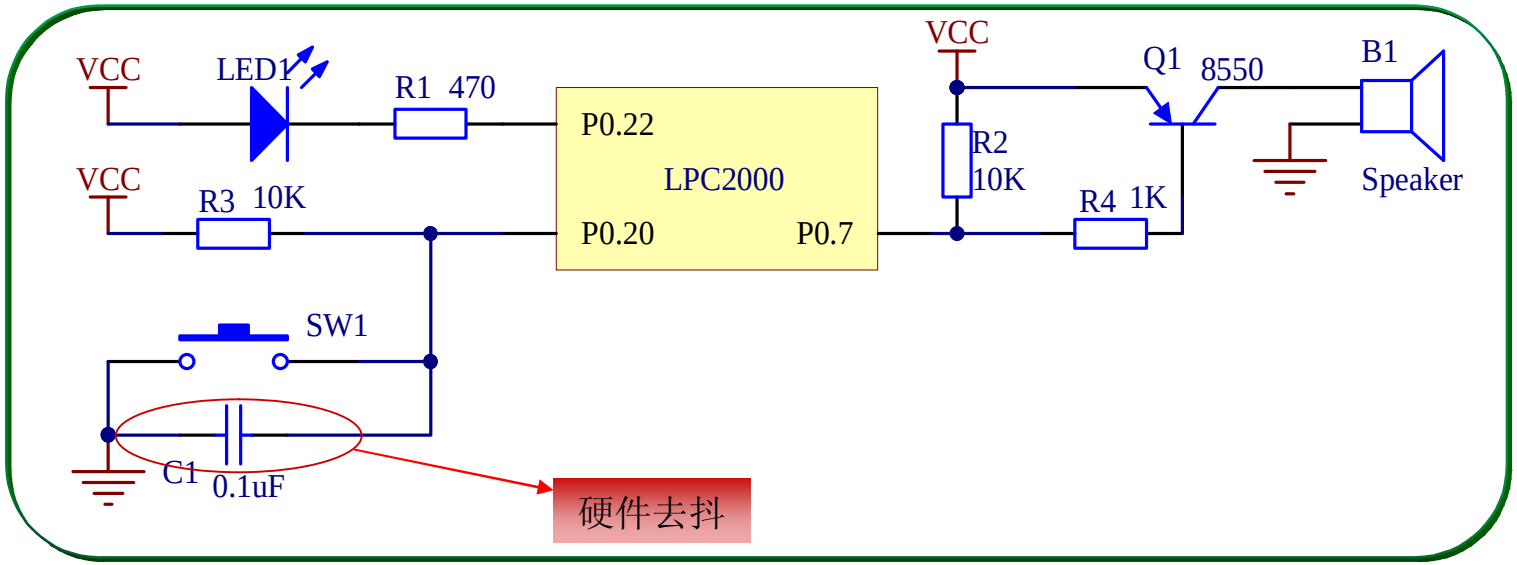
September,2007

μC/OS-II程序设计基础

绪论

μC/OS-II程序设计基础

本章重点介绍μC/OS-II V2.52版本嵌入式实时操作系统常用函数的基本用法，其最大的特点不仅示例程序简洁明了，而且电路也非常简单（如下图），希望初学者一看就懂、一学就会，达到快速入门的目的。



注意：本章所有应用示例都全部默认采用这个图，主要是为了尽量简化示例程序，提高可读性，使用简单明了的语言和方法来解析复杂的理论知识，这是作者在多年的学习、工作和教学中一直倡导的风格和习惯性的行为，同时这也是写作本书的出发点。

目　录

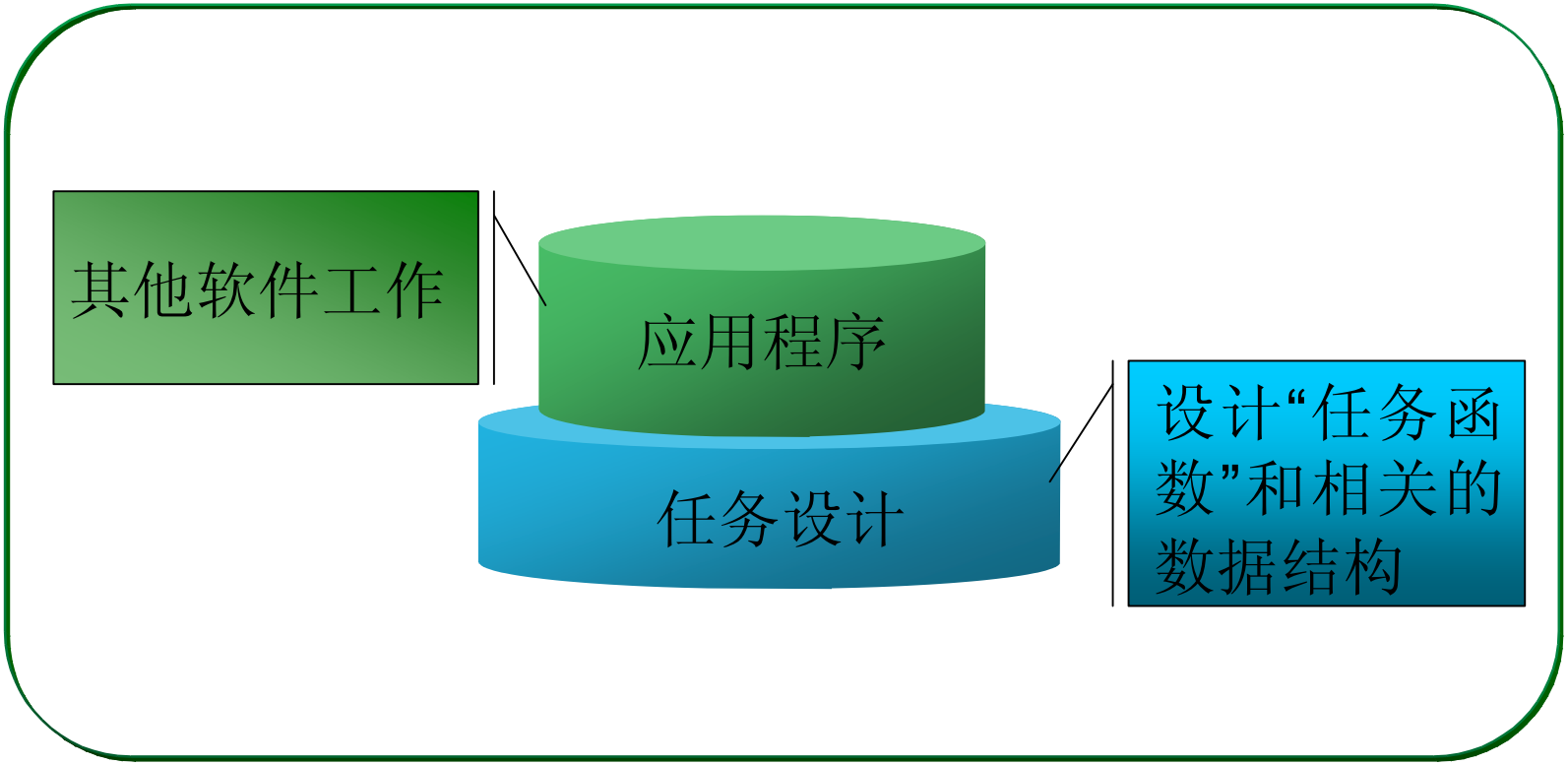
任务设计

μC/OS-II程序设计基础

1. 任务的分类
2. 任务的划分
3. 任务的优先级

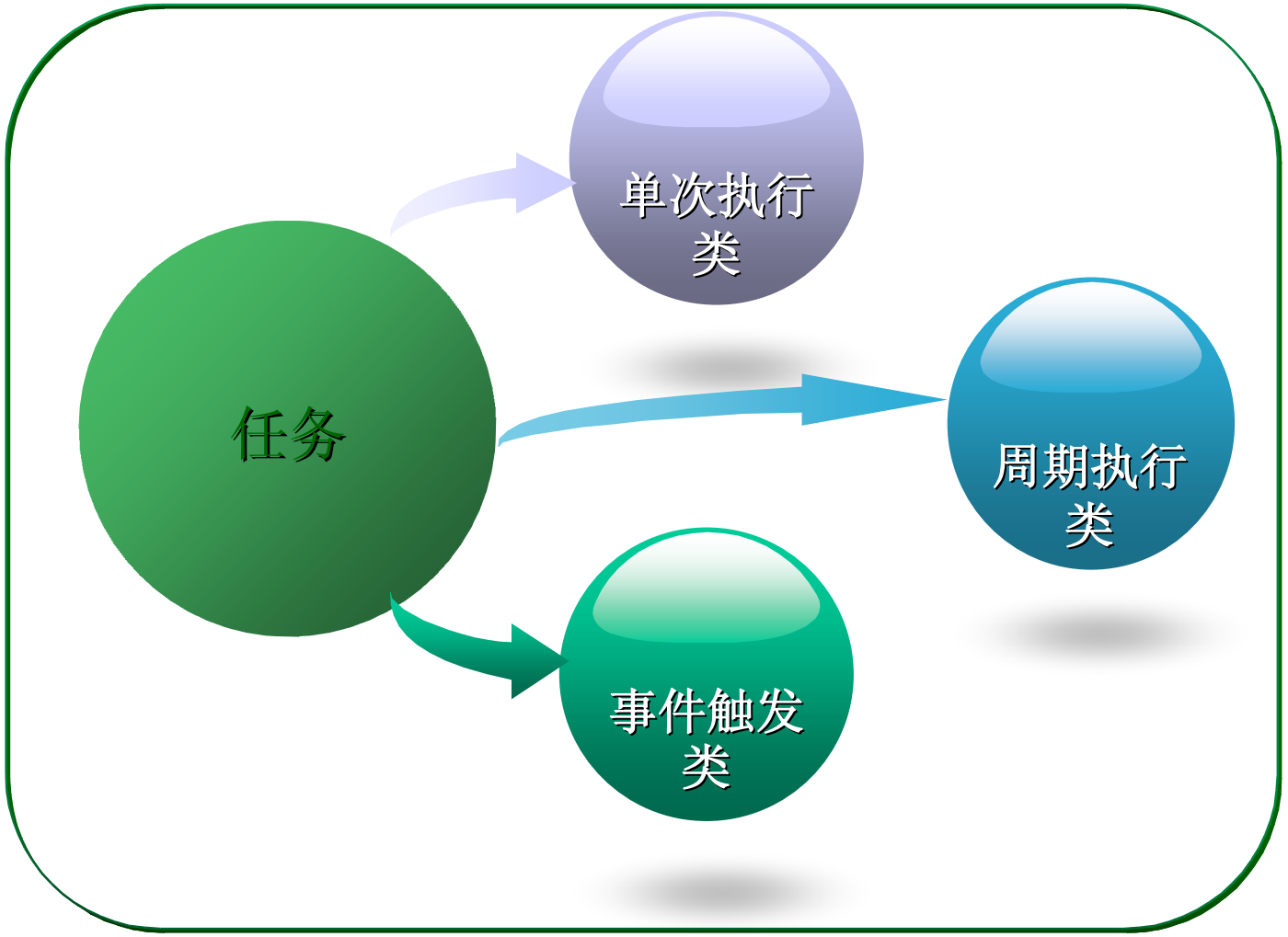
任务设计

在基于实时操作系统的应用程序设计中，任务设计是整个应用程序的基础，其它软件设计工作都是围绕任务设计来展开。



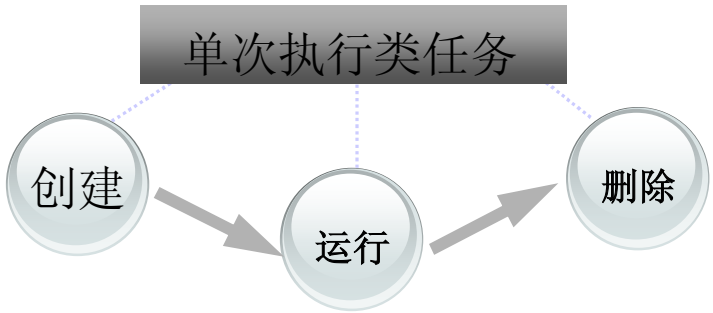
任务的分类

按照执行方式分类



任务的分类

1.单次执行类任务

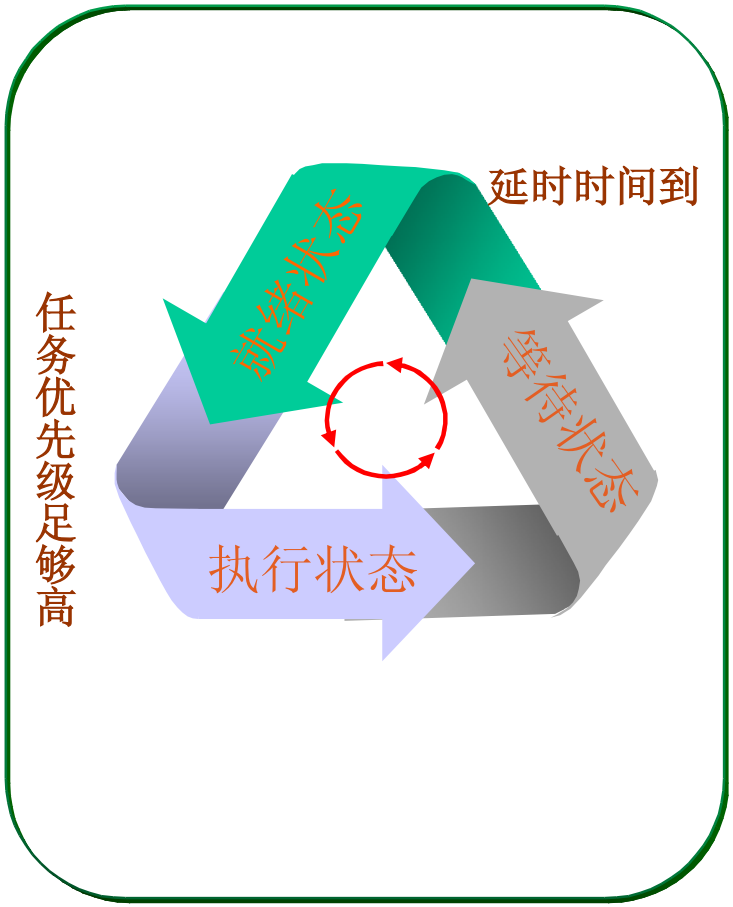



```
void  MyTask (void *pdata)
{
  进行准备工作的代码;
  任务实体代码;
  调用任务删除函数;
}
```

- 定义和初始化变量及硬件设备
- 完成该任务的具体功能
- 将自己删除，操作系统将不再管理它

任务的分类

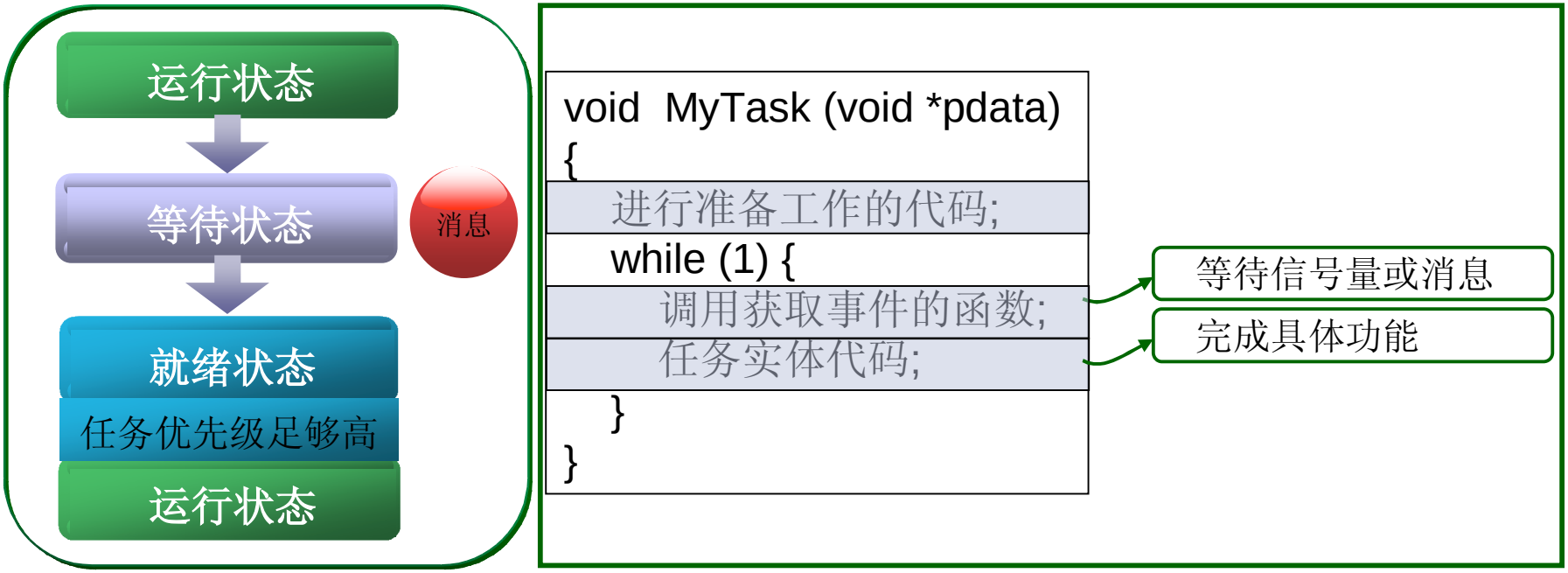
2.周期执行类任务




```
void  MyTask (void *pdata)
{
    进行准备工作的代码;
    while (1) {
        任务实体代码;
        调用系统延时函数;
    }
}
```

任务的分类

3.事件触发执行类任务



任务的划分

1.任务划分的目标

在对一个具体的嵌入式应用系统进行任务划分时，可以有不同的任务划分方案。为了选择最佳划分方案，就必须知道任务划分的目标。

1.首要目标是满足“实时性”指标：即使在最坏的情况下，系统中所有对实时性有要求的功能都能够正常实现；

2.任务数目合理：对于同一个应用系统，合理的合并一些任务，使任务数目适当少一些还是比较有利；

3.简化软件系统：一个任务要实现其功能，除了需要操作系统的调度功能支持外，还需要操作系统的其它服务功能支持，合理划分任务，可以减少对操作系统的服务要求，简化软件系统；

4.降低资源需求：合理划分任务，减少或简化任务之间的同步和通信需求，就可以减少相应数据结构的内存规模，从而降低对系统资源的需求。

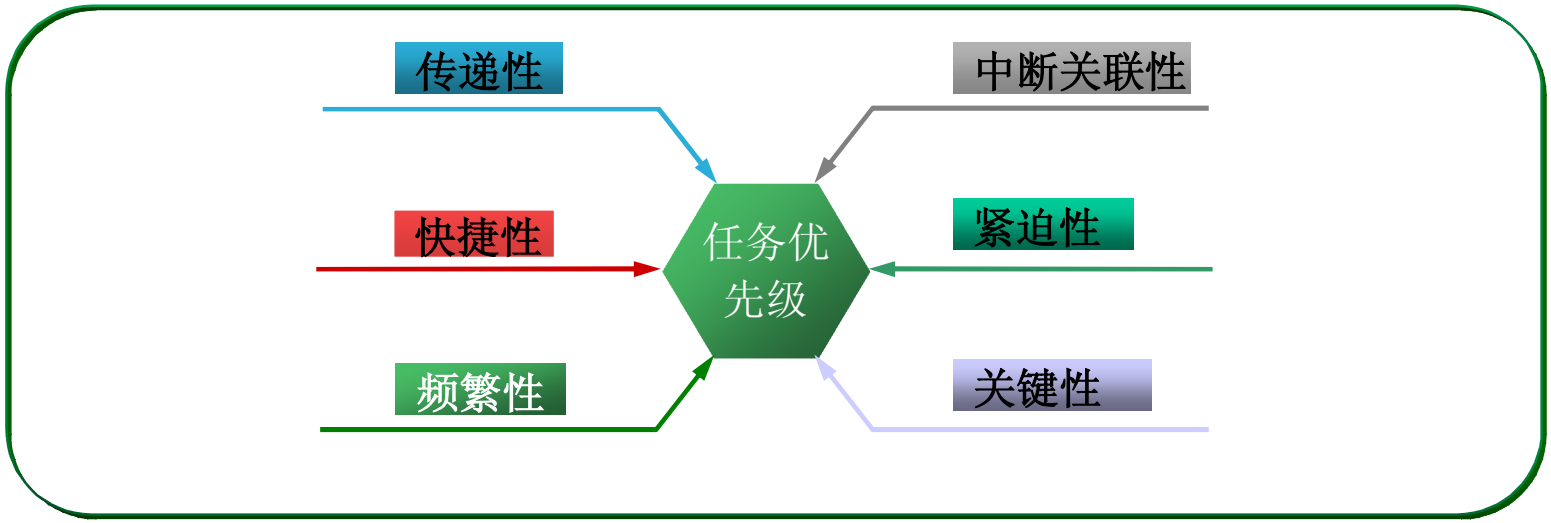
任务的划分

2.任务划分的方法

任务的划分方法，请参考下一章“基于LPC2000的电脑自动打铃器设计与实现”。

任务的优先级

任务的优先级安排原则如下：



任务的优先级

关键性：任务越关键安排的优先级越高，以保障其执行机会；

频繁性：对于周期性任务，执行越频繁，则周期越短，允许耽误的时间也越短，故应该安排的优先级也越高，以保障及时得到执行；

快捷性：在前面各项条件相近时，越快捷（耗时短）的任务安排的优先级越高，以使其它就绪任务的延时缩短；

传递性：信息传递的上游任务的优先级高于下游任务的优先级。如信号采集任务的优先级高于数据处理任务的优先级。

紧迫性：因为紧迫任务对响应时间有严格要求，在所有紧迫任务中，按响应时间要求排序，越紧迫的任务安排的优先级越高。紧迫任务通常与ISR关联；

中断关联性：与中断服务程序（ISR）有关联的任务应该安排尽可能高的优先级，以便及时处理异步事件，提高系统的实时性。如果优先级安排得比较低，CPU有可能被优先级比较高的任务长期占用，以致于在第二次中断发生时连第一次中断还没有处理，产生信号丢失现象；

系统函数概述

μC/OS-II程序设计基础

1. 基本原则
2. 系统函数的分类

基本原则

1.配对性原则

对于μC/OS-II来说，大多数API都是成对的，而且一部分必须配对使用。当然，查询状态的系统函数一般不需要配对使用，而且部分API如延时，也不需要配对使用。配对的函数见下表。

函数1	功能	函数2	功能	备注
OSFlagPend()	等待事件标志组的事件标志位	OSFlagPost()	置位或清0事件标志组中的标志	不必须配对使用，一般不在同一个任务中（用于资源同步时必须同一个任务中配对使用）
OSMboxPend()	等待消息邮箱中的消息	OSMboxPost()或OSMboxPostOpt()	以不同的方式向消息邮箱发送消息	
OSQPend()	等待消息队列中的消息	OSQPost()或OSQPostFront()或OSQPostOpt()	以不同的方式向消息队列发送一条消息	
OSSemPend()	等待一个信号量	OSSemPost()	发送一个信号量	
OSMutexPend()	等待一个互斥信号量	OSMutexPost()	释放一个互斥信号量	必须在同一个任务中配对使用

基本原则

函数1	功能	函数2	功能	备注
OSFlagCreate()	建立事件标志组	**OSFlagDel()**	删除事件标志组	动态使用事件时必须配对使用
OSMboxCreate()	建立消息邮箱	**OSMboxDel()**	删除消息邮箱	
OSMutexCreate()	建立互斥信号量	**OSMutexDel()**	删除互斥信号量	
OSQCreate()	建立消息队列	**OSQDel()**	删除消息队列	
OSSemCreate()	建立信号量	**OSSemDel()**	删除信号量	

基本原则

函数1	功能	函数2	功能	备注
OSMemGet()	分配一个内存块	**OSMemPut()**	释放一个内存块	必须配对使用
OSTaskCreate()或 OSTaskCreateExt()	建立任务	**OSTaskDel()**	删除任务	动态使用任务时必须配对使用
OSTaskSuspend()	挂起任务	**OSTaskResume()**	恢复任务	必须配对使用
OSTimeDly()或 OSTimeDlyHMSM()	延时	**OSTimeDlyResume()**	恢复延时的任务	不必配对使用。OSTimeDlyHMSM()可能需要多个OSTimeDlyResume()才能恢复

基本原则

函数1	功能	函数2	功能	备注
OSTimeGet()	获得系统时间	**OSTimeSet()**	设置系统时间	不必配对使用
OSIntEnter()	进入中断处理	**OSIntExit()**	退出中断处理	必须在中断服务程序中配对使用
OSSchedLock()	给调度器上锁	**OSSchedUnlock()**	给调度调度	必须在一个任务中配对使用
OS_ENTER_CRITICAL()	进入临界区	**OS_EXIT_CRITICAL()**	退出临界区	必须在一个任务或中断中配对使用

基本原则

2.中断服务程序调用函数的限制

中断服务程序不能调用可能会导致任务调度的函数，它们主要是一些等待事件的函数，这些函数及其替代函数见下表。

禁止使用的函数	替代函数	功能	备注
OSFlagPend()	OSFlagAccept()	无等待获得事件标志组的事件标志位	需要程序自己判断是否获得了相应的事件
OSMboxPend()	OSMboxAccept()	无等待获得消息邮箱中的消息	
OSMutexPend()	OSMutexAccept()	无等待获得一个互斥信号量	
OSQPend()	OSQAccept()	无等待获得息队列中的消息	
OSSemPend()	OSSemAccept()	无等待获得一个信号量	

注意：未列入表中的函数OSTaskCreate()、OSTaskCreateExt()、OSTaskDel()、OSTaskResume()、OSTaskChangePrio()、OSTaskSuspend()、OSTimeDly()、OSTimeDlyHMSM()、OSTimeResume()都属于在中断服务程序中禁止调用的函数。

基本原则

一些函数虽然没有明确地规定不能被中断服务程序调用，但因为中断服务程序的特性，一般不会使用。

1.创建事件和删除事件的函数。

2.与任务相关的函数OSTaskChangePrio() 、 OSTaskDelReq() 、OSTaskStkChk() 和OSTaskQuery() 。至于函数OSSchedLock() 和OSSchedUnlock()，在中断服务程序中使用没有任何意义。

基本原则

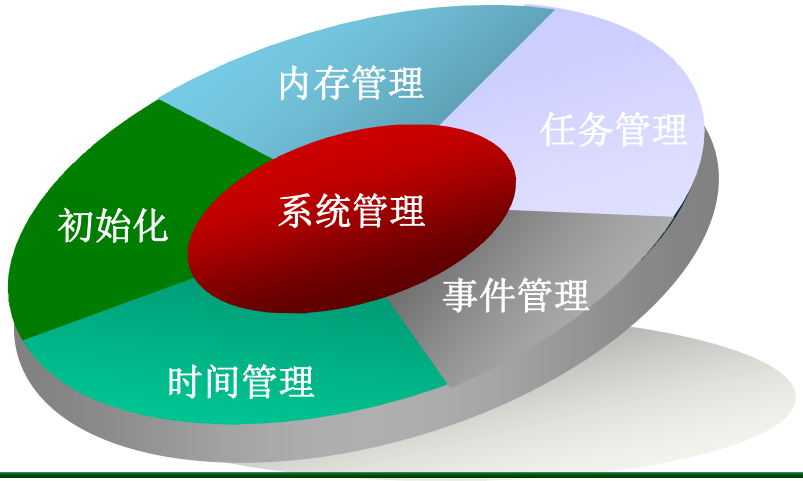
3.任务必须调用某个系统函数

因为μC/OS-II是完全基于优先级的操作系统，所以在一定的条件下必须出让CPU占有权以便比自己优先级更低的任务能够运行，这是通过调用部分系统函数来实现的，这些函数见下表。一般的任务必须调用表中至少一个函数，只有一种情况例外，就是单次执行的任务，因为任务删除后肯定出让CPU，所以可以不调用表中的函数。

函数名	功能	函数名	功能
OSFlagPend	等待事件标志组的事件标志位	OSMutexPend	等待一个互斥信号量
OSQPend	等待消息队列中的消息	OSQPend	等待消息队列中的消息
OSSemPend	等待一个信号量	OSTaskSuspend	挂起任务
OSTimeDly	延时	OSTimeDlyHMSM	延时

系统函数的分类

根据功能分类

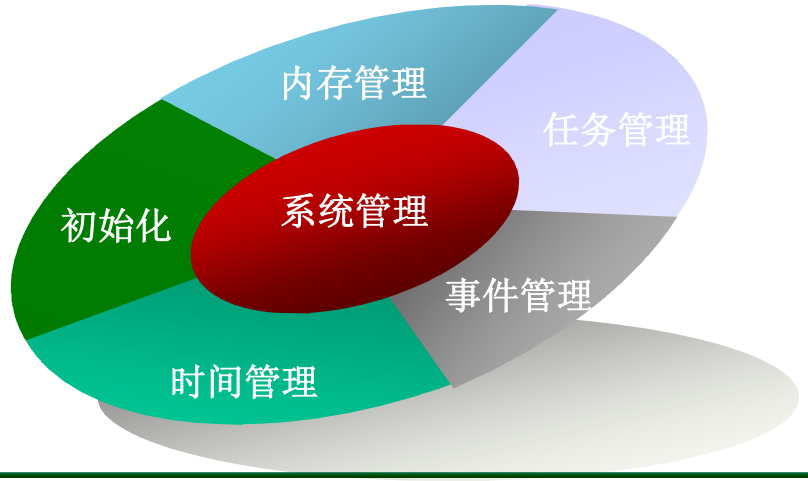


系统管理函数是一些与μC/OS-II内核或功能相关的一些函数，详见下表。

函数名	功能	备注
OSStatInit()	使能任务统计功能	复位一次只能调用一次，并且必须在任务中调用，在调用时其它用户任务不能处于就绪状态
OSIntEnter()	进入中断处理	必须由中断服务程序按照规范调用，使用本公司的模版就不需要调用它们
OSIntExit()	退出中断处理	
OSSchedLock()	锁调度器	必须配对使用，一般情况不需要使用。事实上，μC/OS-II不推荐使用它们
OSSchedUnlock()	解锁调度器	
OS_ENTER_CRITICAL()	进入临界区	必须配对使用，一般通过禁止中断和允许中断来实现的。对于一些移植代码来说，不能嵌套调用
OS_EXIT_CRITICAL()	退出临界区	

系统函数的分类

根据功能分类



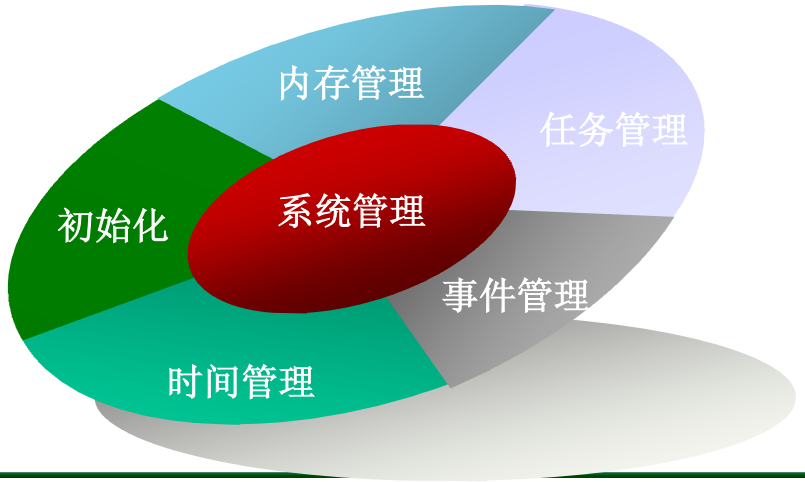
μC/OS-II的初始化函数有2个：OSInit()和OSStart()，它们不能在任何任务和中断服务程序中使用，仅在main()函数中按照一定的规范被调用，其中OSInit()函数初始化μC/OS-II内部变量，OSStart()函数启动多任务环境。

μC/OS-II把信号量等都称为事件，管理它们的就是事件管理函数。μC/OS-II V2.52具有的事件有普通信号量、互斥信号量、事件标志组、消息邮箱和消息队列，这些都是μC/OS-II用于同步与通讯的工具，本章后述的内容将会详细介绍。

一般的操作系统都提供时间管理的函数，最基本的就是延时函数，μC/OS-II也不例外，μC/OS-II所具有的时间管理函数见下表。

系统函数的分类

根据功能分类

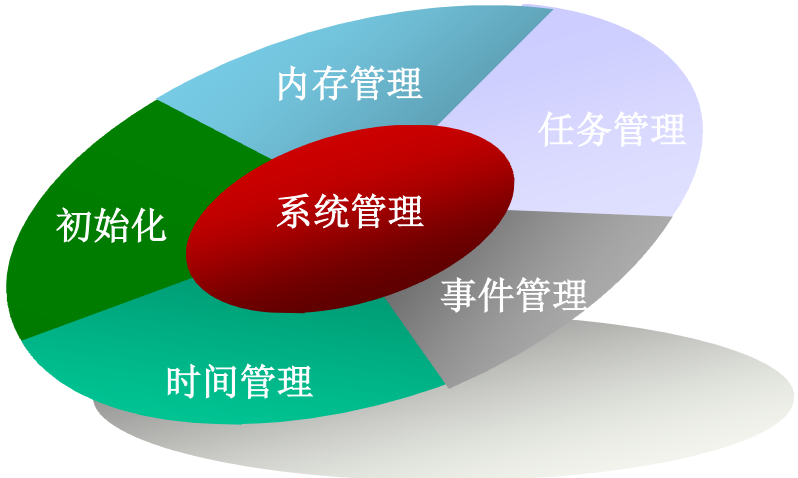


μC/OS-II具有简单的动态内存管理能力。μC/OS-II的动态内存管理函数见下表。

函数名	功能
OSMemCreate()	初始化一个堆
OSMemGet()	从指定堆中获得一个内存块
OSMemPut()	从指定堆中释放一个内存块
OSMemQuery()	查询指定堆的状态

