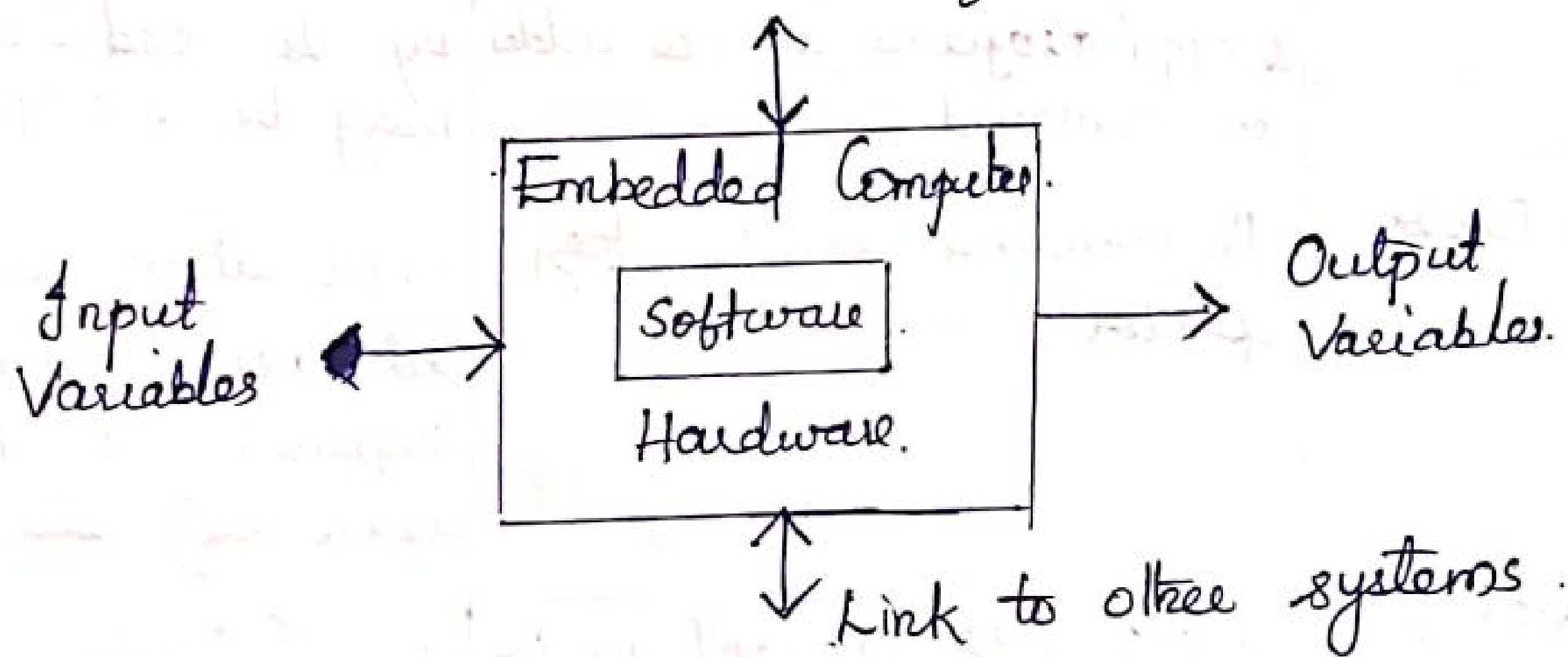


I. INTRODUCTION TO EMBEDDED SYSTEMS

EMBEDDED SYSTEM

- An embedded system is an electronic / electro-mechanical system designed to perform a specific function and is a combination of both hardware & firmware (software) ^(fig)
- Every embedded system is unique; specific to an application domain
- Embedded systems are used in fields like household appliances, telecommunications, medical equipments; industrial control, consumer products etc.



- One of its most important characteristics is, it gives the o/p within the time limits.
- Modern embedded systems are often based on microcontrollers (ie, CPUs with integrated memory or peripheral interfaces), but ordinary microprocessors (using external chips for memory and peripheral interface circuits) are also common, especially in more complex systems

EMBEDDED SYSTEM Vs GENERAL COMPUTING SYSTEMS

Criteria	General Purpose Computing System	Embedded Systems
Contents	A system which is a combination of a generic hardware and a General purpose OS for executing a variety of applications	A system which is a combination of special purpose hardware and embedded OS for executing a specific set of applications
OS	It contains a general purpose OS (GPOS)	It may or may not contain an OS for functioning
Alterations	Applications are alterable (programmable) by the user. (OS can be reinstalled & appl ⁿ programs can be added or removed...	The firmware of embedded system is pre-programmed and it is non-alterable by the end-user (There may be exceptions)
Key factor	Performance is the key factor.	Application specific requirements (like performance, power requirements, memory usage etc) are the key factors.
Power Consumption	More (provides options for power management)	Less (take advantage of the power saving modes supported by the hardware and OS)
Response time	Not critical	Highly critical (especially for mission critical systems)
Execution behaviour	Need not be deterministic <i>execution does not always depend on %</i>	Deterministic for certain types of embedded systems like "Hard Real Time" systems

HISTORY OF EMBEDDED SYSTEMS

- In the older days, ESs were built around old vacuum tube and transistor technologies & algos were written in low-level languages.
- Advances in semiconductor and nano-technology & IT revolution gave way to the development of miniature ESs.
- 1st recognized modern ES → Apollo Guidance Computer (AGC) developed by MIT Instrumentation Lab for the lunar expedition.
- 1st mass-produced ES was the guidance computer for the Minuteman-I missile in 1961. It was the 'Autonetics D-17' guidance computer.
- The first IC was produced in 1958 and was used by NASA for the Apollo guidance Computer and by the US Military in the Minuteman-II missile.

CLASSIFICATION OF EMBEDDED SYSTEM

- The criteria used in the classification of ESs are.
 - 1) Based on Generation
 - 2) " " Complexity & performance requirements
 - 3) " " Deterministic behaviour
 - 4) " " Triggering

1) Classification. Based on Generation

• First Generation :-

- built around 8 bit μ ps like 8085 & Z80 & 4 bit μ Cs.
- Simple hardware circuits with firmware developed in Assembly code.
- eg:- Digital telephone keypads, Stepper motor control units etc.

• Second Generation

- built around 16 bit μ ps and 8/16 bit μ Cs.
- The instruction set were much more complex than 1st Gen μ ps & μ Cs.
- Some of them used embedded OS for their operation.
- eg:- Data Acquisition Systems, SCADA systems.

• Third Generation

- built around 32 bit μ ps & 16 bit μ Cs as well as application & domain specific μ ps & μ Cs like DSP (Digital Signal Processing) and ASICs (Application Specific ICs).
- Instruction set became more complex & powerful.
- Concept of Instruction pipelining was also evolved.
- Dedicated embedded real time & general purpose OS entered into the embedded market.
- Used in areas like robotics, media, Industrial process control, networking etc.

• Fourth Generation

- built around system on chips (SOC), reconfigurable processors and multicore processors.
- provides high performance, tight integration & miniaturisation into the embedded device market
- uses high performance real time embedded OS for their functioning.
- eg for 4th Gen ESs → Smart phone devices, Mobile Internet Devices (MIDs) etc.

2) Classification Based on Complexity & Performance

• Small-scale ESs

- ESs which are simple in application needs & the performance requirements are not time critical
- eg. - Electronic toys.
- built around low performance & low cost 8/16 bit $\mu p/\mu c$.
- may or may not contain an OS for its functioning.

• Medium-scale ESs

- slightly complex in h/w & firmware requirements
- built around - medium performance, low cost 16/32 bit $\mu p/\mu c$ or DSPs.
- usually contain an embedded OS (either general purpose or real time OS) for functioning.

- Large - Scale Embedded systems / Complex systems
 - highly complex hardware & software requirements
 - employed in mission-critical applications demanding high performance.
 - built around high performance 32/64 bit RISC processors/controllers or Reconfigurable System On chip (RSoC) or multi-core processors & Programmable logic devices.
 - may contain multiple MPs/MCs and co-units/hardware accelerators for offloading the processing requirements from the main processor of the system
(eg:- decoding/encoding of media, cryptographic function implementation etc)
 - usually contains a high performance Real Time OS (RTOS) for task scheduling, prioritization and management.

3) Classification based on Deterministic Behaviour

- This classification is applicable for "Real time" systems
- The task execution behaviour can be $\begin{cases} \text{deterministic} \\ \text{Non-deterministic} \end{cases}$
- Based on execution behaviour, RTs are divided into two $\begin{cases} \text{Hard Real time sys} \\ \text{Soft Real time} \end{cases}$
 - it must absolutely hit every deadline.
eg:- Nuclear systems, some medical apps such as pacemakers, defense apps
 - It can miss some deadlines, but that will affect its performance.
eg:- sound sys in a computer, sensor readings etc.

4) Classification Based on Triggering

• ESs which are "Reactive" in nature can be based on triggering.

• Reactive systems can be

→ Event triggered

→ Time triggered

→ When an event happens, it is detected by some sensors, which then causes the attached CPU to get an interrupt. (ie it is a interrupt driven system)
↑ appl^y :-

executes one or more set of tasks according to a predetermined & set task schedule. (involves a single interrupt that is linked to the periodic overflow of timer. appl^y :- Industrial systems, automotive systems, household goods.

MAJOR APPLICATION AREAS OF EMBEDDED SYSTEMS

1. Consumer Electronics : Camcorders, cameras etc
2. Household appliances : TV, DVD players, washing m/c, fridge, microwave etc.
3. Home automation & security systems :- ACs, sprinklers, Intrusion detection alarms, CCTV cameras, fire alarms etc.
4. Automotive Industry : Anti-lock braking system (ABS), engine control, ignition systems, automatic navigation systems etc.
5. Telecom : cellular phones, telephone switches, handheld multimedia appl^s etc.
6. Computer Peripherals : Printers, scanners, fax machines etc.
7. Computer networking systems : N/w routers, switches, hubs, firewalls etc.
8. Healthcare : Scanners, EEG, ECG m/c ...
9. Measurement & Instrumentation : Digital multimeters, digital CROs, logic analysers etc.
10. Banking & Retail : ATM & currency counters, Point of Sale (POS)
11. Card Readers : Barcode, smart card readers, handheld devices
12. Wearable devices : Health & fitness trackers, smartphone screen extension for notification etc.
13. Cloud computing & IoT (Internet of Things)

PURPOSE OF EMBEDDED SYSTEMS

1) Data Collection / Storage / Representation

- ESs designed for the purpose of data collection performs acquisition of data from the external world.
- Data collection is done for storage, analysis, manipulation and transmission.
- Data - text, voice, image, video, electrical signals & any other measurable quantities
- can be analog or digital.
- ESs with analog data capturing technique - collects analog data directly.
- ESs with digital data - collects analog data, convert it to digital using ADC & collect that binary data. Digital data can be directly captured.

- The collected data may be
 - stored directly in the sym
 - transmitted to some other sym
 - processed by the sym
 - deleted instantly after giving a meaningful representation

eg:- Analog/Digital CROs without memory for storage.

- A Digital Camera is a typical example of an ES with data collection / storage / representation of data. Images are captured & stored in memory and are presented to the user through a graphic LCD unit.

2) Data Communication

- Embedded data commⁿ syms are deployed in applications from complex satellite communications to simple home networking systems
- The transmission of data is achieved either by a wire-line medium or by a wireless medium.
↓
(RS232C, USB, TCP/IP, PS2 etc) ^{Personal System} ↓ (Bluetooth, ZigBee, Wi-fi, EDGE, GPRS etc) ^(Cable TV)
- Data can ~~be~~ either be transmitted by analog means or by digital means.
- Network hubs, routers, switches etc are examples of dedicated data transmission ESs.

3) Data (Signal) Processing

- ESs with signal processing functionalities are employed in applications demanding signal-processing like ^{optimized reprⁿ of speech signal} speech coding, ^{up/down (P → speech)} synthesis, ^{ckt or str that compresses digital audio/video} audio video codec, transmission applications etc.
- A Digital Hearing Aid is an example for ES employing data processing.

4) Monitoring

- All embedded products coming under the medical domain are with monitoring functions.
- They are used for determining the state of some variables using input sensors
- Electro cardiogram machine (ECG) is intended to monitor the heartbeat of patient, but it cannot control the heartbeat
- Other examples are Digital CRO, ^{Digital multimeter} Logic Analyzers

5) Control

- ESs with control functionalities impose control over some variables to the changes in input variables
- Such systems contain both sensors & actuators.
 - Sensors → are connected to Yp port for capturing/measuring the changes in environmental variables
 - actuators → connected to op port and are controlled according to the changes in Yp variable to make an impact on the controlling variable.
- eg:- Air conditioner system.
 - Yp variable → ^{current} room temperature
 - Controlled variable → room temperature
 - Controlling variable → cool air flow by the compressor unit.

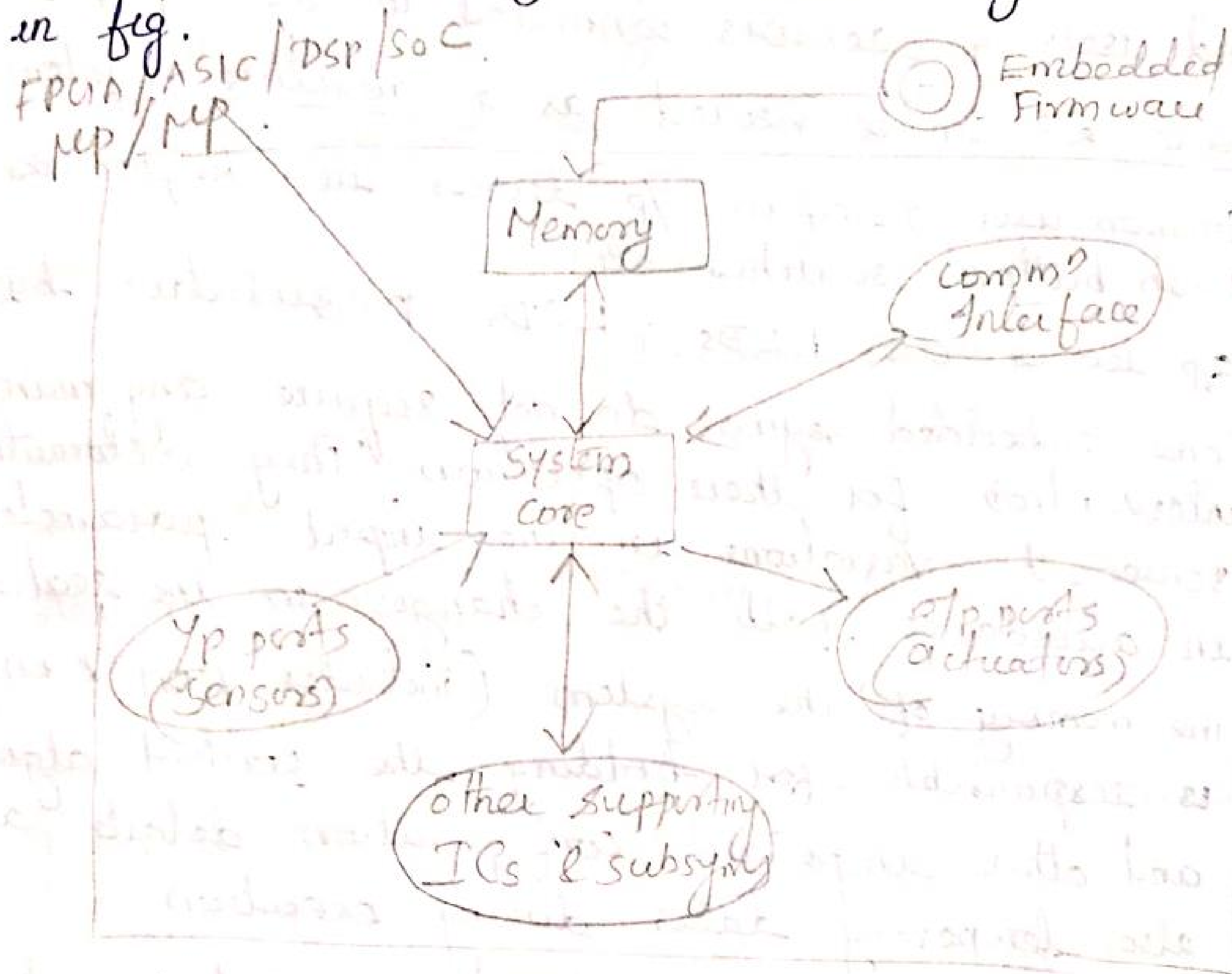
6) Application Specific User Interface

- These are ESs with application-specific user interfaces like Buttons, switches, keypad, lights, bells, display units etc.
- Mobile phone is an eg, in which user interface is provided through the keypad, graphic LCD module, system speaker, vibration alert etc.

Case Study :- "Smart Running Shoes from Adidas"
"Fitness Watch"

II. THE TYPICAL EMBEDDED SYSTEM

- An ES is an electronic / electro-mechanical system designed to perform a specific function and is a combination of both hardware & firmware (software).
- The elements of a typical embedded system is shown in fig.



- A typical ES contains a single chip controller that acts as the master brain of the system. It can be.

- Microprocessor (eg 8085, 8086 ...)
- Microcontroller (Atmel AT 89C51 ...)
- FPGA - Field Programmable Gate Array (Xilinx, Spartan)
- DSP - Digital Signal Processor (eg:- Blackfin)
- Application Specific Integrated Circuits (ASICs)
- Programmable Logic Devices (PLDs) etc

- Embedded h/w / s/w systems ~~can~~ be are basically designed to regulate a physical variable or to manipulate the state of some devices by sending some control signals to the Actuators or devices, connected to the o/p ports of the system, in response to the i/p signals provided by the end users or sensors connected to the i/p port.
- Hence ES can be viewed as a reactive system
- Common user Interface i/p devices are keyboards, push button switches etc.
o/p devices are LEDs, LCDs, piezoelectric buzzers etc.
- Some embedded systems do not require any manual intervention for their operation. They automatically sense the variations in the input parameters in accordance with the changes in the real world.
- The memory of the system (includes ROM & RAM) is responsible for holding the control algorithm and other important configuration details and also temporary data during execution.
- An ES without a control algo is not capable of making decisions depending on the situational as well as real world changes.
- The control algo. can be stored in an on-chip memory or in an off-chip memory depending on the application.

The main elements of the embedded system are

- 1) Core of the embedded system.
- 2) Memory
- 3) Sensors and Actuators
- 4) Communication Interface
- 5) Embedded Firmware
- 6) Other system Components;
- 7) PCB and Passive Components

1) CORE OF THE EMBEDDED SYSTEM

- ESS are domain & application specific & are built around a central core. The core of ES can be any one of the following.

a) General Purpose & Domain Specific Processors

- μP
- μC
- DSPs.

b) ASICs (Application specific Integrated Circuits)

c) Programmable Logic Devices (PLDs)

d) Commercial off-the-shelf components (COTS).

a) General Purpose & Domain specific processors

- 80% of the ESS are processors/controller based (μP , μC , DSPs)
- Most of the ESS in the Industrial control & monitoring applications uses the commonly available μP s & μC s.
- Domains which require signal processing such as speech coding, speech recognition etc make use of DSPs.

→ Microprocessors

- or CPU, capable of performing arithmetic & logical operations.
- CPU contains ALU + CU + working registers.
- It needs to be interfaced with memory, timer unit, interrupt controller etc. for proper functioning.
- Early μ ps :- Intel 4004, Intel 8008, Intel 8080, Motorola 6800, Intel 8085, Zilog's Z80 processor.
- Technical advances in the field of semiconductor industry brought new dimensions to the μ p market; 16, 32, 64 bit μ ps came into the place of 8 bit μ ps.
- Processor Manufacturers :- Intel, AMD, Freescale, Global foundries, TI, Cyrix, NVIDIA, Qualcomm etc.
- Different instruction set & sym architecture are available for the design of μ p.
eg:- Harvard & Non-Neumann system architecture for processor design.

CISC & RISC instruction set architecture.

→ General Purpose Processor (GPP) vs. Application-Specific Instruction Set Processor (ASIP)

GPP

- processor designed for general computational tasks.
- eg:- processor in a laptop/desktop.
- produced in large volumes.
- Thus, ^{per unit} cost of a chip is low.
- contains ALU + CU.

ASIP

- processors with architectural and instruction set optimized to specific domain/application requirements.
- They fill the architectural spectrum b/w GPP and ASICs.
- Its need arises when the traditional GPP are unable to meet the increasing application needs.
- eg:- Some μ cs, SoC, DSPs etc.

→ Microcontrollers

- A highly integrated chip that contains a CPU, RAM, special & general purpose register arrays, on chip ROM/FLASH memory for program storage, timer & Interrupt control units and dedicated I/O ~~ports~~ ports.
- widely used in ES (they are cheap, cost effective and readily available in the market)

- eg:- Texas Instruments TMS 1000, Intel MCS-48 family, Intel ^{MCS-}8051 family (8051), Microchip Technologies PIC family, ARM μ c; Atmel AVR family.

Peripheral Interface Controller

- suppliers:- Infineon, Freescale, Philips, Atmel, Maxim, Microchip, ~~etc~~ Intel, Maxim, Texas Instruments, Sharp, Hitachi etc.

- Atmel manufactures a variety of Flash memory based μ cs, which helps in fast reprogramming of the chip and thereby reduces the product development time

- The instruction set architecture of μ c can be RISC or CISC.

- General Purpose μ c (eg:- 8051 from Intel)

- Application specific Instruction set Processors based μ c (eg:- Atmel AVR μ c family) → for automotive domain

→ Microprocessors Vs Microcontrollers

μ p

- A silicon chip representing a ~~car~~ CPU, which is capable of performing arithmetic & logical operations according to a pre-defined set of instructions.

μ c

- A highly integrated chip that contains a CPU, RAM, special & general purpose reg. arrays, on chip ROM/FLASH memory for program storage, timer & Interrupt control units & I/O ports.

MP

- It is a dependent unit. It requires the combination of other chips like timer, program & data memory chips, interrupt controllers etc for functioning.
- Most of the time general purpose in design & operation.
- doesn't contain a built-in I/O port. (uses 8255 PPI)

• Targeted for high end market where performance is important.

• Limited power saving options compared to MCs.

MC

- It is a self contained unit and it doesn't require external interrupt controller, timer, UART etc for its functioning.

• Mostly application-oriented or domain-specific.

• Contains multiple built-in I/O ports which can be operated as a single 8/16/32 bit port or as individual port pins.

• Targeted for embedded market where performance is not so critical.

• Includes a lot of power saving features.

→ Digital Signal Processors (DSPs)

— powerful special purpose 8/16/32 bit MPs designed specifically to meet the computational demands & power constraints of today's embedded audio, video and comm' applications.

— 2 to 3 times faster than GPP in signal processing applications. bcs

• DSPs implement algos in h/w which speed up the execution

(GPP is f/w & speed of execⁿ depends on clock)

Specific h/w other than prog
eg: - Graphics Card

- Key units of DSP are.

Program memory :- stores prog required by DSP to process the data

Data memory :- Working mem. for storing temporary variables & data/signal to be processed.

Computational Engine :- Performs signal processing based on prog in prog memory. It contains many specialized AUs that can run parallelly & also multiple hardware shifters.

I/O unit

:- Act as an interface b/w the outside world & DSP.

- Some of the operations performed by DSP - Sum of Products (SOP) calculation, Convolution, Fast Fourier Transform (FFT), Discrete Fourier Transform (DFT) etc.
- Application areas - Audio video signal processing, telecomm & multimedia applications.
- Egs :- Blackfin processors from Analog Devices.

→ RISC Vs. CISC Processors / Controllers

- | <u>RISC</u> | <u>CISC</u> |
|--|--|
| • Reduced Instruction set Computing | • Complex Instruction set Computing. |
| • Lesser no. of instructions | • Greater no. of instructions |
| • Instruction pipelining & increased execution speed | • Generally no instruction pipelining feature. |
| • Orthogonal instruction set.
(Any instruction can use any reg & addressing mode) | • Non-orthogonal Instruction set. |

RISC

- Operations are performed on registers only, the only memory operations are load & store
- A large no. of regs are available.
- Programmer needs to write more code to execute a task since the instructions are simpler ones.

• Single, fixed length instructions

• Less silicon usage & power

• With Harvard Architecture

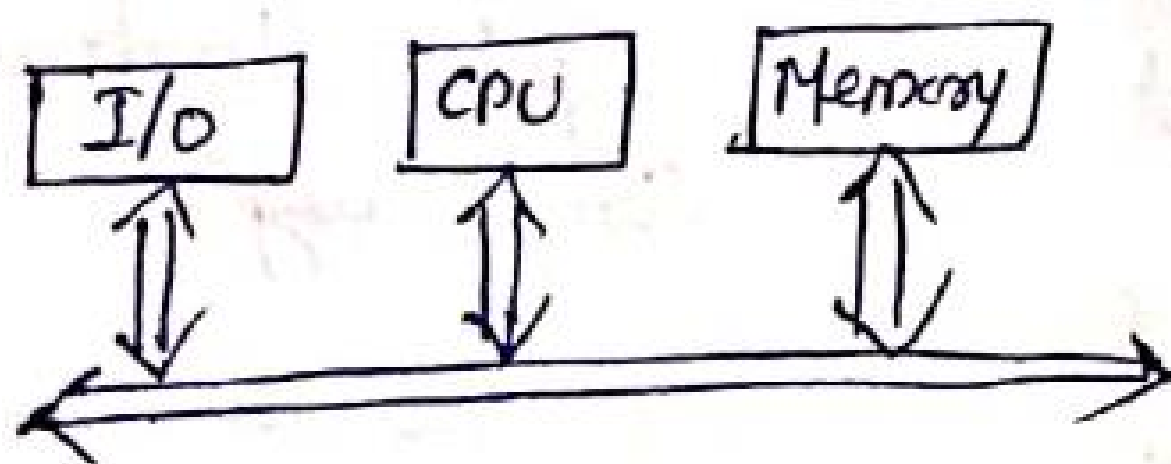
CISC

- Operations are performed on registers or memory depending on the instruction.
- Limited no. of general purpose registers.
- Instructions are like macros in C language. Desired functionality can be achieved with a single instruction.
- Variable length instructions
- More silicon usage since more additional decoder logic is required to implement the complex instruction decoding.
- Harvard or Von-Neumann architecture.

→ Harvard Vs Von-Neumann MP/MC architecture

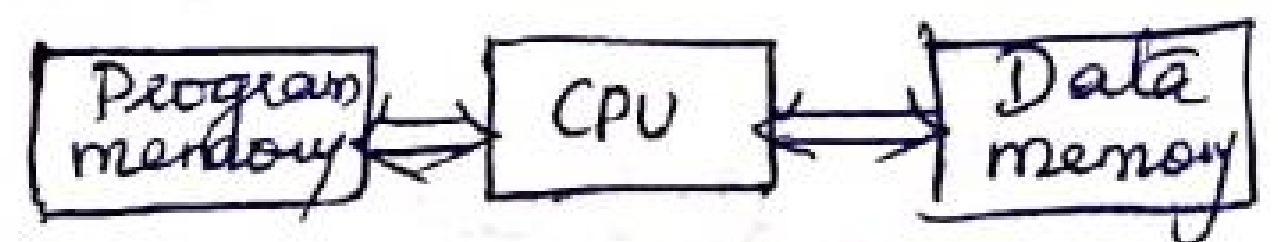
- These are processor architecture designs.

Von-Neumann



- Single shared bus for instruction & data fetching
- low performance compared to Harvard archi.

Harvard



- separate buses for instruction and data fetching.
- Easier to pipeline, so high performance can be achieved.

Von-Neumann

- Cheaper ^{the code that allows its own instructions to be modified}
- Allows self modifying codes ^{executing}
- chances for accidental corruption of prog. memory (bcs of shared prog/data mem)

Harvard

- Comparatively high cost
- No memory alignment problems
- No chances for accidental corruption of prog. memory

→ Big-Endian Vs Little-Endian ^{µps/MCs}

- Endianness specifies the order in which the data is ~~stored~~ stored in the memory by the processor.
- Little-Endian :- lower order byte of data is stored in memory at the lowest address & the higher-order byte at the highest address. (i.e., little-end comes first)

eg:- Suppose the data is

Byte 3 Byte 2 Byte 1 Byte 0

and will be stored in memory as

Byte 0	Base addr + 0
Byte 1	" " + 1
Byte 2	" " + 2
Byte 3	" " + 3

- Big-Endian :- Big end comes first.

