第6章　任务之间的通讯与同步

在 μC/OS-II 中，有多种方法可以保护任务之间的共享数据和提供任务之间的通讯。在前面的章节中，已经讲到了其中的两种：

一是利用宏 OS_ENTER_CRITICAL()和 OS_EXIT_CRITICAL()来关闭中断和打开中断。当两个任务或者一个任务和一个中断服务子程序共享某些数据时，可以采用这种方法，详见 3.00 节　临界段、8.03.02 节 OS_ENTER_CRITICAL()　和 OS_EXIT_CRITICAL()及 9.03.02 节 临界段，OS_CPU.H;

二是利用函数 OSSchedLock()和 OSSchekUnlock()对 μC/OS-II 中的任务调度函数上锁和开锁。用这种方法也可以实现数据的共享，详见 3.06 节 给调度器上锁和开锁。

本章将介绍另外三种用于数据共享和任务通讯的方法：信号量、邮箱和消息队列。

图 F6.1 介绍了任务和中断服务子程序之间是如何进行通讯的。

一个任务或者中断服务子程序可以通过事件控制块 ECB（Event Control Blocks）来向另外的任务发信号[F6.1A(1)]。这里，所有的信号都被看成是事件（Event）。这也说明为什么上面把用于通讯的数据结构叫做事件控制块。一个任务还可以等待另一个任务或中断服务子程序给它发送信号[F6.1A(2)]。这里要注意的是，只有任务可以等待事件发生，中断服务子程序是不能这样做的。对于处于等待状态的任务，还可以给它指定一个最长等待时间，以此来防止因为等待的事件没有发生而无限期地等下去。

多个任务可以同时等待同一个事件的发生[F6.1B]。在这种情况下，当该事件发生后，所有等待该事件的任务中，优先级最高的任务得到了该事件并进入就绪状态，准备执行。上面讲到的事件，可以是信号量、邮箱或者消息队列等。当事件控制块是一个信号量时，任务可以等待它，也可以给它发送消息。

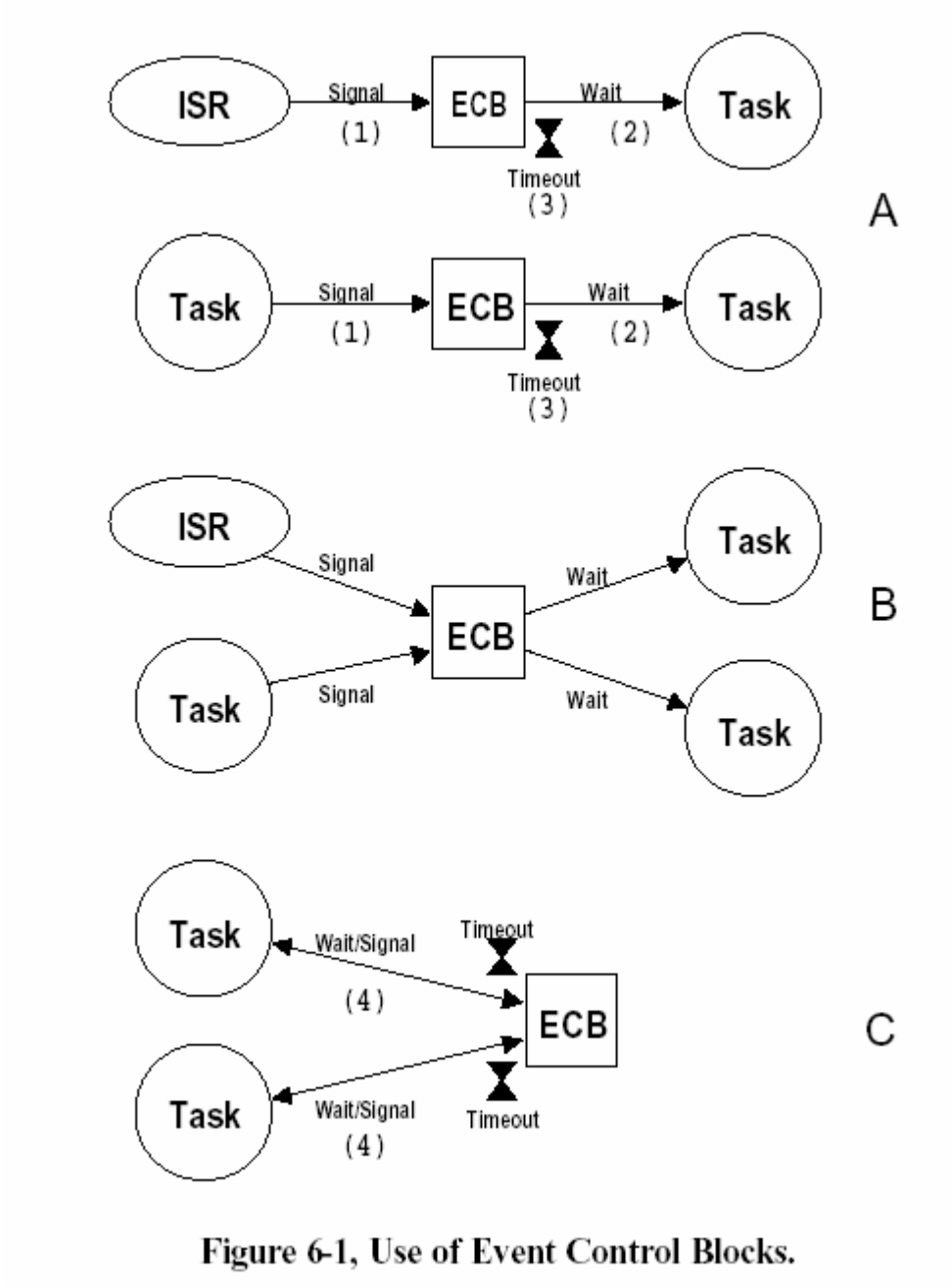

Figure 6-1, Use of Event Control Blocks.

图 6.1 事件控制块的使用

6.0 事件控制块 ECB

μC/OS-II 通过 uCOS_II.H 中定义的 OS_EVENT 数据结构来维护一个事件控制块的所有信息[程序清单 L6.1]，也就是本章开篇讲到的事件控制块 ECB。该结构中除了包含了事件本身的定义，如用于信号量的计数器，用于指向邮箱的指针，以及指向消息队列的指针数组等，还定义了等待该事件的所有任务的列表。程序清单 L6.1 是该数据结构的定义。

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;                    /* 时间类型  */
    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者消息队列时才使用。当所定义的事件是邮箱时，它指向一个消息，而当所定义的事件是消息队列时，它指向一个数据结构，详见 6.06 节消息邮箱和 6.07 节消息队列。

.OSEventTbl[] 和 **.OSEventGrp** 很像前面讲到的 OSRdyTbl[]和 OSRdyGrp，只不过前两者包含的是等待某事件的任务，而后两者包含的是系统中处于就绪状态的任务。（见 3.04 节 就绪表）

.OSEventCnt 当事件是一个信号量时，.OSEventCnt 是用于信号量的计数器，（见 6.05 节信号量）。

.OSEventType 定义了事件的具体类型。它可以是信号量（OS_EVENT_SEM）、邮箱（OS_EVENT_TYPE_MBOX）或消息队列（OS_EVENT_TYPE_Q）中的一种。用户要根据该域的具体值来调用相应的系统函数，以保证对其进行的操作的正确性。

每个等待事件发生的任务都被加入到该事件事件控制块中的等待任务列表中，该列表包括.OSEventGrp 和.OSEventTbl[]两个域。变量前面的[.]说明该变量是数据结构的一个域。在这里，所有的任务的优先级被分成 8 组（每组 8 个优先级），分别对应.OSEventGrp 中的 8 位。当某组中有任务处于等待该事件的状态时，.OSEventGrp 中对应的位就被置位。相应地，该任务在.OSEventTbl[]中的对应位也被置位。.OSEventTbl[]数组的大小由系统中任务的最低优先级决定，这个值由 uCOS_II.H 中的 OS_LOWEST_PRIO 常数定义。这样，在任务优先级比较少的情况下，减少 μC/OS-II 对系统 RAM 的占用量。

当一个事件发生后，该事件的等待事件列表中优先级最高的任务，也即在.OSEventTbl[]中，所有被置 1 的位中，优先级代码最小的任务得到该事件。图 F6.2 给出了.OSEventGrp 和.OSEventTbl[]之间的对应关系。该关系可以描述为：

当.OSEventTbl[0]中的任何一位为 1 时，.OSEventGrp 中的第 0 位为 1。

当.OSEventTbl[1]中的任何一位为 1 时，.OSEventGrp 中的第 1 位为 1。

当.OSEventTbl[2]中的任何一位为 1 时，.OSEventGrp 中的第 2 位为 1。

当.OSEventTbl[3]中的任何一位为 1 时，.OSEventGrp 中的第 3 位为 1。

当.OSEventTbl[4]中的任何一位为 1 时，.OSEventGrp 中的第 4 位为 1。

当.OSEventTbl[5]中的任何一位为 1 时，.OSEventGrp 中的第 5 位为 1。

当.OSEventTbl[6]中的任何一位为 1 时，.OSEventGrp 中的第 6 位为 1。

当.OSEventTbl[7]中的任何一位为 1 时，.OSEventGrp 中的第 7 位为 1。

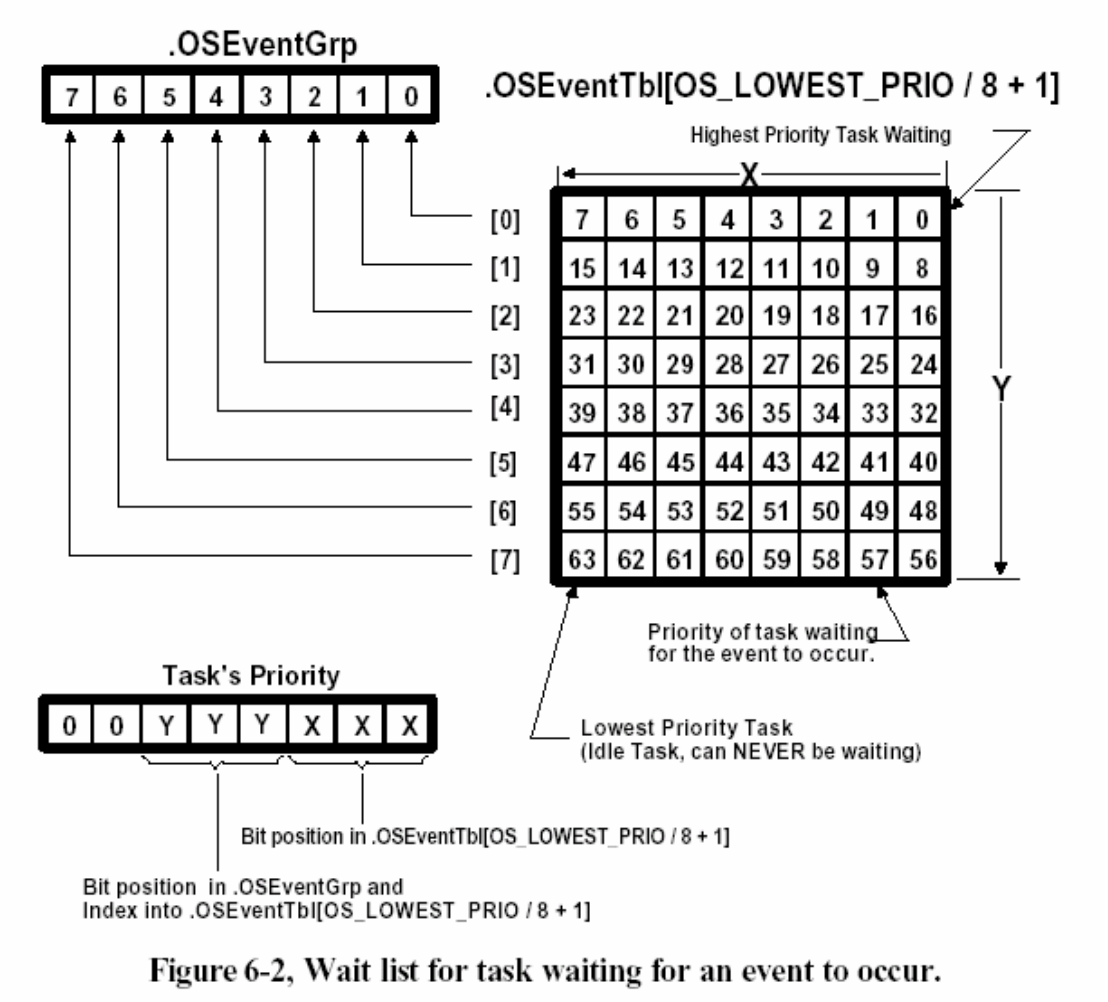

Figure 6-2, Wait list for task waiting for an event to occur.

