Message Queue Management

A message queue (or simply a queue) is a µC/OS-II object that allows a task or an ISR to send pointer-sized variables to another task. Each pointer typically is initialized to point to some application-specific data structure containing a message. µC/OS-II provides nine services to access message queues: OSQCreate(), OSQDel(), OSQPend(), OSQPost(), OSQPostFront(), OSQPostOpt(), OSQAccept(), OSQFlush(), and OSQuery().

Message Queue Configuration

To enable µC/OS-II message queue services, you must set configuration constants in OS_CFG.H. Specifically, table 11.1 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_Q_EN is set to 0 or OS_MAX_QS is set to 0. To enable a specific feature (i.e. service), simply set the corresponding configuration constant to 1. You will notice that OSQCreate() and OSQPend() 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: OSQPost(), OSQPostFront() and OSQPostOpt().

<table>
<thead>
<tr>
<th>µC/OS-II Queue Service</th>
<th>Enabled when set to 1 in OS_CFG.H</th>
</tr>
</thead>
<tbody>
<tr>
<td>OSQAccept()</td>
<td>OS_Q_ACCEPT_EN</td>
</tr>
<tr>
<td>OSQCreate()</td>
<td></td>
</tr>
<tr>
<td>OSQDel()</td>
<td>OS_Q_DEL_EN</td>
</tr>
<tr>
<td>OSQFlush()</td>
<td>OS_Q_FLUSH_EN</td>
</tr>
<tr>
<td>OSQPend()</td>
<td></td>
</tr>
<tr>
<td>OSQPost()</td>
<td>OS_Q_POST_EN</td>
</tr>
<tr>
<td>OSQPostFront()</td>
<td>OS_Q_POST_FRONT_EN</td>
</tr>
<tr>
<td>OSQPostOpt()</td>
<td>OS_Q_POST_OPT_EN</td>
</tr>
<tr>
<td>OSQuery()</td>
<td>OS_Q_QUERY_EN</td>
</tr>
</tbody>
</table>

Table - Table 11.1 Message queue configuration constants in OS_CFG.H.
Figure 11.1 shows a flow diagram to illustrate the relationship between tasks, ISRs, and a message queue. Note that the symbology used to represent a queue looks like a mailbox with multiple entries. In fact, you can think of a queue as an array of mailboxes, except that there is only one wait list associated with the queue. The hourglass represents a timeout that can be specified with the `OSQPend()` call. Again, what the pointers point to is application specific. $N$ represents the number of entries the queue holds. The queue is full when your application calls `OSQPost()` [or `OSQPostFront()` or `OSQPostOpt()`] $N$ times before your application has called `OSQPend()` or `OSQAccept()`.

As you can see from Figure 11.1, a task or an ISR can call `OSQPost()`, `OSQPostFront()`, `OSQPostOpt()`, `OSQFlush()`, or `OSQAccept()`. However, only tasks are allowed to call `OSQDel()`, `OSQPend()`, and `OSQQuery()`.

Figure 11.2 shows the different data structures needed to implement a message queue.
An ECB is required because you need a wait list, and using an ECB allows queue services to use some of the same code used by semaphores, mutexes and mailboxes.

When a message queue is created, a queue control block (i.e., an OS_Q, see OS_Q.C) is allocated and linked to the ECB using the OSEventPtr field in OS_EVENT.

Before you create a queue, however, you need to allocate an array of pointers that contains the desired number of queue entries. In other words, the number of elements in the array corresponds to the number of entries in the queue. The starting address of the array is passed to OSQCreate() as an argument as well as the size (in number of elements) of the array. In fact, you don’t actually need to use an array as long as the memory occupies contiguous locations.

The configuration constant OS_MAX_QS in OS_CFG.H specifies how many queues you are allowed to have in your application and must be greater than 0. When µC/OS-II is initialized, a list of free queue control blocks is created as shown in Figure11.3.
A queue control block is a data structure used to maintain information about the queue. It contains the fields described below. Note that the fields are preceded with a dot to show that they are members of a structure as opposed to simple variables.

.OSQPtr links queue control blocks in the list of free queue control blocks. Once the queue is created, this field is not used.

.OSQStart contains a pointer to the start of the message queue storage area. Your application must declare this storage area before creating the queue.

.OSQEnd is a pointer to one location past the end of the queue. This pointer is used to make the queue a circular buffer.

