嵌入式操作系统UCOS －II

- uCOS－II操作系统的特点
- uCOS －II操作系统内核结构
- uCOS －II操作系统任务管理
- uCOS －II操作系统内存管理
- uCOS －II操作系统时间管理
- uCOS －II操作系统任务间的通讯
- uCOS －II操作系统移植

μC/OS-II的特点

- UC/OS是一个非常小巧的实时操作系统；整个代码分为内核层以及移植层，这样使得它的植性很方便。
- 采用抢占式调度策略，保证任务的实时性。
- 能够管理多达64个任务。
- 提供了邮箱、消息队列、信号量、内存管理、时间管理等系统服务。

µC/OS-II的各种商业应用

- 全世界有数百种产品在应用:
  - 医疗器械
  - 移动电话
  - 路由器
  - 工业控制
  - GPS 导航系统
  - 智能仪器
  - 更多

μC/OS-II的代码结构

uC/OS-II
(与处理器无关的代码)

OS_CORE.C	OS_MUTEX.C
OS_MBOX.C	OS_SEM.C
OS_TIME.C	OS_Q.C
OS_FLAG.C	OS_MEM.C
OS_TASK.C	uCOS_II.H

uC/OS-II设置
(与应用相关的代码)
OS_CFG.H

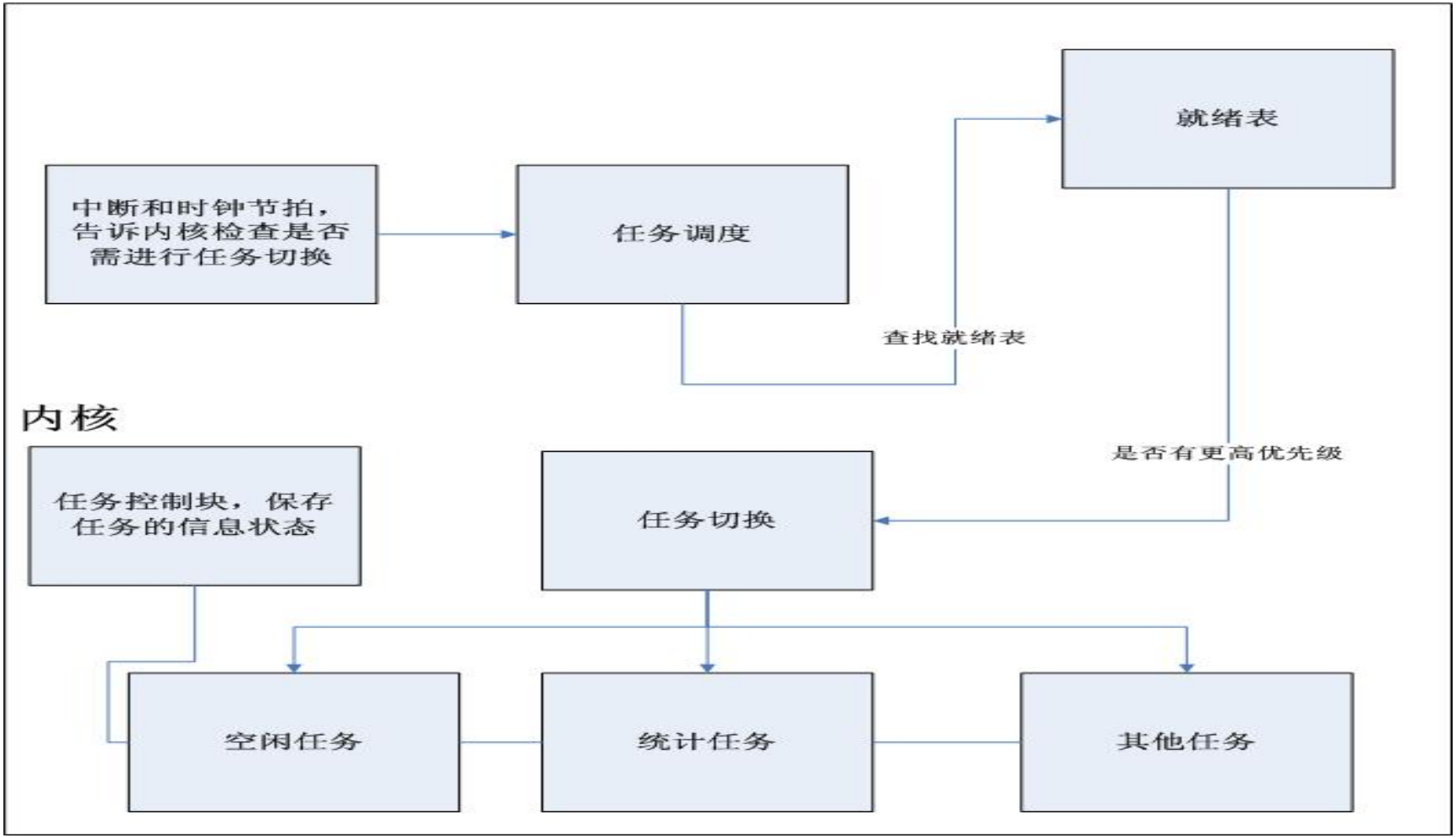
uC/OS-II移植
(与处理器相关的代码)
OS_CPU.H OS_CPU_C.C OS_CPU_A.S

μC/OS-II的内核结构

μC/OS-II内核的主要功能是任务的调度和切换

- 任务调度
- 任务切换
- 中断与时钟节拍
- 临界段代码的处理
- 统计任务和空闲任务

内核结构关系图


就绪表
中断和时钟节拍，告诉内核检查是否需进行任务切换
任务调度
查找就绪表
内核
是否有更高优先级
任务控制块，保存任务的信息状态
任务切换
空闲任务
统计任务
其他任务

任务（task）

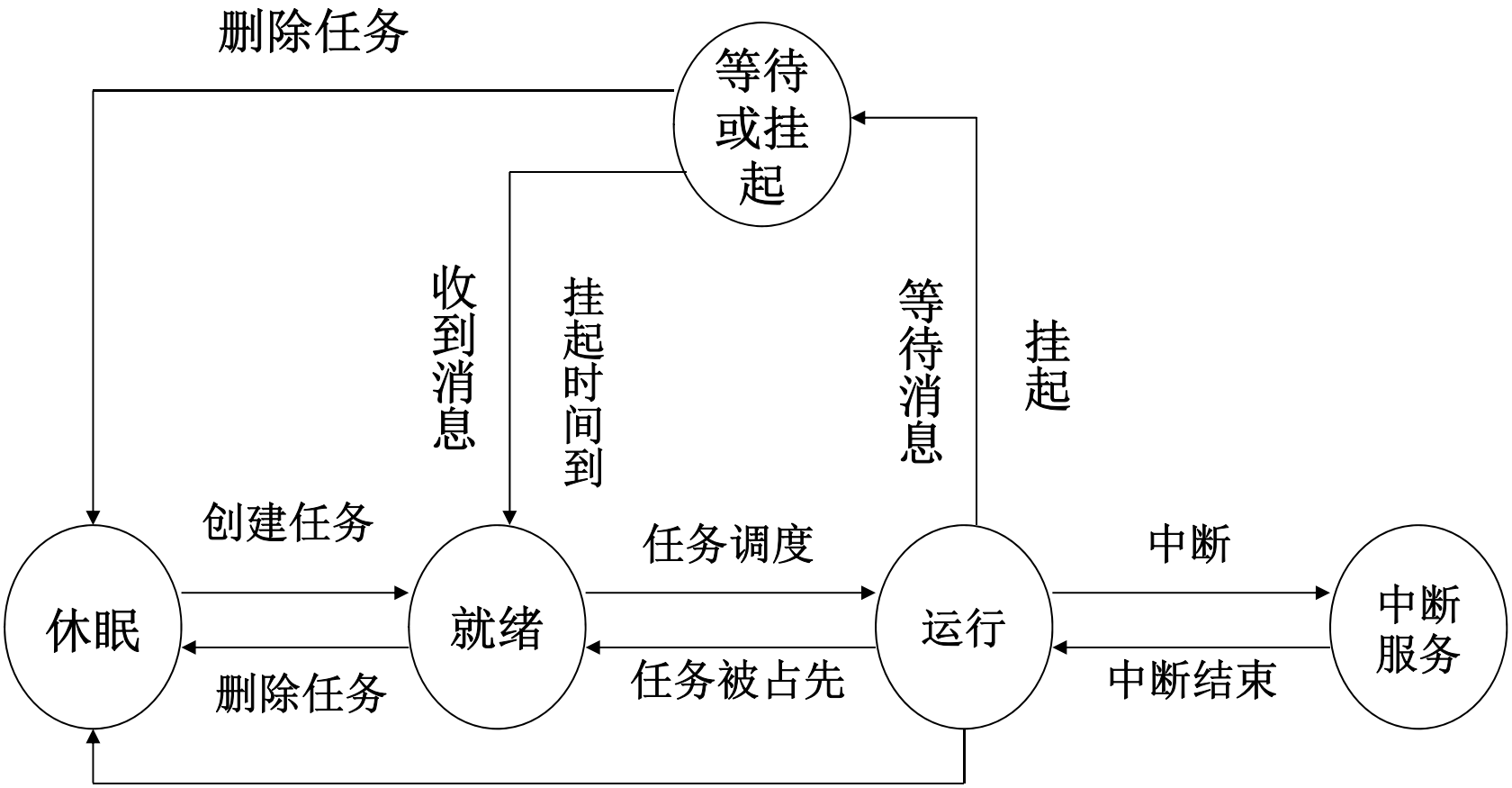
一个任务（也即线程）通常是一个无限的循环:

```
void mytask(void *pdata)
{
        do init
        while (1) {
                do something;
                waiting;
                do something;
        }
}
```

任务（task）

- **μC/OS-**Ⅱ可以管理多达**64**个任务。
- 保留了优先级为**0**、**1**、**2**、**3**、**OS_LOWEST_PRIO-3**、**OS_LOWEST_PRI0-2**，**OS_LOWEST_PRI0-1**以及**OS_LOWEST_PRI0**这**8**个任务以被将来使用。
- 用户可以有多达**56**个应用任务。必须给每个任务赋以不同的优先级。优先级号越低，任务的优先级越高。

任务状态


删除任务
等待或挂起
收到消息
挂起时间到
等待消息
挂起
创建任务
任务调度
中断
休眠
就绪
运行
中断服务
删除任务
任务被占先
中断结束

任务控制块（TCB）

- 任务控制块 **OS_TCB**是一个数据结构，保存该任务的相关参数，包括任务堆栈指针，状态，优先级，任务表位置，任务链表指针等。
- 一旦任务建立了，任务控制块**OS_TCBs**将被赋值。
- 所有的任务控制块分为两条链表，空闲链表和使用链表。

任务控制块结构的主要成员

```
OS_STK        *OSTCBStkPtr; /*当前任务栈顶的指针*/
struct os_tcb *OSTCBNext;   /*任务控制块的双重链接指针*/
struct os_tcb *OSTCBPrev;   /*任务控制块的双重链接指针*/
OS_EVENT      *OSTCBEventPtr; /*事件控制块的指针*/
void          *OSTCBMsg;    /*消息的指针*/
INT16U        OSTCBDly;     /*任务延时*/
INT8U         OSTCBStat;    /*任务的状态字*/
INT8U         OSTCBPrio;    /*任务优先级*/
INT8U         OSTCBX;       /*用于加速进入就绪态的过程*/
INT8U         OSTCBY;       /*用于加速进入就绪态的过程*/
INT8U         OSTCBBitX;    /*用于加速进入就绪态的过程*/
INT8U         OSTCBBitY;    /*用于加速进入就绪态的过程*/
```

任务创建

- 想让**μC/OS-**Ⅱ管理用户的任务，用户必须要先建立任务。
- 用户可以通过传递任务地址和其它参数到以下两个函数之一来建立任务：

  OSTaskCreate()

  OSTaskCreateExt()。
- 任务不能由中断服务程序**(ISR)**来建立。

任务调度（Task Scheduling）

- μC/OS是抢占式实时多任务内核，优先级最高的任务一旦准备就绪，则拥有CPU的所有权开始投入运行。
- μC/OS中不支持时间片轮转法，每个任务的优先级要求不一样且是唯一的，所以任务调度的工作就是：查找准备就绪的最高优先级的任务并进行上下文切换。
- μC/OS任务调度所花的时间为常数，与应用程序中建立的任务数无关。

任务调度（Task Scheduling）

- **μC/OS-**Ⅱ总是运行进入就绪态任务中优先级最高的那一个。确定哪个任务优先级最高，下面该哪个任务运行了的工作是由调度器（**Scheduler**）完成的。

