



# **MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY**

Permanently Affiliated to JNTUH and Approved by AICTE, New Delhi

## **LECTURE NOTES**

**ON**

## **EMBEDDED SYSTEMS DESIGN**

**IV B. Tech I semester (R20)  
(2023-24)**

**Faculty Member**

**Mr. M. Ramanjaneyulu**

**Associate Professor/ECE**

---

---

# **MALLA REDDY COLLEGE OF ENGINEERING AND TECHNOLOGY**

## **IV Year B.Tech. ECE-I Sem**

### **EMBEDDED SYSTEMS DESIGN (R20A0421)**

#### **COURSE OBJECTIVES:**

For embedded systems, the course will enable the students to:

1. Understand the basics of an embedded system.
2. Program an embedded system.
3. To learn the design process of embedded system applications.
4. To understands the RTOS and inter-process communication.
5. To understand different communication interfaces.

#### **UNIT-I 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, design example.

#### **UNIT-II TYPICAL EMBEDDED SYSTEM**

Core of the embedded system-general purpose and domain specific processors, ASICs, PLDs, COTs; Memory-ROM, RAM, memory according to the type of interface, memory shadowing, memory selection for embedded systems; Sensors, actuators and other components-sensors, actuators, seven segment LED, relay, piezo buzzer, push button switch, reset circuit, brownout protection circuit, oscillator circuit real time clock, watch dog timer.

#### **UNIT-III 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; Programming in embedded c.

---

---

## **UNIT-IV 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; task communication-shared memory, message passing.

## **UNIT-V COMMUNICATION INTERFACE**

Onboard communication interfaces-I2C, SPI, UART, 1 wire interface, parallel interface; External communication interfaces-RS232 and RS485,USB, infrared, Bluetooth, wi-Fi, zigbee, GPRS; Automotive networks and sensor networks.

### **TEXT BOOKS:**

1. Computers as Components –Wayne Wolf, Morgan Kaufmann (second edition).
2. Introduction to Embedded Systems - shibu k v, Mc Graw Hill Education.

### **REFERENCE BOOKS:**

1. Embedded System Design -frank vahid, tony grivargis, john Wiley.
2. Embedded Systems- An integrated approach - Lyla b das, Pearson education 2012.
3. Embedded Systems – Raj Kamal, TMH
4. An embedded Software Primer, David e Simon, Pearson education

### **COURSE OUTCOMES:**

Upon completion of this course, the students will be able to:

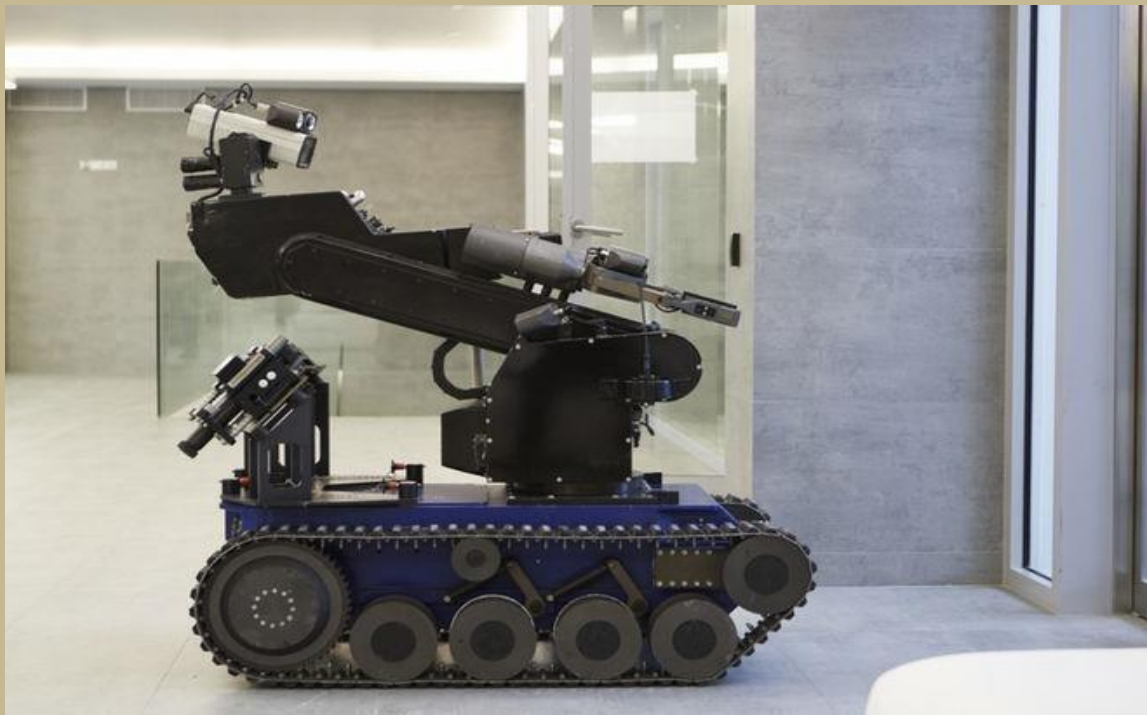
1. Understand and design the embedded systems
  2. Learn the basics of OS and RTOS
  3. Understand types of memory and interfacing to external world
  4. Understand embedded firmware design approaches
-

# **EMBEDDED SYSTEM DESIGN**

## **IV YEAR ECE**

### **UNIT-I**

### **INTRODUCTION TO EMBEDDED SYSTEMS**



**MALLA REDDY COLLEGE OF  
ENGINEERING & TECHNOLOGY**

Permanently Affiliated 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:**

<b>General Purpose Computing System</b>	<b>Embedded System</b>
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

## History of Embedded Systems:

- First Recognized Modern Embedded System: Apollo Guidance Computer (AGC) developed by [Charles Stark Draper](#) at the MIT Instrumentation Laboratory.

- It has two modules
  - 1.Command module(CM) 2.Lunar Excursion module(LEM)
- RAM size 256 , 1K ,2K words
- ROM size 4K,10K,36K words
- Clock frequency is 1.024MHz
- 5000 ,3-input RTL NOR gates are used
- User interface is DSKY(display/Keyboard)



First Mass Produced Embedded System: Autonetics **D-17** Guidance computer for Minuteman-I missile

## Classification of Embedded Systems:

- ☐ Based on Generation
- ☐ Based on Complexity & Performance Requirements
- ☐ Based on deterministic behavior
- ☐ Based on Triggering

(March-2017)

### 1. Embedded Systems - Classification based on Generation

- **First Generation:** The early embedded systems built around 8-bit microprocessors like 8085 and Z80 and 4-bit microcontrollers  
**EX. stepper motor control units, Digital Telephone Keypads etc.**
- Second Generation:** Embedded Systems built around 16-bit microprocessors and 8 or 16-bit microcontrollers, following the first generation embedded systems  
**EX.SCADA, Data Acquisition Systems etc.**
- Third Generation:** Embedded Systems built around high performance 16/32 bit Microprocessors/controllers, Application Specific Instruction set processors like Digital Signal Processors (DSPs), and Application Specific Integrated Circuits (ASICs).The instruction set is complex and powerful.  
**EX. Robotics, industrial process control, networking etc.**

**Fourth Generation:** Embedded Systems built around System on Chips (SoC's), Re-configurable processors and multicore processors. It brings high performance, tight integration and miniaturization into the embedded device market

**EX Smart phone devices, MIDs etc.**

## 2. Embedded Systems - Classification based on Complexity & Performance



**Small Scale:** The embedded systems built around low performance and low cost 8 or 16 bit microprocessors/ microcontrollers. It is suitable for simple applications and where performance is not time critical. It may or may not contain OS.



**Medium Scale:** Embedded Systems built around medium performance, low cost 16 or 32 bit microprocessors / microcontrollers or DSPs. These are slightly complex in hardware and firmware. It may contain GPOS/RTOS.



**Large Scale/Complex:** Embedded Systems built around high performance 32 or 64 bit RISC processors/controllers, RSoC or multi-core processors and PLD. It requires complex hardware and software. These system may contain multiple processors/controllers and co-units/hardware accelerators for offloading the processing requirements from the main processor. It contains RTOS for scheduling, prioritization and management.

**3. Embedded Systems - Classification Based on deterministic behavior:** It is applicable for Real Time systems. The application/task execution behavior for an embedded system can be either deterministic or non-deterministic

**These are classified in to two types**

**1 Soft Real time Systems:** Missing a deadline may not be critical and can be tolerated to a certain degree

**2 Hard Real time systems:** Missing a program/task execution time deadline can have catastrophic consequences (financial, human loss of life, etc.)

## 4. Embedded Systems - Classification Based on Triggering:

**These are classified into two types**

**1 Event Triggered :** Activities within the system (e.g., task run-times) are dynamic and depend upon occurrence of different events .

**2 Time triggered:** Activities within the system follow a statically computed schedule (i.e., they are allocated time slots during which they can take place) and thus by nature are predictable.

## Major Application Areas of Embedded Systems:

- ☐ **Consumer Electronics:** Camcorders, Cameras etc.
- ☐ **Household Appliances:** Television, DVD players, washing machine, Fridge, Microwave Oven etc.
- ☐ **Home Automation and Security Systems:** Air conditioners, sprinklers, Intruder detection alarms, Closed Circuit Television Cameras, Fire alarms etc.
- ☐ **Automotive Industry:** Anti-lock breaking systems (ABS), Engine Control, Ignition Systems, Automatic Navigation Systems etc.
- ☐ **Telecom:** Cellular Telephones, Telephone switches, Handset Multimedia Applications etc.
- ☐ **Computer Peripherals:** Printers, Scanners, Fax machines etc.
- ☐ **Computer Networking Systems:** Network Routers, Switches, Hubs, Firewalls etc.
- ☐ **Health Care:** Different Kinds of Scanners, EEG, ECG Machines etc.
- ☐ **Measurement & Instrumentation:** Digital multi meters, Digital CROs, Logic Analyzers PLC systems etc.
- ☐ **Banking & Retail:** Automatic Teller Machines (ATM) and Currency counters, Point of Sales (POS)
- ☐ **Card Readers:** Barcode, Smart Card Readers, Hand held Devices etc.

## Purpose of Embedded Systems:

(DEC2016)

Each Embedded Systems is designed to serve the purpose of any one or a combination of the following tasks.

- Data Collection/Storage/Representation
- Data Communication
- Data (Signal) Processing
- Monitoring
- Control
- Application Specific User Interface

## 1. Data Collection/Storage/Representation:-

- ❖ Performs acquisition of data from the external world.
- ❖ The collected data can be either analog or digital
- ❖ Data collection is usually done for storage, analysis, manipulation and transmission
- ❖ The collected data may be stored directly in the system or may be transmitted to some other systems or it may be processed by the system or it may be deleted instantly after giving a meaningful representation

## 2. Data Communication:-

Embedded Data communication systems are deployed in applications ranging from complex satellite communication systems to simple home networking systems

Embedded Data communication systems are dedicated for data communication

The data communication can happen through a wired interface (like Ethernet, RS-232C/USB/IEEE1394 etc) or wireless interface (like Wi-Fi, GSM, GPRS, Bluetooth, ZigBee etc)

Network hubs, Routers, switches, Modems etc are typical examples for dedicated data transmission embedded systems

## 3. Data (Signal) Processing:-

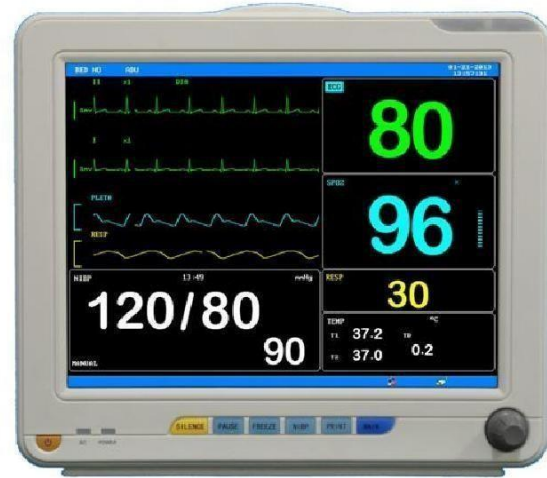
Embedded systems with Signal processing functionalities are employed in applications demanding signal processing like Speech coding, synthesis, audio video codec, transmission applications etc

Computational intensive systems

Employs Digital Signal Processors (DSPs)

## 4. Monitoring:-

- Embedded systems coming under this category are specifically designed for monitoring purpose
- They are used for determining the state of some variables using input sensors
- They cannot impose control over variables.
- Electro Cardiogram (ECG) machine for monitoring the heart beat of a patient is a typical example for this
- The sensors used in ECG are the different Electrodes connected to the patient's body
- Measuring instruments like Digital CRO, Digital Multi meter, Logic Analyzer etc used in Control & Instrumentation applications are also examples of embedded systems for monitoring purpose



## 5. Control:-

- Embedded systems with control functionalities are used for imposing control over some variables according to the changes in input variables
- Embedded system with control functionality contains both sensors and actuators
- Sensors are connected to the input port for capturing the changes in environmental variable or measuring variable
- The actuators connected to the output port are controlled according to the changes in input variable to put an impact on the controlling variable to bring the controlled variable to the specified range



Air conditioner for controlling room temperature is a typical example for embedded system with „Control“ functionality

Air conditioner contains a room temperature sensing element (sensor) which may be a thermistor and a handheld unit for setting up (feeding) the desired temperature

The air compressor unit acts as the actuator. The compressor is controlled according to the current room temperature and the desired temperature set by the end user.

## 6. Application Specific User Interface:-

Embedded systems which are designed for a specific application

Contains Application Specific User interface (rather than general standard UI ) like key board, Display units etc



## COMPLEX SYSTEM AND MICROPROCESSORS:

### ➤ **Three main tasks or components in embedded system design:**

- Selecting and integrating hardware to give computer like functionalities
- Dumping main application software generally into flash or ROM and the application software performs concurrently the number of tasks.
- Integrating with a real time operating system (RTOS), this supervises the application software tasks running on the hardware and organizes the accesses to system resources according to priorities and timing constraints of tasks in the system.

## Embedding Computers:

- **Whirlwind**, a computer designed at MIT in the late 1940s and early 1950s. Whirlwind was also the first computer designed to support *real-time* operation and was originally conceived as a mechanism for controlling an **aircraft simulator**. It was extremely large physically compared to today's computers(e.g., it contained over 4,000 vacuum tubes).
- Very-large-scale integration (VLSI) is the process of creating an integrated circuit (IC) by combining thousands of transistors into a single chip. VLSI began in the 1970s. A microprocessor is a single-chip CPU. Very large scale integration (VLSI) technology allowed us to put a complete CPU on a single chipsince 1970s, but those CPUs were very simple.

In 1971 the **first microprocessor the Intel 4004 invented by Ted Hoff**, was designed for an embedded application, namely, a calculator. The calculator was not a general-purpose computer—it merely provided basic arithmetic functions. The **HP-35 was the first handheld calculator** to perform transcendental functions. It was introduced in 1972, so it used several chips to implement the CPU, rather than a single-chip microprocessor.

- **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.
- **Microprocessors** are usually classified according to their word length.
  - An 8-bit *microcontroller* is designed for low-cost applications and includes on-board memory and I/O devices'

- 16-bit microcontroller is often used for more sophisticated applications that may require either longer word lengths or off-chip I/O and memory;
- 32-bit **RISC** microprocessor offers very high performance for computation-intensive applications.

➤ **House Hold uses of microprocessor:**

- The typical **microwave oven** has at least one microprocessor to control oven operation.
- Many houses have **advanced thermostat systems**, which change the temperature level at various times during the day.

## Characteristics of Embedded Computing Applications:

- a. Complex Algorithms
- b. User Interface
- c. Real Time
- d. Multirate
- e. Manufacturing Cost
- f. Power

**Complex algorithms:** The operations performed by the microprocessor may be very sophisticated. For example, the microprocessor that controls an automobile engine must perform complicated filtering functions to optimize the performance of the car while minimizing pollution and fuel utilization.

**User interface:** Microprocessors are frequently used to control complex user interfaces that may include multiple menus and many options. The moving maps in Global Positioning System (GPS) navigation are good examples of sophisticated user interfaces.

To make things more difficult, embedded computing operations must often be performed to meet deadlines:

**Real time:** Many embedded computing systems have to perform in real time— if the data is not ready by a certain deadline, the system breaks. In some cases, failure to meet a deadline is unsafe and can even endanger lives. In other cases, missing a deadline does not create safety problems but does create unhappy customers— missed deadlines in printers, for example, can result in scrambled pages.

➤ **Multirate:** Not only must operations be completed by deadlines, but many embedded computing systems have several real-time activities going on at the same time. They may simultaneously control some operations that run at slow rates and others that run at high rates. Multimedia applications are prime examples of **multirate** behaviour. The audio and video portions of a multimedia stream run at very different rates, but they must remain closely synchronized. Failure to meet a deadline on either the audio or video portions spoils the perception of the entire presentation.

Costs of various sorts are also very important:

➤ **Manufacturing cost:** The total cost of building the system is very important in many cases. Manufacturing cost is determined by many factors, including the type of microprocessor used, the amount of memory required, and the types of I/O devices.

➤ **Power and energy:** Power consumption directly affects the cost of the hardware, since a larger power supply may be necessary. Energy consumption affects battery life, which is important in many applications, as well as heat consumption, which can be important even in desktop applications.

**Challenges in Embedded Computing System Design:**

- i. *How much hardware do we need?*
- ii. *How do we meet deadlines?*
- iii. *How do we minimize power consumption?*
- iv. *How do we design for upgradability?*
- v. *Does it really work?*
- vi. *Complex testing*
- vii. *Limited observability and controllability*
- viii. *Restricted development environments*

External constraints are one important source of difficulty in embedded system design. Let's consider some important problems that must be taken into account in embedded system design.

***How much hardware do we need?***

We have a great deal of control over the amount of computing power we 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.

***How do we meet deadlines?***

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.

***How do we minimize power consumption?***

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, 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.

***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.

***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. If we wait until we have a running system and try to eliminate the bugs, we will be too late—we won't find enough bugs, it will be too expensive to fix them, and it will take more time.

Let's consider some ways in which the nature of embedded computing machines makes their design more difficult.

***Complex testing:*** Exercising an embedded system is generally more difficult than typing in some data. We may have to run a real machine in order to generate the proper data. The timing of data is often important, meaning that we cannot separate the testing of an embedded computer from the machine in which it is embedded.

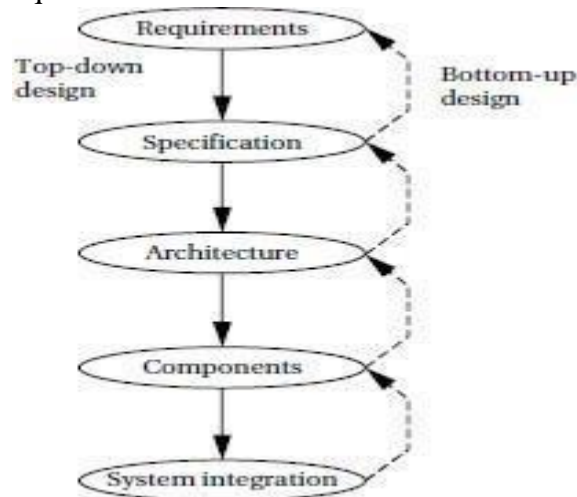
***Limited observability and controllability:*** Embedded computing systems usually do not come with keyboards and screens. This makes it more difficult to see what is going on and to affect the system's operation. We may be forced to watch the values of electrical signals on the microprocessor bus, for example, to know what is going on inside the system. Moreover, in real-time applications we may not be able to easily stop the system to see what is going on inside.

***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.

### 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-downview, 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 digital audio 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.