.OSQIn is a pointer to the location in the queue where the next message will be inserted. .OSQIn is adjusted back to the beginning of the message storage area when .OSQIn equals .OSQEnd.

.OSQOut is a pointer to the next message to be extracted from the queue. .OSQOut is adjusted back to the beginning of the message storage area when .OSQOut equals .OSQEnd. .OSQOut is also used to insert a message [see OSQPostFront () and OSQPostOpt ()].

.OSQSize contains the size of the message storage area. The size of the queue is determined by your application when the queue is created. Note that µC/OS-II allows the queue to contain up to 65,535 entries.

.OSQEntries contains the current number of entries in the message queue. The queue is empty
when `.OSQEntries` is 0 and full when it equals `.OSQSize`. The message queue is empty when the queue is created.

A message queue is basically a circular buffer as shown in Figure 11.4.

![Figure 11.4 A message queue is a circular buffer of pointers.](image)

(1) Each entry contains a pointer. The pointer to the next message is deposited at the entry pointed to by `.OSQIn` unless the queue is full (i.e., `.OSQEntries == .OSQSize`). Depositing the pointer at `.OSQIn` implements a FIFO (First-In-First-Out) queue. This is what `OSQPost()` does.

(2) µC/OS-II implements a LIFO (Last-In-First-Out) queue by pointing to the entry preceeding `.OSQOut` and depositing the pointer at that location (see `OSQPostFront()` and `OSQPostOpt()`).

(3) The pointer is also considered full when `.OSQEntries == .OSQSize`. Message pointers are always extracted from the entry pointed to by `.OSQOut`.

(4) The pointers `.OSQStart` and `.OSQEnd` are simply markers used to establish the beginning and end of the array so that `.OSQIn` and `.OSQOut` can wrap around to
implement this circular motion.

Creating a message queue, OSQCreate()

A message queue (or simply a queue) needs to be created before it can be used. Creating a queue is accomplished by calling OSQCreate() and passing it two arguments: a pointer to an array that will hold the messages and the size of this array. The array must be declared as an array of pointers to void as follows:

```c
void *MyArrayOfMsg[SIZE];
```

You would pass the address of `MyArrayOfMsg[]` to OSQCreate() as well as the size of this array. The message queue is assumed to be initially empty – it doesn’t contain any messages.

The code to create a mailbox is shown in Listing 11.1.
OS_EVENT  *OSQCreate (void **start, INT16U size)
{
  #if OS_CRITICAL_METHOD == 3
    OS_CPU_SR   cpu_sr;                           (1)
  #endif
  OS_EVENT  *pevent;
  OS_Q      *pq;

  if (OSIntNesting > 0) {                      (2)
    return ((OS_EVENT *)0);
  }
  OS_ENTER_CRITICAL();
  pevent = OSEventFreeList;                    (3)
  if (OSEventFreeList != (OS_EVENT *)0) {
    OSEventFreeList = (OS_EVENT *)OSEventFreeList->OSEventPtr;
  }
  OS_EXIT_CRITICAL();
  if (pevent != (OS_EVENT *)0) {               (4)
    OS_ENTER_CRITICAL();
    pq = OSQFreeList;
    if (pq != (OS_Q *)0) {                     (5)
      OSQfreeList = OSQFreeList->OSQPtr;
      OS_EXIT_CRITICAL();
      pq->OSQStart = start;
      pq->OSQEnd = &start[size];
      pq->OSQIn = start;
      pq->OSQOut = start;
      pq->OSQSize = size;
      pq->OSQEntries = 0;
      pevent->OSEventType = OS_EVENT_TYPE_Q;
    } else {                                     (6)
      pevent->OSEventPtr = (void *)OSEventFreeList;
    }
    OS_EventWaitListInit(pevent);                  (7)
  } else {                                     (8)
    pevent->OSEventPtr = (OS_Q *)0;
  }
  return (pevent);                                       (9)
}

Listing - Listing 11.1 Creating a message queue.

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

(2) OSQCreate() 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) OSQCreate() then attempts to obtain an ECB from the free list of ECBs (see Figure 6.5) and adjusts the linked list accordingly.

(4) If there is an ECB available, OSQCreate() attempts to allocate a queue control block
(OS_Q) from the free list of queue control blocks (see Figure 11.3) and adjusts the linked list accordingly.

(5)

(6) If a queue control block was available from the free list, the fields of the queue control block are initialized followed by the ones of the ECB. You should note that the .OSEventType field is set to OS_EVENT_TYPE_Q so that subsequent message queue services can check the validity of the ECB.

