Message Mailbox Management

Mailbox Configuration

A message mailbox (or simply a mailbox) is a µC/OS-II object that allows a task or an ISR to send a pointer-sized variable to another task. The pointer is typically initialized to point to some application specific data structure containing a “message.” µC/OS-II provides seven services to access mailboxes: OSMboxCreate(), OSMboxDel(), OSMboxPend(), OSMboxPost(), OSMboxPostOpt(), OSMboxAccept(), and OSMboxQuery().

To enable µC/OS-II message mailbox services, you must set configuration constants in OS_CFG.H. Specifically, table 10.1 shows which services are compiled based on the value of configuration constants found in OS_CFG.H. You should note that NONE of the mailbox services are enabled when OS_MBOX_EN is set to 0. To enable specific features (i.e. service) listed in Table 10.1, simply set the configuration constant to 1. You will notice that OSMboxCreate() and OSMboxPend() cannot be individually disabled like the other services. That’s because they are always needed when you enable µC/OS-II message mailbox management. You must enable at least one of the post services: OSMboxPost() and OSMboxPostOpt().

<table>
<thead>
<tr>
<th>µC/OS-II Mailbox Service</th>
<th>Enabled when set to 1 in OS_CFG.H</th>
</tr>
</thead>
<tbody>
<tr>
<td>OSMboxAccept()</td>
<td>OS_MBOX_ACCEPT_EN</td>
</tr>
<tr>
<td>OSMboxCreate()</td>
<td></td>
</tr>
<tr>
<td>OSMboxDel()</td>
<td>OS_MBOX_DEL_EN</td>
</tr>
<tr>
<td>OSMboxPend()</td>
<td></td>
</tr>
<tr>
<td>OSMboxPost()</td>
<td>OS_MBOX_POST_EN</td>
</tr>
<tr>
<td>OSMboxPostOpt()</td>
<td>OS_MBOX_POST_OPT_EN</td>
</tr>
<tr>
<td>OSMboxQuery()</td>
<td>OS_MBOX_QUERY_EN</td>
</tr>
</tbody>
</table>

Table - Table 10.1 Mailbox configuration constants in OS_CFG.H.

Figure 10.1 shows a flow diagram to illustrate the relationship between tasks, ISRs, and a message mailbox. Note that the symbology used to represent a mailbox is an I-beam. The
hourglass represents a timeout that can be specified with the `OSMboxPend()` call. The content of the mailbox is a pointer to a message. What the pointer points to is application specific. A mailbox can only contain one pointer (mailbox is full) or a pointer to NULL (mailbox is empty).

As you can see from Figure 10.1, a task or an ISR can call `OSMboxPost()` or `OSMboxPostOpt()`. However, only tasks are allowed to call `OSMboxDel()`, `OSMboxPend()` and `OSMboxQuery()`. Your application can have just about any number of mailboxes. The limit is set by `OS_MAX_EVENTS` in `OS_CFG.H`.

![Figure 10.1 Relationships between tasks, ISRs, and a message mailbox.](image)

**Creating a Mailbox, OSMboxCreate()**

A mailbox needs to be created before it can be used. Creating a mailbox is accomplished by calling `OSMboxCreate()` and specifying the initial value of the pointer. Typically, the initial value is a NULL pointer, but a mailbox can initially contain a message. If you use the mailbox to signal the occurrence of an event (i.e., send a message), you typically initialize it to a NULL pointer because the event (most likely) has not occurred. If you use the mailbox to access a shared resource, you initialize the mailbox with a non-NULL pointer. In this case, you basically use the mailbox as a binary semaphore.

The code to create a mailbox is shown in Listing 10.1.
OS_EVENT *OsmboxCreate (void *msg)
{
    #if OS_CRITICAL_METHOD == 3
        OS_CPU_SR cpu_sr;                                               (1)
    #endif
    OS_EVENT *pevent;
    if (OSIntNesting > 0) {                                          (2)
        return ((OS_EVENT *)0);
    }
    OS_ENTER_CRITICAL();
    pevent = OSEventFreeList;                                        (3)
    if (OSEventFreeList != (OS_EVENT *)0) {                          (4)
        OSEventFreeList = (OS_EVENT *)OSEventFreeList->OSEventPtr;   (5)
    }
    OS_EXIT_CRITICAL();
    if (pevent != (OS_EVENT *)0) {                                   (6)
        pevent->OSEventType = OS_EVENT_TYPE_MBOX;                    (7)
        pevent->OSEventCnt  = 0;                                     (8)
        pevent->OSEventPtr  = msg;                                   (9)
        OS_EventWaitListInit(pevent);                               (10)
    }
    return (pevent);                                                (11)
}

Listing - Listing 10.1 Creating a mailbox.

