# 第2章 uC/OS-II中的任务

## 1、任务的基本概念

* + 从应用程序设计的角度，uC/OS-II的任务就是一个线程。
  + uC/OS-II任务的组成：

任务代码：任务的执行部分

任务堆栈：保存任务工作环境

任务控制块：保存任务的属性

* + uC/OS-II的任务有两种：

用户任务：由应用程序设计者编写的任务，为解决应用问题而编写的

系统任务：系统提供的任务，为应用程序提供某种服务

* + 设优先级别有N个，则OS\_LOWEST\_PRIO为N-1，任务最多有N个，其中

其中：

OS\_LOWEST\_PRIO的优先级（第一低的优先级）自动赋给空闲任务

OS\_LOWEST\_PRIO－1的优先级（第二低的优先级）自动赋给统计任务（存在的话）

## 2、任务堆栈

* 作用：
  1. 任务的重要组成部分
  2. 每个任务都配有自己的堆栈
  3. 满足任务切换和响应中断时保护CPU寄存器中的内容及存储任务私有数据的需要
  4. 是存储器中按数据“后进先出（LIFO）”的原则组织的连续存储空间
* 如何创建任务堆栈？
  1. 在文件**OS\_CPU.H**中的数据类型OS\_STK

typedef unsigned int OS\_STK; //该类型长度为16位

* 1. 在**应用程序**中定义OS\_STK类型的一个数组

# define TASK\_STK\_SIZE 512 //定义堆栈长度

OS\_STK TaskStk[TASK\_STK\_SIZE] //定义数组

* 1. 当需要创建任务堆栈的时候

先调用函数**OSTaskCreate()**来创建一个任务时，把数组的指针传递给函数OSTaskCreate()中的堆栈栈顶参数**ptos**，就可以把该数组与任务关联起来而成为该任务的任务堆栈

* 如何初始化任务堆栈（任务堆栈初始化函数OS\_STK \*OSTaskStkInit(……)内部是怎样的）？

void \*OSTaskStkInit (void (\*task)(void \*pd), void \*ppdata, void \*ptos, INT16U opt) reentrant

{

OS\_STK \*stk;

//以下两句都是确保参数正确的定势

ppdata = ppdata;

opt = opt; //opt没被用到，保留此语句防止告警产生

//给不用C++多年的计院选手解释一下怎么这么多++= ： 相当于把一块连续的内存看成一个结构体（OS\_STK）（忘记什么是结构体的Java选手就看做一个类）（什么？你问我什么是类？（请输入文本））的整体，++=使得指针下移并赋值，也就可以看作给结构体内部的成员变量赋值。为甚不直接定义结构体，因为这样子用运算符更快（个人理解，可能有误）

stk = (OS\_STK \*)ptos; //用户堆栈最低有效地址

\*stk++ = 15; //用户堆栈长度

\*stk++ = (INT16U)task & 0xFF; //任务地址低8位

\*stk++ = (INT16U)task >> 8; //任务地址高8位

\*stk++ = 0x00; //PSW

\*stk++ = 0x0A; //ACC

\*stk++ = 0x0B; //B

\*stk++ = 0x00; //DPL

\*stk++ = 0x00; //DPH

\*stk++ = 0x00; //R0

//R3、R2、R1用于传递任务参数ppdata，其中R3代表存储器类型，R2为高字节偏移，R1为低字节位移。

//通过分析KEIL汇编，了解到任务的void \*ppdata参数恰好是用R3、R2、R1传递，不是通过虚拟堆栈。

// &是位运算符，相当于取和，对低八位字节取和相当于只取低八位字节的值

// >>是数字的8421码右移n位的意思，R2这一行相当于把ppdata的高八位移到低八位存进stk

\*stk++ = (INT16U)ppdata & 0xFF; //R1

\*stk++ = (INT16U)ppdata >> 8; //R2

\*stk++ = 0x01; //R3 因为我用的全是XDATA，所以存储器类型固定为1，见C51.PDF第178页说明。

\*stk++ = 0x04; //R4

\*stk++ = 0x05; //R5

\*stk++ = 0x06; //R6

\*stk++ = 0x07; //R7

//不用保存SP，任务切换时根据用户堆栈长度计算得出。

\*stk++ = (INT16U) (ptos+MaxStkSize) >> 8; //?C\_XBP 仿真堆栈指针高8位

\*stk++ = (INT16U) (ptos+MaxStkSize) & 0xFF; //?C\_XBP 仿真堆栈指针低8位

return ((void \*)ptos);