(7) The wait list is cleared indicating that no task is currently waiting on the message queue.

(8) If an ECB was available but a queue control block was not then, the ECB is returned to the free list since we cannot satisfy the request to create a queue unless we also have a queue control block.

(9) OSQCreate() returns either a pointer to the ECB upon successfully creating a message queue or, a NULL pointer if not. This pointer must be used (if not NULL) in subsequent calls that operate on message queues. The pointer is basically used as the queue’s handle.

Deleting a message queue, OSQDel()

The code to delete a message queue is shown in listing 11.2 and this code will only be generated by the compiler if OS_Q_DEL_EN is set to 1 in OS_CFG.H. This is a function that you must use with caution because multiple tasks could attempt to access a deleted message queue. You should always use this function with great care. Generally speaking, before you would delete a message queue, you would first delete all the tasks that access the message queue.
OS_EVENT *OSQDel (OS_EVENT *pevent, INT8U opt, INT8U *err)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR cpu_sr;
#endif
    BOOLEAN tasks_waiting;
    OS_Q *pq;

    if (OSIntNesting > 0) {
        *err = OS_ERR_DEL_ISR;
        return ((OS_EVENT *)0);
    }
#if OS_ARG_CHK_EN > 0
    if (pevent == (OS_EVENT *)0) {
        *err = OS_ERR_PEVENT_NULL;
        return (pevent);
    }
    if (pevent->OSEventType != OS_EVENT_TYPE_Q) {
        *err = OS_ERR_EVENT_TYPE;
        return (pevent);
    }
#endif
    OS_ENTER_CRITICAL();
    if (pevent->OSEventGrp != 0x00) {
        tasks_waiting = TRUE;
    } else {
        tasks_waiting = FALSE;
    }
    switch (opt) {
    case OS_DEL_NO_PEND:
        if (tasks_waiting == FALSE) {
            pq = pevent->OSEventPtr;
            pq->OSQPtr = OSQFreeList;
            OSQFreeList = pq;
            pevent->OSEventType = OS_EVENT_TYPE_UNUSED;
            OSEventFreeList = pevent;
            OS_EXIT_CRITICAL();
            *err = OS_NO_ERR;
            return ((OS_EVENT *)0);
        } else {
            OS_EXIT_CRITICAL();
            *err = OS_ERR_TASK_WAITING;
            return (pevent);
        }
    case OS_DEL_ALWAYS:
        while (pevent->OSEventGrp != 0x00) {
            OS_EventTaskRdy(pevent, (void *)0, OS_STAT_Q);
        }
        pq = pevent->OSEventPtr;
        pq->OSQPtr = OSQFreeList;
        OSQFreeList = pq;
        pevent->OSEventType = OS_EVENT_TYPE_UNUSED;
        OSEventFreeList = pevent;
        OS_EXIT_CRITICAL();
        if (tasks_waiting == TRUE) {
            OS_Sched();
        }
        *err = OS_NO_ERR;
        return ((OS_EVENT *)0);
    default:
        OS_EXIT_CRITICAL();
        *err = OS_ERR_INVALID_OPT;
    }
}
Listing - Listing 11.2 Deleting a Message Queue.

(1) OSQDel() starts by making sure that this function is not called from an ISR because that’s not allowed.

(2)

(3) If OS_ARG_CHK_EN (see OS_CFG.H) is set to 1, OSQDel() validates pevent to ensure that it’s not a NULL pointer and that it points to an ECB that was created as a queue.

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

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

(5) When opt is set to OS_DEL_NO_PEND and there is no task waiting on the queue, OSQDel() starts by returning the queue control block to the free list.

(6)

(7) OSQDel() then marks the ECB as unused and the ECB is returned to the free list of ECBs. This will allow another message queue (or any other ECB based object) to be created.

(8) You will note that OSQDel() returns a NULL pointer since, at this point, the queue should no longer be accessed through the original pointer. Because of this, you should call OSQDel() as follows:

QPtr = OSQDel(QPtr, opt, &err);

OSQDel() returns an error code if there were task waiting on the queue (i.e., OS_ERR_TASK_WAITING) because by specifying OS_DEL_NO_PEND you indicated that you
didn’t want to delete the queue if there are tasks waiting on the queue.

