

计科嵌入式系统设计复习笔记

刘一喆

21009201180



Part 1 嵌入式系统概述

1. 嵌入式系统

嵌入到对物体中的专用计算机系统；

嵌入性：嵌入到对物体中，有时环境要求；

专用性：软、硬件按对需求要求可裁减；

计算机：实现对物体的智能化功能。

2. 与通用计算机的区别、特点

① 专用性：专门的处理器，专用的功能算法，用于满足对需求的要求
定制硬件

② 小型化：计算资源有限、结构紧凑

③ 软硬件设计一体化：包括应用软件与OS的一体化设计，硬件与软件的依赖性强

④ 需要交叉开发环境，开发由宿主机完成

⑤ 嵌入式系统对实时性要求严格，一般要求RTOS
通用计算机相反说即可

3. 应用领域：汽车、军事、人工智能、日常生活、生物技术

4. 分类

按硬件
 系统级：工控机、PC104

板级：带CPU的主板

片级：单片机、DSP，微处理器

按软件
非实时系统
软实时系统
硬实时系统

5. 嵌入式处理器分类

嵌入式微控制器：单片机

嵌入式开放处理器：ARM, MIPS, POWER PC

CPU与内存集成一个片

DSP：数字信号处理器

SOC, SOPC

CPU独立一片

硬件个体承接机构

6. 存储器组织

冯诺依曼结构

不分

哈佛结构
存储器分指令存储器

和数据存储器

使用独立两条总线

51单片机

7. 控制器的组成方式：

组合逻辑型：快、复杂、

存储逻辑型：慢、规则

8. 流水线技术

取指 → 译码 → 执行 → 写回

对 CPU 性能影响：

并行（指令）提高了 CPU 的利用率和效率

9. CPU 字长

CPU 寄存器的长度，也即微处理器一次运算处理的数据宽度；

CPU 字长越长，处理速度越快

10. CISC 与 RISC

	CISC	RISC
指令集	复杂指令集，变长 执行时间不同	精简指令集，定长 一周期一条指令
流水线	指令的执行需要调用 一个微程序	每周期前进一步
寄存器	通用寄存器数目少	通用寄存器多
Load/store	处理器能够直接处理 存储器中数据	独立 Load/store 指令 在寄存器与内存传输
控制器	微程序	硬布线
代表	X86 系	ARM 系

11. 前后台系统

前台中断、后台循环

处理突发事件

对时间要求严格事件

轮询多任务

对时间要求不严格事件

12. 事件触发结构

状态机 → 事件触发

Part 2 ARM 简介

1. ARM 处理器

- 第一个 ARM 原型 — 剑桥
- 特有 16 / 32 位双指令集
- ARM7 系列应用最广

2. ARM Cortex 系列

M 系列、R 系列，A 系列

3. ARM 体系结构版本

V1 ~ V4 ~ V7 ~ V8

采用

Thumb

出现

Cortex

扩展到 64 位

{ ARM 状态：32 位指令

thumb：16 位压缩指令
状态

} 处理器

A：基于虚拟内存
技术的系统

R：实时 OS 控制
的系统

M：嵌入式

4 Thumb 技术特点

1 16bit 的代码长度

2 32bit 的执行效率

5. ARM 的 7 种运行模式

1. 用户：正常程序工作模式，不能切换

2. 系统：OS 特权状态，有切换模式权限

3. 快中断 (fiq)：支持高速数据传输和通道处理，IRQ 异常响应时进入此模式

4. 中断 (irq)：用于通用中断处理，IRQ 异常响应时进入此模式

5. 管理：OS 保护代码，系统复位和软中断响应时进入此模式

6. 中止：用于支持虚拟内存和内存保护

7. 未定义：支持硬件协处理器的软件仿真，未定义指令异常进入此模式

2-7 为特权模式，3-7 为异常模式

6. ARM 的 37 个寄存器的组织形式

30 个通用寄存器，1 个用作 PC，1 个用作 CPSR，5 个用作 SPSR
R15

其中大多数寄存器共 6 个

- $R_0 \sim R_{13}$ 为保存数据的通用寄存器；完全通用
任何指令都可以用
- * 其中 $R_0 \sim R_7$ 未分组，只有 1 个
 - * $R_8 \sim R_{14}$ 为分组寄存器，不同模式会对应不同的寄存器
 - * $R_8 \sim R_{12}$ 有两个分组的寄存器，与 R_8 与 $R_8\text{-f}ig$
 - * R_{13}, R_{14} 为有 6 个分组的寄存器
 $R_{13}, R_{13\text{-SVC}}, R_{13\text{-abt}}, R_{13\text{-und}}, R_{13\text{-irq}}, R_{13\text{-f}ig}$
对应 5 种异常模式
 - * R_{13} 常用作函数返回地址
 - * R_{14} 为连接寄存器 LR，保存子程序或异常返回的地址
 - * R_{15} ：PC，即程序计数器，存下一条上 CPU 的指令地址。

