



The Real-Time Kernel

任务之间的通信与同步

事件控制块ECB

- 所有的通信信号都被看成是事件(event), μC/OS-II通过事件控制块(ECB)来管理每一个具体事件。

ECB数据结构

```
typedef struct {  
    void *OSEventPtr; /*指向消息或消息队列的指针*/  
    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：信号量、邮箱等*/  
    INT8U OSEventGrp; /*等待任务组*/  
} OS_EVENT;
```

与TCB类似的结构，使用两个链表，空闲链表与使用链表

事件控制块ECB数据结构



任务和ISR之间的通信方式

- 一个任务或ISR可以通过事件控制块ECB（信号量、邮箱或消息队列）向另外的任务发信号；
- 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。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；
- 多个任务可以同时等待同一个事件的发生。当该事件发生后，在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。

等待任务列表

- 每个正在等待某个事件的任务被加入到该事件的ECB的等待任务列表中，该列表包含两个变量OSEventGrp和OSEventTbl[]。
- 在OSEventGrp中，任务按优先级分组，8个任务为一组，共8组，分别对应OSEventGrp 当中的8位。当某组中有任务处于等待该事件的状态时，对应的位就被置位。同时，OSEventTbl[]中的相应位也被置位。

OSEventGrp

7	6	5	4	3	2	1	0
---	---	---	---	---	---	---	---

任务的优先级

0	0	Y	Y	Y	X	X	X
---	---	---	---	---	---	---	---

OSEventTbl[]中相应位的位置

OSEventGrp 中相应位的位置及
OSEventTbl[]中的数组下标

OSEventTbl [OS_LOWEST_PRIO / 8+1]

最高优先级任务

[0]	7	6	5	4	3	2	1	0
[1]	15	14	13	12	11	10	9	8
[2]	23	22	21	20	19	18	17	16
[3]	31	30	29	28	27	26	25	24
[4]	39	38	37	36	35	34	33	32
[5]	47	46	45	44	43	42	41	40
[6]	55	54	53	52	51	50	49	48
[7]	63	62	61	60	59	58	57	56

正在等待该事件的任务的优先级

最低优先级任务(即空闲任务, 不可能处于等待状态)

使任务进入/脱离等待状态

- 将一个任务插入到事件的等待任务列表中

```
pevent->OSEventGrp          |= 0SMapTbl[prio >> 3];  
pevent->OSEventTbl[prio >> 3] |= 0SMapTbl[prio & 0x07];
```

- 从等待任务列表中删除一个任务

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

在等待事件的任务列表中查找优先级最高的任务

- 在等待任务列表中查找最高优先级的任务

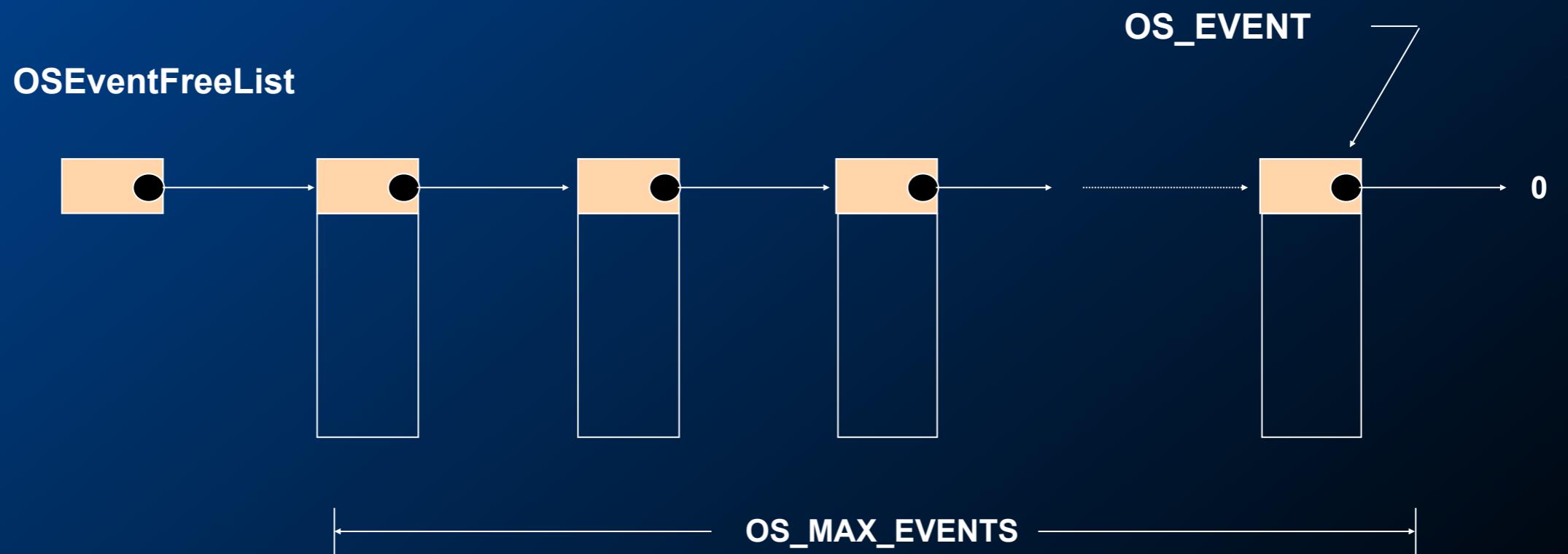
```
y      = OSUnMapTbl[pevent->OSEventGrp];
```

