

# MODULE I

## FUNDAMENTALS OF EMBEDDED SYSTEM

# Syllabus

- Complex systems and microprocessors
- Embedded system design process
  - Specifications
  - Architecture design of embedded system
  - Design of hardware and software components
- Structural and behavioral description.

# System

- A system is a way of working, organizing or doing one or many tasks according to a fixed plan, program or set of rules.
- A system is also an arrangement in which all its units assemble and work together according to the plan or program.

# What is an Embedded system?

- A system that has **hardware with software embedded** in it as one of its most important component and **intended for a dedicated application.**
- A system that has embedded software and computer hardware, which makes it a system dedicated for an application(s) or specific part of an application or product or part of a larger system.

5

- “A combination of computer hardware and software, and perhaps additional mechanical or other parts, designed to perform a dedicated function. In some cases, embedded systems are part of a larger system or product, as is the case of an antilock braking system in a car ”.
- Embedded systems are electronic systems that contains microprocessor or microcontroller, but not computers

# Definition

- **Embedded computing system:** any device that includes a programmable computer but is not itself a general-purpose computer.
- Take advantage of application characteristics to optimize the design:
  - don't need all the general-purpose bells and whistles.

## Why do we need embedded systems?

- It is because general-purpose computers, like PCs, would be far too costly for the majority of products that incorporate some form of embedded system technology (Christoffer, 2006).
- Is because general-purpose solution might also fail to meet a number of functional or performance requirements such as constraints in power-consumption, size-limitations, reliability or real-time performance etc.
- The colossal growth of processing power in small packages has fuelled the digital revolution. All sectors of the economy have been influenced by the digital revolution and the industry has experienced tremendous developments in all aspects of engineering disciplines (Bruce, 2011).

Embedded systems span all aspects of modern life and there are many examples of their use.

- a) Biomedical Instrumentation – ECG Recorder, Blood cell recorder, patient monitor system, scanners
- b) Communication systems – pagers, cellular phones, cable TV terminals, fax and trans receivers, video games and so on.
- c) Peripheral controllers of a computer – Keyboard controller, DRAM controller, DMA controller, Printer controller, LAN controller, disk drive controller.

- d) Industrial Instrumentation – Process controller, DC motor controller, robotic systems, CNC machine controller, close loop engine controller, industrial moisture recorder cum controller.
- e) Scientific – digital storage system, CRT display controller, spectrum analyzer.
- f) Military – control and monitoring of military equipment's.
- g) Automobile controls – ABS, engine and transmission control, door and wiper control.

# Advantages

- Low cost
- Small size
- High reliability
- Fast operations
- Easy to manufacture
- Fewer interconnections

# A “short list” of Embedded Systems

Anti-lock brakes  
Auto-focus cameras  
Automatic teller machines  
Automatic toll systems  
Automatic transmission  
Avionic systems  
Battery chargers  
Camcorders  
Cell phones  
Cell-phone base stations  
Cordless phones  
Cruise control  
Digital cameras  
Disk drives  
Electronic card readers  
Electronic instruments  
Electronic toys/games  
Factory control  
Fax machines  
Fingerprint identifiers  
Home security systems  
Life-support systems  
Medical testing systems

Modems  
MPEG decoders  
Network cards  
Network switches/routers  
On-board navigation  
Pagers  
Photocopiers  
Point-of-sale systems  
Portable video games  
Printers  
Satellite phones  
Scanners  
Smart ovens/dishwashers  
Speech recognizers  
Stereo systems  
Teleconferencing systems  
Televisions



*and the list goes on  
and on...*

# Application Examples

## Toys



Product: Sony Aibo  
ERS-110 Robotic  
Dog.

Microprocessor: 64-bit MIPS RISC.



Product: Palm Vx  
handheld.

Microprocessor:  
32-bit Motorola  
Dragonball EZ.

# Application Examples

## Space



Product: NASA's Mars Sojourner Rover.

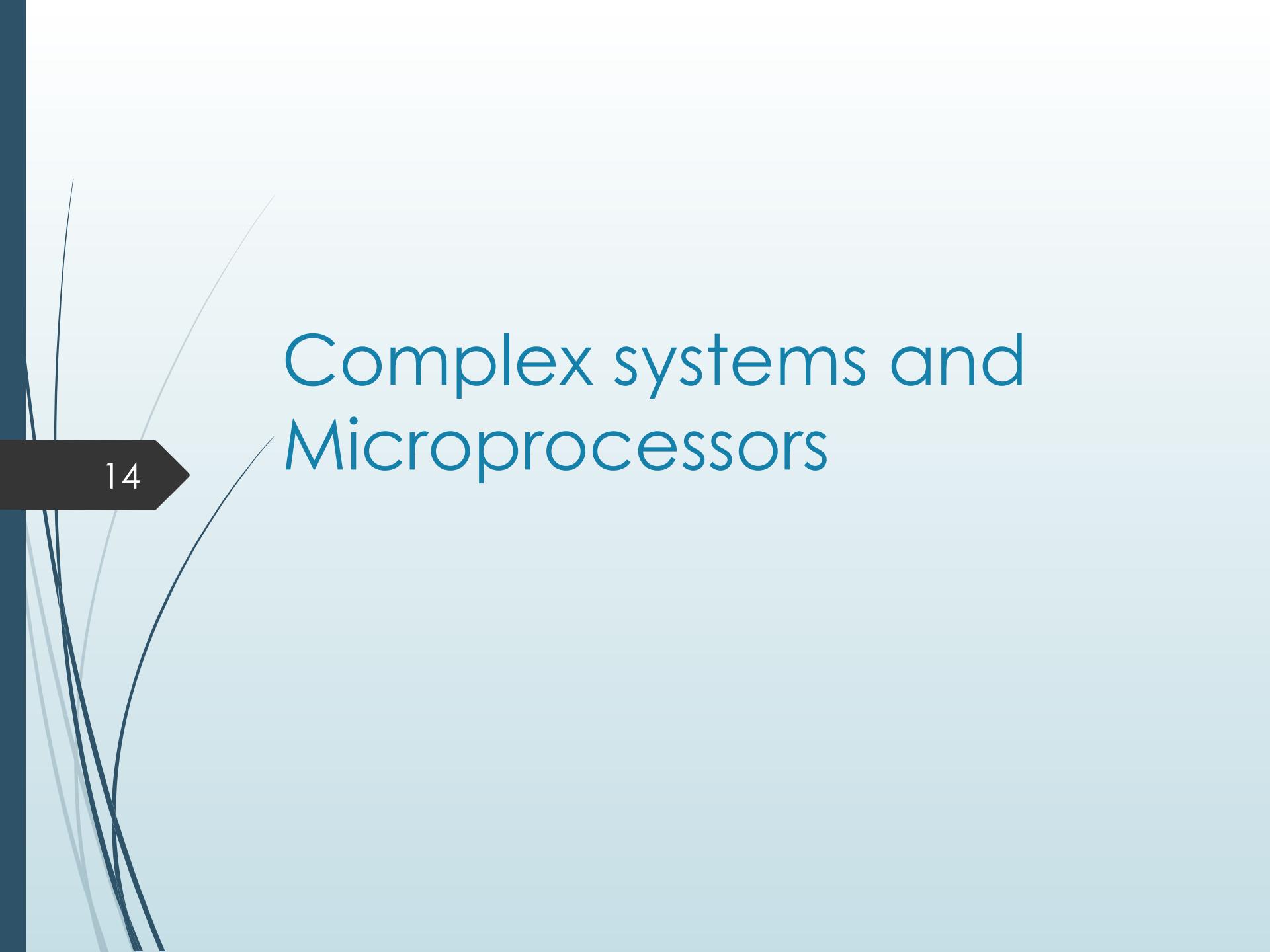
Microprocessor:  
8-bit Intel 80C85.

## Mobile



Product: Garmin StreetPilot GPS Receiver.

Microprocessor: 16-bit.



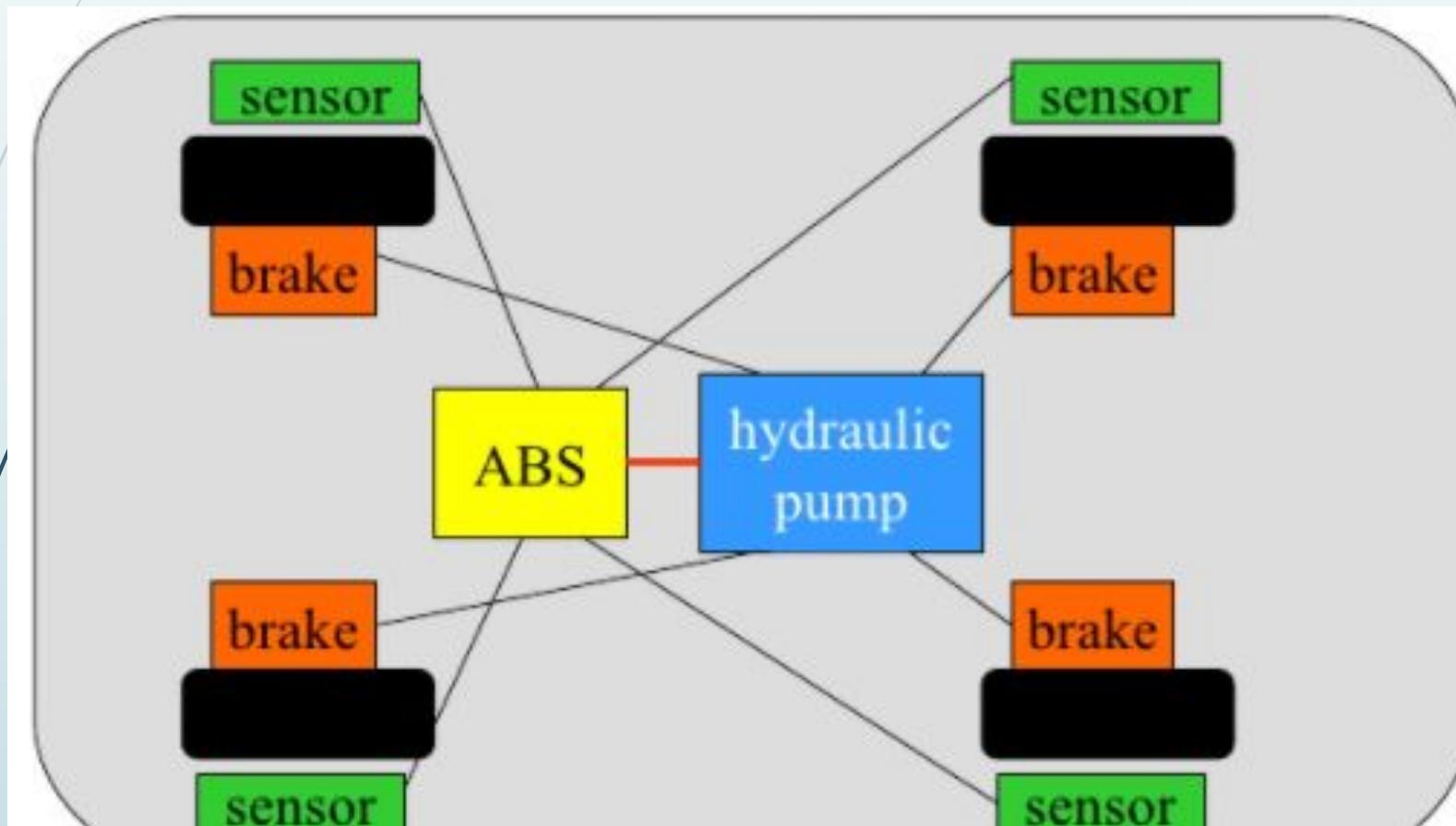
14

# Complex systems and Microprocessors

# Embedding Computers

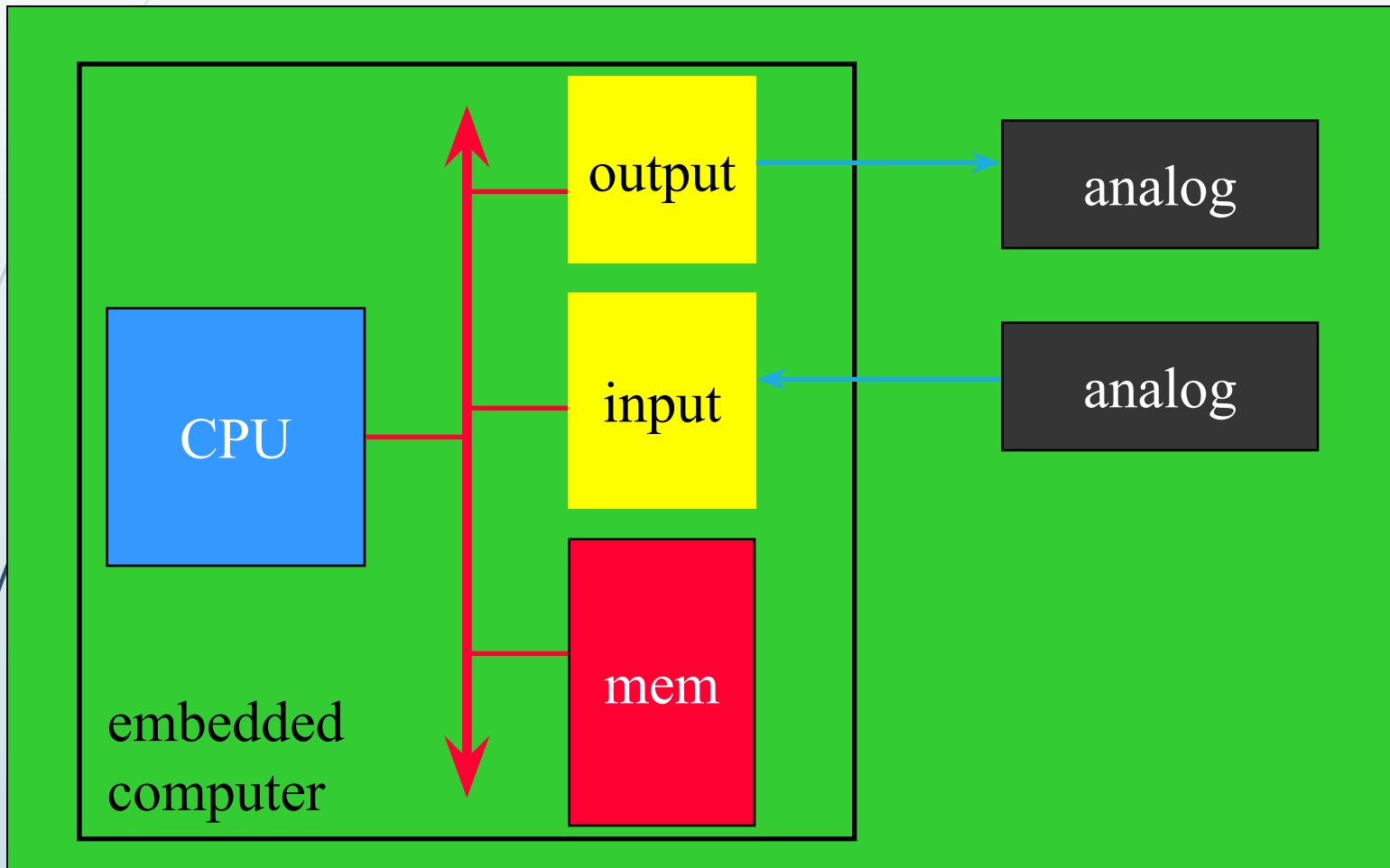
- Whirlwind was also the first computer designed to support real-time operation and was originally conceived as a mechanism for controlling an aircraft simulator.
- A microprocessor is a single-chip CPU.
- Very large scale integration (VLSI) has allowed us to put a complete CPU on a single chip
- The first microprocessor, the Intel 4004, was designed for an embedded application. It is called a calculator.

# Example: BMW 850i braking and control system



- The BMW 850i was introduced with a sophisticated system for controlling the wheels of the car.
- An antilock brake system (ABS) reduces skidding by pumping the brakes.
- An automatic stability control (ASC T) system intervenes with the engine during maneuvering to improve the car's stability.
- These systems actively control critical systems of the car; as control systems, they require inputs from and output to the automobile.

# Embedding a computer



| Criteria          | General Purpose Computer  | Embedded system  |
|-------------------|---|--|
| Contents          | It is combination of generic hardware and a general purpose OS for executing a variety of applications. | It is combination of special purpose hardware and embedded OS for executing specific set of applications |
| Operating System  | It contains general purpose operating system  | It may or may not contain operating system.  |
| Alterations       | Applications are alterable by the user.   | Applications are non-alterable by the user.  |
| Key factor        | Performance" is key factor.   | Application specific requirements are key factors.   |
| Power Consumption | More  | Less   |
| Response Time     | Not Critical  | Critical for some applications   |

# Characteristics of embedded systems

- Complex algorithms: Sophisticated functionality.
  - Example : For example, the microprocessor that controls an automobile engine must perform complicated filtering functions to optimize the performance of the car while minimizing pollution and fuel utilization.
- User interface: complex interface
  - Example: options. The moving maps in Global Positioning System (GPS) navigation.

- Real-time operation: system must follow deadline
  - If data not available on time system will crash
  - Failure to meet deadline may be unsafe to even lives.
  - May result in unhappy customers
- Multirate : several real time activities with different speed rate. All should be synchronized.
  - Example: video and audio working together
- Low manufacturing cost: depends on microprocessor, memory and input output devices.
- Low power and energy(heat).

# Functional complexity

- Often have to run sophisticated algorithms or multiple algorithms.
  - Cell phone, laser printer.
- Often provide sophisticated user interfaces.

# Real-time operation

- Must finish operations by deadlines.
  - Hard real time: missing deadline causes failure.
  - Soft real time: missing deadline results in degraded performance.
- Many systems are multi-rate: must handle operations at widely varying rates.

# Non-functional requirements

- Many embedded systems are mass-market items that must have low manufacturing costs.
  - Limited memory, microprocessor power, etc.
- Power consumption is critical in battery-powered devices.
  - Excessive power consumption increases system cost even in wall-powered devices.

# Design teams

- Often designed by a small team of designers.
- Often must meet tight deadlines.

# Why use microprocessors?

- Alternatives: field-programmable gate arrays (FPGAs), custom logic, etc.
- Microprocessors are often very efficient: can use same logic to perform many different functions.
- Microprocessors simplify the design of families of products.
- There are two factors that work together to make microprocessor-based designs fast
  - Microprocessors execute programs very efficiently. Parallelism can be employed to make faster processing.
  - Microprocessor manufacturers spend a great deal of money to make their CPUs run very fast.

- Several functions can be implemented on a single chip. Hence cost effective.
- we can modify the programs stored inside the microprocessor.
- Program design and hardware design can be done separately and parallelly.
- Family of products require only program change without hardware change.

## Microprocessors

Microprocessors are multitasking in nature. Can perform multiple tasks at a time. For example, on computer we can play music while writing text in text editor.

RAM, ROM, I/O Ports, and Timers can be added externally and can vary in numbers.

Designers can decide the number of memory or I/O ports needed.

External support of external memory and I/O ports makes a microprocessor-based system heavier and costlier.

External devices require more space and their power consumption is higher.

## Microcontroller

Single task oriented. For example, a washing machine is designed for washing clothes only.

RAM, ROM, I/O Ports, and Timers cannot be added externally. These components are to be embedded together on a chip and are fixed in numbers.

Fixed number for memory or I/O makes a microcontroller ideal for a limited but specific task

Microcontrollers are lightweight and cheaper than a microprocessor.

A microcontroller-based system consumes less power and takes less space.

# Challenges in embedded system design

- How much hardware do we need?
  - We need to select not only microprocessors but also amount of memory, peripheral devices etc.
  - Too little hardware will cause the system to fail.
  - Too much hardware will make the system expensive
- How do we meet our deadlines?
  - Need to decide whether we can speed up the CPU by increasing the clock rate or by increasing the amount of memory.
- How do we minimize power?
  - Excessive power consumption increases the heat dissipation in non battery systems and reduces battery life in battery based systems.
  - Power consumption can be reduced by reducing the speed without affecting the deadlines.

# Challenges, etc.

- How do we design for upgradability?
  - Adding new features by changing software
- Does it really work?
  - Is the specification correct?
  - Does the implementation meet the spec?
  - How do we test for real-time characteristics?
  - How do we test on real data?
- How do we work on the system?
  - Observability, controllability?
  - What is our development platform?

# Why embedded system design is difficult?

- Complex testing
  - Need to get real time data
  - Timing of data collection is also important
  - Need to testing on the embedded system itself
- Limited observability and controllability:
  - No keyboard and screen. Hence difficult to observe what is going on in the system.
- Restricted development environments:
  - Usually do the programs in pc and put into embedded machine.

# The performance paradox

- Microprocessors use much more logic to implement a function than does custom logic.
- But microprocessors are often at least as fast:
  - heavily pipelined;
  - VLSI technology.

# What does “performance” mean?

- In general-purpose computing, performance often means average-case, may not be well-defined.
- The main concept behind embedded computing is real time computing.
- Deadline is important in real time computing
- It is the time by which the work has to be completed.
- In real-time systems, performance means meeting deadlines.
  - Missing the deadline by even a little is bad.
  - Finishing ahead of the deadline may not help.

# Layers in an Embedded System

- **CPU:** The CPU influences the behavior of the program, particularly when the CPU is a pipelined processor with a cache.
- **Platform:** The platform includes the bus and I/O devices. The platform components that surround the CPU are responsible for feeding the CPU and can dramatically affect its performance.
- **Program:** Programs are very large and the CPU sees only a small window of the program at a time.
- Task: We generally run several programs simultaneously on a CPU, creating a multitasking system. The tasks interact with each other

- **Multiprocessor:** Many embedded systems have more than one processors, they may include multiple programmable CPUs as well as accelerators which interacts each other.

# EMBEDDED SYSTEM DESIGN PROCESS

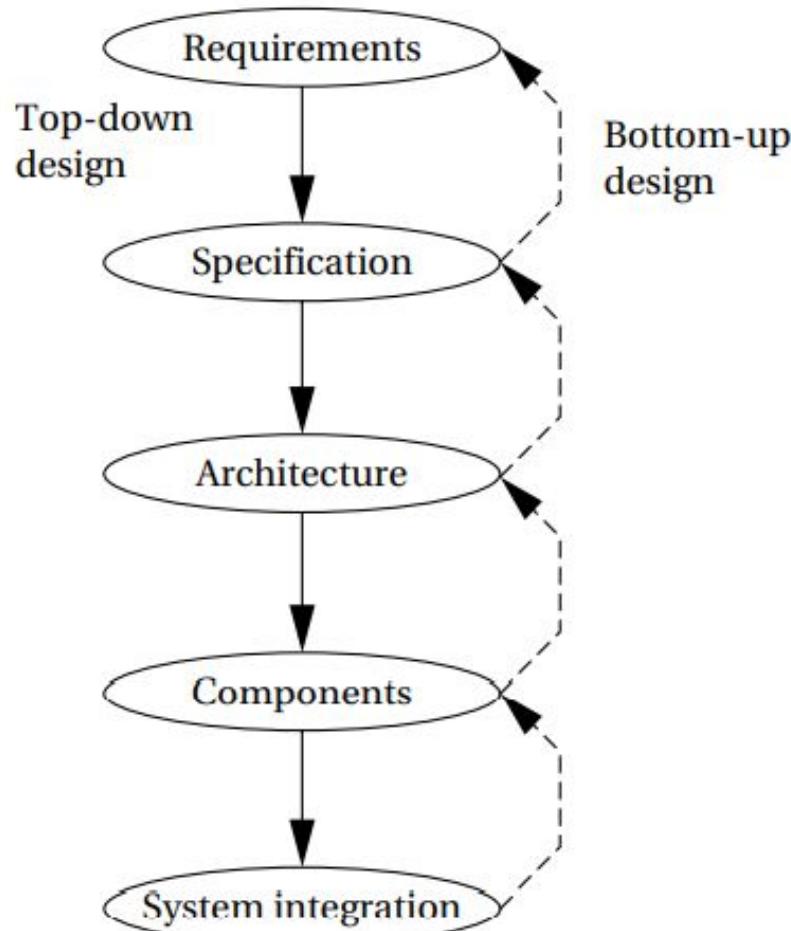
# Objective of Embedded System Design Process

- It will give us an introduction to the various steps in embedded system design
- It will allow us to consider the design methodology itself

# Importance of design methodology

- It allows us to keep a scorecard on a design to ensure that we have done everything we need to do
- It allows us to develop computer-aided design tools
- A design methodology makes it much easier for members of a design team to communicate
- By defining the overall process, team members can more easily understand what they are supposed to do, what they should receive from other team members at certain times, and what they are to hand off when they complete their assigned steps

# Steps in design process



- Requirements:
  - Collect the requirements
- Specification
  - Tells how the system behaves
- Architecture
  - Gives the details of the systems internal structure.
- Components
  - Decide the components and create any software or hardware we need
- System integration
  - Integrate the components to make a complete system

# Types of product development methodology

- Top-down design:
  - start from most abstract description;
  - work to most detailed.
- Bottom-up design:
  - work from small components to big system.
- Real design uses both techniques.

# Major Design goals

- Performance.
  - Overall speed, deadlines.
- Functionality and user interface.
- Manufacturing cost.
- Power consumption.
- Other requirements (physical size, etc.)

# Stepwise refinement

- At each level of abstraction, we must:
  - **Analyze** the design to determine characteristics of the current state of the design;
  - **Refine** the design to add detail.
  - Verify the design to ensure that it still meets all system goals

# Requirements

- An informal description obtained from the customers
- Plain language description of what the user wants and expects to get.
- May be developed in several ways:
  - talking directly to customers;
  - talking to marketing representatives;
  - providing prototypes to users for comment.

# Functional vs. non-functional requirements

- Functional requirements:
  - Output that should be obtained from the system
  - It is a function of input.
- Non-functional requirements:
  - Performance: time required to compute output, deadline by which a process should be completed
  - Cost: it is the purchase price. Cost is of two types.
    - Manufacturing cost: cost of components and assembly
    - Non recurring Engineering cost: personal and cost in designing the system.

- Physical size, weight: depends on application
- Power consumption: important for battery powered device
- reliability;
- etc.

# Our requirements form

47



name  
purpose  
inputs  
outputs  
functions  
performance  
manufacturing cost  
power  
physical size/weight

# Requirement form

- Name: Name of the project
- Purpose: what the system is supposed to do
- Inputs and outputs: Specifies the type of data, Data characteristics, Types of I/O devices.
- Functions: description of what the system do
- Functions: computation must be performed within a time frame
- Manufacturing cost: cost of hardware components
- Power: decide whether battery powered or not
- Physical size and weight

# GPS moving map needs

- **Functionality:** For automotive use. Show major roads and landmarks.
- **User interface:** At least 400 x 600 pixel screen. Three buttons max. Pop-up menu.
- **Performance:** Map should scroll smoothly. No more than 1 sec power-up. Lock onto GPS within 15 seconds.
- **Cost:** \$120 street price = approx. \$30 cost of goods sold.

# GPS moving map needs, cont'd.

- **Physical size/weight:** Should fit in hand.
- **Power consumption:** Should run for 8 hours on four AA batteries.

# GPS moving map requirements form

51

|                             |  |
|-----------------------------|--|
| <b>name</b>                 | GPS moving map   |
| <b>purpose</b>              | consumer-grade<br>moving map for driving   |
| <b>inputs</b>               | power button, two<br>control buttons   |
| <b>outputs</b>              | back-lit LCD 400 X 600   |
| <b>functions</b>            | 5-receiver GPS; three<br>resolutions; displays<br>current lat/lon<br>updates screen within<br>0.25 sec of movement |
| <b>performance</b>          | \$100 cost-of-goods-<br>sold   |
| <b>manufacturing cost</b>   | 100 mW   |
| <b>power</b>                | no more than 2: X 6:,<br>12 oz.  |
| <b>physical size/weight</b> |  |

# Specification

- A more precise description of the system:
  - should not imply a particular architecture;
  - provides input to the architecture design process.
- It accurately reflects the customer's requirements and does so in a way that can be clearly followed during design.
- It should be understandable enough so that someone can verify that it meets system requirements and overall expectations of the customer.
- It should also be unambiguous enough that designers know what they need to build.

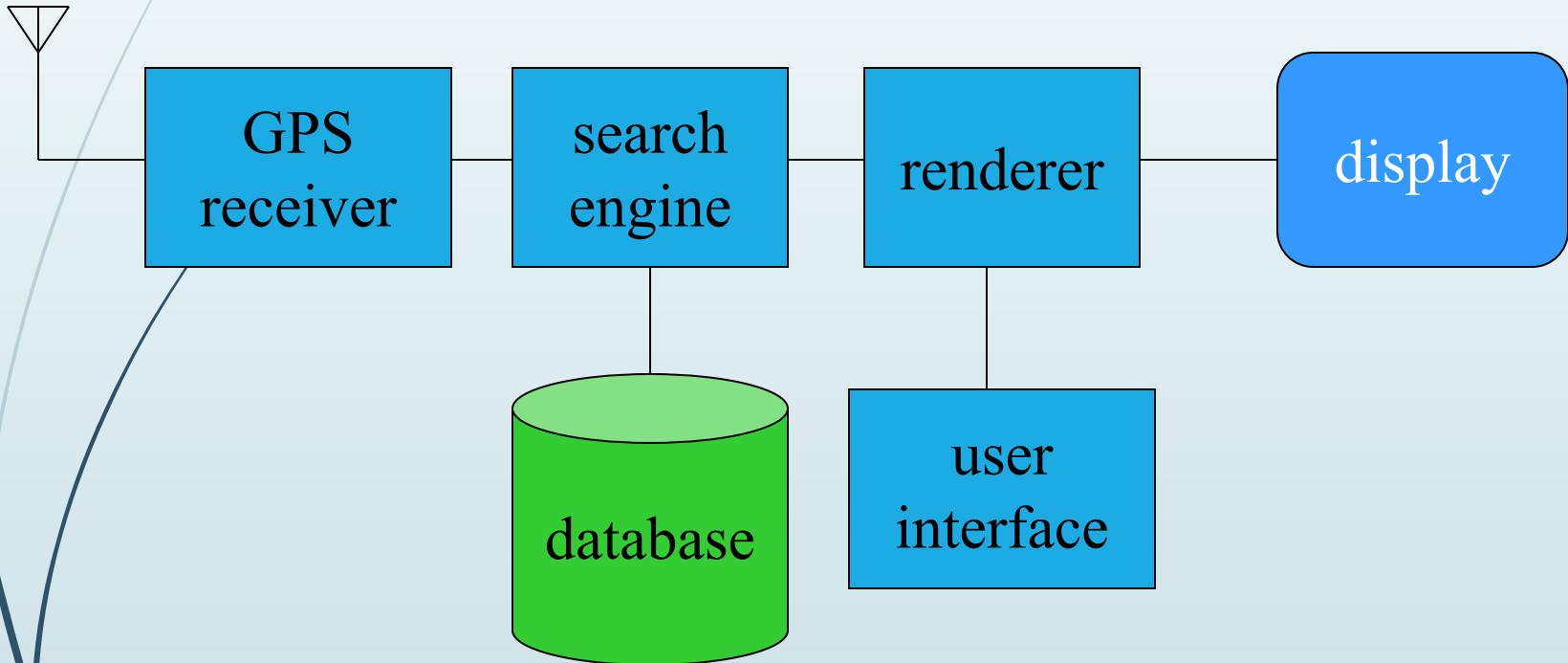
# GPS specification

- Should include:
  - What is received from GPS;
  - map data;
  - user interface;
  - operations required to satisfy user requests;
  - background operations needed to keep the system running.

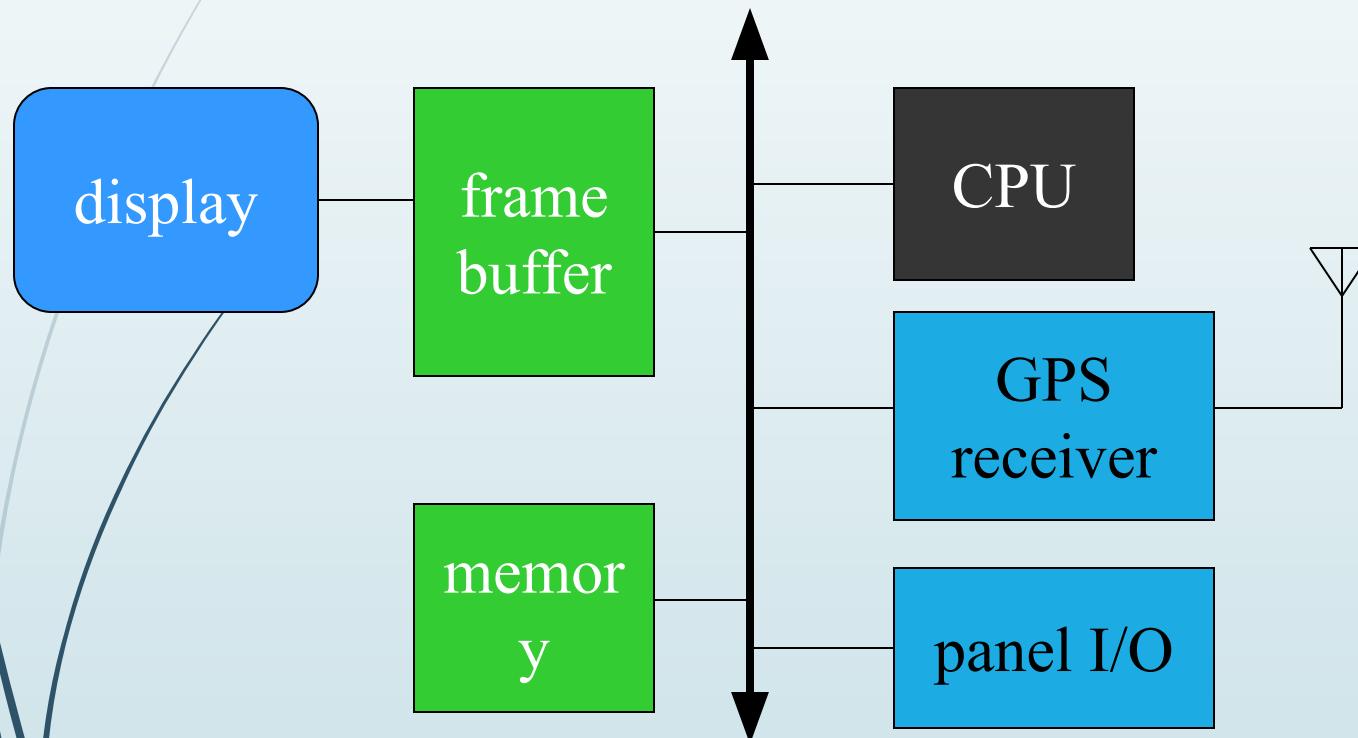
# Architecture design

- The specification does not say how the system does things, only what the system does.
- Describing how the system implements those functions is the purpose of the architecture.
- The architecture is a plan for the overall structure of the system that will be used later to design the components that make up the architecture.
- Hardware components:
  - CPUs, peripherals, etc.
- Software components:
  - major programs and their operations.
- Must take into account functional and non-functional specifications.

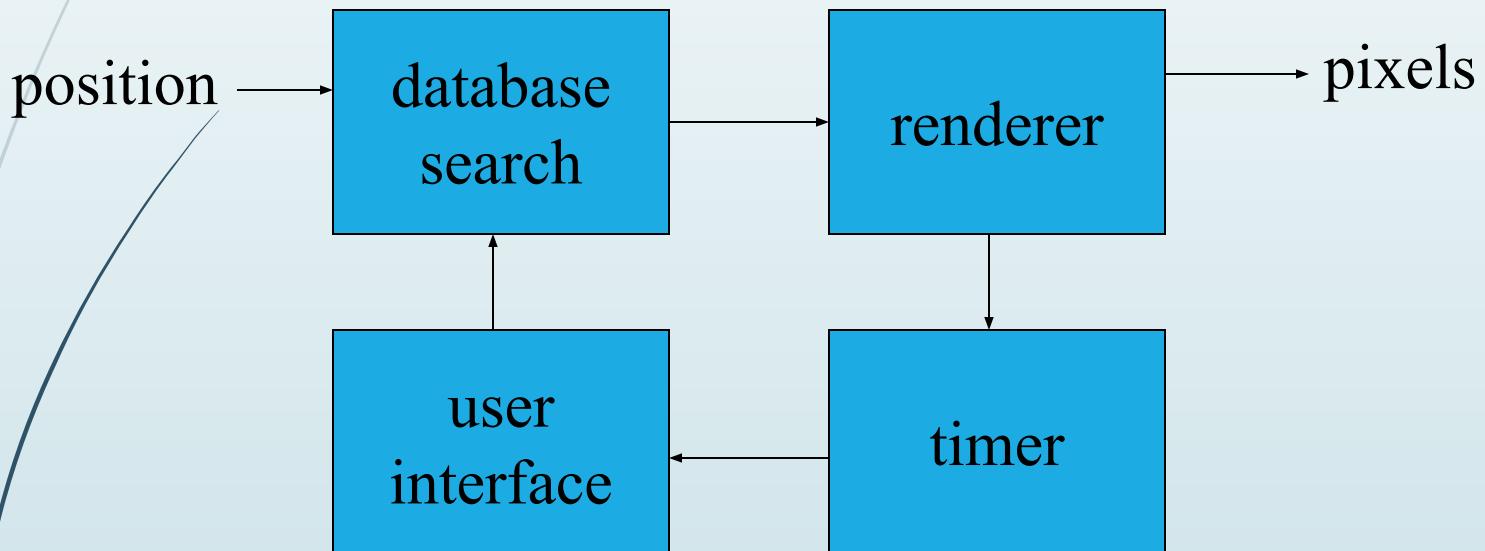
# GPS moving map block diagram



# GPS moving map hardware architecture



# GPS moving map software architecture



# Designing hardware and software components

- Must spend time architecting the system before you start coding.
- Some components are ready-made, some can be modified from existing designs, others must be designed from scratch.

# System integration

- Put together the components.
  - Many bugs appear only at this stage.
- Have a plan for integrating components to uncover bugs quickly, test as much functionality as early as possible.



# FORMALISMS FOR SYSTEM DESIGN

60

Structural and behavioral description.

# Introduction

- The different phases of embedded system design can be conceptualize using UML diagram for better understanding.
- UML is useful because it encourages design by successive refinement and progressively adding detail to the design

# UML

- UML is an object-oriented modeling language.
- Object oriented modelling emphasizes two concepts
  - It encourages the design to be described as a number of interacting objects,
  - At least some of those objects will correspond to real pieces of software or hardware in the system.
  - We can also use UML to model the outside world that interacts with our system

# OO Specification

- Object-oriented specification allows a system to be described in a way that closely models real-world objects and their interactions.
- Object-oriented specification provides a basic set of primitives that can be used to describe systems with particular attributes.

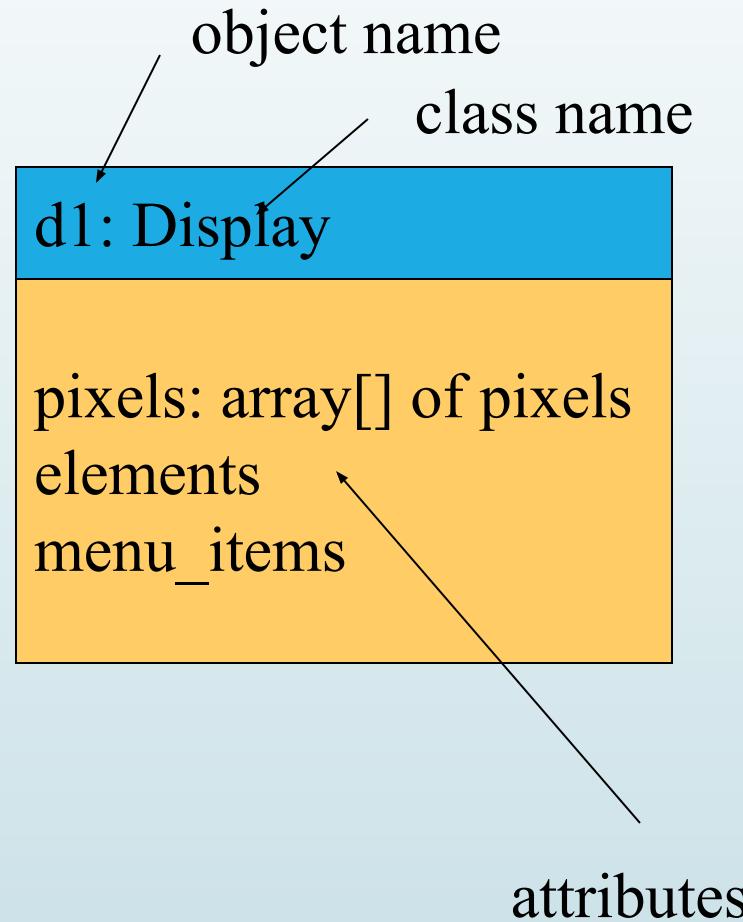
# Structural Description

- Object: The principal component of an object-oriented design is, naturally enough, the object
- An object includes a set of attributes that define its internal state
- Object has a unique name and is part of a class
- Underlined name is an object

# UML object

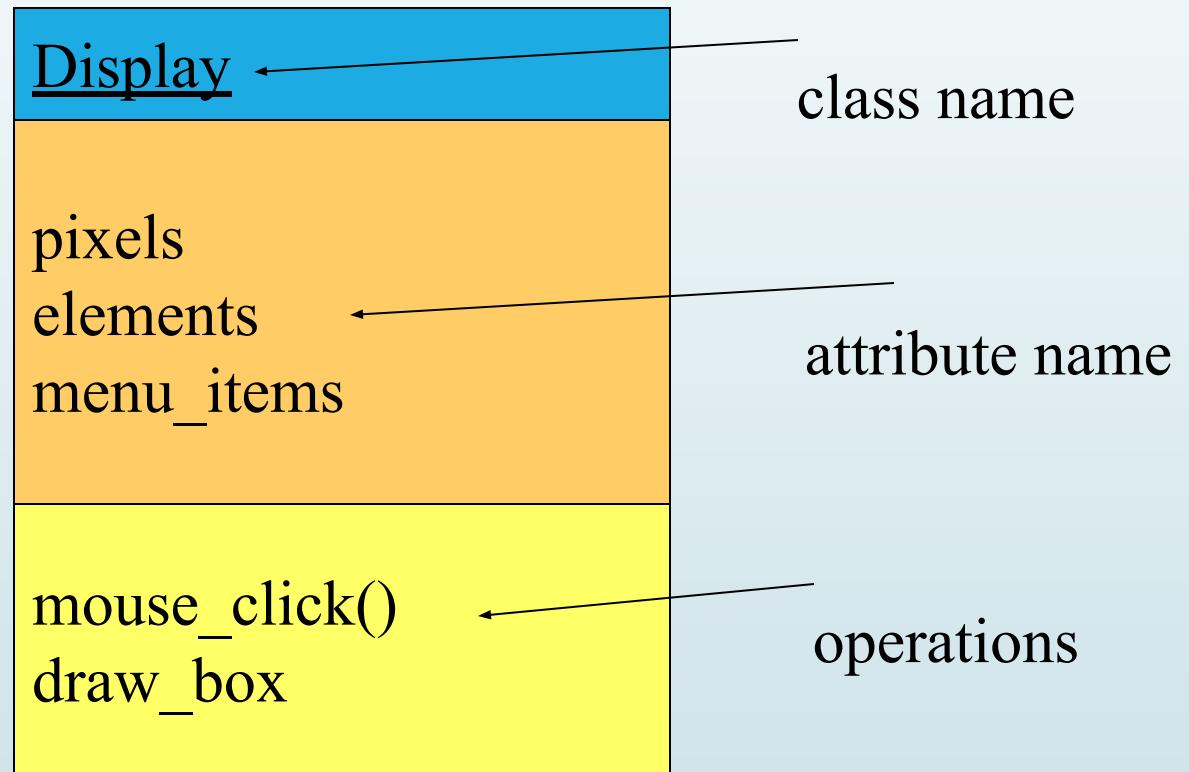
pixels is a  
2-D array

comment



- All objects derived from the same class have the same characteristics, although their attributes may have different values.
- A class defines the attributes that an object may have.
- It also defines the operations that determine how the object interacts with the rest of the world

# UML class



# The class interface

- The operations provide the abstract interface between the class's implementation and other classes.
- Operations may have arguments, return values.
- An operation can examine and/or modify the object's state.

# Choose your interface properly

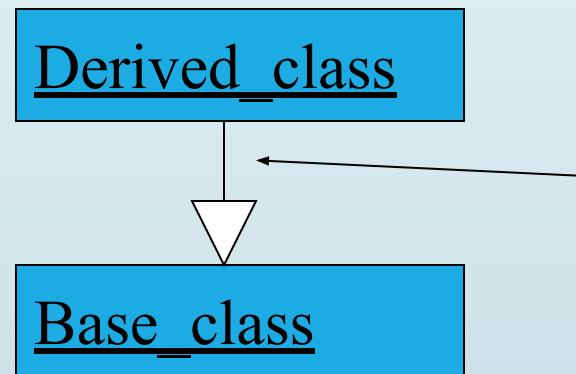
- If the interface is too small/specialized:
  - object is hard to use for even one application;
  - even harder to reuse.
- If the interface is too large:
  - class becomes too cumbersome for designers to understand;
  - implementation may be too slow;
  - spec and implementation are probably buggy.

# Relationships between objects and classes

- **Association**: objects communicate but one does not own the other.
- **Aggregation**: a complex object is made of several smaller objects.
- **Composition**: aggregation in which owner does not allow access to its components.
- **Generalization**: define one class in terms of another.

# Class derivation

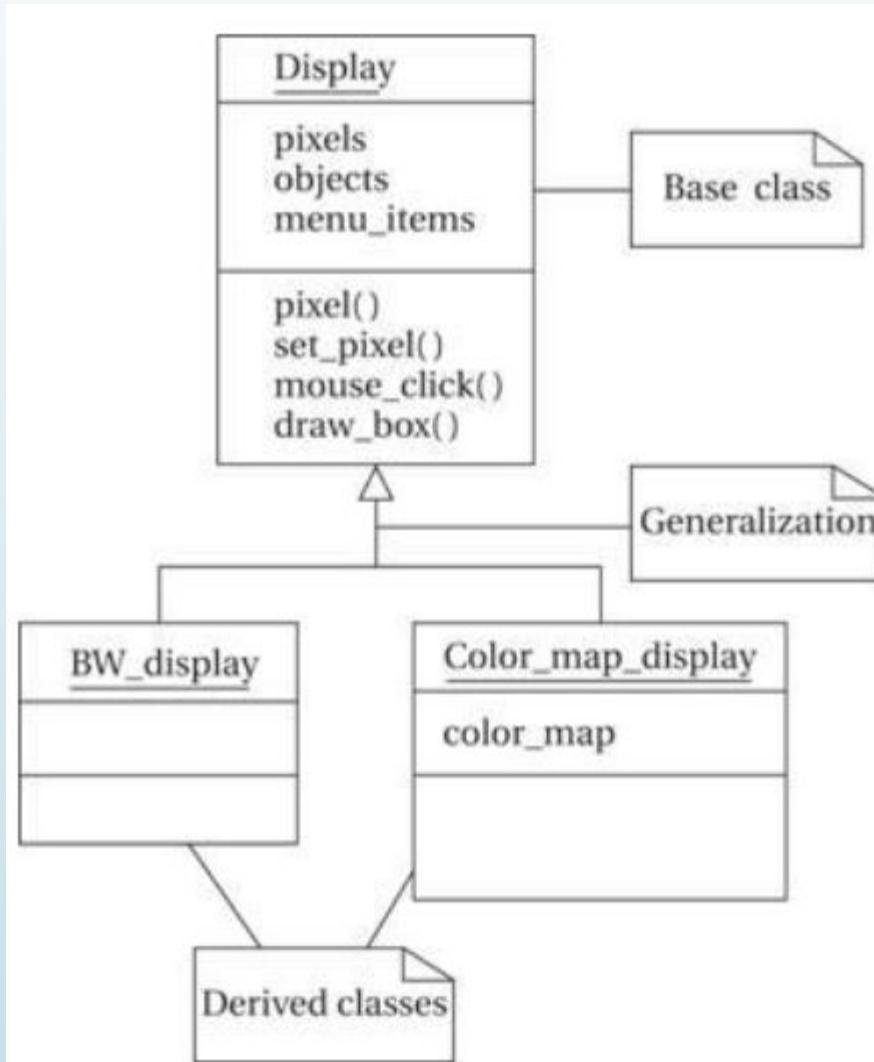
- May want to define one class in terms of another.
  - Derived class **inherits** attributes, operations of base class.



UML  
generalization

# Class derivation example

72



- The first, `BW_display`, describes a blackand-white display. This does not require us to add new attributes or operations, but we can specialize both to work on one-bit pixels.
- The second, `Color_map_display`, uses a graphic device known as a color map to allow the user to select from a large number of available colors even with a small number of bits per pixel. This class defines a `color_map` attribute that determines how pixel values are mapped onto display colors.

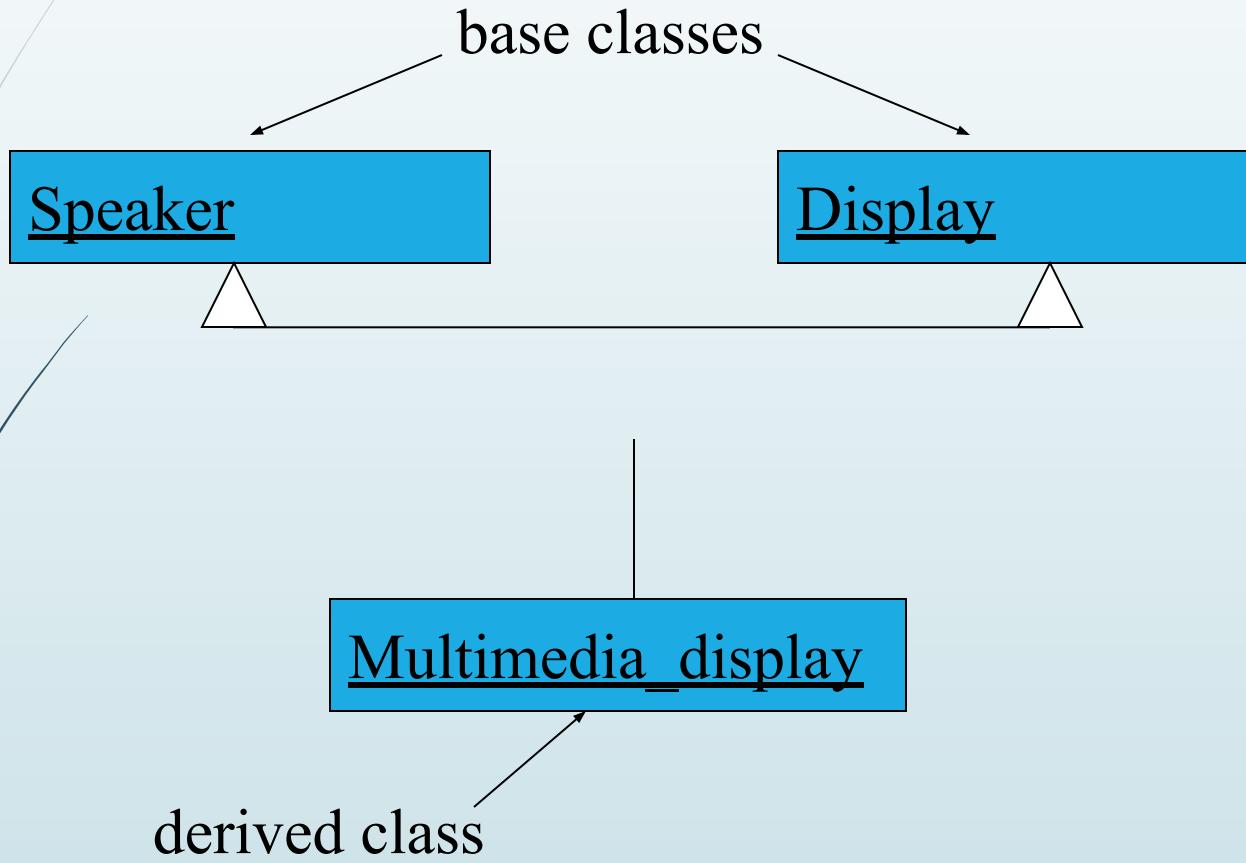
# Inheritance

- A derived class inherits all the attributes and operations from its base class.
- Purpose of inheritance is
  - shares some characteristics with another class
  - it captures those relationships between classes and documents them

# Generalization

- Unified Modeling Language considers inheritance to be one form of generalization.
- A generalization relationship is shown in a UML diagram as an arrow with an open (unfilled) arrowhead.

# Multiple inheritance

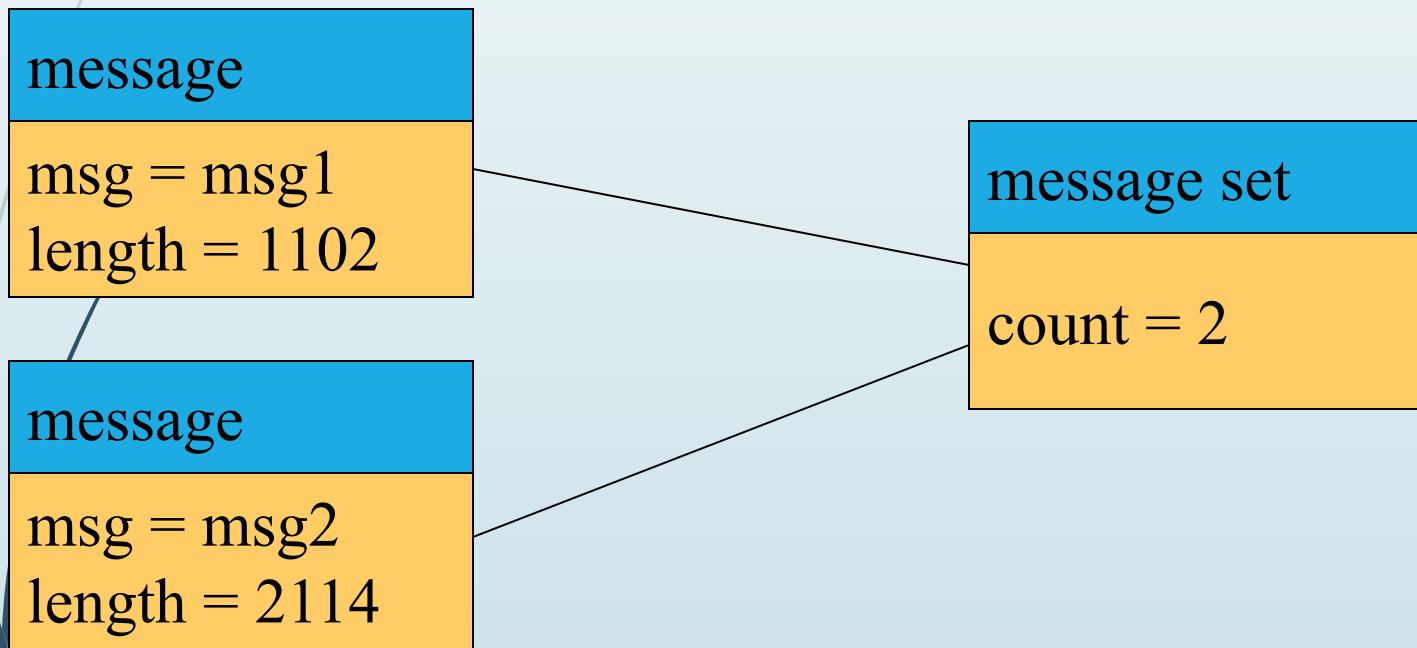


# Links and associations

- **Link**: describes relationships between objects.
- **Association**: describes relationship between classes.

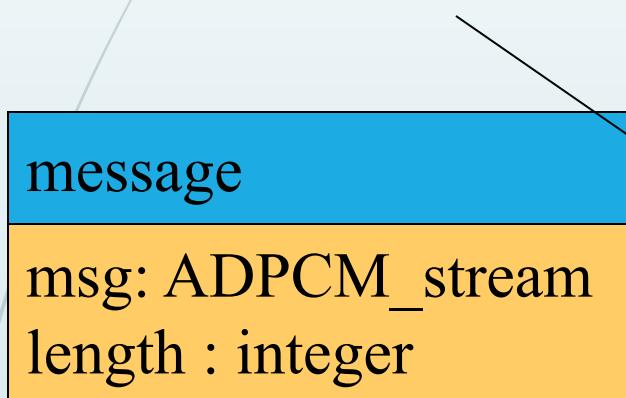
# Link example

- Link defines the **contains** relationship:

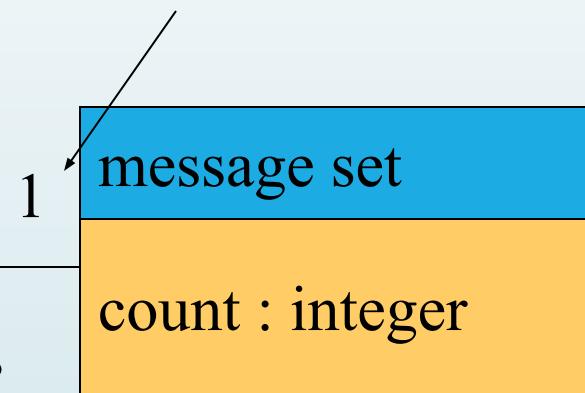


# Association example

# contained messages

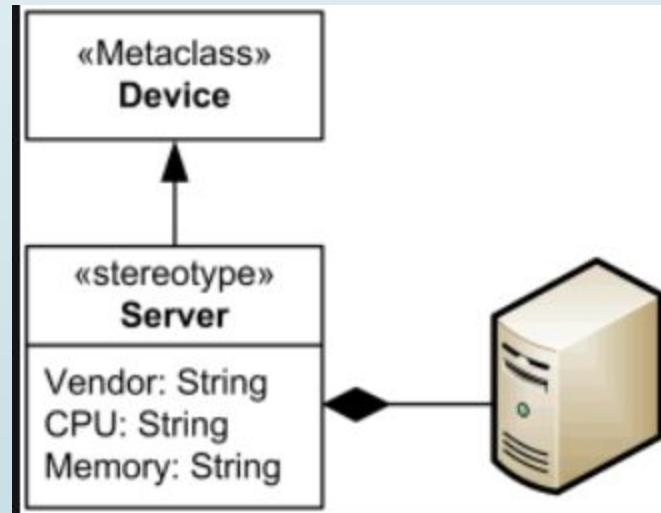


# containing message sets



# Stereotypes

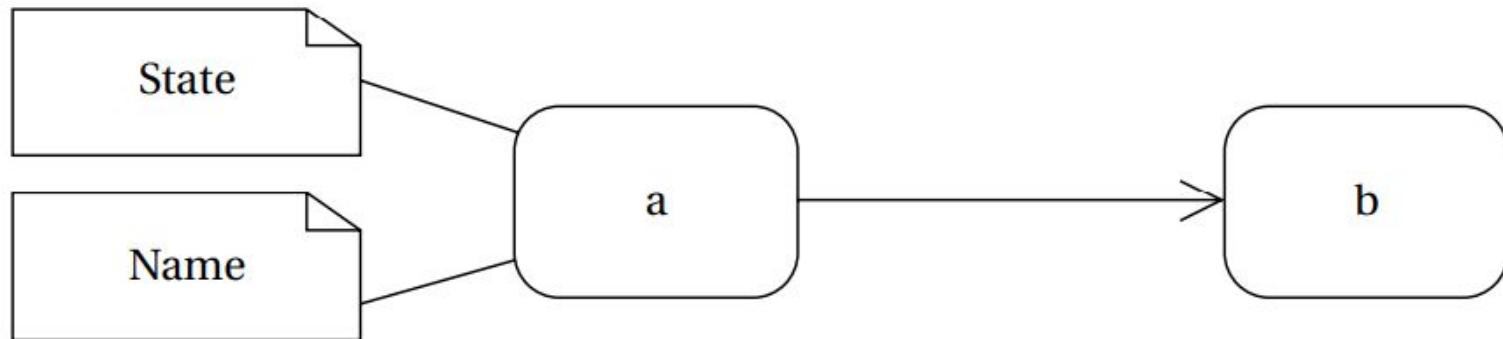
- **Stereotype:** recurring combination of elements in an object or class.
- we find that we use a certain combination of elements in an object or class many times which are called stereotypes
- Example:



# Behavioral description

- Several ways to describe behavior:
  - internal view;
  - external view.

# State machines



# Event-driven state machines

- Behavioral descriptions are written as event-driven state machines.
  - Machine changes state when receiving an input.
- Event is an action
- An event may come from inside(finishes one routine and give its result to another for continuation of that routine) or outside of the system(pressing button).

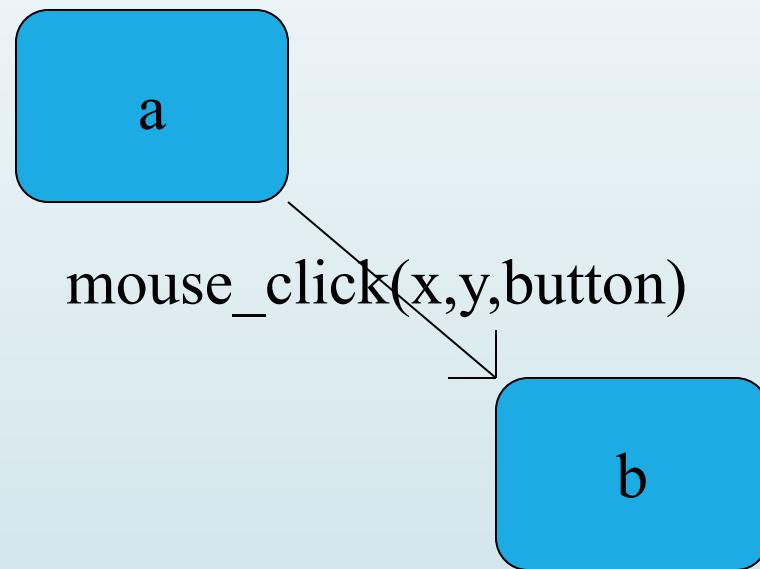
# Types of events

- **Signal**: asynchronous event.
  - It is defined in UML by an object that is labeled as a <<signal>>.
  - The object in the diagram serves as a declaration of the event's existence.
  - Because it is an object, a signal may have parameters that are passed to the signal's receiver.
- **Call**: synchronized communication.
  - follows the model of a procedure call in a programming language.
- **Timer**: activated by time.
  - causes the machine to leave a state after a certain amount of time.
  - The label  $tm(\text{time-value})$  on the edge gives the amount of time after which the transition occurs.

