

# 自主可控嵌入式系统

## 第一讲 概述

### 一、嵌入式系统的定义:

- 标准定义: 以应用为中心、以计算机技术为基础, 软、硬件可裁剪, 适应应用系统对功能、可靠性、成本、体积、功耗等严格要求的专用计算机系统
- 简单描述: 嵌入到对象体中的专用计算机系统

计算技术是嵌入式系统的核心, 软件技术是嵌入式系统的灵魂

### 二、嵌入式系统的三要素

1. 嵌入式: 嵌入到对象体系中, 有对象环境要求
2. 专用性: 软、硬件按对象要求裁剪
3. 计算机: 实现对象的智能化功能

### 三、嵌入式系统和通用计算机的区别

#### 1. 专用性:

- 采用专门的处理器
- 功能算法的专用性
- 系统对用户是透明的, 用户无需了解内部设计细节

#### 2. 小型化(资源有限): 结构紧凑、坚固可靠、计算资源有限

3. 软硬件设计一体化: 软硬件间依赖性强; 应用软件和操作系统一体化设计
4. 需要交叉开发环境: 本身资源首先, 开发由宿主机完成

### 四、嵌入式实时系统的特点

1. 时间约束性: 实时系统任务具有一定的约束时间, 按照实时性要求可以分为软实时系统(超时会导致性能下降)和硬实时系统(超时会导致系统失败)
2. 可预测性: 能够对实时任务的执行时间进行判断, 确定是否能够满足任务的时限要求
3. 可靠性

4. 与外部环境交互性良好

5. 多任务类型：可完成周期任务和非周期任务，还可完成非实时性任务

## 五、嵌入式系统的应用领域

无所不在、无处不在：

- √ 消费电子：手机、平板、摄像机
- √ 信息家电：电视、冰箱、洗衣机
- √ 汽车电子
- √ 机器人：智能玩具、工业机器人、运用机器人
- √ 工业国防

嵌入式技术已经成为后PC时代的主宰

## 六、嵌入式处理器的分类

嵌入式系统硬件的核心部件是嵌入式处理器，按照用途可以分为：

- 嵌入式微控制器 (Micro Controller Unit, MCU): 单片机
- 嵌入式 DSP (Digital Signal Processor, DSP)
- 嵌入式微处理器 (Micro Processor Unit, MPU): ARM
- SOC(System on a Chip, SOC)
- SOPC(System on a Programmable Chip, SOPC)

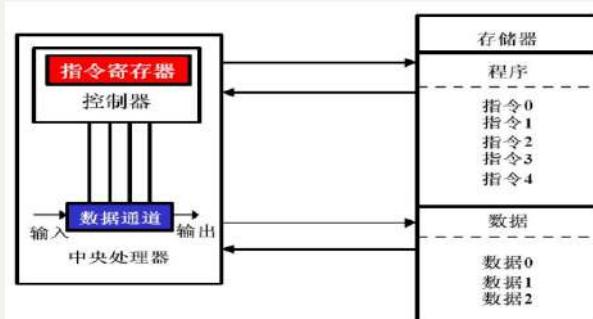
# 第二讲 嵌入式系统基础

## 一、冯诺依曼和哈弗体系结构的区别：

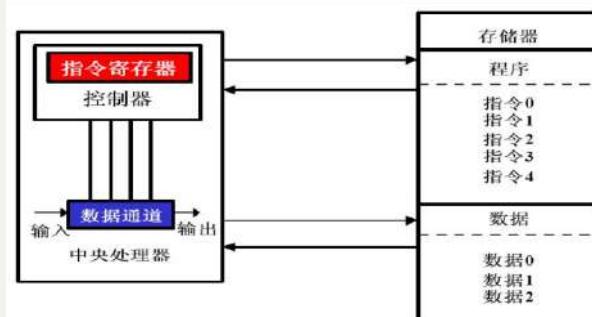
1. 使用两个独立的存储器模块，分别存储指令和数据，每个存储模块都不允许指令和数据并存以便实现并行处理；
2. 使用独立的两条总线，分别作为CPU和每个存储器之间的专用通信路径，这两条总线之间毫无关系
3. 改进的哈弗体系结构采用一条独立的地址总线和数据总线访问两个独立的存储模块

冯诺依曼体系结构：程序存储、程序执行，按照按程序顺序执行；包括输入、存储、运算、控制和输出；程序指令存储器和数据存储器共用同一存储器，统一编址；程序指令和数据的宽度相同

冯诺依曼：



Harvard:

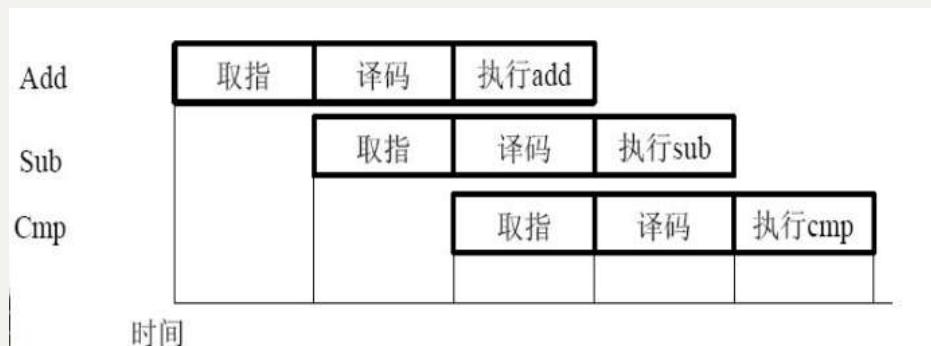


## 二、组合逻辑型和存储逻辑型控制器的区别

- 组合逻辑型：快、复杂
- 存储逻辑型(基于微指令)：慢、规整

## 三、理解流水线、CPU字长对CPU性能的影响

流水线：几个指令可以并行执行，提高了**CPU**的运行效率



**CPU字长**：指的是微处理器一次执行处理的数据宽度；在工作频率相同的情况下，**CPU**字长越长，处理速度越快

#### 四、CPU寄存器和内存的区别

CPU寄存器是位于CPU内部的小型、高速存储单元，用于临时存储和快速访问指令和数据；而内存是位于计算机主板上的存储设备，用于存储程序和数据，供CPU读取和写入。寄存器速度更快但容量有限，用于临时存储；内存速度较慢但容量较大，用于长期存储。

#### 五、RISC(精简指令集)和CISC(复杂指令集)的主要区别(四方面)

指标	RISC	CISC
指令集	一个周期执行一条指令，通过简单指令的组合实现复杂操作；指令长度固定	指令长度不固定，执行需要多个周期
流水线	每周期前进一步	指令的执行需要调用微代码的一个微程序
寄存器	更多通用寄存器	用于特定目的的专用寄存器
Load/Store 结构	独立的Load/Store指令完成数据在寄存器和外部存储器之间的传输	处理器能够直接处理存储器中的数据

CISC缺点：

- 20%与80%的问题：20%的简单指令使用率达到80%，但80%的复杂指令使用率仅20%
- 指令复杂度影响处理器VLSI(集成芯片)实现性能：指令长度不一，高性能VLSI实现难度大
- 软硬件协同设计问题：使用微指令技术增加了硬件复杂度

RISC：采用硬连线的指令译码逻辑，开发更简单，容易实现高性能，但给优化编译程序带来困难

#### 六、前后台系统

1.特点：

前台：中断实现；处理对时间要求严格事件、突发事件

后台：轮询多任务实现；处理对时间要求不严格事件

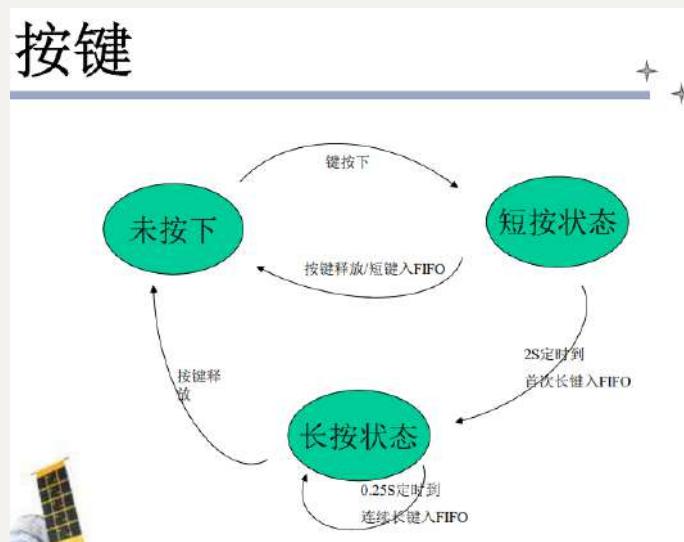
## 2. 前后台程序编写原则：

- 消除阻塞：任何任务都不能阻塞CPU
- 消除前后台之间的直接关联性：使用缓冲区
- 使用函数重入：前后台程序重叠调用同一函数
- 临界代码(不能分割代码)：例如关中断

## 七、状态机(状态图)在事件触发程序结构中的应用

状态机：一种建模方式，可以描述软件并发行为。

### 1. 可以实现基于状态机的程序建模



### 2. 可以通过状态转移图生成代码：

- 在状态中判断事件(事件查询)：在当前状态下，根据不同的事件来执行不同的功能，执行后再做状态转移
- 在事件中判断状态(事件触发)：在每个事件的中断或查询到事件发生函数内，判断当前状态，根据状态执行不同功能，执行后再做相应的状态转移

## 第三讲 ARM基本编程模型

### 一、ARM的历史(了解) ARM=Advanced RISC Machines

- 1985年：第一个ARM原型Acorn计算机诞生
- 20世纪80年代后期：ARM开发出Acorn的台式机产品

- 1990年：成立Advanced RISC Machines Limited(公司)：设计了大量RISC处理器，只设计芯片，而不生产
- 20世纪90年代：ARM32位嵌入式RISC处理器扩展至全世界
- 1991年：推出第一款RISC嵌入式微处理器ARM6
- 1993年：推出ARM7
- 1995年：提出Thumb扩展指令集结构，为16位系统增加32位性能
- 至今：占据32位嵌入式微处理器75%以上市场份额，绝大多数手机及PDA产品均采用ARM体系的嵌入式处理器

## 二、Thumb工作状态(了解)

**“16位的指令长度，32位的执行效率”：**Thumb工作状态是ARM架构中一种用于提高代码密度的指令集状态。在Thumb工作状态下，ARM指令集中的一部分指令被编码成更短的16位指令，相比于ARM指令集的32位指令，Thumb指令集可以使程序更加紧凑，占用更少的存储空间。

## 三、ARM Cortex系列的分类

1. **Cortex: A:** 面向尖端的基于虚拟内存的操作系统和用户应用，如应用像Linux、Windows CE和Symbian等操作系统的消费娱乐产品和无线产品
2. **Cortex: R:** 面向需要运行实时操作系统来进行控制应用的系统，包括汽车电子、网络和影像系统
3. **Cortex: M:** 面向那些对开发费用非常敏感同时对性能要求不断增加的嵌入式应用(如微控制器、汽车车身控制系统和各种大型家电)

## 四、ARM的7种运行模式(用户、系统、5种异常模式)

处理器模式	说明	备注
用户 (usr)	正常程序工作模式	不能直接切换到其它模式
系统 (sys)	用于支持操作系统的特权任务等	与用户模式类似，但具有可以直接切换到其它模式等特权
快中断 (fiq)	支持高速数据传输及通道处理	FIQ异常响应时进入此模式
中断 (irq)	用于通用中断处理	IRQ异常响应时进入此模式
管理 (svc)	操作系统保护代码	系统复位和软件中断响应时进入此模式
中止 (abt)	用于支持虚拟内存和/或存储器保护	在ARM7TDMI没有多大用处
未定义 (und)	支持硬件协处理器的软件仿真	未定义指令异常响应时进入此模式

- 特权模式：除了用户模式外的所有模式
- 异常模式：除了系统模式、用户模式外的所有模式

## 五、ARM异常的概念

异常是指由于程序执行中出现的某些特殊情况而导致正常控制流程被中断的事件

可分为3类：

- 中断：由外部事件触发的异常
- 终止：由于错误或异常情况而导致程序无法继续执行的事件
- 陷阱：有意引发的异常，通常用于在程序执行过程中插入一些特殊操作或系统调用

## 六、37个寄存器的组织形式(R13、R14、R15以及状态寄存器作用)

组织形式：

## ➤ ARM有37个32位长的寄存器：

- 1个用作PC (program counter) ；
- 30个用作一般通用寄存器；
- 1个用作CPSR(current program status register)；
- 5个用作SPSR (saved program status registers) 。

➤ 这些寄存器不能被同时访问；与微处理器的工作状态及具体的运行模式有关。

➤ 通用寄存器R0~R14、程序计数器PC、一个或两个状态寄存器通常是可访问的。

**R13:** 通用寄存器，常作为堆栈指针(**SP**)，尤其是Thumb指令集的某些指令强制要求；

由于处理器的每种运行模式均有自己独立的物理寄存器**R13**，在初始化部分，都要初始化每种模式下的**R13**，这样，当程序的运行进入异常模式时，可以将需要保护的寄存器放入**R13**所指向的堆栈，而当程序从异常模式返回时，则从对应的堆栈中恢复。

**R14:** 链接寄存器(**LR**)：当执行BL子程序调用指令时，可以从R14中得到 R15(程序计数器PC)的备份。其他情况下，R14 用作通用寄存器。

在每一种运行模式下，都可用**R14**保存子程序的返回地址，当用**BL**或**BLX**指令调用子程序时，指令先将下一条指令的**PC**值拷贝给**R14**；当执行完子程序后，再将**R14**的值拷贝回**PC**，即可完成子程序的调用返回。

在结构上有两个特殊功能

- 在每种模式下，模式自身的R14版本用于保存子程序返回地址
- 用于异常处理的返回：当发生异常时，将R14对应的异常模式版本 设置为异常返回地址(有些异常有一个小的固定偏移量)

**R15:** 程序计数器(**PC**)：指向正在取指的地址

- ARM状态下，位[1:0]为0，位[31:2]保存PC；
- Thumb状态下，位[0]为0，位[31:1]保存PC；
- R15虽然也可用作通用寄存器，但一般不这么使用，因为对R15的使用有一些特殊的限制，当违反了这些限制时，程序的执行结果是未知的。
- 由于ARM体系结构采用了多级流水线技术，对于ARM指令集而言，PC总是指向当前指令的下几条指令的地址，即PC的值为当前指令的地址值加8个字节。

状态寄存器：程序状态寄存器(CPSR/SPSR)R16

- CPSR：当前程序状态寄存器(Current Program Status Register)，可在任何运行模式下被访问；包括条件标志位、中断禁止位、当前处理器模式标志位，以及其他一些相关的控制和状态位
- SPSR：程序状态保存寄存器，异常模式中使用，异常发生时，用于保存CPSR的值，从异常退出时则可由SPSR来恢复CPSR

## 七、分组寄存器的作用

R8-R14以及SPSR寄存器对特定的异常模式都进行了分组，其主要作用是采用专用的分组寄存器用于快速异常处理，保证当异常发生时，对应的异常模式下其专用寄存器能够不被其他模式占用，专用于此模式的异常处理。

# 第四讲 ARM指令系统

## 一、充分理解Load/Store结构

- 在通用寄存器中操作，从存储器中读某个值，操作完再将其放回存储器中
- 只有Load和Store指令允许直接在内存和通用寄存器之间进行数据的读取和存储，其他例如算数和逻辑运算指令主要在通用寄存器之间进行，不直接涉及内存。有助于简化指令集结构，提高指令执行效率

## 二、指令分类(主要掌握数据处理类、Load/Store类和跳转类)

指令可分为6类

1. 数据处理指令：完成寄存器中数据的算数和逻辑运算操作，所有操作数都是32位宽度

寻址方式：

- 立即数寻址

## ➤ 也叫立即数寻址

□ 操作数就包含在指令的32位编码中

□ 如:      ADD R0, R0, #1

              AND R3, R4, #0xFF

## ➤ 只有第二源操作数可用立即数

➤ 立即数要以“#”为前缀，“#”后加“0x”或“&”表示16进制，“0b”表示二进制，“0d”或缺省表示十进制。

- 寄存器寻址

➤ 操作数的值在寄存器中，指令中的地址码字段指出的是寄存器编号，指令执行时直接取出寄存器值操作。

□ 如:      MOV R1, R2;      R2 -> R1

□            SUB R0, R1, R2; R1 - R2 -> R0

- 寄存器移位寻址

➤ 寄存器移位寻址是ARM指令集**特有的**寻址方式，是寄存器寻址方式的增强形式

□ 如：MOV R0, R2, LSL #3;

R2 的值左移3位，结果放入  
R0，即  $R0 = R2 * 8$

□ ANDS R1, R1, R2, LSL R3 ;

R2 的值左移R3位，然后和R1  
相与操作，结果放入R1



**注：第2操作数的12位编码空间有冗余位**

分类：

- 数据传送指令
- 算术、逻辑运算指令：
- 比较、测试指令
- 乘法指令

## 2. Load/Store指令

寻址方式：

- 基址寻址

➤ **Load/Store指令的寻址方式**

➤ 利用一个寄存器的值作为存储器地址

□ 如： LDR R0, [R1]

STR R0, [R1]

- 变址寻址：存储器地址由基址寄存器和地址偏移量组成

变址模式：

## ➤ 前变址模式

□ LDR R0, [R1, #4] ;  $R0 \leftarrow \text{mem}_{32}[R1+4]$

## ➤ 自动变址模式（事先更新方式）

□ LDR R0, [R1, #4]! ;  $R0 \leftarrow \text{mem}_{32}[R1+4]$   
;  $R1 \leftarrow R1+4$

## ➤ 后变址模式（事后更新方式）

□ LDR R0, [R1], #4;  $R0 \leftarrow \text{mem}_{32}[R1]$   
;  $R1 \leftarrow R1+4$

地址偏移量形式:

1. 偏移量为立即数:

- 存储器地址为基址寄存器值加上/减去立即数偏移量
- U控制位为1时加上偏移量； U控制位为0时减去偏移量
- 偏移量占12bit (**offset\_12**)
- 例子:
  - LDR R0, [R1, #4]
  - LDR R0, [R1, #-4]

2. 偏移量为寄存器:

- 存储器地址为基址寄存器值加上/减去索引寄存器的值
- U控制位为1时加上索引寄存器的值； U控制位为0时减去索引寄存器的值
- 例子：
  - LDR R0, [R1, R2]
  - LDR R0, [R1, -R2]

3. 偏移量为寄存器及一个移位常数：

- 存储器地址为基址寄存器值加上/减去一个地址偏移量
- U控制位为1时加上偏移量； U控制位为0时减去偏移量
- 该偏移量由索引寄存器通过移位得到
  - 移位的方法如“寄存器移位寻址”方式
- 例子：
  - LDR R0, [R1, R2, LSL #2]

所以可以组合出9中类型的变地寻址方式

3. 跳转指令：跳转范围是+-32MB

寻址方式：相对寻址

- 变址寻址的一种变通
- 其基址寄存器是程序计数器PC
- 主要用于分支指令
- 例子：

BL      SUBR	; 转移到SUBR
	; 返回到此处
SUBR	; 子程序入口
.....	
MOV PC, R14	; 返回

- **B**
  - 跳转到指定的地址执行程序
- **BL**
  - 将下一条指令的地址拷贝到链接寄存器(R14/LR)中，然后跳转到指定地址运行程序
- **BX**
  - 带状态切换的跳转指令
- **BLX**
  - 带连接和状态切换的跳转指令

三、掌握3地址指令格式，理解条件码和条件执行

ARM指令基本格式：

## ➤ 基本格式

<opcode>{<cond>} {S} <Rd>,<Rn>{,<operand2>}

## ➤ 说明

- opcode 指令助记符，如LDR, STR 等
- cond 执行条件，如EQ, NE 等
- S 是否影响CPSR 寄存器的值
- Rd 目标寄存器
- Rn 存放第一操作数的寄存器
- operand2 第二个操作数

注：{}为可选项

3地址指令格式： A = B op C

其中A：目标操作数(存储计算结果)的地址

B和C是两个源操作数地址:从中获取数据进行op操作

条件码和条件执行:

## ➤ ARM状态时，几乎所有的指令均根据CPSR中条件码的状态和指令的条件域有条件的执行

- 条件满足时，指令被执行，
- 否则指令被忽略（相当于NOP指令）。

## ➤ 条件码可用2个字符表示

- 可以添加在指令助记符的后面和指令同时使用

## ➤ 条件码占指令码的高4位

- 16种条件标志中，只有15种可用（如图），第16种（1111）为系统保留。

条件码表

条件码助记符	标志	含义
EQ	Z=1	相等
NE	Z=0	不相等
CS/HS	C=1	无符号数大于或等于
CC/LO	C=0	无符号数小于
MI	N=1	负数
PL	N=0	正数或零
VS	V=1	溢出
VC	V=0	没有溢出
HI	C=1, Z=0	无符号数大于
LS	C=0, Z=1	无符号数小于或等于
GE	N=V	带符号数大于或等于
LT	N! =V	带符号数小于
GT	Z=0, N=V	带符号数大于
LE	Z=1, N! =V	带符号数小于或等于
AL	任何	无条件执行（指令默认条件）

四、掌握寻址方式，重点是：立即寻址、寄存器寻址、寄存器移位寻址、基指寻址和变址寻址。（略）

## 第五讲 ARM指令详细介绍

### 一、了解具体指令

#### ARM指令详细介绍

### 二、BIC指令的作用

格式：BIC{条件}{S} 目的寄存器，操作数1，操作数2

作用：用于清除操作数1的某些位，并把结果放置到目的寄存器中。

操作数1应是一个寄存器，操作数2可以是一个寄存器或被移位的寄存器或一个立即数。操作数2为32位的掩码，如果在掩码中设置了某一位则清除这一位，未设置掩码位保持不变。

```
# 从寄存器R0中获取一个值与立即数“3”的二进制补码，也就是11进行按照位与运算，结果存储回R1
BIC R1, R0, #3      // 清除低两位
```

### 三、长乘指令和短乘指令的区别

长乘指令：

- SMULL 64位有符号数乘法指令
- SMLAL 64位有符号数乘加指令
- UMULL 64位无符号数乘法指令
- UMLAL 64位无符号数乘加指

短乘指令：

- MUL 32位乘法指令
- MLA 32位乘加指令

区别：长乘指令适用于大整数乘法，尤其是处理溢出或累加情况，结果通常需要多个寄存器来存储；短乘指令执行较小整数简单的整数乘法，结果存储在单个寄存器中。

### 四、交换指令的特殊性

交换指令能够在存储器和寄存器之间交换数据。数据交换指令主要用于实现信号量操作，信号量用于进程间的同步和互斥。特殊性在于是原子操作，在并发环境下不会被中断

例：SWP指令：

#### ➤ SWP指令

□ SWP指令的格式为：

**SWP{条件} 目的寄存器, 源寄存器1, [源寄存器2]**

□ SWP指令用于将源寄存器2所指向的存储器中的字数据传送到目的寄存器中，同时将源寄存器1中的字数据传送到源寄存器2所指向的存储器中。显然，当源寄存器1和目的寄存器为同一个寄存器时，指令交换该寄存器和存储器的内容。

SWP R0, R1, [R2] ;  $R0 \leftarrow [R2], [R2] \leftarrow R1$

SWP R0, R0, [R2] ;  $R0(\text{目的}) \leftarrow [R2], [R2] \leftarrow R0(\text{源})$   
即实现了寄存器R0的内容与存储器地址[R2]的内容的交换

# 信号量操作举例

SEM EQU 0x40003000

...