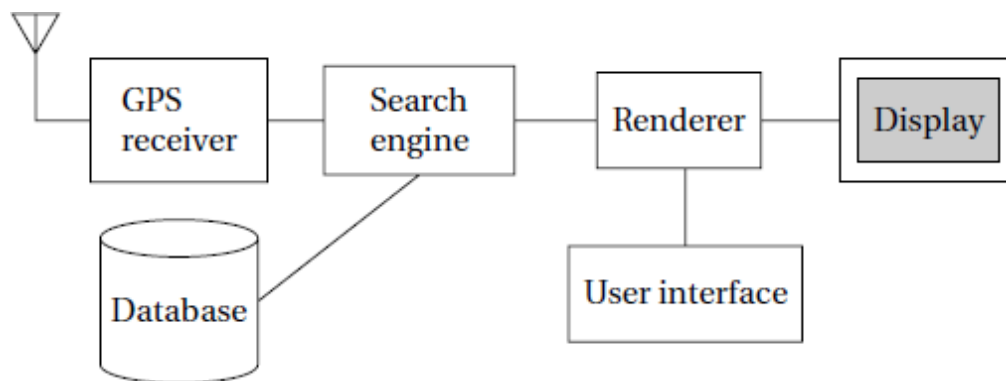
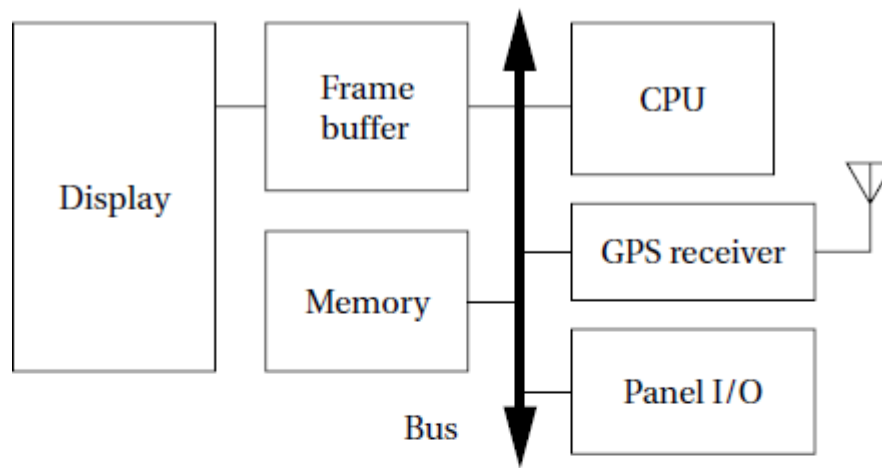
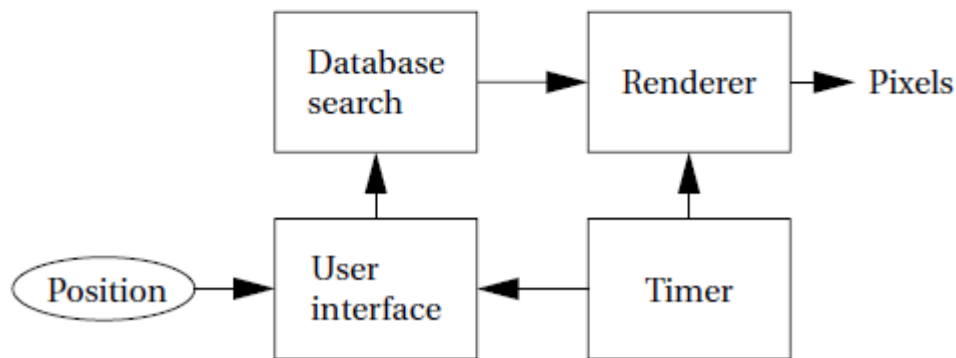


FIG: BLOCK DIAGRAM FOR THE MOVING MAP

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.



### Hardware



### Software

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

- The **modern camera** is a prime example of the powerful features that can be added under microprocessor control.
- **Digital Television** uses embedded processors

## DESIGN EXAMPLE:

### *BMW 850i brake and stability control system*

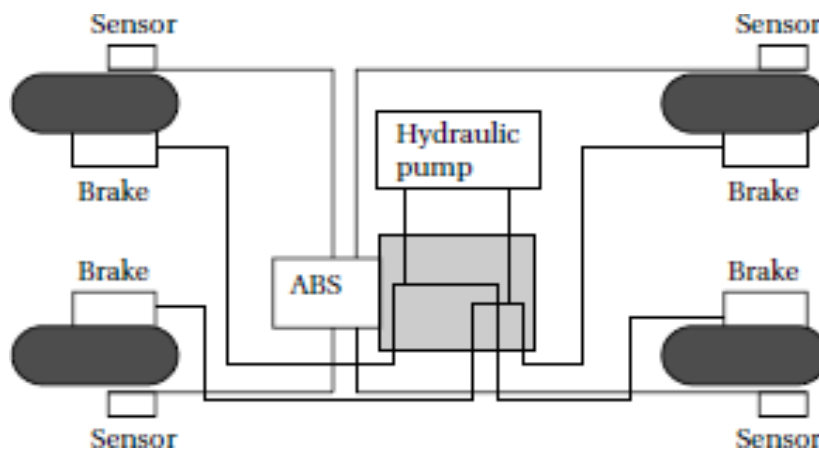
The BMW 850i was introduced with a sophisticated system for controlling the wheels of the car.

An antilock brake system (ABS) reduces skidding by pumping the brakes. An automatic stability control (ASC \_ T) system intervenes with the engine during maneuvering to improve the car's stability.

These systems actively control critical systems of the car; as control systems, they require inputs from and output to the automobile.

Let's first look at the ABS. The purpose of an ABS is to temporarily release the brake on a wheel when it rotates too slowly—when a wheel stops turning, the car starts skidding and becomes hard to control.

It sits between the hydraulic pump, which provides power to the brakes, and the brakes themselves as seen in the below diagram. The ABS system uses sensors on each wheel to measure the speed of the wheel. The wheel speeds are used by the ABS system to determine how to vary the hydraulic fluid pressure to prevent the wheels from skidding.



- The ASC \_ T system's job is to control the engine power and the brake to improve the car's stability. The ASC \_ T controls four different systems: throttle, ignition timing, differential brake, and (on automatic transmission cars) gear shifting.

## SUMMARY

1. An embedded system is an electronic/electromechanical system designed to perform a specific function and is a combination of both hardware and firmware (software).
2. A general purpose computing system is a combination of generic hardware and general purpose operating system for executing a variety of applications, whereas an embedded
3. System is a combination of special purpose hardware and embedded OS/firmware for executing a specific set of applications.
4. Apollo Guidance Computer (AGC) is the first recognized modern embedded system and Autonetics D-17, the guidance computer for the Minuteman-I missile, was the first mass produced embedded system.
5. Based on the complexity and performance requirements, embedded systems are classified into small-scale, medium-scale and large-scale/complex.
6. The presences of embedded system vary from simple electronic system toys to complex flight and missile control systems.
7. Embedded systems are designed to serve the purpose of any one or combination of data collection/storage/representation, data processing, monitoring, control or application specific user interface.
8. Wearable devices refer to embedded systems which are incorporated into accessories and apparels. It envisions the bonding of embedded technology in our day to day lives.

## OBJECTIVE QUESTIONS

1. Embedded systems are
  - (a) General Purpose
  - (b) Special Purpose
2. Embedded system is
  - (a) An electronic system
  - (b) A pure mechanical system
  - (c) An electro-mechanical system
  - (d) (a) or (c)
3. Which of the following is not true about embedded systems?
  - (a) Built around specialized hardware
  - (b) Always contain an operating system
  - (c) Execution behavior may be deterministic
  - (d) All of these
  - (e) none of these
4. Which of the following is not an example of small scale embedded system?
  - (a) Electronic Barbie doll
  - (b) Simple calculator
  - (c) Cell Phone
  - (d) Electronic toy car

- 
5. The first recognized modern embedded system is
    - (a) Apple computer
    - (b) Apollo Guidance Computer
    - (c) Calculator
    - (d) Radio navigation system
  6. The first mass produced embedded system is
    - (a) Minuteman-I
    - (b) Minuteman-II
    - (c) Autonetics D17
    - (d) Apollo Guidance Computer
  7. Which of the following is (are) an intended purpose of embedded systems?
    - (a) Data collection
    - (b) Data processing
    - (c) Data communication
    - (d) All of these
    - (e) None of these
  8. Which of the following is an example of an embedded system for data communication?
    - (a) USB mass storage device
    - (b) Network router
    - (c) Digital camera
    - (d) Music player
    - (e) All of these
    - (f) None of these
  9. A digital multimeter is an example of embedded system for
    - (a) Data communication
    - (b) Monitoring
    - (c) Control
    - (d) All of these
    - (e) None of these
  10. Which of the following is an example of an embedded system for signal processing?
    - (a) Apple iPOD
    - (b) Sandisk USB mass storage device
    - (c) both a and b
    - (d) None of these

## **Reference Text Books:-**

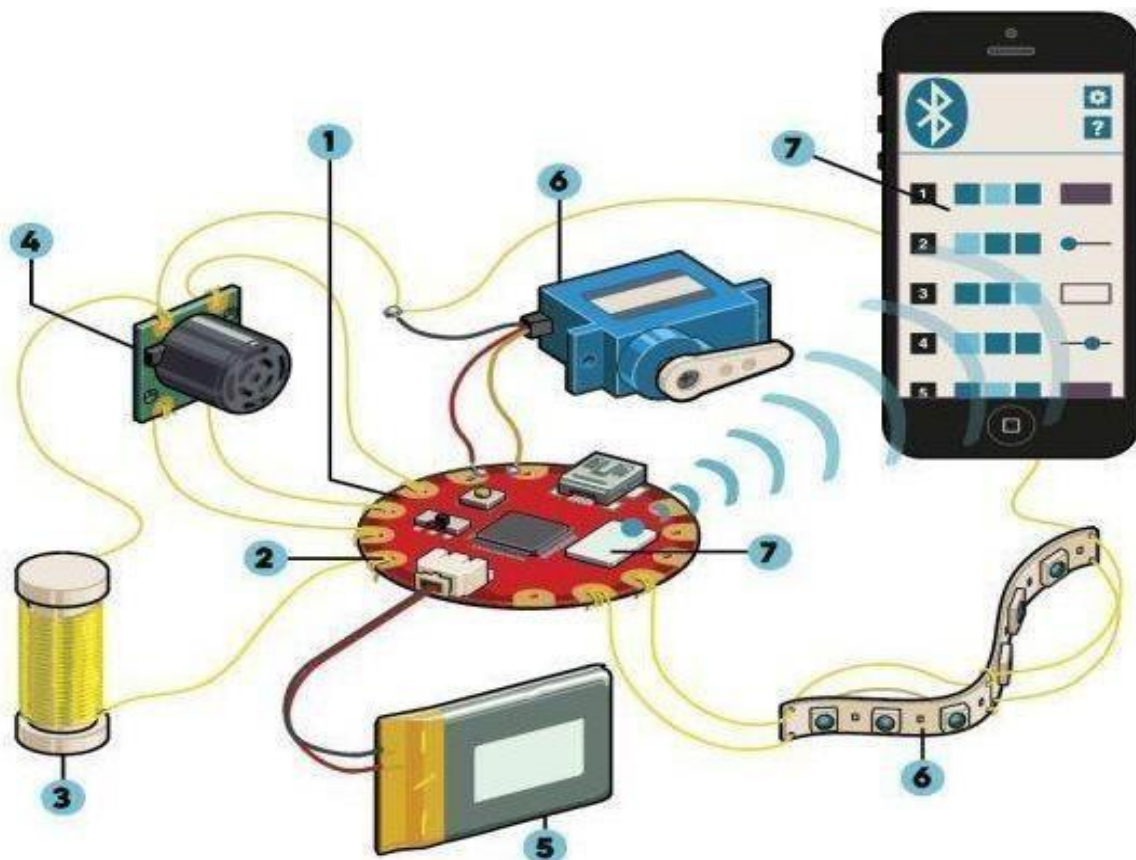
- 1. Introduction to Embedded Systems – Shibu K.V Mc Graw Hill**
- 2. Computers as Components –Wayne Wolf-morgan Kaufmann publications**

IV ECE

## EMBEDDED SYSTEM DESIGN

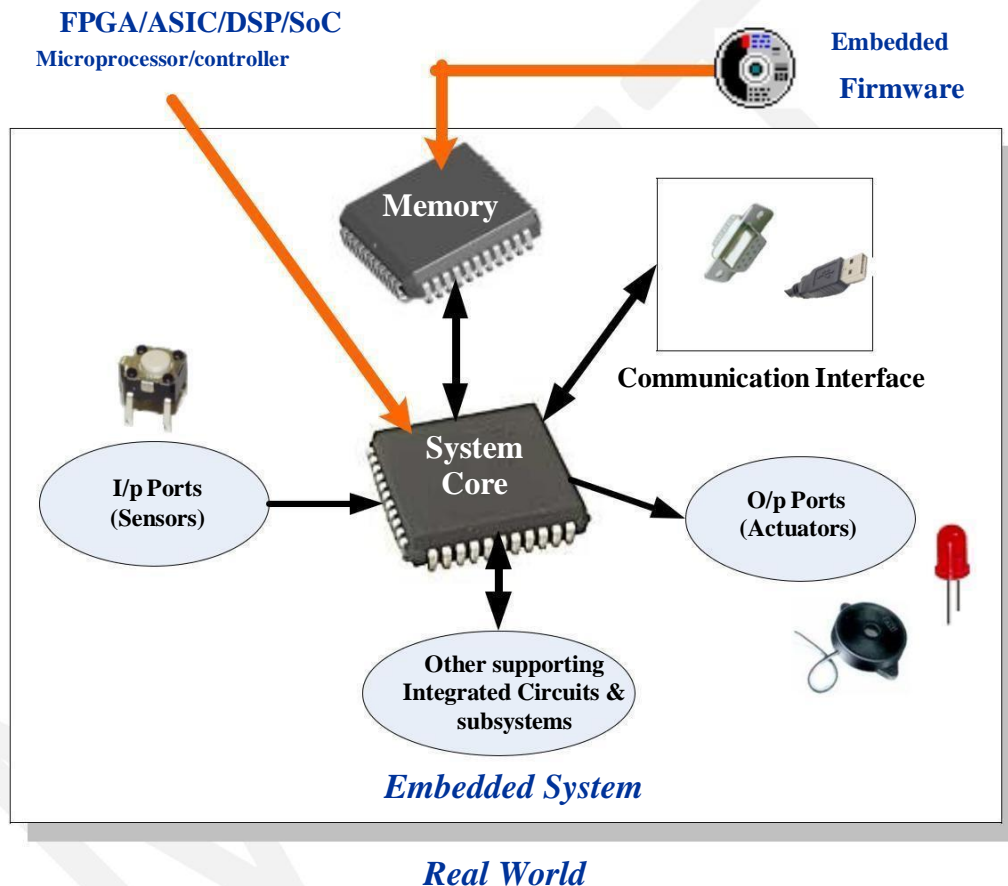
### UNIT-II

### 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 as 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 independent working, 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 Vs Microcontroller:**

<b>Microprocessor</b>	<b>Microcontroller</b>
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“

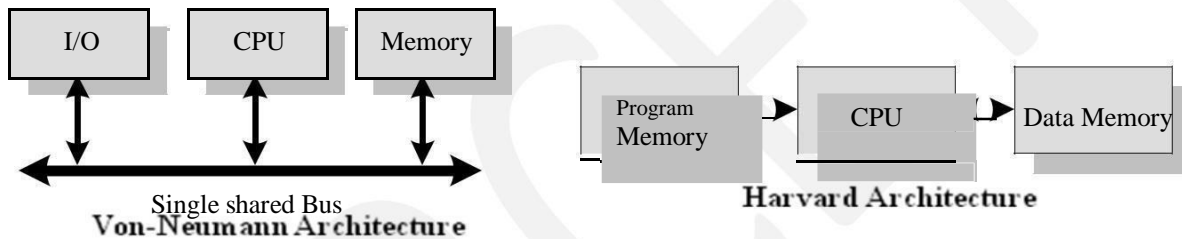
- A typical Digital Signal Processor incorporates the following key units
  - ❖ Program Memory
  - ❖ Data Memory
  - ❖ Computational Engine
  - ❖ I/O Unit
- Audio video signal processing, telecommunication and multimedia applications are typical examples where DSP is employed

### 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 V/s Von-Neumann Processor/Controller Architecture

- The terms Harvard and Von-Neumann refers to the processor architecture design.
- Microprocessors/controllers based on the **Von-Neumann** architecture shares a single common bus for fetching both instructions and data. Program instructions and data are stored in a common main memory
- Microprocessors/controllers based on the **Harvard** architecture will have separate data bus and instruction bus. This allows the data transfer and program fetching to occur simultaneously on both buses
- With Harvard architecture, the data memory can be read and written while the program memory is being accessed. These separated data memory and code memory buses allow one instruction to execute while the next instruction is fetched (“Pre-fetching”)



### Harvard V/s Von-Neumann Processor/Controller 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 <sup>†</sup>
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

**Big-endian V/s Little-endian processors:**

Endianness specifies the order in which the data is stored in the memory by processor operations in a multi byte system (Processors whose word size is greater than one byte). Suppose the word length is two byte then data can be stored in memory in two different ways

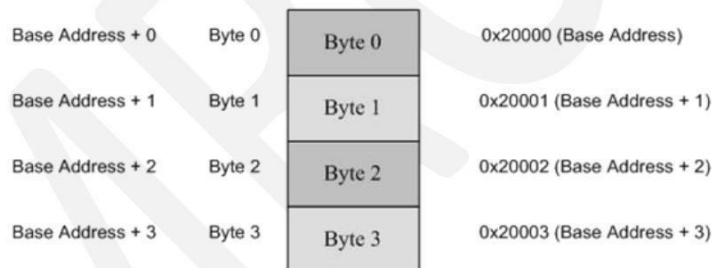
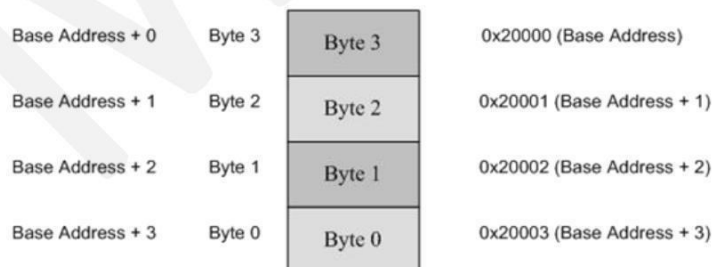
- Higher order of data byte at the higher memory and lower order of data byte at location just below the higher memory
- Lower order of data byte at the higher memory and higher order of data byte at location just below the higher memory



*Little-endian* means the lower-order byte of the data is stored in memory at the lowest address, and the higher-order byte at the highest address. (The little end comes first)

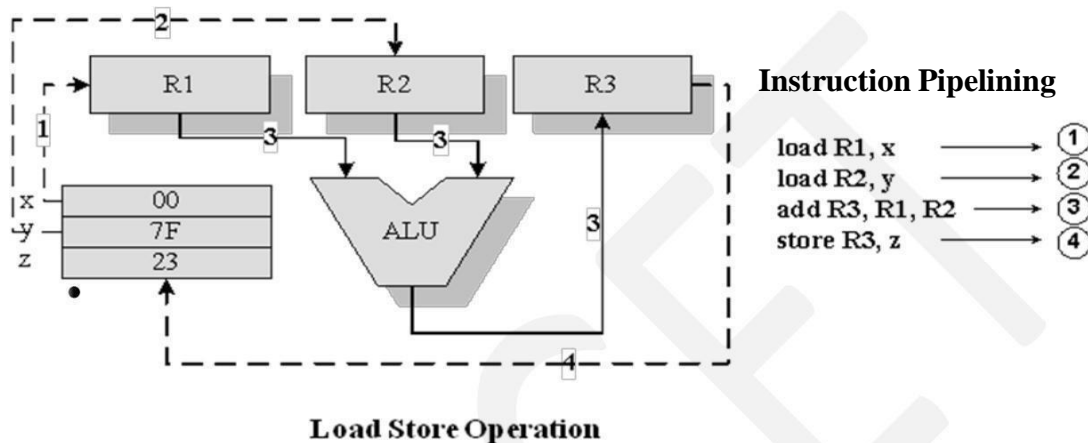


*Big-endian* means the higher-order byte of the data is stored in memory at the lowest address, and the lower-order byte at the highest address. (The big end comes first.)

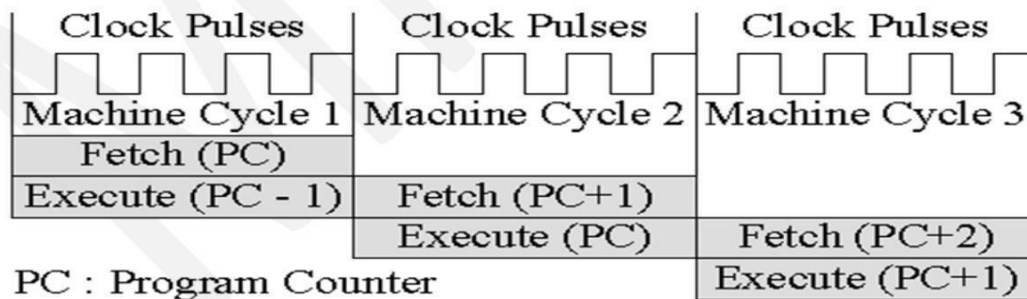
**Big-endian V/s Little-endian processors****Little-endian Operation****Big-endian Operation**

### Load Store Operation & Instruction Pipelining:

The RISC processor instruction set is orthogonal and it operates on registers. The memory access related operations are performed by the special instructions *load* and *store*. If the operand is specified as memory location, the content of it is loaded to a register using the *load* instruction. The instruction *store* stores data from a specified register to a specified memory location



- The conventional instruction execution by the processor follows the fetch-decode-execute sequence
- The „fetch“ part fetches the instruction from program memory or code memory and the decode part decodes the instruction to generate the necessary control signals



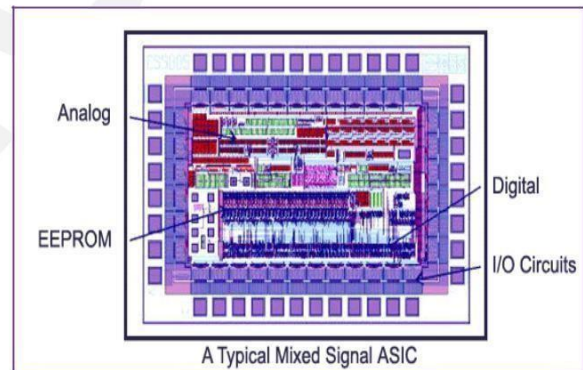
### The Single stage pipelining concept

- The execute stage reads the operands, perform ALU operations and stores the result. In conventional program execution, the fetch and decode operations are performed in sequence

- During the decode operation the memory address bus is available and if it possible to effectively utilize it for an instruction fetch, the processing speed can be increased
- In its simplest form instruction pipelining refers to the overlapped execution of instructions

### 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.

- Some ASICs are proprietary products , the developers are not interested in revealing the internal details.

