# PLC SYSTEMS

## LECTURE NOTES

## B.TECH IV YEAR – I SEM (2020-21)


**Prepared by:**

**Mr. M Ramanjaneyulu, Associate Professor,**

**Department of Electronics and Communication Engineering**



ESTD: 2004

MRCET CAMPUS

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

(AUTONOMOUS INSTITUTION - UGC, GOVT. OF INDIA)

**MALLA REDDY COLLEGE OF ENGINEERING AND TECHNOLOGY**

**IV Year B.Tech. EEE- I Sem**

**PLC SYSTEMS**

**COURSE OBJECTIVES:**

For programmable logic controllers, the course will enable the students

- To provide and ensure a comprehensive understanding of using advanced controllers in measurement and control instrumentation.
- To analyze Programmable Logic Controller (PLC), IO Modules and internal features, Programming in Ladder Logic.
- Understand the core of an embedded system
- To learn the design process of embedded system applications.
- To understands the RTOS and inter-process communication.
- To understand different communication interfaces.

**UNIT-I**

**PLC BASICS:** PLC system, I/O modules and interfacing, CPU processor, programming Equipment, programming formats, construction of PLC ladder diagrams, Devices connected to I/O modules. PLC Programming: Input instructions, outputs, operational procedures, programming examples using contacts and coils. Drill press operation.

**UNIT-II**

**LADDER DIAGRAMS FOR PROCESS CONTROL**: Ladder diagrams and sequence listings, ladder diagram construction and flowchart for spray process system.

**PLC REGISTERS:** Characteristics of Registers, module addressing, holding registers, Input Registers, Output Registers.

**UNIT-III**

**INTRODUCTION TO EMBEDDED SYSTEMS:** Complex systems and microprocessors-embedding computers, characteristics of embedded computing applications, challenges in embedded computing system design, performance in embedded computing; The embedded system design process-requirements, specification, architecture design, designing hardware and software, components, system integration.

**UNIT-IV**

**TYPICAL EMBEDDED SYSTEM:** Core of the embedded system-general purpose and domain specific processors, ASICs, PLDs, COTS; Memory-ROM, RAM, memory selection for embedded systems; Sensors and actuators , Onboard communication interfaces-I2C, SPI. Embedded Firmware Design and Development: Embedded firmware design approaches-super loop based approach, operating system based approach; embedded firmware development languages-assembly language based development, high level language based development.

**UNIT-V**

**RTOS BASED EMBEDDED SYSTEM DESIGN:** Operating system basics, types of operating systems, tasks, process and threads, multiprocessing and multitasking, task scheduling: non-preemptive and pre-emptive scheduling, How to choose an RTOS.

**TEXT BOOKS:**

1. Programmable Logic Controllers- Principles and Applications by John W. Webb and Ronald A. Reiss, Fifth Edition, PHI.

2. Computers as Components –Wayne Wolf, Morgan Kaufmann (second edition).

3. Introduction to Embedded Systems - shibu k v, Mc Graw Hill Education.

**REFERENCE BOOKS:**

1. Programmable Logic Controllers- Programming Method and Applications by JR.Hackworth and F.D Hackworth Jr., Pearson, 2004.

2. Embedded Systems- An integrated approach - Lyla b das, Pearson education 2012.

**COURSE OUTCOMES:** After going through this course the student will be able to

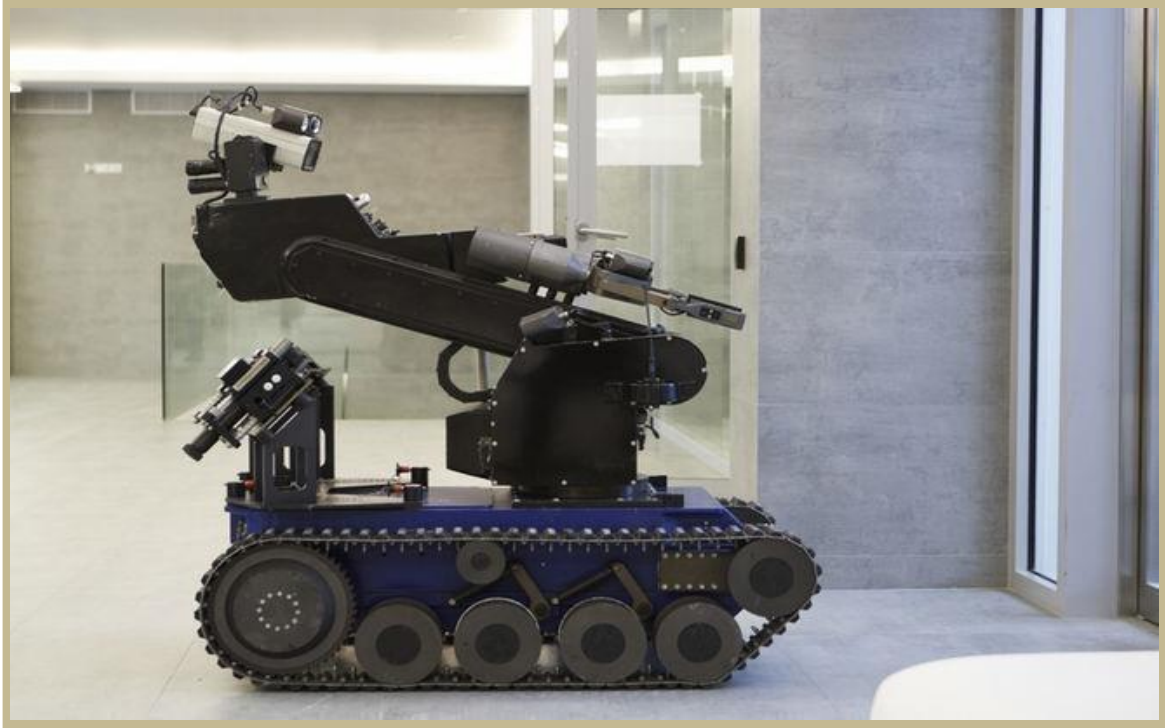
- Describe the main functional units in a PLC and be able to explain how they interact
- Develop ladder logic programming for simple process.
- Understand and design the embedded systems
- Understand Embedded Firmware design approaches
- Learn the basics of RTOS

# PLC SYSTEMS

# IV YEAR EEE

## UNIT-3

## INTRODUCTION TO EMBEDDED SYSTEMS



MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY
Permanently Affliated to JNTUH and Approved by AICTE, New Delhi

# 1. Introduction to Embedded Systems

**What is Embedded System?**

An Electronic/Electro mechanical system which is designed to perform a specific function and is a combination of both hardware and firmware (Software)

E.g. Electronic Toys, Mobile Handsets, Washing Machines, Air Conditioners, Automotive Control Units, Set Top Box, DVD Player etc…

**Embedded Systems are:**

- ❑ Unique in character and behavior
- ❑ With specialized hardware and software

## Embedded Systems Vs General Computing Systems:

| General Purpose Computing System | Embedded System |
|---|---|
| A system which is a combination of generic hardware and General Purpose Operating System 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 |
| Contain a General Purpose Operating System (GPOS) | May or may not contain an operating system for functioning |
| Applications are alterable (programmable) by user (It is possible for the end user to re-install the Operating System, and add or remove user applications) | The firmware of the embedded system is pre-programmed and it is non-alterable by end-user |
| Performance is the key deciding factor on the selection of the system. Always „Faster is Better" | Application specific requirements (like performance, power requirements, memory usage etc) are the key deciding factors |
| Less/not at all tailored towards reduced operating power requirements, options for different levels of power management. | Highly tailored to take advantage of the power saving modes supported by hardware and Operating System |
| Response requirements are not time critical | For certain category of embedded systems like mission critical systems, the response time requirement is highly critical |
| Need not be deterministic in execution behavior | Execution behavior is deterministic for certain type of embedded systems like „Hard Real Time" systems |

## COMPLEX SYSTEMS AND MICROPROCESSORS:

Embedded system: Any device that includes a programmable computer but is not itself a General-Purpose Computer.

*An Electronic/Electro mechanical system which is designed to perform a specific function and is a combination of both hardware and firmware (Software)*

Take advantage of application characteristics to optimize the design:

- Don't need all the general-purpose bells and whistles.
- PC is not itself an embedded computing system,
- PCs are often used to build embedded computing systems.

But a fax machine or a clock built from a microprocessor is an embedded computing system

- Embedded computing system design is a useful skill for many types of product designs.
- Automobiles, cell phones, and even household appliances make extensive use of microprocessors.
- Designers in many fields must be able to identify where microprocessors can be used,
- Design a hardware platform with I/O devices that can support the required tasks, and implement software that performs the required processing.

**Embedding a computer:** An embedded computer is an integral component of most Embedded Systems, is a combination of hardware and software that is designated to perform a highly specific function.

- For example, the type of embedded computer in a washing machine will not be the same as the embedded computer in a Nikon camera.
- Because the software in embedded computers is designed to only execute certain tasks, the computer's software in one device can be totally distinct from that of another.
- First, the word "embedded" implies that the computer must be contained in a larger mechanical or electronic system.
- Automobile designers started making use of the microprocessor soon after single-chip CPUs became available.
- The most important and sophisticated use of microprocessors in automobiles was to control the engine, determining when spark plugs fire, controlling the fuel/air mixture, and so on.
- There was a trend toward electronics in automobiles in general—electronic devices could be used to replace the mechanical distributor.
- But the big push toward microprocessor-based engine control came from two nearly simultaneous developments:
- The oil shock of the 1970s caused consumers to place much higher value on fuel economy, and fears of pollution resulted in laws restricting automobile engine emissions.
- The combination of low fuel consumption and low emissions is very difficult to achieve; to meet these goals without compromising engine performance, automobile manufacturers turned to sophisticated control algorithms that could be implemented only with microprocessors.

## EMBEDDED SYSTEM DESIGN PROCESS:

This section provides an overview of the embedded system design process aimed at two objectives. First, it will give us an introduction to the various steps in embedded system design before we delve into them in more detail. Second, it will allow us to consider the design methodology itself. A design methodology is important for three reasons. First, it allows us to keep a scorecard on a design to ensure that we have done everything we need to do, such as

optimizing performance or performing functional tests. Second, it allows us to develop computer-aided design tools. Developing a single program that takes in a concept for an embedded system and emits a completed design would be a daunting task, but by first breaking the process into manageable steps, we can work on automating (or at least semi automating) the steps one at a time. Third, a design methodology makes it much easier for members of a design team to communicate.

The below Figure summarizes the major steps in the embedded system design process. In this top–down view, we start with the system requirements.

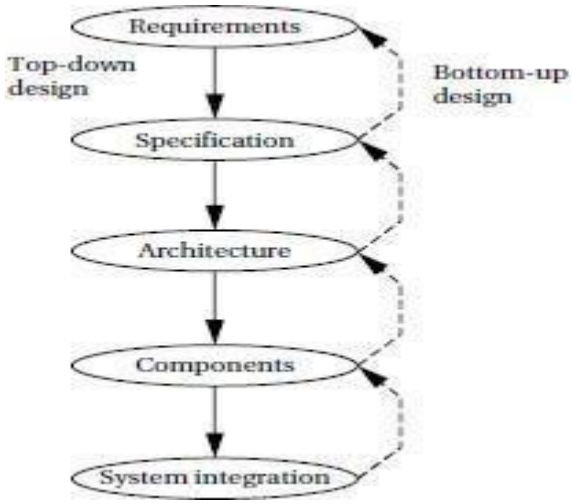



**Fig: Major levels of abstraction in the design process**

## Requirements:

Clearly, before we design a system, we must know what we are designing. The initial stages of the design process capture this information for use in creating the architecture and components. We generally proceed in two phases: First, we gather an informal description from the customers known as requirements, and we refine the requirements into a specification that contains enough information to begin designing the system architecture.

Requirements may be ***functional*** or ***nonfunctional***. We must of course capture the basic functions of the embedded system, but functional description is often not sufficient. Typical nonfunctional requirements include:

■ *Performance:* The speed of the system is often a major consideration both for the usability of the system and for its ultimate cost. As we have noted, performance may be a combination of soft performance metrics such as approximate time to perform a user-level function and hard deadlines by which a particular operation must be completed.

■ *Cost:* The target cost or purchase price for the system is almost always a consideration. Cost typically has two major components: ***manufacturing cost*** includes the cost of components and assembly; ***nonrecurring engineering*** **(NRE)** costs include the personnel and other costs of designing the system.

■ *Physical size and weight:* The physical aspects of the final system can vary greatly depending upon the application. An industrial control system for an assembly line may be designed to fit into a standard-size rack with no strict limitations on weight. A handheld device typically has tight requirements on both size and weight that can ripple through the entire system design.

■ *Power consumption:* Power, of course, is important in battery-powered systems and is often important in other applications as well. Power can be specified in the requirements stage in terms of battery life—the customer is unlikely to be able to describe the allowable wattage.

A sample ***requirements form*** that can be filled out at the start of the project. We can use the form as a checklist in considering the basic characteristics of the system. Let's consider the entries in the form:

■ *Name:* This is simple but helpful. Giving a name to the project not only simplifies talking about it to other people but can also crystallize the purpose of the machine.

■ *Purpose:* This should be a brief one- or two-line description of what the system is supposed to do. If you can't describe the essence of your system in one or two lines, chances are that you don't understand it well enough.

■ *Inputs and outputs:* These two entries are more complex than they seem. The inputs and outputs to the system encompass a wealth of detail:

— *Types of data:* Analog electronic signals? Digital data? Mechanical inputs?

— *Data characteristics:* Periodically arriving data, such as digitalaudio samples? Occasional user inputs? How many bits per data element?

— *Types of I/O devices:* Buttons? Analog/digital converters? Video displays?

■ *Functions:* This is a more detailed description of what the system does. A good way to approach this is to work from the inputs to the outputs: When the system receives an input, what does it do? How do user interface inputs affect these functions? How do different functions interact?

*Performance:* Many embedded computing systems spend at least some time controlling physical devices or processing data coming from the physical world. In most of these cases, the computations must be performed within a certain time frame. It is essential that the performance requirements be identified early since they must be carefully measured during implementation to ensure that the system works properly.

■ *Manufacturing cost:* This includes primarily the cost of the hardware components. Even if you don't know exactly how much you can afford to spend on system components, you should have some idea of the eventual cost range. Cost has a substantial influence on architecture: A machine that is meant to sell at $10 most likely has a very different internal structure than a $100 system.

■ *Power:* Similarly, you may have only a rough idea of how much power the system can consume, but a little information can go a long way. Typically, the most important decision is whether the machine will be battery powered or plugged into the wall. Battery-powered machines must be much more careful about how they spend energy.

■ *Physical size and weight:* You should give some indication of the physical size of the system to help guide certain architectural decisions. A desktop machine has much more flexibility in the components used than, for example, a lapel mounted voice recorder.

**GPS MODULE:**


## REQUIREMENTS FORM OF GPS MOVING MAP MODULE:

**Name : GPS moving map**
**Purpose: Consumer-grade moving map for driving use**
**Inputs : Power button, two control buttons**
**Outputs : Back-lit LCD display 400 _ 600**
**Functions : Uses 5-receiver GPS system; three user-selectable resolutions; always displays current latitude and longitude**
**Performance: Updates screen within 0.25 seconds upon movement**
**Manufacturing cost:$30**
**Power: 100mW**
**Physical size and weight: No more than 2" _ 6," 12 ounces**

### Specification

The specification is more precise—it serves as the contract between the customer and the architects. As such, the specification must be carefully written so that it accurately reflects the customer's requirements and does so in a way that can be clearly followed during design.
The specification should be understandable enough so that someone can verify that it meets system requirements and overall expectations of the customer.
A specification of the GPS system would include several components:

- Data received from the GPS satellite constellation.
- Map data.
- User interface.
- Operations that must be performed to satisfy customer requests.
- Background actions required to keep the system running, such as operating the GPS receiver.

### Architecture Design

The specification does not say how the system does things, only what the system does. Describing how the system implements those functions is the purpose of the architecture. The architecture is a plan for the overall structure of the system that will be used later to design the components that make up the architecture. The creation of the architecture is the first phase of what many designers think of as design.

This block diagram is still quite abstract—we have not yet specified which operations will be performed by software running on a CPU, what will be done by special-purpose hardware, and so on. The diagram does, however, go a long way toward describing how to implement the functions described in the specification. We clearly see, for example, that we need to search the topographic database and to render (i.e., draw) the results for the display. We have chosen to separate those functions so that we can potentially do them in parallel—performing rendering separately from searching the database may help us update the screen more fluidly.

