EMBEDDED SYSTEMS

by

Mr. S Lakshmanachari
Assistant Professor
ECE Department

Bloom’s Taxonomy, PO, PSO, CO, Syllabus

EMBEDDED SYSTEMS

III Semester: ECE								
Course Code	**Category**	**Hours / Week**			**Credits**	**Maximum Marks**		
AEC016	Core	**L**	**T**	**P**	**C**	**CIA**	**SEE**	**Total**
		3	0	0	3	30	70	100
Contact Classes:45	**Tutorial Classes:15**	**Practical Classes: Nil**				**Total Classes:60**		

OUTCOME BASED EDUCATION

OBE incorporates the three elements in different way:

1. Theory of education
2. A systematic structure for education
3. A specific approach to instructional practice.

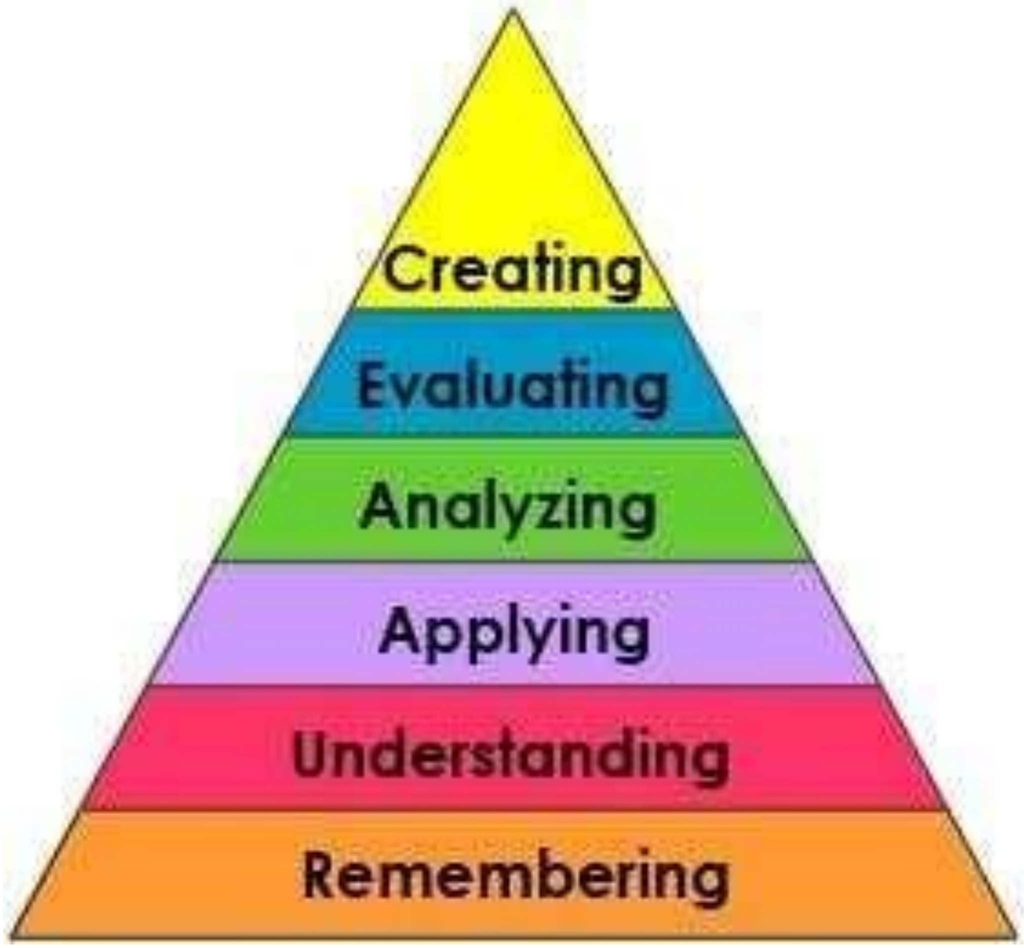
It focuses on the following skills when developing curricula and outcomes:

- Life skills
- Basic skills
- Professional and vocational skills
- Intellectual skills
- Interpersonal and personal skills

Four levels of outcomes from OBE are:

- **Program Educational Objectives (PEOs)**
- **Program Outcomes (POs)**
- **Program Specific Outcomes (PSOs)**
- **Course Outcomes (COs)**

BLOOM’S TAXONOMY



BLOOM'S TAXONOMY

Remembering: recall or remember the information	Define, Duplicate, List, Memorize, Recall, Repeat, Reproduce, State.
Understanding: explain ideas or concepts	Classify, Describe, Discuss, Explain, Identify, Locate, Recognize, Report, Select, Translate, Paraphrase.
Applying: use the information in a new way	Choose, Demonstrate, Dramatize, Employ, Illustrate, Interpret, Operate, Schedule, Sketch, Solve, Use, Write.
Analyzing: distinguish between the different parts	Appraise, Compare, Contrast, Criticize, Differentiate, Discriminate, Distinguish, Examine, Experiment, Question, Test.
Evaluating: justify a stand or decision	Appraise, Argue, Defend, Judge, Select, Support, Value, Evaluate.
Creating: create new product or point of view	Assemble, Construct, Create Develop, Formulate.

PO No.	Program Outcomes
PO 1	Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems (**Engineering knowledge**).
PO 2	Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences (**Problem analysis**).
PO 3	Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations (**Design/development of solutions**).
PO 4	Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions (**Conduct investigations of complex problems**).

PO 5	Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations (Modern tool usage).
PO 6	Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice (The engineer and society).
PO 7	Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development (Environment and sustainability).
PO 8	Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice (Ethics).
PO 9	Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings (Individual and team work.

PO 10	Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions **(Communication).**
PO 11	Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments **(Project management and finance).**
PO 12	Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change **(Life-long learning).**

Program Specific Outcomes

Graduates will have ability to:

1. Formulate and Evaluate the applications in the field of Intelligent Embedded and Semiconductortechnologies.
2. Focus on the practical experience of ASIC prototype designs, Virtual Instrumentation and SOCdesigns.
3. Build the Embedded hardware design and software programming skills for entry level job positions to meet the requirements of employers.

Course Objectives

Students will try to learn:

I. The fundamental concepts of embedded computing, embedded C, RTOS and embedded software development tools for implementing of real time embedded systems.

II. The embedded C is required to develop the software for different applications of the embedded systems.

III. The basics of various development tools are necessary to develop an embedded software.

IV. The architecture and memory organization of advanced general purpose microprocessors and digital signal processors like ARM and SHARC.

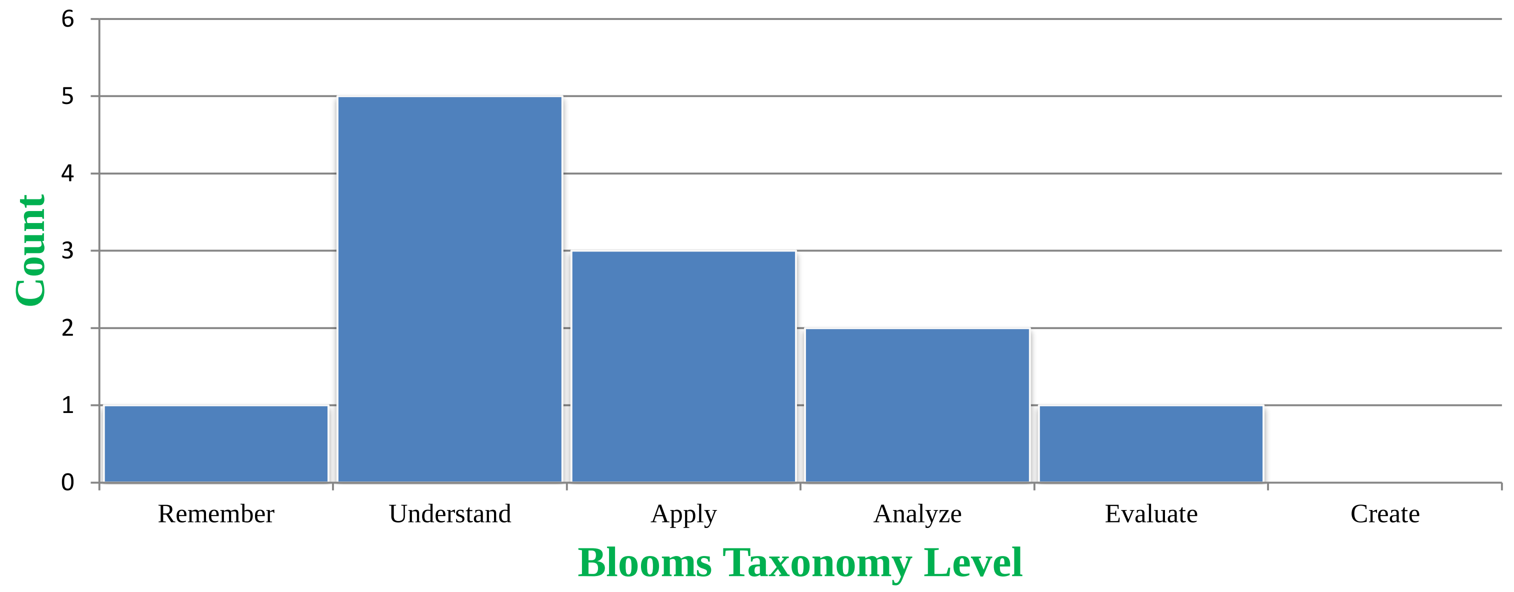
Course Outcomes

After successful completion of the course, students will be able to:		
CO No.	**Course Outcomes**	**Bloom's Knowledge / Cognitive Level**
CO 1	Summarize the applications of embedded systems in various domains.	Understand
CO 2	Analyze the embedded system design process, characteristics and quality attributes of an embedded system.	Analyze
CO 3	Apply the looping structure concept to the programming of embedded C.	Apply
CO 4	Analyze the concepts of interfacing modules using embedded C programming.	Analyze
CO 5	Evaluate the basic techniques used in interfacing in terms of reading and writing data from I/O port pins.	Evaluate
CO 6	Develop the embedded software using the basics and fundamentals of RTOS.	Apply

CO 7	**Demonstrate** the multiprocessing and multitasking in real time operating system to estimate the performance of embedded system.	Understand
CO 8	**Describe** the process of task communication using shared memory and message passing.	Understand
CO 9	**Illustrate** the implementation of real-time operating system using task communication and task synchronization.	Understand
CO 10	**List** the embedded software development tools for getting embedded software into the target system.	Remember
CO 11	**Describe** the concepts of advanced processors in terms of ARM and SHARC processors.	Understand
CO 12	**Explain** the memory organization and instruction level parallelism in advanced processors.	Understand

Distribution of Cognitive Levels

COURSE KNOWLEDGE COMPETENCY LEVELS



Syllabus

MODULE-I	EMBEDDED COMPUTING	Classes: 08
Definition of embedded system, embedded systems vs. general computing systems, history of embedded systems, complex systems and microprocessor, classification, major application areas, the embedded system design process, characteristics and quality attributes of embedded systems, formalisms for system design, design examples		
MODULE-II	**INTRODUCTION TO EMBEDDED C AND APPLICATIONS**	**Classes: 10**
C looping structures, register allocation, function calls, pointer aliasing, structure arrangement, bit fields, unaligned data and endianness, inline functions and inline assembly, portability issues; Embedded systems programming in C, binding and running embedded C program in Keil IDE, dissecting the program, building the hardware; Basic techniques for reading and writing from I/O port pins, switch bounce; Applications: Switch bounce, LED interfacing, interfacing with keyboards, displays, D/A and A/D conversions, multiple interrupts, serial data communication using embedded C interfacing.		
MODULE-III	**RTOS FUNDAMENTALS AND PROGRAMMING**	**Classes: 10**
PART:1 Operating system basics, types of operating systems, tasks and task states, process and threads, multiprocessing and multitasking, how to choose an RTOS ,task scheduling, semaphores and queues, hard real-time scheduling considerations, saving memory and power. **PART:2** Task communication: Shared memory, message passing, remote procedure call and sockets; Task synchronization: Task communication synchronization issues, task synchronization techniques, device drivers.		

Syllabus

MODULA-IV	EMBEDDED SOFTWARE DEVELOPMENT TOOLS	Classes: 09
Host and target machines, linker/locators for embedded software, getting embedded software into the target system; Debugging techniques: Testing on host machine, using laboratory tools, an example system.		
MODULE-V	**INTRODUCTION TO ADVANCED PROCESSORS**	**Classes: 08**
Introduction to advanced architectures: ARM and SHARC, processor and memory organization and instruction level parallelism; Networked embedded systems: Bus protocols, I2C bus and CAN bus; Internet-Enabled systems,		
Text Books:		
1. Shibu K.V, "Introduction to Embedded Systems", Tata McGraw Hill Education Private Limited, 2nd Edition, 2009. 2. Raj Kamal, "Embedded Systems: Architecture, Programming and Design", Tata McGraw-Hill Education, 2nd Edition, 2011. 3. Andrew Sloss, Dominic Symes, Wright, "ARM System Developer's Guide Designing and Optimizing System Software", 1st Edition, 2004.		
Reference Books:		
1. Wayne Wolf, "Computers as Components, Principles of Embedded Computing Systems Design", Elsevier, 2nd Edition, 2009. 2. Frank Vahid, Tony Givargis, "Embedded System Design", John Wiley & Sons, 3 rd Edition, 2006. 3. Michael J. Pont, "Embedded C", Pearson Education, 2nd Edition, 2008.		

Unit-I
EMBEDDED COMPUTING

SYLLABUS:

- Definition of embedded system,
- Embedded systems vs. general computing systems,
- History of embedded systems,
- Complex systems and microprocessor,
- Classification of embedded system,
- Major application areas,
- The embedded system design process,
- Characteristics and quality attributes of embedded systems,
- Formalisms for system design,
- Design examples.

Introduction

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

Embedded Systems

SYSTEM EXAMPLES:

WATCH

- It is a time display SYSTEM
- **Parts**: Hardware, Needles, Battery, Dial, Chassis and Strap

- **Rules**

1. All needles move clockwise only
2. A thin needle rotates every second
3. A long needle rotates every minute
4. A short needle rotates every hour
5. All needles return to the original position after 12 hours

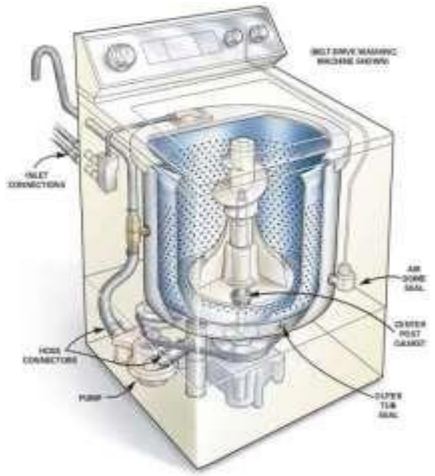
Embedded Systems

SYSTEM EXAMPLES:

WASHING MACHINE

- It is an automatic clothes washing SYSTEM
- **Parts**: Status display panel, Switches & Dials, Motor, Power supply & control unit, Inner water level sensor solenoid valve.
- **Rules**

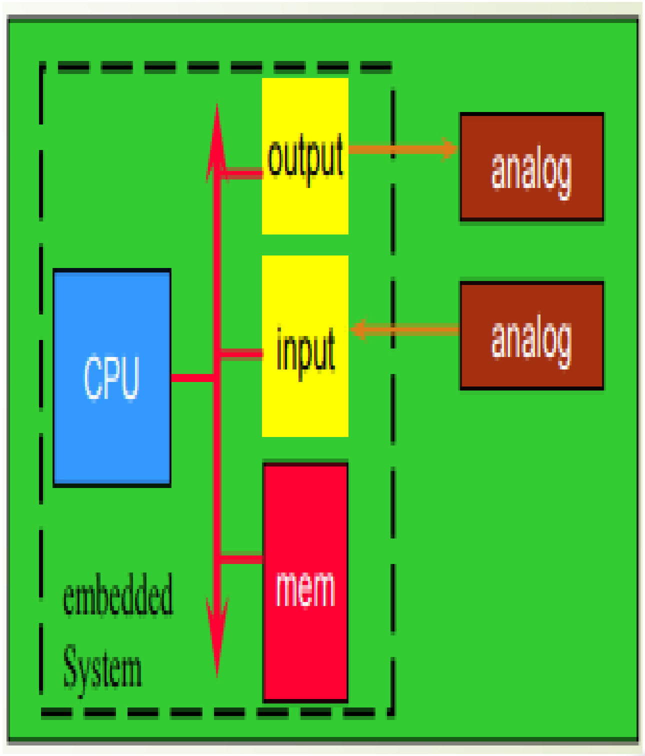
1. Wash by spinning
2. Rinse
3. Drying
4. Wash over by blinking
5. Each step display the process stage
6. In case interruption, execute only the remaining



Embedded Systems

What is an Embedded System?

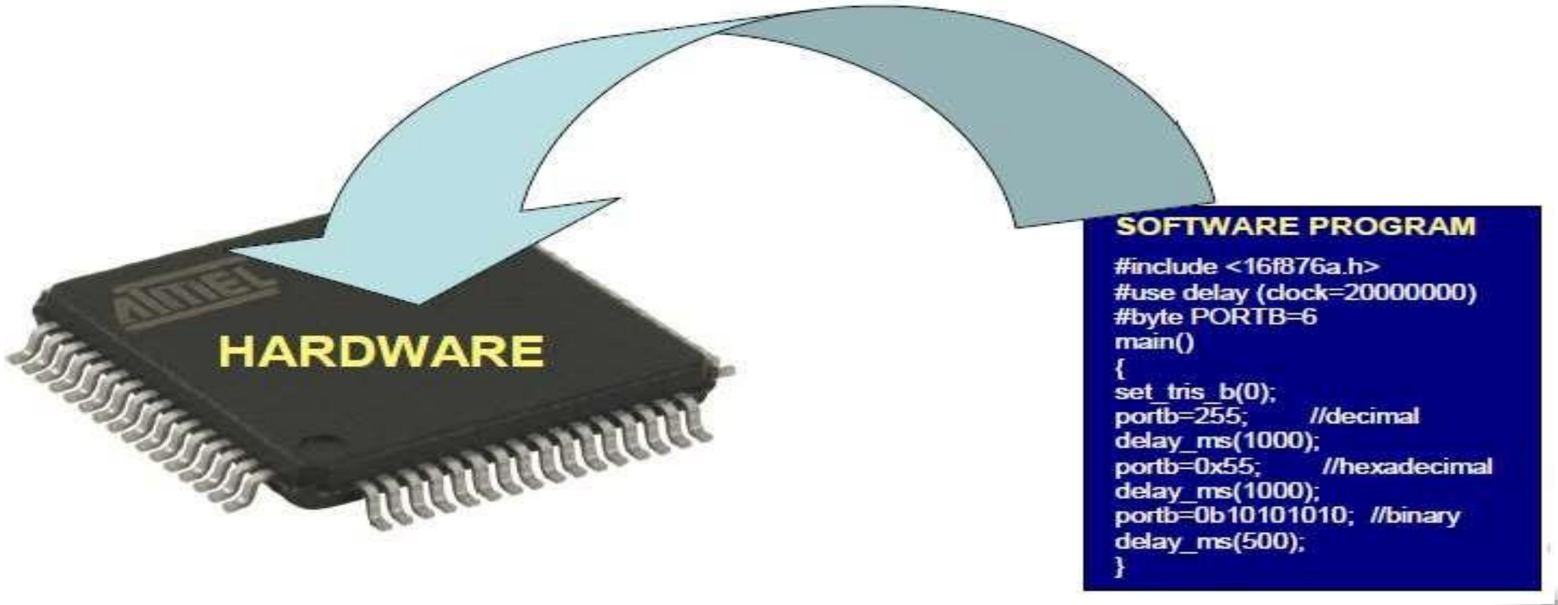
- ❑ An Embedded System is an electronic / electro-mechanical system designed to perform a specific function.
- ❑ And a combination of both hardware and firmware (software).
- ❑ Every E.S is Unique and hardware as well as the firmware is highly specialized to the application domain.
- ❑ ES are becoming an inevitable part of any product or equipment in all fields including household appliances, telecommunications, medical equipment, industrial control, consumer products, etc.



Embedded Systems

EMBEDDED SYSTEM:

- Its software embeds in ROM (Read Only Memory). It does not need secondary memories as in a computer



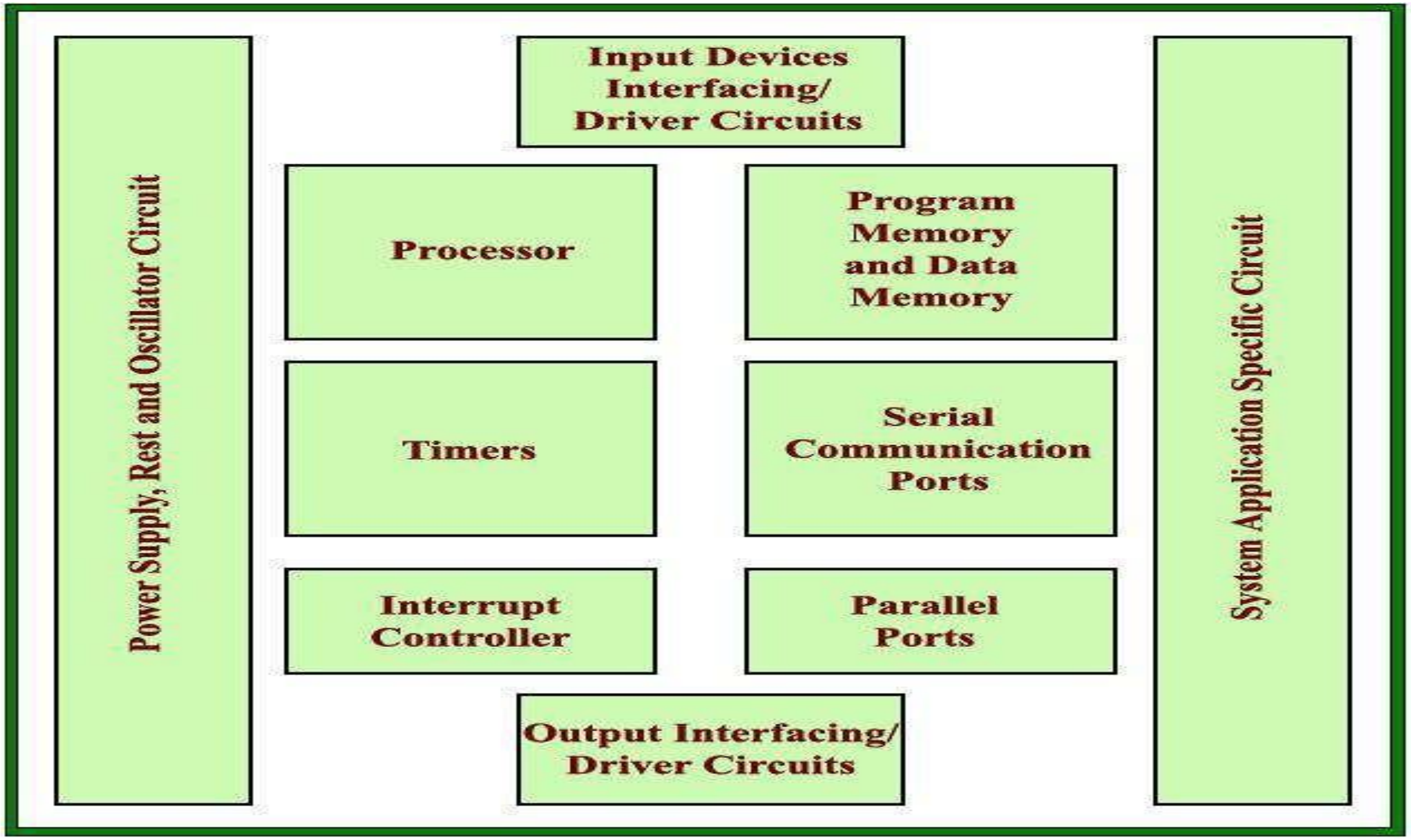
```
#include <16f876a.h>
#use delay (clock=20000000)
#byte PORTB=6
main()
{
set_tris_b(0);
portb=255;        //decimal
delay_ms(1000);
portb=0x55;       //hexadecimal
delay_ms(1000);
portb=0b10101010;  //binary
delay_ms(500);
}
```

COMPONENTS OF EMBEDDED SYSTEM:

1. **It has Hardware**
   - Processor,Timers, Interrupt controller,I/O Devices, Memories,Ports,etc.
2. **It has main Application Software**
   - Which may perform concurrently the series of tasks or multiple tasks.
3. **It has Real Time Operating System (RTOS)**
   - RTOS defines the way the system work, which supervise the application software. It sets the rules during the execution of the application program.A small scale embedded system may not need an RTOS.

Embedded Systems

EMBEDDED SYSTEM HARDWARE:


Power Supply, Rest and Oscillator Circuit
Input Devices Interfacing/ Driver Circuits
Processor
Program Memory and Data Memory
Timers
Serial Communication Ports
Interrupt Controller
Parallel Ports
Output Interfacing/ Driver Circuits
System Application Specific Circuit

Embedded Systems & General Purpose Computers

The **embedded computing requirements** demand 'something special' in terms of

•Response to stimuli,
•Meeting the computational deadlines,
•Power efficiency,
•Limited memory availability, etc.

Embedded Systems & General Purpose Computers

Let's take the case of your personal computer, which may be either a desktop PC or a laptop PC. It is built around a general purpose processor like an Intel® Centrino or a Duo/Quad* core or an AMD Turionm processor and is designed to support a set of multiple peripherals like multiple USB 2.0 ports, Wi-Fi, ethernet, video port, IEEE1394, SD/CF/MMC external interfaces, Bluetooth, etc and with additional interfaces like a CD read/writer, on-board Hard Disk Drive (HDD), gigabytes of RAM, etc.

You can load any supported operating system (like Windows® XP/Vista/7, or Red Hat Linux/Ubuntu Linux, UNIX etc) into the hard disk of your PC. You can write or purchase a multitude of applications for your PC and can use your PC for running a large number of applications.

Embedded Systems & General Purpose Computers

Now let us think about the DVD player you use for playing DVD movies.

Is it possible for you to change the operating system of your DVD?

Is it possible for you to write an application and download it to your DVD player for executing?

Is it possible for you to add a printer soft-ware to your DVD player and connect a printer to your DVD player to take a printout?

Is it possible for you to change the functioning of your DVD player to a television by changing the embedded software?

Embedded Systems & General Purpose Computers

Contents	A system which is a combination of a generic hardware and a General Purpose Operating System for executing **a variety of applications.**	A system which is a combination of special purpose hardware and embedded OS/firmware for executing a **specific set of applications.**
OS	It contains a general purpose operating system (GPOS).	It may or not contain an operating system for functioning.
Alterations	Applications are **alterable** by the user.	Applications are **not-alterable** by the user.
Key factor	**Performance** is a key factor.	**Application specific requirements** are key factors.
Power Consumption	More.	Less.
Response Time	Not critical.	Critical for some applications.
Execution	Need not be deterministic.	Deterministic for certain types of ES like 'Hard Real Time' systems.

History of Embedded Systems

- The first recognized modem embedded system is the **Apollo Guidance Computer (AGC)** developed by the **MIT Instrumentation Laboratory** for the lunar expedition.

- They ran the inertial guidance systems of both the **Command Module (CM)** and the **Lunar Excursion Module (LEM).**

- The **Command Module** was designed to encircle the moon while the **Lunar Module** and its crew were designed to go down to the moon surface and land there safely.

- The Lunar Module featured in total 18 engines. There were 16 reaction control thrusters, a descent engine and an ascent engine.

- The descent engine was *designed to' provide thrust to the lunar module out of the lunar orbit and land it safely on the moon.

History of Embedded Systems

- MIT's original design was based on 4K words of fixed memory (Read Only Memory) and 256 words of erasable memory (Random Access Memory).

- By June 1963, the figures reached 10K of fixed and 1K of erasable memory.

- The final configuration was 36K words of fixed memory and 2K words of erasable memory.

- The clock frequency of the first microchip proto model used in AGC was 1.024 MHz and it was derived from a 2.048 MHz crystal clock.

History of Embedded Systems

- The first mass-produced embedded system was the guidance computer for the **Minuteman-I** missile in 1961.

- It was the **'Autonetics D-17'** guidance computer, built using discrete transistor logic and a hard-disk for main memory.

- The first integrated circuit was produced in September 1958 but computers using them didn't begin to appear until 1963.

Complex Systems and Microprocessors

➢Embedded computer system is a physical system that employs computer control for a specific purpose rather than for general purpose computation. OR

➢Embedded system is a special-purpose computer system designed to perform one or few dedicated functions.

➢Since, the embedded system is dedicated to perform specific tasks, design engineers can optimize it, reducing the size and cost of the product or increasing the reliability and performance.

1	Automotive electronics	6	Military applications
2	Aircraft electronics	7	Authentication
3	Trains	8	Consumer electronics
4	Telecommunication	9	Fabrication equipment
5	Medical systems	10	Robotics

Application areas

Complex Systems and Microprocessors

Examples of Embedded Systems :

Embedded systems have very diversified applications. A few examples of embedded system applications are as follows :

Example1-Digital camera:

- A digital camera is a popular consumer electronic device that can capture images or take pictures and store them in a digital format.

- A digital camera is the best example of Embedded system being very widely used all over the world. It includes a powerful processor capable of handling complex signal processing operations because image capturing, storing and displaying is involved. Below figure shows a general block diagram of a digital camera.

Complex Systems and Microprocessors

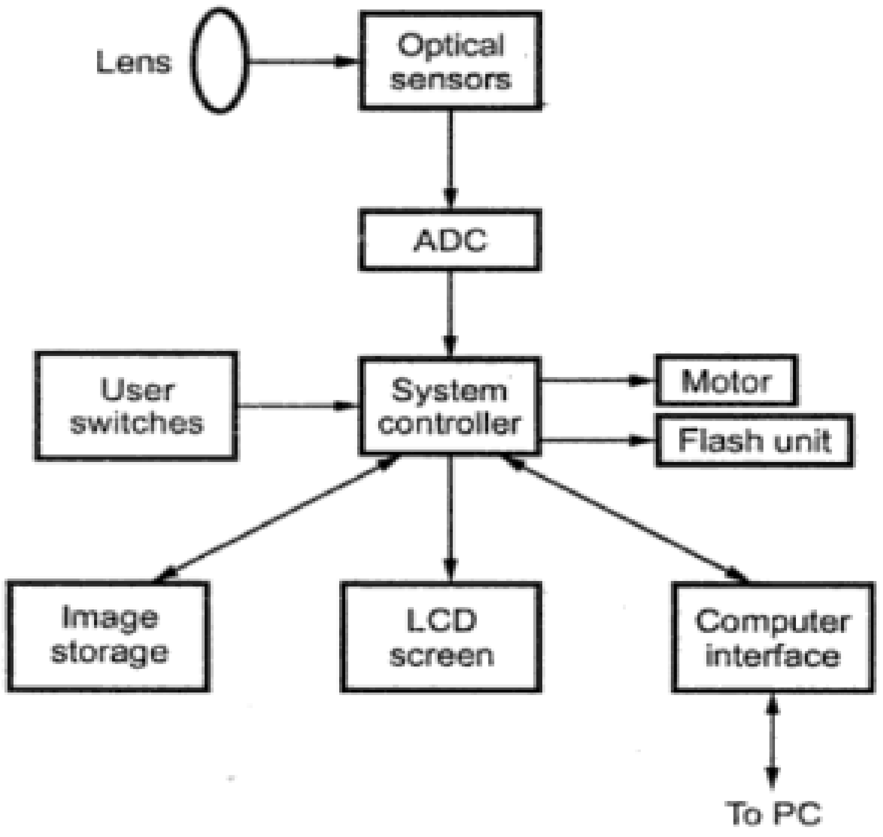


Figure: General block diagram of Digital camera

Complex Systems and Microprocessors

1. In a digital camera, an array of optical sensor is used to capture images. These sensors are the photodiodes which convert light intensity into electrical voltage.

2. An image is made up of picture elements (pixels) of different luminance and chrominance. Each sensing element generates a charge that corresponds to one pixel. Since, the charge is analog quantity, it is converted into digital form using ADC.

3. After ADC, a digital representation of an image is obtained. The colour and intensity of each pixel is represented by number of bits. The digitized image is now considered as a two dimensional matrix having p x q pixels and each pixel is given a equivalent decimal or hexadecimal number.

4. The important functional block of a digital camera is the system Controller. System controller consists of a processor, memory and different interface circuitry to connect to other parts of the system.

Complex Systems and Microprocessors

For compressed images, the format is called JPEG (joint Photographic Experts Group) while for uncompressed images, the format used in BIM (Bit Mapped) or TIFF (Tagged Image File Format).

5. The processed images are stored in Image storage device. Flash memory cards, floppy disks or miniature hard drives can be used for this purpose. The number of images that can be taken and saved depends on the capacity of image storage device.

6. The image is displayed on a LCD screen in the camera. The LCD display normally consumes more power than the processor.

7. A standard computer interface provides a mechanism for transferring images to a computer or a printer.

8. There are some other electromechanical parts such as switches to be operated by the user, motor to rotate the camera for focusing purposes etc. The system controller must be capable generating signals to co-ordinate and control all such activities.

Complex Systems and Microprocessors

Example2-Microwave oven:

Microwave oven is based on a magnetron unit, which generates microwaves used to heat food in a confined space. When turned ON, magnetron generates its maximum power output.

Lower power levels can be obtained by turning the magnetron ON and OFF for controlled time intervals. User can have many options for cooking different dishes using power levels and heating time.

Microwave oven should produce an audio alert signal when cooking operation is completed. It should have an exhaust fan and a light bulb inside. A switch should be provided, which turns the magnetron OFF when the door of the oven is open. All these functions can be controlled by the controller or processor.

Complex Systems and Microprocessors

Microwave oven needs some input/output capability to communicate with the user.

These are :

1. **Input keys :** Consisting of a number pad having digits 0 to 9 for selecting time etc. and function keys such as reset, start, stop, auto defrost, dock set, auto cooking, fan control etc. Some of the functions can be multiplexed to reduce the total number of keys to be provided.

2. Visual output in the form of LCD display.

3. A small speaker that produces the beep tone. Figure shows a simple block diagram of microwave oven.

Complex Systems and Microprocessors

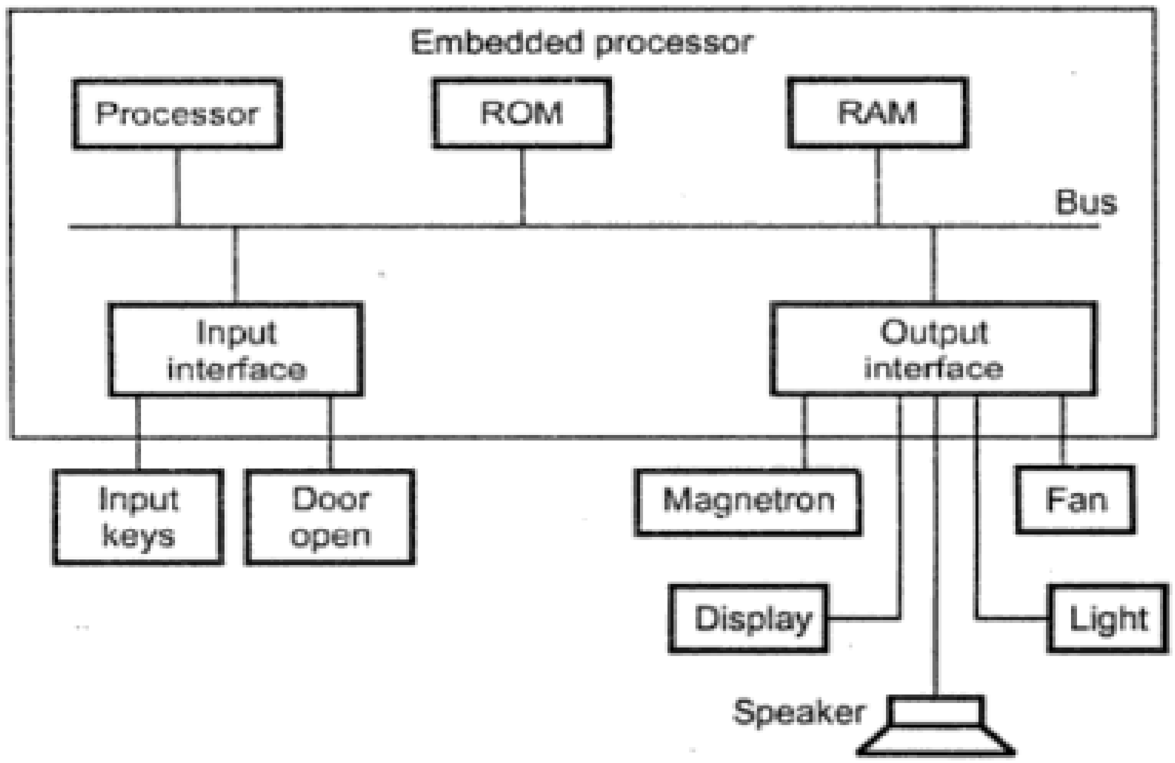


Figure: Block diagram of Microwave oven

Complex Systems and Microprocessors

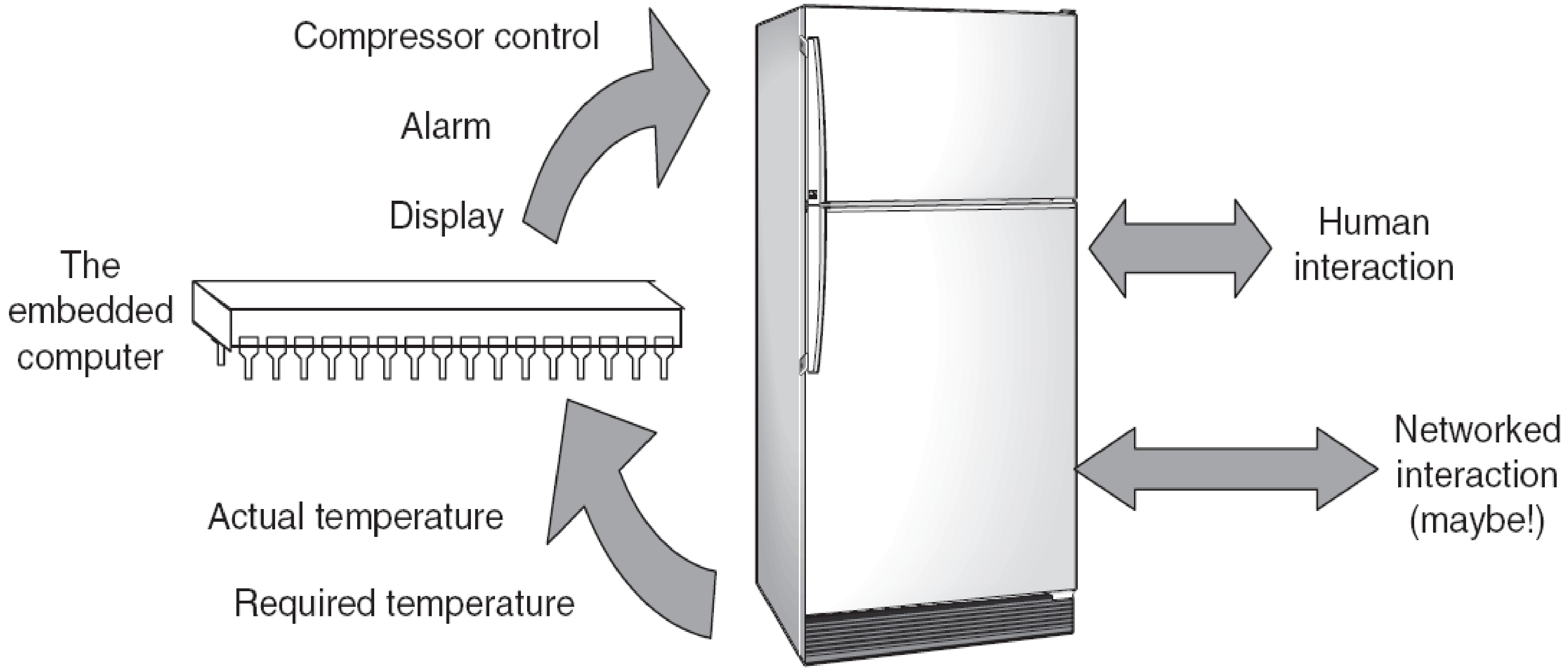
In microwave oven, a simple microprocessor with **small RAM** and **ROM** units are sufficient.

The various operations performed by processor include :

- Maintaining the time-of-day clock,
- Determining the actions required in various cooking options,
- Decrementing timer/counter,
- Generating display information and
- Activating control signals to turn ON or OFF devices such as magnetron and fan.

Complex Systems and Microprocessors

Example3-Refrigerator:



Complex Systems and Microprocessors

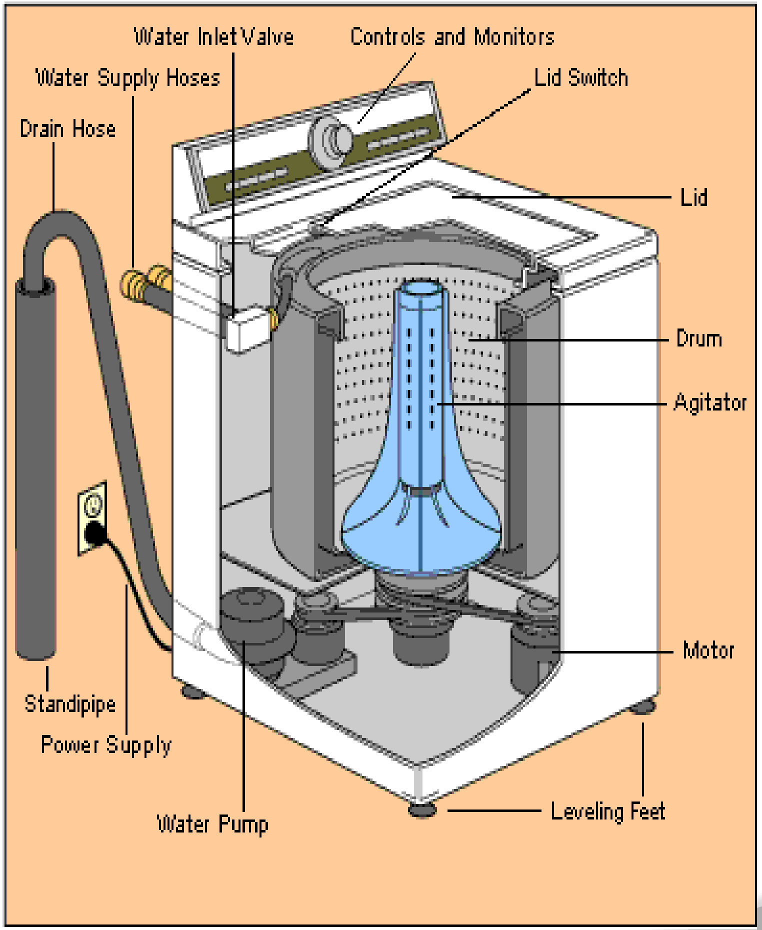
Example4-Washing Machine:

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

•**Hardware:** Buttons, Display & buzzer, electronic circuitry.