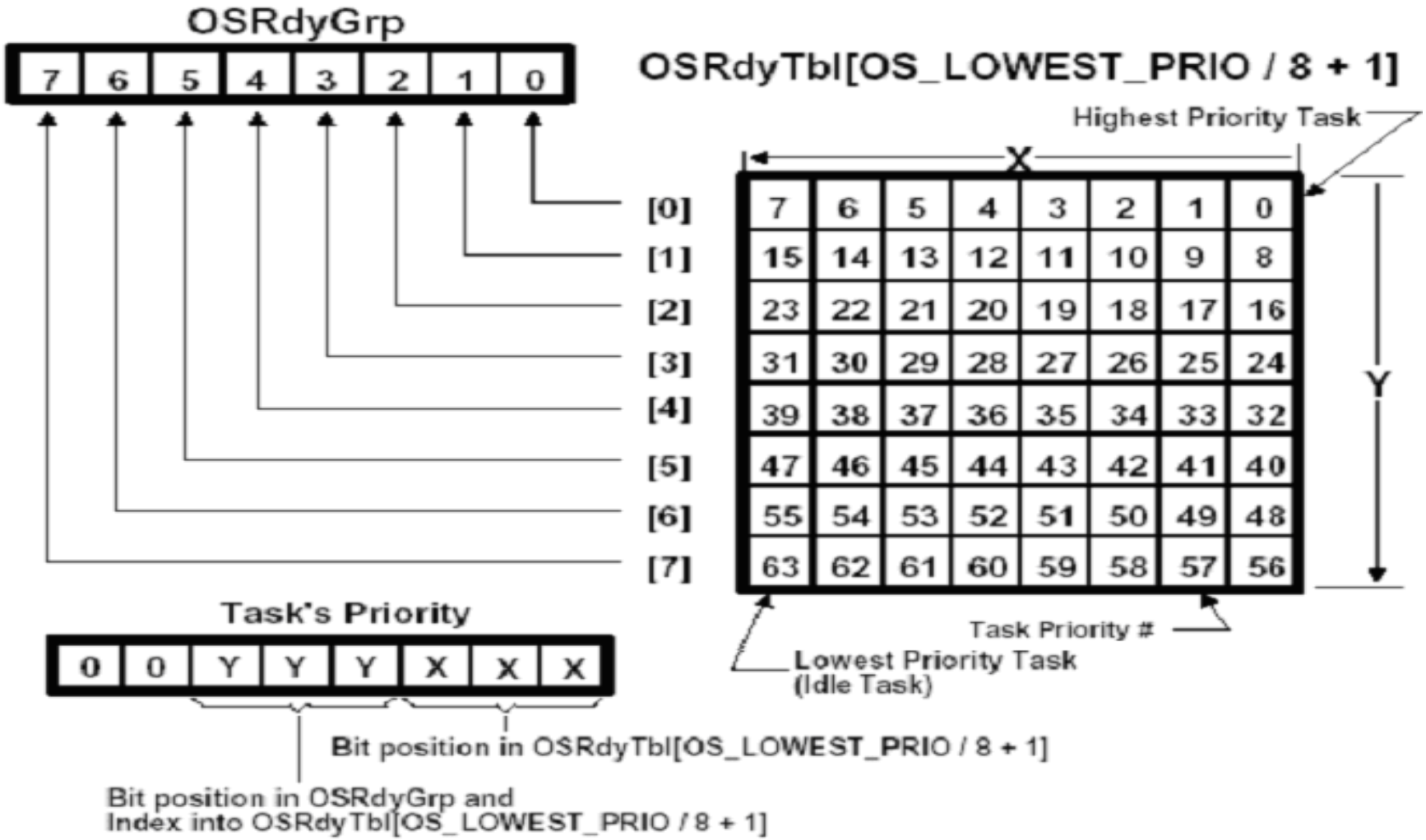
- 任务级的调度是由函数**OSSched()**完成的。中断级的调度是由另一个函数**OSIntExt()**完成的，这个函数将在以后描述。

任务就绪表（Ready List）

- 每个任务的就绪态标志都放入就绪表中的，就绪表中有两个变量**OSRdyGrp**和**OSRdyTbl[]**。

- 在**OSRdyGrp**中，任务按优先级分组，**8**个任务为一组。**OSRdyGrp**中的每一位表示**8**组任务中每一组中是否有进入就绪态的任务。任务进入就绪态时，就绪表**OSRdyTbl[]**中的相应元素的相应位也置位。

任务就绪表


OSRdyGrp
7 6 5 4 3 2 1 0
OSRdyTbl[OS_LOWEST_PRIO / 8 + 1]
Highest Priority Task
X
Y
[0] 7 6 5 4 3 2 1 0
[1] 15 14 13 12 11 10 9 8
[2] 23 22 21 20 19 18 17 16
[3] 31 30 29 28 27 26 25 24
[4] 39 38 37 36 35 34 33 32
[5] 47 46 45 44 43 42 41 40
[6] 55 54 53 52 51 50 49 48
[7] 63 62 61 60 59 58 57 56
Task's Priority
0 0 Y Y Y X X X
Task Priority #
Lowest Priority Task
(Idle Task)
Bit position in OSRdyTbl[OS_LOWEST_PRIO / 8 + 1]
Bit position in OSRdyGrp and
Index into OSRdyTbl[OS_LOWEST_PRIO / 8 + 1]

根据优先级确定就绪表（1）

- 假设优先级为12的任务进入就绪状态，12=1100b,则OSRdyTbl[1]的第4位置1，且OSRdyGrp的第1位置1，相应的数学表达式为:

  OSRdyGrp |=0x02； OSMapTbl[1]=（0000 0010）

  OSRdyTbl[1] |=0x10; OSMapTbl[4]=（0001 0000）

- 而优先级为21的任务就绪21=10 101b，则OSRdyTbl[2]的第5位置1，且OSRdyGrp的第2位置1,相应的数学表达式为：

  OSRdyGrp |=0x04； OSMapTbl[2]=（0000 0100）

  OSRdyTbl[2] |=0x20; OSMapTbl[5]=（0010 0000）

根据优先级确定就绪表（2）

从上面的计算我们可以得到:若**OSRdyGrp**及**OSRdyTbl[]**的第**n**位置**1**，则应该把**OSRdyGrp**及**OSRdyTbl[]**的值与**2∧n** 相或。该值存放在数组**OSMapTbl[7]**中,也就是：

OSMapTbl[0] =2∧0=0x01（0000 0001）

OSMapTbl[1] =2 ∧ 1=0x02（0000 0010）

…… *Index* *Bit Mask (Binary)*

0	00000001
1	00000010
2	00000100
3	00001000
4	00010000
5	00100000
6	01000000
7	10000000

使任务进入就绪态

- 如果prio是任务的优先级，也是任务的识别号，则将任务放入就绪表，即使任务进入就绪态的方法是：

```
OSRdyGrp          |=OSMapTbl[prio>>3];
OSRdyTbl[prio>>3]  |=OSMapTbl[prio & 0x07];
```

- 假设优先级为12——1100b

```
OSMapTbl[1]=00000010
OSMapTbl[4]=00010000
OSRdyGrp      |=0x02;
OSRdyTbl[1]   |=0x10;
```

使任务脱离就绪态

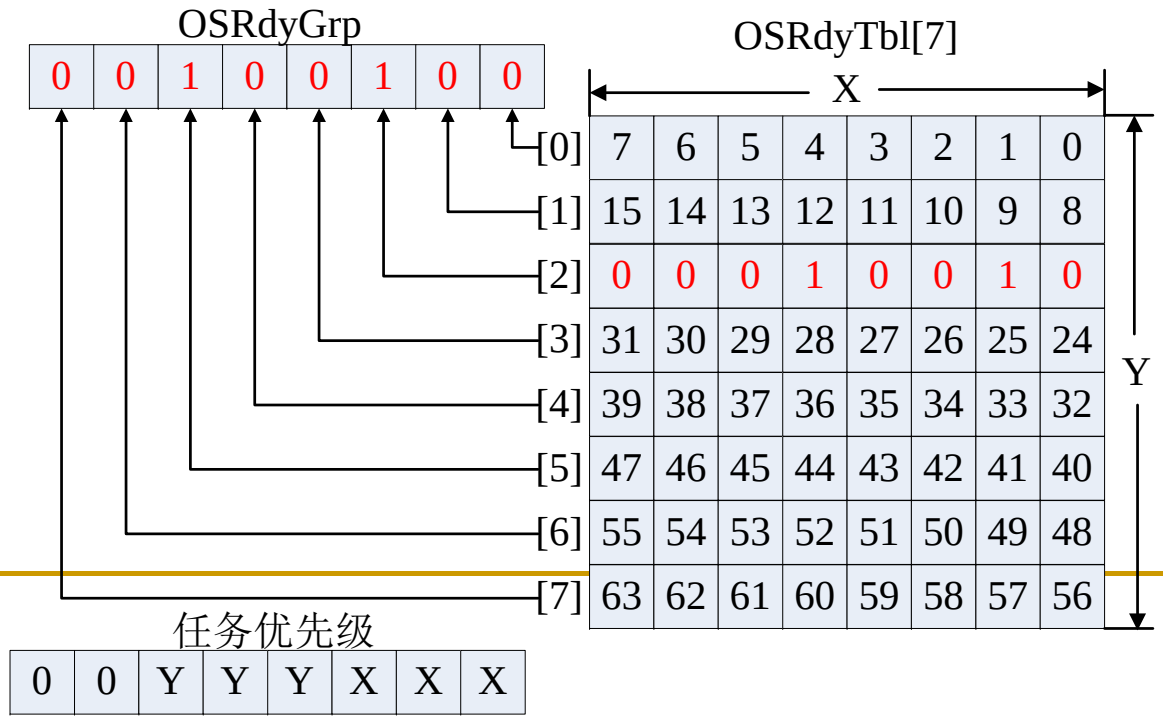
将任务就绪表**OSRdyTbl[prio>>3]**相应元素的相应位清零，而且当**OSRdyTbl[prio>>3]**中的所有位都为零时，即全组任务中没有一个进入就绪态时，**OSRdyGrp**的相应位才为零。

```
if((OSRdyTbl[prio>>3]&=
    ~ OSMapTbl[prio & 0x07])==0);
OSRdyGrp&=OSMapTbl[prio>>3];
```

根据就绪表确定最高优先级

- 通过**OSRdyGrp**值确定高**3**位，假设为**0x24=100 100b, ---〉** 对应**OSRdygGrp[2]** 和**OSRdyGrp[5]**，高优先级为**2**
- 通过**OSRdyTbl[2]**的值来确定低**3**位，

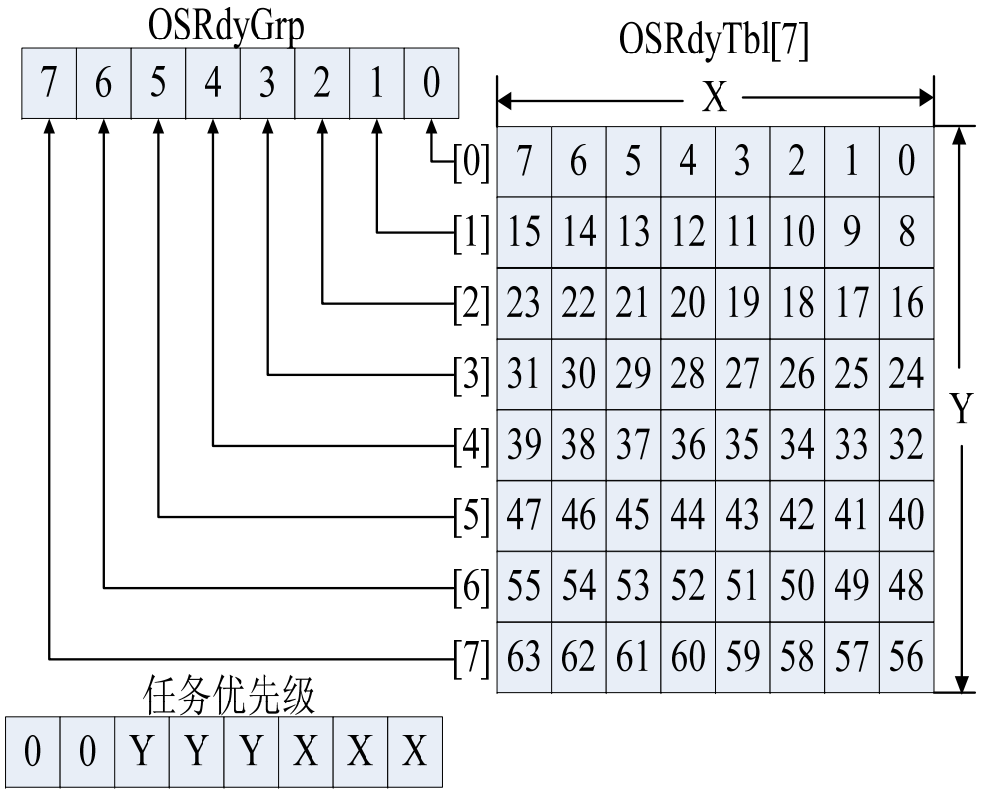
  假设为**0x12=010 010b** ，**---〉**第**2**个和第**5**个任务，取高优先级为**2**，则最高优先级的任务号为**17**



源代码中使用了查表法

查表法具有确定的时间，增加了系统的可预测性，uC/OS中所有的系统调用时间都是确定的

```
High3
=OSUnMapTbl[OSRdyGrp];
Low3
=OSUnMapTbl[OSRdyTbl[High3]];
Prio  =(High3<<3)+Low3;
```



优先级判定表OSUnMapTbl[256]（os_core.c）

如OSRdyGrp的值为00101000B，即0X28，则查得OSUnMapTbl[OSRdyGrp]的值是3，它相应于OSRdyGrp中的第3位置1；

如OSRdyTbl[3]的值是11100100B，即0XE4，则查OSUnMapTbl[OSRdyTbl[3]]的值是2，则进入就绪态的最高任务优先级

Prio=3*8+2=26

```
INT8U const OSUnMapTbl[] = {
    0, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    7, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0
};
```