SEM\_WAIT

MOV R1, #0

LDR R0, = SEM

SWP R1, R1, [R0] ;取出信号量，  
并设置其为0

CMP R1, #0 ;判断是否有信号

BEQ SEM\_WAIT ;若没有，则等待



## 第六讲 ARM汇编

一、了解ARM汇编，能完成简单的汇编程序设计

(参考课件最后的几个汇编示例)

第六讲课件

## 第七讲 ARM下的C编程

一、了解C运行时库

C运行时库(C Runtime Library)用于在C程序运行时提供支持的函数和例程集合，包含C语言所需的基本功能和支持：

- ARM的C编译器支持ANSI C运行时库
- ARM C运行时库以二进制形式提供
- 用户可以建立自己的运行时库

## 二、宏和函数的区别

宏：在C程序中定义的命名代码段。

1. 编译器会将相关代码放到宏名出现的每一个地方，宏的代码在任何出现宏名的地方都会编译一次；使用宏时不需要保存/恢复上下文，也不必返回
2. 在编译器预处理阶段会被直接展开到代码中，编译时用宏内容替换宏调用位置，可能导致代码膨胀
3. 代码简单用宏

函数：1. 函数的代码只需要编译一次

2. 在编译时定义，但实际执行发生在运行时，会有函数调用开销
3. 代码复杂用函数

## 三、什么是内嵌汇编

标识符：`__asm`：用于告诉编译器后面是汇编代码

语法：

```
__asm
{
    指令[;注释]
    .....
}
```

# 示例——使能中断

```
void enable_IRQ(void){  
    int tmp  
    __asm //嵌入汇编代码  
    {  
        MRS tmp, CPSR //读取CPSR的值  
        BIC tmp, tmp, #0x80 //将IRQ中断禁止位I清零，即允许IRQ中断  
        MSR CPSR_c, tmp //设置CPSR的值  
    }  
}
```

## 四、\_irq、volatile关键字的作用

**\_irq:** 使用该关键字声明的函数可以被用作异常中断的中断处理程序，该函数通过将lr-4的值赋予PC寄存器，并将 SPSR的值赋予CPSR实现函数返回

**volatile:** 用于声明变量，告诉编译器该变量可能在程序之外修改；编译时不能优化对 volatile 变量的操作；不能对 volatile 变量使用缓冲技术

## 五、汇编和 C 的混合编程(主要理解参数如何传递)

参数传递规则：

- 参数个数可变的子程序参数传递规则：
  - a. 参数不超过4个时，使用R0~R3来传递；
  - b. 参数超过4个时，可以使用数据栈来传递
- 参数个数固定的子程序参数传递规则：
  - a. 第一个整数参数，通过R0~R3来传递
  - b. 其他参数通过数据栈来传递
  - c. 有关浮点运算，需特别处理
- 子程序结果返回规则：
  - a. 结果为一个32位整数时，可以通过寄存器R0返回

b. 结果为一个64位整数时，可以通过寄存器R0和R1返回，依次类推

## 六、了解 ARM 异常响应过程(理解为主)，及异常向量表的作用。

### ARM异常处理:异常中断发生时

1. 系统执行完当前指令后，跳转到相应的异常中断处理程序处执行
2. 执行完成后，程序返回到发生中断的指令的下一条指令处执行
3. 进入异常中断处理程序时要保存现场，返回时要恢复现场

### 异常向量表：

- 当异常发生时，处理器会把PC设置为一个特定的存储器地址
- 这一地址放在被称为向量表(vector table)的特定地址范围内
- 异常向量表通常放在存储地址的低端



## 第八讲

### 1. 了解指令集架构（ISA）和国内 ISA 生态【掌握程度：了解】

- 指令：是指处理器进行操作的最小单元（如算术/逻辑运算）；
- 指令集：顾名思义是指一组指令的集合；
- 指令集架构：又称为“处理器架构”，不仅是指令的集合，还包括编程需要的硬件信息，如支持的数据类型、存储器、寄存器状态、寻址模式和寄存器模型等；

- 指令集架构（ISA, Instruction Set Architecture）才是区分不同CPU的标准；
- 微架构：处理器的具体硬件实现方案称为微架构；
- 常见指令集架构：**x86(CISC)**, SPARC, **MIPS**, Power(国产), Alpha(国产), ARC, **ARM(RISC)**, Intel

## 2. RISCV的主要特征【掌握程度：能够阐述清楚】

RISCV，是一种基于“精简指令集（RISC）”原则的指令集架构，它具有开源、重新设计(后发优势)、简单哲学、模块化指令集、指令集可扩展的特点。

## 3. 鲲鹏 920 使用的指令集架构是 ARM v8 64 位

## 4. 鲲鹏 920 生产使用的制程是 7nm

## 5. 相比于上一代的鲲鹏 916，鲲鹏 920 处理器的计算核数提升 1 倍，最多支持 64 核

# 第九讲

## 1. 嵌入式系统最小系统的构成（电源、时钟、复位、存储）

CPU能够运行所需的模块有（最小系统）：电源、时钟、复位、内存、调试接口（JTAG）

## 2. DC-DC 转换器的常见形式：线性稳压器、开关稳压器和充电泵

## 3. 晶体和晶振的区别

时钟一般由晶体振荡器提供，而晶体和晶振是构成晶体振荡器的两种方式：

- 晶体(无源)
  - 封装内部只含有晶体，没有内部电源
  - 驱动电路由设计者提供

- 晶振(有源)

- 封装中包含了完整的晶体振荡器电路
- 需要电源

## 4. 复位的基本功能

- 在系统上电时提供复位信号
- 保证能够进行手动复位

## 5. SRAM、DRAM、SDRAM 和 Flash (NandFlash、NorFlash) 的异同

- SRAM(静态RAM)

- 数据存入静态RAM后，只要电源维持不变，其中存储的数据就能够一直维持不变，不需要刷新操作
- 读写速度快
- 由触发器构成基本单元，接口简单
- 存储单元结构复杂，集成度较低
- 常常用作高速缓冲存储器[读写速度快]

处理器内部集成存储器多选择SRAM

- DRAM(动态RAM)

- 依靠电容存储信息，需要不断刷新
- 读写速度慢
- 集成度高，成本低
- 地址引脚少，地址总线采用多路技术，接口复杂

多用于外部存储器扩展(外存)[读写速度慢]

- SDRAM(同步动态RAM)

- SDRAM因为要同CPU和芯片组共享时钟，所以芯片组可以主动的在每个时钟的上升沿发给引脚控制命令
- 动态RAM加上同步特性[在每个时钟的上升沿发给引脚控制命令]

- Nor Flash

- 芯片内执行

- 读速度快（相比Nand Flash）
  - 写入与擦除速度很低
  - 擦除按块进行，写入前必须先擦除
  - 带有SRAM接口，有足够的地址引脚来寻址，可以很容易地存取其内部的每一个字节
  - 常用来存储代码
- Nand Flash
    - NAND读和写操作按**512**字节的块进行
    - 写入与擦除速度比**Nor Flash**快
    - 擦除按块进行，写入前必须先擦除
    - NAND的擦除单元更小，相应的擦除电路更少
    - NAND器件使用复杂的I/O口来串行地存取数据
    - 常用来存储数据

## 6. 理解存储器映射

存储器映射到物理地址，通过将存储器按照物理地址划分为不同模块，进行RAM, DRAM, SDRAM, FLASH等存储设计。

## 第十讲（了解为主）

### 1. 外部接口（**GPIO** 控制、触摸屏和 **LCD**，以及 **ADC** 和 **DAC** 的概念）

I/O（Input/Output）接口是一个微控制器必须具备的最基本的外设功能。

- 每个I/O口一般都对应了两个寄存器：
  - 数据寄存器：数据寄存器的各位都直接引到芯片外部
  - 控制寄存器：控制数据寄存器中每位的信号流通方向和方式
- GPIO  
General-Purpose I/O ports，也就是通用I/O口，是I/O的最基本形式
- LCD

液晶显示器（Liquid Crystal Display），液晶显示器是一种被动光源的显示器，自身不能发光，只能借助外界光源，具有省电、体积小、低成本、低功率等特点，被广泛应用于嵌入式系统中

液晶：以液态形式存在的晶体

- 有电流流过，液晶分子会以电流的方向进行排列；没有电流时，平行排列
- 基本原理：通过给不同的液晶单元供电，控制其光线的通过与否而达到显示的目的
- 触摸屏  
触摸屏由触摸检测部件和触摸屏控制器组成，按照触摸屏的工作原理和传输信息的介质主要分为：电阻式，电容感应式，红外线式，表面声波式四种
- ADC  
模/数转换器就是把电模拟量转换成数字量的电路
- DAC  
数/模转换器就是把数字量转换成模拟量的电路

## 第十一讲

### 1. 嵌入式操作系统的特点

相对于通用操作系统，嵌入式操作系统更为精巧，但并不意味着可以通过裁剪通用操作系统来实现，它有自身的特点：

- 实时性
- 可移植性
- 可配置、可裁剪性
- 可靠性
- 应用编程接口

每个操作系统提供的系统调用的功能和种类都不同

### 2. 嵌入式操作系统的任务

主要任务是尽可能地屏蔽底层硬件的差异，对上层应用软件和底层硬件提供标准化服务

### 3. 嵌入式操作系统的功能

内核是嵌入式操作系统的基础，具有以下功能：

- 任务管理
- 中断管理

- 通信, 同步和互斥机制
- 内存管理
- I/O管理

## 4. 嵌入式操作系统的实时性

实时性是实时内核最重要的特征之一

- 实时系统的正确性不仅依赖于系统计算的逻辑结果(计算逻辑), 还依赖于产生这些结果的时间(计算时间)
  - 相关概念
    - 确定性
      - 实时性是指内核应该尽可能快的响应外部事件
      - 确定性是指对事件响应的最坏时间是可预知的
    - 响应性
      - 确定性关心系统在识别外部事件之前的延迟(响应启动延迟)
      - 响应性关心的是在识别外部事件后, 系统要花多长时间来服务该事件(响应时间)
    - 响应时间
- 确定性和响应性结合在一起构成事件响应时间。
- 中断响应时间 = 最长关中断时间 + 保护CPU内部寄存器的时间 + 进入中断服务函数的执行时间 + 开始执行中断服务程序(ISR)的第一条指令时间
  - 任务响应时间 = 中断响应时间+中断服务时间
  - 对于强实时内核, 响应时间应该在  $\mu s$  级

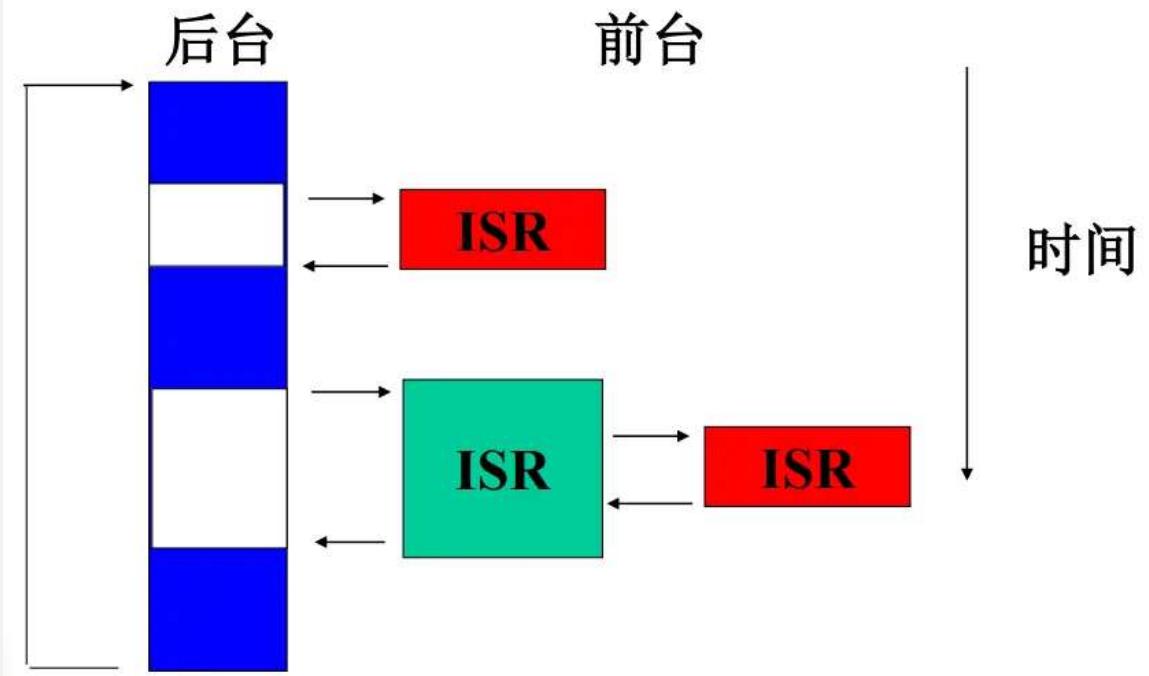
## 5. 多任务程序设计结构(前后台结构、事件触发结构、操作系统的基本概念)

前后台结构【掌握程度: 能够阐述清楚】

后台循环, 前台中断

- 后台行为: 应用程序是一个无限的循环, 循环中调用相应的函数完成相应的操作, 这部分可以看成后台行为

- 前台行为: 中断服务程序处理异步事件, 这部分可以看成前台行为



## 6. 多任务系统的相关概念

嵌入式实时系统中, 多采用多任务程序设计的方法, 其特点有:

- 单个任务规模较小, 容易编码和调试
- 任务间独立性高、耦合性小, 便于扩充功能
- 系统实时性强, 以保证紧急事件得到优先处理

任务的概念(可以和前后台结构做对比)

进程是资源分配的最小单位, 线程是进程内部一个相对独立的控制流, 是调度执行的最小单位

### 【基本概念】

- 大多数实时内核都把整个应用当作一个没有定义的进程, 应用则被划分为多个任务来处理
- 整个内核是一个单进程/多线程模型, 简单称为多任务模型

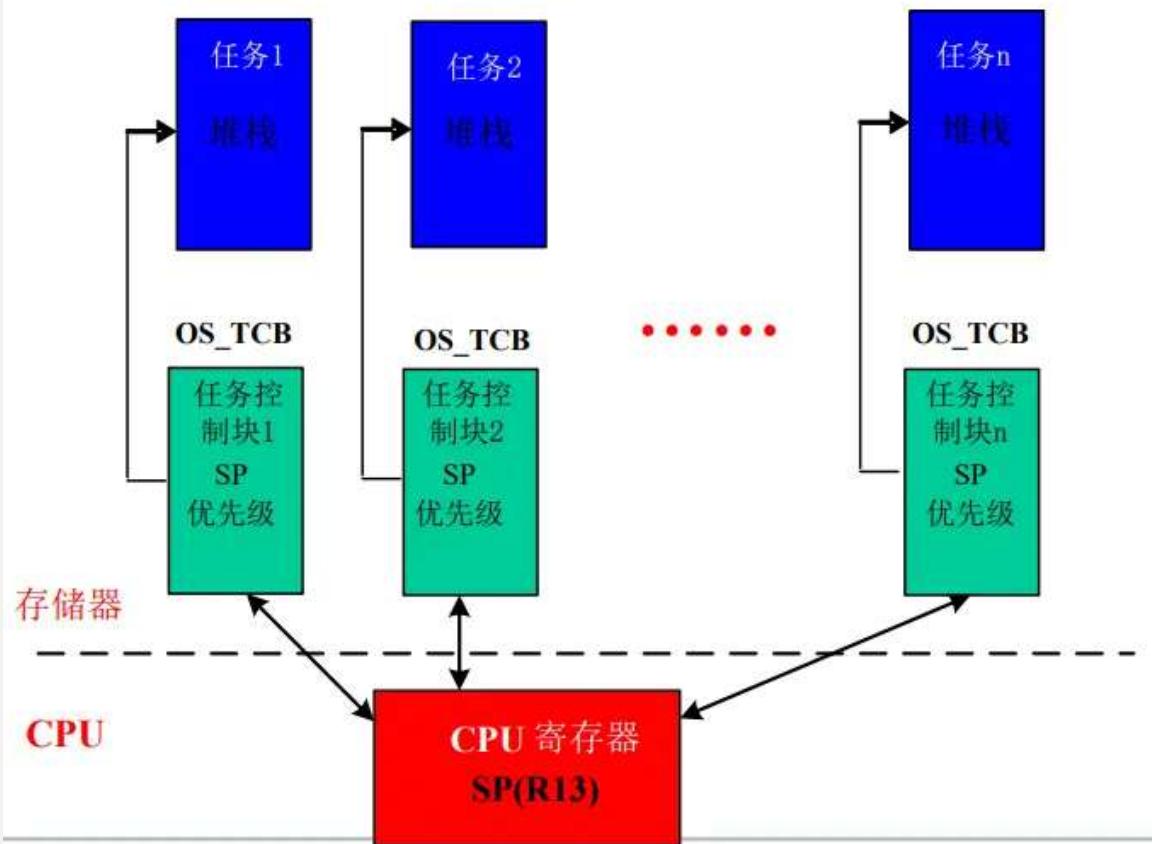
### 【概念阐述】

多任务系统是一个单进程或多进程的系统内核, 其中多个进程是由内核将整个应用划分成多个任务进行处理而建立的.

### 【任务的要素】

- 代码: 一段可执行的程序
- 初始数据: 程序执行所需的相关数据

- 堆栈: 保存局部变量和现场的存储区
- 任务控制块: 包含任务相关信息的数据结构以及任务执行过程中所需要的所有信息  
休眠、就绪、运行、挂起、被中断



### 【任务的特点】

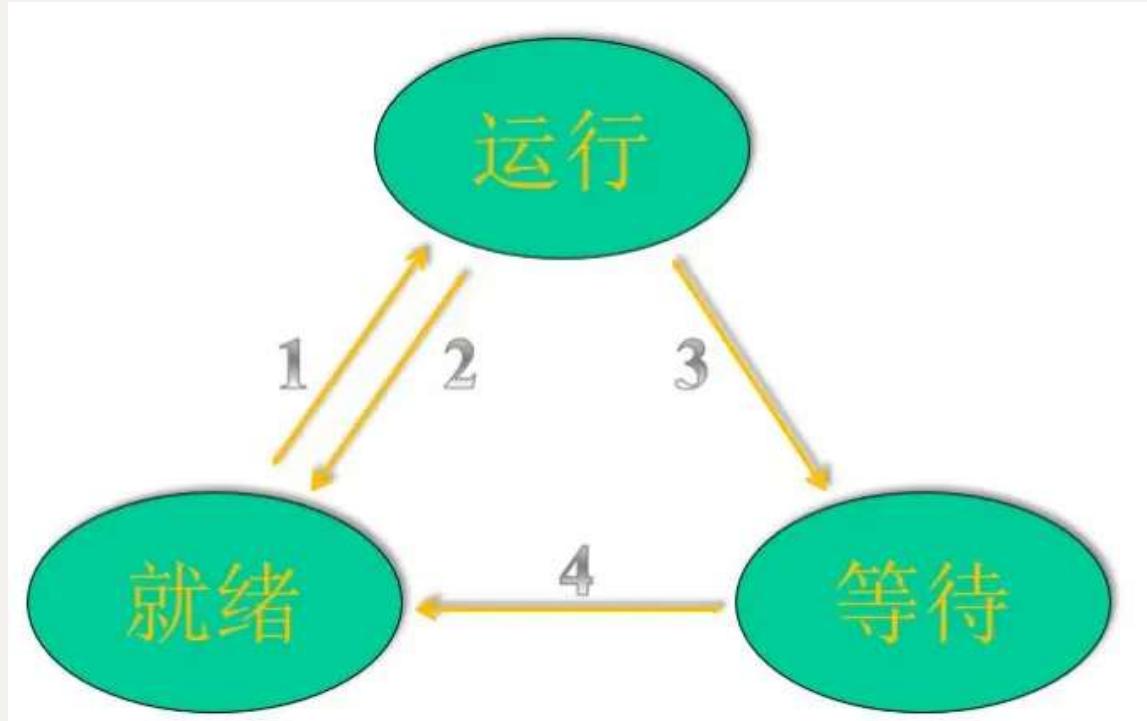
动态性(就绪, 运行或等待), 并行性(同时存在多个任务, 宏观上并行), 异步独立性(各任务相互独立, 运行速度不可预知)

任务的三个基本状态和互相转换关系

- 运行状态(running)  
该任务已获得运行所必需的资源, 它的程序正在处理机上执行
- 阻塞状态(wait)  
任务正等待着某一事件的发生而暂时停止执行; 这时, 即使给它CPU控制权, 它也无法执行, 则称该任务处于阻塞态

- 就绪状态(ready)

任务已获得除CPU之外的运行所必需的资源，一旦得到CPU控制权，立即可以运行



#### 1. 就绪→运行

- 调度程序选择一个新的进程运行(进程被调度)

#### 2. 运行→就绪

- 运行进程用完了时间片(时间片用完)
- 运行进程被中断，因为一高优先级进程处于就绪状态(被高优先级进程抢占)

#### 3. 运行→等待

进程必须等待某个事件：

- OS尚未完成服务
- 对一资源的访问尚不能进行
- 初始化I/O且必须等待结果

#### 4. 等待→就绪

所等待的事件已经发生

### 任务的切换(任务上下文)[掌握程度：能够阐述清楚]

- 保存当前任务的上下文

当多任务内核决定运行另外的任务时，任务切换要求保存正在运行任务的当前状态，即**CPU**寄存器中的全部内容被保存到任务的当前状态保存区(任务自己的堆栈区)

- 恢复需要运行任务的上下文

原任务状态入栈工作完成后，就把下一个将要运行任务的当前状态从该任务的堆栈中重新装入**CPU**寄存器，并开始下一个任务的运行

## 任务的调度(任务级调度、中断级调度)

### 【调度的功能】

调度只是一个函数调用，可在内核各个部分调用

- 用来确定多任务环境下任务执行的顺序(任务执行顺序)
- 用来确定任务获得CPU资源后能够执行的时间(任务执行时间)[基于时间片的调度]