图 F6.2 事件的等待任务列表

下面的代码将一个任务放到事件的等待任务列表中。

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];
```

其中，prio 是任务的优先级，pevent 是指向事件控制块的指针。

从程序清单 L6.2 可以看出，插入一个任务到等待任务列表中所花的时间是相同的，和表中现有多少个任务无关。从图 F6.2 中可以看出该算法的原理：任务优先级的最低 3 位决定了该任务在相应的.OSEventTbl[]中的位置，紧接着的 3 位则决定了该任务优先级在.OSEventTbl[]中的字节索引。该算法中用到的查找表 OSMapTbl[]（定义在 OS_CORE.C 中）一般在 ROM 中实现。

表T6.1　　OSMapTbl[]

Index	*Bit Mask (Binary)*
0	00000001
1	00000010
2	00000100
3	00001000
4	00010000
5	00100000
6	01000000
7	10000000

从等待任务列表中删除一个任务的算法则正好相反，如程序清单 L6.3 所示。

程序清单 L6.3 从等待任务列表中删除一个任务

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

该代码清除了任务在.OSEventTbl[]中的相应位，并且，如果其所在的组中不再有处于等待该事件的任务时（即.OSEventTbl[prio>>3]为 0），将.OSEventGrp 中的相应位也清除了。和上面的由任务优先级确定该任务在等待表中的位置的算法类似，从等待任务列表中查找处于等待状态的最高优先级任务的算法，也不是从.OSEventTbl[0]开始逐个查询，而是采用了查找另一个表 OSUnMapTbl[256]（见文件 OS_CORE.C）。这里，用于索引的 8 位分别代表对应的 8 组中有任务处于等待状态，其中的最低位具有最高的优先级。用这个值索引，首先得到最高优先级任务所在的组的位置（0～7 之间的一个数）。然后利用.OSEventTbl[]中对应字节再在 OSUnMapTbl[]中查找，就可以得到最高优先级任务在组中的位置（也是 0～7 之间的一个数）。这样，最终就可以得到处于等待该事件状态的最高优先级任务了。程序清单 L6.4 是该算法的具体实现代码。

程序清单 L6.4 在等待任务列表中查找最高优先级的任务

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

举例来说,如果.OSEventGrp 的值是 01101000(二进制),而对应的 OSUnMapTbl[.OSEventGrp]值为 3，说明最高优先级任务所在的组是 3。类似地，如果.OSEventTbl[3]的值是 11100100（二进制），OSUnMapTbl[.OSEventTbl[3]]的值为 2，则处于等待状态的任务的最高优先级是 3×8+2＝26。

在 μC/OS-II 中，事件控制块的总数由用户所需要的信号量、邮箱和消息队列的总数决定。该值由 OS_CFG.H 中的**#define OS_MAX_EVENTS** 定义。在调用 OSInit()时（见 3.11 节，μC/OS-II 的初始化），所有事件控制块被链接成一个单向链表——空闲事件控制块链表（图 F6.3）。每当建立一个信号量、邮箱或者消息队列时，就从该链表中取出一个空闲事件控制块，并对它进行初始化。因为信号量、邮箱和消息队列一旦建立就不能删除，所以事件控制块也不能放回到空闲事件控制块链表中。

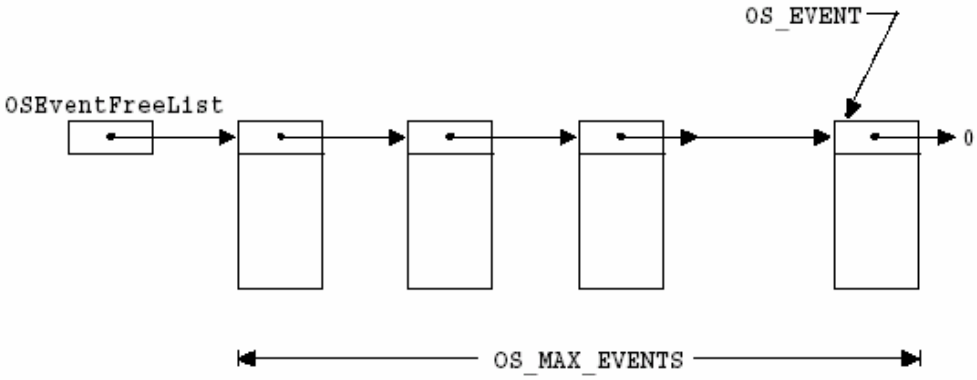

Figure 6-3, List of free ECBs.

图 F6.3 空闲事件控制块链表——Figure 6.3

对于事件控制块进行的一些通用操作包括：

- 初始化一个事件控制块
- 使一个任务进入就绪态
- 使一个任务进入等待该事件的状态
- 因为等待超时而使一个任务进入就绪态

为了避免代码重复和减短程代码长度，μC/OS-II 将上面的操作用 4 个系统函数实现，它们是：OSEventWaitListInit()，OSEventTaskRdy()，OSEventWait()和 OSEventTO()。

6.1 初始化一个事件控制块，OSEventWaitListInit()

程序清单 L6.5 是函数 OSEventWaitListInit()的源代码。当建立一个信号量、邮箱或者消息队列时，相应的建立函数 OSSemInit()，OSMboxCreate()，或者 OSQCreate()通过调用 OSEventWaitListInit()对事件控制块中的等待任务列表进行初始化。该函数初始化一个空的等待任务列表，其中没有任何任务。该函数的调用参数只有一个，就是指向需要初始化的事件控制块的指针 pevent。

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;
    }
}
```

6.2 使一个任务进入就绪态，OSEventTaskRdy()

程序清单 L6.6 是函数 OSEventTaskRdy()的源代码。当发生了某个事件，该事件等待任务列表中的最高优先级任务（Highest Priority Task - HPT）要置于就绪态时，该事件对应的 OSSemPost()，OSMboxPost()，OSQPost()，和 OSQPostFront()函数调用 OSEventTaskRdy()实现该操作。换句话说，该函数从等待任务队列中删除 HPT 任务（Highest Priority Task），并把该任务置于就绪态。图 F6.4 给出了 OSEventTaskRdy()函数最开始的 4 个动作。

该函数首先计算 HPT 任务在.OSEventTbl[]中的字节索引[L6.6/F6.4(1)]，其结果是一个从 0 到 OS_LOWEST_PRIO/8+1 之间的数，并利用该索引得到该优先级任务在.OSEventGrp 中的位屏蔽码[L6.6/F6.4(2)]（从表 T6.1 可以得到该值）。然后，OSEventTaskRdy()函数判断 HPT 任务在.OSEventTbl[]中相应位的位置[L6.6/F6.4(3)],其结果是一个从 0 到 OS_LOWEST_PRIO/8+1 之间的数，以及相应的位屏蔽码[L6.6/F6.4(4)]。根据以上结果，OSEventTaskRdy()函数计算出 HPT 任务的优先级[L6.6(5)]，然后就可以从等待任务列表中删除该任务了[L6.6(6)]。

任务的任务控制块中包含有需要改变的信息。知道了 HPT 任务的优先级，就可以得到指向该任务的任务控制块的指针[L6.6(7)]。因为最高优先级任务运行条件已经得到满足，必须停止 OSTimeTick()函数对.OSTCBDly 域的递减操作，所以 OSEventTaskRdy()直接将该域清澈 0[L6.6(8)]。因为该任务不再等待该事件的发生，所以 OSEventTaskRdy()函数将其任务控制块中指向事件控制块的指针指向 NULL[L6.6(9)]。如果 OSEventTaskRdy()是由 OSMboxPost()或者 OSQPost()调用的，该函数还要将相应的消息传递给 HPT，放在它的任务控制块中[L6.6(10)]。另外，当 OSEventTaskRdy()被调用时，位屏蔽码 msk 作为参数传递给它。该参数是用于对任务控制块中的位清零的位屏蔽码，和所发生事件的类型相对应[L6.6(11)]。最后，根据.OSTCBStat 判断该任务是否已处于就绪状态[L6.6(12)]。如果是，则将 HPT 插入到 μC/OS-II 的就绪任务列表中[L6.6(13)]。注意，HPT 任务得到该事件后不一定进入就绪状态，也许该任务已经由于其它原因挂起了。[见 4.07 节，挂起一个任务，OSTaskSuspend()，和 4.08 节，恢复一个任务，OSTaskResume()]。

另外，.OSEventTaskRdy()函数要在中断禁止的情况下调用。

程序清单 L6.6 使一个任务进入就绪状态

```
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];                              (1)
    bity = OSMapTbl[y];                                                 (2)
    x    = OSUnMapTbl[pevent->OSEventTbl[y]];                           (3)
    bitx = OSMapTbl[x];                                                 (4)
    prio = (INT8U)((y << 3) + x);                                       (5)
    if ((pevent->OSEventTbl[y] &= ~bitx) == 0) {                        (6)
        pevent->OSEventGrp &= ~bity;
    }
    ptcb                 =  OSTCBPrioTbl[prio];                         (7)
    ptcb->OSTCBDly       =  0;                                          (8)
    ptcb->OSTCBEventPtr  = (OS_EVENT *)0;                               (9)
#if (OS_Q_EN && (OS_MAX_QS >= 2)) || OS_MBOX_EN
    ptcb->OSTCBMsg       = msg;                                        (10)
#else
    msg                  = msg;
#endif
    ptcb->OSTCBStat     &= ~msk;                                       (11)
    if (ptcb->OSTCBStat == OS_STAT_RDY) {                              (12)
        OSRdyGrp        |=  bity;                                      (13)
        OSRdyTbl[y]     |=  bitx;
    }
}
```

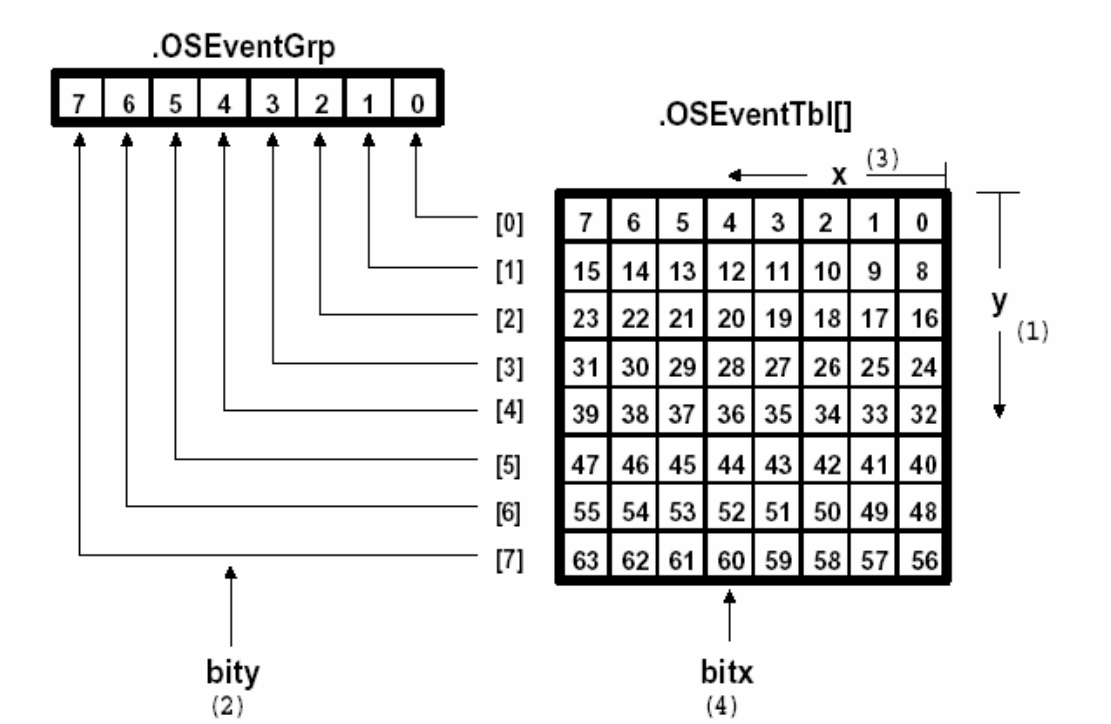

Figure 6-4, Making a task ready-to-run.

