

# 面向二十一世纪的嵌入式系统设计技术

## 第五讲：

ucOS/II实时操作系统

**RTOS(一)：ucOS/II**

任课教员：徐欣 博士

主讲教员：习勇 博士



2002年1月

国防科大电子科学与工程学院  
嵌入式系统开放研究小组



# What is uC/OS?

---

u: Micro C:control

uC/OS : 适合于小的、控制器的操作系统

- 小巧
- 公开源代码，详细的注解
- 可剥夺实时内核
- 可移植性强
- 多任务
- 确定性



## The Story of uC/OS

---

- 美国人Jean Labrosse 1992年编写的
- 商业软件的昂贵
- 应用面覆盖了诸多领域，如照相机、医疗器械、音响设备、发动机控制、高速公路电话系统、自动提款机等
- 1998年uC/OS-II，目前的版本uC/OS-II V2.51
- [www.uCOS-II.com](http://www.uCOS-II.com)





# 概要

---

- 内核结构-任务以及调度机制
- 任务间通信
- uC/OS的移植
- 在PC机上运行uC/OS



# 任务task

---

- 典型的一个无限循环。

```
void mytask(void *pdata)
{
    for (;;) {
        do something;
        waiting;
        do something;
    }
}
```

- 支持64个任务，每个任务一个特定的优先级。优先级越高，数字越小
- 系统占用了两个任务，空闲任务和统计任务。

## 任务的数据结构—任务控制块

- 任务控制块 OS\_tcb，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。
- 所有的任务控制块分为两条链表，空闲链表和使用链表。

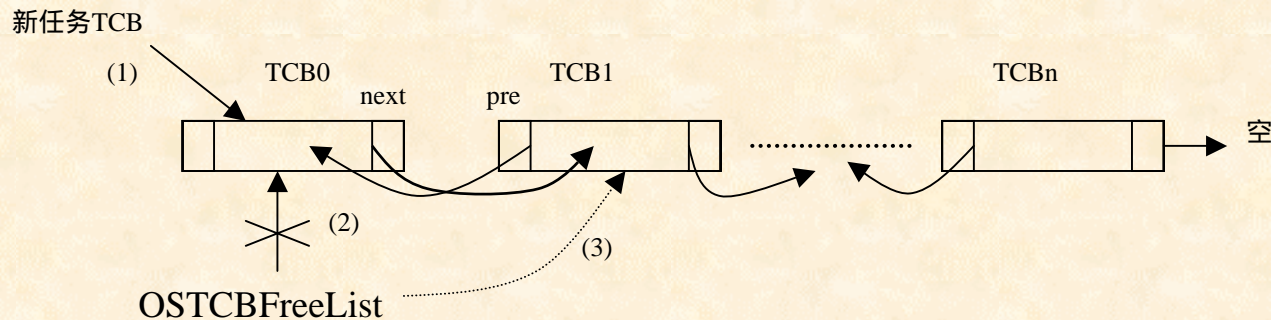


图 4.3 TCB的双向链表结构





## 任务控制块结构

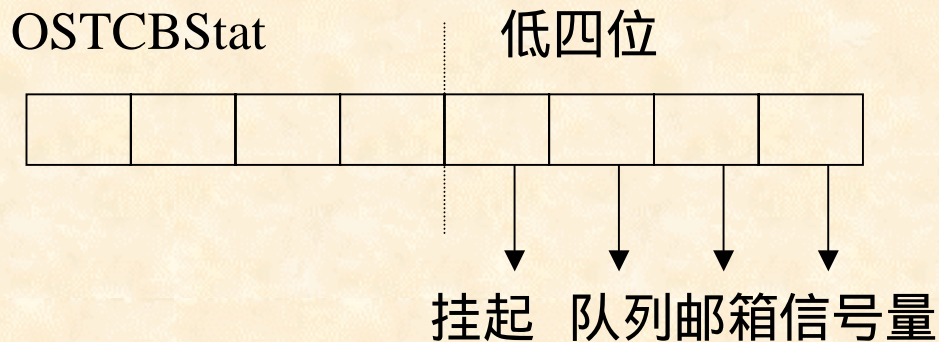
---

- Struct os\_tcb {

```
    OS_STK      *OSTCBStkPtr;  
    struct os_tcb *OSTCBNext;  
    struct os_tcb *OSTCBprev;  
    OS_EVENT    *OSTCBEventPtr;  
    void        *OSTCBMsg;  
    INT16U      OSTCBDly;  
    INT8U       OSTCBStat;  
    INT8U       OSTCBPrio;  
    INT8U       OSTCBX, OSTCBY, OSTCBBitX, OSTCBBitY;  
} OS_TCB
```

## 任务的状态 OSTCBStat

- 运行，就绪，等待，挂起...



