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.

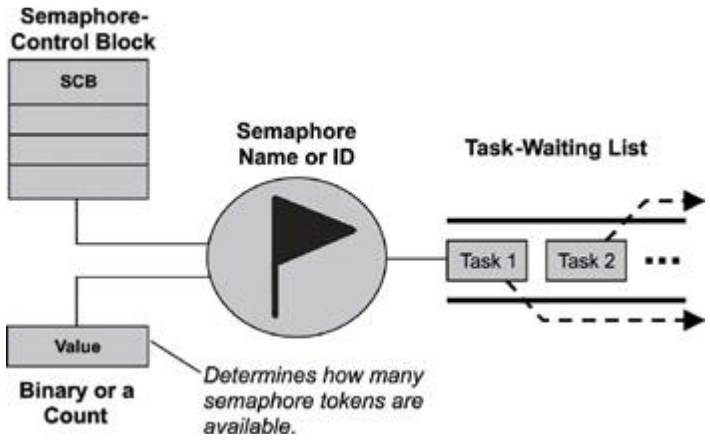


Figure 1: A semaphore, its associated parameters, and supporting data structures

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.

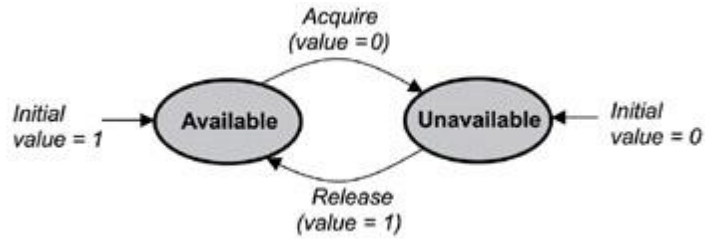


Figure 2: The state diagram of a binary semaphore.

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.

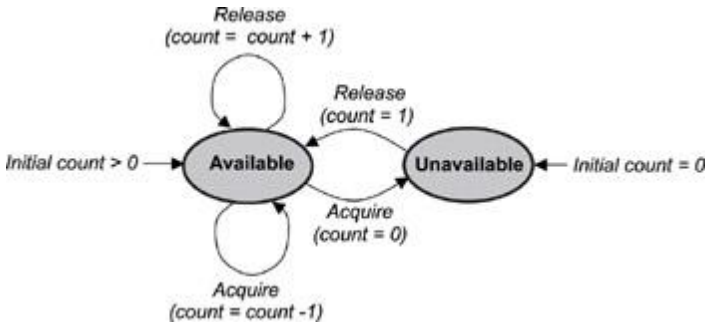


Figure 3: The state diagram of a counting semaphore.

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.

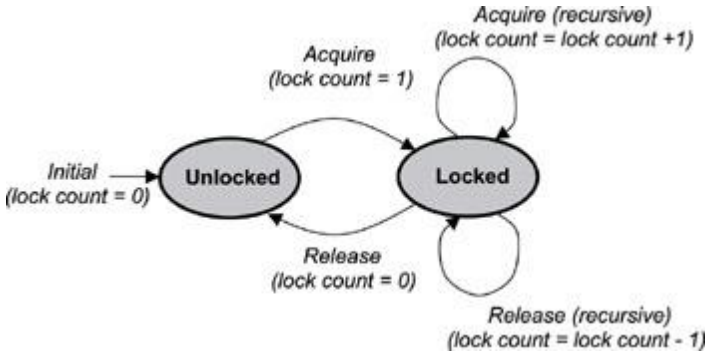


Figure 4: The state diagram of a mutual exclusion (mutex) semaphore.

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.

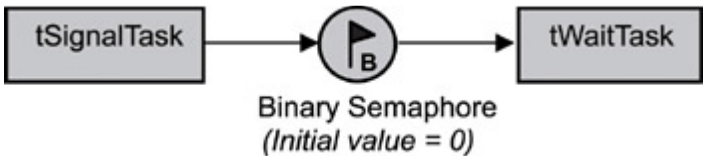


Figure 5: Wait-and-signal synchronization between two tasks.

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.

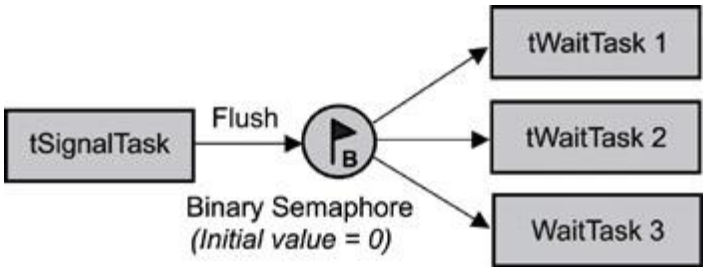


Figure.6: Wait-and-signal synchronization between multiple tasks.

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.

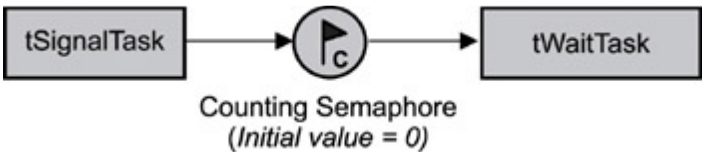


Figure 7: Credit-tracking synchronization between two tasks.

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.

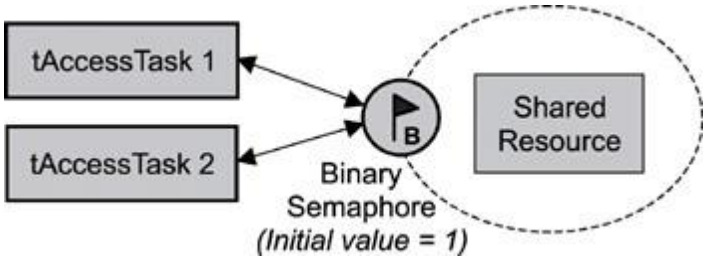


Figure 8: Single shared-resource-access synchronization.

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.

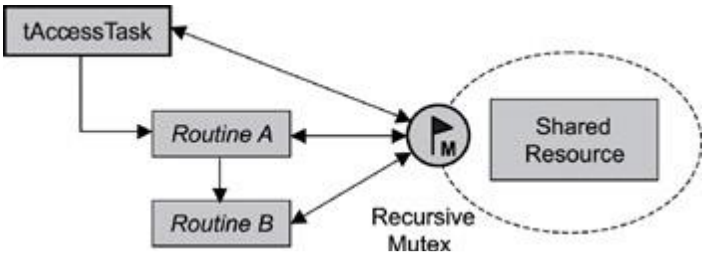


Figure 9: Recursive shared- resource-access synchronization.

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.

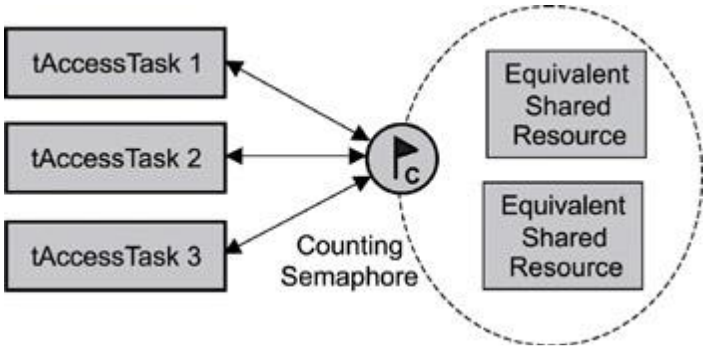


Figure 10: Multiple shared-resource-access synchronization.

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.

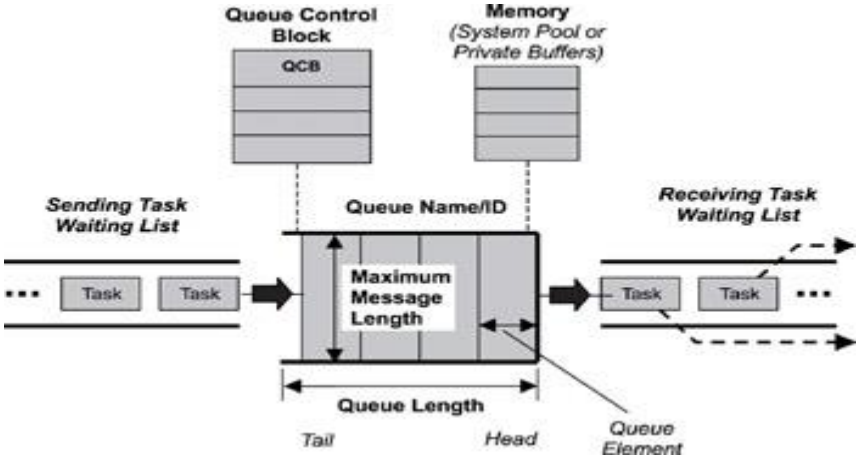


Figure 11: A message queue, its associated parameters, and supporting data structures.

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.

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.

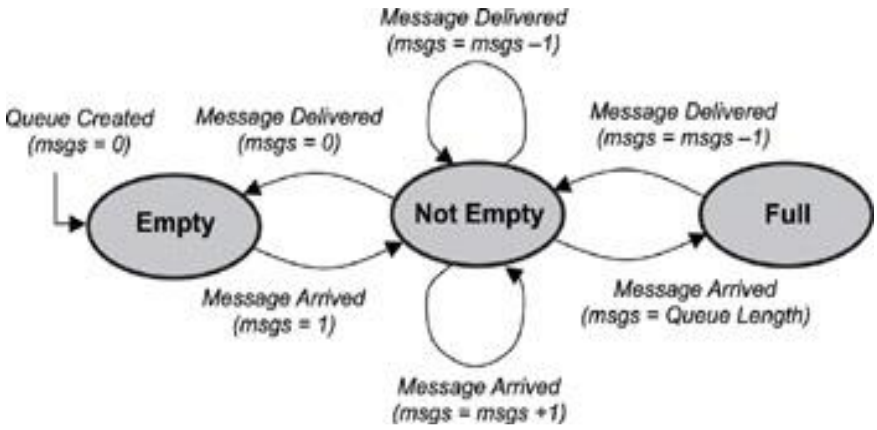


Figure 12: The state diagram for a message queue.

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.

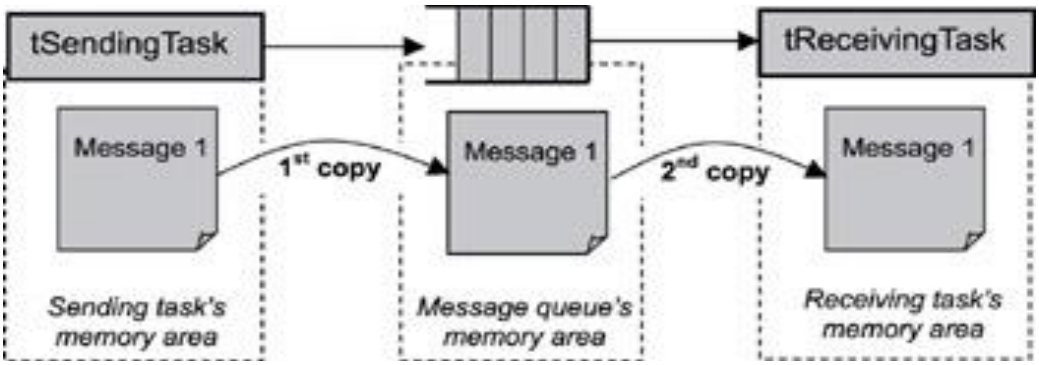


Figure 13: Message copying and memory use for sending and receiving messages.

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.

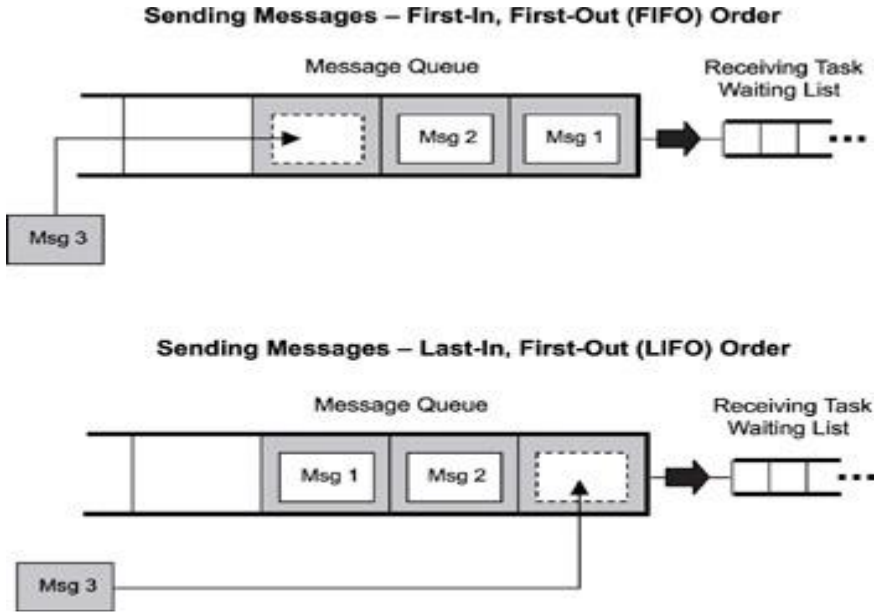


Figure 7.4: Sending messages in FIFO or LIFO order.

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.

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.

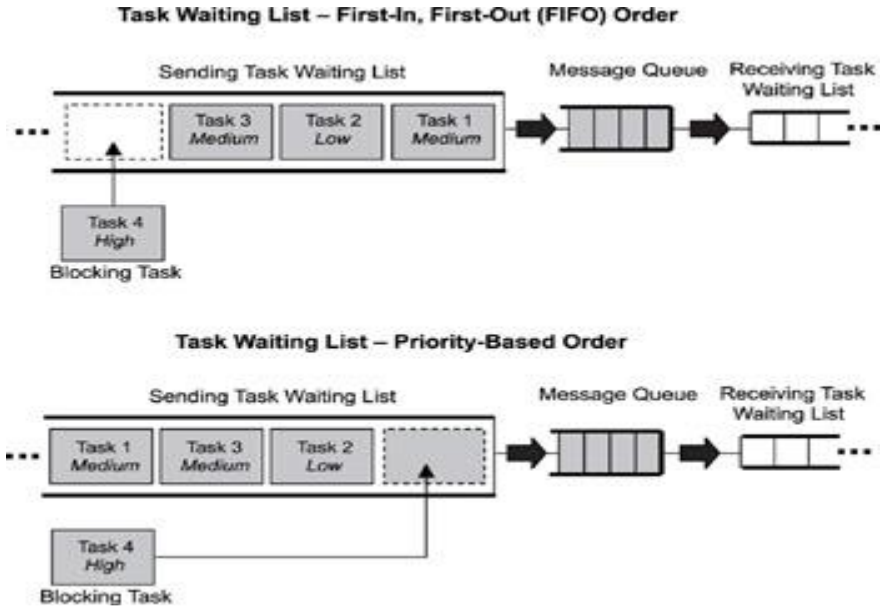


Figure 15: FIFO and priority-based task-waiting lists.

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.

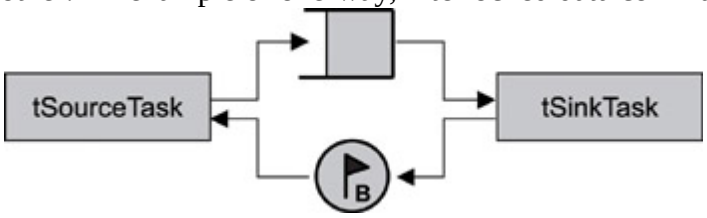


Figure 17: Interlocked, one-way data communication.

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.