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

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

空闲ECB的管理

- ECB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，由OS_CFG.H中的#define OS_MAX_EVENTS定义。
- 在调用OSInit()初始化系统时，所有的ECB被链接成一个单向链表——空闲事件控制块链表；
- 每当建立一个信号量、邮箱或消息队列时，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。



同步与互斥

- 为了实现资源共享，一个操作系统必须提供临界区操作的功能；
- μC/OS采用关闭/打开中断的方式来处理临界区代码，从而避免竞争条件，实现任务间的互斥；
- μC/OS定义两个宏(macros)来开关中断，即：
OS_ENTER_CRITICAL()和OS_EXIT_CRITICAL();
- 这两个宏的定义取决于所用的微处理器，每种微处理器都有自己的OS_CPU.H文件。

任务1

```
...  
OS_ENTER_CRITICAL();  
任务1的临界区代码;  
OS_EXIT_CRITICAL();  
...
```

任务2

```
...  
OS_ENTER_CRITICAL();  
任务2的临界区代码;  
OS_EXIT_CRITICAL();  
...
```

临界资源

μ C/OS-II中开关中断的方法

- 当处理临界段代码时，需要关中断，处理完毕后，再开中断；
- 关中断时间是实时内核最重要的指标之一；
- 在实际应用中，关中断的时间很大程度上取决于微处理器的结构和编译器生成的代码质量；
-

μ C/OS-II中采用了3种开关中断的方法

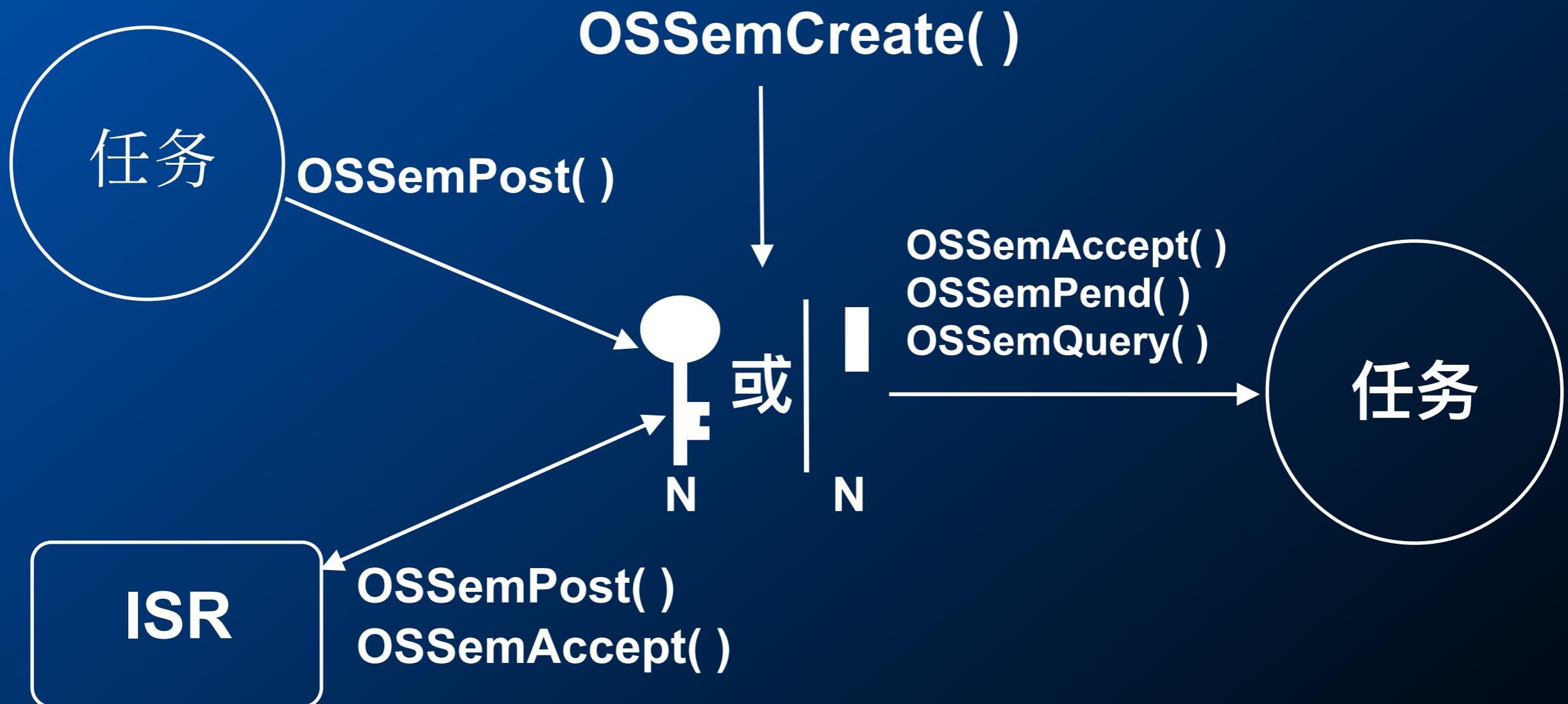
- OS_CRITICAL_METHOD==1
 - 用处理器指令关中断，执行OS_ENTER_CRITICAL(), 开中断执行OS_EXIT_CRITICAL();
- OS_CRITICAL_METHOD==2
 - 实现OS_ENTER_CRITICAL()时，先在堆栈中保存中断的开/关状态，然后再关中断；实现OS_EXIT_CRITICAL()时，从堆栈中弹出原来中断的开/关状态；
- OS_CRITICAL_METHOD==3
 - 把当前处理器的状态字保存在局部变量中（如OS_CPU_SR），关中断时保存，开中断时恢复