- 可以有多个准备就绪的任务，但一个时刻只有一个任务可以运行，OSHighRdy



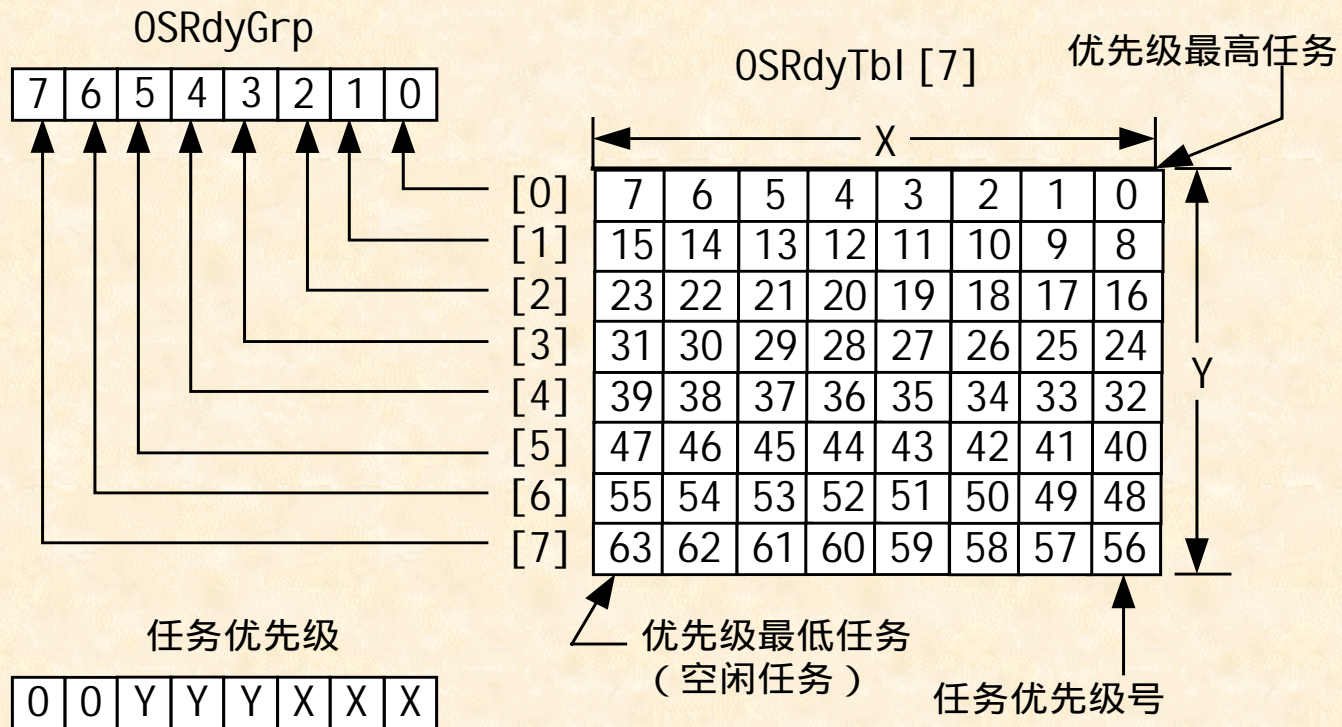


## 任务的调度--OSSched

---

- uC/OS是占先式实时多任务内核，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。
- uC/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。

## 根据优先级找到任务在就绪任务表中的位置



## 根据优先级确定就绪表

- 假设优先级为12的任务进入就绪状态， $12 = 1\ 100b$ ，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 ;

OSRdyTbl[1]   |= 0x10;

- 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 ;

OSRdyTbl[2]   |= 0x20;





## 根据优先级确定就绪表

---

- 从上面的计算我们可以得到：若第 $n$ 位置1，则应该与 $2^n$  相或。uC/OS中，把 $2^n$ 的 $n=0-7$ 的8个值 先计算好存在数组OSMapTbl[7]中,也就是：

OSMapTbl[0] =  $2^0 = 0x1$ ;

OSMapTbl[1] =  $2^1 = 0x2$ ;

.....

