * 以获取任务的句柄，并以句柄是否为OS_NULL检测任务创建成功与否。 */
    OSTaskCreate(   TaskFunction,           /* 指向任务函数的指针 */
                    (void*)pcTextForTask1, /* 没有任务参数 */
                    OSMINIMAL_STACK_SIZE,  /* 栈深度 */
                    OSLOWEAST_PRIORITY+2,  /* 此任务优先级 */
                    "Task1");              /* 任务的文本名字 */
    /* 采用同样方法创建第二个任务 */
    OSTaskCreate(TaskFunction, (void*)pcTextForTask2, OSMINIMAL_STACK_SIZE,
            OSLOWEAST_PRIORITY+1, "Task2");
    /* 启动系统，任务开始执行 */
    OSStart();
    /* 如果一切正常，main()函数不应该执行到这里。如果执行到这里，很可能是
     * 内存堆空间不足导致空闲任务无法创建。 */
    for( ;; );
}
```

示例代码6、创建不同优先级的任务

下面是例3的运行结果：

Task 1 is running
Task 1 is running
Task 1 is running
Task 1 is running
Task 1 is running

Task 1 is running
……

系统调度器总是选择具有最高优先级的可运行任务来执行。任务1的优先级比任务2高，并且总是可运行的，因此任务1是唯一一个一直处于运行态的任务。而任务2不可能进入运行态，所以不可能输出任务2相关的字符串。这种情况我们称为任务2的执行时间被任务1“饿死”了。

任务1之所以总是可运行，是因为其不会等待任何事件——它要么在空循环里打转，要么往终端打印字符串。

下图展现了示例3的运行流程，可以看出任务2根本没有机会得到执行。

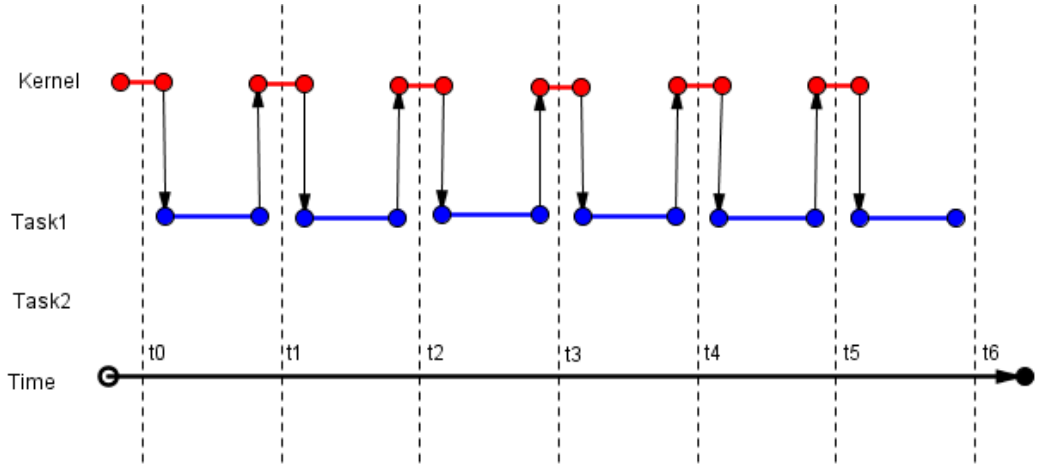

图3、示例3运行流程

需要说明的是，TINIUX系统API接口函数中指定的时间总是以心跳周期（Ticks）为单位。宏定义OSM2T()可用于将以毫秒为单位的时间值转换为系统Ticks的计数值。

心跳计数(tick count)值表示的是从调度器启动开始的心跳中断的总数，并假定心跳计数器不会溢出。用户程序在指定延迟周期时不必考虑心跳计数溢出的问题，因为时间连贯性会在TINIUX的内核中进行管理。

非运行态

到目前为止，所有的示例中，每个任务都只顾不停地处理自己的事情，而没有其它任何事件需要等待——由于它们不需要等待，所以会一直处于运行态。这种“不停处理”的模式限制了任务的应用性，因为它们只能被创建在最低优先级上。如果它们运行在其它优先级上，那么优先级比它们低的任务将永远得不到执行的机会。

为了使创建的任务切实有用，我们需要通过某种事件来驱动任务。这样，任务只能在事件发生后工作(处理)，而在事件没有发生时处于非运行态。

调度器会在所有能够进入运行态的任务中进行选择，选中具有最高优先级的任务。一个高优先级但不能够运行的任务是不会被调度器选中的，而代之以另一个优先级虽然较低但能够运行的任务。因此，采用事件驱动任务的意义就在于任务可以被创建在许多不同的优先级上，而最高优先级任务不会把所有的低优先级任务“饿死”。

阻塞状态

如果一个任务正在等待某个事件，则称这个任务处于“阻塞态”。阻塞态是非运行态的一个子状态。

任务进入阻塞态以等待以下两种不同类型的事件：
1. 定时(时间相关)事件——这类事件可以是延迟到期或是绝对时间到点。如某个任务进入阻塞态以延迟10ms。
2. 同步事件——源于其它任务或中断的事件。如某个任务进入阻塞态以等待队列中有数据到来。同步事件涵盖所有板级范围内的事件类型。TINIUX系统中的消息队列（MsgQ），信号量（Semaphore）和互斥锁（Mutex）都可以用来实现同步事件。后面会对同步事件进一步讲述。

任务可以在进入阻塞态等待同步事件时指定超时时间，这样可以实现阻塞状态下同时等待两种类型的事件。如某个任务可以等待队列中有数据到来，但最多只等10ms。如果10ms内有数据到来，或是10ms过去了还没有数据到来，这两种情况下该任务都将退出阻塞态。

挂起状态

“挂起(suspended)”也是非运行状态的子状态。处于挂起状态的任务对调度器而言是不可见的。在TINIUX系统中，永久等待某一同步事件，直至该事件有效的任务会被设置为挂起状态。

就绪状态

如果任务处于非运行状态，但既没有阻塞也没有挂起，则这个任务处于就绪(ready，准备或就绪)状态。处于就绪态的任务能够被调度器选中执行，但只是“准备(ready)”运行，而尚未运行。

把任务的非运行状态扩充后，TINIUX系统中状态转移图如下所示：

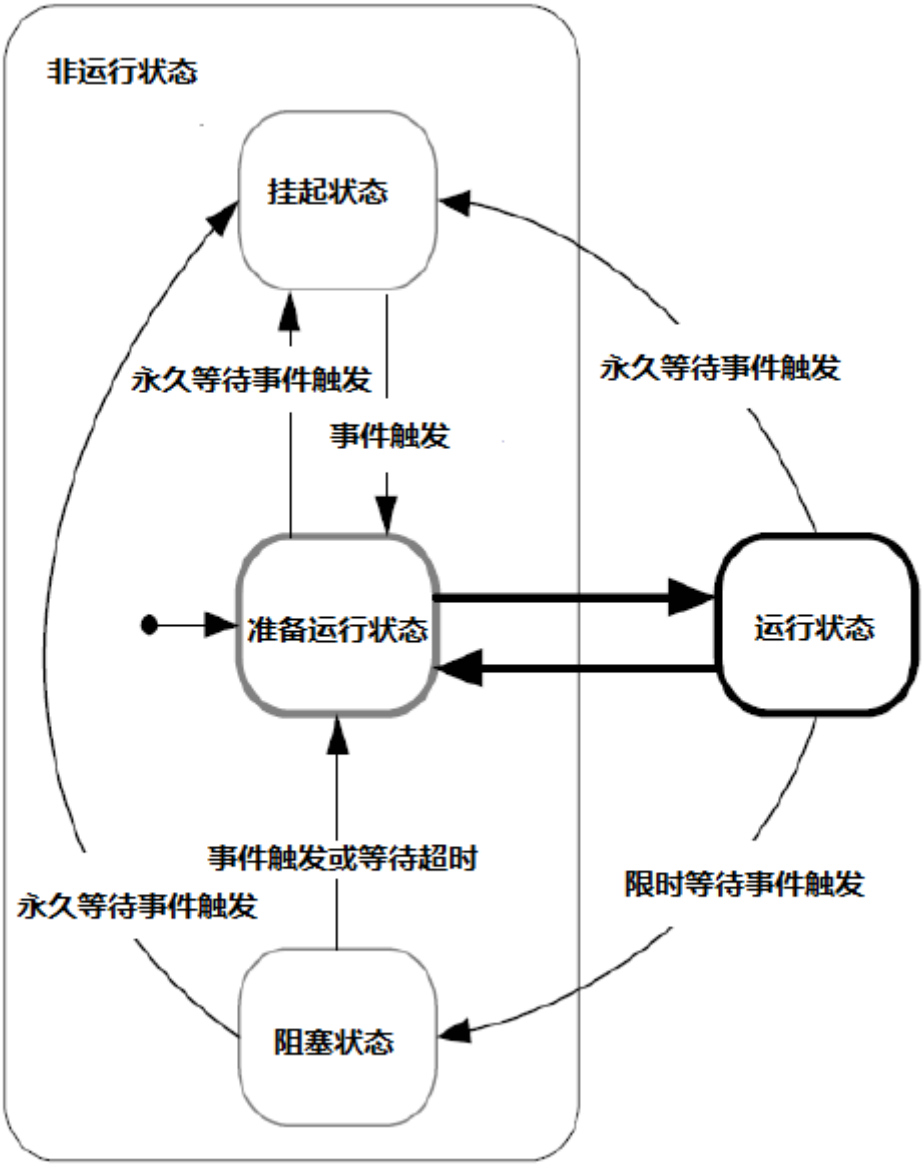

图4、TINIUX中任务状态转移图

示例 4. 利用阻塞态实现延迟执行

之前的示例中所有创建的任务都是“周期性”的——它们延迟一个周期时间，打印输出字符串，再一次延迟，如此周而复始。而产生延迟的方法也相当原始地使用了空循环——不停地查询并递增一个循环计数直至计到某个指定值。例3明确的指出了这种方法的缺点。一直保持在运行态中执行空循环，可能将其它任务饿死。

其实查询执行的方式不仅低效，而且还有其它方面的缺点。在查询执行的过程中，任务实际上并没有做任何有意义的事情，但它依然会耗尽相关处理时间，对处理器的计算资源造成浪费。例**4**通过调用系统API函数OSTaskSleep()来代替空循环，对这种“不良行为”进行纠正。OSTaskSleep()的函数原型定义如下：

