北京北阳电子技术有限公司
Beijing Sunnorth Electronic Technology Co.,Ltd

MiniOS 用户操作手册

发布日：2005 年 10 月 12 日

北京北阳电子技术有限公司.
北京市海淀区上地信息产业基地中黎科技园 1 号楼 6 层
电话: 86-10-62981668　　传真 ：86-10-62985972　　邮编：100085

版权声明

北阳电子技术有限公司保留对此文件修改之权利且不另行通知。北阳电子技术有限公司所提供之资讯相信为正确且可靠的，但并不保证本文件中绝无错误。请于向北阳电子技术有限公司提出订单前，自行确定所使用之相关技术文件及规格为最新之版本。若因贵公司使用本公司之文件或产品，而涉及第三人之专利或著作权等智慧财产权之应用及配合时，则应由贵公司负责取得同意及授权，本公司仅单纯贩售产品，上述关于同意及授权，非属本公司应为保证之责任。又未经北阳电子技术有限公司之正式书面许可，本公司之所有产品不得用于医疗器材，維持生命系統及飞航等相关设备。

版本说明

2005-10-12	V2.0	杨敏彦	针对重写后的 MiniOS 进行文档修订
2004-9-2	V1.4	史小平	MiniOS API 改名.
2004-7-7	V1.3.1	史小平	添加一些 API 函数
2004,4,12	V1.3	马涛	针对 Platform 增加 API,修改 API 的名字
2003,7,16	V1.2	王文艳	针对 API 部分补充说明及示例
2003.6.18	V1.1	王泰运	针对 MiniOS V1.5.X, 性能参数为 V1.5.1 的
2003.5.22	V1.0	王泰运	为 MiniOS V1.5.0 而写

目录

用户操作手册

1 引言

本书为 MiniOS 2.X.X 发行版本的用户手册，适用于 2.X.X 版本。

为了配合 MiniOS 的发行，使用户能对 OS 有比较深入的了解，MiniOS 的开发组成员编写了本书，以求为用户提供一点参考，使用户能在较短的时间内使用 MiniOS 开发自己的应用程序，步入嵌入式实时系统的先进行列。

当前嵌入式系统非常的流行，也成为控制、自动化等行业的热门话题，而实时操作系统又是嵌入式的核心问题，本书力求准确，以通俗的语言为用户理解 MiniOS、应用 MiniOS 做一个指导性作用。

由于凌阳公司推出了新一代的 CPU--μ'nSP，它大幅度的提高了 CPU 执行速度，这为 MiniOS 的诞生提供了必要的硬件条件，另一方面高速 CPU 及外围硬件的丰富又使得程序员编程难度加大，想要完成更多的功能，则必须完成很多硬件操作。凌阳公司为最大减少程序员的工作量，决定开发配套的 OS。伴随着 OS 在嵌入式系统方面的应用，MiniOS 汲取了现在流行的 OS 的许多特点。

MiniOS 的特点如下：

高效的任务管理

MiniOS为基于优先级的抢占式任务调度，总是运行就绪条件下优先级最高的任务。

MiniOS 最多支持 16 个任务（有一个任务保留给系统当作空闲任务，所以用户最多可以有 15 个任务），每个任务的优先级必须是不相同的。每个任务拥有自己的上下文和堆栈。

MiniOS 任务调度快速，执行原始上下文切换只用 4.62us @ 49.152MHz。

任务可以延时等待。

有效的中断管理

MiniOS 使用了一系列的特殊方式，使多个中断服务函数可以共享一个中断源，这样就使得用户在原有中断源入口函数（无源代码）中可以加入自己的中断服务函数而不影响原来的中断服务函数。

MiniOS 的中断处理过程使用系统提供的栈空间（系统栈），而非用户的栈空间，所以每个任务不必为中断多留空间来保证用户任务正确执行，减少空间的浪费。

MiniOS 的调度及关中断时间是可知的，不随任务的多少而改变。

MiniOS 内核最长关中断时间为 6.18us @ 49.152MHz。

多种任务间通信机制

MiniOS提供了三种任务间的通信机制：信号量、消息邮件、消息队列。

核心可配置

MiniOS 提供了一种简单的方法完成核心配置，只通过修改一个配置文件就可以完成配置，配置可以使核心程序占用更少的 RAM。

核心可裁剪

除系统必需的几个函数（如调度程序等），只有用户应用的API函数会被链接到用户程序中，这样可使MiniOS核心程序所占用的空间最小。

提供 C 语言级调试方法

调试程序时，MiniOS可以在调试窗口报告所有任务的状态、事件状态等信息，甚至包括每个任务的寄存器现场信息。

可固化：

MiniOS 提供 API 让用户程序调用，最后编译出的程序直接下载和固化，成为用户产品的一部分。

稳定可靠

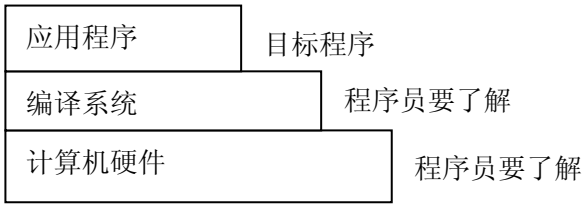
MiniOS 是凌阳公司的开发人员经过长时间的准备，经过广泛的讨论及测试开发完成的。愈来愈多的应用将使它更成熟稳定。使用 MiniOS 开发产品，用户只需将开发重心置于高阶流程控制，而无须关心低层的实现机制。

2 OS 的一般概念及嵌入式 OS

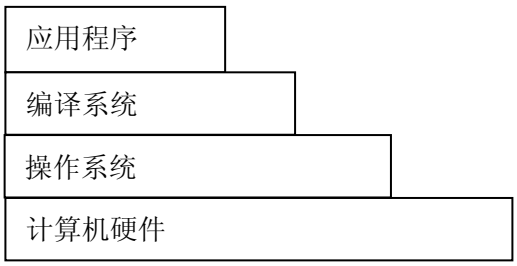
2.1 操作系统概述

操作系统是控制应用程序执行的程序，并充当应用程序和 CPU 硬件之间的接口，使用户使用计算机更容易，操作系统以更有效的方式管理计算机的系统资源。

无操作系统的用户：



从中可以看到程序员要了解计算机的硬件结构和编译系统。要求程序员自己管理硬件资源。而用操作系统



从中可以看到，程序员要了解的只是操作系统提供的功能和编译系统就可以了，无须太多的了解计算机的硬件系统，操作系统会管理计算机的硬件资源。

原来的嵌入式系统开发流程：

举例说明，假设程序由 8 个子程序组合完成全部的功能，其 KeyPress()和 Uart()要等待外部事件发生，如 KeyPress()等待按键，Uart()等待 UART。程序组织方式如下，

注：每个函数还可能调用其它函数，形成嵌套，我们现在并不关心它是否调用其它函数。

```
while(1) {
    function1();          //功能函数 1
    function2();          //功能函数 2
    function3();          //功能函数 3
    KeyPress();           //功能函数 4，本函数判断是否有键按下，如没有则返回主循环
    function5();          //功能函数 5，
    function6();          //功能函数 6
    function7();          //功能函数 7
    Uart();               //功能函数 8，判断是否有数据发过来，如没有则返回。
}
```

KeyPress()和 Uart()这两个函数会检测有否按键以及 UART 的数据是否到达，这部分一般是由中断服务程序来配合完成，中断服务程序的主要工作是将接到的数据放入缓冲区中，并把相应的标志置位，用来通知对应的函数处理。这样就有一个问题产生，如果 UART 接收到数据，当时的程序刚刚运行完 Uart()，这样就要等到把 function1—function7 全部运行一遍才会再进入 Uart()处理接收到的数据，而如果当前 function7 刚好运行完，则马上就会处理接收到的数据。这样情况还算好，如果有按键发生，则会先处理按键。这样设计其实很难知道接收到的数据何时会被处理。当然系统总会处理的，但对一些实时性要求比较高的应用很难满足要求。

多任务的提出：

有好多子程序是可以并行执行的（有时间顺序性的问题下一章会讨论），这些程序可以并发的执行而不会影响程序的正确性。可以把它们拆分成单个的任务，允许 OS 调度运行。从宏观上看，这些任务是并发（同时）执行的，但从微观上看，程序依然是顺序的，因为 CPU 只有一个，不可能真正的并行运行程序，只是调度程序会在两个任务之间根据一些算法来切换任务占有 CPU 的时间。

共享资源

由于多任务的提出，其实也不光是多任务，不用 OS 的系统也存着这样的问题。先从不用 OS 的系统说起。程序如下：

```
int a=0;
function()
{
    …
     a =  0 ;     //后面还要用到 a，这时中断发生了，调用了下面的函数，这样使 a 的值变为 1,
                  //发生了错误。
    …
}
```

中断服务程序如下：

```
ISR（）
{
     a＋＋;
}
```

可以看到，就是不用ＯＳ的系统，也可能会在操作公用变量（共享资源的一种）时产生错误。多任务的情况更为复杂，主要是由于多任务并发执行，可能有很多任务在操作共享资源。互斥是指每个任务使用共享资源时必须独占该资源。比如上例中，可以用关中断的方式保证互斥，在多任务中还有其它方法来保证互斥，这些将在下章介绍。

任务的状态

任务的概念 任务也叫线程，近代操作系统中有进程和线程之分，所谓进程是这样一种概念，进程是资源分配的单位，在进程中可以有多个线程共享分配的资源。进程一般情况下用于多用户的系统中，如 Linux，多进程的系统支持存储保护等机制来保证多用户之间互不影响，而且要有硬件支持，如特权指令及保护机制等来实现。一般的嵌入式系统没有那么多的硬件支持，也不提供保护，主要由程序员自己来保证多个任务之间的资源共享不发生冲突。MiniOS 不支持多进程，本质上为一

个多线程的操作系统。

在微观上，单个 CPU 一次只能运行一条指令，对于任务也一样，一次只能运行一个任务。所以其它的任务是不可能运行的，这样任务就有很多种状态。下面就 MiniOS 的状态分别给出初步的描述：

1、 就绪态：处于这种状态的任务可以被调度成为运行态。在内核中标识为 Ready.

2、 运行态：当前被运行的任务状态就是运行态。在某一瞬间只能有一个任务处于运行态。由于只有就绪的任务才可能成为运行态，所以内核中无运行态，只标识为就绪 Ready.

3、 被中断态，这是临时的状态，运行中的任务被中断时就为此状态，中断后可能恢复被中断的任务，也可能将其转为就绪状态，这是因为可能有更高的优先级的任务要运行。

4、 挂起态：由于某种原因用户希望某个任务不再被调度，用户可以将它挂起，直到用户主动去恢复它为就绪。注意处于这种状态的任务只能由用户通过调用 OS API 来恢复，否则将永远不会被调度。

5、 等待态：任务由于某种原因等待一个事件的发生，如等待其它任务放弃共享资源，自身延时一段时间等等都会将任务变为此种状态，直到事件发生，或是等待的时间超过用户指定的时间为止。

6、 死亡态：任务的功能已经完成，不需要它再运行，而删除这个任务，这时任务就处于死亡态，如果再次运行，则必须重新创建。这个状态和挂起态有点象但还有本质上的差别，挂起态的任务还没有完成它的功能，还要在恢复运行以完成指定功能，所以不会放弃任务控制块，而死亡态是已经完成了指定的任务而放弃了任务控制块，其它任务可以占用这个任务的控制块。

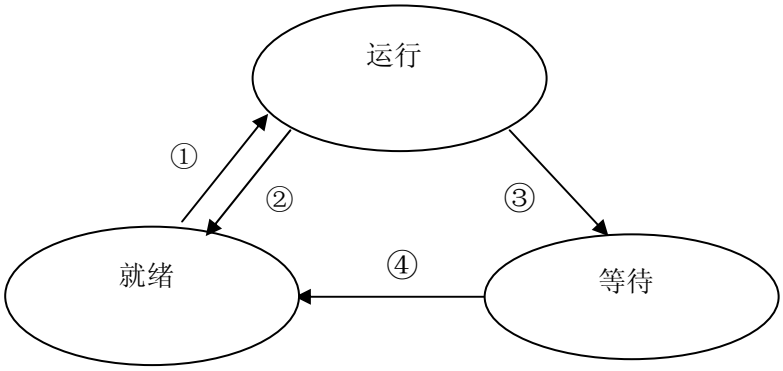
以上是任务的 6 种状态，每个任务必然处于这 6 种状态之一。状态的转换是由于用户调用 OS API 或是由于中断的发生而改变的。

任务有三种基本状态：就绪、运行、等待(这其中的等待包括挂起这种状态)。

任务运行过程中，由于任务自身进展情况及外界环境的变化，这三种基本状态可以依据一定的条件相互转换

1. 就绪—运行
2. 运行—就绪
3. 运行—等待
4. 等待—就绪

其中的 1、2、3、4 的转换方式已在下图中标出，要注意的是有些看似有但却没有的转换方式，比如等待--运行这种转换就没有，等待事件的任务必须先转换成就绪态，才能由就绪态转换成运行态，不能直接转换成运行态；另外就绪态也不能直接转为等待态，原因请读者自己思考。



2.2 内核的启动

由于任何的 OS 都要用一些必要的数据结构来保存一些任务的现场信息，比如当前的任务运行到哪里了，等等。核心没有启动前就要初始化这些数据结构，这个功能在 MiniOS 中是由系统的 API 函数提供的，函数名为 unOSInit(); 这个函数没有入口参数，只是完成一些必要的系统用数据结构和一些硬件的初始化工作。

系统一旦启动就永远不会回到主程序中了，而是在多个任务之间来回切换，所以在启动多任务之前至少要建立一个任务，其它任务可以在启动之前建立，也可在启动后由任务来建立。建立的函数为 unOSTaskCreate(任务名，入口参数，栈顶指针，优先级)，这样一个任务就会建立了，要说明的是任务名对应一个函数名，每个任务要有独立的堆栈，栈顶指针指向用户预先开辟的 RAM 区（数组申请或 Buddy unOStem 申请得到）的尾部（即栈顶），还要指定优先级，由于在实时系统中任务是不平等的，也就是说有一些任务要更高实时性，这可以由优先级来确定。在 MiniOS 中任务为 0-15，号越小优先级越高，0 号任务优先级最高，15 号任务优先级最小，所以只要高优先级的任务不放弃 CPU 的使用权，CPU 会永远被它占用！15 号优先级任务保留给系统的空闲任务了。上面提到过多任务一旦启动，就永远不会回到主程序中了，而在任务之间来回切换，如果在系统中的所有任务都放弃了 CPU 的使用，那 CPU 在做什么呢？去空闲任务运行，空闲任务什么也不做，它永远都是就绪的，也就是说永远会被调度。如果有比 15 号任务更高优先级的就绪，那么空闲任务就不再运行，转而运行高优先级的任务。

要说明一点，任务是不可以返回的，只能是删除，典型的一个任务是死循环。任务可以有入口参数，但不能返回。任务建立后就和其它的任务是并发执行的，而不是调用的关系，所以不能返回。任务和任务之间是协同关系，任务和任务相互通信共同完成一个应用的全过程。

