UNIT - VI

: INTRODUCTION TO REAL-TIME OPERATING SYSTEMS :

Definition :-

"A Real time operating System is an O.S intended to serve real time applications".

[OR]

"It is an operating system that guarantees a certain accomplishment of tasks of an application within a specified time constraint".

Difference between Desktop O.S & RTOS :-

There are many differences between desktop O.S and RTOS, few of them are tabulated as follows

Desktop operating system	Real-Time operating system
① The desktop O.S takes control of the machine peripherals & O.S first and then the application	① In RTOS the O.S takes control of the application first and then the RTOS resets
② The desktop O.S always runs a scan before initiation of an application	② The RTOS does not run a scan before starting an application.
③ the desktop O.S has memory which is present in the machine even if it is not needed.	③ The RTOS has the flexibility to configure memory based on the type of application.

There are numerous vendors that sell the RTOS like Vx works, VRTX, PSOS, Nucleous, C executive, QNX, multitask AMX.

- Most of the RTOS's follow a standard called "POSIX", proposed by IEEE.

Tasks and Task States :-

Task :- "The basic building block of software written under an RTOS, which is simply a subroutine is called a Task".

[OR]

"The Task is a subroutine which acts as a primary building block for a software that is expressed, under an RTOS".

- In a program, task begins by making one or more calls to a function.

States of Task :-

The various states of Task are

(a) **Running :-** Running state means the µp (or) µcontroller is executing the instructions that belong to the task. Hence only one task will be in the running state at any given time.

(b) **Ready :-** Ready state means, a new task is ready, but the µp is busy with some other task that is running inside it.

- this ready task can go ahead only after the Completion of running task by the µp.
- Any no. of tasks can be in the ready state

(c) Blocked :- Blocked tasks have no work to do or does not have any inst to execute even if the µp is available.

This happens because, they are waiting for an external event to occur.

- It remains in idle state unless it receives the action from external environment.

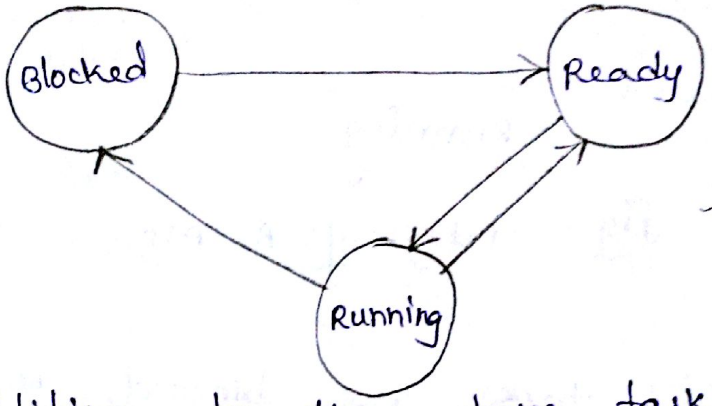

fig: Task states

- In addition to the above task states, RTOS offers some other states like Suspended, pending, waiting, delayed etc

Task scheduler :-

It is a part of RTOS, responsible for keeping track of various tasks states and also decides which among the multiple tasks should enter in to the Running state.

Functions of Task scheduler :-

1) It keeps track on the states of every task.
2) It takes decisions regarding the task that must enter the running state.
3) RTOS sets priorities to the tasks and the highest priority task runs and rest of the tasks will be in ready state.

- If the highest priority task holds the MP for long time, then the schedular solves the problem by assigning limited amount of time for the running task.

