Unit II – Embedded/RTOS Concepts

- Process Management
- Memory Management
- Device Management
- Basic Guidelines to choose an OS for Embedded Applications
- Other Building Blocks
- Component Configuration
- Basic I/O Concepts
- I/O Subsystem
- Kernel Objects:
  - Pipes
  - Semaphores
  - Mailbox
  - Message Queue
  - Signals
  - Event Registers

Introduction

Multiple concurrent threads of execution within an application must be able to synchronize their execution and coordinate mutually exclusive access to shared resources. To address these requirements, RTOS kernels provide a semaphore object and associated semaphore management services.

Defining Semaphores

A *semaphore* (sometimes called a *semaphore token*) is a kernel object that one or more threads of execution can acquire or release for the purposes of synchronization or mutual exclusion.

When a semaphore is first created, the kernel assigns to it an associated semaphore control block (SCB), a unique ID, a value (binary or a count), and a task-waiting list, as shown in Figure 1.

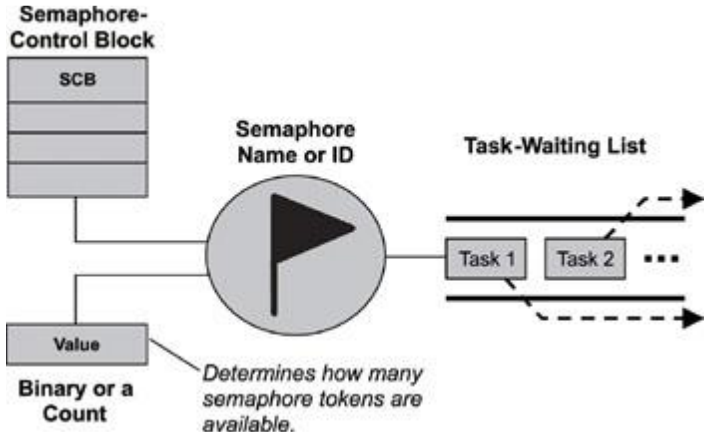


Figure 1: A semaphore, its associated parameters, and supporting data structures

A semaphore is like a key that allows a task to carry out some operation or to access a resource. If the task can acquire the semaphore, it can carry out the intended operation or access the resource. A single semaphore can be acquired a finite number of times. Likewise, when a semaphore s limit is reached, it can no longer be acquired until someone gives a key back or releases the semaphore.

The kernel tracks the number of times a semaphore has been acquired or released by maintaining a token count, which is initialized to a value when the semaphore is created. As a task acquires the semaphore, the token count is decremented; as a task releases the semaphore, the count is incremented. If the token count reaches 0, the semaphore has no tokens left. A requesting task, therefore, cannot acquire the semaphore, and the task blocks if it chooses to wait for the semaphore to become available.

The task-waiting list tracks all tasks blocked while waiting on an unavailable semaphore. These blocked tasks are kept in the task-waiting list in either first in/first out (FIFO) order or highest priority first order.

When an unavailable semaphore becomes available, the kernel allows the first task in the task-waiting list to acquire it. The kernel moves this unblocked task either to the running state, if it is the highest priority task, or to the ready state, until it becomes the highest priority task and is able to run.

A kernel can support many different types of semaphores, including *binary*, *counting*, and *mutual-exclusion (mutex)* semaphores.

Binary Semaphores

A *binary semaphore* can have a value of either 0 or 1. When a binary semaphore s value is 0, the semaphore is considered *unavailable* (or *empty);* when the value is 1, the binary semaphore is considered *available* (or *full*).

Note that when a binary semaphore is first created, it can be initialized to either available or unavailable (1 or 0, respectively). The state diagram of a binary semaphore is shown in Figure 2.

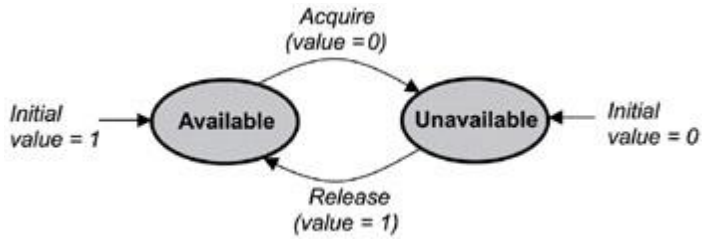


Figure 2: The state diagram of a binary semaphore.

Binary semaphores are treated as *global resources*, which mean they are shared among all tasks that need them. Making the semaphore a global resource allows any task to release it, even if the task did not initially acquire it.

Counting Semaphores

A *counting semaphore* uses a count to allow it to be acquired or released multiple times. When creating a counting semaphore, assign the semaphore a count that denotes the number of semaphore tokens it has initially. If the initial count is 0, the counting semaphore is created in the unavailable state. If the count is greater than 0, the semaphore is created in the available state, and the number of tokens it has equals its count, as shown in Figure 3.

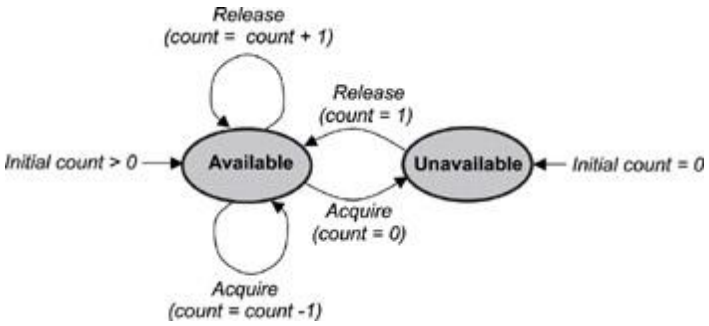


Figure 3: The state diagram of a counting semaphore.

One or more tasks can continue to acquire a token from the counting semaphore until no tokens are left. When all the tokens are gone, the count equals 0, and the counting semaphore moves from the available state to the unavailable state. To move from the unavailable state back to the available state, a semaphore token must be released by any task.

Note that, as with binary semaphores, counting semaphores are global resources that can be shared by all tasks that need them. This feature allows any task to release a counting semaphore token. Each release operation increments the count by one, even if the task making this call did not acquire a token in the first place.

Some implementations of counting semaphores might allow the count to be bounded. A *bounded count* is a count in which the initial count set for the counting semaphore, determined when the semaphore was first created, acts as the maximum count for the semaphore. An *unbounded count* allows the counting semaphore to count beyond the initial count to the maximum value that can be held by the count s data type (e.g., an unsigned integer or an unsigned long value).

Mutual Exclusion (Mutex) Semaphores

A mutual exclusion (mutex) semaphore is a special binary semaphore that supports ownership, recursive access, task deletion safety, and one or more protocols for avoiding problems inherent to mutual exclusion. Figure 4 illustrates the state diagram of a mutex.

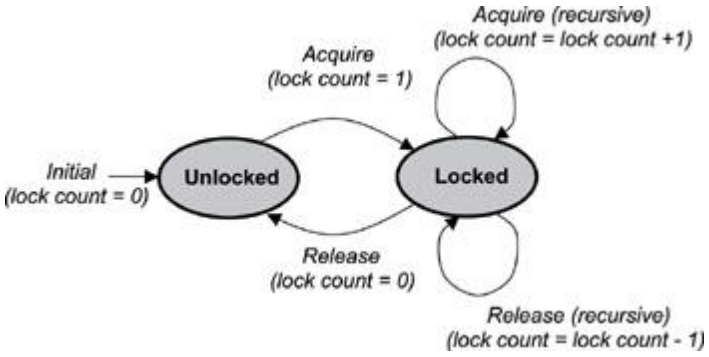


Figure 4: The state diagram of a mutual exclusion (mutex) semaphore.

As opposed to the available and unavailable states in binary and counting semaphores, the states of a mutex are *unlocked* or *locked* (0 or 1, respectively). A mutex is initially created in the unlocked state, in which it can be acquired by a task. After being acquired, the mutex moves to the locked state. Conversely, when the task releases the mutex, the mutex returns to the unlocked state. Note that some kernels might use the terms *lock* and *unlock* for a mutex instead of *acquire* and *release*.

Depending on the implementation, a mutex can support additional features not found in binary or counting semaphores. These key differentiating features include ownership, recursive locking, task deletion safety, and priority inversion avoidance protocols.

Mutex Ownership

Ownership of a mutex is gained when a task first locks the mutex by acquiring it. Conversely, a task loses ownership of the mutex when it unlocks it by releasing it. When a task owns the mutex, it is not possible for any other task to lock or unlock that mutex. Contrast this concept with the binary semaphore, which can be released by any task, even a task that did not originally acquire the semaphore.

Recursive Locking

Many mutex implementations also support *recursive locking* , which allows the task that owns the mutex to acquire it multiple times in the locked state. Depending on the implementation, recursion within a mutex can be automatically built into the mutex, or it might need to be enabled explicitly when the mutex is first created.

The mutex with recursive locking is called a *recursive mutex* . This type of mutex is most useful when a task requiring exclusive access to a shared resource calls one or more routines that also require access to the same resource. A recursive mutex allows nested attempts to lock the mutex to succeed, rather than cause *deadlock* , which is a condition in which two or more tasks are blocked and are waiting on mutually locked resources.

As shown in Figure 4, when a recursive mutex is first locked, the kernel registers the task that locked it as the owner of the mutex. On successive attempts, the kernel uses an internal lock count associated with the mutex to track the number of times that the task currently owning the mutex has recursively acquired it. To properly unlock the mutex, it must be released the same number of times.

In this example, a *lock count* tracks the two states of a mutex (0 for unlocked and 1 for locked), as well as the number of times it has been recursively locked (lock count > 1). In other implementations, a mutex might maintain two counts: a binary value to track its state, and a separate lock count to track the number of times it has been acquired in the lock state by the task that owns it. Additionally, the count for the mutex is always unbounded, which allows multiple recursive accesses.

Task Deletion Safety

Some mutex implementations also have built-in *task deletion safety*. Premature task deletion is avoided by using *task deletion locks* when a task locks and unlocks a mutex. Enabling this capability within a mutex ensures that while a task owns the mutex, the task cannot be deleted. Typically protection from premature deletion is enabled by setting the appropriate initialization options when creating the mutex.

Priority Inversion Avoidance

Priority inversion commonly happens in poorly designed real-time embedded applications. Priority inversion occurs when a higher priority task is blocked and is waiting for a resource being used by a lower priority task, which has itself been preempted by an unrelated medium-priority task. In this situation, the higher priority task s priority level
has effectively been inverted to the lower priority task s level.

Enabling certain protocols that are typically built into mutexes can help avoid priority inversion. Two common protocols used for avoiding priority inversion include:

- **priority inheritance protocol** ensures that the priority level of the lower priority task that has acquired the mutex is raised to that of the higher priority task that has requested the mutex when inversion happens. The priority of the raised task is lowered to its original value after the task releases the mutex that the higher priority task requires.

- **ceiling priority protocol** ensures that the priority level of the task that acquires the mutex is automatically set–to the highest priority of all possible tasks that might request that mutex when it is first acquired until it is released. When the mutex is released, the priority of the task is lowered to its original value.

Semaphore Operations

Typical operations that developers might want to perform with the semaphores in an application include:

- creating and deleting semaphores,
- acquiring and releasing semaphores,
- clearing a semaphore s task-waiting list, and
- getting semaphore information.

.

Creating and Deleting Semaphores

Table 1 identifies the operations used to create and delete semaphores.

Table 1: Semaphore creation and deletion operations.

Operation	Description
Create	Creates a semaphore
Delete	Deletes a semaphore

If a kernel supports different types of semaphores, different calls might be used for creating binary, counting, and mutex semaphores, as follows:

- **binary** specify the initial semaphore state and the task-waiting order.
- **counting** specify the initial semaphore count and the task-waiting order.
- **mutex** specify the task-waiting order and enable task deletion safety, recursion, and priority-inversion avoidance protocols, if supported.