注意：如果用户的程序不是死循环，MiniOS 在它退出时返回到任务的删除程序中，把该任务删除。

例 1、循环亮灯程序，从 B 口输出。只要软件仿真即可。

```
//
// example 1: single task
// descriptions: port B low byte: connect to 8 LEDs
//                    Program run to turn on LEDs one by one,control signal output from port B.
// author: Taiyun Wang
// date: 2003/2/22
//////////////////////////////////////////////////////////////////////////
#define CREATEVAR
#include "spos.h"

int err;                                    //err No
int t1stack[20];                        //Task stack

volatile unsigned int *P_IOB_DATA =(unsigned int*)(0x7005);  //Port B data register
volatile unsigned int *P_IOB_DIR =(unsigned int*)(0x7007);          //Port B direction register
volatile unsigned int *P_IOB_ATTRIB = (unsigned int*)(0x7008);     //Port B attribute register
```

```
main()
 {
 void Task1();
 unOSInit();                                          //initialize OS kernel
 *P_IOB_DIR = 0XFFFF;                                 //Set Prot B is output
 *P_IOB_ATTRIB = 0XFFFF;                              //Set prot B attribute
 err = unOSTaskCreate(Task1,0,t1stack+19,1); //Create a task
 unOSStart();                                         //Start kernel
}

void Task1()                                          //task one
{
 unsigned int i = 1;
      while(1) {
        *P_IOB_DATA = i;
        i<<=1;
        if(i == 0x0100)
              i = 1;
        unOSTimeDly(20);                              //Delay 20 tick
 }
}
```

这个例子没有太多现实意义，但可以从简单的入手来看看系统是如何建立任务的，是不是很简单呢？在任务中调用了延时 API，这将会使任务放弃 CPU 的使用权（空闲任务将会运行），等待时间到，这个任务将会再次运行。这里有一个 tick 的概念，在 MiniOS 中是用系统的 128Hz 来计数的，所以一个 tick 是 1/128 秒。

例 2，在例 1 基础上给出两个任务的情况，这两个任务之间没有通信，是各自独立的任务。这个程序一样不需要 ICE，软件仿真即可。

```
//
// example 2: two tasks
// descriptions: This example based on example 1,
//                    two tasks do not communicate with each other and run independently.
// author: Taiyun Wang
// date: 2003/2/22
////////////////////////////////////////////////////////////////////////////
#define CREATEVAR
#include "spos.h"

int err;                              //err No
int t1stack[20];                  //Task 1 stack
```

```
int t2stack[20];                    //Task 2 stack

volatile unsigned int *P_IOA_DATA =(unsigned int*)(0x7000);         //Port A data register
volatile unsigned int *P_IOA_DIR =(unsigned int*)(0x7002);          //Port A direction register
volatile unsigned int *P_IOA_ATTRIB = (unsigned int*)(0x7003);    //Port A attribute register

volatile unsigned int *P_IOB_DATA =(unsigned int*)(0x7005);         //Port B data register
volatile unsigned int *P_IOB_DIR =(unsigned int*)(0x7007);          //Port B direction register
volatile unsigned int *P_IOB_ATTRIB = (unsigned int*)(0x7008);    //Port B attribute register

main()
{
 void Task1();
 void Task2();
 unOSInit();                                        //Initialize OS kernel

 *P_IOA_DIR = 0XFFFF;                                 //Set Port A output
 *P_IOA_ATTRIB = 0XFFFF;                              //Set Port A attribute

 *P_IOB_DIR = 0XFFFF;                                  //Set port B output
 *P_IOB_ATTRIB = 0XFFFF;                               //Set port b attribute

 err = unOSTaskCreate(Task1,0,t1stack+19,1); //Create first task
 err = unOSTaskCreate(Task2,0,t2stack+19,2); //Create second task
 unOSStart();                                         //Start kenel
}

void Task1()                                          //task 1
{
 unsigned int i = 1;
 while(1) {
        *P_IOB_DATA = i;
        i<<=1;
        if(i == 0x0100)
              i = 1;
        unOSTimeDly(30);                              //Delay 30 tick
 }
}
void Task2()                                          //task 2
{
```

```
    unsigned int i=0;
    while(1) {
        *P_IOA_DATA = i;
        i<<=1;
        if(i == 0)
            i = 1;
        unOSTimeDly(5);                          //Delay 5 tick
    }
}
```

这个例子有一些不同了，两个任务分别控制两个端口的灯，速度上有比较大的差别，读者可以看到实现这个功能的两个任务很简单，而如果用传统的方法，则要用定时器来实现，而且比用 OS 的可读性以及程序的复杂性都要大一些。

以上两个程序按如下步骤运行：

由于怎样建立及编译不是本书的介绍内容，如果不明白可以参看凌阳的使用说明书。

1、 建立工程文件

2、 建立 C 程序文件，将程序写入

3、 设置 MiniOS 的头文件及库文件目录

4、 编译，运行。

注：以上两个程序可以不使用仿真板，而在 IDE 中用软件仿真来观察结果。

这两个程序很简单，实际用户程序会很复杂，使用操作系统将会使程序的可读性、可维护性、开发进度及可靠性等方面都会有很大的提高，如果您用习惯了 OS，那么就再也不想回到老的编程思路上去了。笔者将在后续章节介绍任务间的协作及通信，以建立更加复杂的系统。

3 头文件及内核配置

MiniOS 1.5.X 核心为可配置核心，配置功能通过配置文件实现，本章主要介绍如何配置核心。

用户在使用 MiniOS 1.5.X 时，只须包含一个头文件 spos.h，其它的头文件都由 OS 在此文件中做了相应的处理。另外要注意的是，一般用户在写应用程序时，还须在包含 spos.h 之前，定义 CREATEVAR（需要注意的是，在整个工程中，只可以定义一次 CREATEVAR,否则会有 linker 的错误），OS 会根据此定义将用户的配置信息传递给 OS Kernel。请参考第一章的两个例子。

操作系统运行时除了用户使用的变量外，系统本身也要占用一部分 RAM 及 ROM 空间，用来保存系统运行期间的必要数据。现在让我们来看看 sposconfig.h 头文件的主要内容。以 SPCE061A 为例：

```
#define __BODY_SPCE061A      //注 1
#define MAXTASK          5      //注 2
#define MAXEVENT         3      //注 3
#define MAXQUEUE         3      //注 4
#define MAXMCB       3      //注 5
#define MAXIFT       3      //注 6
#define STACKSIZE        128    //注 7
```

第一行，#define __BODY_SPCE061A, 主要指明用户使用的 body；

第二行，#define MAXTASK 5，定义了系统最大可用的任务数，注意这个数不能小于 1，因为小于 1 就是系统中无任务运行，对系统来说无意义，系统在初始化时会返回错误值，等于 1 也无太大的意义。任务数也不能大于 15，这是 OS 设计时就决定了的。如果大于 15 启动核心时也会返回错误值。 MAXTASK 的大小决定了系统使用多少空间来保存任务的状态等有关信息，数值越大，占用空间越多，它的值应该等于用户同时运行的任务数，注意是同时运行的任务数而不是使用的任务数。

第三行，#define MAXEVENT 3，这个数值是用户同时使用的事件个数，同样它的增大将会使系统所占用的空间增大，但它和 MAXTASK 有一点不同就是它的数值没有其它的限制，0 也可以，表示系统中不使用事件，上限值只受 RAM 空间大小的限制。

第四行 #define MAXQUEUE 3，这个数值是系统所能建立的最多的消息队列的大小，可以为 0 值，但它的大小不可以超过 MAXEVENT 的大小，因为消息队列也是事件的一种，它附加于事件之上。

第五行 #define MAXMCB 3，这个数值表示可用内存控制块的多少。

第六行 #define MAXIFT 3，这个数值表示系统中可用中断服务函数的多少。

第七行 #define STACKSIZE 128，这个数值表示系统栈的大小，MiniOS 中断时所用栈空间是独立于任务的，这个数值指明系统所用系统栈的大小。

sysstruct.h 是系统所用的数据结构的定义文件，包括 TCB、ECB 等等的结构都是定义在这个头文件之中的，系统用它来声明变量，例如在 spos.h 中用如下来声明 TCB：

```
TCB_struct   OS_TCB[MAXTASK];
```

TCB_struct 是在 sysstruct.h 中定义的，它声明了系统用的 TCB 即 OS_TCB, 它的大小为 MAXTASK, 而大小定义在 sposconfig.h 文件中。通过修改 sposconfig.h 中 MAXTASK 的大小就可以

改变系统 TCB 的个数，其它的类似。

这样做可以使核心完全由用户配置，另一方面核心用的数据结构也公开了，给用户调试带来方便。但这个头文件是产生系统用的变量的，所以一个项目中只能有一个程序包含它，不可以两个或是两个以上的程序包含它，否则会有重复定义的错误！还要注意一点用户的程序如果全部用汇编写成，也要生成一个 C 程序包含这个文件来产生核心所需要的数据结构，由 C 来生成系统用变量是为了让调试程序能读出一些调试信息，特别是一些比较复杂的数据结构，这样用户调试比较方便。

另一个主要的头文件是 spos.h，这个文件之中包含了所有操作系统的 API 函数的声明、错误代码的定义以及一些和 body 相关的函数声明。这个文件除了包含了 sposconfig.h 及 sysstruct.h 文件之外，还包含了 sposbody.h。sposbody.h 这个文件利用在 sposconfig.h 文件中定义的 body（如__BODY_SPCE061A），包含与硬体平台相关的另一个头文件（如 sposCE061A.h），其中是和 body 相关的一些定义，比如如何处理中断，中断入口的定义还有一些和 body 相关的函数的定义。用如下的方式实现不同 body 的包含，这里只定义了 spce061A。

```
#ifdef __BODY_SPCE061A        //如果定义了它则包含下面指定的头文件
#include "sposCE061A.h"              //spce061 body 相关文件
#endif
```

再看一下 sposCE061A.h 这个头文件的一部分：

```
extern void unOSEnINTSrc(unsigned int ulVector);
extern void unOSDisINTSrc(unsigned int ulVector);
extern unsigned int SpFGetINTVec(void);
```

为什么会将挂中断的函数放在这里呢？这是因为各种 body 对中断的处理是不一样的，所以中断是与 body 相关的部分，如每个 body 的中断个数，链入的中断入口都不太一样。这样做就可以做到 os 内核与 body 无关。当选择好 body 后，包含与之对应的头文件就可以了。

当然用户不用管这些文件如何包含，只要把 sposconfig.h 这个文件修改到满足需要就可以了。还记得第一行的#define __BODY_SPCE061A 吗？就是配置 body 用的，用户只要按说明书把 define 后面的定义改成指定的就可以了。

所以，所谓的核心配置就是把 sposconfig.h 这个文件修改一下就可以了。

4 通信与同步

第一章的多任务在任务之间并没有通信，在实际的应用中，多个任务是协作完成全部功能的，所以现实的多任务系统中不可避免要通信。这一章将主要概述一下任务间的协作和通信。

4.1 信号量

信号量，这里分两种信号量来描述。第一种是信号灯，它只有两种状态，这和交通中用到的红绿灯相似，如果是绿灯则通过（任务将会继续执行），红灯则停车等待绿灯（程序将转换为等待态，其它任务将会被调度运行，就象路口中其它方向的车会通行一样）。

信号灯也有两种用途，第一种是作为任务之间的互斥方法，关于互斥第一章已经给出初步的回答，就是多任务的运行可能会操作共享资源，操作过程中共享资源不可以被其它的任务所操作。第一个先操作的任务将会占用信号量，其它的任务只能等待这个任务把信号量释放。这种情况下把信号量的初值设置为 1，占用时变为 0，释放时再为 1（初始时为绿灯，占用时把它变成红灯，释放时再变绿）。这样就保证多任务进入操作共享资源时是互斥的（路口是互斥的，只可以一个方面的车通过，否则的话会堵车，对应程序来说会出错）。

临界资源（Critical Resource）：系统中某些资源一次只允许一个任务使用，称这样的资源为临界资源或互斥资源或共享变量。

临界区（互斥区）(Critical Section)：一个程序片段的集合，这些程序片段分散在不同的任务中，对某个共享的数据结构（共享资源）进行操作。在任务中涉及到临界资源的程序段叫临界区，多个任务的临界区称为相关临界区。

使用互斥区的原则：

有空让进：当无任务在互斥区时，任何有权使用互斥区的任务都可进入。

无空等待：不允许两个以上的任务同时进入互斥区。

多中择一：当没有任务在临界区，而同时有多个任务要求进入临界区，只能让其中之一进入临界区，其它任务必须等待。

有限等待：任何进入互斥区的要求应在有限的时间内得到满足。

让权等待：处于等待状态的任务应放弃占用 CPU，以使其它任务有机会得到 CPU 的使用权。

例 3、控制 B 口亮灯，一个任务用来从左到右慢速度移动，直到 8 个灯移动完成，这时通知第二个任务运行，第二个任务将以比较快的速度从右到左的移动亮灯，再通知第一个任务，如此反复。这个程序可以使用软件仿真来观察效果

```
//
// example 3: semaphore
// descriptions: The first task turned on LEDs from left to right,moved slowly until the eighth LED lighten,
//                then the second task started up , it turned on LEDs from right to left,moved fleetly
//                    until the first LED lighten,and then the first task started up again.The program repeat endless.
// author: Taiyun Wang
// date: 2003/2/22
```

```
//////////////////////////////////////////////////////////////////////
#define CREATEVAR
#include "spos.h"

int err;                                //error No
int t1stack[20];                        //Task 1 stack
int t2stack[20];                        //Task 2 stack
OS_EVENT sem;                           //Event handle

volatile unsigned int *P_IOB_DATA =(unsigned int*)(0x7005);        //Port B data register
volatile unsigned int *P_IOB_DIR =(unsigned int*)(0x7007);         //Port B direction register
volatile unsigned int *P_IOB_ATTRIB = (unsigned int*)(0x7008);    //port B attribute register

main()
{
 void Task1();
 unOSInit();                                        //Initialize OS kernel
    *P_IOB_DIR = 0XFFFF;                            //Set port B direction
    *P_IOB_ATTRIB = 0XFFFF;                             //Set prot B attribute
 err = unOSTaskCreate(Task1,0,t1stack+19,1);            //Create first task
 sem = unOSSemCreate(1);                            //Create a semaphone
 unOSStart();                                       //Start kernel
}

void Task1()                                        //task 1
{
      void Task2();
      unsigned int i = 1;
 err = unOSTaskCreate(Task2,0,t2stack+19,2);            //Create second task
 while(1) {
          unOSSemPend(sem,0);                       // Waiting semaphone
        for(i = 1;i<0x100;i<<=1) {
            *P_IOB_DATA = i;
            unOSTimeDly(30);                        //Delay 30 tick
        }
          unOSSemPost(sem);                         //Release semaphone
 }
}

void Task2()                                        //task 2
{
```

```
volatile unsigned int *P_IOB_DATA = (unsigned int*)(0x7005);
unsigned int i = 0x80;
while(1) {
        unOSSemPend(sem,0);                          //Waiting semaphone
    for(i = 0x80;i != 0;i>>=1) {
            *P_IOB_DATA = i;
            unOSTimeDly(5);                          //Delay 5 tick
    }
        unOSSemPost(sem);                            //Release semaphone
}
}
```