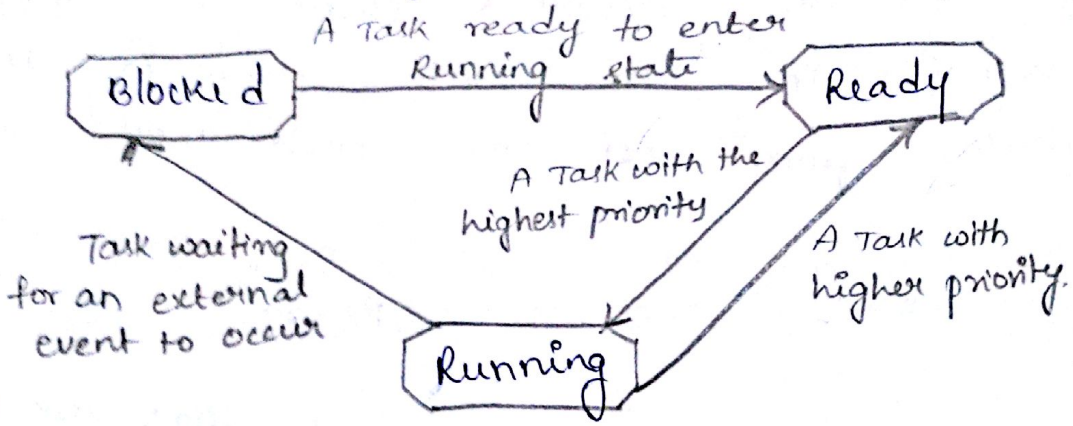

fig: States of A TASK

Note :-

* If all the tasks are blocked, then the schedular will spin in some tight loop somewhere inside of the RTOS, waiting for something to happen.
* If two tasks with same priority are ready, then some RTOS will time-slice between two such tasks.
* If a task is running and another higher-priority task unblocks, then
  - A preemptive RTOS will stop the current running task as soon as the higher-priority task unblocks.
  - A non-preemptive RTOS will not stop the current running task even when higher-priority task unblocks.

Tasks and Data :-

Each task has its own private context, which includes the registers, prog' counter & a stack.

- However, all other data - global, static, initialized, uninitialized and everything else is shared among all of the tasks in the system.
- The tasks in RTOS are like threads in UNIX than like processes.
- An RTOS has its own private data structures which are not available to any of the tasks.
- Since we can share data variables among tasks, it is easy to move data from one task to another.

