UNIT-II

PART A

INTRODUCTION TO REAL-TIME OPERATING SYSTEMS

Contents:

- ✓ Tasks and task states,
- ✓ Tasks and data,
- ✓ Semaphores and shared data,
- ✓ Message queues,
- ✓ Mailboxes and pipes,
- ✓ Timer functions,
- ✓ Events,
- ✓ Memory management,
- ✓ Interrupt routines in an RTOS environment.

INTRODUCTION:

Real Time Operating System (RTOS):

- A real-time operating system (RTOS) is an operating system (OS) intended to serve real-time application process data as it comes in, typically without buffering delays. Processing time requirements (including any OS delay) are measured in tenths of seconds or shorter.
- The Real Time Operating System consists time constraints to complete particular task also called as Deadlines.

Types of RTOS:

- Hard Real Time RTOS
- Soft Real Time RTOS

1. Hard Real Time RTOS:

Hard Real Time RTOS is an RTOS that meets a *deadline* within given time limit and the amount of time it takes to complete task is deterministic.

2. Soft Real Time RTOS:

Soft Real Time RTOS is an RTOS that may or may not meet a *deadline* within given time limit and the amount of time it takes to complete task is not deterministic.

- The most common RTOS designs are:
  - Event-driven which switches tasks only when an event of higher priority needs servicing, called pre-emptive priority, or priority scheduling.
  - Time-sharing designs switch tasks on a regular clocked interrupt, and on events, called round robin.

I. TASKS AND TASK STATES:

Task:

- The basic building block of software written under an RTOS is the **task**
- Each task has its own **registers**, **stack** and **program counter**
- All data is shared among the tasks
- Under most RTOSs a task is simply a subroutine

Task States:

- In most RTOS Designs, a task will be in any one of the following states:
  - Running
  - Ready
  - Blocked
- Most RTOSs may offer many other task states Such as **suspended, pended, waiting, dormant**, and **delayed**.

1. ***Running*** **-** This means that the microprocessor is executing the instructions of a task. Unless yours is a multiprocessor system, there is only one microprocessor, and hence only one task will be in the running state at any given time.

2. ***Ready*** **-** This means that some other task is in the running state but that this task has things that it could do if the microprocessor becomes available. Any number of tasks can be in this state.

3. ***Blocked*** **-**This means that this task hasn't got anything to do right now, even if the microprocessor becomes available. Tasks get into this state because they are waiting for some external event. For example, a task that handles data coming in from a network will have nothing to do when there is no data.

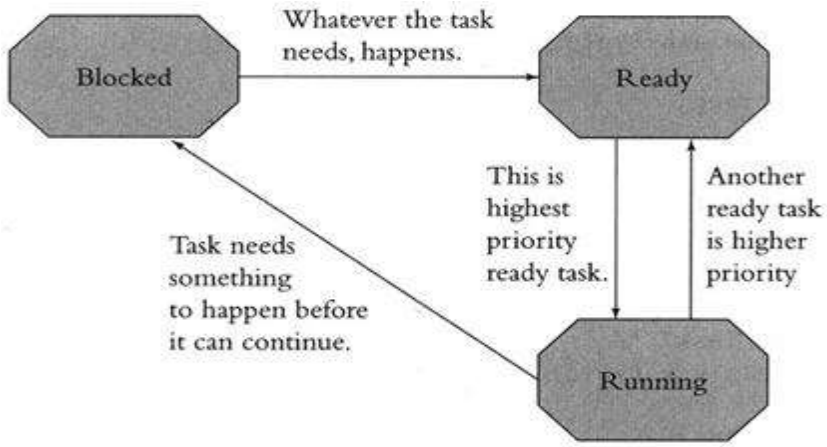


Figure 1: Task States

The Scheduler:

- A part of the RTOS called the **scheduler keeps track of the state of each task** and decides which one task should go into the running state.
- **Very simple behaviour**: schedulers look at priorities you assign to the tasks, and among the tasks that are not in the blocked state, the one with the highest priority runs, and the rest of them wait in the ready state.
- The lower-priority tasks just have to wait; the scheduler assumes that you knew what you were doing when you set the task priorities.

Explanation for Task States figure 1:

- A task will only block because it decides for itself that it has run out of things to do. Other tasks in the system or the scheduler cannot decide for a task that it needs to wait for something. As a consequence of this, a task has to be running just before it is blocked: it has to execute the instructions that figure out that there's nothing more to do.
- While a task is blocked, it never gets the microprocessor. Therefore, an interrupt routine or some other task in the system must be able to signal that whatever the task was waiting for has happened. Otherwise, the task will be blocked forever.
- The shuffling of tasks between the ready and running states is entirely the work of the scheduler. Tasks can block themselves and tasks and interrupt routines can move other tasks from the blocked state to the ready state, but the scheduler has control over the running state. (Of course, if a task is moved from the blocked to the ready state and has higher priority than the task that is running, the scheduler will move it to the running state immediately.)

Preemptive Vs Nonpreemptive Scheduling

The Scheduling algorithms can be divided into two categories with respect to how they deal with clock interrupts.

Nonpreemptive Scheduling

- A scheduling discipline is nonpreemptive if, once a process has been given the CPU, the CPU cannot be taken away from that process.
- Following are some characteristics of nonpreemptive scheduling
  - In nonpreemptive system, short jobs are made to wait by longer jobs but the overall treatment of all processes is fair.
  - In nonpreemptive system, response times are more predictable because incoming high priority jobs can not displace waiting jobs.

- In nonpreemptive scheduling, a schedular executes jobs in the following two situations.
  - When a process switches from running state to the waiting state.
  - When a process terminates.

Preemptive Scheduling

- A scheduling discipline is preemptive if, once a process has been given the CPU can taken away.
- The strategy of allowing processes that are logically runable to be temporarily suspended is called Preemptive Scheduling and it is contrast to the "run to completion" method.

Here are answers to some common questions about the scheduler and task states:

- How does the scheduler know when a task has become blocked or unblocked?

The RTOS provides a collection of functions that tasks can call to tell the scheduler what events they want to wait for and to signal that events have happened.

- What happens if all the tasks are blocked?

If all the tasks are blocked, then the scheduler will spin in some tight loop somewhere inside of the RTOS, waiting for something to happen. If nothing ever happens, then that is your fault. You must make sure that something happens sooner or later by having an interrupt routine that calls some RTOS function that unblocks a task. Otherwise, your software will not be doing very much.

- ***What two tasks with the same priority are ready?***

Ans: The answer to this is all over the map, depending upon which RTOS you use. At least one system solves this problem by making it illegal to have two tasks with the same priority. Some other RTOSs will time slice between two such tasks. Some will run on of them until it blocks and then run the other. In this case, which of the two tasks it runs also depends upon the particular RTOS.

Example Code of underground tank monitoring system for understanding task states:

Explanation:

This pseudo-code is from underground tank monitoring system.

- Here we have two tasks vLevelsTask having low priority, vButtonTask having high priority .The vLevelsTask task uses for computing how much gasoline is in the tanks.
- If user pushes the button, then vButtonTask task unblocks, The RTOS will stop low priority vLevelsTask task in its tracks, move it to the ready state, and run the high priority vButtonTask task.

- When vButtonTask task is finished responding, it blocks and RTOS gives the microprocessor back to the vLevelsTask task once again. This complete processes shown in the below figure 2.

```
/* "Button Task" */
void vButtonTask (void)   /* High priority */
{
   while (TRUE)
   {
      !! Block until user pushes a button
      !! Quick: respond to the user
   }
}

/* "Levels Task" */
void vLevelsTask (void)   /* Low priority */
{
   while (TRUE)
   {
      !! Read levels of floats in tank
      !! Calculate average float level


      !! Do some interminable calculation
      !! Do more interminable calculation
      !! Do yet more interminable calculation

      !! Figure out which tank to do next
   }
}
```