(9)

(10) When opt is set to OS_DEL_ALWAYS then all tasks waiting on the queue will be readied. Each task will think it received a message when in fact no message has been sent. The task should examine the pointer returned to it 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) OSQDel() then returns the queue control block to the free list.

(12)

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

(14) The scheduler is called only if there were tasks waiting on the queue.

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

**Waiting for a message at a queue (blocking), OSQPend()**

The code to wait for a message to arrive at a queue is shown in Listing 11.3.
void *OSQPend (OS_EVENT *pevent, INT16U timeout, INT8U *err)
{
#if OS_CRITICAL_METHOD == 3
  OS_CPU_SR cpu_sr;
#endif
  void      *msg;
  OS_Q      *pq;

  if (OSIntNesting > 0) {
      *(err = OS_ERR_PEND_ISR);
      return ((void *)0);
  }
#if OS_ARG_CHK_EN > 0
  if (pevent == (OS_EVENT *)0) {
      *(err = OS_ERR_PEVENT_NULL);
      return ((void *)0);
  }
  if (pevent->OSEventType != OS_EVENT_TYPE_Q) {
      *(err = OS_ERR_EVENT_TYPE);
      return ((void *)0);
  }
#endif
  OS_ENTER_CRITICAL();
  pq = (OS_Q *)pevent->OSEventPtr;
  if (pq->OSQEntries > 0) {
      msg = *(pq->OSQOut++);
      pq->OSQEntries--;
      if (pq->OSQOut == pq->OSQEnd) {
          pq->OSQOut = pq->OSQStart;
      }
      OS_EXIT_CRITICAL();
      *(err = OS_NO_ERR);
      return (msg);
  }
  OSTCBCur->OSTCBStat |= OS_STAT_Q;
  OSTCBCur->OSTCBDly   = timeout;
  OS_EventTaskWait(pevent);
  OS_EXIT_CRITICAL();
  OS_Sched();
  OS_ENTER_CRITICAL();
  msg = OSTCBCur->OSTCBMsg;
  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);
  }
  OS_EventTO(pevent);
  OS_EXIT_CRITICAL();
  *(err = OS_TIMEOUT);
  return ((void *)0);
}

Listing - Listing 11.3 Waiting for a message to arrive at a queue.

(1) It doesn’t make sense to call OSQPend() from an ISR because an ISR cannot be made
to wait. Instead, you should call OSQAccept() (see section 11.06).
(2)

(3) If `OS_ARG_CHK_EN` (see OS_CFG.H) is set to 1, `OSQPend()` verifies that `pevent` is not a NULL pointer and that the ECB being pointed to by `pevent` has been created by `OSQCreate()`.

(4)

(5) A message is available when `.OSQEntries` is greater than 0. In this case, `OSQPend()` gets the message pointed to by the `.OSQOut` field of the queue control block, stores the pointer to the message in `msg`, and moves the `.OSQOut` pointer so that it points to the next entry in the queue.

(6) `OSQPend()` then decrements the number of entries left in the queue since the previous operation ‘consumed’ the entry (i.e. removed the oldest message).

(7)

(8) Because a message queue is a circular buffer, `OSQPend()` needs to check that `.OSQOut` has not moved past the last valid entry in the array. When this happens, however, `.OSQOut` is adjusted to point back to the beginning of the array.

(9) The message that was extracted from the queue is then returned to the caller of `OSQPend()`. This is the path you are looking for when calling `OSQPend()`. It also happens to be the fastest path.

If the message queue was empty, the calling task needs to be put to sleep until another task (or an ISR) sends a message through the queue (see section 11.04). `OSQPend()` allows you to specify a timeout value (specified 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 queue. If the timeout value is nonzero, `OSQPend()` suspends the task until the queue 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.

(10) To put the calling task to sleep, `OSQPend()` sets the status flag in the task’s TCB (Task Control Block) to indicate that the task is suspended waiting for a queue.
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 OSTCBDly field if it’s nonzero.

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()].

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 OSQPend() and it doesn’t know that it will be suspended until a message arrives. When the queue receives a message (or the timeout period expires) OSQPend() will resume execution immediately after the call to OS_Sched().

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

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

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

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 queue (FIFO), OSQPost()**

The code to deposit a message in a queue is shown in Listing 11.4.
Listing 11.4 Depositing a message in a queue (FIFO),

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

(2) OSQPost() then checks to see if any task is waiting for a message to arrive at the queue. There are tasks waiting when the .OSEventGrp field in the ECB contains a nonzero value.
(5) The highest priority task waiting for the message is removed from the wait list by OS_EventTaskRdy() [see section 10.02, Making a Task Ready, OS_EventTaskRdy()], and this task is made ready to run.