# Signal event

```
<<signal>>  
mouse_click  
leftright: button  
x, y: position
```

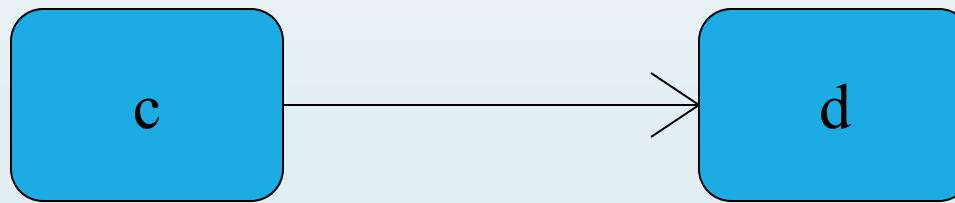
declaration



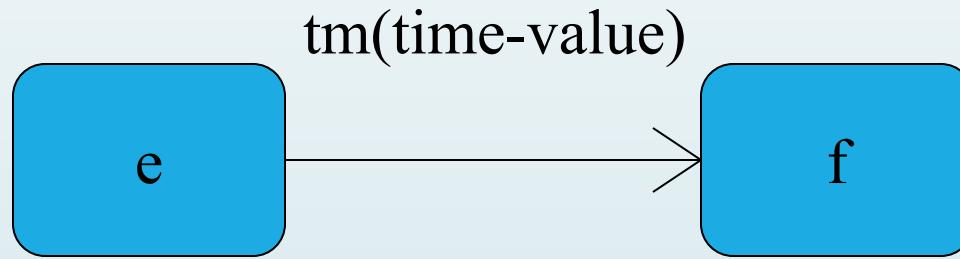
event description

# Call event

```
draw_box(10,5,3,2,blue)
```



# Timer event



# Example state machine

start

mouse\_click(x,y,button)/  
find\_region(region)

region = menu/  
which\_menu(i)

input/output

got menu  
item

call\_menu(I)

called  
menu  
item

region = drawing/  
find\_object(objid)

highlight(objid)

found  
object

object  
highlighted

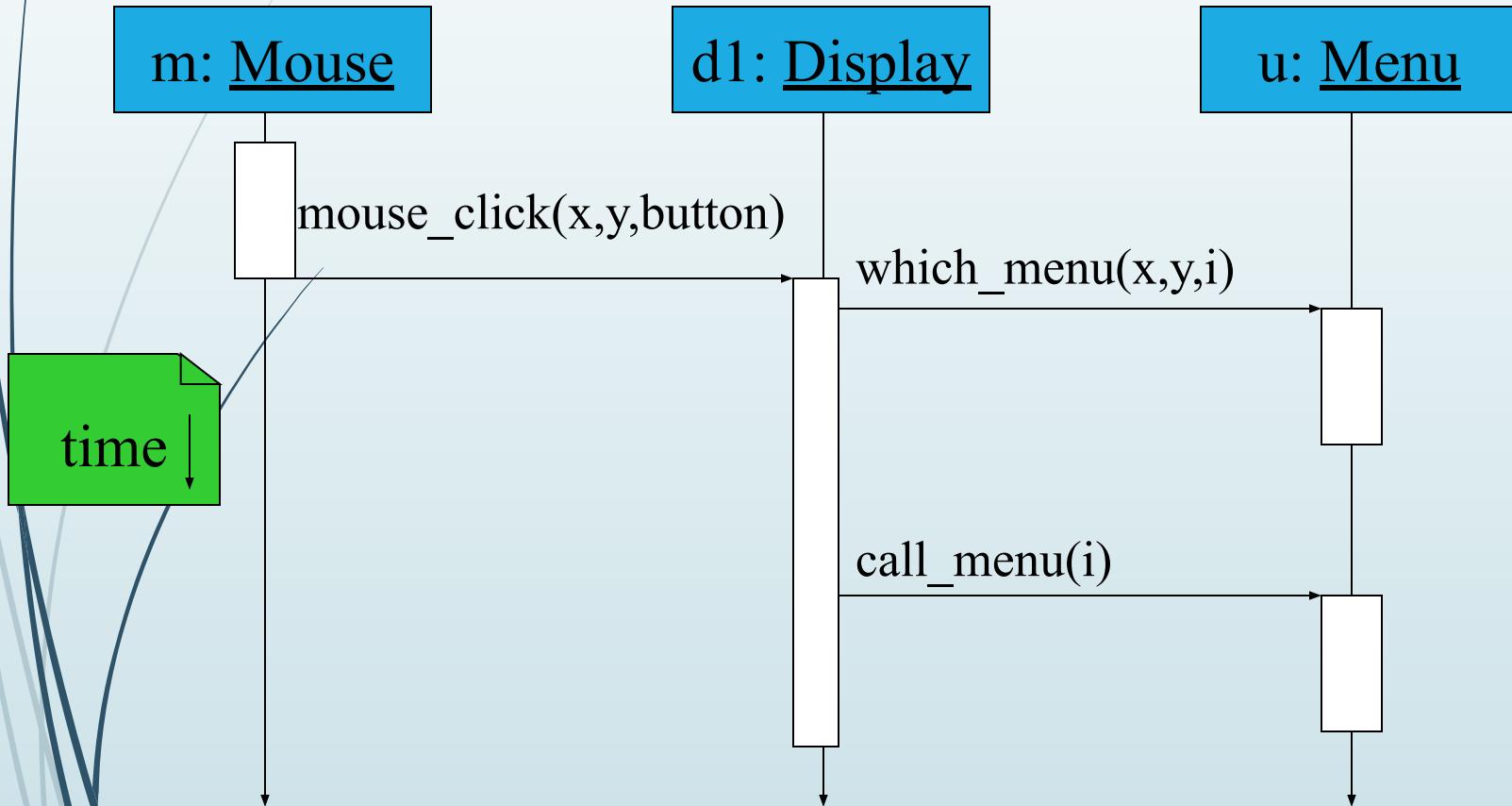
finish

- The start and stop states are special states that help us to organize the flow of the state machine.
- The states in the state machine represent different conceptual operations.
- In some cases, we take conditional transitions out of states based on inputs or the results of some computation done in the state.
- In other cases, we make an unconditional transition to the next state.
- Both the unconditional and conditional transitions make use of the call event.
- Splitting a complex operation into several states helps document the required steps, much as subroutines can be used to structure code.

# Sequence diagram

- Shows sequence of operations over time.
- Relates behaviors of multiple objects.

# Sequence diagram example



# Summary

- Object-oriented design helps us organize a design.
- UML is a transportable system design language.
  - Provides structural and behavioral description primitives.

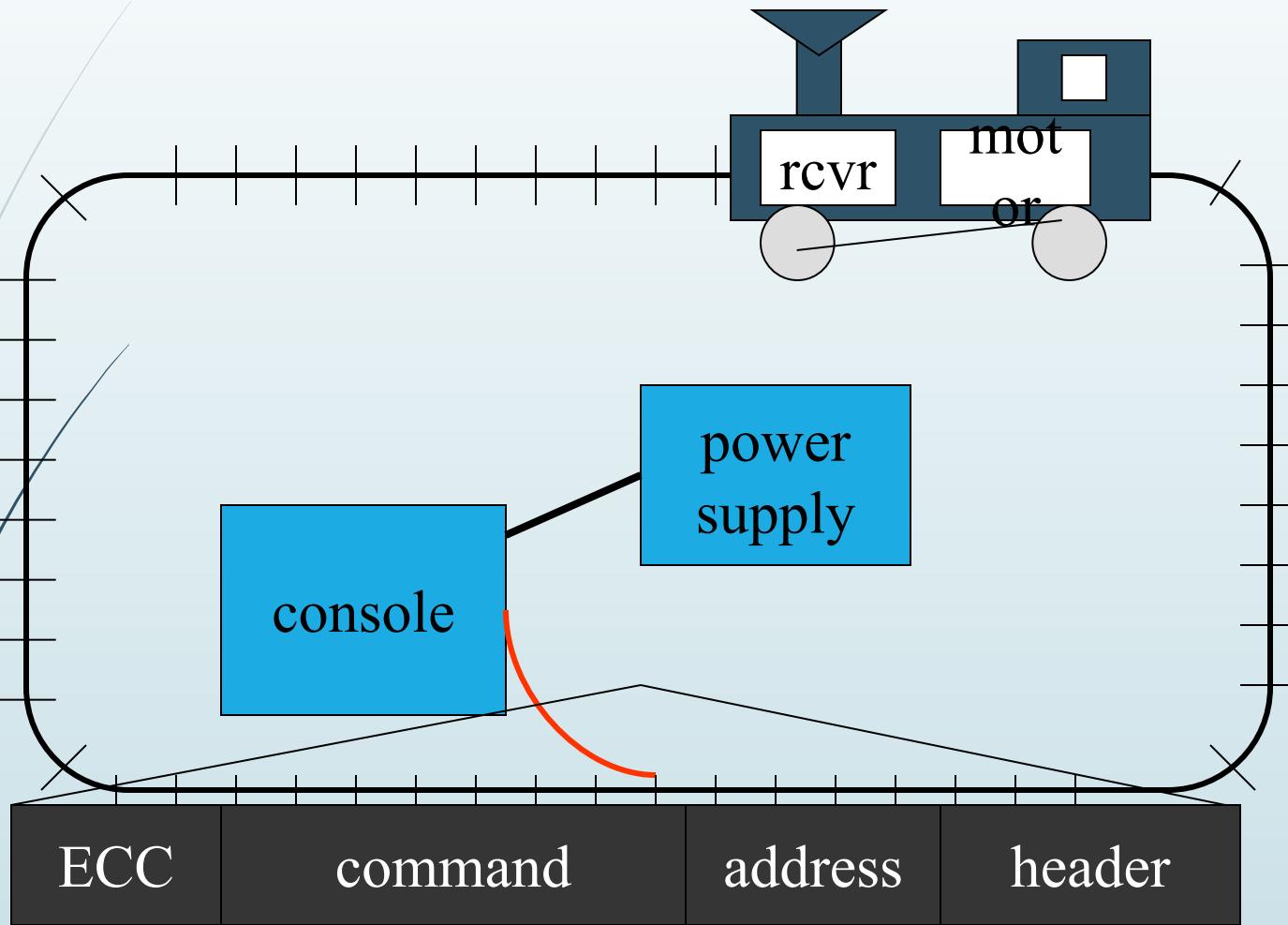
# Introduction

- Example: model train controller.

# Purposes of example

- Follow a design through several levels of abstraction.
- Gain experience with UML.

# Model train setup



# Requirements

- Console can control 8 trains on 1 track.
- Throttle has at least 63 levels.
- Inertia control adjusts responsiveness with at least 8 levels.
- Emergency stop button.
- Error detection scheme on messages.

# Requirements form

97

|                             |   |
|-----------------------------|---|
| <b>name</b>                 | model train controller                          |
| <b>purpose</b>              | control speed of <= 8 model trains              |
| <b>inputs</b>               | throttle, inertia, emergency stop,<br>train #   |
| <b>outputs</b>              | train control signals                           |
| <b>functions</b>            | set engine speed w. inertia;<br>emergency stop  |
| <b>performance</b>          | can update train speed at least 10<br>times/sec |
| <b>manufacturing cost</b>   | \$50  |
| <b>power</b>                | wall powered                                    |
| <b>physical size/weight</b> | console comfortable for 2 hands; < 2<br>lbs.    |

# Digital Command Control

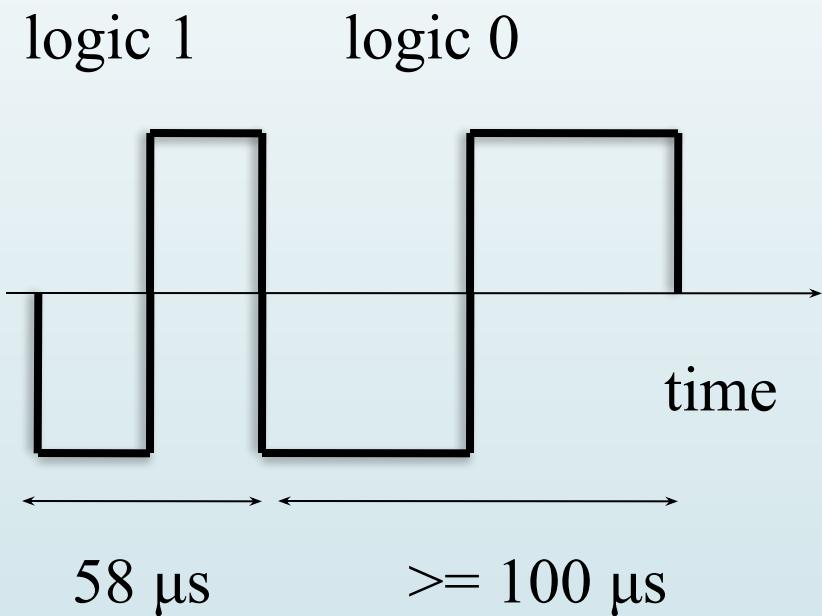
- DCC created by model railroad hobbyists, picked up by industry.
- Defines way in which model trains, controllers communicate.
  - Leaves many system design aspects open, allowing competition.
- This is a simple example of a big trend:
  - Cell phones, digital TV rely on standards.

# DCC documents

- Standard S-9.1, DCC Electrical Standard.
  - Defines how bits are encoded on the rails.
- Standard S-9.2, DCC Communication Standard.
  - Defines packet format and semantics.

# DCC electrical standard

- Voltage moves around the power supply voltage; adds no DC component.
- 1 is 58  $\mu$ s, 0 is at least 100  $\mu$ s.



# DCC communication standard

- Basic packet format: PSA(sD)+E.
- P: preamble = 1111111111.
- S: packet start bit = 0.
- A: address data byte.
- s: data byte start bit.
- D: data byte (data payload).
- E: packet end bit = 1.

# DCC packet types

- Baseline packet: minimum packet that must be accepted by all DCC implementations.
  - Address data byte gives receiver address.
  - Instruction data byte gives basic instruction.
  - Error correction data byte gives ECC.

# Conceptual specification

- Before we create a detailed specification, we will make an initial, simplified specification.
  - Gives us practice in specification and UML.
  - Good idea in general to identify potential problems before investing too much effort in detail.

# Basic system commands

104

**command name**

**set-speed**

**set-inertia**

**estop**

**parameters**

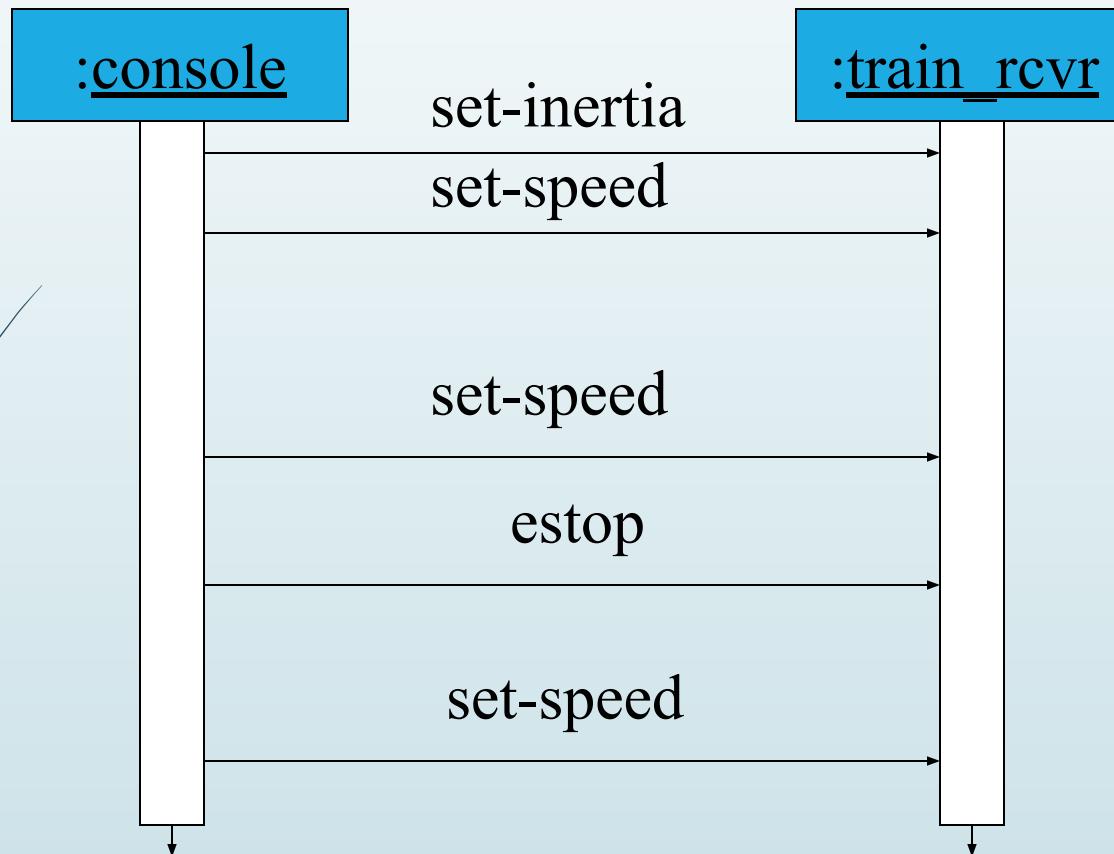
**speed**

(positive/negative)

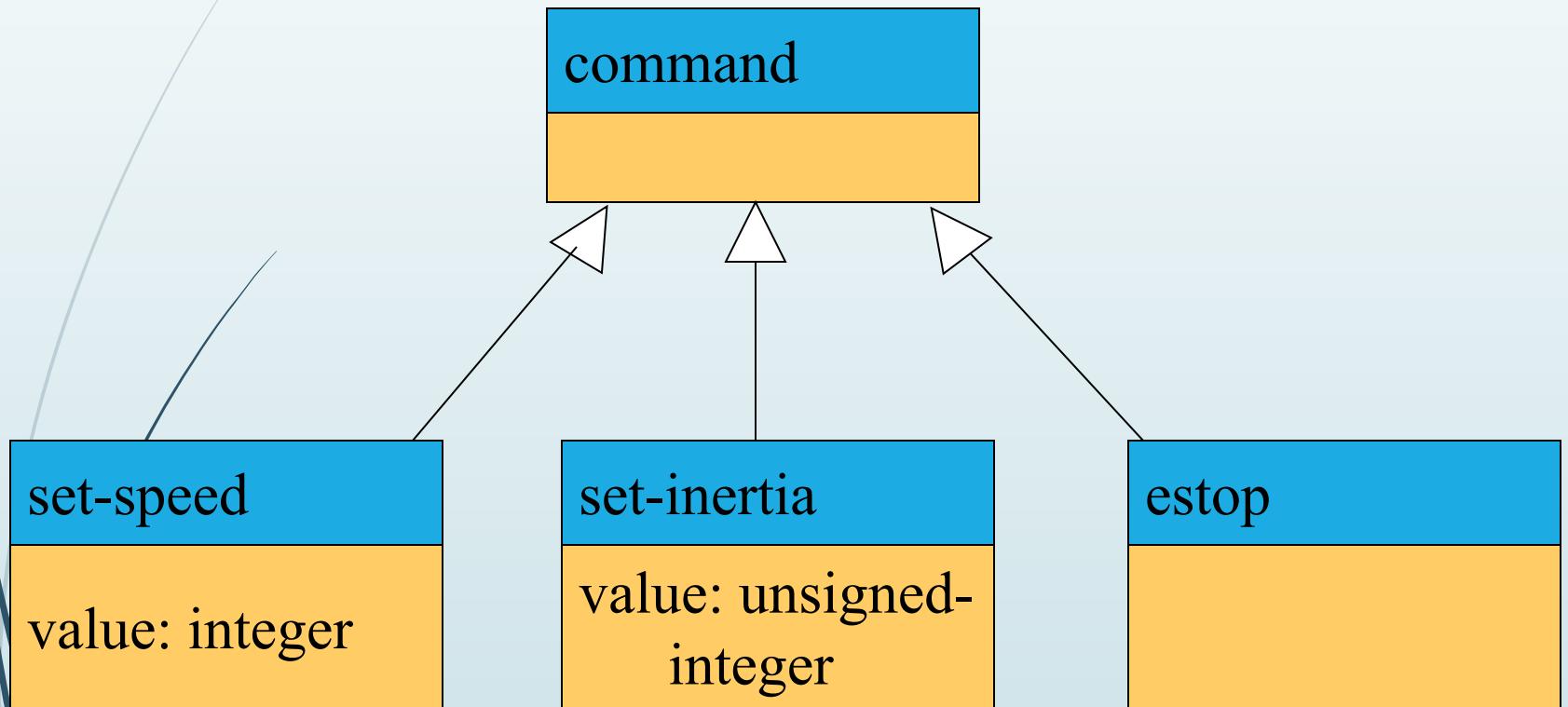
**inertia-value (non-negative)**

**none**

# Typical control sequence



# Message classes

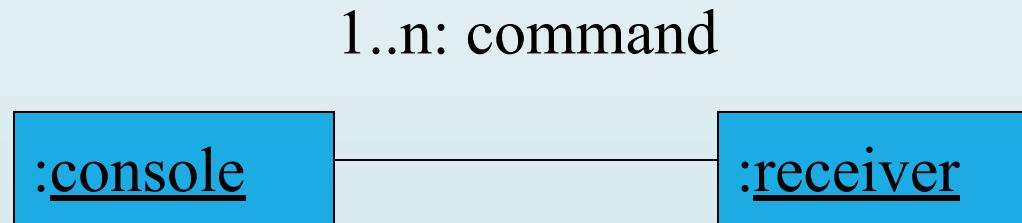


# Roles of message classes

- Implemented message classes derived from message class.
  - Attributes and operations will be filled in for detailed specification.
- Implemented message classes specify message type by their class.
  - May have to add type as parameter to data structure in implementation.

# Subsystem collaboration diagram

Shows relationship between console and receiver (ignores role of track):



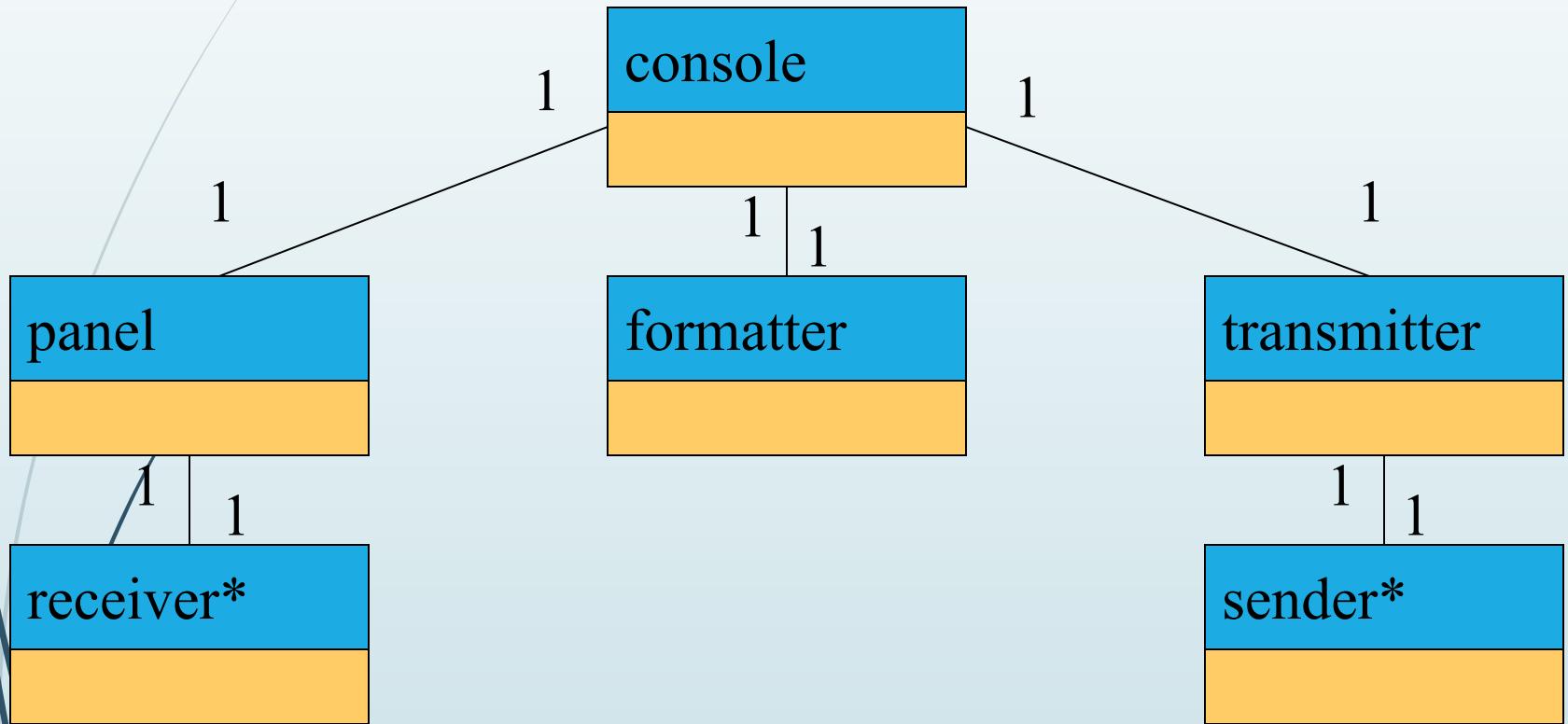
# System structure modeling

- Some classes define non-computer components.
  - Denote by \*name.
- Choose important systems at this point to show basic relationships.

# Major subsystem roles

- **Console:**
  - read state of front panel;
  - format messages;
  - transmit messages.
- **Train:**
  - receive message;
  - interpret message;
  - control the train.

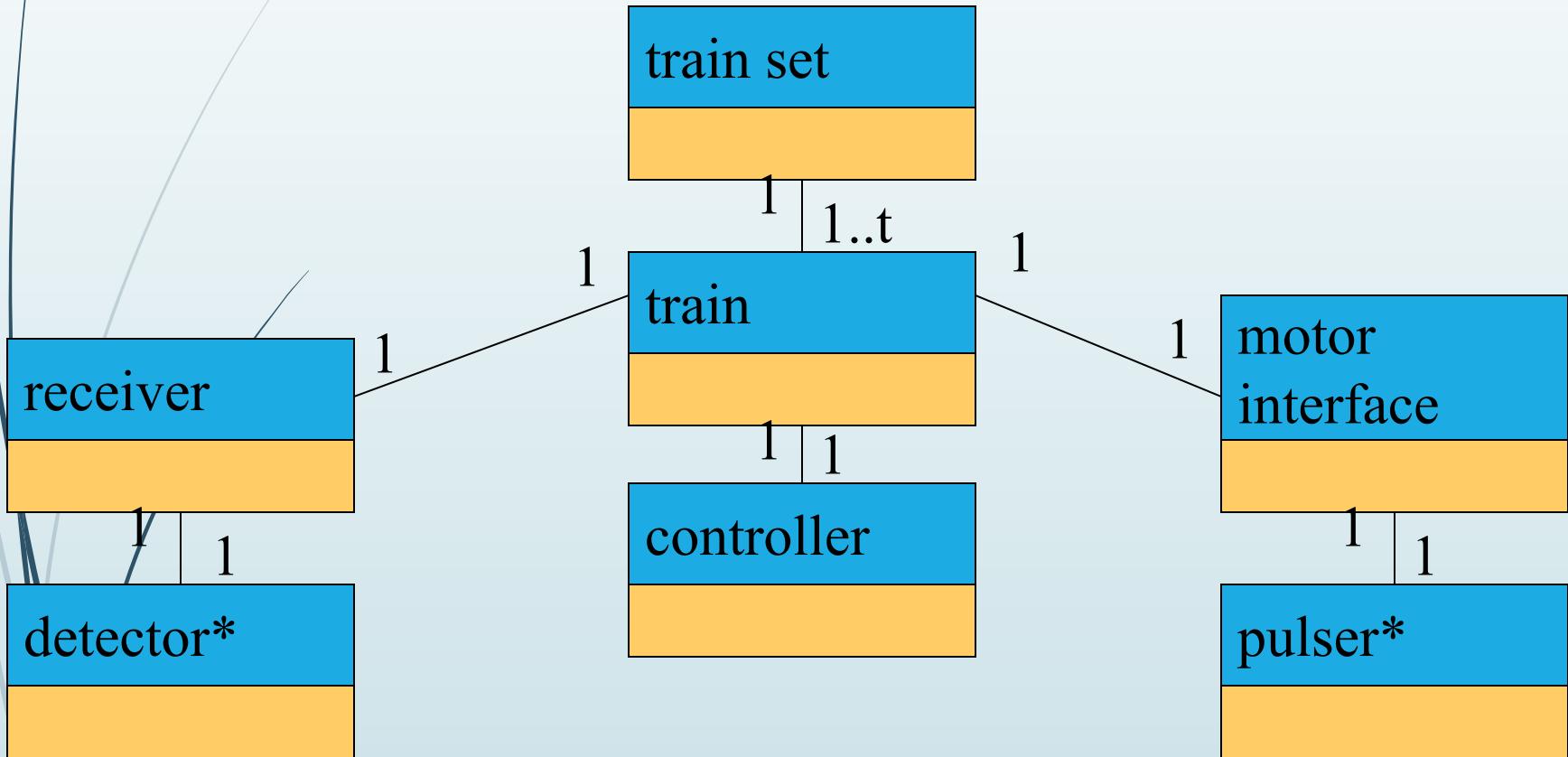
# Console system classes



# Console class roles

- **panel**: describes analog knobs and interface hardware.
- **formatter**: turns knob settings into bit streams.
- **transmitter**: sends data on track.

# Train system classes



# Train class roles

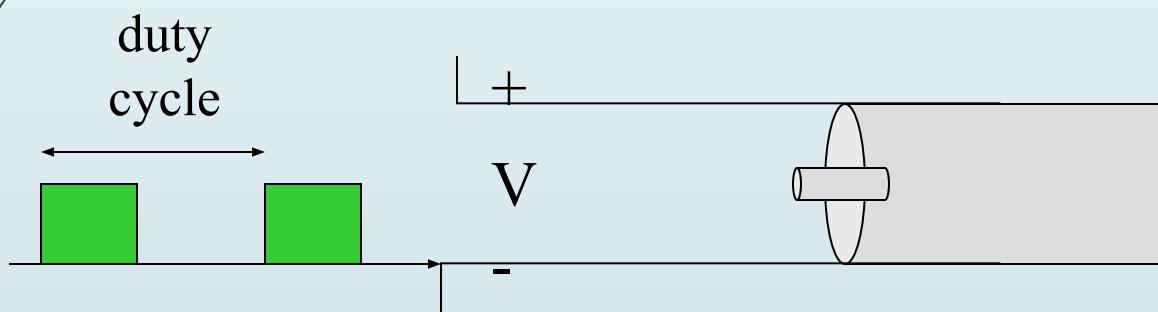
- **receiver**: digitizes signal from track.
- **controller**: interprets received commands and makes control decisions.
- **motor interface**: generates signals required by motor.

# Detailed specification

- We can now fill in the details of the conceptual specification:
  - more classes;
  - behaviors.
- Sketching out the spec first helps us understand the basic relationships in the system.

# Train speed control

- Motor controlled by pulse width modulation:



# Console physical object classes

knobs\*

train-knob: integer  
speed-knob: integer  
inertia-knob: unsigned-integer  
emergency-stop: boolean

pulser\*

pulse-width: unsigned-integer  
direction: boolean

sender\*

send-bit()

detector\*

read-bit() : integer

# Panel and motor interface classes

panel

train-number() :  
integer  
speed() : integer  
inertia() : integer  
estop() : boolean  
new-settings()

motor-interface

speed: integer

# Class descriptions

- panel class defines the controls.
  - new-settings() behavior reads the controls.
- motor-interface class defines the motor speed held as state.

# Transmitter and receiver classes

transmitter

send-speed(adrs: integer,  
speed: integer)  
send-inertia(adrs:  
integer,  
val: integer)  
set-estop(adrs: integer)

receiver

current: command  
new: boolean

read-cmd()  
new-cmd() : boolean  
rcv-type(msg-type:  
command)  
rcv-speed(val: integer)  
rcv-inertia(val:integer)

# Class descriptions

- transmitter class has one behavior for each type of message sent.
- receiver function provides methods to:
  - detect a new message;
  - determine its type;
  - read its parameters (estop has no parameters).

# Formatter class

formatter

current-train: integer

current-speed[ntrains]: integer

current-inertia[ntrains]:

    unsigned-integer

current-estop[ntrains]: boolean

send-command()

panel-active() : boolean

operate()

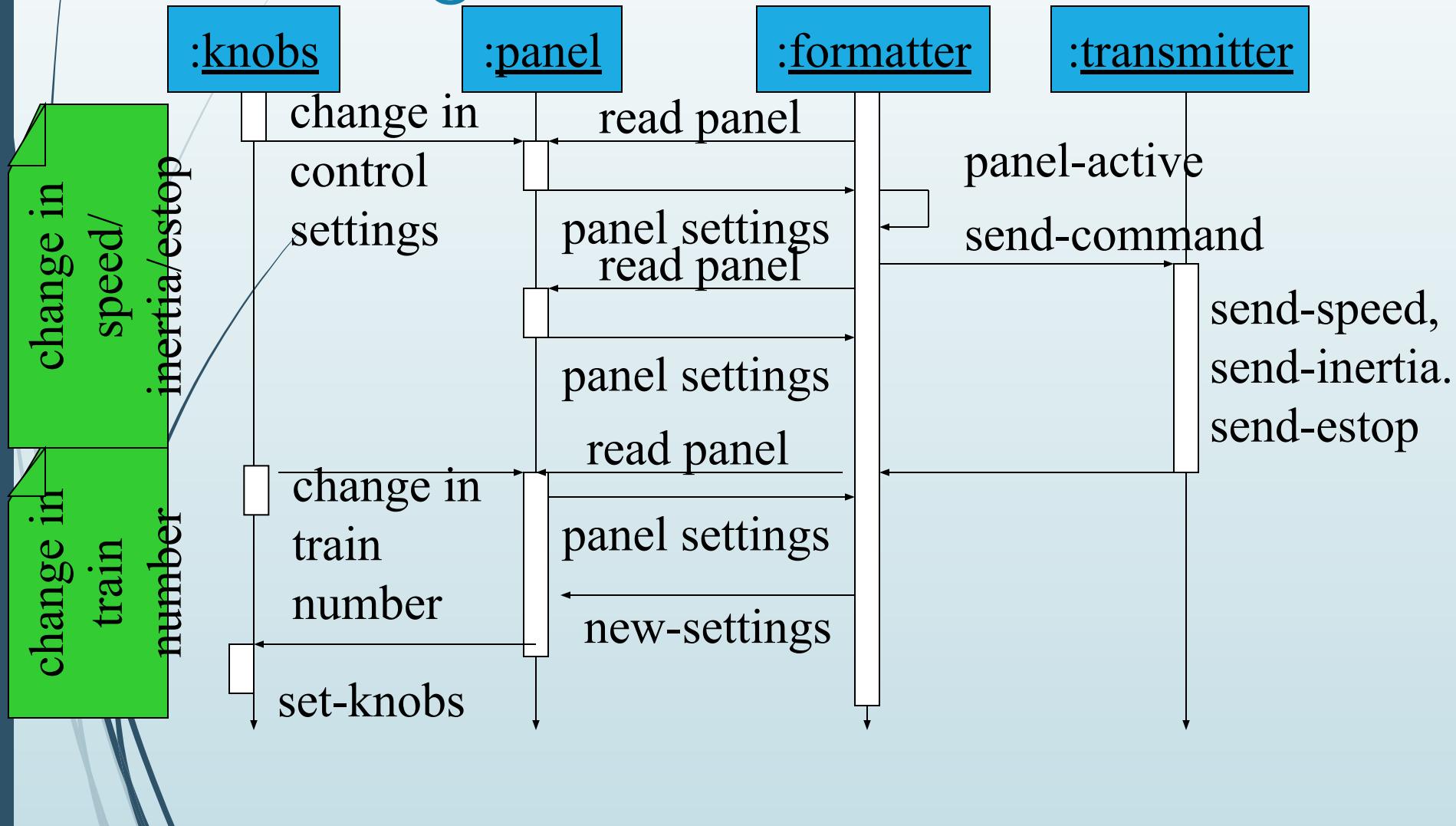
# Formatter class description

- Formatter class holds state for each train, setting for current train.
- The operate() operation performs the basic formatting task.

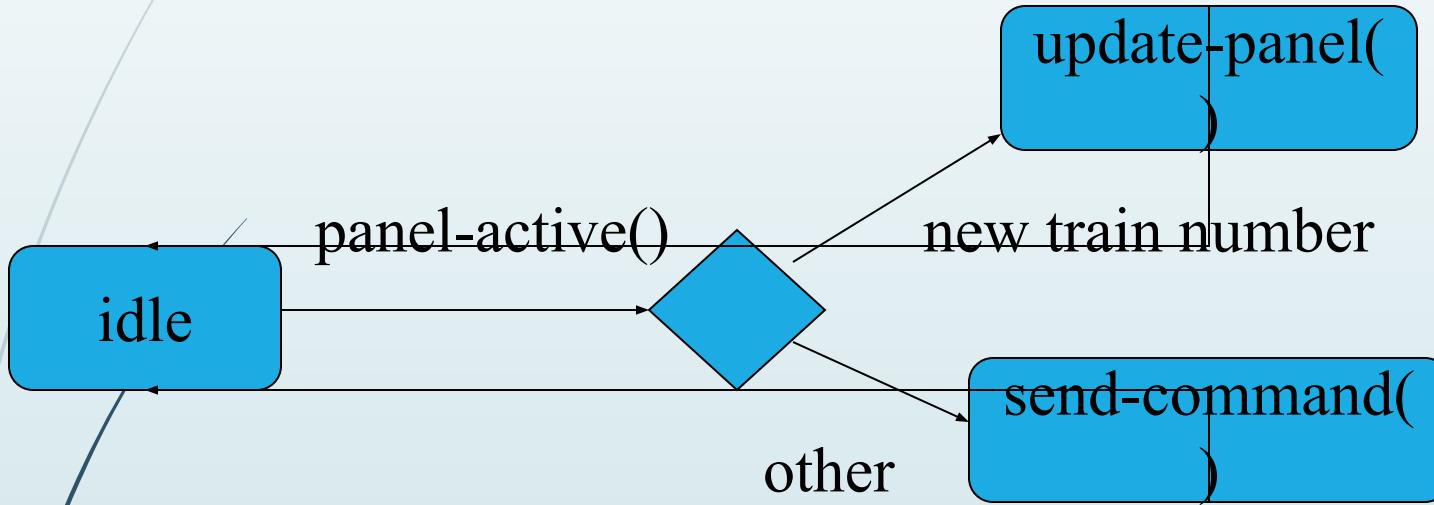
# Control input cases

- Use a soft panel to show current panel settings for each train.
- Changing train number:
  - must change soft panel settings to reflect current train's speed, etc.
- Controlling throttle/inertia/estop:
  - read panel, check for changes, perform command.

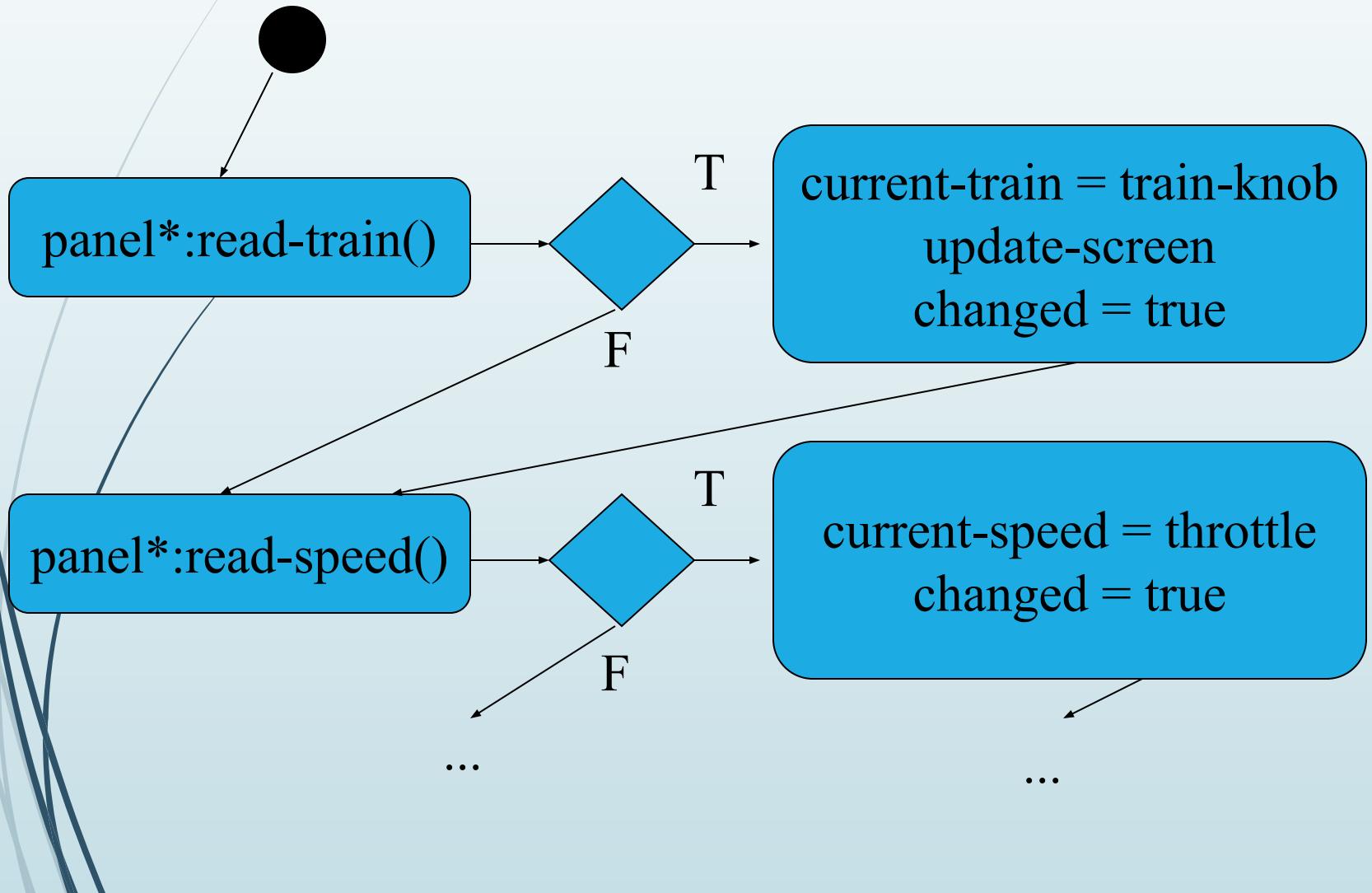
# Control input sequence diagram



# Formatter operate behavior



# Panel-active behavior



# Controller class

controller

current-train: integer

current-speed[ntrains]: integer

current-direction[ntrains]: boolean

current-inertia[ntrains]:  
unsigned-integer

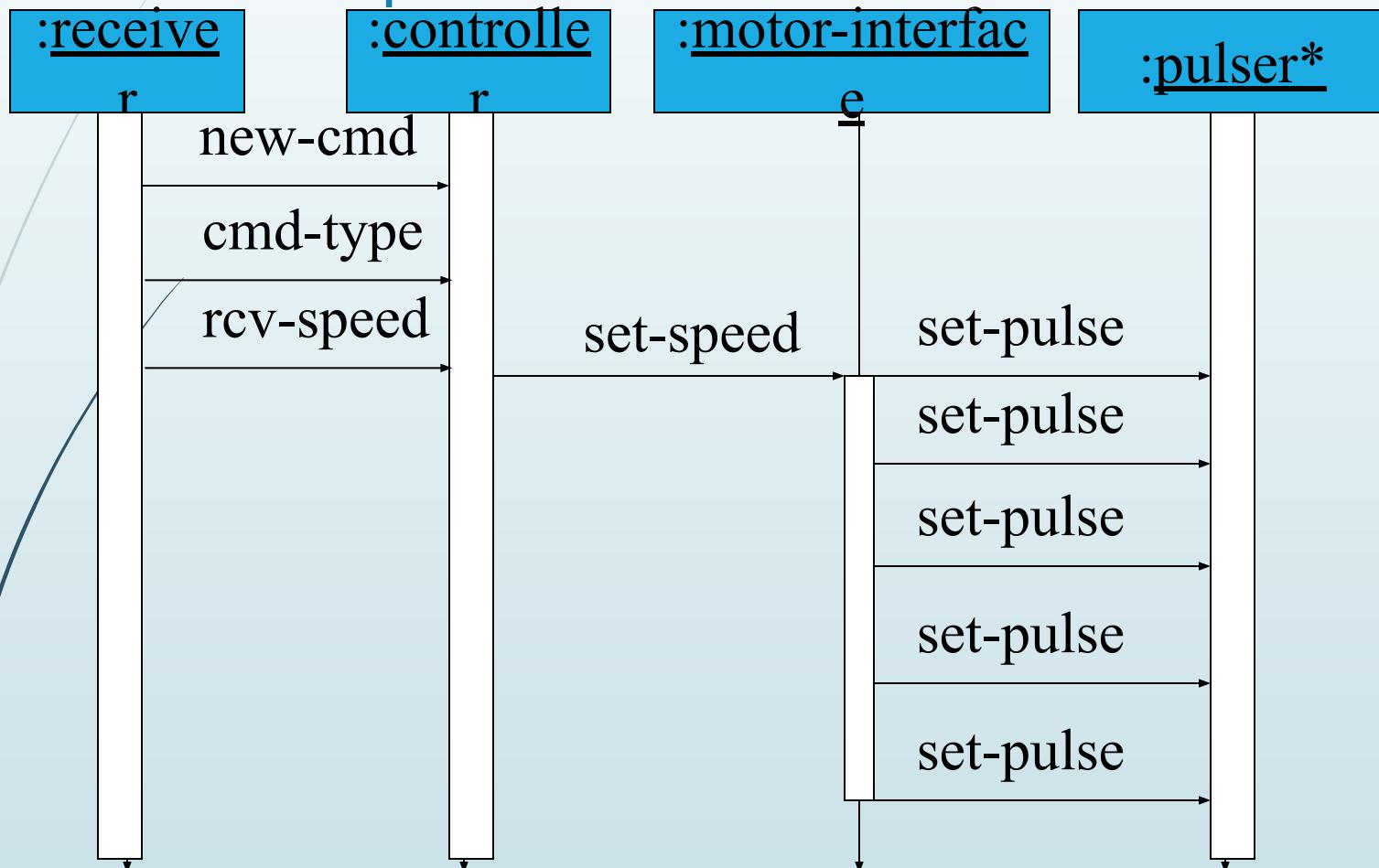
operate()

issue-command()

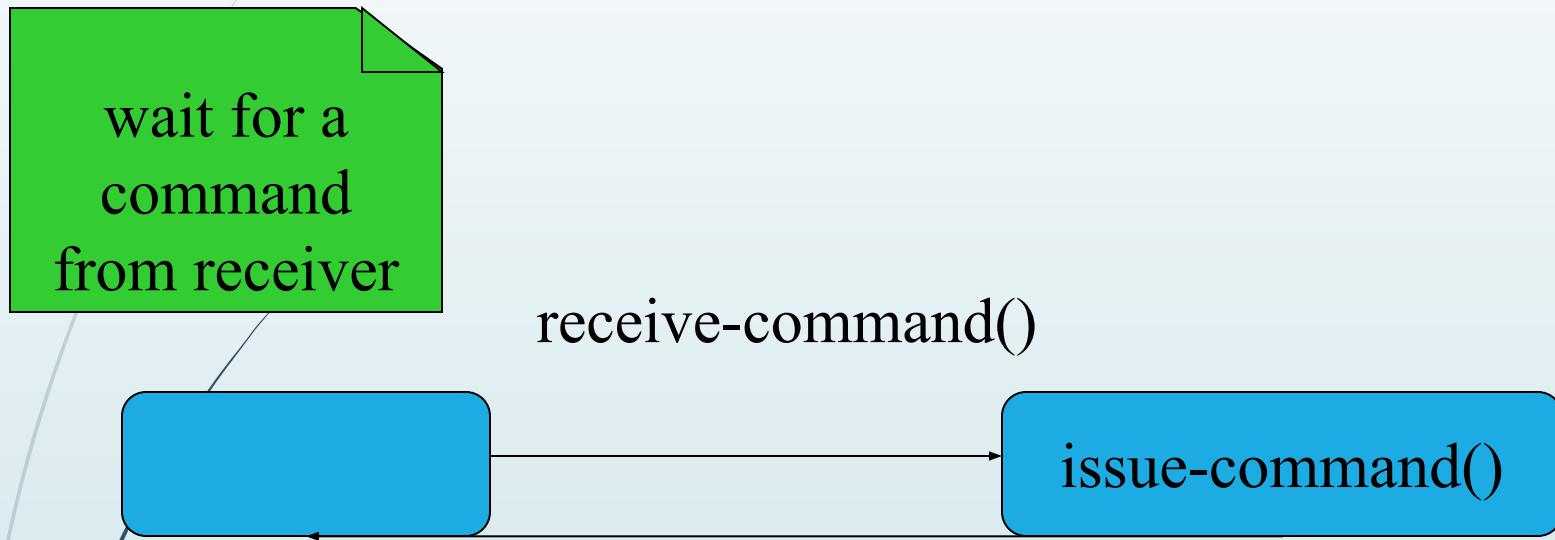
# Setting the speed

- Don't want to change speed instantaneously.
- Controller should change speed gradually by sending several commands.

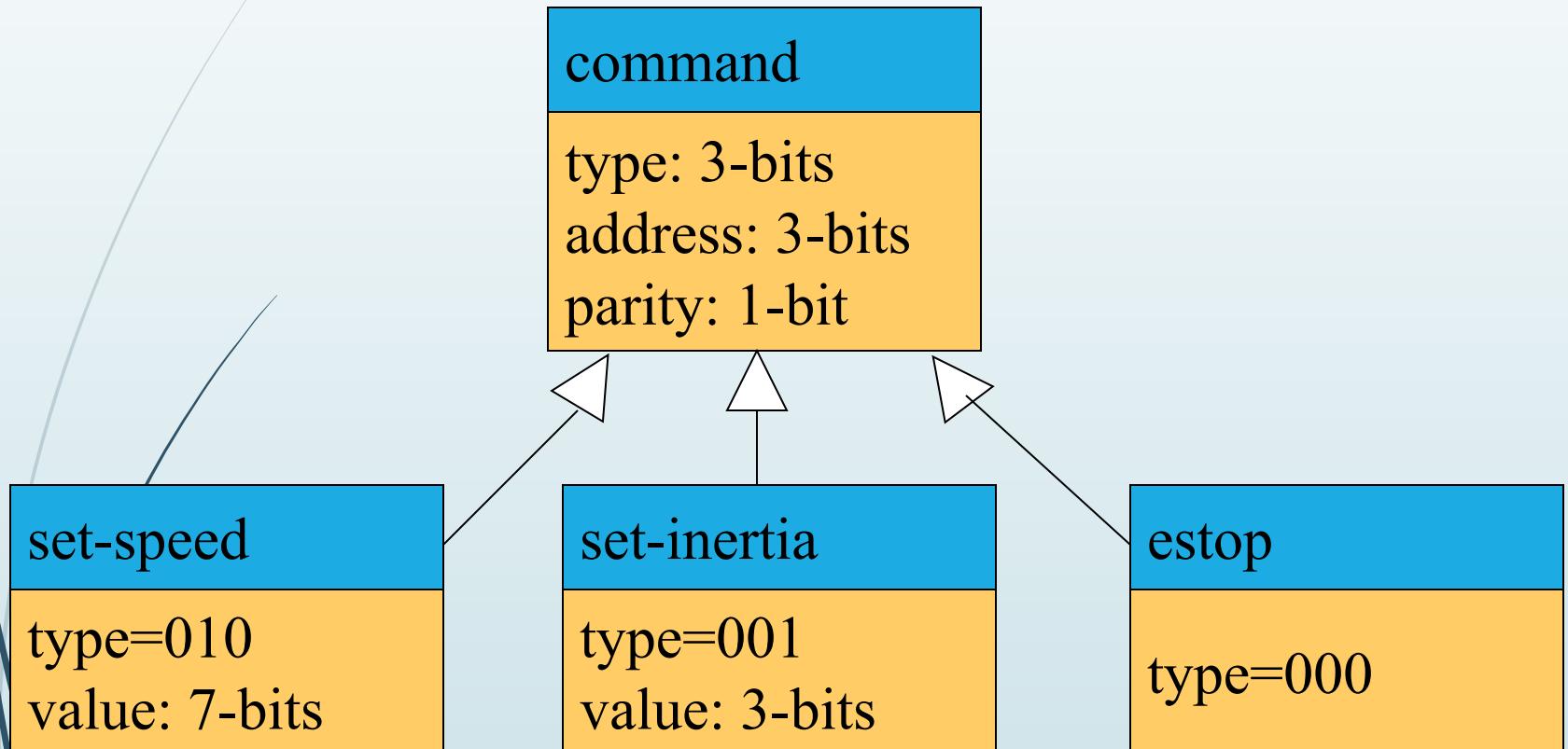
# Sequence diagram for set-speed command



# Controller operate behavior



# Refined command classes



# Summary

- Separate specification and programming.
  - Small mistakes are easier to fix in the spec.
  - Big mistakes in programming cost a lot of time.
- You can't completely separate specification and architecture.
  - Make a few tasteful assumptions.



# Hardware Software Co-Design and Program Modelling

Module 2

# Module II

- Hardware Software Co-Design and Program Modelling
  - Fundamental Issues
  - Computational Models
  - Data Flow Graph
  - Control Data Flow Graph
  - State Machine
  - Sequential Model
  - Concurrent Model
  - Object oriented model
  - UML



# Traditional Embedded System Development Approach

- The hardware software partitioning is done at an early stage
- Engineers from the software group take care of the software architecture development and implementation, and engineers from the hardware group are responsible for building the hardware required for the product
- There is less interaction between the two teams and the development happens either serially or in parallel and once the hardware and software are ready, the integration is performed

# Hardware Software Co-design

- The term 'co-design' implies that hardware and software design be considered concurrent and be done together.
- The product requirement is converted into system level needs.
- Requirements are specified as functional requirements.
- During architectural design phase, system level processing requirement of hardware and software takes place.

# Fundamental Issues in hardware software co design

- Selecting the model
- Selecting the architecture
- Selecting the language
- Partitioning system requirements into hardware and software

# Selecting the model

- Models are used for capturing and describing system characteristics.
- Model is a formal system consisting of objects and composition rules.
- Selecting a model is hard task
- Most designers switch between different models.
- Reasons to switch between models
  - Objective varies in each phase
  - In specification only functionality is important
  - In design the components are important, so models which emphasis the structure should be provided.

# Selecting the Architecture

- Model emphasise only characteristics and not implementation details.
- Architecture gives implementation details in terms of number and types of components and the interconnection between them.
- Commonly used architectures
  - Controller architecture (Application Specific)
  - Datapath architecture (Application Specific)
  - Complex Instruction set Computing (CISC) (General Purpose)
  - Reduced Instruction set computing (RISC) (General Purpose)
  - Very Long Instruction Word Computing (VLIWC) (Parallel Processing)
  - Single Instruction Multiple Data Set (SIMD) (Parallel Processing)
  - Multiple Instruction Single Data Set (MIMD) (Parallel Processing)

# Controller Architecture

- Implements finite state machine model
- Uses state register and two combinational circuits
- State register holds present state
- Combinational circuit implements the logic for the next state and output

# Data path Architecture

- Suited for implementing data flow graph model
- Output is generated as a result of a set of predefined computation on input data
- Data path is a channel between input and output
- Data path contains registers, counters, register files, memories and ports along with high speed arithmetic units.
- Ports connects data path to multiple buses
- Arithmetic units are connected in parallel to improve the performance.

# Finite State Machine Data Path

- FSMD combines controller architecture with data path architecture.
- Controller generates control input and data path processes the data.
- Data path contains two types of input output ports
  - One is control port for receiving and sending control signals to and from control units
  - Second one is I/O port interface with the data path to the outside world for data input and output.

# Complex Instruction set Computing

- Uses instruction set for doing complex operations
- Complex instruction will reduce memory access and program memory size requirements
- It requires additional silicon for implementing micro code decoder for decoding instructions.
- Data path is also complex



# Reduced Instruction Set Architecture

- Uses instruction set representing simple operations
- Complex operations can be performed using multiple RISC instructions.
- Data path contains large number of registers to store data and output
- It supports excessive pipelining

# Very Long Instruction Word

- Architecture supports multiple functional units(ALU, multipliers) in the data path.
- Processes one instruction per functional units

# Parallel Processing Architecture

- Implements multiple concurrent processing elements
- Each one is connected with a separate data path containing register and local elements.
- Eg: SIMD, MIMD
- SIMD:
  - single instruction is executed in parallel
  - The scheduling of instruction execution and controlling of each PE is done through single controller
- MIMD
  - Executed different instructions parallelly
  - It is the basis of multiprocessor system.
  - PEs in multiprocessor system communicates through shared memory and message passing.

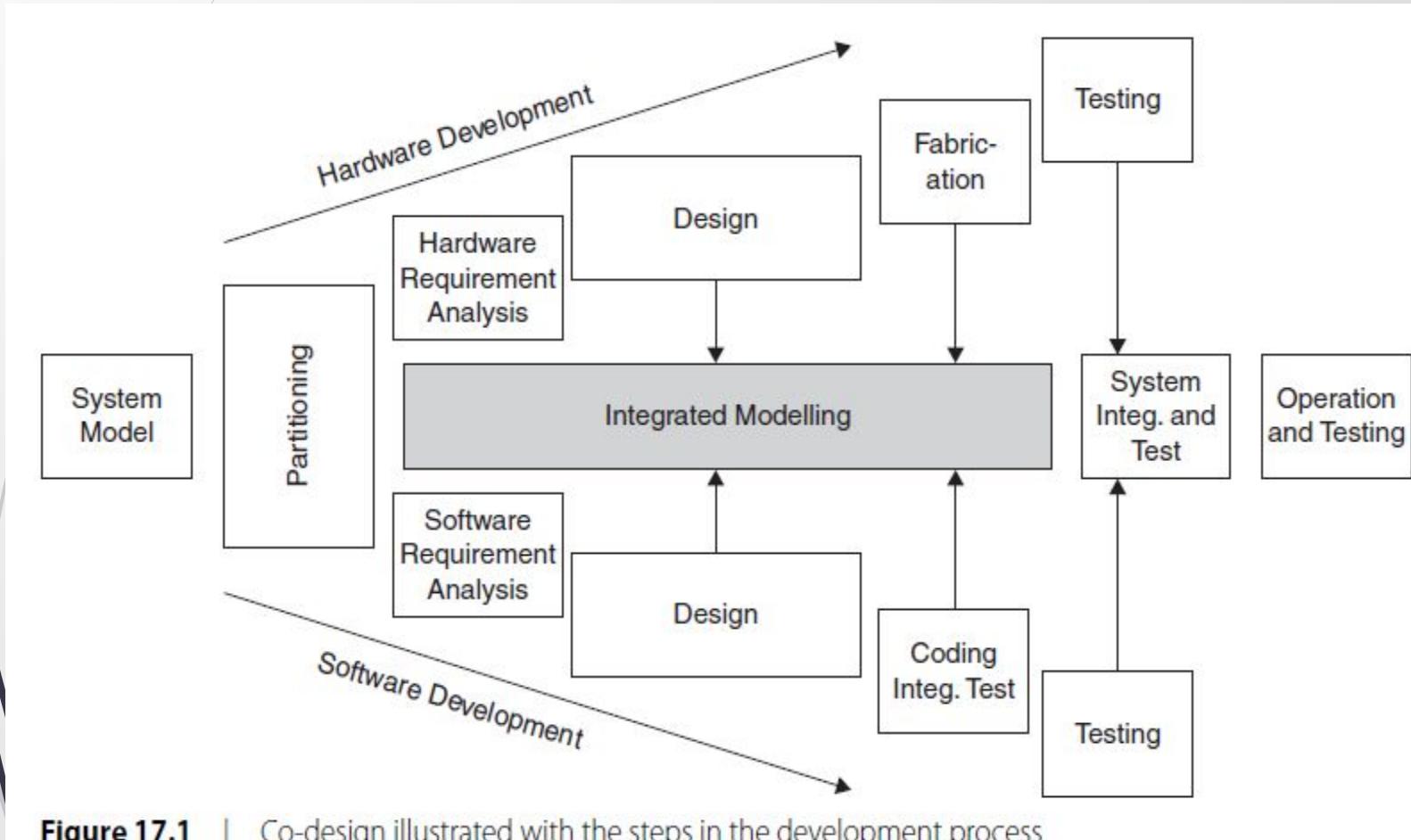
# Language Selection

- A programming Language captures a ‘Computational Model’ and maps it into architecture
- A model can be captured using multiple programming languages like C, C++, C#, Java etc for software implementations and languages like VHDL, System C, Verilog etc for hardware implementations
- Certain languages are good in capturing certain computational model. For example, C++ is a good candidate for capturing an object oriented model.
- The only pre-requisite in selecting a programming language for capturing a model is that the language should capture the model easily

# Partitioning of System Requirements into H/w and S/w

- Implementation aspect of a System level Requirement
- It may be possible to implement the system requirements in either hardware or software (firmware)
- Various hardware software trade-offs like performance, re-usability, effort etc are used for making a decision on the hardware-software partitioning

# Steps in Co-design



**Figure 17.1** | Co-design illustrated with the steps in the development process

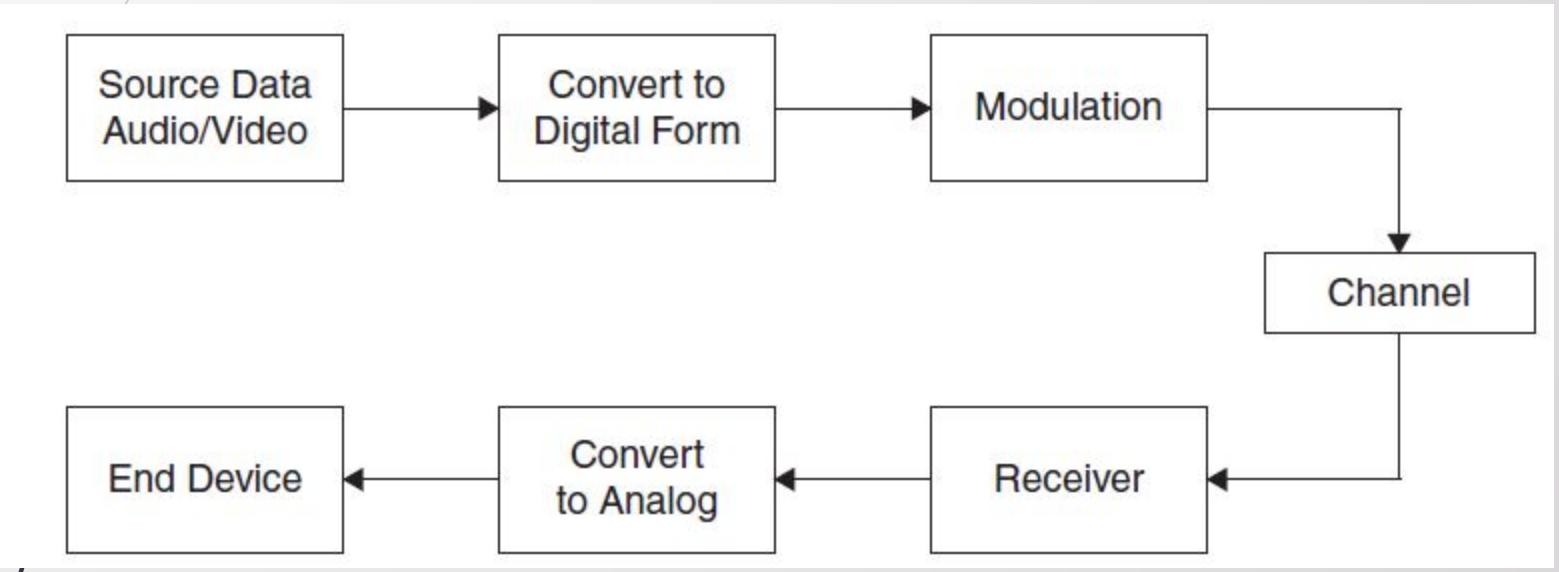


# Computational Models in Embedded Design

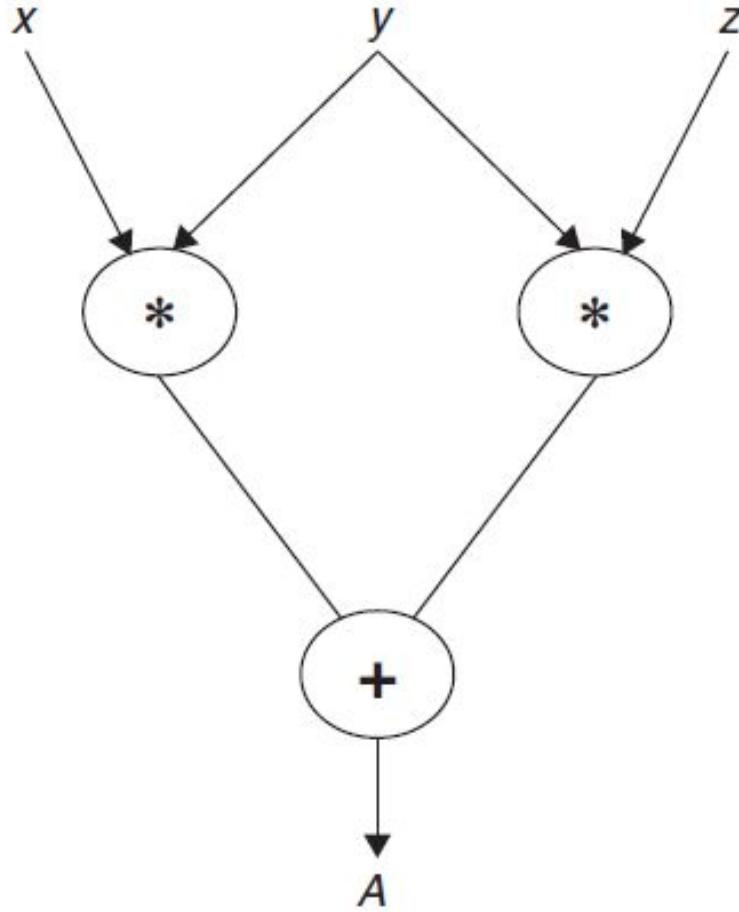
- Data Flow Graph/ Diagram(DFG) Model
- Control/Data Flow Graph
- Finite State Machine
- Sequential process Model
- Concurrent Process Model
- Object Oriented Model
- UML

# Data Flow Graph/ Diagram(DFG) Model

- Translates the data processing requirements into a data flow graph
- A data driven model in which the program execution is determined by data.
- Emphasizes on the data and operations on the data which transforms the input data to output data.
- A visual model in which the operation on the data (process) is represented using a block (circle) and data flow is represented using arrows. An inward arrow to the process (circle) represents input data and an outward arrow from the process (circle) represents output data in DFG notation
- Best suited for modeling Embedded systems which are computational intensive (like DSP applications) .