}

## 3、任务控制块**（OS\_TCB，这个缩写要记住）**

* 作用

用来记录任务的堆栈指针、任务的当前状态、任务的优先级别等一系列与任务管理有关的属性的表

* 结构

typedef struct os\_tcb{

OS\_STK \*OSTCBStkPtr; //指向任务堆栈栈顶的指针

struct os\_tcb \*OSTCBNext; //指向后一个任务控制块的指针

struct os\_tcb \*OSTCBPrev; //指向前一个任务控制块的指针

INT16U OSTCBDly; //任务等待的时限（节拍数）

INT8U OSTCBStat; //任务的当前状态标志

INT8U OSTCBPrio; //任务的优先级别

INT8U OSTCBX; //用于快速访问就绪表的数据

INT8U OSTCBY; //用于快速访问就绪表的数据

INT8U OSTCBBitX; //用于快速访问就绪表的数据

INT8U OSTCBBitY; //用于快速访问就绪表的数据

}OS\_TCB;

* 任务控制块链表

用**两条链表**来管理任务控制块

空任务控制块链表（初始化5个链表之一）：

* + 应用程序调用函数**OSInit()**对uC/OS-II系统进行初始化时建立的
  + 单向链表

任务控制块链表

* + 应用程序调用OSTaskCreate()创建任务时建立的
  + 双向链表
* 如何通过链表管理（初始化）任务控制块

当应用程序调用函数**OSTaskcreate()**创建一个任务时，这个函数会调用系统函数**OSTCBInit()**来为任务控制块进行初始化。

* 首先为被创建任务从空任务控制块链表获取一个空任务控制块
* 然后用任务的属性对任务控制块各个成员进行赋值
* 最后再把这个任务控制块链入到任务控制块链表的头部（注意OSTCBPrioTbl[]的作用）
* 为什么要对TCB进行初始化？
* 获得任务控制块
* 保存任务属性
* OSTCBPrioTbl[]的作用
  + **优先级到 TCB 的映射**：数组下标直接对应任务的优先级（0~63），每个元素存储该优先级任务的 TCB 指针。例如：  
    OSTCBPrioTbl[5] = &Task5\_TCB; // 优先级为5的任务的TCB地址
  + **快速任务调度**：当调度器需要切换到最高优先级就绪任务时，通过 OSPrioHighRdy变量（存储当前最高优先级就绪任务的优先级），直接从OSTCBPrioTbl中获取对应 TCB 指针：  
    OSTCBCur = OSTCBPrioTbl[OSPrioHighRdy]; // 直接定位到目标TCB
  + **任务状态管理**：当任务创建、删除或改变优先级时，系统会更新OSTCBPrioTbl中对应优先级的 TCB 指针，确保映射关系的正确性。

## 4、任务就续表及其操作（**OSRdyTbl[8]**）

* 作用：
  + uC/OS-II进行任务调度的依据
  + uC/OS-II在RAM中设立了一个记录表，系统中的每个任务都在这个表中占据一个位置，并用这个位置的状态（1或0）来表示任务是否处于就绪状态
  + 是一个二维数组
* 什么是调度？
  + 通过一个算法在多个任务中确定哪个任务来运行
  + 特别的，在uC/OS-II中的调度的思想：①每时每刻总是让**优先级最高**的就绪任务处于运行状态；②在**系统或用户任务调用系统函数**及**执行中断服务程序结束**时，调用调度器来确定应该运行的任务并运行它
* 任务组变量OSRdyGrp的作用？

位数对应OSRdyTbl[]的下标，即组号；状态表示该组有无任务就绪

* 如何根据优先级查找就绪表具体位置？
  + 只用6位2进制数表示0~63,64个任务
  + 用高3位表示OSRyGrp的具体数据位（组号），即数组元素的下标；
  + 低3位表示数组元素的具体数据位（位置号）！

eg. 已知某一个已经就绪的任务的优先级别prio=49，试判断应该在就绪表的哪一位置上置1。

法一：49的二进制形式为00110001，其低6位为110001，于是可知应该在OSRdyTbl[6]的D1位上置1，同时要把变量OSRdyGrp的D6位置1。

（这边提供一个不用转二进制的思路，会更快）（师傅，你的招不够快，也不够狠.jpg）

法二：49%8=6，49/8=1；取余数得出的是组号下标，取除法的出来的是在该组的位置。即在OSRdyTbl[6][1]。记得用**8**去做运算

* 对任务就绪表的CRUD
  + 1. 登记：在程序中，用下面的代码把优先级别为prio的任务置为就绪状态

OSRdyGrp|=OSMapTbl[prio>>3];//prio左移三位，取高八位做组号

OSRdyTbl[prio>>3]|=OSMapTbl[prio&0x07];//prio对7取和，取低八位做组内下标

* + 1. 注销：使一个优先级别为prio的任务脱离就绪状态

If((OSRdyTbl[prio>>3]&=~OSMapTbl[prio&0x07])==0)

// 对**该组号下的所有任务**取和，等价于对原操作的组内下标搞针对，组内的其他任务下标不改变，它强制附0

OSRdyGrp&=~OSMapTbl[prio>>3];