### 【调度点与调度时机】

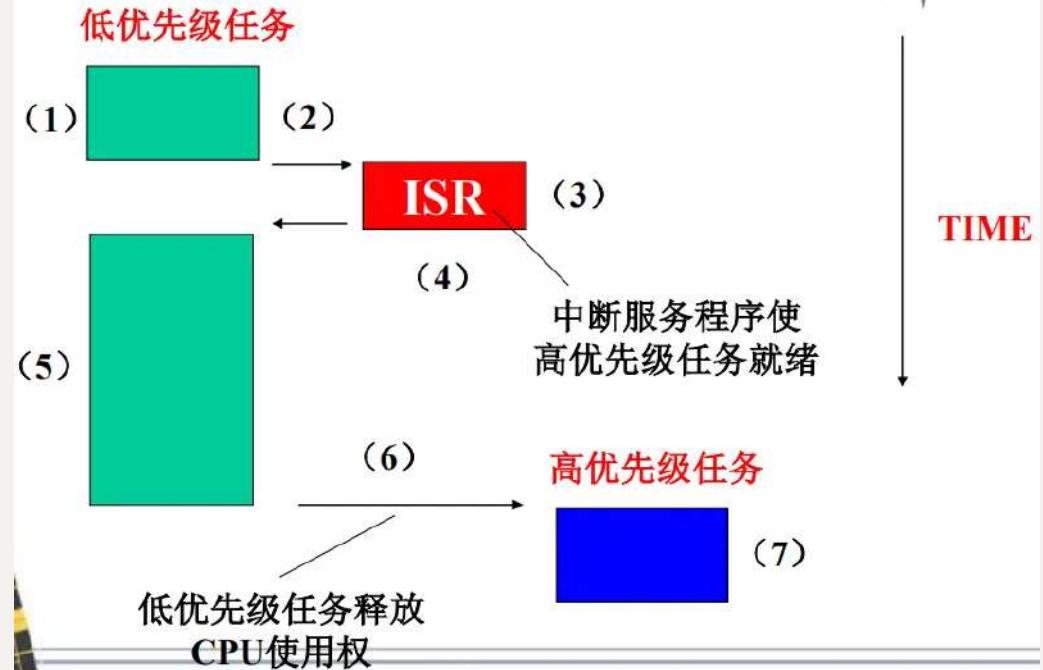
调用调度程序的具体位置称为调度点：

- 中断服务程序的结束位置
- 任务因等待资源而进入等待状态

## 调度策略(抢占式调度、非抢占式调度)

- 实时任务就绪的原因
  - 中断处理过程中使实时任务就绪：存在任务请求中断，中断服务程序会使得中断请求任务就绪【中断处理程序】
  - 当前运行任务调用操作系统功能，使实时任务就绪【系统调用】
- 非抢占式调度[能够阐述清楚]
  - 低优先级任务运行过程中，一个中断到达；
  - 若中断被允许，CPU进入中断服务程序；
  - 中断处理过程使一个高优先级任务就绪；
  - 中断完成后，CPU归还给原先被中断的低优先级任务；
  - 低优先级任务继续运行；
  - 低优先级任务完成或因其它原因被阻塞而释放CPU，内核进行任务调度，切换到就绪的高优先级任务；【调度点】

- 高优先级任务运行



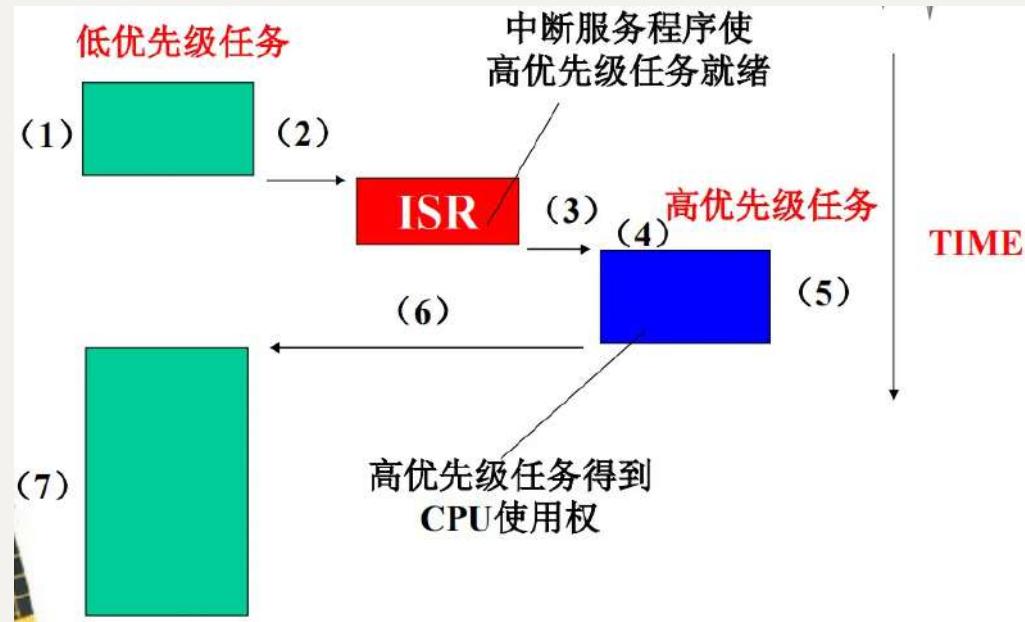
- 特点

- 任务运行空间是封闭的，几乎不需要使用信号量保护共享数据
- 内核的任务级响应时间是不确定的，完全取决于当前任务何时释放CPU

- 抢占式调度[能够阐述清楚]

- 低优先级任务运行过程中，一个中断到达；
- 若中断被允许，CPU进入中断服务程序；
- 中断处理过程使一个高优先级任务就绪；
- 中断完成后，内核进行任务调度，让刚就绪的高优先级任务获得CPU;  
【调度点】
- 高优先级任务运行；
- 高优先级任务完成或因其它原因被阻塞而释放CPU，内核进行任务调度；  
【调度点】

- 低优先级任务获得CPU，从被中断的代码处继续运行



- 特点

- 最高优先级的任务一旦就绪，总能得到CPU的控制权
- 任务运行空间不再是封闭的，任务不可使用不可重入型函数
- 需要对共享数据进行必要的保护
- 实时操作系统大多基于抢占式内核

## 7. 对可重入型函数的理解 [掌握程度：深入理解]

- 可重入型函数

- 该函数可以被一个以上的任务调用，而不必担心数据被破坏 [基本特点]
- 可重入型函数任何时候都可以被中断，一段时间以后又可以运行，而相应数据不会丢失 [可以随时中断或运行]
- 可重入型函数只使用局部变量，即变量保存在CPU寄存器中或堆栈中  
【注解】这使得可重入型函数在发生中断时，其数据会被中断处理程序自动保存，当中断返回后，数据从堆栈中自动恢复，因此不会丢失任何数据。

➤ 一个不可重入型函数的例子

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

➤ 一个可重入型函数的例子

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

## 8. 对临界区的理解(关中断、开中断)[掌握程度: 深入理解]

### 【基本概念】

临界区，又称为代码的临界区，指处理时不可分割的代码，代码一旦开始执行，则不允许任何中断打断。

### 【临界区的保护】

- 在进入临界区之前要关中断
- 临界区代码执行完以后要立即开中断  
    【内核的关中断时间】
- 关中断影响中断延迟时间
- 内核在中断响应时间上的差异主要来自内核最大关中断时间

## 9. 对共享资源的互斥管理

实现资源互斥访问的方法很多，一般有

- 关中断
- 禁止任务切换
- 使用测试并置位指令

- 使用信号量（提供任务间通信、同步和互斥的最优选择

比较项目	关中断	禁止任务切换	使用测试并置位指令	使用信号量
锁定范围	互斥粒度最强，锁定所有外部可屏蔽中断	锁定所有任务	锁定所有使用该指令访问共享资源的代码	只影响竞争共享资源的任务
对系统响应时间的影响	若关中断时间过长，对系统的响应性能有很大影响	若禁止切换时间过长，影响系统的响应性能	较小	对系统的响应性能有一定影响，可能导致优先级反转
系统开销	小	小	小	较大
注意事项	关中断时间要尽量短	禁止调度的时间要尽量短	不是所有处理器都支持该指令，影响可移植性	需要采用一定的策略解决优先级反转问题

## 10. 实时内核的重要性能指标

### 【时间性能指标】

- 中断延迟时间

指从中断发生到系统获知中断，并开始执行中断服务程序所需要的最大滞后时间

- 中断延迟时间 = 最大关中断时间 + 中断嵌套的时间 + 硬件开始处理中断到开始执行ISR第一条指令之间的时间

实时内核应尽量使内核最大关中断时间减小

缩短关中断时间：在临界区的一些非关键代码段开中断，增加内核代码中的可抢占点

- 中断响应时间

中断响应时间指从中断发生到开始执行用户中断服务程序的第一条指令之间的时间

- 中断响应时间 = 中断延迟时间 + 保存CPU内部寄存器的时间 + 该内核的ISR进入函数的执行时间

- 中断恢复时间

中断恢复时间指用户中断服务程序结束后回到被中断代码之前的时间。（对抢占式内核还应包括可能发生的任务切换时间）

- 中断恢复时间 = 恢复CPU内部寄存器的时间 + 执行中断返回指令的时间

- 任务响应时间

任务响应时间指从任务对应的中断产生到该任务真正开始运行所花费的时间，又称调度延迟

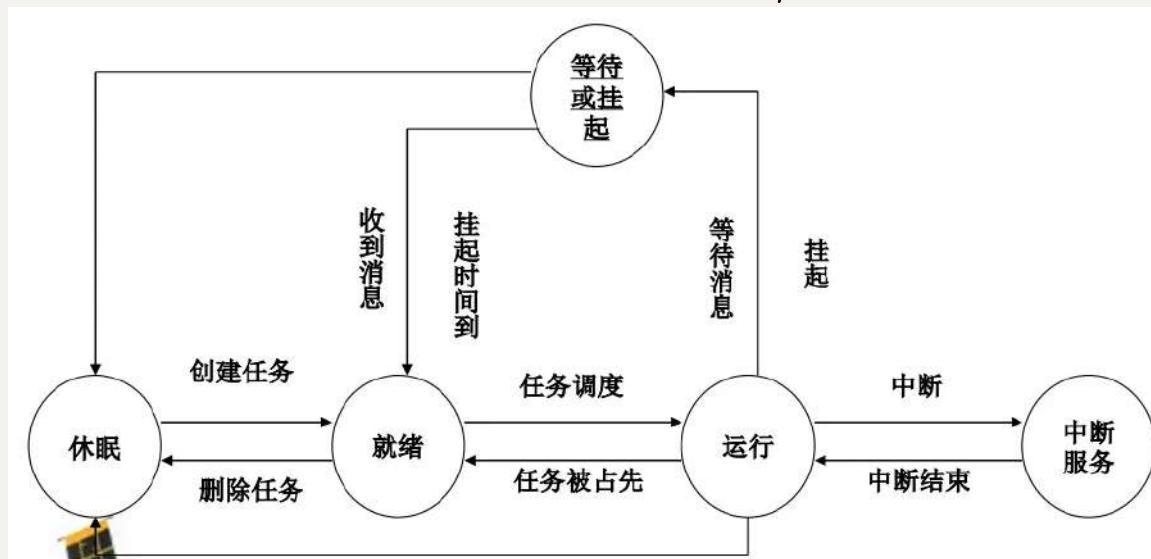
- 内核调度算法是决定调度延迟的主要因素
- 基于优先级的抢占式调度内核中，调度延迟比较小

## 第十二讲

### 1. 了解 uC/OS 中任务的状态(就绪、运行、挂起、休眠和中断)

$\mu$ C/OS-II支持最多64个任务，每个任务有一个特定的优先级，且优先级越高，其数值越小

- 运行态(运行)  
任何时刻只能有一个任务处于运行态
- 就绪态(可以运行)  
任务一旦建立，这个任务就进入就绪态，准备运行
- 挂起态(不可运行)  
正在运行的任务可能需要等待某一事件的发生或将在自己延迟一段时间
- 中断服务态(不属于多任务管理的范围)  
正在运行的任务是可以被中断的，除非该任务将中断关闭；被中断了的任务进入了中断服务态
- 休眠态(不属于多任务管理的范围)  
任务创建之前的状态，仅驻留在程序空间，还没有交给 $\mu$ C/OS-II管理



#### 【任务控制块(TCB)】

任务控制块是管理任务的数据结构：

- 任务控制块OS\_TCB保存着该任务的相关参数
- 任务堆栈指针、状态、优先级、任务表位置、任务链表指针等
- 所有的任务控制块均在μC/OS-II初始化时生成，分别存在于两条链表中：空闲链表和使用链表

## 2. 理解任务就绪表的工作机制(优先级位图算法，3种操作)，体会查表带来的性能优化

### 空闲任务列表

- 初始态：所有任务控制块都被放置在任务控制块列表数组中
- 系统初始化：所有任务控制块被链接成空任务控制块的单向链表
- 任务建立：空任务控制块指针指向的任务控制块分配给该任务，链表中指针进行后移

### 任务队列

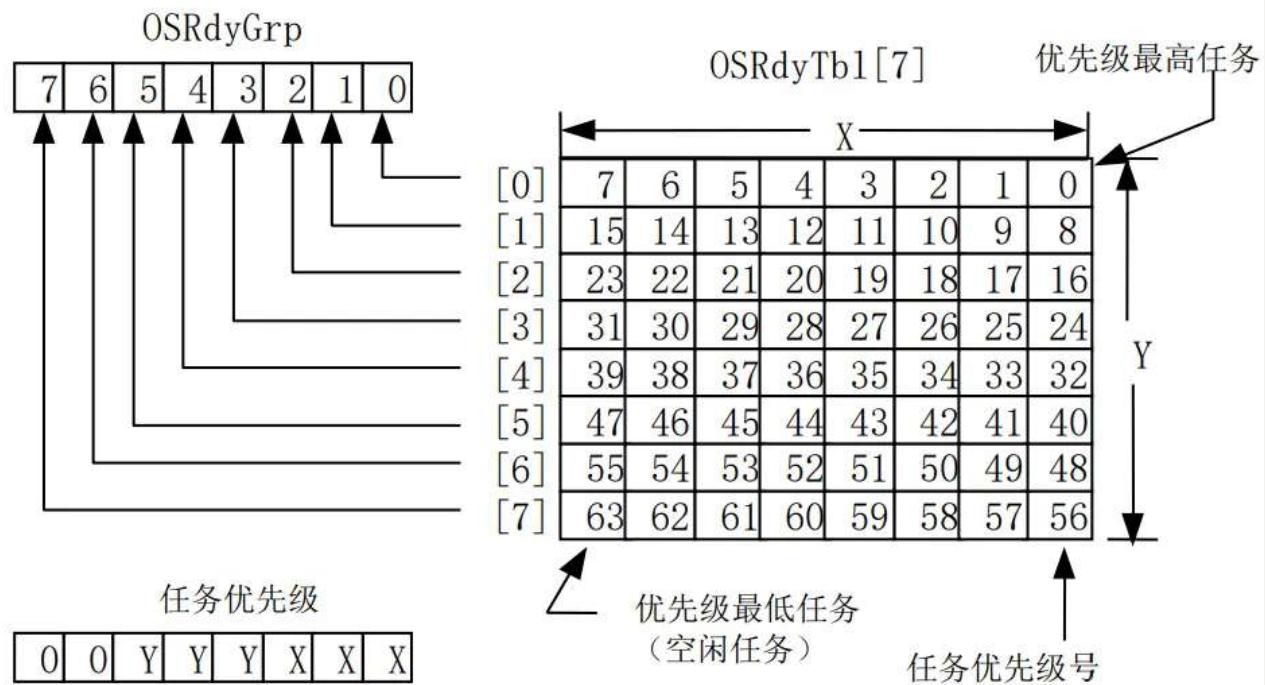
一般情况下，操作系统通过任务队列的方式进行任务管理。将任务组织为就绪队列和等待队列：

- 任务就绪时，把任务控制块放在就绪队列尾
- 任务挂起时，把任务控制块放在等待队列尾
- 从就绪任务队列选择当前运行任务

【特点】任务处理时间与任务数量密切相关

## 优先级位图算法

### 【任务就绪表】



### 【注解】

1. 优先级位图: 任务状态分为就绪态和非就绪态, 对应两个状态, 一比特位编码
  2. 任务优先级(0~ 63)划分为组号(高三位)与组内编号(低三位), 并且高优先级对应小的优先级号, 其中组号索引OSRdyTbl, 用组内编号索引OSRdyGrp
  3. OSRdyTbl类似二维数组, 每个元素对应就绪表每一行
  4. OSRdyGrp数组标记OSRdyTbl的每一行是否存在1, 即全组任务中没有一个进入就绪态时, OSRdyGrp的相应位才为零
- 掩码数组OSMapTbl[7]用于对OSRdyTbl和OSRdyGrp置位, 预存数组, 使用时只是一次取内存的操作
    - $OSMapTbl[0]=2^0=0x01(0000\ 0001)$
    - $OSMapTbl[1]=2^1=0x02(0000\ 0010)$
    - $OSMapTbl[7]=2^7=0x80(1000\ 0000)$
  - Op1: 使任务进入就绪态  
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, 或使任务进入就绪态的方法:

```
OSRdyGrp |= OSMapTbl[prio>>3]
```

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

- Op2: 使任务脱离就绪态

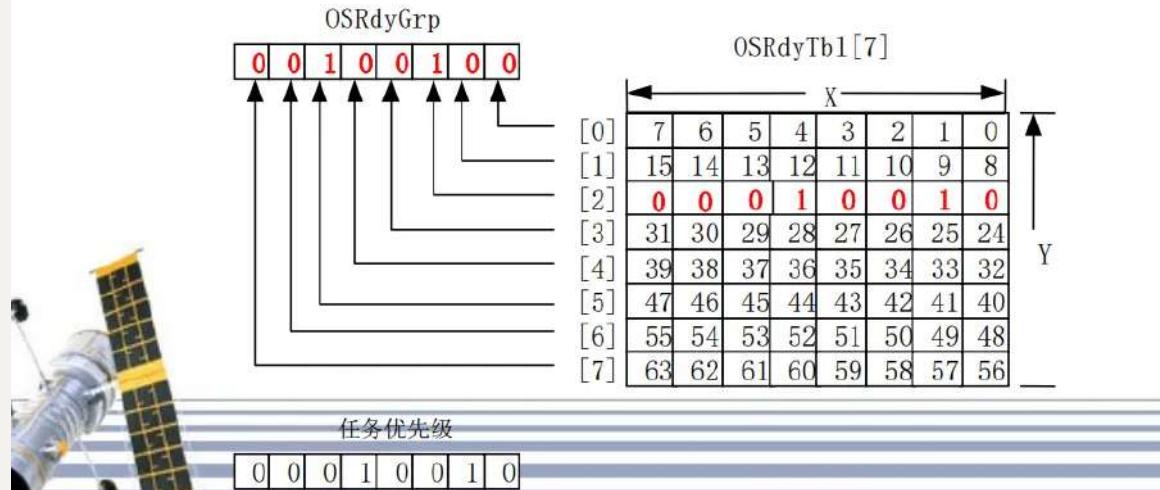
将任务就绪表OSRdyTbl[prio>>3]相应元素的相应位清零(掩码取反), 而且当OSRdyTbl[prio>>3]中的所有位都为零时, 即全组任务中没有一个进入就绪态时, OSRdyGrp的相应位才为零

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

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

- Op3: 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=10 0100b，--> 对应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=01 0010b，--> 第1个和第4个任务，取高优先级为1，则最高优先级的任务为 $2*8+1=17$



$$\min j (j = 0, 1, \dots, 7), \min i (j = 0, 1, \dots, 7)$$

$$\max prio = j * 8 + i$$

### 查表法优化【优先级判定表】

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]; 查表得到组号

Low3 = OSUnMapTbl[OSRdyTbl[High3]]; 查表得到组内编号

;注解: Low3也是获取0 ~ 7位中最低位的1，与组号同理，可以利用查表法获取得到

Prio = (High3<<3)+Low3

### 3. uC/OS 的任务级调度和中断级调度【掌握程度：重在理解】

- μC/OS-II总是选择运行进入就绪态任务中优先级最高的任务；
- 以优先级为下标，即可得相应任务控制块
- μC/OS-II任务调度的执行时间为常数，与当前建立的任务数没有关系

## 任务级调度[注重代码分析]

```
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();
}
```

检查是否处于中断或任务调度禁止

找到优先级高3位

找到最高优先级

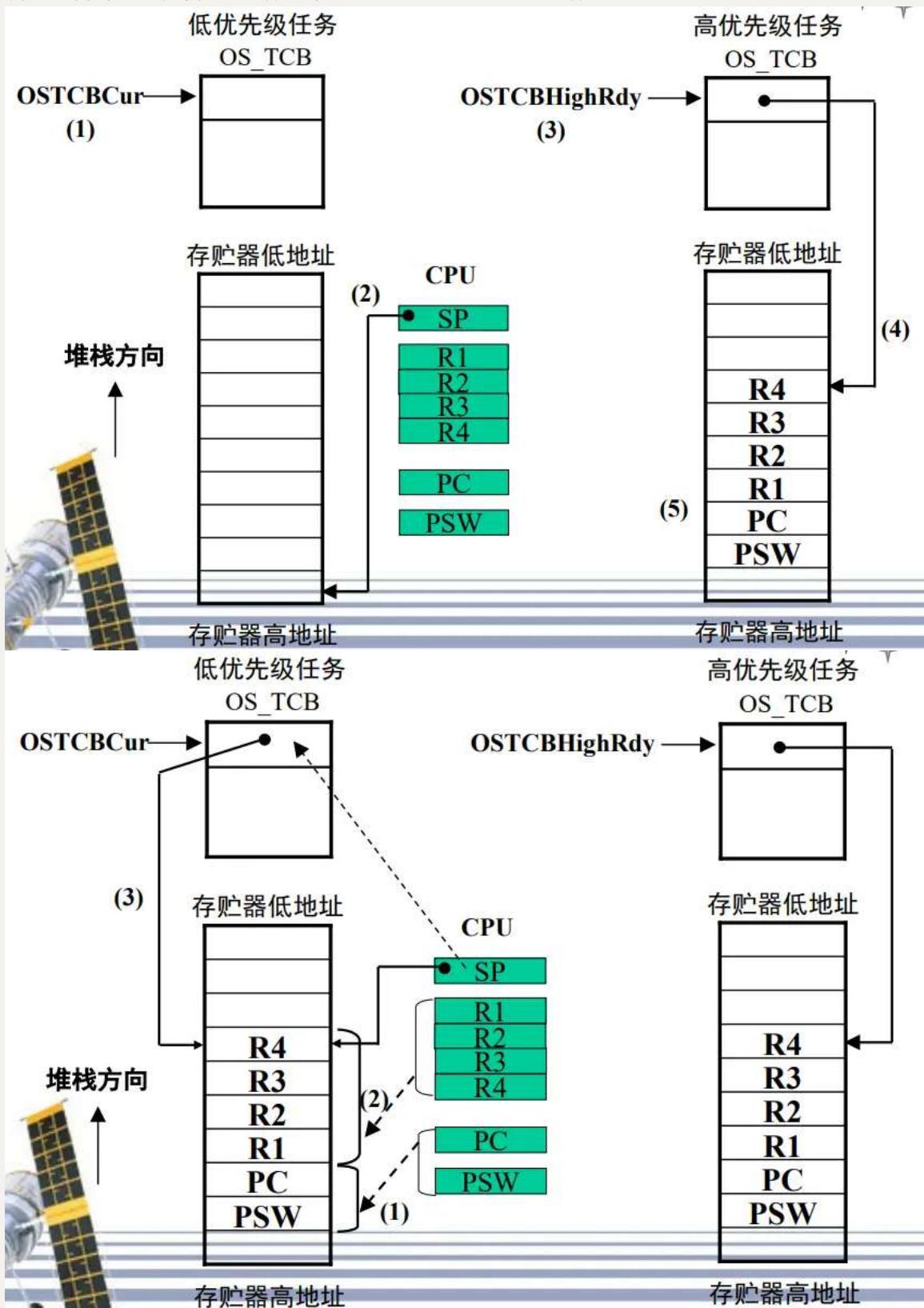
该任务是否是当前任务

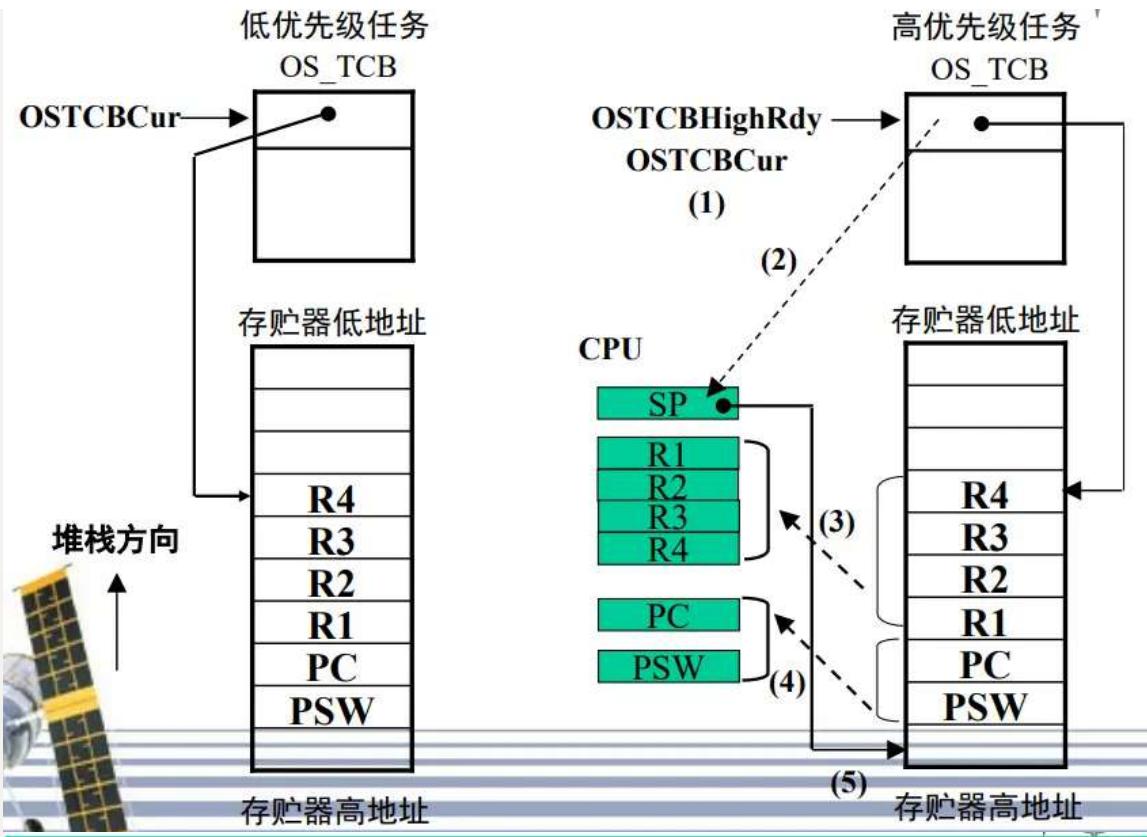
找到优先级最高的任务控制块

## 【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值出栈，回复到CPU寄存器中





```
Void OSCtxSw(void)
```

```
{
```

将R1, R2, R3及R4推入当前堆栈;

```
OSTCBCur→OSTCBSTkPtr = SP;
```

```
OSTCBCur = OSTCBHighRdy;
```

```
SP = OSTCBHighRdy →OSTCBSTkPtr;
```