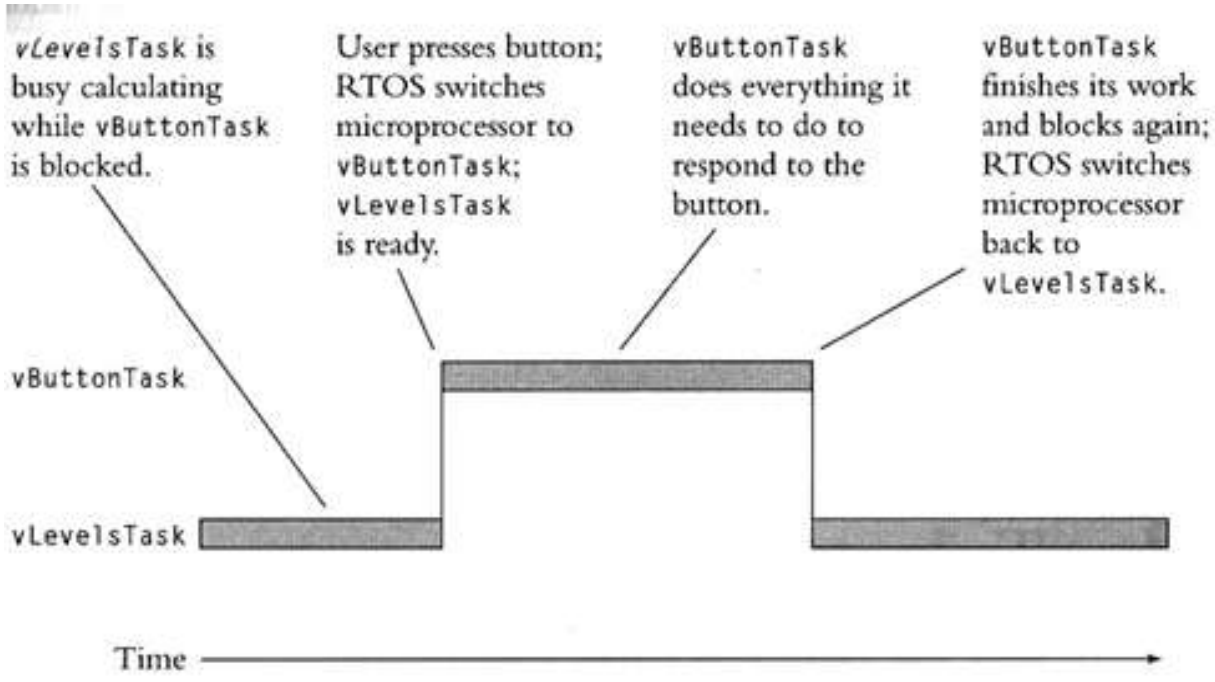


Figure 2: Microprocessor responding to a button under an RTOS

II. TASKS AND DATA:

- Each task has its own private context, which includes the **register values**, a **program counter**, and a **stack**.
- However, all other data - g1obal, static, initialized, uninitialized, and everything else - is shared among all of the tasks in the system.
- The RTOS typically has its **own private data structures**, which are not available to any of the tasks.
- Easy to move data from one task to another

Sharing data variables between different tasks:

- Since you can share data variables among tasks, it is easy to move data from one task to another: the two tasks need only have access to the same variables. You can easily accomplish this by having the two tasks in the same module in which the variables are declared or you can make the variables public in one of the tasks and declare them extern in the other

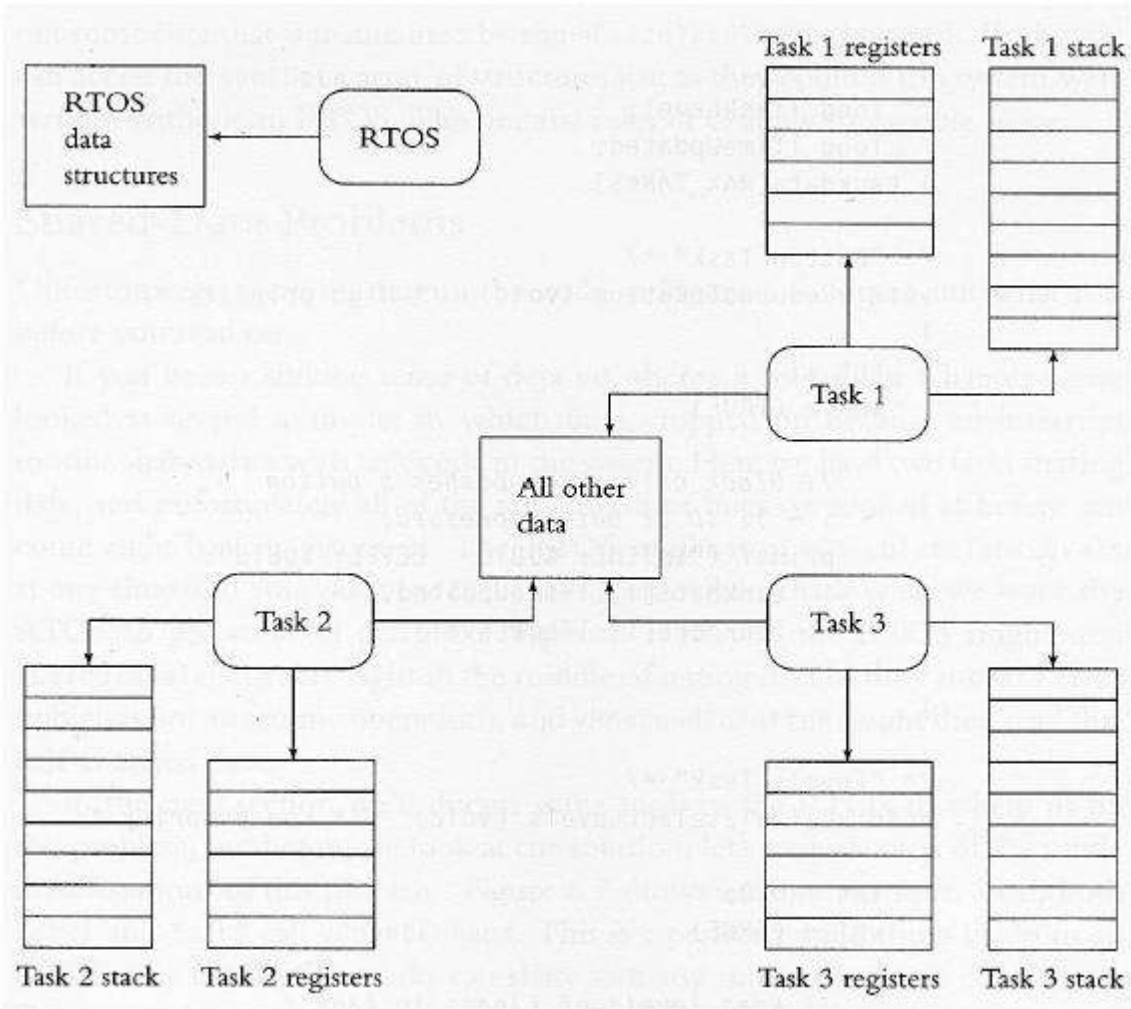


Figure 3: sharing Task data in RTOS

Shared data problem:

- Assume that at an instant when the value of variable operates and during the operations on it, only a part of the operation is completed and another part remains incomplete.
- At that moment, assume that there is an interrupt.
- Assume that there is another function. It also shares the same variable. The value of the variable may differ from the one expected if the earlier operation had been completed.
- For all above reasons, a data inconsistency occurs.

Shared data problems arise due to the following reasons:

- Inconsistencies in data used by a task and updated by an ISR; arises because ISR runs at just the wrong time.
- Arises when task code accesses shared data non-atomically.
- If we have two tasks sharing the same data, it could happen that one of these tasks will read the half-changed data

Example for demonstrating shared data problems:

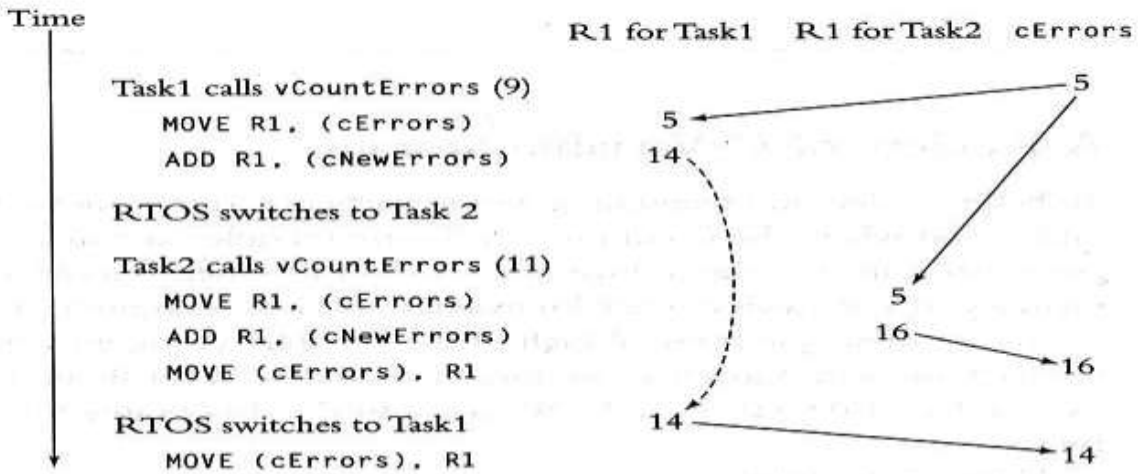
```
void Task1 (void)
{
    :
    :
    vCountErrors (9);
    :
    :
}

void Task2 (void)
{
    :
    :
    vCountErrors (11);
    :
    :
}

static int cErrors;

void vCountErrors (int cNewErrors)
{
   cErrors += cNewErrors;
}
```

- For example in the above code Task1 and Task2 call the *vCountErrors*, and *vCountErrors* uses variable *cErrors*, the variable *cErrors* is now shared by two tasks.
- If Task1 calls *vCountErrors*, and if an interrupt will occur to Task1, then RTOS stops the Task1 and runs Task2, which then calls *vCountErrors*, the variable *cErrors* may get corrupted.

Example with assembly language code

Figure 6.8 Why the Code in Figure 6.7 Fails

```
; Assembly code for vCountErrors
:
; void vCountErrors (int cNewErrors)
;{
;       cErrors += cNewErrors;
        MOVE R1, (cErrors)
        ADD R1, (cNewErrors)
        Move (cErrors), R1
        RETURN
;}
```



- This is the assembly language code for *vCountErrors*. In the above code suppose the value 5 is stored in *cErrors*. From the code1, suppose that Task1 calls *vCountErrors(9)*, and after executing three lines code i.e. up to ADD instruction the result of addition will be in R1 register.
- Now suppose that the RTOS stops the execution of Task1 and runs Task2 and that Task2 calls *vCountErrors (11)*.
- The code in *vCountErrors* fetches the old value of *cErrors*, adds 11 to it, and stores the result.
- Eventually the RTOS switches back to Task1, which then executes the next instruction in *vCountErrors* i.e. MOV *cErrors*, R1 and overwriting the value written by Task2.insted of *cErrors* ending up to 25.it ends up as 14.

