



مدينة زويل للعلوم والتكنولوجيا  
Zewail City of Science and Technology

COMMUNICATION AND INFORMATION ENGINEERING

**CIE 314**

**Embedded Systems Fundamentals**

Lecture #7

Embedded Software Modeling  
Analysis and Design

**Instructor:**

**Dr. Ahmad El-Banna**



ESTABLISHED 2000

SPRING 2017

© Ahmad El-Banna

# Agenda

Software Requirement Specification

State Chart in embedded software analysis and design

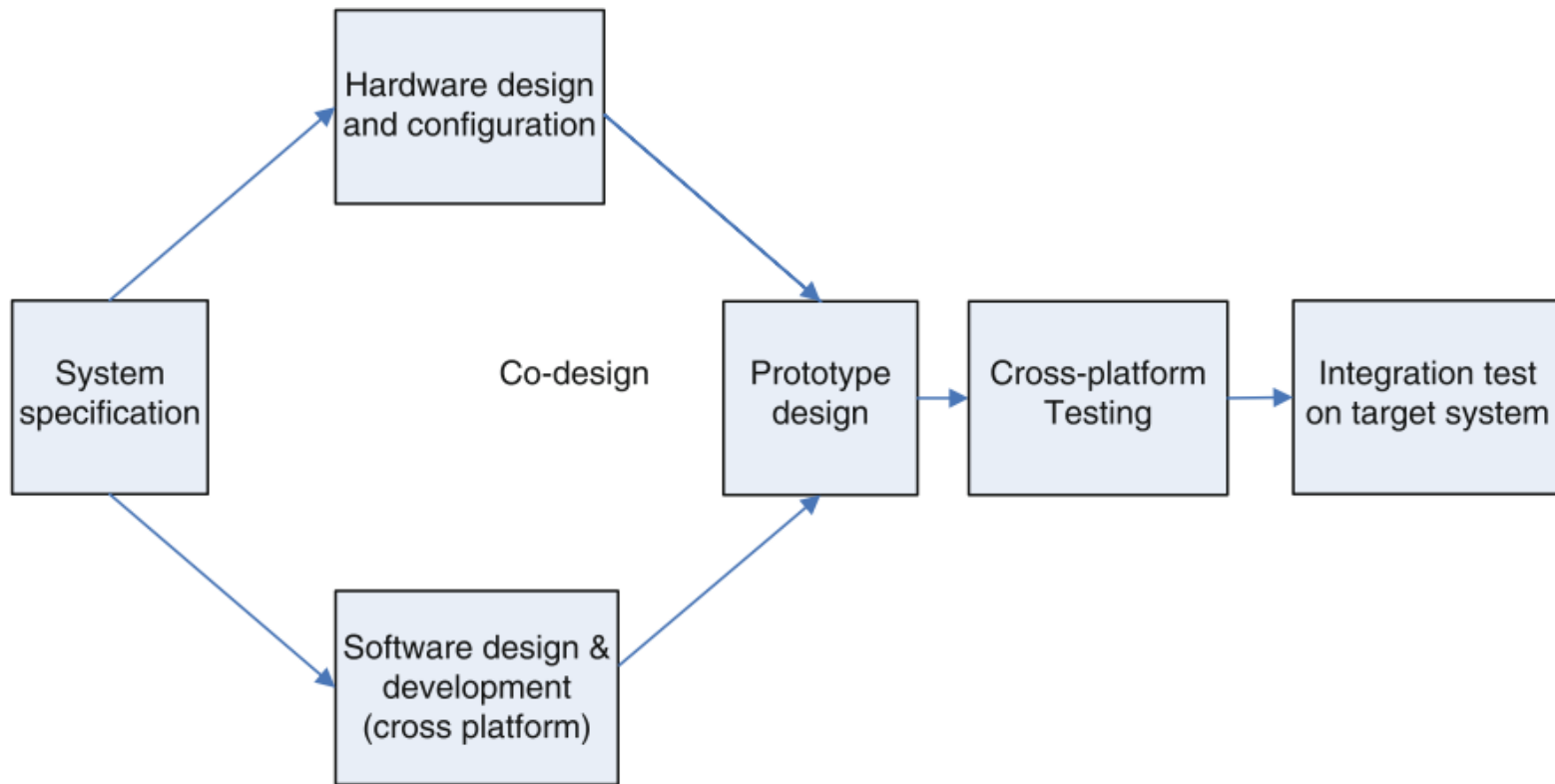
Time requirement analysis for real-time systems

Quiz

# Overview

- The embedded system software development has **same** Software Development Life Cycle (SDLC) just like any other software development **plus its special consideration** in the resource constrains including
  - CPU, time, memory, operating system, multi-tasking concurrency, and many other non-functional attribute constraints.
- In order to **reduce the time-to-market** and **guarantee the reliability** of the embedded system products, **software engineering methodology is recommended** for the software design and development.
- Because the **embedded software** is not deployed on a general-purpose computer system, the embedded software in C/C++ or other high level programming languages must be **developed** and **tested** on a **cross-platform machine** such as PC, and then be loaded to a target microcontroller memory to be tested.

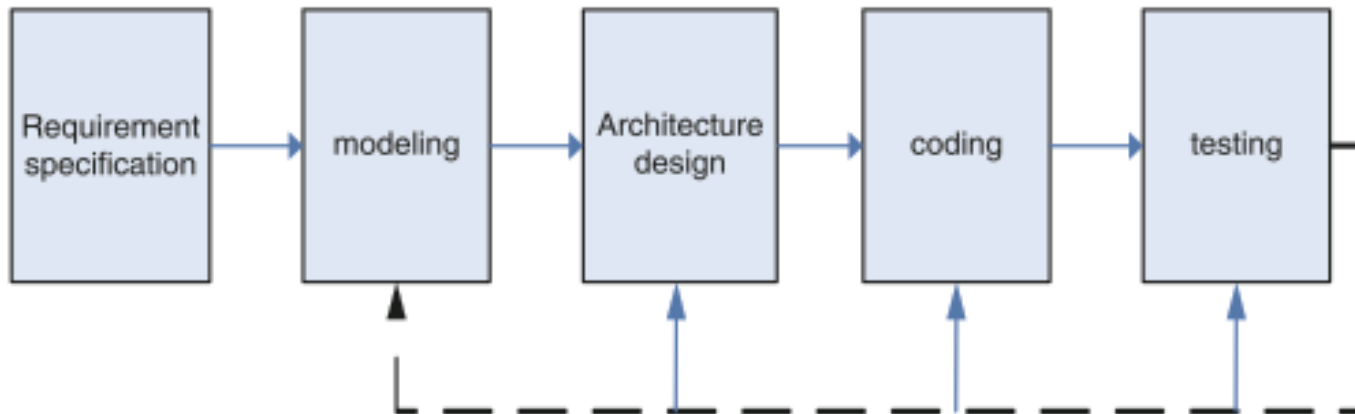
# Embedded system co-design



- In **co-design phase**, software analyst/designer need to work with hardware team members to **decide** on the **type of microcontrollers and microprocessors** according to the requirement of the speed and memory, power consumption, the number of I/O devices such as ADC, cost per unit, size and packaging, and others.

# SDLC

- The SDLC proceeds in **phases** consisting of system specification, analysis, design, coding, and testing.
- The dash-lines indicate the required **iteration processes** when the changes are needed.