将R4, R3, R2及R1从新堆栈中弹出;

执行中断返回指令;

```
}
```

### 【调度器上锁和开锁】

注解：如果需要保护一个任务不被切换，可以使用关中断，但是力度太大导致不能响应中断；μcos采用调度器上锁机制，实现了保护任务同时可以响应中断的目的。

- OSSchedlock()

该函数用于禁止任务调度直到调用给调度器开锁函数为止；中断可以识别，中断服务也能得到；OSLockNesting跟踪该函数被调用的次数；

- OSSchedUnlock()

该函数给OSLockNesting值减1，当值减为0时，执行一次调度(因为可能有更高优先级的任务就绪)

```
void OSSchedLock (void)
{
    if (OSRunning == TRUE) {
        OS_ENTER_CRITICAL();
        OSLockNesting++;
        OS_EXIT_CRITICAL();
    }
}
```

OSLockNesting是一个全局变量，因此在进行访问时需要关中断；同时OSLockNesting记录加锁的个数

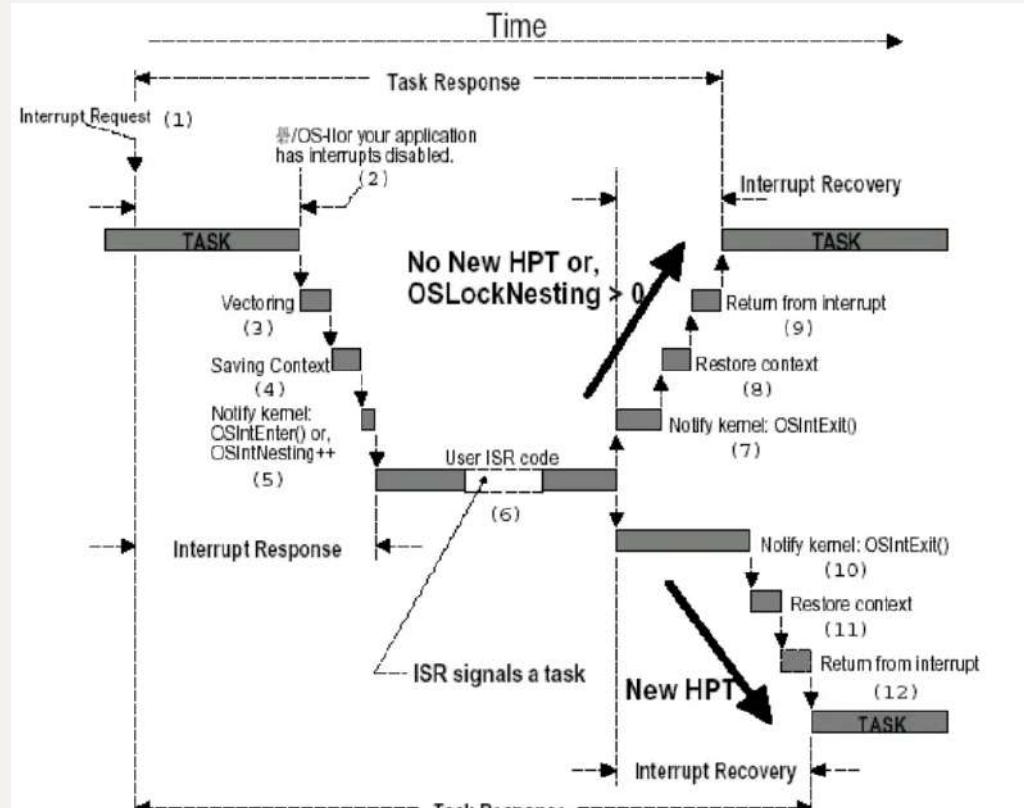
```
void OSSchedUnlock (void)
{
    if (OSRunning == TRUE) {
        OS_ENTER_CRITICAL();
        if (OSLockNesting > 0) {
            OSLockNesting--;
            if ((OSLockNesting | OSIntNesting) == 0) {
                OS_EXIT_CRITICAL();
                OSSched();
            } else {
                OS_EXIT_CRITICAL();
            }
        } else {
            OS_EXIT_CRITICAL();
        }
    }
}
```

当OSLockNesting减到0(没有处于中断)时，再次进行一次任务调度，调度之前可能需要被调度的任务

### 中断级调度【注重代码分析】

基于抢占式的中断调度，在中断返回的时候进行任务的调度，检测是否有更高优先级的任务已经就绪。

- 中断结束时调用返回函数OSIntExit(), OSIntExit()将中断嵌套计数器OSIntNesting减1;
- 当计数器减到0时，对就绪任务做判断，并返回到任务：
  - 有更高优先级任务就绪，返回更高优先级任务
  - 否则，返回到被中断任务



西安电子科技大学计算机学院

```

void OSIntEnter (void)
{
    OS_ENTER_CRITICAL();

    OSIntNesting++;

    OS_EXIT_CRITICAL();
}

void OSIntExit (void)
{
    OS_ENTER_CRITICAL();

    if ((--OSIntNesting | OSLockNesting) == 0) {
        OSIntExitY = OSUnMapTb1[OSRdyGrp];
        OSPrioHighRdy = (INT8U)((OSIntExitY << 3) +
                               OSUnMapTb1[OSRdyTb1[OSIntExitY]]);
        if (OSPrioHighRdy != OSPrioCur) {
            OSTCBHighRdy = OSTCBPrioTb1[OSPrioHighRdy];
            OSCtxSwCtr++;
            OSIntCtxSw();
        }
    }
    OS_EXIT_CRITICAL();
}

```

- OSIntNesting的数值标识中断嵌套的层数，是一个全局变量；
- 中断返回时判断是否为0，当为0时意味着中断已全部返回并且调度器没有上锁时，进行一次任务调度；
- 如果最高优先级任务不是当前任务，则进行任务切换，否则不切换；

区别：

- 任务级调度每次都需要保护现场，并恢复现场；
- 而中断级的调度，在进行中断时以保存过现场，因此在调度时只需恢复现场；
- 因此任务级调度的任务切换函数OS\_TASK\_SW()与中断级调度的任务切换函数OSIntCtxSw()不同；

*OSIntNesting* 与 *OSLockNesting* 两种嵌套方式相互独立，互不影响，保证了任务不切换但是可以响应中断

#### 4. 时钟节拍[掌握程度：重在理解]

- 时钟节拍(时钟滴答)Tick，是一种定时器中断，可通过编程方式实现
- 时钟节拍是一种特殊的中断，是操作系统的心脏
  - 中断服务子程序中，首先对32位的整数OSTime加一(OSTime记录系统启动以来的时钟滴答数)
  - 对任务列表进行扫描，判断是否有延时任务已经处于就绪状态，最后进行上下文切换

```
void OSTickISR(void)
{
    保存处理器寄存器的值;
    调用OSIntEnter(),或是将OSIntNesting加1
    if(OSIntNesting==1){
        OSTCBCur->OSTCBStkPtr=SP;
    }
    调用OSTimeTick(); 真正的用户中断服务程序
    清发出中断设备的中断;
    重新允许中断（可选用）
    调用OSIntExit();
    恢复处理器寄存器的值;
    执行中断返回指令;
}

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)) { 任务是否处于挂起状态，若不是
                    OSRdyGrp |= ptcb->OSTCBBity; 将其列入就绪表
                    OSRdyTb1[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
}
```

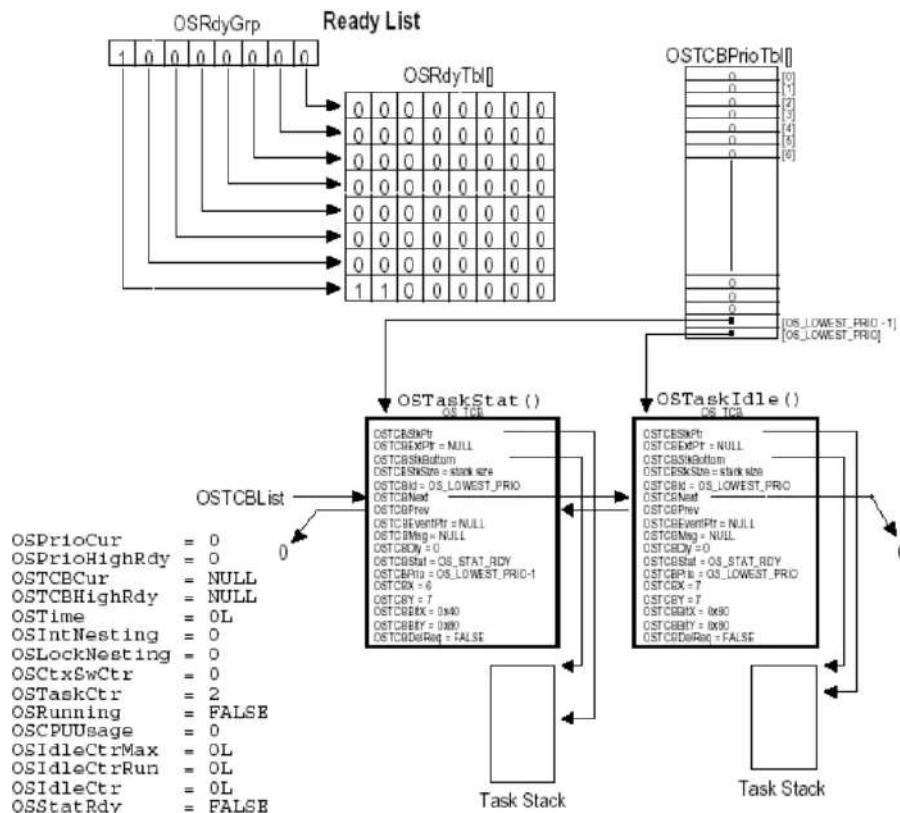
**【注解】**时钟中断做为一种特殊的中断，需要保存上下文环境，然后进入到真正的用户中断服务程序，最后中断返回，回复上下文环境

## 空闲任务

- μC/OS-II总要建立一个空闲任务（idle task）
    - 不停的给一个32位计数器加1。
  - 在没有其它任务进入就绪态时，系统运行空闲任务。
  - 空闲任务永远设置为最低优先级。
  - 空闲任务不能被应用任务删除。

## 统计任务

- μC/OS-II有一个统计运行时间的任务，叫做OSTaskStat()。
  - 该任务运行1次/秒，利用空闲任务的计数器值，计算当前的CPU利用率。



## 5. 怎样理解可以响应中断但不能进行任务切换

### 第十三讲 (要求能编程实现多任务应用)

任务管理用来实现对任务状态的直接控制和访问

内核的任务管理是通过系统调用(等待、延时等)来体现的[不然会阻塞CPU, 保证能够分时进行]

#### 1. 任务的创建(4个参数的作用)

##### 【任务创建时机】

- 多任务调度OSStart()开始前 【初始化创建任务】
- 其他任务执行过程中 【任务执行时又创建或删除任务】
- 任务不能由中断服务程序建立 【中断服务程序不属于多任务管理的范畴，不能参与任何的任务管理工作：不能在中断中创建任务和删除任务，检测OSIntNesting是否为0，大于0表示有中断或中断嵌套】

##### 【参数作用】

- task: 指向任务代码的指针
- pdata: 任务开始时，传递给任务的指针参数
- ptos: 分配给任务的堆栈栈顶指针
- prio: 分配给任务的优先级

## 【任务创建代码分析】

```
INT8U OSTaskCreate (void (*task)(void *pd), void *pdata, OS_STK *ptos, INT8U prio)
{
    void *psp;
    INT8U err;

    if (prio > OS_LOWEST_PRIO) {
        return (OS_PRIO_INVALID);
    }

    OS_ENTER_CRITICAL();
    if (OSTCBPrioTbl[prio] == (OS_TCB *)0) {
        OSTCBPrioTbl[prio] = (OS_TCB *)1;           // 优先级有没有被其他任务占用
        OS_EXIT_CRITICAL();

        psp = (void *)OSTaskStkInit(task, pdata, ptos, 0); // 初始化任务堆栈
        err = OSTCBInit(prio, psp, (void *)0, 0, 0, (void *)0, 0); // 初始化任务控制块
        if (err == OS_NO_ERR) {
            OS_ENTER_CRITICAL();                     // 给任务计数器加1
            OTaskCtr++;
            OS_EXIT_CRITICAL();
        }
    }
}

注：任务堆栈初始化时就将程序入口地址放到了堆栈中
```

```
if (OSRunning) {                                // 如果系统已经开始运行（在某任务过程中建立），则进行任务调度
    OSSched();
}
} else {                                         // 如果创建失败（例如没有可用的任务控制块），将优先级释放
    OS_ENTER_CRITICAL();
    OSTCBPrioTbl[prio] = (OS_TCB *)0;
    OS_EXIT_CRITICAL();
}
return (err);
} else {                                         // 如果优先级已经被占用，返回错误码
    OS_EXIT_CRITICAL();
    return (OS_PRIO_EXIST);
}
}
```

【注解】首先进行优先级占位而非直接申请任务控制块，主要是因为提前判断是否有其他任务占用此优先级，速度比初始化任务控制块更快。

```
void main()
{
    OSInit();
    OSTaskCreate( Task1, (void *)&Task1Data,
                  (void *)&Task1Stk[TASK_STK_SIZE], Task1prio);
    OSTaskCreate( Task2, (void *)&Task2Data,
                  (void *)&Task2Stk[TASK_STK_SIZE], Task2prio);
    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]];                      // 找到优先级最高的就绪任务
        OSPrioHighRdy = (INT8U)((y << 3) + x);
        OSPrioCur = OSPrioHighRdy;                         // 设为当前运行任务优先级
        OSTCBHighRdy = OSTCBPrioTbl[OSPrioHighRdy];
        OSTCBCur = OSTCBHighRdy;                           // 根据任务优先级找到任务控制块，并设置为当前任务控制块
        OSStartHighRdy();                                  // 让优先级最高的任务运行起来
    }
}
```

### 【任务堆栈】

每个任务都有自己的堆栈空间，并由连续的内存空间组成；

- 堆栈的分配

## 静态分配

保证位于程序的RW段

- static OS\_STK MyTaskStack[stack\_size]
- OS\_STK MyTaskStack[stack\_size]

局部静态变量

全局变量

## 动态分配

位于程序的堆中, 但一直不释放

```
□ OS_STK *pstk;  
pstk = (OS_STK*)malloc(stack_size);  
if(pstk != (OS_STK*)0){  
    create the task;  
}
```

注: 绝对不能位于程序的栈中

【注解】malloc在程序堆(不同于堆栈)中分配内存,有可能会产生内存碎片, 导致分配失败。

## 【任务删除】

- 使任务返回, 并处于休眠状态
- 任务代码并不被删除, 只是不再被调用

```

INT8U OSTaskDel (INT8U prio)
{
    OS_TCB *ptcb;
    OS_EVENT *pevent;

    if (prio == OS_IDLE_PRIO) {
        return (OS_TASK_DEL_IDLE);
    }
    if (prio >= OS_LOWEST_PRIO && prio != OS_PRIO_SELF) {
        return (OS_PRIO_INVALID);
    }
    OS_ENTER_CRITICAL();
    if (OSIntNesting > 0) {                                不在中断中删除任务
        OS_EXIT_CRITICAL();
        return (OS_TASK_DEL_ISR);
    }
    if (prio == OS_PRIO_SELF) {                            可以删除自己
        Prio = OSTCBCur->OSTCBPrio;
    }

    if ((ptcb = OSTCBPrioTbl[prio]) != (OS_TCB *)0) {      得到该任务控制块
        if ((OSRdyTbl[ptcb->OSTCBY] &= ~ptcb->OSTCBBitX) == 0) {
            OSRdyGrp &= ~ptcb->OSTCBBitY;                  将该任务从就绪表中删除
        }
        if ((pevent = ptcb->OSTCBEventPtr) != (OS_EVENT *)0) {
            if ((pevent->OSEventTbl[ptcb->OSTCBY] &= ~ptcb->OSTCBBitX) == 0) {
                pevent->OSEventGrp &= ~ptcb->OSTCBBitY;      处理与该任务有关的事件
            }
        }
        Ptcb->OSTCBDly = 0;                                将时钟节拍延迟数清零
        Ptcb->OSTCBStat = OS_STAT_RDY;                    将任务状态置为RDY
        OSLockNesting++;                                    防止调度器进行任务切换
        OS_EXIT_CRITICAL();                                开一次中断，缩短中断响应时间
        OSDummy();
        OS_ENTER_CRITICAL();                             关中断，重新进入临界区
        OSLockNesting--;                                调度器开锁
        OSTaskDelHook(ptcb);                            删删除钩子函数
        OSTaskCtr--;
        OSTCBPrioTbl[prio] = (OS_TCB *)0;              控制块数组指针清零
    }
}

```

```
if (ptcb->OSTCBPrev == (OS_TCB *)0) {           —————— 删除双向链表OSTCBList中  
    ptcb->OSTCBNext->OSTCBPrev = (OS_TCB *)0; 的任务控制块  
    OSTCBLList      = ptcb->OSTCBNext;  
}  
else {  
    ptcb->OSTCBPrev->OSTCBNext = ptcb->OSTCBNext;  
    ptcb->OSTCBNext->OSTCBPrev = ptcb->OSTCBPrev;  
}  
ptcb->OSTCBNext = OSTCBFreeList;          —————— 将任务控制块归还给  
OSTCBFreeList = ptcb;                      OSTCBFreeList链表  
OS_EXIT_CRITICAL();  
OSSched();                                —————— 进行一次任务调度  
return (OS_NO_ERR);  
}  
else {  
    OS_EXIT_CRITICAL();  
    return (OS_TASK_DEL_ERR);  
}  
}  
}
```

【注解】注重Dummy()函数的使用，以及调度器上锁，开关中断的处理，其目的是将任务删除分为两个阶段，两个阶段的间隔能够响应中断但是不能切换任务。

## 2. 任务的挂起与恢复机制

- 任务可以挂起自己或其他任务

- 被OSTaskSuspend()函数挂起的任务只能通过调用OSTaskResume()函数来恢复

```

INT8U OSTaskSuspend (INT8U prio)
{
    BOOLEAN self;
    OS_TCB *ptcb;

    if (prio == OS_IDLE_PRIO) {
        return (OS_TASK_SUSPEND_IDLE);
    }
    if (prio >= OS_LOWEST_PRIO && prio != OS_PRIO_SELF) {
        return (OS_PRIO_INVALID);
    }
    OS_ENTER_CRITICAL();
    if (prio == OS_PRIO_SELF) {
        prio = OSTCBCur->OSTCBPrio;
        self = TRUE;
    } else if (prio == OSTCBCur->OSTCBPrio) {
        self = TRUE;
    } else {

        self = FALSE;
    }
    if ((ptcb = OSTCBPrioTbl[prio]) == (OS_TCB *)0) {
        OS_EXIT_CRITICAL();
        return (OS_TASK_SUSPEND_PRIO);
    } else {
        if ((OSRdyTbl[ptcb->OSTCBY] &= ~ptcb->OSTCBBitX) == 0) {
            OSRdyGrp &= ~ptcb->OSTCBBitY;
        }
        ptcb->OSTCBStat |= OS_STAT_SUSPEND;
        OS_EXIT_CRITICAL();
        if (self == TRUE) {
            OSSched();
        }
        return (OS_NO_ERR);
    }
}

```

不能挂起空闲任务

判断要挂起的任务是否是自己

检查任务是否存在

存在，从就绪表中删除

将任务状态置为挂起OS\_STAT\_SUSPEND

如果挂起当前任务，进行任务调度

### INT8U OSTaskResume (INT8U prio)

```
{  
    OS_TCB *ptcb;  
  
    If (prio >= OS_LOWEST_PRIO) {  
        return (OS_PRIO_INVALID);  
    }  
    OS_ENTER_CRITICAL();  
    If ((ptcb = OSTCBPrioTbl[prio]) == (OS_TCB *)0) {  
        // 任务是否存在  
        OS_EXIT_CRITICAL();  
        return (OS_TASK_RESUME_PRIO);  
    } else {  
        if (ptcb->OSTCBStat & OS_STAT_SUSPEND) {  
            // 任务必须是被挂起的  
            if (((ptcb->OSTCBStat &= ~OS_STAT_SUSPEND) == OS_STAT_RDY) &&  
                (ptcb->OSTCBDly == 0)) {  
                // 清除OS_STAT_SUSPEND标志  
                OSRdyGrp |= ptcb->OSTCBBitY;  
                OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;  
                // 若没有时钟等待，则让任务就绪  
                OS_EXIT_CRITICAL();  
                OSSched();  
                // 有新任务就绪，进行任务调度  
            } else {  
                OS_EXIT_CRITICAL();  
            }  
            return (OS_NO_ERR);  
        } else {  
            OS_EXIT_CRITICAL();  
            return (OS_TASK_NOT_SUSPENDED);  
        }  
    }  
}
```

