RTOS and Microkernel

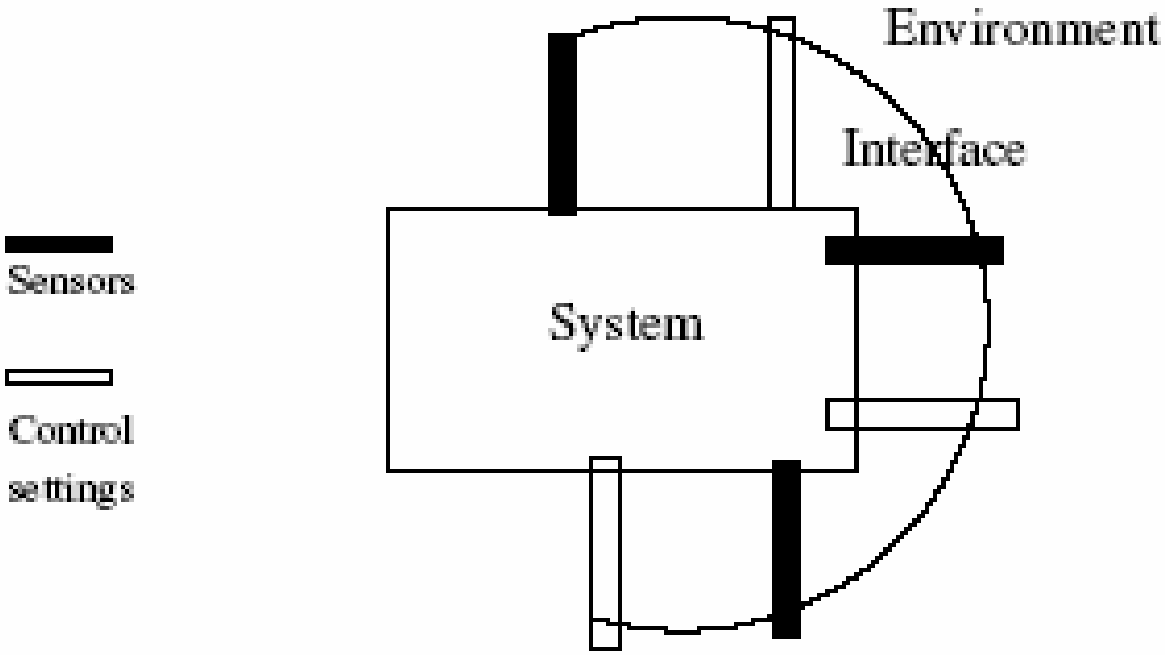
Prof P.C.P. Bhatt

Introduction to RTOS

- The first requirement is that the *system response* must be generated well within a *stipulated time.*
- Response after the stipulated time is irrelevant.
- A *delay* may result in *catastrophe.*

For a system which both *monitors* and *responds* to event from its operative environment, the system responses are required to be *timely*.

Characteristics of Real-Time Systems



In a real–time system the events in the environment are detected by sensors and the responses to these events are generated in a timely manner. A RTOS ensures that control settings (in response to an event) are achieved in real–time.

Operative Environment for RTOS.

Operative Environment for RTOS

The Key Factors

- Sensor
- Control Settings
- System
- Environment
- Interface

Typical Operating Sequence

1. Sense an event
2. Process the data
3. Decide on an action
4. Take corrective action

Examples of RTOS

Interactive systems

- Online stock trading,
- Tele-ticketing,

Life critical system

- Patient monitoring system

Safety Critical system

- Reactor control system
- Antilock braking system (ABS)

Why not use Unix or Windows

- The general purpose OSs are Designed with no specific application in mind. (No fixed domain of operations)
- These OSs are designed to enhance the *throughput.*
- *Events* in these environments arise from *multiple sources which are often unpredictable* sources.

General purpose *OSs stripped down* to meet real time requirements.

- *Windows CE*
- *Embedded Linux*

Classification of Real Time Systems

Consequences of failure in meeting real-time requirements.