```
void OSTaskSleep( const uOSTick_t uxTicksToDelay )
```

函数OSTaskSleep()的参数含义如下表所示：

参数名	描述
uxTicksToDelay	延迟多少个心跳周期。调用该延迟函数的任务将进入阻塞态，经延迟指定的心跳周期数后，再转移到就绪态。 举个例子，当某个任务调用OSTaskSleep(100)时，心跳计数的绝对值为10,000，则该任务将保持在阻塞态，直到心跳计数计到10,100。 宏定义OSM2T()可以用来将以毫秒为单位的时间值转换为以心跳周期(Ticks)为单位的时间值。

而新的任务函数如下示例代码7所示：

```
static void TaskFunction( void *pvParameters )
{
    const char *pcString = ( char* )pvParameters;
    volatile unsigned int i;
    const int   LOOP_COUNT = 10000;
    /* 和大多数任务一样，该任务处于一个死循环中。 */
    for( ;; )
    {
        /* 打印输出字符串. */
        printf( pcString );
        /* 延迟一个循环周期。调用OSTaskSleep()以让任务在延迟期间保持在阻塞态。
         * 延迟时间以心跳周期为单位，OSM2T()可以用来在毫秒和心跳周期
         * 之间相换转换。本例设定200毫秒的循环周期。 */
        OSTaskSleep( OSM2T(200) );
    }
}
```

示例代码7、使用OSTaskSleep()取代空循环

尽管两个任务实例还是创建在不同的优先级上，但现在两个任务都可以得到执行。下面是例 4 运行后的结果：

Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
……

图**5**所示的执行流程可以解释为什么此时不同优先级的两个任务竟然都可以得到执行。图中为了简便，忽略了内核自身的执行时间。

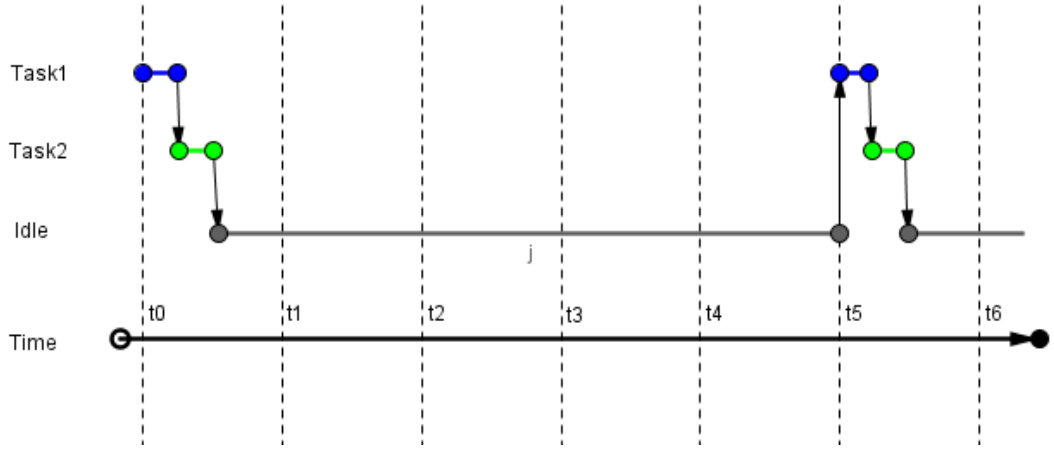

图 5、采用 OSTaskSleep()后的运行流程

本例只改变了两个任务的实现方式，并没有改变其功能。对比图**5**与图**1**可以清晰地看到本例以更有效的方式实现了任务的功能。

图**1**展现的是当任务采用空循环进行延迟时间的执行流程——结果就是任务总是处于运行状态，占用了大量的处理器周期。从图**5**中的执行流程中可以看到，任务在整个延迟周期内都处于阻塞状态，只在完成实际工作的时候才占用处理器时间(本例中任务的实际工作只是简单地打印输出一条信息)。

在图**5**所示的情形中，任务离开阻塞态后，仅仅执行了一个心跳周期的很小部分时间，然后又再次进入阻塞态。所以大多数时间都没有一个应用任务可运行(即没有应用任务处于就绪态)，因此没有应用任务可以被选择进入运行态。这种情况下，空闲任务得以执行。空闲任务可以获得的执行时间量，是系统处理能力裕量的一个度量指标。

在任务实例中采用系统API接口函数OSTaskSleep()之后，每个任务在返回就绪态之前，都会经过阻塞状态。

空闲任务

例**4**中创建的任务大部份时间都处于阻塞态。这种状态下，用户创建的所有任务都不可运行，所以也不能被调度器选中。

但处理器总是需要代码来执行的——所以至少要有确保系统中有一个任务处于运行态。为了保证这一点，当调用API函数OSStart()启动系统时，会自动创建一个空闲任务。空闲任务是一个非常短小的无限循环，可以永远运行。空闲任务总是处于运行状态或者就绪状态。

空闲任务运行在最低优先级(优先级0)，保证一旦有更高优先级的任务进入就绪态，空闲任务就会立即切出运行态。这一点可以在图5的t5时刻看出来，当任务1退出阻塞态时，空闲任务立即切换出来，以供任务1执行，相当于任务1抢占了空闲任务。抢占是自动发生的，并不需要通知被抢占的任务。

空闲任务绝不能被阻塞或挂起，以任何方式阻塞空闲任务都可能导致系统没有任务再次能够进入运行态！

空闲任务还有一个比较重要的功能，在某一个任务被删除后，其负责回收工作，回收该任务占用的内核资源（如内存空间等）。

删除任务

任务可以使用API函数OSTaskDelete()删除自己或其它任务。任务被删除后就不复存在了，不会再进入运行态。

空闲任务的责任是要将分配给已删除任务的内存释放掉。需要说明一点的是，只有内核为任务分配的内存空间才会在任务被删除后自动回收。任务自己占用的内存或资源需要由应用程序显式地释放。

删除任务的API函数原型定义如下所示：

```
void OSTaskDelete( OSTaskHandle_t pxTaskToDelete );
```

消息队列

基于TINIUX的应用程序由一系列独立的任务构成——每个任务都是具有独立权限的函数实例。这些独立的任务之间需要通过相互通信以提供必要的系统同步功能。

TINIUX 中所有的通信与同步机制都是基于队列实现的。

队列的特性

数据存储

队列可以保存有限个具有确定长度的数据单元。队列可以保存的最大单元数目被称为队列的“深度”。在队列创建时需要设定其深度和每个单元的大小。

通常情况下，队列被作为FIFO(先进先出)使用，即数据由队列尾写入，从队列首读出。当然，由队列首写入也是可能的。

往队列写入数据时通过字节拷贝把数据复制存储到队列中；从队列读出数据时需把队列中的数据拷贝删除。图**6**展现了队列的写入与读出过程，以及读写操作对队列中数据的影响。

可被多任务存取

队列是具有自己独立权限的内核对象，并不属于或赋予任何任务。所有任务都可以向同一队列写入或读出数据。在一般的使用场景中，同一个数据队列常常会由多个任务写入，而由一个任务读出（当然，由多个任务读取同一个数据队列也是允许的，用户需要自行设计相关权限，如读取次序等，以避免资源竞争）。

读队列时阻塞

当某个任务试图读一个队列时，其可以指定一个阻塞超时时间。在这段时间中，如果队列为空，该任务将保持阻塞状态以等待队列数据有效。当其它任务或中断服务例程向其等待的队列中写入了数据时，该任务将自动由阻塞态转移为就绪态。当等待的时间超过了指定的阻塞时间时，即使队列中没有收到有效数据，任务也会自动从阻塞态转移为就绪态。

由于队列可以被多个任务读取，所以对单个队列而言，也可能有多个任务处于阻塞状态以等待队列数据有效。这种情况下，一旦队列数据有效，只会有一个任务被解除阻塞，这个任务就是所有等待任务中优先级最高的任务。而如果所有等待

任务的优先级相同，那么被解除阻塞的任务则是等待时间最长的任务。

写队列时阻塞

同读队列一样，任务也可以在写队列时指定一个阻塞超时时间。这个时间在队列满时会发挥作用，是任务进入阻塞态以等待队列空间有效的最长时间。

由于队列可以被多个任务写入，所以对单个队列而言，也可能有多个任务处于阻塞状态以等待队列空间有效。这种情况下，一旦队列空间有效，只会有一个任务被解除阻塞，这个任务就是所有等待任务中优先级最高的任务。而如果所有等待任务的优先级相同，那么被解除阻塞的任务将则是等待时间最长的任务。

队列的读写过程如下图所示：

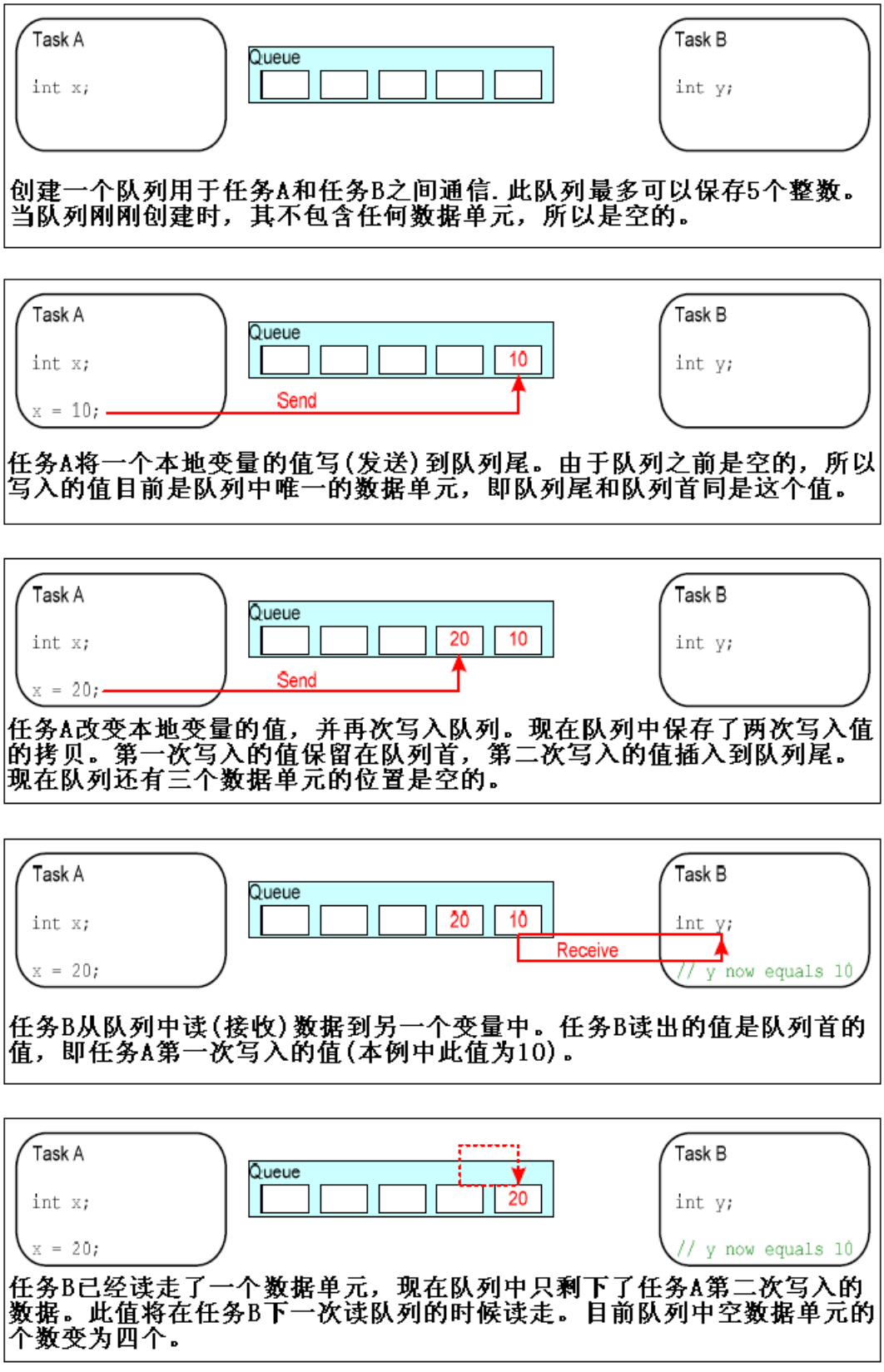

图 **6**、队列读写过程示例

使用消息队列

系统API函数OSMsgQCreate()

消息队列在使用前必须先被创建。系统API函数OSMsgQCreate()用于创建一个消息队列，并返回一个OSMsgQHandle_t句柄，以便于对其创建的队列进行引用。

当创建队列时，系统TINIUX会从堆空间中为其分配相应的内存空间。该内存空间用于存储队列的数据结构以及队列中包含的数据单元。如果内存堆中没有足够的空间来创建队列，OSMsgQCreate()将返回OS_NULL。OSMsgQCreate()的函数原型定义如下：