Byte 3	Base addr + 0
Byte 2	" " + 1
Byte 1	" " + 2
Byte 0	" " + 3

→ Load Store Operation & Instruction Pipelining

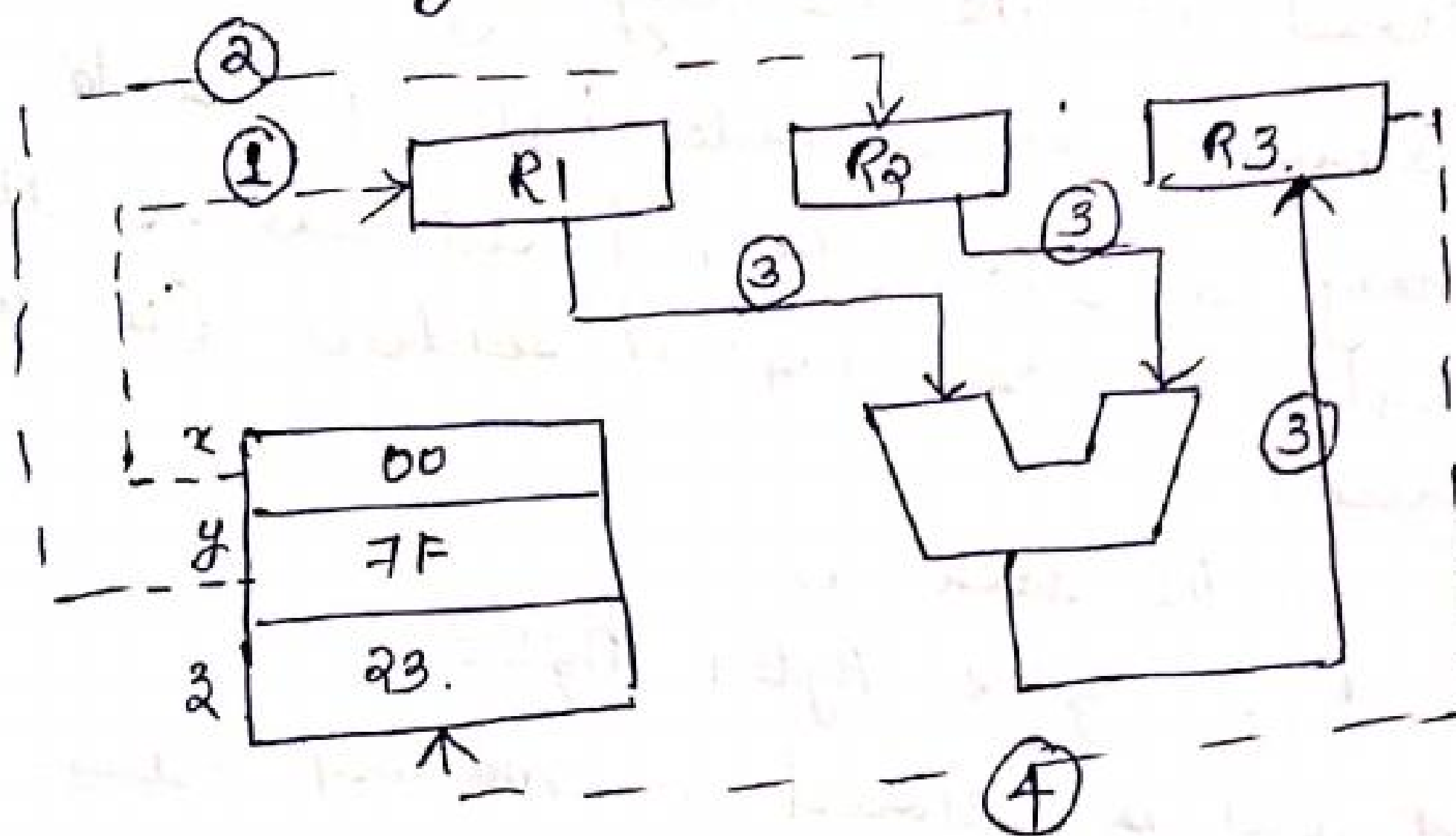
- Load & store operation

Load instruction → the content of a memory location is loaded to a register.

Store " → stores data from a specified reg. to a specified mem. locⁿ.

- The Concept of load & store architecture

eg:- Suppose $x, y, z \rightarrow$ mem. locⁿs, we want to add content of x & y & store the result in z .



load R1, x — ①
load R2, y — ②
add R3, R1, R2 — ③
store R3, z — ④

- Conventional prog execution - fetch & decode, ^{execute.} operations are done in sequence.

- Instruction Pipelining :- A technique for implementing instruction level parallelism within a single processor. It increases the instruction throughput (the no. of instructions that can be executed in a unit of time)

eg:- Suppose instruction execution involves 3 steps - fetch, decode, execute.

Instructions	F1	D1	E1	
		F2	D2	E2
			F3	D3

b) Application Specific Integrated Circuits (ASICs)

- ASICs is a microchip designed to perform a specific and unique applications.
- used as a replacement to conventional general purpose logic chips
- It integrates several functions into a single chip and thereby reduces the system development cost.
- Most of the ASICs are proprietary (which having some trade name) products, it is referred as Application Specific Standard Products (ASSP)

- As a single chip, ASIC consumes a very small area in the total system, thereby helps in the design of smaller system with high capabilities or functionalities
- The developers of such chips may not be interested in revealing the internal details of it.

c) Programmable Logic Devices (PLDs)

- A PLD is an electronic component used to build digital circuits which are reconfigurable (using software)
- A logic gate has a fixed function but a PLD does not have a defined function at the time of manufacturing.
- PLDs offer customers a wide range of logic capacity, features, speed, voltage characteristics

- PLDs can be reconfigured to perform any number of functions at any time.
- A variety of tools are available for the designers of PLDs which are inexpensive and help to develop, simulate and test the designs.
- 2 major types of PLDs.

1) CPLD (Complex Programmable Logic Device)

- offer much smaller amount of logic upto 1000 gates.
- but offer very predictable timing characteristics therefore ideal for critical control applications.
- eg:- Xilinx CoolRunner. (low power; inexpensive; battery operated, portable etc)

2) FPGAs (Field Programmable Gate Arrays)

- offer the highest amount of logic density, the most features & the highest performance.
- eg:- Xilinx Virtex (8 million sym gates, built-in hardwired processors, large memory, clk mgmt syms etc).
- Appl^{ns} :- data processing & storage, instrumentation, telecommunication, digital signal processing.

→ Advantages of PLDs

- 1) PLDs offer customer much more flexibility during the design cycle.

2) PLDs do not require long lead times for prototypes or production parts because PLDs are already on a distributors shelf & ready for shipment.

3) PLDs can be reprogrammed even after a piece of equipment is shipped to the customer. (upgraded thru internet)

d) Commercial Off-the-shelf Components (COTS)

→ A COTS product is one which is used 'as-is'.

→ The COTS components itself may be developed around a general purpose or domain specific processor or an ASIC or a PLDs.

→ Example COTS hardware units are -

- Remote controlled toy car control units
- High performance, high frequency microwave electronics
- high bandwidth ADCs
- UV/IR detectors etc

→ Advantages of COTS

- Ready to use
- Easy to integrate
- Reduces development time

→ Disadvantages of COTS

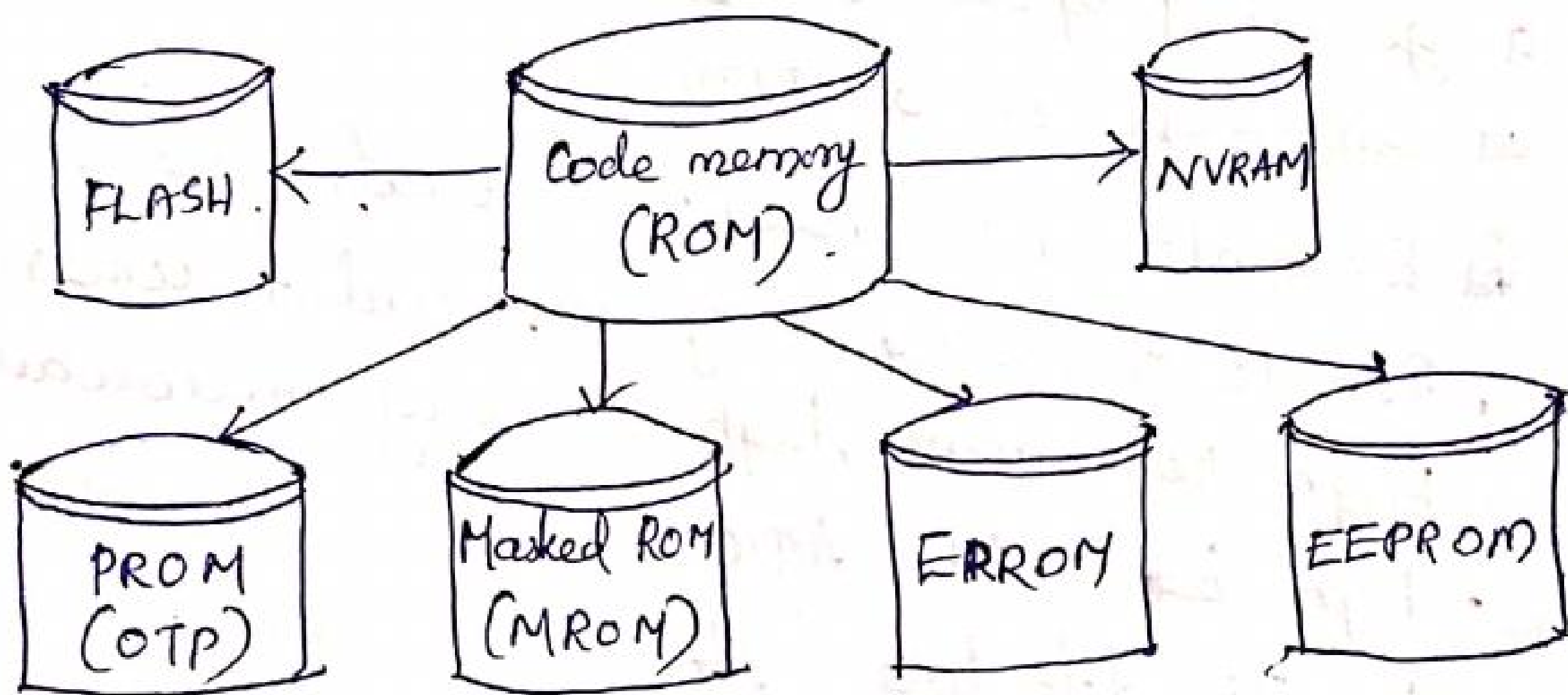
- No operational or manufacturing standard (all proprietary)
- Vendor or manufacturer may discontinue production of a particular COTS product at any time.

2. MEMORY

- Memory is an important part of ES. It can be
 - on-chip memory
 - off-chip "
- The memory is responsible for holding the control algo. and other important configuration details and also temporary data during execution.

a) Program Storage Memory (ROM)

- stores program instructions, non volatile storage.
- classification is shown below.



Masked ROM (MROM)

- one time programmable device (factory programmed)
- uses hardwired technology for storing data
- adv - low cost for high volume production.
- least expensive type of solid state memory.
- used for low cost ES
- disadv - firmware cannot be upgraded.

Programmable Read Only Memory (PROM) / One time Programmable (OTP)

- not factory programmed, end user can program it.
- contains nichrome or polysilicon wires (fuses) arranged in matrix

- Programming the PROM means burning the required fuses. (burned fuses - 0, not burned - 1)
- used for commercial production of ES whose prototype version are proven and the code is finalized.
- low cost
- cannot be reprogrammed.

Erasable PROM (EPROM)

- gives the flexibility to reprogram the same chip.
- It stores the bit information by charging the floating gate of an FET. (Field Effect Transistor)
- It contains a quartz crystal window for erasing the stored info. This window has to be exposed to ultraviolet rays for a fixed duration to erase all the contents. It is a tedious & time consuming process. (20-30 min)

Electrically Erasable PROM (EEPROM)

- the information can be altered by using electrical signals at the register / Byte level.
- They can be erased & reprogrammed in-circuit.
- It takes a few milliseconds to erase the contents.
- disadv - capacity is limited (A few KB)

FLASH

- latest & most popular technology used in ES.
- variation of EEPROM is reprogrammability of EEPROM with high capacity of standard ROMs.
- This memory is arranged as sectors / pages & erasing of memory can be done at sector level or page level.

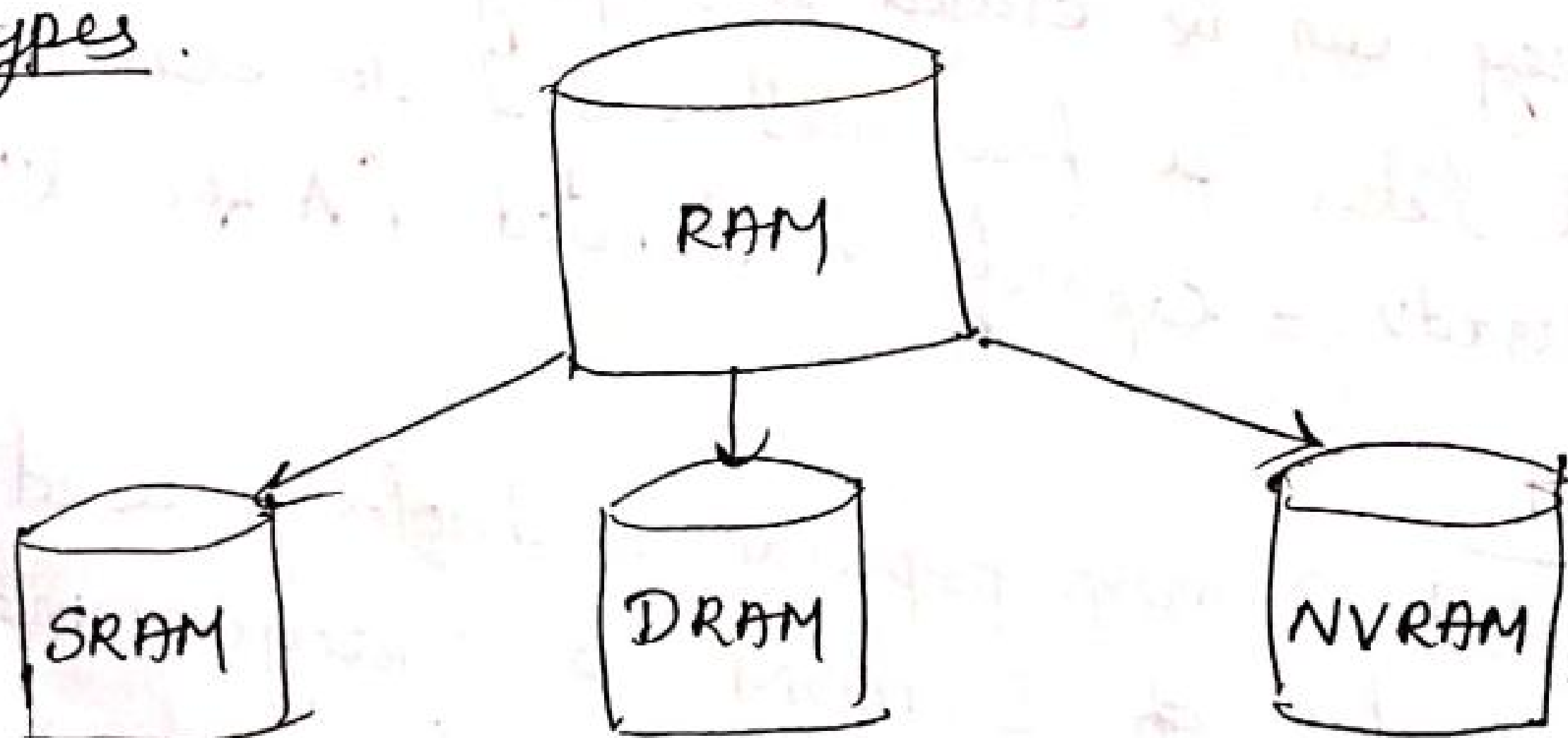
- Each sector/page should be erased before re-programming.
- typical erasable capacity is 1000 cycles.
- eg of 64 KB FLASH - W27C512 from WINBOND.

NVRAM (Non Volatile RAM)

- It's a RAM with battery backup.
- contains a static RAM based memory & a minute battery for providing supply to the memory in the absence of external power supply.
- lifespan → around 10 years.
- eg of 32 KB NVRAM - DS1644 from Maxim/Dallas.

b) Random Access Memory (RAM) / Read-Write Memory

- It is the data memory or working memory of $\mu p / \mu c$.
- volatile (content get deleted when power is turned off).
- direct access (ie, we can access the desired location without traversing the entire location preceded it).
- types.



SRAM (Static RAM)

- stores data in the form of voltage.
- made up of flipflops.

fastest form SRAM

- Static RAM
- stores data in the form of voltage
- made up of flipflops
- made up of 6 CMOS transistors (MOSFET)
- Doesn't require refreshing
- Low capacity (less dense)
- More expensive
- Fast in operation. Typical access time is 10 ns.

NVRAM

DRAM

- Dynamic RAM
- stores data in the form of charge.
- made up of MOS transistor gates
- made up of a MOSFET & a capacitor.
- Requires refreshing.
- High capacity (Highly dense)
- Less expensive
- slow in operation due to refresh requirements. Typical access time is 60 ns. Write operation is faster than read op.

c) Memory According to the Type of Interface

→ Parallel Interface.

→ Serial Interface.

→ SPI (Serial Peripheral Interface) (How memory is interfaced to $\mu p / \mu c$)

→ Single wire interconnection.

d) Memory shadowing

→ A technique used to increase a computer's speed by using high speed RAM memory in place of slower ROM memory (RAM is about 3 times faster than ROM)

→ On PCs, all code to control hardware devices such as keyboards, is normally executed in a special ROM chip called BIOS ROM.

→ However this chip is slower than the general purpose RAM that comprises main memory. Many PC manufacturers therefore configure their PCs to copy the BIOS code into RAM when the computer boots. The RAM used to hold the BIOS code is called the shadow RAM.

Assignment
1

c) Memory Selection for ES

3. SENSORS & ACTUATORS

Sensor

- used for taking input ^{degree that connects} physical quantity to electrical signals or vice versa
- It is a transducer device that converts energy from one form to another for any measurement or control purpose.
- eg:- A temperature sensor, Keyboard etc.

Actuator

- used for output
- It is a transducer that may be either mechanical or electrical which converts signals to corresponding physical actions
- eg:- LED, stepper motor etc.

Assign 1

Write short notes on the following sensors / actuators used in ES.

- 1) Light Emitting Diode (LED)
- 2) 7 segment LED display
- 3) Optocoupler
- 4) Stepper Motor
- 5) Relay
- 6) Piezo Buzzer
- 7) Push Button Switch
- 8) Keyboard
- 9) Programmable Peripheral Interface (PPi)

H. COMMUNICATION INTERFACE

→ It is essential for communicating with various subsystems of the ES & with the external world.

→ Broadly classified into 2

- a) Onboard Communication Interfaces
- b) External Com " "

a) Onboard CI

→ These are used for internal comm' of the ES. i.e., comm' b/w diff^{nt} components present on the systems.

- egs :-
- Inter-Integrated Circuit (I²C)
 - Serial Peripheral Interface (SPI)
 - Universal Asynch. Receiver Transmitter (UART)
 - 1-wire Interface
 - Parallel Interface

→ I²C

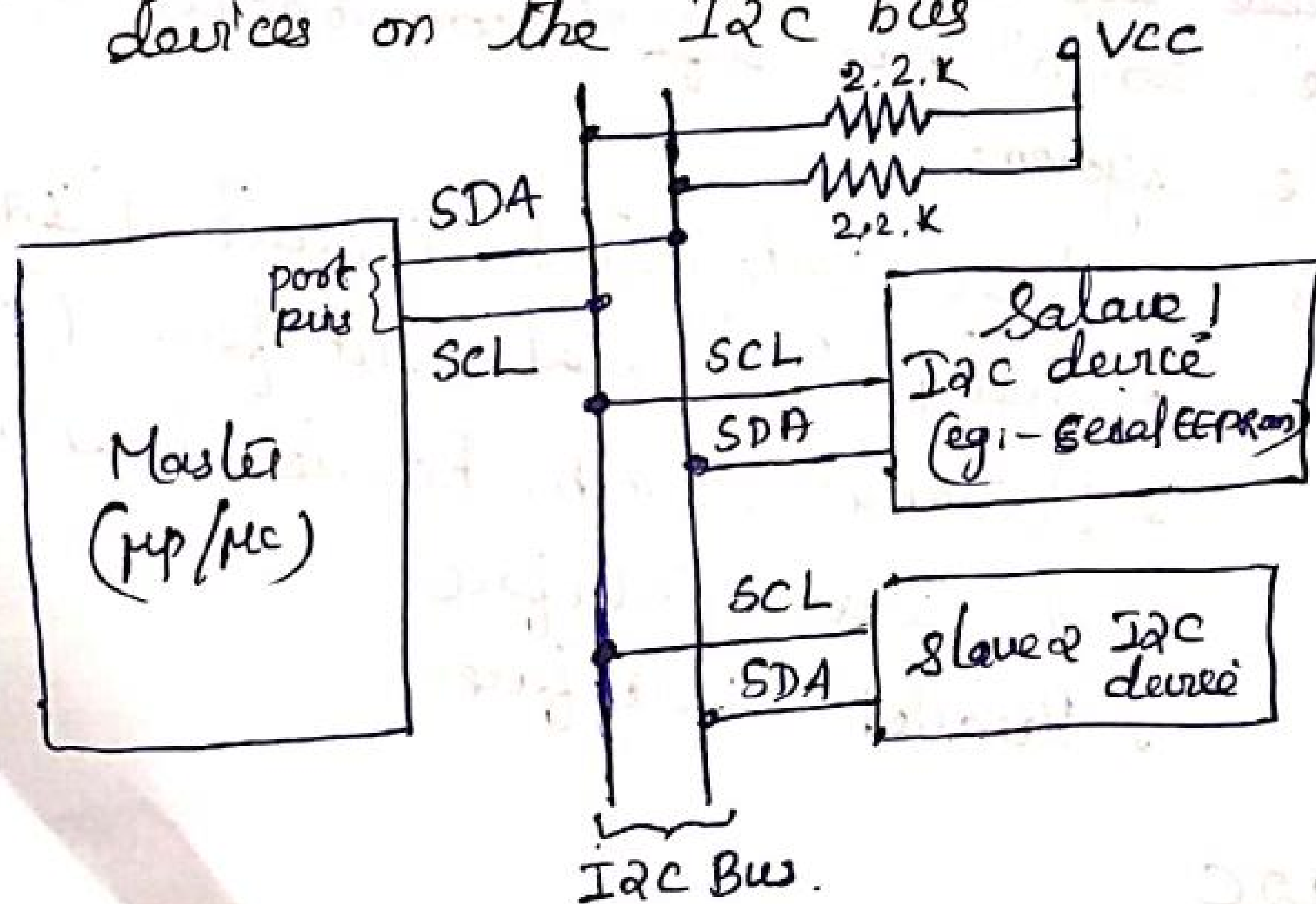
- It is synchronous.
- Bidirectional, half duplex, 2 wire serial Interface bus
- developed by Philips semiconductors in 1982
- It comprises of 2 buses
 - 1) serial clock — SCL
 - 2) serial Data — SDA
- SCL generates synchronization clock pulses
- SDA transmits data serially across devices.

- I²C is a shared bus system to which many devices can be connected
- Devices connected by I²C can act as either master or slave.

master - responsible for controlling commⁿ by initiating / terminating data transfer

slave - wait for commands from the master and respond to those commands.

- fig. shows the connection of master & slave devices on the I²C bus



④ . Operation

1. The master device pulls the clock line SCL of the bus to "HIGH", & pulls the data line SDA 'LOW'. (This is the start condition for data transfer)
2. The master device sends the address (7/10 bit) of the slave to which it wants to communicate over the ~~SDA~~ line. SCL line is used for synchronization.

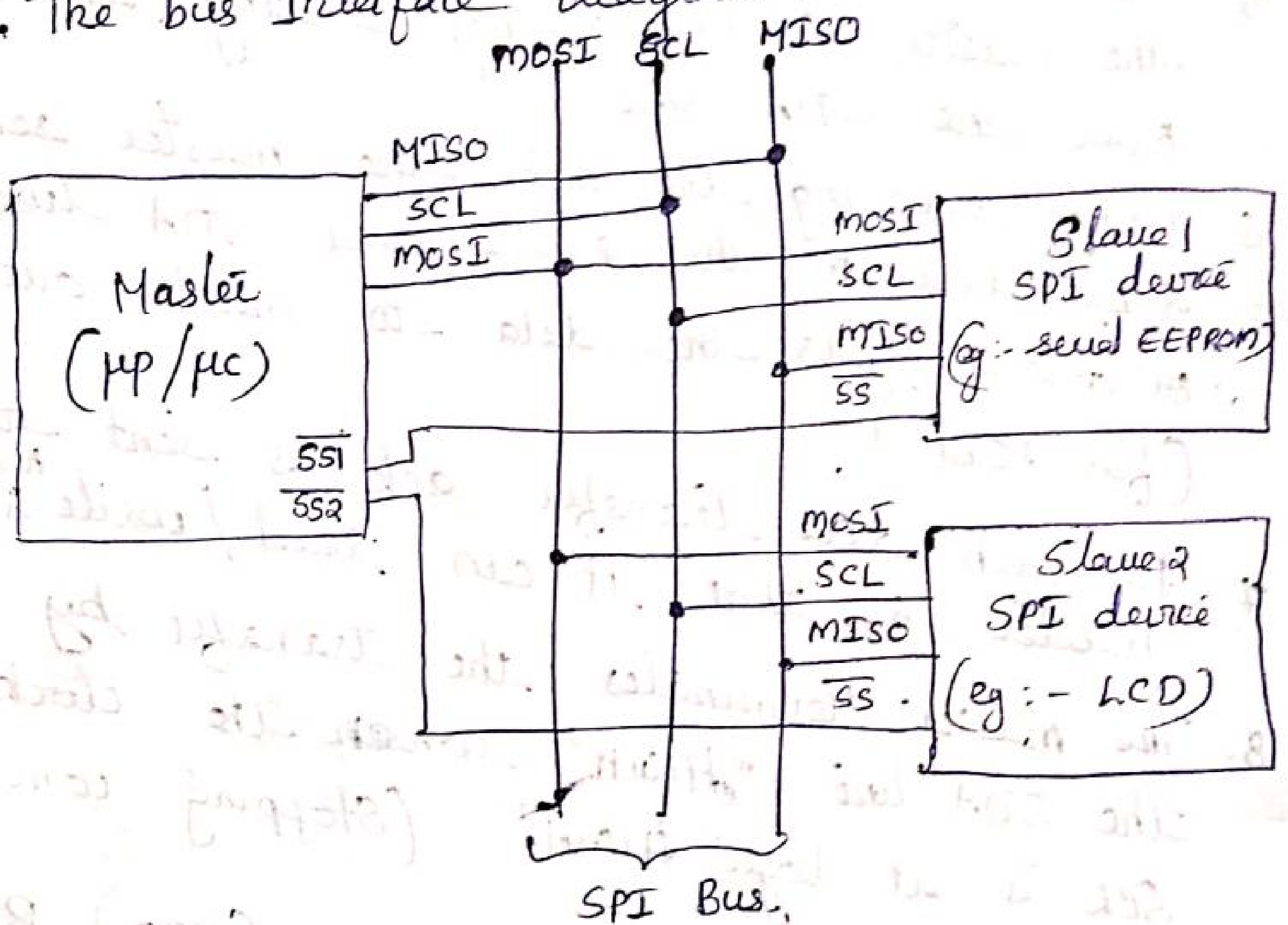
3. The master device sends Read⁽¹⁾ or Write bit (0), according to the requirement
4. Master waits for the ack. from slave
5. The slave device with the address requested by the master responds by sending an ack bit (1) over the SDA line
6. Upon receiving the ack, the master sends the 8 bit data to the slave over SDA line (for write) or slave sends the data to master over SDA line (for Read).
7. For each data transfer, ack is sent to the master, so that it can read/write ^{next} data.
8. The master terminates the transfer by pulling the SDA line "HIGH" when the clock line Sch is at logic "HIGH" (Stopping condition).