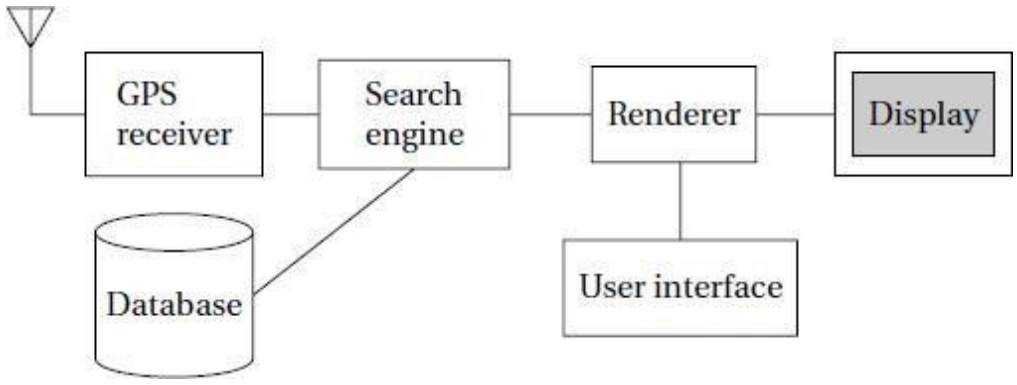



FIG: BLOCK DIAGRAM FOR THE MOVING MAP

The hardware block diagram clearly shows that we have one central CPU surrounded by memory and I/O devices. In particular, we have chosen to use two memories: a frame buffer for the pixels to be displayed and a separate program/data memory for general use by the CPU. The software block diagram fairly closely follows the system block diagram, but we have added a timer to control when we read the buttons on the user interface and render data onto the screen. To have a truly complete architectural description, we require more detail, such as where units in the software block diagram will be executed in the hardware block diagram and when operations will be performed in time.

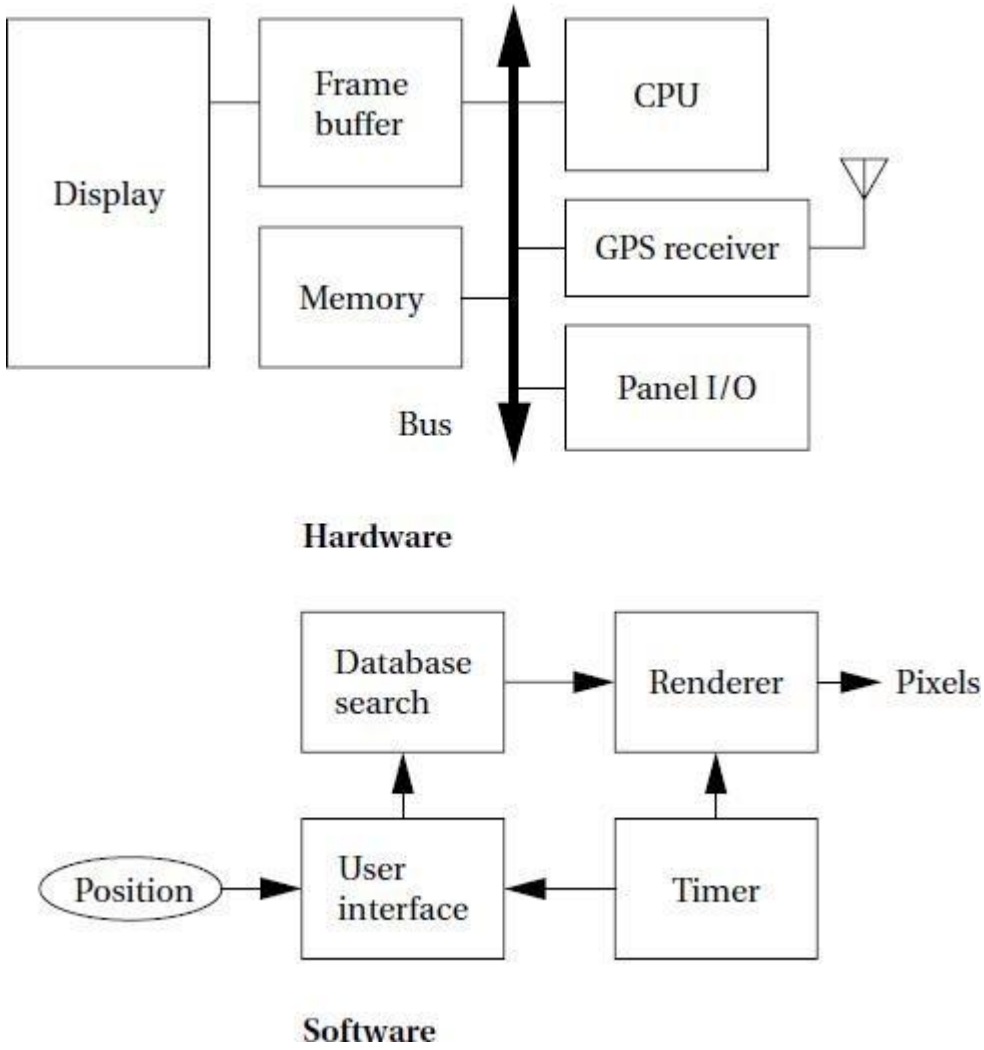



Fig : Hardware and software architectures for the moving map.

The architectural description tells us what components we need. The component design effort builds those components in conformance to the architecture and specification. The components will in general include both hardware—FPGAs, boards, and so on—and software modules. Some of the components will be ready-made. The CPU, for example, will be a standard component in almost all cases, as will memory chips and many other components .In the moving map, the GPS receiver is a good example of a specialized component that will nonetheless be a predesigned, standard component. We can also make use of standard software modules.

## System Integration:

Only after the components are built do we have the satisfaction of putting them together and seeing a working system. Of course, this phase usually consists of a lot more than just plugging everything together and standing back. Bugs are typically found during system integration, and good planning can help us find the bugs quickly. By building up the system in phases and running properly chosen tests, we can often find bugs more easily. If we debug only a few modules at a time, we are more likely to uncover the simple bugs and able to easily recognize them. Only by fixing the simple bugs early will we be able to uncover the more complex or obscure bugs that can be identified only by giving the system a hard workout

## Characteristics of Embedded systems:

Embedded systems possess certain specific characteristics and these are unique to each Embedded system.

1. Application and domain specific
2. Reactive and Real Time
3. Operates in harsh environments
4. Distributed
5. Small Size and weight
6. Power concerns
7. Single-functioned
8. Complex functionality
9. Tightly-constrained
10. Safety-critical

### 1. Application and Domain Specific:-

- Each E.S has certain functions to perform and they are developed in such a manner to do the intended functions only.
- They cannot be used for any other purpose.
- Ex – The embedded control units of the microwave oven cannot be replaced with AC‟S embedded control unit because the embedded control units of microwave oven and AC are specifically designed to perform certain specific tasks.

### 2. Reactive and Real Time:-

- E.S are in constant interaction with the real world through sensors and user-defined input devices which are connected to the input port of the system.
- Any changes in the real world are captured by the sensors or input devices in real time and the control algorithm running inside the unit reacts in a designed manner to bring the controlled output variables to the desired level.
- E.S produce changes in output in response to the changes in the input, so they are referred as reactive systems.
- Real Time system operation means the timing behavior of the system should be deterministic ie the system should respond to requests in a known amount of time.

- Example – E.S which are mission critical like flight control systems, Antilock Brake Systems (**ABS**) etc are Real Time systems.

**3. Operates in Harsh Environment :–**

- The design of E.S should take care of the operating conditions of the area where the system is going to implement.
- Ex – If the system needs to be deployed in a high temperature zone, then all the components used in the system should be of high temperature grade.
- Also proper shock absorption techniques should be provided to systems which are going to be commissioned in places subject to high shock.

**4. Distributed: –**

- It means that embedded systems may be a part of a larger system.
- Many numbers of such distributed embedded systems form a single large embedded control unit.
- Ex – Automatic vending machine. It contains a card reader, a vending unit etc. Each of them are independent embedded units but they work together to perform the overall vending function.

**5. Small Size and Weight:-**

- Product aesthetics (size, weight, shape, style, etc) is an important factor in choosing a product.
- It is convenient to handle a compact device than a bulky product.

**6. Power Concerns:-**

- Power management is another important factor that needs to be considered in designing embedded systems.
- E.S should be designed in such a way as to minimize the heat dissipation by the system.

**7. Single-functioned:-** Dedicated to perform a single function

**8. Complex functionality: -** We have to run sophisticated algorithms or multiple algorithms in some applications.

**9. Tightly-constrained:-**

Low cost, low power, small, fast, etc

**10. Safety-critical:-**

Must not endanger human life and the environment

## Challenges in embedded system design:

### 1. How much hardware do we need?

- A great deal of control over the amount of computing power apply to our problem.
- We cannot only select the type of microprocessor used, but also select the amount of memory, the peripheral devices, and more.
- Since we often must meet both performance deadlines and manufacturing cost constraints,
- The choice of hardware is important—too little hardware and the system fails to meet its deadlines, too much hardware and it becomes too expensive.

### 2. How do we meet our deadlines?

- Faster hardware or cleverer software?
- The brute force way of meeting a deadline is to speed up the hardware so that the program runs faster. Of course, that makes the system more expensive.
- It is also entirely possible that increasing the CPU clock rate may not make enough difference to execution time, since the program's speed may be limited by the memory system

### 3. How do we minimize power?

- Turn off unnecessary logic?
- In battery-powered applications, power consumption is extremely important.
- Even in non-battery applications, excessive power consumption can increase heat dissipation.
- One way to make a digital system consume less power is to make it run more slowly, but naively slowing down the system can obviously lead to missed deadlines.
- Careful design is required to slow down the noncritical parts of the machine for power consumption while still meeting necessary performance goals

### 4. How do we design for upgradability?

- The hardware platform may be used over several product generations, or for several different versions of a product in the same generation, with few or no changes.
- However, we want to be able to add features by changing software.
- How can we design a machine that will provide the required performance for software that we haven't yet written?

### 5. Does it really work?

- Reliability is always important when selling products—customers rightly expect that products they buy will work.

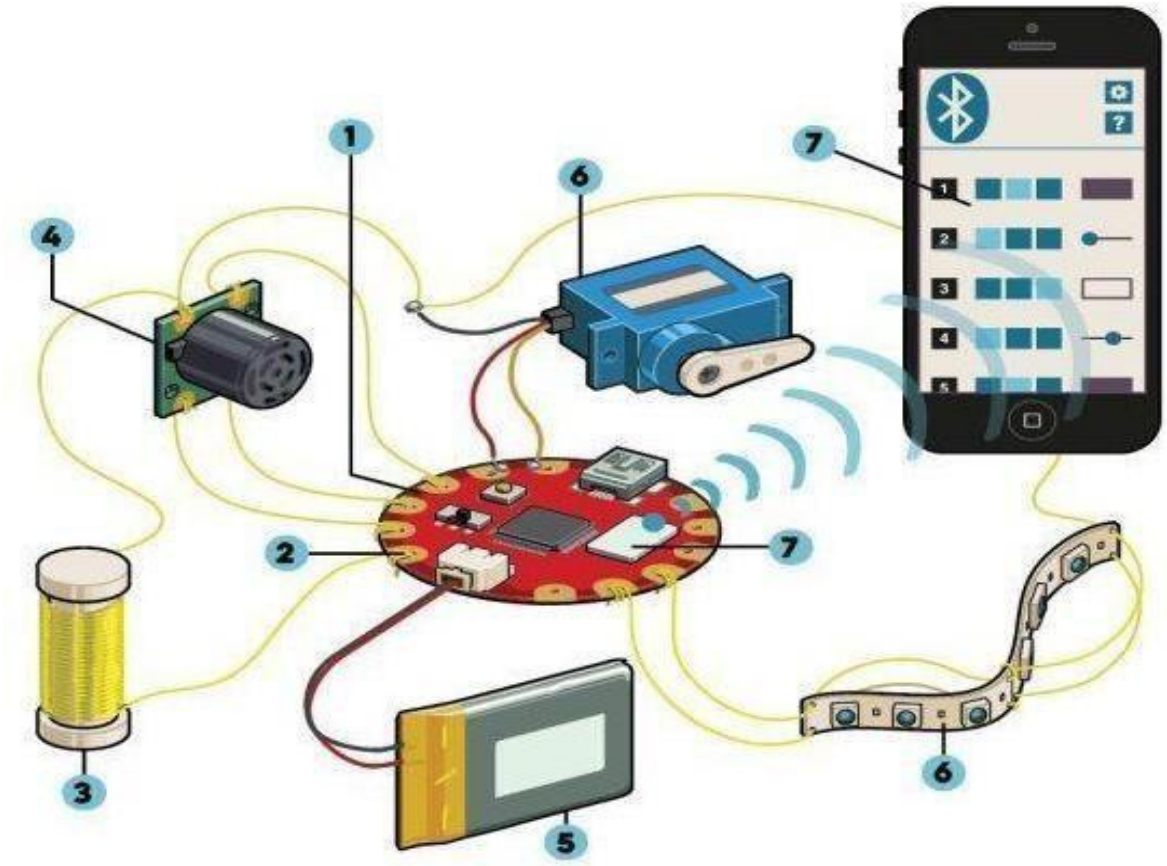
- Reliability is especially important in some applications, such as safety-critical systems.
- Another set of challenges comes from the characteristics of the components and systems
- Let’s consider some ways in which the nature of embedded computing machines makes their design more difficult.
- Complex testing : How do we test for real-time, real data?
- Limited observability and controllability : Limited user interface.
- Restricted development environments : The development environments for embedded systems (the tools used to develop software and hardware) are often much more limited than those available for PCs and workstations.

# IV EEE

# PLC SYSTEMs

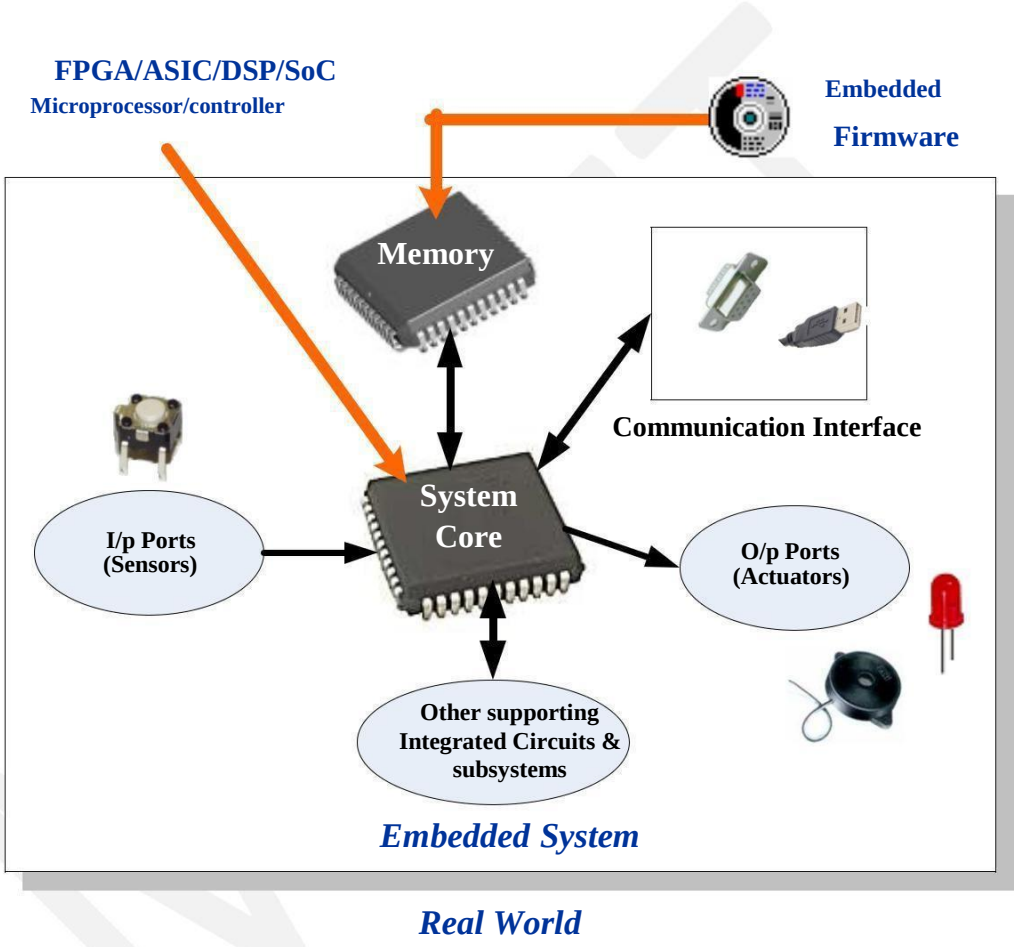
# UNIT-IV

## TYPICAL EMBEDDED SYSTEM



## ELEMENTS OF EMBEDDED SYSTEMS:

An embedded system is a combination of 3 things, Hardware Software Mechanical Components and it is supposed to do one specific task only. A typical embedded system contains a single chip controller which acts as the master brain of the system. Diagrammatically an embedded system can be represented as follows:




Embedded systems are basically designed to regulate a physical variable (such Microwave Oven) or to manipulate the state of some devices by sending some signals to the actuators or devices connected to the output port system (such as temperature in Air Conditioner), in response to the input signal provided by the end users or sensors which are connected to the input ports. Hence the embedded systems can be viewed as a reactive system.

The control is achieved by processing the information coming from the sensors and user interfaces and controlling some actuators that regulate the physical variable.

