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

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.

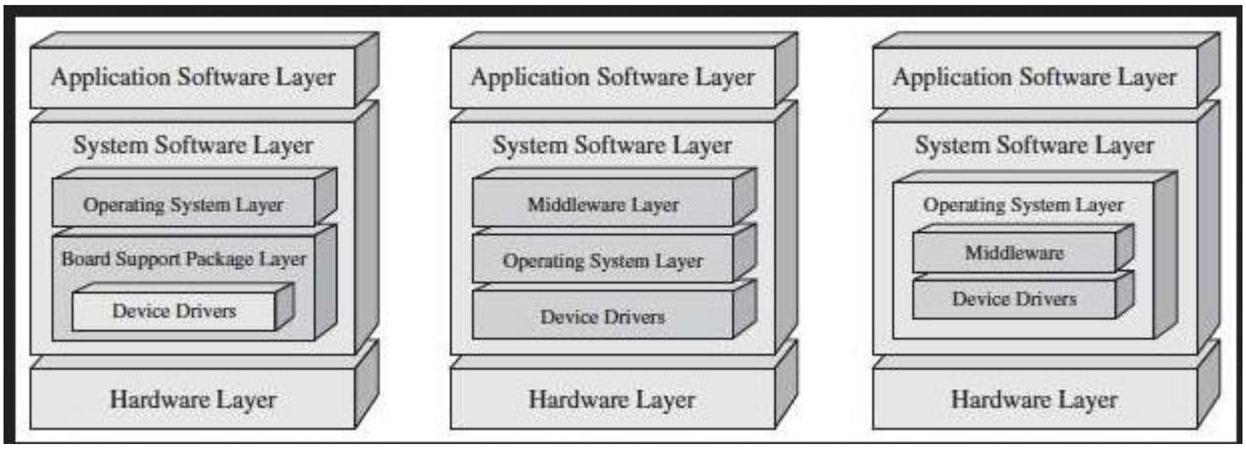


Figure 1. OSs and the Embedded Systems Model.

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.

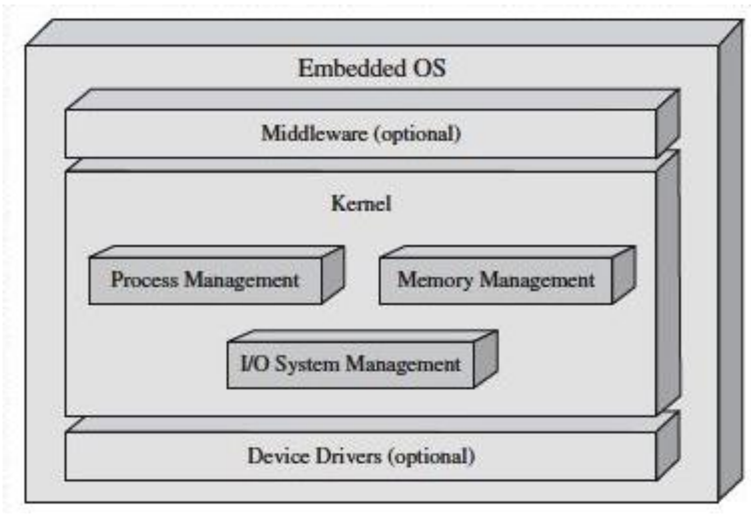


Figure 2a. General OS model.

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).

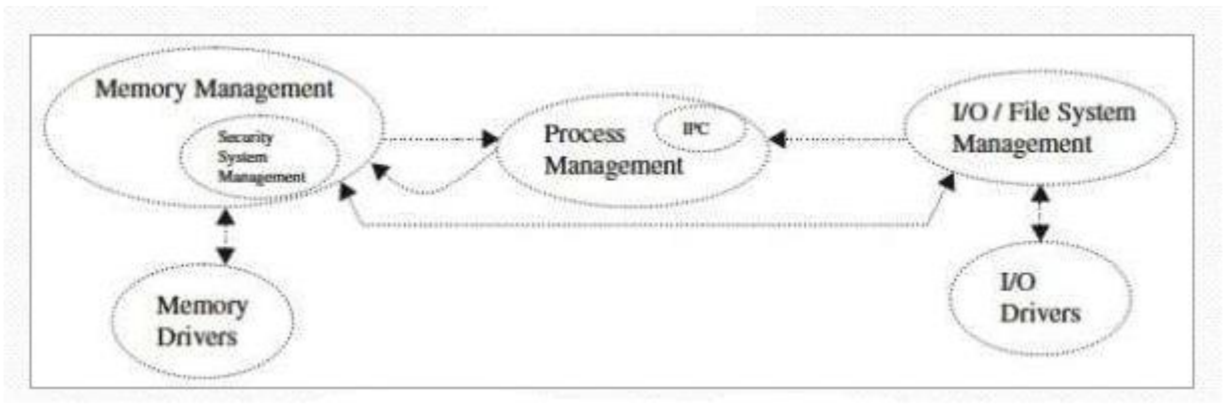


Figure 2b. Kernel subsystem dependencies.

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.