系统函数的分类

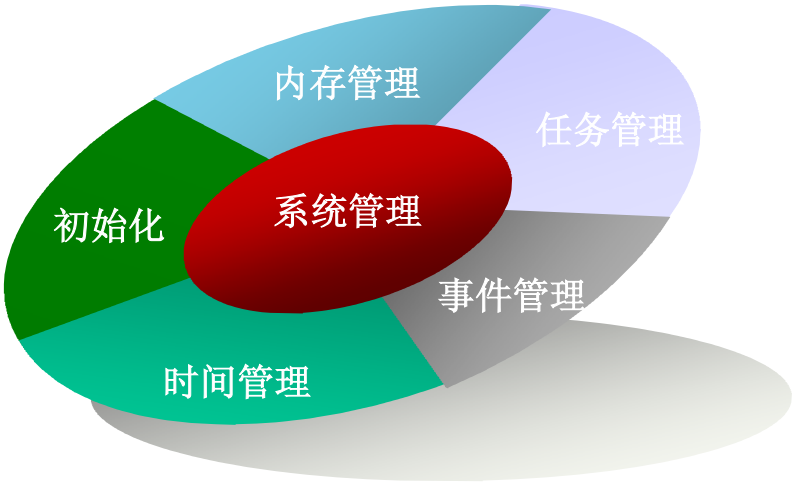
根据功能分类



函数名	功能	备注
OSTimeDly()	以时钟节拍为单位延时	
OSTimeDlyHMSM()	以钟时分秒毫秒为单位延时	
OSTimeDlyResume()	恢复延时的任务	OSTimeDlyHMSM()可能需要多次才能恢复
OSTimeGet()	获得系统时间	以时钟节拍为单位
OSTimeSet()	设置系统时间	以时钟节拍为单位
OSTimeTick()	时钟节拍处理函数	由时钟节拍中断处理程序调用，用户很少使用

系统函数的分类

根据功能分类



任务管理函数是操作与任务相关功能的函数，详见下表。

函数名	功能	函数名	功能
OSTaskChangePrio()	改变任务优先级	OSTaskSuspend()	挂起任务
OSTaskCreate()	建立任务	OSTaskResume()	恢复任务
OSTaskCreateExt()	建立任务，比OSTaskCreate()控制任务属性更多	OSTaskStkChk()	检查堆栈
OSTaskDel()	删除任务	OSTaskQuery()	获得任务信息
OSTaskDelReq()	请求删除任务，有特殊用途		

系统函数使用场合

μC/OS-II程序设计基础

1. 时间管理
2. 资源同步
3. 行为同步

时间管理

1.控制任务的执行周期

在任务函数的代码中可以通过插入延时函数来控制任务周期性运行，定时闲置CPU一段时间，供其它任务使用。

```
void  MyTask (void *pdata)
{
    进行准备工作的代码;
    while (1) {
        任务实体代码;
        调用系统延时函数;
    }
}
```

- while (1) { → 无限循坏
- 调用系统延时函数; → 调用OSTimeDly( )或OSTimeDlyHMSM( )

注意：延时函数OSTimeDly()是以系统节拍数为参数，而延时函数OSTimeDlyHMSM()是以实际时间值为参数，但在执行过程中仍然转换为系统节拍数。如果实际时间不是系统节拍的整数倍，将进行四舍五入处理。设系统节拍为50毫秒，调用OSTimeDly(20)的效果是延时1秒钟，调用OSTimeDlyHMSM(0,1,27,620)的实际时间是延时1分27秒600毫秒。

时间管理

2.控制任务的运行节奏

在任务函数的代码中也可以通过插入延时函数来控制任务的运行节奏。

```
void  MyTask (void *pdata)
{
    进行准备工作的代码;
    while(1) {
        调用获取事件的函数;
        第一部分操作代码;
        调用系统延时函数;
        第二部分操作代码;
        调用系统延时函数;
        第三部分操作代码;
        …
    }
}
```

各种时间顺序控制任务可以用这种结构的任务函数实现。

- 无限循环
- 等待信号量、消息
- 调用OSTimeDly( )或OSTimeDlyHMSM( )
- 调用OSTimeDly( )或OSTimeDlyHMSM( )
- 其他任务代码

时间管理

3.状态查询

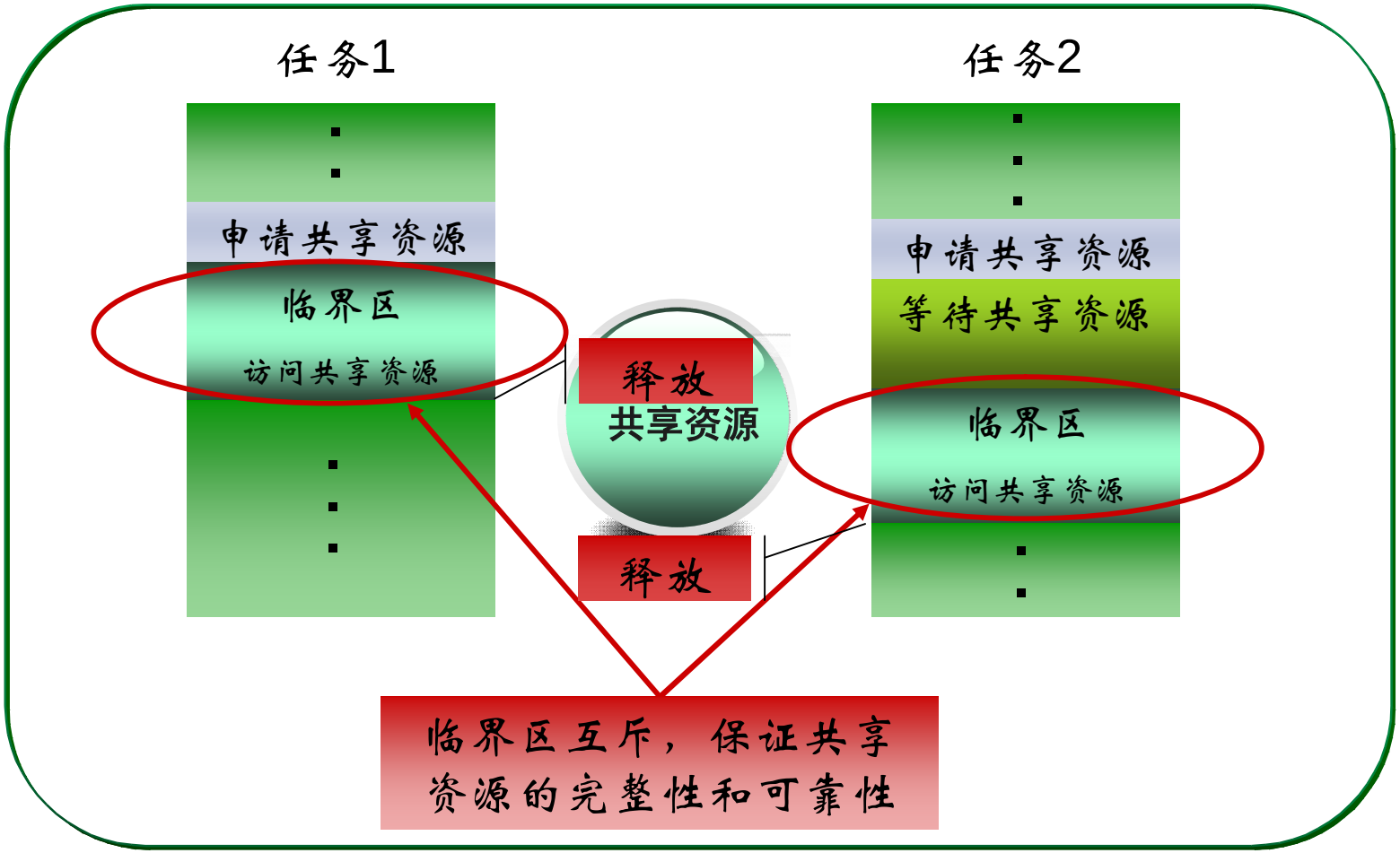
查询过程是一个无限循环过程，只有当希望的状态出现以后才能退出这个无限循环，这种情况在实时操作系统管理下是不允许的，它将剥夺低优先级任务的运行机会。解决这个问题的办法是“用定时查询代替连续查询”。

```
void  MyTask (void *pdata)
{
    进行准备工作的代码;
    while(1) {
        while (查询的条件不成立) {
            调用系统延时函数;
        }
        其它处理代码;
    }
}
```

- 无限循环
- 定时查询条件是否成立
- 不成立，延时再查询，同时让出cpu占有权，供低优先级任务使用
- 查询条件成立，运行后续代码

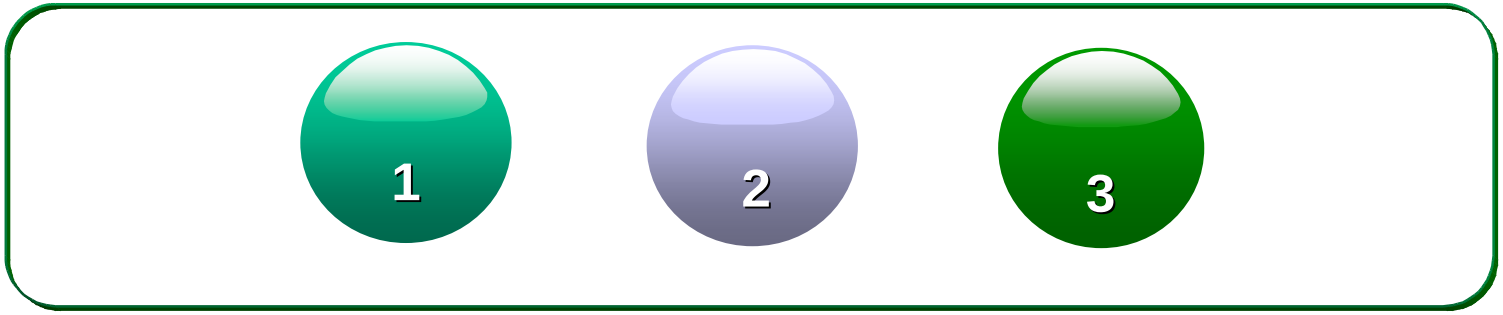
资源同步

1.“资源同步”图解



资源同步

2.“资源同步”实现方式



1.使用关中断：通过调用禁止中断函数OS_ENTER_CRITICAL()和允许中断函数OS_EXIT_CRITICAL()实现的。

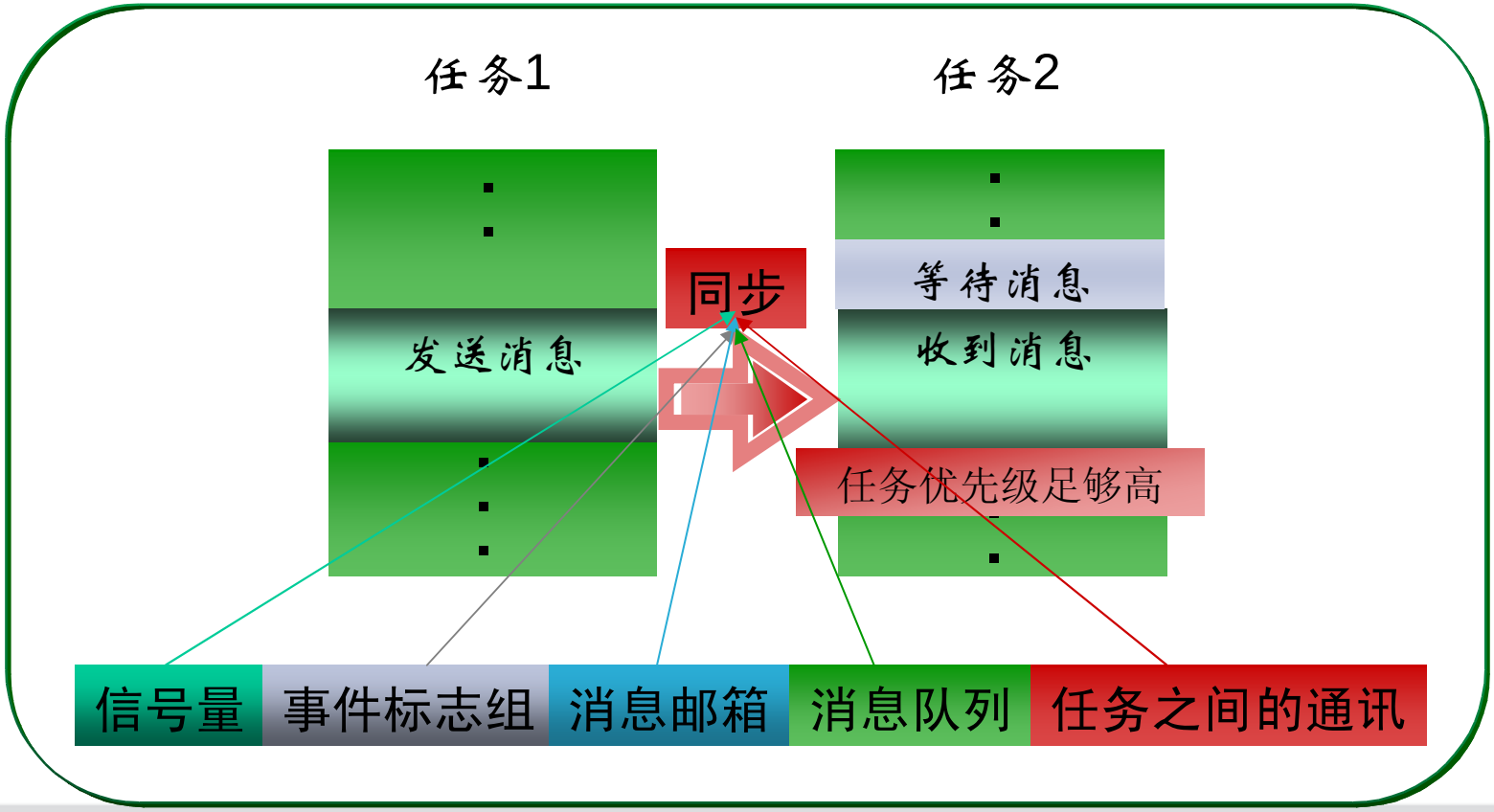
2.使用关调度：通过调用禁止任务调度函数OSSchedLock()和允许任务调度函数OSSchedUnlock()实现的，因为禁止调度违背了多任务的初衷，所以不建议用户使用。

3.使用信号量与互斥信号量：通过等待信号量和发送信号量实现共享资源的独享。

行为同步

1.行为同步

一个任务的运行过程需要和其它任务的运行配合，才能达到预定的效果，任务之间的这种动作配合和协调关系称为“行为同步”。



行为同步

2.数据通信

	数据通讯时是否需要遵守“资源同步”规则	数据通讯的同时是否具有“行为同步”功能
消息邮箱	不需要	有
消息队列	不需要	有
全局变量	需要	无

注意：尽管指针可能是局部变量，但只要指针指向的变量是全局变量，操作指针指向的变量时也需要当作全局变量来处理。

时间管理

μC/OS-II程序设计基础

1. 系统延时函数OSTimeDly()
2. 系统延时函数OSTimeDlyHMSM()
3. 强制延时的任务结束延时OSTimeDlyResume()
4. 获得系统时间OSTimeGet()和设置系统时间OSTimeSet()

概述

μC/OS-II提供了若干个时间管理服务函数，可以满足任务在运行过程中对时间管理的需求。在使用时间管理服务函数时，必须十分清楚一个事实：**时间管理服务函数是以系统节拍为处理单位的，实际的时间与希望的时间是有误差的，最坏的情况下误差接近一个系统节拍。**因此时间管理服务函数只能用在对时间精度要求不高的场合，或者时间间隔较长的场合。

OSTimeDly()

系统延时函数OSTimeDly()调用图解

OSTimeDly()

下面我们设计一个任务，让一个LED以50个时钟节拍为单位闪耀，说明OSTimeDly()函数的用途。由于篇幅关系，只给出任务主要处理代码。

```
void  TaskLED (void  *pdata)
{
    ……
    while (1) {
        IO0CLR = LED1;
        OSTimeDly(25);
        IO0SET = LED1;
        OSTimeDly(25);
    }
}
```

初始化代码

循坏控制LED以固定频率闪烁

点亮LED

延时25个节拍

熄灭LED

延时25个节拍

注意：上面的设计是OSTimeDly() 控制任务的周期性执行，还可以用它来控制任务的运行节拍。

OSTimeDlyHMSM()

μC/OS-Ⅱ提供了OSTimeDlyHMSM()系统延时函数，这个函数是以小时(H)、分(M)、秒(S)和毫秒(m)四个参数来定义延时时间的，函数在内部把这些参数转换为时钟节拍，再通过单次或多次调用OSTimeDly()进行延时和任务调度，所以延时原理和调用延时函数OSTimeDly()是一样的。OSTimeDlyHMSM()详细见下表。

函数名称	OSTimeDlyHMSM	所属文件	OS_TIMC.C
函数原型	INT8U OSTimeDlyHMSM (INT8U hours, INT8U minutes, INT8U seconds, INT16U milli)		
功能描述	延时，指定的延时时间为时、分、秒、毫秒		
函数参数	hours：小时，minutes：分钟，seconds：秒，milli：毫秒		
函数返回值	OS_TIME_INVALID_MINUTES：minutes参数错误 OS_TIME_INVALID_SECONDS：seconds参数错误 OS_TIME_INVALID_MILLI：milli参数错误		
特殊说明	(1) 所有参数为0时不延时，函数直接返回 (2) 必须正确设置全局常数OS_TICKS_PER_SEC，否则延时时间是错误的 (3) 因为OSTimeDlyHMSM()是通过多次(或1次)调用OSTimeDly()实现的，所以**延时分辨率为时钟节拍** (4) 因为OSTimeDlyHMSM()是通过多次(或1次)调用OSTimeDly()实现的，所以**可能需要调用多次OSTimeDlyResume()才能恢复延时的任务**		

OSTimeDlyHMSM()

为了说明OSTimeDlyHMSM()函数的使用方法，下面我们设计一个任务，让一个LED以2Hz的频率闪耀。下面给出任务主要处理代码。

```
void  TaskLED (void  *pdata)
{
    ……
    while (1) {
        IO0CLR = LED1;
        OSTimeDlyHMSM(0, 0, 0, 250);
        IO0SET = LED1;
        OSTimeDlyHMSM(0, 0, 0, 250);
    }
}
```

初始化代码

循坏控制LED以固定频率闪烁

点亮LED

250ms延时

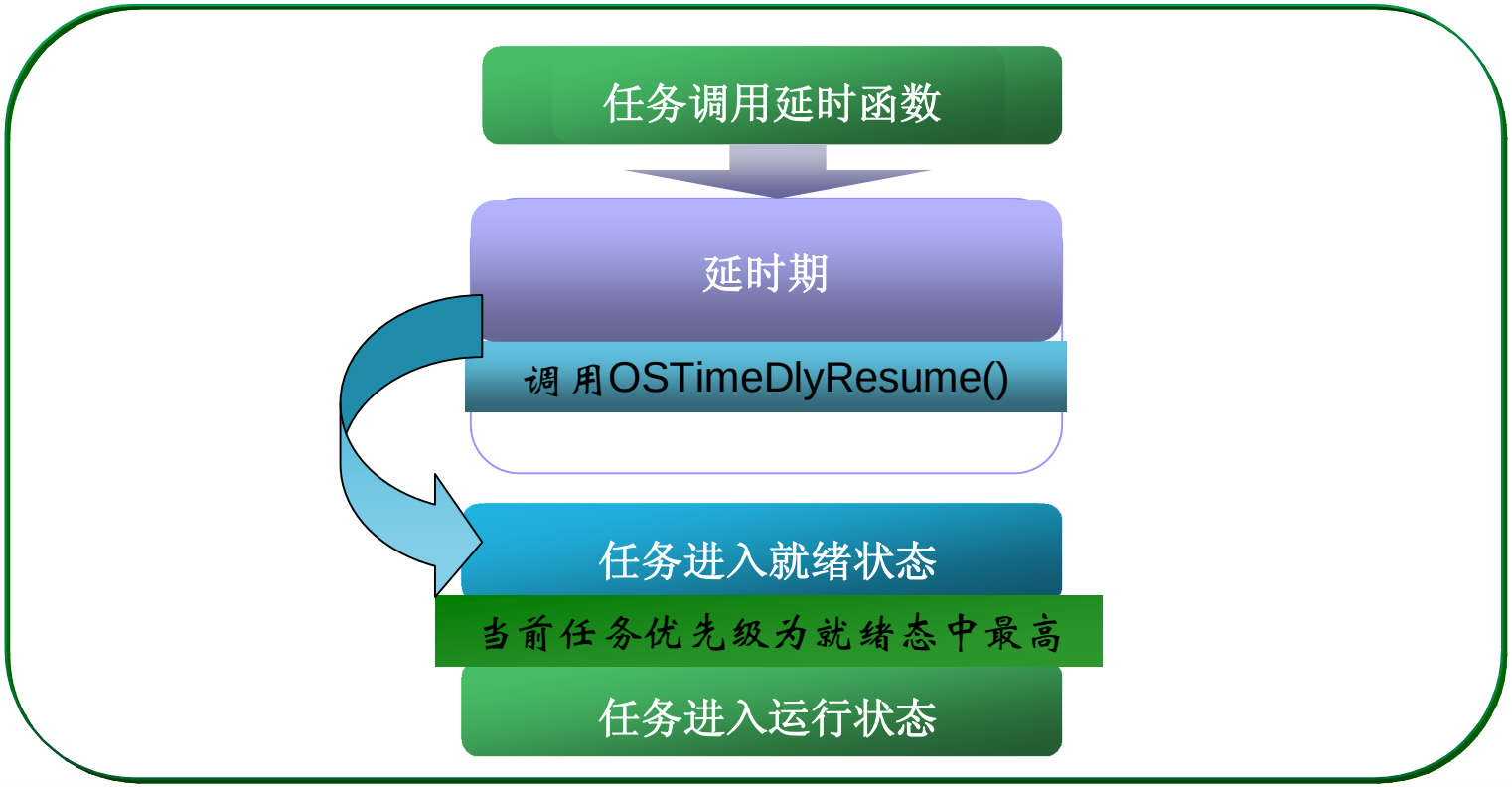
熄灭LED

250ms延时

注意：上面的设计是OSTimeDlyHMSM()控制任务的周期性执行，还可以用它来控制任务的运行节拍。

OSTimeDlyResume()

μC/OS-Ⅱ允许用户结束正处于延时期的任务，延时的任务可以不等待延时期满，而是通过取消其它任务的延时来使自己处于就绪态，可以通过调用OSTimeDlyResume()和指定要恢复的任务的优先级来完成。



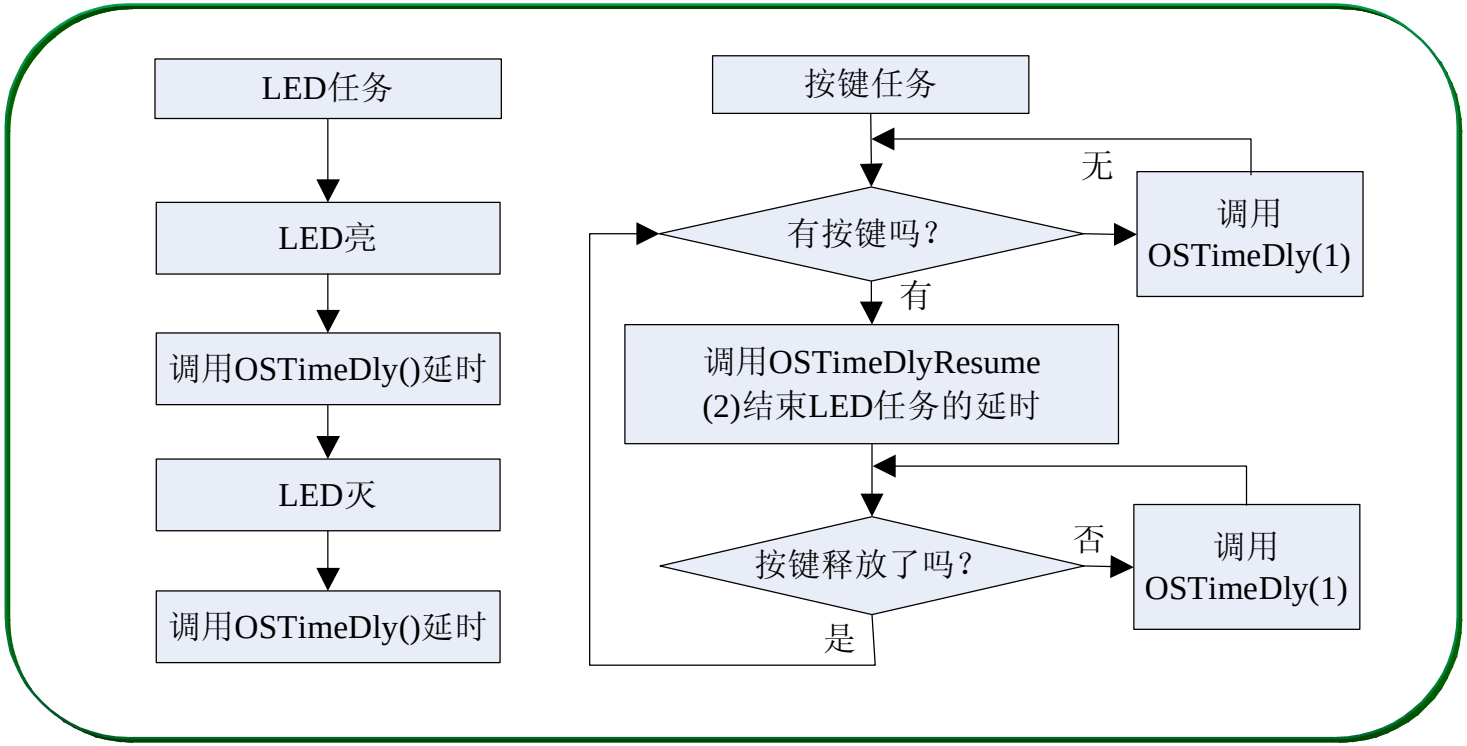
OSTimeDlyResume()

OSTimeDlyResume()的具体信息见下表。

函数名称	OSTimeDlyResume	所属文件	OS_TIMC.C
函数原型	INT8U OSTimeDlyResume(INT8U prio)		
功能描述	让延时的任务结束延时		
函数参数	prio：任务优先级		
函数返回值	OS_NO_ERR：成功，OS_PRIO_INVALID：prio错误 OS_TIME_NOT_DLY：任务没有延时，OS_TASK_NOT_EXIST：任务不存在		
特殊说明	**因为OSTimeDlyHMSM()是通过多次(或1次)调用OSTimeDly()实现的，所以可能需要调用多次OSTimeDlyResume()才能恢复延时的任务**		

OSTimeDlyResume()

为了说明OSTimeDlyResume()函数的使用方法，我们设计一个系统，假设TaskLED的任务优先级为2。让一个LED以0.5Hz的频率闪耀，但每按键一次，LED状态翻转一次。下面是两个任务的处理流程。



OSTimeDlyResume()

TaskLED任务代码如下。

```
void  TaskLED (void  *pdata)
{
      ……
      while (1) {
            IO0CLR = LED1;
            OSTimeDly(OS_TICKS_PER_SEC);
            IO0SET = LED1;
            OSTimeDly (OS_TICKS_PER_SEC);
      }
}
```

初始化代码

循环控制LED以固定频率闪烁

点亮LED

1s延时

熄灭LED

1s延时

OSTimeDlyResume()

TaskKEY任务的代码如下。

```
void  TaskKEY (void  *pdata)
{
      ......
      while (1) {
            while ((IO0PIN & KEY1) != 0) {
                  OSTimeDly(1);
            }
            OSTimeDlyResume(2);
            while ((IO0PIN & KEY1) == 0) {
                  OSTimeDly(1);
            }
      }
}
```

初始化代码

等待按健按下

延时1个节拍，用于任务切换

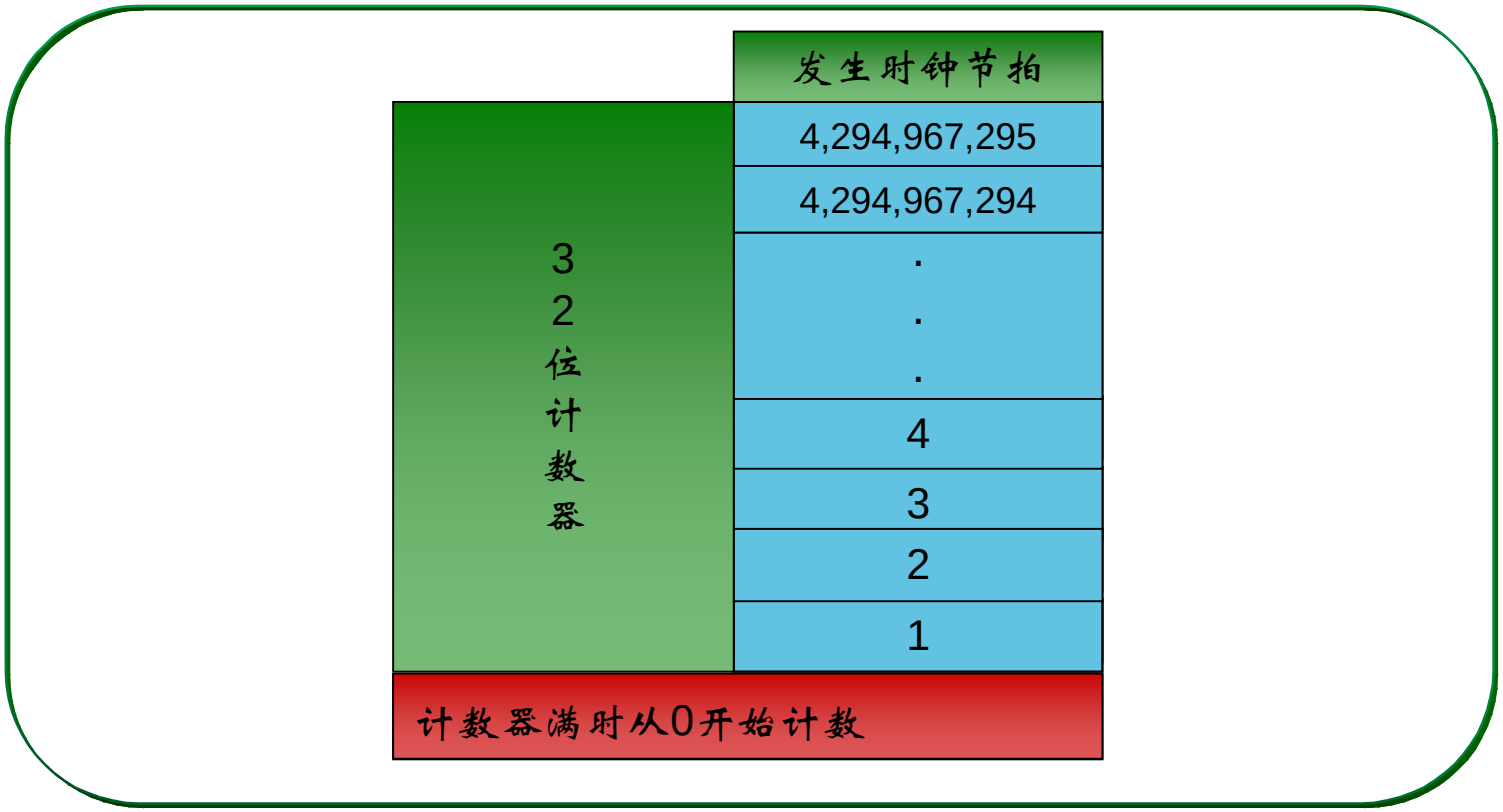
TaskLED优先级为2，恢复TaskLED

等待按健释放

延时1个节拍，用于任务切换

OSTimeGet()、OSTimeSet()

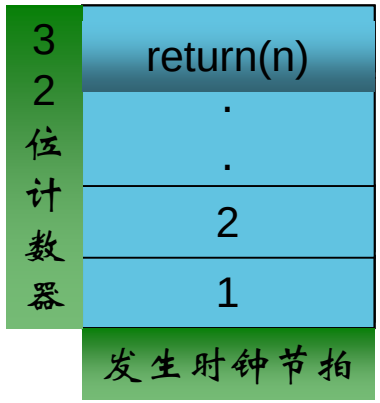
无论时钟节拍何时发生，μC/OS-Ⅱ都会将一个32位的计数器加1，这个计数器在用户调用OSStart()初始化多任务和4,294,967,295个节拍执行完一遍的时候从0开始计数。



OSTimeGet()、OSTimeSet()

用户可以通过调用OSTimeGet()来获得该计数器的当前值，OSTimeGet()的详细信息见下表。

函数名称	OSTimeGet	所属文件	OS_TIMC.C
函数原型	INT32U OSTimeGet(void)		
功能描述	获得系统时间		
函数参数	prio：任务优先级		
函数返回值	系统时间		