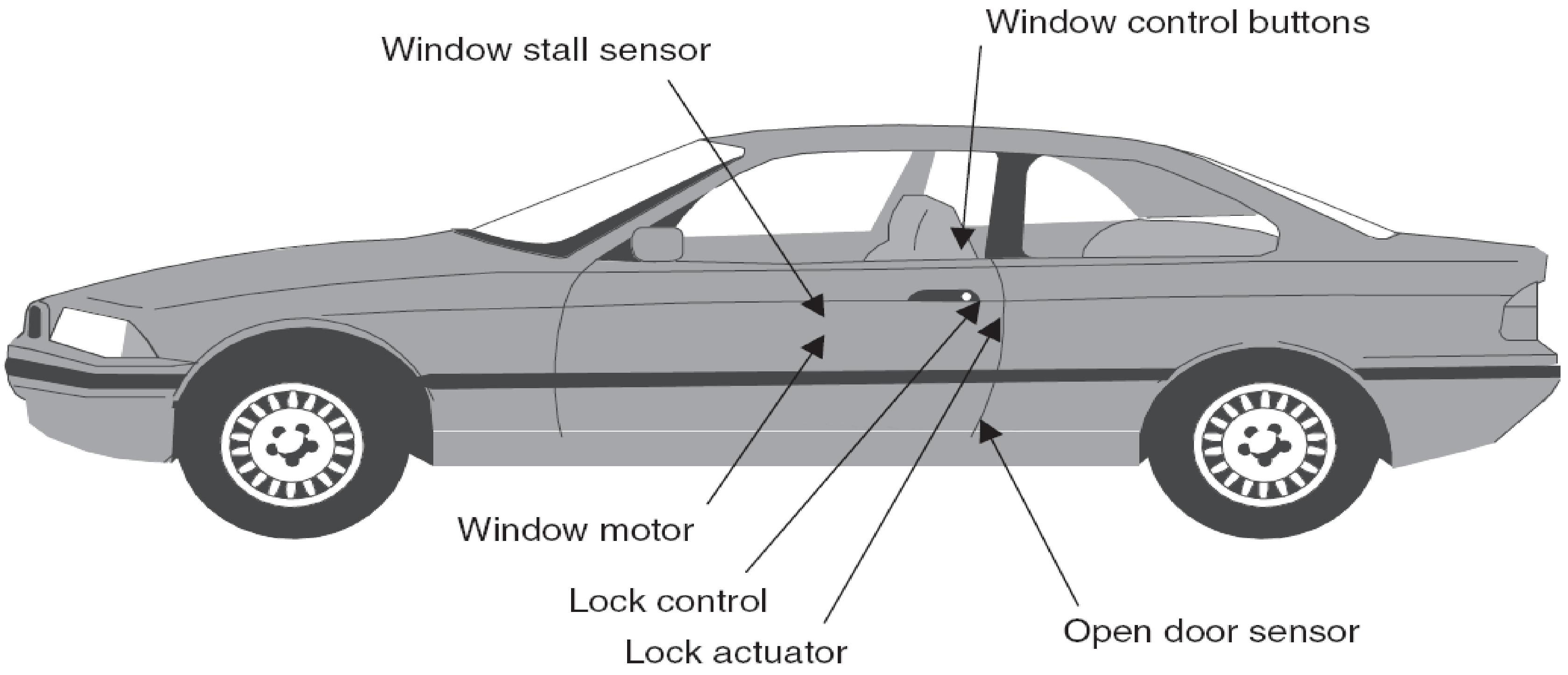
•**Software:** It has a chip on the circuit that holds the software which drives controls & monitors the various operations possible.

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



Complex Systems and Microprocessors

Example5-Car Door:



Complex Systems and Microprocessors

Example6-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

Design metrics

Design issues : The constraints in an embedded systems design are imposed by external as well as internal specifications. Design metrics are introduced to measure the cost function taking into account the economic as well as technical considerations.

Design metrics : A design metric is a measurable feature of the system's performance, cost, time for implementation and safety etc.

Design metrics typically compete with one another : Improving one often leads to worsening of another.

e.g. If the implementation's size is reduced, the implementation's performance may suffer.

Design metrics

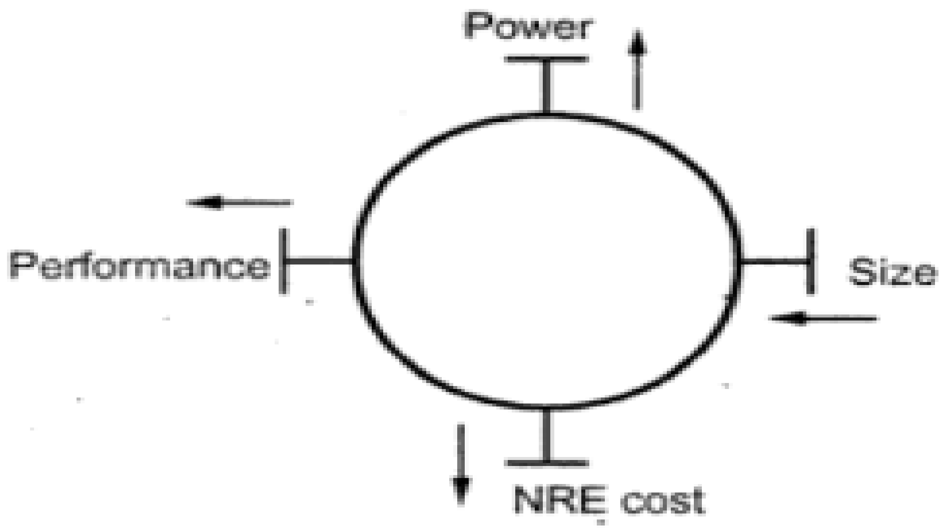


Figure: Design metric competition-improving one may worsen others

Design metrics

i) **NRE (Non-recurring Engineering) cost:** It is the one-time monetary cost of designing the system. Once the system is designed, any number of units can be manufactured without incurring any additional design cost; hence the name non-recurring.

ii) **Unit cost:** It is the monetary cost of manufacturing each copy of the system, excluding NRE cost.

iii) **Size:** It is the physical space required by the system, often measured in bytes for software and gates or transistors for hardware.

iv) **Performance:** It is the execution time of the system.

Design metrics

v) Power consumption: It is the amount of power consumed by the system, which may determine the lifetime of a battery or the cooling requirements of IC, since more power means more heat.

vi) Flexibility: It is the ability to change the functionality of the system without incurring heavy NRE cost Typically software is considered very flexible.

vii) Time-to-prototype: It is the time needed to build a working version of the system, which may be bigger or more expensive than the final system implementation, but it can be used to verify the system's usefulness and correctness and to refine the system's functionality.

Design metrics

viii) Time-to-market: It Is the time required to develop a system to the point that It can be released and sold to customers. This design metric has become especially demanding in recent years Introducing an embedded system to the marketplace early can make a big difference In the systems profitability. The main contributors are design time, manufacturing time and testing time.

ix) Maintainability: It is the ability to modify the system after its initial release, especially by designers who did not originally design the system.

x) Correctness: It is the measure of the confidence that we have implemented the system's functionality correctly. The functionality can be checked throughout the process of designing the system and test circuitry can be Inserted to check that manufacturing was correct.

xi) Safety: It is the probability that the system will not cause harm.

Were the embedded systems existing earlier ?

We have been enjoying the grace of embedded system quite a long time. But they were not so popular because in those days most of the systems were designed around a microprocessor unlike today's systems which were built around a microcontroller.

As we know a microprocessor by itself do not possess any memory, ports etc… So, everything must be connected externally by using peripherals like 8255, 8257, 8259 etc.

So the embedded system designed using microprocessor was not only complicated in design but also large in size. At the same time the speed of a microprocessor is also a Limitation for high end applications.

Major application areas

Major Application Areas of ES

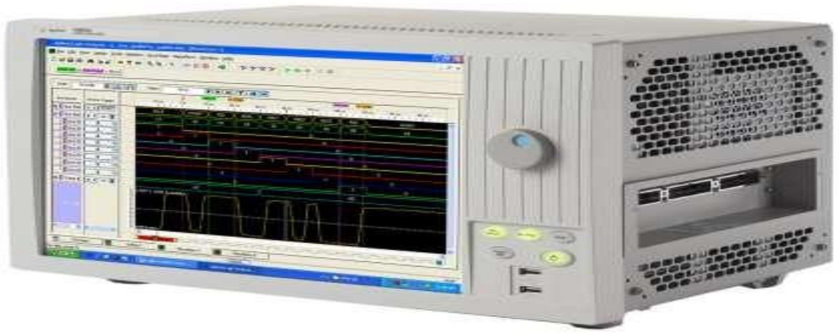
The application areas and the products in the embedded domain are countless.

1. **<u>Consumer electronics</u>:** Camcorders, cameras, etc.
2. **<u>Household appliances</u>:** Television, DVD players, washing machine, refrigerator, microwave oven, etc.
3. **<u>Home automation and security systems</u>:** Air conditioners, sprinklers, intruder detection alarms, closed circuit television cameras, fire alarms, etc.
4. **<u>Automotive industry</u>:** Anti-lock breaking systems (ABS), engine control, ignition systems, automatic navigation systems, etc.
5. **<u>Telecom</u>:** Cellular telephones, telephone switches, handset multimedia applications, etc.

Major application areas

6. **Computer peripherals:** Printers, scanners, fax machines, etc.
7. **Computer Networking systems:** Network routers, switches, hubs, firewalls, etc.
8. **Healthcare:** Different kinds of scanners, ECG , EEG 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.

Major application areas



Logic analyzers PLC systems

Hand Held Devices

Point of Sales (POS)

Classification of Embedded Systems

Classification of Embedded Systems:

I. Based on Generation

II. Based on Complexity & Performance Requirements

III. Based on deterministic behavior

IV. Based on Triggering

Classification of Embedded Systems

I. Classification based on Generation:

First Generation:

The early embedded systems built around 8-bit microprocessors like 8085 and Z80 and 4-bit microcontrollers

EX. Stepper motor control units, Digital Telephone Keypads etc.

Second Generation:

Embedded Systems built around 16-bit microprocessors and 8 or 16-bit microcontrollers, following the first generation embedded systems

EX. SCADA, Data Acquisition Systems etc.

Classification of Embedded Systems

Third Generation:

Embedded Systems built around high performance 16/32 bit Microprocessors/controllers, Application Specific Instruction set processors like Digital Signal Processors (DSPs), and Application Specific Integrated Circuits (ASICs).The instruction set is complex and powerful.

EX. Robotics, industrial process control, networking etc.

Fourth Generation:

Embedded Systems built around System on Chips (SoC's), Reconfigurable processors and multicore processors. It brings high performance, tight integration and miniaturization into the embedded device market

EX. Smart phone devices, Mobile Internet Device (MID)s etc.

Classification of Embedded Systems

II. Classification based on Complexity & Performance:

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

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

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

Classification of Embedded Systems

III. Classification Based on deterministic behavior:

It is applicable for Real Time systems. The application/task execution behavior for an embedded system can be either deterministic or non-deterministic

These are classified into two types

1. **Hard Real time Systems**

2. **Soft Real time systems**

Classification of Embedded Systems

❖A **Real time system** is defined as a data processing system in which the time interval required to process and response to input is so small that it controls the environment.

❖The time taken from input to output task is called **response time**.

❖RTOS responses to input immediately (real time)

❖Here the task is completed with in a specified time delay.

❖In real life situations like controlling traffic signals or a nuclear reactor or an aircraft.

❖The OS has to respond quickly.

Classification of Embedded Systems

A system is said to be **Real Time** if it is required to complete its work and deliver its service on time.

Example: Flight Control System.

Types of Real Time Systems:

1. Hard Real time Systems.
2. Soft Real time systems.

Classification of Embedded Systems

1. **Hard Real time Systems:**

   Hard real time system is **purely deterministic** and **time constraint system**.

   Ex: **Missile Launching system.**

   Satellite system.

   Air Bag controlling and Anti Lock Braking (A.B.S) system in car.

2. **Soft Real time systems:**

   In Soft real time system, the meeting of deadline is not compulsory for every time for every task but process should get completed and give the result.

   Ex: **Personal Computer, Cameras, Smart Phones, online transaction, online bid in stock exchange, Audio and Video System, etc.**

Classification of Embedded Systems

- **Characteristics of RTOS:**
  - Compactness
  - Reliability
  - Predictability
  - Performance
  - Scalability

- **Functions of RTOS:**
  - Task management
  - Scheduling
  - Resource allocation
  - Interrupt Handling

Classification of Embedded Systems

IV. Classification Based on Triggering:

These are classified into two types

1. <u>Event Triggered :</u>

•In an event triggered system a processing activity is initiated as a consequence of the occurrence of a significant event.

•Activities within the system (e.g., task run-times) are dynamic and depend upon occurrence of different events .

2. <u>Time triggered:</u>

•In a time-triggered system, the activities are initiated periodically at predetermined points in real-time.

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

Embedded System Design Process

Embedded System Design Process:

Design process is a sequence of steps necessary to build a product.

An embedded system is designed keeping in view three constraints :

1. Available system memory.
2. Available processor speed.
3. Need to limit power dissipation when running the system continuously in cycles of wait, run, stop, wake-up and sleep.

There are two approaches of embedded system design process : **bottom-up design** and **top-down design.**

In bottom-up design, we start with components to build a system. In top-down design, we start with abstraction of the system and then the details are created.

Embedded System Design Process

There are basically five steps which describe every design problem, be it mathematical, physical, engineering or any other.

These steps are :

1. Requirement definition.
2. System specification.
3. System architecture development.
4. Components.
5. System integration.

Embedded System Design Process

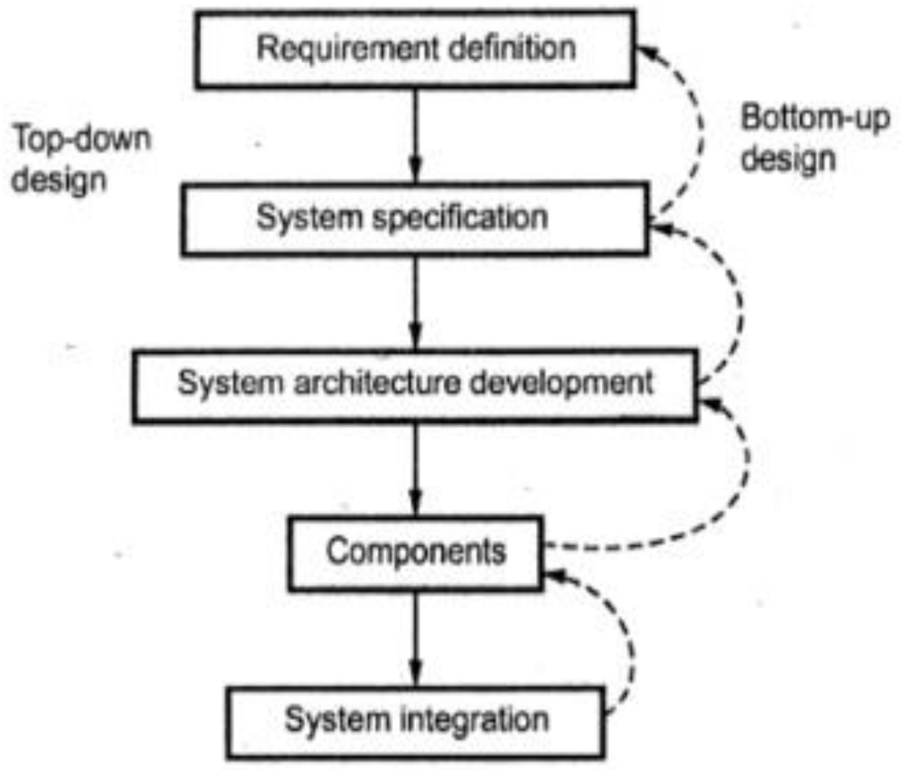


Figure: Embedded System Design Process

Embedded System Design Process

Requirements:

“Requirements” are the informal descriptions of what the customer wants. At this stage, the product can be viewed as a black box. What happens inside is unimportant ?

Types of Requirements:

1. Functional:

A functional requirement states what the system must do.

2. Non- Functional:

A non-functional requirement can be other attributes, including performance, cost, size and power consumption, etc.

Embedded System Design Process

Non-functional requirements include :

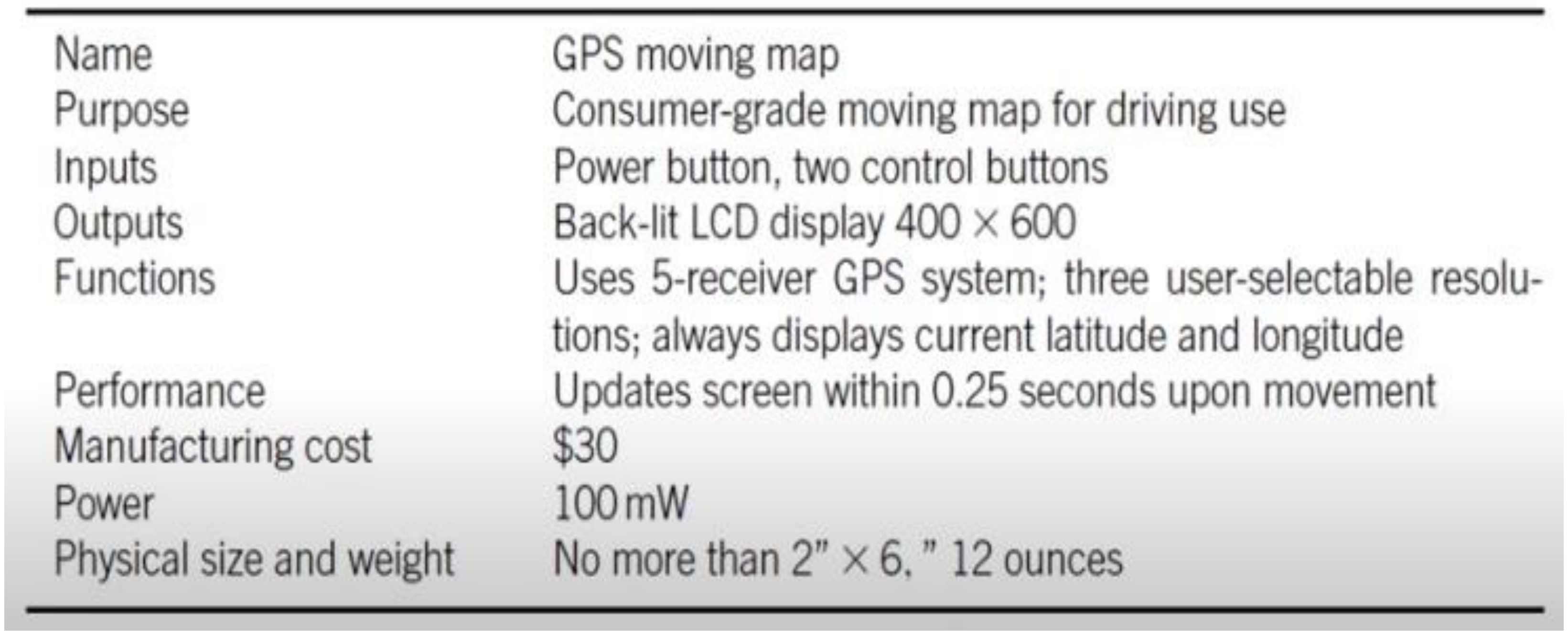
1. **Performance:** The instruction execution time or throughput of the system.
2. **Cost:** Cost has typically two major components :
   i) Manufacturing cost : The cost of components and assembly.
   ii) Non-recurring engineering cost The monetary cost of designing the system.
3. **Size:** The physical space required by the system, often measured in bytes for software and gates or transistors for hardware.
4. **Power consumption:** The amount of power consumed by the system, which determines the lifetime of a battery.

Embedded System Design Process

Requirements should meet the following :

1	Correctness	:	Requirement should describe what the customer wants
2	Unambiguousness	:	Requirements must be clear
3	Completeness	:	All requirements should be included
4	Verifiability	:	Each requirement should be satisfied
5	Consistency	:	One requirement should not contradict another requirement .
6	Modifiability	:	Requirement document should be modifiable in case of changing requirements.

Embedded System Design Process



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

Embedded System Design Process

Specifications:

➢Specifications need to be more precise. Specifications guide customer expectations from the product.

➢They also guide system architecture. Designer needs specifications for hardware specifications, data types and processing specifications, system behavior specifications, design constraints, life cycle specifications.

➢Process specifications are analyzed by making lists of inputs, outputs and how the processes activate on each input.

➢If global characteristics of the specification are wrong or incomplete, the overall system architecture derived from the specification may be inadequate to meet the needs of implementation.

Embedded System Design Process

A specification of the GPS system would include several components:

■ Data received from the GPS satellite constellation.

■ Map data.

■ User interface.

■ Operations that must be performed to satisfy customer requests.

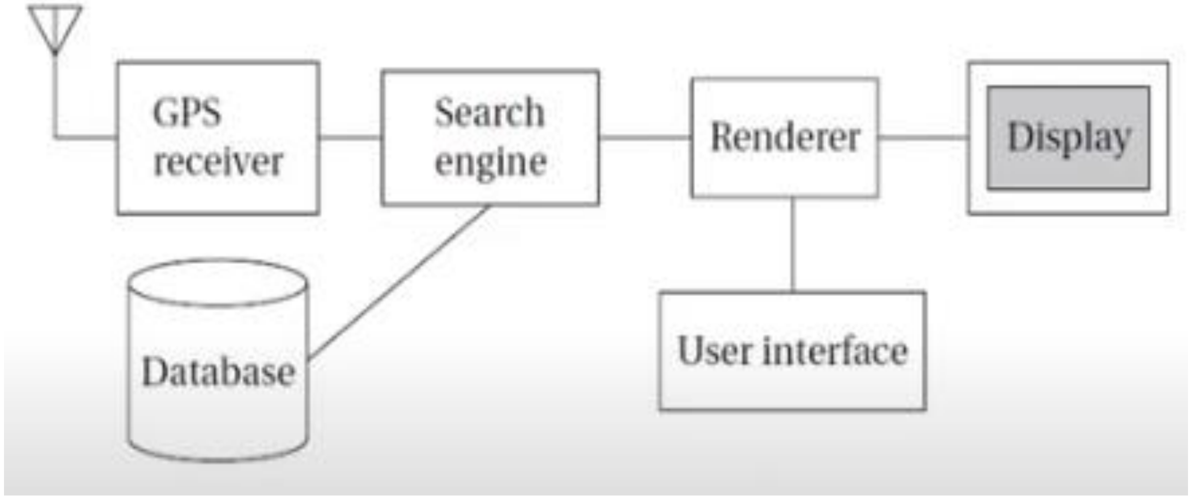
■ Background actions required to keep the system running, such as

operating the GPS receiver.

Embedded System Design Process

Architecture:

Specification only describes what the system does while architecture describes how the system implements those functions. Architecture is a plan for overall system structure that will be used to design the components that make the architecture.



Embedded System Design Process

Components:

The architectural description gives us the idea about the required components. Basically, there are two types of components namely hardware and software.

Common hardware components are processor, memory, peripherals and devices, ports and buses and battery in the system. During software development process, components can be modeled as object oriented software.

Embedded System Design Process

System Integration:

- After the components are built, they are integrated in the system. But after integration, these components may not fulfill the design metrics.

- The system is made to function and validated by choosing appropriate tests.

- Debugging tools are used to correct the erroneous functioning.

CHARACTERISTICS & QUALITY ATTRIBUTES OF EMBEDDED SYSTEMS

CHARACTERISTICS OF EMBEDDED SYSTEMS

CHARACTERISTICS OF EMBEDDED SYSTEMS:

1. Application & Domain specific.
2. Reactive & Real time.
3. Operates in harsh environments.
4. Distributed.
5. Small size & weight.
6. Power concerns.

CHARACTERISTICS OF EMBEDDED SYSTEMS

Application and Domain specific

- An embedded system is designed for a specific purpose only. It will not do any other task.

- Ex. A washing machine can only wash, it cannot cook.

- Certain embedded systems are specific to a domain.

- Ex. A hearing aid is an application that belongs to the domain of signal processing.

CHARACTERISTICS OF EMBEDDED SYSTEMS

Reactive and Real time

➢Certain Embedded systems are designed to react to the events that occur in the nearby environment. These events also occur real-time.

➢Ex. An air conditioner adjusts its mechanical parts as soon as it gets a signal from its sensors to increase or decrease the temperature when the user operates it using a remote control.

➢An embedded system uses Sensors to take inputs and has actuators to bring out the required functionality.

CHARACTERISTICS OF EMBEDDED SYSTEMS

Operation in harsh environment

- Certain embedded systems are designed to operate in harsh environments like very high temperature of the deserts or very low temperature of the mountains or extreme rains.

- These embedded systems have to be capable of sustaining the environmental conditions it is designed to operate in.

CHARACTERISTICS OF EMBEDDED SYSTEMS

Distributed

➢Certain embedded systems are part of a larger system and thus form components of a distributed system.

➢These components are independent of each other but have to work together for the larger system to function properly.

➢Ex. A car has many embedded systems controlled to its dash board. Each one is an independent embedded system yet the entire car can be said to function properly only if all the systems work together.

CHARACTERISTICS OF EMBEDDED SYSTEMS

Small size and weight

- An embedded system that is compact in size and has light weight will be desirable or more popular than one that is bulky and heavy.

- Ex. Currently available cell phones. The cell phones that have the maximum features are popular but also their size and weight is an important characteristic.

- For convenience users prefer mobile phones than phablets. (phone + tablet pc)

CHARACTERISTICS OF EMBEDDED SYSTEMS

Power concerns

- It is desirable that the power utilization and heat dissipation of any embedded system be low.

- If more heat is dissipated then additional units like heat sinks or cooling fans need to be added to the circuit.

- If more power is required then a battery of higher power or more batteries need to be accommodated in the embedded system.

Quality Attributes of Embedded Systems

These are the attributes that together form the deciding factor about the quality of an embedded system.
There are **two types** of quality attributes are:-

1. Operational Quality Attributes.
These are attributes related to operation or functioning of an embedded system. The way an embedded system operates affects its overall quality.

i. Response
ii. Throughput
iii. Reliability
iv. Maintainability
v. Security
vi. Safety

Operational Quality Attributes

(i).Response

- Response is a measure of quickness of the system.
- It gives you an idea about how fast your system is tracking the input variables.
- Most of the embedded system demand fast response which should be real-time.

(ii). Throughput

- Throughput deals with the efficiency of system.
- It can be defined as rate of production or process of a defined process over a stated period of time.
- In case of card reader like the ones used in buses, throughput means how much transaction the reader can perform in a minute or hour or day.

Operational Quality Attributes

(iii). Reliability

•Reliability is a measure of how much percentage you rely upon the proper functioning of the system.

•**Mean Time between failures** and **Mean Time To Repair** are terms used in defining system reliability.

•**Mean Time between failures** can be defined as the average time the system is functioning before a failure occurs.

•**Mean time to repair** can be defined as the average time the system has spent in repairs.

Operational Quality Attributes

(iv). Maintainability

- Maintainability deals with support and maintenance to the end user or a client in case of technical issues and product failures or on the basis of a routine system checkup

- It can be classified into two types :-
- **Scheduled or Periodic Maintenance**
  - This is the maintenance that is required regularly after a periodic time interval.
  - **Example** : Periodic Cleaning of Air Conditioners
    Refilling of printer cartridges.
- **Maintenance to unexpected failure**
  - This involves the maintenance due to a sudden breakdown in the functioning of the system.
  - **Example:** Air conditioner not powering on
    Printer not taking paper in spite of a full paper stack

Operational Quality Attributes

(v). Security

- Confidentiality, Integrity and Availability are three corner stones of information security.

- Confidentiality deals with protection data from unauthorized disclosure.

- Integrity gives protection from unauthorized modification.

- Availability gives protection from unauthorized user.

- Certain Embedded systems have to make sure they conform to the security measures.
- Ex. An **Electronic Safety Deposit Locker** can be used only with a pin number like a password.

Operational Quality Attributes

(vi). Safety

•Safety deals with the possible damage that can happen to the operating person and environment due to the breakdown of an embedded system or due to the emission of hazardous materials from the embedded products.

•A safety analysis is a must in product engineering to evaluate the anticipated damage and determine the best course of action to bring down the consequence of damages to an acceptable level.

Quality Attributes of Embedded Systems

2. Non-operational Quality attributes

- These are attributes **not** related to operation or functioning of an embedded system. The way an embedded system operates affects its overall quality.

- These are the attributes that are associated with the embedded system before it can be put in operation.

i. Testability & Debugability

ii. Evolvability

iii. Portability

iv. Time to prototype & market

v. Per unit & per cost

Non-Operational Quality Attributes

(i). Testability and Debugability

- It deals with how easily one can test his/her design, application and by which mean he/she can test it.
- In hardware testing the peripherals and total hardware function in designed manner.
- Firmware testing is functioning in expected way.
- Debugability is means of debugging the product as such for figuring out the probable sources that create unexpected behavior in the total system.

(ii). Evolvability

- For embedded system, the qualitative attribute "Evolvability" refer to ease with which the embedded product can be modified to take advantage of new firmware or hardware technology.

Non-Operational Quality Attributes

(iii). Portability

- Portability is measured of “system Independence”.
- An embedded product can be called portable if it is capable of performing its operation as it is intended to do in various environments irrespective of different processor and or controller and embedded operating systems.

(iv). Time to prototype and market

- Time to Market is the time elapsed between the conceptualization of a product and time at which the product is ready for selling or use
- Product prototyping help in reducing time to market.
- Prototyping is an informal kind of rapid product development in which important feature of the under consider are develop.
- In order to shorten the time to prototype, make use of all possible option like use of reuse, off the self component etc.

Non-Operational Quality Attributes

(v). Per unit and total cost

- Cost is an important factor which needs to be carefully monitored. Proper market study and cost benefit analysis should be carried out before taking decision on the per unit cost of the embedded product.
- When the product is introduced in the market, for the initial period the sales and revenue will be low
- There won’t be much competition when the product sales and revenue increase.
- During the maturing phase, the growth will be steady and revenue reaches highest point and at retirement time there will be a drop in sales volume.

Formalisms for System Design

Formalisms for System Design:

➢There are different design tasks at different levels of abstraction 'which can be conceptualized in diagrams.

➢Unified Modelling Language (UML) is a textual and graphical notation i.e. language, used to quantify and formalize our understanding of systems.

➢UML has its roots in a branch of computer software development industry known as Object Oriented Analysis and Design (OOA&D) but has applications beyond this field-especially in the field of understanding and documenting business processes.

Formalisms for System Design

Goals of UML :

The primary goals in the design of UML were :

1. Provide users with a ready-to-use, expressive visual modelling language so that they can develop and exchange meaningful models.
2. Provide extensibility and specialization mechanisms to extend the core concepts.
3. Be independent of particular programming languages and development processes.
4. Provide a formal basis for understanding the modelling language.
5. Support higher level development concepts such as collaborations, frameworks, patterns and components.
6. Integrate best practices.

Formalisms for System Design

Structural Description:

Structural description mean basic components of the system.

(i) Class Diagram:

•A class diagram is a diagram showing a collection of classes and interfaces, along with the collaborations and relationships among classes and interfaces. A class diagram is a pictorial representation of detailed system design. Design experts who understand the rules of modelling and designing systems design the system's class diagrams.

•A class diagram is composed primarily of following elements that represent the system's entities :

a. Class :

•A class represents an entity of a given system that provides an encapsulated implementation of certain functionality of a given entity. These are exposed by the class to other classes as methods. The properties of a class are called attributes.

•A class is represented by a rectangle.

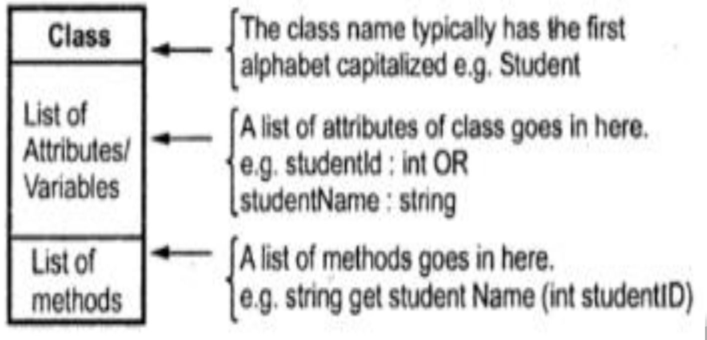

Class
List of Attributes/ Variables
List of methods
The class name typically has the first alphabet capitalized e.g. Student
A list of attributes of class goes in here. e.g. studentId : int OR studentName : string
A list of methods goes in here. e.g. string get student Name (int studentID)

Fig. Structure of a class

Formalisms for System Design

b. Interface : An interface is a variation of class. An interface provides only a definition of business functionality of a system. A separate class implements the actual business functionality.

c. Package : A package provides the ability to group together classes and/or interfaces that are either similar in nature or related. Grouping these design elements in a package element provides for better readability of class diagrams, especially complex class diagrams.

Formalisms for System Design

(ii) Object Diagram:

An object diagram is a pictorial representation of the relationships between instantiated classes at any point of time (called objects). A class diagram for multiplicity relation between college and students is as shown below :

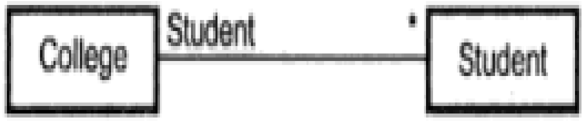


Figure: An example college-student class diagram

This class diagram shows that many students can study in a college. If we were to add attributes to the classes "college" and "student", then it would be as shown below.

Formalisms for System Design

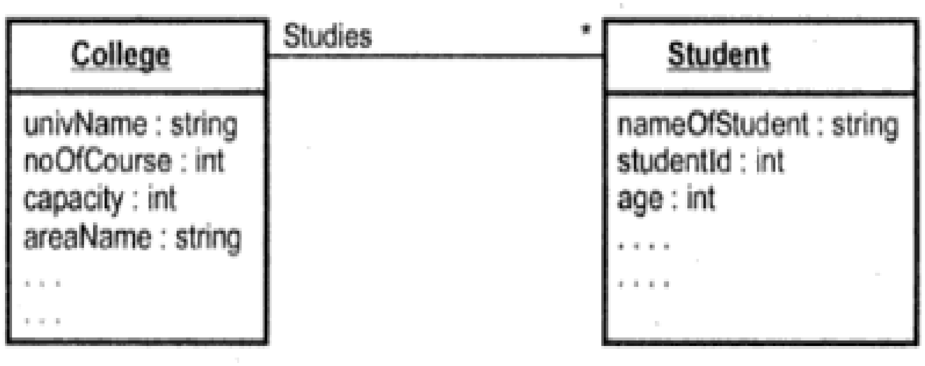


Figure: Class diagram with attribute

Now, when an application with class diagram as shown above is run, instances of college and student class will be created with values of attributes initialized.

Formalisms for System Design

The object diagram for such scenario will be as represented below :

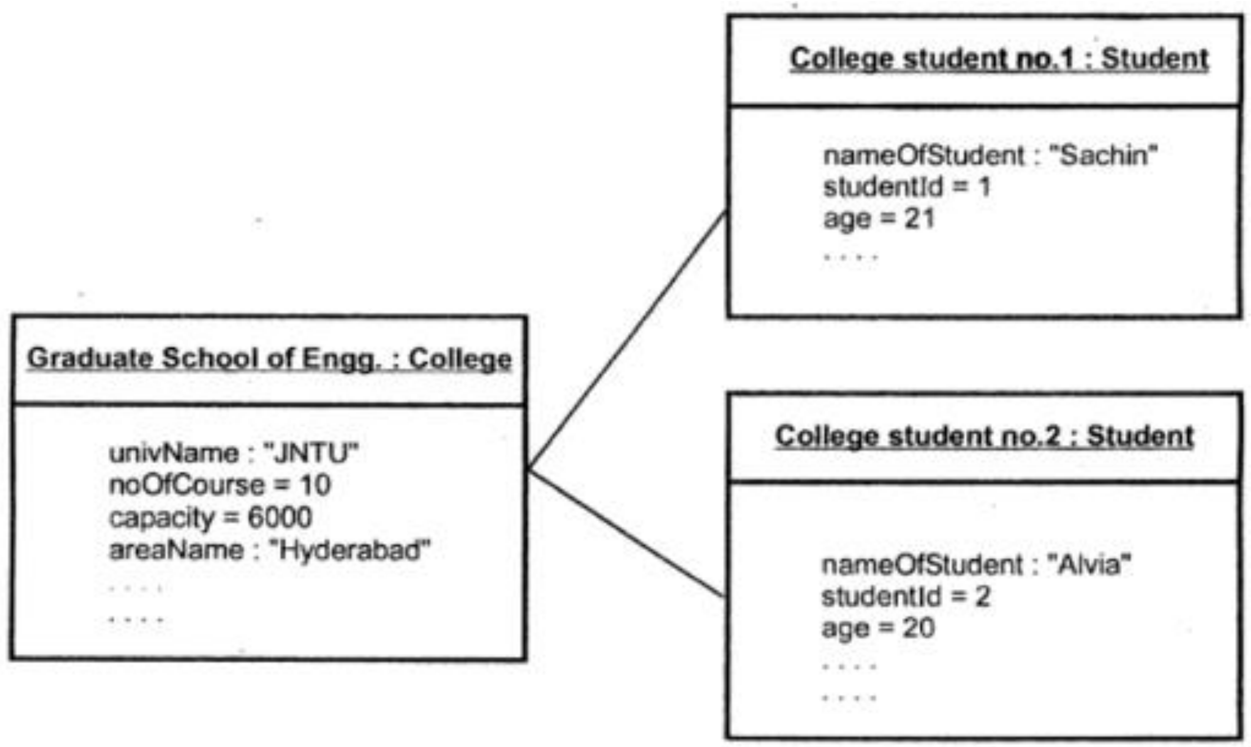
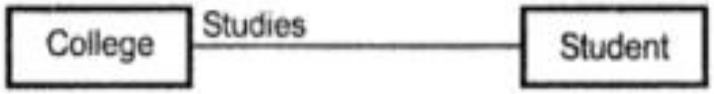


Figure: Object diagram for college-student class diagram

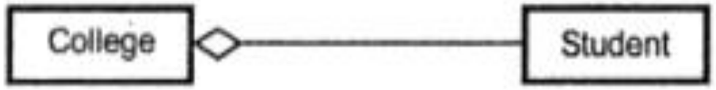
Formalisms for System Design

There are several types of relationships that can exist between objects and classes :

1. **Association** : When two classes are connected to each other in any way, an association relation is established.

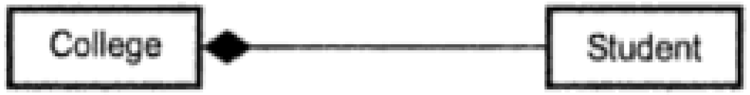


2. **Aggregation** : When a complex class is formed as a collection of smaller classes, it is called an aggregation relation between these classes.

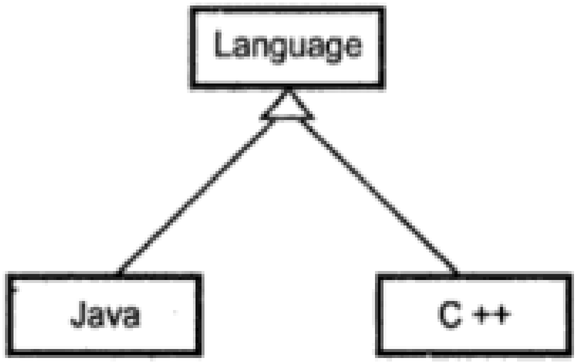


Formalisms for System Design

3. **Composition :** It is a type of aggregation in which the owner does not allow access to component class.



4. **Inheritance / Generalization :** It is the basic type of relation used to define reusable elements in class diagram.



Formalisms for System Design

1.4.2 Behavioral Description

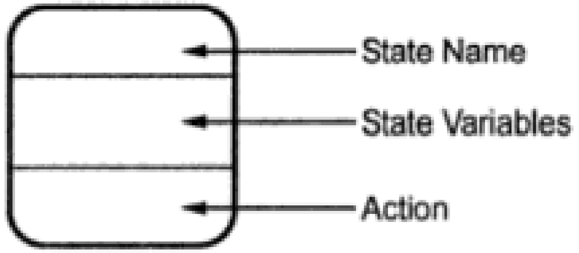
State machine is used to specify the behaviour of the system. A state machine (diagram) consists of following behavioural elements :

i) Initial state : This shows the starting point or first activity of the flow. This is also called a pseudo state, where the state has no variables describing it further and no activities.

Symbol of initial state is ● :

ii) State : Represents the state of object at an instant of time. In a state diagram, there will be multiple of such symbols, one for each state of object.

Symbol of state is :



Formalisms for System Design

iii) Transition : An arrow indicating the object to transition from one state to other. The actual trigger event and action causing the transition are written beside the arrow separated by a slash. Transitions that occur because the state completed an activity are called "triggerless" transitions. If an event has to occur after the completion of some event or action, the event or action is called the guard condition. The transition takes place after the guard condition occurs.

Symbol is : $\xrightarrow{\text{Event / Action}}$

iv) History states : A flow may require that the object go into a trance, or wait state, and on the occurrence of a certain event, go back to the state it was in when it went into a wait state - its last active state.

Symbol is : Ⓗ

Formalisms for System Design

v) Event and action : A trigger that causes a transition to occur is called as an event or action.

