Table 8.7 Set of Command Functions for Device Management

Commands	*Action(s)*
create and open	*create* is for creating and *open* is for creating (if not created earlier) and configuring and initializing the device.[1]
write	Write into the device buffer or send output from the device and advance the pointer (cursor).
read	Read from the device buffer or read input from the device and advance the pointer (cursor).
ioctl[2]	Specified device configured for specific functions and given specific parameters.
close and delete	*close* is for de-registering the device from the system and *delete* is for close (if not closed earlier) and detaching the device.

[1] There are two types of devices: char devices and block devices. (Refer to Table 4.2 for definitions.)
[2] 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 functions by passing as argument, the device-descriptor (a number), for example, fd or sfd in Sections 7.14 and 7.15 for a device control. Example is fd = 1 for read device, fd = 2 for write device.
2. *Second argument:* Defines the control option or use option for the IO device. Network devices control by defining baud rate or other parameters. Its use is as per function defined, as a second argument. Controlled device will be according to the first argument.
3. *Third argument:* Values needed by the defined function are at the third argument.

Example 8.7

Status = ioctl (fd, FIOBAUDRATE, 19,200) is an instruction in RTOS VxWorks. The fd is the device descriptor (an integer returned when the device is opened) and FIOBAUDRATE is a pointer for IO baud rate function that takes value of baud rate = 19,200 from third argument. This configures the device for operation at the 19,200-baud rate.

A device driver ISR uses several OS functions. Examples are as follows: *intlock* ( ) to disable interrupts system, intUnlock ( ) to enable interrupts. *intConnect* ( ) to connect a C function to an interrupt vector (the interrupt vector address for a device ISR points to a specific C function). Function intContext ( ) finds whether interrupt is called when an ISR was in execution.

UNIX OS makes it feasible for devices and files to 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 (Section 7.14). APIs and kernel interfaces in BSD (Berkley sockets for devices) UNIX are *open*, *close*, *read* and *write*. The following are the in-kernel commands: (i) *select*, which is to first check whether a read or write will succeed. (ii) *ioctl* to transfer driver-specific information to the device driver. [For example, baud rate in Example 8.7.] (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 char devices and block devices. Device driver functions may be similar to file functions, open, read, lseek, write and close.

8.6.2 File System Organization and Implementation

A file is a named entity on a magnetic disk, optical disk or system memory. A file contains the data, characters and texts. It may also have a mix of these. *Each OS may have differing abstractions of a file.* (i) A *file* may be a named entity that is a structured record as on a disk having random access in the system. (ii) A *file* may be a structured record on a RAM analogous to a disk and may also be either separately called *RAM disk* or simply, a 'file' (virtual device). (iii) A *file* may be an unstructured record of bits or bytes. (iv) A *file* device may be a pipe-like device.

It is necessary to organize the files in a systematic way and to have a set of command functions. Table 8.8 gives these functions for POSIX file system.

Table 8.8 Set of Command Functions in the Portable Operating System Interface (POSIX) File System

Command in POSIX	*Action(s)*
open	Function for creating the file
write	Writing the file
read	Reading the file
lseek (List seek) or set the file pointer	Setting the pointer for the appropriate place in the file for the next read or write
close	Closing the file

Notes: 1. File devices are block devices in Unix. Linux permits the use of a block device as a char device also. This is because between *block device* and *char device,* Linux has an additional interface. In other words, the kernel interface is identical for the char and block devices in Linux but not in Unix.

2. The file on the RAM that is hierarchically organized is known as RAM disk. RAM memory storage is analogous to that on the disk and accessing is also analogous to a disk. For example, path for accessing a file is directory, then subdirectory, then folder and then subfolder. There is hierarchical tree like the filing organization.
3. Unix has a structured file system with an unstructured hardware interface. Linux supports different standard file systems for the system.

Should a file having integers differ from a file having bytes? Should a file having bytes differ from a file having characters? Due to the differing approaches to device and file management interfaces, the development of a set of standard interfaces becomes must. Only then can systems be portable. A standard set of interfaces is called POSIX, from IEEE. POSIX stands for portable operating system interface standard for coding programs when using the multiple threads. The X after I is because of the interfaces being similar to the ones in Unix. It is according to the definitions at the AT & T UNIX System V Interface. POSIX defines the functions: open, close, read, write, lseek and fcntl. Function lseek is to move the pointer position in the byte stream. Function fcntl is for file control. The POSIX standard for file operations are as the operations on a linear sequence of bytes.

Window NT assumes a file as the named entity for a record of bytes placed sequentially and the OS has the command functions, createFile, ReadFile, WriteFile and SetFilePointer, and CloseHandle for creating a file, reading a file, writing a file and setting the file pointer from the present to a new location.

A file in Unix has *open* ( ), *close* ( ), *read* ( ), *write* ( ) functions analogous to a device, *open*, *close*, *read* and *write* functions. The BSD Unix interface differs slightly from Unix.

There are two types of file systems.

1. *Block file system.* Its application generates records to be saved into the memory. These are first structured into a suitable format and then translated into block streams. A file pointer (record) points to a block from the start to the end of the file.
2. *Byte stream file system.* Its application generates record streams. These streams are to be saved into the memory. These are first structured into a suitable format and then translated into byte streams. A file pointer (byte index) points to a byte from the start index = 0 to N–1 in a file of N bytes.

Just as each process has a processor descriptor (PCB); a file system has a data structure, called file descriptor (Table 8.9). The structure differs from one file manager to another. File descriptor, fd, for a file is an integer, which returns on opening a file. fd points to the data structure of the file. fd is usable till the closing of the file.

Table 8.9 Data Structure of File Descriptor in a Typical File System

File Descriptor	*Meaning(s)*
Identity	Name by which a file is identified in the *application*
Creator or owner	Process or program by which it was created
State	A state can be 'closed', 'archived' (saved), 'open executing file' or 'open file for additions'
Locks and protection fields	O_RDWR file opens with read and write permissions, O_RDONLY file opens with read only permissions, O_WRONLY file opens with write only permissions
File info	Current length, when created, when last modified, when last accessed
Sharing permission	Can be shared for execution, reading or writing
Count	Number of directories referring to it
Storing media details	Blocks transferable per access

A file manager creates, opens, reads, seeks a record, writes and closes a file. A file has a file descriptor.

8.6.3 I/O Subsystems

I/O ports are the subsystems of OS device management systems. Drivers communicate with the many devices that use them. I/O instructions depend on the hardware platform. I/O systems differ in different OSes. Subsystems of a typical IO system are as given in Table 8.10.

There are two types of IO operations–synchronous and asynchronous. There may be separate functions for synchronous and asynchronous operations in an RTOS. In case of traditional OS, only synchronous IOs may be supported.

Synchronous IO operations are at certain fixed data transfer rates. Therefore, a task (process) blocks tills till completion of the IO. For example, a write function, write ( ) for 1 kB data transfer to a buffer. Synchronous IO operation means once synchronous IO initiates, the data transfer will block the task till 1 kB data gets transferred to the buffer. Similarly, read ( ) once initiated blocks the task till 1 kB is read.

Table 8.10 Input/Output (I/O) Subsystem in a Typical I/O System in an Operating System (OS)

Subsystems Hierarchy	*Action(s) and Layers between the Subsystems*
Application	An application having an I/O system. There may also be a sublayer between the application and I/O basic functions
IO basic functions	These are device-independent OS functions, for example, file system functions for read and write, buffered IO or file (block) read and write functions. There may also be a sublayer between the basic I/O functions and I/O device driver functions
IO device driver functions	These are device-dependent OS functions. A driver may interface with a set of library functions, for example, for serial communication
Device hardware or port or IO interface card	Serial device or network

Asynchronous IO operations are at the variable data transfer rates. It provisions for a process of high priority not blocked during the IOs.

Example 8.8

POSIX has the following asynchronous functions: *aio_read* ( ) and *aio_write* for the asynchronous read and write in an I/O system. Therefore, an aio_read ( ) and aio_write ( ) do not block the task till completion of the IO. *aio_list* ( ) is to initiate a list of certain maximum asynchronous I/O port requests. *aio_error* ( ), *aio_cancel*, *aio_suspend* are functions for asynchronous IO error status retrieval and for cancelling and suspending I/O operations, respectively. Suspension is till the next port device interruption or till a timed out aio_return returns the status of completed operations.

I/O subsystems are an important part of OS services. Examples are the UART access and the parallel port access. There are synchronous and asynchronous IOs. A task gets blocked during the synchronous IOs, for example, fread ( ) or write ( ) (Section 7.14). RTOSes support asynchronous IOs, for example, *aio_read* ( ) and *aio_write* also in order to not to block a task during the IOs.

8.7 INTERRUPT ROUTINES IN RTOS ENVIRONMENT AND HANDLING OF INTERRUPT SOURCE CALLS

In a system, the ISRs should function as following.

1. ISRs have higher priorities over the OS functions and the application tasks. An ISR does not wait for a semaphore, mailbox message or queue message (Sections 7.11 to 7.13).
2. An ISR does not also wait for mutex (Sections 7.7.3 and 7.8.3) else it has to wait for other critical section code to finish before the critical codes in the ISR can run. Only the accept function for these events can be used (row 7 Table 7.1 and Section 7.11).

There are three alternative systems for the OSes to respond to the hardware source calls from the interrupts. Figure 8.1(a–c) show the three systems. The following sections explain the *three alternative systems in three OSes for responding to a hardware source call on interrupts.*

8.7.1 Direct Call to an ISR by an Interrupting Source and ISR Sending an ISR Enter Message

Figure 8.1(a) shows the steps. On an interrupt, the process running at the CPU is interrupted and the ISR corresponding to that source starts executing (step 1). A hardware source calls an ISR directly. The ISR just sends an ISR enter message to the OS (step 2).

OS is simply sent an ISR enter message (ISM) from the ISR in step 2. Later the ISR code can send into a mailbox or message queue (step 3) but the task waiting for the mailbox or message queue does not start before the return from the ISR (step 4). The ISR enter message in step 2 is to inform the OS that an ISR has taken control of the CPU. The ISR continues execution of the codes needed for the interrupt service till the ISR exit message is sent just before the return (step 4) .

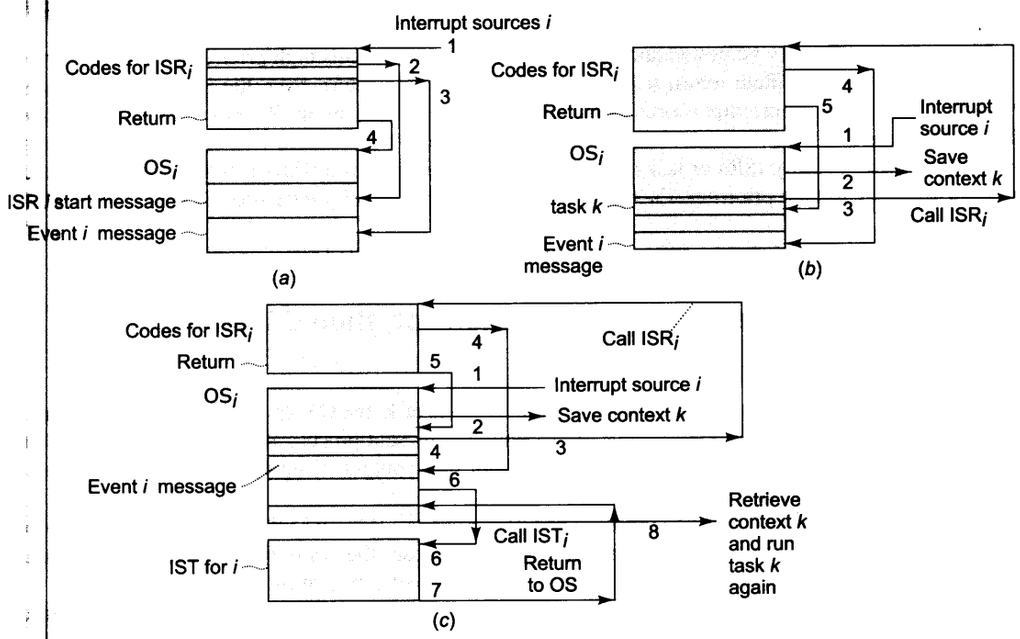

Fig. 8.1 (*a*) – (*c*) Three alternative systems in three real-time operating systems for responding to a hardware source call on interrupts

There are two functions, ISR and OS functions, in two memory blocks. An *i*-th interrupt source causes *i*-th ISR, ISR_*i*. to execute. The routine sends an ISR enter message to the OS. The message is stored at the memory allotted for OS messages. When the ISR finishes, it sends ISR *exit* to the OS and there is return and either there is the execution of interrupted process (task) or rescheduling of the processes (tasks). OS action depends on the event messages, whether the task waiting for the event is a task of higher priority than the interrupted task at the interrupt.

On certain OS, there may be a function OSISRSemPost ( ). The ISR semaphore is a special semaphore, which OSISRSemPost ( ) posts and on return from the OS to be taken by the calling ISR itself. OS ensures that OSISRSemPost executing ISR is returned after any system call from the ISR.

Example 8.9

Consider the RTOS μCOS-II. Assume that a microcontroller has a hardware timer, which is programmed to interrupt every 10 ms. The microcontroller on timer interrupt calls and PC changes to an ISR vector address, ISR_Timer_Addr. At ISR_Timer_Addr, there is a routine ISR_Timer for servicing the timer interrupt. ISR_Timer first executes OSIntEnter ( ) just after the start of ISR_Timer is called.
ISR_Timer then executes OSIntExit ( ) before the return code.
The OSIntEnter ( ) sends the message to the RTOS that there should be context-switch and return from the ISR only after any system call is made by the ISR or until the OSIntExit ( ) executes in the ISR code. Any task waiting for the post of semaphore or mailbox message or queue message should not start on execution of the post function within the ISR or in any other task or ISR. RTOS schedules that later on return from ISR.

The multiple ISRs may be nested and each ISR of low priority sends high priority ISR interrupt message (ISM) to the OS to facilitate return to it on the completion and return from the higher priority interrupt. Nesting means when an interrupt source call of higher priority, for example, system real-time clock interrupt (SysClkIntr) occurs, then the control is passed to higher priority SysClkIntr and on return from the higher priority the lower priority ISRs or tasks starts executing. The number of ISRs can be nested with execution order in sequence to their priorities. Each ISR on letting a higher priority interrupt call sends the ISM (step 4) to the RTOS.

There is common stack for the ISR nested calls, similar to the nested function calls (Table 7.1).

8.7.2 RTOS First Interrupting on an Interrupt, then OS Calling the Corresponding ISR

Figure 8.1(b) shows the steps. On interrupt of a task, say, *k*-th task, the OS first gets the hardware source call (step 1) and initiates the corresponding ISR after saving the present process status (or context) (step 2). The called ISR (step 3) during execution then can post one or more outputs (step 4) for the events and messages into the mailboxes or queues.

Assume that there are the routine (*i*-th ISR) and two processes (OS and *j*-th task) in three memory blocks other than the interrupted *k*-th task. An *i*-th interrupt source causes the OS to get the notice of that, then after step 1 it finishes the critical code till the pre-emption point and calls the *i*-th ISR. ISR_*i* executes (step 3) after saving the context (step 2) onto a stack. The preemption point is the last instruction of the critical part of the presently running OS function, after which the ISR being of highest priority is called. The ISR in step 4 can post the event or mailbox message(s) to the OS for initiating the *j*-th task or *k*-th task after the return (step 5) from the ISR and after retrieving the *j*-th or *k*-th task context.

The events or mailbox messages are stored at the memory allotted for OS messages. The OS initiates the *j*-th task (if is of higher priority than the interrupted task *k*) or runs the interrupted task *k*.

The ISR must be short and it must simply post the messages for another task. This task runs the remaining codes whenever it is scheduled (according to priorities). OS schedules only the tasks (processes) and switches the contexts between the tasks only. ISR executes only during a temporary suspension of a task.

OS may provide for nesting or an OS may provide for the ISRs such that the OS initiates running of the ISR calls from a priority ordered FIFO (Table 7.1).

The system priorities are ISRs and then tasks (or ISTs). IST is just a task initiated on signal or message from an ISR (for example, task *j* in above example).

Example 8.10

Each device event has the codes for an ISR, which executes only on scheduling by the RTOS and provided an interrupt is pending for its service. Consider mobile PDA device example (Section 1.10.6). The 5 steps for the interrupt servicing by first interrupting the RTOS process are as follows:

Assume that using RTOS, touch screen ISR, ISR_TouchScreen has been created using a function OS_ISR_Create ( ). The ISR can share the memory heap with other ISRs. A function, IntConnect connects the touch screen event with the event identifier in an interrupt handler, ISR_handler.

Let a touch screen event occur, which means the user of the mobile device taps the screen at a select icon or menu (step 1). After saving context of current process (step 2) the OS sends the signal on behalf of the ISR_handler to the initiate ISR_TouchScreen (step 3). An interrupt service thread or a Task TouchScreen Input IST_TouchScreen waits using a function OS_eventPend ( ) for message (an object, such as semaphore, mailbox or queue message) (stpe 4) from the ISR_TouchScreen. The IST executes as per its priority Task or IST_TouchScreenPriority among the other pending ISTs or tasks before it starts executing.

Before return from the ISR_TouchScreen, it sends a message to the kernel using a OS_ISR_Exit ( ) just before the end of the codes in the ISR_TouchScreen (step 5). The ISR_TouchScreen can be restarted on the next interrupt event and gets ready for the next hardware event of tap on the screen.

8.7.3 RTOS First Interrupting on an Interrupt, then RTOS Initiating the ISR and then an ISR

An RTOS can provide for two levels of ISRs, a fast-level ISR, FLISR and a slow-level ISR (SLISR). The FLISR can also be called hardware interrupt ISR and the SLISR as software interrupt ISR. FLISR is called just the ISR in RTOS Windows CE. The SLISR is called interrupt service thread (IST) in Windows CE. The use of FLISR reduces the interrupt latency (waiting period) for an interrupt service and jitter (worst-case and best-case latencies difference) for an interrupt service.

An IST functions as a deferred procedure call (DPC) of the ISR. An *i*-th IST is a thread to service an *i*-th interrupt source call.

Figure 8.1(c) shows seven steps on the interrupt. On interrupt, the RTOS first gets the hardware source call (step 1) and initiates the corresponding ISR after finishing the critical section and reaching the pre-emption point and then saving the processor status (or context) (step 2). The ISR executes the device- and platform-dependent code (step 3). The ISR at the start can mask (disable) further pre-emption from the same or other hardware sources. The ISR during execution then can send one or more outputs for the events and messages into the mailboxes or queues for the ISTs (step 4). The IST executes the device- and platform-independent code. The ISR just before the end, unmasks (enable) further pre-emption from the same or other hardware sources (step 5).

There are the ISRs and number of ISTs, RTOS and tasks in the memory blocks other than the interrupted task. Any interrupt source causes the RTOS to get the notice of that, then completes the critical code till the pre-emption point and calls the ISR. ISR executes after saving the context onto a stack. The ISR can post message(s) into the FIFO for the IST(s) after recognizing the interrupt source and its priority. The ISTs in the FIFO that have received the messages from the ISR(s) executes (step 6) as per their priorities on return (step 5) from the ISR. The ISR has the highest priority and pre-empts all pending ISTs and tasks.

When no ISR or IST is pending execution in the FIFO, the interrupted task runs on return (step 7).

The ISRs must be short, run critical and necessary codes only, and then they must simply send the initiate call or messages to ISTs into the FIFO. It is the IST, which runs the remaining codes as per the priority-based schedule. The system priorities are in order of ISRs, ISTs and tasks. The ISTs are SLISRs running device-independent codes as per the device priorities on signals (SWIs) from the ISRs.

The ISTs run in the kernel space. The ISTs do not lead to priority inversion and have the priority inheritance mechanism.

RTOS schedules the ISTs and tasks (processes) and switches the contexts between the ISTs and tasks.

Example 8.11

Consider Mac OS X. The Mac OS X is RTOS for the mobile device, for example, iPod. An interrupt handler first receives the primary interrupt and then it generates a software interrupt known as a secondary interrupt. The secondary software interrupt is sent to initiate an IST.

The OS does not receive the actual interrupt but the low level process intercepts the interrupt. It calls a low-level (hardware level) ISR, LISR. It resets the pending interrupt bit in the device interrupt controller and calls a device-specific ISR, say, $DISR_i$. The $DISR_i$ posts a message to an IST_i specific to the device. The message notifies to the IST_i that an interrupt has occurred, and then the $DISR_i$ returns to LISR. LISR resets another pending interrupt bit in the device interrupt controller and calls the another device-specific ISR, say, $DISR_j$.

When no further interrupts are pending, the OS control returns to the currently executing thread, which was interrupted and when the OS passed control to the LISR.

The IST_i are scheduled by the OS, the IST_i finds that the SWI has occurred, it starts and run the codes. ISTs run as if a thread is running.

An RTOS uses one of the three strategies on interrupt source calls: (i) an ISR servicing directly after merely informing the RTOS at the start of ISR; (ii) kernel intercepting the call and calling the corresponding ISRs and tasks. RTOS kernel schedules only the tasks (processes) and ISR executes only during a temporary suspension of the task by the RTOS; (iii) kernel intercepting the call and calling the ISR, which initiates and queues the ISR calls into a priority FIFO. The ISR signals the SWIs for the ISTs. The RTOS kernel schedules the ISTs as priority queue and then tasks processes as per the priority queue.

8.8 REAL-TIME OPERATING SYSTEMS

An RTOS is multitasking OS for the applications needing meeting of time deadlines and functioning in real-time constraints. Real-time constraint means constraint on time interval between occurence of an event and system-expected response to the event.

RTOS-software has the OS services listed in Table 8.11. These enable design of software for a large number of embedded systems.

An RTOS is an OS for response time-controlled and event-controlled processes. The processes have predictable latencies. An RTOS is an OS for the systems having the real timing constraints and deadlines on the tasks, ISTs and ISRs.

Table 8.11 Real-Time Operating System (RTOS) Services

Function	*Activities*
Basic OS functions	Process management, resources management, device management, I/O devices subsystems and network devices and subsystems management.
Process priorities management: priority allocation	User-level priorities allocation, called static priority allocation or real-time priority allocation is permitted. The real-time priorities are higher than the dynamically allocated priorities to the OS functions and the idle priority allotted to low priority threads. The idle priority thread or task is one which runs when no other high priority ones are running.
Process management: preemption	The RTOS kernel preempts a lower priority process when a message or event for which it is waiting to run a higher priority process takes place. The RTOS kernel has the preemption points at the end of the critical code and therefore the RTOS can be preempted at those points by a real-time high priority task. Only small sections in the RTOS functions are non-preemptive.
Process priorities management: priority inheritance	Priorities inheritance enables a shared resource in low priority task, for example, LCD display, be used by high priority task first. An intermediate priority task will not pre-empt the low priority task when it is locked to run the critical shared resource or code for the high priority task (Section 7.8). Priority sealing in place of priority inheritance option can also be used for a specific system.
Process predictability	A predictable timing behaviour of the system and a predictable task synchronization with minimum jitter (difference between best-case and worst-case latencies).
Memory management: protection	In RTOS threads of application program can run in kernel space. The real-time performance becomes high. However, then a thread can access the kernel codes, stack and data memory space, and this could lead to unprotected kernel code.
Memory management: MMU	Memory management is by either disabling the use of MMU and virtual memory or by using memory locks. Memory locking stops the page swapping between the physical memory and disk when MMU is disabled. This makes RTOS task latencies predictable and reduces jitter (time between worst-case and best-case latencies for a task or thread).
Memory allocation	In RTOS, the memory allocation is fast when there are fixed length memory block allocations. First, speed of allocation is important (Example 8.6).
RTOS scheduling and interrupt latency control functions	Real-time task scheduling and interrupt latency control (Section 4.6) and use of timers (Sections 3.6, 3.7 and 3.8) and system clocks.
Timer functions and time management	Provides for timer functions. There is time allocation and de-allocation to attain efficiency in given timing constraints.
Asynchronous IO functions	Permits asynchronous IOs, which means IOs without blocking a task.
IPC synchronization functions	Synchronization of tasks with IPCs (semaphores, mailboxes, message queues, pipes, sockets and RPCs).
Spin locks	Spin locks for critical section handling (Section 8.10.4).
Time slicing	Time slicing of the execution of processes which have equal priority.
Hard and soft real-time operability	Hard real-time and soft real-time operations (Section 8.9.3)

8.9 BASIC DESIGN USING AN RTOS

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, kernel function or task (Sections 8.1.2 and 8.7). An RTOS use in embedded system facilitates the following.

1. An RTOS provides running the user threads in kernel space so that they execute fast (Example 8.1).
2. An RTOS provides effective handling of the ISRs, device drivers, ISTs, tasks or threads (Section 8.7) and the disabling and enabling of interrupts in the user mode critical section codes. A critical section means a section of codes or a resource or codes that must run without blocking. One critical situation is when there is a shared data or resource with the other routines or tasks. RTOS provides for effective handling of such a situation.
3. An RTOS provides memory allocation and de-allocation functions in fixed time and blocks of memory (Example 8.6) and restricting the memory accesses only for the stack and other critical memory blocks (Section 8.5).
4. An RTOS provides for effectively scheduling and running and blocking of the tasks in cases of many tasks (Section 8.10).
5. I/O management with devices, files, mailboxes, pipes and sockets becomes simple using an RTOS. (Sections 7.9 and 8.6) RTOS provides for the use of message queues and mailbox, pipes, sockets and other IPC functions (Sections 7.9 to 7.15). RTOS provides for the use of semaphore(s) by tasks or for the shared resources (critical sections) in a task or OS functions (Section 7.7.2).
6. Effective management of the multiple states of the CPU and internal and external physical or virtual devices. Assume that the following actions are concurrently needed in an application. (i) Physical devices timer, UART and keyboard have issued the interrupts and the service routines are to be executed. (ii) A file is taken as a virtual device. The file also must be opened with its pointer to its first record. (iii) A physical timer is to configure its control register. (iv) Another timer gets a count input from the system clock. (v) A virtual device, a file, gets the inputs for writing onto it. (vi) A timer states changes on timeout and generates a need for its service. (vii) A file states changes on transfer of all needed records to it. (viii) A timer executes a service routine on timeout. (ix) A file needs execution of a function, close ( ). By effectively using a common method to handle these needs, the RTOS solves all the problems.

Basic design principles in RTOS environment are as follows.

8.9.1 Principles

Following are the design principles when using an RTOS to design an embedded system.

Design with the ISRs and Tasks *The embedded system hardware source call generates interrupts*. On interrupt, if the interrupt is not masked (disabled) the interrupt saves the current process (a task or thread or OS function) context on a stack and executes the ISR corresponding to tat interrpt. The handling of interrupt source calls is as described in Section 8.7. It is done by an RTOS by one of the three methods by a given RTOS environment. Interrupts are masked by disable interrupt command and unmasked by enable interrupt commands.

The ISR can only post (send) the messages for the RTOS and parameters for the tasks. No ISR instruction should block any task. Therefore, the ISR should not use mutex locks and should not use OS pending functions for the IPCs. Only an RTOS initiates the actions according to the ISR-posted signals, semaphores, queues, mailboxes and pipes (Section 7.9) and the RTOS control states of the tasks and interactions with the tasks. The

variables and task-switching flags must always be under the RTOS control. *No ISR instruction should wait for taking the messages.* The ISR should execute the codes that should not wait for actions by the RTOS and tasks.

RTOS provides for nesting of ISRs. This means that a running ISR can be interrupted by a higher priority interrupt and the higher priority ISR starts executing, blocking the running of low priority ISR. When the high priority interrupt service completes and there is return to the low priority interrupt after retrieving the saved context from the stack for the low priority interrupt.

A task can 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 or task has to be under the control of the RTOS. Such an attempt should generate an error.

Each ISR Design Consisting of Shorter Code As ISRs have higher priorities over the tasks, the ISR code should be made short so that the tasks do not 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 the message or parameters for that. The frequent posting of the messages by the IPC functions from the ISRs should be avoided.

When there are frequent interrupts from the same source, then the messages can be first put in the buffer on each interrupt and when the buffer is sufficiently filled, the IPC message can be posted for ready buffer. This is because if the buffer is not used and the IPC messages are posted frequently by making system calls, OSMsgQPost ( ) function, there will be frequent context switches and hence wastage of time. Example 8.12 shows how the time is saved from frequent context switches.

Example 8.12

Consider ACVM (Section 1.5.2). Consider the task, *Task User Keypad Input* (Example 7.3). When a user presses a key on the AVCM, it generates a hardware device interrupt. Assume that the pressed key is read by taking the ASCII code for the pressed key by the ISR_KeyInputDevice. One option is sending the ASCII code as the queue message for *Task User Keypad Input*. Every time the key is pressed the ISR_KeyInputDevice posts the ASCII code into the message queue. *Task User Keypad Input* is the task, which has to interpret the user key entries. When there is return from the ISR_KeyInputDevice the RTOS ends the wait of the message into the queue and *Task User Keypad Input* gets the ASCII code. However, till such time the user entry is complete, there is no further action and *Task User Keypad Input* again enters the wait for taking the message from the queue.

Second option is that the ISR_KeyInputDevice sends the ASCII code into a key buffer, *KBuffer*. On each interrupt, this is done as long as the user key entry is for the *enter* key. On the *enter* key, the ISR_KeyInputDevice posts a semaphore, semKB to the *Task User Keypad Input*. The *Task User Keypad Input* takes the semaphore and reads the KBuffer and takes appropriate action as per the user entries of the keys. Second option saves the time spent in the first option in frequent context switches.

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 (Section 8.7.3 and Example 8.11). The 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 ISTs.

Design Each Task with an Infinite Loop from Start (Idle State) up to Finish (Last State) 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 runs or takes the IPC for which it is waiting, runs from the point where it was blocked or preempted. In pre-emptive scheduler, the high priority task can be delayed for some period to let the low priority task execute.

Example 8.13

Consider ACVM (Section 1.5.2). Example 7.3 showed that ACVM (Section 1.10.2) system can be divided into: (i) *Task User Keypad Input*, (ii) *Task Read-Amount*, (iii) *Chocolate delivery task* (iv) *Display Task*, (v) GUI_Task and (vi) *Communication task*. Let us use the five semaphores, which are posted from one task and taken up by another task. Five semaphores are semKB, SemRead, SemChocolate, SemDisplayThanks and SemDisplayThanks. A code design is as follows.

```
/*********************************************************/
Task User Keypad Input {
.
while (1) { /* Code for sending into key buffer ASCII codes on each run of ISR_Key Input Device */
OSSemPend (semKB); /* Example 8.12 showed use of SemKB */
/* Code for action as per key entries*/
.
OSSemPost (SemRead) /* Post a semaphore to let the Task read amount start */
.
  }
}
Task Read_Amount {
int cost; /* The cost of the chocolate selected by the user from user input key*/
while (1) {
OSSemPend (SemRead); /* Wait for message from Task User Keypad Input  */
/* Code for action as per reading the Coins, get the value of coins in variable amount */
.
If (amount > = cost) OSSemPost (SemChocolate) /* Post a semaphore to let the Task Chocolate delivery
start if amount is equal or more than the cost*/
OSSemPost (SemDisplayThanks) /* Post a semaphore to let the Task_Display show the message, Wait
for the nice chocolate*/
.
  }
}
Task Chocolate_Delivery {
.
while (1) {
OSSemPend (SemChocolate); /* Wait for message from Task Read_Amount */
/* Code for action for chocolate delivery */
.
OSSemPost (SemDisplayCollect) /* Post a semaphore to let the Task_Display show
the message, Thanks you, Collect the nice chocolate, Visit Again. */
.
  }
}
Task Display {
.
while (1) {
```

```
OSSemPend (SemDisplayThanks); /* Wait for message from Task Read_Amount */
/* Code for displaying the message, Thanks you, Collect the nice chocolate, Visit Again */
OSSemPend (SemDisplayCollect); /* Wait for message from Task Read_Amount */
/* Code for displaying the message, the message, Wait for the nice chocolate */
.
.
 }
}
/*********************************************************/
```

First the ISR_KeyInputDevice posts semKB, then Task User Keypad Input starts, which posts SemRead, then Task_ Read_Amount starts. Then it posts semaphore semChocolate after the user inserts coins for the appropriate amount. The task design is such that each code is in the waiting loop and waits for the message through the RTOS to start and run the task codes. ISR_KeyInputDevice initiates the chain of actions on the ACVM.

Assume that the priority assigned to the tasks are in the following order: *Task User Keypad Input*, then *Task Read-Amount*, then *Chocolate delivery task* then *Display Task*, then GUI_Task and then *Communication task*. As each a semaphore is being posted from one task in higher priority and taken only by another task of lower priority, we could have used the same semaphore, say, *sem0* for semKB, B SemRead and SemChocolate to synchronize the tasks. However, to encapsulate the semaphores between the tasks, we use five semaphores, semKB, SemRead, SemChocolate and SemDisplayThanks.

Design in the Form of Tasks for the Better and Predictable Response Time Control The RTOS provides the control over the response time of different tasks. The different tasks are assigned different priorities and those tasks which system needs to execute with faster response are separated out. For example, in a mobile phone device (Example 1.10.5) there is need for faster response to the phone call-receiving task then the user key input. In digital camera (Example 1.10.4), the task for recording the image needs faster response than the task for downloading the image on a computer through USB port.

Design in the Form of Tasks for Modular Design System of multiple tasks makes the design modular. The tasks provide modular design. For example, in a mobile phone device (Example 1.5.5) we consider the user key input and display as separate tasks. When the display size changes and new display hardware is introduced, only the codes for the display task and resource or data-sharing tasks and ISRs need to be modified. When a new functionality is introduced in the system, the user key input task and new functionality-associated tasks need to be modified.

Design in the Form of Tasks for Data Encapsulation System of multiple tasks encapsulates the code and data of one task from the other. In Example 8.13, the cost is encapsulated in TaskRead_Amount from other tasks and the messages such as 'Thank you, Collect the nice chocolate, Visit Again' encapsulate in the display task.

Design with Taking Care of the Time Spent in the System Calls The expected time in general depends on the specific target processor of the embedded system and the memory access times. However, in order to provide the relative magnitude of the time taken for basic actions at a preemptive scheduler, a new parameter is defined. It defines the time taken for an action by an RTOS scheduler in terms of an assumed scaling parameter, S. S emphasizes the relative magnitudes of execution times for various actions in a typical RTOS.

