

Lecture 1

Key software development phases for embedded systems

- Problem definition
- Develop an architecture
- Develop a design
- Implementation (Component-based)
- Verification and Validation

What is an Embedded System?

A combination of hardware and software components to form a computational engine that will perform a specific function.

Perform in reactive and time-constrained environments

Responds to the environment via sensors and controls the environment using actuators

Realtime

Multi-rate - can handle multiple processing rates

Typical Embedded System consists of:

Sensors and actuators

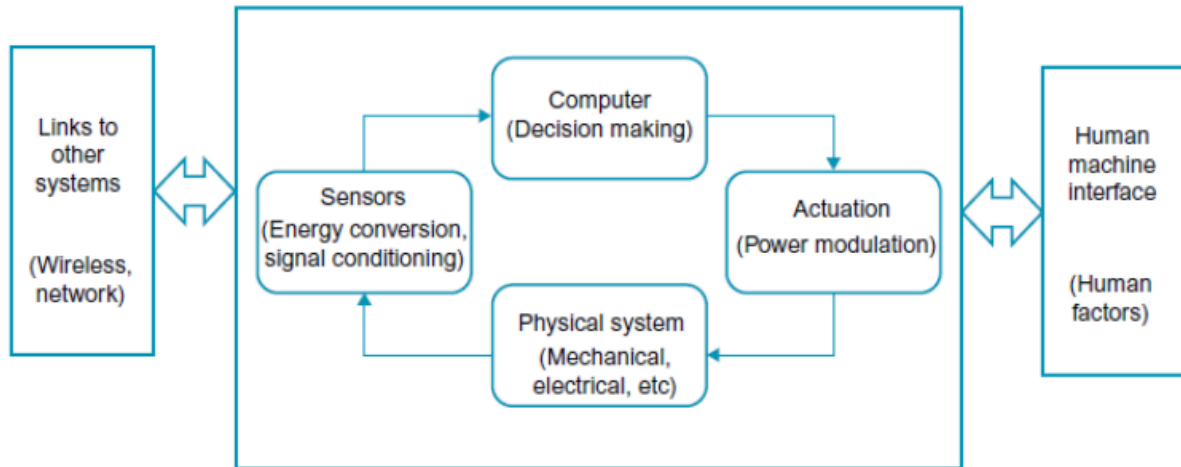
Processor cores and Memory

Emulation and diagnostics and User Interface

Application-specific gates and Analog I/O

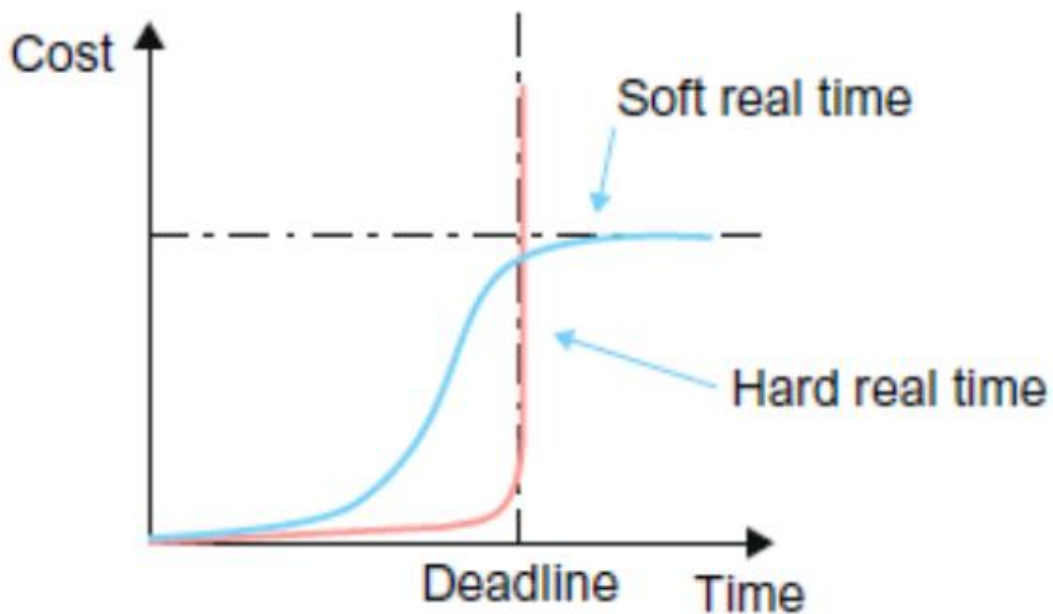
Software/Firmware and Power and Cooling

Abstract Model of an Embedded System



Other systems -> Sensors -> Computer -> Actuation -> Physical system -> Human Machine Interface

Hard vs Soft Real time



Examples of hard and soft real time systems

System type	Hard or soft real time?
Traffic light control	Hard RT – Critical
Automated teller machine	Soft RT – Non-Critical
Controller for radiation therapy machine	Hard RT – Critical
Car simulator for driver training	Hard RT – Non Critical
Highway car counter	Soft RT – Non-Critical
Missile control	Hard RT – Critical
Video games	Hard RT – Non Critical
Network chat	Soft RT – Non-Critical

Real-time vs Time-shared

Characteristic	Time-Shared Systems	Real-Time Systems
System capacity	High throughput	Schedulability and the ability of system tasks to meet all deadlines
Responsiveness	Fast average response time	Ensured worst case latency which is the worst-case response time to events
Overload	Fairness to all	Stability; when the system is overloaded important tasks must meet deadlines while others may be starved

Describe by sample applications the difference between the following three types of events: synchronous, asynchronous, and isochronous events