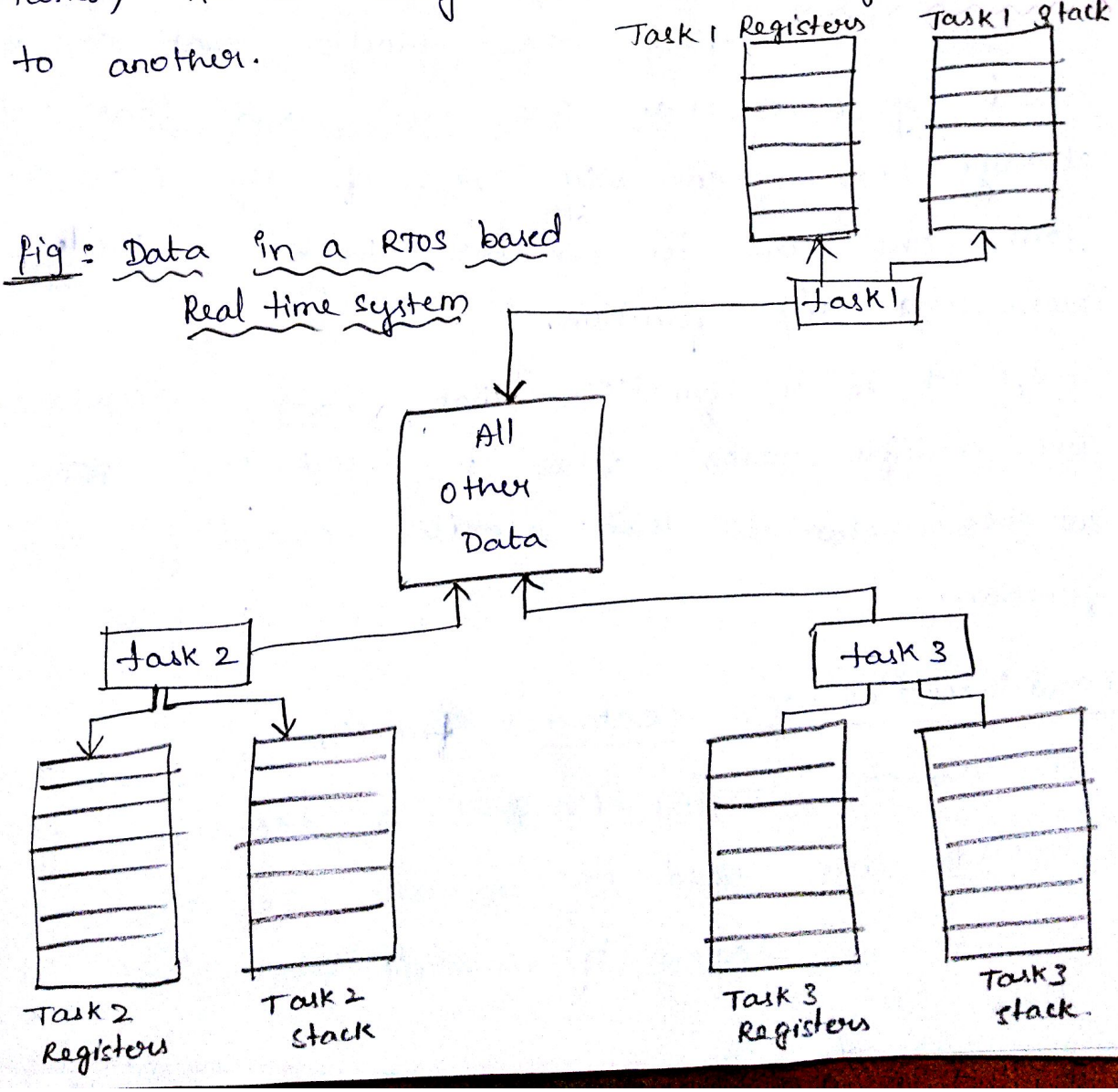

Fig : Data in a RTOS based Real time system

Shared Data problems :-

Shared data refers to the data that can be used by multiple tasks in such a way that exclusive access to the data for avoiding contention and corruption is ensured.

- Due to the same data being shared by more than one task in a RTOS there might arises shared data problem, which can be solved by two methods
  - (a) Reentrant function
  - (b) Semaphore.

Reentrant function :-

These 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.

- i.e, It is a function that works accurately even multiple tasks calls it and the RTOS switches between them while executing the function.

Characteristics of Reentrant function :-

* It cannot modify itself. A sequence of threads are used to execute the code.
* It can be shared by multiple tasks at a time.

* It is a "read-only" function.
* It has the capability to re-invoke itself by interrupting the present file that is being executed.
* A reentrant function can call only other reentrant functions.
* A reentrant function cannot make use of variables and I/O in a non-atomic way.
* A reentrant function can be interrupted and resumed at any time without losing any data.
* A reentrant function is capable of using both the local and global variables.

Semaphores and shared Data :-

A synchronization tool called semaphore is an integer variable that is obtained by using two operations "wait" and "signal".

- the "wait" operation is carried out as

```
wait (s)
{ while (s<=0)
    ;
  s--;
}
```

- the "signal" operation is carried out as

```
signal (s)
{
  s++;
}
```

- when one process alters the value of semaphore then no other processes can alter the value at same time.

- The binary semaphore can call two RTOS functions
  * TakeSemaphore & * ReleaseSemaphore.
- Suppose a task calls a "TakeSemaphore" and does not call "ReleaseSemaphore" then any other task which is in a queue to call "TakeSemaphore" get blocked unless and until the former task calls for "ReleaseSemaphore".
* Semaphore cannot be shared as only one task can use it at a time.
* Semaphore helps in solving the problem of data sharing.