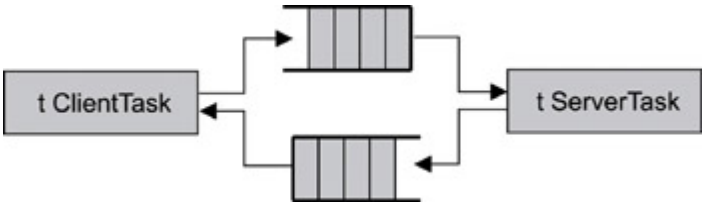


Figure 18: Interlocked, two-way data communication.

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.

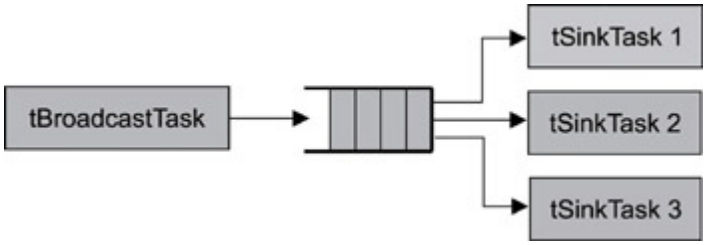


Figure 19: Broadcasting messages.

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.

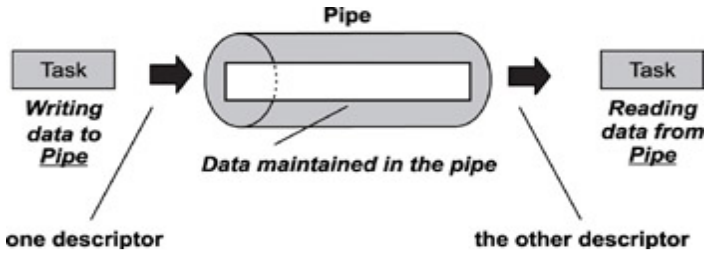


Figure 20: A common pipe unidirectional.

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.

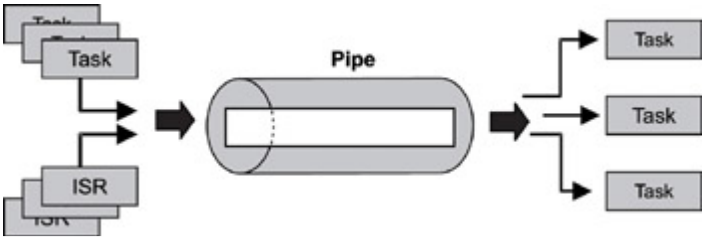


Figure 21: Common pipe operation.

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.

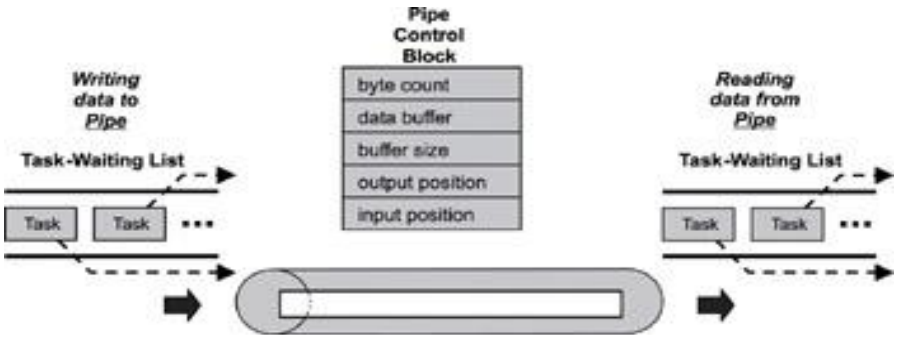


Figure 22: Pipe control block.

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.

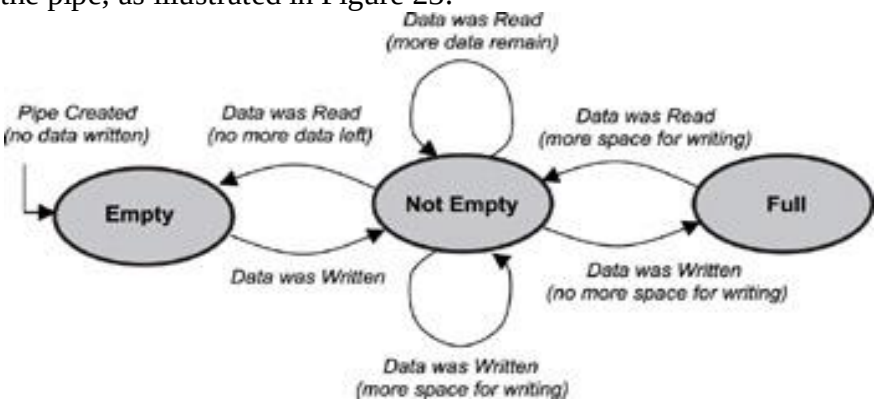


Figure 23: States of a pipe.

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.

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.

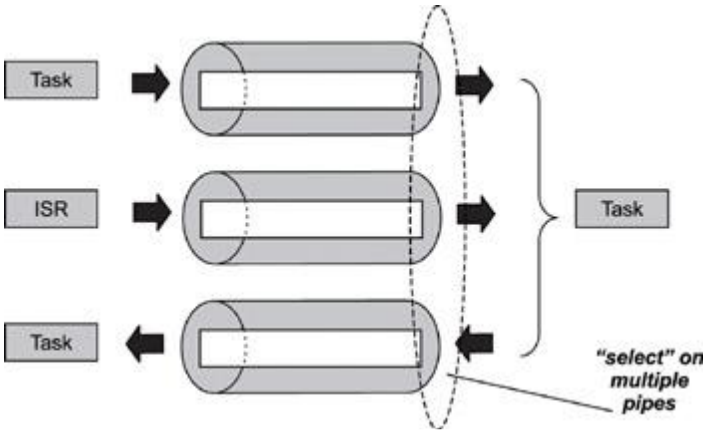


Figure 24: The select operation on multiple pipes.

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.

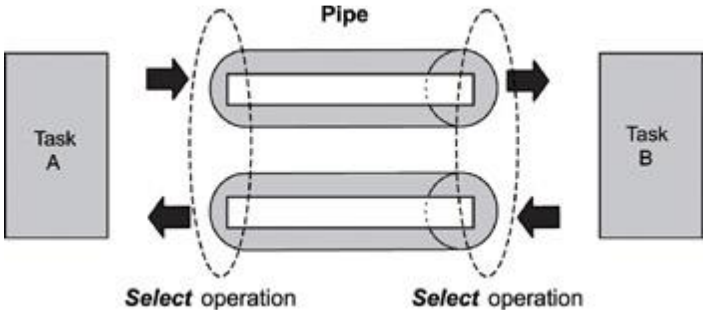


Figure 25: Using pipes for inter-task synchronization.

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.

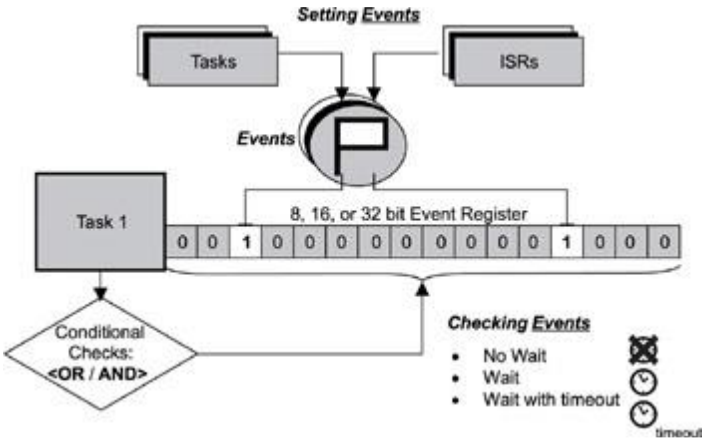


Figure 26: Event register.

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.

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.

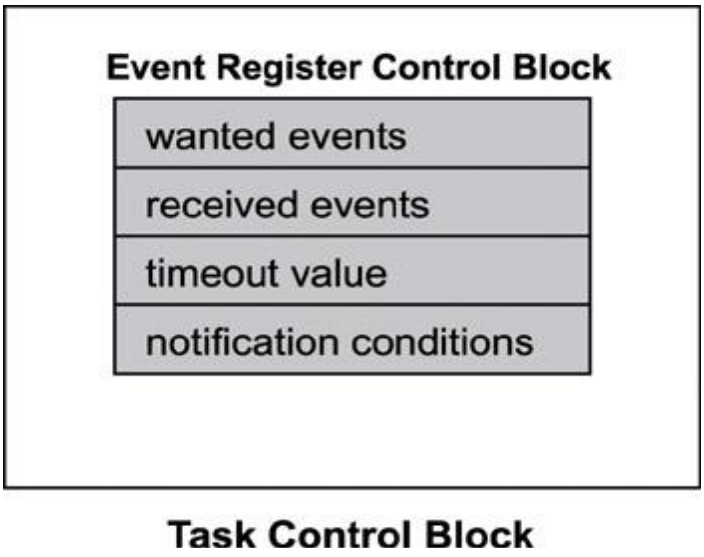


Figure 27: Event register control block.

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.

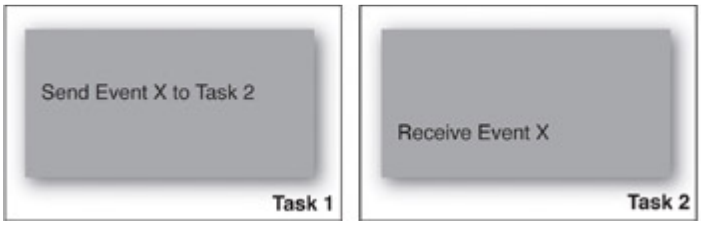
The send operation allows an external source, either a task or an ISR, to send events to another task. The sender can send multiple events to the designated task through a single send operation. Events that have been sent and are pending on the event bits but have not been chosen for reception by the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

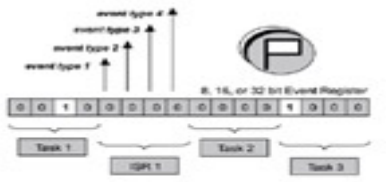


Figure 28: Identifying an event source.

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.

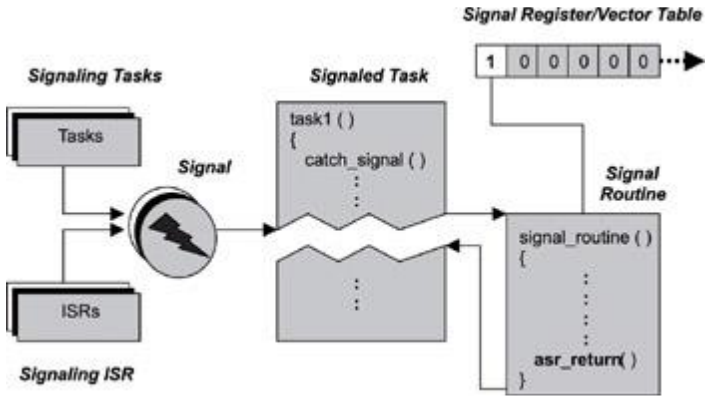


Figure 29: Signals.

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.

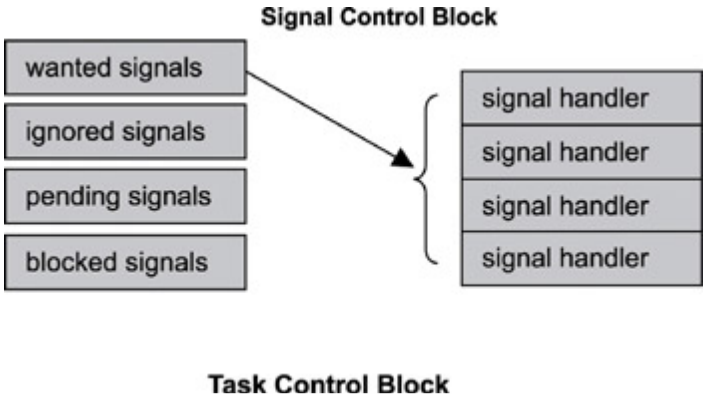


Figure 30: Signal control block.

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.

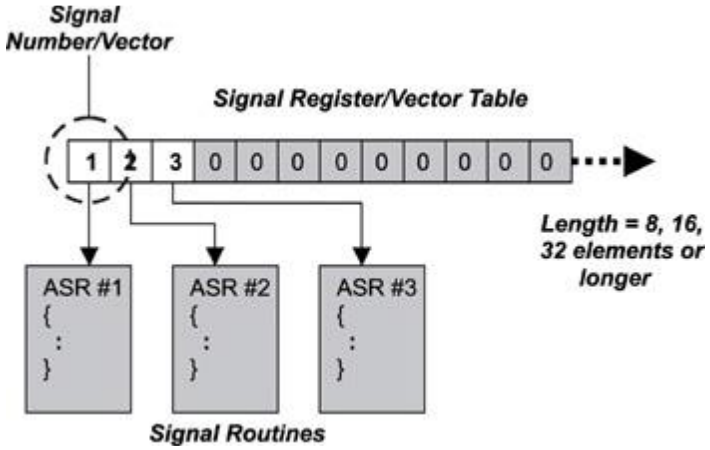


Figure 31: The catch operation.

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, this 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.

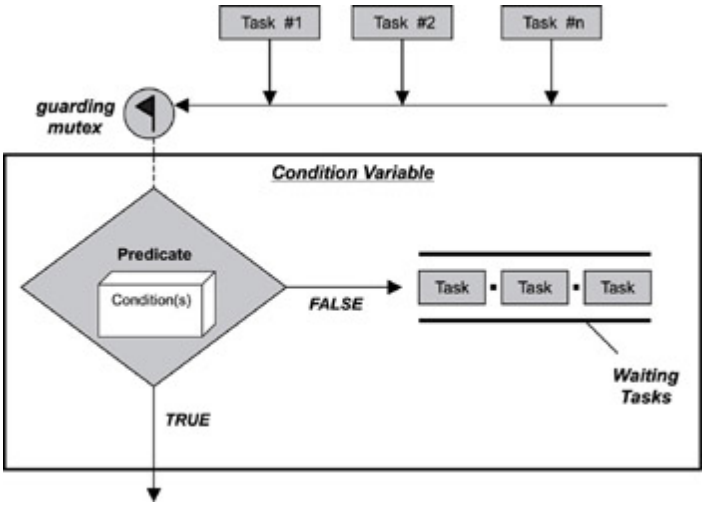


Figure 32: Condition variable.

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.

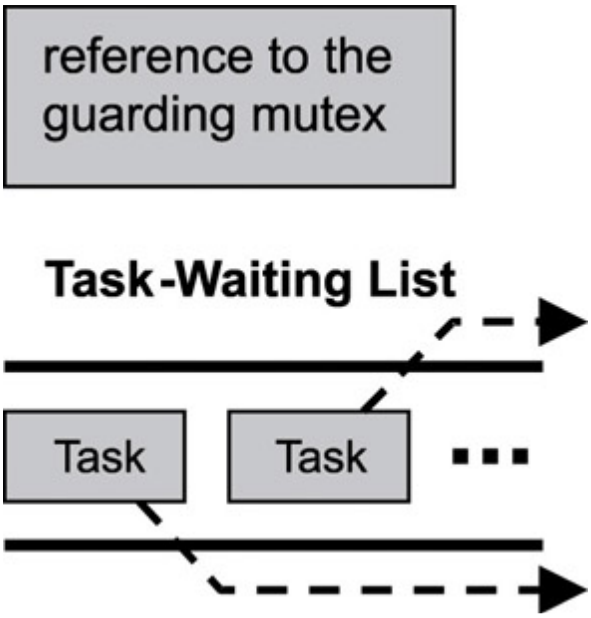


Figure 33: Condition variable control block.

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.

