

EMBEDDED SYSTEMS DESIGN

Unit – I

Introduction to Embedded Systems

1. Define an Embedded system? Give examples?

Ans.) An embedded system is an electronic or electro mechanical system designed to perform a specific function and is a combination of both hardware and software.

An embedded system is a combination of 3 things:
Hardware, Software, Mechanical Components. And it is supposed to do one or set of specific tasks only.

Example 1: Washing Machine

A washing machine from an embedded systems point of view has:

Hardware: Buttons, Display & buzzer, electronic circuitry.

Software: It has a chip on the circuit that holds the software which drives controls & monitors the various operations possible.

Mechanical Components: the internals of a washing machine which actually wash the clothes control the input and output of water, the chassis itself.

Example 2: Air Conditioner

An Air Conditioner from an embedded systems point of view has:

Hardware: Remote, Display & buzzer, Infrared Sensors, electronic circuitry.

Software: It has a chip on the circuit that holds the software which drives controls & monitors the various operations possible. The software monitors the external temperature through the sensors and then releases the coolant or suppresses it.

Mechanical Components: the internals of an air conditioner the motor, the chassis, the outlet, etc

An embedded system is designed to do a specific job only. Example: a washing machine can only wash clothes, an air conditioner can control the temperature in the room in which it is placed. The hardware & mechanical components will consist all the physically visible things that are used for input, output, etc.

An embedded system will always have a chip (either microprocessor or microcontroller) that has the code or software which drives the system.

2. List out the difference between an embedded system and a general purpose computer

Ans.)

| GENERAL PURPOSE COMPUTER | EMBEDDED SYSTEM |
|--------------------------|-----------------|
|--------------------------|-----------------|

| | |
|--|--|
| <p>1) A system which is a combination of generic hardware and a general purpose operating system for executing a variety of applications.</p> <p>2) It contains a general purpose Operating System (GPOS)</p> <p>3) Performance is the key deciding factor in the selection of the system. Always 'faster is better'</p> <p>4) Response requirements are not time critical.</p> <p>5) Applications are alterable by the user.</p> <p>6) Need not be deterministic in execution behavior.</p> | <p>1) A system which is a combination of special purpose hardware and embedded operating system for executing a specific set of applications.</p> <p>2) It may or may not contain operating system for its functioning.</p> <p>3) Application specific requirements like Power requirements, memory usage etc. are the key deciding factors.</p> <p>4) For certain category of embedded systems like machine critical systems, the Response time requirement is highly critical.</p> <p>5) The firmware of the embedded system is preprogrammed and is not alterable by the end user</p> <p>6) Execution behavior is deterministic for certain types of embedded systems like 'Hard real Time systems'</p> |
|--|--|

3. What are the applications of embedded systems?

Ans.) The application areas and the products in the embedded domain are countless.

1. Consumer electronics: Camcorders, cameras, etc.
2. Household appliances: Television, DVD players, washing machine, fridge, microwave oven, etc.
3. Home automation and security systems: Air conditioners, sprinklers, intruder detection alarms, closed circuit television cameras, fire alarms, etc.
4. Automotive industry: Anti-lock breaking systems (ABS), engine control, ignition systems, automatic navigation systems, etc.
5. Telecom: Cellular telephones, telephone switches, handset multimedia applications
6. Computer peripherals: Printers, scanners, fax machines, etc.
7. Computer networking systems: Network routers, switches, hubs, firewalls, etc.
8. Healthcare: Different kinds of scanners, EEG, ECG machines etc.
9. Measurement & Instrumentation: Digital multi meters, digital CROs, logic analyzers PLC systems, etc.
10. Banking & Retail: Automatic teller machines (ATM) and currency counters, point of sales (POS)
11. Card Readers: Barcode, smart card readers, hand held devices, etc.

4) Explain the classification of Embedded systems?

Ans.) **CLASSIFICATION OF EMBEDDED SYSTEMS**

It is possible to have a multitude of classifications for embedded systems, based on different Criteria,

Some of the criteria used in the classification of embedded systems are:

1. Based on generation
2. Complexity and performance requirements
3. Based on deterministic behaviour
4. Based on triggering.

The classification based on deterministic system behaviour is applicable for 'Real Time' systems. The application/task execution behaviour for an embedded system can be either deterministic or non deterministic. Based on the execution behaviour, Real Time embedded systems are classified into Hard and Soft. Embedded Systems which are 'Reactive' in nature (Like process control systems in industrial control applications) can be classified based on the trigger. Reactive systems can be either event triggered or time triggered.

Classification Based on Generation

This classification is based on the order in which the embedded processing systems evolved from the first version to where they are today. As per this criterion, embedded systems can be classified into:

First Generation

The early embedded systems were built around 8bit microprocessors like 8085 and Z80, and 4bit microcontrollers. Simple in hardware circuits with firmware developed in Assembly code. Digital telephone keypads, stepper motor control units etc. are examples of this.

Second Generation

These are embedded systems built around 16bit microprocessors and 8 or 16 bit microcontrollers, following the first generation embedded systems. The instruction set for the second generation processors/controllers were much more complex and powerful than the first generation processors/controllers. Some of the second generation embedded systems contained embedded operating systems for their operation. Data Acquisition Systems, SCADA systems, etc. are examples of second generation embedded systems.

Third Generation

With advances in processor technology, embedded system developers started making use of powerful 32bit processors and 16bit microcontrollers for their design. A new concept of application and domain specific processors/controllers like Digital Signal Processors (DSP) and Application Specific Integrated Circuits (ASICs) came into the picture. The instruction set of processors became more complex and powerful and the concept of instruction pipelining also evolved. The Processor market was flooded with different types of processors from different vendors. Processors like Intel Pentium, Motorola 68K, etc. gained attention in high performance embedded requirements. Dedicated embedded real time and general purpose operating systems entered into the embedded market. Embedded system spread its ground to areas like robotics, industrial process control, media, networking. Etc.

Fourth Generation

The advent of System on Chips (SoC), reconfigurable processors and multicore processors are bringing high performance, tight integration and miniaturisation into the embedded device market. The SoC technique implements a total system on a chip by integrating different functionalities with a processor core on an integrated circuit. The fourth generation embedded systems are making use of high performance real time embedded operating systems for their functioning. Smart phone devices, mobile internet devices (MIDs), etc. are examples of fourth generation embedded systems.

Classification Based on Complexity and Performance

This classification is based on the complexity and system performance requirements. According to this classification, embedded systems can be grouped into:

Small -Scale Embedded Systems Embedded systems

These are simple in application needs and where the performance requirements are not time critical fall under this category. An electronic toy is a typical example of a small-scale embedded system. Small-scale embedded systems are usually built around low performance and low cost 8 or 16 bit microprocessors/microcontrollers. A small-scale embedded system may or may not contain an operating system for its functioning.

Medium -Scale Embedded Systems Embedded systems

These are slightly complex in hardware and firmware (software) requirements fall under this category. Medium -scale embedded systems are usually built around medium performance, low cost 16 or 32 bit microprocessors/microcontrollers or digital signal processors. They usually contain an embedded operating system (either general purpose or real time operating system) for functioning.

Large -Scale Embedded Systems/Complex Systems Embedded systems

These are the systems which involve highly complex hardware and firmware requirements fall under this category. They are employed in mission critical applications demanding high performance. Such systems are commonly built around high performance 32 or 64 bit RISC processors/controllers or Reconfigurable System on Chip(RSoC) or multi -core processors and programmable logic devices. They may contain multiple processors/controllers and co-units/hardware accelerators for offloading the processing requirements from the main processor of the system. Decoding/encoding of media, cryptographic function implementation, etc. are examples for processing requirements which can be implemented using a co-processor/hardware accelerator. Complex embedded systems usually contain a high performance Real Time Operating System (RTOS) for task scheduling, prioritization and management.

Classification Based On deterministic behavior

This classification is applicable for “Real Time” systems.

The task execution behaviour for an embedded system may be deterministic or non-deterministic. Based on execution behavior Real Time embedded systems are divided into Hard and Soft.

Classification Based On triggering

Embedded systems which are “Reactive” in nature can be based on triggering.

Reactive systems can be:

Event triggered, Time triggered

5) Explain the purpose of an Embedded system?

Ans.) Each embedded system is designed to serve the purpose of any one or a combination of the following tasks:

1. Data collection/Storage/Representation
2. Data communication
3. Data (signal) processing
4. Monitoring
5. Control
6. Application specific user interface

Data Collection/Storage/Representation

An embedded system designed for the purpose of data collection performs acquisition of data from the external world. Data collection is usually done for storage, analysis, manipulation and transmission. The term "data" refers all kinds of information, viz, text, voice, image, video, electrical signals and any other measurable quantities. Data can be either analog (continuous) or digital (discrete). Embedded systems with analog data capturing techniques collect data directly in the form of analog signals whereas embedded systems with digital data collection mechanism converts the analog signal to corresponding digital signal using analog to digital (A/D) converters and then collects the binary equivalent of the analog data. If the data is digital, it can be directly captured without any additional interface by digital embedded systems.

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.

Examples:

Analog and digital CROs without storage memory are typical examples of this. Any measuring equipment used in the medical domain for monitoring without storage functionality also comes under this category.

Some of them give the user a meaningful representation of the collected data by visual (graphical/quantitative) or audible means using display units [Liquid Crystal Dis-play (LCD), Light Emitting Diode (LED), etc.] buzzers, alarms, etc.

A digital camera is a typical example of an embedded system with data collection, storage, and representation of data. Images are captured and the captured image may be stored within the memory of the camera. The captured image can also be presented to the user through a graphic LCD unit.

Data Communication

Embedded data communication systems are deployed in applications ranging from complex satellite communication systems to simple home networking systems. The

communication is achieved either by a wire -line medium or by a wireless medium. Wire -line medium was the most common choice in all olden days embedded systems. As technology is changing, wireless medium is becoming the standard for data communication in embedded systems. A wireless medium offers cheaper connectivity solutions and make the communication link free from the hassle of wire bundles. Data can either be transmitted by analog means or by digital means. Modern industry trends are settling towards digital communication. The data collecting embedded terminal itself can incorporate data communication units like wireless modules (Bluetooth, ZigBee, Wi-Fi, EDGE, GPRS, etc.) or wire -line modules (RS -232C, USB, TCP/IP, PS2, etc.).

Data (Signal) Processing

As mentioned earlier, the data (voice, image, video, electrical signals and other measurable quantities) collected by embedded systems may be used for various kinds of data processing. Embedded systems with signal processing functionalities are employed in applications demanding signal processing like speech coding, synthesis, audio video codec, transmission applications, etc.

A digital hearing aid is a typical example of an embedded system employing data processing. Digital hearing aid improves the hearing capacity of hearing impaired persons.

Monitoring

Embedded systems falling under this category are specifically designed for monitoring purpose. Almost all embedded products coming under the medical domain are with monitoring functions only. They are used for determining state of some variables using input sensors. They cannot impose control over variables. A very good example is the electro cardiogram (ECG) machine for monitoring the heartbeat of a patient. The machine is intended to do the monitoring of the heartbeat. It cannot impose control over the heartbeat. The sensors used in ECG are the different electrodes connected to the patient's body. •

Some other examples of embedded systems with monitoring function are measuring instruments like digital CRO, digital multimeters, logic analyzers, etc. used in Control & Instrumentation applications. They are used for knowing (monitoring) the status of some variables like current, voltage, etc. They cannot control the variables in turn.

Control

Embedded systems with control functionalities impose control over some variables according to the changes in input variables. A 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 system used in our home to control the room temperature to a specified limit is a typical example for embedded system for control purpose.

Application Specific User Interface

These are embedded systems with application -specific user interfaces like buttons, switches, keypad, lights, bells, display units, etc. Mobile phone is an example for this. In mobile phone the user interface is provided through the keypad, graphic LCD module, system speaker, vibration alert, etc.

6) Explain the characteristics of Embedded systems?

Ans.) CHARACTERISTICS OF AN EMBEDDED SYSTEM

Unlike general purpose computing systems, embedded systems possess certain specific characteristics and these characteristics are unique to each embedded system. Some of the important characteristics of an embedded system are:

1. Application and domain specific
2. Reactive and Real Time
3. Operates in harsh environments
4. Distributed
5. Small size and weight
6. Power concerns

Application and Domain Specific

Each embedded system is having certain functions to perform and they are developed in such a manner to do the intended functions only. They cannot be used for any other purpose. It is the major criterion which distinguishes an embedded system from a general purpose system. For example, you cannot replace the embedded control unit of your microwave oven with your air conditioner's embedded control unit, because the embedded control units of microwave oven and air conditioner are specifically designed to perform certain specific tasks. Also you cannot replace an embedded control unit developed for a particular domain say telecom with another control unit designed to serve another domain like consumer electronics.

Reactive and Real Time

Embedded systems produce changes in output in response to the changes in the input. So they are generally referred as Reactive Systems. Real Time System operation means the timing behaviour of the system should be deterministic meaning the system should respond to requests or tasks in a known amount of time. A Real Time system should not miss any deadlines for tasks or operations. It is not necessary that all embedded systems should be Real Time - in operations. Embedded applications or systems which are mission critical, like flight control systems, Antilock Brake Systems (ABS), etc. are examples of Real Time systems. The design of an embedded Real time system should take the worst case scenario into consideration.

Operates in Harsh Environment

It is not necessary that all embedded systems should be deployed in controlled environments. The environment in which the embedded system deployed may be a

dusty one or a high temperature zone or an area subject to vibrations and shock. Systems placed in such areas should be capable to with stand all these adverse operating conditions. The design should take care of the operating conditions of the area where the system is going to implement. High temperature zone, Power supply fluctuations, corrosion and component aging, etc. are the factors that need to be taken into consideration for embedded systems to work in harsh environments.

Distributed

The term distributed means that embedded systems may be a part of larger systems. Many numbers of such distributed embedded systems form a single large embedded control unit. For example is the Automatic Teller Machine (ATM). An ATM contains a card reader embedded unit, responsible for reading and validating the user's ATM card, transaction unit for performing transactions, a currency dispatching/ vending currency to the authorised person and a printer unit for printing the transaction details. We can visualise these as independent embedded systems. But they work together to achieve a common goal.

Another typical example of a distributed embedded system is the Supervisory Control And Data Acquisition (SCADA) system used in Control & Instrumentation applications, which contains physically distributed individual embedded control units connected to a supervisory module

Small Size and Weight

Product aesthetics is an important factor in choosing a product. For example, when you plan to buy a new mobile phone, you may make a comparative study on the pros and cons of the products available in the market. Definitely the product aesthetics (size, weight, shape, style, etc.) will be one of the deciding factors to choose a product. People believe in the phrase "Small is beautiful". Moreover it is convenient to handle a compact device than a bulky product. In embedded domain also compactness is a significant deciding factor. Most of the application demands small sized and low weight products.

Power Concerns

Power management is another important factor that needs to be considered in designing embedded systems. Embedded systems should be designed in such a way as to minimise the heat dissipation by the system. The production of high amount of heat demands cooling requirements like cooling fans which in turn occupies additional space and make the system bulky. Nowadays ultra low power components are available in the market. Select the design according to the low power components like low dropout regulators, and controllers/processors with power saving modes. Also power management is a critical constraint in battery operated application. The more the power consumption the less is the battery life.

7) What are the Quality attributes in Embedded Systems? Explain?

Ans.) **QUALITY ATTRIBUTES OF EMBEDDED SYSTEMS**

Quality attributes are the non-functional requirements that need to be documented properly in any system design. If the quality attributes are more concrete and measurable

it will give a positive impact on the system development process and the end product. The various quality attributes that needs to be addressed in any embedded system development are broadly classified into two. namely 'Operational Quality Attributes' and 'Non -Operational Quality Attributes'.

Operational Quality Attributes

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

1. Response
2. Throughput
3. Reliability
4. Maintainability
5. Security
6. Safety

Response

Response is a measure of quickness of the system. It gives you an idea about how fast your system is tracking the changes in input variables. Most of the embedded systems demand fast response which should be almost Real Time. For example, an embedded system deployed in flight control application should respond in a Real Time manner. Any response delay in the system will create potential damages to the safety of the flight as well as the passengers. It is not necessary that all embedded systems should be Real Time in response. For example, the response time requirement for an electronic toy is not at all time -critical. There is no specific deadline that this system should respond within this particular timeline.

Throughput

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

Reliability

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

Maintainability

Maintainability deals with support and maintenance to the end user or client in case of technical issues and product failures or on the basis of a routine system checkup. Reliability and maintainability are considered as two complementary disciplines. A more reliable system means a system with less corrective maintainability requirements and vice versa. As the reliability of the system increases, the chances of failure and non-functioning also reduces, thereby the need for maintainability is also reduced. Maintainability is closely related to the system availability. Maintainability can be broadly classified into two categories, namely, 'Scheduled or Periodic Maintenance (preventive maintenance)' and 'Maintenance to unexpected failures (corrective maintenance)'. A printer is a typical example for illustrating the two types of maintainability. An inkjet printer uses ink cartridges, which are consumable components and as per the printer manufacturer the end user should replace the cartridge after each 'n' number of printouts to get quality prints. This is an example for 'Scheduled or Periodic maintenance'. If the paper feeding part of the printer fails the printer fails to print and it requires immediate repairs to rectify this problem. This is an example of 'Maintenance to unexpected failure'. In both of the maintenances (scheduled and repair), the printer needs to be brought offline and during this time it will not be available for the user.

Hence it is obvious that maintainability is simply an indication of the availability of the product for use.

In any embedded system design, the ideal value for availability is expressed as

$$A_i = \text{MTBF} / (\text{MTBF} + \text{MTTR})$$

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

Security

Confidentiality, Integrity, 'Availability' (The term 'Availability' mentioned here is not related to the term 'Availability' mentioned under the 'Maintainability' section) are the three major measures of information security. Confidentiality deals with the protection of data and application from unauthorised disclosure. Integrity deals with the protection of data and application from unauthorised modification. Availability deals with protection of data and application from unauthorized users.

A very good example of the 'Security' aspect in an embedded product is a Personal Digital Assistant (PDA). The PDA can be either a shared resource (e.g. PDAs used in LAB setups) or an individual one. If it is a shared one there should be some mechanism in the form of a user name and password to access into a particular person's profile—This is an example of 'Availability'. Also all data and applications present in the PDA need not be accessible to all users. Some of them are specifically accessible to administrators only. For achieving this, Administrator and user levels of security should be implemented -An example of Confidentiality. Some data present in the PDA may be visible to all users but there may not be necessary permissions to alter the data by the users. That is Read Only access is allocated to all users—An example of Integrity.

Safety

'Safety' and 'Security' are two confusing terms. Sometimes you may feel both of them as a single attribute. But they represent two unique aspects in quality attributes. Safety deals with the possible damages that can happen to the operators, public and the environment due to the breakdown of an embedded system or due to the emission of radioactive or hazardous materials from the embedded products. The breakdown of an embedded system may occur due to a hardware failure or a firmware failure. Safety analysis is a must in product engineering to evaluate the anticipated damages and determine the best course of action to bring down the consequences of the damages to an acceptable level. As stated before, some of the safety threats are sudden (like product breakdown) and some of them are gradual (like hazardous emissions from the product).

Non -Operational Quality Attributes

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

1. Testability & Debug -ability
2. Evolvability
3. Portability
4. Time to prototype and market
5. Per unit and total cost.

Testability & Debug -ability

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

Evolvability

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

Portability

Portability is a measure of 'system independence'. An embedded product is said to be portable if the product is capable of functioning 'as such' in various environments, target processors/ controllers and embedded operating systems. The ease with which an embedded product can be on to a new platform is a direct measure of the re-work required. A standard embedded product should always be flexible and portable. In embedded products, the term 'porting' represents the migration of the embedded 'firmware written for one target processor (e.g. Intel x86) to a different target processor (say Hitachi SH3 processor). If the firmware is written in a high level language like 'C' with little target processor -specific functions (operating system extensions or compiler specific utilities), it is very easy to port the firmware for the new processor by replacing those 'target processor -specific functions' with the ones for the new target processor and re-compiling the program for the new target processor specific settings.

Time -to -Prototype and Market

Time -to -market is the time elapsed between the conceptualisation of a product and the time at which the product is ready for selling (for commercial product) or use (for non-commercial products). Product prototyping helps a lot in reducing time -to -market. Prototyping is an informal kind of rapid product development in which the important features of the product under consideration are developed. The time to prototype is also another critical factor. If the prototype is developed faster, the actual estimated development time can be brought down significantly. In order to shorten the time to prototype, make use of all possible options like the use of off -the -shelf components, re-usable assets, etc.

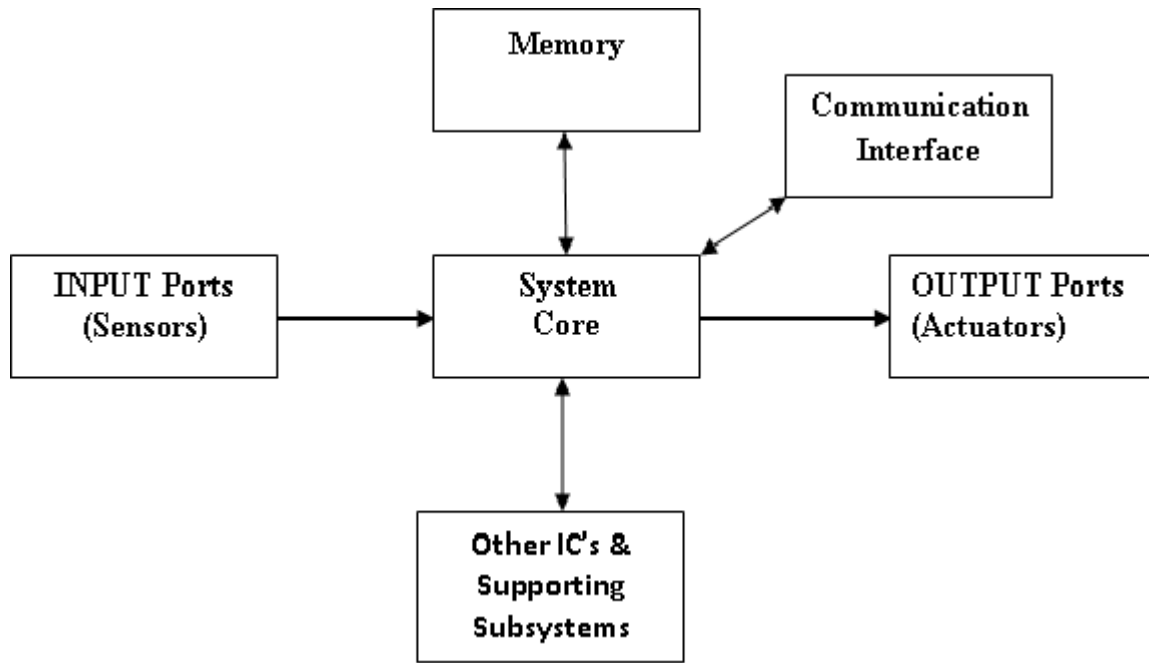
Per Unit Cost and Revenue

Cost is a factor which is closely monitored by both end user (those who buy the product) and product manufacturer (those who build the product). Cost is a highly sensitive factor for-commercial products. Any failure to position the cost of a commercial product at a nominal rate, may lead to the failure of the product in the market. Proper market study and cost benefit analysis should be carried out before taking a decision on the per unit cost of the embedded product. From a designer product development company perspective the ultimate aim of a product is to generate marginal profit. So the budget and total system cost should be properly balanced to provide a marginal profit.

Unit-II

Typical Embedded System

Elements of an Embedded System



1. Explain the System core of an embedded system?

- ▶ **Ans.)** Embedded systems are domain and application specific and are built around a central core. The core of the embedded system falls into any of the following categories:
- ▶ General purpose and Domain Specific Processors
 - Microprocessors
 - Microcontrollers
 - Digital Signal Processors
- ▶ Application Specific Integrated Circuits. (ASIC)
- ▶ Programmable logic devices(PLD's)
- ▶ 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.

Most of the embedded Systems in the industrial control and Monitoring applications makes use of the commonly available MP's or MC's.

Domains which require signal processing such as speech coding, speech recognition etc. makes use of DSP's.

MICROPROCESSORS

- ▶ A microprocessor is a silicon chip representing a central processing unit, which is capable of performing arithmetic as well as logical operations according to a predefined set of instructions.
- ▶ A microprocessor is a dependent unit and it requires the combination of other hardware like memory, timer unit, and *interrupt* controller, etc. for proper functioning.

Example: Intels 8086 Microprocessor

Features of 8086

- 1.The 8086 microprocessor is a 16 bit microprocessor.
- 2.The term "16 bit" means that its arithmetic logic unit, internal registers and most of its instructions are designed to work with 16 bit binary words.
- 3.The 8086 microprocessor has a 16 bit data bus.
- 4.It can read data from or write data to memory or ports either 16 bits or 8 bits at a time.
- 5.The 8086 microprocessor has a 20 bit address bus, so it can directly access 2^{20} or 1,048,576 (1M) locations.
- 6.The 8086 microprocessor can generate 16 bit I/O address; hence it can access 2^{16} or 65536 ports.
- 7.It is possible to perform bit, byte, word, and block operations in the 8086 microprocessor.
- 8.It performs the arithmetic and logical operations on bit, byte, word and decimal numbers including multiplication and division.
- 9.The 8086 microprocessor is designed to operate in two modes, the minimum mode and the maximum mode.
- 10.When only one CPU is used in the system, the 8086 microprocessor operates in the minimum mode.
- 11.In multiprocessor system, 8086 microprocessor operates in the maximum mode.
- 12.It provides 14, 16-bit registers.

MICROCONTROLLERS

- A microcontroller is a highly integrated chip that contains a CPU, 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.
- Texas Instrument's TMS 1000 Is considered as the world's first microcontroller.
- Some embedded system application require only 8 bit controllers whereas some requiring superior performance and computational needs demand 16/32 bit controllers.
- The instruction set of a microcontroller can be RISC or CISC.
- Microcontrollers are designed for either general purpose application requirement or domain specific application requirement.