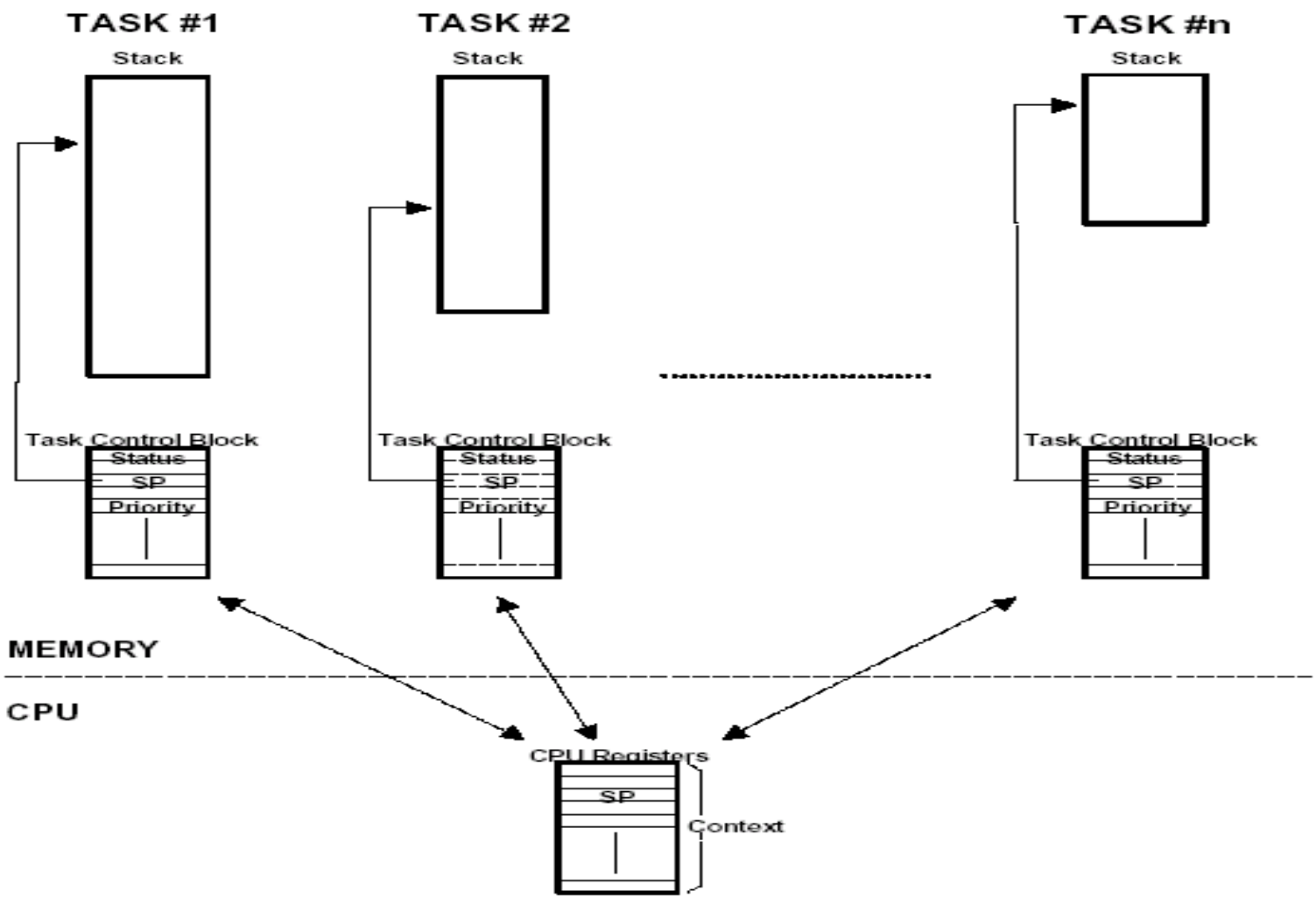
任务调度器

```
void OSSched (void)
{   INT8U y;
    OS_ENTER_CRITICAL();
   if ((OSLockNesting | OSIntNesting) == 0) {
      y            = OSUnMapTbl[OSRdyGrp];
      OSPrioHighRdy = (INT8U)((y << 3) +
                                OSUnMapTbl[OSRdyTbl[y]]);
      if (OSPrioHighRdy != OSPrioCur) {
         OSTCBHighRdy = OSTCBPrioTbl[OSPrioHighRdy];
         OSCtxSwCtr++;
         OS_TASK_SW();
       }
   }
   OS_EXIT_CRITICAL();  }
```

任务切换

- 将被挂起的任务寄存器入栈
- 将较高优先级任务的寄存器出栈

任务切换示意


TASK #1
Stack
TASK #2
Stack
TASK #n
Stack
Task Control Block
Status
SP
Priority
Task Control Block
Status
SP
Priority
Task Control Block
Status
SP
Priority
MEMORY
CPU
CPU Registers
SP
Context

任务级的任务切换OS_TASK_SW()

- 通过系统调用指令完成
- 保护当前任务的现场
- 恢复新任务的现场
- 执行中断返回指令
- 开始执行新的任务

统计任务和空闲任务

- 空闲任务**OS_TaskIdle( )**，这个任务总是处于就绪态。空闲任务**OS_TaskIdle( )**的优先级总是设成最低，即**OS_LOWEST_PRIO**。
- 统计任务**OS_TaskStat( )**，统计**CPU**的运行时间并且使其进入就绪态。**OS_TaskStat( )**的优先级总是设为**OS_LOWEST_PRIO -1**。

中断与时钟节拍

- 当发生中断时，首先应保护现场，将**CPU**寄存器入栈，再处理中断函数，然后恢复现场，将**CPU**寄存器出栈，最后执行中断返回。

- **uC/OS**中提供了**OSIntEnter()**和**OSIntExit()**告诉内核进入了中断状态。

- 时钟节拍是一种特殊的中断，操作系统的心脏。对任务列表进行扫描，判断是否有延时任务应该处于准备就绪状态，最后进行上下文切换。

μC/OS中的中断处理

- μC/OS中，中断服务子程序要用汇编语言来写。然而，如果用户使用的C语言编译器支持在线汇编语言的话，用户可以直接将中断服务子程序代码放在C语言的程序文件中。
- 用户中断服务子程序框架:

① 保存全部CPU寄存器;

② 调用OSIntEnter或OSIntNesting直接加1；

③ 执行用户代码做中断服务;

④ 调用OSIntExit()；

⑤ 恢复所有CPU寄存器；

⑥ 执行中断返回指令；

OSIntEnter()

```
void OSIntEnter (void)
{
    OS_ENTER_CRITICAL();
    OSIntNesting++;
    OS_EXIT_CRITICAL();
}
```

OSIntExit

```
OS_ENTER_CRITICAL();
if ((--OSIntNesting | OSLockNesting) == 0) {
      OSIntExitY    = OSUnMapTbl[OSRdyGrp];
      OSPrioHighRdy = (INT8U)((OSIntExitY << 3) +
                OSUnMapTbl[OSRdyTbl[OSIntExitY]]);
    if (OSPrioHighRdy != OSPrioCur) {
          OSTCBHighRdy=OSTCBPrioTbl[OSPrioHighRdy];
         OSCtxSwCtr++;
         OSIntCtxSw(); }
}
OS_EXIT_CRITICAL();
```

时钟节拍

- **μC/OS**需要用户提供周期性信号源，用于实现时间延时和确认超时。节拍率应在说**10**到**100Hz**。时钟节拍率越高，系统的额外负荷就越重。

- 时钟节拍的实际频率取决于用户应用程序的精度。时钟节拍源可以是专门的硬件定时器，也可以是来自**50/60Hz**交流电源的信号。

- 用户必须在多任务系统启动以后再开启时钟节拍器，也就是在调用**OSStart()**之后。

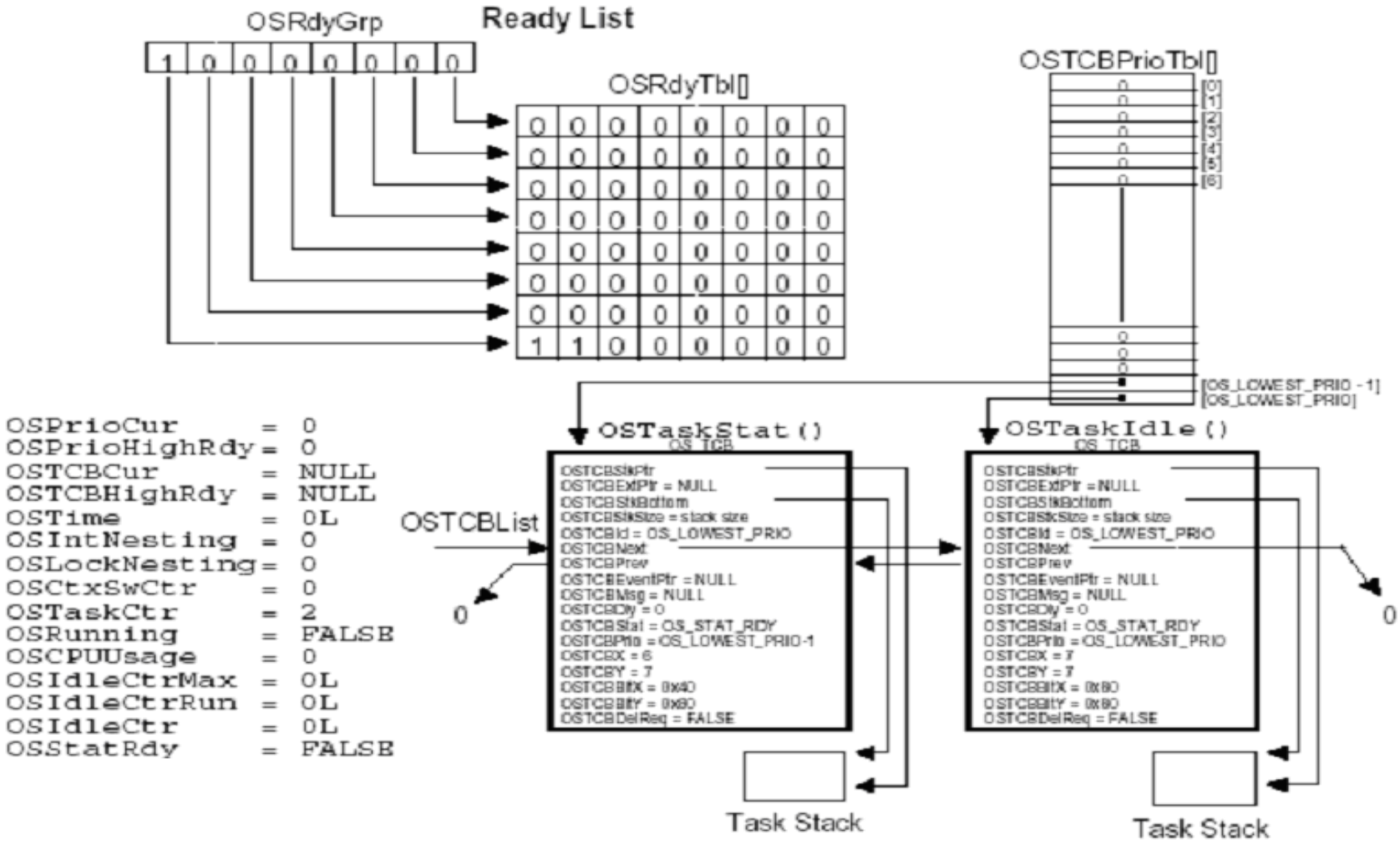
OSTickISR

```
void OSTickISR(void)
{
    保存处理器寄存器的值;
    调用OSIntEnter()或是将OSIntNesting加1;
    调用OSTimeTick();
     调用OSIntExit();
    恢复处理器寄存器的值;
    执行中断返回指令;
}
```

μC/OS-Ⅱ初始化

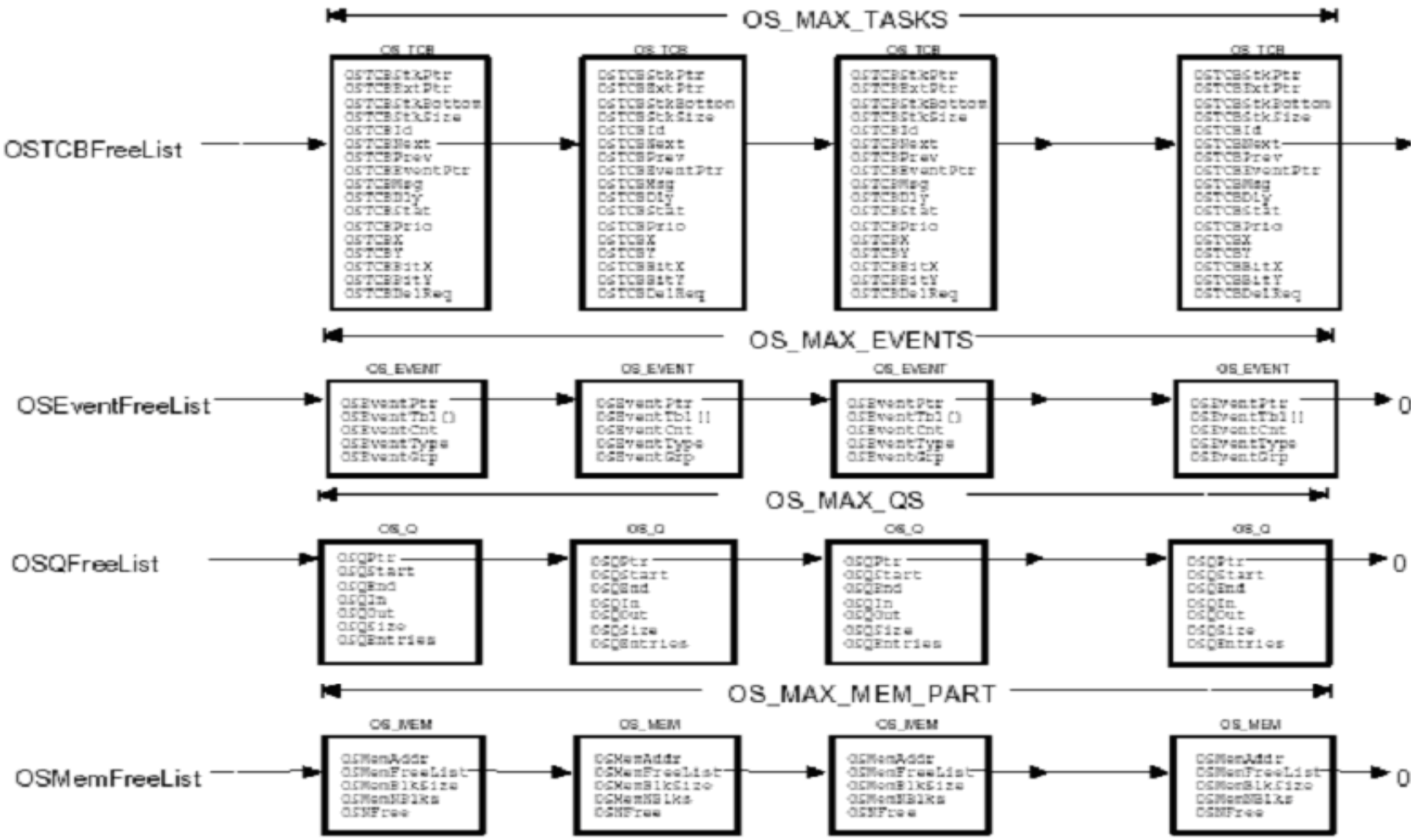
- 在调用**μC/OS-**Ⅱ的任何其它服务之前，**μC/OS-**Ⅱ要求用户首先调用系统初始化函数**OSInit()**。
- **OSInit()**建立空闲任务**idle task**，这个任务总是处于就绪态的。空闲任务**OSTaskIdle**（）的优先级总是设成最低，即**OS_LOWEST_PRIO**。
- **μC/OS-**Ⅱ还初始化了**4**个空数据结构缓冲区。