注：不需要判断是否是自己，被挂起的任务不可能再执行该函数

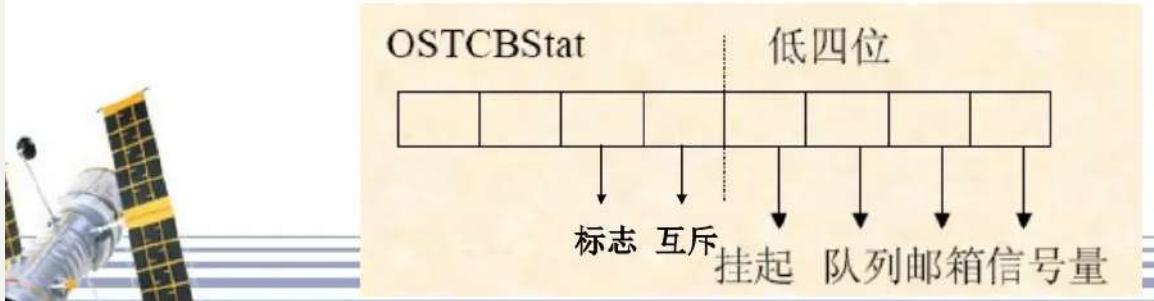
【只能清除对应的位，不能清除其他位】

【任务的状态】

```

/*
 * TASK STATUS (Bit definition for OSTCBStat)
 */
#define OS_STAT_RDY      0x00 /* Ready to run */          */
#define OS_STAT_SEM      0x01 /* Pending on semaphore */    */
#define OS_STAT_MBOX     0x02 /* Pending on mailbox */     */
#define OS_STAT_Q        0x04 /* Pending on queue */      */
#define OS_STAT_SUSPEND  0x08 /* Task is suspended */     */
#define OS_STAT_MUTEX    0x10 /* Pending on mutual exclusion semaphore */ */
#define OS_STAT_FLAG     0x20 /* Pending on event flag group */ */

```



### 3. 任务延迟 OSTimeDly 的实现机制(利用时钟中断)

- 时钟中断

定时器的中断由硬件完成，通过OSTimeTick()实现

```

void OSTimeDly (INT16U ticks)
{
    if (ticks > 0) { —————— 一个非0值会使当前任务从就绪表中删除
        OS_ENTER_CRITICAL();
        if ((OSRdyTbl[OSTCBCur->OSTCBY] &= ~OSTCBCur-
>OSTCBBitX) == 0) {
            OSRdyGrp &= ~OSTCBCur->OSTCBBitY;
        }
        OSTCBCur->OSTCBDly = ticks; —————— 延迟时钟节拍数被保存在任
        OS_EXIT_CRITICAL();务控制块中每隔1个时钟节拍,
        OSSched(); —————— 该数被OSTimeTick()减1
    }
}

```

- 把当前任务从就绪表中删除

- 将ticks赋值给任务控制块的Delay属性

```

void OSTimeTick (void)
{
    OS_TCB *ptcb;
    ptcb = OSTCBLList;          ━━━━━━ OSTCB链表指针, 遍历链表
    while (ptcb->OSTCBPrio != OS_IDLE_PRIO) {
        if (ptcb->OSTCBDly != 0) { ━━━━━━ 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->OSTCBStat & OS_STAT_SUSPEND)) { ━━━━━━ 延时减一, 并判断是否延时结束
                    OSRdyGrp           |= ptcb->OSTCBBitY; ━━━━━━ 若不是, 将其列入就绪表
                    OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
                } else {ptcb->OSTCBDly = 1; }
            }
        }
        ptcb = ptcb->OSTCBNext; ━━━━━━ 指针指向下一个TCB结构
    }
    OSTime++; ━━━━━━ 遍历完成后, 时钟计数器加一, 以记录系统启动以来的时钟滴答数
}

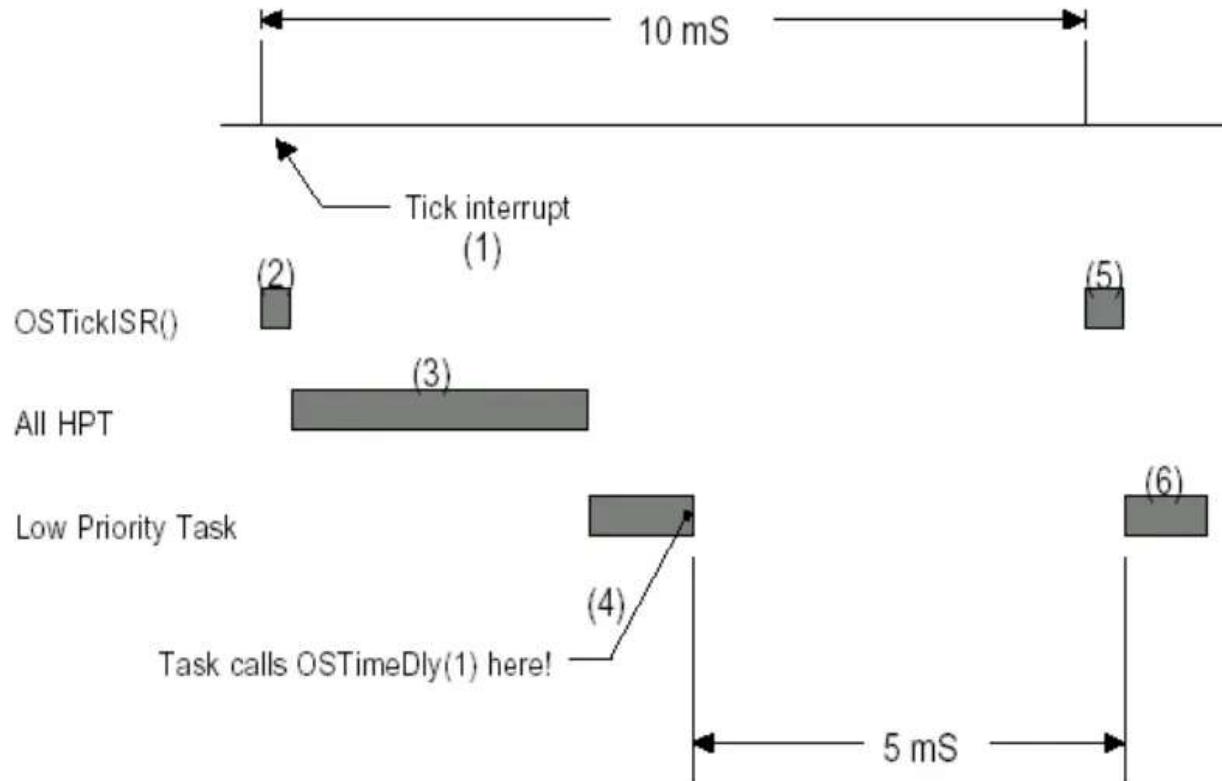
```

## 【改变任务状态的两套机制】

- 任务的挂起与恢复机制
- 任务延迟(自动恢复)

时钟节拍的精度：10ms一次触发时钟中断，10ms做为操作系统的精度；但是精度不能无限制地提高；

# 时钟节拍延迟过程



【`OSTimeDlyResume()`】

- 被延时的任务可以不等待延时期满，而通过OSTimeDlyResume()函数来恢复

```

INT8U OSTimeDlyResume (INT8U prio)
{
    OS_TCB *ptcb;

    if (prio >= OS_LOWEST_PRIO) {
        return (OS_PRIO_INVALID);
    }

    OS_ENTER_CRITICAL();
    ptcb = (OS_TCB *)OSTCBPrioTbl[prio];
    if (ptcb != (OS_TCB *)0) {
        if (ptcb->OSTCBDly != 0) {
            ptcb->OSTCBDly = 0;           └─> 将等待延迟数清零
        }
        if (!(ptcb->OSTCBStat & OS_STAT_SUSPEND)) {
            OSRdyGrp      |= ptcb->OSTCBBitY; └─> 任务没有被挂起就可以就绪
            OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
            OS_EXIT_CRITICAL();
            OSSched();                  └─> 有任务就绪，进行一次调度
        }
    } else {
        OS_EXIT_CRITICAL();
    }
    return (OS_NO_ERR);
} else {
    OS_EXIT_CRITICAL();
    return (OS_TIME_NOT_DLY);
}
} else {
    OS_EXIT_CRITICAL();
    return (OS_TASK_NOT_EXIST);
}
}

```

## 第十四讲 (要求能编程应用信号量)

### 0. 任务通信的基本知识

【信号量】 [关键确定同步和互斥关系]

- 信号量大于等于零的时候代表可供并发进程使用的资源实体数
- 信号量小于零的时候，表示正在等待使用临界区的进程的个数
- p操作和v操作是不可中断的程序段，称为原语(具有原子性)



## 信号量的P、V操作



### ➤ P (S) :

- ①将信号量S的值减1，即 $S=S-1$ ；
- ②如果 $S>0$ ，则该进程继续执行；否则该进程置为等待状态，排入等待队列。

### ➤ V (S) :

- ①将信号量S的值加1，即 $S=S+1$ ；
- ②如果 $S>0$ ，则该进程继续执行；否则释放队列中第一个等待信号量的进程。

### ➤ PV操作的意义:

- 我们用信号量及PV操作来实现进程的同步和互斥。PV操作属于进程（线程）的低级通信。



## 1. uC/OS 中事件的概念(事件等待队列的实现机制和任务就绪表一致)

- μC/OS-II中的任务间通信：通过信号量、消息邮箱和消息队列来实现实任务间通信的

**【事件】** μC/OS-II将信号量、邮箱和队列统称为事件，在事件控制块中通过OSEventType区分

**【事件与任务的区别】** 事件可以改变任务的状态，任务等待某个事件，基于事件驱动的多任务管理，有限状态机中事件的发生与状态的转换

**【事件控制块】** 各种事件控制函数实现的基本数据结构

# 事件控制块

```
typedef struct {  
    void *OSEventPtr;           /* 指向消息或者消息队列的指针 */  
    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 */  
    INT8U OSEventGrp;           /* 等待任务所在的组 */  
} OS_EVENT;
```

#define OS_EVENT_TYPE_UNUSED	0
#define OS_EVENT_TYPE_MBOX	1
#define OS_EVENT_TYPE_Q	2
#define OS_EVENT_TYPE_SEM	3
#define OS_EVENT_TYPE_MUTEX	4
#define OS_EVENT_TYPE_FLAG	5



OSEventGrp和OSEventTbl[]与OSRdyGrp和OSRdyTbl[]相似，只不过前两者包括的是等待某事件的任务，后两者包括的是系统中处于就绪状态的任务。

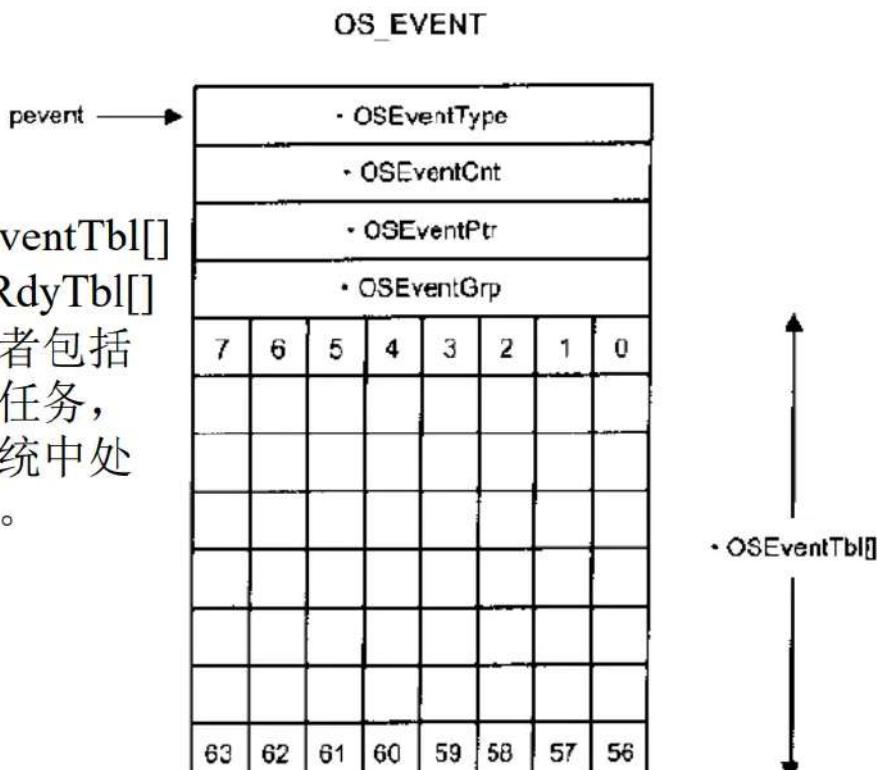
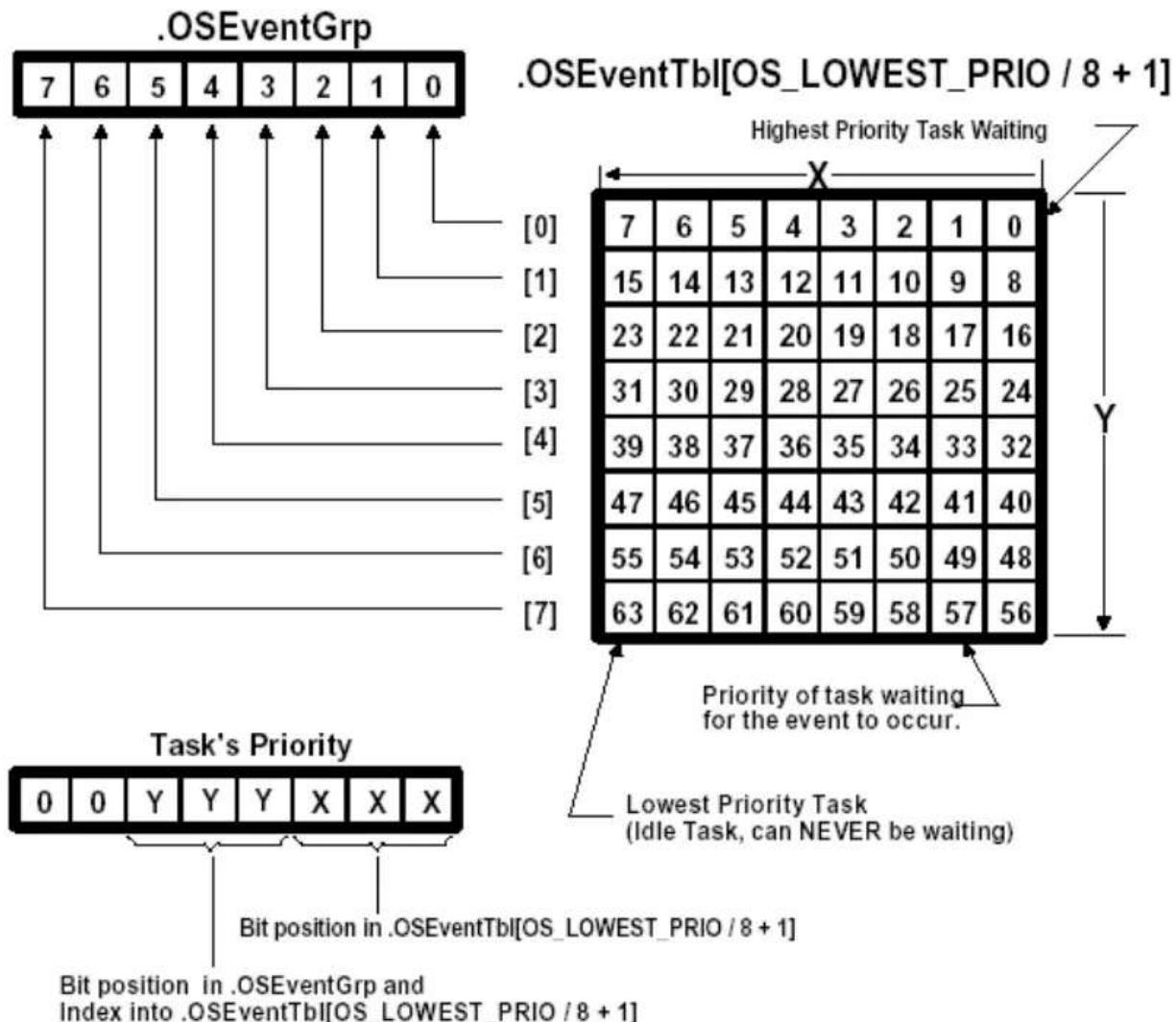


图 F6.2 事件控制块(ECB)

事件控制块是μC/OS-II任务管理的实现机制，统一了信号量，邮箱和队列三种方式

## 【等待任务列表实现机制】



**Figure 6-2, Wait list for task waiting for an event to occur.**

## 【对等待事件任务列表的三种操作】

- 将任务置于等待事件任务列表(加入等待事件)
 

```
pevent->OSEventGrp |= OSMapTbl[prio >> 3]; 对组号置位
pevent->OSEventTbl[prio >> 3] |= OSMapTbl[prio & 0x07] 取出prio的低三位利用掩码数组OSMapTbl对组内编号置位
```

*pevent*是当前的任务控制块

- 从等待事件任务列表中使任务脱离等待状态(释放等待时间)
 

```
if ((pevent->OSEventTbl[prio >> 3] &= ~OSMapTbl[prio & 0x07]) == 0) {
pevent->OSEventGrp &= ~OSMapTbl[prio >> 3];
}

~OSMapTbl[prio & 0x07]对应任务的优先级利用掩码取反使得组内编号清零, 判断如果该组全为0,则对对应组号清零
```

- 在等待事件任务列表中查找优先级最高的任务(寻找最高优先级)

```
y = OSUnMapTbl[pEvent->OSEventGrp];
x = OSUnMapTbl[pEvent->OSEventTbl[y]];
prio = (y << 3) + x;
```

# 空余事件控制块

- 事件控制块的总数是由应用程序所需的信号量、互斥型信号量、消息队列和消息邮箱的总数所决定的。空余事件控制块链表由系统初始化函数建立。

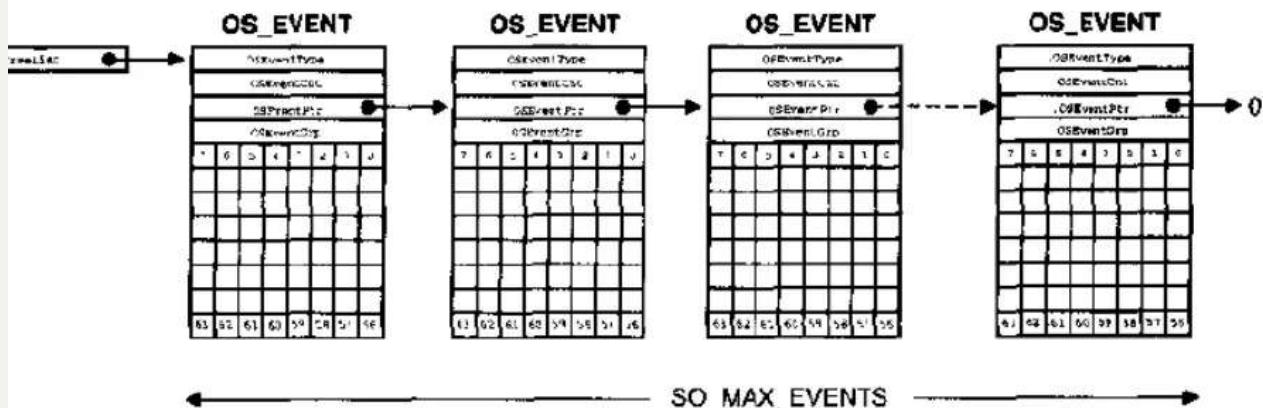


图 F6.5 空余事件控制块链表

单个事件控制块: 创建单个事件, 对应有对多个任务在等待该事件发生, 即等待事件任务列表  
事件控制块链表: 创建有多个事件, 依次串联起来

空闲事件控制块链表: ucos采用静态编译方式, 提前分配好并初始化事件控制块

【对事件控制块的操作】

- 初始化事件控制块: 只初始化等待事件任务列表

```
void OSEventWaitListInit (OS_EVENT *pevent)
```

```
{
```

```
INT8U i;
```

```
pevent->OSEventGrp = 0x00;
```

```
for (i = 0; i < OS_EVENT_TBL_SIZE; i++) {
```

```
    pevent->OSEventTbl[i] = 0x00;
```

```
}
```

- 使当前任务进入等待某事件的状态(P操作)

```
void OSEventTaskWait (OS_EVENT *pevent)
```

```
{
```

```
OSTCBCur->OSTCBEventPtr = pevent;
```

```
if ((OSRdyTbl[OSTCBCur->OSTCBY] &= ~OSTCBCur-
```

```
>OSTCBBitX) == 0) {
```

将当前任务从就绪表中删除

```
    OSRdyGrp &= ~OSTCBCur->OSTCBBitY;
```

```
}
```

给事件控制块中的等待事件任务列表相应位置位

```
    pevent->OSEventTbl[OSTCBCur->OSTCBY] |= OSTCBCur-
```

```
>OSTCBBitX;
```

```
    pevent->OSEventGrp
```

|= OSTCBCur->OSTCBBitY;

```
}
```

当前任务不就绪,并将该任务放置等待任务列表中

操作系统实现多个等待队列, 每个事件维护一个队列

任务不是出于就绪队列(任务就绪表)中,就是在等待队列(事件等待任务列表)中

- 使一个任务就绪（脱离等待状态）(V操作)

```
void OSEventTaskRdy (OS_EVENT *pevent, void *msg, INT8U msk)
```

```
{
```

```
    OS_TCB *ptcb;
```

```
    INT8U x;
```

```
    INT8U y;
```

```
    INT8U bitx;
```

```
    INT8U bity;
```

```
    INT8U prio;
```

```
    y = OSUnMapTbl[pevent->OSEventGrp];
```

```
    bity = OSMapTbl[y];
```

```
    x = OSUnMapTbl[pevent->OSEventTbl[y]]; 
```

```
    bitx = OSMapTbl[x];
```

确定可以进入就绪态的任务优先级

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

```
if ((pevent->OSEventTbl[y] &= ~bitx) == 0) {
```

```
    pevent->OSEventGrp &= ~bity; 从等待事件任务列表中删除该任务  
}
```

```
    ptcb = OSTCBPrioTbl[prio];
```

```
    ptcb->OSTCBDly = 0; 修改将就绪任务控制块中的一些参数
```

```
    ptcb->OSTCBEVENTPTR = (OS_EVENT *)0;
```

```
#if (OS_Q_EN && (OS_MAX_QS >= 2)) || OS_MBOX_EN
```

```
    ptcb->OSTCBMsg = msg;
```

```
#else
```

```
    msg = msg; 邮箱或队列还需将相应的消息传给任务
```

```
#endif
```

```
    ptcb->OSTCBStat &= ~msk; 位屏蔽码将OSTCBStat相应状态位清0
```

```
    if (ptcb->OSTCBStat == OS_STAT_RDY) {
```

```
        OSRdyGrp |= bity; 如果OSTCBStat为就绪态，将该任务插入到就绪任务列表中
```

```
}
```

```
}
```