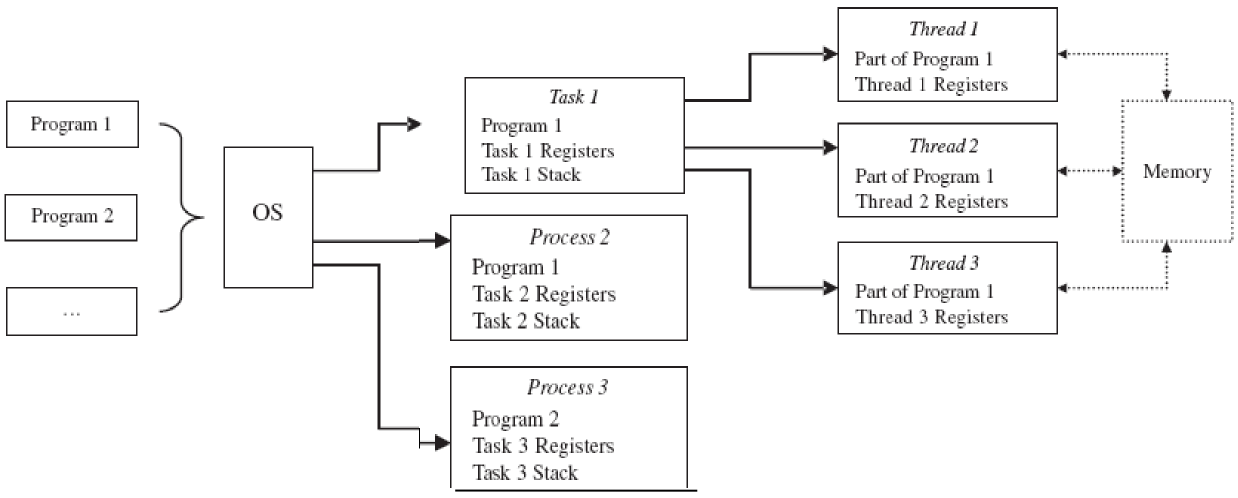
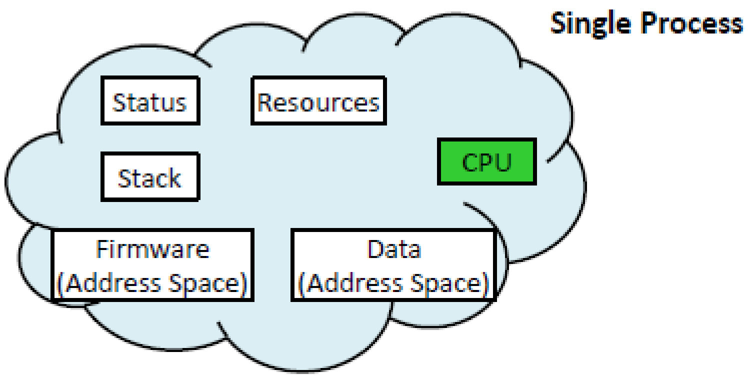


Fig 3 .Tasks, Processes and threads

- 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.