//对**组号下标**取和，等价于对原操作的所有组号搞针对，组内的其他组号下标不改变，它强制附0

* + 1. 查询：从任务就绪表中获得优先级别最高的就绪任务

Y=OSUnMapTbl[OSRdyGrp]; //获得优先级别高3位

X=OSUnMapTbl[OSRdyTbl[y]]; //获得优先级别低3位

Prio=(y<<3)+x; //获得就绪任务的优先级别

或

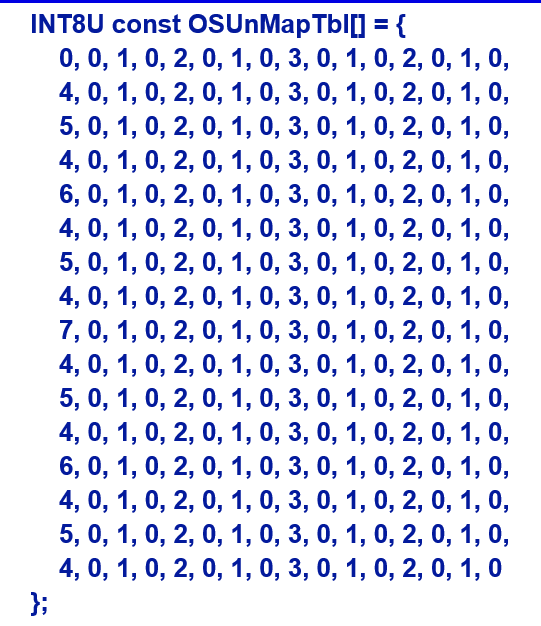
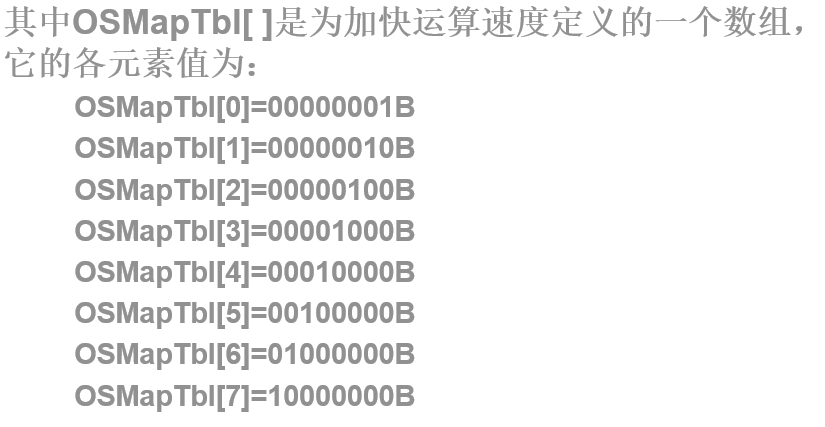
Y=OSUnMapTbl[OSRdyGrp];

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

OSUnMapTbl[ ]是预先定义的数组

ps：什么是OSMapTbl[]、什么是OSMapTbl[]？

其实就是为了加块运算而提前写好的两个数组。



* OSUnMapTbl[ ]的赋值原理
  + 首先分析y=OSUnMapTbl[OSRdyGrp];
  + 当OSRdyGrp的bit0位为1时，表明OSRdyTbl[0]中有任务就绪，则y应该为0，故数组OSUnMapTbl中以奇数（OSRdyGrp的值）为下标的数组元素都应该为0；
  + 当OSRdyGrp的bit1和bit0位分别为10时，表明OSRdyTbl[1]中有任务就绪，则y应该为1，故数组OSUnMapTbl中下标能被2整除且不能被4整除的数组元素值都应该为1；
  + 依此类推，可以得到数组OSUnMapTbl所有的数组元素值……

## 5、任务调度、任务切换

* 什么是任务的调度？（上面也有一版，凉拌都可以答）

在多任务系统中，令CPU中止当前正在运行的任务转而去运行另外一个任务的工作叫做任务切换，而按某种规则进行任务切换的工作叫任务的调度

* 任务调度的过程
  1. **在任务就绪表中查找具有最高优先级别的就绪任务**
  2. **实现任务切换**
     1. **获得待运行任务的TCB指针**
     2. **进行断点数据的切换**
* 有哪些调度器（调度函数）？

调度器有两种：

1. 任务级调度器：OSSched()函数
2. 中断级调度器：OSIntExit()函数

* 任务切换宏OS\_TASK\_SW()

实际工作主要由OSCtxSw（实际上是一个中断服务程序）来完成

#define OS\_TASK\_SW() asm INT uCOS

* OSCtxSw()依次完成的工作

1. 把被中止任务的**断点指针PC**保存到任务堆栈中
2. 把CPU**通用寄存器**的内容保存到被中止任务的堆栈中
3. 把被中止任务的**任务堆栈指针**当前值保存到该任务的任务控制块的OSTCBStkPtr中
4. 获得待运行任务的任务控制块
5. 使CPU通过任务控制块获得待运行任务的**任务堆栈指针**
6. 把待运行任务堆栈中通用寄存器的内容恢复到CPU的**通用寄存器**中
7. 使CPU获得待运行任务的**断点指针PC**