注：最高优先级任务得到该事件后不一定进入就绪态，任务可能还由于其它原因而挂起

OSTCBStat任务状态可以等待多个事件,等信号量,邮箱,队列等,这几种机制互不影响

- 因为等待超时而使任务进入就绪态(不要求)

## 2. uC/OS 中信号量的实现机制(创建、P 操作和 V 操作)

【创建信号量】

➤ **OS\_EVENT OSSemCreate(INT16 cnt)**

➤ 初始值cnt为0~65535的一个整数值。

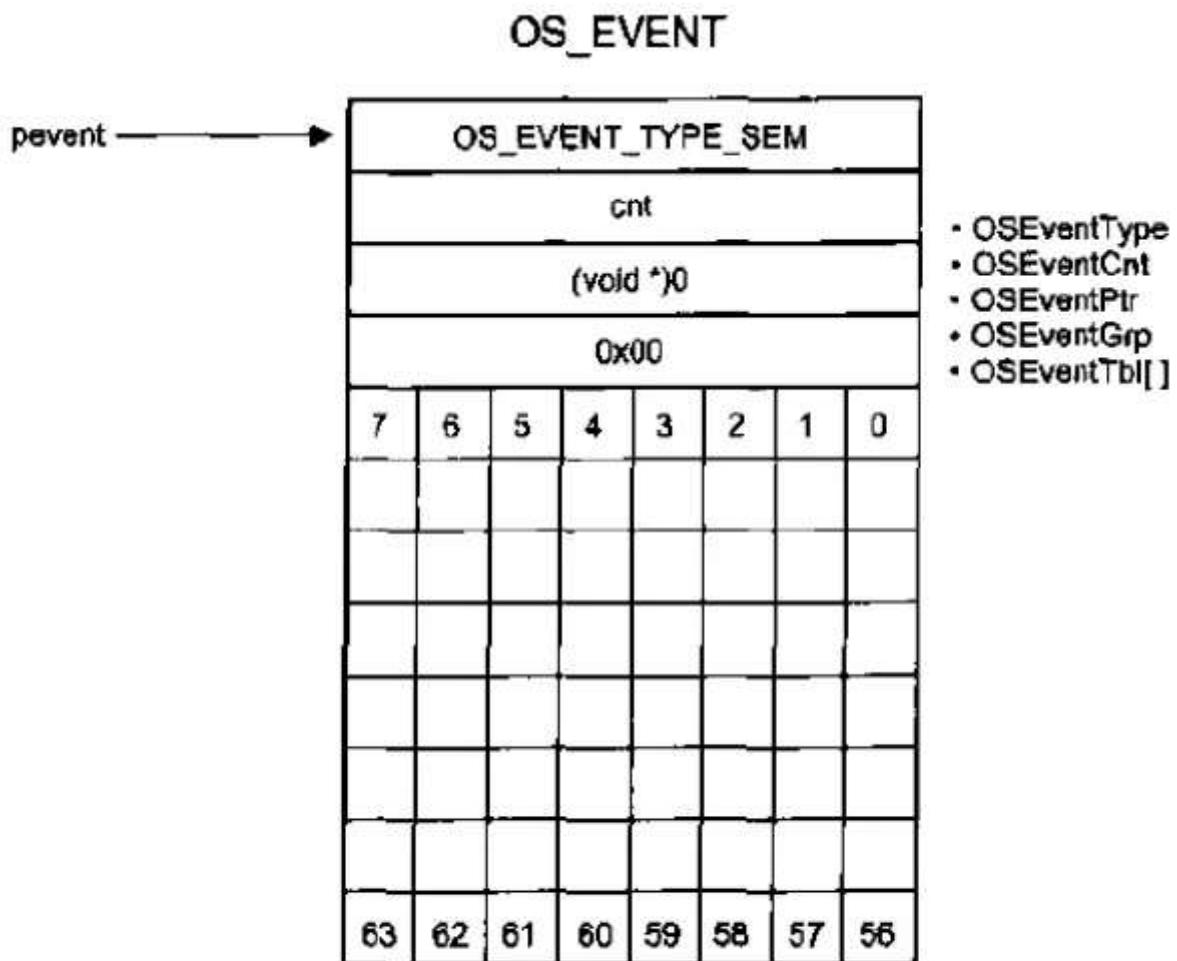
- 如果信号量表示事件的发生，应初始化为0；
- 如果信号量表示共享的资源，应初始化为1；
- 如果信号量表示n个共享资源，则初始化为n。

**OS\_EVENT \*OSSemCreate (INT16U cnt)**

{

    OS\_EVENT \*pevent;

```
    OS_ENTER_CRITICAL();
    pevent = OSEventFreeList; —————— 从空余事件控制块链表中获得一个事件控制块
    if (OSEventFreeList != (OS_EVENT *)0) {
        OSEventFreeList = (OS_EVENT *)OSEventFreeList->OSEventPtr;
    }
    OS_EXIT_CRITICAL();
    if (pevent != (OS_EVENT *)0) {
        pevent->OSEventType = OS_EVENT_TYPE_SEM; —————— 将事件类型设置为信号量
        pevent->OSEventCnt = cnt; —————— 存入信号量初始值
        OSEventWaitListInit(pevent); —————— 调用OS_EventWaitListInit()函数初始化等待任务列表
    }
    return (pevent);
}
```



注：表中所有值均初始化为0。

**图 F7.2 OSSemCreate()函数返回之前的事件控制块 ECB**

【P操作，等待一个信号量】

| *P*操作不能在中断服务程序中进行

- **Void OSSemPend(OS\_EVENT \*pevent, INT16U timeout, INT8U \*err)**
- 当信号量大于0时（有可用资源），给信号量计数值减1并返回；
- 否则（无可用资源），将申请该信号量的任务挂起，进入该事件等待任务列表，并进行任务调度；
- 直至任务被重新唤醒；
  - 被OSSemPost()函数唤醒；
  - 超时唤醒。

```
void OSSemPend (OS_EVENT *pevent, INT16U timeout, INT8U *err)
```

```
{
```

```
  OS_ENTER_CRITICAL();
```

事件类型检查

```
  if (pevent->OSEventType != OS_EVENT_TYPE_SEM) {
```

```
    OS_EXIT_CRITICAL();
```

```
    *err = OS_ERR_EVENT_TYPE;
```

```
}
```

```
  if (pevent->OSEventCnt > 0) {
```

```
    pevent->OSEventCnt--;
```

若信号量有效，则信号量计数值递减，并返回。

```
    OS_EXIT_CRITICAL();
```

```
    *err = OS_NO_ERR;
```

若在中断服务程序中，则直接返回。

```
  } else if (OSIntNesting > 0) {
```

```
    OS_EXIT_CRITICAL();
```

该函数不能在中断服务程序中调用。

```
    *err = OS_ERR_PEND_ISR;
```

```
  } else {
```

否则，信号量无效；任务进入睡眠状态，以等待另一个任务（或中断）发出该信号量。OSSemPend()函数允许定义一个最长等待时间。

```
OSTCBCur->OSTCBStat |= OS_STAT_SEM;
OSTCBCur->OSTCBDly = timeout;
OSEventTaskWait(pevent);
OS_EXIT_CRITICAL();
OSSched();
OS_ENTER_CRITICAL();

if (OSTCBCur->OSTCBStat & OS_STAT_SEM) {
    OSEventTO(pevent);
    OS_EXIT_CRITICAL();
    *err = OS_TIMEOUT;
} else {
    OSTCBCur->OSTCBEEventPtr = (OS_EVENT *)0;
    OS_EXIT_CRITICAL();
    *err = OS_NO_ERR;
}
}
```

将任务控制块中状态位的  
OS\_STAT\_SEM位置位

设置任务控制块中超时时间

使任务进入等待该事件状态

进行任务调度

当任务被再次唤醒时，判断任务控制块中的状态位；  
若状态位中OS\_STAT\_SEM仍置位，则任务是被超时  
唤醒，返回“超时”错误代码；若状态位没有置  
OS\_STAT\_SEM位（post函数会清除），则相应信号  
量已经得到，无出错返回继续执行。

【V操作, 发出一个信号量】

- **INT8U OSSemPost(OS\_EVENT \*pevent)**
- 如果有任务在等待该信号量, 从信号量的事件等待任务列表中选择最高优先级任务就绪;
- 否则给信号量计数器加1。

```
INT8U OSSemPost (OS_EVENT *pevent)
```

```
{
```

```
    OS_ENTER_CRITICAL();
```

```
    if (pevent->OSEventType != OS_EVENT_TYPE_SEM) {  
        OS_EXIT_CRITICAL();  
        return (OS_ERR_EVENT_TYPE);  
    }
```

```
    if (pevent->OSEventGrp) {
```

是否有任务在等待该信号量

```
        OSEventTaskRdy(pevent, (void *)0, OS_STAT_SEM);
```

```
        OS_EXIT_CRITICAL();
```

若有, 调用OS\_EventTaskRdy()函数  
使优先级最高等待任务就绪

```
        OSSched();
```

进行任务调度

```
        return (OS_NO_ERR);
```

```
} else {
```

```
    if (pevent->OSEventCnt < 65535) {
```

若没有任务在等待该信号量, 则信  
号量计数值简单加1, 并返回。但不  
能溢出。

```
        pevent->OSEventCnt++;
```

```
        OS_EXIT_CRITICAL();
```

```
        return (OS_NO_ERR);
```

```
} else {
```

```
    OS_EXIT_CRITICAL();
```

```
    return (OS_SEM_OVF);
```

```
}
```

```
}
```

## 有关信号量等待超时时间

➤ 等待超时时间设置为‘0’，表示不启用等待超时。

- OSTimeTick只对OSTCBDly值非零的任务减1，并进行当前值判断。
- 若OSTCBDly值为‘0’，OSTimeTick不会进入判断过程。

➤ 等待超时时间设置为非零

- OSTimeTick会对OSTCBDly值执行减1操作，当减到‘0’时，会强制将其置为就绪态。
- Pend函数得到的将是一个超时退出信号量。

## 第十五讲(理解为主)

### 1. 优先级反转的概念

背景: μC/OS-II不允许多任务处于同一优先级,在利用信号量实现互斥操作时,任务是不对等的,存在优先级差异

#### 【优先级翻转】

在有多个任务需要使用共享资源的情况下,可能会出现高优先级任务被低优先级任务阻塞,并等待低优先级任务执行的现象;高优先级任务需要等待低优先级任务释放资源,而低优先级任务又正在等待中等优先级任务,这种现象就被称为优先级反转

#### 【解决方案】

- 优先级继承协议

当更高优先级任务Ta被低优先级任务Td阻塞时, Td暂时继承Ta的优先级;这将防止中间优先级任务抢占任务Td,而使高优先级任务Ta的阻塞时间延长

- 优先级天花板  
将申请某资源的任务的优先级提升到可能访问该资源的所有任务中最高优先级任务的优先级(这个优先级称为该资源的优先级天花板)
- 区别
  - 优先级继承：只有当占有资源的低优先级任务被阻塞时，才会提高占有资源任务的优先级
  - 优先级天花板：不论是否发生阻塞都提升

## 2. uC/OS 中互斥型信号量实现机制

μC/OS-II中，在处理互斥型信号量时，没有被占用的、略高于最高优先级的优先级被保留给“优先级继承优先级（PIP）”，融合了“优先级继承”和“优先级天花板”的解决方案【预留优先级】

- 只有在发生优先级反转时才提升优先级

**【uC/OS 中互斥型信号量Mutex】** ——对比信号量事件的实现机制

只能做互斥操作，信号量属于二值信号量;Mutex是事件的一种，只能供任务使用并且因为牵扯到优先级继承优先级等任务级的处理，所以不能用于中断

- 建立一个互斥型信号量

```
OS_EVENT *OSMutexCreate (INT8U prio, INT8U *err)
```

```
{
```

```
    OS_EVENT *pevent;
```

互斥型信号量为二值信号量，  
无需计数值参数；

```
if (OSIntNesting > 0) {
```

需要一个优先级作为PIP

```
    *err = OS_ERR_CREATE_ISR;
```

```
    return ((OS_EVENT *)0);
```

不在中断中调用

```
}
```

```
OS_ENTER_CRITICAL();
```

```
if (OSTCBPrioTbl[prio] != (OS_TCB *)0) {
```

```
    OS_EXIT_CRITICAL();
```

检查PIP是否被占用

```
    *err = OS_PRIO_EXIST;
```

```
    return ((OS_EVENT *)0);
```



```
OSTCBPrioTbl[prio] = (OS_TCB *)1;
```

保留该优先级

```
pevent = OSEventFreeList;
```

获取一个空余事件控制块

```
if (pevent == (OS_EVENT *)0) {
```

```
    OSTCBPrioTbl[prio] = (OS_TCB *)0;
```

```
    OS_EXIT_CRITICAL();
```

```
    *err = OS_ERR_PEVENT_NULL;
```

```
    return (pevent);
```

```
}
```

```
OSEventFreeList = (OS_EVENT *)OSEventFreeList->OSEventPtr;
```

```
OS_EXIT_CRITICAL();
```

```
pevent->OSEventType = OS_EVENT_TYPE_MUTEX; 设置事件类型为MUTEX
```

```
pevent->OSEventCnt = (INT16U)((INT16U)prio << 8) |
```

```
    OS_MUTEX_AVAILABLE;
```

计数器高8位将PIP保存，低8位  
位置信号量（二值）为有效

```
pevent->OSEventPtr = (void *)0;
```

```
OS_EventWaitListInit(pevent);
```

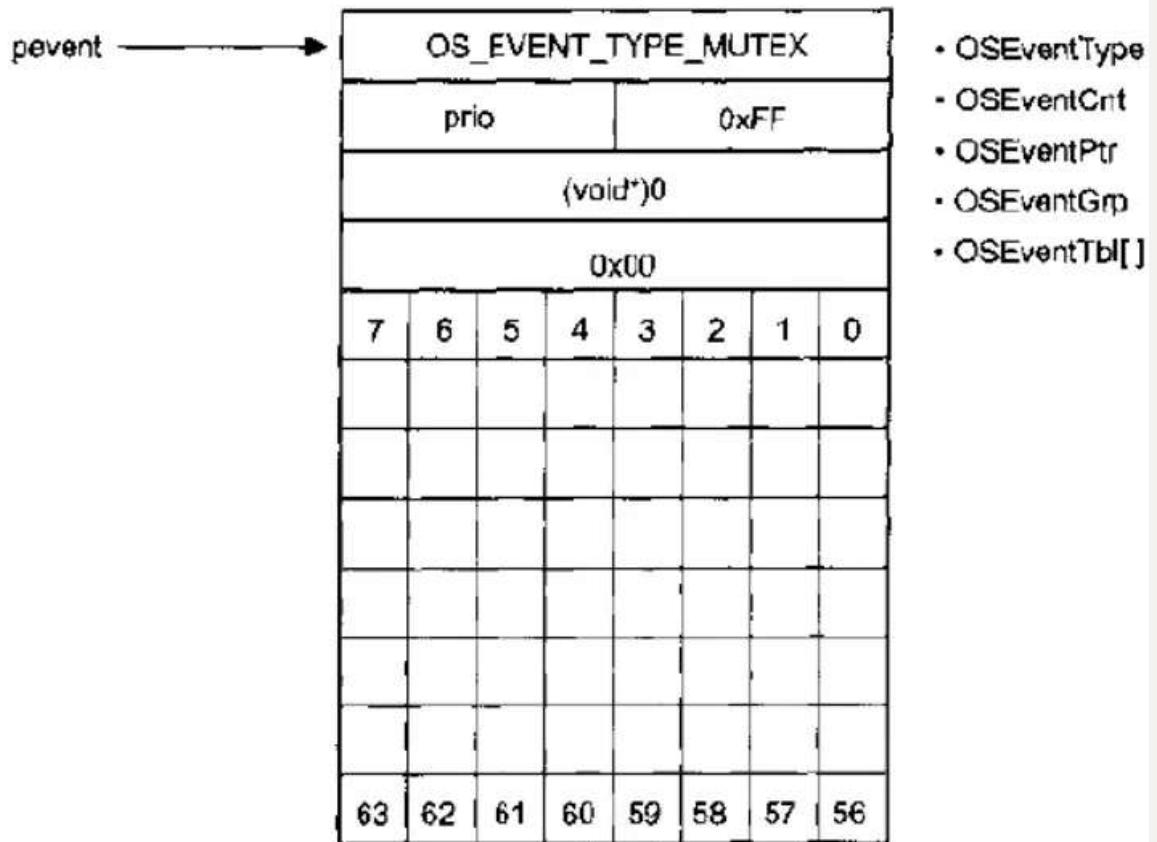
初始化等待事件列表

```
*err = OS_NO_ERR;
```

```
return (pevent);
```

```
}
```

## OS\_EVENT



注：表中所有值均初始化为0。

OSEventCnt被划分为高8位和低8位，其中高8位表示保留优先级；低8位为全1时表示信号量有效没有被占用，当信号量被占用时，低8位标识占有该信号量的任务优先级即0~63

- 等待一个互斥型信号量(P操作)

```

void OSMutexPend (OS_EVENT *pevent, INT16U timeout, INT8U *err)
{
    INT8U    pip;
    INT8U    mprio;
    BOOLEAN   rdy;
    OS_TCB   *ptcb;
    INT8U    y;

    if (pevent->OSEventType != OS_EVENT_TYPE_MUTEX) {
        *err = OS_ERR_EVENT_TYPE;
        return;
    }

    if (OSIntNesting > 0) {
        *err = OS_ERR_PEND_ISR;
        return;
    }

    OS_ENTER_CRITICAL();
    if ((INT8U)(pevent->OSEventCnt & OS_MUTEX_KEEP_LOWER_8) ==
        OS_MUTEX_AVAILABLE) {
        pevent->OSEventCnt &= OS_MUTEX_KEEP_UPPER_8;
        pevent->OSEventCnt |= OSTCBCur->OSTCBPrio;
        pevent->OSEventPtr = (void *)OSTCBCur;
        OS_EXIT_CRITICAL();
        *err = OS_NO_ERR;
        return;
    }
    否则，该任务就要进入等待状态，直到信号量被释放（或者超时唤醒）
    pip = (INT8U)(pevent->OSEventCnt >> 8);
    mprio = (INT8U)(pevent->OSEventCnt & OS_MUTEX_KEEP_LOWER_8);
    ptcb = (OS_TCB *)(pevent->OSEventPtr);
    if (ptcb->OSTCBPrio > pip) {
        if (mprio > OSTCBCur->OSTCBPrio) {
            y = ptcb->OSTCBY;
            if ((OSRdyTbl[y] & ptcb->OSTCBBitX) != 0) {
                OSRdyTbl[y] &= ~ptcb->OSTCBBitX;
                if (OSRdyTbl[y] == 0) {
                    OSRdyGrp &= ~ptcb->OSTCBBitY;
                }
            }
            rdy = TRUE;
        }
    }
}

```

**类型检查**

**不在中断中调用**

**测试该信号量是否可用；如果可用，则cnt的低8位为FF**

**将得到该信号量的任务优先级保存在cnt的低8位**

**将OSEventPtr指向任务控制块**

**提取mutex中的PIP（高8位）**

**占用mutex的优先级及该任务控制块**

**占用mutex的任务优先级不等于PIP（说明没有被提升过），且原始优先级比当前任务优先级低，即出现优先级反转问题。占用mutex的任务优先级就被提升到mutex的优先级继承优先级PIP，以期尽快释放mutex**

**确认占用mutex的任务是否进入就绪态，并做相应处理**

```

} else {
    rdy = FALSE;
}
ptcb->OSTCBPrio = pip; 提升占用mutex的任务优先级为PIP
ptcb->OSTCBY = ptcb->OSTCBPrio >> 3;
ptcb->OSTCBBitY = OSMapTbl[ptcb->OSTCBY];
ptcb->OSTCBX = ptcb->OSTCBPrio & 0x07;
ptcb->OSTCBBitX = OSMapTbl[ptcb->OSTCBX];
if (rdy == TRUE) { 如果任务已经就绪，修改就绪表
    OSRdyGrp |= ptcb->OSTCBBitY;
    OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
}
OSTCBPrioTbl[pip] = ptcb;
}

OSTCBCur->OSTCBStat |= OS_STAT_MUTEX; 设置当前任务相应状态位
OSTCBCur->OSTCBDly = timeout; 设置当前任务等待超时时间
OS_EventTaskWait(pevent); 将当前任务加入事件等待列表
OS_EXIT_CRITICAL();
OS_Sched(); 进行任务调度
OS_ENTER_CRITICAL();
if (OSTCBCur->OSTCBStat & OS_STAT_MUTEX){
    OS_EventTO(pevent);
    OS_EXIT_CRITICAL();
    *err = OS_TIMEOUT;
    return;
}
OSTCBCur->OSTCBEVENTPtr = (OS_EVENT *)0; Mutex超时返回
OS_EXIT_CRITICAL();
*err = OS_NO_ERR; Mutex正常返回

```

- 释放一个互斥型信号量(V操作)

```

INT8U OSMutexPost(OS_EVENT *pevent)
{
    INT8U    pip;
    INT8U    prio;
    INT8U    y;

    if (OSIntNesting > 0) {
        return (OS_ERR_POST_ISR);
    }

    if (pevent->OSEventType != OS_EVENT_TYPE_MUTEX) {
        return (OS_ERR_EVENT_TYPE);
    }

    OS_ENTER_CRITICAL();
    pip = (INT8U)(pevent->OSEventCnt >> 8);           // Mutex的PIP (高8位)
    prio = (INT8U)(pevent->OSEventCnt & OS_MUTEX_KEEP_LOWER_8);
    if (OSTCBCur->OSTCBPrio == pip) {                   // 占用mutex的任务优先级 (低8位)
        y      = OSTCBCur->OSTCBY;
        OSRdyTbl[y] &= ~OSTCBCur->OSTCBBitX;            // 判断当前任务优先级是否为PIP
        if (OSRdyTbl[y] == 0) {                            // 若是, 将当前任务从任务就绪表
            OSRdyGrp &= ~OSTCBCur->OSTCBBitY;             // 中PIP位置上删去, 并放回到原来的优先级位置上
        }
        OSTCBCur->OSTCBPrio    = prio;
        OSTCBCur->OSTCBY      = prio >> 3;
        OSTCBCur->OSTCBBitY   = OSMapTbl[OSTCBCur->OSTCBY];
        OSTCBCur->OSTCBX      = prio & 0x07;
        OSTCBCur->OSTCBBitX   = OSMapTbl[OSTCBCur->OSTCBX];
        OSRdyGrp          |= OSTCBCur->OSTCBBitY;
        OSRdyTbl[OSTCBCur->OSTCBY] |= OSTCBCur->OSTCBBitX;
        OSTCBPrioTbl[prio]    = OSTCBCur;
    }
}

```

不在中断中调用

检查事件类型

Mutex的PIP (高8位)

占用mutex的任务优先级 (低8位)