- *Hard real time* - disaster such as loss of life: as in a satellite launch system
- *Firm real time* - may result in a missed opportunity: as a loss of trading opportunity – because one may miss out on the time for reconciliation of stocks.
- *Soft real time* - degraded service quality: as in video streaming application. A frame lsot may be no major consequence.

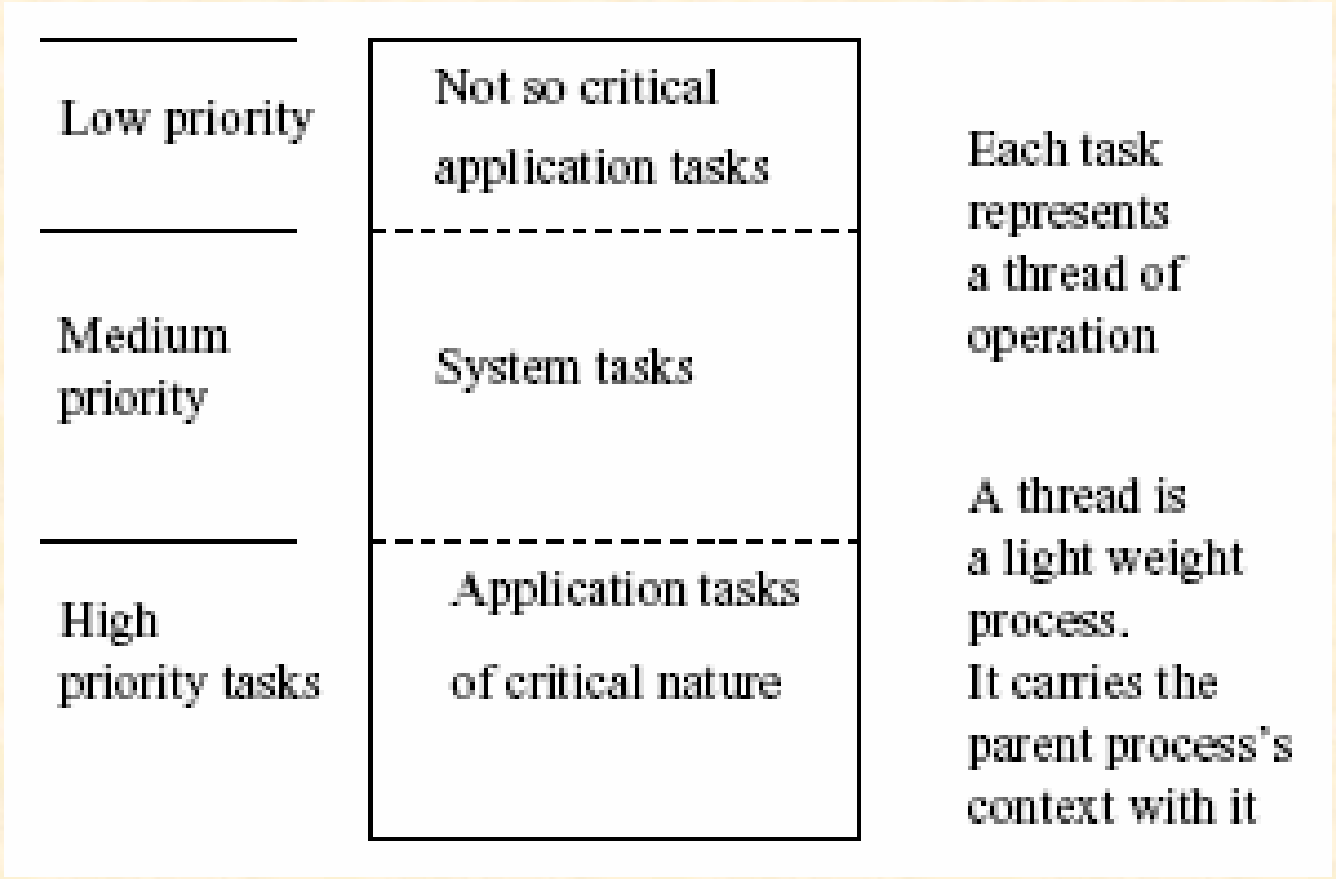
Architecture of Real-Time Systems

Two important features of the real time system:

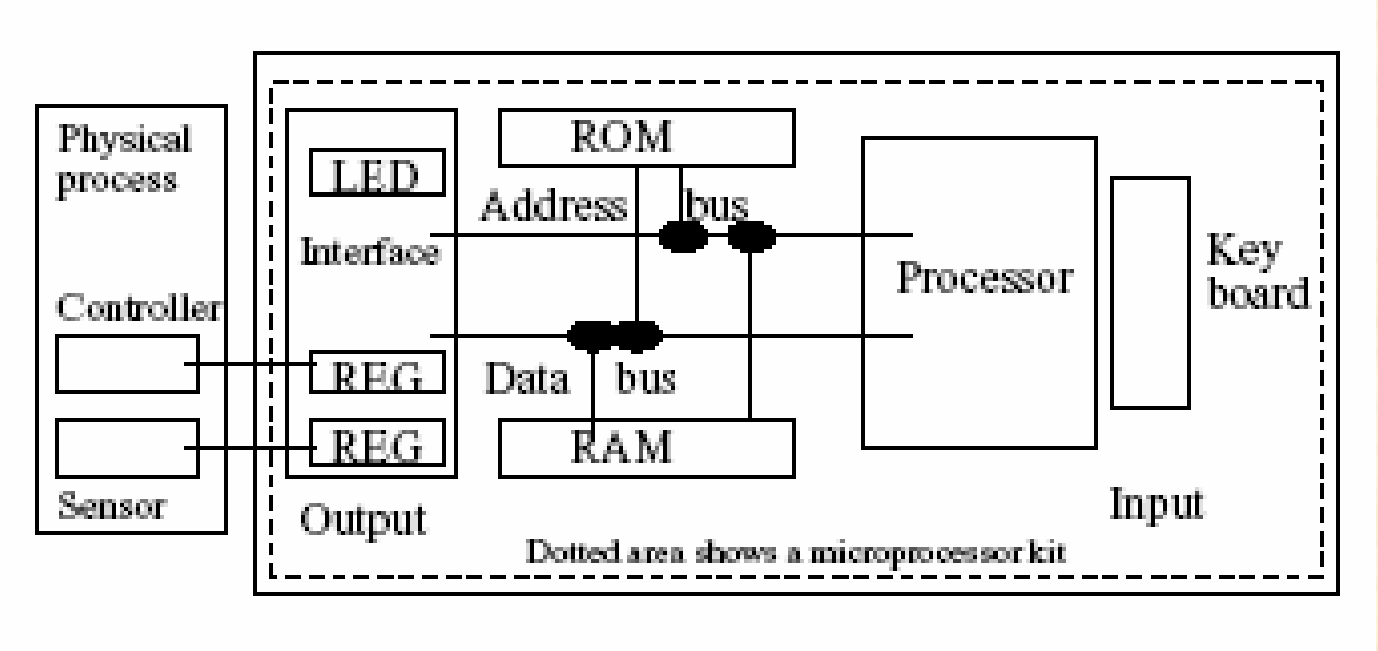
- Almost always a *Fault tolerant design.*
- The scheduling policy must provide for

  pre-emptive actions.

Priority Structure for RTOS Tasks.



Kit-Based Embedded System Architecture



We can program the kit in machine language. The program can be directly stored in memory. On execution we observe output on LEDs. We also have some attached ROM. To simulate the operation of an embedded system input can be read from sensors and we can output to an interface to activate an actuator. We can use the timers and use these to periodically monitor a process. One can demonstrate the operation of an elevator control or a washing machine with these kits.

