## 实时系统概念

### 2.23 消息邮箱

典型的消息邮箱也称作交换消息，是用一个指针型变量，通过内核服务，一个任务或一个中断服务程序可以把一则消息（即一个指针）放到邮箱里去。

每个邮箱有响应的正在等待消息的任务列表，要得到消息的任务因为邮箱是空的而被挂起，且被记录到等待消息的任务表中，直到收到消息。

内核一般提供以下邮箱服务：

1. 邮箱内消息的内容初始化，邮箱里最初可以有，也可以没有消息
2. 将消息放入邮箱（POST）
3. 等待有消息进入邮箱（PEND）
4. 如果邮箱内有消息，就接收这则消息。如果邮箱里没有消息，则任务并不被挂起（ACCEPT），返回代码表示调用结果，是收到了消息还是没有收到消息。

### 2.24 消息队列

消息队列实际上是邮箱阵列。

典型的内核提供的消息队列服务如下：

1. 消息队列初始化。队列初始化时总是清为空。
2. 放一则消息到队列中去（Post）
3. 等待一则消息的到来（Pend）
4. 如果队列中有消息则任务可以得到消息，但如果此时队列为空，内核并不将该任务挂起（Accept）。如果有消息，则消息从队列中取走。没有消息则用特别的返回代码通知调用者，队列没有消息。

### 2.25 中断

1、中断是一种硬件机制。

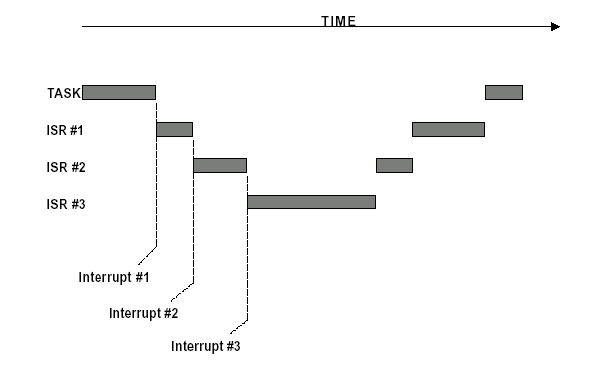
2、关中断和开中断可以让微处理器不响应或响应中断。

3、关中断时间太长可能引起中断丢失。

### 2.26 中断延迟

1、所有实时系统在进入临界区代码段之前都要关中断。

2、中断延迟 = 关中断的最长时间+开始执行中断服务子程序的第一条指令的时间



### 2.31 非屏蔽中断（NMI）

### 2.32 时钟节拍（Clock Tick）

时钟节拍是特定的周期性中断。时钟节拍率越快，系统的额外开销就越大。

## 内核结构

### 3.0 临界段（Critical Section）

1）每种微处理器都有自己的OS\_CPU.H文件。

2）文件中包含以下两个宏：

OS\_ENTER\_CRITICAL() -- 开中断

OS\_EXIT\_CRITICAL() -- 关中断

### 3.1 任务

一个任务通常是一个无限的循环。任务是绝不会返回的。故返回参数必须定义成void。

void YourTask(void \*pdata)

{

for(;;){

/\*用户代码\*/

调用uc/os-II的某种系统服务：

OSMboxPend();

OSQPend();

OSSemPend();

OSTaskDel(OS\_PRIO\_SELF);

OSTimeDly();

OSTimeDlyHMSM();

/\*用户代码\*/

}

}

### 3.2 任务状态

### 3.3任务控制块（Task Control Blocks， OS\_TCBs）

1. OS\_TCBs全部驻留在RAM中。

### 3.4 就绪表

1、就绪表中有两个变量OSReadyGrp和OSRRdyTbl[]

### 3.5 任务调度

1、OSShed()完成任务级的调度。OSIntExt()完成中断级的调度。

2、任务调度器函数

Void OSSched(void)

{

INT8U y;

OS\_ENTER\_CRITICAL();

If ((OSLockNesting | OSIntNesting) == 0 ) {

y = OSUnMaoTbl[OSRdyGrp];

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

If (OSPrioHighRdy != OSPrioCur){

OSTCBHighRdy = OSTCBPrioTbl[OSPrioHighRdy];

OSCtxSwCtr++;

OS\_TASK\_SW();

}

}

OS\_EXIT\_CRITICAL();