OSMapTbl[7] =  $2^7 = 0x80$ ;



## 根据优先级确定就绪表

---

- 利用OSMapTbl，通过任务的识别号-优先级prio来设置任务在就绪组和就绪表数组中相应位置的数学式为：

OSRdyGrp                      |= OSMapTbl[prio >> 3];

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

假设优先级为12，1 100b

OSRdyGrp        |= 0x02 ;

OSRdyTbl[1]     |= 0x10;



## 根据就绪表确定最高优先级

---

两个关键：

- 优先级数分解为高三位和低三位分别确定；
- 高优先级有着小的优先级号；



## 根据就绪表确定最高优先级

- 通过OSRdyGrp值确定高3位，假设为  
 $0x24 = 100\ 100b$ ，--- OSRdyTbl[2] 和  
OSRdyTbl[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，取高优先级第2个

---- 17



## 源代码中使用了查表法

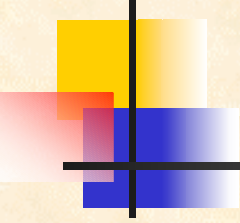
---

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;`

? 为什么频繁的使用查表法

? 请问OSUnMapTbl的来历；



```
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  
};
```





# 任务间通信手段

---

- 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：

程序4.5 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（当事件是信号量时）*/  
    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

---

- 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。





# 中断与时钟节拍

---

- 我们知道：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iret(x86)指令实现中断返回。
- uC/OS中提供了OSIntEnter() 和OSIntExit() 告诉内核进入了中断状态。OSIntNesting
- 时钟节拍是一种特殊的中断，操作系统的心脏。首先32位的整数OSTime加一。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。



# 多任务的启动

---

- 首先创建任务
- 最后调用OSStart开始多任务调度

```
void main( )  
{ OSInit( );  
  .....  
  OSTaskcreat( )  
  .....  
  OSStart();  
}
```



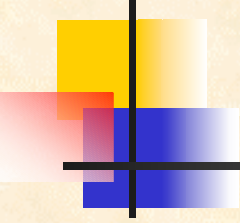
# 任务的格式

---

- 每个任务不能占用全部CPU的资源
- 需要有等待，或延时等系统调用
- 典型的一个无限循环。

```
void mytask(void *pdata)
{
    for (;;) {
        do something;
        waiting;
        do something;
    }
}
```



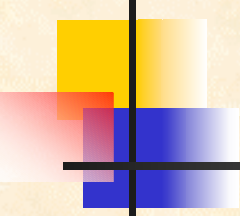


## 揭开神秘的面纱——任务调度全程追踪

- For example1 创建2个任务，每个任务仅仅是进行延时，延时不同的时间片，不同优先级

```
void Task1(void)
{
    while(1)
    {
        blinkled1();
        Task1Data++;
        OSTimeDly(25);
    }
}
```

```
void Task2(void)
{
    while(1)
    {
        blinkled2();
        Task2Data++;
        OSTimeDly(50);
    }
}
```



---

```
void main()
{
    sysinit();
    OSInit ();
    OSTaskCreate ( Task1, (void *)&Task1Data,
                  (void *)&Task1Stk[TASK_STK_SIZE],Task1prio);
    OSTaskCreate (Task2, (void *)&Task2Data,
                  (void *)&Task2Stk[TASK_STK_SIZE],Task2prio);
    ticker_start(OS_TICKS_PER_SEC);
    OSStart();
}
```



```
void OSStart (void)
```

```
{
```

```
    INT8U y, x;
```

```
    if (OSRunning == FALSE) {
```

判断是否没有启动内核

```
        y      = OSUnMapTbl[OSRdyGrp];
```

```
        x      = OSUnMapTbl[OSRdyTbl[y]];
```

```
        OSPrioHighRdy = (INT8U)((y << 3) + x);
```

找到优先级最高的准备就绪任务

```
        OSPrioCur    = OSPrioHighRdy;
```

当前运行任务优先级

```
        OSTCBHighRdy  = OSTCBPrioTbl[OSPrioHighRdy];
```

根据任务优先级找到任务

```
        OSTCBCur      = OSTCBHighRdy;
```

```
        OSStartHighRdy();
```