```
OSMsgQHandle_t OSMsgQCreate(const uOSBase_t uxQueueLength, const uOSBase_t uxItemSize);
```

函数OSMsgQCreate()的参数及返回值含义如下表所示：

参数名	描述
uxQueueLength	队列能够存储的最大单元数目，即队列深度。
uxItemSize	队列中数据单元的长度，以字节为单位。
返回值	OS_NULL表示队列创建失败，系统没有足够的堆空间分配给队列。 非OS_NULL值表示队列创建成功，返回OSMsgQHandle_t类型的消息队列句柄，此句柄应当记录下来用于对消息队列的读写操作。

系统API函数OSMsgQSend()与OSMsgQSendToHead()

如同函数名字面意思所期望的一样，OSMsgQSend()用于将数据发送到消息队列。在TINIUX系统中，默认是发送到消息队列尾部的；系统API接口函数OSMsgQSendToTail()完全等同于OSMsgQSend()。而OSMsgQSendToHead()用于将数据发送到队列首部。

切记不要在中断服务例程中调用OSMsgQSend()或OSMsgQSendToHead()函数。系统提供中断安全版本的OSMsgQSendFromISR()与OSMsgQSendToHeadFromISR()用于在中断服务中实现相同的功能。

相关函数原型定义如下所示：

```
sOSBase_t OSMsgQSend( OSMsgQHandle_t MsgQHandle, constvoid * const pvItemToQueue, uOSTick_t xTicksToWait);
sOSBase_t OSMsgQSendToHead( OSMsgQHandle_t MsgQHandle, constvoid * const pvItemToQueue, uOSTick_t xTicksToWait);
```

函数OSMsgQSend()等的参数含义如下表所示：

参数名	描述
MsgQHandle	目标队列的句柄。这个句柄是系统API函数OSMsgQCreate()创建该消息队列时的返回值。
pvItemToQueue	发送数据的指针。其指向将要复制到目标队列中的数据单元。由于在创建队列时设置了队列中数据单元的长度，所以会从该指针指向的空间复制对应长度的数据到队列的存储区域。
xTicksToWait	阻塞超时时间。如果在发送时队列已满，这个时间即是任务处于阻塞态等待队列空间有效的最长等待时间。 如果xTicksToWait设为0，并且队列已满，则OSMsgQSend()与OSMsgQSendToHead()均会立即返回。 阻塞时间是以系统心跳周期为单位的。宏定义OSM2T()可以用来把毫秒为单位的时间转换为系统的Ticks计数值。 如果把xTicksToWait 设置为OSPEND_FOREVER_VALUE，那么阻塞等待将没有超时限制，直至队列具有有效的数据空间才恢复执行。
返回值	有两个可能的返回值: 1. OS_TRUE 表示数据被成功发送到队列中。 如果设定了阻塞超时时间(xTicksToWait非0)，在函数返回之前，任务将被转移到阻塞态以等待消息队列空间有效。在超时到来前能够将数据成功写入到队列，函数则会返回OS_TRUE。 2. OS_FALSE 如果由于队列已满而无法将数据写入，则将返回OS_FALSE。 如果设定了阻塞超时时间（xTicksToWait非0），在函数返回之前任务将被转移到阻塞态以等待消息队列空间有效。但直到超时也没有其它任务或是中断服务例程读取消息队列而腾出空间，函数则会返回OS_FALSE。

系统API函数OSMsgQReceive()

OSMsgQReceive()用于从队列中接收(读取)数据单元。接收到的单元同时会从队列中删除。切记不要在中断服务例程中调用OSMsgQReceive()。中断安全版本的API函数为OSMsgQReceiveFromISR()，下面是函数OSMsgQReceive()的原型定义：

```
sOSBase_t OSMsgQReceive( OSMsgQHandle_t MsgQHandle, void * const pvBuffer,
uOSTick_t xTicksToWait);
```

函数OSMsgQReceive()的参数含义如下表所示：

参数名	描述

MsgQHandle	消息队列的句柄。这个句柄是系统API函数OSMsgQCreate()创建该消息队列时的返回值。
pvBuffer	接收缓存指针。其指向一段内存区域，用于接收从队列中拷贝来的数据。 数据单元的长度在创建队列时就已经被设定，所以该指针指向的内存区域大小应当足够保存一个数据单元。
xTicksToWait	阻塞超时时间。如果在接收时队列为空，则这个时间是任务处于阻塞状态以等待队列数据有效的最长等待时间。 如果xTicksToWait设为0，并且队列为空，则OSMsgQReceive()会立即返回。 阻塞时间是以系统心跳周期为单位的。宏定义OSM2T()可以用来把毫秒为单位的时间转换为系统的Ticks计数值。 如果把xTicksToWait 设置为OSPEND_FOREVER_VALUE，那么阻塞等待将没有超时限制，直至队列中收到有效数据。
返回值	有两个可能的返回值: 1. OS_TRUE 表示成功地从队列中读到数据。 如果设定了阻塞超时时间(xTicksToWait非0)，在函数返回之前任务将被转移到阻塞态以等待队列数据有效，在超时到来前能够从队列中成功读取数据，函数则会返回OS_TRUE。 2. OS_FALSE 如果在读取时由于队列已空而没有读到任何数据，返回OS_FALSE。 如果设定了阻塞超时时间（xTicksToWait非0），在函数返回之前任务将被转移到阻塞态以等待队列数据有效。但直到超时也没有其它任务或是中断服务例程往队列中写入数据，函数则会返回OS_FALSE。

示例5. 读队列时阻塞

本例示范创建一个队列，由多个任务往队列中写数据，以及从队列中把数据读出。这个队列创建出来保存int型数据单元。往队列中写数据的任务没有设定阻塞超时时间，而读队列的任务设定了超时时间。

往队列中写数据的任务的优先级低于读队列任务的优先级。这意味着队列中永远不会保持超过一个的数据单元。因为一旦有数据被写入队列，读队列任务立即解除阻塞，抢占写队列任务，并从队列中接收数据，同时数据从队列中删除—队列再一次变为空队列。

示例代码**8**展现了写队列任务的代码实现。这个任务被创建了两个实例，一个不停地往队列中写数值100，而另一个实例不停地往队列中写入数值200。任务的入口参数被用来为每个实例传递各自的写入值。