Semaphores can be deleted from within any task by specifying their IDs and making semaphore-deletion calls. Deleting a semaphore is not the same as releasing it. When a semaphore is deleted, blocked tasks in its task-waiting list are unblocked and moved either to the ready state or to the running state (if the unblocked task has the highest priority). Any tasks, however, that try to acquire the deleted semaphore return with an error because the semaphore no longer exists.

Additionally, do not delete a semaphore while it is in use (e.g., acquired). This action might result in data corruption or other serious problems if the semaphore is protecting a shared resource or a critical section of code.

Acquiring and Releasing Semaphores

Table.2 identifies the operations used to acquire or release semaphores.

Table 2: Semaphore acquire and release operations.

Operation	Description
Acquire	Acquire a semaphore token
Release	Release a semaphore token

The operations for acquiring and releasing a semaphore might have different names, depending on the kernel: for example, *take* and *give* , *sm_p* and *sm_v* , *pend* and *post* , and *lock* and *unlock* . Regardless of the name, they all effectively acquire and release semaphores.

Tasks typically make a request to acquire a semaphore in one of the following ways:

- **Wait forever** task remains blocked until it is able to acquire a semaphore.
- **Wait with a timeout** task remains blocked until it is able to acquire a semaphore or until a set interval of time, called the *timeout interval* , passes. At this point, the task is removed from the semaphores task-waiting list and put in either the ready state or the running state.
- **Do not wait** task makes a request to acquire a semaphore token, but, if one is not available, the task does not block.

Note that ISRs can also release binary and counting semaphores. Note that most kernels do not support ISRs locking and unlocking mutexes, as it is not meaningful to do so from an ISR.

Any task can release a binary or counting semaphore; however, a mutex can only be released (unlocked) by the task that first acquired (locked) it. Note that incorrectly releasing a binary or

counting semaphore can result in losing mutually exclusive access to a shared resource or in an I/O device malfunction.

Clearing Semaphore Task-Waiting Lists

To clear all tasks waiting on a semaphore task-waiting list, some kernels support a *flush* operation, as shown in Table 3.

Table 3: Semaphore unblock operations.

Operation	Description
Flush	Unblocks all tasks waiting on a semaphore

The flush operation is useful for broadcast signaling to a group of tasks. For example, a developer might design multiple tasks to complete certain activities first and then block while trying to acquire a common semaphore that is made unavailable. After the last task finishes doing what it needs to, the task can execute a semaphore flush operation on the common semaphore. This operation frees all tasks waiting in the semaphores task waiting list.

Getting Semaphore Information

At some point in the application design, developers need to obtain semaphore information to perform monitoring or debugging. In these cases, use the operations shown in Table.4.

Table 4: Semaphore information operations.

Operation	Description
Show info	Show general information about semaphore
Show blocked tasks	Get a list of IDs of tasks that are blocked on a semaphore

Typical Semaphore Use

Semaphores are useful either for synchronizing execution of multiple tasks or for coordinating access to a shared resource. The following examples illustrate common synchronization design requirements effectively, as listed:

- wait-and-signal synchronization,
- multiple-task wait-and-signal synchronization,

- credit-tracking synchronization,
- single shared-resource-access synchronization,
- recursive shared-resource-access synchronization, and
- multiple shared-resource-access synchronization.

Wait-and-Signal Synchronization

Two tasks can communicate for the purpose of synchronization without exchanging data. For example, a binary
semaphore can be used between two tasks to coordinate the transfer of execution control, as shown in Figure 5.

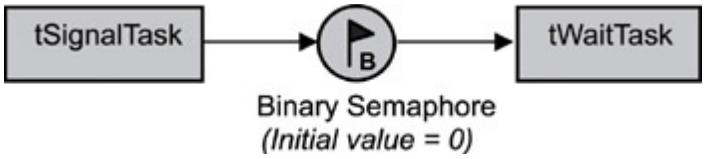


Figure 5: Wait-and-signal synchronization between two tasks.

In this situation, the binary semaphore is initially unavailable (value of 0). tWaitTask has higher priority and runs first. The task makes a request to acquire the semaphore but is blocked because the semaphore is unavailable. This step gives the lower priority tSignalTask a chance to run; at some point, tSignalTask releases the binary semaphore and unblocks tWaitTask. The pseudo code for this scenario is shown in Listing 1.

Listing 1: Pseudo code for wait-and-signal synchronization

```
tWaitTask ( )
{
       :
        Acquire binary semaphore token
       :
}

tSignalTask ( )
{
       :
        Release binary semaphore token
        :
}
```

Because tWaitTask's priority is higher than tSignalTask's priority, as soon as the semaphore is released, tWaitTask preempts tSignalTask and starts to execute.

Multiple-Task Wait-and-Signal Synchronization

When coordinating the synchronization of more than two tasks, use the flush operation on the task-waiting list of a binary semaphore, as shown in Figure 6.

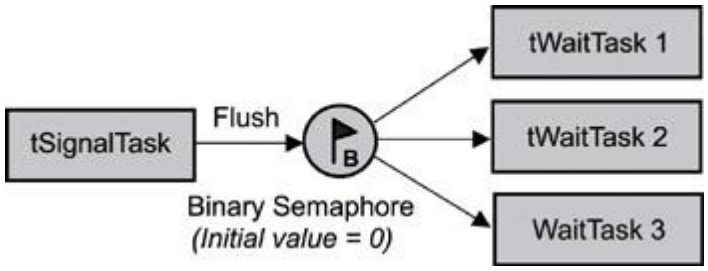


Figure.6: Wait-and-signal synchronization between multiple tasks.

As in the previous case, the binary semaphore is initially unavailable (value of 0). The higher priority tWaitTasks 1, 2, and 3 all do some processing; when they are done, they try to acquire the unavailable semaphore and, as a result, block. This action gives tSignalTask a chance to complete its processing and execute a flush command on the semaphore, effectively unblocking the three tWaitTasks, as shown in Listing 2. Note that similar code is used for
tWaitTask 1, 2, and 3.

Listing 2: Pseudo code for wait-and-signal synchronization.

```
tWaitTask ()
    {
           :
            Do some processing specific to task
            Acquire binary semaphore token
           :
     }
tSignalTask ()
     {
         :
        Do some processing
        Flush binary semaphore's task-waiting list
               :
         }
```

Because the tWaitTasks' priorities are higher than tSignalTask's priority, as soon as the semaphore is released, one of the higher priority tWaitTasks preempts tSignalTask and starts to execute.

Note that in the wait-and-signal synchronization shown in Figure.6 the value of the binary semaphore after the flush operation is implementation dependent. Therefore, the return value of the acquire operation must be properly checked to see if either a return-from-flush or an error condition has occurred.

Credit-Tracking Synchronization

Sometimes the rate at which the signaling task executes is higher than that of the signaled task. In this case, a mechanism is needed to count each signaling occurrence. The counting semaphore provides just this facility. With a counting semaphore, the signaling task can continue to execute and increment a count at its own pace, while the wait task, when unblocked, executes at its own pace, as shown in Figure 7.

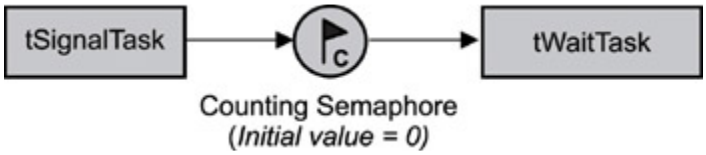


Figure 7: Credit-tracking synchronization between two tasks.

Again, the counting semaphore's count is initially 0, making it unavailable. The lower priority tWaitTask tries to acquire this semaphore but blocks until tSignalTask makes the semaphore available by performing a release on it.

Even then, tWaitTask will waits in the ready state until the higher priority tSignalTask eventually relinquishes the CPU by making a blocking call or delaying itself, as shown in Listing 3.

Listing 3: Pseudo code for credit-tracking synchronization.

```
tWaitTask ()
{
          :
           Acquire counting semaphore token
           :
}
tSignalTask ()
{
      :
           Release counting semaphore token
           :
 }
```

Because tSignalTask is set to a higher priority and executes at its own rate, it might increment the counting semaphore multiple times before tWaitTask starts processing the first request. Hence, the counting semaphore allows a credit buildup of the number of times that the tWaitTask can execute before the semaphore becomes unavailable.

Eventually, when tSignalTask's rate of releasing the semaphore tokens slows, tWaitTask can catch up and eventually deplete the count until the counting semaphore is empty. At this point, tWaitTask blocks again at the counting semaphore, waiting for tSignalTask to release the semaphore again.

Note that this credit-tracking mechanism is useful if tSignalTask releases semaphores in bursts, giving tWaitTask the chance to catch up every once in a while

Single Shared-Resource-Access Synchronization

One of the more common uses of semaphores is to provide for mutually exclusive access to a shared resource. A shared resource might be a memory location, a data structure, or an I/O device-essentially anything that might have to be shared between two or more concurrent threads of execution. A semaphore can be used to serialize access to a shared resource, as shown in Figure 8.

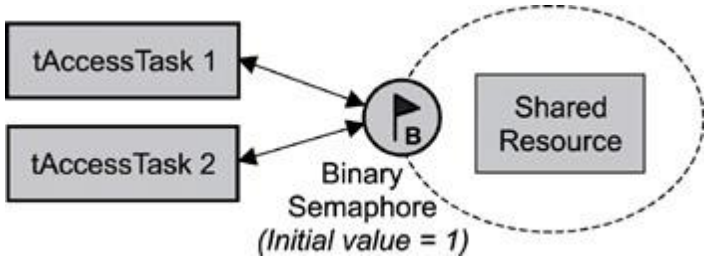


Figure 8: Single shared-resource-access synchronization.

In this scenario, a binary semaphore is initially created in the available state (value = 1) and is used to protect the shared resource. To access the shared resource, task 1 or 2 needs to first successfully acquire the binary semaphore before reading from or writing to the shared resource. The pseudo code for both tAccessTask 1 and 2 is similar to Listing 4.

Listing 4: Pseudo code for tasks accessing a shared resource.

```
tAccessTask ()
    {
          :
    Acquire binary semaphore token
    Read or write to shared resource
    Release binary semaphore token
        :
      }
```

This code serializes the access to the shared resource. If tAccessTask 1 executes first, it makes a request to acquire the semaphore and is successful because the semaphore is available. Having acquired the semaphore, this task is granted access to the shared resource and can read and write to it.

Meanwhile, the higher priority tAccessTask 2 wakes up and runs due to a timeout or some external event. It tries to access the same semaphore but is blocked because tAccessTask 1 currently has access to it. After tAccessTask 1 releases the semaphore, tAccessTask 2 is unblocked and starts to execute.

One of the dangers to this design is that any task can accidentally release the binary semaphore, even one that never acquired the semaphore in the first place. If this issue were to happen in this scenario, both tAccessTask 1 and tAccessTask 2 could end up acquiring the semaphore and reading and writing to the shared resource at the same time, which would lead to incorrect program behavior.

To ensure that this problem does not happen, use a mutex semaphore instead. Because a mutex supports the concept of ownership, it ensures that only the task that successfully acquired (locked) the mutex can release (unlock) it.

Recursive Shared-Resource-Access Synchronization

Sometimes a developer might want a task to access a shared resource recursively. This situation might exist if tAccessTask calls Routine A that calls Routine B, and all three need access to the same shared resource, as shown in Figure 9.