不难发现是一个以④为中轴的轴对称流程

## 6、任务调度器函数OSSched()

* 源码

void OSSched(void)

{

# if OS\_CRITICAL\_METHOD == 3

OS\_CPU\_SR cpu\_sr;

# endif

INT8U y;

OS\_ENTER\_CRITICAL();

if ((OSLockNesting|OSIntNesting)==0)// 任务调度锁计数器和中断嵌套计数器都为0时表示允许调度

{

y=OSUnMapTbl[OSRdyGrp];

OSPrioHighRdy //得到最高优先级任务

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

if (OSPrioHighRdy!=OSPrioCur)

{

OSTCBHighRdy //得到任务控制块指针

=OSTCBPrioTbl[OSPrioHighRdy];

OSCtxSwCtr++; //统计任务切换次数的计数器加1

OS\_TASK\_SW(); //任务切换宏

}

}

OS\_EXIT\_CRITICAL();

}

## 7、任务创建

* 有哪些创建任务的函数？

有两个用来创建任务的函数：

1. OSTaskCreate()
2. OSTaskCreateExt()

一般情况下，我们写程序会在调用函数OSStart()之前先创建一个任务，并赋予它最高的优先级别，从而使它成为起始任务；然后在这个任务中，再创建其它各任务。

eg.

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*主函数\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void main(void)

{

......

OSInit(); //对uC/OS-II进行初始化

......

OSTaskCreate(TaskStart,......); //创建起始任务TaskStart

OSSTart(); //开始多任务调度

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*起始任务\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void TaskStart(void \*pdata)

{

...... //在这个位置安装并启动uC/OS-II的时钟

OSStatInit(); //初始化统计任务

.... //在这个位置创建其它任务

for(;;)

{

起始任务TaskStart的代码

}

}

## 8、任务的挂起(Suspend)和恢复(Resume)

* OSTaskSuspend()的原型: INT8U OSTaskSuspend(INT8U prio)



必须注意的是：挂起任务自身时，函数参数为：OS\_PRIO\_SELF（0xFF）

* 恢复任务函数OSTaskResume()的原型：INT8U OSTaskResume(INT8U prio);

## 9、请求删除任务函数（OSTaskDel）

* 主要流程

把被删除任务的任务控制块从任务控制块链表中删除，并归还给空任务控制块链表，然后在任务就绪表中把该任务的就绪状态置为0，于是该任务就不能再被调度了。

ps:可以删自己本任务（自裁），也可以删别的任务。被动删除的任务，往往存在动态分配资源丢失!!

* 一般删除任务的方法

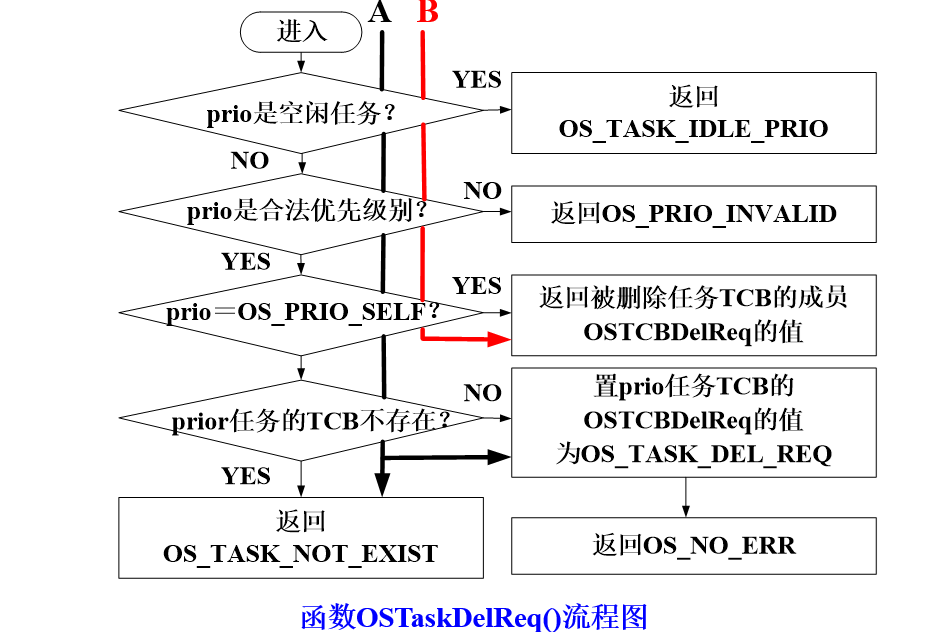
1. 提出删除任务请求的任务A**只负责提出**删除任务请求，而删除工作则由被删除任务B**自己来完成**
2. 联络信号：OSTCBDelReq（B的任务控制块成员）
3. A调用请求删除函数：OSTaskDelReq()
4. 函数原型：