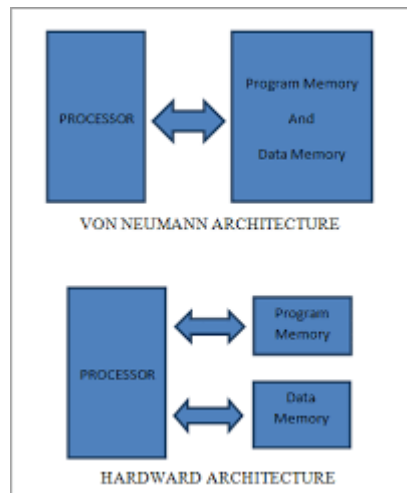
Architectures used for processor/controller design are Harvard or Von- Neumann.

▶ **Harvard architecture**

- ▶ It has separate buses for instruction as well as data fetching.
- ▶ Easier to pipeline, so high performance can be achieve.
- ▶ Comparatively high cost.
- ▶ Since data memory and program memory are stored physically in different locations, no chances exist for accidental corruption of program memory

▶ **Von-Neumann architecture**

- ▶ It shares single common bus for instruction and data fetching.
- ▶ Low performance as compared to Harvard architecture.
- ▶ It is cheaper.
- ▶ Accidental corruption of program memory may occur if data memory and program memory are stored physically in the same chip.



RISC and CISC are the two common Instruction Set Architectures

- ▶ **RISC**
- ▶ Reduced Instruction Set Computing
- ▶ It contains lesser number of instructions.
- ▶ Instruction pipelining and increased execution speed.
- ▶ Orthogonal instruction set(allows each instruction to operate on any register and use any addressing mode.
- ▶ Operations are performed on registers only, only memory operations are load and store.
- ▶ A larger number of registers are available.
- ▶ Programmer needs to write more code to execute a task since instructions are simpler ones.
- ▶ It is single, fixed length instruction.
- ▶ Less silicon usage and pin count.
- ▶ With Harvard Architecture.
- ▶ **CISC**
- ▶ Complex Instruction Set Computing
- ▶ It contains greater number of instructions.
- ▶ Instruction pipelining feature does not exist.

- ▶ Non-orthogonal set (all instructions are not allowed to operate on any register and use any addressing mode).
- ▶ Operations are performed either on registers or memory depending on instruction.
- ▶ The number of general purpose registers is very limited.
- ▶ Instructions are like macros in C language. A programmer can achieve the desired functionality with a single instruction which in turn provides the effect of using simpler single instruction in RISC.
- ▶ It is variable length instruction.
- ▶ More silicon usage since more additional decoder logic is required to implement the complex instruction decoding.
- ▶ Can be Harvard or Von- Neumann Architecture.

Digital Signal Processors

- DSP are powerful special purpose 8/16/32 bit microprocessor designed to meet the computational demands and power constraints of today's embedded audio, video and communication applications.
- DSP are 2 to 3 times faster than general purpose microprocessors in signal processing applications. This is because of the architectural difference between DSP and general purpose microprocessors.
- DSPs implement algorithms in hardware which speeds up the execution whereas general purpose processor implement the algorithm in software and the speed of execution depends primarily on the clock for the processors.

DSP includes following key units:

- ▶ **Program memory:** It is a memory for storing the program required by DSP to process the data.
- ▶ **Data memory:** It is a working memory for storing temporary variables and data/signal to be processed.
- ▶ **Computational engine:** It performs the signal processing in accordance with the stored program memory computational engine incorporated many specialized arithmetic units and each of them operates simultaneously to increase the execution speed. It also includes multiple hardware shifters for shifting operands and saves execution time.

- ▶ **I/O unit:** It acts as an interface between the outside world and DSP. It is responsible for capturing signals to be processed and delivering the processed signals.
 - ▶ **Examples:** Audio video signal processing, telecommunication and multimedia applications.
 - ▶ SOP(Sum of Products) calculation, convolution, FFT(Fast Fourier Transform), DFT(Discrete Fourier Transform), etc are some of the operation performed by DSP.

Application Specific Integrated Circuits. (ASIC)

- ASICs is a microchip design to perform a specific and unique applications.
- Because of using single chip for integrates several functions there by reduces the system development cost.
- Most of the ASICs are proprietary (which having some trade name) products, it is referred as Application Specific Standard Products(ASSP).
- As a single chip ASIC consumes a very small area in the total system. Thereby helps in the design of smaller system with high capabilities or functionalities.
- Examples: Energy Meters, a chip designed to run in a digital voice recorder

Programmable logic devices(PLD's)

A logic device is one which can perform any logic function

- Logic devices are broadly classified in to two Categories .
- They are , Fixed and programmable.
- As the name suggests, the circuits in a fixed logic device are permanent, they perform one function or set of functions - once manufactured, they cannot be changed.
- On the other hand, programmable devices are standard and offers a wide range of logic features and voltage characteristics - and these devices can be changed at any time to perform various logic functions.
- PLDs offer customers a wide range of logic capacity, features, speed, voltage characteristics.
- PLDs can be reconfigured to perform any number of functions at any time.

With programmable logic devices, designers use inexpensive software tools to quickly develop, simulate, and test their designs. Then, a design can be quickly programmed into a device, and immediately tested in a live circuit. The PLD that is used for this prototyping is the exact same PLD that will be used in the final production of a piece of end equipment, such as a network router, a DSL modem, a DVD player, or an automotive navigation system. There are no NRE costs and the final design is completed much faster than that of a custom, fixed logic device.

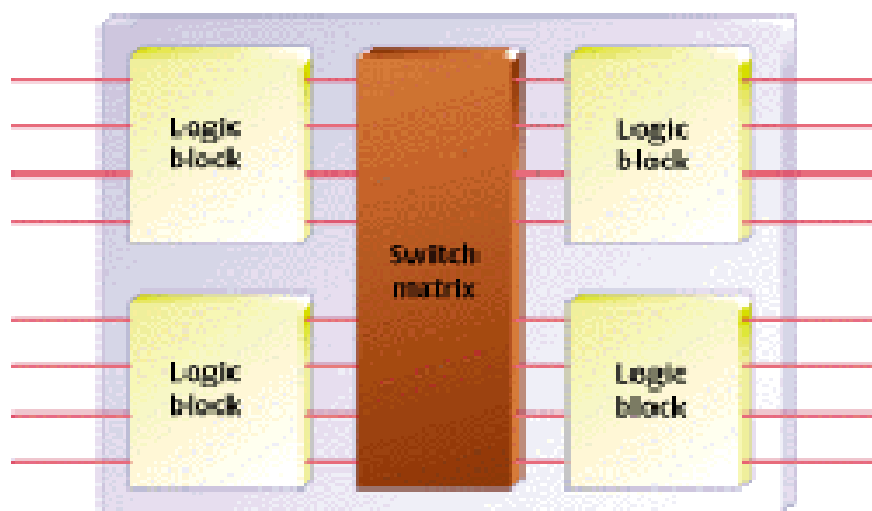
- PLDs having following two major types.

- CPLD(Complex Programmable Logic Device)
- FPGAs(Field Programmable Gate Arrays)

Complex programmable logic devices(CPLDs)

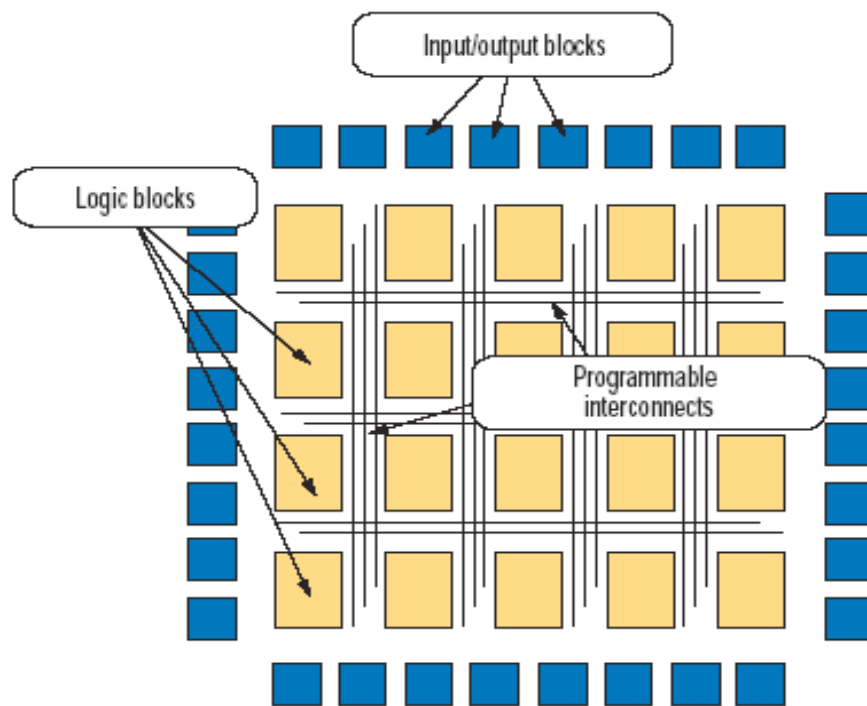
Instead of relying on a programming unit to configure a chip , it is advantageous to be able to perform the programming while the chip is still attached to its circuit board. This method of programming is known as “In-System programming (ISP)”. It is not usually provided for PLAs (or) PALs , but it is available for the more sophisticated chips known as “Complex programmable logic device”.

- ☞ “A Complex programmable logic device is a device that contain multiple combination of PLAs and PALs”. CPLDs offer much smaller amount of logic up to 10000 gates.
- ☞ A simple architecture of CPLD is shown below.



Field programmable Gate Arrays(FPGAs)

- FPGAs offer millions of gates of logic capacity, operate at 300MHz, can cost less than \$10, and offer integrated functions like processors and memory. FPGAs offer all of the features needed to implement most complex designs.



Commercial off-the-shelf components(COTs)

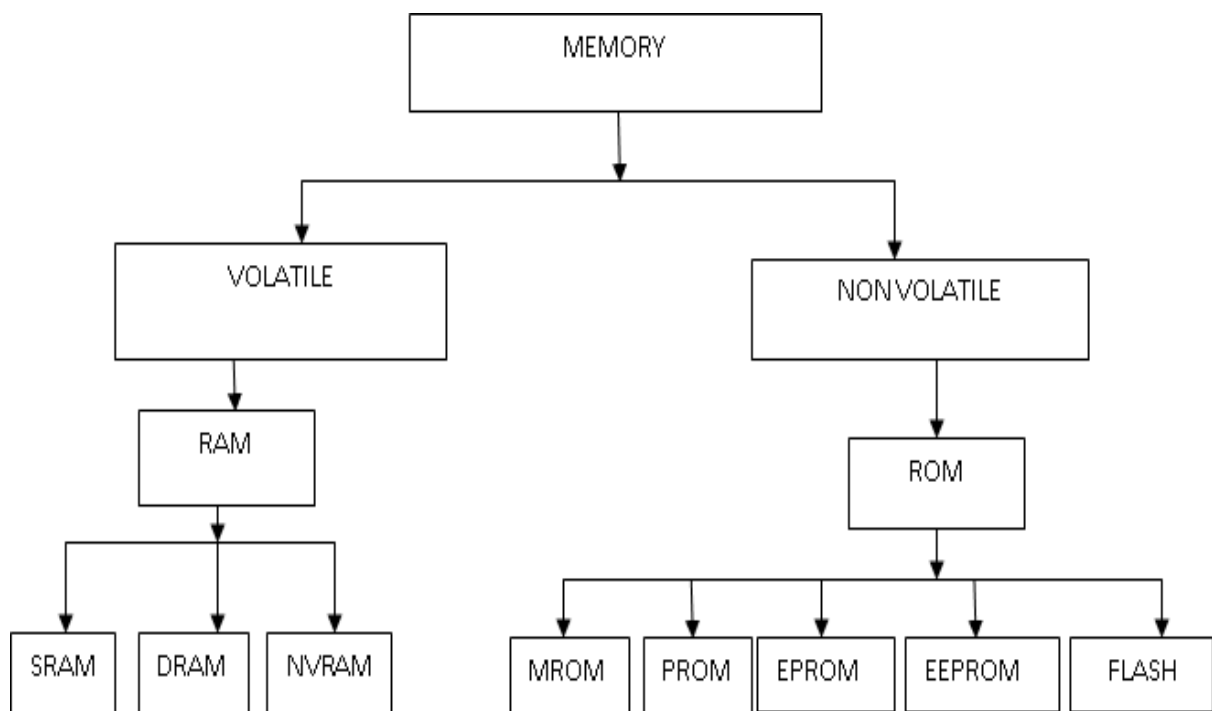
- ▶ A Commercial off the Shelf product is one which is used 'as- is'.
- ▶ The COTS components itself may be develop around a general purpose or domain specific processor or an ASICs or a PLDs.
- ▶ The major advantage of using COTS is that they are readily available in the market, are chip and a developer can cut down his/her development time to a great extent.
- ▶ The major drawback of using COTS components in embedded design is that the manufacturer of the COTS component may withdraw the product or discontinue the production of the COTS at any time if rapid change in technology occurs.
- ▶ **Advantages of COTS:**
 - Ready to use
 - Easy to integrate and Reduces development time
 - Example: The TCP/IP plug-in module available from 'WIZnet'



Explain different types of memory used in embedded systems”?

Ans) Memory is an important part of an Embedded system. It stores the control algorithm or firmware of an Embedded system.

- ▶ Some processors/controllers contain built in memory and it is referred as On-chip memory.
- ▶ Others do not contain any memory inside the chip and requires external memory to be connected with the controller/processor to store the control algorithm. It is called off -chip memory.



ROM(Read only Memory)

- ▶ The program memory or code storage memory of an embedded system stores the program instructions. The code memory 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 the following types.

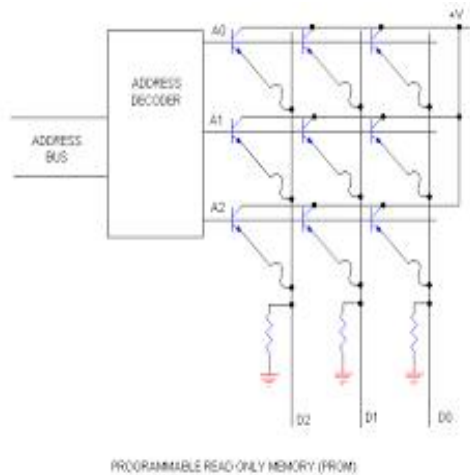
- ▶
- ▶ MROM
- ▶ PROM
- ▶ EPROM
- ▶ EEPROM
- ▶ FLASH

MROM

- ▶ Masked ROM (MROM) Masked ROM is a one-time programmable device. Masked ROM makes use of the hardwired technology for storing data. The device is factory programmed by masking and metallisation process at the time of production itself, according to the data provided by the end user.
- ▶ The primary advantage of this is low cost for high volume production. They are the least expensive type of solid state memory.
- ▶ The limitation with MROM based firmware storage is the inability to modify the device firmware against firmware upgrades. Since the MROM is permanent in bit storage, it is not possible to alter the bit information.

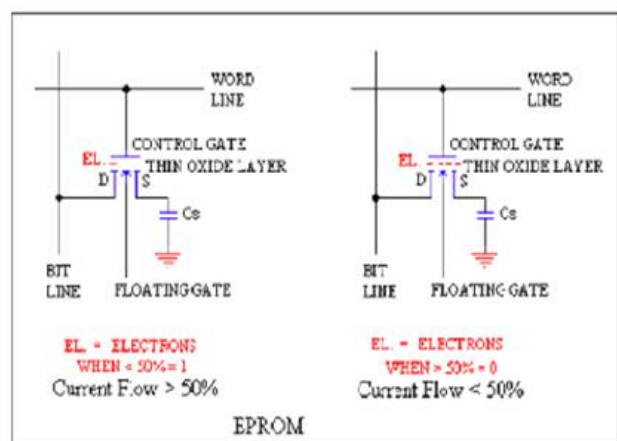
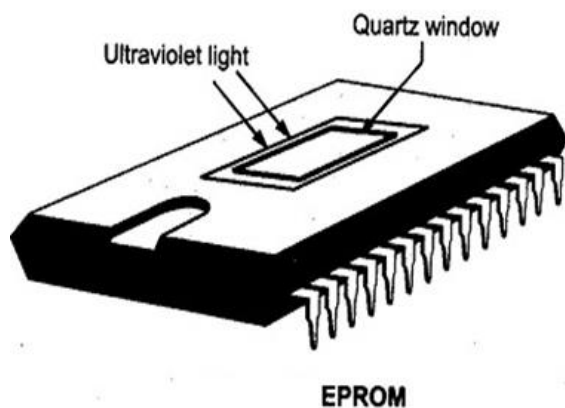
Programmable Read Only Memory (PROM) / (OTP)

Unlike Masked ROM Memory, One Time Programmable Memory (OTP) or PROM is not pre-programmed by the manufacturer. The end user is responsible for programming these devices. This memory 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 represent logic "1" whereas fuses which are blown/burned represent 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 finalised. It is a low cost solution for commercial production. OTPs cannot be reprogrammed.



Erased Programmable Read Only Memory (EPROM):

- EPROM gives the flexibility to reprogram same chip. EPROM stores the bit information by charging floating gate of an FET. Bit information is stored by using an EPROM programmer which applies the high voltage to charge the floating gate. EPROM contains a quartz crystal window for erasing the stored information. If the window is exposed to ultraviolet 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 put in a UV eraser device for 20 to 30 minutes. so it is a tedious and time-consuming process.



Electrically Erasable Programmable Read Only Memory (EEPROM)

- As the name indicates, the information contained in the EEPROM memory can be altered by using electrical signals at the register level. They can be erased and reprogrammed in-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).

FLASH

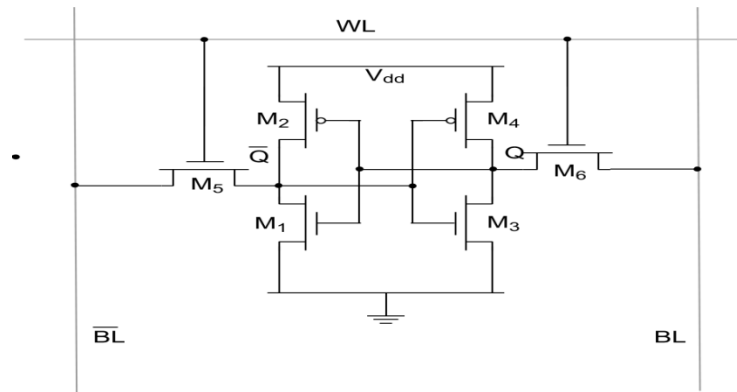
- ▶ FLASH is the latest ROM technology and is the most popular ROM technology used in today's embedded designs. FLASH memory is a variation of EEPROM technology. It combines the re-programmability of EEPROM and the high capacity of standard ROMs. FLASH memory is organised as sectors (blocks) or pages. FLASH memory stores information in an array of floating gate MOS-FET 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 1000 cycles. W27C512 from WINBOND is an example of 64KB FLASH memory.

Read -Write Memory/Random Access Memory (RAM)

- ▶ RAM is the data memory or working memory of the controller/processor. Controller/processor can read from it and write to it. 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).
- ▶ RAM generally falls into three categories: Static RAM (SRAM), dynamic RAM (DRAM) and non-volatile RAM (NVRAM)

Static RAM (SRAM) :

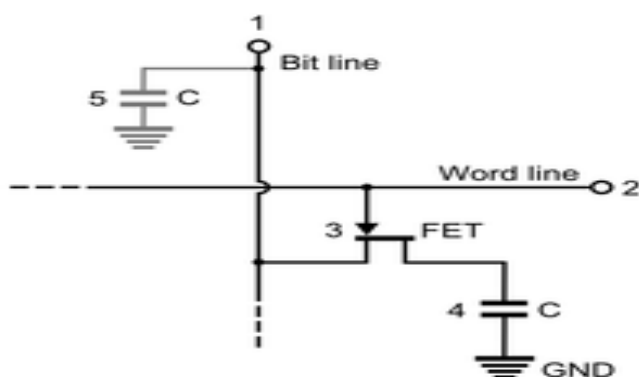
- ▶ Static RAM stores data in the form of voltage. They are made up of flip-flops. Static RAM is the fastest form of RAM available. Static RAM is realised using six transistors (or 6 MOSFETs). Four of the transistors are used for building the latch (flip-flop) part of the memory cell and two for controlling the access. SRAM is fast in operation due to its resistive networking and switching capabilities. In its simplest representation an SRAM cell can be visualised as shown in figure.
- ▶ This implementation in its simpler form can be visualised as two -cross coupled inverters with read/write control through transistors. The four transistors in the middle form the cross-coupled inverters. This can be visualised as shown in Fig From the SRAM implementation diagram, it is clear that access to the memory cell is controlled by the line Word Line, which controls the access Visualisation of SRAM cell transistors (MOSFETs) Q5 and Q6. The access transistors control the connection to bit lines B & B\.



- ▶ In order to write a value to the memory cell, apply the desired value to the bit control lines (For writing '1' make $B = 1$ and $\overline{B} = 0$; For writing 0, make $B = 0$ and $\overline{B} = 1$) and assert the Word Line (Make Word line high). This operation latches the bit written in the flip-flop. For reading operation assert both B and \overline{B} bit lines to 1 and set word line to 1.
- ▶ The major limitations of SRAM are low capacity and high cost.

Dynamic RAM (DRAM) :

- ▶ Dynamic RAM stores 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 milli- seconds interval.
- ▶ The MOSFET acts as the gate for the incoming and outgoing data whereas the capacitor acts as the bit storage unit.



SRAM CELL

- ▶ Made up of 6 CMOS transistors (MOSFET)
- ▶ Doesn't require refreshing
- ▶ Low capacity (Less dense)
- ▶ More expensive
- ▶ Fast in operation. Typical access time is 10ns

DRAM CELL

- ▶ Made up of a MOSFET and a capacitor
- ▶ Requires refreshing
- ▶ High capacity (Highly dense)
- ▶ Less expensive
- ▶ Slow in operation due to refresh requirements. Typical access time is 60ns. Write operation is faster than read operation.

NVRAM:

- ▶ Non-volatile RAM is a random access memory with battery backup. It 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 of NVRAM is expected to be around 10 years

Explain how memory is used in embedded system based on the type of interface?

Memory According to the Type of Interface:

- ▶ The interface (connection) of memory with the processor/controller can be of various types. It may be a parallel interface [The parallel data lines (D0 -D7) for an 8 bit processor/controller will be connected to D0 -D7 of the memory] or the interface may be a serial interface like I2C (Pronounced as I Square C. It is a 2 line serial interface) or it may be an SPI (Serial peripheral interface, 2+n line interface where n stands for the total number of SPI bus devices in the system). It can also be of a single wire interconnection (like Dallas 1-Wire interface). Serial interface is commonly used for data storage memory like EEPROM. The memory density of a serial memory is usually expressed in terms of kilobits, whereas that of a parallel interface memory is expressed in terms of kilobytes. Atmel Corporation's AT24C512 is an example for serial memory with capacity 512 kilobits and 1-wire interface.

Please refer to the section 'Communication Interface' for more details on I2C, SPI and 1-Wire Bus.

What is Memory Shadowing? Explain?

► Ans.) **Memory Shadowing :**

Generally the execution of a program or a configuration from a Read Only Memory (ROM) is very slow (120 to 200 ns) compared to the execution from a random access memory (40 to 70 ns). From the timing parameters it is obvious that RAM access is about three times as fast as ROM access. **Shadowing of memory is a technique adopted to solve the execution speed problem in processor-based systems.** In computer systems and video systems there will be a configuration holding ROM called Basic Input Output Configuration ROM or simply BIOS. In personal computer systems BIOS stores the hardware configuration information like the address assigned for various serial ports and other non-plug 'n' play devices, etc.

Usually it is read and the system is configured according to it during system boot up and it is time consuming. Now the manufacturers included a RAM behind the logical layer of BIOS at its same address as a shadow to the BIOS and the first step that happens during the boot up is copying the BIOS to the shadowed RAM and write protecting the RAM then disabling the BIOS reading. You may be thinking that what a stupid idea it is and why both RAM and ROM are needed for holding the same data. The answer is: RAM is volatile and it cannot hold the configuration data which is copied from the BIOS when the power supply is switched off. Only a ROM can hold it permanently. But for high system performance it should be accessed from a RAM instead of accessing from a ROM.

Explain about selection of memory in Embedded systems?

Ans) **Memory Selection for Embedded Systems**

Embedded systems require a program memory for holding the control algorithm. The memory requirement for an embedded system in terms of RAM and ROM is solely dependent on the type of the embedded system and the applications for which it is designed. For example, if the embedded system is designed using SoC or a microcontroller with on-chip RAM and ROM depending on the application need the on-chip memory may be sufficient for designing the total system.

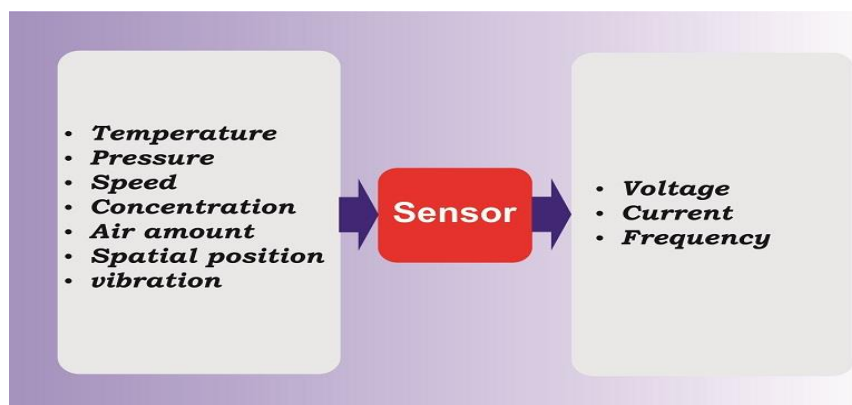
Let's consider a simple electronic toy design as an example. As the complexity of requirements are less and data memory requirement are minimal, we can think of a microcontroller with a few bytes of internal RAM, a few bytes or kilobytes (depending on the number of tasks and the complexity of tasks) of FLASH memory and a few bytes of EEPROM (if required) for designing the system. Hence there is no need for external memory at all. A PIC microcontroller device which satisfies the I/O and memory requirements can be used in this case.