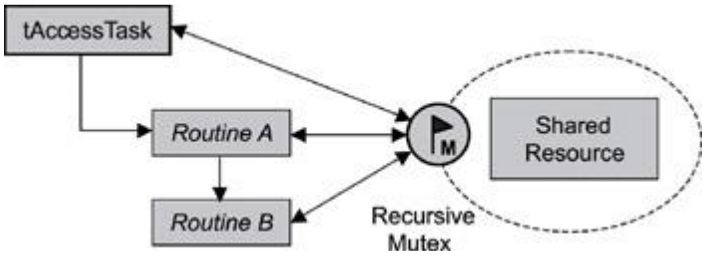


Figure 9: Recursive shared- resource-access synchronization.

If a semaphore were used in this scenario, the task would end up blocking, causing a deadlock. When a routine is called from a task, the routine effectively becomes a part of the task. When Routine A runs, therefore, it is running as a part of tAccessTask. Routine A trying to acquire the semaphore is effectively the same as tAccessTask trying to acquire the same semaphore. In this case, tAccessTask would end up blocking while waiting for the unavailable semaphore that it already has.

One solution to this situation is to use a recursive mutex. After tAccessTask locks the mutex, the task owns it. Additional attempts from the task itself or from routines that it calls to lock the mutex succeed. As a result, when Routines A and B attempt to lock the mutex, they succeed without blocking. The pseudo code for tAccessTask, Routine A, and Routine B are similar to Listing 5.

Listing 5: Pseudo code for recursively accessing a shared resource.

```
tAccessTask ()
   {
      :
      Acquire mutex
      Access shared resource
      Call Routine A
      Release mutex
      :
    }
Routine A ()
    {
        :
       Acquire mutex
      Access shared resource
      Call Routine B
      Release mutex
      :
     }
Routine B ()
```

```
{
   Acquire mutex
   Access shared resource
   Release mutex
 }
```

Multiple Shared-Resource-Access Synchronization

For cases in which multiple equivalent shared resources are used, a counting semaphore comes in handy, as shown in Figure 10.

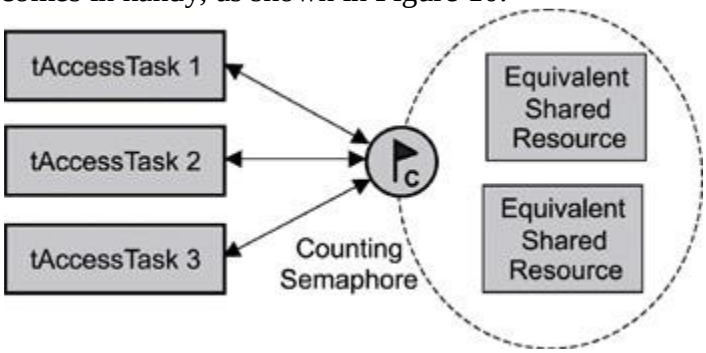


Figure 10: Multiple shared-resource-access synchronization.

Note that this scenario does not work if the shared resources are not equivalent. The counting semaphore's count is initially set to the number of equivalent shared resources: in this example, 2. As a result, the first two tasks requesting a semaphore token are successful. However, the third task ends up blocking until one of the previous two tasks releases a semaphore token, as shown in Listing 6. Note that similar code is used for tAccessTask 1, 2 and 3.

Listing.6: Pseudo code for multiple tasks accessing equivalent shared resources.

```
tAccessTask ()
  {
        :
        Acquire a counting semaphore token
        Read or Write to shared resource
        Release a counting semaphore token
        :
   }
```

As with the binary semaphores, this design can cause problems if a task releases a semaphore that it did not originally acquire. If the code is relatively simple, this issue might not be a problem. If the code is more elaborate, however, with many tasks accessing shared devices using multiple semaphores, mutexes can provide built-in protection in the application design.

As shown in Figure 9, a separate mutex can be assigned for each shared resource. When trying to lock a mutex, each task tries to acquire the first mutex in a non-blocking way. If unsuccessful, each task then tries to acquire the second mutex in a blocking way.

The code is similar to Listing 7. Note that similar code is used for tAccessTask 1, 2, and 3.

Listing 7: Pseudo code for multiple tasks accessing equivalent shared resources using mutexes.

```
tAccessTask ()
   {
        :
        Acquire first mutex in non-blocking way
        If not successful then acquire 2nd mutex in a blocking way
        Read or Write to shared resource
        Release the acquired mutex
        :
    }
```

Using this scenario, task 1 and 2 each is successful in locking a mutex and therefore having access to a shared resource. When task 3 runs, it tries to lock the first mutex in a non-blocking way (in case task 1 is done with the mutex). If this first mutex is unlocked, task 3 locks it and is granted access to the first shared resource. If the first mutex is still locked, however, task 3 tries to acquire the second mutex, except that this time, it would do so in a blocking way. If the second mutex is also locked, task 3 blocks and waits for the second mutex until it is unlocked.

Message Queues

Introduction

In many cases, task activity synchronization alone does not yield a sufficiently responsive application. Tasks must also be able to exchange messages. To facilitate inter-task data communication, kernels provide a message queue object and message queue management services.

Defining Message Queues

A message queue is a buffer-like object through which tasks and ISRs send and receive messages to communicate and synchronize with data. A message queue is like a pipeline. It temporarily holds messages from a sender until the intended receiver is ready to read them. This temporary buffering decouples a sending and receiving task; that is, it frees the tasks from having to send and receive messages simultaneously.

A message queue has several associated components that the kernel uses to manage the queue. When a message queue is first created, it is assigned an associated queue control block (QCB), a message queue name, a unique ID, memory buffers, a queue length, a maximum message length, and one or more task-waiting lists, as illustrated in Figure 11.

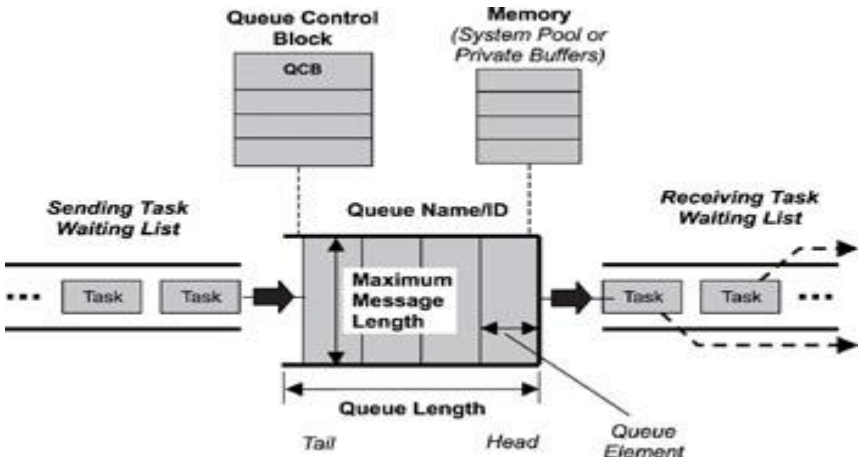


Figure 11: A message queue, its associated parameters, and supporting data structures.

It is the kernels job to assign a unique ID to a message queue and to create its QCB and task-waiting list. The kernel also takes developer-supplied parameters such as the length of the queue and the maximum message length to determine how much memory is required for the message queue. After the kernel has this information, it allocates memory for the message queue from either a pool of system memory or some private memory space.

The message queue itself consists of a number of elements, each of which can hold a single message. The elements holding the first and last messages are called the *head* and *tail* respectively. Some elements of the queue may be empty (not containing a message). The total number of elements (empty or not) in the queue is the *total length of the queue*.

As Figure 11 shows, a message queue has two associated task-waiting lists. The receiving task-waiting list consists of tasks that wait on the queue when it is empty. The sending list consists of tasks that wait on the queue when it is full.

Message Queue States

As with other kernel objects, message queues follow the logic of a simple FSM, as shown in Figure 12 When a message queue is first created, the FSM is in the empty state. If a task attempts to receive messages from this message queue while the queue is empty, the task blocks and, if it chooses to, is held on the message queue's task-waiting list, in either a FIFO or priority-based order.

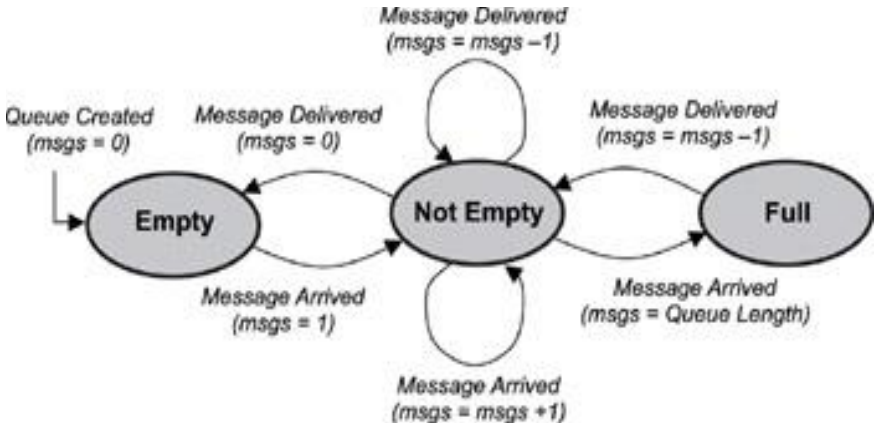


Figure 12: The state diagram for a message queue.

In this scenario, if another task sends a message to the message queue, the message is delivered directly to the blocked task. The blocked task is then removed from the task-waiting list and moved to either the ready or the running state. The message queue in this case remains empty because it has successfully delivered the message. If another message is sent to the same message queue and no tasks are waiting in the message queue's task-waiting list, the message queue's state becomes not empty.

As additional messages arrive at the queue, the queue eventually fills up until it has exhausted its free space. At this point, the number of messages in the queue is equal to the queue's length, and the message queue's state becomes full. While a message queue is in this state, any task sending messages to it will not be successful unless some other task first requests a message from that queue, thus freeing a queue element.

In some kernel implementations when a task attempts to send a message to a full message queue, the sending function returns an error code to that task. Other kernel implementations allow such a task to block, moving the blocked task into the sending task-waiting list, which is separate from the receiving task-waiting list.

Message Queue Content

Message queues can be used to send and receive a variety of data. Some examples include:

- a temperature value from a sensor,
- a bitmap to draw on a display,
- a text message to print to an LCD,
- a keyboard event, and
- a data packet to send over the network.

Some of these messages can be quite long and may exceed the maximum message length, which is determined when the queue is created. One way to overcome the limit on message length is to send a pointer to the data, rather than the data itself. Even if a long message might fit into the queue, it is sometimes better to send a pointer instead in order to improve both performance and memory utilization.

When a task sends a message to another task, the message normally is copied twice, as shown in Figure 13.

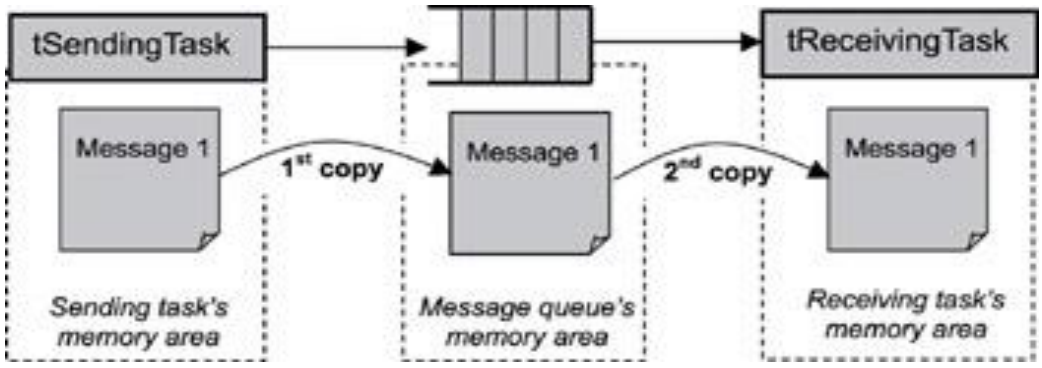


Figure 13: Message copying and memory use for sending and receiving messages.