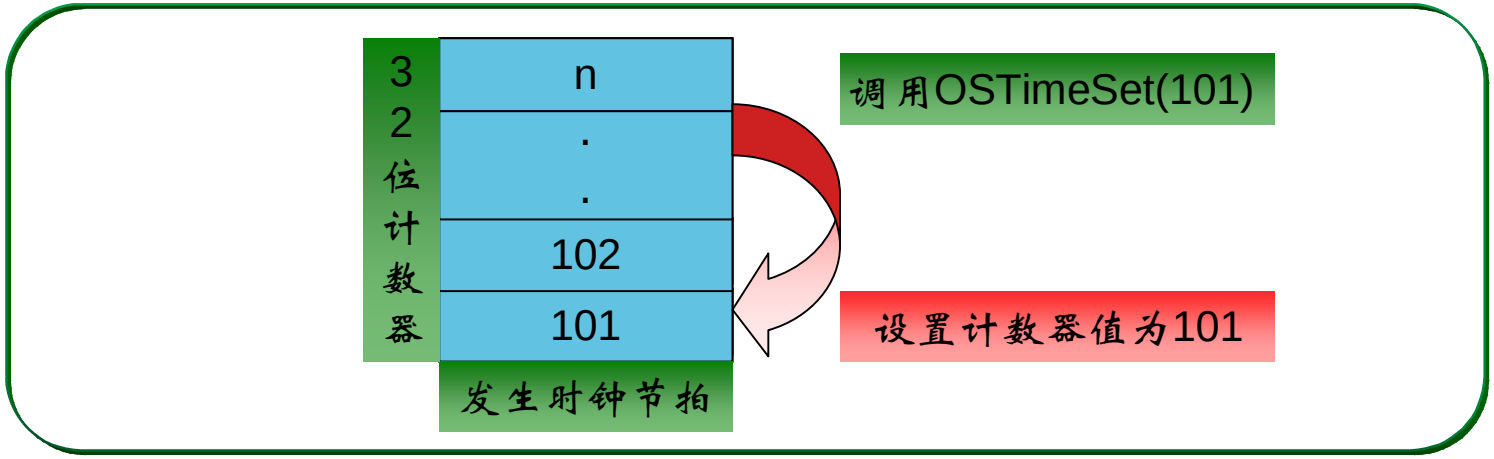



```
ticks = OSTimeGet();
```

OSTimeGet()、OSTimeSet()

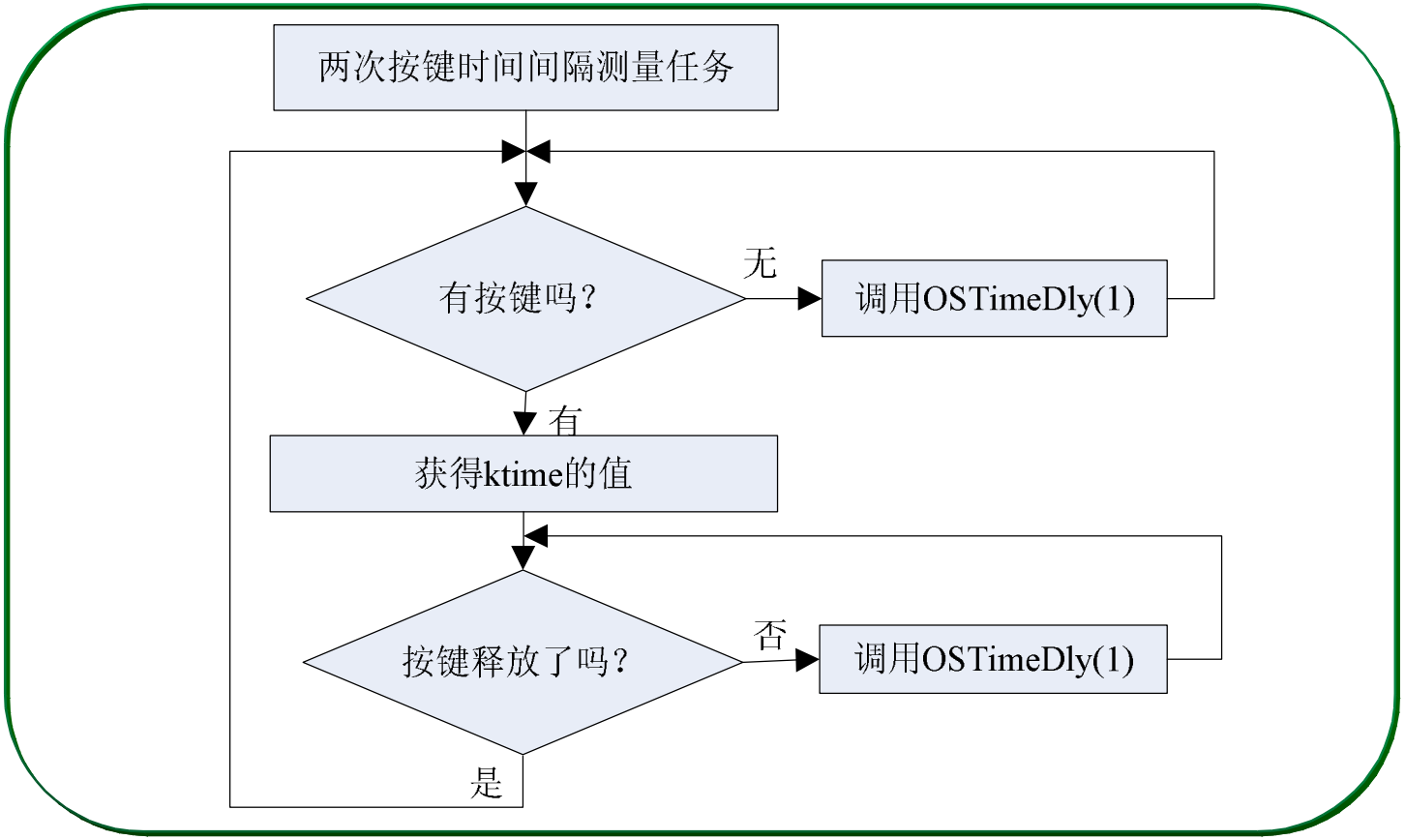
用户可以通过调用OSTimeSet()来改变计数器的值，OSTimeSet()的详细信息见下表。

函数名称	OSTimeSet	所属文件	OS_TIMC.C
函数原型	void OSTimeSet(INT32U ticks)		
功能描述	设置系统时间		
函数参数	ticks：需要设置的值		
函数返回值	无		
特殊说明	很少使用		



OSTimeGet()、OSTimeSet()

为了说明OSTimeGet()函数的使用方法，我们设计一个任务，计算两次按键的时间间隔放在全局变量ktime中。下面是任务的处理流程。



OSTimeGet()、OSTimeSet()

TaskKEY任务代码如下。

```
void  TaskKEY (void  *pdata)
{
    ……
    while (1) {
        while ((IO0PIN & KEY1) != 0) {
            OSTimeDly(1);
        }
        ktime = OSTimeGet() - ktime;
        while ((IO0PIN & KEY1) == 0) {
            OSTimeDly(1);
        }
    }
}
```

初始化代码

等待按健按下

延时1个节拍，用于任务切换

获得按键间隔时间

等待按健释放

延时1个节拍，用于任务切换

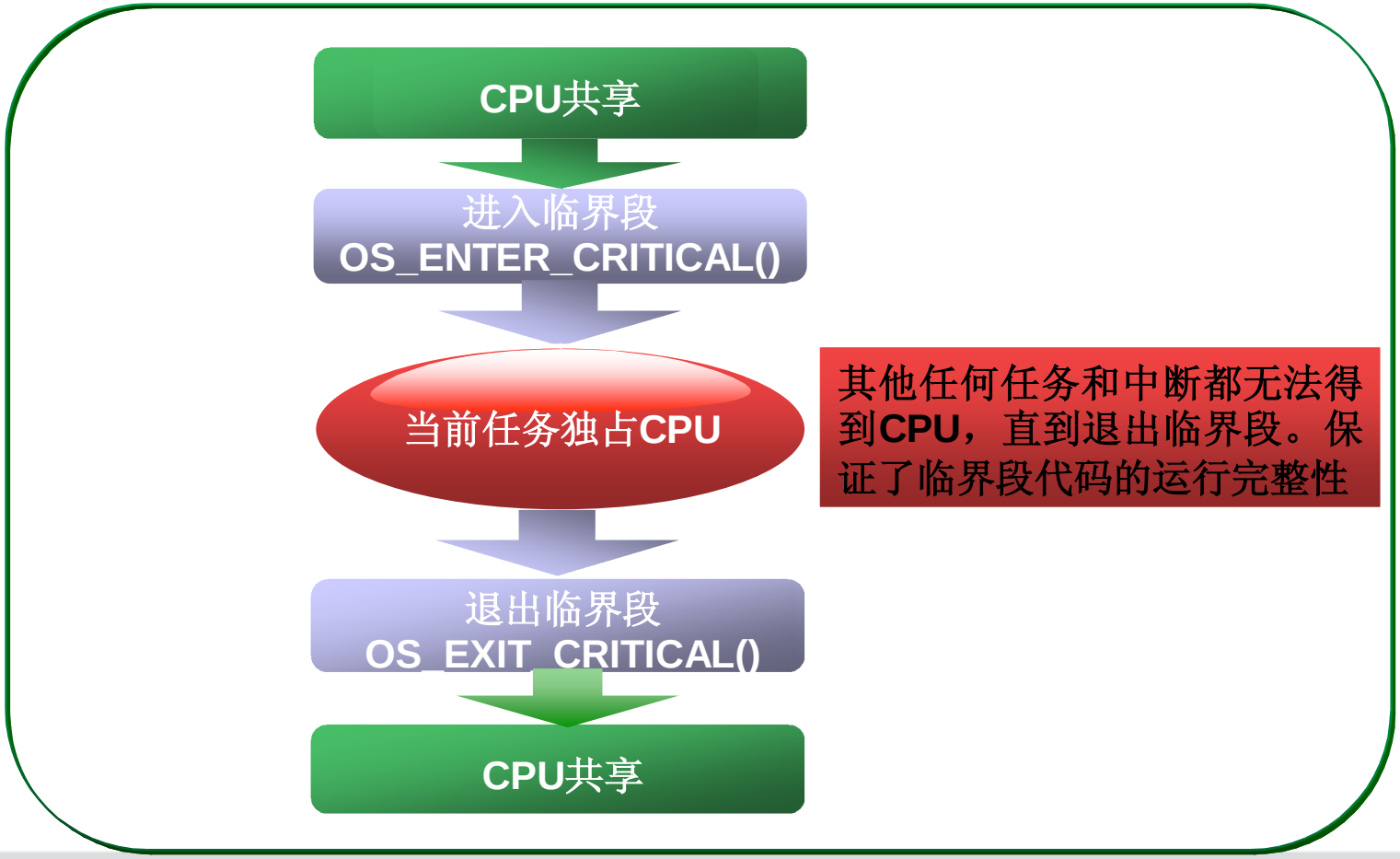
系统管理

μC/OS-II程序设计基础

1. 进入然后退出临界区
2. 禁止然后允许调度

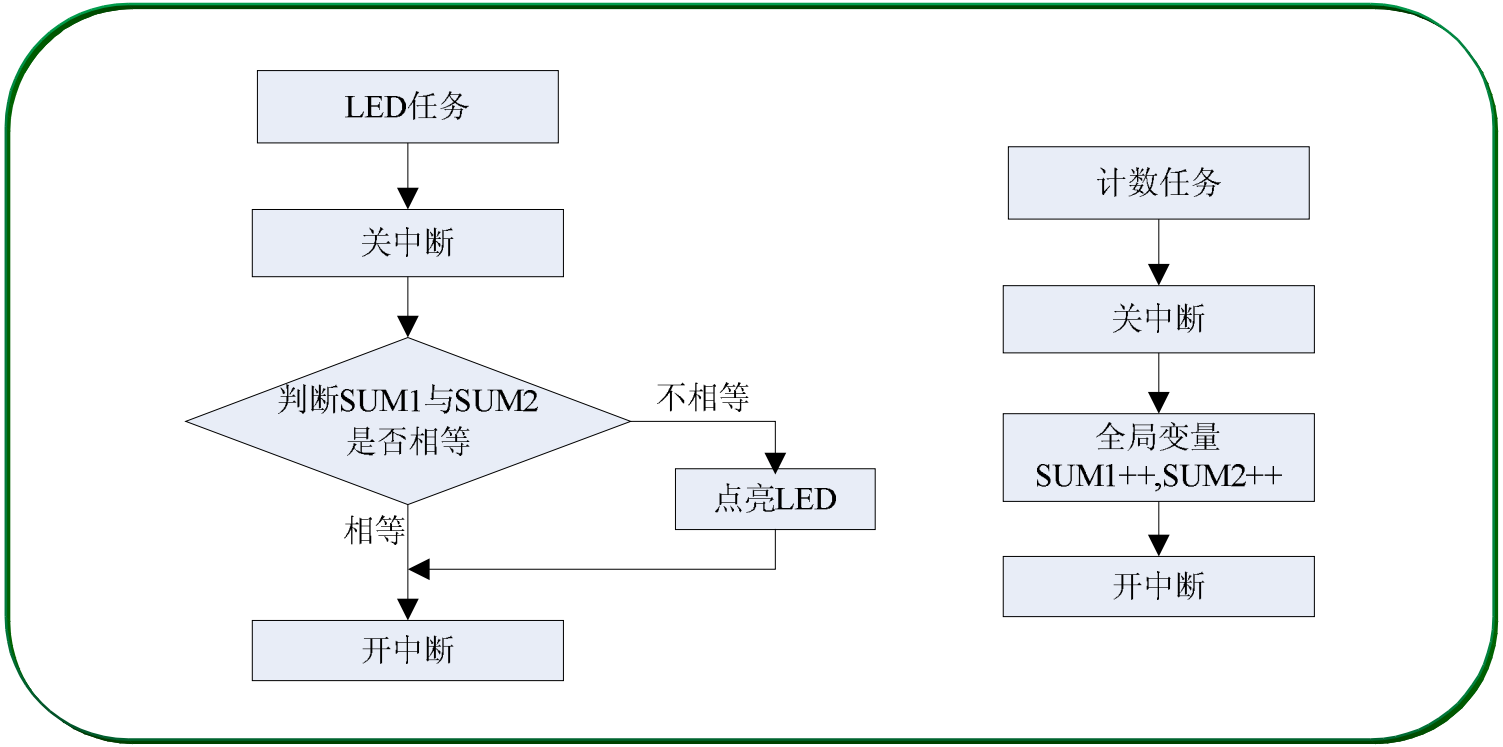
进入然后退出临界区

进入然后退出临界段是“资源同步”的方法之一，能够在访问共享资源时保障信息的可靠性和完整性。



进入然后退出临界区

为了说明它在”资源同步”时的使用，我们设计一个系统，假设有两个任务，它们都对全局变量sum1和sum2操作。低优先级任务让这两个变量始终相等，并不断在计数；高优先级任务不断的判断这两个变量是否相等，不相等则点亮LED，下面是两个任务的处理流程。



进入然后退出临界区

TaskLED任务代码如下。

```
void  TaskLED (void  *pdata)
{
    ……
    while (1) {
        OS_ENTER_CRITICAL();
        if (sum1 != sum2) {
            IO0CLR = LED1;
        }
        OS_EXIT_CRITICAL();
        OSTimeDly(2);
    }
}
```

初始化代码

进入临界段

判断两个全局变量是否相等

两个全局变量不等，点亮LED

退出临界段

延时2个节拍，任务切换

进入然后退出临界区

TaskAdd任务代码如下。

```
void  TaskAdd (void  *pdata)
{
    ……
    while (1) {
        OS_ENTER_CRITICAL();
        sum1++;
        sum2++;
        OS_EXIT_CRITICAL();
        OSTimeDly(1);
    }
}
```

初始化代码

进入临界段

全局变量sum1自加

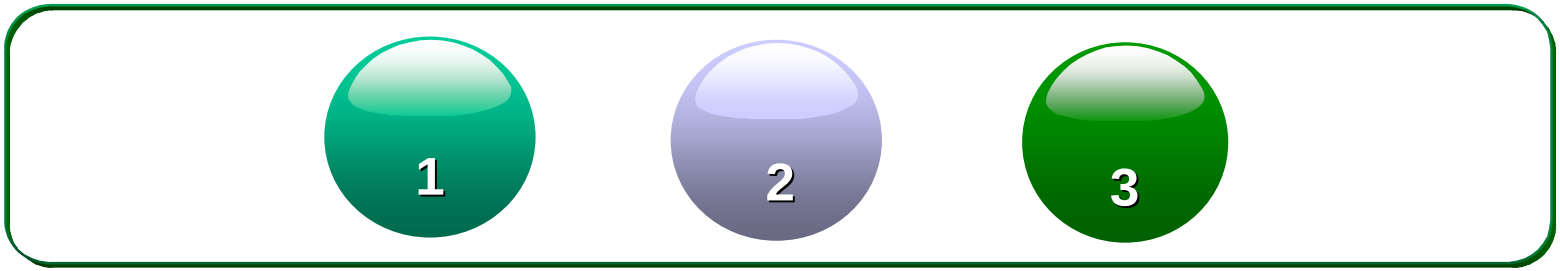
全局变量sum2自加

退出临界段

延时1个节拍，任务切换

禁止然后允许调度

给调度器上锁OSSchedlock()函数用于禁止任务调度，直到任务完成后调用给调度器开锁OSSchedUnlock()函数为止。使用它有3点需要注意。



1. OSSchedlock()和OSSchedUnlock()必须成对使用，也可以嵌套使用

2. OSSchedlock()只是禁止了任务的调度，而没有禁止中断，此时如果允许中断，中断到来时还是会执行对应的中断服务程序；

3.调用OSSchedLock()以后，用户的应用程序不得使用任何能将现行任务挂起的系统调用，直到配对的OSSchedUnlock()调用为止。

注意：对于用户来说，极少使用禁止然后允许调度的方法。不过，很多操作系统内部和驱动程序使用它来减少中断响应时间。

事件的一般使用规则

μC/OS-II程序设计基础

1. 相似性
2. 先创建后使用
3. 配对使用
4. 在ISR中使用

事件的一般使用规则 | μC/OS-II程序设计基础

相似性

事件管理函数是μC/OS-II中最多的系统函数，在μC/OS-II V2.52中总共有34个，而且每种事件具有的管理函数数目不同。但是所有的事件都有类似的6个函数，它们是所有事件的基本功能，其函数名类似，使用方法也类似，详细函数见下表。

功能	信号量	互斥信号量	事件标志组	消息邮箱	消息队列
建立事件	OSSemCreate	OSMutexCreate	OSFlagCreate	OSMboxCreate	OSQCreate
删除事件	OSSemDel	OSMutexDel	OSFlagDel	OSMboxDel	OSQDel
等待事件	OSSemPend	OSMutexPend	OSFlagPend	OSMboxPend	OSQPend
发送事件	OSSemPost	OSMutexPost	OSFlagPost	OSMboxPost	OSQPost
无等待获得事件	OSSemAccept	OSMutexAccept	OSFlagAccept	OSMboxAccept	OSQAccept
查询事件状态	OSSemQuery	OSMutexQuery	OSFlagQuery	OSMboxQuery	OSQQuery

另外还有4个事件管理函数为OSMboxPostOpt()、OSQPostFront()、 OSQPostOpt()、OSQFlush()，各函数详见后面章节。

事件的一般使用规则 | μC/OS-II程序设计基础

先创建后使用

任何一个事件，必须先创建后使用。创建事件是通过调用函数OS???Create()实现的，其中???为事件的类型。创建事件可以在main()函数中，但更多的是在任务初始化部分。使用方法如下。

```
OS_EVENT *event;

void Task0(void *pdata)
{
    pdata = pdata;
    event = OS???Create(…);
    while (1) {
        /* 其它代码 */
    }
}
```

定义事件

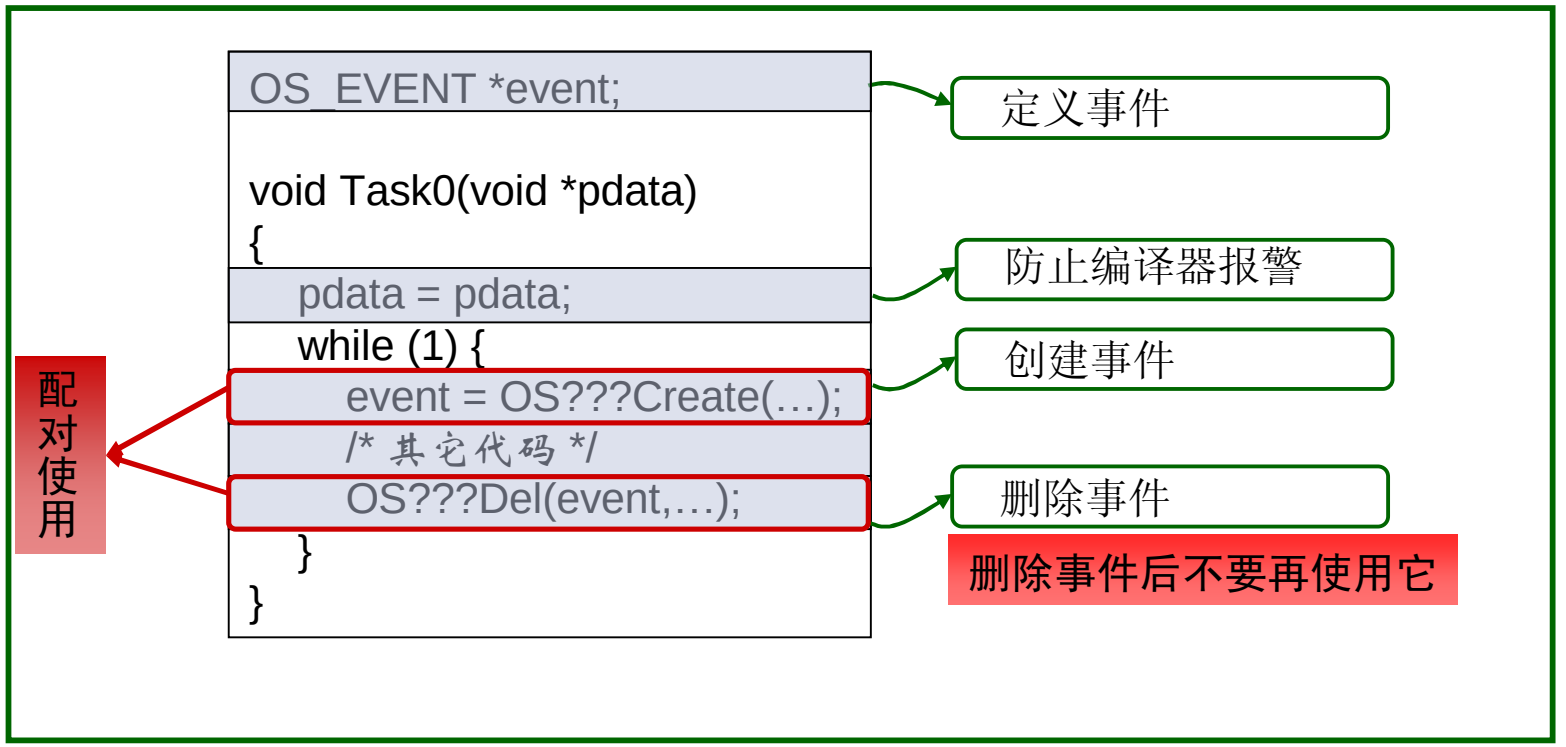
防止编译器报警

创建事件

事件的一般使用规则 | μC/OS-II程序设计基础

先创建后使用

一般来说，在嵌入式系统中，事件是静态使用的，即创建后永远不删除。但有时候需要动态使用，即根据需要创建和删除事件，此时创建事件就是在任务的事件执行代码中，使用方法如下。



事件的一般使用规则 | μC/OS-II程序设计基础

配对使用

由前面介绍可知，事件是动态使用时，建立事件和删除事件必须配对使用。下面给出一个示例，假设Task0为高优先级任务，Task1为低优先级任务。Task0代码如下。

```
OS_EVENT *event;
void Task0(void *pdata)
{
    pdata = pdata;
    event = OS???Create(…);
    while (1) {
        OS???Pend(event,…);
        /* 其它代码 */
    }
}
```

定义事件

防止编译器报警

创建事件

等待获得事件

1. 无等待获得事件OS???Accept()是等待事件的一种特殊形式，有事件时它与等待事件没有差别，没有事件时，它不等待，直接返回错误信息。
2. 因为已经具有无等待获得事件的功能，所以很少使用查询功能OSSemQuery()。

事件的一般使用规则 | μC/OS-II程序设计基础

配对使用

Task1代码如下。

```
void Task1(void *pdata)
{
    pdata = pdata;
    while (1) {
        /* 其它代码 */
        OS???Post(event,…);
        /* 其它代码 */
    }
}
```

防止编译器报警

发送事件

注意：一些事件有多个发送事件的函数，消息邮箱除OSMboxPost()外，还有增强型发送函数OSMboxPostOpt()，消息队列有OSQPost()、OSQPostFront()、OSQPostOpt()三种发送函数，各函数详见后面章节。

事件的一般使用规则 | μC/OS-II程序设计基础

在ISR中使用

要掌握事件函数在中断服务程序中的调用规则，我们必须清楚中断服务有哪些特点。

- 中断与所有的任务异步
- 中断服务程序总体是顺序结构
- 中断服务程序需要尽快退出
- 中断服务程序不能等待

事件的一般使用规则 | μC/OS-II程序设计基础

在ISR中使用

下面给出事件在中断服务程序中使用方法，假设Task0任务接收ISR发送的消息，任务代码如下。

```
OS_EVENT *event;
    ……
void Task0(void *pdata)
{
    pdata = pdata;
    event = OS???Create(…);
    while (1) {
        OS???Pend(event,…);
        /* 其它代码 */
    }
}
```

定义事件

防止编译器报警

创建事件

等待获得事件

事件的一般使用规则 | μC/OS-II程序设计基础

在ISR中使用

ISR中的代码如下。

```
void ISR(void)
{
    /* 其它代码 */
    OS???Post(event,…);
    /* 其它代码 */
}
```

发送事件

注意：

1. 中断服务程序一般不会调用建立和删除事件函数，否则要么没有起到事件的作用，要么程序很复杂；
2. 中断服务程序不能调用等待事件的函数，否则可能造成程序崩溃，可以调用无等待获得事件函数获得信号，但事实上，在中断中调用无等待获得事件的情况都很少。

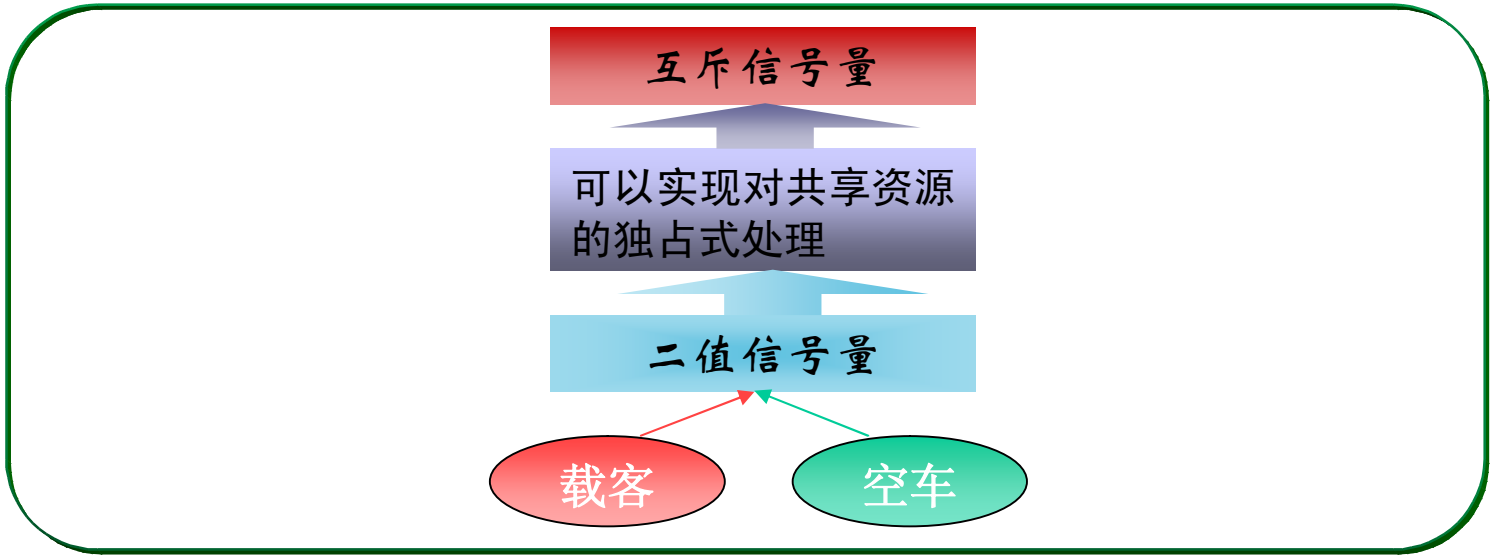
互斥信号量

μC/OS-II程序设计基础

互斥信号量 | μC/OS-II程序设计基础

简介

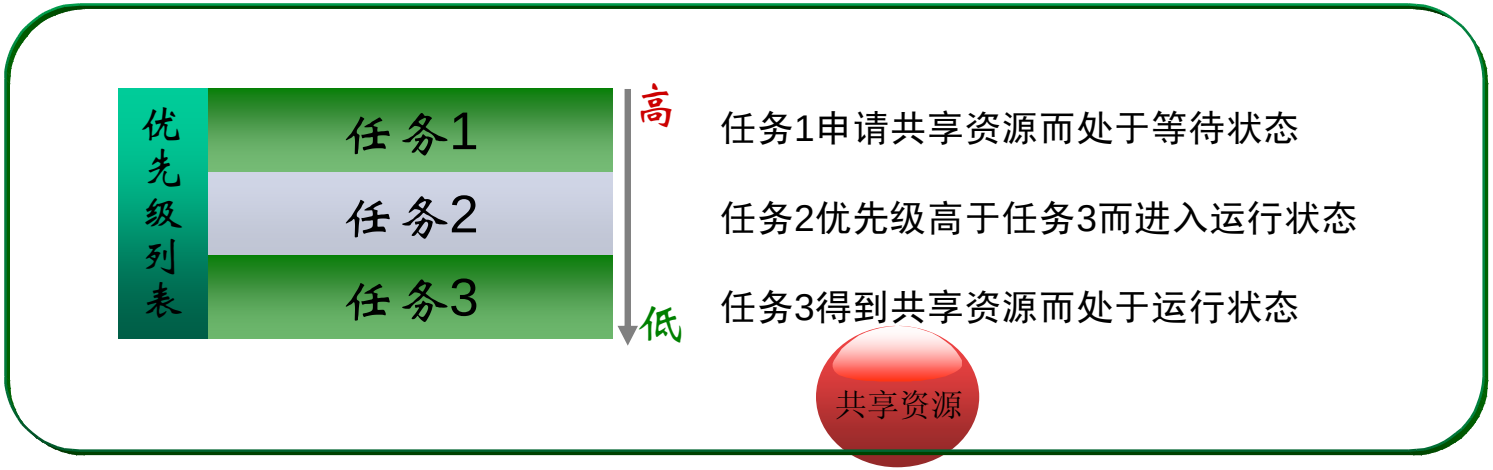
在日常生活中，出租车是一种常用的共享资源，当出租车载客时，从外面可以看到标识为载客；当空闲时，标识为空车。这样等车的人就可以根据标识知道出租车的当前状态，判断是否能够座上这辆车。这个标识牌就是一个二值信号量。由于这种二值信号量可以实现对共享资源的独占式处理，所以叫做互斥信号量。