判断当前任务优先级是否为PIP

若是, 将当前任务从任务就绪表  
中PIP位置上删去, 并放回到原  
来的优先级位置上

```

OSTCBPrioTbl[pip] = (OS_TCB *)1;           // 将PIP优先级置为初始状态
if (pevent->OSEventGrp != 0) {               // 查看是否有等待mutex的任务

    prio      = OS_EventTaskRdy(pevent, (void *)0, OS_STAT_MUTEX);
    pevent->OSEventCnt &= OS_MUTEX_KEEP_UPPER_8;
    pevent->OSEventCnt |= prio;                // 若有，则唤醒最高优先级的任务，并保存该优先级
    pevent->OSEventPtr = OSTCBPrioTbl[prio];
    OS_EXIT_CRITICAL();
    OS_Sched();                                // 进行任务切换
    return (OS_NO_ERR);
}

pevent->OSEventCnt |= OS_MUTEX_AVAILABLE;
pevent->OSEventPtr = (void *)0;                // 否则，将mutex置为有效，并返回
OS_EXIT_CRITICAL();
return (OS_NO_ERR);
}

```

### 3. 了解其它任务间通信机制(邮箱、队列和事件标志组)

- 消息邮箱

**μC/OS-II**中的另一种基于事件的通信机制，可以使一个任务或者中断服务程序向另一个任务发送一个指针型的变量。该指针指向了一个包含了“消息”的特定数据结构



- 消息队列

消息队列是μC/OS-II中又一种基于事件的通信机制。可以使一个任务或者中断服务程序向另一个任务发送多个以指针方式定义的变量。

- 根据具体应用，每个指针指向的数据结构可以有所不同

- 可看做多个邮箱组成的数组，只是共用了同一个等待任务列表

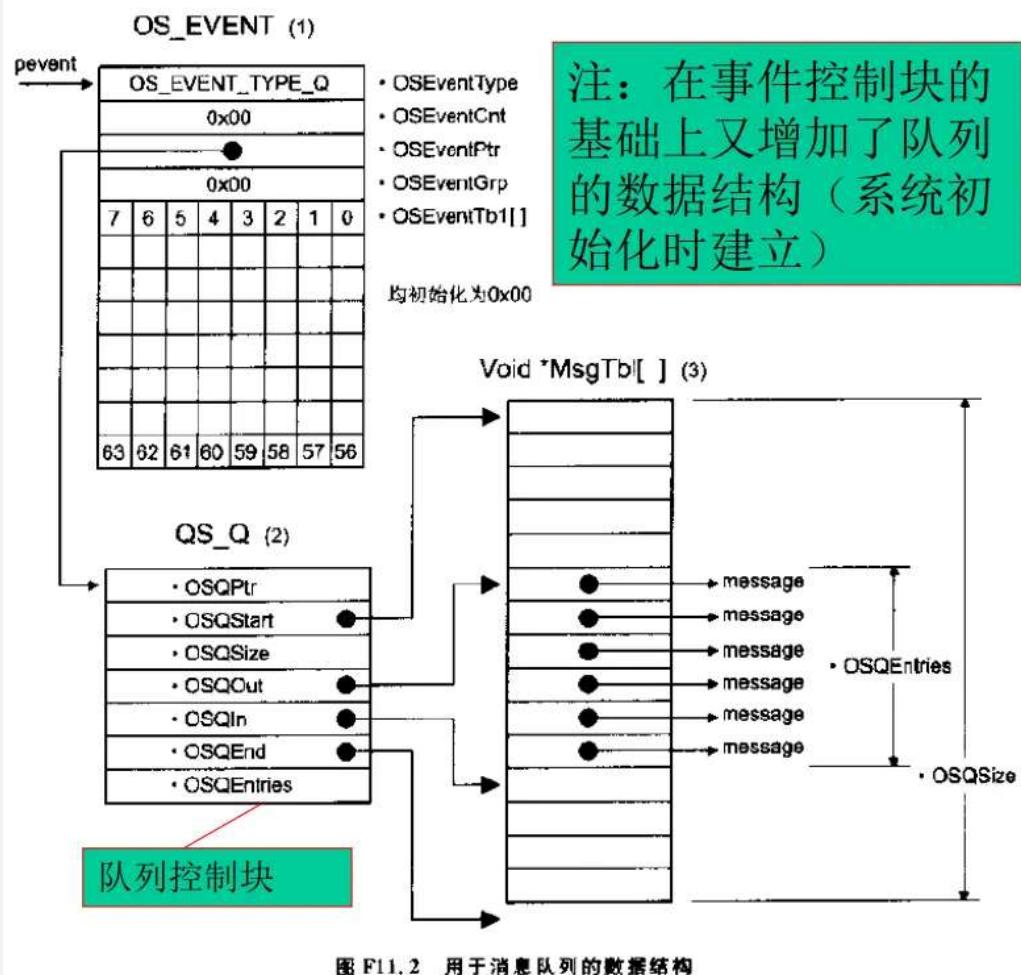
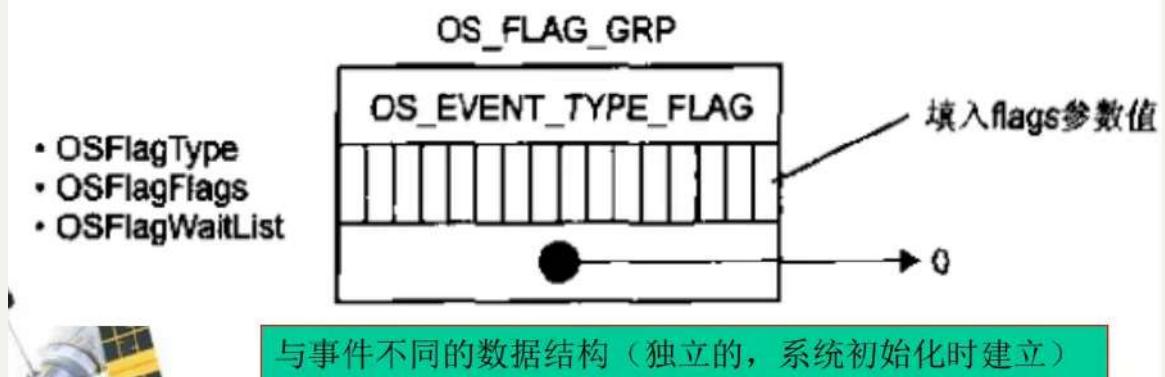


图 F11.2 用于消息队列的数据结构

- 事件标志组

μC/OS-II中任务可以等待多个事件的到来。一个任务也可以发出多个事件，任务和事件之间有着多对多的映射关系。

```
typedef struct {
    INT8U      OSFlagType;
    void       *OSFlagWaitList;
    OS_FLAGS   OSFlagFlags;
} OS_FLAG_GRP;
```



#### 4. 了解内存分区管理

μC/OS-II将内存按分区来处理

- 可以分配和释放固定大小的内存块
- 解决了内存碎片问题
- 分配、释放函数执行时间确定了

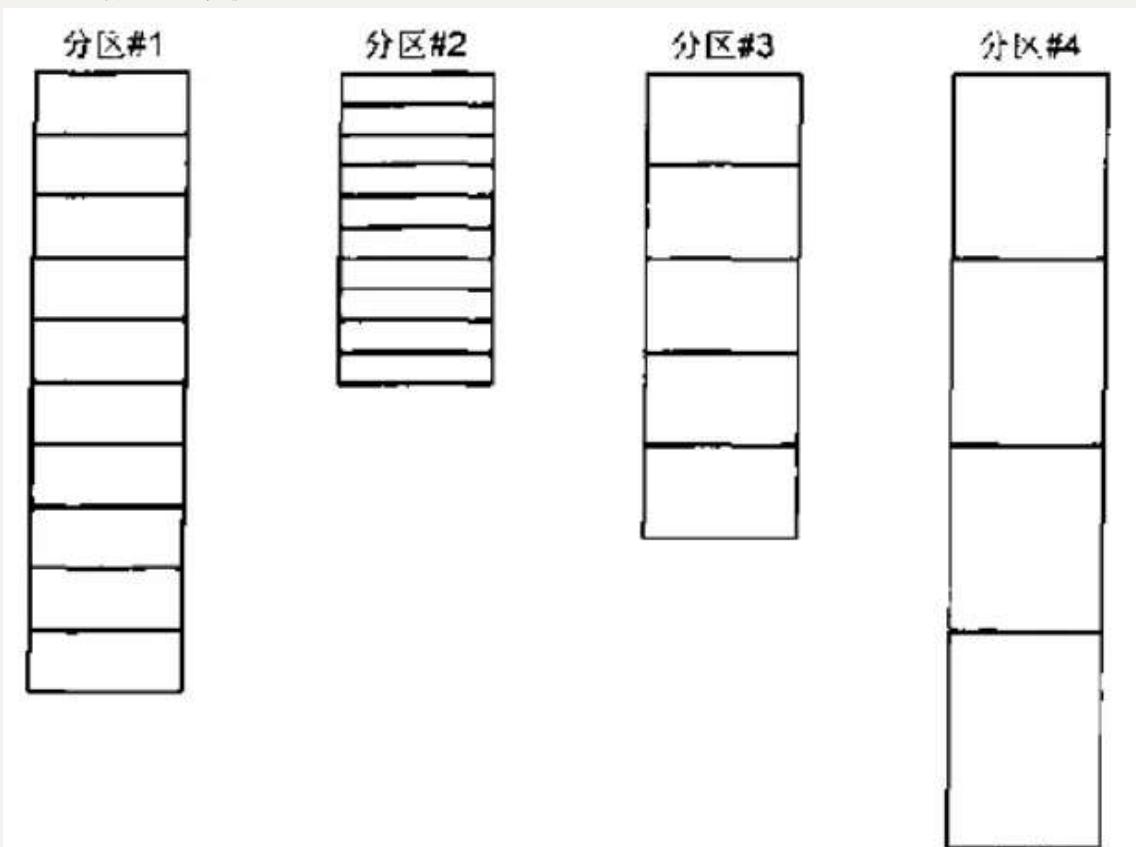


图 F12.2 多个内存分区

为了便于管理, μC/OS-II 使用内存控制块的数据结构跟踪每一个内存分区

```
typedef struct {
    void *OSMemAddr;           // 内存分区起始地址
    void *OSMemFreeList;        // 下一个可用内存块位置
    INT32U OSMemBlkSize;       // 分区内内存块大小
    INT32U OSMemNBlks;         // 分区内内存块总数量
    INT32U OSMemNFree;          // 当前空闲内存块数量
} OS_MEM;
```

# 题型分析

## 简答题

### 嵌入式系统的定义

- 标准定义：以应用为中心、以计算机技术为基础，软、硬件可裁剪，适应应用系统对功能、可靠性、成本、体积、功耗等严格要求的专用计算机系统
- 简单描述：嵌入到对象体中的专用计算机系统

### 嵌入式系统的三要素

1. 嵌入式：嵌入到对象体系中，有对象环境要求
2. 专用性：软、硬件按对象要求裁剪
3. 计算机：实现对象的智能化功能

### 嵌入式系统与通用计算机的区别

1. 专用性
2. 小型化(资源受限)
3. 软硬件设计一体化
4. 需要交叉开发环境

### 简述嵌入式实时系统的特点

- 实时性  
嵌入式实时系统分为硬实时系统和软实时系统，对任务的响应时间有极高的要求。系统必须在规定的时间内完成特定任务，否则可能导致系统故障或性能下降
- 结果正确性  
嵌入式实时系统的正确性不仅依赖于系统计算的逻辑结果(计算逻辑)，还依赖于产生这些结果的时间(计算时间)
- 可预测性  
嵌入式实时系统需要提供任务执行时间的可预测性，即任务在不同情况下都能在规定的时间内完成
- 稳定性和可靠性  
对于嵌入式实时系统来说，稳定性和可靠性至关重要。系统在长时间运行中不能发生崩溃，而且对于异常情况的处理必须是可靠的

#### 正确答案：

嵌入式实时系统的正确性不仅依赖系统计算的逻辑结果，还依赖于产生这个结果的时间。实时系统能够在指定或者确定的时间内完成系统功能和对外部或内部同步或异步事件做出响应。因此实时系统应该具备在事先先定义的时间范围内识别和处理离散事件的能力。

## CPU寄存器与内存的区别

- CPU寄存器是位于CPU内部的小型、高速存储单元，用于临时存储和快速访问指令和数据；
- 内存是位于计算机主板上的存储设备，用于存储程序和数据，供CPU读取和写入；
- 寄存器速度更快但容量有限，用于临时存储；
- 内存速度较慢但容量较大，用于长期存储。

## RISC指令集只包含有少量常用指令，为什么有时反而比CISC指令集的性能更好？

- 指令集分析：指令复杂度影响处理器实现性能，CISC中20%的指令使用率达到80%，但80%的复杂指令使用率仅20%，而RISC只包含有少量常用指令，不会造成指令集的冗余；
- 指令译码逻辑：CISC指令集采用微指令技术增加了硬件复杂度，而RISC采用硬连线的指令译码逻辑，开发简单；
- 指令执行分析：CISC指令长度不一，执行单条指令需要多个周期，而RISC指令长度固定，一个周期执行一条指令，通过少量常用指令的组合实现复杂的功能，容易实现高性能；
- Load/Store结构：RISC还采用Load/Store结构进行优化，完成数据在寄存器和存储器之间的传输，保证处理器直接从寄存器中获取操作数，能够加快运算的执行效率，实现更高的性能。

#### 正确答案：

RISC指令集指令功能简单，指令长度固定，绝大多数为单周期指令，指令和寻址方式种类较少，并且采用分开的Load/Store结构的存取指令，如此有利于采用硬连线的指令译码逻辑，更多级的流水线，以及其他一些提高指令执行效率的技术措施。总体看来，RISC指令集的执行性能并不会低。

## 简述宏和函数的区别？二者各自的优缺点是什么？

- 编译阶段：编译器会将相关代码放到宏名出现的每一个地方并且宏内容会被直接展开到代码中，宏的代码在任何出现宏名的地方都会编译一次；而函数的代码只需要编译一次，在编译时定义，但是实际在执行发生在运行时
- 调用返回：使用宏时不需要保存/恢复上下文，也不必返回；而调用函数时，需要保存/回复上下文，需要返回，会有函数调用开销；
- 优缺点：宏适用于代码简单的情况，不会有函数调用开销，但是容易造成代码膨胀；函数适用于代码复杂的情况，可以完成更复杂的功能，但是会有函数调用开销。

## 简述ARM异常响应过程以及异常向量表的作用

### 【ARM异常处理】

- 当中断发生时，系统执行完当前指令后，会跳转到相应的异常中断处理程序处执行；
- 执行完成后，程序返回到发生中断的指令的下一条指令处执行；
- 进入异常中断处理程序时要保存现场，返回时要恢复现场

#### 【异常向量表】

- 当异常发生时，处理器会把PC设置为一个特定的存储器地址；
- 这一地址放在被称为向量表(vector table)的特定地址范围内，用于对不同模式的异常提供异常处理程序的入口；
- 异常向量表通常放在存储地址的低端；

简述μcos中优先级图算法的优点

- 常数时间选择任务，无论任务的数量如何，选择最高优先级任务的时间是确定的
- 实现简单，容易维护
- 可以快速选择最高优先级任务
- 内存占用低

在ARM汇编语言程序里，什么是伪指令，他们有什么作用？

- ARM伪指令不是真正的ARM指令，在汇编编译器对源程序进行汇编处理时会被替换成对应的ARM指令或指令序列
- 伪指令主要用于指导汇编器怎样对源程序进行汇编，在汇编后不会生成机器码，仅在汇编过程中起作用

能否用全局变量来实现任务间的通信？它有什么优缺点？该如何正确使用？

可以用全局变量实现实务间的通信

- 优点：
  - 简单直观： 使用全局变量简化了任务间通信的实现，因为所有任务都能够直接访问这些共享变量
  - 高效： 全局变量的访问通常比通过消息传递等其他通信机制更为高效，因为直接在内存中进行读写操作
- 缺点：
  - 缺少同步与互斥关系的管理机制： 如果多个任务同时尝试读取或写入全局变量，可能导致意外行为和数据损坏
  - 难以调试： 由于多个任务可以同时访问全局变量，发现和调试由于共享变量而引起的问题可能会更加困难

- 扩展性差：当系统规模扩大时，全局变量的管理和维护可能变得复杂，使得系统的扩展性变差
- 正确使用的方法：
  - 加锁机制：在访问全局变量时，使用加锁机制确保同一时间只有一个任务可以访问该变量，从而避免相互竞争
  - 原子操作：对于某些特定的变量操作，可以使用原子操作，保证执行单条指令不会被中断，例如SWP实现互斥访问
  - 信号量机制：使用信号量机制来确保对全局变量的安全访问

**μcos**中任务的调度一般分为任务级调度和中断级调度，简述二者的区别

- 调度触发条件：任务级调度是当前任务放弃CPU触发的，中断级调度是由中断事件触发的
- 调度时机不同：任务级调度是在时间片用完、被高优先级任务抢占或等待某个事件时进行的，而中断级调度在中断返回的时候进行任务的调度
- 恢复现场不同：任务级调度在调度点每次都需要保护现场并恢复现场，而中断级调度因为在进行中断处理时已保存过现场，因此在调度点只需恢复现场
- 任务级调度与中断级调度是两套独立的机制，互不影响

优先级反转如何发生？又如何解决？

优先级反转会现象：

在有多个任务需要使用共享资源的情况下，可能会出现高优先级任务被低优先级任务阻塞，并等待低优先级任务执行的现象；高优先级任务需要等待低优先级任务释放资源，而低优先级任务又正在等待中等优先级任务，这就造成了优先级反转

解决方案：

- 优先级继承协议  
当更高优先级任务Ta被低优先级任务Td阻塞时，Td暂时继承Ta的优先级；这将防止中间优先级任务抢占任务Td，而使高优先级任务Ta的阻塞时间延长
- 优先级天花板  
将申请某资源的任务的优先级提升到可能访问该资源的所有任务中最高优先级任务的优先级(这个优先级称为该资源的优先级天花板)
- 优先级继承优先级  
在处理互斥型信号量时，没有被占用的、略高于最高优先级的优先级被保留，并用于在发生优先级反转时提升任务的优先级

简述**μcos**中解决互斥的3种办法，以及它们各自的优缺点

- 关中断

- 优点：简单直接，实现较为容易
- 缺点：力度较大，一旦使用会阻塞整个系统；不适用于实时系统，可能导致系统响应不及时
- 禁止任务切换——调度器上锁机制
  - 优点：系统开销较小，
  - 缺点：由于锁定所有任务，所以若禁止切换时间过长，影响系统的响应性能
- 使用信号量
  - 优点：只影响竞争共享资源的任务
  - 缺点：虽然μcos提出了互斥型信号量机制来解决优先级反转问题，但是这种方法的系统开销较大，对系统的响应性能会有一定影响

冯诺依曼体系结构与哈佛体系结构的区别？

- 存储器结构：冯诺依曼体系结构中程序指令存储器和数据存储器共用同一个存储体，而哈佛体系结构使用两个独立的存储器模块，分别存储指令和数据；
- 位宽：冯诺依曼体系结构中程序指令和数据宽度相同，而哈佛体系结构可以不同；
- 总线：冯诺依曼体系结构使用一条总线传输，而哈佛体系结构采用两条独立的总线，互不影响。

**正确答案：**

冯·诺依曼体系结构将指令和数据存储在一起，程序指令存储地址和数据存储地址指向同一个存储器的不同物理位置，因而程序指令和数据的宽度相同。哈佛体系结构使用两个独立的存储器模块分别存储指令和数据，每个存储模块都不允许指令和数据并存，使用独立的两条总线，分别作为CPU与每个存储器之间的专用通信路径，以便实现并行处理。

## 理解Load/Store结构

- 在通用寄存器中操作，从存储器中读某个值到寄存器，操作完再将其放回存储器中
- 只有Load和Store指令允许直接在内存和通用寄存器之间进行数据的读取和存储，其他例如算数和逻辑运算指令主要在通用寄存器之间进行，不直接涉及内存。有助于简化指令集结构，提高指令执行效率

## ARM指令的三地址指令格式与条件执行

- 三地址指令格式：ARM指令的三地址格式是指指令中含有目的操作数地址，第一源操作数地址，第二源操作数地址，根据源操作数地址得到源操作数进行运算再存到目的操作数地址；
- 条件执行：ARM状态下，几乎所有的指令均按照CPSR中条件码的状态以及指令中的条件域有条件地执行：条件满足时，指令执行；不满足时，指令被忽略。

## 交换指令**SWP**的特殊性

- 功能概述：交换指令能够在存储器和寄存器之间交换数据，常用于实现信号量操作，解决进程之间的同步与互斥关系；
- 特殊性体现：交换指令是一个原子操作，在并发环境中不会被中断，因此在解决信号量访问以及修改的过程中，可以用交换指令替代关中断机制实现信号量的操作，提升操作系统的执行效率。

## 简述**RISCV**的主要特征

**RISC-V**，是一种基于“精简指令集（RISC）”原则的指令集架构，它具有开源、重新设计(后发优势)、简单哲学、模块化指令集、指令集可扩展的特点。

## 嵌入式系统最小系统的构成

CPU能够运行所需的模块有（最小系统）：电源、时钟、复位、内存、调试接口

## 简述晶体与晶振的区别

时钟一般由晶体振荡器提供，而晶体和晶振是构成晶体振荡器的两种方式：

- 晶体(无源)：封装内部只含有晶体，没有内部电源，驱动电路由设计者提供；
- 晶振(有源)：封装中包含了完整的晶体振荡器电路，需要电源

## 理解**μcos**中任务级调度与中断级调度之间的关系

- 关系：**μcos**实现的任务级调度与中断级调度是两个相互独立的机制，互不影响；

### 【保护任务同时可以响应中断】

通过给任务调度器上锁机制，保证中断响应可以执行中断服务程序，但是在中断处理结束时，如果调度器已上锁，不会进行最高优先级任务的调度，而直接返回被中断的任务继续执行；否则由于**μcos**是基于抢占式中断调度的，会进行最高优先级任务的调度。

## 程序题

### C 程序与汇编程序混合编程——字符串拷贝

```
#include<stdio.h>
extern void strcpy(char *d, const char *s);
int main()
{
    const char *srcstr = "First string-source";
    char dststr[] = "Second string-destination";
    printf("Before copying:\n");
    printf("%s\n%s\n", srcstr, dststr);
    strcpy(dststr, srcstr);
    printf("After copying:\n");
    printf("%s\n%s\n", srcstr, dststr);
    return 0;
}
```

```
AREA scope CODE, READONLY
EXPORT strcpy
; 默认R0存储第一个参数, R1存储第二个参数
strcpy
    LDRB R2, [R1], #1      ; srcstr
    STRB R2, [R0], #1      ; dststr
    CMP R2, #0
    BNE strcpy
    MOV PC, LR
END
```

### LCD显示——水平直线

```
//已知像素颜色函数
void PutPixel(UINT16T x, UINT16T y, UINT16T c);
```

```
void Lcd_Draw_HLine(INT16T usX0, INT16T usX1, INT16T usY0, UINT16T
uccolor, UINT16T uswidth)
{
    INT16T usLen;

    if( usX1 < usX0 )
```

```
{  
    GUI_SWAP (usX1, usX0); // 保证x0<=x1,否则软中断  
}  
  
while( (uswidth--) > 0 ) // 绘制像素点行数  
{  
    usLen = usX1 - usX0 + 1;  
    while( (usLen--) > 0 ) // 绘制像素点列数  
    {  
        PutPixel(usX0 + usLen, usY0, ucColor);  
    }  
    usY0++; // 移至下一行  
}  
}
```