REENTRANCY:

- Reentrant functions are functions that can be called by more than one task and that will always work correctly even if the RTOS switches from one task to another in the middle of executing the function.
- You apply three rules to decide if a function is reentrant:
    - A reentrant function may not use variables in a non-atomic way unless they are stored on the stack of the task that called. The function or are otherwise the private variables of that task.

- A reentrant function may not call any other functions that are not themselves reentrant.
- A reentrant function may not use the hardware in a non atomic way.

A Reveiw of C VARIABLE STORAGE:

Reentrancy can be understood better by observing the following C Variable declarations functioning:

Figure 6.9 Variable Storage

```
static int static_int;
int public_int;
int initialized = 4;
char *string = "Where does this string go?";
void *vPointer;

void function (int parm, int *parm_ptr)
{
   static int static_local;
   int local;
   .
   .
   .
}
```

- Which of the varialbles are stored in the stack?
- Which in a fixed location in memory?
- What about the string literal “where does this string go?”
- What about the data pointed to by vPointer and By Parm_Ptr?

static_int **-** is in a fixed location in memory and is therefore shared by any task that happens to call function.

public_int **-** Ditto. The only difference between static_int and public_int is that functions in other C files can access public_int, but they cannot access static_int.(This means, of course, that it is even harder to be sure that this variable is not used by multiple tasks, since it might be used by any function in any module anywhere in the-system.)

Initialized - The same. The initial value makes no difference to where the variable is stored.

String **-** The same. *"Where does this string go?*" - Also the same.

vPointer **-** The pointer itself is in a fixed location in memory and is therefore a shared variable. If function uses or changes the data values pointed to by vPointer, then those data values are also shared among any tasks that happen to call function.

parm **-** is on the stack. If more than one task calls function, parm will be in a different location for each, because each task has its own stack. No matter how many tasks call function, the variable parm will not be a problem.

parm_ptr **-** is on the stack. Therefore, function can do anything to the value of parm_ptr without causing trouble. However, if function uses or changes the values of whatever is pointed to by parm_ptr, then we have to ask where that data is stored before we know whether we have a problem.

static_local - is in a fixed location in memory. The only difference between this and static_int is that static_int can be used by other functions in the same C file, whereas static_local can only be used by function.

Local **-** is on the stack.

Applying the Re-entrancy rules:

Examine the function "display" in below program and decide if it is re-entrant and why it is or is not.

Figure 6.10 Another Reentrancy Example

```
BOOL fError;    /* Someone else sets this */

void display (int j)
{
   if (!fError)
   {
      printf ("\nValue: %d", j);
      j = 0;
      fError = TRUE;
   }
   else
   {
      printf ("\nCould not display value");
      fError = FALSE;
   }
}
```

- The function is not reentrant for two reasons

Rule1 (Violates)

- The variable *fError* is in a fixed location in memory and is therefore shared by any task that calls display.
- The use of fError is not atomic, because the RTOS might switch tasks between the time that it is tested and the time that it is set.

Rule2 (Violates)

- For this function to be reentrant, *printf* must also be reentrant.

Gray Areas of Reentrancy:

- *Reentrancy rule #1:* A reentrant function may not use variables in a non-atomic way (unless they are stored on the stack of the task that called the function or are otherwise the private variables of that task.

```
static int cErrors;
void vCountErrors (void)
{
++cErrors;
}
```

- There are some gray areas between reentrant and non-reentrant functions.
- The code here shows a very simple function in the gray area.
- This function obviously modifies a non-stack variable, but rule 1 says that a reentrant function may not use non-stack variables in a non atomic way. The question is: is incrementing cErrors atomic?
- **cErrors is NOT atomic** if we write an assembly language program by **using 8051 instructions** for above piece of code as it is going to **take 9 lines**. In that 9 lines, chance of **occurrence of interrupt** is more.
- **cErrors is atomic if you use 8086 instruction** set to write program for above piece of code as it takes only 2 lines code like below:

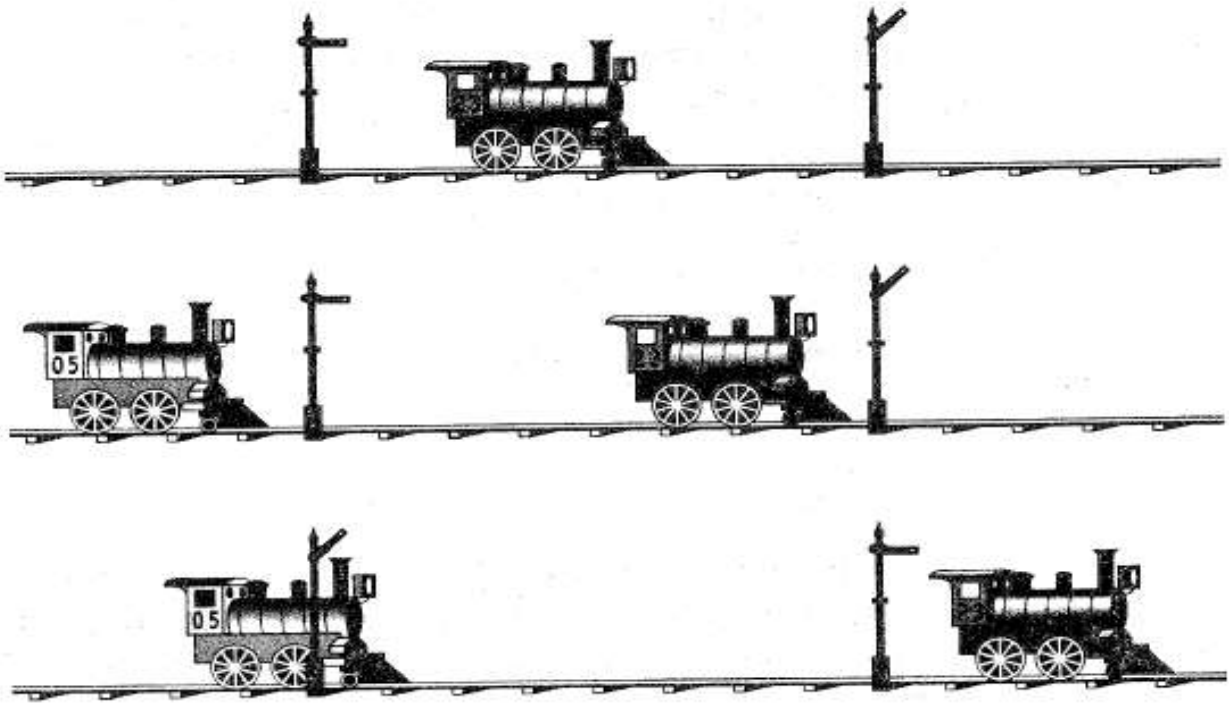
```
INC (cErrors)
RET
```

III. SEMAPHORES AND SHARED DATA:

Semaphores:

- In RTOS shared data problems will occur by switching the microprocessor from one task to another task and like interrupts, changing the flow of execution.

- The RTOS gives some tools to solve shared data problems.
- Semaphores are one such tool.
- Semaphore is a variable/flag/lock used to control access to shared resources.



- When the first train enters the pictured section of track, the semaphore behind it automatically **lowers.**
- When a second train arrives, the engineer notes the **lowered semaphore** and he stops his train and waits for the semaphore to rise.
- When the first train leaves that section of track, the **semaphore rises**, and the engineer on the second train knows that it is safe to proceed on.
- The general idea of a semaphore in an RTOS is similar to the idea of a railroad semaphore.
- No RTOS use the terms **raise** and **lower;** they use **get** and **give, take** and **release, pend** and **post, p** and **v, wait** and **signal.**

Binary Semaphore

- A typical RTOS binary semaphore work like this: tasks can call two RTOS functions, **TakeSemaphore** and **ReleaseSemaphore.**
- If one task has called TakeSemaphore to take the semaphore and has not called ReleaseSemaphore to release it, then any other task that calls TakeSemaphore will block until first task calls ReleaseSemaphore.
- Only one task can have the semaphore at a time.

- A typical RTOS binary semaphore work like this: tasks can call two RTOS functions, **TakeSemaphore** and **ReleaseSemaphore.**
- If one task has called TakeSemaphore to take the semaphore and has not called ReleaseSemaphore to release it, then any other task that calls TakeSemaphore will block until first task calls ReleaseSemaphore.
- Only one task can have the semaphore at a time.