→ Serial Peripheral Interface (SPI) Bus

- It is a synchronous bi-directional full duplex 4-wire serial Interface Bus.
- Single master multi slave system. More than one masters are possible, but only one will be active at a time.
- SPI requires 4 signal lines for comm.
 - i) Master Out Slave In (MOSI) or Slave I/P / Slave Data In (SI/SDI)
 - data line from master to slave
 - ii) Master In Slave Out (MISO) or Slave Output / Slave Data Out (SO/SDO)
 - data line from slave to master

- 3) Serial clock (SCL) - line carrying the clk signal.
- 4) Slave Select (SS) - line for slave device select.
(Active low signal)

• The bus Interface diagram.



- adv - most suitable for applications requiring transfer of data in streams
- disadv - doesn't support an acknowledgement mechanism.

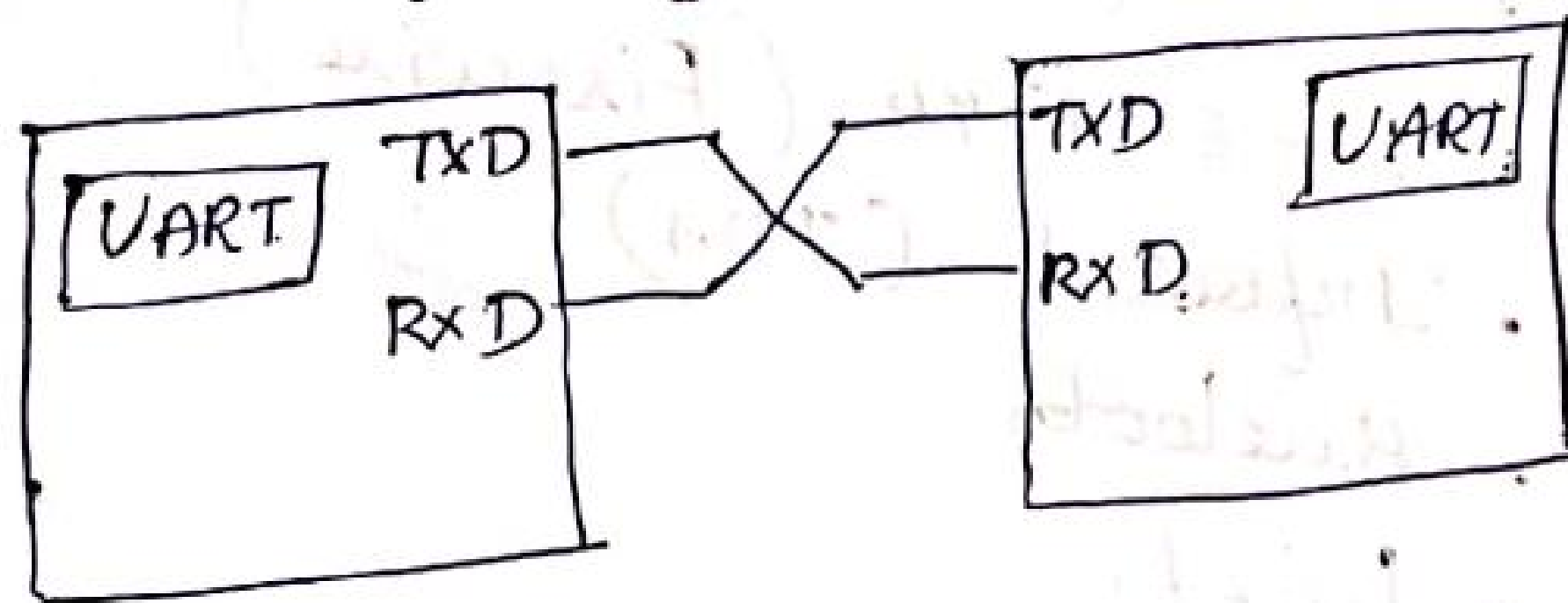
→ UART (Universal Asynch. Receiver Transmitter)

- performs an asynchronous form of serial data transmission.
- doesn't require a clock signal to synchronise both ends. Instead it relies upon a pre-defined agreement.
- The serial comm settings (Baudrate, no. of bits per second, parity, no. of start & stop bits) for both ends should be set as identical.

stop bit. MSB of data bit ----- LSB of data bit start bit. → to receiver

- Baudrate :- The no. of bits transmitted per sec. If the baud rate is 'x' bit/sec, the time slot available for 1 bit is $1/x$ sec.

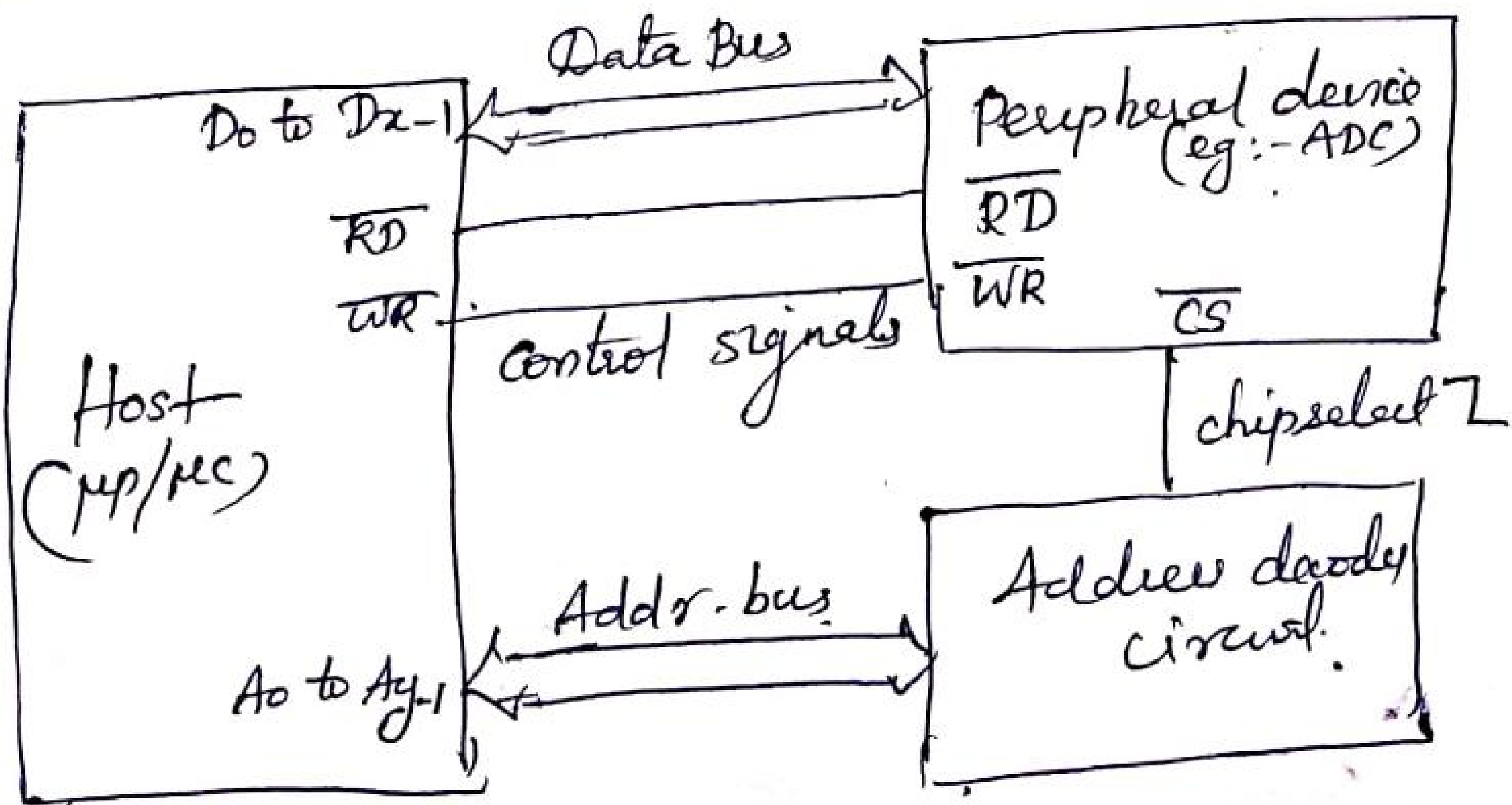
- UART Interfacing is shown in fig.



TXD → Transmitter line
RXD → Receiver line

- UART also provides hardware handshaking signal support for controlling the serial data flow.
- eg:- National Semiconductor's 8250 UART.

→ 1-wire Interface. } Refer Text
→ Parallel Interface. }



b) External / Peripheral Comm Interface

→ These are used for ext. comm of the ES.
ie, comm of different components present on the system with external or peripheral components ^{devices}

- Assign 1
Write short notes on
- egs:-
- RS-232 & RS-485
 - USB (Universal Serial Bus)
 - IEEE 1394 (Firewire)
 - Infrared (IDA) TV remote control
 - Bluetooth
 - Wi-Fi
 - ZigBee
 - GPRS (General Packet Radio Service)
- ← Thunderbolt
audio or visual peripherals
(multimedia purpose)

5. EMBEDDED FIRMWARE

→ refers to the Control Algorithm (Program Instructions) and/or the configuration settings that ES developer dumps into the Program memory of ES.

→ It is an unavoidable part of ES.

→ Methods used for developing the embedded firmware

1) Write the program in HLL like Embedded C/C++ using IDE (Integrated Development Environment, which contains an editor, compiler, linker, debugger, simulator etc).
eg:- Keil microvision 3 IDE for 8051-family μ cs.

2) Write the program in Assembly lang. using the instructions supported by our μ p/ μ C.

→ HEX File Creation :- The process of converting the prog written in either HLL/ALL to machine readable binary code.

HLL (ie using C/C++) → the cross compiler included in the IDE converts it into corresponding μ p/ μ C understandable hex file.

ALL → utilities provided by the μ p/ μ C vendors are used.

Firmware development in HLL

- Writing codes in HLL is easy
- Codes written in HLL are highly portable

Firmware development in ALL

- It is a tedious & time consuming job.
- Specific for a μ p/ μ C. The developer should know the instructions.

- HLL are not developer dependent. Any skilled programmer can traceout the functionalities if the prog contains ^{proper} comments & documentation lines

- ALL are developer dependent. As to implement the same functionality multiple instructions may be available.

→ 2 types of control algo. design exists in embedded firmware development.

1) Infinite^{loop} or Super loop based approach

- control flow runs from top to bottom & then jumps back to the top of the prog.

2) The Embedded OS Based Approach

- Splits the functions to be executed into tasks & running these tasks using a scheduler which is part of RTOS or RTOS.

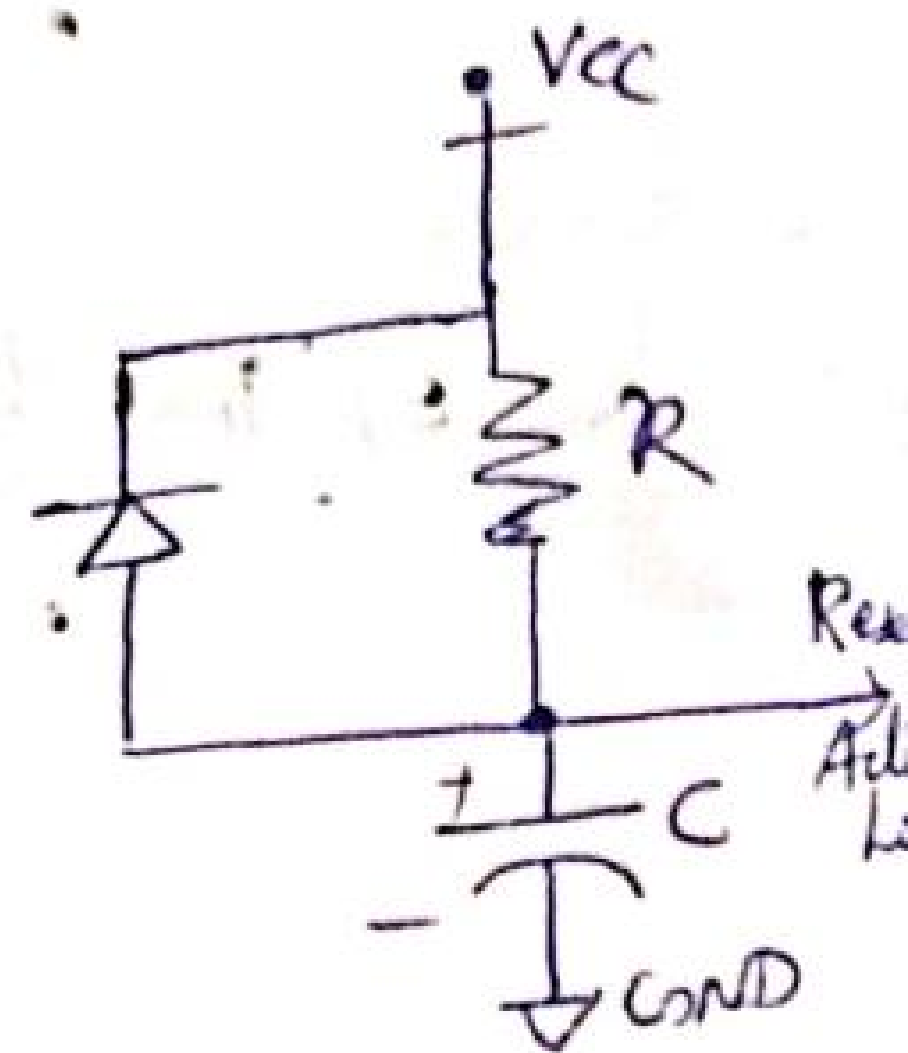
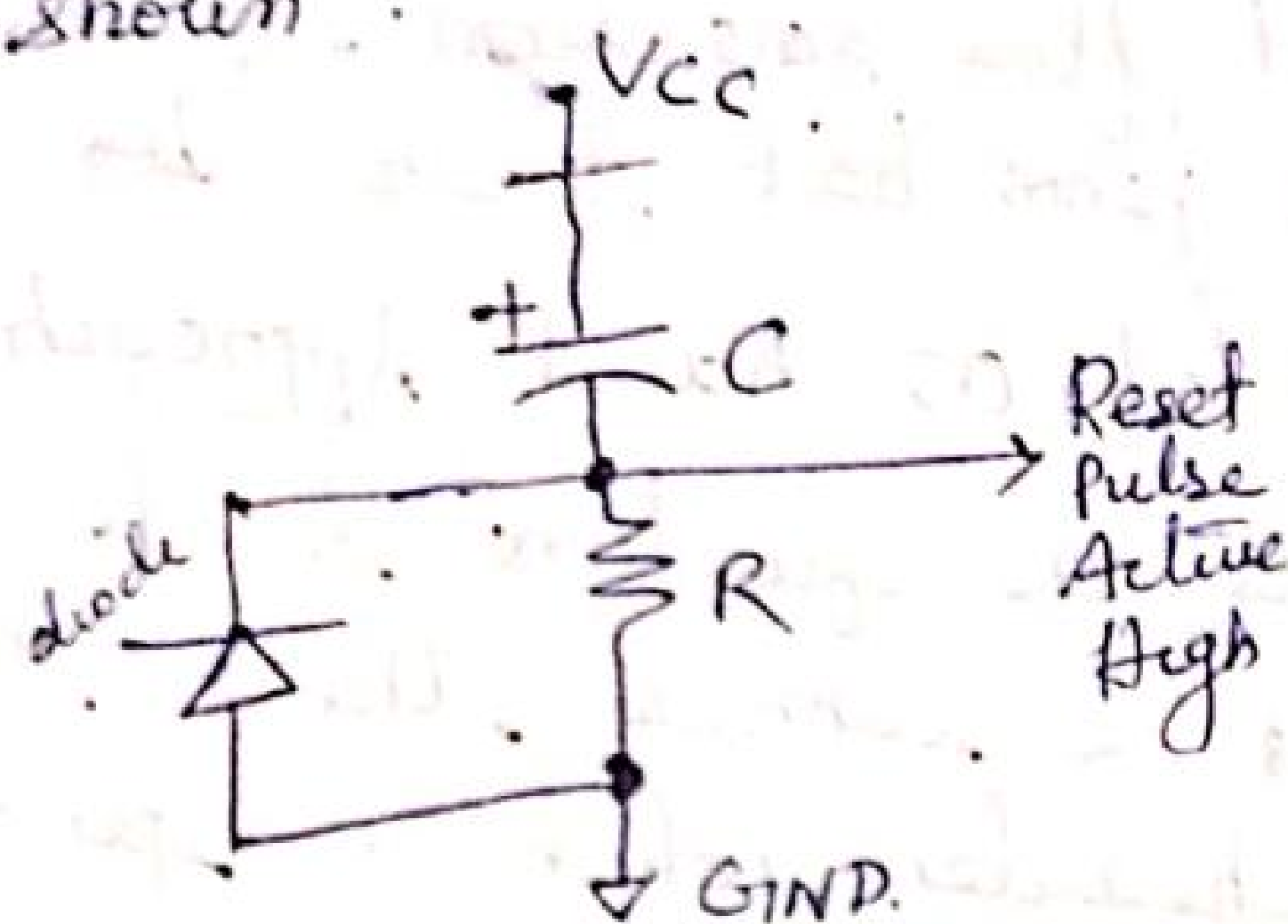
6. OTHER SYSTEM COMPONENTS

→ include components / circuits / ICs necessary for the proper functioning of ES.

→ 1) Reset Circuit

- is essential to ensure that the device is not operating at a voltage level where the device is not guaranteed to operate during system power on.

- The reset signal brings the internal registers and the different hardware systems of the $\mu p/\mu c$ to a known state and starts the firmware execution from the reset vector.
- This signal can be active low or active high based on $\mu p/\mu c$.
- A resistor-capacitor based reset circuit is shown.



2) Brown-out Protection Circuit

- prevents $\mu p/\mu c$ from unexpected program execution behaviour when the supply voltage to the $\mu p/\mu c$ falls below a specified voltage.

3) Oscillator Circuit

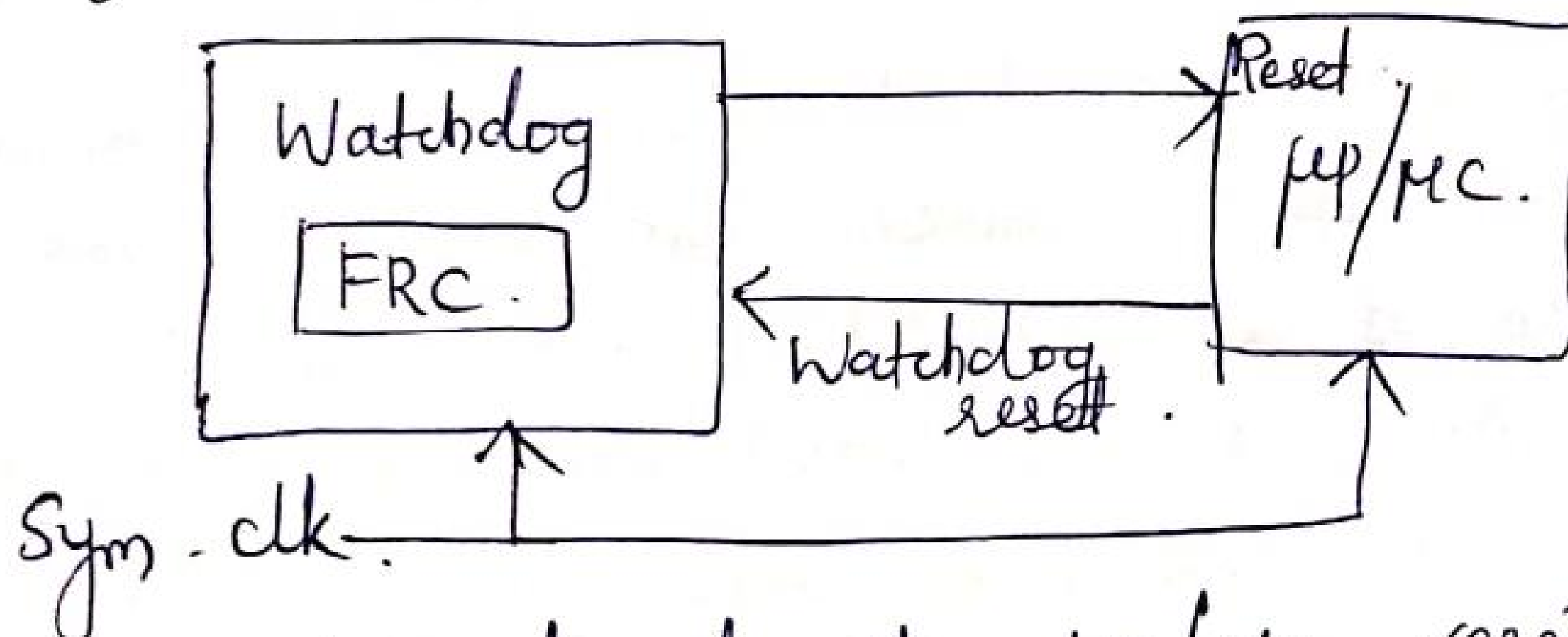
- generates the precise clock for the $\mu p/\mu c$.
- Certain processors contain a built-in oscillator circuit and simply require an external crystal for producing the clock signal.

4) Real-time clock (RTC)

- It is a system component responsible for keeping track of time.
- RTC holds information like current time in 12 hr / 24 hr format, date, month, year, day of the week etc. and supplies timing reference to the system.
- It is intended to function even in the absence of power.
- The RTC can interrupt the OS kernel by asserting the interrupt line of the $\mu p/\mu c$ to which the RTC interrupt line is connected.

5) Watchdog Timer / Watchdog

- is a hardware timer for monitoring the firmware execution.



At periodic intervals the $\mu p/\mu c$ resets the Free Running counter (FRC) of Watchdog to avoid timeout. If $\mu p/\mu c$ is not functioning properly, then watchdog timer counts its max to reach timeout which then resets the μp .

7. PCB & OTHER PASSIVE COMPONENTS

→ PCB (Printed Circuit Board) is the backbone of every ES

→ After finalizing the components & the interconnections among them, a schematic design is created & based on that PCB is fabricated.

→ PCB acts as a platform for mounting all the necessary components as per the design requirement. Also it acts as a platform for testing the embedded firmware.

→ Also ES include passive components like resistor, capacitor, diodes etc.



CHARACTERISTICS OF AN EMBEDDED SYSTEM

Following are some of the characteristics of an embedded system that make it different from a general purpose computer.

1) Application & Domain specific:

- An ES is designed for a specific purpose only.
It will not do any other task.
eg:- A washing m/c can only wash, it cannot cook.
- We cannot replace the embedded ctrl unit of microwave with AC.
- Certain ESs are specific to a domain.
eg:- A hearing aid is an application that belongs to the domain of signal processing.
- We cannot replace an ES for a particular domain say telecom, with another ~~control unit~~ domain like consumer electronics.

2) Reactive & Real time

- Certain ESs are designed to react to the events that occur in the nearby environment. These events also occur real time. Events are captured using sensors / ip devices & the control algorithm reacts to bring the controlled op variables to the desired level.
- Event can be $\begin{cases} \text{Periodic} \\ \text{Unpredictable} \end{cases}$ (don't miss them).
- eg:- AC adjusts its mechanical components as soon as it gets a signal from sensors.

• Real time System Operation means the timing behavior of the sym should be deterministic i.e. the sym should respond to requests or tasks in a known amount of time.

• A real time system should not miss any deadlines.

• Not all ESs are real time syms.

• eg. for RT ESs are mission critical appl^s, flight control syms, etc.

3) Operates in Harsh Environment

• ESs may be deployed in $\left\{ \begin{array}{l} \text{controlled environment} \\ \text{Harsh} \end{array} \right.$

• Harsh environment include $\left\{ \begin{array}{l} \text{dusty} \\ \text{high temperature zone} \\ \text{Area subject to vibration \& shock} \end{array} \right.$

• ESs deployed in harsh environment should be able to withstand all these conditions.

4) Distributed

• Certain ESs are part of a larger system and thus form components of a distributed system.

These components are independent of each other.

eg:- 1) Automatic vending m/c. (It contains a card reader, a vending unit etc, all are independent units, but they work together to perform the vending fn)

2) ATM (Automatic Teller M/c), contains card reader, transaction unit, currency counter, printer unit.

3) SCADA (Supervisory Control & Data Acquisition) sym which contains physically distributed individual control units connected to a supervisory module.

used in ctrl & instrumentation appl^s

5) Small Size & Weight

- An ES that is compact in size and has lightweight will be desirable or more popular than one that is bulky & heavy
- eg:- currently available cell phones. The cell phones that have the max. features are popular but also their size & weight is an imp. characteristic.
- For convenience users prefer mobile phones than phablets (phone + tablet pc)

6) Power Concerns

- Power mgmt is an imp. factor that needs to be considered in ~~the~~ ES design
- It is desirable that the power utilization and heat dissipation of any ESs be low.
- If more heat is dissipated then additional units like heat sinks or cooling fans need to be added to the circuit which make the sys bulky
- Choose low dropout regulators, $\mu\text{P}/\mu\text{Cs}$ with power saving mode etc
- Also power mgmt is critical in battery operated devices/applications
- The more the power consumption, the less is the battery life

QUALITY ATTRIBUTES OF AN ES.

3.2 Quality Attributes of Embedded Systems

- Quality attributes are the non-functional requirements that need to be documented properly in any system design.
- If the quality attributes are more concrete and measurable, it will give a positive impact on the system development process and the end product.
- The various quality attributes that needs to be addressed in any embedded system development are broadly classified into two, namely
 - Operational Quality Attributes
 - Non-Operational Quality Attributes

i. Response

- Response** is a measure of quickness of the system.
- It gives you an idea about how fast your system is tracking the input variables.
- Most of the embedded system demand fast response which should be real-time.
- Ex. An embedded system deployed in flight control application should respond in a Real Time manner.
- Any response delay in the system will create potential damages to the safety of the flight as well as the passengers.
- It is not necessary that all embedded systems should be Real Time in response.
- For example, the response time requirement for an electronic toy is not at all time-critical.

1. Operational Quality Attributes

The operational quality attributes represent the relevant quality attributes related to the embedded system when it is in the operational mode or 'online' mode. The important quality attributes coming under this category are listed below:

- Response
- Throughput
- Reliability
- Maintainability
- Security
- Safety

ii. Throughput

- Throughput deals with the efficiency of system.
- It can be defined as rate of production or process of a defined process over a stated period of time.
- The rates can be expressed in terms of units of products, batch produced, or any other meaningful measurements.
- In case of card reader like the ones used in buses, throughput means how much transactions the Reader can perform in a minute or hour or day.
- Throughput is generally measured in terms of 'Benchmark'. 'Benchmark' is a reference point by which something can be measured.
- Benchmark can be a set of performance criteria that a product is expected to meet or a standard product that can be used for comparing other products of the same product line.

iii. Reliability

- Reliability is a measure of how much percentage you rely upon the proper functioning of the system or what is the % susceptibility of the system to failure.
- Mean Time Between Failures (MTBF) and Mean Time To Repair (MTTR) are the terms used in defining system reliability.
- MTBF gives the frequency of failures in hours/weeks/months.
- MTTR specifies how long the system is allowed to be out of order following a failure.
- For an embedded system with critical application need, it should be of the order of minutes.

iv. Maintainability

- Maintainability deals with support and maintenance to the end user or client in case of technical issues and product failures or on the basis of a routine system checkup.
- Reliability and maintainability are considered as two complementary disciplines. A more reliable system means a system with less corrective maintainability requirements and vice versa.
- Maintainability can be classified into two types:

1. Scheduled or Periodic Maintenance (Preventive Maintenance)

An inkjet printer uses ink cartridges, which are consumable components and as per the printer manufacturer the end user should replace the cartridge after each 'n' number of printouts to get quality prints.