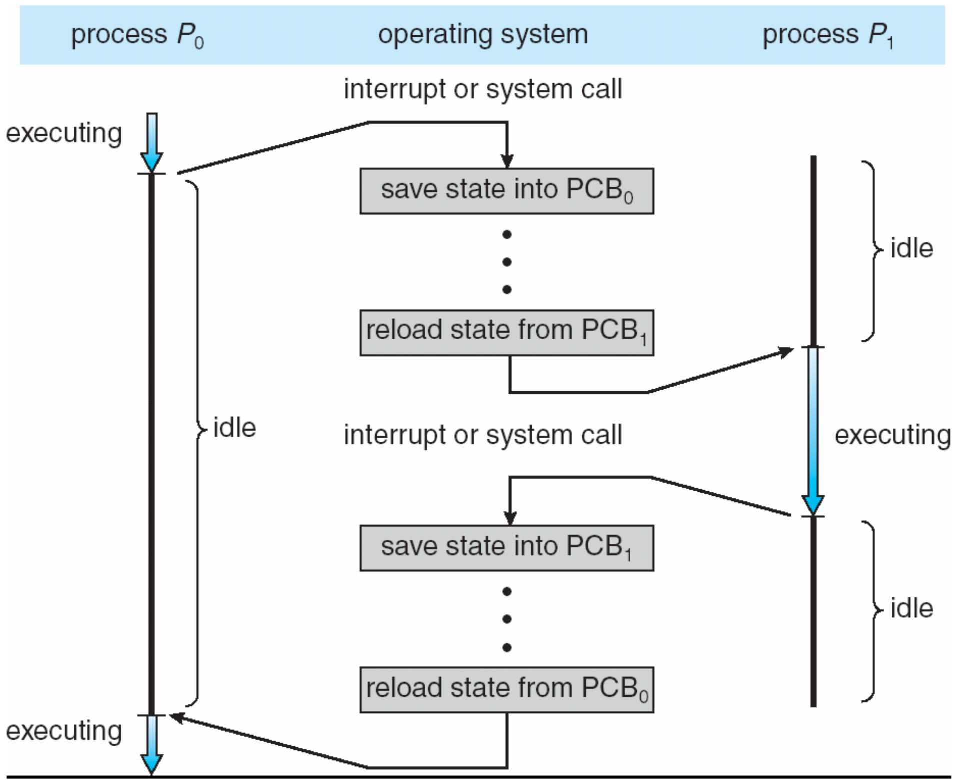


Fig. Context switching in RTOS

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

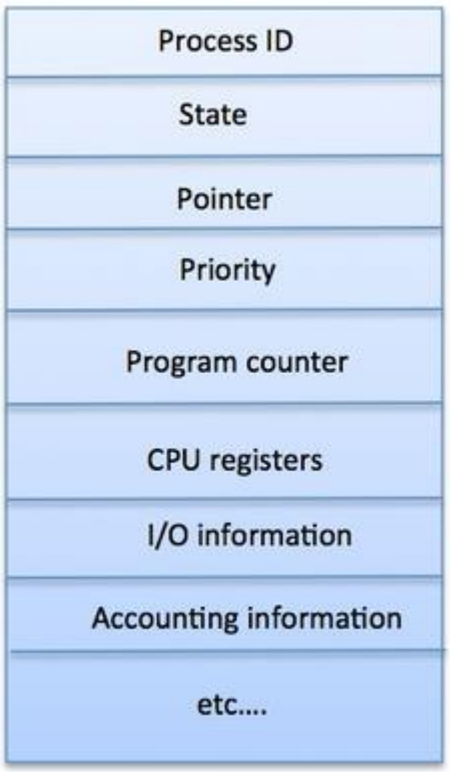
. CPU registers: Like Program Counter (CPU registers must be saved and restored when a process is swapped out and in of CPU)

. 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.

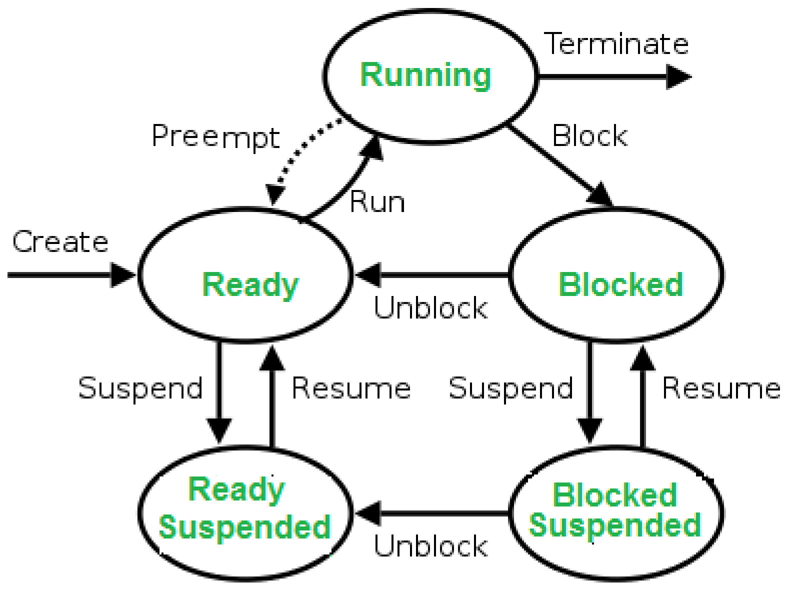


Fig.4 States of a process

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.

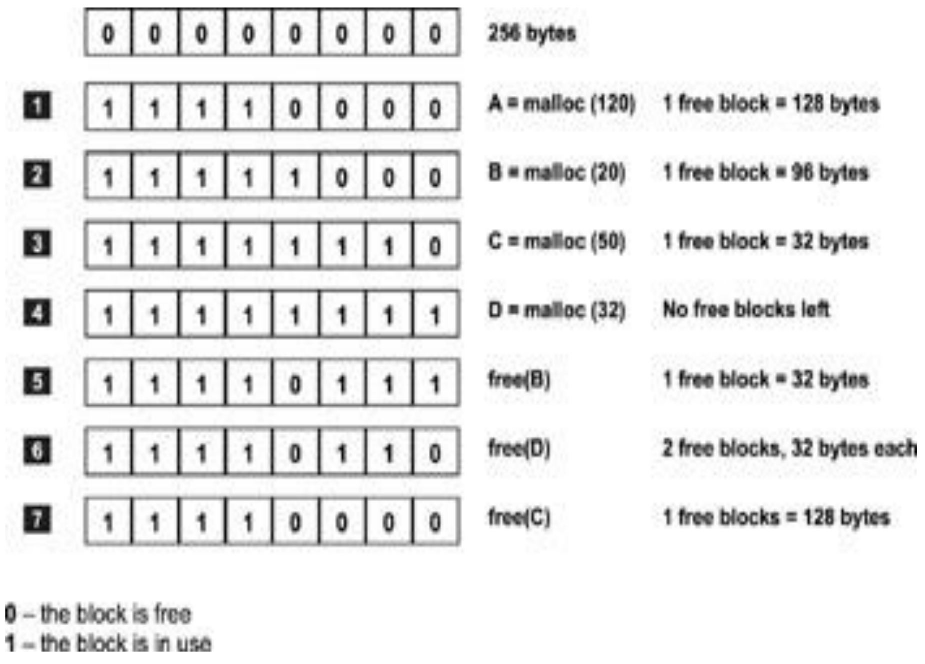
Memory Fragmentation and Compaction

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.