信号量

- 信号量在多任务系统中的功能
 - 实现对共享资源的互斥访问（包括单个共享资源或多个相同的资源）；
 - 实现任务之间的行为同步；
- 必须在OS_CFG.H中将OS_SEM_EN开关常量置为1，这样μC/OS才能支持信号量。

- uC/OS中信号量由两部分组成：信号量的计数值（16位无符号整数）和等待该信号量的任务所组成的等待任务表；
- 信号量系统服务
 - OSSemCreate()
 - OSSemPend(), OSSemPost()
 - OSSemAccept(), OSSemQuery()

任务、ISR和信号量的关系



创建一个信号量

- OSSemCreate()
 - 创建一个信号量，并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；
 - OS_EVENT *OSSemCreate(INT16U cnt);
 - cnt：信号量的初始值。
- 执行步骤
 - 从空闲事件控制块链表中得到一个ECB；
 - 初始化ECB，包括设置信号量的初始值、把等待任务列表清零、设置ECB的事件类型等；
 - 返回一个指向该事件控制块的指针。

等待一个信号量

- OSSemPend()
 - 等待一个信号量，即操作系统中的P操作，将信号量的值减1；
 - OSSemPend (OS_EVENT *pevent,
INT16U timeout, INT8U *err);
- 执行步骤
 - 如果信号量的计数值大于0，将它减1并返回；
 - 如果信号量的值等于0，则调用本函数的任务将被阻塞起来，等待另一个任务把它唤醒；
 - 调用OSSched()函数，调度下一个最高优先级的任务运行。

发送一个信号量

- OSSemPost()
 - 发送一个信号量，即操作系统中的V操作，将信号量的值加1；
 - OSSemPost (OS_EVENT *pevent);
- 执行步骤
 - 检查是否有任务在等待该信号量，如果没有，将信号量的计数值加1并返回；
 - 如果有，将优先级最高的任务从等待任务列表中删除，并使它进入就绪状态；
 - 调用OSSched(), 判断是否需要进行任务切换。

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，则让该任务简单地返回，而不是进入等待状态；
 - INT16U OSSemAccept(OS_EVENT *pevent);
- 执行步骤
 - 如果该信号量的计数值大于0，则将它减1，然后将信号量的原有值返回；
 - 如果该信号量的值等于0，直接返回该值(0)。

查询一个信号量的当前状态

- OSSemQuery()
 - 查询一个信号量的当前状态；
 - INT8U OSSemQuery(OS_EVENT *pevent,
OS_SEM_DATA *pdata);
 - 将指向信号量对应事件控制块的指针pevent所指向的ECB的内容拷贝到指向用于记录信号量信息的数据结构OS_SEM_DATA数据结构的指针pdata所指向的缓冲区当中。

任务间通信

- 低级通信
 - 只能传递状态和整数值等控制信息，传送的信息量小；
 - 例如：信号量
- 高级通信
 - 能够传送任意数量的数据；
 - 例如：共享内存、邮箱、消息队列

共享内存

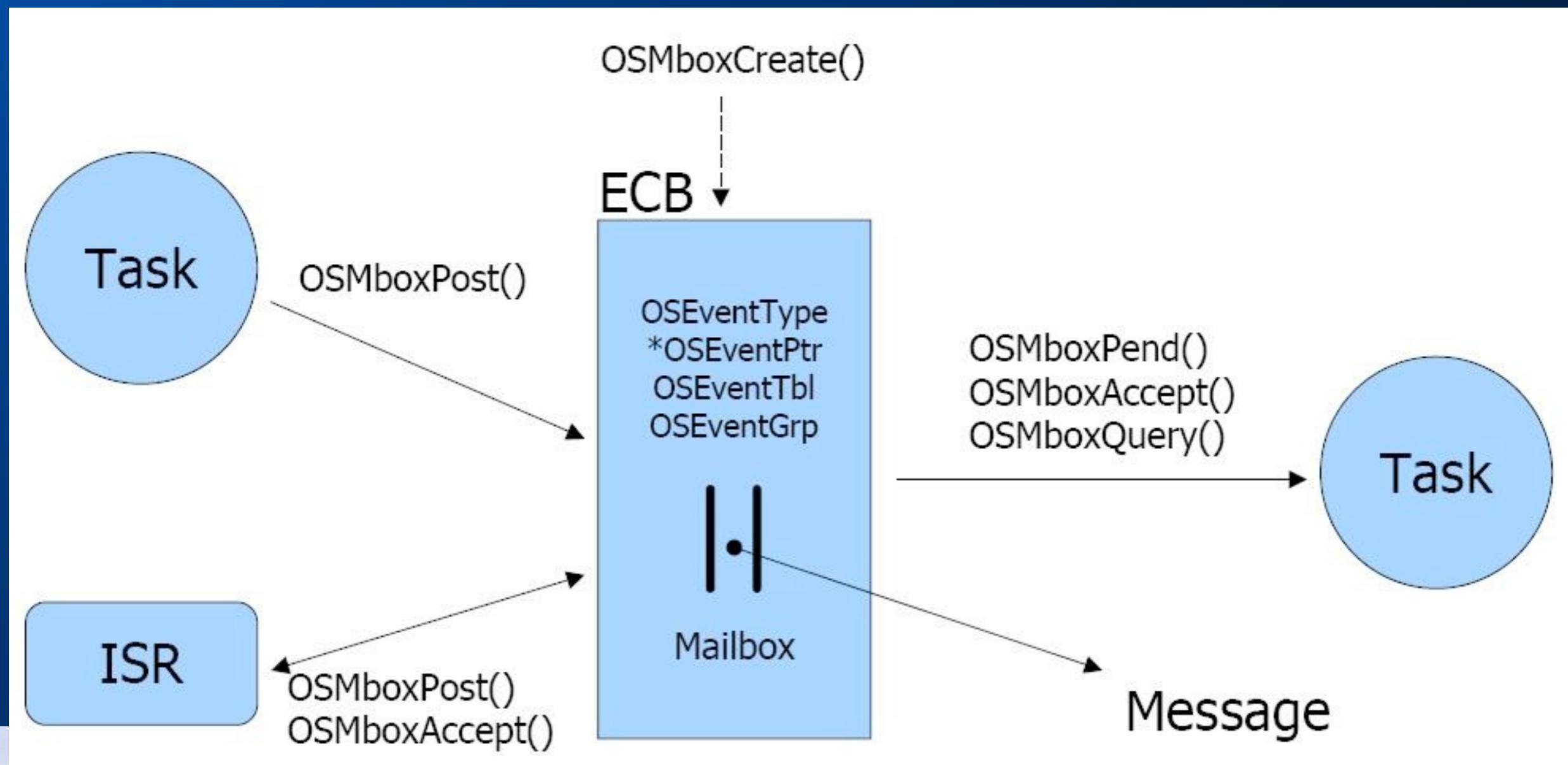
- 在μC/OS-II中如何实现共享内存?
 - 内存地址空间只有一个，为所有的任务所共享！
 - 为了避免竞争状态，需要使用信号量来实现互斥访问。