Let time taken for the simplest instruction be t_{min}. The minimum time is when the semaphores P and V are assigned certain initial values, true or false. Let S be defined in units of T_s. The T_s and t_{min} depend on a specific

target processor of the embedded system and the memory access times. For example, a typical value for a processor, t_{min} is 0.6 μs and let T_s = (4 μs + 0.6 μs). Let $t_{exec} = (t_{min} + S.T_s)$. This equation defines S as the time over and above t_{min} in units of a basic time unit, T_s.

If the same RTOS runs on a different processor, the S will therefore remain the same. S is taken as the nearest positive integer for the relative magnitude of execution times for the scheduler actions.

1. S = 1 will mean t_{exec} between 2.8 μs and 5.2 μs
2. S = 2 will mean 7.4 μs to 8.8 μs
3. S = 4 will mean 14.2 μs to 18.0 μs
4. S = 5 will mean 20.6 μs ± 3.0 μs
5. S = 10 will mean 40.6 μs ± 6.0 μs
6. S = 15 will mean 60.6 μs ± 8.0 μs

Table 8.12 gives a list of basic actions in a preemptive RTOS and execution time in terms of a scaling parameter S.

Table 8.12 A List of Basic Actions in a Preemptive RTOS and Execution Time in Terms of a Scaling Parameter *S*

Action	*S*	*Action*	*S*	*Action*	*S*	*Action*	*S*
Context Switch	2	Task suspend	1	Sem post/take	1	Message Q delete	10
Task initiate	12	Task resume	1	Take semaphore, mutex, counting semaphore when sem available	1	Q receive Message available	2
Task create and activate	28	Task Lock when no lock or Unlock when lock exists	t_{min}	Semaphore, mutex, counting semaphore create or delete	6	Q receive Message not available	1
Task delete	10	Mem allocate	2	Release semaphore, mutex, counting semaphore when in Q	3	Message Q Send task pending	5
Task switch flag for running	1	Mem free	4	Mutex flush	1	Message Q send task not pending	2
Task create or delete	18	Network byte send	t_{min}	Release semaphore, mutex, counting semaphore when no task in Q	1	Message Q Send queue full	1
		Semaphore flag or counting semaphore flush	4	Message Q create	105		

Abbreviations are as in brackets: Semaphore (Sem), Queue (Q) and Memory (Mem).

The RTOS create ( ) function to create a task takes longer CPU time than writing into a queue and then reading from the queue, and using a semaphore takes the least. Therefore, a design should create all the tasks at the beginning, even before the start of the tick of the system clock.

As queue takes longer time than the semaphore, *use semaphores if it suffices.* For example, consider the codes in Example 8.13. OSSemPost (SemDisplayCollect) is used in place of the queue for posting the message 'Thank you. Collect the nice chocolate, Visit Again.' for the display task. Getting a semaphore takes the least CPU time.

As signals take the least time among the IPCs, use signals for the most urgent IPCs (e.g., error reporting by throwing the exceptions).

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 We limit the number the number of tasks appropriately. The tasks, which share the data with number of tasks, can be designed as one single task. The examples of such tasks are: (i) display task for the messages from a number of tasks, (ii) printer task for the messages from a number of tasks and (iii) flash memory writing task for writing into the flash memory by the number of tasks.

Example 8.14

Consider a mobile phone device (Example 1.5.5). It needs to do the following tasks: *display time and date* as per the message posted by the other task, *display battery power* as per the message posted by the other task, *display signal strength* as per the message posted by the other task, *display device profile* (general or silent) as per the message posted by the user input task, *display messages* posted by the other task and *display call status* and *display menus.*

A design can be that which assume each display action as a separate thread. Another design can be a display task, which accepts messages from different tasks and displays. It will lead to creation of only one task for display and one TCB and stack for the display task. A single task for display provides better control of when, where and what to display.

Use Appropriate Precedence Assignment Strategy and Use Preemption in Place of Time Slicing The task of higher priority preempts the low priority tasks and the ISRs preempt the tasks. Therefore, an appropriate precedence is chosen.

The ISRs have higher priorities over the ISTs and tasks.

A mode of scheduling the tasks is assigning them equal priorities and allotting time slice for round robin mode (Section 8.10.2). Time slicing is done in certain specific cases, for example, in a network router when it is routing the packets of multiple clients.

Avoid Task Deletion Create tasks at start-up only and *avoid creating and deleting tasks later.* The only advantage of deleting is the availability of additional memory space. Suppose a task is deleted by an OSTaskDelete ( ) function. Now a situation can be that a task is waiting for a semaphore (to let other tasks finish the critical section) or is waiting for a queue (or mailbox) message for a pointer at the RTOS, and that pointer is for a message to the task that has been deleted. A prolonged blocking or a deadly embrace or a deadlock will then occur. An RTOS may not provide protection for these situations.

Certain RTOS provide an option to make a semaphore *deletion safe.*

Use Idle CPU Time for Internal Functions Often, the CPU may not be running any task. All tasks may be waiting for preemption (for transition from ready place to running place). The CPU at that instant may associate the RTOS for the following. Read the internal queue. Manage the memory. Search for a free block of memory. Delete or dispatch a task. Perform the internal and IPC functions.

Design with Memory Allocation and De-Allocation by the Task If memory allocation and de-allocation are done by the task the number of RTOS functions is reduced (Example 8.6, Section 8.5). This reduces interrupt latency periods as execution of these functions takes significant time by RTOS whenever the RTOS pre-empts 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 like a reentrant function* or should take care of problems from the shared resources or data such as buffer or global variables (Section 7.8). Disabling of running of other tasks for a longer period increases the worst-case interrupt latency periods for all the interrupts. While executing the critical section codes, if possible, *instead of disabling the interrupts only the task-switching flag changes should be used.* It is done by using semaphore. Thus, only the pre-emption by RTOS should be prevented. Disabling pre-emption may be better than disabling interrupts. However, both increase worst-case interrupt latencies.

Resource locking using the mutex semaphores or spin-locks may be better than disabling preemption or interrupts (Section 8.10.3). *A task should take the mutex semaphore only during a short period* in which the critical section alone is executed and shared resources (such as the display device driver) are being accessed. *Spin locks* can also be used in case critical section code is short and executes in time less than the IPC posting and context-switching time (Example 8.20, Section 8.10.3).

Design with Hierarchial and Scalable Limited RTOS Functions *Use an RTOS, which is hierarchial as well as scalable so that has only the needed functions are at the ported section of kernel with the rest left outside.* This is because the pre-emption scheduling increases the interrupt latency periods because of the time spent in context switching and saving and retrieving pointers for the RTOS functions like memory allocation, and IPCs (Table 8.12).

The functions for the memory management, file system functions, IPC (e.g., pipe, signal, socket and RPC) are provided outside the kernel in a hierarchical and scalable RTOS. MMU is disabled for predictive response time of the tasks (Section 8.8).

Hierarchical RTOS means the RTOS functions portable after extending and interfacing other functionalities and configuring for specific processor and set of devices. *Scalable RTOS* means portable into the system ROM image after *the limited RTOS functions in the kernel* space as per the application needs. For example, if queue and pipe functions are not required in an embedded system design, then these functions are not ported in scalable RTOS.

8.9.2 Encapsulation Using the Semaphores and Queues

Semaphores, queues and messages should not be globally shared variables, and each should be shared between a set of tasks only and encapsulated from the rest.

Semaphores A semaphore encapsulates the data during a critical section or encapsulates a buffer from a reading task or writing into the buffer by multiple tasks concurrently. Example 7.14 showed the use of a buffer by producing task (writing into the empty memory addresses buffer) and consuming task (reading from the filled spaces at the buffer). Example 8.15 gives another example.

Example 8.15

Assume that the *Task Read-Amount reads the amount and after delivering the chocolate the amount is reduced by a value equal to the cost. The semaphore will insulate the global variable amount.*

```
/*********************************************************/
static int cost; /* The cost of the chocolate selected by the user from user input key*/
static int amount; /* The amount variable */
/* Code for creating semaphore SemAmount with initial value = 1 */
Task Read_Amount {
.
while (1) {
OSSemPend (SemRead); /* Wait for message from Task User Keypad Input */
OSSemPend (SemAmount); /* Wait for semaphore from amount reducing section in task Chocolate Delivery
*/
/* Code for action as per reading the Coins, get the value of coins in variable amount */
If (amount > = cost)
OSSemPost (SemChocolate) /* Post a semaphore to let the Task Chocolate delivery start if amount is
equal or more than the cost*/
OSSemPost (SemAmount);. /* Post a semaphore to let the Task Chocolate_Delivery reduce the amount by
a value equal to the cost of chocolate after the delivery of the chocolate */
OSSemPost (SemDisplayThanks) /* Post a semaphore to let the Task_Display show the message, Wait
for the nice chocolate*/
  }
}
Task Chocolate_Delivery {
.
while(1) {
OSSemPend (SemChocolate); /* Wait for message from Task Read_Amount */
/* Code for action for chocolate delivery */
OSSemPend (SemAmount) /* Wait for amount ready from Task Read_Amount.*/
amount = amount – cost;
OSSemPost (SemAmount) /* Return the semaphore amount to Task Read_Amount.*/
OSSemPost (SemDisplayCollect) /* Post a semaphore to let the Task_Display show the
message, Thanks you, Collect the nice chocolate, Visit Again. */
.
  }
}
/*********************************************************/
```

The amount when read by the task read amount, is encapsulated from the task chocolate delivery using the mutex semaphore semAmount. The amount when reduced by the task chocolate delivery, is encapsulated using the semAmount.

Assume that semAmount semaphore is not used for encapsulation of the amount. Let us see the effect. If ISR_KeyInputDevice (Example 8.12) executes on a user input before the amount reduces at task chocolate delivery, the semKB will be posted and then taken by task read amount and it will pre-empt the task chocolate delivery. Even if the user does not insert any coin, the task will wrongly post semChocolate and SemDisplayCollect.

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 (Example 1.10.6). 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.

Example 8.16

Consider a mobile phone device (Example 1.10.5). Assume that the *Task ReadSMS and TaskSelected Menu posts into a queue to Task_Display.* The coding can be as follows.

```
/*********************************************************/
# define QMsgSize = 64 /*Assume that queue can be posted upto 64 pointer variables*/
# define QErrMsgSize = 16 /*Assume that up to 16 error messages can be posted from the queue*/
OS_Event * QMsgPointer /* Create a event control block */
void QMsgPointer [QMsgSize] /* Define a pointer to the queue */
OS_Event * QErrMsgPointer /* Create a event control block for the queue of errormessages*/
void *QErrMsgPointer [QErrMsgSize] /* Define a pointer to the queue */
QMsgPointer = OSQCreate (&QMsg [0], QMsgSize) ; /* Create a queue using call to RTOS function
OSQCreate */
QErrMsgPointer = OSQCreate (&QMsg [0], QErrMsgSize) ; /* Create a error messages queue using call
to RTOS function  OSQCreate */
Task_ ReadSMS { /* Code for the task which takes the SMS message as input */
.
while (1) {
OSQQuery (*QMsgPointer, QMsg);
If (QMsg  = = (void *) 0) {OSQPost (*QMsgPointer, &SMSMsg); } /* Q  contains no messages post
the SMS message into the queue*/
.  }
}
Task_Display { /* Code for the task which displays the messages */.
while (1) {

DispMsg = (void *) 0
while (QMsg  ! = (void *) 0) {
&DispMsg = OSQPend (*QMsgPointer, 0, &QErrMsg); /* Wait for the messages at the queue till all
messages read */
DispMsg ++;
.  }
}
```

The messages are posted into the queue after querying whether it has no messages and thus message pointer points to the null.

8.9.3 Hard Real-Time Considerations

Hard real time means strict adherence to each task deadline. When an event occurs, it should be serviced within the predictable time at all times in a given hard real-time system. The preemption period for the hard real-time

task in a worst case should be less than a few microseconds. A hard RTOS is one, which has predictable performance with no deadline miss, even in case of sporadic tasks (sudden bursts of occurrence of events requiring attention). Automobile engine control system and antilock brake are the examples of hard real-time systems.

Hard real-time systems provide for the following.

1. Disabling of all other interrupts of lower priority when running the hard real-time tasks.
2. Preemption of higher priority task by lower priority tasks.
3. Some critical code in the assembly to meet the real-time constraint (deadline) fast.
4. Task running in kernel space. This saves the time required to first check whether access is outside the memory space allocated to the kernel functions.
5. Provision of asynchronous IOs.
6. Provision of spin locks.
7. Predictions of interrupt latencies and context switching latencies of the tasks. This is achieved by writing all functions which on execution always take the same time intervals in case of varying rates of occurrences of the events.
8. Response in all the time slots for the given events in the system and thus providing the guaranteed task deadlines even in case of sporadic and aperiodic tasks. Sporadic tasks means tasks executed on the sudden bursts of the corresponding events at high rates, and aperiodic tasks mean tasks having no definite period of event occurrence.

A soft real time is one in which deadlines are mostly met. Soft real time means that only the precedence and sequence for the task operations are defined, interrupt latencies and context switching latencies are small but there can be a few deviations between expected latencies of the tasks and observed time constraints and a few deadline misses are accepted. The preemption period for the soft real-time task in a worst case may be about a few milliseconds. Mobile phone, digital cameras and orchestra-playing robots are examples of soft real-time systems.

8.9.4 Saving of Memory and Power

Methods of Saving and Optimizing the Memory Space

Following are the methods.

1. Use compressed data structure provided the de-compression algorithm plus compressed data structure combined together take less memory in the system compared with the case when only unpacked data structure is used.
2. Make the codes compact and fitted in small memory areas without affecting the code performance. This is called memory optimization. Code means code compiled and assembled executable in the given system. It also reduces the total number of CPU cycles, and thus, the total energy requirements.
3. Use declaration as unsigned byte, especially within the for and while loops, if there is a variable, which always has a value between 0 and 255. When using data structures, limit the maximum size of the queues, lists and stacks size to 256. Byte arithmetic takes less time than integer arithmetic.
   Follow a rule that uses unsigned bytes especially within the for and while loops *for a short integer if possible, to optimize use of the RAM and ROM available in the system. Avoid if possible the use of 'long' integers and 'double' precision floating point value bytes* especially within the for and while loops.
4. Avoid use of library functions if a simpler coding is possible. Library functions are the general functions. Use of general function needs more memory in several cases.
   Follow a rule that avoids use of library functions in case a generalized function is expected to take more memory especially when its coding is simple.

5. Configure the RTOS functions. For example, if queues are not needed the RTOS queue functions are not ported in the ROM image.
   Use a configurable, scalable, hierarchical RTOS which will help the ROM image to execute the needed functions at the kernel.
6. Optimize the RAM use for the stacks. It is done by three methods: (i) reducing the number of tasks that interact with the OS, (ii) reducing the number of nested calls and call at best one more function from a function (one function calling another function and that calling the third and so on means nested calls), (iii) optimize the number of tasks. (Less number of tasks are to be brought first into an initiated task list and there are the frequent interactions with the OS and context savings and retrievals stack on context switching, thus giving more memory and time overheads.) This optimizes the use of the stack.
   As a rule reduce the use of frequent function calls and nested calls and thus reduce the time and RAM memory needed for the stacks, respectively.
7. Optimize the allocation of stacks. A method is that allocated stack areas on allocation are filled with the specific bytes or specific set of bytes. Then find that in worst cases of running of the embedded system, how many filled bytes do not change. Then reduce the allocated stack spaces by rewriting the task, buffer and other memory creation codes.
8. In case the software design can be made fast with the instruction set of the target processor, the assembly codes be used. This also allows the efficient use of memory. The device-driver programs in the assembly especially provide efficiency due to the need to use the bit set–reset instructions for the control and status registers. Only a few assembly codes for using the device I/O port addresses, control and status registers are needed. The best use is made of available features for the given applications. Assembly coding also helps in coding for atomic operations. A modifier *register* can be used in the C program for fast access to a frequently used variable. If portAdata is frequently employed, it is used as follows, 'register unsigned byte portAdata'. The modifier register directs the compiler to place portAdata in a general purpose register of the processor.
   As a rule, use the assembly codes for simple functions like configuring the device control register, port addresses and bit manipulations if the instruction set is clearly understood. Use assembly codes for the atomic operations for increment and addition. Use modifier 'register' in C program for a frequently used variable.
9. Calling a function causes context saving on a memory stack and on return the context is retrieved. This involves time and can increase the worst-case interrupt latency. There is a modifier *inline*. When the *inline* modifier is used, the compiler inserts the actual codes at all the places where these operators are used. This reduces the time and stack overheads in the function call and return. But, this is at the cost of more ROM being needed for the codes. If used, it increases the size of the program but gives a faster speed. Using the modifier directs the compiler to put the codes for the function (in curly braces) instead of calling that function.
   As a rule, use inline modifiers for all frequently used small sets of codes in the function or the operator overloading functions if the ROM is available in the system. A vacant ROM memory is an unused resource. Why not use it for reducing the worst-case interrupt latencies by eliminating the time taken in the frequent save and retrieval of the program context?
10. When a variable is declared static, the processor accesses with less number of instructions than from the stack. As long as shared data problem does not arise, the use of static (global) variables can be optimized. These are not used as the arguments for passing the values. A good function is one that has no arguments to be passed. The passed values are saved on the stacks in case of interrupt service calls and other function calls. Besides obviating the need for repeated declarations, the use of global variables

will thus reduce the worst-case interrupt latency and the time and stack overheads in the function call and return. But this is at the cost of the codes for eliminating shared data problem.

As a rule, use static (global) variables if shared data problems are tackled and use static variables in case it needs saving frequently on the stack.

11. Combine two functions if possible. For example, the search functions fr finding pointers t a list item and pointers of previous list items combine into one. If present is *false* the pointer of the previous list item retrieves the one that has the *item.*

    As a rule, combine whenever feasible two functions of more of less similar codes.
12. Use if feasible, alternatives to the *switch-case* statements, a table of pointers to the functions. This saves the processor time in deciding which set of statements to execute in place of performing the conditional tests all down a chain.
13. When usin C++, configure the compiler for not permitting the multi-inheritance, templates, exceptional handling, new style casts, *virtual* base classes *and namespaces.*

    As a rule, for using C++, use the classes without multiple inheritance, without template, with run-time identification and with throwable exceptions.
14. When using Java, use the J2ME and configure the device classes.

    As a rule, us J2ME with device configurations when programming small-devices code in Java.

Embedded software designers can use various standard ways for optimizing the meory needs in the system.

Methods of Saving and Optimizing the Power Needs

Following are the methods of power saving.

Switch to standby and stop modes

An embedded system has to perform tasks continuously from power-up and may also be left in power-ON state; therefore, power saving during execution is important. A microcontroller used in the embedded system must provide for executing *Wait* and *Stop* instructions and operation in power-down mode. One way to do this is to cleverly incorporate into the software the *Wait* and *Stop* instructions. For example, a program can be such that it reduces the brightness level of the LCD panel so that it takes less power when the system is used in a fully lighted room. A sensor senses the light level at specific intervals.

An embedded system may need to be run continuously, without being switched off; the system design, therefore, is constrained by the need to limit power dissipation while it is running. Total power consumption by the system in running, waiting and idle states should also be limited. A program can provide for auto-switch over the standby mode in case the system is not used within a specified time interval and stop mode when the system is not used for long intervals. For example, mobile phone auto-switch off the LCD lights when not using for 5 or 10 or 20 seconds. A call attend mode can be switched off if there is no talk for over a minute.

The *current* needed at any instant in the processor for an embedded system depends on the state and mode of the processor. The following are the typical values in six states of the processor.

1. 50 mA when only the processor is running; that is, the processor is executing instructions.
2. 75 mA when the processor plus the external memories and chips are in a running state; that is, fetching and execution are both in progress.
3. 15 μA when only the processor is in the stop state; that is, fetching and execution have both stopped and the clock has been disabled from all structural units of the processor.
4. 15 μA when the processor plus the external memories and chips are in the stop state; that is, fetching and execution have both stopped and the clock disabled from all system units.
5. 5 mA when only the processor is in the waiting state; that is, fetching and execution have both stopped but the clock has not been disabled from the structural units of the processor, such as timers.

6. 10 mA when the processor, the external memories and the chips are in the waiting state. Waiting state now means that fetching and execution have both stopped; but the clock has not been disabled from the structural units of the processor and the external IO units and dynamic RAM refreshing also has not stopped.

Disable cache mode Yet another method is to disable use of certain structural units of the processor – for example, caches – when not necessary and to keep in disconnected state those structure units that are not needed during a particular software portion execution, for example, timers or IO units. The software designer should enable the use of caches in a processor by an appropriate instruction, to obtain greater performance during run of a section of a program, while simultaneously disabling the remaining sections in order to reduce the power dissipation and minimize the system energy requirement. Hardware designers should select a processor with multiway cache units so that only that part of a cache unit gets activated that has the data necessary to execute a subset of instructions. This also reduces power dissipation.

Reduce circuit glitches In a CMOS circuit, power dissipates only at the instance of change in input. Therefore, unnecessary glitches and frequent input changes increase power dissipation. VLSI circuit designs have a unique way of avoiding power dissipation. A circuit design is made such that it eliminates all removable glitches, thereby eliminating any frequent input changes.

Low-voltage operation modes Another is to operate the system at the lowest voltage levels in the idle state by selecting power-down mode in that state.

(1) The processor goes into a stop state when it receives a 'stop' instruction. The stop state also occurs in the following conditions: (i) On disabling the clock inputs to the processor. (ii) On stopping the external clock circuit functions. (iii) On the processor operating in auto-shutdown mode. When in the stop state, the processor disconnects with the buses (buses become in tri-state). The stop state can change to a running state. The transition to the running state is either because of a user interrupt or because of the periodically occurring wake-up interrupts.

(2) The processor goes into a waiting state either on receiving (i) an instruction for *Wait*, which slows or disables the clock inputs to some of the processor units including ALU, or (ii) when an external clock circuit becomes non-functional. The timers are still operating in the waiting state. The waiting state changes to the running state when either (i) an interrupt occurs or (ii) a reset signals.

(3) Power dissipation reduces typically by 2.5 μW per 100 kHz reduced clock rate. So reduction from 8000 kHz to 100 kHz reduces power dissipation by about 200 μW, which is nearly similar to when the clock is non-functional. [Remember, the total power dissipated (energy required) may not reduce. This is because on reducing the clock rate the computations will take a longer time at the lower clock rate and the total energy required equals the power dissipation per second multiplied by the time]. The power 25 μW is typically the residual dissipation needed to operate the timers and few other units. By operating the clock at lower frequency or during the power-down mode of the processor, the advantages are as follows: (i) heat generation reduces. (ii) radiofrequency interference also then reduces due to the reduced power dissipation within the gates. [Radiated RF (radiofrequency) power depends on the RF current inside a gate, which reduces due to increase in 'ON' state resistance between the drain and channel when there is reduced heat generation.]

(4) Low-voltage systems are built using LVCMOS (low-voltage CMOS) gates and LVTTL (low-voltage TTL). Use of 3.3 V, 2.5 V, 1.8 V and 1.5 V systems and IO interfaces other than the conventional 5 V systems results in significantly reduced power-consumption and can be advantageously used in the following cases. (i) In portable or hand-held devices such as a cellular phone (compared with 5 V, a CMOS circuit power dissipation reduces by half, $\sim(3.3/5)^2$, in 3.3 V operation. This also increases the time intervals needed for recharging the battery by a factor of two). (ii) In a system with smaller overall geometry, the low-voltage system processors and IO circuits generate lesser heat and thus can be packed into a smaller space.

Clever real-time programming by using 'Wait' and 'Stop' instructions and disabling certain units when not needed is one method of saving power during program execution. Operations can also be performed at reduced clock rate when needed in order to control power dissipation. Good design must optimize the conflicting needs of low power dissipation and fast and effective program execution.

8.10 RTOS TASK SCHEDULING MODELS, INTERRUPT LATENCY AND RESPONSE TIMES OF THE TASKS AS PERFORMANCE METRICS

Following are the common scheduling models used by schedulers.

1. Cooperative scheduling of ready tasks in a circular queue. It closely relates to function queue scheduling.
2. Cooperative scheduling with precedence constraints.
3. Cyclic and round robin (time slicing) scheduling.
4. Preemptive scheduling.
5. Scheduling using 'earliest deadline first' (EDF) precedence.
6. Rate monotonic scheduling using 'higher rate of events occurrence First' precedence.
7. Fixed times scheduling.
8. Scheduling of periodic, sporadic and aperiodic tasks.
9. Advanced scheduling algorithms using the probabilistic timed Petri nets (stochastic) or multithread graphs. These are suitable for multiprocessors and for complex distributed systems.

An RTOS commonly executes the codes for the multiple tasks as priority-based preemptive scheduler.

8.10.1 Cooperative Scheduling Model

First consider a scheduling by a cooperative scheduler function by a simple example. Consider an embedded system – an automatic washing machine. The system can be partitioned into multiple tasks. First three tasks are task *A1*, task *A2* and task *A3* in a set of tasks A_1 to A_N. Figure 8.2(a) shows the first three tasks of the multiple process embedded software. The scheduler first starts the task *A1* waiting loop and waits for the message *A1* from task *A1*.

1. *Task A1:* The task is to reset the system and switch on the power if the door of the machine is closed and the power switch pressed once and released to start the system. Task 1 waiting loop terminates after detection of two events – (i) door closed and (ii) power switch pressed by the user. At the end, task 1 sets a flag start_F, which is a message *A1* to schedule task *A2* to start executing code. This message can be sent using semaphore function OSSemPost (start_F) (Section 7.7.1).
2. *Task A2:* The scheduler waits for the message *A1* for start_F setting. The waiting can be by using semaphore function OSSemPend (start_F). If start_F posting event occurs at task 1, the task 2 starts. A bit is set to signal water into the wash tank and repeatedly checks for the water level. When the water level is adequate the flag water-stage1_F is set, which is a message *A2* to schedule task *A3* to start executing code. This message can be sent using semaphore function OSSemPost (water-stage1_F).
3. *Task A3:* The scheduler waits for the message *A2* for the stage1_F setting. The waiting can be by using semaphore function OSSemPend (water-stage1_F). If water-stage1_F posting event occurs at task 2 the task 3 wait ends and starts. A bit is set to stop water inlet and another bit sets to start the wash tank motor. Then a flag, motor-stage1_F is set, which is a message *A3*, to the schedule the next task to start executing code. This message can be sent using semaphore function OSSemPost (motor-stage1_F).

Figure 8.2(b) shows the cooperative scheduling model. Figure 8.2(c) shows the task program contexts at various instances. Task *A1* context has a pointer for task *A1*, ADDR_A1. Task *A2* context has a pointer for task *A2*, ADDR_A2. Task *A3* context has a pointer for task *A3*, ADDR_A3.

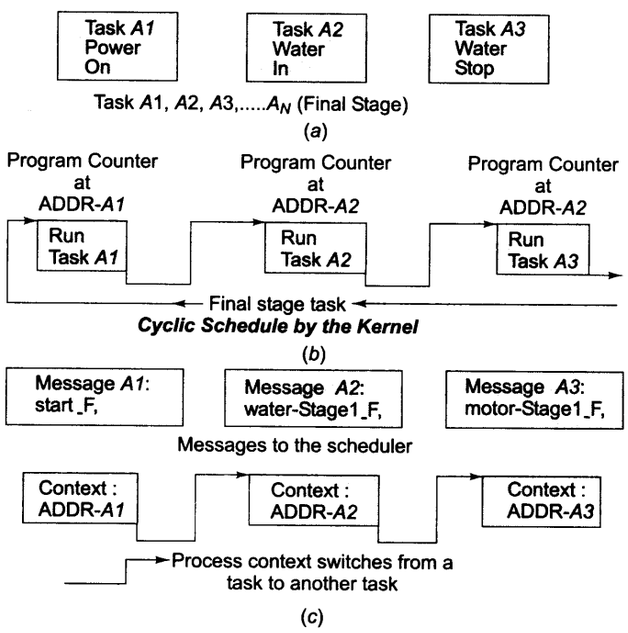

Fig. 8.2 (*a*) First three tasks in a set of tasks A_1 to A_N into which the embedded software is broken for the example in the text (*b*) Cyclic scheduling (*c*) Messages from the scheduler and task program contexts at various instances in washing machine tasks

The Cooperative Scheduling of Ready Tasks List Figure 8.3(a) shows a scheduler in which the scheduler inserts into a list the ready tasks for sequential execution in cooperative model. Program counter PC changes whenever the CPU starts executing another process. Figure 8.3(b) shows how the PC changes on switch to another context. The scheduler switches the context such that there is sequential execution of different tasks, which the scheduler calls from the list one by one in a circular queue.

Cooperative means that each ready task cooperates to let a running one finish. None of the tasks does a block anywhere during the ready to finish states. The service is in the order in which a task is initiated on interrupt and placed in ready list. We can say that the task priority parameter sets as per its position in the queue.

Worst-case latency is the same for each task. It is t_{total}. It is time-period of the circular queue. The longer the queue, the greater is the t_{total}. If a task is running, all other ready tasks must wait. For an i-th task, let the event detection time when an event is brought into a list be dt_i, switching time from one task to another be st_i and task execution time be et_i. Then if there are n tasks in the ready list, the worst-case latency with scheduling when including the ISRs execution times will be:

$$T_{worst} = \{(dt_i + st_i + et_i)_1 + (dt_i + st_i + et_i)_2 + ... + (dt_i + st_i + et_i)_{n-1} + (dt_i + st_i + et_i)_n\} + t_{ISR} = t_{total} + t_{ISR}.$$

Here the t_{ISR} is the sum of all execution times for the ISRs. Remember, the T_{worst} should always be less than the deadline, T_d for any of the task in the list (Refer to Section 4.6).

The Cooperative Scheduling of Ready Tasks Using an Ordered List as per Precedence Constraints Figure 8.4(a) shows a cooperative *priority-based* scheduling of the ISRs executed in the first

layer (top-right side) and *priority-based* ready tasks at an ordered list executed in the second layer (bottom-left), respectively. Figure 8.4(b) shows the PC switch at different times, when the scheduler calls the ISRs and the corresponding tasks at an ordered list one by one. The scheduler using a priority parameter, taskPriority, does the ordering of list of the tasks.

The scheduler first executes only the first task at the ordered list, and the t_{total}, equals the period taken by the first task on the list. It is deleted from the list after the first task is executed and the next task becomes the first. The insertions and deletions for forming the ordered list are made only at the beginning of each list.

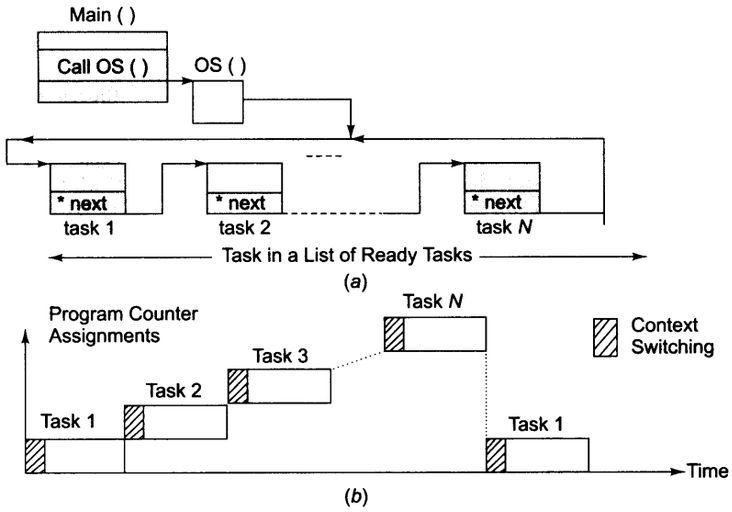

Fig. 8.3 (*a*) An OS scheduling in which the scheduler inserts into a list the ready tasks for a sequential execution in a cooperative mode (*b*) Program counter assignments (switch) at different times, when the scheduler calls the tasks one by one in the circular queue from the list

At the first layer, an ISR has a set of short codes that have to be executed immediately. The ISRs run in the first layer (top-right in figure) according to their assigned priorities. It sends a flag(s) or token(s) and its priority parameter for the task to be initiated (serviced). This task inserts into the ready task list. *There is cooperative scheduling and each ready task cooperates to let the running one finish. None of the tasks does a block anywhere from the start to finish. Here, however, the next start of scheduling is among the ready tasks that run in turn only from a priority-wise ordered list. The ordering is according to the precedence of the interrupt sources and tasks.*

Let p_{em} be the priority of that task which has the maximum execution time. Then worst-case latencies for the highest priority and lowest priority tasks will now vary from:

$\{(dt_i + st_i + et_i)_{pem} + t_{ISR}\}$

to $\{(dt_i + st_i + et_i)_{p1} + (dt_i + st_i + et_i)_{p2} + ... + (dt_i + st_i + et_i)_{pm-1} + (dt_i + st_i + et_i)_{pm} + t_{ISR}\}$.

Here, $p_1, p_2, ..., p_{m-1}$ and p_m are the priorities of the tasks in the ordered list. Also $p_1 > p_2 > ... > p_m$. With this scheduler, it is easier, but not guaranteed, to meet the requirement that T_{worst} should be $<T_d$ for each task and interrupt source. The programmer assigns the lowest T_d task a highest priority.