图 F6.4 使一个任务进入就绪状态——Figure 6.4

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

程序清单 L6.7 是 OSEventTaskWait()函数的源代码。当某个任务要等待一个事件的发生时，相应事件的 OSSemPend()，OSMboxPend()或者 OSQPend()函数会调用该函数将当前任务从就绪任务表中删除，并放到相应事件的事件控制块的等待任务表中。

程序清单 L6.7 使一个任务进入等待状态

```
void OSEventTaskWait (OS_EVENT *pevent)
{
    OSTCBCur->OSTCBEventPtr = pevent;                                    (1)
    if ((OSRdyTbl[OSTCBCur->OSTCBY] &= ~OSTCBCur->OSTCBBitX) == 0) {     (2)
        OSRdyGrp &= ~OSTCBCur->OSTCBBitY;
    }
    pevent->OSEventTbl[OSTCBCur->OSTCBY] |= OSTCBCur->OSTCBBitX;        (3)
    pevent->OSEventGrp                   |= OSTCBCur->OSTCBBitY;
}
```

在该函数中，首先将指向事件控制块的指针放到任务的任务控制块中 [L6.7(1)]，接着将任务从就绪任务表中删除[L6.7(2)]，并把该任务放到事件控制块的等待任务表中[L6.7(3)]。

6.4 由于等待超时而将任务置为就绪态，OSEventTO()

程序清单 L6.8 是 OSEventTO()函数的源代码。当在预先指定的时间内任务等待的事件没有发生时，OSTimeTick()函数会因为等待超时而将任务的状态置为就绪。在这种情况下，事件的 OSSemPend()，OSMboxPend()或者 OSQPend()函数会调用 OSEventTO()来完成这项工作。该函数负责从事件控制块中的等待任务列表里将任务删除[L6.8(1)]，并把它置成就绪状态[L6.8(2)]。最后，从任务控制块中将指向事件控制块的指针删除[L6.8(3)]。用户应当注意，调用 OSEventTO()也应当先关中断。

程序清单 L6.8 因为等待超时将任务置为就绪状态

```
void  OSEventTO (OS_EVENT *pevent)
{
    if ((pevent->OSEventTbl[OSTCBCur->OSTCBY] &= ~OSTCBCur->OSTCBBitX) == 0)
{                                                                          (1)
        pevent->OSEventGrp &= ~OSTCBCur->OSTCBBitY;
    }
    OSTCBCur->OSTCBStat     = OS_STAT_RDY;                                 (2)
    OSTCBCur->OSTCBEventPtr = (OS_EVENT *)0;                               (3)
}
```

6.5 信号量

μC/OS-II 中的信号量由两部分组成：一个是信号量的计数值，它是一个 16 位的无符号整数（0 到 65,535 之间）；另一个是由等待该信号量的任务组成的等待任务表。用户要在 OS_CFG.H 中将 OS_SEM_EN 开关量常数置成 1，这样 μC/OS-II 才能支持信号量。

在使用一个信号量之前，首先要建立该信号量，也即调用 OSSemCreate()函数（见下一节），对信号量的初始计数值赋值。该初始值为 0 到 65,535 之间的一个数。如果信号量是用来表示一个或者多个事件的发生，那么该信号量的初始值应设为 0。如果信号量是用于对共享资源的访问，那么该信号量的初始值应设为 1（例如，把它当作二值信号量使用）。最后，如果该信号量是用来表示允许任务访问 n 个相同的资源，那么该初始值显然应该是 n，并把该信号量作为一个可计数的信号量使用。

μC/OS-II 提供了 5 个对信号量进行操作的函数。它们是：OSSemCreate()，OSSemPend()，OSSemPost()，OSSemAccept()和 OSSemQuery()函数。图 F6.5 说明了任务、中断服务子程序和信号量之间的关系。图中用钥匙或者旗帜的符号来表示信号量：如果信号量用于对共享资源的访问，那么信号量就用钥匙符号。符号旁边的数字 N 代表可用资源数。对于二值信号量，该值就是 1；如果信号量用于表示某事件的发生，那么就用旗帜符号。这时的数字 N 代表事件已经发生的次数。从图 F6.5 中可以看出 OSSemPost()函数可以由任务或者中断服务子程序调用，而 OSSemPend()和 OSSemQuery()函数只能有任务程序调用。

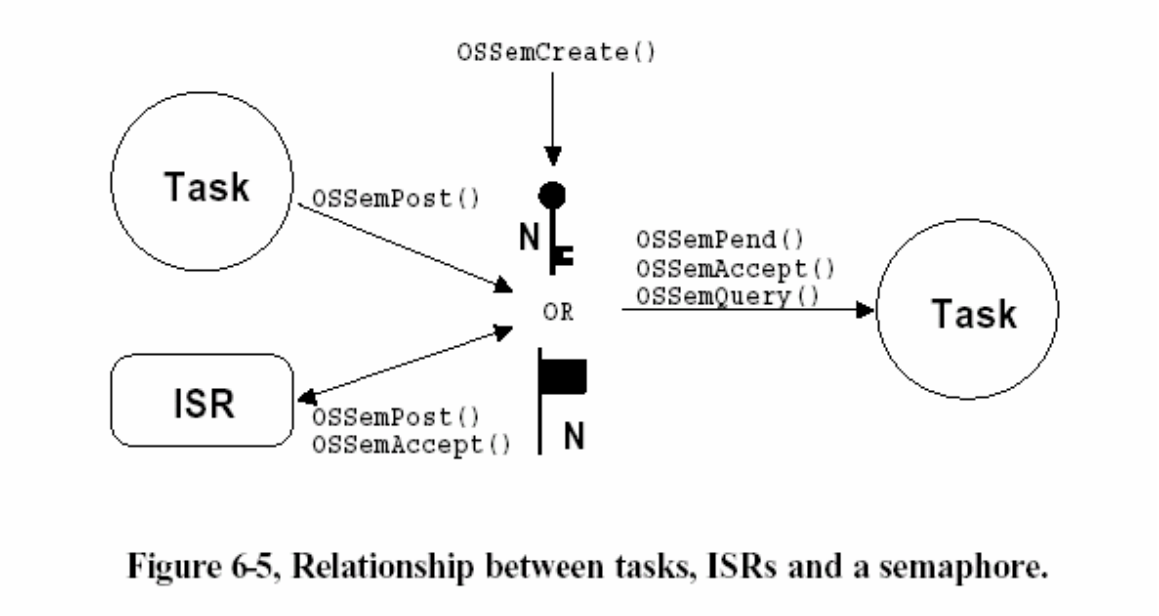

Figure 6-5, Relationship between tasks, ISRs and a semaphore.

图 F6.5 任务、中断服务子程序和信号量之间的关系——Figure 6.5

6.5.1 建立一个信号量, OSSemCreate()

程序清单 L6.9 是 OSSemCreate()函数的源代码。首先，它从空闲任务控制块链表中得到一个事件控制块[L6.9(1)]，并对空闲事件控制链表的指针进行适当的调整，使它指向下一个空闲的事件控制块[L6.9(2)]。如果这时有任务控制块可用[L6.9(3)]，就将该任务控制块的事件类型设置成信号量 OS_EVENT_TYPE_SEM[L6.9(4)]。其它的信号量操作函数 OSSem???()通过检查该域来保证所操作的任务控制块类型的正确。例如，这可以防止调用 OSSemPost()函数对一个用作邮箱的任务控制块进行操作[6.06 节，邮箱]。接着，用信号量的初始值对任务控制块进行初始化[L6.9(5)]，并调用 OSEventWaitListInit()函数对事件控制任务控制块的等待任务列表进行初始化[见 6.01 节，初始化一个任务控制块，OSEventWaitListInit()][L6.9(6)]。因为信号量正在被初始化，所以这时没有任何任务等待该信号量。最后，OSSemCreate()返回给调用函数一个指向任务控制块的指针。以后对信号量的所有操作，如 OSSemPend(), OSSemPost(),

OSSemAccept()和 OSSemQuery()都是通过该指针完成的。因此，这个指针实际上就是该信号量的句柄。如果系统中没有可用的任务控制块，OSSemCreate()将返回一个 NULL 指针。

值得注意的是，在 μC/OS-II 中，信号量一旦建立就不能删除了，因此也就不可能将一个已分配的任务控制块再放回到空闲 ECB 链表中。如果有任务正在等待某个信号量，或者某任务的运行依赖于某信号量的出现时，删除该任务是很危险的。

程序清单 L6.9 建立一个信号量

```
OS_EVENT *OSSemCreate (INT16U cnt)
{
    OS_EVENT *pevent;



    OS_ENTER_CRITICAL();
    pevent = OSEventFreeList;                                    (1)
    if (OSEventFreeList != (OS_EVENT *)0) {                      (2)
        OSEventFreeList = (OS_EVENT *)OSEventFreeList->OSEventPtr;
    }
    OS_EXIT_CRITICAL();
    if (pevent != (OS_EVENT *)0) {                               (3)
        pevent->OSEventType = OS_EVENT_TYPE_SEM;                 (4)
        pevent->OSEventCnt  = cnt;                               (5)
        OSEventWaitListInit(pevent);                             (6)
    }
    return (pevent);                                             (7)
}
```

6.5.2 等待一个信号量，OSSemPend()

程序清单 L6.10 是 OSSemPend()函数的源代码。它首先检查指针 pevent 所指的任务控制块是否是由 OSSemCreate()建立的[L6.10(1)]。如果信号量当前是可用的（信号量的计数值大于 0）[L6.10(2)]，将信号量的计数值减 1[L6.10(3)]，然后函数将“无错”错误代码返回给它的调用函数。显然，如果正在等待信号量，这时的输出正是我们所希望的，也是运行 OSSemPend()函数最快的路径。

如果此时信号量无效（计数器的值是 0），OSSemPend()函数要进一步检查它的调用函数是不是中断服务子程序[L6.10(4)]。在正常情况下，中断服务子程序是不会调用 OSSemPend()函数

的。这里加入这些代码，只是为了以防万一。当然，在信号量有效的情况下，即使是中断服务子程序调用的 OSSemPend()，函数也会成功返回，不会出任何错误。

如果信号量的计数值为 0，而 OSSemPend()函数又不是由中断服务子程序调用的，则调用 OSSemPend()函数的任务要进入睡眠状态，等待另一个任务（或者中断服务子程序）发出该信号量（见下节）。OSSemPend()允许用户定义一个最长等待时间作为它的参数，这样可以避免该任务无休止地等待下去。如果该参数值是一个大于 0 的值，那么该任务将一直等到信号有效或者等待超时。如果该参数值为 0，该任务将一直等待下去。OSSemPend()函数通过将任务控制块中的状态标志.OSTCBStat 置 1，把任务置于睡眠状态[L6.10(5)]，等待时间也同时置入任务控制块中[L6.10(6)]，该值在 OSTimeTick()函数中被逐次递减。注意，OSTimeTick()函数对每个任务的任务控制块的.OSTCBDly 域做递减操作（只要该域不为 0）[见 3.10 节，时钟节拍]。真正将任务置入睡眠状态的操作在 OSEventTaskWait()函数中执行 [见 6.03 节，让一个任务等待某个事件，OSEventTaskWait()][L6.10(7)]。

因为当前任务已经不是就绪态了，所以任务调度函数将下一个最高优先级的任务调入，准备运行[L6.10(8)]。当信号量有效或者等待时间到后，调用 OSSemPend()函数的任务将再一次成为最高优先级任务。这时 OSSched()函数返回。这之后，OSSemPend()要检查任务控制块中的状态标志，看该任务是否仍处于等待信号量的状态[L6.10(9)]。如果是，说明该任务还没有被 OSSemPost()函数发出的信号量唤醒。事实上，该任务是因为等待超时而由 TimeTick()函数把它置为就绪状态的。这种情况下，OSSemPend()函数调用 OSEventTO()函数将任务从等待任务列表中删除[L6.10(10)]，并返回给它的调用任务一个“超时”的错误代码。如果任务的任务控制块中的 OS_STAT_SEM 标志位没有置位，就认为调用 OSSemPend()的任务已经得到了该信号量，将指向信号量 ECB 的指针从该任务的任务控制块中删除，并返回给调用函数一个“无错”的错误代码[L6.10(11)]。