# Software Requirement Specification

- Compared to general purpose software the **non-functional requirement** are especially **important** to all embedded software due to various **constraints** in resources and performance.
- All the **quality attributes** should be **identified** in the **requirement analysis process**.
- Quality attributes can be **categorized** into three groups:
  1. Implementation attributes (not observable at runtime)
  2. Runtime attributes (observable at runtime)
  3. Business attributes

# 1. Implementation attributes (not observable at runtime)

- **Resource availability:** This refers to **memory** space, **power** supplier, **CPU** frequency, and **RTOS**.
- **Maintainability & extensibility:** This refers to the ability to **modify the system** and extend it conveniently.
- **Testability:** This refers to the degree to which the system **facilitates** the establishment of **test cases**. It usually requires a complete set of **documentation** accompanied with system design and implementation.
- **Flexibility:** This refers to the **ease of modification** of a system to cater to different environment or problems for which the system is originally not designed.

## 2. Runtime attributes (observable at runtime)

- **Availability:** This refers to the ability of a system to be **available 24x7**. Availability can be achieved via replication and **careful design to cope with failures** of hardware, software, or the network
- **Security:** This refers to the ability to **cope with malicious attacks** from outside or inside of the system. Security can be improved by installing **firewalls** and establishing **authentication** and authorization processes, and using **encryption**.
- **Performance:** This refers to **increasing efficiency** such as **response time, throughput** and generally resource utilization, which most of the time conflict with each other.



Contd.

- **Usability:** This refers to the **level of “satisfaction” from a human perspective** in using the system. Usability includes **completeness, correctness, compatibility**, and user **friendliness** such as friendly user interface, complete documentation, and help support.
- **Reliability:** This refers to the **failure frequency**, the **accuracy** of output results, the mean-time-to-failure (MTTF), the **ability to recover from failure**, and the failure predictability

### 3. Business attributes

- **Time to market:** This refers to the **time** it takes from **requirement analysis to** the date product is released.
- **Cost:** This refers to expense of **building, maintaining,** and **operating the system.**
- **Lifetime:** This refers to the period of **time** that the product is “**alive**” **before retirement.**

# Embedded Software Modeling Analysis and Design

## ■ Context Diagram

- The analysis process starts with the system context analysis, **system key tasks**, and their **relationship** identification.
- The **focus** is given to the **reactive** behaviors of the embedded software.
- The **event behavior** features include: **periodic** or aperiodic events, parallel or **serial** data processing, **synchronous** or asynchronous communication, real-time or soft-time in response reaction, and others.
- Based on the **system modeling** the design process produces an **implementation guideline** for developers such as multi-tasking processing architecture.

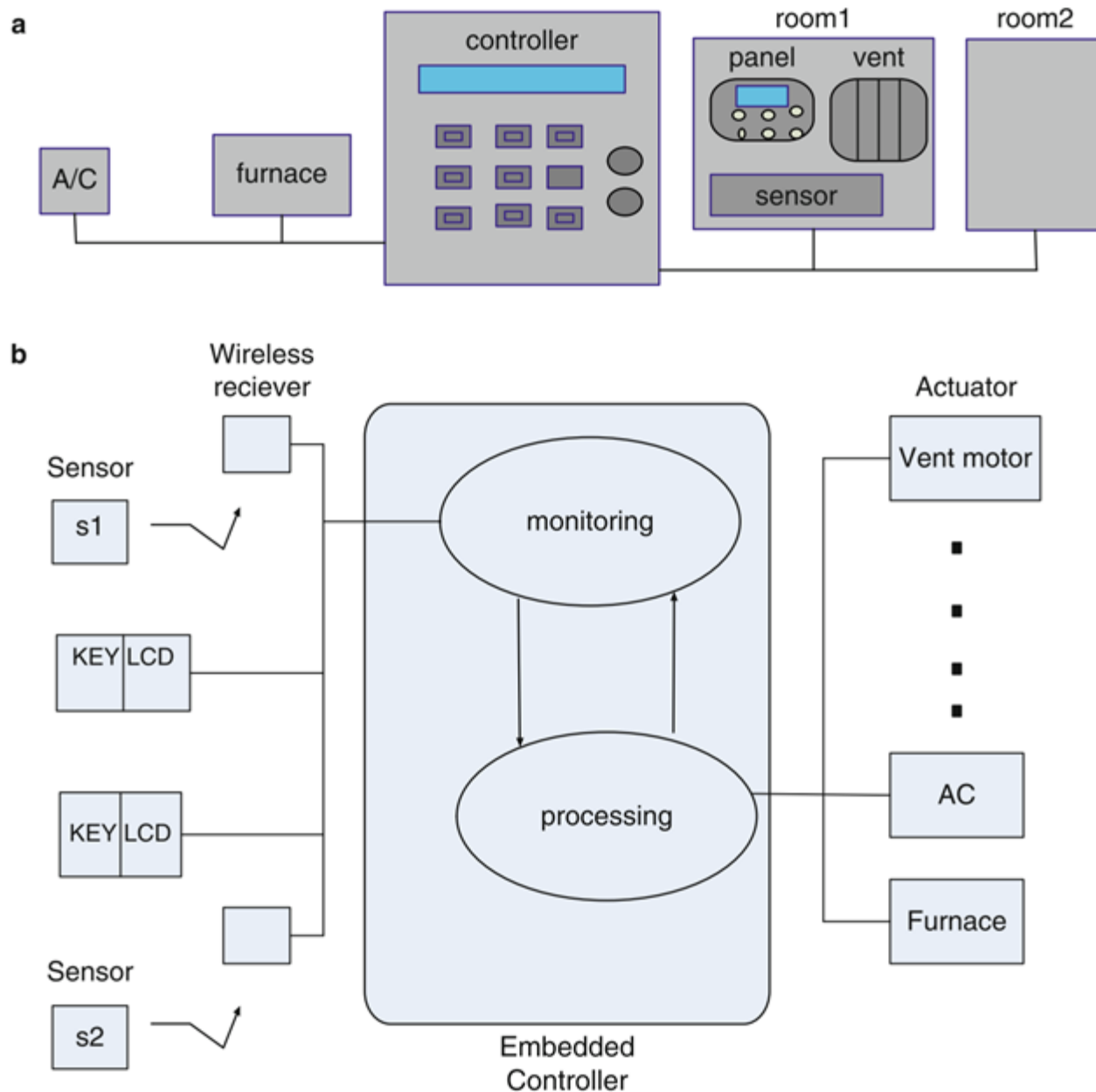
## Contd.

- The context diagram generation may be the **first step** in the embedded software modeling process.
  - The context diagram is **derived from the requirement specification**.
  - It provides a **preliminary draft** of the embedded software and its environment which **shows all events the system interacts with**.
  - It **specifies elements** of a system, **data** and **data flow** around the system, and the system internal and external **communication**.

# Example:

## Multiple room temperature control **system specification**.

- The system provides centralized and distributed temperature control functions.
- A local desired temperature can be set by a room control panel in each individual room or a global temperature can be set at the central control panel.
- The temperature sensors are installed in each room to monitor the current room temperature.
- The A/C and furnace are turned on or off by the controller depending on the desired temperature and the current temperature.
- Each room has a vent driven by a motor to control the air flow so that different rooms may reach different desired temperatures.



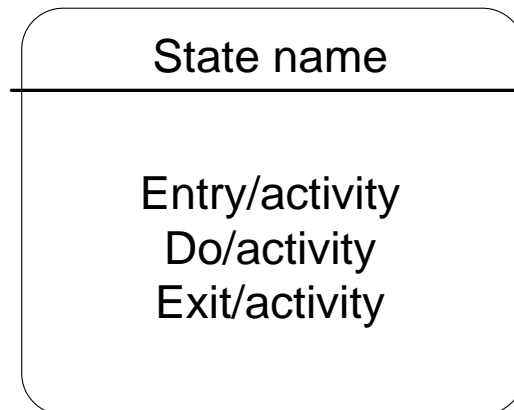
**Fig. 2.3** (a) system (b) Context Diagram

# Finite State Machine (FSM) and StateChart

- The system **modeling** presents an abstraction of system (**a high level description**) in software aspects which **helps understanding** of the functional **requirement** in **block diagram** and helps to identify all required software elements and tasks.
- There are **many modeling tools available** for modeling the behaviors of embedded system software such as Universal Modeling Language (**UML**) and **Petri Net**.
- The **stateChart** diagram is a very **popular** analysis modeling method for **real time embedded software** which is **derived from** Finite State Machine (**FSM**).