### 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

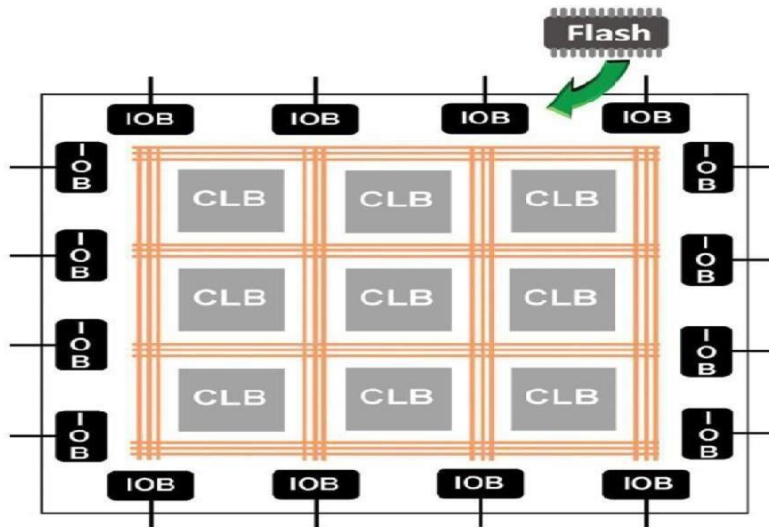
### Programmable Logic Devices (PLDs) – CPLDs and FPGA

- 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



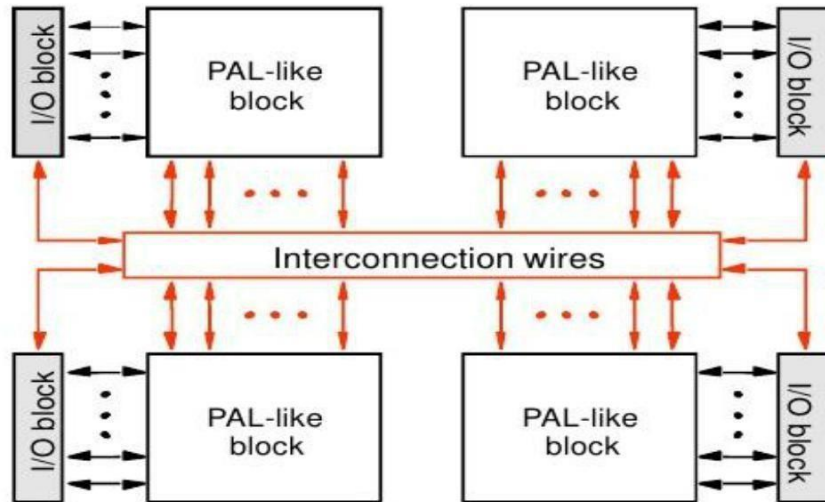
**Figure: FPGA Architecture**

- 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

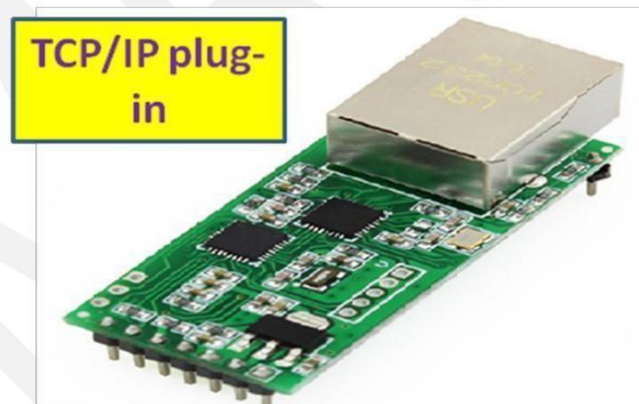
### Commercial off the Shelf Component (COTS):

- 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 extent.
- 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

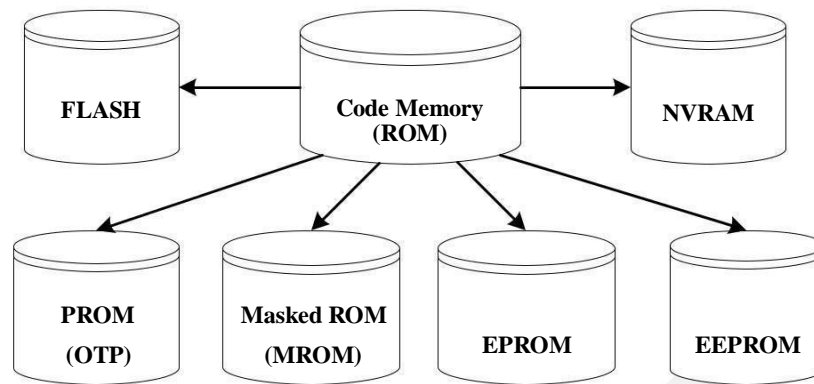
## Memory:

- 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).

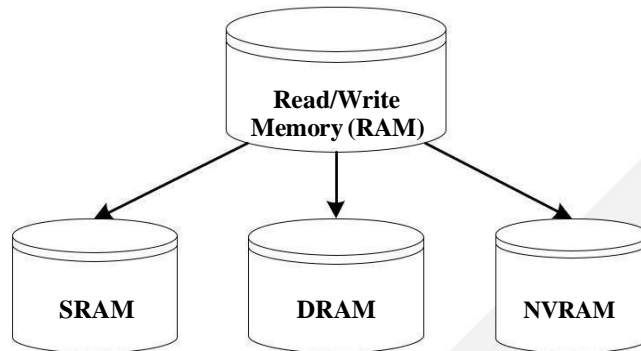
**5. Program Storage Memory – FLASH**

- FLASH memory is a variation of EEPROM technology.
- FLASH 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)**

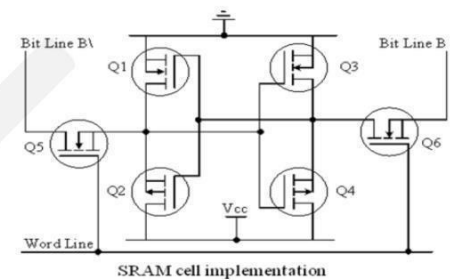
- ❖ RAM is the data memory or working memory of the controller/processor
- ❖ RAM is volatile, meaning when the power is turned off, all the contents are destroyed

- ❖ 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)



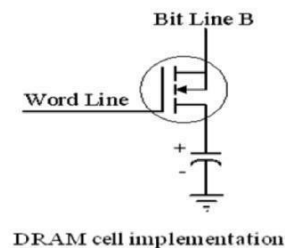
### 1. Static RAM (SRAM):

- ❖ 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).
- ❖ 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
- ❖ 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 that is 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.

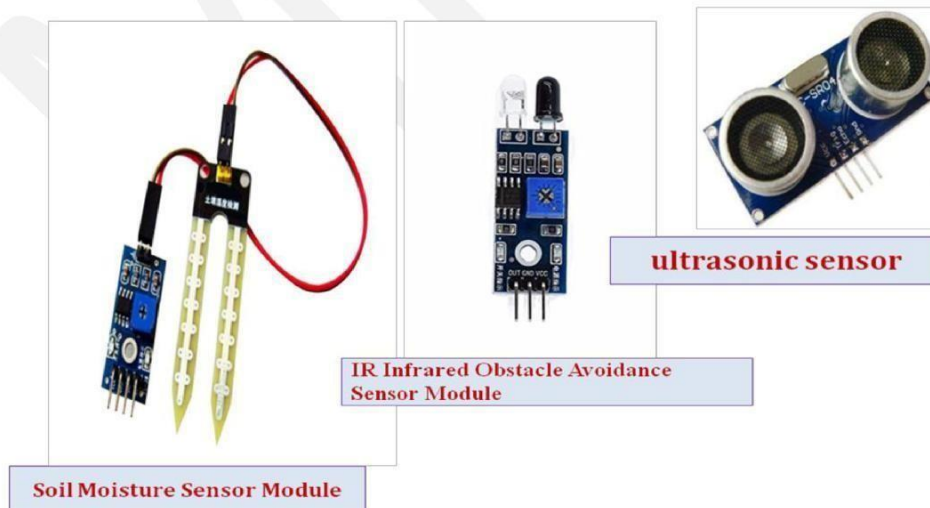
- Industrial grade memory chips are used in certain embedded devices may be operated at extreme environmental conditions like high temperature.

### 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.

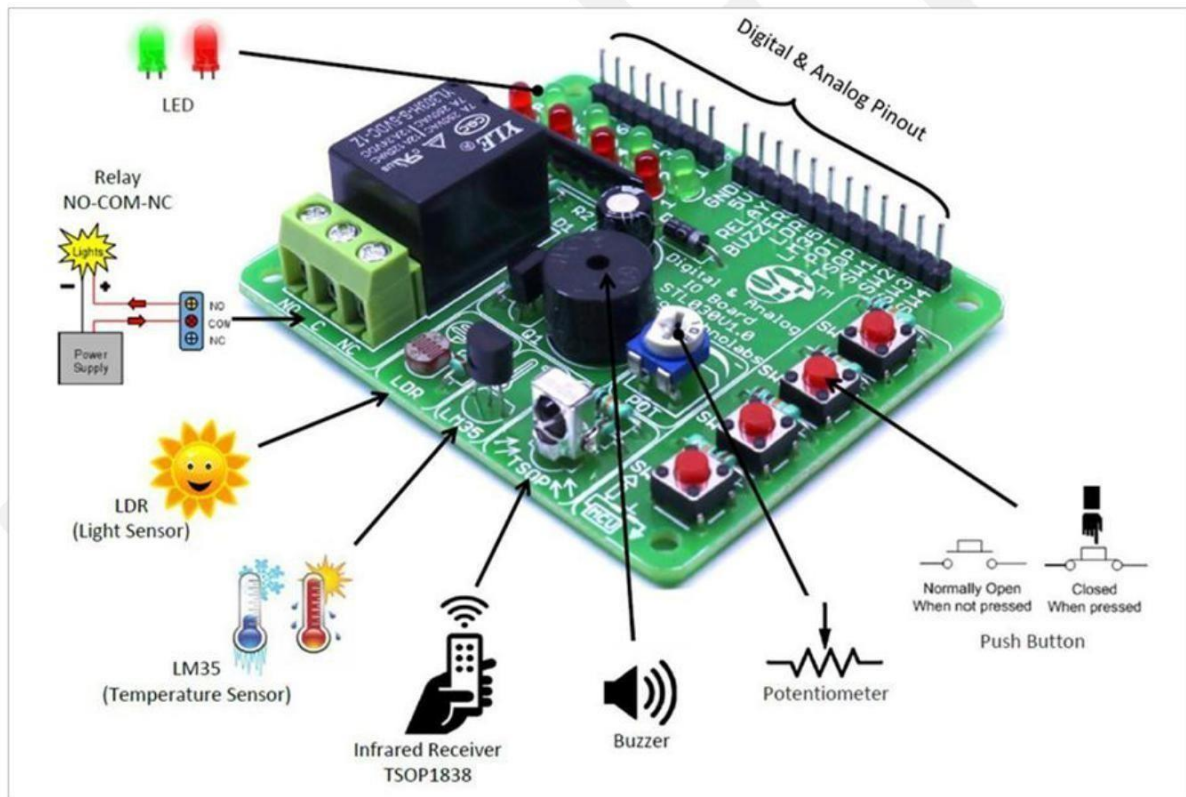
### Sensor:

- 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:**

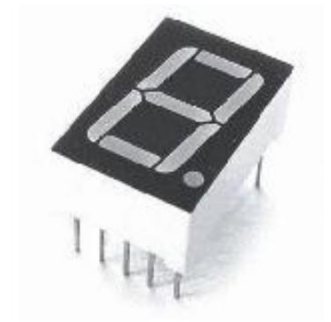
- A form of transducer device (mechanical or electrical) which converts signals to corresponding physical action (motion). Actuator acts as an output device
- Eg. Micro motor actuator which adjusts the position of the cushioning element in the Smart Running shoes from adidas



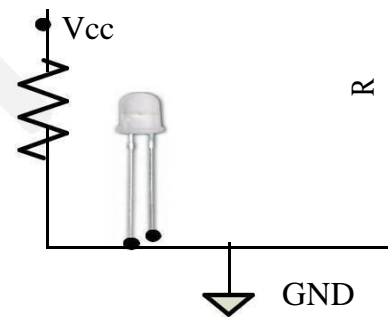
**Silicon TechnoLabs Digital Analog Arduino Starter kit**

**The I/O Subsystem:**

- ❑ The I/O subsystem of the embedded system facilitates the interaction of the embedded system with external world
- ❑ The interaction happens through the sensors and actuators connected to the Input and output ports respectively of the embedded system
- ❑ The sensors may not be directly interfaced to the Input ports, instead they may be interfaced through signal conditioning and translating systems like ADC, Optocouplers etc

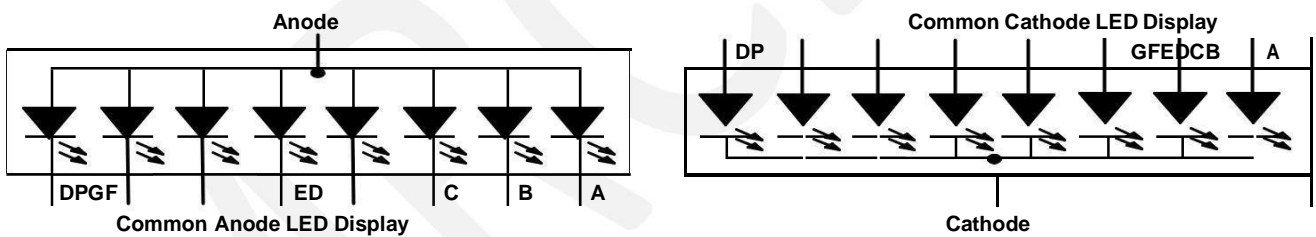
**1. I/O Devices - Light Emitting Diode (LED):**

- Light Emitting Diode (LED) is an output device for visual indication in any embedded system
- LED can be used as an indicator for the status of various signals or situations.
- Typical examples are indicating the presence of power conditions like „Device ON“, „Battery low“ or „Charging of battery“ for a battery operated handheld embedded devices
- LED is a p-n junction diode and it contains an anode and a cathode.
- For proper functioning of the LED, the anode of it should be connected to +ve terminal of the supply voltage and cathode to the –ve terminal of supply voltage
- The current flowing through the LED must limited to a value below the maximum current that it can conduct.
- A resistor is used in series between the power supply and the resistor to limit the current through the LED

**2. I/O Devices – 7-Segment LED Display**

- The 7 – segment LED display is an output device for displaying alpha numeric characters
- It contains 8 light-emitting diode (LED) segments arranged in a special form. Out of the 8 LED segments, 7 are used for displaying alpha numeric characters

- The LED segments are named A to G and the decimal point LED segment is named as DP
- The LED Segments A to G and DP should be lit accordingly to display numbers and characters
- The 7 – segment LED displays are available in two different configurations, namely; Common anode and Common cathode
- In the Common anode configuration, the anodes of the 8 segments are connected commonly whereas in the Common cathode configuration, the 8 LED segments share a common cathode line
- Based on the configuration of the 7 – segment LED unit, the LED segment anode or cathode is connected to the Port of the processor/controller in the order „A“ segment to the Least significant port Pin and DP segment to the most significant Port Pin.
- The current flow through each of the LED segments should be limited to the maximum value supported by the LED display unit

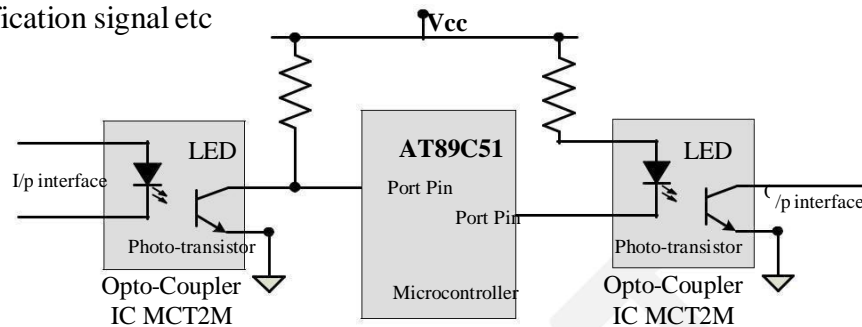


- The typical value for the current falls within the range of 20mA
- The current through each segment can be limited by connecting a current limiting resistor to the anode or cathode of each segment

### 3. I/O Devices – Optocoupler

- Optocoupler is a solid state device to isolate two parts of a circuit.
- Optocoupler combines an LED and a photo-transistor in a single housing (package)

- In electronic circuits, optocoupler is used for suppressing interference in data communication, circuit isolation, High voltage separation, simultaneous separation and intensification signal etc

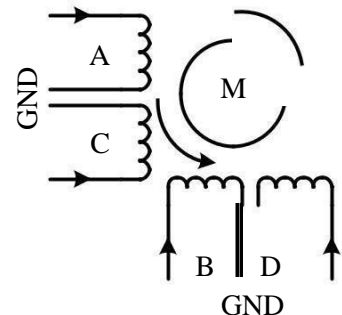


### Optocoupler in input and output circuit

- Optocouplers can be used in either input circuits or in output circuits

#### 4. I/O Devices – Stepper Motor:

- Stepper motor is an electro mechanical device which generates discrete displacement (motion) in response to dc electrical signals
- It differs from the normal dc motor in its operation. The dc motor produces continuous rotation on applying dc voltage whereas a stepper motor produces discrete rotation in response to the dc voltage applied to it
- Stepper motors are widely used in industrial embedded applications, consumer electronic products and robotics control systems
- The paper feed mechanism of a printer/fax makes use of stepper motors for its functioning.
- Based on the coil winding arrangements, a two phase stepper motor is classified into

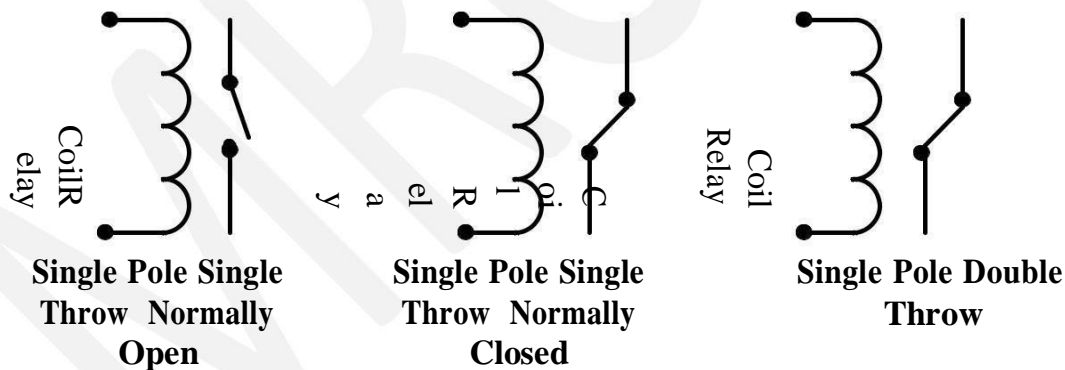


- ✓ Unipolar
- ✓ Bipolar

- ❖ **Unipolar:** A unipolar stepper motor contains two windings per phase. The direction of rotation (clockwise or anticlockwise) of a stepper motor is controlled by changing the direction of current flow. Current in one direction flows through one coil and in the opposite direction flows through the other coil. It is easy to shift the direction of rotation by just switching the terminals to which the coils are connected
- ❖ **Bipolar:** A bipolar stepper motor contains single winding per phase. For reversing the motor rotation the current flow through the windings is reversed dynamically. It requires complex circuitry for current flow reversal

### 5. The I/O Subsystem – I/O Devices – **Relay:**

- An electro mechanical device which acts as dynamic path selectors for signals and power.
- The „Relay“ unit contains a relay coil made up of insulated wire on a metal core and a metal armature with one or more contacts.
- „Relay“ works on electromagnetic principle.
- When a voltage is applied to the relay coil, current flows through the coil, which in turn generates a magnetic field.