The first time, the message is copied when the message is sent from the sending tasks memory area to the message queues memory area. The second copy occurs when the message is copied from the message queue s memory area to the receiving task s memory area.

An exception to this situation is if the receiving task is already blocked waiting at the message queue. Depending on a kernels implementation, the message might be copied just once in this case from the sending tasks memory area to the receiving task s memory area, bypassing the copy to the message queue s memory area.

Message Queue Storage

Different kernels store message queues in different locations in memory. One kernel might use a system pool, in which the messages of all queues are stored in one large shared area of memory. Another kernel might use separate memory areas, called private buffers, for each message queue.

System Pools

Using a system pool can be advantageous if it is certain that all message queues will never be filled to capacity at the same time. The advantage occurs because system pools typically save on memory use. The downside is that a message queue with large messages can easily use most of the pooled memory, not leaving enough memory for other message queues. Indications that this problem is occurring include a message queue that is not full that starts rejecting messages sent to it or a full message queue that continues to accept more messages.

Private Buffers

Using private buffers, on the other hand, requires enough reserved memory area for the full capacity of every message queue that will be created. This approach clearly uses up more memory;

however, it also ensures that messages do not get overwritten and that room is available for all messages, resulting in better reliability than the pool approach.

Message Queue Operations

Typical message queue operations include the following:

- creating and deleting message queues,
- sending and receiving messages, and
- obtaining message queue information.

Creating and Deleting Message Queues

Message queues can be created and deleted by using two simple calls, as shown in Table 5.

Table 5: Message queue creation and deletion operations.

Operation	Description
Create	Creates a message queue
Delete	Deletes a message queue

When created, message queues are treated as global objects and are not owned by any particular task. Typically, the queue to be used by each group of tasks or ISRs is assigned in the design.

When creating a message queue, a developer needs to make some initial decisions about the length of the message queue, the maximum size of the messages it can handle, and the waiting order for tasks when they block on a message queue.

Deleting a message queue automatically unblocks waiting tasks. The blocking call in each of these tasks returns with an error. Messages that were queued are lost when the queue is deleted.

Sending and Receiving Messages

The most common uses for a message queue are sending and receiving messages. These operations are performed in different ways, some of which are listed in Table 6 .

Table 6: Sending and receiving messages.

Operation	Description
Send	Sends a message to a message queue
Receive	Receives a message from a message queue
Broadcast	Broadcasts messages

Sending Messages

When sending messages, a kernel typically fills a message queue from head to tail in FIFO order, as shown in Figure 14. Each new message is placed at the end of the queue.

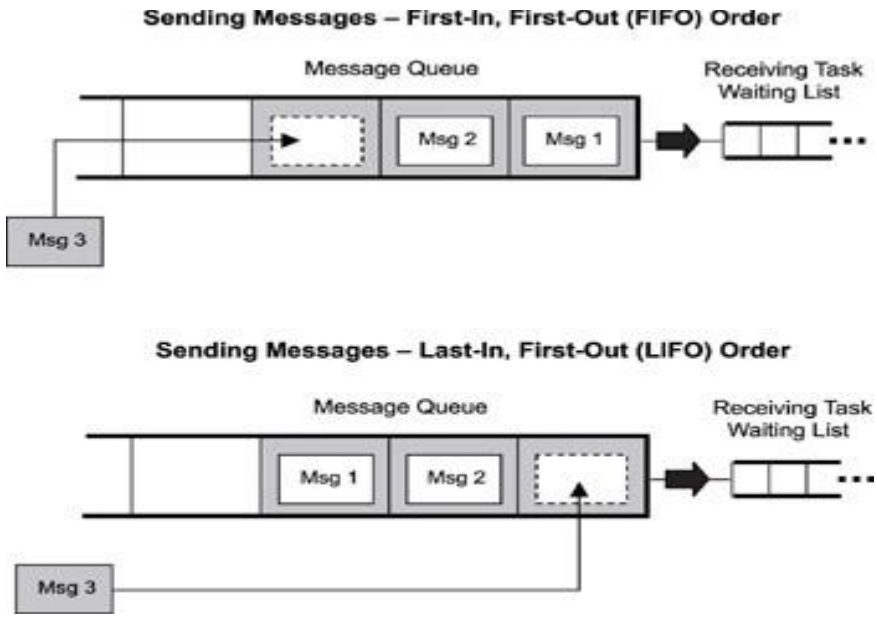


Figure 7.4: Sending messages in FIFO or LIFO order.

Many message-queue implementations allow urgent messages to go straight to the head of the queue. If all arriving messages are urgent, they all go to the head of the queue, and the queuing order effectively becomes last-in/first-out (LIFO). Many message-queue implementations also allow ISRs to send messages to a message queue. In any case, messages are sent to a message queue in the following ways:

- not block (ISRs and tasks),
- block with a timeout (tasks only), and
- block forever (tasks only).

At times, messages must be sent without blocking the sender. If a message queue is already full, the send call returns with an error, and the task or ISR making the call continues executing. This type of approach to sending messages is the only way to send messages from ISRs, because ISRs cannot block.

Most times, however, the system should be designed so that a task will block if it attempts to send a message to a queue that is full. Setting the task to block either forever or for a specified timeout accomplishes this step. (Figure 15). The blocked task is placed in the message queue s task-waiting list, which is set up in either FIFO or priority-based order.

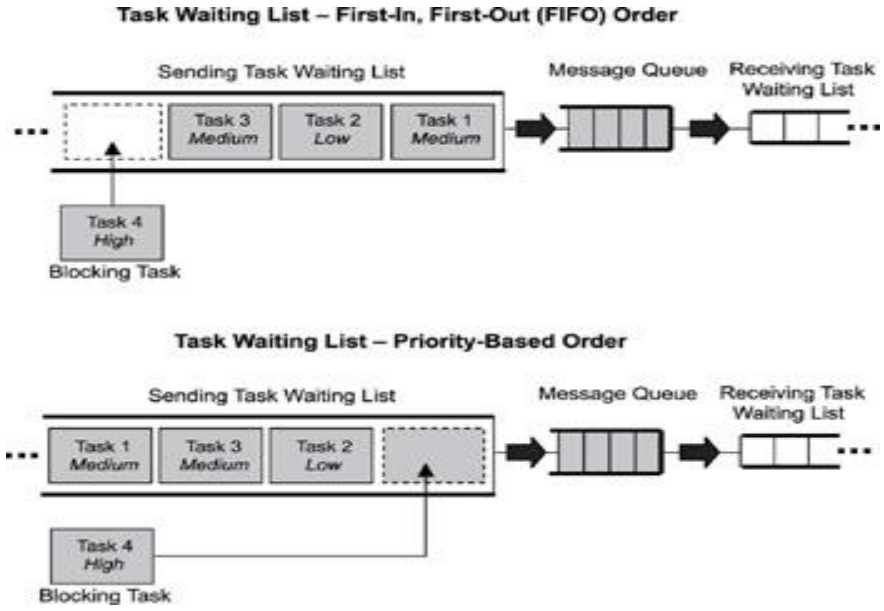


Figure 15: FIFO and priority-based task-waiting lists.

In the case of a task set to block forever when sending a message, the task blocks until a message queue element becomes free (e.g., a receiving task takes a message out of the queue). In the case of a task set to block for a specified time, the task is unblocked if either a queue element becomes free or the timeout expires, in which case an error is returned.

Receiving Messages

As with sending messages, tasks can receive messages with different blocking policies the same way as they send them with a policy of not blocking, blocking with a timeout, or blocking forever. Note, however, that in this case, the blocking occurs due to the message queue being empty, and the receiving tasks wait in either a FIFO or priority based order. The diagram for the receiving tasks is similar to Figure 15, except that the blocked receiving tasks are what fill the task list.

For the message queue to become full either the receiving task list must be empty or the rate at which messages are posted in the message queue must be greater than the rate at which messages are removed. Only when the message queue is full does the task-waiting list for sending tasks start to fill. Conversely, for the task-waiting list for receiving tasks to start to fill, the message queue must be empty.

Messages can be read from the head of a message queue in two different ways:

- destructive read, and
- non-destructive read.

In a destructive read, when a task successfully receives a message from a queue, the task permanently removes the message from the message queues storage buffer. In a non-destructive read, a receiving task peeks at the message at the head of the queue without removing it. Both ways of reading a message can be useful; however, not all kernel implementations support the non-destructive read.

Some kernels support additional ways of sending and receiving messages. One way is the example of peeking at a message, other kernels allow broadcast messaging.

Obtaining Message Queue Information

Obtaining message queue information can be done from an application by using the operations listed in Table 7.

Table 7: Obtaining message queue information operations.

Operation	Description
Show queue info	Gets information on a message queue
Show queues task-waiting list	Gets a list of tasks in the queue s task-waiting list

Different kernels allow developers to obtain different types of information about a message queue, including the message queue ID, the queuing order used for blocked tasks (FIFO or priority-based), and the number of messages queued. Some calls might even allow developers to get a full list of messages that have been queued up.

As with other calls that get information about a particular kernel object, be careful when using these calls. The information is dynamic and might have changed by the time it's viewed. These types of calls should only be used for debugging purposes.

Message Queue Use

The following are typical ways to use message queues within an application:

- non-interlocked, one-way data communication,
- interlocked, one-way data communication,
- interlocked, two-way data communication, and
- broadcast communication.

Non-Interlocked, One-Way Data Communication

One of the simplest scenarios for message-based communications requires a sending task (also called the message source), a message queue, and a receiving task (also called a message sink), as illustrated in Figure 16.

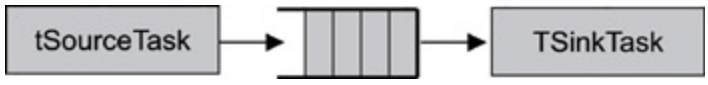


Figure 16: Non-interlocked, one-way data communication.

This type of communication is also called non-interlocked (or loosely coupled), one-way data communication. The activities of tSourceTask and tSinkTask are not synchronized. TSourceTask simply sends a message; it does not require acknowledgement from tSinkTask.

The pseudo code for this scenario is provided in Listing 8.

Listing 8: Pseudo code for non-interlocked, one-way data communication.

```
tSourceTask ()
{
        :
     Send message to message queue
        :
}
tSinkTask ()
{
       :
   Receive message from message queue
       :
}
```

If tSinkTask is set to a higher priority, it runs first until it blocks on an empty message queue. As soon as tSourceTask sends the message to the queue, tSinkTask receives the message and starts to execute again. If tSinkTask is set to a lower priority, tSourceTask fills the message queue with messages. Eventually, tSourceTask can be made to block when sending a message to a full message queue. This action makes tSinkTask wake up and starts taking messages out of the message queue.

ISRs typically use non-interlocked, one-way communication. A task such as tSinkTask runs and waits on the message queue. When the hardware triggers an ISR to run, the ISR puts one or more messages into the message queue. After the ISR completes running, tSinkTask gets an opportunity to run (if it s the highest-priority task) and takes the messages out of the message queue.

When ISRs send messages to the message queue, they must do so in a non-blocking way. If the message queue becomes full, any additional messages that the ISR sends to the message queue are lost.

Interlocked, One-Way Data Communication

In some designs, a sending task might require a handshake (acknowledgement) that the receiving task has been successful in receiving the message. This process is called interlocked communication, in which the sending task sends a message and waits to see if the message is received. This requirement can be useful for reliable communications or task synchronization.

For example, if the message for some reason is not received correctly, the sending task can resend it. Using interlocked communication can close a synchronization loop. To do so, you can construct a continuous loop in which sending and receiving tasks operate in lockstep with each other. An example of one-way, interlocked data communication is illustrated in Figure 17.