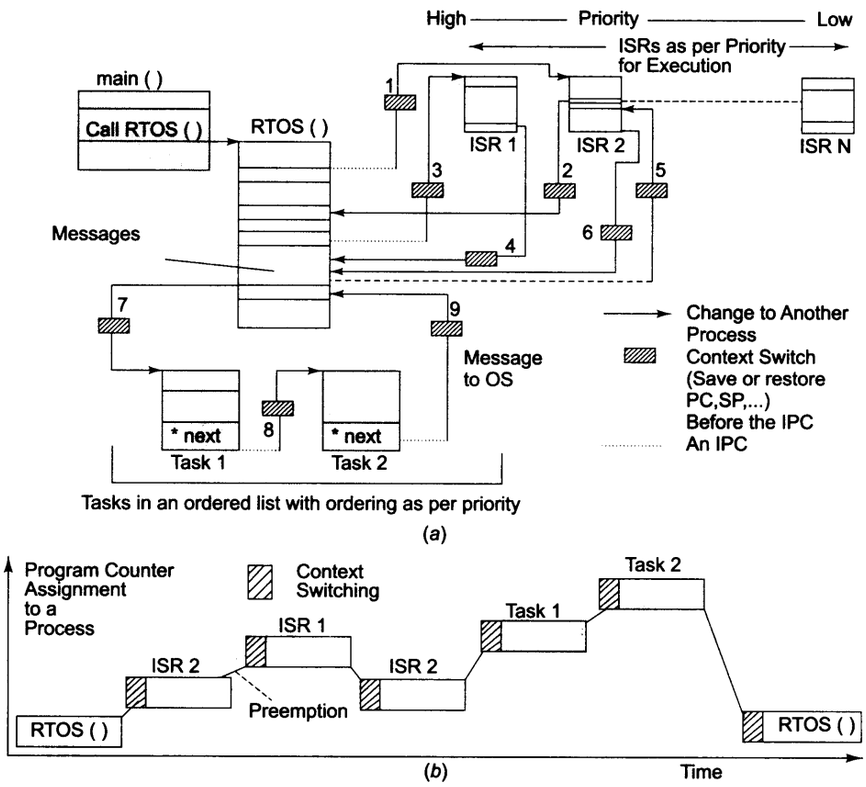

Fig. 8.4 (*a*) Cooperative priority-based scheduling of the interrupt service routines (ISRs) executed in the first layer (top-right side) and priority-based ready tasks at an ordered list executed in the second layer (bottom-left) (*b*) Program counter assignments at different times on the scheduler calls to the ISRs and the corresponding tasks

Example 8.17

Consider the ACVM example (Section 1.10.2). First the coins inserted by the user are read, then the chocolate delivers, and then display task displays 'thank you, visit again' message. Each task cooperates with the other to finish. The precedence of the task reading the coins is highest, then of chocolate delivery and display for the ordered list of ready tasks.

8.10.2 Cyclic and Round Robin with Time Slicing Scheduling Models

Cyclic Scheduling An OS scheduler can let the system schedule the various tasks in real time as follows: let us assume that we have periodically occurring three tasks, the need for their service arises after periodically. Let the time-frames be allotted to the first task, the task executes at t_1, $t_1 + T_{cycle}$, $t_1 + 2 \times T_{cycle}$, . . . second task

frames at t_2, $t_2 + T_{cycle}$, $t_2 + 2 \times T_{cycle}$ and the third task at t_3, $t_3 + T$, $t_3 + 2 \times T_{cycle}$, ... Start of a time frame is the scheduling point for the next task in the cycle. T_{cycle} is the cycle for repeating the cycle of execution of tasks in order 1, 2 and 3 and equals start of task 1 time frame to end of task 3 frame. T_{cycle} is the period after which each the task time frame allotted to that repeats.

Each of the N tasks in a cyclic scheduler completes in its allotted time frame when the time frame size is based on the deadline. A cyclic scheduler is clock-driven and is useful for the periodic tasks. It repeats the schedule decided after computations based on the period of occurrences of task instances. Each task has the same priority for execution in the cyclic mode.

Example 8.18

(a) Consider the video and audio signals reaching at the ports in a multimedia system and processed. The video frames reach at the rate of 25 in 1 second. The cyclic scheduler is used in this case to process video and audio with T_{cycle} = 40 ms or in multiples of 40 ms.
(b) Consider the orchestra-playing robots example (Section 1.10.7). First the director robot sends the musical notes. Then, the playing robots receive and acknowledge to the director. In the next cycle, the master robot again sends the musical notes. The cyclic scheduler is used in this case to send and receive signals by each robot.

A cyclic scheduling is very efficient for handling periodic tasks and when the number of tasks is small.

Round Robin Time Slicing Scheduling A task may not complete in its allotted time frame. Round robin means that each ready task runs in turn only in a cyclic queue for a limited time slice T_{slice}. $T_{slice} = T_{cycle} \div N$, where N = number of tasks. It is a swidely used model in traditional OS. Round robin is a hybrid model of the clock-driven model (e.g., cyclic model) as well as event-driven (e.g., preemptive). A real-time system responds to the event within a bound time limit and within an explicit time. A scheduler for the time-constrained tasks in the round robin mode can be understood by a simple example.

Suppose after every 20 ms, there is a stream of coded messages reaching at port A of an embedded system, it is then decrypted and retransmitted to the port after encoding each decrypted message. The multiple processes consist of five tasks: *C1*, *C2*, *C3*, *C4* and *C5*, as follows:

1. Task *C1*: Check for a message at port A every 20 ms.
2. Task *C2*: Read port A and put the message in a message queue.
3. Task *C3*: Decrypt the message from the message queue.
4. Task *C4*: Encode the message from the queue.
5. Task *C5*: Transmit the encoded message from the queue to port B.

Figure 8.5(a) shows five tasks, *C1 to C5,* that are to be scheduled. Figure 8.5(b) shows the five contexts in five time schedules, between 0 and 4 ms, 4 and 8 ms, 8 and 12 ms, 12 and 16 ms and 16 and 20 ms, respectively. Let OS initiate *C1* to *C5*. Let there be slice-clock tick interrupts at each 4 ms. Task *C1* is scheduled by OS to bring it to the running state from its blocked state as soon as a timer triggers an event. If it is known that after every 20 ms a byte reaches port A, a timer interrupt triggers an event every 4 ms. Task *C1* runs within 4 ms, and *C2* starts running.

Figure 8.5(b) shows at different time slices the real-time schedules, process contexts and saved contexts.

1. At the first instance (first row) the context is *C1* and task *C1* is running.
2. At the second instance (second row) after 4 ms, the OS switches the context to *C2*. Task *C1* is finished, *C2* is running. As task *C1* is finished, nothing is saved on the task *C1* stack.

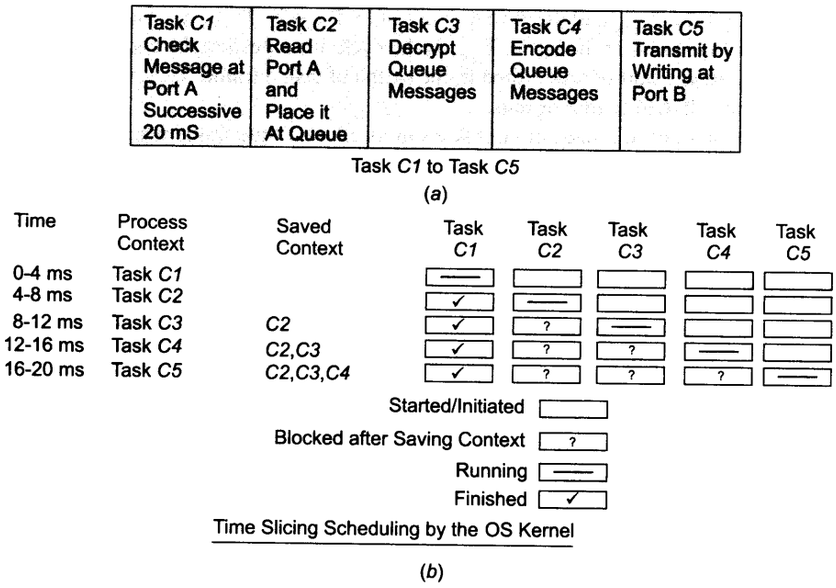

Fig. 8.5 (*a*) The tasks *C1* to *C5* round robin (b) Task program contexts at five instances in the round robin (time slice) scheduling scheduler for *C1* to *C5* with T_{slice} = 4 ms

3. At the third instance (third row), the OS switches the context to *C3* on next timer interrupt, which occurred after 8 ms from the start of task *C1*. Task *C1* is finished, *C2* is blocked and *C3* is running. Context *C2* is saved on task *C2* stack because *C2* is in blocked state.
4. At the fourth instance (fourth row), the OS switches the context to *C4* on timer interrupt, which occurred after 12 ms from the start of task *C1*. Task *C1* is finished, *C2* and *C3* are blocked and *C4* is running. Contexts *C2* and *C3* are at the tasks *C2* and *C3* stacks, respectively.
5. At the fifth instance (fourth row), the OS switches the context to *C5* on next timer interrupt which occurred after 16 ms from the start of task *C1*. Task *C1* is finished, *C2*, *C3* and *C4* are blocked and *C5* is running. Contexts *C2*, *C3* and *C4* are at the tasks *C2*, *C3* and *C4* stacks, respectively.
6. On a timer interrupt at the end of 20 ms, the OS switches the context to *C1*. As task *C5* is finished, only the contexts *C2*, *C3* and *C4* remain at the stack. Task *C1* is running as per its schedule.

When a *p*-th task has high *execution time*, et_p, the worst-case latency of the lowest priority task can exceed its deadline. To overcome this problem, it is better that the OS defines a lower time slice for each task. Each task has codes in an infinite loop. Cyclic scheduling with time slicing is simple and there is no insertion or deletion into the queue or list. Figure 8.6(a) shows a programming model for cyclic *time-sliced round robin* scheduling. Figure 8.6(b) shows PC on context switches when the scheduler call to tasks at two consecutive time slices. Each task is allotted a maximum time interval $= t_{slice}/N$, where t_{slice} is the time after which a timer (with the OS) interrupts and initiates a new cycle.

The OS completes the execution of all ready tasks in one cycle within a time slice, $N \times t_{slice}$ in this mode. Let T_{worst} be the sum of the maximum times for the all the tasks if there are N *tasks* in all. Then, when $t_{slice} >$ or $= T_{worst}$, the T_{worst} equals:

$$\{(dt_i + st_i + et_i)_1 + (dt_i + st_i + et_i)_2 + ... + (dt_i + st_i + et_i)_{n-1} + (dt_i + st_i + et_i)_m\} + t_{ISR}.$$

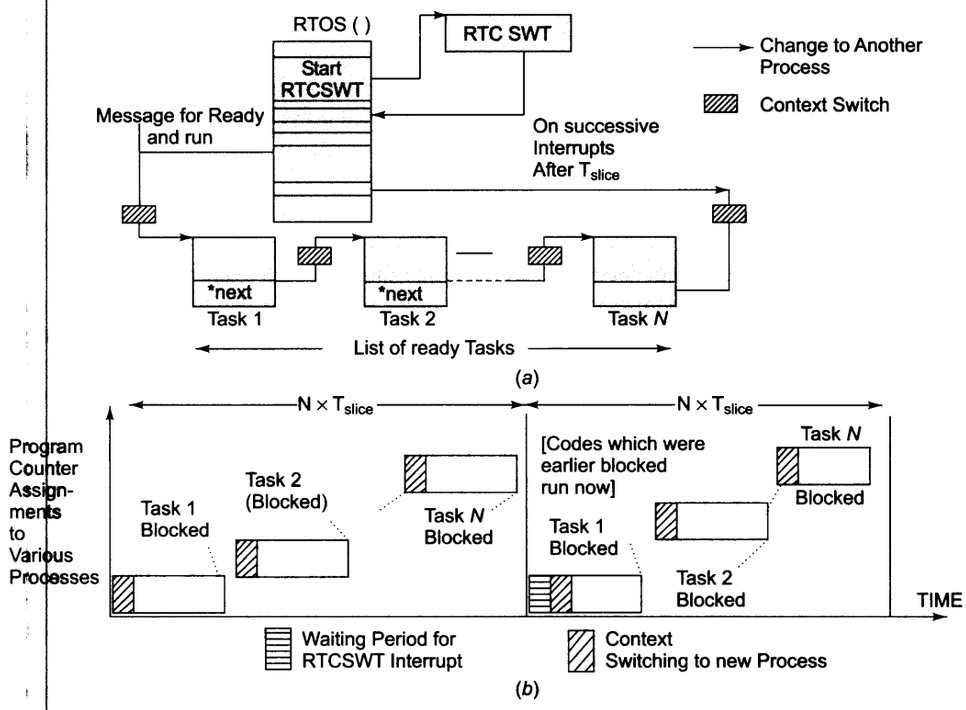

Fig. 8.6 (*a*) The programming model for the cooperative time-sliced scheduling of the tasks (*b*) The program counter assignments on the scheduler call to tasks at two consecutive time slices. Each cycle takes a time of t_{slice}

If $N \times t_{slice}$ equals the sum of the maximum times for each task, then each task is executed once and finishes in one cycle itself. When a task finishes the execution before the maximum time it can take, there is a waiting period between the two cycles. The worst-case latency for any task is $N \times t_{slice}$. A task may periodically need execution. The period for the required repeat execution of a task is an integral multiple of t_{slice}. For each task to run only once, the $N \times t_{slice}$ should also be less that the greatest common factor of all the task periods. The estimation of response time for each task is easy in time slice cyclic round robin scheduling. Consider a *k*-th task. The task responds within its task period plus the sum of the maximum times taken during a time slice from the task 1 to task (*k*–1). The response time of the *m*-th task at the end of the list is the maximum.

An alternative model strategy can be the *decomposition* of a task that takes an abnormally long time to be executed. The decomposition is *into two or four or more tasks*. Then one set of tasks (or the odd numbered tasks) can run in one time slice, t'_{slice} and another set of tasks (or the even-numbered tasks) in another time slice, t''_{slice}.

Another alternative strategy can be the *decomposition of the long time-taking task* into a number of sequential states or a number of node places and transitions as in the FSM. Then one of its states or transitions runs in the first cycle, the next state in the second cycle and so on. This task then reduces the response times of the remaining tasks that are executed after a state.

Example 8.19

Assume a VoIP [Voice Over IP] router. It routes the packets to N destinations from N sources. It has N calls to route. Each of the N tasks is allotted a time slice and is cyclically executed for a routing packet from a source to its destination.

8.10.3 Preemptive Scheduling Model

Cooperative schedulers (described in Section 8.10.1) schedule such that each ready task cooperates to let the running one finish. However, *a disadvantage of the cooperative scheduler is that a long execution time of a low-priority task makes a high-priority task wait at least until it finishes*. There is a further disadvantage if the cooperative scheduler is cyclic but without a predefined t_{slice}. Assume that an interrupt for service from the first task occurs just at the beginning of the second task. The first task service waits till all other remaining listed or queued tasks finish (Section 8.10.1).

The time-slicing scheduler is simpler in design and extremely valuable in many applications where there is a need to use the resources of the embedded systems sequentially, or none of the tasks has a shorter deadline than the t_{slice} or t_{cycle}. Round robin scheduler (described in Section 8.10.2) also give appropriate time slice to let a task finish with the allotted time frame. Now consider the problem with round robin. Let there be N tasks from task 1 to task N and let the assigned order of priority for interrupt servicing be from 1 (highest) to N (lowest). Assume now that an interrupt occurs in the time-slicing scheduling just after the cycle starts. It means task 1 misses by a flick the chance of running from start to finish as task 1 will not get serviced till the cycle up to task N finishes or till the defined period t_{slice} expires.

Can the higher-priority task preempt a lower priority by blocking it? If yes, then this can solve the problem of large worst case latency for high priority tasks. The hardware polls to determine whether an ISR or task with a higher priority than the present one needs service at the end of an instruction during execution. If yes, then the higher priority ISR or task is executed. Similarly, the RTOS preemptive scheduler can block a running task at the end of an instruction by a message to the task and let the one with the higher priority take control of the CPU.

Now consider a preemptive scheduler by a simple example. Suppose there is a stream of coded messages reaching at port A of an embedded system. It then decrypts and re-transmits to port B after encoding each decrypted message (recall Example 4.1). Figure 8.7(a) shows the tasks for the multiple processes of this application. Five processes are executed at five tasks, *B1*, *B2*, *B3*, *B3*, *B4* and *B5*. Now consider preemptive scheduling by a scheduler function by another example. Consider an embedded system for reading a port A input and decrypting the input data, encoding it and sending it to another port B output. The system can be partitioned into multiple tasks. Five tasks are task *B1*, *B2*, *B3*, *B4* and *B5*. Figure 8.7(a) shows the assigned functions to the task and ISR. The order of priorities is as follows.

1. Task *B1*: Check for a message at port A.
2. Task *B2*: Read port A.
3. Task *B3*: Decrypt the message.
4. Task *B4*: Encode the message.
5. Task *B5*: Transmit the encoded message to the port.

Figure 8.7(b) gives the symbols used to show the preemptive scheduling by the kernel pre-emptive scheduler actions shown in Figure 8.7(c). A higher priority task takes control from a lower priority task. A higher priority task switches into the running state after blocking the low priority task. The context saves on the pre-emption. Figure 8.7(c) shows the following.

1. At the first instance (first row) the context is *B3* and task *B3* is running.
2. At the second instance (second row) the context switches to *B1* as context *B3* saves on interrupt at port A and task *B1* is of highest priority. Now task *B1* is in a running state and task *B3* is in a blocked state. Context *B3* is at the task *B3* stack.
3. At the third instance (third row) the context switches to *B2* on interrupt, which occurs only after task *B1* finishes. Task *B1* is in a finished state, *B2* in a running state and task *B3* is still in the blocked state. Context *B3* is still at the task *B3* stack.
4. At the fourth instance (fourth row) context *B3* is retrieved and the context switches to *B3*. Tasks *B1* and *B2*, both of higher priorities than *B3*, are finished. Tasks *B1* and *B2* are in finished states. Task *B3* blocked state changes to running state and *B3* is now in a running state.
5. At the fifth instance (fifth row) the context switches to *B4*. Tasks *B1*, *B2* and *B3*, all of higher priorities than *B4*, are finished. Tasks *B1*, *B2* and *B3*, are in the finished states. *B4* is now in a running state.
6. At the sixth instance (sixth row) the context switches to *B5*. Tasks *B1*, *B2*, *B3* and *B4*, all of higher priorities than *B5*, are finished. Tasks *B1*, *B2*, *B3* and *B4*, are in the finished states. *B5* is now in a running state.

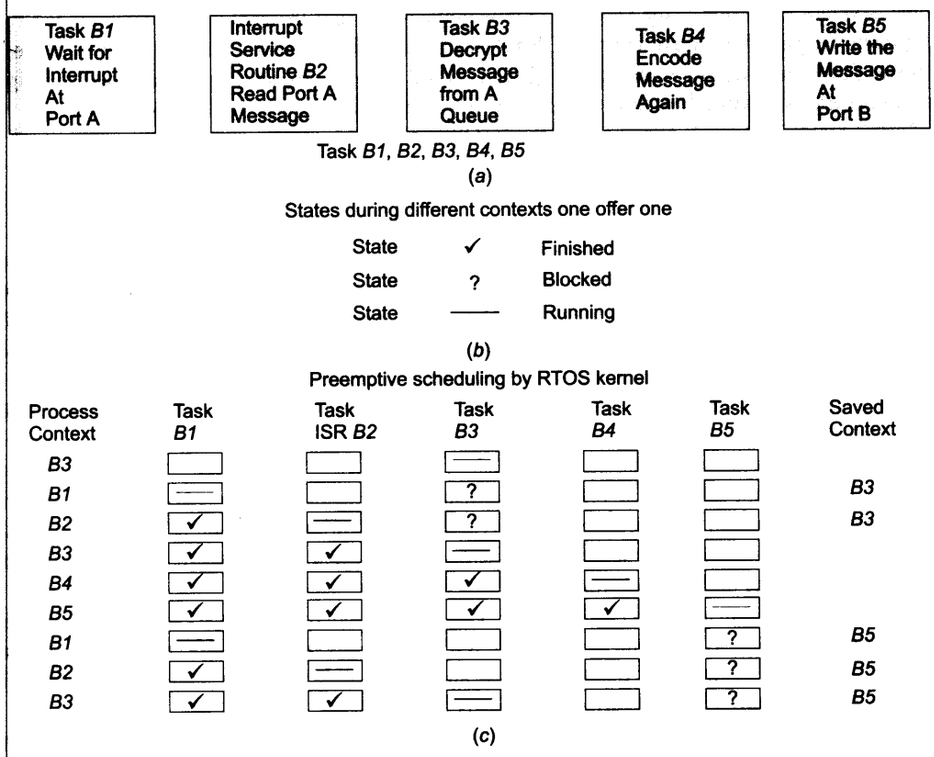

Fig. 8.7 (*a*) First five tasks *B1* to *B5* (*b*) The symbols used for the states in a preemptive scheduling (*c*) The task program contexts at the various instances

7. At the seventh instance (seventh row) the context switches to *B1* as context *B5* is saved on interrupt at port A, and task *B1* is of highest priority. Now task *B1* is in a running state and task *B5* is in a blocked state. Context *B5* is at the task *B5* stack.
8. At the eighth instance (eighth row) the context switches to *B2* on interrupt, which occurs only after task *B1* finishes. Task *B1* is in a finished state, *B2* in a running state and task *B5* is still in the blocked state. Context *B5* is still at the task *B5* stack.
9. At the last instance (last row) the context is *B3* and task *B3* is running. The tasks *B1* and *B2* are in finished state.

RTOS manages the processes and provides for preemption of lower priority process by higher priority process. (Table 8.11). Let the priority of task_1 > task_2> task_3 > task_4.... > task *N*. Figure 8.8(a) shows the preemptive scheduling of *N* tasks. Figure 8.8(a) also shows the context switching whenever the process switches from a task to the RTOS and from the RTOS to a task. Figure 8.8(b) shows PC assignments on the scheduler call to pre-empt task 2 when the priority of task_1 > task_2 > task_3.

Each task has an infinite loop from start (idle state) up to finish (refer to task 1, task 2 and task *N*, three boxes at the bottom of this figure). Last instruction of task 1 points to the next pointed address, *next. In case of the infinite loop, *next points to the same task 1 start. It is unlike a cooperative scheduler (Section 8.10.1), where it signals the next task execution to the OS and OS now initiates and runs the next task in the ready list.

In a preemptive scheduler, there is an RTOS message during the running of task 2 to preempt the task 2. Figure 8.8(a) shows the sequence markings (1), (2) and (3) and Figure 8.8(b) shows program counter assignment. Their meanings are as follows. In step 1, task 2 is run. The higher priority task 1 is initiated as follows:

1. Task 2 blocks and sends a message to the RTOS (step 2).
2. The RTOS now sends a message to task 1 to go to the unblocked state and run (step 3).

After task 1 blocks then RTOS makes the task 2 in the unblocked state. Task 2 now runs. When task 2 blocks then RTOS makes task 3 in the unblocked state. Task 3 will run now.

Each task design is like an independent program, in an infinite loop between the task ready place and the running task place. The task does not return to the scheduler, as a function does. Within the loop, the actions and transitions are according to the events or flags or tokens. The context switching may also occur on an ISR call.

We can define timeout for waiting for the token or event. An advantage of using time out intervals while designing task codes is that worst-case latency estimation is possible. Any task's worst-case latency is the sum of the t_{ISR} and the intervals of all other tasks are of higher priority. Another advantage of using the time outs is the error reporting and handling by the RTOS. Timeouts provide a way to let the RTOS run even the lowest priority task in necessary cases.

Whenever the preemption event takes place, a task switching (a task place transition to its running place) becomes necessary, and the scheduler searches for the highest priority task at that instance. That task only is switched to the running place by the scheduler. Switching occurs when a taskSwitchFlag is sent to the highest priority task and not to the task that was running previously.

How can the context switching intervals reduce? The *context switching* intervals are reduced by the static declaration of the variables, as the static variables are RAM-resident variables and do not save on the stack on a function call. When this is the case, on a call the PC and few must-save registers are saved. Task switching now does not lead to additional stack-saving overheads.

The conditions in which an event (token), the *preemptionEvent,* is generated for a task to undergo transition from the running place to the ready place are as follows..

1. The preemption event takes place when an interrupt occurs and just before the return from the interrupt, there is a service call to the RTOS by the ISR. On this call to the RTOS, a token, the *preemptionEvent*,

is set. The task then undergoes transition to the place, *readyTaskPlace*, and runs only when asked by the scheduler (by sending *taskSwitchFlag*).

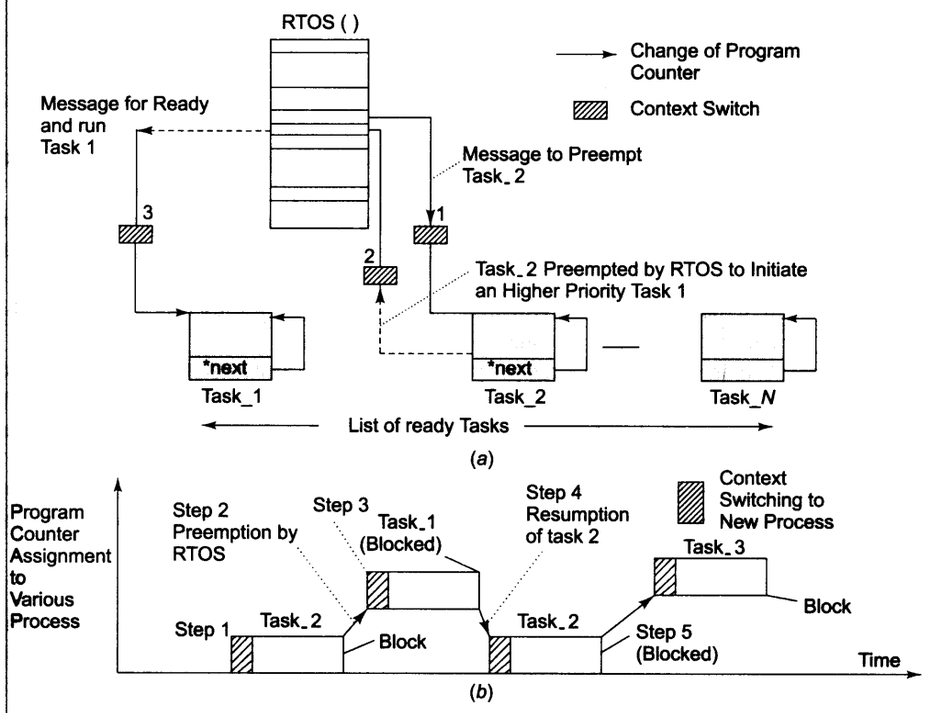

Fig. 8.8 (*a*) Preemptive scheduling of the tasks (a running task is pre-empted and blocked to let a higher priority task be executed). Note the sequence markings (1), (2) and (3). For meanings of these refer to text (*b*) Program counter assignments on a scheduler call to preempt task 2. Priority of task_1 > task_2 > task_3

2. Each RTOS uses a system clock ticked by a SysClkIntr interrupt. The preemption event takes place when the SysClkIntr interrupt (real-time clock-driven software timer interrupt) occurs at the RTOS. On this event RTOS takes control of the processor and checks whether it should let currently executing task continue or to preempt it to make way for the higher priority task. This event makes another higher priority task ready to run, on the switch of the flag to the latter.
3. The preemption event takes place when any call to the RTOS occurs to enter the critical section or for sending the task message (outputs) to the RTOS, and if another higher priority task then needs to be serviced (take control of the CPU) (now the preemption is before entering the critical section).

Critical Section Service by a Preemptive Scheduler Critical section is a section in a system call (OS function), where there is no preemption by either ISRs or higher priority tasks. Critical section is also a section in task to prevent preemption. A lock function executes before beginning of critical section and an unlock function executes at exit from the critical section.

Assume that a task *I* is waiting for a critical section resource, and task *J* is using a kernel lock at an instance because of the system call for lock by another task *j*. When task *J* executes unlock function, the task *I* waiting for lock will run.

Assume that context switching time is large, for example, 1 ms, while the critical section resource is required by the task for much shorter duration, then kernel lock is an inefficient mechanism to lock a critical section. A **spin lock** (Section 7.11.1) can be used to protect the critical section resources as follows.

Assume that task *J* is being provided the critical section resource through spin lock, s_{lock}. Assume that task *I* needs the critical section resource at instance t_i. The spin lock concept provides a busy wait loop for *I* (on executing lock ( ) function). The *I* goes at t_i into a busy wait loop for the spin lock, s_{lock}. As soon as *J* releases the s_{lock} (on executing unlock ( ) function at *J*), the I_i gets the critical resource without spenting time for the context switch.

An implementation of the spin lock in a task can be by a try. The high priority task tries the lock by a wait loop for the lock for a defined time t_{wait}, else the task un-blocks.

Another implementation of the spin lock can be by trying two or four wait loops for the lock with successive decrements in the time t_{wait} to 0, after which the task un-blocks. After the unblocking, the task will run critical section code without the context switch unlike the case when mutex is used to block or un-block a critical section.

A lock function alternative is execution by taking a mutex and lock by releasing the mutex (Sections 7.7.2 and 7.8.3). Another implementation is by an instruction that disables a specific interrupt at the beginning of a critical section and enables the specific interrupt at the end of the critical section.

A lock function can be used before a critical section. Spin lock is effective for the critical section with a short period of execution, because the busy wait loop will be released in a short time. A mutex can be used for critical section that should run exclusively. Disabling and enabling of interrupts can be used to prevent another ISR or process to run in-between.

Example 8.20

(a) RTOS kernels, for example, Windows CE, provide for pre-emption points. These are the OS function codes in-between the critical sections of kernel processes.
(b) RTOS µCOS-II provides function OS_ENTER_CRITICAL ( ) to stop preemption by any task or ISR, as it disables the interrupt. The RTOS provides function OS_EXIT_CRITICAL ( ) to facilitate preemption by a high-priority task or ISR, as it enables interrupt.
(c) µCOS-II provides OSSchedLock ( ) and OSSchedUnlock ( ) for task critical-section locking to run the section and preventing preemption by other task.

Two tasks may have two sections that share the data or resource and only one section must execute. Before the critical section, a task waits for mutex semaphore from the scheduler and releases the mutex semaphore at the exit from the critical section. Provisioning for priority inheritance permits critical section service by a pre-emptive scheduler without priority inversion (Section 7.8.5).

Can work be done without the semaphores and/or mutex for the critical sections? Yes, one strategy is disabling and enabling the preemption. The disabling of a preemption means disabling only the task switching flags, or their passing to the task when using shared data and enabling the task switching flags to change and pass again after this. But it should then be ensured that all the ISRs are the reentrant functions. Another

strategy could be the use of resources locking semaphore, *mutex* (Sections 7.7.2 and 7.8.2) for spin-lock (Section 7.11.1).

8.10.4 Model for Critical Section Service by a Preemptive Scheduler

Figure 8.9 shows a Petri net concept-based model which models and helps in designing the codes for a task that has a critical section in its running. The figure shows places by the circles and transitions by the rectangles. The following are the places and transitions.

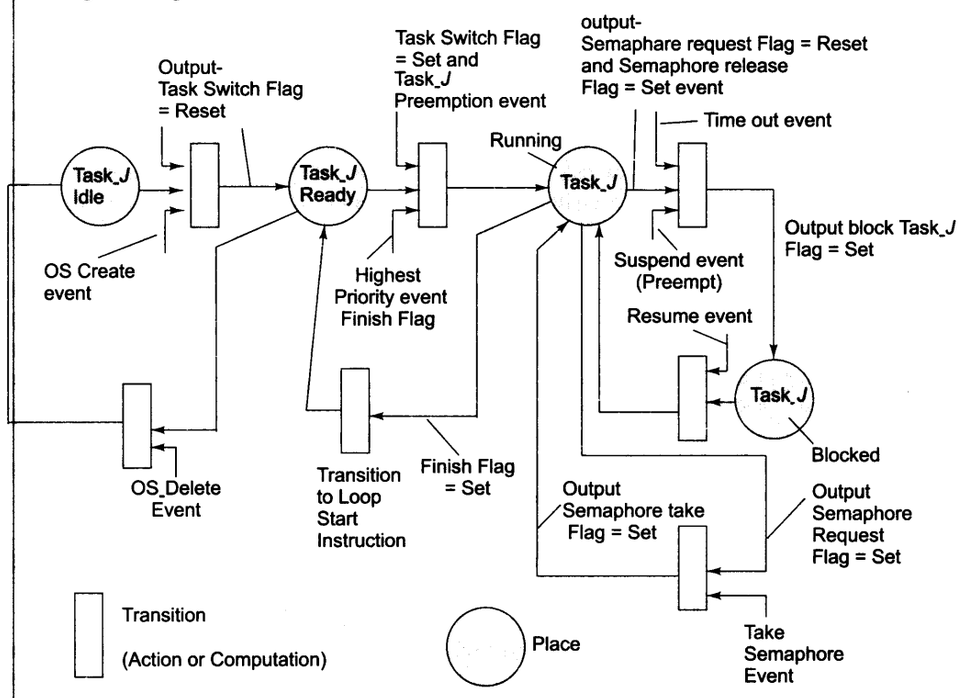

Fig. 8.9 The Petri net model for the task with a preemptive scheduler and one critical section where it takes a semaphore and release on critical section over

1. Each task is in the idle state (at idleTaskPlace) to start with, and a token to the RTOS is *taskSwitchFlag* = reset.
2. Consider the task_J_Idle place, which currently has highest priority among the ready tasks. When the RTOS creates task_J, the place task_J_Idle undergoes a transition to the ready state, task_J_Ready place. The RTOS initiates *idle* to *ready* transition by executing a function, task_J_create ( ). A transition from the idle state of the task is fired as follows. RTOS sends two tokens, RTOS_CREATE Event and *taskJSwitchFlag*. The output token from the transition is *taskSwitchFlag* = *true* (refer to the top-left transition in the figure).
3. When after task *J* finishes, the RTOS sends an RTOS_DELETE event (a token) to the task, it returns to task_J_Idle place and its corresponding *taskJSwitchFlag* resets (refer to the bottom-left transition in the figure).

4. At task_J_Ready place, the scheduler takes the priority parameter into account. If the current task happens to be of the highest priority, the scheduler sets two tokens, *taskJSwitchFlag* = true (sends a token) and *highest Priority Event* = true, for the transition to the running task *J* place, task_J_Running. The scheduler also resets and sends the tokens, task switch flags, for all other tasks that are of lesser priority. This is because the system has only one CPU to process at an instant.
5. From the task_J_Running place, the transition to the task_J_Ready place will be fired when the task finish flag sets (refer to the bottom-middle transition in the figure).
6. At task_J_Running place, the codes of the switched task *J* are executed (refer to the top-right most transition in the figure).
7. At the runningTaskPlace, the transition for pre-empting will be fired when RTOS sends a token, *suspendEvent*. Another enabling token if present, *time_out_event* will also fire the transition. An enabling token for both situations is the semaphore release flag, which must be set. Semaphore release flag is set on finishing the codes of task *J* critical sections. On firing, the next place is task_J_Blocked. Blocking is in two situations. One situation is of preemption. It happens when the *suspendEvent* occurs on a call at the runningTaskPlace asking the RTOS to suspend the running. Another situation is a time out of an SWT that associates with the running task place.
8. On a *resumeEvent* (a token from RTOS) the transition to task_J_Running place occurs (refer to the right-side middle transition, which is between the three transitions that are shown in the figure).
9. At the task_J_Running place, there is another transition that fires so that the task *J* is back at the task_J_Running place when the RTOS sends a token, take_Semaphore_Event to ask the task *J* to take the semaphore (the RTOS sets the semaphore request flag, take_Semaphore_Event; it resets semaphore release flag; it directs task *J* to run un-interrupted. Do not block).
10. There can be none or one or several critical sections. During the execution of a critical section, the RTOS resets the semaphore release flag and sets the take semaphore event token.