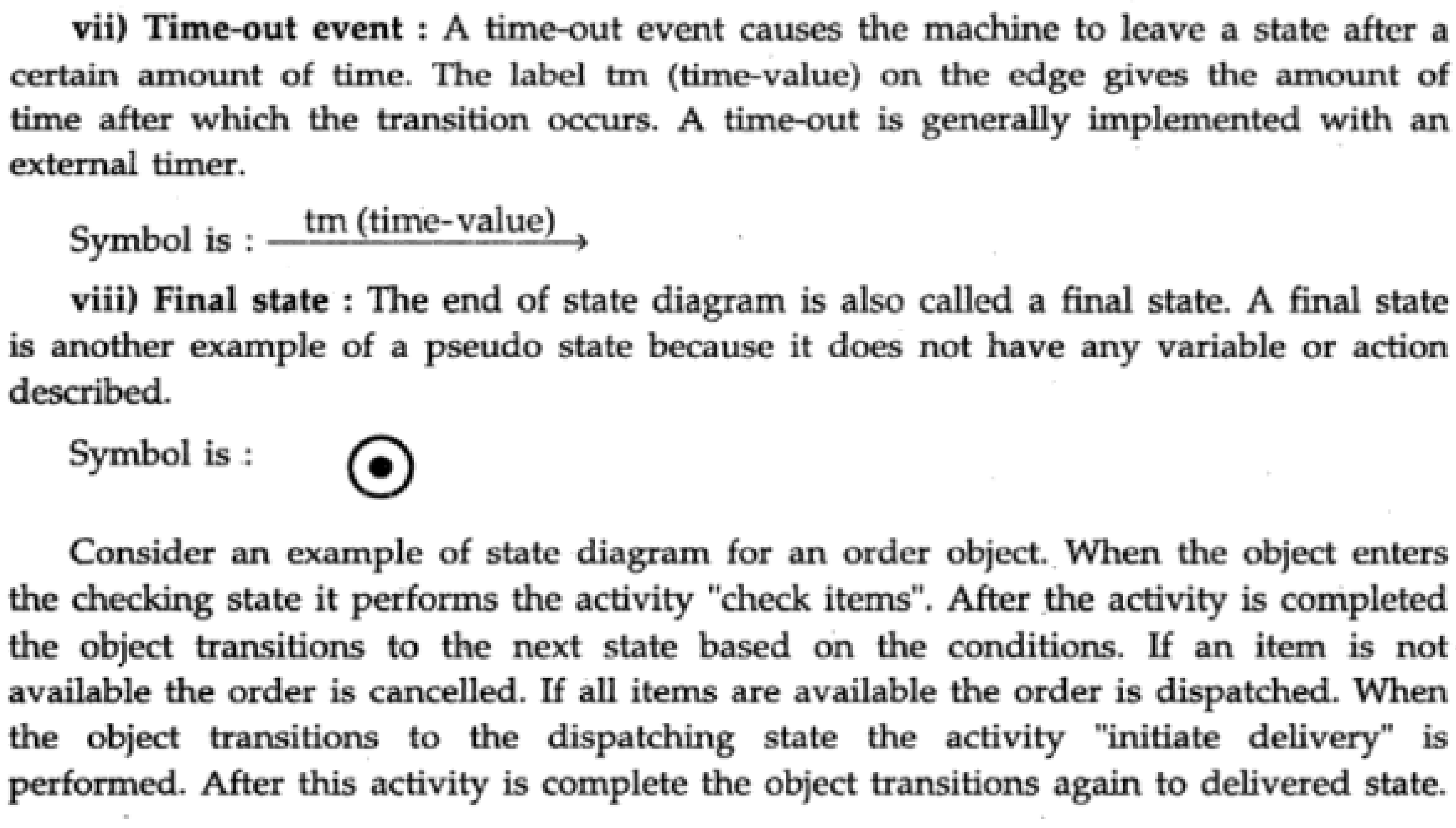
Symbol is : $\xrightarrow{\text{Event / Action}}$

vi) Signal : When an event causes a message / trigger to be sent to a state that causes the transition, then that message sent by the event is called signal.

Symbol is :

$$\boxed{\begin{array}{c} \text{<< Signal >>} \\ \text{Event / Action} \end{array}}$$

Formalisms for System Design



vii) Time-out event : A time-out event causes the machine to leave a state after a certain amount of time. The label tm (time-value) on the edge gives the amount of time after which the transition occurs. A time-out is generally implemented with an external timer.

Symbol is : $\xrightarrow{\text{tm (time-value)}}$

viii) Final state : The end of state diagram is also called a final state. A final state is another example of a pseudo state because it does not have any variable or action described.

Symbol is : ⦿

Consider an example of state diagram for an order object. When the object enters the checking state it performs the activity "check items". After the activity is completed the object transitions to the next state based on the conditions. If an item is not available the order is cancelled. If all items are available the order is dispatched. When the object transitions to the dispatching state the activity "initiate delivery" is performed. After this activity is complete the object transitions again to delivered state.

Formalisms for System Design

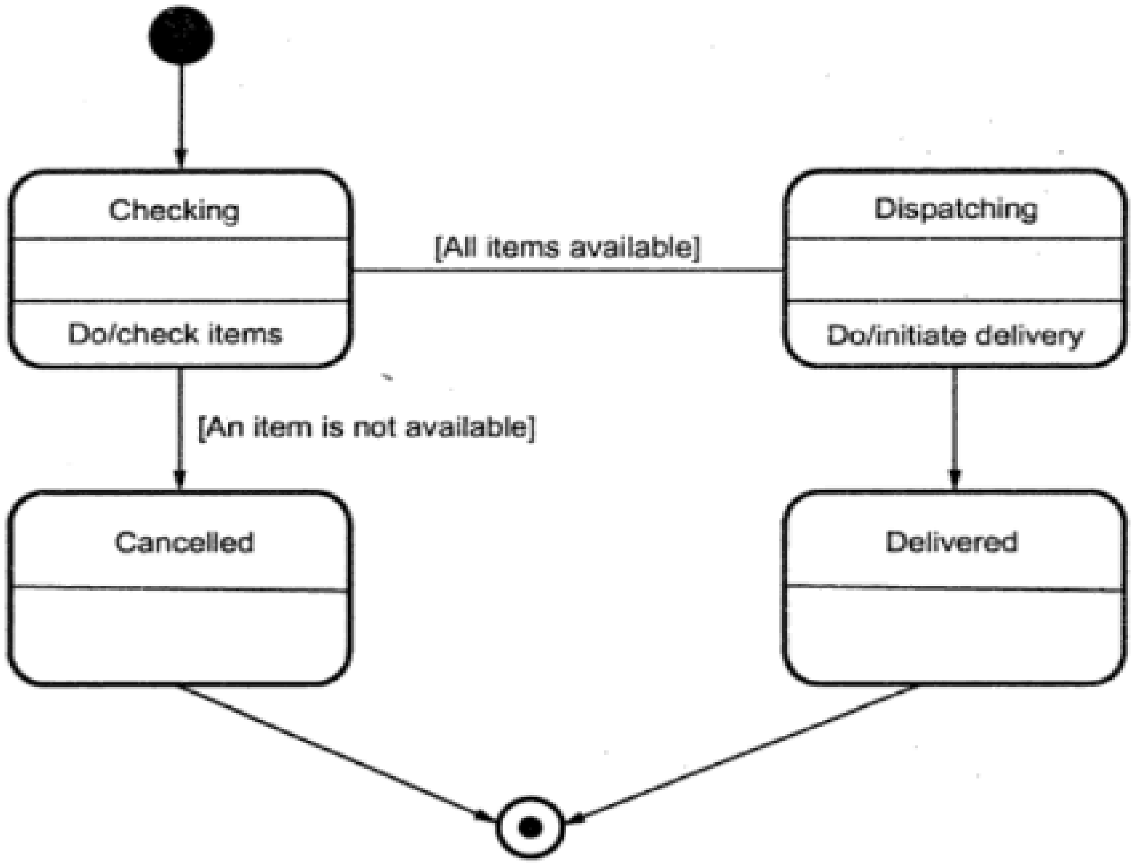


Fig. 1.11 State diagram for order object

Design example-Model Train Controller

Model Train Controller:

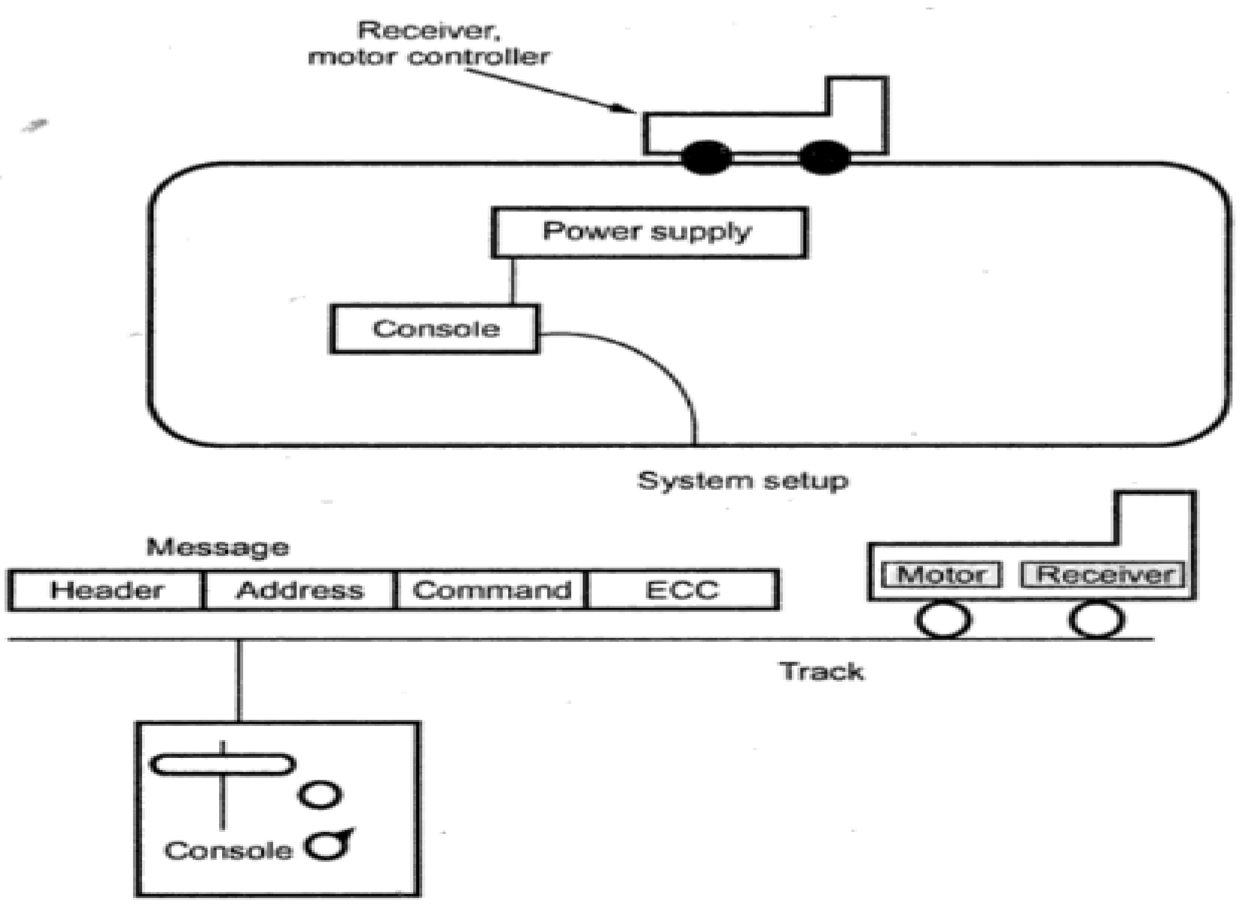


Figure: A model train control system

Design example-Model Train Controller

➢The user sends message to the train with a control box attached to the tracks. The control box can send signals to the train over the tracks by modulating power supply voltage [since train receives its electrical power from two rails of the track].

➢The control panel sends packets over the tracks to the receiver. The train includes analog electronics to sense the bits being transmitted and control system to set motor's speed and direction.

➢Each packet includes an address to control several trains on the same track and Error Correction Code (ECC) to correct transmission errors.

Design example-Model Train Controller

Requirements :

The requirements of the system are as follows :

Name	**Model train controller**
Purpose	**Control speed of trains**
Inputs	**Throttle, inertia setting, emergency stop, train no.**
Outputs	**Train control signals**
Functions	**Set engine speed based on inertia settings; respond to emergency stop**
Performance	**can update train speed at least 10 times/sec.**
Manufacturing cost	**Rs. 2500**
Power dissipation	**10 W (plugs into wall)**

Design example-Model Train Controller

Specifications:

➢Consider how the console controls the train by sending messages over the tracks. The transmissions are packetized.

➢Each packet includes an address and a message. Figure shows a typical sequence of train control commands.

➢Both the console and receiver run continuously. Packets can be sent at any time. A set_inertia message will be sent rarely since train's mass would change only when cars are added or removed.

➢Set_speed messages will be sent frequently to speed up or slow down the train smoothly. An emergency stop (Estop) command may be received occasionally to shut down the train motor immediately.

Design example-Model Train Controller

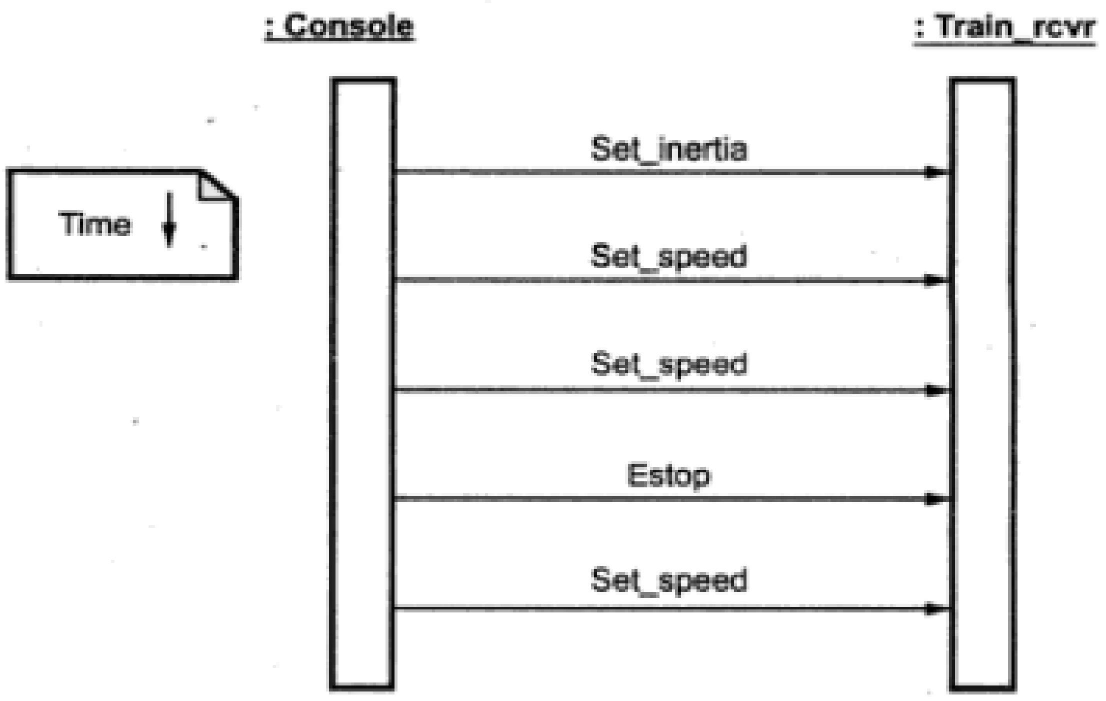


Figure: A typical sequence of train control commands using UML.

Design example-Model Train Controller

The **console** needs to perform three functions :

i) Read the state of front panel.

ii) Format the messages.

iii) Transmit the messages.

The train receiver must perform three functions :

i) Receive the messages.

ii) Interpret the messages.

iii) Control the motor.

Fig. 1.15 shows a UML collaboration diagram for major subsystems of train controller.

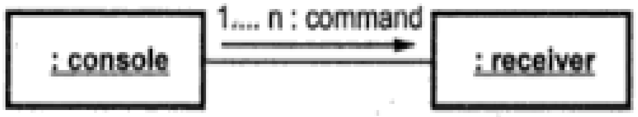
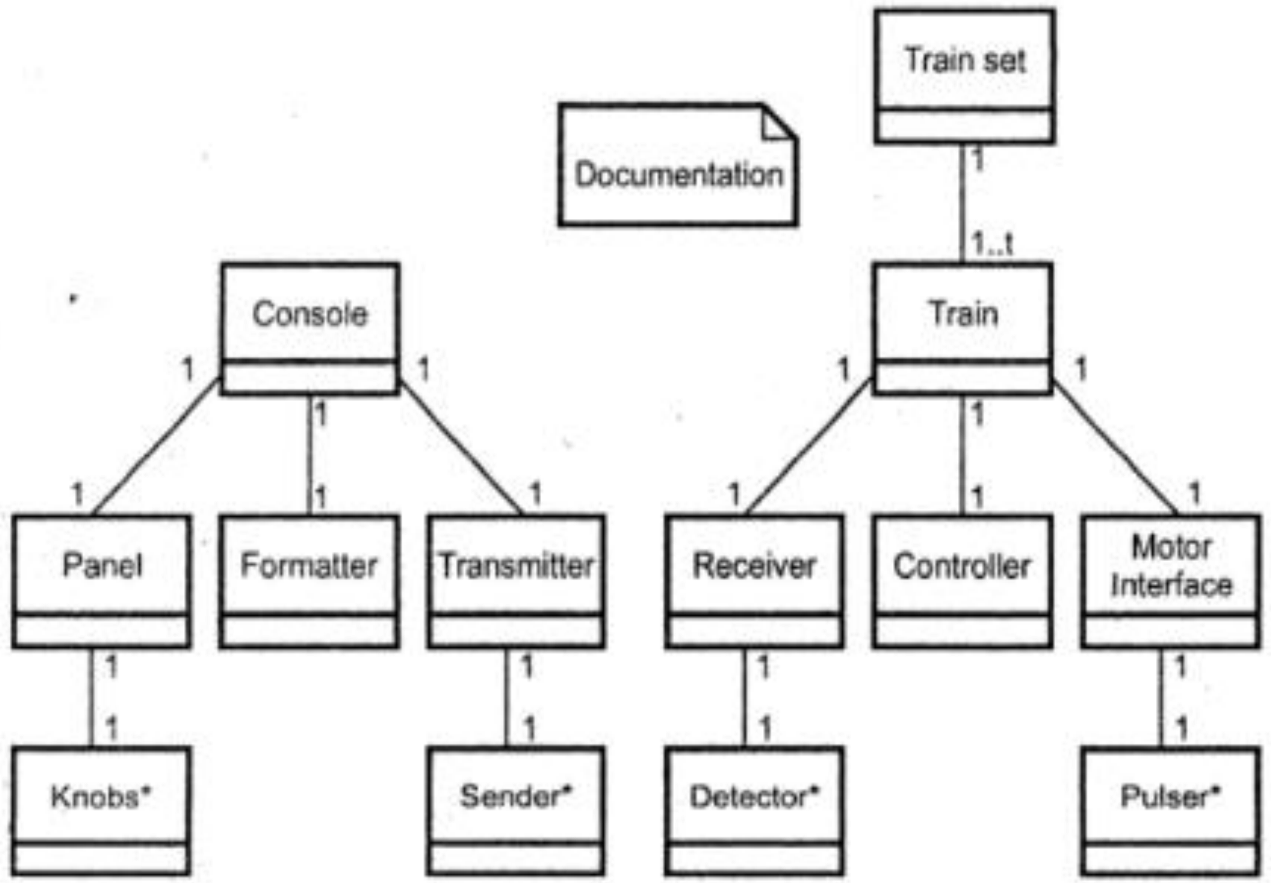


Figure: The console and receiver are both represented by objects.

Design example-Model Train Controller



* = Physical object

Figure: UML class diagram for train controller

Design example-Model Train Controller

The console class uses three classes that defines its components.

1) **Panel class** describes the console's front panel, which contains analog knobs and hardware to interface to digital parts of system.

2) **Formatter class** includes behaviors that know how to read the knobs and creates a bit stream for the required message.

3) **Transmitter class** interfaces to analog electronics to send message.

Design example-Model Train Controller

Similarly, the train uses three other classes that defines its components :

1) **Receiver class** knows how to turn analog signals into digital.

2) **Controller class** includes behaviors that interpret the commands and find out how to control the motor.

3) **Motor interface class** defines how to generate analog signals required to control the motor.

Design example-Model Train Controller

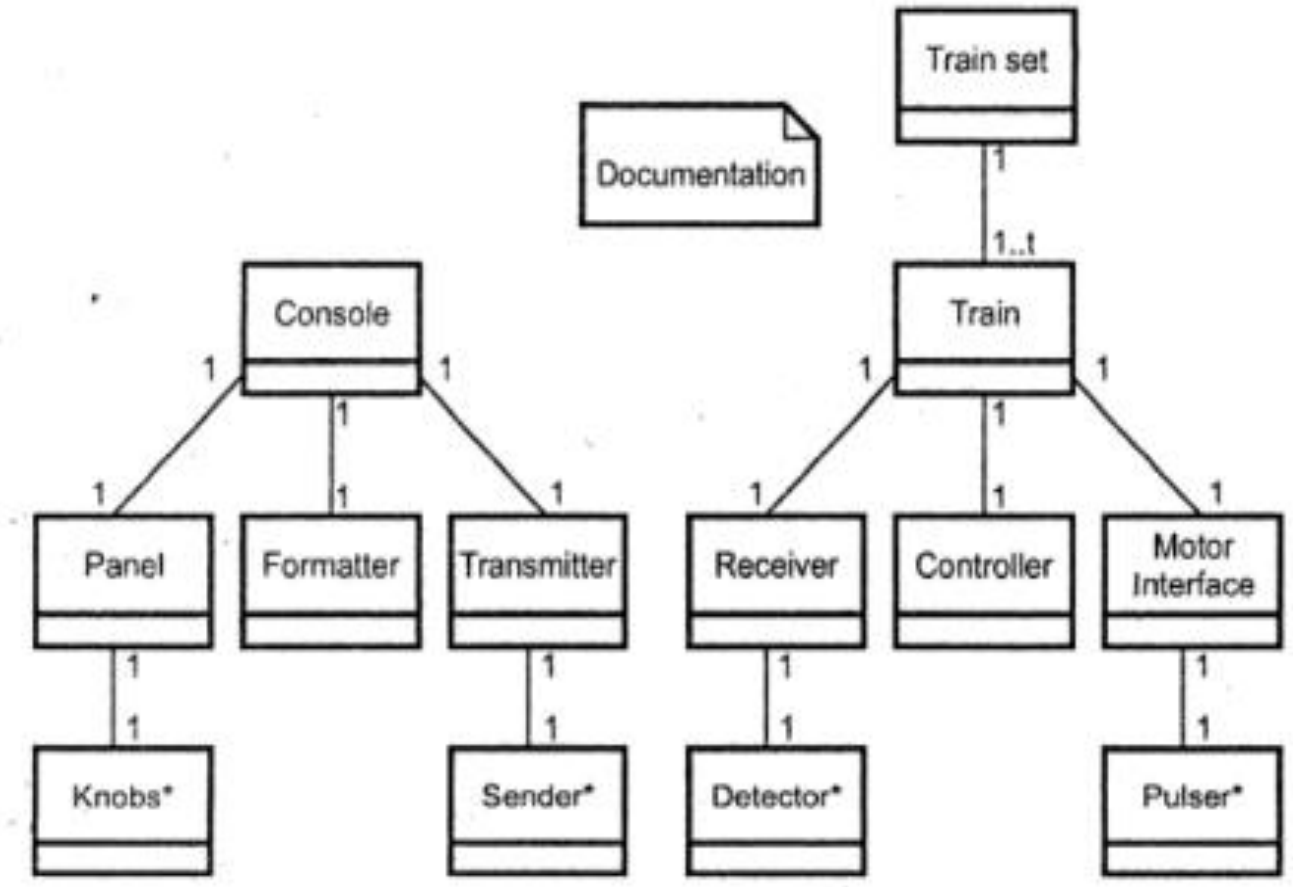
There are two classes to represent analog components :

i) **Detector***detects analog signals on the track and converts them into digital form.

ii) **Pulser*** converts digital commands into analog signals required to control motor speed.

Train set is a special class to indicate that system can handle multiple trains.

Design example-Model Train Controller



* = Physical object

Figure: UML class diagram for train controller

Design example-Model Train Controller

Detailed specification :

Here, we need to define the analog components in more detail as their characteristics will strongly influence the formatter and controller. Figure shows a class diagrams for analog components.

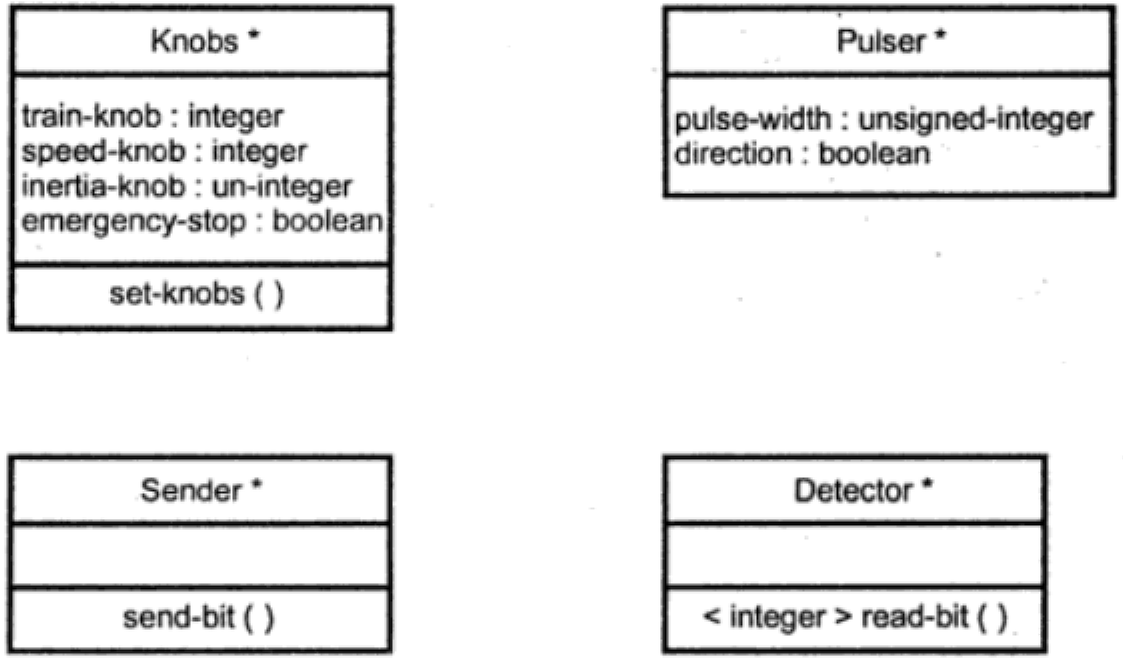


Figure: Classes describing analog components

Design example-Model Train Controller

Panel has three knobs :
rain no., speed and inertia. It also has emergency stop button.

When the train no. setting is changed, the other controls must also be set to proper values for that train.

For this purpose, knobs must provide set-knobs behavior that allows the rest of the system to modify the knob settings. The motor system takes its motor commands in two parts : Sender class simply put out a bit and detector class pick up a bit.

Design example-Model Train Controller

➢Pulser class can be understood by considering the actual process of motor's speed control. Below figure shows how the speed of motors is controlled using pulse width modulation.

➢Power is applied for a fraction of time, the fraction of time that power is applied determines the speed. The digital interface to the motor system specifies pulse width as an integer (maximum value indicating maximum engine speed). Binary value controls the direction.

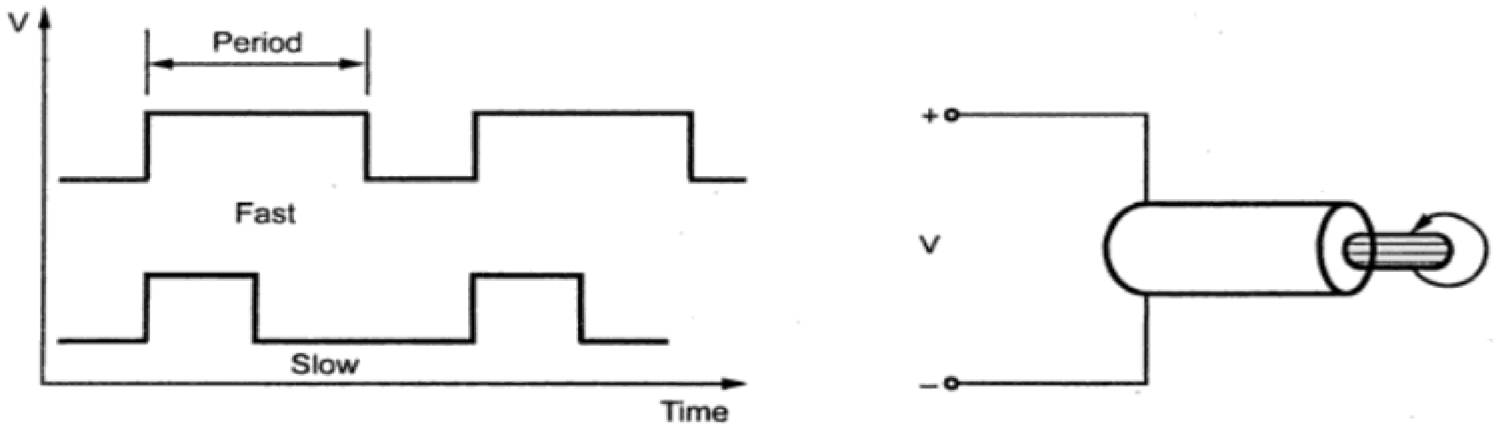


Figure: **Motor speed control using pulse width modulation**

Design example-Model Train Controller

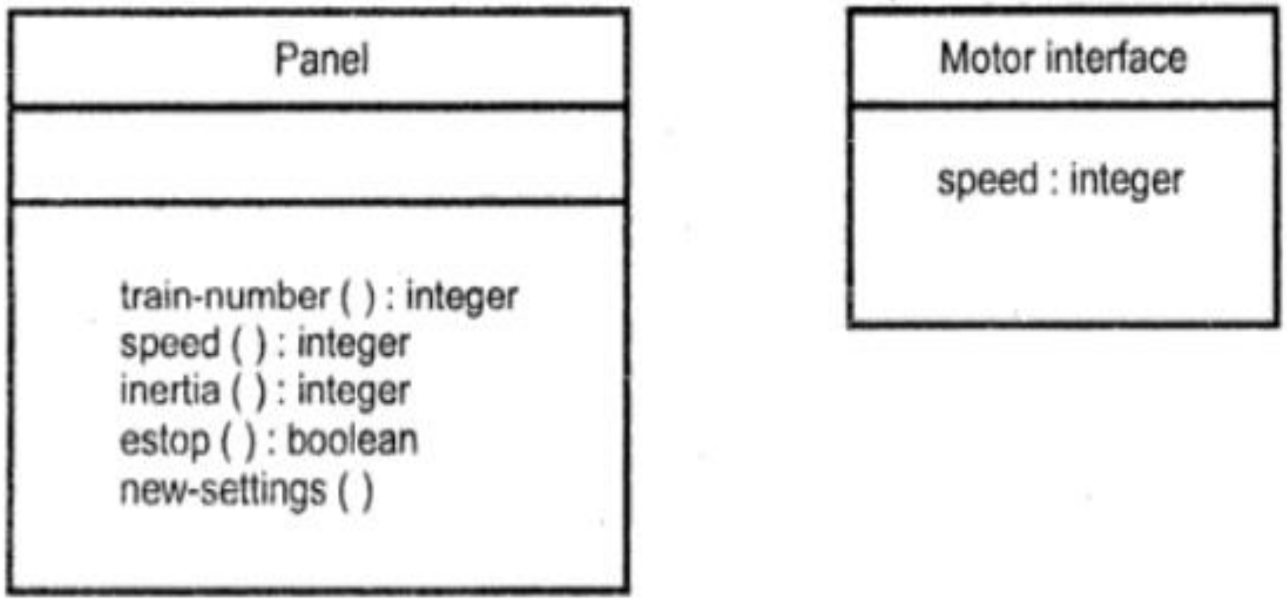


Figure: **Class diagram for Panel and Motor interface.**

Classes for panel and motor interface form the software interfaces to there respective physical devices.

Design example-Model Train Controller

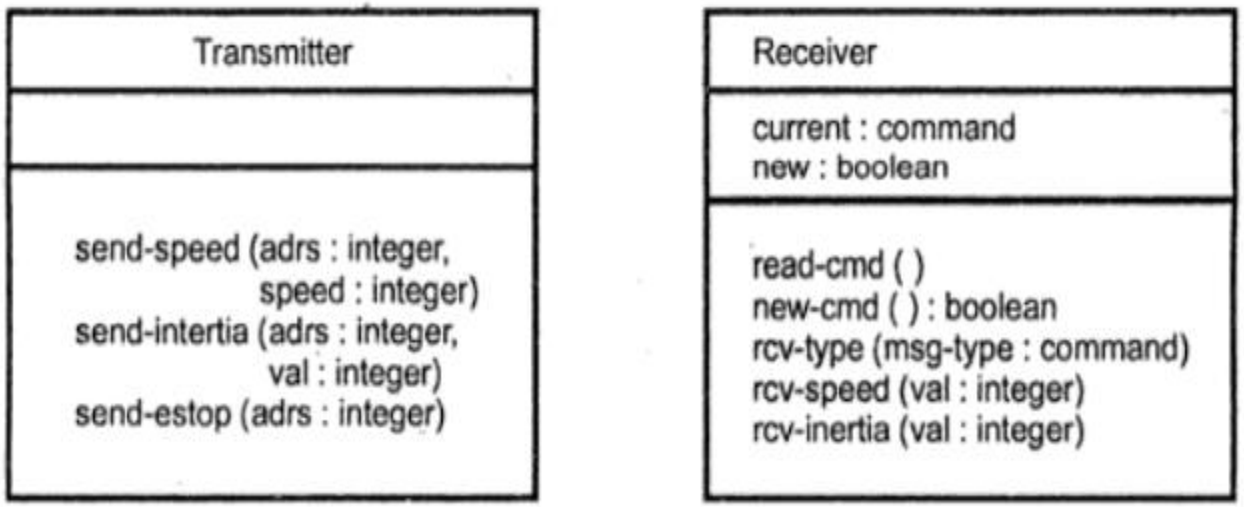
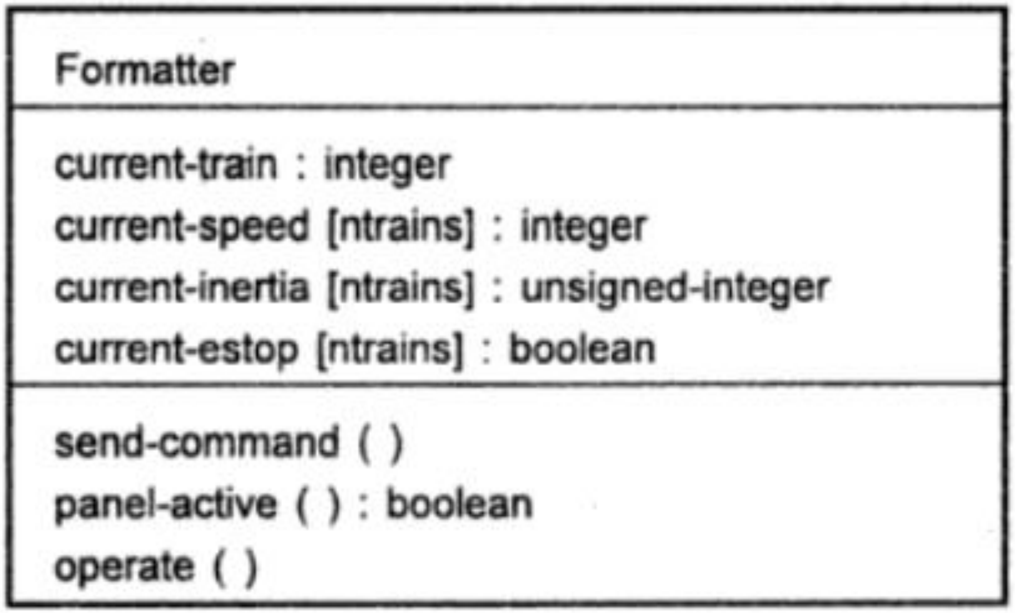

Transmitter
send-speed (adrs : integer, speed : integer)
send-intertia (adrs : integer, val : integer)
send-estop (adrs : integer)
Receiver
current : command
new : boolean
read-cmd ( )
new-cmd ( ) : boolean
rcv-type (msg-type : command)
rcv-speed (val : integer)
rcv-inertia (val : integer)

Figure: **Class diagram for transmitter and receiver.**

These classes provide the software interface to their respective physical devices. The transmitter provides a distinct behavior for each type of message that can be sent. It also takes care of formatting the message. The receiver provides read-and behavior to read the message. This behavior run continuously to monitor the tracks and intercept the next command.

Design example-Model Train Controller

The **formatter class** which holds the current control settings for all the trains. The send-command function serves as a interface to the transmitter. The operate function performs basic actions for the object. The panel-active behaviour returns true whenever panel's values does not correspond to current values.


Formatter
current-train : integer
current-speed [ntrains] : integer
current-inertia [ntrains] : unsigned-integer
current-estop [ntrains] : boolean
send-command ( )
panel-active ( ) : boolean
operate ( )

Design example-Model Train Controller

Figure shows the role of formatter during panel’s operation by sequence diagram. There are two changes to knob settings: first to throttle, inertia or emergency stop; then to train number.

Design example-Model Train Controller

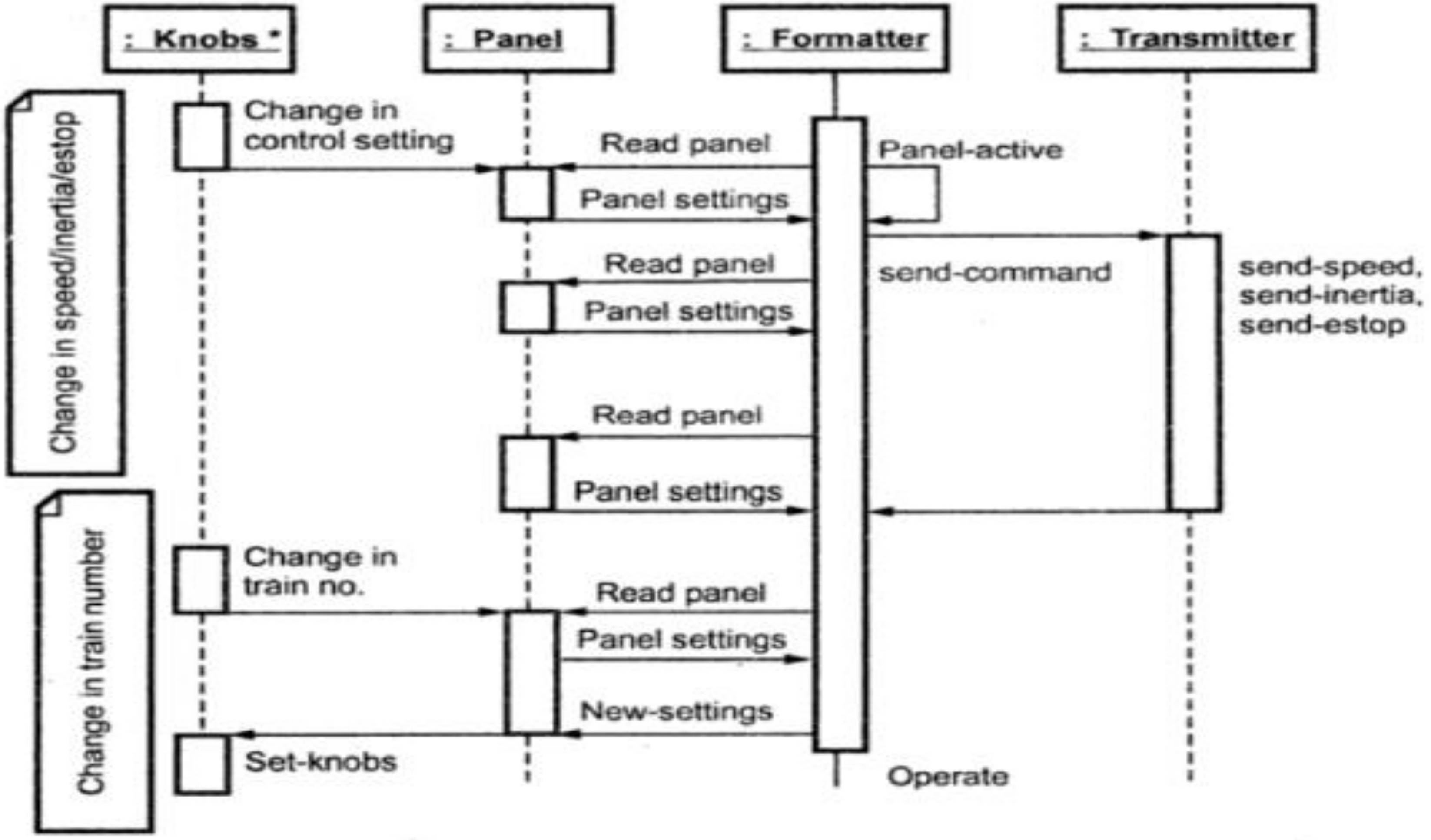


Figure: **Sequence diagram for transmitting a control input.**

Unit-II
INTRODUCTION TO EMBEDDED C AND APPLICATIONS

- **SYLLABUS:**

  C looping structures, register allocation, function calls, pointer aliasing, structure arrangement, bit fields, unaligned data and endianness, inline functions and inline assembly, portability issues; Embedded systems programming in C, binding and running embedded C program in Keil IDE, dissecting the program, building the hardware; Basic techniques for reading and writing from I/O port pins, switch bounce;

  Applications: Switch bounce, LED interfacing, interfacing with keyboards, displays, D/A and A/D conversions, multiple interrupts, serial data communication using embedded C interfacing.

C Looping Structures

This section looks at the most efficient ways to code **for** and **while** loops on the ARM. We start by looking at loops with a fixed number of iterations and then move on to loops with a variable number of iterations. Finally we look at loop unrolling.