程序清单 L6.10 等待一个信号量

```
void OSSemPend (OS_EVENT *pevent, INT16U timeout, INT8U *err)
{
    OS_ENTER_CRITICAL();
    if (pevent->OSEventType != OS_EVENT_TYPE_SEM) {                    (1)
        OS_EXIT_CRITICAL();
        *err = OS_ERR_EVENT_TYPE;
    }
    if (pevent->OSEventCnt > 0) {                                      (2)
        pevent->OSEventCnt--;                                          (3)
        OS_EXIT_CRITICAL();
        *err = OS_NO_ERR;
```

```
    } else if (OSIntNesting > 0) {                                    (4)
        OS_EXIT_CRITICAL();
        *err = OS_ERR_PEND_ISR;
    } else {
        OSTCBCur->OSTCBStat    |= OS_STAT_SEM;                        (5)
        OSTCBCur->OSTCBDly      = timeout;                            (6)
        OSEventTaskWait(pevent);                                      (7)
        OS_EXIT_CRITICAL();
        OSSched();                                                    (8)
        OS_ENTER_CRITICAL();
        if (OSTCBCur->OSTCBStat & OS_STAT_SEM) {                      (9)
            OSEventTO(pevent);                                       (10)
            OS_EXIT_CRITICAL();
            *err = OS_TIMEOUT;
        } else {
            OSTCBCur->OSTCBEventPtr = (OS_EVENT *)0;                 (11)
            OS_EXIT_CRITICAL();
            *err = OS_NO_ERR;
        }
    }
}
```

6.5.3 发送一个信号量，OSSemPost()

程序清单 L6.11 是 OSSemPost()函数的源代码。它首先检查参数指针 pevent 指向的任务控制块是否是 OSSemCreate()函数建立的[L6.11(1)]，接着检查是否有任务在等待该信号量[L6.11(2)]。如果该任务控制块中的.OSEventGrp 域不是 0，说明有任务正在等待该信号量。这时，就要调用函数 OSEventTaskRdy()[见 6.02 节，使一个任务进入就绪状态，OSEventTaskRdy()]，把其中的最高优先级任务从等待任务列表中删除[L6.11(3)]并使它进入就绪状态。然后，调用 OSSched()任务调度函数检查该任务是否是系统中的最高优先级的就绪任务[L6.11(4)]。如果是，这时就要进行任务切换[当 OSSemPost()函数是在任务中调用的]，准备执行该就绪任务。如果不是，OSSched()直接返回，调用 OSSemPost()的任务得以继续执行。如果这时没有任务在等待该信号量，该信号量的计数值就简单地加 1[L6.11(5)]。

上面是由任务调用 OSSemPost()时的情况。当中断服务子程序调用该函数时，不会发生上面的任务切换。如果需要，任务切换要等到中断嵌套的最外层中断服务子程序调用 OSIntExit()函数后才能进行（见 3.09 节，μC/OS-II 中的中断）。

程序清单 L6.11 发出一个信号量

```
INT8U OSSemPost (OS_EVENT *pevent)
{
    OS_ENTER_CRITICAL();
    if (pevent->OSEventType != OS_EVENT_TYPE_SEM) {                    (1)
        OS_EXIT_CRITICAL();
        return (OS_ERR_EVENT_TYPE);
    }
    if (pevent->OSEventGrp) {                                          (2)
        OSEventTaskRdy(pevent, (void *)0, OS_STAT_SEM);                (3)
        OS_EXIT_CRITICAL();
        OSSched();                                                     (4)
        return (OS_NO_ERR);
    } else {
        if (pevent->OSEventCnt < 65535) {
            pevent->OSEventCnt++;                                      (5)
            OS_EXIT_CRITICAL();
            return (OS_NO_ERR);
        } else {
            OS_EXIT_CRITICAL();
            return (OS_SEM_OVF);
        }
    }
}
```

6.5.4 无等待地请求一个信号量，OSSemAccept()

当一个任务请求一个信号量时，如果该信号量暂时无效，也可以让该任务简单地返回，而不是进入睡眠等待状态。这种情况下的操作是由 OSSemAccept()函数完成的，其源代码见程序清单 L6.12。该函数在最开始也是检查参数指针 pevent 指向的事件控制块是否是由 OSSemCreate()函数建立的[L6.12(1)]，接着从该信号量的事件控制块中取出当前计数值[L6.12(2)]，并检查该信号量是否有效（计数值是否为非 0 值）[L6.12(3)]。如果有效，则将信号量的计数值减 1[L6.12(4)]，然后将信号量的原有计数值返回给调用函数[L6.12(5)]。调用函数需要对该返回值进行检查。如果该值是 0，说明该信号量无效。如果该值大于 0，说明该信号量有效，同时该值也暗示着该信号量当前可用的资源数。应该注意的是，这些可用资源中，已经被该调用函数

自身占用了一个(该计数值已经被减1)。中断服务子程序要请求信号量时,只能用OSSemAccept()而不能用 OSSemPend()，因为中断服务子程序是不允许等待的。

程序清单 L6.12 无等待地请求一个信号量

```
INT16U OSSemAccept (OS_EVENT *pevent)
{
    INT16U cnt;



    OS_ENTER_CRITICAL();
    if (pevent->OSEventType != OS_EVENT_TYPE_SEM) {                    (1)
        OS_EXIT_CRITICAL();
        return (0);
    }
    cnt = pevent->OSEventCnt;                                          (2)
    if (cnt > 0) {                                                     (3)
        pevent->OSEventCnt--;                                          (4)
    }
    OS_EXIT_CRITICAL();
    return (cnt);                                                      (5)
}
```

6.5.5 查询一个信号量的当前状态, OSSemQuery()

在应用程序中，用户随时可以调用函数 OSSemQuery()[程序清单 L6.13]来查询一个信号量的当前状态。该函数有两个参数：一个是指向信号量对应事件控制块的指针 pevent。该指针是在生产信号量时，由 OSSemCreate()函数返回的；另一个是指向用于记录信号量信息的数据结构 OS_SEM_DATA（见 uCOS_II.H）的指针 pdata。因此，调用该函数前，用户必须先定义该结构变量，用于存储信号量的有关信息。在这里，之所以使用一个新的数据结构的原因在于，调用函数应该只关心那些和特定信号量有关的信息,而不是象OS_EVENT数据结构包含的很全面的信息。该数据结构只包含信号量计数值.OSCnt 和等待任务列表.OSEventTbl[]、.OSEventGrp，而 OS_EVENT 中还包含了另外的两个域.OSEventType 和.OSEventPtr。

和其它与信号量有关的函数一样，OSSemQuery()也是先检查 pevent 指向的事件控制块是否是 OSSemCreate()产生的[L6.13(1)]，然后将等待任务列表[L6.13(2)]和计数值[L6.13(3)]从 OS_EVENT 结构拷贝到 OS_SEM_DATA 结构变量中去。

程序清单 L6.13 查询一个信号量的状态

```
INT8U OSSemQuery (OS_EVENT *pevent, OS_SEM_DATA *pdata)
{
    INT8U  i;
    INT8U *psrc;
    INT8U *pdest;

    OS_ENTER_CRITICAL();
    if (pevent->OSEventType != OS_EVENT_TYPE_SEM) {                    (1)
        OS_EXIT_CRITICAL();
        return (OS_ERR_EVENT_TYPE);
    }
    pdata->OSEventGrp = pevent->OSEventGrp;                            (2)
    psrc              = &pevent->OSEventTbl[0];
    pdest             = &pdata->OSEventTbl[0];
    for (i = 0; i < OS_EVENT_TBL_SIZE; i++) {
        *pdest++ = *psrc++;
    }
    pdata->OSCnt      = pevent->OSEventCnt;                            (3)
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);
}
```

6.6 邮箱

邮箱是 μC/OS-II 中另一种通讯机制，它可以使一个任务或者中断服务子程序向另一个任务发送一个指针型的变量。该指针指向一个包含了特定“消息”的数据结构。为了在 μC/OS-II 中使用邮箱，必须将 OS_CFG.H 中的 OS_MBOX_EN 常数置为 1。

使用邮箱之前，必须先建立该邮箱。该操作可以通过调用 OSMboxCreate()函数来完成（见下节），并且要指定指针的初始值。一般情况下，这个初始值是 NULL，但也可以初始化一个邮箱，使其在最开始就包含一条消息。如果使用邮箱的目的是用来通知一个事件的发生（发送一条消息），那么就要初始化该邮箱为 NULL，因为在开始时，事件还没有发生。如果用户用邮箱来共享某些资源，那么就要初始化该邮箱为一个非 NULL 的指针。在这种情况下，邮箱被当成一个二值信号量使用。

μC/OS-II 提供了 5 种对邮箱的操作：OSMboxCreate()，OSMboxPend()，OSMboxPost()，OSMboxAccept()和 OSMboxQuery()函数。图 F6.6 描述了任务、中断服务子程序和邮箱之间的关

系，这里用符号“I”表示邮箱。邮箱包含的内容是一个指向一条消息的指针。一个邮箱只能包含一个这样的指针（邮箱为满时），或者一个指向 NULL 的指针（邮箱为空时）。从图 F6.6 可以看出，任务或者中断服务子程序可以调用函数 OSMboxPost()，但是只有任务可以调用函数 OSMboxPend()和 OSMboxQuery()。

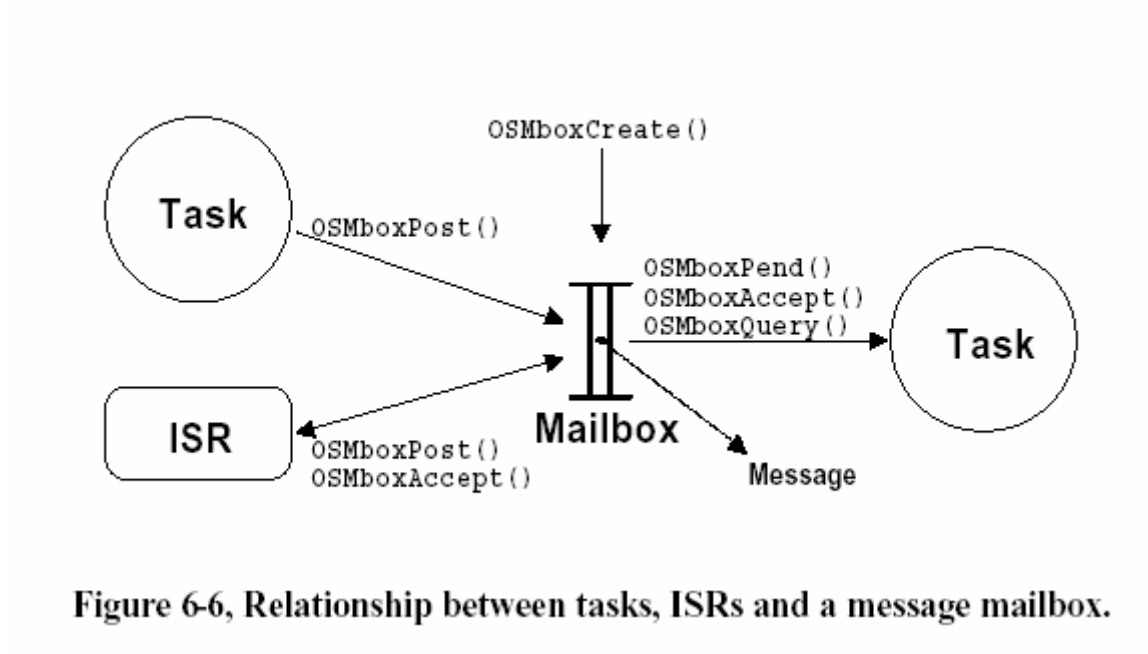

Figure 6-6, Relationship between tasks, ISRs and a message mailbox.