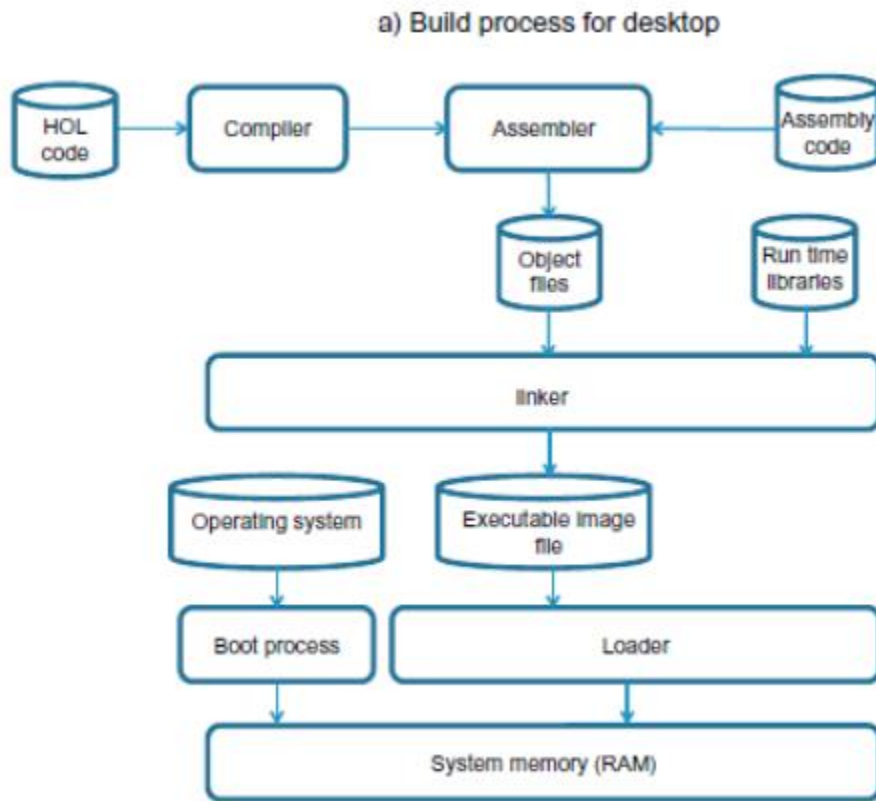
Asynchronous events: entirely unpredictable, e.g. cell phone call arriving at a cellular base station

Synchronous events predictable events and occur with precise regularity, e.g. video streaming

Isochronous events occur with regularity within a given time window, e.g. audio data in a networked multimedia application must appear within a window of time when the corresponding video stream arrives

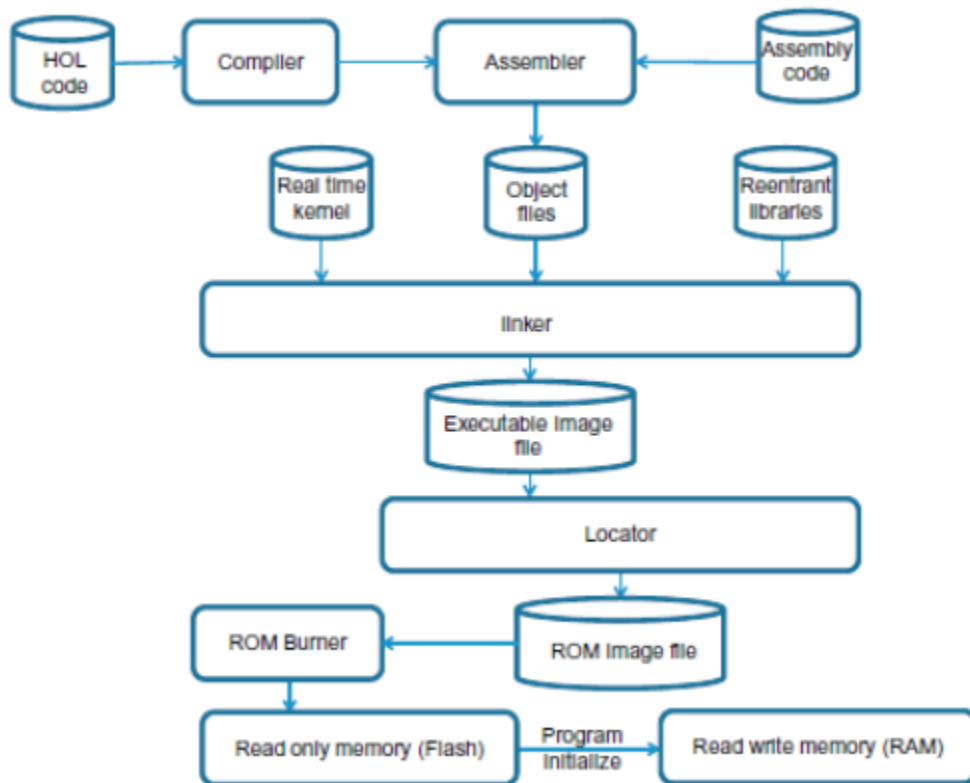
Lecture 2

Desktop System Build Process



Higher Order Logic (HOL) code -> Compiler -> Assembler with Assembly code -> Object files with run time libraries -> Linker -> Executable Image file -> Loader -> System Memory (RAM) with Operating System and boot process.