- **Loops with a Fixed Number of Iterations**

This shows how the compiler treats a loop with incrementing count i++.

```
int checksum_v5(int *data)
{
unsigned int i;
int sum=0;

for (i=0; i<64; i++)
{
sum += *(data++);
}
return sum;
}
```

This compiles to

```
checksum_v5
         MOV      r2,r0                  ; r2 = data
         MOV      r0,#0                  ; sum = 0
         MOV      r1,#0                  ; i = 0
checksum_v5_loop
         LDR      r3,[r2],#4             ; r3 = *(data++)
         ADD      r1,r1,#1               ; i++
         CMP      r1,#0x40               ; compare i, 64
         ADD      r0,r3,r0               ; sum += r3
         BCC      checksum_v5_loop       ; if (i<64) goto loop
         MOV      pc,r14                 ; return sum
```

It takes three instructions to implement the for loop structure:

- An ADD to increment i
- A compare to check if i is less than 64
- A conditional branch to continue the loop if i < 64

This example shows the improvement if we switch to a decrementing loop rather than an incrementing loop.

```
int checksum_v6(int *data)
{
  unsigned int i;
  int sum=0;

  for (i=64; i!=0; i--)
  {
    sum += *(data++);
  }
  return sum;
}
```

This compiles to

```
checksum_v6
        MOV     r2,r0               ; r2 = data
        MOV     r0,#0               ; sum = 0
        MOV     r1,#0x40            ; i = 64
checksum_v6_loop
        LDR     r3,[r2],#4          ; r3 = *(data++)
        SUBS    r1,r1,#1            ; i-- and set flags
        ADD     r0,r3,r0            ; sum += r3
        BNE     checksum_v6_loop    ; if (i!=0) goto loop
        MOV     pc,r14              ; return sum
```

Loops Using a Variable Number of Iterations

Now suppose we want our checksum routine to handle packets of arbitrary size. We pass in a variable N giving the number of words in the data packet. Using the lessons from the last section we count down until N 0 and don't require an extra loop counter i.

The checksum_v7 example shows how the compiler handles a **for** loop with a **variable number** of iterations N.

```
int checksum_v7(int *data, unsigned int N)
{
int sum=0;
for (; N!=0; N--)
{
sum += *(data++);
}
return sum;
}
```

This compiles to

```
checksum_v7
            MOV     r2,#0                   ; sum = 0
            CMP     r1,#0                   ; compare N, 0
            BEQ     checksum_v7_end         ; if (N==0) goto end

checksum_v7_loop
            LDR     r3,[r0],#4              ; r3 = *(data++)
            SUBS    r1,r1,#1                ; N-- and set flags
            ADD     r2,r3,r2                ; sum += r3
            BNE     checksum_v7_loop        ; if (N!=0) goto loop
checksum_v7_end
            MOV     r0,r2                   ; r0 = sum
            MOV     pc,r14                  ; return r0
```

EXAMPLE3:

This example shows how to use a do-while loop to remove the test for N being zero that occurs in a for loop.

```
int checksum_v8(int *data, unsigned int N)
{
   int sum=0;

   do
   {
     sum += *(data++);
   } while (--N!=0);
   return sum;
 }
```

The compiler output is now

```
checksum_v8
        MOV     r2,#0               ; sum = 0
checksum_v8_loop
        LDR     r3,[r0],#4          ; r3 = *(data++)
        SUBS    r1,r1,#1            ; N-- and set flags
        ADD     r2,r3,r2            ; sum += r3
        BNE     checksum_v8_loop    ; if (N!=0) goto loop
        MOV     r0,r2               ; r0 = sum
        MOV     pc,r14              ; return r0
```

Loop Unrolling:

We saw in Section 5.3.1 that each loop iteration costs two instructions in addition to the body of the loop: a subtract to decrement the loop count and a conditional branch.

We call these instructions the *loop overhead*. On ARM7 or ARM9 processors the subtract takes one cycle and the branch three cycles, giving an overhead of four cycles per loop.

You can save some of these cycles by *unrolling* a loop—repeating the loop body several times, and reducing the number of loop iterations by the same proportion. For example, let’s unroll our packet checksum example four times.

Register Allocation

Register Allocation:

➢The compiler attempts to allocate a processor register to each local variable you use in a C function. It will try to use the same register for different local variables if the use of the variables do not overlap.

➢When there are more local variables than available registers, the compiler stores the excess variables on the processor stack. These variables are called ***spilled*** **or** ***swapped out variables*** *since they are written out to*.

➢Spilled variables are slow to access compared to variables allocated to registers.

Register Allocation

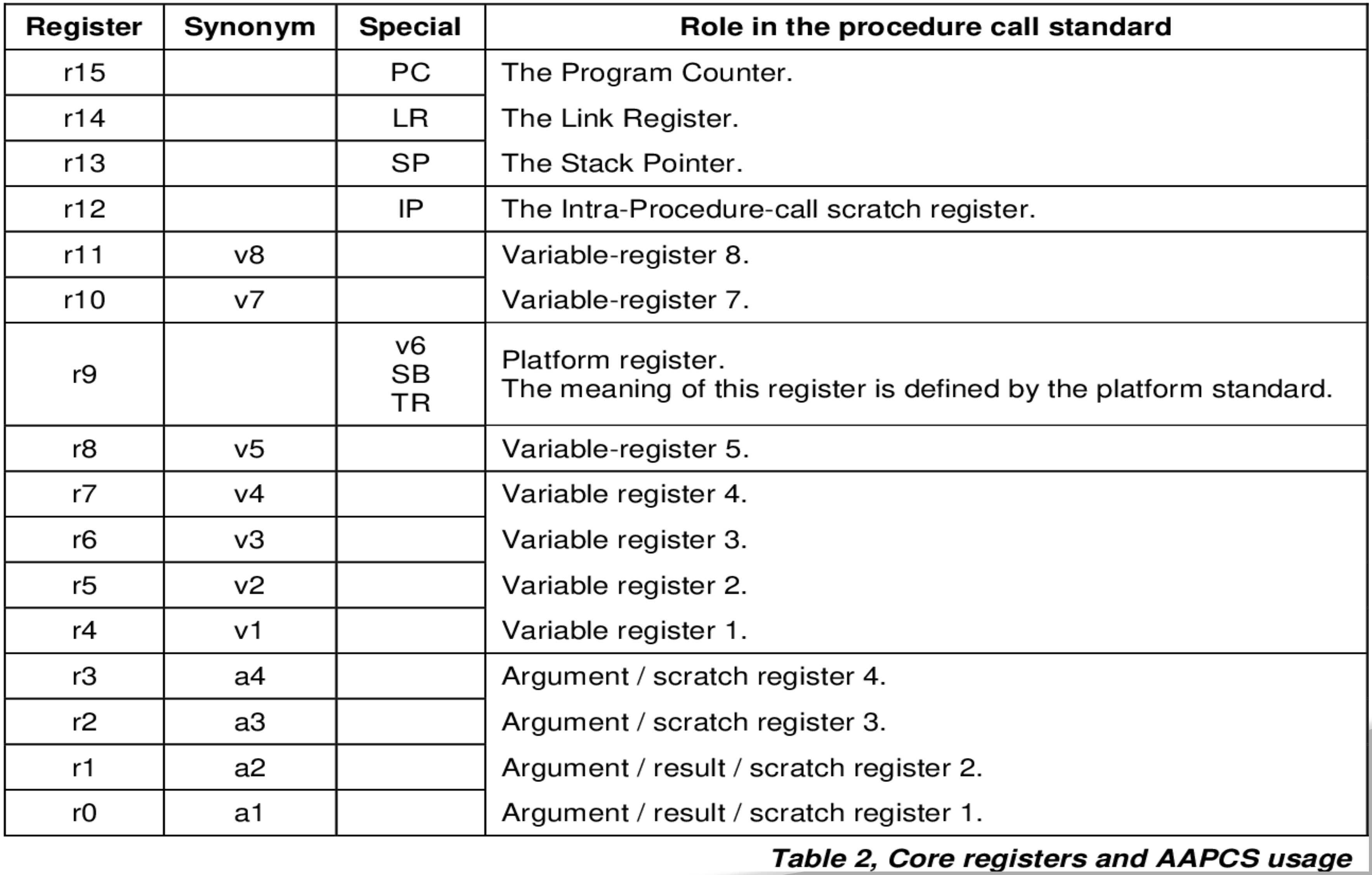
To implement a function efficiently, you need to

- ■ minimize the number of spilled variables
- ■ ensure that the most important and frequently accessed variables are stored in registers

First let’s look at the number of processor registers the ARM C compilers have available for allocating variables.

Table shows the standard register names and usage when following the ARM-Thumb procedure call standard (ATPCS), which is used in code generated by C compilers.

Register Allocation



Register	Synonym	Special	Role in the procedure call standard
r15		PC	The Program Counter.
r14		LR	The Link Register.
r13		SP	The Stack Pointer.
r12		IP	The Intra-Procedure-call scratch register.
r11	v8		Variable-register 8.
r10	v7		Variable-register 7.
r9		v6 SB TR	Platform register. The meaning of this register is defined by the platform standard.
r8	v5		Variable-register 5.
r7	v4		Variable register 4.
r6	v3		Variable register 3.
r5	v2		Variable register 2.
r4	v1		Variable register 1.
r3	a4		Argument / scratch register 4.
r2	a3		Argument / scratch register 3.
r1	a2		Argument / result / scratch register 2.
r0	a1		Argument / result / scratch register 1.

Table 2, Core registers and AAPCS usage

Introduction

- The embedded firmware is responsible for controlling the various peripherals of the embedded hard-ware and generating response in accordance with the functional requirements mentioned in the requirements for the particular embedded product.

- Firmware is considered as the master brain of the embedded system.

- Imparting intelligence to an Embedded system is a one time process and it can happen at any stage, it can be immediately after the fabrication of the embedded hardware or at a later stage.

Embedded systems programming in C

PROGRAMMING IN EMBEDDED C:

- Whenever the conventional 'C' Language and its extensions are used for programming embedded systems, it is referred as **'Embedded C'** programming.
- Programming in 'Embedded C' is quite different from conventional Desktop application development using 'C' language for a particular OS platform.
- Desktop computers contain working memory in the range of Megabytes (Nowadays Giga bytes) and storage memory in the range of Giga bytes. For a desktop application developer, the resources available are surplus in quantity and they can be very lavish in the usage of RAM and ROM and no restrictions are imposed at all.

Embedded systems programming in C

- This is not the case for embedded application developers.
- Almost all embedded systems are limited in both storage and working memory resources.
- Embedded application developers should be aware of this fact and should develop applications in the best possible way which optimizes the code memory and working memory usage as well as performance.
- In other words, the hands of an embedded application developer are always tied up in the memory usage context.

Embedded systems programming in C

'C' v/s. 'Embedded C':

- 'C' is a well structured, well defined and standardized 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 pay roll systems, banking applications, etc).
- The conventional 'C' language follows ANSI(American National Standards Institute) 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 systems programming in C

- 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'.
- 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).

Embedded systems programming in C

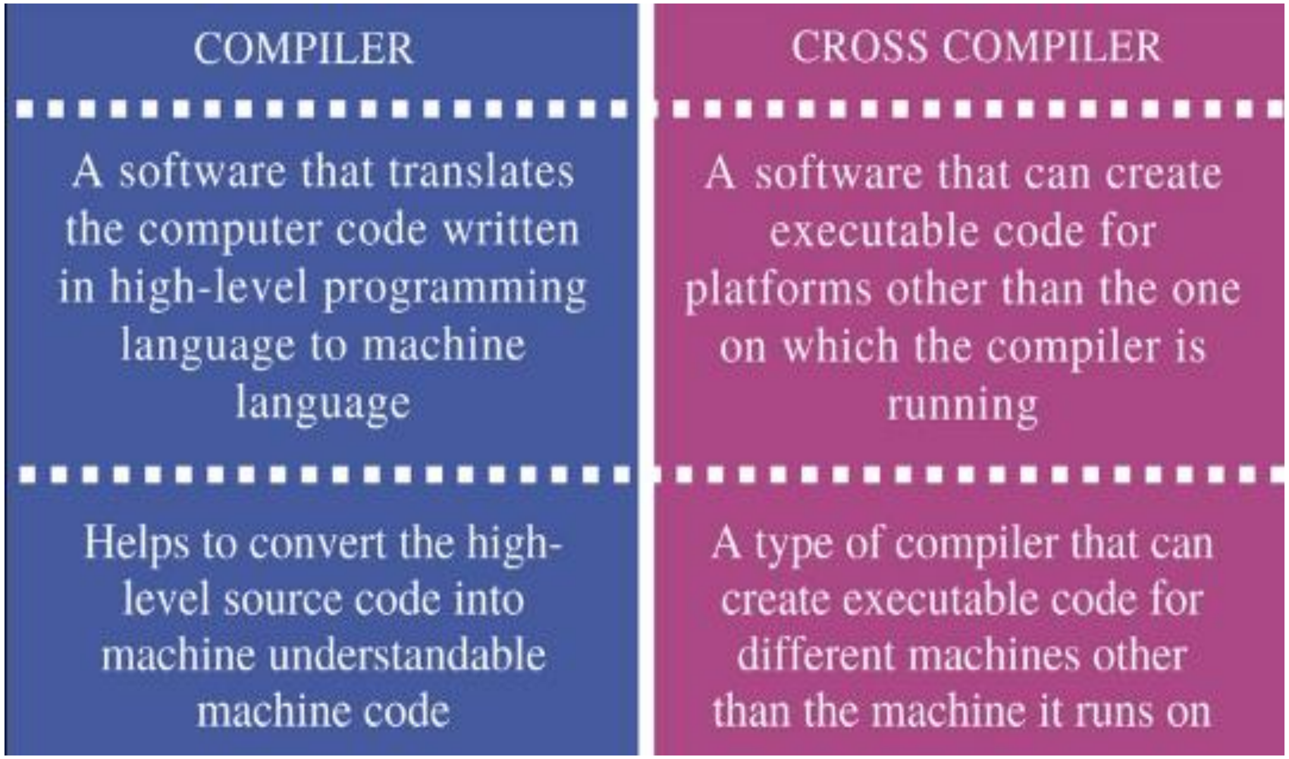
Compiler vs. Cross-Compiler:

- 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.

Embedded systems programming in C

- 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.
- 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. Whenever you see the term 'Compiler' related to any embedded firmware application, please understand that it is referring to the cross-compiler.

Embedded systems programming in C



COMPILER	CROSS COMPILER
A software that translates the computer code written in high-level programming language to machine language	A software that can create executable code for platforms other than the one on which the compiler is running
Helps to convert the high-level source code into machine understandable machine code	A type of compiler that can create executable code for different machines other than the machine it runs on

For example: A compiler that runs on windows platform also generates a code that runs on Linux platform is a cross compiler.

Embedded systems programming in C

Using ‘C’ in ‘Embedded C’:

- Let us brush up whatever we learned in conventional 'C' programming. Remember we will only go through the peripheral aspects and will not go in deep.

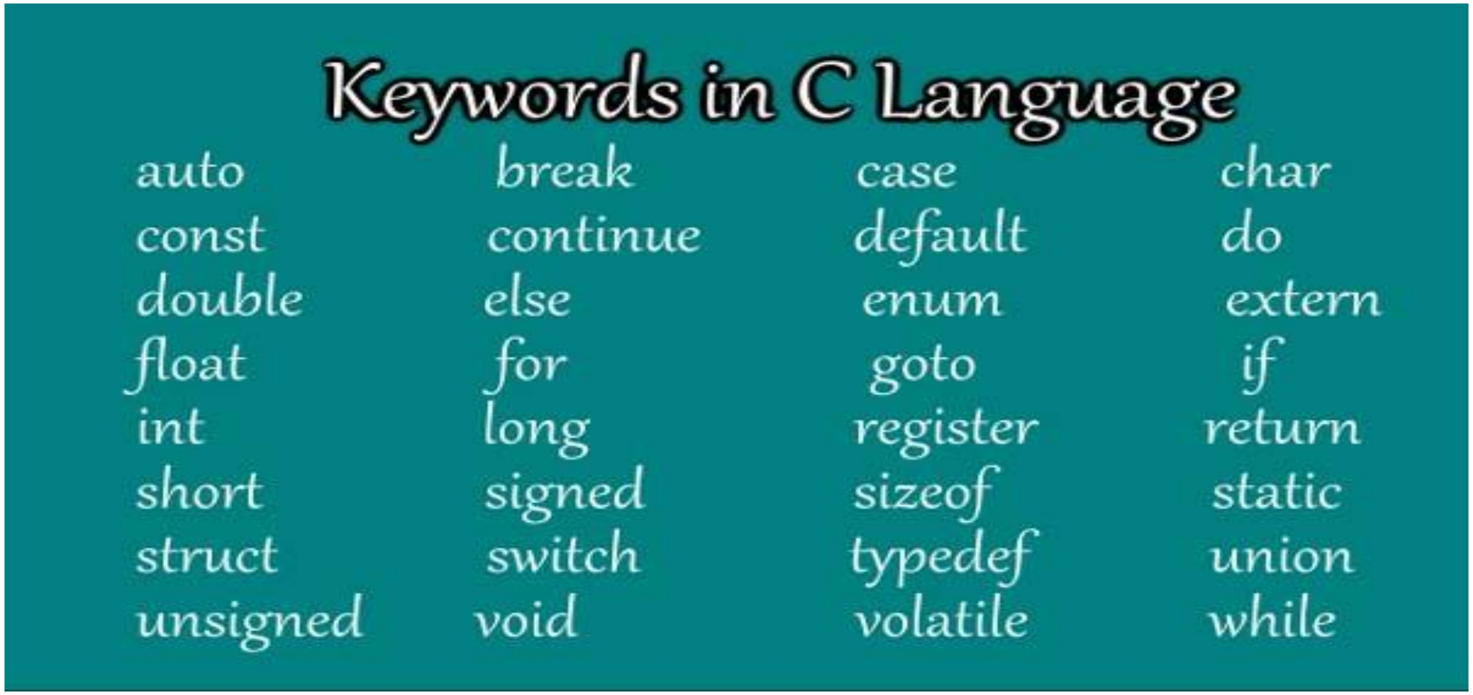
Keywords and Identifiers:

- Keywords are the reserved names used by the 'C' language. All keywords have a fixed meaning in the 'C' language context and they are not allowed for programmers for naming their own variables or functions. ANSI 'C' supports 32 keywords and they are listed below.

- All 'C' supported keywords should be written in 'lowercase' letters.

- **C Keywords** are predefined, reserved words used in programming that have special meanings to the compiler.

Embedded systems programming in C



Keywords in C Language

auto	break	case	char
const	continue	default	do
double	else	enum	extern
float	for	goto	if
int	long	register	return
short	signed	sizeof	static
struct	switch	typedef	union
unsigned	void	volatile	while

Identifiers are user defined names and labels. Identifiers can contain letters of English alphabet (both upper and lower case) and numbers. The starting character of an identifier should be a letter. The only special character allowed in identifier is underscore (_).

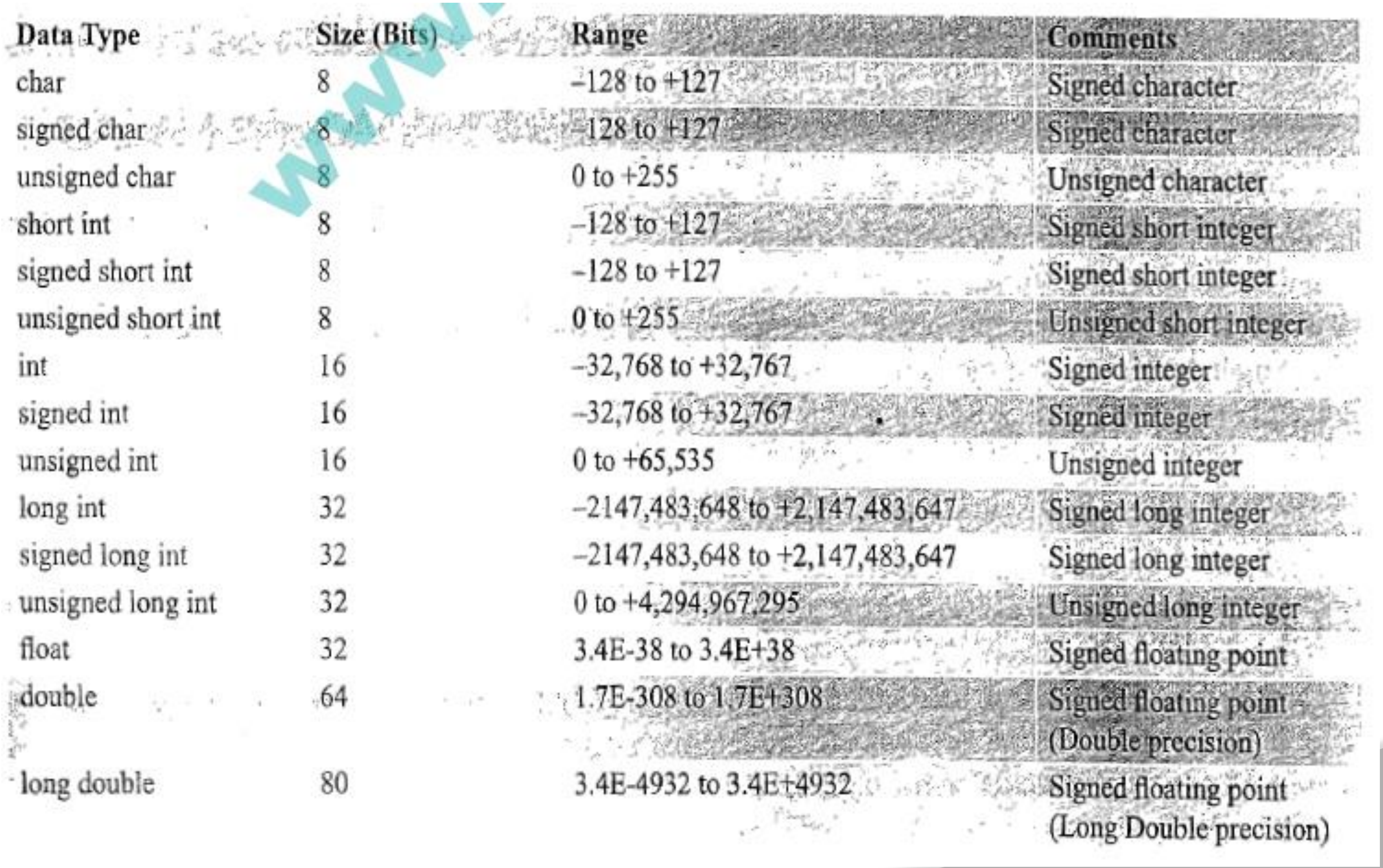
Ex: Root, _getchar, _sin, x_1, x1, If

Embedded systems programming in C

Data Types:

- Data type represents the type of data held by a variable. The various data types supported, their storage space (bits) and storage capacity for 'C' language are tabulated below.

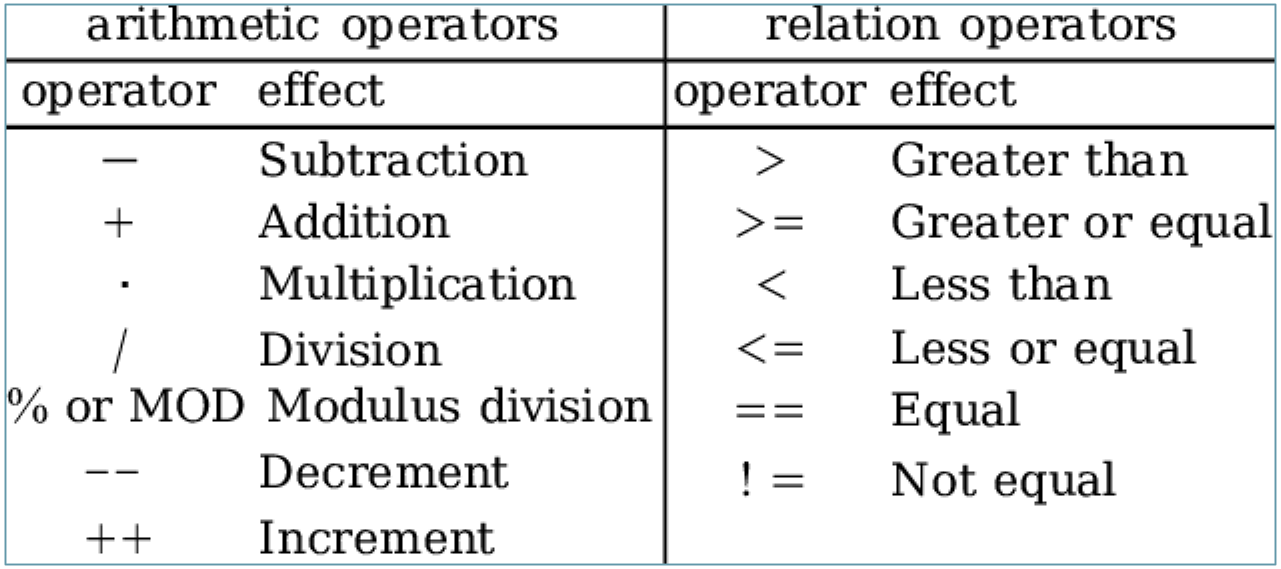
Embedded systems programming in C



Data Type	Size (Bits)	Range	Comments
char	8	–128 to +127	Signed character
signed char	8	–128 to +127	Signed character
unsigned char	8	0 to +255	Unsigned character
short int	8	–128 to +127	Signed short integer
signed short int	8	–128 to +127	Signed short integer
unsigned short int	8	0 to +255	Unsigned short integer
int	16	–32,768 to +32,767	Signed integer
signed int	16	–32,768 to +32,767	Signed integer
unsigned int	16	0 to +65,535	Unsigned integer
long int	32	–2147,483,648 to +2,147,483,647	Signed long integer
signed long int	32	–2147,483,648 to +2,147,483,647	Signed long integer
unsigned long int	32	0 to +4,294,967,295	Unsigned long integer
float	32	3.4E-38 to 3.4E+38	Signed floating point
double	64	1.7E-308 to 1.7E+308	Signed floating point (Double precision)
long double	80	3.4E-4932 to 3.4E+4932	Signed floating point (Long Double precision)

Arithmetic and Relational Operations:

- **The list of arithmetic operations supported by ‘C’ are listed below.**

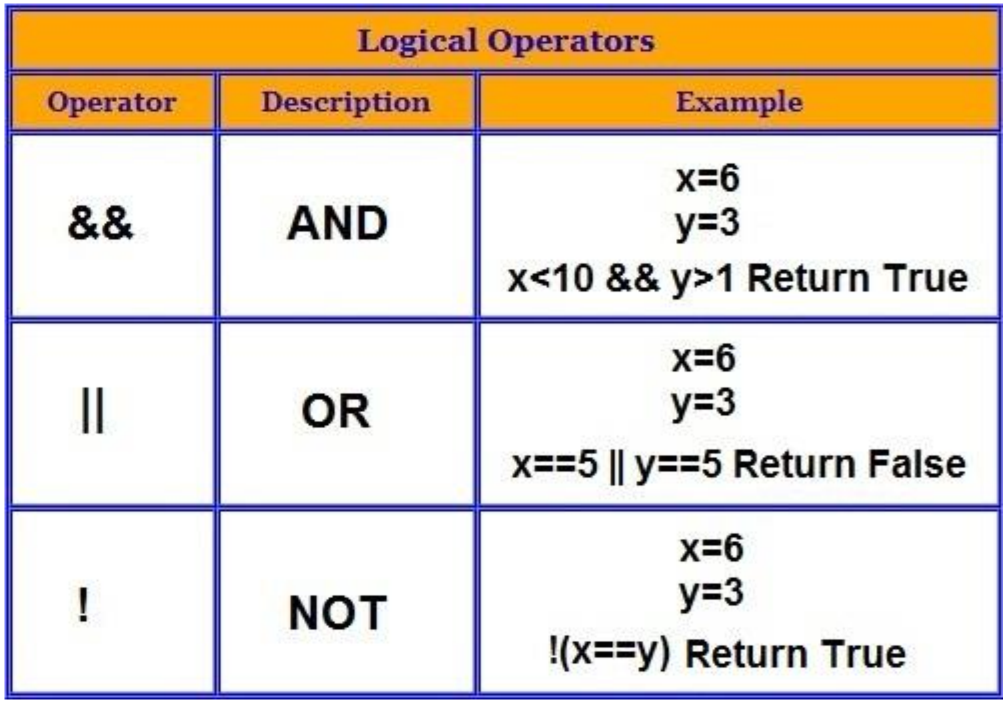


arithmetic operators		relation operators	
operator	effect	operator	effect
−	Subtraction	>	Greater than
+	Addition	>=	Greater or equal
·	Multiplication	<	Less than
/	Division	<=	Less or equal
% or MOD	Modulus division	==	Equal
−−	Decrement	! =	Not equal
++	Increment		

Embedded systems programming in C

Logical Operations:

- Logical operations are usually performed for decision making and program control transfer.



Logical Operators		
Operator	**Description**	**Example**

&&	AND	x=6
		y=3
		x<10 && y>1 Return True

||	OR	x=6
		y=3
		x==5 || y==5 Return False

!	NOT	x=6
		y=3
		!(x==y) Return True

Embedded systems programming in C

Looping Instructions:

- Looping instructions are used for executing a particular block of code repeatedly till a condition is met or wait till an event is fired.
- Embedded programming often uses the looping instructions for checking the status of certain I/O ports, registers, etc. and also for producing delays. Certain devices allow write/read operations to and from some registers of the device only when the device is ready and the device ready is normally indicated by a status register or by setting/clearing certain bits of status registers.
- Hence the program should keep on reading the status register till the device ready indication comes. The reading operation forms a loop. The looping instructions supported by are listed below.

Embedded systems programming in C

Looping Instructions:

```
//while statement
While (expression)
{
Body of while loop
}
```

```
//do while statement
do
{
Body of do loop
}
While (expression)
```

```
//for loop
for (initialization; test for condition; update variable)
{
Body of for loop
}
```

Embedded systems programming in C

Arrays and Pointers:

- **Array** is a collection of related elements (data types).
- Arrays are usually declared with data type of array, name of the array and the number of related elements to be placed in the array.
- For example the following array declaration declares a character array with name ‘arr’ and reserves space for 5 character elements in the memory as below figure.

char arr [5]

0x10	**0x10**	**0x23**	**0x03**	**0x45**	**(contents)**
arr[0]	arr[1]	arr[2]	arr[3]	arr[4]	
0x8000	**0x8001**	**0x8002**	**0x8003**	**0x8004**	**(Addresses)**

Embedded systems programming in C

- The elements of an array are accessed by using the array index or subscript.
- The index of the first element is '0'. For the above example the first element is accessed by arr[0], second element by arr[1], and so on. In the above example, the array starts at memory location 0x8000 (arbitrary value taken for illustration) and the address of the first element is 0x8000.
- The `address of operator (&) returns the address of the memory location where the variable is stored. Hence &arr[0] will return 0x8000 and &arr[1] will return 0x8001, etc.. The name of the array itself with no index (subscript) always returns the address of the first element. If we examine the first element arr[0] of the above array, we can see that the variable arr[0] is allocated a memory location 0x8000 and the contents of that memory location holds the value for arr[0].

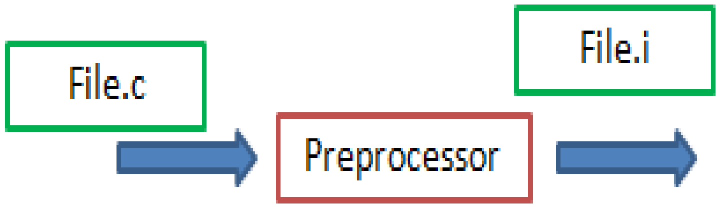
Embedded systems programming in C

Pointers:

- Pointer is a flexible at the same time most dangerous feature, capable of creating potential damages leading to firmware crash, if not used properly.
- Pointer is a memory pointing based technique for variable access and modification. Pointers are very helpful in
- 1. Accessing and modifying variables
- 2. Increasing speed of execution
- 3. Accessing contents within a block of memory
- 4. Passing variables to functions by eliminating the use of a local copy of variables
- 5. Dynamic memory.

Code Compilation process

- **Code Compilation process**

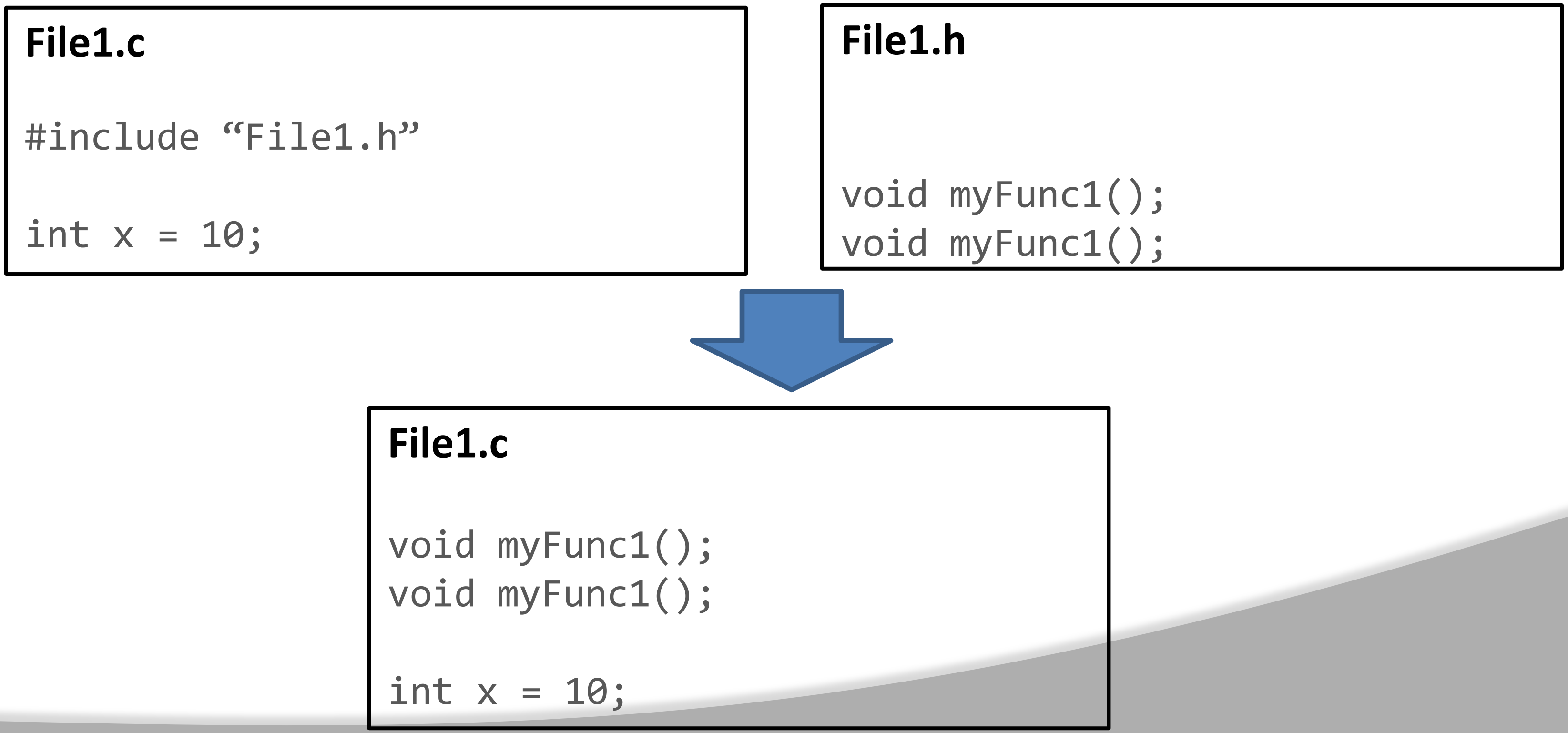


- **Preprocessing:**

  It is the first stage of compilation. It processes preprocessor directives like include-files, conditional compilation instructions and macros.

Code Compilation process

- **Examples on preprocessor directives:**
  - **Including files:**
    - `#include <stdio.h>`
  - **Tells the preprocessor to copy the content of file `stdio.h` and paste it here.**



File1.c

```
#include “File1.h”

int x = 10;
```

File1.h

```
void myFunc1();
void myFunc1();
```

File1.c

```
void myFunc1();
void myFunc1();

int x = 10;
```

Code Compilation process

➢ **Examples on preprocessor directives:**

- **Object-like Macro**

```
#define  LED_PIN  10
```

Tells the preprocessor that whenever the symbol `LED_PIN is found` inside the code, replace it with `10`.

So we can type inside the code:

```
int x = LED_PIN;          /* x will have the value 10 */
ledInit(LED_PIN);         /*Initialize LED with value 10*/

#define  MY_SECOND_NUMBER  LED_PIN
```

Now `MY_SECOND_NUMBER` also has the value `10`

Code Compilation process

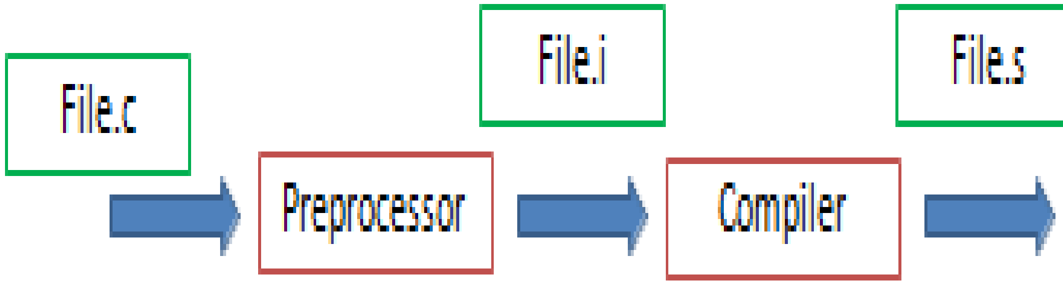
- **Examples on preprocessor directives:**

- Macro definition is really helpful in code maintainability and change, for example when a specific configuration value is used in all over the code in a lot of lines, so to change this value only one line will be changed which is the definition line itself instead of changing the value in all lines of code.
  - **Conditional compilation:**

```
#if(LED_PIN==10)
    printf("LED_PIN=10");
#endif
```

The `printf` line will be compiled only if the macro `LED_PIN` is defined with value `10`.

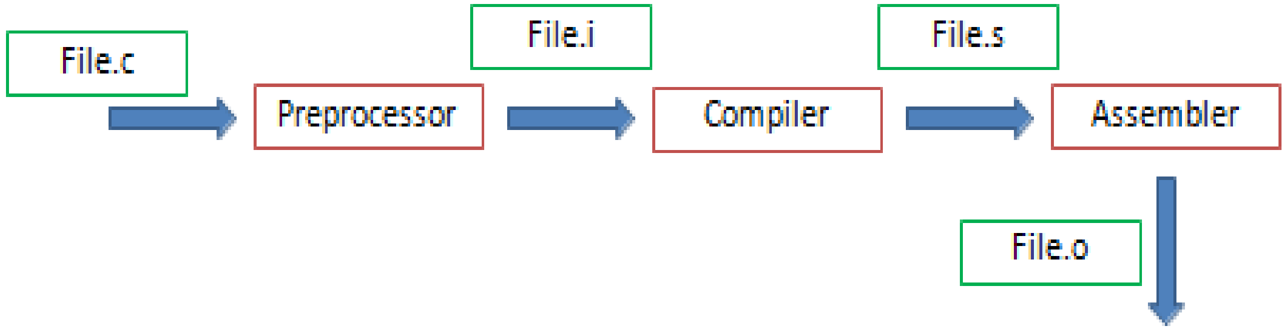
Code Compilation process



- **Compilation:**

  It is the second stage. It takes the output of the preprocessor with the source code, and generates assembly source code.

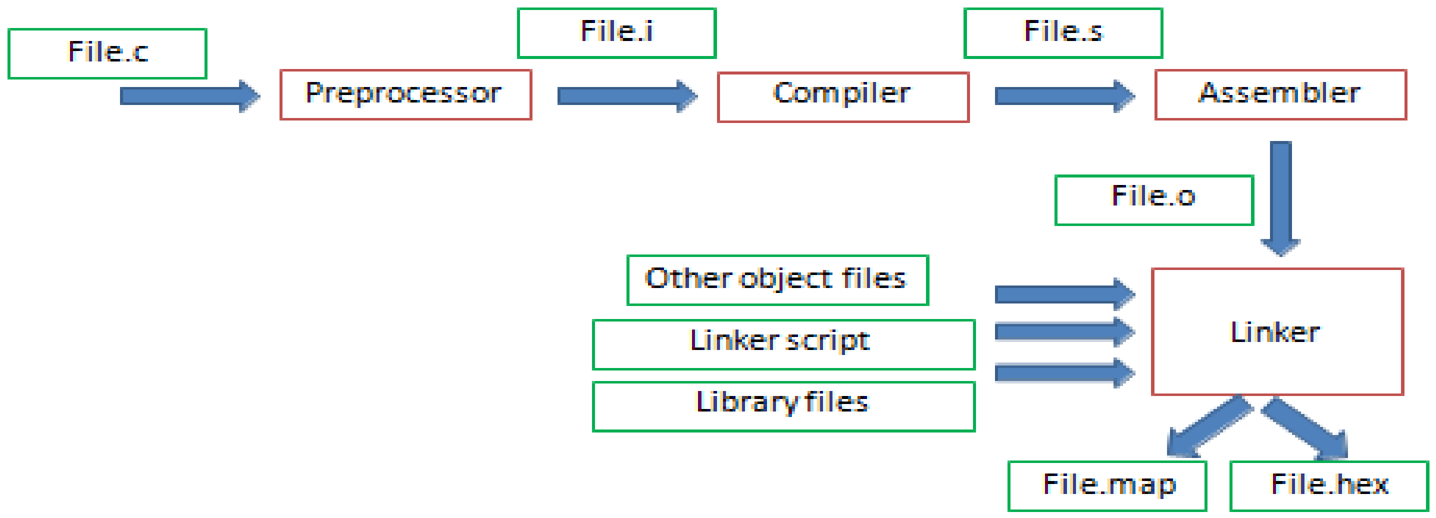
Code Compilation process



➢ **Assembler stage**

It is the third stage of compilation. It takes the assembly source code and produces the corresponding object code.