**Figure 17.2** | Flow of data through a digital communications system



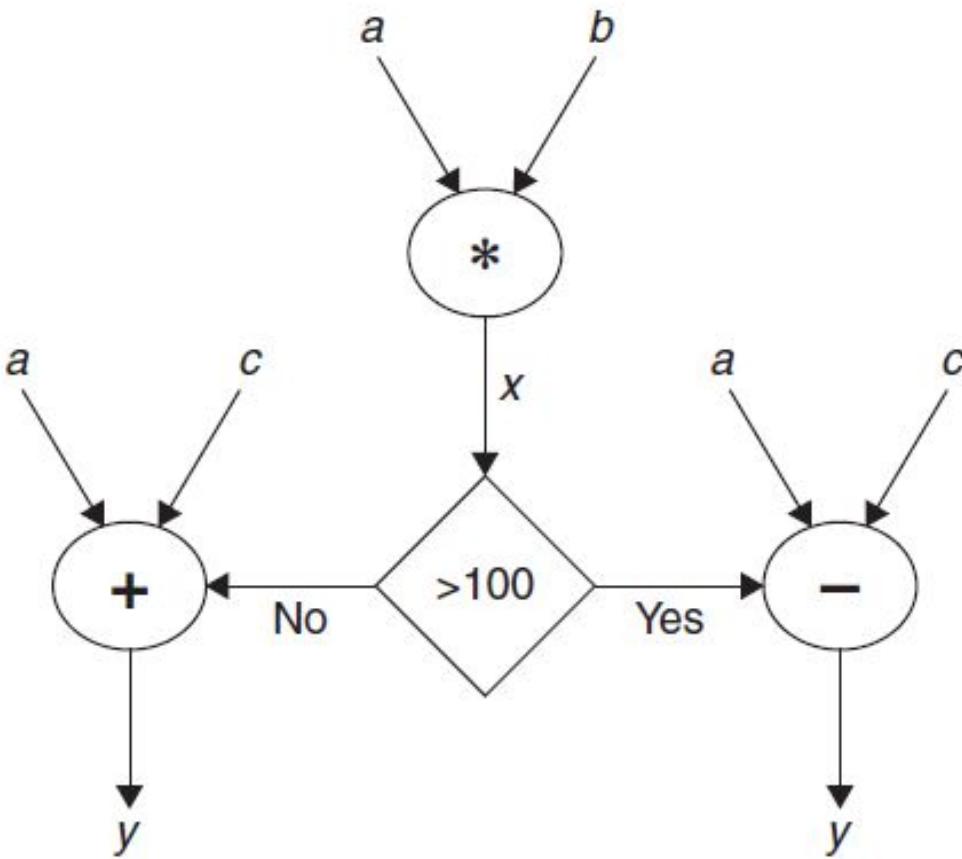
**Figure 17.3** | Data flow diagram for  $A = xy + yz$

- 
- A DFG model is acyclic if it doesn't contain multiple values for input variables and multiple values for output for the same input.
  - A DFG model translates program as a single sequence of execution.

# Control/Data Flow Graph

- Translates the data processing requirements into a data flow graph
- Model applications involving conditional program execution
- Contains both data operations and control operations
- Uses Data Flow Graph (DFG) as element and conditional (constructs) as decision makers.
- CDFG contains both data flow nodes and decision nodes, whereas DFG contains only data flow nodes
- A visual model in which the operation on the data (process) is represented using a block (circle) and data flow is represented using arrows. An inward arrow to the process (circle) represents input data and an outward arrow from the process (circle) represents output data in DFG notation.

- The control node is represented by a ‘Diamond’ block which is the decision making element in a normal flow chart based design
- Translates the requirement, which is modeled to a concurrent process model
- The decision on which process is to be executed is determined by the control node
- Capturing of image and storing it in the format selected (bmp, jpg, tiff, etc.) in a digital camera is a typical example of an application that can be modeled with CDFG



**Figure 17.4** | A data flow graph with control

# State Machine Model

- Used for modeling reactive or event driven modeling system whose processing behavior is dependent on state transition system.
- Based on 'States' and 'State Transition'
- Describes the system behavior with 'States', 'Events', 'Actions' and 'Transitions'
- State is a representation of a current situation.
- An event is an input to the state. The event acts as stimuli for state transition.
- Transition is the movement from one state to another.
- Action is an activity to be performed by the state machine.

- A Finite State Machine (FSM) Model is one in which the number of states are finite.
- In other words the system is described using a finite number of possible states
- Most of the time State Machine model translates the requirements into sequence driven program

# Example: Seat Belt Warning in automotive

- When the vehicle ignition is turned on and the seat belt is not fastened within 10 seconds of ignition ON, the system generates an alarm signal for 5 seconds.
- The alarm is turned off when the alarm time expires or if the driver fastens the belt or if the alarm ignition switch is turned off whichever happens first.
- States are marked as circles and events are marked with arrows between the states.

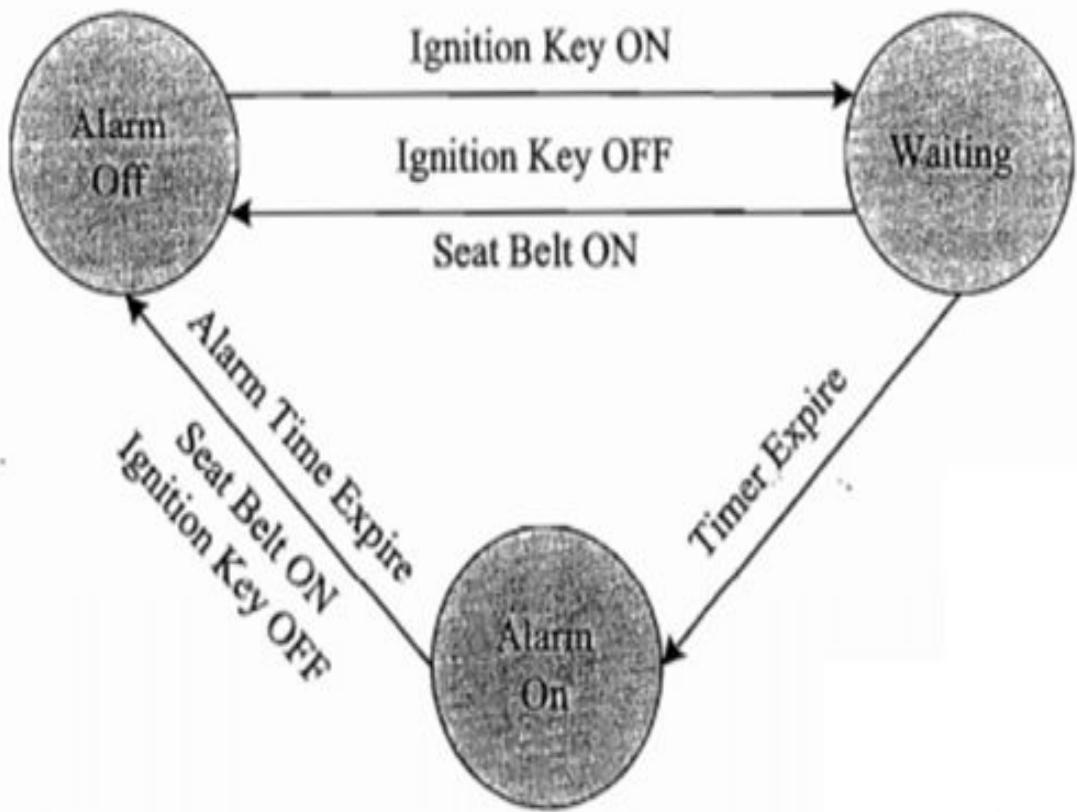
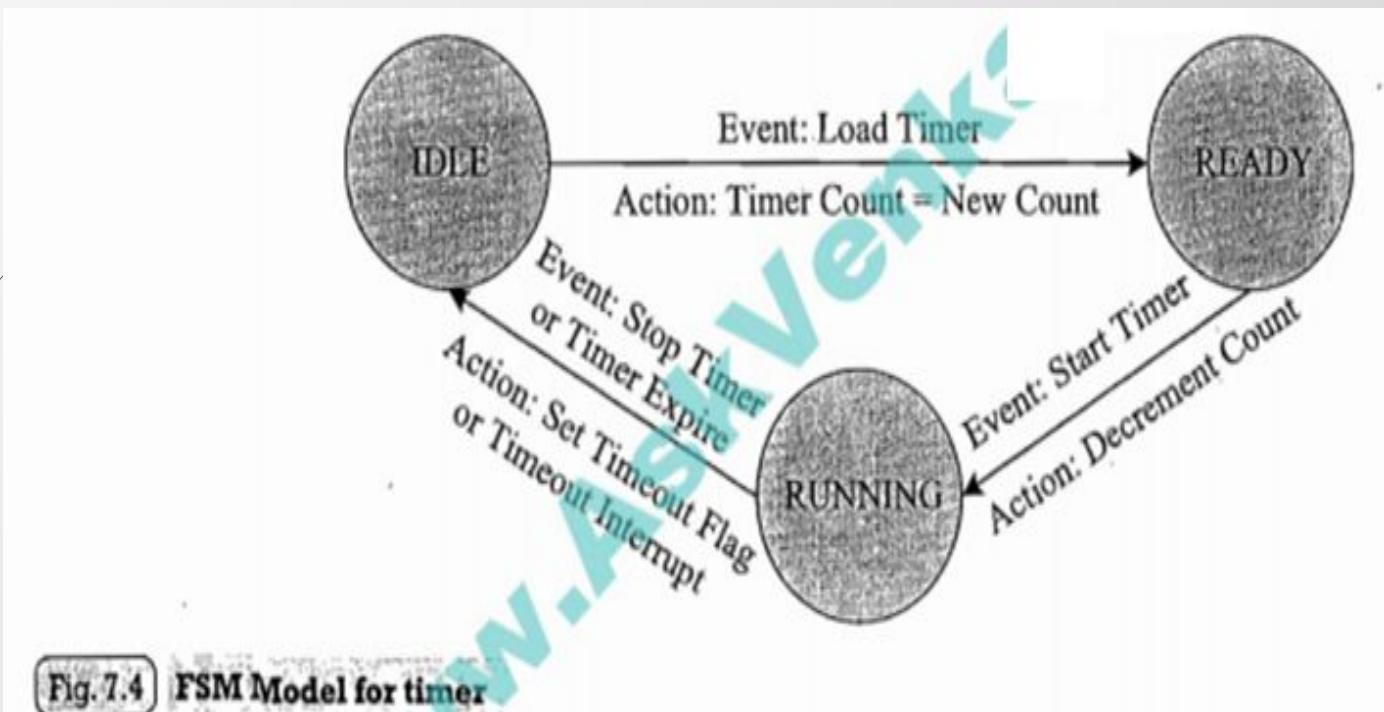


Fig. 7.3

**FSM Model for Automatic seat belt warning system**

# Example : Timer



# Exercise:

- Design a tea/coffee vending machine based on FSM model

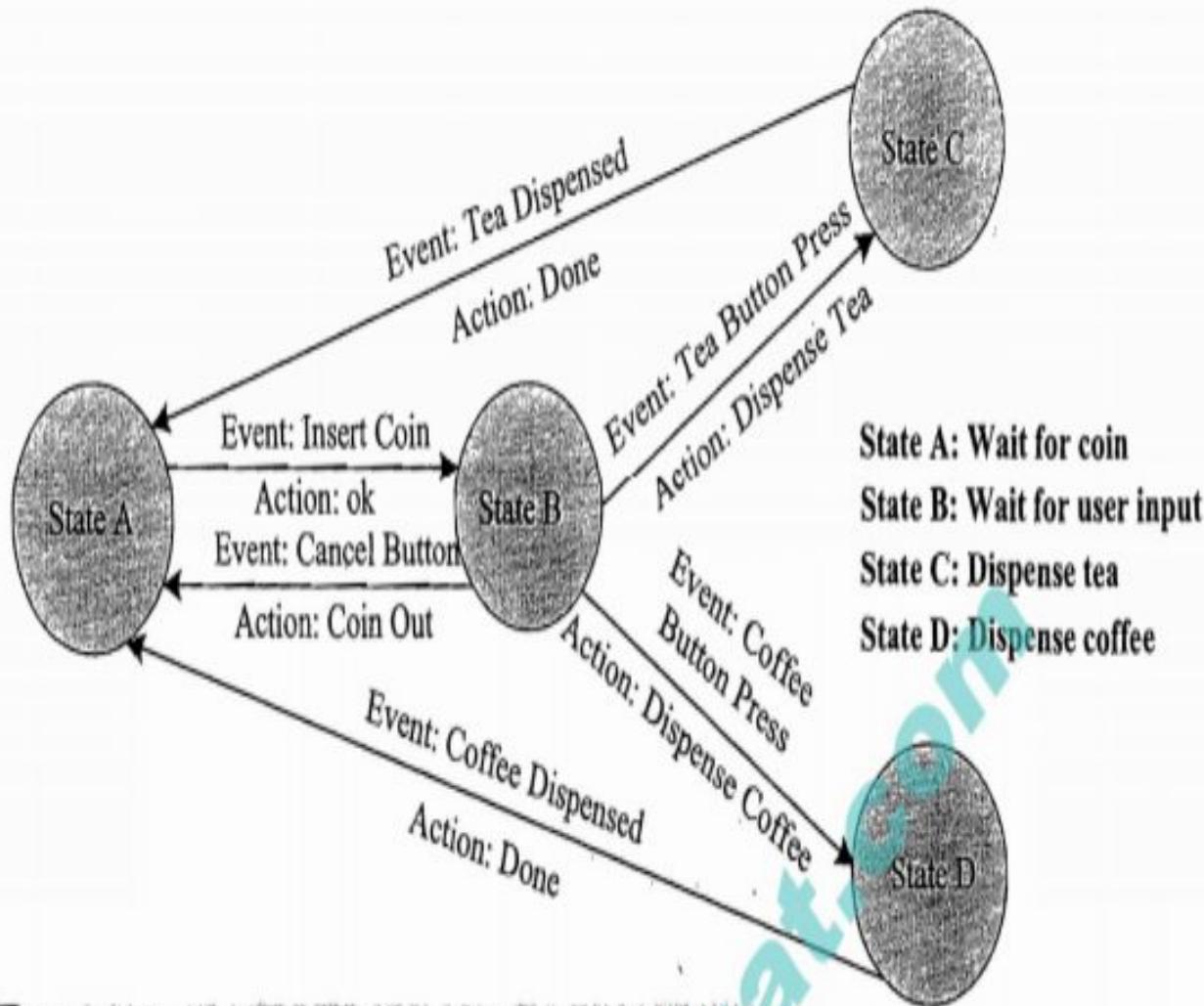


Fig. 7.5 FSM Model for Automatic Tea\Coffee Vending Machine

## Example 3:

- Coin operated Telephone system.

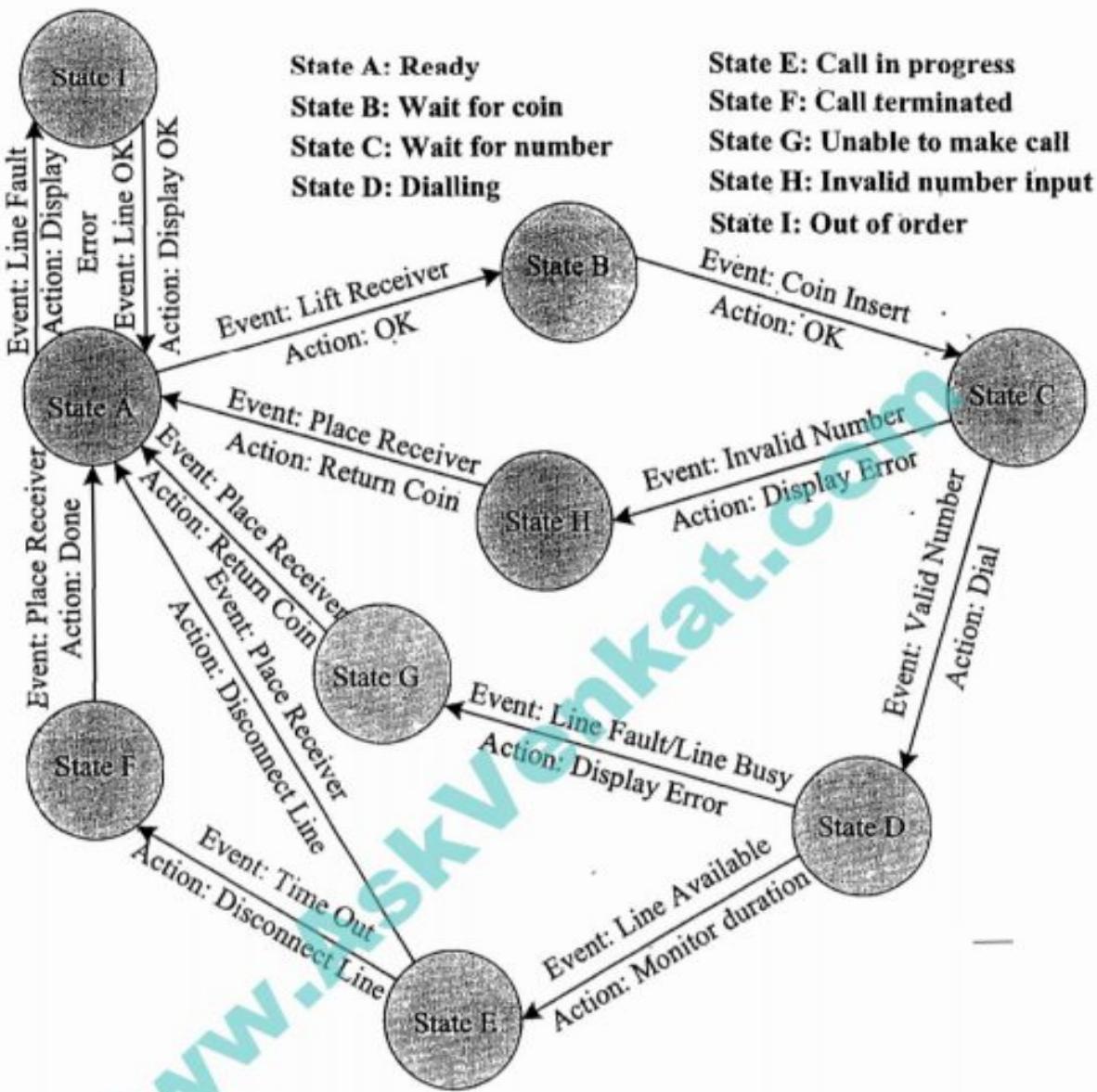


Fig. 7.6 FSM Model for Coin Operated Telephone System

# Sequential Process Model

- In this models the function or processing requirements are executed in sequential order.
- Similar to procedural programming
- FSM and flow charts are tools used for modelling
- FSM represents states and transitions and flow charts represents sequence of execution flow.

# Sequential Process Model example

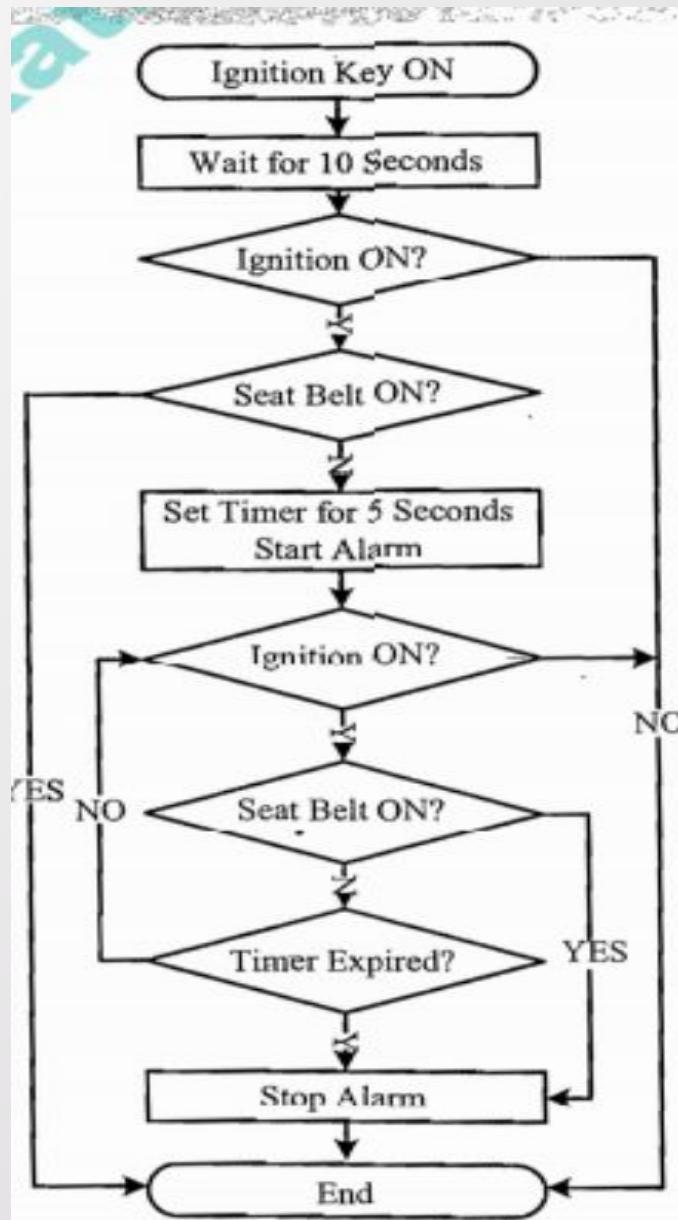


Fig. 7.7 Sequential Program Model for seat belt warning system

# Concurrent Process Model

- It models concurrently executing process.
- Sequential execution leads to poor processor utilization.
- If the task is split into multiple tasks, CPU utilization can be improved as one process goes into waiting stage another subtask can enter into execution.
- But it creates overheads in task synchronization, scheduling and communication.

# Example for CPM(Seat Belt Warning System)

- Task is split into
  - Timer task for waiting 10 seconds
  - Task for checking the ignition key status
  - Task for checking the seat belt status
  - Task for starting and stopping the alarm
  - Alarm timer task for waiting for 5 seconds.

- 
- Among these we cannot execute them randomly or sequentially.
  - We need to synchronize the execution through some mechanism.
  - We need to start the alarm only after the expiration of 10 seconds and only if the ignition key is ON and seat belt is OFF.
  - Wait\_timer\_expires event is associated with timer task event.
  - It is set when the timer expires.

# Concurrent Process Model

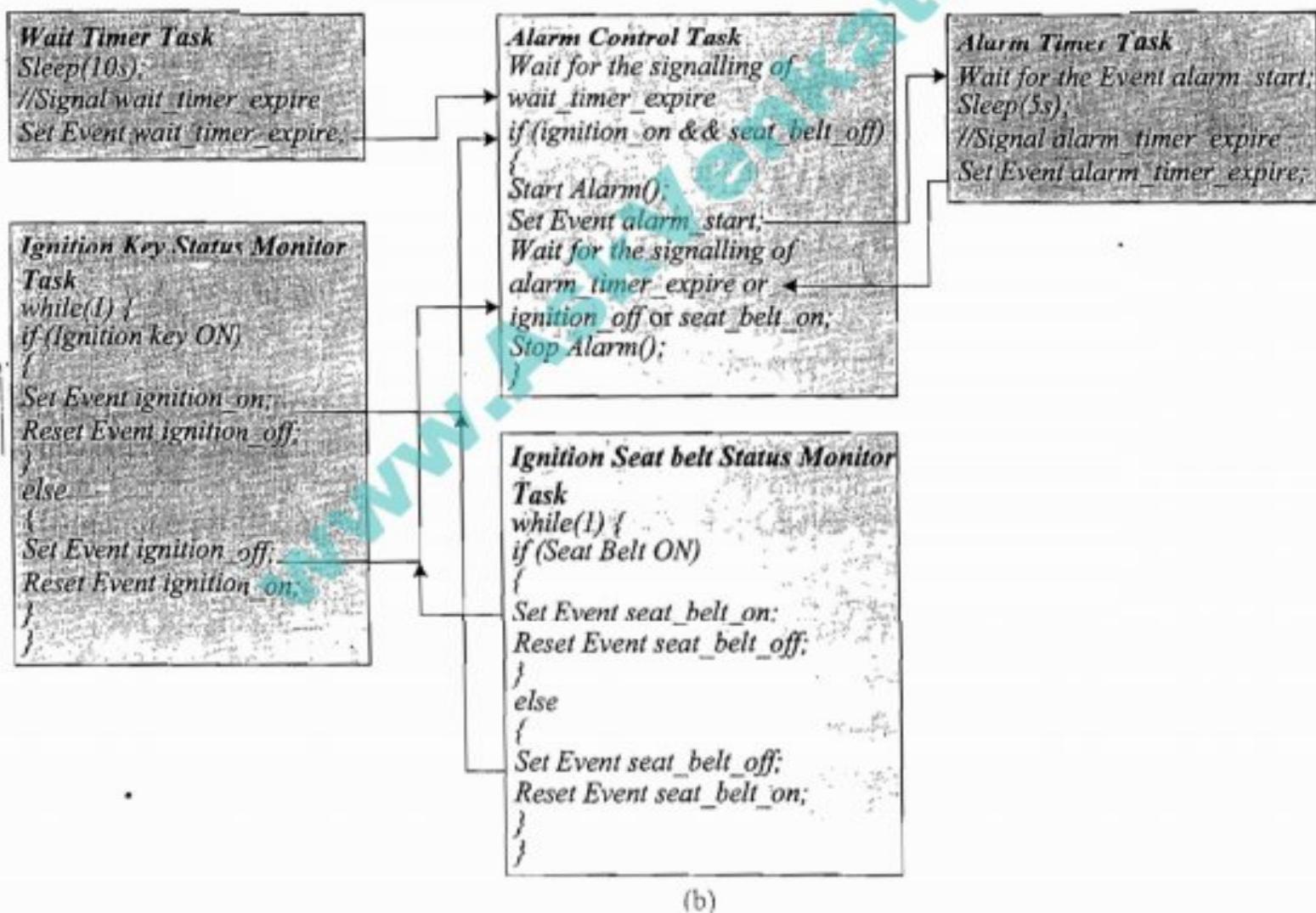
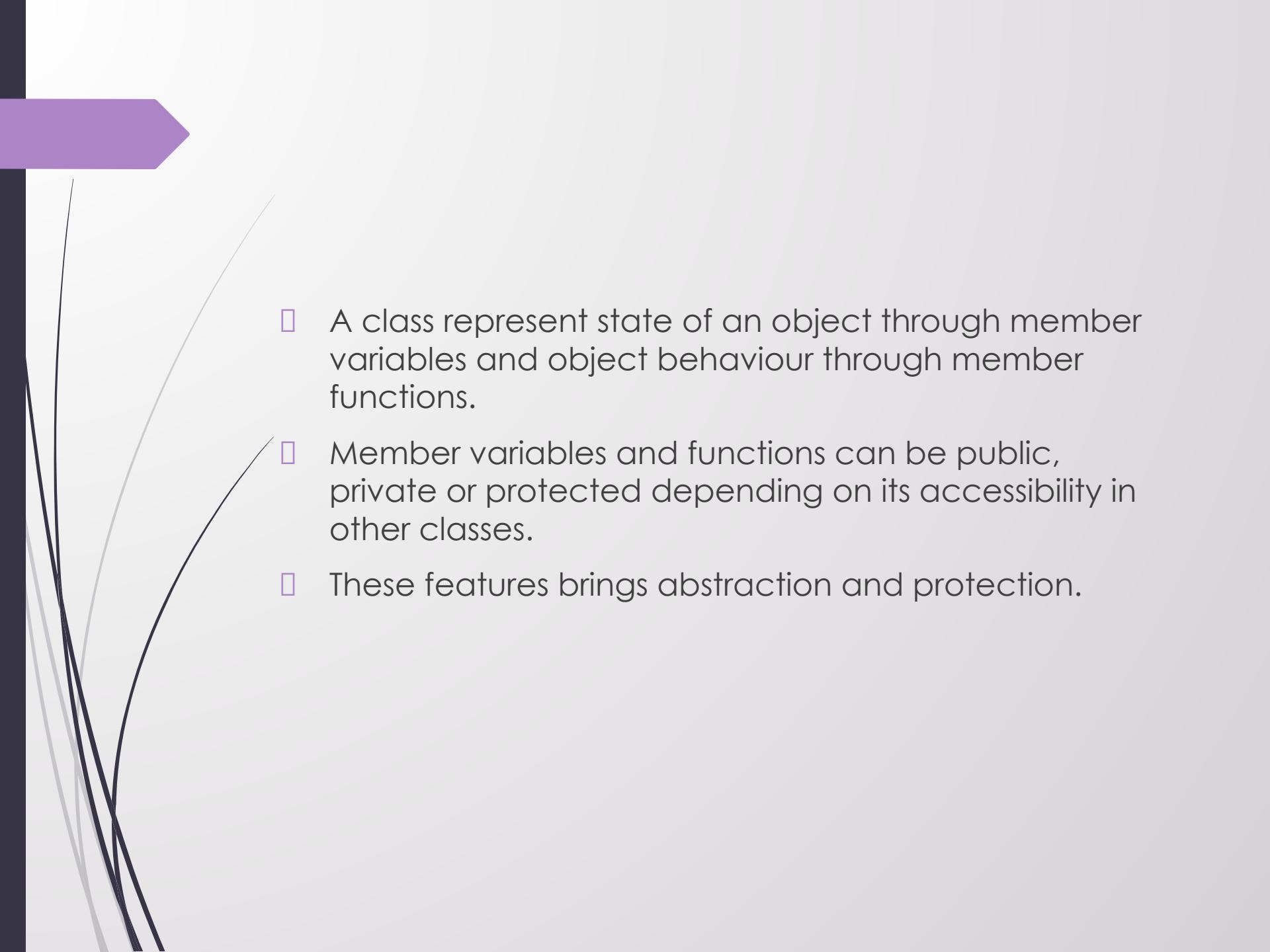


Fig. 7.8 (a) Tasks for 'Seat Belt Warning System' (b) Concurrent processing Program model for 'Seat Belt Warning System'

# Object oriented model

- It divides complex software requirements into simple well defined pieces called objects.
- This model brings reusability, productivity and maintainability in system design.
- Object is an entity which models a particular piece in the system
- Each object is characterised by a unique set of behaviours and state.
- Class is abstract representation of a set of objects or called the blue print of an object.

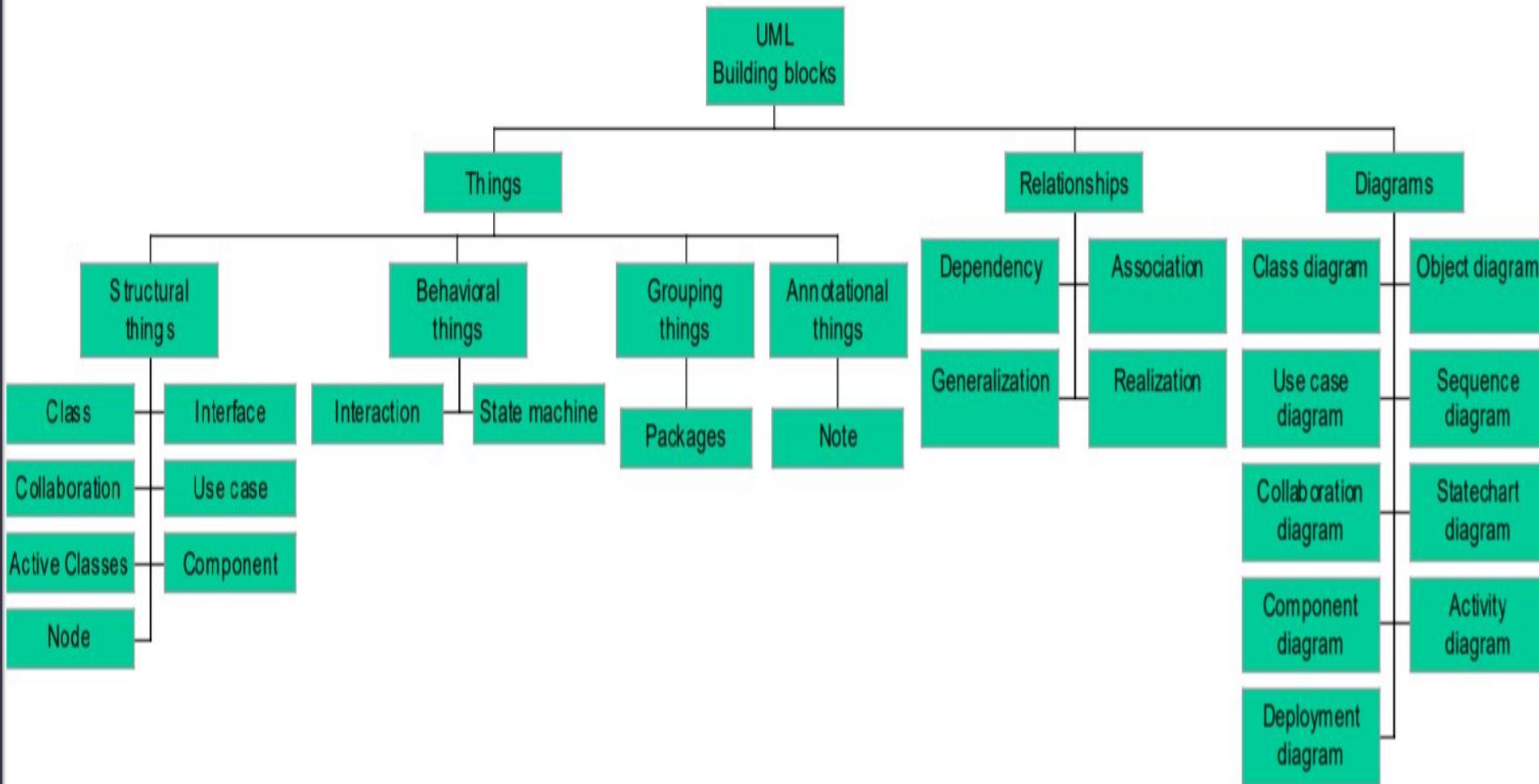
- 
- A class represent state of an object through member variables and object behaviour through member functions.
  - Member variables and functions can be public, private or protected depending on its accessibility in other classes.
  - These features brings abstraction and protection.



# UML

- Unified Modelling Language is a visual modelling language for object oriented design.
- UML helps in all phases of system design through a set of unique diagrams for requirement capturing, designing and deployment.

# UML Building Blocks



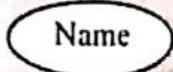
# Four Kinds of Things

- Structural
- Behavioural
- Grouping
- Annotational

# Structural things

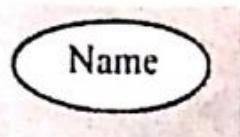
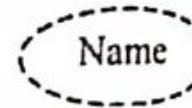
- Represents the static part of UML model
- Also known as classifiers.

# Structural things

| Thing      | Element      | Description   | Representation  |
|------------|--------------|---|---|
| Structural | Class        | A template describing a set of objects which share the same attributes, relationships, operations and semantics. It can be considered as a blueprint of object.                   | Identifier<br>Variables<br>Methods  |
|            | Active Class | Class presenting a thread of control in the system. It can initiate control activity. Active class is represented in the same way as that of a class but with thick border lines. | Identifier<br>Variables<br>Methods  |
|            | Interface    | A collection of externally visible operations which specify a service of a class. It is represented as a circle attached to the class   |  |
|            | Use case     | Defines a set of sequence of actions. It is normally represented with an ellipse indicating the name.   |  |

# Structural things

## Structural things

|  |   |  |   |
|--|---|--|---|
|  | <b>Use case</b>                                 | Defines a set of sequence of actions. It is normally represented with an ellipse indicating the name.  |    |
|  | <b>Collaboration<br/>(Use case Realisation)</b> | Interaction diagram specifying the collaboration of different use cases. It is normally represented with a dotted ellipse indicating the name. |    |
|  | <b>Component</b>                                | Physical packaging of classes and interfaces.  |    |
|  | <b>Node</b>                                     | A computational resource existing at run time.<br>Represented using a cube with name.  |  |

# Behavioural Things

- Represents the dynamic part of UML model
- Interaction, state machine and activity are behavioural things.

| Behavioural | Interaction   | Behaviour comprising a set of objects exchanging messages to accomplish a specific purpose.<br>Represented by arrow with name of operation | Name |
|-------------|---------------|--|------|
|             | State Machine | Behaviour specifying the sequence of states in response to events, through which an object traverses during its lifetime.                  | Name |

# Grouping things

- Represents organizational parts in UML model
- Packages and sub systems are grouping things.

Grouping

Package

Organises elements into packages. It is only a conceptual thing. Represented as a tabbed folder with name.

Name

# Annotational things

- It is the explanatory part of UML model.

Annotational

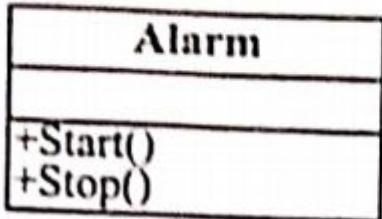
Note

Explanatory element in UML models. Contains formal informal explanatory text. May also contain embedded image.

Text

# Example

a) The Alarm class

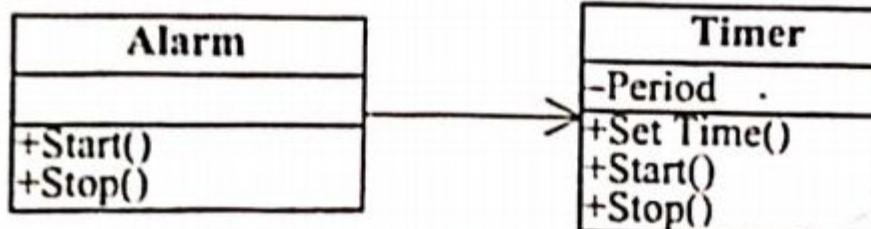


+ - public attribute  
- - private "  
# - protected "  
~ - package "

State representation  
for 'Alarm ON' state

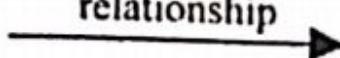
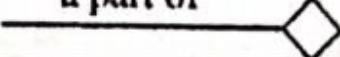
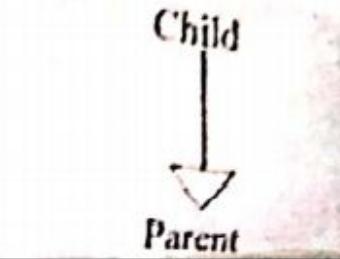
Alarm ON

c) Alarm - Timer class interaction  
for the seat-belt warning sys

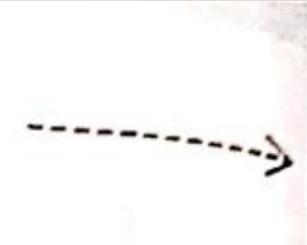
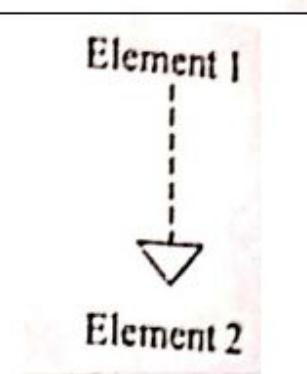


# Relationships

- It represents the relationship between the UML elements.

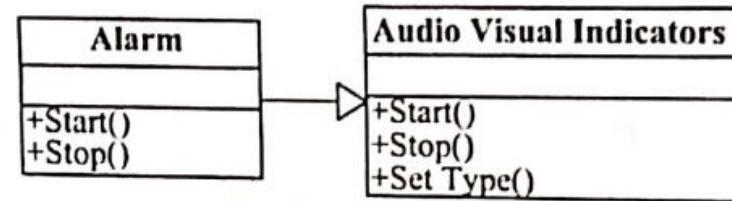
| Relationship   | Description  | Representation  |
|----------------|--|---|
| Association    | <p>It is a structural relationship describing the link between objects. The association can be one-to-one or one-to-many. Aggregation and Composition are the two variants of Association.</p> |    |
| Aggregation    | <p>It represents is "a part of" relationship. Represented by a line with a hollow diamond at the end.</p>  |    |
| Composition    | <p>Aggregation with strong ownership relation to represent the component of a complex object. Represented by a line with a solid diamond at the end.</p>                                       |   |
| Generalisation | <p>Represents a parent-child relationship. The parent may be more generalised and child being specialised version of the parent object.</p>  |  |

# Relationship

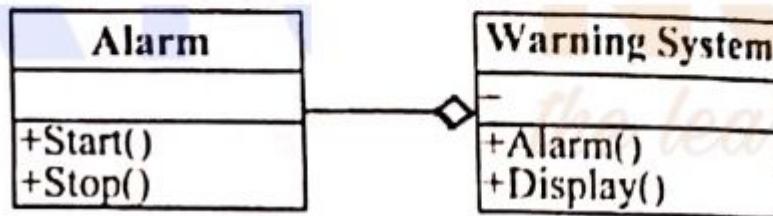
|             |   |   |
|-------------|---|---|
| Dependency  | Represents a relationship in which one element (object, class) uses or depends on another element (object, class). Represented by a dotted arrow with head pointing to the dependent element. |    |
| Realisation | The relationship between two elements in which one element realises the behaviour specified by the other element.   |  <p>Element 1</p> <p>↓</p> <p>Element 2</p> |

# Example

Eg :- a) Alarm is a special type of Audio Visual Indicator (Generalization)



b) Alarm is a part of Warning system. (Aggregation)



# UML Diagrams

- UML diagram gives a pictorial representation of static aspects, behavioural aspects and management of different modules of the system.
- They are grouped into static and behavioural diagrams.

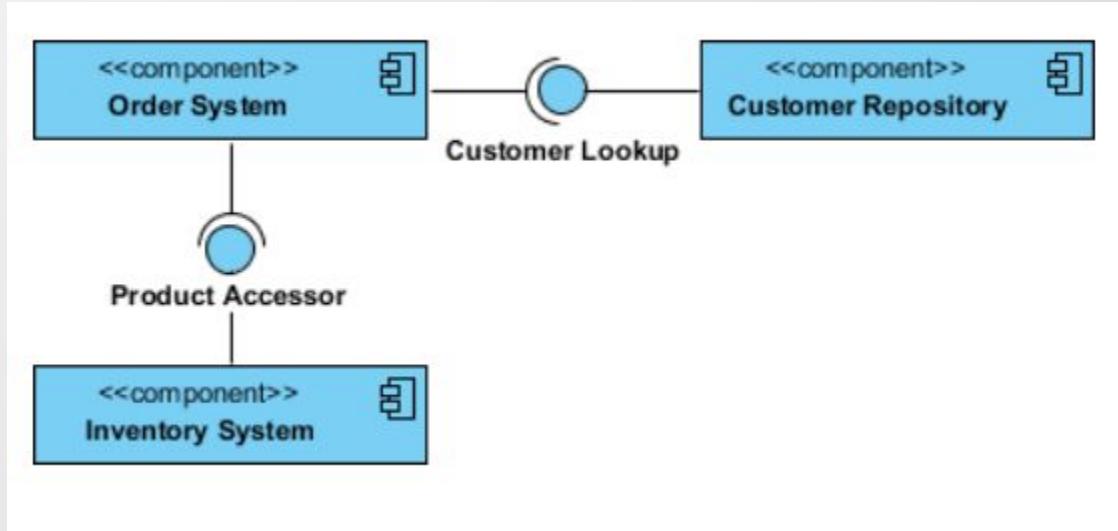
# Static diagrams

- Represents the statistical aspects of the system.
- Different static diagrams are object diagram, class diagram, component diagram, package diagram and deployment diagram.

# Different static diagrams

| Diagram             | Description  |
|---------------------|--|
| Object diagram      | Gives a pictorial representation of a set of objects and their relationships. Represents the structural organization between objects   |
| Class diagram       | Gives a pictorial representation of a set of classes, their interfaces, the collaborations, interactions and relationship between classes. It captures static design of the system.              |
| Component diagram   | It is pictorial representation of the implementation view of the system. It comprises components(physical packaging of classes and interfaces), relationships, and association among components. |
| Deployment diagrams | It is pictorial representation of the configuration of runtime processing nodes and the components associated with them.   |

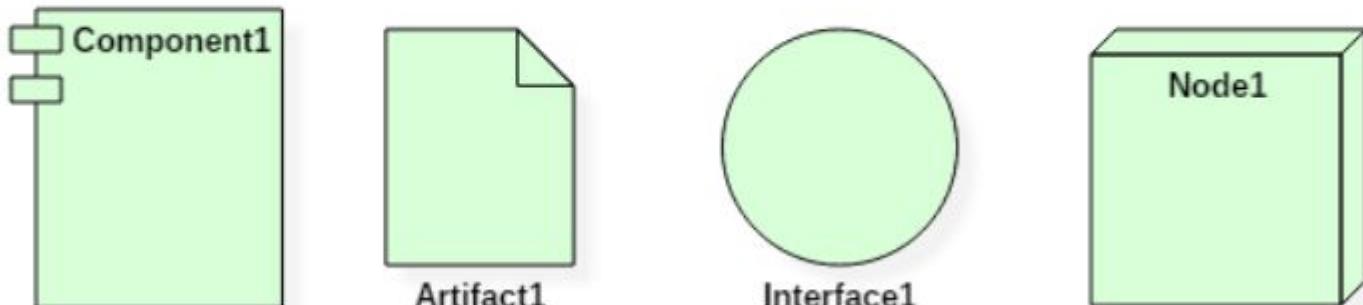
# Example of component diagram



# Deployment diagram

- Deployment diagrams are used with the sole purpose of describing how software is deployed into the hardware system.
- It visualizes how software interacts with the hardware to execute the complete functionality.
- It is used to describe software to hardware interaction

## Deployment Diagram Symbol and notations



Deployment Diagram Notations

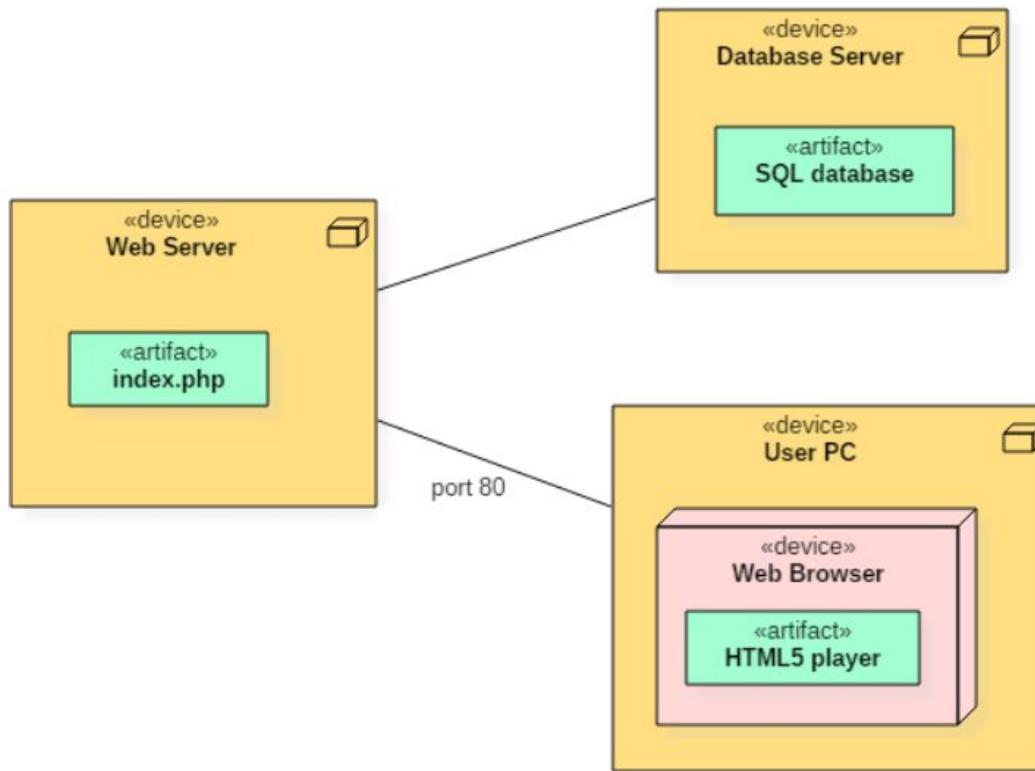


# Notations of deployment diagram

1. A node: Node is a computational resource upon which artifacts are deployed for execution. A node is a physical thing that can execute one or more artifacts.
2. A component
3. An artifact: An artifact represents the specification of a concrete real-world entity related to software development.  
Examples : Source files, Executable files, Database tables
4. An interface

# Example

- Following deployment diagram represents the working of HTML5 video player in the browser:



# Behavioural Diagram

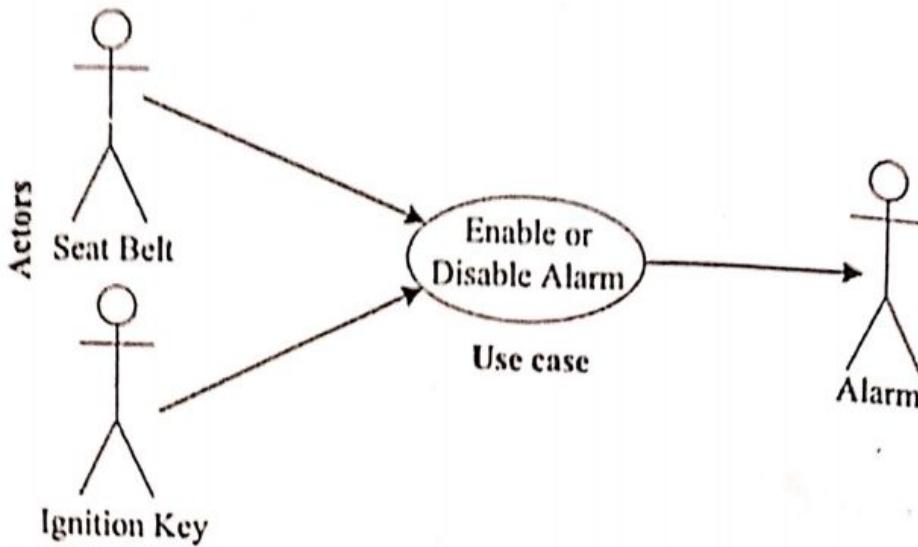
- These diagram represents the dynamic aspects of the system.
- Use case diagram, sequence diagram, collaboration diagram, state chart diagram, and activity diagram etc. are behavioural diagram.

| Diagram               | Description  |
|-----------------------|--|
| Use case diagram      | <ul style="list-style-type: none"> <li>• It is used for describing the system functionality as seen by the user.</li> <li>• Useful for capturing system requirements.</li> <li>• It comprises use cases, actors(users), and the relationship between them.</li> <li>• Actor is one who interacts with the system.</li> <li>• Use case is a sequence of interaction between actors and system.</li> </ul> |
| Sequence diagram      | <ul style="list-style-type: none"> <li>• It is a type of interaction between diagram representing object interaction with respect to time.</li> <li>• It emphasise on the time ordering of messages.</li> <li>• Suitable for interaction modelling of real world system.</li> </ul>  |
| Collaboration diagram | <ul style="list-style-type: none"> <li>• It represents the object interactions and their linking with others.</li> <li>• It gives emphasis to the structural organization of objects that sends and receive messages.</li> </ul>   |

| Diagram             | Description   |
|---------------------|---|
| State chart diagram | <ul style="list-style-type: none"><li>• A diagram showing states, transition, events and activities</li><li>• Suitable for modelling reactive systems</li></ul> |
| Activity diagram    | <ul style="list-style-type: none"><li>• It shows activity transition</li><li>• It emphasis flow of control among objects.</li></ul>                             |

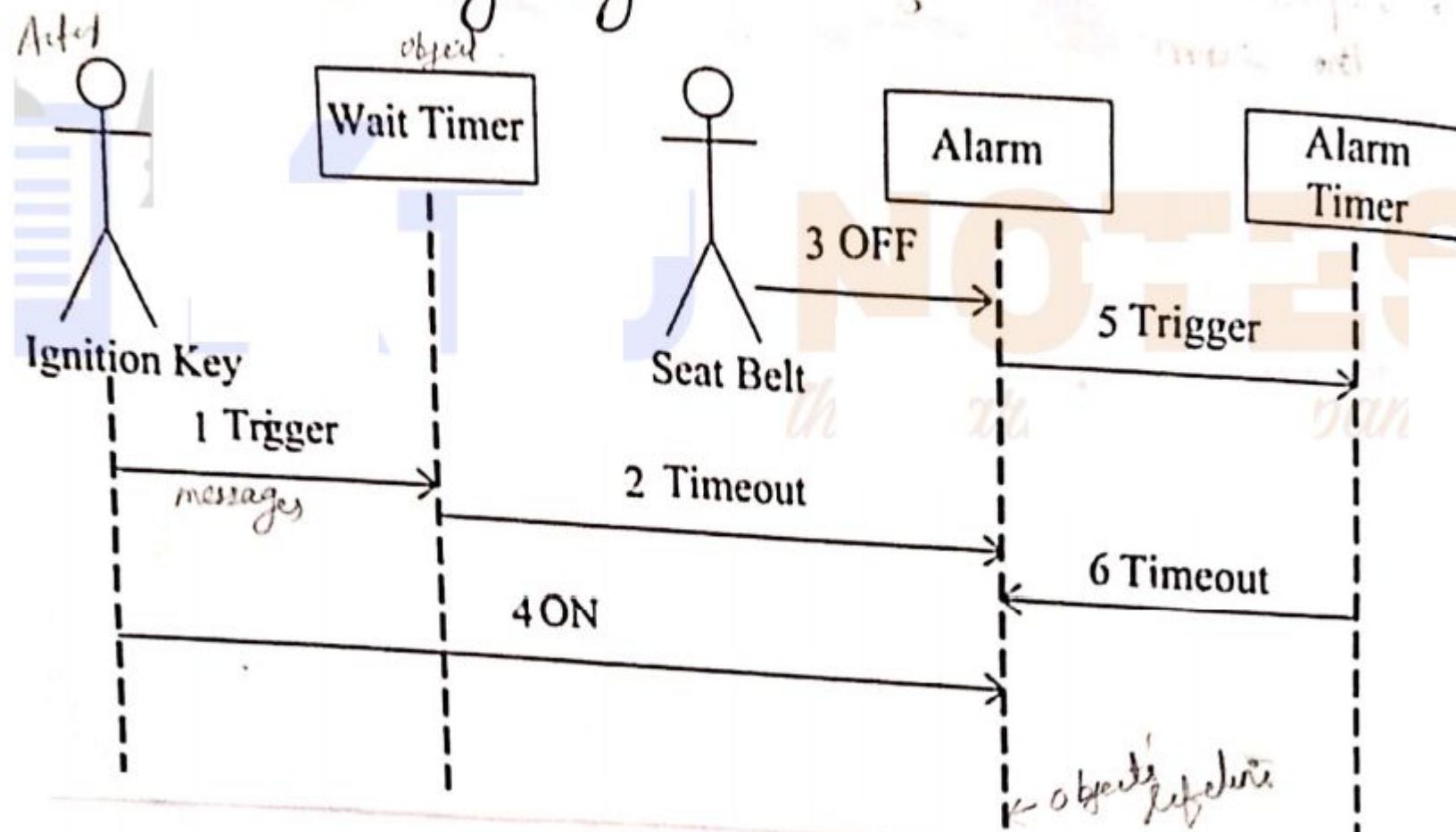
# Use case diagram example

Eg:- Use case diagram for seat belt warning sys



# Sequence diagram Example

Sequence diagram for one possible sequence for the seat belt warning system



# UML Tools

- Used for building UML based models
- Available from different vendors
- Commercial and open source tools are available

| Tool                      | Provider/Comments   |
|---------------------------|---|
| Rational Rose Enterprise  | IBM Software ( <a href="http://www-03.ibm.com/software/products/en/enterprise">http://www-03.ibm.com/software/products/en/enterprise</a> )  |
| Rational System Developer | Eclipse <sup>†</sup> based tool from IBM Software ( <a href="http://www-03.ibm.com/software/products/en/ratisostarch">http://www-03.ibm.com/software/products/en/ratisostarch</a> )   |
| Telelogic Rhapsody        | UML/SysML-based model-driven development tool for real-time or embedded systems from IBM Software ( <a href="http://www-03.ibm.com/software/products/en/ratirhafmii">http://www-03.ibm.com/software/products/en/ratirhafmii</a> ) |
| Borland® Together®        | Borland ( <a href="http://www.borland.com">www.borland.com</a> )  |
| Enterprise Architect      | Sparx Systems ( <a href="http://www.sparxsystems.com">http://www.sparxsystems.com</a> )   |
| ARTiSAN Studio            | Artisan Software Tools Inc ( <a href="http://www.artisansoftwaretools.com/">http://www.artisansoftwaretools.com/</a> )  |
| Microsoft® Visio          | Microsoft Corporation. The Microsoft Visio (Part of Microsoft® Office product) supports UML model Diagram generation. <a href="http://www.microsoft.com/office/visio">www.microsoft.com/office/visio</a>                          |

# MODULE 3



# Design and Development of Embedded Products

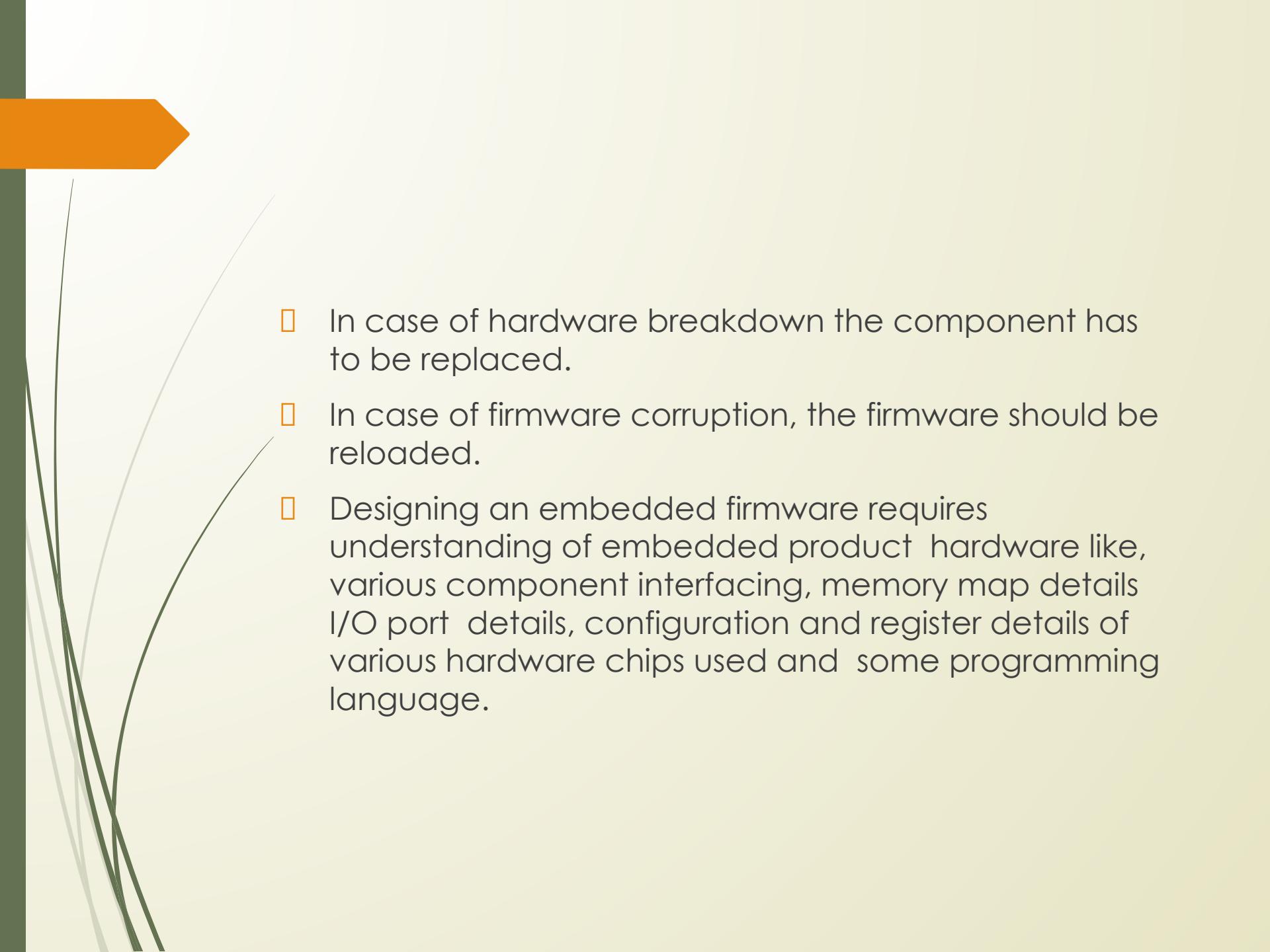


# Topics

- Firmware Design and Development
  - Design Approaches
  - Firmware Development Languages

# INTRODUCTION

- Embedded firmware is responsible for controlling various peripherals of the embedded hardware and generating responses in accordance with the functional requirements mentioned in the requirements for the particular product
- Firmware is considered as the master brain of the embedded systems
- Imparting intelligence to an embedded system is a one time process and it can happens at any stage of the design
- Once the intelligence is imparted to the embedded product, by embedding the firmware in the hardware, the product start functioning properly and will continue serving the assigned task till hardware breakdown occurs or a corruption in embedded firmware occurs

- 
- In case of hardware breakdown the component has to be replaced.
  - In case of firmware corruption, the firmware should be reloaded.
  - Designing an embedded firmware requires understanding of embedded product hardware like, various component interfacing, memory map details I/O port details, configuration and register details of various hardware chips used and some programming language.