INT8U OSTaskDelReq(

INT8U prio //待删除任务的优先级别

);

1. 提出删除任务请求的调用参数：prio
2. 下图为样例流程图



任务请求删除代码

While(OSTaskDelReq(44)!=OS\_TASK\_NOT\_EXIST)

{

OSTimeDly(1)

}

被删除方典型代码

If(OSTaskDelReq(OS\_PRIO\_SELF)==OS\_TASK\_DEL\_REQ)

{

//释放资源和动态内存代码

OSTaskDel(OS\_PRIO\_SELF);

}

Else

{

//其它应用代码

}

## 10、任务初始化

* uC/OS-II的初始化（OSInit()）

1. 对所有的全局变量和数据结构进行初始化
2. 创建空闲任务OSTaskIdle，并赋之以最低优先级和永远的就绪状态
3. 如果常数**OS\_TASK\_STAT\_EN**=1，则以优先级别为**OS\_LOWEST\_PRIO-1**来创建**统计任务**
4. **OSInit()**对数据结构初始化

* 主要是创建包括空任务控制快链表在内的5个空数据缓冲区
* 创建数组OSTCBPrioTbl[OS\_LOWEST\_PRIO+1]
* uC/OS-II的启动（OSStart()）
  + 源码（看懂就行）

void OSStart (void)

{

INT8U y;

INT8U x;

if (OSRunning == FALSE) {

y = OSUnMapTbl[OSRdyGrp]; /\* Find highest priority's task priority number \*/

x = OSUnMapTbl[OSRdyTbl[y]];

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

OSPrioCur = OSPrioHighRdy;

OSTCBHighRdy = OSTCBPrioTbl[OSPrioHighRdy]; /\* Point to highest priority task ready to run \*/

OSTCBCur = OSTCBHighRdy;

OSStartHighRdy(); /\* Execute target specific code to start task \*/

}

}

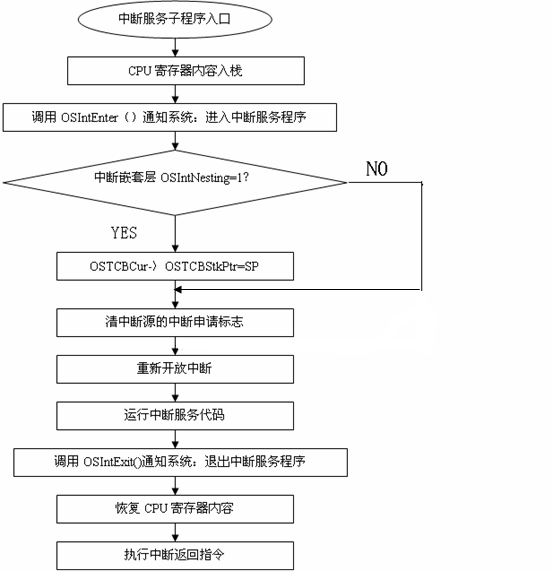
# 第3章 uC/OS-II的中断和时钟

## 1、中断响应过程

* μC/OS-II响应中断的过程
  1. **接收到中断请求后，这时如果CPU处于中断允许状态（即中断是开放的），系统就会中止正在运行的当前任务，而按照中断向量的指向转而去运行中断服务子程序**
  2. **接收到中断请求后，这时如果CPU处于中断允许状态（即中断是开放的），系统就会中止正在运行的当前任务，而按照中断向量的指向转而去运行中断服务子程序**
  3. **有以下图表**

**图片1**

* 中断服务程序（**OSTickISR**）的流程

****

## 2、临界段

* 定义

在应用程序中经常有些代码段必须不受任何干扰地连续运行，这样的代码段叫做临界段

* 宏OS\_ENTER\_CRITICAL( ) 和 OS\_EXIT\_CRITICAL( ) 的三种实现方法

1. 直接用处理器的开、关中断指令来实现；使用时置**OS\_CRITICAL\_METHOD =1**

#define OS\_ENTER\_CRITICAL() \

asm(“DI”) //关中断

#define OS\_EXIT\_CRITICAL() \

asm(“EI”) //开中断

1. 进入临界区时，先保存CPU的中断允许标志；退出时，恢复中断允许标志；使用时置**OS\_CRITICAL\_METHOD == 2**

#define OS\_ENTER\_CRITICAL() \

asm(“PUSH PSW”) \ //通过保存PSW保存中断允许标志

asm(“DI”) //关中断

#define OS\_EXIT\_CRITICAL() \

asm(“POP PSW”) //恢复中断允许标志

1. 进入临界区时，先保存CPU的中断允许标志；退出时，恢复中断允许标志；使用时置**OS\_CRITICAL\_METHOD == 3**

#define OS\_ENTER\_CRITICAL() \

cpu\_sr =get\_processor\_psw() ; \ //获得PSW并保存在全局变量sr中

disable\_interrupts(); //关中断

#define OS\_EXIT\_CRITICAL() \

set\_processor\_psw(cpu\_sr); //用sr恢复程序状态字

## 3、时钟中断

* 时钟和时钟节拍的定义

1. 任何系统都需要一个周期性信号源处理与时间相关的事件，这个信号源叫时钟
2. uC/OS-Ⅱ用**硬件定时器产生周期为毫秒级的周期性中断**来**实现时钟**，最小时钟单位叫时钟节拍