A smart phone device with Windows mobile operating system is a typical example for embedded device with OS. Say 64MB RAM and 128MB ROM are the minimum requirements for running the Windows mobile device, indeed you need extra RAM and ROM for running user applications.

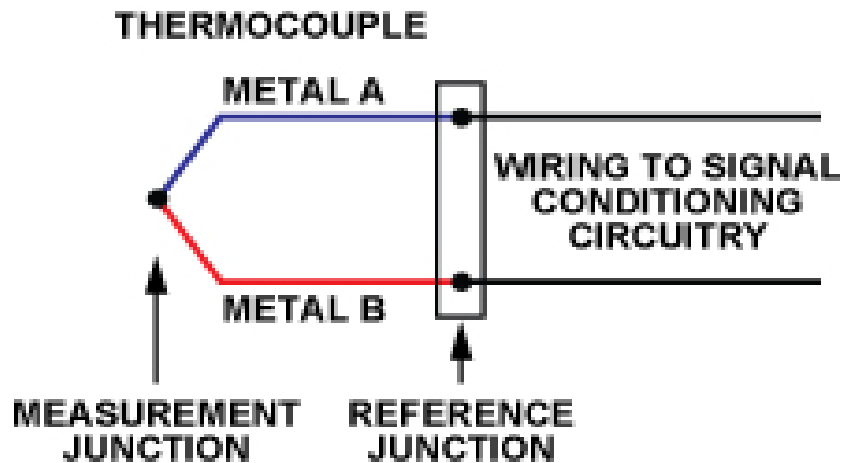
FLASH memory is the popular choice for ROM (program storage memory) in embedded applications. It is a powerful and cost-effective solid-state storage technology for mobile electronics devices and other consumer applications. FLASH memory comes in two major variants, namely, NAND and NOR FLASH. NAND FLASH is a high-density low cost non-volatile storage memory. On the other hand, NOR FLASH is less dense and slightly expensive. But it supports the Execute in Place (XIP) technique for program execution. The XIP technology allows the execution of code memory from ROM itself without the need for copying it to the RAM as in the case of conventional execution method. It is a good practice to use a combination of NOR and NAND memory for storage memory requirements, where NAND can be used for storing the program code and or data like the data captured in a camera device. NAND FLASH doesn't support XIP and if NAND FLASH is used for storing program code, a DRAM can be used for copying and executing the program code. NOR FLASH supports XIP and it can be used as the memory for boot loader or for even storing the complete program code.

What are sensors? Explain?

- ▶ A sensor is a device that detects and responds to some type of input from the physical environment. The specific input could be light, heat, motion, moisture, pressure, or any one of a great number of other environmental phenomena.



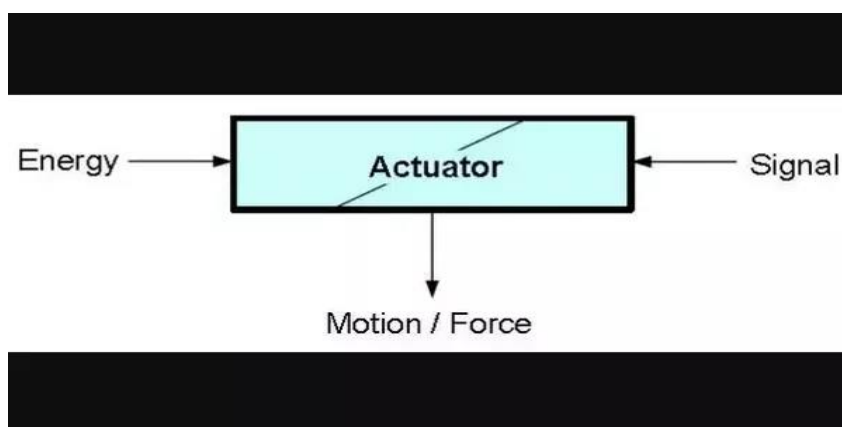
- ▶ **Examples:**
 - ▶ In a mercury-based glass thermometer, the input is temperature. The liquid contained expands and contracts in response, causing the level to be higher or lower on the marked gauge, which is human-readable.
- ▶ **Thermocouple:**
 - ▶ A thermocouple consists of two wires of dissimilar metals joined together at one end, called the *measurement* ("hot") junction. The other end, where the wires are not joined, is connected to the signal conditioning circuitry traces, typically made of copper. This junction between the thermocouple metals and the copper traces is called the *reference* ("cold") junction.*



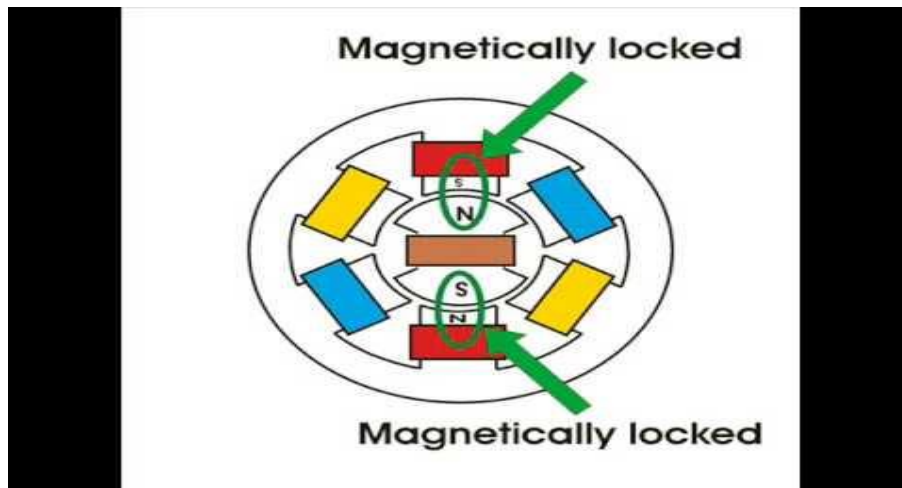
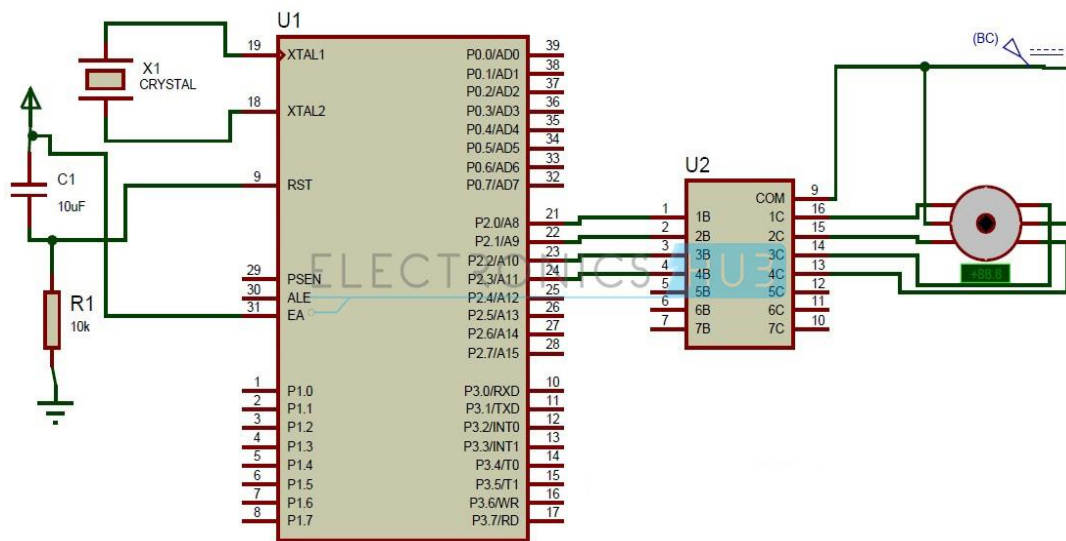
- ▶ The operating principal of a thermocouple is very simple and basic. When fused together the junction of the two dissimilar metals such as copper and constantan produces a “thermo-electric” effect which gives a constant potential difference of only a few millivolts (mV) between them. The voltage difference between the two junctions is called the “Seebeck effect” as a temperature gradient is generated along the conducting wires producing an emf. Then the output voltage from a thermocouple is a function of the temperature changes. If both the junctions are at the same temperature the potential difference across the two junctions is zero in other words, no voltage output as $V_1 = V_2$. However, when the junctions are connected within a circuit and are both at different temperatures a voltage output will be detected relative to the difference in temperature between the two junctions, $V_1 - V_2$. This difference in voltage will increase with temperature until the junctions peak voltage level is reached and this is determined by the characteristics of the two dissimilar metals used.

What are Actuators? Explain?

- ▶ Ans) An actuator is a mechanism for turning energy into motion. Actuator is used for output. It is a transducer that may be either mechanical or electrical which converts signals to corresponding physical actions.



Example: Stepper Motor



- ▶ **Circuit Components:**
- ▶ AT89C51 micro controller
- ▶ ULN2003A
- ▶ Stepper Motor
- ▶ Crystal
- ▶ Resistor
- ▶ Capacitor
- ▶ **Stepper Motor Control using 8051 Microcontroller Circuit Design:**

- ▶ The circuit consists of AT89C51 microcontroller, ULN2003A, Motor. AT89c51 is low power, high-performance, CMOS 8bit, 8051 family microcontroller. It has 32 programmable I/O lines. It has 4K bytes of Flash programmable and erasable memory. An external crystal oscillator is connected at the 18 and 19 pins of the microcontroller. Motor is connected to the port2 of the microcontroller through a driver IC.
- ▶ The ULN2003A is a current driver IC. It is used to drive the current of the stepper motor as it requires more than 60mA of current. It is an array of Darlington pairs. It consists of seven pairs of Darlington arrays with common emitter. The IC consists of 16 pins in which 7 are input pins, 7 are output pins and remaining are VCC and Ground. The first four input pins are connected to the microcontroller. In the same way, four output pins are connected to the stepper motor.
- ▶ Stepper motor has 6 pins. In these six pins, 2 pins are connected to the supply of 12V and the remaining are connected to the output of the stepper motor. Stepper rotates at a given step angle. Each step in rotation is a fraction of full cycle. This depends on the mechanical parts and the driving method.
- ▶ Similar to all the motors, stepper motors will have stator and rotor. Rotor has permanent magnet and stator has coil. The basic stepper motor has 4 coils with 90 degrees rotation step. These four coils are activated in the cyclic order. The below figure shows you the direction of rotation of the shaft. There are different methods to drive a stepper motor. Some of these are explained below.
- ▶ **Full Step Drive:** In this method two coils are energized at a time. Thus, here two opposite coils are excited at a time.
- ▶ **Half Step Drive:** In this method coils are energized alternatively. Thus it rotates with half step angle. In this method, two coils can be energized at a time or single coil can be energized. Thus it increases the number of rotations per cycle. It is shown in the below figure.
- ▶ Initially , switch on the circuit.
- ▶ Microcontroller starts driving the stepper motor.
- ▶ One can observe the rotation of the stepper motor
- ▶ The stepper motor has four wires. They are yellow, blue, red and white. These are energized alternatively as given below.
- ▶ In full step driving, use the following sequence

| Yellow | Blue | Red | White |
|--------|------|-----|-------|
| 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |

- ▶ To drive the motor in half step angle, use the following sequence

| Yellow | Blue | Red | White |
|--------|------|-----|-------|
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 |

- ▶ **Stepper Motor Controller Circuit Advantages:**

- ▶ It consumes less power.
- ▶ It requires low operating voltage

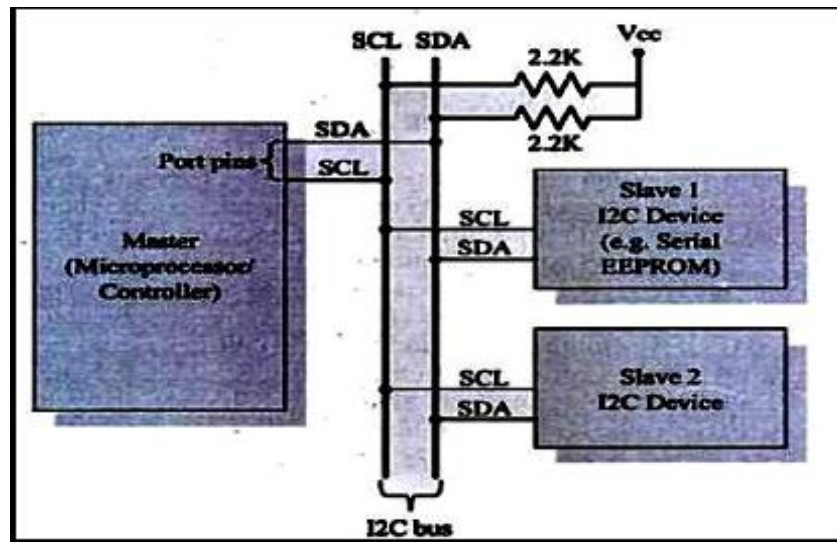
- ▶ **Stepper Motor Control Applications:**

- ▶ This circuit can be used in the robotic applications.
- ▶ This can also be used in mechatronics applications.
- ▶ The stepper motors can be used in disk drives, matrix printers, etc

What is Communication Interface? Explain different types of communication interfaces used in Embedded system?

Ans) **COMMUNICATION INTERFACE**

- ▶ Communication interface is essential for communicating with various subsystems of the embedded system and with the external world. For an embedded product, the communication interface can be viewed in two different perspectives; namely: Device/board level communication interface (onboard Communication Interface) and Product level communication interface (External Communication Interface). Embedded product is a combination of different types of components (chips/devices) arranged on a printed circuit board (PCB). The communication channel which interconnects the various components within an embedded product is referred to as device board level communication interface (onboard communication interface). Serial interfaces like I2C, SPI, UART, I²-Wire, etc. and parallel bus interface are examples of 'Onboard 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 a wired media or a wireless media and it can be a serial or a parallel interface. Infrared (IR), Bluetooth (BT), Wireless LAN (Wi-Fi), Radio Frequency waves (RF), GPRS, etc. are examples for wireless communication interface. RS-232C/RS-422/RS-485, USB, Ethernet IEEE 1394 port, Parallel port, CF-II interface, SDIO, PCMCIA, etc. are examples for wired interfaces. It is not mandatory that an embedded system should contain an external communication interface. Mobile communication equipment is an example for embedded system with external communication interface.
- ▶ **Onboard Communication Interfaces:**
- ▶ Onboard Communication Interface refers to the different communication channels/buses for interconnecting the various integrated circuits and other peripherals within the embedded system.
- ▶ **Inter Integrated Circuit (I2C) Bus:**
- ▶ The 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 1980s. 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 comprises of two bus lines, namely; Serial Clock—SCL and Serial Data—SDA. SCL line is responsible for generating synchronisation 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 synchronisation 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 synchronisation clock signal is generated by the 'Master' device only. I2C supports multi masters on the same bus.
- ▶ The sequence of operations for communicating with an I2C slave device is listed below:
 - ▶ 1. The master device pulls the clock line (SCL) of the bus to 'HIGH'
 - ▶ 2. The 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. The master device sends the address (7 bit or 10 bit wide) of the 'slave' device to which it wants to communicate, over the SDA line. Clock pulses are generated at the SCL line for synchronising the bit reception by the slave device. The MSB of the data is always transmitted first. The data in the bus is valid during the 'HIGH' period of the clock signal.
 - ▶ 4. The master device sends the Read or Write bit (Bit value = 1 Read operation; Bit value = 0 Write operation) according to the requirement

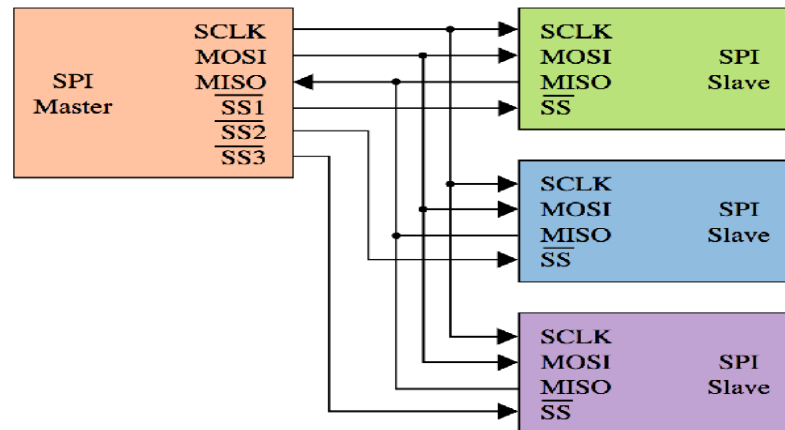
- ▶ 5. The master device waits for the acknowledgement bit from the slave device whose address is sent on the bus along with the Read Write operation command. Slave devices connected to the bus compares the address received with the address assigned to them.
- ▶ 6. The slave device with the address requested by the master device responds by sending an acknowledge bit (Bit value = 1) over the SDA line
- ▶ 7. Upon receiving the acknowledge bit, the Master device sends the 8bit data to the slave device over SDA line, if the requested operation is 'Write to device'. If the requested operation is 'Read from device', the slave device sends data to the master over the SDA line
- ▶ 8. The master device 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
- ▶ 9. The master device terminates the transfer by pulling the SDA line 'HIGH' when the clock line SCL is at logic 'HIGH'
- ▶ I2C bus supports three different data rates. They are: Standard mode (Data rate up to 100kbts/sec (100 kbps)), Fast mode (Data rate up to 400kbts/sec (400 kbps)) and High speed mode (Data rate up to 3.4Mbts/sec (3.4 Mbps)). The first generation I2C devices were designed to support data rates only up to 100kbps. The new generation I2C devices are designed to operate at data rates up to 3.4Mbts/sec.

▶ **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 was 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 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. It selects the required slave device by asserting the corresponding slave device's 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 a certain set of registers for holding these configurations.

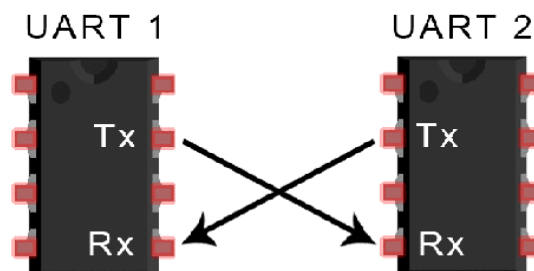
- ▶ The serial peripheral control register holds the various configuration parameters like master slave selection for the device. baud rate selection for communication. clock signal control, etc. The status register holds the status of various conditions for - transmission and reception.



- ▶ 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. In summary, the shift registers of 'master' and 'slave' devices form a circular buffer.
- ▶ For some devices, the decision on whether the LS/MS bit of data needs to be sent out first is configurable through configuration register (e.g. LSBF bit of the SPI control register for Motorola's 68HC12 controller).
- ▶ When compared to I2C. SPI bus is most suitable for applications requiring transfer of data in 'streams'. The only limitation is SPI doesn't support an acknowledgement mechanism.
- ▶ **Universal Asynchronous Receiver Transmitter (UART)**
- ▶ Universal Asynchronous Receiver Transmitter (UART) based data transmission is an asynchronous form of serial data transmission. UART based serial data transmission

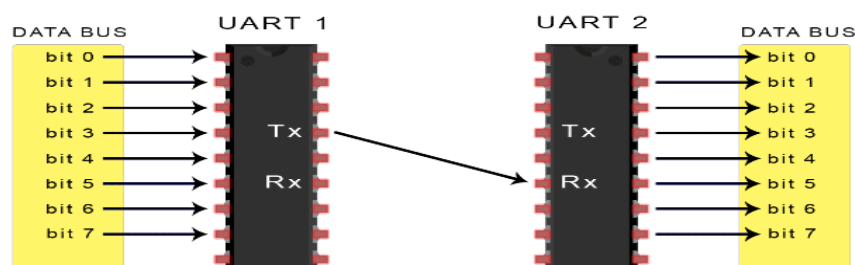
doesn't require a clock signal to synchronise the transmitting end and receiving end for transmission. Instead it relies upon the **predefined agreement** between the transmitting device and receiving device. The serial communication settings (**Baud rate, number of bits per byte, parity, number of start bits and stop bit and flow control**) for both transmitter and receiver should be set as identical.

The transmitting UART converts parallel data from a controlling device like a CPU into serial form, transmits it in serial to the receiving UART, which then converts the serial data back into parallel data for the receiving device. Only two wires are needed to transmit data between two UARTs. Data flows from the Tx pin of the transmitting UART to the Rx pin of the receiving UART:

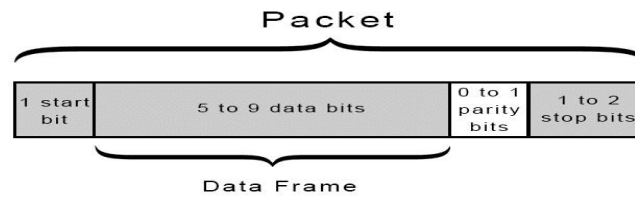


► How UART Works

- The UART that is going to transmit data receives the data from a data bus. The data bus is used to send data to the UART by another device like a CPU, memory, or microcontroller. Data is transferred from the data bus to the transmitting UART in parallel form. After the transmitting UART gets the parallel data from the data bus, it adds a start bit, a parity bit, and a stop bit, creating the data packet. Next, the data packet is output serially, bit by bit at the Tx pin.
- The receiving UART reads the data packet bit by bit at its Rx pin. The receiving UART then converts the data back into parallel form and removes the start bit, parity bit, and stop bits. Finally, the receiving UART transfers the data packet in parallel to the data bus on the receiving end:



- UART transmitted data is organized into *packets*. Each packet contains 1 start bit, 5 to 9 data bits (depending on the UART), an optional *parity* bit, and 1 or 2 stop bits:



► **Start Bit**

- The UART data transmission line is normally held at a high voltage level when it's not transmitting data. To start the transfer of data, the transmitting UART pulls the transmission line from high to low for one clock cycle. When the receiving UART detects the high to low voltage transition, it begins reading the bits in the data frame at the frequency of the baud rate.

► **Data Frame**

- The data frame contains the actual data being transferred. It can be 5 bits up to 8 bits long if a parity bit is used. If no parity bit is used, the data frame can be 9 bits long. In most cases, the data is sent with the least significant bit first.

► **Parity**

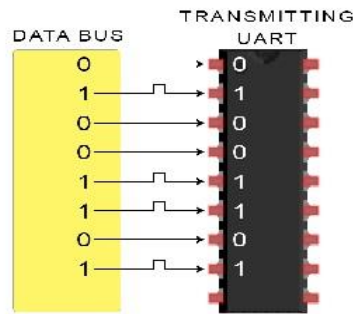
- Parity describes the evenness or oddness of a number. The parity bit is a way for the receiving UART to tell if any data has changed during transmission. Bits can be changed by electromagnetic radiation, mismatched baud rates, or long distance data transfers. After the receiving UART reads the data frame, it counts the number of bits with a value of 1 and checks if the total is an even or odd number. If the parity bit is a 0 (even parity), the 1 bits in the data frame should total to an even number. If the parity bit is a 1 (odd parity), the 1 bits in the data frame should total to an odd number. When the parity bit matches the data, the UART knows that the transmission was free of errors. But if the parity bit is a 0, and the total is odd; or the parity bit is a 1, and the total is even, the UART knows that bits in the data frame have changed.

► **Stop Bits**

- To signal the end of the data packet, the sending UART drives the data transmission line from a low voltage to a high voltage for at least two bit durations.