消息邮箱

- 邮箱（MailBox）：一个任务或ISR可以通过邮箱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”（message）的数据结构；
- 必须在OS_CFG.H中将OS_MBOX_EN开关常量置为1，这样μC/OS才能支持邮箱。

- 一个邮箱可能处于两种状态：
 - 满的状态： 邮箱包含一个非空指针型变量；
 - 空的状态： 邮箱的内容为空指针NULL；
- 邮箱的系统服务
 - OSMboxCreate()
 - OSMboxPost()
 - OSMboxPend()
 - OSMboxAccept()
 - OSMboxQuery()

任务、ISR和消息邮箱的关系



邮箱的系统服务 (I)

- OSMboxCreate(): 创建一个邮箱
 - 在创建邮箱时，须分配一个ECB，并使用其中的字段OSEventPtr指针来存放消息的地址；
 - OS_EVENT *OSMboxCreate(void *msg);
 - msg: 指针的初始值，一般情形下为NULL。
- OSMboxPend(): 等待一个邮箱中的消息
 - 若邮箱为满，将其内容（某消息的地址）返回；若邮箱为空，当前任务将被阻塞，直到邮箱中有了消息或等待超时；
 - OSMboxPend (OS_EVENT *pevent,
INT16U timeout, INT8U *err);

邮箱的系统服务 (2)

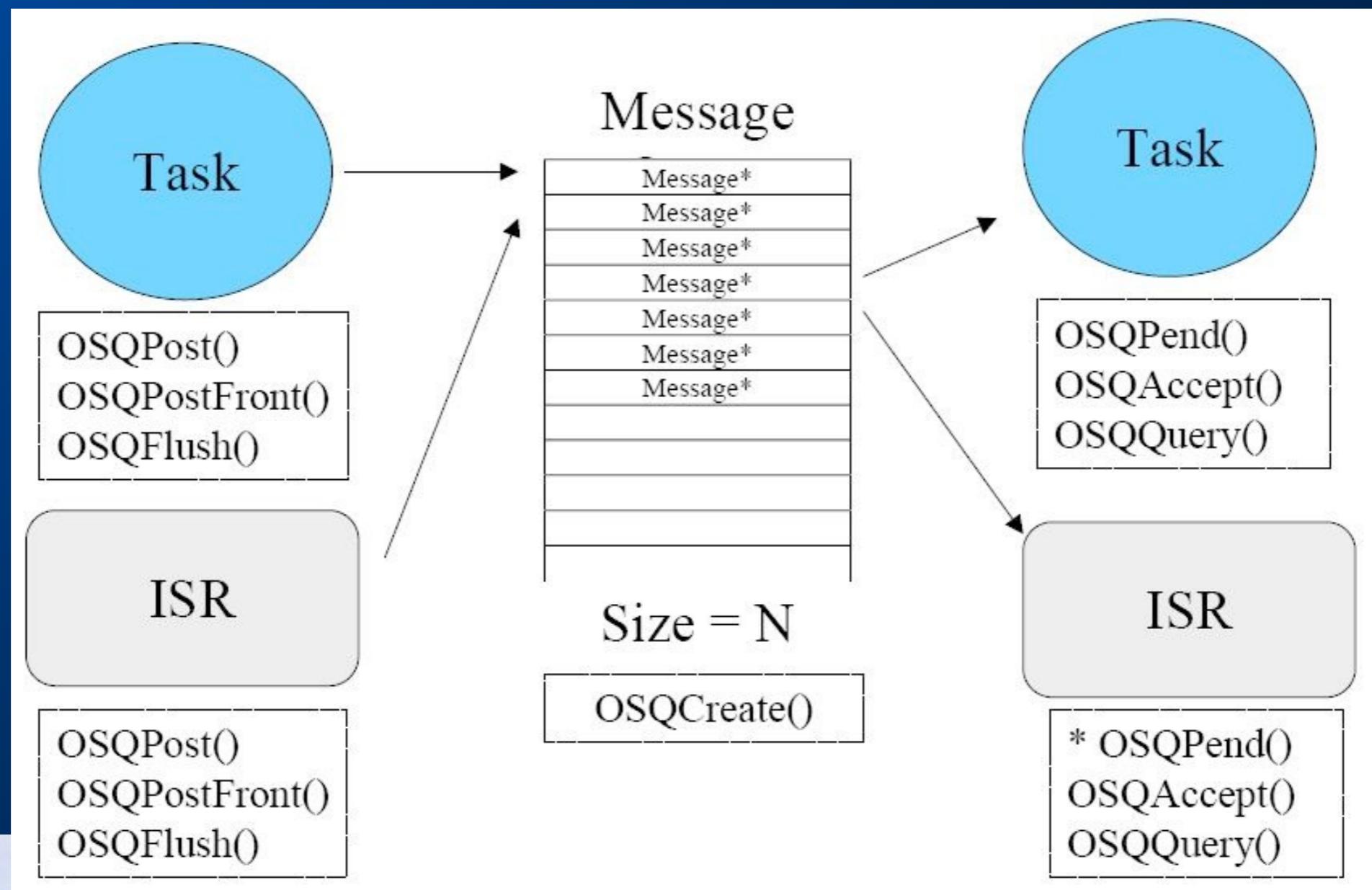
- OSMboxPost(): 发送一个消息到邮箱中
 - 如果有任务在等待该消息，将其中的最高优先级任务从等待列表中删除，变为就绪状态；
 - OSMboxPost(OS_EVENT *pevent, void *msg);
- OSMboxAccept(): 无等待地请求邮箱消息
 - 若邮箱为满，返回它的当前内容；若邮箱为空，返回空指针；
 - OSMboxAccept (OS_EVENT *pevent);
- OSMboxQuery(): 查询一个邮箱的状态
 - OSMboxQuery (OS_EVENT *pevent, OS_MBOX_DATA *pdata);