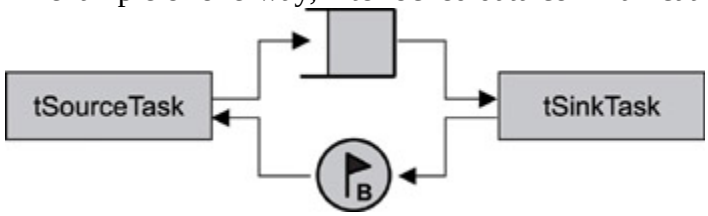


Figure 17: Interlocked, one-way data communication.

In this case, tSourceTask and tSinkTask use a binary semaphore initially set to 0 and a message queue with a length of 1 (also called a mailbox). tSourceTask sends the message to the message queue and blocks on the binary semaphore. tSinkTask receives the message and increments the binary semaphore. The semaphore that has just been made available wakes up tSourceTask. tSourceTask, which executes and posts another message into the message queue, blocking again afterward on the binary semaphore.

The pseudo code for interlocked, one-way data communication is provided in Listing 9.

Listing 9: Pseudo code for interlocked, one-way data communication.

```
tSourceTask ()
{
          :
       Send message to message queue
       Acquire binary semaphore
            :
}
tSinkTask ()
{
          :
     Receive message from message queue
     Give binary semaphore
          :
}
```

The semaphore in this case acts as a simple synchronization object that ensures that tSourceTask and tSinkTask are in lockstep. This synchronization mechanism also acts as a simple acknowledgement to tSourceTask that it s okay to send the next message.

Interlocked, Two-Way Data Communication

Sometimes data must flow bidirectionally between tasks, which is called interlocked, two-way data communication (also called full-duplex or tightly coupled communication). This form of communication can be useful when designing a client/server-based system. A diagram is provided in Figure 18.

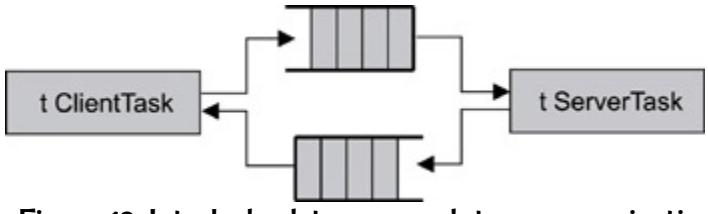


Figure 18: Interlocked, two-way data communication.

In this case, tClientTask sends a request to tServerTask via a message queue. tServer-Task fulfills that request by sending a message back to tClientTask.

The pseudo code is provided in Listing 10.

Listing 10: Pseudo code for interlocked, two-way data communication.

```
tClientTask ()
{
          :
    Send a message to the requests queue
    Wait for message from the server queue
          :
}
tServerTask ()
{
          :
    Receive a message from the requests queue
    Send a message to the client queue
          :
}
```

Note that two separate message queues are required for full-duplex communication. If any kind of data needs to be exchanged, message queues are required; otherwise, a simple semaphore can be used to synchronize acknowledgement.

In the simple client/server example, tServerTask is typically set to a higher priority, allowing it to quickly fulfill client requests. If multiple clients need to be set up, all clients can use the client message queue to post requests, while tServerTask uses a separate message queue to fulfill the different client's requests.

Broadcast Communication

Some message-queue implementations allow developers to broadcast a copy of the same message to multiple tasks, as shown in Figure 19.

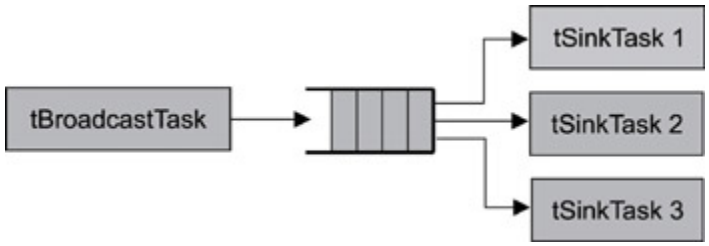


Figure 19: Broadcasting messages.

Message broadcasting is a one-to-many-task relationship. tBroadcastTask sends the message on which multiple tSink-Task are waiting.

Pseudo code for broadcasting messages is provided in Listing 11.

Listing 11: Pseudo code for broadcasting messages.

```
tBroadcastTask ()
{
     :
  Send broadcast message to queue
     :
}
```

```
Note: similar code for tSignalTasks 1, 2, and 3.
tSignalTask ()
{
        :
    Receive message on queue
        :
}
```

In this scenario, tSinkTask 1, 2, and 3 have all made calls to block on the broadcast message queue, waiting for a message. When tBroadcastTask executes, it sends one message to the message queue, resulting in all three waiting tasks exiting the blocked state.

Pipes

Pipes are kernel objects that provide unstructured data exchange and facilitate synchronization among tasks. In a traditional implementation, a pipe is a unidirectional data exchange facility, as shown in Figure 20. Two descriptors, one for each end of the pipe (one end for reading and one for writing), are returned when the pipe is created. Data is written via one descriptor and read via the other. The data remains in the pipe as an unstructured byte stream. Data is read from the pipe in FIFO order.

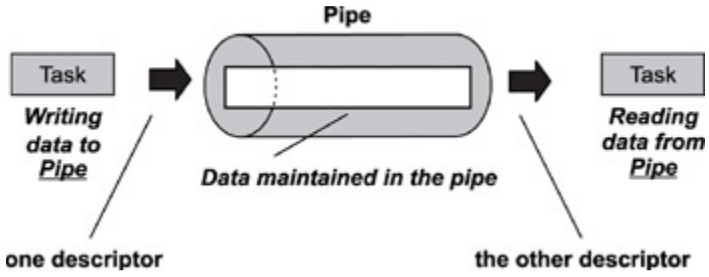


Figure 20: A common pipe unidirectional.

A pipe provides a simple data flow facility so that the reader becomes blocked when the pipe is empty, and the writer becomes blocked when the pipe is full. Typically, a pipe is used to exchange data between a data-producing task and a data-consuming task, as shown in Figure 21. It is also permissible to have several writers for the pipe with multiple readers on it.

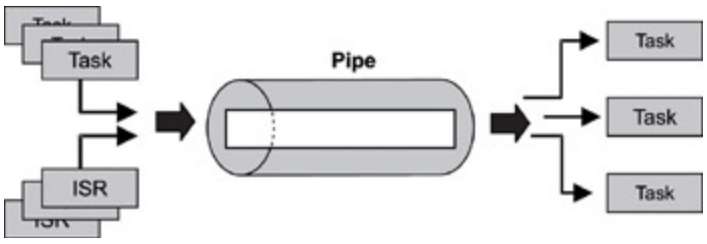


Figure 21: Common pipe operation.

Note that a pipe is conceptually similar to a message queue but with significant differences. For example, unlike a message queue, a pipe does not store multiple messages. Instead, the data that it stores is not structured, but consists of a stream of bytes. Also, the data in a pipe cannot be prioritized; the data flow is strictly first-in, first-out FIFO. Finally, as is described below, pipes support the powerful select operation, and message queues do not.

Pipe Control Blocks

Pipes can be dynamically created or destroyed. The kernel creates and maintains pipe-specific information in an internal data structure called a *pipe control block.* The structure of the pipe control block varies from one implementation to another. In its general form, a pipe control block contains a kernel-allocated data buffer for the pipes input and output operation. The size of this buffer is maintained in the control block and is fixed when the pipe is created; it cannot be altered at run time.

The current data byte count, along with the current input and output position indicators, are part of the pipe control block. The current data byte count indicates the amount of readable data in the pipe. The input position specifies where the next write operation begins in the buffer. Similarly, the output position specifies where the next read operation begins. The kernel creates two descriptors that are unique within the system I/O space and returns these descriptors to the creating task. These descriptors identify each end of the pipe uniquely.

Two task-waiting lists are associated with each pipe, as shown in Figure 22. One waiting list keeps track of tasks that are waiting to write into the pipe while it is full; the other keeps track of tasks that are waiting to read from the pipe while it is empty.

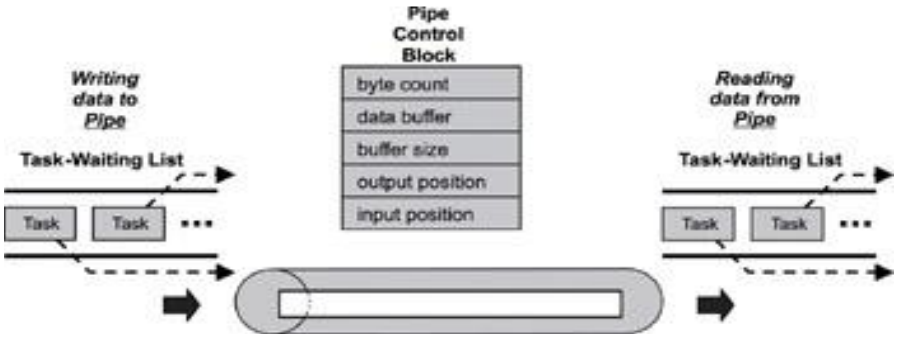


Figure 22: Pipe control block.

Pipe States

A pipe has a limited number of states associated with it from the time of its creation to its termination. Each state corresponds to the data transfer state between the reader and the writer of the pipe, as illustrated in Figure 23.

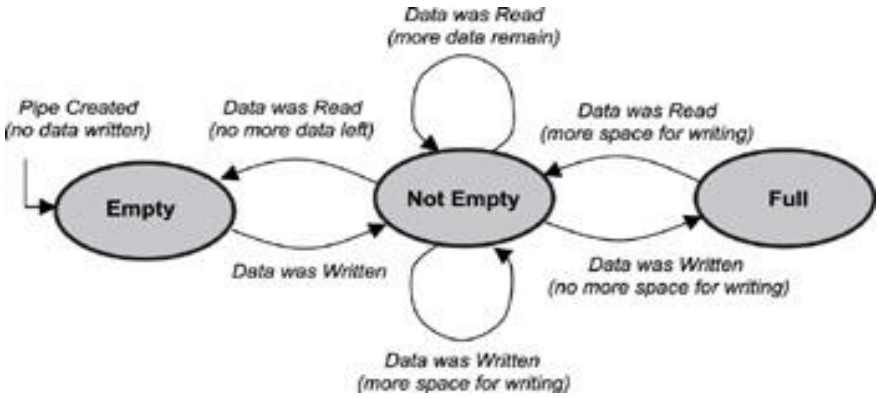


Figure 23: States of a pipe.

Named and Unnamed Pipes

A kernel typically supports two kinds of pipe objects: named pipes and unnamed pipes. A *named pipe* , also known as FIFO, has a name similar to a file name and appears in the file system as if it were a file or a device. Any task or ISR that needs to use the named pipe can reference it by name. The *unnamed pipe* does not have a name and does not appear in the file system. It must be referenced by the descriptors that the kernel returns when the pipe is created.

Typical Pipe Operations

The following set of operations can be performed on a pipe:

- create and destroy a pipe,
- read from or write to a pipe,
- issue control commands on the pipe, and
- select on a pipe.

Create and Destroy

Create and destroy operations are available, as shown in Table 8.

Table 8: Create and destroy operations.

Operation	Description
Pipe	Creates a pipe
Open	Opens a pipe
Close	Deletes or closes a pipe

The pipe operation creates an unnamed pipe. This operation returns two descriptors to the calling task, and subsequent calls reference these descriptors. One descriptor is used only for writing, and the other descriptor is used only for reading.

Creating a named pipe is similar to creating a file; the specific call is implementation-dependent. Some common names for such a call are ***mknod*** and ***mkfifo***. Because a named pipe has a recognizable name in the file system after it is created, the pipe can be opened using the open operation. The calling task must specify whether it is opening the pipe for the read operation or for the write operation; it cannot be both.