# State

- A **FSM** consists of a finite number of **states**, **transitions** between the states, and **actions** to be performed.
- A **state** represents a **certain behavior** which lasts for a **period of time** in performing tasks **or** being **idle** waiting for events .
- In the **state notation** where you can specify the **entrance actions**, **state actions**, and **exit actions**

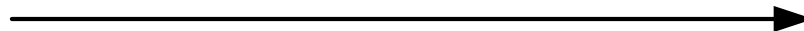




# Transition

- A **transition** indicates a **state change guided by condition** or events.
- A transition is a **response to an event** with a set of attributes that moves the system from one state to another state.
- There **may be many transition links from one state to other** states with different events or with same event but different guide conditions.
- An **event** can refer to either **internal** event (such as timer timeout overflow) or **external** event (a car detected by sensor), a **condition** can refer to a **set of variable values** and a **reaction** can refer to **reassignment** of these **variables** or new **event creation**.

Event(attributes)[guide condition]/reaction



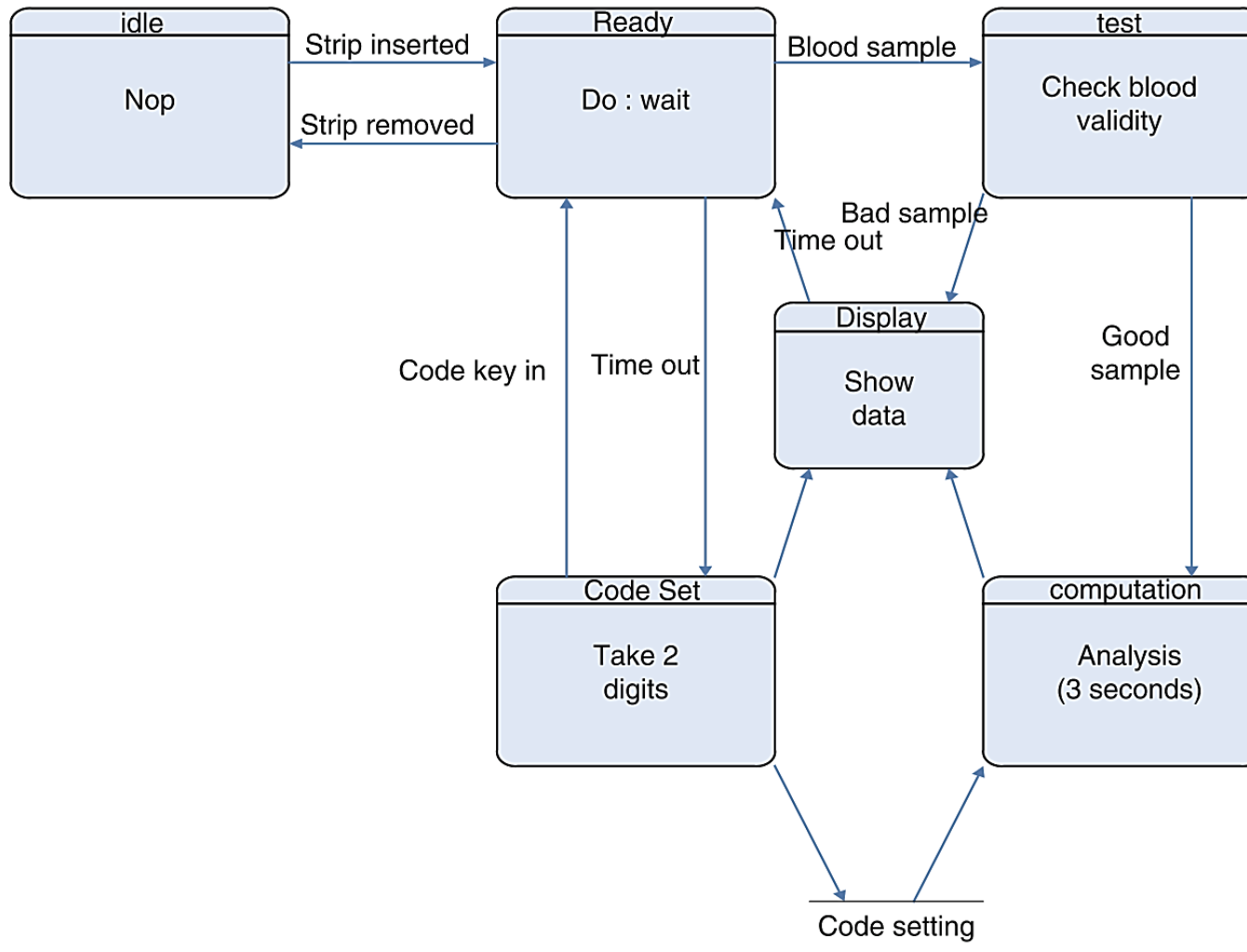
# Example:

## glucose monitor System **functional requirements**

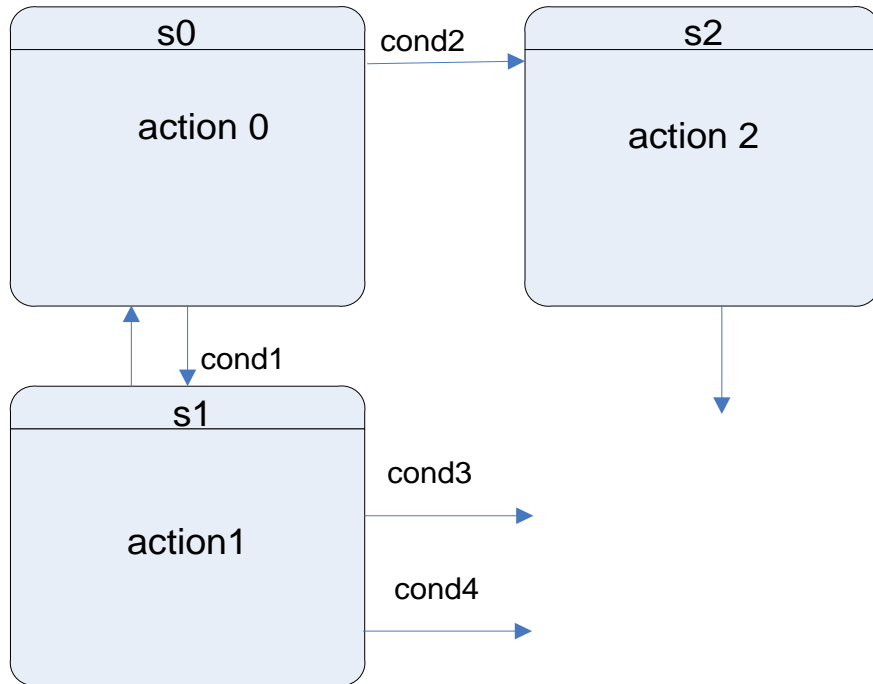
- When the test strip is inserted into the monitor the system moves to a ready state waiting for the blood sample and the system moves back to the idle state when the strip is removed.
- At this time patient can set the strip code in two digits and it is saved in the data store. If the blood is not enough it displays the error message is displayed in the display state and system goes back to the ready state to wait for new blood sample.
- If the blood sample is valid it moves to the computation state to analyze the glucose level with the code reference from the data store and report the result.