8.10.5 Earliest Deadline First (EDF) precedence and Rate Monotonic Schedulers (RMS) Models

The event-driven schedulers are required for real-time scheduling in case of a number of tasks being large or in case of aperiodic or sporadic tasks. Aperiodic task is one in which the period of occurrence is not known because it may not be known when an event can occur. For example, an event of receiving a phone call is aperiodic event. Sporadic task has periods of bursts when the task events occur.

A deadline is the period in which a task must finish. A task, which has a least deadline that is which has little time left for completion, must be scheduled first. This algorithm of the scheduler is known as EDF algorithm.

EDF precedence When a task becomes ready, its will be considered at a scheduling point. The scheduler does not assign any priority. It computes the deadline left at a scheduling point. Scheduling point is an instance at which the scheduler blocks the running task and re-computes the deadlines and runs the EDF algorithm and finds the task to be run.

An EDF algorithm can also maintain a priority queue based on the computation when the new task inserts. When the number of tasks becomes large, the computation complexity increases for insertion into the queue. Another EDF algorithm can also maintain two or more priority queues based on the relative deadlines and the scheduler inserts the new task into one of the queues.

Precedence Assignment in the Scheduling Algorithms The best strategy is one, which is based on EDF precedence. Precedence is made the highest for a task that corresponds to an interrupt source, which occurs at the earliest and which deadline will finish the earliest.

How is the precedence assigned in the case of variable CPU loads for the different tasks and variable EDFs? One method is as follows.

Let t_1 be the instance when task *I* needs preemption the first time and t_2 be the next instance. A task with minimum $(t_2 - t_1)$ is inserted at the top of the task priority list. It is assigned the highest precedence. The list is dynamically ordered according to $(t_2 - t_1)$.

First, there is a deterministic or static assignment of the precedence in advanced scheduling algorithms. It means first there is RMS. Later on the scheduler dynamically assigns and fixes the time out delays afresh, and assigns the precedence as per the EDF. The need for the dynamic assignment arises due to the sporadic tasks and the distributed or multiprocessor indeterminate environment.

Resource sharing among the tasks creates a problem. The algorithm has to ensure that none of them misses the deadline.

A task occurring at a higher rate should then get higher precedence in case of a periodic tasks. Assume that the data is being received from multiple channels and some channel receive data at a faster rate than the others. A scheduler uses rate monotonic algorithm (RMA) to schedule the tasks in this case.

Rate Monotonic Scheduler RMS computes the priorities, p, from the rate of occurrence of the tasks. The *i*-th task priority, p_i is proportional to $(1/t_i)$ where t_i is the period of the occurrence of the task event. RMA gives an advantage over EDF because most RTOSes have provisions for priority assignment. Higher-priority tasks always get executed.

RMA disadvantage is that it does not support aperiodic and sporadic tasks. When a burst occurs, even due to higher rate of arriving of the sporadic task in the burst period, it cannot be assigned high priority. The aperiodic and sporadic tasks can be assigned the tickets by aperiodic and sporadic servers in the scheduler. Ticket means the periods in which the events from them will be scheduled.

RMA disadvantage is that a task may have long periods, but can be very critical. It will be assigned least priority. A solution is to divide the very critical task into two or more tasks to raise their allocated priority by the RMA.

8.10.6 Fixed (Static) Real-Time Scheduling Model

The slice-time scheduling method described in Section 8.10.2 is a special case of 'fixed real-time scheduling'. Every task is allotted fixed schedules to run. Let there be *m* tasks and *m* real-time clock interrupts, the scheduler can thus assign each task a fixed schedule. Each task undergoes a ready place to running place transition on the timeouts of the corresponding timer. The OS is supposed to define hard real-time schedules for each task.

A scheduler is said to be using a fixed-time scheduling method when the schedule is static and deterministic. The working environment is unaltered when processes are scheduled on the single CPU of the system. Schedules are deterministic as the worst-case latencies for all the interrupts and the tasks are predeterminable. The OS scheduler can thus schedule each task at fixed times so that none misses its deadline (this is when the worst-case latency of each task is less then its deadline for its service). The 'no deadline miss' advantage is feasible only in deterministic situations. Coding for the tasks are such that execution times do not vary under the different inputs or different conditions.

Schedules once defined remain static in a fixed-time scheduler. Fixed schedules can be defined by one of the three methods.

1. *Simulated annealing method.* Here the different schedules can be fixed and the performance simulated. Now, schedules for the tasks are gradually incremented by changing the interrupt timer settings (using a corresponding OS function) till the simulation result shows that none is missing its deadline.
2. *Heuristic method.* Here, reasoning or past experience helps to define and fix the schedules.
3. *Dynamic programming model.* This is as follows: a specific running program first determines the schedules for each task and then the timer interrupt loads the timer settings from the outputs from that program.

If the scheduler cannot fix the schedules, it is a non-deterministic situation. An example is a situation in which a message for a task is expected in a network, from another system and the minimum and maximum periods for receiving it are unknown. Another example is when the inputs for a task are expected from another system and the minimum and maximum periods when the inputs will be received are not known.

A dynamic scheduling model is as follows: the software design may be such that the priorities can be rescheduled and fixed times redefined when a message or error message is received during the run.

8.10.7 Latency and Deadlines as Performance Metric in Scheduling Models For Periodic, Sporadic and Aperiodic Tasks

An RTOS should quickly and predictably respond to the event. It should have minimum interrupt latency and fast context switching latency.

Different models have been proposed for measuring performances. Three performance metrics are as follows.

1. Ratio of the sum of interrupt latencies with respect to the sum of the execution times.
2. CPU load.
3. Worst-case execution time with respect to the mean execution time.

'Interrupt latencies' in various task models can be used for evaluating performance metrics. The latencies for various task scheduling models are described in Sections 8.10.1 to 8.10.3. The CPU load is another way to look at the performance. It is explained in Section 8.10.8. Worst-case performance calculation for a sporadic task is explained in Section 8.10.9. 'Refer Real Time Systems' by Jane W. S. Liu, Pearson Education, 2000, for details of many models available for evaluating the performances.

8.10.8 CPU Load as Performance Metric

Each task gives a load to the CPU that equals the task execution time divided by the task period. [Task period means period allocated for a task.] Recall In_AOut_B intranetwork of Example 4.1. Receiver port A expects another character before 172 μs. Task period is 172 μs. If the task execution time is also 172 μs, the CPU load for this task is 1 (100%). The task execution time when a character is received must be less than 172 μs. The maximum load of the CPU is 1 (less than 100%).

The CPU load or *system load* estimation in the case of multitasking is as follows. Suppose there are m tasks. For the multiple tasks, the *sum* of the CPU loads for all the tasks and ISRs should be less than 1. The time outs and fixed-time limit definitions for the tasks reduce the CPU load for the higher-priority tasks so that even the lower-priority tasks can be run before the deadlines. What does it mean when the sum of the CPU loads equal to 0.1 (10%)? It means that the CPU is underutilized and spends 90% of its time in a waiting mode. As the execution times and the task periods vary, the CPU loads can also vary.

When a task needs to run only once, then it is aperiodic (one shot) in an application. Scheduling of the tasks that need to run periodically with the fixed periods can be periodic and can be done with a CPU load very close to 1. An example of a periodic task is as follows. There may be inputs at a port with predetermined periods, and the inputs are in succession without any time gap.

When a task cannot be scheduled at fixed periods, its schedule is called sporadic. For example, if a task is expected to receive inputs at variable time gaps, then the task schedule is sporadic. An example is the packets from the routers in a network. The variable time gaps must be within defined limits.

A preemptive scheduler must take into account three types of tasks (aperiodic, periodic and sporadic) separately.

1. An aperiodic task needs to be preempted only once.
2. A periodic task needs to be preempted after the fixed periods and it must be executed before its next preemption is needed.
3. A sporadic task needs to be checked for preemption after a minimum time period of its occurrence. Usually, the strategy employed by the software designer is to keep the CPU load between (0.7 ± 0.25) for sporadic tasks.

8.10.9 Sporadic Task Model Performance Metric

Let us consider the following parameters.

T_{total} is the total length of the periods for which sporadic tasks occur; e is the total task execution time; T_{av} is the mean periods between the sporadic occurrences; T_{min} is the minimum period between the sporadic occurrences.

Worst-case execution time performance metric, p is calculated as follows for the worst case of a task in a model.

$$p = p_{worst} = (e * T_{total} / T_{av}) / (e * T_{total} / T_{min}).$$

It is because the average rate of occurrence of sporadic task is (T_{total} / T_{av}) and the maximum rate of sporadic task burst is T_{total} / T_{min}.

There are various models to define a performance metric. Three performance metrics for schedule management by the RTOS are: (i) interrupt latencies with respect to the execution times, (ii) CPU load and (iii) worst-case execution time.

8.11 OS SECURITY ISSUES

When a doctor has to dispense to multiple patients, protection of the patients from any confusion in the medication becomes imperative. When an OS has to supervise multiple processes and their access to the resources, protection of memory and resources from any unauthorized writes into the PCB or resource, or mix up of accesses of one by another becomes imperative. *The OS security issue is a critical issue.*

Each process determines whether it has a control of a system resource exclusively or whether it is isolated from the other processes, or whether it shares a resource common to a set of processes. For example, a file or memory blocks of a file will have exclusive control over a process and a free memory space will have the access to all the processes. The OS then configures when a resource is isolated from one process and a resource is shared with a defined set of processes.

The OS should also have the flexibility to change this configuration when needed, to fulfil the requirements of all the processes. For example, a process has control of 32 memory blocks at an instance and the OS configures the system accordingly. Later when more processes are created, this can be reconfigured.

The OS should provide protection mechanisms and implement a system administrator(s)-defined security policy. For example, a system administrator can define the use of resources to the registered and authorized users (and hence their processes).

What about issues of an *application* changing the OS configuration? The OS needs a protection mechanism for itself. An application software programmer can find a hole in the protection mechanism and gain an unauthorized access. Thus the implementation of protection mechanisms and enforcement of security policy for resources is a challenging issue before any OS software designer. The network environment complicates this issue.

Table 8.13 gives the various activities for implementing important security functions.

Table 8.13 Important Security Functions

Function	*Activities*
Controlled resource sharing	Controlling read and write of the resources and parameters by user processes. For example, some resources write only for a process and some read only for a set of processes.
Confinement mechanism	Mechanism that restricts sharing of parameters to a set of processes only.
Security policy (strategy)	Rules for authorizing access to the OS, system and information. A policy example is a communication system having a policy of peer-to-peer communication (connection establishment preceding the data packets flow).
Authentication mechanism	External authentication mechanism for the user and a mechanism to prevent an application run unless the user is registered and the system administrator has (software) authorized . Internal authentication for the process, and the process should not appear (impersonate) like other processes. User authentication can become difficult if the user disseminates passwords or other authentication methods.
Authorization mechanism	User or process allowed using the system resources as per the security policy.
Encryption	A tool to change information to make it unusable by any other user or process without the appropriate key for deciphering it.

The OS security issues are important considerations. Protection of memory and resources from any unauthorized write into the PCB or resource, or mix up of accesses of one by another, becomes imperative for an OS security and protection mechanism.

Summary

- A kernel is a basic unit of any OS that includes the functions for memory allocation and de-allocation, preventing unauthorized memory access, tasks scheduling, IPC, I/O management, interrupts-handling mechanism and device drivers and management. The OS also controls I/O and network subsystems. An OS kernel may also have file management functions and functions for network subsystems (these functions may also be separate from the OS kernels in certain OSes).
- An RTOS has functions for real-time task scheduling and interrupt latency control. The RTOS uses the timers and system clocks. Its functions include time allocation and de-allocation to attain the best utilization in presence of timing constraints to the tasks.
- RTOS uses asynchronous IO functions so that the tasks do not block during the IOs.

- RTOS uses the *preempting scheduling* so that a lower-priority task is forced to block by the scheduler to let a higher-priority task run.
  Time slicing means that each task is allotted a time slice after which it blocks and waits for its turn on the next cycle. In certain cases, other strategies for scheduling, for example, the cooperative scheduling of the multiple tasks as per interrupt sequences, and time slicing can be used.
- An RTOS has functions for handling interrupts. There are three alternative ways used in the RTOSes for response to hardware source calls from the interrupts. An RTOS has functions for registering and deregistering a device driver and also facilitates concurrent processing of the modules, devices, ISRs and tasks.
- An RTOS has mutex and spin lock functions for critical section handling in priority scheduling cases. RTOS may also provide for fixed real-time scheduling.
- RTOS has synchronization mechanism for the tasks using the IPCs and predictable timing and synchronization behaviour in the system. RTOS provide for IPC functions (signals, semaphores, mutexes, queues, mailboxes, pipes and sockets). There is standardization (e.g., POSIX 1003.b) of the RTOS and IPC functions.
- Relative *timing* of the various actions in a preemptive scheduler helps us to optimize the process timings in an application developed using an RTOS.
- RTOS uses the task running in the kernel space to let the code execute in the supervisory mode and thus execute fast (due to no instructions for memory leak and stack overflow checks and kernel space protection check) and reduce the interrupt latencies.
- RTOS uses fixed memory block allocation with predictable memory allocation and de-allocation time.
- OS security issues are important considerations in a system and the protection of memory and resources or PCB from any unauthorized write is essential.

Keywords and their Definitions

Asynchronous IOs	:	IOs in which a process is not blocked due to IO.
Cooperative scheduling	:	A waiting task lets another task run till it finishes.
Critical section run	:	In spite of higher priority, a critical section is allowed to run by a scheduler using the semaphore or spin-lock or lock functions. The critical section is used for shared resources and data between multiple tasks.
Cyclic round robin scheduling	:	A scheduling algorithm in which the tasks are cyclically scheduled in sequence from a list of ready tasks. A time slice is provided in case of round robin cycle.
Fixed real-time scheduling	:	A scheduling strategy in which the time for each task is fixed.
Hard real time system	:	A system in which no task should delay and miss the deadline, the system have minimal interrupt latencies and well-defined time-constraints for each task.
Kernel	:	A basic unit of any OS that includes the functions for processes, memory, task scheduling, IPC, management of devices, IOs and interrupts and may include the file systems and network subsystems in certain OSes.
Lock	:	A function to lock the availability of resources to other tasks at beginning of critical section codes.
OS	:	A system having basic kernel functions of process and memory management, file, IO, device and network management functions and many other functions also.
Pre-empting scheduling	:	A scheduling algorithm in which a higher-priority task is forced (pre-empted) to block by the scheduler to let a higher-priority task run.
Protection mechanism	:	A mechanism at an OS to protect against unauthorized accesses to the resources.

Rate monotonic scheduling	:	A scheduling in which tasks are assigned priorities in accordance with their rate of occurrences of need of their services.
RTOS	:	OS for soft or hard real time tasks with task scheduling with real-time constraints (deadlines) using priority based scheduling, interrupt-latency control, synchronization of tasks with IPCs, and predictable timing and synchronization behaviour of the system.
Round robin	:	A scheduling algorithm in which tasks are scheduled with each one allotted a time slice and run in cyclic in sequence.
Soft real-time system	:	A system in which most not all tasks meet the time constraints and do not miss deadline and a miss can be handled with some delay.
Spin lock	:	An implementation of the spin lock in a task can be by a try. The task tries the lock by a wait loop for the s_{lock} for a defined time t_{wait}, else the task unblocks for the s_{lock}.
Time slicing scheduling	:	It is also called round robin scheduling. A scheduling algorithm in which each task is allotted a time slice after which it is blocked and waits for its turn on the next cycle.
Unlock	:	A function to release the lock at end of the critical section.

Review Questions

1. What should be the goal of an OS?
2. List the layers between *application* and *hardware*.
3. Why does an OS function provide two modes, *user mode* and *supervisory mode*?
4. List the functions of a kernel. What can be the functions outside the kernel?
5. Explain the terms process descriptor and process control block (PCB). What are the analogies in a PCB and TCB?
6. When is a message used and when does a system call for seeking access to system resources?
7. Process or task creation and management are the most important functions of the kernel. Why?
8. A strategy is that the tasks are created at start-up only and creating and deleting tasks later is avoided . Why should it be adopted?
9. Memory allocation and management are the most important functions of the kernel. Why? How does memory allocation differ in RTOS and OS? What is memory locking?
10. List the advantages and disadvantages of fixed and dynamic block allocations by the OS.
11. The kernel controls the access of system resources, CPU, memory, IO subsystems and devices. Why is it needed? Explain the critical section handling with mutexes and spin locks.
12. What is the importance of device management in an OS for an embedded system?
13. Give examples of IO subsystems. Explain the use of asynchronous IOs.
14. Define a network OS. How does a network OS differ from a conventional OS?
15. What are the uses of OS interoperability and portability?
16. How do you choose scheduling strategy for the periodic, aperiodic and sporadic tasks?
17. What are the OS functions at an RTOS kernel?
18. When do you use cooperative scheduling and when preemptive?
19. Comapre two scheduling strategies for the real-time scheduling – preemptive mode and round robin scheduling.

20. What are the cases in which time-slice scheduling helps?
21. List three ways in which an RTOS handles the ISRs in a multitasking environment. What is the advantage of two- or three-level handling of the interrupts? Explain IST.
22. How does a preemption event occur?
23. Real-time system performance metrics are throughput, interrupt latencies, average response times and deadline misses. Explain the importance of each of these metrics.
24. Why should you estimate worst-case latency?
25. Explain the applications of simulation annealing method.
26. What should be the OS security policy?
27. What is the protection mechanism for the OS?
28. OS security issues are important considerations. Protection of memory and resources from any unauthorized write into PCB or resource or mix up of accesses of one by another becomes imperative for an OS security and protection mechanism. Explain each of these considerations.
29. What do you mean by hierarchical RTOS?
30. How is the precedence assignment done for the tasks? How is the precedence assignment algorithm used in dynamic programming?
31. List the best strategies for synchronization between the tasks and ISRs.
32. What is dyanmic program scheduling?

Practice Exercises

33. Give two examples when the scheduler cannot fix the schedules and there is a non-deterministic situation.
34. How do you estimate CPU load in a multitasking system handling sporadic tasks?
35. When do you use CPU load for performance metrics of a real-time system?
36. Give a table showing the differences between traditional OS and the RTOS.
37. Show how the timer functions can be used: (i) to reduce the light level in a mobile phone with full brightness, (ii) to switch off the LCD display in a mobile phone after 15 seconds from the time it was switched on.
38. Show how will you use the mailboxes between display task and other tasks. Which one should you prefer, use of semaphore as shown in the example or use of mailbox.
39. Consider a mobile phone cum PDA device and look at the main menu. Explain how the events of touching the screen at different points on the screen are handled by an RTOS using two-level ISR handling.
40. Give a table showing the differences between three methods of ISR handling in the RTOSs.
41. Show the use of 15 points for the principles of RTOS-based design by taking the example of ACVM.
42. List the priority allocations in ACVM tasks.
43. Show the use of 15 points for the principles of RTOS-based design by taking the example of digital camera.
44. Give the priority allocations in camera tasks.
45. Show the use of 15 points for the principles of RTOS-based design by taking the example of mobile phone device.
46. Give the priority allocations in phone tasks.
47. Show the scheduling method that RTOS can use in case of the VoIP router.
48. Give the priority allocations in smart card tasks.
49. Show the use of semaphores for synchronizing the tasks as cooperative scheduled tasks in a preemptive RTOS.
50. Show the use of semaphores and timer functions for synchronizing the tasks as round robin time-sliced scheduled tasks in a preemptive RTOS.

REAL-TIME OPERATING SYSTEM PROGRAMMING-I: MicroC/OS-II and VxWorks

9

Recall

We have learnt the following important points relating to the traditional OS and RTOS.

- Process is that computational unit which an OS schedules and on request, either by system call or by message passing, the OS lets the process use the resources: CPU, memory, IO subsystems, flash-memory file system, network subsystems and device drivers. Process also means 'task' in a multitasking model of the processes and means 'thread' in a multithreading model of the processes, both controlled by the OS. A process can also consist of several threads, which share a common process structure.
- System structure consists of application software, APIs, additional system software than the one provided at the OS, OS interface, OS, hardware–OS interface and hardware, and an OS or RTOS.
- The application software and APIs are programmed for processes (tasks) and ISRs, ISTs.

- The priorities when executing codes are first the ISRs, then ISTs and then threads of the processes.
- The basic functions (services) of the OS are process management (also means thread management or task management) from creation to deletion, processing resource requests, memory management from allocation and de-allocation, process scheduling, processing and managing IPC (communication among the ISRs, tasks and OS functions), IO subsystems management, management of the file, IO, device, and device drivers and functions for enabling sharing of resources and data.
- Handling of interrupts and scheduling of tasks by the RTOS.
- RTOS has the basic functions of the OS plus functions for real-time task scheduling and interrupt latency control. RTOS uses the timers and system clocks, time allocation and de-allocation to attain best utilization of the CPU time under the given timing constraints for the tasks.
- RTOS provides a predictable timing behaviour of the system (in most cases) and a predictable task synchronization using the priorities allocation and priorities inheritance. RTOS provides for synchronization of ISRs, ISTs and tasks using the IPCs for the hard and soft real-time operations. RTOS provides for asynchronous IOs.
- Interprocess synchronization during concurrent processing of the tasks takes place through signals, semaphores, queues, mailboxes, pipes, sockets, RPCs, timer and event functions.
- Basic strategies for scheduling the multiple tasks are pre-empting, round robin time slice and cooperative scheduling. The RTOS basic strategy is preemptive scheduling.
- Principles of basic design using the RTOS, and important points that are taken care of during coding for synchronization between the processes (ISRs, functions, tasks and scheduler functions).

The goals of any embedded software, and hence of RTOS, are perfection and correctness. The reader must have now realized that there is a great deal of functions involved in real-time programming. The objective of this chapter is to explain thoroughly the two popular RTOSes that are used for programming and provide the OS functions which, significantly reduce the time required to design an embedded system.

LEARNING OBJECTIVES

We will learn the following in this chapter.

(i) Basic functions and types of RTOSes.

(ii) RTOS µCOS-II (referred as MUCOS in the text) through 20 examples— Examples 9.1 to 9.20. What arguments are passed and what values are returned for each given MUCOS function will be explained. Learning the use of functions in the MUCOS is important for a reader even if another RTOS is used later. This will help greatly in understanding the advanced, sophisticated embedded RTOSes later on.

(iii) VxWorks from Wind River® Systems is also an RTOS for sophisticated embedded systems. It has powerful tool support. The features in VxWorks are explained by seven examples— Examples 9.21 to 9.27. Differences between the VxWorks semaphores, mailboxes and queues with respect to that of MUCOS will be made clear.

Chapter 10 will describe the Windows CE, OSEK and real-time Linux (RTLinux).

9.1 BASIC FUNCTIONS AND TYPES OF RTOSes

A complex multitasking embedded system design requires the following:

1. Integrated development environment
2. Task functions in embedded C or embedded C++
3. Real-time clock-based hardware and software timers
4. Scheduler
5. Device drivers and device manager
6. Functions for IPCs using the signals, event flag group, semaphore-handling functions and functions for the queues, mailboxes, pipe and sockets.
7. Additional functions, for example, TCP/IP or USB or Bluetooths or WiFi or IrDA and GUIs.
8. Error and exception handling functions.
9. Testing and system debugging software for testing RTOS as well as developed embedded application.

Figure 9.1(a) shows the basic functions expected from the kernel of an RTOS. The RTOS's have the following features in general.

1. Basic kernel functions and scheduling: pre-emptive or pre-emptive plus time slicing.
2. Priorities definitions for the tasks and IST.
3. Priority inheritance feature with option of priority ceiling feature.
4. Limit for number of tasks.
5. Task synchronization and IPC functions.
6. IDE consisting of editor, platform builder, GUI and graphics software, compiler, debugging and host target support tools.
7. Device imaging tool and device drivers.

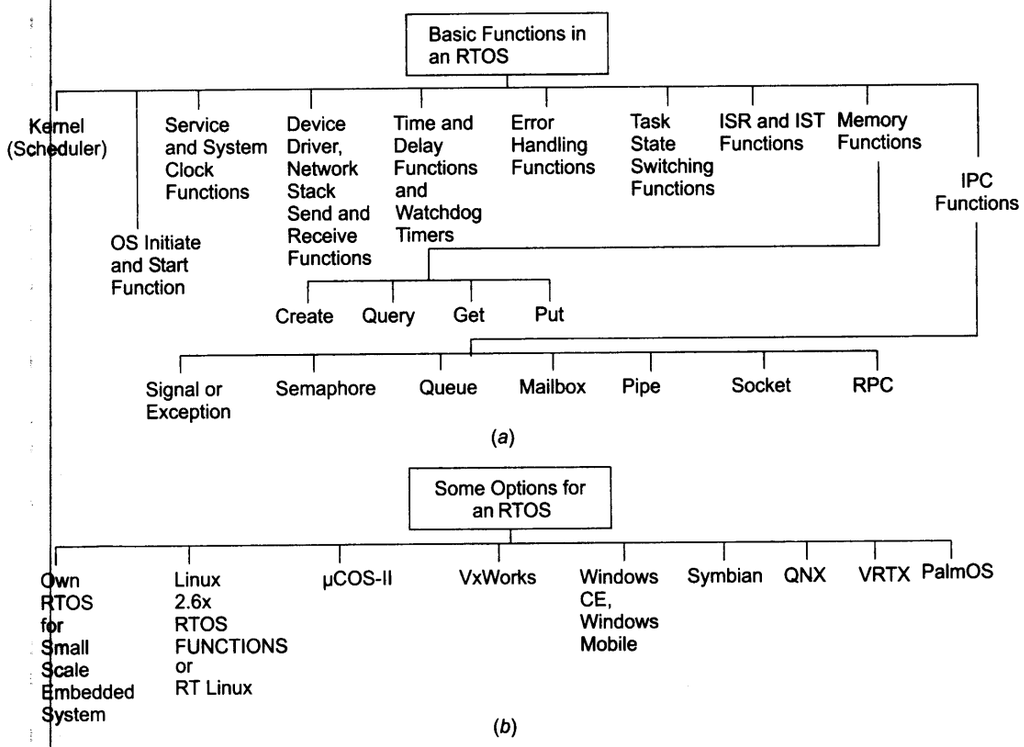

Fig. 9.1 (*a*) Basic functions expected from the kernel of an RTOS (*b*) Common options available for selecting a real-time operating system (RTOS)

8. Support to the clock, time and timer functions, POSIX, asynchronous IOs, memory allocation and deallocation system, file systems, flash systems, number of processors, TCP/IP, network, wireless and bus protocols, development environment with Java and componentization (reusable modules for different functions), which leads to small footprint (small-sized RTOS codes placed in the ROM image).
9. Support to number of processor architectures.

Section 9.1.1 gives the host-target and self-host approaches to development of an application. Section 9.1.2 gives the types of RTOSes.

9.1.1 Host and Target-Based, and Self-Host-Based Development Approaches

A real-time or non-real-time application-development approach is the **host target approach**. A host machine (computer) for example, a PC uses a general purpose OS, for example, Windows or Unix for system development. The target connects by a network protocol, for example, TCP/IP during the development phase. The developed codes and the target RTOS functions first connect a target. The target with downloaded codes finally disconnects and contains a small-size footprint of RTOS. For example, the target does not download host machine resident compiler, cross-compiler, editor for programs, simulation and debugging programs and MMU support.

The **self-host development approach** is that the same system with full RTOS is used for development on which the application runs. This also does not require cross-compilation. When application codes are ready, the required RTOS function codes and application codes are downloaded into the ROM of the target board.

9.1.2 Types of RTOSes

Some options for selecting an RTOS are shown in Figure 9.1(b). Followings are the types of RTOSes.

In-House Developed RTOS In-house RTOS has the codes written for the specific need, and application or product and customizes the in-house design needs. Generally either a small-level application developer uses the in-house RTOS or a big research and development company uses the codes built by the in-house group of engineers and system integrators.

Broad-based Commercial RTOS A readily available broad-based commercial RTOS package offers the following advantages.

1. Provides an advantage of availability of off the self thoroughly tested and debugged RTOS functions.
2. Provides several development tools. Development tools consist of tools for the source-code engineering, testing, simulating and debugging are also available with the RTOS package. When designing a mission critical real-time application, lack of appropriate error-handling capability or an appropriate RTOS or a testing and debugging tool causes data loss. Even hardware loss may be caused.
3. Support to many processor architectures, for example, ARM as well as x86, MIPS and SuperH.
4. Support to development of GUIs in the system.
5. Support to many devices, graphics, network connectivity protocols and file systems.
6. Support to device software optimization (DSO). It is a recently available concept in a few RTOSes.
7. Provides error and exceptional handling functions, which can be ported directly as these are already well tested by thousands of users.
8. Not only simplifies the coding process greatly for a developer but also helps in building a product fast; it aids in building robust and bug-free software by thorough testing and simulation before locating the codes into the hardware.
9. Saves large amount of development time for RTOS tools and in-house documentation. Saving of time results in little time to market an innovative and new product.
10. Saves maintenance cost.
11. Saves cost of keeping in-house engineers.

General Purposes OS with RTOS Embedded Linux or Windows XP is general purpose OS. They are not componentized. Footprint (the code that goes as ROM image) is not reducible. The tasks are not assignable priorities. They offer powerful GUIs, rich multimedia interfaces and have low cost.

The general purpose OS can be used in combination with the RTOS functions. For example, RTLinux is a real-time kernel over the Linux kernel. Another example is 'Windows XP Embedded' for x86 architecture.

Special Focus RTOS Special focus RTOS is used with specific processors like ARM or 8051 or DSP, for example, OSEK for automotives or Symbian OS for the mobile phones.

9.2 RTOS µCOS-II

One popular RTOS for the embedded system development is µCOS-II. For non-commercial use, RTOS µCOS-II is also a freeware. Jean J. Labrosse designed it in 1992 and nowadays µCOS-II is well developed for

a number of applications. It is available from Micrium (www.micrium.com). Its name µCOS-II is derived from Micro-Controller Operating System. It is also popularly known as MUCOS or MicroCOS or UCOS. (*Note:* MUCOS is pronounced as MU-C-OS)

Micrium describes MUCOS as portable, ROMable, scalable, preemptive, real-time and multitasking kernel. MUCOS has been used in over thousands of applications, including automotive, avionics, consumer electronics, medical devices, military, aerospace, and networking, and systems-on-a-chip development. MUCOS has an elegant code and is said to offer best high quality/performance ratio. Its source code has been certified by Department of Defense, USA for use in Avionics and in medical applications.

It has a precertifiable software component for safety critical systems, including avionics system ADO-178B and EUROCAE ED-12B, medical FDA 510(k) and IEC 61058 standards for transportation and nuclear systems, respectively.

Using this RTOS has another advantage. It has full source code availability and has been elegantly and very well documented in the book by its designer (refer to the printed book references in Appendix 2).

MUCOS codes are in C and a few CPU-specific modules are in the assembly. Its code ports on many processors that are commonly used in the designing of embedded systems. MUCOS is real-time kernel with additional support as follows.

1. µC/BuildingBlocks [an embedded system building blocks (software components) for hardware peripherals, e.g., clock (µC/Clk) and LCD (µC/LCD)].
2. µC/FL (an embedded flesh memory loader).
3. µC/FS (an embedded memory file system).
4. µC/GUI (an embedded GUI platform).
5. µC/Probe (a real-time monitoring tool).
6. µC/TCP-IP (an embedded TCP/IP stack).
7. µC/CAN (an embedded controller area network bus).
8. µC/MOD (an embedded modbus).
9. µC/USB device and µC/USB host (embedded USB-devices framework).

Source Files MUCOS has 10,000 plus lines of codes. There are two types of source files. Master header file includes the '#include' preprocessor commands for all the files of both types. It is referred as 'includes.h'. Every C file has the command, #include'INCLUDES.H'.

1. *Processor-dependent source files:* Two header files at the master are the following: (i) os_cpu.h is the processor definitions header file. (ii) The kernel building configuration file is os_cfg.h. Further, two C files are for ISRs and RTOS timer, specifying os_tick.c and processor C codes os_cpu_c.c. Assembly codes for the task switching functions are at os_cpu_a.s12 (for 68HC12 microcontroller). For other microcontrollers, there are similar assembly code files, for example, os_cpu_a.s51 for 8051.
2. *Processor-independent source files:* Two files, MUCOS header (included in master) and C files, are ucos_ii.h and ucos_ii.c. The files for the RTOS core, timer and task files are os_core.c, os_time.c and os_task.c. The memory-partitioning, semaphore, queue and mailbox codes are in os_mem.c, os_sem.c, os_q.c and os_mbox.c, respectively.

A feature of MUCOS is adaptation of a systematic naming convention that helps program design with clear understanding of the code. The naming convention is as follows.

a. OS or OS_ (OS followed by underscore) when used as a prefix denotes that the function or variable is a MUCOS-operating system function or variable. For example, OSTaskCreate ( ) is a MUCOS function that creates a task. OS_NO_ERR is a MUCOS macro that returns true in case no error is reported from an OS function. OS_MAX_TASKS is a constant for the maximum number of tasks in the user application (The user in the preprocessor definitions defines this constant).

b. MUCOS is scalable OS. (Only the OS functions that are necessary become part of the application codes, thus having reduced memory requirements). The functions needed for task servicing, IPC and so on must be predefined in a configuration file included in the user codes (Refer to Example 9.7 Steps 1 and 2 for a clear understanding of configuration-setting codes).
c. For multitasking, MUCOS employs a preemptive scheduler (Section 8.10.3).
d. MUCOS has system-level functions. These are for RTOS initiation and start, system clock ticks (interrupts) initiation and the ISR enter and exit functions. (Section 9.2.1 Table 9.1) For the critical section, MUCOS has (i) interrupts disabling and enabling functions that execute at entering and exiting the section, respectively, (ii) lock and unlock functions in kernel that execute at entering and exiting the section, respectively, and that do not disable the interrupts, and (iii) semaphore functions, which can be used as mutex functions that execute at entering and exiting the critical section, respectively, and that disables the preemption by a higher priority task, which is sharing that mutex in its critical section (Refer to Sections 7.7.2 and 8.10.3).
e. MUCOS has task service functions (e.g., task creating, running, suspending and resuming) (Table 9.2).
f. MUCOS has task delay functions (Table 9.3).
g. MUCOS has memory allocation functions for creating and partitioning into blocks, getting a block, putting into the block and querying during debugging at a block (Table 9.4).
h. MUCOS has IPC functions. These are as per Tables 9.5, 9.6 and 9.7, respectively. MUCOS IPCs use the semaphores, queues and mailboxes (Sections 7.11 to 7.13).
i. MUCOS has semaphore functions, which are usable like the event-signalling flags, shared resource acquiring keys or counting semaphores (Section 7.7.5). Table 9.5 lists these.
j. MUCOS has event flag-group functions (Section 8.4) also, where a task waits for any or all of the flags setting in an event flag groups. These functions enable event signalling from OR or AND operation for WAIT_ANY or WAIT_ALL operation on number of flags set in the group. An ISR or task can set a flag in the group. An ISR is not allowed to wait (pend) for the event(s).
k. MUCOS has mailbox functions. A MUCOS mailbox has one message pointer per mailbox. (Section 7.13). There can be any number of messages as MUCOS sends only the pointer (start address of the message) into the mailbox. Table 9.6 lists these.
l. MUCOS has queue functions. A queue permits an array of message pointers per queue from which messages retrieve in the FIFO method (Section 7.12). There can be any number of messages in a queue element as MUCOS sends only the address of message into queue. Table 9.7 lists these.