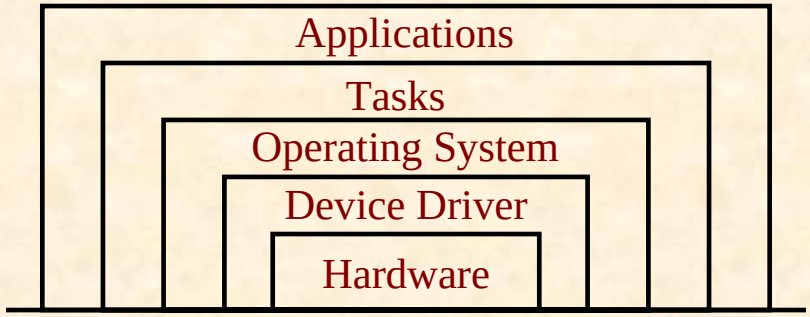
Microkernel - 1

- *Minimalist kernel* design suitable for RTOS.
- All IO requires use of *communication through kernel*, thus it is important that kernel overhead to minimum.
- Microkernel are minimal kernels which offer *kernel services* with *minimum overheads.*

The kernel *used* in *hard real time systems* are often Micro-kernels.

Microkernel - 2

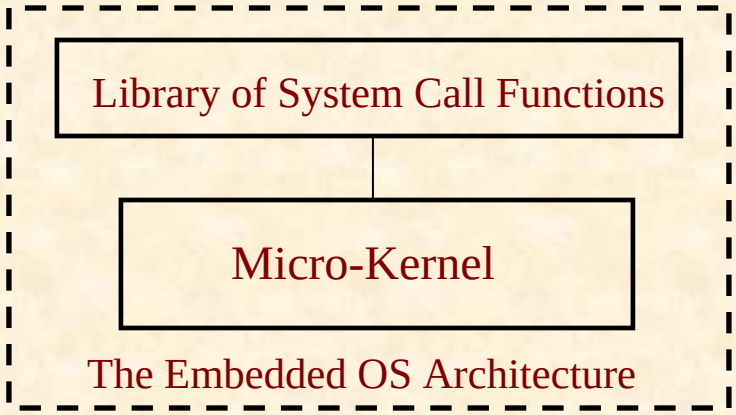
The *device driver functions* in this context are the following:



- Do the initialization when started. May need to store an initial value in a register.
- Move the data from the device to the system. This is the most often performed task by the device driver.
- Bring the hardware to a safe state, if required. This may be needed when a recovery is required or the system needs to be reset.
- Respond to interrupt service routine. The interrupt service routine may need some status information.