Semaphore as a Signaling Device :-

Another common use of semaphores is as a simple way to communicate from one task to another or from an interrupt routine to a task.
- For Eg, Suppose the task that formats printed reports builds those reports in to a fixed memory buffer
- Suppose also that the printer interrupts after each line, and that the printer interrupt routine feeds the next line to the printer each time it interrupts. In such a system, after formatting one report in to the fixed buffer, the task must wait until the interrupt routine has finished printing that report before it can format the next report. One way to accomplish this is to have the task wait for a semaphore after it has formatted each report. The interrupt routine signals the task when the report has been fed to printer, by releasing the semaphore.

∴ Semaphore act as signalling Device.

problems due to semaphores :-

The problems that may arise due to semaphore are.

⇒ A semaphore functions only when all the task uses it to access shared data for read or write purposes. If any task forgets to take the semaphore then the RTOS shifts itself from that task thereby leading to various unexpected bugs.

⇒ If a task that has taken semaphore to execute its various instructions and forgets to release it then the other tasks that are in the queue waiting for the semaphore release gets blocked and will remain in blocked state forever.

⇒ priority Inversion :- If a task takes a semaphore and holds it for a long time then the other tasks that are waiting for the release of semaphore can miss out their real-time deadlines.

⇒ deadly embrace :- Reserving a semaphore causes a situation called "deadly embrace" which can be avoided if the semaphores are not reserved or by specifying a subroutine for each semaphore.

Different kinds of Semaphores :-

The different kinds of semaphores are

(a) Counting Semaphores :- counting semaphores are basic semaphores that can be used many times. They are actually the integers which can either be incremented or decremented. Taking the integer decreases the value and releasing it increases the value.

(b) Resource Semaphore :- These semaphores are released only by the task that took it. They are helpful in shared data problem but cannot by applied for comm among two tasks.

(c) Mutex Semaphore (or) Mutex :- RTOS provides two types of semaphores. The first kind of semaphores called mutex, are those that deal with priority inversion problem. The other semaphore do not deal with any such problem.

Various ways to protect shared data :-

The three methods to protect shared data are (i) disabling Interrupts.
(ii) Taking semaphores
(iii) Disabling task switches.

(i) Disabling Interrupts :-

→ when the interrupts are disabled they effect the response time of every interrupt routine and also those tasks that don't need as semaphore.

→ once the interrupt is disabled, it also leads to the disabling of task switches as it is not pomible for the scheduler to switch to the task that do not have any control over the procesor.

→ there are two merits of disabling interrupts

* Disabling interrupt is the only method applicable when the data is shared among task code and

interrupt routine. Interrupt routine have no rights to take semaphore. Also the interrupt are not prevented by disabling task switches.

* It is fast as many of the µp can enable or disable the interrupts using any one instruction

(ii) Taking Semaphores :- This method has its effect only on those tasks that are waiting for the same semaphore but doesn't have any effect on the tasks that don't need semaphore. Even the response time of interrupt routines remains unchanged.

(iii) Disabling Task switches :- This method comes in between the disabling interrupts and taking semaphores. It doesn't have any effect on interrupt routines but halts the generation of response for all the other tasks.

Message Queues, Mailboxes and Pipes :-

Inter-Task Communication :-

Tasks must be able to communicate with one another and coordinate their activites so as to share their data is called "Inter process Communication".

- The various methods that most RTOS's offers for inter procen Comm' are.
  (a) semaphores (b) Message queues
  (c) Mailboxes (d) Pipes.

⇒(a) Semaphores :-

Refer to page 4.

⇒(b) Message Queues :-

A Queue is an ordered collection of items in which items are inserted from one end (rear end) and deleted from other (front end)

- Queue is also known as FIFO (First in First out)

Primitive Operations :-

There are two basic operations that can be performed on Queue. They are

(1) Insertion : Adds a new element at rear end.

(2) Deletion : Delets an element from the front end of the Queue.