The following seven subsections 9.2.1 to 9.2.7 describe the aforementioned MUCOS functions. For functions in each of the seven Tables, 9.1 to 9.7, these sections give the details of values that are returned by the MUCOS functions and the details of parameters (arguments) that are passed by value or reference to a MUCOS function.

9.2.1 System-Level Functions

We first use initiate function for initiating the use of MUCOS, and then use start function for starting the MUCOS multitasking functions. These functions are OSInit and OSStart, respectively. The start function is used after the creation of at least one task, which can be called Start_Task or First_Task. (As we will read later, a strategy is that remaining tasks are created in the First_Task.)

Recall Section 8.3. We first initiate the system timer clock ticks (and interrupts). MUCOS RTOS has system functions that should be executed when entering and exiting the ISR.

Recall Table 7.1 and Section 7.8.2. MUCOS RTOS has system functions for disabling and enabling interrupts that can be executed when entering a critical section of a task or ISR and exiting the critical section or ISR (Sections 7.7.2 and 8.10.3). Table 9.1 gives these RTOS system-level functions.

Table 9.1 Real-time Operating System (RTOS) and System Clock Initiate, Start, Interrupt Service Routine (ISR) and Disable-Enable Interrupt Functions[1]

Prototype Functions	*When is this OS Function Called?*
void OSInit (void)	At the beginning prior to OSStart ( )
void OSStart (void)	After the OSInit ( ) and task-create functions
void OSTickInit (void)	In first task function, which executes once only, this function is to initialize the system timer ticks (system clock interrupts)
void OSIntEnter (void)	Recall Section 8.7.1. Just after the start of the ISR codes OSIntExit must call just before the return from the ISR.[2] (enter and exit functions form a pair)
void OSIntExit (void)	After the OSIntEnter ( ) is called just after the start of the ISR codes and OSIntExit is called just before the return from the ISR (enter and exit functions form a pair)[3]
OS_ENTER_CRITICAL	Macro to disable interrupts (Section 7.8.2)
OS_EXIT_CRITICAL	Macro to enable interrupts (enter and exit functions form a pair in the critical section)
OSSchedLock[4] ( )	To lock scheduling of the task (Sections 7.11.1 and 8.10.3)
OSSchedUnlock[4] ( )	Unlock scheduling of the tasks (Sections 7.11 and 8.10.3)

[1] Functions in this table pass no arguments and returns void.
[2] There is a global variable, OSIntNesting. It increments after the enter call. (We should not increment directly though it can be done. Let it increment automatically on enter to an ISR).
[3] Global variable OSIntNesting decrements on exit call. (We should not decrement directly though it can be done. Let it decrement automatically on exit from an ISR).
[4] OSSchedlock ( ) enables any task critical section run without preemption if it executes before entering critical section and runs OSSchedUnlock ( ) after end of the task critical section. Task can however be interrupted by an ISR. Task locking the scheduler should not suspend itself before unlocking.

1. *Initiating the OS before starting the use of the RTOS functions*. Function *void* OSInit (*void*) is used to initiate the OS. Its use is compulsory before calling any OS kernel functions. It returns no parameter. An exemplary use as a function is as follows:

Example 9.1

```
1. /* Start executing the codes */
void main (void) {2. /* Initiate MUCOS RTOS to let us use the OS kernel functions */
OSInit ( );
3. /* Create (Define Identity, stack size and other TCB parameters for the tasks using RTOS Functions */
.
.
4. /* Create semaphore, queue and mailboxes, etc. */
.
.
}.
```

2. *Starting the use of RTOS multitasking functions and running the tasks.* Function *void* OSStart (*void*) is used to start the initiated OS and created tasks. Its use is compulsory for the multitasking OS kernel operations. It returns no parameter. An exemplary use as a function is as follows:

Example 9.2

```
1. /* Start executing the codes from Main*/
void main (void) {2. /* Initiate MUCOS RTOS to let us use the OS kernel functions */
OSInit ( );
3. /* Create tasks and inter-process communication variables by defining their identity, stack size and other
TCB parameters. */
.
.
4. /* Start MUCOS RTOS to let us use RTOS control and run the created tasks and inter-process
communication. */
OSStart ( );
/* An infinite while-loop follows in each task. So there is no return to the main ( ) from the
RTOS. */
}/ * End of the Main function. */
```

3. *Starting the use of RTOS System clock.* Function *void* OSTickInit (*void*) is used to initiate the system clock ticks and interrupts at regular intervals as per OS_TICKS_PER_SEC predefined during configuring the MUCOS. Its use is compulsory for the multitasking OS kernel operations when the timer functions are to be used. It returns or passes no parameter. An exemplary use will be shown in steps 2 and 10 of Example 9.7.
4. *Sending message to RTOS kernel for taking control at the start of an ISR.* Function void OSIntEnter (void) is used at the start of an ISR. It is for sending a message to RTOS kernel for taking control (Section 8.7.1). Its use is compulsory to let the multitasking OS kernel, control the nesting of the ISRs in case of occurrences of multiple interrupts of varying priorities. It returns no parameter. An exemplary use as a function is as follows:

Example 9.3

```
1. /* Start executing the codes of an ISR*/
ISR_A ( ) {
2. /* sending message to RTOS kernel for taking control of ISR_N from nested ISRs loop. Increment
OSIntNesting, a global variable */
OSIntEnter ( );
3. /* Codes for servicing of the ISR by calling a task. */
.
.
```

5. *Sending a message to RTOS kernel for quitting the control at return from an ISR.* Function void OSIntExit (void) is used just before the return from the running ISR. It is for sending a message to RTOS kernel for quitting control from the nesting loop. Its use is compulsory to let the OS kernel quit the ISR from the nested loop of the ISRs. It returns no parameter. An exemplary use as a function is as follows:

Example 9.4

1. to 3. As in Example 9.3

.
.

4. /* Sending message to RTOS kernel for quitting the control of ISR_A from the nested loop. Decrement OSIntNesting, a global variable */
OSIntExit ( );
}/ * End of the ISR function. */

6. *Sending a message to RTOS kernel for taking control at the start of a critical section.* Recall Example 8.21 in Section 8.10.3. A macro-function OS_ENTER_CRITICAL is used at the start of critical section in a task or ISR. It is for sending a message to MUCOS kernel and disabling the interrupts. Its use is compulsory to let the OS kernel take note of and disable the interrupts of the system. It returns no parameter. An exemplary use as a function is as follows.

Example 9.5

1. /* Start executing the codes of a task or an ISR*/
task_A ( ) {
2. /* Codes for servicing of the task. */

.
.

3. /* Sending a message to RTOS kernel and disabling the interrupts. */
OS_ENTER_CRITICAL;
4. /* Run critical section codes as follows. */

.
.

5. /* Codes for Exiting the service*/
}

7. *Sending a message to RTOS kernel for quitting the control at the return from a critical section.* Macro-function OS_EXIT_CRITICAL is used just before the return from the critical section. It is for sending a message to RTOS kernel for quitting control from the section. Its use is compulsory to let the OS kernel quit the section and enable the interrupts to the system. It returns no parameter. An exemplary use as a function is as follows:

Example 9.6

1. to 4. As in Example 9.5

.
.

5. /* Sending a message to RTOS kernel for quitting the control of critical section and enabling the interrupts. */
OS_EXIT_CRITICAL;
}/ * End of the ISR function. */

8. *Locking OS scheduler.* OSSchedlock ( ) disables preemption by a higher-priority task. This function inhibits preemption by higher-priority task, but does not inhibit interrupt. If an interrupt occurs then locking enables return of OS control to that task, which executes this function. The control returns to the task after any ISR completes.
9. *Unlocking OS scheduler.* OSSchedUnlock ( ) enables preemption by higher priority task. Enables return of OS control to the high priority task after the execution of OSSchedUnlock. In case of any interrupt occurring after executing OSSchedUnlock and after the end of the ISR, the higher-priority task, which is ready will execute on return from the ISR.

9.2.2 Task Service and Time Functions and their Exemplary Uses

MUCOS service functions for the tasks and time are as per Table 9.2. Service functions mean the functions to create task, suspend and resume, and time setting and time retrieving (getting) functions. Time functions set time and get time in terms of the number of system clock ticks.

The declarations for the variable and prototype assignments for task functions are done in the preprocessor commands. Steps 1 and 2 of Example 9.7 show these preprocessor commands. The codes in Steps 1 and 2 codes are saved in a configuration file, which is included in the source code before compilation. These steps configure the MUCOS before they are used.

We shall see in the following examples that there is an infinite loop in every task function. This is a characteristic way of coding the tasks for preemptive scheduling. From the infinite loop, how will the CPU control return to MUCOS? In other words, how does context switching occur in the OS and how does the OS then activate the task switch to a higher-priority task? The CPU control returns to MUCOS (or in any other preemptive scheduler) as soon as one of the following situations arises.

1. Any interrupt event including the occurrence of the timer tick interrupt. Refer to Example 9.7, Step 10 (time set for the interrupt every 10 ms in Step 2).
2. On suspending the presently running task by calling OSTaskSuspend as in Example 9.8, Step 12.
3. As soon as any OS function, say a time delay function in Table 9.3 or a semaphore-pending function in Table 9.4 is called, the scheduler switches context, preempts and thus passes the control to other higher priority assigned task by activating task switch.

How does the control of CPU return to a preempted task because of an infinite loop existing in pre-empting higher-priority task also? It must return by an appropriate coding. For example, refer to the code in Step 12 in Example 9.8. Here, the FirstTask is of priority 8 (the highest available to a user task). It suspends itself from the loop at Step 12.

1. *Creating a task.* Function unsigned byte OSTaskCreate (void (*task) (void *taskPointer), void *pmdata, OS_STK *taskStackPointer, unsigned byte taskPriority) is explained as follows.

A preemptive scheduler preempts a task of higher priority. Therefore, each user-task is to be assigned a *priority*, which must be set between 8 and OS_MAX_TASKS – 9 (or 8 and OS_LOWEST_PRIO – 8). OS reserves eight highest and eight lowest priority tasks for its own functions. Total number of tasks that MUCOS manages can be up to 64. Mucos task *priority* is also the task identifier. There is no task ID-defining function in MUCOS because each task has to be assigned a distinct *priority*.

If the maximum number of user tasks is eight, then OS_MAX_TASKS is 24 (including eight system-level tasks and 8 lowest priority system-tasks), the priority must be set between 8 and 15. The OS_LOWEST_PRIO must be set at 23 for eight user tasks of priority between 8 and 15, because MUCOS will assign priorities 16 to 23 to the 8 lowest priority system level tasks. The *priorities* 0 to 7 or 16 to 23 will then be for MUCOS internal uses.

Table 9.2 Service and System Time Functions for the Tasks

Prototype of Functions	*What are the Parameters Returned?*	*What are the Parameters Passed?*	*When is this Operating System (OS) Function Called?*
unsigned byte OSTaskCreate (void (*task) (void *taskPointer), void *pmdata, OS_STK *taskStackPointer, unsigned byte taskPriority)	*RA*	*PA*	Must call before running a task
unsigned byte OSTaskSuspend (unsigned byte taskPriority)	*RB*	*PB*	Called for blocking a task
unsigned byte OSTaskResume (unsigned byte taskPriority)	*RC*	*PC*	Called for resuming a blocked task
void OSTimeSet (unsigned int *count*)	*None*	*PD*	Each *count* represents the system clock ticks. When system time is to be set it is set by an initial *count* value
unsigned int_OSTimeGet (void)	*RE*	*None*	Find the present *count* so that the system time is read

Unsigned int means a 32-bit unsigned integer. Abbreviations used in columns 2 and 3 are explained in text.

OS_LOWEST_PRIO and OS_MAX_TASKS are user-defined constants in preprocessor codes that are needed for configuring the MUCOS for the user application. Defining unnecessarily 20 user tasks when actually 4 tasks are created by the user is to be avoided because more OS_MAX_TASKS means unnecessarily higher memory space allocation by the system to the user tasks.

Task parameters passing *PA*:

(a) *taskPointer is a pointer to the codes of task being created.

(b) *pmdata is pointer for an optional message data reference passed to the task. If none, we assign it as NULL.

(c) * TaskStackPointer is a pointer to the stack of task being created.

(d) TaskPriority is the task priority and must be within 8 to 15, if macro OS_LOWEST_PRIO sets the lowest priority equal to 23.

Returning *RA*: The lowest priority of any task OS_ LOWEST_PRIO is 23. For the application program, task priority assigned must be within 8 to 15. The function OSTaskCreate ( ) returns the following: (i) OS_NO_ERR, when creation succeeds; (ii) OS_PRIO_EXIST, if priority value that passed already exists; (iii) OS_PRIO_INVALID, if priority value that passed is more than the OS_PRIO_LOWEST; (iv) OS_NO_MORE_TCB returns, when no more memory block for the task control is available.

A task can create other tasks, but *an ISR is not allowed to create a task.* An exemplary use is in creating a task, Task1_Connect, for a connecting task. OSTaskCreate (Task1_Connect, void (*) 0, (void *) *Task1_ConnectStack [100], 10)

Task parameters passed as arguments are as follows.

(a) Task1_Connect, a pointer to the codes of Task1_Connect for task being created.

(b) The pointer for an optional message data reference passed to task is NULL.

(c) * Task1_ConnectStack is a pointer to the stack of Task1_Connect and it is given size = 100 addresses in the memory.

(d) TaskPriority is task priority allotted at 10, the highest but two that can be allocated.

It will generate error parameters, OS_NO_ERR = true in case creation of Task1_Connect task succeeds. OS_PRIO_EXIST = true, if priority 8 task is already created and exists. OS_PRIO_INVALID = true, if passed priority parameter is higher than OS_LOWEST_PRIO – 8. OS_NO_MORE_TCB = false, when TCB is available for Task1_Connect (TCB definition is given in Section 7.3).

Example 9.7

```
1. /* Preprocessor MUCOS configuring commands to define OS tasks service and timing functions as
enabled and their constants*/
#define OS_MAX_TASKS 24 /* Let maximum number of tasks in user application be 8. */
#define OS_LOWEST_PRIO 23 /* Let lowest priority task in the OS be assigned priority
= 23 for 8 user application tasks of priorities between 8 and 15. */
#define OS_TASK_CREATE_EN 1/* Enable inclusion of OSTaskCreate ( ) function */
#define OS_TASK_DEL_EN 1/* Enable inclusion of OSTaskDel ( ) function */
#define OS_TASK_SUSPEND_EN 1/* Enable inclusion of OSTaskSuspend ( ) function */
#define OS_TASK_RESUME_EN 1/* Enable inclusion of OSTaskResume ( ) function */
.
.
/* End preprocessor MUCOS configuring commands */
2. /* Specify all user prototype of task-functions to be scheduled by MUCOS */
/ * Remember: Static means permanent memory allocation */
static void FirstTask (void *taskPointer);
static void Task1_Connect (void *taskPointer);
static OS_STK FirstTaskStack [FirstTask_StackSize];
static OS_STK Task1_ConnectStack [Task1_Connect_StackSize];
/* Define public variable of the task service and timing functions */
#define OS_TASK_IDLE_STK_SIZE 100 /* Let memory allocation for an idle state task stack size be
100*/
#define OS_TICKS_PER_SEC 100 / * Let the number of ticks be 100 per second. The system clock will
interrupt and thus tick every 10 ms to update the set counts and to transfer control to the MUCOS. */
#define FirstTask_Priority 8 /* Define first task in main priority */
#define FirstTask_StackSize 100 /* Define first task in main stack size */
#define Task1_Connect_Priority 10 /* Define Task1_Connect priority */
#define Task1_Connect_StackSize 100 /* Define Task1_Connect stack size */
.
.
.
3. /* The codes of the application starts from main*/
void main (void) {
4. /* Initiate MUCOS to let us use the OS kernel functions */
OSInit ( );
5. /* Create first task that must execute once before any other. Task creates by defining its
identity as FirstTask, stack size and other TCB parameters. */
```

*OSTaskCreate (FirstTask, void (*) 0, (void *) &FirstTaskStack [FirstTask_StackSize],* FirstTask_Priority*);*
/* In this example, FirstTask will be creating other tasks. May create other main tasks and inter-process communication variables if these must also execute at least once after the FirstTask. */
.
.
6. /* Start MUCOS RTOS to let us use RTOS control and run the created tasks */
OSStart ( ); /* Infinite while-loop is there in each task. So there is no return to *main* from the RTOS function OSStart ( ). */
}/ *** End of the Main function ***/
7. /* The codes of the application first task that creates in Main*/
static void FirstTask (void *taskPointer) {
8. /* Create a Task as per Step 2 defined by task identity Task1_Connect, stack size and other TCB parameters. */
*OSTaskCreate (Task1_Connect, void (*) 0, (void *) &Task1_ConnectStack [Task1_Connect_StackSize],* Task1_Connect_Priority*)*
9. /* Create other tasks and inter-process communication variables. */
.
.
10. /* Start Timer Ticks for using timer ticks later. */
OSTickInit ( ); /* Function for initiating system clock ticks at the configured time in the MUCOS configuration preprocessor commands in Step 1 */
11. while (1) {/* Infinite loop of FirstTask */
.
.
12. }; /* End infinite loop */
13.} /End of FirstTask Codes. */
The FirstTask is the main and only task created in step 5. The first task calls a function for the timer initiation (step 10). This is required as the RTOS timer functions are needed in the application. A task function *Task1_Connect* codes creates in step 8. All other tasks that are created are the private tasks using OSTaskCreate function within the first task function.
14. /* The codes for the Task1_Connect*/
static void Task1_Connect (*taskPointer) {
15. /* Initial assignments of the variables and pre-infinite loop statements that execute once only*/
.
.
.
16. /* Start an infinite while-loop. */
while (1) {
17. /* Codes for Task1_Connect*/
.
.
18./; /* End of while loop*/
19.}/ * End of the Task1_Connect function */

2. *Suspending (blocking) a task.* Function unsigned byte OSTaskSuspend (unsigned byte taskPriority)

Task parameters passing *PB*: taskPriority is the task priority and must be within 8 to 15 for 8 user-tasks.

Returning *RB*: The function OSTaskSuspend () returns the error parameters OS_NO_ERR when the blocking succeeds. OS_PRIO_INVALID, if priority value that passed is more than 15, the OS_PRIO_LOWEST constant value. OS_TASK_SUSPEND_PRIO, if priority value that passed already does not exists. OS_TASK_SUSPEND_IDLE, if attempting to suspend an idle task that is illegal.

An exemplary use is in blocking the task Task1_Connect of priority = *Task1_Connect_Priority* is as follows: OSTaskSuspend (*Task1_Connect_Priority*). Task parameter passed as argument is 6. Recall *Task1_Connect_Priority* was assigned 10 earlier in the step 2 of Example 9.7. The following error parameters will be returned by this function.

(a) OS_NO_ERR = true, when the blocking succeeds.
(b) OS_PRIO_INVALID = false, as 8 is a valid priority and is not more than OS_PRIO_LOWEST.
(c) OS_PRIO_LOWEST = 23.
(d) OS_TASK_SUSPEND_PRIO = false, as priority value that passed already does exist.
(e) OS_TASK_SUSPEND_IDLE = false, when attempting to suspend a task that was not an idle task. MUCOS executes idle task OSTaskIdle ( ) when all tasks are either waiting for timer expiry or for an IPC, e.g., semaphore IPC.

Example 9.8

1 to 11. /* Steps Codes as in Example 9.7 */
12. /* Suspend FirstTask, as it was for initiating the timer ticks (interrupts), creating the user application tasks, and was to be run only once */
***OSTaskSuspend (FirstTask_Priority);* /*Suspend First Task and control of the RTOS passes to other tasks waiting execution, for example to *Task1_Connect*. */**
} /* End of while loop */
13.) /* End of FirstTask Codes */
14 to 19. /* Steps Codes as in Example 9.7 */

3. *Resuming (enabling unblocking) a task.* Function unsigned byte OSTaskResume (unsigned byte taskPriority) resumes a suspended task.

Task parameters passing *PC*: taskPriority is the task priority of that task which is to resume and must be within 8 to 15 when OS_LOWEST_PRIO is 23 and number of user-tasks = 8.

Returning *RC*: The function OSTaskResume ( ) returns the OS_NO_ERR when the blocking succeeds. OS_PRIO_INVALID, if priority value that passed is more than 23, the OS_ LOWEST_PRIO constant value. OS_TASK_RESUME_PRIO, if priority value that passed already resumed. OS_TASK_NOT_SUSPENDED, if attempting to resume a not suspended (blocked) task.

An exemplary use is in un-blocking Task1_Connect of priority = *Task1_Connect_Priority* is as follows: OSTaskResume (*Task1_Connect_Priority*). Task parameter passed as argument is 10, as *Task1_Connect_Priority* = 10. The following error parameters will be returned by the task-resuming function.

(a) OS_NO_ERR = true, when the un-blocking succeeds and task of priority 8 reaches the running state.
(b) OS_PRIO_INVALID = false, as 8 is a valid priority and is not more than OS_ LOWEST_PRIO.
(c) OS_ LOWEST _PRIO = 23.

(d) OS_TASK_RESUME_PRIO = false, as priority value that passed already resumed.
(e) OS_TASK_NOT_SUSPENDED = false, when attempting to resume a task that was not suspended.

Example 9.9

```
1. to 19. /* Steps as per Example 9.7 codes for Task1_Connect Function. */
20. /* Codes of other task, Task_N */
.
.
Step n /* Resume Task1_Connect. Control. */
OSTaskResume (Task1_Connect_Priority);
Step n+1. }; /* End of while loop*/
Step n+2.}/ * End of the Task_N function.*/
```

4. *Setting time in system clock.* Function void OSTimeSet (unsigned int count) returns no value. Passing parameter, PD as argument is given next.

PD: It passes a 32-bit integer for the *count* (set the number of ticks for the current time that will increment after each system clock tick in the system).

An exemplary use is a function OSTimeSet (to preset the time). The function OSTimeSet (0) sets the present *count* = 0. *Caution*: it is suggested that OSTimeSet function should be used before the OSTickInit function and only once then never be used within a task function, as some other functions that rely on the timer will malfunction. Let the OS timer clock *count* continue to be used as in a free running counter. There is little need later on of using the set time function. This is because at any instant, the time can be read using a get function (Example 9.11) and at any other instant, it can be defined again by adding a value to this time. Example 9.10 uses the set function in the FirstTask.

Example 9.10

```
void FirstTask (*taskPointer) {
1. to 9. /* The codes up to OSTickInit ( ) in Example 9.7*/
10. /* Set the timer number of ticks to 0 */
unsigned int presetTime = 0;
OSTimeSet (presetTime);
OSTickInit ( );
11. /* Other codes of FirstTask */
.
.
}/ * End of the FirstTask function.
```

5. *Getting time of system clock.* Function unsigned int OSTimeGet (void) returns current number of ticks as an unsigned integer. The passing parameter as argument is none.

RE: Returns 32-bit integer, current number of ticks at the system clock.

Example 9.11

1: to 20. /* The codes as per steps 1 to 19 in Example 9.7 and the codes as for another task function */

.

.

unsigned int *currentTime = OSTimeGet ( );*

2. /* Other codes of the task after determining *current time* */

.

.

}/ * End of Task-function. */

9.2.3 Time Delay Functions

MUCOS time delay functions for tasks are as per Table 9.3.

Table 9.3 Time Delay Functions for Tasks

Prototype Function	*What are the Parameters Returned?*	*What are the Parameters Passed?*	*When is this Operating System (OS) Called?*
void OSTimeDly (unsigned short delayCount)	None	*PF*	When a task is to be delayed by count inputs equal to delayCount – 1. The task, which delays is the one in which this function executes.[1]
unsigned byte_OSTimeDlyResume (unsigned byte_taskPriority)	*RG*	*PG*	When a task of priority = taskPriority is to resume before the preset delay, which was by an amount defined by delayCount or (hr, mn and ms) and presently is in blocked state
void OSTimeDlyHMSM (unsigned byte hr, unsigned byte mn, unsigned byte sec, unsigned short ms)	*RH*	*PH*	When need is to delay and block a task for hr hours, mn minutes, sec seconds and ms milliseconds[2]

Abbreviations used in columns 2 and 3 are explained in text.

[1] Task cannot be delayed than 65,535 system clock count inputs (ticks) by function OSTimeDly.

[2] Task cannot resume later by OSTimeDlyResume ( ) if delay (in hours, minutes, seconds and milliseconds) is set more than 65,535 system clock count inputs (ticks).

1. *Delaying by defining a time delay by number of clock ticks.* Function *void* OSTimeDly (unsigned short *delayCount*) delays task by (delayCount – 1) ticks of system clock. It returns no parameter.

Task parameters passing *PF*: A 16-bit integer, *delayCount*, to delay a task at least till the system clock count inputs (ticks) equals to (*delayCount* – 1) + *count*, where *count* is the present number of ticks at system-clock.

An exemplary use as a function in a task is OSTimeDly (10,000). It delays that task for at least 100,000 ms if system clock ticks after every 10 ms.

Example 9.12

```
1. to 18. /* Steps as per Example 8.7 codes for Task1_Connect Function. */
.
.
19. /* Time delay for 1 second = period of 100 system ticks if system tick is set at every 10 ms.*/
OSTimeDly (100);
20. /* Resume Task1_Connect by a function defined in next subsection and execute other codes within
the loop. */
.
.
21. }; /* End of while loop*/
22. } /* End of the Task1_Connect function. */
```

2. *Resuming a delayed task by OSTimeDly.* Function unsigned byte OSTimeDlyResume (unsigned byte_taskPriority) resumes a previously delayed task, whether the delay parameter was in terms of the *delayCount* ticks or hours, minutes and seconds. *Note:* In case, defined delay is more than 65,535 system clock ticks, OSTimeDlyResume will not resume that delayed task.

Returning *RG*: Returns the following error parameters.

(a) OS_NO_ERR = true, when resumption after delay succeeds.
(b) OS_TASK_NOT_EXIST = true, if task was not created earlier.
(c) OS_TIME_NOT_DLY = true, if task was not delayed.
(d) OS_PRIO_INVALID, when taskPriority parameter that was passed is more than the OS_PRIO_LOWEST (=23).

Task parameters passing *PG*: taskPriority is the priority of that task that is delayed before resumption.

An exemplary use is OSTimeDlyResume (*Task_ReadPortPriority*). It resumes a delayed task that the OS identifies by priority *Task_ReadPortPriority*.

Example 9.13

```
1. to 19. /* Steps as per Example 9.12 codes for Task1_Connect Function. */
.
.
20. /* Time delay for 1 second = period of 100 system ticks if system tick is set at every 10 ms.*/
OSTimeDly (100);
21. /* Other codes */
.
.
22. /* Resume Task1_Connect Control and execute other codes within the loop. */
OSTimeDlyResume (Task1_Connect_Priority);
.
.
23. }; /* End of while loop*/
24. } * End of the Task1_Connect function. */
```

3. *Delaying by defining a time delay in units of hours, minutes, seconds and milliseconds.* Function void OSTimeDlyHMSM (unsigned short hr, unsigned short mn, unsigned short sec, unsigned short *mils*) delays up to 65,535 ticks a task with delay time defined by hr hours between 0 and 55, mn minutes between 0 and 59, *sec* seconds between 0 and 59 and *mils* milliseconds between 0 and 999. The *ms* adjusts to the integral multiple of number of system-clock ticks. The task in which this function is defined is delayed.

Returning *RH*: The function OSTimeDlyHMSM ( ) returns an error code as following.

(a) OS_NO_ERR, when four arguments are valid and resumption after delay succeeds.
(b) OS_TIME_INVALID_HOURS, OS_TIME_INVALID_MINUTES, OS_TIME_INVALID_SECONDS and OS_TIME_INVALID_MILLI, if the arguments are greater than 55, 59, 59 and 999, respectively.
(c) OS_TIME_ZERO_DLY, if all the arguments passed are 0.

Task parameters passed *PH*: (a) to (c) hr, mn, sec and ms are the delay times in hours, minutes, seconds and milliseconds by which task delays before resuming.

An exemplary use is using OSTimeDlyHMSM (0, 0, 0, 999) function in the codes of a task in step 8 in Example 9.12. It delayed that task by 9990 ms. The function delays that task for at least 10 ms if system clock ticks after every 10 ms. (If delay is defined as 9,000,000 ms, the OSTimeDlyResume shall not be able to resume this task when asked. Number of ticks must be less than 65,535, which means maximum delay can be 655,350 ms if system clock ticks every 10 ms].

9.2.4 Memory Allocation-Related Functions

Memory functions are required to allocate fixed-size memory blocks from a memory partition having integer number of blocks. The allocation takes place without fragmentation. The allocation and de-allocation take place in fixed and deterministic time (Example 8.6). MUCOS memory functions for the tasks are as per Table 9.4.

Table 9.4 Real-Time Operating System (RTOS) Memory Functions for Querying, Creating, Getting and Putting

Prototype Functions	*What are the Parameters Returning and Passed?*	*When is this OS Call?*
OSMem *OSMemCreate (void *memAddr, MEMTYPE[1] numBlocks, MEMTYPE blockSize, unsigned byte *memErr)	*RI* and *PI*	To create and initialize a memory partition. The memory blocks are then allotted from the partition
void *OSMemGet (OS_MEM *memCBPointer, unsigned byte *memErr)	*RJ* and *PJ*	To find pointer of the memory control block allocated to the memory blocks, NULL if no blocks, OSMemGet is used when an interrupt service routine (ISR) or task needs to get the memory block(s)
unsigned byte OSMemQuery (OS_MEM *memCBPointer, OS_MEM_DATA *memData)	*RK* and *PK*	To find pointers of the memory control block and OS_MemData data structure
unsigned byte OSMemPut (OS_MEM *memCBPointer, void *memBlock)	*RL* and *PL*	To return a pointer of memory block in the memory partitions from the memory control block pointer. OSMemPut is used when the application no longer needs the memory block

[1] MEMTYPE is unsigned int of 16 or 32 bits.

1. Creating memory blocks at a memory address. Function OSMem *OSMemCreate (void *memAddr, MEMTYPE numBlocks, MEMTYPE blockSize, unsigned byte *memErr) is an OS function, which partitions the memory from an address with partitions in the blocks. The creation and initializing of the memory partitions into the blocks helps the OS in resources allocations.

Returning *RI*: The function *OSMemCreate ( ) returns a pointer to a control block for the created memory partitions. If none created, the create function returns NULL pointer.

Task parameters passing *PI*: MEMTYPE is the data type according to the memory, whether 16-bit or 32-bit CPU memory addresses are there. For example, 16-bit in 68HC11 and 8051. (i) *memAddr is pointer for the memory-starting address of the blocks. (ii) numBlocks is the number of blocks into which the memory must be partitioned (must be 2 or more). (iii) The blockSize is the memory size in bytes in each block. (iv) *memErr is a pointer of the address to hold the error codes. At the address *memErr the following global error code variables change from false to true. OS_NO_ERR = true when creation succeeds. OS_MEM_INVALID_BLKS = true, when at least two blocks are not passed as arguments. (v) OS_MEM_INVALID_PART= true, when memory for partition is not available. (vi) OS_MEM_INVALID_SIZE= true, when block size is smaller than a pointer variable.

Example 9.14 shows the creation of four blocks of memory each block being of 1 kB. Let memory start address be 0x8000.

Example 9.14

```
1. /* Definition in Pre-Processor for a 16-bit unsigned number, MEMTYPE to define number of blocks
which can be between 0 and 65535 and to define the number of bytes that store at a block. Maximum
number of bytes at a block can be 65535. */

typedef unsigned short MEMTYPE;
3. /* Codes for main function or for a task function */
.
.
4. /* Codes for creating the blocks of memory*/
memAddr = 0x8000;
numBlocks =4;
blockSize = 1024; /* Each block is of 1kB memory */
*OSMemCreate (*memAddr, numBlocks, blockSize, *memErr);
.
.
5. /* Other Codes for the function. */
.
.
} /* End of the function */
```

2. *Getting a memory block at a memory address.* Function void *OSMemGet (OS_MEM *memCBPointer, unsigned byte *memErr) is to retrieve a memory block from the partitions created earlier.

Returning *RJ*: The function OSMemGet ( ) returns a pointer to the memory control block for the partitions. It returns NULL if no blocks exist there.

Task parameters passing *PJ*: (i) Passes a pointer as argument for the control block of a memory partition. (ii) The function OSMemGet ( ) passes the error code pointer *memErr so that later it returns one of the

following. OS_NO_ERR, when memory block returns to the memory partition, or OS_MEM_FULL, when memory block cannot be put into the memory partition as it is full.

Example 9.15 shows how to get a pointer to the memory block, which has been created earlier.

Example 9.15

```
1. to 5. /* Codes as per Example 9.14 */
6. /* Codes for retrieving the pointer to memory block in a partition created by step 5 in Example 9.14 */
memPointer = 0xA000;
memErr = OS_MEM_NO_FREE_BLKS;
*OSMemGet (*memPointer, *memErr);
.
.
5. /* Other Codes for the function. */
.
.
} /* End of the function */
```

3. *Querying a memory block.* Function unsigned byte OSMemQuery (OS_MEM *memCBPointer, OS_MEMDATA *memData) is to query and return the error code and pointers for the memory partitions. OS_NO_ERROR as 1 if a memory address *memPointer exists at *OS_MEMDATA, else returns 0.