让优先级最高的任务运行起来

```
    }
```

```
}
```



OSStartHighRdy:

bl OSTaskSwHook 用户自定义函数

把OSRunning设为1

li r0,1

lis r11,OSRunning@ha

stb r0,OSRunning@l(r11)

获取准备运行的任务TCB指针

lis r11,OSTCBHighRdy@ha

lwz r11,OSTCBHighRdy@l(r11)

设置当前运行任务TCB

lis r12,OSTCBCur@ha

stw r11,OSTCBCur@l(r12)

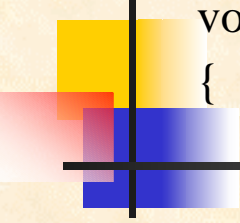
获取新的任务的堆栈指针

lwz r1,0(r11)

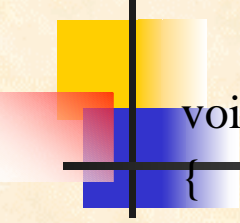
恢复新任务的上下文

lwz r2,XR2(r1)

.....



```
void OSTimeDly (INT16U ticks)
{
    if (ticks > 0) { 确保tick大于0
        OS_ENTER_CRITICAL(); 进入临界段代码
        if ((OSRdyTbl[OSTCBCur->OSTCBBY] &= ~OSTCBCur->OSTCBBitX) == 0)
        {
            /* Delay current task */
            OSRdyGrp &= ~OSTCBCur->OSTCBBitY; 设置任务为非就绪状态
        }
        OSTCBCur->OSTCBDly = ticks; 在TCB中装载延时数
        OS_EXIT_CRITICAL(); 退出临界段代码
        OSSched(); 调度下一个任务开始运行
    }
}
```



```
void OSSched (void)
```

```
{
```

```
    INT8U y;
```

```
    OS_ENTER_CRITICAL();
```

```
    if ((OSLockNesting | OSIntNesting) == 0) { 调度锁，或者处于中断状态禁止调度
```

```
        y = OSUnMapTbl[OSRdyGrp];
```

```
        OSPrioHighRdy = (INT8U)((y << 3) + OSUnMapTbl[OSRdyTbl[y]]);
```

```
        获取准备就绪组里最高优先级的任务
```

```
        if (OSPrioHighRdy != OSPrioCur) {
```

```
            OSTCBHighRdy = OSTCBPrioTbl[OSPrioHighRdy];
```

```
            设置运行任务为最高优先级任务
```

```
            OSCtxSwCtr++;
```

```
            OS_TASK_SW(); 执行上下文切换
```

```
        } }
```

```
    OS_EXIT_CRITICAL();
```

```
}
```





## OS\_TASK\_SW 任务的上下文切换

---

- 通过sc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



# 什么也不做的空闲任务

---

只是为了消耗CPU的时间片

```
void OSTaskIdle ( )  
{  
  for (;;) {  
    OS_ENTER_CRITICAL();  
    OSIdleCtr++;  
    OS_EXIT_CRITICAL();  
  }  
}
```

## 嵌入式操作系统—uC/OS

```
void OSTimeTick (void)
```

```
{
```

```
    OS_TCB *ptcb;
```

```
    ptcb = OSTCBLList;    ---OSTCB链表指针
```

```
    while (ptcb->OSTCBPrio != OS_IDLE_PRIO) {  看是不是空闲任务，空闲任务是最后的任务
```

```
        if (ptcb->OSTCBDly != 0) {    是否延时
```

```
            if (--ptcb->OSTCBDly == 0) {    延时减一，看是否延时结束
```

```
                if (!(ptcb->OSTCBStat & OS_STAT_SUSPEND)) {
```

```
                    OSRdyGrp    |= ptcb->OSTCBBitY;    是的话将其列入准备就绪表
```

```
                    OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
```

```
                } else {ptcb->OSTCBDly = 1; }
```

```
            } }
```

```
    ptcb = ptcb->OSTCBNext;  指针指向下一个TCB结构
```

```
}
```