(1) A local variable called cpu_sr to support OS_CRITICAL_METHOD #3 is allocated.

(2) OsmboxCreate() starts by making sure you are not calling this function from an ISR because this is not allowed. All kernel objects need to be created from task level code or before multitasking starts.

(3) OsmboxCreate() then attempts to obtain an ECB (Event Control Block) from the free list of ECBs (see Figure 6.5).

(4)

(5) The linked list of free ECBs is adjusted to point to the next free ECB.

(6)

(7) If there is an ECB available, the ECB type is set to OS_EVENT_TYPE_MBOX. Other Osmbox???() function calls will check this structure member to make sure that the ECB is of the proper type (i.e. a mailbox). This prevents you from calling OsmboxPost() on an ECB that was created for use as a message queue.
(8) The .OSEventCnt field is then initialized to zero since this field is not used by message mailboxes.

(9) The initial value of the message is stored in the ECB.

(10) The wait list is then initialized by calling OS_EventWaitListInit() [see 6.??, Initializing an ECB, OS_EventWaitListInit()]. Because the mailbox is being initialized, there are no tasks waiting for it and thus, OS_EventWaitListInit() clears the .OSEventGrp and .OSEventTbl[] fields of the ECB.

(11) Finally, OSMboxCreate() returns a pointer to the ECB. This pointer must be used in subsequent calls to manipulate mailboxes [OSMboxAccept(), OSMboxDel(), OSMboxPend(), OSMboxPost(), OSMboxPostOpt() and OSMboxQuery()]. The pointer is basically used as the mailbox handle. If there are no more ECBs, OSMboxCreate() returns a NULL pointer. You should make it a habit to check return values to ensure that you are getting the desired results. Passing NULL pointers to µC/OS-II will not make it fail because µC/OS-II validates arguments (only if OS_ARG_CHK_EN is set to 1, though). Figure 10.2 shows the content of the ECB just before OSMboxCreate() returns.

---

**Figure - Figure 10.2 ECB just before OSMboxCreate() returns.**
Deleting a Mailbox, OSMboxDel()

The code to delete a mailbox is shown in listing 10.2 and this code will only be generated by the compiler if OS_MBOX_DEL_EN is set to 1 in OS_CFG.H. This is a function you must use with caution because multiple tasks could attempt to access a deleted mailbox. You should always use this function with great care. Generally speaking, before you would delete a mailbox, you would first delete all the tasks that can access the mailbox.
OS_EVENT  *OSmboxDel (OS_EVENT *pevent, INT8U opt, INT8U *err)
{
    #if OS_CRITICAL_METHOD == 3
        OS_CPU_SR  cpu_sr;
    #endif
    BOOLEAN    tasks_waiting;

    if (OSIntNesting > 0) {                                   (1)
        *err = OS_ERR_DEL_ISR;
        return (pevent);
    }

    if (OS_ARG_CHK_EN > 0) {                                  (2)
        if (pevent == (OS_EVENT *)0) {
            *err = OS_ERR_PEVENT_NULL;
            return (pevent);
        }
        if (pevent->OSEventType != OS_EVENT_TYPE_MBOX) {      (3)
            *err = OS_ERR_EVENT_TYPE;
            return (pevent);
        }
    }  
    OS_ENTER_CRITICAL();  