- The magnetic field attracts the armature core and moves the contact point.
- The movement of the contact point changes the power/signal flow path.
- The Relay is normally controlled using a relay driver circuit connected to the port pin of the processor/controller
- A transistor can be used as the relay driver. The transistor can be selected depending on the relay driving current requirements.

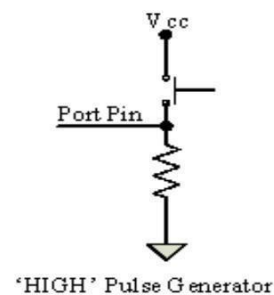
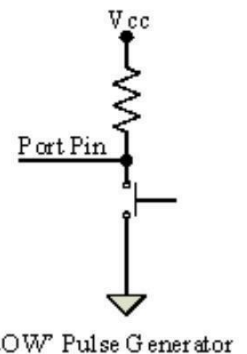
## 6. The I/O Subsystem – I/O Devices -Piezo Buzzer:

- It is a piezoelectric device for generating audio indications in embedded applications.
- A Piezo buzzer contains a piezoelectric diaphragm which produces audible sound in response to the voltage applied to it.
- Piezoelectric buzzers are available in two types
  1. Self-driving
  2. External driving
- Self-driving contains are the necessary components to generate sound at a predefined tone.
- External driving piezo Buzzers supports the generation of different tones.
- The tone can be varied by applying a variable pulse train to the piezoelectric buzzer.
- A Piezo Buzzer can be directly interfaced to the port pin of the processor/Controller.



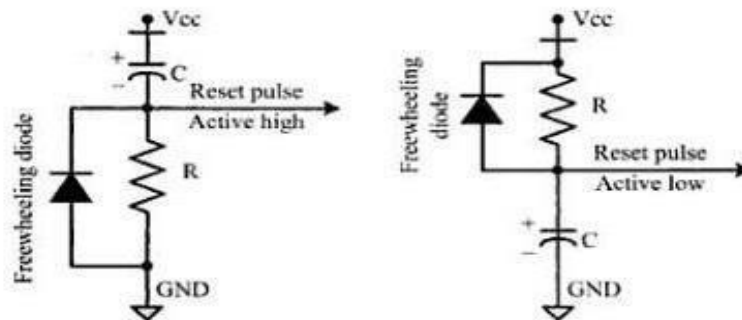
## 7. The I/O Subsystem – I/O Devices – Push button switch:

- Push Button switch is an input device.
- Push button switch comes in two configurations, namely „Push to Make“ and „Push to Break“
- The switch is normally in the open state and it makes a circuit contact when it is pushed or pressed in the „Push to Make“ configuration.
- In the „Push to Break“ configuration, the switch is normally in the closed state and it breaks the circuit contact when it is pushed or pressed
- The push button stays in the „closed“ (For Push to Make type) or „open“ (For Push to Break type) state as long as it is kept in the pushed state and it breaks/makes the circuit connection when it is released.
- Push button is used for generating a momentary pulse



The reset circuit is essential to ensure that the device is not operating at a voltage level where the device is not guaranteed to operate, during system power ON. The reset signal brings the internal registers and the different hardware systems of the processor/controller to a known state and starts the firmware execution from the reset vector (Normally from vector address 0x0000 for conventional processors/controllers. The reset vector can be relocated to an address for processors/controllers supporting bootloader). The reset signal can be either active high (The processor undergoes reset when the reset pin of the processor is at logic high) or active low (The processor undergoes reset when the reset pin of the processor is at logic low). Since the processor operation is synchronised to a clock signal, the reset pulse should be wide enough to give time for the clock oscillator to stabilise before the internal reset state starts. The reset signal to the processor can be applied at power ON through an external passive reset circuit comprising a Capacitor and Resistor or through a standard Reset IC like MAX810 from Maxim Dallas ([www.maxim-ic.com](http://www.maxim-ic.com)). Select the reset IC based on the type of reset signal and logic level (CMOS/TTL) supported by the processor/controller in use. Some microprocessors/controllers contain built-in internal

reset circuitry and they don't require external reset circuitry. Figure 2.35 illustrates a resistor capacitor based passive reset circuit for active high and low configurations. The reset pulse width can be adjusted by changing the resistance value  $R$  and capacitance value  $C$ .



### 2.6.2 Brown-out Protection Circuit

Brown-out protection circuit prevents the processor/controller from unexpected program execution behaviour when the supply voltage to the processor/controller falls below a specified voltage. It is essential for battery powered devices since there are greater chances for the battery voltage to drop below the required threshold. The processor behaviour may not be predictable if the supply voltage falls below the recommended operating voltage. A brown-out protection circuit holds the processor/controller in reset state, when the operating voltage falls below the threshold, until it rises above the threshold voltage. Certain processors/controllers support built in brown-out protection circuit which monitors the supply voltage internally. If the processor/controller doesn't integrate a built-in brown-out protection circuit, the same can be implemented using external passive circuits or supervisor ICs. Figure 2.36 illustrates a brown-out protection circuit implementation using Zener diode and transistor for processor/controller with active low Reset logic.

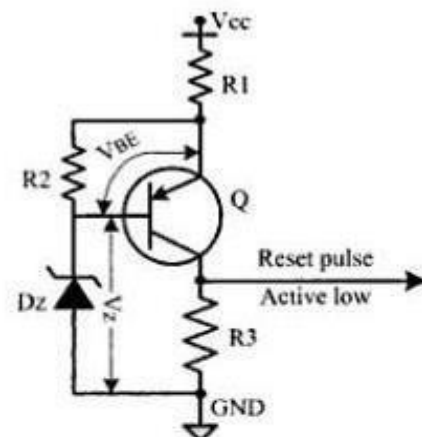


Fig. 2.36 Brown-out protection circuit with Active low output

The Zener diode  $D_z$  and transistor  $Q$  forms the heart of this circuit. The transistor conducts always when the supply voltage  $V_{cc}$  is greater than that of the sum of  $V_{BE}$  and  $V_z$  (Zener voltage). The transistor stops conducting when the supply voltage falls below the sum of  $V_{BE}$  and  $V_z$ . Select the Zener diode with required voltage for setting the low threshold value for  $V_{cc}$ . The values of  $R_1$ ,  $R_2$ , and  $R_3$  can be selected based on the electrical characteristics (Absolute maximum current and voltage ratings) of the transistor in use. Microprocessor Supervisor ICs like DS1232 from Maxim Dallas ([www.maxim-ic.com](http://www.maxim-ic.com)) also provides Brown-out protection.

### 2.6.3 Oscillator Unit

A microprocessor/microcontroller is a digital device made up of digital combinational and sequential circuits. The instruction execution of a microprocessor/controller occurs in sync with a clock signal. It is analogous to the heartbeat of a living being which synchronises the execution of life. For a living being, the heart is responsible for the generation of the beat whereas the oscillator unit of the embedded system is responsible for generating the precise clock for the processor. Certain processors/controllers integrate a built-in oscillator unit and simply require an external ceramic resonator/quartz crystal for producing the necessary clock signals. Quartz crystals and ceramic resonators are equivalent in operation, however they possess physical difference. A quartz crystal is normally mounted in a hermetically sealed metal case with two leads protruding out of the case. Certain devices may not contain a built-in oscillator unit and require the clock pulses to be generated and supplied externally. Quartz crystal Oscillators are available in the form chips and they can be used for generating the clock pulses in such a cases. The speed of operation of a processor is primarily dependent on the clock frequency. However we cannot increase the clock frequency blindly for increasing the speed of execution. The logical circuits lying inside the processor always have an upper threshold value for the maximum clock at which the system can run, beyond which the system becomes unstable and non functional. The total system power consumption is directly proportional to the clock frequency. The power consumption increases with increase in clock frequency. The accuracy of program execution depends on the accuracy of the clock signal. The accuracy of the crystal oscillator or ceramic resonator is normally expressed in terms of  $\pm$ -ppm (Parts per million). Figure 2.37 illustrates the usage of quartz crystal/ceramic resonator and external oscillator chip for clock generation.

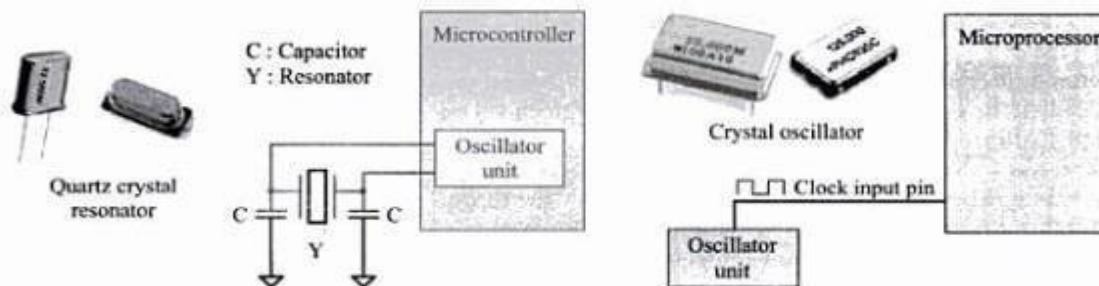


Fig. 2.37 Oscillator circuitry using quartz crystal and quartz crystal oscillator

### 2.6.4 Real-Time Clock (RTC)

Real-Time Clock (RTC) is a system component responsible for keeping track of time. RTC holds information like current time (In hours, minutes and seconds) in 12 hour/24 hour format, date, month, year, day of the week, etc. and supplies timing reference to the system. RTC is intended to function even in the absence of power. RTCs are available in the form of Integrated Circuits from different semiconductor manufacturers like Maxim/Dallas, ST Microelectronics etc. The RTC chip contains a microchip for holding the time and date related information and backup battery cell for functioning in the absence of power, in a single IC package. The RTC chip is interfaced to the processor or controller of the embedded system. For Operating System based embedded devices, a timing reference is essential for synchronising

the operations of the OS kernel. The RTC can interrupt the OS kernel by asserting the interrupt line of the processor/controller to which the RTC interrupt line is connected. The OS kernel identifies the interrupt in terms of the Interrupt Request (IRQ) number generated by an interrupt controller. One IRQ can be assigned to the RTC interrupt and the kernel can perform necessary operations like system date time updation, managing software timers etc when an RTC timer tick interrupt occurs. The RTC can be configured to interrupt the processor at predefined intervals or to interrupt the processor when the RTC register reaches a specified value (used as alarm interrupt).

### 2.6.5 Watchdog Timer

In desktop Windows systems, if we feel our application is behaving in an abnormal way or if the system hangs up, we have the 'Ctrl + Alt + Del' to come out of the situation. What if it happens to our embedded system? Do we really have a 'Ctrl + Alt + Del' to take control of the situation? Of course not ☹, but we have a watchdog to monitor the firmware execution and reset the system processor/microcontroller when the program execution hangs up. A watchdog timer, or simply a watchdog, is a hardware timer for monitoring the firmware execution. Depending on the internal implementation, the watchdog timer increments or decrements a free running counter with each clock pulse and generates a reset signal to reset the processor if the count reaches zero for a down counting watchdog, or the highest count value for an upcounting watchdog. If the watchdog counter is in the enabled state, the firmware can write a zero (for upcounting watchdog implementation) to it before starting the execution of a piece of code (subroutine or portion of code which is susceptible to execution hang up) and the watchdog will start counting. If the firmware execution doesn't complete due to malfunctioning, within the time required by the watchdog to reach the maximum count, the counter will generate a reset pulse and this will reset the processor (if it is connected to the reset line of the processor). If the firmware execution completes before the expiration of the watchdog timer you can reset the count by writing a 0 (for an upcounting watchdog timer) to the watchdog timer register. Most of the processors implement watchdog as a built-in component and provides status register to control the watchdog timer (like enabling and disabling watchdog functioning) and watchdog timer register for writing the count value. If the processor/controller doesn't contain a built in watchdog timer, the same can be implemented using an external watchdog timer IC circuit. The external watchdog timer uses hardware logic for enabling/disabling, resetting the watchdog count, etc instead of the firmware based 'writing' to the status and watchdog timer register. The Microprocessor supervisor IC DS1232 integrates a hardware watchdog timer in it. In modern systems running on embedded operating systems, the watchdog can be implemented in such a way that when a watchdog timeout occurs, an interrupt is generated instead of resetting the processor. The interrupt handler for this handles the situation in an appropriate fashion. Figure 2.38 illustrates the implementa-

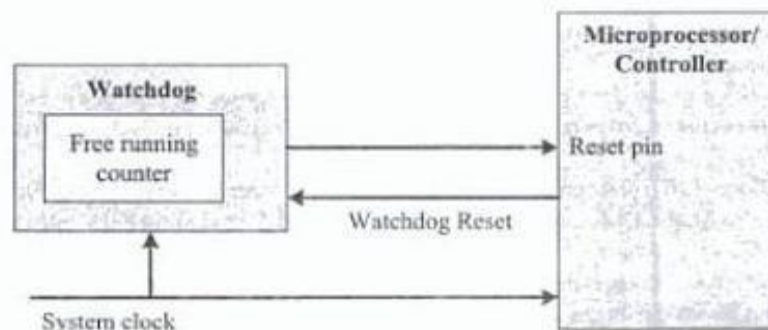


Fig. 2.38 Watchdog timer for firmware execution supervision

Text Book:-

1. Introduction to Embedded Systems – Shibu K.V Mc Graw Hill

# Embedded Systems



## UNIT-3

### EMBEDDED FIRMWARE DESIGN & DEVELOPMENT



**MALLA REDDY COLLEGE OF  
ENGINEERING & TECHNOLOGY**

Permanently Affiliated to JNTUH and Approved by AICTE, New Delhi

---

## Embedded Firmware

### 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

### **1. 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
- **Mix of Assembly & High level Language**
  - Mixing High Level Language (Like C) with Assembly Code
  - Mixing Assembly code with High Level Language (Like C)
  - Inline Assembly

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

- ‘*Assembly Language*’ is the human readable notation of ‘*machine language*’
- ‘*Machine 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

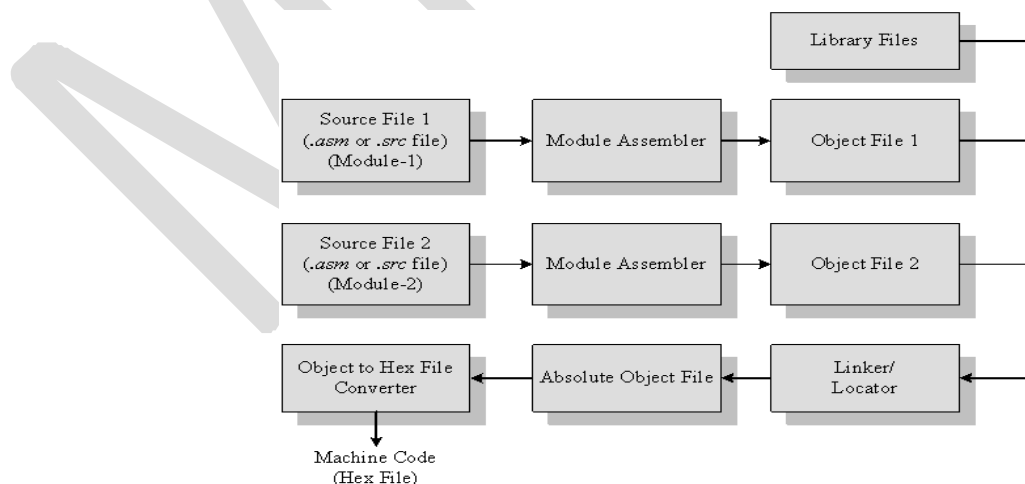
          DJNZ R1, DELAY; Decrement R1 and loop till    R1= 0

          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/locator 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 but also improves the total system performance.
- Through effective assembly coding, optimum performance can be 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:**

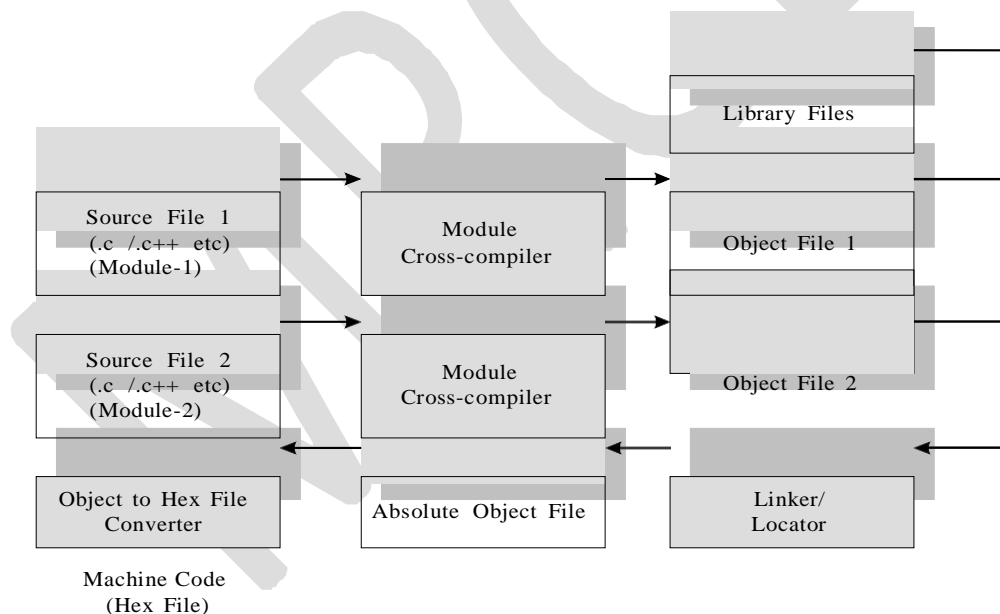
- Target applications written in assembly instructions are valid only for that particular family of processors and cannot be re-used for another target processors/controllers.
- If the target processor/controller changes, a complete re-writing of the application using assembly language for new target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

**Embedded firmware Development Languages/Options – Mixing of Assembly Language with High Level Language**

- Embedded firmware development may require the mixing of Assembly Language with high level language or vice versa.
- Interrupt handling, Source code is already available in high level language\Assembly Language etc are examples

- High Level language and low level language can be mixed in three different ways
  - ✓ Mixing Assembly Language with High level language like 'C'
  - ✓ Mixing High level language like 'C' with Assembly Language
  - ✓ In line Assembly
- The passing of parameters and return values between the high level and low level language is cross-compiler specific

### **1. Mixing Assembly Language with High level language like 'C' (Assembly Language with 'C'):**

- Assembly routines are mixed with 'C' in situations where the entire program is written in 'C' and the cross compiler in use do not have built in support for implementing certain features like ISR.
- If the programmer wants to take advantage of the speed and optimized code offered by the machine code generated by hand written assembly rather than cross compiler generated machine code.
- For accessing certain low level hardware ,the timing specifications may be very critical and cross compiler generated machine code may not be able to offer the required time specifications accurately.
- Writing the hardware/peripheral access routine in processor/controller specific assembly language and invoking it from 'C' is the most advised method.
- Mixing 'C' and assembly is little complicated.
- The programmer must be aware of how to pass parameters from the 'C' routine to assembly and values returned from assembly routine to 'C' and how Assembly routine is invoked from the 'C' code.

- Passing parameter to the assembly routine and returning values from the assembly routine to the caller 'C' function and the method of invoking the assembly routine from 'C' code is cross compiler dependent.
- There is no universal written rule for purpose.
- We can get this information from documentation of the cross compiler.
- Different cross compilers implement these features in different ways depending on GPRs and memory supported by target processor/controller

## **2. Mixing High level language like 'C' with Assembly Language ('C' with Assembly Language)**

- The source code is already available in assembly language and routine written in a high level language needs to be included to the existing code.
- The entire source code is planned in Assembly code for various reasons like optimized code, optimal performance, efficient code memory utilization and proven expertise in handling the assembly.
- The functions written in 'C' use parameter passing to the function and returns values to the calling functions.
- The programmer must be aware of how parameters are passed to the function and how values returned from the function and how function is invoked from the assembly language environment.
- Passing parameter to the function and returning values from the function using CPU registers , stack memory and fixed memory.
- Its implementation is cross compiler dependent and varies across compilers.

### 3.In line Assembly:

- Inline assembly is another technique for inserting the target processor/controller specific assembly instructions at any location of source code written in high level language 'C'
- Inline Assembly avoids the delay in calling an assembly routine from a 'C' code.
- Special keywords are used to indicate the start and end of Assembly instructions
- *E.g #pragma asm*

*Mov A,#13H*

*#pragma endasm*

- Keil C51 uses the keywords *#pragma asm* and *#pragma endasm* to indicate a block of code written in assembly.