Embedded System Build Process



HOL Code -> Compiler -> Assembler with Assembly code -> object files with real time kernel and reentrant libraries (like runtime libraries in desktop process) -> Linker -> Executable image file -> Locator -> ROM image file -> ROM Burner -> ROM (Flash) -> program initialize Read Write memory (RAM)

Super Loop Architecture

```
Function Main_Function()
{
    Initialization();
    Do_Forever
    {
        Check_Status_of_Task();
        Perform_Calculations();
        Output_Result();
    }
}
```

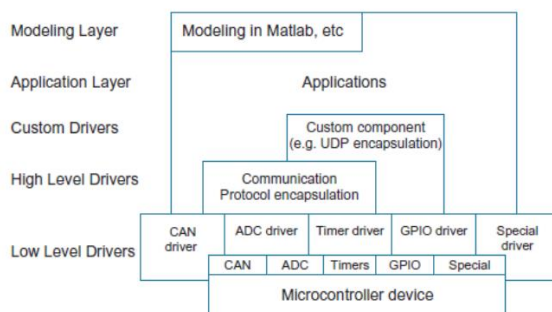
Power-Saving Super Loop Architecture

```
Function Main_Function()
{
    Initialization();
    Do_Forever
    {
        Check_Status_of_Task();
        Perform_Calculations();
        Output_Result();
        Delay_Before_Starting_Next_Loop();
    }
}
```

Hardware abstraction layer

provides an interface between hardware and software so applications can be device independent, also encapsulates peripherals of a microcontroller and several API implementations can be provided at different levels of abstraction

HAL for embedded systems

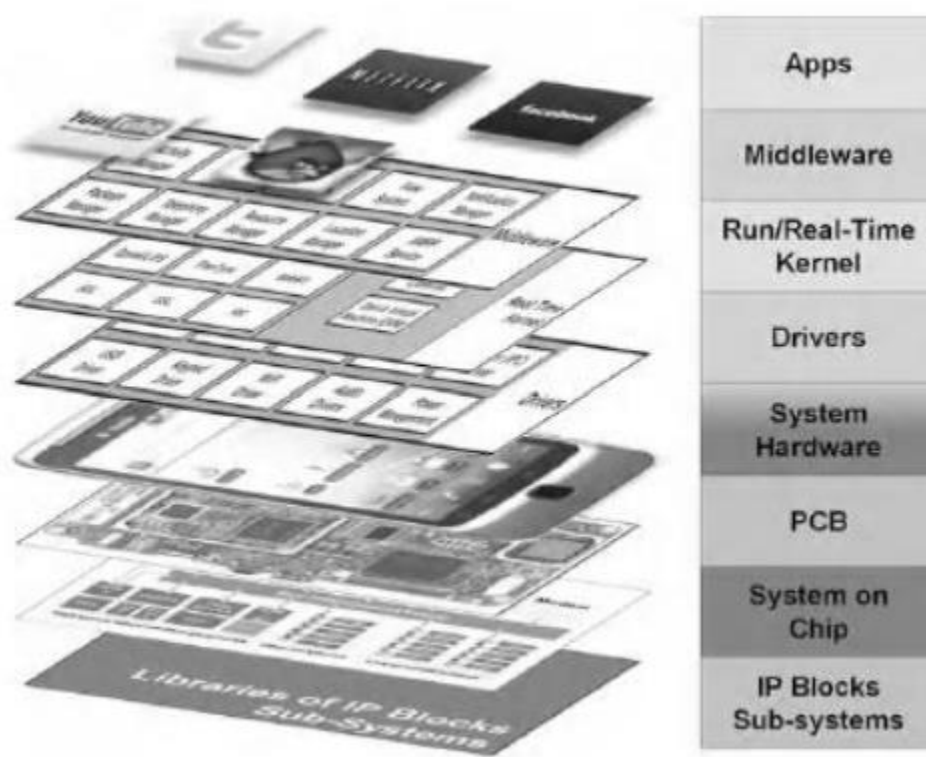


CAN: Controller Area Network protocol

ADC: Analog to digital convertor

GPIO: General Purpose input/output: *a signal pin on an integrated circuit or board that can be used to perform digital input or output functions*

Layers of HW/SW in embedded systems



IP -> Intellectual property (براءة اختراع)

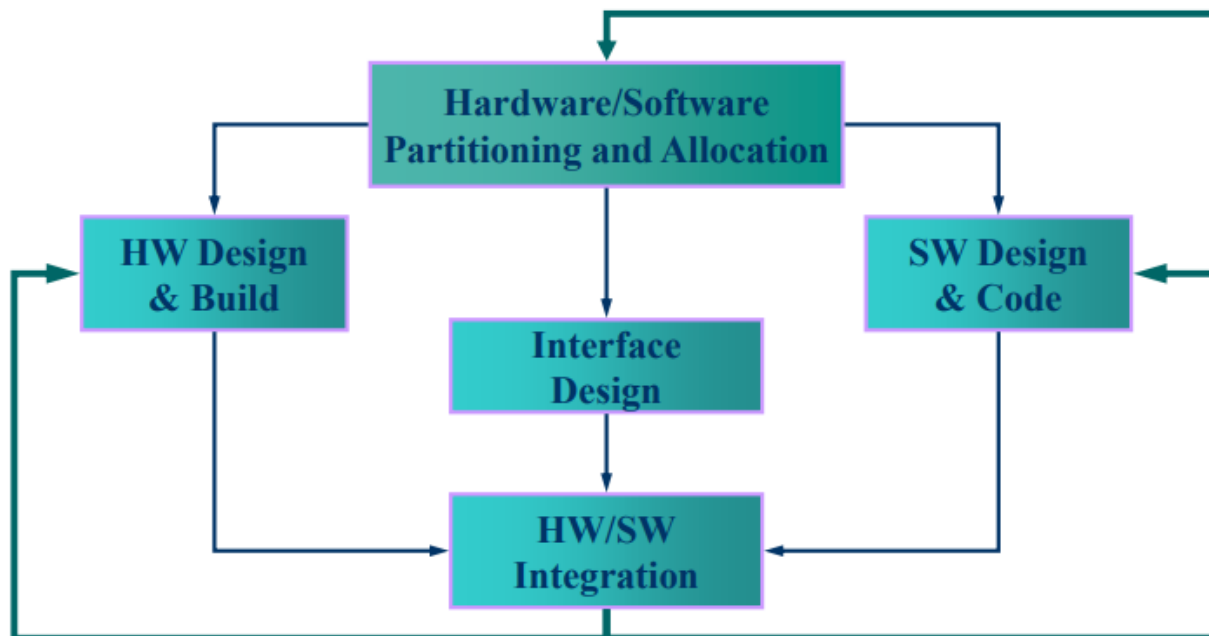
Bare Metal -> Has no operating system

Lecture 3

System Architecture

- Hardware
 - One micro-controller
 - ASICs application-specific integrated circuit
- Software
 - Set of concurrent tasks
 - Customized operating system (Real-Time scheduler)
- Interfaces
 - Hardware modules
 - Software I/O drivers (polling, interrupt handlers, ...)