2. Maintenance to Unexpected Failures (Corrective Maintenance)

If the paper feeding part of the printer fails the printer fails to print and it requires immediate repairs to rectify this problem.

Hence it is obvious that maintainability is simply an indication of the availability of the product for use. In any embedded system design, the ideal value for availability is expressed as

$$A_i = \frac{MTBF}{(MTBF + MTTR)}$$

Where A_i = Availability in the ideal condition, MTBF = Mean Time Between Failures, and MTTR = Mean Time To Repair

v. Security

- 'Confidentiality', 'Integrity', and 'Availability' are three major measures of information security.
- 'Confidentiality' deals with the protection of data and application from unauthorized disclosure.
- 'Integrity' deals with the protection of data and application from unauthorized modification.
- 'Availability' deals with protection of data and application from unauthorized users.
- Certain embedded systems have to make sure they conform to the security measures.
- Ex. An electronic safety Deposit Locker can be used only with a pin number like a password.

vi. Safety

- Safety deals with the possible damages that can happen to the operators, public and the environment due to the breakdown of an embedded system or due to the emission of radioactive or hazardous materials from the embedded products.
- The breakdown of an embedded system may occur due to a hardware failure or a firmware failure.
- Safety analysis is a must in product engineering to evaluate the anticipated damages and determine the best course of action to bring down the consequences of the damages to an acceptable level.

2. Non Operational Attributes

- The quality attributes that needs to be addressed for the product 'not' on the basis of operational aspects are grouped under this category. The important quality attributes coming under this category are listed below:

- Testability & Debug-ability
- Evolvability
- Portability
- Time to prototype and market
- Per unit and total cost

i. Testability & Debug-ability

- Testability deals with how easily one can test his/her design, application and by which means he/she can test it.
- For an embedded product, testability is applicable to both the embedded hardware and firmware.
- Debug-ability is a means of debugging the product as such for figuring out the probable sources that create unexpected behavior in the total system.
- Debug-ability has two aspects in the embedded system development context, namely, hardware level debugging and firmware level debugging.
- Hardware debugging is used for figuring out the issues created by hardware problems whereas firmware debugging is employed to figure out the probable errors that appear as a result of flaws in the firmware.

ii. Evolvability

- Evolvability is a term which is closely related to Biology.
- Evolvability is referred as the non-heritable variation.
- For an embedded system, the quality attribute 'Evolvability' refers to the ease with which the embedded product (including firmware and hardware) can be modified to take advantage of new firmware or hardware technologies.

iii. Portability

- Portability is a measure of 'system independence'.
- An embedded product can be called portable if it is capable of functioning in various environments, target processors/controllers and embedded operating systems.
- A standard embedded product should always be flexible and portable.

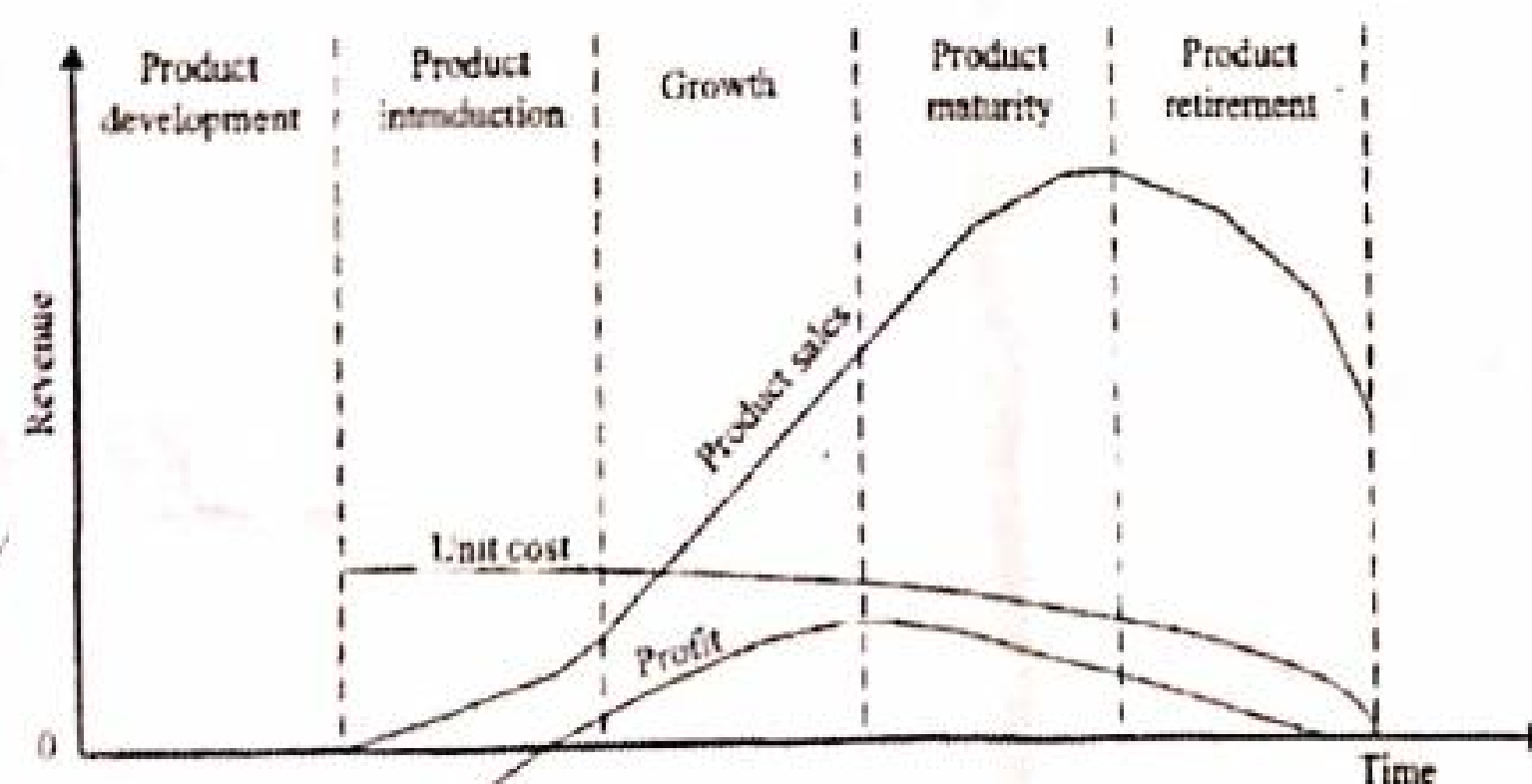
iv. Time-to-Prototype and Market

- Time-to-market is the time elapsed between the conceptualization of a product and the time at which the product is ready for selling (for commercial product) or use (for non-commercial products).
- The commercial embedded product market is highly competitive and time to market the product is a critical factor in the success of a commercial embedded product.
- Product prototyping helps a lot in reducing time-to-market.

v. Per Unit Cost and Revenue

- Cost is a factor which is closely monitored by both end user (those who buy the product) and product manufacturer (those who build the product).
- Cost is a highly sensitive factor for commercial products.
- Proper market study and cost benefit analysis should be carried out before taking decision on the per unit cost of the embedded product.
- When the product is introduced in the market, for the initial period the sales and revenue will be low.
- There won't be much competition when the product sales and revenue increase.
- During the maturing phase, the growth will be steady and revenue reaches highest point and at retirement time there will be a drop in sales volume.

Cont'd



Product life cycle (PLC) curve

MODULE - 2

HARDWARE SOFTWARE CO-DESIGN & PROGRAM MODELLING

HARDWARE SOFTWARE CO-DESIGN

→ Traditional ES development

- Software-hardware partitioning is done first
- S/w group take care of s/w architecture development
h/w " " h/w building.
- Less interaction b/w 2 teams
- Once hardware & s/w are ready, the integration is performed.

→ Co-design Approach for ES development

- H/w & s/w are co-developed (or concurrently developed) instead of independently developing both
- Verification is also done concurrently.
- It tries to exploit the ^{interaction} synergy of h/w & s/w with the goal to optimize & satisfy design constraints such as cost, performance and power of the final product.
- steps ^{sym design} < Architecture design.
 - The product requirements captured from the customer are converted into system level needs or processing requirements (or functional requirements)

sym design → These requirements are then transferred into functions which can be simulated & verified against performance & functionality.

architectural design → The partitioning of sym-level requirements into h/w & s/w take place during the architecture design phase. The partitioning is performed based on the hardware-s/w trade-offs.

→ The architectural design results in the detailed behavioural description of the h/w requirement & the definition of the s/w required for the hardware.

→ The processing requirement behaviour is usually captured using computational models (eg:- Data flow diagram model, state m/c model, sequential program model etc) and ultimately the models representing the s/w processing requirements are translated into firmware implementation using programming languages.

• h/w & s/w trade-offs

→ Certain sym-level processing requirements may be possible to develop in either h/w or s/w.

The actual selection is based on < the trade-offs actual sym requirements.

eg:- Some multimedia codec requirement.

Here trade-off is the performance & Re-
h/w implementation → efficient, low processing, low power, optimized
s/w " → reusable, reconfigurable etc.

→ Some imp. h/w s/w trade-offs in ES design are.

- 1) Processing speed & Performance
- 2) Frequency of change (Re-configurability)
- 3) Memory size and gate count
- 4) Reliability
- 5) Man Hours (Effort) & cost.

FUNDAMENTAL ISSUES IN HARDWARE - SOFTWARE CO-DESIGN

1) Selecting the Model

- Model → is a formal sym consisting of objects and composition rules. e.g. - DFD, state m/c model, sequential program model etc.
- In h/w s/w co-design, models are used for capturing & describing the sym characteristics.
- Choosing a model is a difficult job. Most often designers switch b/w a variety of models from the requirement specification to the implementation aspect of the sym design. (because the objective varies with each phase)

2) Selecting the Architecture

- Architecture specifies how a sym is going to implement in terms of the number & types of different components & the interconnection among them.

• Commonly used architectures in sym design are.

- Application specific architecture → Controller architecture
- General purpose architecture { → Datapath "
- CISC
- RISC.
- Parallel processing architecture { → Very long Instruction Word Computing (VLIW)
- SIMD (Single Instrⁿ Multiple Data)
- MIMD (Multiple " " " ") etc.

• Controller Architecture

- implements the FSM model using a state register & two combinational circuits. The state reg. holds the present state & the combinational circuits implement the logic for next state and o/p.

• Datapath Architecture

- implements the data flow graph model, where the o/p is generated as a result of a set of predefined computations on the i/p data.
- Data path → a channel b/w the i/p & o/p and it may contain registers, counters, reg. files, memories & ports along with high speed arithmetic units.
- Ports connect the datapaths to multiple buses.
- Most of the time, arithmetic units are connected in parallel with pipelining support for bringing high performance.

- CISC & RISC (Refer pip/pe - core of ES)
- VLIW
 - implements multiple functional units (ALU, multipliers etc) in the datapath.
 - The VLIW architecture packages one stand instruction per functional unit of the datapath.
- SIMD & MIMD
 - Parallel processing architecture implements multiple concurrent Processing Elements (PE) and each processing element may associate a datapath containing register & local memory. eg:- SIMD, MIMD
 - SIMD :- Single Instruction is executed in parallel with the help of PEs. They form the basis of re-configurable processors.
 - MIMD :- PEs execute different instructions at a given point of time.. They form the basis of multiprocessed systems.

3) Selecting the Language

- A programming lang. captures a "Computational Model" and maps it into architecture.
- A model can be captured using multiple programming languages.
 - eg:- C, C++, C#, Java, etc for S/W implementation
 - VHDL, System C, Verilog for h/w "

- Also, a single language can be used for capturing a variety of models.
- Certain langs. are good for capturing certain models.
eg:- C++ is good for capturing objected oriented model.
- Lang. selection criteria → Lang. should capture the model easily.

4) Partitioning System Requirements into hardware & software

- It is possible to implement the system requirements in either h/w or s/w.
- The decision depends on various h/w s/w trade-offs. like.

(Refer Prev. Page)

COMPUTATIONAL MODELS IN EMBEDDED DESIGN

1) Data Flow Graph / Diagram (DFG) Model

- It translates the data processing requirements into a data flow graph.
- It is a data driven model in which the program execution is determined by data.
- It focus on data & operations on the data which transforms the ip data to op data.
- It is a visual model in which

○ → process (or operation on data)
→ → dataflow. $\left[\begin{array}{cc} \text{ip data} & \text{op data} \\ \rightarrow \text{○} & \text{○} \rightarrow \end{array} \right]$

- Embedded applications which are computational intensive & data driven are modeled using DFG.

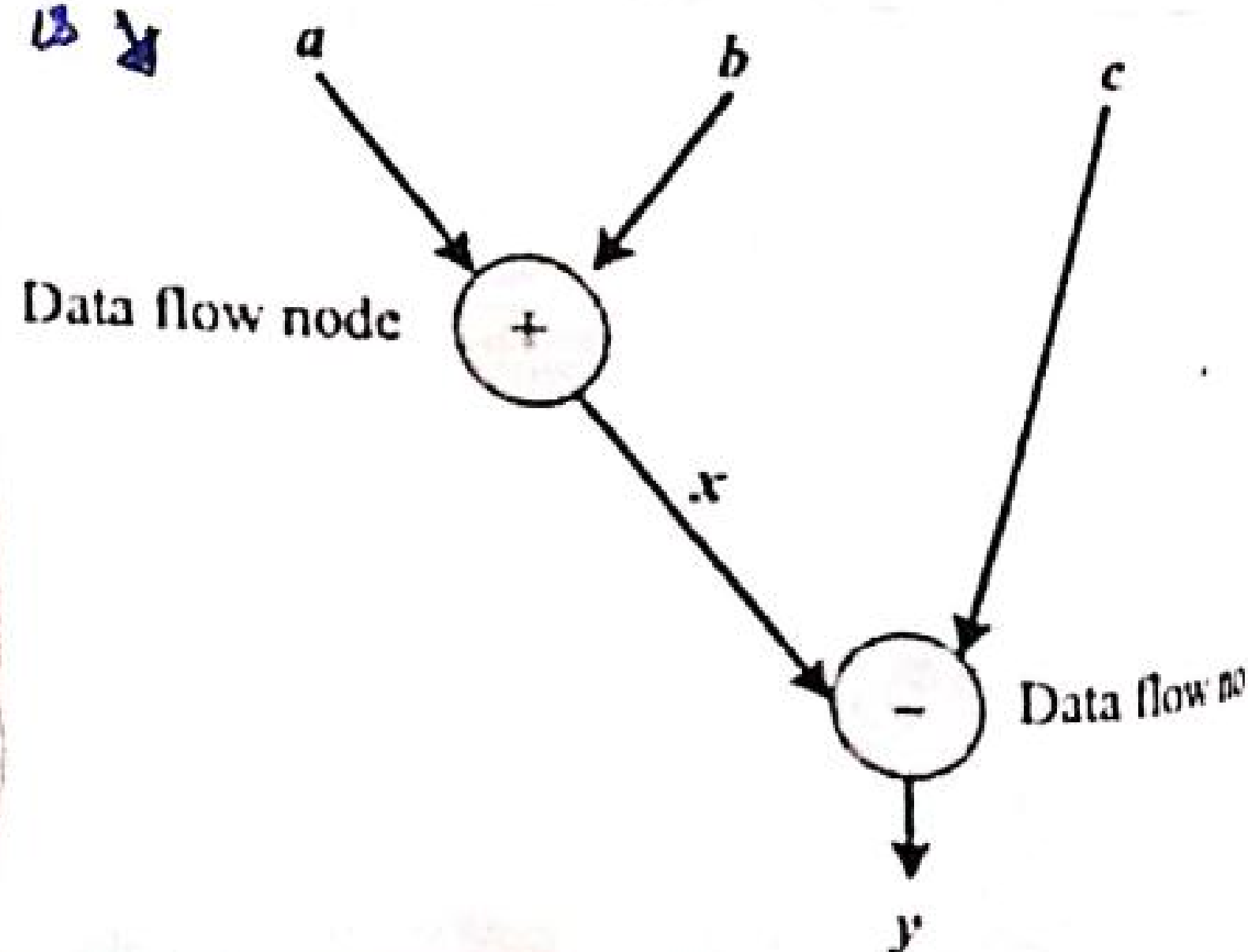
eg :- DSP Applications.

- eg ~~error~~ : Suppose the ^{computational} requirements are,
 $x = a + b$ and $y = x - c$.

the corresponding DFG is

- Here datapath is the data flow path from ip to op.

- Acyclic DFG (ADFG) → It doesn't contain multiple values for ip variable & multiple op values for a given set of input(s).



→ Feedback inputs, events etc are examples for non-acyclic inputs.

→ A DFG model translates the program as a single sequential process execution.

2) Control Data Flow Graph / Diagram (CDFG)

→ Used for modelling applications involving conditional program execution.

→ This model contains both data operations and control operations.

→ It uses DFG as element & conditional constructs as decision makers.

→ It contains. < data flow nodes & decision nodes

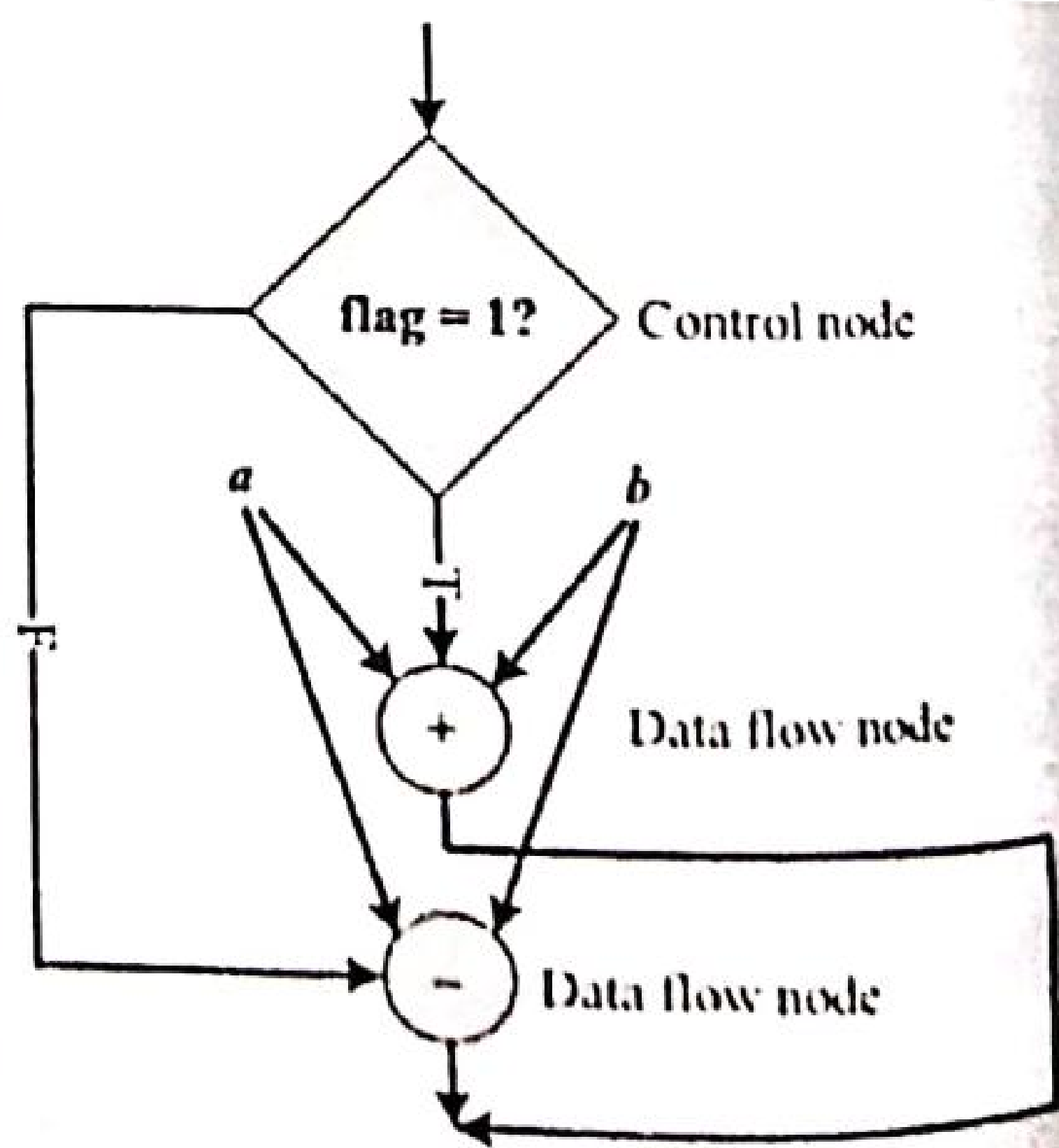
→ Consider the requirement.

If $\text{flag} = 1$, $x = a + b$; else $y = a - b$;

The CDFG model for this is shown in fig.

◇ → Control node (decision making element).

→ The decision on which process is to be executed is determined by the control node.



→ Real World eg :- Capturing & saving of the image to a format set by the user in a digital still camera. The decision on, in which format the image is stored (formats like JPEG, TIFF, BMP etc) is controlled by the camera settings, configured by the user.

3) State Machine Model

→ Used for modelling Reactive or Event-driven ESs, whose processing behaviour are depended on state transitions.
eg. for event driven ESs → ESs used in control & Industrial applications.

→ This model describes the system behaviour with

- States : A representation of a current situation

- Events : An input to the state, which act as stimuli for state transition

- Actions : An activity to be performed by the state machine.

- Transition : Movement from one state to another.

→ FSM (Finite State Machine) model.

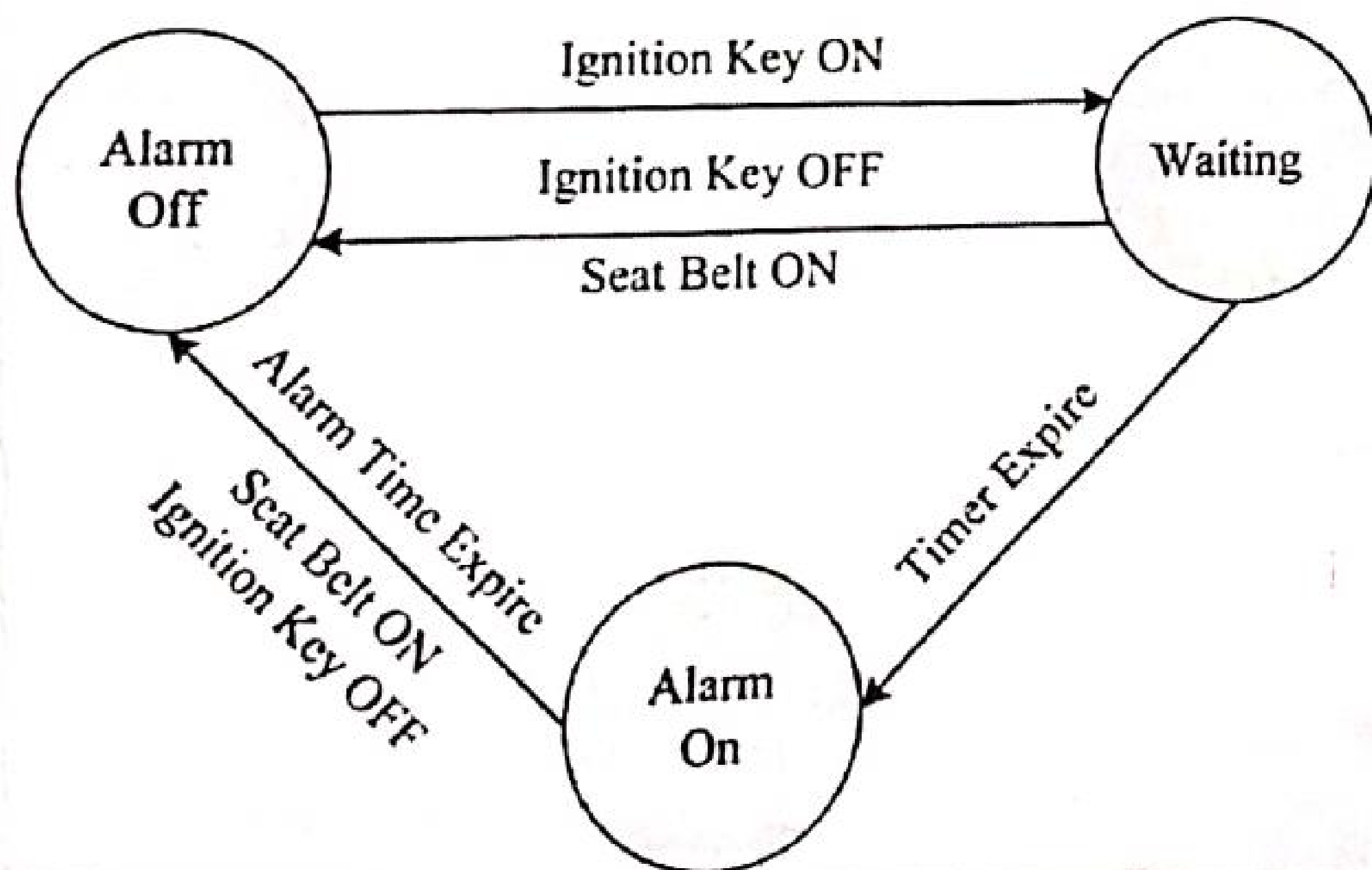
- In FSM, the no. of states are finite. ie, the sys is described using a finite number of possible states.

eg1) FSM model for Automatic seat belt warning system.

Suppose the system requirements are,

- 1) 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.
- 2) The Alarm is turned off when the alarm-time (5 sec) expires or if the driver/passenger fastens the belt or if the ignition switch is turned off, whichever happens first.

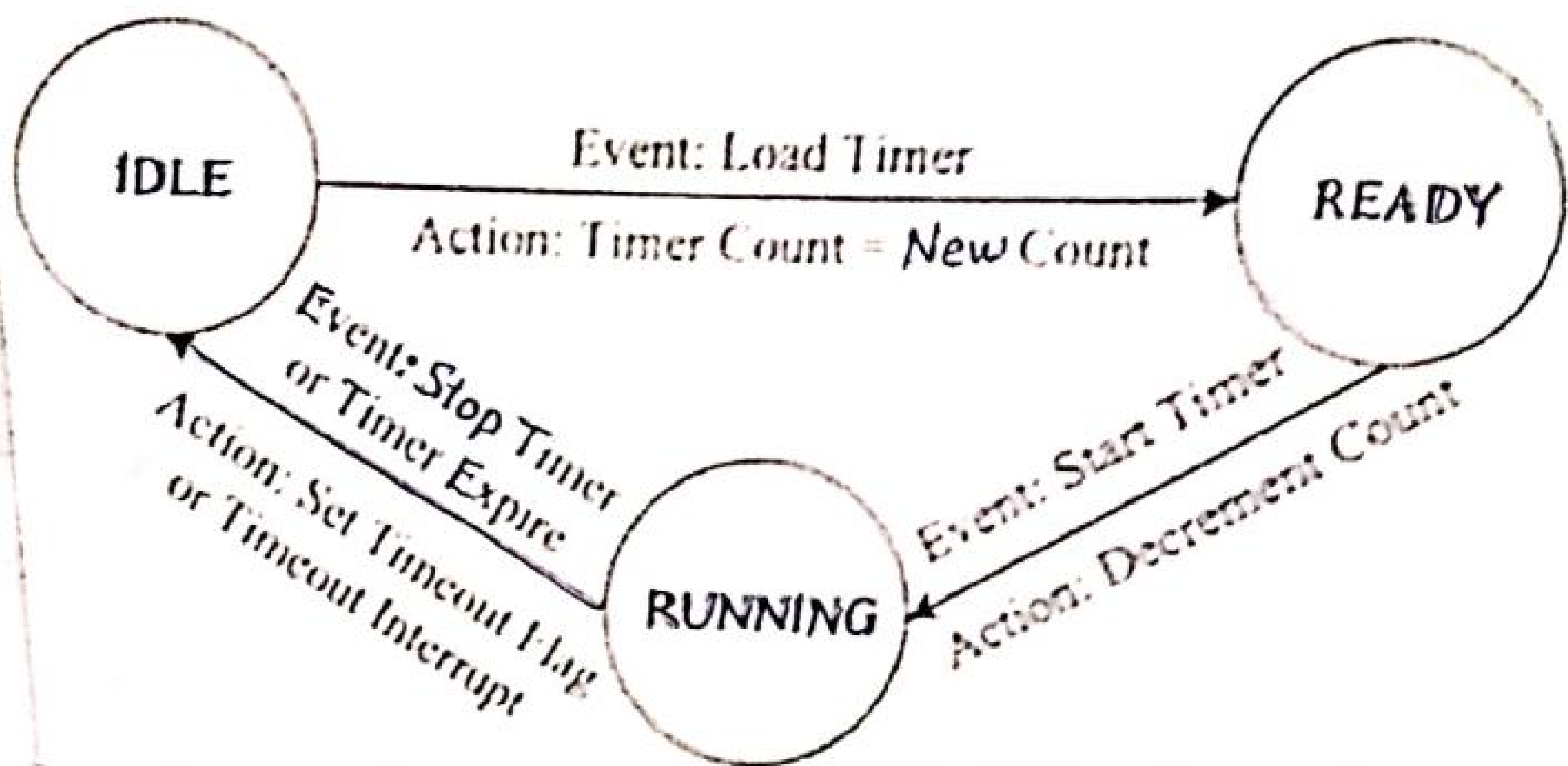
These requirements can be modelled using FSM as shown below.