```
static void SenderTask( void *pvParameters )
```

```
{
    int iValueToSend;
    sOSBase_t xStatus;
    /* 该任务会被创建两个实例，所以写入队列的值通过任务入口参数传递,这种方式使得
     * 每个实例使用不同的值。队列创建时指定其数据单元为int型，所以把入口参数强制
     * 转换为数据单元要求的类型 */
    iValueToSend = ( int ) pvParameters;
    /* 和大多数任务一样，本任务也处于一个死循环中 */
    for( ;; )
    {
        /* 往队列发送数据
        第一个参数是要写入的队列。队列在系统启动前就在main()函数中创建了。
        第二个参数是被发送数据的地址，本例中即变量iValueToSend的地址。
        第三个参数是阻塞超时时间 - 当队列满时，任务转入阻塞状态以等待队列空间有效。
        本例中没有设定超时时间，因为此队列决不会保持有超过一个数据单元的机会，
        所以也决不会满。 */
        xStatus = OSMsgQSend( MsgQHandle, &iValueToSend, 0 );
        if( xStatus != OS_TRUE )
        {
            /* 发送操作由于队列满而失败，这必然存在错误，因为本例的队列不可能满。*/
            printf("Could not send to the queue.\r\n" );
        }
        /* 允许其它发送任务执行。 OSSchedule()通知调度器现在就切换到其它任务，
         * 而不必等到本任务的时间片耗尽 */
        OSSchedule();
    }
}
```

示例代码 8、例 5 中的写队列任务实现代码

示例代码**9**展现了读队列任务的代码实现。读队列任务设定了100毫秒的阻塞超时时间，所以会进入阻塞态以等待队列数据有效。一旦队列中数据单元有效，或者即使队列数据无效但等待时间超过100毫秒，此任务将会解除阻塞。在本例中，将永远不会出现100毫秒超时，因为有两个任务在不停地往队列中写数据。

```
static void ReceiverTask( void *pvParameters )
{
    /* 声明变量，用于保存从队列中接收到的数据。 */
    int iReceivedValue;
    sOSBase_t xStatus;
    const uOSTick_t xTicksToWait = OSM2T(100);
    /* 本任务依然处于死循环中。 */
    for( ;; )
    {
        /* 从队列中接收数据
```

```
        第一个参数是被读取的队列。队列在系统启动前就在main()函数中创建了。
        第二个参数是保存接收到的数据的缓冲区地址，本例中即变量iReceivedValue的地址。
        此变量类型与队列数据单元类型相同，所以有足够的大小来存储接收到的数据。
        第三个参数是阻塞超时时间 – 当队列空时，任务转入阻塞状态以等待队列数据有效。
        宏定义OSM2T()用来将100毫秒绝对时间转换为以系统心跳为单位的时间值。*/
        xStatus = OSMsgQReceive( MsgQHandle, &iReceivedValue, xTicksToWait );
        if( xStatus == OS_TRUE )
        {
            /* 成功读出数据，打印出来。 */
            printf("Received = %d", iReceivedValue );
        }
        else
        {
            /* 等待100ms也没有收到任何数据。
            必然存在错误，因为发送任务在不停地往队列中写入数据 */
            printf("Could not receive from the queue.\r\n" );
        }
    }
}
```

示例代码 9、例 5 中的读队列任务实现代码

示例代码**10**包含了main()函数的实现。其在系统启动之前创建了一个队列和三个任务。尽管对任务的优先级的设计使得队列实际上在任何时候都不可能多于一个数据单元，本例代码还是创建了一个可以保存最多5个int型值的队列。

```
OSMsgQHandle_t MsgQHandle = OS_NULL;
int main( void )
{
    //Initialize the parameter in TINIUX
    OSInit();

    /* 新建队列最多保存5个值，每个数据单元都有足够的空间来存储一个int型变量 */
    MsgQHandle = OSMsgQCreate( 5, sizeof( int ) );
    if( MsgQHandle != OS_NULL )
    {
        /* 创建两个写队列任务实例，任务入口参数用于传递发送到队列的值。所以一个实例
         * 不停地往队列发送100，而另一个任务实例不停地往队列发送200。
         * 两个任务的优先级都设为1。 */
        OSTaskCreate( SenderTask, ( void * ) "100", OSMINIMAL_STACK_SIZE,
                OSLOWEAST_PRIORITY+1, "Sender1" );
        OSTaskCreate( SenderTask, ( void * ) "200", OSMINIMAL_STACK_SIZE,
                OSLOWEAST_PRIORITY+1, "Sender2" );
        /* 创建一个读队列任务实例。其优先级设为2，高于写任务优先级 */
        OSTaskCreate( ReceiverTask, OS_NULL, OSMINIMAL_STACK_SIZE,
```

```
                OSLOWEAST_PRIORITY+2, "Receiver" );
        /* 启动调度器，任务开始执行 */
        OSStart();
    }
    else
    {
        /* 队列创建失败*/
    }
    /* 如果一切正常，main()函数不应该会执行到这里。但如果执行到这里，
     * 很可能是内存堆空间不足导致空闲任务无法创建。 */
    for( ;; );
}
```

示例代码 10、例 5 中的 main()函数实现代码

写队列任务在每次循环中都调用OSSchedule()。OSSchedule()通知调度器立即进行任务切换，而不必等到当前任务的时间片耗尽。任务调用OSSchedule()等效于其自愿放弃运行态。由于本例中两个写队列任务具有相同的任务优先级，所以一旦其中一个任务调用了OSSchedule()，另一个任务将会得到执行，调用OSSchedule()的任务转移到就绪态，同时另一个任务进入运行态。这样就可以使得这两个任务轮翻地往队列发送数据。下面是例5的输出结果：

Received = 100
Received = 200
Received = 100
Received = 200
Received = 100
Received = 200
……

大型数据单元传输

如果队列存储的数据单元尺寸较大，那最好是利用队列来传递数据的指针而不是对数据本身在队列上一字节一字节地拷贝进或拷贝出。传递指针无论是在处理速度上还是内存空间利用上都更有效。但是，当你利用队列传递指针时，一定要十分小心地做到以下两点：

1. 指针指向的内存空间的所有权必须明确

当任务间通过指针共享内存时，应该保证不会有多个任务同时修改共享内存中的数据，或是以其它行为方式使得共享内存数据无效或产生一致性问题。原则上，共享内存在其指针发送到队列之前，其内容只允许被发送任务访问；共享内存指针从队列中被读出之后，其内容亦只允许被接收任务访问。

2. 指针指向的内存空间必须有效

如果指针指向的内存空间是动态分配的，只应该有一个任务负责对其进行内存释

放。当这段内存空间被释放之后，就不应该有任何一个任务再访问这段空间。切忌用指针访问任务栈上分配的空间。因为当栈帧发生改变后，栈上的数据将不再有效。

信号量与中断

嵌入式实时系统需要对整个系统环境产生的事件作出反应。举个例子，以太网外围部件收到了一个数据包(事件)，需要送到TCP/IP 协议栈进行处理(反应)。更复杂的系统需要处理来自各种源头产生的事件，这些事件对处理时间和响应时间都有不同的要求。

TINIUX并没有为设计人员提供具体的事件处理策略，但是提供了一些特性使得设计人员采用的策略可以得到实现，而实现方式不仅简单，而且具有可维护性。

必须说明的是，系统中只有以“FromISR”结束的API函数或宏才可以在中断服务例程中使用。

延迟中断处理

采用信号量同步

信号量可以在某个特殊的中断发生时，让系统中处于阻塞状态的任务解除阻塞，相当于让任务与中断信号进行关联。这样就可以把中断事件中耗时较大的工作转移到相应的关联任务中完成，而中断服务例程(ISR)中只需快速处理少部份工作。这样一来，中断处理相当于被“推迟”到一个“处理”任务中。

如果某个中断处理要求特别紧急，其延迟处理任务的优先级可以设为最高，以保证延迟处理任务随时都可抢占系统中的其它任务。这样，延迟处理任务就成为其对应的ISR退出后第一个执行的任务，在时间上紧接着ISR执行，相当于所有的处理都在ISR中完成一样。

延迟处理任务对一个信号量进行带阻塞性质的“Pend”调用，意思是进入阻塞态以等待事件发生。当事件发生后，ISR对同一个信号量进行“Post”操作，使得延迟处理任务解除阻塞，从而确保事件在延迟处理任务中得到相应的处理。

“Pend”和“Post”信号量从概念上讲，在不同的应用场合有不同的含义。在经典的信号量术语中，获取信号量等同于一个 P()操作，而给出信号量等同于一个 V()操作。

在中断同步的情形下，信号量可以看作是一个深度为N（系统中默认设置为0xFF）的队列。延迟处理任务调用OSSemPend()时，等效于带阻塞时间地读取队列，如果队列为空的话任务则进入阻塞态。当事件发生后，ISR简单地通过调用

OSSemPostFromISR()放置一个令牌(信号量)到队列中。这也使得延迟处理任务切出阻塞态，并移除令牌。当任务完成处理后，再次读取队列，发现队列为空，又进入阻塞态，等待下一次事件发生。

系统API函数OSSemCreate()

TINIUX中各种信号量的句柄都存储在OSSemHandle_t类型的变量中。在使用信号量之前，必须先创建它。创建信号量使用OSSemCreate()API函数, 函数原型定义如下所示：

```
OSSemHandle_t  OSSemCreate( const uOSBase_t uxInitialCount);
```

函数OSSemCreate()的返回值含义如下表所示：

参数名	描述
uxInitialCount	信号量创建后的初始有效信号数量；
返回值	OS_NULL，则信号量创建失败。 非OS_NULL，OSSemHandle_t类型的信号量句柄；

系统API函数OSSemPend()

“Pend”一个信号量意为“获取”或“接收”信号量。只有当信号量有效的时候才可以被获取。在经典信号量术语中，OSSemPend()等同于一次P()操作。

信号量可以通过调用函数OSSemPend()来等待并获取。但OSSemPend()不能在中断服务例程中调用。下面是OSSemPend()函数的原型定义：