}

1. Uc/os-II任务调度所花的时间是常数，与应用程序中的任务个数没有关系。
2. 如果在中断服务子程序中调用OSSched()，此时中断嵌套成熟OSIntNesting>0，或者由于用户至少调用了一次给任务调度上锁函数OSSchedLock(),使OSLockNesting>0。退出调度。
3. 最后OS\_TASK\_SW()来完成实际上的任务切换。

### 3.6 给调度器上锁和开锁

1、OSSchedlock()：给调度器上锁。

OSSchedUnlock()：给调度器开锁。

1. 在调度器上锁过程中，中断仍然可以被识别。
2. OSLockNesting在上锁是加1，在开锁时减1。

### 3.7 空闲任务

空闲任务的优先级最低（OS\_LOWEST\_PRIO），空闲任务OSTaskIdle()什么都不做，只是不停的给一个32位的变量OSIdleCtr的计数器加1，统计任务使用这个计数器以确定先行应用软件实际消耗的CPU时间。空闲任务不可能被应用软件删除。

Void OSTaskIdle(void \*pdata)

{

Pdata = pdata;

For(;;)

{

OS\_ENTER\_CRITICAL();

OSIdleCtr++;

OS\_EXIT\_CRITICAL();

｝

｝

### 3.8 统计任务

### 3.11 uc/os-II初始化

1、首先调用系统始化函数OSIint()，初始化所有的变量和数据结构。

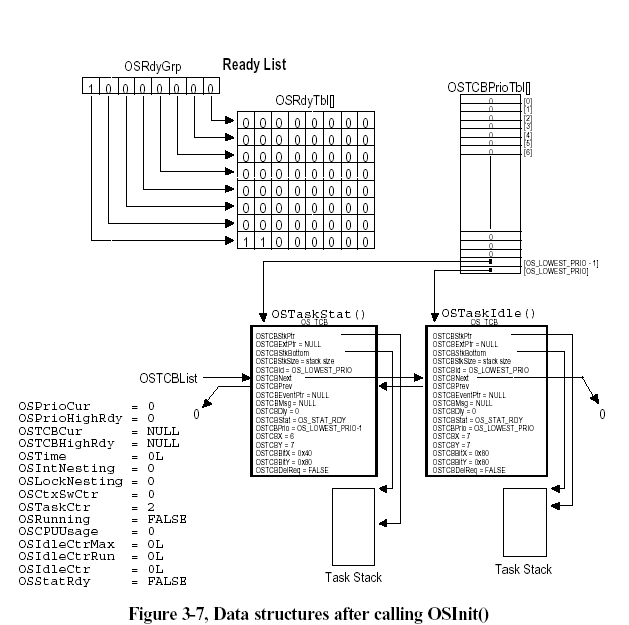
2、如果OS\_TASK\_STAT\_EN和任务建立扩展允许都为1，则OSInit()还得建立统计任务OSTaskStat()并且让其进入就绪态。

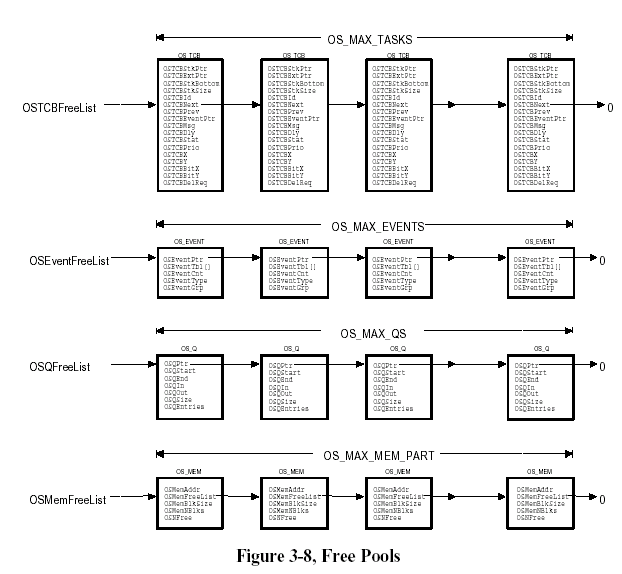
3、uc/os-II还初始化了4个空数据结构缓冲区。

### 3.12 uc/os-II的启动

1、多任务的启动是用户通过调用OSStart()实现的。在启动之前，用户必须要建立一个应用任务。

|  |
| --- |
| 程序清单 L3.24 初始化和启动μC/OS-Ⅱ |
| void main (void) |
| { |
| OSInit(); /\* 初始化uC/OS-II \*/ |
| . |
| . |
| 通过调用OSTaskCreate()或OSTaskCreateExt()创建至少一个任务; |
| . |
| . |
| OSStart(); /\* 开始多任务调度!OSStart()永远不会返回 \*/ |
| } |