(6) 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 OSQPost() 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 OSQPost() continues execution.

(7) If no task is waiting for a message, the message to post needs to be placed in the queue. In this case, OSQPost() makes sure that there is still room in the queue. An error code would be returned if an attempt was made to add a message to an already full queue.

(8) 

(9) If there are no tasks waiting for a message to arrive at the queue and the queue is not already full then the message to post is inserted in the next free location (FIFO order) and the number of entries in the queue is incremented.

(10) Finally, OSQPost() adjust the circular buffer pointer to prepare for the next post.

Note that a context switch does not occur if OSQPost() 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).

**Sending a message to a queue (LIFO), OSQPostFront()**

OSQPostFront() is basically identical to OSQPost(), except that OSQPostFront() uses OSQOut instead of OSQIn as the pointer to the next entry to insert. The code is shown in Listing 11.5.
INT8U OSQPostFront (OS_EVENT *pevent, void *msg)
{
    #if OS_CRITICAL_METHOD == 3
        OS_CPU_SR cpu_sr;
    #endif
    OS_Q *pq;

    #if OS_ARG_CHK_EN > 0
        if (pevent == (OS_EVENT *)0) {
            return (OS_ERR_PEVENT_NULL);
        }
        if (msg == (void *)0) {
            return (OS_ERR_POST_NULL_PTR);
        }
        if (pevent->OSEventType != OS_EVENT_TYPE_Q) {
            return (OS_ERR_EVENT_TYPE);
        }
    #endif
    OS_ENTER_CRITICAL();
    if (pevent->OSEventGrp != 0x00) {
        OS_EventTaskRdy(pevent, msg, OS_STAT_Q);
        OS_EXIT_CRITICAL();
        OS_Sched();
        return (OS_NO_ERR);
    }
    pq = (OS_Q *)pevent->OSEventPtr;
    if (pq->OSQEntries == pq->OSQSize) {
        OS_EXIT_CRITICAL();
        return (OS_Q_FULL);
    }
    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);
}

Listing - Listing 11.5 Depositing a message in a queue (LIFO),

(1)

(2) You should note, however, that .OSQOut points to an already inserted entry, so .OSQOut must be made to point to the previous entry. If .OSQOut points at the beginning of the array, then a decrement really means positioning .OSQOut at the end of the array.

(3) However, .OSQEnd points to one entry past the array and thus .OSQOut needs to be adjusted to be within range. OSQPostFront() implements a LIFO queue because the next message extracted by OSQPend() is the last message inserted by OSQPostFront().
Sending a message to a queue (FIFO or LIFO), OSQPostOpt()

You can also post a message to a queue using an alternate an more flexible function called OSQPostOpt(). The reason there are three post calls is for backwards compatibility with previous versions of µC/OS-II. OSQPostOpt() is the newer function and can replace both OSQPost() and OSQPostFront() with a single call. In addition, OSQPostOpt() allows posting a message to all tasks (i.e. broadcast) waiting on the queue. The code to deposit a message in a queue is shown in Listing 11.6.
INT8U OSQPostOpt (OS_EVENT *pevent, void *msg, INT8U opt) 
{
    #if OS_CRITICAL_METHOD == 3
    OS_CPU_SR cpu_sr;
    #endif
    OS_Q *pq;

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

pq = (OS_Q *)pevent->OSEventPtr;
if (pq->OSQEntries >= pq->OSQSize) {              (9)
    return (OS_Q_FULL);
}
if ((opt & OS_POST_OPT_FRONT) != 0x00) {          (10)
    if (pq->OSQOut == pq->OSQEnd) {             (11)
        pq->OSQOut = pq->OSQStart;
    }
    pq->OSQOut--;                                 (12)
    *pq->OSQOut = msg;
} else {                                       (13)
    if (pq->OSQIn == pq->OSQEnd) {
        pq->OSQIn = pq->OSQStart;
    }
    *pq->OSQIn++ = msg;
}

pq->OSQEntries++;                                (14)
OS_EXIT_CRITICAL();
return (OS_NO_ERR);
}

Listing - Listing 11.6 Depositing a message in a queue (Broadcast, FIFO or LIFO),

(1)

(2)
(3) If OS_ARG_CHK_EN is set to 1 in OS_CFG.H, OSQPostOpt() checks to see that pevent is not a NULL pointer, checks that the message being posted is also not a NULL pointer and finally, checks to make sure that the ECB is a queue.