- Embedded firmware development process start with conversion of firmware requirements into a program model using modeling tools like UML or flow chart based representation
- Flow charts or UML diagram gives diagrammatic representation of the decision items to be taken and the task to be performed
- Once the program modeling is created, next step is the implementation of the task and actions by capturing the model using a language which is understandable by the target processor

# EMBEDDED FIRMWARE DESIGN APPROACHES

- Firmware design approaches depends on the
  - Complexity of the function to be performed
  - Speed of operation required ..etc
- Two basic approaches for firmware design
  - Conventional Procedure based Firmware Design/Super Loop Design
  - Embedded Operating System Based Design

# SUPER LOOP BASED APPROACH

- This approach is applied for the applications that are not **time critical and the response time is not so important**
- Similar to the conventional procedural programming where the code is executed task by task
- Task listed at the top of the program code is executed first and task below the first task are executed after completing the first task
- True procedural one
- In multiple task based systems, each task executed in serial

□ Firmware execution flow of this will be

1. Configure the common parameter and perform initialization for various hardware components, memory, registers etc.
2. Start the first task and execute it
3. Execute the second task
4. Execute the next task
5. ....
6. ....
7. Execute the last defined task
8. Jump back to the first task and follow the same flow

- From the firmware execution sequence, it is obvious that the **order** in which the task to be executed are fixed and they are **hard coded** in the code itself
- Operations are infinite loop based approach
- In terms of C program code as:

```
Void main()
{ configuration();
initializations();
while(1{
    □ task1();
    □ task2();
    □ ....
    □ taskn();
    □ } }
```

- Almost all task in embedded applications are non-ending and are repeated infinitely throughout the operation
- By analyzing C code we can see that the task 1 to n are performed one after another and when the last task is executed, the firmware execution is again redirected to task 1 and it is repeated forever in the loop.
- This repetition is achieved by using an infinite loop(`while(1)`)
- Therefore Super loop based Approach

- Since the tasks are running inside an infinite loop, the only way to come out of the loop is either
  - Hardware reset or
  - Interrupt assertion
- A **Hardware reset** brings the program execution back to the main loop
- Whereas the **interrupt** suspends the task execution temporarily and performs the corresponding interrupt routine and on completion of the interrupt routine it restarts the task execution from the point where it got interrupted

- Super Loop based design **does not require** an OS, since there is no need for scheduling which task is to be executed and assigning priority to each task.
- In a super Loop based design, **the priorities** are fixed and the order in which the task to be executed are also fixed
- Hence the code for performing these task will be residing in the code memory without an operating system image

- This type of design is deployed in **low-cost embedded products** where the response time is not time critical
- Some embedded products demand this type of approach if some tasks itself are sequential
- For example, reading/writing data to and from a card using a card reader requires a sequence of operations like checking the presence of the card, authenticating the operation, reading/writing etc..
  - It should strictly follows a specified sequence and the combination of these series of tasks constitutes a single task namely read write
  - There is no use in putting the sub tasks into independent task and running them parallel

- Example of “ Super Loop Based Design” is
  - **Electronic video game toy** containing keypad and display unit
  - The program running inside the product must be designed in such a way that it reads the key to detect whether user has given any input and if any key press is detected the graphic display is updated. The keyboard scanning and display updating happens at a reasonable high rate
  - Even if the application misses the key press , it won't create any critical issue
  - Rather it will treated as a bug in the firmware

# Drawback of Super Loop based Design

- Major drawback of this approach is that any failure in any part of a single task will affect the total system
  - If the program hang up at any point while executing a task, it will remain there forever and ultimately the product will stop functioning
  - Some remedial measures are there
    - Use of Hardware and software Watch Dog Timers (WDTs) helps in coming out from the loop when an unexpected failure occurs or when the processor hang up
    - – May cause additional hardware cost and firmware overhead

- Another major drawback is lack of real **timeliness**
  - If the number of tasks to be executed within an application increases, the time at which each task is repeated also increases. This brings the probability of missing out some events
  - For example in a system with keypad, there will be task for monitoring the keypad connected I/O lines and this need not be the task running while you press the keys
  - That is keypressing event may not be in sync with the keypad press monitoring task within the firmware
  - To identify the keypress, you may have to press the key for a sufficiently long time till the keypad status monitoring task is executed internally.
  - Lead to lack of real timeliness

# Embedded Operating System Based Approach

- Contains OS, which can be either a General purpose Operating System (GPOS) or real Time Operating System (RTOS)

# GPOS based design

- GPOS based design is very similar to the conventional PC based Application development where the device contain an operating system and you will be creating and running user applications on top of it
  - Examples of Microsoft Windows XP OS are PDAs, Handheld devices/ Portable Devices and point of Sale terminals
  - Use of GPOS in embedded product merges the demarcation of Embedded systems and General Purpose systems in terms of OS
  - For developing applications on the top of the OS , OS supported APIs are used
  - OS based applications also requires 'Driver Software' for different hardware present on the board to communicate with them

# RTOS BASED DESIGN

- RTOS based design approach is employed in embedded product demanding Real Time Responses
- RTOS respond in a timely and predictable manner to events
- RTOS contain a real time Kernel responsible for performing pre-emptive multi tasking scheduler for scheduling the task, multiple thread etc.
- RTOS allows a flexible scheduling of system resources like the CPU and Memory and offer some way to communicate between tasks
  - Examples of RTOS are
    - Windows CE, pSOS, VxWorks, ThreadX, Micro C/OS II, Embedded Linux, Symbian etc...



# **EMBEDDED FIRMWARE DEVELOPMENT LANGUAGES**

# EMBEDDED FIRMWARE DEVELOPMENT LANGUAGES

- For embedded firmware development you can use either
  - Target processor/controller specific language (Assembly language) or
  - Target processor/ controller independent language (High level languages) or
  - Combination of Assembly and high level language

# Assembly language based development

- Assembly language is human readable notation of machine language whereas machine language is a processor understandable language
- Processor deal only with binaries
- Machine language is a binary representation and it consists of 1s and 0s
- Machine language is made readable by using specific symbols called 'mnemonics'
- Hence machine language can be considered as an interface between processor and programmer
- Assembly language and machine languages are processor dependent and assembly program written for one processor family will not work with others

- **Assembly language programming is the task of writing processor specific machine code in mnemonics form, converting the mnemonics into actual processor instructions (machine language) and associated data using an assembler**

- Assembly language program was the most common type of programming adopted in the beginning of software revolution
- Even in 1990s majority of console video games were written in assembly languages
- Even today almost all low level, system related, programming is carried out using assembly language
- Some OS dependant task requires low level languages
  - In particular assembly language is used in writing low level interaction between the OS and the hardware, for instance in device drivers

- The general format of an assembly language instruction is Opcode followed by the Operand
- Opcode tells what to do and Operand gives the information to do the task
- The operand may be single operand, dual operand or more
  - MOV A, #30
  - Move the decimal value 30 to the accumulator register of 8051
  - Here MOV A is the opcode and 30 is Operand
  - Same instruction in machine language like this  
01110100 00011110
  - Here the first 8 bit represent opcode MOV A and next 8 bit represent the operand 30

- The mnemonic INC A is an example for the instruction holding operand implicitly in the Opcode
- The machine language representation is 00000100
  - This instruction increment the 8051 Accumulator register content by 1
- LJMP *16 bit address* is an example of dual operand instruction
- ~~The machine language for the same is~~
  - The first binary data is the representation of LJMP machine code
  - The first operand that immediately follow the opcode represent the bit 8 to 15 of the 16 bit address to which the jump is requited and the second operand represent the bit 0 to 7 of the address to which the jump targeted

- Assembly language instructions are written in one per line
- A machine code program thus consisting of a sequence of assembly language instructions, where each statement contains a mnemonic

- Each line of assembly language program split into four field as given below

| LABEL | OPCODE | OPERAND | COMMENTS |
|-------|--------|---------|----------|
|-------|--------|---------|----------|

- Label is an optional field. A label is an identifier to remembering where data or code is located

- **LABEL** is commonly used for representing
  - A memory location, address of a program, subroutine, code portion etc...
  - The max length of the label differ between assemblers. Labels are always suffixed by a colon and begin with a valid character. Labels can contain numbers from 0 to 9 and special character \_
  - Labels are used for representing subroutine names and jump locations in Assembly language programming
  - Label is only an optional field

DELAY:           MOV R0, #255                 ;load Register R0 with 255  
                DJNZ R1, DELAY                 ;Decrement R1 and loop  
                ; till R1=0  
                RET                             ;return to calling program

- The assembly program contain a main routine which start at address 0000H and it may or may not contain subroutines.
- In main program subroutine is involked by the assembly instruction  
`LCALL DELAY`
- Executing this instruction transfers the program flow to the memory address referenced by the ‘LABEL’ `DELAY`
- While assembling the code a ‘;’ inform the assembler that the rest of the part coming in a line after the ‘;’ symbol is comments and simply ignore it
- Each assembly instruction should be written in a separate line
- More than one ASM code lines are not allowed in

- In the previous example LABEL DELAY represent the reference to the start of the subroutine

|        |                |                                       |
|--------|----------------|---------------------------------------|
| DELAY: | MOV R0, #255   | ;load Register R0 with 255            |
|        | DJNZ R1, DELAY | ;Decrement R1 and loop<br>; till R1=0 |
|        | RET            | ;return to calling program            |

- We can directly replace the LABEL by putting desired address first and then writing assembly code for the routine

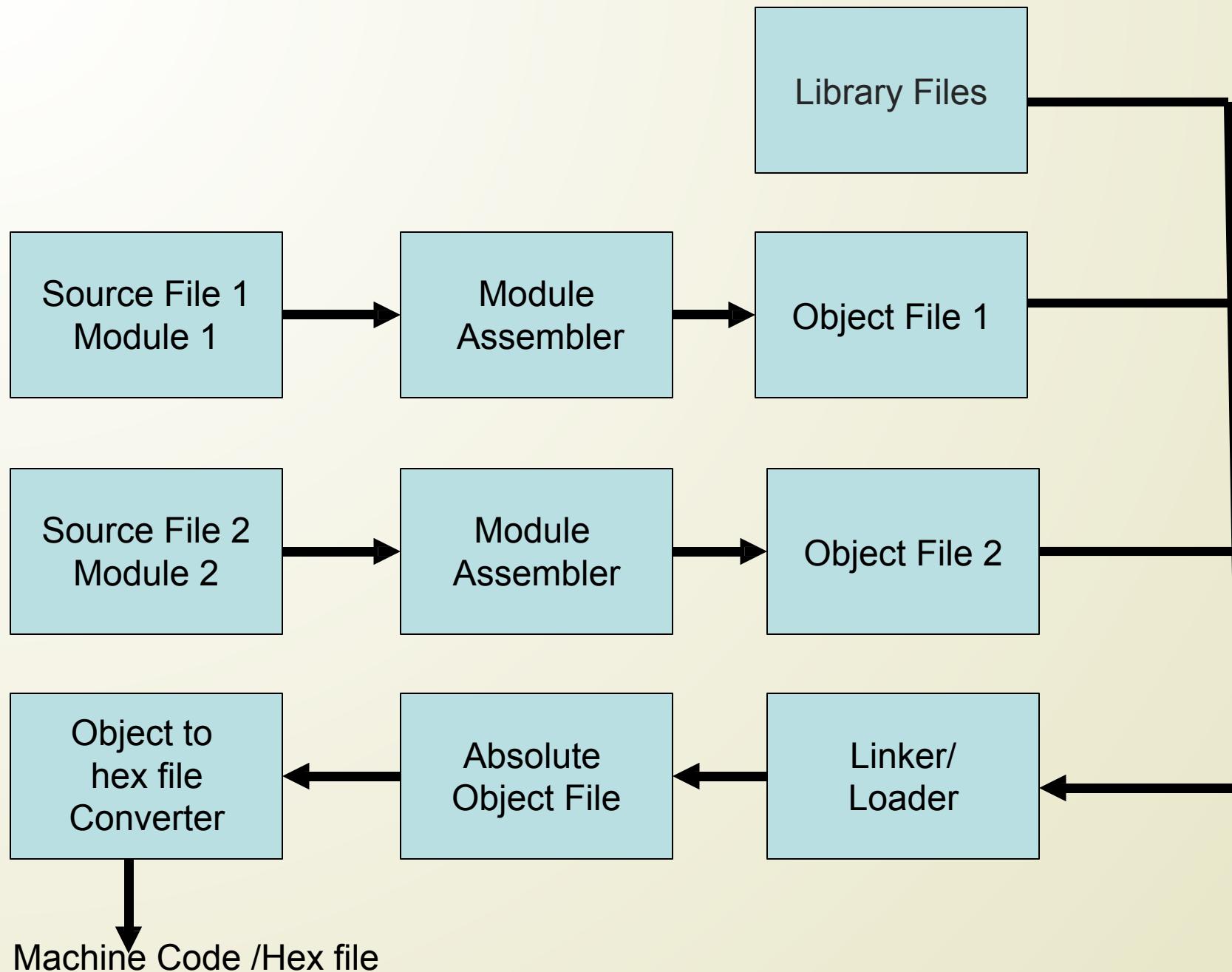
|           |                |                                       |
|-----------|----------------|---------------------------------------|
| ORG 0100H |                |                                       |
|           | MOV R0, #255   | ;load Register R0 with 255            |
|           | DJNZ R1, DELAY | ;Decrement R1 and loop<br>; till R1=0 |
|           | RET            | ;return to calling program            |

- ORG 0100H is not an assembly language instruction; it is an assembler directive instruction. It tells the assembler that the instruction from here onwards should be placed at location starting from 0100H
- Assembler directive instructions are known as ‘pseudo ops’
- They are used for
  - Determining the start address of the program (eg. ORG 0100H)
  - Determining the entry address of the program (eg. ORG 0100H)
  - Reserving the memory for data variables, arrays and structures (eg. Var EQU 70H)
  - Initializing variable values (e.g. val DATA 12H)
- EQU directive is used for allocating memory to a variable and DATA directive is used for initializing a variable with data
- No machine codes are generated for the ‘Pseudo-ops’

- Assembly language program written in assembly code is saved as .asm file or an .src file
- Any text editor can be used for writing assembly instructions
- Similar to other high level programming, you can have multiple source files called modules in assembly language programming.
- Each module is represented by .asm or .src file
- This approach is known as modular programming
- Modular program is employed when program is too complex or too big.
- In modular programming the entire code is divided into sub modules and each module is made reusable
- Modular programs are usually easy to code, debug and alter

# SOURCE FILE TO OBJECT FILE TRANSLATION

- Translation of assembly code to machine code is performed by assembler.
- The assemblers for different target machines are different and it is common that assemblers from multiple vendors are available in the market for the same target machines.
- Some assemblers are supplied by single vendor only (proprietary).
- Some assemblers are freely available.
- Some are commercial and requires license from vendors
  - A51 Macro Assembler from Keil software is a popular assembler for 8051 family micro controller.



- Each source module is written in assembly and is stored in .src or .asm file
- Each file can be assembled separately to examine the syntax errors and incorrect assembly instructions
- On assembling of each .src/.asm file a corresponding object file is created with extension .obj
- The object file does not contain the absolute address of where the generated code need to be placed on the program memory and hence it is called relocatable segment
- It can be placed at any code memory location and it is responsibility of the linker/loader to assign absolute address for this module
- Absolute address allocation is done at absolute object file creation stage
- Each module can share variables and subroutine among them
- Exporting a variable from a module is done by declaring that variable as PUBLIC in source module

- Importing a variable or a function from a module is done by declaring that variable or function as EXTRN in the module where it is going to be accessed
- PUBLIC keyword inform the assembler that the variable / function need to be exported
- EXTRN inform that the variable/function need to be imported from some other modules
- While assembling a module , on seeing variable /function with keyword EXTRN , assembler understand that these variables or function come from an external module and it proceeds assembling the entire module without throwing any errors, though the assembler cannot find the definition of variables and implementation of that function

# Extern Keyword

- Declares a global reference defined in another file to be visible by current file
  - Can be bss or data
  - Initial definition must be a global variable

```
extern int VARA;  
extern char VARB;  
extern int VARC;
```

File1.c

```
int VARA = 1;  
char VARB;  
int VARC = 0;
```

File2.c



- Corresponding to a variable /function declared as PUBLIC in a module, there can be one or modules using these variables/function using EXTRN keyword
- For all those modules using variables or function with EXTRN keyword, there should be one and only one module which export those variables/functions PUBLIC keyword
- If more than one module in a project tries to export variables or functions with the same name using PUBLIC keyword, it will generate linker errors

- If a variable or function declared as EXTRN in one or two modules, there should be one module defining these variables or function and exporting them using PUBLIC keyword
- If no module in a project export the variable or functions which are declared as EXTRN in other modules it will generate linker warnings or error depending on the error level/warning level setting of the linker

# Library file creation and usage

- Libraries are specially formatted, ordered program collection of object modules that may be used by the linker at a later time
- When a linker process a library, only those object modules in the library that are necessary to create the program are used
- Library files are generated with the extension ‘.lib’
- Library file is some kind of source code hiding technique
- If you don’t want to reveal the source code behind the various functions you have written in your program and at the same time you want them to be distributed to application developers for making use of them in their applications, you can supply them as library files and give them the details of the public functions available

- For using a library file in a project, add library to the project
- If you are using a commercial version of assembler suit for your development, the vendor of utility may provide you pre written library files for performing multiplication, floating point arithmetic, etc. as an add-on utility
- Example LIB51 from keil software

# Linker and Locator

- Linker and locator is another software utility responsible for “linking the various object modules in a multi module project and assigning absolute address to each module”
- Linker generate an absolute object module by extracting the object module from the library, if any and those obj files created by the assembler, which is generated by assembling the individual modules of a project
- It is the responsibility of the linker to link any external dependent variables or functions declared on various modules and resolve the external dependencies among the modules
- An absolute object file or modules does not contain any re-locatable code or data
- All code and data reside at fixed memory locations
- The absolute object file is used for creating hex files for dumping into the code memory of the processor/controller
- Example ‘BL51’ from keil software

# Object to Hex File Converter

- This is the final stage in the conversion of Assembly language to machine understandable language
- Hex file is the representation of the machine code and the hex file is dumped into the code memory of the processor
- Hex file representation varies depending on the target processor make
- For intel processor the target hex file format will be ‘Intel HEX’ and for Motorola, hex file should be in ‘Motorola HEX’ format
- HEX files are ASCII files that contain a hexadecimal representation of target application
- Hex file is created from the final ‘Absolute Object File’ using the Object to Hex file Converter utility
- Example ‘OH51’ from keil software

# Advantage of Assembly Language Based Development

- Assembly language based development is the most common technique adopted from the beginning of the embedded technology development
- Thorough understanding of the processor architecture , memory organization , register set and mnemonics is very essential for Assembly Language based Development

- Efficient Code Memory and data Memory Usage (Memory Optimization)
  - Since the developer is well versed with the target processor architecture and memory organization, optimized code can be written for performing operations
  - This lead to the less utilization of code memory and efficient utilization of data memory
  - Memory is the primary concern in any embedded product

- High Performance
  - Optimized code not only improve the code memory usage but also improve the total system performance
  - Though effective assembly coding optimum performance can be achieved for target applications

- Low level Hardware access
  - Most of the code for low level programming like accessing external device specific registers from the operating system kernel, device drivers and low level interrupt routine etc. are making use of direct assembly coding since low level device specific operation support is not commonly avail with most of the high level language compilers

- Code Reverse Engineering
  - Reverse Engineering is the process of understanding the technology behind a product by extracting the information from the finished product
  - Reverse engineering is performed by ‘hawkers’ to reveal the technology behind the proprietary product
  - Though most of the products employ code memory protection, if it may be possible to break the memory protection and read the code memory, it can easily be converted into assembly code using dis-assembler programs for the target machine

# DRAWBACKS OF ASSEMBLY LANGUAGE BASED DEVELOPMENT

- **High Development time**
  - Assembly language programs are much harder to program than high level languages
  - Developer must have thorough knowledge of architecture, memory organization and register details of target processor in use
  - Learning the inner details of the processor and its assembly instructions are high time consuming and it create delay impact in product development
  - **Solution**
    - Use a readily available developer who is well versed in target processor architecture assembly instructions
    - Also more lines of assembly code are required for performing an action which can be done with a single instruction in a <sup>A</sup>h<sub>rs</sub><sup>h</sup>, <sup>a</sup>g<sub>JK</sub>, <sup>A</sup>e<sup>P</sup>, <sup>C</sup>l<sup>S</sup><sub>E</sub><sup>D</sup>, <sup>V</sup>l<sup>J</sup>, <sup>g</sup>u<sup>CE</sup><sub>T</sub> like C

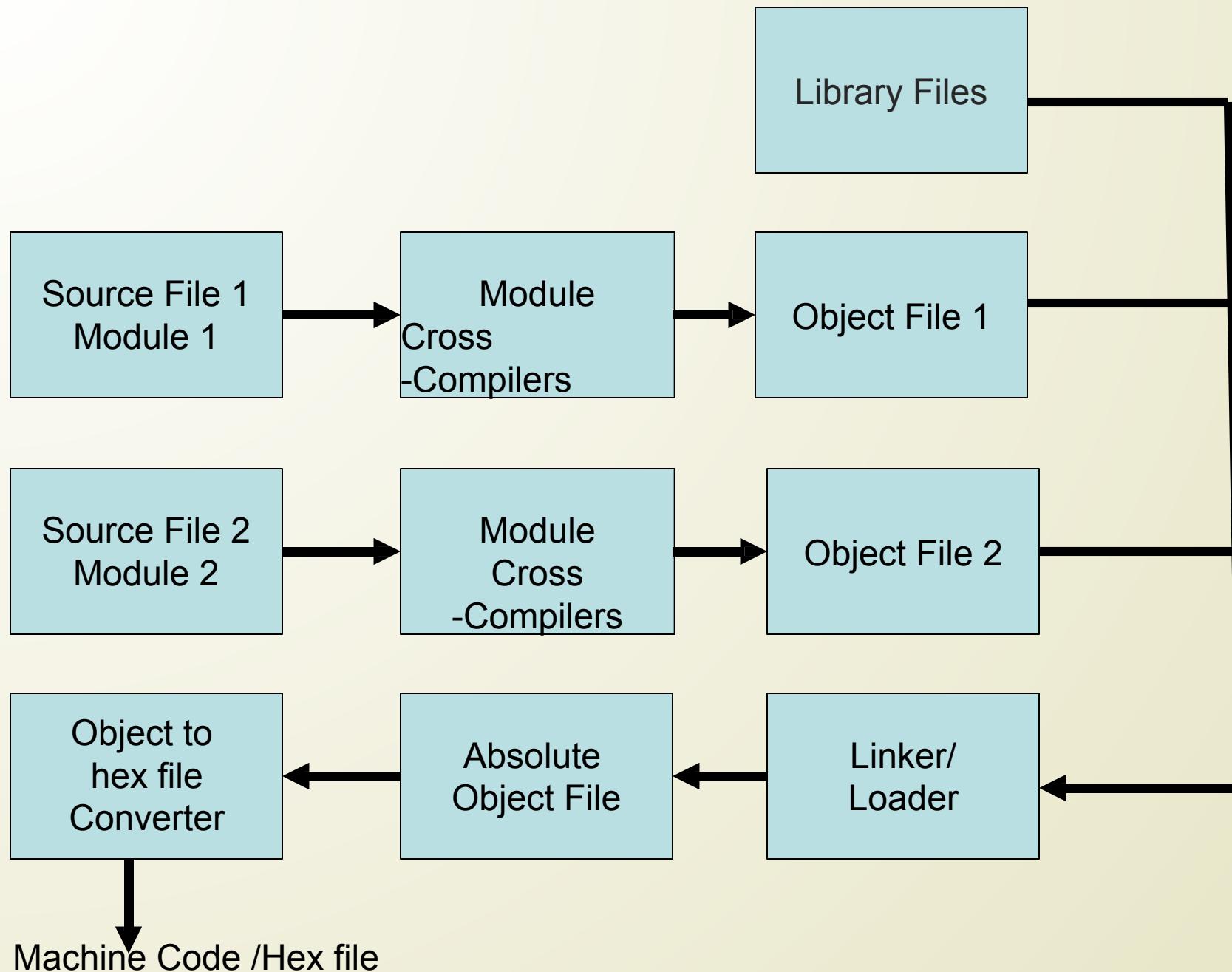
- **Developer Dependency**

- There is no common rule for developing assembly language based applications whereas all high level language instruct certain set of rules for application development
- In Assembly language programming, the developers will have the freedom to choose the different memory locations and registers
- Also programming approach varies from developers to developers depending on their taste
- For example moving a data from a memory location to accumulator can be achieved through different approaches
- If the approach is done by a developer is not documented properly at the development stage, it may not be able to recollect at later stage or when a new developer is instructed to analyze the code , he may not be able to understand what is done and why it is done
- Hence upgrading/modifying on later stage is more difficult
- Solution
  - Well Documentation

- Non- Portable
  - Target applications written in assembly instructions are valid only for that particular family of processors
    - Example—Application written for Intel X86 family of processors
  - Cannot be reused for another target processors
  - If the target processor changes, a complete rewriting of the application using assembly instructions for the new target processor is required

# High Level Language Based Development

- Any High level language with a supported cross compilers for the target processor can be used for embedded firmware development
  - Cross Compilers are used for converting the application development in high level language into target processor specific assembly code
- Most commonly used language is C
  - C is well defined easy to use high level language with extensive cross platform development tool support



- The program written in any of the high level language is saved with the corresponding language extension
- Any text editor provided by IDE tool supporting the high level language in use can be used for writing the program
- Most of the high level language support modular programming approach and hence you can have multiple source files called modules written in corresponding high level language
- The source file corresponding to each module is represented by a file with corresponding language extension
- Translation of high level source code to executable object code is done by a cross compiler
- The cross compiler for different high level language for same target processor are different
- Without cross-compiler support a high level language cannot be used for embedded firmware development
  - Example C51 Compiler fro Keil

# Advantages of High Level Language based Development

- Reduced Development Time
  - Developers requires less or little knowledge on the internal hardware details and architecture of the target processor
  - Syntax of high level language and bare minimal knowledge of memory organization and register details of target processor are the only pre- requisites for high level language based firmware development
  - With High level language, each task can be accomplished by lesser number of lines of code compared to the target processor specific assembly language based development

- Developer Independence
  - The syntax used by most of the high level languages are universal and a program written in high level language can be easily understood by a second person knowing the syntax of the language
  - High level language based firmware development makes the firmware , developer independent
  - High level language always instruct certain set of rules for writing code and commenting the piece of code

- Portability
  - Target applications written in high level languages are converted to target processor understandable format by a cross compiler
  - An application written in high level language for a particular target processor can be easily converted to another target processor with little effort by simply recompiling the code modification followed by the recompiling the application for the required processor
  - This makes the high level language applications are highly portable

## Limitations of High level language based development

- Some cross compilers available for the high level languages may not be so efficient in generating optimized target processor specific instructions
- Target images created by such compilers may be messy and not optimized in terms of performance as well as code size

# Mixing Assembly and High level Language

- High level language and assembly languages are usually mixed in three ways
  - Mixing assembly language with high level language
  - Mixing high level language with Assembly
  - In line assembly programming

# Mixing Assembly Language with High level

## Language (Assembly Language with ‘C’)

- Assembly routines are mixed with C in situations where
  - entire program is written in C and the *cross compiler in use do not have built in support for implementing certain features like Interrupt Service Routine or*
  - if the programmer want to take the advantage of speed and optimized code offered by machine code generated by hand written assembly rather than cross compiler generated machine code
- When accessing certain low level hardware, the timing specification may be very critical and a cross compiler generated binary may not be able to offer the required time specifications accurately
- Writing the hardware access routine in processor specific assembly language and invoking it from C is the most advised method

- Mixing C and Assembly is little complicated in the sense-
  - the programmer must be aware of how parameters are passed from the C routine to Assembly and
  - values are returned from assembly routine to C and
  - how the assembly routine is invoked from the C code
- These are cross compiler dependent
- There is no universal rule for it
- You must get the information from the documentation of cross compiler you are using
- Different cross compilers implement these features in different ways depending upon the general purpose registers and the memory supported by the target processor

- Example
  1. Write a simple function in C that passes parameters and return values the way you want your assembly routine to
  2. Use the SRC directive (#pragma SRC) so that C compiler generate an SRC file instead of .OBJ file
  3. Compile the C code. Since the SRC directive is specified the .SRC file is generated. The .SRC file contain the assembly code generated for the C code you wrote
  4. Rename .SRC to .A51 file
  5. Edit .A51 file and insert the assembly code you want to execute in the body of the assembly function shell included in the .A51 file

```
#pragma src
Unsigned char my_assembly_func(unsigned int
    argument)
{
    Return (argument+1);
}
```

**This C function on cross compilation generate the  
following assembly SRC file**

```
NAME TESTCODE
?PR?_my_assembly_func?TESTCODE segment code
    PUBLIC _my_assembly_func
;#pragma SRC
;unsigned char my_assembly_func(
    RSEG ?PR?_my_assembly_func?TESTCODE
    USING 0
_my_assembly_func:
;--variable 'argument?040' assigned to Register 'R6/R7'----
; SOURCE LINE #2
; unsigned int argument)
;{
;SOURCE LINE #4
;return (argument+1);
;SOURCE LINE #5
        MOV A,R7
        INC A
        MOV R7,A
;}
;source line #6
;?C0001:
        RET
;END OF _my_assembly_func
    END
```

- The special compiler directive SRC generates the Assembly code corresponding to the ‘C’ function and each line of the source code is converted to the corresponding Assembly instruction.
- You can easily identify the Assembly code generated for each line of the source code since it is implicitly mentioned in the generated .SRC file
- By inspecting this code segments you can find out which register are used for holding the variables of the ‘C’ function and you can modify the source code by adding the assembly routine you want

# Mixing High level Language with Assembly

## Language ('C' with Assembly Language)

- Mixing the code written in high level language like C and assembly language is useful in the following scenarios;
  - The source code is already available in Assembly language and a routine written in a high level language like C need to be included to the existing code
  - The entire code is planned in Assembly code for various reasons like optimized code, optimal performance, efficient code memory utilization and proven expertise in handling the Assembly, etc. But some portions of the code may be very difficult and tedious to code in Assembly
  - To include built in library functions written in C language provided by the cross compiler

- Most often the functions written in C use parameter passing to the function and returns values to the calling function
- By mixing C with Assembly
  - How parameters are passed to the function
  - How values are returned from the function
  - How the function is invoked from the assembly language environment
- Parameters are passed to the function and values are returned from the function using CPU registers, stack memory and fixed memory
- Its implementation is cross compiler dependant and it varies across cross compilers
- Example Keil C51 cross compiler

- C51 allows passing of maximum of three arguments through general purpose registers R2 to R7
- If three arguments are char variables, they are passed to the functions using registers R7,R6, and R5
- If the parameters are int variables they are passing using register pairs(R7,R6),(R5,R4) and (R4,R3)
- If the number of arguments are greater than three, the first three arguments are passed through registers and the rest is passed through fixed memory locations
- Return values are usually passed through fixed memory locations
- R7 is used for returning char value and register pair (R5,R6)

- The C subroutine can be invoked from the assembly program using the subroutine call Assembly instruction

LCALL \_Cfunction

- Where Cfunction is a function written in C
- The prefix inform the cross compiler that the parameters to the function are passed through registers
- If the function is invoked without the \_prefix, it is understood that the parameters are passed through fixed memory locations

# INLINE ASSEMBLY

- This is another technique for inserting target processor/controller specific assembly instructions at any location of source code written in high level language C
- This avoid the delay in calling an assembly routine from a C code
- Special keywords are used to indicate that the start and end of the Assembly instructions
- The keywords are cross compiler specific
- C51 uses #pragma asm and #pragma endasm to indicate a block of code written in assembly



# Integration and Testing of Embedded firmware & Hardware

Module 4

# Topics to Cover

- Integration of Hardware and Firmware
- Embedded System Development Environment-IDEs
- Cross Compilers
- Disassemblers
- Decompilers
- Simulators, Emulators and Debuggers.

# Introduction

- Integration and testing of embedded hardware and software is the immediate step following the embedded hardware and firmware development.
- The final embedded hardware constitutes of a PCB with all necessary components affixed to it as per the original schematic diagram.
- Embedded firmware represents the control algorithm and configuration data necessary to implement the product requirement on the product.
- Embedded firmware will be in processor understandable machine code form.
- Both embedded hardware and firmware should be independently tested to ensure the proper functioning.

# Verifying hardware sections

- Functioning of individual hardware sections can be verified by writing small utilities which checks the operations of the specified part.
- The functionality of the firmware part can be checked by simulator environment provided by the embedded firmware development tools IDE.
- By simulating the firmware, the register details, memory contents and status of flags can be easily monitored and gives a idea about what happens inside the processor.

# Integration of Hardware & Firmware

- Deals with embedding of firmware into the target hardware board
- It's the process of *embedding intelligence* to the product
- If the processor's internal memory has enough space to accommodate the firmware, it will be loaded there, else additional EPROM/Flash chips are used for saving the firmware.

# Integration of Hardware & Firmware

- There are different techniques for embedding firmware to hardware
- Out-of-Circuit Programming
- In System Programming(ISP)
- In Application Programming
- Use of Factory programmed chip

# Out-of-Circuit Programming

- It's performed outside the target board

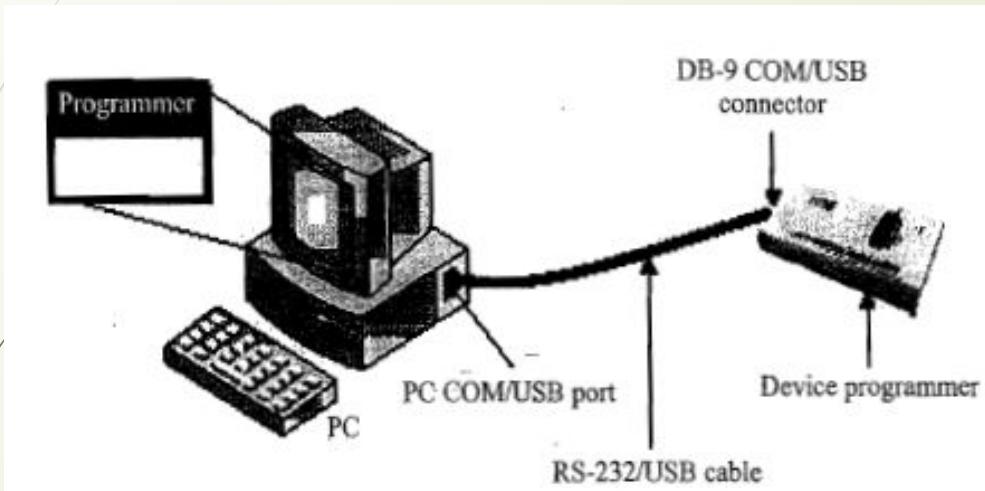


Labtool-48UXP

# Out-of-Circuit Programming

- The processor or memory chip into which the firmware needs to be embedded is taken out of the target board and its programmed with the programming device
- The programming device is a dedicated unit which contains the necessary hardware circuits generate the programming signals.
- Most of this devices are capable to support different family of devices.
- The programmer contains ZIF socket locking pin to hold the device to be programmed.
- The programming device is under the control of a utility program running in the PC.

# Out-of-Circuit Programming



- Interfacing of device programmer with pc

# Out-of-Circuit Programming

- The sequence of operations for embedded firmware with a programmer is
  1. Connect the programming device to specified port(USB/COM/Parallel port)
  2. Power up the device
  3. Execute the programming utility on the pc and ensure proper connectivity between pc and programmer
  4. Unlock the ZIF socket by turning the lock pin
  5. Insert the device to be programmed into the open socket
  6. Lock ZIP socket

# Out-of-Circuit Programming

7. Select the device name from the list of supported devices
8. Load the hex file which is to be embedded into the device
9. Program the device by program option of utility program
10. Wait till the completion of programming operation
11. Ensure that programming is successful by checking the status LED on the programmer
12. Unlock the ZIF socket & take device out of programmer

- After doing these operations the firmware is successfully embedded into the device.
- Insert the device into the board, power up the board, power up the board and test it for required functionalities.
- The ZIF socket supports only Dual Inline Package(DIP)
- If security is required, enable the memory protection on the utility before programming the device
- Only EEPROM/FLASH memory are erasable

# Drawback of Out-of-Circuit Programming

- High development time. Whenever changes occurs in the firmware chip should be taken out from the device and programmed. This may cause chip damage due to frequent insertion and removal. A socket can be used at the board side to hold the chip till modification is over.
- Programmer allows to program only one chip at a time. Hence not suitable for batch production. Gang programmer resolves this problem.
- Very difficult to update firmware after the product is released

# In System Programming

- Firmware is embedded into the target device without removing it from the target board
- The target device must have ISP support.
- The required facilities are board, PC, ISP utility and ISP cable.
- Normally serial interface communication and protocols preferred. The protocols used for ISP are Joint Test Action Group (JTAG) or Serial Peripheral Interface (SPI) or any other proprietary protocol.
- In order to perform ISP operations, the target device must be in ISP mode.
- This mode allows the device to communicate with an external host through serial interface. The device receives commands and data from the host, erases and reprograms the code memory according to the received command.
- Once the ISP operations are completed, the device is reconfigured.

# In System Programming with SPI protocol

- Devices with SPI in System Programming support contains a built in SPI interface and the on chip EEPROM or FLASH memory is programmed through this interface.
- The primary I/O lines involved in SPI are
  - MOSI-Master Out Slave In-program data is sent to the MOSI pin of the target device.
  - MISO-Master In Slave Out - The device acknowledgement is originated from the MISO pin of the device.
  - SCK-System Clock – SCK pin acts for the clock for data transfer.
  - RST-Reset of Target Device -
  - GND-Ground of Target Device
- PC acts as master and target device acts like slave in ISP
- A utility program can be developed on the PC to generate these signal lines and these lines are connected to the parallel port of the PC.
- The pins of the parallel port to which the ISP pins of the device need to be connected depends on the program else we can fix the lines and then use it all.

# Power up sequence of ISP for Atmel's AT89S series microcontroller

1. Apply supply voltage b/w VCC and GND pin of target chip
2. Set RST pin to HIGH
3. If crystal is not connected across pins XTAL1 and XTAL2,apply 3 to 24 Mhz clock to XLAL1 and wait for 10ms
4. Enable serial programming by sending the programming enable serial instruction to pin MOSI/P1.5. the frequency of the shift supplied at pin SCK/P1.7 need to be less than the CPU clock at XTAL1 divided by 40.

# In System Programming

5. The code or data array is programmed one byte at a time by supplying the address and data together with the appropriate Write instruction. The selected memory location is first erased before the new data is written
6. Any memory location is verified by using read instruction, which returns the content at the selected address at serial output MISO/P1.6
7. After successfully programming the device, set RST to low or turn off the chip power supply and turn it to ON to commence the normal operation

# In Application Programming

- It's a technique used by firmware running on the target device for modifying a selected portion of the code memory
- It modifies the program code memory under the control of embedded application including updating calibration data, look up tables etc. which are stored in embedded applications.
- The Boot ROM resident API instruction which performs various functions such as programming, erasing, reading the flash memory during the ISP mode is made available to the end user written firmware for IAP.
- Thus it is possible for an end user application to perform operation on the flash memory.
- A common entry point to these API routines is provided for interfacing them to the end users applications.
- Functions are performed by setting up specific registers as required by specific operations and performing a call to the common entry point.

- The Boot ROM is shadowed with the user code memory and its address range.
- The shadowing is controlled by a status bit.
- When this bit is set, access to the internal code memory in this address range is from the Boot ROM.
- When this bit is not set access will be from users code memory.

# Use of factory Programmed Chip

- Here embed the firmware into the target processor/controller memory at the time of chip fabrication itself. Such chips are called factory programmed chip.
- Once the firmware design is over and the firmware achieved operational stability, the firmware files can be sent to the chip fabricator to embed it into the code memory
- It reduces the product development time.
- They are bit expensive
- It is not recommended as the firmware goes frequent modifications.

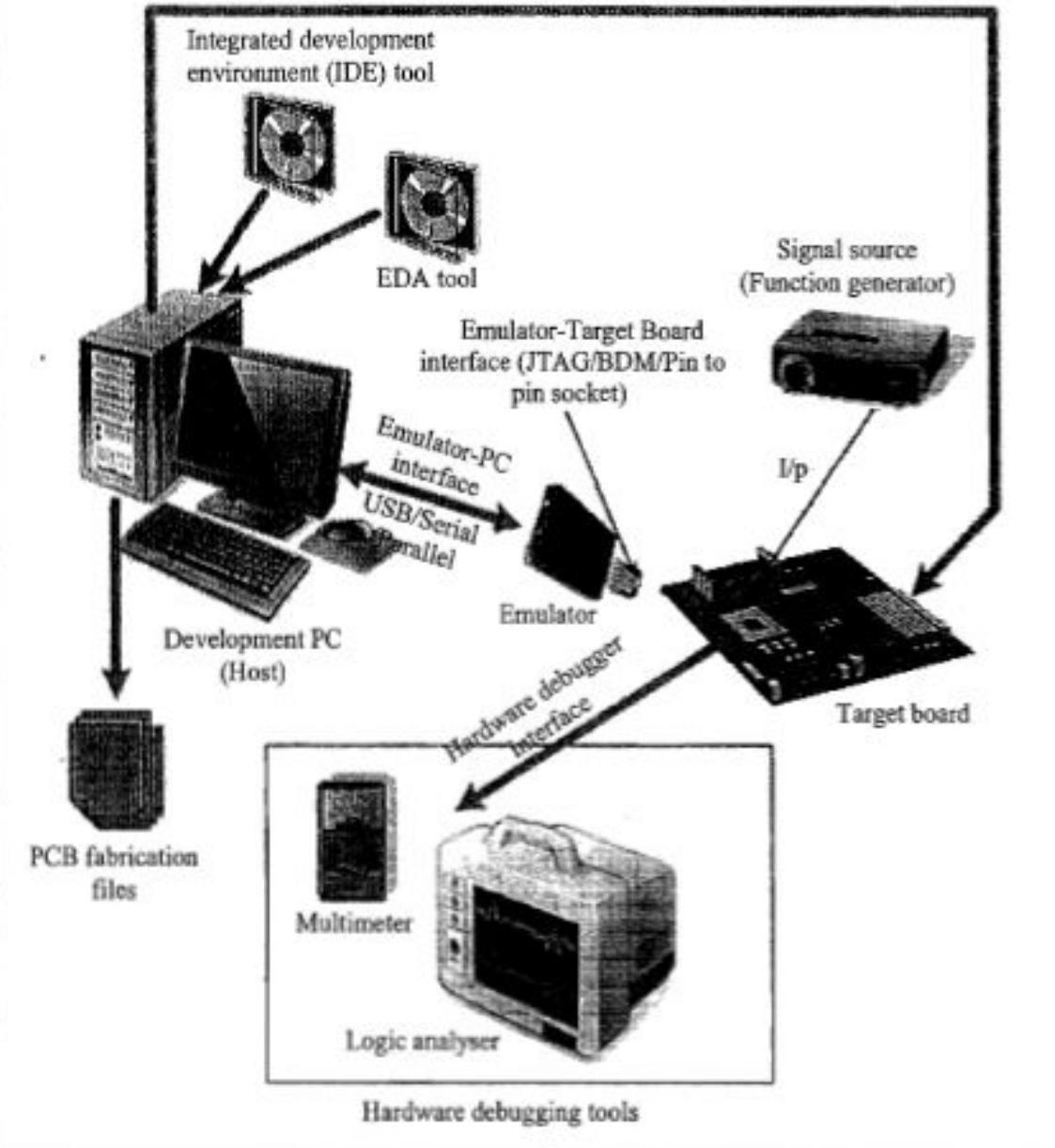
# Firmware Loading for OS based devices

- It's programmed using ISP technique
- OS based system contain a special piece of code called *boot loader* program which takes control of the OS and application firmware embedding and copying of the OS images to the RAM of the system for execution.
- Bootloader comes as preloaded or it can be loaded in to the memory using various interface support like JTAG.
- Boot loader contains necessary driver initialization implementation for initializing the support interface like UART,TCP/IP etc
- E.g. Load from FLASH ROM, Load from network ,Load through UART etc
- In case of network based loading, the bootloader broadcast the target's presence over the network and the host machine on which the OS image resides can identify the target device capturing the message. Once the communication link is established between the target and host, OS can be directly downloaded into the flash memory of the target device.



# EMBEDDED SYSTEM DEVELOPMENT ENVIRONMENT

In system programming (ISP) interface (Serial/USB/Parallel/TCP-IP)



# Components of Embedded development environment

- Host Computer
  - Acts as the heart of development environment.
- IDE Tools
  - Tools for firmware design and development
- Electronic Design Automation Tools
  - Embedded Hardware Design
    - IDE & EDA are selected based on the target hardware.
    - They are supplied as installable files.
- Emulator hardware
  - Debugging target board

# Components of Embedded development environment

- Signal Sources (function generator)
  - Simulates inputs to target board
- Target Hardware Debugging tools
  - CRO(Cathode Ray Oscilloscope), Multimeter ,Logic Analyser
  - For debugging hardware
- Target Hard ware

# IDE

- In E.S, IDE stands for an integrated environment for developing and debugging the target processor specific embedded firmware
- An IDE is also known as integrated design environment or integrated debugging environment.
- IDE is a software package which bundles a “Text Editor”, “Cross-compiler”, “Linker” and a “Debugger”
- IDE is a software application that provides facilities to computer programmers for software development. IDEs can either command line based or GUI based
- GUI based IDEs are called visual IDE. Eg: visual C++, NetBEans, Eclipse.

# Examples.

- The IDEs for embedded system development is supplied either by
  - The target processor/controller manufacturer
    - Eg: MPLAB supplied by micro chip for PIC family of microcontrollers.
  - A Third party vendors
    - Eg. Keil mVision5 from ARMKeil for 8051/ARM microcontrollers
  - Open source
    - CodeWarrior for ARM family

# IDE Components

- 1.Text Editor or Source code editor
  - 2.A compiler and an interpreter
  - 3.Build automation tools
  - 4.Debugger
  - 5.Simulators
  - 6.Emulators and logic analyzer
- E.g. Turbo C/C++,Microsoft visual c++ etc

# Cross Compilation

- Cross compilation is the process of converting a source code written in high level language to a target processor/controller understandable machine code
- The conversion of the code is done by software running on a processor/controller which is different from the target processor.
- The software performing this operation is referred as the Cross-compiler
- In other words cross-compilation the process of cross platform software/firmware development.
- A cross complier is a compiler that runs on one type of processor architecture but produces object code for a different type of processor architecture.

# Cross Compiler-Advantages

- By using cross compliers we can not only develop complex E.S , but reliability can be improved and maintenance is easy.
- Knowledge of the processor instruction set is not required.
- Register allocation and addressing mode details are managed by the compiler.
- The ability to combine variable selection with specific operations improves program readability.

# Cross Compiler-Advantages cont

- Keywords and operational functions that more nearly resemble the human thought process can be changed.
- Program development and debugging time will be dramatically reduced when compared to assembly language programming
- The library files that are supplied provide many standard routines that may be incorporated into our application.
- Existing routine can be reused in new programs by utilizing the modular programming techniques available with C.
- The C language is very portable and very popular

# Types of Files Generated on Cross compilation

- 1.List File
- 2.Pre-processor Output file
- 3.Hex File (.hex)
- 4.Map File (File extension linker dependent)
- 5.Object File (.obj)

# 1.List Files(.lst files)

- Generated at the time of cross compilation
- Contain information about cross compilation process like
  - Cross compiler details
  - Formatted source text
  - Assembly code generated from the source file
  - Symbol table
- Errors and warning detected by the cross compiler system
- The type of information contained in the list file is cross compiler specific
- List file is very useful for application debugging in case of any cross compiler issues.

# Sections of List file

## □ PAGE HEADER

- A header on each page of the listing file which indicates the compiler version number, source file name, date, time and page number.
- Eg: C51 COMPILER V9.53.0.0 SAMPLE 10/16/2014 15:47:10  
PAGE 1

## □ Command Line

- Represents the entire command line that was used for invoking the compiler.

## □ Source code

- Outputs the line number along with the source code on that line.
- Special cross compiler directives can be used to include or exclude the conditional codes.
- It also contains comments and include files.
- Special cross compiler commands can also be included to add all the lines of include file`

- Assembly Listing
  - Contains assembly code generated
- Symbol Listing
  - Contains symbolic information about the various symbols present in the cross compiled source file.
  - It contains symbol name, symbol classification, memory space, data type, offset and size in bytes.
  - Symbol listing in the list file presentation can turn on or off by cross compiler.
- Module Information
  - It provides the size of initialized and un initialized memory areas defined by the source file.
- Warning and errors
  - Records the errors encountered or any statement that may create issues in application during cross compilations.

## 2. Preprocessor output file

- Generated during cross compilation
- Contain preprocessor output for the preprocessor instructions used in the source file
- This file is used for verifying the operation of macros and conditional preprocessor directives
- Is a valid C file
- File extension is cross compiler dependant

### 3. Object files(.obj files)

- It is the lowest level file format for any platform.
- Cross compiling each source module converts the various Embedded instructions and other directives present in the module to an object(.OBJ) file.
- The object file is specially formatted file with data records for symbolic information, object code, debugging information etc.

# Object files cont..

- OMF1 & OMF2 are the 2 object files supported by C51 Cross compiler
- List of details included in object file are
  1. Reserved memory for global variables
  2. Public symbol(variable or function )names
  3. External symbol(variable or function )references
  4. Library files with which to link
  5. Debugging information to help synchronize source lines with object code
- The object code present in the object file is not absolute
- It is the responsibility of the linker or loader to assign absolute memory locations to the object code.

# 4.Map Files

- Object file created contains re-locatable codes where their location in memory is not fixed
- It is the responsibility of linker to link these object modules
- The locator is responsible for locating the absolute address to each module in the code memory
- Map files are generated by the linker and loader.
- It is also called linker list files.
- These files are used to keep the information of linking and locating process.
- Map files use extensions .H.,.HH.,.HM depends on linker or loader

# Sections of map files

- Page Header
  - Indicates linker version number, name, date, time and page number
- Command line
  - Represent the command line that used to invoke the linker
- CPU details
  - Contains the details about target CPU and memory model.
- Input modules
  - This section includes the names of all object modules, library files and modules that are in linking process.
- Memory map
  - Lists the starting address, length, relocation types and name of each segment in the program

- Symbol table
  - It contains the value, type and name for all symbols from the different input modules
- Inter module cross reference
  - It includes the session name, memory type and the name of the module in which it is defined and all modules in which it is accessed.
- Program size
  - Contains the size of various memory areas as well as constant and code space for the entire application.
- Warning and errors
  - Contains errors and warnings generated while linking the program.

## 5. Hex file

- It is binary executable file created from the source code.
- The absolute object file created by the linker or loader is converted into processor understandable binary code.
- The utility used to convert an object file to hex file is known as Object to Hex file converter.
- Hex file embed machine code in a particular format.
- Format of Hex files varies across the family of processor.
- Eg: Intel Hex File, Motorola Hex File

- The lines in intel Hex files are corresponding to a Hex Record
- Each record is made up of hexadecimal numbers that represent machine language code or constant data
- Individual records are terminated with a carriage return and a line feed
- It is used for transferring a program and data to a ROM or EEPROM which is used as code memory storage.

# Intel Hex Files

- Intel HEX file is composed of a number of Hex records
- Each record is made up of five fields
- Eg: :llaaaattdd...cc
- Each group of letters corresponds to a different field.
- Each letter represents a single hexadecimal digits
- Each field is composed of two hexadecimal digits.

| Field | Description  |
|-------|--|
| :     | Start of every Intel Hex record  |
| ll:   | Record length field.<br>Indicates the number of data bytes in the record.  |
| aaaa: | Represents the starting address of subsequent data in the record.  |
| tt:   | Indicates HEX record type<br>00: Data Record<br>01:End of File Record<br>02: 8086 segment address record<br>04: extended linear address record |
| dd:   | Data field that represents one byte of data. A record  |

# Motorola Hex File format

- Represented as ASCII text file.
- It represents a HEX Record.
- It is made up of hexadecimal numbers
- The format of record is

|     |    |        |                  |           |          |
|-----|----|--------|------------------|-----------|----------|
| SOR | RT | LENGTH | START<br>ADDRESS | DATA/CODE | CHECKSUM |
|-----|----|--------|------------------|-----------|----------|

| FILED                 | DESCRIPTION   |
|-----------------------|---|
| SOR                   | Stands for Start of Record. The ASCII character S is used as the Start of Record. Every record begin with character 'S'   |
| RT                    | Stands for Record Type. T represents the type of record. There are 4 different types of record.<br>0: Header. Indicates the beginning of HEX file.<br>1: Data record with 16bit address<br>2: Data record with 24bit address<br>9:End of File Record. |
| Length(lI):           | Count of character pair in the record, excluding the type and record length. Each 'l' represent a number between 0 to 9 and A to F  |
| Start address(aaaa ): | Representing starting address for the subsequent data   |
| Code/data (dd):       | Data field that represents one byte of data   |
| Checksum              | Checksum of record.   |

# Disassembler/Decompiler

- Both are reverse engineering tools
- Reverse engineering is a technology used to reveal the technology behind the working of a product
- Used to find out the secret behind popular proprietary product
- Helps the reverse engineering process by translating embedded firmware to assembly /high level instruction
- Powerful tools are available for analyzing the presence of malicious contents
- The exact code is not available. But some what similar code is obtained.

## DISASSEMBLER

- Utility program that convert machine code into assembly code
- The process of converting machine code into assembly code is called disassembling.
- It is complementary to assembling or cross assembling

## DECOMPILER

- Is a utility program that convert machine language instruction to high level language instruction
- For different processors different decompilers are available.
- Performs reverse operation of compiler or cross compiler

# Simulators, Emulators and debugging

## SIMULATORS

- Simulator is a software tool for simulating various functionality of the application software
- IDE provides simulator support
- Simulator simulates target hardware and firmware execution can be inspected using simulators.

# Features of simulator based debugging

- Purely software based
- Doesn't require a real target system
- Very primitive (Lack of featured I/O support.)
- Lack of real time behavior

# Advantage of simulator based debugging

- **No need of target board**

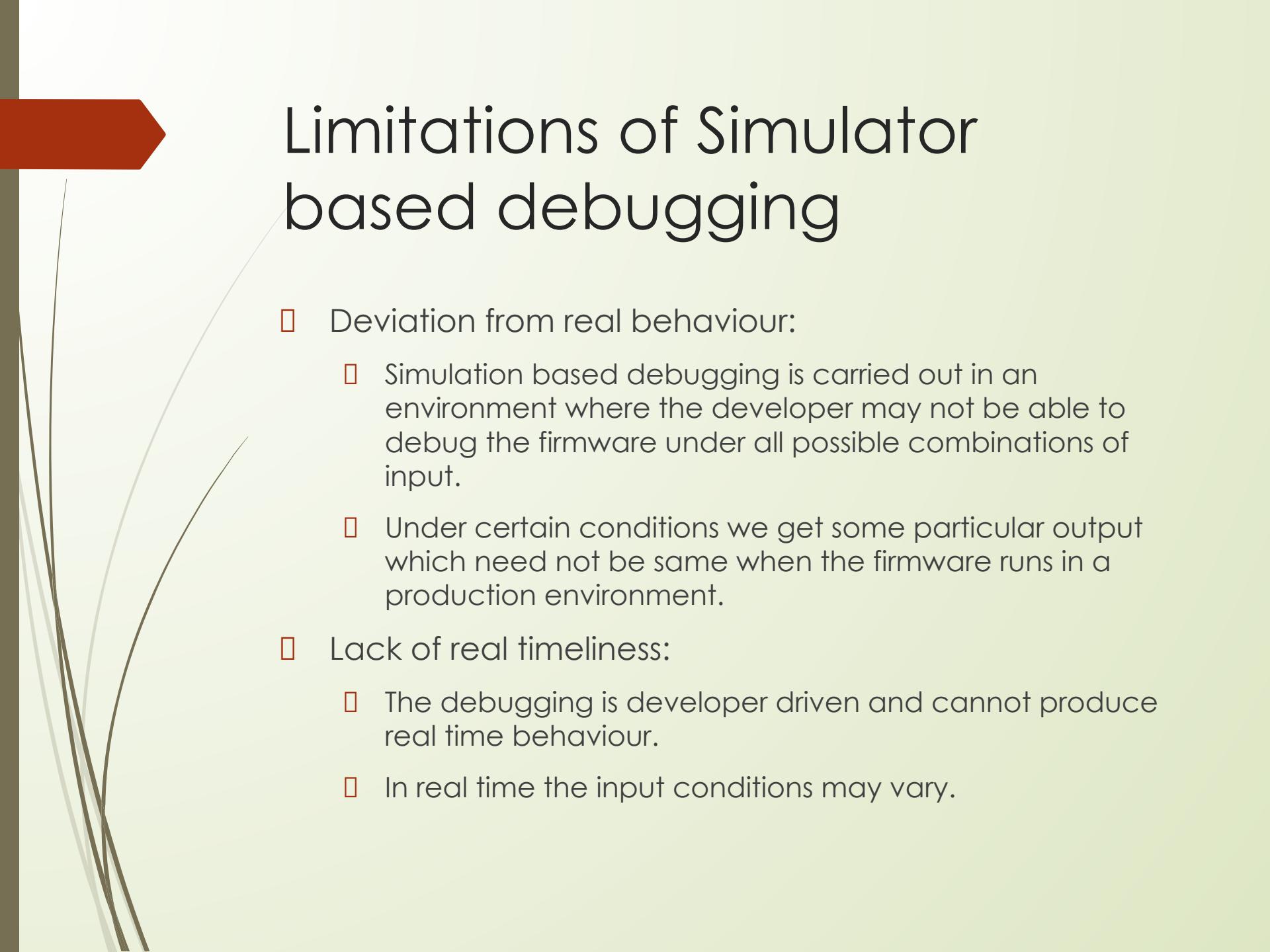
- Purely software oriented , IDE simulates the target board
- Since real hardware is not needed we can start immediately after the device interface and memory maps are finalized this saved development time
- User needs to know only the memory mapping of various devices with target board.
- After the device interface and memory map the developer can start programming

- **Simulated I/O peripherals**

- It provides an option to simulate various input output devices.
- Using simulators input output support we can edit the values for I/O registers and can be used as input/output values in firmware execution.
- It eliminates the need for connecting I/O devices for debugging the firmware.

- **Simulates abnormal conditions**

- Can input any parameter as input during debugging hence can check for abnormal conditions easily



# Limitations of Simulator based debugging

- Deviation from real behaviour:
  - Simulation based debugging is carried out in an environment where the developer may not be able to debug the firmware under all possible combinations of input.
  - Under certain conditions we get some particular output which need not be same when the firmware runs in a production environment.
- Lack of real timeliness:
  - The debugging is developer driven and cannot produce real time behaviour.
  - In real time the input conditions may vary.

# Debugging

- It is the process of diagnosing the firmware execution, monitoring the target processors registers and memory while the firmware is running.
- It also checks the signals from various embedded hardware.
- Two types of debugging
  - Hardware debugging:
    - deals with monitoring of various bus signals
    - Checking the status lines of target hardware.
  - Firmware debugging
    - Deals with examining the firmware execution, execution flow, changes to various CPU registers and status registers on execution of the firmware to ensure that the firmware is running as per design.

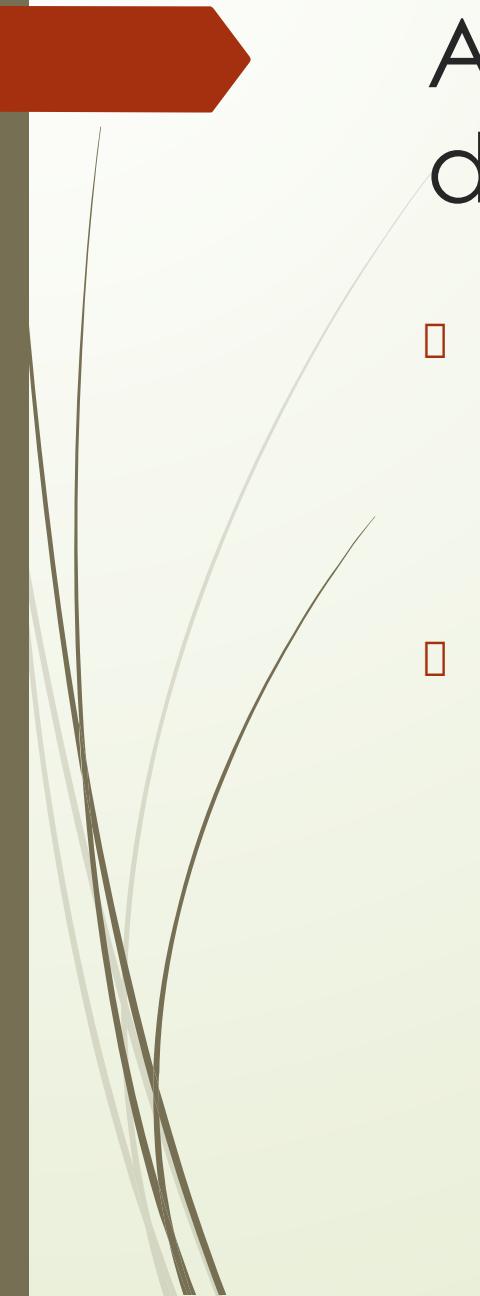
# Firmware debugging

- Performed to figure out the bug or the error in the firmware which creates unexpected behaviour.
- The main firmware debugging techniques are
  - Incremental EEPROM burning technique
  - Inline breakpoint based Firmware debugging
  - Monitor program based Firmware debugging
  - In Circuit Emulator based Firmware debugging



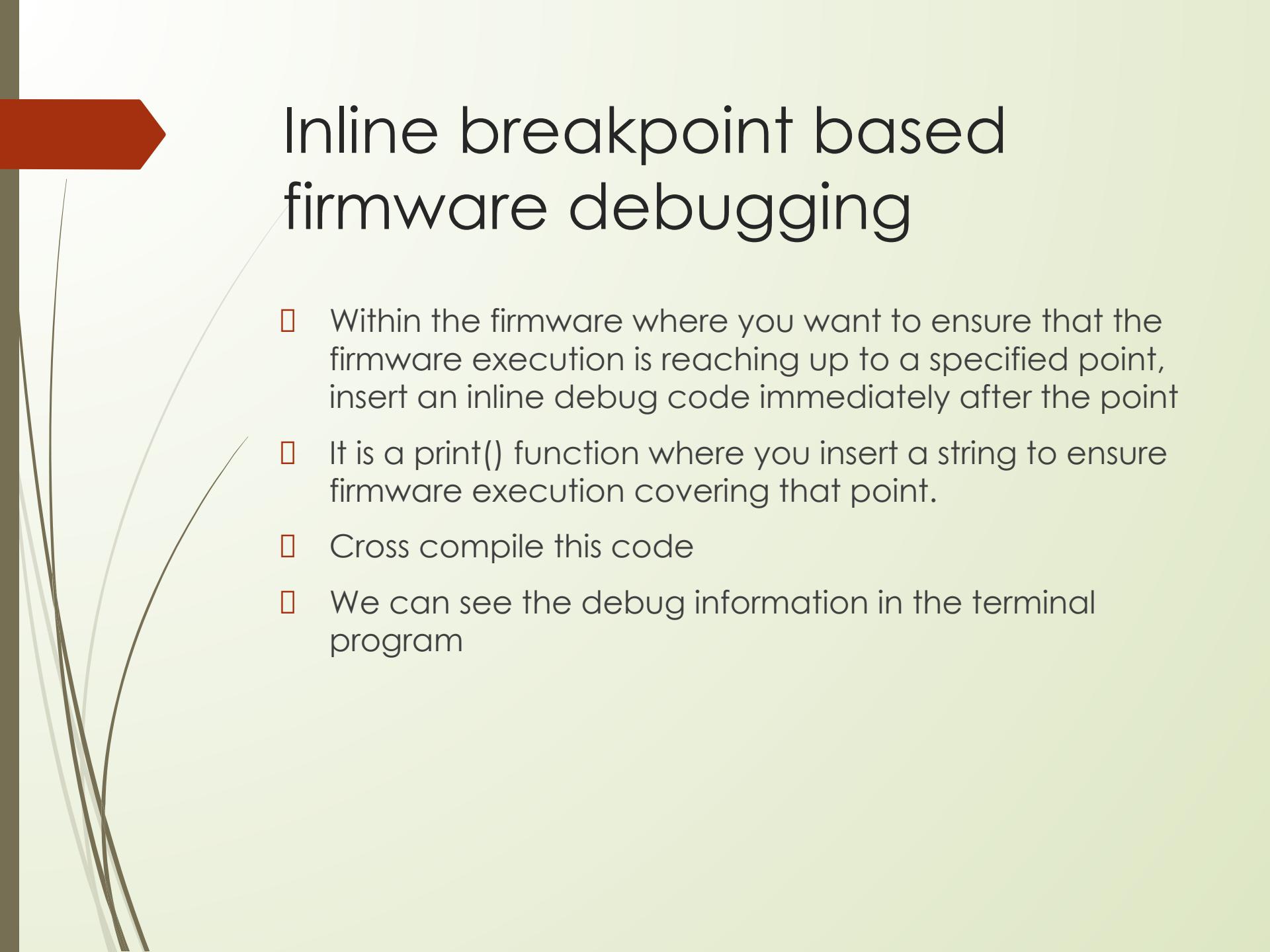
# Incremental EEPROM burning technique

- Most primitive type of firmware debugging technique.
- In this code is separated into different functional code units.
- Code is burnt in incremental order.
- The code corresponding to all functionalities are separately coded, cross-compiled and burned into the chip one by one.
- If the first functionality is working well, go for the next and so on.
- If all functionalities are working well combine the code for all functionalities, recompile and burn the code for the total system functioning.