### Text Books:

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

# EMBEDDED SYSTEM DESIGN

## UNIT-IV

### RTOS Based Embedded System Design



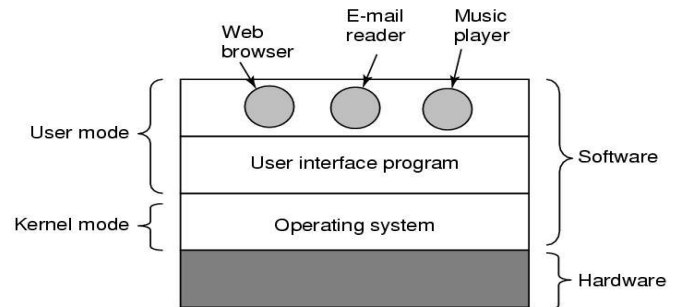
**MALLA REDDY COLLEGE OF  
ENGINEERING & TECHNOLOGY**

Permanently Affiliated to JNTUH and Approved by AICTE, New Delhi

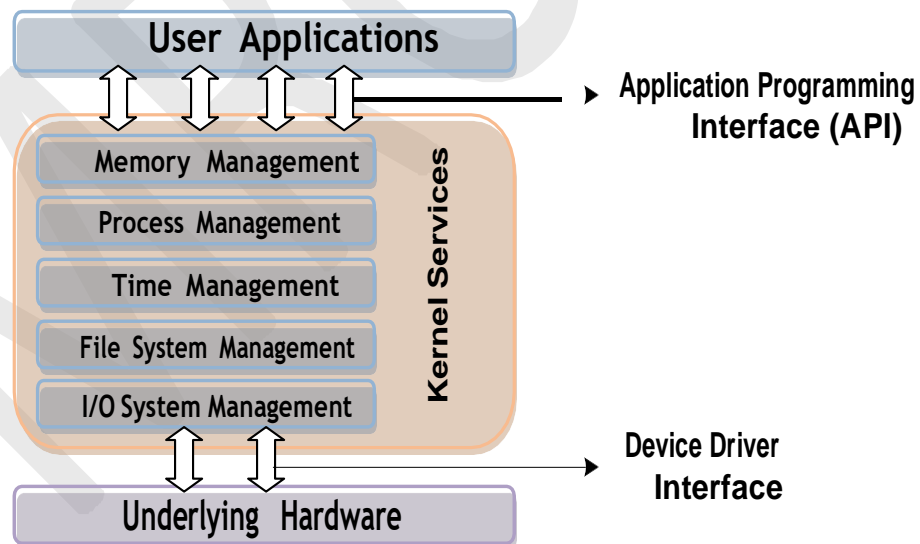
## Operating System Basics:

- 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 and correctly



**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 services like
  - Process Management
  - Primary Memory Management
  - File System management
  - I/O System (Device) Management
  - Secondary Storage Management
  - Protection
  - Time management
  - Interrupt Handling

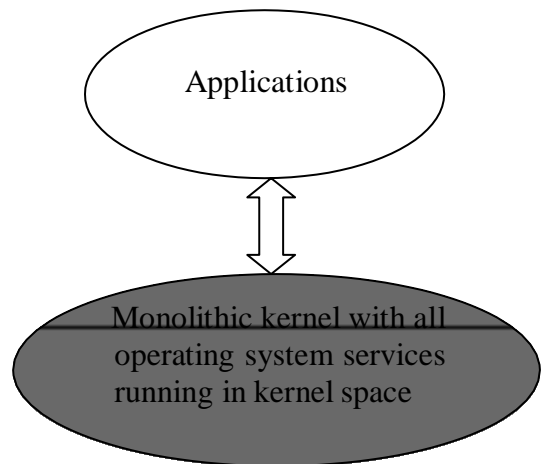
### Kernel Space and User Space:

- 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 user programs/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:**

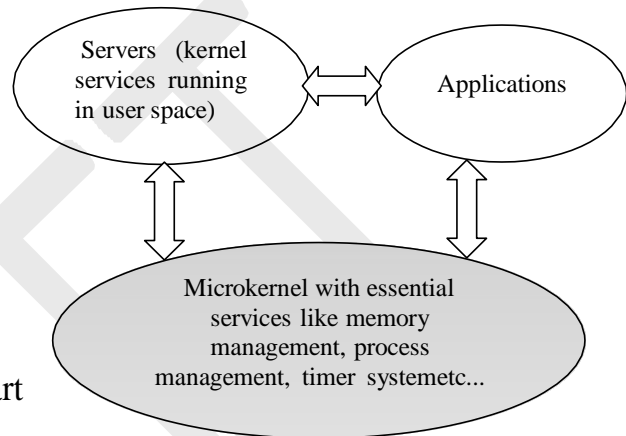
- All kernel services run in the kernel space
- All kernel modules run within the same memory space under a single kernel thread
- The tight internal integration of kernel modules in monolithic kernel architecture allows the effective utilization of the low-level features of the underlying system
- The 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



**Figure 2: The Monolithic Kernel Model**

## 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 programs known 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



**Figure 3: The Microkernel Model**

- 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 services to execute generic applications
- Need not be deterministic in execution behavior
- May inject random delays into application software and thus cause slow responsiveness of an application at unexpected times
- Usually deployed in computing systems where deterministic behavior is not an important criterion
- Personal Computer/Desktop system is a typical example for a system where 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 a task which is ready to execute)
    - ❖ *Task Type:* Task type. Indicates what is the type for this task. The task can 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 memory and 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, next and 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 uninitialized 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

$$2^{32} * 10^{-6} / (24 * 60 * 60) = 49700 \text{ Days} \approx 0.0497 \text{ Days} = 1.19 \text{ Hours}$$

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

$$2^{32} * 10^{-3} / (24 * 60 * 60) = 497 \text{ Days} = 49.7 \text{ Days} \approx 50 \text{ Days}$$

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 executing task)
- 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 = '*count down*')
- 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 loss and irrecoverable damages to the system/users
- ❖ Emphasize on the principle '*A late answer is a wrong answer*'
- ❖ 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 unexpected delays in the task execution. Most of the Hard Real Time Systems are automatic and does not contain a 'human in the loop'

### ● **Soft Real-time System:**

- ❖ Real Time Operating Systems that does not guarantee meeting deadlines, but, offer the best effort to meet the deadline
- ❖ Missing deadlines for tasks are acceptable if the frequency of deadline 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 bit faster*'
- ❖ 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.

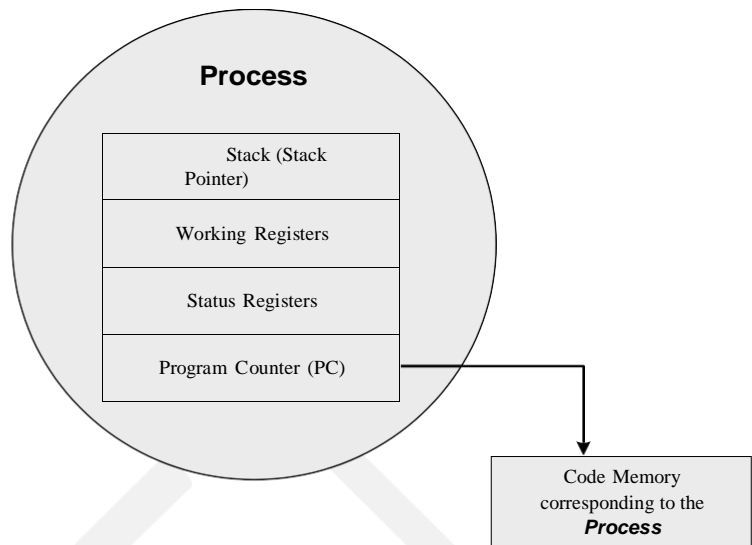
### □ **Tasks, Processes & Threads :**

- 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 system context
- 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 used interchangeably
- 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

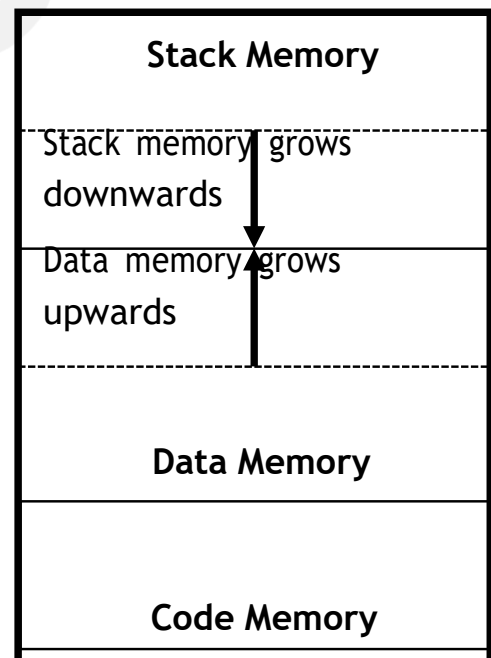


**Figure: 4 Structure of a Process**

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

### Memory organization of Processes:

- The memory occupied by the *process* is segregated into three regions namely; Stack memory, Data memory and Code memory
- The 'Stack' memory holds all temporary data such as variables local to the process
- Data memory holds all global data for the process
- The code memory contains the program code (instructions) corresponding to the process



**Fig: 5 Memory organization of a Process**

- On loading a process into the main memory, a specific area of memory is allocated for the process
- The stack memory usually starts at the highest memory address from the memory area allocated for the process (Depending on the OS kernel implementation)

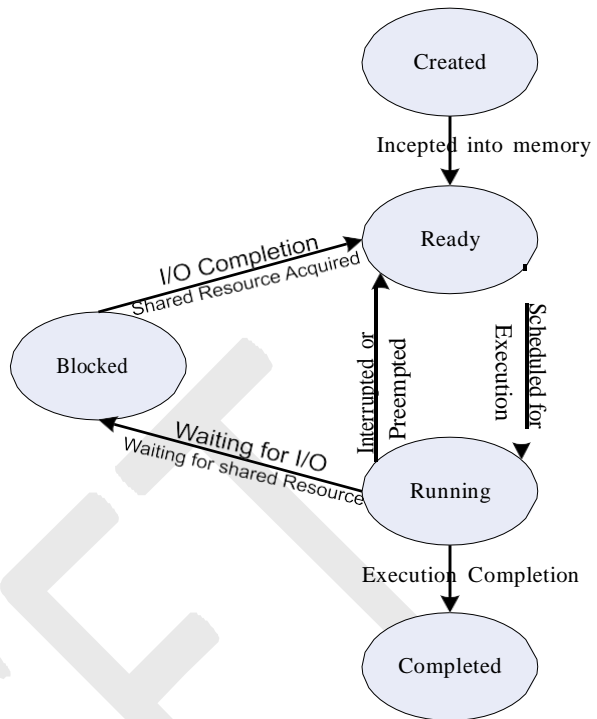
#### □ **Process States & State Transition**

- The creation of a process to its termination is not a single step operation
- 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 do next

#### □ **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



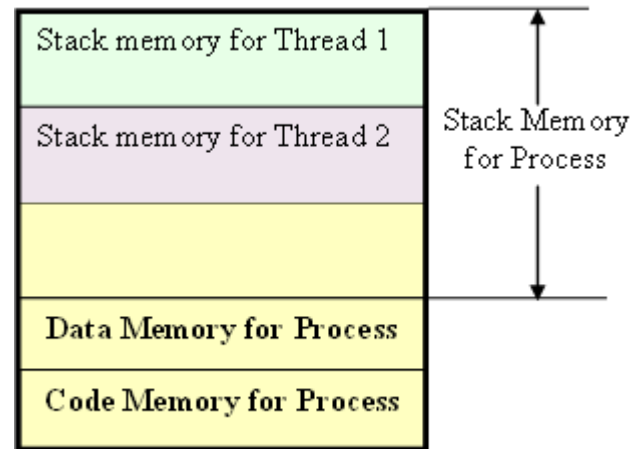
**Figure 6. Process states and State transition**

- **Completed State:** A state where the process completes its execution
- ★ The transition of a process from one state to another is known as '*Statetransition*'
- ★ When a process changes its state from Ready to running or from running to blocked or terminated or from blocked to running, the CPU allocation for the process may also change

## □ 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

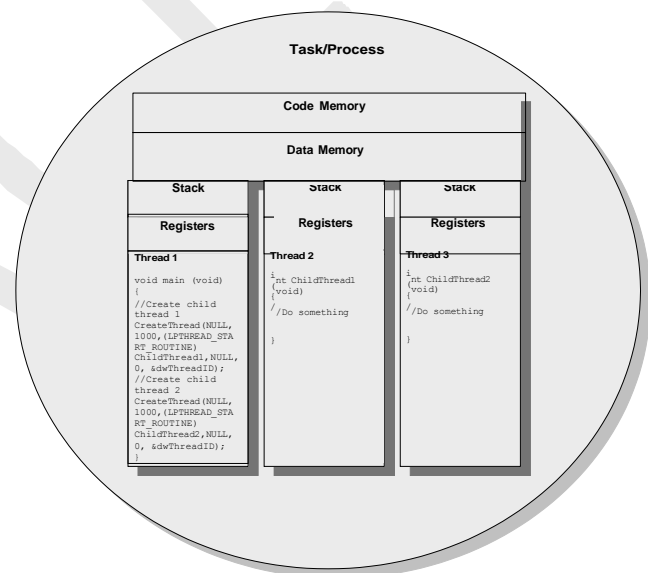


**Figure 7 Memory organization of process and its associated Threads**

### □ 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



**Figure 8 Process with multi-threads**

- threads of the process that do not require the event, which the other thread is waiting, for processing. This speeds up the execution of the process.
- 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 the threads.
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.

### **Multiprocessing & Multitasking**

- The ability to execute multiple processes simultaneously is referred as *multiprocessing*
- Systems which are capable of performing multiprocessing are known as *multiprocessor* systems
  - *Multiprocessor* systems possess multiple CPUs and can execute multiple processes simultaneously
  - The ability of the Operating System to have multiple programs in memory, which are ready for execution, is referred as *multiprogramming*
  - *Multitasking* refers to the ability of an operating system to hold multiple processes in memory and switch the processor (CPU) from executing one process to another process
  - *Multitasking* involves ‘*Context switching*’, ‘*Context saving*’ and ‘*Context retrieval*’
  - *Context switching* refers to the switching of execution context from task to other
  - When a task/process switching happens, the current context of execution should be saved to (*Context saving*) retrieve it at a later point of time when the CPU executes the process, which is interrupted currently due to execution switching
  - During context switching, the context of the task to be executed is retrieved from the saved context list. This is known as *Context retrieval*

## Multitasking – Context Switching:

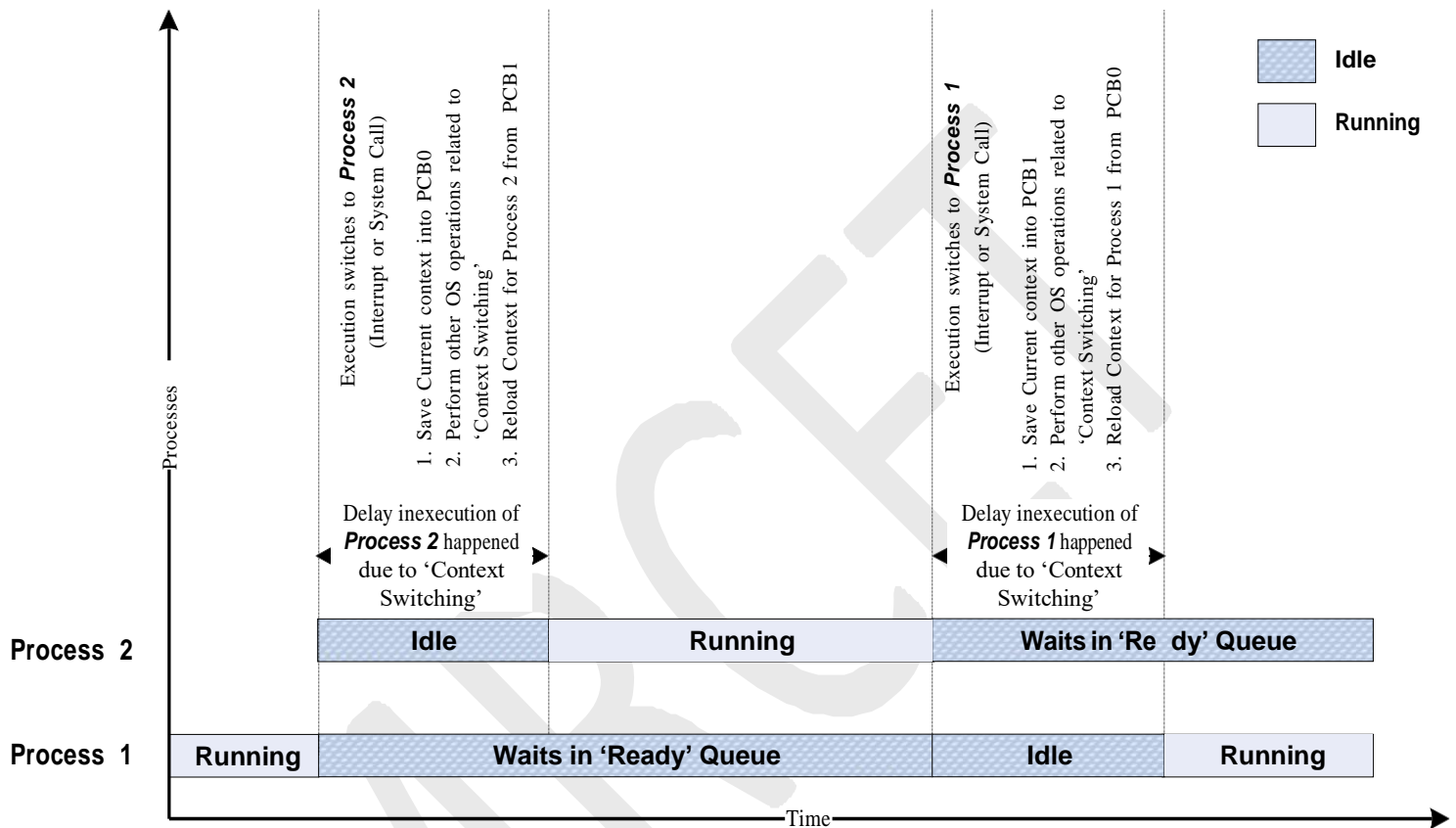


Figure 9 Context Switching

- **Multiprogramming:** The ability of the Operating System to have multiple programs in memory, which are ready for execution, is referred as multiprogramming.