    if (pevent->OSEventGrp != 0x00) {                         (4)
        tasks_waiting = TRUE;
    } else {
        tasks_waiting = FALSE;
    }
    switch (opt) {                                            
        case OS_DEL_NO_PEND:
            if (tasks_waiting == FALSE) {    (5)
                pevent->OSEventType = OS_EVENT_TYPE_UNUSED;
                pevent->OSEventPtr  = OSEventFreeList;
                OSEventFreeList     = pevent;       (6)
                OS_EXIT_CRITICAL();          (7)
                *err = OS_NO_ERR;
                return ((OS_EVENT *)0);        (8)
            } else {                      
                OS_EXIT_CRITICAL();
                *err = OS_ERR_TASK_WAITING;
                return (pevent);             
            }
        case OS_DEL_ALWAYS:
            while (pevent->OSEventGrp != 0x00) {                  (9)
                OS_EventTaskRdy(pevent, (void *)0, OS_STAT_MBOX); (10)
            }
            pevent->OSEventType = OS_EVENT_TYPE_UNUSED;          (11)
            pevent->OSEventPtr  = OSEventFreeList;                (12)
            OSEventFreeList     = pevent;
            OS_EXIT_CRITICAL();
            if (tasks_waiting == TRUE) {                          (13)
                OS_Sched();                                      
            }
            *err = OS_NO_ERR;
            return ((OS_EVENT *)0);                               (14)
        default:                                                  
            OS_EXIT_CRITICAL();                                 
            *err = OS_ERR_INVALID_OPT;
            return (pevent);                                  
    }
}
(1) OSBoxDel() starts by making sure that this function is not called from an ISR because that’s not allowed.

(2)

(3) We then validate pevent to ensure that it’s not a NULL pointer and that it points to an ECB that was created as a mailbox.

(4) OSBoxDel() then determines whether there are any tasks waiting on the mailbox. The flag tasks_waiting is set accordingly.

Based on the option (i.e., opt) specified in the call, OSBoxDel() will either delete the mailbox only if no tasks are pending on the mailbox (opt == OS_DEL_NO_PEND) or, delete the mailbox even if tasks are waiting (opt == OS_DEL_ALWAYS).

(5)

(6)

(7) When opt is set to OS_DEL_NO_PEND and there is no task waiting on the mailbox, OSBoxDel() marks the ECB as unused and the ECB is returned to the free list of ECBs. This will allow another mailbox (or any other ECB based object) to be created.

(8) You will note that OSBoxDel() returns a NULL pointer since, at this point, the mailbox should no longer be accessed through the original pointer. You ought to call OSBoxDel() as follows:

```c
MbxPtr = OSBoxDel(MbxPtr, opt, &err);
```

This allows the pointer to the mailbox to be altered by the call. OSBoxDel() returns an error code if there were task waiting on the mailbox (i.e. OS_ERR_TASK_WAITING) because by specifying OS_DEL_NO_PEND you indicated that you didn’t want to delete the mailbox if there are tasks waiting on the mailbox.

(9)

(10) When opt is set to OS_DEL_ALWAYS then all tasks waiting on the mailbox will be
readied. Each task will think it received a NULL message. Each task should examine the returned pointer to make sure it’s non-NULL. Also, you should note that interrupts are disabled while each task is being readied. This, of course, increases interrupt latency of your system.

(11)

(12) Once all pending tasks are readied, OSMboxDel() marks the ECB as unused and the ECB is returned to the free list of ECBs.

(13) The scheduler is called only if there were tasks waiting on the mailbox.

(14) Again, you will note that OSMboxDel() returns a NULL pointer since, at this point, the mailbox should no longer be accessed through the original pointer.

**Waiting for a Message at a Mailbox, OSMboxPend()**

The code to wait for a message to arrive at a mailbox is shown in Listing 10.3.
void *OSMboxPend (OS_EVENT *pevent, INT16U timeout, INT8U *err)
{
#if OS_CRITICAL_METHOD == 3
  OS_CPU_SR cpu_sr;
#endif
  void      *msg;