μC/OS-Ⅱ初始化后的一些数据结构内容



μC/OS-II的内核结构

μC/OS-Ⅱ初始化后的缓冲区



μC/OS-II的内核结构

μC/OS-Ⅱ的启动

- 多任务的启动是用户通过调用**OSStart()**实现的。然而，启动**μC/OS-Ⅱ**之前，用户至少要建立一个应用任务。

OSInit(); /* 初始化**uC/OS-II***/

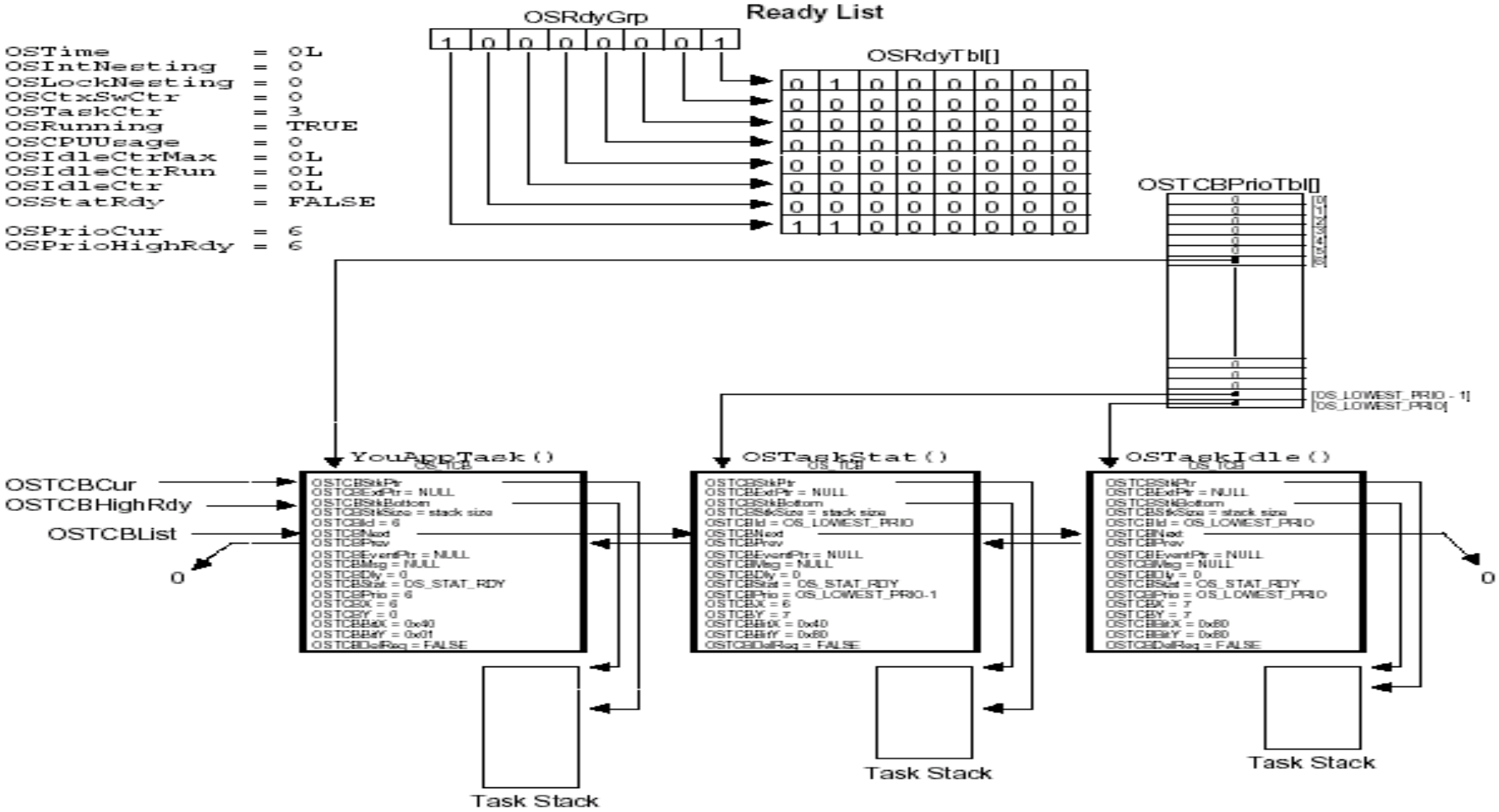
……

调用**OSTaskCreate()**或**OSTaskCreateExt()**；

……

OSStart(); /*开始多任务调度 */

OSStart()从任务就绪表中找出用户建立的优先级最高的任务的任务控制块，再调用高优先级就绪任务启动函数**OSStartHighRdy()**（此函数的功能是将任务栈中保存的值弹回到**CPU**寄存器中，然后执行一条中断返回指令，中断返回指令强制执行该任务代码。



OSStart

```
if (OSRunning == FALSE) {
    y             = OSUnMapTbl[OSRdyGrp];
    x             = OSUnMapTbl[OSRdyTbl[y]];
    OSPrioHighRdy = (INT8U)((y << 3) + x);
    OSPrioCur     = OSPrioHighRdy;
    OSTCBHighRdy  = OSTCBPrioTbl[OSPrioHighRdy];

    OSTCBCur      = OSTCBHighRdy;
    OSStartHighRdy();
}
```

临界段代码的处理

- 同其他的内核一样，μC/OS-II为了处理临界段代码，须关中断，处理完毕，后再开中断。关中断的时间是设计实时内核的一个重要的指标，但就μC/OS-Ⅱ而言，其时间却很大程度上取决于微处理器的结构以及编译器所生成的代码质量。
- 微处理器一般都有关中断/开中断指令，用户使用的C语言编译器必须具有某种机制能够在C中直接实现关中断/开中断。为了方便移植，定义了两个宏来开/关中断（OS_ENTER_CRITICAL()和OS_EXIT_CRITICAL()）。这两个宏的代码是不同的移植对象而作相应的定义（OS_CPU.H中定义）

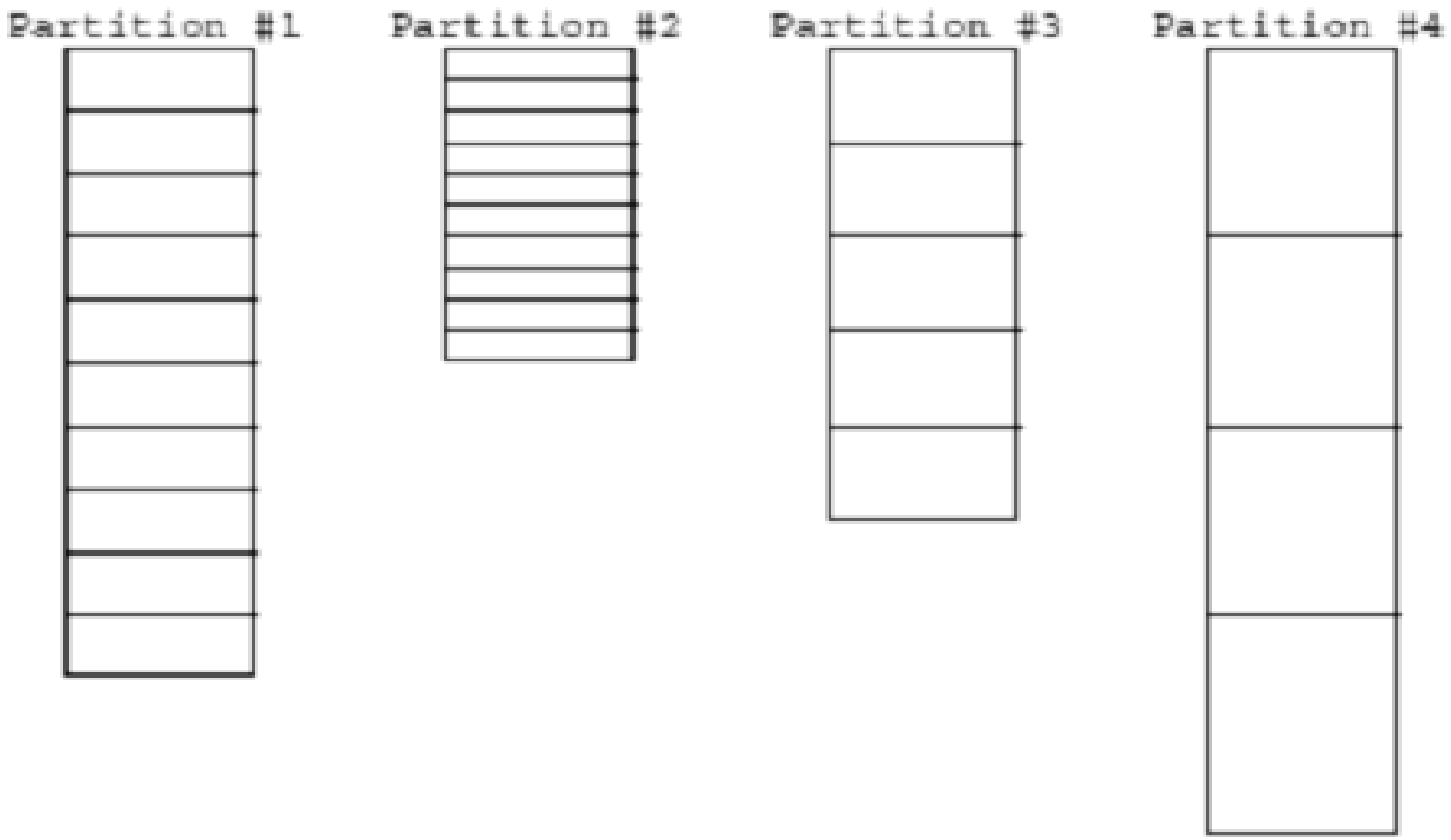
μC/OS-II的其他机制

- 内存管理
- 时间管理
- 任务间的同步——信号量管理
- 任务间的同步——互斥型信号量管理
- 任务间的同步——事件标志组的管理
- 任务间的通信——消息邮箱管理
- 任务间的同步——消息队列管理

内存管理

- 在**ANSI C**中可以用**malloc()**和**free()**两个函数动态地分配内存和释放内存。在嵌入式实时操作系统中，容易产生内存碎片。
- **μC/OS-II**中，操作系统把连续的大块内存按分区来管理。每个分区中包含有整数个大小相同的内存块。
- 在一个系统中可以有多个内存分区。这样，用户的应用程序就可以从不同的内存分区中得到不同大小的内存块。但是，特定的内存块在释放时必须重新放回它以前所属于的内存分区。

内存分区示意图



内存控制块

为了便于内存的管理，在**μC/OS-II**中使用内存控制块（**memory control blocks**）的数据结构来跟踪每一个内存分区，系统中的每个内存分区都有它自己的内存控制块。

```
typedef struct {
    void   *OSMemAddr;      /*分区起始地址*/
    void   *OSMemFreeList;          /*下一个空闲内存块*/
    INT32U  OSMemBlkSize;           /*内存块的大小*/
    INT32U  OSMemNBlks; /*内存块数量*/
    INT32U  OSMemNFree; /*空闲内存块数量 */
} OS_MEM;
```

内存管理初始化

如果要在**μC/OS-II**中使用内存管理，需要在**OS_CFG.H**文件中将开关量**OS_MEM_EN**设置为**1**。这样**μC/OS-II** 在启动时就会对内存管理器进行初始化 （**OSMemInit()** ）。

建立一个内存分区，OSMemCreate()

- 在使用一个内存分区之前，必须使用**OSMemCreate()**先建立该内存分区。该函数共有**4**个参数：内存分区的起始地址、分区内的内存块总块数、每个内存块的字节数和一个指向错误信息代码的指针。
- 每个内存分区必须含有至少两个内存块，每个内存块至少为一个指针的大小 。
  - **OS_MEM *CommTxBuf;**
  - **INT8U CommTxPart[100][32];**
  - **CommTxBuf = OSMemCreate(CommTxPart, 100, 32, &err);**

分配一个内存块，OSMemGet()

- 调用**OSMemGet()**函数从已经建立的内存分区中申请一个内存块。该函数的唯一参数是指向特定内存分区的指针，该指针在建立内存分区时，由**OSMemCreate()**函数返回。
- 注意的是，用户可以在中断服务子程序中调用**OSMemGet()**，因为在暂时没有内存块可用的情况下，**OSMemGet()**不会等待，而是马上返回**NULL**指针。

释放一个内存块，OSMemPut()

- 应用程序不再使用一个内存块时，必须及时地把它释放并放回到相应的内存分区中。这个操作由**OSMemPut()**函数完成。
- 必须注意的是，**OSMemPut()**并不知道一个内存块是属于哪个内存分区的。 释放内存块时必须将它释放到正确的分区。

时间管理

- **μC/OS-**Ⅱ**(**其它内核也一样**)**要求用户提供定时中断来实现延时与超时控制等功能。这个定时中断叫做时钟节拍，它应该每秒发生**10**至**100**次。时钟节拍的频率越高，系统的负荷就越重。
- 与时钟管理相关的系统服务有：
  - OSTimeDLY()
  - OSTimeDLYHMSM()
  - OSTimeDlyResmue()
  - OStimeGet()
  - OSTimeSet()

任务间通信手段

- μC/OS中，采用多种方法保护任务之间的共享数据和提供任务之间的通信。
  - 提供**OS_ENTER_CRITICAL**和**OS_EXIT_CRITICAL**来对临界资源进行保护
  - **OSSchedLock( )**禁止调度保护任务级的共享资源。
  - 提供了经典操作系统任务间通信方法：信号量、邮箱、消息队列，事件标志。

事件控制块ECB

所有的通信信号都被看成是事件(event), 一个称为事件控制块(ECB, Event Control Block)的数据结构来表征每一个具体事件，ECB的结构如下：