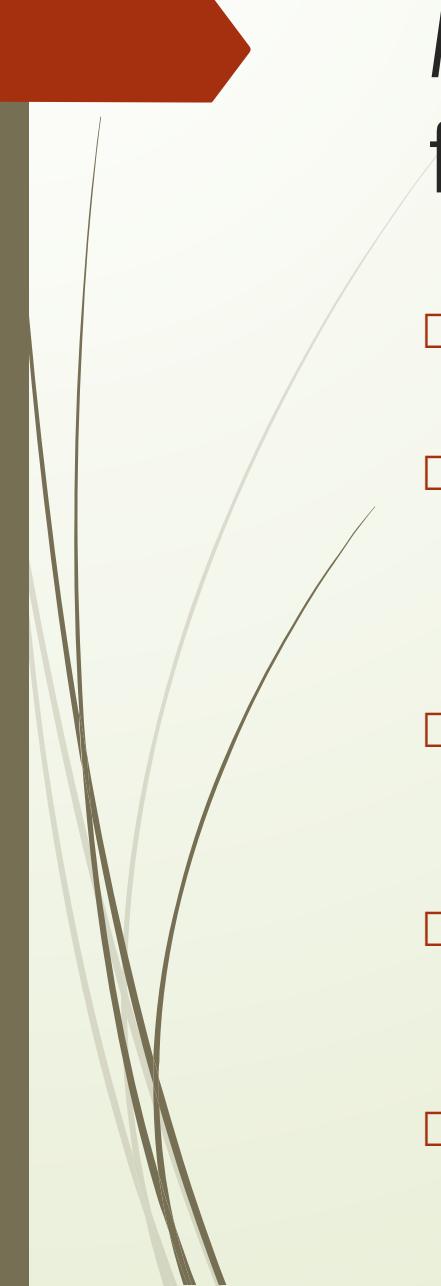
# Advantages and disadvantages

- Advantages
  - Once the testing is completed production can start.
  - Useful in small, simple systems
  - Useful when no debugging tools are available
- Disadvantages
  - Time consuming



# Inline breakpoint based firmware debugging

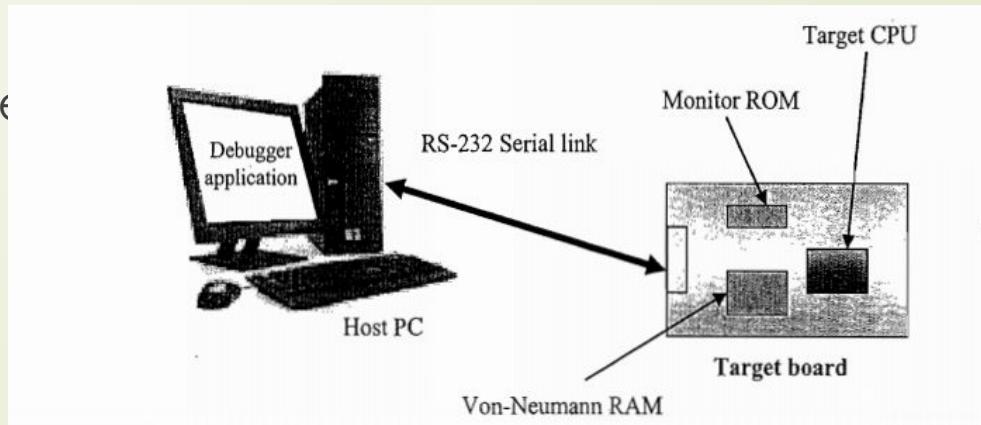
- Within the firmware where you want to ensure that the firmware execution is reaching up to a specified point, insert an inline debug code immediately after the point
- It is a print() function where you insert a string to ensure firmware execution covering that point.
- Cross compile this code
- We can see the debug information in the terminal program

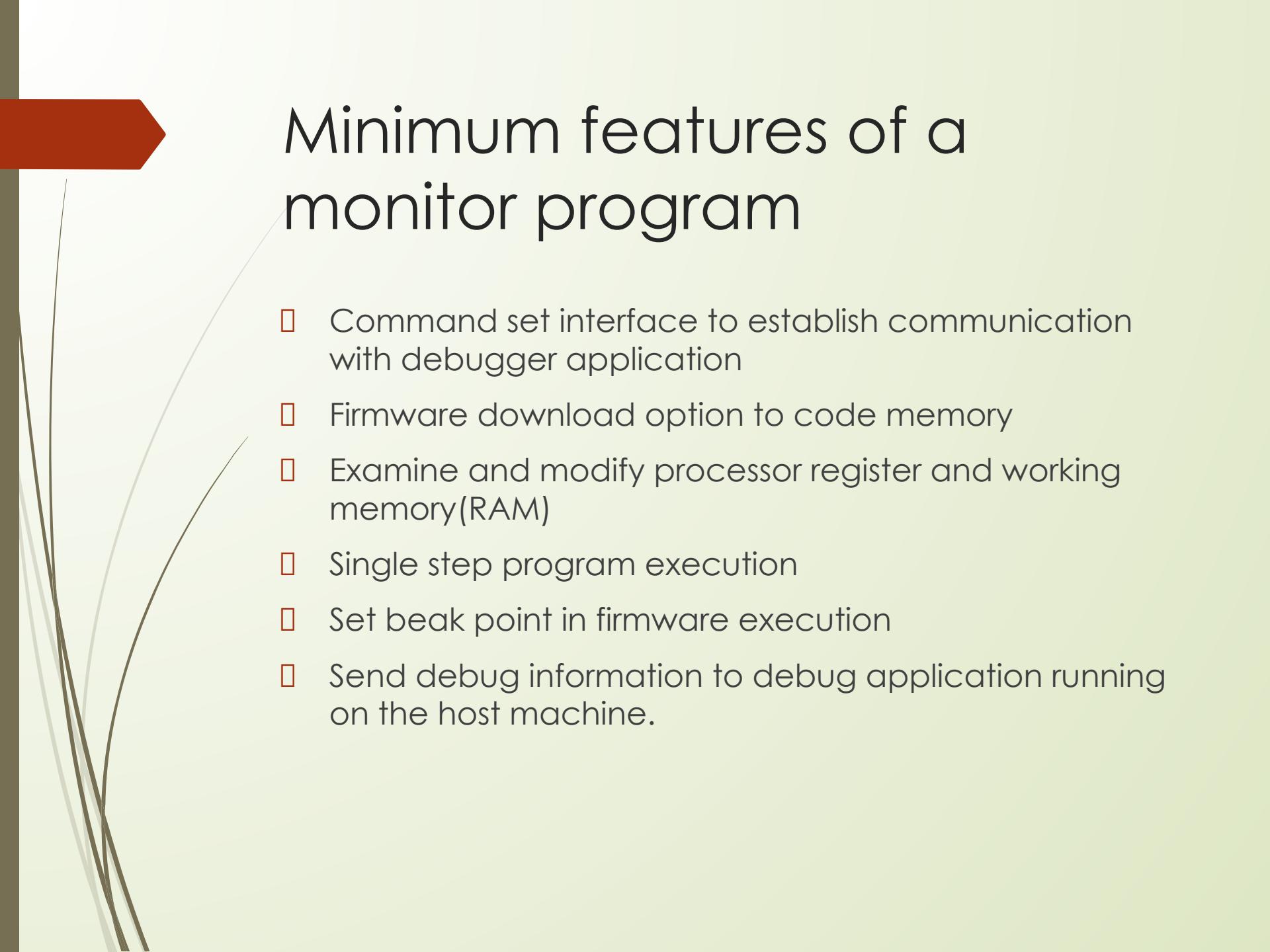


# Monitor Program based firmware debugging

- In this method a monitor program which acts as a supervisor is developed.
- The monitor program controls the downloading of the source code into the code memory, inspects or modify the register/ memory locations, allows stepping of source code etc.
- The monitor program implements debug functions as per pre defined command set from the debug application interface.
- The monitor program always listen to the serial port of the target device and according to the command received from the serial port it performs actions.
- The monitor program will handle the command reception from the serial interface.

- The entire code stuff handling the command reception and the corresponding action implementation is known as monitor program.
- After the successful completion of the monitor program, it is compiled and burned into FLASH memory or ROM of the target board.
- The





# Minimum features of a monitor program

- Command set interface to establish communication with debugger application
- Firmware download option to code memory
- Examine and modify processor register and working memory(RAM)
- Single step program execution
- Set break point in firmware execution
- Send debug information to debug application running on the host machine.

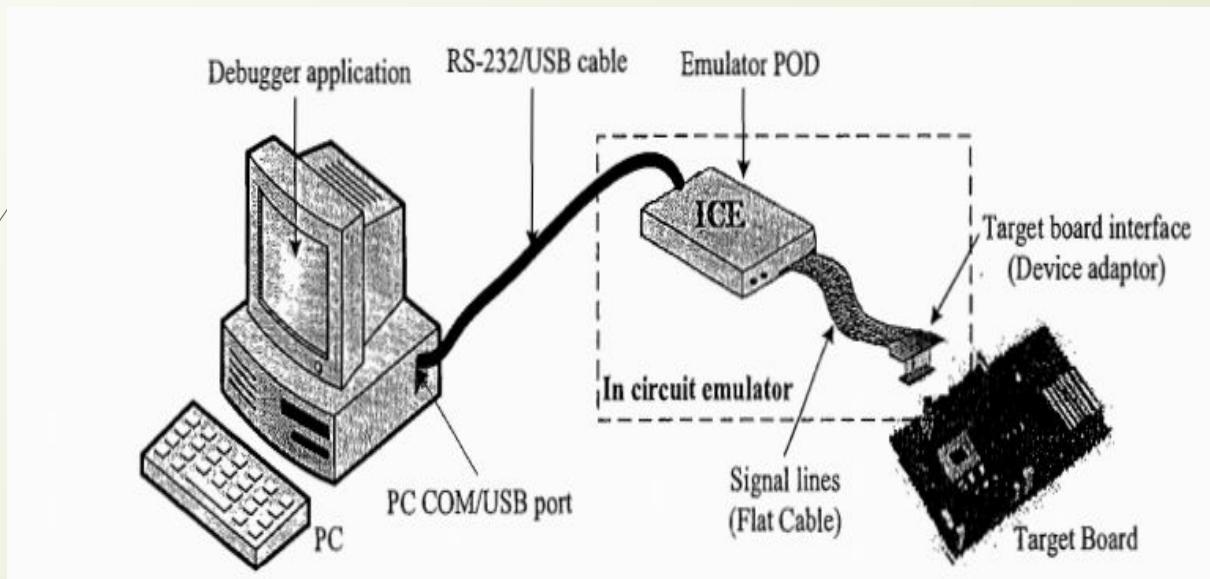
# Drawbacks

- In monitor based debugging shrinks the total available memory into Von-Neumann memory and it needs to accommodate all kinds of memory requirement.
- The serial port of the target processor is dedicated for the monitor applications and it cannot be used for any other device interfacing. Hence wastage of serial port.

# Comparison between simulator and Emulator

| Simulator   | Emulator  |
|---|---|
| A software application which duplicates the functionality of the features and instructions supported by target CPU. | Self contained hardware device which emulates the target CPU. It contains emulation logic, hooked into the debugging application running on the development PC on one end and connect through target board through some interface on the other end. |
| Simulates target board  | Emulates target board   |

# In Circuit Emulator based Firmware debugging





# Emulation Device

- It is a replica of the target CPU which receives various signal from the target board through a device adaptor connected to the target board and performs the execution of firmware under the control of debug command from the debug application.
- It may be a standard chip or a programmable Logic Device configured to function as the target CPU.
- If a standard chip is used it will provide real time behaviour. But cannot be used for any other device.
- PLD based chip can be easily reprogrammed to suit any other device of the same type.
- PLD based emulators are simple to implement for simple CPU., but accuracy is an issue in the replication of target functionalities.

# Emulation Memory

- It is the random access memory incorporated in the emulator device.
- It acts as a replacement of EEPROM where code memory is copied.
- It is called ROM Emulation.
- It eliminates the problems of ROM burning and helps for infinite number of re programming.
- It also acts as a trace buffer which stores instruction executed or registers modified or related data by the processor.

# Common features of trace buffer memory

- Records each bus cycle in a frame.
- Trace data can be viewed in the debugger application as Assembler or source code.
- Trace buffering can be done on the basis of a trace trigger
- It can record signals from the target board other than CPU signals.
- Trace data is very useful information in firmware debugging

# Emulator Control Logic

- Logic circuit used for implementing complex hardware breakpoints, trace buffer trigger detection, trace buffer control etc.
- Used for implementing logic analyser functions.

# Device Adaptors

- Act as an interface between the target board and emulator POD.
- They are pin-to pin compatible sockets.
- Connected using ribbon cables.

# Types of emulators

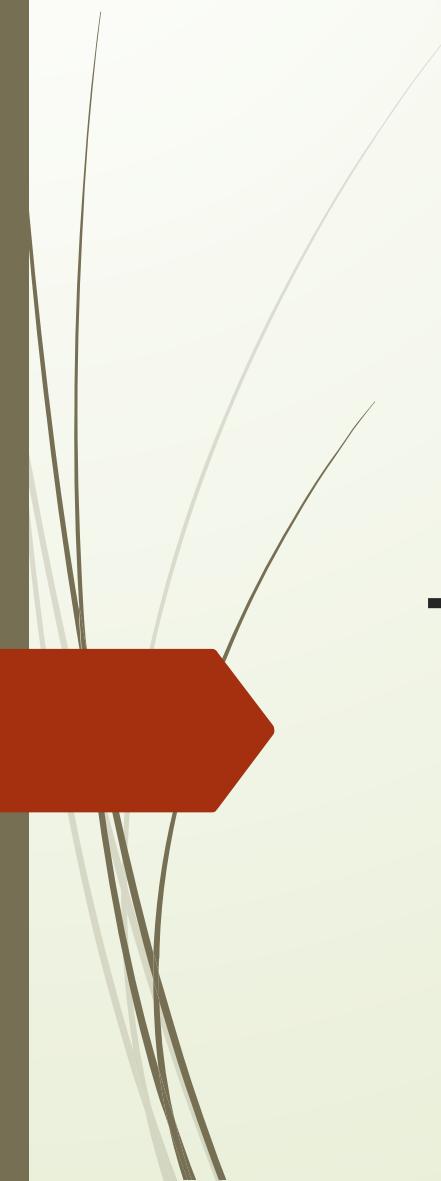
- Debug Board Modules
  - Combines emulation control logic and emulation device in a single board.
- Base terminal and probe card
  - This type supports a variety of processors.
  - Base terminal contains emulator hardware and control logic.
  - Base terminal is connected with development PC.
  - Emulation chip is mounted on a separate PCB and connected to the base terminal through a ribbon cable.
  - Probe card board contains device adaptor sockets to plug the board into the target development board.
  - Board containing emulation chip is the probe card.
  - Probe card is different for different CPUs

# On chip firmware debugging

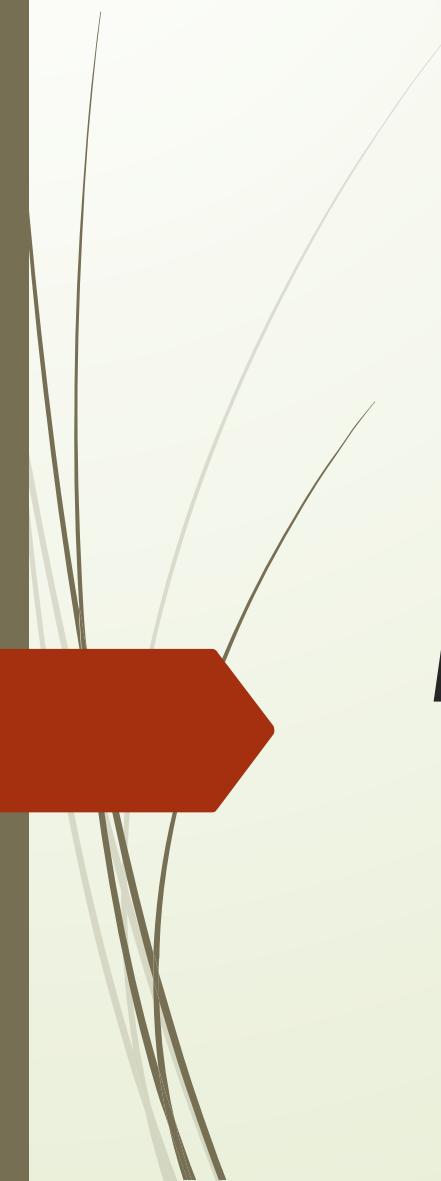
- They are built in debugging modules.
- They supports fast and efficient firmware debugging.
- Processor with OCD support incorporates a dedicated debugging module with existing architecture.
- OCD provides means to support simple breakpoint, query internal state of chip, and step through code.
- Dedicated registers are provided for debugging support.
- Debugging can be enabled using OCD enabled bit.
- Some hardware logic is implemented between CPU OCD and host PC to capture the debugging information.

# BDM

- Background debugging module is a proprietary on chip debug solution from Motorola
- It uses 10-26 pin connector to connect the target board.
  - Serial Data in (SDI)
  - Serial Data Out(SDO)
  - Serial clock are three important pins



# Thank You



# Module 5

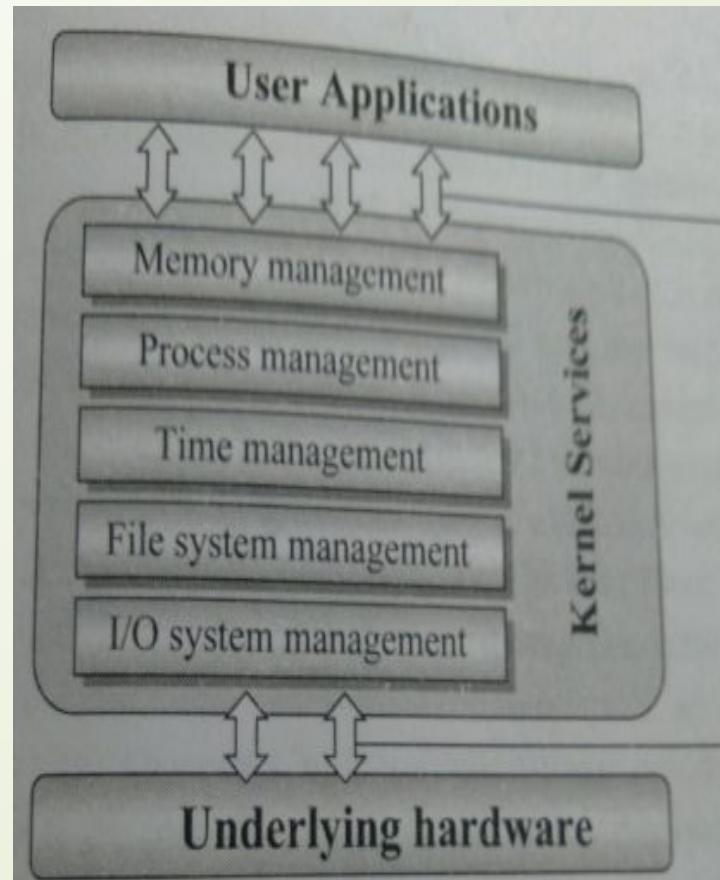
# Topics to cover

- RTOS based Design
  - Basic operating system services.
  - Interrupt handling in RTOS environment
  - Design Principles
  - Task scheduling models.
  - How to Choose an RTOS.
  - Case Study – MicroC/OS-II.

# Introduction: OS

- OS acts as an interface between user applications and underlying system resources through a set of system functionalities and services.
- The OS manages the resources and make them available to the user applications.
- Primary functions of OS is
  - Make the system convenient to use
  - Organize and use the system resources effectively and correctly.

# OS Architecture



Application  
Programmin  
g Interface

Device  
Driver  
Interface

# The Kernel

- Kernel is the core of the OS
- Manages system resources and communication among the hardware and other services.
- Kernel contains set of system resources and libraries.

# Services provided by Kernel

- Process management
  - Deals with managing process or tasks
  - Includes
    - Setting up of memory space for the process
    - Loading the process code in to the memory space, allocating system resources, scheduling and managing execution of the process
    - Setting up and managing PCB
    - Inter process communication and synchronization
    - Process termination and deletion

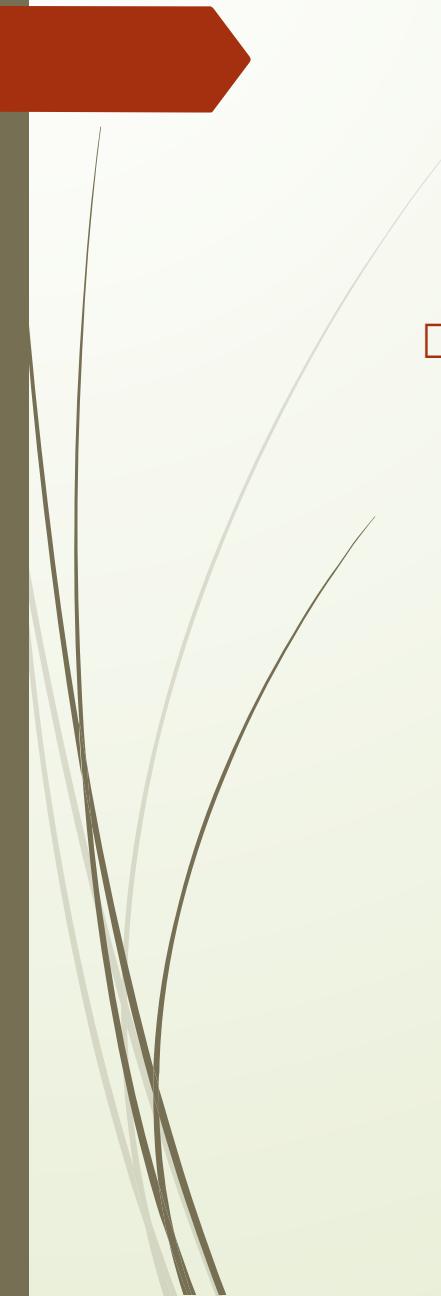


## □ Primary Memory management

- MMU is responsible for
  - Keeping track of which part of the memory area is currently used by which process
  - Allocating and deallocating memory space on a need basis

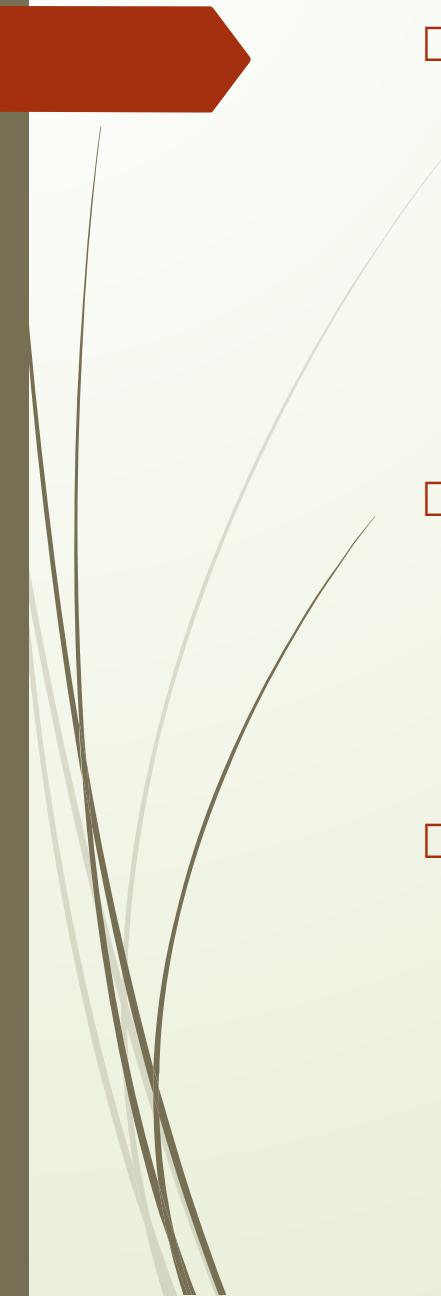
## □ File System Management

- File is a collection of related information
- there are different types of files which stores different kinds of data.
- FSM is responsible for
  - Creation, deletion and alteration of files
  - Creation, deletion and alteration of directories
  - Saving files in the secondary storage
  - Providing automatic allocation of file space based on the amount of free space available.
  - Providing naming conventions for file.



## □ I/O System Management

- Routes the I/O request coming from different user application to the appropriate I/O devices of the system.
- Device manager is responsible for the management of device related operations.
- Kernel communicates to the I/O devices using system calls, which are implemented in a service called device driver.
- Each device has its own device driver.
- Device manager is responsible for
  - Loading and unloading of device drivers
  - Exchanging information and system specific control signals to and from the device.



## □ Secondary Storage Management

- Deals with secondary storage memory devices.
- The SSM deals with
  - Disk storage allocation
  - Disk scheduling
  - Free disk space management

## □ Protection System

- Modern OS supports multiple users with different access permissions.
- Protection deals with implementing security policies to restrict the access to both user and system resources.

## □ Interrupt handler

- Handles different kinds of interrupt.

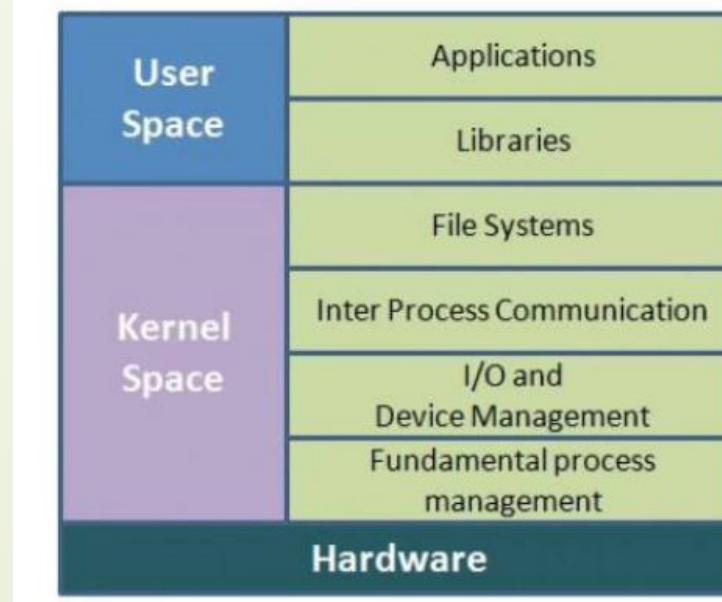
# Kernel space and User space

- The memory space of the primary memory where the kernel code is placed is called Kernel space.
- The memory space of the primary memory where the user program code is placed is called User space.
- The partition of the memory into user space and kernel space completely depends on OS.

# Monolithic kernel and Micro kernel

- Based on the kernel design the kernel is divided into monolithic and micro kernels.
- The entire OS is working in the kernel space.
- In monolithic kernel all kernel modules runs within the same memory space under a single kernel thread.
- The tight internal integration of the monolithic kernel allows effective utilization of low level features of the underlying system.
- Drawback
  - Any error in any one of the kernel module crash the entire system.
  - Eg: LINUX, DOLARIS, MS DOS

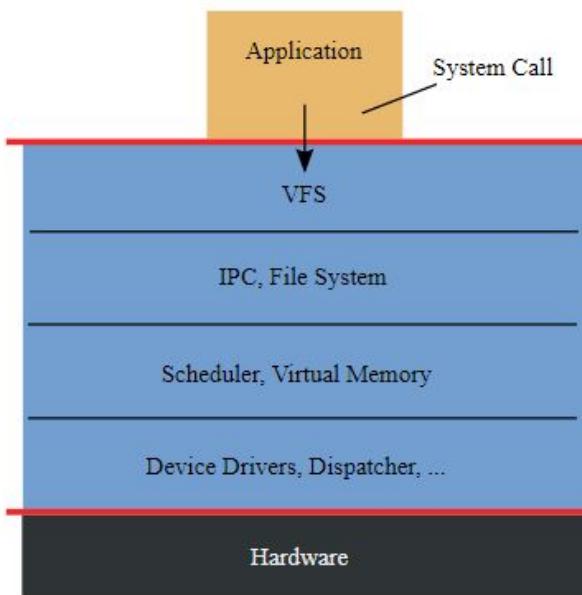
# Monolithic Kernel Model



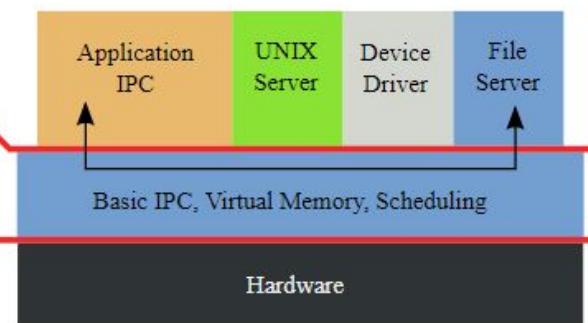
# Microkernel

- Only the essential part of OS are in the Kernel space.
- Rest of the OS services are implemented in programs known as servers which runs in user space.
- The essential functions are memory management, process management, timer system and interrupt handlers.
- Eg: Mach, ONX, Minix 3

## Monolithic Kernel based Operating System



## Microkernel based Operating System



# Benefits

- Robustness
  - If any problem occurs in server program, it can be reconfigured and restarted without disturbing OS. This is good for system which required high availability
- Configurability
  - The same feature make it dynamically configurable.

# Types of OS

- General Purpose OS
  - This is the OS deployed in general purpose OS
  - It is non deterministic in behavior
  - The services can inject random delays, and may cause slow responsiveness of an application at unexpected times.
  - Eg: Windows10/8.x/XP/MS-DOS

# Types of OS

- Real Time OS
  - It implies deterministic behavior
  - OS services consumes only known and expected amount of time regardless of the number of services.
  - It implements policies and rules concerning time critical allocation of systems resources.
  - RTOS will decides which application will run in which order and how much time need to be allocated for each application.
  - Example : QNX, VxWorks MicroC/OS-II

# Real Time Kernel

- Kernel of real time OS
- It contains only minimal set of services required to run the user applications or tasks
- The function of RTK are,
  - Task/process management
  - Task/process scheduling
  - Task/process synchronization
  - Memory management
  - Interrupt handling
  - Time Management

# Task/process management

- Deals with,
  - Setting up the memory space for the tasks
  - Loading the task code into the memory space
  - Allocating system resources
  - Setting up task control block
  - Process termination and deletion

# Task Control Block

- Holding the information corresponding to a task
  - Task ID: task identification no
  - Task State: current state of the task
  - Task type: indicate the type of task whether it is hard real time or soft real time
  - Task priority:
  - Task context pointer: pointer for context saving
  - Task memory pointer: pointer to the code memory, data memory and stack memory for the task
  - Task system resource pointer: pointer to system resources
  - Task pointer: pointer to other TCBs
  - Other parameters

# Function of task manager

- Creates a TCB for a task on creating a task
- Delete/remove the TCB of a task when the task is terminated or deleted.
- Reads the TCB to get the state of a task
- Update the TCB with updated parameters on need basis
- Modify the TCB to change the priority of the task dynamically

# Task/process scheduling

- Deals with sharing of CPU among various task/process.
- A program called scheduler will take care of it

# Task/process synchronization

- Deals with optimizing the current access of a resource, which is shared among multiple task and communicates between various tasks.

# Error/Exception handling

- Deals with registering and handling of error or exceptions occurred during the execution of a task.
- Example for error/exception
  - Insufficient memory
  - Timeout
  - Deadlock
  - Deadline missing
  - Bus error
  - Divide by zero
  - Unknown instruction execution

# Error/Exception handling

- Error or exception can handle at kernel level(dead lock) or task level (timeout).
- OS kernel gives information about the error in the form of API.
- Watchdog is a program which handles timeout.

# Memory Management

- MM of RTOS is different from GPOS
- RTOS uses block based memory allocation instead of dynamic allocation of various sized blocks.
- Blocks are of fixed sized and allocation of blocks to a process is demand based.
- The blocks are stored in a free buffer queue
- The tasks are allowed to access any free memory blocks to avoid timing overhead.
- This blocks will be allotted as a unit.
- Hence there is no memory fragmentation issue.
- The garbage collection overhead also avoided.
- Certain RTOS supports virtual memory concept.

# Interrupt handling

- Handles various interrupts
- Interrupts tell the system that an external device or task needs immediate attention of the CPU
- Interrupts can be synchronous or asynchronous
- Interrupts which occurs in sync with currently executing task is called synchronous interrupts.
- Software interrupts are synchronous interrupts.
  - Eg: divide by zero, memory segmentation error
- For synchronous interrupts the interrupt handler works in the same context of interrupting task

- Asynchronous interrupts occurs at any point of execution of any task.
- It is not in sync with currently executing task
- Examples: interrupts generated by external devices connected to the processor, timer overflow interrupts etc.
- Here interrupt handler is always a different program and works in different context.
- Hence context switch takes place while handling the asynchronous interrupts.
- Priority levels can be assigned to interrupts.
- Each interrupts can be enabled or disabled individually
- Nesting of interrupts is also possible which allows servicing of an interrupt by high priority interrupt.

# Time management

- The time reference to kernel is given by high resolution real time clock hardware chip.
- Hardware timer is programmed to interrupt the processor or controller at a fixed rate
- Timer interrupt is called as timer tick
- Timer tick is taken as the timing reference by the kernel
- It vary depends on the hardware timer
- The system timer is updated based on timer tick.

- If the system time register is 32 bits and the timer tick interval is 1 microseconds, the system register will reset in
$$2^{32} \cdot 10^{-6} / (24 \cdot 60 \cdot 60) = 1.19 \text{ hours.}$$
- If the system time register is 32 bits and the timer tick interval is 1 milliseconds, the system register will reset in
$$2^{32} \cdot 10^{-3} / (24 \cdot 60 \cdot 60) = 50 \text{ days}$$
- Timer tick interval is handled by timer interrupt

# Usage of timer tick interrupt

- Save the current context
- Increment the system time register by one
- Generate timing error and reset system time register if the timer tick count is greater than the maximum range available for system time register
- Update timers implemented in kernel
- Activate the periodic task which are in idle
- Invoke the scheduler and schedules the task again based on the scheduling algorithm
- Delete all the terminated task and their associated data structures.
- Load the context in the first task in the ready queue.

# Hard Real time

- Real time operating system that strictly adhere to the timing constraint for a task is called hard real time system
- They must meet deadline for a task without any slippage
- Missing any deadline may cause catastrophic results.
- Proper scheduling is needed to achieve this result.
- Eg: airbag control system, antilock braking system
- There is no human interaction in this
- Such system will not use virtual memory concept as it create delays due to context switching.

# Soft real time

- Real time OS which does not guarantee to meet the deadline, but offers the best effort to meet the deadline is called soft real time systems.
- It is acceptable if the frequency of deadline missing is within the compliance limit of the quality of service.
- There is the presence of a human in the loop
- Eg: ATM, audio video play back

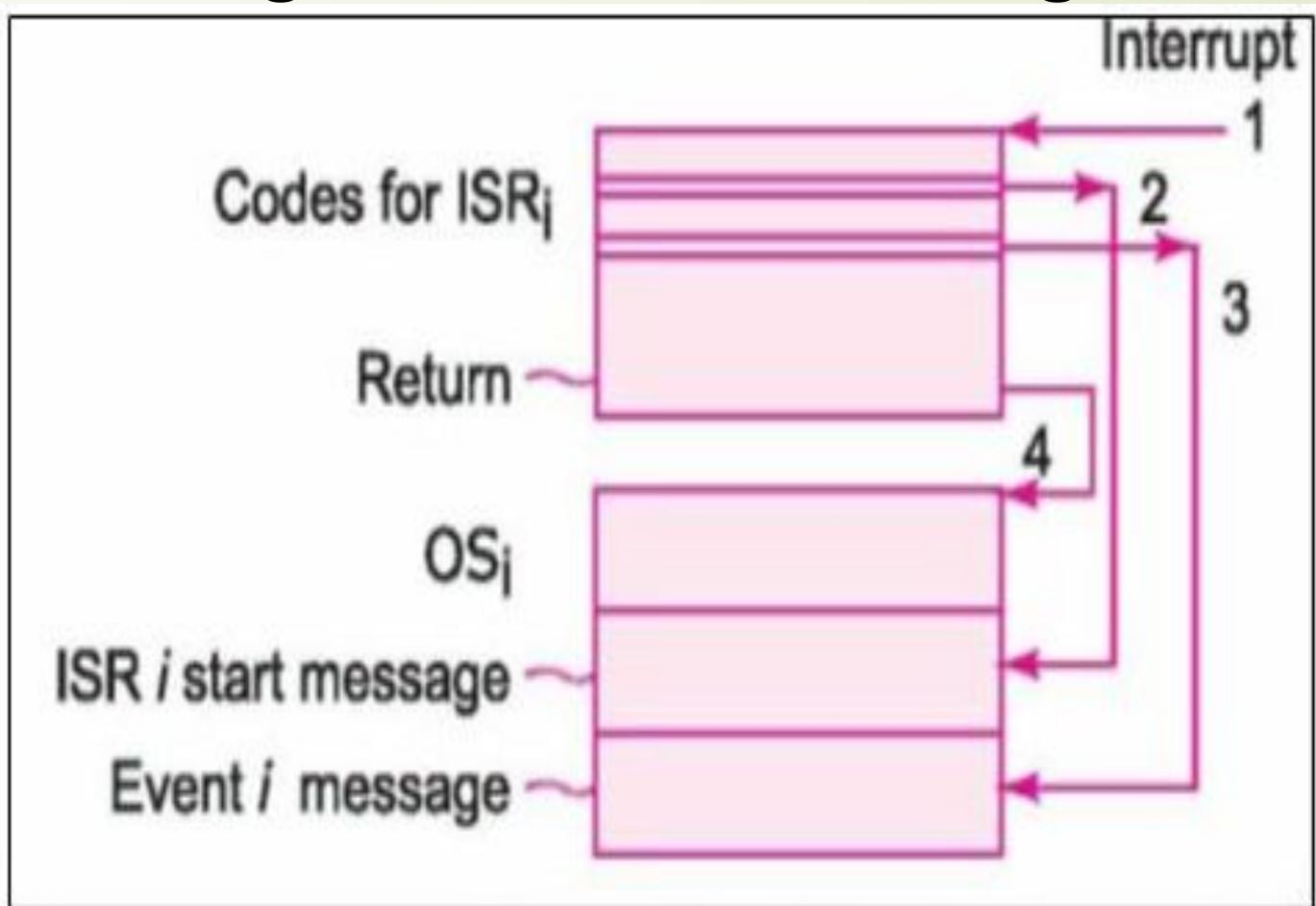
# Interrupt Handling in RTOS Environment

- In a system, the ISRs should function as follows:
  - ISRs have higher priorities over the OS functions and the application tasks. An ISR does not wait for a semaphore, mailbox message or message queue.
  - An ISR does not also wait for mutex else it has to wait for other critical section code to finish before the critical codes in the ISR can run.
- There are three alternative systems for the OSes to respond to the hardware source calls from the interrupts.
  - Direct call to an ISR by an Interrupting source and ISR sending an ISR Enter Message.
  - RTOS First Interrupting on an Interrupt, then OS calling the corresponding ISR.
  - RTOS First Interrupting on an Interrupt, then RTOS Initiating the ISR and then an IST.

# Direct call to an ISR by an Interrupting source and ISR sending an ISR Enter Message

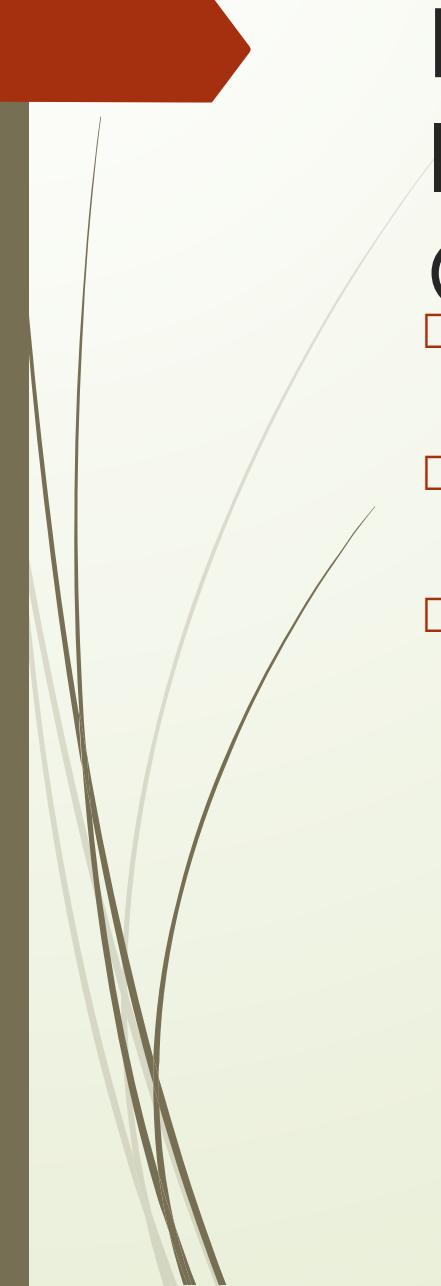
- On an interrupt, the process running at the CPU is interrupted and the ISR corresponding to that source starts executing (step 1).
- A hardware source calls an ISR directly.
- The ISR just sends an **ISR enter message** to the OS (step 2).
- Later the ISR code can send into a mailbox or message queue (step 3).
- The task waiting for the mailbox or message queue does not start before the return from the ISR
- ISR enter message in step 2 is to inform the OS that an ISR has taken control of the CPU.
- The ISR continues execution of the codes needed for the interrupt service till the **ISR exit message** is sent just before the return (Step 4).

# Direct call to an ISR by an Interrupting source and ISR sending an ISR Enter Message



# Direct call to an ISR by an Interrupting source and ISR sending an ISR Enter Message

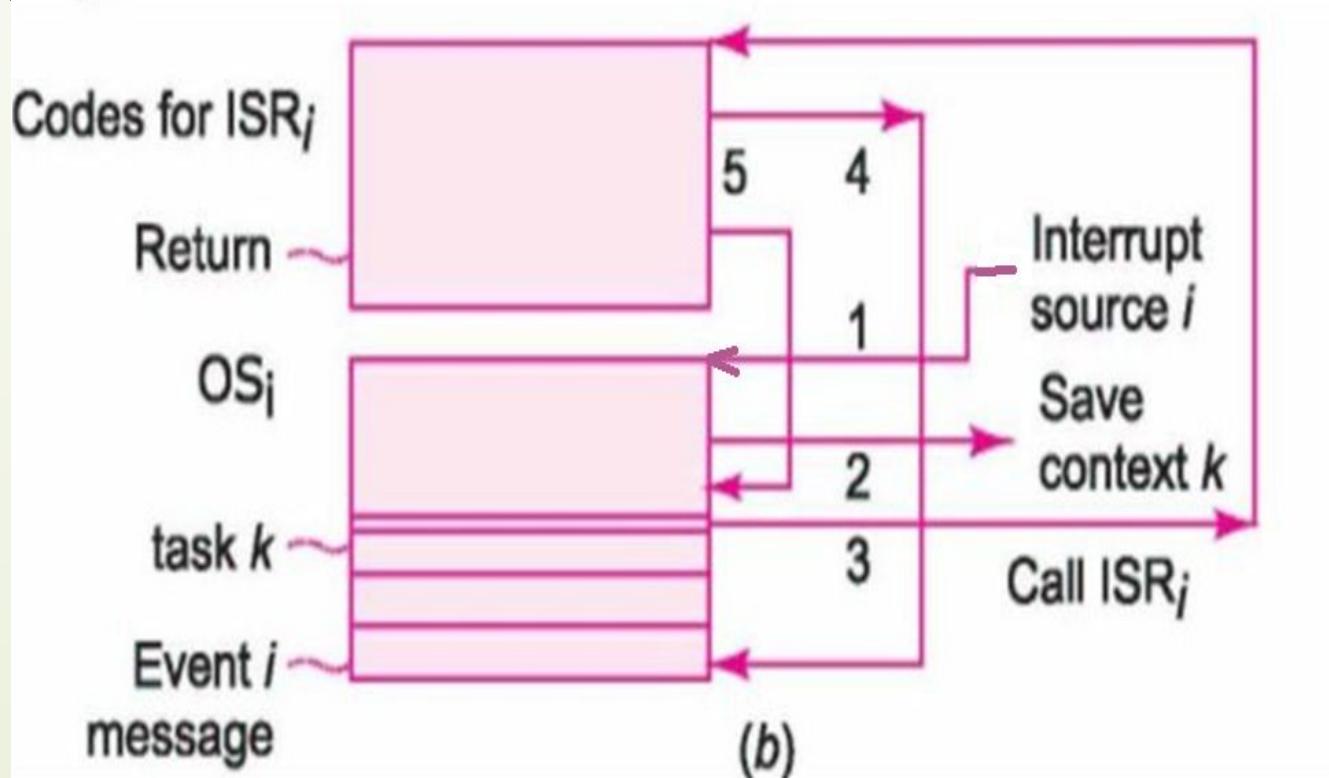
- There are two functions, ISR and OS functions, in two memory blocks.
- An i-th interrupt source causes i-th ISR,  $ISR_i$ , to execute.
- The routine sends an ISR enter message to the OS.
- The message is stored at the memory allotted for OS messages.
- When the ISR finishes, it sends ISR exit to the OS.
- There is a function `OSISRSemPost()`.
- The ISR semaphore is a special semaphore, which `OSISRSemPost()` posts and on return from the OS to be taken by the calling ISR itself.



# RTOS First Interrupting on an Interrupt, then OS Calling the Corresponding ISR

- On interrupt of a task, say, k-th task, the OS first gets the hardware source call(step 1).
- It then initiates the corresponding ISR after saving the present process status(or context) (step 2).
- The called ISR (Step 3) during execution then can post one or more outputs(step 4) for the events and messages into the mailboxes or queues.

# RTOS First Interrupting on an Interrupt, then OS Calling the Corresponding ISR



# RTOS First Interrupting on an Interrupt, then OS Calling the Corresponding ISR

- Assume that there are the routine(i-th ISR) and two processes(OS and j-th task) in three memory blocks other than the interrupted k-th task.
- An i-th interrupt source causes the OS to get the notice of that, then after step 1 it finishes the critical code till the pre-emption point and calls the i-th ISR.
- ISR, executes (step 3) after saving the context(step 2) onto a stack.
- The preemption point is the last instruction of the critical part of the presently running OS function, after which the ISR being of highest priority is called.
- The ISR in step 4 can post the event or mailbox messages to the OS for initiating the j-th task or k-th task after the return(step 5) from the ISR and after retrieving the j-th or k-th task context.
- The OS initiates the j-th task if it is of higher priority than the interrupted k-th task, or runs the interrupted k-th task.

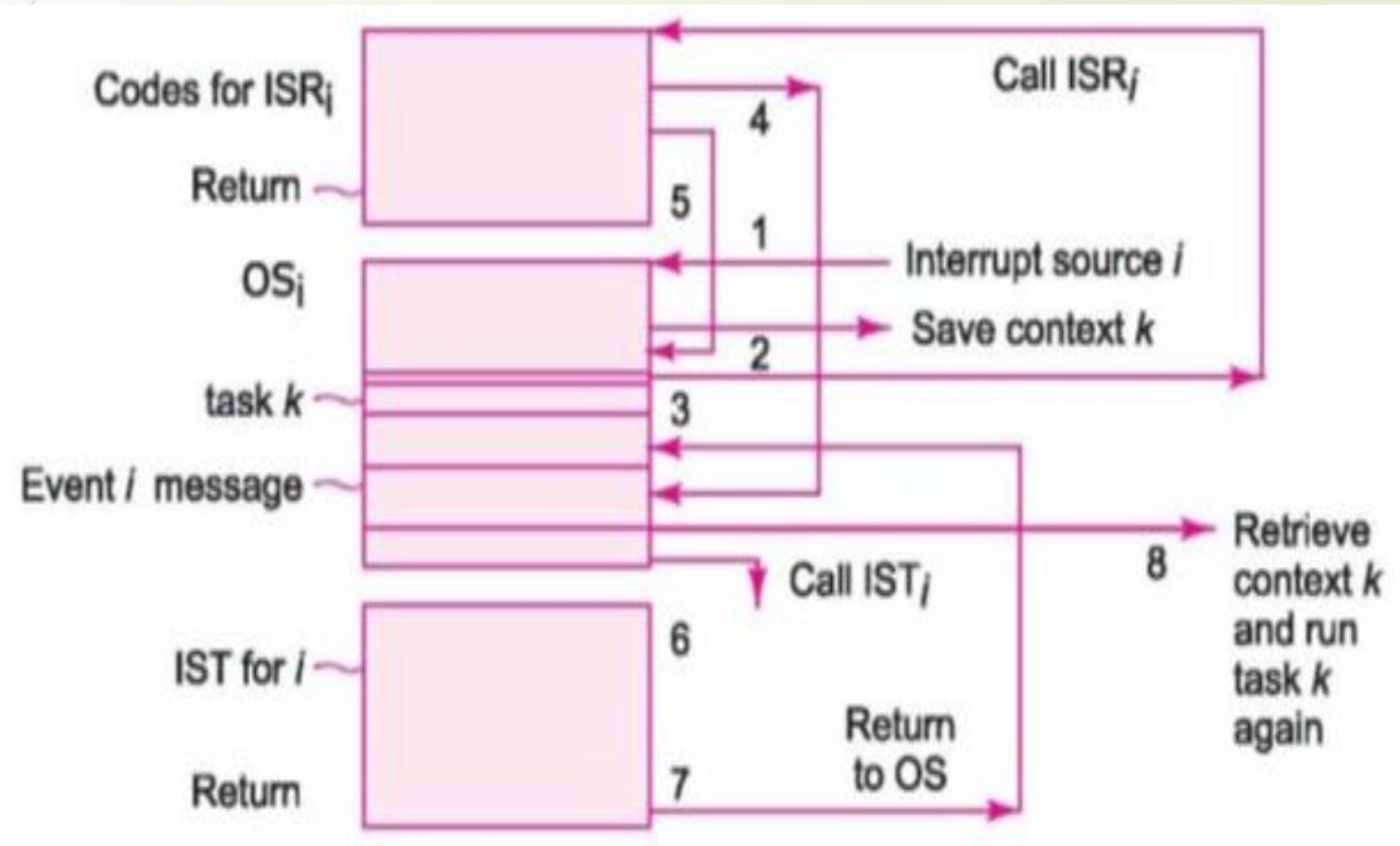
# RTOS First Interrupting on an Interrupt, then RTOS Initiating the ISR and then an ISR

- An RTOS can provide for two levels of ISRs, a fast- level ISR (FLISR) and a slow- level ISR(SLISR).
- FLISR- Hardware interrupt ISR of just ISR.
- SLISR- Software interrupt ISR (interrupt service thread-IST).

# RTOS First Interrupting on an Interrupt, then RTOS Initiating the ISR and then an IST

- On interrupt, the RTOS first gets the hardware source call(step 1)
- It then initiates the corresponding ISR after finishing the critical section and reaching the preemption point and then saving the processor status or context(step 2).
- The ISR executes the device- and platform- dependent code(step 3).
- The ISR at the start can mask (disable) further preemption from the same or other hardware sources.
- The ISR during execution then can send one or more outputs for the events and messages into the mailboxes or queues for the ISTs (step 4).

# RTOS First Interrupting on an Interrupt, then RTOS Initiating the ISR and then an IST



# RTOS First Interrupting on an Interrupt, then RTOS Initiating the ISR and then an IST

- The ISR just before the end, unmasks(enable) further preemption from the same or other hardware sources (step 5).
- The ISTs in the FIFO that have received the messages from the ISRs executes(step 6) as per their priorities on return (step 5) from the ISR.
- When no ISR or IST is pending execution in the FIFO, the interrupted task runs(step 8) on return (step 7).

# Task, Process and Threads

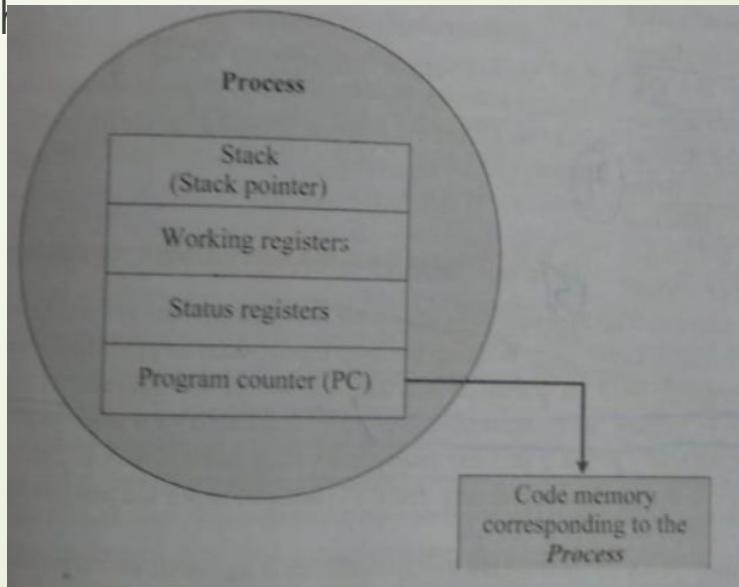
- Task is the program in execution and the related information stored by the OS.
- Task, process, job all are refers to the same entity

# Process

- Process is a program or part of a program in execution
- It is also called as an instance of a program
- Multiple instance of the same program can execute simultaneously

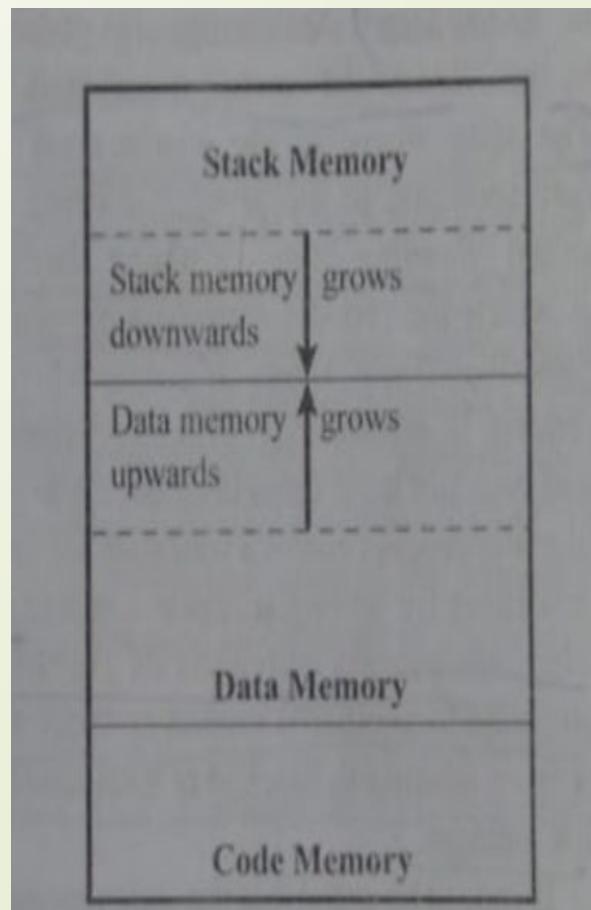
# Structure of a process

- A process holds a set of registers, process status, program counter, to point to the next executable state of the process, a stack for holding the local variables associated with process and the code corresponding to the process.



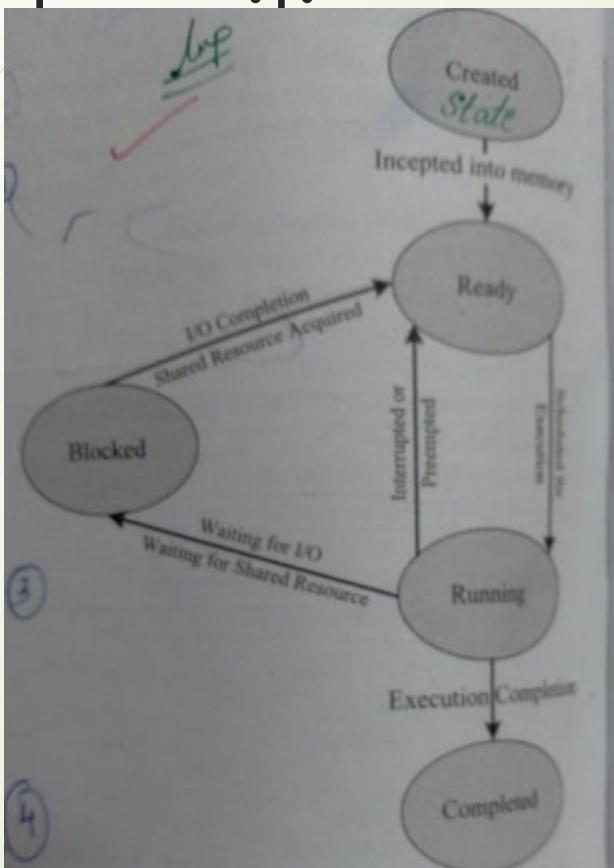
- A process which inherits all the properties of CPU is considered as a virtual processor waiting to get the chance to run in CPU.
- When the process gets its chance its registers and program counter registers becomes mapped to the physical registers of the CPU from a memory.
- The memory occupied by the process is segregated into three regions namely stack memory, data memory and code memory.

# Memory organization of a process



- The stack memory holds all the temporary data such as variable local to the process.
- The data memory holds all global data for the process.
- The code memory contains the program code corresponding to the process.
- The stack memory usually starts at the highest memory address from the memory area allocated for the process.
- Eg: the memory map of the memory area allocated for the process is 2048 to 2100, the stack memory starts at address 2100 and grows downwards to accommodate the variables local to the process.

# Process state and state



- The process traverse through a series of states during its transition from the newly created state to the terminated state.
- The cycle through which a process changes its state from newly created to execution completed is known as process life cycle.
- The state at which a process is being created is called as created state.
- The OS recognizes a process in the created state but no resources are allocated to the process
- The state where a process is incepted into the memory and awaiting the processor time for execution is known as ready state.
- At this state the process is placed in the ready queue maintained by the OS.

- In this state where in the source code instruction corresponding to the process being executed is called the running state
- Blocked state /wait state refers to a state where a running process is temporarily suspended from execution and does not have immediate access to resources.
- The blocked state might be invoked by various conditions like the process enters a wait state for an event to occur or waiting for getting access to a shared resource.
- A state where the processes completes its execution is known as completed state.
- The transition of a process from one state to another state is called as state transition.

- 
- When a process changes its state from ready to running or from running to blocked or terminated or from running the CPU allocation for the process may also change.

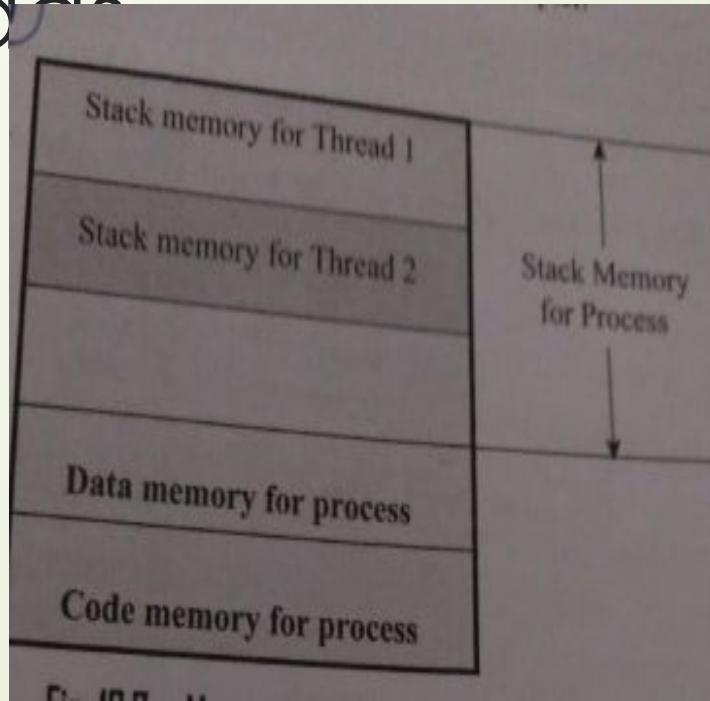
# Process management

- It deals with,
  - the creation of a process
  - Setting of the memory space for the process
  - Loading the process's code into the memory space,
  - Allocating system resources
  - Setting up the PCB for the process
  - Termination and deletion of process.

# Threads

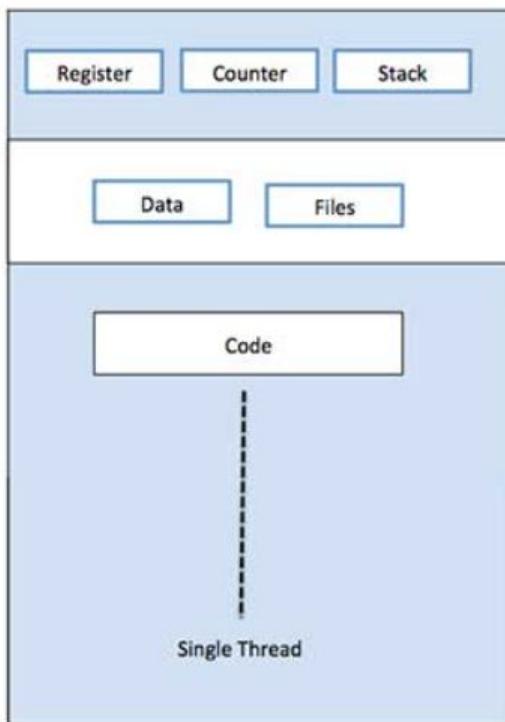
- A thread is the primitive that can execute code
- A thread is a single sequential flow of control within a process.
- It is a light weight process
- A process can have many threads of execution
- Different threads which are part of a process share the same address space.
- They share the same code memory, data memory, code memory and heap memory area.
- Thread maintain their own thread status, PC and stack.