Microkernel - 3

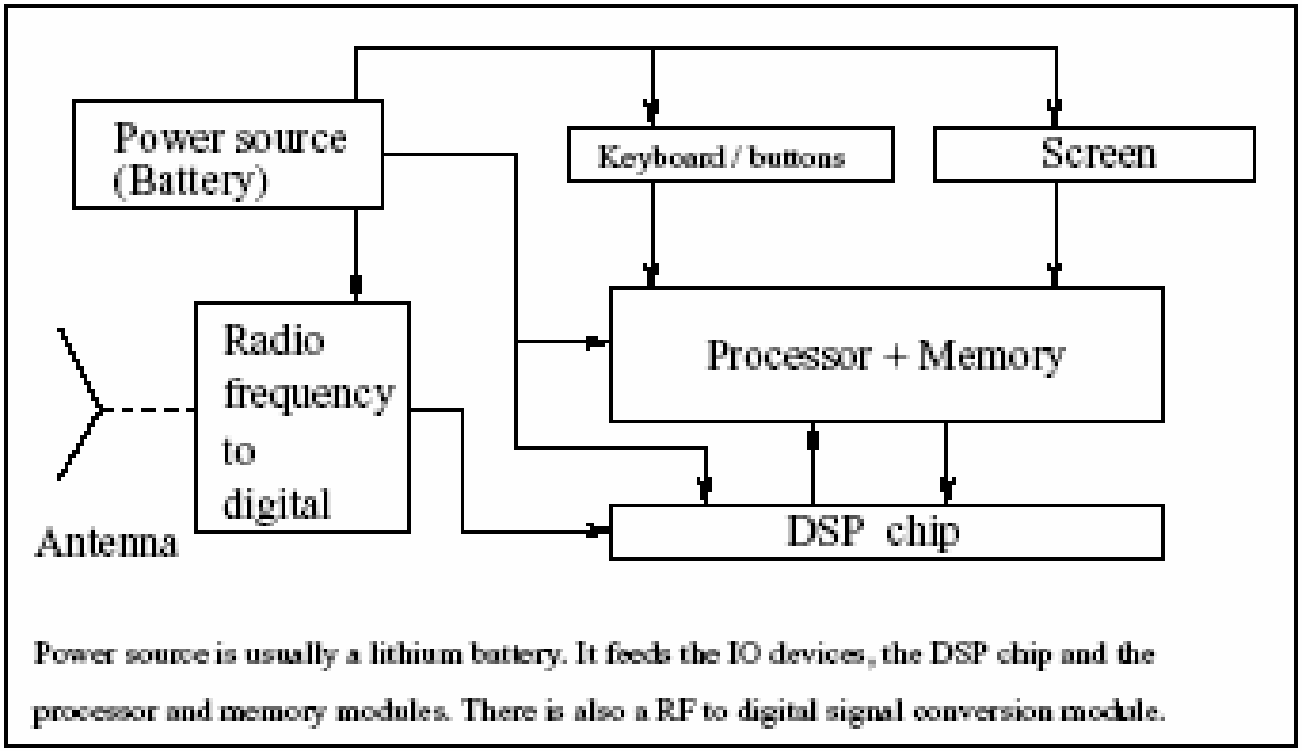
The microkernel together with this library is capable of the following:



The Embedded OS Architecture

- Identify and create a task.
- Resource allocation and reallocation amongst tasks.
- Delete a task.
- Identify task state like running, ready-to-run, blocked for IO etc.
- To support task operations (launch, block, read-port, run etc.), i.e. should facilitate low level message passing (or signals communication).
- Memory management functions (allocation and deallocation to processes).
- Support preemptive scheduling policy.
- Should have support to handle priority inversion.

OS for Hand-held Devices



A Typical Hand-Held Device Architecture.

Considerations in RTOS Scheduling

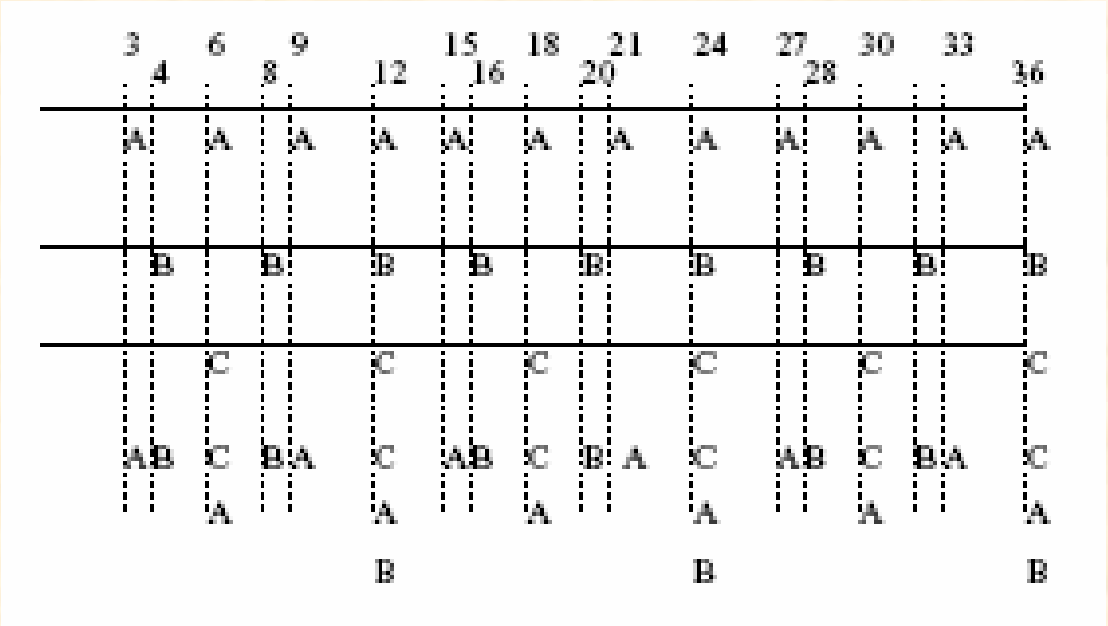
The two most important considerations in the *RTOS Scheduling* are :