Example: Showing how the semaphores protect shared data

Figure 6.12 Semaphores Protect Data

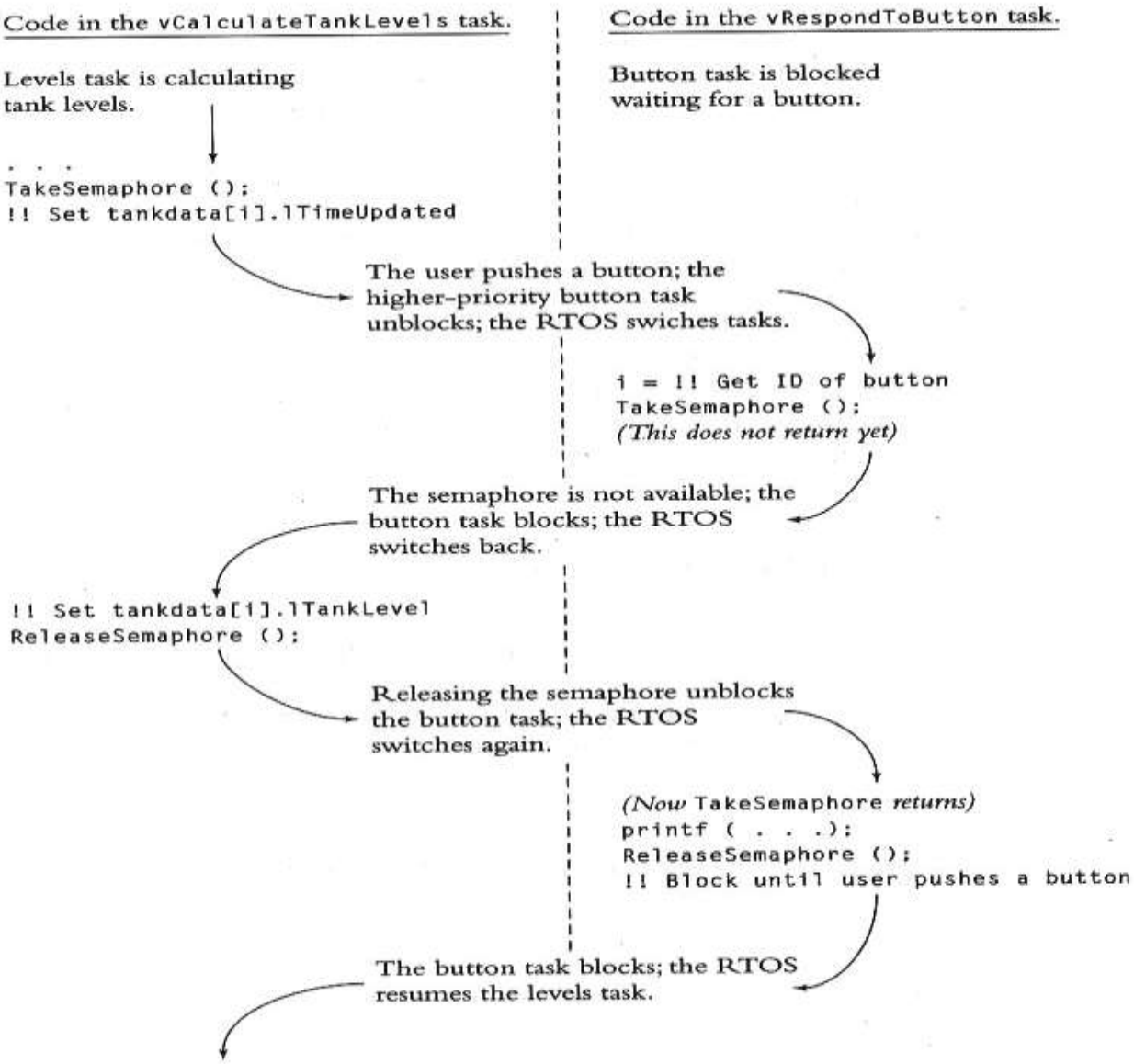
```
struct
{
    long lTankLevel;
    long lTimeUpdated;
} tankdata[MAX_TANKS];

/* "Button Task" */
void vRespondToButton (void)  /* High priority */
{
    int i;
    while (TRUE)
    {
        !! Block until user pushes a button
        i = !! Get ID of button pressed
        TakeSemaphore ();
        printf ("\nTIME: %08ld   LEVEL: %08ld",
            tankdata[i].lTimeUpdated,
            tankdata[i].lTankLevel);
        ReleaseSemaphore ();
    }
}
```

(continued)

Figure 6.12 *(continued)*

```
/* "Levels Task" */
void vCalculateTankLevels (void)   /* Low priority */
{
    int i = 0;
    while (TRUE)
    {
        .
        .
        .
        TakeSemaphore ();
        !! Set tankdata[i].lTimeUpdated
        !! Set tankdata[i].lTankLevel
        ReleaseSemaphore ();
        .
        .
        .
    }
}
```

Figure 6.13 Execution Flow with Semaphores



- Before the "levels task" (vCalculateTankLevels) updates the data in the structure, it calls Take Semaphore to take (lower) the semaphore.
- If the user presses a button while the levels task is still modifying the data and still has the semaphore, then the following sequence of events occurs:
  1. The RTOS will switch to the button task, just as before moving the levels task to ready state.
  2. When the button task tries to get semaphore by calling TakeSemaphore, it will block because the levels task already has the semaphore.
  3. The RTOS will then look around for another task to run and will notice that levels task is still ready. With the button task blocked, the levels task will get to run until it releases the semaphore.
  4. When the levels task release the semaphore by calling ReleaseSemaphore, the button task no longer be blocked and the RTOS will switch back to it.

Reentrancy and Semaphores:

- The shared function vCountErrors which in pervious example was not reentrant.
- In above figure the code that modifies the static variable cErrors is surrounded by calls to semaphore routines.
- Whichever task calls vCountErrors second will be blocked when it tries to take the semaphore.
- In the language of reentrancy we have made the use of cErrors atomic and therefore have made function vCountErrors reentrant.
- The functions and data structures whose names begin with NU are those used in an RTOS called Nucleus.

Figure 6.15 Semaphores Make a Function Reentrant

```
void Task1 (void)
{
    :
    :
    vCountErrors (9);
    :
    :
}

void Task2 (void)
{
    :
    :
    vCountErrors (11);
    :
    :
}

static int cErrors;
static NU_SEMAPHORE semErrors;

void vCountErrors (int cNewErrors)
{
    NU_Obtain_Semaphore (&semErrors, NU_SUSPEND);
    cErrors += cNewErrors;
    NU_Release_Semaphore (&semErrors);
}
```

Multiple Semaphores:

- What happens if we have multiple data items that need to be protected?
- Using a single semaphore is impractical
- Most RTOSs allow multiple semaphores
- It is up to the programmer to allocate different semaphores to different data items .Since most RTOSs allow you to have as many semaphores as you like, each call to the RTOS must identify the semaphore on which to operate.
- The semaphores are all independent of one another: if one task takes semaphore A, another task can take semaphore B without blocking. Similarly, if one task is waiting for semaphore C, that task will still be blocked even if some other task releases semaphore D.

What is the Advantage of having multiple semaphores?

- Different semaphores can correspond to different shared resources.
- Multiple semaphores will increase the efficiency of resources.

How does the RTOS know which semaphore protects which data?

- It doesn't. If you are using multiple semaphores, it is up to you to remember which semaphore corresponds to which data.
- A task that is modifying the error count must take the corresponding semaphore. You must decide what shared data each of your semaphore protects.

Semaphores as a Signaling Device:

- Another common use of semaphore is as a simple way to communicate from one task to another or from an interrupt routine to task.

Semaphore Problems:

- *Forgetting to take the semaphore.*
- *Forgetting to release the semaphore.*
- *Taking the wrong semaphore.*
- *Holding a semaphore for too long.*
- *Causing a deadly embrace.*

Forgetting to take the semaphore.

Semaphores only work if every task that accesses the shared data, for read or for write, uses the semaphore. If anybody forgets, then the RTOS may switch away from the code that forgot to take the semaphore and cause an ugly shared-data bug.

Forgetting to release the semaphore.

If any task fails to release the semaphore, then every other task that ever uses the semaphore will sooner or later block waiting to take that semaphore and will be blocked forever.

Taking the wrong semaphore.

If you are using multiple semaphores, then taking the wrong one is as bad as forgetting to take one.

Holding a semaphore for too long.

Whenever one task takes a semaphore, every other task that subsequently wants that semaphore has to wait until the semaphore is released. If one task takes the semaphore and then holds it for too long, other tasks may miss real - time deadlines.

Priority Inversion Problem:

- In **priority inversion** a high priority task waits because a low priority task has a semaphore, but the lower priority task is not given CPU time to finish its work. A typical solution is to have the task that owns a semaphore run at, or 'inherit,' the priority of the highest waiting task.
- But this simple approach fails when there are multiple levels of waiting: task *A*waits for a binary semaphore locked by task *B*, which waits for a binary semaphore locked by task *C*. Handling multiple levels of inheritance without introducing instability in cycles is complex and problematic.
- Because of low priority task, if any high priority task goes to the blocked state and misses its real-time deadline, this situation is called a priority inversion problem.