Keyboards, push button, switches, etc. are Examples of common user interface input devices and LEDs, LCDs, Piezoelectric buzzers, etc examples for common user interface output devices for a typical embedded system. The requirement of type of user interface changes from application to application based on domain.

Some embedded systems do not require any manual intervention for their operation. They automatically sense the input parameters from real world through sensors which are connected at input port. The sensor information is passed to the processor after signal conditioning and digitization. The core of the system performs some predefined operations on input data with the help of embedded firmware in the system and sends some actuating signals to the actuator connect connected to the output port of the system.

The memory of the system is responsible for holding the code (control algorithm and other important configuration details). There are two types of memories are used in any embedded system. Fixed memory (ROM) is used for storing code or program. The user cannot change the firmware in this type of memory. The most common types of memories used in embedded systems for control algorithm storage are OTP,PROM,UVEPROM,EEPROM and FLASH

An embedded system without code (i.e. the control algorithm) implemented memory has all the peripherals but is not capable of making decisions depending on the situational as well as real world changes.

Memory for implementing the code may be present on the processor or may be implemented as a separate chip interfacing the processor

In a controller based embedded system, the controller may contain internal memory for storing code such controllers are called Micro-controllers with on-chip ROM, eg. Atmel AT89C51.

**The Core of the Embedded Systems:** The core of the embedded system falls into any one of the following categories.

- **General Purpose and Domain Specific Processors**
  - Microprocessors
  - Microcontrollers
  - Digital Signal Processors

- **Programmable Logic Devices (PLDs)**
- **Application Specific Integrated Circuits (ASICs)**
- **Commercial off the shelf Components (COTS)**

## GENERAL PURPOSE AND DOMAIN SPECIFIC PROCESSOR:

Almost 80% of the embedded systems are processor/ controller based.

The processor may be microprocessor or a microcontroller or digital signal processor, depending on the domain and application.

### Microprocessor:

A silicon chip representing a Central Processing Unit (CPU), which is capable of performing arithmetic as well as logical operations according to a pre-defined set of Instructions, which is specific to the manufacturer

In general the CPU contains the Arithmetic and Logic Unit (ALU), Control Unit and Working registers

Microprocessor is a dependant unit and it requires the combination of other hardware like Memory, Timer Unit, and Interrupt Controller etc for proper functioning.

Intel claims the credit for developing the first Microprocessor unit Intel 4004, a 4 bit processor which was released in Nov 1971

- Developers of microprocessors.
  - Intel – Intel 4004 – November 1971(4-bit)
  - Intel – Intel 4040.
  - Intel – Intel 8008 – April 1972.
  - Intel – Intel 8080 – April 1974(8-bit).
  - Motorola – Motorola 6800.
  - Intel – Intel 8085 – 1976.
  - Zilog - Z80 – July 1976

### Microcontroller:

- A highly integrated silicon chip containing a CPU, scratch pad RAM, Special and General purpose Register Arrays, On Chip ROM/FLASH memory for program storage, Timer and Interrupt control units and dedicated I/O ports
- Microcontrollers can be considered as a super set of Microprocessors

Microcontroller can be general purpose (like Intel 8051, designed for generic applications and domains) or application specific (Like Automotive AVR from Atmel Corporation. Designed specifically for automotive applications)

Since a microcontroller contains all the necessary functional blocks for independentworking, they found greater place in the embedded domain in place of microprocessors

Microcontrollers are cheap, cost effective and are readily available in the market

- Texas Instruments TMS 1000 is considered as the world‟s first microcontroller

| Microprocessor | Microcontroller |
|---|---|
| A silicon chip representing a Central Processing Unit (CPU), which is capable of performing arithmetic as well as logical operations according to a pre-defined set of Instructions | A microcontroller is a highly integrated chip that contains a CPU, scratch pad RAM, Special and General purpose Register Arrays, On Chip ROM/FLASH memory for program storage, Timer and Interrupt control units and dedicated I/O ports |
| It is a dependent unit. It requires the combination of other chips like Timers, Program and data memory chips, Interrupt controllers etc for functioning | It is a self contained unit and it doesn't require external Interrupt Controller, Timer, UART etc for its functioning |
| Most of the time general purpose in design and operation | Mostly application oriented or domain specific |
| Doesn‟t contain a built in I/O port. The I/O Port functionality needs to be implemented with the help of external Programmable Peripheral Interface Chips like 8255 | Most of the processors contain multiple built-in I/O ports which can be operated as a single 8 or 16 or 32 bit Port or as individual port pins |
| Targeted for high end market where performance is important | Targeted for embedded market where performance is not so critical (At present this demarcation is invalid) |
| Limited power saving options compared to microcontrollers | Includes lot of power saving features |

## General Purpose Processor (GPP) Vs Application Specific Instruction Set Processor (ASIP)

- General Purpose Processor or GPP is a processor designed for general computational tasks
- GPPs are produced in large volumes and targeting the general market. Due to the high volume production, the per unit cost for a chip is low compared to ASIC or other specific ICs
- A typical general purpose processor contains an Arithmetic and Logic Unit (ALU) and Control Unit (CU)
- Application Specific Instruction Set processors (ASIPs) are processors with architecture and instruction set optimized to specific domain/application requirements like Network processing, Automotive, Telecom, media applications, digital signal processing, control applications etc.
- ASIPs fill the architectural spectrum between General Purpose Processors and Application Specific Integrated Circuits (ASICs)
- The need for an ASIP arises when the traditional general purpose processor are unable to meet the increasing application needs
- Some Microcontrollers (like Automotive AVR, USB AVR from Atmel), System on Chips, Digital Signal Processors etc are examples of Application Specific Instruction Set

Processors (ASIPs)

- ASIPs incorporate a processor and on-chip peripherals, demanded by the application requirement, program and data memory

**Digital Signal Processors (DSPs):**

- Powerful special purpose 8/16/32 bit microprocessors designed specifically to meet the computational demands and power constraints of today's embedded audio, video, and communications applications
- Digital Signal Processors are 2 to 3 times faster than the general purpose microprocessors in signal processing applications
- DSPs implement algorithms in hardware which speeds up the execution whereas general purpose processors implement the algorithm in firmware and the speed of execution depends primarily on the clock for the processors
- DSP can be viewed as a microchip designed for performing high speed computational operations for „addition", „subtraction", „multiplication" and „division"
- A typical Digital Signal Processor incorporates the following key units
  - Program Memory
  - Data Memory
  - Computational Engine
  - I/O Unit

Audio video signal processing, telecommunication and multimedia applications are typical examples where DSP is employed

**RISC V/s CISC Processors/Controllers:**

| RISC | CISC |
|---|---|
| Lesser no. of instructions | Greater no. of Instructions |
| Instruction Pipelining and increased execution speed | Generally no instruction pipelining feature |
| Orthogonal Instruction Set (Allows each instruction to operate on any register and use any addressing mode) | Non Orthogonal Instruction Set (All instructions are not allowed to operate on any register and use any addressing mode. It is instruction specific) |
| Operations are performed on registers only, the only memory operations are load and store | Operations are performed on registers or memory depending on the instruction |
| Large number of registers are available | Limited no. of general purpose registers |

| | |
|---|---|
| Programmer needs to write more code to execute a task since the instructions are simpler ones | . A programmer can achieve the desired functionality with a single instruction which in turn provides the effect of using more simpler single instructions in RISC |
| Single, Fixed length Instructions | Variable length Instructions |
| Less Silicon usage and pin count | More silicon usage since more additional decoder logic is required to implement the complex instruction decoding. |
| With Harvard Architecture | Can be Harvard or Von-Neumann Architecture |

| **Harvard Architecture** | **Von-Neumann Architecture** |
|---|---|
| Separate buses for Instruction and Data fetching | Single shared bus for Instruction and Data fetching |
| Easier to Pipeline, so high performance can be achieved | Low performance Compared to Harvard Architecture |
| Comparatively high cost | Cheaper |
| No memory alignment problems | Allows self modifying codes† |
| Since data memory and program memory are stored physically in different locations, no chances for accidental corruption of program memory | Since data memory and program memory are stored physically in same chip, chances for accidental corruption of program memory |

## Application Specific Integrated Circuit (ASIC):

- A microchip designed to perform a specific or unique application. It is used as replacement to conventional general purpose logic chips.
- ASIC integrates several functions into a single chip and thereby reduces the system development cost
- Most of the ASICs are proprietary products. As a single chip, ASIC consumes very small area in the total system and thereby helps in the design of smaller systems with high capabilities/functionalities.
- ASICs can be pre-fabricated for a special application or it can be custom fabricated by using the components from a re-usable „*building block*“ library of components for a particular customer application
- Fabrication of ASICs requires a non-refundable initial investment (Non Recurring Engineering (NRE) charges) for the process technology and configuration expenses

- If the Non-Recurring Engineering Charges (NRE) is born by a third party and the Application Specific Integrated Circuit (ASIC) is made openly available in the market, the ASIC is referred as Application Specific Standard Product (ASSP)

- The ASSP is marketed to multiple customers just as a general-purpose product , but to a smaller number of customers since it is for a specific application

## Programmable Logic Devices (PLDs):

❖ Logic devices provide specific functions, including device-to-device interfacing, data communication, signal processing, data display, timing and control operations, and almost every other function a system must perform.

❖ Logic devices can be classified into two broad categories - Fixed and Programmable. The circuits in a fixed logic device are permanent, they perform one function or set of functions - once manufactured, they cannot be changed

❖ Programmable logic devices (PLDs) offer customers a wide range of logic capacity, features, speed, and voltage characteristics - and these devices can be re-configured to perform any number of functions at any time

❖ Designers can use inexpensive software tools to quickly develop, simulate, and test their logic designs in PLD based design. The design can be quickly programmed into a device, and immediately tested in a live circuit

❖ PLDs are based on re-writable memory technology and the device is reprogrammed to change the design

### Programmable Logic Devices (PLDs) – CPLDs and FPGA

Field Programmable Gate Arrays (FPGAs) and Complex Programmable Logic Devices (CPLDs) are the two major types of programmable logic devices

## FPGA:

- FPGA is an IC designed to be configured by a designer after manufacturing.

- FPGAs offer the highest amount of logic density, the most features, and the highest performance.

- Logic gate is Medium to high density ranging from **1K to 500K** system gates

- These advanced FPGA devices also offer features such as built-in hardwired processors (such as the IBM Power PC), substantial amounts of memory, clock management systems, and support for many of the latest, very fast device-to-device signaling technologies

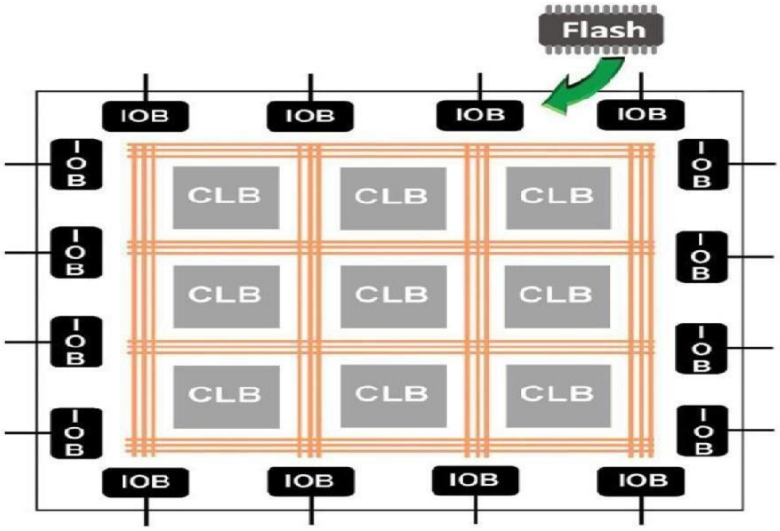


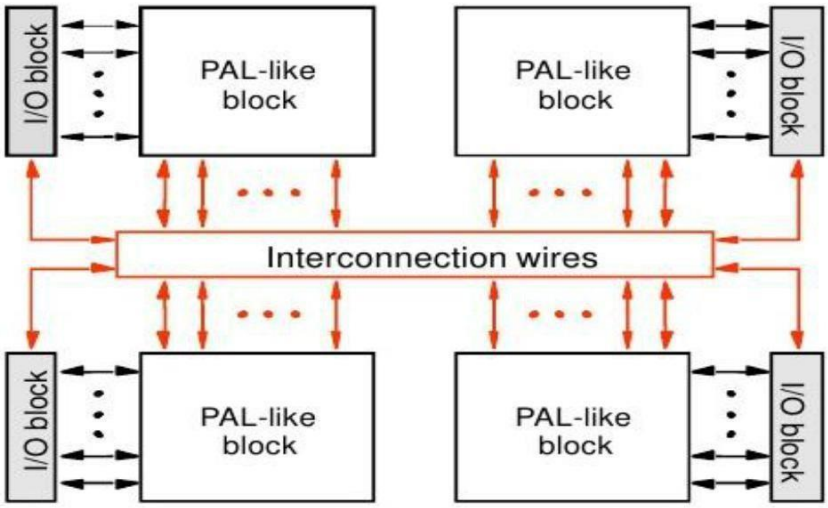

**Figure: FPGA Architecture**

- These advanced FPGA devices also offer features such as built-in hardwired processors, substantial amounts of memory, clock management systems, and support for many of the latest, very fast device-to-device signaling technologies.
- FPGAs are used in a wide variety of applications ranging from data processing and storage, to instrumentation, telecommunications, and digital signal processing

## CPLD:

- A **complex programmable logic device** (**CPLD**) is a programmable logic device with complexity between that of PALs and FPGAs, and architectural features of both.
- CPLDs, by contrast, offer much smaller amounts of logic - up to about 10,000 gates.
- CPLDs offer very predictable timing characteristics and are therefore ideal for critical control applications.

Structure of a CPLD




- CPLDs such as the Xilinx **CoolRunner** series also require extremely low amounts of power and are very inexpensive, making them ideal for cost-sensitive, battery-operated, portable applications such as mobile phones and digital handheld assistants.

### ADVANTAGES OF PLDs:

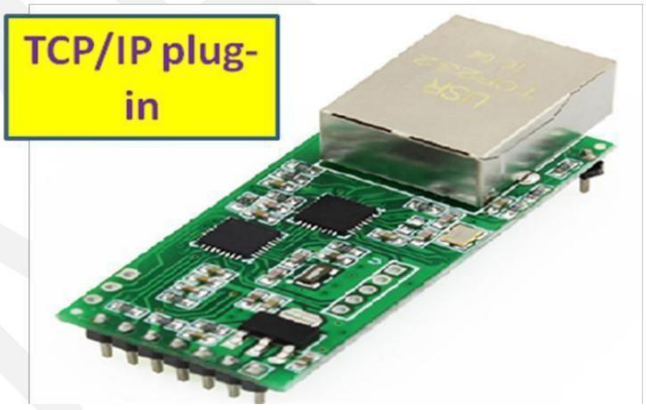
- PLDs offer customer much more flexibility during design cycle
- PLDSs do not require long lead times for prototype or production-the PLDs are already on a distributor‟s self and ready for shipment
- PLDs do not require customers to pay for large NRE costs and purchase expensive mask sets
- PLDs allow customers to order just the number of parts required when they need them. allowing them to control inventory.
- PLDs are reprogrammable even after a piece of equipment is shipped to a customer.
- The manufacturers able to add new features or upgrade the PLD based products that are in the field by uploading new programming file

## Commercial off the Shelf Component (COTS):

- A Commercial off-the-shelf (COTS) product is one which is used „*as-is*‟
- COTS products are designed in such a way to provide easy integration and interoperability with existing system components

- Typical examples for the COTS hardware unit are Remote Controlled Toy Car control unit including the RF Circuitry part, High performance, high frequency microwave electronics (2 to 200 GHz), High bandwidth analog-to-digital converters, Devices and components for operation at very high temperatures, Electro-optic IR imaging arrays, UV/IR Detectors etc


- A COTS component in turn contains a General Purpose Processor (GPP) or Application Specific Instruction Set Processor (ASIP) or Application Specific Integrated Chip (ASIC)/Application Specific Standard Product (ASSP) or Programmable Logic Device (PLD)




- The major advantage of using COTS is that they are readily available in the market, cheap and a developer can cut down his/her development time to a great extend.
- There is no need to design the module yourself and write the firmware .
- Everything will be readily supplied by the COTs manufacturer.