Flags :- Queues in embedded systems internally maintains the following five flags. They are

* QerrorFlag :- This flag becomes true (1) when ever an error occurs.

* headerFlag :- It becomes true when header byte are present in a Queue. Header bytes are not items of Queue, but they store some additional information about the Queue (like size of Queue etc)

* trailingFlag :- It becomes true when trailing bytes are present in a Queue. These are also addition info bytes which stores about the 'checksum' for error checking.

* cirQuflag :- It becomes true when some function is performing "circular queuing" of bytes in a queue.
* polyQuflag :- This flag becomes true when a function is doing "circular queuing" on a queue which is present in more than one block.

Operations :-

→ Insertion operation :- Algorithm :-

1. If queue is not "full" increment 'Rear'. then insert an item into the queue.
2. otherwise, if queue is empty, set QerrorFlag = true and call ISR_Qerror() function to display an error menage.

code :-

```
void insert (int item)
{ if (!(rear == q.size-1))
    { rear++;
      Queue [rear]= item;
    }
  else
    { QerrorFlag = true;
      ISR_Qerror (QerrorFlag, "Queue is full");
      /* ISR to print error message */.
    }
}
```

→ Deletion operation :- Algorithm :-

1. If queue is not "empty", retrieve the item pointed by 'head' and store it in 'result' variable and increment the 'front' pointer by 1.

2. otherwise, if Queue is empty.

Set Qerrorflag = TRUE and call ISR_Qerror() to print error message.

code :-

```
int delete()
{
  if (! front == -1)
  {
    result = Queue [front];
    Queue [front] = 0;
    if (front == rear)
      front = rear = -1;
      front++;
      return (item);
  }
  else
  { Qerrorflag = true;
    ISR_Qerror (Qerrorflag, "Queue Empty");
  }
}
```

(c) Mailboxes :-

Mailboxes are similar to queues.

- Some RTOS uses create, write, read functions from the mailboxes.
- RTOS checks the menages in the mailboxes and can even delete them when not in use.
- the capacity, functioning of mailboxes varies for different RTOS.

* when a mailbox is created, the capacity of menages are fixed and only that no. of

messages are allowed to be written.

* various RTOS allows any no. of messages to be written in every mailboxes.
* The total no. of messages in all the mailboxes are limited as per requirements.
* Some RTOS prefer prioritization of mailbox msgs (i.e, messages of higher priority is read first irrespective of the order in which they are written).

(d) pipes :-

pipes like queues and mailboxes can be created, written and read by the RTOS with some variation from one RTOS to another. The variations are

* variable length msgs can be written onto pipes.
* Some RTOS contains pipes that are byte oriented.
* "fread" and "fwrite" are the standard clibrary functions that are used by various RTOS to read and write onto the pipes.

Timer Functions :-

Tracing the passage of time is important in embedded systems for two reasons.

(a) To extend the battery life, if the system is battery based.

(b) To wait for an acknowledgement and perform data retransmission, if no acknowledgement is

received and if the system is connected to a network.

- Timers are used to measure the elapsed time of events. For instance, the kernel has to keep track of different times,

* A particular task may need to be executed periodically, say, every 10 msec. A timer is used to keep track of this periodicity.
* A task may be waiting in a queue for an event to occur. If the event does not occur for a specified time, it has to take appropiate action.
* A task may be waiting in a queue for a shared resource. If the resource is not available for a specified time, an appropiate action has to be taken.

Example :-

When a caller uses a telephone to make a call, listens the presed digital tone fore one-tenth of a second and then listens the tone of second digit after a span of one-tenth second.

The below code is used to make a telephone call through the function VMakeCallTask.

- A phone number from the RTOS message Queue is obtained by mMakecallTask when the phone number copied to a_chphoneno by

msgqrecv function.