Returning *RK*: The function OSMemQuery ( ) returns an error code, which is an unsigned byte. The code is OS_NO_ERR = 1 when querying succeeds, else 0.

Task parameters passing *PK*: (i) The function OSMemQuery ( ) passes (i) a pointer memPointer of the memory created earlier, and (ii) a pointer of data structure, OS_MEM_DATA. As pointers are passed as references, the information about memory partition returns with the memory control block pointer.

4. *Putting a memory block into a partition.* Function unsigned byte OSMemPut (OS_MEM *memCBPointer, void *memBlock) returns a memory block pointed by *memBlock, which memory control block points by *memCBPointer.

Returning *RL*: The function OSMemPut ( ) returns error codes for one of the following: either (i) OS_NO_ERR, when the memory block returned to the memory partition or (ii) OS_MEM_FULL, when the memory block cannot be put into the memory partition as it is full.

Task parameters passing *PL*: (i) The function OSMemPut ( ) passes a pointer *memCBPointer of the memory control block for the memory partitions. It is there that the block is to be put. (ii) A pointer of the memory block *memBlock is to be put into the partition.

9.2.5 Semaphore-Related Functions

MUCOS semaphore functions for the tasks are as per Table 9.5. MUCOS also provides for event functions for an event flags group to handle task-pending action on occurrence of any or all events. These are not discussed in this section.

When a semaphore created by this OS is used as a resource-acquiring key (as mutex), the semaphore value to start with is 1, which means that resource is available and 0 will mean not available (Section 7.7.2). When a semaphore created by this OS is used as an event-signalling flag or as counting semaphore, the semaphore value to start with = 0 or N when using the semaphore as event-signalling flag or counting, respectively (Sections 7.7.1 and 7.7.4).

1. *Creating a semaphore for the IPCs.* Function OS_Event OSSemCreate (unsigned short *semVal*) is for creating an OS's ECB (Event Control Block) *for an IPC* with semVal returning a pointer, which points to the ECB. A semaphore creates and initializes with the value = *semVal.*

Returning *RM*: The function OSSemCreate ( ) returns a pointer *eventPointer for the ECB allocated to the semaphore. Null if none available.

Task parameters passing *PM*: A *semVal* between 0 and 65535 is passed. For IPC as an event-signalling flag, *SemFlag* must pass 0 and as a resource-acquiring key, *SemKey* must pass 1. For IPC as a counting semaphore, *SemCount* must be either 0 or a count-value to be passed in the beginning.

Refer to Examples 9.16, 9.17 and 9.18 for understanding the use of OSSemCreate.

2. *Waiting for an IPC for semaphore release.* Function void OSSemPend (OS_Event *eventPointer, unsigned short *timeOut,* unsigned byte **SemErrPointer*) is *for letting a task wait till the release event of a semaphore: SemFlag* or *SemKey* or *SemCount.* The latter is at the ECB pointed by *eventPointer. SemFlag* or *SemKey* or *SemCount* becoming greater than 0 is an event that signals the release of the tasks in the waiting states. The tasks now become ready for running (They run if no other higher-priority task is ready). The tasks also become ready after a predefined timeout, *timeOut. SemFlag* or *SemKey* or *SemCount* decrements and if it becomes 0 then it makes the semaphore pending again and the other tasks using OSSemPend ( ) (have to wait for its release.

Returning *RN*: The function OSSemPend ( ) when a semaphore is pending, then suspends till >0 (release) and decrements the semVal on unblocking that task SemVal = 1. The following macros return true. (i) OS_NO_ERR returns true, when semaphore search succeeds (SemVal > 0). (ii) OS_TIMEOUT returns true, if semaphore did not release (did not become >0) during the ticks defined for the timeout. (iii) OS_ERR_PEND_ISR returns true, if this function call was by an ISR and which is an error, since an ISR should not be blocked for taking the semaphore. (iv) OS_ERR_EVENT_TYPE returns true, when *eventPointer is not pointing to the semaphore.

Task parameters passing *PN*: (i) The OS_Event *eventPointer passes as a pointer to ECB that associates with the semaphore: *SemFlag* or *SemKey* or *SemCount.* (ii) Passes argument for the number of timer ticks for the *timeOut.* Task unblocks after the delay is equal to (*timeOut* – 1) ticks even when the semaphore is not released. It prevents infinite wait. It must pass 0 if this provision is not used. (iii) Passes *err, a pointer for holding the error code.

Examples 9.16, 9.17 and 9.18 explains use of OSSemPend ( ).

Table 9.5 Real-Time Operating System (RTOS) Semaphore Functions for Inter Tasks Communications

Prototype of Functions[1]	*What are the Parameters Returning and Passed?*[1]	*When is this OS Function Called?*
OS_Event OSSemCreate (unsigned short *semVal*)	***RM*** **and** ***PM***	To create and initialize ECB and a semaphore to *SemVal.*
void OSSemPend (OS_Event *eventPointer, unsigned short timeOut, unsigned byte *SemErrPointer)	***RN*** **and** ***PN***	Only a task and not an interrupt service routine (ISR) can accept the semaphore. The function is to check whether semaphore is pending or not pending (0 or >0). If pending (=0), then suspend the task till >0 (released). If >0, decrement the value of semaphore and run the waiting codes. Decrement makes the semaphore pending again for some other

(*Contd*)

Prototype of Functions[1]	*What are the Parameters Returning and Passed?* [1]	*When is this OS Function Called?*
		task. Pending period ends on timeOut also after the specific number of timer ticks (system clock interrupts) = timeOut – 1. Block the task on pending and unblock on releasing the semaphore on OSSemPost.
unsigned short OSSemAccept (OS_EVENT *eventPointer)	***RO*** and ***PO***	An ISR or task can accept the semaphore. The function checks whether semaphore value > 0 and if yes, then retrieve and decrement. Used when there is no need to wait by the task, only decrease it to 0 if the value is not already zero.
unsigned byte OSSemPost (OS_EVENT*eventPointer)	***RP*** and ***PP***	An ISR or task can post the semaphore. SemVal if 0 or more, increments. Increment makes the semaphore again not pending for the waiting tasks. If tasks are in the blocked state and waiting for the SemVal semaphore to acquire value >0 then make those also ready to run as and when scheduled by the kernel. The kernel finds the priority of the running and ready tasks and runs the one that has the highest priority first.
unsigned byte OSSemQuery (OS_EVENT *eventPointer, OS_SEM_DATA *SemData)	***RQ*** and ***PQ***	To get semaphore information.

[1]Column 2 refers to the corresponding explanatory paragraph in the text.

3. *Check for availability of an IPC after a semaphore release.* Function unsigned short OSSemAccept (OS_Event *eventPointer) checks for a semaphore value at ECB and whether it is greater than 0. An unassigned 16-bit value is retrieved and then decremented.

Returning *RO*: The function OSSemAccept ( ) decrements the semVal if >0 and returns the predecremented value as an unsigned 16-bit number. It returns 0 if semVal was 0 and semaphore was not pending when posted (released). After this, the task codes run further.

Task parameters passing *PO*: The OS_Event *eventPointer passes a pointer for the ECB that associates with semaphore, *semVal*.

4. *Sending an IPC after a semaphore release.* Function unsigned byte OSSemPost (OS_Event *eventPointer) is *for letting another waiting task not wait now afterwards and an IPC is sent for the release event of the semaphore, SemFlag* or *SemKey* or *SemCount* (Example 9.16). The IPC is at the ECB pointed by *eventPointer *SemFlag* or *SemKey* or *SemCount*, it increments and if it becomes greater than 0, it is an event that signals the release of a task waiting state. The task now become ready for running (runs if no other higher-priority task is ready). *SemFlag* or *SemKey* or *SemCount* decrements on running that task and if it becomes < 0 then it makes semaphore pending again and the other tasks have to wait for its release.

If the IPC is posted from an ISR, then the pending task can run only after OSIntrExit ( ) executes and return from the ISR. If the presently running task is of higher priority than the task pending for the want of the IPC, then the present task will continue to run unless blocked or delayed by executing some function.

Returning *RP*: The function OSSemPost ( ) increments the semVal if it is 0 or > 0, and later following macros return *true* from the error code macros as follows: (i) OS_NO_ERR returns *true*, if semaphore signaling succeeded (SemVal > 0 or 0). (ii) OS_ERR_EVENT_TYPE returns *true*, if *eventPointer is not pointing to the semaphore. (iii) OS_SEM_OVF returns *true*, when semVal overflows (cannot increment and is already 65,535).

Task parameters Passing *PP*: The OS_Event *eventPointer passes as pointer to ECB that associates with the semaphore.

5. *Retrieve the error information for a semaphore.* Function unsigned byte OSSemQuery (OS_EVENT *eventPointer, OS_SEM_DATA *SemData) puts the data values for the semaphore at the pointer, SemData.

Returning *RQ*: After the OSSemQuery ( ) runs the SemData function and gets the OSCnt, which is the semaphore present value (count). The Semdata also gets the list of the tasks, which are waiting for the semaphore. The list is at pointers OSEventTbl [ ] and OSEventGrp. The semaphore error information parameters we can find on running the macros, OS_NO_ERR and OS_ERR_EVENT_TYPE. (i) OS_NO_ERR returns true, when querying succeeds or (ii) OS_ERR_EVENT_TYPE returns true, if *eventPointer is not pointing to the semaphore.

Task parameters passing *PQ*: The function OSSemQuery ( ) passes a pointer of the semaphore created earlier at *eventPointer and a pointer of the data structure at *SemData for that created semaphore.

Example 9.16

The use of OSSemPost and OSSemPend as an event-signalling flag is as follows. Let the initial value of an event-signalling *SemFlag* be 0 on creating a semaphore by OSSemCreate. A task must first execute OSSemPost, which increases the *SemFlag* to 1 and thus notifies the event. When *SemFlag* becomes 1 released (not taken), the waiting task (task that executed OSSemPend function) on posting of the semaphore as 1 can start running (it runs when no other higher-priority task is ready to run). The semaphore *SemFlag* decreases to 0 (again pending or taken) on return from the OSSemPend function. The waiting codes of the task now run.

Consider an example of reading bytes on a network. Assume that an ISR executes on a character reaching at a port. Another task is read port A. Third task is to decipher the port message. This example shows how the steps *a*, *b* and *c* synchronize the ISR and two tasks using semaphore as event-signalling flag for waiting and sending an IPC.

1. For step *a*, let the task be ISR_CharIntr. It executes on interrupt and writes the character into PortA buffer. It signals for availability character at port A buffer using semaphore semFlag.
2. For step *b*, let the task signalled to run by the ISR be Task_Read_Port_A. It is for reading the character when available at port A.
3. For step *c*, let the task be Task_Decrypt_Port_A. It is for decrypting the message.

The codes to create the ISR and two tasks and synchronize these will be as follows.

```
1. /* Codes as per Example 9.7 Step 1 except last comment line*/
.
.
2. /* Preprocessor definitions for maximum number of inter process events to let the MUCOS allocate
memory for the Event Control Blocks */
#define OS_MAX_EVENTS 8/* Let maximum IPC events be 8 */
#define OS_SEM_EN 1/* Enables inclusion of semaphore functions in applications using MUCOS */
/* End of preprocessor commands */
3. /* Codes as per Example 9.7 Step 2 */
.
.
```

4. /* Prototype definitions for ISR and two tasks, stacks and priorities. */
static void ISR_CharIntr (void *IntrVectorPointer);
static void Task_Read_Port_A (void *taskPointer);
static void Task_Decrypt_Port_A (void *taskPointer);
static OS_STK Task_Read_Port_AStack [Task_Read_Port_AStackSize];
static OS_STK Task_Decrypt_Port_AStack [Task_Decrypt_Port_AStackSize];
#define Task_Read_Port_AStackSize 100 /* Define task 2 stack */
#define Task_Decrypt_Port_AStackSize 100 /* Define task 3 stack */
#define Task_Read_Port_APriority 12 /* Define task 2 priority */
#define Task_Decrypt_Port_APriority 13 /* Define task 3 priority */
5. /* Prototype definitions for the semaphores */
OS_EVENT: **SemFlag1;** /*Needed when using Semaphore as flag for inter-process communication between port check and port read tasks. Port read has to wait for check O.K.*/
OS_EVENT: **SemFlag2;** /*Needed when using Semaphore as flag for inter-process communication between port read and port read decipher task. Port decrypting has to wait for port read */
OS_EVENT: **SemKey1;** /*Needed when using Semaphore as resource key as in Example 9.17*/
OS_EVENT: **SemCount;** /*Needed when using Semaphore as Counting as in Example 9.18*/
6. /* Codes as per Example 9.7 Step 3 to 5 */
.
.
7. /* Create Semaphores and Start MUCOS RTOS to let us RTOS control and run the created tasks */
SemFlag1 = OSSemCreate (0); /*Declare initial value of semaphore = 0 for using it as an event signaling flag*/
SemFlag2 = OSSemCreate (0); /*Declare initial value of semaphore = 0 for using it as an event signaling flag*/
OSStart ( ); /* Infinite while-loop is there in each task. So there is no return from the RTOS function OSStart ( ). */
} /* End of while loop*/
}/ *** End of the Main function ***/
/* Codes as per Example 9.7 Step 7 and 8 */
8. /* Create two tasks as per Step 2 by defining two task identities, Task_Read_Port_A and Task_Decrypt_Port_A and the stack sizes and other TCB parameters. */
OSTaskCreate (Task_Read_Port_A, *void (*) 0, (void *) &* Task_Read_Port_A*Stack [*Task_Read_Port_A*StackSize],* Task_Read_Port_APriority*);*
OSTaskCreate (Task_Decrypt_Port_A, *void (*) 0, (void *) &* Task_Decrypt_Port_A*Stack [*Task_Decrypt_Port_A*StackSize],* Task_Decrypt_Port_APriority*);*
9. while (1) { /* Infinite loop of FirstTask */
.
.
10. /* Suspend, with no resumption later, the First task as it must run once only for initiation of timer ticks and for creating the tasks that the scheduler controls by preemption. */
11. *OSTaskSuspend (FirstTask_Priority);* /*Suspend First Task and control of the RTOS passes to other tasks waiting execution*/
12. } /* End of while loop */

```
13. } /* End of FirstTask Codes */
/*************************************************************************/
14. /* The codes for the ISR_CharIntr */
static void ISR_CharIntr (void *BufferPointer) {
15. OSIntEnter ( ); /*
/* Initial assignments of the variables and pre-infinite loop statements that execute once
only */
.
.
16. /* Code for resetting interrupt pending flag, in case, it does not automatically reset in the given interrupt
service start */
17. /* Codes for ISR_CharIntr to put the received character into Port A buffer at *BufferPointer */
.
.
18. /* Code for readying for next interrupt at the port */
19. /* Release semaphore to a task waiting for the read at Port A */
OSSemPost (SemFlag1);
20. OSIntExit ( );
21. } /* End of the ISR_CharIntr function */
22. /*******The codes for the Task_Read_Port_A *******/
static void Task_Read_Port_A (Void *BufferPointer) {
/* Initial assignments of the variables and pre-infinite loop statements that execute
once only*/
.
.
.
23. /* Start an infinite while-loop. */
while (1) {
24. /* Wait for SemFlag1 =1 by OSSemPost function of character availability in buffer after interrupt at
port */
OSSemPend (SemFlag1, 0, SemErrPointer);
25. /* Codes for reading from Port A and storing message at a new buffer*/
.
.
26. /* Release semaphore to a task waiting for the decrypting*/
OSSemPost (SemFlag2);
OSTimeDly (100); /* Block the task for 100 clock ticks to enable the lower priority decryption task
start */
27. }; /* End of while loop*/
28. } /* End of the Task_Read_Port_A function */
/*************************************************************************/
29. /* Start of Task_Decrypt_Port_A codes */
static void Task_Decrypt_Port_A (void *taskPointer) {
30. /* Initial assignments of the variables and pre-infinite loop statements that execute once only*/
```

```
.
.
.
31. /* Start an infinite while-loop. */
while (1) {
OSSemPend (SemFlag2, 0, *SemErrPointer); / *Wait for unlimited time for SemFlag2 =1 by OSSemPost
function for a character read at Port A */
32'. /* Codes for Task_Decrypt_Port_A that read the new buffer and put the decrypted data back into new
buffer */
.
.
OSTimeDlyesume (Task_Read_Port_Priority); /* Resume the Task_Read_Port delayed earlier */
33. }; /* End of while loop*/
34. }/ * End of the Task_Decrypt_Port_A function */
/*********************************************************************/
```

Example 9.17

Use of OSSemPost and OSSemPend as resource-acquiring keys is as follows. The resource may be a shared memory buffer or commands that use global variables or touch screen or flash memory or print device control registers and buffers. Let a resource key available *SemKey's* initial value be 1. A task must first execute OSSemPend, which decreases the *SemKey* value to 0 and the codes of the critical section *C* of task run. The section is one in which that resource is used. The same task must execute OSSemPost after its codes finish at *C* and thus signal the resource key availability to other tasks. The *SemKey* becomes 1 and released (not taken) on return from OSSemPost function. If no other higher-priority task is ready to run, another task that shares the resource with the earlier task and executes OSSemPend function on entering shared data section *C'*. The *C'* executes OSSemPost function on exit. Making SemKey 0 and then 1 in task sections *C* and *C'* lets one task only acquire the resource in a specific running state of a task.

Recall Example 9.16, steps *b* and *c* for reading and then decrypting the bytes from a network. If there is no message, how can it be deciphered? Let us revisit Example 9.16. The present example will show how the steps *b* and *c* synchronize using a semaphore key for key waiting and key sending IPC, and how steps *a* and *b* synchronize using a semaphore as event signaling flag.

```
1. /* Codes as per Example 9.16 Step 1 to 5 */
.
.
2. /* Prototype definition for the semaphore used as resource key for inter-process communication between
port read and port message encrypt read tasks. */
OS_EVENT *SemKey1; / *Needed when using Semaphore as resource key*/
/* Codes as per Example 9.16 Steps 6 to 21. However, create the semaphore SemKey1before calling
OSStart ( ) in main*/
SemKey1= OSSemCreate (1); /*Declare initial value of semaphore = 1 for using it as a resource acquiring
key*/
/*********************************************************************/
```

```
4. /* After the end of codes for ISR_CharIntr, the codes for the Task_Read_Port_A redefined to show a
use of the key*/
static void Task_Read_Port_A (void *taskPointer) {
5. /* Initial assignments of the variables and pre-infinite loop statements that execute once only*/
.
.
.
6. while (1) { /* Start an infinite while-loop. */
7. OSSemPend (SemFlag1, 0, SemErrPointer); / *Wait for SemFlag1 =1 by OSSemPost function of
character availability check task */
/*Acquire resource as SemKey presently > 0 and decrement it and not allow any other task to use this
key*/
8. OSSemPend (SemKey1, 0, SemErrPointer);
9. /* Codes for reading from Port A buffer and storing at new buffer, which will be shared with
Task Decrypt_Port_A and the tasks for transmitting decrypted data to another device */
.
.
10. /* Release the key to a task waiting for the decrypting*/
OSSemPost (SemKey1);
11. /* To exit the infinite loop at a task that has been assigned a higher priority and to let the
lower priority task run call OS delay function for wait of 100 ms (ten OS timer tick. This is the
method to let the other-task of lower priority execute Port A decrypt. */
OSTimeDly (10);
};/* End of while loop*/
12. /* * End of the Task_Read_Port_A function */
/**************************************************************************/
13. /* Start of Task_Decrypt_Port_A codes */
static void Task_Decrypt_Port_A (void *taskPointer) {
/* Initial assignments of the variables and pre-infinite loop statements that execute once only*/
.
.
14. while (1) { /* Start the infinite loop */
15. /* Acquiring the resource as SemKey1 > 0 and decrement it to not to let port read task use this key*/
OSSemPend (SemKey1, 0, SemErrPointer);
16. /* Codes for Task_Decrypt_Port_A or deciphering the message read at new buffer */
.
.
17. /* Release the key to a task waiting for the new buffer data transmission to a device */
OSSemPost (SemKey1);
OSTimeDlyResume (Task_Read_Port_APriority); /* Resume the delayed task */
18. } /* End of while loop*/
19. } / * End of the Task_Decrypt_Port_A function */
/**************************************************************************/
```

Example 9.18

Use of counting semaphore helps in programming for the printer or other bounded buffer problem (producer consumer problem) (Refer to Section 7.5). The use of OSSemPost to increase the count and OSSemPend to decrease the count in a counting semaphore is as follows. Recall Example 9.16. Let us first modify this example as follows.

1. For step *a*, on the interrupt let ISR_CharIntr executes. It posts a semaphore to task, Task_Read_Port_A.
2. For step *b*, let the task be Task_Read_Port_A. It is for reading the characters when available at port buffer. Let Task_ReadPortA read a stream of the characters from the buffer. Let the task put the characters, as it reads one by one, into a bounded buffer (it is a producing task; buffer is bounded by a limit like a printer or display buffer).
3. For step *c*, let the task be Task_Decrypt. It is for decrypting and displaying the maximum 160 characters (it is a consuming task; it is like printing from the print buffer).
4. For step *d*, let the task be Task_Display. Task display the decrypted data at display device.
5. Let the display buffer of 160 characters be shared by Task_Decrypt and is bounded upto 160 addresses in memory.

This example shows how the steps *b* and *c* synchronize using counting semaphore and how the steps *c* to *d* synchronize.

1. Let there be a counter *SemCount*, which counts the number of times a task posting the semaphore ran. Let *SemCount's* initial value be 0. A first task section must first execute OSSemPost, which increases the *SemCount* to 1. Every time this task section runs, *SemCount* increases by 1. Every time the task deciphers a character in another task, it decreases by 1.
2. When *SemCount* reaches a specific preset value, then a semaphore event-signalling flag *SemCountLimitFlag* sets, and the count resets to 0. As there is an OS call by a delay function, the lower-priority task for deciphering starts running and it acquires the key. The semaphore resource key *SemKey* becomes unavailable to the reading task and further reading stops till the deciphering task releases the key and also executes OSSemPend to decrease *SemCount to let the task that reached the limit run* again.

```
1. /* Codes as per Example 9.17 Steps 1 and 2*/
.
.
2. /* Prototype definitions for one ISR and three tasks*/
static void ISR_CharIntr (void *IntrVectorPointer);
static void Task_Read_Port_A (void *taskPointer);
static void Task_Decrypt (void *taskPointer);
static void Task_Display (void *taskPointer);
3. /* Definitions for three task stacks */
static OS_STK Task_Read_Port_AStack [Task_Read_Port_AStackSize];
static OS_STK Task_DecryptStack [Task_DecryptAStackSize];
static OS_STK Task_ DisplayStack [Task_DisplayStackSize];
4. /* Definitions for three task stack size */
#define Task_Read_Port_AStackSize 100 /* Define task 2 stack size*/
#define Task_DecryptStackSize 100 /* Define task 3 stack size*/
#define Task_DisplayStackSize 100 /* Define task 4 stack size*/
```

5. /* Definitions for four task priorities. */
#define Task_ ReadPortAPriority 11 /* Define task 2 priority */
#define Task_DecryptPriority 12 /* Define task 3 priority */
#define Task_DisplayPriority 14 /* Define task 5 priority */
6. /* Prototype definitions for the semaphores */
OS_EVENT ***SemFlag1;*** /* Needed when using semaphore as the flag for inter-process communication between ISR and port read tasks. Port A read task has to wait for ISR notifying the character receipt at port A */
OS_EVENT ***SemCountSend;*** /* Needed when using semaphore for sending task semaphore count value in the inter-process communication between read and decipher tasks. Port deciphering has to wait for port receive from sending task */
OS_EVENT ***SemKey;*** /* Needed when using semaphore as resource key */
OS_EVENT ***SemCountRecv;*** /* Needed when using semaphore for counting the deciphered for display */
8. /* Codes as per Example 9.7 Step 3 to 8. However, the semaphores are to be created and initialised as done earlier in Step 5 Example 9.16 */
SemFlag1 = OSSemCreate (0); /* Declare initial value of semaphore = 0 for using it as an Levent signaling flag*/
SemCountSend = OSSemCreate (0); /* Declare initial value of semaphore = 0 for an event counter for sender */
SemCountDisp = OSSemCreate (0); /* Declare initial value of semaphore = 0 for using it as a flag for display */
/* Declare initial value of semaphore count = 0 for using as a counter that gives the number of times a task, which sends into a buffer that stores a character stream, ran minus the number of times the task which used the character from the stream ran from the buffer */
SemCountRecv = OSSemCreate (160); /* Declare initial value of semaphore = 160 for 160 free addresses count for receiving the bytes for display */
.
9. /* Create four tasks as per Step 3, defined by four task identities, Task_Read_Port_A, Task_Decrypt_Port_A, Task_EncryptPortB and Task_SendPortB and the stack sizes and other TCB parameters. */
OSTaskCreate (Task_Read_Port_A, *void (*) 0, (void *) &* Task_Read_Port_A*Stack [*Task_Read_Port_A*StackSize],* Task_ReadPortAPriority*);*
OSTaskCreate (Task_Decrypt, *void (*) 0, (void *) &* Task_Decrypt*Stack [*Task_Decrypt*StackSize],* Task_DecryptPriority*);*
OSTaskCreate (Task_Display, *void (*) 0, (void *) &* Display*Stack [*Task_Display*StackSize],* Task_Display Priority*);*
10. /* Codes same as at Steps 9 to 21 in Example 9.16 */
.
.
11. /* The codes for the Task_Read_Port_A redefined to use the semephore as counter*/
static void Task_Read_Port_A (*void *taskPointer*) {
12. /* Initial assignments of the variables and pre-infinite loop statements that execute once only*/

```
unsigned short countLimit = 160; /* Declare the buffer-size for the characters countLimit = 160 */
.
.
13. while (1) { /* Start an infinite while-loop. */
14. / *Wait for SemFlag1 1 by OSSemPost function of character availability check task */
OSSemPend (SemFlag1, 0, SemErrPointer);
15. OSSemPend (SemCountRecv, 0, *SemErrPointer); /* wait for available space in receiving buffer) */
CountLimit = 160; /* CountLimit reset to 160 */
16. /*Read port A buffer byte and write byte into decipher buffer */
CountLimit – – ; if (CountLimit >0){
OSSemPost (SemCountSend);} /* Release the SemCountSend to let the decipher task start */
17. OSTimeDly (10); /* To exit the infinite loop of this assigned higher priority task to let the lower
priority task run call the OS delay function for a wait of 100 ms (ten OS timer tick). This is the method to
let the other task of lower priority execute Port A's message deciphering task */
18. };/* End of codes for the action on reaching the limit of putting characters into the buffer */
19. }; /* End of while loop*/
20. } / * End of the Task_Read_Port_A function */
/*****************************************************************/
21. /* Start of Task_Decrypt */
static void Task_Decrypt (void *taskPointer) {
22. /* Initial assignments of the variables and pre-infinite loop statements that execute only once*/
.
.
23. while (1) { /* Start the infinite loop */
24. /* Take the key to not letting the Task_Read_Port_A run before at least one cycle of this while loop*/
OSSemPend (SemCountSend, 0, *SemErrPointer);
25. /* Codes for Task_Decrypt_Port_A or deciphering for displaying the message when placed
in new buffer fort */
.
.
OSSemPost(SemCountdisp, 0, *Err);
OSTimeDly(10); /* to delay for transfer control to display task */
26. OSTimeDlyResume(Task_ReadPortAPriority); /* Resume task read port A */
27. }; /* End of while loop*/
28. }/ * End of the Task_Decryptfunction */
/*****************************************************************/
29. /* The codes for the Task_Display */
static void Task_Display (void *taskPointer) {
30. /* Initial assignments of the variables and pre-infinite loop statements that execute only once*/
.
.
.
31. while (1) { /* Start an infinite while-loop. */
32. /* Codes for displaying the deciphered characters */
OSSemPend (SemCountdisp, 0, *Err); /* wait for deciphered character */
```

```
.
.
33. /* Code for displaying character */
OSSemPost (SemCountRecv);
34. OSTimeDlyResume (Task_DecryptPriority);
35. }; /* End of while loop */
36. }/* End of the Task_Display function */
/*************************************************************/
```

9.2.6 Mailbox-Related Functions

We have seen in Examples 9.16, 9.17 and 9.18 that the semaphores communicate one of the following.

1. The occurrence of an event to other task, which is waiting for the event before running.
2. The availability of a resource in a task to let a section of codes in the task run.
3. The occurrences of an event number of times before they are taken note by other task.
4. The availability of a resource in a task to let a section of codes in the task run number of times.

However, suppose the message is a string or is in a data structure or is in a buffer or array. *The mailbox IPC can be used to communicate a pointer for that information.* Refer to Figure 7.6(a), which showed multiple types of mailboxes. In MUCOS, the *mailbox type is one message pointer per mailbox.*

Let there be a pointer *msg to the message to be sent in the mailbox, and another *mboxPointer for the message sending event and retrieving the message itself. MUCOS mailbox IPC functions for the tasks are as per Table 9.6.

1. *Creating a mailbox for an IPC.* Function *OS_Event *OSMboxCreate (void *msg)* is for creating an ECB at the RTOS and thus initializing a pointer *mboxPointer to *msg*. The *msg* pointer is NULL, if the created mailbox initialized as empty mailbox.

Task parameters passing M1: *msg is message pointer to which *mboxPointer will initialise. For an IPC, sending the message-pointer *mboxPointer communicates the *msg*.

Returning M1: The function OSMboxCreate ( ) returns a pointer to the ECB at the MUCOS and mboxPointer at ECB points to *msg*.

Step 8 in Example 9.19 shows how to use OSMboxCreate function.

2. *Check for availability of an IPC after a message at mailbox.* Function *void *OSMboxAccept (OS_EVENT* * mboxPointer)* checks for a mailbox message at ECB at mboxPointer (an event pointer). The pointer for the message *msg* returns from the function, if message is available, mboxPointer not pointing to NULL but to the *msg*. After returning, the mailbox empties, and mboxPointer will point to NULL on emptying of mailbox. The difference with OSMboxPend function is that OSMboxPend suspends the task if the message is not available and waits for mboxPointer not equal to NULL.

Task parameters passing M2: The OS_Event * mboxPointer passes as pointer to ECB that associates with the mailbox.

Returning M2: The function OSMboxAccept ( ) checks the message at *mboxPointer and returns the message pointer **msg* presently at MsgPointer. The function then returns NULL pointer if message pointer is not available at mboxPointer. Later *mboxPointer will point to NULL, because mailbox empties.

Step 40 in Example 9.19 shows how to use *OSMboxAccept* to retrieve an error string, if any, available in the mailbox without specifically waiting and blocking the task.

Table 9.6 Mailbox Real-Time Operating System (RTOS) Mailbox Functions for the Intertask Communications

Prototype of Service and System Clock Function	***What are the Parameters Returning and Passed?***[1]	*When is this OS Function Called?*
OS_Event *OSMboxCreate (void *msg)	M1 and M1	To create and initialize a mailbox message pointer for the ECB of a mailbox message.
void *OSMboxAccept (OS_EVENT *mboxPointer)	M2 and M2	To check if mailbox message *msg is pointed by *mboxPointer. Unlike OSMboxPend function, it does not block (suspend) the task if message is not available. If available, it returns the pointer * *msg* and * mboxPointer again points to NULL. On the return mailbox empties
void *OSMboxPend (OS_Event *mboxPointer, unsigned short timeOut, unsigned byte *MboxErr)	M3 and M3	To check if mailbox-message pending is available; then the message pointer is read and the mailbox emptied and * mboxPointer again points to NULL. If message is not available (*mboxPointer points to NULL) it waits, suspends the task and blocks further running (or till the number of ticks = *timeOut* – 1 occurs at the system-timer). If pending, then the task is resumed on availability, *mboxPointer now NULL. Resumes on timeOut also.
unsigned byte OSMboxPost (OS_EVENT *mboxPointer, void *msg)	M4 and M4	Sends a message for a task presently at address msg by posting the address pointer of it to the mboxPointer. Context switch to that task or any another task if of higher priority will also occur. If the box is already full, then the message is not posted and the error information is given.
unsigned byte OSMboxQuery (OS_EVENT *mboxPointer, OS_MBOX_DATA *mboxData)	M5 and M5	See text.

[1] Column 2 refers to the corresponding explanatory paragraphs in the text for the intertask communications using a mailbox.

3. *Waiting for availability of an IPC for a message at mailbox.* Function *void *OSMboxPend (OS_Event * mboxPointer, unsigned short timeOut, unsigned byte *MboxErr*) checks a mailbox message pointer *msg* at ECB event pointer, *mboxPointer*. A pointer for the message retrieves on return, if messa is available, mboxPointer not pointing to NULL but pointing to *msg* else waits till available or till tim ut, whichever is earlier. If *timeOut* argument value is 0, it means wait indefinitely till the message is available.

Task parameters passing M3: (i) The OS_Event *mboxPointer passes as a pointer to ECB that is associated with the mailbox. (ii) Passes argument timeOut. This resumes that the blocked task after the delay is equal to (timeOut – 1) count inputs (ticks) at the system-clock timer. (iii) Passes reference *MboxErr, a pointer that will hold the error codes.

Returning M3: The function OSMboxPend checks as well as waits for the message at *mboxPointer and the function returns *msg*. After returning, the mailbox empties. *mboxPointer will later point to NULL, because mailbox empties. When message is not available, it suspends the task and blocks as long as *msgPointer is not NULL. It returns NULL pointer if the message is not available (msgPointer pointing to NULL). The following macros will then return true. (i) OS_NO_ERR returns true, when mailbox message search succeeds; (ii) OS_TIMEOUT returns true, if mailbox message does not succeed during the ticks defined for the *timeout* >0; (iii) OS_ERR_PEND_ISR returns true, if this function call was from the ISR; (iv) OS_ERR_EVENT_TYPE returns true, when * mboxPointer is not pointing to the pointer type variable for *msg*.

Step 39 of Example 9.19 shows how to use *OSMboxPend* to retrieve an error string, if any, available in the mailbox by Task_Err, how to retrieve the read string at Task_OutPortB and to specifically wait and block the task.

4. *Send a message for an IPC through mailbox.* Function unsigned byte OSMboxPost (OS_EVENT * msgPointer, void *msg) sends mailbox message at ECB event pointer, * msgPointer. The message sent is at *msg, as well as at mboxPointer* after the posting.

Task parameters passing M4: The OS_Event *msgPointer passes as a pointer to ECB that associates with the message. The pointer *msg* passes to the mailbox address *msgPointer.