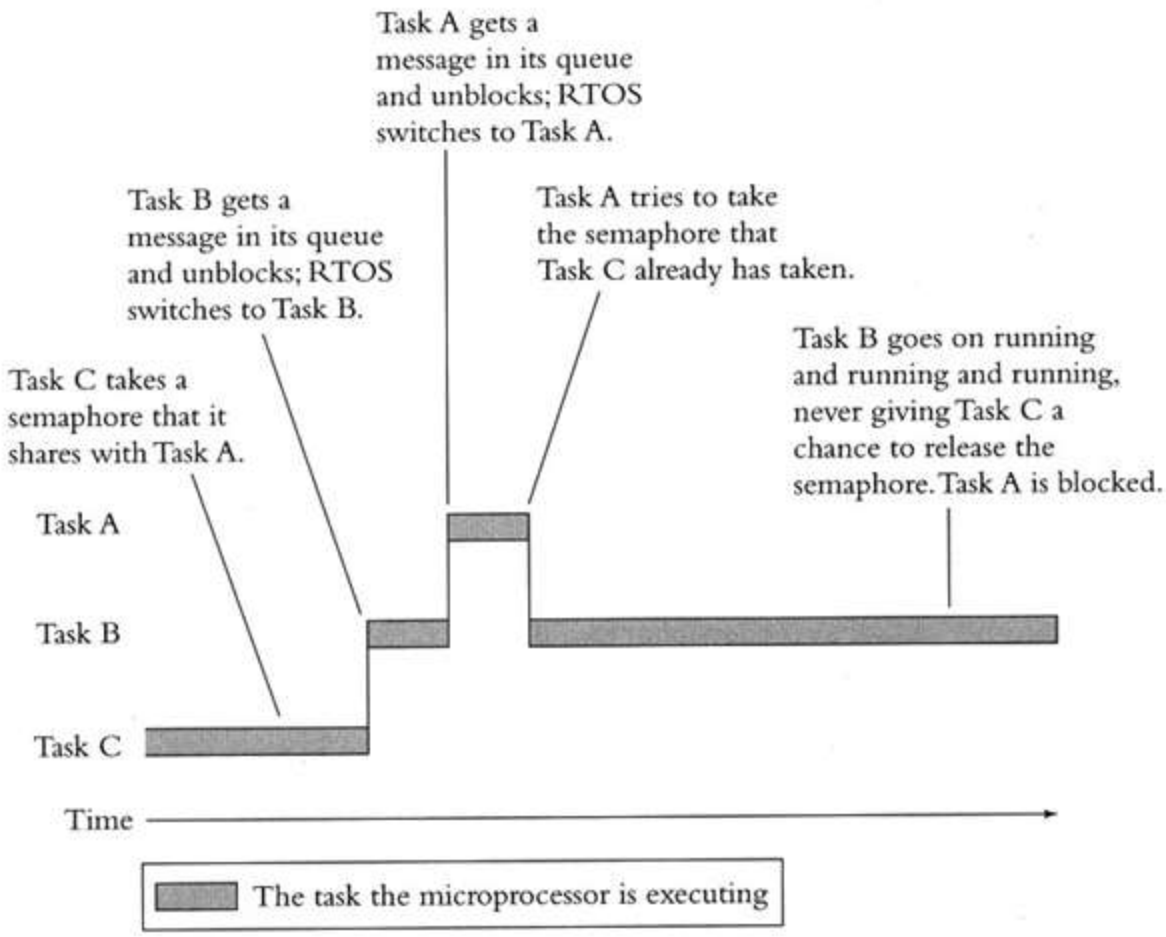


Figure 4: priority inversion

Causing a deadly embrace or dead lock:

- The below program in figure 6.18 illustrate the problem called deadly embrace.
- The function *ajsmrsv* and *ajsmrls* in that figure are from an RTOS called AMX.
- The function *ajsmrsv* "reserve" a semaphore and the function *ajsmrls* "release" that semaphore.
- The two additional parameters to *ajsmrsv* are time-out and priority information.

- In the above code both Task1 and Task2 operate on variables *a* and *b* after getting permission to use them by getting semaphores *hSemaphoreA* and *hSemaphoreB*.
- Consider what happens if vTask1 calls *ajsmrsv* to get *hSemaphoreA*, but before it call *ajsmrsv* to get *hSemaphoreB*, the RTOS stops it and runs vTask2.
- The vTask2 now calls *ajsmrsv* and gets *hSemaphoreB*. When vTask2 calls *ajsmrsv* to get *hSemaphoreA*, it blocks, because another task (vTask1) already has that semaphore.
- The RTOS will now switch back to vTask1, which now calls *ajsmrsv* to get *hSemaphoreB*. Since vTask2 has *hSemaphoreB*, however vTask1 now also blocks.
- There is no escape from this for either task, since both are now blocked waiting for the semaphore that the other has.

Figure 6.18 Deadly-Embrace Example

```
int a;
int b;
AMXID hSemaphoreA;
AMXID hSemaphoreB;
void vTask1 (void)
{
    ajsmrsv (hSemaphoreA, 0, 0);
    ajsmrsv (hSemaphoreB, 0, 0);
    a = b;
    ajsmrls (hSemaphoreB);
    ajsmrls (hSemaphoreA);
}

void vTask2 (void)
{
    ajsmrsv (hSemaphoreB, 0, 0);
    ajsmrsv (hSemaphoreA, 0, 0);
    b = a;
    ajsmrls (hSemaphoreA);
    ajsmrls (hSemaphoreB);
}
```

Types or Variants of semaphores:

- There are a number of different kinds of semaphores. Here is an overview of some of the more common variations:
    - Counting semaphores
    - Resource semaphores
    - Mutex semaphore

Counting semaphores:

- Some systems offer semaphores that can be taken multiple times. Essentially, such semaphores are integers; taking them decrements the integer and releasing them increments the integer. If a task tries to take the semaphore when the integer is equal to zero, then the task will block. These semaphores are called counting semaphores, and they were the original type of semaphore.

Resource semaphores:

- These semaphores are useful for the shared - data problem, but they cannot be used to communicate between two tasks. Such semaphores are sometimes called resource semaphores or resources.

Mutex semaphore:

- Some RTOSs offer one kind of semaphore that will automatically deal with the priority inversion problem and another that will not. The former kind of semaphore commonly called a mutex semaphore or mutex.

Ways to Protect Shared Data:

- Three methods of protecting shared data:
  - Disabling interrupts
  - Taking semaphores
  - Disabling task switches

1. Disabling interrupts: It is the most drastic in that it will affect the response times of all the interrupt routines and of all other tasks in the system. (If you disable interrupts, you also disable task switches, because the scheduler cannot get control of the microprocessor to switch.) On the other hand, disabling interrupts has two advantages.

(I) It is the only method that works if your data is shared between your task code and your interrupt routines. Interrupt routines are not allowed to take semaphores, and disabling task switches does not prevent interrupts.

(II) It is fast. Most processors can disable or enable interrupts with a single instruction.

2. Taking semaphores: it is the most targeted way to protect data, because it affects only those tasks that need to take the same semaphore. The response times of interrupt routines and of tasks that do not need the semaphore are unchanged.

3. Disabling task switches: it is somewhere in between the two. It has no effect on interrupt routines, but it stops response for all other tasks cold.

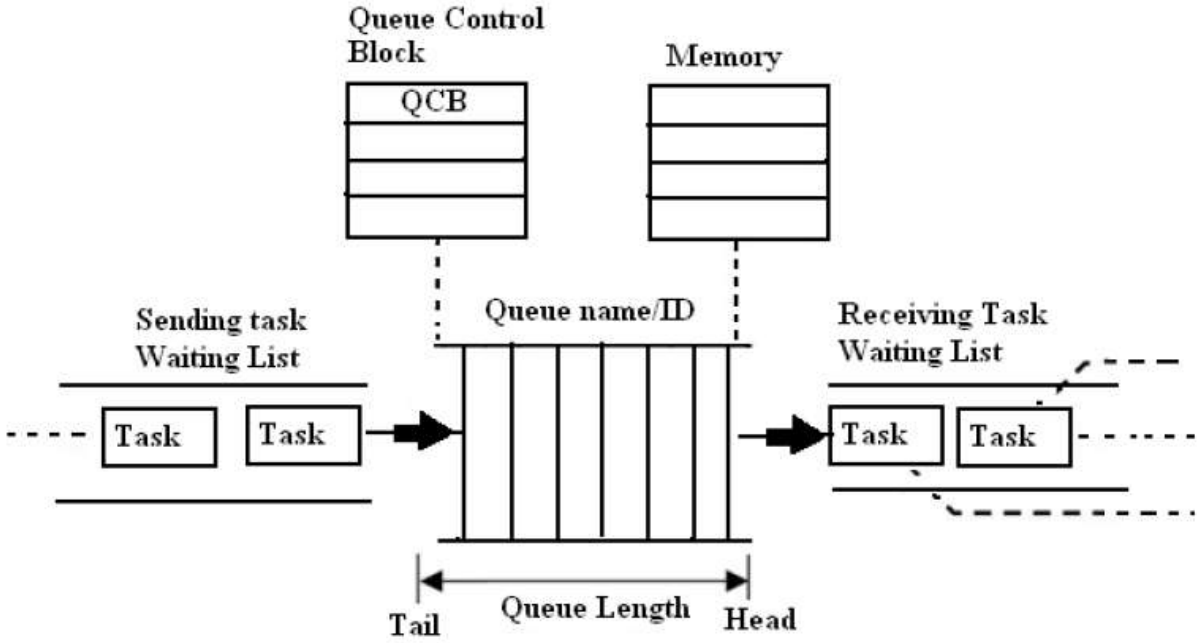
MORE OPERATING SYSTEM SERVICES or INTER TASK COMMUNICATION METHODS:

- Tasks must be able to communicate with one another to coordinate their activities or to share data.
- Shared data and semaphores allow tasks to communicate with one another. The RTOS offers several methods to provide the communication between tasks, those are:
  - Message queues
  - Mail boxes
  - Pipes

IV: MESSAGE QUEUES:

Here is a very simple example.