(expⁿ)

Here wait state is implemented using a timer. The timer also has certain set of states and events for state transitions.

The FSM model for timer is shown below.



(expt)

eg:- FSM Model for Automatic Tea/Coffee Vending Machine

requirement - The tea/coffee vending is initiated by user inserting a 5 rupee coin. After inserting the coin, the user can either select 'Coffee' or 'Tea' or press 'Cancel' to cancel the order & take back the coin.

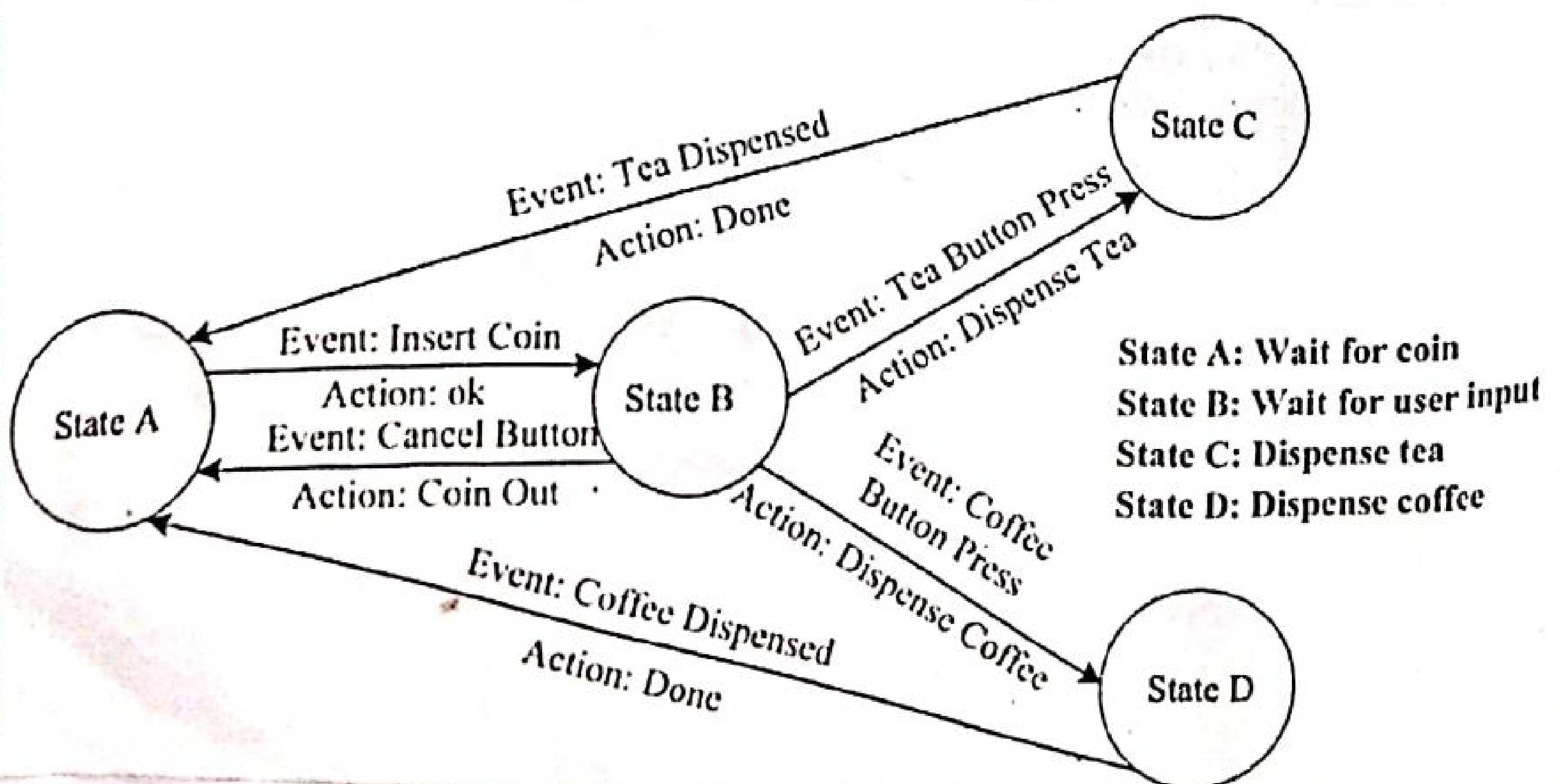
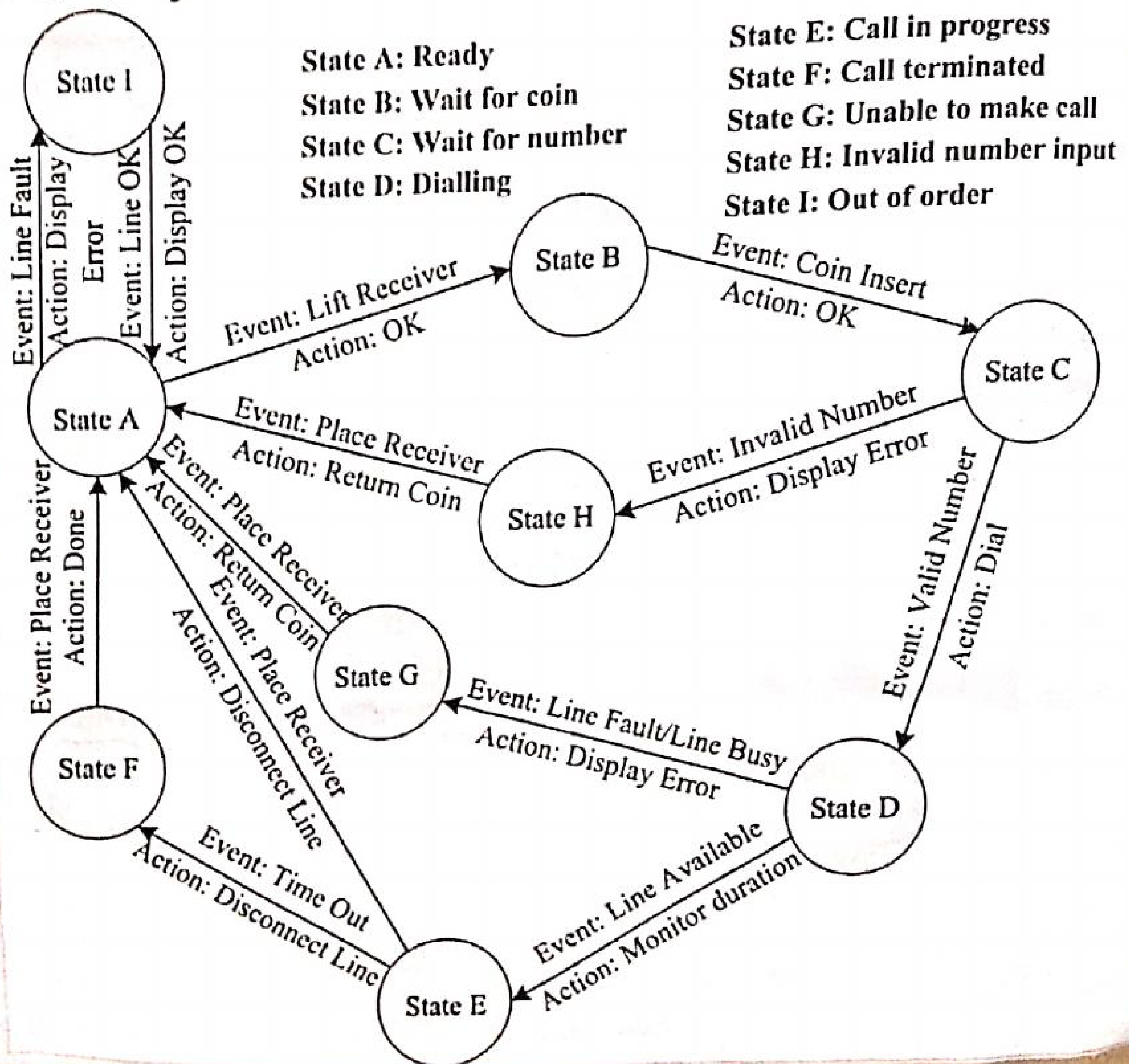


Fig 3 :- FSM Model for Coin Operated Telephone System

Requirements

1. The calling process is initiated by lifting the receiver (off-hook) of the telephone unit.
2. After lifting the phone the user needs to insert a 1 rupee coin to make the call.
3. If the line is busy, the coin is returned on placing the receiver back on the hook (on-hook).
4. If the line is through, the user is allowed to talk till 60 sec and at the end of 45th sec, prompt for inserting another 1 rupee coin for continuing the call is initiated.
5. If the user doesn't insert 1st coin, call is terminated after 60 sec.
6. The sym is ready to accept new call request when the receiver is placed back on the hook.
7. The sym goes to the 'Out of order' state when there is a line fault.



4) Sequential Program Model

- Same as Conventional Procedural Programming
- the functions or processing requirements are executed in sequence.

→ tools used for sequential prog modelling < ^{FSM} Flowcharts.

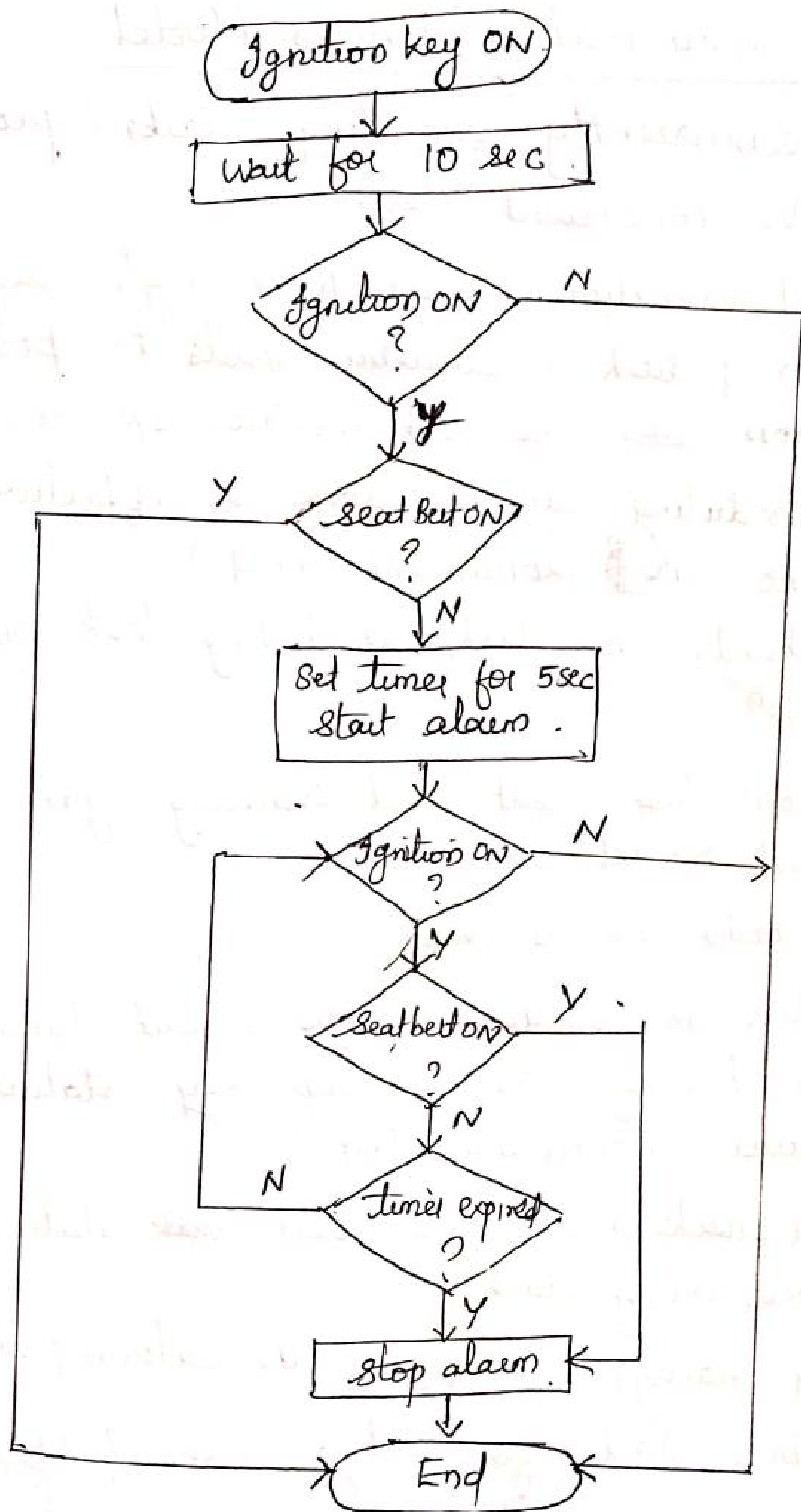
FSM represents the states, events, transitions & actions.
Flowchart models the execution flow.

- eg:- Seat Belt warning sys → The execution of functions in a sequential prog model is given below.

```
# define ON 1
# define OFF 0
# define YES 1
# define NO 0.
```

```
void seat_belt_warn ()
{
    wait_10sec ();
    if (check_ignition_key () == ON)
    {
        if (check_seat_belt () == OFF)
        {
            set_timer (5);
            start_alarm ();
            while ((check_seat_belt () == OFF) &&
                (check_ignition_key () == OFF) &&
                timer_expired () == NO);
            stop_alarm ();
        }
    }
}
```

- Fig. below shows the flowchart approach for of Sequential Program model for seat belt warning system.



5) Concurrent / Communicating Process Model

→ It models concurrently executing tasks / processes

→ Sequential Vs Concurrent model

- Sequential execution leads to a single sequential execution of task & thereby leads to poor processor utilisation. when the task involves I/O waiting, sleeping etc.
- Task scheduling can be done to effectively use the CPU (concurrent model)
- But overheads in task scheduling, task synchronization and comm.

→ eg :- Implementing 'Seat Belt Warning Sym' in concurrent model.

Split the tasks into 5 ~~steps~~.

1. Timer task for waiting 10 sec (wait timer task)
2. Task for checking the ignition key status (ignition key status monitoring task)
3. Task for checking the seat belt ~~lock~~ status (seatbelt status monitoring task)
4. Task for starting & stopping the alarm (alarm control task)
5. Alarm timer task for waiting 5 sec (Alarm timer task)

Here we need to synchronize the execution of these tasks through some mechanism (we can't execute them randomly/sequentially). Some events are associated with each task to manage this scenario.

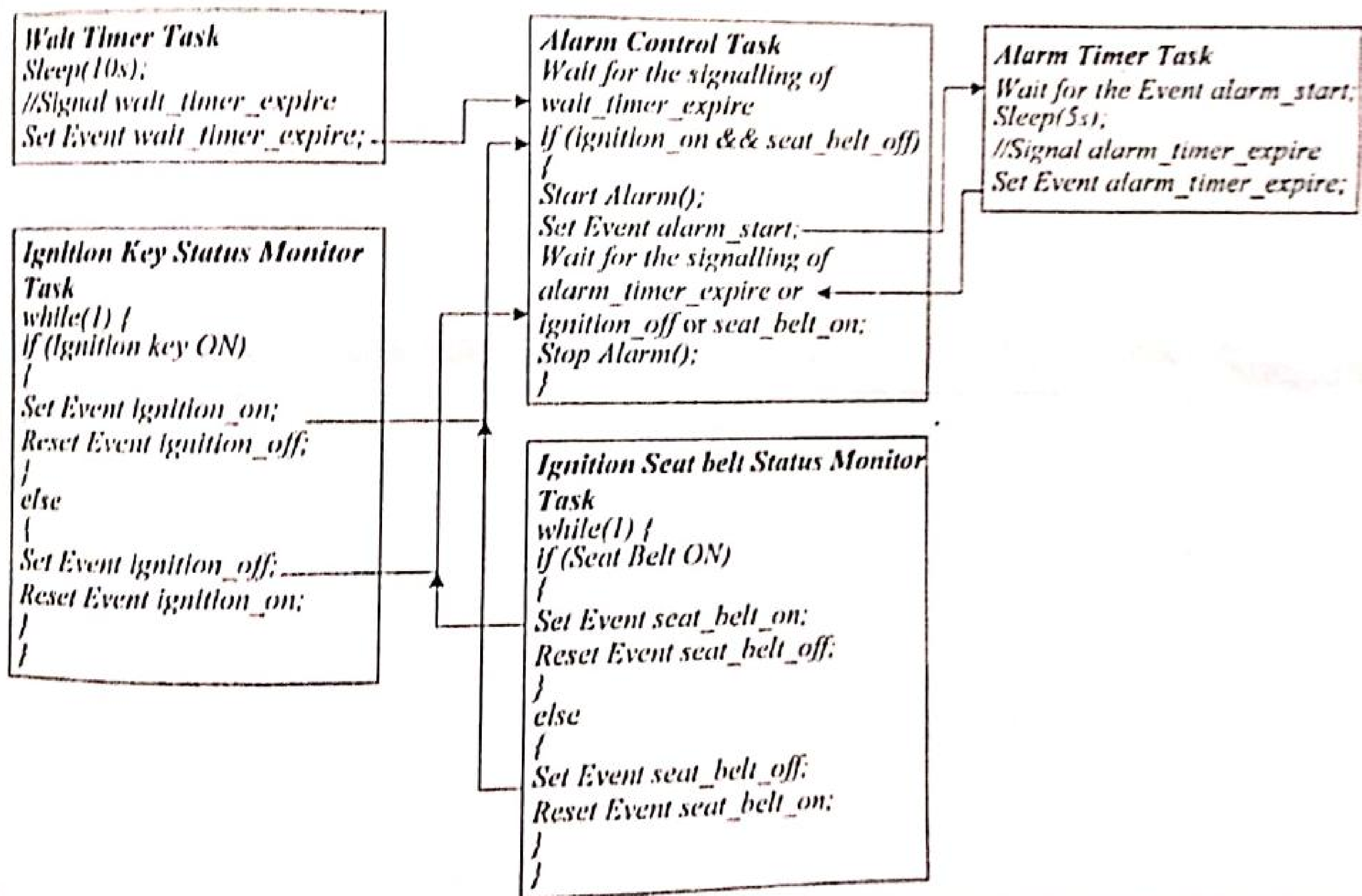
Events are ⇒ Wait timer expires → task ①
ignition-on, ignition-off → ②
seat-belt-on, seat-belt-off → ③
alarm-start - ④ ~~alarm timer start~~, alarm timer expires ⑤

Tasks & events for 'seat belt warning sym'

Create and initialise events
 { wait_timer_expire, ignition_on, ignition_off,
 seat_belt_on, seat_belt_off,
 alarm_timer_start, alarm_timer_expire
 Create task Wait Timer
 Create task Ignition Key Status Monitor
 Create task Seat Belt Status Monitor
 Create task Alarm Control
 Create task Alarm Timer

Concurrent Processing Program model for 'seat belt warning sym'

(a)



- The concurrent model is commonly used for modelling of 'Real time systems'
- Various techniques like shared memory, Message Passing, Events etc are used for comm & synchronization b/w concurrent processes

6) Object - Oriented Model

→ an object based model, where object is an entity used for representing or modelling a particular piece of the system. Each object is characterized by a set of unique behaviour & state.

→ class :- An abstract description of a set of objects (i.e., a blueprint of objects).

A class represents the state of an object through member variables and object behaviour through member functions.

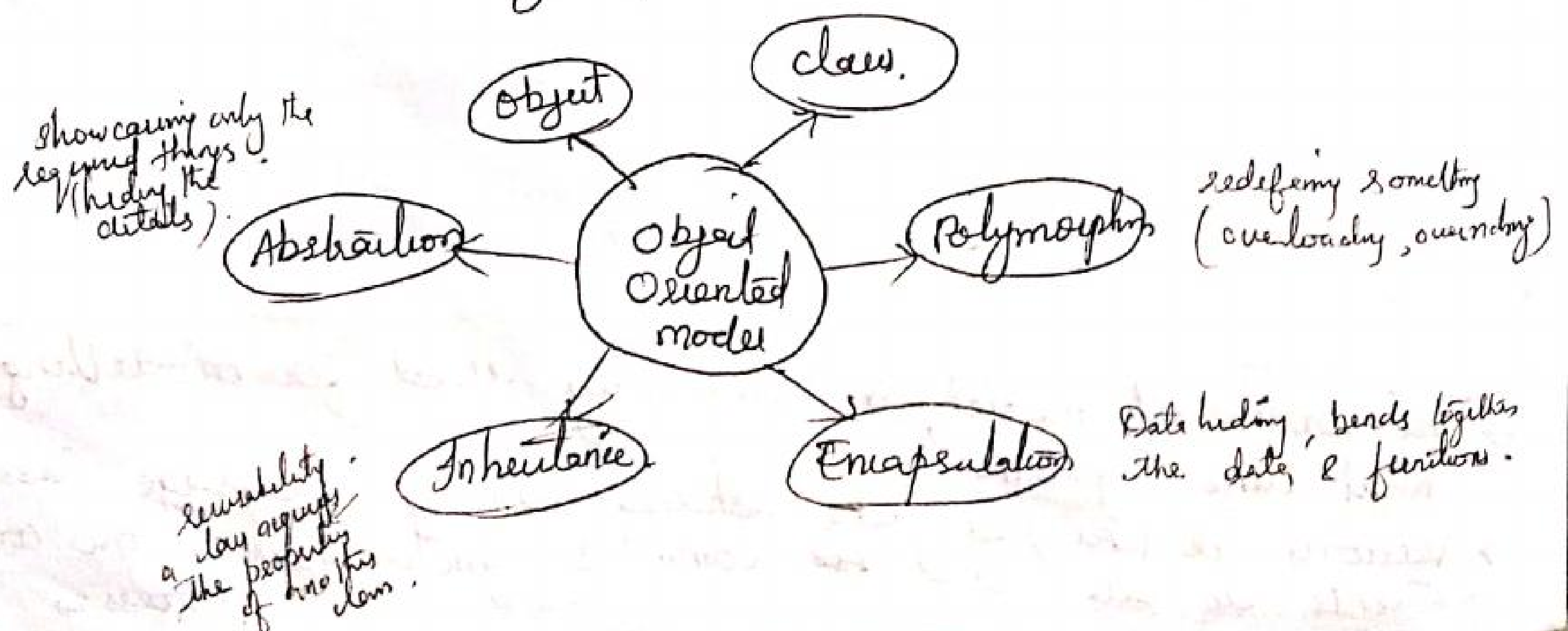
→ This can be private, public or protected

public : accessible within the class and outside the class

private : accessible only within in the class

protected : protected from external access, but classes derived from a parent class can also access the protected members.

→ This model also brings concepts like Abstraction, Inheritance, Polymorphism etc.



UNIFIED MODELLING LANGUAGE (UML)

- A visual modelling language for Object Oriented Design (OOD)
- It helps in all phases of sys design like requirements
capturing
design
deployment
- UML Building blocks

① Things :- A thing is an abstraction of the UML model.

a) Structural things :-

- Represents mostly the static parts of a UML model
- also known as classifiers
- eg:- class, interface, use case, use case realization (collaboration), actor class, component & node

b) Behavioural things

- Represents the dynamic parts of a UML model
- eg:- Interaction, state machine & activity

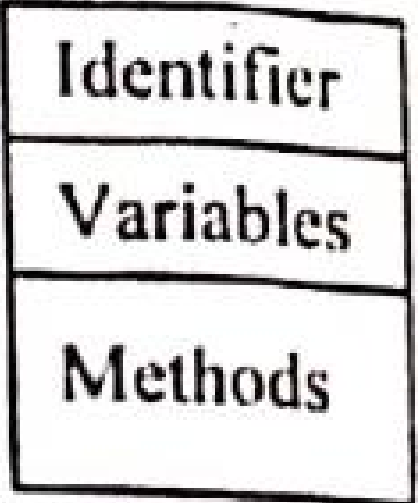
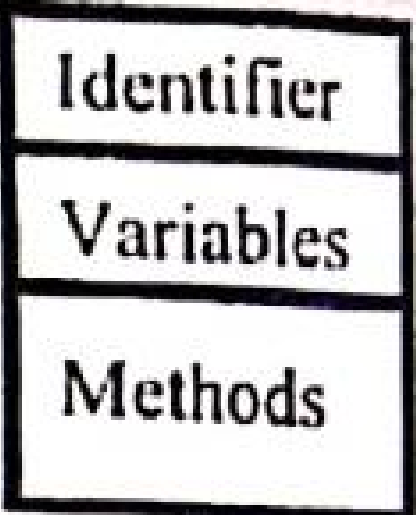
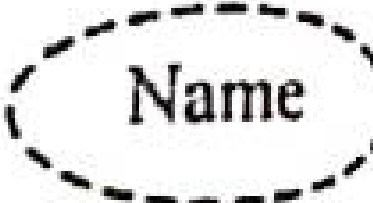
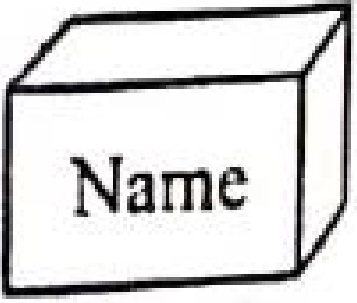
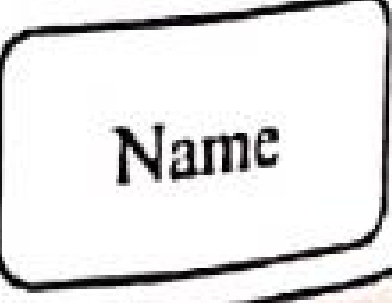
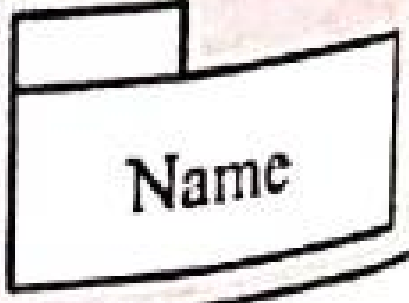
c) Grouping things

- Organisational parts of a UML model
- eg:- Package & subsystem

d) Annotational things

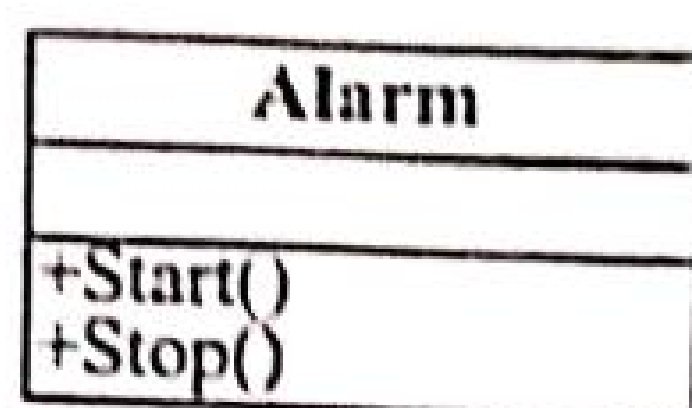
- Explanatory parts of a UML model
- eg:- Note

Fig. shows the 'things' in a UML model.