## Types of Multitasking :

Depending on how the task/process execution switching act is implemented, multitasking can be 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/process priority
- **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' state, 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 for running.
- **Device Queue:** Contains the set of processes, which are waiting for an I/O device

### Task Scheduling – Task transition through various Queues

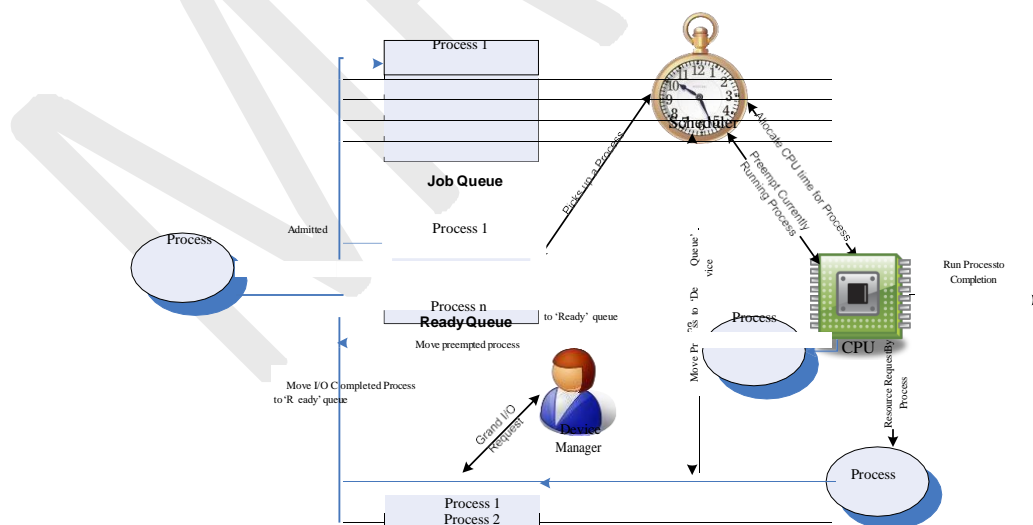


Figure 10. Process Transition through various queues

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

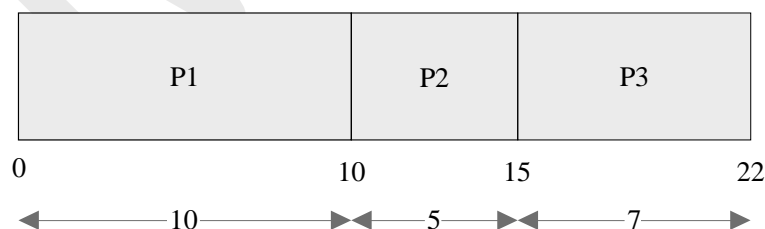
- Allocates CPU time to the processes based on the order in which they enter the 'Ready' queue
- The first entered process is serviced first
- It is same as any real world application where queue systems are used; E.g. Ticketing

#### 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 enter 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

$$= (\text{Waiting time for (P1+P2+P3)}) / 3$$

$$= (0+10+15)/3 = 25/3 = 8.33 \text{ milliseconds}$$

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

$$= (\text{Turn Around Time for (P1+P2+P3)}) / 3$$

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

$$= 15.66 \text{ milliseconds}$$

### **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 they are entered in the 'Ready' queue
- The last entered process is serviced first

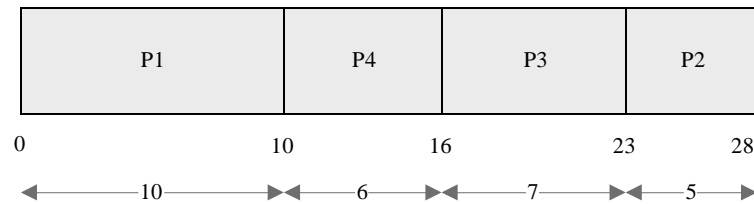
- LCFS scheduling is also known as Last In First Out (LIFO) where the process, which is put last into the '*Ready*' queue, is serviced first

### **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, 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

$$= (\text{Waiting time for (P1+P4+P3+P2)}) / 4$$

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

$$= 11 \text{ milliseconds}$$

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

$$= (\text{Turn Around Time for (P1+P4+P3+P2)}) / 4$$

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

$$= 18 \text{ milliseconds}$$

### **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.

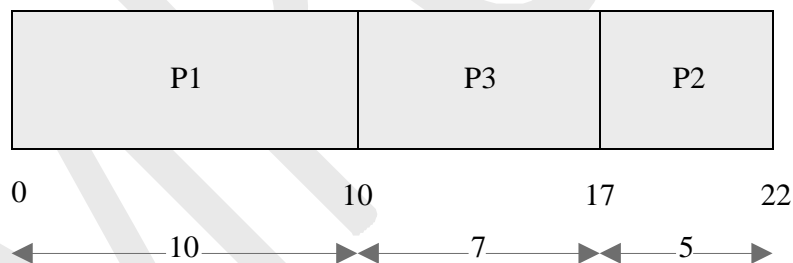
### **Non-preemptive scheduling – Priority based Scheduling**

- 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

$$= (\text{Waiting time for (P1+P3+P2)}) / 3$$

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

$$= 9 \text{ milliseconds}$$

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

$$= (\text{Turn Around Time for (P1+P3+P2)}) / 3$$

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

$$= 16.33 \text{ milliseconds}$$

### 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 for execution
- 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

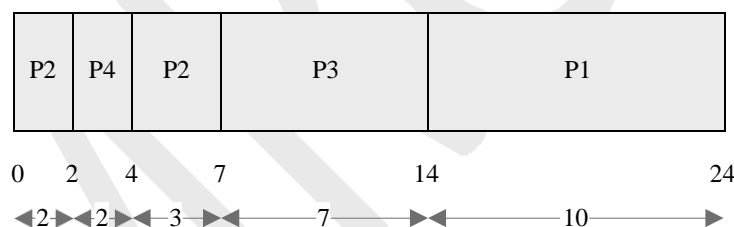
## Preemptive scheduling – Preemptive SJF Scheduling/ Shortest Remaining Time (SRT):

- 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 for execution.

**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 following order



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

$$= (\text{Waiting time for (P4+P2+P3+P1)}) / 4$$

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

$$= 5.75 \text{ milliseconds}$$

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

$$= (\text{Turn Around Time for (P2+P4+P3+P1)}) / 4$$

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

$$= 11.75 \text{ milliseconds}$$

### 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

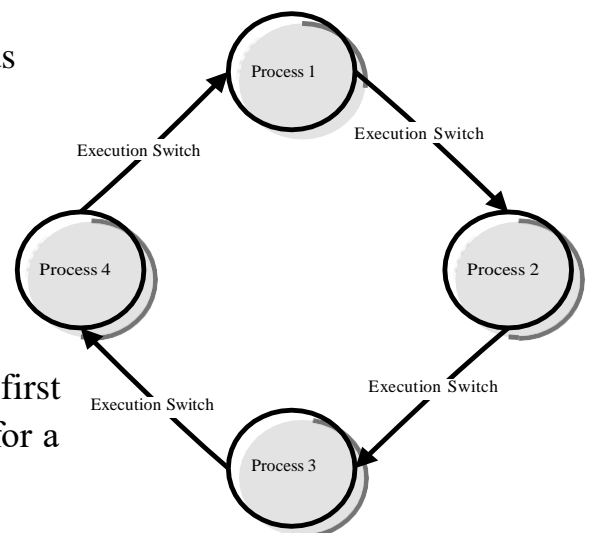
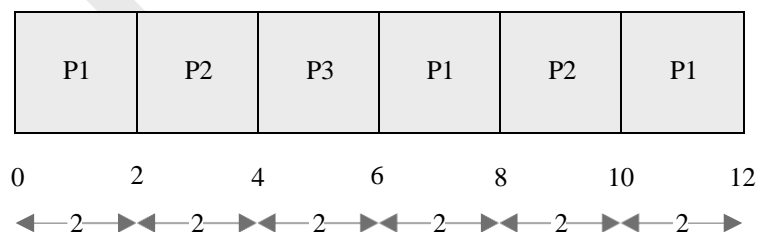


Figure 11 Round Robin Scheduling

- 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 =  $0 + (6-2) + (10-8) = 0+4+2 = 6\text{ms}$  (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 =  $(2-0) + (8-4) = 2+4 = 6\text{ms}$  (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 =  $(4-0) = 4\text{ms}$  (P3 starts executing after completing the first time slices for P1 and P2 and completes its execution in a single time slice.)

$$\begin{aligned}\text{Average waiting time} &= (\text{Waiting time for all the processes}) / \text{No. of Processes} \\ &= (\text{Waiting time for (P1+P2+P3)}) / 3 \\ &= (6+6+4)/3 = 16/3 \\ &= 5.33 \text{ milliseconds}\end{aligned}$$

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-)

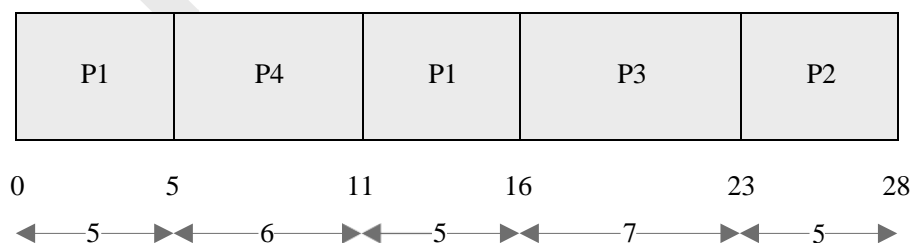
$$\begin{aligned}\text{Average Turn Around Time} &= (\text{Turn Around Time for all the processes}) / \text{No. of Processes} \\ &= (\text{Turn Around Time for (P1+P2+P3)}) / 3 \\ &= (12+10+6)/3 = 28/3 \\ &= 9.33 \text{ milliseconds.}\end{aligned}$$

### 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 the CPU
- 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

$$= (\text{Waiting time for (P1+P4+P3+P2)}) / 4$$

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

$$= 11.25 \text{ milliseconds}$$

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

$$= (\text{Turn Around Time for (P2+P4+P3+P1)}) / 4$$

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

$$= 18.25 \text{ milliseconds}$$

## How to chose 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 decision on 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.
- ❑ **7.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.

### **8. 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 available OS.
- 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:**

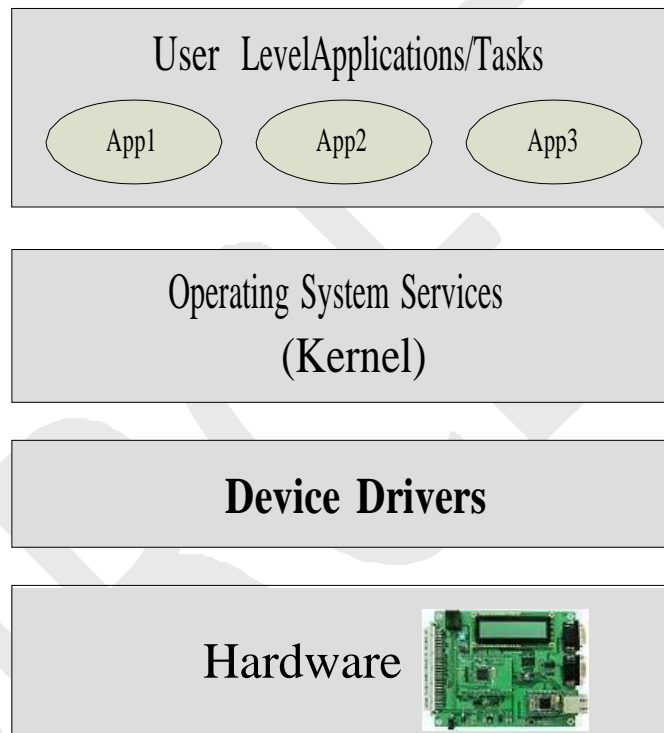
- How easy it is to use a commercial RTOS is another important feature that needs to be considered in the RTOS selection.

### **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.

## Device Drivers:

- Device driver is a piece of software that acts as a bridge between the operating system and the hardware
- The user applications talk to the OS kernel for all necessary information exchange including communication with the hardware peripherals



- The architecture of the OS kernel will not allow direct device access from the user application
- All the device related access should flow through the OS kernel and the OS kernel routes it to the concerned hardware peripheral
- OS Provides interfaces in the form of Application Programming Interfaces (APIs) for accessing the hardware
- The device driver abstracts the hardware from user applications

- Device drivers are responsible for initiating and managing the communication with the hardware peripherals
- Drivers which comes as part of the Operating system image is known as 'built-in drivers' or 'onboard' drivers. Eg. NAND FLASH driver
- Drivers which needs to be installed on the fly for communicating with add-on devices are known as 'Installable drivers'
- For installable drivers, the driver is loaded on a need basis when the device is present and it is unloaded when the device is removed/detached
- The 'Device Manager' service of the OS kernel is responsible for loading and unloading the driver, managing the driver etc.
- The underlying implementation of device driver is OS kernel dependent
- The driver communicates with the kernel is dependent on the OS structure and implementation.
- Device drivers can run on either user space or kernel space
- Device drivers which run in user space are known as *user mode drivers* and the drivers which run in kernel space are known as *kernel mode drivers*
- User mode drivers are safer than kernel mode drivers
- If an error or exception occurs in a user mode driver, it won't affect the services of the kernel
- If an exception occurs in the kernel mode driver, it may lead to the kernel crash
- The way how a device driver is written and how the interrupts are handled in it are Operating system and target hardware specific.
- The device driver implements the following:
  - Device (Hardware) Initialization and Interrupt configuration

- Interrupt handling and processing
- Client interfacing (Interfacing with user applications)
- The basic Interrupt configuration involves the following.
- Set the interrupt type (Edge Triggered (Rising/Falling) or Level Triggered (Low or High)), enable the interrupts and set the interrupt priorities.
- The processor identifies an interrupt through IRQ.
- IRQs are generated by the Interrupt Controller.
- Register an Interrupt Service Routine (ISR) with an Interrupt Request (IRQ).
- When an interrupt occurs, depending on its priority, it is serviced and the corresponding ISR is invoked
- The processing part of an interrupt is handled in an ISR
- The whole interrupt processing can be done by the ISR itself or by invoking an Interrupt Service Thread (IST)
- The IST performs interrupt processing on behalf of the ISR
- It is always advised to use an IST for interrupt processing, to make the ISR compact and short

**Reference Books:**

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

## EMBEDDED SYSTEM DESIGN

IV YEAR ECE (R18)

### UNIT-V

## COMMUNICATION INTERFACE



**Communication Interface:**

- Communication interface is essential for communicating with various subsystems of the embedded system and with the external world
- The communication interface can be viewed in two different perspectives; namely;

1. Device/board level communication interface (Onboard Communication Interface)

2. Product level communication interface (External Communication Interface)

**1. Device/board level communication interface (Onboard Communication Interface):**

---

The communication channel which interconnects the various components within an embedded product is referred as Device/board level communication interface (Onboard Communication Interface)

- Examples: Serial interfaces like I2C, SPI, UART, 1-Wire etc and Parallel bus interface

**2. Product level communication interface (External Communication Interface):**

The „Product level communication interface“ (External Communication Interface) is responsible for data transfer between the embedded system and other devices or modules. The external communication interface can be either wired media or wireless media and it can be a serial or parallel interface.

- Examples for wireless communication interface: Infrared (IR), Bluetooth (BT), Wireless LAN (Wi-Fi), Radio Frequency waves (RF), GPRS etc.
- Examples for wired interfaces: RS-232C/RS-422/RS 485, USB, Ethernet (TCP-IP), IEEE 1394 port, Parallel port etc.



## 1. Device/board level or On board communication interfaces: The

Communication channel which interconnects the various components within an embedded product is referred as Device/board level communication interface (Onboard Communication Interface)

These are classified into

- I2C (Inter Integrated Circuit) Bus

- SPI (Serial Peripheral Interface) Bus

- UART (Universal Asynchronous Receiver Transmitter)

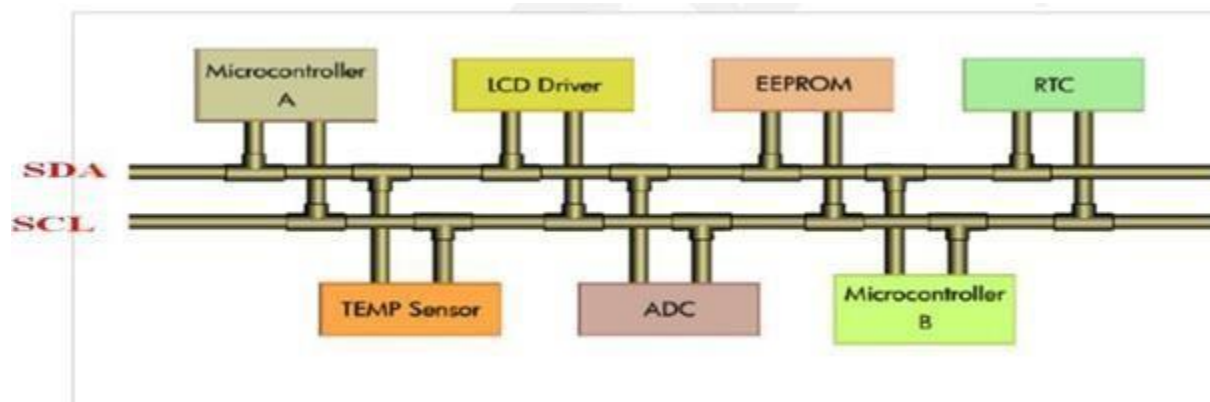
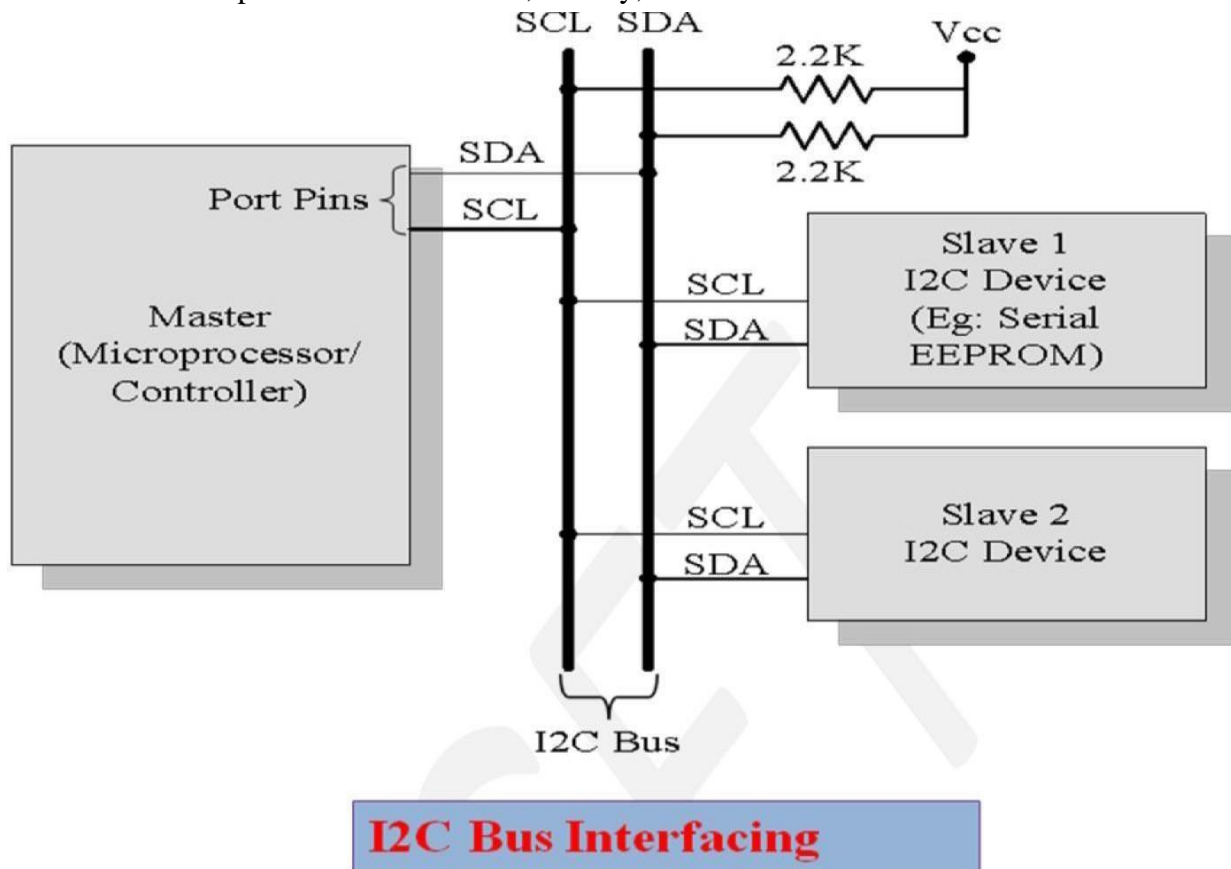
- 1-Wires Interface

- Parallel Interface

### 1 I2C (Inter Integrated Circuit) Bus:

Inter Integrated Circuit Bus (I2C - Pronounced „I square C“) is a synchronous bi-directional half duplex (one-directional communication at a given point of time) two wire serial interface bus. The concept of I2C bus was developed by „Philips Semiconductors“ in the early 1980’s. The original intention of I2C was to provide an easy way of connection between a microprocessor/microcontroller system and the peripheral chips in Television sets.

The I2C bus is comprised of two bus lines, namely; Serial Clock – SCL and Serial Data – SDA.



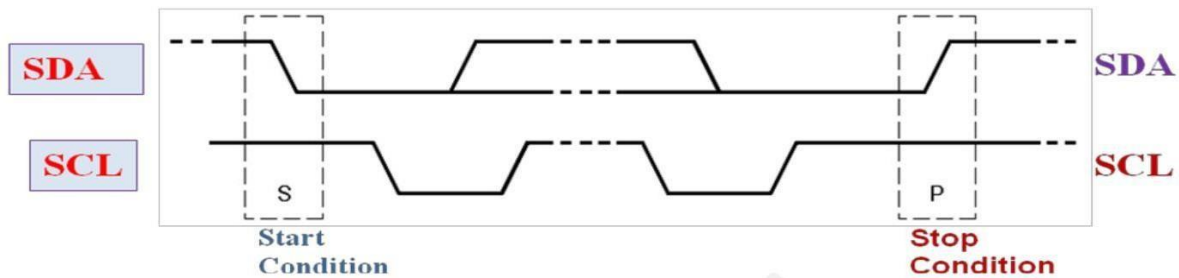
SCL line is responsible for generating synchronization clock pulses and SDA is responsible for transmitting the serial data across devices. I2C bus is a shared bus system to which many number of I2C devices can be connected. Devices connected to the I2C bus can act as either „Master“ device or „Slave“ device.

The „Master“ device is responsible for controlling the communication by initiating/terminating data transfer, sending data and generating necessary synchronization clock pulses.

Slave devices wait for the commands from the master and respond upon receiving the commands. Master and „Slave“ devices can act as either transmitter or receiver. Regardless whether a master is acting as transmitter or receiver, the synchronization clock signal is generated by the „Master“ device only. I2C supports multi masters on the same bus.

**The sequence of operation for communicating with an I2C slave device is:**

1. Master device pulls the clock line (SCL) of the bus to „HIGH“
2. Master device pulls the data line (SDA) „LOW“, when the SCL line is at logic „HIGH“ (This is the „Start“ condition for data transfer)



3. Master sends the address (7 bit or 10 bit wide) of the „Slave“ device to which it wants to communicate, over the SDA line.
4. Clock pulses are generated at the SCL line for synchronizing the bit reception by the slave device.
5. The MSB of the data is always transmitted first.
6. The data in the bus is valid during the „HIGH“ period of the clock signal
7. In normal data transfer, the data line only changes state when the clock is low.