1. *Predictability of responses*: for both synchronous as well as asynchronous inputs.
2. *Worst case schedulability* scenario analysis.

Rate Monotonic Scheduling

- *Periodic monitoring* and regulation.
- *Events* are *cyclic in nature.*
- *Predictable event detection* and consequent decision on control setting.

Rate Monotonic Scheduling : An Example



As an example, consider that events *A*, *B* and *C* happen with time periods 3, 4, and 6, and when an event occurs the system must respond to these. Then we need to draw up a schedule as shown in figure above.

Note that at times 12, 24, and 36 all the three tasks need to be attended to while at time 21 only task *A* needs to be attended. This particular schedule is drawn taking its predictability into account.

Earliest Deadline First Policy

- Handles tasks with *dynamic priorities.*
- *Priorities* determined by the *proximity of the deadline* for task completion.
- *Cannot guarantee optimal performance.*

Earliest Least Laxity First Policy

Try to utilize the slack time which may be available to start a task.

slack_time = dead_line - remaining_processing_time

- The slack defines the *laxity* of the system.
- Useful for *multimedia applications* in *multiprocessor environment.*

Priority Inversion

- A case where a lower priority tasks holds a mutually shared resource and blocks a higher priority task.
- Predictability of events get adversely affected.

Priority Inversion : An Example - 1

Suppose we have three tasks t_1, t_2, and t_3 with priorities in the order of their index with task t_1 having the highest priority. Now suppose there is a shared resource R which task t_3 and task t_1 share. Suppose t_3 has obtained resource R and it is to now execute. Priority inversion occurs with the following plausible sequence of events.

1. Resource R is free and t_3 seeks to execute. It gets resource R and t_3 is executing.
2. Task t_3 is executing. However, before it completes task t_1 seeks to execute.
3. Task t_3 is suspended and task t_1 begins to execute till it needs resource R. It gets suspended as the mutually exclusive resource R is not available.

Priority Inversion : An Example - 2

4. Task t_3 is resumed. However, before it completes task t_2 seeks to be scheduled.
5. Task t_3 is suspended and task t_2 begins executing.
6. Task t_2 is completed. Task t_1 still cannot begin executing as resource R is not available (held by task t_3).
7. Task t_3 resumes to finish its execution and releases resource R.
8. Blocked task t_1 now runs to completion.

The main point to be noted in the above sequence is : even though the highest priority task t_1 gets scheduled using a pre-emptive strategy, it completes later than the lower priority tasks t_2 and t_3!! Now that is priority inversion.

How to tackle priority inversion?

- In the previous example the priority inversion happened because we could not schedule task t_1 due to conflict on a mutually exclusive resource.
- Suppose when task t_1 sought resource – we were to temporarily raise the priority of t_3 to the level of t_1. We can immediately see that task t_3 would complete and task t_1 will get scheduled thereafter. In other words, task t_2 will not be scheduled as we saw in the previous example.
- This solution mitigates the dangers that lurk from the priority inversion from happening.

Resources

- *www.microsoft.com (Windows CE)*
- *www.linuxdevices.com (articles)*
- *www.linuxdoc.org*
- *www.gnu.org (open source information)*
- *www.symbian.com (hand held devices)*
- *oasis.palm.com (for PalmOS)*
- *www.windriver.com (VX works RTOS)*