CPSR：状态寄存器，内部有一些标志位、模式位
SPSR：每一种异常模式下又有一个备用的状态寄存器 SPSR，SPSR 用于备份原始 CPSR，异常退出时向 SPSR 恢复 CPSR。

7. Load / Store 结构

- 在通用寄存器中操作；
- 从存储器中读某个值，操作完后再放回存储器中；

8. ARM 指令集

* 指令分类

数据处理类，load/store类，跳转类，其他

* 指令基本格式

$\langle OP \rangle \{ \langle cond \rangle \} \{ S \}, \langle Rd \rangle, \langle R, n \rangle, \{ \langle operand2 \rangle \}$

OP：助记符（操作码）

cond：执行条件

S：是否影响CPSR的值

Rd：目的寄存器

Rn：存第一操作数寄存器

operand2：第二操作数

* 常见指令

ADD R2, R1, R0 加法，数据处理类，R₀+R₁ → R₂

ADDS R1, R1, #1 加法，R₁+1 → R₁，影响CPSR

LDR R0, [R1] , [R1] → R0

BEQ DATAEVEN , EQ 条件满足，跳转到]

SUBNE \$R1, \$R1, #0X0D

条件相减法 (NE时) , \$R1 → 0X0D → \$R1, 影响 CPSR
* 条件码:

EQ (\$Z=1) 相等

NE (\$Z=0) 不相等

GT 带符号数大于

LT 带符号数小于

条件码占指令码的高4位

* ARM 寻址方式:

1> 立即寻址: 地址码部分包含操作数

2> 寄存器寻址: 地址码全为寄存器

 SUB \$R0, \$R1, \$R2

3> 寄存器移位寻址: 有

如 MOV \$R0, \$R2, LSL #3

ANDS \$R1, \$R1, \$R2, LSL \$3

\$R2 左移 \$R3 位, \$R1 相与, 双入 \$R1

LSL: 左移左移, LSR 右移右移

ASL: 算术左移, ASR 算术右移

4> 基址寻址 (类似于 8086 的寄存器间接寻址)

LDR \$R0, [\$R1]

load/store 类指令寻址

STR \$R0, [\$R1]

方式

8086的寄存器

⇒ 变址寻址 (类似于相对寻址) 寄存器
例: LDR R0, [R1, #4] 立即数也可以
LDR R0, [R1, #-4]
它是 Load / Store 类指令的寻址方式

9. 具体指令介绍

1> MOV 用于寄存器之间，或传立即数

如 MOV R1, R0, LSL#3

2> MVN 先取反，再位左

MVN R0, #0

3> ADD 指令 ADD R0, R2, R3, LSL# |

4> ADC 指令 ADC 带 C标志位一起加

5> SUB SUB R0, R2, R3, LSL# |

6> SBC SBC要减去 C标志位的值

7> AND 指令 '与' AND R0, R0, #3

8> ORR 指令 '或' ORR R0, R0, #3

9> EOR 指令 '异或' EOR R0, R0, #3

常用于反转操作数某些位

10> BIC 指令 消除操作数1中的某些位

BIC R0, R0, #3 消除低两位

- 11> CMP 指令 $CMP R1, R0$
无条件更新 CPSR 中条件标志位的值
- 12> CMN 指令，操作数 1 与操作数 2 取补后进行比较。
- 13> TST 指令 TST 操作数 1，操作数 2
按位与，不保留结果，但置位 CPSR
- 14> TEQ 按位异或 TEQ R1, R2，置位 CPSR
- 15> 乘法指令
 $MUL R0, R1, R2$
- 16> LDR：左到右，内存 \rightarrow 寄存器
STR：左到右，寄存器 \rightarrow 内存
+B：字节 +H：半字
- 17> SWP 指令
SWP: $R0, R1, [R2]$
- SWP: $R0, R0, [R2]$ 实现寄存器和内存数据块的交换
- 18> B 指令 B: 同标址
BL 指令 R14 保存返回地址