消息队列

- 消息队列（Message Queue）：消息队列可以使一个任务或ISR向另一个任务发送多个以指针方式定义的变量；
- 为了使μC/OS能够支持消息队列，必须在OS_CFG.H中将OS_Q_EN开关常量置为1，并且通过常量OS_MAX_QS来决定系统支持的最多消息队列数。

- 一个消息队列可以容纳多个不同的消息，因此可把它看作是由多个邮箱组成的数组，只是它们共用一个等待任务列表：
- 消息队列的系统服务
 - OSQCreate()
 - OSQPend()、OSQAccept()
 - OSQPost()、OSQPostFront()
 - OSQFlush()
 - OSQQuery()

消息队列的体系结构



回忆一下ECB数据结构

- 在实现消息队列时，哪些字段可以用？

ECB数据结构

```
typedef struct {

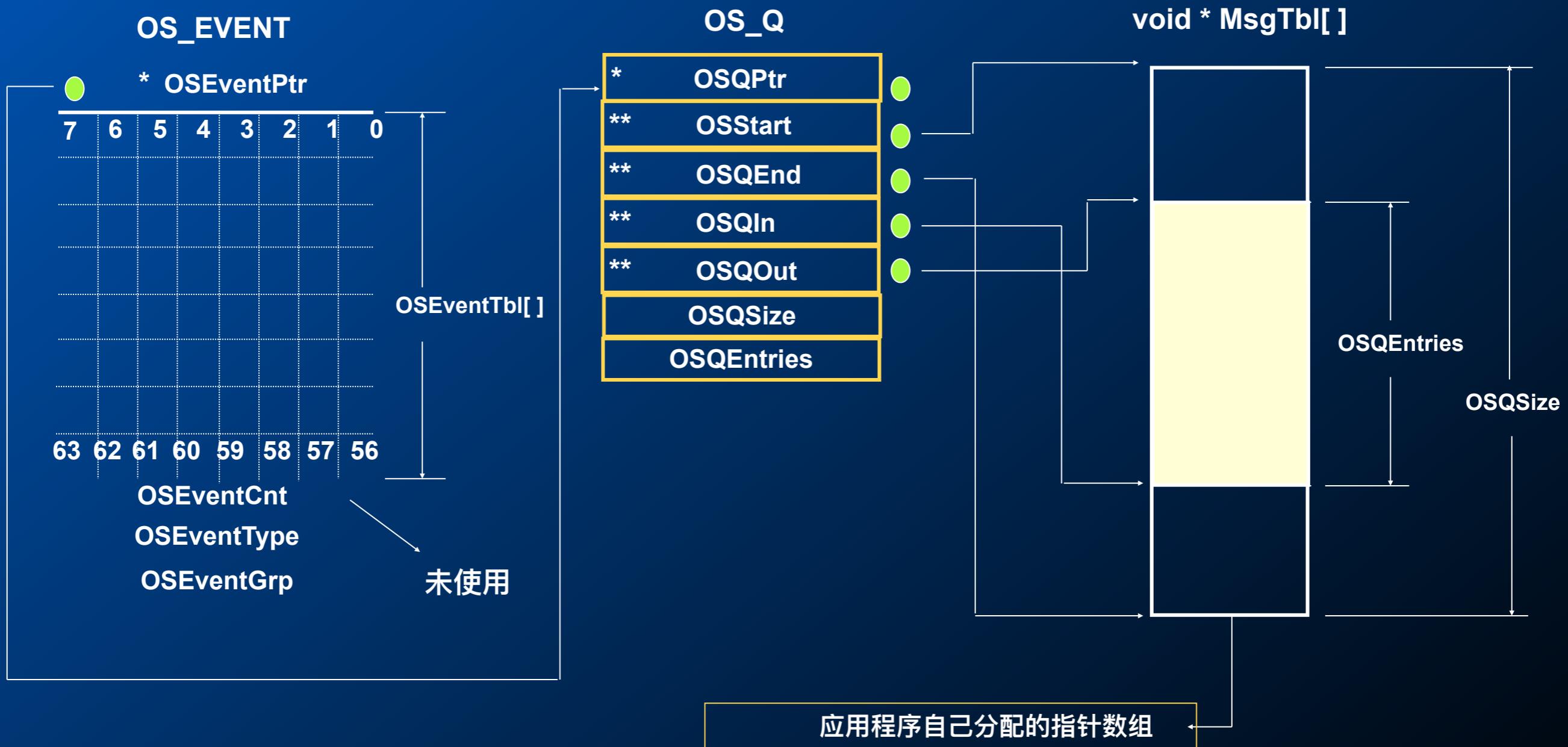
    void *OSEventPtr; /*指向消息或消息队列的指针*/
    INT8U OSEventTbl[OS_EVENT_TBL_SIZE]; //等待任务列表
    INT16U OSEventCnt; /*计数器（当事件是信号量时）*/
    INT8U OSEventType; /*事件类型：信号量、邮箱等*/
    INT8U OSEventGrp; /*等待任务组*/
} OS_EVENT;
```