  if (OSIntNesting > 0) {                            (1)
    *err = OS_ERR_PEND_ISR;
    return ((void *)0);
  }
#if OS_ARG_CHK_EN > 0
  if (pevent == (OS_EVENT *)0) {                     (2)
    *err = OS_ERR_PEVENT_NULL;
    return ((void *)0);
  }
  if (pevent->OSEventType != OS_EVENT_TYPE_MBOX) {   (3)
    *err = OS_ERR_EVENT_TYPE;
    return ((void *)0);
  }
#endif
  OS_ENTER_CRITICAL();
  msg = pevent->OSEventPtr;                          (4)
  if (msg != (void *)0) {
    pevent->OSEventPtr = (void *)0;                (5)
    OS_EXIT_CRITICAL();
    *err = OS_NO_ERR;
    return (msg);                                  (6)
  }
  OSTCBCur->OSTCBStat |= OS_STAT_MBOX;               (7)
  OSTCBCur->OSTCBDly = timeout;                    (8)
  OS_EventTaskWait(pevent);                          (9)
  OS_EXIT_CRITICAL();
  OS_Sched();                                       (10)
  OS_ENTER_CRITICAL();
  msg = OSTCBCur->OSTCBMsg;                          (11)
  if (msg != (void *)0) {
    OSTCBCur->OSTCBMsg = (void *)0;
    OSTCBCur->OSTCBStat = OS_STAT_RDY;
    OSTCBCur->OSTCBEventPtr = (OS_EVENT *)0;
    OS_EXIT_CRITICAL();
    *err = OS_NO_ERR;
    return (msg);                                  (12)
  }
  OS_EventTO(pevent);                               (13)
  OS_EXIT_CRITICAL();
  *err = OS_TIMEOUT;
  return ((void *)0);                               (14)
}

Listing - Listing 10.3 Waiting for a message at a Mailbox (blocking),

(1) OSMboxPend() checks to see if the function was called by an ISR. It doesn’t make
sense to call OSMboxPend() from an ISR because an ISR cannot be made to wait.
Instead, you should call OSMboxAccept() (see section 10.05).

(2)
If OS_ARG_CHK_EN (see OS_CFG.H) is set to 1, OSMboxPend() checks that pevent is not a NULL pointer and the ECB being pointed to by pevent has been created by OSMboxCreate().

(4)

(5)

(6) If a message has been deposited in the mailbox (non NULL pointer), the message is extracted from the mailbox and replaced with a NULL pointer and the function returns to its caller with the message that was in the mailbox. An error code is also set indicating success. If your code calls OSMboxPend(), this is the outcome you are looking for because it indicates that another task or an ISR already deposited a message. This happens to be the fastest path through OSMboxPend().

If the mailbox was empty, the calling task needs to be put to sleep until another task (or an ISR) sends a message through the mailbox (see section 10.04). OSMboxPend() allows you to specify a timeout value (in integral number of ticks) as one of its arguments (i.e., timeout). This feature is useful to avoid waiting indefinitely for a message to arrive at the mailbox. If the timeout value is nonzero, OSMboxPend() suspends the task until the mailbox receives a message or the specified timeout period expires. Note that a timeout value of 0 indicates that the task is willing to wait forever for a message to arrive.

(7) To put the calling task to sleep, OSMboxPend() sets the status flag in the task’s TCB (Task Control Block) to indicate that the task is suspended waiting at a mailbox.

(8) The timeout is also stored in the TCB so that it can be decremented by OSTimeTick(). You should recall (see section 3.11, Clock Tick) that OSTimeTick() decrements each of the created task’s OSTCB.Dly field if it’s nonzero.

(9) The actual work of putting the task to sleep is done by OS_EventTaskWait() [see section 6.06, Making a Task Wait for an Event, OS_EventTaskWait()].

(10) Because the calling task is no longer ready to run, the scheduler is called to run the next highest priority task that is ready to run. As far as your task is concerned, it made a
call to OSMboxPend() and it doesn’t know that it will be suspended until a message arrives. When the mailbox receives a message (or the timeout period expires) OSMboxPend() will resume execution immediately after the call to OS_Sched().

(11) When OS_Sched() returns, OSMboxPend() checks to see if a message was placed in the task’s TCB by OSMboxPost().

(12) If so, the call is successful and the message is returned to the caller.

(13) If a message is not received then OS_Sched() must have returned because of a timeout. The calling task is then removed from the mailbox wait list by calling OS_EventTO().

(14) Note that the returned pointer is set to NULL because there is no message to return. The calling task should either examine the contents of the return pointer or the return code to determine whether a valid message was received.