小结：运算类（算术，逻辑，BIC）以及 swap
是3地址格式，传送类（MOV, LDR, STR）
为2地址格式，测试类为2地址格式，
跳转类为1地址格式

10. ARM 汇编伪操作

* 定义全局变量

GBLA: 算术
* 定义局部变量

LCLA, LCLL, LCLS

* 对变量赋值

SETA, SETL, SETS

* 汇编控制伪操作

1. If 逻辑表达式

指令1

ELSE

指令2

ENDIF

2. WHILE

WHILE (逻辑表达式)

指令

WEND

> MACRO 宏定义伪操作 MEND 结束
MEXIT 从宏定义跳转出去

* AREA: 定义一个代码段或数据段

AREA Init, CODE, READONLY
只读

指令序列

END 一段结束

* ENTRY: 汇编程序入口标记

* EQU: 定义常量名称, '*' 可代替

如 Test EQU (*) 50;

* EXPORT: 声明全局变量

* IMPORT: 引用其他文件定义的变量

* EXTERN: 等同于 IMPORT

* GET: INCLUDE

11. ARM伪指令

常写加载到寄存器

> 不是真正的 ARM 指令

> 汇编时要替换成对应的 ARM 指令序列

③ 例子: ADR, AND, LDR (LDR R1, =0xFF)

12. ARM 子程序调用 $\hookrightarrow \text{MOV R1, } 0xFF$

BL 标号(地址), 子程序结尾 MOV PC, LR

13. ARM 汇编示例：求最大公约数

Start : CMP R0, R1

SUB GT R0, R0, R1

SUB LT: R1, R1, R0

BNE Start

MOV PC, LR ;子程序结束

Part 3 ARM汇编语言

1. C 运行时库

一种被编译器用来实现编程语言内置函数的程序库

提供 memcpy, printf, malloc 等

为应用程序提供启动函数

(加载 main 函数)

2. 宏和函数的区别

1) 宏是在 C 程序中的定义的命名代码段

2) 函数代码只要编译一次，宏在任何调用的地方都要展开，编译一次

3> 宏使用时不用保护现场，也不以返回

4> 代码简单用宏，复杂用函数

define 用于定义全局变量、常量和宏

3. C语言内嵌 ARM 汇编

-asm

↓

一般先把相关参数导入

指令序列

}

4. -irq 使用 -irq 声明的断点被用作

irq 或 firq 异常中断向量的中断处理函数

5. volatile

> 在汇编语句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

3> 不能对 volatile 变量使用缓冲技术。

6. ARM 汇编与 C 的混合编程

↑ 传递过程：R0 存第一个参数，R1 存第二个参数

C 调用汇编

↑ 传递过程：要调用的参数依次存在 R0, R1 ...
汇编调用 C

6. ARM 异常处理过程：

- ▷ 拷贝完当前指令，跳转到异常服务程序
执行
- ▷ 执行完成后，返回到下一条指令执行
- ▷ 进入 IRQ 时要保护现场，返回时要恢复现场

7. 异常向量表

存放中断服务程序的地址
(中断向量地址)