图 F6.6 任务、中断服务子程序和邮箱之间的关系

6.6.1 建立一个邮箱，OSMboxCreate()

程序清单 L6.14 是 OSMboxCreate()函数的源代码，基本上和函数 OSSemCreate()相似。不同之处在于事件控制块的类型被设置成 OS_EVENT_TYPE_MBOX[L6.14(1)]，以及使用.OSEventPtr 域来容纳消息指针，而不是使用.OSEventCnt 域[L6.14(2)]。

OSMboxCreate()函数的返回值是一个指向事件控制块的指针[L6.14(3)]。这个指针在调用函数 OSMboxPend()，OSMboxPost()，OSMboxAccept()和 OSMboxQuery()时使用。因此，该指针可以看作是对应邮箱的句柄。值得注意的是，如果系统中已经没有事件控制块可用，函数 OSMboxCreate()将返回一个 NULL 指针。

邮箱一旦建立，是不能被删除的。比如，如果有任务正在等待一个邮箱的信息，这时删除该邮箱，将有可能产生灾难性的后果。

程序清单 L6.14 建立一个邮箱

```
OS_EVENT *OSMboxCreate (void *msg)
```

```
{
    OS_EVENT *pevent;


    OS_ENTER_CRITICAL();
    pevent = OSEventFreeList;
    if (OSEventFreeList != (OS_EVENT *)0) {
        OSEventFreeList = (OS_EVENT *)OSEventFreeList->OSEventPtr;
    }
    OS_EXIT_CRITICAL();
    if (pevent != (OS_EVENT *)0) {
        pevent->OSEventType = OS_EVENT_TYPE_MBOX;                    (1)
        pevent->OSEventPtr  = msg;                                   (2)
        OSEventWaitListInit(pevent);
    }
    return (pevent);                                                 (3)
}
```

6.6.2 等待一个邮箱中的消息，OSMboxPend()

程序清单 L6.15 是 OSMboxPend()函数的源代码。同样，它和 OSSemPend()也很相似，因此，在这里只讲述其中的不同之处。OSMboxPend()首先检查该事件控制块是由 OSMboxCreate()函数建立的[L6.15(1)]。当.OSEventPtr 域是一个非 NULL 的指针时，说明该邮箱中有可用的消息[L6.15(2)]。这种情况下，OSMboxPend()函数将该域的值复制到局部变量 msg 中，然后将.OSEventPtr 置为 NULL[L6.15(3)]。这正是我们所期望的，也是执行 OSMboxPend()函数最快的路径。

如果此时邮箱中没有消息是可用的（.OSEventPtr 域是 NULL 指针），OSMboxPend()函数检查它的调用者是否是中断服务子程序[L6.15(4)]。象 OSSemPend()函数一样，不能在中断服务子程序中调用 OSMboxPend()，因为中断服务子程序是不能等待的。这里的代码同样是为了以防万一。但是，如果邮箱中有可用的消息，即使从中断服务子程序中调用 OSMboxPend()函数，也一样是成功的。

如果邮箱中没有可用的消息，OSMboxPend()的调用任务就被挂起，直到邮箱中有了消息或者等待超时[L6.15(5)]。当有其它的任务向该邮箱发送了消息后（或者等待时间超时），这时，该任务再一次成为最高优先级任务，OSSched()返回。这时，OSMboxPend()函数要检查是否有消息被放到该任务的任务控制块中[L6.15(6)]。如果有，那么该次函数调用成功，对应的消息被返回到调用函数。

程序清单 L6.15 等待一个邮箱中的消息

```
void *OSMboxPend (OS_EVENT *pevent, INT16U timeout, INT8U *err)
{
    void  *msg;

    OS_ENTER_CRITICAL();
    if (pevent->OSEventType != OS_EVENT_TYPE_MBOX) {                    (1)
        OS_EXIT_CRITICAL();
        *err = OS_ERR_EVENT_TYPE;
        return ((void *)0);
    }
    msg = pevent->OSEventPtr;
    if (msg != (void *)0) {                                             (2)
        pevent->OSEventPtr = (void *)0;                                 (3)
        OS_EXIT_CRITICAL();
        *err = OS_NO_ERR;
    } else if (OSIntNesting > 0) {                                      (4)
        OS_EXIT_CRITICAL();
        *err = OS_ERR_PEND_ISR;
    } else {
        OSTCBCur->OSTCBStat |= OS_STAT_MBOX;                            (5)
        OSTCBCur->OSTCBDly   = timeout;
        OSEventTaskWait(pevent);
        OS_EXIT_CRITICAL();
        OSSched();
        OS_ENTER_CRITICAL();
        if ((msg = OSTCBCur->OSTCBMsg) != (void *)0) {                  (6)
            OSTCBCur->OSTCBMsg      = (void *)0;
            OSTCBCur->OSTCBStat     = OS_STAT_RDY;
            OSTCBCur->OSTCBEventPtr = (OS_EVENT *)0;
            OS_EXIT_CRITICAL();
            *err                    = OS_NO_ERR;
        } else if (OSTCBCur->OSTCBStat & OS_STAT_MBOX) {                (7)
            OSEventTO(pevent);                                          (8)
            OS_EXIT_CRITICAL();
            msg                     = (void *)0;                        (9)
            *err                    = OS_TIMEOUT;
        } else {
```

```
            msg                      = pevent->OSEventPtr;              (10)
            pevent->OSEventPtr       = (void *)0;
(11)
            OSTCBCur->OSTCBEventPtr = (OS_EVENT *)0;                    (12)
            OS_EXIT_CRITICAL();
            *err                     = OS_NO_ERR;
        }
    }
    return (msg);
}
```

在 OSMboxPend()函数中，通过检查任务控制块中的.OSTCBStat 域中的 OS_STAT_MBOX 位，可以知道是否等待超时。如果该域被置 1，说明任务等待已经超时[L6.15(7)]。这时，通过调用函数 OSEventTo()可以将任务从邮箱的等待列表中删除[L6.15(8)]。因为此时邮箱中没有消息，所以返回的指针是 NULL[L6.15(9)]。如果 OS_STAT_MBOX 位没有被置 1，说明所等待的消息已经被发出。OSMboxPend()的调用函数得到指向消息的指针[L6.15(10)]。此后，OSMboxPend()函数通过将邮箱事件控制块的.OSEventPtr 域置为 NULL 清空该邮箱，并且要将任务任务控制块中指向邮箱事件控制块的指针删除[L6.15(12)]。

6.6.3 发送一个消息到邮箱中，OSMboxPost()

程序清单 L6.16 是 OSMboxPost()函数的源代码。检查了事件控制块是否是一个邮箱后[L6.16(1)]，OSMboxPost()函数还要检查是否有任务在等待该邮箱中的消息[L6.16(2)]。如果事件控制块中的 OSEventGrp 域包含非零值，就暗示着有任务在等待该消息。这时，调用 OSEventTaskRdy()将其中的最高优先级任务从等待列表中删除[见 6.02 节，使一个任务进入就绪状态，OSEventTaskRdy()][L6.16(3)]，加入系统的就绪任务列表中，准备运行。然后，调用 OSSched()函数[L6.16(4)]，检查该任务是否是系统中最高优先级的就绪任务。如果是，执行任务切换[仅当 OSMboxPost()函数是由任务调用时]，该任务得以执行。如果该任务不是最高优先级的任务，OSSched()返回，OSMboxPost()的调用函数继续执行。如果没有任何任务等待该消息，指向消息的指针就被保存到邮箱中[L6.16(6)]（假设此时邮箱中的指针不是非 NULL 的[L6.16(5)]）。这样，下一个调用 OSMboxPend()函数的任务就可以立刻得到该消息了。

注意，如果 OSMboxPost()函数是从中断服务子程序中调用的，那么，这时并不发生上下文的切换。如果需要，中断服务子程序引起的上下文切换只发生在中断嵌套的最外层中断服务子程序对 OSIntExit()函数的调用时（见 3.09 节，μC/OS-II 中的中断）。

程序清单 L6.16 向邮箱中发送一条消息

```
INT8U OSMboxPost (OS_EVENT *pevent, void *msg)
{
    OS_ENTER_CRITICAL();
    if (pevent->OSEventType != OS_EVENT_TYPE_MBOX) {                    (1)
        OS_EXIT_CRITICAL();
        return (OS_ERR_EVENT_TYPE);
    }
    if (pevent->OSEventGrp) {                                           (2)
        OSEventTaskRdy(pevent, msg, OS_STAT_MBOX);                      (3)
        OS_EXIT_CRITICAL();
        OSSched();                                                      (4)
        return (OS_NO_ERR);
    } else {
        if (pevent->OSEventPtr != (void *)0) {                          (5)
            OS_EXIT_CRITICAL();
            return (OS_MBOX_FULL);
        } else {
            pevent->OSEventPtr = msg;                                   (6)
            OS_EXIT_CRITICAL();
            return (OS_NO_ERR);
        }
    }
}
```

6.6.4 无等待地从邮箱中得到一个消息，OSMboxAccept()

应用程序也可以以无等待的方式从邮箱中得到消息。这可以通过程序清单 L6.17 中的OSMboxAccept()函数来实现。OSMboxAccept()函数开始也是检查事件控制块是否是由OSMboxCreate()函数建立的 [L6.17(1)]。接着，它得到邮箱中的当前内容[L6.17(2)]，并判断是否有消息是可用的[L6.17(3)]。如果邮箱中有消息，就把邮箱清空[L6.17(4)]，而邮箱中原来指向消息的指针被返回给 OSMboxAccept()的调用函数[L6.17(5)]。OSMboxAccept()函数的调用函数必须检查该返回值是否为 NULL。如果该值是 NULL，说明邮箱是空的，没有可用的消息。如果该值是非 NULL 值，说明邮箱中有消息可用，而且该调用函数已经得到了该消息。中断服务子程序在试图得到一个消息时，应该使用 OSMboxAccept()函数，而不能使用 OSMboxPend()函数。

OSMboxAccept()函数的另一个用途是，用户可以用它来清空一个邮箱中现有的内容。

程序清单 L6.17 无等待地从邮箱中得到消息

```
void *OSMboxAccept (OS_EVENT *pevent)
{
    void  *msg;



    OS_ENTER_CRITICAL();
    if (pevent->OSEventType != OS_EVENT_TYPE_MBOX) {                    (1)
        OS_EXIT_CRITICAL();
        return ((void *)0);
    }
    msg = pevent->OSEventPtr;                                           (2)
    if (msg != (void *)0) {                                             (3)
        pevent->OSEventPtr = (void *)0;                                 (4)
    }
    OS_EXIT_CRITICAL();
    return (msg);                                                       (5)
}
```

6.6.5 查询一个邮箱的状态，OSMboxQuery()

OSMboxQuery()函数使应用程序可以随时查询一个邮箱的当前状态。程序清单 L6.18 是该函数的源代码。它需要两个参数：一个是指向邮箱的指针 pevent。该指针是在建立该邮箱时，由 OSMboxCreate()函数返回的；另一个是指向用来保存有关邮箱的信息的 OS_MBOX_DATA（见 uCOS_II.H）数据结构的指针 pdata。在调用 OSMboxCreate()函数之前，必须先定义该结构变量，用来保存有关邮箱的信息。之所以定义一个新的数据结构，是因为这里关心的只是和特定邮箱有关的内容，而非整个 OS_EVENT 数据结构的内容。后者还包含了另外两个域（.OSEventCnt 和.OSEventType），而 OS_MBOX_DATA 只包含邮箱中的消息指针（.OSMsg）和该邮箱现有的等待任务列表（.OSEventTbl[]和.OSEventGrp）。

和前面的所以函数一样，该函数也是先检查事件控制是否是邮箱[L6.18(1)]。然后，将邮箱中的等待任务列表[L6.18(2)]和邮箱中的消息[L6.18(3)]从 OS_EVENT 数据结构复制到 OS_MBOX_DATA 数据结构。

程序清单 L6.18 查询邮箱的状态

```
INT8U OSMboxQuery (OS_EVENT *pevent, OS_MBOX_DATA *pdata)
{
    INT8U  i;
```

```
    INT8U *psrc;
    INT8U *pdest;

    OS_ENTER_CRITICAL();
    if (pevent->OSEventType != OS_EVENT_TYPE_MBOX) {                    (1)
        OS_EXIT_CRITICAL();
        return (OS_ERR_EVENT_TYPE);
    }
    pdata->OSEventGrp = pevent->OSEventGrp;                             (2)
    psrc              = &pevent->OSEventTbl[0];
    pdest             = &pdata->OSEventTbl[0];
    for (i = 0; i < OS_EVENT_TBL_SIZE; i++) {
        *pdest++ = *psrc++;
    }
    pdata->OSMsg      = pevent->OSEventPtr;                             (3)
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);
}
```

6.6.6 用邮箱作二值信号量

一个邮箱可以被用作二值的信号量。首先，在初始化时，将邮箱设置为一个非零的指针（如 void *1）。这样，一个任务可以调用 OSMboxPend()函数来请求一个信号量，然后通过调用 OSMboxPost()函数来释放一个信号量。程序清单 L6.19 说明了这个过程是如何工作的。如果用户只需要二值信号量和邮箱，这样做可以节省代码空间。这时可以将 OS_SEM_EN 设置为 0，只使用邮箱就可以了。

程序清单 L6.19 使用邮箱作为二值信号量

```
OS_EVENT *MboxSem;



void Task1 (void *pdata)
{
    INT8U err;

    for (;;) {
        OSMboxPend(MboxSem, 0, &err);   /* 获得对资源的访问权  */
```

```
        .
        .     /* 任务获得信号量,对资源进行访问   */
        .
        OSMboxPost(MboxSem, (void*)1);  /* 释放对资源的访问权 */
    }
}
```

6.6.7 用邮箱实现延时，而不使用 OSTimeDly()

邮箱的等待超时功能可以被用来模仿 OSTimeDly()函数的延时，如程序清单 L6.20 所示。如果在指定的时间段 TIMEOUT 内，没有消息到来，Task1()函数将继续执行。这和 OSTimeDly(TIMEOUT)功能很相似。但是，如果 Task2()在指定的时间结束之前，向该邮箱发送了一个“哑”消息，Task1()就会提前开始继续执行。这和调用 OSTimeDlyResume()函数的功能是一样的。注意，这里忽略了对返回的消息的检查，因为此时关心的不是得到了什么样的消息。