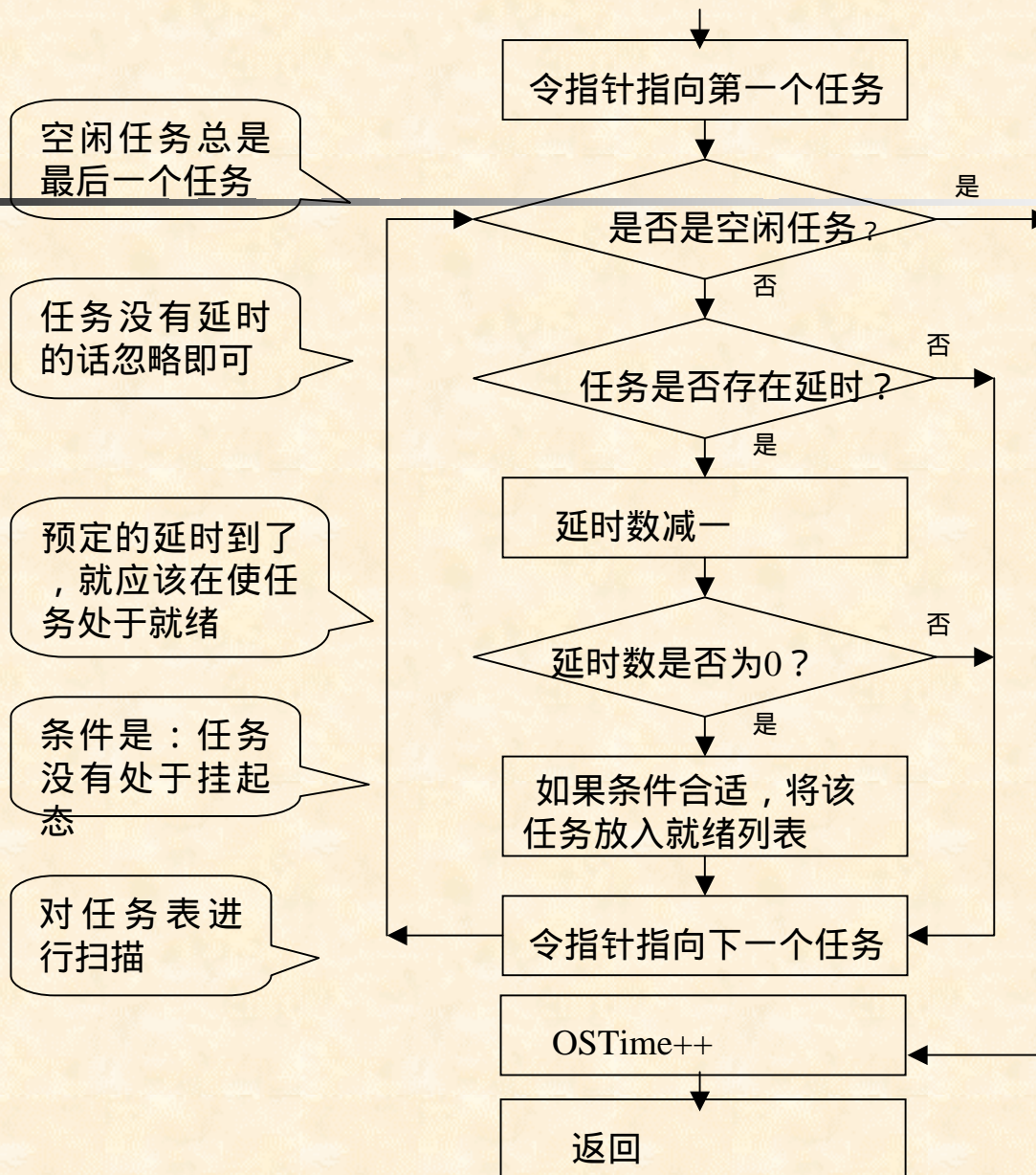
```
    OSTime++;    变量加一，记录系统启动以来的时钟滴答数
```