* OSTimeTick（）的流程/在时钟节拍服务程序中做了什么工作？

1. 给计数器OSTime加1
2. 遍历所有TCB，将它们的OSTCBDly减1
3. 若OSTCBDly为0，且任务未被挂起，将其置为就绪态并相应的修改就绪表

## 4、延时函数

* 作用

为防止高优先级任务独占CPU，所有任务必须在合适位置调用延时函数OSTimeDly()或OSTimeDlyHMSM()，让当前任务暂停一段时间并进行一次调度，让出CPU使用权

ps：使用例子

OSTimeDlyHMSM(1，2，3，4) //延时1小时2分钟3秒4毫秒

* 取消任务的延时（OSTimeDlyResume()）

INT8U OSTimeDlyResume（INT8U prio）;

//取消优先级为prio的延时状态，直接进入就绪队列

# 第4章 任务的同步与通信

## 1、任务同步

* 任务有哪些制约关系？

1. 有两种制约关系：
2. 直接制约关系：源于任务间的合作
3. 间接制约关系：源于对资源的共享/竞争

* 多任务协作时操作系统需要解决哪些问题？

1. 相关任务在执行上要有先后次序
2. 各任务间应该具有一种互斥关系

## 2、事件

* 消息和事件的作用
  + 任务间制约性的合作运行机制叫做任务间的同步，系统中的同步是依靠任务间互发消息来保证的！
  + 在uC/OS-II中，使用信号量、邮箱（消息邮箱）和消息队列这些被称为事件的中间环节来实现任务之间的通信
* 事件邮箱的作用
  1. 多任务操作系统中，需要在任务间传递消息以实现通信
  2. 在内存创建一个消息缓冲区用来存放该消息，任务间传递消息时只需要传递消息缓冲区的指针
  3. 用来传递消息缓冲区指针的数据结构叫做消息邮箱

1. 消息队列的作用
2. 类似“二维消息邮箱”
3. 定义一个指针数组，每个元素都存放一个消息缓冲区指针，就可以通过传递这个指针数组的方法传递多个消息了，这种可传递多个消息的数据结构叫做消息队列

## 3、事件控制块

1. 事件控制块的结构
2. uC/OS-II使用使用事件控制块ECB的数据结构来描述诸如信号量、邮箱（消息邮箱）和消息队列这类事件。
3. 定义在uC/OS-II.H中的事件控制块数据结构如下（了解就行）:

typedef struct{

INT8U OSEventType; //事件的类型

INT16U OSEventCnt; //信号量计数器

void \* OSEventPtr; //消息或消息队列的指针

INT8U OSEventGrp; //等待事件的任务组变量

INT8U OSEventTbl[OS\_EVENT\_TBL\_SIZE]; //任务等待表

}OS\_EVENT;

1. 功能完善的事件为了对等待该事件的任务具有一定管理功能需要做到哪两个方面？uC/OS-II是怎么实现的？

|  |  |
| --- | --- |
| 方面 | uC/OS-II的解决方案 |
| 对所有等待事件的任务进行**记录和排序** | 采用类似于任务就绪表的**任务等待表**来完成对等待事件的任务的记录与排序 |
| 允许任务有一定的**等待时限** | 等待时限记录在**TCB成员OSTCBDly**中；当有任务的时限已到时，将其从**任务等待表**中删除，并设置**任务就续表**，使其进入就绪状态 |

1. 操作事件控制块的函数
2. 事件控制块初始化函数OS\_EventWaitListInit()

被OSXXXCreate（）调用，**该函数将OSEventGrp及任务等待表中每位清0**

1. 使任务进入等待状态的函数OS\_EventTaskWait()

被OSXXXPend（）调用，**将一个任务置于等待状态**

1. 使正在等待的任务进入就绪态的函数OS\_EventTaskRdy()

被OSXXXPost（）调用，**该函数将调用它的等待任务在OSEventTbl[]中的相应位清0，将任务在OSRdyTbl[]中相应位置1,并引发一次任务调度，使等待任务进入就绪态**

1. 使一个等待超时的任务进入就绪态的函数OS\_EventTO()

被OSXXXPend（）调用，场景是**若一个任务已经超过了等待的时间，却仍因为未获取事件等原因而未具备可运行的条件，却又要使它进入就绪状态**

【XXX代表某类事件 见下表】

|  |  |
| --- | --- |
| XXX | 含义 |
| Sem | 对信号量进行操作的函数 |
| Mutex | 对互斥信号量进行操作的函数 |
| Mbox | 对消息邮箱进行操作的函数 |
| Q | 对消息队列进行操作的函数 |