上例中，第一个任务先得到信号量，它一直运行到 8 个灯连续点亮，然后释放了信号量，第二个任务得到信号量而运行，这两个任务同时操作 B 口，使用信号量来完成互斥。顺便提一下，第二个任务是在第一个任务中建立的，这演示了任务不一定在启动核心之前全部建立，也可以由其它的任务建立。但要注意，核心启动之前一定要建立一个任务，虽然第二个任务在第一个任务之中建立，但两个任务是并发执行的，这一点请读者一定要搞清楚，可以设断点试试看。

第二种信号灯的用途是做同步用的，比如运行的任务要等待另一个任务运行到一部分时再运行下面程序，使用这种用法。但初始值（建立信号量时）为 0（这会使该任务转为等待态，等待信号量为 1 时再运行），等待另一个任务把它设置为 1，然后任务将继续运行下去，运行完后再次设置为 0。这样这个任务就和另一个任务完成了同步过程。

任务的同步：synchronism 指系统中多个任务中发生的事件存在某种时序关系，需要相互合作，共同完成一项任务。具体说，一个任务运行到某一点时要求另一伙伴任务为它提供消息，在未获得消息之前，该任务处于等待状态，获得消息后被唤醒进入就绪态。

例 4、这个程序演示了同步，第一个任务等一个信号量，等到后把灯点亮，并移一位，为下次点亮下一个灯做准备。第二个任务定时（0.5 秒）发一个同步信号给第一个任务，让第一个任务点亮下一个灯，这里是用定时来实现的，主要是为了能使读者不用外接任何的硬件就可以完成实验，甚至可以不用接 ICE，在 IDE 中的软件仿真就可以运行。第二个任务可以改成键盘扫描，如有键按下则发信号量给第一个任务。具体读者可以自己试一下。读者应该亲身体验一下效果。

```
//
// example 4: task synchronization
// descriptions: The first task wait for a semaphore,when the semaphore is Realeased,
//               this task turn on LEDs one by one.
//               The second task sends a semaphore at the rate of 0.5 second.
// author: Taiyun Wang
// date: 2003/2/22
//////////////////////////////////////////////////////////////////////////////

#define CREATEVAR
#include "spos.h"
```

```
int err;                        //Error No
int t1stack[20];                //Task 1 stack
int t2stack[20];                //Task 2 stack
OS_EVENT sem;                   //Event handle

volatile unsigned int *P_IOB_DATA =(unsigned int*)(0x7005);      //Port B data register
volatile unsigned int *P_IOB_DIR =(unsigned int*)(0x7007);           //Port B dirction register
volatile unsigned int *P_IOB_ATTRIB = (unsigned int*)(0x7008);       //Port B attribue register

main()
{
 void Task1();
 unOSInit();
 *P_IOB_DIR = 0XFFFF;                                   //Set Port B output
 *P_IOB_ATTRIB = 0XFFFF;                                //Set Port B attribute
 err = unOSTaskCreate(Task1,0,t1stack+19,1); //Create first task
 sem = unOSSemCreate(0);                                //Create semaphore
 unOSStart();                                           //Start kernel
}
void Task1()                                            //Task one
{
      void Task2();
      unsigned int i = 1;
     err = unOSTaskCreate(Task2,0,t2stack+19,2);        //Create second task
        while(1) {
        for(i = 1;i<0x100;i<<=1) {
              unOSSemPend(sem,0);                       //Waiting semaphore
              *P_IOB_DATA = i;
        }
    }
}

void Task2()
{
 while(1) {
            unOSSemPost(sem);                           //Realease semaphore
            unOSTimeDly(32);                            //Delay 64 tick
 }
}
```

这个程序是任务 1 等待信号量，而任务 2 释放信号量给任务 1，让任务 1 从等待态转为就绪态，继而转为运行态，然后再次进入等待态，周而复始。而第二个任务是每隔 64 个 tick 就释放信号量

给任务 1，所以外在就表现为灯每隔半秒钟就会移动一次。

以上是信号灯的两种应用，另外一种信号量是计数式的信号量。信号灯和信号量在内核不做区别，只是在建立时初值设置的不一样。信号灯值是 1 或 0，如果不是这两样，那就是计数的信号量了。它的机制是这样的，在初始时给它一个值。在程序中先用 PEND 等信号量，如果这时信号量不为 0 则把信号量减 1，且这个任务继续执行，如果为 0 则等待（这个任务会变为等待态，并调度其它的任务运行）。而释放的过程相反，如果有任务在等这个信号量，就把那个任务转为就绪态，否则把信号量加 1。这里还可能出现一种情况，那就是可能有多个任务在等这个信号量，把哪个任务变为就绪呢？MiniOS 是占先式内核，所以最高优先级的任务会得到信号量。低优先级的只能再等了！所以信号量在初始值大于 1 时，就说明可以有两个任务得到信号量或是一个任务两次得到信号量。一种应用比如 N 个共享资源可以被分配使用，则可以把信号量初始化为 N，这样任务请求分配时，只要信号量不为 0（即还有共享资源，信号量为 m，m<N，则表示还有 m 个资源可用），就可以使任务得到，并分配到共享资源。直到为 0 时不可分配，这时再次请求的任务将会被转换为等待态。

例 5、这程序演示计数式信号量的用法，也用灯来演示，程序中把信号量初始化为 3，也就是同时可以有 3 个灯亮（注意，其它灯也是可以点亮的，但要得到信号量，如果得不到，也就不能点亮），每个任务对应一个灯，一共启动了 8 个任务，这 8 个任务将抢 3 个资源，得到资源将把自己对应的灯点亮，同时可以点亮 3 个灯，点亮后延时一段时间，将灯熄灭，并释放信号量，让其它的任务占有信号量并运行。所以每个任务都有机会运行并点亮自己的灯，但有个条件，就是不能有三个以上的灯同时点亮。三个灯的情况比较复杂，用户看不清同时有几个灯在亮，如果把它改成 2，看得就比较清楚了，但由于系统运行抢占的原因，有些灯可能就亮不了。这个就是任务的饥饿，也就是有些任务根本没机会运行。读者可以把信号量的初值改动一下，看看有什么结果。这个程序可以在软件仿真环境下运行。

```
//
// example 5: counting semaphore
// descriptions: Semaphore initialized is 3, thus 3 LEDs can be turned on at the same time.
//                There are 8 initialized tasks,one task corresponds to one LED,they share the
//                   resource each other. Each task occupys the semaphore some cycles, then it releases
//                the semaphore,so other task can gain semaphore and turn on its own LED.
// author: Taiyun Wang
// date: 2003/2/22
////////////////////////////////////////////////////////////////////////////
#define CREATEVAR
#include "spos.h"

typedef struct {
 int portval;
 int dlyval;
} par_t;                                    //parameter struct
int err;                                    //Error No
int stack[8][25];                    //Task stack
```

```
OS_EVENT sem;                          //Event handle

volatile unsigned int *P_IOB_DATA =(unsigned int*)(0x7005);      //Port B data register
volatile unsigned int *P_IOB_DIR =(unsigned int*)(0x7007);       //Port B direction register
volatile unsigned int *P_IOB_ATTRIB = (unsigned int*)(0x7008);   //Port B attribute register
int portb = 0;

main()
{
 void Task(void* inval);
 int i;
 par_t par[8];                                                   //structure array
 int dlypar[8] = {53,43,31,29,23,19,17,13};                      //Delay tick of every task
 unOSInit();
 *P_IOB_DIR = 0XFFFF;                                            //Set Port B is output
 *P_IOB_ATTRIB = 0XFFFF;                                         //Set Port B attribute
 for(i = 0;i<8;i++) {
        par[i].portval = 1<<i;
        par[i].dlyval = dlypar[i];
 }
 for (i = 0; i<8;i++)
        err = unOSTaskCreate(Task,(void*)&par[i],&stack[i][24],i+1); //Create task
 sem = unOSSemCreate(3);                                         //Create semaphore
 unOSStart();                                                    //Start OS kernel
}

void Task(void* inval)                                  //task n(n=1...8)
{
 while(1) {
        unOSSemPend(sem,0);                                      //Waiting semaphore
        portb |= ((par_t*)inval)->portval;
        *P_IOB_DATA = portb;
        unOSTimeDly(((par_t*)inval)->dlyval);
        portb &= ~((par_t*)inval)->portval;
        *P_IOB_DATA = portb;
        unOSSemPost(sem);                                        //Release semaphore
        unOSTimeDly(((par_t*)inval)->dlyval);                    //Delay
 }
}
```

这个程序的建立比较特殊，一个程序建立了 8 个复本，这不影响 8 个任务的独立运行。程序区是共享的，但每个任务有自己的堆栈空间，每个任务在运行时等待事件发生的点也不同，即当时的

程序计数器（PC）的值不同，所以这是 8 个独立的任务。另外一点要说明的是，建立任务时是通过传递参数的形式建立的，传参时要注意，传的是一个任意数（或结构）的指针，用 void＊的形式能保证可以传任何数据类型的指针，这里是一个结构的指针。结构中的第一个数是用来说明这个任务对应的是哪个灯，第二个参数说明的是任务的延时时间数。这个时间数是开发人员特意选定的素数，保证每个任务的延时时间不太一样，以让所有的任务都有机会运行。

4.2 邮箱

相信通过上一节的描述，读者已经对同步及互斥有一定的了解了，可是任务之间还要通信，当然可以用信号量的方法解决通信的问题。做法是使用公用变量，并让两个任务互斥的读写，这样就可以完成简单的通信。这样做有两个问题：第一，要申请公用变量，传递的数过多的话，将会使程序变复杂；第二，信号量用得不好容易产生错误，而且还要求申请的公用变量互斥，使程序的可读性差。MiniOS 在内核中使用邮箱来解决上述问题，一个邮箱可以放一个 void*的指针，可以指向任意的数据类型。接收任务时检查邮箱有没有邮件，如果没有将会等待邮件的到来（这个任务将会被转换成等待态，其它任务将会运行）。发送任务时会将任务等待列表中的最高优先级的任务变为就绪态，那个任务将会运行并处理收到的邮件。所以邮箱不但可以使任务同步，而且可以向任务发送数据。

例 6、这个例子将演示这样一个功能，一个任务以二进制灯的形式显示一个 char 型的值，它在等待另外一个任务通知它要显示的值。另外一个任务将计算开机运行以来程序运行的秒数（模２５６，以保证它的数值在２５５以内），发送到任务 1 用来显示。需要说明一下，本程序考虑可以软件仿真运行，所以以二进制灯的形式演示，给用户观看带来一定的困难，如果读者有趣并有条件的话，可以改动程序，使之配合ＬＥＤ显示十进制的数，则程序运行的可视效果将提高。

```
//
// example 6: Mail box
// descriptions: First task displays the value of byte in binary way,
//                  the second task sends the value to the first task in runing time .
// author: Taiyun Wang
// date: 2003/2/22
//////////////////////////////////////////////////////////////////////////
#define CREATEVAR
#include "spos.h"

int err;                                //Error No
int t1stack[25];                        //Task 1 stack
int t2stack[25];                        //Task 2 stack
OS_EVENT mailbox;                       //Event handle

volatile unsigned int *P_IOB_DATA =(unsigned int*)(0x7005);         //Port B data register
volatile unsigned int *P_IOB_DIR =(unsigned int*)(0x7007);              //Port B direction register
volatile unsigned int *P_IOB_ATTRIB = (unsigned int*)(0x7008)    ;      //Port B attribute register
int portb = 0;
```

```
main()
{
 void Task1();
 void Task2();
 unOSInit();
 *P_IOB_DIR = 0XFFFF;                          //Set Port B is output
 *P_IOB_ATTRIB = 0XFFFF;                       //Set Port B arribute
 err = unOSTaskCreate(Task1,0,t1stack+24,1);   //Create first task
 err = unOSTaskCreate(Task2,0,t2stack+24,2);   //Create second task
 mailbox = unOSMboxCreate((void*)0);           //Create mailbox
 unOSStart();                                  //Start OS kernel
}
void Task1()
{
 int msg = 0;
 unOSMboxPost(mailbox,&msg);
 while(1) {
        unOSTimeDly(128);                      //Delay 128 tick
        msg = (msg+1)%256;
        unOSMboxPost(mailbox,&msg);            //Send a mail to task 2
 }
}
void Task2()
{
 int msg;
 int * pmsg;
 int err;
 while(1) {
        pmsg = unOSMboxPend(mailbox,0,&err);       //Waiting mail
        msg = *pmsg;                          //Copy mail to local variable
        *P_IOB_DATA = msg;                    //Write mail value to prot B
 }
}
```

注意，发送邮件可以是任何数据的指针，包括结构的指针，需要指出的是邮件发送的只是指针，真实内容保存在发送者的内存区域中，邮件发送后，在接收的任务没有复制之前，如果发送者将内容更改，则接收任务读到的将是最新的内容。也就是说由程序员自己保证数据的正确性！

4.3 消息队列

上节中介绍的邮箱一次只能存发一个邮件，如果来不及处理就发第二个邮件会出现溢出错误，于是产生了消息队列。可以简单的认为消息队列是个可以放多个邮件的邮箱。可以放入邮件的多少

是建立队列时设定的。下面直接给出例子程序。

这个例子是作者精心安排的，队列不太好演示，因为很可能发送者刚发送一个消息，没等发送第二个消息，另一个任务已经把这个消息处理了，那样就不能看出消息队列的特点来了，所以作者在一个高优先级的任务中连发两个消息，处理任务直接处理两次。之所以用高优先级发送是因为高优先级的任务不能被抢占，可以保证队列中先放入的消息不被低优先级的任务处理，直到高优先级的任务放弃 CPU，低优先级的任务才可能处理。

例 7、这个例子程序和上面给出的例子相仿，只是处理方式不同，第一个任务（高优先级的任务）每隔两秒发送两个消息入队列，一个是当前的流逝过的秒数（模 256）另一个是下一次的秒数。入队列时采用的是前插入方式，即在队列头放入第二个消息。这样使第二个任务先处理第二个消息，再处理第一个消息。显示为：1、0、3、2、5、4….，处理任务（低优先级的任务）则处理第一个消息以二进制灯的形式显示，延时一秒后再显示第二个消息。这样这个程序和上个例子的外在表现基本是一样的，但请读者仔细看代码并理解其中的不同。有兴趣的读者改动一下第一个任务中队列的发送方式为后插入方式，即插入队列时把要插入的消息放入队列的尾部，这样这个程序的表现将和上一个例子程序一模一样，请读者注意实现的方式的不同。

```
//
// example 7: This program demo how to use Message queue.
// descriptions: The first task send message with front insert mode,
//                        the second task receive message and display time.
// author: Taiyun Wang
// date: 2003/2/22
//////////////////////////////////////////////////////////////////////////
#define CREATEVAR
#include "spos.h"

int err;                                //Error No
int t1stack[25];                        //Task 1 stack
int t2stack[25];                        //Task 2 stack
OS_EVENT queue;                                 //Event handle
int *Q[5];                              //Message queue array

volatile unsigned int *P_IOB_DATA =(unsigned int*)(0x7005);         //Port B data register
volatile unsigned int *P_IOB_DIR =(unsigned int*)(0x7007);                  //Port B direction register
volatile unsigned int *P_IOB_ATTRIB = (unsigned int*)(0x7008);              //Port B attribute register
int portb = 0;

main()
{
 void Task1();
 void Task2();
 unOSInit();
```

```
 *P_IOB_DIR = 0XFFFF;                          //Set Port B output
 *P_IOB_ATTRIB = 0XFFFF;                       //Set Prot B attribute
 err = unOSTaskCreate(Task1,0,t1stack+24,1);   //Create fist task
 err = unOSTaskCreate(Task2,0,t2stack+24,2);   //Create second task
 queue = unOSQCreate((void*)Q,5);              //Create Message queue
 unOSStart();                                  //Start Os kernel
}
void Task1()
{
 int msg[2];
 msg[0]=0;
 msg[1]=1;
 unOSQPostFront(queue,&msg[0]);                //insert queue head
 unOSQPostFront(queue,&msg[1]);                //insert queue head
 //unOSQPost(queue,&msg[0]);                   //insert queue
 //unOSQPost(queue,&msg[1]);                   //insert queue
 while(1) {
        unOSTimeDly(256);                      //Delay 256 tick
        msg[0] = (msg[0]+2)%256;
        msg[1] = msg[0]+1;
        unOSQPostFront(queue,&msg[0]);
        unOSQPostFront(queue,&msg[1]);
        //unOSQPost(queue,&msg[0]);
        //unOSQPost(queue,&msg[1]);
 }
}
void Task2()
{
 int msg;
 int * pmsg;
 int err;
 while(1) {
        pmsg = unOSQPend(queue,0,&err);        //Waiting first message
        msg = *pmsg;
        *P_IOB_DATA = msg;                     //Send first message to prot B
        unOSTimeDly(128);
        pmsg = unOSQPend(queue,0,&err);        //Waiting second message
        msg = *pmsg;
        *P_IOB_DATA = msg;                     //Send second message to prot B
        unOSTimeDly(128);
 }
```