Sketch the FSM of this system!

# Simplified FSM diagram for glucose monitor



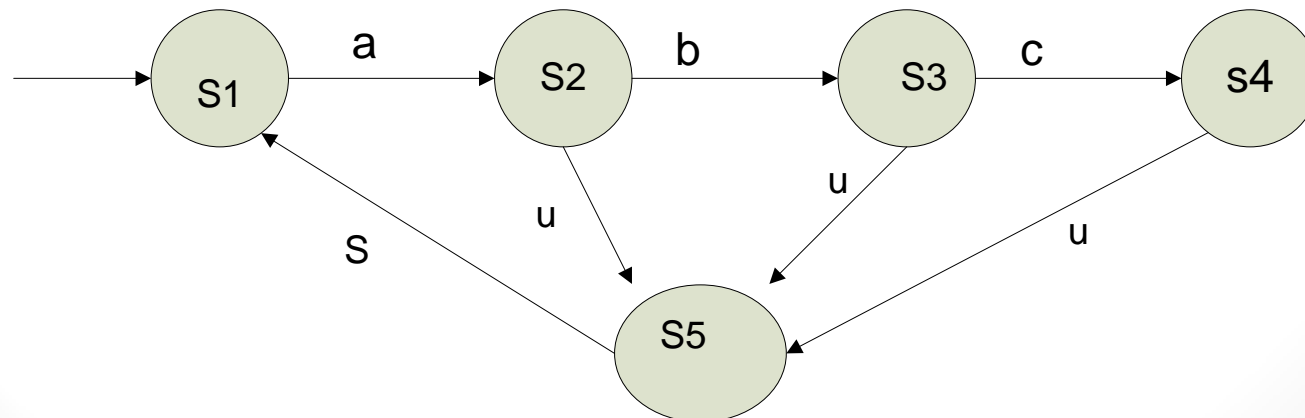
# Simple FSM Implementation



```
init state S0;
while(1)
{
switch(state)
{ S0: action0;
  if (cond1) state = S1;
  else (cond2) state = ...;
  break;
S1: action1;
  if (cond3) state= ...;
  else if (cond4) state = ...;
  else state = S0;
  break;
S2: action2;
  If (...) ...
}
}
```

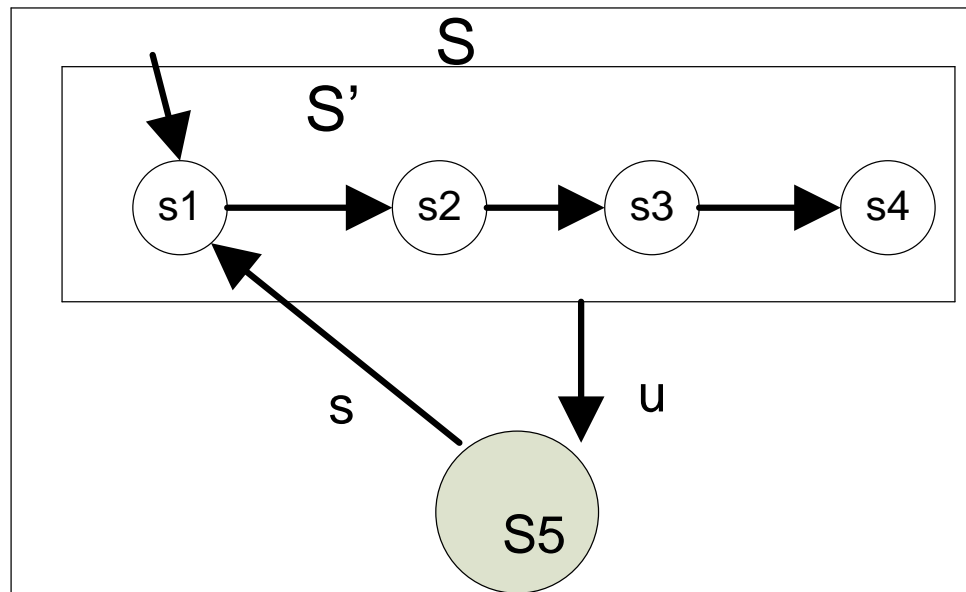
# Flat FSM

- FSM **can't represent multiple states active simultaneously** and it can't describe any nested states within states in a hierarchical structure neither.
- It can **only** be used for **simple system** with few states.
- A simple **flat FSM** for a counter or validation task, where s1-s4 take inputs and validate input, if the input are not valid then the system moves to state S5 to handle exception, and then goes back to initial state to start over again.
- It will be too **complicated** if the number of **states goes large**.

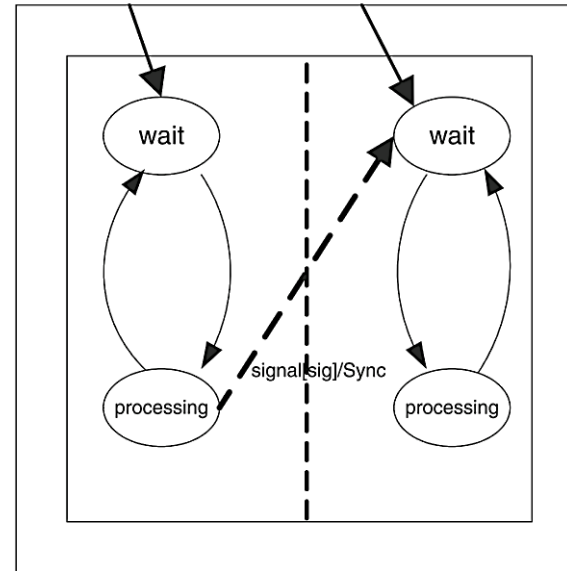


# Nested State Chart

- This figure groups these state to make a **nested state S'** so that the **transition** between S' and S is greatly **simplified**.
- In this way a state can be refined into a set of sub-states until **each state is simple enough to have its sole responsibility**.



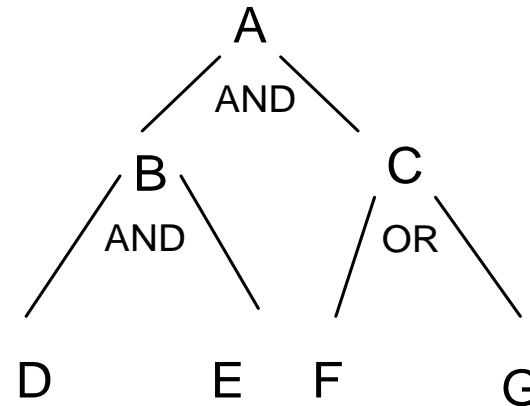
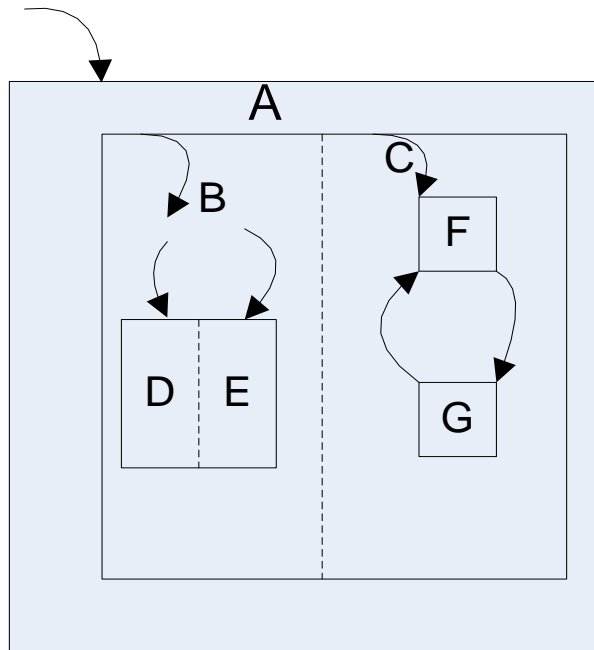
# A simple State Chart with concurrent tasks



- The figure demonstrates **two concurrent tasks** that are **active at same time**.
- Two tasks can be **running concurrently** with links between them.
- The **dash links** indicate the **synchronization** and **message pass communication** between tasks such that one task can notify another task by a **signal of job completion** or other events.
- There may also be an **external event notification** such as **interrupt request** from any external event resource.