- Suppose that we have two tasks, Task1 and Task2, each of which has a number of high-priority, urgent things to do. Suppose also that from time to time these two tasks discover error conditions that must be reported on a network, a time consuming process.
- In order not to delay Task1 and Task2, it makes sense to have a separate task, Errors Task that is responsible for reporting the error conditions on the network.
- Whenever Task1 or Task2 discovers an error, it reports that error to ErrorsTask and then goes on about its own business.
- The error reporting process undertaken by ErrorsTask does not delay the other tasks. An RTOS queue is the way to implement this design.
- The below figure gives the brief idea about how the queues are implemented in the RTOS.



Simple use of a queue

```
//RTOS queue function prototypes
void AddToQueue   (int iData);
void ReadFromQueue   (int *p,iData);

//global variables
static int cErrors;

void Task1 (void)
{
   (...)
   if   (!! problem arises)
     vLogError (ERROR_TYPE_X);
   !! Do other things
   (...)
}

void Task2 (void)
   (...)
   if   (!! problem arises)
     vLogError (ERROR_TYPE_X);
   !! Do other things
   (...)
}
```

Task1 and **Task2**, each has a number of high-priority, urgent things to do.

When Task1 or Task2 needs to log errors, it calls **vLogError**.

Simple use of a queue

The **vLogError** function puts the error on a queue of errors for ErrorsTask to deal with.

AddToQueue function adds the value of the integer parameter it is passed to a queue of integer values the RTOS maintains internally.

```
void  vLogError  (int  iErrorType)
{
  AddToQueue  (iErrorType);
}

//This task is running on background
void  ErrorsTask  (void)
{
  int  iErrorType;
  while   (FOREVER)
  {
    ReadFromQueue (&iErrorType);
    ++cErrors;

    !!Send cErrors out on network
    !!Send iErrorType out on network
  }
}
```

Some Ugly Details:

Queues are not quite simple. Here are some of the complications that you will have to deal with in most RTOSs:

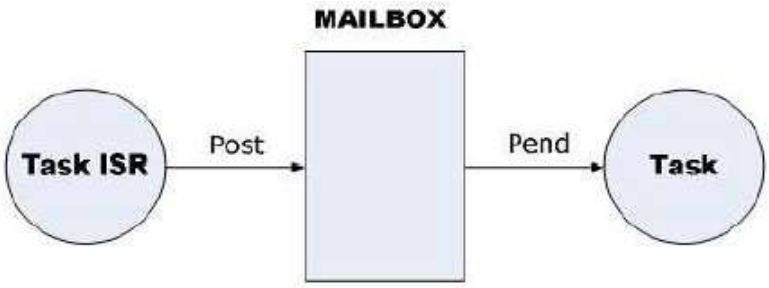
- Most RTOSs require that you initialize your queues before you use them, by calling a function provided for this purpose. On some systems, it is also up to you to allocate the memory that the RTOS will

manage as a queue. As with semaphores, it makes most sense to initialize queues in some code that is guaranteed to run before any task tries to use them.

- Since most RTOSs allow you to have as many queues as you want, you pass an additional parameter to every queue function: the identity of the queue to which you want to write or from which you want to read.
- If your Task tries to write to a queue when the queue is full, the RTOS must either return an error to let you know that the write operation failed (a more common RTOS behavior) or it must block the task until some other task reads data from the queue and thereby creates some space (a less common RTOS behavior).
- Many RTOSs include a function that will read from a queue if there is any data and will return an error code if not. This function is in addition to the one that will block your task if the queue is empty.
- The amount of data that the RTOS lets you write to the queue in one call may not be exactly the amount that you want to write. Many RTOSs are inflexible about this. One common RTOS characteristic is to allow you to write onto a queue in one call the number of bytes taken up by a void pointer.

V. MAILBOXES:

- In general, mailboxes are much like queues.
- The typical RTOS has functions to create, to write to, and to read from mailboxes, and perhaps functions to check whether the mailbox contains any messages and to destroy the mailbox if it is no longer needed.
- The details of mailboxes are different in different RTOSs.
- The below figure is for explaining how the mailboxes work in RTOS.



Here are some of the variations that you might see:

- Although some RTOSs allow a certain number of messages in each mailbox, a number that you can usually choose when you create the mailbox, others allow only one message in a mailbox at a time.

Once one message is written to a mailbox under these systems, the mailbox is full; no other message can be written to the mailbox until the first one is read.

- In some RTOSs, the number of messages in each mailbox is unlimited. There is a limit to the total number of messages that can be in all of the mailboxes in the system, but these messages will be distributed into the individual mailboxes as they are needed.
- In some RTOSs, you can prioritize mailbox messages. Higher-priority messages will be read before lower-priority messages, regardless of the order in which they are written into the mailbox.

For example, in multitask system each message is avoid pointer. you must create all of the mailboxes you need when you configure the system, after which you can use these three functions

Int sndmsg(unsigned int uMbid, void *p_vmsg, unsigned int uPriority);

Void *rcvmsg(unsigned int uMid, unsigned uTimeout);

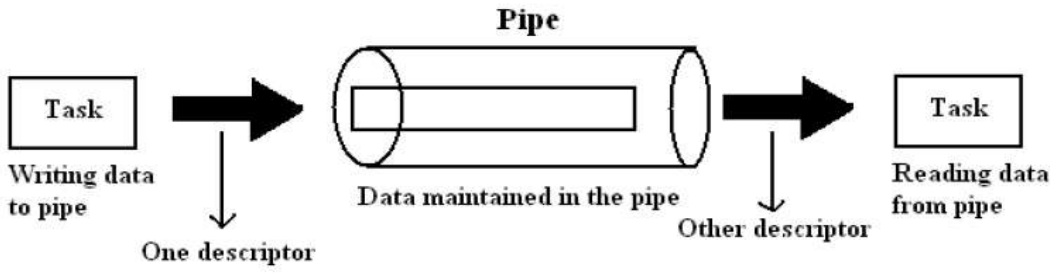
Void *chkmsg(unsigned int uMid)

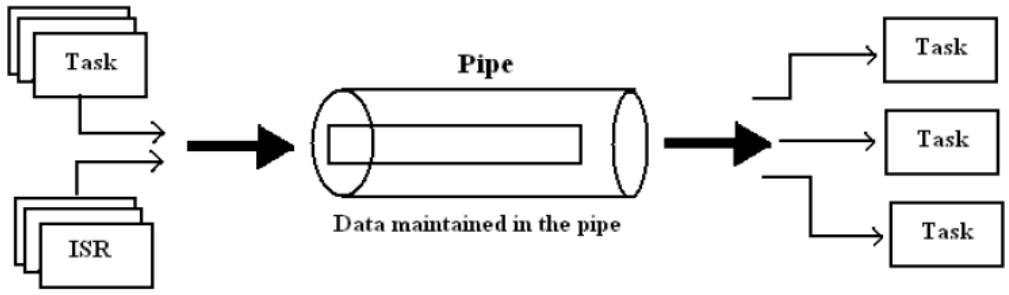
In the above all three functions uMid parameter identifies the mailbox on which to operate.

- The **sndmsg function** adds p_vmsg into the queue of messages held by the uMid mailbox with the priority indicated by uPriority .it returns an error if uMid is invalid or too many messages are already pending in mailboxes.
- The **rcmsg function** returns the highest-priority message from the specified mailbox. It blocks the task that called it if the mailbox is empty. The task use uTimeout parameter to limit how long it will wait if there are no messages.
- The **chkmsg function** returns the first message in the mailbox. It returns a NULL immediately if the mailbox is empty.

V. PIPES:

- Pipes are also much like queues.
- The RTOS can create them, write to them, read from them, and so on.
- The details of pipes, however, like the details of mailboxes and queues, vary from RTOS to RTOS.





Some variations you might see include the following:

- Some RTOSs allow you to write messages of varying lengths onto pipes (unlike mailboxes and queues, in which the message length is typically fixed).
- Pipes in some RTOSs are entirely byte-oriented: if Task A writes 11 bytes to the pipe and then Task B writes 19 bytes to the pipe, then if Task C reads 14 bytes from the pipe, it will get the 11 that Task A wrote plus the first 3 that Task B wrote. The other 16 that task B wrote remain in the pipe for whatever task reads from it next.
- Some RTOSs use the standard C library functions ***fread*** and ***fwrite*** to read from and write to pipes.

Pitfalls in using Queues, Mailboxes and Pipes:

- Most RTOSs do not restrict which tasks can read from or write to any given queue, mailbox, or pipe. Therefore, you must ensure that tasks use the correct one each time. If some task writes temperature data onto a queue read by a task expecting error codes, your system will not work very well. This is obvious, but it is easy to mess up.
- The RTOS cannot ensure that data written onto a queue, mailbox, or pipe will be properly interpreted by the task that reads it. If one task writes an integer onto the queue and another task reads it and then treats it as a pointer, your product will not ship until the problem is found and fixed.
- Running out of space in queues, mailboxes, or pipes is usually a disaster for embedded software. When one task needs to pass data to another, it is usually not optional. For example, if the RTOS fail to report errors if its queue filled, the data will be stored in garbage location which cannot be identified, Good solutions to this problem is to make your queues, mailboxes, and pipes large enough in the first place.
- Passing pointers from one task to another through a queue, mailbox, or pipe is one of several ways to create shared data is not necessary in all cases.

VI. TIMER FUNCTIONS

- Most embedded systems must keep track of the passage of time.