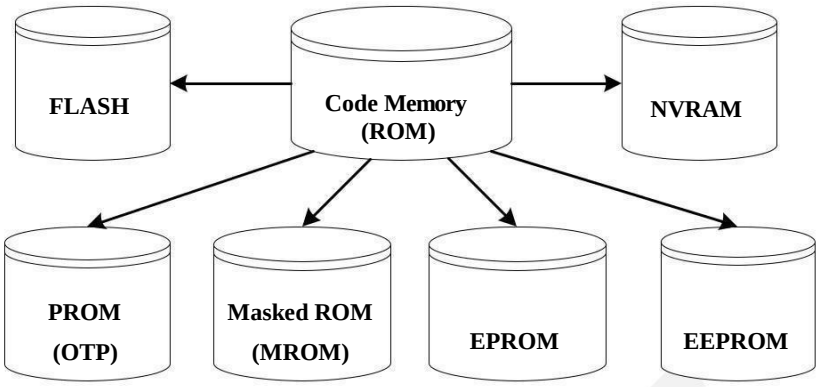
- The major problem faced by the end-user is that there are no operational and manufacturing standards.
- The major drawback of using COTs component in embedded design is that the manufacturer may withdraw the product or discontinue the production of the COTs at any time if rapid change in technology
- This problem adversely affect a commercial manufacturer of the embedded system which makes use of the specific COTs

## Memory:

- Memory is an important part of an embedded system. The memory used in embedded system can be either Program Storage Memory (ROM) or Data memory (RAM)
- Certain Embedded processors/controllers contain built in program memory and data memory and this memory is known as on-chip memory
- Certain Embedded processors/controllers do not contain sufficient memory inside the chip and requires external memory called **off-chip memory or external memory.**


### Memory – Program Storage Memory:

- Stores the program instructions
- Retains its contents even after the power to it is turned off. It is generally known as Non volatile storage memory
- Depending on the fabrication, erasing and programming techniques they are classified into




## 1. Masked ROM (MROM):

- One-time programmable memory.
- Uses hardwired technology for storing data.
- The device is factory programmed by masking and metallization process according to the data provided by the end user.
- The primary advantage of MROM is low cost for high volume production.
- MROM is the least expensive type of solid state memory.
- Different mechanisms are used for the masking process of the ROM, like
  - Creation of an enhancement or depletion mode transistor through channel implant
  - By creating the memory cell either using a standard transistor or a high threshold transistor.
  - In the high threshold mode, the supply voltage required to turn ON the transistor is above the normal ROM IC operating voltage.
  - This ensures that the transistor is always off and the memory cell stores always logic 0.
- The limitation with MROM based firmware storage is the inability to modify the device firmware against firmware upgrades.
- The MROM is permanent in bit storage, it is not possible to alter the bit information

## 2. Programmable Read Only Memory (PROM) / (OTP) :

- It is not pre-programmed by the manufacturer
- The end user is responsible for Programming these devices.
- PROM/OTP has *nichrome* or *polysilicon* wires arranged in a matrix, these wires can be functionally viewed as fuses.
- It is programmed by a PROM programmer which selectively burns the fuses according to the bit pattern to be stored.
- Fuses which are not blown/burned represents a logic “1” where as fuses which are blown/burned represents a logic “0”.The default state is logic “1”.
- OTP is widely used for commercial production of embedded systems whose proto-typed versions are proven and the code is finalized.
- It is a low cost solution for commercial production.
- OTPs cannot be reprogrammed.

## 3. Erasable Programmable Read Only Memory (EPROM):

- Erasable Programmable Read Only (EPROM) memory gives the flexibility to re-program the same chip.
- During development phase , code is subject to continuous changes and using an OTP is not economical.
- EPROM stores the bit information by charging the floating gate of an FET
- Bit information is stored by using an EPROM Programmer, which applies high voltage to charge the floating gate
- EPROM contains a quartz crystal window for erasing the stored information. If the window is exposed to Ultra violet rays for a fixed duration, the entire memory will be erased
- Even though the EPROM chip is flexible in terms of re-programmability, it needs to be taken out of the circuit board and needs to be put in a UV eraser device for 20 to 30 minutes

### 4. Electrically Erasable Programmable Read Only Memory (EEPROM):

- Erasable Programmable Read Only (EPROM) memory gives the flexibility to re-program the same chip using electrical signals
- The information contained in the EEPROM memory can be altered by using electrical signals at the register/Byte level
- They can be erased and reprogrammed within the circuit
- These chips include a chip erase mode and in this mode they can be erased in a few milliseconds
- It provides greater flexibility for system design
- The only limitation is their capacity is limited when compared with the standard ROM (A few kilobytes).

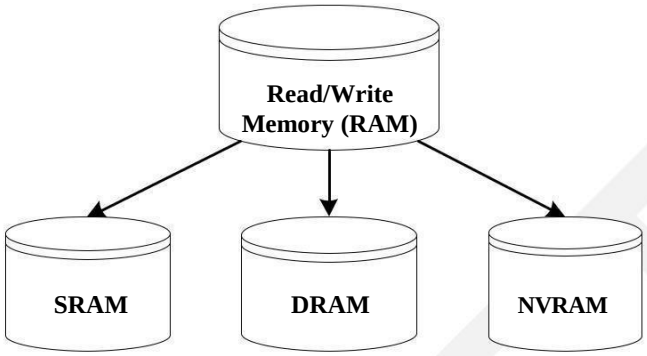
### 5. Program Storage Memory – FLASH

- FLASH memory is a variation of EEPROM technology.
- FALSH is the latest ROM technology and is the most popular ROM technology used in today‟s embedded designs
- It combines the re-programmability of EEPROM and the high capacity of standard ROMs
- FLASH memory is organized as sectors (blocks) or pages
- FLASH memory stores information in an array of floating gate MOSFET transistors
- The erasing of memory can be done at sector level or page level without affecting the other sectors or pages
- Each sector/page should be erased before re-programming
- The typical erasable capacity of FLASH is of the order of a few 1000 cycles.

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

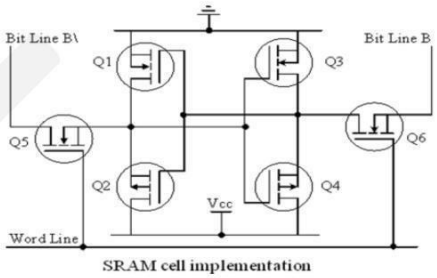
- RAM is the data memory or working memory of the controller/processor
- RAM is volatile, meaning when the power is turned off, all the contents are destroyed

- RAM is a direct access memory, meaning we can access the desired memory location directly without the need for traversing through the entire memory locations to reach the desired memory position (i.e. Random Access of memory location)




## 1. Static RAM (SRAM):

- Static RAM stores data in the form of Voltage.
- They are made up of flip-flops
- In typical implementation, an SRAM cell (bit) is realized using 6 transistors (or 6 MOSFETs).

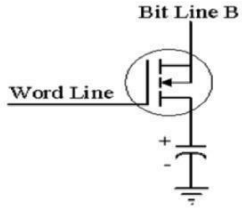


SRAM cell implementation

- Four of the transistors are used for building the latch (flip-flop) part of the memory cell and 2 for controlling the access.
- Static RAM is the fastest form of RAM available.
- SRAM is fast in operation due to its resistive networking and switching capabilities

## 2. Dynamic RAM (DRAM)

- Dynamic RAM stores data in the form of charge. They are made up of MOS transistor gates
- The advantages of DRAM are its high density and low cost compared to SRAM



DRAM cell implementation

- The disadvantage is that since the information is stored as charge it gets leaked off with time and to prevent this they need to be refreshed periodically
- Special circuits called DRAM controllers are used for the refreshing operation. The refresh operation is done periodically in milliseconds interval

### SRAM Vs DRAM:

| SRAM Cell | DRAM Cell |
|---|---|
| Made up of 6 CMOS transistors (MOSFET) | Made up of a MOSFET and a capacitor |
| Doesn‟t Require refreshing | Requires refreshing |
| Low capacity (Less dense) | High Capacity (Highly dense) |
| More expensive | Less Expensive |
| Fast in operation. Typical access time is 10ns | Slow in operation due to refresh requirements. Typical access time is 60ns. Write operation is faster than read operation. |

### 3. Non Volatile RAM (NVRAM):

- Random access memory with battery backup
- It contains Static RAM based memory and a minute battery for providing supply to the memory in the absence of external power supply
- The memory and battery are packed together in a single package
- NVRAM is used for the non volatile storage of results of operations or for setting up of flags etc
- The life span of NVRAM is expected to be around 10 years
- DS1744 from Maxim/Dallas is an example for 32KB NVRAM

## Memory selection for Embedded Systems:

- Selection of suitable memory is very much essential step in high performance applications, because the challenges and limitations of the system performance are often decided upon the type of memory architecture.
- Systems memory requirement depend primarily on the nature of the application thatis planned to run on the system.
- Memory performance and capacity requirement for low cost systems are small, whereas memory throughput can be the most critical requirement in a complex, high performance system.

- Following are the factors that are to be considered while selecting the memory devices,
  - Speed
  - Data storage size and capacity
  - Bus width
  - Power consumption
  - Cost

- **Embedded system requirements:**
  - Program memory for holding control algorithm or embedded OS and the applications designed to run on top of OS.
  - Data memory for holding variables and temporary data during task execution.
  - Memory for holding non-volatile data which are modifiable by the application.

- The memory requirement for an embedded system in terms of RAM (SRAM/DRAM) and ROM (EEPROM/FLASH/NVRAM) is solely dependent on the type of the embedded system and applications for which it is designed.
- There is no hard and fast rule for calculating the memory requirements.
- Lot of factors need to be considered for selecting the type and size of memory for embedded system.
- **Example:** Design of Embedded based electronic Toy.
- SOC or microcontroller can be selected based type(RAM &ROM) and size of on-chip memory for the design of embedded system.
- If on-chip memory is not sufficient then how much external memory need to be interfaced.
- If the ES design is RTOS based ,the RTOS requires certain amount of RAM for its execution and ROM for storing RTOS Image.
- The RTOS suppliers gives amount of run time RAM requirements and program memory requirements for the RTOS.
- Additional memory is required for executing user tasks and user applications.

- On a safer side, always add a buffer value to the total estimated RAM and ROM requirements.
- A smart phone device with windows OS is typical example for embedded device requires say 512MB RAM and 1GB ROM are minimum requirements for running the mobile device.
- And additional RAM &ROM memory is required for running user applications.
- So estimate the memory requirements for install and run the user applications without facing memory space.
- Memory can be selected based on size of the memory ,data bus and address bus size of the processor/controller.
- Memory chips are available in standard sizes like 512 bytes,1KB,2KB ,4KB,8KB,16 KB ….1MB etc.
- FLASH memory is the popular choice for ROM in embedded applications .
- It is powerful and cost-effective solid state storage technology for mobile electronic devices and other consumer applications.
- Flash memory available in two major variants
- 1. NAND FLASH 2. NOR FLASH
- NAND FLASH is a high density low cost non-volatile storage memory**.**
- NOR FLASH is less dense and slightly expensive but supports Execute in place(XIP).
- The XIP technology allows the execution of code memory from ROM itself without the need for copying it to the RAM.
- The EEPROM is available as either serial interface or parallel interface chip.
- If the processor/controller of the device supports serial interface and the amount of data to write and read to and from the device (Serial EEPROM) is less.
- The serial EEPROM saves the address space of the total system.
- The memory capacity of the serial EEPROM is expressed in bits or Kilobits.

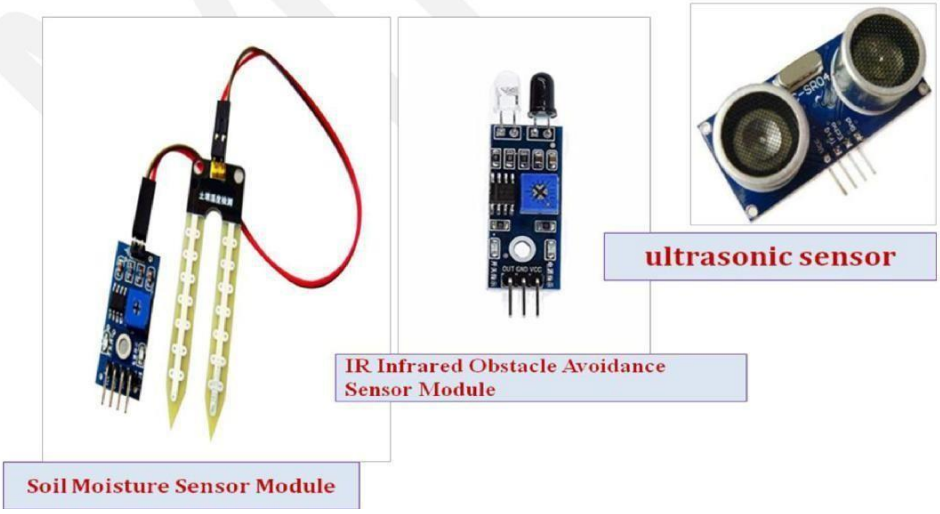
- Industrial grade memory chips are used in certain embedded devices may be operated at extreme environmental conditions like high temperature.

## Sensors & Actuators:

- Embedded system is in constant interaction with the real world
- Controlling/monitoring functions executed by the embedded system is achieved in accordance with the changes happening to the Real World.
- The changes in the system environment or variables are detected by the sensors connected to the input port of the embedded system.
- If the embedded system is designed for any controlling purpose, the system will produce some changes in controlling variable to bring the controlled variable to the desired value.
- It is achieved through an actuator connected to the out port of the embedded system.

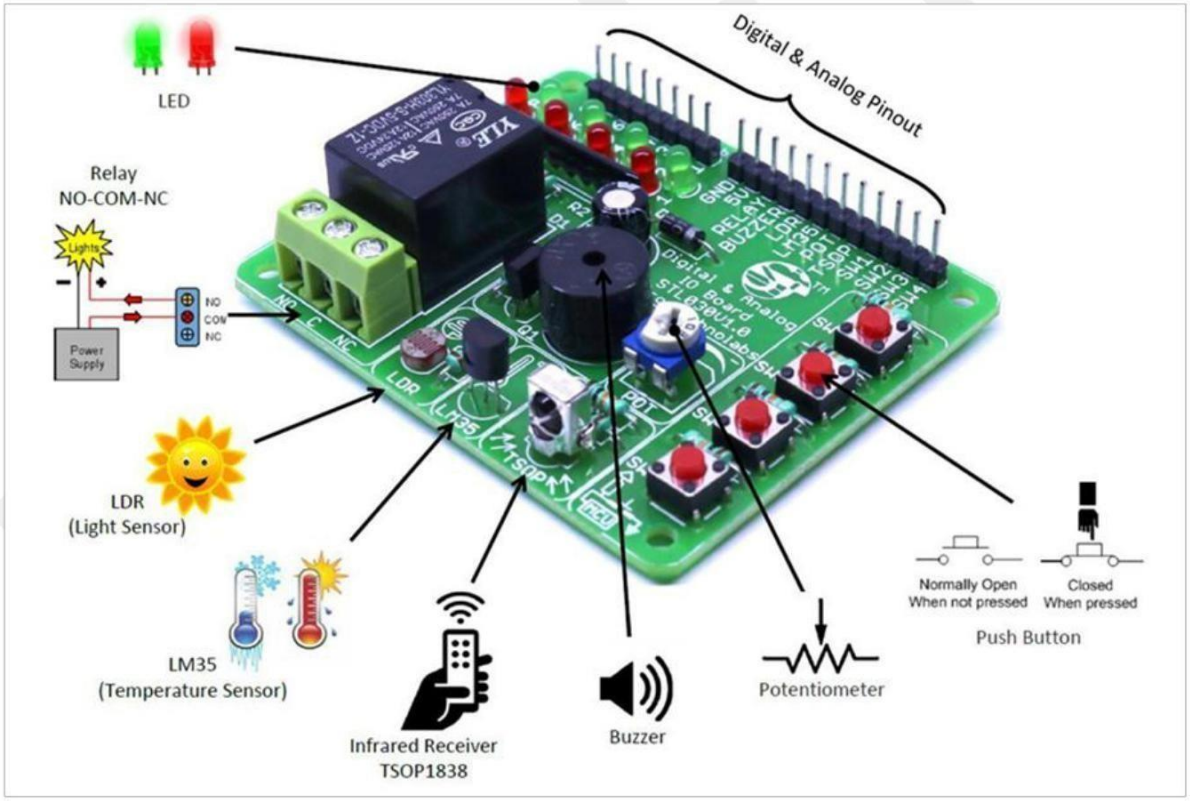
### Sensor:

- A transducer device which converts energy from one form to another for any measurement or control purpose. Sensors acts as input device
- Eg. Hall Effect Sensor which measures the distance between the cushion and magnet in the Smart Running shoes from adidas
- **Example: IR, humidity , PIR(passive infra red) , ultrasonic , piezoelectric , smoke sensors**



**Actuator:**

- A form of transducer device (mechanical or electrical) which converts signals to corresponding physical action (motion). Actuator acts as an output device
- Eg. Micro motor actuator which adjusts the position of the cushioning element in the Smart Running shoes from adidas


Silicon TechnoLabs Digital Analog Arduino Starter kit

# EMBEDDED FIRMWARE DESIGN & DEVELOPMENT

## Introduction:

The control algorithm (Program instructions) and or the configuration settings that an embedded system developer dumps into the code (Program) memory of the embedded system

It is an un-avoidable part of an embedded system.