2、OSStart()在任务就绪表中找出那个永固建立的优先级最高任务控制块，然后调用优先级就绪任务启动函数OSStartHighRdy()，将任务栈中保存的值弹回到CPU寄存器中，然后执行一条终端返回指令。

|  |
| --- |
| 程序清单 L3.25 启动多任务. |
| void OSStart (void) |
| { |
| INT8U y; |
| INT8U x; |
|  |
| If(OSRunning ==FALSE){ |
| Y = OSUbMapTbl[OSRdyGrp]; |
| X = OSUnMapTbl[OSRdyTbl[y]]; |
| OSPrioHighRdy = (INY8U)((y<<3)+x);  OSPrioCur = OSPrioHighRdy; |
| OSTCBHighRdy = OSTCBPrioTbl[OSPrioHighRdy]; |
| OSTCBCur = OSTCBHoghRdy; |
| OSStartHighRdy(); |
| } |
|  |
| } |

### 3.13 获取当前uc/os-II的版本号

调用OSVersion()可以得到当前uc/os-II的版本号。返回值=版本号\*100.

### 3.14 Event函数

OSEventWaitListInit()

OSEventTaskRdt()

OSEventTaskWait()

OSEventTO()

这几个函数放在文件OS\_CORE.C中。

## 任务管理

### 4.0 建立任务，OSTaskCreate()

OSTaskCraete()需要四个参数：task是任务代码的指针，pdata是当任务开始执行时传递给任务的参数的指针，pots是分配给任务的堆栈的栈顶指针，prio是分配给任务的优先级。

INT8U OSTaskCreate(void (\*task) (void \*pd), void \*pdata, OS\_STK \*ptos, INT8U prio)

{

Void \*psp;

INT8U err;

If(

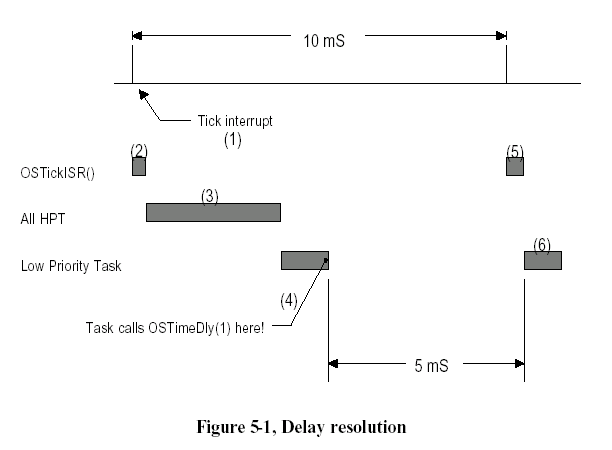
## 时间管理

### 5.0 任务延时函数，OSTimeDly()

调用该函数uc/os-II会进行一次任务调度，该任务一旦规定的时间期满或者有其他的任务调用OSTimeDlyResume()取消延时，它就马上进入就绪状态。

该函数的参数代表时钟节拍数（1-65535），这个延时节拍被保存在当前任务的OS\_TCB中，并通过OSTimeTick()每隔一个时钟节拍就减少一个计数。

|  |
| --- |
| 程序清单 L 5.1 ***OSTimeDly()***. |
| void OSTimeDly (INT16U ticks) |
| { |
| if (ticks > 0) { (1) |
| OS\_ENTER\_CRITICAL(); |
| if ((OSRdyTbl[OSTCBCur->OSTCBY] &= ~OSTCBCur->OSTCBBitX) == 0) { (2) |
| OSRdyGrp &= ~OSTCBCur->OSTCBBitY; |
| } |
| OSTCBCur->OSTCBDly = ticks; (3) |
| OS\_EXIT\_CRITICAL(); |
| OSSched(); (4) |
| } |
| } |