队列控制块

- 队列控制块数据结构

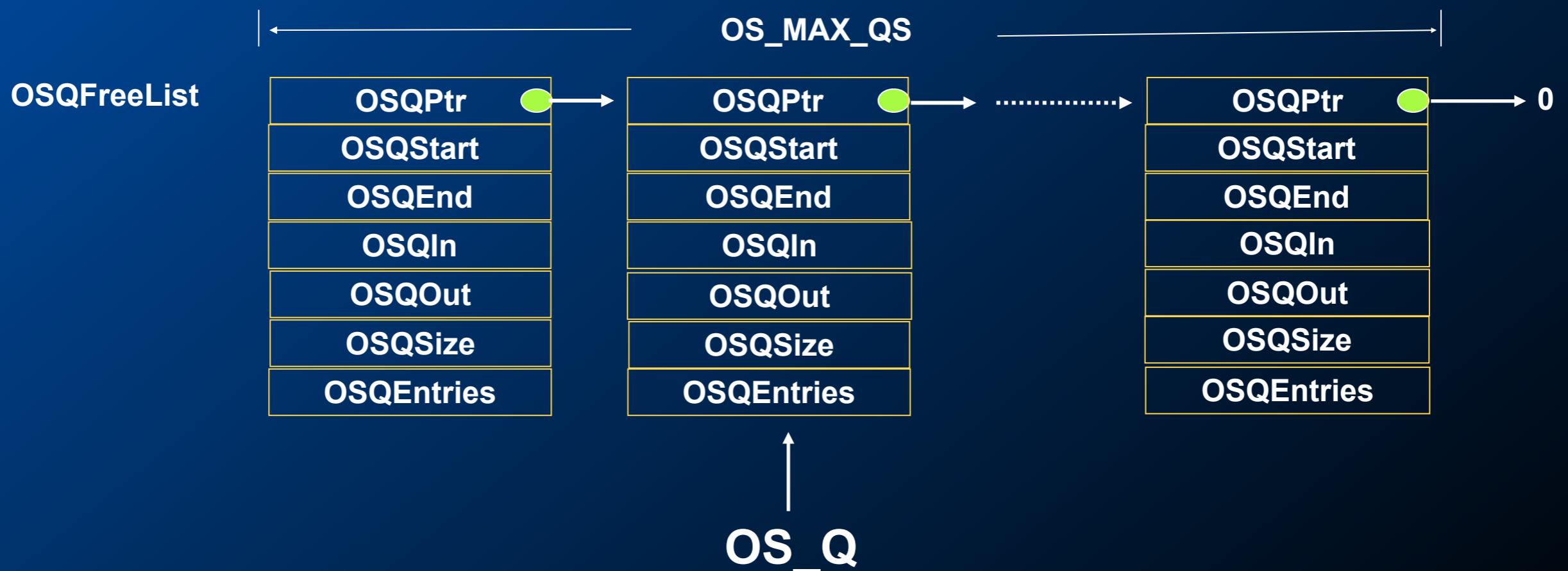
```
typedef struct os_q {  
    struct os_q *OSQPtr;//空闲队列控制块指针  
    void **OSQStart; //指向消息队列的起始地址  
    void **OSQEnd; //指向消息队列的结束地址  
    void **OSQIn; //指向消息队列中下一个插入消息的位置  
    void **OSQOut;//指向消息队列中下一个取出消息的位置  
    INT16U OSQSize; //消息队列中总的单元数  
    INT16U OSQEntries; //消息队列中当前的消息数量  
} OS_EVENT;
```