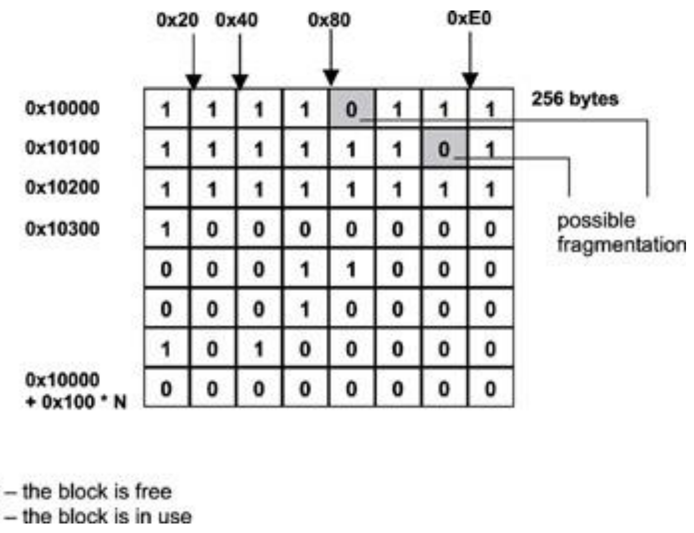
0 0 0 0 0 0 0 0 256 bytes
1 1 1 1 1 0 0 0 0 A = malloc (120) 1 free block = 128 bytes
2 1 1 1 1 1 0 0 0 B = malloc (20) 1 free block = 96 bytes
3 1 1 1 1 1 1 1 0 C = malloc (50) 1 free block = 32 bytes
4 1 1 1 1 1 1 1 1 D = malloc (32) No free blocks left
5 1 1 1 1 0 1 1 1 free(B) 1 free block = 32 bytes
6 1 1 1 1 0 1 1 0 free(D) 2 free blocks, 32 bytes each
7 1 1 1 1 0 0 0 0 free(C) 1 free blocks = 128 bytes
0 – the block is free
1 – the block is in use

Figure 5: States of a memory allocation map.

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 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.



0x10000	1	1	1	1	0	1	1	1
0x10100	1	1	1	1	1	1	0	1
0x10200	1	1	1	1	1	1	1	1
0x10300	1	0	0	0	0	0	0	0
	0	0	0	1	1	0	0	0
	0	0	0	1	0	0	0	0
	1	0	1	0	0	0	0	0
0x10000 + 0x100 * N	0	0	0	0	0	0	0	0

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.

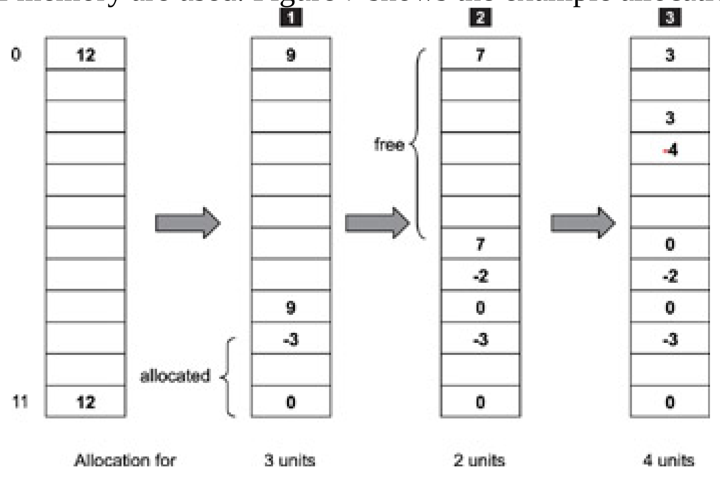


Figure 7: Static array implementation of the allocation map.

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.

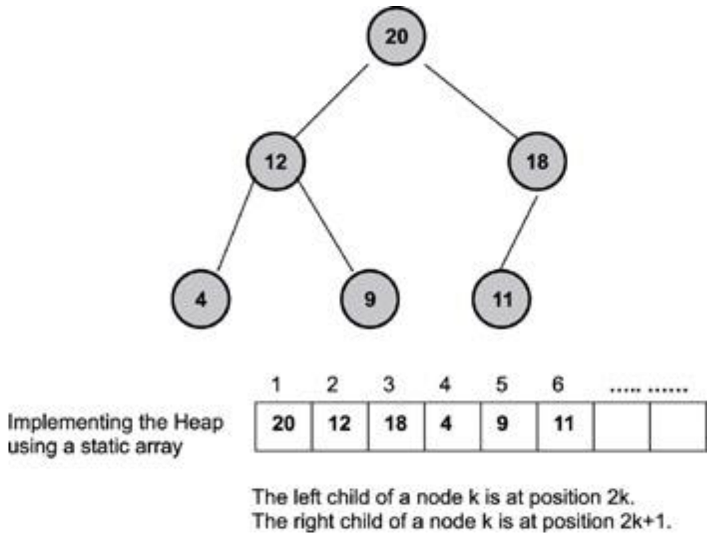

20
12
18
4
9
11
1 2 3 4 5 6 ....
Implementing the Heap using a static array
20 12 18 4 9 11
The left child of a node k is at position 2k.
The right child of a node k is at position 2k+1.