Embedded System Design Traditional Methodology

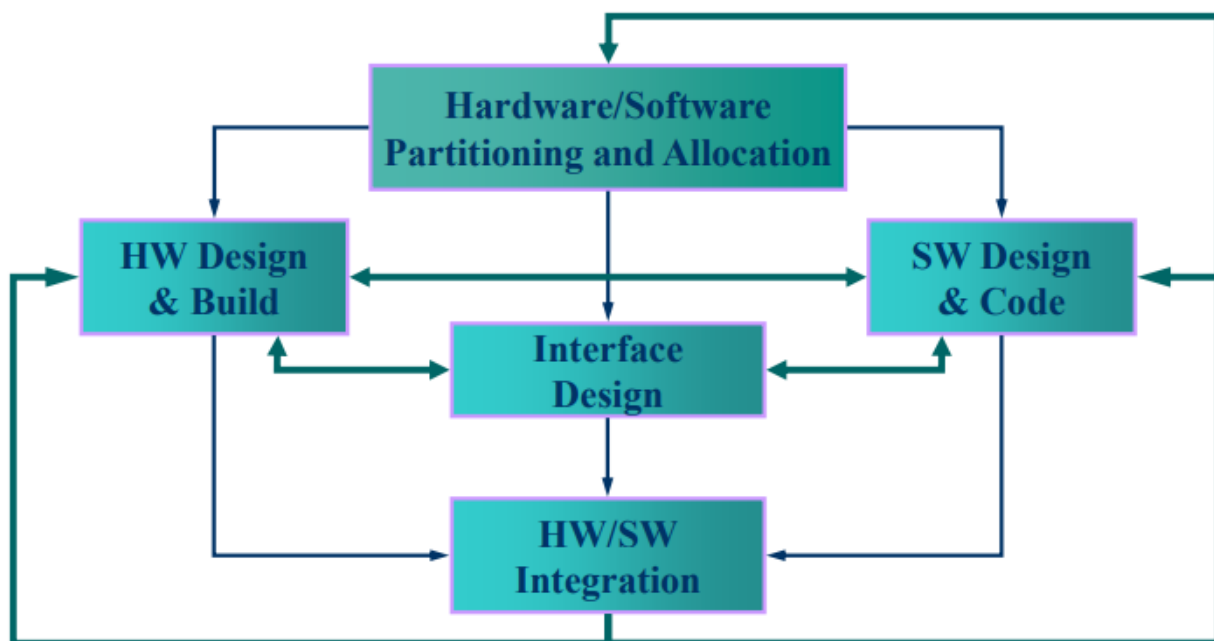


Hardware/Software partitioning and allocation -> HW Design & Build with SW Design and Code with Interface Design -> HW/SW Integration and going back if any modification.

Problems with Traditional Method

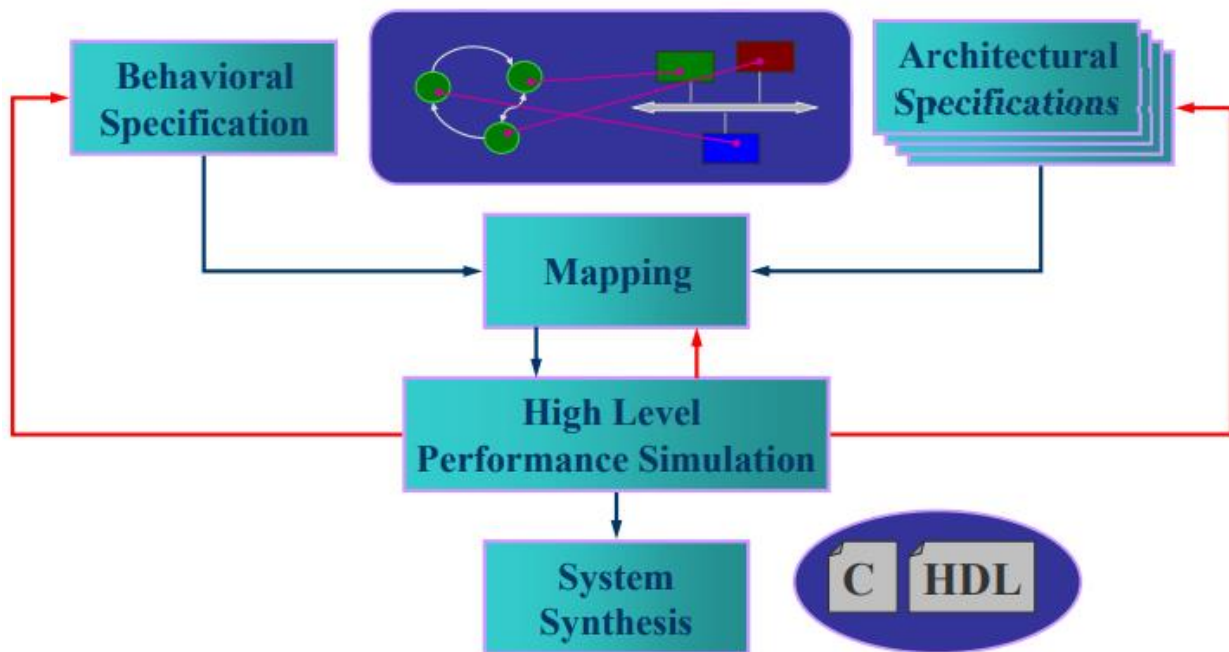
- Lack of unified system-level representation
 - Cannot verify the entire HW-SW system – Hard to find incompatibilities across HW-SW boundary (often found only when prototype is built)
- Architecture is defined a priori, based on expert evaluation of the functionality and constraints
- Lack of well-defined design flow
 - Time-to-market problems
 - Specification revision becomes difficult

Embedded System Design HW/SW Co-Design Methodology



Hardware/Software partitioning and allocation -> HW Design & Build with SW Design and Code with Interface Design, with the ability of returning back modifying any Design-> HW/SW Integration and going back if any modification.