- the taskDelay function in the while loop is used to generate first silence and then tone for each of the one-tenth second of time.
- The tone generator is turned ON & OFF by using the functions vTone On & vToneOff.

```
extern  QID  qcall;
void  vMakeCallTask (void)
{  #define LENGTH 11;
   char a-chphoneno[length];
   char *p-chphoneno;
   =
   while (TRUE)
  {
  msgqrecv (qcall, a-chphoneno , LENGTH , WAIT_FOREVER)
   p- chphoneno  = a-chphoneno;
   while (*p-chphoneno)
   { taskDelay (100);
     vTone On (*p-chphoneno- '0');
     taskDelay (100);
     vTone OFF ();
     ++p-chphoneno;
   }
  }
}
```

- the taskDelay function uses system ticks as its parameter which can be controlled when a system is setup.

- There are other timing services that are based on system tick where time limit is allotted for accomplishing a task.
- For eg, if a time limit is assigned to a high priority task for taking a semaphore and if it expires then the task cannot access the shared data. The code is needed to be written for accessing the semaphore again.

∴ By sending the instruction to lower priority task, the higher priority task can continue with its own execution.

Events :-

An event is a boolean flag that can be programmed by the tasks and can be used by some other waiting tasks.

for eg :- In a cordless barcode scanner, an event can be set whenever the trigger is pulled by the user, and the laser scanning procen begins its operation.

- Thus the purpose of an event differs from regular operating system to an RTOS.

def :- Events are the tasks, which the RTOS can trigger.

The features of events are

* One event can block many tasks, once the event is completed the RTOS unblockes all the tasks.

* All the events are grouped together, so any subset of task can wait for any task in the same group.
* the events can fire any interrupt once they are complete.
* Every RTOS has its own way of handling the following issues.
  (a) Resetting of an event after its occurance.
  (b) unblocking the tasks that are blocked and waiting in a queue.

Memory Management :-

memory management in operating System is same as memory management in RTOS.

RTOS designers mostly don't use "malloc" and "free" functions provided by C, because they are slow and their execution time is unpredictable.

RTOS designers, design their own fixed size memory blocks and use them for memory management. They refer this as pool of memory blocks.

- In a pool all the memory blocks are of the same size.
- There are two functions similar to malloc & free, which will provide us with the memory block.

They are
→ reqbuf and
→ getbuf

} These two functions are the requesting, obtain & releasing a memory buffer.

- The difference between 'reqbuf' and 'getbuf' are reqbuf returns NULL if the pool is empty and 'getbuf' blocks the task calling it.

```
void *getbuf (unsigned int upoolid, unsigned int utimeout);
void *reqbuf (unsigned int upoolid);
void *relbuf (unsigned int upoolid, void *p_vBuffer);
```

- unlike Desktop systems, in RTOS the memory has to be initialized.
- The disadvantage is that there are fixed size of memory buffer and even if wee need less amount of memory, our functions will return the fixed size memory.

This may result in lot of memory wastage, but the response and performance is good. To overcome this problem we can have pools of different memory blocks.

The memory blocks in each pool will be of same size, but the different pools have different size memory blocks.

- This results in using shared memory from all the pools, which may reduce the memory wastage to some extent.

Interrupt Routine in an RTOS environment :-

There are two rules that must be followed by the interrupt routines in RTOS.

Rule 1 :- (No Blocking) :

An interrupt routine must not call any RTOS function that might block the caller.

If an interrupt is called in b/w a task and if the interrupt is blocked then the task blocked for executing interrupt will also be blocked. Hence interrupt are not allowed to wait for semaphores, events, read from queues or mailboxes.

Rule 2 :- (No RTOS calls without fair warning) :-

An interrupt routine may not call any RTOS functions that might cause the RTOS to switch tasks unless the RTOS knows that an interrupt routine and not a task is executing.

If an interrupt routine and a tasks share data, then while executing interrupt RTOS may switch between tasks and may block for a long time. Hence, interrupt are not allowed to release semaphores or use events since they may be used by other tasks of low priority.

~~ISR > Blocking~~

- In Interrupt Routine the original flow sequence should be in below mentioned way.