```
sOSBase_t OSSemPend( OSSemHandle_t SemHandle, uOSTick_t xTicksToWait);
```

函数OSSemPend()的参数及返回值含义如下表所示：

参数名	描述
SemHandle	信号量句柄，函数OSSemCreate()创建信号量时的返回值，信号量在使用前必须先创建。
xTicksToWait	阻塞超时时间。任务进入阻塞态以等待信号量有效的最长时间。 如果xTicksToWait 为0，则OSSemPend()在信号量无效时会立即返回。 阻塞时间是以系统心跳周期为单位的。宏定义OSM2T()可以用来把毫秒为单位的时间转换为系统的Ticks计数值。 如果把xTicksToWait 设置为OSPEND_FOREVER_VALUE，那么阻塞等待将没有超时限制。
返回值	有两个可能的返回值: 1. OS_TRUE，表示成功获得信号量。 如果设定了阻塞超时时间(xTicksToWait非0)，在函数返回之前任务将被转移到阻塞态以等待信号量有效。如果在超时到来前信号量变为有效，亦可被成功获取，返回OS_TRUE。 2. OS_FALSE，表示未能获得信号量。 如果设定了阻塞超时时间（xTicksToWait 非0），在函数返回之前任务将被转移到阻塞态以等待信号量有效。但直到超时信号量也没有变为有效，所以不会获得信号量，返回OS_FALSE。

系统API函数OSSemPostFromISR()

TINIUX支持的信号量都可以通过调用OSSemPostFromISR()给出。OSSemPostFromISR()是OSSemPost()的特殊形式，专门用于中断服务例程中。下面为其函数原型定义：

```
sOSBase_t OSSemPostFromISR( OSSemHandle_t SemHandle );
sOSBase_t OSSemPost( OSSemHandle_t SemHandle);
```

函数OSSemPost()的参数含义如下表所示：

参数名	描述
SemHandle	信号量句柄，函数OSSemCreate()创建信号量时的返回值，信号量在使用前必须先创建。
返回值	有两个可能的返回值: 1. OS_TRUE，信号量“Post”成功 2. OS_FALSE，信号量无法“Post”，则返回OS_FALSE。

示例 6 使用信号量作为任务之间的同步事件

本例示范创建一个信号量，同时创建两个任务实例，其中一个任务等待（Pend）信号量的有效信号，等待时没有设定等待超时，这样这个任务会一直阻塞下去，直到接收到有效的信号；另一个任务负责释放（Post）信号量，释放完信号量后就进入休眠延迟状态，延迟时间设置为200ms。

释放信号量的任务的优先级低于等待信号量任务的优先级。这意味一旦信号量有效，等待信号量的任务立即解除阻塞，抢占释放信号量的任务。

```
//定义一个信号量句柄类型的变量，用于保存创建的信号量。
OSSemHandle_t   SemHandle = OS_NULL;

static void PostTask( void *pvParameters )
{
    sOSBase_t xStatus;

    /* 和大多数任务一样，本任务也处于一个死循环中 */
    for( ;; )
    {
        xStatus = OSSemPost(SemHandle);
        if( xStatus != OS_TRUE )
        {
            /* 若信号量无效，会导致信号量释放异常。 */
            printf("Could not post semaphore.\r\n" );
```

```
        }
        /* 调用函数OSTaskSleep()进行周期性的休眠。防止其不间断的释放信号量 */
        OSTaskSleep( OSM2T(200) );
    }
}

static void PendTask( void *pvParameters )
{
    sOSBase_t xStatus;
    /* 本任务依然处于死循环中。 */
    for( ;; )
    {
        /* 等待信号量，变量OSPEND_FOREVER_VALUE表示永久等待，直到接收有效的信号*/
        xStatus = OSSemPend(SemHandle, OSPEND_FOREVER_VALUE);
        if( xStatus == OS_TRUE )
        {
            /* 成功读出数据，打印出来。 */
            printf("Semaphore is posted" );
        }
        else
        {
            /* 信号量无效*/
            printf("Semaphore is invalid.\r\n" );
        }
    }
}

int main( void )
{
    //Initialize the parameter in TINIUX
    OSInit();

    /* 创建信号量，用于任务之间的同步 */
    SemHandle = OSSemCreate(0);
    if( SemHandle != OS_NULL )
    {
        /* 创建一个释放信号量的任务，其优先级为1，该任务周期性的释放信号量。 */
        OSTaskCreate( PostTask, OS_NULL, OSMINIMAL_STACK_SIZE,
                OSLOWEAST_PRIORITY+1, "Post" );
        /* 创建一个等待信号量的任务。其优先级设为2，高于释放信号量任务的优先级 */
        OSTaskCreate( PendTask, OS_NULL, OSMINIMAL_STACK_SIZE,
                OSLOWEAST_PRIORITY+2, "Pend" );
        /* 启动调度器，任务开始执行 */
        OSStart();
```

```
    }
    else
    {
        /* 信号量创建失败*/
    }
    /* 如果一切正常，main()函数不应该会执行到这里。但如果执行到这里，
     * 很可能是内存堆空间不足导致空闲任务无法创建。 */
    for( ;; );
}
```

示例代码 11、使用信号量进行任务之间同步

注意：上例中，若释放信号量的代码处于中断服务函数中，则 OSSemPost()函数需要更改为中断安全的 API 函数 OSSemPostFromISR()；

互斥锁

访问一个被多任务共享，或是被任务与中断共享的资源时，需要采用“互斥”技术以保证数据在任何时候都保持一致性。这样做的目的是要确保任务从开始访问资源就具有排它性，直至这个资源又恢复到完整状态。

TINIUX提供了多种特性用以实现互斥，但是最好的互斥方法（如果可能的话，任何时候都当如此）还是通过精心设计应用程序，尽量不要共享资源，或者是每个资源都通过单任务访问。

挂起(锁定)中断

中断锁定模式是提供互斥功能的一种非常原始的实现方法。锁定时仅仅是简单地把高于某一优先级的中断全部关掉。因为TINIUX系统中的任务切换只可能在中断中完成，所以任务实例在调用OSIntLock()后，系统屏蔽了中断功能，锁定后的程序就可以一直保持运行，直到退出中断锁定。

系统要求锁定后的程序只能占用很短的时间，否则会影响应用程序对中断的响应时间。必须强调的是，在每次调用OSIntLock()进行中断锁定后，须尽快地配套调用一个OSIntUnlock()退出中断锁定，这个两个函数永远是配对使用的。

系统内核维护了一个中断锁定嵌套深度计数，中断锁定函数允许嵌套使用。锁定区只会在嵌套深度为0时才会真正解锁退出，即在每个调用的OSIntLock()函数后，都配套调用了OSIntUnlock()函数，中断锁定功能才会退出，系统中断功能才会恢复运行。

挂起(锁定)调度器

可以通过挂起系统调度器的方式来保护一段代码不被其他任务打断。挂起调度器也被称为锁定调度器。这种方式下，系统不再进行任务调度的操作，可以确保被保护的代码一直运行下去，而不被其他任务干扰，不过这时系统中断仍是使能的。

如果待保护的区域运行耗时较长，简单采用中断锁定的方式会影响系统对中断的响应性能，这时可以考虑采用挂起调度器的方式实现。但是调度器解锁却是一个相对较长的操作。所以需要结合实际情况评估哪种是最佳方式。调度器锁定与解锁的系统接口函数如下所示：

系统API函数OSScheduleLock()

系统API函数OSScheduleLock()的原型定义如下：

```
void OSScheduleLock( void );
```

系统提供API函数OSScheduleLock()来锁定调度器。调度器锁定后会停止任务切换而不用关中断。如果在调度器锁定过程中，某个中断要求进行任务切换，则这个请求也会被挂起，直到调度器被解锁后才会得到执行。

注意：在调度器处于锁定状态时，不能调用TINIUX系统的API接口函数。

系统API函数OSScheduleUnlock()

系统API函数OSScheduleUnlock()的原型定义如下：

```
sOSBase_t OSScheduleUnlock( void );
```

OSScheduleUnlock()返回值如下表所示：

参数名	描述
返回值	在调度器锁定过程中，任务切换请求也会被挂起，直到调度器被解锁后才会得到执行。如果有任务切换请求发生在函数OSScheduleUnlock()调用前，则该函数返回OS_TRUE。在其它情况下，返回OS_FALSE。

系统内核维护了一个调度器锁定计数，因此嵌套调用OSScheduleLock()和OSScheduleUnlock()是允许的，调度器只会在嵌套深度计数为0时才会被解锁，即在每个调用的OSScheduleLock()函数后，都配套调用了OSScheduleUnlock()函数，这时调度器才会恢复调度工作。

互斥锁

互斥锁是一种特殊的信号量，用于控制在两个或多个任务间访问共享资源。在用于互斥的场合，互斥锁从概念上可看作是与共享资源关联的令牌。一个任务想要合法地访问资源，其必须先成功地得到该资源对应的令牌(成为令牌持有者)。

当令牌持有者完成资源使用后，其必须马上归还令牌。只有归还了令牌，其它任务才可能持有该令牌，也才可能安全地访问该共享资源。一个任务除非持有了令牌，否则不允许访问共享资源。

虽然互斥锁与信号量之间具有很多相同的特性。两者间最大的区别在于信号量在被获得之后所发生的事情：用于互斥的信号量必须归还, 用于同步的信号量通常是完成同步之后便丢弃，不再归还。基于这一特性，互斥锁的占用与释放必须在同一个任务中，而信号量的使用却没有这个要求。

在TINIUX系统中，互斥锁的占用被称作“锁定”，互斥锁的释放被称作“解锁”。

系统API接口函数OSMutexCreate()

互斥锁是一种特殊的信号量。在TINIUX系统中，互斥锁句柄的定义为OSMutexHandle_t。互斥锁在使用前必须先创建。创建一个互斥锁需要使用系统API接口函数OSMutexCreate()，该函数原型定义如下所示：

```
OSMutexHandle_t OSMutexCreate();
```

函数OSMutexCreate()的返回值如下表所示：

参数名	描述
返回值	OS_NULL，表示互斥锁创建失败。 原因是内存堆空间不足导致TINIUX无法为互斥锁分配数据空间。 非OS_NULL，表示互斥锁创建成功。 该返回值应当保存起来作为该互斥锁的句柄。

互斥锁锁定的API函数原型定义如下所示：