The embedded firmware can be developed in various methods like

- Write the program in high level languages like Embedded C/C++ using an Integrated Development Environment (The IDE will contain an editor, compiler, linker, debugger, simulator etc. IDEs are different for different family of processors/controllers.
- Write the program in Assembly Language using the Instructions Supported by your application's target processor/controller

- **Embedded Firmware Design & Development:**
- The embedded firmware is responsible for controlling the various peripherals of the embedded hardware and generating response in accordance with the functional requirements of the product.
- The embedded firmware is the master brain of the embedded system. The embedded firmware imparts intelligence to an Embedded system. It is a onetime process and it can happen at any stage.
- The product starts functioning properly once the intelligence imparted to the product by embedding the firmware in the hardware.
- The product will continue serving the assigned task till hardware breakdown occurs or a corruption in embedded firmware.
- In case of hardware breakdown , the damaged component may need to be replaced and for firmware corruptions the firmware should be re-loaded, to bring back the embedded product to the normal functioning.

- The embedded firmware is usually stored in a permanent memory (ROM) and it is non alterable by end users.
- Designing Embedded firmware requires understanding of the particular embedded product hardware, like various component interfacing, memory map details, I/O port details, configuration and register details of various hardware chips used and some programming language (either low level Assembly Language or High level language like C/C++ or a combination of the two)
- The embedded firmware development process starts with the conversion of the firmware requirements into a program model using various modeling tools.
- The firmware design approaches for embedded product is purely dependent on the complexity of the functions to be performed and speed of operation required.
- There exist two basic approaches for the design and implementation of embedded firmware, namely;

  - **The Super loop based approach**
  - **The Embedded Operating System based approach**

    - The decision on which approach needs to be adopted for firmware development is purely dependent on the complexity and system requirements

## Embedded firmware Design Approaches – The Super loop:

- The Super loop based firmware development approach is Suitable for applications that are not time critical and where the response time is not so important (Embedded systems where missing deadlines are acceptable).
- It is very similar to a conventional procedural programming where the code is executed task by task
- The tasks are executed in a never ending loop.
- The task listed on top on the program code is executed first and the tasks just below the top are executed after completing the first task

A typical super loop implementation will look like:

1. Configure the common parameters and perform initialization for various hardware components memory, registers etc.
2. Start the first task and execute it
3. Execute the second task

4. Execute the next task 5.

:

6. :

7. Execute the last defined task

8. Jump back to the first task and follow the same flow.

The 'C' program code for the super loop is given below void

```
main ()

{

Configurations ();
Initializations ();

while (1)

{
Task 1 ();
Task 2 ();
:

:

Task n ();

}

}
```

**Pros:**

Doesn't require an Operating System for task scheduling and monitoring and free from OS related overheads

Simple and straight forward design

Reduced memory footprint

**Cons:**

Non Real time in execution behavior (As the number of tasks increases the frequency at which a task gets CPU time for execution also increases)

Any issues in any task execution may affect the functioning of the product (This can be effectively tackled by using Watch Dog Timers for task execution monitoring)

**Enhancements:**

- Combine Super loop based technique with interrupts
- Execute the tasks (like keyboard handling) which require Real time attention as Interrupt Service routines.

## 2. Embedded firmware Design Approaches – Embedded OS based Approach:

- The embedded device contains an Embedded Operating System which can be one of:
  - **A Real Time Operating System (RTOS)**
  - **A Customized General Purpose Operating System (GPOS)**
- The Embedded OS is responsible for scheduling the execution of user tasks and the allocation of system resources among multiple tasks
- It Involves lot of OS related overheads apart from managing and executing user defined tasks

Microsoft® Windows XP Embedded is an example of GPOS for embedded devices

Point of Sale (PoS) terminals, Gaming Stations, Tablet PCs etc are examples of embedded devices running on embedded GPOSs

'*Windows CE*', '*Windows Mobile*','*QNX*', '*VxWorks*', '*ThreadX*', '*MicroC/OS-II*', '*Embedded Linux*', '*Symbian*' etc are examples of RTOSs employed in Embedded Product development

Mobile Phones, PDAs, Flight Control Systems etc are examples of embedded devices that runs on RTOSs

**Embedded firmware Development Languages/Options**

**Assembly Language**

**High Level Language**

- Subset of C (Embedded C)
- Subset of C++ (Embedded C++)
- Any other high level language with supported Cross-compiler

## 1. Embedded firmware Development Languages/Options – Assembly Language

- '***Assembly Language***' is the human readable notation of '***machine language***'
- 'M***achine language***' is a processor understandable language
- Machine language is a binary representation and it consists of 1s and 0s Assembly language and machine languages are processor/controller dependent
- An Assembly language program written for one processor/controller family will not work with others
- Assembly language programming is the process of writing processor specific machine code in mnemonic form, converting the mnemonics into actual processor instructions (machine language) and associated data using an assembler
- The general format of an assembly language instruction is an Opcode followed by Operands
- The Opcode tells the processor/controller what to do and the Operands provide the data and information required to perform the action specified by the opcode
- It is not necessary that all opcode should have Operands following them. Some of the Opcode implicitly contains the operand and in such situation no operand is required. The operand may be a single operand, dual operand or more

The 8051 Assembly Instruction

MOV A, #30

Moves decimal value 30 to the 8051 Accumulator register. Here *MOV A* is the Opcode and 30 is the operand (single operand). The same instruction when written in machine language will look like 01110100 00011110

The first 8 bit binary value 01110100 represents the opcode *MOV A* and the second 8 bit binary value 00011110 represents the operand 30.

- Assembly language instructions are written one per line

  A machine code program consists of a sequence of assembly language instructions, where each statement contains a mnemonic (Opcode + Operand)

  Each line of an assembly language program is split into four fields as:

**LABEL OPCODE OPERAND COMMENTS**

LABEL is an optional field. A 'LABEL' is an identifier used extensively in programs to reduce the reliance on programmers for remembering where data or code is located. LABEL is commonly used for representing

- A memory location, address of a program, sub-routine, code portion etc.
- The maximum length of a label differs between assemblers. Assemblers insist strict formats for labeling. Labels are always suffixed by a colon and begin with a valid character. Labels can contain number from 0 to 9 and special character _ (underscore).

```
;#################################################################
;    SUBROUTINE FOR GENERATING DELAY
;    DELAY PARAMETR PASSED THROUGH REGISTER R1
;    RETURN VALUE NONE,REGISTERS USED: R0, R1
;#################################################################
#####    DELAY:      MOV R0, #255        ; Load Register R0 with 255
           DJNZ R1, DELAY; Decrement R1 and loop till          R1= 0
           RET                 ; Return to calling program
```

- The symbol ; represents the start of a comment. Assembler ignores the text in a line after the ; symbol while assembling the program
- DELAY is a label for representing the start address of the memory location where the piece of code is located in code memory
- The above piece of code can be executed by giving the label DELAY as part of the instruction. E.g. LCALL DELAY; LMP DELAY

## 2. Assembly Language – Source File to Hex File Translation:

The Assembly language program written in assembly code is saved as *.asm* (Assembly file) file or a *.src* (source) file or a format supported by the assembler

Similar to 'C' and other high level language programming, it is possible to have multiple source files called modules in assembly language programming. Each module is represented by a '***.asm***' or '***.src***' file or the assembler supported file format similar to the '***.c***' files in C programming

The software utility called 'Assembler' performs the translation of assembly code to machine code

The assemblers for different family of target machines are different. A51 Macro Assembler from Keil software is a popular assembler for the 8051 family micro controller

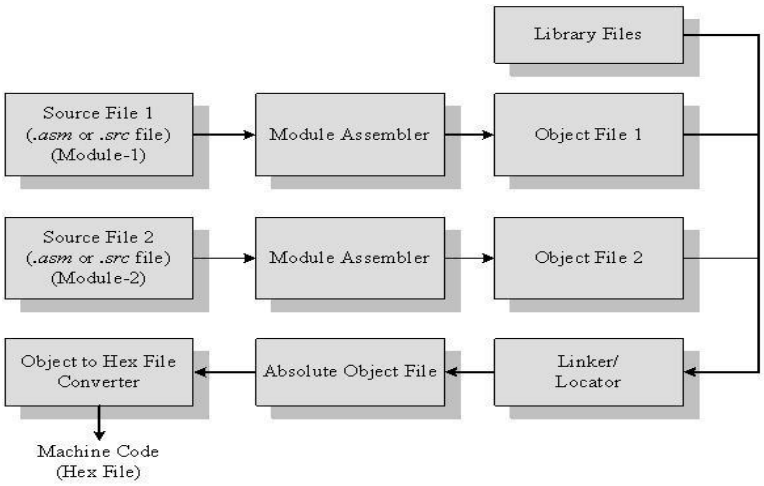



**Figure 5: Assembly Language to machine language conversion process**

- Each source file can be assembled separately to examine the syntax errors and incorrect assembly instructions
- Assembling of each source file generates a corresponding object file. The object file does not contain the absolute address of where the generated code needs to be placed (a re-locatable code) on the program memory

  The software program called linker/locater is responsible for assigning absolute address to object files during the linking process
  The Absolute object file created from the object files corresponding to different source code modules contain information about the address where each instruction needs to be placed in code memory

  A software utility called 'Object to Hex file converter' translates the absolute object file to corresponding hex file (binary file)

**Advantages:**

**1.Efficient Code Memory & Data Memory Usage (Memory Optimization):**

The developer is well aware of the target processor architecture and memory organization, so optimized code can be written for performing operations.

This leads to less utilization of code memory and efficient utilization of data memory.

**2.High Performance:**

- Optimized code not only improves the code memory usage butalso improves the total system performance.
- Through effective assembly coding, optimum performance canbe achieved for target processor.

### 3.Low level Hardware Access:

- Most of the code for low level programming like accessing external device specific registers from OS kernel ,device drivers, and low level interrupt routines, etc are making use of direct assembly coding.

### 4.Code Reverse Engineering:

- It is the process of understanding the technology behind a product by extracting the information from the finished product.
- It can easily be converted into assembly code using a dis-assembler program for the target machine.

**Drawbacks:**

### 1.High Development time:

- The developer takes lot of time to study about architecture ,memory organization, addressing modes and instruction set of target processor/controller.
- More lines of assembly code is required for performing a simple action.

### 2.Developer dependency:

- There is no common written rule for developing assembly language based applications.

### 3.Non portable:

- Target applications written in assembly instructions are valid only for that particular family of processors and cannot be re-used for another target processors/controllers.
- If the target processor/controller changes, a complete re-writing of the application using assembly language for new target processor/controller is required.

## 2. Embedded firmware Development Languages/Options – High Level Language

- The embedded firmware is written in any high level language like C, C++
- A software utility called 'cross-compiler' converts the high level language to target processor specific machine code
- The cross-compilation of each module generates a corresponding object file. The object file does not contain the absolute address of where the generated code needs to be placed (a re-locatable code) on the program memory
- The software program called linker/locater is responsible for assigning absolute address to object files during the linking process

- The Absolute object file created from the object files corresponding to different source code modules contain information about the address where each instruction needs to be placed in code memory

- A software utility called 'Object to Hex file converter' translates the absolute object file to corresponding hex file (binary file)

**Embedded firmware Development Languages/Options – High Level Language – Source File to Hex File Translation**

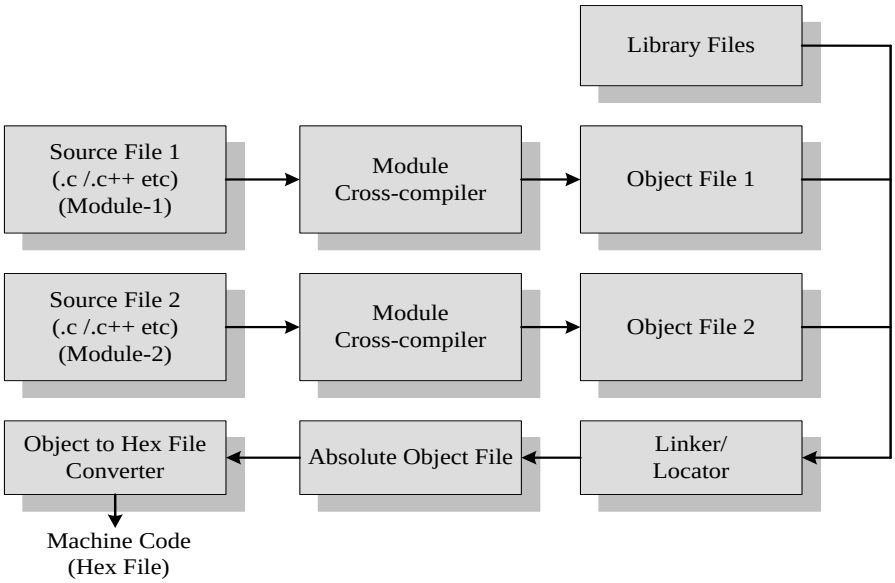



**Figure 6: High level language to machine language conversion process**

**Advantages:**

- **Reduced Development time:** Developer requires less or little knowledge on internal hardware details and architecture of the target processor/Controller.

- **Developer independency:** The syntax used by most of the high level languages are universal and a program written high level can easily understand by a second person knowing the syntax of the language

- **Portability:** An Application written in high level language for particular target processor /controller can be easily be converted to another target processor/controller specific application with little or less effort

**Drawbacks**

- The cross compilers may not be efficient in generating the optimized target processor specific instructions.
- Target images created by such compilers may be messy and non- optimized in terms of performance as well as code size.
- The investment required for high level language based development tools (IDE) is high compared to Assembly Language based firmware development tools.

**Text Books:**


1. **Introduction to Embedded Systems – Shibu K.V Mc Graw Hill**
2. **Embedded System Design-Raj Kamal TMH**

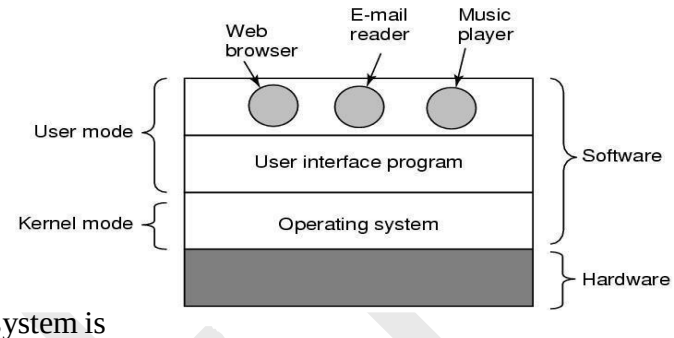
# UNIT-V

# RTOS Based Embedded System Design


MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY
Permanently Affliated to JNTUH and Approved by AICTE, New Delhi

## Operating System Basics:

- The Operating System acts as a bridge between the user applications/tasks and the underlying system resources through a set of system functionalities and services
- OS manages the system resources and makes them available to the user applications/tasks on a need basis




- The primary functions of an Operating system is
  - Make the system convenient to use
  - Organize and manage the system resources efficiently andcorrectly

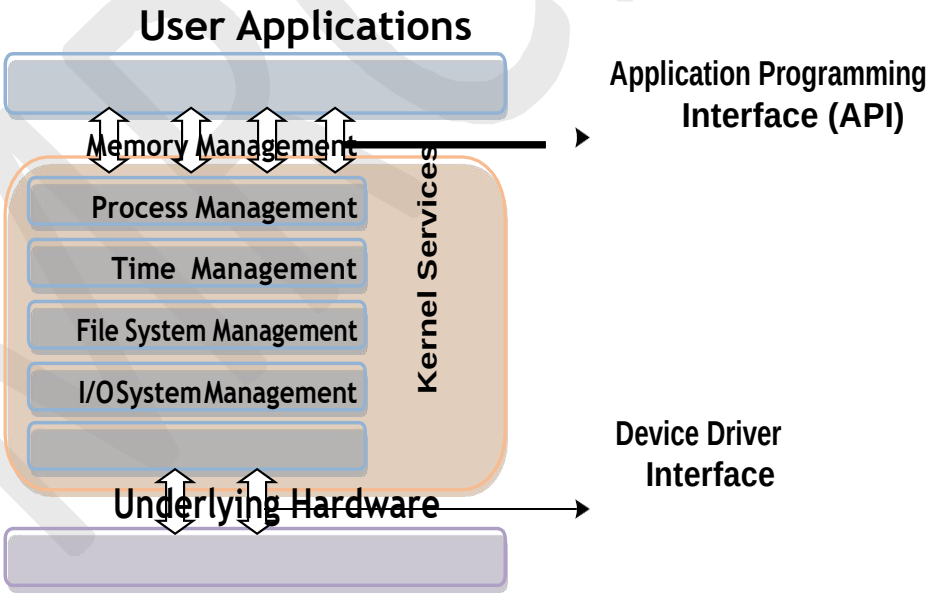



**Figure 1: The Architecture of Operating System**

## The Kernel:

- The kernel is the core of the operating system
- It is responsible for managing the system resources and the communication among the hardware and other system services
- Kernel acts as the abstraction layer between system resources and user applications
- Kernel contains a set of system libraries and services.
- For a general purpose OS, the kernel contains different serviceslike
  - Process Management
  - Primary Memory Management
  - File System management
  - I/O System (Device) Management
  - Secondary Storage Management
  - Protection
  - Time management
  - Interrupt Handling

## Kernel Space and User Space:

- The program code corresponding to the kernel applications/services are kept in a contiguous area (OS dependent) of primary (working) memory and is protected from the un-authorized access by userprograms/applications
- The memory space at which the kernel code is located is known as '*Kernel Space'*
- All user applications are loaded to a specific area of primary memory and this memory area is referred as '*User Space*'
- The partitioning of memory into kernel and user space is purely Operating System dependent
- An operating system with virtual memory support, loads the user applications into its corresponding virtual memory space with demand paging technique

- Most of the operating systems keep the kernel application code in main memory and it is not swapped out into the secondary memory

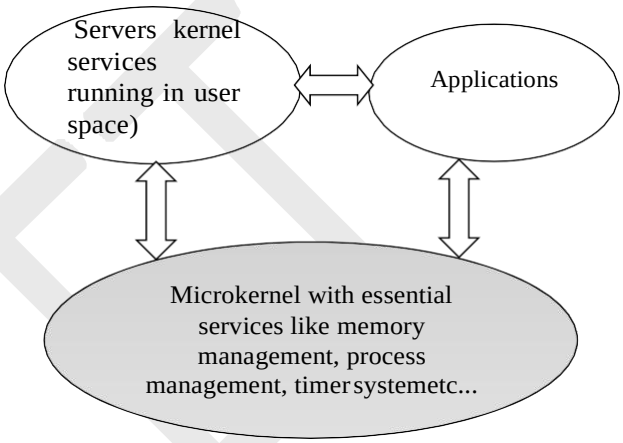
**Monolithic Kernel:**

- All kernel services run in the kernel space
- All kernel modules run within the same memory space under a singlekernel thread
- The tight internal integration of kernel modules in monolithic kernel architecture allows the effective utilization of the low-level features of the underlying system
- The major drawback of monolithic kernel is that any error or failure in any one of the kernel modules leads to the crashing of the entire kernel application

- LINUX, SOLARIS, MS-DOS kernels are examples of monolithic kernel Applications

**Microkernel**

- The microkernel design incorporates only the essential set of Operating System services into the kernel
- Rest of the Operating System services are implemented in programsknown as '*Servers*' which runs in user space
- The kernel design is highly modular provides OS-neutral abstraction.
- Memory management, process management, timer systems and interrupt handlers are examples of essential services, which forms the part of the microkernel



- QNX, Minix 3 kernels are examples for microkernel.