Part 4. 其他

1. RISC-V的主要特征：

- 1) 开源
- 2) 重新设计(后发优势)
- 3) 简单的美学
- 4) 模块化的指令集
- 5) 指令集可扩展

2. 龙蜥鸿920使用的指令集架构是 ARMv8 64位

7nm 制程

相比916，920的计算核数提升1倍，最高支持64核

3. 嵌入式最小系统构成

包括电源、时钟、复位、存储器

· 电源的稳定性直接影响系统的稳定性。

· 复位端是一个时序电路，需要一个时钟信号才能工作。

· 微控制器上电时状态不确定，因此需要复位逻辑初始化。

· 存储器有闪存、双倍、快闪等。

4. DC-DC 变换转换器、线性稳压器、开关稳压器和光耦合器

5. 晶体与晶振的区别

晶体：封装内部只有晶体，驱动电路由设计者提供

晶振：封装中包含了完整的晶体振荡器电路，内部有源。

6. 复位的基本功能：

- 上电时初始化状态不锁定，复位逻辑初始化
- 手动复位也要支持

7. 存储器

SRAM：电源不及，一直不变

DRAM：需要不断刷新，DRAM耗电多 存储器

Nor Flash：断电保存，芯片功耗低，读快，写入耗电

Nand Flash：写快，存数据 | 擦除

Part 5 嵌入式操作系统

一、嵌入式操作系统简介

1. 定义：应用于嵌入式系统的 OS，有方便应用程序使用的系统调用接口（API）

2. 分类

实时性（RTOS）：
| 硬实时 OS：Vxworks, uC/OS ✓
| 软实时 OS：WinCE, 嵌入式 Linux

3. 功能

1> OS 内核基本功能：

多任务、中断、内存、I/O 管理等

2> 嵌入式 TCP/IP 网络系统

3> 嵌入式文件系统

4. 特点，确定性：可确定的最坏时间

实时性 — 嵌入式 OS 的核心，响应时间在 ms 级

5. 多任务程序设计结构 — 前后台结构

后台循环，前台中断



中断请求

实时性强

6. 任务的相关概念

进程：资源分配 线程：调度
任务：即线程

* 任务的要素：代码、数据、堆栈，任务控制块(TCB)

任务的特性：动态性、并发性、异步独立性。

* 任务的基本状态

运行态：CPU上 阻塞态：^{↑ 包括挂起态，等待态} 缺资源、数据

就绪态：万事俱备只欠CPU.

* 任务状态转换

就绪 → 运行：选择就绪队头元素上CPU

运行 → 就绪：用完时间片 / 任务被中断

运行 → 等待(忙)：少年西：OS未完成服务，I/O结果没来
与CPU无关

等待 → 就绪：等待的事件已经发生(猝然来了)

* 任务切换

· 保存当前任务状态(压栈)

· 恢复需运行任务(出栈)

* 调度：确定任务执行的顺序，任务在CPU运行时间

· 调度发生的位置：调度点

包括：中断服务程序的结束位置 } 任务离开CPU

任务因申请资源变为等待态

* 任务优先级：系统总是让处于就绪态，且优先级最高的任务运行。
 └ 分类 { 静态优先级：任务执行过程优先级不变
 动态优先级：优先级可变

* 调度策略：非抢占式、抢占式

非抢占式：| 调度点：正在被执行的任务被阻塞
 | 中断结束仍要使原CPU任务执行完(非调度点)

抢占式：| 调度点1：中断结束
 | 调度点2：CPU任务被阻塞

7. 可重入型函数

任何时候都可以被中断切换，后续运行不受影响。
这类函数只使用局部变量（不使用全局变量）

8. 暂界区（暂界代码段）

一旦开始执行，不允许任何中断打入。

保护机制：进入前关中断；结束后开中断；

* 先中断时间要尽量短，否则可能会中断丢失。

9. 共享资源

资源：任何可被任务占用的实体，可是 I/O 设备硬件，
也可是变量、数组等（均属）

共享资源：被多个任务访问

互斥访问的方法：关中断，禁止任务切换

使用测试并置位指令，使用信号量 ✓ [最优]

二、μC/OS-II 的内核结构

1. 事件和任务

事件：可以理解为信号，由中断服务程序发

任务：一个无限循环，与函数类似，但无返回值

* μC/OS-II 的任务：

2.5 版本 支持最多 64 个任务，都有优先级

* μC/OS-II 的任务状态

任务位前的状态

运行态，就绪态，挂起态，中断服务态，休眠态

* TCB (任务控制块)

定义：管理任务的数据结构，包括堆栈指针、状态、优先级等

操作系统初始化

所有任务控制块在 μC/OS 初始化时生成

包括空闲 TCB 链表和 使用 TCB 链表 (也叫任务链表)

* 任务队列

就绪 → 就绪队列

挂起 → 等待队列

2. 优先级位图算法 — 就绪表

OS Rdy Grp：某一位为 1 的条件：对应位图该行还有 1

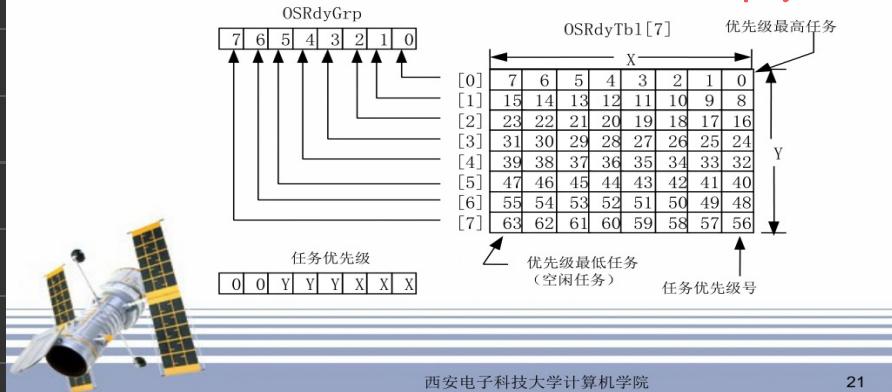
OS RdyTbl[]：二维矩阵，该优先级任务就绪，则该位为 1



优先级位图算法

每个就绪的任务都放入就绪表中（ready list）中，就绪表有两个变量：
OSRdyGrp、OSRdyTbl[]