Code Compilation process

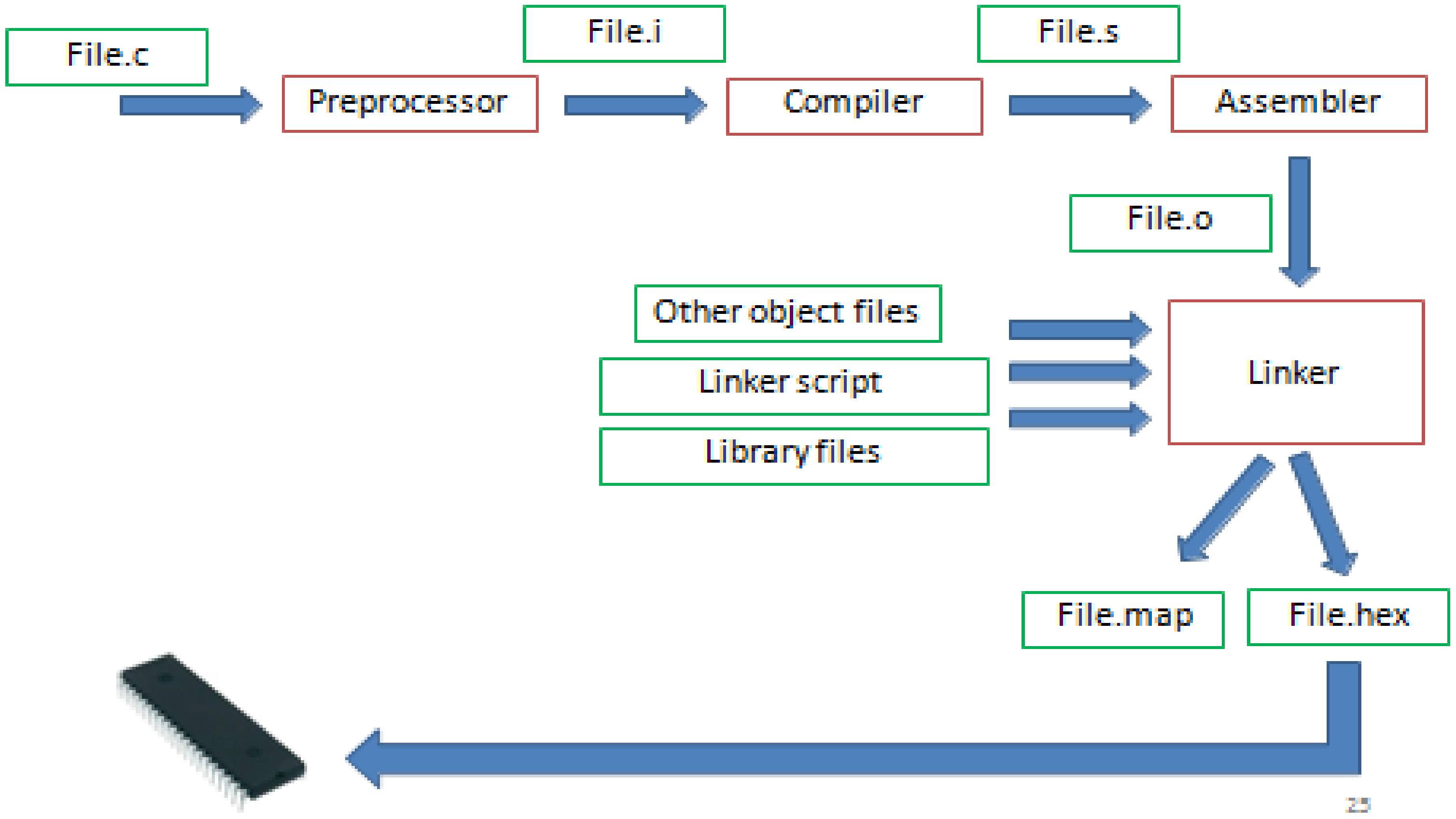


Linking:

It is the final stage of compilation. It takes one or more object files or libraries and linker script as input and combines them to produce a single executable file.

In doing so, it resolves references to external symbols, assigns final addresses to procedures/functions and variables, and revises code and data to reflect new addresses (a process called relocation).

Code Compilation process



Code Compilation process

- **Preprocessing**

  It is the first stage of compilation. It processes preprocessor directives like include-files, conditional compilation instructions and macros.

- **Compilation**

  It is the second stage. It takes the output of the preprocessor with the source code, and generates assembly source code.

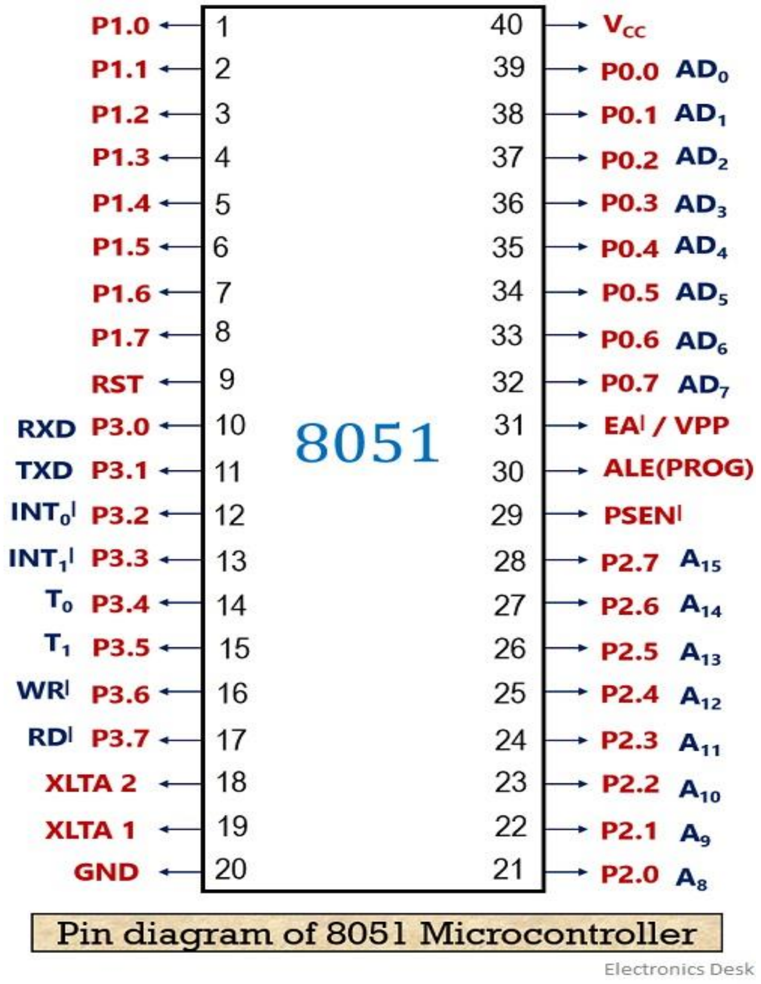
- **Assembler stage**

  It is the third stage of compilation. It takes the assembly source code and produces the corresponding object code.

- **Linking**

  It is the final stage of compilation. It takes one or more object files or libraries and linker script as input and combines them to produce a single executable file. In doing so, it resolves references to external symbols, assigns final addresses to procedures/functions and variables, and revises code and data to reflect new addresses (a process called relocation).

Basic techniques for reading and writing from I/O Port pins



Pin diagram of 8051 Microcontroller

```
;Toggle all bits of continuously.
        MOV A,#55
BACK:   MOV P2,A
        ACALL DELAY
        CPL A ;complement(inv) reg.A
        SJMP BACK
```

Basic techniques for reading and writing from I/O Port pins

The control of the 8051 ports is carried out using 8-bit latches (SFRs).

We can send some data to Port 1 as follows:

```
sfr P1 = 0x90;              // Usually in header file
P1 = 0x0F;                  // Write 00001111 to Port 1
```

In exactly the same way, we can read from Port 1 as follows:

```
unsigned char Port_data;
P1 = 0xFF; // Set the port to 'read mode'
Port_data = P1; // Read from the port
```

Basic techniques for reading and writing from I/O Port pins

After the 8051 microcontroller is reset, the port latches all have the value 0xFF (11111111 in binary): that is, all the port-pin latches are set to values of ‘1’. It is tempting to assume that writing data to the port is therefore unnecessary, and that we can get away with the following version:

```
unsigned char Port_data;
// Assume nothing written to port since reset
Port_data = P1;
```

Basic techniques for reading and writing from I/O Port pins

The problem with this code is that, in simple test programs it works: this can lull the developer into a false sense of security. If, at a later date, someone modifies the program to include a routine for writing to all or part of the same port, this code will not generally work as required:

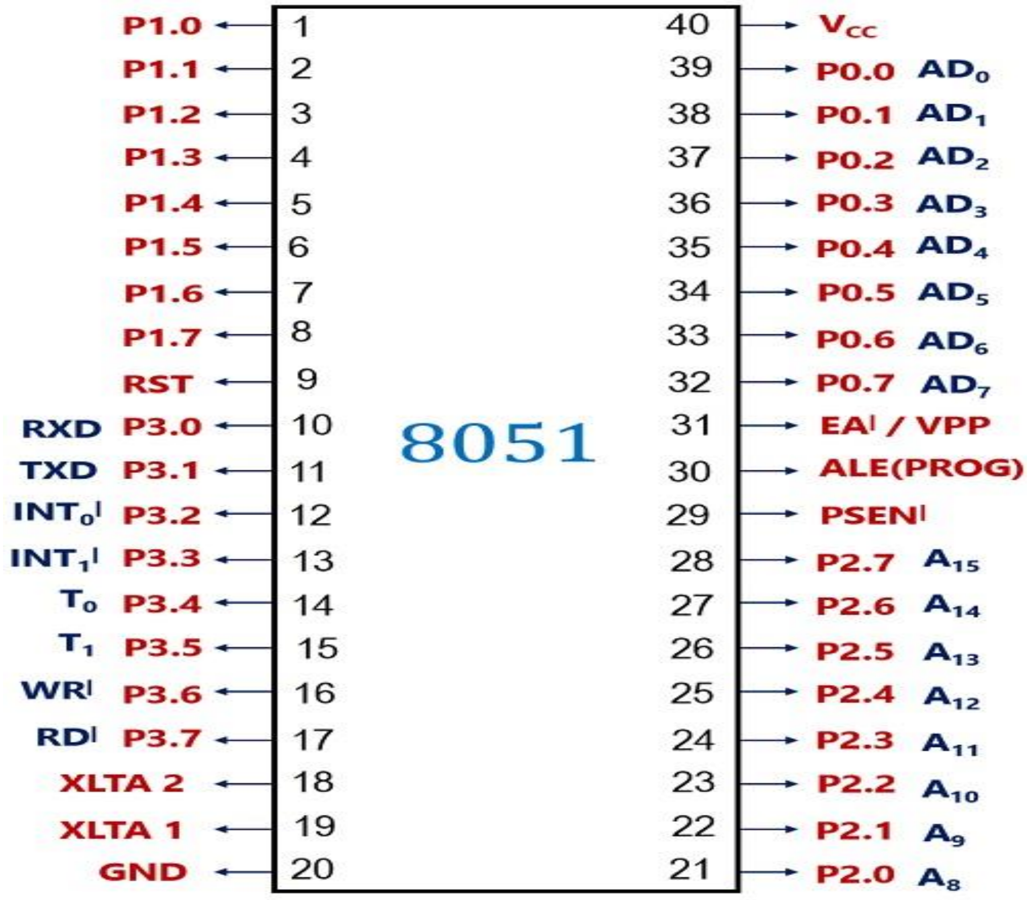
```
unsigned char Port_data;
P1 = 0x00;
            // Assumes nothing written to port since reset
            // – WON’T WORK
            Port_data = P1;
```

In most cases, initialization functions are used to set the port pins to a known state at the start of the program. Where this is not possible, it is safer to always write ‘1’ to any port pin before reading from it.

Basic techniques for reading and writing from I/O Port pins

- **Reading and writing bits:**
- Demonstrated how to read from or write to an entire port. However, suppose we have a **switch connected to Pin 1.1** and an **LED connected to Pin 2.1**.
- We might also have input and output devices connected to the other pins on Port 1.
- These pins may be used by totally different parts of the same system, and the code to access them may be produced by other team members, or other companies.
- It is therefore essential that we are able to read-from or write-to individual port pins without altering the values of other pins on the same port.
- We provided a simple example to illustrates how we can read from Pin 1.1, and write to Pin 2.1, without disrupting any other pins on this (or any other) port.

Basic techniques for reading and writing from I/O Port pins

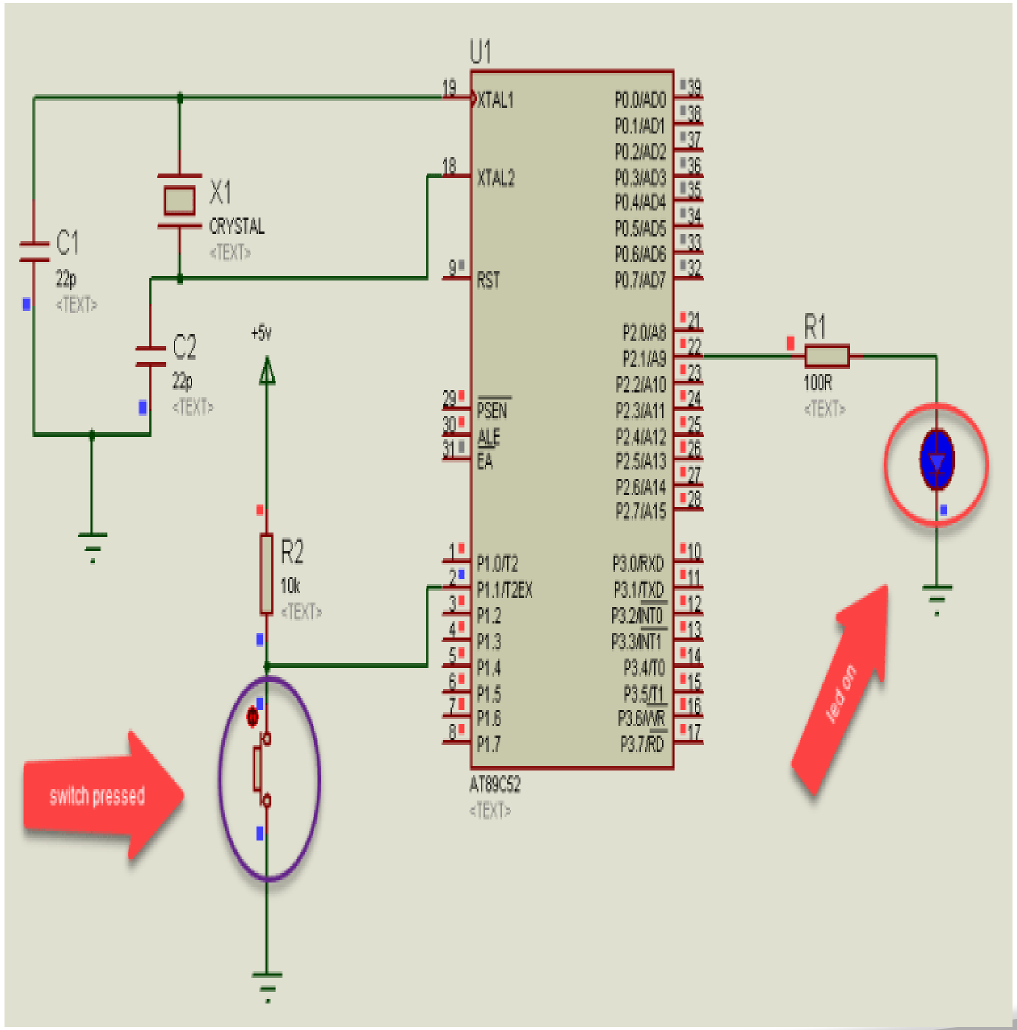


Pin diagram of 8051 Microcontroller

Electronics Desk

Basic techniques for reading and writing from I/O Port pins

LED BLINKING:



```
#include <REG51.H>
   sbit SW1 = P1^1;
   sbit LED = P2^1;
void main (void)
{
   LED = 0;
   while(1)
    {
   if(SW1 == 0)
    {
       BUZZER = 1;
     }
   else
     {
        BUZZER = 0;
      }
  }
}
```

Basic techniques for reading and writing from I/O Port pins

```
#include<reg51.h>
sbit Led = P2^1;       //pin connected to toggle Led
sbit Switch =P1^1;  //Pin connected to toggle led
void main(void)
{
Led = 0;          //configuring as output pin
Switch = 1;      //Configuring as input pin
while(1)         //Continuous monitor the status of
      the switch.
{
if(Switch == 0)
{
Led =1;         //Led On
}
```

```
else
{
Led =0; //Led Off
}
}
return 0;
}
```

Switch bounce

Switch bounce:

In an ideal world, this change in voltage obtained by connecting a switch to the In an ideal world, this change in voltage obtained by connecting a switch to the port pin of an 8051 microcontroller would take the form illustrated in below figure.

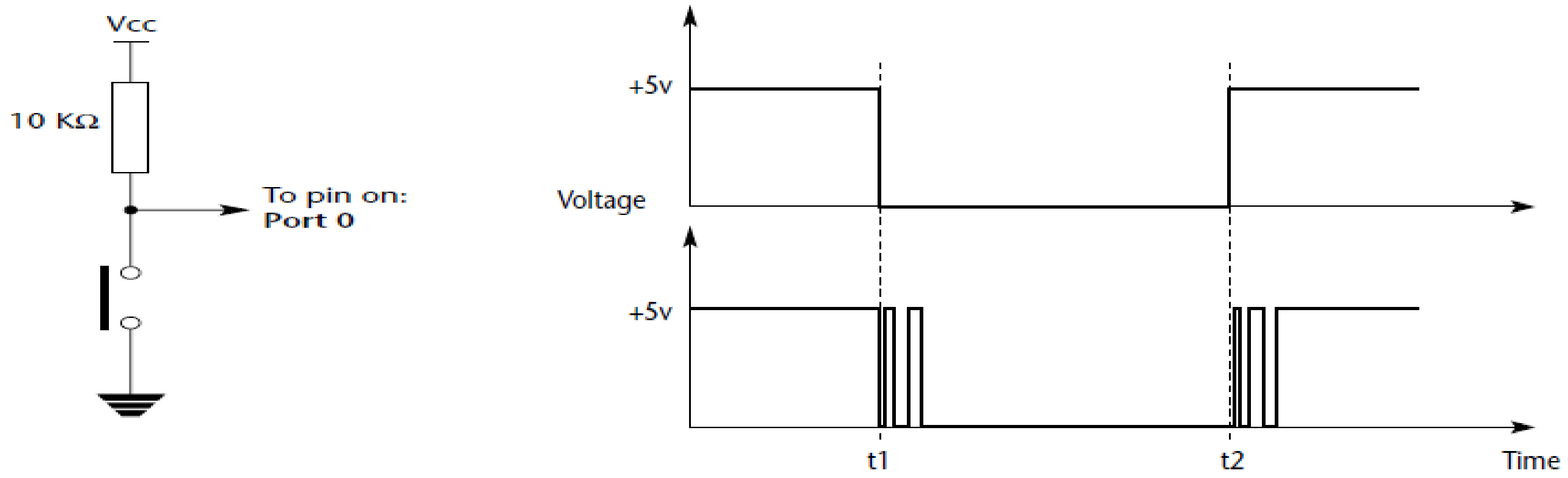
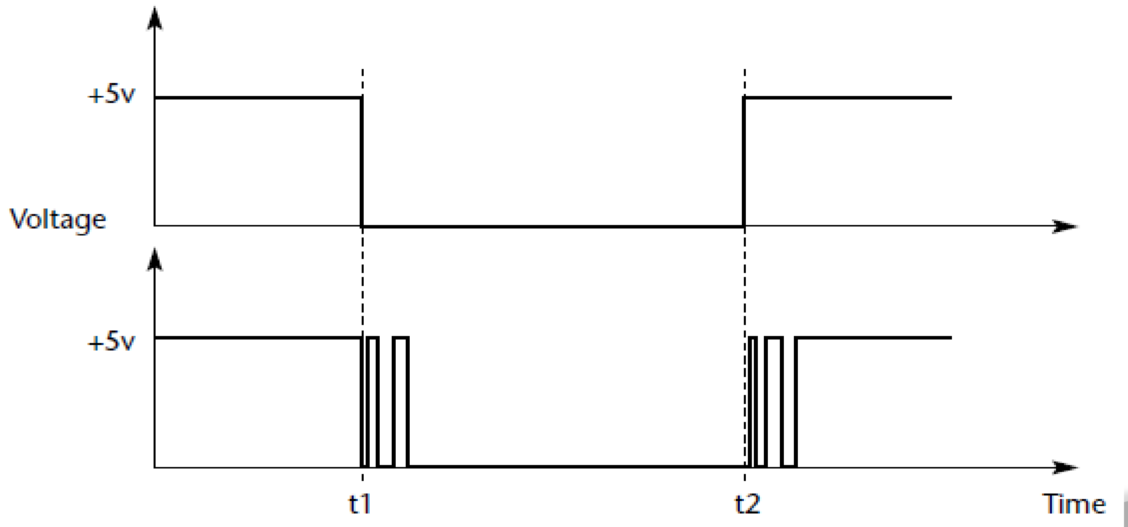


FIGURE: The voltage signal resulting from the switch shown in left side figure. Idealized waveform resulting from a switch depressed at time t1 and released at time t2. Actual waveform showing leading edge bounce following switch depression and trailing edge bounce following switch release

Switch bounce

➢In practice, all mechanical switch contacts *bounce (that is, turn on and off,* repeatedly, for a short period of time) after the switch is closed or opened. As a result, the actual input waveform looks more like that shown in below figure(bottom).

➢Usually, switches bounce for **less than 20 ms**: however large mechanical switches exhibit bounce behaviour for 50 ms or more.



Switch bounce

When you turn on the lights in your home or office with a mechanical switch, the switches will bounce. As far as humans are concerned, this bounce is imperceptible.

However, as far as the microcontroller is concerned, each ‘**bounce**’ is equivalent to **one press and release** of an ‘ideal’ switch.

Without appropriate software design, this can give rise to a number of problems, not least:

a. **Rather than reading ‘A’ from a keypad, we may read ‘AAAAA’.**
b. **Counting the number of times that a switch is pressed becomes extremely difficult.**
c. **If a switch is depressed once, and then released some time later, the ‘bounce’ may make it appear as if the switch has been pressed again (at the time of release).**

Binding and running embedded C program in Keil IDE

Binding and Running embedded C program in Keil IDE

Installing the Keil software and loading the project:

Rather than using the projects on the CD (where changes cannot be saved), please copy the files from the CD onto an appropriate directory on your hard disk.

Note: you will need to change the file properties after copying: files transferred from the CD will be ‘read only’.

When you have copied the files onto your hard disk, please run the Keil µVision application, and use the ‘Open Project’ option (from the ‘Project’ menu) to and work with your project.

Binding and Running embedded C program in Keil IDE

Configuring the simulator:

➢**Simulator**-µVision allows developers to execute and debug their **programs** on 8051 controller/Arm processor **simulations** without using a physical target and debug hardware.

➢Having loaded the 'Hello' project in the Keil µVision environment, we will begin by exploring the project settings.

➢First, using the Project menu, we will look at the 8051 device which we are intending to use for this application.

Binding and Running embedded C program in Keil IDE

Keil MicroVision is a free software which solves many of the pain points for an embedded program developer. This software is an integrated development environment (IDE), which integrated a text editor to write programs, a compiler and it will convert your source code to hex files too.

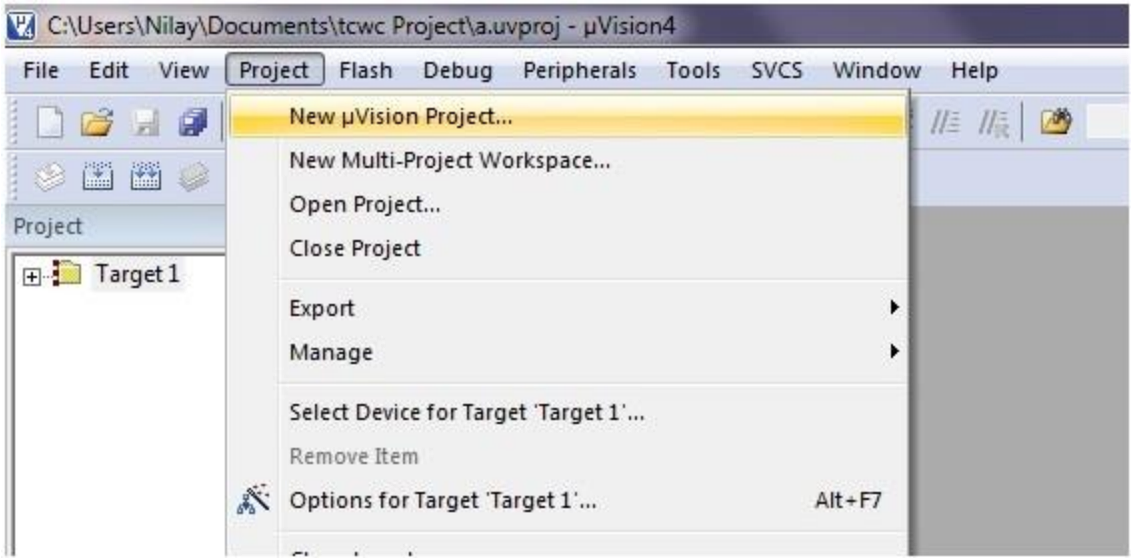
Here is simple guide to start working with Keil uVision which can be used for

- Writing programs in C/C++ or Assembly language
- Compiling and Assembling Programs
- Debugging program
- Creating Hex file
- Testing your program without Available real Hardware (Simulator Mode)

Binding and Running embedded C program in Keil IDE

<u>Step 1</u>: After opening Keil uV4, Go to **Project** tab and **Create new uVision project.**

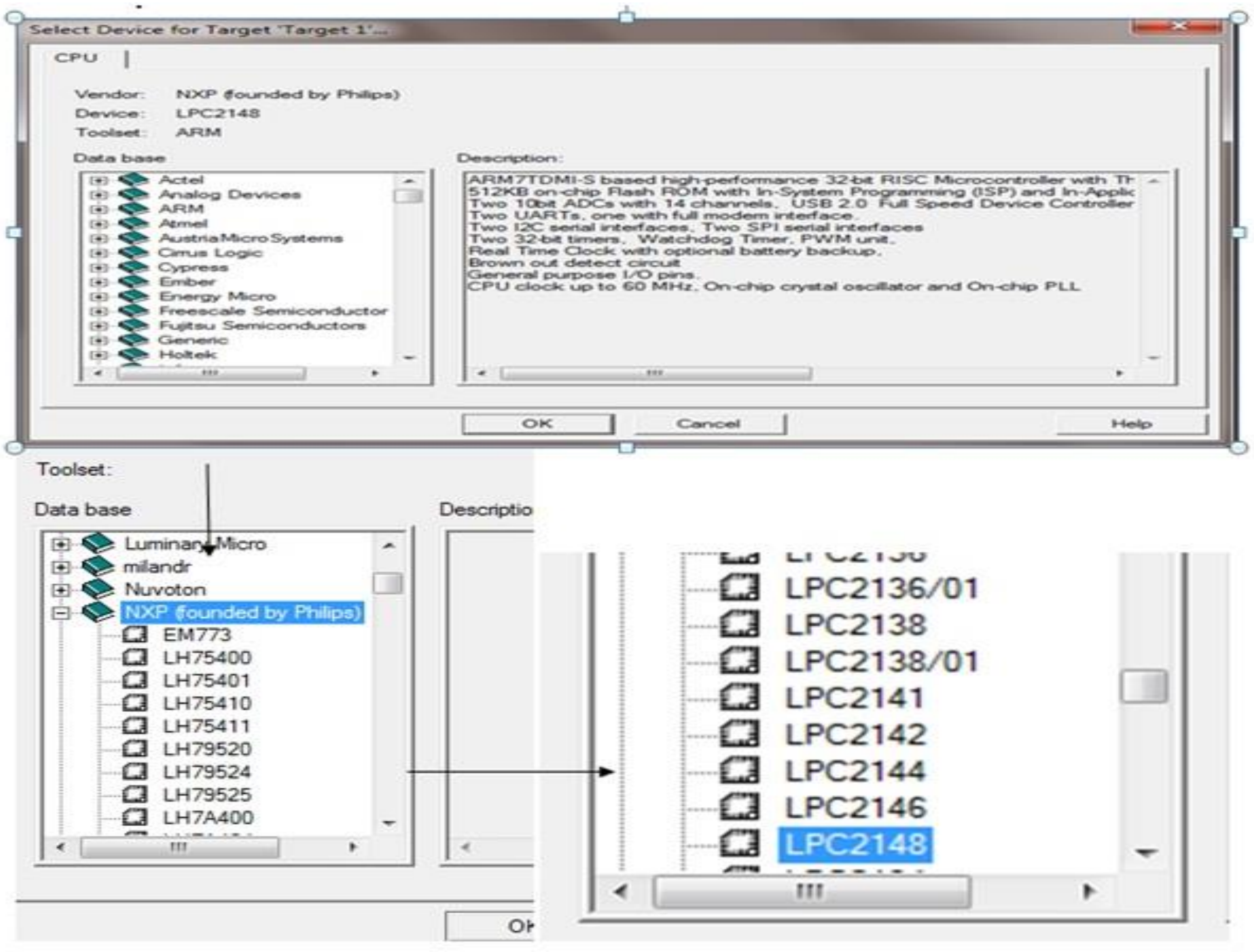
Now Select new folder and give name to Project.


C:\Users\Nilay\Documents\tcwc Project\a.uvproj - µVision4
File Edit View Project Flash Debug Peripherals Tools SVCS Window Help
New µVision Project...
New Multi-Project Workspace...
Open Project...
Close Project
Export
Manage
Select Device for Target 'Target 1'...
Remove Item
Options for Target 'Target 1'... Alt+F7
Project
Target 1

Binding and Running embedded C program in Keil IDE

Step 2: After Creating project now **Select your device model**.
Example.NXP-LPC2148 (or P89V51RD2 microcontroller).

Binding and Running embedded C program in Keil IDE



Vendor: NXP (founded by Philips)
Device: LPC2148
Toolset: ARM

Data base:
- Actel
- Analog Devices
- ARM
- Atmel
- AustriaMicroSystems
- Cirrus Logic
- Cypress
- Ember
- Energy Micro
- Freescale Semiconductor
- Fujitsu Semiconductors
- Generic
- Holtek

Description:

```
ARM7TDMI-S based high-performance 32-bit RISC Microcontroller with Th
512KB on-chip Flash ROM with In-System Programming (ISP) and In-Applic
Two 10bit ADCs with 14 channels, USB 2.0 Full Speed Device Controller
Two UARTs, one with full modem interface.
Two I2C serial interfaces, Two SPI serial interfaces
Two 32-bit timers, Watchdog Timer, PWM unit.
Real Time Clock with optional battery backup.
Brown out detect circuit
General purpose I/O pins.
CPU clock up to 60 MHz, On-chip crystal oscillator and On-chip PLL
```

Toolset:

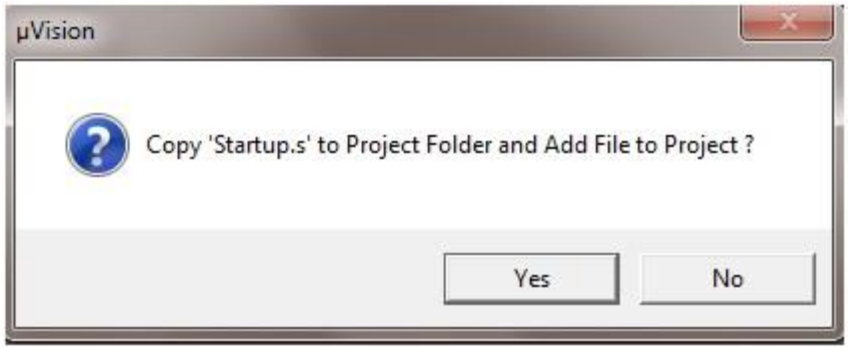
Data base
- Luminary Micro
- milandr
- Nuvoton
- NXP (founded by Philips)
  - EM773
  - LH75400
  - LH75401
  - LH75410
  - LH75411
  - LH79520
  - LH79524
  - LH79525
  - LH7A400

Descriptio

- LPC2136/01
- LPC2138
- LPC2138/01
- LPC2141
- LPC2142
- LPC2144
- LPC2146
- LPC2148

Binding and Running embedded C program in Keil IDE

- **Step 3**: so now your project is created and **Message** window will appear to add startup file of your Device click on **Yes** so it will be added to your project folder

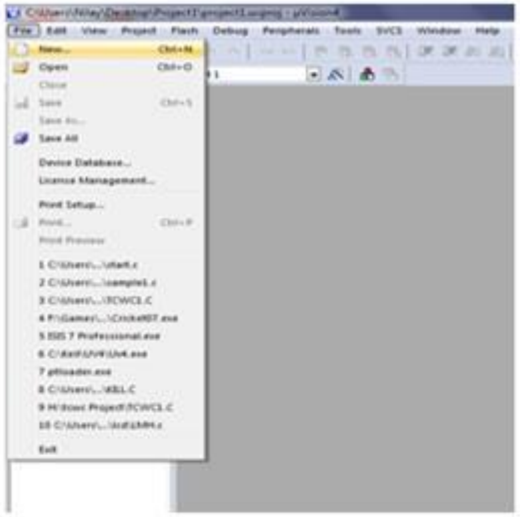

µVision

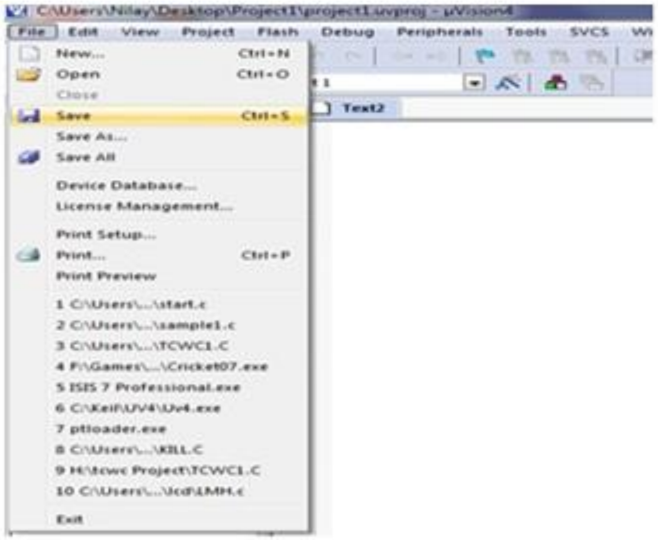
Copy 'Startup.s' to Project Folder and Add File to Project ?

Binding and Running embedded C program in Keil IDE

Step 4: Now go to File and create new file and save it with **.C** extension if you will write program in C language or save with **.asm** for **assembly** language.

i.e., **Led.c**





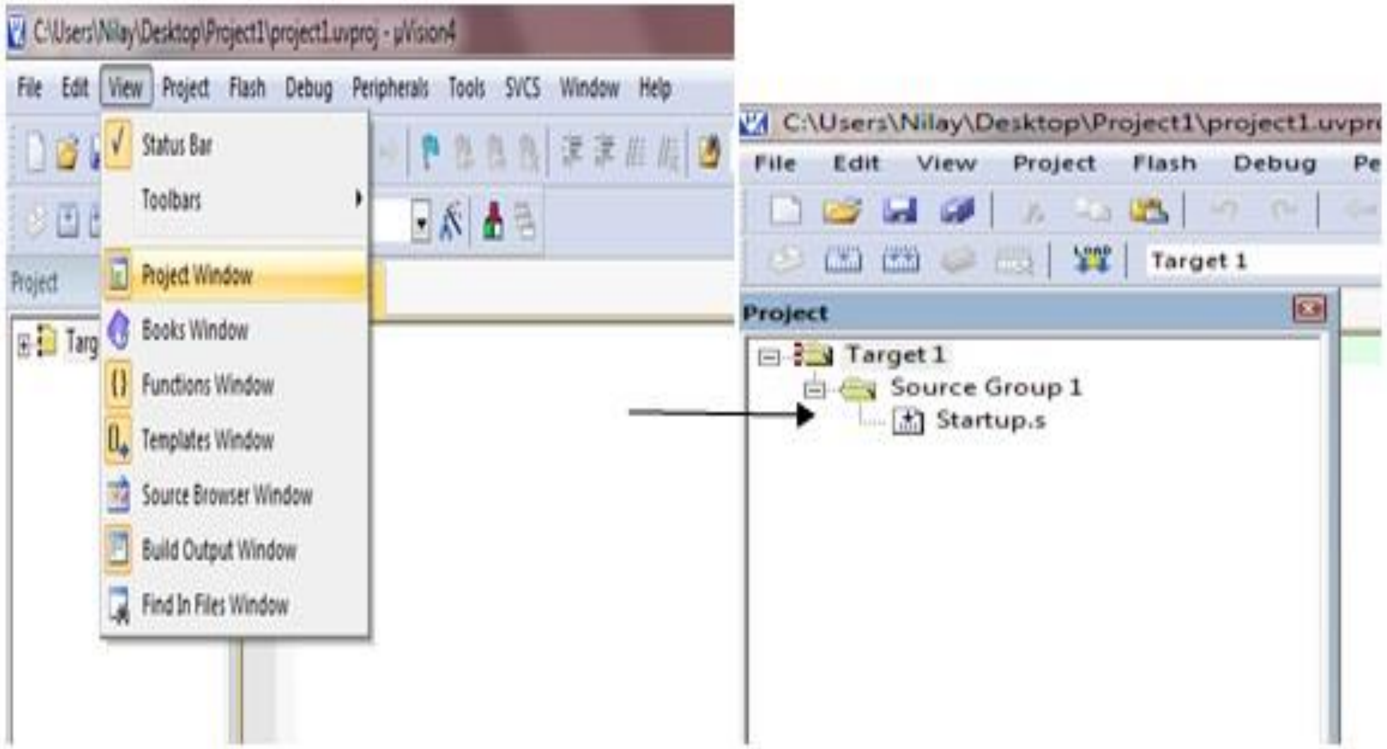
Binding and Running embedded C program in Keil IDE

Step 5: Now write your program and save it again. You can try example given at end of this tutorial.

Step 6: After that on left you see project window [if it’s not there….go to View tab and click on project window]

Now come on Project window.

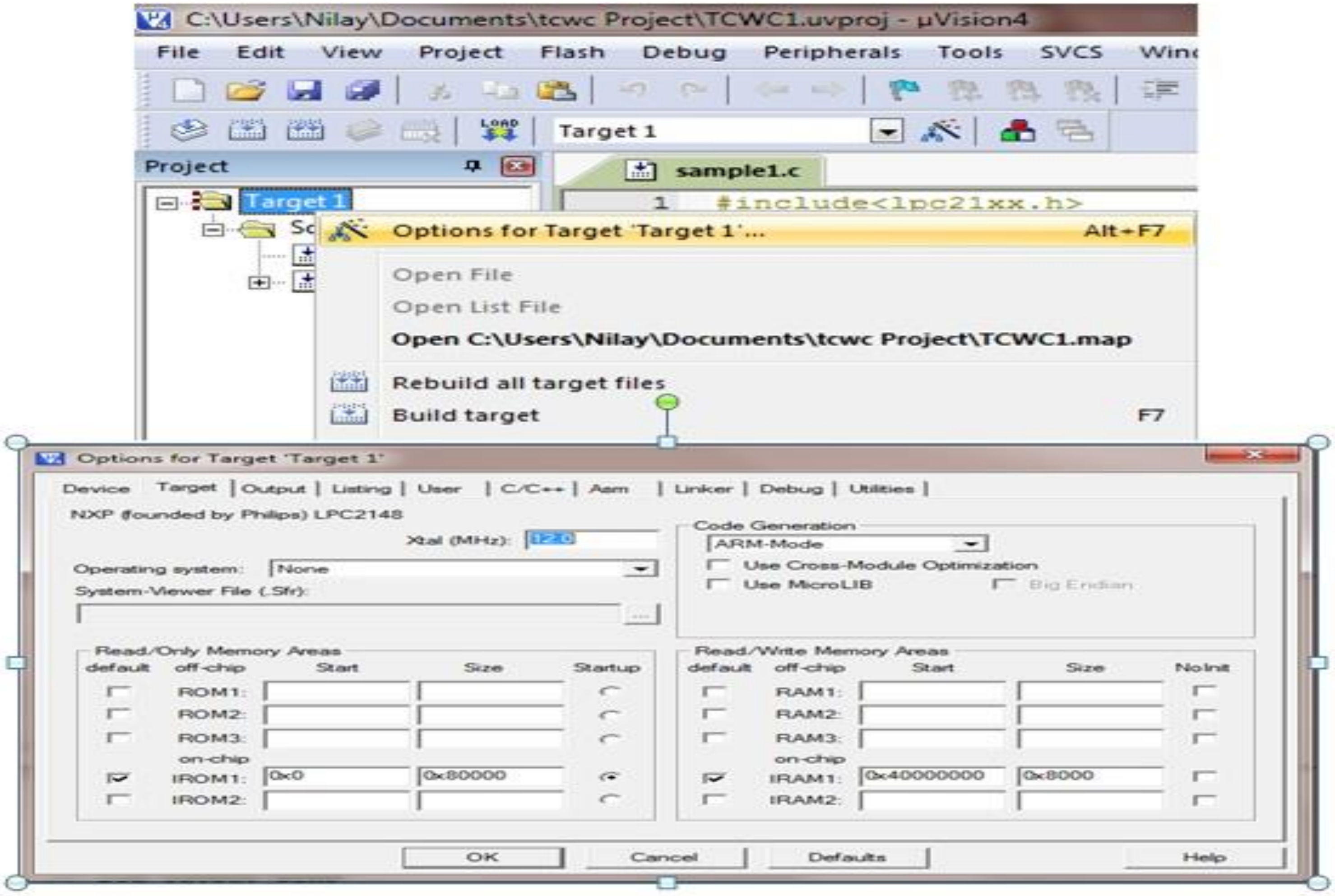
Binding and Running embedded C program in Keil IDE


C:\Users\Nilay\Desktop\Project1\project1.uvproj - µVision4
File Edit View Project Flash Debug Peripherals Tools SVCS Window Help
Status Bar
Toolbars
Project Window
Books Window
Functions Window
Templates Window
Source Browser Window
Build Output Window
Find In Files Window
Project
C:\Users\Nilay\Desktop\Project1\project1.uvpr
File Edit View Project Flash Debug Pe
Target 1
Project
Target 1
Source Group 1
Startup.s

Right click on target and click on **options for target**

Binding and Running embedded C program in Keil IDE

- Here you can change your device also.



```
1  #include<lpc21xx.h>
```

NXP (founded by Philips) LPC2148

Xtal (MHz): 12.0

Operating system: None

System-Viewer File (.Sfr):

Code Generation: ARM-Mode

Use Cross-Module Optimization

Use MicroLIB

Big Endian

Read/Only Memory Areas

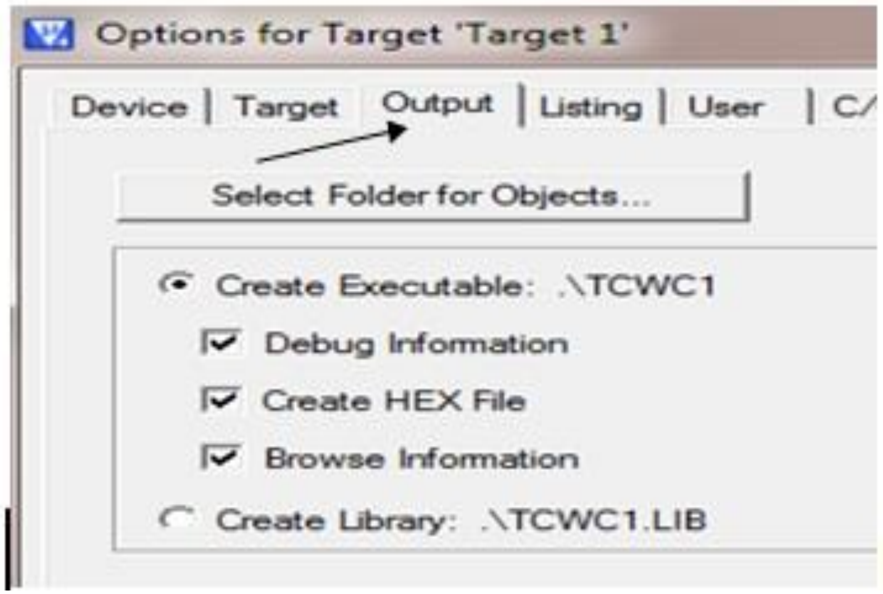
default	off-chip	Start	Size	Startup
	ROM1:			
	ROM2:			
	ROM3:			
	on-chip			
✓	IROM1:	0x0	0x80000	●
	IROM2:			

Read/Write Memory Areas

default	off-chip	Start	Size	NoInit
	RAM1:			
	RAM2:			
	RAM3:			
	on-chip			
✓	IRAM1:	0x40000000	0x8000	
	IRAM2:			

Binding and Running embedded C program in Keil IDE

- Click **output** tab here & check **create Hex file** if you want to generate hex file
- Now click on ok so it will save changes.