The close operation is the counterpart of the open operation. Similar to open, the close operation can only be performed on a named pipe. Some implementations will delete the named pipe permanently once the close operation completes.

Read and Write

Read and write operations are available, as shown in Table 9.

Table 9: Read and write operations.

Operation	Description
Read	Reads from the pipe
Write	Writes to a pipe

The read operation returns data from the pipe to the calling task. The task specifies how much data to read. The task may choose to block waiting for the remaining data to arrive if the size specified exceeds what is available in the pipe.

Remember that a read operation on a pipe is a destructive operation because data is removed from a pipe during this operation, making it unavailable to other readers. Therefore, unlike a message queue, a pipe cannot be used for broadcasting data to multiple reader tasks.

A task, however, can consume a block of data originating from multiple writers during one read operation.

The write operation appends new data to the existing byte stream in the pipe. The calling task specifies the amount of data to write into the pipe. The task may choose to block waiting for additional buffer space to become free when the amount to write exceeds the available space.

No message boundaries exist in a pipe because the data maintained in it is unstructured. This issue represents the main structural difference between a pipe and a message queue. Because there are no message headers, it is impossible to determine the original producer of the data bytes.

Another important difference between message queues and pipes is that data written to a pipe cannot be prioritized. Because each byte of data in a pipe has the same priority, a pipe should not be used when urgent data must be exchanged between tasks.

Control

Control operations are available, as shown in Table 10.

Table 10: Control operations.

Operation	Description
Fcntl	Provides control over the pipe descriptor

The *Fcntl* operation provides generic control over a pipe s descriptor using various commands, which control the behavior of the pipe operation. For example, a commonly implemented command is the non-blocking command. The command controls whether the calling task is blocked if a read operation is performed on an empty pipe or when a write operation is performed on a full pipe.

Another common command that directly affects the pipe is the flush command. The flush command removes all data from the pipe and clears all other conditions in the pipe to the same state as when the pipe was created. Sometimes a task can be preempted for too long, and when it finally gets to read data from the pipe, the data might no longer be useful. Therefore, the task can flush the data from the pipe and reset its state.

Select

Select operations are available, as shown in Table 11.

Table 11: Select operations.

Operation	Description
Select	Waits for conditions to occur on a pipe

The *select* operation allows a task to block and wait for a specified condition to occur on one or more pipes. The wait condition can be waiting for data to become available or waiting for data to be emptied from the pipe(s).

Figure 24. illustrates a scenario in which a single task is waiting to read from two pipes and write to a third. In this case, the select call returns when data becomes available on either of the top two pipes. The same select call also returns when space for writing becomes available on the bottom pipe.

In general, a task reading from multiple pipes can perform a select operation on those pipes, and the select call returns when any one of them has data available. Similarly, a task writing to multiple pipes can perform a select operation on the pipes, and the select call returns when space becomes available on any one of them.

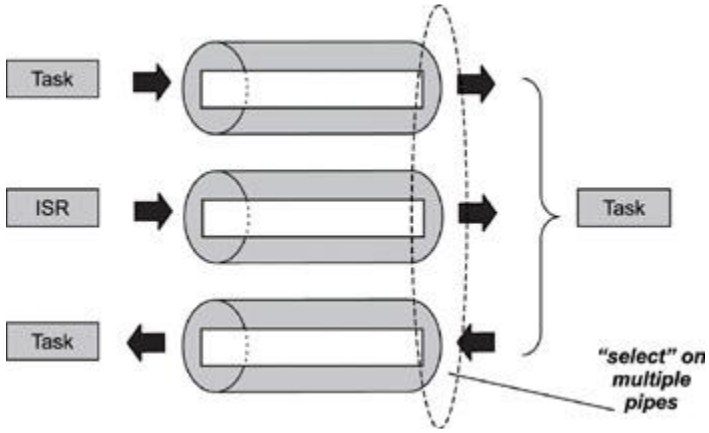


Figure 24: The select operation on multiple pipes.

In contrast to pipes, message queues do not support the select operation. Thus, while a task can have access to multiple message queues, it cannot block-wait for data to arrive on any one of a group of empty message queues.

The same restriction applies to a writer. In this case, a task can write to multiple message queues, but a task cannot block-wait on a group of full message queues, while waiting for space to become available on any one of them.

It becomes clear then that the main advantage of using a pipe over a message queue for intertask communication is that it allows for the select operation.

Typical Uses of Pipes

Because a pipe is a simple data channel, it is mainly used for task-to-task or ISR-to-task data transfer, as illustrated in Figure 20 and Figure 21. Another common use of pipes is for inter-task synchronization.

Inter-task synchronization can be made asynchronous for both tasks by using the select operation. In Figure 25, task A and task B open two pipes for inter-task communication. The first pipe is opened for data transfer from task A to task B. The second pipe is opened for acknowledgement (another data transfer) from task B to task A. Both tasks issue the select operation on the pipes.

Task A can wait asynchronously for the data pipe to become writeable (task B has read some data from the pipe). That is, task A can issue a non-blocking call to write to the pipe and perform other operations until the pipe becomes writeable.

Task A can also wait asynchronously for the arrival of the transfer acknowledgement from task B on the other pipe. Similarly, task B can wait asynchronously for the arrival of data on the data pipe and wait for the other pipe to become writeable before sending the transfer acknowledgement.

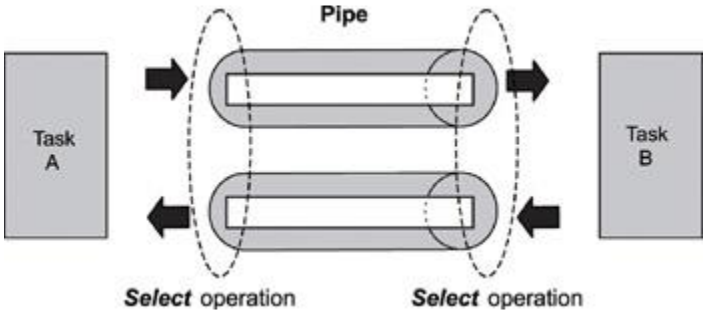


Figure 25: Using pipes for inter-task synchronization.

Event Registers

Some kernels provide a special register as part of each task s control block, as shown in Figure 26. This register, called an *event register,* is an object belonging to a task and consists of a group of binary event flags used to track the occurrence of specific events. Depending on a given

kernel s implementation of this mechanism, an event register can be 8-, 16-, or 32-bits wide, maybe even more. Each bit in the event register is treated like a binary flag (also called an event flag) and can be either set or cleared.

Through the event register, a task can check for the presence of particular events that can control its execution. An external source, such as another task or an ISR, can set bits in the event register to inform the task that a particular event has occurred.

Applications define the event associated with an event flag. This definition must be agreed upon between the event sender and receiver using the event registers.

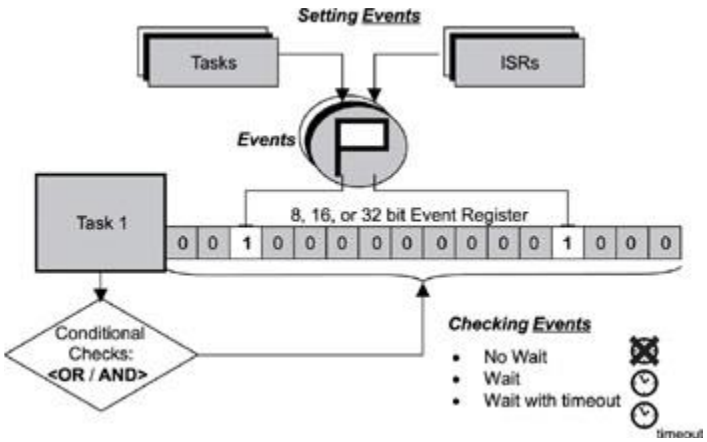


Figure 26: Event register.

Event Register Control Blocks

Typically, when the underlying kernel supports the event register mechanism, the kernel creates an event register control block as part of the task control block when creating a task, as shown in Figure 27.

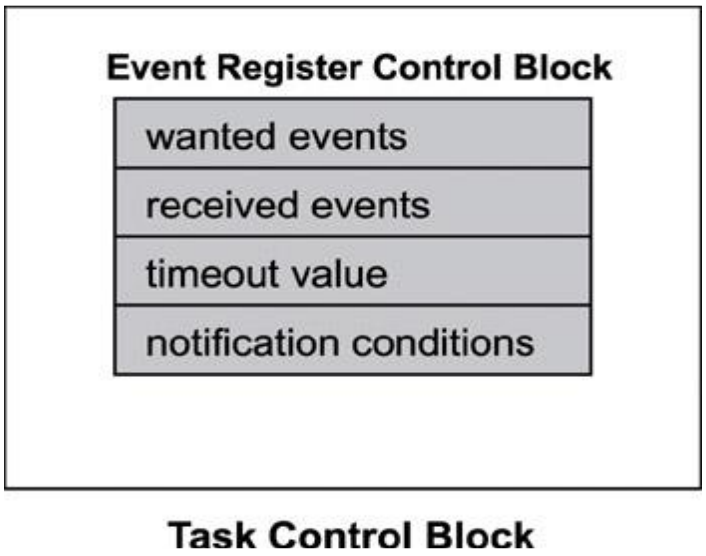


Figure 27: Event register control block.

The task specifies the set of events it wishes to receive. This set of events is maintained in the wanted events register. Similarly, arrived events are kept in the received events register. The task indicates a timeout to specify how long it wishes to wait for the arrival of certain events. The kernel wakes up the task when this timeout has elapsed if no specified events have arrived at the task.

Using the notification conditions, the task directs the kernel as to when it wishes to be notified (awakened) upon event arrivals. For example, the task can specify the notification conditions as send notification when both event type 1 and event type 3 arrive or when event type 2 arrives. This option provides flexibility in defining complex notification patterns.

Typical Event Register Operations

Two main operations are associated with an event register, the sending and the receiving operations, as shown in Table 12.

Table 12: Event register operations.

Operation	Description
Send	Sends events to a task
Receive	Receives events

The receive operation allows the calling task to receive events from external sources. The task can specify if it wishes to wait, as well as the length of time to wait for the arrival of desired events before giving up. The task can wait forever or for a specified interval.

Specifying a set of events when issuing the receive operation allows a task to block-wait for the arrival of multiple events, although events might not necessarily all arrive simultaneously. The kernel translates this event set into the notification conditions. The receive operation returns either when the notification conditions are satisfied or when the timeout has occurred. Any received events that are not indicated in the receive operation are left pending in the received events register of the event register control block. The receive operation returns immediately if the desired events are already pending.

The event set is constructed using the bit-wise AND/OR operation. With the AND operation, the task resumes execution only after every event bit from the set is on. A task can also block-wait for the arrival of a single event from an event set, which is constructed using the bit-wise OR operation. In this case, the task resumes execution when any one event bit from the set is on.

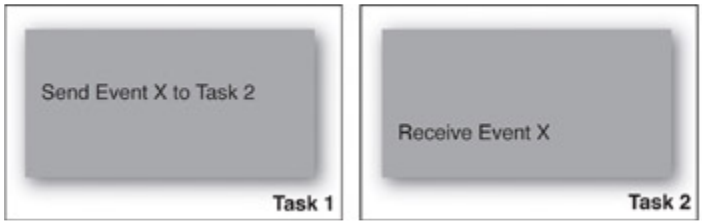
The send operation allows an external source, either a task or an ISR, to send events to another task. The sender can send multiple events to the designated task through a single send operation. Events that have been sent and are pending on the event bits but have not been chosen for reception by the task remains pending in the received events register of the event register control block.