# Hierarchy and Concurrency in StateChart

- More **complicated logical relationships** can be represented by the combinations of logic AND (concurrency) and logic OR (selection) in a stateChart.

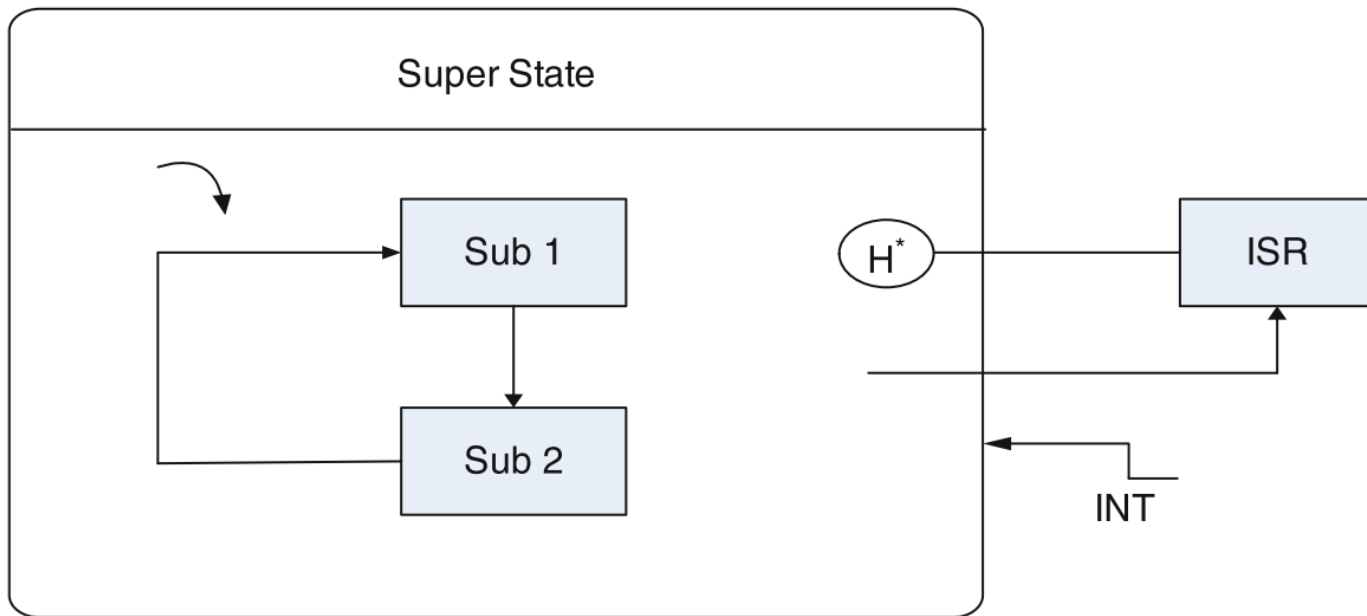


- state A is a super state
- states B , C are nested in state A.



# History state

- A **concurrent** state can **contain one history state** and one **deep history state**.
- A **transition to the history** state will **resumes** the state it last had in this state region.
- A typical **application** of the history state is for the Interrupt Service Routine(**ISR**) return, which returns the control to where it left.
- A history state can have **multiple incoming events/actions**.
- It can have **at most one outgoing events/actions**, which points to the “**resume**” state if the transition occurs for the first time without any history.
- The regular **history state** only **applies to** the **level** in which it **appears** while the **deep history state** entrance **applies to** the **most recently visited state** in the **deepest level**.
- It is noted as **H\***.



- After completion of interrupts the control will **return to where it left** as noted by the H states.
- **E.g.**, a timer can trigger a timeout event which interrupts a task, then an ISR handle the event, and the interrupted task resumes afterwards.

# FSM v.s. State chart

- The **FSM** is **good for control oriented system analysis** which monitors inputs, controls the transitions by stimulus/events, and takes actions on the states.
- The **stateChart** is **multi-tasking oriented** which overcomes these shortcoming to **support the real time multi-tasking** software analysis modeling.
- Instead of a single state machine, the **stateChart** can support **multiple state machines, communication, and synchronization** among these machines.

# FSM v.s. State chart

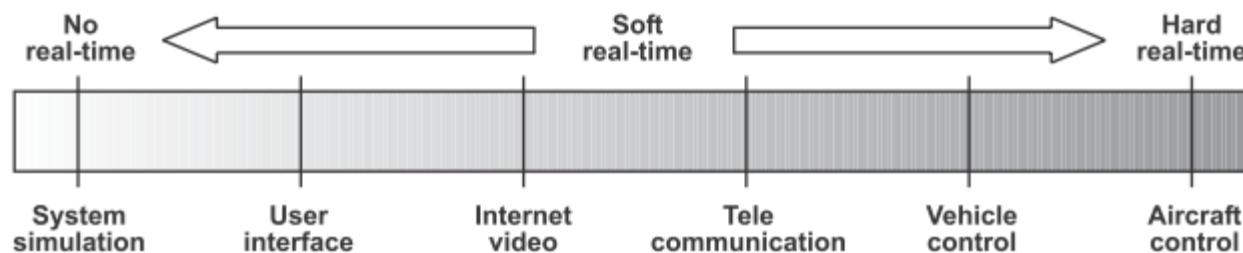
- **Similar to FSM**, the State Chart consists of **states** (including initial and final states), **signals**(events), **transitions** with triggers( event and data conditions), and **actions** (e.g., emitting signals).
- In addition, it **supports state hierarchy**, i.e., nested states, and various communication between two active states.
- When the State Chart is used, we can assign a **concurrent task** (a concurrent state) for each important device control so that we can easily **divide the whole software design into multiple partitions**.

# State chart simulation development tools

- There are **many** statechart simulation development **tools** available in market, StateMate, SpeedChart, StateFlow, BetterState, VisualVHDL, StateVision, and LabView are well known among these tools.
- Some other StateChart development tools such as Rational Rose, Rhapsody can even convert the stateChart into actual code in C or VHDL.
- In **summary**, the State chart provides a **formal model for reactive dynamic behavior of embedded system** especially for the concurrency behavior.
- It is a **good fit for large system modeling and design** because it has eliminated the state explosion problem.

# Time Requirement Analysis for Real-Time Systems

- Most embedded software are *real-time systems*, some waiting tasks are guaranteed to be given the CPU when an external event occurs.
- Real time systems are designed to control devices such as *industrial robots*, which require *timely processing*.
- Such system *must meet timing requirements* regardless hard real-time embedded system or soft real-time embedded systems.
- Almost all embedded software are *multitask oriented* in which multiple tasks, also known as *process*, share common processing *resources* such as a CPU.



timing criticality

# Contd.

- With a **single CPU**, **only one task** is said to be **running at** any point in **time**, meaning that the CPU is actively executing instructions for that task while **all others are waiting**.
- The act of reassigning a CPU from one task to another one is called a **context switch**.
- **Running multiple tasks concurrently** can make maximum use of CPU time, **reduce the CPU idle time** to a minimum degree, and the urgent requests can be handled immediately.
- **A task represents an activity such as a process or a thread**.
- **Threads are lightweight processes** which **share entire memory** space.