}

4.4 总结

这一章我们给出了 MiniOS 的同步、互斥、通信的概念及一些演示程序。尽力使读者有一个总体的认识，了解其基本概念及用法，下面给出相关函数的列表，以方便读者查找和理解。这里只给出目前为止用过的函数，以帮助读者容易理解其功能，函数中没有给出入口参数和返回值，只是让读者有一个总体上的理解。如果必要的话可以查本书的附录，那里给出每个 API 的详细说明。

1. 信号量
   建立　　unOSSemCreate()
   等待　　unOSSemPend()
   发送　　unOSSemPost()
2. 邮箱
   建立　　unOSMboxCreate()
   等待　　unOSMboxPend()
   发送　　unOSMboxPost()
3. 消息队列
   建立　　unOSQCreate()
   等待　　unOSQPend()
   插入队列尾　　unOSQPost()
   插入队列头　　unOSQPostFront()

例 8、为开拓读者的思路，本章最后给出一个比较复杂的例子程序，这个程序运行过程是：

1. 由主程序建立任务 1 并启动核心；
2. 由任务 1 建立任务 2，并通过邮件向任务 2 发送任务 3 的结构，包括任务 3 的启动地址、参数等，并每隔 17tick，就把灯点亮；
3. 在任务 2 中接收邮件，由于邮件内容是任务 3 的参数结构，所以可以建立任务 3。并每隔 11 个 tick 把自己的灯点亮；
4. 任务 3 运行后先延时 4 秒钟，以让任务 1 和任务 2 的灯交替亮一会儿。然后任务 3 将会把任务 1 及任务 2 删除，到这时只有任务 3 会运行了，任务 1 和 2 永远不会再运行了。任务 3 采用移灯显示。

```
//
// example 8: Use mail to create complex task:
//              1.The main program creates first task and runs OS kenel.
//              2.First task creates second task,
//              send the third task's structure including parameter and address and so on
//              and turn on lamp every other 17 tick.
//              3. Second task receives mailbox create third task and turn on lamp every other 11 tick.
//              4. After third task delay 4 second, it delete first task and second task and display lamp.
// author: Taiyun Wang
// date:2003/2/22
////////////////////////////////////////////////////////////////////////////
```

```
#define CREATEVAR
#include "spos.h"

typedef struct {
 void (*start_add)(void *para);          //function pointer
 void* para;                             //task parameter
 int* stacktop;                          //task stack size
 int priority;                           //task priority
} stru_task;                             //structure name

int err;                                 //Error No
int t1stack[25];                         //Task 1 stack
int t2stack[25];                         //Task 2 stack
int t3stack[25];                         //Task 3 stack
OS_EVENT mailbox;                        //Event handle


volatile unsigned int *P_IOB_DATA =(unsigned int*)(0x7005);      //Port B data register
volatile unsigned int *P_IOB_DIR =(unsigned int*)(0x7007);           //Port B direction register
volatile unsigned int *P_IOB_ATTRIB = (unsigned int*)(0x7008);       //Port B attribute register
int portb = 0;

main()
{
 void Task1();
 unOSInit();
 *P_IOB_DIR = 0XFFFF;                                    //Set port B direction
 *P_IOB_ATTRIB = 0XFFFF;                                 //Set prot B attribute
 err = unOSTaskCreate(Task1,0,t1stack+24,1);             //Create first task
 mailbox = unOSMboxCreate((void*)0);                     //create mailbox
 unOSStart();                                            //Start OS kernel
}
void Task1()
{
 void Task2();
 void Task3();
 stru_task msg;
 msg.start_add = Task3;
 msg.para = (void*) 0;
 msg.stacktop = t3stack+24;
 msg.priority = 3;
```

```
 err = unOSTaskCreate(Task2,0,t2stack+24,2);                    //Create second task
 unOSTimeDly(20);                                               //Delay 20 tick,running task2
 unOSMboxPost(mailbox,&msg);                                    //Send mail to task 2
 while(1) {
        *P_IOB_DATA = 0x55;
        unOSTimeDly(17);
 }
}
void Task2()
{
 stru_task msg;
 stru_task* pmsg;
 int err;
 pmsg = unOSMboxPend(mailbox,0,&err);                           //Waiting mail
 msg = *pmsg;                                                   //copy mail to local variable
 //create task 3
 err = unOSTaskCreate(msg.start_add,msg.para,msg.stacktop,msg.priority);
 while(1) {
        *P_IOB_DATA = 0xAA;
        unOSTimeDly(11);
 }
}
void Task3()
{
 int i = 1;
 unOSTimeDly(512);                                              //Delay 512 tick
 unOSTaskDel(1);                                                //Delete task 1
 unOSTaskDel(2);                                                //Delete task 2
 while(1) {
        *P_IOB_DATA = i;
        i <<=1;
        if (i == 0x100)
              i = 1;
        unOSTimeDly(20);                                        //Delay 20 tick
 }
}
```

这个程序主要让读者对任务和邮件有更深一步的认识和理解。有如下优点：

1. 虽然任务 1 建立任务 2，任务 2 又建立了任务 3，但任务之间没有从属关系，他们会并发的运行。注意没有从属关系并不是说没有优先的关系，在这个程序中任务 1 的优先级最高，任务 2 次之，任务 3 最低。也就是说如果任务有两个以上进入就绪的话，优先级

最高的将被运行，优先级低的只能等。甚至任务 3 将建立它的任务 2 以及建立任务 2 的任务 1 都删除了!!!

2. 邮件的内容可以很广泛，这里演示了一个邮件的内容是建立另一个任务的必要信息。并把它发送到了任务 2。且由任务 2 建立了邮件内容的那个任务（即任务 3）。在被建立的任务 3 中又将任务 1 及 2 删除了。

读者读到现在应该停一下好好看看这个比较复杂的程序，仔细研读一下，加深理解。

5 中断过程及处理

5.1 中断处理过程

有 OS 和没有 OS 的中断操作过程不太一样，在这一章里将详细介绍 OS 处理中断的过程。先说一点概念然后说明操作方式。

在多任务系统中，各个任务并发执行，中断发生时被中断的任务是不确定的，而中断函数还要用到堆栈空间，因此所建立的每个任务都要保留给中断栈所用的空间，这无疑浪费了堆栈的空间。MiniOS 提供系统栈，即中断发生时，中断的前期处理程序会将用户的堆栈指针更换为系统事先留出的空间中，等中断退出时再恢复用户的堆栈指针。这样中断将不再占用户的堆栈空间，大大提高内存空间的利用率，且随着任务的增加，这种技术的效果越明显。但用户要注意并不代表一点空间也不占用，中断发生时硬件将会把用户程序被中断处的下一条指令地址及 SR 压入用户栈，加上中断退出时要调用调度函数，所以会占用用户栈 4 个字的空间。

用户在看本书之前应该比较了解凌阳的这款 CPU，如果还不太清楚请复习一下中断部分。CPU 的中断通过汇编定义，中断向量的地址是固定的。MiniOS 的处理是，在中断发生时先做前期处理，将堆栈换成系统栈，然后调用用户定义的中断函数。

系统中有一个结构，其中一个域是一个指向函数的指针，该指针就是第一个中断函数的入口地址。另一个域指向下一个结构的地址。也就是一个链表。这样就可以保证有多个中断函数可以使用，处理完第一个中断，系统会自动处理第二个…，直到这个中断对应的函数全部处理完成，然后把系统栈换回用户栈。最后系统根据情况判断是调用调度函数完成调度，还是返回到用户中断点的下一条指令继续执行。

多中断函数有什么优点呢？其实很明显，如果第三方提供了库函数，而库函数中用到了中断，而且不向用户公开源代码，其他用户就无法在原中断中加入自己的功能；而多中断函数向用户提供了在原中断函数后加入其它的功能的方法，且不会影响原有的功能。但用户应注意，不要滥用这个功能，否则将会加大中断延迟时间。

用户的中断由 MiniOS 管理，这就是说，用户中断函数必须通过 MiniOS 提供的 API 来处理。MiniOS 提供如下 API 来帮助用户管理中断函数。为了使用户有更加全面的了解，这里只写出具体的功能，对于参数没有详细说明，用户可以查本书的附录。

int unOSRegisterISR (int Vector_No,void (*pISR)());

这个函数用来将用户的中断处理函数挂到 MiniOS 上。第一个参数是中断号，第二参数是一个指向函数的指针，即用户中断函数名，系统会将用户定义的中断函数挂接到中断结构链表的尾部。

int unOSReleaseISR (int Vector_No,void (*pISR)());

这个函数用来将用户的中断处理函数从 MiniOS 上摘除。第一个参数是中断号，第二参数是一个指向函数的指针，即用户中断函数名，系统会将用户定义的中断函数从中断结构链表中删除。

unsigned long SpFGetINTVec(void);

这个函数返回中断源的开关状态。

void unOSEnINTSrc(unsigned long ulVector);

这个函数是用来开中断的，把中断函数挂到 OS 上并不说明该中断就打开了，需要用本函数将对应的中断设置为开，使中断有效。

void unOSDisINTSrc(unsigned long ulVector);

这个函数将关闭中断源，使该中断源的中断不再被响应。

看了 MiniOS 这 4 个中断管理，读者是不是清楚了 MiniOS 的中断处理过程呢？现在给出一些程序的片段，来说明怎样使用 MiniOS 的中断管理 API 来管理用户自己的中断函数。注意这些只是为了使用户更了解中断 API 的使用过程，程序片段本身不能运行。

```
Void Interrupt_function();
Unsinged int t1stack[20];
Main()
{
    unOSInit();                 //初始化核心
    //以下函数将把 Interrupt_function()这个函数挂到操作系统中断号为 1 的代码下。
    //被操作系统中断管理程序所管理，并在中断号为 1 的中断发生时调用中断函数。
    //注意，这时中断还没有开，只是把中断函数挂上去了，还要开中断，中断才能被响应
    unOSRegisterISR(0x1,Interrupt_fuction);
    //这个函数将把中断号为 1 的中断源打开，使中断变有效，这样有中断才会被响应。
    unOSEnINTSrc(0x1);
    tid1 = unOSTaskCreate(Task1,0,t1stack+19,1);
    unOSStart();
}
    //这个函数就是中断函数的主体了。
Void Interrupt_function()
{
    //代码略
    …
    …
}
```

中断就象程序片段所示的一样被挂到 OS 上，并被 OS 所管理，读者注意程序的第 8 行：

unOSRegisterISR(0x1,Interrupt_function);

第一个参数为 0x1，指中断号，作者在这里是故意这样写的,，只是为了让读者更清楚为什么以及怎样定义中断函数，用户程序应该避免这样写。中断号已经在对应的 body.h 中用 define 定义了，读者可以看一下这个文件,以下是 spce061a 中定义的：

```
#define     FIQ_PWM_VEC     0
#define     FIQ_TMA_VEC         1
#define     FIQ_TMB_VEC     2
#define     IRQ0_PWM_VEC    3
#define     IRQ1_TMA_VEC    4
#define     IRQ2_TMB_VEC    5
#define     IRQ3_EXT2_VEC   6
#define     IRQ3_EXT1_VEC   7
#define     IRQ3_KEY_VEC    8
#define     IRQ4_4KHZ_VEC   9
#define     IRQ4_2KHZ_VEC   10
#define     IRQ4_1KHZ_VEC   11
#define     IRQ5_4HZ_VEC    12
#define     IRQ5_2HZ_VEC    13
#define     IRQ6_TMB1_VEC   14
#define     IRQ6_TMB2_VEC   15
#define     UART_RX_VEC     16
#define     UART_TX_VEC     17
```

用户应该尽量使用头文件中的定义，这样原来的函数就可以改成如下：

```
unOSRegisterISR(FIQ_PWM_VEC,Interrupt_function);
```

这样更好理解，一看就知道设置的是 PWM 中断函数。

5.2 在中断中的通信

在嵌入式系统中，事件的发生多是由中断触发的，所以不可避免的要在中断中和任务通信，可是中断中的任务通信有一些限制。

1. 因为中断函数不从属于任何任务，中断函数也不是任务。所以中断函数是不可以被挂起的（即没有等待态），只能是中断发生后运行。也就是说事件的 Pend 将是非法操作。比如：unOSSemPend, SpSMboxpend, unOSQPend 等在中断中是不可使用的。但中断中可以给其它的任务发送事件、邮件、消息，以通知事件的到来。读者会问，那任务可以给中断发送事件吗？回答当然是可以的，但用的不是 Pend 而是 Accept, 这些函数(如 unOSSemAccept、unOSQAccept 等函数)在没有事件到来时直接返回一个代码通知当前程序没有事件发生，而不会把该程序转变为等待态，等待被发送的事件发生。
2. 另一点要注意的是，在中断中给其它任务发送事件后系统不会马上调度，因为现在还在中断之中！要等中断退出时系统才会自动调用调度函数。所以，如果用户的中断程序过长或是有很多的中断函数，将会延长任务被调度的时间。如果用户的任务要求实时性比较高，就要注意中断函数的执行时间，以确保实时性能满足设计需求。

例 9、现在给出中断过程一个例子程序，外部表现和例 2 是一样的，但实现方式却不同，这里写了一个中断函数给任务 1 和任务 2 发送信号量。用户注意一下，这个中断函数是挂到系统时钟中