**Sending a message to a mailbox, OSMboxPost()**

The code to deposit a message in a mailbox is shown in Listing 10.4.
Listing - Listing 10.4 Posting a message to a mailbox,

(1) If `OS_ARG_CHK_EN` is set to 1 in `OS_CFG.H`, `OSMboxPost()` checks to see that `pevent` is not a NULL pointer, that the message being posted is not a NULL pointer and finally, makes sure that the ECB is a mailbox.

(2) `OSMboxPost()` then checks to see if any task is waiting for a message to arrive at the mailbox. There are tasks waiting when the .OSEventGrp field in the ECB contains a nonzero value.

(3) The highest priority task waiting for the message is removed from the wait list by `OS_EventTaskRdy()` [see section 6.05, Making a Task Ready, `OS_EventTaskRdy()`], and this task is made ready to run.

(4) `OS_Sched()` is then called to see if the task made ready is now the highest priority task ready to run. If it is, a context switch results [only if `OSMboxPost()` is called from a task] and the readied task is executed. If the readied task is not the highest priority task,
OS_Sched() returns and the task that called OSMboxPost() continues execution.

(5) At this point, there are no tasks waiting for a message at the specified mailbox. OSMboxPost() then checks to see that there isn’t already a message in the mailbox. Because the mailbox can only hold one message, an error code is returned if we get this outcome.

(6) If there are no tasks waiting for a message to arrive at the mailbox, then the pointer to the message is saved in the mailbox. Storing the pointer in the mailbox allows the next task to call OSMboxPend() to get the message immediately.

Note that a context switch does not occur if OSMboxPost() is called by an ISR because context switching from an ISR only occurs when OSIntExit() is called at the completion of the ISR and from the last nested ISR (see section 3.09, Interrupts under µC/OS-II).

**Sending a message to a mailbox, OSMboxPostOpt()**

You can also post a message to a mailbox using an alternate and more powerful function called OSMboxPostOpt(). The reason there are two post calls is for backwards compatibility with previous versions of µC/OS-II. OSMboxPostOpt() is the newer function and can replace OSMboxPost(). In addition, OSMboxPostOpt() allows posting a message to all tasks (i.e. broadcast) waiting on the mailbox. The code to deposit a message in a mailbox is shown in Listing 10.5.
INT8U OSMboxPostOpt (OS_EVENT *pevent, void *msg, INT8U opt)
{
    #if OS_CRITICAL_METHOD == 3
        OS_CPU_SR cpu_sr;
    #endif

    #if OS_ARG_CHK_EN > 0
        if (pevent == (OS_EVENT *)0) {                              (1)
            return (OS_ERR_PEVENT_NULL);
        }
        if (msg == (void *)0) {
            return (OS_ERR_POST_NULL_PTR);
        }
        if (pevent->OSEventType != OS_EVENT_TYPE_MBOX) {
            return (OS_ERR_EVENT_TYPE);
        }
    #endif
    OS_ENTER_CRITICAL();
    if (pevent->OSEventGrp != 0x00) {                           (2)
        if ((opt & OS_POST_OPT_BROADCAST) != 0x00) {            (3)
            while (pevent->OSEventGrp != 0x00) {                (4)
                OS_EventTaskRdy(pevent, msg, OS_STAT_MBOX);     (5)
            }
        } else {                                               (6)
            OS_EventTaskRdy(pevent, msg, OS_STAT_MBOX);
        }
    } else {                                                   (7)
        OS_EXIT_CRITICAL();
    }
    OS_Sched();                                             (8)
    return (OS_NO_ERR);
}

Listing - Listing 10.5 Posting a message to a Mailbox,

(1) If OS_ARG_CHK_EN is set to 1 in OS_CFG.H, OSMboxPostOpt() checks to see that
    pevent is not a NULL pointer, that the message being posted is not a NULL pointer and
    finally, checks to make sure that the ECB is a mailbox.

(2) OSMboxPost() then checks to see if any task is waiting for a message to arrive at the
    mailbox. There are tasks waiting when the .OSEventGrp field in the ECB contains a
    nonzero value.