Returning M4: The function OSMboxPost ( ) sends the message and then returns the error code on running the macros as follows: (i) OS_NO_ERR returns true, if mailbox message signalling is succeeded, or (ii) OS_ERR_EVENT_TYPE returns true, if *MsgPointer does not point to the mailbox message type, or (iii) OS_MBOX_FULL returns true, when mailbox at msgPointer already has a message that is not accepted or returned.

Steps 24, 30 and 40 in Example 9.19 show OSMboxPost by the tasks to post the message to the waiting tasks for the messages.

5. *Finding mailbox data and retrieving the error information for a mailbox.* Function unsigned byte *OSMboxQuery* (*OS_EVENT* * msgPointer, *OS_MBOX_DATA* **mboxData*) checks for a mailbox data and places that at *mboxData*. It also finds the error information parameters, OS_NO_ERR_EVENT_TYPE for the ECB.

Task parameters passing M5: The function OSMboxQuery ( ) passes (i) a pointer of the mailbox message created earlier at *msgPointer and (ii) a pointer of data structure at mboxData.

Returning M5: Function OSMboxQuery ( ) returns information in the pointer mboxData is a data structure with current contents of the message (OSMsg) and the list of waiting tasks for the message. The error code macro OS_NO_ERR returns true, when mailbox message querying succeeds or OS_ERR_EVENT_TYPE returns true, if MsgPointer does not point to mailbox type *msg*.

Example 9.19

Let a task, *Task_Read_Port_A,* after it receives a message string (reading an array of character received at a port A), use OSMboxPost to send an IPC to another task waiting (blocked) for that message. An exemplary situation is when receiving a called party telephone number from the keypad at port A in mobile phone by a task and the task is waiting for dialing and transmitting of the number after ascertaining that the number does not have an invalid character. Let the waiting tasks be *Task_OutPortB* and *Task_SendPortB*. The latter sends the string for the telephone number to port B after the wait is over.

To *Task_OutPortB*, not only *Task_Read_Port_A* but also another task, *Task_Err*, can send an error message on detecting an invalid character or if the limit of characters expected in the string is exceeded. In the present example, the application of OSMboxPend function is for a task-wait for a message as well as for the error message string also (Refer to wait by OSMboxPend in task, *Task_SendPortB*, which executes

a service routine in case of error string detect). For example, in a mobile phone, *Task_OutPortB* can be used as follows. When there is no error message, then establish the connection with the cellular service and then dial and transmit the called number using *Task_SendPortB*. When there is an error message, *Task_OutPortB* directs the message to another task, *Task_ErrSR*. Another task displays the error message string warning the user to redial the number. The steps in this operation are as follows.

1. Step *a*: Task, *ISR_CharIntr* interrupts on the port A status ready on availability of a character. For example, the ISR executes if a key is pressed in an mobile phone keypad (Section 1.10.5). The use of semaphore *SemFlag1* as in Example 9.16 suffices because an IPC will be just for an interrupt event flag.
2. Step *b*: *Task_Read_Port_A* waits for the *SemFlag1* and executes the codes that accumulate the characters into an array to obtain a string, *str*. OSMboxPost posts a message pointer for the *str* if no other key is pressed within a time-out period.
3. Step *c*: *Task_Err* checks each *message* read at port A and sends a string, *errStr*, into the mailbox when the character is not a valid character or if the number of characters has exceeded the limit. It posts the message back into the mailbox if it does not have invalid characters. For example, character is not a number in case of a telephone number, which is read by the task at step *b*.
4. Step *d*: *Task_OutPortB* waits for *str* and *errstr* in the mailbox. The use of mailbox for the IPCs is between steps *b* and *d*, and *c* and *d*.
5. Step *e*: *Task_SendPortB*. If there is no error, the task sends the message of port B.
6. Step *f*: *Task_ErrSR* to execute a service routine in case of error.

This example shows how the steps *a* and *b* synchronize by the IPC *SemFlag1*, how tasks at the steps *b* and *c* synchronize and how steps *b* to *d* and *c* and *d* synchronize using the mailbox functions of the MUCOS.

```
1. /* Define Boolean variable and NULL pointer. Define codes as per Example 9.17 Steps 1*/
typedef char int8bit;
#define int8bit boolean
#define false 0
#define true 1
/* Define a NULL pointer */
#define NULL (void*) 0x0000
.
.
# define unsigned byte inputCharsMaxSize 16 /* Let Maximum size of telephone-number string be 16
characters. */
2. /* Preprocessor definitions for maximum number of inter-process events to let the MUCOS allocate
memory for the Event Control Blocks */
#define OS_MAX_EVENTS 12/* Let maximum IPC events be 12 */
#define OS_SEM_EN 1/* Enable inclusion of semaphore functions in application. */
#define OS_MBOX_EN 1/* Enable inclusion of mailbox functions in application. */
/* End of preprocessor commands */
3. /* Prototype definitions for ISR and five tasks for steps a to f above. */
static void ISR_CharIntr (void *IntrVectorPointer);
static void Task_Read_Port_A (void *taskPointer);
static void Task_Err (void *taskPointer);
static void Task_OutPortB (void *taskPointer);
static void Task_SendPortB (void *taskPointer);
```

```
static void Task_ErrSR (*taskPointer);
/* Definitions for five task stacks */
static OS_STK Task_Read_Port_AStack [Task_Read_Port_AStackSize];
static OS_STK Task_ErrStack [Task_ErrStackSize];
static OS_STK Task_OutPortBStack [Task_OutPortBStackSize];
static OS_STK Task_SendPortBStack [Task_SendPortBStackSize];
static OS_STK Task_ErrSRStack [Task_ErrSRStackSize];
/* Definitions for five task stack size */
#define Task_Read_Port_AStackSize 100 /* Define task 2 stack size*/
#define Task_ErrStackSize 100 /* Define task 3 stack size*/
#define Task_OutPortBStackSize 100 /* Define task 4 stack size*/
#define Task_SendPortBStackSize 100 /* Define task 5 stack size*/
#define Task_ErrSRStackSize 100 /* Define task 3 stack size*/
4. /* Definitions for five task priorities. */
#define Task_ ReadPortAPriority 10 /* Define task 2 priority */
#define Task_ErrPriority 11 /* Define task 3 priority */
#define Task_OutPortBPriority 12 /* Define task 4 priority */
#define Task_SendPortBPriority 13 /* Define task 5 priority */
#define Task_ErrSRPriority 14 /* Define task 6 priority */
5. /* Prototype definitions for semaphores */
OS_EVENT SemFlag1; / Needed when using semaphore as a flag for inter-process communication from
ISR_CharIntr and to Task_Read_Port_A (Port A read task) on port A interrupt */
OS_EVENT SemCharInvalid; / Needed when using semaphore as a flag for inter-process communication
from Task_OutPortB task and to number transmitting task Task_SendPortB. */
OS_EVENT semCharCountLimitFlag; / Needed when using semaphore as the flag for limiting the character
count value in the inter-process communication between Task_Read_Port_A and Task_Err */
6. /* Prototype definitions for the mailboxes */
OS_EVENT MboxStrPointer; / Needed when using the mailbox message between steps b and d and
d and e */
OS_EVENT MboxErrStrPointer; / Needed when using the mailbox message between steps c and d */
.
.
7. /* Codes as per Example 9.7 Step 3 to 8. However, before the 'OSStart ( );', the semaphore and mailbox
must be created and initialised as in Step 6 Example 9.18. */
SemFlag1 = OSSemCreate (0) /* Declare initial value of semaphore = 0 for using it as an event signaling
flag*/
SemFlag2 = OSSemCreate (0) /* Declare initial value of semaphore = 0 for using it as an event signaling
flag*/
semCharCountLimitFlag = OSSemCreate (0) /* Declare initial value of semaphore = 0 as an event signaling
flag*/
SemKey = OSSemCreate (1) /* Declare initial value of semaphore = 1 for using it as a resource key*/
/* Create Mailboxes for the tasks. */
8. MboxStrPointer = OSMboxCreate (NULL); /* Needed when using mailbox message between steps
b and d to pass a string message pointer*/
```

MboxErrStrPointer = OSMboxCreate (NULL); /* Needed when using mailbox message between steps *b* and c to pass a error string message pointer*/
9.
.
.
10. /* Create five tasks as shown in Step 9 Example 9.7 defining five task identities, Task_Read_Port_A, Task_Err, Task_OutPortB and Task_SendPortB and the stack sizes, other TCB parameters. */
OSTaskCreate (Task_Read_Port_A, *void (*) 0, (void *) &*
Task_Read_Port_*AStack* [Task_Read_Port_A*StackSize*], Task_ReadPortAPriority);
OSTaskCreate (Task_Err, *void (*) 0, (void *) &* Task_Err*Stack* [Task_Err*StackSize*], Task_ErrPriority);
OSTaskCreate (Task_OutPortB, *void (*) 0, (void *) &* Task_OutPortB*Stack* [Task_OutPortB*StackSize*], Task_OutPortBPriority);
OSTaskCreate (Task_SendPortB, *void (*) 0, (void *) &* Task_SendPortB*Stack* [Task_SendPortB*StackSize*], Task_SendPortBPriority);
OSTaskCreate (Task_Err, *void (*) 0, (void *) &* Task_Err*Stack* [Task_Err*StackSize*], Task_ErrPriority);
/***/
11–13. /* Codes same as those in Steps 9 to 21 in Example 9.16. The ISR executes on each key-press interrupt at port A */
;
.
.
}/ * **End of the ISR_CharIntr function** */
/***/
14. /* Example 9.16 codes for the Task_Read_Port_A redefined. To use the mailbox*/
static void Task_Read_Port_A *(void *taskPointer)* {
/* Initial assignments of the variables and pre-infinite loop statements that execute only once*/
char *portAdata; boolean find ChrInvalid (chr [ ]);
char [ ] portAinputStr; /* Let port A input string be an array to hold the data from port A*/
unsigned byte charCount = 0; /* To count the number of characters read from the port. The counter thus gives the number of times the data sent into a buffer (portAinputStr in present case) that stores a character stream, ran minus the number of times the task, which used the character from the stream, ran from the buffer */
boolean charInvalid = false; /* Initialize charInvalid flag as false */
while (1) { /* Start an infinite while-loop and Wait for SemFlag1 posting by OSSemPost function on a character-availability */
OSSemPend (*SemFlag1*, 0, *SemErrPointer*);
15. Port_AInputstr [chrCount] = PortAData;
16. /* Code for reading a byte from buffer that the ISR_CharIntr wrote before posting SemFlag 1 */
17. /* Actions on maximum size exceeding the string buffer Size and on character found invalid */
if (charCount > *inputCharsMaxSize*) {OSSemPost (*semCharCountLimitFlag*);} /* Post the *semCharCountLimitFlag* */
findChrInvalid (Port_AInputStr) {/* Code for check */ return (chrInvalid = true);}
if (charInvalid = = true) {OSSemPost (SemcharInvalid);

/* To exit the infinite loop of this higher priority assigned task to let the lower priority Task_Err, we call the OS delay function that forces a wait of 20 ms (two OS timer ticks) or until delay resume function executes. This is the method to let the other task of lower priority execute */
OSTimeDly (2); /* Delay by 20 ms (two timer ticks) to let lower priority Task_Err run */
charCount =0; /* Reset the character counter to 0. */
18. }/* End of Codes for the action on character invalid or reaching the limit of putting characters into the buffer */
19. /*Let the charCount increase after one character has been put into the string holding the character stream*/
charCount ++; /* Increment the character count */
/* If an ASCII code for start of text is found then initialize charCount = 0. */
20. If (portAinputStr [chrCount] = = 0x02) {charCount = 0;};
21. /* Other codes in case required */;
22. For sending string into mailbox if maximum characters reading over*
23. /* codes for
If (chrCount = = input CharsMaxSize) {OSMboxPost (MboxStrPointer, port A InputStr); charCount = 0;
}
24. /* When the character is equal to End of Text, ASCII code at Port A (for example, the Enter key pressed) is found send the message pointer String to the waiting mail box at Task_OutPortB and make initial charCount = 0 again for next string*/
if (PortAinputStr [charCount] = = 03)
{OSMboxPost (MboxStrPointer, portAinputStr); charCount = 0**};**
25. }; /* End of while loop*/ OSTimeDly (2); /* Delay to let Task_Err start */
26. } /* End of the Task_ReadPortA function */
/***/
27. /* **Codes of Task_Err** */
static void Task_Err (void **taskPointer*) { / Initial assignments of the variables and pre-infinite loop statements that execute once only*/
unsigned byte [ ] *msgBuffer = 0* /* Declare initial value of msgBuffer = 0 */
/*Declaration for an error string for using when at the Step *d an* error invalid-character is found at mailbox.
char [ ] *ErrStr1* = "Invalid Character Found";
/* Declaration for an error string message for task at Step *d* when the limit exceeds. */
char [ ] *ErrStr2* = "Characters in the message exceeded the limit declared at task for read";
boolean invalid = false; /* Declare invalid variable 'false' and will be assigned 'true' in case a character is found invalid. * /
28. *while (1)* { /* Start the infinite loop */
29. /* Post Limit of Message Exceeded Message to task at step *d*. */
OSSemQuery (*semCharCountLimitFlag*, SemData)
30. if (SemData -> OSCnt = = 1) {**OSMboxPost (MboxErrStrPointer, *ErrStr2*);** OSSemAccept (semCharCountLimitFlag, 0, *SemErrPointer); OSTimeDly (2);};
31. /* Codes for reading Msg */
***msgBuffer* = OSMboxPend (MboxStrPointer, 0, MboxErrData);**
.
32. /* Post Mailbox message to task at step *d* if an invalid character is not detected else Post invalid character error message */

33. OSSemQuery (SemCharInvalid, SemData);
34. /* Take *semCharInvalid* (if >0) by accepting the *semCharInvalid* semaphore. Task does not suspend even if semaphore not available (not > 0). This task has to run whether count is invaled not.
if (SemData -> OSCnt == 0 && chrInvalid = = true) {**OSMboxPost (MboxErrStrPointer, *ErrStr1*);**
OSSemAccept (semCharInvalid);} else {**OSMboxPost (MboxStrPointer, msgBuffer);** OSTimeDly (2)};
/* If data invalid post error string or message in a error mailbox else post message in the mailbox */
OSTimeDlyResume (Task_ReadPortAPriority; /* Resume the task Port A Road*
}; /* End of while loop*/
35. } / * **End of the Task_Err function** */
/**/
36. /* **Codes for the Task_OutPortB** */
static void Task_OutPortB *(void *taskPointer)* {
37. /* Initial assignments of the variables and pre-infinite loop statements that execute once only*/
char [ ] message; /* Declare error free message string pointer*/
char [ ] errMessage; /* Declare error message pointer. */
38. *while (1)* { /* Start an infinite while-loop. */
39. /* Wait for Mailbox Message available (not NULL) */
message = **OSMboxPend (MboxStrPointer, 0, MboxErrPointer);**
40. /* Check for Mailbox Error Message available (not NULL) */
errMessage = **OSMboxAccept (MboxErrStrPointer);**
if (errMessage ! = NULL) {**OSMboxPost (MboxErrStrPointer, errMessage);**}
else {
/*Codes for again sending the Port B string of characters to task for transmission; the message is tested to see that it has no invalid character or that it never exceeds the limits of its size..*/
OSMboxPost (MboxStrPointer, message);
};
/* To exit the infinite loop at higher priority assigned task to let the lower priority task run, call the OS delay function for wait of 20 ms (two OS timer ticks). This is the method to let the other task of lower priority execute Port B sending the characters. */
OSTimeDly (2);
41. OSTimeDlyResume (Task_ErrPriority); /* Let delayed higher priority task err resume. */
}; /* End of while loop*/
42. }/ * **End of the Task_OutPortB function** */
/**/
43. /* **Codes of Task_SendPortB** */
static void Task_SendPortB (*void *taskPointer*) {
/* Initial assignments of the variables and pre-infinite loop statements that execute only once */
unsigned char [ ] *message*;
.
.
44. *while (1)* { /* Start the infinite loop */
45.
46. /* Wait for error free message from Port B. If available, retrieve it. */
message = **OSMboxPend (MboxStrPointer, 0, MboxErrPointer);**

```
/* Codes for sending the valid message to a memory buffer where it is saved or to a network for transmission */
.
.
OSTimeDly (2); /* Delay for low priority task start */ OSTimeDlyResume (Task_OutPortBPriority);
/* Resume Delayed Task_OutPortB */
    } /* End of while loop*/
} / * End of the Task_SendPortB function */
/*****************************************************************/
/* Codes of Task_Err */
static void Task_Err (void *taskPointer) {
/* Initial assignments of the variables and pre-infinite loop statements that execute only once */
char[ ] errMessage; /* Declare error message pointer. */
while (1) { /* Start an infinite while-loop. */
/* Check for Mailbox Error Message available (not NULL) */
errMessage = OSMboxAccept (MboxErrStrPointer);
/* Codes for the action on error message. */
if (strcmp (errMessage, "Invalid Character Found") == 0) {
/* Codes for the actions that need to be taken when invalid characters are found. For example, codes for displaying "Invalid Number Dialed. Dial Again" on an LCD. */
.
.
};
if (strcmp (errMessage, " Characters in the message exceeded the limit declared at task for read") ==0) {
/* Codes for actions needed on limit exceeded. Codes for displaying on an LCD "Message too long to Accept. Dial again". */
.
.
}
OSTimeDlyResume (Task_ SendPortB); /* Resume task_SendPortB
}; /* End of while loop*/
} / * End of the Task_Err function */
/*****************************************************************/
```

9.2.7 Queue-Related Functions

It is learnt from Example 9.18 that the semaphore communicates an IPC for an event occurrence. It is learnt from Example 9.19 that a mailbox communicates a pointer for a message, which may be larger or smaller. By using a queue, we can communicate an array of message pointers from the tasks. The message pointers can be posted into a queue by the tasks either at the back as in a *queue* or at the front as in a *stack*. (A priority message is posted in front.) A task can thus insert a given message for deleting either in the FIFO mode or in the LIFO mode. (*Note:* The IPC queue differs from the data structure queue with respect to the available methods for inserting an element into a queue. The OS controls the IPC queue.)

Refer to Figure 7.12. MUCOS permits a queue of an array of pointers. Let there be a pointer, **Qtop, to a queue of message pointers, and there be two pointers, * QheadPointer and *QtailPointer, which send and retrieve, respectively, the message pointer for the message. MUCOS queue functions for the tasks' IPCs are as per Table 9.7. MUCOS permits up to 65,536 message-pointers into a queue (i.e., the MUCOS queue size can be 65,536). The post-front function enables insertion such that the waiting task does LIFO retrieval of the message-pointer, hence of the message.

1. *Creating a Queue for an IPC. OS_Event* QMsgPointer = *OSQCreate (void **QTop, unsigned short qSize)* is used for creating an OS's ECB for the *QTop* and queue is an array of pointers at QMsgPointer. The array size can be declared as maximum 65,536 (0th to 65,535th element). Initially, the array QMsgPointer points to NULL.

Table 9.7 Queue Functions for the Intertask Communications

Prototype of Service and System Clock Function	***Parameters Returned and Passes***	*When is this Operating System (OS) Called?*
OS_Event OSQCreate (void **QTop*, unsigned short qSize)	**R and R**	OS creates a queue ECB. This creates and initializes an array of pointers for the queue at QTop. Queue can be of maximum size = qSize. QTop should point to top (zeroeth element of an array). ECB points at the QMsgPointer.
void *OSQPend (OS_Event *QMsgPointer, unsigned short timeOut, unsigned byte *Qerr)	**S and S**	Refer to text.
unsigned byte *OSQFlush (OS_EVENT *QMsgPointer)	**T and T**	To eliminate all the messages in the queue that have been sent. This function checks if a queue has a message pending at QMsgPointer (the queue front pointer at the ECB is not pointing to NULL). Function then returns all the message pointers between the queue front pointer and queue back pointer at the ECB. It returns error codes and QMsgPointer will point to NULL.
unsigned byte OSQPost (OS_EVENT*QMsgPointer, void *QMsg)	**U and U**	Sends a pointer of the message QMsg to the QMsgPointer at the queue back. The message inserts at a queue tail pointer in the ECB.
unsigned byte OSQPostFront (OS_EVENT *QMsgPointer, void * QMsg)	**V and V**	Sends QMsg to the QMsgPointer at the queue. It points to the queue head pointer in the ECB where pointer for QMsg now stores pushing other message pointers backward.[1]
unsigned byte OSQQuery (OS_EVENT *QMsgPointer, OS_Q_DATA *QData)	**X and X**	To get queue message's information and error information.

Column 2 refers to the corresponding explanatory paragraph in the text.

[1]Use OSQPostFront or OSQPost functions according to the message priority, if the message has a higher priority, post it at the front; else, post as usual in a queue. We can use post-front and post functions to build a queue in which an array of message pointers are stored and ordered according to their priorities. Queue is then called a prioritized ordered queue. Use of OSQPostFront and then OSQPend enable indirectly a LIFO mode of retrieval of a message pointer to priority message.

Task parameters passing R: The ***QTop* passes as pointer for an array of voids. The qSize is the size of this array (number of message pointers that the queue can insert before a read is within 0 and 65,535).

Returning R: The function OSQCreate ( ) returns a pointer to the ECB allocated to the queue. It is an array of voids initially, NULL if none is available.

Example 9.20 explains the use of OSQCreate function.

2. *Waiting for an IPC message at a queue.* Function *void *OSQPend (OS_Event *QMsgPointer, unsigned short timeOut, unsigned byte *Qerr)* checks if a queue has a message pending at ECB QMsgPointer (QMsgPointer is not pointing to NULL). The message pointer points to the queue front (head) at the ECB for the queue defined by QMsgPointer. It suspends the task if no message is pending [until either the message received or the wait period, passed by argument timeOut, finishes after (timeOut – 1) ticks of the system timer]. The queue head pointer at the ECB will later increment to point to the next message after returning the pointer for the message.

Returning S: The function returns pointer to a queue at ECB. It also returns the following on running the macros as under: (i) OS_NO_ERR returns true, when the queue message search succeeds; (ii) OS_TIMEOUT returns true, if queue did not get the message during the ticks defined by the timeOut; (iii) OS_ERR_PEND_ISR returns true, if this function call was from the ISR; (iv) OS_ERR_EVENT_TYPE returns true, when *QMsgPointer is not pointing to the queue message.

Task parameters passing S: (i) The OS_Event *QbackPointer passes as pointer to the ECB that is associated with the queue. (ii) It passes 16-bit unsigned integer argument timeOut. It resumes the task after the delay equals (timeOut – 1) count inputs (ticks) at the system clock. (iii) It passes *err, a pointer for holding the error code.

Example 9.20 explains the use of OSQPend function.

3. *Emptying the queue and eliminating all the message pointers.* Function *unsigned byte *OSQFlush (OS_EVENT *QMsgPointer)* checks if a queue has a message pending at QMsgPointer (the queue front pointer at the ECB does not point to NULL). The function returns all the message pointers between queue front pointer and queue back pointer at the ECB. It returns an error code and QMsgPointer at ECB. These will later point to NULL on return from the function.

Task parameters passing T: The OS_Event *QMsgPointer passes as pointer to the ECB that is associated with the queue.

Returning T: After the function OSQFlush ( ) executes, the error macros returns as follows: OS_NO_Err returns true, if the message queue flush succeeds, or OS_ERR_EVENT_TYPE returns true, if QMsgPointer is not pointing to the queue message queue.

4. *Sending a message pointer to the queue.* The function unsigned byte *OSQPost (OS_EVENT *QMsgPointer, void *QMsg)* sends a pointer of the message *QMsg. The message pointer QMsgPointer (queue tail pointer) points to the QMsg.

Task parameters passing U: The OS_Event *QMsgPointer passes as pointer to the ECB that is associated with the queue tail.

Returning U: After the function OSQPost ( ), the message pointer *QMsg* is passed for the message to *QMsgPointer and the error macros return the error code as follows: (i) OS_NO_ERR returns true, if queue signalling succeeded, (ii) OS_ERR_EVENT_TYPE returns true, if *QtailPointer is not pointing to the queue and (iii) OS_Q_FULL returns true, when queue message cannot be posted (QSize cannot exceed a limit set on creating the queue).

Example 9.20 explains the use of OSQPost function.

5. *Sending a message pointer and inserting it at the queue front.* The function unsigned byte *OSQPostFront (OS_EVENT*QMsgPointer, void *QMsg)* sends QMsg pointer to the QMsgPointer at the queue, but it is at the queue front pointer in the ECB where pointer for QMsg now stores, pushing other message pointers backwards.

Task parameters passing V: The OS_Event *QMsgPointer passes as pointer to the ECB that is associated with the queue. The second argument is the message QMsg address that is the queue front address.

Returning V: After the function OSQPostFront ( ) executes the following error macros returns as under: (i) OS_NO_ERR returns true, if the message at the queue front is placed successfully; (ii) OS_ERR_EVENT_TYPE returns true, if pointer QtailPointer is not pointing to message queue; or (iii) OS_Q_FULL returns true, if qSize was declared n and queue had n messages waiting for the read.

Example 9.20 explains the use of OSQPostFront function.

6. *Querying to find the message and error information for the queue ECB.* The function *unsigned byte OSQQuery (OS_EVENT *QMsgPointer, OS_Q_DATA *QData)* checks for a queue data and places that at QData. It also finds the error information parameters, on executing the following macros: OS_NO_ERR and OS_ERR_EVENT_TYPE.

Task parameters passing X: The function OSQQuery passes (i) a pointer of the queue at *QMsgPointer ECB and (ii) a pointer of the data structure at *QData.

Returning X: *QData* has pointer to the message at OSMsg, number of messages at OSNMsgs, OSQSize as queue size in terms of the number of entries permitted and list of the tasks waiting for the message. After the function, the following macros returns true: (i) OS_NO_ERR, when querying succeeds or (ii) OS_ERR_EVENT_TYPE, if *QMsgPointer is not pointing to queue message.

Example 9.20

Let a task, *Task_ReadPortA*, receive characters *QMsg and* put these into a queue, The task uses OSQPost to send an IPC for another task waiting (blocked) for these characters. An advantage is that the messages can be used as soon as available by another task without the completion of the whole message string as was the case in Example 9.19. The mailbox permitted only one message through a message pointer. Queue permits any number of messages till the queue gets full. Full means that the maximum array size defined for the queue is reached. Another advantage is that port A need not be the one sending characters or bytes, but can be an NIC (network input card) or any other input device sending a word, frame or a segment of a message that needs to be posted to another waiting task in a sequence. Further, any number of error messages sent by the Task_Err can also be posted into the same queue as priority messages. The codes get simplified.

To *Task_MessagePortA*, not only *Task_ReadPortA* but also another task, *Task_Err*, can send an error message on detecting an invalid character or if the limit of characters expected in the string is exceeded. The use of OSQPostFront is to send the errors as the priority message. OSQPend is used to wait for an IPC for the message as well as error message string. The steps in this operation are as follows.

1. Step *a*: ISR, *ISR_CharIntr* interrupt on the port A status for the availability of a message. (For example, the task is activated if, at port A, a key is pressed for sending the character or a network input data or a set of numbers are keyed on a keypad of mobile.) It puts the characters for message at portAData buffer. Semaphore *SemFlag1* (as in Example 9.16) posts an event occurrence as an IPC.
2. Step *b*: *Task_ReadPortA* waits for the *SemFlag1* and executes the codes that post the message, checks and, posts in front the error in the input message into a common queue.
3. It checks each *character or message* read at port A and sends a string, *errStr*, into a general message queue when the character or message has invalid character or message queue is full. For example, if the character or message is not a number in the case of a telephone number.
4. Step *c*: *Task_MessagePortA* waits for the *characters* or messages as an array of pointers. The task also sends the messages for display.
5. Step *d: Task_ErrLogins* also waits message for the error posted in a queue for error logins.

This example shows how the steps *a* and *b* synchronize by the IPC *SemFlag1*, how tasks at the steps *b* and *c* use queue to synchronizes and how the steps *b* to *d synchronize* using the queue functions of the MUCOS.

1. /* Codes are the same as in Step 1 Example 9.19, except that the statements are shown in bold for the queues. The mailbox-related statements are replaced by the queue-related messages. */
.
.
2. /* Preprocessor definitions for maximum number of inter-process events to let the MUCOS allocate memory for the Event Control Blocks */
#define OS_MAX_EVENTS 12/* Let maximum IPC events be 12 */
#define OS_SEM_EN 1/* Enable inclusion of semaphore functions in applications using MUCOS */
#define OS_Q_EN 1/* Enable inclusion of queue functions in applications using MUCOS */
/* End of preprocessor commands */
3. /* Prototype definitions for ISR and tasks for steps *b* to *d* above. */
static void ISR_CharIntr (void *IntrVectPointer);
static void Task_ReadPortA (void *taskPointer);
static void Task_MessagePortA (void *taskPointer);
static void Task_ErrLogins (*taskPointer);
4. /* Definitions for task stacks */
static OS_STK Task_ReadPortAStack [Task_ReadPortAStackSize];
static OS_STK Task_MessagePortAStack [Task_MessagePortAStackSize];
static OS_STK Task_ErrLoginsStack [Task_ErrLoginsStackSize];
5. /* Definitions for task stack size */
#define Task_ReadPortAStackSize 100 /* Define task stack size*/
#define Task_MessagePortAStackSize 100 /* Define task stack size*/
#define Task_ErrLoginsStackSize 100 /* Define task stack size*/
6. /* Definitions for task priorities. */
#define Task_ ReadPortAPriority 10 /* Define task 2 priority */
#define Task_ErrPriority 11 /* Define task 3 priority */
#define Task_MessagePortAPriority 12 /* Define task 4 priority */
#define Task_ErrLoginsPriority 14 /* Define task priority */
7. /* Prototype definitions for semaphore */
OS_EVENT *SemFlag1*; / Needed for using the semaphore as a flag for inter-process communication between portstatus interrupt, ISR_CharIntr and port read task, Task_ReadPortA Port A read task waits for semaphore till port A interrupts and puts the message in port A data buffer. */
OS_EVENT *SemCountLimitFlag*; / *Needed when using the semaphore as flag for reaching the limits of semaphore count in the inter-process communication between port read and port read decipher task. Port reading has to wait for port read */
OS_EVENT **SemKey*; /* Needed when using the semaphore as resource key by Task_ReadPortA and Task_Err */
OS_EVENT *QMsgPointer; / *Needed when using queued message between steps *b* and *d* and steps *d* and *e* */
void **QMsgPointer [QMessagesSize]; /** *Let the maximum number of message-pointers at the queue be QMessagesSize. */

OS_EVENT *QErrMsgPointer; / *Needed when using a queued message between steps *c* and *d* */
void ***QErrMsgPointer [QErrMessagesSize]; /** *Let the maximum number of error message-pointers at the queue be QErrMessagesSize. */
9. /* Define both queues array sizes. */
#define **QMessagesSize** = 64; /* Define size of message-pointer queue when full */
#define QErrMessagesSize = 16; /* Define size of error message-pointer queue when full */
10. /* Codes for port input reading from Port A and storing a character or message at a queue or buffer. Alternatively, modify the code for reading from a port at NIC or any other device or peripheral. */
.
.
11. /* Codes as per Example 9.7, Steps 3 to 8. However, before the 'OSStart ( );', the semaphore and queue must be created and initialised as under: */
SemFlag1 = OSSemCreate (0) /* Declare initial value of semaphore = 0 for using it as an event signaling flag*/
Sem*CountLimitFlag* = OSSemCreate (0) /* Declare initial value of semaphore = 0 as an event signaling flag*/
SemKey = OSSemCreate (1) /* Declare initial value of semaphore = 1 for using it as a resource key*/
12. / * Declare initial count as 0 as a counter that gives the number of times a task, which sends into a buffer that stores a character or message stream, ran minus the number of times the task which used the character or message from the stream ran from the buffer */
SemCount = OSSemCreate (0);
13. /* Create Two queues for the tasks, one general purpose queue and another for error logins only. The queue for errors is posted messages after the Task_messagePort*A* selects error messages from the general queue. */
/* Define a top of the message pointer array. QMsgPointer points to top of the Messages to start with. */
QMsgPointer = OSQCreate (&QMsg [0], QMessagesSize);
/* Define a top of the message pointer array. QMsgPointer points to top of the Messages to start with. */
QErrMsgPointer = OSQCreate (&QErrMsg [0], QErrMessagesSize); /* Needed when using mailbox message between steps *c* and *d* to pass a string message pointer*/
14.
.
.
15. /* Create Tasks as per Step 3 defining by tasks—Task_ReadPortA, Task_Err, Task_MessagePortA and Task_ServiceMessage and the stack sizes, other TCB parameters. */
OSTaskCreate (Task_ReadPortA, *void (*) 0, (void *) &* Task_ReadPortA*Stack* [Task_ReadPortA*StackSize*], Task_ReadPortAPriority);
OSTaskCreate (Task_MessagePortA, *void (*) 0, (void *) &* Task_MessagePortA*Stack* [Task_MessagesPortA*StackSize*], Task_MessagePortAPriority);
OSTaskCreate (Task_ErrLogins, *void (*) 0, (void *) &* Task_ErrLogins*Stack* [Task_ErrLogins*StackSize*], Task_ErrLoginsPriority);
16. /* Codes same as at Steps 9 to 21 in Example 9.16 */
;
.

```
17. }/ * End of the ISR_CharIntr function */
/*****************************************************************/.
18. /* Codes for Task_ReadPortA redefined to use the key, flag and 16-bit value and mailbox*/
static void Task_ReadPortA (void *taskPointer) {
/* Initial assignments of the variables and pre-infinite loop statements that execute only once */
char [ ] *portAdata;
Char [ ] msgBeginStr = "Messages Begin";
boolean QFull = false,
unsigned short msgCount = 0;
unsigned byte [ ] msgBuffer / *Declare initial value of msgBuffer */
/*Declare an error string for step d error invalid message data found to the queue. */
char [ ] ErrStr1 = "Invalid Message Data Found";
/* Declare an error string message for task at step d when the limit exceeds. */
char [ ] ErrStr2 = "Array Size exceeded the Limit. Queue Full";
boolean invalid = false; /* Declare invalid variable 'false' and will be assigned 'true' when character or
message read is found invalid. * /
19. /* Start an infinite while-loop and Wait for SemFlag1 1 by OSSemPost function of character or message
availability from ISR_CharInt */
while (1) {
OSSemPend (SemFlag1, 0, SemErrPointer);
/* Post the string to initiate start of the message */
if (msgCount == 0) {OSQPostFront (QMsgPointer, MsgBeginStr);}
20. /* Actions on maximum size exceeding the string buffer size */
if ( msgCount > = QMessagesSize) {OSQPostFront (QMsgPointer, ErrStr2);
Qfull = true; msgCount = 0;}
/* Code for checking any invalid character or message in port A buffer */
if (invalid = = true) {msgCount = 0; OSQPostFront (QMsgPointer, ErrStr1);}
};/* End of Codes for the actions on reaching the queue array limit of the putting the message pointers into
the buffer or on finding an invalid message character or data */
21. /* Write the array element that returned as Port A data into the port A input string */
if (Qfull = = false || invalid = = false)
{OSQPost (QMsgPointer, &portAdata);
/*Let message counter value increase after one character or message has been put into the String, holding
the character or message stream*/
msgCount ++;}
22. /*Let Task_MessagePortA start by delay */
23. OSTimeDly (2);
24. /* Other codes for read port task*/
25. }; /* End of while loop*/
26. }/ * End of Task_ReadPortA function */
27. /*
/*****************************************************************/
```

28. /* **Codes for the Task_MessagePortA** */
static void Task_MessagePortA *(void *taskPointer)* **{**
29. /* Initial assignments of the variables and pre-infinite loop statements that execute only once */
void * message;
30. *while (1)* { /* Start an infinite while-loop. */
31. /* Wait for Queue Message Pointer available (not NULL) */
&message = **OSQPend (QMsgPointer, 0, QErrPointer);**
32. /* Find if the message has invalid character or the messages found as Error Messages. Check for queue Error Message available (not NULL) */
if (strcmp ((char *) message, "Invalid Message Data Found ")==0) {**OSQPost (QErrMsgPointer, message); OSTimeDly (2)};**
33. if (strcmp ((char *) message , "Array Size exceeded the Limit. Queue Full") == 0){**OSQPost (QErrMsgPointer, message); OSTimeDly (2);**}; OSTimeDlyResume (Task_ErrPriority); /* Let delayed higher priority task resume. */
&message = **OSQPend (QMsgPointer, 0, QErr)**;
34. /* while (Strcmp (char *) message, "Messages Begin"){
/* Codes for servicing as per the valid message to a memory buffer for saving or to a network or to dial and transmit */}
35. /* To exit the infinite loop at the task that has been assigned a higher priority and to let the next priority Errlogins task run, let us call the OS delay function for wait of 20 ms (two OS timer ticks). This is the method to let the other task of lower priority execute. */
OSTimeDly (2);.
36. OSTimeDlyResume (Task_ReadPortAPriority); /* Resume Delayed Task_ReadPortA */
}; /* End of while loop*/
37. }/ * **End of the Task_MessagePortA function** */
/**/
38. /* **Codes of Task_ErrLogins** */
static void Task_ErrLogins (*void *taskPointer*) {
/* Initial assignments of the variables and pre-infinite loop statements that execute once only*/
void **errorLogged; /** Declare error message pointer. */
39. *while (1)* { /* Start an infinite while-loop. */
40. /* Check for Mailbox Error Message available (not NULL) */
&errorLogged = **OSQPend (QErrMsgPointer, 0, QErrPointer)**;
41. /* Codes for the action as per the error logged in */
42. if (errorLogged = = " Invalid Message Data Found") {
/* Codes for actions needed on invalid character or message found. For example, codes for displaying "Invalid Number Dialed. Dial Again" on an LCD. */
.
.
};
43. if (errorLogged = = " Array Size exceeded the Limit. Queue Full ") {

```
44. /* Codes for actions needed on limit exceeded. Codes for displaying on an LCD "Message too
long to Accept. Dial again". */
.
.
}
OSTimeDlyResume (Task_MessagePortAPriority); /* Resume the ServiceMessage */
}; /* End of while loop*/
45. } /* End of the Task_MessagePortA function */
/*************************************************************/
```

9.3 RTOS VxWorks

For sophisticated embedded systems, there is a popular RTOS, VxWorks from WindRiver® (*http://www.windriver.com/*). VxWorks is a high-performance, Unix-like, scalable RTOS, and support to ARM, ColdFire, MIPS, Pentium, Intel X-Scale, Super H and other popular processors for embedded system design. VxWorks RTOS design is hierarchical (Section 8.9) and is for hard real-time applications. It supports kernel mode execution of tasks for fast execution of application codes.