断里的，即它挂在 tick 中断后面，Timer tick 中断执行完后就执行用户的中断函数，然后才退出。这个例子演示了一个中断源挂接多个中断函数的情况，也演示了在中断中给用户发事件的情况。演示程序不需要ＩＣＥ，只软件仿真即可。

```
//
// example 9: Send message in interrupt
// descriptions: The interrupt function send semaphore to task 1 and task 2.
//                      pay attention to the interrupt function that lie in system clock interrupt.
// author: Taiyun Wang
// date: 2003/2/22
//////////////////////////////////////////////////////////////////////////
#define CREATEVAR
#include "spos.h"

int err;                                //Error No
int t1stack[20];                        //Task 1 stack
int t2stack[20];                        //Task 2 stack
OS_EVENT sem1,sem2;                             //Event handle
int tick1 = 0,tick2 = 0;

volatile unsigned int *P_IOA_DATA =(unsigned int*)(0x7000);       //Port A data register
volatile unsigned int *P_IOA_DIR =(unsigned int*)(0x7002);        //Port A direction register
volatile unsigned int *P_IOA_ATTRIB = (unsigned int*)(0x7003);          //Port A attribute register

volatile unsigned int *P_IOB_DATA =(unsigned int*)(0x7005);       //Port B data register
volatile unsigned int *P_IOB_DIR =(unsigned int*)(0x7007);              //Port B direction register
volatile unsigned int *P_IOB_ATTRIB = (unsigned int*)(0x7008);          //Port B attribute register

main()
{
 void Timer_INT(void);
 void Task1();
 void Task2();
 unOSInit();
 *P_IOA_DIR = 0XFFFF;                                   //Set Port A direction
 *P_IOA_ATTRIB = 0XFFFF;                                //Set Port A attribute

 *P_IOB_DIR = 0XFFFF;                                   //Set Port B direction
 *P_IOB_ATTRIB = 0XFFFF;                                //Set Port B attribute
 unOSRegisterISR(IRQ6_TMB2_VEC,Timer_INT);              //Set interrupt function
 err = unOSTaskCreate(Task1,0,t1stack+19,1);            //Create first task
 err = unOSTaskCreate(Task2,0,t2stack+19,2);            //Create second task
```

```
 sem1 = unOSSemCreate(0);                    //Create semaphore
 sem2 = unOSSemCreate(0);
 unOSStart();                                //Start OS kernel
}

void Task1()                                 //task one
{
 unsigned int i = 1;
 while(1) {
        unOSSemPend(sem1,0);                 //Waiting semaphore
        *P_IOB_DATA = i;
        i<<=1;
        if(i == 0x0100)
              i = 1;
 }
}

void Task2()                                 //task two
{
 unsigned int i=0;
 while(1) {
        unOSSemPend(sem2,0);                 //Waiting semaphore
        *P_IOA_DATA = i;
        i<<=1;
        if(i == 0)
              i = 1;
 }
}
void Timer_INT(void)
{
 tick1++;
 tick2++;
 if (tick1 == 30) {                          //tick==30?
        unOSSemPost(sem1);                   //Release semaphore 1 ,running task 1
        tick1 = 0;
 }
 if (tick2 == 5) {                           //tick==5
        unOSSemPost(sem2);                   //Release semaphore 2 ,running task 2
        tick2 = 0;
 }
}
```

6 驱动程序

本章主要针对一些比较高级的用户，特别是为想提供驱动程序库的读者而写。把程序分类有利于提高编程效率。而对自己写一些应用简单的程序，比如前面所举的例子，也在操作硬件，但由于大多数情况一个硬件只有一个任务在操作，不存在共享的问题，则用户也可以直接操作，但库的编写者应该避免用这种写法，因为不能要求用户保证只有一个任务在调用这个库！嵌入式系统不可避免的要和硬件相关，为了增加程序的可靠性，把程序分为两类，一类为驱动程序，另一类才是通常的用户程序。驱动将和硬件一起完成一些规定的功能，驱动程序将有利于系统的维护，提高可靠性以及用户程序的安全性，本章主要说明一些驱动程序的编写方法和一般原则。

驱动程序一般都和硬件的地址相关，并有相当部分的硬件使用中断，MiniOS 的驱动程序和一般任务没有太大的区别，驱动程序本身就是一个任务。这和 Windows 及 Linux 的驱动有相同之处，也有很多的不同。下面给出它们的相同点及不同点，以使读者有一个更加全面的了解。

相同点：

1. 它们都对硬件相关部分直接操作；
2. 相关中断都会被操作系统所管理；
3. 都会提供资源共享，解决两个以上任务的资源分配；
4. 都会让用户不直接跟硬件打交道，提供一个良好的硬件接口。

不同点：

Windows 2000 是在 386 保护模式下运行的，操作系统将会保证用户的程序不可以直接读写硬件端口，否则的话会出现非法的操作异常。所以硬件驱动都是用 DDK 写，写过 DDK 的读者一定对 DDK 的难度有一定认识。Linux 系统也一样在 386 的保护模式运行，编写驱动也困难。象 Windows 和 Linux 这样做有很多的好处，硬件决定有一部分代码在用户态无法运行，所以安全性比较高，特别是在多用户的状态下。而 MiniOS 只提供多线程，所有任务共享内存空间，不提供硬件保护。所以驱动程序和一般任务就没有太多区别。这样驱动程序的编写就容易很多，而嵌入式的应用没有安全性的问题，也就是说由程序员保证一个任务不恶意的破坏别的任务的运行环境和参数。而这种简单的驱动也会提高整体的运行效率，主要是因为不必为保护提供更多的代码，保证运行高效。

下面说一下驱动程序编写的一般原则：

1. 驱动程序和硬件相关，属于资源，要解决资源共享的问题，以防止 2 个或是 2 个以上的任务同时使用相同的硬件资源。也就是用信号量来保证互斥，对于单个资源要用信号灯，而对于多个资源则用计数式信号量来实现。
2. 注意中断的处理，一般硬件相关的程序都会涉及中断问题，要正确处理并估计中断所占用之时间满足系统设计之要求。因为中断函数运行期间其它任务不可能运行，而所占用之中断还可能有其它的中断函数。

比如 UART 驱动，就可以用信号灯保证，一个任务在占用 UART（发送或接收）时，不允许有其它的任务也占用 UART。而对于 LED，就可以用计数式信号量来解决，只要还有灯可用就可分配给任务，直到所有都分配完，再申请的任务只能等待。关于这两种用法和例子读者可以从前面的例子中细细体会。

7 综合应用

本章将给出一些比较复杂的例子程序，并加以详细说明。给出的这两个应用都与声音有关，所以这两个例子要求运行在 ICE 上才能体现它的效果。

例 10、原声播放，播放的声音是未经压缩的声音。这里给出全部源代码，使读者对声音和 MiniOS 的结合有一个比较深入的认识。本例由两个任务和一个中断组成，其中移灯任务和前面所举例子没有太大的区别，给出它只是说明这些任务是并发执行的。任务模拟解码的过程，这个程序的解码没有意义，只是为了下一个例子做准备工作。解码工作过程是这样的，先由解码程序把它压缩后的声音展开并把解压后的数据放入缓冲区中，等待播放，播放完一个缓冲区的内容后通知解码任务（这里用的是信号量的方式）让解码任务解下一帧的内容，而播放另一个缓冲区的内容以保证播放声音的连续性。如此反复让两个缓冲区交替工作在解码及播放过程中。这里的演示只是将原声放入缓冲区中，并无解码过程。特别提示：这个程序要 ICE 才能运行，所需要的硬件只有一个扬声器，把扬声器接到任意一个接口上即可（对应 ICE 上的 J24 或是 J25）。

```
//
// example 10: This example Plays original sound on OS, explain how to Play sound on OS.
// author: Taiyun Wang
// date:2003/2/22
////////////////////////////////////////////////////////////////////////////
#define CREATEVAR
#include "spos.h"

#define BUFLEN 60
#define BUFSIZE 2

extern long RES_ORIG2_RAW_SA;
extern long RES_ORIG2_RAW_EA;


unsigned int* pStart;
unsigned int* pEnd;
int SegStart;
int SegEnd;
int playpointer = 0;
int playbuf = 0;

int buff[BUFSIZE][BUFLEN];              //sound buffer

int err;                                //Error No
```

```
int t1stack[32];                    //Task 1 stack
int t2stack[20];                    //Task 2 stack
OS_EVENT sem;                       //Event handle

volatile unsigned int *P_IOB_DATA =(unsigned int*)(0x7005);      //Port B data register
volatile unsigned int *P_IOB_DIR =(unsigned int*)(0x7007);       //Port B direction register
volatile unsigned int *P_IOB_ATTRIB = (unsigned int*)(0x7008);       //Port B attribute register


volatile unsigned int* P_DAC_Ctrl = (unsigned int*) 0x7015;      //DAC control register
volatile unsigned int* P_DAC1 = (unsigned int*) 0x7017;          //DAC1 data register
volatile unsigned int* P_DAC2 = (unsigned int*) 0x7016;          //DAC2 data register

void int_func(void);                                        //interrupt function

main()
{
 void Task1();
 void Task2();
 void DACInit();
 unOSInit();
 *P_IOB_DIR = 0XFFFF;                                       //Set Port B direction
 *P_IOB_ATTRIB = 0XFFFF;                                    //Set Port B attribute
 DACInit();
 *P_DAC1 = 0x0;
 *P_DAC2 = 0x0;

 err = unOSTaskCreate(Task1,(void*)0,t1stack+31,1);             //Create first task
 err = unOSTaskCreate(Task2,(void*)0,t2stack+19,2);             //Create second task
 sem = unOSSemCreate(0);
 unOSStart();                                               //Start OS kernel
}
void DACInit()
{
 pStart =      (unsigned int *)(int)RES_ORIG2_RAW_SA;       //Start pointer
 pEnd =          (unsigned int *)(int)RES_ORIG2_RAW_EA;     //End pointer
 SegStart =   (unsigned int)(RES_ORIG2_RAW_SA>>6)&0xFC00;   //Start segment
 SegEnd =      (unsigned int)(RES_ORIG2_RAW_EA>>6)&0xFC00;//End segment
 SpFSetTimer(OS_SET_TIMEA,0x0030,0xFA23);
}
void Task1()
```

```
{
 int fillbuf(int buffno);
 int ret;
 fillbuf(0);
 fillbuf(1);
 unOSRegisterISR(FIQ_TMA_VEC,int_func);
 unOSEnINTSrc(FIQ_TMA_EN);
 while(1) {
        unOSSemPend(sem,0);
        ret = fillbuf(((unsigned int)(playbuf-1)%BUFSIZE));
        if(ret) {
              unOSTimeDly(1);
              unOSDisINTSrc(FIQ_TMA_EN);
              *P_DAC1 = 0x78f0;
              *P_DAC2 = 0x78f0;
              unOSTaskExit();
        }
 }
}
void Task2()
{
 unsigned int i = 1;
      while(1) {
        *P_IOB_DATA = i;
        i<<=1;
        if(i == 0x0100)
              i = 1;
        unOSTimeDly(20);
 }

}
void int_func(void)
{

 *P_DAC1 = (unsigned char)buff[playbuf][playpointer];
 *P_DAC2 = (unsigned char)buff[playbuf][playpointer++];
 if(playpointer == BUFLEN) {
        playbuf = (unsigned int)(playbuf+1)%BUFSIZE;
        playpointer = 0;
        unOSSemPost(sem);
 }
```

```
}
int fillbuf(int buffno)
{
 int i;
 unsigned int tmp;
 __asm__("INT OFF \n\t"                                  //Set data segment
         "SR = SR AND 0x03FF \n\t"
         "SR = SR OR %0"
         :
         :"m"(SegStart)
         );
 unOSEnableINT();                                        //Enable interrupt
 for(i = 0;i <= BUFLEN;i++) {
     __asm__("%0 = D:[%1] \n\t"                          //Read sound data to tmp
             :"=r"(tmp)
             :"r"(pStart)
             );
     pStart++;
     buff[buffno][i] = tmp;
     if((pStart >= pEnd)&&(SegStart == SegEnd))          //Play finished
         return 1;
     if(pStart == (void*)0) {
         SegStart+=0x0400;
         __asm__("INT OFF \n\t"                          //Set data segment
                 "SR = SR AND 0x03FF \n\t"
                 "SR = SR OR %0"
                 :
                 :"m"(SegStart)
                 );
         unOSEnableINT();
     }
 }
 return 0;
}
```

例 11、压缩声音的播放，这个例子用了凌阳声音压缩技术，本例子程序用的是 S480，压缩率为 4.8Kbps。解压后的声音播放速率为 16K。本书并非讨论声音压缩技术方面的专业书，有兴趣的读者可以看相关方面的书。这里要说明一下，S480 for MiniOS library 是以库的形式提供的。在这里只做和 MiniOS 相关的部分介绍，其它从略。

1. 库中用到了一个信号量，这个信号量用来使解码任务和中断播放函数同步。每当中断播放完一个缓冲区时，就会释放这个信号量，让解码程序去把已经播放完的缓冲区再次填

充新的内容。关于如何实现，用户查看前一个例子就会明白。所以在解码任务中建立了这个信号量给库中函数使用。

2. 键盘扫描任务中使用了一个邮箱，这个邮箱用来保存扫描到的键值，解码任务取这个邮件来判断键值并决定操作。注意解码任务中没有使用 unOSSemPend，而是用 unOSSemAccept，这是为什么呢？因为 unOSSemPend 函数会把解码程序转为等待态，直到有键按下才将其转为就绪态；但如果解码不运行的话，声音播放将是错误的，即没有声音可以播放。而 unOSSemAccept 不会使任务等待，即无按键时返回空指针（void* ）0，如果有键按下，将返回邮件的地址。

3. 解码程序永远处于就绪态，这里就绪态是不严格的，为什么呢？其实这个任务也会转为等待态，这是因为声音解码部分会等待一个信号量（在 S480 库中）。但从用户的角度去看，这个程序是永远就绪的。S480 库作为一种驱动程序的形式存在，在实际应用中，驱动程序以库的形式提供给用户，只要将需要的信号量建立在库中即可，用户并不需要知道它是否存在。其它的通信方式也是一样，放入库中后用户不需要知道如何实现通讯，以及以何种方式通信，只要清楚它的接口部分就可以直接使用了。本例中，信号量是在用户的任务之中建立的，这是作者故意安排的，目的是让读者知道它的存在！至于为什么有这个信号量及其他相关部分，请读者再复习一下上一个例子即可明白。