Figure 8: Free blocks in a heap arrangement.

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.

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.

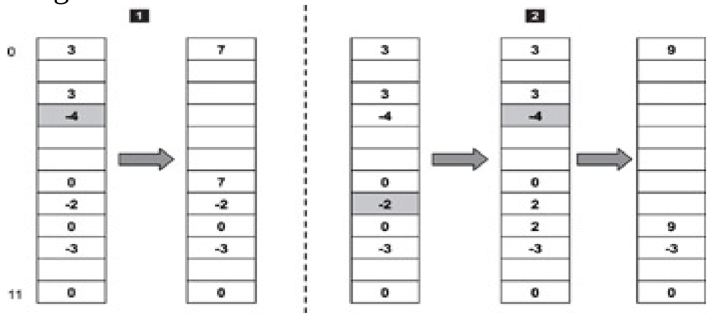


Figure 9: The free operation.

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.

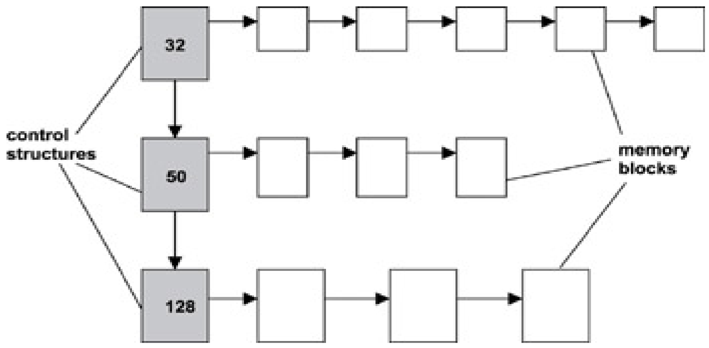


Figure 10: Management based on memory pools.

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.

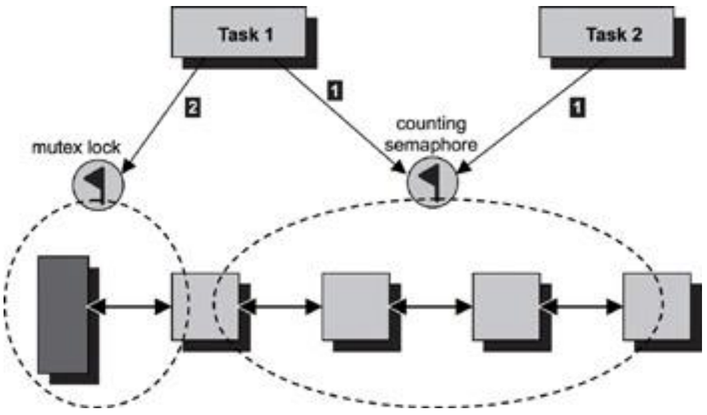


Figure 11: Implementing a blocking allocation function using a mutex and a counting semaphore.

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.

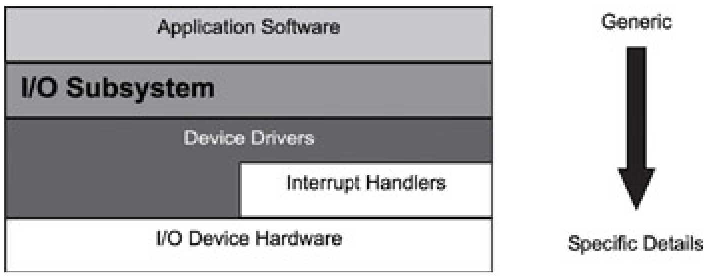


Figure 12: I/O subsystem and the layered model.

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.

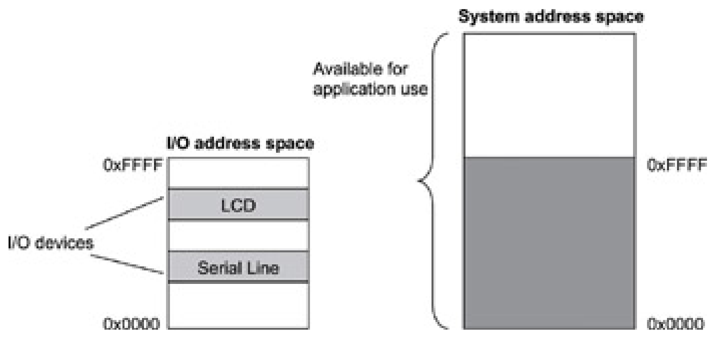


Figure 13: Port-mapped I/O.