互斥信号量也称为mutex，专用于资源同步。互斥信号量具有一些特性：占用一个空闲优先级，以便解决优先级反转问题。

简介

假设任务1和任务3共享一个资源，任务2为优先级介于任务1和任务3之间的一个与该共享资源无关任务，分析优先级反转问题。

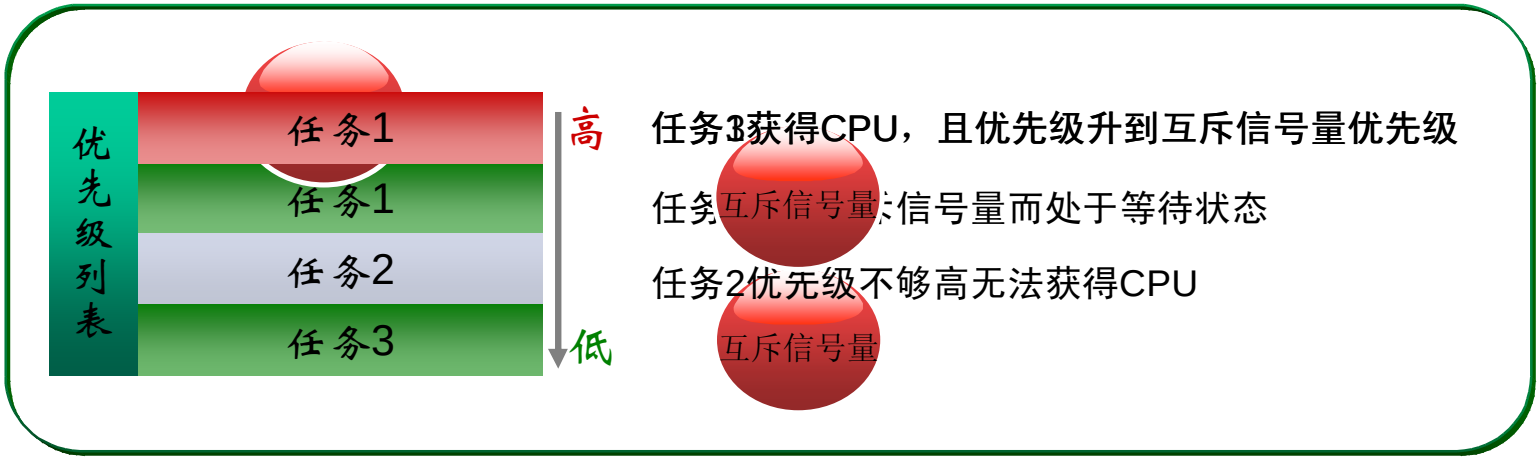


此时，虽然任务1比任务2优先级更高，但却在任务2之后运行，这种现象就是优先级反转。

互斥信号量 | μC/OS-II程序设计基础

简介

假设任务1和任务3共享一个资源，使用互斥信号量进行资源同步，任务2为优先级介于任务1和任务3之间的一个与该共享资源无关任务，通过互斥信号量解决优先级反转问题。

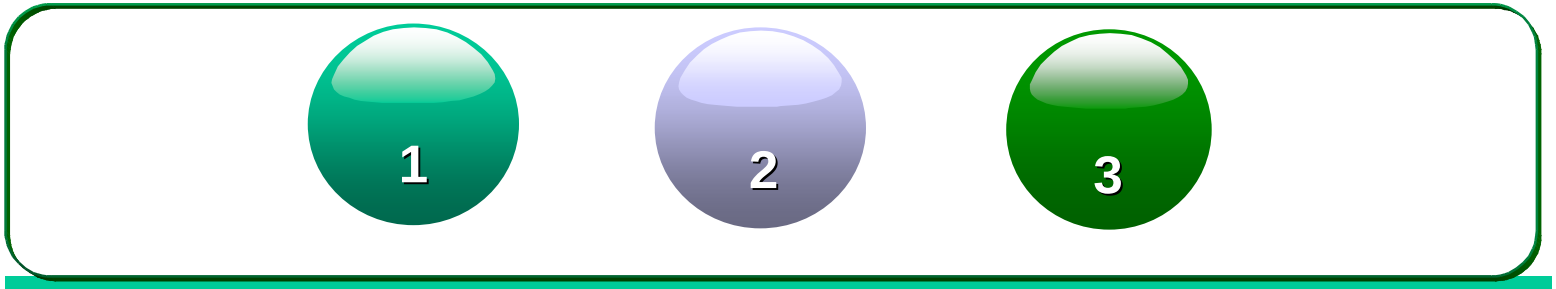


此时，任务2无法在任务1之前得到运行，不发生优先级反转

综上所述，可以说能防止优先级反转现象的信号就是互斥信号量。

简介

使用互斥信号量有以下3点需要注意。



1.在嵌入式系统中，经常使用互斥信号量访问共享资源来实现资源同步。而用来实现资源同步的互斥信号量在创建时初始化，这是由OSMutexCreate ()函数来实现的；

2. OSMutexPost ()发送互斥信号量函数与OSMutexPend ()等待互斥信号量函数必须成对出现在同一个任务调用的函数中，因此我们需要编写一个公共的库函数，因为有多个任务可能调用这个函数；

3. 信号量最好在系统初始化的时候创建，不要在系统运行的过程中动态地创建和删除。在确保成功地创建信号量之后，才可对信号量进行接收和发送操作。

函数列表

互斥信号量函数的6个基本函数如下。

OSMutexAccept函数

函数名称	OSMutexAccept	所属文件	OS_MUTEX.C
函数原型	INT8U OSMutexAccept (OS_EVENT *pevent, INT8U *err)		
功能描述	**查看指定的互斥信号量是否有效**：不同于OSMutexPend()函数，如果互斥信号量无效，则OSMutexAccept ()并不挂起任务		
函数参数	pevent：指向需要查看的消息邮箱的指针，OSSemCreate()的返回值；err：用于返回错误码		
函数返回值	1：有效；0：无效；*err可能为以下值： OS_NO_ERR ： 调用成功 OS_ERR_EVENT_TYPE ： pevent 不是指向互斥信号量的指针 OS_ERR_PEVENT_NULL ： 错误，pevent为NULL OS_ERR_PEND_ISR ： 在中断中调用该函数所引起的错误		

互斥信号量 | μC/OS-II程序设计基础

函数列表

函数名称	OSMutexQuery	所属文件	OS_MUTEX.C
函数原型	INT8U OSMutexQuery (OS_EVENT *pevent, OS_MUTEX_DATA *pdata)		
功能描述	**取得互斥信号量的状态**：用户程序必须分配一个OS_MUTEX_DATA的数据结构，该结构用来从互斥信号量的事件控制块接收数据。通过调用OSMutexQuery ()函数可以知道任务是否有其它任务等待互斥信号量，得到PIP，以及确认互斥信号量是否有效		
函数参数	pevent ： 指向互斥信号量的指针，OSMutexCreate ()的返回值 pdata ： 指向OS_MUTEX_DATA数据结构的指针，该数据结构包含下述成员： OSValue： 0－－互斥信号量无效，1－－互斥信号量有效 OSOwnerPrio： 占用互斥信号量的任务优先级 OSMutexPIP： 互斥信号量的优先级继承优先级(PIP) OSEventTbl[]： 互斥信号量等待队列的拷贝 OSEventGrp： 互斥信号量等待队列索引的拷贝		
函数返回值	OS_NO_ERR ： 调用成功 OS_ERR_EVENT_TYPE ： 错误，pevent 不是指向互斥信号量的指针 OS_ERR_PEVENT_NULL ： 错误，pevent为NULL OS_ERR_POST_ISR ： 在中断中调用该函数所引起的错误		

互斥信号量 | μC/OS-II程序设计基础

函数列表

函数名称	OSMutexDel	所属文件	OS_MUTEX.C
函数原型	OS_EVENT *OSMutexDel (OS_EVENT *pevent, INT8U opt, INT8U *err)		
功能描述	**删除互斥信号量**：在删除互斥信号量之前，应当先删除可能会使用这个互斥信号量的任务		
函数参数	pevent：指向互斥信号量的指针，OSMutexCreate ()的返回值 opt ：定义互斥信号量的删除条件 OS_DEL_NO_PEND：没有任务等待信号量才删除；OS_DEL_ALWAYS：立即删除 err ：用于返回错误码		
函数返回值	NULL：成功删除；pevent：删除失败；*err可能为以下值： OS_NO_ERR：成功删除互斥信号量 OS_ERR_DEL_ISR：在中断中删除互斥信号量所引起的错误 OS_ERR_INVALID_OPT：错误，opt值非法 OS_ERR_TASK_WAITING：有一个或多个任务在等待互斥信号量 OS_ERR_EVENT_TYPE ：错误，pevent不是指向互斥信号量的指针 OS_ERR_PEVENT_NULL：错误，pevent为NULL		
特殊说明	挂起任务就绪时，中断关闭时间与挂起任务数目有关		

互斥信号量 | μC/OS-II程序设计基础

函数列表

函数名称	OSMutexPost	所属文件	OS_MUTEX.C
函数原型	INT8U OSMutexPost (OS_EVENT *pevent)		
功能描述	**发送互斥信号量**：只有任务已调用OSMutexAccept()或OSMutexPend()请求得到互斥信号量时，OSMutexPost()才起作用。如果占用互斥信号量的任务的优先级被提高，OSMutexPost()会恢复其原来的优先级；如果有任务等待互斥信号量，优先级最高的任务将被唤醒；如果没有任务等待互斥信号量，OSMutexPost()把互斥信号量设置为有效状态		
函数参数	pevent ： 指向互斥信号量的指针，OSMutexCreate ()的返回值		
函数返回值	OS_NO_ERR ： 调用成功；OS_ERR_POST_ISR ： 在中断中调用该函数所引起的错误 OS_ERR_EVENT_TYPE ： pevent 不是指向互斥信号量的指针 OS_ERR_PEVENT_NULL ： 错误，pevent为NULL OS_ERR_NOT_MUTEX_OWNER：发送互斥信号量的任务实际上并不占用互斥信号量		

函数列表

函数名称	OSMutexPend	所属文件	OS_MUTEX.C
函数原型	void OSMutexPend (OS_EVENT *pevent, INT16U timeout, INT8U *err)		
功能描述	**等待互斥信号量**：当互斥信号量有效时，则直接返回；如果互斥信号量无效，则等待任务获得互斥信号量后才能解除该等待状态或在超时的情况下运行		
函数参数	pevent ：指向互斥信号量的指针，OSMutexCreate ()的返回值 timeout：超时时间，以时钟节拍为单位；err：用于返回错误码		
函数返回值	*err可能为以下值： OS_NO_ERR ： 调用成功 OS_ERR_EVENT_TYPE ： pevent 不是指向互斥信号量的指针 OS_ERR_PEVENT_NULL ： 错误，pevent为NULL OS_ERR_PEND_ISR ： 在中断中调用该函数所引起的错误 OS_TIMEOUT ： 超过等待时间		

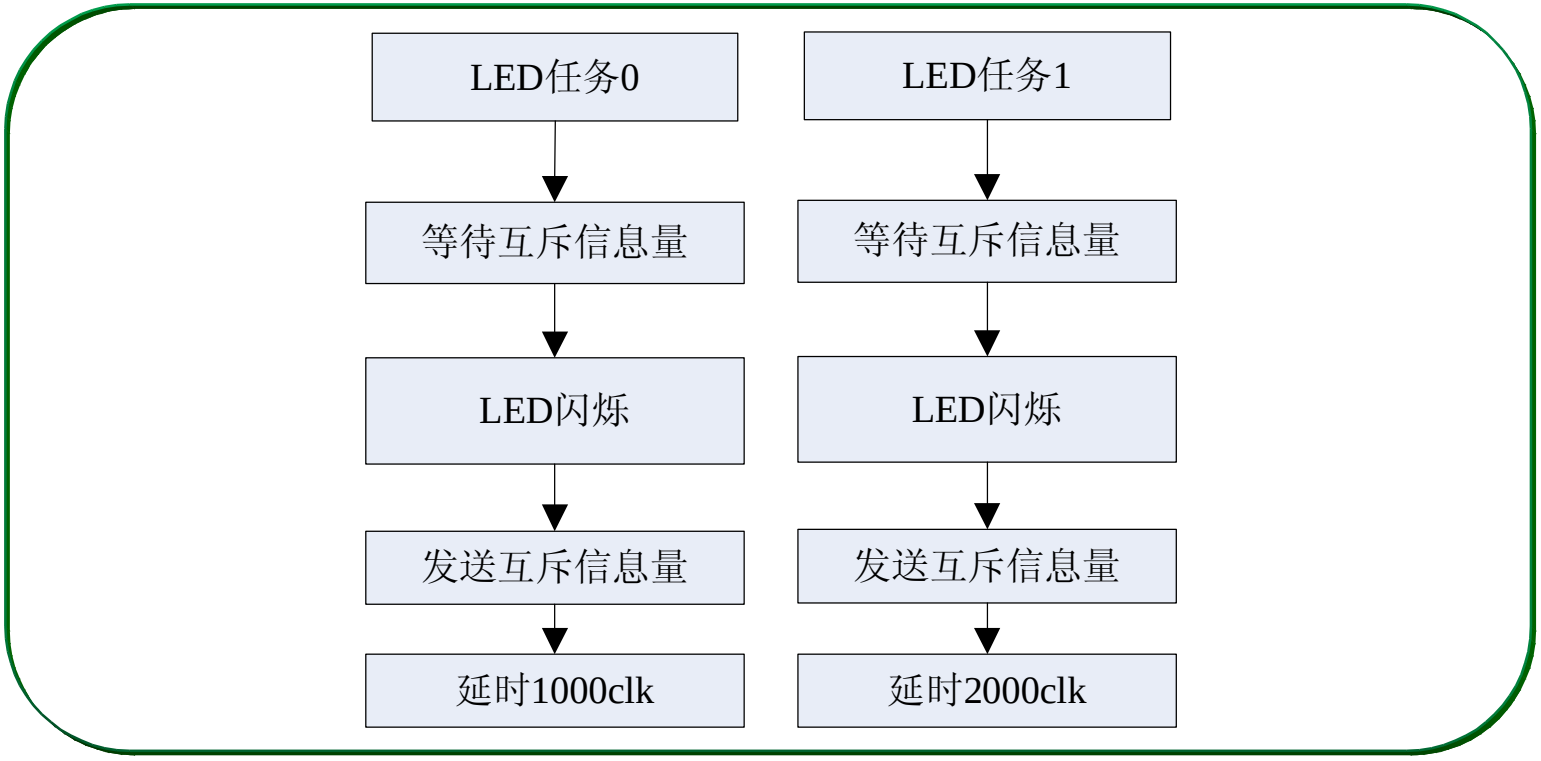
互斥信号量 | μC/OS-II程序设计基础

函数列表

函数名称	OSMutexCreate	所属文件	OS_MUTEX.C
函数原型	OS_EVENT *OSMutexCreate (INT8U prio, INT8U *err)		
功能描述	建立并初始化一个互斥信号量		
函数参数	prio：优先级继承值（PIP）；err：用于返回错误码		
函数返回值	指向分配给所建立的互斥信号量的事件控制块的指针。如果没有可用的事件控制块，则返回空指针。*err可能为以下值： OS_NO_ERR： 成功创建互斥信号量 OS_ERR_CREATE_ISR：在中断中调用该函数所引起的错误 OS_PRIO_INVALID：错误，指定的优先级非法 OS_PRIO_EXIST：错误，指定的优先级已经有任务存在 OS_ERR_PEVENT_NULL：错误，已经没有可用的事件控制块		

资源同步

为了说明使用互斥信号量访问共享资源实现资源同步，假设TaskLED0为高优先级任务，且低于优先级5。设计两个任务，它们以不同的频率让LED点亮30个时钟节拍，然后熄灭60个时钟节拍，要求这两个任务不会互相干扰。下面是两个任务的处理流程。



资源同步

为了实现资源同步，我们需要保证OSMutexPost()与OSMutexPend()成对出现在同一个任务函数中，所以我们编写一个库函数LED()供两个任务调用，代码如下。

```
void  LED (void)
{
    INT8U err;
    OSMutexPend(mutex, 0, &err);
    IO0CLR = LED1;
    OSTimeDly(30);
    IO0SET = LED1;
    OSTimeDly(60);
    OSMutexPost(mutex);
}
```

- 等待互斥信号量
- LED亮
- 延时30个节拍
- LED灭
- 延时60个节拍
- 发送互斥信号量

互斥信号量 | μC/OS-II程序设计基础

资源同步

下面给出两个LED任务的主要处理代码。

```
void  TaskLED0 (void  *pdata)
{
    ……
    mutex = OSMutexCreate (6, &err);
    while (1) {
        LED();
        OSTimeDly(1000);
    }
}
```

初始化工作

创建互斥信号量，分配优先级为6

调用LED函数，函数中已做互斥

延时1000个节拍

```
void  TaskLED1 (void  *pdata)
{
    pdata = pdata;
    while (1) {
        LED();
        OSTimeDly(2000);
    }
}
```

防止编译器报警

调用LED函数，函数中已做互斥

延时2000个节拍

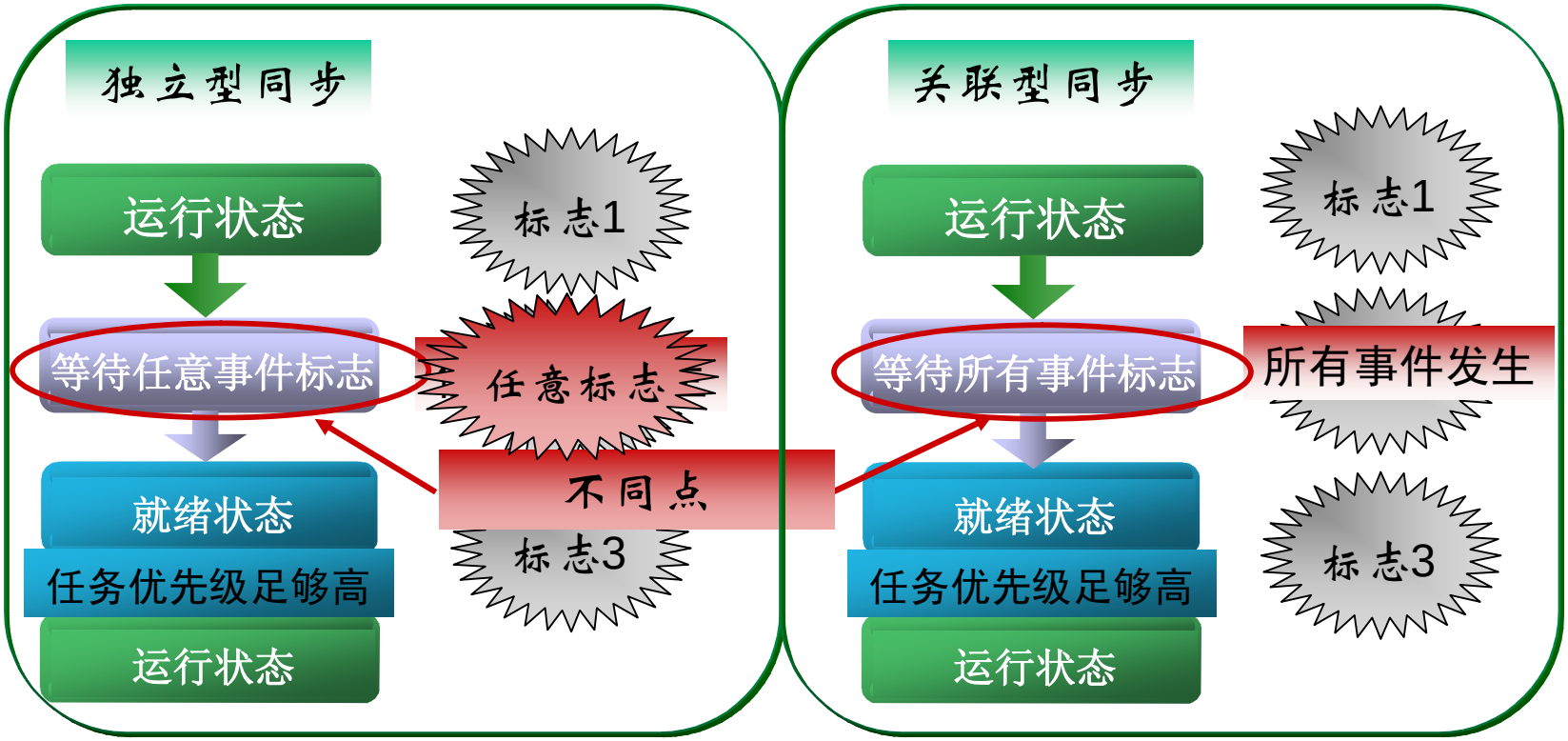
事件标志组

μC/OS-II程序设计基础

事件标志组| μC/OS-II程序设计基础

简介

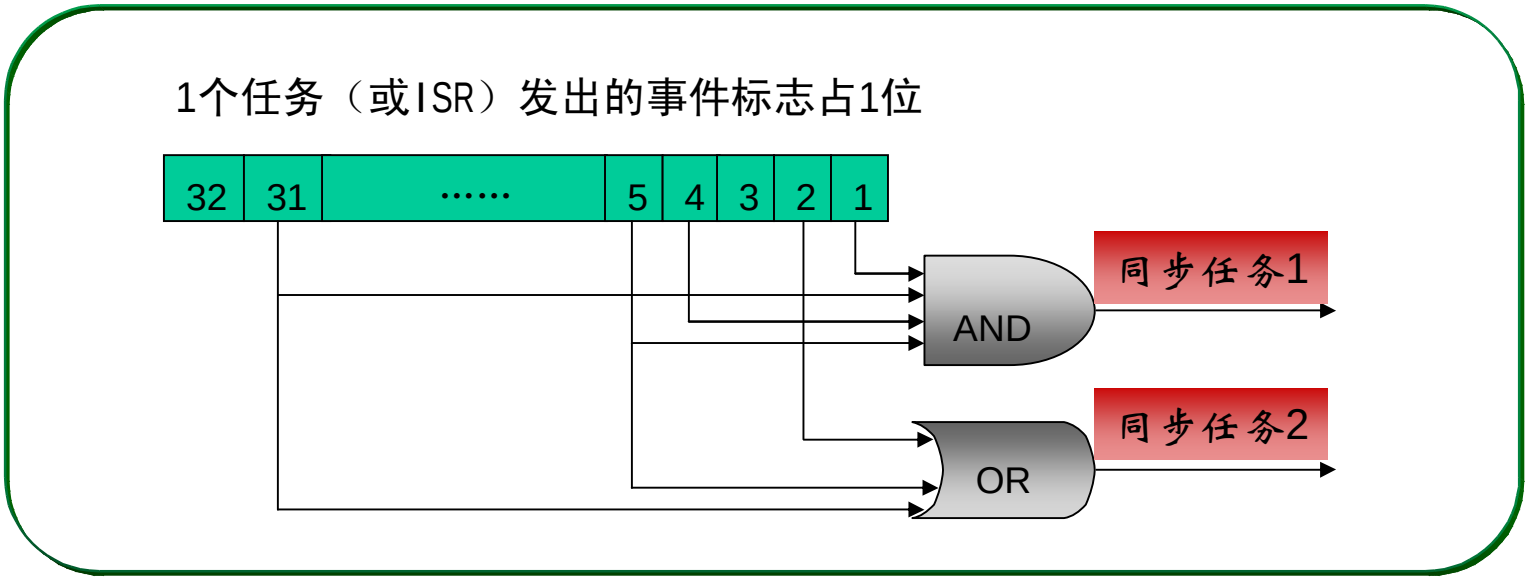
当任务要与多个事件同步时，就要使用事件标志组。一个事件标志就是一个二值信号，事件标志组是若干二值信号的组合。使用事件标志组同步任务分为独立性同步和关联性同步。假设一个任务与3个事件标志有关，如下图。



事件标志组| μC/OS-II程序设计基础

简介

可以用多个事件的组合发信号给多个任务，典型的有8个、16个或32个事件可以组合在一起，取决于用的是哪种内核。



注意：系统在一组新事件发生后判断是否有任务已经接收到需求的标志，任务在收到标志后进行状态切换

事件标志组| μC/OS-II程序设计基础

函数列表

完成事件标志组的各种功能的函数详见下表。

函数名称	OSFlagCreate	所属文件	OS_FLAG.C
函数原型	OS_FLAG_GRP *OSFlagCreate (OS_FLAGS flags, INT8U *err)		
功能描述	建立并初始化一个事件标志组		
函数参数	flags： 事件标志组的初始值 err：用于返回错误码		
函数返回值	指向分配给所建立的事件标志组的指针。如果没有空闲的事件标志组，返回空指针 *err可能为以下值： OS_NO_ERR ：成功创建事件标志组 OS_ERR_CREATE_ISR：在中断中调用该函数所引起的错误 OS_FLAG_GRP_DEPLETED：没有空闲的事件标志组		

事件标志组| μC/OS-II程序设计基础

函数列表

函数参数	pgrp：指向事件标志组的指针，OSFlagCreate()的返回值 flags：指定需要检查的事件标志位，置1，则检查对应位；设置为0，则忽略对应位 wait_type: 定义等待事件标志位的方式 OS_FLAG_WAIT_CLR_ALL：所有指定的事件标志位清0 OS_FLAG_WAIT_CLR_ANY：任意指定的事件标志位清0 OS_FLAG_WAIT_SET_ALL：所有指定的事件标志位置位 OS_FLAG_WAIT_SET_ANY：任意指定的事件标志位置位 OS_FLAG_CONSUME：得到期望的标志位后，恢复相应的标志位。它必须与前面4种方式或，例如：OS_FLAG_WAIT_SET_ANY| OS_FLAG_CONSUME表示等待任意指定的事件标志位置位，并且在任意标志位置位后清除该位 Err：用于返回错误码

事件标志组| μC/OS-II程序设计基础

函数列表

函数名称	OSFlagDel	所属文件	OS_FLAG.C
函数原型	OS_FLAG_GRP *OSFlagDel (OS_FLAG_GRP *pgrp, INT8U opt, INT8U *err)		
功能描述	**删除事件标志组**：在删除事件标志组之前，应当先删除可能会使用它的任务		
函数参数	pgrp：指向事件标志组的指针，OSFlagCreate()的返回值 opt ： 定义事件标志组删除条件 OS_DEL_NO_PEND：没有任何任务等待事件标志组时才删除 OS_DEL_ALWAYS：立即删除 err ： 用于返回错误码		
函数返回值	NULL：成功删除；pgrp：删除失败；*err可能为以下值： OS_NO_ERR ： 成功删除事件标志组 OS_ERR_DEL_ISR ： 在中断中删除事件标志组所引起的错误 OS_ERR_INVALID_OPT ： 错误，opt值非法 OS_ERR_TASK_WAITING ： 有一个或多个任务在事件标志组 OS_ERR_EVENT_TYPE ： 错误，pgrp不是指向事件标志组的指针 OS_FLAG_INVALID_PGRP ： 错误，pgrp 为NULL		
特殊说明	当挂起任务就绪时，中断关闭时间与挂起任务数目有关		

函数名称	OSFlagPost	所属文件	OS_FLAG.C
函数原型	OS_FLAGS OSFlagPost (OS_FLAG_GRP *pgrp, OS_FLAGS flags, INT8U opt, INT8U *err)		
功能描述	**设置事件标志位**：可以置位或清0指定标志位。如果设置标志位后正好满足某个等待该事件标志组的任务，该任务将切换到就绪态		
函数参数	pgrp：指向事件标志组的指针，OSFlagCreate()的返回值 Flags：指定需要设置的标志位，置位的位将被设置 opt：设置方式： OS_FLAG_CLR：清0指定标志位；OS_FLAG_SET：置位标志位 err：用于返回错误码		
函数返回值	事件标志组新的事件标志状态，*err可能为以下值： OS_NO_ERR ： 成功调用 OS_FLAG_INVALID_PGRP ： 错误，pgrp为NULL OS_ERR_EVENT_TYPE ： 错误，pgrp不是指向事件标志组 OS_ERR_INVALID_OPT ： 错误，opt值非法		

事件标志组| μC/OS-II程序设计基础

函数列表

函数名称	OSFlagPend	所属文件	OS_FLAG.C
函数原型	OS_FLAGS OSFlagPend (OS_FLAG_GRP *pgrp, OS_FLAGS flags, INT8U wait_type, INT16U timeout, INT8U *err)		
功能描述	**等待事件标志组中的指定事件标志是否置位(或清0)：**应用程序可以检查任意一位(或多位)是置位还是清0，如果所需的事件标志没有产生，任务挂起，直到所需条件满足		
函数返回值	返回事件标志组的状态；超时返回0；*err可能为以下值： OS_NO_ERR ： 成功调用 OS_ERR_PEND_ISR：在中断中调用该函数所引起的错误 OS_ERR_EVENT_TYPE ： 错误，pgrp不是指向事件标志组 OS_FLAG_ERR_WAIT_TYPE ： 错误，wait_type值非法 OS_FLAG_INVALID_PGRP ： 错误，pgrp为NULL OS_TIMEOUT：超时		

事件标志组| μC/OS-II程序设计基础

函数列表

函数名称	OSFlagQuery	所属文件	OS_FLAG.C
函数原型	OS_FLAGS OSFlagQuery (OS_FLAG_GRP *pgrp, INT8U *err)		
功能描述	**用于取得事件标志组的状态：**当前版本还不能返回等待事件标志组的任务列表		
函数参数	pgrp：指向事件标志组的指针，OSFlagCreate()的返回值；err ：用于返回错误码		
函数返回值	当前事件标志组的状态，*err可能为以下值： OS_NO_ERR ： 调用成功；OS_FLAG_INVALID_PGRP ： 错误，pgrp为NULL OS_ERR_EVENT_TYPE ： 错误，pgrp不是指向事件标志组		

事件标志组| μC/OS-II程序设计基础

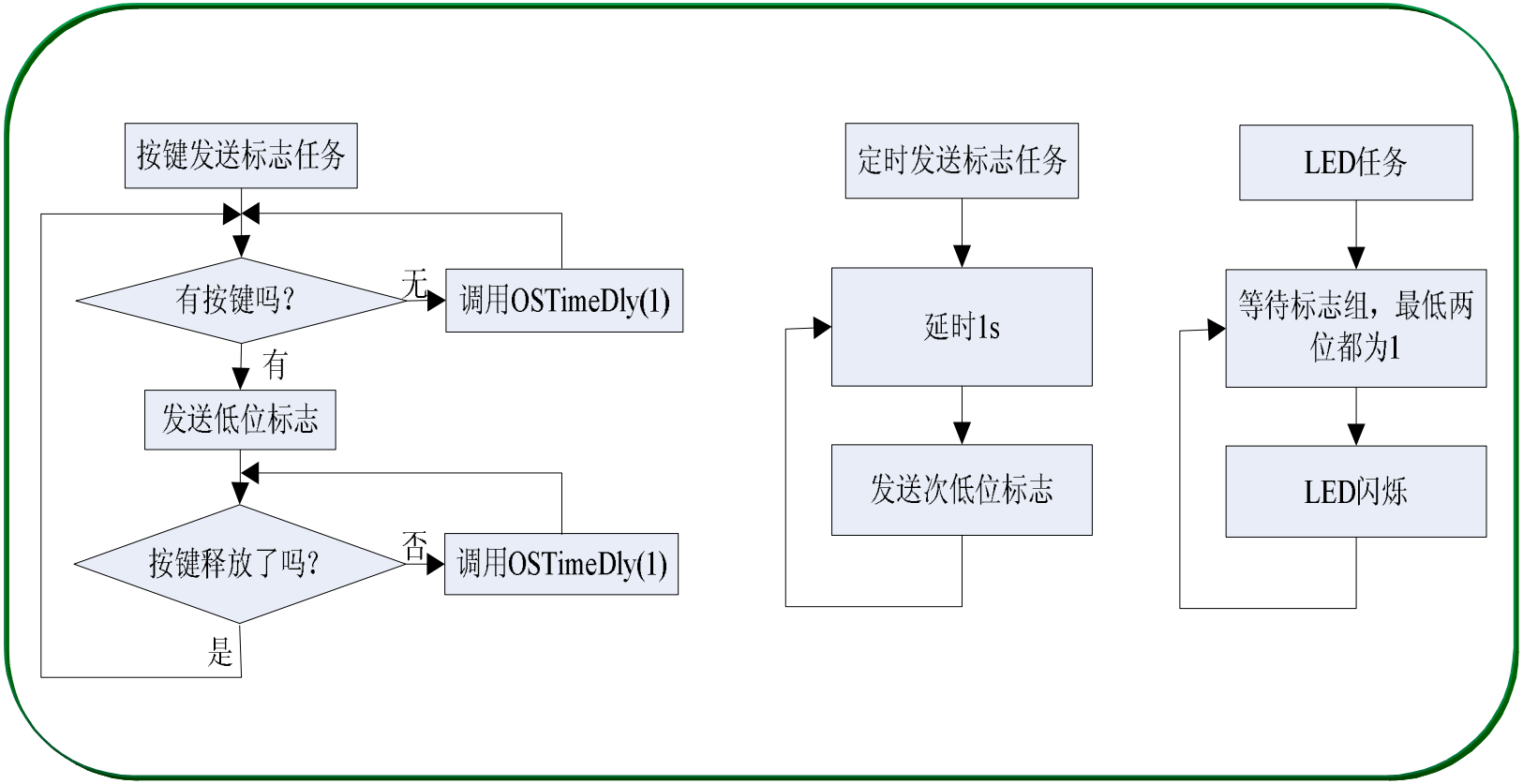
函数列表

函数名称	OSFlagAccept	所属文件	OS_FLAG.C
函数原型	OS_FLAGS OSFlagAccept (OS_FLAG_GRP *pgrp, OS_FLAGS flags, INT8U wait_type, INT8U *err)		
功能描述	**无等待地获得事件标志组中的指定事件标志是否置位（或清0）：** 应用程序可以检查任意一位(或多位)是置位还是清0，它与OSFlagPend()的区别是，如果所需的事件标志没有产生，任务并不挂起		
函数返回值	返回事件标志组的状态，*err可能为以下值： OS_NO_ERR ： 成功调用 OS_ERR_EVENT_TYPE ： 错误，pgrp不是指向事件标志组 OS_FLAG_ERR_WAIT_TYPE ： 错误，wait_type值非法 OS_FLAG_INVALID_PGRP ： 错误，pgrp为NULL OS_FLAG_ERR_NOT_RD ： 指定的事件标志没有发生		

事件标志组| μC/OS-II程序设计基础

标志与

为了说明如何使用标志事件组实现任务与若干个事件同步，我们设计一个系统，当时间到且独立按健被按下过，让LED1闪耀一下。假设TaskLED为高优先级任务，三个任务的处理流程如图。



事件标志组| μC/OS-II程序设计基础

标志与

TaskKEY任务主要代码如下。

```
void  TaskKEY (void  *pdata)
{
    ……
    while (1) {
        while ((IO0PIN & KEY1) != 0) {
            OSTimeDly(1);
        }
        OSFlagPost(flag, KEY1FLAG,
        OS_FLAG_SET, &err);
        while ((IO0PIN & KEY1) == 0) {
            OSTimeDly(1);
        }
    }
}
```

初始化代码

等待按健按下

延时1个节拍，用于任务切换

发送标志，最低位，1有效

等待按健释放

延时1个节拍，用于任务切换

标志与

TaskDly任务主要代码如下。

```
void  TaskDly (void  *pdata)
{
    ……
    while (1) {
        OSTimeDly(OS_TICKS_PER_SEC);
        OSFlagPost(flag, DLY1FLAG,
        OS_FLAG_SET, &err);
    }
}
```