4. 任务 2 还是移灯任务。

这个程序需要外部的硬件支持：

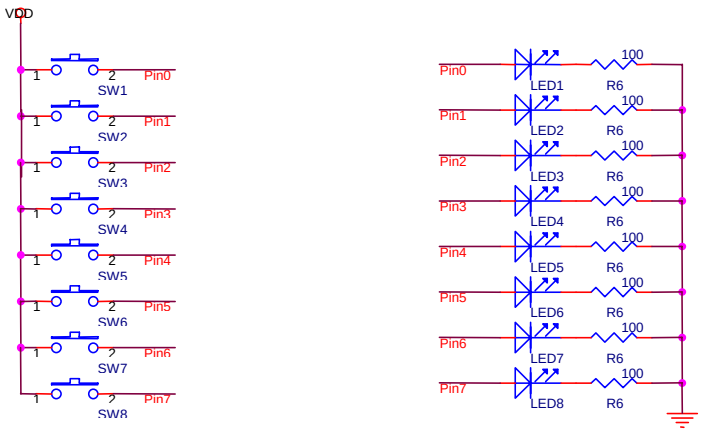
硬件：ICE，带灯的键盘一个

软件：MiniOS 1.5.X 和 S480 for MiniOS library

键盘如下制作：

键盘接到 A 口的低 8 位上

灯接到 B 口的低 8 位上。




```
//
// example 11: S480 on MiniOS
// description: The program uses a semaphore to synchronize decoding task and interrupt playing task
//                    keyboard scan uses a mailbox to save key value.
// author: Taiyun Wang
```

```
// date: 2003/2/22
////////////////////////////////////////////////////////////////////////
#define CREATEVAR
#include "spos.h"

#include "s480.h"

#define MaxSpeechNum                3
#define MaxVolume                   15

void CDecoderTask();
void shiftLED();
void ScanKeyTask();

int t1stack[20];                    //Task 1 stack
int t2stack[20];                    //Task 2 stack
int t3stack[100];                   //task 3 stack
OS_EVENT hDecoderSem;               //Event handle
OS_EVENT hKeyMbox;                  //Event handle

volatile unsigned int *P_IOA_DATA=(unsigned int*)(0x7000);
volatile unsigned int *P_IOB_DATA =(unsigned int*)(0x7005);
volatile unsigned int *P_IOB_DIR =(unsigned int*)(0x7007);
volatile unsigned int *P_IOB_ATTRIB = (unsigned int*)(0x7008);
//主程序
int main()
{
 unOSInit();
 hKeyMbox = unOSMboxCreate(0);                  //Create mialbox
 unOSTaskCreate(CDecoderTask,0,t3stack+99,3);   //Create task of decoder
 unOSTaskCreate(shiftLED,0,t2stack+19,2);       //Create task of LED
 unOSTaskCreate(ScanKeyTask,0,t1stack+19,1);        //Create task of scan keyboard
 unOSStart();                                   //Start OS kernel
}
void ScanKeyTask( )
{
 unsigned int bDown=0;
 unsigned int bkeepDown=0;
 unsigned int iTemp;
 while(1)
 {
```

```
        iTemp = *P_IOA_DATA;
        iTemp &=0x00FF;
        if(iTemp == 0) {
            bDown = 0;
            bkeepDown = 0;
        }
        else {
            if(bDown == 0)
                bDown = 1;
            else if (bkeepDown == 0) {
                unOSMboxPost(hKeyMbox,&iTemp);          //Send scan keyboard value
                bkeepDown = 1;
            }
        }
        unOSTimeDly(2);
    }
}
void shiftLED( )
{
 unsigned int i = 1;
 while(1)
 {
        *P_IOB_DIR = 0XFFFF;
        *P_IOB_ATTRIB = 0XFFFF;
        *P_IOB_DATA = i;
        i<<=1;
        if(i == 0x0100)
            i = 1;
        unOSTimeDly(20);
 }
}
void CDecoderTask()
{
 int Ret = 0;
 long Addr;
 int Mode;

 void * pKey;

 int Key = 0;
 int SpeechIndex = 0;
```

```
int VolumeIndex = 8;
unsigned int i=0;

Mode = Auto;
    unOSRegisterISR(FIQ_TMA_VEC,F_FIQ_Service_SACM_S480);          //Set interrupt function
hDecoderSem = unOSSemCreate(1);                                    //Create semaphore
while(Mode == Auto){
        Ret = unOStem_Initial();
        Ret = SACM_S480_Initial(Auto);
        SACM_S480_Play(SpeechIndex,DAC1+DAC2, Ramp_UpDn_On);
        while(1) {
          pKey=unOSMboxAccept(hKeyMbox);                           //Read key
          if(pKey != (void*)0)
              Key = *(unsigned int*)pKey;
          else
              Key = 0;
            switch(Key) {
            case 0x01:
              SACM_S480_Play(SpeechIndex,DAC1+DAC2, Ramp_UpDn_On);
              break;
            case 0x02:
              SACM_S480_Stop();
              break;
            case 0x04:
              SACM_S480_Pause();
              break;
            case 0x08:
              SACM_S480_Resume();
              break;
            case 0x10:
              VolumeIndex++;
              if(VolumeIndex > MaxVolume)
                  VolumeIndex = MaxVolume;
              SACM_S480_Volume(VolumeIndex);
              break;
            case 0x20:
              if(VolumeIndex == 0)
                  VolumeIndex = 0;
              else
                  VolumeIndex--;
              SACM_S480_Volume(VolumeIndex);
```

```
                break;
            case 0x40:
                if( ++SpeechIndex == MaxSpeechNum)
                    SpeechIndex = 0;
                SACM_S480_Play(SpeechIndex,DAC1+DAC2, Ramp_UpDn_On);
                break;

            case 0x80:
                if( --SpeechIndex < 0)
                SpeechIndex = MaxSpeechNum-1;
                SACM_S480_Play(SpeechIndex,DAC1+DAC2, Ramp_UpDn_On);
                break;
            default:
                break;
            }
        SACM_S480_ServiceLoop();
    }
 }

}
```

恭喜读者，到这里 MiniOS 的功能已经介绍完了，下一章将进入 MiniOS 的程序调试。

8 Mini OS API 列表

本附录给出 MiniOS 全部的系统调用及参数说明。

8.1 初始化部分

8.1.1 int unOSInit(void);

功能：MiniOS 初始化，将会初始化 MiniOS 系统本身用到的所有数据结构。
入口参数：无
返回值：成功：OS_NO_ERR
失败：
OS_OVER_MAX　　最大的任务数超过 15 个
OS_UNDER_ONE　　配置文件中设定的任务个数为 0
注意：unOSInit( ) 应在 unOSStart( )之前调用。
示例：

```
Void main()
{
    .
    .
    unOSInit();                /* Initialize MiniOS*/
    .
    unOSStart();          /* Start Multi-Tasking */
}
```

8.1.2 int unOSStart(void);

功能：启动操作系统的核心，启动后将会把控制权交给优先级最高的任务，让其运行。
入口参数：无
返 回 值：
正确时永远不返回
失败：OS_NO_TASK　　在启动之前没有一个任务被建立
注意：unOSInit( ) 应在 unOSStart( )之前调用，unOSStart( )只能调用一次
示例：参考 unOSInit( )

8.2 任务部分

8.2.1 INT8U　　unOSTaskCreate(long task, void *pdata, OS_STK *ptos, INT8U prio);

功能：建立一个任务

入口参数：

第一个参数 **function** 为指向任务函数的指针转换而来的长整形整数。

第二个参数 **pParameter** 为指向入口参数的指针

第三个参数 **pStackTop** 定义为任务栈顶的指针（为用户事先定义的任务栈数组的最后一个元素的地址）。任务栈用来保存局部变量、函数参数和返回地址等。

第四个参数 **iPrio** 为任务的优先级。

返回值：成功：OS_NO_ERR

失败：

OS_PRIO_EXIST　　优先级已经存在

OS_PRIO_INVALID　无效的优先级，优先级大于 14

OS_NO_TCB　　没有 TCB 可以使用了

注意：共有 15 个任务

示例：

```
int err;                        /* error number */
int t1stack[20];                    /* Task stack */

Void main()
{
    void Task1();
    unOSInit();                     /* Initialize miniOS*/
    .
    err = unOSTaskCreate(Task1,0,t1stack+19,1);     /* Create a task */
    .
    unOSStart();                /* Start Multi-Tasking */
}
void Task1()
{
    .
}
```

8.2.2　int unOSTaskExit(void);

功能：任务自我删除，这是一个自我删除的功能，这个函数将会置本任务为死亡态，系统不再调度该任务。

入口参数：无

返回值：成功：不会返回

失败：OS_IN_ISR 在中断中不能删除任务自身

注意：不可以在中断函数中调用 unOSTaskExit( )

示例：

```
void TaskN()
{
```

```
    .
    unOSTaskExit();
}
```

8.2.3 int unOSTaskDel(unsigned int iPrio);

功能：删除任务，unOSTaskDel（）和 unOSTaskExit（）的根本区别在于 unOSTaskDel（）可以删除别的任务，而 unOSTaskExit（）只能是自我删除。 如果删除自身且成功的话，将不会返回，更没有返回值。

入口参数：要删除任务的优先级

返回值：

成功：OS_NO_ERR（如果是任务自身删除，将不返回）

失败：

OS_PRIO_INVALID　　无效的优先级，要删除的任务的优先级大于 14

OS_PRIO_NULL　　优先级不存在，对应的任务不存在

注意：不可以在中断中调用本函数

示例：

```
void TaskN()
{
    .
    unOSTaskDel(10);            /* delete task with priority 10 */
    .
}
```

8.2.4 int unOSTaskSuspend(int iPrio);

功　　能：挂起一个任务，只能用 unOSTaskResume()恢复。如果当前任务被挂起，MiniOS 内核将重新调度，下一个优先级最高的任务将被执行。

入口参数：要挂起任务的优先级

返 回 值：成功：OS_NO_ERR

失败：

OS_PRIO_INVALID　　无效的优先级，优先级大于 14

OS_PRIO_NULL　　优先级不存在，对应的任务不存在

注　　意：挂起的任务只能用 unOSTaskResume()恢复

示　　例：

```
void TaskN()
{
    .
    while(1)
        {
            .
```

```
            err = unOSTaskSuspend(10);        /* suspend task with priority 10 */
            .
        }
}
```

8.2.5 int unOSTaskResume(int iPrio);

功能：恢复由 unOSTaskSuspend 函数挂起的任务，
入口参数：要恢复任务的优先级
返回值：成功：OS_NO_ERR
失败：

OS_PRIO_INVALID	无效的优先级，优先级大于 14
OS_PRIO_NULL	优先级不存在，对应的任务不存在
OS_RESUME_SELF	恢复自己
OS_NO_SUSPEND	要恢复的任务未被挂起

注意：无
示例：

```
void TaskN()
{
  .
  while(1)
    {
      .
      err = unOSTaskResume(10);              /*resume task with priority 10 */
      .
    }
}
```

8.2.6 int unOSTaskChangePrio(int oldprio, int newprio);

修改任务的优先级
参数：

oldprio	任务原先的优先级
newpiro	任务的新优先级

返回值：
unOSTaskChangePrio()的返回值为下述之一：

OS_NO_ERR	任务优先级被成功改变
OS_PRIO_INVALID	参数中任务原先的优先级或新优先级的值>=14
OS_PRIO_EXIST	参数中任务的新优先级已经存在
OS_PRIO_ERR	参数中任务原先的优先级不存在。

注意/警告:

参数中的新优先级必须是没有使用过的，否则会返回错误码。在 unOSTaskChangePrio()中，还会先判断要改变的优先级的任务是否存在。

范例：

```
void Task(void *data)
{
    int err;

    for(;;){
    …
    err = unOSTaskChangePrio(7,8);
    …
    }
}
```

8.2.7 void unOSTimeDly(unsigned int iTimeCount);

功能：任务自身要求延时一段时间再运行，这个函数会将自身挂起，等待时间到后再进入就绪态，由调度函数调度执行。注意，中断函数不可以调用它。入口参数为 0 的话，将不延时。

入口参数：延时的 time tick 数。

返回值：无

注意：不能在中断函数中调用。

延时时间取决于选取的时钟，在 MiniOS 中，1 tick = 1/256s，约为 7.8ms。

示例：

```
void TaskN()
{
    .
    while(1)
        {
            .
            unOSTimeDly(10);            /* delay task for 10 ticks */
            .
        }
}
```

8.2.8 int unOSTimeDlyResume(int iPrio);

功能：恢复由 unOSTimeDly 延时但时间还没到的任务，会将该任务设置为就绪态。中断函数可以调用。

入口参数：要恢复任务的优先级

返回值：成功：OS_NO_ERR
失败：
OS_PRIO_INVALID　无效的优先级，优先级大于 14
OS_PRIO_NULL　　优先级不存在，对应的任务不存在
OS_NO_TIMEDLY　　任务没处于延时状态
注意：不能用 unOSTimeDlyResume () 去唤醒一个超时等待事件的任务，否则该任务会转为就绪态并返回超时错。
示例：

```
void TaskN()
{
    int err;
    .
    while(1)
        {
            .
            err = unOSTimeDlyResume(10);        /* resume task with priority */
            if (err == OS_NO_ERR)
            .
        }
}
```

8.2.9　unsigned long unOSTimeGet();

unOSTimeGet ()获取当前系统时钟数值。系统时钟是一个 32 位的计数器，记录系统上电后或时钟重新设置后的时钟数值。
参数：
无

返回值：
当前时钟数值（时钟节拍数）

注意/警告:
无
范例：

```
void Task(void *data)
{

    unsigned long clk;
    for(;;){
    …
    clk = unOSTimeGet ();        //获取当前系统时钟的值
```

```
        …
        }
    }
```

8.2.10 void unOSTimeSet(unsigned long ticks);

unOSTimeSet ()设置当前系统时钟数值。系统时钟是一个 32 位的计数器，记录系统上电后或时钟重新设置后的时钟值

参数：

ticks 要设置的时钟数，单位是时钟节拍数。

返回值：

无

注意/警告:

无

范例：

```
void Task(void *data)
{


    for(;;){
    …
    unOSTimeSet (0L);       //复位系统时钟
    …
    }
}
```

8.2.11 int unOSTimeDlyHMSM(int hours,int minutes,int seconds,int milli);

unOSTimeDlyHMSM()将一个任务延时若干时间，延时的单位是[小]时、分、秒及毫秒，所以使用 unOSTimeDlyHMSM()比使用 OSTimedly()更方便。调用 unOSTimeDlyHMSM()后，如果延时时间不为 0,系统将立即进行任务调度。

参数：

hours	延时[小]时数
minutes	延时分[钟]数，范围 0-59
seconds	延时秒数，范围 0-59
milli	延时毫秒数，范围 0-999

返回值：

unOSTimeDlyHMSM ()的返回值为下述之一：

OS_NO_ERR	函数调用成功
OS_TIME_INVALID_MINIUTES	参数错误，分[钟]数大于 59

OS_TIME_INVALID_SECONDS	参数错误，秒数大于 59
OS_TIME_INVALID_MILLI	参数错误，毫秒数大于 999
OS_TIME_ZERO_DLY	4 个参数全为 0

注意/警告:

延时操作函数都是以时钟节拍为单位。实际的延时时间是时钟节拍的整数倍。例如系统每个时钟节拍间隔是 10ms,如果设定的延时为 5ms,将不产生任何延时操作；而设定延时 15ms,实际的延时是 2 个时钟节拍，也就是 20ms.