对于多任务，不能确保设定的时钟节拍数=程序等待的真实节拍数。

### 5.1 按时分秒延时函数 OSTimeDlyHMSM()

真正有意义的只是OS\_TICKS\_PER\_SEC，决定了最接近需要延时的时间的时钟节拍总数。500/OS\_TICKS\_PER\_SECOND的值基本上与0.5个节拍对应的毫秒数相同。

|  |
| --- |
| 程序清单 L 5.2 ***OSTimeDlyHMSM()***. |
| INT8U OSTimeDlyHMSM (INT8U hours, INT8U minutes, INT8U seconds, INT16U milli) |
| { |
| INT32U ticks; |
| INT16U loops; |
|  |
|  |
| if (hours > 0 || minutes > 0 || seconds > 0 || milli > 0) { (1) |
| if (minutes > 59) { |
| return (OS\_TIME\_INVALID\_MINUTES); |
| } |
| if (seconds > 59) { |
| return (OS\_TIME\_INVALID\_SECONDS); |
| } |
| If (milli > 999) { |
| return (OS\_TIME\_INVALID\_MILLI); |
| } |
| ticks = (INT32U)hours \* 3600L \* OS\_TICKS\_PER\_SEC (2) |
| + (INT32U)minutes \* 60L \* OS\_TICKS\_PER\_SEC |
| + (INT32U)seconds \* OS\_TICKS\_PER\_SEC |
| + OS\_TICKS\_PER\_SEC \* ((INT32U)milli  + 500L/OS\_TICKS\_PER\_SEC) / 1000L; (3) |
| loops = ticks / 65536L; (4) |
| ticks = ticks % 65536L; (5) |
| OSTimeDly(ticks); (6) |
| while (loops > 0) { (7) |
| OSTimeDly(32768); (8) |
| OSTimeDly(32768); |
| loops--; |
| } |
| return (OS\_NO\_ERR); |
| } else { |
| return (OS\_TIME\_ZERO\_DLY); (9) |
| } |
| } |

### 5.2 让处在延时期的任务结束延时，OSTimeDlyResume()

SOTimeDlyResume()可以唤醒正在等待的事件。

|  |
| --- |
| 程序清单 L 5.3 恢复正在延时的任务 |
| INT8U OSTimeDlyResume (INT8U prio) |
| { |
| OS\_TCB \*ptcb; |
|  |
| if (prio >= OS\_LOWEST\_PRIO) { (1) |
| return (OS\_PRIO\_INVALID); |
| } |
| OS\_ENTER\_CRITICAL(); |
| ptcb = (OS\_TCB \*)OSTCBPrioTbl[prio]; |
| if (ptcb != (OS\_TCB \*)0) { (2) |
| if (ptcb->OSTCBDly != 0) { (3) |
| ptcb->OSTCBDly = 0; (4) |
| if (!(ptcb->OSTCBStat & OS\_STAT\_SUSPEND)) { (5) |
| OSRdyGrp |= ptcb->OSTCBBitY; (6) |
| OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX; |
| OS\_EXIT\_CRITICAL(); |
| OSSched(); (7) |
| } 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); |
| } |
| } |