Step1 :- RTOS starts executing a low priority task.

Step2 :- An interrupt occurs and RTOS blocks low priority task and starts executing the interrupt.

Step3 :- Interrupt requires a message to be sent to other task, during this time a high priority task arrives to the Queue.

Step4 :- when RTOS has finished sending the msg, it unblocks ISR and finishes it.

Step5 :- RTOS start executing the high priority task. Finish this and then complete the blocked low priority task.

Nested Interrupts :-

Nested Interrupt are same as nested loops. If an interrupt is called inside another interrupt then it is called nested Interrupt.

- There is a problem in nested Interrupt.

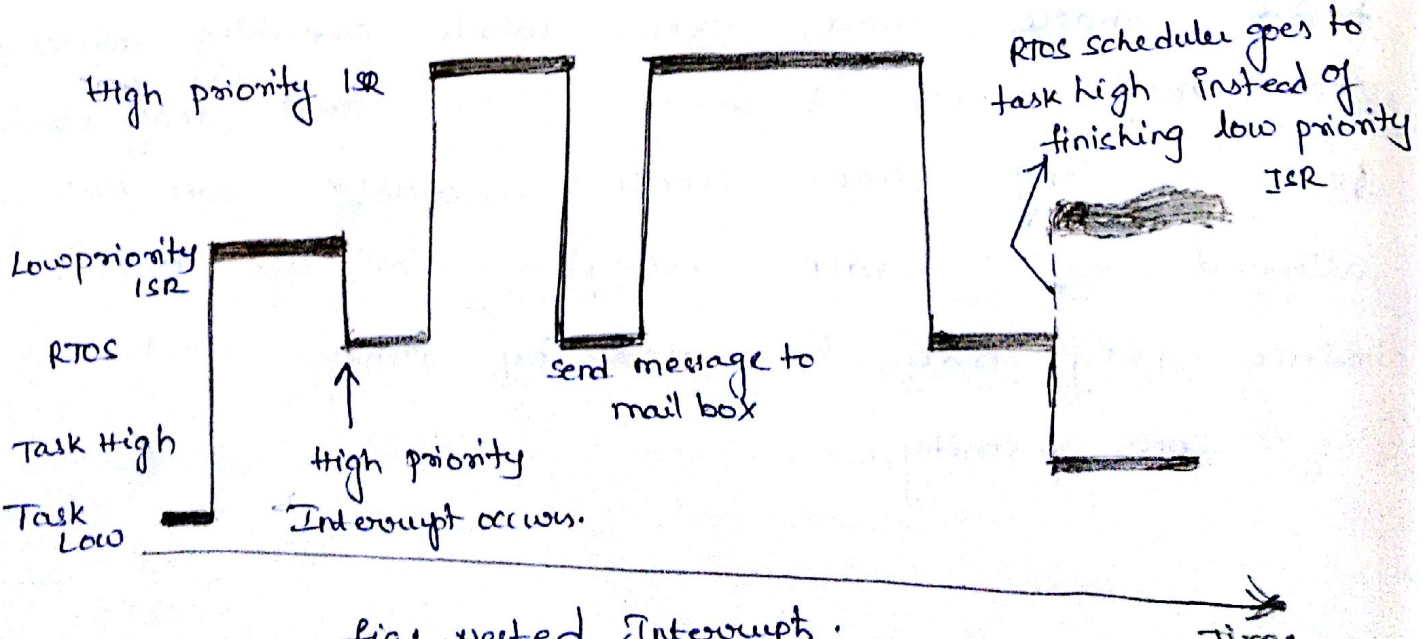

fig: Nested Interrupt.

- If low priority interrupt is nested in between high priority task, then the low priority task has to inform when it get completed to RTOS, otherwise the high priority task ends and leaves the control to RTOS scheduler, which may run some other task.
- To avoid this we should disable RTOS scheduler for the time Interrupt routines are executed.
- This concept is shown in the before figure.