高优先级对应小号



西安电子科技大学计算机学院

21

示例1：优先级为21的任务就绪：

$21_d = 0|0|0|1|b$ “|”按位或
行号 —— 列号

OSRdyGrp 第2位置1，OSRdyTbl[7]第5位置1

*优先级映射表

DSMapTbl[8]存放从 00000001 至 10000000 便于或运算

示例2：任务进入就绪态代码：(操作1)

$OSRdyGrp |= DSMapTbl[prio >> 3]$

$OSRdyTbl[prio >> 3] = DSMapTbl[prio \& 0x07]$

示例3：任务脱离就绪表（操作2）
if ([OSRdyTbl [pri0>>3] &=~ OSMapTbl [pri0&0x07])
== 0)

OSRdyGrp &= ~OSMapTbl [pri0>>3];

示例4：根据就绪表确定最高优先级：（操作3）
若 OSRdyGrp 值为 0x24 = 100100b，则最高优先级出现在位图第 2 行；

通过 OSRdyTbl [2] 的值来确定低三位，假设为
0x12 = 00010010， 则最高优先级在第 1 列；

则最高优先级为 010001b = 17

* 源代码使用了查表法

优点：降低时间复杂度，有确定时间，可预测性增加

优先级判定表 OSUnMapTbl [256]

3. 任务调度

· 这是选择优先级最高任务上 CPU 任务被阻塞时

· 由调度器完成

任务级调度：OSScheduler() ————— 包括 { OS启动时
中断级调度：OSIntExit() } 新任务创建后
新任务就绪后

OS_TASK_SW() 是宏调用，用于任务级切换

中断级切换：OSIntExit()

该函数将中断嵌套计数器 OSIntNesting
减1，减到0时，进行一次任务切换

4. 闹钟节拍

一种定时器中断，实时OS 的心脏；

每很短的时间响应一次，进行一次任务切换

5. 开中断与关中断

OS-ENTER-CRITICAL

关中断

OS-EXIT-CRITICAL

开中断

可用于保护临界代码区，不被中断打断

中断不参与任何任务处理相关机制

6. uCOS 启动

OSInit() → OSTaskCreate() → OSSstart()

初始化

创建任务

OS启动

三、uCOS-II 的任务管理

1. 任务的格式：无限循环

2. OSTaskCreate(task, pdata, pOS, prio)

指向任务代码

↓

↓

任务创建 → 任务指针 参数指针 栈顶 优先级
指针

基本步骤

参数检查 → 优先级是否被占用 → 初始化堆
占用该优先级 初始化任务控制块 →
给任务计数 → 如果系统已经开始运行 → 创建关联
器加 1 则进行任务调度 释放优先级

3. 删除任务 → 休眠态

OSTaskDel (INT8U prio); 任务可以删除自己
4. 改变任务优先级 → OS_TaskChange_Priority!
5. 挂起/恢复任务

OS_TaskSuspend (prio);

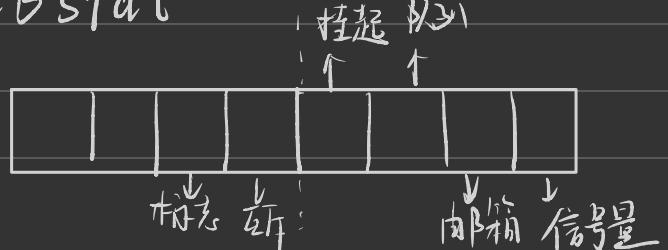
任务可以挂起自己，但不能挂起空闲任务

检查任务是否 → 从就绪表 → 任务状态置为
自己存在 中删除 OS_STAT_SUSPEND

如果挂起 → 当前任务自己 → 则要进行
任务调度

*任务的新状态 OS_TCB_Status

被挂起，则挂起位置 |



OS Task Resume (prio);

任务是否存在 → 清除
是否被挂起 → 挂起标志 → 检查是否有其他无法就绪的原因

OS_STAT_SUSPEND

→ 没有则
任务就绪 → 进记任务
调度

6. 获取任务的信息 有到这生

OSTaskQuery (prio, pdata)