# Memory organization of a process and its associated Threads

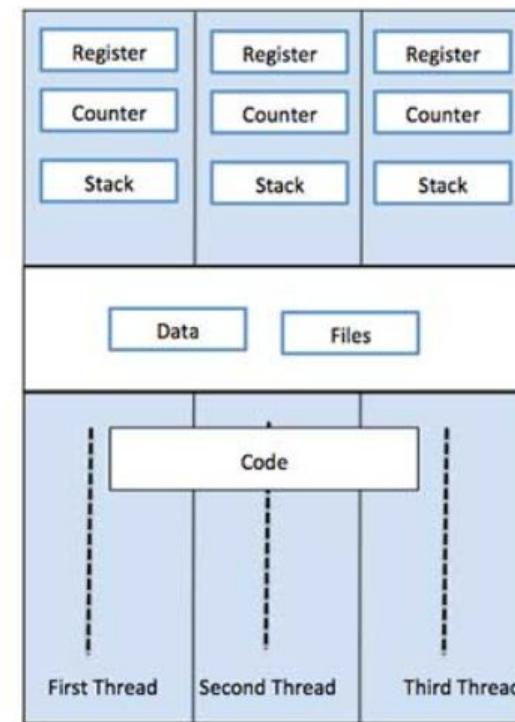


# Multithreading

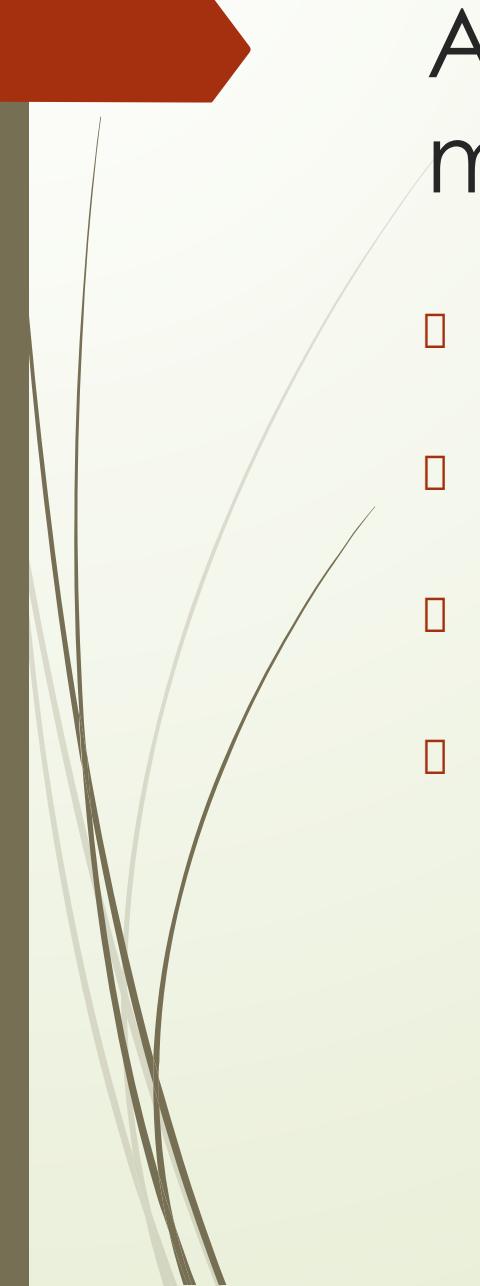
- A process in ES may consists of sub operations.
- If these subtasks are executed sequentially, CPU utilization will not be effective
- If a task enter into wait state due to some I/o , the task can again be split into different threads performing different operations, the CPU can be effectively utilized.



Single Process P with single thread



Single Process P with three threads

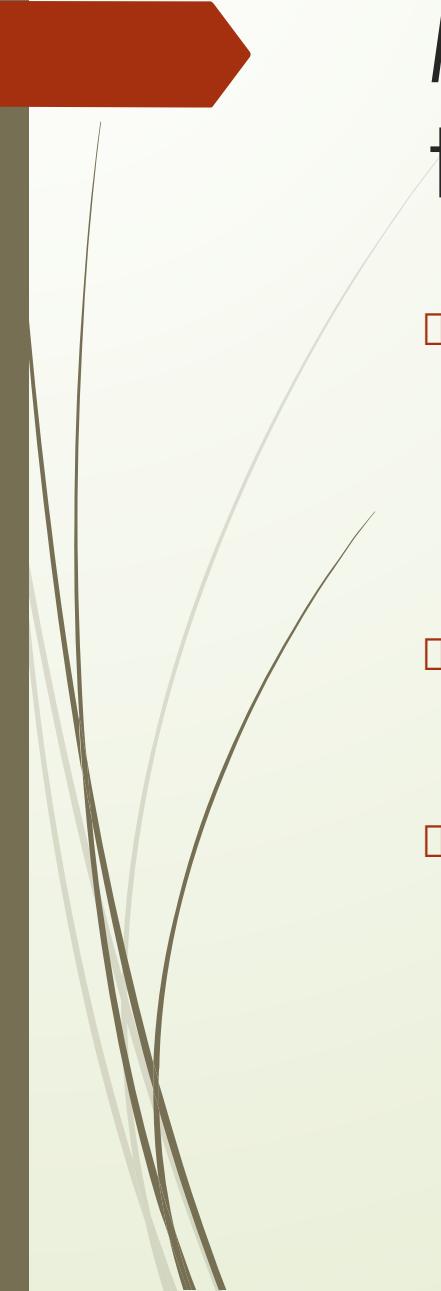


# Advantages of using multithreading

- Better memory utilization as the threads of the same process share the address space for data memory
- Interthread communication can be made simple by sharing the variables across threads.
- When one thread enters a wait state, others can utilize the CPU. Hence maximum utilization of CPU
- Process execution speed is increased due to this

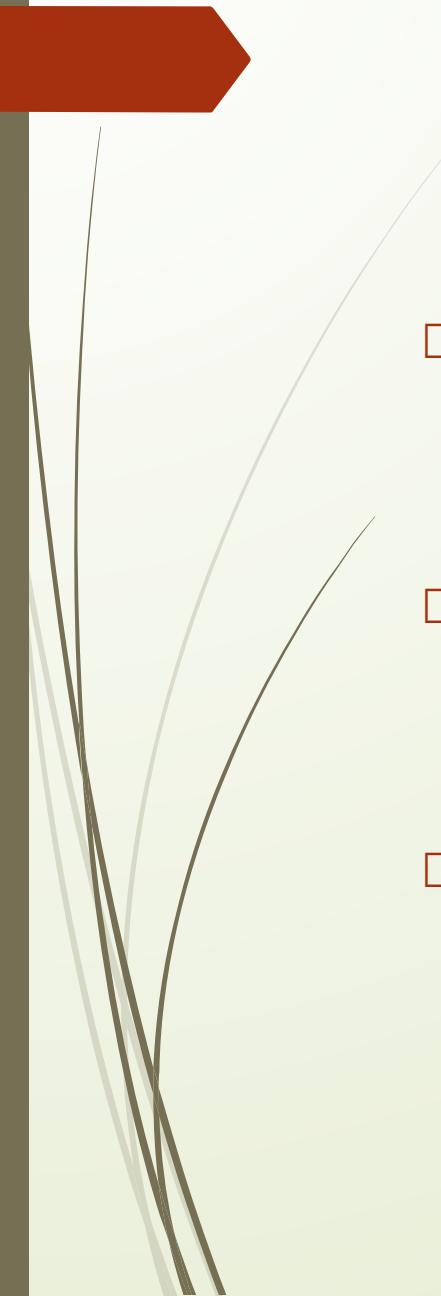
# Thread v/s process

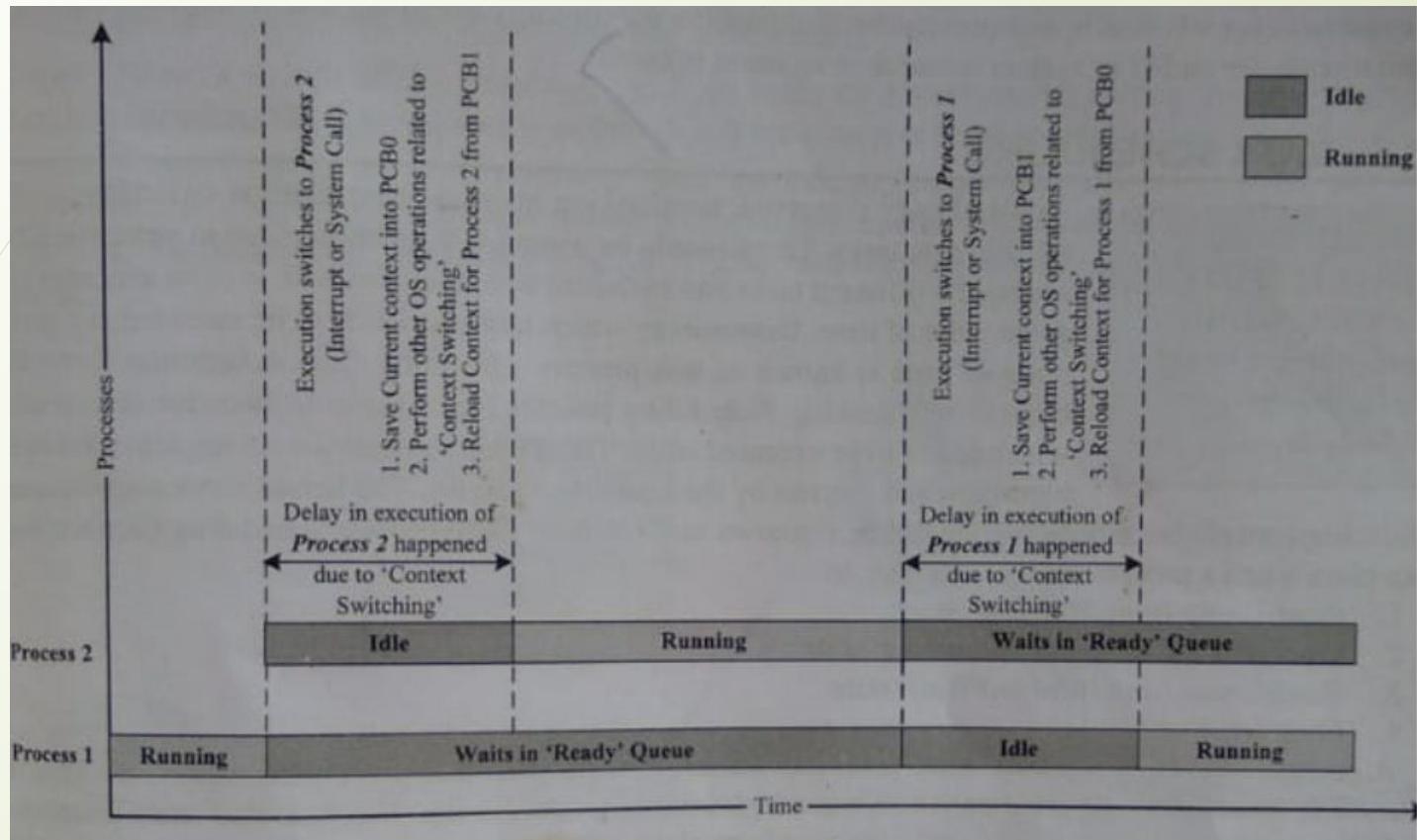
| Thread  | Process   |
|---|---|
| Thread is a single unit of execution and is part of process   | Process is a program in execution and contains one or more threads  |
| A thread shares the data memory and heap memory with other threads of the same process.   | Process has its own code memory, data memory and stack memory.  |
| A thread has no existence of its own, but lives within the process.   | Process contains at least one thread.   |
| There can be multiple threads in a process. The first thread calls the main function and occupies the start of the stack memory of the process. | Threads within a process share the data, code and heap memory. Each thread holds separate memory area of the stack. |
| Threads are inexpensive   | Expensive. Involves many OS overhead  |
| Context switching is  | Context switching is expensive  |



# Multi processing and multi tasking

- Multiprocessing
  - The ability to execute multiple process simultaneously
  - Systems which are capable of doing multiple process simultaneously is called multiprocessing systems
  - Such systems possesses multiple CPUs.
- Multiprogramming
  - The ability of OS to have multiple programs in memory, which are ready for execution is termed as multi programming
- Multi tasking
  - The ability of an OS to hold multiple processes in memory and switch the processor from executing one process to another process is called multitasking.
  - Multi tasking involves context switching, context saving and context retrieval

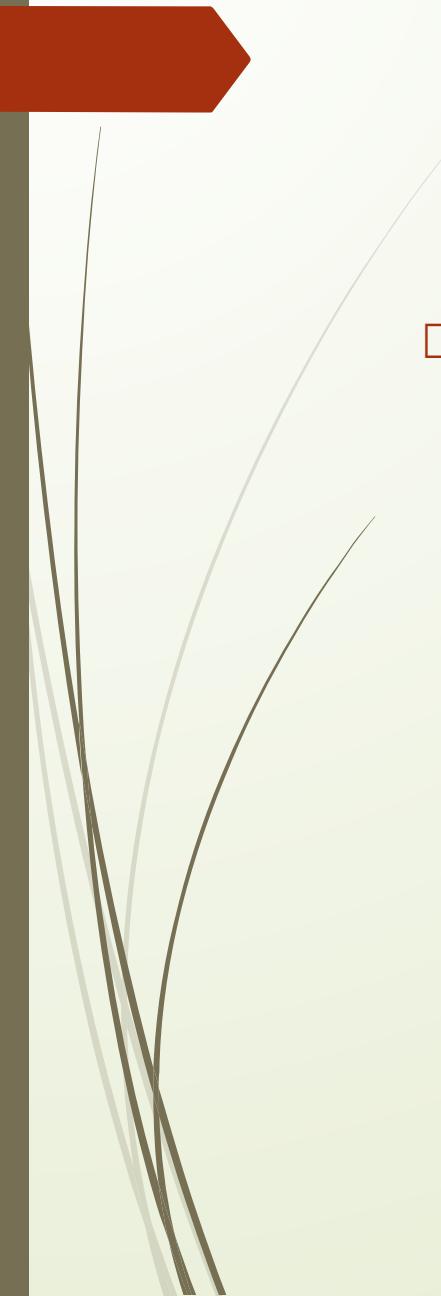
- 
- Context switching
    - The act of switching CPU among the process or changing the current execution context is known as context switching.
  - Context saving
    - The act of saving the current context which contains the context details for the current running process at the time of CPU switching is known as context saving
  - Context retrieval
    - The process of retrieving the saved context details for a process which is going to be executed due to CPU switching is called as context retrieval.



# Types of multitasking

## □ Co-operative multitasking

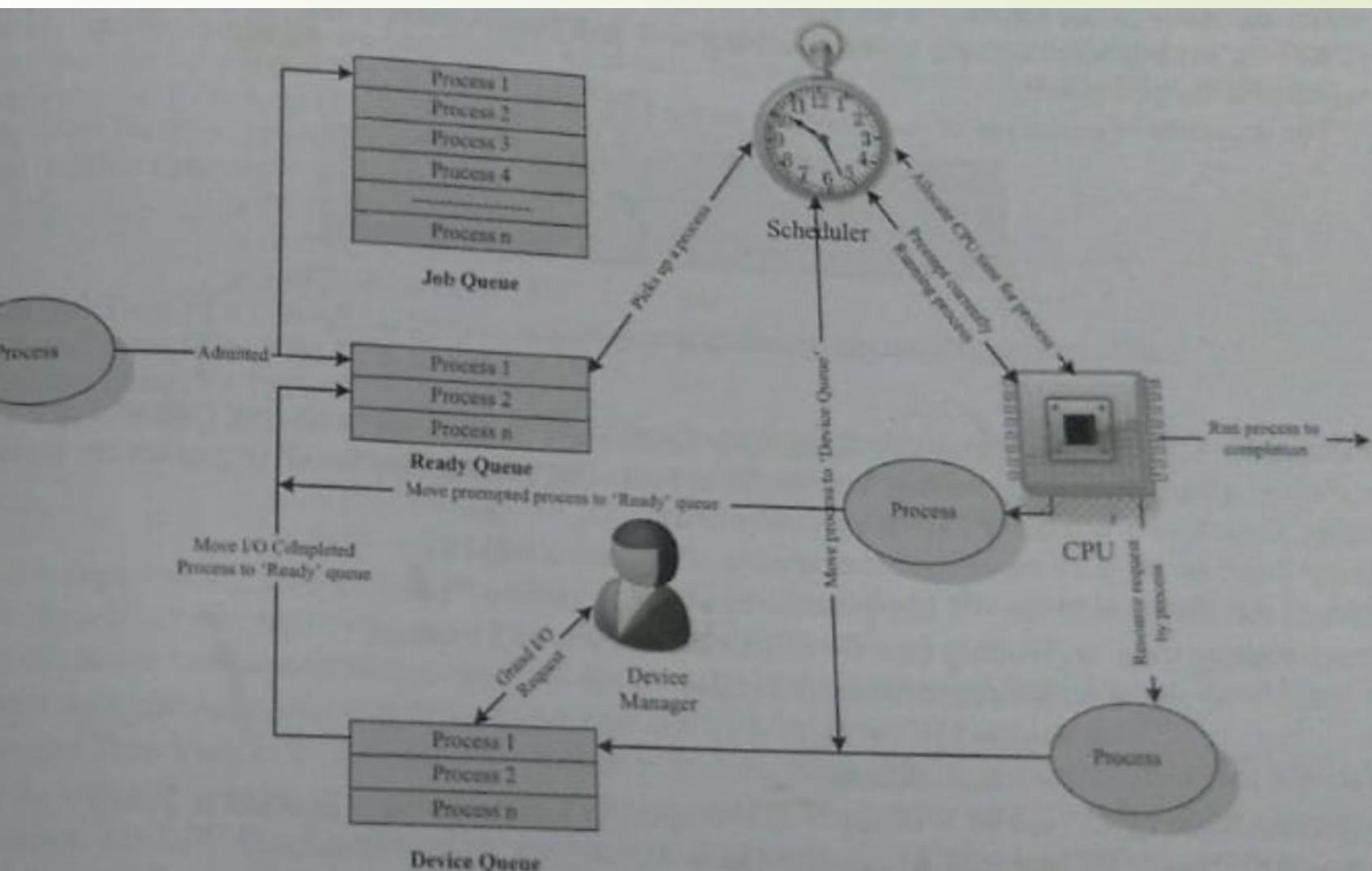
- In this type of multi tasking a task/process get a chance to execute only when the currently executing task/process voluntarily relinquishes the CPU.
- In this method any task/process can hold the CPU as much as time it wants.
- Since this type of implementation involves the mercy of the task each other for getting the CPU, it is known as cooperative multi tasking.
- If the currently executing task is non cooperative, the other task may wait for a long time.

- 
- Pre-emptive multitasking
    - It ensures that every process gets a chance to execute
    - When and how much time a process gets depends on the implementation of the pre-emptive scheduling
    - The currently running process is pre-empted to give chance to some other process.
    - The pre-emption is based on time slot or priority

- 
- Non pre-emptive multitasking
    - In this method the task/process which is currently allocated to the CPU is allowed to execute to its completion or enters into a state of Blocked/wait, waiting for system resource or I/O

# Task scheduling

- Determining which process should execute in the CPU is called the task/process scheduling.
- The kernel program which implements the scheduling algorithm is called scheduler
- The process scheduling decision may take place when a process switches its state to,
  - Ready to running
  - blocked./wait to running
  - Ready to blocked
  - Completed



# Criterion for selecting scheduling algorithm

## CPU utilization

- Is a measure of how much percentage of CPU being utilized. Prefer to be high
- Throughput
  - It is the number of process executed per unit time. Prefer to have high value
- Turnaround time
  - Time taken by a process to complete its execution.
  - Includes time spent by the process waiting for main memory, time spent in ready queue, time spent for completing the I/O operation, and the time spent for execution.
  - Prefer to have minimal value.
- Waiting time
  - It is the time spent by a process waiting in the ready queue to obtain CPU for execution.
  - Prefer to have minimal value
- Response time
  - Time elapsed between submission of a process and its first response.

# Various queues

- Job queue
  - Contains all processes in the system
- Ready queue
  - Contains all processes which are ready for execution and waiting for CPU to get their turn for execution
- Device queue
  - Contains set of processes which are waiting for an I/O device

# Scheduling algorithms

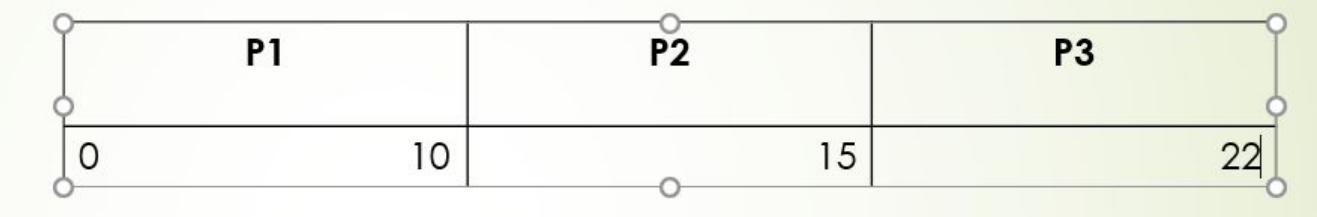
- Pre-emptive
- Non pre-emptive

# Non pre-emptive

- First come first served (FCFC/FIFO)
- Last Come First Served (LCFS/LIFO)
- Shortest Job First (SJF)
- Priority based

# First come first served (FCFC/FIFO)

- This algorithm allocates CPU time to the process based on the order in which they enters the ready queue.
- First entered process is serviced first
- Example
  - Three processes enters in the ready queue in the order P1,P2,P3. their estimated completion time is 10,5,7 milli seconds. Calculate the waiting time and Turn around



# Solution: average waiting time

- Assuming that CPU is readily available at the time of arrival of P1.
- Waiting time of P1=0
- Waiting time of P2=10
- Waiting time of P3=15
- Average waiting time= $(0+10+15)/3=8.33\text{ms}$

# Solution: Average turn around time

- TAT for P1=10
- TAT for P2=15
- TAT for P3=22
- Average Turn Around Time= $(10+15+22)/3=15.66$
- Execution time of P1=10
- Execution time of P2=5
- Execution time of P3=7
- Average Execution time = $(10+5+7)/3=7.33$
- Average turn around time = Average turn around time + Average Execution time = $8.33+7.33=15.66$

# Last Come First Served (LCFS/LIFO)

- This algorithm allocates CPU time to the process based on the order in which they enters the ready queue.
- First entered process is serviced first
- Example
  - Three processes enters in the ready queue in the order P1,P2,P3. their estimated completion time is 10,5,7 milli seconds. Calculate the waiting time and Turn around time. When P1 is executing, after 5ms, P4 with 6ms entered in to the ready queue.

| P1 | P4 | P3 | P2 |
|----|----|----|----|
| 0  | 10 | 16 | 23 |

# Solution: average waiting time

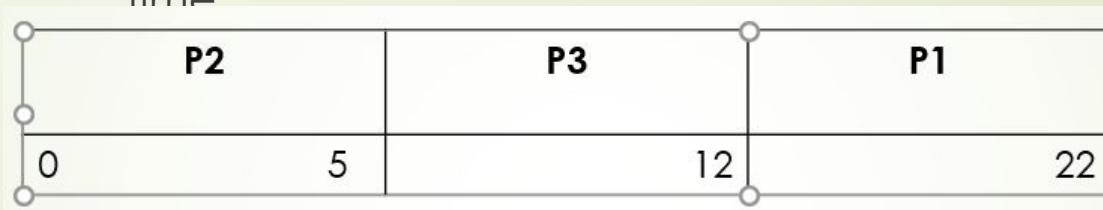
- Assuming that CPU is readily available at the time of arrival of P1.
- Waiting time of P1=0
- Waiting time of P4=10
- Waiting time of P3=16
- Waiting time of P2=23
- Average waiting time= $(0+10+16+23)/4=11\text{ms}$

# Solution: Average turn around time

- TAT for P1=10
- TAT for P4=16-5(After 5ms, arrived)
- TAT for P3=23
- TAT for P2=28
- Average Turn Around Time= $(10+11+23+28)/4=18$
- Execution time of P1=10
- Execution time of P2=5
- Execution time of P3=7
- Average Execution time = $(10+5+7)/3=7.33$

# Shortest Job First (SJF) Scheduling

- It always sorts the queue based on the execution time, whenever the process relinquishes the CPU.
- It picks the job with shortest execution time
- Example
  - Three processes enter in the ready queue in the order P1, P2, P3. Their estimated completion time is 10, 5, 7 milliseconds. Calculate the waiting time and Turn around time.



# Solution: average waiting time

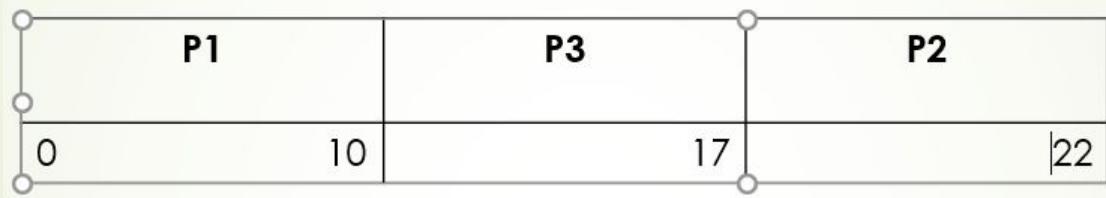
- Assuming that CPU is readily available at the time of arrival of P1.
- Waiting time of P2=0
- Waiting time of P3=5
- Waiting time of P1=12
- Average waiting time= $(0+5+12)/3=5.66\text{ms}$

# Solution: Average turn around time

- TAT for P2=5
- TAT for P3=12
- TAT for P2=22
- Average Turn Around Time= $(5+12+22)/3=13$
- Execution time of P2=5
- Execution time of P3=7
- Execution time of P1=10
- Average Execution time = $(10+5+7)/3=7.33=13\text{ms}$

# Priority Based Scheduling

- A process with highest priority will be scheduled first
- Priority can be specified as a number between 0 and max
- Max is OS dependant
- Example
  - Three processes enters in the ready queue in the order P1,P2,P3. their estimated completion time is 10,5,7 milli seconds and priority 0,3,2 respectively. Calculate the



# Solution: average waiting time

- Assuming that CPU is readily available at the time of arrival of P1.
- Waiting time of P2=0
- Waiting time of P3=10
- Waiting time of P1=17
- Average waiting time= $(0+10+17)/3=9\text{ms}$

# Solution: Average turn around time

- TAT for P2=10
- TAT for P3=17
- TAT for P2=22
- Average Turn Around Time= $(10+17+22)/3=16.33$

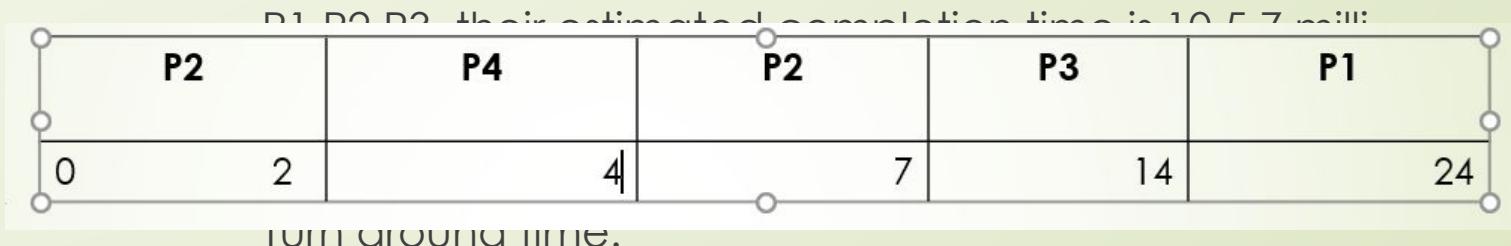
# Pre-emptive scheduling

- Pre-emptive SJF scheduling/ Shortest Remaining Time (SRT)
- Round robin scheduling
- Priority based scheduling

# Pre-emptive SJF scheduling/ Shortest Remaining Time (SRT)

- This algorithm sorts the ready queue when new process enters the ready queue and checks whether the execution time of new process is shorter than the remaining total estimated time for the currently executing process.
- If the execution time of the new process is less, the currently executing process is pre-empted and the new process is scheduled for execution.
- Example

- Three processes enters in the ready queue in the order



# Solution: average waiting time

- Assuming that CPU is readily available at the time of arrival of P1.
- Waiting time of P2=2
- Waiting time of P4=0
- Waiting time of P3=7
- Waiting time of P1=17
- Average waiting time= $(0+2+7+17)/4=5.75\text{ms}$

# Solution: Average turn around time

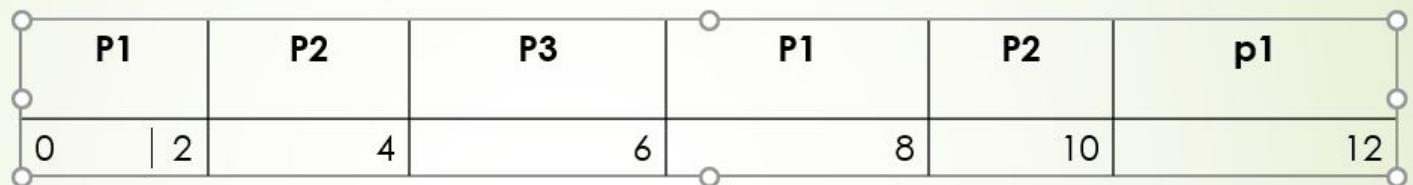
- TAT for P2=7
- TAT for P3=14
- TAT for P4=2
- TAT for P1=24
- Average Turn Around Time= $(7+14+2+24)/4=11.75$

# Round Robin Scheduling

- Follows the concept of “Equal chance to all”
- Each process in the ready queue is executed for a predefined period of time
- When this time period elapses, it will be moved to the end of the ready queue if execution is not completed.
- Next process in the ready queue is scheduled for execution
- If the process completes before the elapse of time slice, it voluntarily relinquishes CPU and next process takes charge of CPU.
- Similar to FCFS, except in case of time slicing.
- This process repeated.
- Time slice is managed by the timer tick feature of the time management part of OS

# Example

- Three processes enters in the ready queue in the order P1,P2,P3. their estimated completion time is 6,4,2 milli seconds. Calculate the waiting time and Turn around time. Time slice is 2ms.



# Solution: average waiting time

- Assuming that CPU is readily available at the time of arrival of P1.
- Waiting time of P1=0+(6-2)+(10-8)=6ms
- Waiting time of P=(2-0)+(8-4)=6ms
- Waiting time of P3=(4-0)=4ms
- Average waiting time=(6+6+4)/3=5.33ms



# Solution: Average turn around time

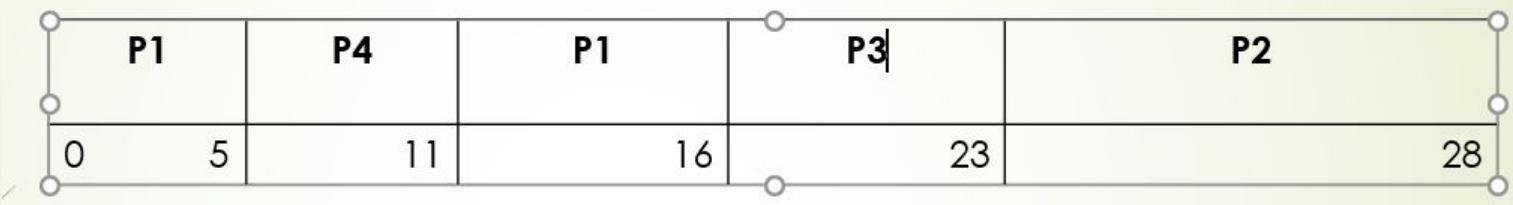
- TAT for P1=12ms
- TAT for P2=10
- TAT for P3=6ms
- Average Turn Around Time= $(12+10+6)/3=9.33$

# Priority based scheduling

- If any process with highest priority enters the ready queue, the executing process relinquishes the CPU and the highest priority process takes charge.
- Currently running process moves to the end of ready queue.
- It can process only when the high priority process completes its execution or voluntarily relinquishes the CPU

# Example

- Three processes enters in the ready queue in the order P1,P2,P3. Their estimated completion time is 10,5,7 milli seconds and priority is 1,3,2. All process enters the queue at the same time. After 5ms a new process P4 with execution time 6ms and priority 0 enters into the ready queue. Calculate the waiting time and Turn around time.



# Solution: average waiting time

- Assuming that CPU is readily available at the time of arrival of P1.
- Waiting time of P1=0+(11-5)=6ms
- Waiting time of P4=0ms
- Waiting time of P3=16ms
- Waiting time of P2=28ms
- Average waiting time=(6+0+16+28)/4=11.25ms

# Solution: Average turn around time

- TAT for P1=16ms
- TAT for P4=6ms
- TAT for P2=28ms
- TAT for P3=23ms
- Average Turn Around Time= $(16+6+23+28)/4=18.25$

# Real time behaviour

- Priority based scheduling gives Real Time attention to high priority task.
- This algorithm is used in system which demands real time behaviour
- Drawbacks
  - Starvation: continuously moving back in the ready queue
  - Aging can be used to overcome this



# How to chose an RTOS

- The factors to be considered while selecting RTOS may either be,
  - Functional
  - Non functional

# Functional requirements

- Processor support
  - Ensure processor support by RTOS
- Memory requirement
  - Certain OS files are stored in ROM and certain files are stored in RAM.
  - Embedded systems are memory constrained. Hence ensure minimal RAM and ROM
- Real time capabilities
  - All OS in Embedded system need not be RTOS.
  - Analyse the real time behaviour of OS under consideration and the standard met by the OS for real time capabilities.

- Kernel and interrupt latency
  - The kernel of the OS may disable the interrupt while executing certain services.
  - This causes interrupt latency
  - For ES whose response requirements are high, this latency should be minimal
- Inter process communication and task synchronization
  - It is OS kernel dependant
- Modularization support
  - It is very useful if the OS supports modularization where in which the developer can choose the essential modules and re-compile the OS image for functioning.

- 
- Support for networking and communication
    - Ensure that OS under consideration provide support for all interfaces required by the embedded product.
  - Development language support
    - The OS should contain built-in support for development language or check the help for third party vendor.

# Non-functional Requirement

- Custom Developed or off the shelf
  - Customize the OS if required.
  - The decision to select depends on development cost, licensing fee of OS, development time, availability of skilled resources.
- Cost
  - The total cost of developing or buying and maintaining should be considered before selection of OS
- Development and debugging tool availability
- Ease of Use
- After sales
  - After sale of OS, the support of bug fixes, critical patch updates, support of production issues etc. should be analysed.

# Module 6

Embedded Systems



# Topics

- Networks
  - Distributed Embedded Architectures
  - Networks for embedded systems
  - Network based design
  - Internet enabled systems
- Embedded Product Development Life Cycle
  - Description
  - Objectives
  - Phases
  - Approaches
- Recent Trends in Embedded Computing.



# Distributed Embedded Architectures

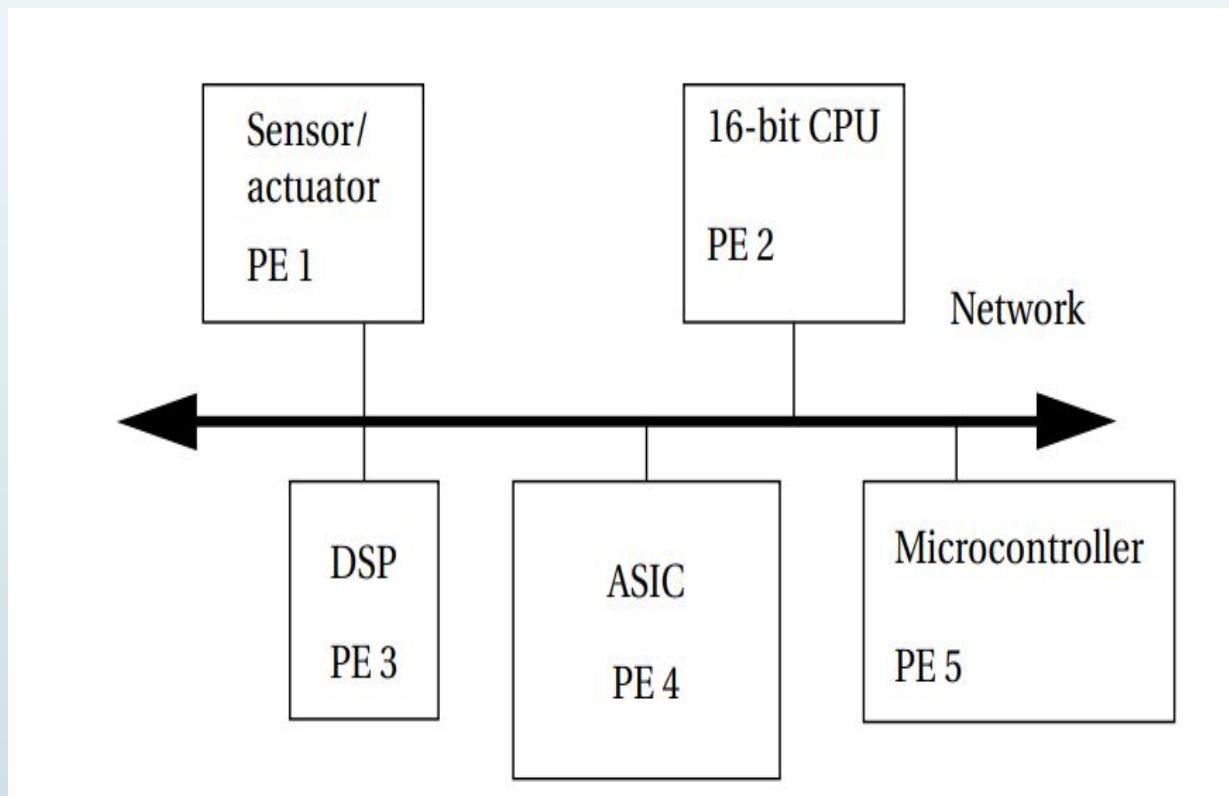
# Introduction

- In a distributed embedded system, several processing elements (PEs) are connected by a network that allows them to communicate.
- The application is distributed over the PEs, and some of the work is done at each node in the network.

# Reason for networked design

- In a distributed environment computing power also should be distributed.
- Data reduction
- Modularity
- Easy to debug

# DISTRIBUTED EMBEDDED ARCHITECTURES



- 
- The basic unit of distributed embedded system is PE
  - A PE may be an instruction set processor such as a DSP, CPU, or microcontroller, as well as a nonprogrammable unit such as the ASICs
  - Connection between PEs provided by the network as a communication link.
  - The system of PEs and networks forms the hardware platform on which the application runs.

# Why Distributed ?

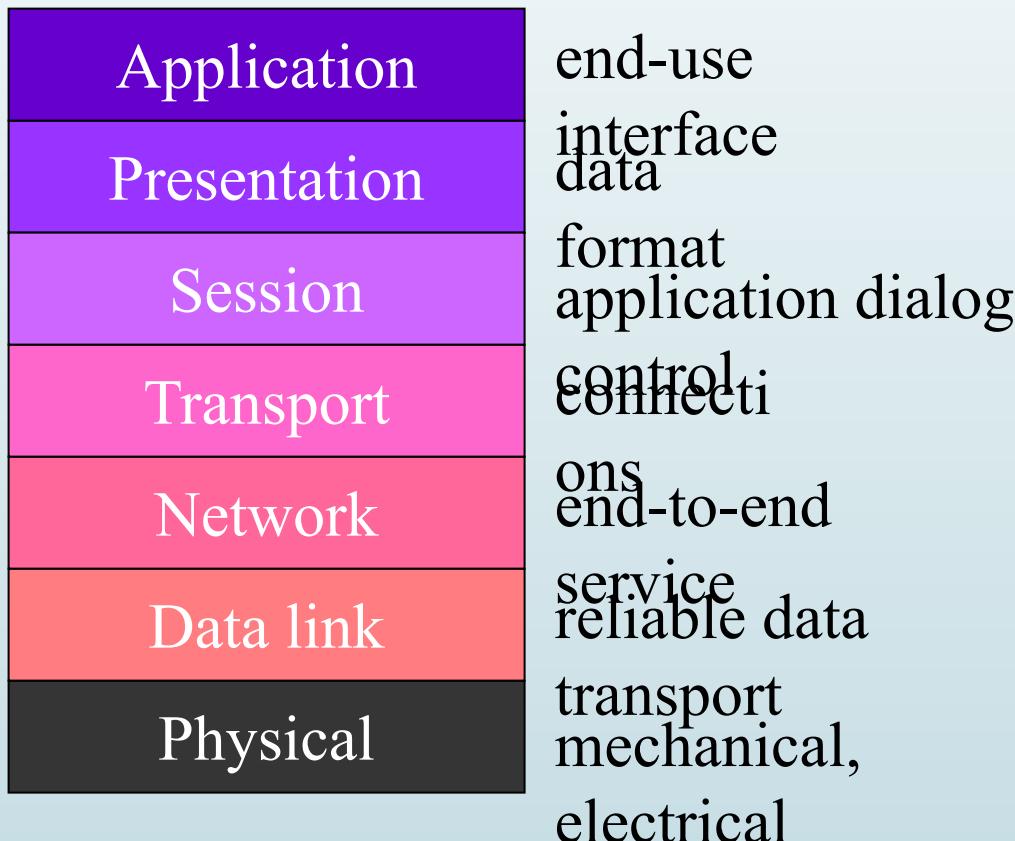
- Higher performance at lower cost.
- Physically distributed activities---time constants may not allow transmission to central site.
- Improved debugging---use one CPU in network to debug others.
- One system can be used to generate input and other one output

# Network abstractions

- International Standards Organization (ISO) developed the **Open Systems Interconnection (OSI)** model to describe networks:
  - 7-layer model.
  - Provides a standard way to classify network components and operations.

# Network abstractions

- Open Systems Interconnection (OSI) model





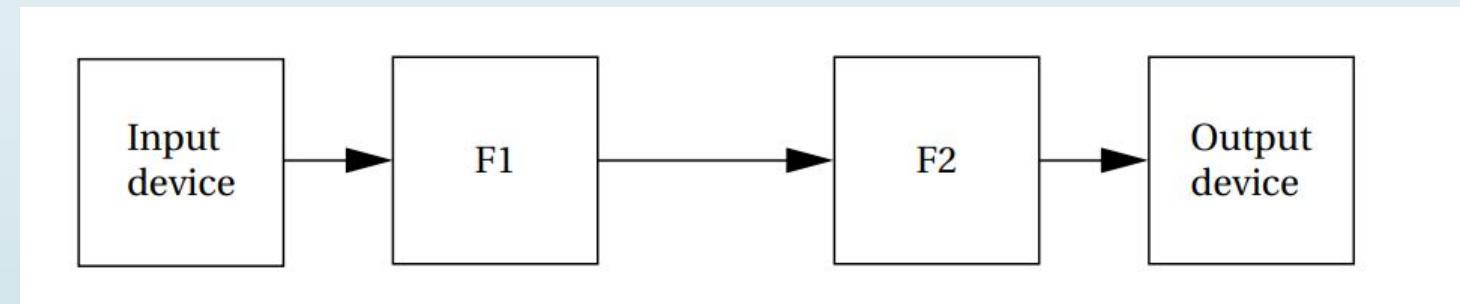
# Network Layers

- Physical: The physical layer defines the basic properties of the interface between systems, including the physical connections (plugs and wires) electrical properties, basic functions of the electrical and physical components, and the basic procedures for exchanging bits.
- Data link: The primary purpose of this layer is error detection and control across a single link. However, if the network requires multiple hops over several data links, the data link layer does not define the mechanism for data integrity between hops, but only within a single hop.
- Network: This layer defines the basic end-to-end data transmission service. The network layer is particularly important in multihop networks.
- Transport: The transport layer defines

- 
- Session: A session provides mechanisms for controlling the interaction of end user services across a network, such as data grouping and checkpointing.
  - Presentation: This layer defines data exchange formats and provides transformation utilities to application programs.
  - Application: The application layer provides the application interface between the network and end-user programs. Although it may seem that embedded systems wo

# Hardware and Software Architectures

- A point-to-point link establishes a connection between exactly two PEs.
- Point to-point links are simple to design precisely because they deal with only two components.



- The input signal is sampled by the input device and passed to the first digital filter, F1, over a point-to-point link.
- The results of that filter are sent through a second point-to-point link to filter F2.
- The results in turn are sent to the output device over a third point-to-point link.
- A digital filtering system requires that its outputs arrive at strict intervals, which means that the filters must process their inputs in a timely fashion.
- Using point-to-point connections allows both F1 and F2 to receive a new sample and send a new output at the same time without worrying about collisions on the communications network.

- It is possible to build a full-duplex, point-to-point connection that can be used for simultaneous communication in both directions between the two PEs.
- A bus is a more general form of network since it allows multiple devices to be connected to it.
- PEs connected to the bus have addresses.
- Communications on the bus generally take the form of

|        |         |      |                  |
|--------|---------|------|------------------|
| Header | Address | Data | Error correction |
|--------|---------|------|------------------|



Time

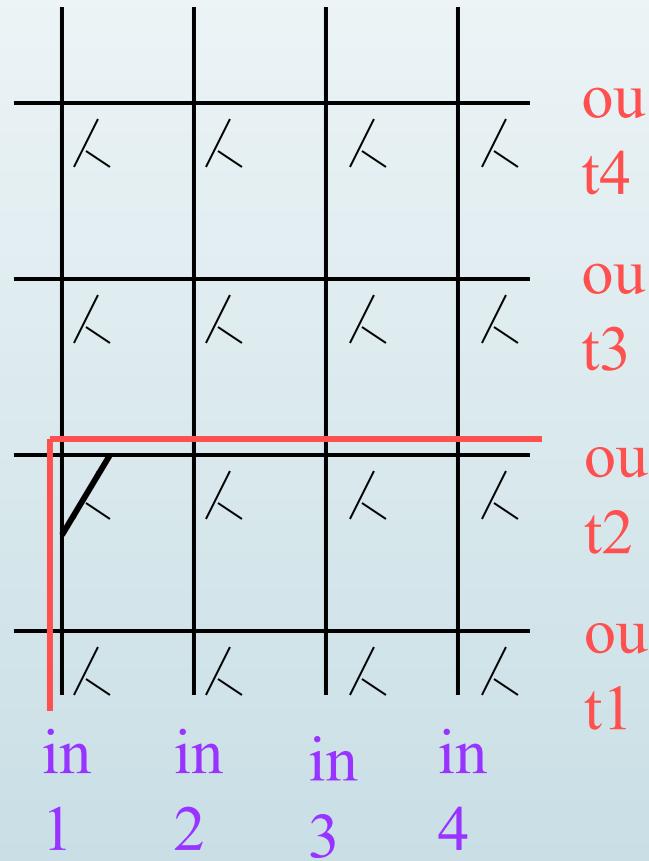
- A packet contains an address for the destination and the data to be delivered.
- It frequently includes error detection/correction information such as parity.
- It also may include bits that serve to signal to other PEs that the bus is in use, such as the header shown in the figure.
- The data to be transmitted from one PE to another may not fit exactly into the size of the data payload on the packet.
- It is the responsibility of the transmitting PE to divide its data into packets; the receiving PE must of course reassemble the complete data message from the packets.

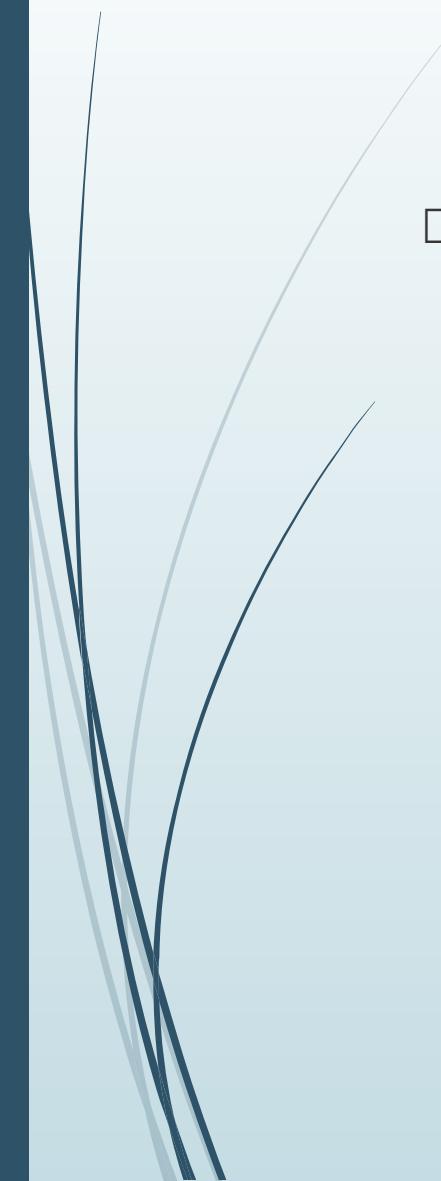
- Distributed system buses must be arbitrated to control simultaneous access, just as with microprocessor buses.
- Arbitration scheme types are summarized below.
  - Fixed-priority arbitration always gives priority to competing devices in the same way. If a high-priority and a low-priority device both have long data transmissions ready at the same time, it is quite possible that the low-priority device will not be able to transmit anything until the high-priority device has sent all its data packets.
  - Fair arbitration schemes make sure that no device is starved. Round-robin arbitration is the most commonly used of the fair arbitration schemes. The PCI bus requires that the arbitration scheme used on the bus must be fair, although it does not specify a particular arbitration scheme. Most implementations of PCI use round-robin arbitration.

- A bus has limited available bandwidth.
- Since all devices connect to the bus, communications can interfere with each other.
- Other network topologies can be used to reduce communication conflicts.
- At the opposite end of the generality spectrum from the bus is the crossbar network.
- A crossbar not only allows any input to be connected to any output, it also allows all combinations of input/output connections to be made.
- A crosspoint is a switch that connects an input to an output. To connect an input to an output, we activate the crosspoint at the intersection between the corresponding input and output lines in the crossbar

# Hardware and Software Architectures

Crossbar



- 
- Drawback of cross bar
    - The major drawback of the crossbar network is expense:  
The size of the network grows as the square of the number  
of inputs

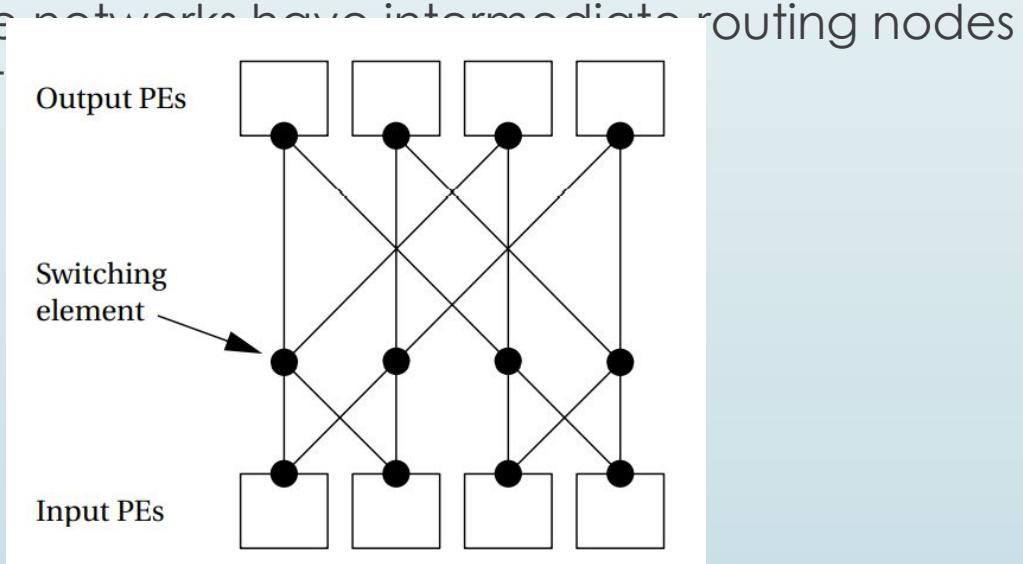
# Hardware and Software Architectures

## Message passing programming

- Transport layer provides message-based programming interface

```
send_msg(adrs, data);
```
- Data must be broken into packets at source, reassembled at destination.
- **Data-push programming:** Make things happen in network based on data transfers

- Many other networks have been designed that provide varying amounts of parallel communication at varying hardware costs. Figure shows an example multistage network.
- Multistage networks have intermediate routing nodes to guide traffic



- 
- networks differ from microprocessor buses in how they implement communication protocols.
  - Both need handshaking to ensure that PEs do not interfere with each other.

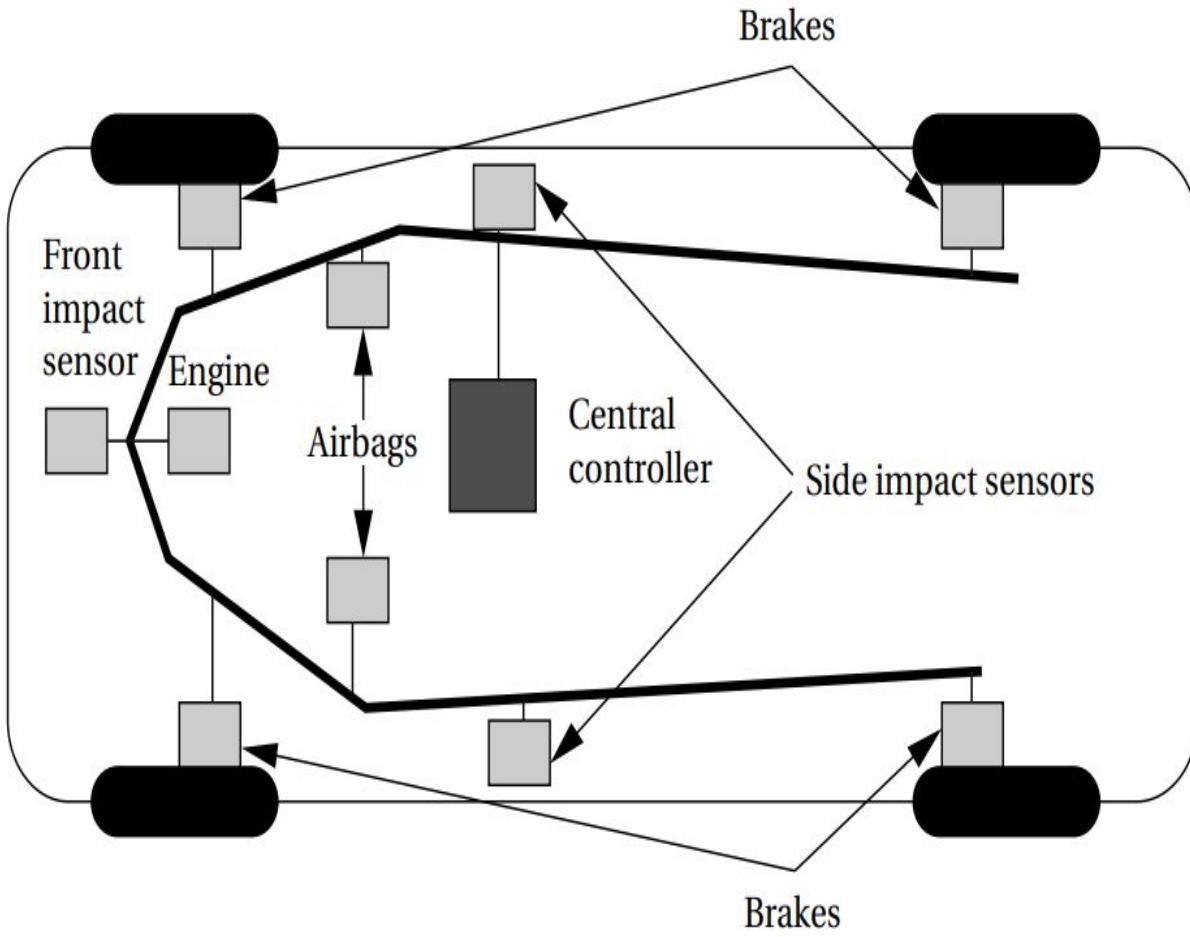
# Message Passing Programming

- Distributed embedded systems do not have shared memory, so they must communicate by passing messages
- A message must be broken up into packets to be sent on the network.
- A procedural interface for sending a packet might look like the following,
  - `send_packet(address,data);`
- The routine should return a value to indicate whether the message was sent successfully if the network
  - ```
for (i = 0; i < message.length; i = i + PACKET_SIZE)
    send_packet(address,&message.data[i]);
```
- must be broken up into packet-size data segments as follows

- The above code uses a loop to break up an arbitrary-length message into packet size chunk
- Reception of a packet will probably be implemented with interrupts.
- The simplest procedural interface will simply check to see whether a received message is waiting in a buffer.

- Communication may be blocking or non-blocking.
- The simplest implementation of message passing is blocking, with the routine not returning until it has transmitted or received.
- A non-blocking network interface requires a queue of data to be sent, with the network driver sending packets off the head of the queue and placing received packets on the tail of the queue.
- A non-blocking communication mechanism makes sense only when concurrency is available between computing and data transfer.
- Network protocols may encourage a data-push

- Network protocols may encourage a data-push design style for the system built around the network.
- In a single-CPU environment, a program typically initiates a read whenever it wants data.
- In many networked systems, nodes send values out without any request from the intended user of the system.
- Data-push programming makes sense for periodic data—if the data will always be used at regular intervals, we can reduce data traffic on the network by automatically sending it when it is needed.



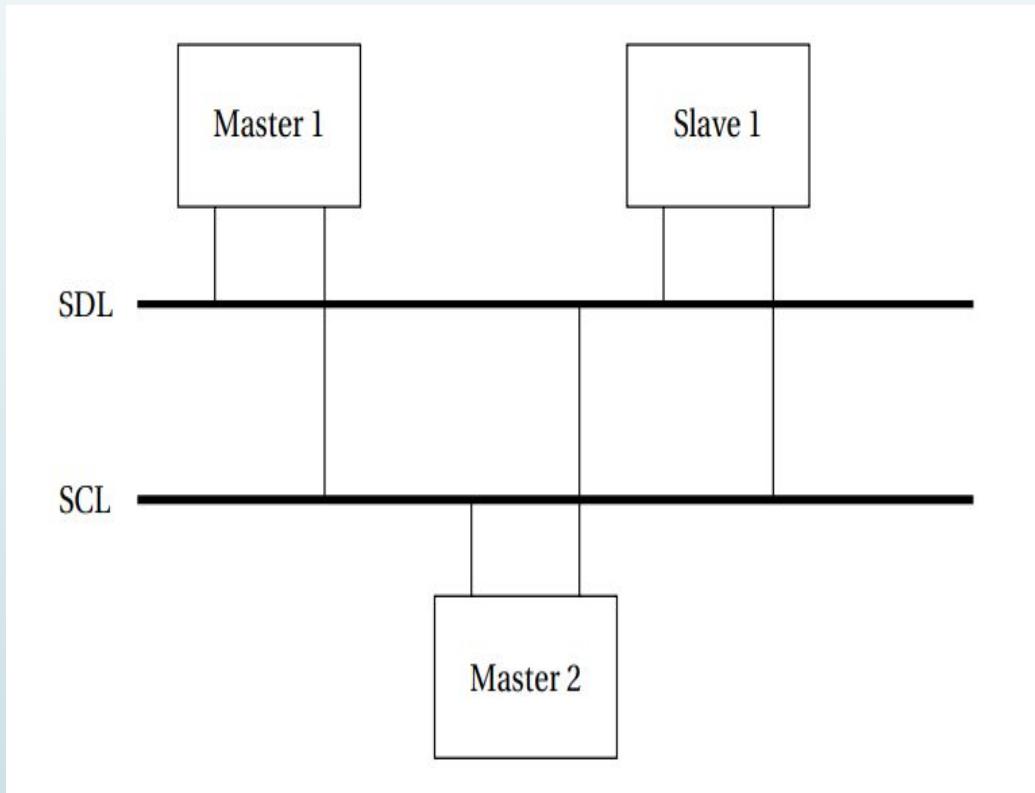
# Networks for Embedded Systems

- Several interconnect networks have been developed especially for distributed embedded computing:
  - I<sup>2</sup>C bus-used in microcontroller-based systems.
  - CAN (Controller Area Network) bus-developed for automotive electronics
  - Ethernet and variations of standard Ethernet - used for a variety of control applications.

# I2C Bus

- Used to link microcontrollers into systems.
- It has even been used for the command interface in an MPEG-2 video chip, while a separate bus was used for high-speed video data, setup information was transmitted to the on-chip controller through an I2C bus interface.
- Advantages
  - Low cost
  - Easy to implement
  - Moderate speed
- Uses two lines
  - The **serial data line** (SDL) for data
  - The **serial clock line** (SCL), which indicates when valid data are on the data line.

# Structure of an I2C bus system

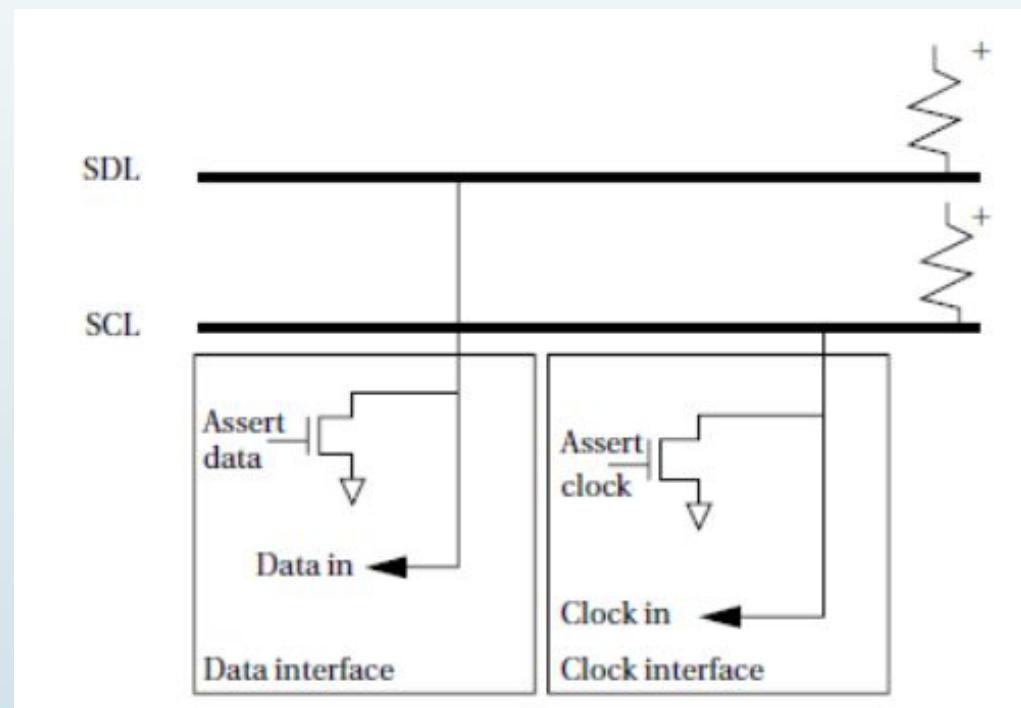




# Structure of an I2C bus system

- Every node in the network is connected to both SCL and SDL.
- Some nodes may be able to act as bus masters and the bus may have more than one master.
- Other nodes may act as slaves that only respond to requests from masters.

# I<sup>2</sup>C bus electrical interface



# I2C signalling

- The bus does not define particular voltages to be used for high or low so that either bipolar or MOS circuits can be connected to the bus.
- Both bus signals use open collector/open drain circuits.
- A pull-up resistor keeps the default state of the signal high, and transistors are used in each bus device to pull down the signal when a 0 is to be transmitted.
- Open collector/open drain signaling allows several devices to simultaneously write the bus without causing electrical damage.
- The open collector/open drain circuitry allows a slave device to stretch a clock signal during a read from a slave.
- The master is responsible for generating the SCL clock, but the slave can stretch the low period of the clock if necessary

# I2C bus system

- The I2C bus is designed as a multi master bus—any one of several different devices may act as the master at various times.
- As a result, there is no global master to generate the clock signal on SCL.
- Instead, a master drives both SCL and SDI when it is sending data.
- When the bus is idle, both SCL and SDI remain high.
- When two devices try to drive either SCL or SDI to different values, the open collector/ open drain circuitry prevents errors, but each master device must listen to the bus while transmitting to be sure that it is not interfering with another message
- If the device receives a different value than it is trying to transmit, then it knows that it is interfering with

# Addressing of I2C

- Every I2C device has an address.
- The addresses of the devices are determined by the system designer, usually as part of the program for the I2C driver.
- The addresses must of course be chosen so that no two devices in the system have the same address.
- A device address is 7 bits in the standard I2C definition.
- The address 0000000 is used to signal a general call or bus broadcast, which can be used to signal all devices simultaneously.
- The address 11110XX is reserved for the extended 10-bit addressing scheme; there are several other reserved addresses as well.