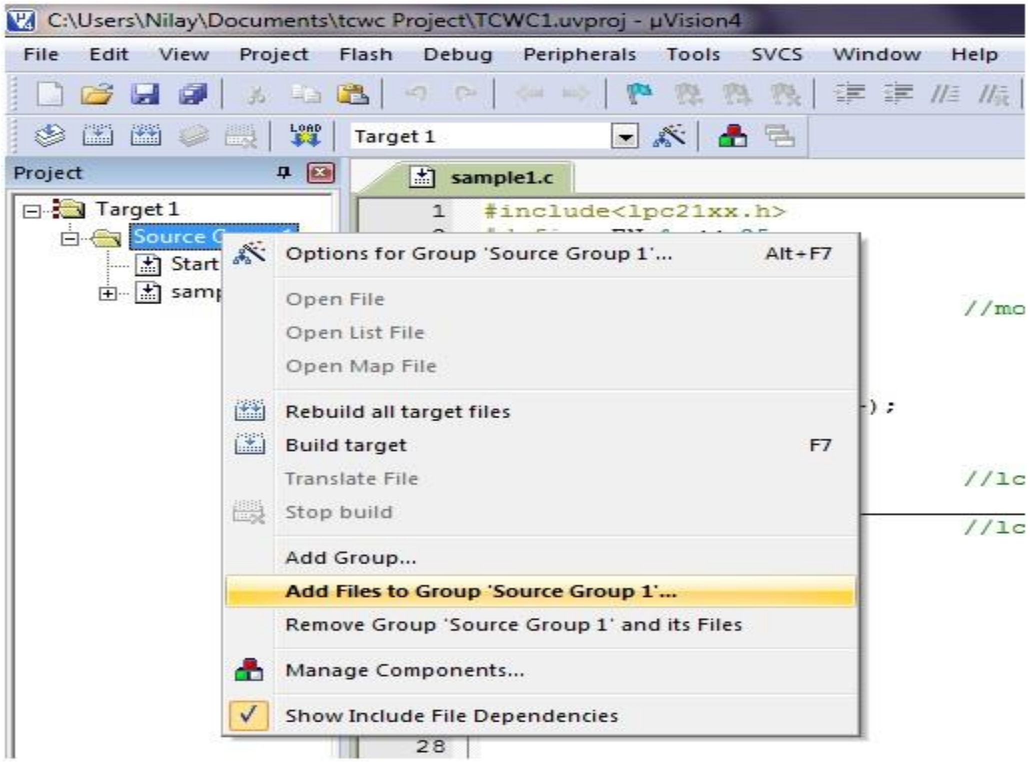


● Create Executable: .\TCWC1
✓ Debug Information
✓ Create HEX File
✓ Browse Information
○ Create Library: .\TCWC1.LIB

Binding and Running embedded C program in Keil IDE

Step 7: Now Expand target and you will see source group

Right click on group and click on **Add files to source group**


C:\Users\Nilay\Documents\tcwc Project\TCWC1.uvproj - µVision4
File Edit View Project Flash Debug Peripherals Tools SVCS Window Help
Target 1
Project
Target 1
Source G
Start
samp
sample1.c
Options for Group 'Source Group 1'... Alt+F7
Open File
Open List File
Open Map File
Rebuild all target files
Build target F7
Translate File
Stop build
Add Group...
Add Files to Group 'Source Group 1'...
Remove Group 'Source Group 1' and its Files
Manage Components...
Show Include File Dependencies

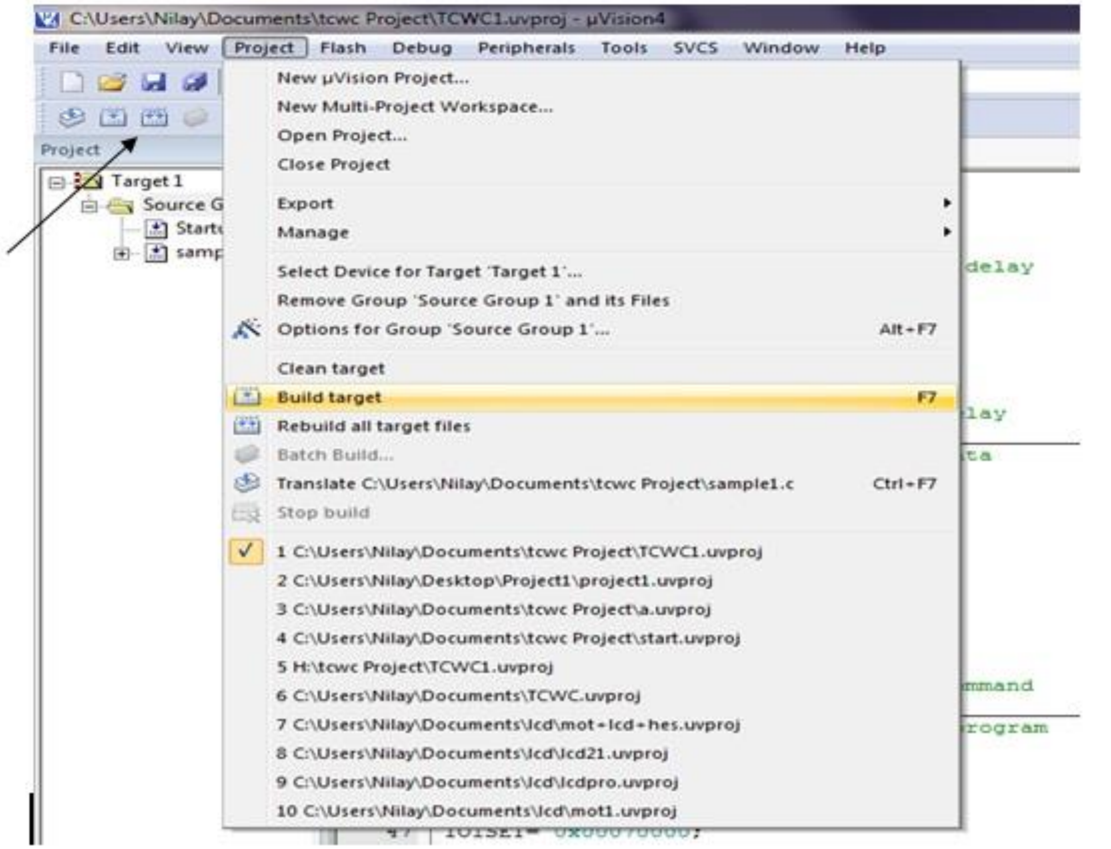
```
#include<lpc21xx.h>
```

•Now add your program file which you have written in C/assembly.

•You can see program file added under source group.

Binding and Running embedded C program in Keil IDE

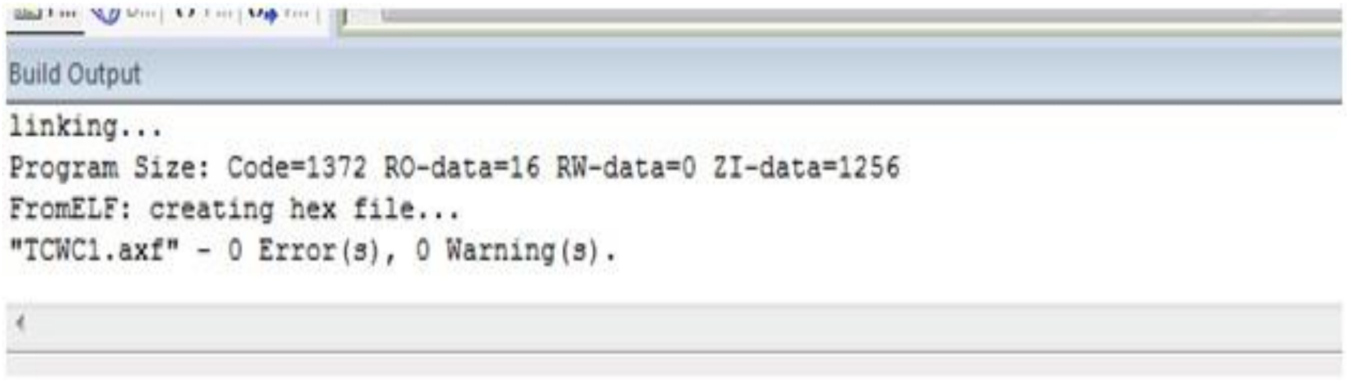
- **Step 8**: Now Click on **Build target**. You can find it under Project tab or in tool bar. It can also be done by pressing **F7** key.


C:\Users\Nilay\Documents\tcwc Project\TCWC1.uvproj - µVision4
File Edit View Project Flash Debug Peripherals Tools SVCS Window Help
Project
Target 1
Source G
Start
samp
New µVision Project...
New Multi-Project Workspace...
Open Project...
Close Project
Export
Manage
Select Device for Target 'Target 1'...
Remove Group 'Source Group 1' and its Files
Options for Group 'Source Group 1'... Alt+F7
Clean target
Build target F7
Rebuild all target files
Batch Build...
Translate C:\Users\Nilay\Documents\tcwc Project\sample1.c Ctrl+F7
Stop build
1 C:\Users\Nilay\Documents\tcwc Project\TCWC1.uvproj
2 C:\Users\Nilay\Desktop\Project1\project1.uvproj
3 C:\Users\Nilay\Documents\tcwc Project\a.uvproj
4 C:\Users\Nilay\Documents\tcwc Project\start.uvproj
5 H:\tcwc Project\TCWC1.uvproj
6 C:\Users\Nilay\Documents\TCWC.uvproj
7 C:\Users\Nilay\Documents\lcd\mot-lcd-hes.uvproj
8 C:\Users\Nilay\Documents\lcd\lcd21.uvproj
9 C:\Users\Nilay\Documents\lcd\lcdpro.uvproj
10 C:\Users\Nilay\Documents\lcd\mot1.uvproj

Binding and Running embedded C program in Keil IDE

Step 9: you can see Status of your program in **Build output** window

[If it’s not there go to view and click on Build output window]



```
linking...
Program Size: Code=1372 RO-data=16 RW-data=0 ZI-data=1256
FromELF: creating hex file...
"TCWC1.axf" - 0 Error(s), 0 Warning(s).
```

Now you are done with your program and you can now debug and Simulate your Program.

Dissecting the program

Dissecting the program:

So far we have simply demonstrated the operation of the Keil 8051 simulator.

We now begin to consider how the code in actually works.

The complete program:

The complete ‘Hello, Embedded World’ program is shown in below.

```
#include <reg51.h>
sbit LED_pin = P1^5;                      // LED is to be connected to this pin
bit LED_state_G;                          // Stores the LED state
void LED_FLASH_Init(void);
void LED_FLASH_Change_State(void);             // Function prototypes
void DELAY_LOOP_Wait(const unsigned int);
```

Dissecting the program

```
void main(void)
{
LED_FLASH_Init();
while(1)
{
LED_FLASH_Change_State();   // Change the LED state (OFF to ON, or vice versa)
DELAY_LOOP_Wait(1000);      // Delay for *approx* 1000 ms
}
}
/*------------------------------------------------------------*-
LED_FLASH_Init()
Prepare for LED_Change_State() function – see below.
-*------------------------------------------------------------*/
void LED_FLASH_Init(void)
{
LED_state_G = 0;
}
```

Dissecting the program

```
/*--------------------------------------------------------------*-
LED_FLASH_Change_State()
Changes the state of an LED (or pulses a buzzer, etc) on a specified port pin. Must call at twice the
required flash rate: thus, for 1 Hz flash (on for 0.5 seconds, off for 0.5 seconds), this function must be
called twice a second.
-*--------------------------------------------------------------*/
void LED_FLASH_Change_State(void)
{
if (LED_state_G == 1)                // Change the LED from OFF to ON (or vice versa)
{
LED_state_G = 0;
LED_pin = 0;
}
else
{
LED_state_G = 1;
LED_pin = 1;
}
}
```

Dissecting the program

```
/*------------------------------------------------------------*-
DELAY_LOOP_Wait()
Delay duration varies with parameter. Parameter is, *ROUGHLY*, the delay, in
milliseconds, on 12MHz 8051 (12 osc cycles). You need to adjust the timing for
your application!
-*------------------------------------------------------------*/
void DELAY_LOOP_Wait(const unsigned int DELAY)
{
unsigned int x, y;
for (x = 0; x <= DELAY; x++)
{
for (y = 0; y <= 120; y++);
}
}
```

Function Calls

Function Calls:

➢The ARM Procedure Call Standard (APCS) defines how to pass function arguments and return values in ARM registers. The more recent ARM-Thumb Procedure Call Standard (ATPCS) covers ARM and Thumb interworking as well.

➢The first four integer arguments are passed in the first four ARM registers: *r0, r1, r2,* and *r3. Subsequent integer arguments are placed on the full descending stack, ascending in* memory as in figure. Function return integer values are passed in *r0.*

Function Calls

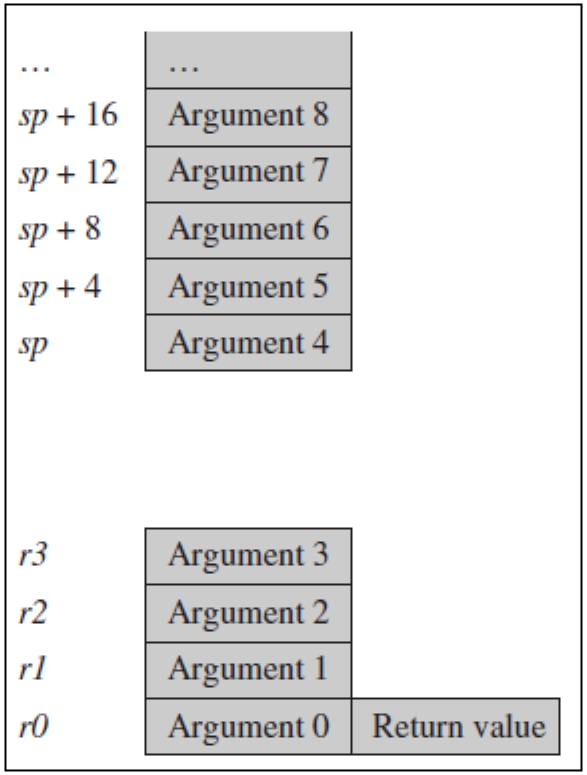


Figure: ATPCS argument passing

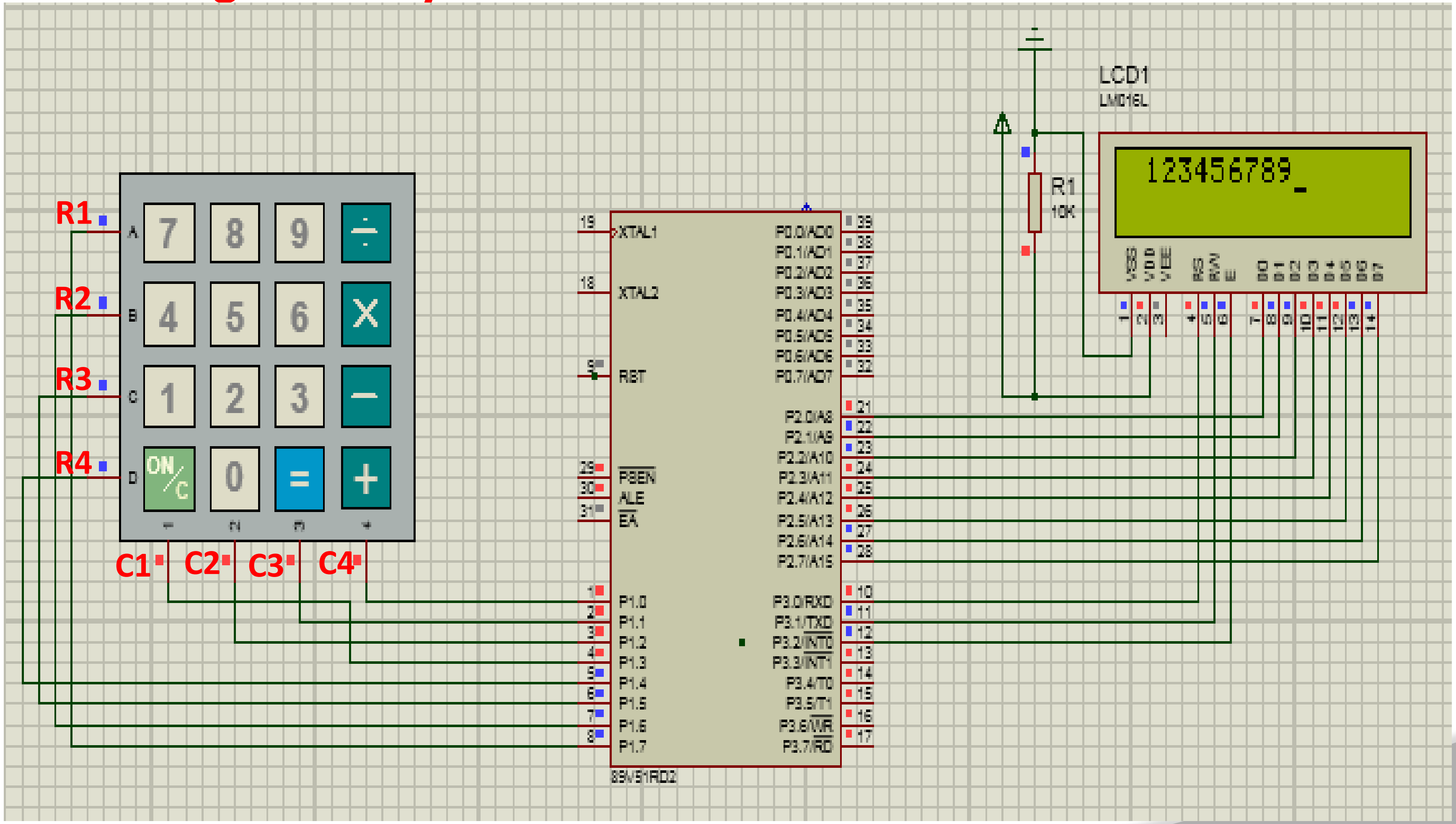
This description covers only integer or pointer arguments. Two-word arguments such as long long or double are passed in a pair of consecutive argument registers and returned in *r0, r1. The compiler may pass structures in registers or by reference according to command* line compiler options.

Function Calls

- The first point to note about the procedure call standard is the *four-register rule.* Functions with four or fewer arguments are far more efficient to call than functions with five or more arguments.
- For functions with four or fewer arguments, the compiler can pass all the arguments in registers.
- For functions with more arguments, both the caller and callee must access the stack for some arguments. Note that for C++ the first argument to an object method is the *this pointer. This argument is implicit and additional to the* explicit arguments.
- If your C function needs more than four arguments, or your C++ method more than three explicit arguments, then it is almost always more efficient to use structures.
- Group related arguments into structures, and pass a structure pointer rather than multiple arguments. Which arguments are related will depend on the structure of your software.

Interfacing Keyboard with 8051

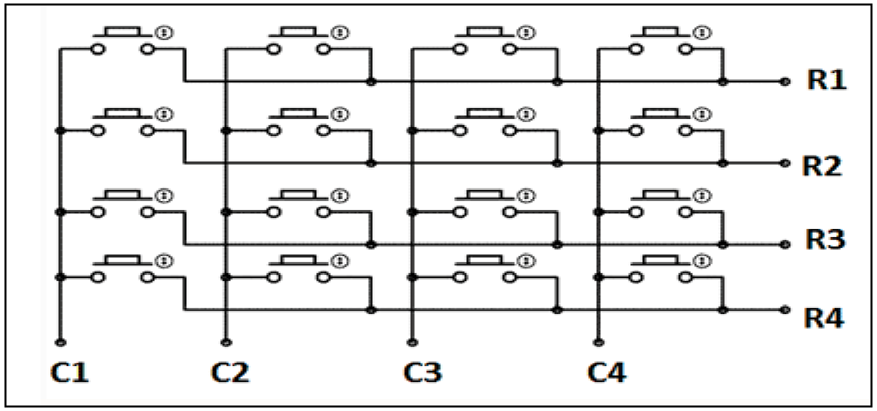
Interfacing with Keyboard:



Interfacing Keyboard with 8051

4X4 Matrix Keyboard

- Matrix keypad/Keyboard consists of set of Push buttons, which are interconnected. Like in our case we are using 4X4 matrix keyboard, in which there are 4 push buttons in each of four rows. And the terminals of the push buttons are connected according to diagram.
- In first row, one terminal of all the 4 push buttons are connected together and another terminal of 4 push buttons are representing each of 4 columns, same goes for each row. So we are getting 8 terminals to connect with a microcontroller.



Interfacing Keyboard with 8051

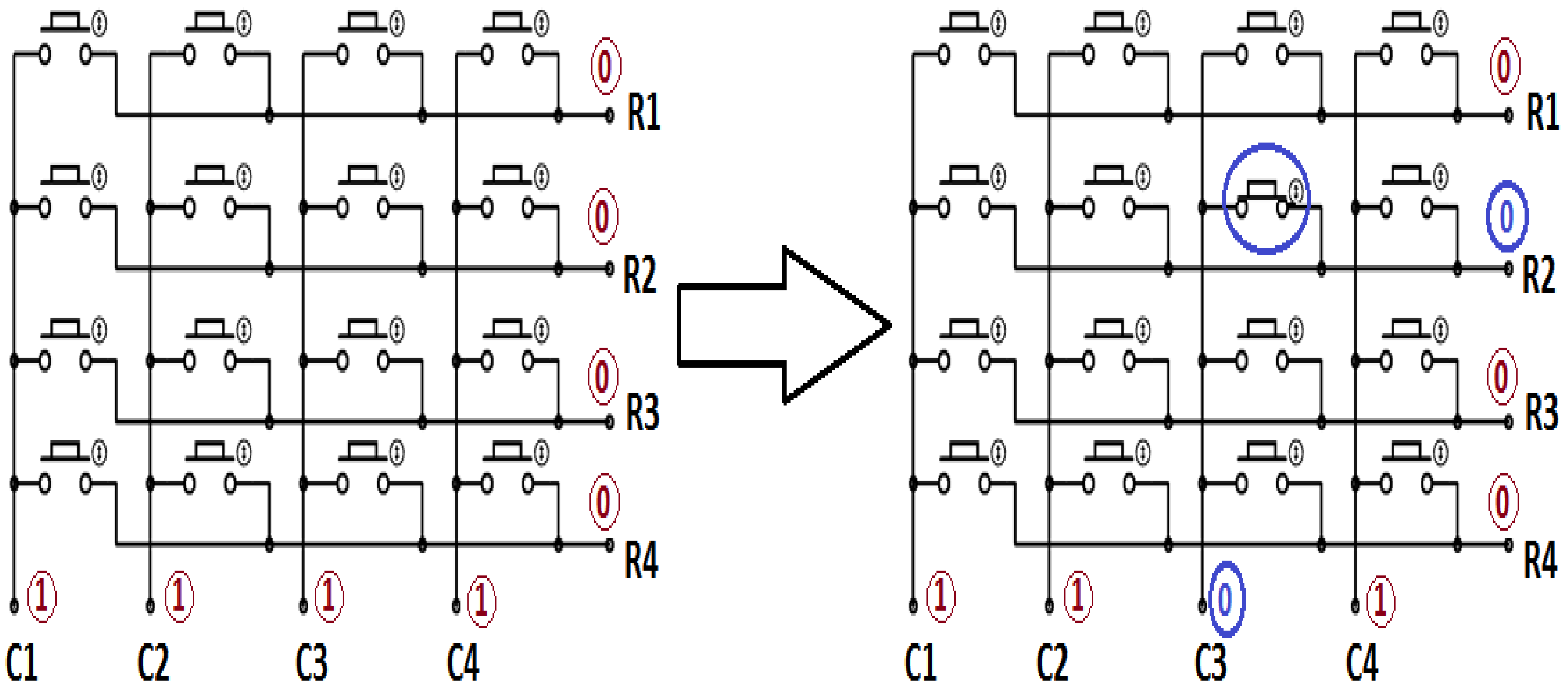
Whenever any button is pressed we need to get the location of the button, means the corresponding ROW and COLUMN no. Once we get the location of the button, we can print the character accordingly.

Now the question is how to get the location of the pressed button? I am going to explain this in below steps.

1. First we have made all the Rows to Logic level 0 and all the columns to Logic level 1.

2. Whenever we press a button, column and row corresponding to that button gets shorted and makes the corresponding column to logic level 0. Because that column becomes connected (shorted) to the row, which is at Logic level 0. So we get the column no. See main() function.

Interfacing Keyboard with 8051



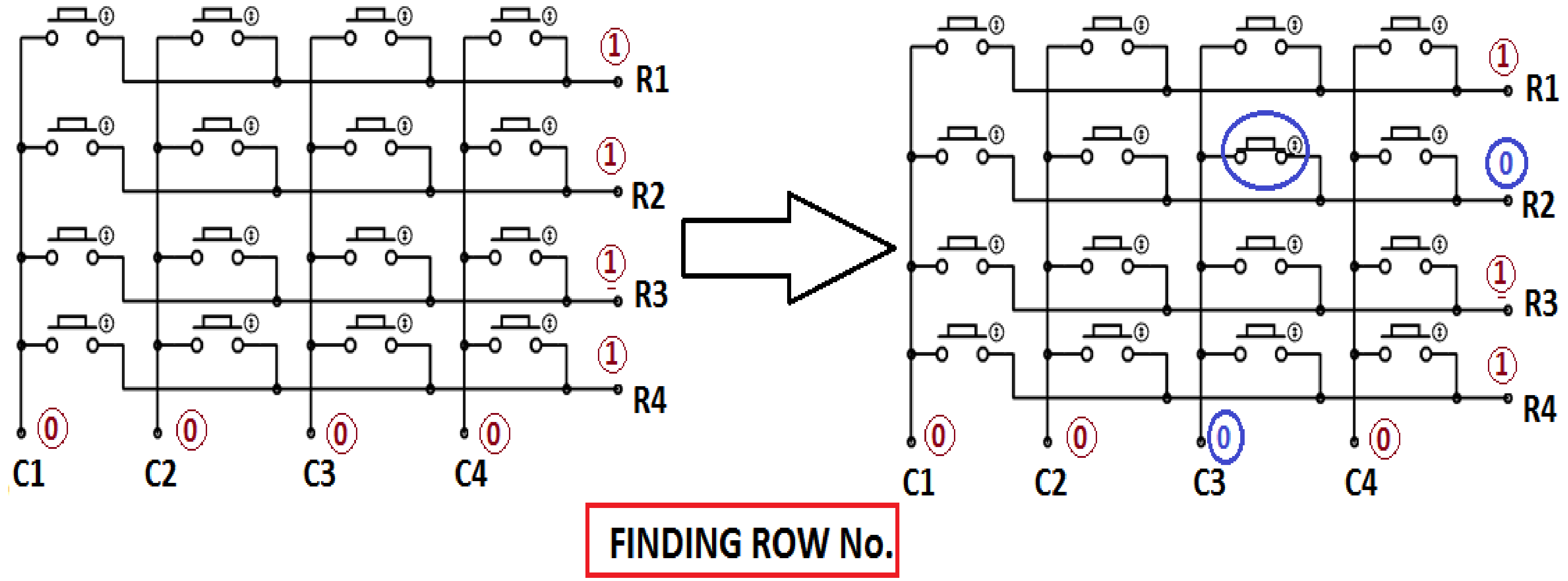
FINDING COLUMN No.

Interfacing Keyboard with 8051

3. Now we need to find the Row no., so we have created four functions corresponding to each column. Like if any button of column one is pressed, we call function row_finder1(), to find the row no.

4. In row_finder1() function, we reversed the logic levels, means now all the Rows are 1 and columns are 0. Now Row of the pressed button should be 0 because it has become connected (shorted) to the column whose button is pressed, and all the columns are at 0 logic. So we have scanned all rows for 0.

Interfacing Keyboard with 8051



5. So whenever we find the Row at logic 0, means that is the row of pressed button. So now we have column no (got in step 2) and row no., and we can print no. of that button using lcd_data function.

Same procedure follows for every button press, and we are using while(1), to continuously check, whether button is pressed or not.

Interfacing Keyboard with 8051

```
#include<reg51.h>
#define display_port P2      //Data pins connected to port 2 on microcontroller
sbit rs = P3^0;                        //RS pin connected to pin 2 of port 3
sbit rw = P3^1;                        // RW pin connected to pin 3 of port 3
sbit e =  P3^2;                         //E pin connected to pin 4 of port 3

sbit C4 = P1^0;                        // Connecting keypad to Port 1
sbit C3 = P1^1;
sbit C2 = P1^2;
sbit C1 = P1^3;
sbit R4 = P1^4;
sbit R3 = P1^5;
sbit R2 = P1^6;
sbit R1 = P1^7;
```

Interfacing Keyboard with 8051

```
void msdelay(unsigned int time)  // Function for creating delay in milliseconds.
{
   unsigned i,j ;
   for(i=0;i<time;i++)
   for(j=0;j<1275;j++);
}
void lcd_cmd(unsigned char command)  //Function to send command instruction
     to LCD
{
   display_port = command;
   rs= 0;
   rw=0;
   e=1;
   msdelay(1);
   e=0;
}
```

Interfacing Keyboard with 8051

```
void lcd_data(unsigned char disp_data)  //Function to send display data to LCD
{
    display_port = disp_data;
    rs= 1;
    rw=0;
    e=1;
    msdelay(1);
    e=0;
}
```

Interfacing Keyboard with 8051

```
void lcd_init()                //Function to prepare the LCD  and get it ready
{
        lcd_cmd(0x38);     // for using 2 lines and 5X7 matrix of LCD
        msdelay(10);
        lcd_cmd(0x0F);     // turn display ON, cursor blinking
        msdelay(10);
        lcd_cmd(0x01);     //clear screen
        msdelay(10);
        lcd_cmd(0x81);     // bring cursor to position 1 of line 1
        msdelay(10);
}
```

Interfacing Keyboard with 8051

```
void row_finder1() //Function for finding the row for column 1
{
R1=R2=R3=R4=1;
C1=C2=C3=C4=0;

if(R1==0)
lcd_data('7');
if(R2==0)
lcd_data('4');
if(R3==0)
lcd_data('1');
if(R4==0)
lcd_data('N');
}
```

Interfacing Keyboard with 8051

```
void row_finder2()            //Function for finding the row for column 2
{
R1=R2=R3=R4=1;
C1=C2=C3=C4=0;

if(R1==0)
lcd_data('8');
if(R2==0)
lcd_data('5');
if(R3==0)
lcd_data('2');
if(R4==0)
lcd_data('0');
}
```

Interfacing Keyboard with 8051

```
void row_finder3() //Function for finding the row for column 3
{
R1=R2=R3=R4=1;
C1=C2=C3=C4=0;

if(R1==0)
lcd_data('9');
if(R2==0)
lcd_data('6');
if(R3==0)
lcd_data('3');
if(R4==0)
lcd_data('=');
}
```

Interfacing Keyboard with 8051

```
void row_finder4() //Function for finding the row for column 4
{
R1=R2=R3=R4=1;
C1=C2=C3=C4=0;

if(R1==0)
lcd_data('%');
if(R2==0)
lcd_data('*');
if(R3==0)
lcd_data('-');
if(R4==0)
lcd_data('+');
}
```

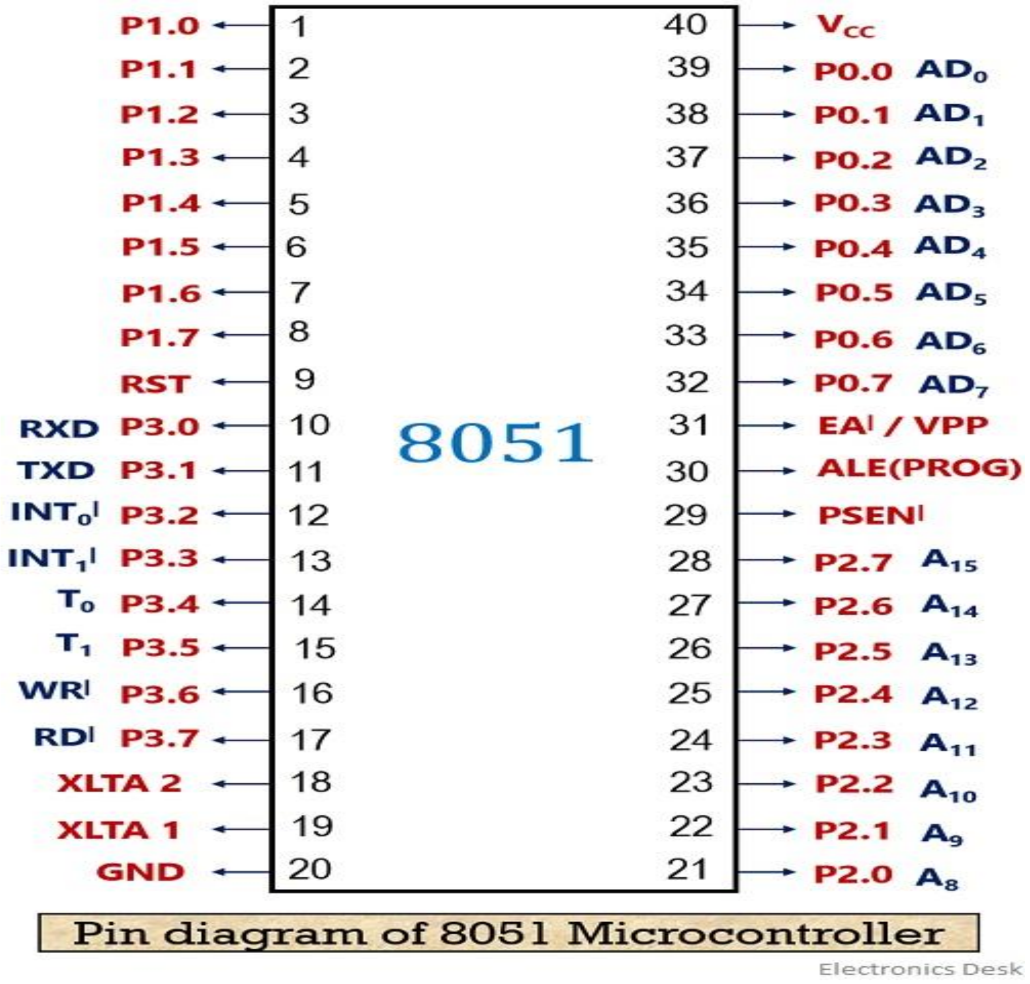
Interfacing Keyboard with 8051

```
void main()
{
   lcd_init();
   while(1)
   {
      msdelay(30);
      C1=C2=C3=C4=1;

     R1=R2=R3=R4=0;
       if(C1==0)
       row_finder1();
       else if(C2==0)
        row_finder2();
       else if(C3==0)
       row_finder3();
             else if(C4==0)
             row_finder4();
          }

       }
```

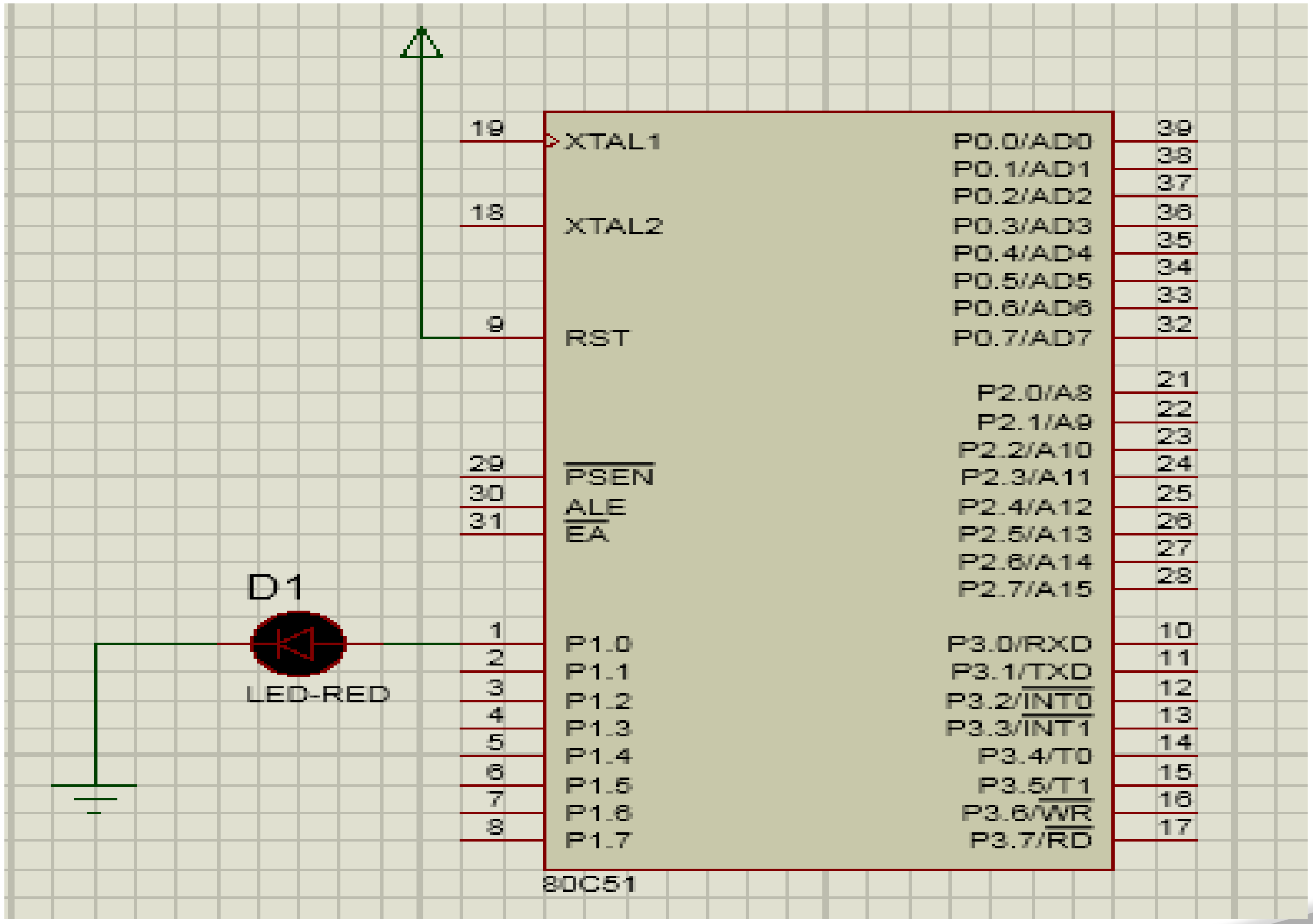
8051 Microcontroller



Pin diagram of 8051 Microcontroller

LED Interfacing with 8051

- **LED interfacing with 8051 to a single Pin:**



LED Interfacing with 8051

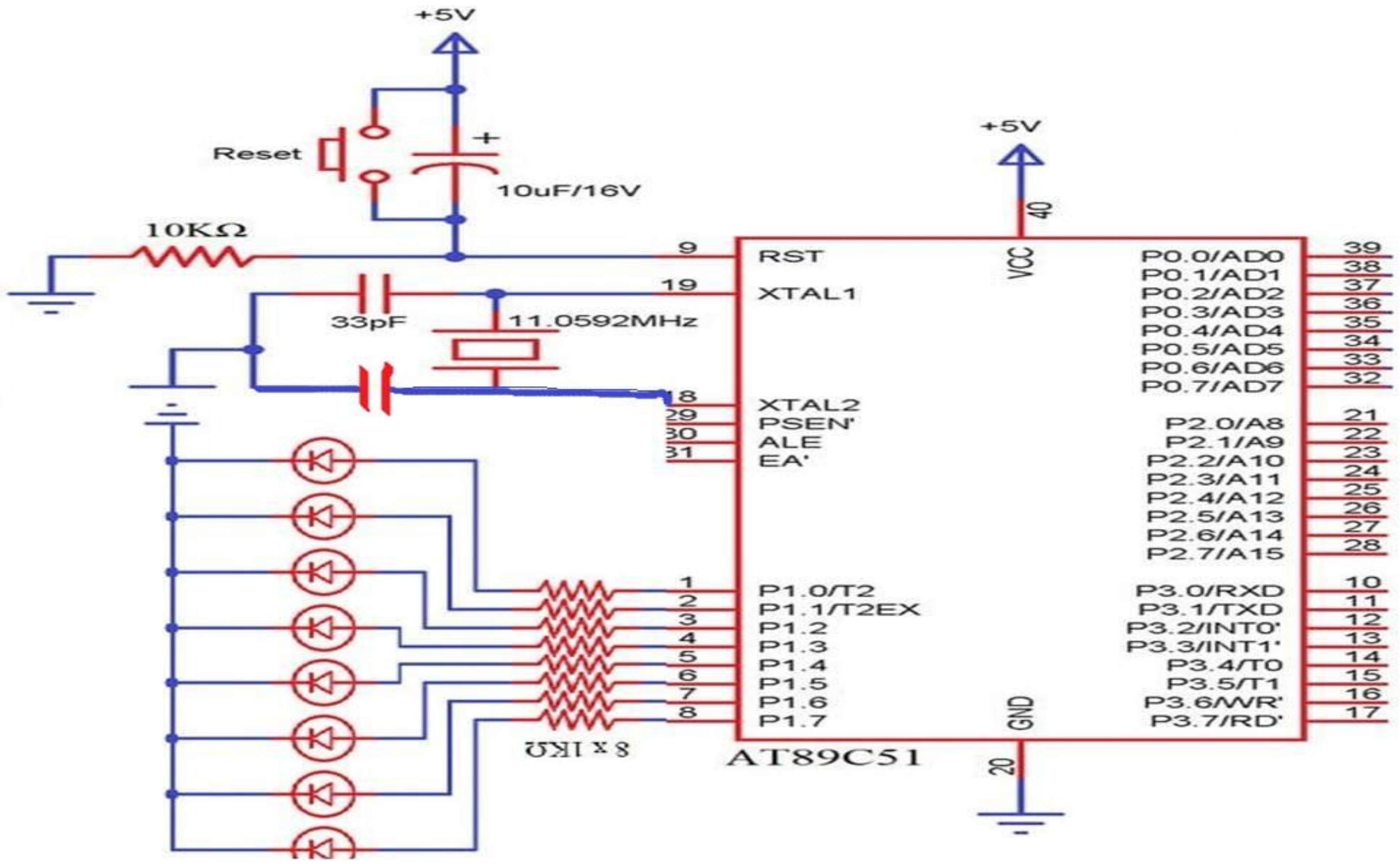
```
#include<reg51.h>          // special function register declarations
sbit LED = P2^0;           // Defining LED pin
void Delay(void);          // Function prototype declaration
void main (void)
{
while(1)                   // infinite loop
     {
     LED = 0;              // LED ON
     Delay();
     LED = 1;              // LED OFF
     Delay();
     }
}
```