**Benefits of Microkernel:**

1. Robustness: If a problem is encountered in any services in server can reconfigured and re-started without the need for re-starting the entire OS.
2. Configurability: Any services , which run as 'server' application can be changed without need to restart the whole system.

**Types of Operating Systems:**

Depending on the type of kernel and kernel services, purpose and type of computing systems where the OS is deployed and the responsiveness to applications, Operating Systems are classified into

1. **General Purpose Operating System (GPOS):**
2. **Real Time Purpose Operating System (RTOS):**

## 1. General Purpose Operating System (GPOS):

- Operating Systems, which are deployed in general computing systems
- The kernel is more generalized and contains all the required servicesto execute generic applications
- Need not be deterministic in execution behavior
- May inject random delays into application software and thus causeslow responsiveness of an application at unexpected times
- Usually deployed in computing systems where deterministic behavior isnot an important criterion
- Personal Computer/Desktop system is a typical example for a systemwhere GPOSs are deployed.
- Windows XP/MS-DOS etc are examples of General Purpose Operating System

## 2. Real Time Purpose Operating System (RTOS):

- Operating Systems, which are deployed in embedded systems demanding real-time response
- Deterministic in execution behavior. Consumes only known amount of time for kernel applications

- Implements scheduling policies for executing the highest priority task/application always
- Implements policies and rules concerning time-critical allocation of a system's resources
- Windows CE, QNX, VxWorks , MicroC/OS-II etc are examples of Real Time Operating Systems (RTOS)

**The Real Time Kernel:** The kernel of a Real Time Operating System is referred as Real Time kernel. In complement to the conventional OS kernel, the Real Time kernel is highly specialized and it contains only the minimal set of services required for running the user applications/tasks. The basic functions of a Real Time kernel are

a) Task/Process management

b) Task/Process scheduling

c) Task/Process synchronization

d) Error/Exception handling

e) Memory Management

f) Interrupt handling

g) Time management

- **Real Time Kernel Task/Process Management:** Deals with setting up the memory space for the tasks, loading the task's code into the memory space, allocating system resources, setting up a Task Control Block (TCB) for the task and task/process termination/deletion. A Task Control Block (TCB) is used for holding the information corresponding to a task. TCB usually contains the following set of information
  - *Task ID:* Task Identification Number
  - *Task State:* The current state of the task. (E.g. State= 'Ready' for atask which is ready to execute)
  - *Task Type*: Task type. Indicates what is the type for this task. The taskcan be a hard real time or soft real time or background task.
  - *Task Priority*: Task priority (E.g. Task priority =1 for task with priority =1)
  - *Task Context Pointer*: Context pointer. Pointer for context saving

- *Task Memory Pointers*: Pointers to the code memory, data memoryand stack memory for the task
- *Task System Resource Pointers*: Pointers to system resources (semaphores, mutex etc) used by the task
- *Task Pointers:* Pointers to other TCBs (TCBs for preceding, nextand waiting tasks)
- *Other Parameters* Other relevant task parameters

The parameters and implementation of the TCB is kernel dependent. The TCB parameters vary across different kernels, based on the task management implementation

- **Task/Process Scheduling:** Deals with sharing the CPU among various tasks/processes. A kernel application called '*Scheduler*' handles the task scheduling. Scheduler is nothing but an algorithm implementation, which performs the efficient and optimal scheduling of tasks to provide a deterministic behavior.
- **Task/Process Synchronization:** Deals with synchronizing the concurrent access of a resource, which is shared across multiple tasks and the communication between various tasks.
- **Error/Exception handling:** Deals with registering and handling the errors occurred/exceptions raised during the execution of tasks. Insufficient memory, timeouts, deadlocks, deadline missing, bus error, divide by zero, unknown instruction execution etc, are examples of errors/exceptions. Errors/Exceptions can happen at the kernel level services or at task level. Deadlock is an example for kernel level exception, whereas timeout is an example for a task level exception. The OS kernel gives the information about the error in the form of a system call (API).

**Memory Management:**

- The memory management function of an RTOS kernel is slightly different compared to the General Purpose Operating Systems
- The memory allocation time increases depending on the size of the block of memory needs to be allocated and the state of the allocated memory block (initialized memory block consumes more allocation time than un- initialized memory block)
- Since predictable timing and deterministic behavior are the primary focus for an RTOS, RTOS achieves this by compromising the effectiveness of memory allocation
- RTOS generally uses '*block*' based memory allocation technique, instead of the usual dynamic memory allocation techniques used by the GPOS.

- RTOS kernel uses blocks of fixed size of dynamic memory and the block is allocated for a task on a need basis. The blocks are stored in a '*Free buffer Queue*'.
- Most of the RTOS kernels allow tasks to access any of the memory blocks without any memory protection to achieve predictable timing and avoid the timing overheads
- RTOS kernels assume that the whole design is proven correct and protection is unnecessary. Some commercial RTOS kernels allow memory protection as optional and the kernel enters a *fail-safe* mode when an illegal memory access occurs
- The memory management function of an RTOS kernel is slightly different compared to the General Purpose Operating Systems
- A few RTOS kernels implement *Virtual Memory* concept for memory allocation if the system supports secondary memory storage (like HDD and FLASH memory).
- In the '*block*' based memory allocation, a block of fixed memory is always allocated for tasks on need basis and it is taken as a unit. Hence, there will not be any memory fragmentation issues.
- The memory allocation can be implemented as constant functions and thereby it consumes fixed amount of time for memory allocation. This leaves the deterministic behavior of the RTOS kernel untouched.

### Interrupt Handling:

- Interrupts inform the processor that an external device or an associated task requires immediate attention of the CPU.
- Interrupts can be either *Synchronous* or *Asynchronous*.
- Interrupts which occurs in sync with the currently executing task is known as *Synchronous* interrupts. Usually the software interrupts fall under the Synchronous Interrupt category. Divide by zero, memory segmentation error etc are examples of Synchronous interrupts.
- For synchronous interrupts, the interrupt handler runs in the same context of the interrupting task.
- Asynchronous interrupts are interrupts, which occurs at any point of execution of any task, and are not in sync with the currently executing task.
- The interrupts generated by external devices (by asserting the Interrupt line of the processor/controller to which the interrupt line of the device is connected) connected to the processor/controller, timer overflow interrupts, serial data reception/ transmission interrupts etc are examples for asynchronous interrupts.

- For asynchronous interrupts, the interrupt handler is usually written as separate task (Depends on OS Kernel implementation) and it runs in a different context. Hence, a context switch happens while handling the asynchronous interrupts.
- Priority levels can be assigned to the interrupts and each interrupts can be enabled or disabled individually.
- Most of the RTOS kernel implements '*Nested Interrupts*' architecture. Interrupt nesting allows the pre-emption (interruption) of an Interrupt Service Routine (ISR), servicing an interrupt, by a higher priority interrupt.

**Time Management:**

- Interrupts inform the processor that an external device or an associated task requires immediate attention of the CPU.
- Accurate time management is essential for providing precise time reference for all applications
- The time reference to kernel is provided by 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 timer interrupt is referred as 'Timer tick'
- The 'Timer tick' is taken as the timing reference by the kernel. The 'Timer tick' interval may vary depending on the hardware timer. Usually the 'Timer tick' varies in the microseconds range
- The time parameters for tasks are expressed as the multiples of the 'Timer tick'
- The System time is updated based on the 'Timer tick'
- If the System time register is 32 bits wide and the 'Timer tick' interval is 1microsecond, the System time register will reset in

  232 * 10-6/ (24 * 60 * 60) = 49700 Days =~ 0.0497 Days = 1.19 Hours

  If the 'Timer tick' interval is 1 millisecond, the System time register will reset in

  232 * 10-3 / (24 * 60 * 60) = 497 Days = 49.7 Days =~ 50 Days

The '*Timer tick*' interrupt is handled by the 'Timer Interrupt' handler of kernel. The '*Timer tick*' interrupt can be utilized for implementing the following actions.

- Save the current context (Context of the currently executingtask)
- Increment the System time register by one. Generate timing error and reset the System time register if the timer tick count is greater than the maximum range available for System time register
- Update the timers implemented in kernel (Increment or decrement the timer registers for each timer depending on the count direction setting for each register. Increment registers with count direction setting = '*count up*' and decrement registers with count direction setting = '*countdown*')
- Activate the periodic tasks, which are in the idle state
- Invoke the scheduler and schedule the tasks again based on the scheduling algorithm
- Delete all the terminated tasks and their associated data structures(TCBs)
- Load the context for the first task in the ready queue. Due to the re- scheduling, the ready task might be changed to a new one from the task, which was pre-empted by the 'Timer Interrupt' task

**Hard Real-time System:**

- ❖ A Real Time Operating Systems which strictly adheres to the timing constraints for a task
- ❖ A Hard Real Time system must meet the deadlines for a task without any slippage
- ❖ Missing any deadline may produce catastrophic results for Hard Real Time Systems, including permanent data lose and irrecoverable damages to the system/users
- ❖ Emphasize on the principle '*A late answer is a wronganswer*'
- ❖ Air bag control systems and Anti-lock Brake Systems (ABS) of vehicles are typical examples of Hard Real Time Systems
- ❖ As a rule of thumb, Hard Real Time Systems does not implement the virtual memory model for handling the memory. This eliminates the delay in swapping in and out the code corresponding to the task to and from the primary memory
- ❖ The presence of *Human in the loop* (*HITL*) for tasks introduces un- expected delays in the task execution. Most of the Hard Real Time Systems are automatic and does not contain a 'human in theloop'

- **Soft Real-time System:**

  - ❖ Real Time Operating Systems that does not guarantee meetingdeadlines, but, offer the best effort to meet the deadline
  - ❖ Missing deadlines for tasks are acceptable if the frequency ofdeadline missing is within the compliance limit of the Quality of Service(QoS)
  - ❖ A Soft Real Time system emphasizes on the principle '*A late answer is an acceptable answer, but it could have done bitfaster'*
  - ❖ Soft Real Time systems most often have a '*human in the loop (HITL)*'

❖ Automatic Teller Machine (ATM) is a typical example of Soft Real Time System. If the ATM takes a few seconds more than the ideal operation time, nothing fatal happens.

❖ An audio video play back system is another example of Soft Real Time system. No potential damage arises if a sample comes late by fraction of a second, for play back.

**Tasks, Processes & Threads** :

- In the Operating System context, a task is defined as the program in execution and the related information maintained by the Operating system for the program
- Task is also known as '*Job*' in the operating systemcontext
- A program or part of it in execution is also called a'*Process*'
- The terms '*Task*', '*job*' and '*Process*' refer to the same entity in the Operating System context and most often they are usedinterchangeably
- A process requires various system resources like CPU for executing the process, memory for storing the code corresponding to the process and associated variables, I/O devices for information exchange etc

**The structure of a Processes**

- The concept of '*Process*' leads to concurrent execution (pseudo parallelism) of tasks and thereby the efficient utilization of the CPU and other system resources
- Concurrent execution is achieved through the sharing of CPU among the processes.
- A process mimics a processor in properties and holds a set of registers, process status, a Program Counter (PC) to point to the next executable instruction of the process, a stack for holding the local variables associated with the process and the code corresponding to the process
- A process, which inherits all the properties of the CPU, can be considered as a virtual processor, awaiting its turn to have its properties switched into the physical processor

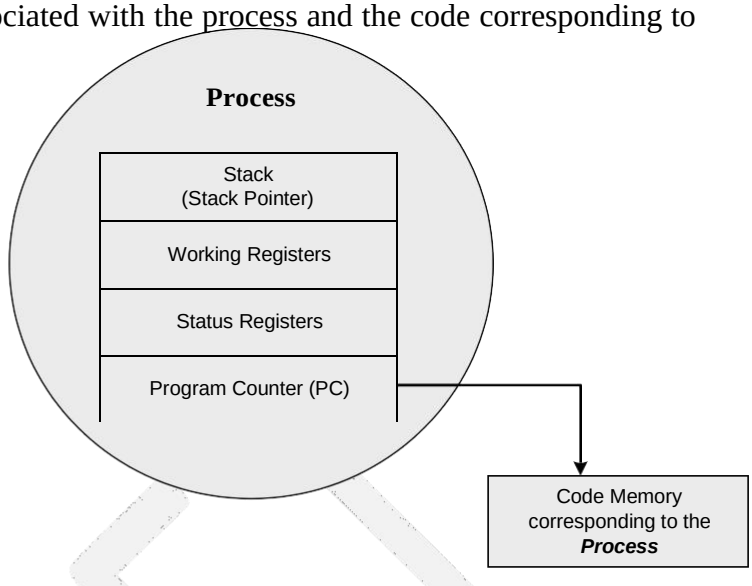



**Figure: 4 Structure of a Process**

- When the process gets its turn, its registers and Program counterregister becomes mapped to the physical registers of the CPU

## Process States & State Transition

- The creation of a process to its termination is not a single stepoperation
- The process traverses through a series of states during its transition from the newly created state to the terminated state
- The cycle through which a process changes its state from '*newly created*' to '*execution completed*' is known as '*Process Life Cycle*'. The various states through which a process traverses through during a Process Life Cycle indicates the current status of the process with respect to time and also provides information on what it is allowed to donext

## Process States & State Transition:

- **Created State:** The state at which a process is being created is referred as 'Created State'. The Operating System recognizes a process in the '*Created State*' but no resources are allocated to the process
- **Ready State:** The state, where a process is incepted into the memory and awaiting the processor time for execution, is known as '*Ready State*'. At this stage, the process is placed in the '*Ready list*' queue maintained by the OS
- **Running State:** The state where in the source code instructions corresponding to the process is being executed is called '*Running State*'. Running state is the state at which the process execution happens

- . **Blocked State/Wait State:** Refers to a state where a running process is temporarily suspended from execution and does not have immediate access to resources. The blocked state might have invoked by various conditions like- the process enters a wait state for an event to occur (E.g. Waiting for user inputs such as keyboard input) or waiting for getting access to a shared resource like semaphore, mutex etc

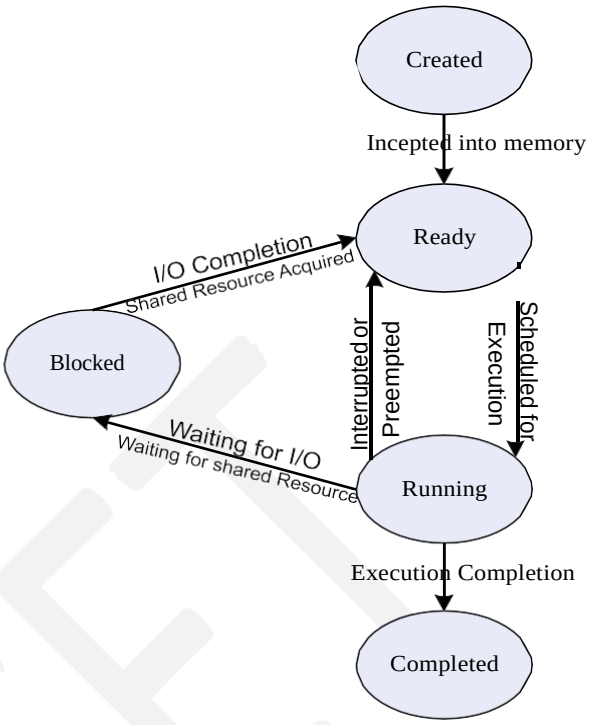



**Figure 6.Process states and State transition**

- **Completed State:** A state where the process completes its execution

- The transition of a process from one state to another is known as '*State transition*'
- When a process changes its state from Ready to running or from running toblocked or terminated or from blocked to running, the CPU allocation for the process may alsochange

## Threads

- A *thread* is the primitive that can execute code
- A *thread* is a single sequential flow of control within a process
- '*Thread*' is also known as lightweight process
- A process can have many threads of execution

- Different threads, which are part of a process, share the same address space; meaning they share the data memory, code memory and heap memory area
- Threads maintain their own thread status (CPU register values), Program Counter (PC) and stack

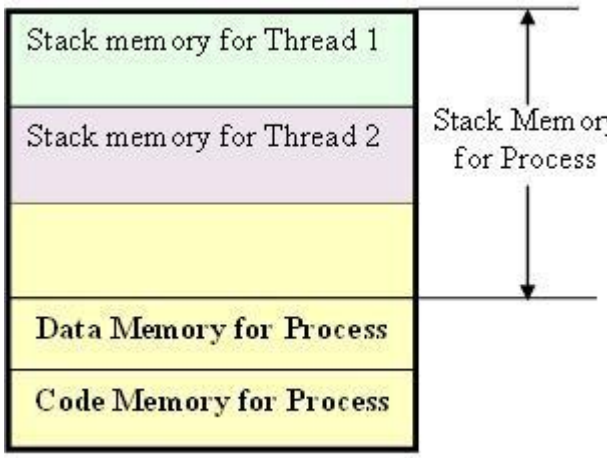



**Figure 7 Memory organization of process and its associated Threads**

## The Concept of multithreading

Use of multiple threads to execute a process brings the following advantage.