(3)

(4)
(5) If you set the `OS_POST_OPT_BROADCAST` bit in the opt argument then all tasks waiting for a message will receive the message. All tasks waiting for the message are removed from the wait list by `OS_EventTaskRdy()` [see section 6.05, Making a Task Ready, `OS_EventTaskRdy()`]. You should notice that interrupt disable time is proportional to the number of tasks waiting for a message from the mailbox.

(6) If a broadcast was not requested then, only the highest priority task waiting for a message will be made ready to run. The highest priority task waiting for the message is removed from the wait list by `OS_EventTaskRdy()`.

(7) `OS_Sched()` is then called to see if the task made ready is now the highest priority task ready to run. If it is, a context switch results [only if `OSMboxPostOpt()` is called from a task] and the readied task is executed. If the readied task is not the highest priority task, `OS_Sched()` returns and the task that called `OSMboxPostOpt()` continues execution.

(8) If nobody is waiting for a message, the message to post needs to be placed in the mailbox. In this case, `OSMboxPostOpt()` makes sure that there isn’t already a message in the mailbox. Remember that a mailbox can only contain one message. An error code would be returned if an attempt was made to add a message to an already full mailbox.

(9) `OSMboxPostOpt()` then deposits the message in the mailbox.

Note that a context switch does not occur if `OSMboxPostOpt()` is called by an ISR because context switching from an ISR only occurs when `OSIntExit()` is called at the completion of the ISR and from the last nested ISR (see section 3.10, Interrupts under µC/OS-II).

### Getting a message without waiting (non-blocking), `OSMboxAccept()`

You can obtain a message from a mailbox without putting a task to sleep if the mailbox is empty. This is accomplished by calling `OSMboxAccept()`, shown in Listing 10.6.
void *OSMboxAccept (OS_EVENT *pevent)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR cpu_sr;
#endif
    void  *msg;

#if OS_ARG_CHK_EN > 0
    if (pevent == (OS_EVENT *)0) {                       (1)
        return ((void *)0);
    }
    if (pevent->OSEventType != OS_EVENT_TYPE_MBOX) {     (2)
        return ((void *)0);
    }
#endif

    OS_ENTER_CRITICAL();
    msg = pevent->OSEventPtr;                          (3)
    pevent->OSEventPtr = (void *)0;                    (4)
    OS_EXIT_CRITICAL();
    return (msg);                                      (5)
}

Listing - Listing 10.6 Getting a message without waiting.

(1)

(2) If OS_ARG_CHK_EN is set to 1 in OS_CFG.H, OSMboxAccept() starts by checking that pevent is not a NULL pointer and that the ECB being pointed to by pevent has been created by OSMboxCreate().

(3) OSMboxAccept() then gets the current contents of the mailbox in order to determine whether a message is available (i.e., a non-NULL pointer).

(4) If a message is available, the mailbox is emptied. You should note that this operation is done even if the message already contains a NULL pointer. This is done for performance considerations.

(5) Finally, the original contents of the mailbox is returned to the caller.

The code that called OSMboxAccept() must examine the returned value. If OSMboxAccept() returns a NULL pointer, then a message was not available. A non-NULL pointer indicates that a message was deposited in the mailbox. An ISR should use OSMboxAccept() instead of OSMboxPend().

You can use OSMboxAccept() to flush (i.e., empty) the contents of a mailbox.
Obtaining the status of a mailbox, OSMboxQuery()