## Contd.

- Thus, **threads** are basically processes that **run in the same memory context**.
- **Context switch** between threads does **not** involve **changing the memory context**.
- A **task can even have multiple threads**. As a notation, the term *task* will represent both process and thread.
- The **multi-tasking** design and implementation **can separate the design concerns, divide and conquer the problem domains easier but interaction and resource sharing** between tasks still bring many design and implementation complexities.
- Each task must **keep tracks on its own concern** and has its own state including its own **status, CPU register status, program count, its own memory space and stack**, so that CPU can switch back and forth between these processes/tasks.

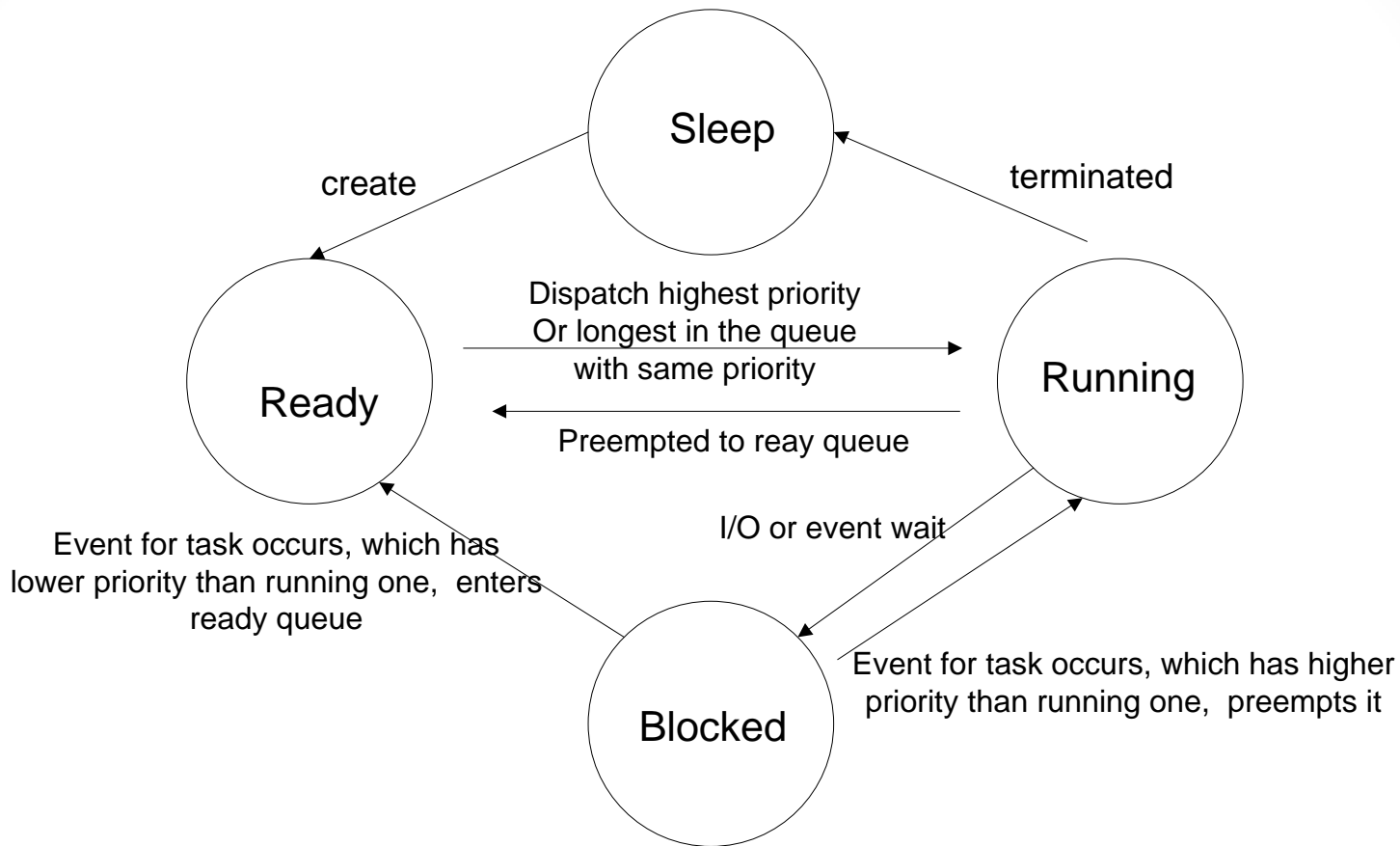


# What is the timing requirement?

- When an **event request occurs** in a reactive embedded system the **target task must respond** the event **within a period time**.
- The Worst-Case Execution Time (**WCET**) of a task tells the **maximum length of time the task could take to execute**.
  - A task often shows a certain **variation of execution times** depending on the input data or behavior of the environment.
  - The **actual response time may be shorter** that the WCET.
  - **Knowing worst-case execution times** is of prime **importance** for the **timing analysis** of hard real time system.
  - There is also a **deadline** for each task to complete for either periodic events or aperiodic events.
  - Understanding **life cycle of a task** is very important for the time requirement analysis.

# Example multi-tasking states

- An **example** is the multi-tasking 8051 RTX-51.
- It recognizes **four task states**; a **process is always in one of these states at any time**:
  1. **READY** Tasks which are READY to run CPU are in this state.
  2. **RUNNING (ACTIVE)** Task which is currently being executed by CPU processor. Only one task can be in this state at any time.
  3. **BLOCKED (WAITING)** Task waits for an event or resource.
  4. **SLEEPING** Tasks before started or after terminated are in this state.



## RTX51 Tasks Status Diagram

# Tasks Status Diagram

- In the life cycle of a task in a multitasking environment, a **task moves from one state to another** state and **complete CPU** with other state.
- The **task selection** for CPU (running state) is **determined by a scheduler** (also called **dispatcher**).
  1. The task with the **highest priority** of all tasks in the READY state is executed.
  2. If several tasks of the **same priority** are in the READY state, the task that has been **ready the longest** will be the next to execute.
  3. **Task switch** is only executed if the first rule would have been otherwise violated (exception: round-robin scheduling).

# Non-Preemptive Scheduling

- Non-preemptive multitasking is a **simple task scheduling for periodic time requirement system**.
- Such systems are either **statically scheduled** which most often are **periodic systems**, or in form of **cooperative multitasking**, in which case the tasks can **self-interrupt** and **voluntarily** give control to other tasks.
- **Cooperative** multitasking is a type of multitasking in which the **process** currently controlling the CPU must **offer control to other processes**.
- It is called “cooperative” because all **tasks must work cooperatively**.
- In the **non-preemptive scheduling**, once a task started it will **hold CPU until it completes** or it may **give up CPU by itself** due to the lack of resources.

## Contd.

- It does **not need any context switch** from a running task to another task.
- The **context switch takes a lot of time** to save its current status including its stack and Program Count(PC) address so that it can resume its task after it comes back to run the CPU.
- A **simplified modeling** for a non-preemptive scheduling can be a **cyclic scheduling** that the tasks can be scheduled in a **fixed static table off-line**.
- Assume that the embedded software consists a set of tasks and all of them have their fixed period. A **periodic events** can be **estimated by** their **worst case interval gap** between two consecutive task events.
- Also, **assume** that all tasks are **independent** and WCET are known in advance so that the **deadline** is at the **end of WCET**.