- Better memory utilization. Multiple threads of the same process share the address space for data memory. This also reduces the complexity of inter thread communication since variables can be shared across the threads.
  - Since the process is split into different threads, when one thread enters a wait state, the CPU can be utilized by other threads of the process that do not require the event, which the other thread is waiting, for processing. This speeds up the execution of theprocess.

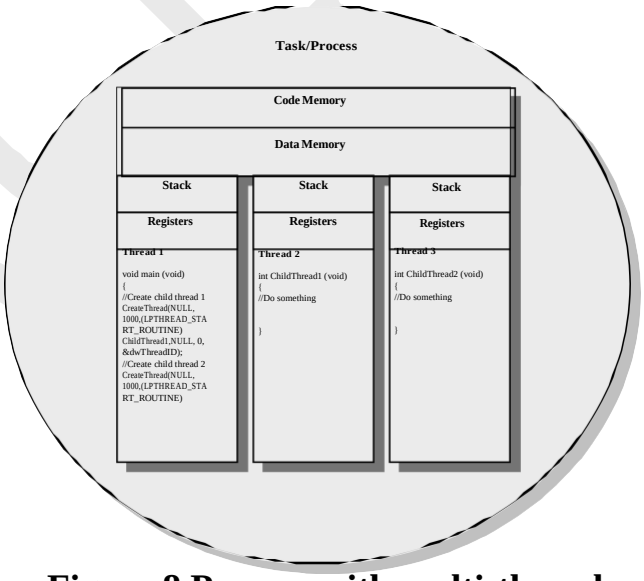



**Figure 8 Process with multi-threads**

- Efficient CPU utilization. The CPU is engaged all time.

## Thread V/s Process

| Thread | Process |
|---|---|
| Thread is a single unit of execution and is part of process. | Process is a program in execution and contains one or more threads. |
| A thread does not have its own data memory and heap memory. It shares the data memory and heap memory with other threads of the same process. | Process has its own code memory, data memory and stack memory. |
| A thread cannot live independently; it lives within the process. | A process contains at least one thread. |
| There can be multiple threads in a process. The first thread (main thread) calls the main function and occupies the start of the stack memory of the process. | Threads within a process share the code, data and heap memory. Each thread holds separate memory area for stack (shares the total stack memory of the process). |
| Threads are very inexpensive to create | Processes are very expensive to create. Involves many OS overhead. |
| Context switching is inexpensive and fast | Context switching is complex and involves lot of OS overhead and is comparatively slower. |
| If a thread expires, its stack is reclaimed by the process. | If a process dies, the resources allocated to it are reclaimed by the OS and all the associated threads of the process also dies. |

## Advantages of Threads:

1. **Better memory utilization:** Multiple threads of the same process share the address space for data memory. This also reduces the complexity of inter thread communication since variables can be shared across thethreads.
2. **Efficient CPU utilization:** The CPU is engaged all time.

3. **Speeds up the execution of the process:** The process is split into different threads, when one thread enters a wait state, the CPU can be utilized by other threads of the process that do not require the event, which the other thread is waiting, for processing.

## Types of Multitasking :

Depending on how the task/process execution switching act is implemented, multitasking can is classified into

- **Co-operative Multitasking:** Co-operative multitasking is the most primitive form of multitasking in which a task/process gets a chance to execute only when the currently executing task/process voluntarily relinquishes the CPU. In this method, any task/process can avail the CPU as much time as it wants. Since this type of implementation involves the mercy of the tasks each other for getting the CPU time for execution, it is known as co-operative multitasking. If the currently executing task is non-cooperative, the other tasks may have to wait for a long time to get the CPU

- **Preemptive Multitasking:** Preemptive multitasking ensures that every task/process gets a chance to execute. When and how much time a process gets is dependent on the implementation of the preemptive scheduling. As the name indicates, in preemptive multitasking, the currently running task/process is preempted to give a chance to other tasks/process to execute. The preemption of task may be based on time slots or task/processpriority

- **Non-preemptive Multitasking:** The process/task, which is currently given the CPU time, is allowed to execute until it terminates (enters the 'Completed' state) or enters the 'Blocked/Wait' state, waiting for an I/O. The co- operative and non-preemptive multitasking differs in their behavior when they are in the 'Blocked/Wait' state. In co-operative multitasking, the currently executing process/task need not relinquish the CPU when it enters the 'Blocked/Wait' sate, waiting for an I/O, or a shared resource access or an event to occur whereas in non-preemptive multitasking the currently executing task relinquishes the CPU when it waits for an I/O.

## Task Scheduling:

- In a multitasking system, there should be some mechanism in place to share the CPU among the different tasks and to decide which process/task is to be executed at a given point of time

- Determining which task/process is to be executed at a given point of time is known as task/process scheduling

- Task scheduling forms the basis of multitasking

- Scheduling policies forms the guidelines for determining which task is to be executed when
- The scheduling policies are implemented in an algorithm and it is run by the kernel as a service
- The kernel service/application, which implements the scheduling algorithm, is known as '*Scheduler*'
- The task scheduling policy can be *pre-emptive, non-preemptive* or *co- operative*
- Depending on the scheduling policy the process scheduling decision may take place when a process switches its state to
    - '*Ready*' state from '*Running*' state
    - '*Blocked/Wait*' state from '*Running*' state
    - '*Ready*' state from '*Blocked/Wait*' state
    - '*Completed*' state

**Task Scheduling - Scheduler Selection**:

The selection of a scheduling criteria/algorithm should consider

- **CPU Utilization:** The scheduling algorithm should always make the CPU utilization high. CPU utilization is a direct measure of how much percentage of the CPU is being utilized.
- **Throughput:** This gives an indication of the number of processes executed per unit of time. The throughput for a good scheduler should always be higher.
- **Turnaround Time:** It is the amount of time taken by a process for completing its execution. It includes the time spent by the process for waiting for the main memory, time spent in the ready queue, time spent on completing the I/O operations, and the time spent in execution. The turnaround time should be a minimum for a 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. The waiting time should be minimal for a good scheduling algorithm.
- **Response Time:** It is the time elapsed between the submission of a process and the first response. For a good scheduling algorithm, the response time should be as least as possible.

To summarize, a good scheduling algorithm has high CPU utilization, minimum Turn Around Time (TAT), maximum throughput and least response time.

**Task Scheduling - Queues**

The various queues maintained by OS in association with CPU scheduling are

- Job Queue: Job queue contains all the processes in the system
- Ready Queue: Contains all the processes, which are ready for execution and waiting for CPU to get their turn for execution. The Ready queue is empty when there is no process

ready forrunning.

- Device Queue: Contains the set of processes, which are waiting for an I/O device

**Non-preemptive scheduling – First Come First Served (FCFS)/First In First Out (FIFO) Scheduling**:

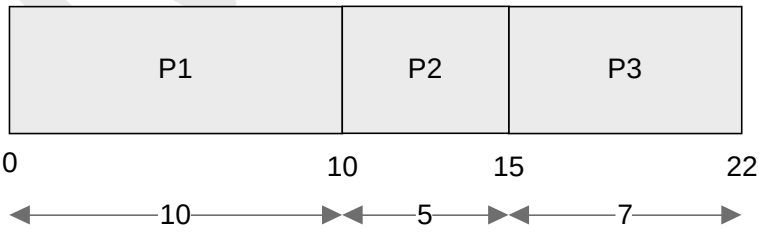
- Allocates CPU time to the processes based on the order in which they enters the '*Ready*' queue
- The first entered process is serviced first
- It is same as any real world application where queue systems are used; E.g. Ticketing

**Drawbacks:**

- Favors monopoly of process. A process, which does not contain any I/O operation, continues its execution until it finishes its task
- In general, FCFS favors CPU bound processes and I/O bound processes may have to wait until the completion of CPU bound process, if the currently executing process is a CPU bound process. This leads to poor device utilization.
- The average waiting time is not minimal for FCFS scheduling algorithm

**EXAMPLE:** Three processes with process IDs P1, P2, P3 with estimated completion time 10, 5, 7 milliseconds respectively enters the ready queue together in the order P1, P2, P3. Calculate the waiting time and Turn Around Time (TAT) for each process and the Average waiting time and Turn Around Time (Assuming there is no I/O waiting for the processes).

**Solution:** The sequence of execution of the processes by the CPU is represented as



Assuming the CPU is readily available at the time of arrival of P1, P1 starts executing without any waiting in the 'Ready' queue. Hence the waiting time for P1 is zero.

Waiting Time for P1 = 0 ms (P1 starts executing first)

Waiting Time for P2 = 10 ms (P2 starts executing after completing P1) Waiting Time for

P3 = 15 ms (P3 starts executing after completing P1 and P2) Average waiting time =

(Waiting time for all processes) / No. of Processes

= (Waiting time for (P1+P2+P3)) / 3

= (0+10+15)/3 = 25/3 = 8.33 milliseconds

Turn Around Time (TAT) for P1 = 10 ms (Time spent in Ready Queue + Execution Time)

Turn Around Time (TAT) for P2 = 15 ms (-Do-)

Turn Around Time (TAT) for P3 = 22 ms (-Do-)

Average Turn Around Time= (Turn Around Time for all processes) / No. of Processes

= (Turn Around Time for (P1+P2+P3)) / 3

= (10+15+22)/3 = 47/3

= 15.66 milliseconds

**Non-preemptive scheduling – Last Come First Served (LCFS)/Last In First Out (LIFO) Scheduling:**

- Allocates CPU time to the processes based on the order in which theyare entered in the '*Ready*' queue
- The last entered process is serviced first

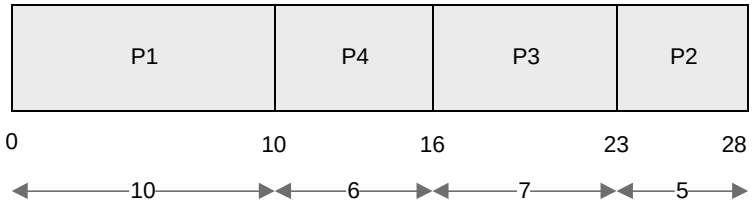
- LCFS scheduling is also known as Last In First Out (LIFO) where the process, which is put last into the '*Ready*' queue, is servicedfirst

**Drawbacks:**

- Favors monopoly of process. A process, which does not contain any I/O operation, continues its execution until it finishes itstask
- In general, LCFS favors CPU bound processes and I/O bound processes may have to wait until the completion of CPU bound process, if the currently executing process is a CPU bound process. This leads to poor device utilization.
- The average waiting time is not minimal for LCFS scheduling algorithm

**EXAMPLE:** Three processes with process IDs P1, P2, P3 with estimated completion time 10, 5, 7 milliseconds respectively enters the ready queue together in the order P1, P2, P3 (Assume only P1 is present in the '*Ready*' queue when the scheduler picks up it and P2, P3 entered '*Ready*' queue after that). Now a new process P4 with estimated completion time 6ms enters the 'Ready' queue after 5ms of scheduling P1. Calculate the waiting time and Turn Around Time (TAT) for each process and the Average waiting time and Turn Around Time (Assuming there is no I/O waiting for the processes).Assume all the processes contain only CPU operation and no I/O operations are involved.

**Solution:** Initially there is only P1 available in the Ready queue and the scheduling sequence will be P1, P3, P2. P4 enters the queue during the execution of P1 and becomes the last process entered the '*Ready*' queue. Now the order of execution changes to P1, P4, P3, and P2 as given below.




The waiting time for all the processes are given as

Waiting Time for P1 = 0 ms (P1 starts executing first)

Waiting Time for P4 = 5 ms (P4 starts executing after completing P1. But P4 arrived after 5ms of execution of P1. Hence its waiting time = Execution start time – Arrival Time = 10-5 = 5)

Waiting Time for P3 = 16 ms (P3 starts executing after completing P1 and P4)

Waiting Time for P2 = 23 ms (P2 starts executing after completing P1, P4 and P3)

Average waiting time = (Waiting time for all processes) / No. of Processes

= (Waiting time for (P1+P4+P3+P2)) / 4

= (0 + 5 + 16 + 23)/4 = 44/4

= 11 milliseconds

Turn Around Time (TAT) for P1 = 10 ms (Time spent in Ready Queue + Execution Time)

Turn Around Time (TAT) for P4 = 11 ms (Time spent in Ready Queue + Execution Time = (Execution Start Time – Arrival Time) + Estimated Execution Time = (10-5) + 6 = 5 +6)

Turn Around Time (TAT) for P3 = 23 ms (Time spent in Ready Queue + Execution Time)

Turn Around Time (TAT) for P2 = 28 ms (Time spent in Ready Queue + Execution Time)

Average Turn Around Time = (Turn Around Time for all processes) / No. of Processes

= (Turn Around Time for (P1+P4+P3+P2)) / 4

= (10+11+23+28)/4 = 72/4

= 18 milliseconds

## Non-preemptive scheduling – Shortest Job First (SJF) Scheduling.

- Allocates CPU time to the processes based on the execution completion time for tasks
- The average waiting time for a given set of processes is minimal in SJF scheduling
- Optimal compared to other non-preemptive scheduling like FCFS

**Drawbacks:**

- A process whose estimated execution completion time is high may not get a chance to execute if more and more processes with least estimated execution time enters the '*Ready*' queue before the process with longest estimated execution time starts its execution
- May lead to the 'Starvation' of processes with high estimated completion time
- Difficult to know in advance the next shortest process in the '*Ready*' queue for scheduling since new processes with different estimated execution time keep entering the '*Ready*' queue at any point of time.

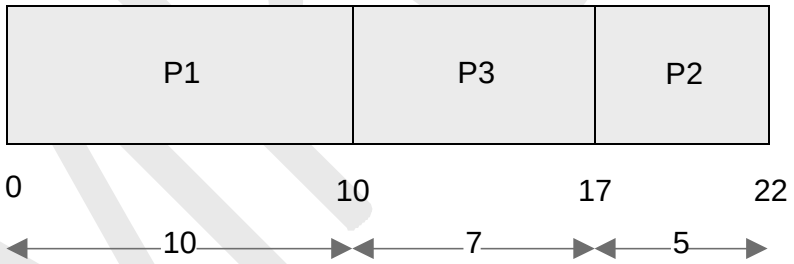
## Non-preemptive scheduling – Priority based Scheduling

- A priority, which is unique or same is associated with each task
- The priority of a task is expressed in different ways, like a priority number, the time required to complete the execution etc.
- In number based priority assignment the priority is a number ranging from 0 to the maximum priority supported by the OS. The maximum level of priority is OS dependent.
- Windows CE supports 256 levels of priority (0 to 255 priority numbers, with 0 being the highest priority)

- The priority is assigned to the task on creating it. It can also be changed dynamically (If the Operating System supports this feature)
- The non-preemptive priority based scheduler sorts the 'Ready' queue based on the priority and picks the process with the highest level of priority for execution

**EXAMPLE:** Three processes with process IDs P1, P2, P3 with estimated completion time 10, 5, 7 milliseconds and priorities 0, 3, 2 (0- highest priority, 3 lowest priority) respectively enters the ready queue together. Calculate the waiting time and Turn Around Time (TAT) for each process and the Average waiting time and Turn Around Time (Assuming there is no I/O waiting for the processes) in priority based scheduling algorithm.