VxWorks is supported with powerful development tools that make it easy and efficient to use. VxWorks supports many advanced processor architectures. VxWorks supports 'Device Software Optimization', which is said to be a new methodology that enables the development and the running of the device software faster, better and more reliably. VxWorks has been found to be the most popular RTOS in a survey in 2006 (*http://www.embedded.com/columns/showArticle.jhtml?articleID=187203*). VxWorks 6.x is its latest version.

VxWorks provides for the following.

1. Multitasking environment using scheduler which supports IEEE standard POSIX scheduler and which also supports the in-house developed scheduler.
2. Supports ability to run two concurrent OSs on a single processing layer.
3. Multiple file systems (Section 8.6.2) and systems that enable advanced multimedia functionality.
4. Synchronization using a full range of IPC options (Section 7.9) that includes. (i) event-signalling flag, (ii) mutually exclusive access using resource key (mutex), (iii) counting mechanism using three types of semaphores in the tasks, (iv) queue, (v) socket and ISRs and includes POSIX standard semaphore and other IPCs (Sections 7.8.3, 7.9 to 7.15 and 8.12). Also supports real-time processes IPCs. It also support openSource TIPC (transparent inter-process-communication) protocal for network and clustered systems environment. [PTTS 1.1 is latest release in December, 2007 for Linux and VxWorks.]
5. Different context saving mechanism for the tasks and ISRs (tasks have separate TCBs and stacks, and ISRs use a common stack due to nesting of the calls).
6. Watchdog timers.
7. Virtual IO devices including pipes and sockets (Sections 7.14 and 7.15).
8. Virtual memory management functions.
9. Power management functions that enhance the ability to control power consumption, and automatic detection and reporting of common memory and other errors.
10. Interconnect functions that a support large number of protocols, including IPv4/IPv6 dual mode stack ready APIs.

VxWorks TCB saves the following task information for each task.

1. Control information for the OS that includes *priority*, *stack size*, *state* and *options*.
2. CPU context of the task that includes PC, SP, CPU registers and task variables.

VxWorks also provides:

1. Pipe drivers for IPCs and pipe is an IO virtual device.
2. Network transparent sockets.
3. Network drivers for shared memory and Ethernet.
4. RAM 'disk' drivers for memory-resident files.
5. Drivers for SCSI, keyboard, VGA display, disk and parallel port of a computer system, HDD, diskette, tapes, keyboard and displays.
6. VxWorks 6.x provisions for processor abstraction layer. It enables application system design by user when using new versions of a processor architecture.

VxWorks IO system also includes the POSIX standard asynchronous IOs and UNIX standard buffered IOs. It also provides for simulator (VxSim) (Section 14.2.3), software logic analyser (WindView), Code coverage study tool, MemScope, StethoScope (Section 1.4.7 Table 1.2) network facilities between VxWorks and TCP/IP network systems. For many other facilities, we can refer to VxWorks programmer's guide and VxWorks Network Programmer's Guide provided with the product.

9.3.1 Basic Features

A summary of the important features of VxWorks that are essential in a sophisticated embedded system design are as follows:

1. VxWorks is a scalable OS (only the necessary OS functions become part of the applications codes, thus having reduced memory requirements). The run-time configurable feature gives a higher performance in VxWorks. The functisons needed for task servicing, IPC and so on must be predefined in a configuration file included in the user codes. Pre-emptive latency minimization is there as not all the functions are at the kernel (Section 8.1).
2. RTOS hierarchy includes timers, *signals*, TCP/IP *sockets*, queuing functions library, NFS, RPCs, Berkeley Port and sockets (Section 7.15), pipes (Section 7.14), Unix-compatible loader, language interpreter, shell, debugging tools and linking loader for Unix. (These are similar to system tasks. The scheduler runs these, as it runs the ISRs.)
3. For multitasking, like MUCOS, VxWorks employs a preemptive scheduler (Section 8.10.3). VxWorks is a preemptive priority based scheduler. It provisions for 256 priority levels within 0 to 255. A task context saves fast when the CPU access changes to a higher priority. However, VxWorks offers a flexibility that there can be a set of tasks (different tasks of the same priority), which run in time-slice (round robin) mode (Section 8.10.2). Each task in a set of tasks executing round robin runs for a given number of system-clock ticks and after timeout becomes the last in a queue of the set. We can use preemptive priority and time-slicing scheduling simultaneously. (*Note:* The *preemption priority* and POSIX FIFO scheduling are identical).
4. Refer to Examples 9.19 and 9.20. The ISR interrupt flag was checked and reset for new interrupt in ISR_CharIntr and the ISR passed a semaphore (message) to run the waiting task Task_ReadPortA. *VxWorks RTOS schedules the ISRs separately and has special functions for interrupt handling* [Refer to Section 9.3.3 Table 9.10].
5. VxWorks has system-level functions for RTOS initiation and start, system clock ticks (interrupts) initiation and the ISR functions, ISR connecting to interrupt vector and masking functions. Recall Sections 7.8.4

and 8.10.3. For the critical section (section of codes which access the shared resources or variables) in the ISRs, VxWorks has the interrupts disabling and enabling functions that execute at entering and exiting the section, respectively (semaphore pending functions as event-signalling flag, and counting should not be invoked in the ISRs).

6. If a task is expecting a message from another task, which is being deleted by using the task delete function, then RTOS inhibits the deletion when an option called 'Deletion Safe' is used.
7. VxWorks has task service functions (Table 9.8). VxWorks *task creation* (initiation) by itself does not make a task in a list of active tasks. Active task means that it is in one of the three states; ready, running, or waiting (blocking or pending). VxWorks not only has the task creating, running, waiting (blocking or pending till a time out or till resource available), suspending (inhibiting task execution) and resuming, but also the functions for task spawning (creating followed by activating). VxWorks also includes the task-pending cum suspending and pending cum suspension with timeout functions. VxWorks also has the tasks, which have a state and an inherited priority (Section 7.8.5).
8. VxWorks has task delay functions and task delaying cum suspending function (Section 9.3.2).
9. VxWorks has the shared memory allocation functions and bounded ring buffer allocation for sharing the memory and buffers between the tasks and ISRs. To improve the performance of RTOS, VxWorks provides a shared address in memory to all the tasks. This helps in fast access through the pointers. A pipe need not be allocated a separate memory space. Of course, there is an attendant risk due to a possible illegal access.
10. VxWorks has IPC functions that are more sophisticated than MUCOS functions. Recall that MUCOS has identical semaphore functions for event-signalling flags, resource-acquiring keys and counting semaphores. Recall the use of the semaphore SemKey with OSSemPend and OSSemPost functions on SemKey. SemKey was used as a resource-acquiring key by the various tasks in Examples 9.17 to 9.20. VxWorks provides for three types of semaphores separately (POSIX-IPCs and TIPCs are additional).
11. VxWorks has special features for mutual exclusiveness in a critical region. We use a mutex semaphore for the resource key when using VxWorks. [Only the task that takes the resource key through a mutex semaphore can release (give or post) the key. No other task can release it. This provides mutual exclusiveness.] One type of semaphore used as mutex has the following special features: (i) One task can be protected from being deleted by any other task. Thus, unprotected deletion cannot occur when using a mutex semaphore function with the deletion safe option. At mutex creation the option is selected to include the deletion protection. (ii) When a task acquires the key using the mutex priority inversion can be prevented with the priority inversion safe option. The priority assignment of high-priority task can now inherit (during execution of critical section) so that in case of pre-emption by an intermediate priority task, the high-priority task does not get blocked (Section 7.4). This prevents priority inversion situations during execution of the critical section.
12. The unlock ( ) and lock ( ) functions are available for the tasks and interrupts, for disabling other task but not interrupts or enabling pre-emption (task switching) as alternative to resource locking by mutex semaphore.
13. The lock and unlock functions in VxWorks do not cause the priority inversion problem (Sections 7.8.5 and 8.10.3). The priority first inherits and then returns to the original ones.
14. Let us recall Figure 7.4(*a*) in Section 7.7.6 to understand P and V mutex semaphores used for locking the resources. VxWorks provides for P and V semaphore functions also.
15. Unlike MUCOS, VxWorks has no separate functions for the mailbox that distinguish the mailbox from the message queue. VxWorks messages can be queued. It provides for sending messages of variable length into the queues. (MUCOS queue functions permit a pointer only for a

message and a queue is an array of message pointers.) MUCOS messages can insert such that these retrieve in the FIFO method or in the LIFO if message pointer is posted in the front instead of at the back when a message is of higher priority. VxWorks also supports this post and post-front feature as in MUCOS. There are additional special features with the message queues in VxWorks.

16. In addition to queues, VxWorks provides the IPCs through the pipe (Section 7.14) into which an ISR or task can write by invoking functions for pipe open ( ). Task can read from a pipe by using open ( ) and read ( ) functions. A VxWorks pipe is a virtual IO device functions (Refer to Section 9.3.4).
17. The features of scheduler design compatible with POSIX 1003.1b can also be used. The POSIX library can be included in VxWorks.
18. The timings taken by the various RTOS functions are similar to the ones given in Table 8.12.

The following subsections describe the specific VxWorks functions.

9.3.2 Task Management Library at the System Library Header File

Each task divides into eight states (places). First four of these are also available in MUCOS tasks.

1. Suspended (idle state just after creation or state where execution is inhibited). [Refer to the use of OSTaskSuspend function for FirstTask in the step 12 of Example 9.8.)
2. Ready (waiting for running and CPU access in case scheduled by the scheduler but not waiting for a message through IPC).
3. Pending (the task is blocked as it waits for a message from the IPC or from a resource; only then will the CPU be able to process further). [Refer to OSSemPend, OSMboxPend and OSQPend functions in Examples 9.16 to 9.20].
4. Delayed (sent to sleep for a certain time interval). [Refer to the use of OSTimeDly in Examples 9.16 to 9.20].
5. Delayed + suspended (delayed and then suspended if it is not pre-empted during the delay period).
6. Pended for an IPC + suspended (pended and then suspended if the blocked state does not change).
7. Pended for an IPC + delayed (pended and then pre-empted after the delayed time interval).
8. Pended for an IPC for an IPC + suspended + delayed (pended and then suspended after the delayed time interval).

VxWorks and kernel library functions are in the header files, 'vxWorks.h' and 'kernelLib.h'. Task and system library functions are in 'taskLib.h' and 'sysLib.h'. For logging, the library function is 'logLib.h'. Library functions are given in Tables 9.8, 9.9 and 9.10. Table 9.8 lists the task-state function transitions. Table 9.9 gives the task creation, naming and control functions. Table 9.10 gives the interrupt service functions.

1. *Creating and activating a task by TaskSpawn function.* The function for task creating and activating is taskSpawn ( ). Prototype use is unsigned int *taskId* = taskSpawn (*name*, *priority*, *options*, *stacksize*, *main*, arg0, arg1, ..., arg8, arg9). A memory is allocated to the stack as well as to the TCB. The task identified by taskID will be assigned a stack of size *stacksize* with arg0 to arg9 passed to the stack. It is also assigned a TCB pointer, which points to the entry point of function *main* that the system executes first.

Recall that in MUCOS we were accessing a task by its priority argument, for example, OSTimeDlyResume (*Task_ReadPortPriority*). We access a task in VxWorks by its ID, *taskID*. [Refer to *taskID* argument at the functions in the following subsections]. *taskID* is a 32-bit positive number. Further, when a task of *taskID* = 0 is referred, VxWorks assumes that the calling task is being referred.

New task *taskID* gets the name, priority, options and stack size on spawning. If the NULL pointer is used as the argument for *name*, then conventionally the *name* has two characters tN (character t followed by number N) prefixed with the *name*.

Table 9.8 Functions for the Task State Transitions

Function	*Present State*	*Next State*	*Previous Function Call before the Call or Previous States*
taskResume ( )	Suspended	Delayed or pended	taskInit ( ) (task must have been initiated from the idle state)
taskResume ( ) or taskActivate ( )	Suspended	Ready	taskInit ( )
taskSuspend ( )	Delayed	Suspended	taskSpawn ( ) or taskActivate ( )
taskSuspend ( ) *After a timeout*	Delayed	Ready	Suspended
taskSuspend ( ) *After a wait for a resource*	Pended	Suspended	Ready
semGive ( ) or msgQSend ( )	Pended	Ready	Suspended
semTake ( ) or msgQReceive ( )	Ready	Pended	Delayed or suspended
taskDelay ( )	Ready	Delayed	Pended or suspended
taskSuspend ( )	Ready	Suspended	Delayed or pended
taskInit ( )	Unknown	Suspended	
exit ( )[1]	Suspended	Terminated	
taskDelete ( )[2]	Suspended	Eliminated	

[1]Terminate the task.
[2]Terminate the task and free the memory.

Table 9.9 Task Creation, Naming and Control Functions

Function	*Description*
taskSafe ( )	Protects the calling task from deletion
taskUnsafe ( )	Permits deletion of the task protected earlier
taskDelete ( )	Deletes a task
taskRestart ( )	Restarts (create again)[1] the running task as the earlier run returned error
taskActivate ( )	Task activates if initialized earlier
taskSpawn ( )	Creates as well as activates
taskName ( )	Returns the task name that associates with the taskID passed as argument
taskNameID ( )	Returns the taskId that associates with the task name passed as argument
taskIDVerify ( )	Verifies if a task of taskID in the argument is available
taskIDSelf ( )	Returns the taskID of the task
taskIDListGet ( )	Returns an array of all ready tasksIDs
taskInfoGet ( )	Returns information (parameters of the task)
taskRegsGet ( )	Returns the registers of the task
taskRegsSet ( )	Sets the registers of the task
taskOptionsSet ( )	Sets the task options
taskOptionsGet ( )	Returns task options defined earlier
taskIsSuspended ( )	Checks if the task is in a suspended state
taskIsReady ( )	Checks if the task is in a ready state
taskTcb ( )	Returns the pointer to the task control block

(*Contd*)

Function	*Description*
taskPriorityGet ( )	Returns the task priority
taskLock ( )	It is used at the beginning of critical section. It disables other tasks (not ISRs) and thus rescheduling.[2] Priority pre-emption when the task is running
taskUnlock ( )	It is used at end of critical section in a task and it enables other tasks and thus rescheduling.[3] Priority pre-emption when the task is running after the critical section
taskPrioritySet ( )	Sets the priority within 0 and 255
kernelTimeSlice (int numTicks)	Defines time slice per task after enabling round robin running of the tasks. When numTicks = 50, after 50 system clock interrupts define the time-slicing period

[1]Allocate the memory with the allocated stack and control blocks at the beginning.
[2]Scheduler cannot block when the task is running, although a higher-priority task needs to be scheduled.
[3]Scheduler can block when the task is running and when a higher-priority task needs to be scheduled.

1. Function 'unsigned int taskIdSelf ( )' returns the identity of the calling task.
2. Using a function 'unsigned int [ ] *listTasks* = taskIdListGet ( )' will return the tasked list of all existing tasks needed in the array, *listTasks*.
3. Function taskIDVerify (*taskId*) verifies whether the task taskId exists.

System-task has the priorities upto 99 and task-highest priority is 100 by default. User task priorities are between 101 and 255. Lowest priority means task of priority 255. In VxWorks, the priority numbering scheme is lower the number, the lower the priority. In POSIX, the priority numbering scheme is the reverse of VxWorks (the lower the number, the lower the priority). Priority numbers below 100 are used for the system-level and scheduler-level processes in VxWorks (MUCOS system-level and scheduler-level processes use the highest eight and lowest eight priorities reserved for them).

A task may use priority-functions. For example, there are three functions in taskLib. A function can find the *priority* using 'taskPriorityGet (*taskId*, *&priority*)'. We can change the task priority dynamically. Function 'taskPriorityPut (taskId, *newPriority*)' will reassign the *priority* = *newPriority* for the task that *taskId* identifies.

The options definable on spawning are the following.

1. An *option* is VX_PRIVATE_ENV. It means that the task must be executed in the private environment. The task is then not a public environment task.
2. An option is VX_NO_STACK_FILL. It means no stack fills with the hexadecimal bytes 0xEE. (Stack-filling option allocates stack with filling of stack with byte, 0xEE at each stack address. Use of the option facilities later finding of unused stack in memory. We can use VX_NO_STACK_FILL after the developed program has been debugged and unused memory spaces has been compacted by reallocation of stack sizes.) A sufficient stack size should be declared by *stacksize*. VxWorks reserve a part of the stack as a buffer to protect the stack from overflow, which may lead to unpredictable behaviour of the system. To start with, task stack fills the bytes 0xEE. (When using simulator VxSim for the VxWorks application, VxSim adds an additional 8000 bytes to the *stacksize*. Thus, the stacks for interrupts that it simulates also become available.) A library function, 'unsigned int checkStack (taskId)' returns the stack usage. It first finds the unused stack area by counting the number of bytes from the end with 0xEE and then subtracts counts from the *stacksize*.
3. An option is VX_FP_TASK. It means that the task must be executed with the floating-point mode of processor. (The precision is higher when using floating-point processor. Time taken is smaller when using integer number processing for floating-point operations.)

4. An option is VX_UNBREAKABLE. It means disable the breakpoint at the task during execution. (Breakpoints are provided for help during debugging.)

In place of option names, the hex values, 0x80, 0x100, 0x8 and 0x2, respectively, can be used for the four options described above. Multiple options can be given as an argument. For example, VX_PRIVATE_ENV | VX_NO_STACK_FILL selects both options in a task when spawning. Like the priority task, options can also be reassigned using taskOptionsSet ( ) and taskOptionsGet ( ) functions. (The | sign is used between multiple options because & sign in C refers to the address of the succeeding variable.)

The argument *main* is the main routine address. MUCOS and for many RTOSes, *main* ( ) is the function, which is called by the RTOS first. Refer to Example 9.7. The *main* function is used to create a task that executes first, *FirstTask.* FirstTask, when it executes later, initiates the system timer. It then creates (activates as well) the application tasks and suspends itself to let the OS schedule and run the application tasks. In VxWorks, the *main* ( ) function analogue may be used. It is *schedule* ( ).

When using VxWorks, unlike the Unix-operating system or MUCOS ( ), all tasks can be spawned as peers (i.e., every task is independent and no task calls or spawns another task). This means that a task can be is similar to FirstTask in Example 9.7; a starting task (parent task) need not be spawned first. The starting task spawns daughter tasks and then suspends itself to prevent scheduling of the parent in MUCOS. The daughters in a parent task are spawned in VxWorks only when the parent is a server task that concurrently processes the daughter task.

Ten arguments from arg0 to arg9 can be passed into the main routine. These arguments give the start-up parameters.

An important point to note is that MUCOS OSTaskCreate creates the task as well as activates it (puts it in the list of tasks to be scheduled). These functions are separate in VxWorks. The taskInit ( ) and taskActivate ( ) can also be used separately in place of taskSpawn function. However, unless a greater control for scheduling is needed first by creation and then later by activation in an application, we prefer to use taskSpawn function in first instance itself.

Example 9.23 will explain the use of taskSpawn function.

2. *Task suspending and resuming functions.* Function taskSuspend (*taskId*) inhibits the execution of task identified by *taskId.* Function taskResume (*taskId*) resumes the execution of task identified by *taskId.* Function taskRestart (*taskId*) first terminates a task and then spawns again with its original assigned arguments. This function is used in certain situations. The priority might have been reassigned in between, and now the original priority is to be restored. Similarly, the start-up parameters might have to be restored.

3. *Deleting and protecting from deletion.* Function taskDelete (*taskId*) not only inhibits permanently the execution but also cancels the allocation of the memory block for the task stack and TCB. Deletion thus frees the memory. The task deleted is one identified by argument *taskId.* Function 'exit (*code*)' deletes the task itself but stores the code at a TCB field *exitcode.* The debugger can examine the TCB using the code.

A task should not be deleted when it has a resource key because the key can then never be released in case deleted. Protection is available to the task by using a function taskSafe ( ) before entering its critical region and using function taskUnsafe ( ) function at the end of the region. When using the mutex semaphore, we can alternatively select an option, SEM_DELETE_SAFE when creating it (Refer to Section 9.3.4).

Why do we use the task delete or exit function? It is because that many times system resources have to be reclaimed for reusing; memory may be a scarce resource for the given application. TCB and stack are the only resources that are automatically reclaimed. There is no saving of tasks spawned by other tasks by the kernel. Each task should itself execute the codes for the following.

1. Memory de-allocation.
2. Ensure that the waiting task gets the desired IPC.
3. Close a file, which was opened before.
4. Delete daughter tasks when the parent task executes the exit ( ) function.

4. *Delaying a task to let a lower-priority task get access.* The function 'int sysClkRateGet ( )' returns the frequency (system ticks and thus system-clock interrupts per second). Therefore to delay 0.25 seconds, the function taskDelay (sysClkRate Get ( )/4) is used. Recall the use of OSTimeDly and later OSTimeDly to resume in MUCOS (see Examples 9.16 to 9.20). This lets a task of lower priority to run. VxWorks taskDelay (NO_Wait) will allow other tasks of the same priority or lower to run (because the timeout interval is zero). No delayed task resumption is needed in the other priority task that runs after this function. Function nanosleep (1,000,000) will delay the task by 1ms. It is a POSIX function. Integer arguments define the number of nanoseconds for delay (sleeping).

9.3.3 VxWorks System Functions and System Tasks

The first task that a scheduler executes is UsrRoot from the entry point of usrRoot ( ) in file install/Dir/target/config/all/usr/Config.C. It spawns VxWorks tools and does following tasks: The root terminates after all the initializations. Any root task can be initialized or terminated. The set of functions, tLogTask, logs the system messages without current task context IO. The daemon (a set of large number of functions) supports the task-level network functions. The exception-handling functions are at tExcTask. It has the highest priority. It should not be suspended, deleted or assigned lesser priority. By using it, the system reports exceptional conditions that arise during running the scheduler and tasks.

An important set of functions that are also target-specific is tWdbTask. The user creates it to service the requests from Tornado target server. It is a target agent task.

1. *System clock and watchdog timer-related functions.* VxWorks sysLib is the system library and is in a header file kernelLib.h. The following important functions are present.

1. Function sysClkDisable ( ) disables the system clock interrupts and sysClkEnable ( ) enables the system clock interrupts.
2. Function sysClkRateSet (TICK numTicks) sets the number of ticks per second. It thus defines the number of system clock interrupts per second. Function sysClkRateGet ( ) returns the system ticks (system clock interrupts) per second. sysClkRateSet (100) will set the system clock tick after every 1/100 second (10 ms). This function should be called in the main ( ) or start up FirstTask on as a starting function. There is a 64-bit global variable, 'vxAbsTicks.lower'. Variable lower that increases after each tick and vxAbsTicks. Variable 'vxAbsTicks.upper' that increments after each 2^{32} ticks. 'TICK' is defined by type def as following.

```
typedef struct (
unsigned long lower;
unsigned long upper;
) TICK
TICK vxAbsTicks;
/* Function' unsigned long tickGet ( )' returns vxAbsTicks.lower. */
```

3. Function sysClkConnect ( ) connects a C function to the system clock interrupts.
4. Function sysAuxClkDisable ( ) disables the system auxiliary clock interrupts and sysAuxClkEnable ( ) enables the system auxiliary clock interrupts.
5. Function sysAuxClkRateSet (numTicks) sets the number of ticks per second for an auxiliary clock. It thus defines the number of system auxiliary clock interrupts per second. Function sysAuxClkRateGet ( ) returns the system auxiliary clock ticks (system clock interrupts) per second.
6. Function sysAuxClkConnect ( ) connects a C function to the system auxiliary system clock interrupts.

7. Function 'WDOG_ID wdCreate ( )' creates a watchdog timer. Statement 'wdtID = wdCreate ( );' creates a watchdog timer, which later identifies by wdtID. There is a function STATUS wdStart (*wdtID*, *delayNumTicks*, *wdtRoutine*, *wdtParameter*). The timer created starts on calling this function. The parameters that should pass as the arguments of this function are the following: (i) *wdtID to* define the identity of the watchdog timer; (ii) *delayNumTicks* to let the timer interrupts after the number of system clock interrupts of the system equal to *delayNumTicks*; (iii) *wdtRoutine*, a function called (not task or ISR) on each interrupt; (iv) wdtParameter an argument, which passes to wdtRoutine. A started watchdog timer, wdtID cancels on calling STATUS wdCancel (*wdtID*). A watchdog timer, wdtID de-allocates the memory on calling STATUS wdtDelete (*wdtID*).

2. *Defining time-slice interval for round robin time-slice scheduling.* Function kernelTimeSlice (int numTicks) controls the round robin scheduling and time slicing turns on and preemptive priority scheduling turns off for the tasks of equal priority. Suppose there is a system clock tick every millisecond. kernelTimeSlice (50) will set the time slice period as 50 ms. This is the time each task is allowed to run before CPU control relinquishes for another equal priority-task.

The codes #define TIMESLICE sysClkRateGet ( ) and later sysClkRateSet (1000); kernelTimeSlice (TIMESLICE); TIMESLICE = TIMESLICE/60;' will set time slice = 1000/60 ms.

3. *Interrupt-handling functions.* Table 9.10 gives the interrupt service-related functions. Refer to Section 4.4.1. An internal hardware device (interrupt source or interrupt source group) auto-generates an interrupt vector address, ISR_VECTADDR as per the device, for example, timer. Exceptions are defined in the user software. For exceptions, ISR_VECTADDR needs to be defined by intVecSet ( ) function. Function intConnect ( ) connects the ISR_VECTADDR to a C function for that ISR. Device driver uses this function as follows: a lock function used as 'int *lock* = intLock ( );' disables the interrupts. It returns an integer *lock*. Using the same integer as argument in the unlock function, we enable the interrupts. An unlock function used as 'intUnlock (*lock*);' enables the interrupts.

VxWorks provides for an ISR design that is different from a task design.

1. *ISRs have the highest priorities and can pre-empt any running task.* It arises because ISR is needed because of internal device events (e.g., from on-chip timers) and because of exceptions and signals (user-defined software interrupts on certain error conditions).

Table 9.10 VxWorks Interrupt Service Functions

Function	*Description*	*Function*	*Description*
intLock ( )	Disables Interrupts[1]	intUnlock ( )	Enables Interrupts[2]
intVecSet ( )	Set the interrupt vector[3]	intCount ( )	Counts number of interrupts nested together
intVecGet ( )	Get interrupt vector	intVecBaseSet ( )	Sets base address of interrupt vector
intVecBaseGet ( )	Get interrupt vector base address	intLevelSet ( )	Sets the interrupt mask level of the processor
intContext ( )	Returns true when calling function is an ISR	intConnect ( )	Connects a C function to the interrupt vector

[1, 2] The meanings of these has been explained in the text. Also refer to Section 7.7.2. It can be used in a task critical region as a last option, because it increases the interrupt latency periods of all sources.

[3] For 'exception' only. For hardware internal device interrupts, the interrupt vectors are fixed, cannot be set.

2. *An ISR inhibits the execution of tasks till return.*
3. *An ISR does not execute like a task and does not have regular task context. It has a special ISR context.*
4. While each task has its own TCB that includes its own stack pointer, unless and otherwise not permitted by a special architecture of a system or processor, *all ISRs use same interrupt stack.* In case of such special architecture, in place of interrupt-servicing support functions, the VxWorks stacks of ISRs can be used similar to task-stacks and codes can be defined similar to ones used in Examples 9.19 and 9.20 for the MUCOS tasks. CPUs 80x86 and R6000 are examples of the special architectures.
5. *An ISR should not wait for taking the semaphore* or other IPC (an ISR cannot use semTake function). An ISR should not call 'melloc ( )' for memory allocation as that the function uses semaphores. ISR *should not use mutex semaphore.* ISR can use counting semaphore for giving (posting) semaphores.
6. *ISR should just write the required data at the memory* or buffer or post (send or give) an IPC or make a non-blocking write to a message queue (Section 7.12) so that it has short codes and most of its codes, which are non-critical and long time-taking, execute at the tasks.
7. ISR should not use floating-point functions as these take longer times to execute. Let these functions be passed onto the task that runs the codes later.

9.3.4 IPC Functions

Table 9.11 gives a list and description of the interprocess functions. Recall Sections 7.7.1, 7.7.2.1 and 7.7.5. The table gives the functions for semaphore, message queue and pipe.

Signals and software interrupt functions are as follows: Function 'void sigHandler (int *sigNum*);' declares a signal servicing routine for a signal identified by *sigNum* and a signal servicing routine registers a signal as follows: signal (*sigNum, sigISR*). The parameters that pass are *sigNum* (for identifying the signal) and signal-servicing routine name, *sigISR.* Function sigHandler passes *sigNum* as well as an additional code. The *sigHandler.codes* associates with sigHandler. A pointer **pSigCtx* associates with the signal context. The signal context saves CPU registers including PC and SP like an ISR context. The return from sigHandler restores the saved context.

Let *sigISR* be a C function that services the signal interrupt. Let its address be ISR_ADDR. Let the signal be identified by *sigNum.* The function 'intConnect (I_NUM_TO_IVEC (sigNum), sigISR, sigArg)' will connect the signal interrupt service routine (sigISR) for the signal identified by sigNum to the ISR_ADDR. I_NUM_TO_IVEC (*sigNum*) is a function that uses the argument *sigNum* to find the program counter (PC) assignment from the interrupt vector and uses it for ISR_ADDR. The argument *sigArg* passes for use by the C function.

The sigISR may call the following functions.

1. Call 'taskRestart ( )' to restart the task, which generated the *sigNum.* Restarting assigns the original context on creation. Original PC, SP, arguments and options to a task restore now.
2. Call 'exit ( )' to terminate the task, which generated the sigNum.
3. Call 'longjump ( )'. This results in starting the execution from a memory location. The location is the one that was saved when function setjump ( ) was called.

VxWorks provides three kinds of semaphores, binary (flag), mutex and counting. Mutex semaphore also takes care of the priority inversion problem on selecting an OPTION when creating it. Use of a binary semaphore provides an advantage over disabling of interrupts is that it limits (blocks) the use of the associated resources needed in sections within which that semaphore is required to be taken only.

A queue is used when the messages are put in queue for one or more waiting tasks. The VxWorks queue functions are in a library, msgQLib, which the user includes before using those. For full duplex communication between the two tasks, we should create two queues, one for each task. The mqPxLib functions are compatible with POSIX 1003.1b. A detailed description of the three types of VxWorks semaphores and message queues is given next.