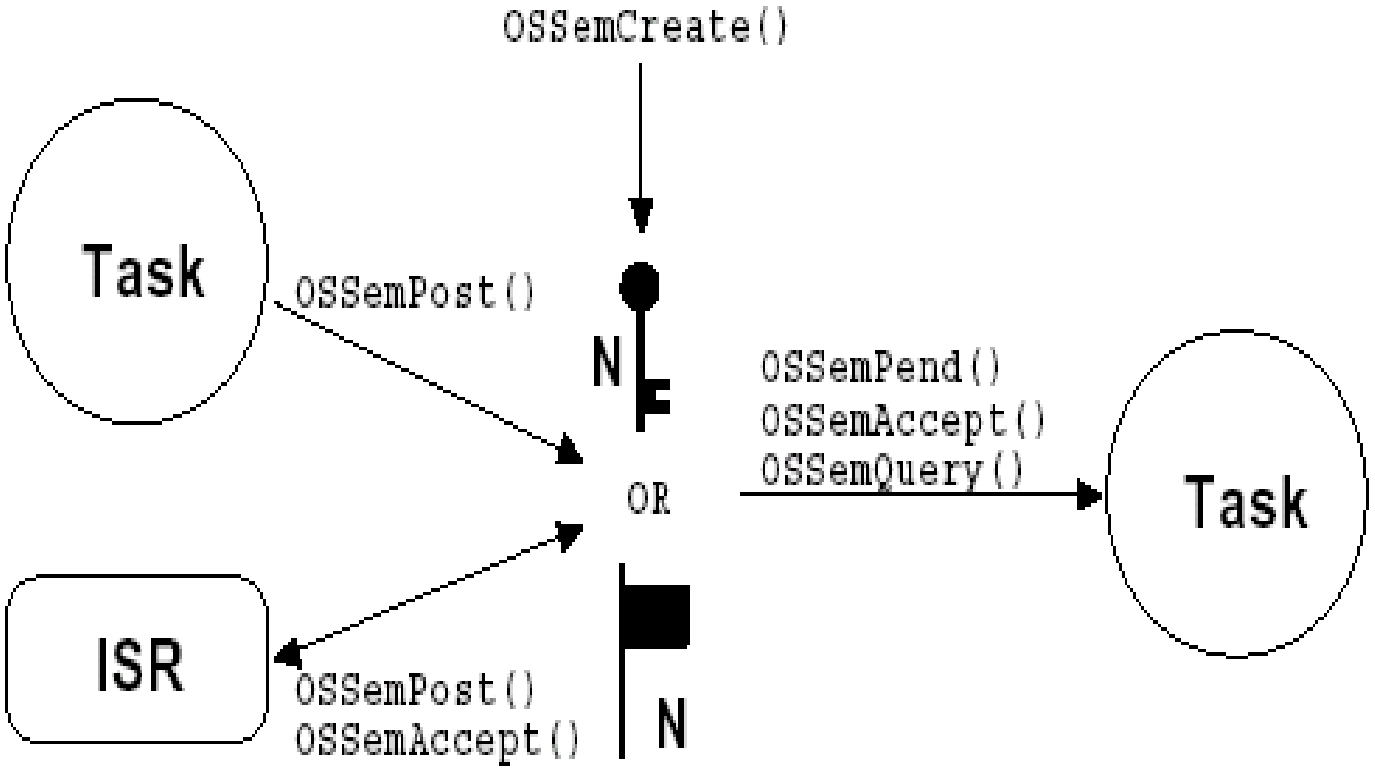
```
typedef struct {
    void    *OSEventPtr; /*指向消息或消息队列的指针*/
    INT8U   OSEventTbl[OS_EVENT_TBL_SIZE]; /*等待任务列表*/
    INT16U  OSEventCnt; /*计数器（事件是信号量时）*/
    INT8U   OSEventType; /*事件类型：信号量、邮箱等*/
    INT8U   OSEventGrp; /*等待任务组*/
  } OS_EVENT;
```

与TCB类似的结构，使用两个链表，空闲链表与使用链表

信号量semaphore

- 信号量在多任务系统中用于：控制共享资源的使用权、标志事件的发生、使两个任务的行为同步。
- **uC/OS**中信号量由两部分组成：信号量的计数值和等待该信号任务的等待任务表。信号量的计数值可以为二进制, 也可以是其他整数。
- 系统通过**OSSemPend( )**和**OSSemPost( )**来支持信号量的两种原子操作**P()**和**V()**。**P()**操作减少信号量的值，如果新的信号量的值不大于**0**,则操作阻塞；**V()**操作增加信号量的值。

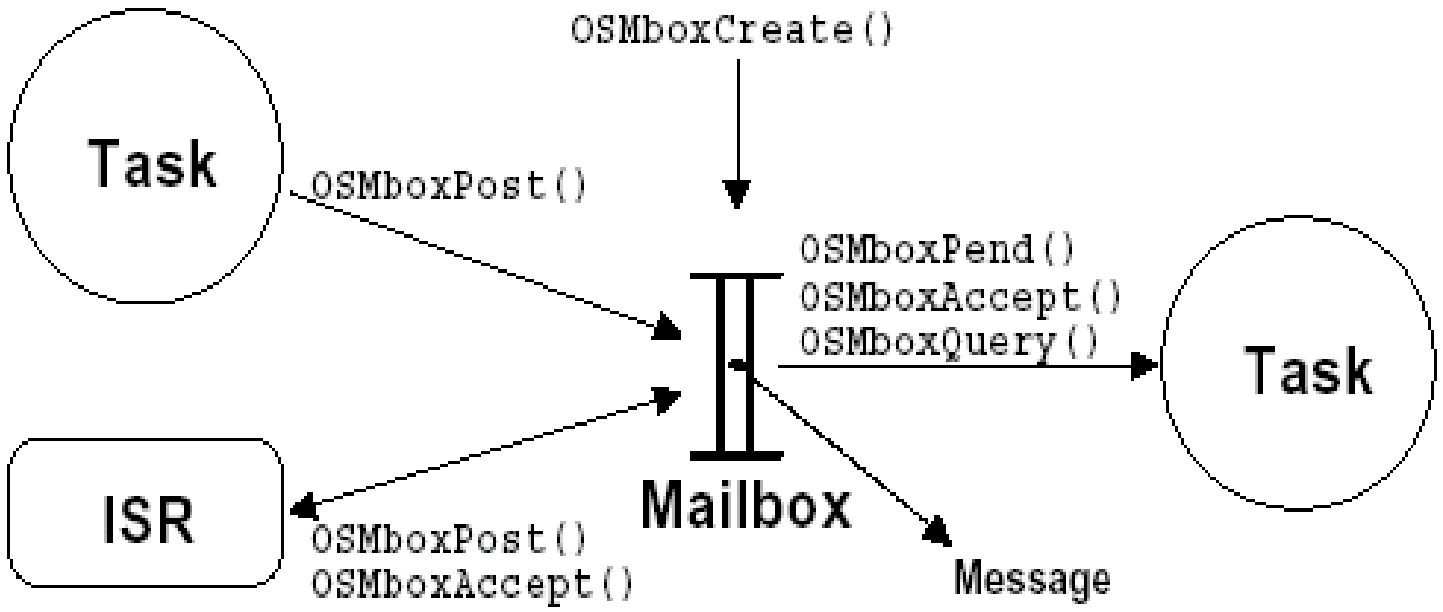
任务、中断服务子程序和信号量之间的关系



消息邮箱管理

- 消息邮箱管理用于任务的互相传送消息。其中包括建立、删除一个邮箱，等待邮箱中的消息，向邮箱发送一则消息，无等待地从邮箱中得到一则消息，查询一个邮箱的状态。

任务、中断服务子程序和邮箱之间的关系



μCOS-II的移植

所谓移植，是指使一个实时操作系统能够在某个微处理器平台上运行。μCOS-II的主要代码都是由标准的C语言写成的，移植方便。

移植μCOS-II满足的条件

- 处理器的C编译器能产生可重入代码
- 用C语言可以打开或者关闭中断
- 处理器支持中断，并且能产生定时中断（通常在10—1000Hz之间）
- 处理器支承能够容纳一定量数据的硬件堆栈
- 处理器有将堆栈指针和其他CPU寄存器存储和读出到堆栈（或者内存）的指令

可重入代码指的是可以被多个任务同时调用，但不会破坏数据的一段代码，或者说代码具有在执行过程中打断后再次被调用的能力。μC/OS-Ⅱ是多任务内核，函数可能会被多个任务调用。所以要完成多任务，代码的可重入性是基础。

```
int temp;
void swap (int *x,int*y)
{
        temp=*x;
        *x=*y;
        *y=temp;
}
```

```
void swap (int *x,int*y)
{
        int temp;
        temp=*x;
        *x=*y;
        *y=temp;
}
```

上图列举的两个函数的区别在于左边temp作为全局变量存在，右边函数中temp作为函数的局部变量存在，因此左边的函数是不可重入的，而右边的函数是可以重入的。

用C语言可以打开/关闭中断

- 在μCOS-II中，可以通过：

  OS_ENTER_CRITICAL ()

  OS_EXIT_CRITICAL()

  宏来控制系统关闭或者打开中断。这需要处理器的支持。

- 在ARM7TDMI的处理器上，可以设置相应的寄存器来关闭或者打开系统的所有中断。

```
#define OS_ENTER_CRITICAL() ARMDisableInt()

ARMDisableInt:
        mrs     r12, CPSR           /* 获取模式寄存器 */
        orr     r12, r12, #I_BIT    /* 设置禁止中断位 */
        msr     CPSR_c, r12         /* 设置模式寄存器 */
        bx      lr
```

处理器支持中断并且能产生定时中断

- μCOS-II是通过处理器产生的定时器的中断来实现多任务之间的调度的。ARM7TDMI的处理器上可以产生定时器中断。
- 本系统工作在60MHz的主频下，定时器的中断的频率为100Hz。也就是系统的响应时间为10ms。

处理器支持硬件堆栈

- μCOS-II进行任务调度的时候，会把当前任务的CPU寄存器存放到此任务的堆栈中，然后，再从另一个任务的堆栈中恢复原来的工作寄存器，继续运行另一个任务。所以，寄存器的入栈和出栈是μCOS-II多任务调度的基础。
- ARM7处理器中有专门的指令处理堆栈，可以灵活的使用堆栈。

ARM处理器中汇编指令stmfd可以将所有寄存器压栈，对应也有一个出栈的指令ldmfd。

μC/OS-II的具体移植的代码修改

- 设置OS_CPU.H中与处理器和编译器相关的代码
- 用C语言编写六个操作系统相关的函数（OS_CPU_C.C）
- 用汇编语言编写四个与处理器相关的函数（OS_CPU.ASM）

设置与处理器和编译器相关的代码

- OS_CPU.H中定义了与编译器相关的数据类型。比如：INT8U、INT8S等。
- 与 ARM处理器相关的代码，使用 OS_ENTER_CRITICAL() 和 OS_EXIT_CRITICAL() 宏开启 / 关闭中断
- 设施堆栈的增长方向 ：堆栈由高地址向低地址增长

用汇编语言编写四个与处理器相关的函数

- OSStartHighRdy()

OSStart( )调用本函数，OSStart( )负责使就绪状态的任务开始运行;OSStartHighRdy( )负责获取新任务的堆栈指针并从堆栈指针中恢复新任务的所有处理器寄存器。

- OSCtxSw()

OS_TASK_SW宏调用本函数;OS_TASK_SW由OSSched()调用，负责任务之间的切换; OSCtxSw( )负责保存当前任务对应的处理器寄存器到堆栈中，并将需要恢复的任务对应的处理器寄存器从堆栈中恢复出来。

- OSIntCtxSw()

OSIntExit( )调用本函数; OSIntExit由OSTickISR调用，负责在定时中断中任务之间的切换;OSIntCtxSw( )主要保存当前任务堆栈指针，并将新任务对应的处理器寄存器从堆栈中恢复出来。

- OSTickISR()

定时中断函数; OSTickISR( )主要负责在进入时保存处理器寄存器，完成任务的切换，退出时恢复寄存器并返回。

用C语言编写六个操作系统相关的函数

- void *OSTaskStkInit (void (*task)(void *pd),void *pdata, void *ptos, INT16U opt)

任务创建时调用本函数; OSTaskStkInit( ) 负责初始化任务的堆栈结构;

- void OSTaskCreateHook (OS_TCB *ptcb)
- void OSTaskDelHook (OS_TCB *ptcb)
- void OSTaskSwHook (void)
- void OSTaskStatHook (void)
- void OSTimeTickHook (void)

后5个函数为钩子函数，可以不加代码

```
void*OSTaskStkInit (void(*task)(void*pd),void*pdata,void*ptos,INT16U opt）
 {
unsigned Int*stk：
opt＝opt；  / *prevent warning* /
stk＝（unsigned int*）ptos；  / *Load stackpoInter* /
 / *build a context for the new task* /
*--stk＝（unsigned nit）task；  / *pc* /
*--stk＝（unsigned nit）task；  / *lr* /
*--stk ＝0；  / *rl2* /
*--stk＝0；  / *r11* /
*--stk＝0；  / * r10* /
*--stk＝0；  / *r9* /
*--stk＝0；  / *r8 * /
*--stk＝0；  / *r7 * /
*--stk＝0；  / *r6* /
*--stk＝0；  / *r5* /
*--stk＝0；  / *r4* /
*--stk＝0；  / *r3* /
*--stk＝0；  / *r2* /
*--stk＝0；  / *rl* /
*--stk＝（unsigned nit）pdata；  / *r0* /
*--stk＝（SVC32MODE| 0x0）；  / *cpsrIRQ，FIQ  disable* /
*--stk＝（SVC32MODE|0xo）；  / *spsrIRQ，FIQ disable * /
return（（void*）stk）；}
```

移植要点

- 定义函数OS_ENTER_CRITICAL和OS_ENTER_CRITICAL。
- 定义函数OS_TASK_SW执行任务切换。
- 定义函数OSCtxSw实现用户级上下文切换，用纯汇编实现。
- 定义函数OSIntCtxSw实现中断级任务切换，用纯汇编实现。
- 定义函数OSTickISR。
- 定义OSTaskStkInit来初始化任务的堆栈。

uC/OS的改进

- 固定的基于优先级的调度，不支持时间片，使用起来不方便。一个任务的基础上增加一个基于时间片的微型调度核
- 在对临界资源的访问上使用关闭中断实现，没有使用CPU提供的硬件指令，例如测试并置位。
- 系统时钟中断，没有提供用户使用定时器，可以借鉴linux的定时器加以修改
- 可以加上文件系统和TCP/IP协议栈

基于uC/OS的应用系统构建

外设的初始化 一串口

```
typedef volatile struct
  {
  volatile unsigned int comdata;
  volatile unsigned int comstatus;
  volatile unsigned int comcontrol;
  volatile unsigned int comscaler;
  } np_com;
np_com      *pOSSPARCCOM;
```

串口的初始化

串口初始化为中断式的，可接受发送数据的，8位数据位，1位停止位，无校验位的模式，设置如下
OS_SPARC_COM2_ADDRESS为串口基地址）：

```
pOSSPARCCOM=(np_com*)OS_SPARC_COM2_ADDRESS;
pOSSPARCCOM->comcontrol = 0x00000007;
```