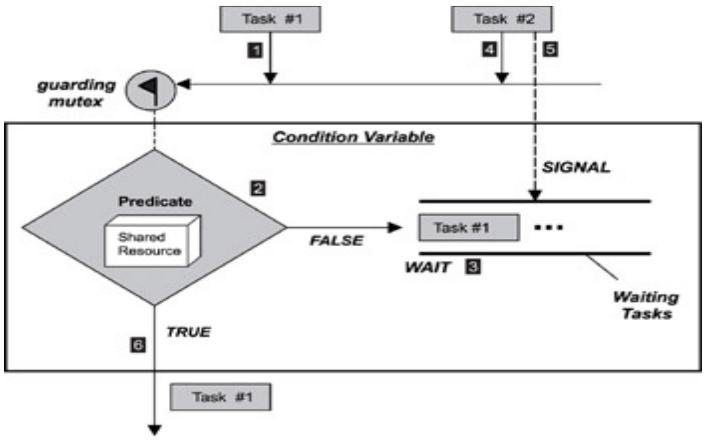


Figure 34: Execution sequence of wait and signal operations.

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);
```

Introduction:

The OS is a set of software libraries that serves two main purposes in an embedded system: providing an abstraction layer for software on top of the OS to be less dependent on hardware, making the development of middleware and applications that sit on top of the OS easier, and managing the various system hardware and software resources to ensure the entire system operates efficiently and reliably. While embedded OSs varies in what components they possess, all OSs have a kernel at the very least.

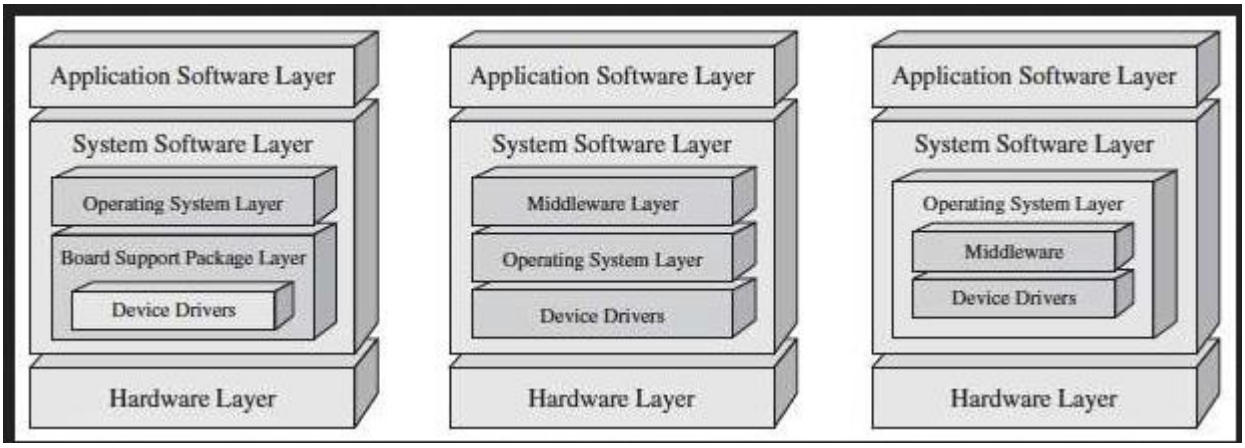


Figure 1. OSs and the Embedded Systems Model.

The kernel is a component that contains the main functionality of the OS, specifically all or some combination of features and their interdependencies, shown in Figures 2a–b, including:

- **Process Management**: how the OS manages and views other software in the embedded system (via processes— Multitasking and Process Management). A sub function typically found within process management is interrupt and error detection management. The multiple interrupts and/or traps generated by the various processes need to be managed efficiently, so that they are handled correctly and the processes that triggered them are properly tracked.
- **Memory Management**: the embedded system's memory space is shared by all the different processes, so that access and allocation of portions of the memory space need to be managed. Within memory management, other sub functions such as security system management allow for portions of the embedded system sensitive to disruptions that can result in the disabling of the system, to remain secure from unfriendly, or badly written, higher-layer software.
- **I/O System Management**: I/O devices also need to be shared among the various processes and so, just as with memory, access and allocation of an I/O device need to be managed .Through I/O system management, file system management can also be provided as a method of storing and managing data in the forms of files.

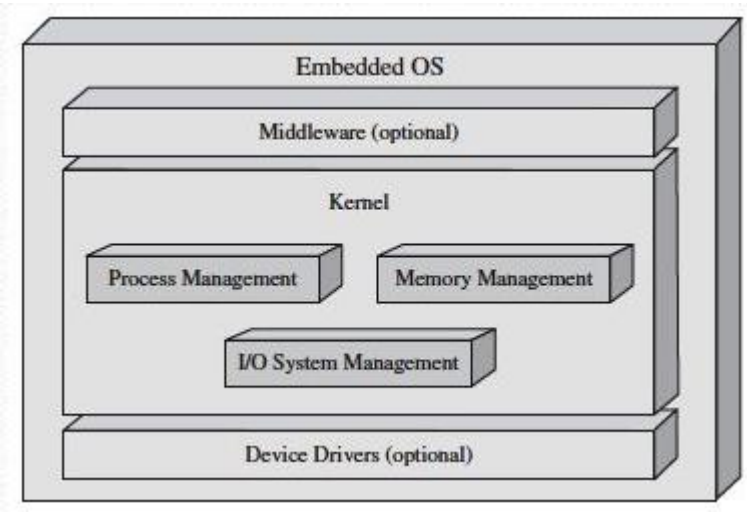


Figure 2a. General OS model.

Because of the way in which an OS manages the software in a system, using processes, the process management component is the most central subsystem in an OS. All other OS subsystems depend on the process management unit.

Since all code must be loaded into main memory (random access memory (RAM) or cache) for the master CPU to execute, with boot code and data located in non-volatile memory (read-only memory (ROM), Flash, etc.), the process management subsystem is equally dependent on the memory management subsystem.

I/O management, for example, could include networking I/O to interface with the memory manager in the case of a network file system (NFS).

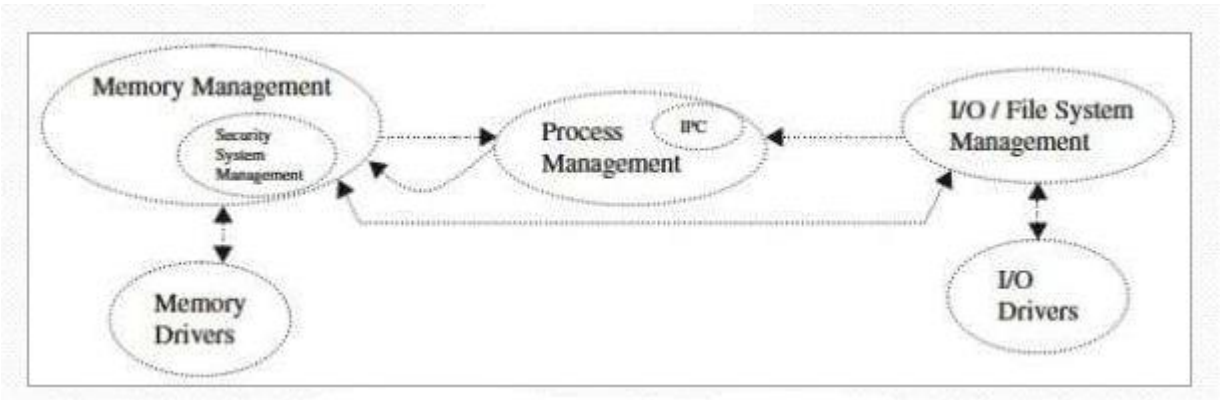


Figure 2b. Kernel subsystem dependencies.

Outside the kernel, the Memory Management and I/O Management subsystems then rely on the device drivers, and vice-versa, to access the hardware.

Difference between THREAD, PROCESS and TASK

A program in execution is known as '**process**'. A program can have any number of

processes. Every process has its own address space.

Threads uses address spaces of the process. The difference between a thread and a process is, when the CPU switches from one process to another the current information needs to be saved in Process Descriptor and load the information of a new process. Switching from one thread to another is simple.

A **task** is simply a set of instructions loaded into the memory. Threads can themselves split themselves into two or more simultaneously running tasks.

Process Management

A ***process*** (commonly referred to as a ***task*** in many embedded OSes) is created by an OS to encapsulate all the information that is involved in the executing of a program (i.e., stack, PC, the source code and data, etc.). This means that a program is only part of a task.

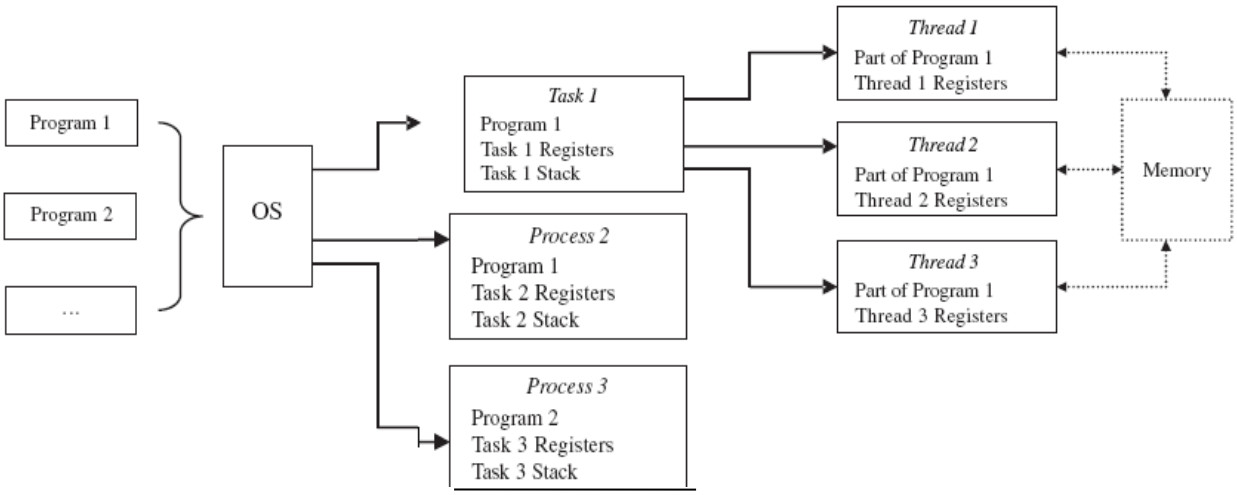
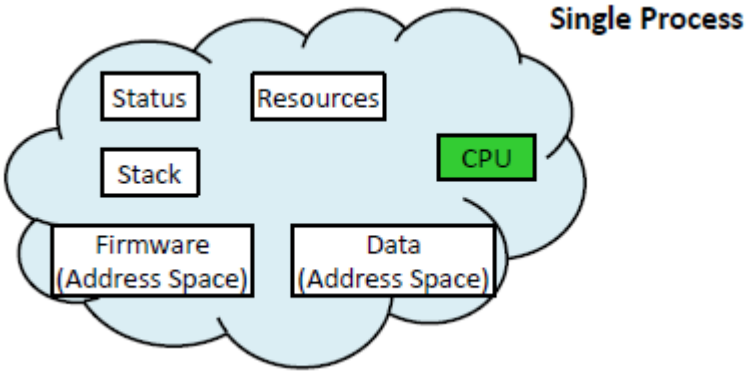


Fig 3 .Tasks, Processes and threads

- When a process is created, it is allocated a number f resources by the OS, which may include:
  - –Process stack
  - –Memory address space
  - –Registers (through the CPU)
  - –A program counter (PC)
  - –I/O ports, network connections, file descriptors, etc.
- These resources are generally not shared with other processes



Multiple Processes

- If another process is added to the system, potential resource contention problems arise
- This is resolved by carefully managing how the resources are allocated to each process and by controlling how long each can retain the resources
- The main resource, CPU, is given to processes in a time multiplexed fashion (i.e., time sharing); when done fast enough, it will appear as if both processes are using it at the same time
- The execution time of the program will be extended, but operation will give the *appearance* of simultaneous execution. Such a scheme is called **multitasking .**

Note: Multiple processes can be executed in RTOS using Context switching.

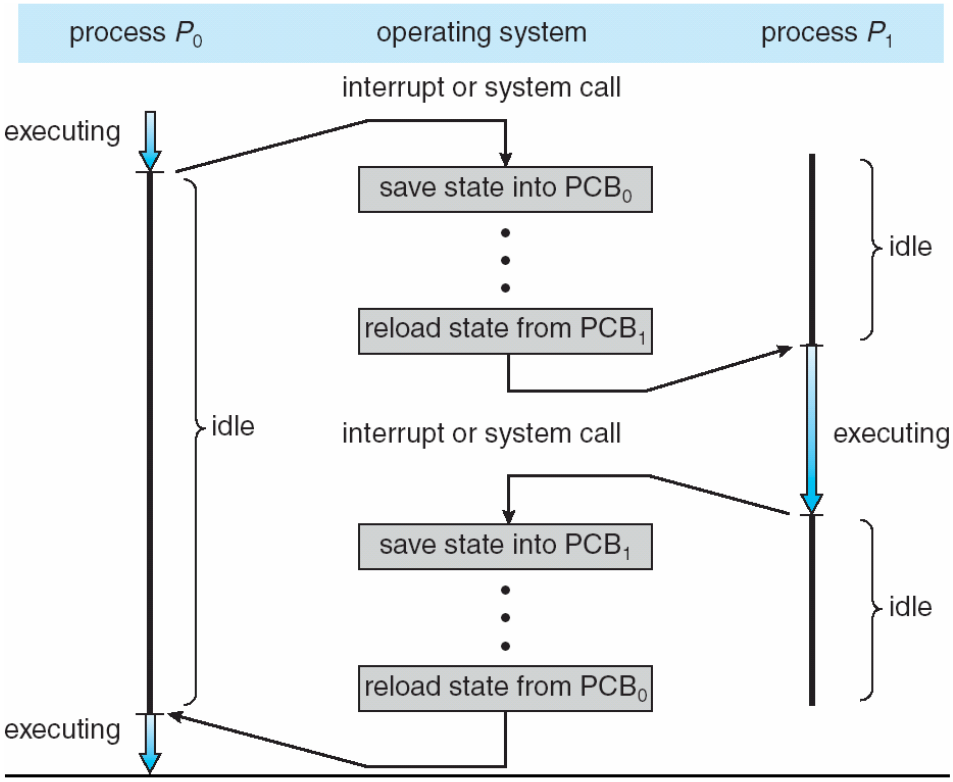


Fig. Context switching in RTOS

Attributes or Characteristics of a Process

A process has following Attributes.

. **Process Id:** A unique identifier assigned by operating system

. **Process State:** Can be ready, running, etc

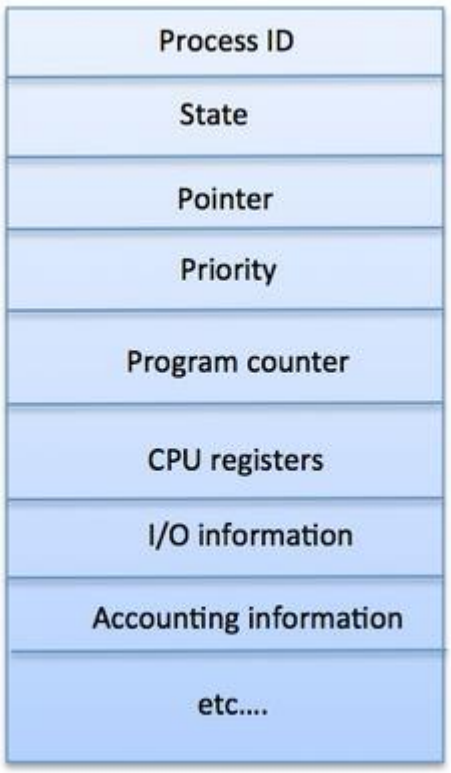
. **CPU registers:** Like Program Counter (CPU registers must be saved and restored when a process is swapped out and in of CPU)

. **I/O status information:** For example devices allocated to process, open files, etc

. **CPU scheduling information:** For example Priority (Different processes may have different priorities)

All the above attributes of a process are also known as ***Context of the process***. Every Process has its known Program control Block (PCB) i.e each process will have a unique PCB. All the Above Attributes are the part of the PCB.

The architecture of a PCB is completely dependent on Operating System and may contain different information in different operating systems. Here is a simplified diagram of a PCB −



States of Process:

A process is in one of the following states

1. New: Newly Created Process (or) being created process.

2. Ready: After creation Process moves to Ready state, i.e., process is ready for execution.

3. Run: Currently running process in CPU (only one process at a time can be under execution in a single processor).

4. Wait (or Block): When process request for I/O request.

5. Complete (or terminated): Process Completed its execution.

6. Suspended Ready: When ready queue becomes full, some processes are moved to suspend ready state

7. Suspended Block: When waiting queue becomes full.

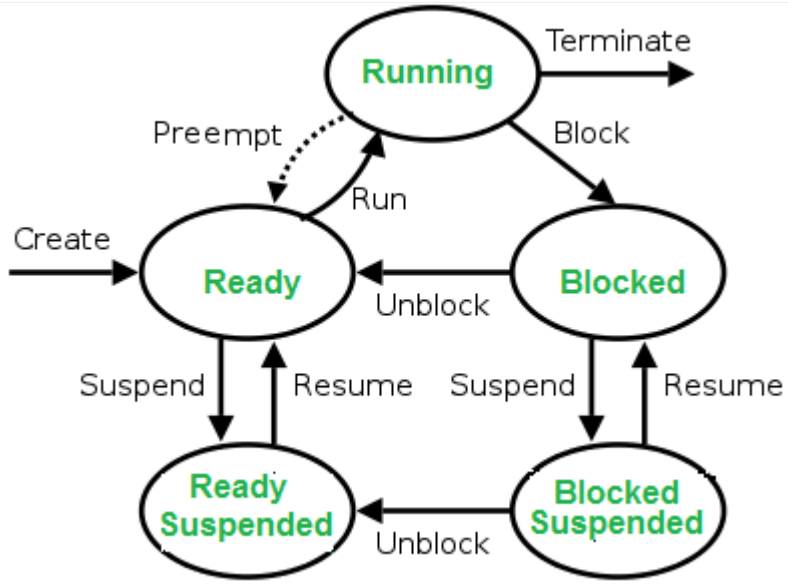


Fig.4 States of a process

Process Creation

- Processes need to be created
  - _ Processes are created by other processes
  - _ System call **create_process**

- Parent process create children processes
  - _ which, in turn create other processes, forming a tree of processes.
- Resource sharing
  - _ Parent and children share all resources.
  - _ Children share subset of parent's resources.
  - _ Parent and child share no resources.

- Execution
  - _ Parent and children execute concurrently.
  - _ **Parent waits until children terminate.**
- e.g. on Unix: ***fork()*** system call creates a new process
- NT/2K/XP: ***CreateProcess()*** syscall includes name of program to be executed.

Process Termination

- Process terminates when executing the last statement
  - _ the last statement is usually **exit**
  - --Process resources are deallocated by operating system.
- Parent may terminate execution of children processes **(abort).**
  - _ Child has exceeded allocated resources.
  - _ Task assigned to child is no longer required.
  - _ Parent is exiting.
    - Operating system does not allow child to continue if its parent terminates.
    - Cascading termination.
- e.g. Unix has ***wait(), exit() and kill()***
- e.g. NT/2K/XP has ***ExitProcess()*** for self termination and ***TerminateProcess()*** for killing others.

CPU Scheduler

Selects from among the ready processes and allocates the CPU to one of them.CPU scheduling decisions may take place in different situations

- ***Non-preemptive scheduling***
  - _ The running process terminates
  - _ The running process performs an I/O operation or waits for an event
- ***Preemptive scheduling***
  - _ The running process has exhausted its time slice
  - _ A process A transits from blocked to ready and is considered more important than process B that is currently running

The fork () System Call

System call **fork()** is used to create processes. It takes no arguments and returns a process ID. The purpose of **fork ()** is to create a ***new*** process, which becomes the *child* process of the caller. After a new child process is created, ***both*** processes will execute the next instruction following the ***fork ()*** system call. Therefore, we have to distinguish the parent from the child. This can be done by testing the returned value of **fork ()**:

- If **fork ()** returns a negative value, the creation of a child process was unsuccessful.
- **fork ()** returns a zero to the newly created child process.
- **fork ()** returns a positive value, the ***process ID*** of the child process, to the parent. The returned process ID is of type **pid_t** defined in **sys/types.h**. Normally, the process ID is an integer. Moreover, a process can use function **getpid()** to retrieve the process ID assigned to this process.

Therefore, after the system call to **fork ()**, a simple test can tell which process is the child. **Please note that Unix will make an exact copy of the parent's address space and give it to the child. Therefore, the parent and child processes have separate address spaces**.

Let us take an example to make the above points clear. This example does not distinguish parent and the child processes.