初始化工作

延时1s

发送标志，次低位，1有效

事件标志组| μC/OS-II程序设计基础

标志与

TaskLED任务主要代码如下。

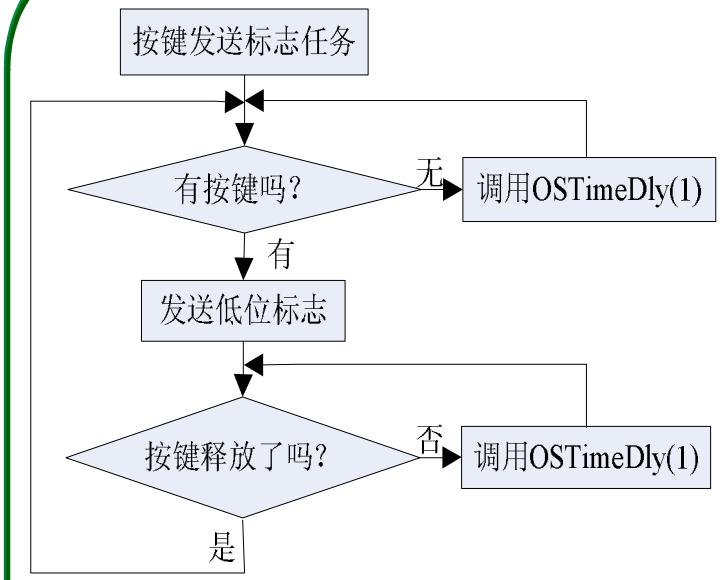
```
void  TaskLED (void  *pdata)
{
    ……
    flag = OSFlagCreate(0, &err);
    while (1) {
        OSFlagPend (flag, KEY1FLAG | DLY1FLAG ,
        OS_FLAG_WAIT_SET_ALL |
        OS_FLAG_CONSUME , 0, &err);
        IO0CLR = LED1;
        OSTimeDly(OS_TICKS_PER_SEC);
        IO0SET = LED1;
        OSTimeDly(OS_TICKS_PER_SEC);
    }
}
```

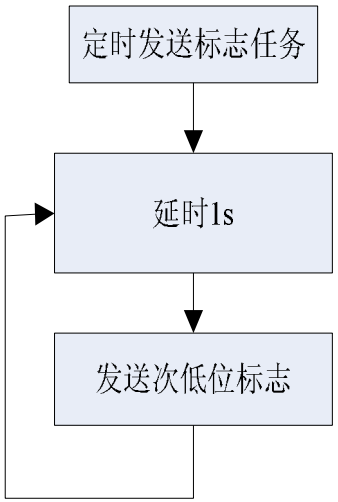
- 初始化工作
- 创建事件标志组
- 等待标志组，最低两位全为1 复位标志，一直等待
- LED亮
- 延时1s
- LED灭
- 延时1s

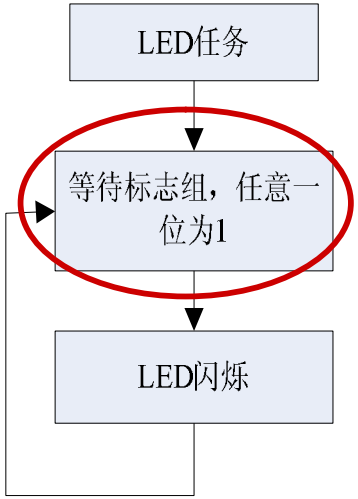
事件标志组| μC/OS-II程序设计基础

标志或

为了说明如何使用标志事件组实现任务与任何事件之一同步，我们设计一个系统，当时间到或独立按健被按下过，让LED1闪耀一下。假设TaskLED为高优先级任务，三个任务的处理流程如图。







注意：标志或和标志与操作只在等待标志的LED任务有所不同，其余都相同。

事件标志组| μC/OS-II程序设计基础

标志或

由于发送标志的两个任务代码和标志与操作的相同，这里不做重复。标志或中的 TaskLED任务主要代码如下。

```
void  TaskLED (void  *pdata)
{
    ……
    flag = OSFlagCreate(0, &err);
    while (1) {
        OSFlagPend (flag, KEY1FLAG | DLY1FLAG ,
        OS_FLAG_WAIT_SET_ANY |
        OS_FLAG_CONSUME , 0, &err);
        IO0CLR = LED1;
        OSTimeDly(OS_TICKS_PER_SEC);
        IO0SET = LED1;
        OSTimeDly(OS_TICKS_PER_SEC);
    }
}
```

初始化工作

创建事件标志组

等待标志组，最低两位任意一位为1,复位标志，一直等待

LED亮

延时1s

LED灭

延时1s

信号量

µC/OS-II程序设计基础

信号量 | μC/OS-II程序设计基础

简介

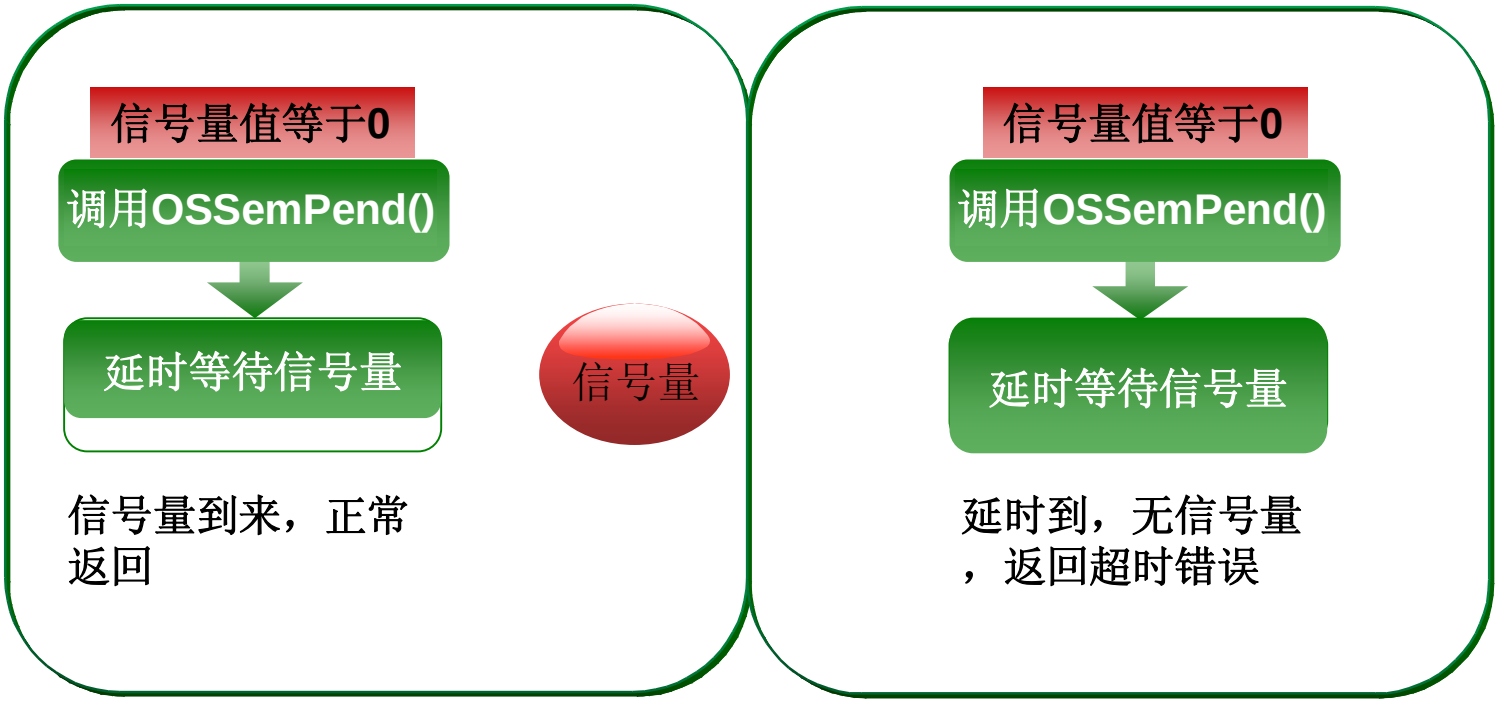
在实时多任务系统中，信号量被广泛用于：任务间对共享资源的互斥、任务和中断服务程序之间的同步、任务之间的同步。

当任务调用OSSemPost()函数发送信号量时；

当信号量值大于0，任务调用OSSemPend()函数接收信号量时；

简介

当信号量值等于0，任务调用OSSemPend()函数接收信号量时。



注意： μC/OS-II不允许在中断服务程序中等待信号量。

简介

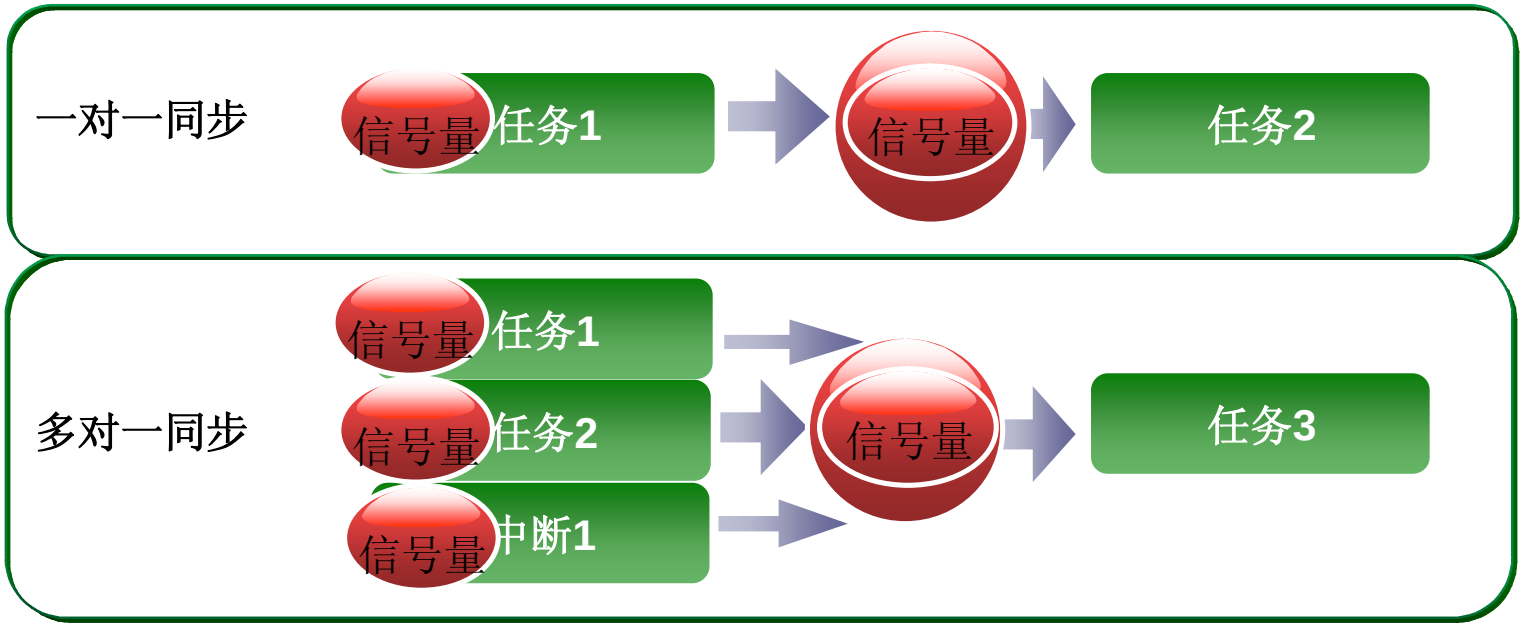
前面章节我们学习了互斥信号量，下面对计数信号量与互斥信号量做一个对比。

	取值	能否用于资源同步以实现对共享资源的独占	能否解决优先级反转问题	能否实现任务间以及中断与任务间的同步	实现同步时能否传递数据
互斥信号量	0或1	能	能	不能	不能
计数信号量	0～65535	能	不能	能	不能

信号量的工作方式

1.任务间同步

在实际的应用中，常用信号量实现任务间的同步，OSSemPend()和OSSemPost()会出现在不同任务的不同函数中，但不一定成对出现。



注意：在实际的应用中，还有多对多、一对多信号量操作的情况，但很不常见，建议读者不要设计出这样的操作方式，因为这样会带来很多的麻烦。

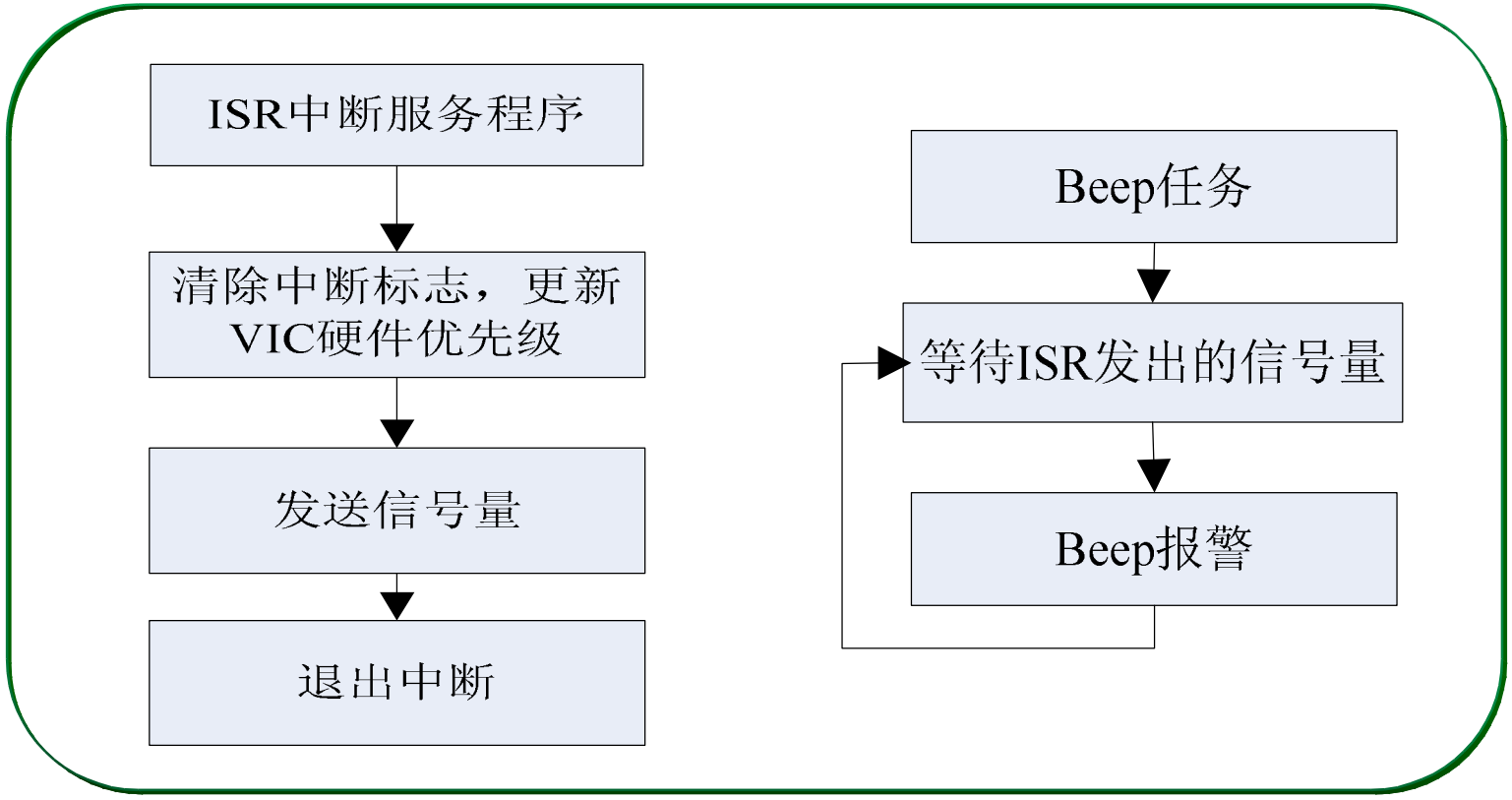
信号量的工作方式

2.资源同步

在嵌入式系统中，经常使用信号量访问共享资源来实现资源同步。在使用时，注意发送信号量函数OSSemPost()与等待信号量函数OSSemPend()必须成对出现在同一个任务调用的函数中，才能实现资源同步。

ISR与任务同步

下面以示例来说明如何现实ISR与任务间同步。假设定时器1中断服务程序发送信号量，任务完成了信号量的创建并在接收到信号量后让蜂鸣器响一声。处理流程如下。



ISR与任务同步

中断服务程序ISR示例代码如下。

```
void  Timer1_Exception (void)
{
    T1IR = 0x01;
    VICVectAddr = 0;
    OSSemPost (sem);
}
```

- 清除中断标志
- 更新中断优先级
- 发送信号量

ISR与任务同步

蜂鸣器报警任务示例代码如下。

```
void  TaskBeep (void  *pdata)
{
    ……
    sem = OSSemCreate(0);
    OS_ENTER_CRITICAL();
    /* 初始化VIC(省略)                    */
    /* 初始化定时器1(省略)                */
    /* 目标板初始化TargeInit() (省略) */
    OS_EXIT_CRITICAL();
    while (1) {
        OSSemPend(sem, 0, &err);
        IO0CLR = BEEP;
        OSTimeDly(60);
        IO0SET = BEEP;
        OSTimeDly(60);
    }
}
```

初始化工作

创建信号量

进入临界段

读者补出相应的代码

退出临界段

等待信号量

蜂鸣器响

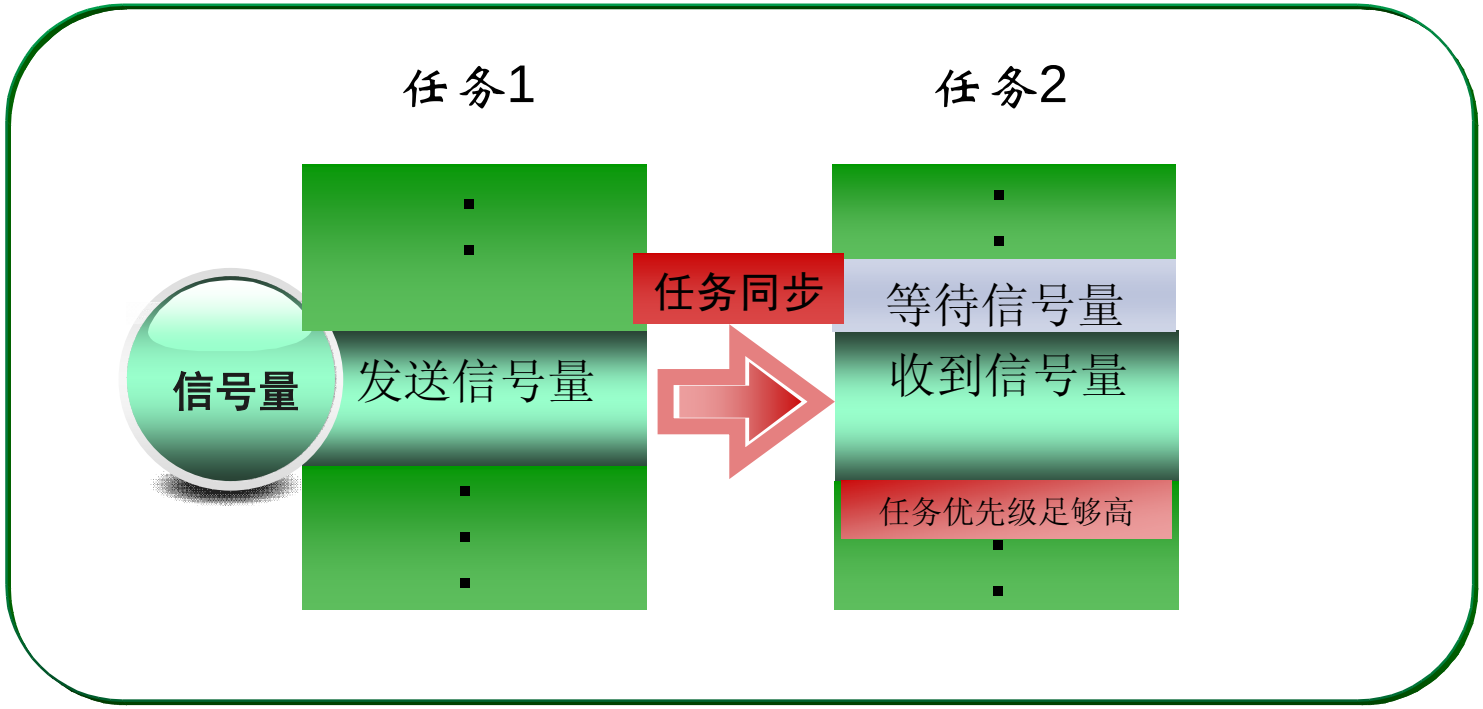
延时60个节拍

蜂鸣器灭

延时60个节拍

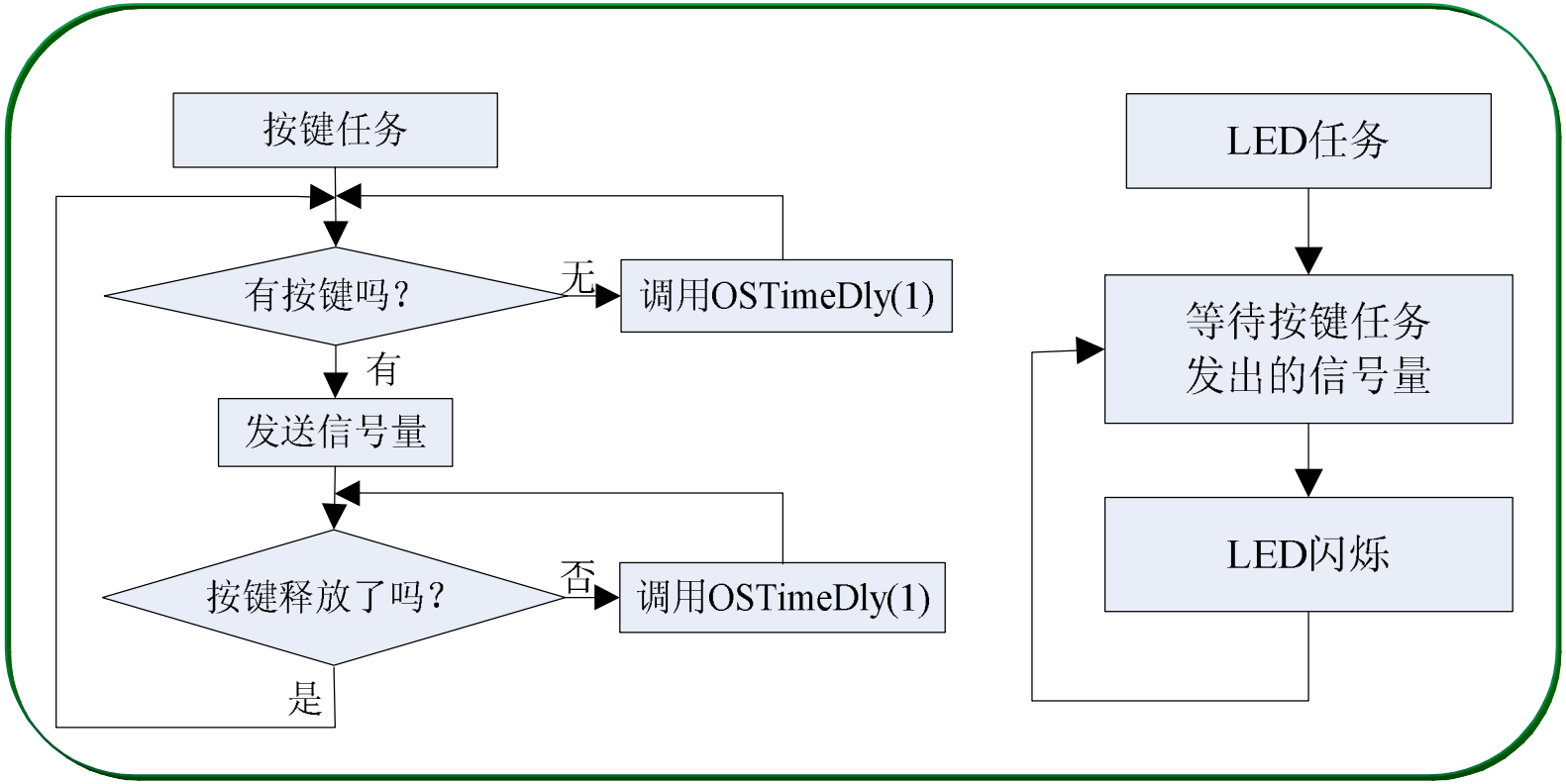
任务间同步

在嵌入式系统中，经常使用信号量来实现多个任务之间的同步。而用来实现任务间同步的信号量在创建时初始值可以为0或者1，这是由OSSemCreate()函数来实现的。



任务间同步

让一个LED以0.5Hz的频率闪耀，每按键一次，LED闪耀一次。我们通过此例来说明如何使用信号量实现任务间同步，假设TaskLED为高优先级的任务。两个任务处理流程如下。



任务间同步

TaskKEY任务主要代码如下。

```
void  TaskKEY (void  *pdata)
{
    ……
    while (1) {
        while ((IO0PIN & KEY1) != 0) {
            OSTimeDly(1);
        }
        OSSemPost (sem);
        while ((IO0PIN & KEY1) == 0) {
            OSTimeDly(1);
        }
    }
}
```

初始化代码

等待按健按下

延时1个节拍，用于任务切换

发送信号量

等待按健释放

延时1个节拍，用于任务切换

任务间同步

TaskLED任务主要代码如下。

```
void  TaskLED (void  *pdata)
{
    ……
    sem = OSSemCreate(0);
    while (1) {
        OSSemPend(sem, 0, &err);
        IO0CLR = LED1;
        OSTimeDly(OS_TICKS_PER_SEC);
        IO0SET = LED1;
        OSTimeDly(OS_TICKS_PER_SEC);
    }
}
```

初始化工作

创建信号量，初始化为0

等待消息

LED亮

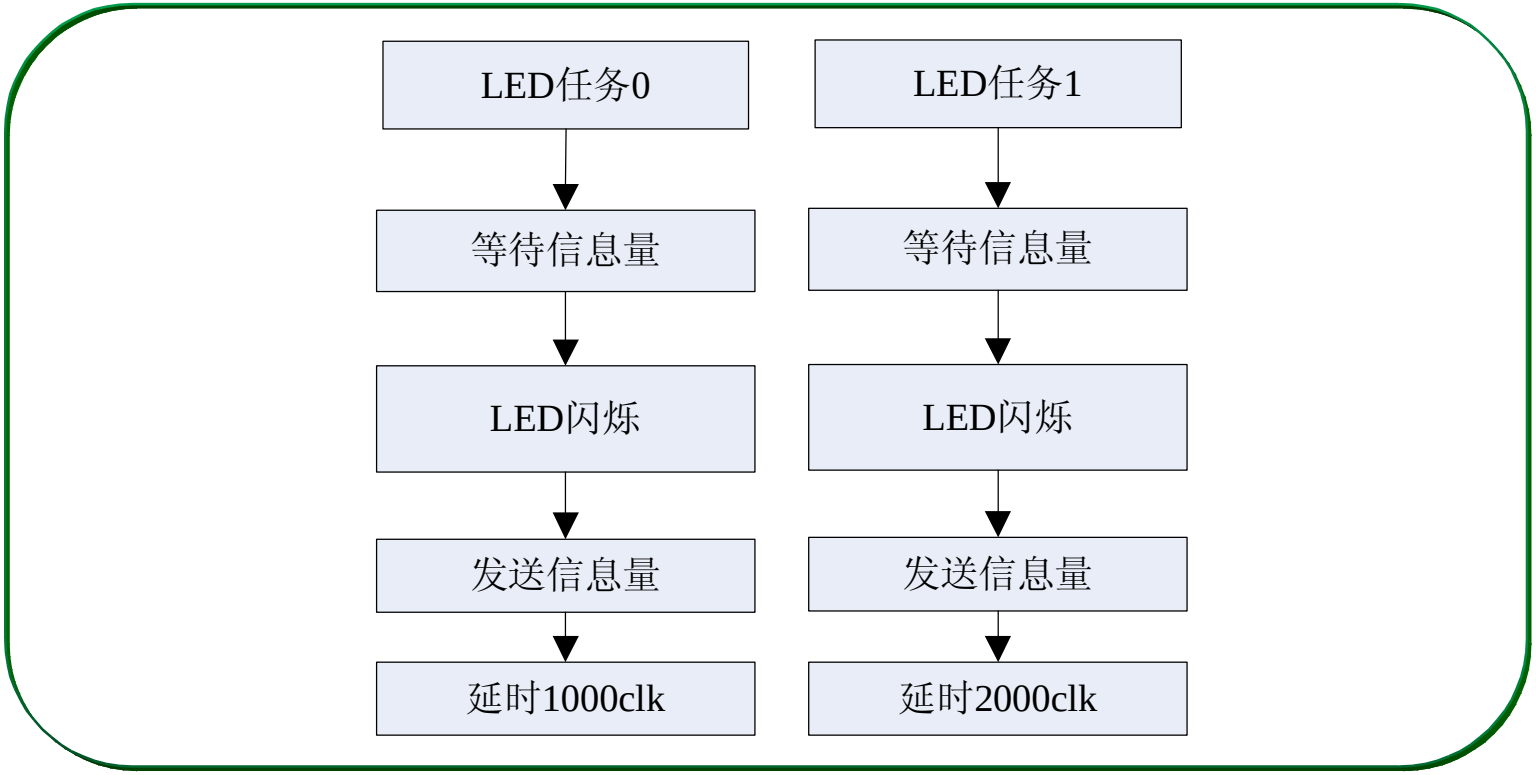
延时1s

LED灭

延时1s

资源同步

为了说明使用信号量访问共享资源实现资源同步，设计两个任务，它们以不同的频率让LED点亮30个时钟节拍，然后熄灭60个时钟节拍，要求这两个任务不会互相干扰。假设TaskLED0为高优先级任务，下面是两个任务的处理流程。



资源同步

为了实现资源同步，我们需要保证OSSemPost()与OSSemPend()成对在同一个任务函数中调用，所以我们编写一个库函数LED()供两个任务调用，代码如下。

```
void  LED (void)
{
    INT8U err;
    OSSemPend(sem, 0, &err);
    IO0CLR = LED1;
    OSTimeDly(30);
    IO0SET = LED1;
    OSTimeDly(60);
    OSSemPost(sem);
}
```

- 等待信号量
- LED亮
- 延时30个节拍
- LED灭
- 延时60个节拍
- 发送信号量

资源同步

下面给出两个LED任务的主要处理代码。

```
void  TaskLED0 (void  *pdata)
{
    ……
    sem = OSSemCreate(1);
    while (1) {
        LED();
        OSTimeDly(1000);
    }
}
```

初始化工作

创建信号量

调用LED函数

延时1000个节拍

用来实现资源同步的信号量在创建时初始值为相同资源的数目，不过嵌入式系统中极少出现完全等同的资源，所以一般初始化为1。

```
void  TaskLED1 (void  *pdata)
{
    pdata = pdata;
    while (1) {
        LED();
        OSTimeDly(2000);
    }
}
```

防止编译器报警

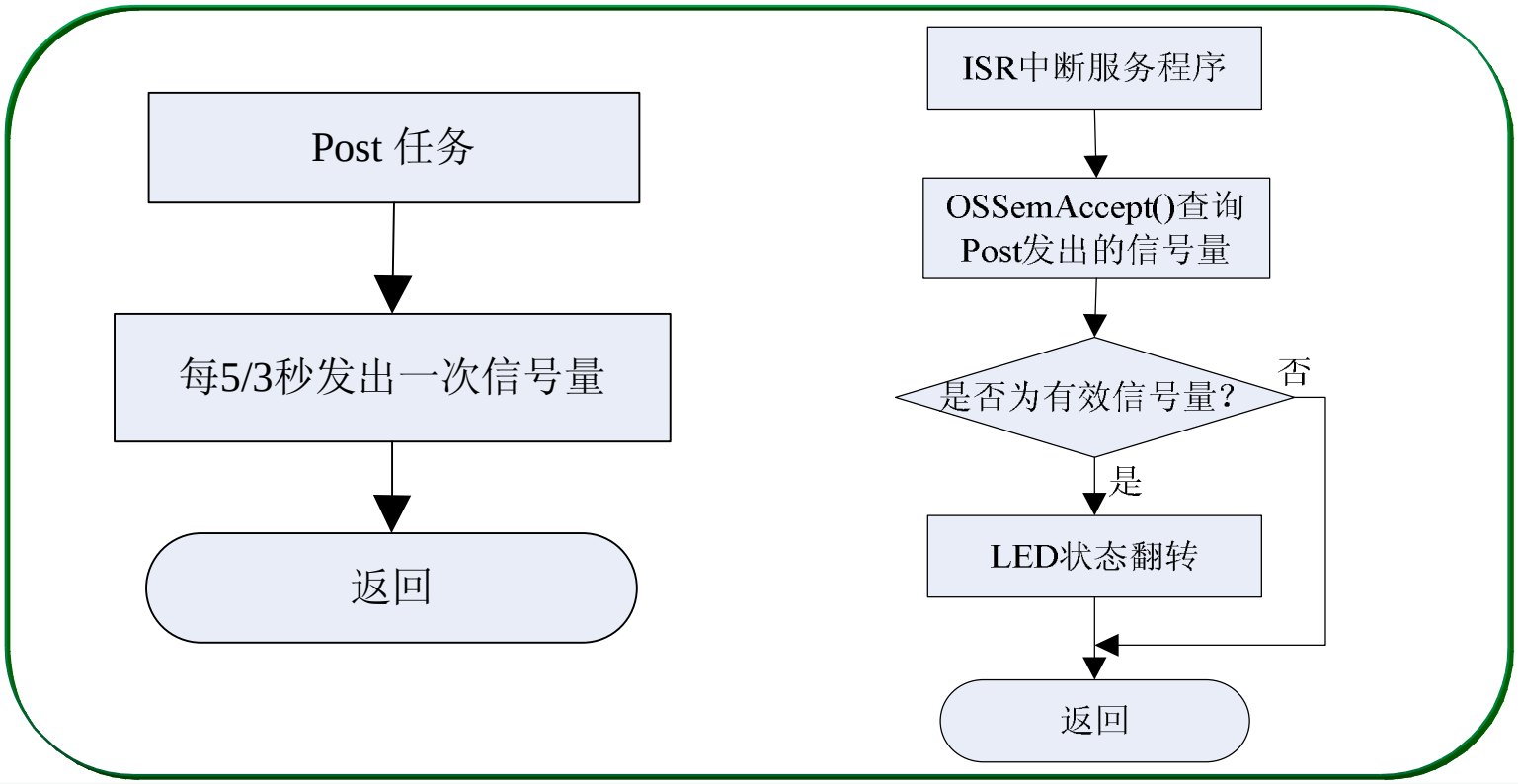
调用LED函数

延时2000个节拍

LED()函数已包含互斥

在中断中获得信号量

建立一个任务，它每5／3 秒发送一次信号量。定时器1每1秒钟产生一次中断，在中断服务程序中获得信号量，如果有，则翻转LED。以此示例来说明如何在中断中获得信号量，使用函数OSSemAccept()实现。两个任务处理流程如下。