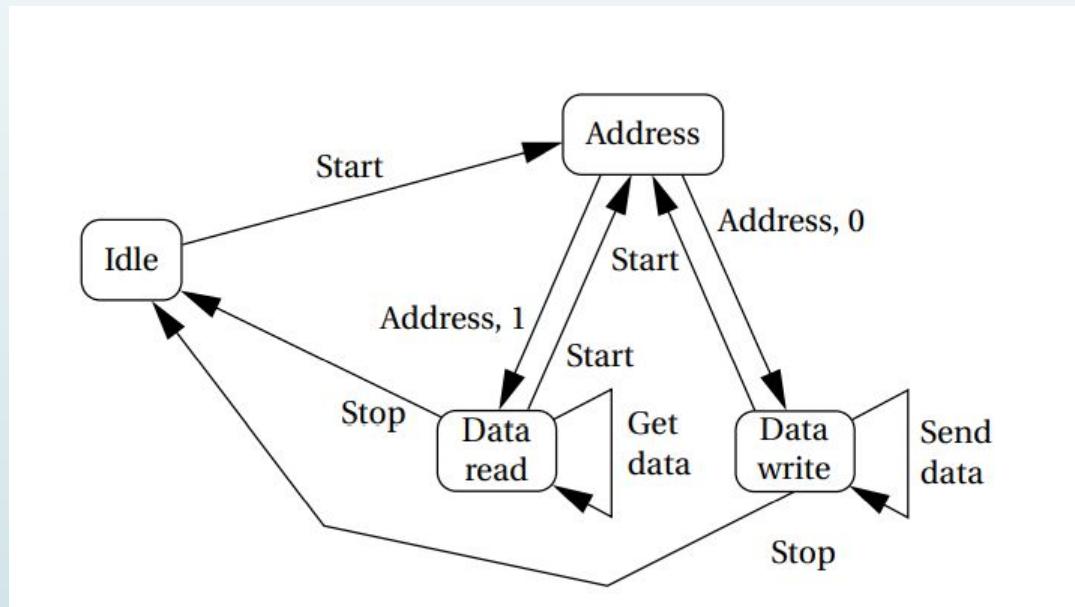


# I2C bus transaction

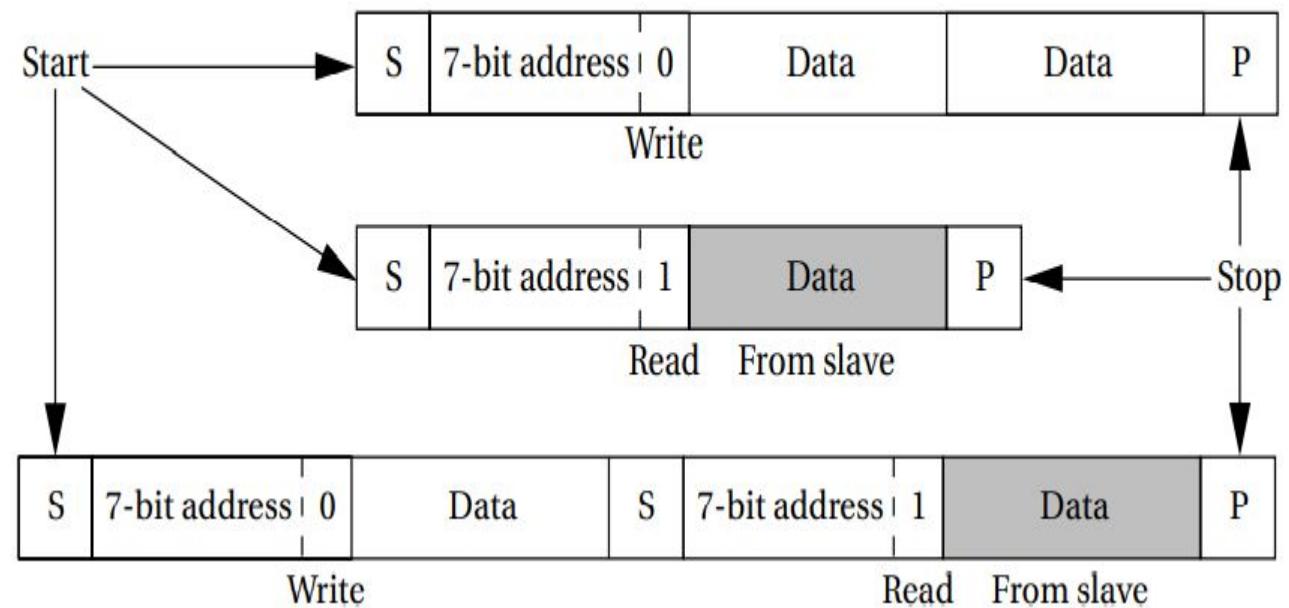
- A bus transaction comprised a series of 1-byte transmissions and an address followed by one or more data bytes.
- I2C encourages a data-push programming style.
- When a master wants to write a slave, it transmits the slave's address followed by the data.
- Since a slave cannot initiate a transfer, the master must send a read request with the slave's address and let the slave transmit the data.
- Therefore, an address transmission includes the 7-bit address and 1 bit for data direction: 0 for writing from the master to the slave and 1 for reading from the slave to the master.

- A bus transaction is initiated by a start signal and completed with an end signal as follows:
  - A start is signaled by leaving the SCL high and sending a 1 to 0 transition on SDL.
  - A stop is signaled by setting the SCL high and sending a 0 to 1 transition on SDL

# State transition graph for an I2C bus master



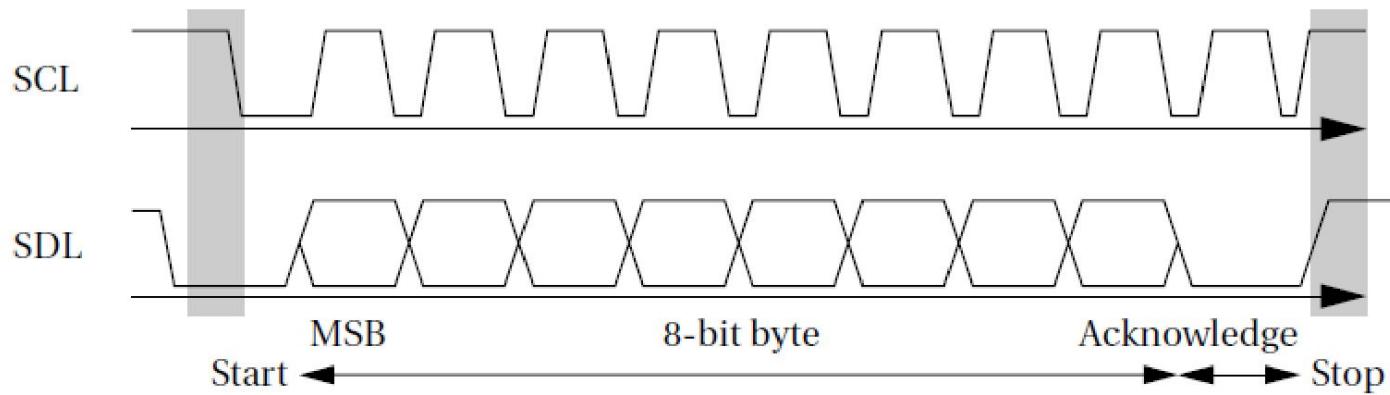
# Typical bus transactions on the I2C bus



- In the first example, the master writes 2 bytes to the addressed slave.
- In the second, the master requests a read from a slave.
- In the third, the master writes 1 byte to the slave, and then sends another start to initiate a read from the slave.

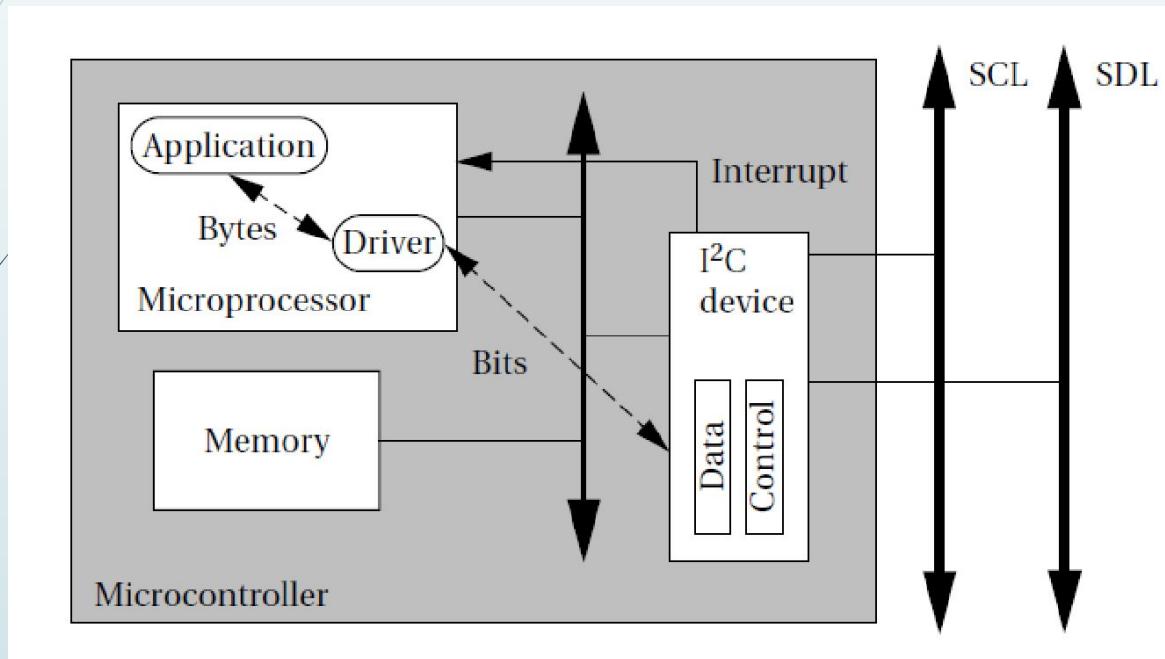
# Transmitting a byte on the I<sup>2</sup>C bus.

Typical bus transactions on the I<sup>2</sup>C bus.



- The transmission starts when SCL is pulled low while SDA remains high.
- After this start condition, the clock line is pulled low to initiate the data transfer.
- At each bit, the clock line goes high while the data line assumes its proper value of 0 or 1.
- An acknowledgment is sent at the end of every 8-bit transmission, whether it is an address or data.
- For acknowledgment, the transmitter does not pull down the SDA, allowing the receiver to set the SDA to 0 if it properly received the byte.
- After acknowledgment, the SDA goes from low to high while the SCL is high, signaling the stop condition.

# An I<sup>2</sup>C interface in a microcontroller

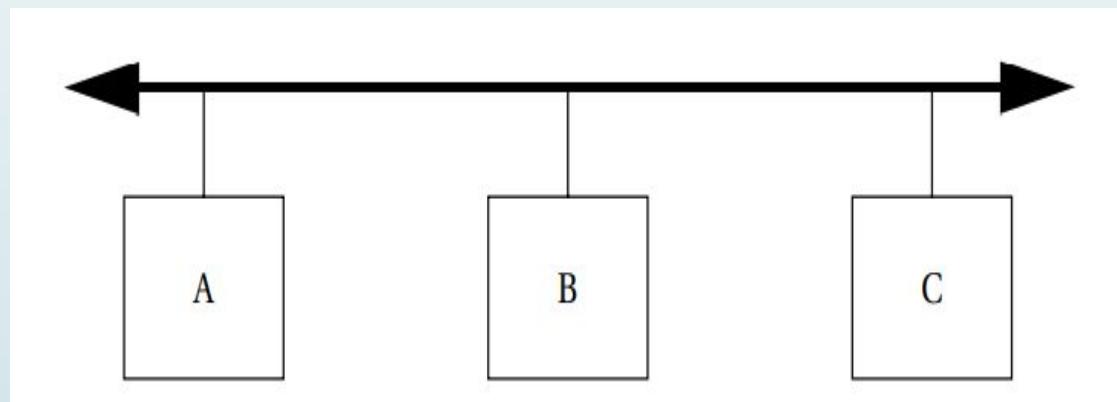


- A typical system has a 1-bit hardware interface with routines for byte level functions.
- The I2C device takes care of generating the clock and data.
- The application code calls routines to send an address, send a data byte, and so on, which then generates the SCL and SDL, acknowledges, and so forth.
- One of the microcontroller's timers is typically used to control the length of bits on the bus.
- Interrupts may be used to recognize bits.
- However, when used in master mode, polled I/O may be acceptable if no other pending tasks can be performed, since masters initiate their own transfers.

# Ethernet

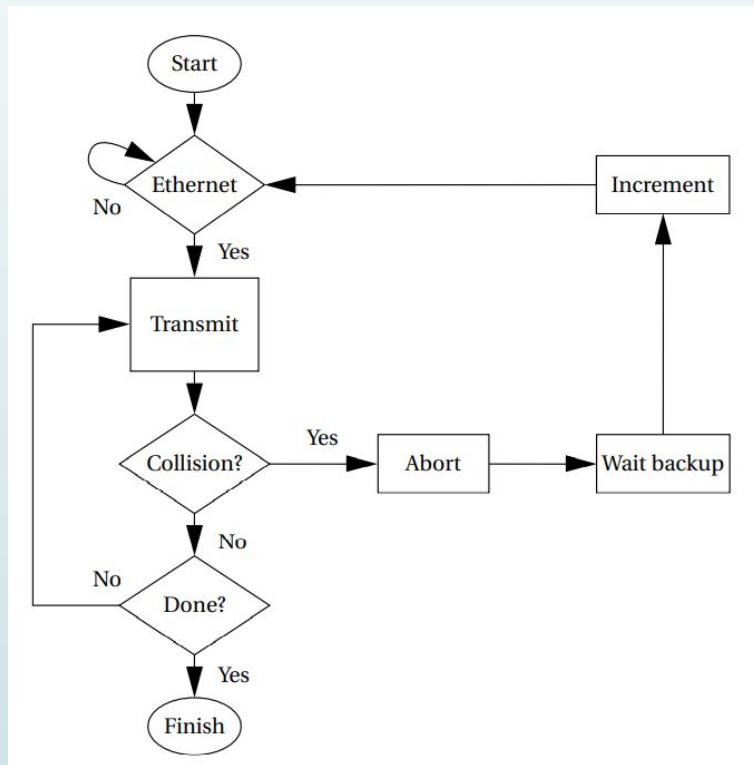
- Ethernet is very widely used as a local area network for general-purpose computing.
- Ethernet is particularly useful when PCs are used as platforms, making it possible to use standard components, and when the network does not have to meet rigorous real-time requirements.
- The network is a bus with a single signal path; the Ethernet standard allows for several different implementations such as twisted pair and coaxial cable.

# Ethernet organization

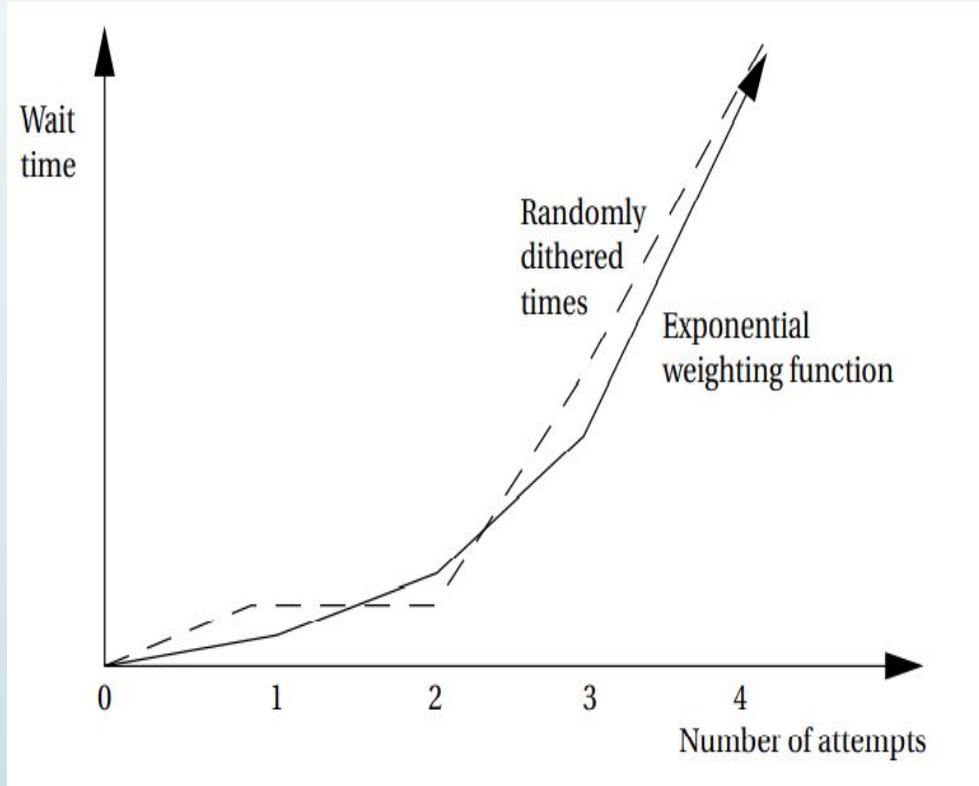


- Unlike the I2C bus, nodes on the Ethernet are not synchronized—they can send their bits at any time.
- Since Ethernet nodes are not synchronized, if two nodes decide to transmit at the same time, the message will be ruined.
- The Ethernet arbitration scheme is known as Carrier Sense Multiple Access with Collision Detection (CSMA/CD).
- A node that has a message waits for the bus to become silent and then starts transmitting.
- It simultaneously listens, and if it hears another transmission that interferes with its transmission, it stops transmitting and waits to retransmit.
- The waiting time is random, but weighted by an exponential function of the number of times the message has been aborted.
- Figure 8.16 shows the exponential backoff function both before and after it is modulated by the random wait time. Since a message may be interfered with several times before it is successfully transmitted, the exponential backoff technique helps to ensure that the network does not become overloaded at high demand factors. The random factor in the wait time minimizes the chance that two messages will repeatedly interfere with each other.

# CSMA/CD algorithm



# Exponential backoff times.



# Format of Ethernet packet

- It provides addresses of both the destination and the source.
- It also provides for a variable-length data payload.
- Ethernet was not designed to support real-time operations; the exponential backoff scheme cannot

|          |             |                     |                |        |      |         |     |
|----------|-------------|---------------------|----------------|--------|------|---------|-----|
| Preamble | Start frame | Destination address | Source address | Length | Data | Padding | CRC |
|----------|-------------|---------------------|----------------|--------|------|---------|-----|

# Fieldbus

- Manufacturing systems require networked sensors and actuators.
- Fieldbus is a set of standards for industrial control and instrumentation systems.
- The H1 standard uses a twisted-pair physical layer that runs at 31.25 MB/s.
- It is designed for device integration and process control.
- The High Speed Ethernet standard is used for backbone networks in industrial plants.
- It is based on the 100 MB/s Ethernet standard. It can integrate devices and subsystems.

# Network Based Design

- Designing a distributed embedded system around a network schedule computations in time and allocate them to PEs.
- Many embedded networks are designed for low cost and therefore do not provide excessively high communication speed.
- If we are not careful, the network can become the bottleneck in system design.

# Network Based Design

- The **message delay** for a single message with no contention (as would be the case in a point-to-point connection) can be modeled as

$$t_m = t_x + t_n + t_r$$

- $t_m$  – message delay
- $t_x, t_r$  – transmitter and receiver overhead
- $t_n$  – network transmission time

# Network Based Design

## **Simple message delay for an I<sup>2</sup>C message**

Let's assume that our I<sup>2</sup>C bus runs at the rate of 100 KB/s and that we need to send one 8-bit byte. Based on the message format shown in Figure 8.9, we can compute the number of bits in the complete packet:

$$\begin{aligned}n_{\text{packet}} &= \text{startbit} + \text{address} + \text{data} + \text{stopbit} \\&= 1 + 8 + 8 + 1 = 18 \text{ bits}\end{aligned}$$

The time required, then, to transmit the packet is

$$t_n = n_{\text{packet}} \times t_{\text{bit}} = 1.8 \times 10^{-4} \text{ s.}$$

Some of the instructions in the transmitter and receiver drivers—namely, the loops that send bytes to and receive bytes from the network interface—will run concurrently with the message transmission. If we assume that 20 instructions outside of these loops are executed by the transmitter and receiver, overheads on an 8 MHz microcontroller would be as follows:

$$t_x = t_r = 20 \times 0.125 \times 10^{-6} = 2.5 \times 10^{-6}.$$

The total message delay is:

$$t_m = 2.5 \times 10^{-6} + 1.8 \times 10^{-4} + 2.5 \times 10^{-6} = 1.85 \times 10^{-4}.$$

Overhead is <3% of the total message time in this case.

# Network Based Design

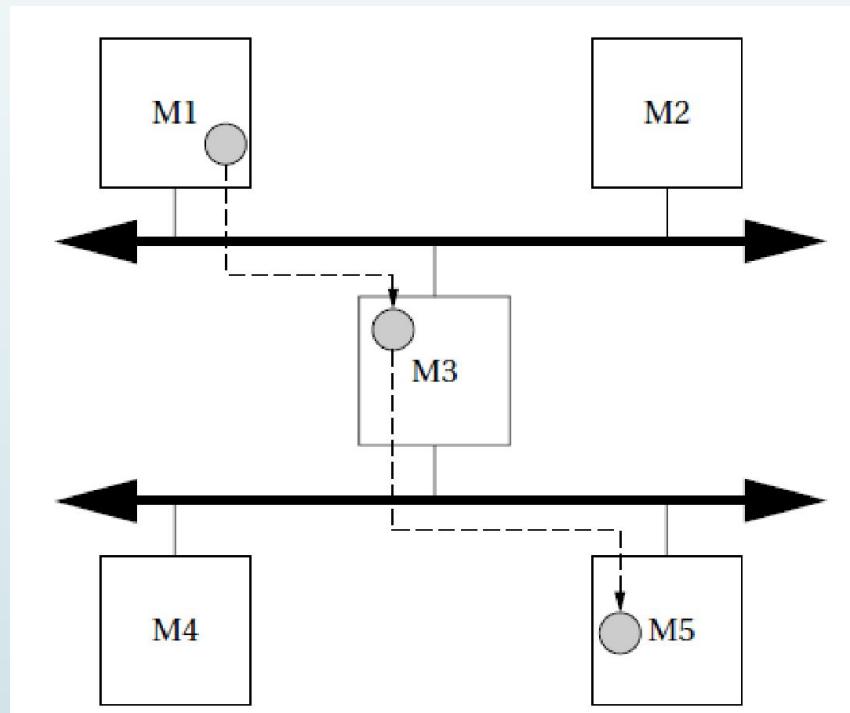
- Considering other interferences, message delay

$$t_y = t_d + t_m$$

- Availability delay based on scheduling
- Fixed priority arbitration
  - If the network uses fixed-priority arbitration, the network availability delay is unbounded for all but the highest-priority device. Since the highest-priority device always gets the network first, unless there is an application-specific limit on how long it will transmit before relinquishing the network, it can keep blocking the other devices indefinitely
- Fair arbitration
  - If the network uses fair arbitration, the network availability delay is bounded. In the case of round-robin arbitration, if there are N devices, then the worst case network availability delay is  $N(t_{tx} + t_{arb})$ , where  $t_{arb}$  is the delay incurred for arbitration.  $t_{arb}$  is usually small compared to transmission time.

# Multihop networks

Multi

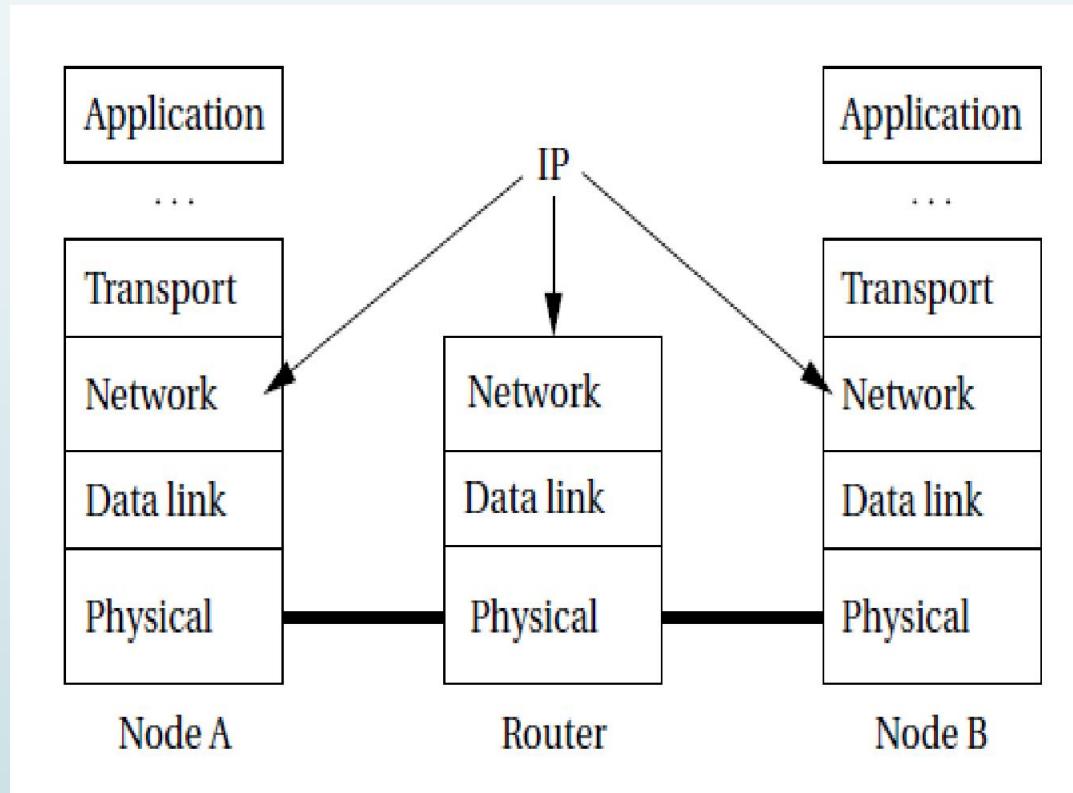


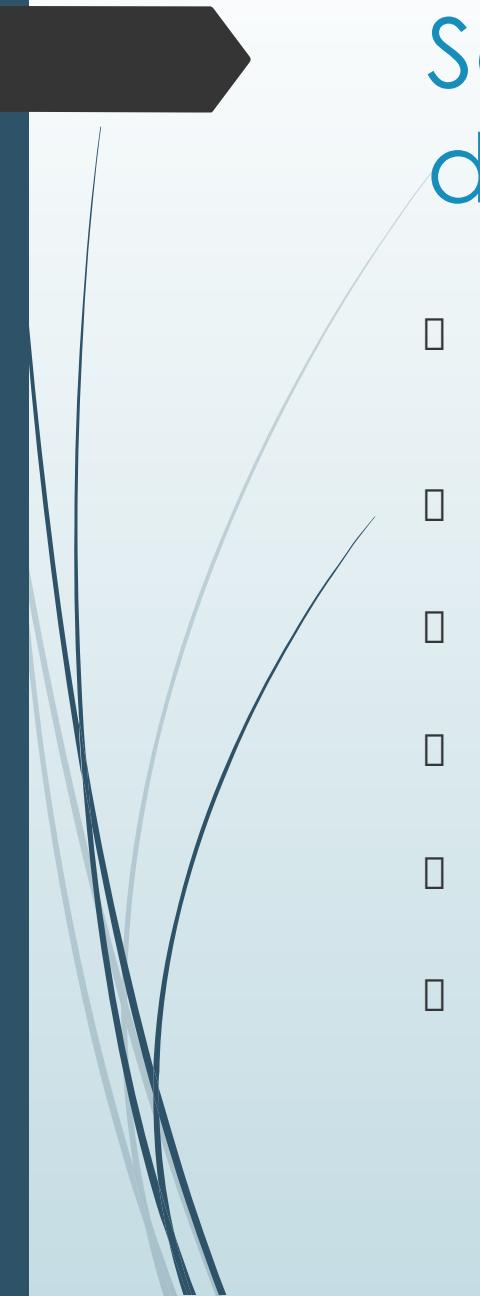
- It is possible to build multihop networks in which messages are routed through network nodes to get to their destinations.
- The hardware platform has two separate networks ( perhaps so that communications between subsets of the PEs do not interfere), but there is no direct path from M1 to M5.The message is therefore routed through M3, which reads it from one network and sends it on to the other one.
- Analyzing delay in multihop network is difficult

# INTERNET-ENABLED SYSTEMS

- The **Internet Protocol (IP)** is the fundamental protocol on the Internet.
- It provides connectionless, packet-based communication
- 32 bit IPV4 and 128 bit IPV6

# Protocol utilization in internet

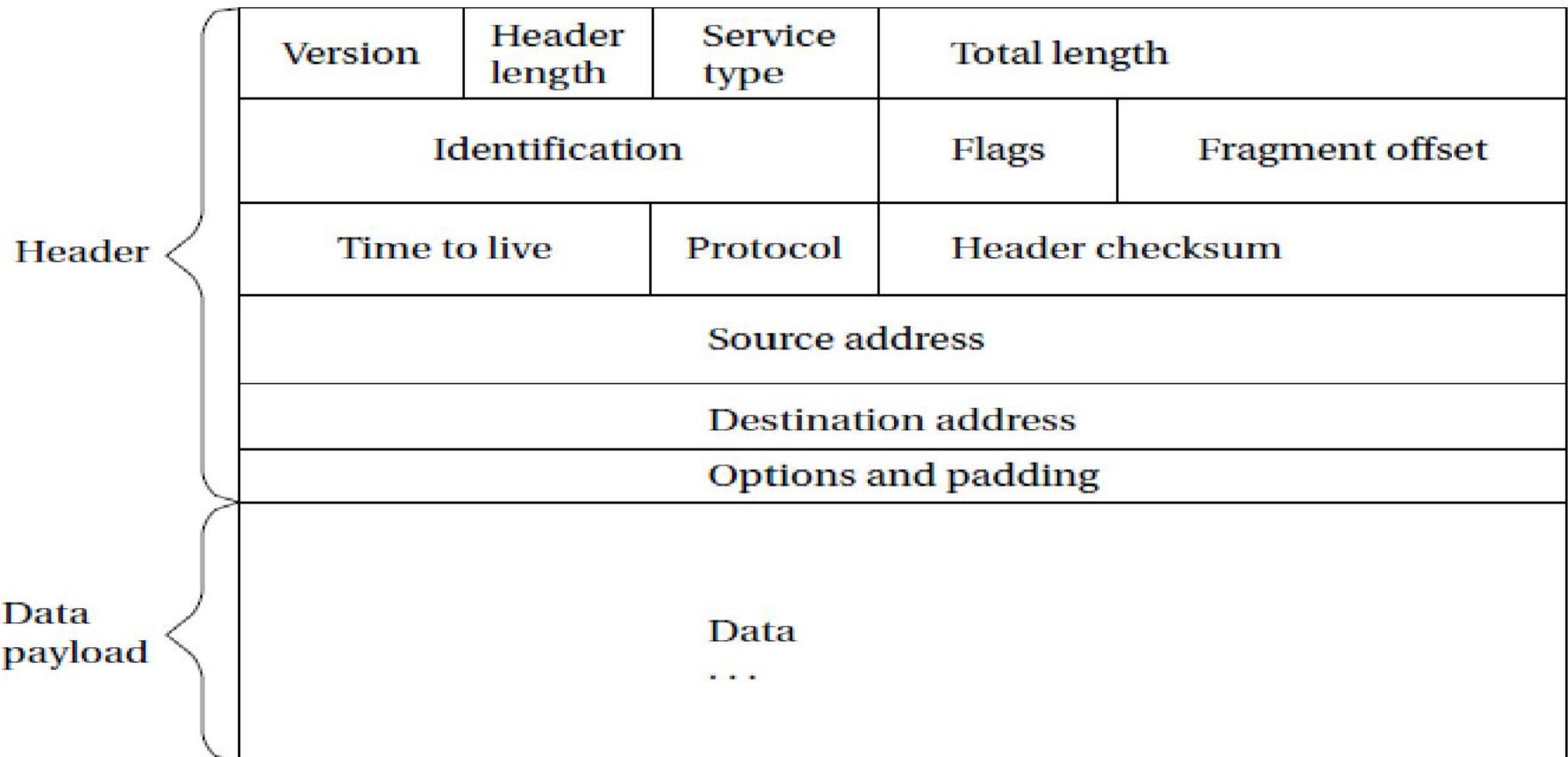




# Sending data from source to destination

- IP works at the network layer. When node A wants to send data to node B, the application's data pass through several layers of the protocol stack to send to the IP.
- IP creates packets for routing to the destination, which are then sent to the data link and physical layers.
- A node that transmits data among different types of networks is known as a router.
- In general, a packet may go through several routers to get to its destination.
- At the destination, the IP layer provides data to the transport layer and ultimately the receiving application.
- As the data pass through several layers of the protocol stack, the IP packet data are encapsulated in packet formats appropriate to each layer

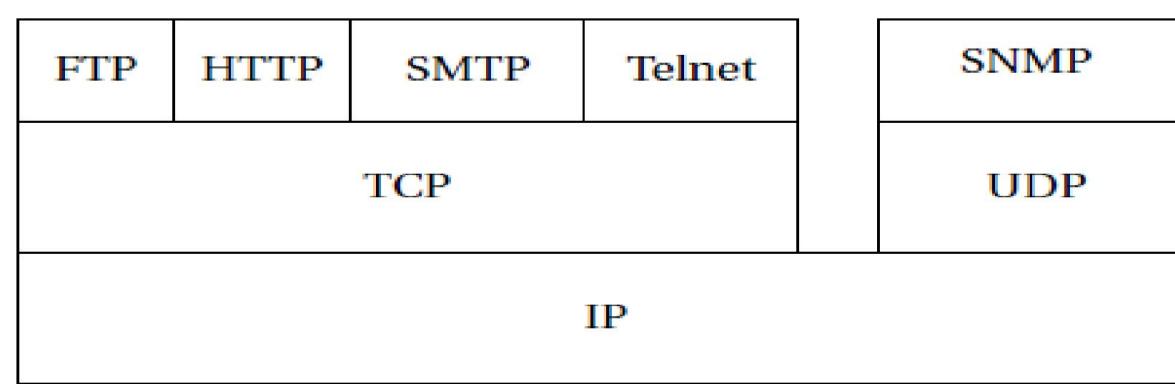
# IP packet structure



- The header and data payload are both of variable length.
- The maximum total length of the header and data payload is 65,535 bytes.
- The IP address is typically written in the form xxx.xx.xx.xx. The names by which users and applications typically refer to Internet nodes, such as foo.baz.com, are translated into IP addresses via calls to a Domain Name Server.
- The fact that IP works at the network layer tells us that it does not guarantee that a packet is delivered to its destination.
- Furthermore, packets that do arrive may come out of order. This is referred to as best-effort routing

- The Internet also provides higher-level services built on top of IP.
- The Transmission Control Protocol (TCP) is one such example.
- It provides a connection oriented service that ensures that data arrive in the appropriate order, and it uses an acknowledgment protocol to ensure that packets arrive

# The Internet service stack



- Using IP as the foundation, TCP is used to provide
  - File Transport Protocol for batch file transfers,
  - Hypertext Transport Protocol (HTTP) for World Wide Web service,
  - Simple Mail Transfer Protocol for email,
  - Telnet for virtual terminals.
- A separate transport protocol, User Datagram Protocol, is used as the basis for the network management services provided by the Simple Network Management Protocol.

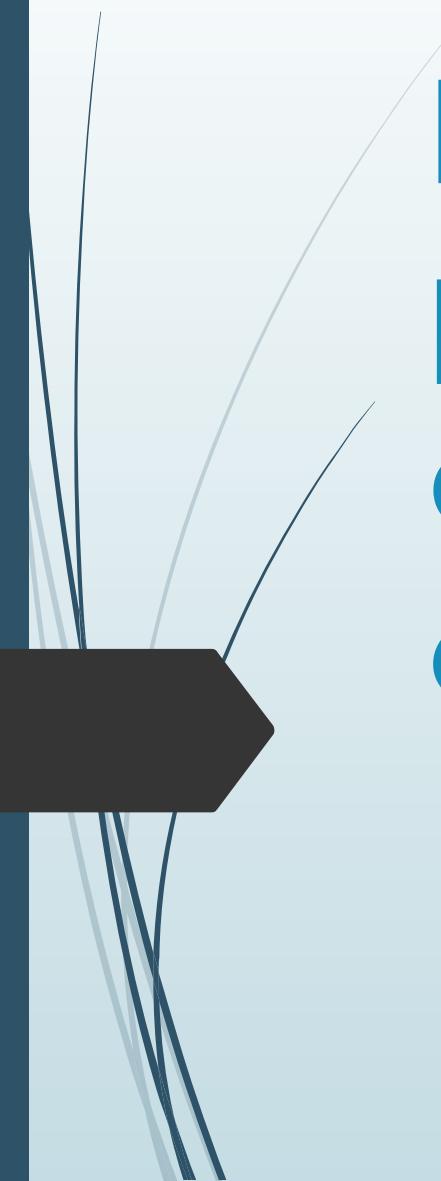


# Internet Applications

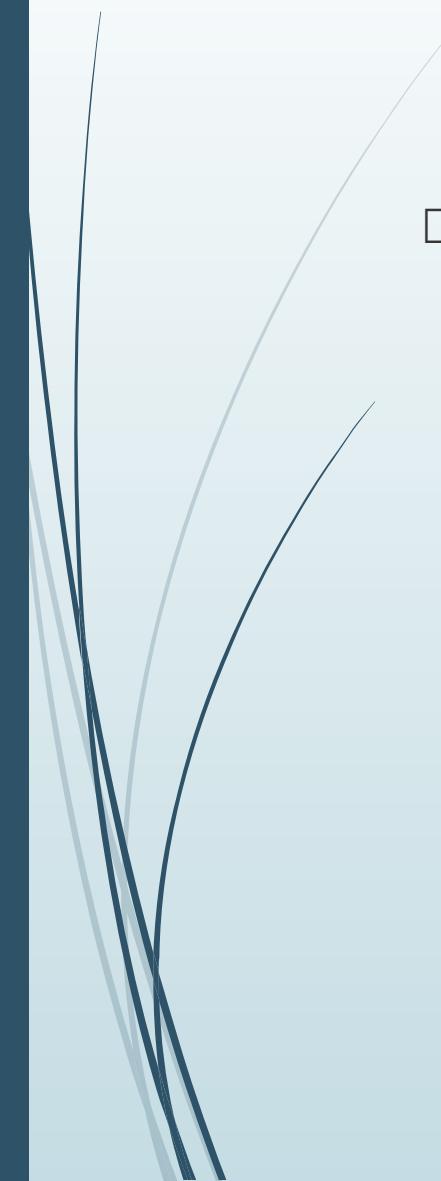
- One of the earliest Internet-enabled embedded systems was the laser printer. High-end laser printers often use IP to receive print jobs from host machines.
- Portable Internet devices can display Web pages, read email, and synchronize calendar information with remote computers.
- A home control system allows the homeowner to remotely monitor and control home cameras, lights, and so on

# Internet Security

- They point out that security can be enforced at all levels of the network stack.
- General network security principles can be applied to Internet-enabled embedded systems;
- Various industrial standards also deal with measures specific to industrial networks.



# Embedded product development life cycle



# EDLC

- It is Analysis-Design-Implementation based standard problem solving approach for Embedded Product Development

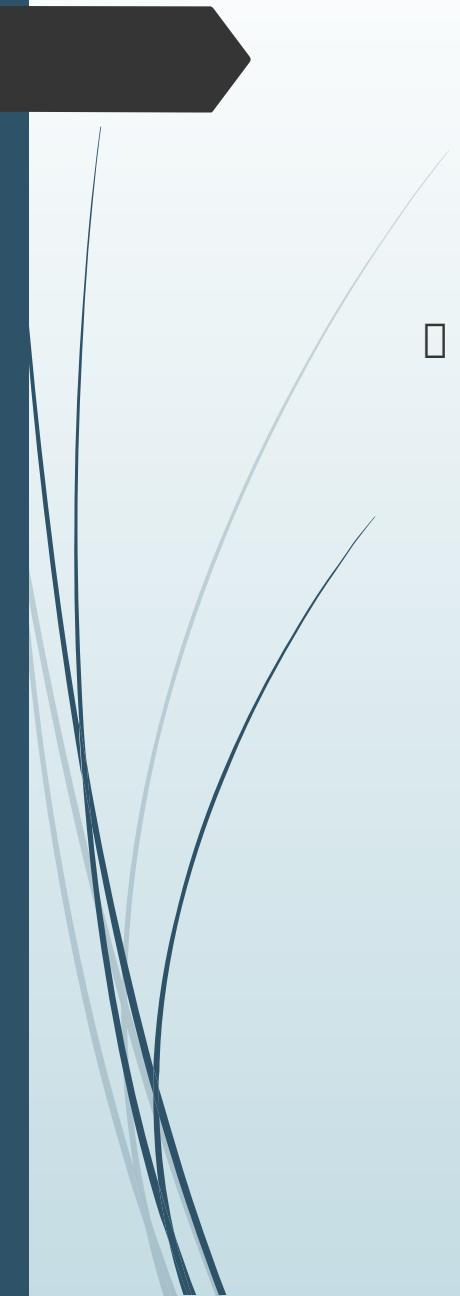


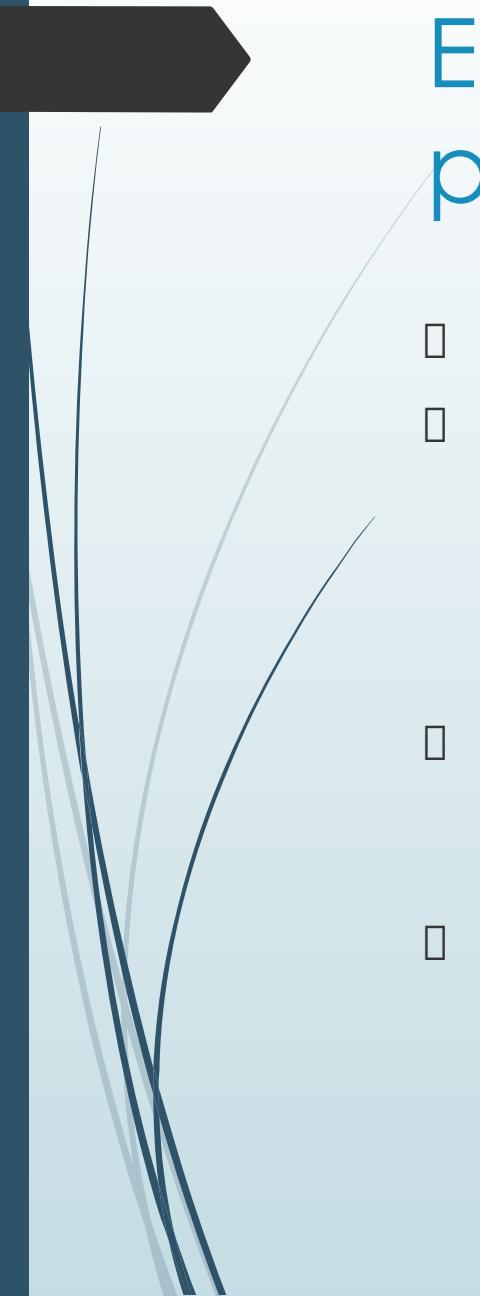
# Why EDLC

- EDLC is essential to understand the scope and complexity of the work involved in embedded system product development.
- EDLC defines the interactions and activities among various group of product development sector including project management, system design and development, system testing, release management and quality assurance.
- This makes the product developer independent and uniformity in development approaches.

# Objectives of EDLC

- Produce marginal benefit
- It is expressed in terms of Return of Investments(ROI)
- The investment for a product development includes initial investment, manpower investment, infrastructure investment, supply chain and marketing investment etc.
- A product is said to be profitable if the turn over from selling of the product is more than that of overall investment expenditure.
- For this the product should be acceptable to the end user.
- So it is essential to make sure that the product is meeting all the criteria through design, development, implementation and support phases.

- 
- Main objectives
    - Produce marginal profit
    - Ensure high quality products are delivered to the user
    - Risk minimization and defect prevention in product development through project management
    - Maximise the productivity



# Ensuring high quality products

- Quality of embedded product is ROI
- Expense includes,
  - Initial investment
  - Training, recruitment
  - Infrastructure set up
- There should be some budget allocation for the development of product, allocated by higher officials by studying the market trends
- EDLC ensure that the development of the product has taken account of all the qualitative attributes of the embedded system



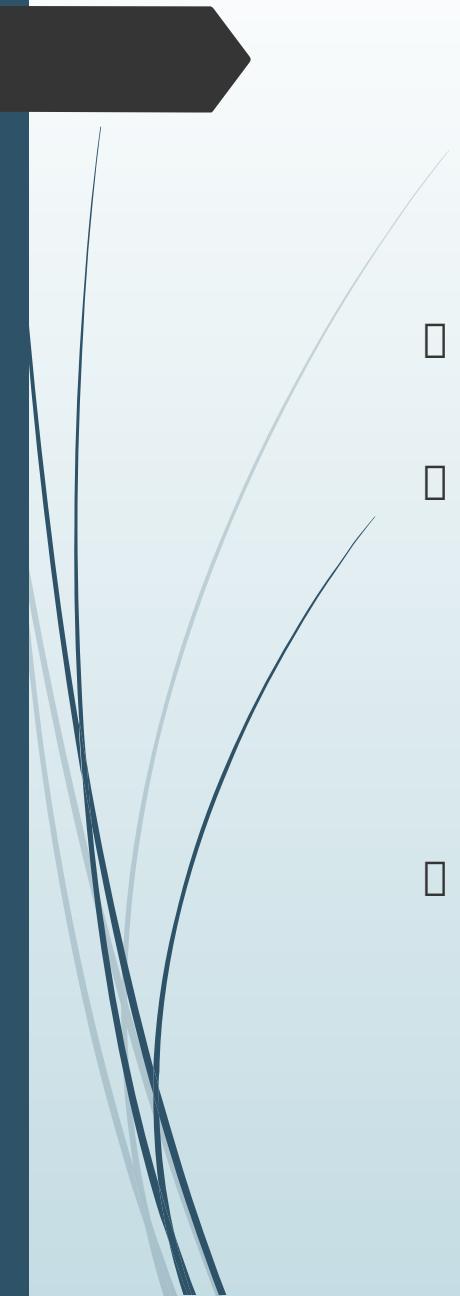
# Risk minimization and defect prevention

- If the project development process is a simple one team manager can handle the development process
- Higher official will just monitor the process.
- But if the development process is tight a skilled project management team will ensure that the development process is going in the right direction.

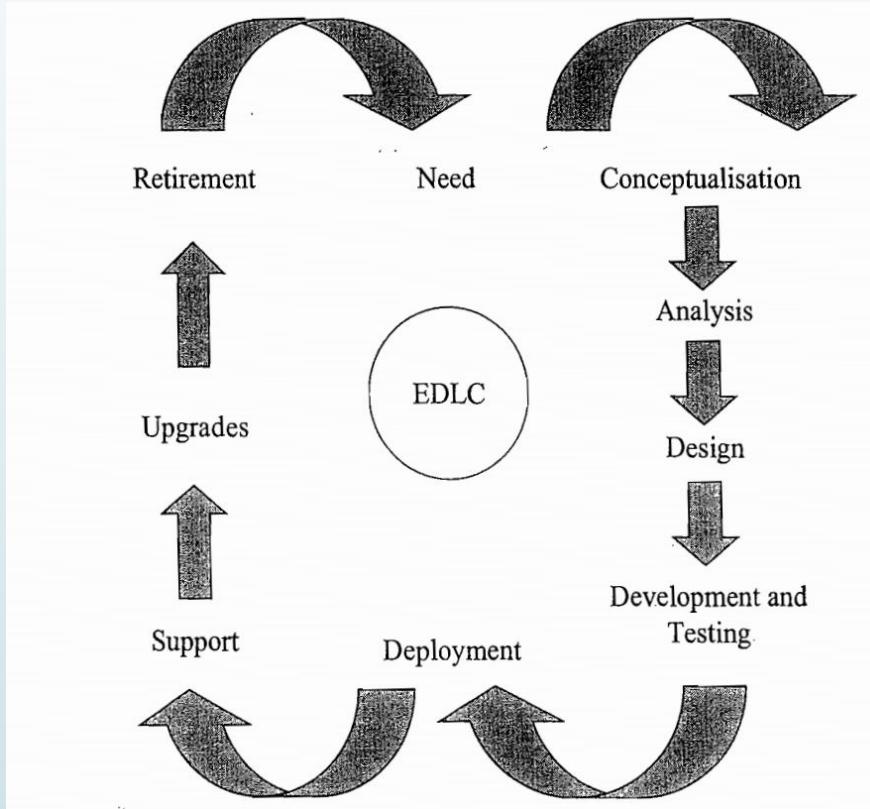
- Project management is essential for
  - predictability
    - Estimate on the time of completion (use previous history)
    - Resource allocation(persons, system, backup facility)
  - Coordination
    - Communication with partners, suppliers and client should be proper
  - risk minimization
    - Backup of resources to overcome critical situation
    - Ensuring defective product is not developed

# Increased productivity

- Measure of efficiency as well as ROI\
- Different ways to improve the productivity are
  - Saving the manpower
    - X members – X period
    - X/2 members – X period
  - Use of automated tools where ever is required
  - Re-usable effort – work which has been done for the previous product can be used if similarities present b/w previous and present product.
  - Use of resources with specific set of skills which exactly matches the requirements of the product, which reduces the time in training the resource

- 
- A life cycle of product development is commonly referred as the “model”
  - A simple model contains five phases
    - Requirement analysis
    - Design
    - Development and test
    - Deployment and maintenance
  - The no of phases involved in EDLC model depends on the complexity of the product

# Phases of EDLC





# Need

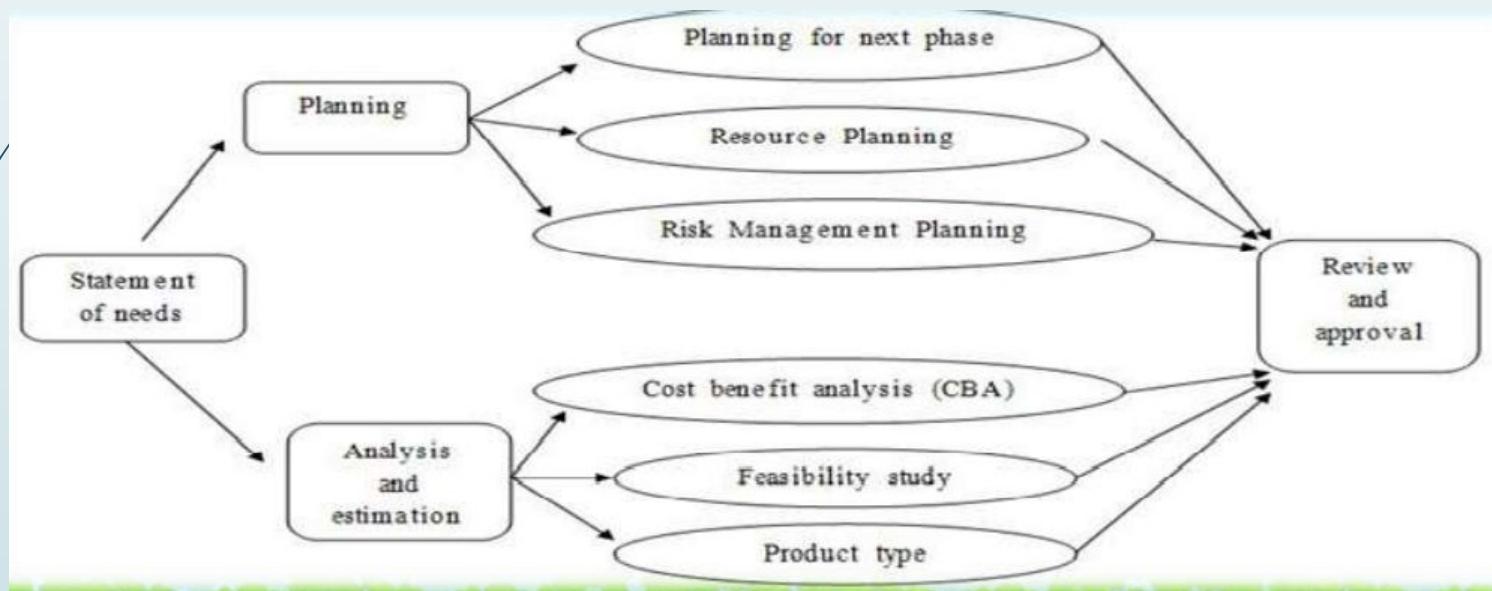
- Any embedded product may evolves as an output of a need.
- Need may come from an individual/from public/from company(generally speaking from an end user/client)
- It is developed as a conceptual proposal
- The proposal must be reviewed by funding agency and by senior management for approval
- The product development need involves,
  - New/custom product development
  - Product re-engineering
  - Product maintenance

- 
- New or custom product development
    - A need for a particular product which does not exist in the market or a product which act as a competitor to an existing product in the market will lead to the development of a completely new product.
  - Product re engineering
    - It is the process of making changes in an existing product design and launching it as a new version
    - Termed as product upgrade
    - Result of re engineering
      - Change in business requirement
      - User interface enhancement
      - Technology upgrade

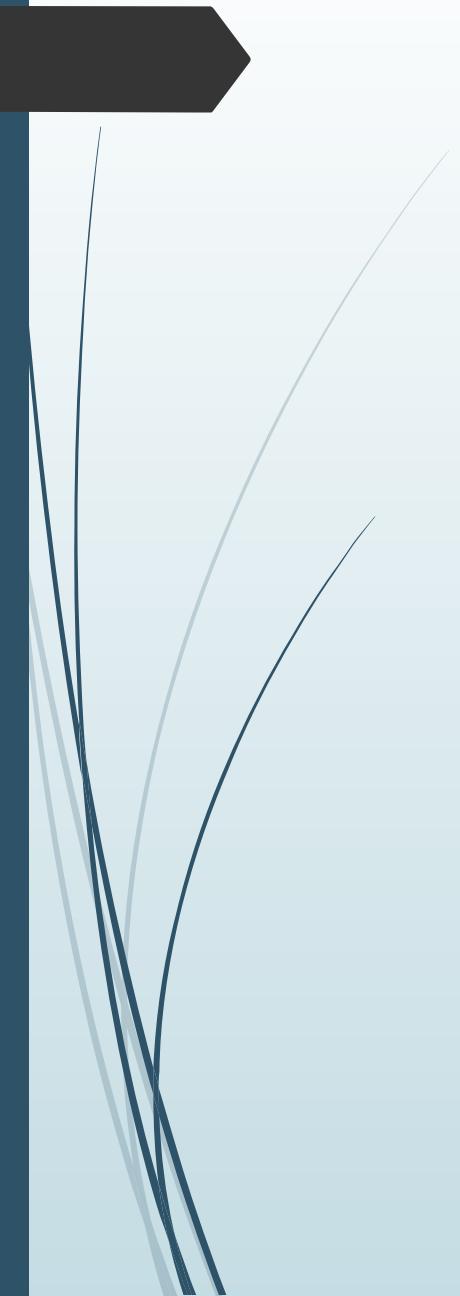
- Product maintenance
  - Provide technical support to the end user
  - Comes as a result of product non functioning
  - Two types
    - Corrective
      - Making corrective action due to non functioning
      - May need the replacement or correction of components
    - Preventive
      - Periodic check to avoid failure

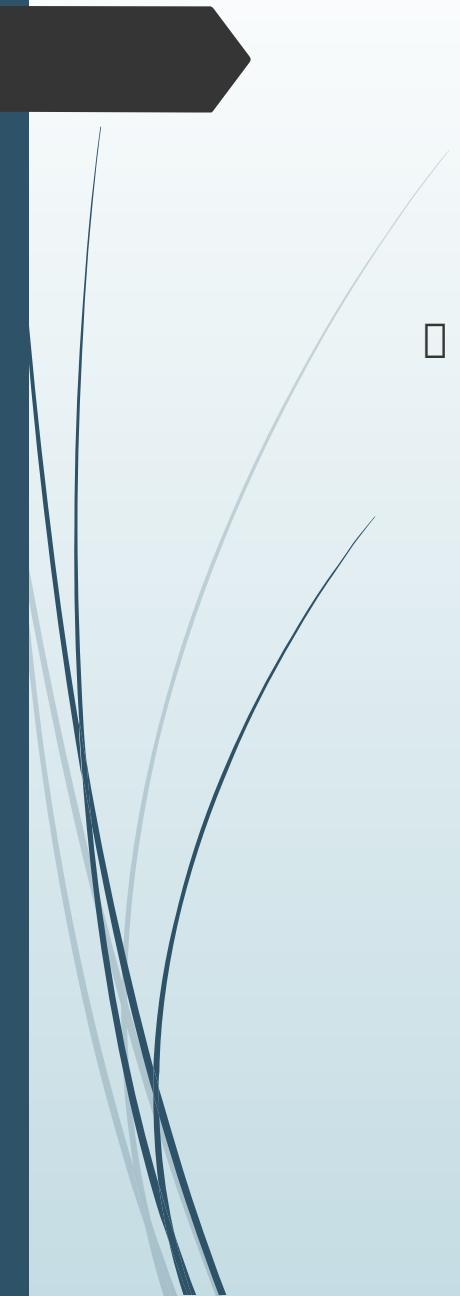
# Conceptualization

- Defines the scope of the concept, cost benefit analysis, feasibility study and prepares project management and risk management plans
- The end user is convinced at this phase



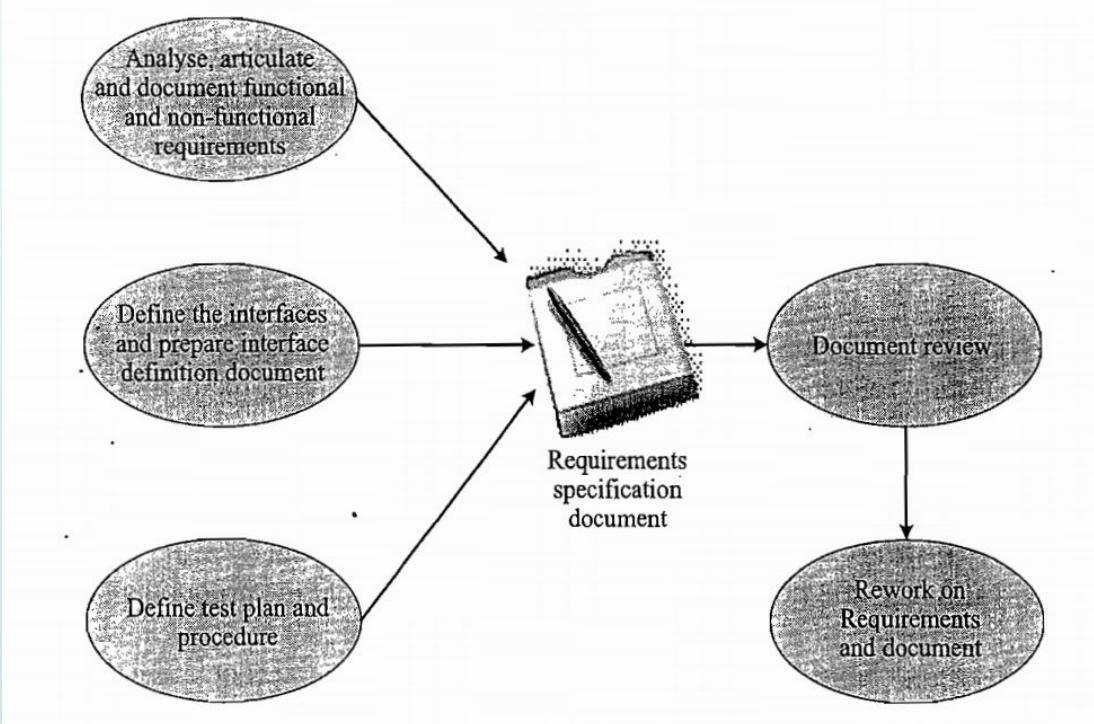
- Conceptualization involves two activities
  - Planning
  - Analysis and study
    - To understand the opportunity of the product in the market
  - Involves
    - Feasibility study
      - Involves examine need of product
      - Find out other alternative solutions
      - Analyse technical and financial feasibility

- 
- Cost benefit analysis
    - Identifying total development cost
    - Identify profit from the product
    - Principles of CBA
      - Common unit of measurement(money)
      - Market choice based benefit measurement (price)
      - Targeted end users
    - Product scope
  - Planning activities
    - Resource planning
    - Risk management plans

- 
- At the end of conceptualization analysis and planning activities are submitted to the client/project sponsor with any of the recommendation
    - Product is feasible proceed to next step
    - Product is not feasible, scrap the project

# Analysis

- Requirement analysis is performed to develop a detailed functional model of the product under consideration





# Phases in analysis

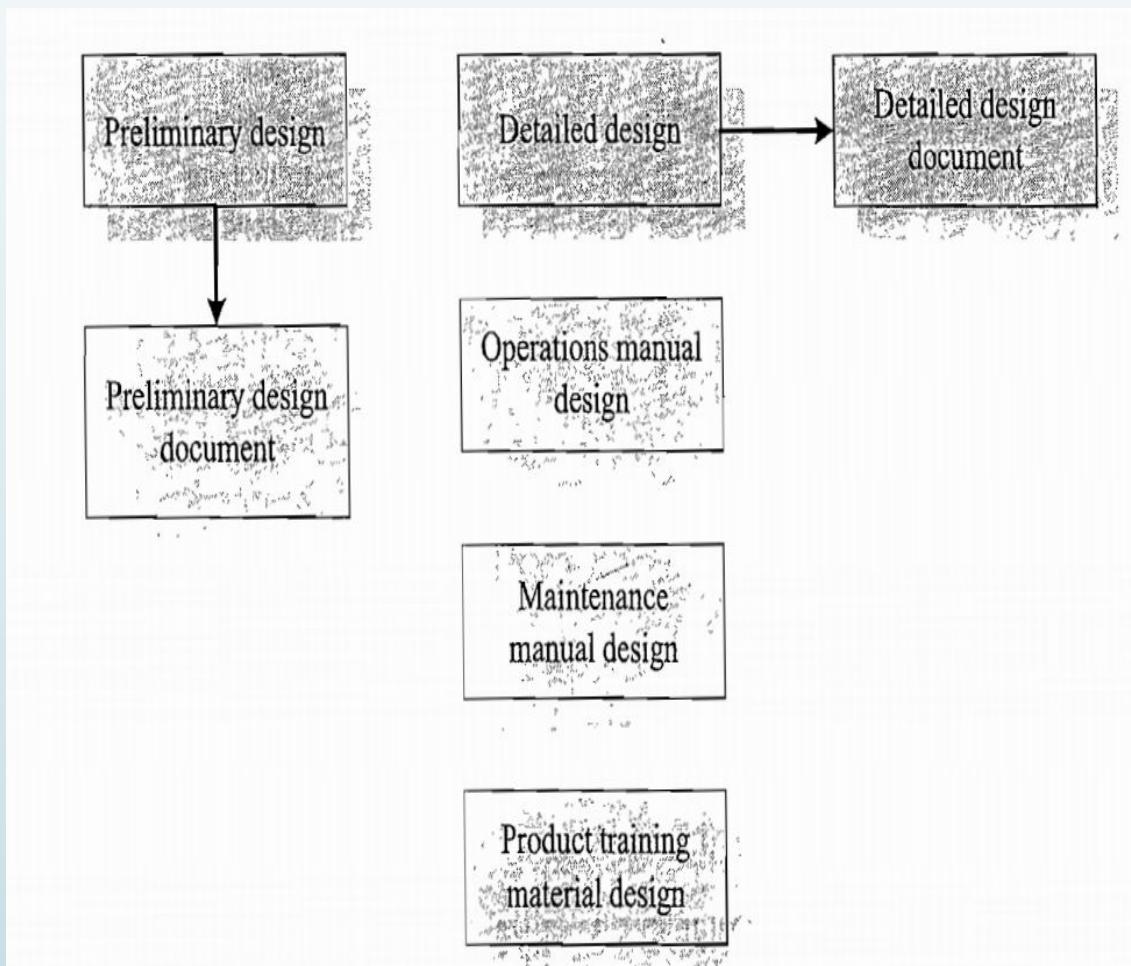
- Analysis and documentation
  - Consolidates business need of the product
  - Analyses purposes of product
  - Addresses various functional and non functional aspects of product
  - Identify data requirement
  - Identify operational and non operational attributes
  - Product external interface requirement
  - User manuals
  - Maintenance requirements
  - General assumption

- Interface definition and documentation
  - If part of another system interface needed
- Defining test plan and procedures
  - Define test procedure
  - Test set up
  - Test environment
  - Create master test plan
  - Needed for ensuring total quality

# Test performed

- Unit testing
- Integration testing
- System testing-function testing
  - Usability testing
  - Load testing
  - Security testing
  - Scalability testing
  - Sanity testing
  - Smoke testing
  - Performance testing
  - Endurance testing
- User acceptance testing
- Final document is Requirement Specification Document

# Design

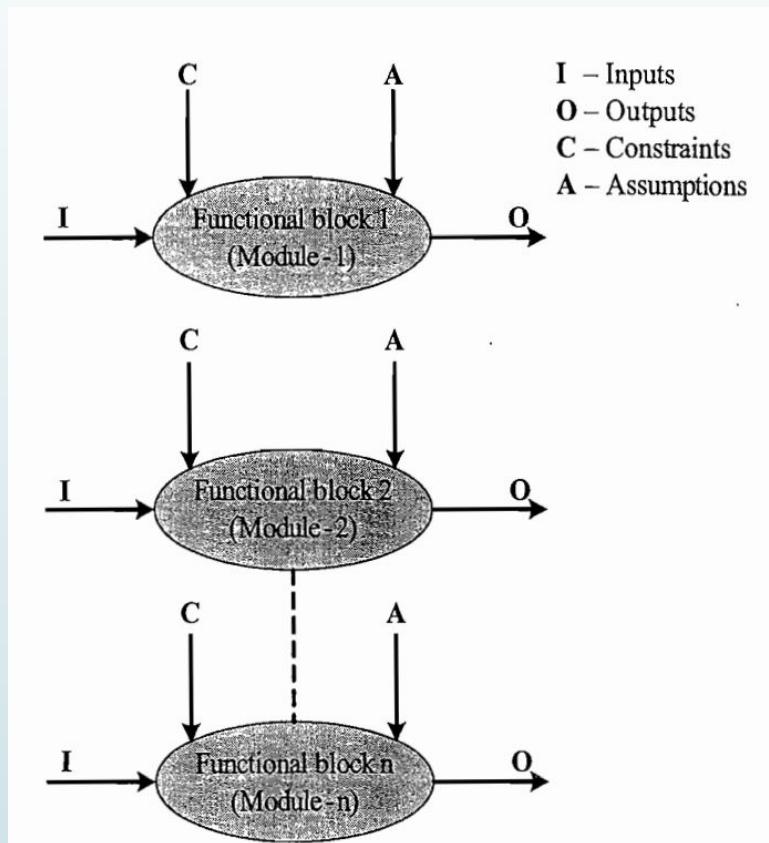


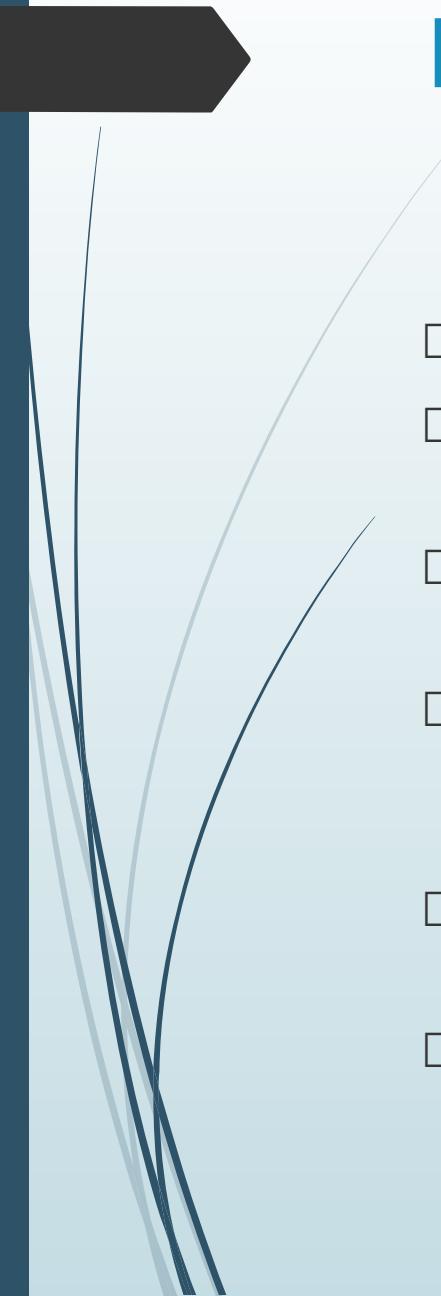
# Preliminary design

- Establishes top level architecture of the product
- List various functional blocks
- Define input and output of each functional blocks
- It is given to the end user for verification

# Detailed design

- Generates detailed architecture, identifies and lists various components for each functional block, interconnection of various functional blocks, control algorithms etc.
- Get approval from end user





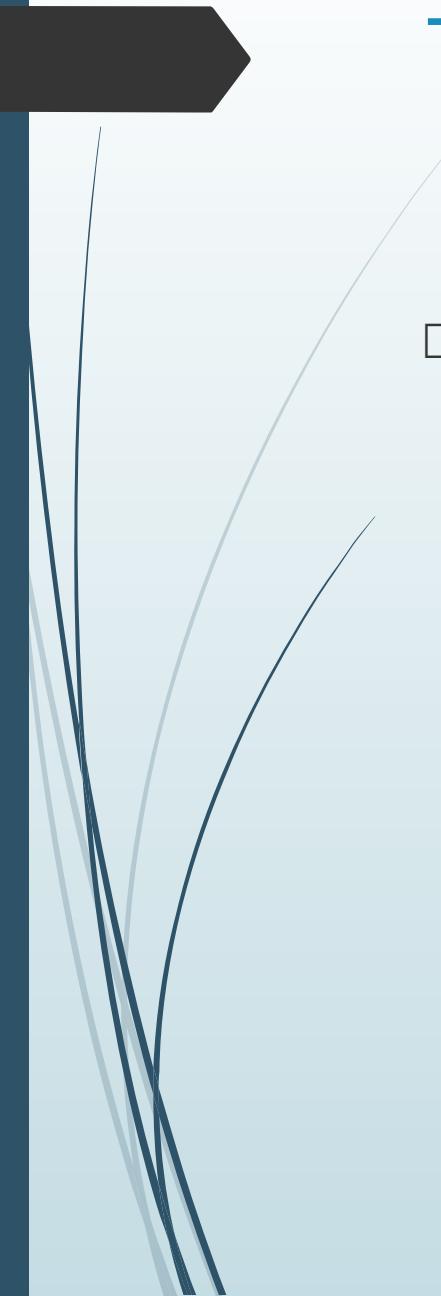
# Development and testing

- Transform design into reliable products
- Detailed specification in the design phase is translated into hardware and firmware.
- During development phase the installation and setting up of various development tool is performed.
- The development can be partitioned into embedded hardware development, firmware development and product enclosure development.
- Hardware component refers to the platform for placing the components-PCB
- Mechanical enclosure is designed using Solid Works, AutoCaD etc.