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

(5)

(6) 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 10.02, 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 queue.

(7) 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().

(8) 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 OSQPostOpt() 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 OSQPostOpt() continues execution.

(9) If nobody is waiting for a message, the message to post needs to be placed in the queue. In this case, OSQPostOpt() makes sure that there is still room in the queue. An error code would be returned if an attempt was made to add a message to an already full queue.

(10) OSQPostOpt() then checks the opt argument to see if the calling task desires to post the message in FIFO or LIFO (setting opt to OS_POST_OPT_FRONT) order.

(11) If LIFO order is selected, OSQPostOpt() emulates OSQPostFront().

(12) If FIFO order, OSQPostOpt() emulates OSQPost().
In either case, the number of entries in the queue is incremented.

Note that a context switch does not occur if OSQPostOpt() 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, OSQAccept()**

You can obtain a message from a queue without putting a task to sleep by calling OSQAccept() if the queue is empty. The code for this function is shown in Listing 11.7.

```c
void  *OSQAccept (OS_EVENT *pevent)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    void      *msg;
    OS_Q      *pq;

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

    OS_ENTER_CRITICAL();
    pq = (OS_Q *)pevent->OSEventPtr;
    if (pq->OSQEntries > 0) {                              (3)
        msg = *pq->OSQOut++;                               (4)
        pq->OSQEntries--;                                  (5)
        if (pq->OSQOut == pq->OSQEnd) {                    (6)
            pq->OSQOut = pq->OSQStart;
        } else {
            msg = (void *)0;                               (7)
        }
    OS_EXIT_CRITICAL();
    return (msg);
}
```

Listing - Listing 11.7 Getting a message without waiting (non-blocking),

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

(3) OSQAccept() then checks to see if there are any entries in the queue by looking at the .OSQEntries queue control block field.

(4)

(5) If a message is available, the oldest message (FIFO order) is retrieved from the queue and copied to the local pointer msg and the number of entries in the queue is decreased by one to reflect the extraction.

(6) OSQAccept() then adjust the circular queue pointer by moving the .OSQOut pointer to the next entry.

(7) If there were no entries in the queue, the local pointer is set to NULL.

The code that calls OSQAccept() needs to examine the returned value. If OSQAccept() returns a NULL pointer, then a message was not available. You don’t want your application to dereference a NULL pointer because, by convention, a NULL pointer is invalid. A non-NULL pointer indicates that a message pointer is available. An ISR should use OSQAccept() instead of OSQPend().

**Flushing a Queue, OSQFlush()**

OSQFlush() allows you to remove all the messages posted to a queue and basically start with a fresh queue. The code for this function is shown in Listing 11.8.
INT8U OSQFlush (OS_EVENT *pevent)  
{  
#if OS_CRITICAL_METHOD == 3  
    OS_CPU_SR cpu_sr;  
#endif  
    OS_Q *pq;  

    #if OS_ARG_CHK_EN > 0  
        if (pevent == (OS_EVENT *)0) {   (1)  
            return (OS_ERR_PEVENT_NULL);  
        }  
        if (pevent->OSEventType != OS_EVENT_TYPE_Q) {   (2)  
            return (OS_ERR_EVENT_TYPE);  
        }  
    #endif  
    OS_ENTER_CRITICAL();  
    pq             = (OS_Q *)pevent->OSEventPtr;      (3)  
    pq->OSQIn      = pq->OSQStart;  
    pq->OSQOut     = pq->OSQStart;  
    pq->OSQEntries = 0;  
    OS_EXIT_CRITICAL();  
    return (OS_NO_ERR);  
}  

Listing - Listing 11.8 Flushing the contents of a queue.