在中断中获得信号量

发送信号量任务主要代码如下。

```
void TaskPost (void  *pdata)
{
    ……
    sem = OSSemCreate(0);
    OS_ENTER_CRITICAL();
    /* 初始化VIC(省略)                    */
    /* 初始化定时器1(省略)                */
    /* 目标板初始化TargeInit() (省略)     */
    OS_EXIT_CRITICAL();
    while (1) {
        OSTimeDly(OS_TICKS_PER_SEC * 5/3);
        OSSemPost (sem);
    }
}
```

初始化工作

创建信号量

进入临界段

读者补出相应的代码

退出临界段

延时5/3秒

发送信号量

在中断中获得信号量

中断服务程序ISR代码如下。

```
void  Timer1_Exception (void)
{
    T1IR = 0x01;
    VICVectAddr = 0;
    if (OSSemAccept (sem) > 0) {
        if (IO0PIN & LED1) {
            IO0CLR = LED1;
        }
        else {
            IO0SET = LED1;
        }
    }
}
```

清除中断标志

更新中断优先级

无等待地请求一个信号量

LED灭，点亮LED

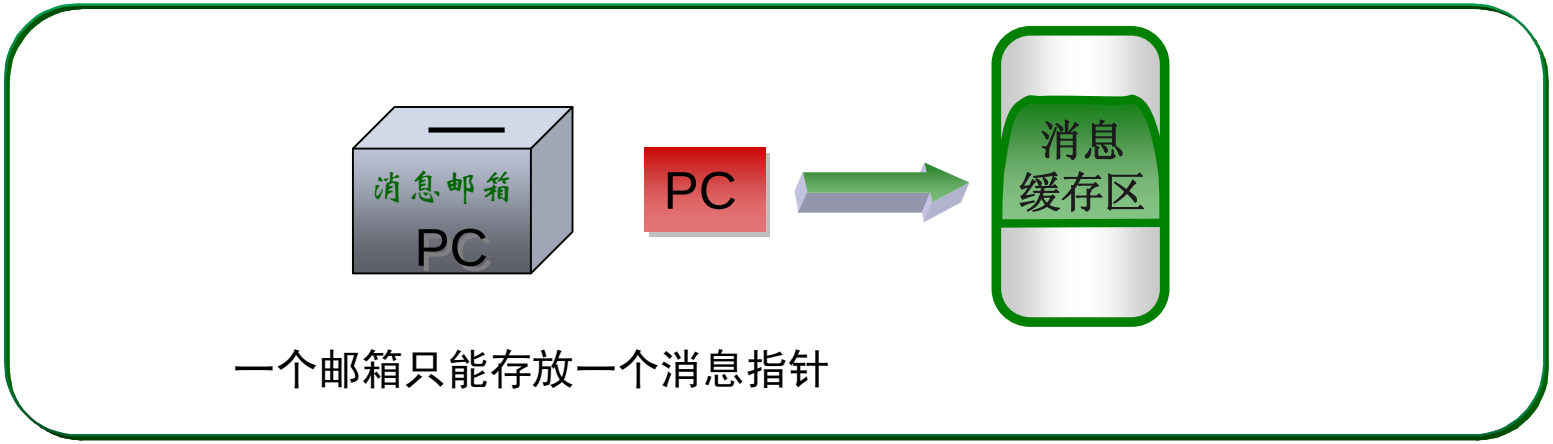
LED亮，熄灭LED

消息邮箱

μC/OS-II程序设计基础

简介

消息是任务之间的一种通信手段，当同步过程需要传输具体内容时就不能使用信号量，此时可以选择消息邮箱，即通过内核服务可以给任务发送带具体内容的消息。



通过上述图解可知，用来传递消息缓冲区指针的数据结构就是消息邮箱。

消息邮箱 | μC/OS-II程序设计基础

简介

1.向消息邮箱发送消息

有任务在等待消息

发送消息指针 PC

等待列表中最高优先级的任务

获得消息

就绪状态

任务优先级足够高

运行状态

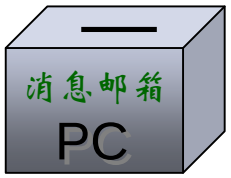
注意：如果发送消息指针是以广播的形式发送，那么所有等待此消息的任务都获得消息

无任务在等待消息

发送消息指针 PC

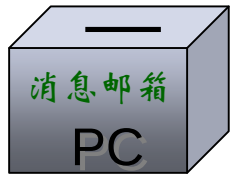
发送失败

空



操作成功

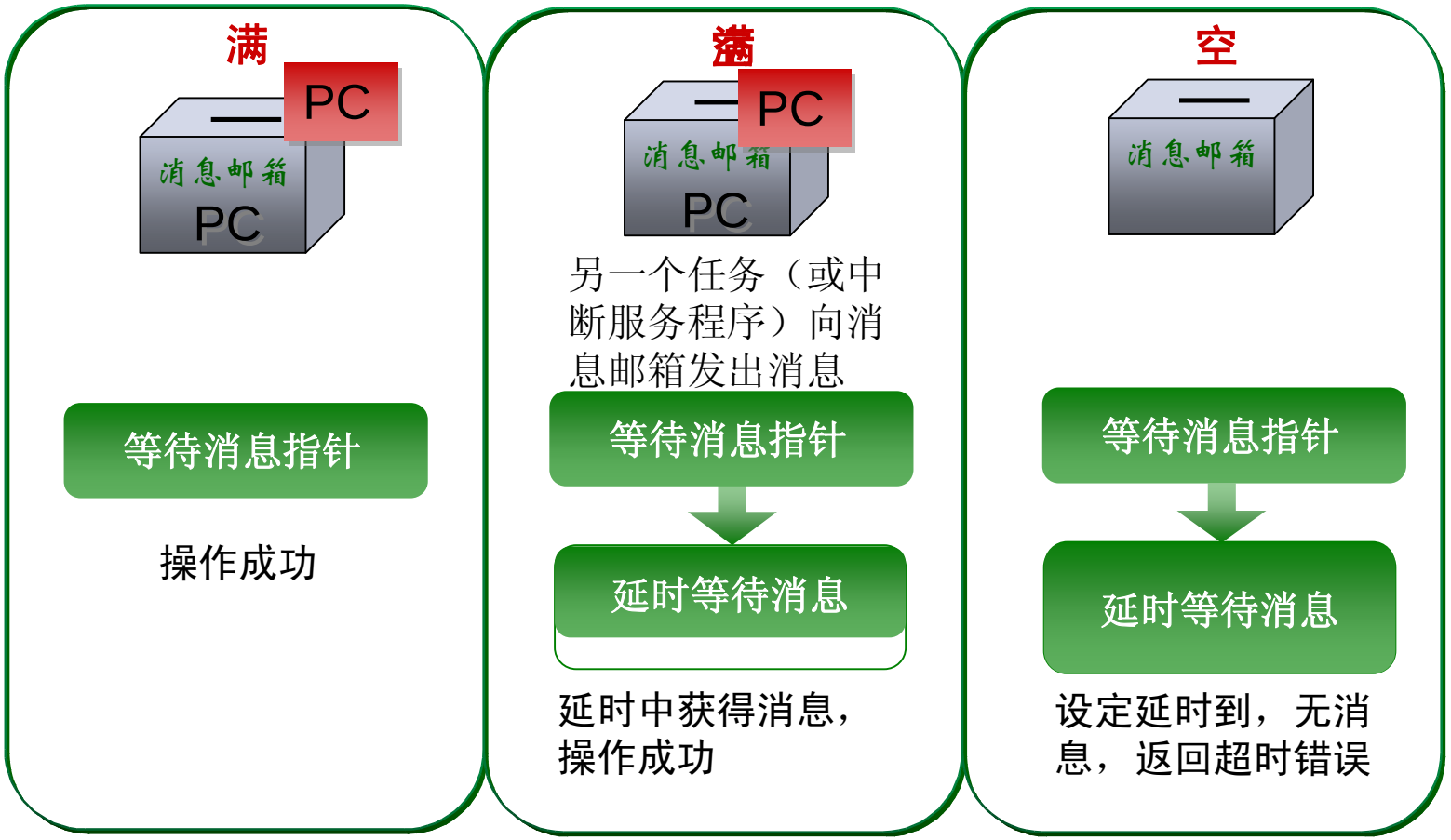
满



返回错误码说明消息邮箱已满

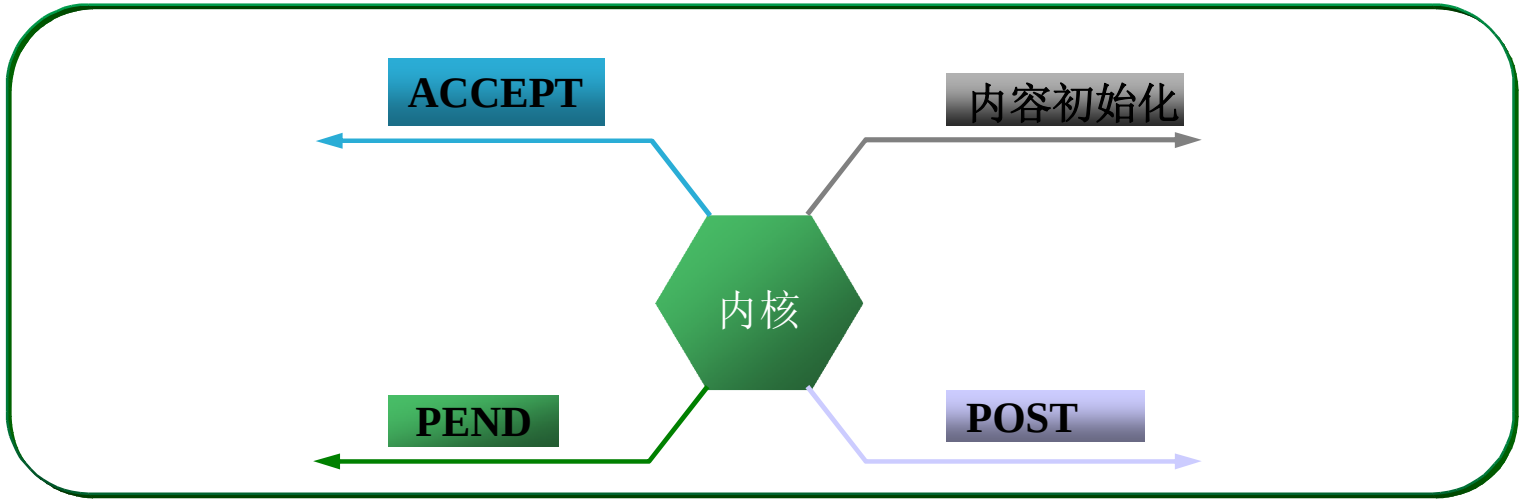
简介

2.从消息邮箱接收消息



简介

内核一般提供以下邮箱服务：



消息邮箱与信号量最大的区别：消息邮箱可以存放一条完整的内容信息，而用信号量进行行为同步时，只能提供同步时刻的信息，不能提供内容信息。

简介

内核一般提供以下邮箱服务：

邮箱内消息的内容初始化，此时邮箱内是否有消息并不重要；

将消息放入邮箱(POST)；

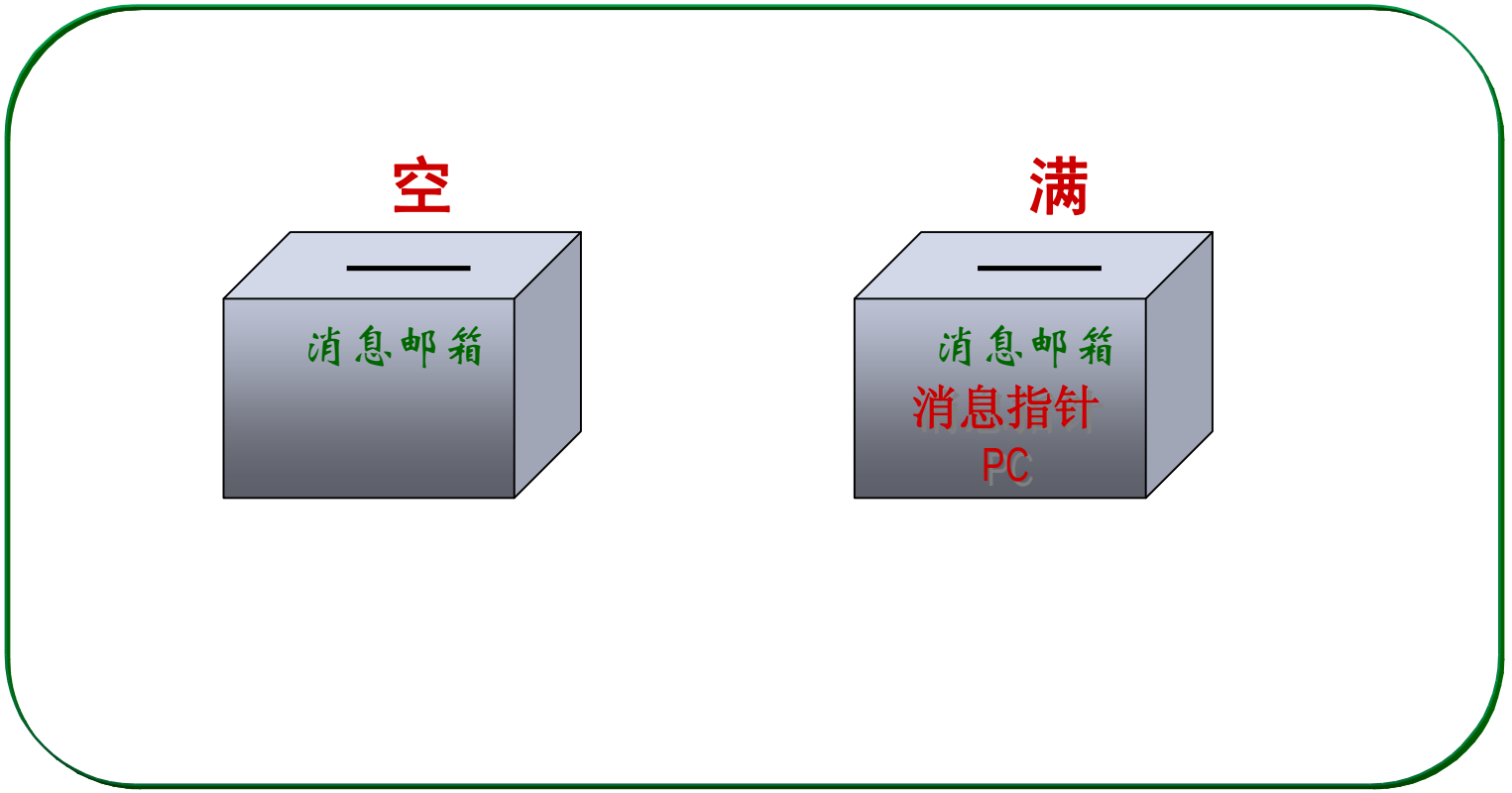
等待有消息进入邮箱(PEND)；

如果邮箱内有消息，则任务将消息从邮箱中取走；如果邮箱内没有消息，则内核不将该任务挂起(ACCEPT)，返回空指针。

消息邮箱与信号量最大的区别：消息邮箱可以存放一条完整的内容信息，而用信号量进行行为同步时，只能提供同步时刻的信息，不能提供内容信息。

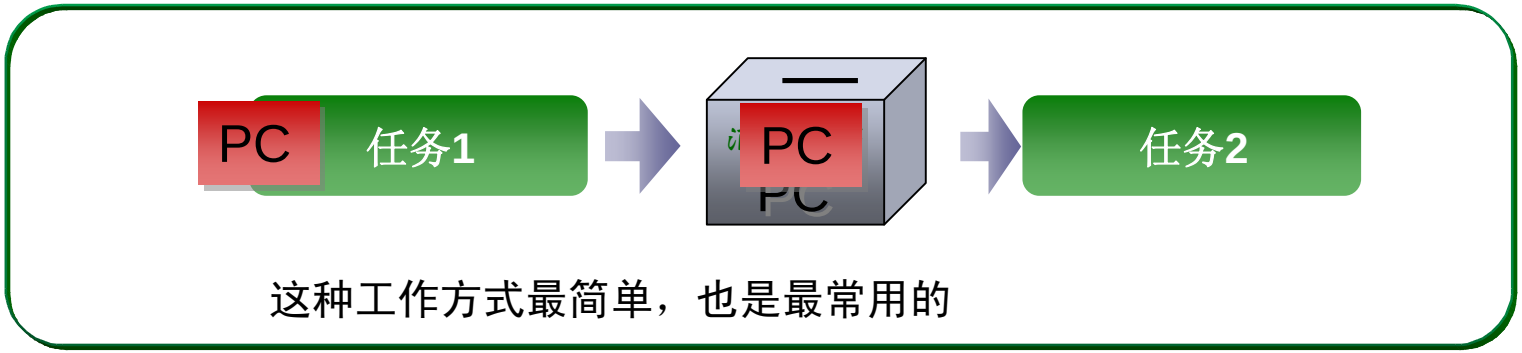
消息邮箱的状态

一般来说，消息邮箱只有2种状态：即空状态（消息邮箱中没有消息）、满状态（消息邮箱中存放了消息）。



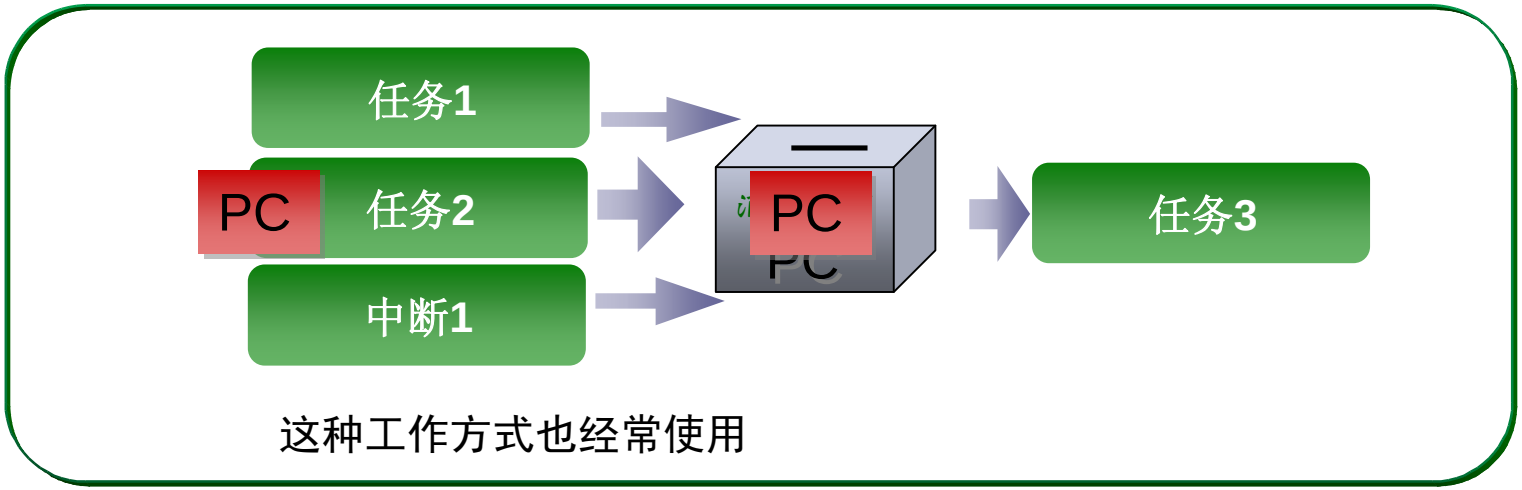
消息邮箱的工作方式

1.一对一



这种工作方式最简单，也是最常用的

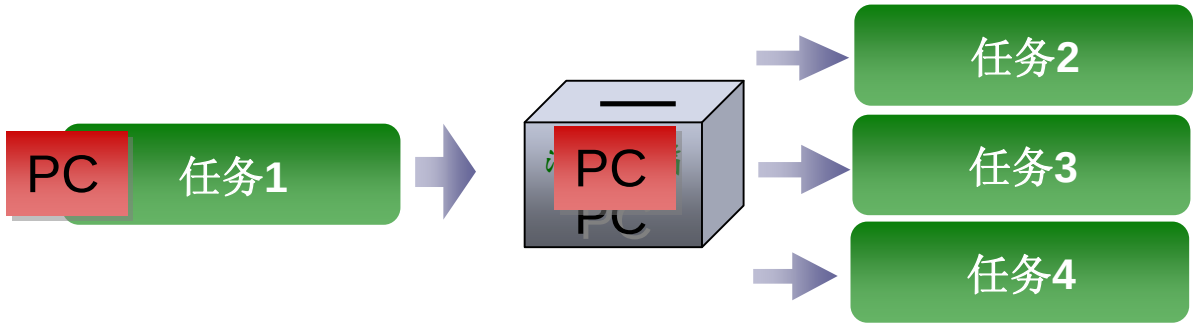
2.多对一



这种工作方式也经常使用

消息邮箱的工作方式

3.一对多



这种工作方式虽然不常见，但还是有极少场合使用，比如智能仪器仪表常常采用声、光与短信报警信号输出功能就是典型的一对多工作方式的应用

多对多与全双工的工作方式均不常见，在此不再作介绍。

消息邮箱的函数列表

消息邮箱7个函数详细信息如下表。

OSMboxPostOpt函数

函数名称	OSMboxCreate	所属文件	OS_MBOX.C
函数原型	OS_EVENT *OSMboxCreate(void *msg)		
功能描述	**建立并初始化一个消息邮箱**，邮箱允许任务或中断向其它一个或几个任务发送消息		
函数参数	msg： 用来初始化建立的消息邮箱。如果该指针不为空，则建立的消息邮箱将含有消息		
函数返回值	指向分配给所建立的消息邮箱的事件控制块的指针，如果没有可用的事件控制块，则返回空指针		

消息邮箱的函数列表

函数名称	OSMboxPend	所属文件	OS_MBOX.C
函数原型	void *OSMboxPend (OS_EVNNT *pevent, INT16U timeout, int8u *err)		
功能描述	任务等待消息		
函数参数	pevent：指向即将接收消息的消息邮箱的指针，OSMboxCreate()的返回值 timeout：最多等待的时间（超时时间），以时钟节拍为单位 err ：用于返回错误码		
函数返回值	如果消息已经到达，返回指向该消息的指针；如果消息邮箱没有消息，返回空指针*err可能为以下值： OS_NO_ERR： 得到消息；OS_TIMEOUT：超过等待时间 OS_ERR_PEND_ISR：在中断中调用该函数所引起的错误 OS_ERR_EVENT_TYPE：错误，pevent 不是指向邮箱的指针 OS_ERR_PEVENT_NULL：错误，pevent为NULL		

消息邮箱的函数列表

函数名称	OSMboxPost	所属文件	OS_MBOX.C
函数原型	INT8U OSMboxPost（OS_EVENT *pevent, void *msg）		
功能描述	通过消息邮箱向任务发送消息		
函数参数	pevent ： 指向即将接收消息的消息邮箱的指针，OSMboxCreate()的返回值 msg ： 将要发送的消息，不能为NULL		
函数返回值	OS_NO_ERR ： 消息成功地放到消息邮箱中 OS_MBOX_FULL ： 消息邮箱已经包含了其它消息，不空 OS_ERR_EVENT_TYPE ： pevent 不是指向消息邮箱的指针 OS_ERR_PEVENT_NULL ： 错误，pevent为NULL OS_ERR_POST_NULL_PTR：错误，msg为NULL		

消息邮箱的函数列表

函数名称	OSMboxDel	所属文件	OS_MBOX.C
函数原型	OS_EVENT *OSMboxDel (OS_EVENT *pevent, INT8U opt, INT8U *err)		
功能描述	**删除消息邮箱：**在删除消息邮箱之前，应当先删除可能会使用该邮箱的任务		
函数参数	pevent：指向信号量的指针，OSSemCreate()的返回值；err：用于返回错误码 opt：定义消息邮箱删除条件 OS_DEL_NO_PEND：没有任务等待信号量才删除；OS_DEL_ALWAYS：立即删除		
函数返回值	NULL：成功删除；pevent：删除失败；*err可能为以下值： OS_NO_ERR：成功删除邮箱 OS_ERR_DEL_ISR：在中断中删除邮箱所引起的错误 OS_ERR_INVALID_OPT：错误，opt值非法 OS_ERR_TASK_WAITING：有一个或多个任务在等待邮箱 OS_ERR_EVENT_TYPE：错误，pevent不是指向邮箱的指针 OS_ERR_PEVENT_NULL：错误，pevent为NULL		
特殊说明	当挂起任务就绪时，中断关闭时间与挂起任务数目有关		

消息邮箱的函数列表

函数名称	OSMboxAccept	所属文件	OS_MBOX.C
函数原型	void *OSMboxAccept(OS_EVENT *pevent)		
功能描述	**查看指定的消息邮箱是否有需要的消息：**不同于OSMboxPend()函数，如果没有需要的消息，则OSMboxAccept()函数并不挂起任务。如果消息已经到达，则该消息被传递给用户任务并且从消息邮箱中清除。通常中断调用该函数，因为中断不允许挂起等待消息		
函数参数	pevent：指向需要查看的消息邮箱的指针，OSSemCreate()的返回值		
函数返回值	如果消息已经到达，则返回指向该消息的指针；如果消息邮箱没有消息，则返回空指针		

消息邮箱的函数列表

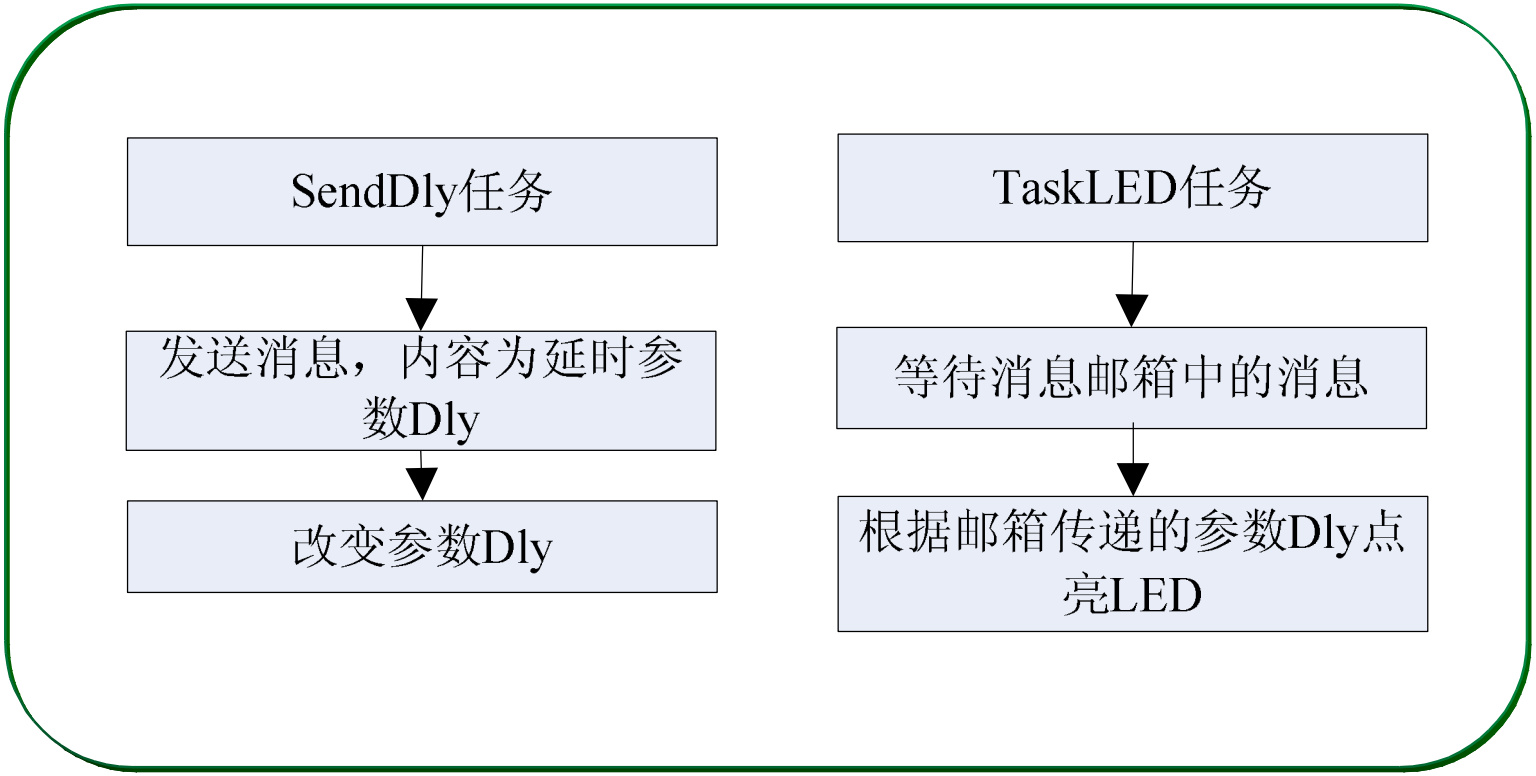
函数名称	OSMboxPostOpt	所属文件	OS_MBOX.C
函数原型	INT8U OSMboxPostOpt (OS_EVENT *pevent, void *msg, INT8U opt)		
功能描述	**OSMboxPost()的扩展**，也是通过消息邮箱发送消息，不过可选择发送方式。如果有任何任务在等待消息邮箱的消息，则可选最高优先级的任务或所有等待的任务获得这个消息并进入就绪状态。然后进行任务调度，决定当前运行的任务是否仍然为处于最高优先级就绪态的任务		
函数参数	pevent：指向即将接收消息的消息邮箱的指针，OSMboxCreate()的返回值 msg：将要发送的消息，不能为NULL opt：OS_POST_OPT_NONE：等待此邮箱优先级最高的任务获得消息 OS_POST_OPT_BROADCAST：所有等待此邮箱的任务均获得消息		
函数返回值	OS_NO_ERR：消息成功地放到消息邮箱中 OS_MBOX_FULL：消息邮箱已经包含了其它消息，不空 OS_ERR_EVENT_TYPE ：pevent 不是指向消息邮箱的指针 OS_ERR_PEVENT_NULL：错误，pevent为NULL OS_ERR_POST_NULL_PTR：错误，msg为NULL		
特殊说明	OS_POST_OPT_BROADCAST方式发送消息，本函数执行时间不确定		

消息邮箱的函数列表

函数名称	OSMboxQuery	所属文件	OS_MBOX.C
函数原型	INT8U OSMboxQuery(OS_EVENT *pevent, OS_MBOX_DATA *pdata)		
功能描述	**取得消息邮箱的信息：**用户程序必须分配一个OS_MBOX_DATA的数据结构，该结构用来从消息邮箱的事件控制块接收数据。通过调用OSMboxQuery()函数可以知道任务是否在等待消息以及有多少个任务在等待消息，还可以检查消息邮箱现在的消息		
函数参数	pevent ：指向即将接收消息的消息邮箱的指针，OSMboxCreate()的返回值 pdata ：指向OS_MBOX_DATA数据结构的指针，该数据结构包含下述成员： OSMsg：消息邮箱中消息的拷贝 OSEventTbl[]：消息邮箱等待队列的拷贝 OSEventGrp：消息邮箱等待队列索引的拷贝		
函数返回值	OS_NO_ERR ：调用成功 OS_ERR_EVENT_TYPE ：pevent 不是指向消息邮箱的指针 OS_ERR_PEVENT_NULL ：错误，pevent为NULL		

任务间数据通信

让一个LED以传递过来的参数确定点亮时间，以此示例来说明如何使用消息邮箱来实现任务间的数据通信，假设TaskLED为高优先级的任务。两个任务的处理流程如下。



任务间数据通信

发送延时参数任务SendDly的代码如下。

```
void SendDly (void)
{
    pdata = pdata;
    while (1) {
        OSMboxPost(mbox, & dly);
        OSTimeDly(1000);
        dly = dly + 20;
        if (dly >= 1000) {
            dly = 20;
        }
    }
}
```

防止编译器报警

发送延时参数消息

延时1000个节拍

每次延时参数加20

延时参数大于1000，返回到20

任务间数据通信

LED任务的代码如下。

```
void  TaskLED (void  *pdata)
{
    ……
    mbox = OSMboxCreate(NULL);
    while (1) {
        pd = (INT16U *)OSMboxPend(mbox, 0, &err);
        IO0CLR = LED1;
        OSTimeDly(*pd);
        IO0SET = LED1;
        OSTimeDly(10);
    }
}
```