```
#include  <stdio.h>
#include  <string.h>
#include  <sys/types.h>

#define   MAX_COUNT  200
#define   BUF_SIZE   100

void  main(void)
{
     pid_t  pid;
     int    i;
     char   buf[BUF_SIZE];

     fork();
     pid = getpid();
     for (i = 1; i <= MAX_COUNT; i++) {
          sprintf(buf, "This line is from pid %d, value = %d\n", pid, i);
          write(1, buf, strlen(buf));
     }
}
```

Memory Management

Introduction:

Knowing the capability of the memory management system can aid application design and help avoid pitfalls. For example, in many existing embedded applications, the dynamic memory allocation routine, malloc, is called often. It can create an undesirable side effect called memory fragmentation. This generic memory allocation routine, depending on its implementation, might impact an application's performance. In addition, it might not support the allocation behavior required by the application.

Many embedded devices (such as PDAs, cell phones, and digital cameras) have a limited number of applications (tasks) that can run in parallel at any given time, but these devices have small amounts of physical memory onboard. Larger embedded devices (such as network routers and web servers) have more physical memory installed, but these embedded systems also tend to operate in a more dynamic environment, therefore making more demands on memory.

Regardless of the type of embedded system, the common requirements placed on a memory management system are minimal fragmentation, minimal management overhead, and deterministic allocation time.

Dynamic Memory Allocation in Embedded Systems

Program code, program data, and system stack occupy the physical memory after program initialization completes. Either the RTOS or the kernel typically uses the remaining physical memory for dynamic memory allocation. This memory area is called the *heap*.

In general, a memory management facility maintains internal information for a heap in a reserved memory area called the *control block*.

Typical internal information includes:

- the starting address of the physical memory block used for dynamic memory allocation,
- the overall size of this physical memory block, and
- the allocation table that indicates which memory areas are in use, which memory areas are free, and the size of each free region.

Memory Fragmentation and Compaction

In the example implementation, the heap is broken into small, fixed-size blocks. Each block has a unit size that is power of two to ease translating a requested size into the corresponding required number of units.

In this example, the unit size is 32 bytes. The dynamic memory allocation function, ***malloc***, has an input parameter that specifies the size of the allocation request in bytes. malloc allocates a larger block, which is made up of one or more of the smaller, fixed-size blocks. The size of this larger memory block is at least as large as the requested size; it is the closest to the multiple of the unit size.

For example, if the allocation requests 100 bytes, the returned block has a size of 128 bytes (4 units x 32 bytes/unit). As a result, the requestor does not use 28 bytes of the allocated memory, which is called ***memory fragmentation***. This specific form of fragmentation is called internal fragmentation because it is internal to the allocated block.

The allocation table can be represented as a bitmap, in which each bit represents a 32-byte unit. Figure 5 shows the states of the allocation table after a series of invocations of the malloc and free functions. In this example, the heap is 256 bytes.

Step 6 shows two free blocks of 32 bytes each. Step 7, instead of maintaining three separate free blocks, shows that all three blocks are combined to form a 128-byte block. Because these blocks have been combined, a future allocation request for 96 bytes should succeed.

Figure 6 shows another example of the state of an allocation table. Note that two free 32-byte blocks are shown. One block is at address 0x10080, and the other at address 0x101C0, which cannot be used for any memory allocation requests larger than 32 bytes. Because these isolated blocks do not contribute to the contiguous free space needed for a large allocation request, their existence makes it more likely that a large request will fail or take too long. The existence of these two trapped blocks is considered ***external fragmentation*** because the fragmentation exists in the table, not within the blocks themselves.

One way to eliminate this type of fragmentation is to compact the area adjacent to these two blocks. The range of memory content from address 0x100A0 (immediately following the first

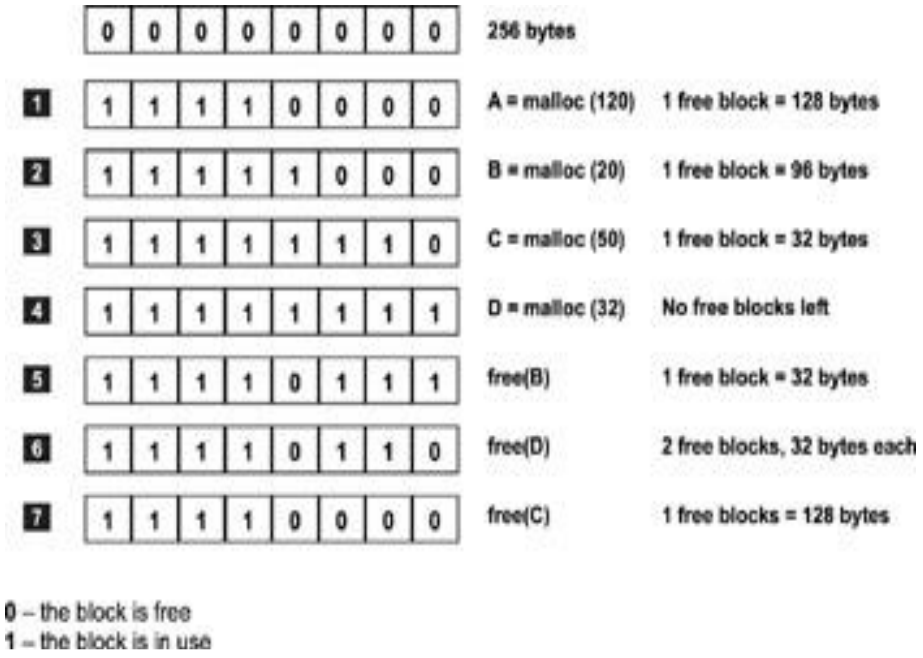


Figure 5: States of a memory allocation map.

free block) to address 0x101BF (immediately preceding the second free block is shifted 32 bytes lower in memory, to the new range of 0x10080 to 0x1019F, which effectively combines the two free blocks into one 64-byte block. This new free block is still considered memory fragmentation if future allocations are potentially larger than 64 bytes. Therefore, memory compaction continues until all of the free blocks are combined into one large chunk.

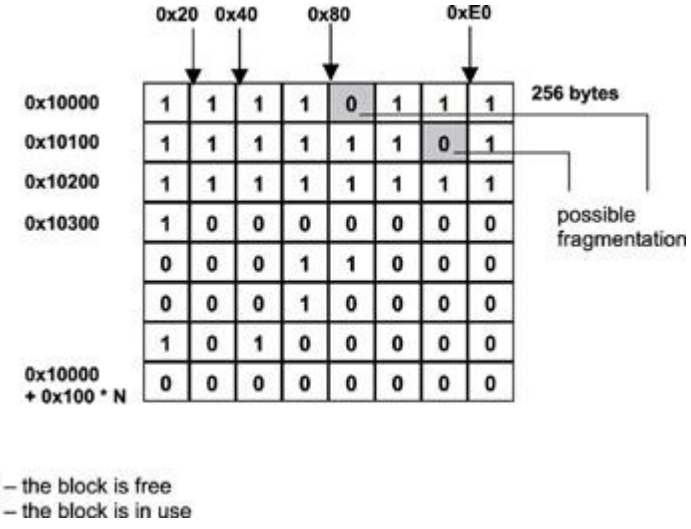


Figure 6: Memory allocation map with possible fragmentation.

Several problems occur with memory compaction. It is time-consuming to transfer memory content from one location to another. The cost of the copy operation depends on the length of the contiguous blocks in use. The tasks that currently hold ownership of those memory blocks are prevented from accessing the contents of those memory locations until the transfer operation completes. Memory compaction is almost never done in practice in embedded designs. The free memory blocks are combined only if they are immediate neighbors, as illustrated in Figure 5.

Memory compaction is allowed if the tasks that own those memory blocks reference the blocks using virtual addresses. Memory compaction is not permitted if tasks hold physical addresses to the allocated memory blocks. In many cases, memory management systems should also be concerned with architecture-specific memory alignment requirements.

Memory alignment refers to architecture-specific constraints imposed on the address of a data item in memory. Many embedded processor architectures cannot access multi-byte data items at any address. For example, some architecture requires multi-byte data items, such as integers and long integers, to be allocated at addresses that are a power of two. Unaligned memory addresses result in bus errors and are the source of memory access exceptions.

Some conclusions can be drawn from this example. An efficient memory manager needs to perform the following chores quickly:

- Determine if a free block that is large enough exists to satisfy the allocation request. This work is part of the malloc operation.
- Update the internal management information. This work is part of both the malloc and free operations.
- Determine if the just-freed block can be combined with its neighboring free blocks to form a larger piece. This work is part of the free operation.

The structure of the allocation table is the key to efficient memory management because the structure determines how the operations listed earlier must be implemented. The allocation table is part of the overhead because it occupies memory space that is excluded from application use. Consequently, one other requirement is to minimize the management overhead.

An Example of malloc and free

The following is an example implementation of malloc's allocation algorithm for an embedded system. A static array of integers, called the *allocation array*, is used to implement the allocation map. The main purpose of the allocation array is to decide if neighboring free blocks can be merged to form a larger free block. Each entry in this array represents a corresponding fixed-size block of memory.

In this sense, this array is similar to the map shown in Figure 6, but this one uses a different encoding scheme. The number of entries contained in the array is the number of fixed-size blocks available in the managed memory area. For example, 1MB of memory can be divided into 32,768 32-byte blocks. Therefore, in this case, the array has 32,768 entries.

To simplify the example for better understanding of the algorithms involved, just 12 units of memory are used. Figure 7 shows the example allocation array.

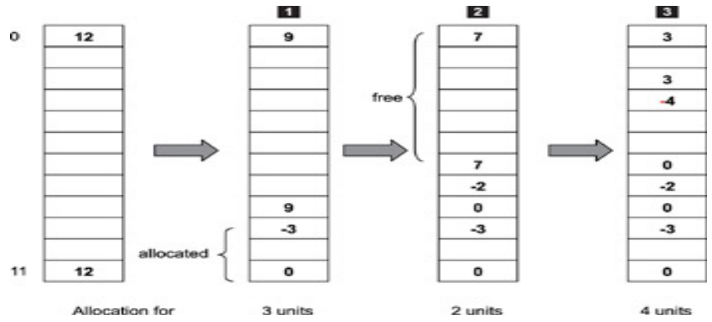


Figure 7: Static array implementation of the allocation map.

In Figure 7, let the allocation-array index start at 0. Before any memory has been allocated, one large free block is present, which consists of all 12 units of available memory. The allocation array uses a simple encoding scheme to keep track of allocated and free blocks of memory. To indicate a range of contiguous free blocks, a positive number is placed in the first and last entry representing the range. This number is equal to the number of free blocks in the range. For example, in the first array shown on the left, the number of free units (12 in this case) is placed in the entries at index 0 and index 11.

Placing a negative number in the first entry and a zero in the last entry indicates a range of allocated blocks. **The number placed in the first entry is equal to -1 times the number of allocated blocks.**

In this example, the first allocation request is for three units. The array labeled 1 in Figure 7 represents the state of the allocation array after this first allocation request is made. The value of -3 at index 9 and the value of 0 at index 11 mark the range of the allocated block. The size of the free block is now reduced to nine. Step 3 in Figure 7 shows the state of the allocation array at the completion of three allocation requests. This array arrangement and the marking of allocated blocks simplify the merging operation that takes place during the free operation.

Not only does this allocation array indicate which blocks are free, but it also implicitly indicates the starting address of each block, because a simple relationship exists between array indices and starting addresses, as shown