1. 空事件控制块链表有什么用
2. uC/OS-II初始化时，函数OSInit()创建OS\_MAX\_EVENTS个空事件控制块并借用成员OSEventPtr将它们链接成一个单向链表
3. 当新创建一个事件时，从该空事件控制块链表取下一个空ECB并对其初始化
4. 当删除一个事件时，将事件的ECB归还给空事件控制块链表

## 4、事件的操作（创建、请求、发送）

以下都已信号量事件为例展示源码，理解为主。

1. 创建

OS\_EVENT \*OSSemCreate(INT16U cnt)

{

#if OS\_CRITICAL\_METHOD==3

OS\_CPU\_SR cpu\_sr;

#endif

OS\_EVENT \*pevent;

if(OSIntNesting>0)

{

return((OS\_EVENT \*)0);

}

OS\_ENTER\_CRITICAL();

pevent=OSEventFreeLIst;

if(OSEventFreeLIst!=(OS\_EVENT \*)0)

{ OSEventFreeLIst=(OS\_EVENT \*)OSEventFreeLIst->OSEventPtr; }

OS\_EXIT\_CRITICAL();

if(pevent!=(OS\_EVENT \*)0)

{

pevent->OSEventType=OS\_ENVENT\_TYPE\_SEM; //设置为信号量

pevent->OSEventCnt =cnt; //设置计数器初值

pevent->OSEventPtr =(void \*)0; //置空指针

OS\_EventWaitListInit(pevent);

}

return(pevent);

}

1. 请求
2. 调用OSSemPend()请求信号量。函数原型如下:

void OSSemPend (OS\_EVENT \*pevent, INT16U timeout, INT8U \*err)

pevent 被请求信号量的指针

timeout 等待时限，为0时表示等待时间无限长

err 返回值，成功时返回OS\_NO\_ERR，失败时根 据具体情况返回相应的值

1. 任务访问一个共享资源时

if (OSEventCnt>0) { OSEventCnt- -; 使任务继续运行 ；}

if (OSEventCnt=0) { 将任务列入OSEventTbl[];

将timeout保存在OSTCBDly中，使任务处于等待状态；

}

1. 另一个信号量请求函数OSSemAccept()允许任务在信号量无效时不进入等待状态而继续运行
2. 发送
3. 任务调用OSSemPost()释放信号量
4. 如果有任务等待该信号量，运行最高优先级等待任务
5. 如果没有，把信号量计数器加1
6. 成功时返回OS\_NO\_ERR,否则根据具体错误返回OS\_ERR\_EVENT\_TYPE、OS\_SEM\_OVF
7. OSSemPost()原型：

INT8U OSSemPost(

OS\_EVENT \* pevent //信号量的指针

);

* 删除

1. 调用OSSemDel()来删除信号量。原型如下：

OS\_EVENT \* OSSemDel(

OS\_EVENT \* pevent, //信号量的指针

INT8U opt, //删除条件选项

INT8U \* err //错误信息

);

1. 参数opt有两个值可选：

* 选OS\_DEL\_NO\_PEND,则当等待任务表无等待任务时才删除
* 选OS\_DEL\_ALLWAYS,则不管有无等待任务立即删除
* 查询

调用OSSemQuery()查询信号量状态，原型如下:

INT8U OSSemQuery(

OS\_EVENT \* pevent, //信号量指针

OS\_SEM\_DATA \* pdata //存储信号量状态的结构

)

## 5、优先级反转

* 定义

在可剥夺型内核中，当某任务以独占方式共享资源时，会出现低优先级任务先于高优先级任务而被运行的现象，这叫做优先级反转

* 原因

一个低优先级的任务在获得了信号量使用共享资源时，被具有较高优先级的任务打断而不能释放信号量，从而使正在等待这个信号量的更高优先级的任务因得不到信号量而被迫处于等待状态。在这个等待期间，就让优先级别低于它而高于占据信号量的任务的任务先运行。

* 影响

优先级反转极大的恶化了高优先级任务的运行环境，是实时系统所无法容忍的

## 6、相关例题（例9）

待完善

# 第5章 信号量集

## 1、信号量集的组成

uC/OS-II的信号量集由两部分组成：

1. 标志组：存放了信号量集中的所有信号
2. 等待任务链表：每个节点都对应一个正在等待信号量集的等待任务

uC/OS-II初始化时，系统创建OS\_MAX\_FLAGS个标志组，并将其连接成一个空标志组链表；信号量集用一个双向链表(和TCB组织方式一样)来组织等待任务，每个等待任务都是一个节点

## 2、信号量集的操作（创建、请求和发送）

* 任务调用OSFlagCreate()来创建一个信号量集

OS\_FLAG\_GRP \* OSFlagCreate(

OS\_FLAGS flags; //信号量的初始值

INT8U \* err //错误信息

);