初始化工作

创建消息邮箱

等待消息

点亮LED

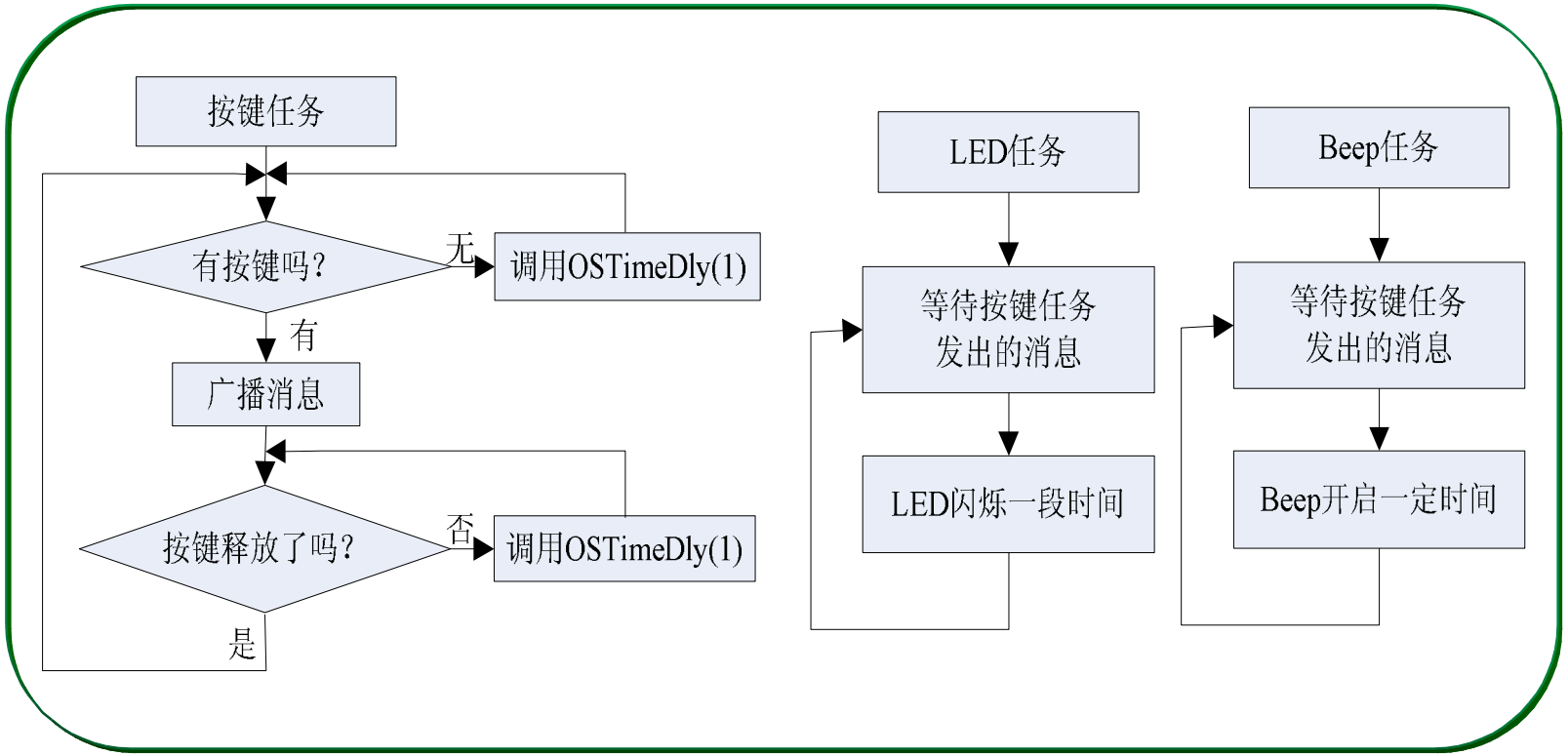
以消息内容为延时参数

熄灭LED

延时10clk

任务间同步

为了说明如何使用邮箱来实现任务间的同步，我们设计一个系统，按键一按下，LED按照一定的频率闪耀一定的时间，蜂鸣器开启一定的时间。假设TaskLED为高优先级的任务，三个任务的处理流程如下。



任务间同步

TaskKEY任务主要代码如下。

```
void  TaskKEY (void  *pdata)
{
    ……
    while (1) {
        while ((IO0PIN & KEY1) != 0) {
            OSTimeDly(1);
        }
        OSMboxPostOpt (mbox, &dly,
        OS_POST_OPT_BROADCAST);
        while ((IO0PIN & KEY1) == 0) {
            OSTimeDly(1);
        }
    }
}
```

初始化代码

等待按健按下

延时1个节拍，用于任务切换

以广播形式发送消息，所有等待此邮箱的任务均获得消息

等待按健释放

延时1个节拍，用于任务切换

任务间同步

LED任务主要代码如下。

```
void  TaskLED (void  *pdata)
{
    ……
    mbox = OSMboxCreate(NULL);
    while (1) {
        pd = (INT16U *)OSMboxPend(mbox, 0, &err);
        for (i = 0; i < 10; i++) {
            IO0CLR = LED1;
            OSTimeDly(10);
            IO0SET = LED1;
            OSTimeDly(10);
        }
    }
}
```

初始化工作

创建消息邮箱

等待消息

循环10次

点亮LED

延时10clk

熄灭LED

延时10clk

任务间同步

Beep任务主要代码如下。

```
void  TaskBeep (void  *pdata)
{
    ……
    while (1) {
        pd = (INT16U *)OSMboxPend(mbox, 0, &err);
        IO0CLR = BEEP ;
        OSTimeDly(150);
        IO0SET = BEEP ;
        OSTimeDly(50);
    }
}
```

初始化工作

等待消息

开启蜂鸣器

延时150clk

关闭蜂鸣器

延时50clk

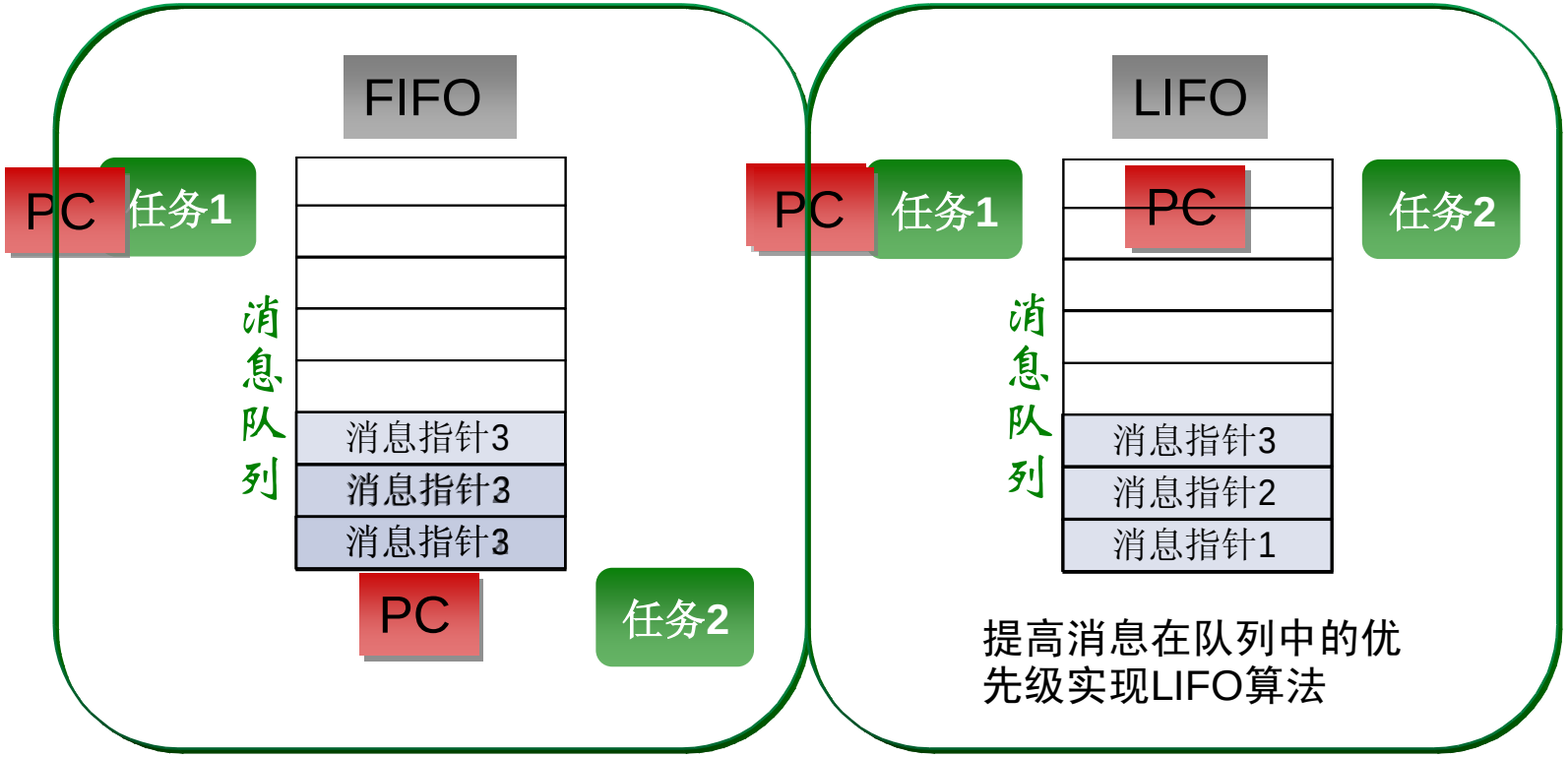
消息队列

μC/OS-II程序设计基础

消息队列 | μC/OS-II程序设计基础

简介

消息队列就象一个类似于缓冲区的对象，通过消息队列任务和ISR发送和接收消息，实现数据的通信和同步。消息队列具有一定的容量，可以容纳多条消息，因此可以看成是多个邮箱的组合。



简介

1.向消息队列发送消息

当任务向消息队列中发送消息时，它首先判断是否有任务在等待消息队列的消息。

注意：如果发送消息指针是以广播的形式发送，那么所有等待此消息的任务都获得消息

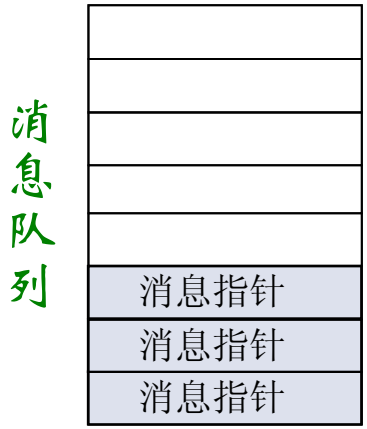
简介

如果没有任务在等待消息队列的消息，那么就会再判断消息队列当前是否已满。

无任务在等待消息

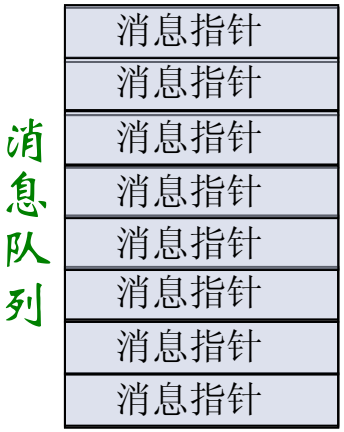
发送消息指针 PC

未满



操作成功

已满 发送失败

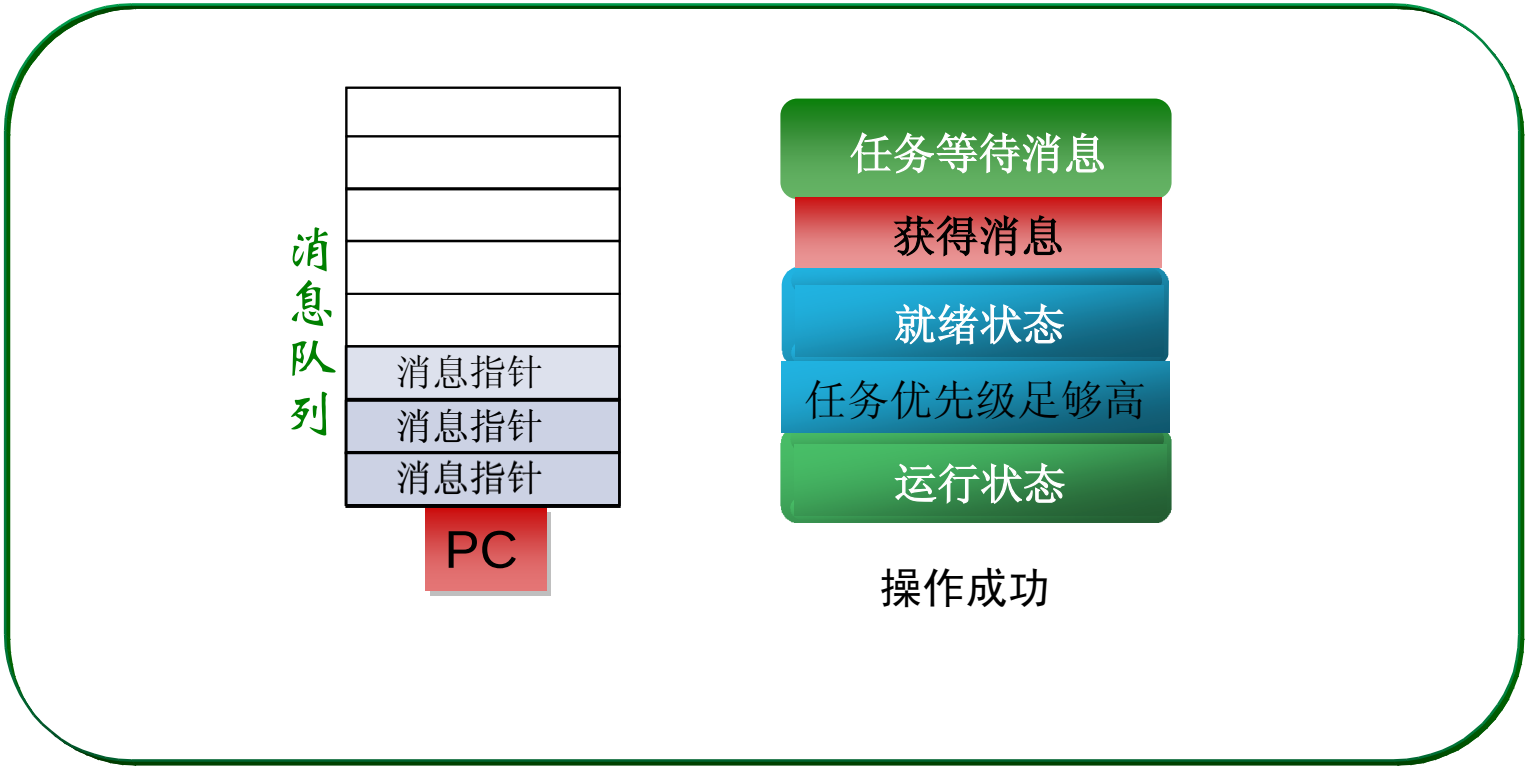


返回错误码说明
消息队列已满

简介

2.从消息队列接收消息

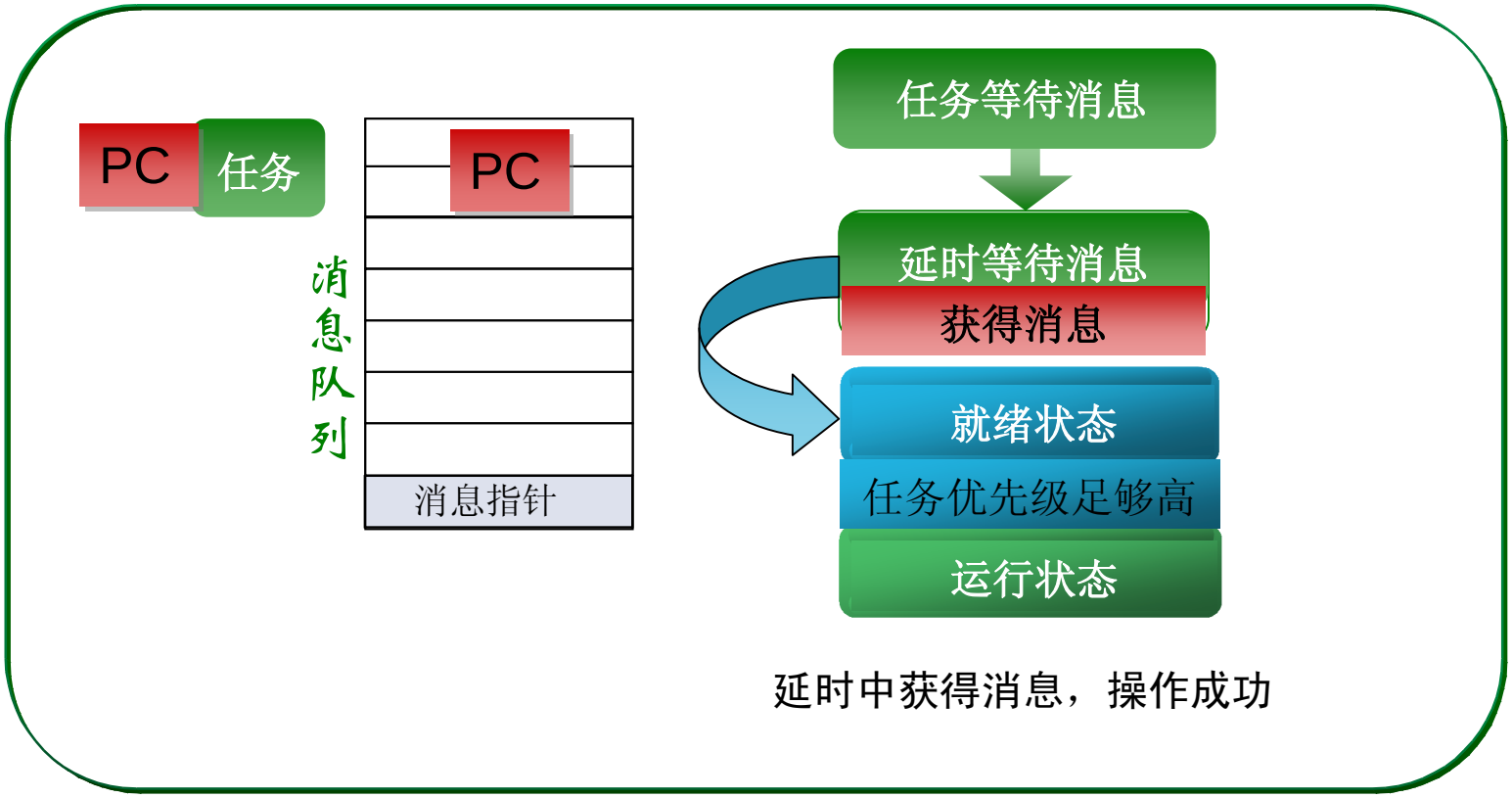
消息队列中已存在消息，通过内核服务将消息传递给等待消息的任务中优先级最高的任务，或最先进入等待消息任务列表的任务。



消息队列 | μC/OS-II程序设计基础

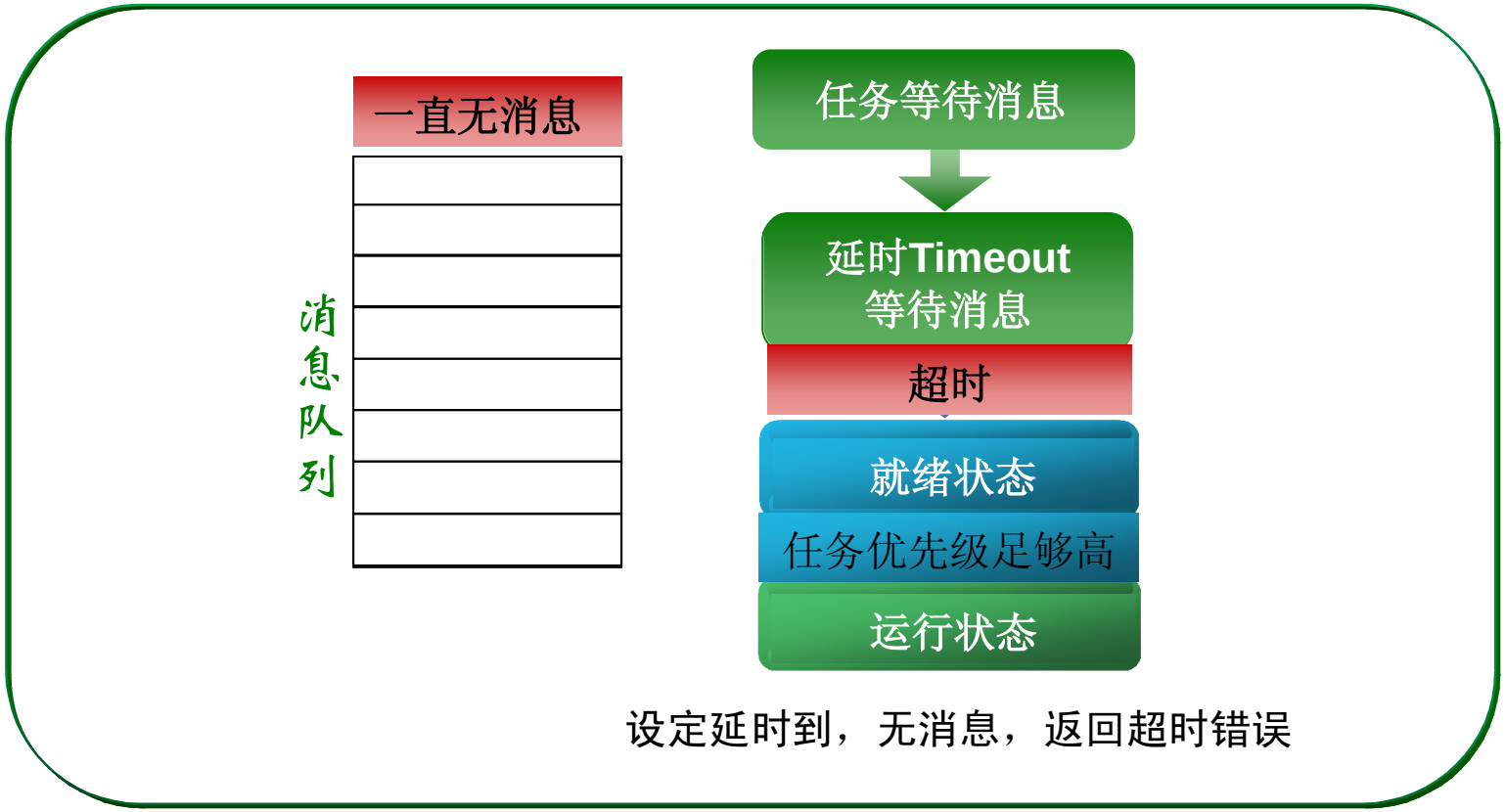
简介

如果消息队列为空，则等待消息的任务被放入等待消息的任务列表中，直到有其它任务向消息队列发送消息后才能解除该等待状态或在超时的情况下运行。



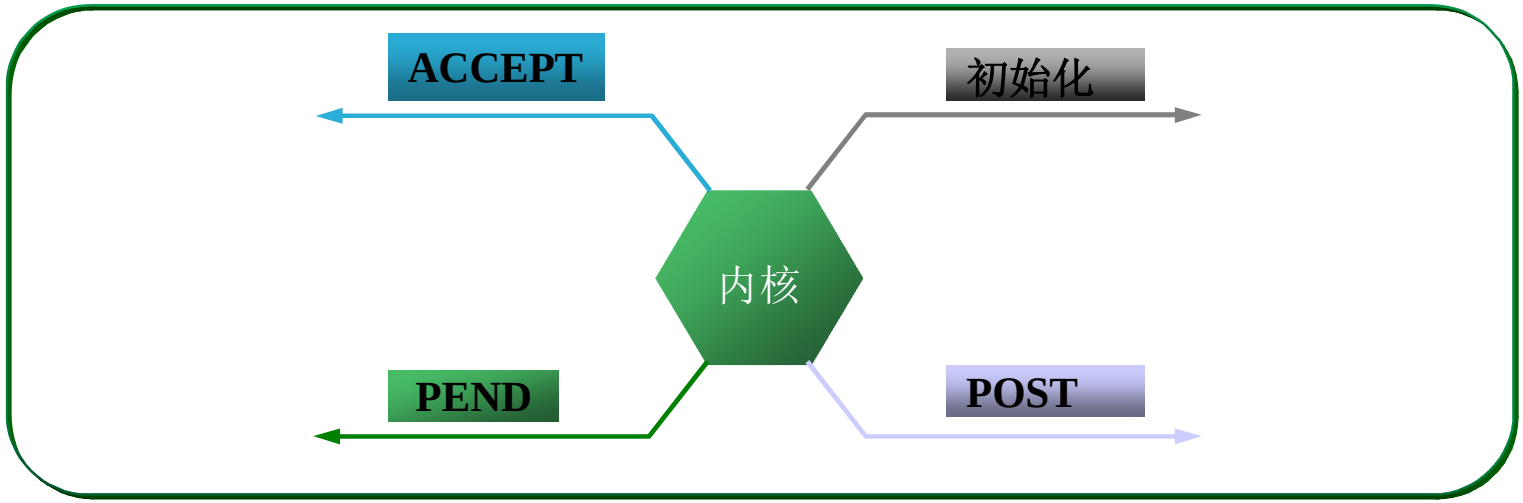
简介

一般来说，OSQPend()函数允许用户定义一个最长的等待时间Timeout作为它的参数，这样可以避免该任务无休止地等待下去。



简介

内核一般提供以下消息队列服务：



与信号量和邮箱相比，消息队列的最大优点就是通过缓冲的方式来传递多个消息，从而避免了信息的丢失或混乱。

简介

内核一般提供以下消息队列服务：

消息队列初始化，队列初始化时总是清为空；

将消息放入队列中去（POST）；

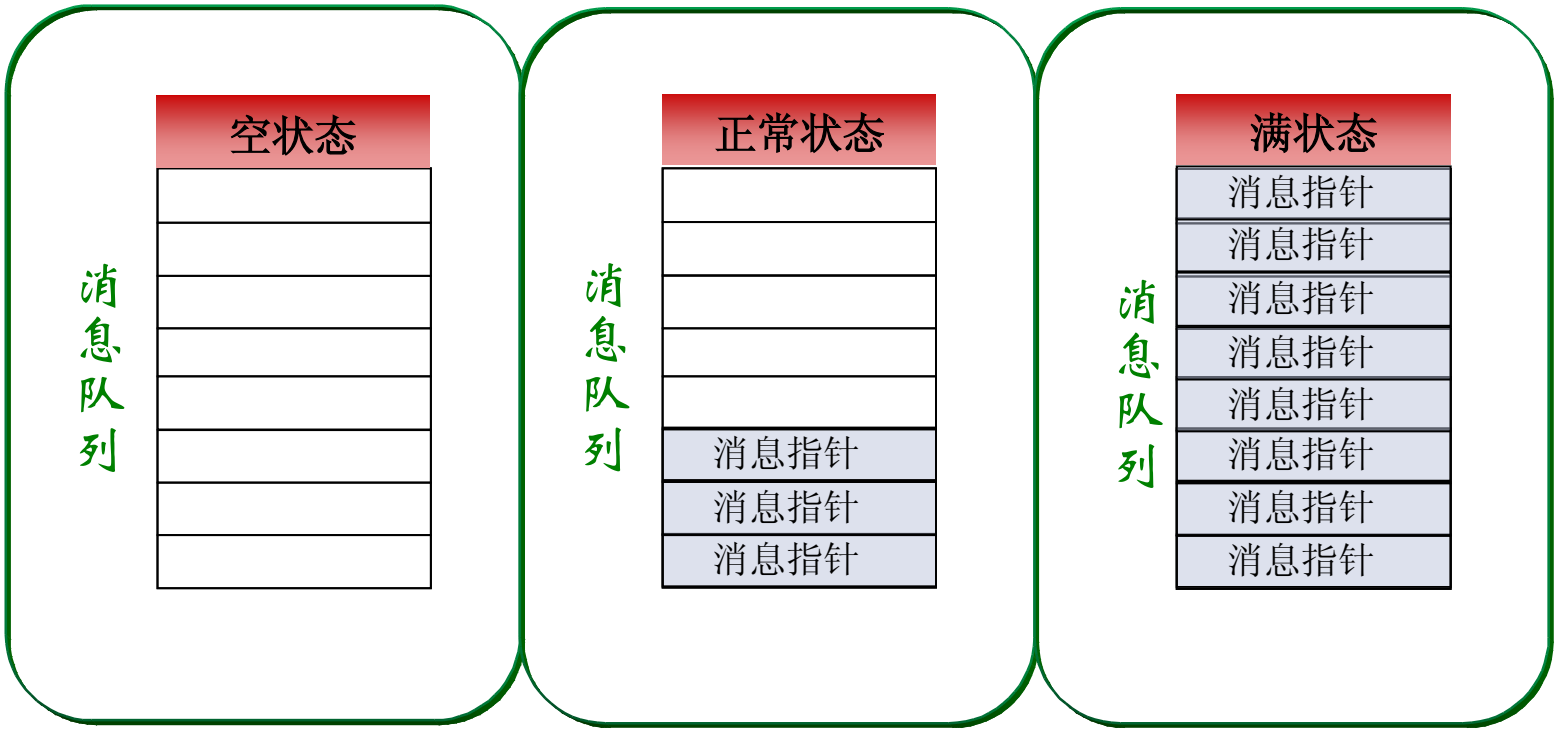
等待消息的到来（PEND）；

无等待取得消息，如果消息队列中有消息，则任务将消息从消息队列中取走；如果消息队列为空，则内核不将该任务挂起，返回空指针。

与信号量和邮箱相比，消息队列的最大优点就是通过缓冲的方式来传递多个消息，从而避免了信息的丢失或混乱。

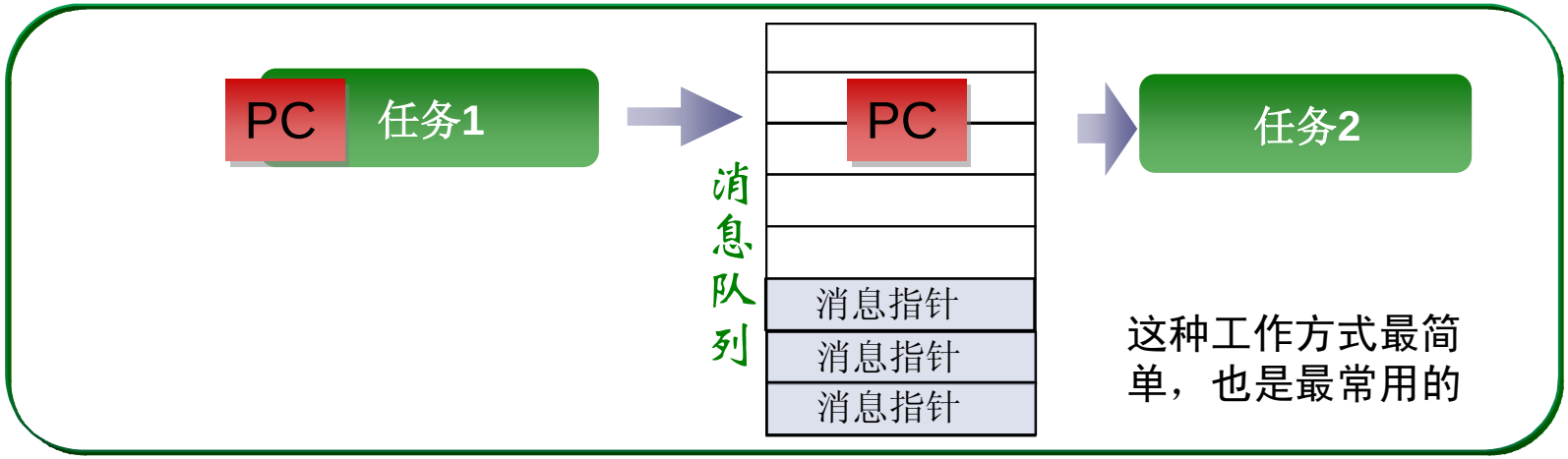
消息队列的状态

一般来说，消息队列有3种状态，即空状态（消息队列中没有任何消息）、满状态（消息队列中的每个存储单元都存放了消息）、正常状态（消息队列中消息但又没有到满的状态）。

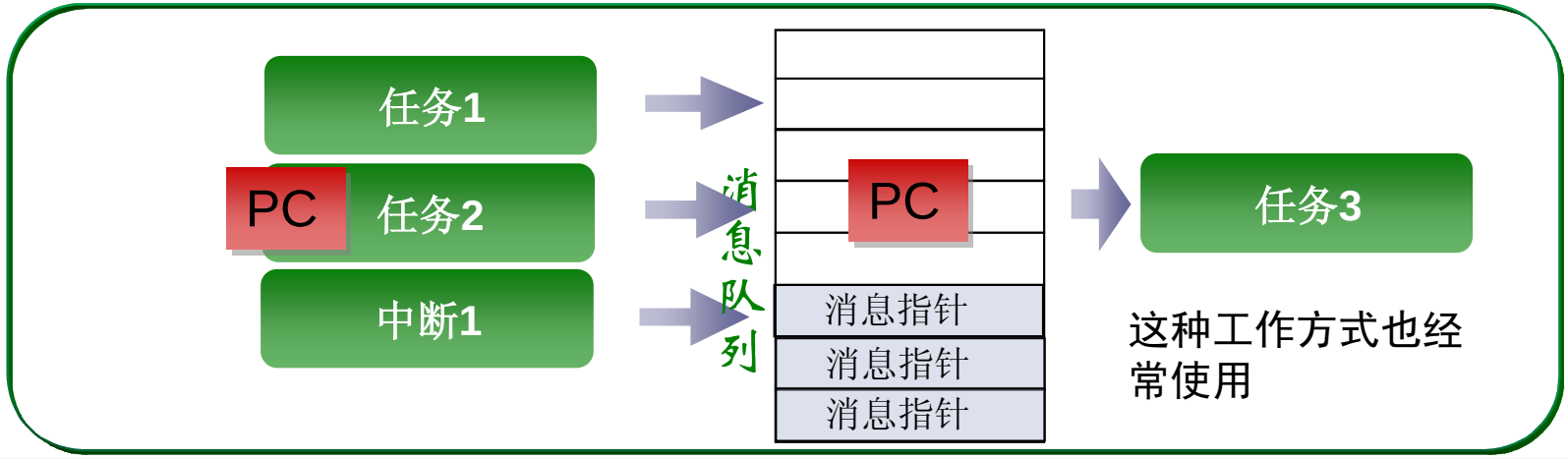


消息队列的工作方式

1.一对一

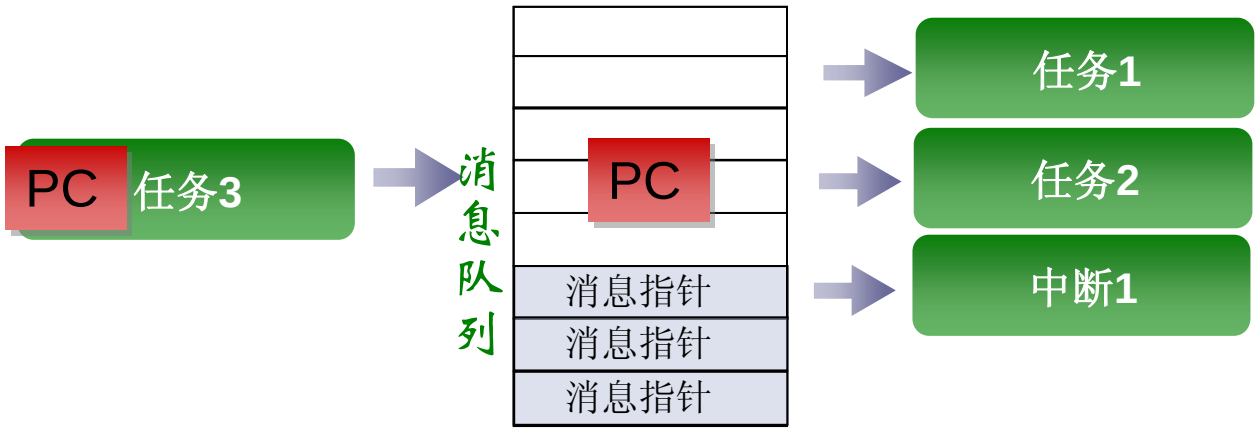


2.多对一



消息队列的工作方式

3.一对多



这种工作方式虽然不常见，但还是有极少场合使用，比如智能仪器仪表常常采用声、光与短信报警信号输出功能就是典型的一对多工作方式的应用

多对多与全双工的工作方式均不常见，在此不再作介绍。

消息队列的函数列表

消息队列9个函数详细信息如下表。

OSQPostFront函数

函数名称	OSQCreate	所属文件	OS_Q.C
函数原型	OS_EVENT *OSQCreate(void **start, INT8U size)		
功能描述	**建立一个消息队列：**任务或中断可以通过消息队列向其它一个或多个任务发送消息		
函数参数	start：消息内存区的基地址，消息内存区是一个指针数组；size：消息内存区的大小		
函数返回值	返回一个指向消息队列事件控制块的指针，如果没有空余的事件空闲块，则返回空指针		