OSMboxQuery() allows your application to take a snapshot of an ECB used for a message mailbox. The code for this function is shown in Listing 10.7. OSMboxQuery() is passed two arguments: pevent contains a pointer to the message mailbox, which is returned by OSMboxCreate() when the mailbox is created, and pdata is a pointer to a data structure (OS_MBOX_DATA, see uCOS_II.H) that holds information about the message mailbox. Your application needs to allocate a variable of type OS_MBOX_DATA that will be used to receive the information about the desired mailbox. I decided to use a new data structure because the caller should only be concerned with mailbox-specific data, as opposed to the more generic OS_EVENT data structure, which contains two additional fields (.OSEventCnt and .OSEventType). OS_MBOX_DATA contains the current contents of the message (.OSMsg) and the list of tasks waiting for a message to arrive (.OSEventTbl[] and .OSEventGrp).
INT8U OSMboxQuery (OS_EVENT *pevent, OS_MBOX_DATA *pdata) {
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR cpu_sr;
#endif
    INT8U *psrc;
    INT8U *pdest;
#if OS_ARG_CHK_EN > 0
    if (pevent == (OS_EVENT *)0) {                       (1)
        return (OS_ERR_PEVENT_NULL);
    }
    if (pevent->OSEventType != OS_EVENT_TYPE_MBOX) {     (2)
        return (OS_ERR_EVENT_TYPE);
    }
#endif
    OS_ENTER_CRITICAL();
    pdata->OSEventGrp = pevent->OSEventGrp;              (3)
    psrc              = &pevent->OSEventTbl[0];
    pdest             = &pdata->OSEventTbl[0];
#if OS_EVENT_TBL_SIZE > 0
    *pdest++          = *psrc++;
#endif
#if OS_EVENT_TBL_SIZE > 1
    *pdest++          = *psrc++;
#endif
#if OS_EVENT_TBL_SIZE > 2
    *pdest++          = *psrc++;
#endif
#if OS_EVENT_TBL_SIZE > 3
    *pdest++          = *psrc++;
#endif
#if OS_EVENT_TBL_SIZE > 4
    *pdest++          = *psrc++;
#endif
#if OS_EVENT_TBL_SIZE > 5
    *pdest++          = *psrc++;
#endif
#if OS_EVENT_TBL_SIZE > 6
    *pdest++          = *psrc++;
#endif
#if OS_EVENT_TBL_SIZE > 7
    *pdest = *psrc;
#endif
    pdata->OSMsg = pevent->OSEventPtr;                  (4)
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);
}

Listing - Listing 10.7 Obtaining the status of a mailbox.

(1)
(2) As always, if OS_ARG_CHK is set to 1, OSMboxQuery() checks that pevent is not a NULL pointer and that it points to an ECB containing a mailbox.

(3) OSMboxQuery() then copies the wait list. You should note that I decided to do the copy as inline code instead of using a loop for performance reasons.

(4) Finally, the current message, from the OS_EVENT structure is copied to the OS_MBOX_DATA structure.

### Using a Mailbox as a Binary Semaphore

A message mailbox can be used as a binary semaphore by initializing the mailbox with a non-NULL pointer [(void *)1 works well]. A task requesting the “semaphore” calls OSMboxPend() and releases the “semaphore” by calling OSMboxPost(). Listing 10.8 shows how this works. You can use this technique to conserve code space if your application only needs binary semaphores and mailboxes. In this case, set OS_MBOX_EN to 1 and OS_SEM_EN to 0 so that you use only mailboxes instead of both mailboxes and semaphores.

```c
OS_EVENT *MboxSem;

void Task1 (void *pdata)
{
    INT8U err;
    for (;;) {
        OSMboxPend(MboxSem, 0, &err); /* Obtain access to resource(s) */
        /* Task has semaphore, access resource(s) */
        OSMboxPost(MboxSem, (void *)1); /* Release access to resource(s) */
    }
}
```

Listing - Listing 10.8 Using a mailbox as a binary semaphore.
Using a Mailbox instead of OSTimeDly()

The timeout feature of a mailbox can be used to simulate a call to OSTimeDly(). As shown in Listing 10.9, Task1() resumes execution after the time period expires if no message is received within the specified TIMEOUT. This is basically identical to OSTimeDly(TIMEOUT). However, the task can be resumed by Task2() when Task(2) post a “dummy” message to the mailbox before the timeout expires. This is the same as calling OSTimeDlyResume() had Task1() called OSTimeDly(). Note that the returned message is ignored because you are not actually looking to get a message from another task or an ISR.

```c
OS_EVENT *MboxTimeDly;

void Task1 (void *pdata)
{
    INT8U err;

    for (;;) {
        OSMboxPend(MboxTimeDly, TIMEOUT, &err); /* Delay task */
    
            /* Code executed after time delay or dummy message is received */
    
    }
}

void Task2 (void *pdata)
{
    INT8U err;

    for (;;) {
        OSMboxPost(MboxTimeDly, (void *)1); /* Cancel delay for Task1 */
    
    }
}
```

Listing - Listing 10.9 Using a mailbox as a time delay.