► **Steps of UART Transmission**

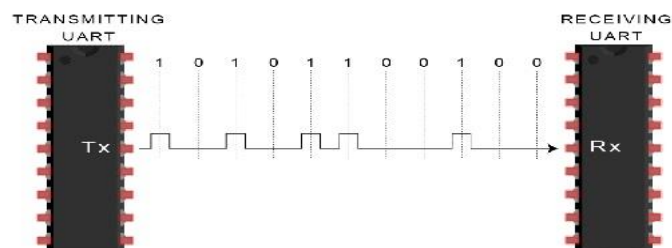
- 1. The transmitting UART receives data in parallel from the data bus:



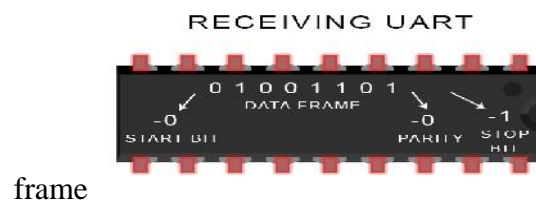
- 2. The transmitting UART adds the start bit, parity bit, and the stop bit(s) to the data frame:



- 3. The entire packet is sent serially from the transmitting UART to the receiving UART. The receiving UART samples the data line at the pre-configured baud rate



The receiving UART discards the start bit, parity bit, and stop bit from the data



- 5. The receiving UART converts the serial data back into parallel and transfers it to the data bus on the receiving end:



► **Advantages**

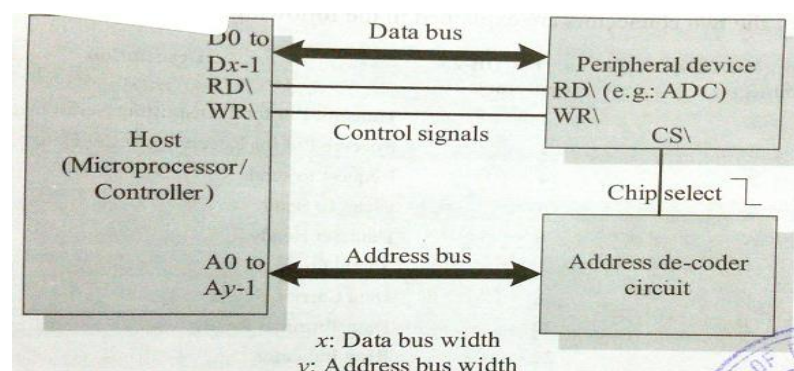
- Only uses two wires
- No clock signal is necessary
- Has a parity bit to allow for error checking
- The structure of the data packet can be changed as long as both sides are set up for it
- widely used method

► **Disadvantages**

- The size of the data frame is limited to a maximum of 9 bits
- Doesn't support multiple slave or multiple master systems
- The baud rates of each UART must be within 10% of each other

PARALLEL INTERFACE BUS:

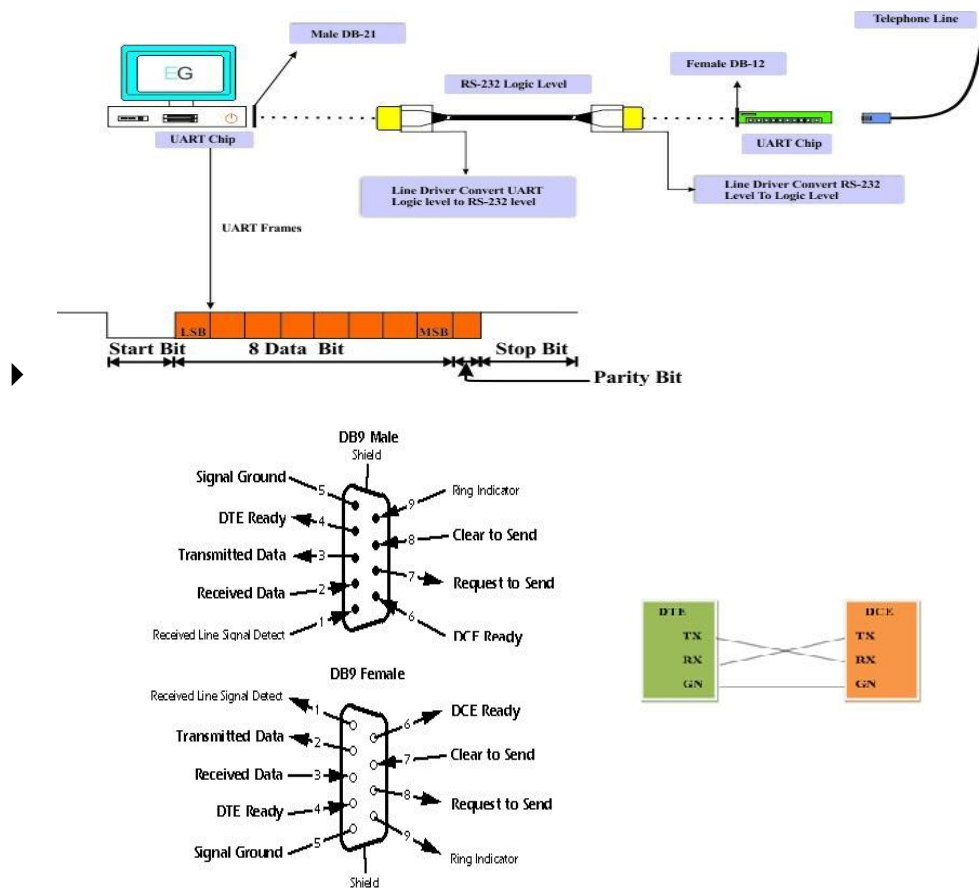
- The on-board parallel interface is normally used for communicating with peripheral devices which are memory mapped to the host of the system. The host processor/controller of the embedded system contains a parallel bus and the device which supports parallel bus can directly connect to this bus system. The communication through the parallel bus is controlled by the control signal interface between the device and the host. The 'Control Signals' for communication includes 'Read! Write' signal and device select signal. The device normally contains a device select line and the device becomes active only when this line is asserted by the host processor.
- The direction of data transfer (Host to Device or Device to Host) can be controlled through the control signal lines for 'Read' and 'Write'. Only the host processor has control over the 'Read' and 'Write' control signals. The device is normally memory mapped to the host processor and a range of address is assigned to it. An address decoder circuit is used for generating the chip select signal for the device. When the address selected by the processor is within the range assigned for the device, the decoder circuit activates the chip select line and thereby the device becomes active.



- ▶ The processor then can read or write from or to the device by asserting the corresponding control line (RD and WR respectively). Strict timing characteristics are followed for parallel communication. As mentioned earlier, parallel communication is host processor initiated. If a device wants to initiate the communication, it can inform the same to the processor through interrupts. For this, the interrupt line of the device is connected to the interrupt line of the processor and the corresponding interrupt is enabled in the host processor. The width of the parallel interface is determined by the data bus width of the host processor. It can be 4bit, 8bit, 16bit, 32bit or 64bit etc. The bus width supported by the device should be the same as that of the host processor.
- ▶ Parallel data communication offers the highest speed for data transfer.


External Communication Interface :

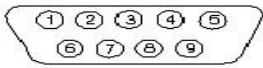
- ▶ It is responsible for data transfer between the embedded system and other devices or modules.
- ▶ The external communication interface can be either a wired media or a wireless media and it can be a serial or a parallel interface.
- ▶ Infrared (IR). Bluetooth (BT). Wireless LAN (Wi-Fi), Radio Frequency waves (RF), GPRS. etc. are examples for wireless communication interface.
- ▶ RS-232C/RS-422/RS-485, USB, Ethernet IEEE 1394 port. Parallel port, CF -II interface, SDIO, PCMCIA, etc. are examples for wired interfaces.
- ▶ **RS-232 C & RS- 485:**
- ▶ RS-232 stands for Recommend Standard number 232 and C is the latest revision of the standard.
- ▶ RS-232 is a standard communication protocol for linking computer and its peripheral devices to allow serial data exchange. In simple terms RS232 defines the voltage for the path used for data exchange between the devices.
- ▶ **NEED OF RS232:**
- ▶ As the technology was growing many electronic devices were being developed during this time like computers, printers, test instrument etc. There came a time where manufacturers felt the need to exchange information between these electronic devices. For example data exchange between a computer and a printer or two computers. But there was no standard or method to accomplish this task.
- ▶ **RS232** was the only available standard at the time which was used for data exchange. So, they thought of adopting this standard in electronic devices for digital data exchange. But the standard was unable to fulfil the requirements as it was developed specifically for modem and teletypewriter. To overcome this problem, designers started implementing an RS232 interface compatible to their equipments.



► DCE and DTE Devices

- Two terms you should be familiar with are DTE and DCE. DTE stands for Data Terminal Equipment, and DCE stands for Data Communications Equipment. These terms are used to indicate the pin-out for the connectors on a device and the direction of the signals on the pins. Your computer is a DTE device, while most other devices are usually DCE devices.
- If you have trouble keeping the two straight then replace the term "DTE device" with "your PC" and the term "DCE device" with "remote device" in the following discussion.
- The RS-232 standard states that DTE devices use a 25-pin male connector, and DCE devices use a 25-pin female connector. You can therefore connect a DTE device to a DCE using a straight pin-for-pin connection. However, to connect two like devices, you must instead use a null modem cable. Null modem cables cross the transmit and receive lines in the cable, and are discussed later in this chapter. The listing below shows the connections and signal directions for both 25 and 9-pin connectors.

| 25 Pin Connector on a DTE device (PC connection) | |
|--|---|
| Male RS232 DB25 |  |
| Pin Number | Direction of signal: |
| 1 | Protective Ground |
| 2 | Transmitted Data (TD) Outgoing Data (from a DTE to a DCE) |
| 3 | Received Data (RD) Incoming Data (from a DCE to a DTE) |
| 4 | Request To Send (RTS) Outgoing flow control signal controlled by DTE |
| 5 | Clear To Send (CTS) Incoming flow control signal controlled by DCE |
| 6 | Data Set Ready (DSR) Incoming handshaking signal controlled by DCE |
| 7 | Signal Ground Common reference voltage |
| 8 | Carrier Detect (CD) Incoming signal from a modem |
| 20 | Data Terminal Ready (DTR) Outgoing handshaking signal controlled by DTE |
| 22 | Ring Indicator (RI) Incoming signal from a modem |

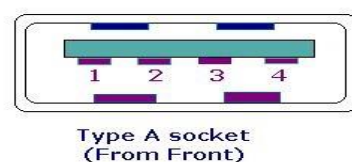
| 9 Pin Connector on a DTE device (PC connection) | |
|---|---|
| Male RS232 DB9 |  |
| Pin Number | Direction of signal: |
| 1 | Carrier Detect (CD) (from DCE) Incoming signal from a modem |
| 2 | Received Data (RD) Incoming Data from a DCE |
| 3 | Transmitted Data (TD) Outgoing Data to a DCE |
| 4 | Data Terminal Ready (DTR) Outgoing handshaking signal |
| 5 | Signal Ground Common reference voltage |
| 6 | Data Set Ready (DSR) Incoming handshaking signal |
| 7 | Request To Send (RTS) Outgoing flow control signal |
| 8 | Clear To Send (CTS) Incoming flow control signal |
| 9 | Ring Indicator (RI) (from DCE) Incoming signal from a modem |

- ▶ The **TD (transmit data)** wire is the one through which data from a DTE device is transmitted to a DCE device. This name can be deceiving, because this wire is used by a DCE device to receive its data. The TD line is kept in a mark condition by the DTE device when it is idle. The RD (receive data) wire is the one on which data is received by a DTE device, and the DCE device keeps this line in a mark condition when idle.
- ▶ **RTS** stands for **Request To Send**. This line and the CTS line are used when "hardware flow control" is enabled in both the DTE and DCE devices. The DTE device puts this line in a mark condition to tell the remote device that it is ready and able to receive data. If the DTE device is not able to receive data (typically because its receive buffer is almost full), it will put this line in the space condition as a signal to the DCE to stop sending data. When the DTE device is ready to receive more data (i.e. after data has been removed from its receive buffer), it will place this line back in the mark condition.
- ▶ The complement of the RTS wire is CTS, which stands for Clear To Send. The DCE device puts this line in a mark condition to tell the DTE device that it is ready to receive the data. Likewise, if the DCE device is unable to receive data, it will place this line in the space condition. Together, these two lines make up what is called RTS/CTS or "hardware" flow control.
- ▶ **DTR** stands for **Data Terminal Ready**. Its intended function is very similar to the RTS line. DSR (Data Set Ready) is the companion to DTR in the same way that CTS is to RTS. Some serial devices use DTR and DSR as signals to simply confirm that a device is connected and is turned on.
- ▶ **CD** stands for **Carrier Detect**. Carrier Detect is used by a modem to signal that it has made a connection with another modem, or has detected a carrier tone. The last remaining line is **RI** or **Ring Indicator**. A modem toggles the state of this line when an incoming call rings your phone.

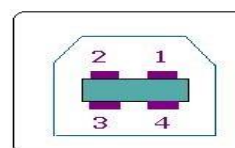
Universal Serial Bus (USB) is a plug and play interface that allows a computer to communicate with peripheral and other devices. USB-connected devices cover a broad range; anything from keyboards and mice, to music players and flash drives. The USB was originally developed in 1995 by many of the industry leading companies like Intel, Compaq, Microsoft, Digital, IBM, and Northern Telecom.

- ▶ USB offers users simple connectivity. It eliminates the mix of different connectors for different devices like printers, keyboards, mice, and other peripherals. That means USB-bus allows many peripherals to be connected using a single standardized interface socket. Another main advantage is that, in USB environment, DIP-switches are not necessary for setting peripheral addresses and IRQs. It supports all kinds of data, from slow mouse inputs to digitized audio and compressed video.

- ▶ USB also allows hot swapping. The "**hot-swapping**" means that the devices can be plugged and unplugged without rebooting the computer or turning off the device. That means, when plugged in, everything configures automatically. So the user needs not worry about terminations, terms such as IRQs and port addresses, or rebooting the computer. Once the user is finished, they can simply unplug the cable out, the host will detect its absence and automatically unload the driver. This makes the USB a plug-and-play interface between a computer and add-on devices.
- ▶ The USB has already replaced the RS232 and other old parallel communications in many applications
- ▶ USB sends data in serial mode i.e. the parallel data is serialized before sends and de-serialized after receiving.
- ▶ The benefits of USB are low cost, expandability, auto-configuration, hot-plugging and outstanding performance. It also provides power to the bus, enabling many peripherals to operate without the added need for an AC power adapter.
- ▶ The USB standard specifies two kinds of cables and connectors. The USB cable will usually have an "A" connector on one end and a "B" on the other. That means the USB devices will have an "A" connection on it. If not, then the device has a socket on it that accepts a USB "B" connector.
- ▶ The USB standard uses "A" and "B" connectors mainly to avoid confusion:
 1. "A" connectors head "upstream" toward the computer.
 2. "B" connectors head "downstream" and connect to individual devices.
- ▶ By using different connectors on the upstream and downstream end, it is impossible to install a cable incorrectly, because the two types are physically different.
- ▶ Inside the USB cable there are two wires that supply the power to the peripherals-- +5 volts (red) and ground (brown)-- and a twisted pair (yellow and blue) of wires to carry the data. On the power wires, the computer can supply up to 500 milliamps of power at 5 volts. A peripheral that draws up to 100ma can extract all of its power from the bus wiring all of the time. If the device needs more than a half-amp, then it must have its own power supply. That means low-power devices such as mice can draw their power directly from the bus. High-power devices such as printers have their own power supplies and draw minimal power from the bus. Hubs can have their own power supplies to provide power to devices connected to the hub.



Type A socket
(From Front)



Type B socket
(From front)

USB SOCKETS & PINS

- ▶ USB supports four different types of data transfers, namely; Control, Bulk, Isochronous and Interrupt. **Control transfer** is used by USB system software to query, configure and issue commands to the USB device.
- ▶ **Bulk transfer** is used for sending a block of data to a device. Bulk transfer supports error checking and correction. Transferring data to a printer is an example for bulk transfer.
- ▶ **Isochronous data transfer** is used for real-time data communication. In Isochronous transfer, data is transmitted as streams in real-time. Isochronous transfer doesn't support error checking and re-transmission of data in case of any transmission loss. All streaming devices like audio devices and medical equipment for data collection make use of the isochronous transfer.
- ▶ **Interrupt transfer** is used for transferring small amount of data. Interrupt transfer mechanism makes use of polling technique to see whether the USB device has any data to send. The frequency of polling is determined by the USB device and it varies from 1 to 255 milliseconds. Devices like Mouse and Keyboard, which transmits fewer amounts of data, uses Interrupt transfer.
- ▶ Presently USB supports four different data rates namely; Low Speed , Speed (12Mbps), High Speed (480Mbps) and Super Speed (4.8Gbps)

IEEE 1394 (High Performance Serial Bus, FireWire):

- ▶ Originally created by Apple and standardized in 1995 as the specification **IEEE 1394 High Performance Serial Bus**, FireWire is very similar to USB. The designers of FireWire, which actually precedes the development of USB, had several particular goals in mind when they created the standard:
 - ▶ Fast transfer of data (up to 400 Mbps)
 - ▶ Lots of devices on the bus
 - ▶ Ease of use
 - ▶ Hot pluggable
 - ▶ Provide power through the cable
 - ▶ Plug-and-play
 - ▶ Low cabling cost
 - ▶ Low implementation cost
- ▶ When the host computer powers up, it queries all of the devices connected to the bus and assigns each one an address, a process called **enumeration**. FireWire is **plug-and-play**, so if a new FireWire device is connected to a computer, the operating

system auto-detects it and asks for the driver disk. If the device has already been installed, the computer activates it and starts talking to it. FireWire devices are **hot pluggable**, which means they can be connected and disconnected at any time, even with the power on.

| Feature | Firewire | USB |
|--------------------------|--|---|
| Data Transfer Rate | 400 Mbps | 480Mbps |
| Number of devices | 63 | 127 |
| Plug and play | Yes | Yes |
| Hot pluggable | Yes | Yes |
| Isochronous devices | Yes | Yes |
| Bus power | Yes | Yes |
| Bus termination required | No | No |
| Bus type | Serial | Serial |
| Cable type | Twisted pair (6 wires: 2 power, 2 twisted pair sets) | Twisted pair (4 wires: 2 power, 1 twisted pair set) |
| Networkable | Yes | Yes |
| Network topology | Daisy chain | Hub |

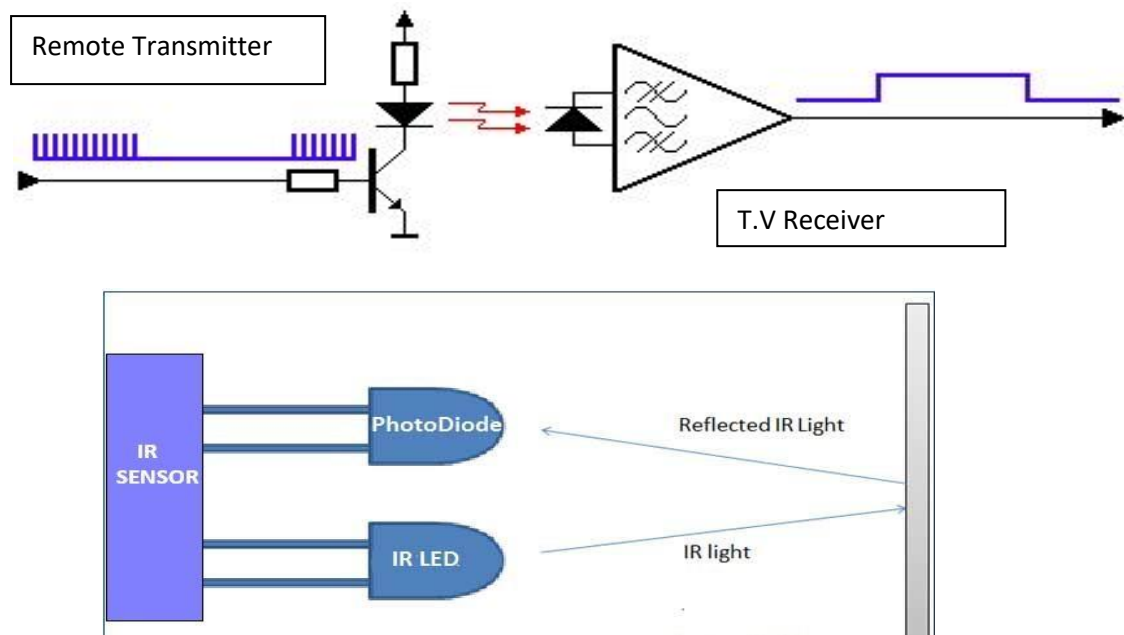
External Communication Interface(wireless)

- ▶ **Infrared (IrDA)** :It is a serial, half duplex, line of sight based wireless technology for data communication between devices.
- ▶ The remote control of your TV, VCD player, etc. works on Infrared data communication principle.
- ▶ Infrared communication technique uses infrared waves of the electromagnetic spectrum for transmitting the data. IrDA supports point to point and point-to-multipoint communication, provided all devices involved in the communication are within the line of sight.
- ▶ The typical communication range for IRDA lies in the range 10 cm to 1 m. The range can be increased by increasing the transmitting power of the IR device.
- ▶ Depending on the speed of data transmission IR is classified into Serial IR (SIR), Medium IR (MIR). Fast IR (FIR), Very Fast IR (VFIR) and Ultra Fast IR (UFIR).
- ▶ SIR supports transmission rates ranging from 9600bps to 115.1 kbps. MW supports data rates of 0.576Mbps and 1.152Mbps.

FIR supports data rates up to 4Mbps. VF1R is designed to support high data rates up to 16Mbps.

- ▶ The UFIR supports data rate up to 100Mbps.

- ▶ IrDA communication involves a transmitter unit for transmitting the data over IR and a receiver for receiving the data.
- ▶ Infrared Light Emitting Diode (LED) is the IR source for transmitter and at the receiving end a photodiode acts as the receiver.
- ▶ Both transmitter and receiver unit will be present in each device supporting IrDA communication for bidirectional data transfer. Such IR units are known as 'Transceiver'.
- ▶ Certain devices like a TV remote control always require unidirectional communication and so they contain either the transmitter or receiver unit (The remote control unit contains the transmitter unit and TV contains the receiver unit).
- ▶ IrDA communication has two essential parts: a physical link part and a protocol part. The physical link is responsible for the physical transmission of data between devices supporting IR communication and protocol part is responsible for defining the rules of communication. The physical link works on the wireless principle making use of Infrared for communication
- ▶ **Applications of IR Communication:**
- ▶ **IR sensor:**
- ▶ An IR sensor is a device that detects IR radiation falling on it. Proximity sensors (used in touch screen phones and edge avoiding robots), contrast sensors (used in line following robots) and obstruction counters/sensors (used for counting goods and in burglar alarms) are some applications involving IR sensors.
- ▶ **Principle of Working**
- ▶ An IR sensor consists of two parts, the emitter circuit and the receiver circuit. This is collectively known as a photo-coupler or an optocoupler.
- ▶ The emitter is an IR LED and the detector is an IR photodiode. The IR photodiode is sensitive to the IR light emitted by an IR LED. The photodiode's resistance and output voltage change in proportion to the IR light received. This is the underlying working principle of the IR sensor.
- ▶ The type of incidence can be direct incidence or indirect incidence. In direct incidence, the IR LED is placed in front of a photodiode with no obstacle in between. In indirect incidence, both the diodes are placed side by side with an opaque object in front of the sensor. The light from the IR LED hits the opaque surface and reflects back to the photodiode.



► **Line Follower Robots**

- In line following robots, IR sensors detect the colour of the surface underneath it and send a signal to the microcontroller or the main circuit which then takes decisions according to the algorithm set by the creator of the bot. Line followers employ reflective or non-reflective indirect incidence. The IR is reflected back to the module from the white surface around the black line. But IR radiation is absorbed completely by black colour. There is no reflection of the IR radiation going back to the sensor module in black colour.

► **Item Counter**

- Item counter is implemented on the basis of direct incidence of radiation on the photodiode. Whenever an item obstructs the invisible line of IR radiation, the value of a stored variable in a computer/microcontroller is incremented. This is indicated by LEDs, seven segment displays and LCDs. Monitoring systems of large factories use these counters for counting products on conveyor belts.

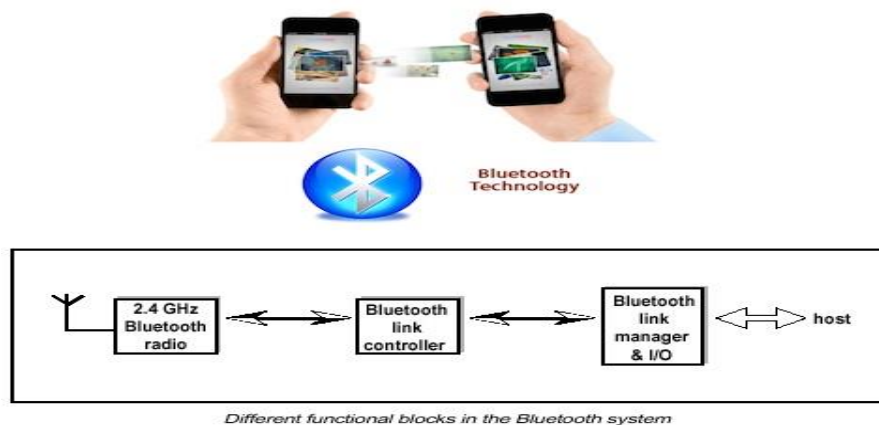
Bluetooth:

- Bluetooth is a low cost, low power, short range wireless technology for data and voice communication. Bluetooth was first proposed by 'Ericsson' in 1994.
- Bluetooth operates at 2.4GHz of the Radio Frequency spectrum .This frequency band has been set aside by international agreement for the use of industrial, scientific and medical devices (ISM) and uses the Frequency Hopping Spread Spectrum (FHSS) technique for communication.
- In this technique, a device will use 79 individual, randomly chosen frequencies within a designated range, changing from one to another on a regular basis. In the case of

Bluetooth, the transmitters change frequencies 1,600 times every second, meaning that more devices can make full use of a limited slice of the radio spectrum. Since every Bluetooth transmitter uses spread-spectrum transmitting automatically, it's unlikely that two transmitters will be on the same frequency at the same time.

- ▶ Literally it supports a data rate of up to 1Mbps and a range of approximately 30 feet or 10m for data communication.

Like IrDA, Bluetooth communication also has two essential parts: a physical link part and a protocol part. The physical link is responsible for the physical transmission of data between devices supporting Bluetooth communication and protocol part is responsible for defining the rules of communication.



- ▶ The physical link works on the wireless principle making use of RF waves for communication. Bluetooth enabled devices essentially contain a Bluetooth wireless radio for the transmission and reception of data. The rules governing the Bluetooth communication is implemented in the 'Bluetooth protocol stack'. The Bluetooth communication IC holds the stack.
- ▶ Bluetooth communication follows packet based data transfer. Bluetooth supports point-to-point (device to device) and point-to-multipoint (device to multiple device broadcasting) wireless communication.
- ▶ The point-to-point communication follows the master-slave relationship. A Bluetooth device can function as either master or slave. When a network is formed with one Bluetooth device as master and more than one device as slaves, it is called a Piconet.
- ▶ A Piconet supports a maximum of seven slave devices.
- ▶ Bluetooth is the favourite choice for short range data communication in handheld embedded devices.
- ▶ Bluetooth technology is very popular among cell phone users as they are the easiest communication channel for transferring ringtones, music files, pictures, media files, etc. between neighbouring Blue-tooth enabled phones.