**R/W<sub>r</sub>**

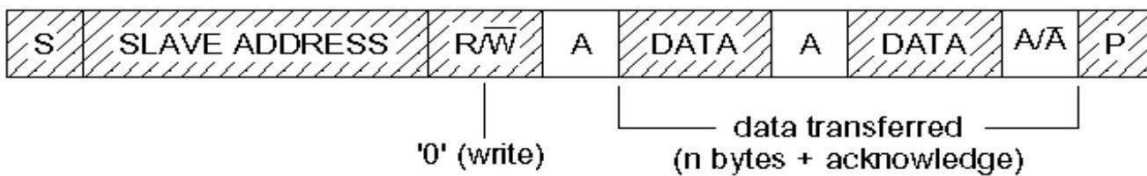
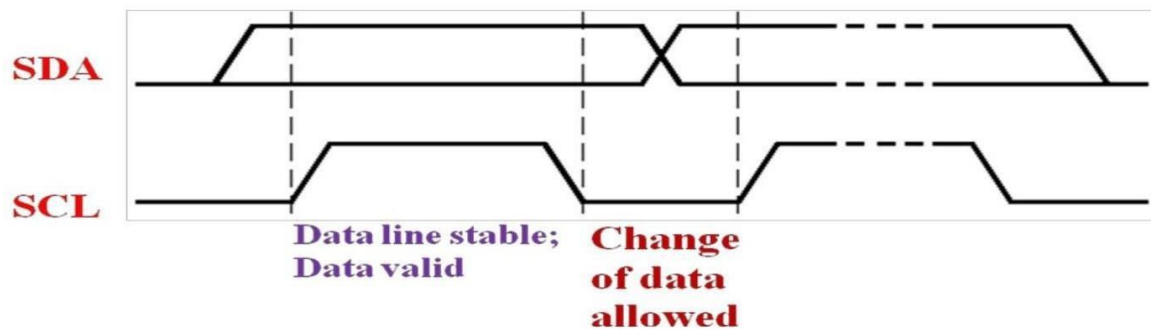
**0** – Master writes to the slave


**1** – Master read from slave

**ACK** – Generated by the slave whose address has been output.

8. Master waits for the acknowledgement bit from the slave device whose address is sent on the bus along with the Read/Write operation command.

9. Slave devices connected to the bus compares the address received with the address assigned to them
10. The Slave device with the address requested by the master device responds by sending an acknowledge bit (Bit value =1) over the SDA line
11. Upon receiving the acknowledge bit, master sends the 8bit data to the slave device over SDA line, if the requested operation is „Write to device“.
12. If the requested operation is „Read from device“, the slave device sends data to the master over the SDA line.
13. Master waits for the acknowledgement bit from the device upon byte transfer complete for a write operation and sends an acknowledge bit to the slave device for a read operation
14. Master terminates the transfer by pulling the SDA line „HIGH“ when the clock line SCL is at logic „HIGH“ (Indicating the „STOP“ condition).



 from master to slave

 from slave to master

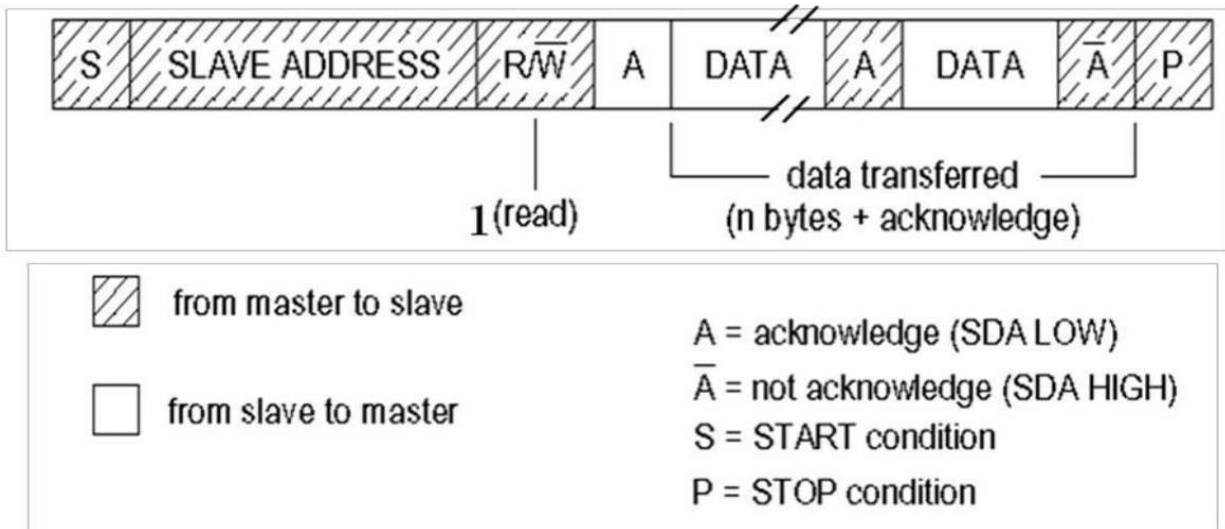
MBC605

A = acknowledge (SDA LOW)

$\bar{A}$  = not acknowledge (SDA HIGH)

S = START condition

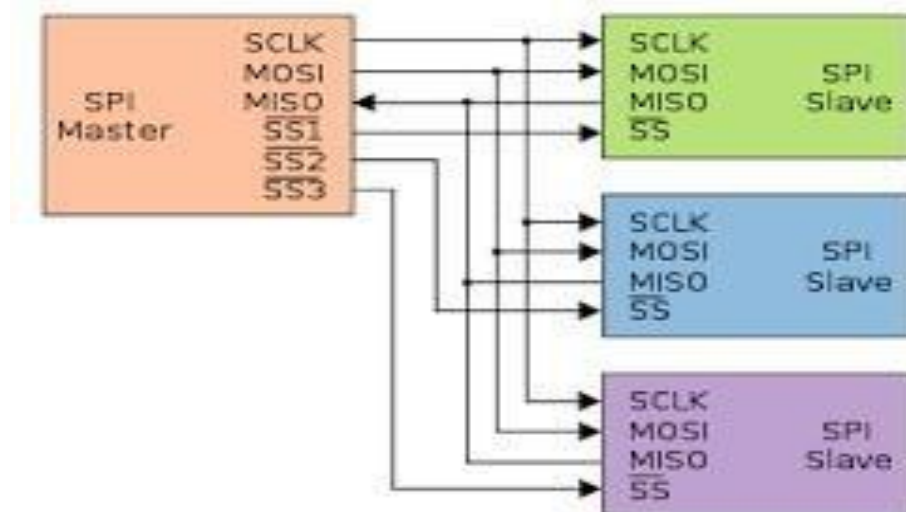
P = STOP condition



### Serial Peripheral Interface (SPI) Bus:

The Serial Peripheral Interface Bus (SPI) is a synchronous bi-directional full duplex four wire serial interface bus. The concept of SPI is introduced by Motorola. SPI is a single master multi-slave system.

- It is possible to have a system where more than one SPI device can be master, provided the condition only one master device is active at any given point of time, is satisfied.
- SPI is used to send data between Microcontrollers and small peripherals such as shift registers, sensors, and SD cards.



SPI requires four signal lines for communication. They are:

**Master Out Slave In (MOSI):** Signal line carrying the data from master to slave device. It is also known as Slave Input/Slave Data In (SI/SDI)

**Master In Slave Out (MISO):** Signal line carrying the data from slave to master device. It is also known as Slave Output (SO/SDO)

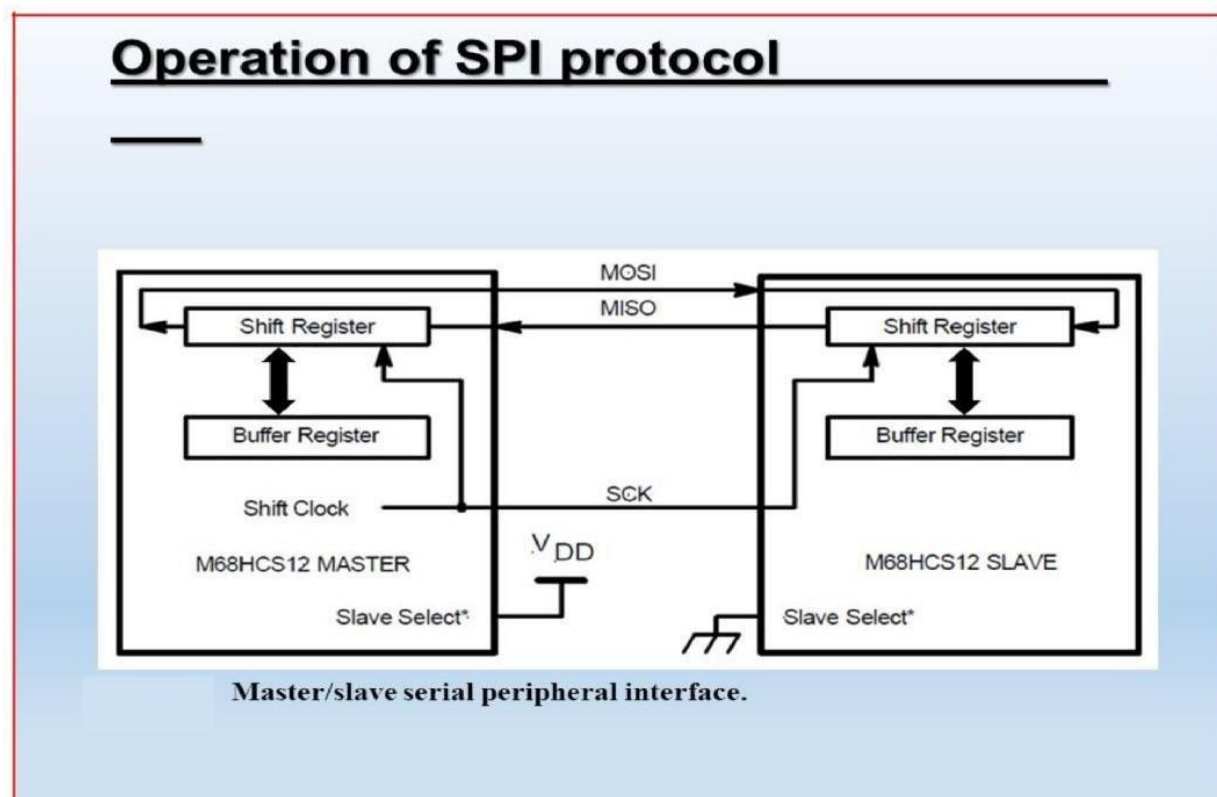
**Serial Clock (SCLK):** Signal line carrying the clock signals

**Slave Select (SS):** Signal line for slave device select. It is an active low signal.

The master device is responsible for generating the clock signal.

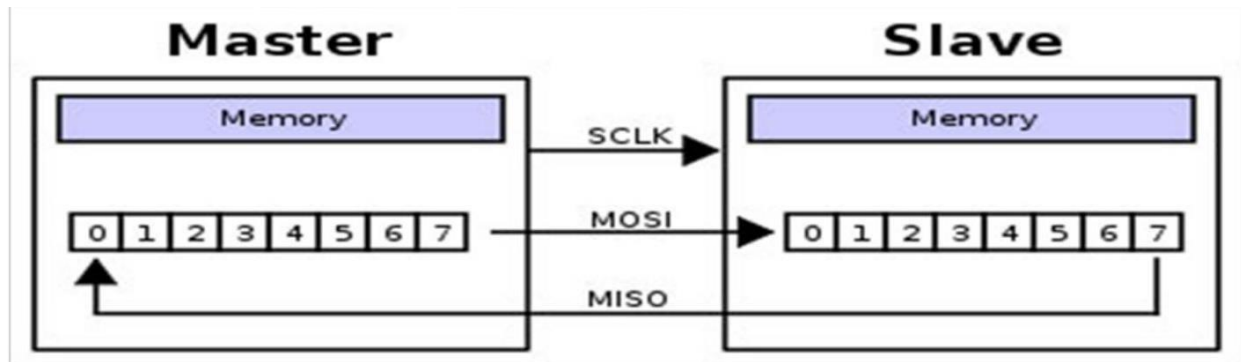
Master device selects the required slave device by asserting the corresponding slave devices slave select signal „LOW“.

- The data out line (MISO) of all the slave devices when not selected floats at high impedance state
- The serial data transmission through SPI Bus is fully configurable.
- SPI devices contain certain set of registers for holding these configurations.
- The Serial Peripheral Control Register holds the various configuration parameters like master/slave selection for the device, baudrate selection for communication, clock signal control etc.
- The status register holds the status of various conditions for transmission and reception. SPI works on the principle of „Shift Register“.
- The master and slave devices contain a special shift register for the data to transmit or receive.
- The size of the shift register is device dependent. Normally it is a multiple of 8.



- During transmission from the master to slave, the data in the master's shift register is shifted out to the MOSI pin and it enters the shift register of the slave device through the MOSI pin of the slave device.

- At the same time the shifted out data bit from the slave device's shift register enters the shift register of the master device through MISO pin



**Master shifts out data to Slave, and shift in data from Slave**

### **I2C V/S SPI:**

I2C	SPI
Speed limit varies from 100kbps, 400kbps, 1mbps, 3.4mbps depending on i2c version.	More than 1mbps, 10mbps till 100mbps can be achieved.
Half duplex synchronous protocol	Full Duplex synchronous protocol
Support Multi master configuration	Multi master configuration is not possible
Acknowledgement at each transfer	No Acknowledgement
Require Two Pins only SDA, SCL	Require separate MISO, MOSI, CLK & CS signal for each slave.
Addition of new device on the bus is easy	Addition of new device on the bus is not much easy as I2C
More Overhead (due to acknowledgement, start, stop)	Less Overhead
Noise sensitivity is high	Less noise sensitivity

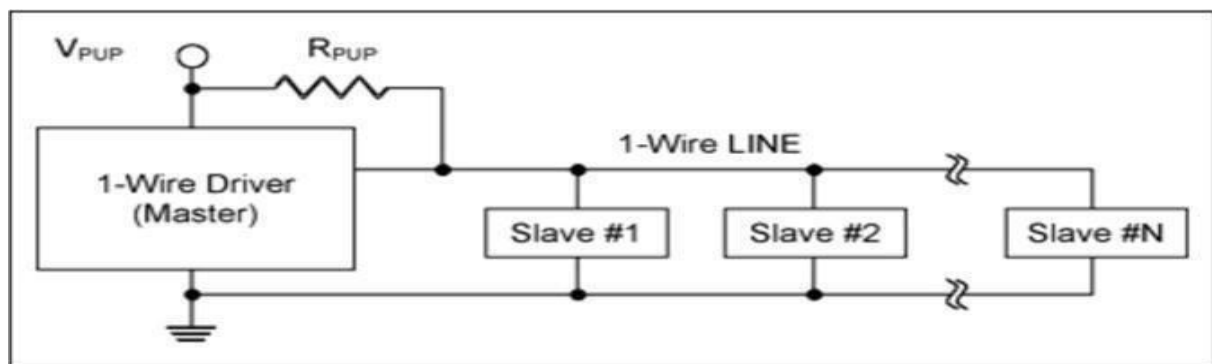
### 1-wire interface (protocol)

**1-** Wire is a device communications bus system designed by Dallas Semiconductor Corp. that provides low-speed data, signaling, and power over a single conductor.

1-Wire is similar in concept to I<sup>2</sup>C, but with lower data rates and longer range. It is typically used to communicate with small inexpensive devices such as digital thermometers and weather instruments.

One distinctive feature of the bus is the possibility of using only two wires: data and ground. To accomplish this, 1-Wire devices include an 800 pF capacitor to store charge, and to power the device during periods when the data line is active.

There is always one master in overall charge, which may be a PC or a microcontroller. The master initiates activity on the bus, simplifying the avoidance of collisions on the bus. Protocols are built into the software to detect collisions. After a collision, the master retries the required communication.



Many devices can share the same bus. Each device on the bus has a unique 64-bit serial number. The least significant byte of the serial number is an 8-bit number that tells the type of the device. The most significant byte is a standard (for the 1-wire bus) 8-bit CRC.

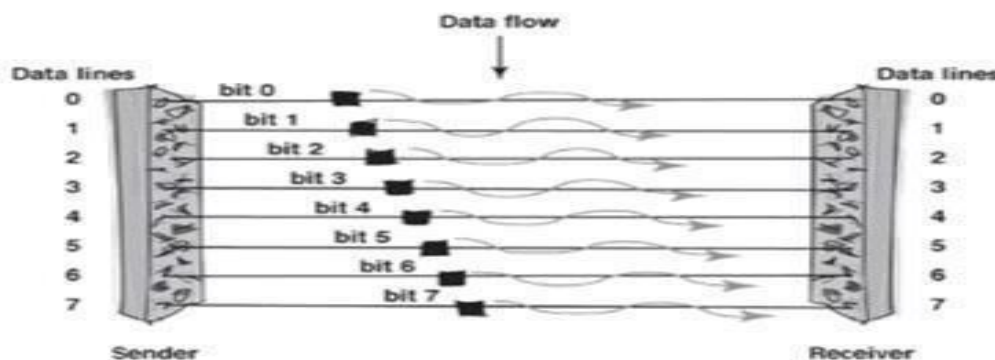
The master starts a transmission with a *reset* pulse, which pulls the wire to 0 volts for at least 480  $\mu$ s. This resets every slave device on the bus. After that, any slave device, if present, shows that it exists with a "presence" pulse: it holds the bus low for at least 60  $\mu$ s after the master releases the bus.

To send a "1", the bus master sends a very brief (1– 15  $\mu$ s) low pulse. To send a "0", the master sends a 60  $\mu$ s low pulse. When receiving data, the master starts sending a 1–15- $\mu$ s 0-volt pulse to slave each bit. If the transmitting device wants to send a "1", it lets the bus go transmitting pulled-up voltage. If the "0", it pulls slave wants to send the data line to ground for 60  $\mu$ s.

### **PARALLEL COMMUNICATION:**

In data transmission, parallel communication is a method of conveying multiple binary digits (bits) simultaneously. It contrasts with serial communication. The communication channel is the number of electrical conductors used at the physical layer to convey bits.

Parallel communication implies more than one such conductor. For example, an 8-bit parallel channel will convey eight bits (or a byte) simultaneously, whereas a serial channel would convey those same bits sequentially, one at a time. Parallel communication is and always has been widely used within integrated circuits, in peripheral buses, and in memory devices such as RAM.



## **2. Product level communication interface (External Communication**

**Interface):** The Product level communication interface" (External Communication Interface) is responsible for data transfer between the embedded system and other devices or modules

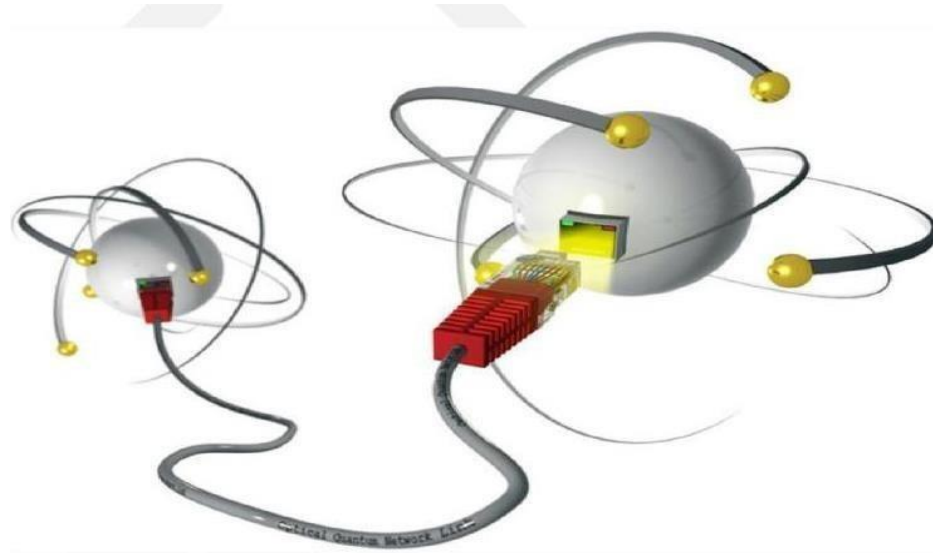
**It is classified into two types**

1. Wired communication interface
2. Wireless communication interface:

**1. Wired communication interface:** Wired communication interface is an interface used to transfer information over a wired network.

It is classified into following types.