- To extend its battery life, the cordless bar-code scanner must turn itself off after a certain number of seconds. Systems with network connections must wait for acknowledgements to data that they have sent and retransmit the data if an acknowledgement doesn't show up on time.
- One simple service that most RTOSs offer is a function that **delays** a task for a period of time; that is, blocks it until the period of time expires.

Questions

How do I know that the taskDelay function takes a number of milliseconds as its parameter?

- You don't. In fact, it doesn't. The taskDelay function in VxWorks, like the equivalent delay function in most RTOSs, takes the number of system ticks as its parameter. The length of time represented by each system tick is something you can usually control when you set up the system.

How accurate are the delays produced by taskDelay function?

- They are accurate to the nearest system tick. The RTOS works by setting up a single hardware timer to interrupt periodically, say, every millisecond, and bases all timings on that interrupt. This timer is often called the heartbeat timer. For example, if one of your tasks passes 3 to taskDelay, that task will block until the heartbeat timer interrupts three times The first timer interrupt may come almost immediately after the call to taskDelay or it may come after just under one tick time or after any amount of time between those two extremes.(Note that the task will unblock when the delay time expires; when it will run depends as always upon what other, higher-priority tasks are competing for the microprocessor at that time.).This can be seen in the below figure:

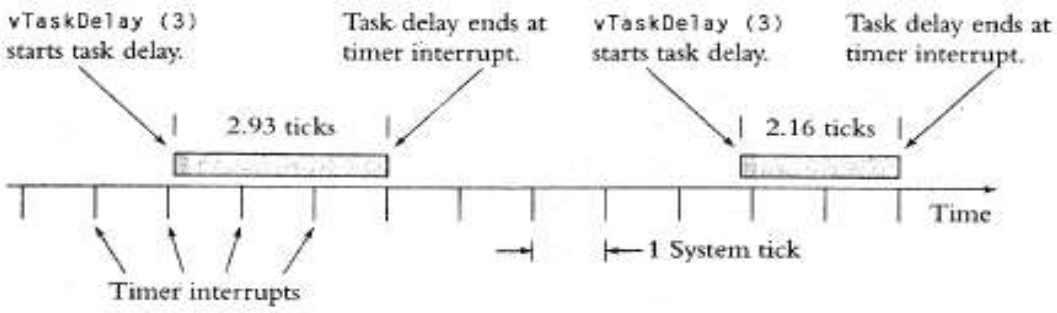


Fig 5: Timer Function Accuracy

How does the RTOS know how to set up the timer hardware on my particular hardware?

- It is common for microprocessors used in embedded systems to have timers in them. Since RTOSs, like other operating systems, are microprocessor-dependent, the engineers writing the RTOS know what kind of microprocessor the RTOS will run on and can therefore program the timer on it.

What is a "normal" length for the system tick?

- There really isn't one. The advantage of a short system tick is that you get accurate timings. The disadvantage is that the microprocessor must execute the timer interrupt routine frequently. a short system tick can decrease system throughput quite considerably by increasing the amount of microprocessor time spent in the timer interrupt routine. Real - time system designers must make this trade-off.

What if my system needs extremely accurate timing?

- You have two choices. One is to make the system tick short enough that RTOS timings fit your definition of "extremely accurate". The second is to use a separate hardware timer for those timings that must be extremely accurate.. The advantage of the RTOS timing functions is that one hardware timer times any number of operations simultaneously.

VII. EVENTS:

- Another service many RTOSs offer is the management of events within the system.
- An event is essentially a Boolean flag that tasks can set or reset and that other tasks can wait for.
- For example, when the user pulls the trigger on the cordless bar-code scanner, the task that turns on the laser scanning mechanism and tries to recognize the bar-code must start.

Some **standard features of events** are listed below:

- More than one task can block waiting for the same event, and the RTOS will unblock all of them (and run them in priority order) when the event occurs. For example, if the radio task needs to start warming up the radio when the user pulls the trigger, then that task can also wait on the trigger-pull event.
- RTOSs typically form groups of events, and tasks can wait for any subset of events within the group.
- Different RTOSs deal in different ways with the issue of resetting an event after it has occurred and tasks that were waiting for it have been unblocked. Some RTOSs reset events automatically; others require that your task software do this. It is important to reset events: if the trigger-pull event is not reset, for example, then tasks that need to wait for that event to be set will never again wait.

A Brief Comparison of the Methods for Inter task Communication:

Methods to provide the communication between two tasks or between an interrupt routine and a task are queues, pipes, mailboxes, semaphores, and events. Here is a comparison of these methods:

- **Semaphores are usually the fastest and simplest methods**. However, not much information can pass through a semaphore, which passes just a 1-bit message saying that it has been released.
- Events are a little more complicated than semaphores and take up just a hair more microprocessor time than semaphores. The advantage **of events over semaphores is that a task can wait for any one of several events at the same time, whereas it can only wait for one semaphore**. (Another advantage is that some RTOSs make it convenient to use events and make it inconvenient to use semaphores for this purpose.)
- **Queues allow you to send a lot of information from one task to another**. Even though the task can wait on only one queue (or mailbox or pipe) at a time, the fact that you can send data through a queue make it even more flexible than events.

VIII. MEMORY MANAGEMENT:

- Most RTOSs have some kind of memory management subsystem.
- Some offer the equivalent of the C library functions ***malloc*** and ***free***, real-time systems engineers often avoid these two functions because they are typically slow and because their execution times are unpredictable.
- They favour instead functions that allocate and free fixed-size buffers, and most RTOSs offer fast and predictable functions for that purpose.
- The Multi Task system is a fairly typical RTOS
- Memory is divided into pools each pool is divided into memory buffers all of the buffers are the same size.
- The ***reqbuf*** and ***getbuf*** functions allocate memory buffer from a pool

  void *getbuf (unsigned int uPoolId, unsigned int uTimeout);

  void *reqbuf (unsigned int uPoolId);

  void relbuf (unsigned int uPoolId, void *p_vBuffer);

  - In each of these functions, the **uPoolId** parameter indicates the pool from which the memory buffer is to be allocated.
  - The **uTimeout** parameter in getbuf indicates the length of time that the task is willing to wait for a buffer if none are free.
  - The size of the buffer that is returned is determined by the pool from which the buffer is allocated, since all the buffers in anyone pool are the same size. The tasks that call, these functions must know the sizes of the buffers in each pool.

- The **relbuf** function frees a memory buffer.

- The Multi Task system is also typical of many RTOSs in that it does not know where the memory on your system is. Remember that in most embedded systems, unlike desktop systems, your software, not the operating system, gets control of a machine first. When it starts, the RTOS has no way of knowing what memory is free and what memory your application is already using.
- Multi Task will manage a pool of memory buffers for you, but you must tell it where the memory is.
- The init_mem_pool function allows you to do this.

Initialization of memory is done by the below function in RTOS:

int init_mem_pool (unsigned int uPoolId, void *p_vMemory, unsigned int uBufSize,

unsigned int uBufCount, unsigned int uPoolType);

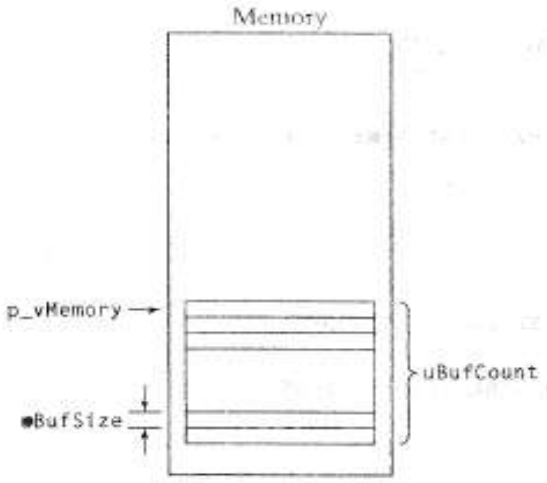


Figure 6 : the init_mem_pool Function in MultiTask

IX. INTERRUPT ROUTINES IN AN RTOS ENVIRONMENT:

- Interrupt routines in most RTOS environments must follow two rules that do not apply to task code.
  - **Rule 1**: ***An interrupt routine must not call any RTOS function that might block the caller*. Therefore, interrupt routines must not get semaphores, read from queues or mailboxes that might be empty, wait for events, and so on.**
    - If an interrupt routine calls an RTOS function and gets blocked, then, in addition to the interrupt routine, the task that was running when the interrupt occurred will be blocked, even if that task is the highest- priority task.
    - Also, most interrupt routines must run to completion to reset the hardware to be ready for the next interrupt.

- **<u>Rule 2:</u> An interrupt routine may not call any RTOS function that might cause the RTOS to switch tasks unless the RTOS knows that an interrupt routine, and not a task, is executing.**
    - This means that interrupt routines may not write to mailboxes or queues on which tasks may be waiting, set events, release semaphores, and so on - unless the RTOS knows it is an interrupt routine that is doing these things.
    - If an interrupt routine breaks this rule, the RTOS might switch control away from the interrupt routine (which. the RTOS think is a task) to run another task, and the interrupt routine may not complete for a long time, blocking at least all lower-priority interrupts and possibly all interrupts.

Rule 2: No RTOS Calls without Fair Warning

- A view of how an interrupt routine should work under an RTOS. The below figure 8 shows how the microprocessor's attention shifted from one part of the code to another over time.
- The interrupt routine interrupts the lower-priority task, and, among other things, calls the RTOS to write a message to a mailbox (legal under rule l, assuming that function can't block). When the interrupt routine exits, the RTOS arranges for the microprocessor to execute either the original task, or, if a higher-priority task was waiting on the mailbox, that higher priority task.