(1)

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

(3) The IN and OUT pointers are reset to the beginning of the array and the number of entries is cleared. I decided to not check to see if any tasks were pending on the queue because it would be irrelevant anyway and would take more processing time. In other words, if tasks are waiting on the queue, then .OSQEntries would already be set to 0. The only difference is that .OSQIn and .OSQOut may be pointing elsewhere in the array. There is also no need to fill the queue with NULL pointers.
Obtaining the Status of a Queue, OSQQuery()

OSQQuery() allows your application to take a snapshot of the contents of a message queue. The code for this function is shown in Listing 11.9. OSQQuery() is passed two arguments: pevent contains a pointer to the message queue, which is returned by OSQCreate() when the queue is created, and pdata is a pointer to a data structure (OS_Q_DATA, see uCos_II.h) that holds information about the message queue. Your application thus needs to allocate a variable of type OS_Q_DATA that will receive the information about the desired queue. OS_Q_DATA contains the following fields:

- .OSMsg contains the contents pointed to by .OSQOut if there are entries in the queue. If the queue is empty, .OSMsg will contain a NULL pointer.
- .OSNMsgs contains the number of messages in the queue (i.e., a copy of .OSQEntries).
- .OSQSize contains the size of the queue (in number of entries).
- .OSEventTbl[]
- .OSEventGrp contain a snapshot of the message queue wait list. The caller to OSQQuery() can thus determine how many tasks are waiting for the queue.
INT8U OSQuery (OS_EVENT *pevent, OS_Q_DATA *pdata)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR cpu_sr;
#endif
    OS_Q *pq;
    INT8U *psrc;
    INT8U *pdest;
#endif OS_ARG_CHK_EN > 0
    if (pevent == (OS_EVENT *)0) {                         (1)
        return (OS_ERR_PEVENT_NULL);
    }
    if (pevent->OSEventType != OS_EVENT_TYPE_Q) {          (2)
        return (OS_ERR_EVENT_TYPE);
    }
    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
    pq = (OS_Q *)pevent->OSEventPtr;
    if (pq->OSQEntries > 0) {                              (4)
        pdata->OSMsg = *pq->OSQOut;
    } else {
        pdata->OSMsg = (void *)0;
    }
    pdata->OSMMsgs = pq->OSQEntries;
    pdata->OSQSize = pq->OSQSize;
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);
}
Listing - Listing 11.9 Obtaining the status of a queue.

(1)

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

(3) OSQuery() 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) If the queue is not empty, the oldest message is extracted (but not removed) from the queue and copied to .OSMsg. In other words, OSQuery() will not move the .OSQOut pointer. If there are no messages in the queue, the .OSmsg will contain a NULL pointer.

(5)

(6) Finally, the current number of entries and the queue size are placed in the .OSNMsgs and .OSQSize fields of the OS_Q_DATA structure, respectively.

Using a Message Queue When Reading Analog Inputs

It is often useful in control applications to read analog inputs at a regular interval. To accomplish this, create a task, call OSTimeDly() [see section 5.00, Delaying a Task, OSTimeDly()], and specify the desired sampling period.

As shown in Figure 6.11, you could use a message queue instead and have your task pend on the queue with a timeout. The timeout corresponds to the desired sampling period. If no other task sends a message to the queue, the task is resumed after the specified timeout, which basically emulates the OSTimeDly() function.

You are probably wondering why I decided to use a queue when OSTimeDly() does the trick just fine. By adding a queue, you can have other tasks abort the wait by sending a message, thus forcing an immediate conversion. If you add some intelligence to your messages, you can tell the ADC task to convert a specific channel, tell the task to increase the sampling rate, and
more. In other words, you can say to the task: “Can you convert analog input 3 for me now?” After servicing the message, the task would initiate the pend on the queue, which would restart the scanning process.

![Diagram of analog inputs](image)

**Figure - Figure 11.5 Reading analog inputs**

### Using a Queue as a Counting Semaphore

A message queue can be used as a counting semaphore by initializing and loading a queue with as many non-NULL pointers [(void *)1 works well] as there are resources available. A task requesting the “semaphore” calls OSQPend() and releases the “semaphore” by calling OSQPost(). Listing 11.10 shows how this works. You can use this technique to conserve code space if your application only needs counting semaphores and message queues (you would then have no need for the semaphore services). In this case, set OS_SEM_EN to 0 and only use queues instead of both queues and semaphores. Note that this technique consumes a pointer-sized variable for each resource that the semaphore is guarding and requires a queue control block. In other words, you are sacrificing RAM space in order to save code space. Also, message queue services are slower than semaphore services. This technique would be very inefficient if your counting semaphore (in this case a queue) is guarding a large amount of resources (you would require a large array of pointers).
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);     /* Obtain access to resource(s) */
        /* Task has semaphore, access resource(s) */
        
        OSMQPost(QSem, (void*)1);    /* Release access to resource(s) */
    }
}