**Solution:** The scheduler sorts the '*Ready*' queue based on the priority and schedules the process with the highest priority (P1 with priority number 0) first and the next high priority process (P3 with priority number 2) as second and so on. The order in which the processes are scheduled for execution is represented as




The waiting time for all the processes are given as

Waiting Time for P1 = 0 ms (P1 starts executing first)

Waiting Time for P3 = 10 ms (P3 starts executing after completing P1)

Waiting Time for P2 = 17 ms (P2 starts executing after completing P1 and P3)

Average waiting time = (Waiting time for all processes) / No. of Processes

= (Waiting time for (P1+P3+P2)) / 3

= (0+10+17)/3 = 27/3

= 9 milliseconds

Turn Around Time (TAT) for P1 = 10 ms (Time spent in Ready Queue + Execution Time)

Turn Around Time (TAT) for P3 = 17 ms (-Do-)

Turn Around Time (TAT) for P2 = 22 ms (-Do-)

Average Turn Around Time= (Turn Around Time for all processes) / No. of Processes

= (Turn Around Time for (P1+P3+P2)) / 3

= (10+17+22)/3 = 49/3

= 16.33 milliseconds

**Drawbacks:**

- Similar to SJF scheduling algorithm, non-preemptive priority based algorithm also possess the drawback of '*Starvation*' where a process whose priority is low may not get a chance to execute if more and more processes with higher priorities enter the '*Ready*' queue before the process with lower priority starts its execution.
- '*Starvation*' can be effectively tackled in priority based non-preemptive scheduling by dynamically raising the priority of the low priority task/process which is under starvation (waiting in the ready queue for a longer time for getting the CPU time)
- The technique of gradually raising the priority of processes which are waiting in the 'Ready' queue as time progresses, for preventing '*Starvation*', is known as '*Aging*'.

**Preemptive scheduling:**

- Employed in systems, which implements preemptive multitasking model
- Every task in the '*Ready*' queue gets a chance to execute. When and how often each process gets a chance to execute (gets the CPU time) is dependent on the type of preemptive scheduling algorithm used for scheduling the processes
- The scheduler can preempt (stop temporarily) the currently executing task/process and select another task from the '*Ready*' queue forexecution
- When to pre-empt a task and which task is to be picked up from the '*Ready*' queue for execution after preempting the current task is purely dependent on the scheduling algorithm

- A task which is preempted by the scheduler is moved to the '*Ready*' queue. The act of moving a '*Running*' process/task into the '*Ready*' queue by the scheduler, without the processes requesting for it is known as '*Preemption*'

- Time-based preemption and priority-based preemption are the two important approaches adopted in preemptive scheduling

**Preemptive scheduling – Preemptive SJF Scheduling/ Shortest Remaining Time (SRT):**

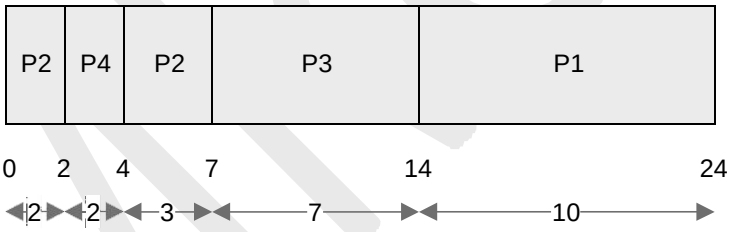
- The *non preemptive SJF* scheduling algorithm sorts the 'Ready' queue only after the current process completes execution or enters wait state, whereas the *preemptive SJF* scheduling algorithm sorts the 'Ready' queue when a new process enters the 'Ready' queue and checks whether the execution time of the new process is shorter than the remaining of the total estimated execution time of the currently executing process

- If the execution time of the new process is less, the currently executing process is preempted and the new process is scheduled for execution

- Always compares the execution completion time (ie the remaining execution time for the new process) of a new process entered the 'Ready' queue with the remaining time for completion of the currently executing process and schedules the process with shortest remaining time forexecution.

**EXAMPLE:** Three processes with process IDs P1, P2, P3 with estimated completion time 10, 5, 7 milliseconds respectively enters the ready queue together. A new process P4 with estimated completion time 2ms enters the 'Ready' queue after 2ms. Assume all the processes contain only CPU operation and no I/O operations are involved.

**Solution:** At the beginning, there are only three processes (P1, P2 and P3) available in the '*Ready*' queue and the SRT scheduler picks up the process with the Shortest remaining time for execution completion (In this example P2 with remaining time 5ms) for scheduling. Now process P4 with estimated execution completion time 2ms enters the '*Ready*' queue after 2ms of start of execution of P2. The processes are re-scheduled for execution in the followingorder




The waiting time for all the processes are given as

Waiting Time for P2 = 0 ms + (4 -2) ms = 2ms (P2 starts executing first and is interrupted by P4 and has to wait till the completion of P4 to get the next CPU slot)

Waiting Time for P4 = 0 ms (P4 starts executing by preempting P2 since the execution time for completion of P4 (2ms) is less than that of the Remaining time for execution completion of P2 (Here it is 3ms))

Waiting Time for P3 = 7 ms (P3 starts executing after completing P4 and P2)

Waiting Time for P1 = 14 ms (P1 starts executing after completing P4, P2 and P3)

Average waiting time = (Waiting time for all the processes) / No. of Processes

= (Waiting time for (P4+P2+P3+P1)) / 4

= (0 + 2 + 7 + 14)/4 = 23/4

= 5.75 milliseconds

Turn Around Time (TAT) for P2 = 7 ms (Time spent in Ready Queue + Execution Time)

Turn Around Time (TAT) for P4 = 2 ms
(Time spent in Ready Queue + Execution Time = (Execution Start Time – Arrival Time) + Estimated Execution Time = (2-2) + 2)

Turn Around Time (TAT) for P3 = 14 ms (Time spent in Ready Queue + Execution Time)

Turn Around Time (TAT) for P1 = 24 ms (Time spent in Ready Queue + Execution Time)

Average Turn Around Time = (Turn Around Time for all the processes) / No. of Processes

= (Turn Around Time for (P2+P4+P3+P1)) / 4

= (7+2+14+24)/4 = 47/4

= 11.75 milliseconds

## Preemptive scheduling – Round Robin (RR) Scheduling:

- Each process in the 'Ready' queue is executed for a pre-defined time slot.
- The execution starts with picking up the first process in the 'Ready' queue. It is executed for a pre-defined time

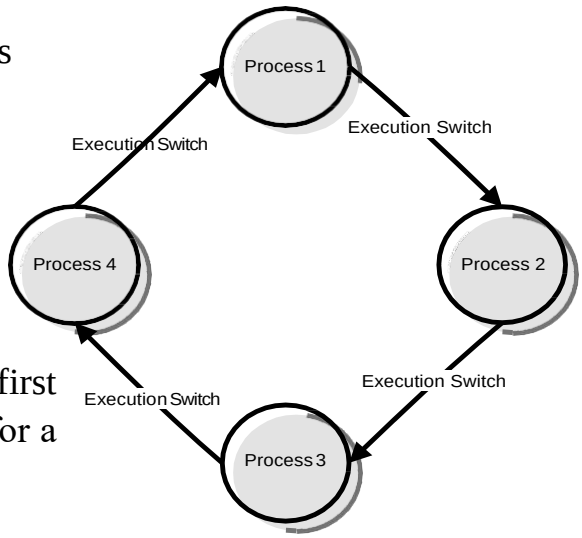
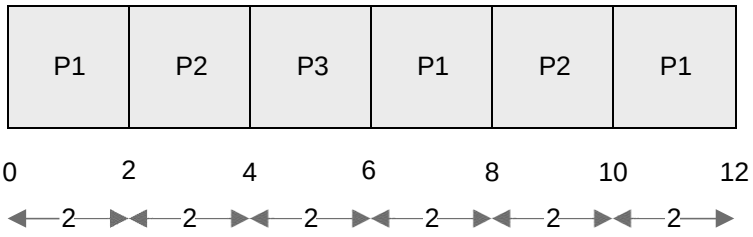



**Figure 11 Round Robin Scheduling**

- When the pre-defined time elapses or the process completes (before the pre- defined time slice), the next process in the 'Ready' queue is selected for execution.
- This is repeated for all the processes in the 'Ready' queue
- Once each process in the 'Ready' queue is executed for the pre-defined time period, the scheduler comes back and picks the first process in the 'Ready' queue again for execution.
- Round Robin scheduling is similar to the FCFS scheduling and the only difference is that a time slice based preemption is added to switch the execution between the processes in the 'Ready' queue

**EXAMPLE:** Three processes with process IDs P1, P2, P3 with estimated completion time 6, 4, 2 milliseconds respectively, enters the ready queue together in the order P1, P2, P3. Calculate the waiting time and Turn Around Time (TAT) for each process and the Average waiting time and Turn Around Time (Assuming there is no I/O waiting for the processes) in RR algorithm with Time slice= 2ms.

**Solution:** The scheduler sorts the '*Ready*' queue based on the FCFS policy and picks up the first process P1 from the 'Ready' queue and executes it for the time slice 2ms. When the time slice is expired, P1 is preempted and P2 is scheduled for execution. The Time slice expires after 2ms of execution of P2. Now P2 is preempted and P3 is picked up for execution. P3 completes its execution within the time slice and the scheduler picks P1 again for execution for the next time slice. This procedure is repeated till all the processes are serviced. The order in which the processes are scheduled for execution is represented as



The waiting time for all the processes are given as

Waiting Time for P1 = 0 + (6-2) + (10-8) = 0+4+2= 6ms (P1 starts executing first and waits for two time slices to get execution back and again 1 time slice for getting CPU time)

Waiting Time for P2 = (2-0) + (8-4) = 2+4 = 6ms (P2 starts executing after P1 executes for 1 time slice and waits for two time slices to get the CPU time)

Waiting Time for P3 = (4 -0) = 4ms (P3 starts executing after completing the first time slices for P1 and P2 and completes its execution in a single time slice.)

Average waiting time = (Waiting time for all the processes) / No. ofProcesses

= (Waiting time for (P1+P2+P3)) / 3

= (6+6+4)/3 = 16/3

= 5.33 milliseconds

Turn Around Time (TAT) for P1 = 12 ms (Time spent in Ready Queue + Execution Time)

Turn Around Time (TAT) for P2 = 10 ms (-Do-)

Turn Around Time (TAT) for P3 = 6 ms (-Do-)

Average Turn Around Time = (Turn Around Time for all the processes) / No. of Processes

= (Turn Around Time for (P1+P2+P3)) / 3

= (12+10+6)/3 = 28/3

= 9.33 milliseconds.

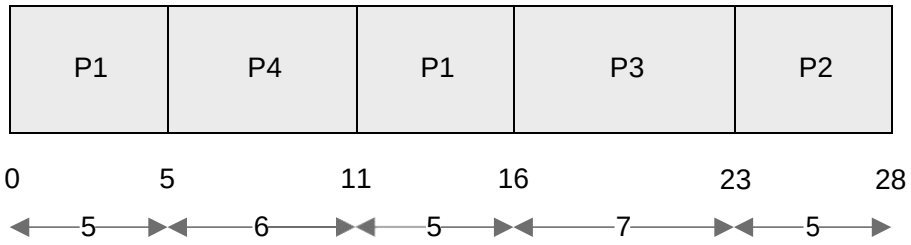
**Preemptive scheduling – Priority based Scheduling**

- Same as that of the ***non-preemptive priority*** based scheduling except for the switching of execution between tasks
- In ***preemptive priority*** based scheduling, any high priority process entering the 'Ready' queue is immediately scheduled for execution whereas in the ***non-preemptive*** scheduling any high priority process entering the 'Ready' queue is scheduled only after the currently executing process completes its execution or only when it voluntarily releases theCPU

- The priority of a task/process in preemptive priority based scheduling is indicated in the same way as that of the mechanisms adopted for non- preemptive multitasking.

**EXAMPLE:** Three processes with process IDs P1, P2, P3 with estimated completion time 10, 5, 7 milliseconds and priorities 1, 3, 2 (0- highest priority, 3 lowest priority) respectively enters the ready queue together. A new process P4 with estimated completion time 6ms and priority 0 enters the 'Ready' queue after 5ms of start of execution of P1. Assume all the processes contain only CPU operation and no I/O operations are involved.

**Solution:** At the beginning, there are only three processes (P1, P2 and P3) available in the '*Ready*' queue and the scheduler picks up the process with the highest priority (In this example P1 with priority 1) for scheduling. Now process P4 with estimated execution completion time 6ms and priority 0 enters the '*Ready*' queue after 5ms of start of execution of P1. The processes are re-scheduled for execution in the following order



The waiting time for all the processes are given as

Waiting Time for P1 = 0 + (11-5) = 0+6 =6 ms (P1 starts executing first and gets Preempted by P4 after 5ms and again gets the CPU time after completion of P4)

Waiting Time for P4 = 0 ms (P4 starts executing immediately on entering the 'Ready' queue, by preempting P1)

Waiting Time for P3 = 16 ms (P3 starts executing after completing P1 and P4) Waiting Time for P2 = 23 ms (P2 starts executing after completing P1, P4 and P3) Average waiting time = (Waiting time for all the processes) / No. of Processes

= (Waiting time for (P1+P4+P3+P2)) / 4

= (6 + 0 + 16 + 23)/4 = 45/4

= 11.25 milliseconds

Turn Around Time (TAT) for P1 = 16 ms (Time spent in Ready Queue + Execution Time)

Turn Around Time (TAT) for P4 = 6ms (Time spent in Ready Queue + Execution Time
= (Execution Start Time – Arrival Time) + Estimated Execution Time = (5-5) + 6 = 0 + 6)

Turn Around Time (TAT) for P3 = 23 ms (Time spent in Ready Queue + Execution Time)

Turn Around Time (TAT) for P2 = 28 ms (Time spent in Ready Queue + Execution Time)

Average Turn Around Time= (Turn Around Time for all the processes) / No. of Processes

= (Turn Around Time for (P2+P4+P3+P1)) / 4

= (16+6+23+28)/4 = 73/4

= 18.25 milliseconds

## How to choose RTOS:

- The decision of an RTOS for an embedded design is very critical.
- A lot of factors need to be analyzed carefully before making a decisionon the selection of an RTOS.

These factors can be either

1. **Functional**
2. **Non-functional requirements.**

## 1. Functional Requirements:

### 1. Processor support:

It is not necessary that all RTOS's support all kinds of processor architectures.

It is essential to ensure the processor support by the RTOS

### 2. Memory Requirements:

- The RTOS requires ROM memory for holding the OS files and it is normally stored in a non-volatile memory like FLASH.

OS also requires working memory RAM for loading the OS service.

Since embedded systems are memory constrained, it is essential to evaluate the minimal RAM and ROM requirements for the OS under consideration.

**3.** Real-Time Capabilities:

- It is not mandatory that the OS for all embedded systems need to be Real- Time and all embedded OS's are 'Real-Time' in behavior.
- The Task/process scheduling policies plays an important role in the Real- Time behavior of an OS.

### 3. Kernel and Interrupt Latency:

- The kernel of the OS may disable interrupts while executing certain services and it may lead to interrupt latency.
- For an embedded system whose response requirements are high, this latency should be minimal.

**5. Inter process Communication (IPC) and Task Synchronization:** The implementation of IPC and Synchronization is OS kernel dependent.

**6. Modularization Support:**

- Most of the OS's provide a bunch of features.
- It is very useful if the OS supports modularization where in which the developer can choose the essential modules and re-compile the OS image for functioning.
- Support for Networking and Communication:
- The OS kernel may provide stack implementation and driver support for a bunch of communication interfaces and networking.
- Ensure that the OS under consideration provides support for all the interfaces required by the embedded product.

**7. Development Language Support:**

Certain OS's include the run time libraries required for running applications written in languages like JAVA and C++.

The OS may include these components as built-in component, if not , check the availability of the same from a third party.

**2. Non-Functional Requirements:**

**1. Custom Developed or Off the Shelf:**

- It is possible to go for the complete development of an OS suiting the embedded system needs or use an off the shelf, readily availableOS.
- It may be possible to build the required features by customizing an open source OS.
- The decision on which to select is purely dependent on the development cost, licensing fees for the OS, development time and availability of skilled resources.

**2. Cost:**

The total cost for developing or buying the OS and maintaining it in terms of commercial product and custom build needs to be evaluated before taking a decision on the selection of OS.

### 3. Development and Debugging tools Availability:

- The availability of development and debugging tools is a critical decision making factor in the selection of an OS for embedded design.
- Certain OS's may be superior in performance, but the availability of tools for supporting the development may be limited.

### 4. Ease of Use:

How easy it is to use a commercial RTOS is another important feature that needs to be considered in the RTOS selection.

### 5. After Sales:

For a commercial embedded RTOS, after sales in the form of e-mail, on-call services etc. for bug fixes, critical patch updates and support for production issues etc. should be analyzed thoroughly.