Embedded System Design Behavior/Architecture Co-Design Methodology



Behavioral Specification and Architectural Specification -> Mapping -> High Level Performance Simulation with the ability of returning back modifying specification -> System Synthesis (بناء السیستم کھار دویر)

HDL: Hardware Description Language

Goals of the Design

- Clear separation between – behavior – architecture – communication
- Same framework for
 - specification and behavioral simulation
 - performance simulation
 - refinement to implementation
 - HW, SW and interface synthesis
 - rapid prototyping

Lecture 4

Hard real-time task

- one that must meet its deadline
- otherwise it will cause unacceptable damage or a fatal error to the system

Soft real-time task

- has an associated deadline that is desirable but not mandatory
- it still makes sense to schedule and complete the task even if it has passed its deadline

Periodic tasks

- requirement may be stated as:
- once per period T
- exactly T units apart

Aperiodic tasks

- has a deadline by which it must finish or start
- may have a constraint on both start and finish time

Characteristics of Real Time Systems

- Determinism

Concerned with how long an operating system delays before acknowledging an interrupt

- Responsiveness

Concerned with how long, after acknowledgment, it takes an operating system to service the interrupt

- User Control

It is essential to allow the user fine-grained control over task priority

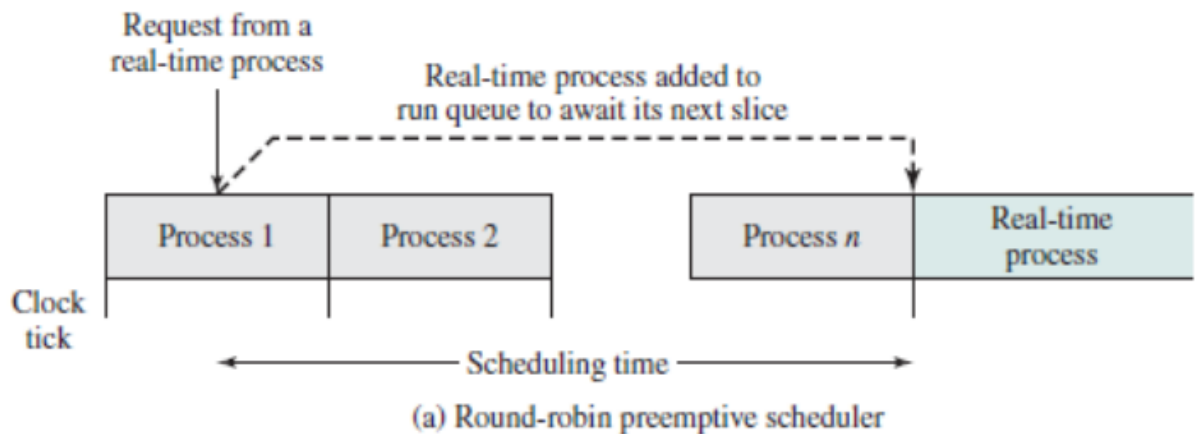
- Reliability

Real-time systems respond to and control events in real time so loss or degradation of performance may have catastrophic consequences

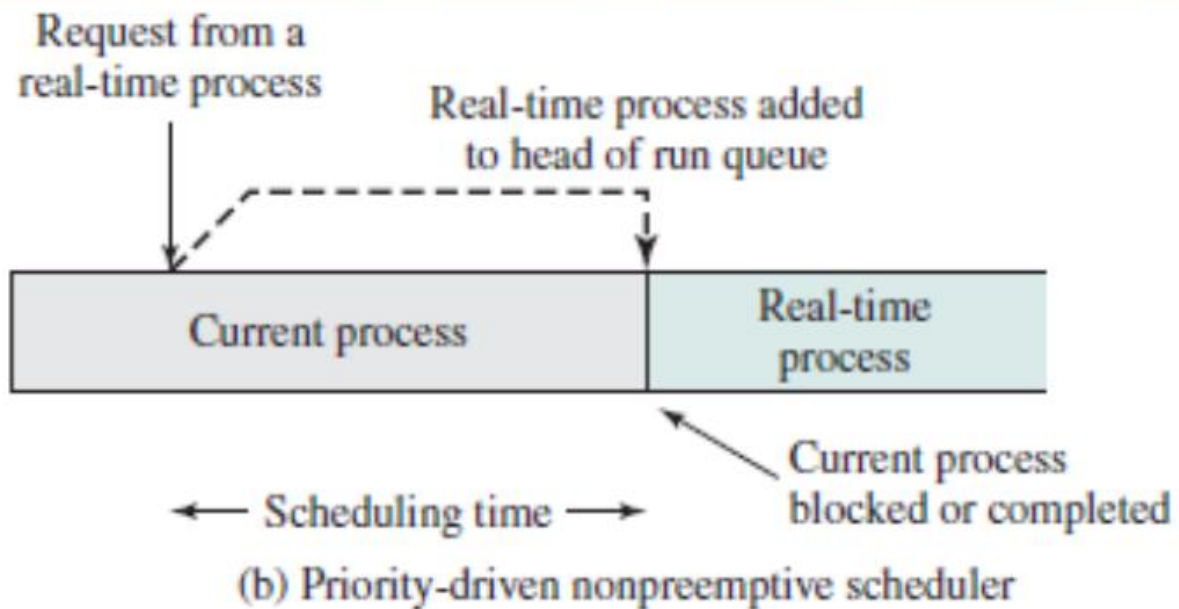
- Fail-Soft Operation

A characteristic that refers to the ability of a system to fail in such a way as to preserve as much capability and data as possible