► **Wi-Fi:**

- Wi-Fi is the popular wireless communication technique for networked communication of devices. Wi-Fi follows the IEEE 802.11 standard.
- Wi-Fi is intended for network communication and it supports Internet Protocol (IP) based communication. It is essential to have device identities in a multipoint communication to address specific devices for data communication.
- In an IP based communication each device is identified by an IP address, which is unique to each device on the network. 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. The hardware part of it is known as Wi-Fi Radio.
- Wi-Fi operates at 2.4GHz or 5GHz of radio spectrum and they co-exist with other ISM band devices like Bluetooth



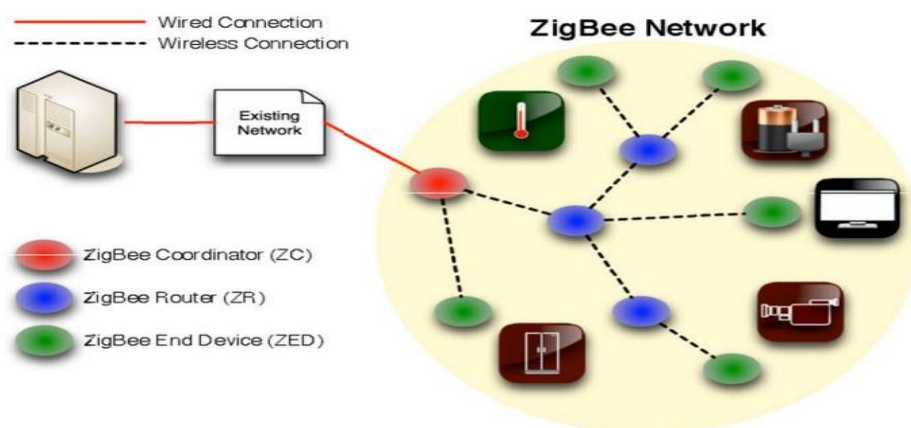
- For communicating with devices over a Wi-Fi network, the device when its Wi-Fi radio is turned ON, searches the available WiFi network in its vicinity and lists out the Service Set Identifier (SSID) of the available networks.
- If the network is security enabled, a password may be required to connect to a particular SSID. Wi-Fi employs different security mechanisms like Wired Equivalency Privacy (WEP) Wireless Protected Access (WPA), etc. for securing the data communication.
- Wi-Fi supports data rates ranging from 1Mbps to 150Mbps (Growing towards higher rates as technology Progresses) depending on the standards (802.11a/b/g/n) and access/modulation method.

- ▶ **802.11a** transmits at 5 GHz and can move up to 54 megabits of data per second. It also uses **orthogonal frequency-division multiplexing** (OFDM), a more efficient coding technique that splits that radio signal into several sub-signals before they reach a receiver. This greatly reduces interference.
- ▶ **802.11b** is the slowest and least expensive standard. For a while, its cost made it popular, but now it's becoming less common as faster standards become less expensive.
- ▶ 802.11b transmits in the 2.4 GHz frequency band of the radio spectrum. It can handle up to 11 megabits of data per second, and it uses **complementary code keying** (CCK) modulation to improve speeds.
- ▶ **802.11g** transmits at 2.4 GHz like 802.11b, but it's a lot faster -- it can handle up to 54 megabits of data per second. 802.11g is faster because it uses the same OFDM coding as 802.11a.
- ▶ **802.11n** is the most widely available of the standards and is backward compatible with a, b and g. It significantly improved speed and range over its predecessors. For instance, although 802.11g theoretically moves 54 megabits of data per second, it only achieves real-world speeds of about 24 megabits of data per second because of network congestion. 802.11n, however, reportedly can achieve speeds as high as 140 megabits per second. 802.11n can transmit up to four streams of data, each at a maximum of 150 megabits per second, but most routers only allow for two or three streams.
- ▶ **802.11ac** is the newest standard as of early 2013. It has yet to be widely adopted, and is still in draft form at the **Institute of Electrical and Electronics Engineers (IEEE)**, but devices that support it are already on the market.
- ▶ 802.11b transmits in the 2.4 GHz frequency band of the radio spectrum. It can handle up to 11 megabits of data per second, and it uses **complementary code keying** (CCK) modulation to improve speeds.
- ▶ **802.11g** transmits at 2.4 GHz like 802.11b, but it's a lot faster -- it can handle up to 54 megabits of data per second. 802.11g is faster because it uses the same OFDM coding as 802.11a.
- ▶ **802.11n** is the most widely available of the standards and is backward compatible with a, b and g. It significantly improved speed and range over its predecessors. For instance, although 802.11g theoretically moves 54 megabits of data per second, it only achieves real-world speeds of about 24 megabits of data per second because of network congestion. 802.11n, however, reportedly can achieve speeds as high as 140 megabits per second. 802.11n can transmit up to four streams of data, each at a maximum of 150 megabits per second, but most routers only allow for two or three streams.

- ▶ **802.11ac** is the newest standard as of early 2013. It has yet to be widely adopted, and is still in draft form at the **Institute of Electrical and Electronics Engineers (IEEE)**, but devices that support it are already on the market.
- ▶ 802.11ac is backward compatible with 802.11n (and therefore the others, too), with n on the 2.4 GHz band and ac on the 5 GHz band. It is less prone to interference and far faster than its predecessors, pushing a maximum of 450 megabits per second on a single stream, although real-world speeds may be lower.
- ▶ Depending on the type of antenna and usage location (indoor/outdoor), Wi-Fi offers a range of 100 to 300 feet.

ZigBee

- ▶ ZigBee is a low power, low cost, wireless network communication protocol based on the IEEE 802.15.4-2006 standard.
- ▶ ZigBee is targeted for low power, low data rate and secure applications for Wireless Personal Area Networking (WPAN).
- ▶ The ZigBee specifications support a robust mesh network containing multiple nodes. This networking strategy makes the network reliable by permitting messages to travel through a number of different paths to get from one node to another.
- ▶ ZigBee operates worldwide at the unlicensed bands of Radio spectrum, mainly at 2.400 to 2.484 GHz, 902 to 928 MHz and Device 1868.0 to 868.6 MHz. ZigBee Supports an operating distance of up to 100 metres and a network data rate of 20 to 250Kbps.
- ▶ In the ZigBee terminology, each ZigBee device falls under any one of the following ZigBee device category.
- ▶ **ZigBee Coordinator (ZC)/Network Coordinator:** The ZigBee coordinator acts as the root of the Zig-Bee network. The ZC is responsible for initiating the ZigBee network and it has the capability to storing information about the network.



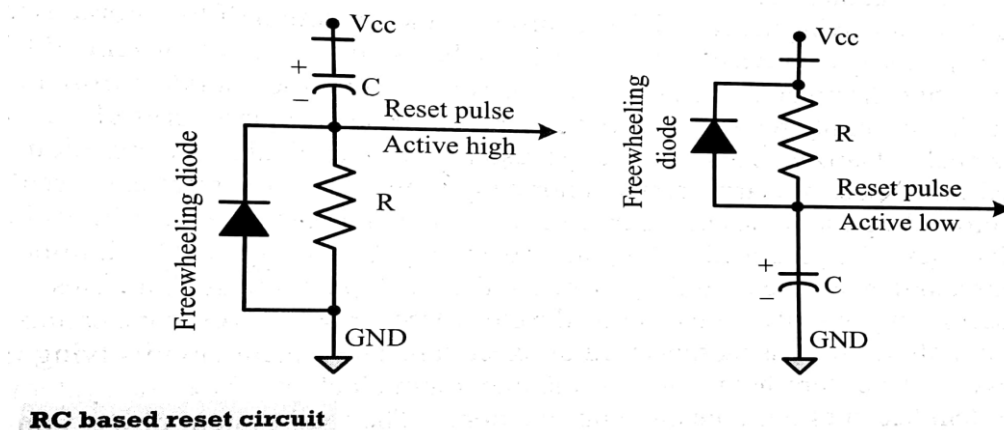
- ▶ **ZigBee Router (ZR) Full function Device (FFD):** Responsible for passing information from device to another device or to another ZR.
- ▶ **ZigBee End Device (ZED)/Reduced Function Device (RFD):** End device containing ZigBee functionality for data communication. It can talk only with ZR and ZC it doesn't have capability to act as mediator for transferring data from one device to another.
- ▶ ZigBee is targeting applications like home and industrial automation, energy management, home security, patient tracking, lighting control and environmental controls.

UNIT-III

What is a reset circuit? Explain

The reset circuit is essential for the proper functioning of processors/controllers of embedded systems. It ensures that the device is not operating at a voltage level where the device is not guaranteed to operate. 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 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). 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. Some processors/controllers contains built-in internal circuitry and they don't require external reset circuitry. Below figure shows the resistor capacitor based passive reset circuit for active high and low configurations. The reset pulse can be adjusted by changing the resistance value R and capacitance value C.



What is brown-out Protection circuit?

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 required threshold.

The processor behaviour may not be predictable if the supply voltage falls below the recommended operating voltage. It may lead to situations like data corruption. 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 built-in brown -out protection circuit, the same can be implemented using external passive circuits or supervisor ICs. Figure 2.30 illustrates brown-out protection circuit implementation using Zener diode and transistor for processor/controller with active low Reset logic.

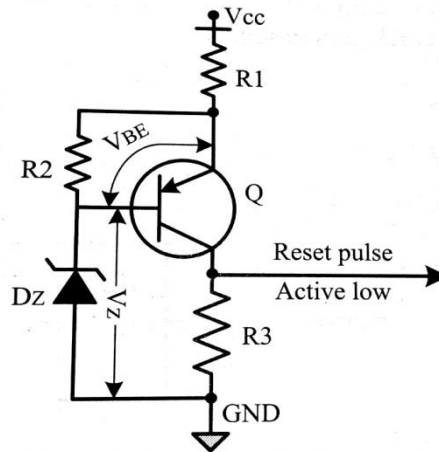


Fig. 2.36 Brown-out protection circuit with Active low output

The Zener diode Dz 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 . 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 , R_3 can be selected based on the electrical characteristics

What is an oscillator circuit?

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. 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.

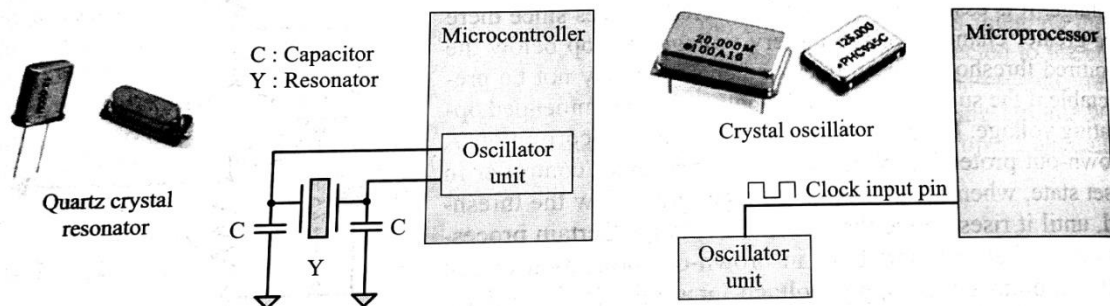


Fig. 2.37 Oscillator circuitry using quartz crystal and quartz crystal oscillator

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.

What is Real Time Clock?

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.

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 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.

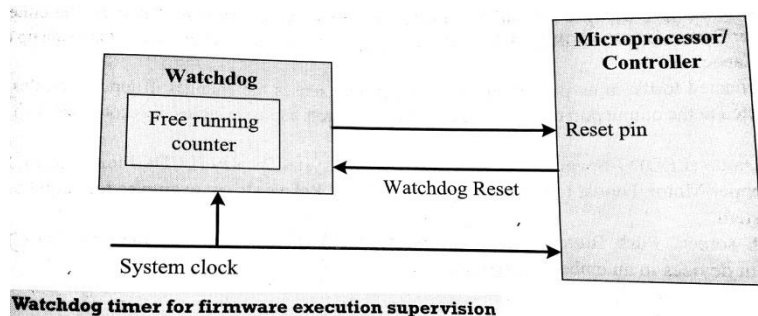
What is Watchdog timer? How it is helpful for an Embedded system?

Watchdog Timer

A watchdog timer, or simply a watchdog, is a hardware timer for monitoring the firm are 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.

Explain various approaches used for Embedded firmware design?

EMBEDDED FIRMWARE DESIGN APPROACHES

The firmware design approaches for embedded product is purely dependent on the complexity of the functions to be performed, the speed of operation required, etc. Two basic approaches are used for Embedded firmware design. They are

- 1) The Super Loop Based Approach
- 2) The embedded operating systems based approach

The Super Loop Based Approach

The Super Loop based firmware development approach is adopted 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 task listed at the top of the program code is executed first and the tasks just below the top are executed after completing the first task. This is a true procedural one. In a multiple task based system, each task is executed in serial in this approach. The firmware execution flow for this will be

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

From the firmware execution sequence, it is obvious that the order in which the tasks to be executed are fixed and they are hard coded in the code itself. Also the operation is an infinite loop based approach. We can visualise the operational sequence listed above in terms of a 'C' program code as

```
void main( )  
{  
Configurations ();  
Initializations ();  
while (1)  
{  
Task 1 ();  
Task 2 ();  
:  
:
```

```
Task n ( );  
}  
}
```

Almost all tasks in embedded applications are non-ending and are repeated infinitely throughout the operation. From the above 'C' code we can see that the tasks 1 to n are performed one after another and when the last task (nth task) is executed, the firmware execution is again re-directed to Task 1 and it is repeated forever in the loop. This repetition is achieved by using an infinite loop. This approach is also referred as 'Super loop based Approach'.

The 'Super loop based design' doesn't require an operating system, since there is no need for scheduling the task is to be executed and assigning priority to each task. This type of design is deployed in low-cost embedded products and products where response time is not time critical.

For example, reading/writing data to and from a card using a card reader requires a sequence of operations like checking the presence of card, authenticating the operation, reading/writing, etc. A typical example of a 'Super loop based' product is an electronic video game toy containing keypad and display unit. .

Another major drawback of the Super loop' design approach is the lack of real timeliness. If the number of tasks to be executed within an application increases, the time at which each task is repeated also increases. This brings the probability of missing out some events. .

The Embedded Operating System (OS) Based Approach

The Operating System (OS) based approach contains operating systems, which can be either a General Purpose Operating System (GPOS) or a Real Time Operating System (RTOS) to host the user written application firmware. The General Purpose OS (GPOS) based design is very similar to a conventional PC based application development where the device contains an operating system (Windows/Unix/Linux, etc. for Desktop PCs) and you will be creating and miming user applications on top of it.

Example of a GPOS used in embedded product development is Microsoft® Windows XP Embedded. Examples of Embedded products using Microsoft® Windows XP OS are Personal Digital Assistants(PDAs), Hand held devices/Portable devices and Point of Sale (PoS) terminals.

Real Time Operating System (RTOS) based design approach is employed in embedded products demanding Real-time response. RTOS respond in a timely and predictable manner to events. Real Time operating system contains a Real Time kernel responsible for performing pre-emptive multitasking, scheduler for scheduling tasks, multiple threads, etc. A Real Time Operating System (RTOS) all flexible scheduling of system resources like the CPU and memory and offers some way to communicate between tasks.

Embedded Linux' 'Symbian', 'Windows CE', 'pSOS', 'VxWorks' 'ThreadX'. MicroC/OS11 etc are examples of RTOS employed in embedded product development. Mobile phones, PDAs (Based on Windows CE/Windows Mobile Platforms), handheld devices, etc. are examples of Embedded Products based on RTOS. Most of the mobile phones are built around the popular RTOS 'Symbian'.

Explain various firmware development languages?

EMBEDDED FIRMWARE DEVELOPMENT LANGUAGES

We can use either a target processor controller specific language (Generally known as Assembly language or low level language) or a target processor controller independent language (Like C, C++, JAVA. etc. commonly known as High Level Language) or a combination of Assembly and High level Language. We will discuss where each of the approach is used and the relative merits and de -merits of each, in the following sections.

ASSEMBLY LANGUAGE BASED DEVELOPMENT

'Assembly language' is the human readable notation of 'machine language' whereas 'machine language' is a processor understandable language. Processors deal only with binaries 0s and 1s). Machine language is a binary representation and it consists of 1s and 0s. Machine language is made readable by using specific symbols called 'mnemonics'. Hence machine language can be considered as an interface between processor and programmer.

Assembly language and machine languages are processor controller dependent and an assembly program written for one processor, controller family will not work with others. Assembly language programming is the task of writing processor specific machine code in mnemonic form, converting the mnemonics into actual processor instructions 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. We will analyse each of them with the 8051 ASM instructions as an example.

MOV A, #30

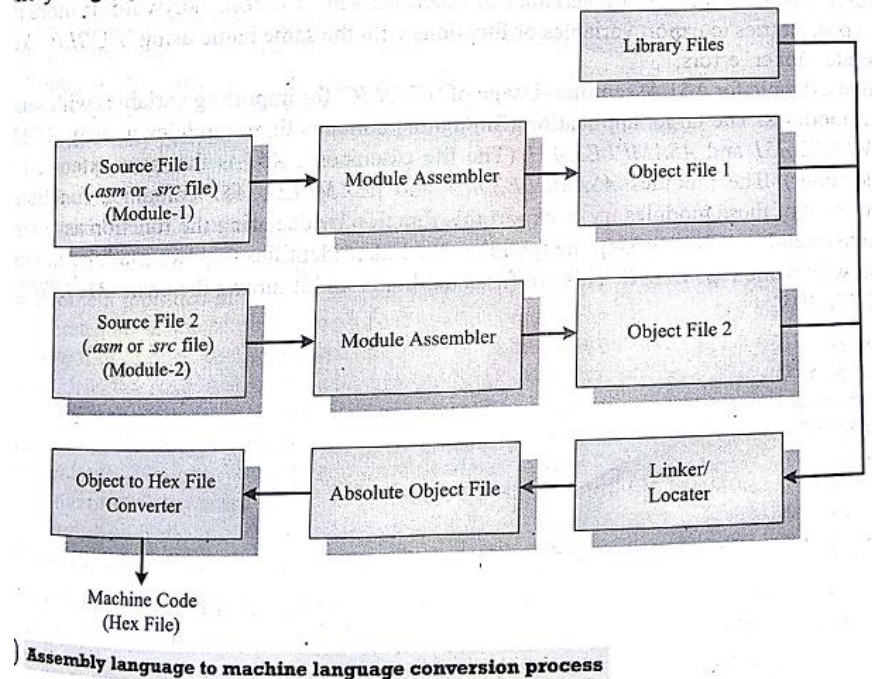
This instruction mnemonic moves decimal value 30 to the 8051 Accumulator register Here MOV A is the Opcode and 30 is the operand (single operand). The same instruction when written in machine language will look like

01110100 00011110

Where the first 8 bit binary value 01110100 represents the opcode. MOVA and the second 8 bit binary value 00011110 represents the operand 30.

Similar to 'c' language in assembly level language also have multiple source files called modules. Each module is represented by '.asm' or '.src' file similar to '.c' files in C programming. This approach is known as 'Modular Programming'. Conversion of the

assembly level language to machine language is carried out by a sequence of operations.



SOURCE FILE TO OBJECT FILE TRANSLATION

Translation of assembly code to machine code is performed by assembler. A51 Macro Assembler from keil software is a popular assembler for the 8051 family microcontroller. The various steps involved in the conversion of a program written in assembly language to corresponding binary file/machine language is illustrated in above figure.

Each source module is written in Assembly and is stored as .src file or .asm file. Each file can be assembled separately to examine the syntax errors and incorrect assembly instructions. On successful assembling of each .src/.asm file a corresponding object file is created with extension '.obj'. The object file does not contain the absolute address of where the generated code needs to be placed on the program memory and hence it is called a re-locatable segment. It can be placed at any code memory location and it is the responsibility of the linker/locater to assign absolute address for this module.

LIBRARY FILE CREATION AND USAGE

Libraries are specially formatted, ordered program collections of object modules that may be used by the linker at a later time. When the linker processes a library, only those object modules in the library that are necessary to create the program are used. Library files are generated with extension '.lib', Library file is some kind of source code hiding technique.

For using a library file in a project, add the library to the project. 'LIB51' from keil Software is an example for a library creator and it is used for creating library files for A51 Assembler/C51 Compiler for 8051 specific controller.

LINKER AND LOCATER

Linker and Locater is another software utility responsible for "linking modules in a multi-module project and assigning absolute address to each module". Linker generates an absolute

object module by extracting the object modules from the library. The absolute object file is used for creating hex files for dumping into the code memory of the processor/controller.

'BL51' from Keil Software is an example for a Linker or locator for A51 Assembler/C51 Compiler for 8051 specific controller.

OBJECT TO HEX FILE CONVERTER

This is the final stage in the conversion of Assembly language (mnemonics) to machine understandable language (machine code). Hex File is the representation of the machine code and the hex file is dumped into the code memory of the processor/controller. The hex file representation varies depending on the target processor/controller make. For Intel Processors/controllers the target hex file format will be 'Intel HEX' and for Motorola, the hex file should be in 'Motorola HEX' format. HEX files are ASCII files that contain a hexadecimal representation of target application. Hex file is created from the final 'absolute object file' using the Object to Hex File Converter utility.

'OH51' from Keil software is an example for Object to Hex File Converter utility for A51 Assembler/C51 Compiler for 8051 specific controller.

ADVANTAGES OF ASSEMBLY LANGUAGE BASED DEVELOPMENT

Assembly Language based development is the most common technique adopted from the beginning of embedded technology development. The major advantages of Assembly Language based development is listed below.

Efficient Code Memory and Data Memory Usage (Memory Optimisation)

Since the developer is well versed with the target processor architecture and memory organisation, optimised code can be written for performing operations. This leads to less utilisation of code memory and efficient utilisation of data memory.

High Performance

Optimised 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 a target application.

Low Level Hardware Access

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

Code Reverse Engineering

Reverse engineering is the process of understanding the technology behind a product by extracting the information from a finished product. Reverse engineering is performed by 'hawkers' to reveal the technology behind 'Proprietary Products'.

DRAW BACKS OF ASSEMBLY LANGUAGE BASED DEVELOPMENT

Limitations of assembly language development

High Development Time

Assembly language is much harder to program than high level languages. The developer must pay attention to more details and must have thorough knowledge of the architecture/ memory organisation and register details of the target processor in use. Learning the inner details of the processor and its assembly instructions is highly time consuming and it creates a delay impact in product development.

Developer Dependency

There is no common written rule for developing assembly language based applications whereas all high level languages instruct certain set of rules for application development. In assembly language programming, the developers will have the freedom to choose the different memory location and registers.

Also the programming approach varies from developer to developer depending on his/her taste. For example moving data from a memory location to accumulator can be achieved through different approaches. If the approach done by a developer is not documented properly at the development stage, he/she may not be able to recollect why this approach is followed at a later stage or when a new developer is instructed to analyse this code, he/she also may not be able to understand what is done and why it is done. Hence upgrading an assembly program or modifying it on a later stage is very difficult.

Non -Portable

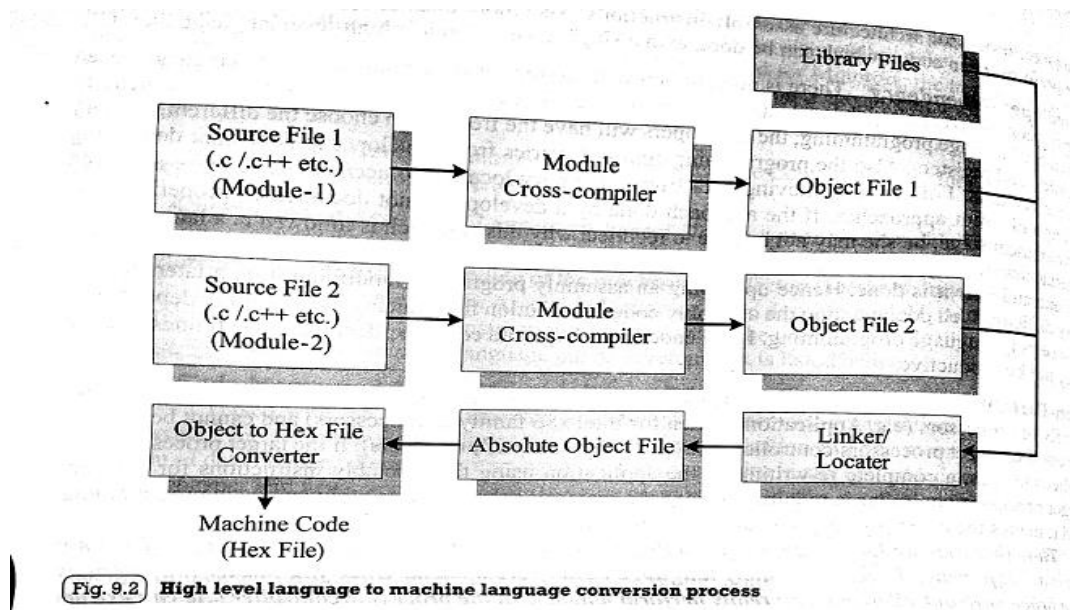
Target applications written in assembly instructions are valid only for that particular family of processors (e.g. Application written for Intel x86 family of processors) and cannot be reused for another target processors/controllers (Say ARM1 1 family of processors). If the target processor/controller changes, a complete re -writing of the application using the assembly instructions for the new target processor/controller is required. This is the major drawback of assembly language programming and it makes the assembly language applications non -portable.

HIGH LEVEL LANGUAGE BASED DEVELOPMENT

The most commonly used high level language for embedded firmware application development is 'C'. 'C' is the well defined, easy to use high level language with extensive cross platform development tool support".

The various steps involved in high level language based embedded firmware development is same as that of assembly language based development except that the conversion of source file written in high level language to object file is done by a cross - compiler, whereas in Assembly language based development it is carried out by an assembler. The various steps involved in the conversion of a program written in high level language to corresponding binary file/machine language is illustrated in Fig. 9.2.

The program written in any of the high level language is saved with the corresponding language extension (.c for C, .cpp for C++ etc).



Most of the high level languages support modular programming approach and hence you can have multiple source files called modules written in corresponding high level language. The source files corresponding to each module is represented by a file with corresponding language extension. Translation of high level source code to executable object code is done by a cross -compiler. The cross -compilers for different high level languages for the same target processor are different.