### 5.3 系统时间，OSTimeGet()和OSTimeSet()

Uc/os-II会对时钟节拍进行累加，保存在一个32位的计数器中，调用OSStart()初始化多任务和4294967295个节拍执行完一遍时从0开始计数。

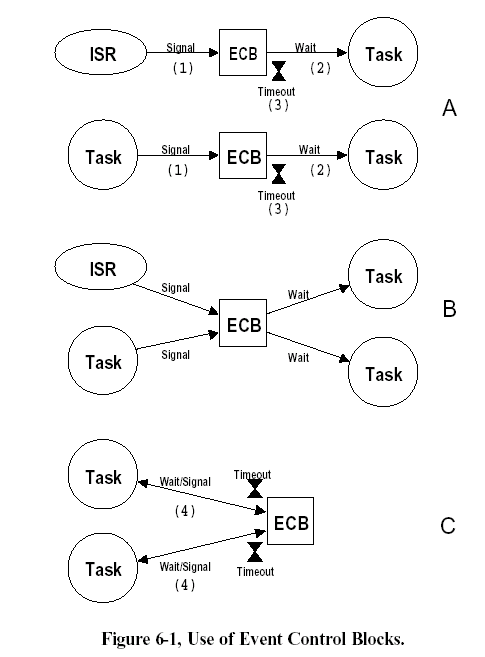
在访问OSTime的时候中断是关掉的。

|  |
| --- |
| 程序清单 L 5.4 得到和改变系统时间 |
| INT32U OSTimeGet (void) |
| { |
| INT32U ticks; |
|  |
| OS\_ENTER\_CRITICAL(); |
| ticks = OSTime; |
| OS\_EXIT\_CRITICAL(); |
| return (ticks); |
| } |
|  |
| void OSTimeSet (INT32U ticks) |
| { |
| OS\_ENTER\_CRITICAL(); |
| OSTime = ticks; |
| OS\_EXIT\_CRITICAL(); |
| } |

## 任务之间的通讯与同步

只有任务可以等待事件发生，终端服务子程序是不能这样做的。

一个任务或中断服务可以通过事件控制块ECB(Event Control Blocks)来向另外的任务发信号。所有的信号被看成是事件。



### 6.1事件控制块ECB

|  |
| --- |
| 程序清单 L6.1 ECB数据结构 |
| 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; |

.OSEventPtr指针:只有在所定义的事件是邮箱或消息队列时才使用。

.OSEventTbl[]和.OSEventGrp：记录事件的优先级

|  |
| --- |
| 程序清单 L6.2——将一个任务插入到事件的等待任务列表中 |
| pevent->OSEventGrp |= OSMapTbl[prio >> 3]; |
| pevent->OSEventTbl[prio >> 3] |= OSMapTbl[prio & 0x07]; |

#### 6.1.1 初始化一个事件控制块，OSEventWaitListInit()

当建立信号量、邮箱或者消息队列，相应的建立函数OSSemInit()，OSMboxCreate()，OSQCreate()，都是通过调用OSEventWaitListInit（）对事件控制块的等待任务列表进行初始化。

|  |
| --- |
| 程序清单 L6.5 初始化ECB块的等待任务列表 |
| void OSEventWaitListInit (OS\_EVENT \*pevent) |
| { |
| INT8U i; |
|  |
|  |
| pevent->OSEventGrp = 0x00; |
| for (i = 0; i < OS\_EVENT\_TBL\_SIZE; i++) { |
| pevent->OSEventTbl[i] = 0x00; |
| } |
| } |

#### 6.1.2 使一个任务进入就绪态，OSEventTaskRdy()

#### 6.1.3 使一个任务进入等待某事件发生状态，OSEventTaskWait()

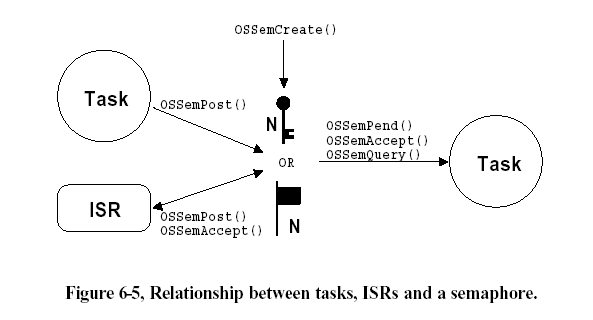
#### 6.1.4 由于等待超时而将任务置为就绪态，OSEventTO()

### 6.2 信号量

信号量的使用，用户要在OS\_CFG.H中将OS\_SEM\_EN开关量常数进行使能（=1），这样uc/os-II才支持信号量。

信号量操作用到的函数：OSSemCreate(),OSSemPend(), OSSemPost(), OSSemAccept()和OSSemQuery()函数。

信号量可以表示一个或多个事件的发生，那么该信号量的初始值应设为0；如果是用于对共享资源的访问，

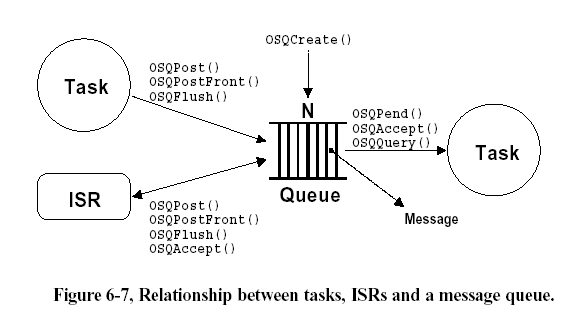


#### 6.2.1 建立一个信号量，OSSemCreate()

### 6.4 消息队列

需要在OS\_CFG.H文件中，将OS\_Q\_EN常数设置为1，并通过常数OS\_MAX\_QS来决定uc/os-II支持的最多消息队列数。

7个操作函数：OSQCreate(), OSQPend(), OSQPost(), OSQPostFront(), OSQAccept(), OSQFlush()和OSQQuery()函数。