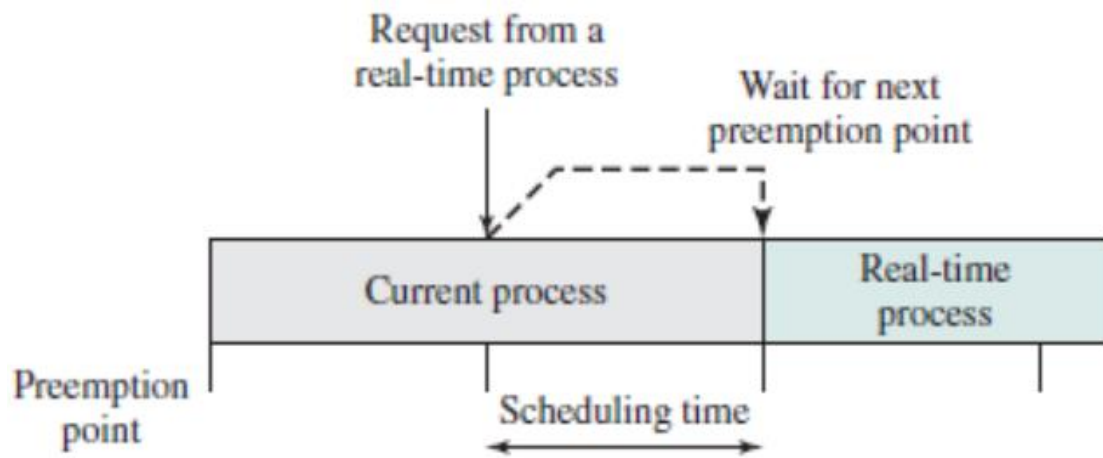
Round-robin Preemptive scheduler



Priority-Driven nonpreemptive scheduler

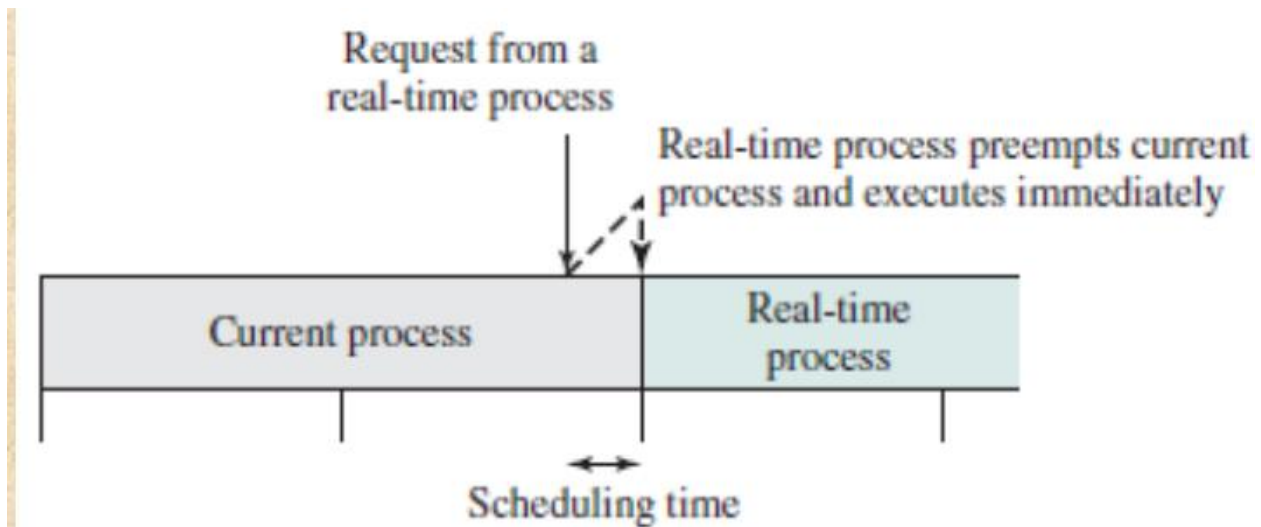


Priority-Driven preemptive scheduler on preemption points



(c) Priority-driven preemptive scheduler on preemption points

Immediate Preemptive scheduler



(d) Immediate preemptive scheduler

Classes of Real-Time Scheduling Algorithms

Static table-driven

performs a **static analysis of feasible schedules of dispatching** which result in a schedule that determines, at run time, when a task must begin execution, applicable to tasks that are **periodic**. This is a predictable approach but one that is **inflexible**, because any change to any task requirements requires that the schedule be **redone**. Example: **Earliest-deadline-first**

Static priority-driven preemptive

a **static analysis** is performed but **no schedule is drawn up**, analysis is **used to assign priorities to tasks** so that a traditional priority-driven preemptive scheduler can be used. One example of this approach is the **rate monotonic scheduling** algorithm, which assigns static priorities to tasks based on the length of their periods

Dynamic Planning-based

feasibility is determined at run time (dynamically) rather than offline prior to the start of execution (statically), an arriving task is **accepted** for execution only **if it is feasible to meet its time constraints**, one result of the analysis is a schedule or plan that is used to decide when to dispatch this task, an attempt is made to create a schedule that contains the previously scheduled tasks as well as the new arrival. If the **new arrival can be scheduled in such a way that its deadlines are satisfied and that no currently scheduled task misses a deadline**, then the schedule is revised to accommodate the new task