It should be noted that each high level language should have a cross -compiler for converting the high level source code into the target processor machine code. Without cross -compiler support a high level language cannot be used for embedded firmware development. C51 Cross -compiler from Keil software is an example for Cross -compiler. C51 is a popular cross -compiler available for 'C' language for the 8051 family of micro controller. Conversion of each module's source code to corresponding object file is performed by the cross compiler. Rest of the steps involved in the conversion of high level language to target processor's machine code are same as that of the steps involved in assembly language based development.

ADVANTAGES OF HIGH LEVEL LANGUAGE BASED DEVELOPMENT

Reduced Development Time

Developer requires less or little knowledge on the internal hardware details and architecture of the target processor/controller. The time required for the developer in understanding the target hardware and target machines assembly instructions is reduced by the cross compiler and it reduces the development time.

Developer Independency

The syntax used by most of the high level languages are universal and program written in the high level language can easily be understood by a second person knowing the syntax of the language. High level languages always instruct certain set of rules for writing the code and

commenting the piece of code. If the developer strictly adheres to the rules, the firmware will be 100% developer independent.

Portability

Target applications written in high level languages are converted to target processor/controller understandable format (machine codes) by a cross -compiler. An application written in high level language for a particular target processor can easily be converted to another target processor/controller specific application, with little or less effort by simply re-compiling/little code modification followed by recompiling the application for the required target processor/controller, provided, the cross –compiler has support for the processor/controller selected. This makes applications written in high level language highly portable

LIMITATIONS OF HIGH LEVEL LANGUAGE BASED DEVELOPMENT

Some cross –compilers available for high level languages may not be so efficient in generating optimised target processor specific instructions. The time required to execute a task also increases with the number of instructions.

High level language based Code may not be efficient in accessing low level hardware where hardware access timing is critical order of nano or micro seconds.

The investment required is high compared to Assembly Language based firmware development tools.

What are pointer to constant data and constant pointer to data?

POINTER TO CONSTANT DATA

Pointer to constant data is a pointer which points to a data which is read only. The pointer pointing to the data can be changed but the data is non -modifiable. Example of pointer to constant data

```
const int* x; //Integer pointer x to constant data
int const* x //same meaning as above definition
```

CONSTANT POINTER TO DATA

Constant pointer has significant importance in Embedded C applications. An embedded system under consideration may have various external chips like, data memory, Real Time Clock (RTC), etc interfaced to the target processor/controller, using memory mapped technique.

The range of address assigned to each chips and their registers are dependent on the hardware design. At the time of designing the hardware itself, address ranges are allocated to the different chips and the target hardware is developed according to these address allocations.

For example, assume we have an RTC chip which is memory mapped at address range 0 x 3000 to 0x3010 and the memory mapped address of register holding the time information is at 0x3007. This memory address is fixed and to get the time information we have to look at the register residing at location 0x3007. But the content of the register located at address 0x3007 is subject to change according to the change in time. We can access this data using a constant pointer. The declaration of a constant pointer is given below.

```
// constant character pointer x to constant/variable data
char *const x;
/*Explicit declaration of character pointer pointing to 8bit memory
location,
mapped at location 0x3007; RTC example illustrated above*/
char *const x= (char*) 0x3007;
```

What is constant pointer to constant data?

CONSTANT POINTER TO CONSTANT DATA

Constant pointers pointing to constant data are widely used in embedded programming applications. Typical uses are reading configuration data held at ROM chips which are memory mapped at a specified address range, Read only status registers of different chips, memory mapped at a fixed address. Syntax of declaring a constant pointer to constant data is given below.

```
/*Constant character pointer x pointing to constant data*/
const char *const x;
char const* const x; //Equivalent to above declaration
/*Explicit declaration of constant character pointer* pointing to constant
data/
Char const* const x = (char*) 0x3007;
```

Write the differences between C and embedded C?

'C' V/S. 'EMBEDDED C'

'C' is a well structured, well defined and standardised general purpose programming language with extensive bit manipulation support. 'C' offers a combination of the features of high level language and assembly and helps in hardware access programming (system level programming) as well as business package developments (Application developments like payroll systems, banking applications, etc).

The conventional 'C' language follows ANSI standard and it incorporates various library files for different operating systems. A platform (operating system) specific application, known as, compiler is used for the conversion of programs written in 'C' to the target processor (on which the OS is running) specific binary files. Hence it is a platform specific development.

Embedded 'C' can be considered as a subset of conventional 'C' language. Embedded 'C' supports all 'C' instructions and incorporates a few target processor specific functions/instructions. It should be noted that the standard ANSI 'C' library implementation is always tailored to the target processor/controller library files in Embedded 'C'. The implementation of target processor/controller specific functions/instructions depends upon the processor/controller as well as the supported cross -compiler for the particular Embedded 'C' language. A software program called 'Cross -compiler' is used for the conversion of programs written in Embedded 'C' to target processor/controller specific instructions (machine language).

Write the differences between COMPILER VS. CROSS –COMPILER?

COMPILER VS. CROSS –COMPILER

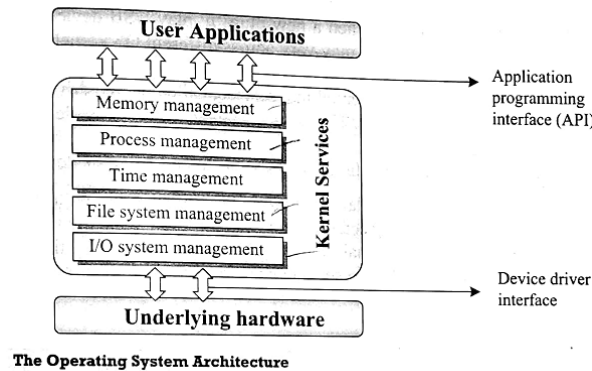
Compiler is a software tool that converts a source code written in a high level language on top of a particular operating system running on a specific target processor architecture (e.g. Intel x86/Pentium). Here the operating system, the compiler program and the application making use of the source code run on the same target processor. The source code is converted to the target processor specific machine instructions. The development is platform specific (OS as well as target processor on which the OS is running). Compilers are generally termed as 'Native Compilers'. A native compiler generates machine code for the same machine (processor) on which it is running.

Cross -compilers are the software tools used in cross -platform development applications. In cross- platform development, the compiler running on a particular target processor/OS converts the source code to machine code for a target processor whose architecture and instruction set is different from the processor on which the compiler is running or for an operating system which is different from the current development environment OS. Embedded system development is a typical example for cross- platform development where embedded firmware is developed on a machine with Intel/AMD or any other target processors and the same is converted into machine code for any other target processor architecture (e.g. 8051, PIC, ARM etc). Keil C51 is an example for cross - compiler. The term 'Compiler' is used interchangeably with 'Cross -compiler' in embedded firmware applications.

Unit-IV

RTOS BASED EMBEDDED SYSTEM DESIGN

1) What is an operating system(OS)? What are the functions of an OS?



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.

The 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
- Organise and manage the system resources efficiently and correctly

2) What is Kernel? Explain the Kernel services of general purpose operating system?

The kernel is the core of the operating system and 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 for handling the following.

Process Management :

Process management deals with managing processes/tasks. It includes setting up the memory space for the process, loading the process's code into the memory space, allocating system resources, scheduling and managing the execution of the process, setting up and managing the Process Control Block (PCB). Inter Process Communication and synchronisation, process termination/deletion, etc.

Primary Memory Management :

The term primary memory refers to the volatile memory (RAM) where processes are loaded and variables and shared data associated with each process are stored. The Memory Management Unit (MMU) of the kernel is responsible for

- Keeping track of which part of the memory area is currently used by which process
- Allocating and De -allocating memory space on a need basis (Dynamic memory allocation).

File System Management:

File is a collection of related information. A file could be a program (source code or executable), text files, image files, word documents, audio/video files, etc. Each of these files differ in the kind of information they hold and the way in which the information is stored. The file operation is a useful service provided by the OS. The file system management service of Kernel is responsible for

- The creation, deletion and alteration of files
- Creation, deletion and alteration of directories
- Saving of files in the secondary storage memory (e.g. Hard disk storage)
- Providing automatic allocation of file space based on the amount of free space available

I/O System (Device) Management

Kernel is responsible for routing the I/O requests coming from different user applications to the appropriate I/O devices of the system. The kernel maintains a list of all the I/O devices of the system. The service 'Device Manager: (Name may vary across different OS kernels) of the kernel is responsible for handling all I/O related operations. The kernel talks to the I/O device through

a set of low-level systems calls, which are implemented in a service, called device drivers.

The device manager is responsible for

- Loading and unloading of device drivers
- Exchanging information and the system specific control signals to and from the device.

Secondary Storage Management

The secondary storage management deals with managing the secondary storage memory devices, if any, connected to the system. Secondary memory is used as backup medium for programs and data since the main memory is volatile. In most of the systems, the secondary storage is kept in disks (Hard Disk). The secondary storage management service of kernel deals with

- Disk storage allocation
- Disk scheduling (Time interval at which the disk is activated to backup data)
- Free Disk space management

Interrupt Handler

Kernel provides handler mechanism for all external/internal interrupts generated by the system.

An interrupt handler or interrupt service routine (ISR) is the function that the kernel runs in response to a specific interrupt:

- Each device that generates interrupts has an associated interrupt handler.
- The interrupt handler for a device is part of the device's driver (the kernel code that manages the device).

Because an interrupt can occur at any time, an interrupt handler can be executed at any time. It is imperative that the handler runs quickly, to resume execution of the interrupted code as soon as possible. It is important that

- To the hardware: the operating system services the interrupt without delay.
- To the rest of the system: the interrupt handler executes in as short a period as possible.

At the very least, an interrupt handler's job is to acknowledge the interrupt's receipt to the hardware.

3) What are Kernel space and User space of an Operating System?

Ans.) The memory space at which the kernel code is located is known as 'Kernel Space'. Similarly, all user applications are loaded to a specific area of primary memory and this memory area is referred as 'User Space'. User space is the memory area where user applications are loaded and executed. The partitioning of memory into kernel and user space is purely Operating System dependent.

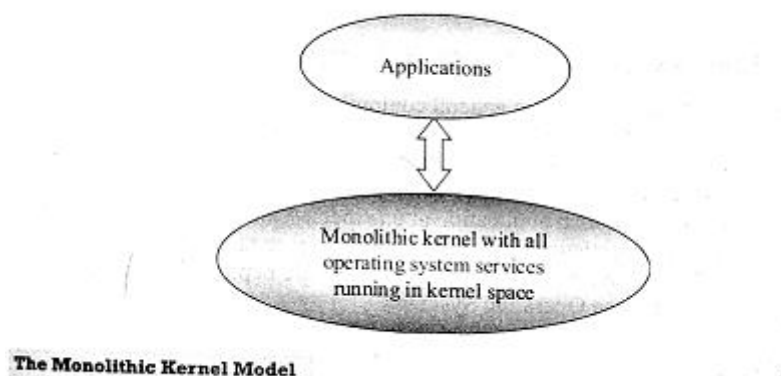
Some OS implements this kind of partitioning and protection whereas some OS do not segregate the kernel and user application code storage into two separate areas. In an operating system with virtual memory support, the user applications are loaded into its corresponding virtual memory space with demand paging technique; Meaning, the entire code for the user application need not be loaded to the main (primary) memory at once; instead the user application code is split into different pages and these pages are loaded into and out of the main memory area on a need basis. The act of loading the code into and out of the main memory is termed as '**Swapping**'. Swapping happens between the main (primary) memory and secondary storage memory.

4) What are the types of Kernel in OS? Explain?

Ans) The kernel forms the heart of an operating system. Different approaches are adopted for building an Operating System kernel. Based on the kernel design, kernels can be classified into 'Monolithic' and 'Micro'.

Monolithic Kernel

In monolithic kernel architecture, all kernel services run in the kernel space. Here 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 utilisation 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. The architecture representation of a monolithic kernel is given in below Figure



Monolithic kernel with all operating system services running in kernel space

Micro kernel

The microkernel design incorporates only the essential set of Operating System services into the kernel. The rest of the Operating System services are implemented in programs known as 'Servers' which runs in user space.

Memory management, process management, timer systems and interrupt handlers are the essential services, which forms the part of the microkernel. Mach, QNX, Minix 3 kernels are examples for microkernel. The architecture representation of a microkernel is shown in below figure

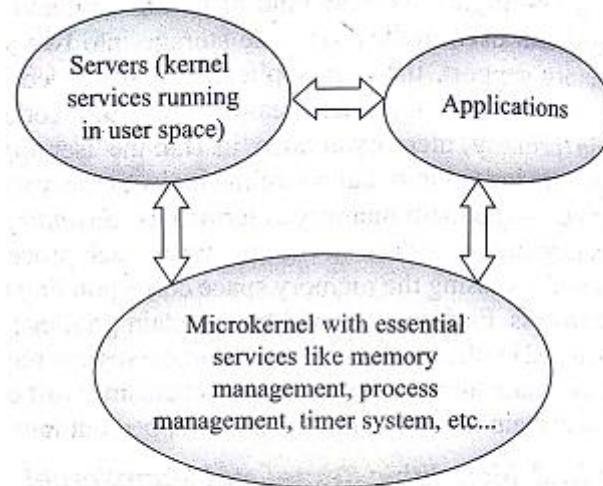


Fig. 10.3 The Microkernel model

Microkernel based design approach offers the following benefits

- **Robustness:** If a problem is encountered in any of the services, which runs as 'Server' application, the same can be reconfigured and re-started without the need for restarting the entire OS. Thus, this approach is highly useful for systems, which demands high 'availability'.
- **Configurability:** Any services, which run as 'Server' application can be changed without the need to restart the whole system. This makes the system dynamically configurable.

5) Explain different types of operating systems?

Ans) 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 different types.

General Purpose Operating System (GPOS):

The operating systems, which are deployed in general computing systems, are referred as General Purpose Operating Systems (GPOS). The kernel of such an OS is more generalised and it contains all kinds of services required for executing generic applications. General-purpose operating systems are often quite Non-deterministic in behaviour. Their services can inject random delays into application software and may cause slow responsiveness of an application at unexpected times.

GPOS are usually deployed in computing systems where deterministic behaviour 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 for General Purpose Operating Systems.

Real -Time Operating System (RTOS)

'Real -Time' implies deterministic timing behaviour. Deterministic timing behaviour in RTOS context means the OS services consumes only known and expected amounts of time regardless the number of services.

A Real -Time Operating System or RTOS implements policies and rules concerning time -critical allocation of a system's resources. The RTOS decides which applications should run in which order and how much time needs to be allocated for each application.

Windows CE, QNX, VxWorks MicroC/OS-11, etc. Are examples of Real -Time Operating Systems (RTOS).

6) What is a Real-Time Kernel? Explain the services of Real-Time Kernel?

Ans) 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 specialised 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 listed below:

- Task/Process management
- Task/Process scheduling
- Task/Process synchronisation
- Error/Exception handling
- Memory management
- Interrupt handling
- Time management

Task/Process Management : It 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 TCB parameters vary across different kernels, based on the task management implementation. Task management service utilises the TCB of a task in the following way

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

Task/Process Scheduling: It 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 behaviour.

Task/Process Synchronisation: It Deals with synchronising the concurrent access of a resource, which is shared across multiple tasks and the communication between various tasks.

Error/Exception Handling: It Deals with registering and handling the errors occurred/exception 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. Error /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. Watchdog timer is a mechanism for handling the timeout for tasks.

Memory Management: Compared to the General Purpose Operating Systems, the memory management function of an RTOS kernel is slightly different. In general, 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 (initialised memory block consumes more allocation time than un-initialised memory block). Since predictable timing and deterministic behaviour are the primary focus of an RTOS, RTOS achieves this by compromising the effectiveness of memory allocation. RTOS makes use of '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'. To achieve predictable timing and avoid the timing overheads, most of the RTOS kernels allow tasks to access any of the memory blocks without any memory protection.

Interrupt Handling : It Deals with the handling of various types of interrupts. Interrupts provide Real-Time behaviour to systems. 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.

Synchronous interrupts:

- 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:

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 over-flow interrupts, and serial data reception / transmission interrupts, etc. are examples for asynchronous interrupts.

Time Management: 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. The System time is updated based on the 'Timer tick'.

The 'Timer tick' interrupt is handled by the 'Timer Interrupt' handler of kernel. The 'Timer tick' interrupt can be utilised 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 preempted by the 'Timer Interrupt' task.

7) Explain Hard and Soft Real-Time Operating Systems?

Ans) **'Hard Real -Time' systems:** Real -Time Operating Systems that strictly adhere to the timing constraints for a task are referred as 'Hard Real -Time' systems. A Hard Real -Time system must meet the deadlines for a task without any delay. Missing any deadline may produce catastrophic results for Hard Real -Time systems, including permanent data loss and irrecoverable damages to the system/users.

Air bag control systems and Anti -lock Brake Systems (ABS) of vehicles are typical examples for Hard Real -Time Systems..

In general, 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' systems:

Real -Time Operating System that does not guarantee meeting deadlines, but offer the best effort to meet the deadline are referred as 'Soft Real -Time' systems. Missing deadlines for tasks are acceptable for a Soft Real-time system if the frequency of deadline missing is within the compliance limit of the Quality of Service (QoS). Soft Real -Time systems most often have a '*human in the loop (HITL)*'.

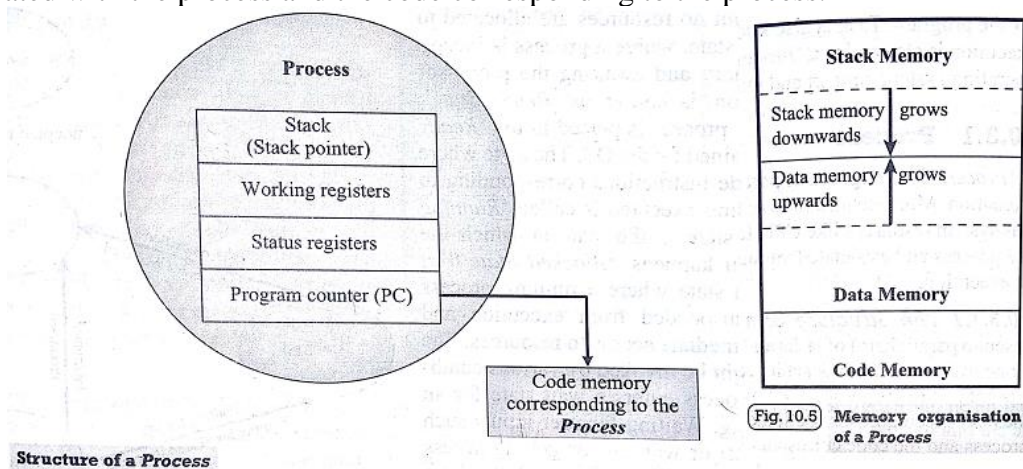
Automatic Teller Machine. (ATM) is a typical example for Soft Real -Time System. If the ATM takes a few seconds more than the ideal operation time, nothing fatal happens. An audio -video playback system is another example for 'Soft Real -Time system. No potential damage arises if a sample comes late by fraction of a second, for playback.

8) What is a Task/Process? Explain?

Ans) A 'Process/Task' is a program, or part of it, in execution. Process is also known as instance of a program in execution. Multiple instances of the same program can execute simultaneously. 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. A process is sequential in execution.

The Structure of a Process:

The concept of 'Process' leads to concurrent execution (pseudo parallelism) of tasks and thereby the efficient utilisation 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.



From a memory perspective, the memory occupied by the process is segregated into three regions, namely, "stack memory, Data memory and Code memory" (Fig. 10.5).

The 'Stack' memory holds all temporary data such as variables local to the process. Data memory holds all the global data for the process. The code memory contains the program code (instructions) corresponding to the process. On loading a process into the main memory, a specific area of memory is allocated for the process.

The stack memory usually starts (OS Kernel implementation dependent) at the highest memory address from the memory area allocated for the process. Say for example, the memory map of the memory area allocated for the process is 2048 to 2100, the stack memory starts at address 2100 and grows downwards to accommodate the variables local to the process.

9) Explain Process life cycle (or) Process states and state Transition?

Ans). Process States and 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.

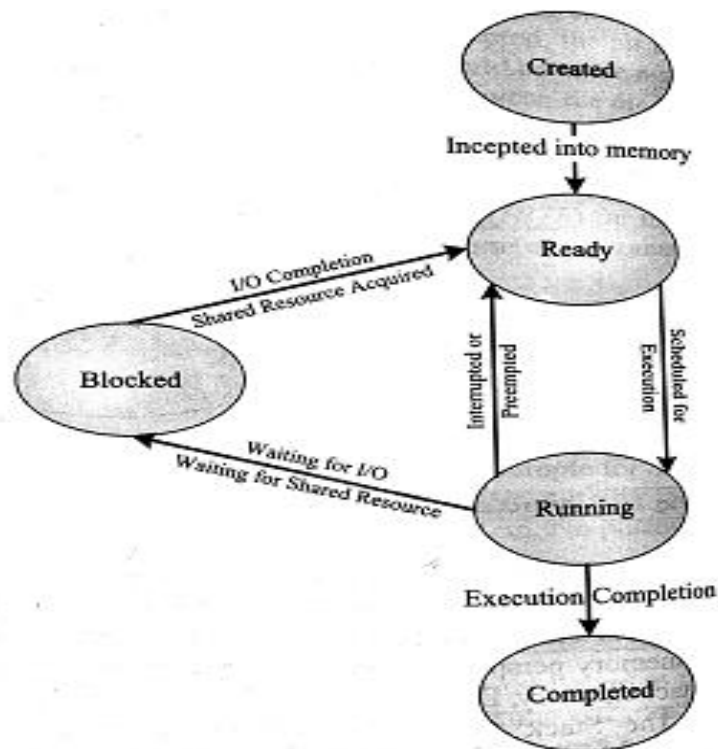


Fig. 10.6 Process states and state transition representation

Figure 10.6 represents the various states associated with a process.

The state at which a process is being created is referred as 'Created State'. The Operating System recognises a process in the 'Created State' but no resources are allocated to the process. The state, where a process is incepted into the memory and awaiting the processor time for execution, is known as 'Ready State'. At this stage, the process is placed in the 'Ready list' queue maintained by the OS. 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 be 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 (will be discussed at a later section of this chapter). A state where the process completes its execution is known as 'Completed State'. The transition of a process from one state to another is known as 'State transition'. When a process changes its state from Ready to running or from running to blocked or terminated or from blocked to running, the CPU allocation for the process may also change.

10) What is a thread? Explain the concept of multi-threading?

Ans). A "thread" is the primitive that can execute code. A thread is a single sequential flow of control within a process this is also known as light weight process. A process can have many threads of execution. Different threads, which are part of process share the same address space; meaning they share the data memory, code memory and stack memory area.

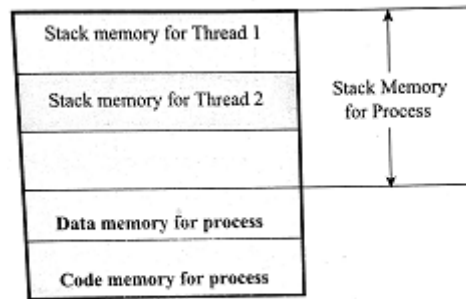
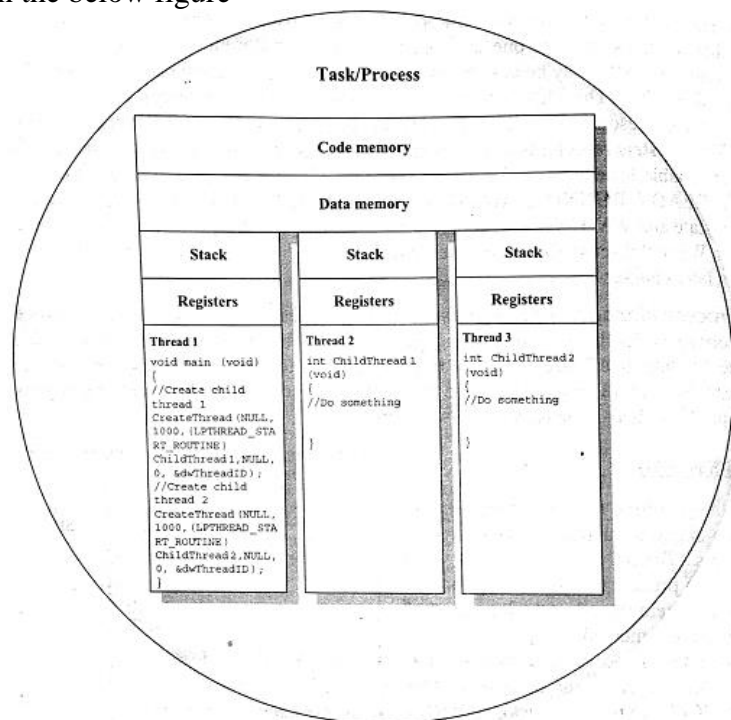


Fig. 10.7 Memory organisation of a Process and its associated Threads

Threads maintain their own thread status, Program Counter PC and stack. The memory model for a process and its associated threads are given in Fig10.7

The Concept of Multithreading

A process/task in embedded application may be a complex or lengthy one and it may contain various suboperations like getting input from I/O devices connected to the processor, performing some internal calculations, operations, updating some I/O devices etc. If all the sub functions of a task are executed in sequence, the CPU utilisation may not be efficient. If the process is waiting for a user input, the CPU enters the wait state for the event, and the process execution also enters a wait state. Instead of this single sequential execution of the whole Process, if the task/process is split into different threads carrying out the different subfunctionalities of the process the CPU can be effectively utilised and when the thread corresponding to the I/O operation enters the wait state, another threads which do not require the I/O event for their operation can be switched into execution. This leads to more speedy execution of the process and the efficient utilisation processor time and resources. The multithreaded architecture of a process can be better visualised with the thread -process diagram shown in the below figure



3) Process with multi-threads

If the process is split into multiple threads, which executes a portion of the process, there will be a main thread and rest of the threads will be created within the main thread. Use of multiple threads to execute a process brings the following advantage.