```
starting address = offset + unit_size*index
```

When allocating a block of memory, malloc uses this formula to calculate the starting address of the block. For example, in Figure 3, the first allocation for three units begins at index 9. If the offset in the formula is 0x10000 and the unit size is 0x20 (32 decimal), the address returned for index 9 is

```
0x10000 + 0x20*9 = 0x10120
```

Finding Free Blocks Quickly

In this memory management scheme, malloc always allocates from the largest available range of free blocks. The allocation array described is not arranged to help malloc perform this task quickly. The entries representing free ranges are not sorted by size. Finding the largest range always entails an end-to-end search. For this reason, a second data structure is used to speed up the search for the free block that can satisfy the allocation request. The sizes of free blocks within the allocation array are maintained using the heap data structure, as shown in Figure 8.
The heap data structure is a complete binary tree with one property: the value contained at a node is no smaller than the value in any of its child nodes.

The size of each free block is the key used for arranging the heap. Therefore, the largest free block is always at the top of the heap. The malloc algorithm carves the allocation out of the largest available free block. The remaining portion is reinserted into the heap. The heap is rearranged as the last step of the memory allocation process.

Although the size of each free range is the key that organizes the heap, each node in the heap is actually a data structure containing at least two pieces of information: the size of a free range and its starting index in the allocation array.

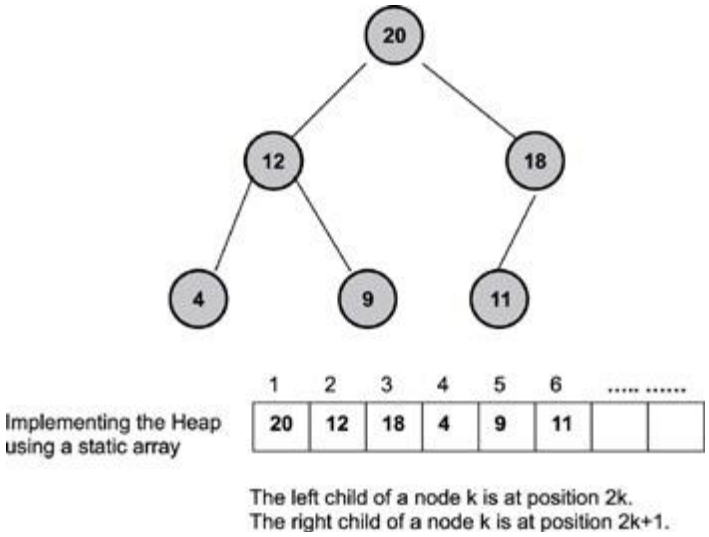


Figure 8: Free blocks in a heap arrangement.

The malloc operation involves the following steps:

1. Examine the heap to determine if a free block that is large enough for the allocation request exists.
2. If no such block exists, return an error to the caller.
3. Retrieve the starting allocation-array index of the free range from the top of the heap.
4. Update the allocation array by marking the newly allocated block, as illustrated in Fig. 7.
5. If the entire block is used to satisfy the allocation, update the heap by deleting the largest node. Otherwise update the size.
6. Rearrange the heap array.

Before any memory has been allocated, the heap has just one node, signifying that the entire memory region is available as one, large, free block. The heap continues to have a single node either if memory is allocated consecutively without any free operations or if each memory free operation results in the deallocated block merging with its immediate neighbors. The heap structure in Figure 8 represents free blocks interleaved with blocks in use and is similar to the memory map in Figure 6.

The heap can be implemented using another static array, called the *heap array*, as shown in Figure 4. The array index begins at 1 instead of 0 to simplify coding in C. In this example, six free blocks of 20, 18, 12, 11, 9, and 4 blocks are available. The next memory allocation uses the 20-block range regardless of the size of the allocation request.

Note that the heap array is a compact way to implement a binary tree. The heap array stores no pointers to child nodes; instead, child-parent relationships are indicated by the positions of the nodes within the array.

The free Operation

Note that the bottom layer of the malloc and free implementation is shown in Figure 7 and Figure 8. In other words, another layer of software tracks, for example, the address of an allocated

block and its size. Let's assume that this software layer exists and that the example is not concerned with it other than that this layer feeds the necessary information into the free function.

The main operation of the free function is to determine if the block being freed can be merged with its neighbors. The merging rules are

1. If the starting index of the block is not 0, check for the value of the array at (index -1). If the value is positive (not a negative value or 0), this neighbor can be merged.
2. If (index + number of blocks) does not exceed the maximum array index value, check for the value of the array at (index + number of blocks). If the value is positive, this neighbor can be merged.

These rules are illustrated best through an example, as shown in Figure 9.

Figure 9 shows two scenarios worth discussion. In the first scenario, the block starting at index 3 is being freed. Following rule #1, look at the value at index 2. The value is 3; therefore, the neighboring block can be merged. The value of 3 indicates that the neighboring block is 3 units large. The block being freed is 4 units large, so following rule #2, look at the value at index 7. The value is -2; therefore, the neighboring block is still in use and cannot be merged.

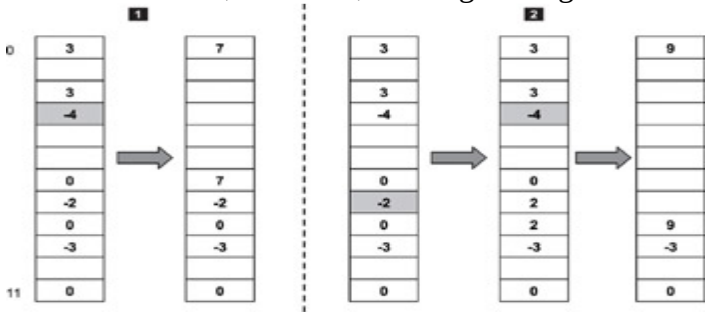


Figure 9: The free operation.

The result of the free operation in the first scenario is shown as the second table in Figure 9. In the second scenario, the block at index 7 is being freed. Following rule #1, look at the value at index 6, which is 0. This value indicates the neighboring block is still in use. Following rule #2, look at the value at index 9, which is -3. Again, this value indicates that this block is also in use. The newly freed block remains as independent piece.

After applying the two merge rules, the next free operation of the block starting at index 3 results in the allocation table shown as the last table in Figure 5. When a block is freed, the heap must be updated accordingly.

Therefore, the free operation involves the following steps:

1. Update the allocation array and merge neighboring blocks if possible.
2. If the newly freed block cannot be merged with any of its neighbors, insert a new entry into the heap array.
3. If the newly freed block can be merged with one of its neighbors, the heap entry representing the neighboring block must be updated, and the updated entry rearranged according to its new size.
4. If the newly freed block can be merged with both of its neighbors, the heap entry representing one of the neighboring blocks must be deleted from the heap, and the heap entry representing the other neighboring block must be updated and rearranged according to its new size.

Fixed-Size Memory Management in Embedded Systems

Another approach to memory management uses the method of fixed-size memory pools. This approach is commonly found in embedded networking code, such as in embedded protocol stacks implementation. As shown in Figure 10, the available memory space is divided into variously sized memory pools. All blocks of the same memory pool have the same size. In this example, the memory space is divided into three pools of block sizes 32, 50, and 128 respectively. Each memory-pool control structure maintains information such as the block size, total number of blocks, and number of free blocks. In this example, the memory pools are linked together and sorted by size. Finding the smallest size adequate for an allocation requires searching through this link and examining each control structure for the first adequate block size.

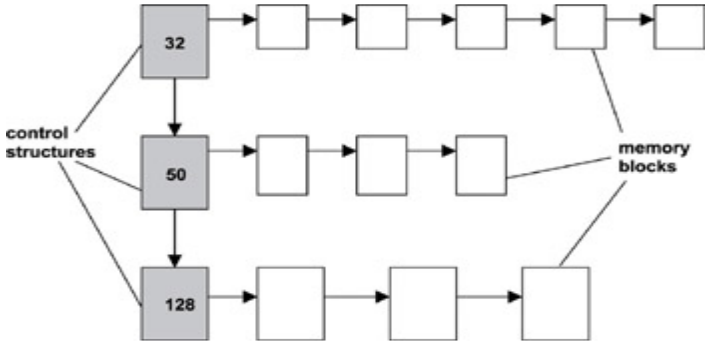


Figure 10: Management based on memory pools.

A successful allocation results in an entry being removed from the memory pool. A successful deallocation results in an entry being inserted back into the memory pool. The memory pool structure shown in Figure 10 is a singly linked list. Therefore, memory allocation and deallocation takes place at the beginning of this list. This method is not as flexible as the algorithm introduced earlier in 'Dynamic Memory Allocation in Embedded Systems' and also has some drawbacks.

In real-time embedded systems, a task's memory requirement often depends on its operating environment. This environment can be quite dynamic. This method does not work well for embedded applications that constantly operate in dynamic environments because it is nearly impossible to anticipate the memory block sizes that the task might commonly use. This issue results in increased internal memory fragmentation per allocation. In addition, the number of blocks to allocate for each size is also impossible to predict.

In many cases, the memory pools are constructed based on a number of assumptions. The result is that some memory pools are under used or not used at all, while others are overused. On the other hand, this memory allocation method can actually reduce internal fragmentation and provide high utilization for static embedded applications. These applications are those with predictable environments, a known number of running tasks at the start of application execution, and initially known required memory block sizes.

One advantage of this memory management method is that it is more deterministic than the heap method algorithm. In the heap method, each malloc or free operation can potentially trigger a rearrangement of the heap. In the memory-pool method, memory blocks are taken or are returned from the beginning of the list so the operation takes constant time. The memory pool does not require restructuring.

Blocking vs. Non-Blocking Memory Functions

The malloc and free functions do not allow the calling task to block and wait for memory to become available. In many real-time embedded systems, tasks compete for the limited system memory available. Oftentimes, the memory exhaustion condition is only temporary. For some tasks when a memory allocation request fails, the task must backtrack to an execution checkpoint and perhaps restart an operation. This issue is undesirable as the operation can be expensive. If tasks have built-in knowledge that the memory congestion condition can occur but only momentarily, the tasks can be designed to be more flexible. If such tasks can tolerate the allocation delay, the tasks can choose to wait for memory to become available instead of either failing entirely or backtracking.

For example, the network traffic pattern on an Ethernet network is bursty. An embedded networking node might receive few packets for a period and then suddenly be flooded with packets at the highest allowable bandwidth of the physical network. During this traffic burst, tasks in the embedded node that are in the process of sending data can experience temporary memory exhaustion problems because much of the available memory is used for packet reception. These sending tasks can wait for the condition to subside and then resume their operations.

In practice, a well-designed memory allocation function should allow for allocation that permits blocking forever, blocking for a timeout period, or no blocking at all. This chapter uses the memory-pool approach to demonstrate how to implement a blocking memory allocation function.

As shown in Figure 11, a blocking memory allocation function can be implemented using both a counting semaphore and a mutex lock. These synchronization primitives are created for each memory pool and are kept in the control structure. The counting semaphore is initialized with the total number of available memory blocks at the creation of the memory pool. Memory blocks are allocated and freed from the beginning of the list.

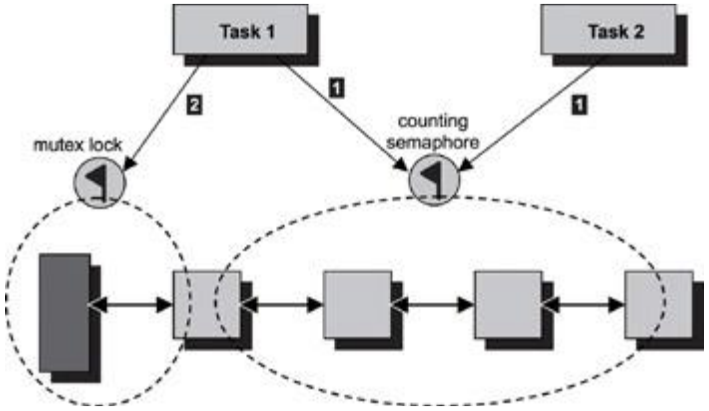


Figure 11: Implementing a blocking allocation function using a mutex and a counting semaphore.

Multiple tasks can access the free-blocks list of the memory pool. The control structure is updated each time an allocation or a deallocation occurs. Therefore, a mutex lock is used to guarantee a task exclusive access to both the free-blocks list and the control structure. A task might wait for a block to become available, acquire the block, and then continue its execution. In this case, a counting semaphore is used.

For an allocation request to succeed, the task must first successfully acquire the counting semaphore, followed by a successful acquisition of the mutex lock.

The successful acquisition of the counting semaphore reserves a piece of the available blocks from the pool. A task first tries to acquire the counting semaphore. If no blocks are available, the task blocks on the counting semaphore, assuming the task is prepared to wait for it. If a resource is available, the task acquires the counting semaphore successfully. The counting semaphore token count is now one less than it was. At this point, the task has reserved a piece of the available blocks but has yet to obtain the block.

Next, the task tries to lock the mutex. If another task is currently getting a block out of the memory pool or if another task is currently freeing a block back into the memory pool, the mutex is in the locked state. The task blocks waiting for the mutex to unlock. After the task locks the mutex, the task retrieves the resource from the list. The counting semaphore is released when the task finishes using the memory block.

The pseudo code for memory allocation using a counting semaphore and mutex lock is provided in Listing 1.

Listing 1: Pseudo code for memory allocation.

```
Acquire(Counting_Semaphore)
Lock(mutex)
Retrieve the memory block from the pool
Unlock(mutex)
```

The pseudo code for memory deallocation using a mutex lock and counting semaphore is provided in Listing 2.

Listing 2: Pseudo code for memory deallocation.