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.	
	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.	
	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.	
	Collaboration (Use case Realization)	Interaction diagram specifying the collaboration of different use cases. It is normally represented with a dotted ellipse indicating the name.	
	Component	Physical packaging of classes and interfaces.	
	Node	A computational resource existing at run time. Represented using a cube with name.	
	Interaction	Behaviour comprising a set of objects exchanging messages to accomplish a specific purpose. Represented by arrow with name of operation	
	State Machine	Behaviour specifying the sequence of states in response to events, through which an object traverses during its lifetime.	
Grouping	Package	Organises elements into packages. It is only a <u>conceptual thing</u> . Represented as a <u>tabbed folder</u> with name.	
Annotational	Note	Explanatory element in UML models. Contains formal informal <u>explanatory text</u> . May also contain <u>embedded image</u> .	

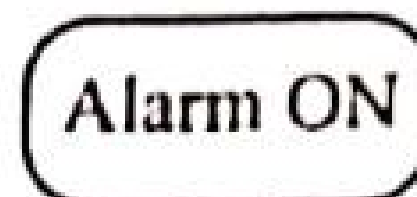
Eg:-

a) The Alarm class

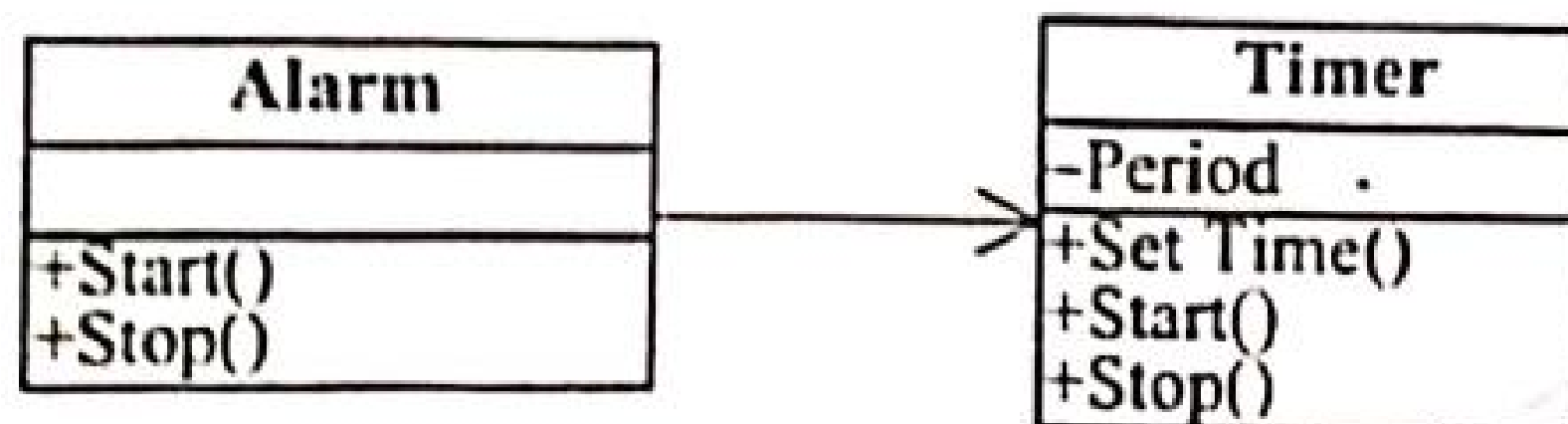


+ - public attribute
 - - private "
 # - protected "
 ~ - package "

b) State representation for 'Alarm ON' state



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



② Relationships

They express the type of relationship between UML elements (objects, classes etc).

Table shows the various relationships in UML

Relationship	Description	Representation
Association	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.	relationship →
Aggregation	It represents is "a part of" relationship. Represented by a line with a hollow diamond at the end.	a part of ◇
Composition <i>parts are deleted when the whole is deleted</i>	Aggregation with strong ownership relation to represent the component of a complex object. Represented by a line with a solid diamond at the end.	→ ◆
Generalisation	Represents a parent-child relationship. The parent may be more generalised and child being specialised version of the parent object.	Child ↓ Parent

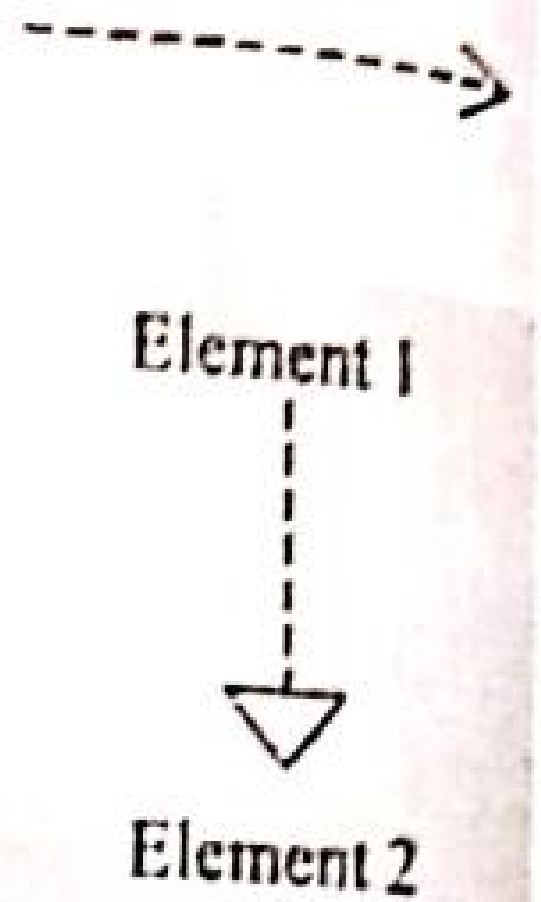
Dependency

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.

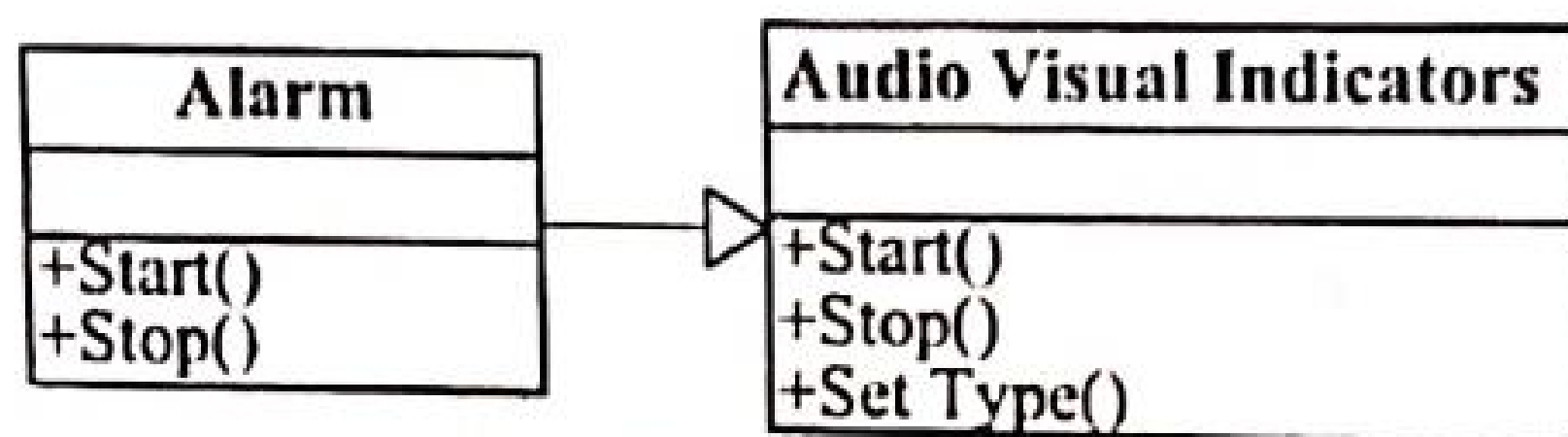
Realisation

The relationship between two elements in which one element realises the behaviour specified by the other element.

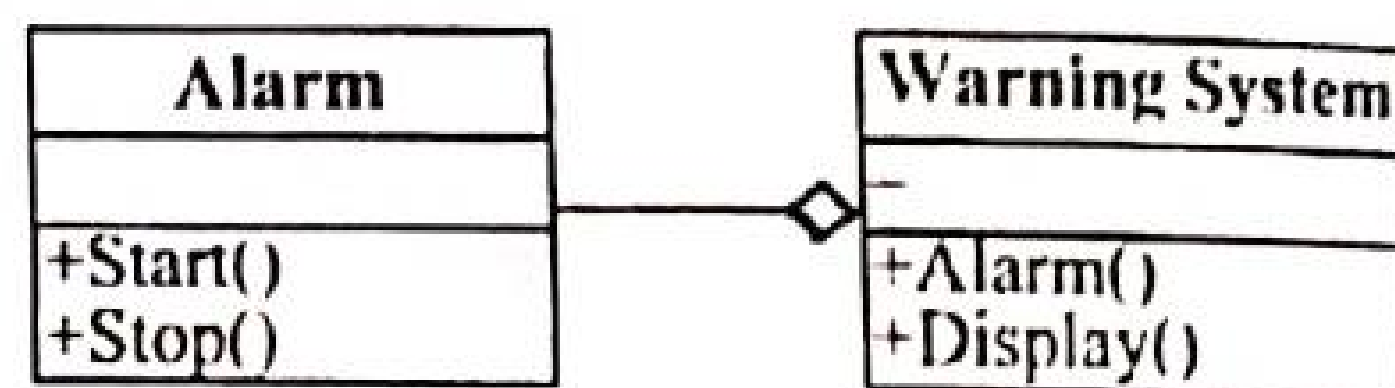
e.g. - the supplier & the client
one represents a specification & the other represents the implementation of that



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



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



③ UML Diagrams

give a pictorial representation of the static aspects, behavioural aspects and organisations and mgmt of diff^{nt} modules (classes, packages etc).

2 types < static diagrams
Behavioural "

static diagrams

- represents the structural (static) aspects of the system

- eg :- Object diagram, Component diagram, class diagram, Package diagram, deployment diagram

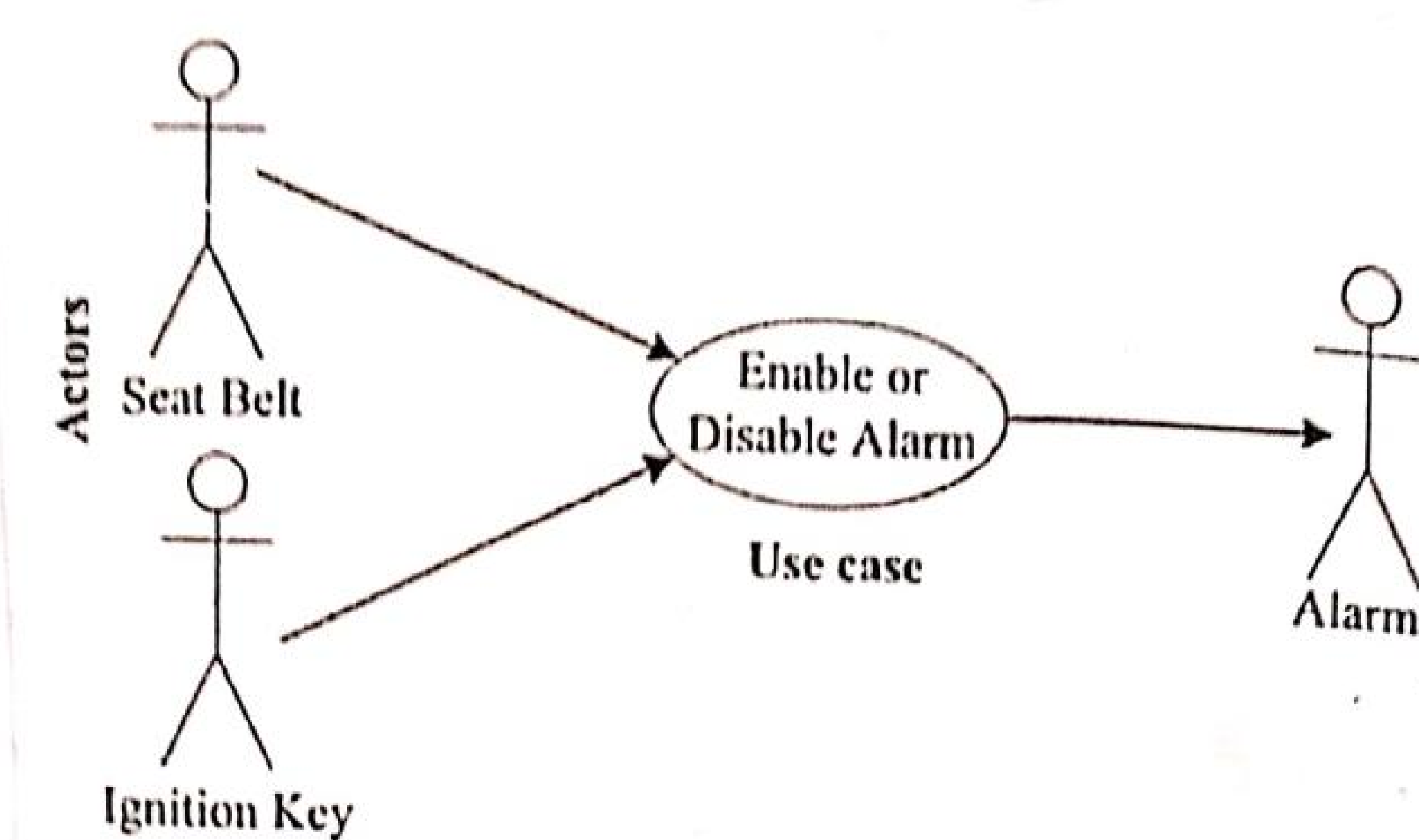
Diagram	Description
Object diagram	Gives a pictorial representation of a set of objects and their relationships. It represents the structural organisation between objects.
Class diagram	Gives a pictorial representation of the different classes in a UML model, their interfaces, the collaborations, interactions and relationship between the classes, etc. It captures the static design of the system.
Component diagram	It is a pictorial representation of the implementation view of a system. It comprises components (physical packaging of classes and interfaces), relationships and associations among the components.
Package diagram	It is a representation of the organisation of packages and their elements. Package diagrams are mostly used for organising use case diagrams and class diagrams.
Deployment diagram	It is a pictorial representation of the configuration of run time processing nodes and the components associated with them.

Behavioural diagrams

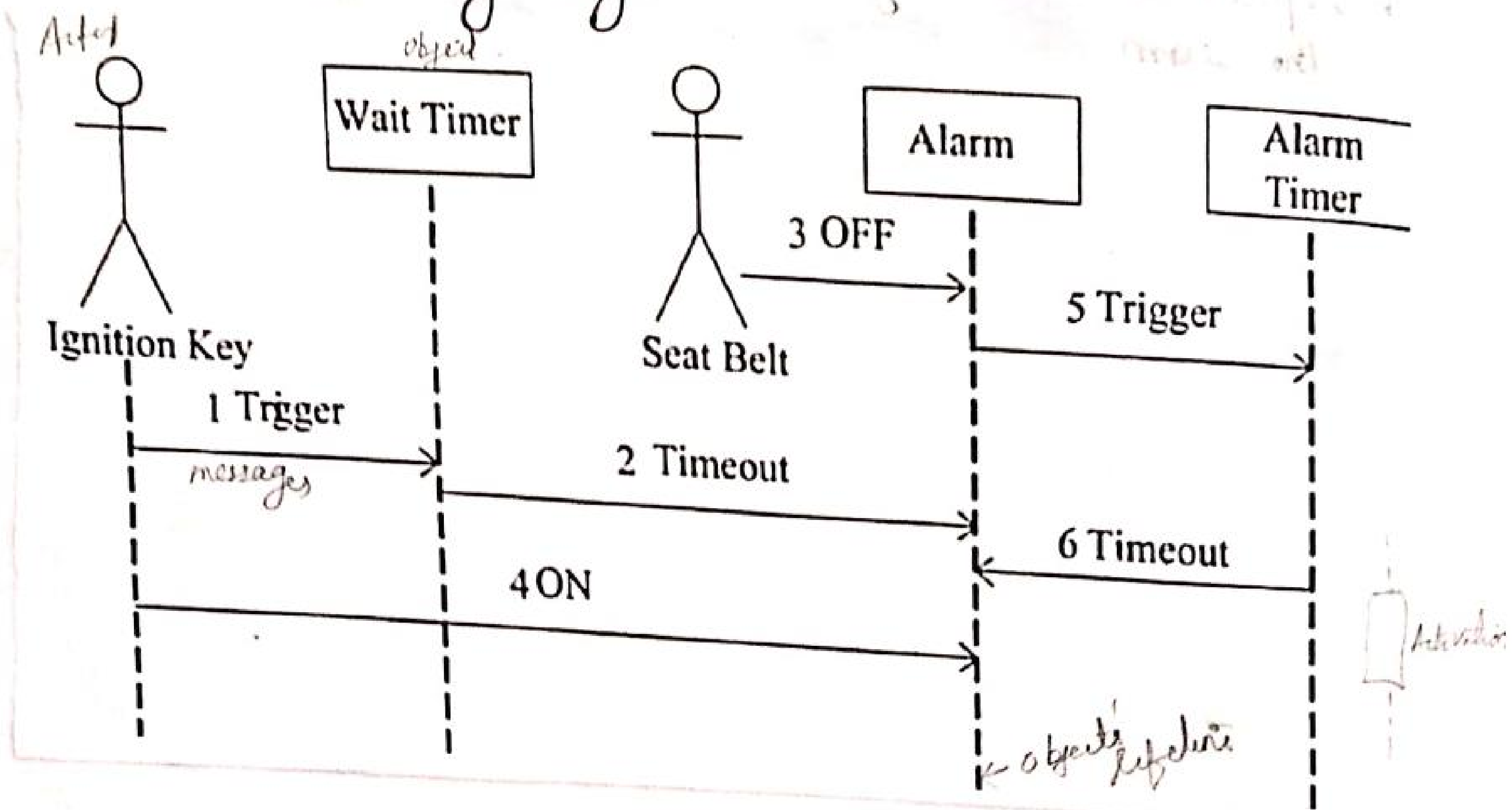
- represent the dynamic (behavioural) aspects of the system
- eg:- Use case dia., sequence dia., state dia., communication dia., Activity dia., Timing dia., interaction overview dia.

Diagram	Description
Use Case diagram <i><<include>> used when splitting a large use case</i> <i><<extends>> independent & extending use case</i>	Use Case diagrams are used for capturing system functionality as seen by users. It is very useful in system requirements capturing. Use case diagram comprise use cases, <i>actors</i> (users) and the relationship between them. In use case diagram, an <i>actor</i> is one (or something) who (or which) interacts with the system and use case is the sequence of interaction between the actor and system.
Sequence diagram	Sequence diagram is a type of interaction diagram representing object interactions with respect to time. It emphasises on the time ordering of messages. Best suited for the interaction modelling of real-time systems.
Collaboration (Comm) diagram	Collaboration or Communication diagram is a type of interaction diagram representing the <u>object interaction</u> and <u>'how they are linked together'</u> . It gives emphasis to the structural organisation of objects that send and receive messages. In short, it represents the <u>collaboration of objects using messages</u> .
State Chart diagram	A diagram showing the states, transitions, events and activities similar to a state machine representation. Best suited for modelling reactive systems.
Activity diagram	It is a special type of state chart diagram showing activity to activity transition in place of state transition. It emphasises on the flow control among objects.

Eg:- Use case diagram for seat belt warning sym



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



→ The UML Tools

- used for building UML based models
- available from different vendors
- can be $\begin{cases} \text{commercial} \\ \text{free / open source} \end{cases}$
- Some of the popular UML modelling tools are.

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

other tools

Eclipse
 UMLet
 AegouML
 Modelio
 Dia etc

Module IV

CS404: Embedded Systems.

Integration and Testing

Both embedded hardware and firmware should be independently tested to ensure their proper functioning. As far as the embedded firmware is concerned, its targeted functionalities can be easily be checked by simulator environment provided by the embedded firmware development tools.

Integration & Testing

Integration of hardware and firmware deals with the embedding of firmware into the hardware board. It is the process of 'Embedded Intelligence' to the product. The embedded processors/controllers used in the target board may or may not have built in code memory. For non-operating system based product, ~~if the~~ controller/processor contains internal memory and the ~~firmware~~ total size of firmware is fitting into the code memory or. If the processor doesn't support built in memory and external dedicated EPROM/FLASH memory chip is used for holding the firmware.

This chip is interfaced to the processor/controller.
* The type of firmware storage, either processor storage or external storage is decided at the time of hardware design by taking the firmware complexity into consideration.

The commonly used firmware embedded techniques for non-OS based systems are explained below.

1) Out-of-Circuit Programming

* This is Out of circuit programming is done by outside the target board. i.e. processor or memory chip into which the firmware needs to be embedded is taken out of the target board and it is programmed with the help of programming chip device.

* The programming chip is a dedicated board unit which contains the necessary h/w limits to generate the programming signals.

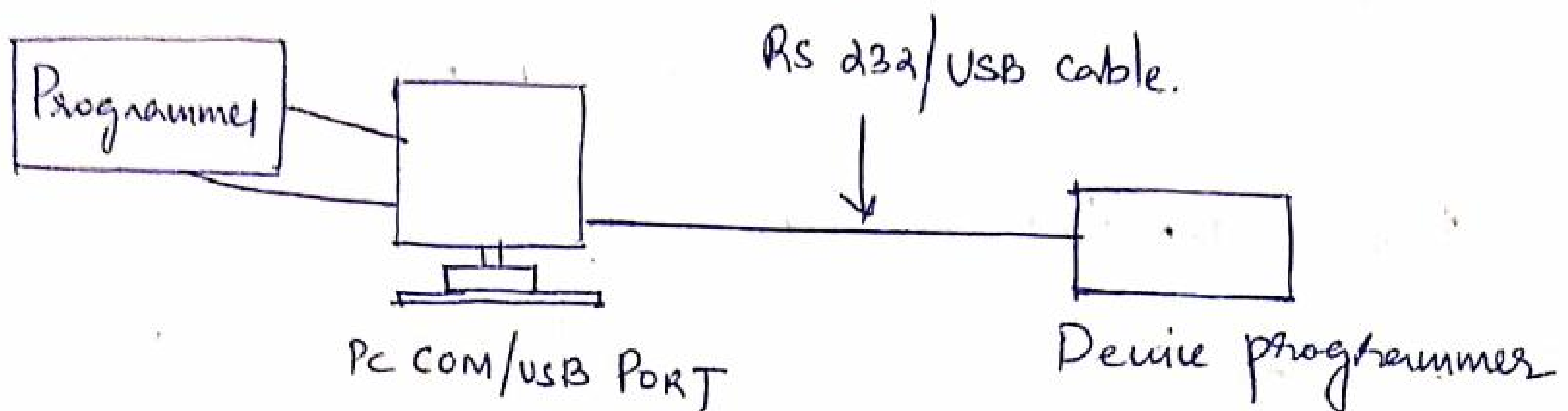
* Most of the programming devices available in the market are capable of programming different family of devices with pin cuts.

The sequence of operation for embedding the firmware with programmer is listed below: -

- Connect the device to the specified port
(USB/COM port/parallel port)
- Power up the device
- Execute the programming utility on the PC and ensure power connectivity is established b/w PC and programmer.
- Unlock the ZIF socket by turning the lock pin
- Insert the device to be programmed into the open socket.
- Lock the ZIF socket
- Select the device name from list of supported device.
- Load the hex file which is to be embedded.
- Program the device using program option.
- Wait till completion of programming & ensure that it is done successfully
- Unlock the ZIF socket and take the device out of programmer.

* The major drawback of out of circuit programming is high development time.

Whenever the firmware is changed, the chip should be taken out of the development board for re-programming.



2) In system Programming. (ISP)

* With ISP programming is done within the system i.e. firmware is embedded into the target device without removing it from target board.

* It is the most flexible and easy way of firmware embedding.

* The pre-requisite is apart from the target board the target must have an ISP support.
apart from target board, PC, ISP cable. e

* The target board can be interfaced to the utility program running on PC through serial port/parallel port USB, and the communication b/w ISP utility and target device will be in serial format

* In order to perform ISP operations the target device should be powered in a special ISP mode. which allows the device to communicate with an external host through serial interface, such as PC.

In system programming with SPI protocol.

Devices with In-system programming support contains a built in SPI interface and 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
MISO - Master in Slave out
SCK - System Clock
RST → Reset of Target device
GND - Ground of target device.

* PC acts as Master and target device acts as slave in ISP. The pgm data is sent to MOSI pin and device acknowledgement is originated from the MISO pin of device.

* SCK is a clock for data transfer.

In - Application Programming. (IAP).

* IAP is a technique used by firmware running on target device for modifying a selected portion of the code memory.

* It is not a first time embedding of user written firmware but it modifies the program code memory under the control of embedding application. Updating calibration data, look up table etc, which are stored in code memory.

* Boot ROM resident instructions which perform various ^{functions} instructions for programming, erasing and reading the Flash m/y are made available to the end user written firmware for IAP.

Use of Factory programmed chip.

* Embedding firmware into the target processor controller memory at the time of chip fabrication is known as 'Factory programmed chip' development.

* Once firmware design is over and firmware achieved operational stability, the files can be sent to chip fabricator to embed it into code m/y.

* This is convenient for mass production applications and reduces product development time.

* This is not suitable for development purpose where firmware undergoes frequent changes.

* These are little expensive.

Firmware loading for OS based devices.

* The OS based s/m are programmed using the In System Programming (ISP) technique. OS based s/m contains a special piece of code called Boot loader which takes the control of OS and application firmware embedding and copying of OS image to the RAM of the system for application

The Boot loader for such embedded s/m are pre-loaded or it can be loaded to the memory using various interface supported like JTAG.

* The Boot loader contains necessary driver implementation, menu option, for supported interfaces like UART/I²C, TCP/IP etc.

* In the case of network based loading, the boot loader broadcasts the target presence over the network and the host machine, on which OS image resides.

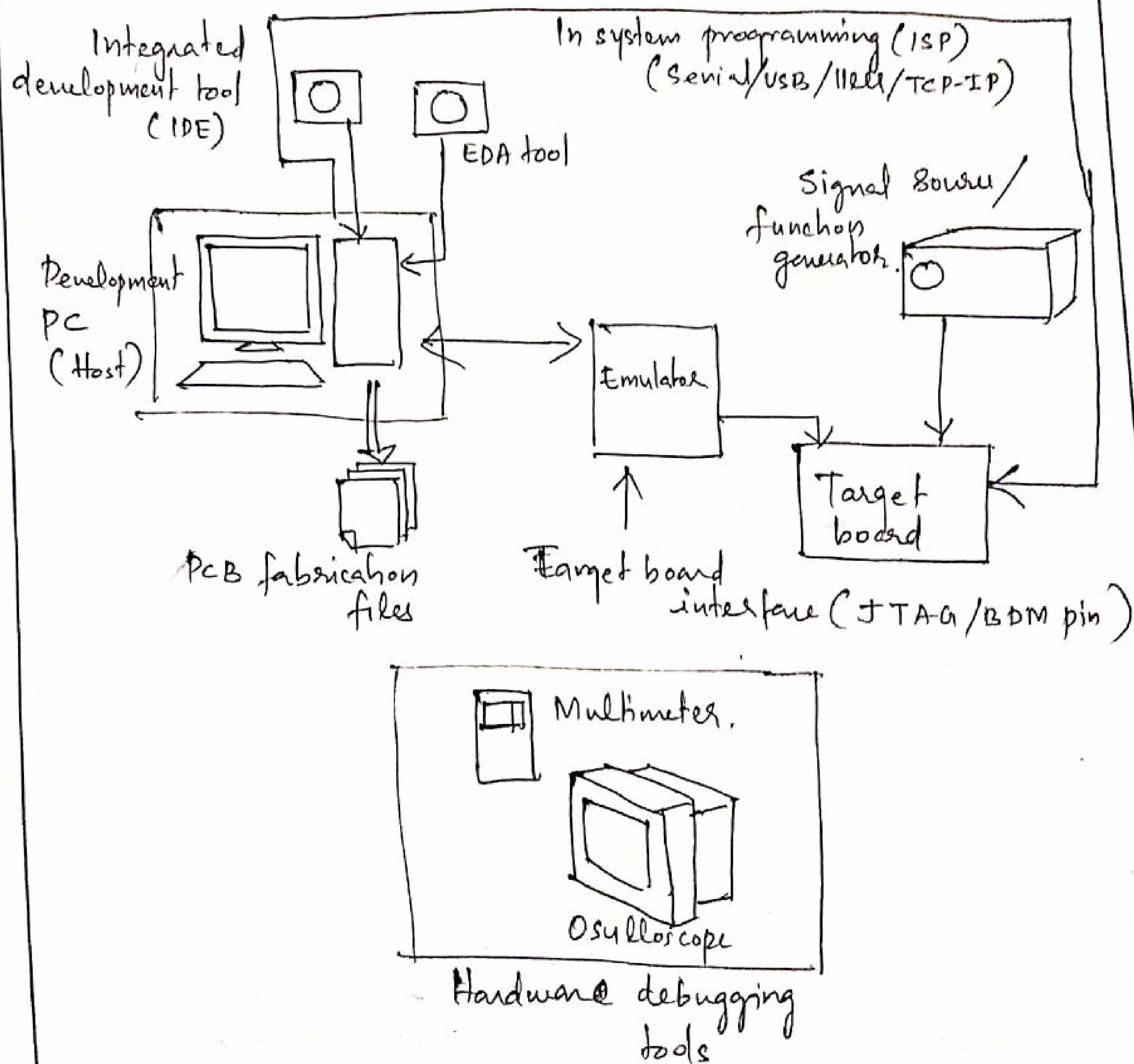
Board bring up.