```
sOSBase_t OSMutexLock( OSMutexHandle_t MutexHandle, uOSTick_t xTicksToWait);
```

函数OSMutexLock()参数及返回值如下表所示：

参数名	描述
MutexHandle	互斥锁句柄，函数OSMutexCreate()创建互斥锁时的返回值，互斥锁在使用前必须先创建。
xTicksToWait	阻塞超时时间。任务进入阻塞态以等待互斥锁有效的最长时间。 如果xTicksToWait 为0，则OSMutexLock()在互斥锁无效时会立即返回。 阻塞时间是以系统心跳周期为单位的。宏定义OSM2T()可以用来把毫秒为单位的时间转换为系统的Ticks计数值。 如果把xTicksToWait 设置为OSPEND_FOREVER_VALUE，那么阻塞等待将没有超时限制。
返回值	有两个可能的返回值: 1. OS_TRUE，表示成功获得互斥锁。 如果设定了阻塞超时时间(xTicksToWait非0)，在函数返回之前任务将被转移到阻塞态以等待互斥锁有效。如果在超时到来前互斥锁变为有效，亦可被成功获取，返回OS_TRUE。 2. OS_FALSE，表示未能获得互斥锁。 如果设定了阻塞超时时间（xTicksToWait 非0），在函数返回之前任务将被转移到阻塞态以等待互斥锁有效。但直到超时互斥锁也没有变为有效，所以不会获得互斥锁，返回OS_FALSE。

互斥锁解锁的API函数原型定义如下所示：

```
sOSBase_t OSMutexUnlock( OSMutexHandle_t MutexHandle);
```

函数OSMutexUnlock()参数及返回值如下表所示：

参数名	描述
MutexHandle	互斥锁句柄，函数OSMutexCreate()创建互斥锁时的返回值，互斥锁在使用前必须先创建。
返回值	有两个可能的返回值: 1. OS_TRUE，互斥锁“Unlock”成功； 2. OS_FALSE，互斥锁“Unlock”失败。

下面简单介绍一下使用互斥锁时可能带来的问题及解决方案：

优先级反转

优先级反转是采用互斥锁提供互斥功能的潜在缺陷之一。在互斥锁使用时，若优先级低的任务已经锁定了互斥锁，在其未对互斥锁解锁前，高优先级的任务抢断执行，并且高优先级任务也需要锁定该互斥锁，此时，高优先级的任务竟然必须等待低优先级的任务对互斥锁的解锁后才可执行。

高优先级任务被低优先级任务阻塞推迟的行为被称为“优先级反转”。这是一种

不合理的行为方式，如果把这种行为再进一步放大，当高优先级任务正等待互斥锁的时候，一个介于两个任务优先之间的中等优先级任务开始执行——这就会导致一个高优先级任务在等待一个低优先级任务，而低优先级任务却无法执行！

优先级反转可能会产生重大问题。但是在一个小型的嵌入式系统中，通常可以在设计阶段就通过规划好资源的访问方式避免出现这个问题。

优先级继承

TINIUX系统中为互斥锁提供了一种基本的“优先级继承”机制。优先级继承是最小化优先级反转负面影响的一种方案，其并不能修正优先级反转带来的问题，仅仅是减小优先级反转的影响。优先级继承使得系统行为的数学分析更为复杂，所以不建议应用程序设计者对优先级继承功能有所依赖。

优先级继承机制会暂时将互斥锁持有者的优先级提升，达到所有等待该互斥锁任务的最高优先级。这样，持有互斥锁的低优先级任务相当于“继承”了等待互斥锁的任务的优先级。互斥锁持有者在归还互斥锁时，优先级会自动设置为其原来的优先级。

值得注意的是，TINIUX系统只实现了最基本的互斥锁优先级继承机制，而这种优先级继承机制基于一个任务在任意时刻只会持有一个互斥锁模型完成。因此建议应用程序设计者在使用互斥锁时进行仔细的规划。

死锁

死锁是利用互斥锁提供互斥功能的另一个潜在缺陷。当两个任务都在等待被对方持有的资源时，两个任务都无法再继续执行，这种情况就被称为死锁。

考虑如下情形，任务A与任务B都需要获得互斥锁X与互斥锁Y以完成各自的工作：
1. 任务A执行，并成功锁定了互斥锁X。
2. 任务A被任务B抢占。
3. 任务B成功锁定了互斥锁Y，之后又试图锁定互斥锁X，但互斥锁X已经被任务A锁定，所以对任务B无效。任务B进入阻塞态以等待互斥锁X解锁。
4. 任务A得以继续执行。其试图锁定互斥锁Y，但互斥锁Y已经被任务B锁定。任务A进入阻塞态以等待互斥锁Y解锁。

这种情形的最终结局是，任务A在等待一个被任务B锁定的互斥锁，而任务B也在等待一个被任务A锁定的互斥锁。这样两个任务都不可能再执行下去了，导致死锁现象发生。

和优先级反转一样，避免死锁的最好方法就是在设计阶段就考虑到这种潜在的风险，这样设计出来的系统就不应该出现死锁的情况。鉴于实践经验，对于一个微型嵌入式系统，死锁并不是一个大问题，因为系统设计者对整个应用程序都非常清楚，所以能够找出发生死锁的代码区域，并消除死锁问题。

守护任务

守护任务提供了一种干净利落的方法来实现互斥功能，而不用担心会发生优先级反转和死锁。

守护任务是对某个资源具有唯一所有权的任务。只有守护任务才可以直接访问其守护的资源——其它任务要访问该资源只能间接地通过守护任务提供的服务实现。

附：蹄牛操作系统TINIUX常用接口函数介绍

蹄牛操作系统 TINIUX 在设计时，其文件命名、函数名及变量命名由专用的前缀进行区分：前缀为 OS，表示为 TINIUX 的内核，这些是与平台无关的内核部分，在进行跨平台移植时，无需更改；前缀为 Fit，表示为硬件（芯片类型等）相关的部分，在进行移植时，这一部分的文件、函数及变量需要根据硬件平台（芯片类型等）进行适当的调整；

为了突显嵌入式操作系统配置及裁剪的灵活性，在 TINIUX 系统设计之初就进行了全面的考量，系统中相关参数及功能模块采用了宏定义的方式进行配置，常用的配置参数均保存在文件“OSType.h”中，为便于移植，相应配置信息均可以在“OSPreset.h”文件中进行重新定义，不推荐用户直接更改“OSType.h”文件。系统中常用的配置如下所示：

OSTICK_RATE_HZ：该参数配置了系统运行的“速率”，其决定了系统内核调度的最小时间粒度。默认配置为 1000Hz，最小时间粒度为 1 毫秒。在 Demo 示例工程中，采用 SETOS_TICK_RATE_HZ 配置；
OSNAME_MAX_LEN：系统中名称的最大长度，包括任务名称，软件定时器名称等，默认配置为 10 字节。在 Demo 示例工程中，采用 SETOS_MAX_NAME_LEN 配置；
OSLOWEAST_PRIORITY：系统最低优先级数值，配置为 0；
OSHIGHEAST_PRIORITY：系统可使用的优先级数。原则上讲，系统不会限制优先级数量，不过优先级越多，占用的资源越多，推荐优先级数不超过 32，系统默认优先级数为 8。由于任务最低优先级从 0 开始，则用户实际可使用的优先级范围为 0 到 OSHIGHEAST_PRIORITY-1。在附带的 Demo 示例工程中，通过宏定义变量 SETOS_MAX_PRIORITIES 对优先级数进行配置；
OSTOTAL_HEAP_SIZE：为 TINIUX 操作系统分配的栈空间大小。用户采用系统函数创建的任务、信号量、互斥锁、消息队列、软件定时器等均使用该栈空间。在 Demo 示例工程中，采用 SETOS_TOTAL_HEAP_SIZE 配置。
OSMINIMAL_STACK_SIZE：为任务分配的最小栈空间大小，默认配置 32 个字长。任务使用的栈空间量由具体的任务决定，若任务中局部变量较多，使用空间较大，则需配置更大的栈空间。在 Demo 示例工程中，采用 SETOS_MINIMAL_STACK_SIZE 配置；
OSPEND_FOREVER_VALUE：定义永久挂起/等待的数值，用于信号量、互斥锁、消息队列等永久等待的定义数值，在 32 位宽的芯片中，推荐配置为 0xFFFFFFFF。在 Demo 示例工程中，采用 SETOS_PEND_FOREVER_VALUE 配置；
OS_SEMAPHORE_ON：是否启用系统信号量的标识。数值为 1 则启用信号量功能，数值为 0，则不启用信号量。在 Demo 示例工程中，采用 SETOS_USE_SEMAPHORE 配置；
OS_MSGQ_ON：是否启用消息队列的标识。数值为 1 则启用消息队列，数值为 0，则不启用消息队列。在 Demo 示例工程中，采用 SETOS_USE_MSGQ 配置；
OSMSGQ_MAX_MSGNUM：消息队列中保存的消息数量，默认配置为 5。在向该消息队列发送消息时，若消息数达到该数值时，则消息队列已满，需挂起等待，或者把旧数据覆盖掉。在 Demo 示例工程中，采用 SETOS_MSGQ_MAX_MSGNUM 配置；
OS_MUTEX_ON：是否启用互斥锁的标识。数值为 1 则启用互斥锁，数值为 0，则不启用互斥锁。在 Demo 示例工程中，采用 SETOS_USE_MUTEX 配置；
OS_TIMER_ON：是否启用软件定时器的标识。数值为 1 则启用软件定时器，数值为 0，则不启用软件定时器。需要注意一点的是，系统软件定时器功能依赖消息队列，在使用软件定时器前，需要启用系统的消息队列功能。在 Demo 示例工程中，采用 SETOS_USE_TIMER 配置启用软件定时器；
OSCALLBACK_TASK_PRIO：用于配置软件定时器指令（如定时器启动与停止等）响应任务的优先级，推荐采用 OSHIGHEAST_PRIORITY-1。在 Demo 示例工程中，采用 SETOS_CALLBACK_TASK_PRIORITY 配置；

OS_LOWPOWER_ON:是否启用低功耗标识。数值为 0，则不启用低功耗模式；数值为非 0，则启用低功耗模式，系统会在空闲任务中进入低功耗状态，直至有效任务恢复执行或者被外部中断唤醒。需要注意的是，采用低功耗模式，有可能会影响系统的时钟精准状态，产生时钟漂移；部分芯片不支持低功耗模式，因此系统默认不支持低功耗。

OS_TASK_SIGNAL_ON：是否启用轻量级同步信号/消息的标识。数值为 0，则不启动轻量级同步信号/消息，数值非 0，则启用轻量级同步信号/消息。轻量级同步信号与轻量级同步消息可以实现旗语 Semaphore 与消息队列 MsgQ 的部分功能，而占用的空间更小，该功能通常在芯片 Ram 非常小的情况下配置使用。

系统 TINIUX 是多任务抢占式操作系统，高优先级任务可以抢占执行，体现了操作系统的实时性。在 TINIUX 系统中，优先级 0 为系统最低优先级，该优先级为空闲任务 OSIdleTask 使用的优先级。用户可以根据任务的重要程度自行配置。

操作系统运行频率（速率）常用 Ticks 表示，Ticks 亦被称为时钟滴答。操作系统“滴答”由硬件以规律性的定时中断产生。在 TINIUX 系统中，时钟“滴答”决定了系统的最小时间粒度，这个参数可以根据硬件平台进行配置。在 TINIUX 系统中，该参数采用宏定义 SETOS_TICK_RATE_HZ 进行配置。在 TINIUX 提供的大部分示例工程中，均配置的为 1000Hz，即每个时钟滴答间隔是 1 毫秒。

操作系统启动后，会按照任务的优先级选择性的执行，最先执行处于等待状态的最高优先级任务，直至该任务让出执行权或者被其它更高优先级的任务抢断。在系统运行的过程中，如果没有符合条件的任务需要执行，则运行系统中预留的 OSIdleTask（空闲任务）。

下面是蹄牛操作系统 TINIUX 中经常用到的接口函数，供大家使用时参考；

一、系统中任务相关的 API 函数