LED Interfacing with 8051

```
void Delay(void)
    {
    int j;
    int i;
    for(i=0;i<10;i++)
       {
          for(j=0;j<10000;j++)
          {
          }
       }
     }
```

LED Interfacing with 8051

LED's interfacing with Port, P1 of 8051:



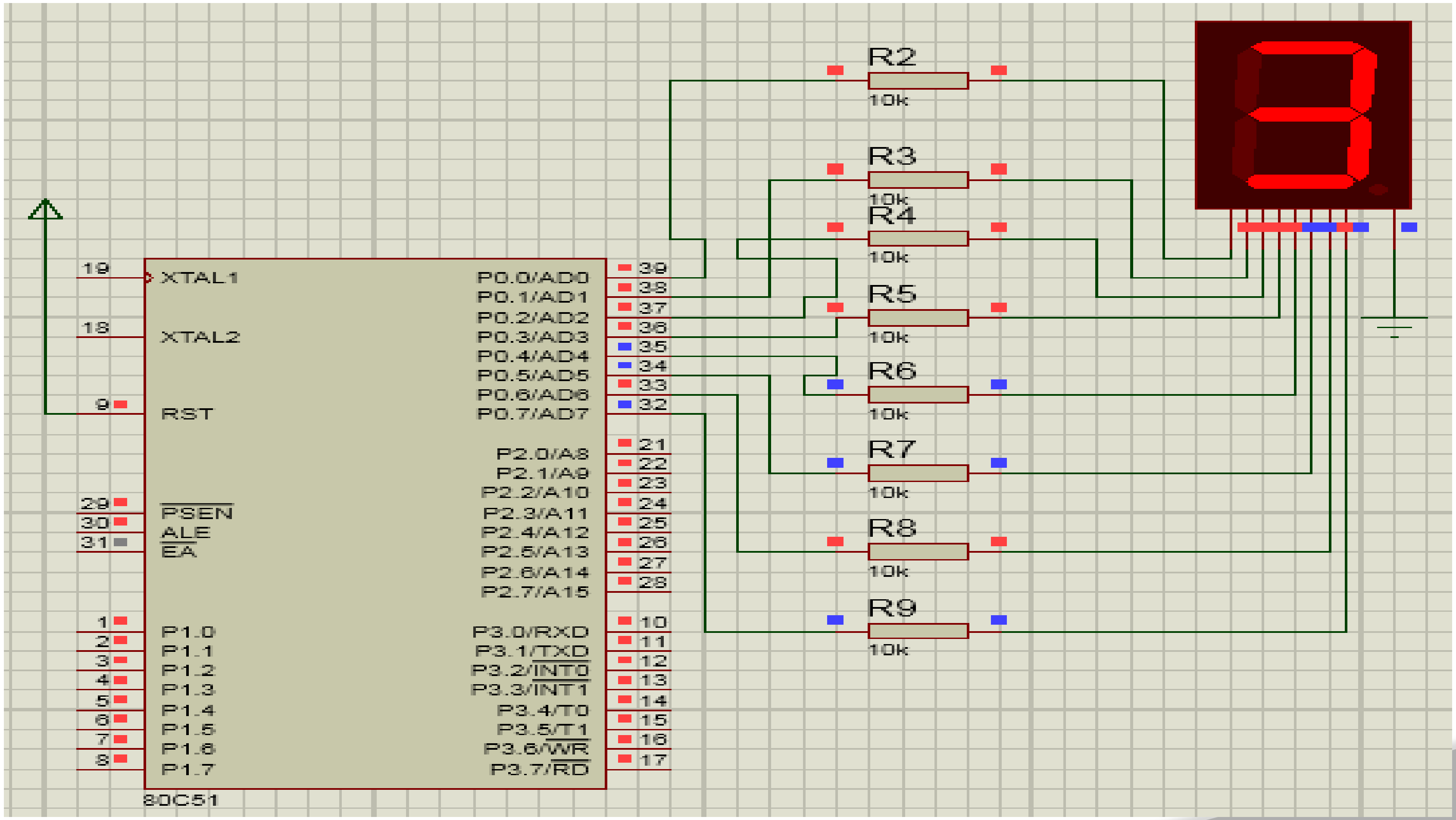
LED Interfacing with 8051

```
#include<REG51.H>
#define LEDPORT P1
void delay(unsigned int);
void main(void)
{
    LEDPORT =0x00;
    while(1)
      {
      LEDPORT = 0X00;
      delay(250);
      LEDPORT = 0xff;
      delay(250);
      }
}
```

```
void delay(unsigned int itime)
{
    unsigned int i,j;
    for(i=0;i<itime;i++)
    {
        for(j=0;j<250;j++);
    }
}
```

7 Segment Display Interfacing

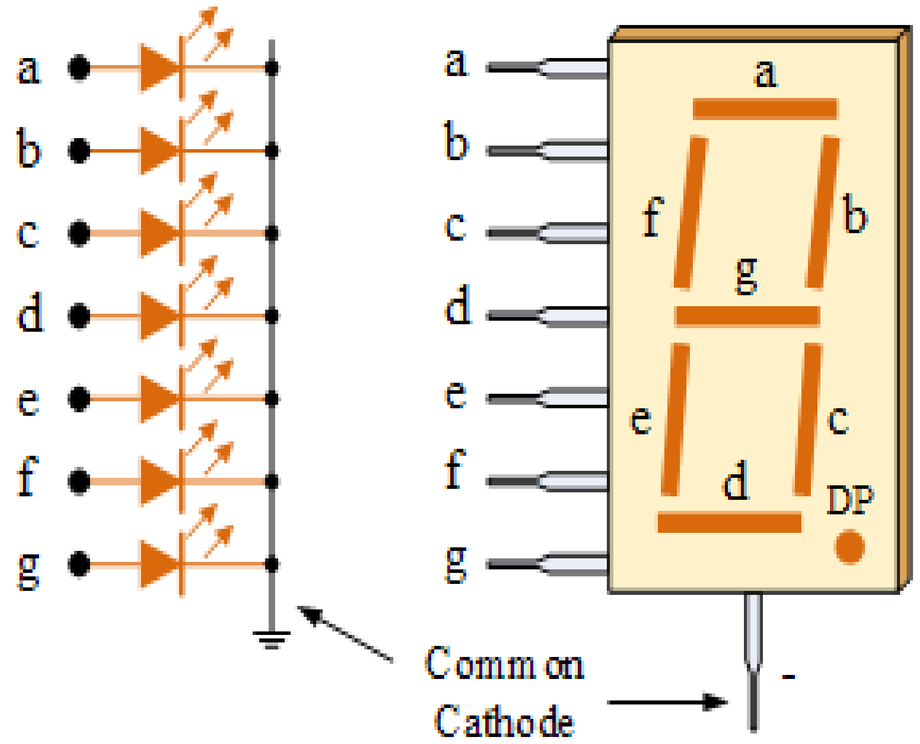
7 Segment Display Interfacing:

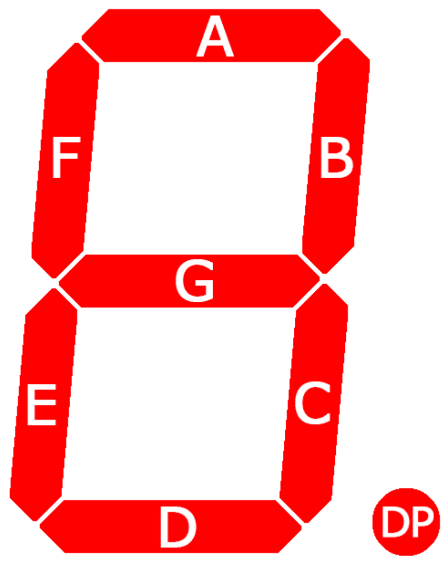


7 Segment Display Interfacing

- This is how to interface a seven segment LED display to an 8051 microcontroller. 7 segment LED display is very popular and it can display digits from 0 to 9 and quite a few characters. Knowledge about how to interface a seven segment display to a micro controller is very essential in designing embedded systems.
- Seven segment displays are of two types, ***common cathode and common anode.***
- In common cathode type , the cathode of all LEDs are tied together to a single terminal which is usually labeled as '**com**'   and the anode of all LEDs are left alone as individual pins labeled as a, b, c, d, e, f, g & h (or dot) .
- In common anode type, the anode of all LEDs are tied together as a single terminal and cathodes are left alone as individual pins.

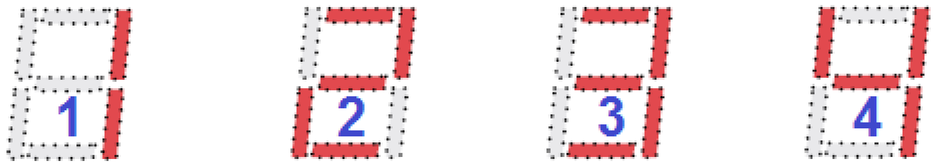
7 Segment Display Interfacing





7 Segment Display Interfacing

Numbers	Common Cathode		Common Anode	
	(DP)GFEDCBA	HEX Code	(DP)GFEDCBA	HEX Code
0	00111111	0x3F	11000000	0xC0
1	00000110	0x06	11111001	0xF9
2	01011011	0x5B	10100100	0xA4
3	01001111	0x4F	10110000	0xB0
4	01100110	0x66	10011001	0x99
5	01101101	0x6D	10010010	0x92
6	011111101	0x7D	10000010	0x82
7	00000111	0x07	11111000	0xF8
8	01111111	0x7F	10000000	0x80
9	01101111	0x6F	10010000	0x90



7 Segment Display Interfacing

/*Program to interface seven segment display unit.*/

```
#include <REG51.H>
#define LEDPORT P0
#define ZERO 0x3f
#define ONE 0x06
#define TWO 0x5b
#define THREE 0x4f
#define FOUR 0x66
#define FIVE 0x6d
#define SIX 0x7d
#define SEVEN 0x07
#define EIGHT 0x7f
#define NINE 0x6f
#define TEN 0x77
```

7 Segment Display Interfacing

```
#define ELEVEN 0x7c
#define TWELVE 0x39
#define THIRTEEN 0x5e
#define FOURTEEN 0x79
#define FIFTEEN 0x71
void Delay(void);
void main (void)
{
while(1)
{
LEDPORT = ZERO;
Delay();
```

7 Segment Display Interfacing

```
LEDPORT = ONE;
Delay();
LEDPORT = TWO;
Delay();
LEDPORT = THREE;
Delay();
LEDPORT = FOUR;
Delay();
LEDPORT = FIVE;
Delay();
LEDPORT = SIX;
Delay();
LEDPORT = SEVEN;
Delay();
```

7 Segment Display Interfacing

```
LEDPORT = EIGHT;
Delay();
LEDPORT = NINE;
Delay();
LEDPORT = TEN;
Delay();
LEDPORT = ELEVEN;
Delay();
LEDPORT = TWELVE;
Delay();
LEDPORT = THIRTEEN;
Delay();
```

7 Segment Display Interfacing

```
LEDPORT = FOURTEEN;
Delay();
LEDPORT = FIFTEEN;
Delay();
}
}
void Delay(void)
{
    int j; int i;
    for(i=0;i<30;i++)
    {
    for(j=0;j<10000;j++)
    {
    }
    }
}
```

LCD Display Interfacing

LCD Display Interfacing:

- In this, we will have brief discussion on how to interface 16×2 LCD module to P89V51RD2, which is an 8051 family microcontroller.

- We use LCD display for the displaying messages in a more interactive way to operate the system or displaying error messages etc.
- Interfacing 16×2 LCD with 8051 microcontroller is very easy if you understanding the working of LCD.
- 16×2 Liquid Crystal Display which will display the 32 characters at a time in two rows (16 characters in one row). Each character in the display is of size 5×7 pixel matrix.

LCD Display Interfacing

There are 16 pins in the LCD module, the pin configuration us given below

PIN NO	NAME	FUNCTION
1	**VSS**	Ground pin
2	**VCC**	Power supply pin of 5V
3	**VEE**	Used for adjusting the contrast commonly attached to the potentiometer.
4	**RS**	RS is the register select pin used to write display data to the LCD (characters), this pin has to be high when writing the data to the LCD. During the initializing sequence and other commands this pin should low.
5	**R/W**	Reading and writing data to the LCD for reading the data R/W pin should be high (R/W=1) to write the data to LCD R/W pin should be low (R/W=0)
6	**E**	Enable pin is for starting or enabling the module. A high to low pulse of about 450ns pulse is given to this pin.

LCD Display Interfacing

7	**DB0**	DB0-DB7 Data pins for giving data(normal data like numbers characters or command data) which is meant to be displayed
8	**DB1**	DB0-DB7 Data pins for giving data
9	**DB2**	DB0-DB7 Data pins for giving data
10	**DB3**	DB0-DB7 Data pins for giving data
11	**DB4**	DB0-DB7 Data pins for giving data
12	**DB5**	DB0-DB7 Data pins for giving data
13	**DB6**	DB0-DB7 Data pins for giving data
14	**DB7**	DB0-DB7 Data pins for giving data
15	**LED+**	Back light of the LCD which should be connected to Vcc
16	**LED-**	Back light of LCD which should be connected to ground.

LCD Display Interfacing

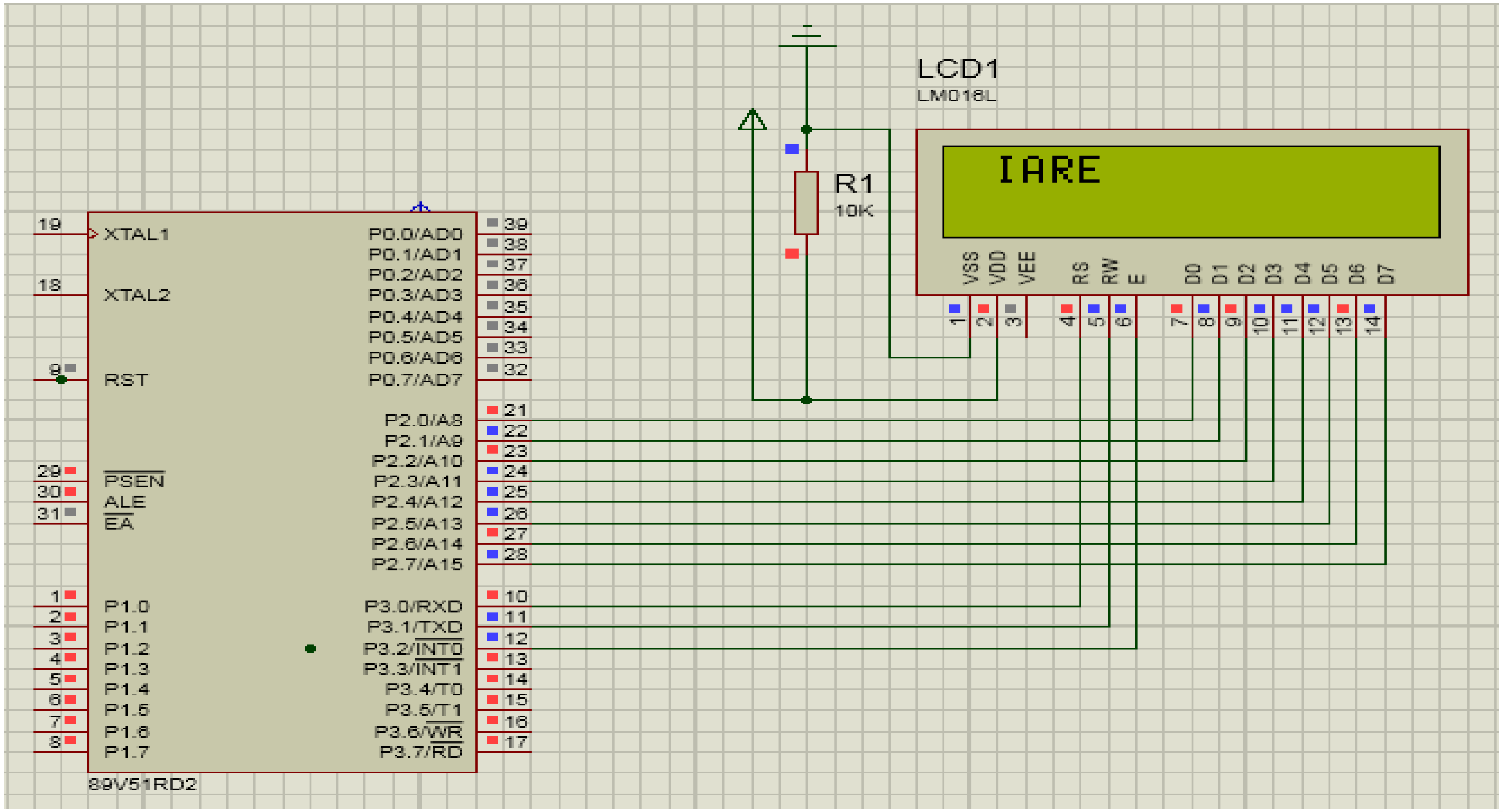

LCD1
LM016L
IARE
R1
10K
VSS VDD VEE RS RW E D0 D1 D2 D3 D4 D5 D6 D7
XTAL1
XTAL2
RST
PSEN
ALE
EA
P1.0 P1.1 P1.2 P1.3 P1.4 P1.5 P1.6 P1.7
P0.0/AD0 P0.1/AD1 P0.2/AD2 P0.3/AD3 P0.4/AD4 P0.5/AD5 P0.6/AD6 P0.7/AD7
P2.0/A8 P2.1/A9 P2.2/A10 P2.3/A11 P2.4/A12 P2.5/A13 P2.6/A14 P2.7/A15
P3.0/RXD P3.1/TXD P3.2/INT0 P3.3/INT1 P3.4/T0 P3.5/T1 P3.6/WR P3.7/RD
89V51RD2

Figure: LCD Display Interfacing

LCD Display Interfacing

Follow these simple steps for displaying a character or data

E=1; enable pin should be high

RS=1; Register select should be high

R/W=0; Read/Write pin should be low.

To send a command to the LCD just follows these steps:

E=1; enable pin should be high

RS=0; Register select should be low

R/W=0; Read/Write pin should be low.

LCD Display Interfacing

No.	Instruction	Hex	Decimal
1	Entry Mode	0x06	6
2	Display off Cursor off (clearing display without clearing DDRAM content)	0x08	8
3	Display on Cursor on	0x0E	14
4	Display on Cursor off	0x0C	12
5	Display on Cursor blinking	0x0F	15
6	Shift entire display left	0x18	24
7	Shift entire display right	0x1C	30
8	Move cursor left by one character	0x10	16
9	Move cursor right by one character	0x14	20
10	Clear Display (also clear DDRAM content)	0x01	1

LCD Commands

LCD Display Interfacing

```
#include<reg51.h>
sbit rs=P3^0;
sbit rw=P3^1;
sbit en=P3^2;
void lcdcmd(unsigned char);
void lcddat (unsigned char);
void delay();
void main()
{
P2=0x00;
while(1)
{
lcdcmd(0x38);                  // Use two lines and 5x7 matrix
delay();
```

LCD Display Interfacing

```
lcdcmd(0x01);      //Clearing the screen
delay();
lcdcmd(0x10);
delay();
lcdcmd(0x0c);
delay();
lcdcmd(0x81);
delay();
```

```
lcddat('I');
delay();
lcddat('A');
delay();
lcddat('R');
delay();
lcddat('E');
delay();
}
}
```

LCD Display Interfacing

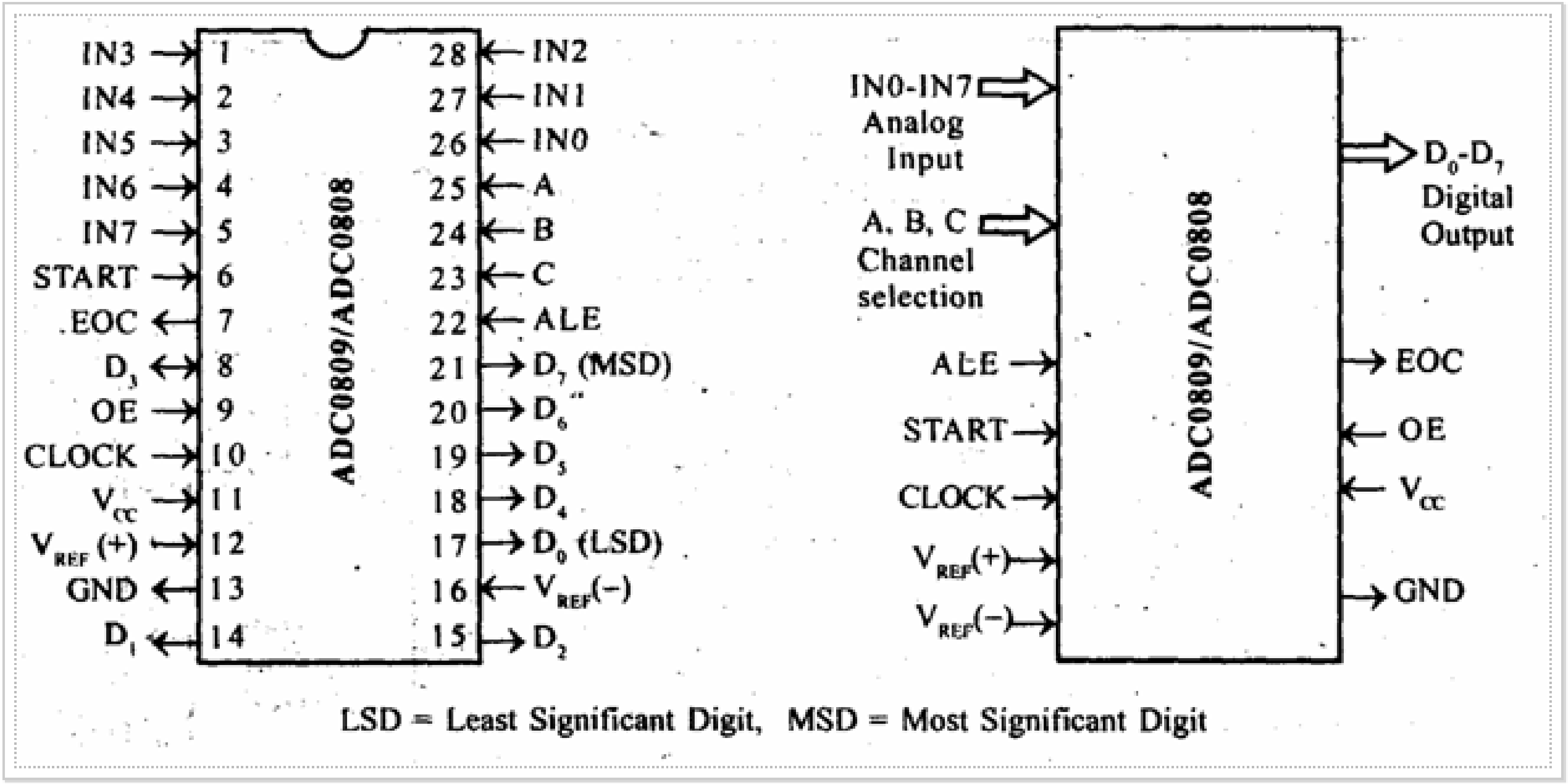
```
void lcdcmd(unsigned char val)
{
P2=val;
rs=0;
rw=0;
en=1;
delay();
en=0;
}
```

```
void lcddat(unsigned char val)
{
P2=val;
rs=1;
rw=0;
en=1;
delay();
en=0;
}
void delay()
{
unsigned int i;
for(i=0;i<1000;i++);
}
```

ADC Interfacing with 8051 microcontroller

ADC Interfacing with 8051 microcontroller:

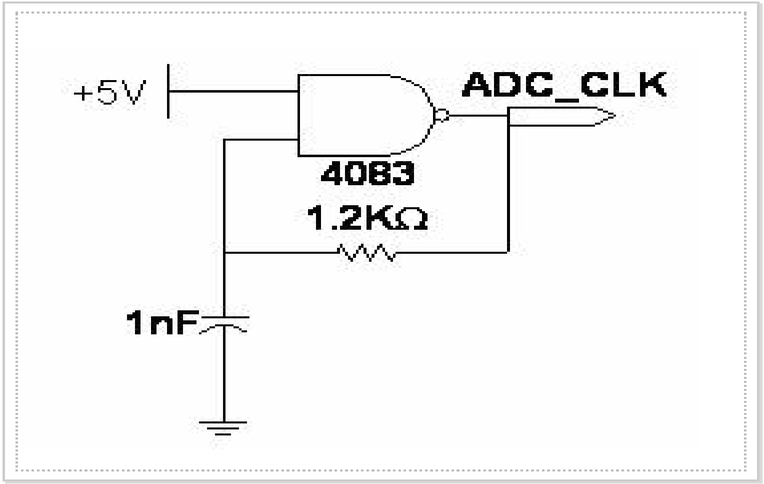
- ADC0808/ADC0809 is an 8 channel 8-bit analog to digital converter. Unlike ADC0804 which has one Analog channel, this ADC has 8 multiplexed analog input channels.



LSD = Least Significant Digit, MSD = Most Significant Digit

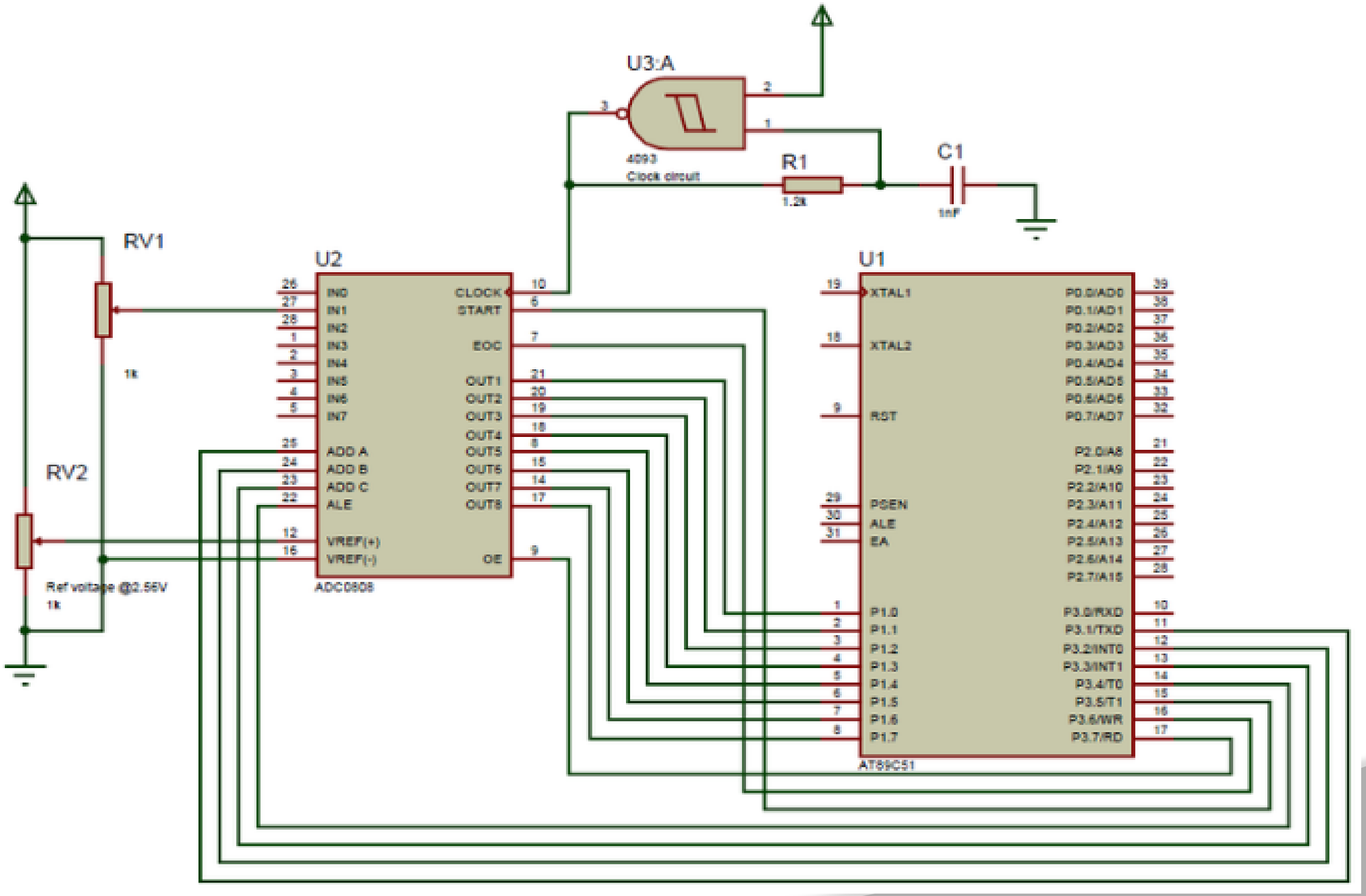
ADC Interfacing with 8051 microcontroller

- The ADC 0808 is an 8-bit A-to-D converter, having data lines D0-D7. It works on the principle of successive approximation. It has a total of eight analogue input channels, out of which any one can be selected using address lines A, B and C. Here, in this case, input channel IN1 is selected by A=1, B & C=0 address lines.
- The ADC 0808 IC requires clock signal of typically 550 kHz and conversion delay 100 µs.



ADC0808 Clock input

ADC Interfacing with 8051 microcontroller

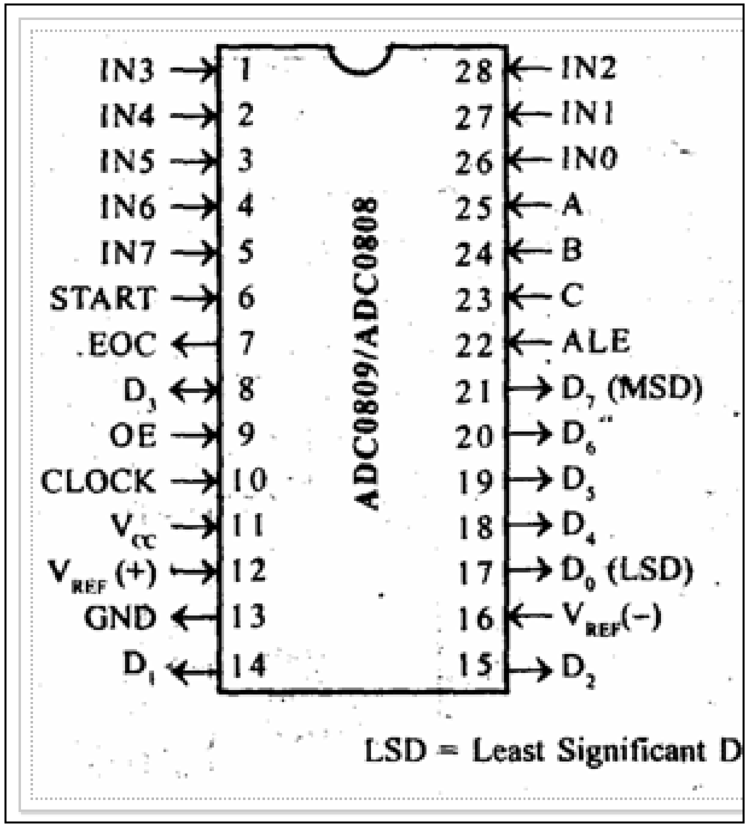


ADC Interfacing with 8051 microcontroller

Program:

```
#include <reg51.h>


#define ALE P3_4
#define OE P3_7
#define START P3_5
#define EOC P3_6
#define SEL_A P3_1
#define SEL_B P3_2
#define SEL_C P3_3
#define ADC_DATA P1
```



ADC Interfacing with 8051 microcontroller

```
void main()
{
unsigned char adc_data;
 /* Data port to input */
ADC_DATA = 0xFF;

EOC = 1;              /* EOC as input */
ALE = OE = START = 0;

while (1)
{
/* Select channel 1 */
SEL_A = 1;            /* LSB */
SEL_B = 0;
SEL_C = 0;            /* MSB */
```

SEL_C	SEL_B	SEL_A	Channel Number
0	0	0	CH0
0	0	1	CH1
0	1	0	CH2
0	1	1	CH3
1	0	0	CH4
1	0	1	CH5
1	1	0	CH6
1	1	1	CH7

ADC Interfacing with 8051 microcontroller

```
/* Latch channel select/address */
ALE = 1;

/* Start conversion */
START = 1;

ALE = 0;
START = 0;
/* Wait for end of conversion */
while (EOC == 1);
while (EOC == 0);
```

```
/* Assert Read signal */
OE = 1;
/* Read Data */
adc_data = ADC_DATA;
OE = 0;

/* Now adc data is stored
*/
/* start over for next
conversion */
}
}
```

DAC Interfacing with 8051 microcontroller

DAC INTERFACING with 8051

The digital-to-analog converter (DAC) is a device widely used to convert digital pulses to analog signals.

In this section we discuss the basics of interfacing a DAC to the 8051.

Recall from your digital electronics book the two methods of creating a DAC:

1. Binary weighted.
2. R/2R ladder.

DAC Interfacing with 8051 microcontroller

- The vast majority of integrated circuit DACs, including the MC1408 (DAC0808) used in this section use the R/2R method since it can achieve a much higher degree of precision.

- The first criterion for judging a DAC is its resolution, which is a function of the number of binary inputs. The common ones are 8, 10, and 12 bits.

- The number of data bit inputs decides the resolution of the DAC since the number of analog output levels is equal to 2^n, where *n* is the number of data bit inputs.

- Therefore, an 8-input DAC such as the DAC0808 provides 256 discrete voltage (or current) levels of output. Similarly, the 12-bit DAC provides 4096 discrete voltage levels. There are also 16-bit DACs, but they are more expensive.

DAC Interfacing with 8051 microcontroller

MC1408 DAC (or DAC0808):

In the MC1408 (or DAC0808), the digital inputs are converted to current (I_{out}), and by connecting a resistor to the I_{out} pin, we convert the result to voltage.

The total current provided by the I_{out} pin is a function of the binary numbers at the D0 – D7 inputs of the DAC0808 and the reference current (I_{ref}), and is as follows:

$$\boxed{I_{out} = I_{ref}\left(\frac{D7}{2} + \frac{D6}{4} + \frac{D5}{8} + \frac{D4}{16} + \frac{D3}{32} + \frac{D2}{64} + \frac{D1}{128} + \frac{D0}{256}\right.}$$

Where D0 is the LSB, D7 is the MSB for the inputs, and I_{ref} is the input current that must be applied to pin 14. The I_{ref} current is generally set to 2.0 mA. Figure shows the generation of current reference (setting I_{ref} = 2 mA) by using the standard 5-V power supply and IK and 1.5K-ohm standard resistors. Some DACs also use the zener diode (LM336), which overcomes any fluctuation associated.

DAC Interfacing with 8051 microcontroller

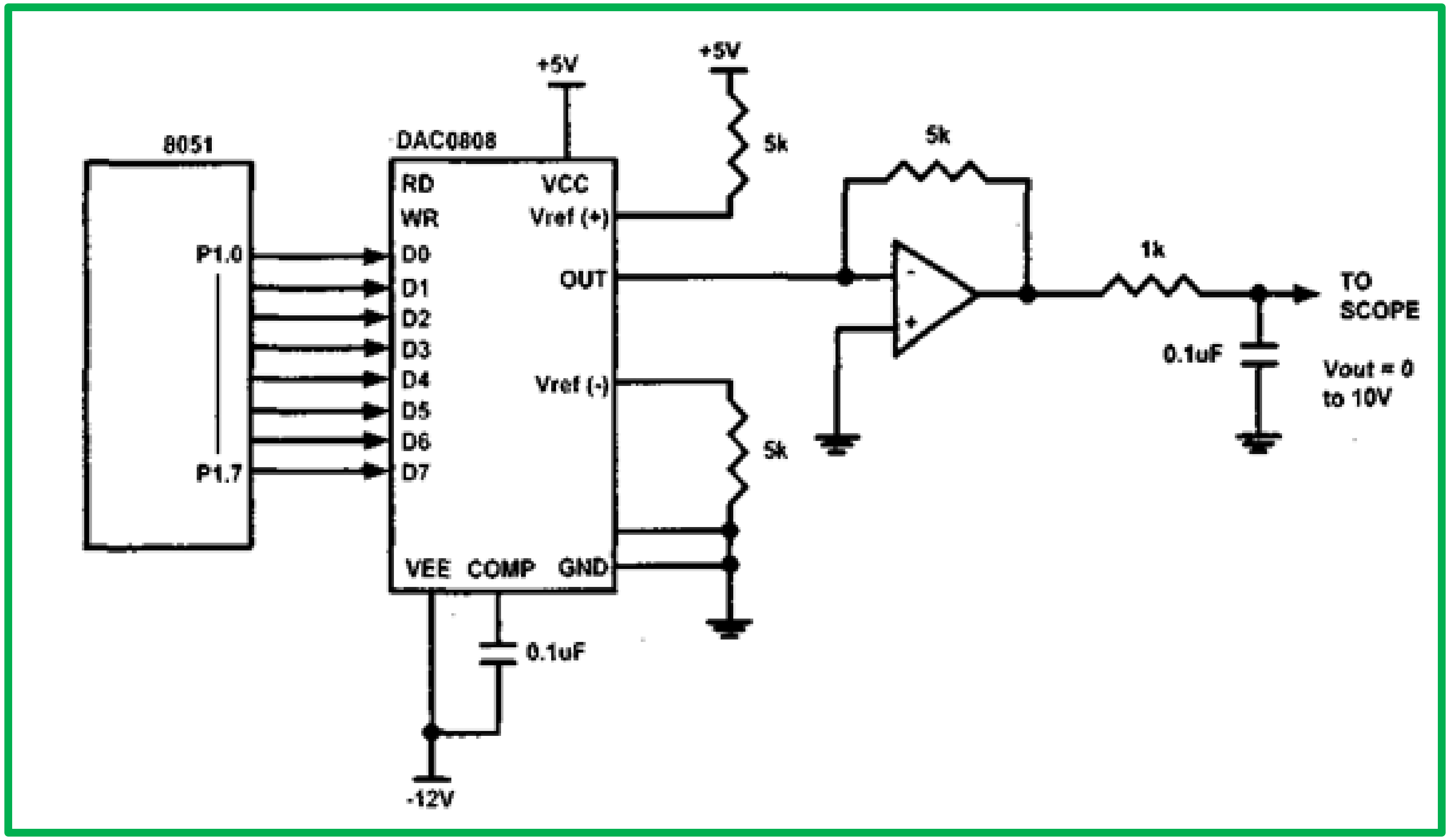


Figure: 8051 Connection to DAC0808

DAC Interfacing with 8051 microcontroller

Generating a sine wave:

- To generate a sine wave, we first need a table whose values represent the magnitude of the sine of angles between 0 and 360 degrees. The values for the sine function vary from -1.0 to +1.0 for 0- to 360-degree angles.
- Therefore, the table values are integer numbers representing the voltage magnitude for the sine of theta. This method ensures that only integer numbers are output to the DAC by the 8051 microcontroller.
- Table shows the angles, the sine values, the voltage magnitudes, and the integer values representing the voltage magnitude for each angle (with 30-degree increments).
- To generate Table, we assumed the full-scale voltage of 10 V for DAC output. Full-scale output of the DAC is achieved when all the data inputs of the DAC are high. Therefore, to achieve the full-scale 10 V output, we use the following equation.

$$V_{out} = 5\text{ V} + (5 \times \sin\theta)$$

DAC Interfacing with 8051 microcontroller

V_{out} of DAC for various angles is calculated and shown in Table.