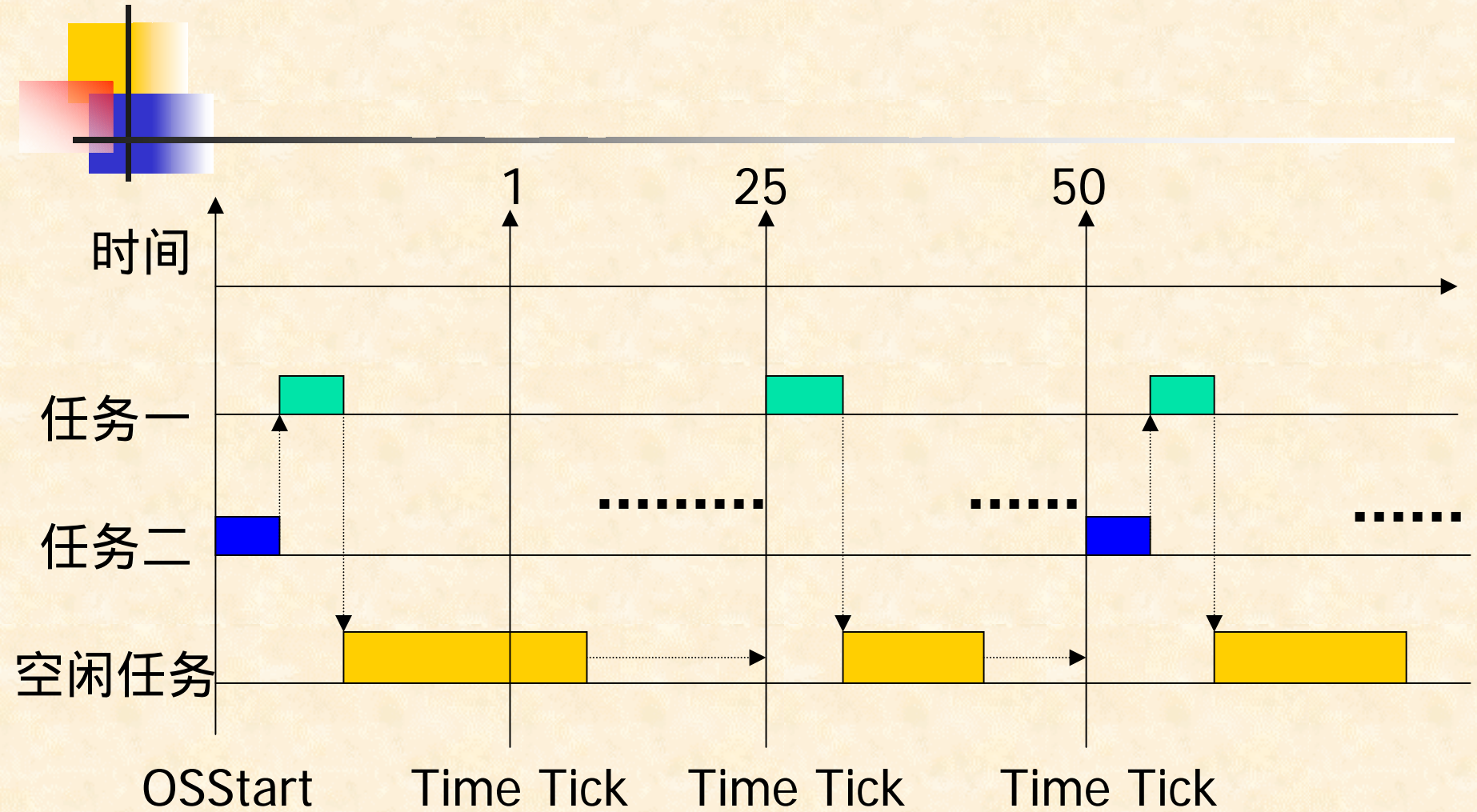
```
}
```



# 嵌入式操作系统—uC/OS

OSTimeTick( void )







## 总结

---

- 不存在一个内核任务/实体，内核的管理是通过调用系统函数来实现的。
- 每个任务有自己的堆栈空间。内核对任务的占先式调度不会干扰每个任务的总的运行结果。





## uC/OS的移植

---

- 代码分为三部分：
- 与 CPU 无关的内核代码，包括 `os_core.c`, `os_mbox.c`, `os_mem.c`, `os_q.c`, `os_sem.c`, `os_task.c`, `os_time.c`, `ucos_ii.c`, `ucos_ii.h`;
- 与应用相关的设置头文件，包括：`os_cfg.h`, `include.h`;
- 处理器相关的代码，包括：`os_cpu.h`, `os_cpu_a.asm`, `os_cpu.c.c`。



## 移植要点

---

- 定义函数OS\_ENTER\_CRITICAL和OS\_EXIT\_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





## 学习uC/OS的步骤

---

- 学习与研究uC/OS的起点：在PC上执行uC/OS，环境：TC编译环境，新建一个工程
- 应用程序设计：消费者与生产者经典问题。
- 源代码阅读
- 移植：DSP，单片机