Dynamic Best Effort

no feasibility analysis is performed, when a task arrives, the system assigns a priority based on the characteristics of the task, **system tries to meet all deadlines and aborts any started process whose deadline is missed**. Typically, the tasks are aperiodic and so no static scheduling analysis is possible. With this type of scheduling, until a deadline arrives or until the task completes, **we do not know whether a timing constraint will be met**. This is the **major disadvantage** of this form of scheduling. Its advantage is that it is **easy to implement**

Information Used for Deadline Scheduling

Ready time

- time task becomes ready for execution

Resource requirements

- resources required by the task while it is executing

Starting deadline

- time task must begin

Priority

- measures relative importance of the task

Completion deadline

- time task must be completed

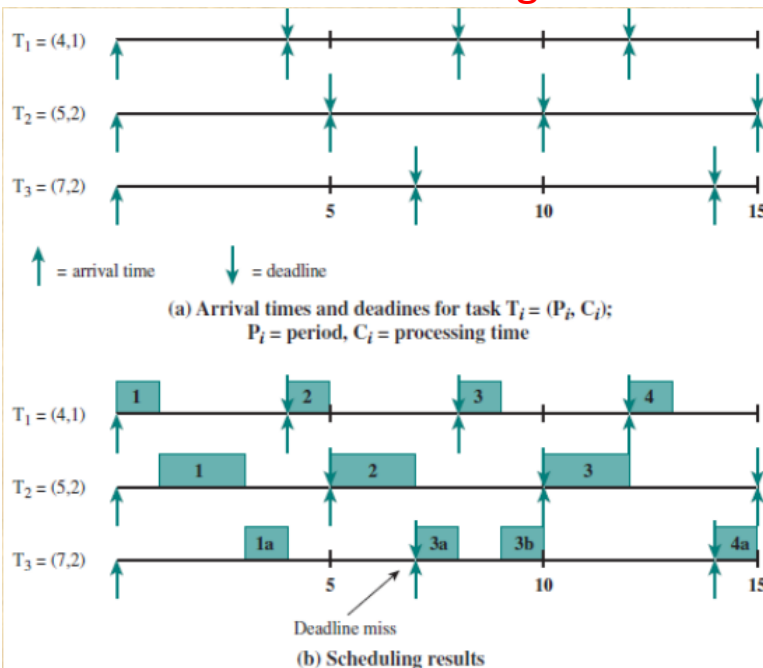
Subtask scheduler

- a task may be decomposed into a mandatory subtask and an optional subtask

Processing time

- time required to execute the task to completion

Rate Monotonic Scheduling



For Perfect Scheduling

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \dots + \frac{C_n}{T_n} \leq 1$$

For RMS only

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \dots + \frac{C_n}{T_n} \leq n(2^{1/n} - 1)$$

As an example, consider the case of three periodic tasks, where $U_i = C_i/T_i$:

- Task P₁: $C_1 = 20$; $T_1 = 100$; $U_1 = 0.2$
- Task P₂: $C_2 = 40$; $T_2 = 150$; $U_2 = 0.267$
- Task P₃: $C_3 = 100$; $T_3 = 350$; $U_3 = 0.286$

The total utilization of these three tasks is $0.2 + 0.267 + 0.286 = 0.753$. The upper bound for the schedulability of these three tasks using RMS is

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \frac{C_3}{T_3} \leq n(2^{1/3} - 1) = 0.779$$

Priority Inversion

Best-known instance involved the **Mars Pathfinder mission in 1997**, occurs when circumstances within the system force a **higher priority task to wait for a lower priority task**.

If a **lower-priority task has locked a resource** and a **higher-priority task attempts to lock that resource**, the higher-priority task will be put in a blocked state until the resource is available.

If the lower-priority task soon **finishes with the resource and releases it**, the **higher-priority task may quickly resume** and it is possible that no real-time constraints are violated.

Unbounded Priority Inversion

The duration of a priority inversion depends not only on the time required to handle a shared resource, but also on the unpredictable actions of other unrelated tasks

Priority Inheritance

A method is used to eliminate the problems of Priority inversion. With the help of this, a process-scheduling algorithm increases the priority of a process, to the maximum priority of any other process waiting for any resource

Priority Ceiling

A priority is associated with each resource.

The priority assigned to a resource is one level higher than the priority of its highest priority user.

The scheduler then dynamically assigns this priority to any task that accesses the resource.