串口的波特率设置为115200，设置如下：

```
pOSSPARCCOM->comscaler  = 0x00000043;
```

串口的驱动程序设计

串口程序主要是从串口接收命令，然后给控制任务发送消息，并在LCD上显示相关信息，设计如下：

```
void OSCOMHandle(void)
{           INT8U err;
            char ch;
            ch = (unsigned char)pOSSPARCCOM->comdata;
            if(ch!='\n')
            {               if(ch!='\r')            //while press enter
              {               temp[charcnt] = ch;
               write_char(temp[charcnt]);      // display on LCD
               charcnt++;
      }     }
            else
            {temp[charcnt]='\0';
            charcnt=0;
            err = OSMboxPost(charMbox,(void *)1); //send the command to the control task
  }}
```

文件系统简介

文件系统采用可配置的μC/FS评估版。

- 文件系统的定义

处理文件的操作系统部分称为文件系统.是操作系统中统一管理信息资源的一种软件，管理文件的存储、检索、更新，提供安全可靠的共享和保护手段，并且方便用户使用。

- 文件系统的功能

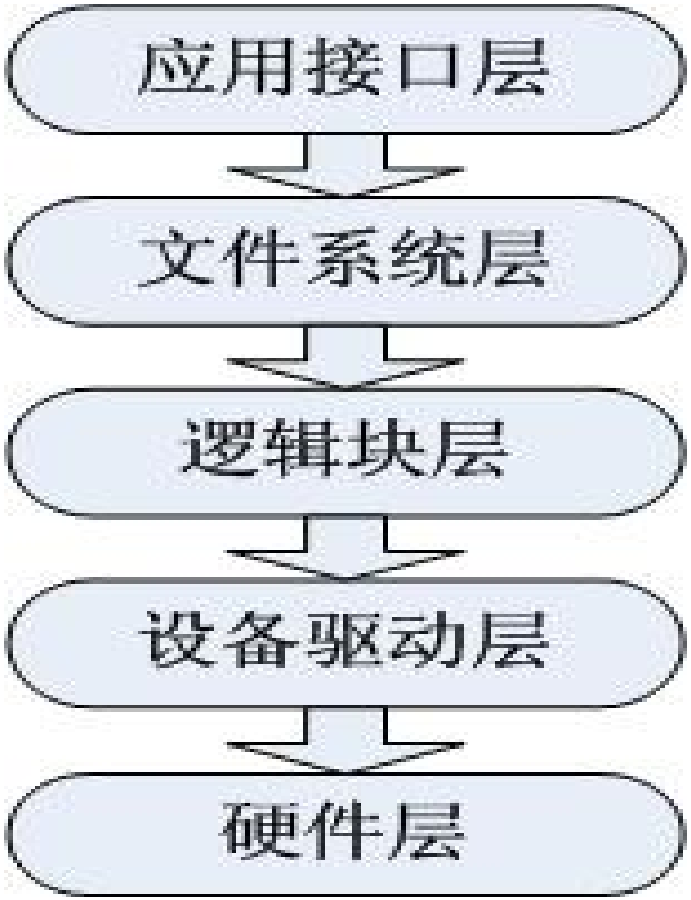
文件的创建、删除、更名，文件目录的创建、删除、更名、显示当前目录的内容。

- 文件系统的存储体

本例子中采用的是4GB大小的CF卡作为存储体

文件系统的结构层次

文件系统为FAT32，每簇4KB。文件系统总共分为五个层：应用接口层，文件系统层，逻辑块层，设备驱动层，硬件层。其中后两层可合为一层，下层为上层提供服务，上层逐级调用下层来访问实际存储体中的数据。



1、应用接口层

向应用程序提供对文件操作的接口，例如对文件的打开、读/写、删除/创建，对目录的打开、读/写、删除创建等。

2、文件系统层

把上层对文件的操作转换为对逻辑块的操作，这当中包含了读取文件系统的相关信息（比如读取FAT表，目录表用以查找文件所在的逻辑块）。

3、逻辑块层

此层的目的主要是为了同步对于存储设备的操作，以及向上层提供一个简单的接口，这层将调用下层对设备产生一个块操作。

4、设备驱动层

驱动设备进行工作，计算分区首地址，向下层传递逻辑块号。

5、硬件层

对设备进行块的读/写操作（每个块的大小512 byte）。

CF卡的访问规则

1、MBR区（主引导扇区）

主引导记录区位于整个硬盘的0磁道0柱面1扇区。不过，在总共512字节的主引导扇区中，MBR只占用了其中的446个字节（偏移0--偏移1BDH），另外的64个字节（偏移1BEH--偏移1FDH）交给了DPT(Disk Partition Table硬盘分区表), 最后两个字节“55，AA”（偏移1FEH，偏移1FFH）是分区的结束标志。这个整体构成了硬盘的主引导扇区。

CF卡的访问规则

主引导记录中包含了硬盘的一系列参数和一段引导程序。其中的硬盘引导程序的主要作用是检查分区表是否正确并且在系统硬件完成自检以后引导具有激活标志的分区上的操作系统，并将控制权交给启动程序。MBR是由分区程序（如Fdisk.com）所产生的，它不依赖任何操作系统，而且硬盘引导程序也是可以改变的，从而实现多系统共存。

CF卡的访问规则

2、DBR（Dos Boot Record）

操作系统引导记录区。它通常位于硬盘的0磁道1柱面1扇区，是操作系统可以直接访问的第一个扇区，它包括一个引导程序和一个被称为BPB（Bios Parameter Block）的本分区参数记录表。引导程序的主要任务是当MBR将系统控制权交给它时，判断本分区跟目录前两个文件是不是操作系统的引导文件。如果确定存在，就把其读入内存，并把控制权 交给该文件。BPB参数块记录着本分区的起始扇区、结束扇区、文件存储格式、硬盘介质描述符、根目录大小、FAT个数，分配单元的大小等重要参数。

CF卡的访问规则

3、FAT 区

在DBR之后的是FAT（File Allocation Table文件分配表）区。文件占用磁盘空间时，基本单位是簇。簇的大小与磁盘的规格有关，每个簇可包含4、8、16、32、64个扇区。

一般文件的存储方式采用链式存储。硬盘上的文件常常要进行创建、删除、增长、缩短等操作，这样的操作越多，盘上的文件就可能被分得越零碎（每段至少是1簇）。但是，由于硬盘上保存着段与段之间的连接信息（即FAT），操作系统在读取文件时，总是能够准确地找到各段的位置并正确读出。

CF卡的访问规则

4、DIR（Directory）

DIR（Directory）是根目录区，紧接着第二FAT表（即备份的FAT表）之后，记录着根目录下每个文件（目录）的起始单元，文件的属性等。定位文件位置时，操作系统根据DIR中的起始单元，结合FAT表就可以知道文件在硬盘中的具体位置和大小了。