## Contd.

- You can also **schedule the higher priority task before** the tasks with **lower** priority.
- The **advantage** of such cyclic scheduling: **simple, zero overhead** of context switch between process/task;
- The **disadvantage: inflexibility.**
- If there are  $n$  tasks and the WCET of  $i^{\text{th}}$  task is  $c_i$ , then the period of any task  **$T_i$  must satisfy**

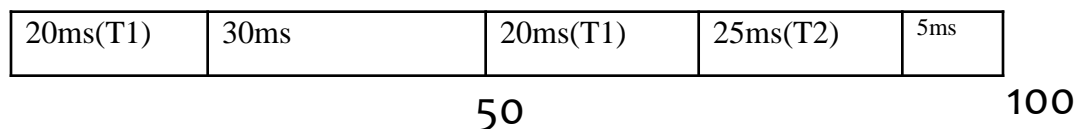
$$T_i \geq \sum c_j (j = 1, 2, \dots, n)$$

- **Otherwise** some task will **miss its deadline.**

- software analyst and design should **make each task as shorter as possible**.
- For these **long tasks** you may need to **break it into many small pieces**.
- Here is a cyclic schedule **example**:

Task	Execution Time	period
Task1	20ms	50ms
Task2	25ms	100ms

You can find that the period of T1 (50ms)  $>$   $c_1(20ms) + c_2(25ms)$ .





# non-preemptive cyclic scheduling implementation

The template for the non-preemptive cyclic scheduling implementation. Assume there is a timer set on the period of 50 ms.

```
while(1)
{
    Wait_for_50ms_timer_interrupt;
    do task1;
    Wait_for_50ms_timer_interrupt;
    do task1;
    do task2;
}
```

# Round-robin (RR)

- Round-robin (RR) is one of the **simplest time shared cooperative scheduling algorithms** which assigns **prefixed time slices to each process in order**, handling all processes without priority (**equal priority**).
- The **task with CPU** will **either voluntarily yield the CPU to other task or be preempted through an internal timer interrupt**.
- Round-robin scheduling is both **simple and easy to implement**.
- **Many RTOS support RR scheduling**.

# Example of RR Implementation

- Round robin scheduling **Example** in **RTX51 RTOS**
- Two counting tasks: job0 and job1

```
int c1, c2;
//Round robin scheduling
void job0(void) _task_ 0 //starting task by default
{
    os_create (1); //make task 1 ready
    while(1)
        c1 ++;
}
void job1(void) _task_ 1
{
    while(1)
        c2 ++;
}
```

RTOS will switch CPU between task0 and task1 after task0 starts.

# Pre-emptive Scheduling

- In reality, a reactive embedded system must respond and handle external or internal events with different urgent extents.
- Some events have hard time constraints while the others may only have soft time constraints.
- In other word, some task should be assigned higher priority than other tasks.
- An important design aspect of real time software systems is priority-based scheduling to ensure that the critical timing constructs such as deadline and response time must be met.

# Priority-based scheduling

- In priority-based scheduling, **algorithms assign priority to each process** in the multi-tasking real time systems.
- **CPU always goes to the highest priority** process which is ready.
- Popular **fixed priority scheduling** algorithms are:
  - Static timing scheduling
  - Round-robin scheduling
  - Rate Monotonic Scheduling(RMS) and its deadline based analysis: Rate Monotonic Analysis(RMA)
- The popular **dynamic priority-based scheduling is**
  - Earliest Deadline First(EDF) scheduling assigns priorities at runtime, based on upcoming execution deadline where the priority is time varying rather than fixed.

# RMS

- Rate Monotonic Scheduling (**RMS**) is a priority based static scheduling for **preemptive real time systems**.
- It is a **popular static scheduling algorithm** for such system where the round-robin scheduling analysis failed to meet task deadlines all the time.
- It can **guarantee the time requirement and maximize the schedulability** as long as the CPU utilization is **below 0.7**.
- It has been proved that the **RMA is optimal** among all static priority scheduling algorithms

## Contd.

- The rate-monotonic analysis Static priorities **assigns shorter period/deadline process higher priority** at design time **assumed** that processes have the **following properties**:
  - **No resource sharing** (processes do not share resources, e.g., a hardware resource, a queue, or any kind of semaphore blocking or busy wait non-blocking)
  - Deterministic **deadlines are exactly equal to periods**
  - **Context switch times** are free and **have no impact** on the model
  - Once **the priority of a task is** assigned they will remain **constant for the life time of the task.**

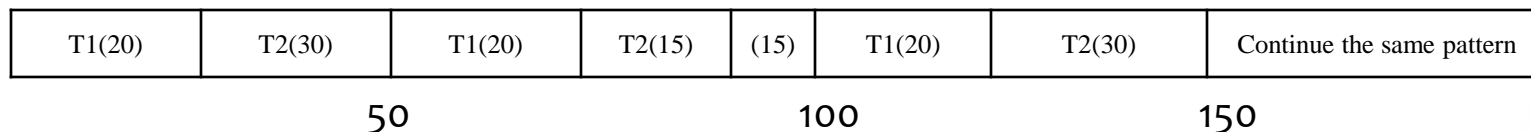
# RMS scenario

- Scenario with two tasks.

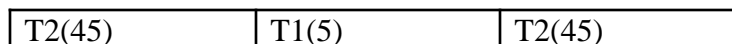
Task	Execution time	Period(Deadline)	Priority	Utilization
Task1	20ms	50ms	2(high)	40%
Task2	45ms	100ms	1(low)	45%

- The task1 must meet its deadline of 50, 100, 150, ... and task 2 must meet its deadline at 100, 200, 300, ... . Because task1 has shorter period therefore is assigned higher priority that once it starts it will not be preempted until it competes.

The **static schedule** is shown as follows.



- At time 50, task 2 is preempted by task1 because task1 is ready every 50ms and task1 has higher priority than the task 2.
- This schedule guarantees all tasks meet their deadline.
- If task 2 gets higher priority over task1 then task 1 will miss its deadline at time 50 as follows.



50



# Contd...

- An example of three tasks (processes) :

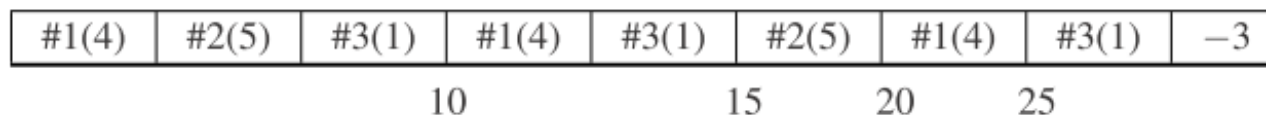
Task	Execution time	Period(Deadline)	Priority	Utilization
Task1	4ms	10ms	3(high)	40%
Task2	5ms	15ms	2(medium)	33%
Task3	6ms	25ms	1(low)	24%

Because  $T(1) < T(2) < T(3)$  ( where  $T(i)$  is the period of task  $i$  ),  
 Therefore  $P(1) > P(2) > P(3)$  (where  $P(i)$  is the priority of task  $i$ ).