程序清单 L6.20 使用邮箱实现延时

```
OS_EVENT *MboxTimeDly;


void Task1 (void *pdata)
{
    INT8U err;


    for (;;) {
        OSMboxPend(MboxTimeDly, TIMEOUT, &err);   /* 延时该任务 */
        .
        .     /* 延时结束后执行的代码   */
        .
    }
}


void Task2 (void *pdata)
{
    INT8U err;
```

```
    for (;;) {
        OSMboxPost(MboxTimeDly, (void *)1);        /* 取消任务1的延时 */
        .
        .
    }
}
```

6.7 消息队列

消息队列是 μC/OS-II 中另一种通讯机制，它可以使一个任务或者中断服务子程序向另一个任务发送以指针方式定义的变量。因具体的应用有所不同，每个指针指向的数据结构变量也有所不同。为了使用 μC/OS-II 的消息队列功能，需要在 OS_CFG.H 文件中，将 OS_Q_EN 常数设置为 1，并且通过常数 OS_MAX_QS 来决定 μC/OS-II 支持的最多消息队列数。

在使用一个消息队列之前，必须先建立该消息队列。这可以通过调用 OSQCreate()函数（见 6.07.01 节），并定义消息队列中的单元数（消息数）来完成。

μC/OS-II 提供了 7 个对消息队列进行操作的函数：OSQCreate()，OSQPend()，OSQPost()，OSQPostFront()，OSQAccept()，OSQFlush()和 OSQQuery()函数。图 F6.7 是任务、中断服务子程序和消息队列之间的关系。其中，消息队列的符号很像多个邮箱。实际上，我们可以将消息队列看作时多个邮箱组成的数组，只是它们共用一个等待任务列表。每个指针所指向的数据结构是由具体的应用程序决定的。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()就会把消息队列填满。从图 F6.7 中可以看出，一个任务或者中断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。

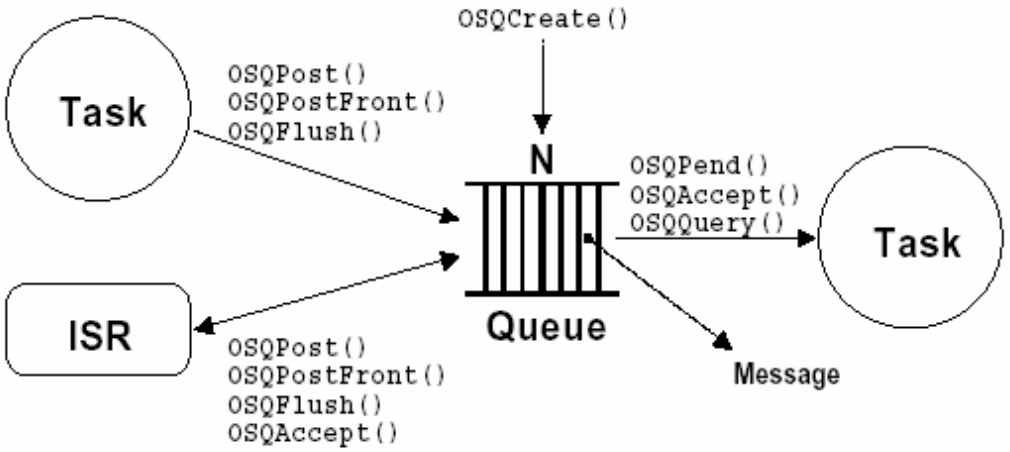

Figure 6-7, Relationship between tasks, ISRs and a message queue.

图 F6.7 任务、中断服务子程序和消息队列之间的关系——Figure 6.7

图 F6.8 是实现消息队列所需要的各种数据结构。这里也需要事件控制块来记录等待任务列表[F6.8(1)]，而且，事件控制块可以使多个消息队列的操作和信号量操作、邮箱操作相同的代码。当建立了一个消息队列时，一个队列控制块（OS_Q 结构，见 OS_Q.C 文件）也同时被建立，并通过 OS_EVENT 中的.OSEventPtr 域链接到对应的事件控制块[F6.8(2)]。在建立一个消息队列之前，必须先定义一个含有与消息队列最大消息数相同个数的指针数组[F6.8(3)]。数组的起始地址以及数组中的元素数作为参数传递给 OSQCreate()函数。事实上，如果内存占用了连续的地址空间，也没有必要非得使用指针数组结构。

文件 OS_CFG.H 中的常数 OS_MAX_QS 定义了在 μC/OS-II 中可以使用的最大消息队列数，这个值最小应为 2。μC/OS-II 在初始化时建立一个空闲的队列控制块链表，如图 F6.9 所示。

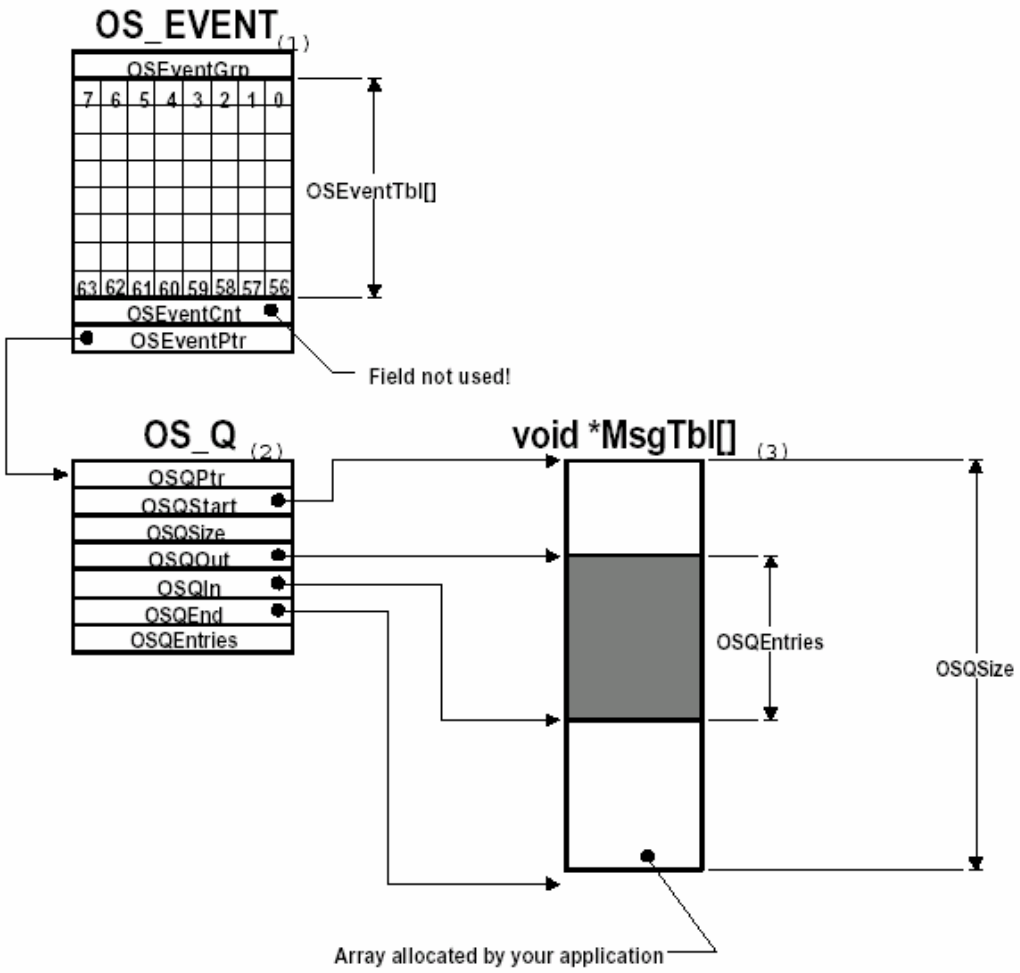

Figure 6-8, Data structures used in a message queue.

图 F6.8 用于消息队列的数据结构——Figure 6.8

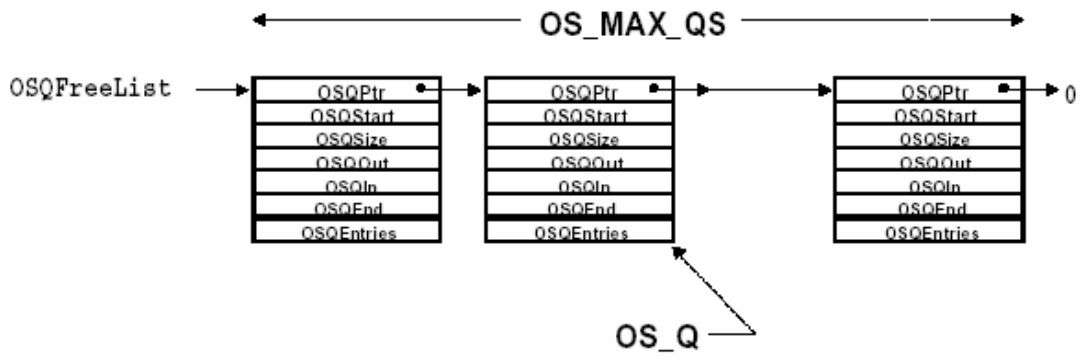

Figure 6-9, List of free queue control blocks.

图 F6.9 空闲队列控制块链表——Figure 6.9

队列控制块是一个用于维护消息队列信息的数据结构，它包含了以下的一些域。这里，仍然在各个变量前加入一个[.]来表示它们是数据结构中的一个域。

.OSQPtr 在空闲队列控制块中链接所有的队列控制块。一旦建立了消息队列，该域就不再有用了。

.OSQStart 是指向消息队列的指针数组的起始地址的指针。用户应用程序在使用消息队列之前必须先定义该数组。

.OSQEnd 是指向消息队列结束单元的下一个地址的指针。该指针使得消息队列构成一个循环的缓冲区。

.OSQIn 是指向消息队列中插入下一条消息的位置的指针。当.OSQIn 和.OSQEnd 相等时，.OSQIn 被调整指向消息队列的起始单元。

.OSQOut 是指向消息队列中下一个取出消息的位置的指针。当.OSQOut 和.OSQEnd 相等时，.OSQOut 被调整指向消息队列的起始单元。

.OSQSize 是消息队列中总的单元数。该值是在建立消息队列时由用户应用程序决定的。在μC/OS-II 中，该值最大可以是 65,535。

.OSQEntries 是消息队列中当前的消息数量。当消息队列是空的时，该值为 0。当消息队列满了以后，该值和.OSQSize 值一样。 在消息队列刚刚建立时，该值为 0。

消息队列最根本的部分是一个循环缓冲区，如图F6.10。其中的每个单元包含一个指针。队列未满时，.OSQIn [F6.10(1)]指向下一个存放消息的地址单元。如果队列已满（.OSQEntries与.OSQSize相等），.OSQIn [F6.10(3)]则与.OSQOut指向同一单元。如果在.OSQIn指向的单元插入新的指向消息的指针，就构成FIFO（First-In-First-Out）队列。相反，如果在.OSQOut指向的单元的下一个单元插入新的指针，就构成LIFO队列（Last-In-First-Out）[F6.10(2)]。当.OSQEntries和.OSQSize相等时，说明队列已满。消息指针总是从.OSQOut [F6.10(4)]指向的单元取出。指针.OSQStart和.OSQEnd [F6.10(5)]定义了消息指针数组的头尾，以便在.OSQIn和.OSQOut到达队列的边缘时，进行边界检查和必要的指针调整，实现循环功能。

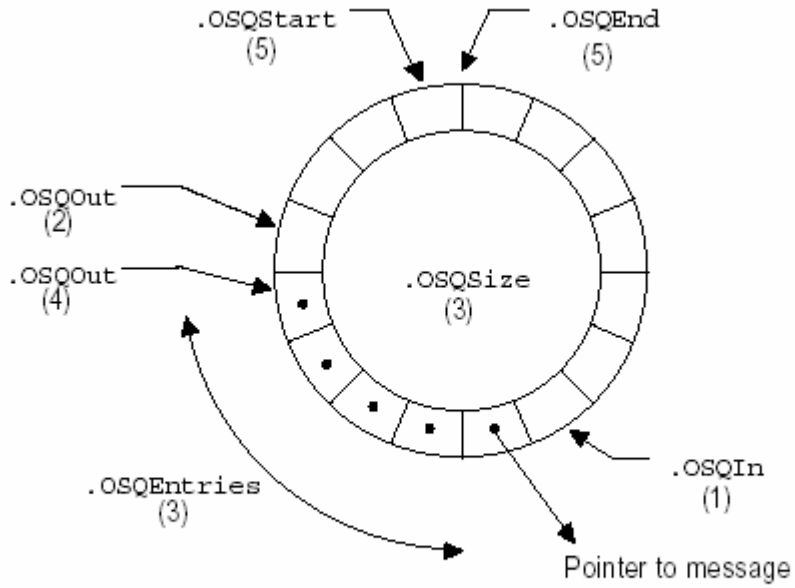

Figure 6-10, Message queue is a circular buffer of pointers.