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.

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.

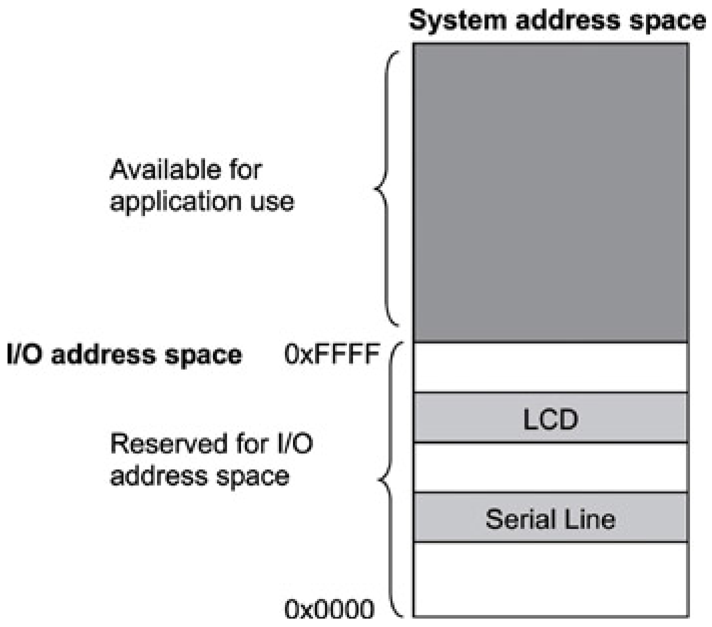


Figure 14: Memory-mapped I/O.

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.

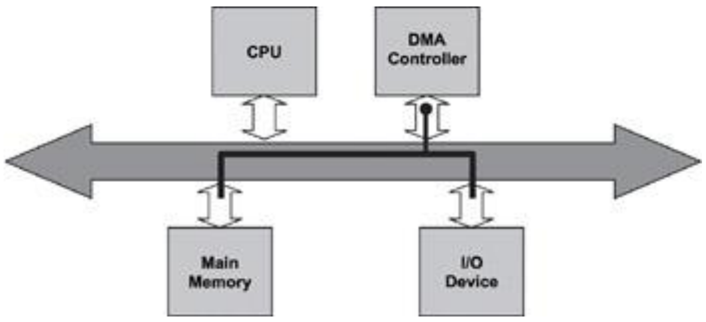


Figure 15: DMA I/O.

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.

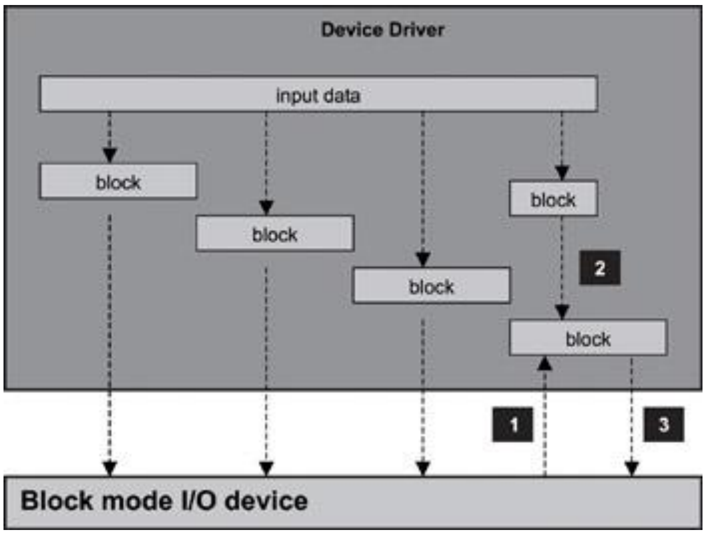


Figure 16: Servicing a write operation for a block-mode device.

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.

Each block is transferred to the device in separate write requests. The first three are straightforward write operations. The device driver must handle the last block differently from the first three because the last block has a different size. The method used to process this last block is device specific. In some cases, the driver pads the block to the required size. The example in Figure 16 is based on a hard-disk drive. In this case, the device driver first performs a read operation of the affected block and replaces the affected region of the block with the new data. The modified block is then written back.

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.

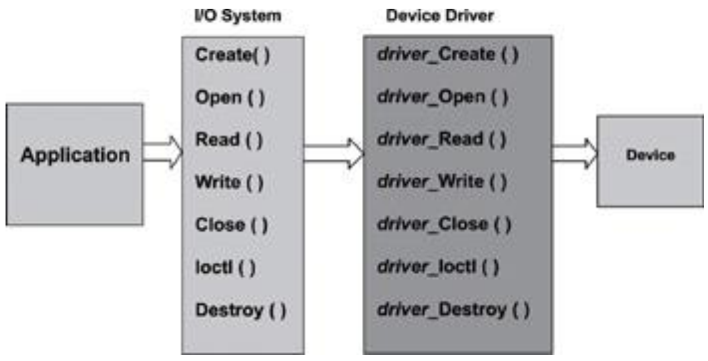


Figure 17: I/O function mapping.