7. 时间管理 十个滴答数

时钟节拍频率 (10-100 次/秒) ↑
由 OSTimeTick() 实现定时中断，对应 1 次

OSTimeDly (ticks)

功能：延时 ticks 个滴答数
步骤：

阻塞当前 → ticks 保存在 OS tcb 中 → 产生一次
任务 调度



OSTimeTick() → 减到 0

每次定时中断来 可唤醒就绪
OSTCBDly - 1

OSTime 记录系统启动以来的滴答数

四. UCOS-II 的任务通信机制

1. 信号量

$S > 0$, 表示资源实体数

$S \leq 0$, 表示等待资源的进程数

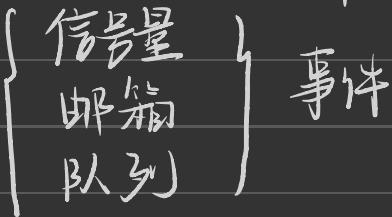
P(S): $S--$; $S > 0$, 进程执行, 否则进入等待队列

V(S): $S++$; 释放资源, 唤醒等待队列的一个进程

* 实现互斥: S 初值为 1 | * 实现同步:

P(S);	empty: 初值 n
临界区 资源	full: 初值 0
V(S);	mutex1: 1
	mutex2: 1

2. UCOS-II 的通信 - 事件

通信 (信道间) 

* 事件 等待任务列表

事件控制块中有 OSEventGrp 和 OSEventTbl
共同构成任务等待列表。

同样有 3 种操作 (类似于就绪表)

* 空余事件 按制块：OS 初始化时产生
* 基本操作：

OS_EventWaitListInit(OS_EVENT *pEvent)

初始化 等待事件优先级表

OS_EventTaskWait();

使当前任务进入等待某事件的状态

OS_EventTaskReady();

使一个任务就绪

OS_EventTD();

因为等待超时而就绪。

3. uCOS-II 中的信号量

16bit Unsigned int 计数值

等待该信号量的等待任务列表

* 建立一个信号量

OS_EVENT OSSemCreate(INT16 Cnt)

Cnt 为信号量初值，会存到 OSEventCnt 中

从空余事件按制块 → 事件类型设 → 写入初 → 调用

链表获得一个块 为信号量 值 ↑ 初始化

与传统信号量定义略有不同，就是

它的值只能为无符号整型，不能为负

* 删除一个信号量 事件控制块头指针
OSSemDel (pevent, opt, err)
↓
删除方式

* 等待一个信号量 — P 操作

OSSemPend (pevent, timeout, err)

信号量 > 0 → OSIntNesting > 0 → = 0, 信号量无效,
OSEventCnt-- 判断是否在中断和语义 使任务进入该信号量的
不能在 ISR 中调用 等待状态

与传统 PV

操作不完全一样

| 设置任务控制块中的状态位 OSTCBStat
| 设置超时时间 OSTCBLdy = timeout
| 使任务等待事件 OSEventTaskWait (pevent)
| 进行调度调度
↑ 包括

OSSemPost (pevent) — V 操作

查看任务等待表 → 若有，则优先级 若没有
是否有任务等待 → 最高任务就绪 → OSEventCnt++
该信号量

* 申请请求一个信号量

OSSemAccept (pevent)：不会进入等待状态，可在

↑(post也可以，其他不行)

中断服务程序中调用

*查询信号量当前状态

存储状态结果



OSSemQuery (pevent, pdatta)

五、uclos-TI的其他任务通信机制

1. 优先级反转

多个任务需要共享资源的情况下，出现了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；这种情况下可能出现许多高优先级任务抢占到CPU，高优先级优先级无法执行。

解决方案：优先级继承协议和优先级天花板

UCLOS-TI 结合了上述两种方法



PIP

*互斥型信号量 mutex：二值信号量

用于实现可能产生优先级反转问题

*建立一个互斥型信号量 → 传入PIP优先级

OSMutexCreate [prio, *err]

原来的 OSEventCnt：

高8位：PIP（优先级天花板）

低8位：如果有交叉，则为0xFFFF

如果无效10），存占用该信号量的优先级

* 等待一个互斥型信号量

DSMutex Pend (pEvent, timeout, err)

* 释放一个互斥型信号量

DSMutex Post (pEvent)

不行

Mutex 不能用于中断服务程序 [所有相关函数都]

3. 其他机制

· 消息邮箱：不存计数值，存消息指针

· 消息队列：存多个消息指针

· 事件标志组