- Better memory utilisation. 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 utilised by other threads of the process that do not require the event, which the other thread is waiting, for processing. This speeds up the execution of the process.
- Efficient CPU utilisation. The CPU is engaged all time.

11) Write short notes on different types of thread standards?

Ans.) Thread standards deal with the different standards available for thread creation and management. These standards are utilised by the operating systems for thread creation and thread management. It is a set of thread class libraries. The commonly available thread class libraries are explained below.

POSIX Threads: POSIX stands for Portable Operating System Interface. The POSIX.4 standard deals with the Real-Time extensions and POSIX.4a standard deals with thread extensions. The POSIX standard library for thread creation and management is 'Pthreads'. 'Pthreads' library defines the set of POSIX thread creation and management functions in C language.

Win32 Threads: Win32 threads are the threads supported by various flavours of Windows Operating Systems. The Win32 Application Programming Interface (Win32 API) libraries provide the standard set of Win32 thread creation and management functions. Win32 threads are created with the API.

Java Threads :

Java threads are the threads supported by Java programming Language. The java thread class 'Thread' is defined in the package 'java.lang'. This package needs to be imported for using the thread creation functions supported by the Java thread class. There are two ways of creating threads in Java: Either by extending the base 'Thread' class or by implementing an interface. Extending the thread class allows inheriting the methods and variables of the parent class (Thread class) only whereas interface allows a way to achieve the requirements for a set of classes. The following piece of code illustrates the implementation of Java threads with extending the thread base class 'Thread'.

12) Write the differences between threads and process?

Ans)

| 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 memory 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. | Threads within a process share the code, data |

| | |
|---|---|
| The first thread (main thread) calls the main function and occupies the start of the stack memory of the process. | 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. |

13) What is Multiprocessing & Multitasking? Explain the concept of Context Switching?

Ans) **Multiprocessing:** The terms multiprocessing and multitasking are a little confusing and sounds alike. In the operating system context multiprocessing describes the ability to execute multiple processes simultaneously. 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:

The ability of an operating system to hold multiple processes in memory and switch the processor (CPU) from executing one process to another process is known as Multitasking.

Multitasking creates the illusion of multiple tasks executing in parallel. Multitasking involves the switching of CPU from executing one task to another. Whenever a CPU switching happens, the current context of execution should be saved to retrieve it at a later point of time when the CPU executes the process, which is interrupted currently due to execution switching. The context saving and retrieval is essential for resuming a process exactly from the point where it was interrupted due to CPU switching.

Context switching:

The act of switching CPU among the processes or changing the current execution context is known as 'Context switching'. The act of saving the current context which contains the context details (Register details, memory details, system resource usage details, execution details, etc.) for the currently running process at the time of CPU switching is known as 'Context saving'. The process of retrieving the saved context details for a process, which is going to be executed due to CPU switching, is known as 'Context retrieval'. Multitasking involves 'Context switching' (Fig. 10.11), 'Context saving' and 'context retrieval'.

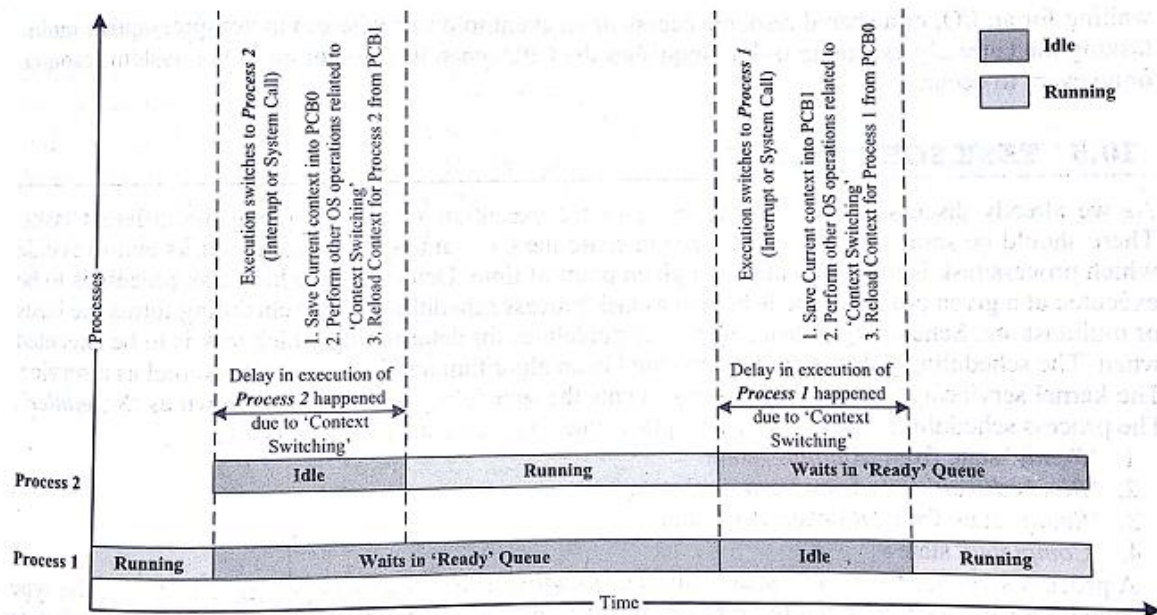


Fig. 10.11 Context switching

14) Explain different types of Multitasking in operating systems?

Ans) Depending on how the switching act is implemented, multitasking can be classified into different types. The following section describes the various types of multitasking existing in the Operating System's context.

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 hold 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.

Nonpreemptive Multitasking

In 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 or system resource. The co-operative and non-preemptive multitasking differs in their behaviour 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, shared resource access or an

event to occur where as in non preemptive multitasking the currently executing task relinquishes the CPU when it waits for an I/O or an event to occur.

15) What is Task Scheduling? Explain different types of task scheduling mechanisms?

Ans) TASK SCHEDULING

Determining which process/task is to be executed at a given point of time is known as task/process scheduling. Task scheduling forms the basis of multitasking. 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 process scheduling decision may take place when a process switches its state to

1. 'Ready' state from 'Running' state
2. 'Blocked/Wait state from 'Running' state
3. 'Ready' state from 'Blocked/Wait' state
4. 'Completed state'

The selection of a scheduling algorithm should consider the following factors:

CPU Utilisation: The scheduling algorithm should always make the CPU utilisation high. CPU utilisation is a direct measure of how much percentage of the CPU is being utilised.

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 minimal 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 summarise, a good scheduling algorithm has high CPU utilisation, minimum Turn Around Time (TAT), maximum throughput and least response time.

The Operating System maintains various queues in connection with the CPU scheduling, and a process passes through these queues during the course of its admittance to execution completion.

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. A process migrates through all these queues during its journey from 'Admitted to 'Completed' stage.

The following diagrammatic representation illustrates the transition a process through Various queues.

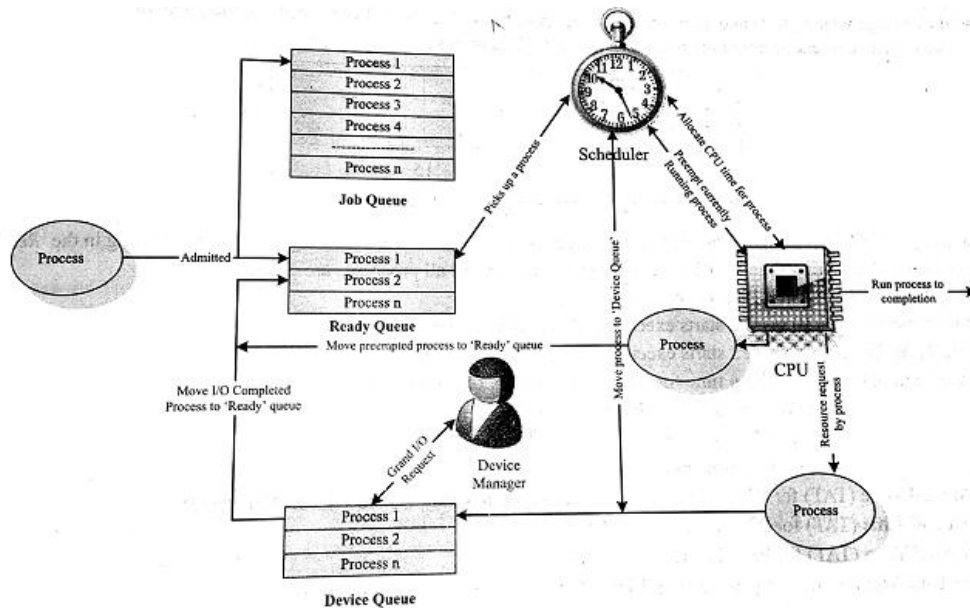


Fig. 10.12 Illustration of process transition through various queues

Types of Task Scheduling

Non -preemptive Scheduling

Non -preemptive scheduling is employed in systems, which implement non-preemptive multitasking model. In this scheduling type, the currently executing task/process is allowed to run until it terminates or enters the 'Wait' state waiting for an I/O or system resource. The various types of non –preemptive scheduling adopted in task/process scheduling are listed below.

First -Come -First -Served (FCFS)/ FIFO Scheduling :As the name indicates, the First- Come -First -Served (FCFS) scheduling algorithm allocates CPU time to the processes based on the order in which they enter the 'Ready queue'. The first entered process is serviced first.

Example Ticketing reservation system.

The major drawback of FCFS algorithm is that it favours monopoly of process. A process, which does not contain any I/O operation, continues its execution until it finishes its task. If the process contains any I/O operation, the CPU is relinquished by the process. In general, FCFS favours 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 utilisation. The average waiting time is not minimal for FCFS scheduling algorithm.

Last -Come -First Served (LCFS)/LIFO Scheduling: The Last -Come -First Served (LCFS) scheduling algorithm also 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.

Shortest Job First (SJF) Scheduling: Shortest Job First (SJF) scheduling algorithm `sorts the 'Ready' queue' each time a process relinquishes the CPU (either the process terminates or enters the 'Wait' state waiting for I/O or system resource) to pick the process with shortest

(least) estimated completion/run time. In SJF, the process with the shortest estimated run time is scheduled first, followed by the next shortest process, and so on.

The average waiting time for a given set of process is minimal in SJF scheduling and so it is optimal compared to other non -preemptive scheduling like FCFS. The major drawback of SJF algorithm is that 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 started its execution (In non -preemptive SJF). This condition is known as '**Starvation**'.

Priority Based Scheduling :

Priority based non –preemptive scheduling algorithm ensures that a process with high priority is serviced at the earliest compared to other low priority processes in the 'Ready' queue. The priority of a task/process can be indicated through various mechanisms. The Shortest Job First (SJF) algorithm can be viewed as a priority based scheduling where each task is prioritised in the order of the time required to complete the task. The lower the time required for completing a process the higher is its priority in SJF algorithm. Another way of priority assigning is associating a priority to the task/process at the time of creation of the task/process. The priority is a number ranging from 0 to the maximum priority supported by the OS. The maximum level of priority is OS dependent. For Example, Windows CE supports 256 levels of priority (0 to 255 priority numbers). While creating the process/task, the priority can be assigned to it. The priority number associated with a task/process is the direct indication of its priority.

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 started 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

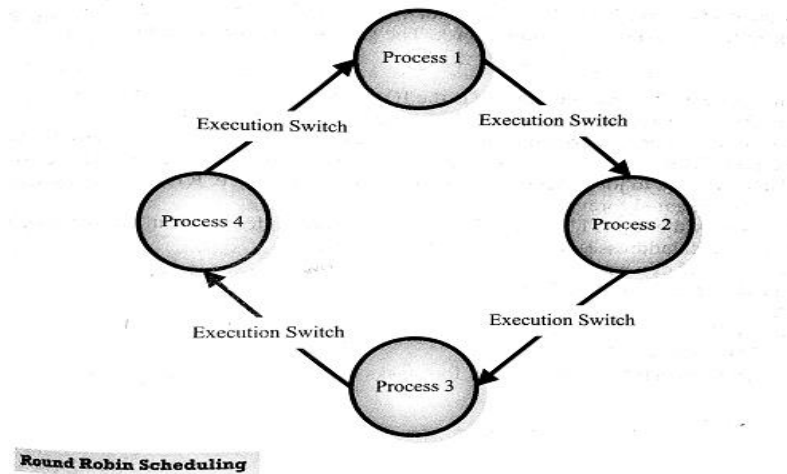
Preemptive scheduling is employed in systems, which implements preemptive multitasking model. In preemptive scheduling, every task in the 'Ready' queue gets a chance to execute. Types of preemptive scheduling adopted in task/process scheduling are explained below

Preemptive SJF Scheduling/Shortest Remaining Time (SRT) :

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 time for the currently executing process. If the execution time of the new process is less, the currently executing process is pre-empted and the new process is scheduled for execution. Preemptive SJF scheduling is also known as Shortest Remaining Time (SRT) scheduling.

Round Robin (RR) Scheduling

In Round Robin 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 (see Fig.).



Round Robin Scheduling

It is executed for a pre -defined time and 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. The sequence is repeated.

This reveals that the Round Robin scheduling is similar to the FCFS scheduling and the only difference is that time slice based preemption is added to switch the execution between the processes in the 'Ready' queue. The 'Ready' queue can be considered as a circular queue in which the scheduler picks up the first process for execution and moves to the next till the end of the queue and then comes back to the beginning of the queue to pick up the first process. RR scheduling involves lot of overhead in maintaining the time slice information for every process which is currently being executed.

Priority Based Scheduling :

Priority based preemptive scheduling algorithm is same as that of the non -preemptive priority based scheduling except for the switching of execution between tasks. In preemptive 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 relinquishes the CPU. The priority of a task/process in preemptive scheduling is indicated in the same way as that of the mechanism adopted for non -preemptive multitasking.

UNIT-V

TASK COMMUNICATION

1. What is Inter process/ Task Communication? Explain different Inter process/ Task Communication mechanisms? (Mar 2017-R13)

Ans) The mechanism through which processes/tasks communicate each other is known as Inter Process/Task Communication (IPC). Inter Process Communication is essential for process co-ordination. The various types of Inter Process-Communication (IPC) mechanisms adopted by process are kernel (Operating System) dependent. Some of the important IPC mechanisms adopted by various kernels are explained below.

Shared Memory:

Processes share some area of the memory to communicate among them (Fig. 10.16).

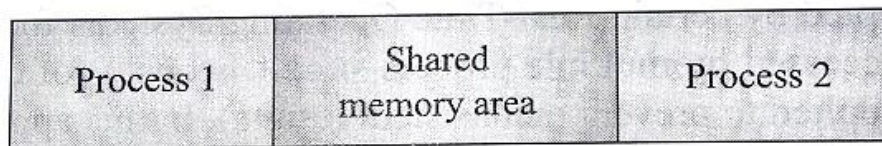


Fig. 10.16 Concept of Shared Memory

Information to be communicated by the process is written to the shared memory area. Other processes which require this information can read the same from the shared memory area. It is same as the real world example where 'Notice Board' is used by corporate to publish the public information among the employees (The only exception is; only corporate have the right to modify the information published on the Notice board and employees are given 'Read' only access, meaning it is only a one way channel).

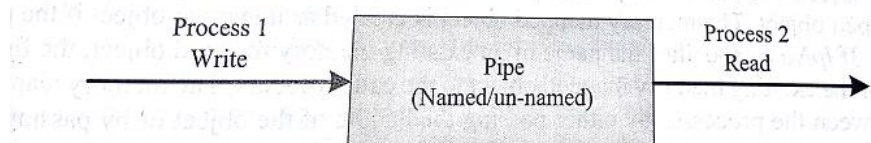
The implementation of shared memory concept is kernel dependent. Different mechanisms are adopted by different kernels for implementing this. A few among them are:

Pipes :

'Pipe' is a section of the shared memory used by processes for communicating. Pipes follow the client server architecture. A process which creates a pipe is known as a pipe server and a process which connects to a pipe is known as Pipe Client. A pipe can be considered as a channel for information flow and has two conceptual ends.

A unidirectional pipe allows the process connecting at one end of the pipe to write to the pipe and the process connected at the other end of the pipe to read the data, whereas a bi-directional pipe allows both reading and writing at one end.

The unidirectional pipe can be visualised as



Concept of Pipe for IPC

The implementation of 'Pipes' is also OS dependent. Microsoft® Windows Desktop Operating Systems support two types of 'Pipes' for Inter Process Communication. They are:

Anonymous Pipes: The anonymous pipes are unnamed, unidirectional pipes used for data transfer between two processes.

Named Pipes: Named pipe is a named, unidirectional or bi-directional pipe for data exchange between processes. Like anonymous pipes, the process which creates the named pipe is known as pipe server. A process which connects to the named pipe is known as pipe client. With named pipes, any process can act as both client and server allowing point-to-point communication. Named pipes can be used for communicating between processes running on the same machine or between processes running on different machines connected to a network.

Memory Mapped Objects: Memory mapped object is a shared memory technique adopted by certain Real-Time Operating Systems for allocating a shared block of memory which can be accessed by multiple processes simultaneously. In this approach a mapping object is created and physical storage for it is reserved and committed. A process can map the entire committed physical area or a block of it to its virtual address space. All read and write operation to this virtual address space by a process is directed to its committed physical area. Any process which wants to share data with other processes can map the physical memory area of the mapped object to its virtual memory space and use it for sharing the data.

Windows CE 5.0 RTOS uses the memory mapped object based shared memory technique for Inter Process Communication (Fig. 10.18).

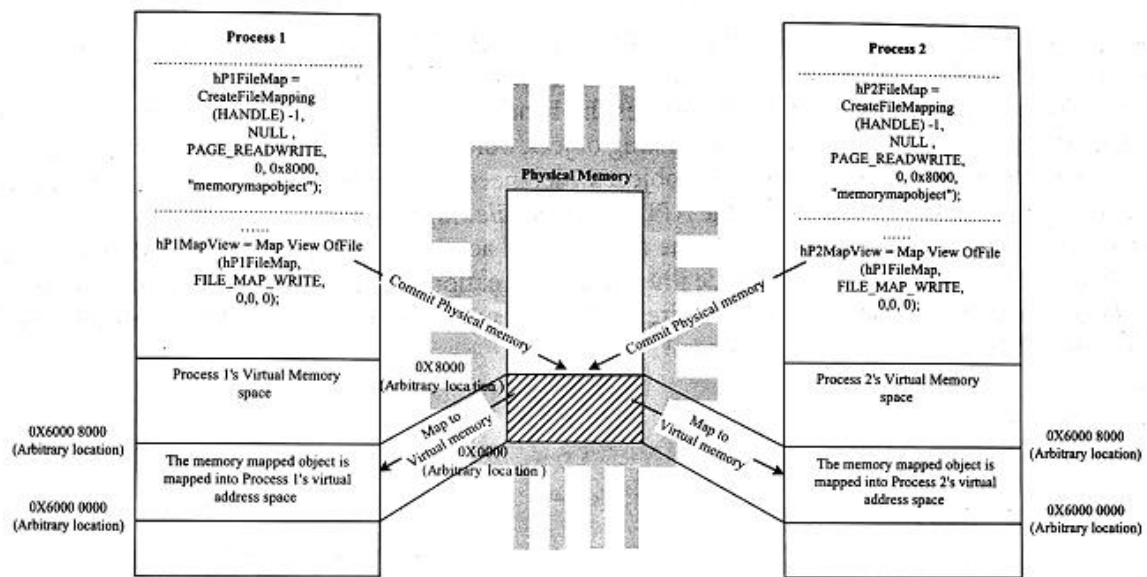


Fig. 10.18 Concept of memory mapped object

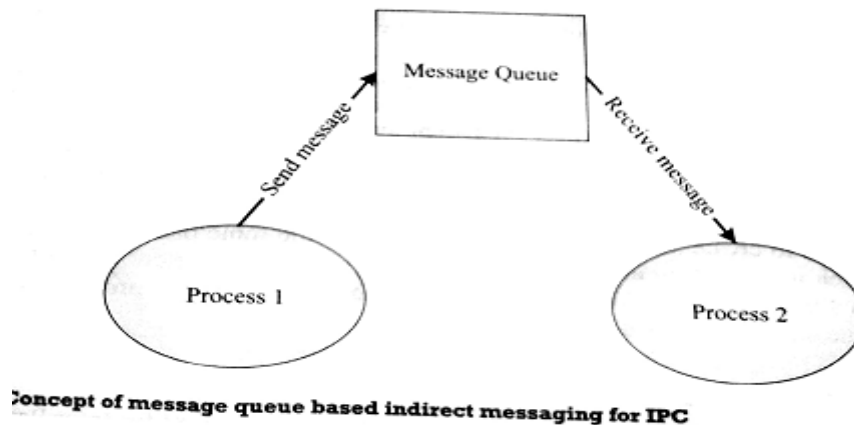
Message Passing:

Message passing is an asynchronous information exchange mechanism used for Inter Process/Thread Communication. The major difference between shared memory and message passing technique is that, through shared memory lots of data can be shared whereas only

limited amount of info/data is passed through message passing. Also message passing is relatively fast and free from the synchronisation overheads compared to shared memory.

Message Queue:

Usually the process which wants to talk to another process posts the message to a First-In-First-Out (FIFO) queue called 'Message queue', which stores the messages temporarily in a system defined memory object, to pass it to the desired process .



Messages are sent and received through *send* (Name of the process to which the message is to be sent, message) and *receive* (Name of the process from which the message is to be received, message) methods. The messages are exchanged through a message queue. The implementation of the message queue, send and receive methods are OS kernel dependent.

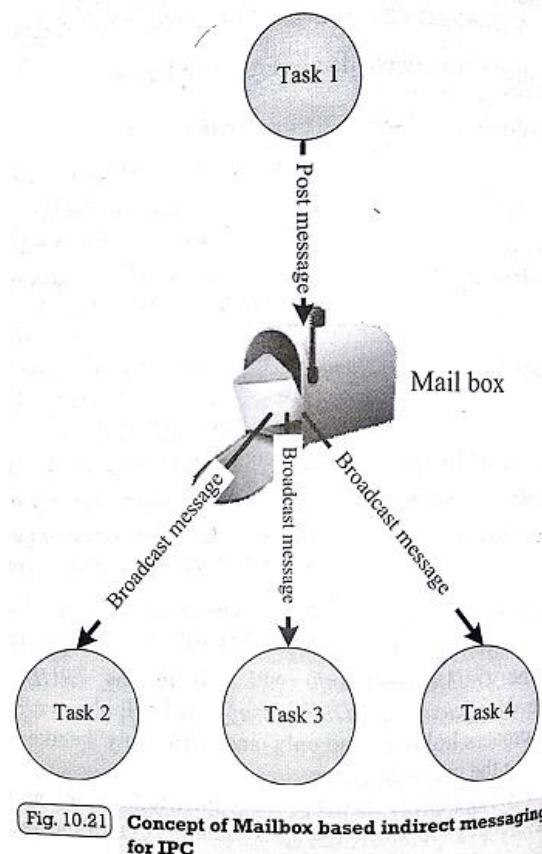
A Process/thread which wants to communicate with another Process/thread, posts the message to the system message queue. The kernel picks up the message from the system message queue one at a time and examines the message for finding the destination Process/thread and then posts the message to the message queue of the corresponding Process/thread

The messaging mechanism is classified into synchronous and asynchronous based on the behaviour of the message posting thread. In asynchronous messaging, the message posting thread just posts the message to the queue and it will not wait for an acceptance (return) from the thread to which the message is posted, whereas in synchronous messaging, the thread which posts a message enters waiting state and waits for the message result from the thread to which the message is posted.

Mailbox :

Mailbox is an alternate form of and it is used in certain Real-Time Operating Systems for IPC. Mailbox technique for IPC in RTOS is usually used for one way messaging. The task/thread which wants to send a message to other tasks/threads creates a mailbox for posting the messages. The threads which are interested in receiving the messages posted to the mailbox by the mailbox creator thread can subscribe to the mailbox. The thread which creates the mailbox is known as 'mailbox server' and the threads which subscribe to the mailbox are known as 'mailbox clients'. The mailbox server posts messages to the mailbox and notifies it to the clients which are subscribed to the mailbox. The clients read the message from the mailbox on receiving the notification.

The Micro C/OS-II implements mailbox as a mechanism for inter-task communication. Figure 10.21 given below illustrates the mailbox based IPC technique.



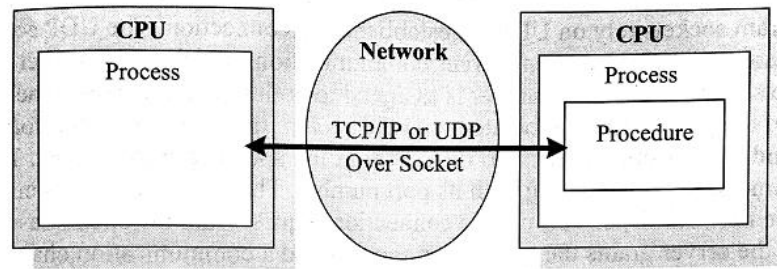
Signalling :

Signalling is a primitive way of communication, between processes/threads. Signals are used for asynchronous notifications where one process/thread fires a signal, indicating the occurrence of a scenario which the other processes/threads is waiting. Signals are not queued and they do not carry any data. The communication mechanisms used in RTX51 Tiny OS is an example for Signalling. The *os _send_signal* kernel call under RTX 51 sends a signal from task to a specified task. Similarly the one for a specified signal *os _wait* kernel call waits.