```
Lock(mutex)
Release the memory block back to into the pool
Unlock(mutex)
Release(Counting_Semaphore)
```

This implementation shown in Listing 1 and 2 enables the memory allocation and deallocation functions to be safe for multitasking. The deployment of the counting semaphore and the mutex lock eliminates the priority inversion problem when blocking memory allocation is enabled with these synchronization primitives.

Hardware Memory Management Units

Thus far, the discussion on memory management focuses on the management of physical memory. Another topic is the management of virtual memory. Virtual memory is a technique in which mass storage (for example, a hard disk) is made to appear to an application as if the mass storage were RAM. Virtual memory address space (also called *logical address space*) is larger than the actual physical memory space. This feature allows a program larger than the physical memory to execute. The *memory management unit* (MMU) provides several functions. First, the MMU translates the virtual address to a physical address for each memory access. Second, the MMU provides memory protection.

The address translation function differs from one MMU design to another. Many commercial RTOSes do not support implementation of virtual addresses.

If an MMU is enabled on an embedded system, the physical memory is typically divided into *pages*. A set of attributes is associated with each memory page. Information on attributes can include the following:

- whether the page contains code (i.e., executable instructions) or data,
- whether the page is readable, writable, executable, or a combination of these, and
- whether the page can be accessed when the CPU is not in privileged execution mode, accessed only when the CPU is in privileged mode, or both.

All memory access is done through MMU when it is enabled. Therefore, the hardware enforces memory access according to page attributes. For example, if a task tries to write to a memory region that only allows for read access, the operation is considered illegal, and the MMU does not allow it. The result is that the operation triggers a memory access exception.

Device Management

There are number of device driver ISRs for each device in a system. Each device or device function having a separate driver, which is as per its hardware. The device driver function of device (open, close, read) calls a separate ISR. ***Device manager*** is software that manages these for all. The manager coordinates between application-process, driver and device-controller and effectively operates and adopts appropriate strategy for obtaining optimal performance for the devices. A process sends a request to the driver by an interrupt using SWI; and the driver provides the actions on calling and executing the ISR.

The Device manager

- Polls the requests at the devices and the actions occur as per their priorities.
- Manages IO Interrupts (requests) queues.
- Creates an appropriate kernel interface and API and that activates the control register specific actions of the device. [Activates device controller through the API and kernel interface.]

An OS Device manager provides and executes the modules for managing the devices and their drivers ISRs.

- Manages the physical as well as virtual devices like the pipes and sockets through a common strategy.

- Device management has three standard approaches for three types of device drivers:
  (i) Programmed I/Os by polling from each device it's the service need from each device.
  (ii) Interrupt(s) from the device drivers device- ISR and
  (iii) Device uses DMA operation used by the devices to access the memory. Most common is the use of device driver ISRs.

Device Manager Functions

- Device Detection and Addition
- Device Deletion
- Device Allocation and Registration

- Detaching and Deregistration
- Restricting Device to a specific process
- Device Sharing
- Device control
- Device Access Management
- Device Buffer Management
- Device Queue, Circular-queue or blocks of queues Management
- Device drivers updating and upload of new device-functions
- Backup and restoration

Set of Command Functions for Device Management

- create
- open
- write
- read
- **ioctl** - specified device configured for specific functions and given specific papameters.
- **close and delete** – *close* is for de-registering the device from the system and *delete* is for close and detaching the device.

In a system the **ioctl()** is used for the following: (*i*) Accessing specific partition information (*ii*) Defining commands and control functions of device registers (*iii*) IO channel control

The ioctl ( ) has three arguments for the device-specific parameters.

1. First Argument: Defines the chosen device and its function by passing as argument the device descriptor (a number), for example, fd or sfd
   Example is fd = 1 for read, fd = 2 for write.
2. Second Argument: Defines the control option or uses option for the IO device, for example, baud rate or other parameter optional function
3. Third Argument: Values needed by the defined function are at the third argument

Example
Status = ioctl (fd, FIOBAUDRATE, 19200) is an instruction in RTOS VxWorks. *fd* is the device descriptor (an integer returned when the device is opened) and FIOBAUDRATE is the function that takes value = 19200 from the argument. This at configures the device for operation at 19200-baud rate.

A device Driver ISR uses several OS functions. Examples are as follows:

- *intlock* ( ) to disable device-interrupts systems,
- intUnlock ( ) to enable device-interrupts,
- *intConnect* ( ) to connect a C function to an interrupt vector
- Interrupt vector address for a device ISR points to its specified C function.
- intContext ( ) finds whether interrupt is called when an ISR was in execution

UNIX Device driver functions

Unix OS facilitates that for devices and files have an analogous implementation as far as possible. A device has *open* ( ), *close* ( ), *read* ( ), *write* ( ) functions analogous to a file *open*, *close*, *read* and *write* functions.

The following are the **in-kernel** commands:
(i) select ( ) to check whether read/write will succeed and then select
(*ii*) *ioctl* ( ) to transfer driver-specific information to the device driver.
(*iii*) *stop* ( ) to cancel the output activity from the device.
(iv) *strategy* ( ) to permit a block *read or write* or character *read or write*

The device manager initializes, controls and drives the physical and virtual devices of the system. The main classes of devices are

1. Char devices - data read in stream of characters(bytes) and
2. Block devices - data read in blocks(KB)

I/O Subsystem

Introduction

All embedded systems include some form of input and output (I/O) operations. These I/O operations are performed over different types of I/O devices. A vehicle dashboard display, a touch screen on a PDA, the hard disk of a file server, and a network interface card are all examples of I/O devices found in embedded systems.

Often, an embedded system is designed specifically to handle the special requirements associated with a device. A cell phone, pager, and a handheld MP3 player are a few examples of embedded systems built explicitly to deal with I/O devices. I/O operations are interpreted differently depending on the viewpoint taken and place different requirements on the level of understanding of the hardware details.

From the perspective of the RTOS, I/O operations imply locating the right device for the I/O request, locating the right device driver for the device, and issuing the request to the device driver. Sometimes the RTOS is required to ensure synchronized access to the device. The RTOS must facilitate an abstraction that hides both the device characteristics and specifics from the application developers.

Basic I/O Concepts

The combination of I/O devices, associated device drivers, and the I/O subsystem comprises the overall I/O system in an embedded environment. The purpose of the I/O subsystem is to hide the device-specific information from the kernel as well as from the application developer and to provide a uniform access method to the peripheral I/O devices of the system.

Figure 12 illustrates the I/O subsystem in relation to the rest of the system in a layered software model. As shown, each descending layer adds additional detailed information to the architecture needed to manage a given device.

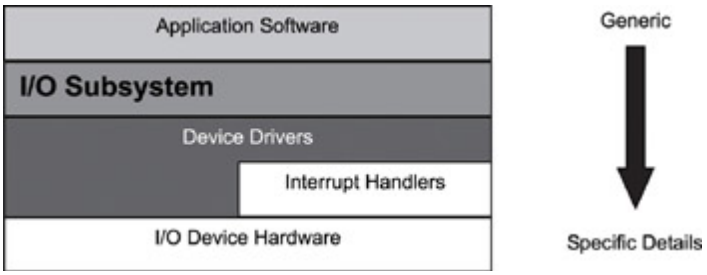


Figure 12: I/O subsystem and the layered model.

Port-Mapped vs. Memory-Mapped I/O and DMA

The bottom layer contains the I/O device hardware. The I/O device hardware can range from low-bit rate serial lines to hard drives and gigabit network interface adaptors. All I/O devices must be initialized through device control registers, which are usually external to the CPU. They are located on the CPU board or in the devices themselves. During operation, the device registers are accessed again and are programmed to process data transfer requests, which is called *device control*. To access these devices, it is necessary for the developer to determine if the device is port mapped or memory mapped. This information determines which of two methods, port-mapped I/O or memory-mapped I/O, is deployed to access an I/O device.

When the I/O device address space is separate from the system memory address space, special processor instructions, such as the IN and OUT instructions offered by the Intel processor, is used to transfer data between the I/O device and a microprocessor register or memory.

The I/O device address is referred to as the *port number* when specified for these special instructions. This form of I/O is called *port-mapped I/O*, as shown in Figure 13.

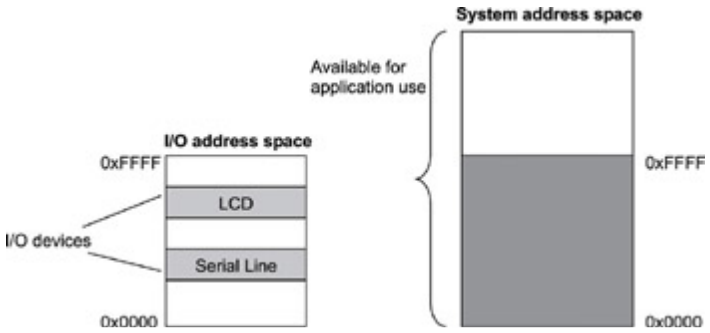


Figure 13: Port-mapped I/O.

The devices are programmed to occupy a range in the I/O address space. Each device is on a different I/O port. The I/O ports are accessed through special processor instructions, and actual physical access is accomplished through special hardware circuitry. This I/O method is also called *isolated I/O* because the memory space is isolated from the I/O space, thus the entire memory address space is available for application use.

The other form of device access is memory-mapped I/O, as shown in Figure 14. In *memory-mapped I/O*, the device address is part of the system memory address space. Any machine instruction that is encoded to transfer data between a memory location and the processor or between two memory locations can potentially be used to access the I/O device. The I/O device is treated as if it were another memory location. Because the I/O address space occupies a range in the system memory address space, this region of the memory address space is not available for an application to use.

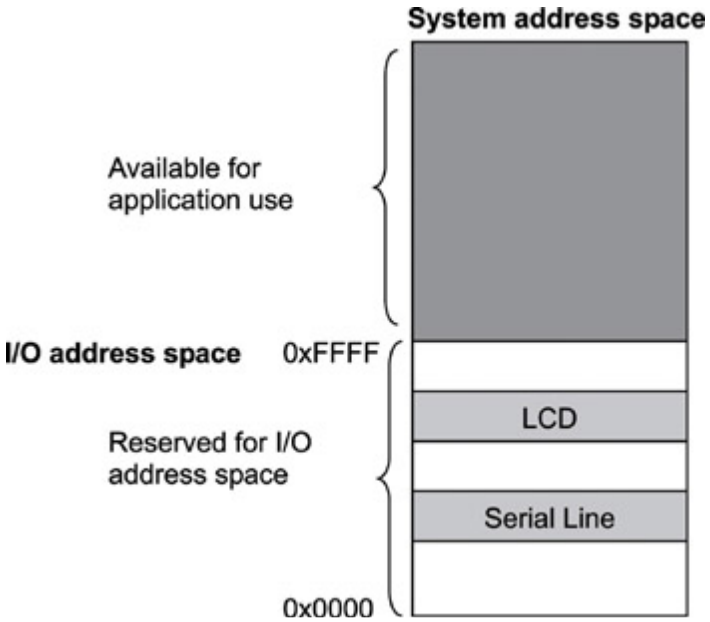


Figure 14: Memory-mapped I/O.

The memory-mapped I/O space does not necessarily begin at offset 0 in the system address space, as illustrated in Figure 14. It can be mapped anywhere inside the address space. This issue is dependent on the system implementation.

Commonly, tables describing the mapping of a device's internal registers are available in the device hardware data book. The device registers appear at different offsets in this map. Sometimes the information is presented in the *"base + offset"* format. This format indicates that the addresses in the map are relative, i.e., the offset must be added to the start address of the I/O space for port-mapped I/O or the offset must be added to the base address of the system memory space for memory-mapped I/O in order to access a particular register on the device.

The processor has to do some work in both of these I/O methods. Data transfer between the device and the system involves transferring data between the device and the processor register and then from the processor register to memory. The transfer speed might not meet the needs of high-speed I/O devices because of the additional data copy involved.

Direct memory access (DMA) chips or controllers solve this problem by allowing the device to access the memory directly without involving the processor, as shown in Figure 15. The processor is used to set up the DMA controller before a data transfer operation begins, but the processor is bypassed during data transfer, regardless of whether it is a read or write operation. The transfer speed depends on the transfer speed of the I/O device, the speed of the memory device, and the speed of the DMA controller.

In essence, the DMA controller provides an alternative data path between the I/O device and the main memory. The processor sets up the transfer operation by specifying the source address, the destination memory address, and the length of the transfer to the DMA controller.

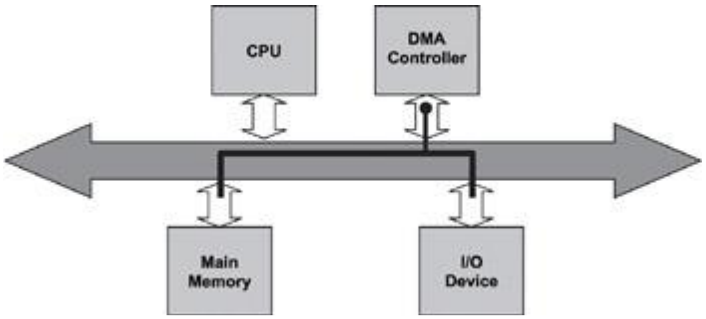


Figure 15: DMA I/O.

Character-Mode vs. Block-Mode Devices

I/O devices are classified as either character-mode devices or block-mode devices. The classification refers to how the device handles data transfer with the system.

Character-mode devices allow for unstructured data transfers. The data transfers typically take place in serial fashion, one byte at a time. Character-mode devices are usually simple devices, such as the serial interface or the keypad. The driver buffers the data in cases where the transfer rate from system to the device is faster than what the device can handle.

Block-mode devices transfer data one block at time, for example, 1,024 bytes per data transfer. The underlying hardware imposes the block size. Some structure must be imposed on the data or some transfer protocol enforced. Otherwise an error is likely to occur. Therefore, sometimes it is necessary for the block-mode device driver to perform additional work for each read or write operation, as shown in Figure 16.

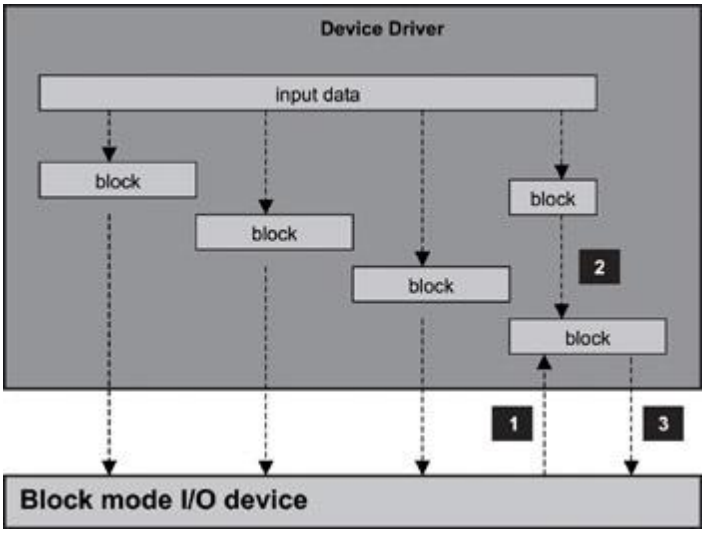


Figure 16: Servicing a write operation for a block-mode device.

As illustrated in Figure 16, when servicing a write operation with large amounts of data, the device driver must first divide the input data into multiple blocks, each with a device-specific block size. In this example, the input data is divided into four blocks, of which all but the last block is of the required block size. In practice, the last partition often
is smaller than the normal device block size.

Another strategy used by block-mode device drivers for small write operations is to accumulate the data in the driver cache and to perform the actual write after enough data has accumulated for a required block size. This technique also minimizes the number of device accesses. Some disadvantages occur with this approach. First, the device driver satisfies a read operation. The delayed write associated with caching can also cause data loss if a failure occurs and if the driver is shut down and unloaded ungracefully. Data caching in this case implies data copying that can result in lower I/O performance.

The I/O Subsystem

Each I/O device driver can provide a driver-specific set of I/O application programming interfaces to the applications. This arrangement requires each application to be aware of the nature of the underlying I/O device, including the restrictions imposed by the device. The API set is driver and implementation specific, which makes the applications using this API set difficult to port. To reduce this implementation-dependence, embedded systems often include an *I/O subsystem.*

The I/O subsystem defines a standard set of functions for I/O operations in order to hide device peculiarities from applications. All I/O device drivers conform to and support this function set because the goal is to provide uniform I/O to applications across a wide spectrum of I/O devices of varying types.

The following steps must take place to accomplish uniform I/O operations at the application-level.

1. The I/O subsystem defines the API set.
2. The device driver implements each function in the set.
3. The device driver exports the set of functions to the I/O subsystem.
4. The device driver does the work necessary to prepare the device for use. In addition, the driver sets up an association between the I/O subsystem API set and the corresponding device-specific I/O calls.
5. The device driver loads the device and makes this driver and device association known to the I/O subsystem. This action enables the I/O subsystem to present the illusion of an abstract or virtual instance of the device to applications.

Standard I/O Functions

The I/O subsystem defines a set of functions as the standard I/O function set. Table 1 lists those functions that are considered part of the set in the general approach to uniform I/O.
The number of functions in the standard I/O API set, function names, and functionality of each is dependent on the embedded system and implementation. The next few sections put these functions into perspective.

Table 1: I/O functions.

Function	Description
Create	Creates a virtual instance of an I/O device

Destroy	Deletes a virtual instance of an I/O device
Open	Prepares an I/O device for use.
Close	Communicates to the device that its services are no longer required, which typically initiates device-specific cleanup operations.
Read	Reads data from an I/O device
Write	Writes data into an I/O device
Ioctl	Issues control commands to the I/O device (I/O control)

Note that all these functions operate on a so-called 'virtual instance' of the I/O device. In other words, these functions do not act directly on the I/O device, but rather on the driver, which passes the operations to the I/O device. When the open, read, write, and close operations are described, these operations should be understood as acting indirectly on an I/O device through the agency of a virtual instance.

The *create* function creates a virtual instance of an I/O device in the I/O subsystem, making the device available for subsequent operations, such as open, read, write, and ioctl. This function gives the driver an opportunity to prepare the device for use. Preparations might include mapping the device into the system memory space, allocating an available interrupt request line (IRQ) for the device, installing an ISR for the IRQ, and initializing the device into a known state. The driver allocates memory to store instance-specific information for subsequent operations. A reference to the newly created device instance is returned to the caller.

The *destroy* function deletes a virtual instance of an I/O device from the I/O subsystem. No more operations are allowed on the device after this function completes. This function gives the driver an opportunity to perform cleanup operations, such as un-mapping the device from the system memory space, de-allocating the IRQ, and removing the ISR from the system. The driver frees the memory that was used to store instance-specific information.

The *open* function prepares an I/O device for subsequent operations, such as read and write. The device might have been in a disabled state when the create function was called. Therefore, one of the operations that the open function might perform is enabling the device. Typically, the open operation can also specify modes of use; for example, a device might be opened for read-only operations or write-only operations or for receiving control commands. The reference to the newly opened I/O device is returned to the caller. In some implementations, the I/O subsystem might supply only one of the two functions, create and open, which implements most of the functionalities of both create and open due to functional overlaps between the two operations.

The *close* function informs a previously opened I/O device that its services are no longer required. This process typically initiates device-specific cleanup operations. For example, closing a device might cause it to go to a standby state in which it consumes little power. Commonly, the I/O subsystem supplies only one of the two functions, destroy
and close, which implements most of the functionalities of both destroy and close, in the case where one function implements both the create and open operations.

The *read* function retrieves data from a previously opened I/O device. The caller specifies the amount of data to retrieve from the device and the location in memory where the data is to be stored. The caller is completely isolated from the device details and is not concerned with the I/O restrictions imposed by the device.

The *write* function transfers data from the application to a previously opened I/O device. The caller specifies the amount of data to transfer and the location in memory holding the data to be transferred. Again, the caller is isolated from the device I/O details.

The *Ioctl* function is used to manipulate the device and driver operating parameters at runtime. An application is concerned with only two things in the context of uniform I/O: the device on which it wishes to perform I/O operations and the functions presented in this section. The I/O subsystem exports this API set for application use.

Mapping Generic Functions to Driver Functions

The individual device drivers provide the actual implementation of each function in the uniform I/O API set. Figure 17 gives an overview of the relationship between the I/O API set and driver internal function set.

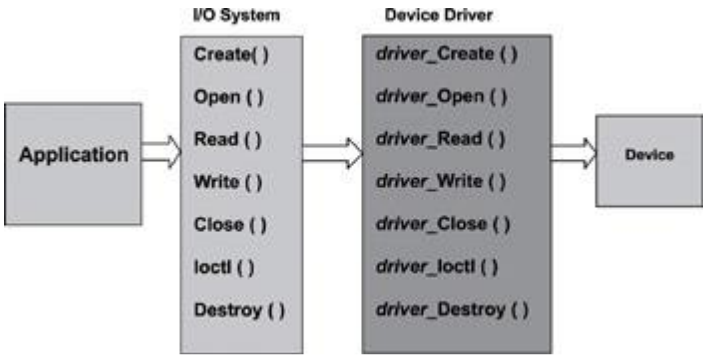


Figure 17: I/O function mapping.

As illustrated in Figure 17, the I/O subsystem-defined API set needs to be mapped into a function set that is specific to the device driver for any driver that supports uniform I/O. The functions that begin with the *driver_* prefix in Figure 17 refer to implementations that are specific to a device driver. The uniform I/O API set can be represented in the C programming language syntax as a structure of function pointers, as shown in the left-hand side of Listing 3.

Listing 3: C structure defining the uniform I/O API set.