N代表了消息队列中的总单元数，当调用OSQPend()或者OSQAccept()之前，调用N次OSQPost()或者OSQPostFront（）就会吧消息队列填满。

一个任务或者终端服务子程序可以调用OSQPost（），OSQPostFront（），OSQFlush（）或者OSQAccept（）函数。但是只有任务可以调用OSQPend（）和OSQQuery（）函数。

## 内存管理

在一个系统中可以有很多个内存分区，这些内存分区提前定义好，这叫做动态内存池。

### 7.1 内存控制块

|  |
| --- |
| 程序清单 L7.1 内存控制块的数据结构 |
| typedef struct { |
| void \*OSMemAddr; |
| void \*OSMemFreeList; |
| INT32U OSMemBlkSize; |
| INT32U OSMemNBlks; |
| INT32U OSMemNFree; |
| } OS\_MEM; |

.OSMemAddr：是指向内存分区起始地址的指针。在建立内存分区时被初始化，在此之后不能更改。

.OSMemFreeList：是指向下一个空闲内存控制块或者下一个空闲的内存块的指针，具体含义要根据该内存分区是否已经建立来决定。

.OSMemBlkSize：是内存分区中内存块的大小，是用户建立该内存分区时指定的。

.OSMenNBlks：是内存分区中总的内存块数量，也是用户建立该内存分区时指定的。

.OSMemNFree：是内存分区中当前可以用的空闲内存块数量。

如果要在uc/os-II中使用内存管理，需要在OS\_CFG.H文件中将开关OS\_MEM\_EN设置为1。这样OSInit()调用OSMemInit()实现。

### 7.2 建立一个内存分区，OSMemCreate()

在使用一个内存分区之前，必须先建立该内存分区。

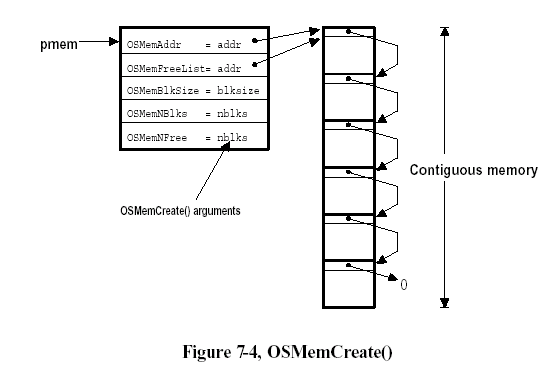
创建一个含有100个内存块，每个内存块32字节的内存分区。

|  |
| --- |
| 程序清单 L7.2 建立一个内存分区 |
| OS\_MEM \*CommTxBuf; |
| INT8U CommTxPart[100][32]; |
|  |
|  |
| void main (void) |
| { |
| INT8U err; |
|  |
|  |
| OSInit(); |
| . |
| . |
| CommTxBuf = OSMemCreate(CommTxPart, 100, 32, &err); |
| . |
| . |
| OSStart(); |
| } |