```
OSTaskHandle_t OSTaskCreate(OSTaskFunction_t  pxTaskFunction,
                            void*             pvParameter,
                            const uOS16_t     usStackDepth,
                            uOSBase_t         uxPriority,
                            sOS8_t*           pcTaskName)
```

OSTaskCreate 为 TINIUX 系统的任务创建函数,其中参数 OSTaskFunction_t pxTaskFunction 为任务函数，该任务函数定义为 void TaskFunction(void *pParameters)；任务函数中的参数 void *pParameters 亦即 OSTaskCreate 的第二个参数；第三个参数为任务的栈空间 usStackDepth，栈空间需要根据任务占用的空间多少进行调整；第四个参数为任务的优先级，用户可根据该任务的重要程度自行配置其优先级。第五个参数为任务名字，任务名字也就是任务的标签，主要方便在调试时区分不同的任务。

函数 OSTaskCreate 的返回值为 OSTaskHandle_t 类型的任务句柄，该句柄可以被其它系统函数调用，以设置或控制任务的状态；

```
void OSTaskSleep( const uOSTick_t uxTicksToDelay )
```

OSTaskSleep 为 TINIUX 系统中任务延迟执行的设置函数，通过此函数，可以把当前任务休眠若干毫秒的时间。参数 uOSTick_t uxTicksToDelay 代表休眠的时间长短，单位为 Ticks，用户可以通过 OSM2T()把毫秒转换为 Ticks 计数；

```
OSSchedule()
```

OSSchedule 函数为 TINIUX 系统中的任务控制类函数，在任务函数中调用，用于让出当前任务的执行权，并切换到下一个处于等待执行状态的任务；函数 OSSchedule 会把当前任务移到等待执行队列的末尾，若只有当前任务处于等待执行状态，则仍然执行该任务；

```
void OSTaskSuspend( OSTaskHandle_t TaskHandle )
```

OSTaskSuspend 函数为系统任务挂起函数，任务挂起后，将不再参与系统的调度，但任务仍然保留在系统中，任务占用的资源不会释放。参数 TaskHandle 用于表示待挂起的任务句柄，该数值若为 OS_NULL，则挂起当前正在运行的任务；

```
void OSTaskResume( OSTaskHandle_t TaskHandle )
```

OSTaskResume 函数用于把挂起的任务恢复（激活），使该任务重新参与系统的调度，参数 TaskHandle 表示待恢复的任务句柄，该参数不允许为 OS_NULL。此函数不是中断安全函数，不允许在中断函数中调用该函数；

```
sOSBase_t OSTaskResumeFromISR( OSTaskHandle_t TaskHandle )
```

OSTaskResumeFromISR 函数用于把挂起的任务恢复（激活），使该任务重新参与系统的调度，参数 TaskHandle 表示待恢复的任务句柄，该参数不允许为 OS_NULL；此函数为中断安全函数，只能在中断服务函数中调用；

```
uOSBase_t OSInit( void )
```

OSInit 函数为 TINIUX 中的系统参数初始化函数；在该函数中，会为系统的堆栈进行初始化，便于为系统的任务、信号量、同步消息与互斥锁等分配空间；同时，会对系统中必要的全局变量进行初始化。尽管大部分变量已经在定义时进行了初始化，但是部分芯片首次上电启动后默认数值仍然不明确，有必要调用此接口函数进行初始化确认；**注：此函数需要在使用 TINIUX 任何接口函数前调用；**

```
uOSBase_t OSStart( void )
```

OSStart 函数是 TINIUX 系统中的任务调度启动函数；在该函数中，系统会设置空闲任务 OSIdleTask 及时钟中断（时钟滴答）；OSIdleTask 任务为系统空闲任务，若系统中无其它需要执行的任务，则会调用该空闲任务，空闲任务可以用于统计当前系统的利用率，及释放处于待删除状态任务的资源；时钟中断则为系统的 ticks 配置，整个系统的运行即依赖此 ticks 驱动运行；调用此函数后，TINIUX 系统就开始启动运行了，因此在正常情况下不会接收到该函数的返回值。一旦接收到该函数的返回值，则表明 TINIUX 系统启动异常，有可能是为系统分配的堆栈空间太小，导致系统无法启动，此时需要进一步排查确认；

二、任务同步信号量相关的 API 函数

```
OSSemHandle_t  OSSemCreate( const uOSBase_t uxInitialCount )
```

函数 OSSemCreate 为信号量（Semaphore）创建函数，用于创建任务间及中断与任务间同步的信号量。信号量创建时，可以通过参数 uxInitialCount 设置该信号量的有效数值，若设置为 0，表示该信号量创建后无有效信号，对应的 OSSemPend（）函数处于阻塞状态，等待 OSSemPost()函数发送有效信号；信号量中的信号容量默认为 0xFF；函数返回值为 OSSemHandle_t 类型的句柄，方便用于对该信号量的操控；

```
sOSBase_t OSSemPend( OSSemHandle_t SemHandle, uOSTick_t xTicksToWait)
```

函数 OSSemPend 为信号量等待函数，在任务执行函数中调用，用于等待相关同步的信号量；参数 OSSemHandle_t SemHandle 为信号量句柄，参数 uOSTick_t xTicksToWait 为任务阻塞时间，单位为 Tick 数，若设置为 OSPEND_FORVER_VALUE，则会永远阻塞，直至指定信号量 SemHandle 获取到有效信号；

```
sOSBase_t OSSemPost( OSSemHandle_t SemHandle)
```

函数 OSSemPost 用于向指定信号量发送有效信号，使处于等待该信号量的任务获取同步信号，以便恢复执行。注：此函数不能在中断服务函数中调用。

```
sOSBase_t OSSemPostFromISR( OSSemHandle_t SemHandle)
```

函数 OSSemPostFromISR 用于向指定信号量发送有效信号，使处于等待该信号量的任务获取同步信号，以便恢复执行。注：此函数只能在中断服务函数中调用。

三、任务同步消息相关的 API 函数

```
OSMsgQHandle_t OSMsgQCreate( const uOSBase_t uxQueueLength, const uOSBase_t uxItemSize)
```

函数 OSMsgQCreate 为消息队列的创建函数，用于创建任务间同步操作的消息队列，参数 uxQueueLength 为消息队列中的消息数的容量（消息数目超过此容量，则发送任务挂起，直到消息队列有空闲位置），第二个参数 uxItemSize 为单个消息的长度。其返回值为 OSMsgQHandle_t 类型的消息队列句柄，方便对消息队列的操控；

```
sOSBase_t OSMsgQReceive( OSMsgQHandle_t MsgQHandle, void * const pvBuffer, uOSTick_t xTicksToWait)
```

函数 OSMsgQReceive 用于在任务中接收指定消息队列的消息，在任务函数中调用。该函数为任务阻塞函数。参数 MsgQHandle 为消息队列句柄，参数 pvBuffer 表示消息的指针，参数 xTicksToWait 为消息队列等待接收时间，单位为 Tick，若设置为 OSPEND_FORVER_VALUE，则会永远等待，直至指定消息队列 MsgQHandle 获取到有效消息为止。函数返回值代表消息接收状态，若为 OS_FALSE 则未接收到有效消息，若为 OS_TRUE 则接收到有效消息；

```
sOSBase_t OSMsgQReceiveFromISR( OSMsgQHandle_t MsgQHandle, void * const pvBuffer)
```

函数 OSMsgQReceiveFromISR 用于在中断函数中接收指定消息队列的消息；参数 MsgQHandle 为消息队列句柄，参数 pvBuffer 表示消息的指针，该函数不会阻塞。函数返回值代表消息接收状态，若为 OS_FALSE 则未接收到有效消息，若为 OS_TRUE 则接收到有效消息；注：此函数只能在中断服务函数中调用。

```
sOSBase_t OSMsgQSend( OSMsgQHandle_t MsgQHandle, const void * const pvItemToQueue, uOSTick_t xTicksToWait)
```

函数 OSMsgQSend 用于向指定的消息队列发送消息，使处于等待该消息的任务获取同步消息，并恢复执行，其中参数 MsgQHandle 表示消息队列，参数 pvItemToQueue 表示消息地址（指针），参数 xTicksToWait 为消息发送等待接收时间，单位为 Tick，若设置为 OSPEND_FORVER_VALUE，则会永远等待，直至指定消息队列 MsgQHandle 有空闲位置。函数返回值代表消息发送状态，若为 OS_FALSE 则消息发送失败，若为 OS_TRUE 则消息发送成功；注：此函数不能在中断服务函数中调用。