范例：

```
void Task(void *data)
{


    for(;;){
    …
    unOSTimeDlyHMSM (0,0,1,0);          //任务延时 1s
    …
    }
}
```

8.2.12 void unOSSchedLock(void);

功能：任务的调度锁，上锁之后调度函数将不再调度任务，由 unOSSchedUnLock（）解锁，锁是一个计数值。每次调用本函数加 1，而调用解锁函数将减 1，为 0 时可以调度。

入口参数：无

返回值：无

注意：调用 unOSSchedLock（）之后，就不能再调用 unOSTimeDly（）和 unOSSemPend（）等 API 来挂起当前任务了，因为调度已被锁住，其它任务无法运行，系统将会自锁。

示例：

```
Void TaskN(void *parm)
{
    .
    unOSSchedLock();        /* Prevent other tasks to run */
    .
    .                       /* code protected from content switch */
    unOSSchedUnLock();      /* Enable other tasks to run */
    .
}
```

8.2.13 void unOSSchedUnLock(void);

功能：解任务调度锁。解锁后调度函数可以调度任务的执行，但注意：任务锁是一个计数值，可以多次加锁，解锁也要多次。直到锁值为 0 时才可以调度。

入口参数：无
返回值：无
注意：
示例：参见 unOSSchedLock（）

8.3 事件部分

8.3.1 OS_EVENT *unOSSemCreate(INT16U cnt);OS_EVENT

功能：建立一个信号量，并初始化这个信号量。
入口参数：信号量的初始值。
返回值：成功：指向 ECB 的指针
失败：NULL（无 ECB 可用）
注意：信号量必须先建立，再使用。
示例：

```
OS_EVENT sem;                    //Event handle
main()
{
    .
    unOSInit();
    .
    sem = unOSSemCreate(0);      //Create semaphore
    .
    unOSStart();                 //Start kernel
}
```

8.3.2 unOSSemPend(OS_EVENT *pevent, INT16U timeout, INT8U *err);

OS_EVENTOS_EVENT 功能：等待一个信号量，如果信号量是 0 则会将任务挂起，直到占有信号量的任务释放信号量，或者等待超时。如果在未超时的情况下，信号量被释放，等待这个信号量的优先级最高的任务将恢复执行。被 unOSTaskSuspend()挂起的任务也可以得到信号量，但是任务将保持挂起状态，只有 unOSTaskResume()能恢复任务的执行。

入口参数：
OS_EVENT：事件句柄，它是建立信号量时的返回值。
iTimeout：等待的 TICK 数。
返回值：成功：OS_NO_ERR
失败：

OS_PEVENT_NULL	事件句柄为空
OS_ECB_TYYPE_ERR	事件类型错
OS_IN_ISR	中断中调用
OS_TIME_OUT	超时错

注意：信号量必须先建立，再使用。

示例：

```
OS_EVENT sem1;                  //Event handle
void TaskN()
{
    .
    while (1) {
        unOSSemPend (sem1, 0);      //Waiting semaphore
        .
    }
}
```

8.3.3 INT8U unOSSemPost(OS_EVENT *pevent);OS_EVENTOS_EVENT

功能：发送（释放）指定的信号量。如果信号量值大于 0，而且没有任务在等这个信号量，信号量值将加 1，否则，信号量值保持不变。如果有多个任务在等这个信号量，unOSSemPost（）会把其中优先级最高的任务从等待列表中删除，变成就绪态，内核将发生调度。

入口参数：事件句柄，它是建立信号量时的返回值。

返回值：成功：OS_NO_ERR

失败：

OS_ECB_TYPE_ERR 事件类型错

OS_PEVENT_NULL 事件句柄为空

注意：信号量必须先建立，再使用。

示例：

```
OS_EVENT sem;                   /* Event handle */
void TaskN()
{
    .
    while(1) {
        .
        unOSSemPost(sem);           /* Realease semaphore */
        .
      }
}
```

8.3.4 INT16U unOSSemAccept(OS_EVENT *pevent);OS_EVENTOS_EVENT

功能：读指定信号量的值，这个函数和 unOSSemPend（）函数的最大区别是不会将任务挂起，适合中断中使用。

入口参数：事件句柄，是建立信号量时的返回值。

返回值：

成功：

如果读到的信号量值大于 0，信号量值会减 1，原来的信号量值会返回。

0：没有得到信号量

注意：信号量必须先建立，再使用。

示例：

```
OS_EVENT sem;                                   //Event handle
void TaskN()
{
    int value;
    while(1)
        {
            value = unOSSemAccept(sem);          //reading semaphore
            if (value>0)
                {
                    .
                }
            .
        }
}
```

8.3.5 INT8U unOSSemQuery(OS_EVENT *pevent, OS_SEM_DATA *pdata);OS_EVENTOS_EVENT

unOSSemQuery ()函数用于获取某个信号量的信息。在使用 unOSSemQuery()之前，应用程序要先建立类型为 ECB_struct 的数据结构，用来保存从信号量的事件控制块中取得的数据。使用 unOSSemQuery()可以得知，是否有以及有多少个任务位于信号量的任务等待队列，还可以获取信号量的值。

参数：

peventOS_EVENT 一个指向信号量的句柄。该句柄在信号量建立之后返回。

pdata 一个指向数据结构 ECB_struct 的指针。

返回值：

unOSSemQuery ()的返回值为下述之一：

OS_NO_ERR	函数调用成功
OS_ECB_TYPE_ERR	OS_EVENT 不指向信号量
OS_PEVENT_NULL	OS_EVENT 是空指针

注意/警告:

无

范例：

```
OS_EVENT eventsem;
void Task(void *data)
{
ECB_struct sem_data;
int err;
int count;

for(;;){
    …
    err = unOSSemQuery(eventsem,&sem_data);
    if(err == OS_NO_ERR){
        count = sem_data.count; //获取信号量的值
    }
    …
    }
}
```

8.3.6 OS_EVENT *unOSSemDel(OS_EVENT *pevent, INT8U opt, INT8U *err);OS_EVENTOS_EVENTOS_EVENT

unOSSemDel ()函数用于删除信号量。使用本函数有风险，因为多任务中的其他任务可能还想使用这个信号量。使用本函数须特别小心。一般地说，在删除信号量之前，应该先删除所有可能用到这个信号量的任务。

参数：

peventOS_EVENT 一个指向信号量的句柄。该句柄在信号量建立之后返回。

opt 该选项定义信号量的删除条件。可以选择只能在已经没有任何任务在等待该信号量时，才能删除该信号量(OS_DEL_NO_PEND)；或者，不管有没有任务在等待该信号量，立即删除这个信号量(OS_DEL_ALWAYS)，在这种情况下，所有等待该信号量的任务都立即进入就绪态。

err 指向包含错误码的变量的指针。返回的错误码可能为下述几种之一：

OS_NO_ERR	调用成功，信号量已被删除
OS_ERR_DEL_ISR	试图在中断服务程序中删除信号量
OS_ERR_INVALID_OPT	没有将 opt 参数定义为 2 种合法参数之一。
OS_ERR_TASK_WAITING	有一个或一个以上的任务在等待信号量。
OS_ERR_EVENT_TYPE	OS_EVENT 不是指向信号量的句柄。
OS_ERR_PEVENT_NULL	hveent 是空句柄。

返回值：

如果信号量已被删除了，则返回空指针；若信号量没能被删除，则返回 OS_EVENT.后一种情况下，应该查看出错代码，以查明原因。

注意/警告:

1. 使用本函数调用须特别小心，因为其他任务可能还会用到这个信号量
2. 当挂企的任务进入就绪态时，中断是关闭的，这就意味着中断延迟时间与等待信号量的任务数目有关。

范例：

```
OS_EVENT OS_EVENT;
void Task(void *data)
{
for(;;){
    …
    OS_EVENT = unOSSemDel(OS_EVENT,OS_DEL_ALWAYS,&err);
    if(OS_EVENT == 0){
        //信号量已被删除.
    }
    …
    }
}
```

8.3.7 OS_EVENT unOSMboxCreate(void* pMail);

功能：建立并初始化一个邮箱，邮箱允许任务或中断服务程序给一个或多个任务发送一个消息指针(邮件)。

入口参数：pMail 用来初始化邮箱的内容，如果为空指针，邮箱内容为空，如果不是空指针，邮箱里初始化为有一条消息。

返回值：成功：指向 ECB 的指针

失败：NULL 无事件控制块可用

注意：邮箱必须先建立再使用

示例：

```
OS_EVENT mailbox;                          //Event handle
main()
{
    .
    unOSInit();
    .
    mailbox = unOSMboxCreate((void*)0);          //Create mailbox
    unOSStart();                           //Start OS kernel
}
```

8.3.8 void* unOSMboxPend(OS_EVENT hEvent, unsigned int iTimeOut, int* err);

功能：用于任务等待消息时。消息由中断服务程序或另一个任务发送，消息是一个指针型变量。

如果调用 unOSMboxPend（）时，邮箱里有消息，邮箱会被清空，消息返回给调用者；如果邮箱里没有消息，unOSMboxPend（）会将当前任务挂起，直到收到消息或超时。如果邮箱收到一个消息，而有多个任务在等这个消息，最高优先级的任务会恢复执行。被 unOSTaskSuspend（）挂起的任务也可以收消息，但仍然为挂起状态，直到被 unOSTaskResume（）恢复。

入口参数：

hEvent：邮箱的句柄，建立邮箱时的返回值

iTimeout：等待的 tick 数，如果超过时间，err 将被置为 OS_TIME_OUT，任务被置成就绪态。如果此值为 0，意味着任务将永远等待消息。

err：是一个保存错误码的指针变量。

Err：

OS_NO_ERR　　无错

OS_PEVENT_NULL　　事件句柄为空

OS_ECB_TYPE_ERR　　ECB 类型错

OS_IN_ISR　　在中断服务程序中调用

OS_TIME_OUT　　等待超时

返回值：

成功：返回指向邮件的指针，并置 err 为 OS_NO_ERR

失败：返回 NULL 指针，并置 err

注意：邮箱必须先建立再使用；不能在中断服务程序中调用 unOSTaskSuspend（）。

示例：

```
OS_EVENT mailbox;                                   /* Event handle */
void TaskN()
{
    int * pmsg;
    int err;
    while(1)
        {
            pmsg = unOSMboxPend(mailbox, 0, &err);  /* Waiting mail */
            if (err == OS_NO_ERR)
                {
                    .
                }
            .
        }
}
```

8.3.9　int unOSMboxPost(OS_EVENT hEvent, void* pMail);

功能：将邮件送到指定的邮箱中。如果邮箱中已经有邮件，会返回错误码 OS_MBOX_FULL，邮件不能放到邮箱中。如果多个任务在等一个邮件，优先级最高的任务会得到邮件。如果等邮件的任务的优先级高于发邮件的任务，高优先级的任务会被执行，发邮件的任务会被转入非运行的就绪

态。

入口参数：

hEvent：邮箱的句柄，建立邮箱时的返回值

pMail：是指向要发邮件的指针

返回值：成功：OS_NO_ERR

失败：

OS_PEVENT_NULL　　事件句柄为空

OS_ECB_TYPE_ERR　　事件类型错

OS_MBOX_FULL　　邮箱中有邮件

注意：邮箱必须先建立再使用；邮件指针 pMail 不能为空，因为空指针意味着邮箱为空。

示例：

```
OS_EVENT mailbox;                    /* Event handle */
void TaskN()
{
    int err, msg = 0;
    while(1)
        {
            err = unOSMboxPost(mailbox, &msg); /* Waiting mail */
            .
        }
}
```

8.3.10 int unOSMboxPostOpt(OS_EVENT　OS_EVENT,void* msg,int opt);

unOSMboxPostOpt()的工作方式与 unOSMboxPost()相同，只是允许用户程序发消息给多个任务。换句话说，unOSMboxPostOpt()允许将消息广播给所有的等待邮箱消息的任务。

unOSMboxPostOpt()通过邮箱给任务发消息。消息是一个以指针表示的某种数据类型的变量，在不同的程序中消息的使用也可能不同。如果消息邮箱中已存在消息，则返回错误码，说明消息邮箱已满。unOSMboxPostOpt()函数立即返回给调用者，消息也没有能够发到消息邮箱。如果有任何任务在等待消息邮箱的消息，那么 unOSMboxPostOpt()允许用户选择以下 2 种情况之一：或者让最高优先级的任务得到这则消息(opt=OS_POST_OPT_NONE)，或者让所有等待邮箱消息的任务都得到消息(opt=OS_POST_OPT_BROADCAST)。无论在哪种情况下，如果得到消息的任务优先级比发送消息的任务优先级高，那么得到消息的最高优先级的任务将恢复执行，发消息的任务将被挂起。也就是说，发生了一次任务切换。

参数：

OS_EVENT　消息邮箱的句柄。在创建消息邮箱时返回。

msg　　即将实际发送给任务的消息。消息是一个以指针表示的某种数据类型的变量，在不同的程序中消息的使用也可能不同。不允许传递一个空指针，因为这意味着消息邮箱为空。

opt　　定义消息只发给等待邮箱消息的任务中优先级最高的任务 (OS_POST_OPT_NONE)

或让所有等待邮箱消息的任务都得到消息
(OS_POST_OPT_BROADCAST)

返回值：

unOSMboxPostOpt ()的返回值为下述之一：

OS_NO_ERR	调用成功，消息已经发出。
OS_MBOX_FULL	邮箱中已有消息，每次只能发送一则消息到消息邮箱
OS_ERR_EVENT_TYPE	hevent 不是消息邮箱的类型句柄
OS_ERR_PEVENT_NULL	hevent 是空句柄

注意/警告:

unOSMboxPostOpt ()在广播方式下，即将 opt 置为 OS_POST_OPT_BROADCAST，函数的执行时间取决于等待邮箱消息的任务数目。

范例：

```
OS_EVENT    hevent;
Char commrxbuf[100];
void Task(void *data)
{
    int err;

    for(;;){
    …
    err = unOSMboxPostOpt(hevent,(void*)&commrxbuf[0],OS_POST_OPT_BROADCAST);
    …
    }
}
```

8.3.11 void* unOSMboxAccept(OS_EVENT hEvent);

功能：从指定的邮箱读邮件，本函数和 unOSMboxPend（）最大的区别是 unOSMboxPend ()函数将会把任务挂起，等待邮件的到来，本函数并不等待邮件的到来，如无邮件，则返回 NULL。

入口参数：邮箱的句柄，建立邮箱时的返回值

返回值：成功：返回指向邮件的指针

失败：NULL：无邮件

注意：邮箱必须先建立再使用

示例：

```
OS_EVENT mailbox;                    /* Event handle */
void TaskN()
{
    void *msg;
    while(1)
```

```
        {
            msg = unOSMboxAccept (mailbox);     /* Waiting mail */
            if (msg != (void *)0)
                {
                    .
                }
            else
                {
                    .
                }
            }
}
```

8.3.12 Viud unOSMboxQuery(OS_EVENT hEvent, ECB_struct *pdata) ;

unOSMboxQuery ()函数用于获取消息邮箱的信息。在使用 unOSMboxQuery ()之前，应用程序要先建立类型为 ECB_struct 的数据结构，用来保存从消息邮箱的事件控制块中取得的数据。使用 unOSMboxQuery ()可以得知，是否有以及有多少个任务位于消息邮箱的任务等待队列，还可以获取消息邮箱现在的消息。

参数：

hEvent 一个消息邮箱的句柄。该句柄在消息邮箱建立之后返回。

pdata 一个指向数据结构 ECB_struct 的指针。

返回值：

unOSMboxQuery ()的返回值为下述之一：

OS_NO_ERR	函数调用成功
OS_ERR_EVENT_TYPE	hEvent 不指向信号量
OS_ERR_EVENT_NULL	hEvent 是空指针

注意/警告:

无

范例：

```
OS_EVENT eventsem;
void Task(void *data)
{
ECB_struct mbox_data;
int err;

for(;;){
    …
    err = unOSSemQuery(eventsem,&mobx_data);
    if(err == OS_NO_ERR){
    //如果 mbox_data. msg 为非空指针，说明消息邮箱非空
```

```
        }
        …
        }
    }
```