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.

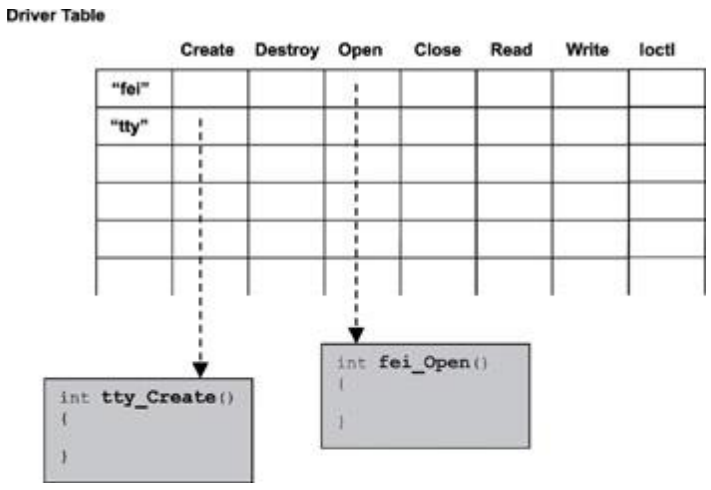


Figure 18: Uniform I/O driver table.

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.

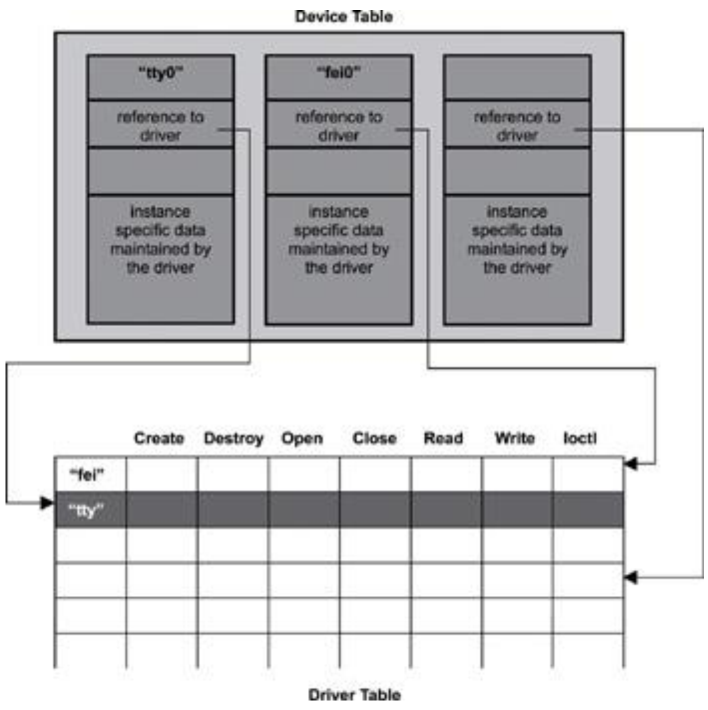

Device Table
"tty0"
"fei0"
reference to driver
reference to driver
reference to driver
instance specific data maintained by the driver
instance specific data maintained by the driver
instance specific data maintained by the driver
Create
Destroy
Open
Close
Read
Write
Ioctl
"fei"
"tty"
Driver Table

Figure 19: Associating devices with drivers.

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.

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.

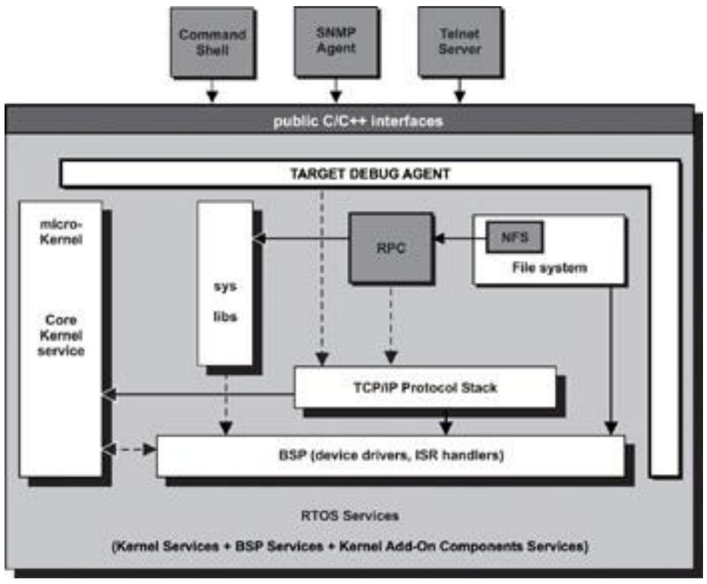


Figure 20: Overview.

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.