Events in the event register are not queued. An event register cannot count the occurrences of the same event while it is pending; therefore, subsequent occurrences of the same event are lost. For example, if an ISR sends an event to a task and the event is left pending; and later another task sends the same event again to the same task while it is still pending, the first occurrence of the event is lost.

Typical Uses of Event Registers

Event registers are typically used for unidirectional activity synchronization. It is unidirectional because the issuer of the receive operation determines when activity synchronization should take place. Pending events in the event register do not change the execution state of the receiving task.

In following the diagram, at the time task 1 sends the event X to task 2, no effect occurs to the execution state of task2 if task 2 has not yet attempted to receive the event.



No data is associated with an event when events are sent through the event register. Other mechanisms must be used when data needs to be conveyed along with an event. This lack of associated data can sometimes create difficulties because of the noncumulative nature of events in the event register. Therefore, the event register by itself is an inefficient mechanism if used beyond simple activity synchronization.

Another difficulty in using an event register is that it does not have a built-in mechanism for identifying the source of an event if multiple sources are possible. One way to overcome this problem is for a task to divide the event bits in the event register into subsets.

The task can then associate each subset with a known source. In this way, the task can identify the source of an event if each relative bit position of each subset is assigned to the same event type.

In Figure 28, an event register is divided into 4-bit groups. Each group is assigned to a source, regardless of whether it is a task or an ISR. Each bit of the group is assigned to an event type.

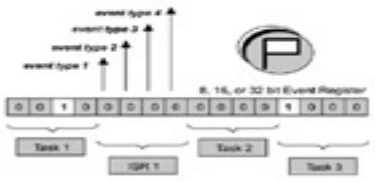


Figure 28: Identifying an event source.

Signals

A *signal* is a software interrupt that is generated when an event has occurred. It diverts the signal receiver from its normal execution path and triggers the associated asynchronous processing. Essentially, signals notify tasks of events that occurred during the execution of other tasks or ISRs.

As with normal interrupts, these events are asynchronous to the notified task and do not occur at any predetermined point in the tasks execution.

The difference between a signal and a normal interrupt is that signals are so-called software interrupts, which are generated via the execution of some software within the system. By contrast, normal interrupts are usually generated by the arrival of an interrupt signal on one of the CPUs external pins. They are not generated by software within the system but by external devices.

The number and type of signals defined is both system-dependent and RTOS-dependent. An easy way to understand signals is to remember that each signal is associated with an event. The event can be either unintentional, such as an illegal instruction encountered during program execution, or the event may be intentional, such as a notification to one task from another that it is about to terminate.

While a task can specify the particular actions to undertake when a signal arrives, the task has no control over when it receives signals. Consequently, the signal arrivals often appear quite random, as shown in Figure 29.

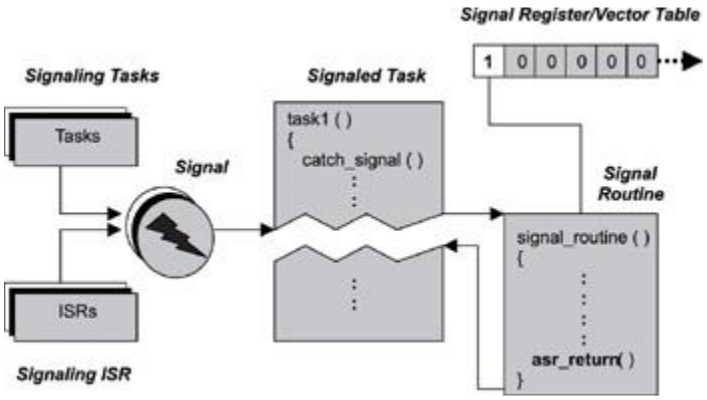


Figure 29: Signals.

When a signal arrives, the task is diverted from its normal execution path, and the corresponding signal routine is invoked. The terms *signal routine*, *signal handler*, *asynchronous event handler*, and *asynchronous signal routine* are interchangeable. This book uses *asynchronous signal routine* (ASR). Each signal is identified by an integer value, which is the *signal number* or *vector number*.

Signal Control Blocks

If the underlying kernel provides a signal facility, it creates the signal control block as part of the task control block as shown in Figure 30.

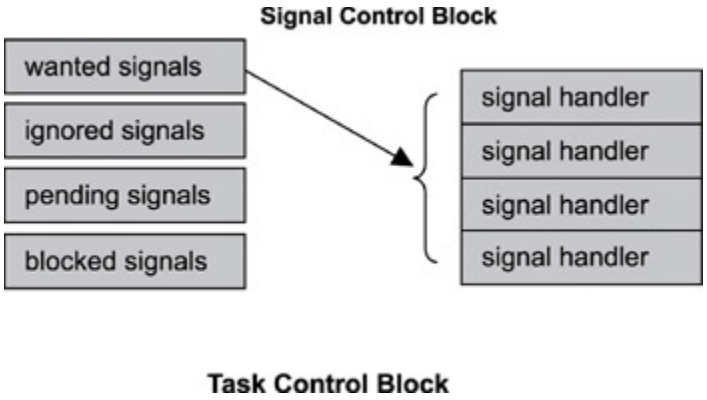


Figure 30: Signal control block.

The signal control block maintains a set of signals the wanted signals which the task is prepared to handle. When a task is prepared to handle a signal, it is often said, the task is *ready to catch* the signal.

When a signal interrupts a task, it is often said, the signal is *raised* to the task. The task can provide a signal handler for each signal to be processed, or it can execute a default handler that the kernel provides. It is possible to have a single handler for multiple types of signals.

Signals can be ignored, made pending, processed (handled), or blocked. The signals to be ignored by the task are maintained in the ignored signals set. Any signal in this set does not interrupt the task.

Other signals can arrive while the task is in the midst of processing another signal. The additional signal arrivals are kept in the pending signals set. The signals in this set are raised to the task as soon as the task completes processing the previous signal. The pending signals set is a subset of the wanted signals set.

To process a particular signal, either the task-supplied signal handler can be used for signal processing or the default handler supplied by the underlying kernel can be used to process it. It is also possible for the task to process the signal first and then pass it on for additional processing by the default handler.

A fourth kind of response to a signal is possible. In this case, a task does not ignore the signal but blocks the signal from delivery during certain stages of the task s execution when it is critical that the task not be interrupted.

Typical Signal Operations

Signal operations are available, as shown in Table 13.

Table 13: Signal operations.

Operation	Description
Catch	Installs a signal handler
Release	Removes a previously installed handler
Send	Sends a signal to another task

Ignore	Prevents a signal from being delivered
Block	Blocks a set of signal from being delivered
Unblock	Unblocks the signals so they can be delivered

A task can catch a signal after the task has specified a handler (ASR) for the signal. The catch operation installs a handler for a particular signal. The kernel interrupts the tasks execution upon the arrival of the signal, and the handler is invoked. The task can install the kernel-supplied default handler, the *default actions*, for any signal.

The task-installed handler has the options of either processing the signal and returning control to the kernel or processing the signal and passing control to the default handler for additional processing. Handling signals is similar to handling hardware interrupts, and the nature of the ASR is similar to that of the interrupt service routine.

After a handler has been installed for a particular signal, the handler is invoked if the same type of signal is received by any task, not just the one that installed it. In addition, any task can change the handler installed for a particular signal. Therefore, it is good practice for a task to save the previously installed handler before installing its own and then to restore that handler after it finishes catching the handler s corresponding signal.

Figure.31 shows the signal vector table, which the kernel maintains. Each element in the vector table is a pointer or offset to an ASR. For signals that don t have handlers assigned, the corresponding elements in the vector table are NULL. The example shows the table after three catch operations have been performed. Each catch operation installs one ASR, by writing a pointer or offset to the ASR into an element of the vector table.

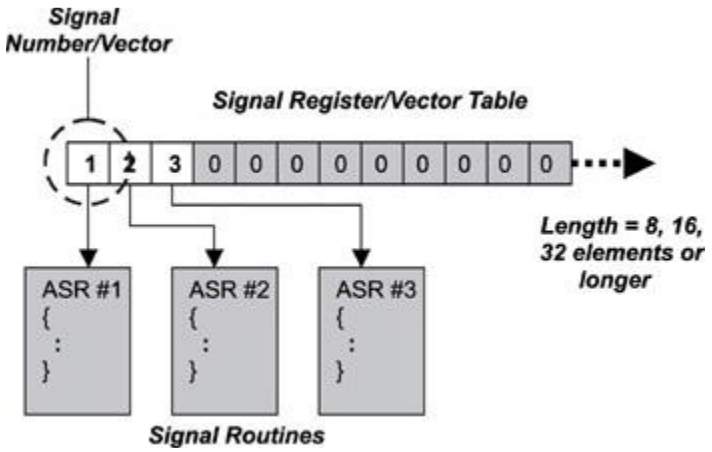


Figure 31: The catch operation.

The release operation de-installs a signal handler. It is good practice for a task to restore the previously installed signal handler after calling release.

The send operation allows one task to send a signal to another task. Signals are usually associated with hardware events that occur during execution of a task, such as generation of an unaligned memory address or a floating-point exception. Such signals are generated automatically

when their corresponding events occur. The send operation, by contrast, enables a task to explicitly generate a signal.

The ignore operation allows a task to instruct the kernel that a particular set of signals should never be delivered to that task. Some signals, however, cannot be ignored; when these signals are generated, the kernel calls the default handler.

The block operation does not cause signals to be ignored but temporarily prevents them from being delivered to a task. The block operation protects critical sections of code from interruption.

Another reason to block a signal is to prevent conflict when the signal handler is already executing and is in the midst of processing the same signal. A signal remains pending while it s blocked. The unblock operation allows a previously blocked signal to pass. The signal is delivered immediately if it is already pending.

Typical Uses of Signals

Some signals are associated with hardware events and thus are usually sent by hardware ISRs. The ISR is responsible for immediately responding to these events. The ISR, however, might also send a signal so that tasks affected by these hardware events can conduct further, task-specific processing.

As depicted in Figure 29, signals can also be used for synchronization between tasks. Signals, however, should be used sparingly for the following reasons:

- Using signals can be expensive due to the complexity of the signal facility when used for inter-task synchronization. A signal alters the execution state of its destination task. Because signals occur asynchronously, the receiving task becomes nondeterministic, which can be undesirable in a real-time system.
- Many implementations do not support queuing or counting of signals. In these implementations, multiple occurrences of the same signal overwrite each other. For example, a signal delivered to a task multiple times before its handler is invoked has the same effect as a single delivery. The task has no way to determine if a signal has arrived multiple times.
- Many implementations do not support signal delivery that carries information, so data cannot be attached to a signal during its generation.
- Many implementations do not support a signal delivery order, and signals of various types are treated as having equal priority, which is not ideal. For example, a signal triggered by a page fault is obviously more important than a signal generated by a task indicating it is about to exit. On an equal-priority system, the page fault might not be handled first.
- Many implementations do not guarantee when an unblocked pending signal will be delivered to the destination task. Some kernels do implement real-time extensions to traditional signal handling, which allows
  - for the prioritized delivery of a signal based on the signal number,
  - each signal to carry additional information, and
  - multiple occurrences of the same signal to be queued.

Condition Variables

Tasks often use shared resources, such as files and communication channels. When a task needs to use such a resource, it might need to wait for the resource to be in a particular state. The way the resource reaches that state can be through the action of another task. In such a scenario, a task needs some way to determine the condition of the resource.

One way for tasks to communicate and determine the condition of a shared resource is through a condition variable. A *condition variable* is a kernel object that is associated with a shared resource, which allows one task to wait for other task(s) to create a desired condition in the shared resource. A condition variable can be associated with multiple conditions.