Also notice that **CPU utilization** is

$$4/10 + 5/15 + 6/25 = 0.40 + 0.33 + 0.24 = 0.97 < 100\%.$$

Let's schedule these three tasks according to their priorities:



## Contd...

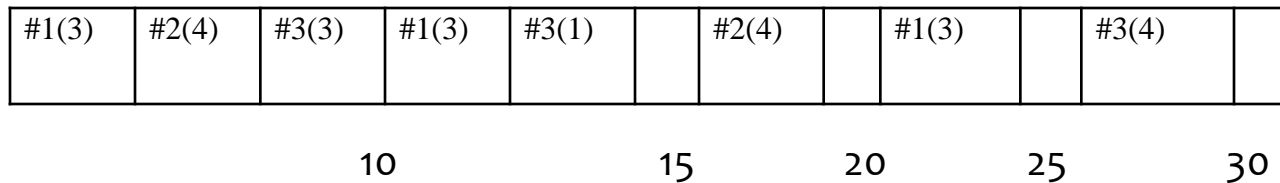
- What happens?
- At the time of 25ms the task<sub>3</sub> miss its deadline by 3ms.
- Why? It is due to that total utilization rate is 94% which is way beyond schedulable bound 70%.
- One major limitation of fixed-priority scheduling is that it is not always possible to fully utilize the CPU.
- It has a worst-case schedulable bound of:
$$U_n = \sum_{i=1..n} C_i/T_i \leq n * (2^{1/n} - 1)$$
where n is the number of tasks in a system.
- As the number of tasks increases, the schedulable bound decreases, eventually approaching its limit of  $\ln 2 = 69.3\%$ .
- Liu & Layland (1973) proved that for a set of n periodic tasks with unique periods,  
a feasible schedule that will meet deadlines exists if the CPU utilization is  $< U_n$ .

# Contd.

- Let's reduce the utilization rates as follows:

Task	Execution time	Period(Deadline)	Priority	Utilization
Task1	3ms	10ms	3(high)	20%
Task2	4ms	15ms	2(mediaum)	20%
Task3	4ms	25ms	1(low)	16%

The total CPU utilization is 56%. let's re-schedule it.



The same pattern will continue after time 30.

## Contd.

- In general **RMS** case can **meet** all the **deadlines** if CPU utilization is 70%.
- The **other 30%** of the CPU can be dedicated to **lower-priority non real-time tasks**.
- The **context switch cost** for the RMS is very high although its **CPU utilization is not perfect**.
- In order to **fully** make use of CPU time and also to meet all deadlines is to **use a dynamic priority scheduling algorithm**.

# Dynamic scheduling with EDF

- The priority is fixed in the static priority-based scheduling but **the priority of a process can change** in the dynamic priority-based scheduling **to increase the CPU utilization and to make all tasks meet their deadline.**
- The Earliest deadline First(EDF) **assigns higher priority to these tasks that are closer to their deadline at run time.**
- The **process closest to its deadline is assigned with the highest priority.**
- The EDF can be **applied to** scheduling for **both periodic and aperiodic tasks** if deadlines are known in advance.
- The **advantage of it is its CPU utilization** but its **disadvantages including high costs on context switches and no guarantees on all deadline requirements.**

## Contd.

- EDF is **not very practical** in many cases **due to its complexity** and therefore is **not as popular as RMS**.
- EDF **must recalculate** the priority of each process at **every context switch time (preemption time)**. This is **another overhead cost addition** to the cost of context switches.
- Let's explore a multi-tasking case as follows:

Tasks	Execution time( $C_i$ )	Deadline(period)	Utilization
Task1	4ms	10ms	40%
Task2	4ms	15ms	27%
Task3	5ms	25ms	20%

## Contd...

- The total CPU utilization = 87%

T1(4)	T2(4)	T3(2)	T1(4)	T3(1)	T2(4)	T3(1)	T1(4)	T3(1)	T3(5)	
0	4	8	10	14	15	19	20	24	25	30

- Assume tasks t1, t2, t3 are ready at time 0.
- The deadline of t2 are at the time of 10, 20, 30, 40, ..., the deadlines for t2 are at the time of 15, 30, 45,, and the deadlines for t3 are at the time of 25, 50, 75, ....
- At time 0, t1 has the highest priority because its next deadline is shorter than the other two.
- At time 4 context switch time, t1 is just finished and is not ready, t2 is closer to its next deadline than t3, t2 goes first.
- At time 8, only t3 is available so that t3 gets its 2 units until time 10 when the t1 is ready again.
- At time 10, t1 gets back CPU and completes at time 14.
- At time 14, only t3 is ready and t2 will be ready at time 15 therefore t3 gets another 1 unit CPU time.

# Contd.

T1(4)	T2(4)	T3(2)	T1(4)	T3(1)	T2(4)	T3(1)	T1(4)	T3(1)	T3(5)	
0	4	8	10	14	15	19	20	24	25	30

- At time 15, t2 must run CPU and completes at time 19. T3 gets another one unit CPU time and completes before its deadline of time 25.
- At time 20, t1 gets CPU and run 4 unit time, t3 gets another one unit time to complete its execution time.
- At time 25, neither t1 nor t2 is ready but t3 is ready for next round therefore t3 runs 5 unit time.
- At the time 30, the cycle will start over again.
- Compared to RMS you can see **much more frequent context switches** take place during the concurrent executions



# QUIZ

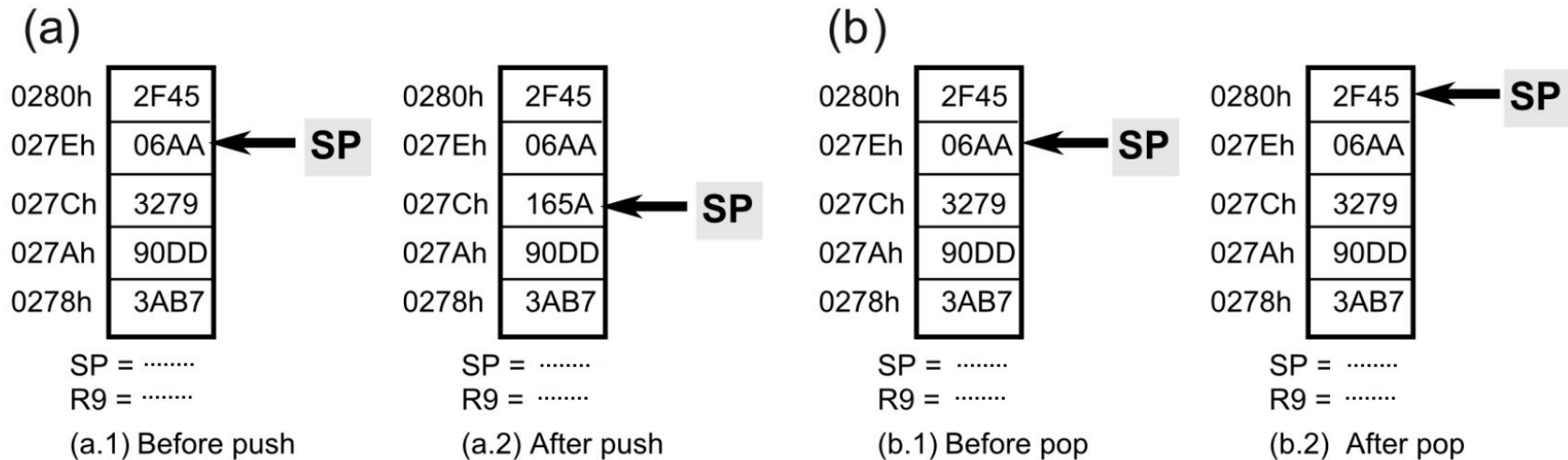
( 57 )



ESTABLISHED 2000

# Quiz (3 marks)

1. What's Power-on Reset? Mention the other Reset triggers.
2. Figure is Pushing R9 in (a) then popping it directly in (b), Write the SP, R9 values in each step below.



- For more details, refer to:
  - Chapter 2 at **Embedded Software Development with C**, Springer 2009 by Kai Qian et al.
  - Chapter 9 at **Introduction to Embedded Systems**, Springer 2014 by Manuel Jiménez et al.
- The lecture is available online at:
  - <http://bu.edu.eg/staff/ahmad.elbanna-courses>
- For inquiries, send to:
  - [ahmad.elbanna@feng.bu.edu.eg](mailto:ahmad.elbanna@feng.bu.edu.eg)