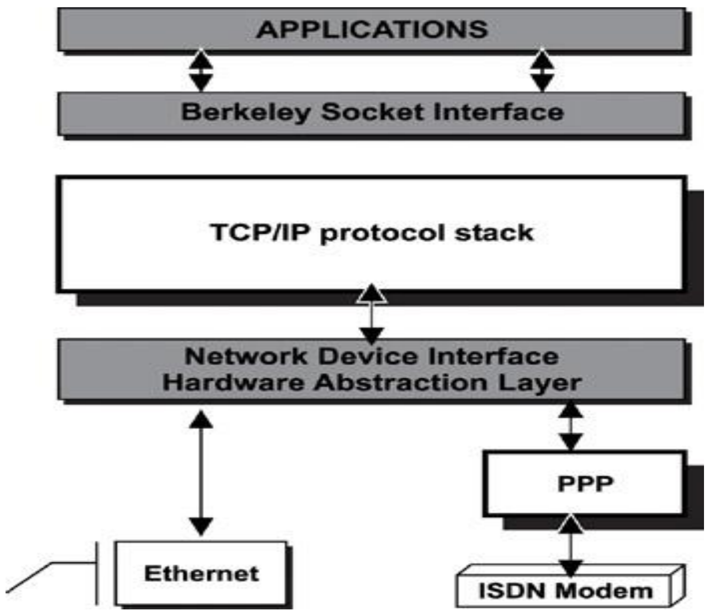


Figure 21: TCP/IP protocol stack component.

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.

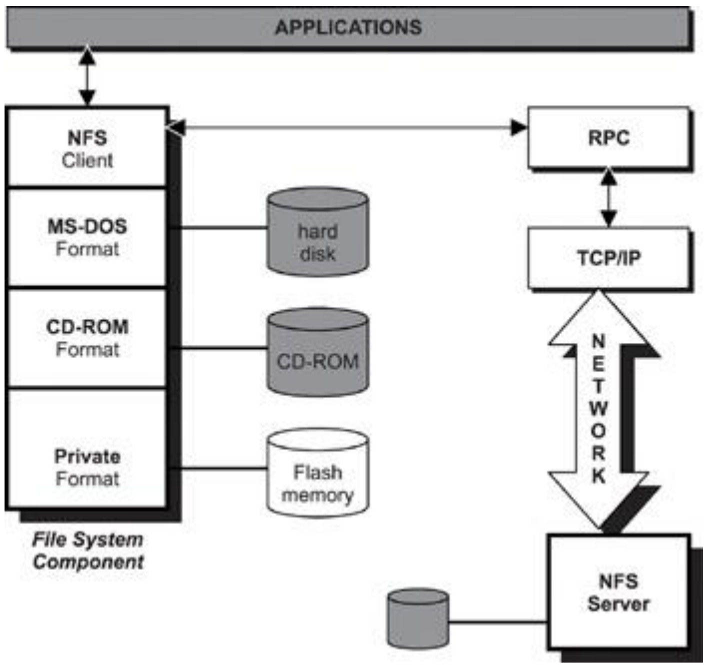


Figure 22: File system component.

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.

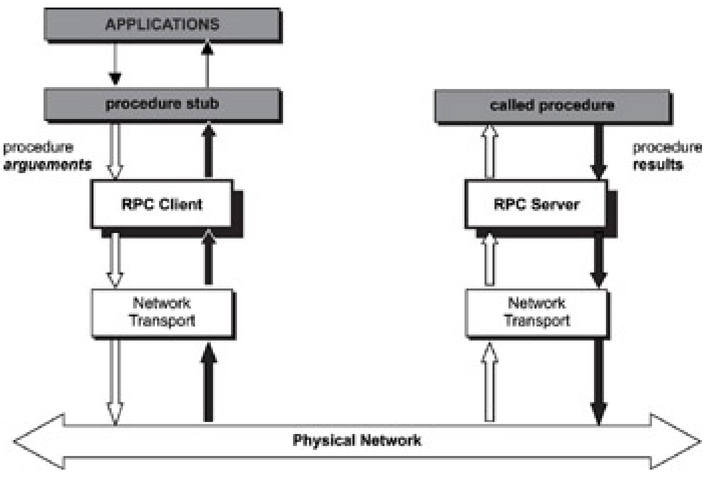


Figure 23: Remote procedure calls.

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.

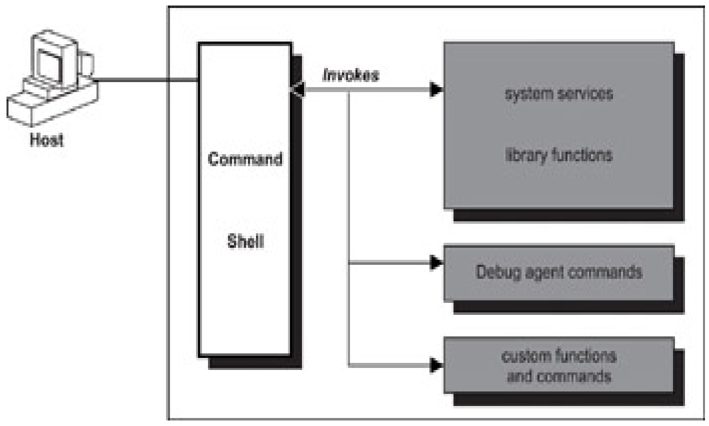


Figure 24: RTOS command shell.

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