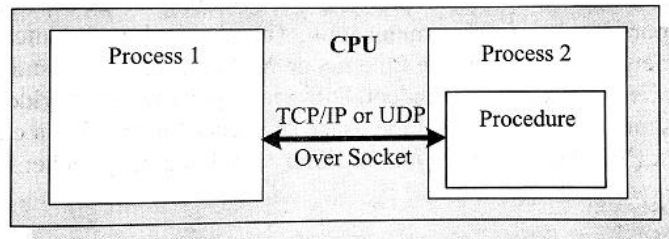
Remote Procedure Call (RPC) and Sockets:

Remote Procedure Call or RPC (Fig. 10.22) is the Inter Process Communication mechanism used by a process to call a procedure of another process running on the same CPU or on a different CPU which is interconnected in a network. In the object oriented language terminology RPC is also known as Remote Invocation or Remote Method Invocation is mainly used for distributed applications like client -server applications. With RPC it is possible to communicate over a heterogeneous network (i.e. network where Client and server applications are running on different Operating systems).

The CPU/process containing the procedure which needs to be invoked remotely is known as server. The CPU/process which initiates an RPC request is known as client.



Processes running on different CPUs which are networked



Processes running on same CPU

Concept of Remote Procedure Call (RPC) for IPC

The RPC communication can be either Synchronous (Blocking) or Asynchronous (Non - blocking). In the Synchronous communication, the process which calls the remote procedure is blocked until it receives a response back from the other process. In asynchronous RPC calls, the calling process continues its execution while the remote process performs the execution of the procedure.

Sockets are used for RPC communication. Socket is a logical endpoint in a two-way communication link between two applications running on a network. A port number is associated with a socket so that the network layer of the communication channel can deliver the data to the designated application. Sockets are of different types, namely, Internet sockets (INET), UNIX sockets, etc. The INET socket works on internet communication protocol. TCP/IP, UDP, etc. are the communication protocols used by INET sockets. INET sockets are classified into:

1. Stream sockets
2. Datagram sockets

Stream sockets are connection oriented and they use TCP to establish a reliable connection. On the other hand, Datagram sockets rely on UDP for establishing a connection. The UDP connection is unreliable when compared to TCP.

2) What is Task/Process synchronisation? Explain different Task/Process synchronisation issues? (Nov 2016-R13)

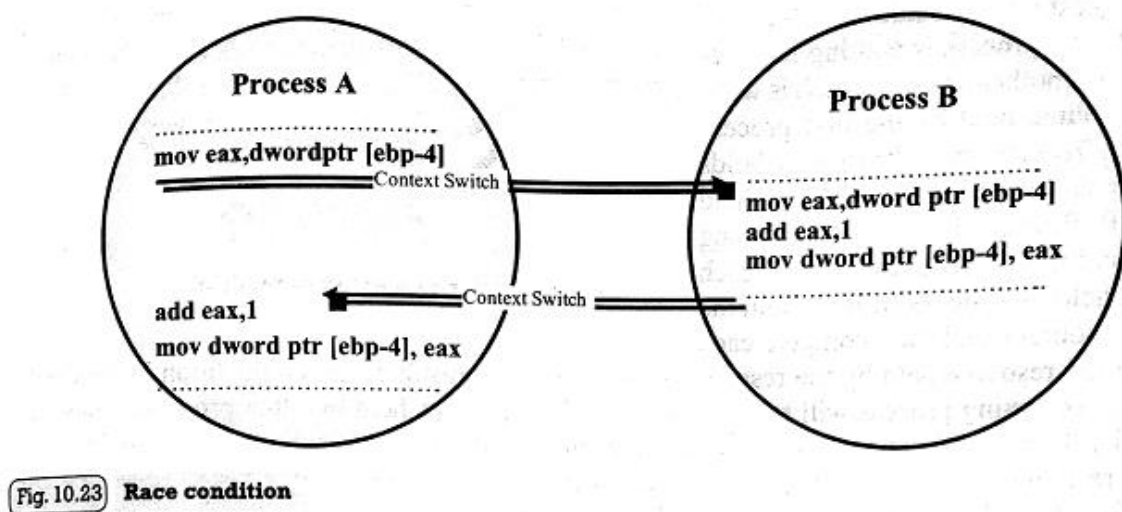
Ans) The act of making processes aware of the access of shared resources by each process to avoid conflicts is known as 'Task/Process Synchronisation'. Various synchronisation issues may arise in a multitasking environment if processes are not synchronised properly. The following sections describe the major task communication synchronisation issues observed in multitasking and the commonly adopted synchronisation techniques to overcome these issues.

Racing

Racing or Race condition is the situation in which multiple processes compete (race) each other to access and manipulate shared data concurrently. In a Race condition the final value of the shared data depends on the process which acted on the data finally.

Example:

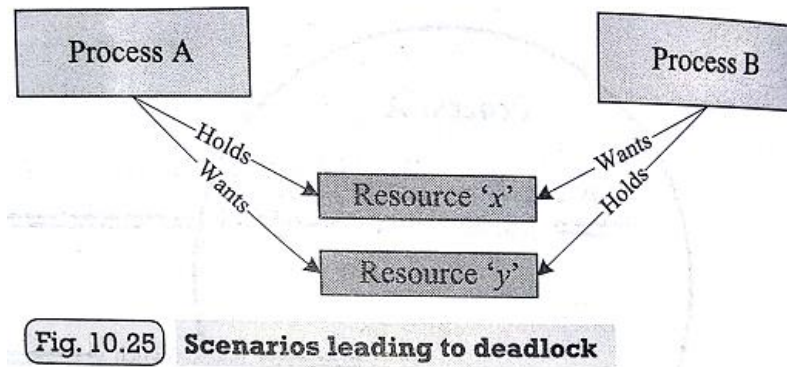
Imagine a situation where a process switching (context switching) happens from Process A to Process B when Process A is executing the `counter++;` statement. Process A accomplishes the `counter++;` statement through three different low level instructions. Now imagine that the process switching happened at the point where Process A executed the low level instruction, `'mov eax,dword ptr [ebp-4]'` and is about to execute the next instruction `'add eax,1'`. The scenario is illustrated in Fig. 10.23. Process B increments the shared variable 'counter' in the middle of the operation where Process A tries to increment it. When Process A gets the CPU time for execution, it starts from the point where it got interrupted (If Process B is also using the same registers `eax` and `ebp` for executing `counter++;`;



instruction, the original content of these registers will be saved as part of the context saving and it will be retrieved back as part of context retrieval, when process A gets the CPU for execution. Hence the content of `eax` and `ebp` remains intact irrespective of context switching). Though the variable counter is incremented by Process B, Process A is unaware of it and it increments the variable with the old value. This leads to the loss of one increment for the variable counter. This problem occurs due to uninterruptable operation on variables. This Condition is called Race.

Deadlock

A race condition produces incorrect results whereas a deadlock condition creates a situation where none of the processes are able to make any progress in their execution, resulting in a set of deadlocked processes. In its simplest form 'deadlock' is the condition in which a process is waiting for a resource held by another process which is waiting for a resource held by the first process (Fig. 10.25).



To elaborate: Process A holds a resource 'x' and it wants a resource 'y' held by Process B. Process B is currently holding resource 'y' and it wants the resource 'x' which is currently held by Process A. Both hold the respective resources and they compete each other to get the resource held by the respective processes. The result of the competition is 'deadlock'.

None of the competing process will be able to access the resources held by other processes since they are locked by the respective processes (If a mutual exclusion policy is implemented for shared resource access, the resource is locked by the process which is currently accessing it).

Livelock: The Livelock condition is similar to the deadlock condition except that a process in Livelock condition changes its state with time. While in deadlock a process enters in wait state for a resource and continues in that state forever without making any progress in the execution, in a livelock condition a process always does something but is unable to make any progress in the execution completion.

Starvation: In the multitasking context, starvation is the condition in which a process does not get the resources required to continue its execution for a long time. As time progresses the process starves on resource. Starvation may arise due to various conditions like by product of preventive measures of deadlock, scheduling policies favouring high priority tasks and tasks with shortest execution time, etc.

3) What are Coffman Conditions? (Nov 2016-R13)

Ans). 'Deadlock' is a result of the combined occurrence of these four conditions listed below. These conditions are first described by E. G. Coffman in 1971 and it is popularly known as *Coffman* conditions.

Mutual Exclusion: The criteria that only one process can hold a resource at a time. Meaning processes should access shared resources with mutual exclusion. Typical example is the accessing of display hardware in an embedded device.

Hold and Wait: The condition in which a process holds a shared resource by acquiring the lock controlling the shared access and waiting for additional resources held by other processes.

No Resource Pre-emption: The criteria that operating system cannot take back a resource from a process which is currently holding it and the resource can only be released voluntarily by the process holding it.

Circular Wait: A process is waiting for a resource which is currently held by another process which in turn is waiting for a resource held by the first process. In general, there exists a set of waiting process P_0, P_1, \dots, P_n with P_0 is waiting for a resource held by P_1 and P_1 is waiting for a resource held by P_0 , P_n is waiting for a resource held by P_0 and P_0 is waiting for a resource held by P_n and so on... This forms a circular wait queue.

4) Explain different types of Task Synchronisation Techniques? (Nov-2014)

Ans) Various techniques are used for synchronisation in concurrent access in multitasking. Generally Process/Task synchronisation is essential for

1. Avoiding conflicts in resource access (racing, deadlock, starvation, livelock, etc.) in a multitasking environment.
2. Ensuring proper sequence of operation across processes.
3. Communicating between processes.

The code memory area which holds the program instructions (piece of code) for accessing a shared resource (like shared memory, shared variables, etc.) is known as "*critical section*". In order to synchronise the access to shared resources, the access to the critical section should be exclusive. The exclusive access to critical section of code is provided through mutual exclusion mechanism.

A mutual exclusion policy enforces mutually exclusive access of critical sections. Mutual exclusions can be enforced in different ways. Mutual exclusion blocks a process. Based on the behaviour of the blocked process, mutual exclusion methods can be classified into two categories. In the following section we will discuss them in detail.

Mutual Exclusion through Busy Waiting/Spin Lock 'Busy waiting' : It is the simplest method for enforcing mutual exclusion. The following code snippet illustrates how 'Busy waiting' enforces mutual exclusion.

The 'Busywaiting' technique uses a lock variable for implementing mutual exclusion. Each process/ thread checks this lock variable before entering the critical section. The lock is set to '1' by a process/ thread if the process/thread is already in its critical section; otherwise the lock is set to '0'. The major challenge in implementing the lock variable based synchronisation is the non -availability of a single atomic instruction which combines the reading, comparing and setting of the lock variable.

The lock based mutual exclusion implementation always checks the state of a lock and waits till the lock is available. This keeps the processes/threads always busy and forces the processes/threads to wait for the availability of the lock for proceeding further. Hence this synchronisation mechanism is popularly known as 'Busy waiting'.

The 'Busy waiting' technique can also be visualised as a lock around in which the process/thread spins, checking for its availability. Spin locks are useful in handling scenarios where the processes/threads are likely to be blocked for a shorter period of time on waiting the lock, as they avoid OS overheads on context saving and process re -scheduling. Another drawback of Spin lock based synchronisation is that if the lock is being held for a long time by a process and if it is pre-empted by the OS, the other threads waiting for this lock may have to spin a longer time for getting it. The 'Busy waiting' mechanism keeps the process/threads always active, performing a task which is not useful and leads to the wastage of processor time and high power consumption.

Mutual Exclusion through Sleep & Wakeup:

The 'Busy waiting' mutual exclusion enforcement mechanism used by processes makes the CPU always busy by checking the lock to see whether they can proceed. This results in the wastage of CPU time and leads to high power consumption. This is not affordable in embedded systems powered on battery, since it affects the battery backup time of the device. An alternative to 'busy waiting' is the 'Sleep & Wakeup' mechanism.

When a process is not allowed to access the critical section, which is currently being locked by another process, the process undergoes 'Sleep' and enters the 'blocked' state. The process which is blocked on waiting for access to the critical section is awakened by the process which currently owns the critical section. The process which owns the critical section sends a wakeup message to the process, which is sleeping as a result of waiting for the access to the critical section, when the process leaves the critical section. The 'Sleep & Wakeup' policy for mutual exclusion can be implemented in different ways. Implementation of this policy is OS kernel dependent.

Semaphore

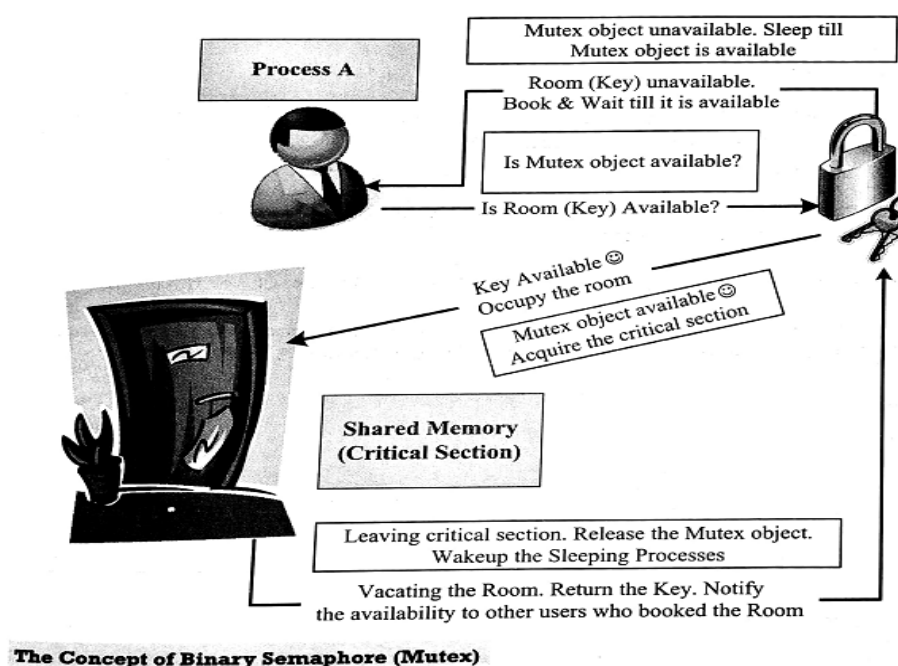
Semaphore is a sleep and wakeup based mutual exclusion implementation for shared resource access. Semaphore is a system resource and the process which wants to access the shared resource can first acquire this system object to indicate the other processes which wants the shared resource that the shared resource is currently acquired by it.

Based on the implementation of the sharing limitation of the shared resource, semaphores are classified into two; namely '*Binary Semaphore*' and '*Counting Semaphore*'.

Binary Semaphore (Mutex)

The binary semaphore provides exclusive access to shared resource by allocating the resource to a single process at a time and not allowing the other processes to access it when it is being owned by a process. The implementation of binary semaphore is OS kernel dependent. Under certain OS kernel it is referred as *mutex*. Binary Semaphore (Mutex) is a synchronisation object provided by OS for process/thread synchronisation.

A real world example for the mutex concept is the hotel accommodation system (lodging system) as shown in figure.



The rooms in a hotel are shared for the public. Any user who pays and follows the norms of the hotel can avail the rooms for accommodation. A person wants to avail the hotel room facility can contact the hotel reception for checking the room availability. If room is available the receptionist will handover the room key to the user. If room is not available currently, the user can book the room to get notifications when a room is available. When a person gets a room he/she is granted the exclusive access to the room facilities like TV, telephone, toilet, etc. When a user vacates the room, he/ she gives the keys back to the receptionist. The receptionist informs the users, who booked in advance, about the room's availability.

Counting Semaphore:

Unlike a binary semaphore, the 'Counting Semaphore' limits the access of resources by a fixed number of processes/threads. 'Counting Semaphore' maintains a count between zero and a value. It limits the usage of the resource to the maximum value of the count supported by it.

A real world example for the counting semaphore concept is the dormitory system for accommodation (Fig. 10.34). A dormitory contains a fixed number of beds (say 5) and at any point of time it can be shared by the maximum number of users supported by the dormitory. If a person wants to avail the dormitory facility, he/she can contact the dormitory caretaker for checking the availability. If beds are available in the dorm the caretaker will hand over the keys to the user. If beds are not available currently, the user can register his/her name to get notifications when a slot is available.

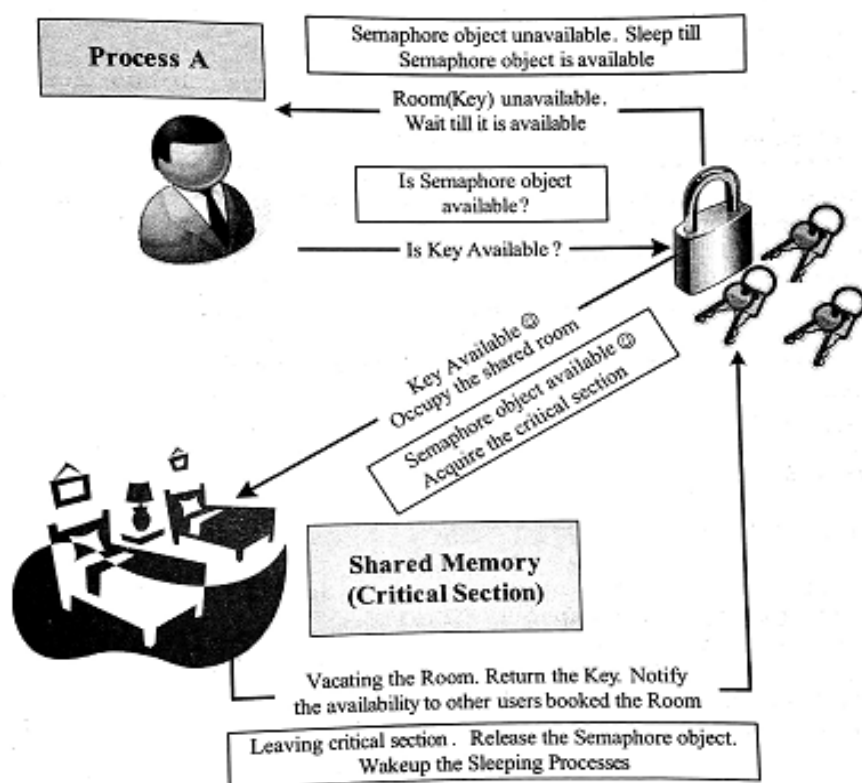


Fig. 10.34 The Concept of Counting Semaphore

Those who are availing the dormitory shares the dorm facilities like TV, telephone, toilet, etc. When a dorm user vacates, he/she gives the keys back to the caretaker. The caretaker informs the users, who booked in advance, about the dorm availability.

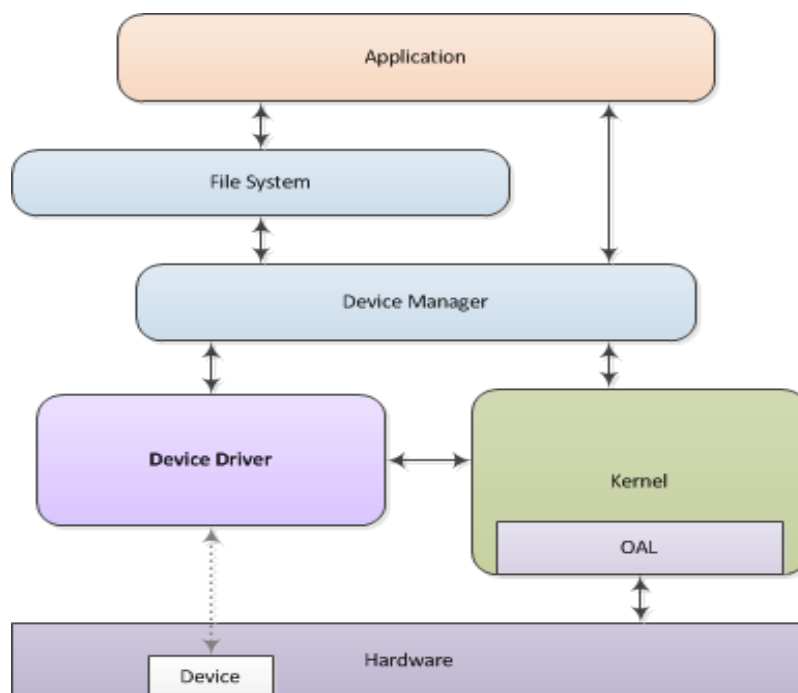
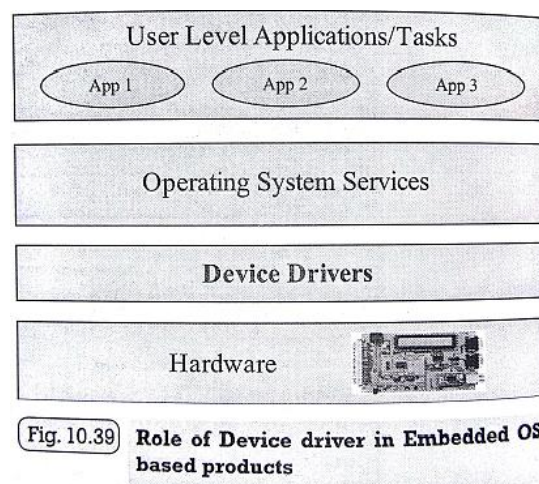
5) What is a device driver? Explain the role of device driver in an embedded OS?
(Mar 2017-R13)

(OR)

Explain the architecture of device driver, with neat sketch and give the applications of device drivers. (Nov 2016-R13)

Ans) **DEVICE DRIVERS:**

Device driver is a piece of software that acts as a bridge between the operating system and the hardware. In an operating system based product architecture, the user applications talk to the Operating System kernel for all necessary information exchange including communication with the hardware peripherals.



Architecture of Device Driver

The device driver abstracts the hardware from user applications. The topology of user applications and hardware interaction in an RTOS based system is depicted in Fig. 10.39. Device drivers link physical or virtual devices with the OS, making devices available to the OS and applications through an interface. As shown in architecture diagram, applications communicate with device drivers through the file system and the *Device Manager*. The Device Manager interacts with device drivers and the kernel to manage device access. Device drivers call kernel APIs to access hardware; the kernel, in turn, calls functions in the OEM Adaptation Layer (OAL) to read and write device registers. When the device generates an interrupt, the kernel intercepts the interrupt and forwards it to the device driver as an interrupt event.

Device drivers are responsible for initiating and managing the communication with the hardware peripherals. They are responsible for establishing the connectivity, initialising the hardware and transferring data. An embedded product may contain different types of hardware components like Wi-Fi module, File systems, Storage device interface, etc. The initialisation of these devices and the protocols required for communicating with these devices may be different.

If an external device, whose driver software is not available with OS kernel image, is connected to the embedded device (Say a medical device with custom USB class implementation is connected to the USB port of the embedded product), the OS prompts the user to install its driver manually. Device drivers which are part of the OS image are known as 'Built-in drivers' or '**On-board drivers**'. These drivers are loaded by the OS at the time of booting the device and are always kept in the RAM. Drivers which need to be installed for accessing a device are known as '**Installable drivers**'. These drivers are loaded by the OS on a need basis.

The driver files are usually in the form of a '**dll**' file. Drivers can run on either user space or kernel space. 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. On the other hand, if an exception occurs in the kernel mode driver, it may lead to the kernel crash. However regardless of the OS types, a device driver implements the following:

1. Device (Hardware) Initialisation and Interrupt configuration
2. Interrupt handling and processing
3. Client interfacing (Interfacing with user applications)

Applications of Device Drivers

Because of the diversity of modern hardware and operating systems, drivers operate in many different environments. Drivers may interface with:

- Printers
- Video adapters
- Network cards
- Sound cards
- Local buses of various sorts—in particular, for bus mastering on modern systems
- Low-bandwidth I/O buses of various sorts (for pointing devices such as mice, keyboards, USB, etc.)

- Computer storage devices such as hard disk, CD-ROM, and floppy disk buses (ATA, SATA, SCSI)
- Implementing support for different file systems
- Image scanners
- Digital cameras

Common levels of abstraction for device drivers include:

- For hardware:
 - Interfacing directly
 - Writing to or reading from a device control register
 - Using some higher-level interface (e.g. Video BIOS)
 - Using another lower-level device driver (e.g. file system drivers using disk drivers)
 - Simulating work with hardware, while doing something entirely different^[8]
- For software:
 - Allowing the operating system direct access to hardware resources
 - Implementing only primitives
 - Implementing an interface for non-driver software (e.g., TWAIN)
 - Implementing a language, sometimes quite high-level (e.g., PostScript)

6) Explain different functional and non functional requirements that need to be evaluated in the selection of RTOS? (Nov-2014)

Ans) The decision of choosing an RTOS for an embedded design is very crucial. A lot of factors need to be analysed carefully before making a decision on the selection of an RTOS. These factors can be either functional or non-functional.

FUNCTIONAL REQUIREMENTS:

Processor Support It is not necessary that all RTOS's support all kinds of processor architecture. It is essential to ensure the processor support by the RTOS.

Memory Requirements

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

Real-time Capabilities

It is not mandatory that the operating system for all embedded systems need to be Real-time and all embedded Operating systems -are 'Real -tin - le' iii behaviour. The task/process scheduling policies plays an important role in the 'Real-time' behaviour of an OS. Analyse the real-time capabilities of the OS under consideration and the standards met by the operating system for real-time capabilities.

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.

Inter Process Communication and Task Synchronisation

The implementation of Inter Process communication and Synchronisation is OS kernel dependent. Certain kernels may provide a bunch of options whereas others provide very

limited options. Certain kernels implement policies for avoiding priority inversion issues in resource sharing.

Development Language Support

Certain operating systems include the run time libraries required for running applications written in languages like Java and C#. A Java Virtual Machine (JVM) customised for the Operating System is essential for running java applications. Similarly the .NET Compact Framework (.NETCF) is required for running Microsoft® .NET applications on top of the Operating System. The OS may include these components as built-in component, if not, check the availability of the same from a third party vendor for the OS under consideration.

NON-FUNCTIONAL REQUIREMENTS

Custom Developed or Off the Shelf

Depending on the OS requirement, it is possible to go for the complete development of an operating system suiting the embedded system needs or use an off the shelf, readily available operating system, which is either a commercial product or an Open Source product, which is in close match with the system requirements. Sometimes it may be possible to build the required features by customising 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.

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.

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 Operating Systems may be superior in performance, but the availability of tools for supporting the development may be limited. Explore the different tools available for the OS under consideration.

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.

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 analysed thoroughly.