Once the task finishes with the resource, its priority returns to normal

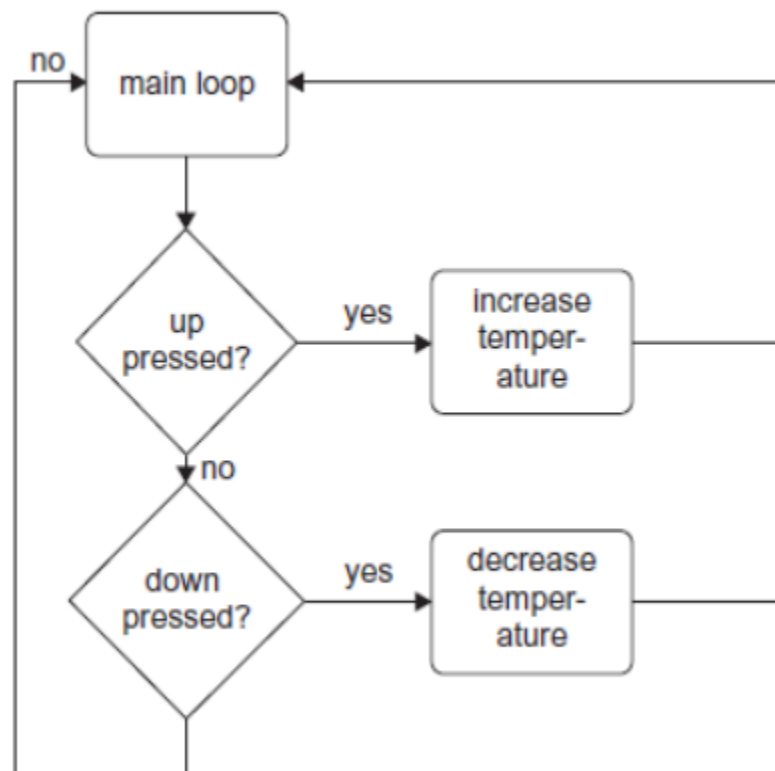
Lecture 5

Event: something that occurs at a time

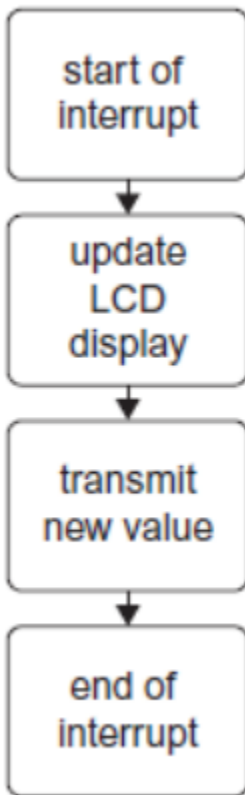
Trigger: When an event happens, the system should react in a timely manner.

Not only has the system to produce the correct result, it has to produce the **correct result** at the **right time**.

Example: Air Conditioning System



AC System Interrupt Service Routine



this approach is simple, but has some problems

Updating the display and transmitting the new value might take some time

All other interrupts might be holding off during the interrupt execution

It violates a fundamental design rule for ISRs:

1. Keep interrupt handlers as small and fast as possible.
2. Only do things in the interrupt handler which cannot wait

Event System Design, ISR System

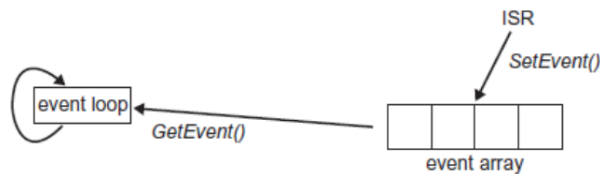


Figure 5.3:
Event ISR system.

Event Polling System

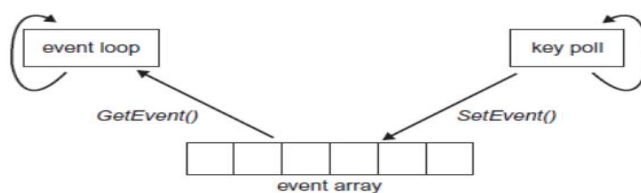


Figure 5.4:
Event polling system.

Event Module Design

Static number of events: number and kind of events are **known** at **compilation time**.

Singularity: an event of any kind can only **exist once**

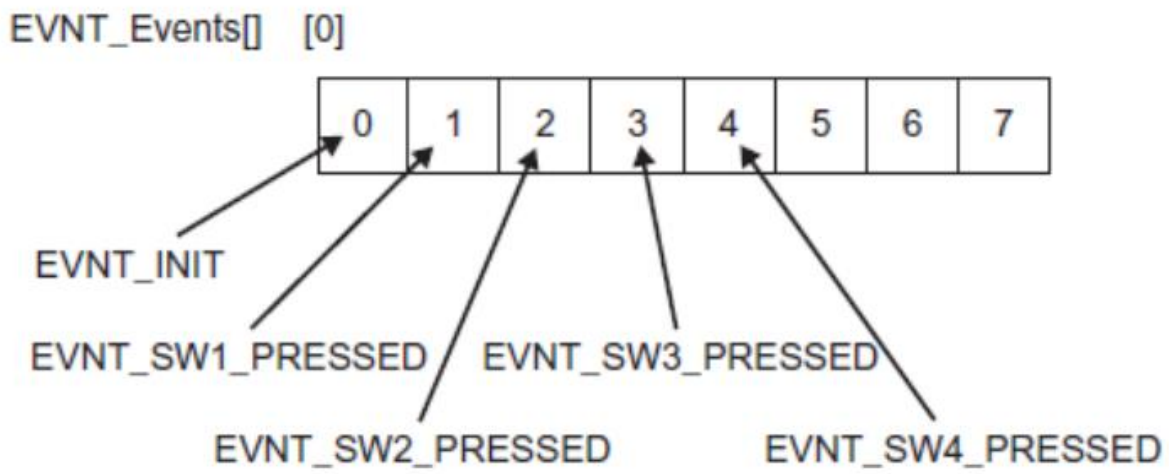
Static memory: a **static array** of events is used to hold each event handle.

Event handle: each event owns an event handle or identifier. This event handle is an **index into the event array**.

Event Types

```
typedef enum {  
  
    EVNT_INIT, /*!< System Initialization Event */  
  
    EVNT_SW1_PRESSED, /*!< SW1 pressed */  
  
    EVNT_SW2_PRESSED, /*!< SW2 pressed */  
  
    EVNT_SW3_PRESSED, /*!< SW3 pressed */  
  
    EVNT_SW4_PRESSED, /*!< SW4 pressed */  
  
    EVNT_NOF_EVENTS, /*!< Must be last one! */  
  
} EVNT_Handle;
```

Event Array



Event Methods

- `void EVNT_SetEvent(EVNT_Handle event);`
- `void EVNT_ClearEvent(EVNT_Handle event);`
- `bool EVNT_GetEvent(EVNT_Handle event);`
- `void EVNT_HandleEvent(void(*callback)(EVNT_Handle));`
- `void EVNT_Init(void);`