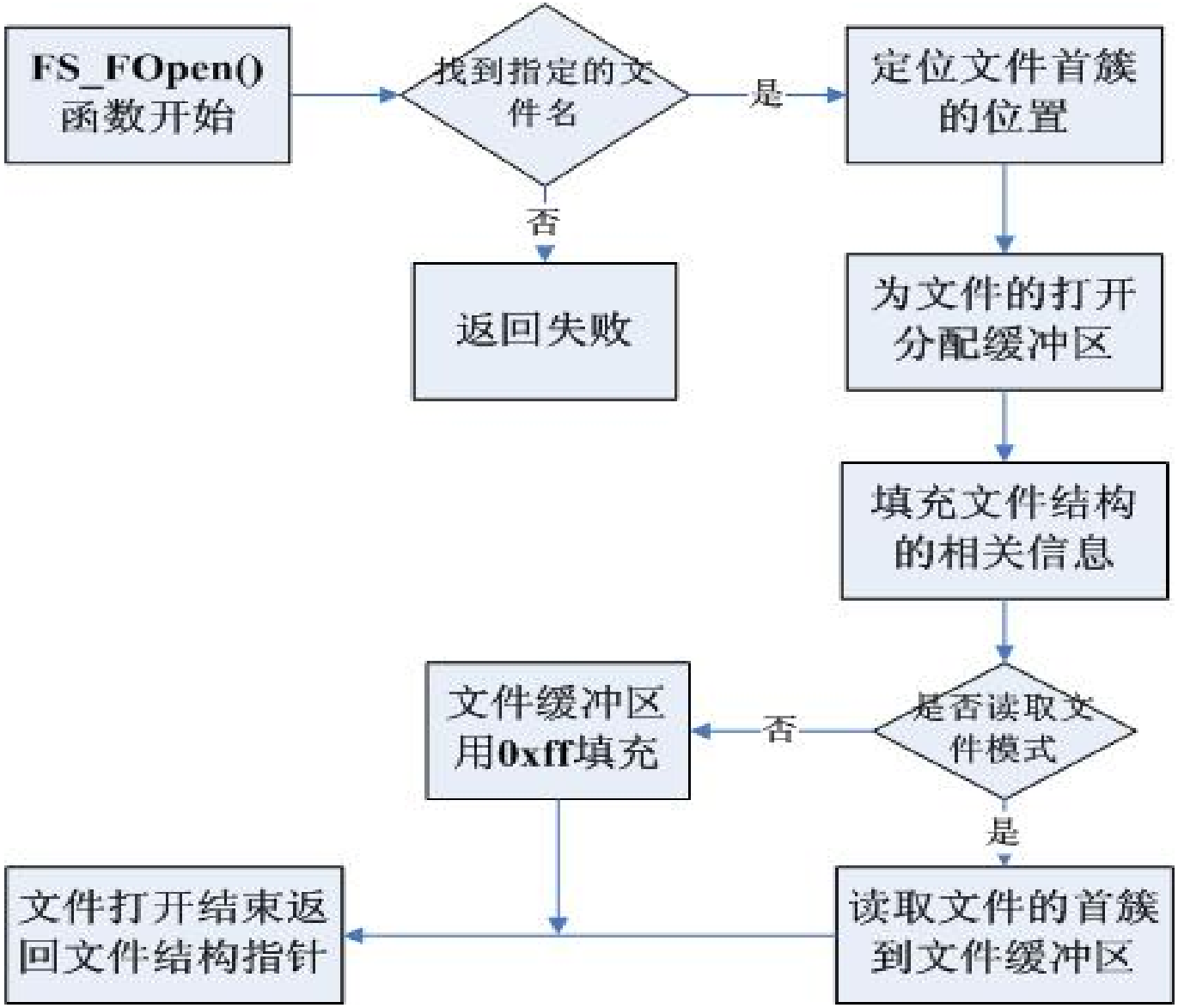
5、数据区

数据区是真正意义上的数据存储的地方，位于DIR区之后，占据硬盘上的大部分数据空间。

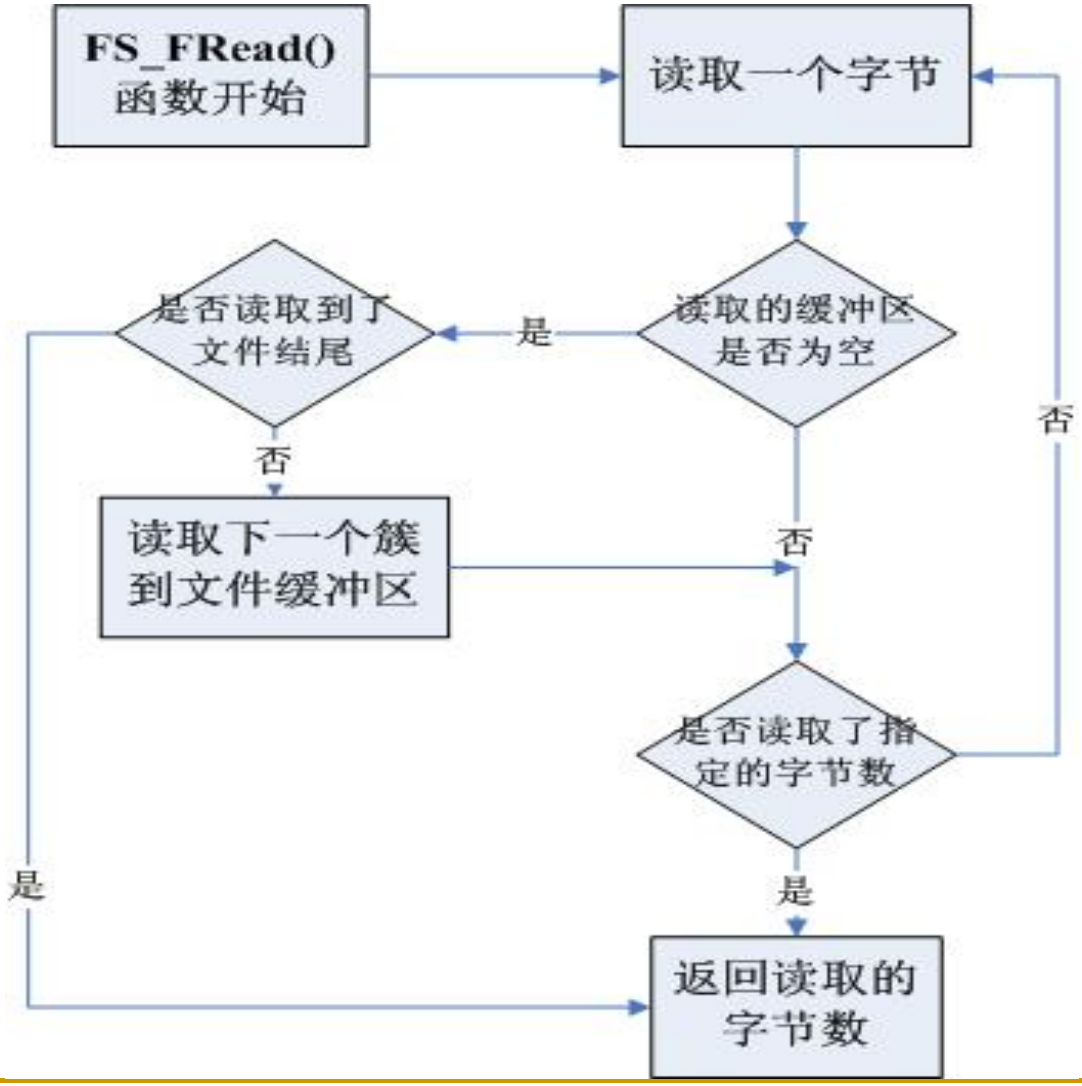
文件的读写过程

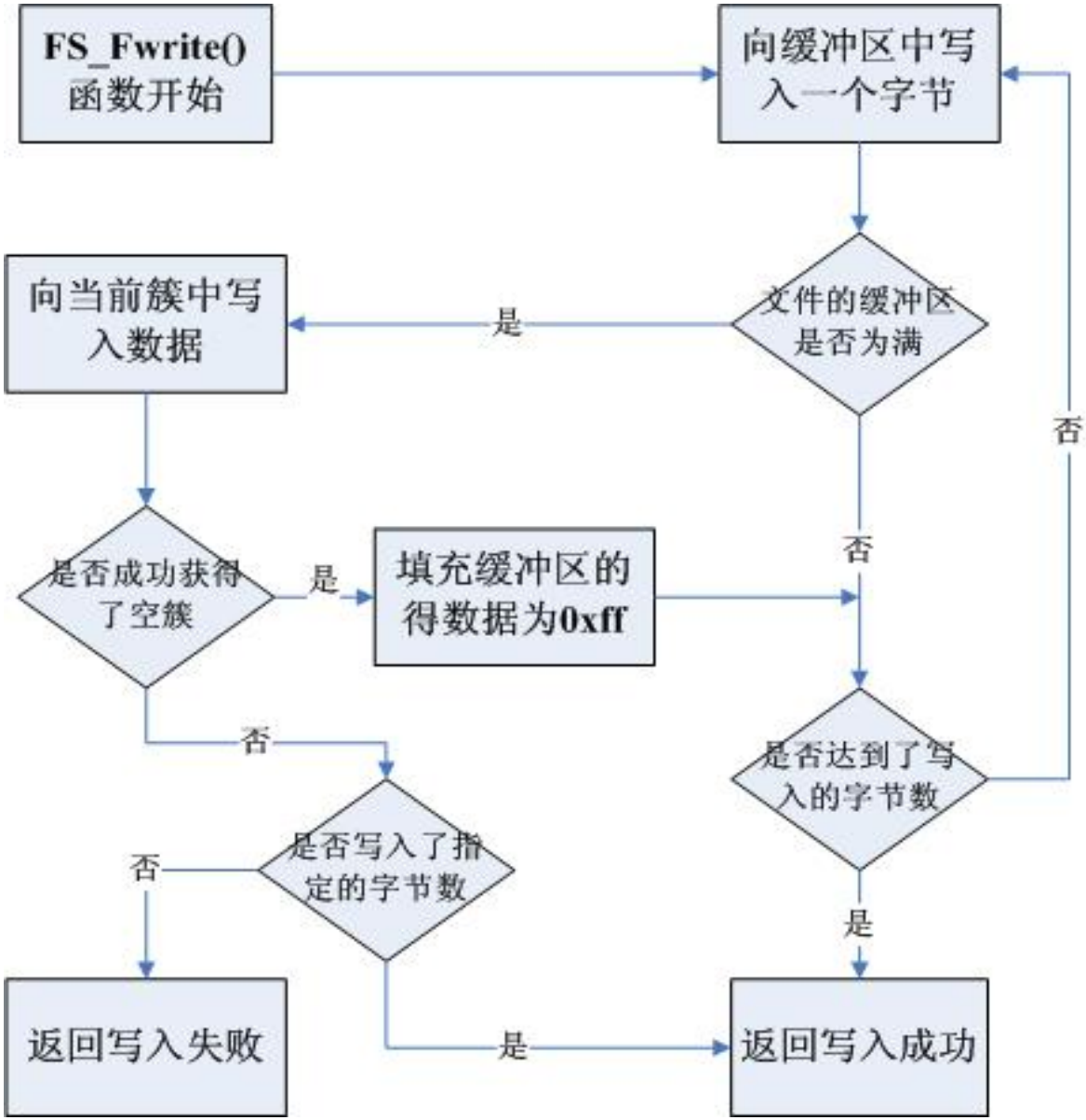
文件的打开

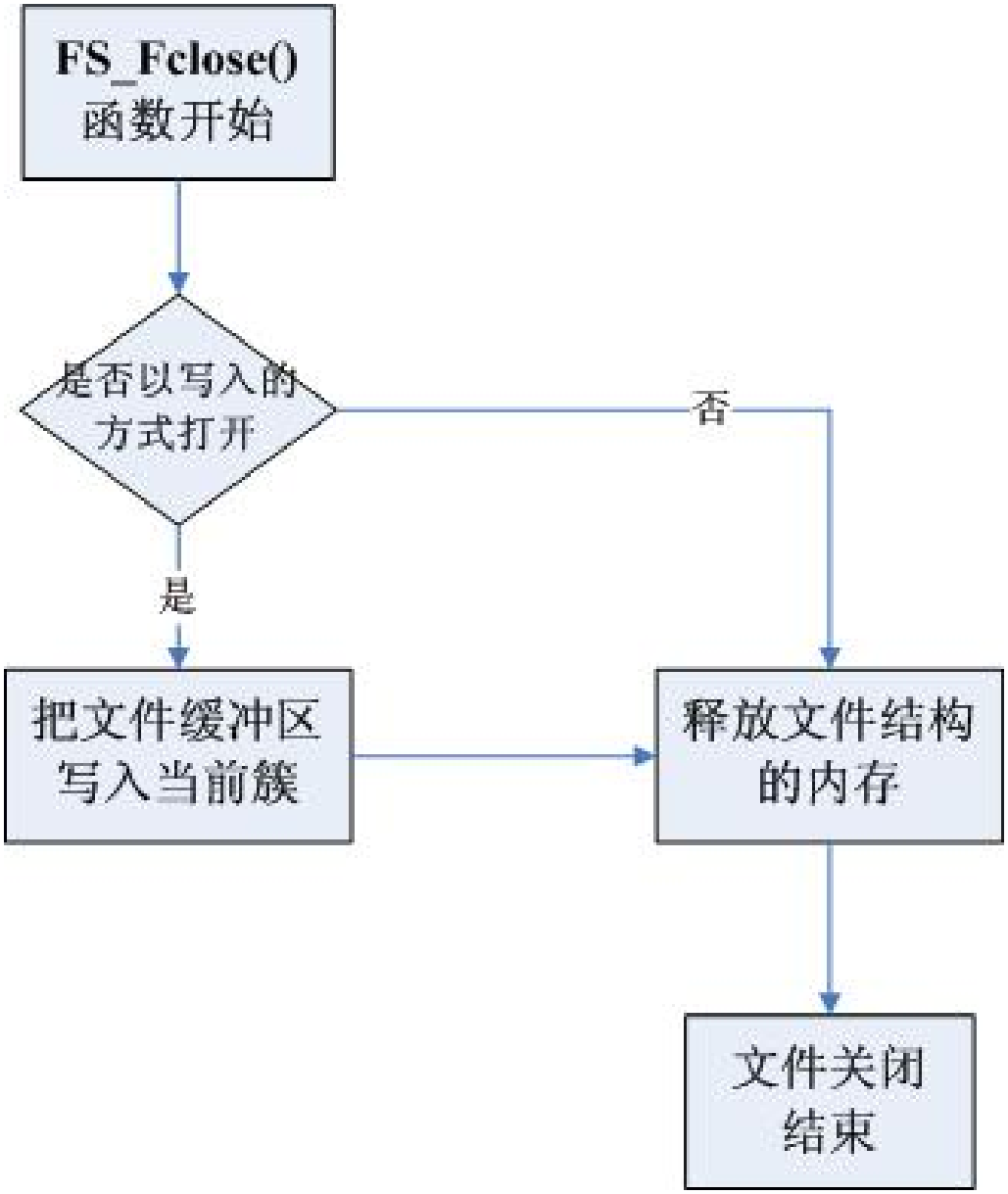
在第一次调用FS_FOpen()函数时，在查找文件名之前会有对存储体的读写。依次是MBR区（1个扇区，得到相关磁盘信息，并计算分区的首地址），DBR区（1个扇区，主要是读取BPB表（本分区参数表）BPB参数块记录着本分区的起始扇区、结束扇区、文件存储格式、硬盘介质描述符、根目录大小、FAT个数，分配单元的大小等重要参数。），读取FAT表，读取目录表，等到这些信息都读到内存后，就会查找文件名，定位文件首簇。


FS_FOpen()
函数开始
找到指定的文件名
是
否
定位文件首簇的位置
返回失败
为文件的打开分配缓冲区
填充文件结构的相关信息
是否读取文件模式
否
是
文件缓冲区用0xff填充
读取文件的首簇到文件缓冲区
文件打开结束返回文件结构指针

在文件读取之前应该先调用FS_FOpen()函数打开文件，




FS_Fwrite()
函数开始
向缓冲区中写
入一个字节
文件的缓冲区
是否为满
是
否
向当前簇中写
入数据
是否成功获得
了空簇
是
填充缓冲区的
得数据为0xff
否
否
是否达到了写
入的字节数
是
是否写入了指
定的字节数
否
是
返回写入失败
返回写入成功


FS_Fclose()
函数开始
是否以写入的
方式打开
否
是
把文件缓冲区
写入当前簇
释放文件结构
的内存
文件关闭
结束

5、建立文件系统的shell

实现一个与用户交互的界面（命令行），实现文件的创建、删除，目录的创建、删除，进入下一级目录、退回上一级目录，显示当前目录的内容等基本功能，在整个系统集后，这个shell将作为操作系统的一个控制台任务，并添加其他命令。

CF卡驱动的编写

1） 驱动层

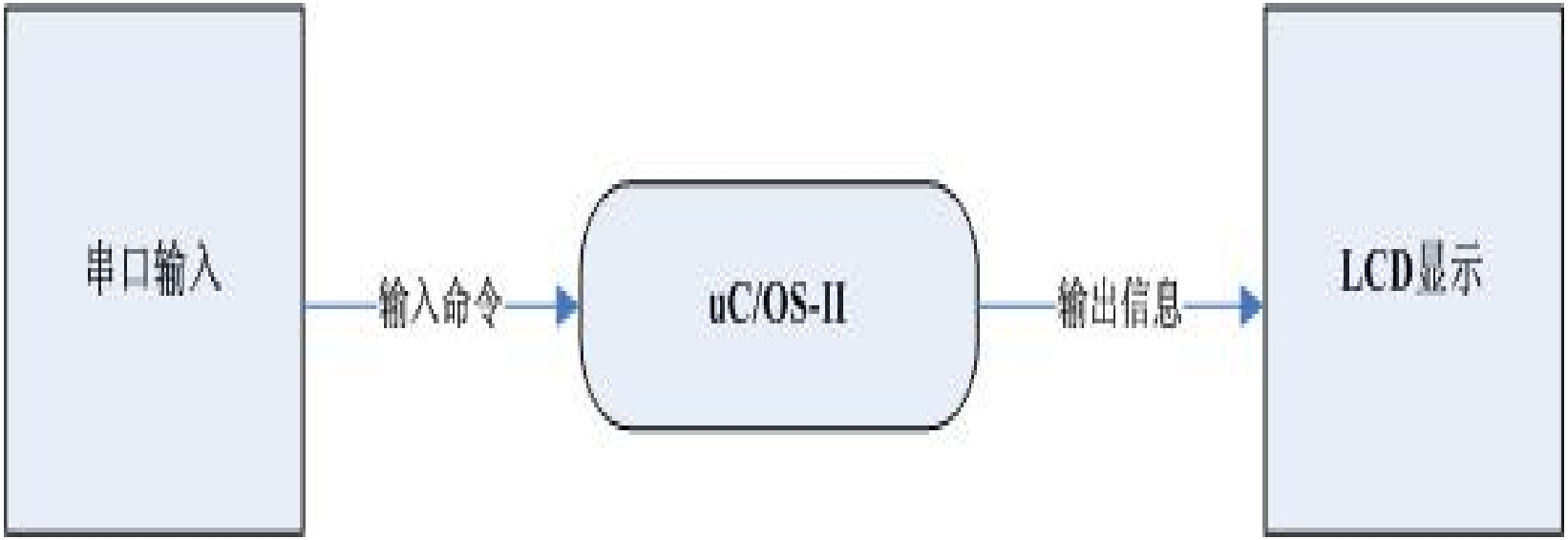
对CF卡的驱动的编写，实现对CF卡的控制及按块（512B）读写，这主要是为文件系统的第五层硬件层添加相应的代码。在设备驱动层，会有3个函数，一个是CF的获得状态，在其中如果得到的CF卡状态是改变的（通常是在第一次调用这个函数时），将从CF卡的0柱面0磁道第1个扇区，即sector0，读取MBR（主引导区）中的内容，以确定第一个分区的sector地址。其余那两个函数即为读扇区和写扇区，他们把上层得到的扇区号传给下一层用以读写CF卡以调用最底层的FS__IDE_ReadSector对扇区进行读写。

2）硬件层

对于底层硬件操作在得到相应的扇区号和缓冲区后，由于采用DMA传送数据的方式，首先须对DMA做初始化，然后把扇区号和缓冲区首地址送入CF卡的相应的寄存器中即可。这样一个扇区操作完毕后函数就返回，CF卡中的数据就读到了缓冲区中（或者缓冲区的的内容写入CF卡中）。