图 F6.10 消息队列是一个由指针组成的循环缓冲区——Figure 6.10

6.7.1 建立一个消息队列，OSQCreate()

程序清单 L6.21 是 OSQCreate()函数的源代码。该函数需要一个指针数组来容纳指向各个消息的指针。该指针数组必须声名为 void 类型。

OSQCreate()首先从空闲事件控制块链表中取得一个事件控制块（见图 F6.3）[L6.21(1)]，并对剩下的空闲事件控制块列表的指针做相应的调整，使它指向下一个空闲事件控制块[L6.21(2)]。接着，OSQCreate()函数从空闲队列控制块列表中取出一个队列控制块[L6.21(3)]。如果有空闲队列控制块是可以的，就对其进行初始化[L6.21(4)]。然后该函数将事件控制块的类型设置为 OS_EVENT_TYPE_Q [L6.21(5)]，并使其.OSEventPtr 指针指向队列控制块[L6.21(6)]。OSQCreate()还要调用 OSEventWaitListInit()函数对事件控制块的等待任务列表初始化[见 6.01 节，初始化一个事件控制块，OSEventWaitListInit()][L6.21(7)]。因为此时消息队列正在初始化，显然它的等待任务列表是空的。最后，OSQCreate()向它的调用函数返回一个指向事件控制块的指针[L6.21(9)]。该指针将在调用 OSQPend()，OSQPost()，OSQPostFront()，OSQFlush()，OSQAccept()和 OSQQuery()等消息队列处理函数时使用。因此，该指针可以被看作是对应消息队列的句柄。值得注意的是，如果此时没有空闲的事件控制块，OSQCreate()函数将返回一个 NULL 指针。如果没有队列控制块可以使用，为了不浪费事件控制

块资源，OSQCreate()函数将把刚刚取得的事件控制块重新返还给空闲事件控制块列表[L6.21(8)]。

另外，消息队列一旦建立就不能再删除了。试想，如果有任务正在等待某个消息队列中的消息，而此时又删除该消息队列，将是很危险的。

程序清单 L6.21 建立一个消息队列

```
OS_EVENT *OSQCreate (void **start, INT16U size)
{
    OS_EVENT *pevent;
    OS_Q     *pq;



    OS_ENTER_CRITICAL();
    pevent = OSEventFreeList;                                          (1)
    if (OSEventFreeList != (OS_EVENT *)0) {
        OSEventFreeList = (OS_EVENT *)OSEventFreeList->OSEventPtr;     (2)
    }
    OS_EXIT_CRITICAL();
    if (pevent != (OS_EVENT *)0) {
        OS_ENTER_CRITICAL();
        pq = OSQFreeList;                                              (3)
        if (OSQFreeList != (OS_Q *)0) {
            OSQFreeList = OSQFreeList->OSQPtr;
        }
        OS_EXIT_CRITICAL();
        if (pq != (OS_Q *)0) {
            pq->OSQStart        = start;                               (4)
            pq->OSQEnd          = &start[size];
            pq->OSQIn           = start;
            pq->OSQOut          = start;
            pq->OSQSize         = size;
            pq->OSQEntries      = 0;
            pevent->OSEventType = OS_EVENT_TYPE_Q;                     (5)
            pevent->OSEventPtr  = pq;                                  (6)
            OSEventWaitListInit(pevent);                               (7)
        } else {
            OS_ENTER_CRITICAL();
            pevent->OSEventPtr = (void *)OSEventFreeList;              (8)
```

```
            OSEventFreeList    = pevent;
            OS_EXIT_CRITICAL();
            pevent = (OS_EVENT *)0;
        }
    }
    return (pevent);                                                    (9)
}
```

6.7.2 等待一个消息队列中的消息，OSQPend()

程序清单 L6.22 是 OSQPend()函数的源代码。OSQPend()函数首先检查事件控制块是否是由 OSQCreate()函数建立的[L6.22(1)]，接着，该函数检查消息队列中是否有消息可用（即.OSQEntries 是否大于 0）[L6.22(2)]。如果有，OSQPend()函数将指向消息的指针复制到 msg 变量中，并让.OSQOut 指针指向队列中的下一个单元[L6.22(3)],然后将队列中的有效消息数减 1 [L6.22(4)]。因为消息队列是一个循环的缓冲区，OSQPend()函数需要检查.OSQOut 是否超过了队列中的最后一个单元 [L6.22(5)]。当发生这种越界时，就要将.OSQOut 重新调整到指向队列的起始单元 [L6.22(6)]。这是我们调用 OSQPend()函数时所期望的，也是执行 OSQPend()函数最快的路径。

程序清单 L6.22 在一个消息队列中等待一条消息

```
void *OSQPend (OS_EVENT *pevent, INT16U timeout, INT8U *err)
{
    void  *msg;
    OS_Q  *pq;



    OS_ENTER_CRITICAL();
    if (pevent->OSEventType != OS_EVENT_TYPE_Q) {                       (1)
        OS_EXIT_CRITICAL();
        *err = OS_ERR_EVENT_TYPE;
        return ((void *)0);
    }
    pq = pevent->OSEventPtr;
    if (pq->OSQEntries != 0) {                                          (2)
        msg = *pq->OSQOut++;                                            (3)
        pq->OSQEntries--;                                               (4)
        if (pq->OSQOut == pq->OSQEnd) {                                 (5)
```

```
            pq->OSQOut = pq->OSQStart;                                  (6)
        }
        OS_EXIT_CRITICAL();
        *err = OS_NO_ERR;
    } else if (OSIntNesting > 0) {                                      (7)
        OS_EXIT_CRITICAL();
        *err = OS_ERR_PEND_ISR;
    } else {
        OSTCBCur->OSTCBStat    |= OS_STAT_Q;                            (8)
        OSTCBCur->OSTCBDly      = timeout;
        OSEventTaskWait(pevent);
        OS_EXIT_CRITICAL();
        OSSched();                                                      (9)
        OS_ENTER_CRITICAL();
        if ((msg = OSTCBCur->OSTCBMsg) != (void *)0) {                  (10)
            OSTCBCur->OSTCBMsg      = (void *)0;
            OSTCBCur->OSTCBStat     = OS_STAT_RDY;
            OSTCBCur->OSTCBEventPtr = (OS_EVENT *)0;                    (11)
            OS_EXIT_CRITICAL();
            *err                    = OS_NO_ERR;
        } else if (OSTCBCur->OSTCBStat & OS_STAT_Q) {                   (12)
            OSEventTO(pevent);                                          (13)
            OS_EXIT_CRITICAL();
            msg                     = (void *)0;                        (14)
            *err                    = OS_TIMEOUT;
        } else {
            msg = *pq->OSQOut++;                                        (15)
            pq->OSQEntries--;
            if (pq->OSQOut == pq->OSQEnd) {
                pq->OSQOut = pq->OSQStart;
            }
            OSTCBCur->OSTCBEventPtr = (OS_EVENT *)0;                    (16)
            OS_EXIT_CRITICAL();
            *err = OS_NO_ERR;
        }
    }
    return (msg);                                                       (17)
}
```

如果这时消息队列中没有消息（.OSEventEntries 是 0），OSQPend()函数检查它的调用者是否是中断服务子程序[L6.22(7)]。象 OSSemPend()和 OSMboxPend()函数一样，不能在中断服务子程序中调用 OSQPend()，因为中断服务子程序是不能等待的。但是，如果消息队列中有消息，即使从中断服务子程序中调用 OSQPend()函数，也一样是成功的。

如果消息队列中没有消息，调用 OSQPend()函数的任务被挂起[L6.22(8)]。当有其它的任务向该消息队列发送了消息或者等待时间超时，并且该任务成为最高优先级任务时，OSSched()返回[L6.22(9)]。这时，OSQPend()要检查是否有消息被放到该任务的任务控制块中[L6.22(10)]。如果有，那么该次函数调用成功，把任务的任务控制块中指向消息队列的指针删除[L6.22(17)]，并将对应的消息被返回到调用函数[L6.22(17)]。

在 OSQPend()函数中，通过检查任务的任务控制块中的.OSTCBStat 域，可以知道是否等到时间超时。如果其对应的 OS_STAT_Q 位被置 1，说明任务等待已经超时[L6.22(12)]。这时，通过调用函数 OSEventTo()可以将任务从消息队列的等待任务列表中删除[L6.22(13)]。这时，因为消息队列中没有消息，所以返回的指针是 NULL[L6.22(14)]。

如果任务控制块标志位中的 OS_STAT_Q 位没有被置 1，说明有任务发出了一条消息。OSQPend()函数从队列中取出该消息[L6.22(15)]。然后，将任务的任务控制中指向事件控制块的指针删除[L6.22(16)]。

6.7.3 向消息队列发送一个消息（FIFO），OSQPost()

程序清单 L6.23 是 OSQPost()函数的源代码。在确认事件控制块是消息队列后[L6.23(1)]，OSQPost()函数检查是否有任务在等待该消息队列中的消息[L6.23(2)]。当事件控制块的.OSEventGrp 域为非 0 值时，说明该消息队列的等待任务列表中有任务。这时，调用 OSEventTaskRdy()函数 [见 6.02 节，使一个任务进入就绪状态，OSEventTaskRdy()]从列表中取出最高优先级的任务[L6.23(3)]，并将它置于就绪状态。然后调用函数 OSSched() [L6.23(4)]进行任务的调度。如果上面取出的任务的优先级在整个系统就绪的任务里也是最高的，而且 OSQPost()函数不是中断服务子程序调用的，就执行任务切换，该最高优先级任务被执行。否则的话，OSSched()函数直接返回，调用 OSQPost()函数的任务继续执行。

程序清单 L6.23 向消息队列发送一条消息

```
INT8U OSQPost (OS_EVENT *pevent, void *msg)
{
    OS_Q   *pq;
```

```
    OS_ENTER_CRITICAL();
    if (pevent->OSEventType != OS_EVENT_TYPE_Q) {                    (1)
        OS_EXIT_CRITICAL();
        return (OS_ERR_EVENT_TYPE);
    }
    if (pevent->OSEventGrp) {                                        (2)
        OSEventTaskRdy(pevent, msg, OS_STAT_Q);                      (3)
        OS_EXIT_CRITICAL();
        OSSched();
(4)
        return (OS_NO_ERR);
    } else {
        pq = pevent->OSEventPtr;
        if (pq->OSQEntries >= pq->OSQSize) {                         (5)
            OS_EXIT_CRITICAL();
            return (OS_Q_FULL);
        } else {
            *pq->OSQIn++ = msg;                                      (6)
            pq->OSQEntries++;
            if (pq->OSQIn == pq->OSQEnd) {
                pq->OSQIn = pq->OSQStart;
            }
            OS_EXIT_CRITICAL();
        }
        return (OS_NO_ERR);
    }
}
```

如果没有任务等待该消息队列中的消息，而且此时消息队列未满[L6.23(5)]，指向该消息的指针被插入到消息队列中[L6.23(6)]。这样，下一个调用 OSQPend()函数的任务就可以马上得到该消息。注意，如果此时消息队列已满，那么该消息将由于不能插入到消息队列中而丢失。

此外，如果 OSQPost()函数是由中断服务子程序调用的，那么即使产生了更高优先级的任务，也不会在调用 OSSched()函数时发生任务切换。这个动作一直要等到中断嵌套的最外层中断服务子程序调用 OSIntExit()函数时才能进行（见 3.09 节，μC/OS-II 中的中断）。

6.7.4 向消息队列发送一个消息（后进先出 LIFO），OSQPostFront()

OSQPostFront()函数和 OSQPost()基本上是一样的，只是在插入新的消息到消息队列中时，使用.OSQOut 作为指向下一个插入消息的单元的指针，而不是.OSQIn。程序清单 L6.24 是它的源代码。值得注意的是，.OSQOut 指针指向的是已经插入了消息指针的单元，所以再插入新的消息指针前，必须先将.OSQOut 指针在消息队列中前移一个单元。如果.OSQOut 指针指向的当前单元是队列中的第一个单元[L6.24(1)]，这时再前移就会发生越界，需要特别地将该指针指向队列的末尾[L6.24(2)]。由于.OSQEnd 指向的是消息队列中最后一个单元的下一个单元，因此.OSQOut 必须被调整到指向队列的有效范围内[L6.24(3)]。因为 QSQPend()函数取出的消息是由 OSQPend()函数刚刚插入的，因此 OSQPostFront()函数实现了一个 LIFO 队列。