每个内存分区必须含有至少两个内存块，每个内存块至少为一个指针大小。

|  |
| --- |
| 程序清单 L7.3 OSMemCreate() |
| OS\_MEM \*OSMemCreate (void \*addr, INT32U nblks, INT32U blksize, INT8U \*err) |
| { |
| OS\_MEM \*pmem; |
| INT8U \*pblk; |
| void \*\*plink; |
| INT32U i; |
|  |
|  |
| if (nblks < 2) { (1) |
| \*err = OS\_MEM\_INVALID\_BLKS; |
| return ((OS\_MEM \*)0); |
| } |
| if (blksize < sizeof(void \*)) { (2) |
| \*err = OS\_MEM\_INVALID\_SIZE; |
| return ((OS\_MEM \*)0); |
| } |
| OS\_ENTER\_CRITICAL(); |
| pmem = OSMemFreeList; (3) |
| if (OSMemFreeList != (OS\_MEM \*)0) { |
| OSMemFreeList = (OS\_MEM \*)OSMemFreeList->OSMemFreeList; |
| } |
| OS\_EXIT\_CRITICAL(); |
| if (pmem == (OS\_MEM \*)0) { (4) |
| \*err = OS\_MEM\_INVALID\_PART; |
| return ((OS\_MEM \*)0); |
| } |
| plink = (void \*\*)addr; (5) |
| pblk = (INT8U \*)addr + blksize; |
| for (i = 0; i < (nblks - 1); i++) { |
| \*plink = (void \*)pblk; |
| plink = (void \*\*)pblk; |
| pblk = pblk + blksize; |
| } |
| \*plink = (void \*)0; |
| OS\_ENTER\_CRITICAL(); |
| pmem->OSMemAddr = addr; (6) |
| pmem->OSMemFreeList = addr; |
| pmem->OSMemNFree = nblks; |
| pmem->OSMemNBlks = nblks; |
| pmem->OSMemBlkSize = blksize; |
| OS\_EXIT\_CRITICAL(); |
| \*err = OS\_NO\_ERR; |
| return (pmem); (7) |
| } |

#### 7.3 分配一个内存块，OSMenGet()



|  |
| --- |
| 程序清单 L7.5 OSMemPut() |
| INT8U OSMemPut (OS\_MEM \*pmem, void \*pblk) (1) |
| { |
| OS\_ENTER\_CRITICAL(); |
| if (pmem->OSMemNFree >= pmem->OSMemNBlks) { (2) |
| OS\_EXIT\_CRITICAL(); |
| return (OS\_MEM\_FULL); |
| } |
| \*(void \*\*)pblk = pmem->OSMemFreeList; (3) |
| pmem->OSMemFreeList = pblk; |
| pmem->OSMemNFree++; (4) |
| OS\_EXIT\_CRITICAL(); |
| return (OS\_NO\_ERR); |
| } |

#### 7.4 释放一个内存块，OSMemPut()

因为内存块释放时，不知道释放到哪个内存块，所以需要手工引导释放的目的地，函数的第一个参数是指向内存控制块的指针。

|  |
| --- |
| 程序清单 L7.5 OSMemPut() |
| INT8U OSMemPut (OS\_MEM \*pmem, void \*pblk) (1) |
| { |
| OS\_ENTER\_CRITICAL(); |
| if (pmem->OSMemNFree >= pmem->OSMemNBlks) { (2) |
| OS\_EXIT\_CRITICAL(); |
| return (OS\_MEM\_FULL); |
| } |
| \*(void \*\*)pblk = pmem->OSMemFreeList; (3) |
| pmem->OSMemFreeList = pblk; |
| pmem->OSMemNFree++; (4) |
| OS\_EXIT\_CRITICAL(); |
| return (OS\_NO\_ERR); |
| } |

#### 7.5 查询一个内存分区的状态，OSMenQuery()

函数返回内存分区中内存块的大小，可用内存块的数量和正在使用的内存块数等相关信息。这些信息包含在OS\_MEM\_DATA结构体中。

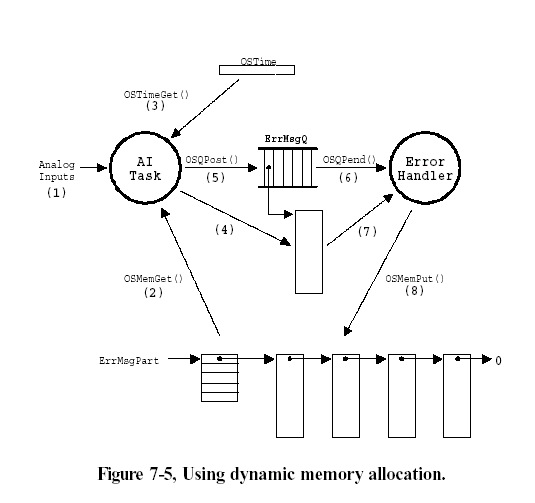
|  |
| --- |
| 程序清单 L7.6 OS\_MEM\_DATA数据结构 |
| typedef struct { |
| void \*OSAddr; /\* 指向内存分区首地址的指针 \*/ |
| void \*OSFreeList; /\* 指向空闲内存块链表首地址的指针 \*/ |
| INT32U OSBlkSize; /\* 每个内存块所含的字节数 \*/ |
| INT32U OSNBlks; /\* 内存分区总的内存块数 \*/ |
| INT32U OSNFree; /\* 空闲内存块总数 \*/ |
| INT32U OSNUsed; /\* 正在使用的内存块总数 \*/ |
| } OS\_MEM\_DATA; |