Bring up of prototype or production is one of the most important step in embedded product design. The evaluation steps must pass through varied set of test to verify that the embedded hardware ~~work~~ and soft firmware work perfectly. The term bring up deals with performing series of validations in a controlled environment, in a well defined sequence.

Bring up procedure usually include basic

hardware spot check, checking power clock, functional connectivities, fetching code and execution etc.

The Embedded System Development Environment.



As illustrated in the figure, the development environment consists of a * development Computer (PC) or Host, which acts as the heart of development environment, * Integrated development tool for embedded firmware development, * Electronic design automation ^(EDA), tool for embedded hardware design, an emulator hardware for debugging target board, An emulator hardware for debugging target board, Signal sources like function generator, and target hardware debugging tools (CRO, multimeter etc).

EDA, and IDE tools are selected based target hardware development ~~requirements~~ requirement and supplied by 'Installable files'. These files can be installed on PC for development purposes.

The Integrated Development Environment (IDE)

* The Integrated development environment (IDE) stands for an integrated environment for developing and debugging the target processor.

Specific embedded applications firmware.

- * IDE is a software package which bundles a 'Text editor' — (Source code editor), Cross compiler (for cross platform development and compiler for same platform development), linker and ~~loader~~ debugger.

- * Some IDE's provide interface to target board emulators, flash memory programmer etc.

- * IDEs can be either command line based or GUI based. ~~or~~ Command line based IDEs may include less or little GUI support. GUI-based IDE support provide a visual development environment with user interaction through touch/mouse click interface.

Compiler & Cross compiler

Compiler is a software tool that converts source code written in a high level language on a top of a particular operating system running on a specific target processor architecture (eg, Intel pentium). The OS, compiler program

and the application making use of the source code run on the target processor

- * The source code is converted into target processor specific machine instructions. The development ^{is} platform specific. (OS and target processor on which OS is running). They are also termed 'native compilers'.

- * Cross compilers are the software tool used in cross platform development.

- * Here source code is converted to machine code for a target processor whose architecture and instruction set is different from current development environment OS.

- * Embedded system development is a typical example for cross-platform development, where an embedded firm is developed on a machine with Intel/AMD or any other target devices and same is converted into machine code for any other target processor architecture (~~in~~ 8051, PIC ARM etc.).

Disassemblers & Decompilers

* Disassembler is a utility program which converts machine code into target processor specific Assembly codes/instructions. This process of is known as Disassembling.

* In operation disassembling is complementary to assembling.

* Similarly decompiler is a utility program for translating machine codes into ~~correspond~~ corresponding high level language code. Decompiler performs the reverse operation of Compiler or Cross compiler.

* The disassemblers/decompilers for different family of processors/controllers are different.

These utilities are deployed in Reverse engineering. Reverse engineering is the process of revealing the technology behind the working of a product. This is used in ES product development to find out the secret behind the working of a product. These are also useful for analysing

the presence of malicious codes (virus information)

Simulators & Emulators

Simulator is a software tool used for simulating the various condition for checking the functionality of the application firmware. The IDE itself provide simulator support, and help in debugging. In certain scenarios the simulator provide a GUI model of ES product.

Emulator is a hardware device which mimics the target device and allows, real time debugging of ES firmware in a hardware environment.

The features of simulator based debugging.

1. Purely software based.
2. Doesn't require real target system.
3. Lack of I/O support
4. Lack of real time behaviour.

Advantages of simulator based debugging.

- * No need of original target board.

Purely Software Oriented. User only needs memory map of various devices. Saves development time.

- * Simulate I/O peripherals.

Simulator provides options to simulate various I/O peripherals. Using this we can edit I/O register values. It eliminates needs for connecting I/O devices for debugging firmware.

- * Simulate abnormal conditions.

Using simulation we can add any value for any parameter during debugging firmware and observe control flow. Which help to study the response of an ES product at abnormal conditions.

Limitations of ~~sim~~

- * Deviation from real behaviour :-
- * Lack of real timeliness

Debugging.

Debugging is the process of diagnosing the firmware execution, monitoring the target processor registers and memory while the firmware is running and checking the signals from various buses of embedded hardware.

This is classified as hardware debugging and ~~software~~ ^{firmware} debugging.

- * Hardware debugging deals with monitoring of various buses, and checking status of target processor.

- * Firmware debugging deals with examining the firmware flow, execution flow, examining CPU registers, status registers etc.

Different types of firmware debugging is explained below:-

Incremental EEPROM Burning technique.

- * The most primitive method.

- * The code is separated into different functional code units.

Page No. 7

- * Instead of burning entire code into EEPROM chip at once, the code is burned in incremental order.

- * The code corresponding to all functionalities are separately coded, cross compiled and burned one by one.

- * The code will incorporate some indication support like lighting up an LED or activating a buzzer etc.

- * If first functionality is found working perfectly go for burning code corresponding to next functionality and check whether it is working.

- * Repeat the process till all functionalities are covered.

- * It is a time consuming process, but a one-time process, and ~~go~~ can go for mass production.

Inline break point based firmware debugging.

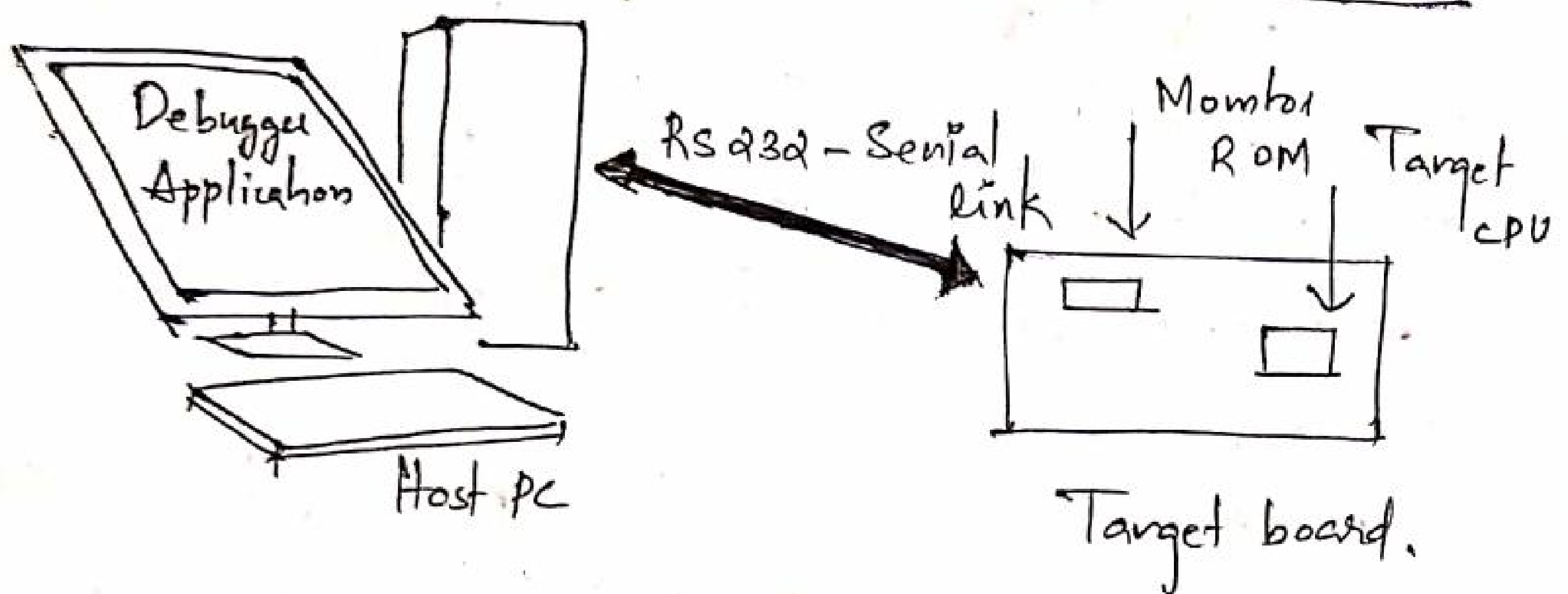
- * Within the firmware where you want to ensure the firmware execution reaching up to a specified point, insert an inline debug code ~~at~~ immediately after the point.

* The debug code is a 'printf()' function which prints a string given as per the firmware.

* The debug codes are cross-compiled with the source code.

* If the firmware is error free, we will get debug messages on Terminal program.

Monitor program based Firmware debugging.



* In this approach a monitor program which acts as a supervisor is developed.

* The monitor program controls the downloading of user code, into memory, inspects and modifies register/memory locations.

* The monitor program implements debug functions as per a pre-defined command set from the debug application interface.

* The monitor program always listens to the serial program ~~the~~ port of the target device a/c to command received from the serial interface

* The entire code stuff handling the command reception and corresponding action implementation is known as "monitor program"

* The most common type of interface used between target board and debug application is RS232/USB serial interface.

The monitor program contains following set of minimal features.

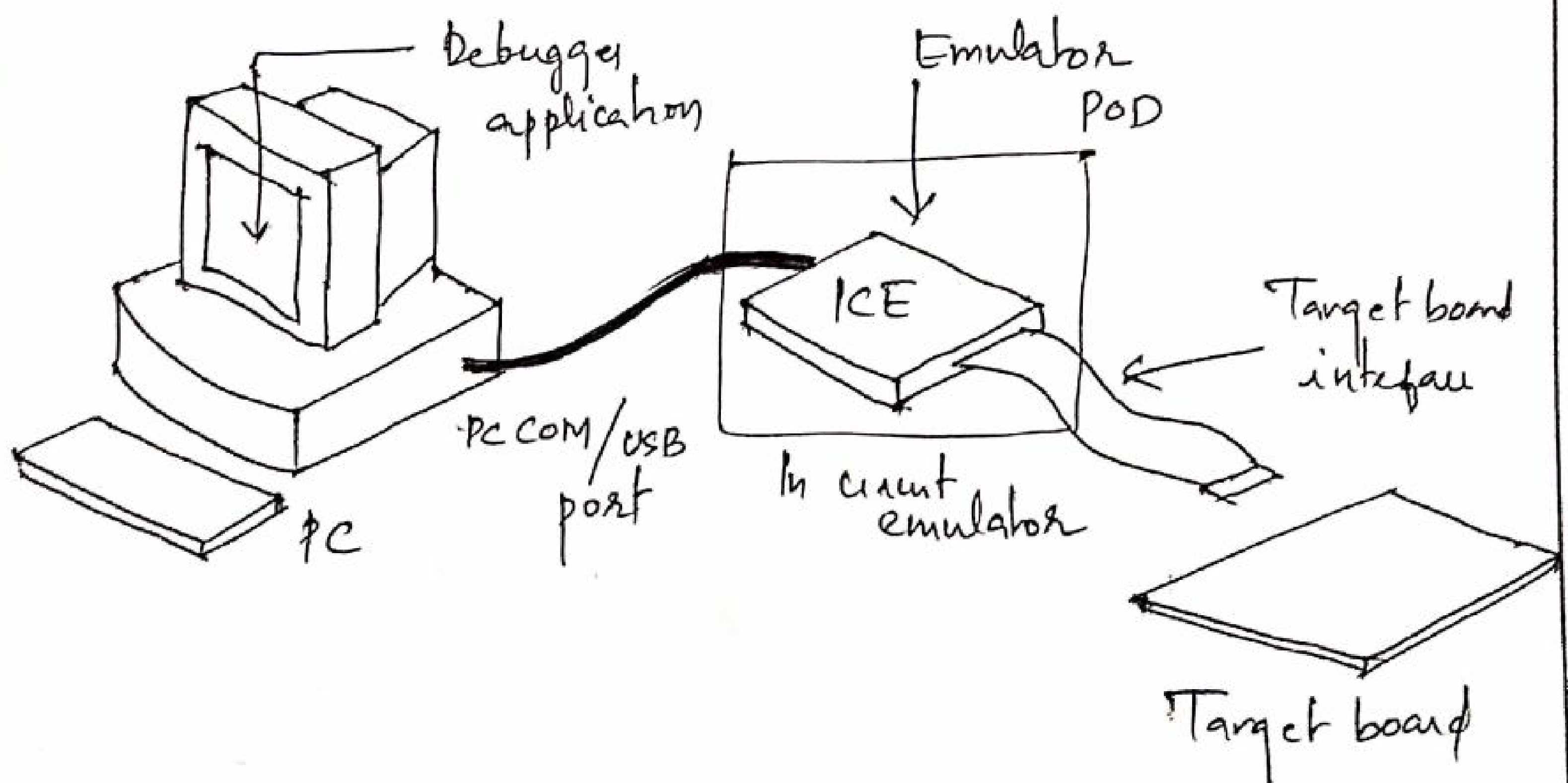
1. Command set interface to establish communication with debugging application.
2. Firmware download option to code memory.
3. Examine & modify processor registers and working memory.
4. Single step program execution
5. Set breakpoints in firmware execution
6. Send debug information to debug application running on host machine.

Drawbacks of monitor program based debugging:-

* The mem total memory is reduced to accomodate all kind of memory requi monitor program memory requirement such as monitor code, monitor data, etc.

* One serial port is used as a dedicated port for monitor program, wastage of port is happened.

In-circuit Emulator (ICE) based firmware debugging.



The emulator hardware application running
The in-circuit emulator based debugging

intelsx, a hardware emulator which is controlled by a debugger application running on the development PC.

* Emulator device is a replica of the target CPU which receives various signals from target board through a device adaptor connected to the target board and performs the execution of firmware under the control of debug commands from the debug application.

* The emulation device can be ^{either} a standard chip ~~or~~ same as target processor or a programmable logic device configured to a function as CPU. (PLD)

* If a standard chip is used emulation is dedicated to that particular device and can't be re-used for other chip, but it provides real time execution behaviour.

* PLD chips can be re-configured and re-use for different target processors, but accuracy of replication of target device is low.

* RAM memory is incorporated in Emulator device which is a replacement of EEPROM in target board.

* Emulator memory also act as a trace buffer holding instruction executed, register memory modified, by processor while debugging.

* Different logic units used for implementing control logic in emulator devices.

* Device adaptors ~~are used~~ acts as an interface b/w target board and emulator devices. They are normally pin to pin compatible sockets.

On-chip firmware debugging (OCD).

* Today almost all processors & debuggers make use of on-chip debug support (OCD)

* This is a fast and efficient firmware debugging method

* These facilities integrated to processor are chip dependant and uses background debug

model (BDM), and Joint Action Test Action group (JTAG) etc. There are two commonly used interfaces to communicate between debug applications running on development PC and OCD modules.

* OCD adds silicon complexity and cost factor but fast and efficient firmware debugging method.

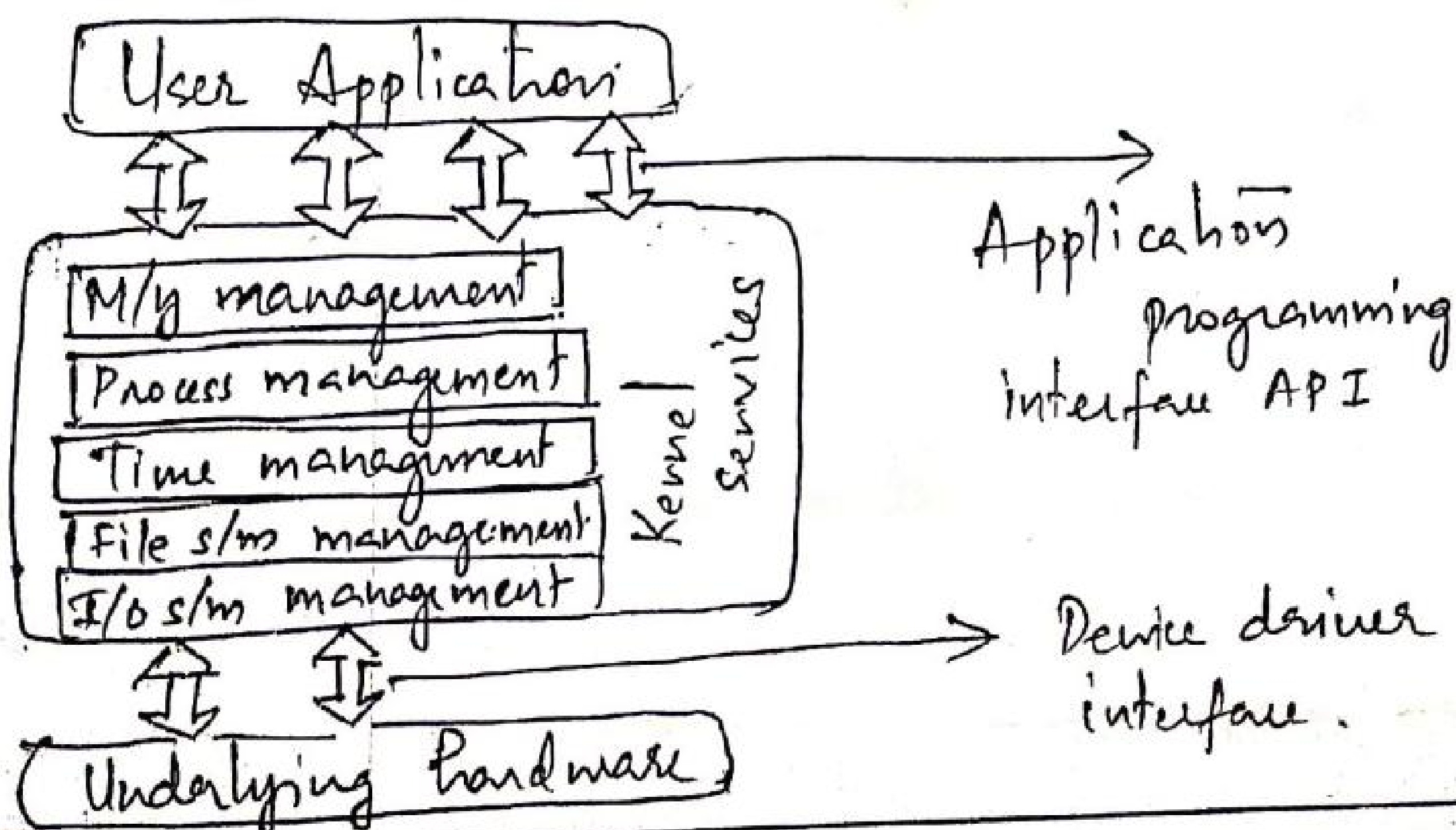
Operating system Basics

The operating system acts as a bridge between user application/tasks and the underlying system resources through a set of system functionalities and services.

The OS manages the system resources and makes them available to the user applications/tasks on a need basis. A normal computing system is a collection of different I/O subsystems, working and storage m/y.

The primary function of OS is:

- * Make the system convenient to use.
- * Organise the system and manage system resources efficiently and correctly.



The Kernel

The Kernel is the core of the OS and is responsible for managing the S/M resources and the communication among the hardware and other system services. Kernel acts as an abstraction layer between system resources and user applications. Kernel contains a set of system libraries and services. For a general purpose OS, Kernel contains different services for handling the following.

Process management

- * Managing processes/tasks
- * Setting up memory space for process, loading code into m/y space, allocating S/M resources
- * Scheduling and managing execution of the process
- * Setting up and managing the PCB (Process Control Block) etc.

Memory management.

* Primary memory management

Primary memory refers to volatile memory RAM. The kernel is responsible for

- * Keeping track of which part of m/y area is currently used by which process.

- * Allocating and de-allocating memory space on need basis (Dynamic m/y allocation).

* Secondary storage management

* Deals with managing secondary storage m/y devices, if any connected to the s/m.

* The secondary storage management deals with Disk storage allocation, Disk scheduling, Free disk space management.

* File s/m management.

* File is a collection of related information

* File could be text files, image files, audio/video files

* The file s/m management kernel responsible for

* The creation, deletion, and alteration of directories

* Saving of files in secondary m/y.

* providing automatic allocation of file space based on the amount of free space available

* Providing flexible naming convention for files.

* I/O system management

* Kernel is responsible for routing I/O requests coming from different user applications

* In a well structured OS, I/O devices are allowed through a set of Application Programming Interfaces

* The kernel maintain list of all I/O devices

Time management

* A real time clock (RTC) with regular time intervals are predefined by the number of RTC ticks. will cause s/m a timer interrupt.

* Time management provide proper synchronization for the every process.

* The kernel provides timer services such as delays, timeouts etc.

Types of Operating Systems -

Depending on the type of kernel and kernel services, purpose and type of computing s/m and the responsiveness of to application OS are classified as.

* General purpose OS (GPOS)

* Real Time OS (RTOS).

General Purpose OS (GPOS)

* General computing s/m are ~~referred~~ referred as GPOS.

* The Kernel is more generalised and contains all kind of services.

* Quite non-deterministic in behaviour, Random delays and slow responsiveness are occurred at unexpected times.

* GPOS ~~was~~ are usually deployed where deterministic behaviour is not an important criterion.

Real time OS

* Deterministic timing behaviour. i.e OS consumes only known and expected amount of time regardless the number of services.

* RTOS implements policies, rules ~~and~~ ^{and} for time critical ~~alloca~~ applications.

* Predictable performance can be achieved by the constant application of policies and rules.

* Windows Embedded Compact, QNX, Vxworks, MicroC/OS II etc are examples of RTOS.

RTOS are again classified according to the adherence to timing constraints, such as.

Hard Real time .

* RTOS ~~are~~ that strictly adhere to timing constraints.

* A hard real time must meet deadlines for a task without any slippage.

* Missing any deadline causes catastrophic result for hard real time system, including permanent data loss, and irrecoverable damages to s/ms.

* Airbag control s/ms, Antilock brake s/ms in vehicles are examples of hard real time s/ms.

* Hard real time s/ms doesnot implement the virtual m/y ~~for~~ for handling memory. This eliminates delay in swapping in and out code

* Most of the hard real time s/ms are automatic

Soft real time OS.

* RTOS doesnot ~~contain~~ guarantee meeting deadline

* Missing deadlines are acceptable within a compliance ~~result~~ limit of Quality of service.

* Automatic Teller Machine (ATM) is a soft real time s/ms.

* No potential damage occurs if the s/m doesn't meet deadlines.

The Real Time Kernel

- * The Kernel of a RTOS.
- * The RTOS Kernel is highly specialised and it contains only minimal set of services.
- * The basic functions of real time Kernel are listed below.

* Task process management

* Deals with setting up memory space for the task, loading tasks ~~to~~ code into memory space, allocating system resources, setting up task control block (TCB) for task termination and deletion. TCB is used for holding the information corresponding to a task such as Task ID, Task state, Task type, Task priority etc.

* Task process scheduling.

* Deals with sharing the CPU among various tasks/process

* An algorithm called scheduler handles task scheduling.

Task/process synchronisation :: Deals with concurrent access of resources, which shared across multiple tasks and communication between various tasks.

Error/Exception handling :-

* Deals with handling the errors occurred such as insufficient memory, timeouts deadlocks bus error etc.

* Deadlock is an example of kernel level exception, whereas timeout is an example of task level exception.

Memory management

* RTOS makes use of block based memory allocation technique, instead of usual dynamic m/y allocation.

* RTOS kernel uses blocks of fixed size of dynamic m/y and block is allocated for a task on a need basis.

Interrupt handling.

- * Deals with handling of various types of interrupt

- * Interrupt can be either synchronous or asynchronous. Usually software interrupts ~~are~~ fall under synchronous interrupt category. Divide by zero, memory segmentation error etc are ~~of~~ examples of synchronous interrupt.

- * Interrupt generated from external devices at any point of execution, at any task, and are not ~~is~~ sync with currently executing tasks are asynchronous interrupt. The interrupt generated by external devices are asynchronous interrupt.
eg: serial data reception timer over flow interrupts etc

- * Priority levels can be assigned to the interrupts and each interrupts can be enabled or disabled interrupts.

Time management

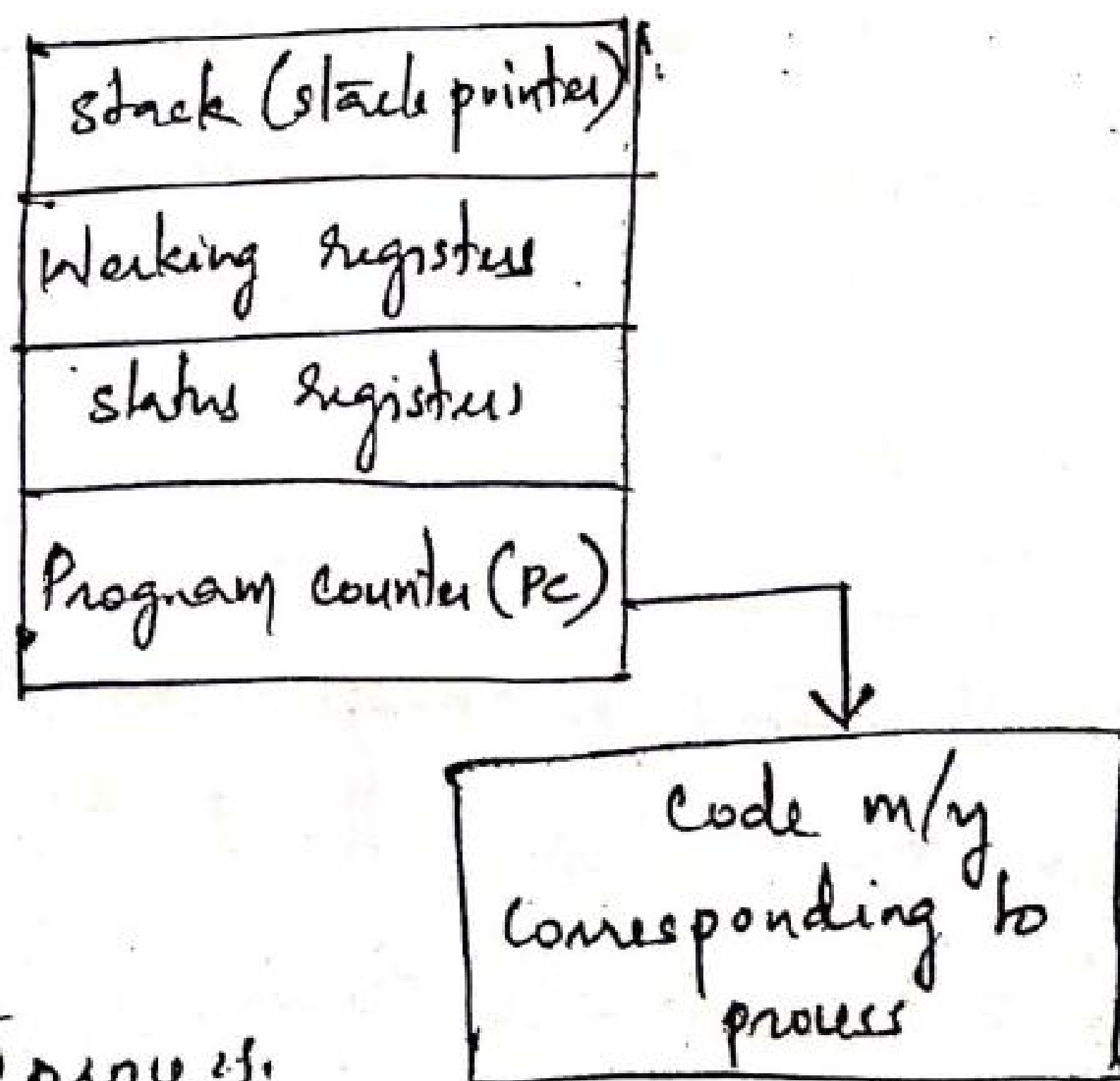
- * Accurate time management can be done using a high resolution Real time clock (RTC) hardware chip (hardware timer)

* The hardware timer is programmed to interrupt the processor/controller at a fixed rate, This is known as timer tick.