8.3.13 void unOSMobxDel(OS_EVENT hEvent,int opt,int *err);

unOSSemDel ()函数用于删除消息邮箱。使用本函数有风险，因为多任务中的其他任务可能还想使用这个消息邮箱。使用本函数须特别小心。一般地说，在删除消息邮箱之前，应该先删除所有可能用到这个消息邮箱的任务。

参数：

hEvent 一个指向消息邮箱的句柄。该句柄在消息邮箱建立之后返回。

opt 该选项定义消息邮箱的删除条件。可以选择只能在已经没有任何任务在等待该消息邮箱时，才能删除该消息邮箱(OS_DEL_NO_PEND)；或者，不管有没有任务在等待该消息邮箱，立即删除这个信号量(OS_DEL_ALWAYS)，在这种情况下，所有等待该消息邮箱的任务都立即进入就绪态。

err 指向包含错误码的变量的指针。返回的错误码可能为下述几种之一：

OS_NO_ERR	调用成功，消息邮箱已被删除
OS_ERR_DEL_ISR	试图在中断服务程序中删除消息邮箱
OS_ERR_INVALID_OPT	没有将 opt 参数定义为 2 种合法参数之一。
OS_ERR_TASK_WAITING	有一个或一个以上的任务在等待消息邮箱。
OS_ERR_EVENT_TYPE	hevent 不是指向消息邮箱的句柄。
OS_ERR_PEVENT_NULL	hveent 是空句柄。

返回值：

如果消息邮箱已被删除了，则返回空指针；若消息邮箱没能被删除，则返回 hevent.后一种情况下，应该查看出错代码，以查明原因。

注意/警告:

使用本函数调用须特别小心，因为其他任务可能还会用到这个消息邮箱

当挂企的任务进入就绪态时，中断是关闭的，这就意味着中断延迟时间与等待消息邮箱的任务数目有关。

范例：

```
OS_EVENT hevent;
void Task(void *data)
{
for(;;){
        …
        hevent = unOSMobxDel(hevent,OS_DEL_ALWAYS,&err);
        if(hevent == 0){
            //消息邮箱已被删除.
        }
        …
        }
```

}

8.3.14 OS_EVENT unOSQCreate(void* pStart, unsigned int iSize);

功能：建立一个消息队列。消息队列允许任务或 ISR 给一个或多个任务发多条消息。
入口参数：
pStart：指向一个数组的指针（这个数组要由用户指定）
iSize：队列的长度（也就是那个数组的大小）。
返回值：
成功：返回 ECB 的指针
失败：NULL　　无 ECB 或是 QCB 可用
注意：消息队列要先建立再使用
示例：

```
OS_EVENT queue;                    /* Event handle */
int *Q[5];                         /* Message queue array */
void main(void)
{
    unOSInit();
    .
    queue = unOSQCreate((void*)Q,5);    /* Create Message queue */
    .
    unOSStart();                   /* Start Os kernel */
}
```

8.3.15 void* unOSQPend(OS_EVENT hEvent, unsigned int iTimeOut, int* err);

功能：等待一个队列中的消息。中断服务程序或另一个任务可以给等消息的任务发送消息。如果调用 unOSQPend（）时队列中有至少一条消息，任务得到消息。如果队列中没有消息，unOSQPend（）会将当前任务挂起，直到收到消息或等待超时。如果有多个任务在等一个消息，其中优先级最高的任务会恢复执行。被 unOSTaskSuspend（）挂起的任务也可以收消息，但要等调用 unOSTaskResume（）后，任务才可以恢复执行。
入口参数：
hEvent：消息队列的句柄，是建立队列时的返回值
iTimeout：等待的时间。如果超过时间，err 将被置为 OS_TIME_OUT，任务被置成就绪态。如果此值为 0，意味着任务将永远等待消息。
err：错误代码，具体含义如下：

OS_NO_ERR	无错
OS_TIME_OUT	超时
OS_ECB_TYPE_ERR	类型错
OS_IN_ISR	在中断中调用
OS_PEVENT_NULL	事件句柄为空

返回值：

成功：返回消息，并置 err 为 OS_NO_ERR；
失败：返回空指针，并置 err 为对应的错误码。
注意：消息队列要先建立再使用；
不能在中断服务程序中调用。
示例：

```
int err;                        //Error No
OS_EVENT queue1;                //Event handle

void TaskN()
{
    unsigned int *pmsg;
    while(1) {
        pmsg = unOSQPend (queue1, 0, &err); //waiting queue
        .
    }
}
```

8.3.16 int unOSQPost(OS_EVENT hEvent, void* pMsg);

功能：将消息送到指定事件句柄的消息队列中，如果消息队列满，返回错误码，消息不放到队列中。如果有几个任务在等一个消息，优先级高的任务会得到这个消息。如果等消息的任务的优先级高于发消息的任务，高优先级的任务会恢复执行，发消息的任务被挂起。本函数是将消息放入消息队列的尾部(FIFO)，也就是说先收到的消息先得到。

入口参数：
hEvent：消息队列的句柄，建立队列时的返回值
pMsg：指向发送消息的指针
返回值：成功：OS_NO_ERR
失败：
OS_QUEUE_FULL　　消息队列满
OS_ECB_TYPE_ERR　　事件类型错
OS_PEVENT_NULL　　事件句柄为空
注意：消息队列要先建立再使用
不能发送空指针
示例：

```
OS_EVENT queue;                 //Event handle
int msg[5];

void TaskN()
{
    .
    while(1) {
```

```
            unOSQPost(queue, &msg[0]);          //insert queue
            .
        }
    }
```

8.3.17 int unOSQPostOpt(OS_EVENT hevent,void* msg,int opt);

unOSQPostOpt()的工作方式与 unOSQPost()相同，只是允许用户程序发消息给多个任务。换句话说，unOSQPostOpt()允许将消息广播给所有的等待消息队列的任务。

unOSQPostOpt()通过消息队列给任务发消息。消息是一个以指针表示的某种数据类型的变量，在不同的程序中消息的使用也可能不同。如果消息队列中的消息已满，则返回错误码，说明消息队列已满。unOSQPostOpt()函数立即返回给调用者，消息也没有能够发到消息队列。如果有任何任务在等待消息队列的消息，那么 unOSQPostOpt()允许用户选择以下 2 种情况之一：或者让最高优先级的任务得到这则消息(opt=OS_POST_OPT_NONE)，或者让所有等待消息队列的任务都得到消息(opt=OS_POST_OPT_BROADCAST)。无论在哪种情况下，如果得到消息的任务优先级比发送消息的任务优先级高，那么得到消息的最高优先级的任务将恢复执行，发消息的任务将被挂起。也就是说，发生了一次任务切换。

参数：

hevent	消息队列的句柄。在创建消息队列时返回。
msg	即将实际发送给任务的消息。消息是一个以指针表示的某种数据类型的变量，在不同的程序中消息的使用也可能不同。不允许传递一个空指针，因为这意味着消息队列为空。
opt	定义消息只发给等待消息队列的任务中优先级最高的任务(OS_POST_OPT_NONE),如果没有等待任务，相当 unOSQPost() 或让所有等待消息队列的任务都得到消息(OS_POST_OPT_BROADCAST)

返回值：

unOSQPostOpt ()的返回值为下述之一：

OS_NO_ERR	调用成功，消息已经发出。
OS_QUEUE_FULL	消息队列已满，不能在接收消息
OS_ERR_EVENT_TYPE	hevent 不是消息队列的类型句柄
OS_ERR_PEVENT_NULL	hevent 是空句柄

注意/警告:

unOSQPostOpt ()在广播方式下，即将 opt 置为 OS_POST_OPT_BROADCAST，函数的执行时间取决于等待邮箱队列的任务数目。

范例：

```
OS_EVENT    hevent;
Char commrxbuf[100];
```

```
    void Task(void *data)
    {
        int err;

        for(;;){
        …
        err = unOSQPostOpt(hevent,(void*)&commrxbuf[0],OS_POST_OPT_BROADCAST);
        …
        }
}
```

8.3.18 int unOSQPostFront(OS_EVENT hEvent, void* pMsg);

功能：将消息送到指定事件句柄的消息队列中，与 unOSQPost（）很相似，只是本函数是将消息放入消息队列的头部(LIFO)，其它同 unOSQPost（）。

入口参数：

hEvent：消息队列的句柄，建立队列时的返回值

pMsg：指向发送消息的指针

返回值：

成功：OS_NO_ERR

失败：

OS_QUEUE_FULL　　消息队列满

OS_ECB_TYPE_ERR　　事件类型错

OS_PEVENT_NULL　　事件句柄为空

注意：消息队列要先建立再使用

　　不能发送空指针

示例：

```
OS_EVENT queue;                    //Event handle
int msg[5];

void TaskN()
{
    .
    while(1) {
    unOSQPostFront (queue, &msg[0]);        //insert queue
            .
        }
}
```

8.3.19 void* unOSQAccept(OS_EVENT hEvent);

功能：读队列中的消息，本函数和 unOSQPend（）的区别在于 Pend 等待消息的到来，本函数

并不等待，如队列中无消息，则返回 NULL。

入口参数：消息队列的句柄，建立队列时的返回值

返回值：成功：返回消息的指针

失败：NULL （队列中无消息）

注意：消息队列要先建立再使用

示例：

```
OS_EVENT queue;                          /* Event handle */
void TaskN()
{
    void *msg;
    while(1)
        {
            msg = unOSQAccept (queue);       /* check queue for a message */
            if (msg != (void *)0)
                {
                    .                        /* message received, process */
                }
            else
                {
                    .                        /* message not received */
                }
        }
}
```

8.3.20 Void unOSQQuery(OS_EVENT hEvent, ECB_struct *pdata);

unOSQQuery ()函数用于获取消息邮箱的信息。在使用 unOSQQuery ()之前，应用程序要先建立类型为 ECB_struct 的数据结构，用来保存从消息队列的事件控制块中取得的数据。使用 unOSQQuery ()可以得知，是否有以及有多少个任务位于消息队列的任务等待队列，还可以获取消息队列现在的消息。

参数：

hEvent 一个消息队列的句柄。该句柄在消息队列建立之后返回。

pdata 一个指向数据结构 ECB_struct 的指针。

返回值：

unOSQQuery ()的返回值为下述之一：

OS_NO_ERR	函数调用成功
OS_ERR_EVENT_TYPE	hEvent 不指向消息队列
OS_ERR_EVENT_NULL	hEvent 是空指针

注意/警告:

无

范例：


```
E_handle eventsem;
void Task(void *data)
{
ECB_struct queue_data;
```

8.3.21 void unOSQDel(OS_EVENT hEvent,int opt,int *err);

unOSSemDel ()函数用于删除消息队列。使用本函数有风险，因为多任务中的其他任务可能还想使用这个消息队列。使用本函数须特别小心。一般地说，在删除消息队列之前，应该先删除所有可能用到这个消息队列的任务。

参数：

hEvent 一个指向消息队列的句柄。该句柄在消息队列建立之后返回。

opt 该选项定义消息队列的删除条件。可以选择只能在已经没有任何任务在等待该消息队列时，才能删除该消息队列(OS_DEL_NO_PEND)；或者，不管有没有任务在等待该消息队列，立即删除这个信号量(OS_DEL_ALWAYS)，在这种情况下，所有等待该消息队列的任务都立即进入就绪态。

err 指向包含错误码的变量的指针。返回的错误码可能为下述几种之一：

OS_NO_ERR	调用成功，消息队列已被删除
OS_ERR_DEL_ISR	试图在中断服务程序中删除消息队列
OS_ERR_INVALID_OPT	没有将 opt 参数定义为 2 种合法参数之一。
OS_ERR_TASK_WAITING	有一个或一个以上的任务在等待消息队列。
OS_ERR_EVENT_TYPE	hevent 不是指向消息队列的句柄。
OS_ERR_PEVENT_NULL	hveent 是空句柄。

返回值：

如果消息队列已被删除了，则返回空指针；若消息队列没能被删除，则返回 hevent.后一种情况下，应该查看出错代码，以查明原因。

注意/警告:

1.使用本函数调用须特别小心，因为其他任务可能还会用到这个消息队列

2.当挂起的任务进入就绪态时，中断是关闭的，这就意味着中断延迟时间与等待消息队列的任务数目有关。

范例：

```
OS_EVENT hevent;
void Task(void *data)
{
for(;;){
    …
    hevent = unOSQDel(hevent,OS_DEL_ALWAYS,&err);
    if(hevent == 0){
        //消息队列已被删除.
    }
    …
    }
}
```

8.3.22 int unOSQFlush(OS_EVENT hevent);

OSQFlush()函数清空消息队列，并且忽略发往队列的所有消息。不管队列中是否有消息，这个函数的执行时间都是相同的。

参数：

hEvent　一个指向消息队列的句柄。该句柄在消息队列建立之后返回。

返回值：

OSQFlush ()的返回值为下述之一：

OS_NO_ERR	消息队列被成功清空
OS_ERR_EVENT_TYPE	hEvent 不指向消息队列
OS_ERR_EVENT_NULL	hEvent 是空指针

注意/警告:

无

范例：

```
OS_EVENT hevent;
void Task(void *data)
{

int err;
    …
    err = unOSQFlush (hevent);
    if(err == OS_NO_ERR) {
        //消息队列被成功清空
        …
```

```
        }
        …
        }
    }
```

8.4 中断处理部分

8.4.1 INT8U unOSRegisterISR(INT8U vectornumber,INT32U pISR);

功能：将中断函数 UserIntService 挂到中断 Vector 上。
入口参数：
Vector：OS 中定义的中断号(可查询头文件： 如 SposCE061A.h 或 SPOSL16256A.h)))))))))))；
pISR：用户中断函数的指针所转的长整数；
返回值：
成功： OS_NO_ERR
失败：OS_NO_IFT 无中断函数表
注意：
使用 OS，推荐用户用此函数来挂接中断服务程序，OS 会为用户作特别的处理，以方便用户的使用；
示例：

```
Void TaskN()
{
    void Alarm_Service();

    unOSRegisterISR (INT_RTC_ALARM, Alarm_Service);
    //将函数 Alarm_Service ()挂到 SPL16256a 的 RTC_ALARM 中断中；
}
void Alarm_Service ()
{
    //User can do some alarm program
}
```

8.4.2 INT8U unOSReleaseISR(INT8U vectornumber,INT32U pISR);

功能：将中断函数 UserIntService 从中断 Vector 上摘下。
入口参数：
Vector：OS 中定义的中断号(可查询头文件： 如 SposCE061A.h 或 SPOSL16256A.h)；
pISR：用户中断函数的指针所转的长整数；
返回值：
成功： OS_NO_ERR
失败：OS_NO_IFT 无中断函数表
注意：

示例：

```
Void TaskN()
{
    void Alarm_Service();

    unOSReleaseISR (INT_RTC_ALARM, Alarm_Service);
    //将函数 Alarm_Service ()挂到 SPL16256a 的 RTC_ALARM 中断中；
}
void Alarm_Service ()
{
    //User to do some alarm program
}
```

8.5 其它

8.5.1 int unOSGetPri();

功能：得到当前任务的优先级
入口参数：无
返回值：任务的优先级
注意：无
示例：

```
Void TaskN(void)
{
    int pid;
    for (; ;) {
        .
        pid = unOSGetPri();          /* get priority */
        .
    }
}
```

9