```
typedef struct
{
int (*Create)( );
int (*Open) ( );
int (*Read)( );
int (*Write) ( );
int (*Close) ( );
int (*Ioctl) ( );
int (*Destroy) ( );
} UNIFORM_IO_DRV;
```

The mapping process involves initializing each function pointer with the address of an associated internal driver function, as shown in Listing 4. These internal driver functions can have any name as long as they are correctly mapped.

Listing 4: Mapping uniform I/O API to specific driver functions.

```
UNIFORM_IO_DRV ttyIOdrv;
ttyIOdrv.Create = tty_Create;
ttyIOdrv.Open = tty_Open;
ttyIOdrv.Read = tty_Read;
ttyIOdrv.Write = tty_Write;
ttyIOdrv.Close = tty_Close;
ttyIOdrv.Ioctl = tty_Ioctl;
ttyIOdrv.Destroy = tty_Destroy;
```

An I/O subsystem usually maintains a *uniform I/O driver table*. Any driver can be installed into or removed from this driver table by using the utility functions that the I/O subsystem provides. Figure 18 illustrates this concept.

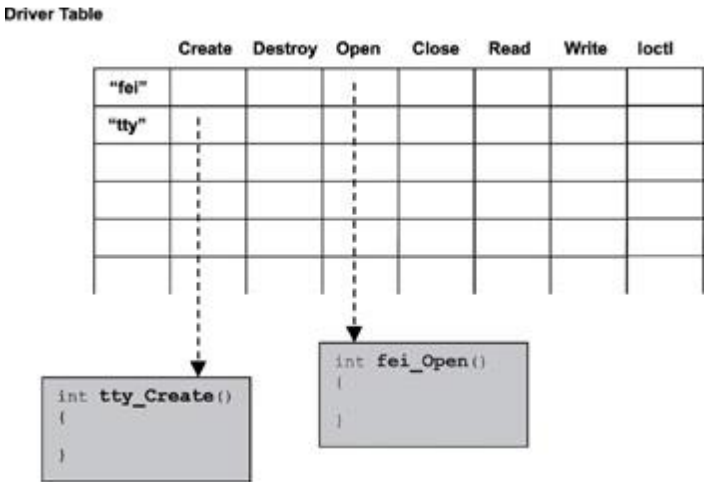


Figure 18: Uniform I/O driver table.

Each row in the table represents a unique I/O driver that supports the defined API set. The first column of the table is a generic name used to associate the uniform I/O driver with a particular type of device. In Figure 18, a uniform I/O driver is provided for a serial line terminal device, tty. The table element at the second row and column contains a pointer to the internal driver function, tty_Create(). This pointer, in effect, constitutes an association between the generic create function and the driver-specific create function. The association is used later when creating virtual instances of a device.

These pointers are written to the table when a driver is installed in the I/O subsystem, typically by calling a utility function for driver installation. When this utility function is called, a reference to the newly created driver table entry is returned to the caller.

Associating Devices with Device Drivers

In standard I/O functions, the create function is used to create a virtual instance of a device. The I/O subsystem tracks these virtual instances using the *device table*. A newly created virtual instance is given a unique name and is inserted into the device table, as shown in Figure 19. Figure 19 also illustrates the device table's relationship to the driver table.

Each entry in the device table holds generic information, as well as instance-specific information. The generic part of the device entry can include the unique name of the device instance and a reference to the device driver. In Figure 19, a device instance name is constructed using the generic device name and the instance number. The device named tty0 implies that this I/O device is a serial terminal device and is the first instance created in the system. The driver-dependent part of the device entry is a block of memory allocated by the driver for each instance to hold instance-specific data. The driver initializes and maintains it. The content of this information is dependent on the driver implementation. The driver is the only entity that accesses and interprets this data.

A reference to the newly created device entry is returned to the caller of the create function. Subsequent calls to the open and destroy functions use this reference.

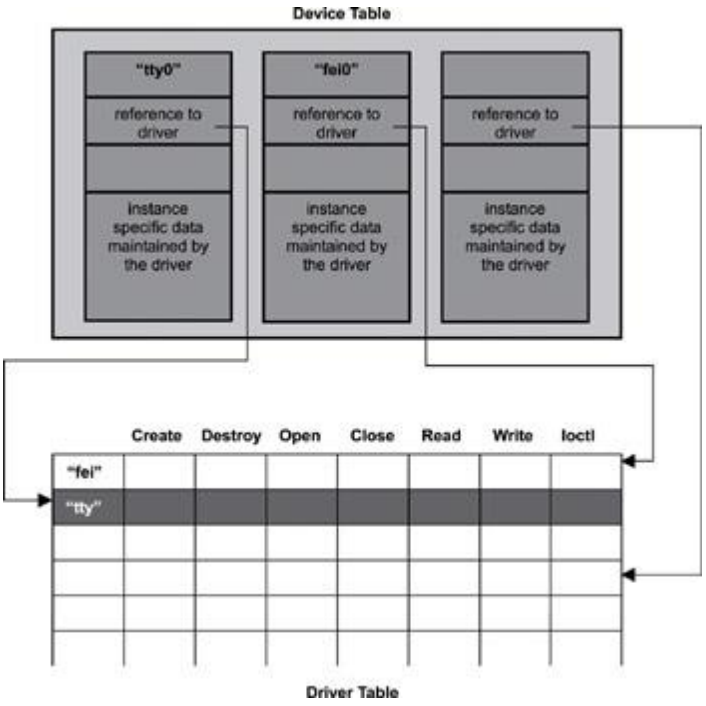


Figure 19: Associating devices with drivers.

Other RTOS Services

Introduction

A good real-time embedded operating system avoids implementing the kernel as a large, monolithic program. The kernel is developed instead as a micro-kernel. The goal of the micro-kernel design approach is to reduce essential kernel services into a small set and to provide a framework in which other optional kernel services can be implemented as independent modules. These modules can be placed outside the kernel. Some of these modules are part of special server tasks. This structured approach makes it possible to extend the kernel by adding additional services or to modify existing services without affecting users. This level of implementation flexibility is highly desirable. The resulting benefit is increased system configurability because each embedded application requires a specific set of system services with respect to its characteristics. This combination can be quite different from application to application.

The micro-kernel provides core services, including task-related services, the scheduler service, and synchronization primitives. This chapter discusses other common building blocks, as shown in Figure 20.

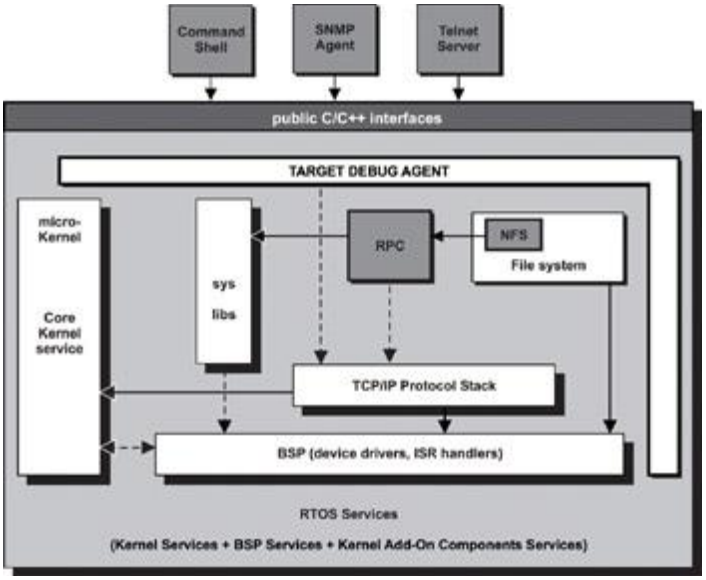


Figure 20: Overview.

Other Building Blocks

These other common building blocks make up the additional kernel services that are part of various embedded applications. The other building blocks include the following:

- TCP/IP protocol stack,
- file system component,
- remote procedure call component,
- command shell,
- target debut agent, and
- other components.

TCP/IP Protocol Stack

The network protocol stacks and components, as illustrated in Figure 21, provide useful system services to an embedded application in a networked environment. The TCP/IP protocol stack provides transport services to both higher layer, well-known protocols, including Simple Network Management Protocol (SNMP), Network File System (NFS), and Telnet, and to user-defined protocols. The transport service can be either reliable connection-oriented service over the TCP protocol or unreliable connectionless service over the UDP protocol. The TCP/IP protocol stack can operate over various types of physical connections and networks, including Ethernet, Frame Relay, ATM, and ISDN networks using different frame encapsulation protocols, including the point-to-point protocol. It is common to find the transport services offered through standard Berkeley socket interfaces.

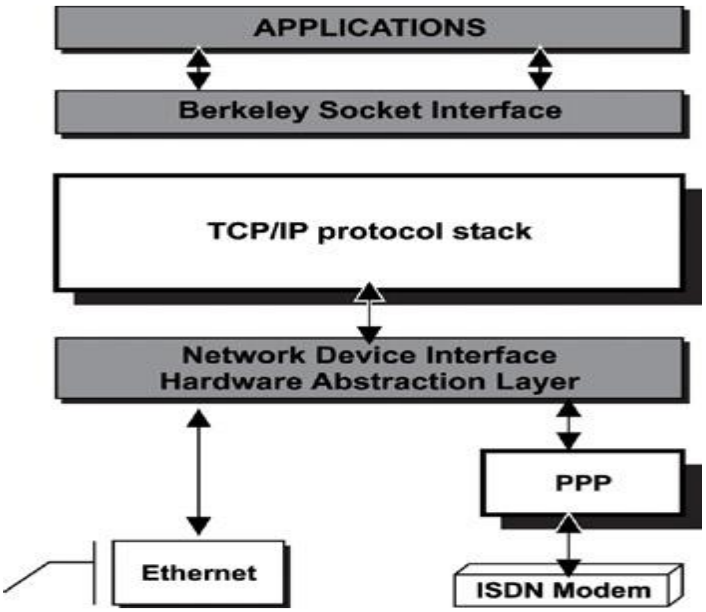


Figure 21: TCP/IP protocol stack component.

File System Component

The file system component, as illustrated in Figure 22, provides efficient access to both local and network mass storage devices. These storage devices include but are not limited to CD-ROM, tape, floppy disk, hard disk, and flash memory devices. The file system component structures the storage device into supported formats for writing information to and for accessing information from the storage device. For example, CD-ROMs are formatted and managed according to ISO 9660 standard file system specifications; floppy disks and hard disks are formatted and managed according to MS-DOS FAT file system conventions and specifications; NFS allows local applications to access files on remote systems as an NFS client. Files located on an NFS server are treated exactly as though they were on a local disk. Because NFS is a protocol, not a file system format, local applications can access any format files supported by the NFS server. File system components found in some real-time RTOS provide high-speed proprietary file systems in place of common storage devices.

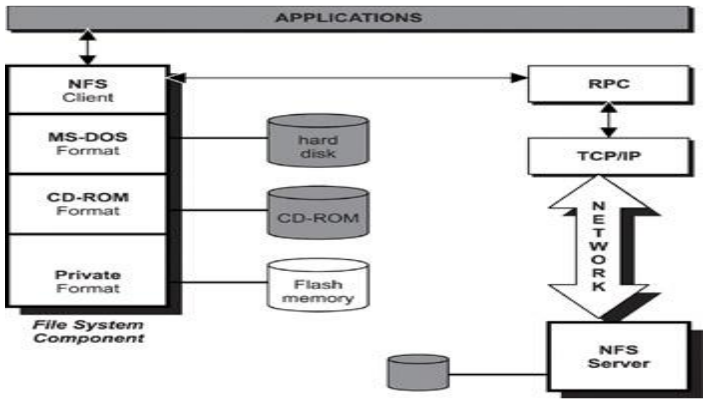


Figure 22: File system component.

Remote Procedure Call Component

The remote procedure call (RPC) component allows for distributed computing. The RPC server offers services to external systems as remotely callable procedures. A remote RPC client can invoke these procedures over the network using the RPC protocol. To use a service provided by an RPC server, a client application calls routines, known as *stubs*, provided by the RPC client residing on the local machine.

The RPC client in turn invokes remote procedure calls residing in the RPC server on behalf of the calling application. The primary goal of RPC is to make remote procedure calls transparent to applications invoking the local call stubs. To the client application, calling a stub appears no different from calling a local procedure. The RPC client and server can run on top of different operating systems, as well as different types of hardware. As an example of such transparency, note that NFS relies directly upon RPC calls to support the illusion that all files are local to the client machine.

To hide both the server remoteness, as well as platform differences from the client application, data that flows between the two computing systems in the RPC call must be translated to and from a common format. External data representation (XDR) is a method that represents data in an OS- and machine-independent manner. The RPC client translates data passed in as procedure parameters into XDR format before making the remote procedure call.

The RPC server translates the XDR data into machine-specific data format upon receipt of the procedure call request. The decoded data is then passed to the actual procedure to be invoked on the server machine. This procedure's output data is formatted into XDR when returning it to the RPC client. The RPC concept is illustrated in Figure 23.

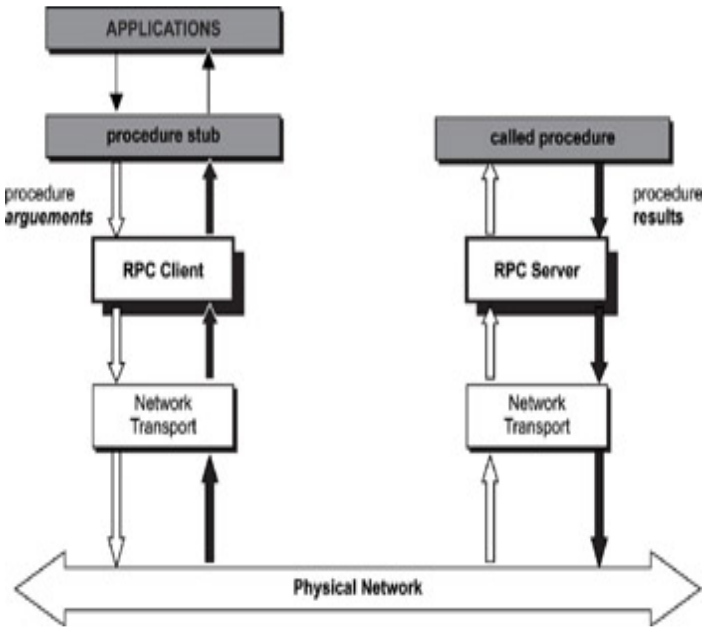


Figure 23: Remote procedure calls.

Command Shell

The *command shell*, also called the *command interpreter*, is an interactive component that provides an interface between the user and the real-time operating system. The user can invoke commands, such as ping, ls, loader, and route through the shell. The shell interprets these commands and makes corresponding calls into RTOS routines. These routines can be in the form of loadable program images, dynamically created programs (dynamic tasks), or direct system function calls if supported by the RTOS. The programmer can experiment with different global system calls if the command shell supports this feature. With this feature, the shell can become a great learning tool for the RTOS in which it executes, as illustrated in Figure 24.

Some command shell implementations provide a programming interface. A programmer can extend the shell's functionality by writing additional commands or functions using the shell's application program interface (API). The shell is usually accessed from the host system using a terminal emulation program over a serial interface. It is possible to access the shell over the network, but this feature is highly implementation-dependent. The shell becomes a good debugging tool when it supports available debug agent commands. A host debugger is not always available and can be tedious to set up. On the other hand, the programmer can immediately begin debugging when a debug agent is present on the target system, as well as a command shell.

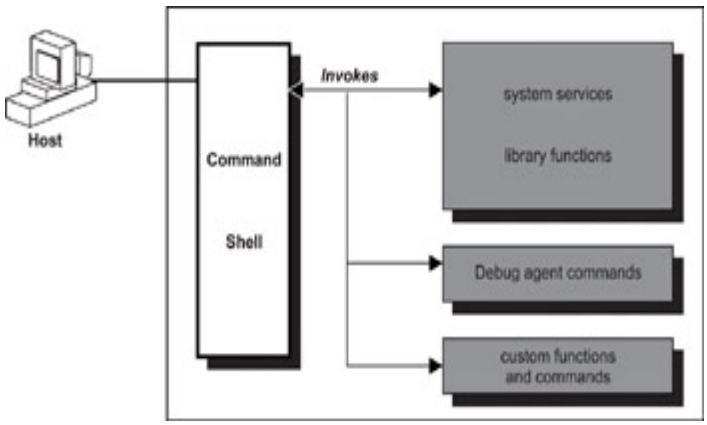


Figure 24: RTOS command shell.

Target Debug Agent

Every good RTOS provides a target debug agent. Through either the target shell component or a simple serial connection, the debug agent offers the programmer a rich set of debug commands or capabilities. The debug agent allows the programmer to set up both execution and data access break points. In addition, the programmer can use the debug agent to examine and modify system memory, system registers, and system objects, such as tasks, semaphores and message queues. The host debugger can provide source-level debug capability by interacting with the target debug agent. With a host debugger, the user can debug the target system without having to understand the native debug agent commands. The target debug agent commands are mapped into host debugger commands that are more descriptive and easier to understand. Using an established debug protocol, the host debugger sends the user-issued debug commands to the target debug agent over the serial cable or the Ethernet network. The target debug agent acts on the commands and sends the results back to the host debugger. The host debugger displays the results in its user-friendly debug interface. The debug protocol is specific

to the host debugger and its supported debug agent. Be sure to check the host debugging tools against the supported RTOS debug agents before making a purchase.

Other Components

What has been presented so far is a very small set of components commonly found in available RTOS. Other service components include the SNMP component. The target system can be remotely managed over the network by using SNMP. The standard I/O library provides a common interface to write to and read from system I/O devices. The standard system library provides common interfaces to applications for memory functions and string manipulation functions. These library components make it straightforward to port applications written for other operating systems as long as they use standard interfaces. The possible services components that an RTOS can provide are limited only by imagination. The more an embedded RTOS matures the more components and options it provides to the developer. These components enable powerful embedded applications programming, while at the same time save overall development costs. Therefore, choose the RTOS wisely.

Component Configuration

The available system memory in many embedded systems is limited. Therefore, only the necessary service components are selected into the final application image. Frequently programmers ask how to configure a service component into an embedded application. In a simplified view, the selection and consequently the configuration of service components are accomplished through a set of system configuration files. Look for these files in the RTOS development environment to gain a better understanding of available components and applicable configuration parameters.

The first level of configuration is done in a component inclusion header file. For example, call it sys_comp.h , as shown in Listing 5.

Listing 5: The sys_comp.h inclusion header file.