消息队列的数据结构

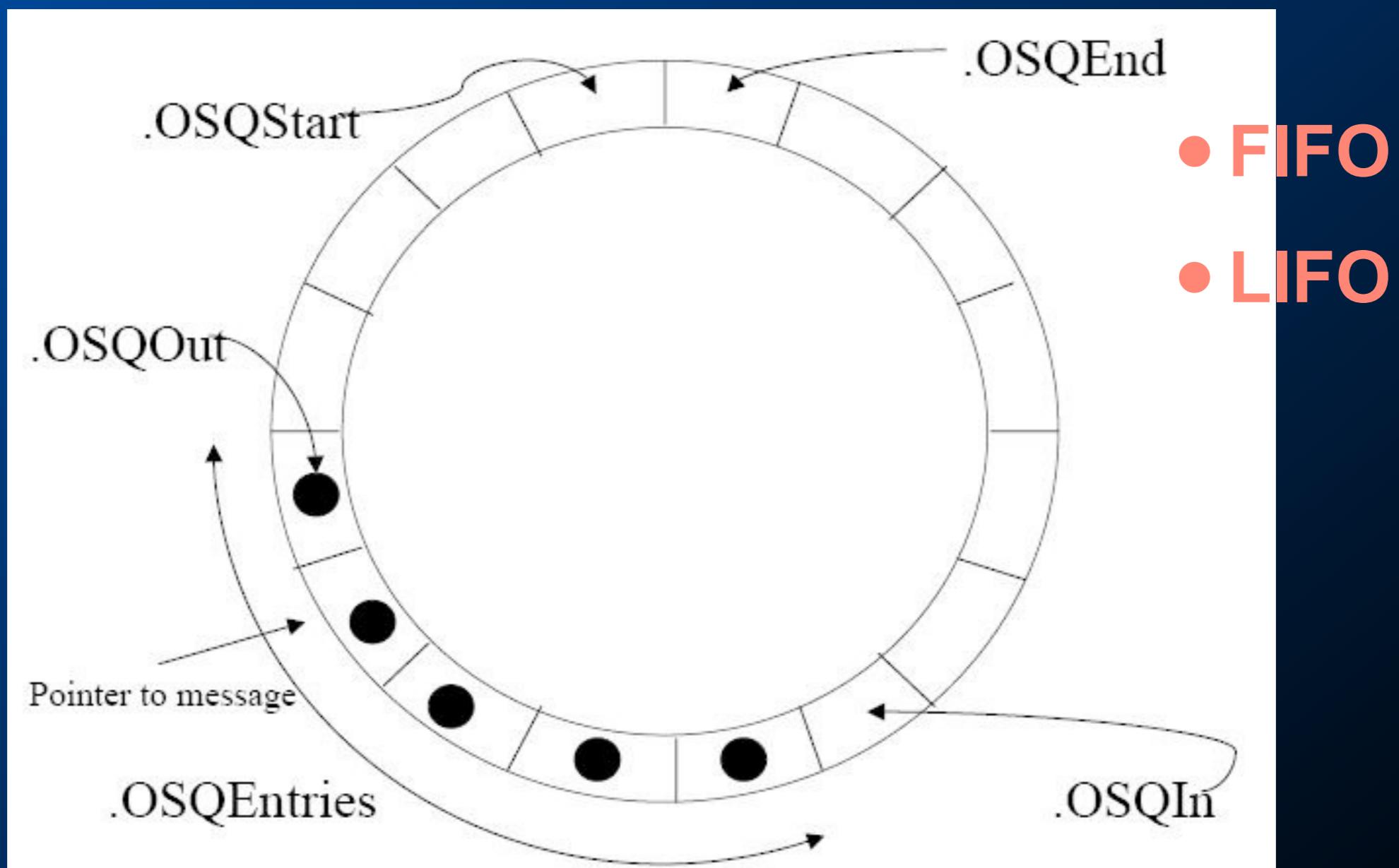


空闲队列控制块的管理

- 每一个消息队列都要用到一个队列控制块。在μC/OS中，队列控制块的总数由 OS_CFG.H 中的常量 OS_MAX_QS 定义。
- 在系统初始化时，所有的队列控制块被链接成一个单向链表——空闲队列控制块链表 OSQFreeList。