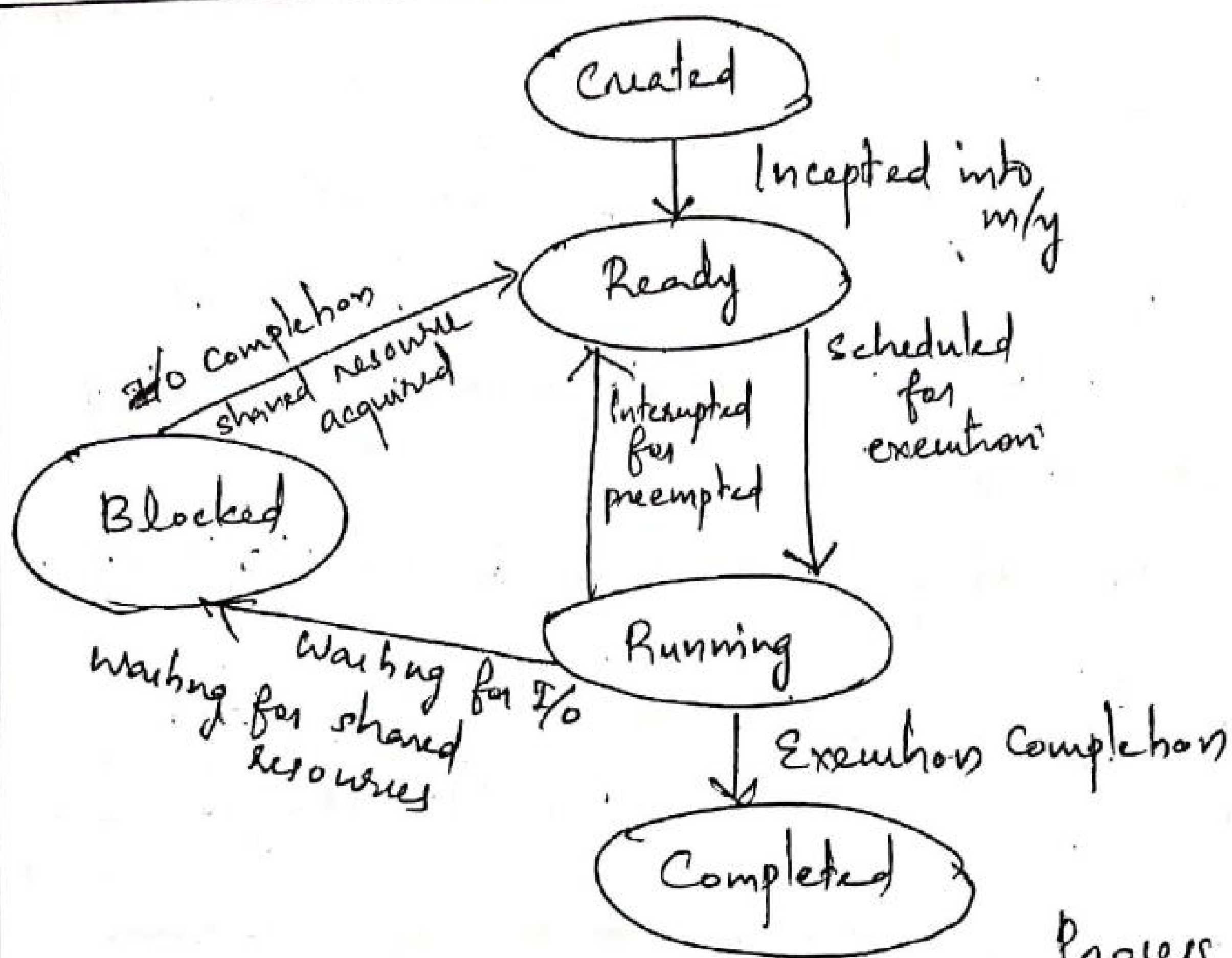
Tasks, Process and Threads

* The term task refers to something that need to be done. In OS context, task is defined as the program in execution and its related information maintained by the system.

* A process is program or part of it. Process is known as instance of a program in execution. Multiple instances of same program can execute simultaneously. A process is sequential in execution.



Structure of
a process.



Process states & state transitions

Threads

- * A thread is the primitive that can execute the code
- * A thread is a single sequential flow of control within the process
- * A thread can process can have many threads of execution. Different threads which are part of a process, share the same address space.
- * Thread maintain their own thread status program counter and stack

Multiprocessing & Multitasking

* multiprocessing describes the ability to execute multiple processes simultaneously, systems which are capable of multiprocessing is called multiprocessor s/m.

* Multiprocessing s/m include multiple CPUs

* The ability of a s/m to have multiple programs in m/y are called multiprogramming

* The ability of processor ~~to~~ achieve some degree of to hold multiple processors in memory and switch the processor from executing one process to another process is known as multitasking.

* Multitasking creates an illusion of multiple tasks executing in parallel.

* Whenever a CPU switching happens, the current context of execution should be saved to retrieve it at later point of time when CPU executes the process. The context saving and retrieval is essential for resuming a process exactly from the point where it is interrupted

due to CPU switching. The act of switching among the processors or changing current execution context details is known as Context switching.

* The act of saving current context for currently running process at the time of CPU switching is known as Context saving.

* The process of retrieving the saved context details is known as Context retrieval.

Types of multitasking.

* Multitasking involves switching of execution among multiple tasks.

Classified as,

* Co-operative multitasking.

* The task/process gets a chance to execute only when the currently executing task/process relinquishes CPU. In this method any task/process can hold the CPU as much as it wants.

*

Preemptive Multitasking.

- * every task/process gets a chance to execute when and how much time a process gets is dependant on the implementation of the preemptive scheduling.
- * Currently running task is pre-empted to give a chance to other tasks.

Non-preemptive Multitasking.

- * The process/task, is currently given the CPU time, is allowed to execute until it terminates or enters 'Blocked/wait' state, waiting for an I/O or system resources

- * In-cooperative multitasking, the currently executing process/task need not relinquish the CPU when it enters the 'blocked/wait' state, waiting an I/O or a shared resources access or an event to occur whereas in non-preemptive scheduling multitasking the currently executing task ~~relinquish~~ relinquishes the CPU, when it waits for an I/O or system resources or an event to occur

Task Scheduling.

* Determining which task/process is to be executed at a given point of time is known as task scheduling. Task scheduling forms the basis of multitasking.

* The kernel service application which implements the scheduling algorithm is known as 'Scheduler'.

The selection of a scheduling algorithm should always ~~make the~~ consider the following factors.

CPU utilisation :- A direct measure of how much percentage of CPU is being utilised.

Throughput :- Number of processes executed per unit time. Throughput of a good scheduler should always be high.

Turn around time :- Amount of time taken by a process for completing its execution. It includes the time spent by process for waiting for the req. main memory, time spent in ready queue, time spent on completing I/O operations and time spent in execution. The turn around time should be minimal for good scheduling algorithm.

Waiting time . It is the amount of time spent by a process in the Ready queue, waiting to get the CPU time for execution . It should be minimal .

Response time :- Time elapsed between the submission of a process and the first response . ~~For a go~~
The response time should be minimal for good Scheduler.

Non-preemptive scheduling.

* In this scheduling type, the currently executing task/process is allowed to run until it terminates or enters the 'wait' state for I/O s/m or resources. The various non-pre-emptive schedul are listed below.

First come First served (FCFS/FIFO) scheduling:

As name indicates this scheduling algorithm allocates CPU time to the processes based on the order in which they enter the Ready queue. The first entered process served first. eg; Ticket reservation system

Last come First Served (LCFS) scheduling.

LCFS algorithm also allocates CPU time to processes based on the order in which they are entered in Ready queue.

Shortest Job first (SJF).

This algorithm sorts the Ready queue each time a process relinquishes the CPU, to pick process with shortest estimated completion time.

Priority Based scheduling.

* The process with high priority is served serviced at the earliest compared to other low priority processes in Ready queue.

* The SJF algorithm can be viewed as a priority based scheduling where each task is prioritized in order of completion of task.

* Another way of priority assigning is associating priority to the task/process at the time of creation of task/process.

* The priority is number ranging from 0 to maximum priority supported by OS. The maximum level of priority is OS dependent.

Preemptive Scheduling.

* In pre-emptive scheduling, every task in ready queue gets a chance to execute. When and how often each process gets a chance to execute is dependent on that type of pre-emptive scheduling algorithm.

* The scheduler can stop temporarily (preempt) the currently executing task and select another task from Ready queue for execution.

* The act of moving a 'running' task into ready queue by scheduler, without requesting for it is known as 'preemption'.

* Preemptive SJF scheduling.

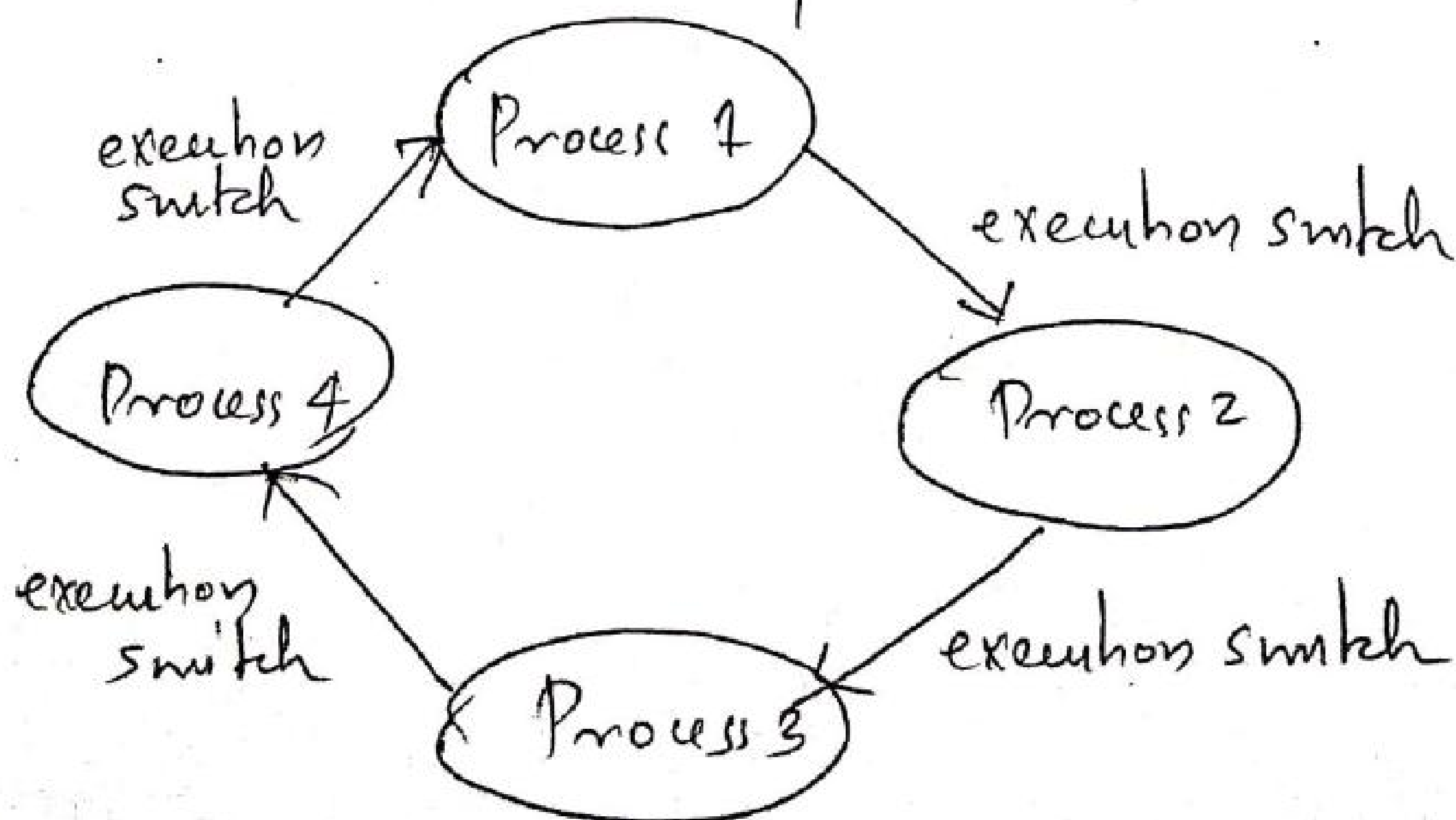
* In non-preemptive SJF sorts the Ready queue only after completing the execution of the current process or when the process enters 'Ready' queue and checks whether execution time of new process is shorter than remaining total estimated time of currently executing task.

* If execution time of new process is less

the currently executing ~~scheduled~~ process is pre-empted and new process is scheduled for execution. SJF always compares execution completion time of new process entered in the Ready queue with remaining time of currently executing process.

Round Robin scheduling.

This scheduling algorithm indicates ~~same message~~ 'equal chance to all'. In RR each process in ready queue is executed for a pre-defined time slot. The execution starts with picking up first process in Ready queue. It is executed for a predefined time and when it elapses or completes, the next task is selected. RR is similar to FCFS and only difference is that a time slice based pre-emption is added to switch execution between processes.



priority based scheduling.

* Same as that of non-preemptive priority based scheduling except for switching of execution b/w tasks.

Interrupt Routines in RTOS environment and handling of Interrupt source calls.

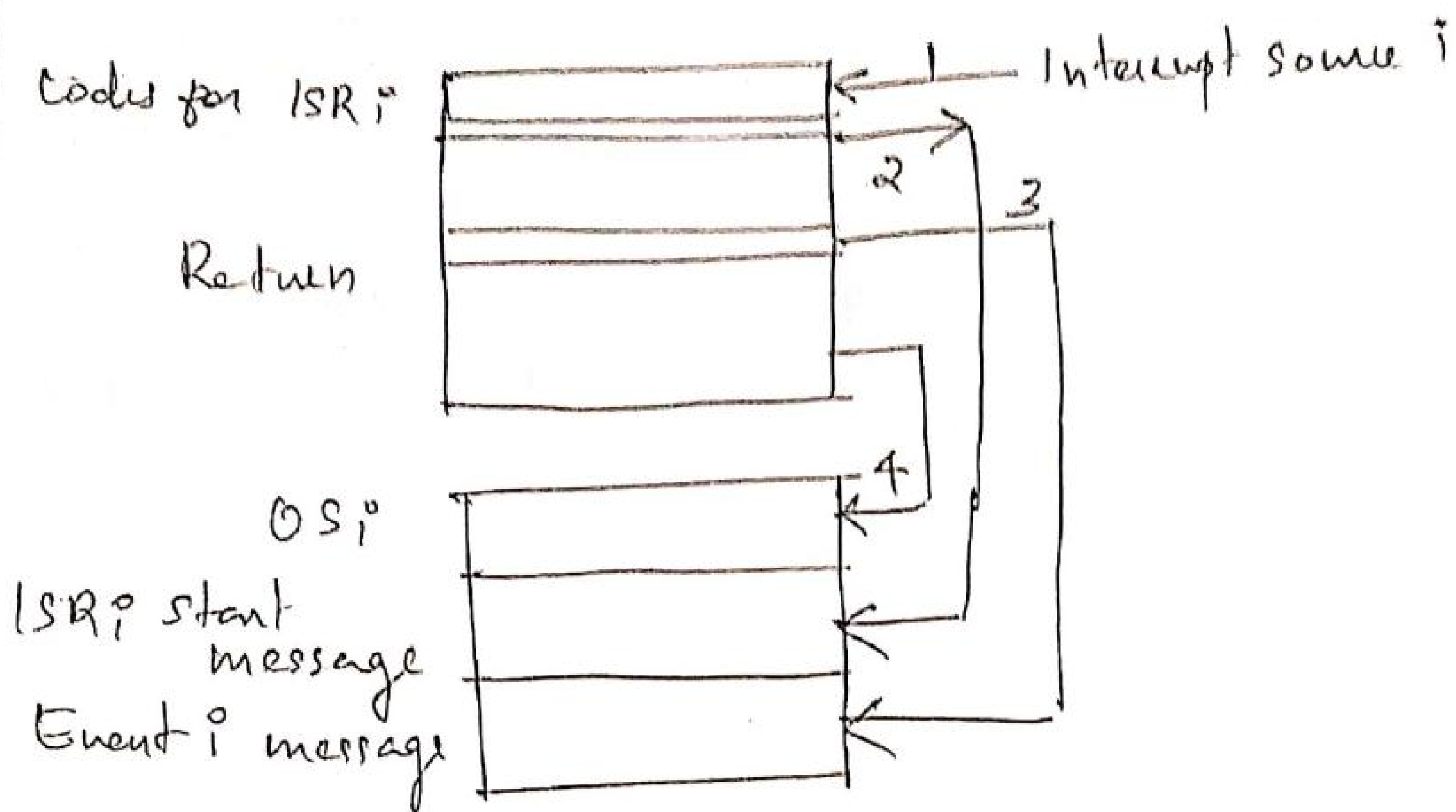
An RTOS or OS has an interrupt servicing mechanism. Following sections explain their alternatives for responding to interrupt.

Direct call to an ISR by an Interrupt Source.

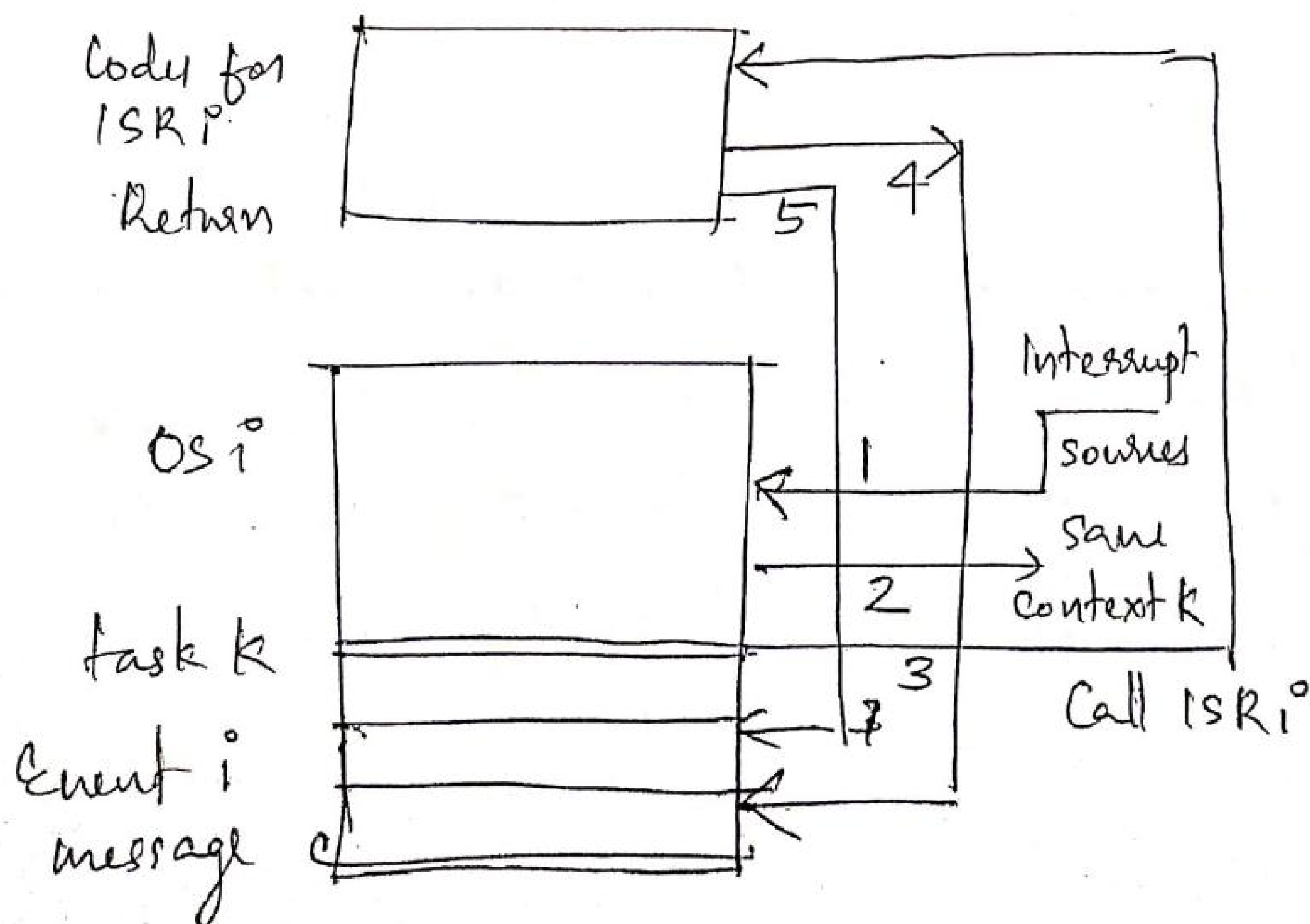
- * When interrupt occurs, the process running at CPU interrupts. Context switching takes place directly to the ISR.

- * An i^{th} interrupt source leads to switch to i^{th} ISR. 'ISR- i ' just sends an ISR enter message at start for the RTOS.

- * Later the ISR code can post an IPC. Task waiting for that IPC does not start before the return from the ISR. The ISR enter message is to inform the RTOS that an ISR has taken control of the CPU.



RTOS ISR_i interrupting on an interrupt then RTOS calling the corresponding ISR_i .



* When a task is interrupted on i^{th} interrupt source then context first switches to RTOS.

* The RTOS for h/w interrupt, switches the context to ISR- i . The ISR during execution then can send post one or more IPC for tasks.

* Context switches back to RTOS on return from ISR- i . Context then switches from RTOS to pending interrupt ISR or highest priority task.

RTOS First interrupts, calls to corresponding ISR then ISR sending messages to Interrupt service

Threads

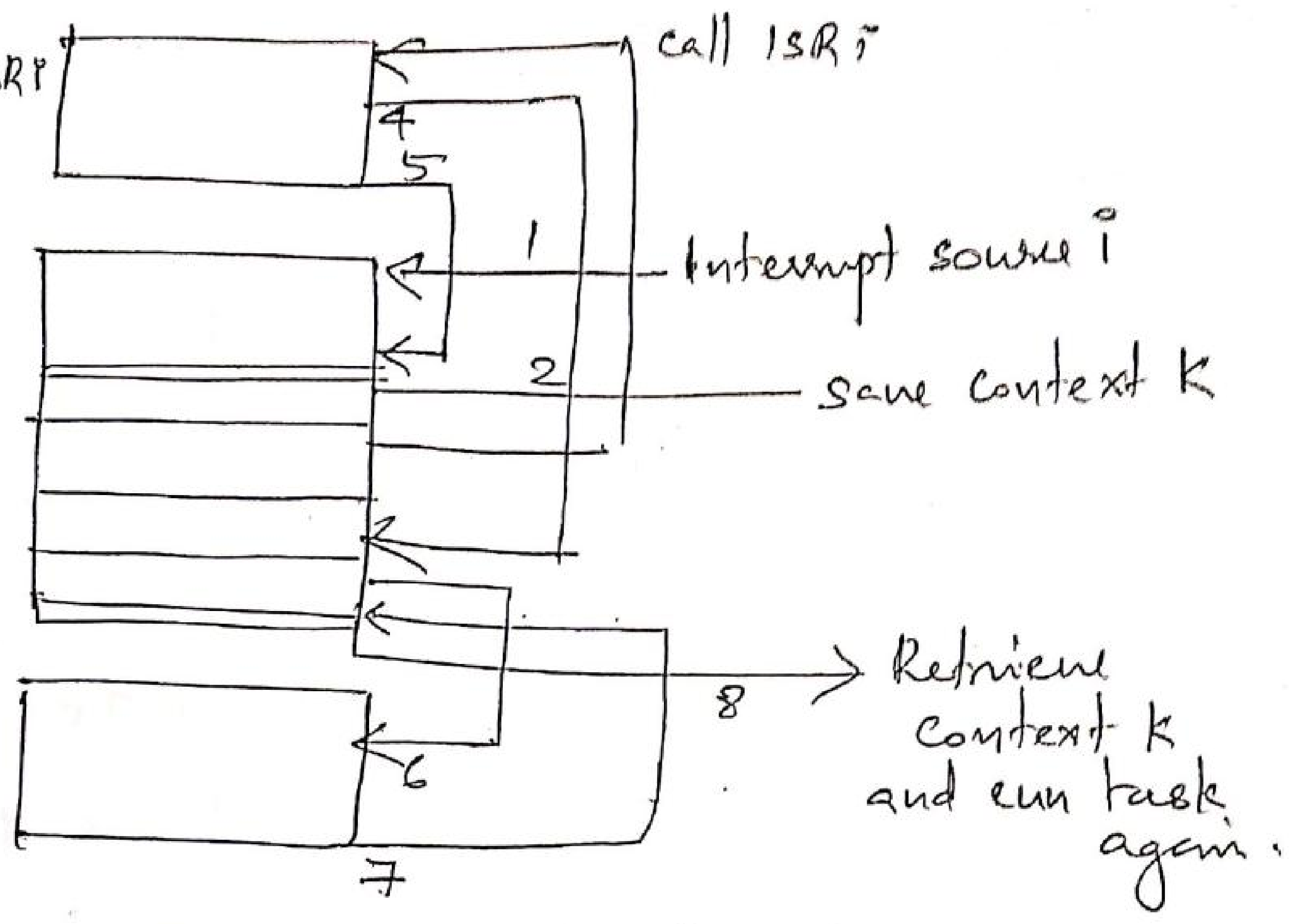
* Two level of interrupt service routines

* FLISR (First level ISR)

* SLISR (Slow level ISR)

* FLISR ~~is~~ reduces interrupt latency (waiting period) for interrupt service.

Code for ISR
return
OS;



How to choose an RTOS

A brief introduction of functional and non-functional requirements that needs to be analysed in selection of RTOS is explained below.

Functional Requirements

Processor support:- Not necessary that all RTOS support all kind of processor architecture. It is essential to ensure processor support by the RTOS.

Memory Requirement : OS require ROM m/y for holding OS files and it is normally stored in non-volatile m/y like FLASH. OS also require working m/y RAM. Minimal ROM and RAM requirement are recommended.

Real time capabilities :- Task/process scheduling policies plays an important role in real time behaviour.

Kernel & Interrupt latency :- The kernel of OS may disable interrupts while executing certain services and it may do interrupt latency. The latency should be minimal.

Inter process communication and task synchronisation

* This is OS kernel dependant, certain kernel provide a bunch of options and implement policies for priority inversion issue in resource sharing.

Modularisation support :- Developer can choose essential modules and re-compile OS image for functioning.

Support for networking & communication

* This include driver support for communication interfaces & networking.

Development language support Certain OS includes the run time libraries required for running application written in languages like Java & C

Non functional support

* Custom Developed or Off the shelf

* Indicates either an open source product or commercial product. This is purely dependant on development cost, licencing fees for the OS

Cost :- The total cost for developing ~~and~~ or buying and maintaining in terms of commercial product is evaluated

Development & debugging Tools availability

Ease of use :