As shown in Figure.32, a condition variable implements a predicate. The predicate is a set of logical expressions concerning the conditions of the shared resource. The predicate evaluates to either true or false. A task evaluates the predicate. If the evaluation is true, the task assumes that the conditions are satisfied, and it continues execution.

Otherwise, the task must wait for other tasks to create the desired conditions.

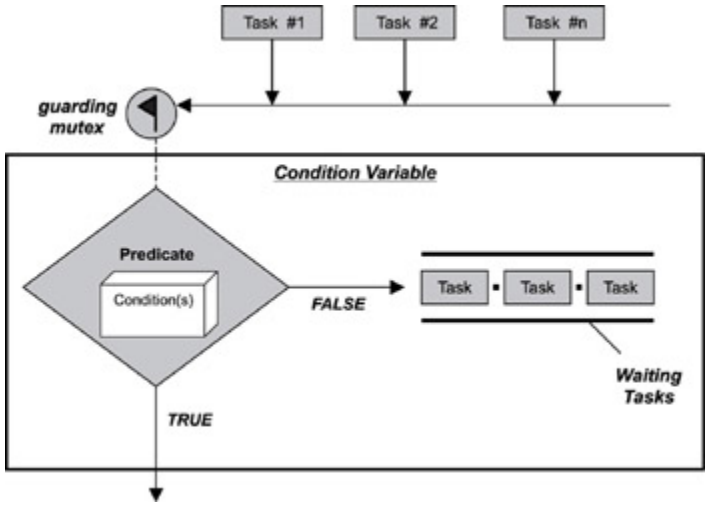


Figure 32: Condition variable.

When a task examines a condition variable, the task must have exclusive access to that condition variable. Without exclusive access, another task could alter the condition variable's conditions at the same time, which could cause the first task to get an erroneous indication of the variable's state.

Therefore, a mutex is always used in conjunction with a condition variable. The mutex ensures that one task has exclusive access to the condition variable until that task is finished with it. For example, if a task acquires the mutex to examine the condition variable, no other task can simultaneously modify the condition variable of the shared resource.

A task must first acquire the mutex before evaluating the predicate. This task must subsequently release the mutex and then, if the predicate evaluates to false, wait for the creation of the desired conditions. Using the condition variable, the kernel guarantees that the task can release the mutex and then block-wait for the condition in one atomic operation, which is the essence of the condition variable. An *atomic operation* is an operation that cannot be interrupted.

Remember, however, that condition variables are not mechanisms for synchronizing access to a shared resource. Rather, most developers use them to allow tasks waiting on a shared resource to reach a desired value or state.

Condition Variable Control Blocks

The kernel maintains a set of information associated with the condition variable when the variable is first created. As stated previously, tasks must block and wait when a condition variable's predicate evaluates to false. These waiting tasks are maintained in the task-waiting list.

The kernel guarantees for each task that the combined operation of releasing the associated mutex and performing a block-wait on the condition will be atomic. After the desired conditions have been created, one of the waiting tasks is awakened and resumes execution.

The criteria for selecting which task to awaken can be priority-based or FIFO-based, but it is kernel-defined. The kernel guarantees that the selected task is removed from the task-waiting list, reacquires the guarding mutex, and resumes its operation in one atomic operation. The essence of the condition variable is the atomicity of the unlock-and-wait and the resume-and-lock operations provided by the kernel. Figure.33 illustrates a condition variable control block.

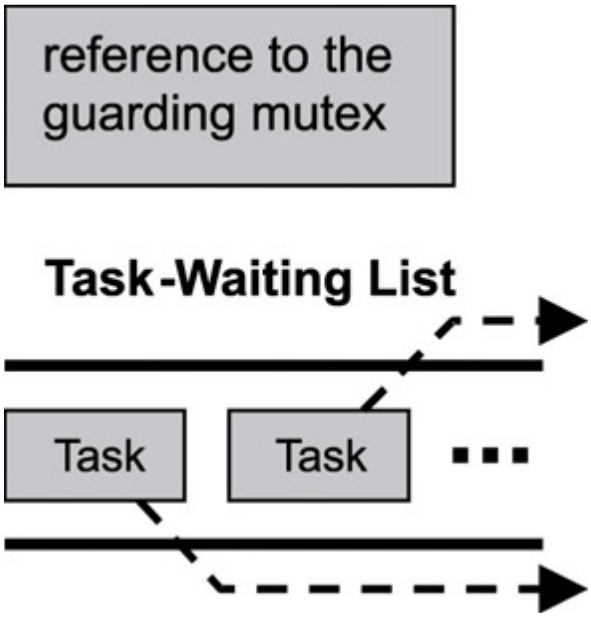


Figure 33: Condition variable control block.

The cooperating tasks define which conditions apply to which shared resources. This information is not part of the condition variable because each task has a different predicate or condition for which the task looks. The condition is specific to the task.

Typical Condition Variable Operations

A set of operations is allowed for a condition variable, as shown in Table 14.

Table 14: Condition variable operations.

Operation	Description
Create	Creates and initializes a condition variable
Wait	Waits on a condition variable
Signal	Signals the condition variable on the presence of a condition
Broadcast	Signals to all waiting tasks the presence of a condition

The create operation creates a condition variable and initializes its internal control block. The wait operation allows a task to block and wait for the desired conditions to occur in the shared resource. To invoke this operation, the task must first successfully acquire the guarding mutex. The wait operation puts the calling task into the task-waiting queue and releases the associated mutex in a single atomic operation.

The signal operation allows a task to modify the condition variable to indicate that a particular condition has been created in the shared resource. To invoke this operation, the signaling task must first successfully acquire the guarding mutex. The signal operation unblocks one of the tasks waiting on the condition variable.

The selection of the task is based on predefined criteria, such as execution priority or system-defined scheduling attributes. At the completion of the signal operation, the kernel reacquires the mutex associated with the condition variable on behalf of the selected task and unblocks the task in one atomic operation.

The broadcast operation wakes up every task on the task-waiting list of the condition variable. One of these tasks is chosen by the kernel and is given the guarding mutex. Every other task is removed from the task-waiting list of the condition variable, and instead, those tasks are put on the task-waiting list of the guarding mutex.

Typical Uses of Condition Variables

Listing.12 illustrates the usage of the wait and the signal operations.

Listing 12: Pseudo code for wait and the signal operations.

```
Task 1
 Lock mutex
 Examine shared resource
 While (shared resource is Busy)
 WAIT (condition variable)
 Mark shared resource as Busy
 Unlock mutex

Task 2
 Lock mutex
 Mark shared resource as Free
 SIGNAL (condition variable)
 Unlock mutex
```

Task 1 on the left locks the guarding mutex as its first step. It then examines the state of the shared resource and finds that the resource is busy. It issues the wait operation to wait for the resource to become available, or free.

The free condition must be created by task 2 on the right after it is done using the resource. To create the free condition, task 2 first locks the mutex; creates the condition by marking the resource as free, and finally, invokes the signal operation, which informs task 1 that the free condition is now present.

A signal on the condition variable is lost when nothing is waiting on it. Therefore, a task should always check for the presence of the desired condition before waiting on it. A task should also always check for the presence of the desired condition after a wakeup as a safeguard against improperly generated signals on the condition variable.

This issue is the reason that the pseudo code includes a while loop to check for the presence of the desired condition. This example is shown in Figure 34.

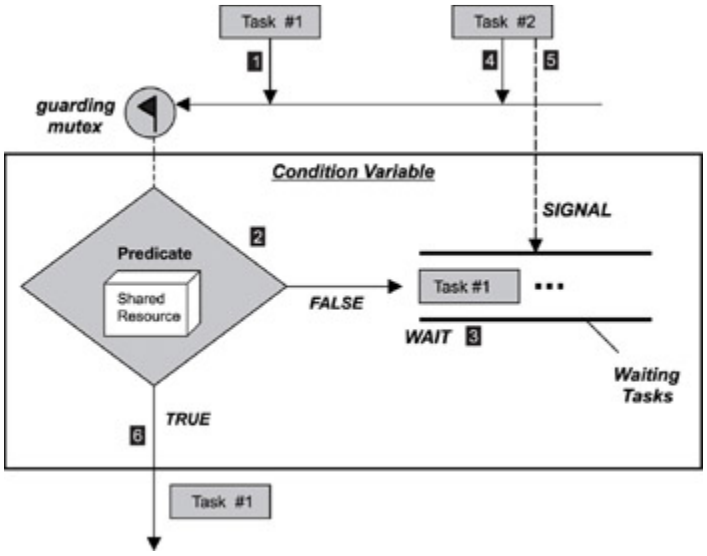


Figure 34: Execution sequence of wait and signal operations.

Implementing Reader-Writer Locks Using Condition Variables

This section presents another example of the usage of condition variables. Consider a shared memory region that both readers and writers can access. The example reader-writer lock design has the following properties: multiple readers can simultaneously read the memory content, but only one writer is allowed to write data into the shared memory at any one time. The writer can begin writing to the shared memory when that memory region is not accessed by a task (readers or writers). Readers precede writers because readers have priority over writers in term of accessing the shared memory region.

The implementation that follows can be adapted to other types of synchronization scenarios when prioritized access to shared resources is desired, as shown in Listings.

The following assumptions are made in the program listings:

1. The mutex_t data type represents a mutex object and condvar_t represents a condition variable object; both are provided by the RTOS.
2. lock_mutex, unlock_mutex, wait_cond, signal_cond, and broadcast_cond are functions provided by the RTOS. Lock_mutex and unlock_mutex operate on the mutex object. wait_cond, signal_cond, and broadcast_cond operate on the condition variable object.

Listing 1. shows the data structure needed to implement the reader-writer lock.

Data structure for implementing reader-writer locks.

```
typedef struct {
    mutex_t guard_mutex;
    condvar_t read_condvar;
    condvar_t write_condvar;
    int rw_count;
    int read_waiting;
}   rwlock_t;
   rw_count == -1 indicates a writer is active
```

Listing 2 shows the code that the writer task invokes to acquire and to release the lock.

Code called by the writer task to acquire and release locks.

```
acquire_write(rwlock_t *rwlock)
{
 lock_mutex(&rwlock->guard_mutex);
 while (rwlock->rw_count != 0)
 wait_cond(&rwlock->write_condvar, &rwlock->guard_mutex);
 rwlock->rw_count = -1;
 unlock_mutex(&rwlock->guard_mutex);
}
release_write(rwlock_t *rwlock)
{
 lock_mutex(&rwlock->guard_mutex);
 rwlock->rw_count = 0;
 if (rwlock->r_waiting)
 broadcast_cond(&rwlock->read_condvar, &rwlock->guard_mutex);
```

```
 else
 signal_cond(&rwlock->write_condvar, &rwlock->guard_mutex);
 unlock_mutex(&rwlock->guard_mutex);
}
```

Listing 3 shows the code that the reader task invokes to acquire and release the lock.

Code called by the reader task to acquire and release locks.

```
acquire_read(rwlock_t *rwlock)
{
 lock_mutex(&rwlock->guard_mutex);
 rwlock->r_waiting++;
 while (rwlock->rw_count < 0)
 wait_cond(&rwlock->read_condvar, &rwlock->guard_mutex);
 rwlock->r_waiting = 0;
 rwlock->rw_count++;
 unlock_mutex(&rwlock->guard_mutex);
}
release_read(rwlock_t *rwlock)
{
 lock_mutex(&rwlock->guard_mutex);
 rwlock->rw_count --;
 if (rwlock->rw_count == 0)
 signal_cond(&rwlock->write_condvar, &rwlock->guard_mutex);
 unlock_mutex(&rwlock->guard_mutex);
}
```

In case broadcast_cond does not exist, use a for loop as follows

```
 for (i = rwlock->read_waiting; i > 0; i--)
 signal_cond(&rwlock->read_condvar, &rwlock->guard_mutex);
```