|  |
| --- |
| 程序清单 L7.7 OSMemQuery() |
| INT8U OSMemQuery (OS\_MEM \*pmem, OS\_MEM\_DATA \*pdata) |
| { |
| OS\_ENTER\_CRITICAL(); |
| pdata->OSAddr = pmem->OSMemAddr; (1) |
| pdata->OSFreeList = pmem->OSMemFreeList; |
| pdata->OSBlkSize = pmem->OSMemBlkSize; |
| pdata->OSNBlks = pmem->OSMemNBlks; |
| pdata->OSNFree = pmem->OSMemNFree; |
| OS\_EXIT\_CRITICAL(); |
| pdata->OSNUsed = pdata->OSNBlks - pdata->OSNFree; (2) |
| return (OS\_NO\_ERR); |
| } |

### 7.6 案例

第一个任务读取并检查模拟输入量的值（如气压、温度、电压等），如果其超过了一定的阈值，就向第二个任务发送一个消息。该消息中含有时间信息、出错的通道号和错误代码等可以想象的任何可能的信息。

错误处理程序是该例子的中心。任何任务、中断服务子程序都可以向该任务发送出错消息。错误处理程序则负责在显示设备上显示出错信息，在磁盘上登记出错记录，或者启动另一个任务对错误进行纠正等。



|  |
| --- |
| 程序清单 L7.8 内存分配的例子——扫描模拟量的输入和报告出错 |
| AnalogInputTask() |
| { |
| for (;;) { |
| for (所有的模拟量都有输入) { |
| 读入模拟量输入值; (1) |
| if (模拟量超过阈值) { |
| 得到一个内存块; (2) |
| 得到当前系统时间 (以时钟节拍为单位); (3) |
| 将下列各项存入内存块: (4) |
| 系统时间 (时间戳); |
| 超过阈值的通道号; |
| 错误代码; |
| 错误等级; |
| 等. |
| 向错误队列发送错误消息; (5) |
| (一个指向包含上述各项的内存块的指针) |
| } |
| } |
| 延时任务,直到要再次对模拟量进行采样时为止; |
| } |
| } |
|  |
|  |
|  |
| ErrorHandlerTask() |
| { |
| for (;;) { |
| 等待错误队列的消息; (6) |
| (得到指向包含有关错误数据的内存块的指针) |
| 读入消息,并根据消息的内容执行相应的操作; (7) |
| 将内存块放回到相应的内存分区中; (8) |
| } |
| } |

|  |
| --- |
| 程序清单 L7.9 等待从一个内存分区中分配内存块 |
| OS\_EVENT \*SemaphorePtr; (1) |
| OS\_MEM \*PartitionPtr; |
| INT8U Partition[100][32]; |
| OS\_STK TaskStk[1000]; |
|  |
|  |
| void main (void) |
| { |
| INT8U err; |
|  |
| OSInit(); (2) |
| . |
| . |
| SemaphorePtr = OSSemCreate(100); (3) |
| PartitionPtr = OSMemCreate(Partition, 100, 32, &err); (4) |
| . |
| OSTaskCreate(Task, (void \*)0, &TaskStk[999], &err); (5) |
| . |
| OSStart(); (6) |
| } |
| void Task (void \*pdata) |
| { |
| INT8U err; |
| INT8U \*pblock; |
|  |
|  |
| for (;;) { |
| OSSemPend(SemaphorePtr, 0, &err); (7) |
| pblock = OSMemGet(PartitionPtr, &err); (8) |
| . |
| . /\* 使用内存块 \*/ |
| . |
| OSMemPut(PartitionPtr, pblock); (9) |
| OSSemPost(SemaphorePtr); (10) |
| } |
| } |