程序清单 L6.24 向消息队列发送一条消息（LIFO）

```
INT8U OSQPostFront (OS_EVENT *pevent, void *msg)
{
    OS_Q   *pq;



    OS_ENTER_CRITICAL();
    if (pevent->OSEventType != OS_EVENT_TYPE_Q) {
        OS_EXIT_CRITICAL();
        return (OS_ERR_EVENT_TYPE);
    }
    if (pevent->OSEventGrp) {
        OSEventTaskRdy(pevent, msg, OS_STAT_Q);
        OS_EXIT_CRITICAL();
        OSSched();
        return (OS_NO_ERR);
    } else {
        pq = pevent->OSEventPtr;
        if (pq->OSQEntries >= pq->OSQSize) {
            OS_EXIT_CRITICAL();
            return (OS_Q_FULL);
        } else {
            if (pq->OSQOut == pq->OSQStart) {                    (1)
                pq->OSQOut = pq->OSQEnd;                         (2)
            }
            pq->OSQOut--;                                        (3)
            *pq->OSQOut = msg;
```

```
            pq->OSQEntries++;
            OS_EXIT_CRITICAL();
        }
        return (OS_NO_ERR);
    }
}
```

6.7.5 无等待地从一个消息队列中取得消息，OSQAccept()

如果试图从消息队列中取出一条消息，而此时消息队列又为空时，也可以不让调用任务等待而直接返回调用函数。这个操作可以调用 OSQAccept()函数来完成。程序清单 L6.25 是该函数的源代码。OSQAccept()函数首先查看 pevent 指向的事件控制块是否是由 OSQCreate()函数建立的[L6.25(1)]，然后它检查当前消息队列中是否有消息[L6.25(2)]。如果消息队列中有至少一条消息，那么就从.OSQOut 指向的单元中取出消息[L6.25(3)]。OSQAccept()函数的调用函数需要对 OSQAccept()返回的指针进行检查。如果该指针是 NULL 值，说明消息队列是空的，其中没有消息可以 [L6.25(4)]。否则的话，说明已经从消息队列中成功地取得了一条消息。当中断服务子程序要从消息队列中取消息时，必须使用 OSQAccept()函数，而不能使用 OSQPend()函数。

程序清单 L6.25 无等待地从消息队列中取一条消息

```
void *OSQAccept (OS_EVENT *pevent)
{
    void  *msg;
    OS_Q  *pq;


    OS_ENTER_CRITICAL();
    if (pevent->OSEventType != OS_EVENT_TYPE_Q) {                    (1)
        OS_EXIT_CRITICAL();
        return ((void *)0);
    }
    pq = pevent->OSEventPtr;
    if (pq->OSQEntries != 0) {                                       (2)
        msg = *pq->OSQOut++;                                         (3)
        pq->OSQEntries--;
        if (pq->OSQOut == pq->OSQEnd) {
            pq->OSQOut = pq->OSQStart;
        }
```

```
    } else {
        msg = (void *)0;                                                    (4)
    }
    OS_EXIT_CRITICAL();
    return (msg);
}
```

6.7.6 清空一个消息队列，OSQFlush()

OSQFlush()函数允许用户删除一个消息队列中的所有消息，重新开始使用。程序清单 L6.26 是该函数的源代码。和前面的其它函数一样，该函数首先检查 pevent 指针是否是执行一个消息队列[L6.26(1)]，然后将队列的插入指针和取出指针复位，使它们都指向队列起始单元，同时，将队列中的消息数设为 0 [L6.26(2)]。这里，没有检查该消息队列的等待任务列表是否为空，因为只要该等待任务列表不空，.OSQEntries 就一定是 0。唯一不同的是，指针.OSQIn 和.OSQOut 此时可以指向消息队列中的任何单元，不一定是起始单元。

程序清单 L6.26 清空消息队列

```
INT8U OSQFlush (OS_EVENT *pevent)
{
    OS_Q  *pq;



    OS_ENTER_CRITICAL();
    if (pevent->OSEventType != OS_EVENT_TYPE_Q) {                           (1)
        OS_EXIT_CRITICAL();
        return (OS_ERR_EVENT_TYPE);
    }
    pq             = pevent->OSEventPtr;
    pq->OSQIn      = pq->OSQStart;                                          (2)
    pq->OSQOut     = pq->OSQStart;
    pq->OSQEntries = 0;
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);
}
```

6.7.7 查询一个消息队列的状态，OSQQuery()

OSQQuery()函数使用户可以查询一个消息队列的当前状态。程序清单 L6.27 是该函数的源代码。OSQQuery()需要两个参数：一个是指向消息队列的指针 pevent。它是在建立一个消息队列时，由 OSQCreate()函数返回的；另一个是指向 OS_Q_DATA（见 uCOS_II.H）数据结构的指针 pdata。该结构包含了有关消息队列的信息。在调用 OSQQuery()函数之前，必须先定义该数据结构变量。OS_Q_DATA 结构包含下面的几个域：

.OSMsg 如果消息队列中有消息，它包含指针.OSQOut 所指向的队列单元中的内容。如果队列是空的，.OSMsg 包含一个 NULL 指针。

.OSNMsgs 是消息队列中的消息数（.OSQEntries 的拷贝）。

.OSQSize 是消息队列的总的容量

.OSEventTbl[]和**.OSEventGrp** 是消息队列的等待任务列表。通过它们， OSQQuery()的调用函数可以得到等待该消息队列中的消息的任务总数。

OSQQuery()函数首先检查 pevent 指针指向的事件控制块是一个消息队列[L6.27(1)]，然后复制等待任务列表[L6.27(2)]。如果消息队列中有消息[L6.27(3)]，.OSQOut 指向的队列单元中的内容被复制到 OS_Q_DATA 结构中[L6.27(4)]，否则的话，就复制一个 NULL 指针[L6.27(5)]。最后，复制消息队列中的消息数和消息队列的容量大小[L6.27(6)]。

程序清单 L6.27 程序消息队列的状态

```
INT8U OSQQuery (OS_EVENT *pevent, OS_Q_DATA *pdata)
{
    OS_Q   *pq;
    INT8U   i;
    INT8U  *psrc;
    INT8U  *pdest;



    OS_ENTER_CRITICAL();
    if (pevent->OSEventType != OS_EVENT_TYPE_Q) {                    (1)
        OS_EXIT_CRITICAL();
        return (OS_ERR_EVENT_TYPE);
    }
    pdata->OSEventGrp = pevent->OSEventGrp;                          (2)
    psrc              = &pevent->OSEventTbl[0];
    pdest             = &pdata->OSEventTbl[0];
    for (i = 0; i < OS_EVENT_TBL_SIZE; i++) {
```

```
        *pdest++ = *psrc++;
    }
    pq = (OS_Q *)pevent->OSEventPtr;
    if (pq->OSQEntries > 0) {                                    (3)
        pdata->OSMsg = pq->OSQOut;                               (4)
    } else {
        pdata->OSMsg = (void *)0;                                (5)
    }
    pdata->OSNMsgs = pq->OSQEntries;                             (6)
    pdata->OSQSize = pq->OSQSize;
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);
}
```

6.7.8 使用消息队列读取模拟量的值

在控制系统中，经常要频繁地读取模拟量的值。这时，可以先建立一个定时任务 OSTimeDly()[见 5.00 节，延时一个任务，OSTimeDly()]，并且给出希望的抽样周期。然后，如图 F6.11 所示，让 A/D 采样的任务从一个消息队列中等待消息。该程序最长的等待时间就是抽样周期。当没有其它任务向该消息队列中发送消息时，A/D 采样任务因为等待超时而退出等待状态并进行执行。这就模仿了 OSTimeDly()函数的功能。

也许，读者会提出疑问，既然 OSTimeDly()函数能完成这项工作，为什么还要使用消息队列呢？这是因为，借助消息队列，我们可以让其它的任务向消息队列发送消息来终止 A/D 采样任务等待消息，使其马上执行一次 A/D 采样。此外，我们还可以通过消息队列来通知 A/D 采样程序具体对哪个通道进行采样，告诉它增加采样频率等等，从而使得我们的应用更智能化。换句话说，我们可以告诉 A/D 采样程序，“现在马上读取通道 3 的输入值！”之后，该采样任务将重新开始在消息队列中等待消息，准备开始一次新的扫描过程。

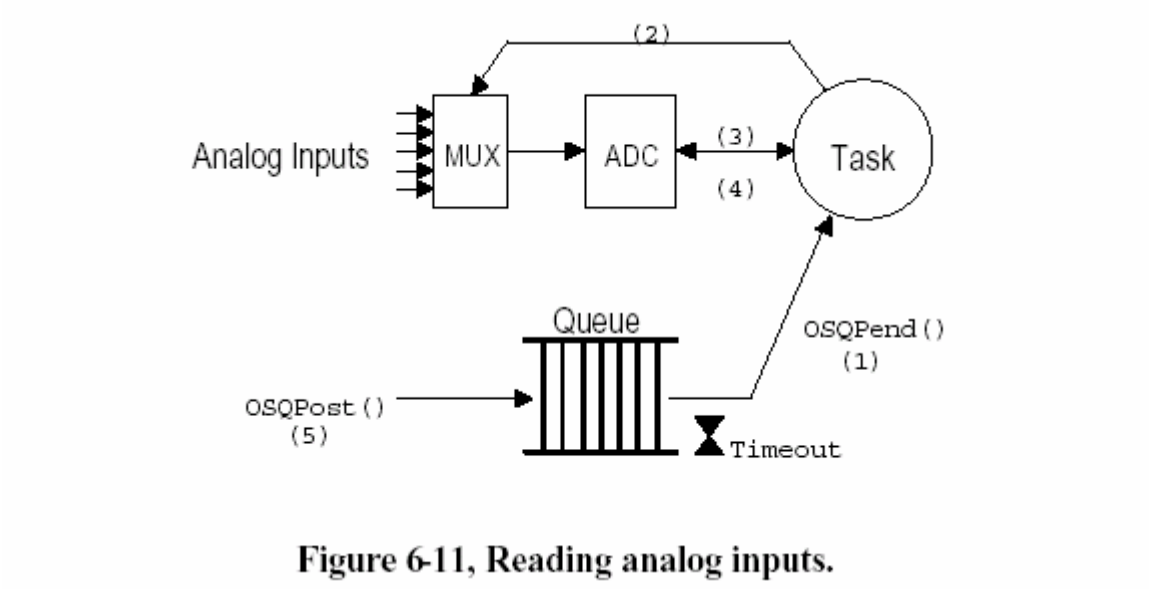

Figure 6-11, Reading analog inputs.

图 F6.11 读模拟量输入——Figure 6.11

6.7.9 使用一个消息队列作为计数信号量

在消息队列初始化时，可以将消息队列中的多个指针设为非 NULL 值（如 void* 1），来实现计数信号量的功能。这里，初始化为非 NULL 值的指针数就是可用的资源数。系统中的任务可以通过 OSQPend()来请求“信号量”，然后通过调用 OSQPost()来释放“信号量”，如程序清单 L6.28。如果系统中只使用了计数信号量和消息队列，使用这种方法可以有效地节省代码空间。这时将 OS_SEM_EN 设为 0，就可以不使用信号量，而只使用消息队列。值得注意的是，这种方法为共享资源引入了大量的指针变量。也就是说，为了节省代码空间，牺牲了 RAM 空间。另外，对消息队列的操作要比对信号量的操作慢，因此，当用计数信号量同步的信号量很多时，这种方法的效率是非常低的。

程序清单 L6.28 使用消息队列作为一个计数信号量

```
OS_EVENT *QSem;
void     *QMsgTbl[N_RESOURCES]


void main (void)
{
    OSInit();
    .
```

```
    .
    QSem = OSQCreate(&QMsgTbl[0], N_RESOURCES);
    for (i = 0; i < N_RESOURCES; i++) {
        OSQPost(Qsem, (void *)1);
    }
    .
    .
    OSTaskCreate(Task1, .., .., ..);
    .
    .
    OSStart();
}


void Task1 (void *pdata)
{
    INT8U err;


    for (;;) {
        OSQPend(&QSem, 0, &err);          /* 得到对资源的访问权  */
        .
        .    /* 任务获得信号量,对资源进行访问  */
        .
        OSMQPost(QSem, (void*)1);        /* 释放对资源的访问权 */
    }
}
```