函数返回值	OS_NO_ERR ： 消息成功地放到消息队列中 OS_Q_FULL ： 消息队列已经满 OS_ERR_EVENT_TYPE ： pevent 不是指向消息队列的指针 OS_ERR_PEVENT_NULL ： 错误，pevent为NULL OS_ERR_POST_NULL_PTR：错误，msg为NULL
特殊说明	包含OS_POST_OPT_BROADCAST的方式发送消息，本函数执行时间不确定

消息队列的函数列表

函数名称	OSQPend	所属文件	OS_Q.C
函数原型	void *OSQPend(OS_EVENT *pevent, INT16U timeout, INT8U *err)		
功能描述	任务等待消息		
函数参数	pevent ： 指向即将接收消息的消息队列的指针，OSQCreate()的返回值 timeout：超时时间，以时钟节拍为单位；err ： 用于返回错误码		
函数返回值	如果消息已经到达，则返回指向该消息的指针；如果消息队列没有消息，则返回空指针，*err可能为以下值： OS_NO_ERR ： 得到消息 OS_TIMEOUT ： 超过等待时间 OS_ERR_PEND_ISR ： 在中断中调用该函数所引起的错误 OS_ERR_EVENT_TYPE ： 错误，pevent 不是指向消息队列的指针 OS_ERR_PEVENT_NULL ： 错误，pevent为NULL		

消息队列的函数列表

函数名称	OSQFlush	所属文件	OS_Q.C
函数原型	INT8U OSQFlush (OS_EVENT *pevent)		
功能描述	**清空消息队列：**并忽略发往消息队列的所有消息		
函数参数	pevent ： 指向消息队列的指针，OSQCreate ()的返回值		
函数返回值	OS_NO_ERR ： 成功清空消息队列 OS_ERR_EVENT_TYPE ： 错误，pevent不是指向消息队列的指针 OS_ERR_PEVENT_NULL ： 错误，pevent为NULL		
特殊说明	当挂起任务就绪时，中断关闭时间与挂起任务数目有关		

消息队列的函数列表

函数名称	OSQPost	所属文件	OS_Q.C
函数原型	INT8U OSQPost(OS_EVENT *pevent, void *msg)		
功能描述	通过消息队列向任务发送消息		
函数参数	pevent ： 指向即将接收消息的消息队列的指针，OSQCreate()的返回值 msg ： 将要发送的消息，不能为NULL		
函数返回值	OS_NO_ERR ： 消息成功地放到消息队列中 OS_Q_FULL ： 消息队列已经满 OS_ERR_EVENT_TYPE ： pevent 不是指向消息队列的指针 OS_ERR_PEVENT_NULL ： 错误，pevent为NULL OS_ERR_POST_NULL_PTR：错误，msg为NULL		

消息队列的函数列表

函数名称	OSQDel	所属文件	OS_Q.C
函数原型	OS_EVENT *OSQDel (OS_EVENT *pevent, INT8U opt, INT8U *err)		
功能描述	**删除消息队列：**在删除消息队列之前，应当先删除可能会使用这个消息队列的任务		
函数参数	pevent ：指向消息队列的指针，OSQCreate ()的返回值；err：用于返回错误码 opt ：定义消息队列删除条件 OS_DEL_NO_PEND：没有任务等待消息队列才删除；OS_DEL_ALWAYS：立即删除		
函数返回值	NULL：成功删除；Pevent：删除失败；*err可能为以下值： OS_NO_ERR ：成功删除消息队列 OS_ERR_DEL_ISR ：在中断中删除消息队列所引起的错误 OS_ERR_INVALID_OPT ：错误，opt值非法 OS_ERR_TASK_WAITING ：有一个或多个任务在等待消息队列 OS_ERR_EVENT_TYPE ：错误，pevent不是指向消息队列的指针 OS_ERR_PEVENT_NULL ：错误，pevent为NULL		
特殊说明	当挂起任务就绪时，中断关闭时间与挂起任务数目有关		

消息队列的函数列表

函数名称	OSQPostFront	所属文件	OS_Q.C
函数原型	INT8U OSQPostFront(OS_EVENT *pevent, void *msg)		
功能描述	**通过消息队列以LIFO方式向任务发送消息：**它与OSQPost()的唯一不同，就是后发送的消息将更早的被获得。也就是说，OSQPostFront()将消息插到消息队列的最前面		
函数参数	pevent ： 指向即将接收消息的消息队列的指针，OSQCreate()的返回值 msg ： 将要发送的消息，不能为NULL		
函数返回值	OS_NO_ERR ： 消息成功地放到消息队列中 OS_Q_FULL ： 消息队列已经满 OS_ERR_EVENT_TYPE ： pevent 不是指向消息队列的指针 OS_ERR_PEVENT_NULL ： 错误，pevent为NULL OS_ERR_POST_NULL_PTR：错误，msg为NULL		

消息队列的函数列表

函数名称	OSQAccept	所属文件	OS_Q.C
函数原型	void *OSQAccept(OS_EVENT *pevent)		
功能描述	**检查消息队列中是否已经有需要的消息：**不同于OSQPend()，如果没有需要的消息，则OSQAccept()并不挂起任务；如果消息已经到达，则该消息被传递给用户任务。通常中断调用该函数，因为中断不允许挂起等待消息		
函数参数	pevent：指向需要查看的消息队列的指针，OSQCreate ()的返回值		
函数返回值	如果消息已经到达，则返回指向该消息的指针；如果消息队列没有消息，则返回空指针		

消息队列的函数列表

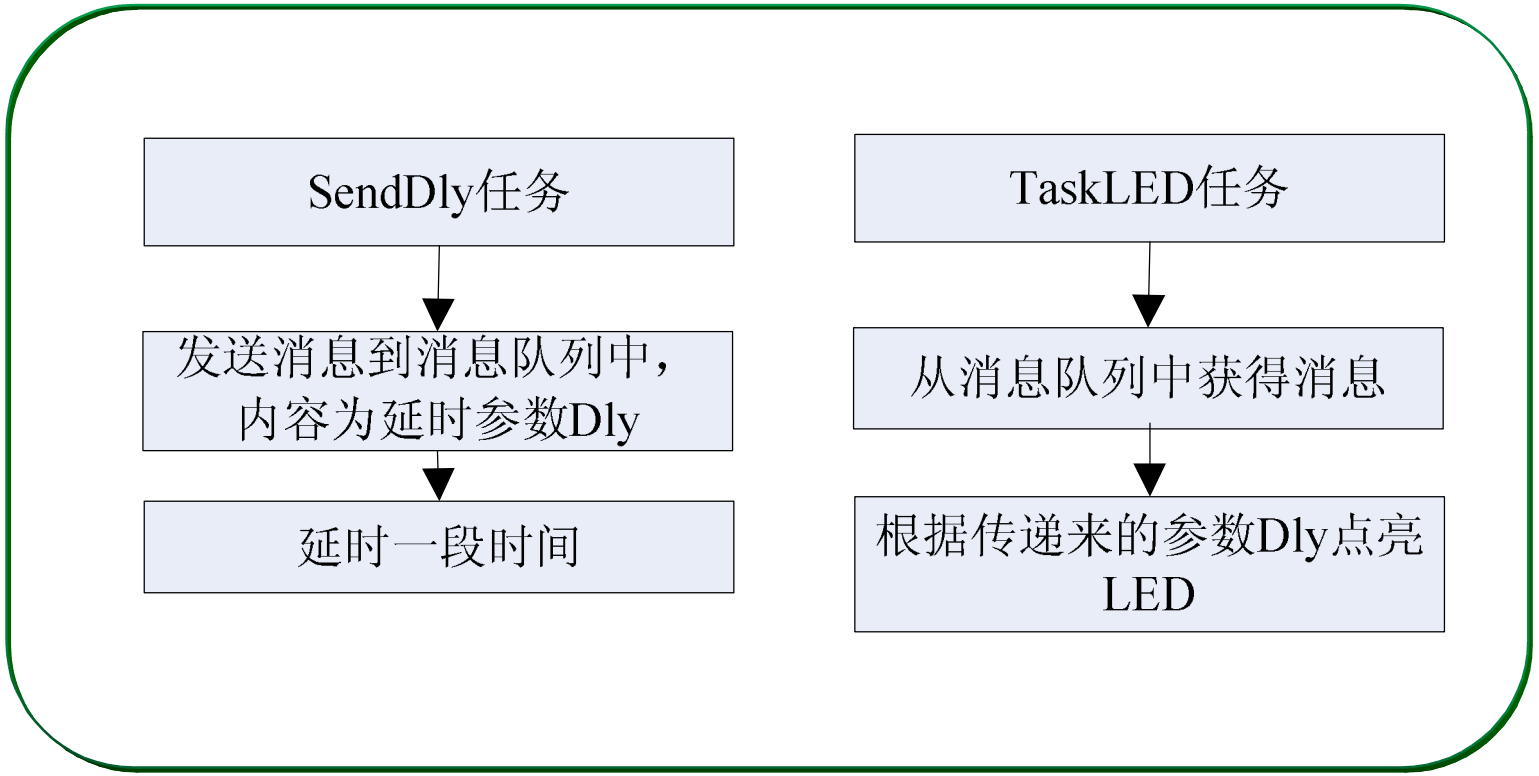
函数名称	OSQPostOpt	所属文件	OS_Q.C
函数原型	INT8U OSQPostOpt (OS_EVENT *pevent, void *msg, INT8U opt)		
功能描述	OSQPostOpt ()是OSMboxPost()和OSQPostFront()的扩展，包含它们的功能同时进行了增强：可选择发送方式。如果消息队列中已经存满消息，则返回错误码说明消息队列已满，立即返回调用者，消息也没有发到消息队列。如果有任何任务在等待消息队列的消息，则可选最高优先级的任务或所有等待的任务获得这个消息并进入就绪状态。然后进行任务调度，决定当前运行的任务是否仍然为处于最高优先级就绪态的任务		
函数参数	pevent ：指向即将接收消息的消息队列的指针，OSQCreate()的返回值 msg ：将要发送的消息，不能为NULL Opt ：OS_POST_OPT_NONE：等同OSQPost() OS_POST_OPT_BROADCAST：等同OSQPost()，但所有等待此消息队列的任务均获得消息 OS_POST_OPT_FRONT：等同OSQPostFront() OS_POST_OPT_BROADCAST | OS_POST_OPT_FRONT：等同OSQPostFront()，但所有等待此消息队列的任务均获得消息		

消息队列的函数列表

函数名称	OSQQuery	所属文件	OS_Q.C
函数原型	INT8U OSQQuery(OS_EVENT *pevent, OS_Q_DATA *pdata)		
功能描述	**取得消息队列的信息：**用户程序必须建立一个OS_Q_DATA的数据结构，该结构用来保存从消息队列的事件控制块得到的数据。通过调用OSQQuery()可以知道任务是否在等待消息、有多少个任务在等待消息、队列中有多少消息以及消息队列可以容纳的消息数，OSQQuery()还可以得到即将被传递给任务的消息的信息		
函数参数	pevent ： 指向即将接收消息的消息队列的指针，OSQCreate()的返回值 pdata ： 指向OS_Q_DATA数据结构的指针，该数据结构包含下述成员： OSMsg:： 下一个可用的消息 OSNMsgs： 队列中的消息数目 OSQSize： 消息队列的大小 OSEventTbl[]：消息队列的等待队列的拷贝 OSEventGrp：消息队列等待队列索引的拷贝		
函数返回值	OS_NO_ERR ： 调用成功 OS_ERR_EVENT_TYPE ： 错误，pevent 不是指向消息队列的指针 OS_ERR_PEVENT_NULL ： 错误，pevent为NULL		

数据通信

让一个LED以传递过来的参数确定点亮时间，以此示例来说明如何使用消息队列来实现任务之间的数据通信，假设TaskLED为高优先级的任务。两个任务的处理流程如下。



LED任务的代码如下。

```
void  TaskLED (void  *pdata)
{
    ……
    q = OSQCreate(msg, QSIZE);
    while (1) {
        pd = (INT16U *) OSQPend (q, 0, &err);
        IO0CLR = LED1;
        OSTimeDly(*pd);
        IO0SET = LED1;
        OSTimeDly(10);
    }
}
```

初始化工作

创建消息队列

等待接收消息指针

点亮LED

以消息内容为延时参数

熄灭LED

延时10clk

发送延时参数任务SendDly的代码如下。

```
void SendDly (void)
{
    pdata = pdata;
    for (i = 0; i < QSIZE; i++) {
        dly[i] = i*20;
    }
    while (1) {
        for (i = 0; i < QSIZE; i++) {
            OSQPost (q, &dly[i]);
        }
        OSTimeDly(QSIZE * QSIZE *20);
    }
}
```

防止编译器报警

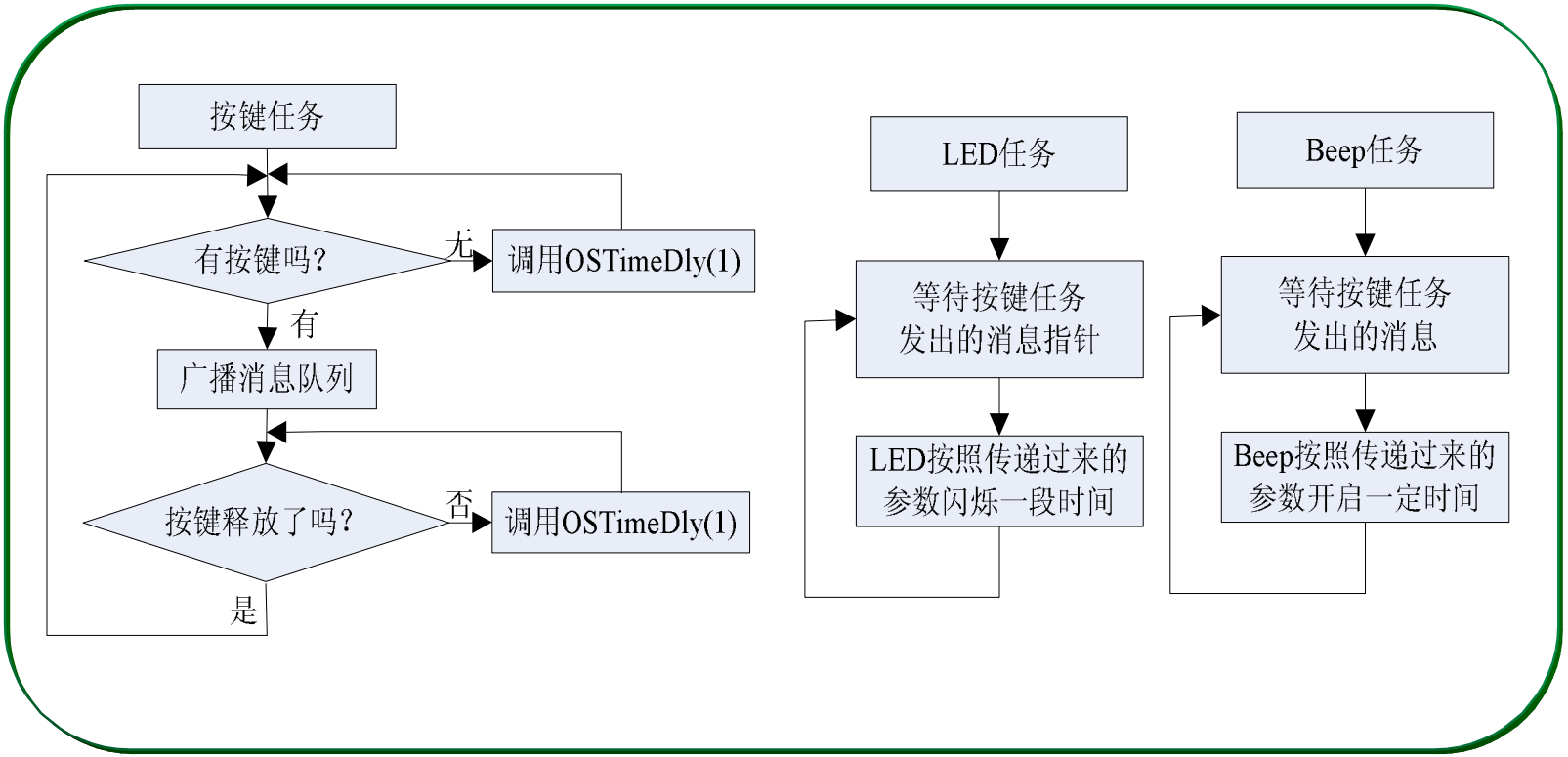
存储延时参数，每次加20

将延时参数存放地址放入队列中

延时一段时间

多任务接收数据

为了说明如何使用消息队列来实现多任务接收数据，我们设计一个系统，按键一按下， LED按照指定节奏闪耀，蜂鸣器按照指定节奏鸣响。假设TaskLED为高优先级的任务，三个任务的处理流程如下。



任务间同步

TaskKEY任务主要代码如下。

```
void  TaskKEY (void  *pdata)
{
    ……
    for (i = 0; i < QSIZE; i++) {
        dly[i] = i*20;
    }
    while (1) {
        while ((IO0PIN & KEY1) != 0) {
            OSTimeDly(1);
        }
        for (i = 0; i < QSIZE; i++) {
            OSQPostOpt (q, &dly[i],
            OS_POST_OPT_BROADCAST);
        }
        while ((IO0PIN & KEY1) == 0) {
            OSTimeDly(1);
        }
    }
}
```

初始化代码

存储延时参数，每次加20

等待按健按下

延时1个节拍，用于任务切换

以广播形式发送消息，所有等待此队列的任务均获得消息

等待按健释放

延时1个节拍，用于任务切换

任务间同步

LED任务的代码如下。

```
void  TaskLED (void  *pdata)
{
    ……
    q = OSQCreate(msg, QSIZE);
    while (1) {
        pd = (INT16U *) OSQPend (q, 0, &err);
        IO0CLR = LED1;
        OSTimeDly(*pd / 2 );
        IO0SET = LED1;
        OSTimeDly(*pd / 2 );
    }
}
```

初始化工作

创建消息队列，长度为QSIZE

等待接收消息指针

点亮LED

以消息内容作为参数延时

熄灭LED

以消息内容作为参数延时

任务间同步

Beep任务主要代码如下。

```
void  TaskBeep (void  *pdata)
{
    ……
    while (1) {
        pd = (INT16U *) OSQPend (q, 0, &err);
        IO0CLR = BEEP ;
        OSTimeDly(*pd * 3 / 4 );
        IO0SET = BEEP ;
        OSTimeDly(*pd / 4 );
    }
}
```

初始化工作

等待接收消息指针

开启蜂鸣器

以消息内容作为参数延时

关闭蜂鸣器

以消息内容作为参数延时

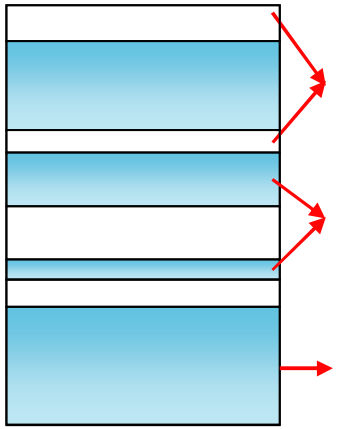
动态内存管理

µC/OS-II程序设计基础

1. 简介
2. 函数列表
3. 数据通信

简介

ANSI C中，可以使用malloc()和free()两个函数来动态分配内存，在嵌入式系统中，它们一般也是可用的，但并不适合。如图为被两个函数分配过的内存区。



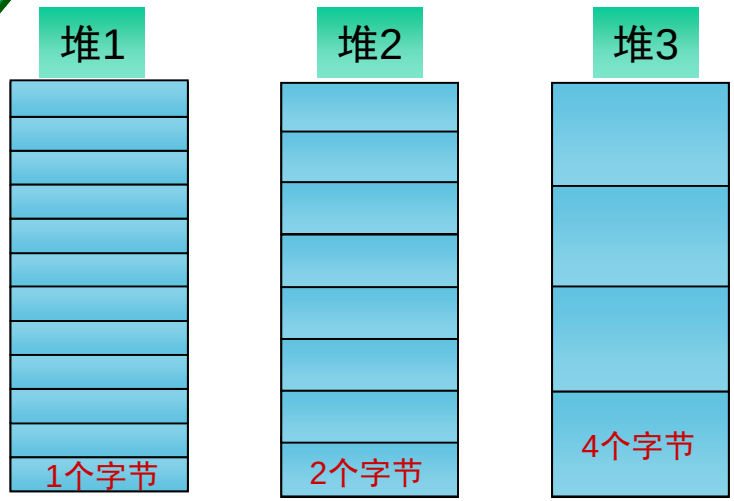
产生碎片，造成连续内存不够大，导致有内存也不能分配，程序执行失败；

分配和释放的内存块大小不同，所以执行时间不确定；

内存一般是不可重入的。

简介

为了避免上面的3个问题，μC/OS-II自己设计了一套动态内存分配系统。μC/OS-II的动态内存分配是以块为单位分配的，一次只能分配一个块，块的大小可以由用户来定义。



系统可以管理多个堆（用于动态内存管理的空间），各个堆中的块的大小可以定义得不一样，但同一个堆中的块大小是相同的。

μC/OS-II的动态内存管理是数据队列的绝佳伴侣，配合使用异常方便。

函数列表

动态内存管理的4个函数详细信息如下表。

OSMemQuery函数

函数名称	OSMemCreate	所属文件	OS_MEM.C
函数原型	OS_MEM *OSMemCreate(void *addr, INT32U nblks ,INT32U blksize, INT8U *err)		
功能描述	**建立并初始化一块内存区：**一块内存区包含指定数目的大小确定的内存块，程序可以包含这些内存块并在用完后释放回内存区		
函数参数	addr：建立的内存区的起始地址 nblks：需要的内存块的数目，每一个内存区最少需要定义两个内存块 blksize：每个内存块的大小，最少应该能够容纳一个指针 err：用于返回错误码		
函数返回值	指向内存区控制块的指针。如果没有剩余内存区，则返回空指针 *err可能为以下值： OS_NO_ERR ：成功建立内存区 OS_MEM_INVALID_PART ：没有空闲的内存区 OS_MEM_INVALID_BLKS ：没有为每一个内存区建立至少两个内存块 OS_MEM_INVALID_SIZE ：内存块大小不足以容纳一个指针变量		

函数列表

函数名称	OSMemPut	所属文件	OS_MEM.C
函数原型	INT8U OSMemPut(OS_MEM *pmem, void *pblk)		
功能描述	**释放一个内存块：**内存块必须释放回原先申请的内存区		
函数参数	pmem ： 指向内存区控制块的指针，OSMemCreate()的返回值 pblk ： 指向将被释放的内存块的指针		
函数返回值	OS_NO_ERR ： 成功释放内存块 OS_MEM_FULL ： 错误，内存区已经不能再接收更多释放的内存块。这种情况说明用户程序出现了错误，释放了多于用OSMemGet()函数得到的内存块 OS_MEM_INVALID_PMEM ： 错误：pmem为NULL OS_MEM_INVALID_PBLK: 错误：pblk为NULL		

函数列表

函数名称	OSMemGet	所属文件	OS_MEM.C
函数原型	void *OSMemGet(OS_MEM *pmem, INT8U *err)		
功能描述	**从内存区分配一个内存块：**用户程序必须知道所建立的内存块的大小，同时用户程序必须在使用完内存块后释放内存块，可以多次调用本函数		
函数参数	pmem ： 指向内存区控制块的指针，OSMemCreate()的返回值；err ： 用于返回错误码		
函数返回值	指向内存区块的指针，如果没有空间分配给内存块，则返回空指针； *err可能为以下值： OS_NO_ERR ： 成功得到一个内存块 OS_MEM_NO_FREE_BLKS ： 错误：内存区已经没有空间分配给内存块 OS_MEM_INVALID_PMEM ： 错误：pmem为NULL		

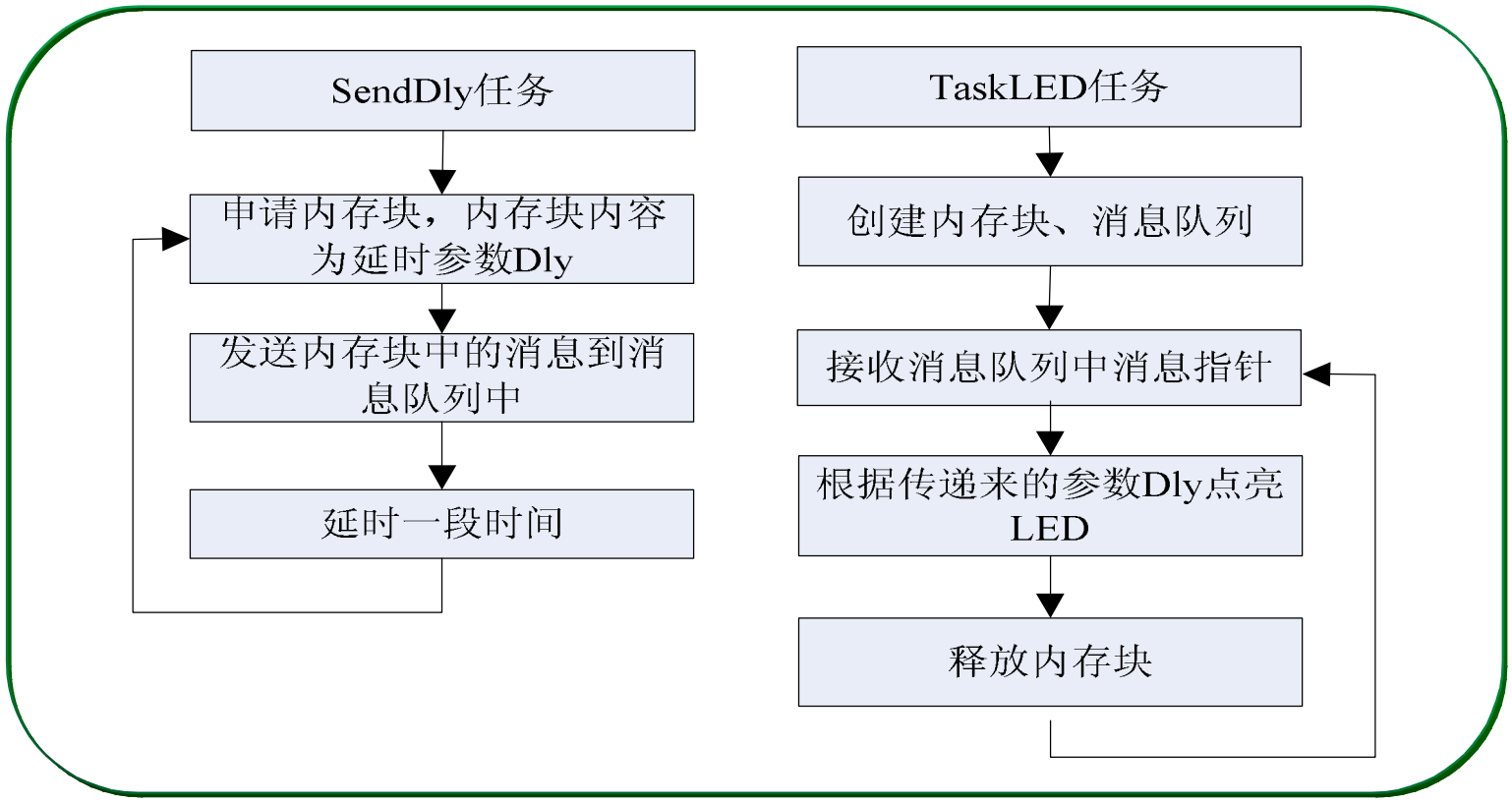
函数列表

函数名称	OSMemQuery	所属文件	OS_MEM.C
函数原型	INT8U OSMemQuery(OS_MEM *pmem, OS_MEM_DATA *pdata)		
功能描述	**得到内存区的信息：**该函数返回OS_MEM结构包含的信息，但使用了一个新的OS_MEM_DATA的数据结构，OS_MEM_DATA数据结构还包含了正被使用的内存块数目的域		
函数参数	pmem ：指向内存区控制块的指针，OSMemCreate()的返回值 pdata ：指向OS_MEM_DATA数据结构的指针，该数据结构包含了以下的域 OSAddr：指向内存区起始地址的指针 OSFreeList：指向空闲内存块列表起始地址的指针 OSBlkSize：每个内存块的大小 OSNBlks：该内存区的内存块总数 OSNFree：空闲的内存块数目 OSNUsed：使用的内存块数目		
函数返回值	OS_NO_ERR ：存储块有效 OS_MEM_INVALID_PMEM ：错误，pmem为NULL OS_MEM_INVALID_PDATA：错误，pdata为NULL		

动态内存管理| μC/OS-II程序设计基础

数据通信

让一个LED以传递过来的参数确定点亮时间，以此示例来说明如何使用动态内存管理来实现数据通信，假设TaskLED为高优先级的任务。两个任务的处理流程如下。



LED任务的代码如下。

```
void  TaskLED (void  *pdata)
{
    ……
    q = OSQCreate(msg, QSIZE);
    mem = OSMemCreate(dly,
                QSIZE, sizeof(INT32U ), &err);
    while (1) {
    pd = (INT32U *) OSQPend (q, 0, &err);
    IO0CLR = LED1;
    OSTimeDly(*pd);
    OSMemPut(mem, pd);
    IO0SET = LED1;
    OSTimeDly(10);
    }
}
```

初始化工作

创建消息队列

创建内存分区，保存消息

等待接收消息指针

点亮LED

以消息内容为延时参数

释放内存块

熄灭LED

延时10clk

发送消息任务SendDly的代码如下。

```
void SendDly (void)
{
    pdata = pdata;
    while (1) {
        for (i = 0; i < QSIZE; i++) {
            tp = OSMemGet(mem, &err);
           *tp = i * 20;
            OSQPost (q, (void *)tp);
        }
        OSTimeDly(QSIZE * QSIZE * 20);
    }
}
```

防止编译器报警

申请一个内存块

存储延时参数到内存中

发送内存块中消息指针

延时一段时间

本章小结

μC/OS-II程序设计基础

通过对本章的学习，相信对基于μC/OS-II 操作系统的程序设计有了基础的掌握， 与程序设计技术有关的内容，请参考本书的姊妹篇《基于嵌入式实时操作系统的程序设计技术》（周航慈教授著，北京航空航天大学出版社）。