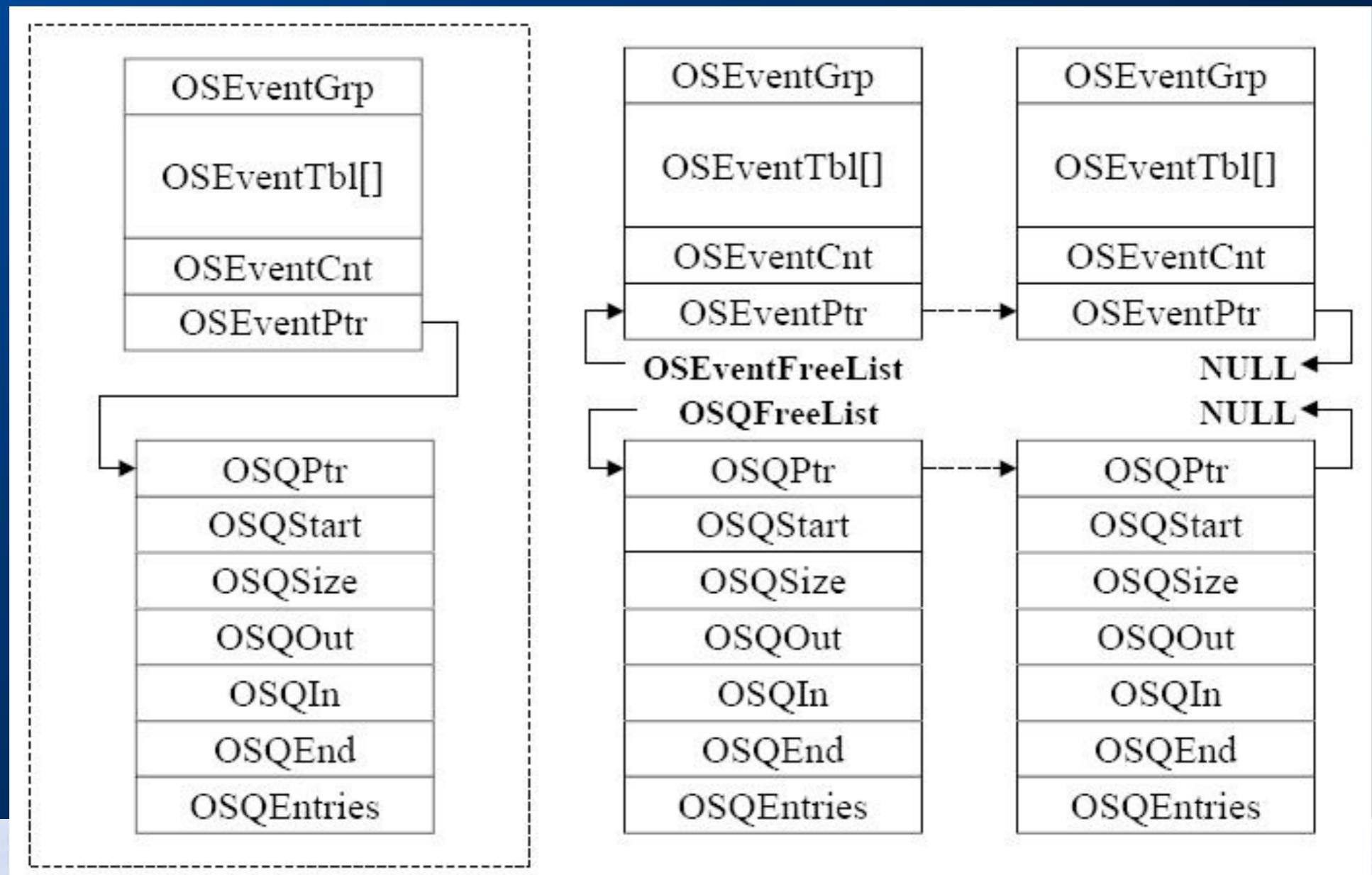
消息缓冲区



创建一个消息队列

- OSQCreate()
 - OS_EVENT *OSQCreate (void **start,
INT16U size);
 - start: 指针数组, 用来存放各个消息的地址
 - size: 数组的大小 (即消息队列的元素个数)
- 执行步骤
 - 从空闲事件控制块链表中取得一个ECB;
 - 从空闲队列控制块列表中取出一个队列控制块, 并对其进行初始化;
 - 初始化ECB的内容 (事件类型、等待任务列表), 并将OSEventPtr指针指向队列控制块。

队列控制块与事件控制块



请求消息队列中的消息

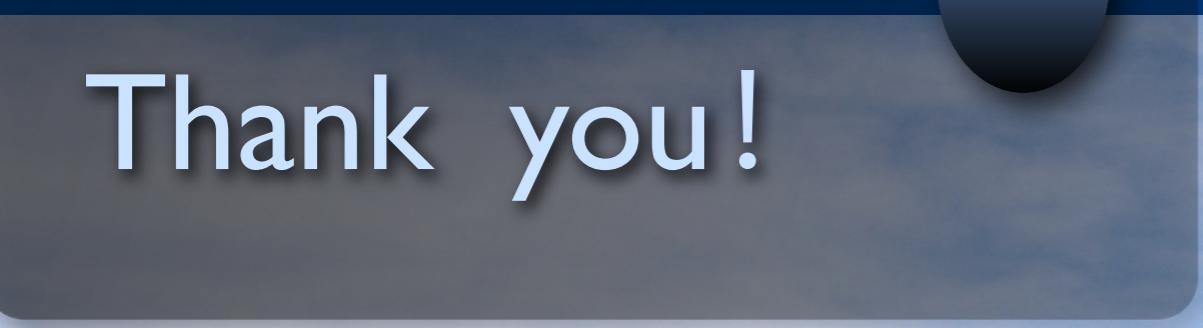
- OSQPend(): 等待一个消息队列中的消息
 - `void *OSQPend (OS_EVENT *pevent,
INT16U timeout, INT8U *err);`
 - 如果消息队列中有至少一条消息，返回消息的地址；
 - 如果没有消息，相应任务进入等待状态。
- OSQAccept(): 无等待地请求消息队列中的消息
 - `void *OSQAccept(OS_EVENT *pevent);`
 - 如果消息队列中有消息，返回消息的地址；
 - 如果消息队列中没有消息，返回NULL。

向消息队列发送一个消息

- OSQPost(): 以FIFO方式向消息队列发送一个消息
 - INT8U OSQPost (OS_EVENT *pevent,
void *msg);
 - 如果有任务在等待该消息队列，唤醒其中优先级最高的任务，并重新调度；
 - 如果没有任务在等待该消息队列，而且此时消息队列未满，则以FIFO方式插入这个消息。
- OSQPostFront(): 以LIFO方式向消息队列发送一个消息
 - INT8U OSQPostFront(OS_EVENT *pevent, void *msg);

清空操作与查询操作

- OSQFlush(): 清空一个消息队列
 - INT8U OSQFlush (OS_EVENT *pevent);
 - 删除一个消息队列中的所有消息；
- OSQuery(): 查询一个消息队列的状态
 - INT8U OSQuery (OS_EVENT *pevent, OS_Q_DATA *pdata);



Thank you!