```
#define INCLUDE_TCPIP           1
#define INCLUDE_FILE_SYS        0
#define INCLUDE_SHELL           1
#define INCLUDE_DBG_AGENT       1
```

In this example, the target image includes the TCP/IP protocol stack, the command shell, and the debug agent. The file system is excluded because the sample target system does not have a mass storage device. The programmer selects the desired components through sys_comp.h.

The second level of configuration is done in a component-specific configuration file, sometimes called the component description file. For example, the TCP/IP component configuration file could be called net_conf.h, and the debug agent configuration file might be called the dbg_conf.h. The component-specific configuration file contains the user-configurable, component-specific operating parameters. These parameters contain default values. Listing 6 uses net_conf.h.

Listing 6: The net_conf.h configuration file.

```
#define NUM_PKT_BUFS       100
```

```
#define NUM_SOCKETS        20
#define NUM_ROUTES         35
#define NUM_NICS           40
```

In this example, four user-configurable parameters are present: the number of packet buffers to be allocated for transmitting and receiving network packets; the number of sockets to be allocated for the applications; the number of routing entries to be created in the routing table used for forwarding packets; and the number of network interface data structures to be allocated for installing network devices. Each parameter contains a default value, and the
programmer is allowed to change the value of any parameter present in the configuration file. These parameters are applicable only to the TCP/IP protocol stack component.

Component-specific parameters must be passed to the component during the initialization phase. The component parameters are set into a data structure called the component configuration table. The configuration table is passed into the component initialization routine. This level is the third configuration level. Listing 7 shows the configuration file named net_conf.c , which continues to use the network component as the example.

Listing 7: The net_conf.c configuration file.

```
#include "sys_comp.h"
#include "net_conf.h"
#if (INCLUDE_TCPIP)
struct net_conf_parms params;
params.num_pkt_bufs = NUM_PKT_BUFS;
params.num_sockets = NUM_SOCKETS;
params.num_routes = NUM_ROUTES;
params.num_NICS = NUM_NICS;
tcpip_init(&params);
#endif
```

The components are pre-built and archived. The function tcpip_init is part of the component. If INCLUDE_TCPIP is defined as 1 at the time the application is built, the call to this function triggers the linker to link the component into the final executable image. At this point, the TCP/IP protocol stack is included and fully configured.

Obviously, the examples presented here are simple, but the concepts vary little in real systems. Manual configuration, however, can be tedious when it is required to wading through directories and files to get to the configuration files. When the configuration file does not offer enough or clear documentation on the configuration parameters, the process is even harder. Some host development tools offer an interactive and visual alternative to manual component configuration. The visual component configuration tool allows the programmer to select the offered components visually. The configurable parameters are also laid out visually and are easily editable. The outputs of the configuration tool are automatically generated files similar to sys_comp.h and net_conf.h. Any modification completed through the configuration tool regenerates these files.

Basic Guidelines to choose an OS for Embedded Applications

Real time systems are classified as

- Systems with absolute deadlines, such as the nuclear reactor system are called *hard real-time systems.*
- Systems that demand good response but that allows some fudge in the deadlines are called *soft real-time systems.*

An embedded system with a single CPU can run only one process at an instance. The process at any instance may either be an ISR or kernel function or task. An RTOS use in embedded system facilities the following.

1. Provides running the user threads in kernel space so that they execute fast.
2. Provides effective handling of the ISRs, device drivers, ISTs, tasks or threads
3. Disabling and enabling of interrupts in user mode critical section code
4. Provides memory allocation and deallocation functions in fixed time and blocks of memory Provides for effectively scheduling and running and blocking of the tasks in cases of number of many tasks
5. I/O Management with devices, files, mailboxes, pipes and sockets becomes simple using an RTOS
6. Provides for the uses of Semaphore(s) by the tasks or for the shared resources (Critical sections) in a task or OS functions
7. Effective management of the multiple states of the CPU and, internal and external physical or virtual devices

Real Time Operating System based System design Principles:

Design with the ISRs and Tasks

- The embedded system hardware source calls generates interrupts
- ISR ─ only post (send) the messages
- Provides of nesting of ISRs, while tasks run concurrently
- A task─ wait and take the messages (IPCs) and post (send) the messages using the system calls.
- A task or ISR should not call another task or ISR

Each ISR design consisting of shorter code

- Since ISRs have higher priorities over the tasks, the ISR code should be made short so that the tasks don't wait longer to execute.
- A design principle is that the ISR code should be optimally short and the detailed computations be given to an IST or task by posting a message or parameters for that.
- The frequent posting of the messages by the IPC functions from the ISRs be avoided

Design with using Interrupt Service Threads or Interrupt Service tasks

- In certain RTOSes, for servicing the interrupts, there are two levels, fast level ISRs and slow level ISTs, the priorities are first for the ISRs, then for the ISTs and then the task

Design with using Interrupt Service Threads

- ISRs post the messages for the ISTs and do the detailed computations.
- If RTOS is providing for only one level, then use the tasks as interrupt service Threads

Design Each Task with an infinite loop

- Each task has a while loop which never terminates.
- A task waits for an IPC or signal to start.
- The task, which gets the signal or takes the IPC for which it is waiting, runs from the point where it was blocked or preempted.
- In preemptive scheduler, the high priority task can be delayed for some period to let the low priority task execute

Design in the form of tasks for the Better and Predictable Response Time Control

- Provide the control over the response time of the different tasks.
- The different tasks are assigned different priorities and those tasks which system needs to execute with faster response are separated out.

Response Time Control

- For example, in mobile phone device there is need for faster response to the phone call receiving task then the user key input.
- In digital camera, the task for recording the image needs faster response than the task for down loading the image on computer through USB port

Design in the form of tasks Modular Design

- System of multiple tasks makes the design modular.
- The tasks provide modular design

Design in the form of tasks for Data Encapsulation

- System of multiple tasks encapsulates the code and data of one task from the other by use of global variables in critical sections getting exclusive access by mutex.

Design with taking care of the time spent in the system calls

- Expected time in general depends on the specific target processor of the embedded system and the memory access times.

Design with Limited number of tasks

- Limit the number of tasks and select the appropriate number of tasks to increase the response time to the tasks, better control over shared resource and reduced memory requirement for stacks
- The tasks, which share the data with number of tasks, can be designed as one single task.

Use appropriate precedence assignment strategy and Use Preemption

- Use appropriate precedence assignment strategy and Use Preemption in place of Time Slicing

Avoid Task Deletion

- Create tasks at start-up only and *avoid creating and then deleting tasks later*
- Certain RTOS provide an option to make a semaphore *deletion safe.*

Use CPU idle CPU time for internal functions

Often, the CPU may not be running any task. The CPU at that instant may

- Read the internal queue.
- Manage the memory.
- Search for a free block of memory.

Design with Memory Allocation and De-Allocation by the Task

- If memory allocation and de-allocation are done by the task then the number of functions required as the RTOS functions is reduced.
- This reduces the interrupt-latency periods. As execution of these functions takes significant time by the RTOS whenever the RTOS preempts a task.
- Further, if fixed sized memory blocks are allocated, then the predictability of time taken in memory allocation is there.

Design with taking care of the Shared Resource or Data among the Tasks

- The ISR coding should be as like a reentrant function or should take care of problems from the shared resources or data such as buffer or global variables
- If possible, instead of disabling the interrupts only the task-switching flag changes should only be prevented. [It is by using the semaphore.]

Design with limited RTOS functions

- Use an RTOS, which can be configured. RTOS provides during execution of the codes enabling the limited RTOS functions. For example, if queue and pipe functions are not used during execution, then disable these during run.
- Scalable RTOS
- Hierarchical RTOS

Use an RTOS, which provides for creation of scalable code

- Only the needed functions include in the executable files, and those functions of kernel and RTOS not needed do not include in the executable files

Use an RTOS, which is hierarchical

- The needed functions extended and interfaced with the functionalities
- Configured with specific processor and devices

Encapsulation Using the Semaphores

- Semaphores, queues, and messages should not be not globally shared variables, and let each should one be shared between a set of tasks only and encapsulated from the rest.
- A semaphore encapsulates the data during a critical section or encapsulates a buffer from reading task or writing into the buffer by multiple tasks concurrently.

Encapsulation Using Queues

- A queue can be used to encapsulate the messages to a task at an instance from the multiple tasks.
- Assume that a display task is posted a menu for display on a touch screen in a PDA
- Multiple tasks can post the messages into the queue for display.
- When one task is posting the messages and these messages are displayed, another task should be blocked from posting the messages.
- We can write a task, which takes the input messages from other tasks and posts these messages to the display task only after querying whether the queue is empty