1. **RS-232C/RS-422/RS 485**
2. **USB**



## **RS-232C:**

- RS-232 C (Recommended Standard number 232, revision C from the Electronic Industry Association) is a legacy, full duplex, wired, asynchronous serial communication interface
- RS-232 extends the UART communication signals for external data communication.
- UART uses the standard TTL/CMOS logic (Logic „High“ corresponds to bit value 1 and Logic „LOW“ corresponds to bit value 0) for bit transmission whereas RS232 use the EIA standard for bit transmission.
- As per EIA standard, a logic „0“ is represented with voltage between +3 and +25V and a logic „1“ is represented with voltage between -3 and -25V.
- In EIA standard, logic „0“ is known as „Space“ and logic „1“ as „Mark“.

**The RS232 interface define various handshaking and control signals for communication apart from the „Transmit“ and „Receive“ signal lines for data communication**

RS-232 supports two different types of connectors, namely; DB-9: 9-Pin connector and DB-25: 25-Pin connector.

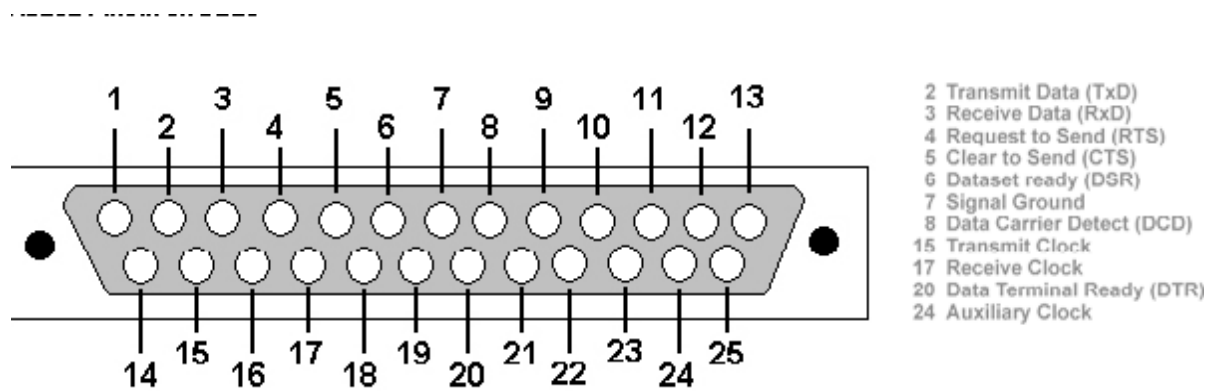


Fig: DB-25:25-Pin connector.

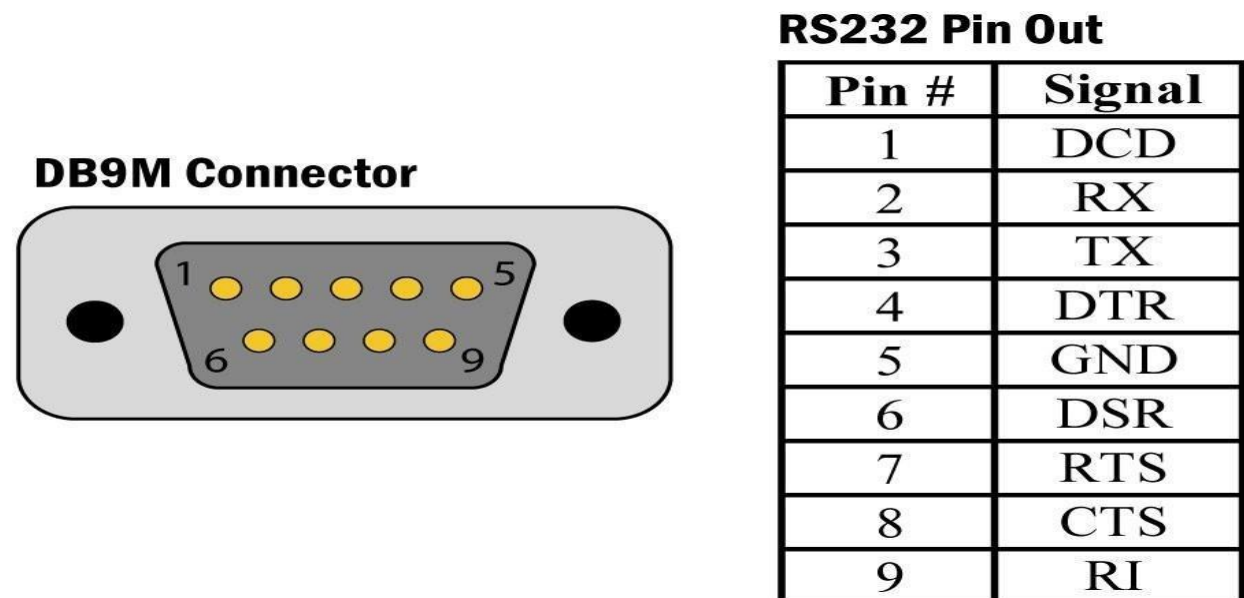
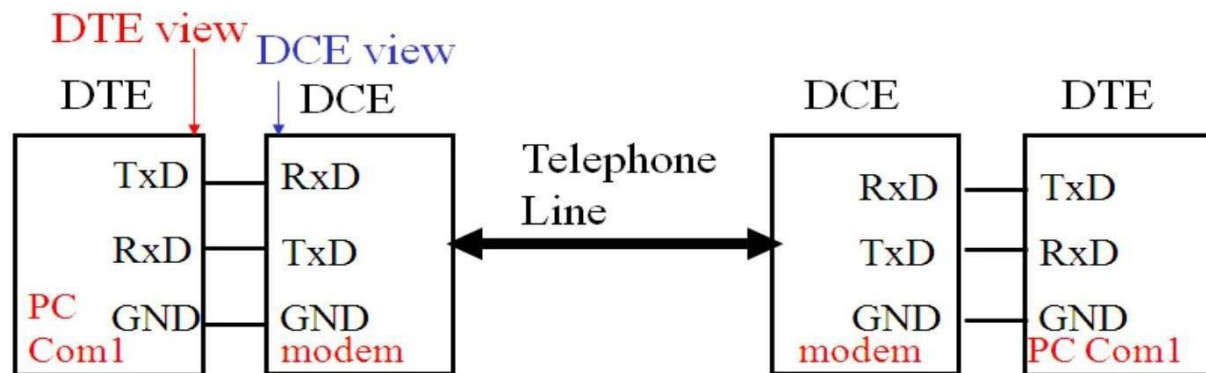


Fig: DB-9:9-Pin connector.



- RS-232 is a point-to-point communication interface and the devices involved in RS-232 communication are called „Data Terminal Equipment (DTE)“ and „Data Communication Equipment (DCE)“.
- If no data flow control is required, only TXD and RXD signal lines and ground line (GND) are required for data transmission and reception.
- The RXD pin of DCE should be connected to the TXD pin of DTE and vice versa for proper data transmission.
- If hardware data flow control is required for serial transmission, various control signal lines of the RS-232 connection are used appropriately.
- The control signals are implemented mainly for modem communication and some of them may be irrelevant for other type of devices.
- The Request to Send (RTS) and Clear To Send (CTS) signals co-ordinate the communication between DTE and DCE.
- Whenever the DTE has a data to send, it activates the RTS line and if the DCE is ready to accept the data, it activates the CTS line.
- The Data Terminal Ready (DTR) signal is activated by DTE when it is ready to accept data.
- The Data Set Ready (DSR) is activated by DCE when it is ready for establishing a communication link.
- DTR should be in the activated state before the activation of DSR.
- The Data Carrier Detect (DCD) is used by the DCE to indicate the DTE that a good signal is

being received.

- Ring Indicator (RI) is a modem specific signal line for indicating an incoming call on the telephone line.
- As per the EIA standard RS-232 C supports baudrates up to 20Kbps (Upper limit 19.2Kbps).
- The commonly used baudrates by devices are 300bps, 1200bps, 2400bps, 9600bps, 11.52Kbps and 19.2Kbps.
- The maximum operating distance supported in RS-232 communication is 50 feet at the highest supported baudrate.
- Embedded devices contain a UART for serial communication and they generate signal levels conforming to TTL/CMOS logic.
- A level translator IC like MAX 232 from Maxim Dallas semiconductor is used for converting the signal lines from the UART to RS-232 signal lines for communication.
- On the receiving side the received data is converted back to digital logic level by a converter IC.
- Converter chips contain converters for both transmitter and receiver.
- RS-232 uses single ended data transfer and supports only point-to-point communication and not suitable for multi-drop communication.

### **USB (UNIVERSAL SERIAL BUS):**

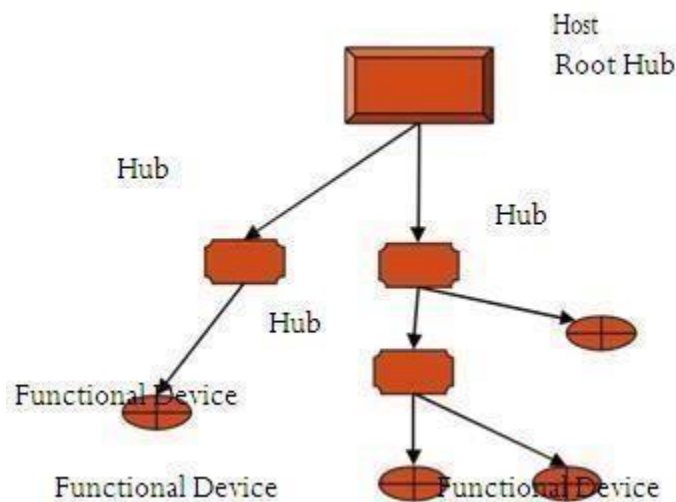
- External Bus Standard.
- Allows connection of peripheral devices.
- Connects Devices such as keyboards, mice, scanners, printers, joysticks, audio devices, disks.
- Facilitates transfers of data at 480 (USB 2.0 only), 12 or 1.5 Mb/s (mega-bits/second).
- Developed by a Special Interest Group including Intel, Microsoft, Compact, DEC, IBM, Northern Telecom and NEC originally in 1994.
- Low-Speed: 10 – 100 kb/s
- 1.5 Mb/s signaling bit rate

- Full-Speed: 500 kb/s – 10 Mb/s 12 Mb/s signaling bit rate
- High-Speed: 400 Mb/s

- 480 Mb/s signaling bit rate
- NRZI with bit stuffing used
- SYNC field present for every packet
- There exist two pre-defined connectors in any USB system - Series “A” and Series “B” Connectors.
- Series “A” cable: Connects USB devices to a hub port.
- Series “B” cable: Connects detachable devices (hot- swappable)

### Bus Topology:

- Connects computer to peripheral devices.
- Ultimately intended to replace parallel and serial ports
- Tiered Star Topology
- All devices are linked to a common point referred to as the root hub.
- Specification allows for up to  $127 (2^7 - 1)$  different devices.



- Four wire cable serves as interconnect of system - power, ground and two differential signaling lines.
- USB is a polled bus-all transactions are initiated by host.

**USB HOST:** Device that controls entire system usually a PC of some form. Processes data arriving to and from the USB port.

**USB HUB:** Tests for new devices and maintains status information of child devices. Serve as repeaters, boosting strength of up and downstream signals. Electrically isolates devices from one another - allowing an expanded number of devices.

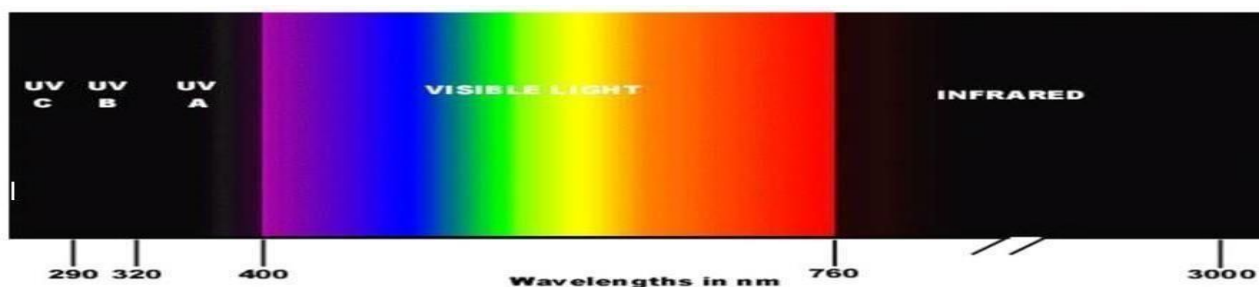
**2. Wireless communication interface :** Wireless communication interface is an interface used to transmission of information over a distance without help of wires, cables or any other forms of electrical conductors.

They are basically classified into following types

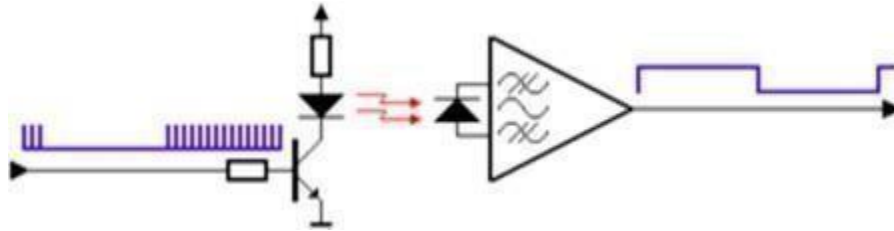
1. Infrared
2. Bluetooth
3. Wi-Fi
4. Zigbee
5. GPRS

### **INFRARED:**

- Infrared is a certain region in the light spectrum
- Ranges from  $.7\mu$  to  $1000\mu$  or  $.1\text{mm}$
- Broken into near, mid, and far infrared
- One step up on the light spectrum from visible light
- Measure of heat



Most of the thermal radiation emitted by objects near room temperature is infrared. Infrared radiation is used in industrial, scientific, and medical applications. Night-vision devices using active near-infrared illumination allow people or animals to be observed without the observer being detected.



### **IR transmission:**

The transmitter of an IR LED inside its circuit, which emits infrared light for every electric pulse given to it. This pulse is generated as a button on the remote is pressed, thus completing the circuit, providing bias to the LED.

The LED on being biased emits light of the wavelength of 940nm as a series of pulses, corresponding to the button pressed. However since along with the IR LED many other sources of infrared light such as us human beings, light bulbs, sun, etc, the transmitted information can be interfered. A solution to this problem is by modulation. The transmitted signal is modulated using a carrier frequency of 38 KHz (or any other frequency between 36 to 46 KHz). The IR LED is made to oscillate at this frequency for the time duration of the pulse. The information or the light signals are pulse width modulated and are contained in the 38 KHz frequency.

IR supports data rates ranging from 9600bits/second to 16Mbps

Serial infrared: 9600bps to 115.2 kbps

Medium infrared: 0.576Mbps to 1.152 Mbps

Fast infrared: 4Mbps

### **BLUETOOTH:**

**Bluetooth** is a wireless technology standard for short distances (using short-wavelength UHF band from 2.4 to 2.485 GHz) for exchanging data over radio waves in the ISM and mobile devices, and building personal area networks (PANs). Invented by telecom vendor Ericsson in 1994, it was originally conceived as a wireless alternative to RS-232 data cables.

Bluetooth uses a radio technology called frequency- hopping spread spectrum. Bluetooth divides transmitted data into packets, and transmits each packet on one of 79 designated Bluetooth channels. Each channel has a bandwidth of 1 MHz. It usually performs 800 hops per second, with Adaptive Frequency-Hopping (AFH) enabled

Originally, Gaussian frequency-shift keying (GFSK) modulation was the only modulation scheme available. Since the introduction of Bluetooth 2.0+EDR,  $\pi/4$ -DQPSK (Differential Quadrature Phase Shift Keying) and 8DPSK modulation may also be used between compatible devices. Bluetooth is a packet-based protocol with a master- slave structure. One master may communicate with up to seven slaves in a piconet. All devices share the master's clock. Packet exchange is based on the basic clock, defined by the master, which ticks at 312.5  $\mu$ s intervals.

A master BR/EDR Bluetooth device can communicate with a maximum of seven devices in a piconet (an ad-hoc computer network using Bluetooth technology), though not all devices reach this maximum. The devices can switch roles, by agreement, and the slave can become the master (for example, a headset initiating a connection to a phone necessarily begins as master—as initiator of the connection—but may subsequently operate as slave).

### **Wi-Fi:**

- Wi-Fi is the name of a popular wireless networking technology that uses radio waves to provide wireless high-speed Internet and network connections
- Wi-Fi follows the IEEE 802.11 standard
- Wi-Fi is intended for network communication and it supports Internet Protocol (IP) based communication
- Wi-Fi based communications require an intermediate agent called Wi-Fi router/Wireless Access point to manage the communications.
- The Wi-Fi router is responsible for restricting the access to a network, assigning IP address to devices on the network, routing data packets to the intended devices on the network.

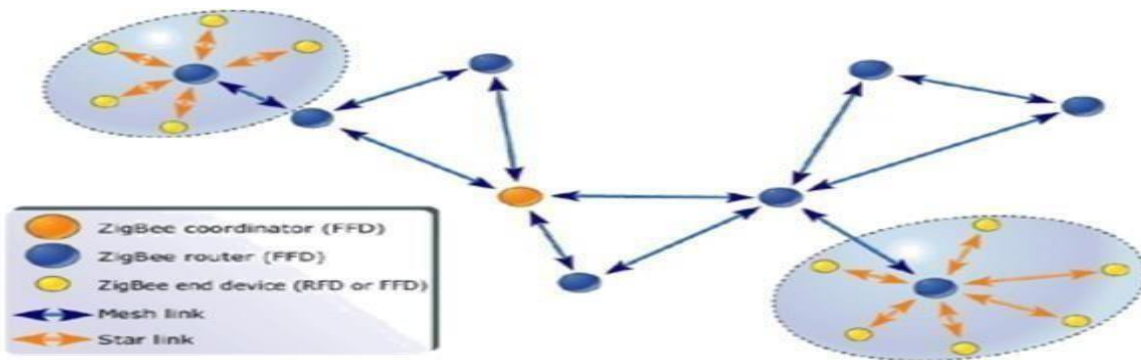


- Wi-Fi enabled devices contain a wireless adaptor for transmitting and receiving data in the form of radio signals through an antenna.
- Wi-Fi operates at 2.4GHZ or 5GHZ of radio spectrum and they co-exist with other ISM band devices like Bluetooth.
- A Wi-Fi network is identified with a Service Set Identifier (SSID). A Wi-Fi device can connect to a network by selecting the SSID of the network and by providing the credentials if the network is security enabled
- Wi-Fi networks implements different security mechanisms for authentication and data transfer.
- Wireless Equivalency Protocol (WEP), Wireless Protected Access (WPA) etc are some of the security mechanisms supported by Wi-Fi networks in data communication.

### **ZIGBEE:**

**Zigbee** is an IEEE 802.15.4-based specification for a suite of high- level communication protocols used to create personal area networks with small, low-power digital radios, such as for home automation, medical device data collection, and other low-power low-bandwidth needs, designed for small scale projects which need wireless connection. Hence, zigbee is a low-power, low data rate, and close proximity (i.e., personal area) wireless ad hoc network. The technology

defined by the zigbee specification is intended to be simpler and less expensive than other wireless personal area networks (WPANs), such as Bluetooth or Wi-Fi . Applications include wireless light switches, electrical meters with in-home-displays, traffic management systems, and other consumer and industrial equipment that require short-range low- rate wireless data transfer. Its low power consumption limits transmission distances to 10– 100 meters line-of-sight, depending on power output and environmental characteristics. Zigbee devices can transmit data over long distances by passing data through a mesh network of intermediate devices to reach more distant ones.



**Zigbee Coordinator:** The zigbee coordinator acts as the root of the zigbee network. The ZC is responsible for initiating the Zigbee network and it has the capability to store information about the network.

**Zigbee Router:** Responsible for passing information from device to another device or to another ZR.

**Zigbee end device:**End device containing zigbee functionality for data communication. It can talk only with a ZR or ZC and doesn't have the capability to act as a mediator for transferring data from one device to another.

Zigbee supports an operating distance of up to 100 metres at a data rate of 20 to 250 Kbps.

**General Packet Radio Service(GPRS):**

**General Packet Radio Service (GPRS)** is a packet oriented mobile data service on the 2G and 3G cellular communication system's global system for mobile communications (GSM). GPRS was originally standardized by European Telecommunications Standards Institute (ETSI) GPRS usage is typically charged based on volume of data transferred, contrasting with circuit switched data, which is usually billed per minute of connection time. Sometimes billing time is broken down to every third of a minute. Usage above the bundle cap is charged per megabyte, speed limited, or disallowed.

**Services offered:**

- GPRS extends the GSM Packet circuit switched data capabilities and makes the following services possible:
- SMS messaging and broadcasting
- "Always on" internet access
- Multimedia messaging service (MMS)
- Push-to-talk over cellular (PoC)
- Instant messaging and presence-wireless village Internet applications for smart devices through wireless application protocol (WAP).
- Point-to-point (P2P) service: inter-networking with the Internet (IP).
- Point-to-multipoint (P2M) service]: point-to- multipoint multicast and point-to-multipoint group calls.

**Text Book:-****1. Introduction to Embedded Systems – Shibu K.V Mc Graw Hill**