创建一个信号量集分两步：

1. 定义一个全局的OS\_FLAG\_GRP类型的指针
2. 在要创建信号量集的地方调用OSFlagCreate()

* 任务调用OSFlagPend()请求一个信号量集

OS\_FLAGS OSFlagPend(

OS\_FLAG\_GRP \* pgrp, //所请求的信号量集的指针

OS\_FLAGS flags, //滤波器

INT8U wait\_type, //逻辑运算类型

INT16U timeout, //等待时限

INT8U \*err //错误信息

);

* 任务调用OSFlagPost()向信号量集发信号

OS\_FLAGS OSFlagPend(

OS\_FLAG\_GRP \* pgrp, //所请求的信号量集的指针

OS\_FLAGS flags, //滤波器

INT8U wait\_type, //逻辑运算类型

INT16U timeout, //等待时限

INT8U \*err //错误信息

);

opt=OS\_FLAG\_SET为置1操作； opt=OS\_FLAG\_CLR为置0操作

## 3、相关例题（例1、例3）

待完善

# 第6章 内存的动态分配

## 1、操作系统对内存的二级管理

uC/OS-II对内存进行两级管理，即把一个大片连续的内存空间分成若干个分区，每个分区又分成了若干大小相等的内存块来管理。而操作系统以**分区为单位**来管理动态内存，而任务以**内存块为单位**来获得和释放动态内存。

uC/OS-II要求同一个分区的内存块的字节数必须相同，且每个分区与该分区内存块的数据结构必须相同。所以可用一个二维数组来在内存中划分一个内存分区和内存块 ，只有把内存控制块与分区关联起来之后，系统才能对该分区进行相应的管理和控制，它才是一个真正的动态内存区。例如

INT16U IntMemBuf[10][30];

## 2、内存分区的组织—单向链表

uC/OS-II初始化时，调用OS\_MemInit()定义并初始化一个空内存控制块链表，其中共有空内存控制块OS\_MAX\_MEM\_PART个（第5个全局链表）

每当应用程序创建一个**内存分区**时，从空内存控制块表头取下**一个空内存控制块**

每当应用程序释放一个**内存分区**时，把该分区对应的**内存控制块**归还给空内存控制块链表

## 3、动态内存的管理（创建、请求、释放）及相关例题（例1、例2）

调用函数OSMemCreate()来建立一个内存分区

调用函数OSMemGet()向某分区获得一个内存块

调用函数OSMemPut()释放一个内存块

# 第7章 在51单片机上移植uCOS-II

## 1、可重入函数

定义：多任务操作系统中，系统提供的函数应该能允许同时被多个任务所调用，而不会通过函数中变量的耦合引起任务之间的互相干扰。这样的函数叫做可重入函数。

①可重入函数应该在函数中**只使用局部变量**，因为函数的局部变量存储在任务的堆栈中。

②如果函数一定要**使用全局变量**，则需要对使用的全局变量做**必要的保护**。

## 2、系统堆栈和任务堆栈映像

* 系统堆栈与任务堆栈映象的关系：

把应用程序中各个任务堆栈的内容存放在片外RAM中的任务堆栈映象中，而只在片内RAM中设置一个公用的系统堆栈：这是因为把应用程序中各个任务堆栈的内容存放在片外RAM中的任务堆栈映象中，而只在片内RAM中设置一个公用的系统堆栈

# 第8章 在ARM上移植uCOS-II

## 1、工作模式的选择

1. ARM处理器核具有用户、系统、管理、中止、未定义、中断、和快中断7种模式。
2. 管理、中止、未定义、中断、和快中断为“异常模式”，不太适合UCOS－II使用
3. 所以uCOS－II主要使用系统（特权）、用户（普通）模式
4. 其中，用户模式是缺省模式，而系统模式是可选模式
5. 两个专门用于工作模式切换的UCOS－II函数：

ChangeToSYSMode（）

ChangeToUSRMode（）

## 2、与处理器相关的三个移植文件

1. 文件OS\_CPU.H的修改
2. 数据类型：将常用的数据类型改写成不依赖于编译的数据类型，**必须移植！**
3. 利用软中断（SWI）实现底层接口：
4. 堆栈增长方式:ADS只支持向下增长的满递减堆栈，因此只能写成如下形式：

#define OS\_STK\_GROWTH 1

1. 文件OS\_CPU\_C.C的编写
2. 任务堆栈初始化函数
3. 临界段
4. 处理器模式转换函数
5. 设置任务的初始指令集函数
6. 软件中断服务程序的C语言部分
7. OSStartHighRdy()
8. UCOS－II的HOOK钩子函数在移植时一般可保持全为空
9. 文件OS\_CPU\_A.S的编写
10. 软件中断服务程序的汇编语言部分
11. OS\_TASK\_SW()和OSIntCtxSW()
12. \_OSStartHighRdy()的汇编语言代码