```
sOSBase_t OSMsgQSendFromISR( OSMsgQHandle_t MsgQHandle, const void * const pvItemToQueue)
```

函数 OSMsgQSendFromISR 用于向指定的消息队列发送消息，使处于等待该消息的任务获取同步消息，并恢

复执行，其中参数 MsgQHandle 表示消息队列，参数 pvItemToQueue 表示消息地址（指针），如果消息队列已满，则函数不会阻塞，直接返回发送失败信息。注：此函数只能在中断服务函数中调用。

四、定时器相关的 API 函数

```
OSTimerHandle_t OSTimerCreate(const uOSBase_t uxTimerTicks, const uOS16_t uiIsPeriod, const OSTimerFunction_t Function, void* pvParameter, sOS8_t* pcName)
```

接口函数 OSTimerCreate 用于创建定时器。其中参数 uxTimerTicks 为定时器周期值，单位为系统滴答 Ticks，可以通过宏定义 OSM2T()把毫秒数值转化为系统 Ticks 数；参数 uiIsPeriod 表示定时器是否为周期性定时器，取值为 OS_OS_TRUE 时为周期性定时器，取值为 OS_OS_FALSE 时为单次定时器；参数 Function 为定时器的服务函数，用于响应定时器，函数 OSTimerFunction_t 的定义类型为 void TimerFunction(void *pParameters)。pvParameter 为定时器服务函数的参数，不用时可以设置为 OS_NULL；参数 pcName 为定时器的名称，方便区分不同的定时器；注意：定时器服务函数中禁止添加信号量等待、消息队列等待等阻塞函数，为不影响系统的性能，定时器服务函数耗时越少越好。

```
uOSBase_t OSTimerStart(OSTimerHandle_t const TimerHandle)
```

定时器创建完毕后并不会自动启动，需要用户调用启动函数 OSTimerStart()，之后定时器才会生效。参数 TimerHandle 为定时器句柄，为定时器创建函数 OSTimerCreate()的返回值；

```
uOSBase_t OSTimerStop(OSTimerHandle_t const TimerHandle)
```

定时启动后，用户可以通过接口函数 OSTimerStop()停止定时。参数 TimerHandle 为定时器句柄，为定时器创建函数 OSTimerCreate()的返回值；

五、轻量级同步信号的 API 函数

```
uOSBool_t OSTaskSignalWait( uOSTick_t const uxTicksToWait)
```

函数 OSTaskSignalWait()为轻量级信号量（Signal）等待函数，用于等待系统中别的任务函数或者中断响应函数发出的同步信号。任务函数在调用该接口函数后，即转入挂起等待状态。参数 uxTicksToWait 为等待时间，单位为 Tick，若设置为 OSPEND_FORVER_VALUE，则会永远等待，直至接收到同步信号。函数返回值代表信号接收状态，若为 OS_FALSE 则未接收到有效信号，若为 OS_TRUE 则接收到有效信号；注意，该函数不允许在中断响应函数中使用。

```
uOSBool_t OSTaskSignalEmit( OSTaskHandle_t const TaskHandle )
```

函数 OSTaskSignalEmit()为轻量级信号量（Signal）发射函数，向目标任务函数发送同步信号。参数 TaskHandle 为等待接收信号的目标任务句柄，禁止传入空指针；函数返回值代表信号发送状态，若为 OS_FALSE 则未发送出有效信号，若为 OS_TRUE 则成功发送出有效信号；注意，该函数不允许在中断响应函数中使用。

```
uOSBool_t OSTaskSignalEmitFromISR( OSTaskHandle_t const TaskHandle )
```

函数 OSTaskSignalEmitFromISR()为轻量级信号量（Signal）发射函数，向目标任务函数发送同步信号。参数 TaskHandle 为等待接收信号的目标任务句柄，禁止传入空指针；函数返回值代表信号发送状态，若为 OS_FALSE 则未发送出有效信号，若为 OS_TRUE 则成功发送出有效信号；注意，该函数只允许在中断响应函数中使用。

```
uOSBool_t OSTaskSignalClear( OSTaskHandle_t const TaskHandle )
```

函数 OSTaskSignalClear()用于对指定任务的信号进行清除，参数 TaskHandle 为等待接收信号的任务句柄，禁止传入空指针；函数返回值代表信号清除状态，若为 OS_FALSE 则未成功清除信号，若为 OS_TRUE 则成功清除对应任务中的信号数值，信号数值进行复位。

六、轻量级同步消息的 API 函数

```
uOSBool_t OSTaskSignalWaitMsg( sOSBase_t xSigValue, uOSTick_t const uxTicksToWait)
```

函数 OSTaskSignalWaitMsg()为轻量级同步消息等待函数，用于等待系统中别的任务函数或者中断响应函数发出的同步消息。任务函数在调用该接口函数后，即转入挂起等待状态。参数 xSigValue 为输出参数，在成功接收到消息后，该参数会获取具体的消息数值信息。参数 uxTicksToWait 为等待时间，单位为 Tick，若设置为 OSPEND_FORVER_VALUE，则会永远等待，直至接收到同步消息。函数返回值代表消息接收状态，若为 OS_FALSE 则未接收到有效消息，若为 OS_TRUE 则接收到有效消息；注意，该函数不允许在中断响应函数中使用。

```
uOSBool_t OSTaskSignalEmitMsg( OSTaskHandle_t const TaskHandle, sOSBase_t const xSigValue, uOSBool_t bOverWrite )
```

函数 OSTaskSignalEmitMsg()为轻量级消息发射函数，用于向指定的目标任务发送同步消息。参数 TaskHandle 为等待接收消息的任务句柄，即目标任务句柄，禁止向该参数传入空指针；参数 xSigValue 为待发送出的消息数值；参数 bOverWrite 表示既有的消息数值是否在目标任务未响应的情况下可以覆盖重写，若为 OS_TRUE，则代表可以覆盖重新赋值，若为 OS_FALSE，则该信号数值不允许覆盖，该函数直接返回 OS_FALSE，代表向目标任务发送消息失败。函数返回值代表消息发送状态，若为 OS_FALSE 则未发送出有效消息，若为 OS_TRUE 则发送出有效消息；注意，该函数不允许在中断响应函数中使用。

```
uOSBool_t OSTaskSignalEmitMsgFromISR( OSTaskHandle_t const TaskHandle, sOSBase_t const xSigValue, uOSBool_t bOverWrite )
```

函数 OSTaskSignalEmitMsgFromISR()为轻量级消息发射函数，用于向指定的目标任务发送同步消息。参数 TaskHandle 为等待接收消息的任务句柄，即目标任务句柄，禁止向该参数传入空指针；参数 xSigValue 为待发送出的消息数值；参数 bOverWrite 表示既有的消息数值是否在目标任务未响应的情况下可以覆盖重写，若为 OS_TRUE，则代表可以覆盖重新赋值，若为 OS_FALSE，则该信号数值不允许覆盖，该函数直接返回 OS_FALSE，代表向目标任务发送消息失败。函数返回值代表消息发送状态，若为 OS_FALSE 则未发送出有效消息，若为 OS_TRUE 则发送出有效消息；注意，该函数只允许在中断响应函数中使用。

```
uOSBool_t OSTaskSignalClear( OSTaskHandle_t const TaskHandle )
```

函数 OSTaskSignalClear()用于对指定任务的消息进行清除，参数 TaskHandle 为等待接收消息的任务句柄，禁止传入空指针；函数返回值代表消息清除状态，若为 OS_FALSE 则未成功清除消息，若为 OS_TRUE 则成功清除目标任务中的消息数值，消息数值进行复位。

简单示例程序

下面是采用蹄牛操作系统 TINIUX 实现的多任务处理演示示例，主要包括任务创建、信号量创建、消息队列创建、任务延时等功能。代码如下：

```
//任务句柄
OSTaskHandle_t      TaskCtrlHandle = OS_NULL;
OSTaskHandle_t      TaskTest1Handle = OS_NULL;
OSTaskHandle_t      TaskTest2Handle = OS_NULL;
//信号量句柄
OSSemHandle_t       SemHandle = OS_NULL;
//消息队列句柄
OSMsgQHandle_t      MsgQHandle = OS_NULL;
//任务函数
static void TaskCtrl( void *pvParameters );
static void TaskTest1( void *pvParameters );
static void TaskTest2( void *pvParameters );

int main( void )
{
    //Initialize the parameter in TINIUX
    OSInit();

    // 创建消息队列，消息容量5个，消息长度为int类长度.
    MsgQHandle = OSMsgQCreate( 5, sizeof( uint32_t ) );
    // 创建信号量
    SemHandle = OSSemCreate(0);
    // 创建任务
    TaskCtrlHandle = OSTaskCreate(TaskCtrl, OS_NULL, OSMINIMAL_STACK_SIZE,
OSLOWEAST_PRIORITY+1, "Ctrl");
    TaskTest1Handle = OSTaskCreate(TaskTest1, OS_NULL, OSMINIMAL_STACK_SIZE,
OSLOWEAST_PRIORITY+1, "Test1");
    TaskTest2Handle = OSTaskCreate(TaskTest2, OS_NULL, OSMINIMAL_STACK_SIZE,
OSLOWEAST_PRIORITY+1, "Test2");

    // 启动系统
    OSStart();
    for( ;; );
}

static void TaskCtrl( void *pvParameters )
{
    unsigned int uiValueToSend = 0;
    for( ;; )
    {
        //发送信号量
        OSSemPost(SemHandle);
        //发送消息队列
        OSMsgQSend( MsgQHandle, &uiValueToSend, OSPEND_FOREVER_VALUE );
```

```
        uiValueToSend += 1;
        //延时等待
        OSTaskSleep(200*OSTICKS_PER_MS);
    }
}

static void TaskTest1( void *pvParameters )
{
    unsigned int uiReceivedValue;
    for( ;; )
    {
        //等待接收消息
        OSMsgQReceive( MsgQHandle, &uiReceivedValue, OSPEND_FOREVER_VALUE );
        //do something here
    }
}

static void TaskTest2( void *pvParameters )
{
    for( ;; )
    {
        //等待信号量
        OSSemPend(SemHandle, OSPEND_FOREVER_VALUE);
        //do something here
    }
}
```