## μcos信号量任务管理——键盘识别显示

```
// 预定义函数  
INT8U keyScan(void); // 识别按键代号  
void keyPrint(INT8U x, INT8U y, INT8U c); // 显示按键代号
```

```
// 键盘中断服务程序  
extern OS_EVENT *Keypad_Sem;  
void __KeyPad(void)  
{  
    INT8U err;  
    ClearPending(BIT_EINT1); // 清除中断源  
    err = OSSemPost(Keypad_Sem); // 释放键盘信号量  
}
```

```

// 键盘显示任务
OS_EVENT *Keypad_Sem;
void KeyTask(void *data)
{
    Keypad_Sem = OSSemCreate(0);
    INT8U err, key, x = 100, y = 100; // 显示坐标
    while(1){
        OSSemPend(Keypad_Sem, 0, &err); // 申请键盘信号量
        key = keyScan(); // 获取中断按键
        keyPrint(x, y, key);
    }
}

```

基于**SWP**指令的信号量访问

```

SEM EQU 0x20001000
SEM_CHECK
    MOV R1, #0
    LDR R0, =SEM ; 信号量地址
    SWP R1, R1, [R0] ; 原子操作，访问信号量
    CMP R1, #0
    BEQ SEM_CHECK

```

**LED**--基于**GPIOB**口的发光二极管设计

```

#define rGPBCON (*(volatile unsigned *)0xCD74) //控制端口地址
#define rGPBDAT (*(volatile unsigned *)0xCD78) //数据端口地址
#define rGPBUP  (*(volatile unsigned *)0xCD7C) //上拉电阻端口地址
void led_on_off(void)
{
    rGPBCON = 0x15400; //配置控制端口
    rGPBUP = 0xFF; //配置上拉电阻
    rGPBDAT = 0; //配置数据端口
    deLay(30);
    rGPBDAT = 0x0010;
    deLay(1000);
}
int main(void)
{
    sys_init();
    while(1){

```

```
    led_on_off();
}
}
```

## μcos互斥管理——串口输出信息

```
OS_EVENT *UART_sem;
void myTask(void *data)
{
    unsigned char err;
    UART_sem = OSSemCreate(1);      // 创建信号量
    while(1){
        OSSemPend(UART_sem, 0, &err);
        uart_sending("I'm running now!");
        OSTimeDly(30);
        OSSemPost(UART_sem);
    }
}
```

## 程序题准备部分

### 多任务应用函数

```
/*-----创建任务的堆栈-----*/
#define STACKSIZE 256 // 堆栈大小
OS_STK Stack[STACKSIZE]; // 全局静态分配
const char Id = '1'; // 任务指针参数
/*-----任务创建函数-----*/
INT8U OSTaskCreate (void (*task)(void *pd), void *pdata, OS_STK
*pitos, INT8U prio);
// task: 指向任务代码的指针; pdata: 任务开始时, 传递给任务的指针参数
// pitos: 任务的堆栈栈顶指针; prio: 任务的优先级
/*-----任务删除函数-----*/
INT8U OSTaskDel(INT8U prio);
OSTaskDel(OS_PRIO_SELF);
// OS_PRIO_SELF: 删除当前任务及其创建的子任务
/*-----内核启动函数-----*/
void osstart (void);
// 创建完任务后必须通过osstart函数启动操作系统内核运行任务
```

```
// 其他函数
/*-----改变任务优先级函数-----*/
INT8U OSTaskChangePrio (INT8U oldprio, INT8U newprio);
/*-----挂起任务函数-----*/
INT8U OSTaskSuspend (INT8U prio);
/*-----恢复任务函数-----*/
INT8U OSTaskResume (INT8U prio);
/*-----时钟延迟函数-----*/
void OSTimeDly (INT16U ticks);
/*-----时钟延迟恢复函数-----*/
INT8U OSTimeDlyResume (INT8U prio);
// 提前恢复被延时的任务
```

```
// 创建多任务的实验
OS_EVENT *UART_sem;
unsigned char err;

/* allocate memory for tasks' stacks */
#define STACKSIZE 256

/* Global variable */
OS_STK Stack1[STACKSIZE];
OS_STK Stack2[STACKSIZE];
OS_STK StackMain[STACKSIZE];

const char Id1 = '1';
const char Id2 = '2';

void Task1(void *Id)
{
    while(1){
        OSSemPend(UART_sem, 0, &err);
        uart_sendstring("Task1 called.\r\n");
        OSTimeDly(50);
        OSSemPost(UART_sem);
    }
}

void Task2(void *Id)
{
    while(1)
```

```

    {
        OSSemPend(UART_sem, 0, &err);
        uart_sendstring("Task2 called.\r\n");
        OSSemPost(UART_sem);
    }
}

// 启动任务函数, TaskStart任务中创建两个子任务
void TaskStart(void *Id)
{
    UART_sem = OSSemCreate(1);

    OSTaskCreate(Task1, (void *)&Id1, &Stack1[STACKSIZE - 1], 2);
    OSTaskCreate(Task2, (void *)&Id2, &Stack2[STACKSIZE - 1], 3);
    OSTaskDel(OS_PRIO_SELF); // Delete current task--TaskStart
}

void Main (void)
{
    OSInit();
    OSTimeSet(0);

    /* create the start task */
    OSTaskCreate(TaskStart,(void *)0, &StackMain[STACKSIZE - 1],
0);

    /* start the operating system */
    osstart();
}

```

```

// 创建多任务--PPT实例
INT8U Task1Data, Task2Data;
INT8U Task1prio=1, Task2prio=2;
#define STACKSIZE 256
OS_STK Task1stk[STACKSIZE];
OS_STK Task1stk[STACKSIZE];
void Task1(void)
{
    while(1){
        Task1Data++;
        OSTimedly(25);
    }
}

```

```

}

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

void main()
{
    OSInit(); // 必要的
    OSTaskCreate(Task1, (void*)&Task1Data, (void
*)&Task1Stk[STACKSIZE-1], Task1prio);
    OSTaskCreate(Task2, (void *)&Task2Data, (void
*)&Task2Stk[STACKSIZE-1], Task2prio);
    OSStart(); // 必要的
}

```

## 信号量应用函数

```

OS_EVENT *pevent; // 创建事件变量
OS_EVENT *OSSemCreate(INT16U cnt); // 信号量创建函数
// cnt = 0~65535
void OSSemPend(OS_EVENT *pevent, INT16U timeout, INT8U *err); // 
等待信号量函数
// 如果timeout=0表示不启用等待超时
INT8U OSSemPost(OS_EVENT *pevent); // 释放一个信号量
/*-----常用操作-----*/
OS_EVENT *pevent;
unsigned char err;
OSSemPend(pevent, 0, &err); // P操作
err = OSSemPost(pevent); // V操作

```

```

// 信号量操作的实验--两个任务同步关系
// 交替执行，向串口输出信息
#define TaskStkLeath 128
OS_STK TestTask1Stk[TaskStkLeath];
OS_STK TestTask2Stk[TaskStkLeath];

OS_EVENT *sem1;

```

```
OS_EVENT *sem2;
unsigned char err;

void Task1(void *pdata)
{
    while(1)
    {
        OSSemPend(sem1, 0, &err);
        uart_sendstring("Now task1 is running!\r\n");
        OSTimeDly(30);
        uart_sendstring("Task1 resumed task2!\r\n");
        OSSemPost(sem2);
    }
}

void Task2(void *pdata)
{
    while(1)
    {
        OSSemPend(sem2, 0, &err);
        uart_sendstring("Now task2 is running!\r\n");
        OSTimeDly(30);
        uart_sendstring("Task2 resumed task1!\r\n");
        OSSemPost(sem1);
    }
}

void Main (void)
{
    OSInit();
    // 创建信号量
    sem1 = OSSemCreate(1);
    sem2 = OSSemCreate(0);
    // 创建两个任务
    OSTaskCreate(Task1,(void *)1,&TestTask1Stk[TaskStkLeath-1],5);
    OSTaskCreate(Task2,(void *)2,&TestTask2Stk[TaskStkLeath-1],6);

    OSStart();
}
```

// 信号量操作的实验--两个任务互斥关系

```
// 任务的挂起与恢复机制

#define TaskStkLeath 128
OS_EVENT *UART_sem;
unsigned char err;

OS_STK TestTask1stk[TaskStkLeath];
OS_STK TestTask2stk[TaskStkLeath];

void Task1(void *pdata)
{
    while(1)
    {
        OSSemPend(UART_sem, 0, &err);
        uart_sendstring("Task1: i will Suspend by myself\r\n");
        OSSemPost(UART_sem);

        OSTaskSuspend(2);          // 挂起任务 1

        OSSemPend(UART_sem, 0, &err);
        uart_sendstring("Task1: i Resumed by Task2\r\n");
        OSSemPost(UART_sem);

        OSTimeDly(10);
    }
}

void Task2(void *pdata)
{
    while(1)
    {
        OSSemPend(UART_sem, 0, &err);
        uart_sendstring("Task2: i am executing\r\n");
        OSSemPost(UART_sem);

        if( OSTaskResume(2) == OS_ERR_NONE ) // 唤醒任务1
        {
            OSSemPend(UART_sem, 0, &err);
            uart_sendstring("Task2: i Resumed Task1\n");
            OSSemPost(UART_sem);
        }
        OSTimeDly(5);
    }
}
```

```

}

void Main(void)
{
    OSInit();
    UART_sem = OSSemCreate(1);

    OSTaskCreate(Task1,(void *)1,&TestTask1Stk[TaskStkLeath-1],2);
    OSTaskCreate(Task2,(void *)2,&TestTask2Stk[TaskStkLeath-1],1);

    OSStart();
}

```

## 互斥型信号量应用函数

```

OS_EVENT *pevent; // 创建事件变量
OS_EVENT *OSMutexCreate (INT8U prio, INT8U *err); // 互斥信号量创建函数
// prio 保留优先级 PIP
void OSMutexPend (OS_EVENT *pevent, INT16U timeout, INT8U *err);
// 等待互斥信号量函数
// 如果timeout=0表示不启用等待超时
INT8U OSMutexPost (OS_EVENT *pevent); // 释放一个信号量
/*-----常用操作-----*/
OS_EVENT *pevent;
INT8U err;
pevent = OSMutexCreate(2, &err);
OSMutexPend(pevent, 0, &err); // P操作
err = OSMutexPost(pevent); // V操作

```

```

// 互斥型信号量操作的实验--两个任务互斥关系
#define TaskStkLeath 128
OS_EVENT *UART_sem;
unsigned char err;

OS_STK TestTask1Stk[TaskStkLeath];
OS_STK TestTask2Stk[TaskStkLeath];

void Task1(void *pdata)
{
    while(1)

```

```

    {
        OSMutexPend(UART_sem, 0, &err);
        uart_sendstring("Task1: i will Suspend by myself\r\n");
        OSMutexPost(UART_sem);

        OSTimeDly(10);
    }
}

void Task2(void *pdata)
{
    while(1)
    {
        OSMutexPend(UART_sem, 0, &err);
        uart_sendstring("Task2: i am executing\r\n");
        OSMutexPost(UART_sem);

        OSTimeDly(5);
    }
}

void Main(void)
{
    OSInit();
    UART_sem = OSMutexCreate(1, &err);

    OSTaskCreate(Task1,(void *)1,&TestTask1Stk[TaskStkLeath-1],2);
    OSTaskCreate(Task2,(void *)2,&TestTask2Stk[TaskStkLeath-1],3);

    osstart();
}

```

## LCD显示屏相关函数

```

/***********************
* func:      指定颜色填充区域
* para:      usLeft,usTop,usRight,usBottom -- 矩形区域坐标
*             ucColor -- appointed color value
* ret:       none
************************/

```

```
void Lcd_clr_rect(INT16T usLeft, INT16T usTop, INT16T usRight,
INT16T usBottom, UINT16T ucColor)
{
    UINT32T i, j;

    for (i = usLeft; i < usRight; i++)
        for (j = usTop; j < usBottom; j++)
    {
        PutPixel_16Bit_320240(i, j, ucColor); // 绘制像素点
    }
}
```

```
/****************************************************************************
 * func:      指定颜色绘制水平直线
 * para:      usX0,usY0 -- 起点坐标
 *             usX1 -- 终点x坐标
 *             ucColor -- appointed color value
 *             usWidth -- line's width
 *****/
void Lcd_Draw_HLine(INT16T usX0, INT16T usX1, INT16T usY0, UINT16T
ucColor, UINT16T uswidth)
{
    INT16T usLen;

    if( usX1 < usX0 )
    {
        GUI_SWAP (usX1, usX0);
    }

    while( (uswidth--) > 0 )
    {
        usLen = usX1 - usX0 + 1;
        while( (usLen--) > 0 )
        {
            PutPixel_16Bit_320240(usX0 + usLen, usY0, ucColor);
        }
        usY0++;
    }
}
```

```
/*****************************************************************************  
* func:      指定颜色绘制垂直直线  
* para:      usX0,usY0 -- 起点坐标  
*             usY1 -- 终点Y坐标  
*             ucColor -- appointed color value  
*             usWidth -- line's width  
*****/  
void Lcd_Draw_VLine (INT16T usY0, INT16T usY1, INT16T usX0, UINT16T  
ucColor, UINT16T usWidth)  
{  
    INT16T usLen;  
  
    if( usY1 < usY0 )  
    {  
        GUI_SWAP (usY1, usY0);  
    }  
  
    while( (usWidth--) > 0 )  
    {  
        usLen = usY1 - usY0 + 1;  
        while( (usLen--) > 0 )  
        {  
            PutPixel_16Bit_320240(usX0, usY0 + usLen, ucColor);  
        }  
        usX0++;  
    }  
}
```

```

 ****
 * func:      指定颜色绘制矩形
 * para:      usLeft,usTop,usRight,usBottom -- 矩形区域坐标
 *           ucColor -- appointed color value
 * ret:       none
 ****
void Lcd_Draw_Box(INT16T usLeft, INT16T usTop, INT16T usRight,
INT16T usBottom, UINT16T ucColor)
{
    Lcd_Draw_HLine(usLeft, usRight, usTop, ucColor, 1);
    Lcd_Draw_HLine(usLeft, usRight, usBottom, ucColor, 1);
    Lcd_Draw_VLine(usTop, usBottom, usLeft, ucColor, 1);
    Lcd_Draw_VLine(usTop, usBottom, usRight, ucColor, 1);
}

```

```

// 其他函数
 ****
* name:      Lcd_Draw_Line()
* func:      指定颜色绘制直线
* para:      usX0,usY0 -- 起点坐标
*           usX1,usY1 -- 终点坐标
*           ucColor -- appointed color value
*           usWidth -- line's width
 ****
void Lcd_Draw_Line(INT16T usX0, INT16T usY0, INT16T usX1, INT16T
usY1, UINT16T ucColor, UINT16T usWidth)
{
    INT16T usDx;
    INT16T usDy;
    INT16T y_sign;
    INT16T x_sign;
    INT16T decision;
    INT16T wCurx, wCury, wNextx, wNexty, wpy, wpx;

    if( usY0 == usY1 )
    {
        Lcd_Draw_HLine (usX0, usX1, usY0, ucColor, usWidth);
        return;
    }
    if( usX0 == usX1 )
    {

```

```

    Lcd_Draw_VLine (usY0, usY1, usX0, ucColor, uswidth);
    return;
}

usDx = abs(usX0 - usX1);
usDy = abs(usY0 - usY1);
if( ((usDx >= usDy && (usX0 > usX1)) ||
      ((usDy > usDx) && (usY0 > usY1))) )
{
    GUI_SWAP(usX1, usX0);
    GUI_SWAP(usY1, usY0);
}
y_sign = (usY1 - usY0) / usDy;
x_sign = (usX1 - usX0) / usDx;

if( usDx >= usDy )
{
    for( wCurx = usX0, wCury = usY0, wNextx = usX1,
          wNexty = usY1, decision = (usDx >> 1);
          wCurx <= wNextx; wCurx++, wNextx--, decision += usDy )
    {
        if( decision >= usDx )
        {
            decision -= usDx;
            wCury += y_sign;
            wNexty -= y_sign;
        }
        for( wpy = wCury - uswidth / 2;
              wpy <= wCury + uswidth / 2; wpy++ )
        {
            PutPixel_16Bit_320240(wCurx, wpy, ucColor);
        }

        for( wpy = wNexty - uswidth / 2;
              wpy <= wNexty + uswidth / 2; wpy++ )
        {
            PutPixel_16Bit_320240(wNextx, wpy, ucColor);
        }
    }
}
else
{
    for( wCurx = usX0, wCury = usY0, wNextx = usX1,

```

```

        wNexty = usY1, decision = (usDy >> 1);
        wCury <= wNexty; wCury++, wNexty--, decision += usDx )
    {
        if( decision >= usDy )
        {
            decision -= usDy;
            wCurx += x_sign;
            wNextx -= x_sign;
        }
        for( wpix = wCurx - uswidth / 2;
             wpix <= wCurx + uswidth / 2; wpix++ )
        {
            PutPixel_16Bit_320240(wpix, wCury, ucColor);
        }

        for( wpix = wNextx - uswidth / 2;
             wpix <= wNextx + uswidth / 2; wpix++ )
        {
            PutPixel_16Bit_320240(wpix, wNexty, ucColor);
        }
    }
}

```

## ARM汇编程序设计

```

//c程序g()返回5个整数的和
int g(int a, int b, int c, int d ,int e)
{
    return a + b + c + d + e;
}

```

```

; 利用堆栈进行子程序调用返回
TITLE 汇编函数中调用C函数实现加: a+2a+3a+4a+5a, a=12
EXPORT f ; 声明符号f可被其他文件引用
AREA f, CODE, READONLY
IMPORT g ; 使用伪操作IMPORT声明C程序g()
        STR lr, [sp, #-4]! ; 保存返回地址(堆栈满递减), 被调用, 保存的地址为调用
程序的下一条指令地址
        MOV r0, #12      ; a = 12
        ADD r1, r0, r0  ; r1 = 2*r0

```

```

    ADD r2, r1, r0 ; r2 = 3*r0
    ADD r3, r1, r2 ; r3 = 5*r0
    STR r3, [sp, #-4]! ; 第五个参数通过数据栈传递
    ADD r3, r1, r1 ; r3值设为4*i
    BL g ; 调用C程序
    ADD sp, sp, #4 ; 调整数据栈指针，准备返回
    LDR pc, [sp], #4 ; 返回
END

```

```

; 利用LR进行子程序调用返回
TITLE 子程序调用示例
AREA Init, CODE, READONLY
ENTRY
start
    MOV R0, #10
    MOV R1, #30
    BL subname ; 调用子程序subname
return
    MOV R2, R0
    ...
subname
    ADD R0, R0, R1 ; 子程序
    MOV PC, LR ; 从子程序返回
END

```

```

// 求最大公约数
int gcd (int a , int b)
{
    while (a!=b){
        if (a>b)
            a=a-b;
        else
            b=b-a;
    }
    return a;
}

```

```

TITLE 求最大公约数--Version1
AREA gcd, CODE, READONLY ALIGN=3
ENTRY

```

```
start
    CMP R0, R1
    BEQ stop
    BLT less
    SUB R0, R0, R1
    B start
less
    SUB R1, R1, R0
    B start
stop
    NOP
    MOV PC, LR
```

```
TITLE 求最大公约数--version2[条件执行]
AREA gcd, CODE, READONLY ALIGN=3
ENTRY
start
    CMP R0, R1      ; 比较a和b大小
    SUBGT R0, R0, R1 ; if(a>b) a=a-b
    SUBLT R1, R1, R0 ; if(a<b) b=b-a
    BNE start        ; if(a!=b) 跳转, 继续
    MOV PC, LR
```

```
if (a==0 || b==1)
    c = d + e;
```

```
; 条件判断的等价写法
CMP R0, #0
CMPNE R1, #1
ADDEQ R2, R3, R4
```

```
TITLE 数据块复制
AREA Block, CODE, READONLY ; 设置本段程序的名称(Block)及属性
num EQU 20 ; 设置将要复制的字数
ENTRY
start
    LDR r0, =src ; r0寄存器指向源数据区src
    LDR r1, =dst ; r1寄存器指向目标数据区dst
    MOV r2, #num ; r2指定将要复制的字数
```

```

    MOV sp, #0x400 ; 设置数据栈指针(r13), 用于保存工作寄存器数值
Blockcopy ; 进行以8个字为单位的数据复制
    MOVS r3, r2, LSR #3 ; 需要进行的以8个字为单位的复制次数
    BEQ Copywords ; 对于剩下不足8个字的数据, 跳转到copywords, 以字为
单位复制
    STMFD sp!, {r4-r11} ; 保存工作寄存器

Octcopy
    LDMIA r0!, {r4-r11} ; 从源数据区读取8个字的数据, 放到8个寄存器中, 并更新
源数据区指针r0
    STMIA r1!, {r4-r11} ; 将这8个字数据写入到目标数据区中, 并更新目标数据区指
针r1
    SUBS r3, r3, #1 ; 将块复制次数减1
    BNE Octcopy ; 循环, 直到完成以8个字为单位的块复制
    LDMFD sp!, {r4-r11} ; 恢复工作寄存器值

Copywords
    ANDS r2, r2, #7 ; 剩下不足8个字的数据的字数
    BEQ Stop ; 数据复制完成

Wordcopy
    LDR r3, [r0], #4 ; 从源数据区读取1个字的数据, 放到r3寄存器中并更新目
标数据区指针r0
    STR r3, [r1], #4 ; 将这r3中数据写入到目标数据区中, 并更新目标数据区指
针r1
    SUBS r2, r2, #1 ; 将字数减1
    BNE Wordcopy ; 循环, 直到完成以字为单位的数据复制

Stop
    MOV r0, #0x18 ; 程序退出
    LDR r1, =0x20026
    SWI 0x123456

AREA BlockData, DATA, READWRITE ; 定义数据区
    Src DCD 1, 2, 3, 4, 5, 6, 7, 8, 1, 2, 3, 4, 5, 6, 7, 8, 1, 2,
3, 4
        ; 定义源数据区src
    Dst DCD 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
0, 0
        ; 定义目标数据区dst
END ;结束汇编

```

**TITLE** 跳转表实现程序跳转

```

AREA Jump, CODE, READONLY ; 设置本段程序的名称(Jump)及属性
num EQU 2 ; 跳转表中的子程序个数

```

```
ENTRY ; 程序执行的入口点
Start
    MOV r0, #0 ; 设置3个参数，然后调用子程序arithfunc，进行算术运算
    MOV r1, #3
    MOV r2, #2
    BL arithfunc ; 调用子程序arithfunc

Stop
    MOV r0, #0x18      ; 程序退出
    LDR r1, =0x20026
    SWI 0x123456

Arithfunc ; 子程序arithfunc入口点
    CMP r0, #num ; 判断选择子程序的参数是否在有效范围之内
    MOVHS pc, lr ; 若不在，则直接返回
    ADR r3, JumpTable ; 读取跳转表的基地址(ADR是伪指令)
    LDR pc, [r3, r0, LSL #2] ; 根据参数r0的值跳转到相应的子程序

JumpTable
    DCD DoAdd
    DCD DoSub

DoAdd ; 子程序DoAdd执行加法操作
    ADD r0, r1, r2
    MOV pc, lr

DoSub ; 子程序DoSub执行减法操作
    SUB r0, r1, r2
    MOV pc, lr

END ; 结束汇编
```