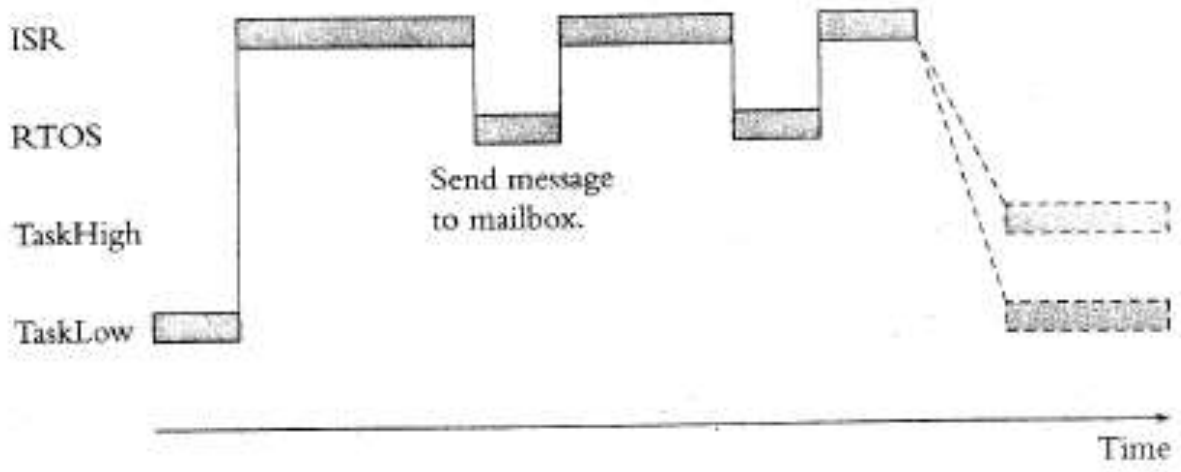


Fig8: How interrupt routine should work

- Figure9 shows what really happens, at least in the worst case.
- If the higher-priority task is blocked on the mailbox, then as soon as the interrupt routine writes to the mailbox, the RTOS unblocks the higher-priority task.
- Then the RTOS (knowing nothing about the interrupt routine) notices that the task that it thinks is running is no highest-priority task that is ready to run.

- Therefore, instead of returning to the interrupt routine (which the RTOS thinks is part of the lower priority task), the RTOS switches to the higher-priority task.
- The interrupt routine doesn't get to finish until later.

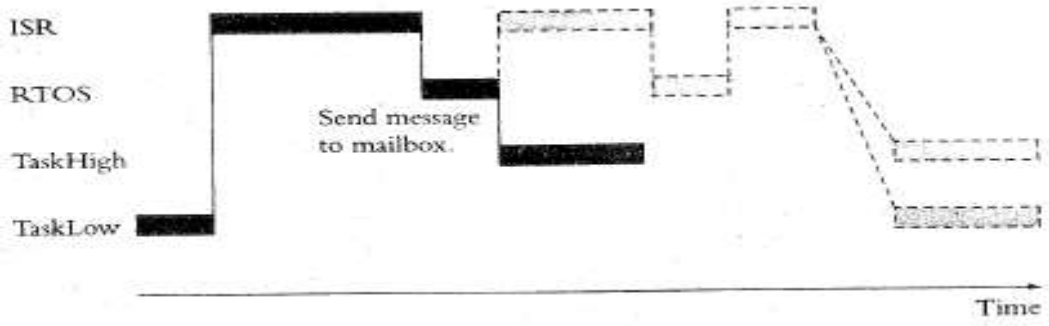


Fig 9: what would really happened

- RTOSs use various methods for solving this problem.
- Figure10 shows the first scheme. In it, the RTOS intercepts all the interrupts and then calls your interrupt routine. By doing this, the RTOS finds out when an interrupt routine has started.
- When the interrupt routine later writes to mailbox, the RTOS knows to return to the interrupt routine and not to switch tasks, no matter what task is unblocked by the write to the mailbox.
- When the interrupt routine is over, it returns, and the RTOS gets control again. The RTOS scheduler then figures out what task should now get the microprocessor.

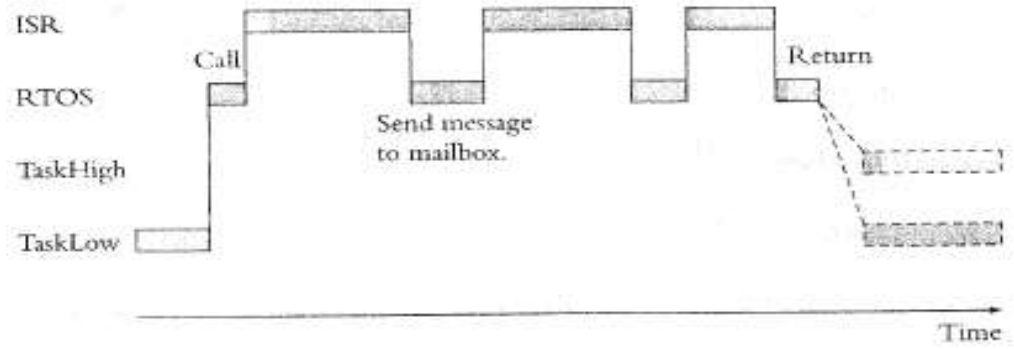


Fig10: How interrupt routine do work: Plan A

- If your RTOS uses this method, then you will need to call some function within the RTOS that tells the RTOS where your interrupt routines are and which hardware interrupts correspond to which interrupt routines.

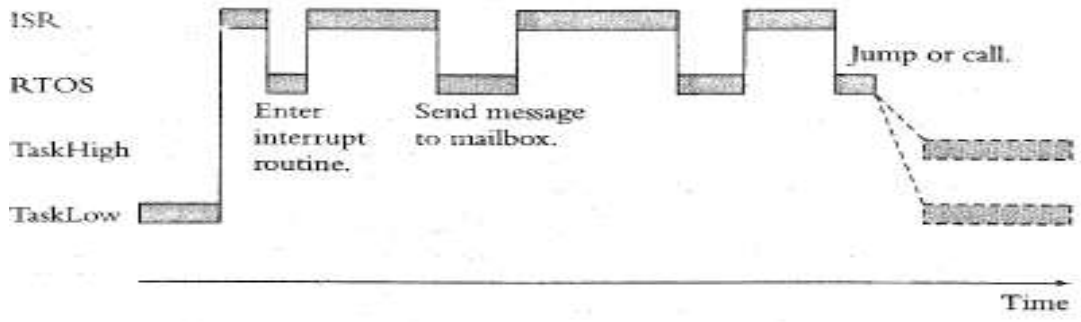


Fig11: How interrupt routines do work: plan B

- Figure11 shows an alternative scheme, in which the RTOS provides a function that the interrupt routines call to let the RTOS know that an interrupt routine is running.
- After the call to that function, the RTOS knows that an interrupt routine is in progress, and when the interrupt routine writes to the mailbox the RTOS always returns to the interrupt routine, no matter what task is ready, as in the figure.
- When the interrupt routine is over, it jumps to or calls some other function in RTOS, which calls the scheduler to figure out what task should now get the microprocessor.
- Essentially, this procedure disables the scheduler for the duration of the interrupt routine.

Rule 2 and Nested Interrupts:

- If your system allows interrupt routines to nest, that is, it a higher-priority interrupt can interrupt a lower-priority interrupt routine, then another consideration comes into play.
- If the higher-priority interrupt routine makes any calls to RTOS functions, then the lower-priority interrupt routine must let the RTOS know when the lower-priority interrupt occurs.
- Otherwise, when the higher-priority interrupt routine ends, the RTOS scheduler may run some other task rather than let the lower-priority interrupt routine complete.
- Obviously, the RTOS scheduler should not run until all interrupt routines are complete.

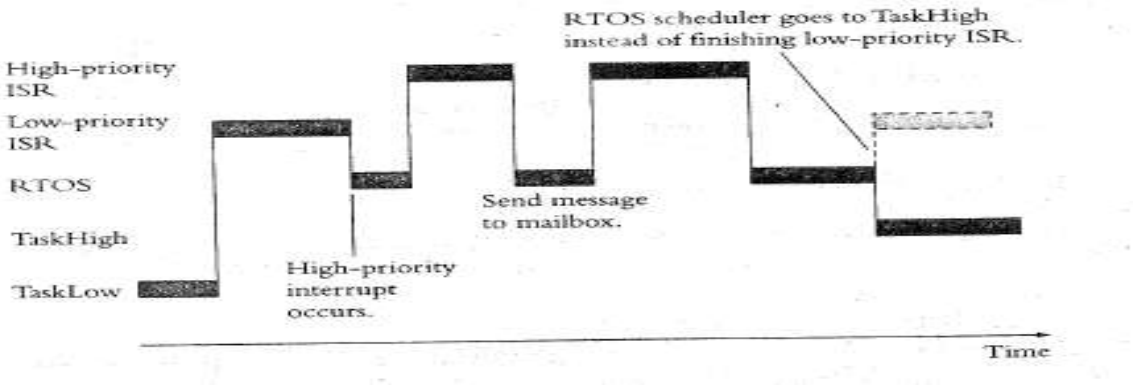


Fig 12: Nested interrupts and the RTOS