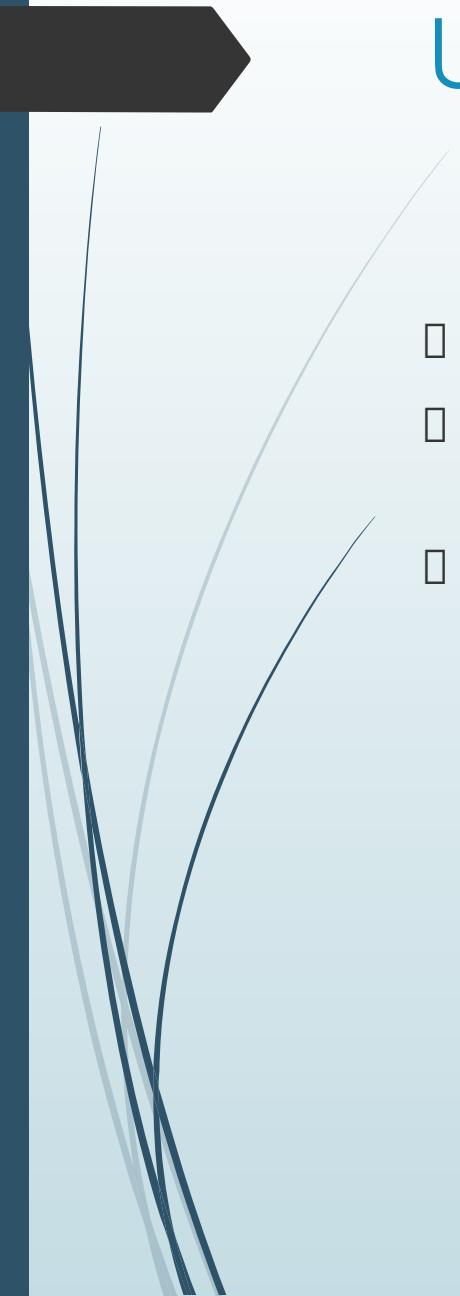
# Lead time

- The time required for the component to deliver after placing the purchase order
- Component procurement should be planned well in advance to avoid delay in development
- Other important activities during development phase
  - Preparation of test cases
  - Preparation of test files
  - Development of different manuals



# Testing

- Divided into
  - Independent testing of firmware and hardware(unit testing)
  - Testing of product after integration (integration testing)
  - Testing of the whole system for functionality and non functionality (System testing)
  - Testing of the whole product against all acceptance criteria mentioned by the client/end user for each functionality (Acceptance testing)



# Unit testing

- Test plan is prepared
- Test cases are identified for the testing of functionality of the product as per design
- Unit test is usually performed by the hardware or firmware developer as part of development phase



# Integration test

- Once unit testing is completed, they are integrated using different firmware integration tools
- Integrated product is tested for different functionalities
- Test plan is generated
- Test cases are developed
- It ensures that the functionality of independent unit is achieved well even after integration
- Integration test results are logged in and firmware/hardware is reworked against any flows detected during integration testing
- It is done at the end of the development phase.

# System testing

- Consists of various test for functional and non functional requirements verification
- Kind of black box testing, doesn't need any kind of design information
- Evaluates the products compliance with the requirements

# Acceptance testing

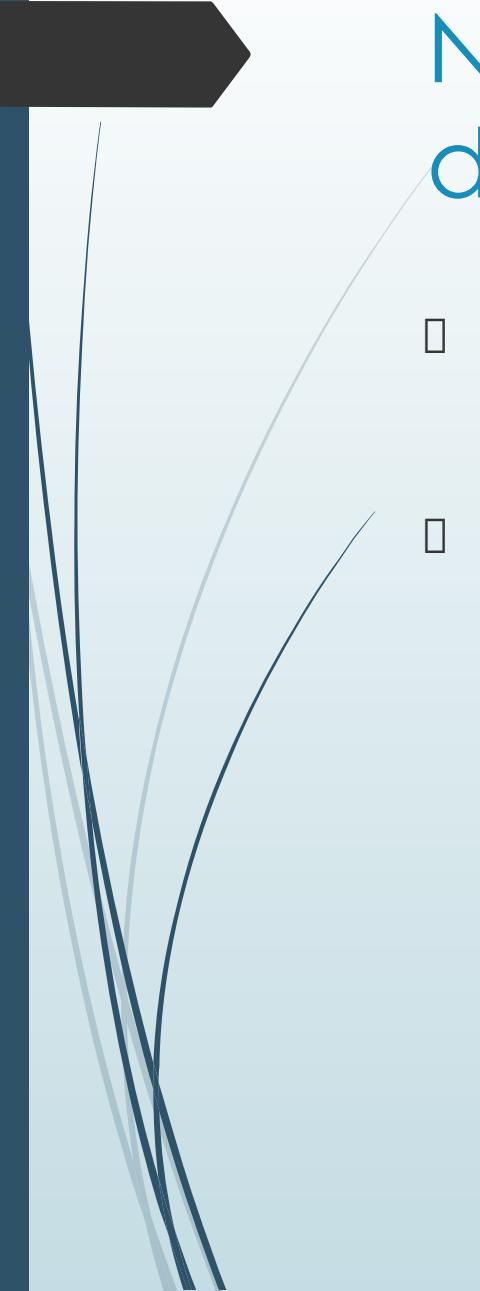
- Product evaluation performed by the customer as a condition of the product purchase
- During this, the product is tested against the acceptance value set by the customer for each requirement
  - Eg: loading time

# Deliverables

- Deliverables from development and testing phase is
  - Firmware source code
  - Firmware binaries
  - Finished hardware
  - Various test plan
  - Test cases
  - Test reports

# Deployment

- A process of launching fully functional model into the market.
- It is known as First Customer Shipping (FCS)
- Deployment phase is initiated after the system is tested and accepted by the end user.
- Important task during this phase
  - Notification of product deployment
  - Execution or training plan
  - Product installation
  - Product post implementation review



# Notification of product deployment

- The product launching ceremony details should be communicated to the stake holders and to the public in case of commercial product
- Notification should include
  - Deployment date, time and venue
  - Briefing about the product
  - Targeted end users
  - Extra features compared to an existing product
  - Product support information



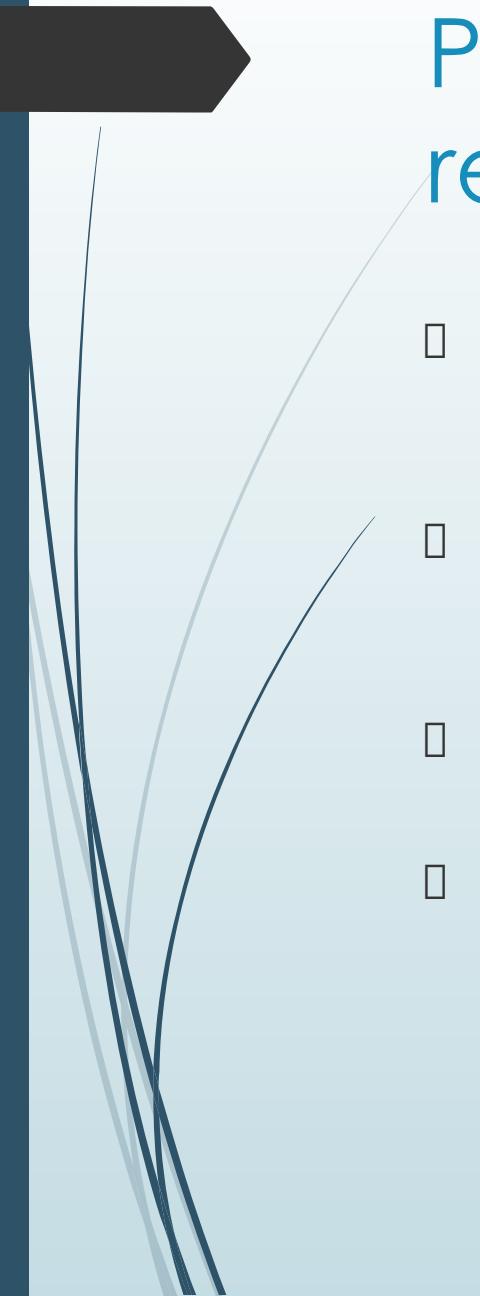
# Execution or training plan

- Training should be given to the end user.
- Training plan should be developed at the early stages
- Proper training will help to reduce possible damages to the product as well as the operating person including product mal function, personal injuries due to inappropriate usage
- User manual helps in understanding the product, its usage, and access its functionalities
-



# Product installation

- Install the product as per installation manual
- Eg: installing complete mobile set



# Product post implementation review

- After product is launched in the market, a post implementation review is to conduct to determine the success of the product
- The aim of this review is to document the problems faced during installation and the solution adopted to overcome them
- It will act as a reference document for future product development
- Helps in understanding the customer needs and expectations of the customer to be implemented in the next version of the product

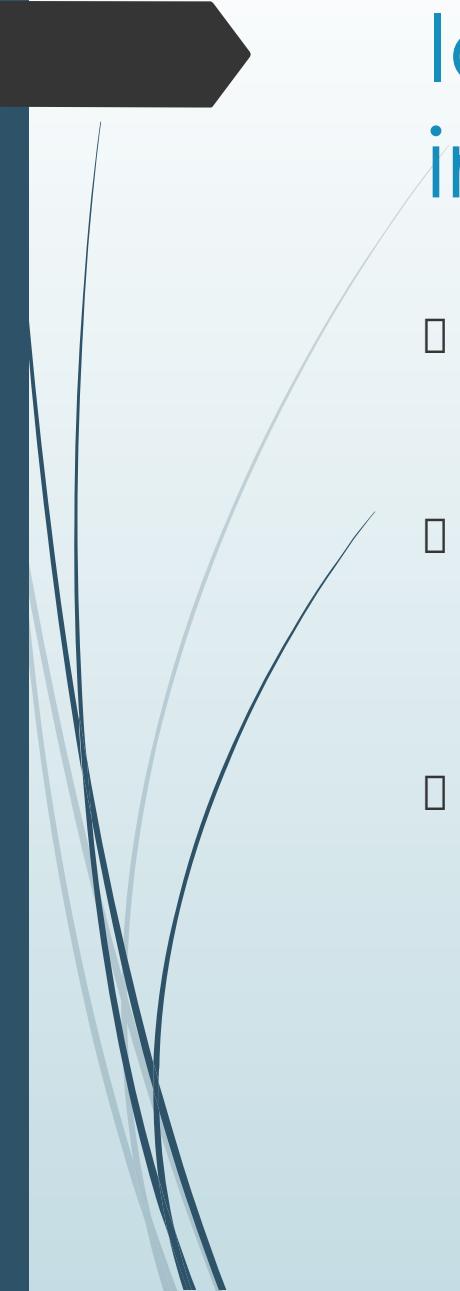
# Support

- Deals with the operation and maintenance of the product
- Support should be provided to the end user/client to fix the bugs of the product
- Support phase ensures that the product meets the user needs and continues functioning in the production environment
- Activities of support wing
  - Set up a dedicated support wing
  - Identify bugs and areas of improvement



# Set up a dedicated support wing

- Certain ES requires 24X7 support in case of product failure or malfunctioning
  - Eg: patient monitoring system
- Set up a dedicated support wing and ensure high quality service is delivered to the end user
- This helps in product move in the market
- Support wing should be easily reachable



# Identify bugs and Areas of improvement

- These are occurred due to unexpected operating condition which is not being identified during testing phase
- Give the end user a chance to express their views about the product and suggestions, if any in terms of modification or future enhancements. Through user feedback.
- Conduct product specific surveys and collect as much as data as possible from end user.

# Upgrades

- Releasing of new version for the product which is already exists in the market
- Releasing of major bug fixes or feature enhancement requirement from the end user.
- In this phase sometimes the detailed design may change or new features are incorporated in order to provide support.
- This modification may be for hardware(hardware upgrades) or firmware(firmware modification)
- Firmware modification will provide a new version for backward traceability



# RETIREMENT/DISPOSAL

- Everything changes, the technology you feel as the most advanced and best today may not be the same tomorrow
- Due to this the product cannot sustain in the market for long
- When the product manufacturer finds that there is another powerful tool in the market for the same purpose, the announce the current product as obsolete and newer version is released soon
- Dispositions reasons are,
  - Rapid technology advancement
  - Increased user needs

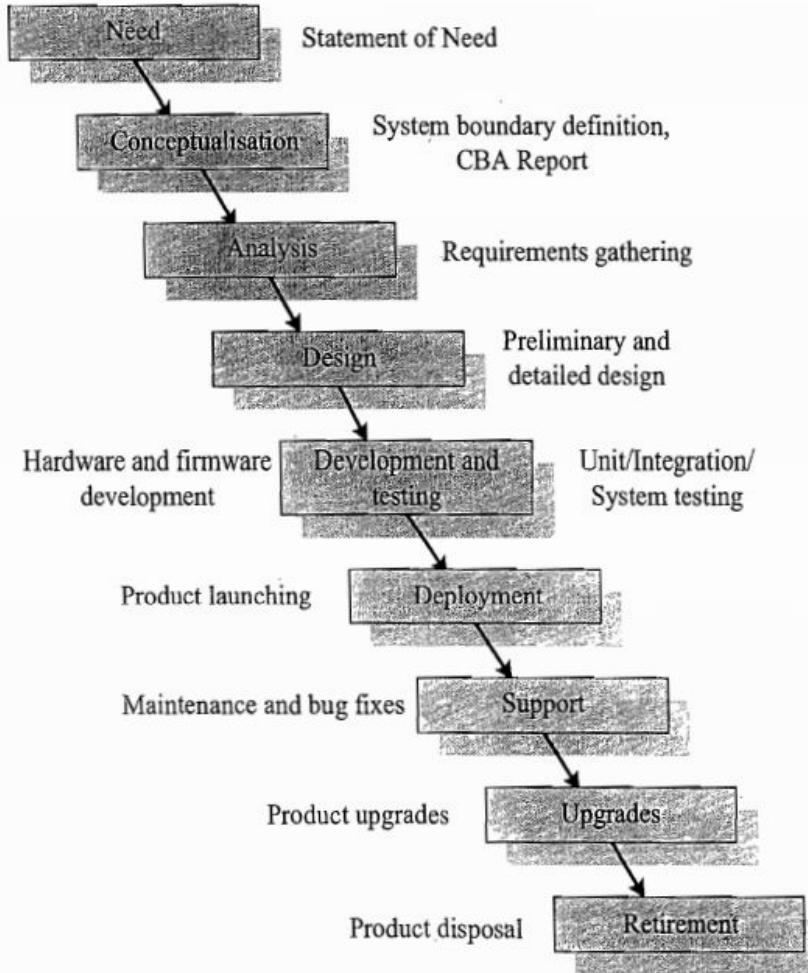


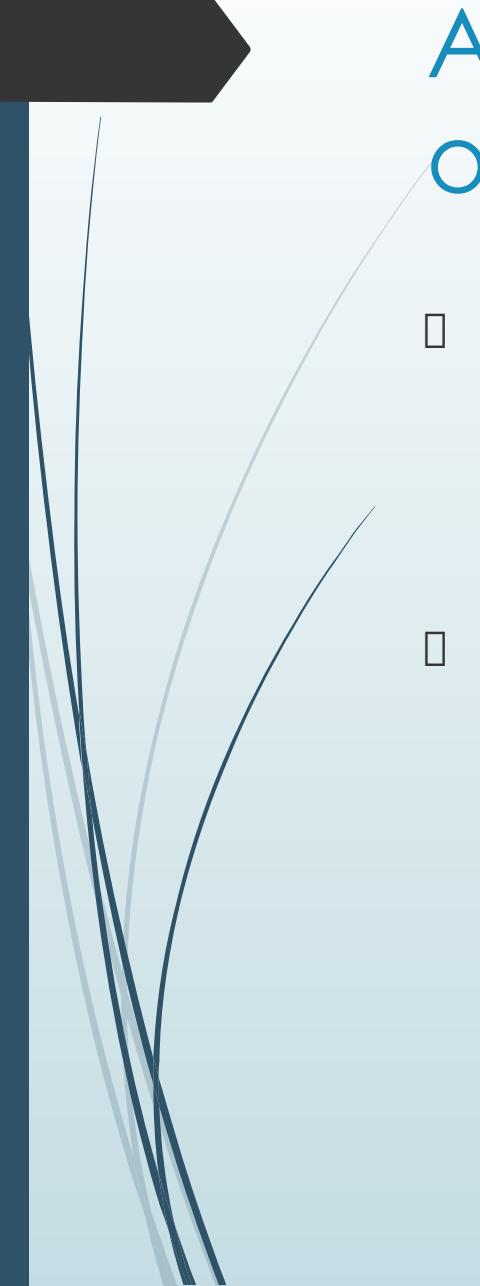
# EDLC approaches

- Linear or Waterfall model
- Iterative/Increment or Fountain Model
- Prototyping/ Evolutionary Model
- Spiral Model

# Linear/Waterfall model

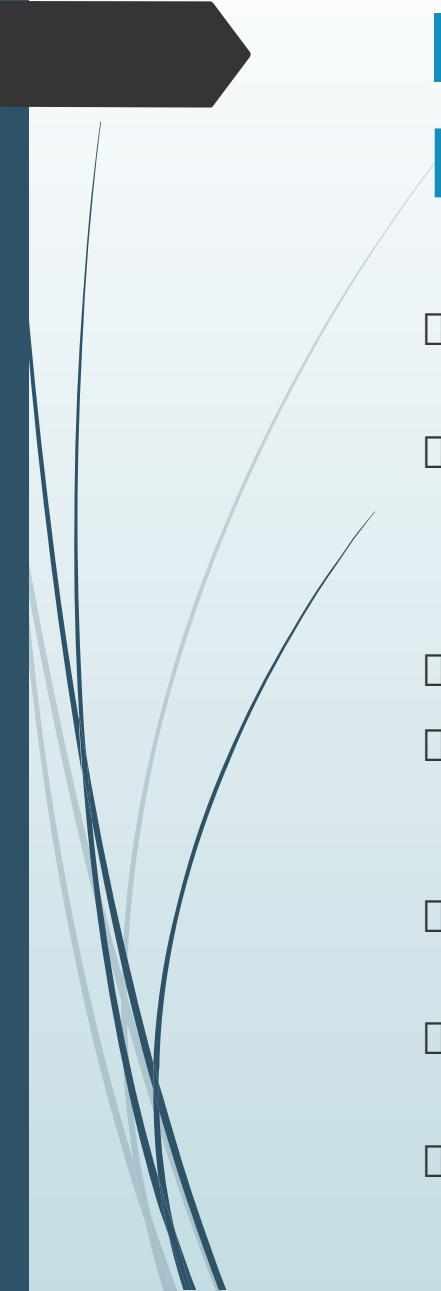
- Each EDLC phase is executed in sequence
- Output of one phase is input of another phase
- Feedback of each phase is available locally as document
- The linear model implements excessive review mechanism to ensure that the process flow is going in right direction and validates the effort during a phase
- Even if bugs are identified, corrective action will be taken only in the support phase.





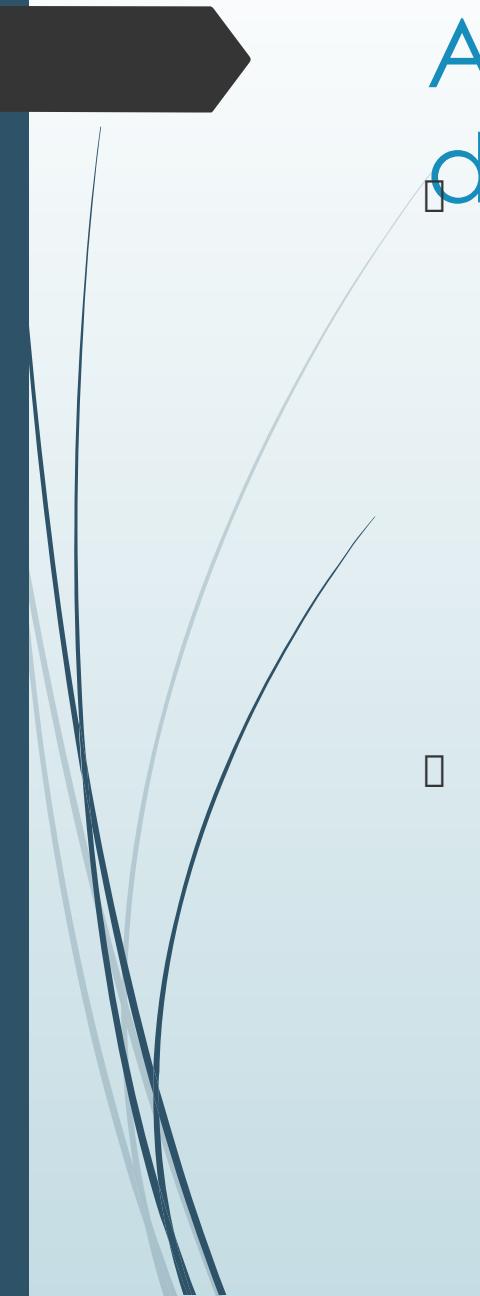
# Advantages and drawbacks of Waterfall model

- Advantages
  - Rich documentation
  - Easy project management
  - Good control over cost and schedule
- Drawbacks
  - Risk analysis is done only once
  - Risk after the change is not made



# Iterative/Incremental or Fountain model

- This model Do some analysis, follow some design, then some implementation.
- Evaluate it, and based on the short coming, cycle back through and conduct more analysis, opt for new design and implementation and repeat the cycle till the requirements are met completely.
- It is a cascade series of linear models
- Incremental model is a superset of iterative model where the requirements are known at the beginning and are divided into different groups.
- The core set of functions in the first cycle is identified and is built and deployed as the first release
- Second set of requirement and the bus from the first release will be fixed and release as the second release.
- This process is completed as all the functionalities and requirements are met.



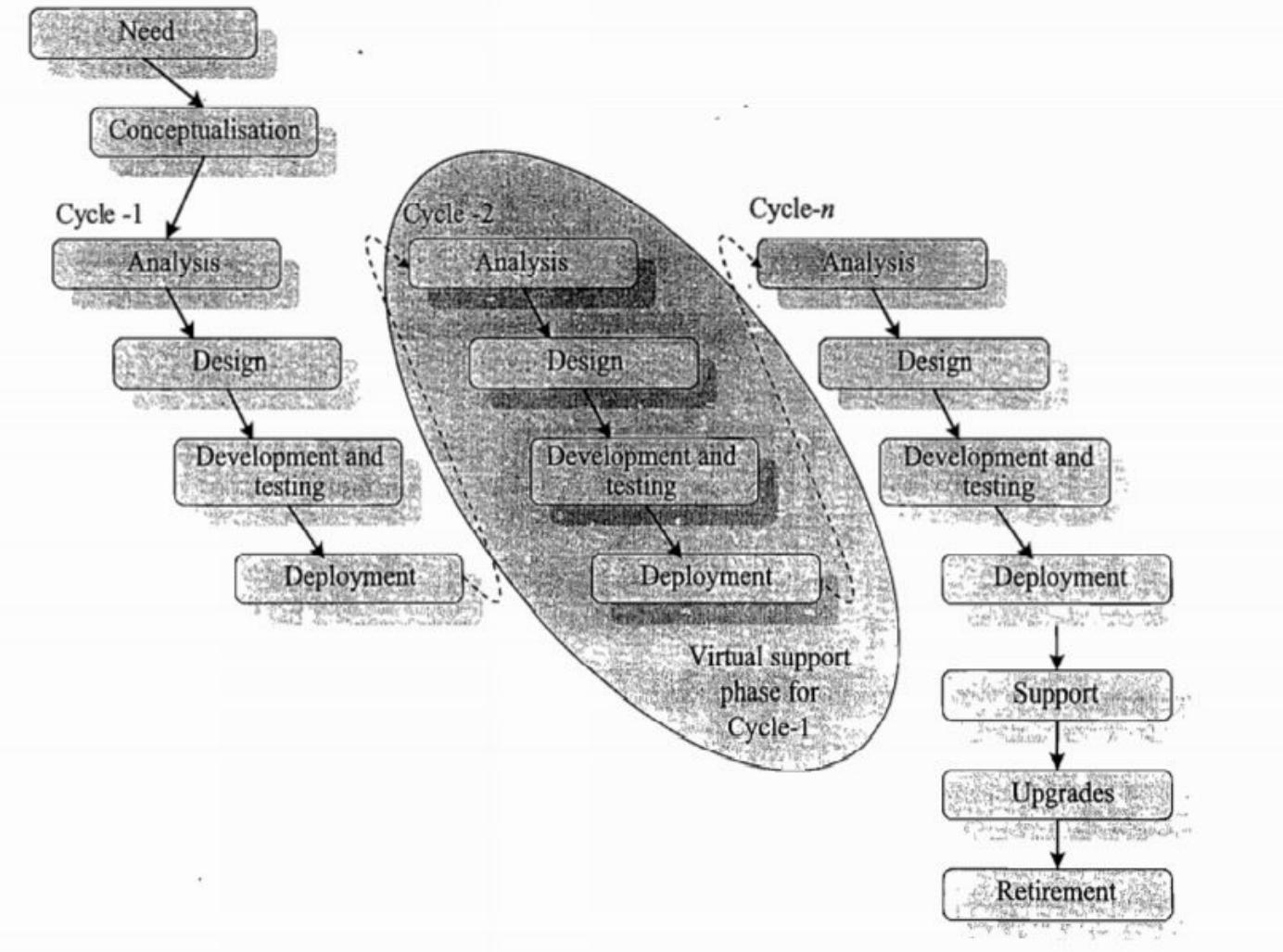
# Advantages and disadvantages

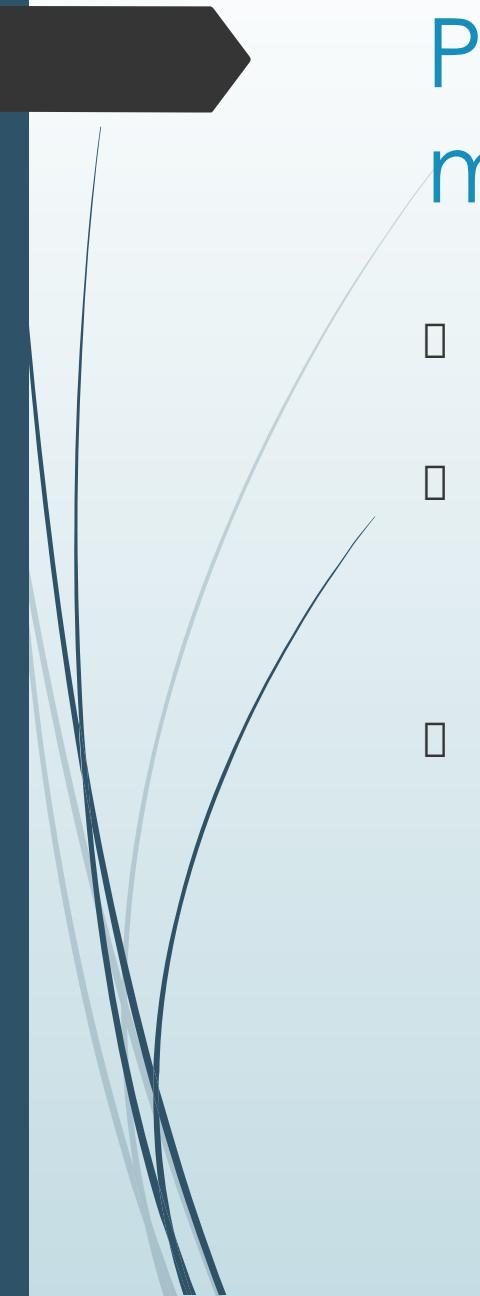
## □ Advantages

- Very good development cycle
- Feedback at each function implementation
- Data can be used as a reference for similar projects
- More responsive to changing needs of user
- Risk can be minimized easily.
- Project management and testing is simpler product development can be stopped at any stage. A product with minimum functionality is obtained.

## □ Disadvantages

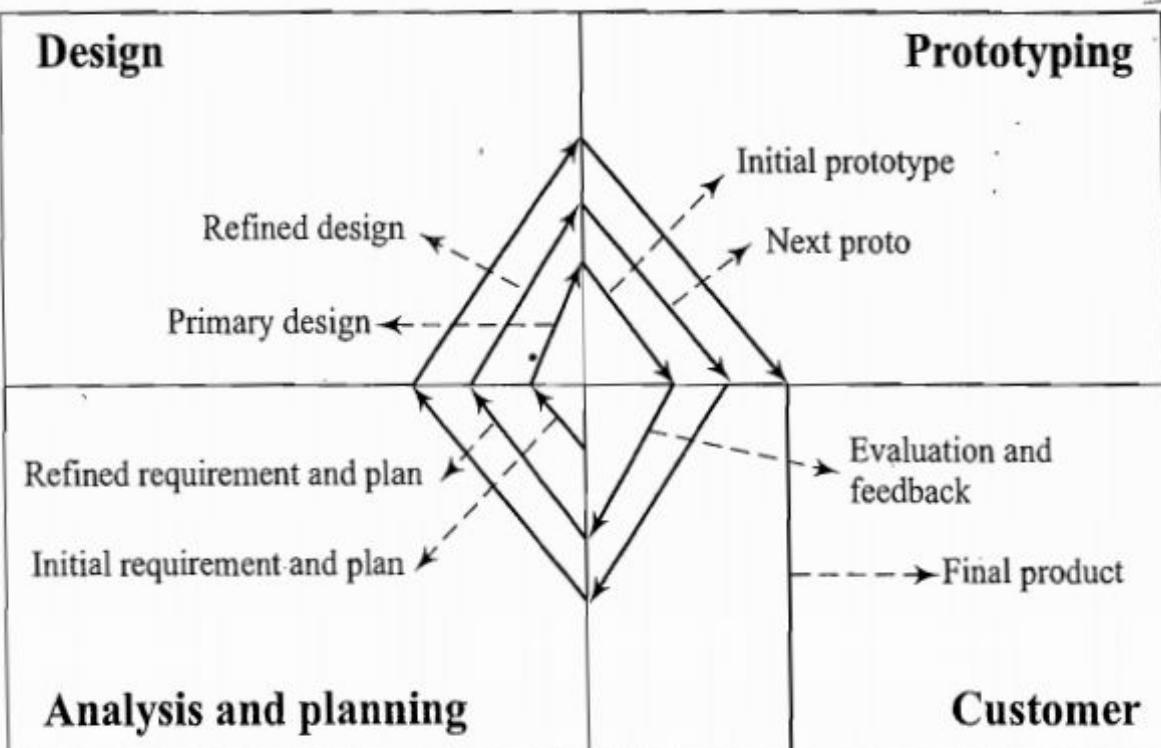
- Excessive review requirement at each cycle.
- Impact on operations due to new releases
- Training requirement for each new deployment
- Structured and well documented interface definition across modules to accommodate changes.

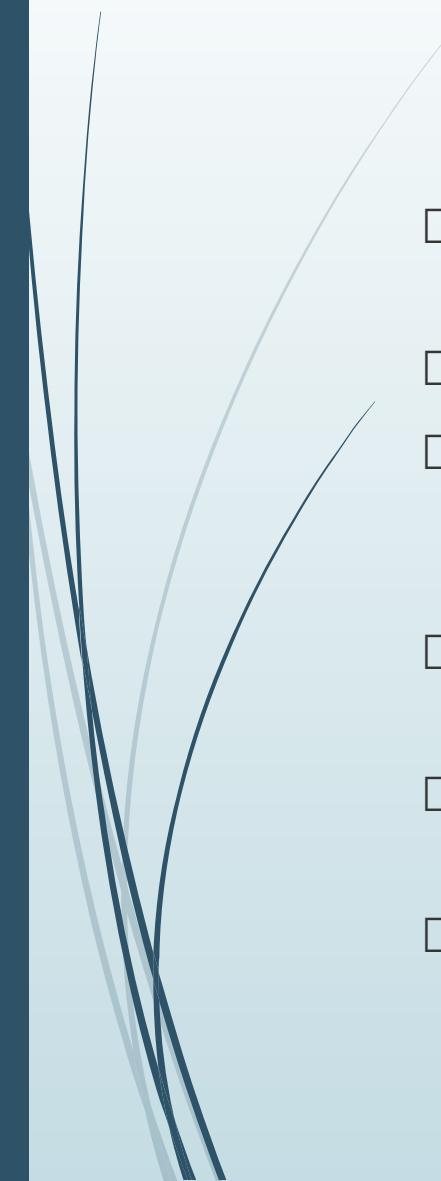




# Prototyping/evolutionary model:

- Similar to iterative model, product is developed in multiple cycles
- The only difference is the model produces more refined prototype of the product at each cycle instead of just adding the functionality at each cycle like in iterative model
- The short coming of the model is evaluated at each cycle and corrected in the next cycle.



- 
- After the initial requirement analysis, the design of the first prototype is made.
  - This is sent to customer for evaluation
  - Customer evaluates it for a set of requirements and gives feedback to the developer in terms of shortcomings and improvement needed.
  - Developer refines the product according to the customers expectation and release new prototype.
  - After some number of iterations, final product launched to the market.
  - Follows the approach requirement definition-pro type development- pro-type evaluation and requirement refining.

# Advantages and drawbacks

## □ Advantages

- Development depends on user feedback. Hence fine refinement is possible.
- Risk is spread across each prototype and is well controlled.

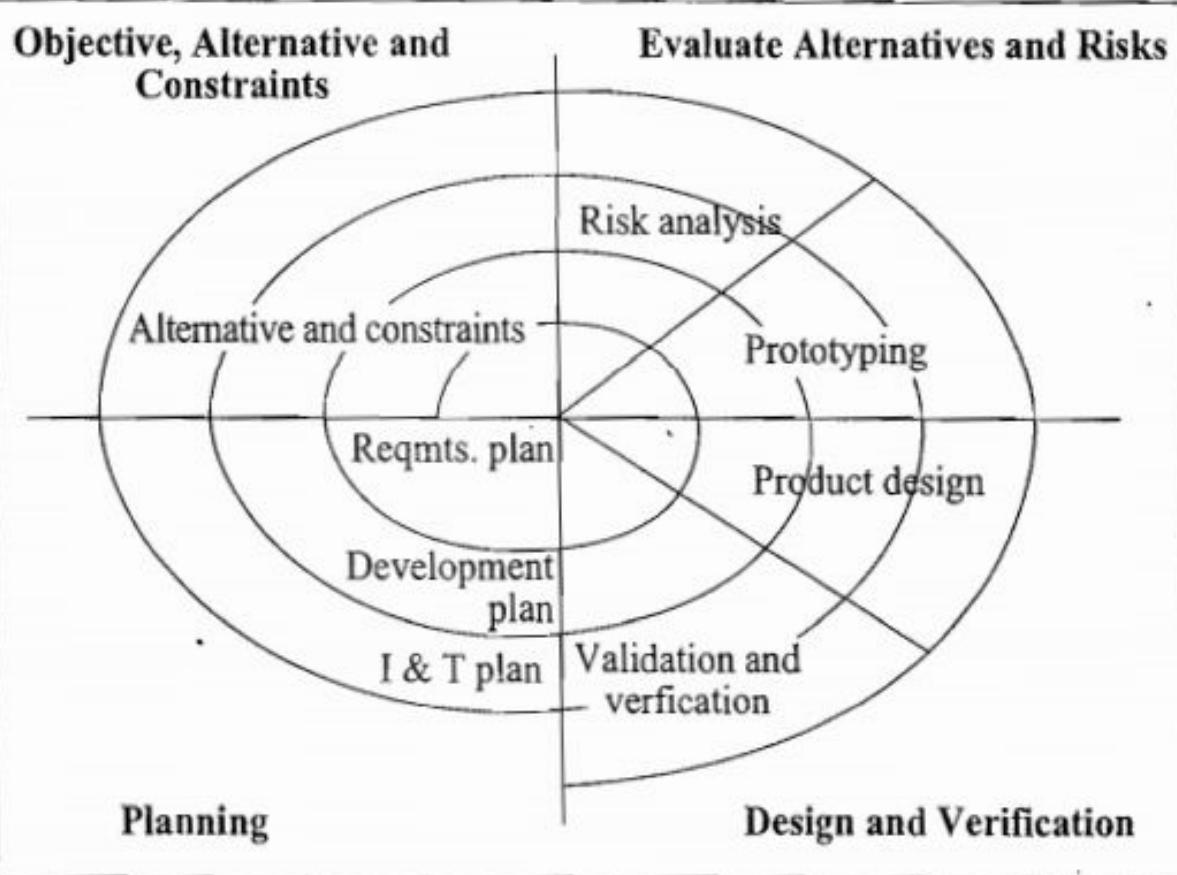
## □ Drawbacks

- Deviation from expected cost and schedule due to refinement
- Increased project management
- Minimal documentation on each prototype, hence problem in backward prototype traceability
- Increased configuration management activities



# Spiral model

- Spiral model combines elements of linear and prototype models to give best possible risk minimization.
- Product development starts with project definition and traverse through all phases of EDLC through multiple phases.
- The activities involved in the spiral model can be associated with four quadrants of a spiral.





# Recent Trends in Embedded Computing.

- Processor trends
- Embedded OS trends
- Development language trends
- Development platform trends

# Processor trends

- System on Chip
- System in Package
- Multicore processor/Chip level multi processor
- Reconfigurable processors

# System on Chip

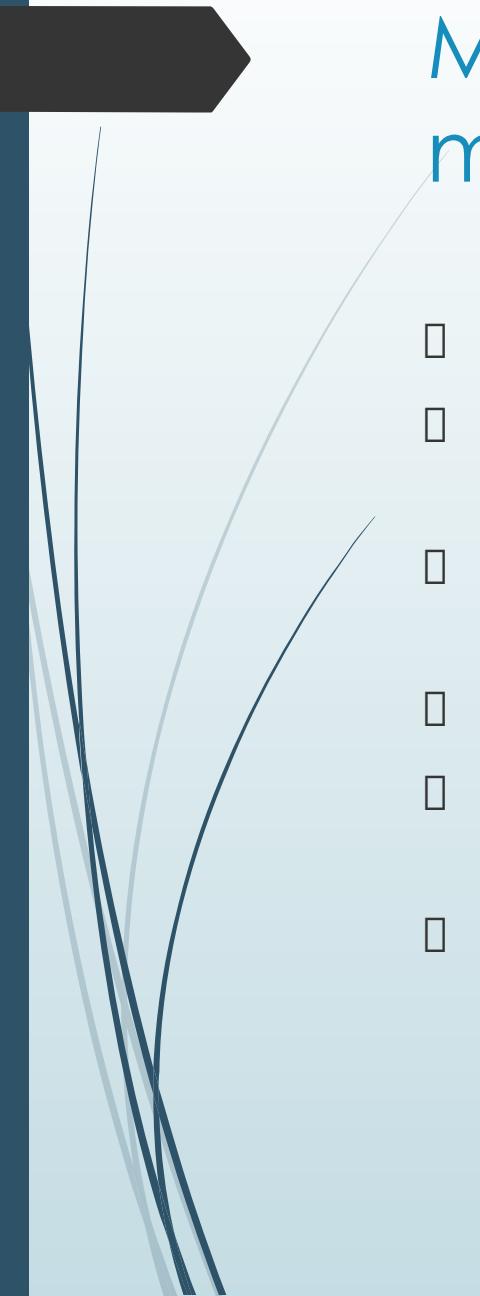
- It makes a system on a chip
- Multiple functions can be performed on this chip
- Soc made the concept of embedded processor from general purpose design to device specific design.
- ,SOC are available for many applications like Set Top Boxes, Portable media player, smart phones etc.
- Eg: iMX31- for multimedia applications.
- Advantages
  - Save board space
  - Leads to concept miniaturization

# System in Packages

- In this sub systems are assembled into a single package.
- It is characterized by one or more ICs of different functionalities which may include massive components assembled into a single package that functions as a system or sub system.
- Good for RF and wireless communication devices, GPS navigation system etc.
- Advantages
  - Reduced time to market
  - Reduced sizing
  - Reduced PCB routing complexity
  - Cost saving on PCB
  - Enhanced performance through shorter interconnection paths
- Users
  - Samsung, Amkor technology, Atmel, Microsemi, CHIPSIP

# Comparison of SoC and SiP

| SoC                           | SiP                                                    |
|-------------------------------|--------------------------------------------------------|
| Built on a single Silicon die | Contains more than one silicon die in a single package |
|                               | Overcome limits of SoC.                                |



# Multicore processor/Chip level multi processor

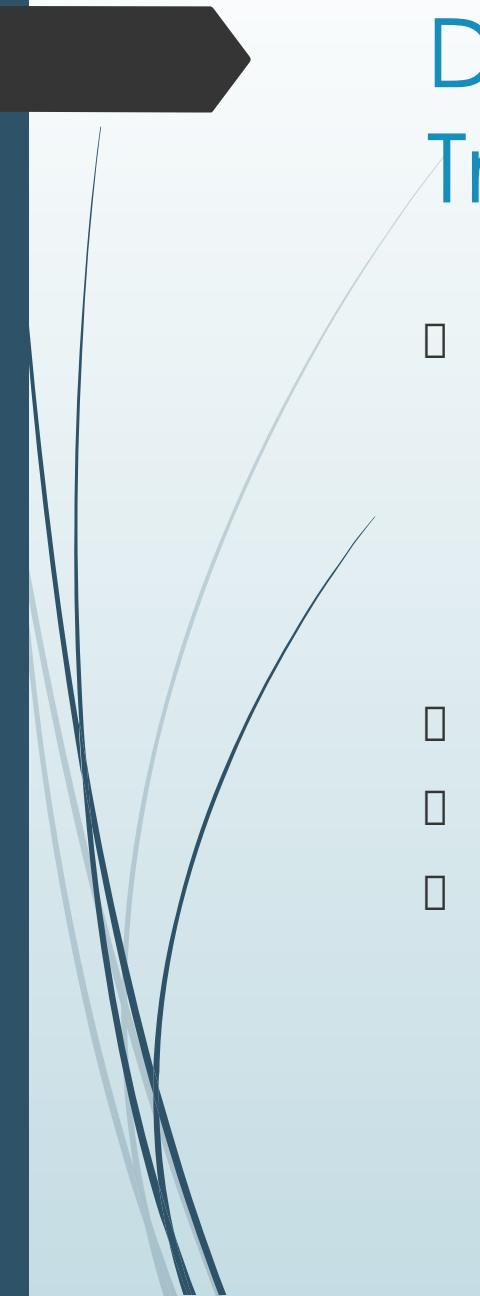
- Incorporates multiple core processor on the same chip
- Works on the same clock frequency supplied to the chip
- Based on the number of cores they are known as dual core, tri core, quad core etc.
- Implements multiprocessing
- Each core implements pipe lining, multithreading and super scalar execution.
- Eg: ARM Cortex-4 provides 4 symmetric multi core

# Reconfigurable processors

- It is a microcontroller or processor with reconfigurable hardware features.
- They contain an array of programming elements along with microprocessor
- PE can be computational engine or memory elements hardware features can be changed statically or dynamically changing of hardware configuration makes the chip adaptable to the firmware running on the processor
- SoC has to be configured to the required functionality through software support at the time of initialization.
- Eg: compressing mpeg file by media player based on usage

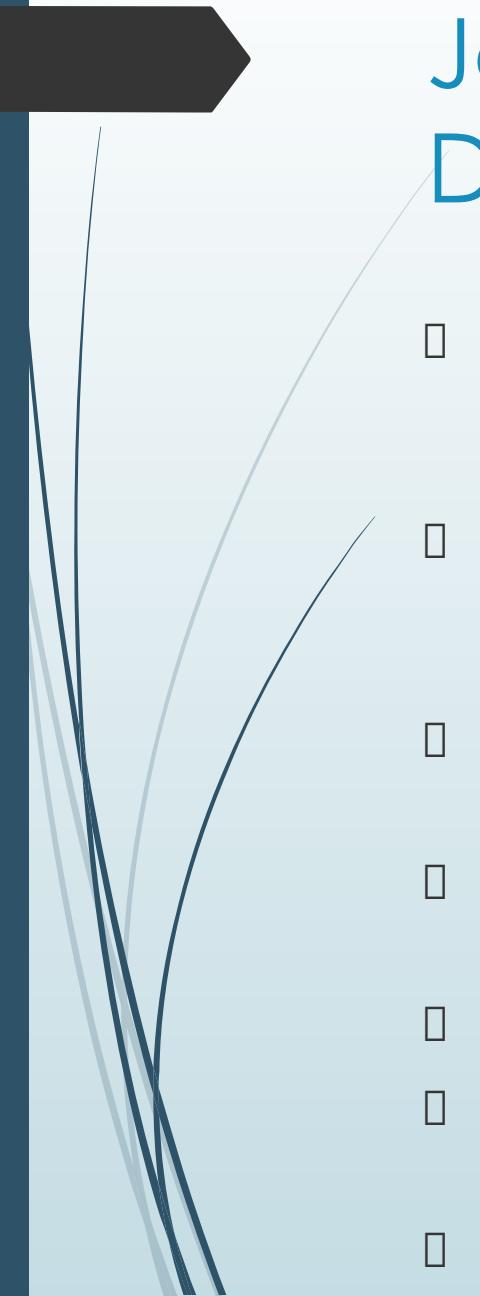
# Embedded OS trends

- Most embedded OS try to implement virtualization concepts
- They adopt micro kernel architectures where only essential features are placed in the kernel and remaining appears as services and placed in the user space.
- There are OS customized for the product.
  - Eg: MS Embedded OS
  - MS Windows Phone OS is designed for mobiles
  - VxWorks RTOS is multicore in nature



# Development Language Trends

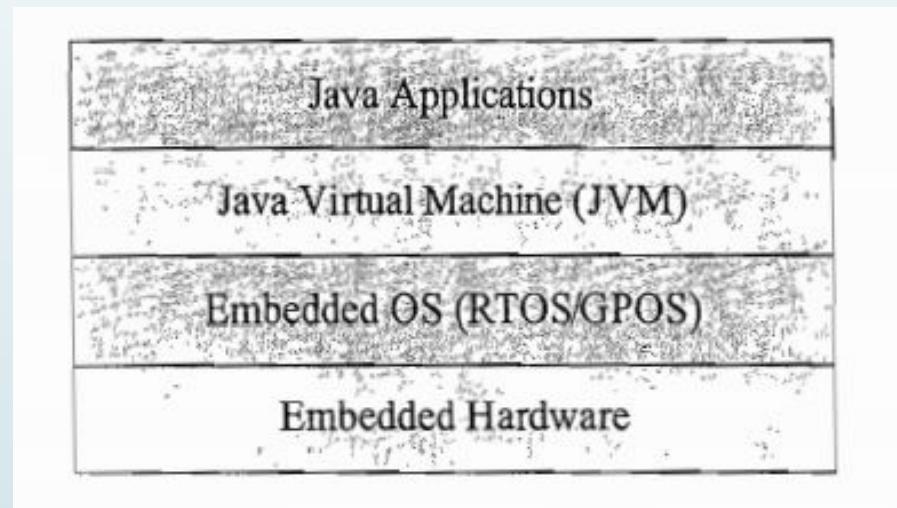
- Two types of embedded development takes place.
  - System wide application(eg: platform development)
    - In ES it is called as Embedded Firmware
  - User application development
    - In ES it is called Embedded Software
- Embedded firmware comes as installed inside the chip.
- Embedded software, user can install by themselves.
- The new type of languages used for ES development is Java and .NET



# Java for Embedded Development

- In java development, the java code is compiled by a java class compiler to platform independent code called java byte code
- The java bytecode is converted into processor specific object code by a utility called Java Virtual Machine (JVM)
- During execution the, the byte code is interpreted by the JVM for execution.
- JVM abstracts the processor dependency from Java applications
- The interpretation makes java application slower.
- Just In Time compiler speeds up the java program execution by catching all previously interpreted code.
- This avoids the delay in interpreting a bytecode which is not yet executed.

# Java based embedded application development

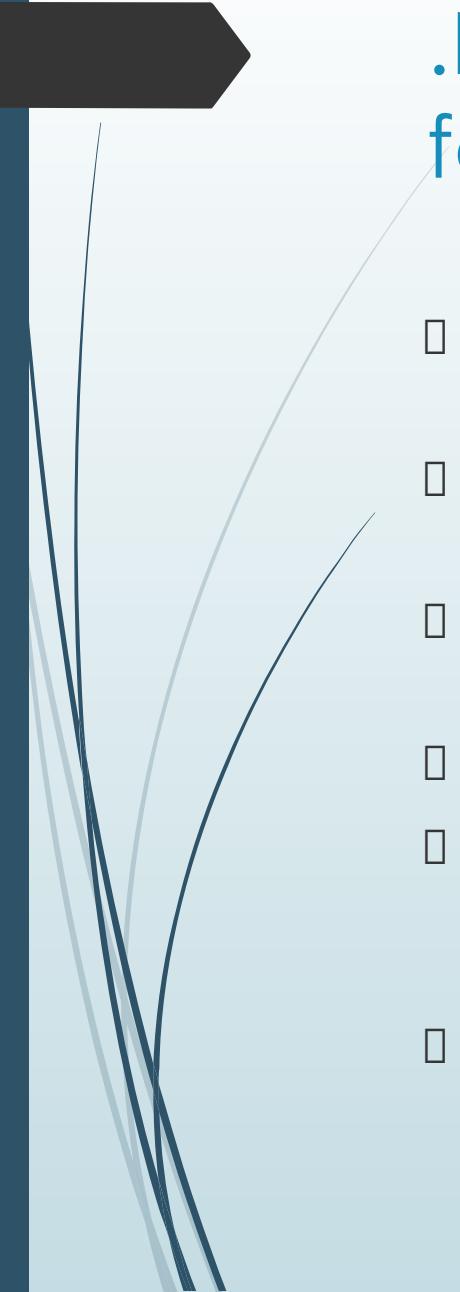


# Limitation of Java in Embedded Application Development

- The interpreted version of java is slower. It is not suitable for real time applications.
- Garbage collector of java is non deterministic. Hence its is not useful for hard real time system.
- Processor which doesn't have java built in support, requires the JVM ported for the processor architecture
- The resource access supported by Java is limited
- The runtime memory requirement of JVM and its libraries are more, hence ES which are constrained on memory cannot afford this.

- AOT(Ahead of Time)compiler
  - Converts java bytecode to target processor specific assembly code during compile time. Eliminates the need of JVM.
  - For garbage collection special piece of code is needed
- Java Native Interface (JNI)
  - Used for invoking the functions written in other languages
  - JNI feature can be explored for implementing system software

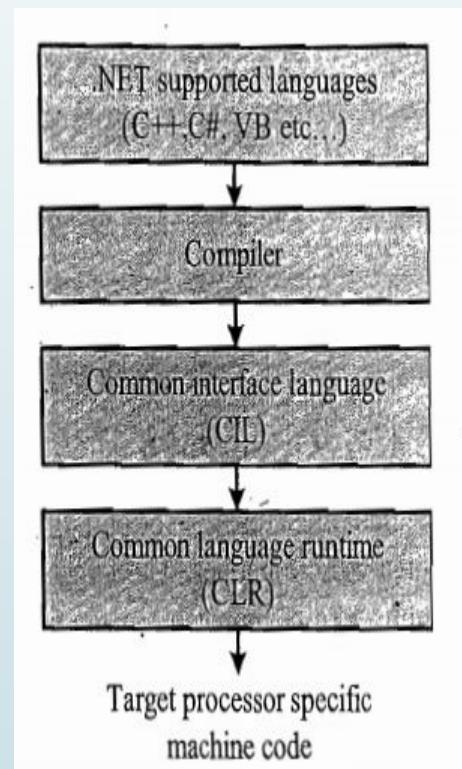
- Safety Critical Java Technology expert group has come up with real time version of java which is capable of providing compatible execution speed with C.
- J2ME is designed for Embedded Application Software
- It provides flexible user interface, built in protocols, security etc.
- Oracle provides Embedded version of Java Standard Edition with optimized memory requirements.

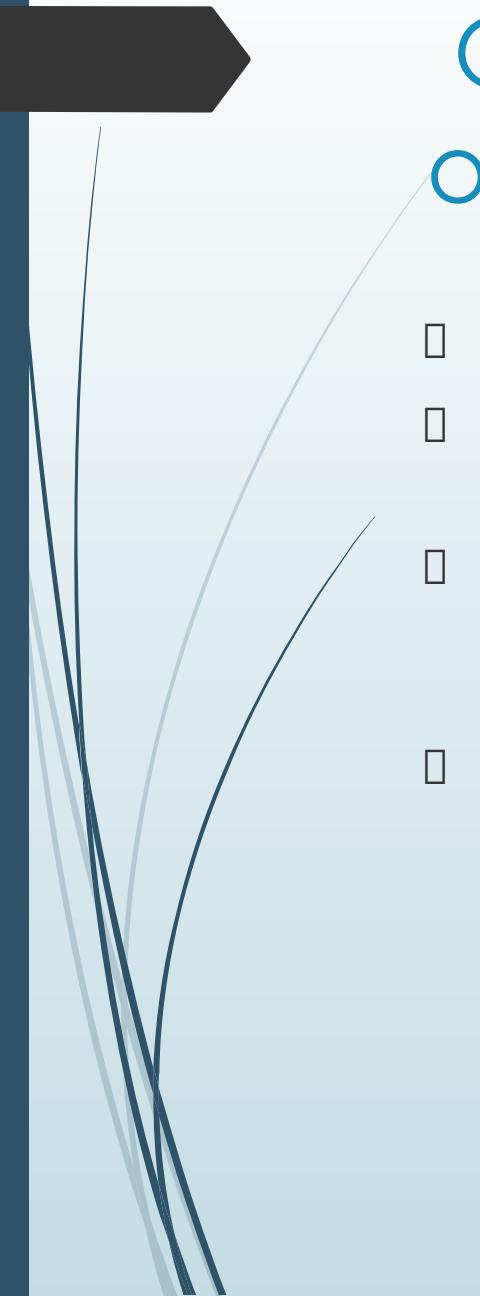


# .NET Compact/Micro Framework for Embedded Development

- .NET is a framework for application development for desktop OS from Microsoft
- It is a collection of pre-coded libraries and act as the run time environment for .NET supported languages.
- The runtime environment of .NET is known as Common Language Runtime (CLR)
- CLR works similar like JVM
- CLR compiles .NET programs into platform neutral intermediate language called Common Intermediate Language
- CIL is converted into target processor specific machine code by CLR

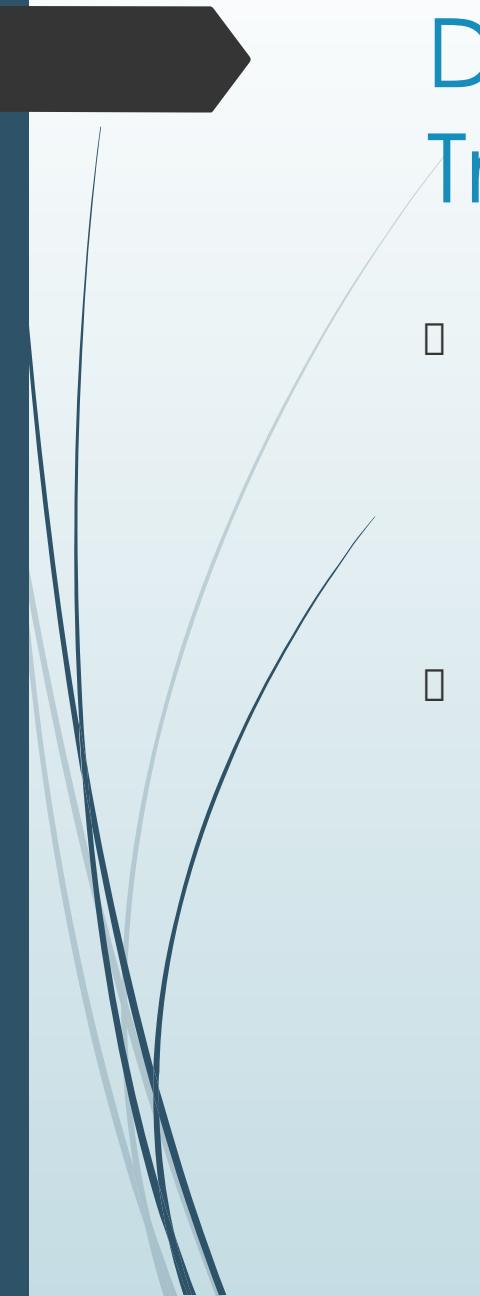
# .NET based Embedded Application Development





# Overcoming the drawbacks of .NET

- The .NET framework consumes large amount of memory
- A stripped down customised version of .NET framework called .NET Compact Framework helps for this.
- .NET Micro Framework is an open source platform that enables developers to build embedded products on resource constrained embedded devices.
- .NET MF doesn't need an OS to run



# Development Platform Trends

- Development platform is an integrated hardware platform which incorporates a processing unit, integrated peripherals, expansion ports for connectivity with other external peripherals, USB/RS-232/TCP Connectivity and debug interface for interfacing with host computer for firmware development.
- The main hardware development platform
  - ADRUINO
  - BeagleBone
  - Shark Cove
  - MinnowBoard MAX
  - RASPBERRY PI
  - Intel Galileo Gen2