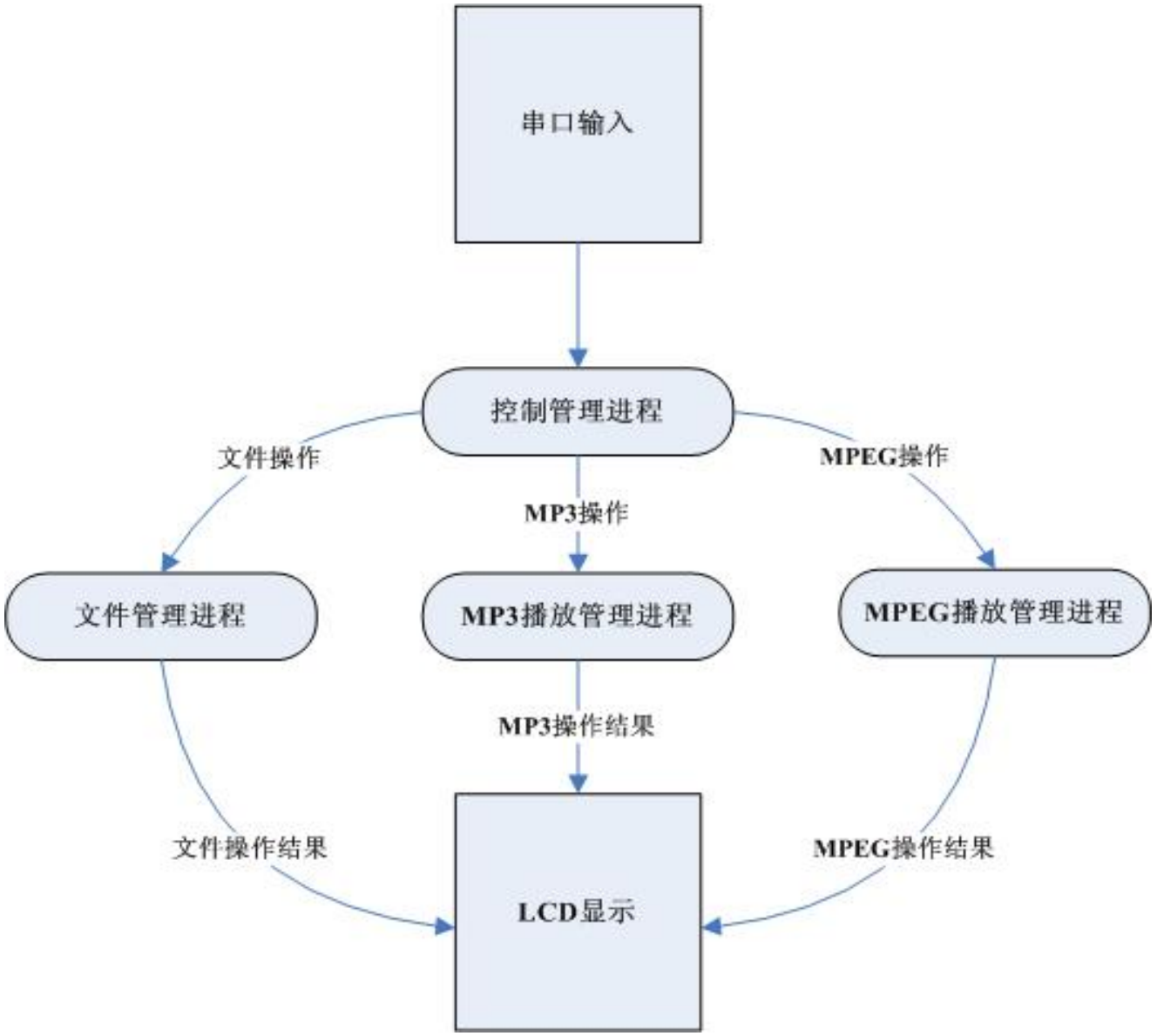
人机交互的设计

采用命令行式的交互设计，μC/OS-II与外部设备交互图



μC/OS-II内部操作设计

整个系统包括控制管理，文件管理，MP3播放，MPEG播放四个模块。控制管理进程管理其他三个进程，其中在最终实现时把文件管理的进程并入控制管理进程以减少任务切换带来的负荷。当操作系统初始化完成后，由控制进程建立其他进程，从输入终端等待接收命令，并由信号量或者邮箱等机制来触发其他进程的执行。


串口输入
控制管理进程
文件操作
MP3操作
MPEG操作
文件管理进程
MP3播放管理进程
MPEG播放管理进程
MP3操作结果
文件操作结果
MPEG操作结果
LCD显示

系统功能设计说明

系统功能可从三个方面说明：

1、对于文件的操作：

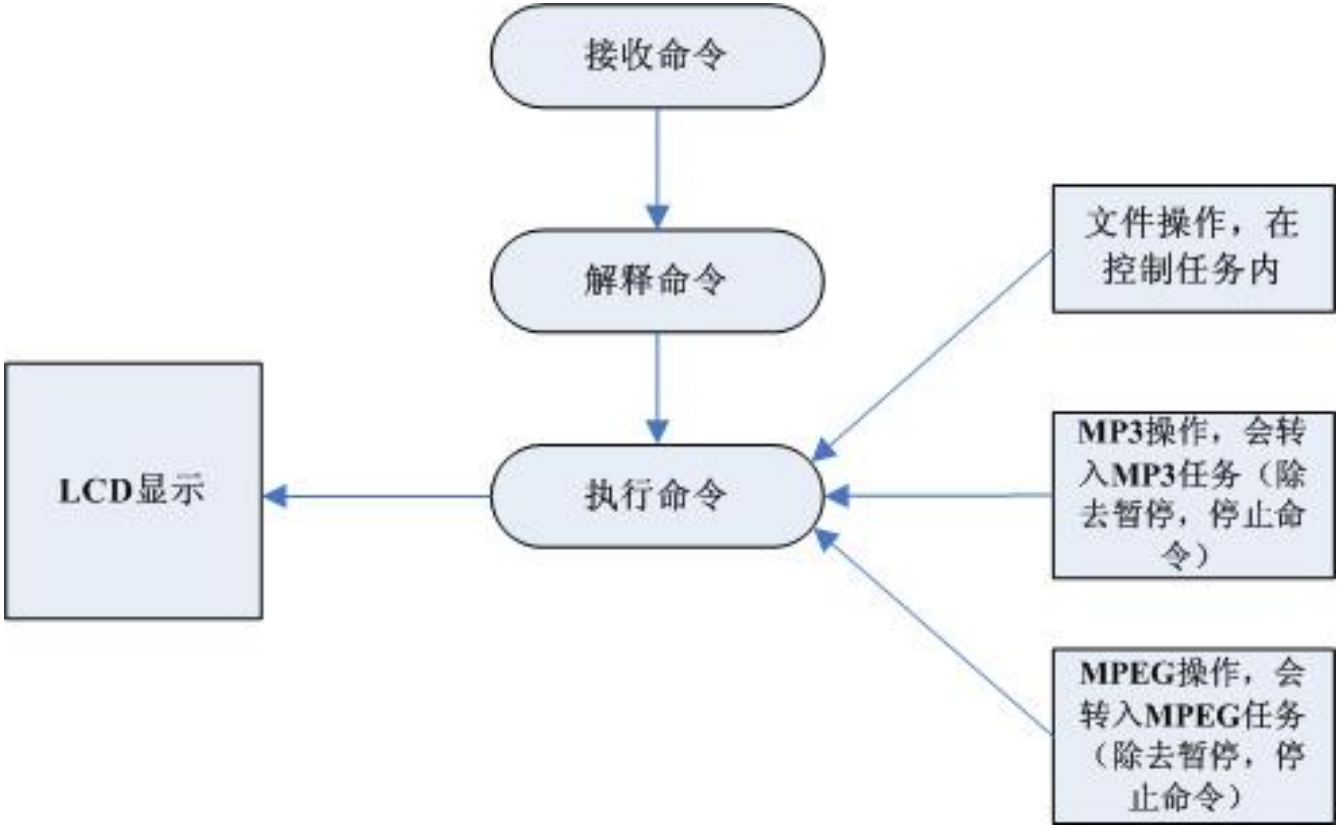
对于文件的操作设计了打开（open），删除（del），现实当前目录的内容（dir），建立目录（mkdir），删除目录（rmdir），进入下级目录或者返回上一级目录（cd）。

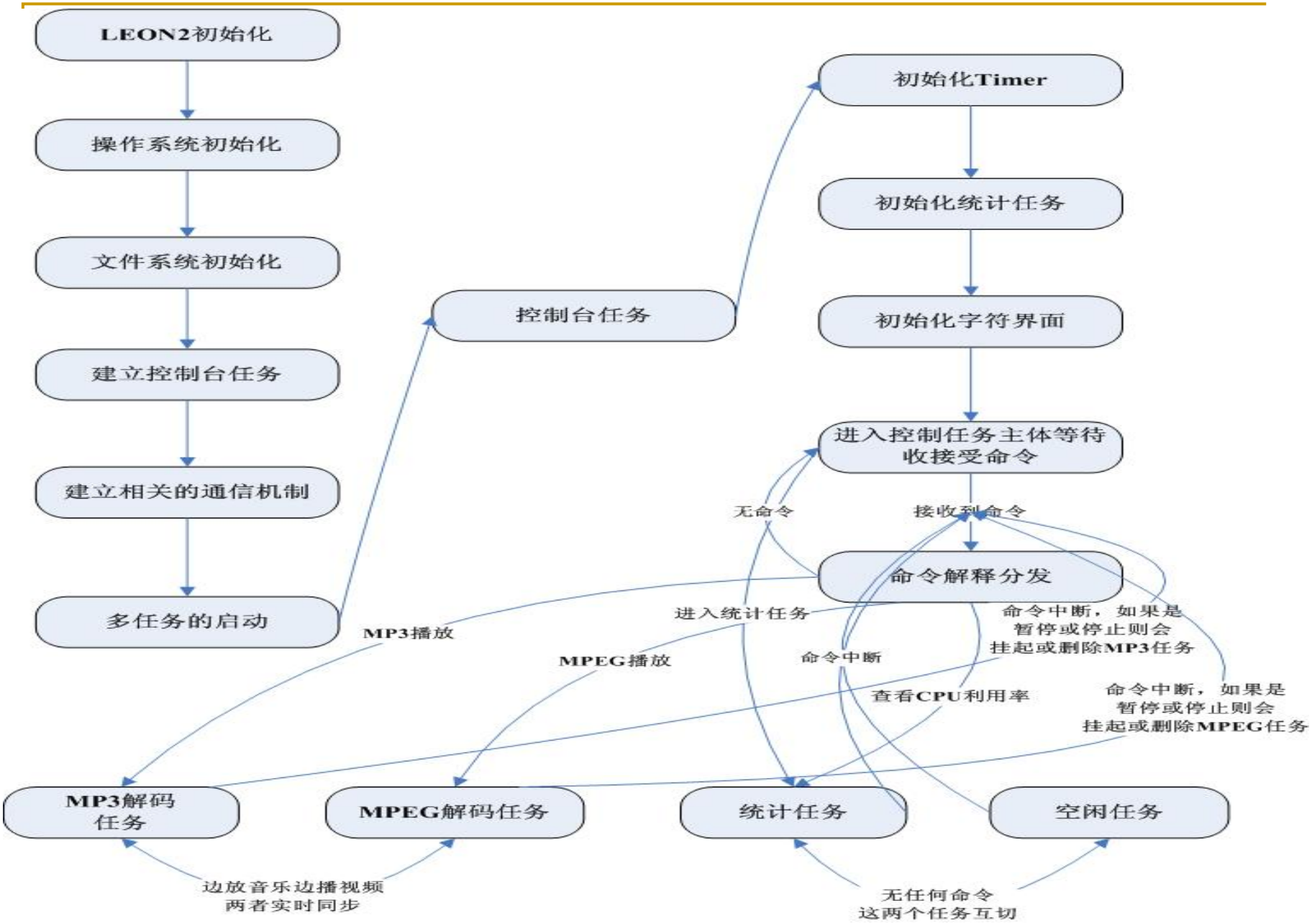
2、查看当前CPU状态：perf

3、媒体播放：播放指定媒体文件（play *.mp3或play *.mpg），暂停播放（pause），继续播放（continue），停止播放（stop）

控制模块设计

控制模块会始终从输入终端等待接收命令，对接收到的命令进行解释分发，执行相应的命令操作，并在输出接口（LCD）进行显示




LEON2初始化
操作系统初始化
文件系统初始化
建立控制台任务
建立相关的通信机制
多任务的启动
控制台任务
初始化Timer
初始化统计任务
初始化字符界面
进入控制任务主体等待
收接受命令
无命令
接收到命令
命令解释分发
MP3播放
进入统计任务
命令中断，如果是
暂停或停止则会
挂起或删除MP3任务
MPEG播放
命令中断
查看CPU利用率
命令中断，如果是
暂停或停止则会
挂起或删除MPEG任务
MP3解码
任务
MPEG解码任务
统计任务
空闲任务
边放音乐边播视频
两者实时同步
无任何命令
这两个任务互切

多任务启动后就会跳转至控制台任务，初始化Timer，初始界面以及统计任务后，就会等待接受命令。而这时程序则会跳转至统计任务，由于统计任务中有时钟等待，则又会跳转至空闲任务，这两个程序就会交替执行，直到发生中断得到命令就会返回控制台任务。这时会有两种情况，一、得到的是文件操作或者查看CPU使用率的命令，则程序仍在控制台任务中执行相应的操作，然后再等待下一条命令，这时又会转至上述状态；二是如果得是播放命令，则会跳转至MP3 或MPEG的解码程序，MP3 或MPEG的解码任务就会一直同步运行，直到新的命令到来，如果新的命令是和解码任务相关的，这个任务就有可能被挂起、删除、或者删除后重新建立（播放其他文件），如果不是和解码任务相关的，那么执行完相应的操作后就会再次进入被中断掉的解码任务继续解码。

主函数

```
void main()
{
  初始化SPARC开发板相关部件；/* 亦可在boot中初始，
除去Timer（多任务启动后才可启动） */
  初始化操作系统;
  初始化文件系统
  调用OSTaskCreatExt()建立第一个任务ControlTask,优先
级定为8;
  建立传递消息所用的通信机制（邮箱）；
  调用OSStart()启动多任务;
}
```

控制台任务

```
void ControlTask(void *pdata)
{ 初始化Timer；
  初始化统计任务；
调用DisplayInit()函数初始化显示界面；
  while(1)
  { 等待接受命令（采用邮箱）；
    命令解释分发程序；
    /* if dir    do   displayDirectory();
      if del    do   removeFile();
      if open   do   openFile();
      if cd     do  changeDirectory();
      if mkdir  do  makeDirectory();
      if rmdir  do  removeDirectory();
      if perf   do  do_displayCPUUseage();
      if help   do  help();
      if play   do  playmedia(); /* 辨认play后的参数以确定播放那种格式的文件,调
用相应的OSMBoxPost()向解码任务发送消息命令,即要播放的文件名指针*/
      if stop   do  stopplay();  /* 向解码任务发送消息命令用以停止播放*/
      if pause  do  pauseplay(); /*向解码任务发送消息命令用以暂停播放*/
      if continue  do  continue();/*向解码任务发送消息命令用以继续播放*/
}}
```

```
void MP3Decoder()
{
 while(1)
{
   等待接收控制台发送的消息，调用msgcmd=OSMBoxPend();
   mp3_decoder(msgcmd);
}
}
```

```
void MPEGDecoder()
{
 while(1)
{
等待接收控制台发送的消息，调用msgcmd=OSMBoxPend();
   MPEG_decoder(msgcmd);
}
```