Angle θ (degrees)	Sin θ	V_{out} (Voltage Magnitude) 5 V + (5 V x sin θ)	Values Sent to DAC (decimal) (Voltage Mag. X 25.6)
0	0	5	128
30	0.5	7.5	192
60	0.866	9.33	238
90	1.0	10	255
120	0.866	9.33	238
150	0.5	7.5	192
180	0	5	128
210	-0.5	2.5	64
240	-0.866	0.669	17
270	-1.0	0	0
300	-0.866	0.669	17
330	-0.5	2.5	64
360	0	5	128

DAC Interfacing with 8051 microcontroller

Program:

```
#include <reg51.h>
sfr DACDATA = Pl;
void main ()
{
unsigned char WAVEVALUE [12]={128,192,238,255, 238,192,128,64, 17,0,17,64} ;
unsigned char x ,
while (1)
     {
          for(x=0;x<12;x++)
          {
               DACDATA = WAVEVALUE[x];
          }
     }
 }
```

DAC Interfacing with 8051 microcontroller

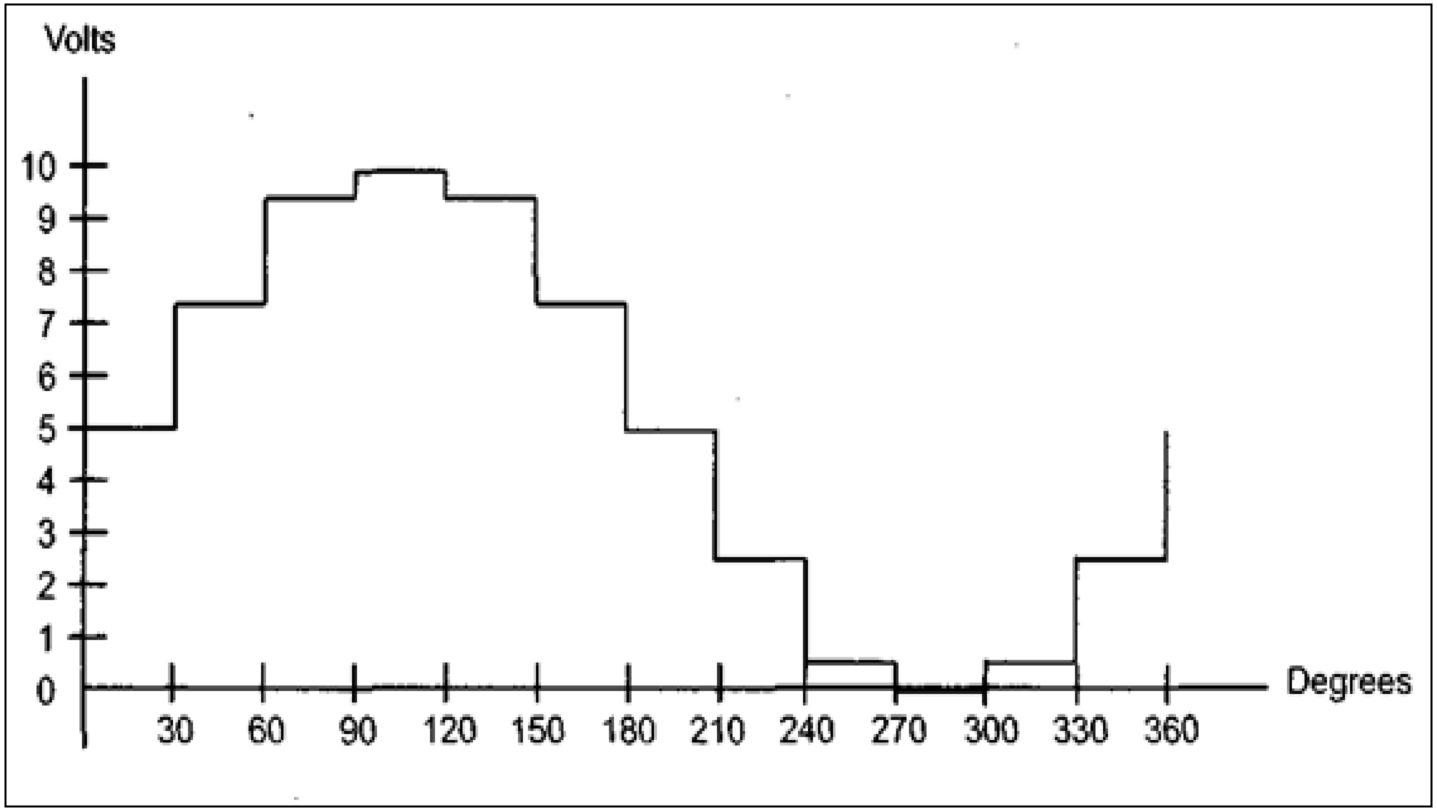


Figure: Angle vs. Voltage Magnitude for Sine Wave

Multiple Interrupts in 8051 microcontroller

Multiple Interrupts

Interrupts vs. polling:

- A single microcontroller can serve several devices. There are two ways to
- do that: interrupts or polling.

- In the ***interrupt method****, whenever any device needs* its service the device notifies the microcontroller by sending It an interrupt signal.

- Upon receiving an interrupt signal, the microcontroller interrupts whatever it is doing and serves the device.

- The program associated with the interrupt is called the ***interrupt service routine (ISR)*** *or* ***interrupt handler.***

Multiple Interrupts in 8051 microcontroller

- ***In polling**, the microcontroller* continuously monitors the status of a given device; when the status condition is met, it performs the service. After that, it moves on to monitor the next device until each one is serviced.

- Although polling can monitor the status of several devices and serve each of them as certain conditions are met, it is not an efficient use of the microcontroller.

- The advantage of interrupts is that the microcontroller can serve many devices (not all at the same time, of course); each device can get the attention of the microcontroller based on the priority assigned to it.

- The polling method cannot assign priority since it checks all devices in a round robin fashion.

Multiple Interrupts in 8051 microcontroller

Six interrupts in the 8051:

In reality, only five interrupts are available to the user in the 8051, but many manufacturers data sheets state that there are six interrupts since they include reset. The six interrupts in the 8051 are allocated as follows.

1. Reset. When the reset pin is activated, the 8051 jumps to address location 0000. This is the power-up reset.

2. Two interrupts are set aside for the timers: one for **Timer 0** and one for **Timer1**. Memory locations 000BH and 001BH in the interrupt vector table belong to Timer 0 and Timer 1, respectively.

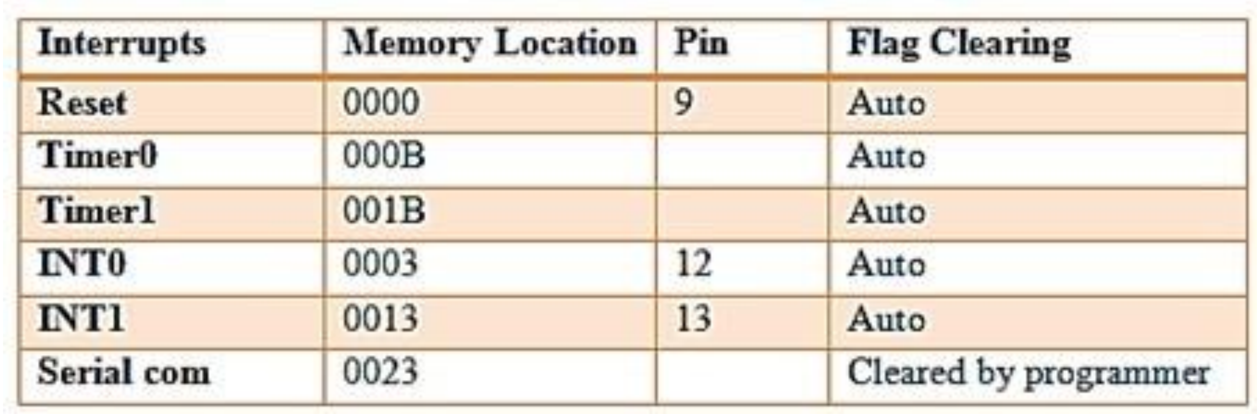
Multiple Interrupts in 8051 microcontroller

Six interrupts in the 8051:

3. Two interrupts are set aside for hardware **external hardware interrupts**, Pin numbers 12 (P3.2) and 13 (P3.3) in port 3 are for the external hardware interrupts **INT 0** and **INT 1**, respectively. These external interrupts are also referred to as EX 1 and EX 2. Memory locations 0003H and 0013H In the interrupt vector table are assigned to INT0 and INT1, respectively.

4. Serial communication has a single interrupt that belongs to both receive and transmit. The interrupt vector table location 0023H belongs to this interrupt.

Multiple Interrupts in 8051 microcontroller



Interrupts	Memory Location	Pin	Flag Clearing
Reset	0000	9	Auto
Timer0	000B		Auto
Timer1	001B		Auto
INT0	0003	12	Auto
INT1	0013	13	Auto
Serial com	0023		Cleared by programmer

Table: Interrupt Vector Table for the 8051

Multiple Interrupts in 8051 microcontroller

Enabling and Disabling an interrupt:

- Upon reset, all interrupts are disabled (masked), meaning that none will be responded to by the microcontroller if they are activated.

- The interrupts must be enabled by software in order for the microcontroller to respond to them. There is a register called IE (interrupt enable) that is responsible for enabling (unmasking) and disabling (masking) the interrupts.

- Figure shows the IE register. Note that IE is a bit-addressable register.

- From figure notice that bit D7 in the IE register is called EA (enable
- all). This must be set to 1 in order for the rest of the register to take effect. D6 is unused. D5 is used by the 8052. The D4 bit is for the serial interrupt, and so on.

Multiple Interrupts in 8051 microcontroller

Steps in enabling an interrupt:

To enable an interrupts, we take the following steps:

1. Bit D7 of the IE register (EA) must be set to high to allow the rest of register to take the effect.

2. If EA =1, interrupts are enabled and will be responded to if their corresponding bits in IE are high. If EA=0, no interrupt will be responded to, even if the associated bit in the IE register is high.

Multiple Interrupts in 8051 microcontroller

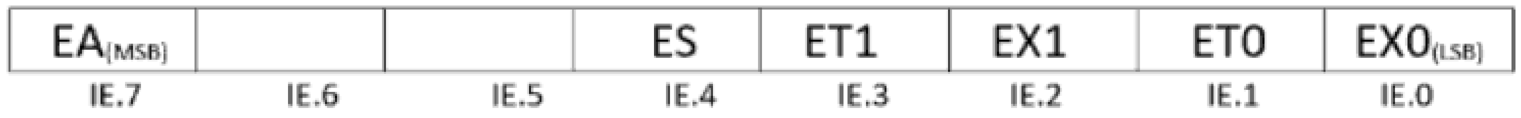
IE (Interrupt Enable) Register:

•This register is responsible for enabling and disabling the interrupt.

•EA register is set to 1 for enabling interrupts and

•EA register is set to 0 for disabling the interrupts.

•Its bit sequence and their meanings are shown in the following figure.



EA(MSB)			ES	ET1	EX1	ET0	EX0(LSB)
IE.7	IE.6	IE.5	IE.4	IE.3	IE.2	IE.1	IE.0

INTERRUPT ENABLE REGISTER AT A8H

Multiple Interrupts in 8051 microcontroller

EA	IE.7	It disables all interrupts. When EA = 0 no interrupt will be acknowledged and When EA = 1 enables the interrupt individually.
-	IE.6	Reserved for future use.
-	IE.5	Reserved for future use.
ES	IE.4	Enables/disables serial port interrupt.
ET1	IE.3	Enables/disables timer1 overflow interrupt.
EX1	IE.2	Enables/disables external interrupt1.
ET0	IE.1	Enables/disables timer0 overflow interrupt.
EX0	IE.0	Enables/disables external interrupt0.

Pointer Aliasing

Pointer Aliasing:

Two pointers are said to *alias when they point to the same address. If you write to one* pointer, it will affect the value you read from the other pointer.

In a function, the compiler often doesn’t know which pointers can alias and which pointers can’t. The compiler must be very pessimistic and assume that any write to a pointer may affect the value read from any other pointer, which can significantly reduce code efficiency.

Unaligned Data And Endianness

Unaligned Data And Endianness :

- **Unaligned data** and **endianness** are two issues that can complicate memory accesses and portability.
- In computing, **endianness** is the ordering or sequencing of bytes of a word of digital data in computer memory storage or during transmission.
- A big-**endian** system stores the most significant byte of a word at the smallest memory address and the least significant byte at the largest memory address .
- A **memory access** is said to be **aligned** when the data being **accessed** is n bytes long and the datum **address** is n-byte **aligned**. ... A **memory** pointer that refers to primitive data that is n bytes long is said to be **aligned** if it is only allowed to contain addresses that are n-byte **aligned**, otherwise it is said to be **unaligned**.

Inline Functions and Inline Assembly

Inline Functions and Inline Assembly:

Generally the **inline** term is used to instruct the compiler to insert the code of a **function** into the code of its caller at the point where the actual call is made. Such **functions** are called "**inline functions**". ... It is just a set of **assembly** instructions written as **inline functions**.

Serial Data Communication in 8051

Serial Data Communication Using Embedded C Interfacing

Serial Data Communication in 8051

- Computers transfer data in two ways: **parallel** and **serial**,

- In **parallel data transfers**, often 8 or more lines (wire conductors) are used to transfer data to a device that is only a few feet away.

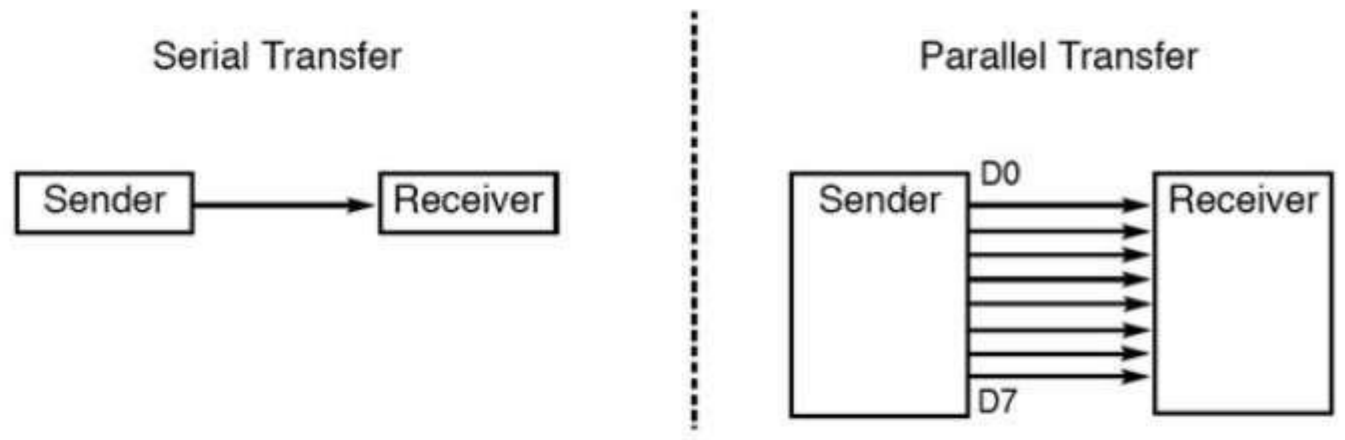
  Examples: Printers and Hard disks

- To transfer to a device located many meters away, the serial method is used.
- In **serial communication**, the data is sent one bit at a time, in contrast to parallel communication, in which the data is sent a byte or more at a time.

- The 8051 has serial communication capability built into it, thereby making possible fast data transfer using only a few wires.

Serial Data Communication in 8051

Serial v/s Parallel Communication:



Serial versus Parallel Data Transfer

Serial Data Communication in 8051

Parallel Communication	Serial Communication
Often 8 or more lines (wire conductors) are used to transfer data. Multiple bits are transferred at a time.	The data is sent one bit at a time on a single line (wire)
Preferred for short-distance communication	Preferred over long-distance communication
Costly as more resources are required	Comparatively cheaper
Speed of data transfer is high	Slow
Example: SPI, I2C, UART	**Example:** PCI

Serial Data Communication in 8051

Basics of Serial Communication:

- Serial communication uses single data line making it much cheaper.
- Enables two computers in different cities to communicate over the telephone.
- Byte of data must be converted to serial bits using a parallel-in-serial- out shift register and transmitted over a single data line
- At the receiving end there must be a serial-in-parallel-out shift register.
- If transferred on the telephone line, it must be converted to audio tones by modem for short distance.

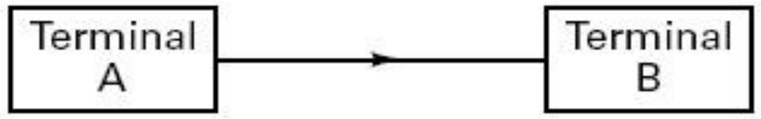
Serial Data Communication in 8051

Modes of Serial Communication:

- In **simplex transmissions**, the computer can only send data. There is only one wire.
- If the data can be transmitted and received, then it is a **duplex transmission**
- Duplex transmissions can be half or full duplex depending on whether or not the data transfer can be simultaneous.
- If the communication is only one way at a time, it is **half duplex**
- If both sides can communicate at the same time, it is **full duplex**
  - ✓ Full duplex requires two wire conductors for the data lines (in addition to the signal ground)

Serial Data Communication in 8051

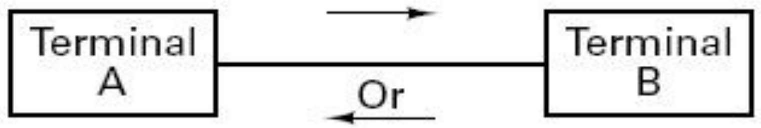
Modes of Serial Communication



Transmission in only one direction

(a)

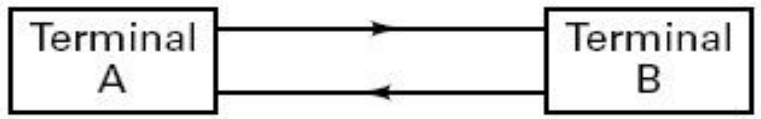
Ex: Radio and Television transmission



Transmission in either direction, but not simultaneously

(b)

Ex: Walkie-talkie



Transmission in both directions simultaneously

(c)

Ex: Telephone

Serial Data Communication in 8051

Basics of Serial Communication

- Serial Communication can be
  - ✓ Asynchronous
  - ✓ Synchronous

Synchronous Communication:

- Synchronous methods transfer a block of data (characters) at a time
- The events are referenced to a clock
- **Example:** SPI bus, I2C bus

Asynchronous Communication:

- Asynchronous methods transfer a single byte at a time
- There is no clock. The bytes are separated by start and stop bits.
- **Example:** UART

Serial Data Communication in 8051

Basics of Serial Communication

- To support serial communication, special interfaces are built in the microcontroller.
- The microcontrollers use special IC chips called UART (universal asynchronous receiver-transmitter) and USART (universal synchronous asynchronous receiver-transmitter)
- 8051 chip has a built-in UART

Serial Data Communication in 8051

Data Framing in Asynchronous Serial Communication

- Data is transmitted in 0s and 1s.
- To have a sense of synchronization between transmitter and receiver and to make sense of the data, transmitter and receiver agree on a set of rules i.e protocol, which describes
  - ✓ how the data is packed
  - ✓ how many bits constitute a character
  - ✓ when the data begins and ends

Serial Data Communication in 8051

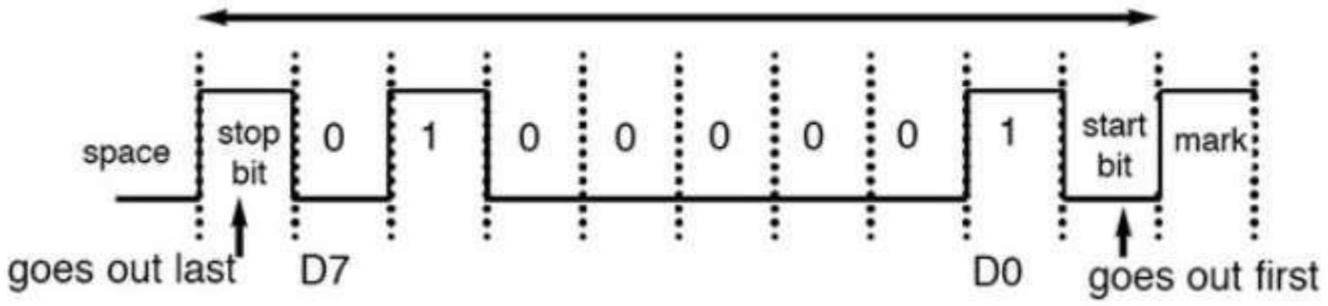
Data Framing in Asynchronous Serial Communication

Start and stop bits

- Each character is placed between start and stop bits. This is called framing.
- Start bit is always one bit, stop bit can be one, two or one and half bits.
- In 8051 serial port, when there is no transmission, the TxD line is held high. This is called mark.
- Start bit is always a 0 (low), stop bit(s) is 1 (high).
- LSB is sent out first.

Serial Data Communication in 8051

Data Framing in Asynchronous Serial Communication



- The transmission begins with a start bit, followed by the LSB(D_0), then the rest of the bits until MSB (D_7), and finally, the one stop bit indicating the end of the character
- When there is no transfer, the signal is 1 (high), which is referred to as *mark*

Serial Data Communication in 8051

Data Transfer Rate in Asynchronous Serial Communication

- The rate of data transfer in serial data communication is stated in bps (bits per second)
- Another widely used terminology for bps is baud rate
  - ✓ It is modem terminology and is defined as the number of signal changes per second
  - ✓ In modems, there are occasions when a single change of signal transfers several bits of data
- As far as the conductor wire is concerned, the baud rate and bps are the same, and we use the terms interchangeably
- The data transfer rate of given computer system depends on communication ports incorporated into that system
  - ✓ IBM PC/XT could transfer data at the rate of 100 to 9600 bps

Serial Data Communication in 8051

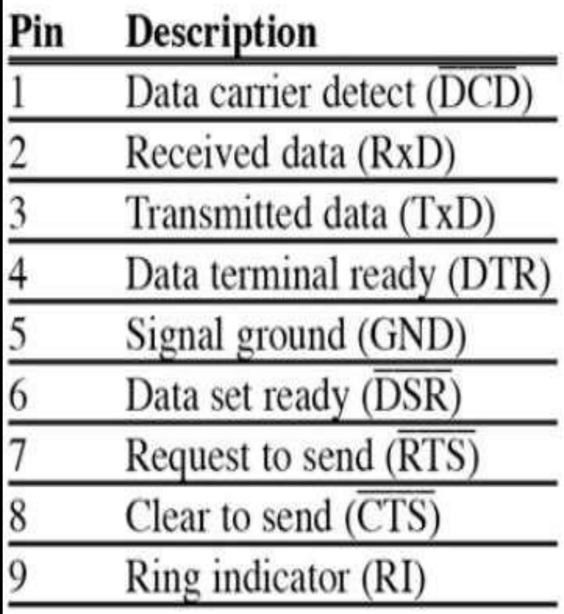
RS232 Standards

- An interfacing standard RS232 was set by the Electronics Industries Association (EIA) in 1960.
- In RS232, a 1 is represented by -3 to -25 V, while a 0 bit is +3 to +25 V, making -3 to +3 undefined.
- The standard was set long before the advent of the TTL logic family, its input and output voltage levels are not TTL compatible.
- A microcontroller system must use voltage converters such as **MAX232** to convert the TTL logic levels to the RS232 voltage levels, and vice versa.
- MAX232 IC chips are commonly referred to as line drivers.

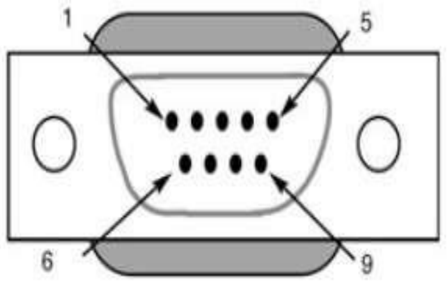
Serial Data Communication in 8051

DB9 pin connections:

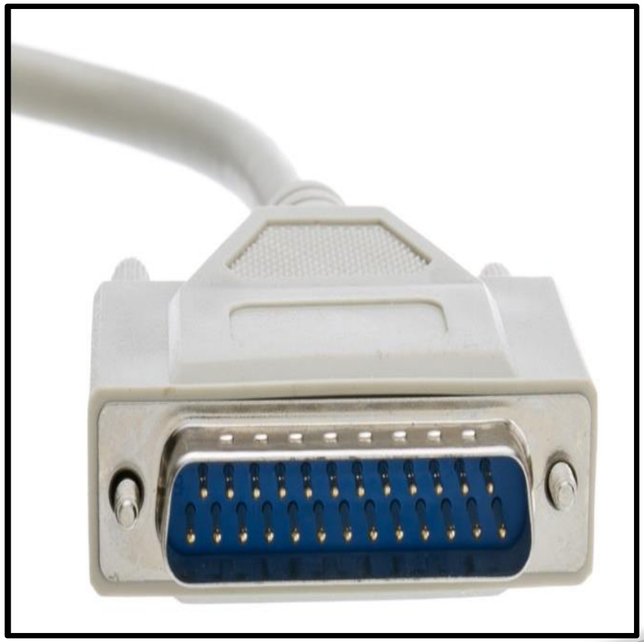
- RS232 supports both DB25 and DB 9 pin connector
- DB-9 Pin Connector



Pin	Description
1	Data carrier detect ($\overline{\text{DCD}}$)
2	Received data (RxD)
3	Transmitted data (TxD)
4	Data terminal ready (DTR)
5	Signal ground (GND)
6	Data set ready ($\overline{\text{DSR}}$)
7	Request to send ($\overline{\text{RTS}}$)
8	Clear to send ($\overline{\text{CTS}}$)
9	Ring indicator (RI)



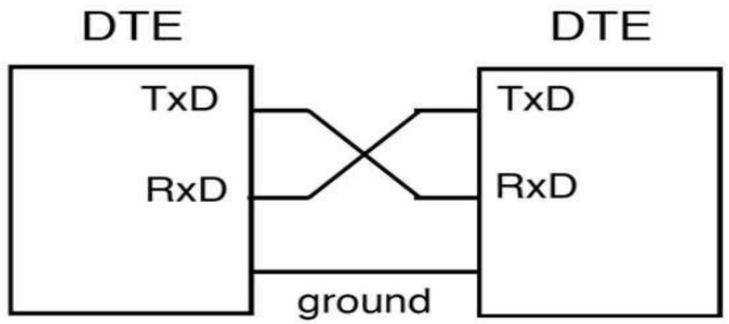
DB-9 9-Pin Connector



Serial Data Communication in 8051

Handshaking in RS232:

- Current terminology classifies data communication equipment as
    - ✓ **DTE** (data terminal equipment) refers to terminal and computers that send and receive data
    - ✓ **DCE** (data communication equipment) refers to communication equipment, such as modems
- The simplest connection between a PC and microcontroller requires a minimum of three pins, TxD, RxD, and ground



Null Modem Connection

Serial Data Communication in 8051

Handshaking signals in Rs232:

DTR (data terminal ready)

When terminal is turned on, it sends out signal DTR to indicate that it is ready for communication

DSR (data set ready)

When DCE is turned on and has gone through the self-test, it assert DSR to indicate that it is ready to communicate

RTS (request to send)

When the DTE device has byte to transmit, it assert RTS to signal the modem that it has a byte of data to transmit

CTS (clear to send)

When the modem has room for storing the data it is to receive, it sends out signal CTS to DTE to indicate that it can receive the data now

Serial Data Communication in 8051

Handshaking signals in Rs232

DCD (data carrier detect)

The modem asserts signal DCD to inform the DTE that a valid carrier has been detected and that contact between it and the other modem is established

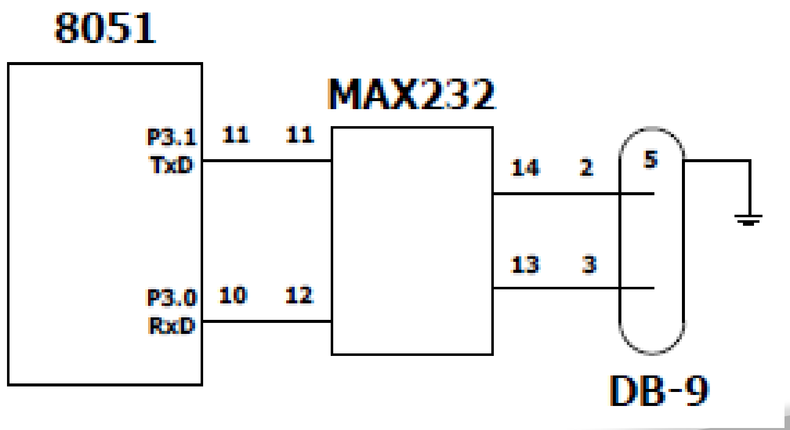
RI (Ring Indicator)

- This is an input for DTE devices and an output for DCE devices. This signals the DTE device that there is an incoming call. This signal is maintained "Off" at all times except when the DCE receives a ringing signal.
- An output from the modem and an input to a PC indicates that there is an incoming call.

Serial Data Communication in 8051

TxD and RxD in 8051

- 8051 has two pins that are used specifically for transferring and receiving data serially.
  - ✓ These two pins are called TxD and RxD and are part of the port 3 group (P3.0 and P3.1)
  - ✓ These pins are TTL compatible; therefore, they require a line driver to make them RS232 compatible.
- We need a line driver (voltage converter) to convert the R232's signals to TTL voltage levels that will be acceptable to 8051's TxD and RxD pins



Serial Data Communication in 8051

PC Baud rate

- PC/compatible COM ports PC/compatible computers (Pentium) microprocessors normally have two COM ports
- Both ports have RS232-type connectors
- COM ports are designated as COM 1 and COM 2 (replaced by USB ports)
- To allow data transfer between the PC and an 8051 system without any error, we must make sure that the baud rate of 8051 system matches the baud rate of the PC's COM port
- Baud rate supported by IBM PC: 19200, 9600, 4800, 2400, 1200, 600, 300, 150 and 110

Serial Data Communication in 8051

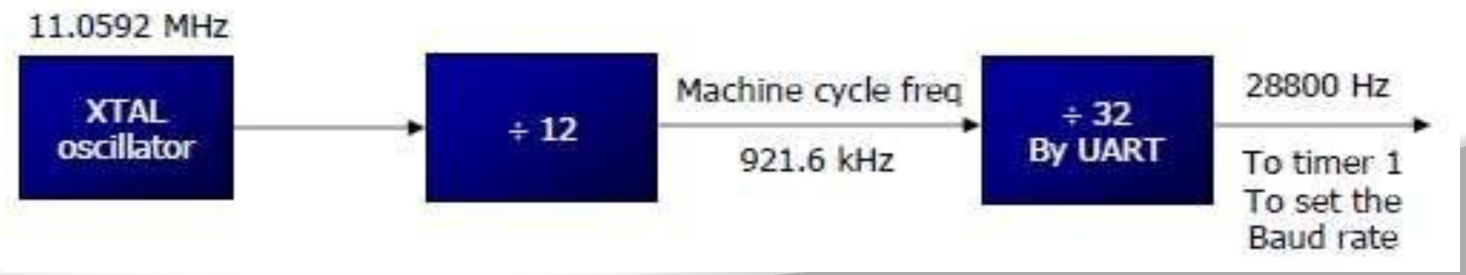
Setting Baud rate in 8051

- Baud rate in the 8051 is programmable.

Relationship between the crystal frequency and the baud rate in the 8051

✓ 8051 divides the crystal frequency by 12 to get the machine cycle frequency

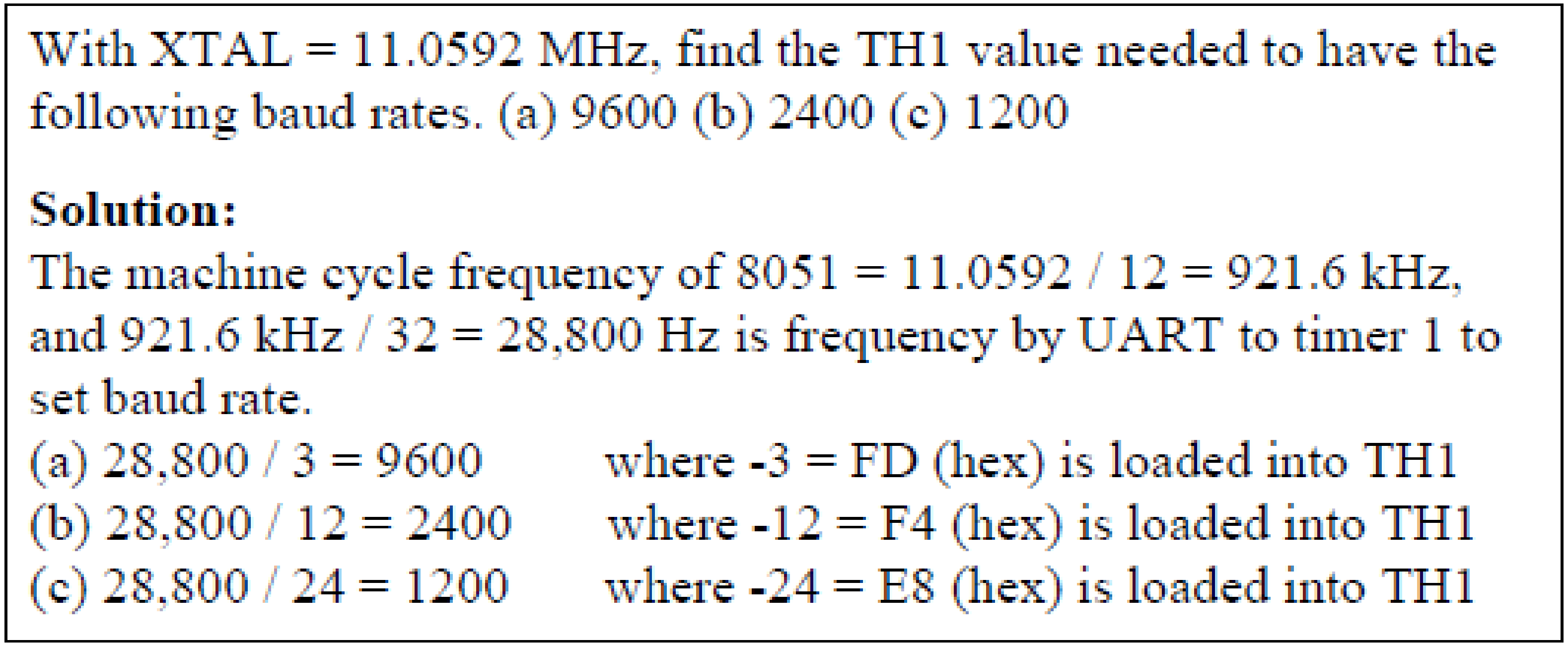
✓ XTAL = 11.0592 MHz, the machine cycle frequency is 921.6 kHz 8051's UART divides the machine cycle frequency of 921.6 kHz by 32 once more before it is used by Timer 1 to set the baud rate 921.6 kHz divided by 32 gives 28,800 Hz



Serial Data Communication in 8051

Setting Baud rate in 8051

- Timer 1 must be programmed in mode 2, that is 8-bit, auto-reload



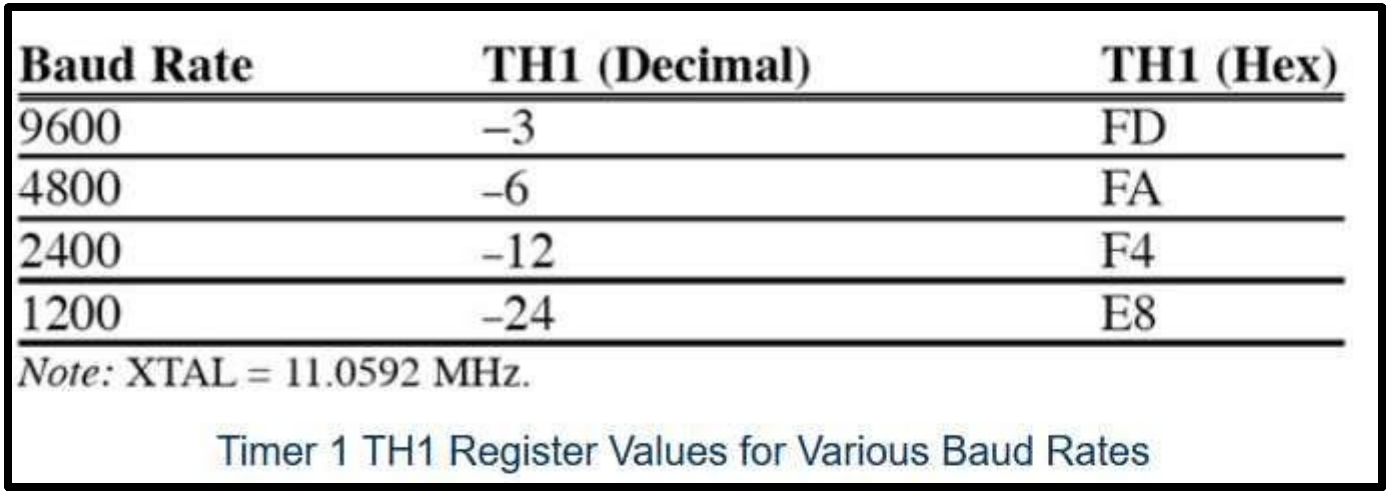
With XTAL = 11.0592 MHz, find the TH1 value needed to have the following baud rates. (a) 9600 (b) 2400 (c) 1200

Solution:
The machine cycle frequency of 8051 = 11.0592 / 12 = 921.6 kHz, and 921.6 kHz / 32 = 28,800 Hz is frequency by UART to timer 1 to set baud rate.
(a) 28,800 / 3 = 9600 where -3 = FD (hex) is loaded into TH1
(b) 28,800 / 12 = 2400 where -12 = F4 (hex) is loaded into TH1
(c) 28,800 / 24 = 1200 where -24 = E8 (hex) is loaded into TH1

Serial Data Communication in 8051

Setting Baud rate in 8051

- Timer 1 must be programmed in mode 2, that is 8-bit, auto-reload



Baud Rate	TH1 (Decimal)	TH1 (Hex)
9600	–3	FD
4800	–6	FA
2400	–12	F4
1200	–24	E8

Note: XTAL = 11.0592 MHz.

Timer 1 TH1 Register Values for Various Baud Rates

Serial Data Communication in 8051

SBUF Register

- A byte of data to be transferred via the TxD line must be placed in the SBUF register
- SBUF holds the byte of data when it is received by the RxD line
- SBUF can be accessed like any other register
  - MOV SBUF, #'D' ; load SBUF=44H, ASCII for 'D‘
  - MOV SBUF, A ; copy accumulator into SBUF
  - MOV A, SBUF ; copy SBUF into accumulator

✓ When a byte is written in SBUF, it is framed by 8051 with the start and stop bits and transferred serially via the TxD pin

✓ When the bits are received serially via RxD, it is deframed by 8051 by eliminating the stop and start bits, making a byte out of the data received, and then placing it in the SBUF

✓ Framing need not be done by programmer explicitly

Serial Data Communication in 8051

SBUF Register:

- The special function register SBUF is physically two registers.
  - ✓ One is, write-only and is used to hold data to be transmitted out of the 8051 via TXD.
  - ✓ The other is, read-only and holds the received data from external sources via RXD.
- Both mutually exclusive registers have the same address 099H.
- SBUF is not bit addressable.

Serial Data Communication in 8051

SCON Register

SCON is an 8-bit register used to program the start bit, stop bit, and data bits of data framing, among other Things

SM0	SM1	SM2	REN	TB8	RB8	TI	RI

- SM0 : Serial Mode Specifier
- SM1 : Serial Mode Speceifier
- SM2 : Used for multiprocessor Communication
- REN : Set/Cleared by Software to enable/disable reception
- TB8 : not used in Mode 1
- RB8 : Not used in Mode 1
- TI : Transmit interrupt flag. Set by HW at the begin of the stop bit mode 1. And cleared by SW
- RI : Receive interrupt flag. Set by HW at the begin of the stop bit mode 1. And cleared by SW

Serial Data Communication in 8051

SCON Register:

SM0, SM1 determine the framing of data by specifying the number of bits per character, and the start and stop bits

SM0	SM1	Serial Mode	Description	Baud Rate
0	0	Mode 0	Shift register	(fosc/12)
0	1	Mode 1	9 bit UART	variable
1	0	Mode 2	9 bit UART	(fosc/64) or (fosc/32)
1	1	Mode 3	9 bit UART	variable

Serial Data Communication in 8051

Serial Data Transmission Modes:

Mode 0

- In this mode, the serial port works like a shift register and the data transmission works synchronously with a clock frequency of $f_{osc}/12$.
- Serial data is received and transmitted through RXD.
- 8 bits are transmitted/ received at a time.
- Pin TXD outputs the shift clock pulses of frequency $f_{osc}/12$, which is connected to the external circuitry for synchronization.
- The shift frequency or baud rate is always 1/12 of the oscillator frequency.

Serial Data Communication in 8051

Mode 1

- In mode-1, the serial port functions as a standard Universal Asynchronous Receiver Transmitter (UART) mode.
- 10 bits are transmitted through TXD or received through RXD.
- The 10 bits consist of one start bit (which is usually '0'), 8 data bits (LSB is sent first/received first), and a stop bit (which is usually '1').
- Once received, the stop bit goes into RB8 in the special function register SCON. The baud rate is variable.

Serial Data Communication in 8051

Mode 2

- In this mode 11 bits are transmitted through TXD or received through RXD.
- The various bits are as follows: a start bit (usually '0'), 8 data bits (LSB first), a programmable 9 th (TB8 or RB8)bit and a stop bit (usually '1').
- While transmitting, the 9 th data bit (TB8 in SCON) can be assigned the value '0' or '1'.
  - ✓ For example, if the information of parity is to be transmitted, the parity bit (P) in PSW could be moved into TB8. On reception of the data, the bit goes into RB8 in 'SCON', while the stop bit is ignored.
- The baud rate is programmable to either 1/32 or 1/64 of the oscillator frequency.

$$f_{baud} = (2^{SMOD}/64) f_{osc}.$$

Serial Data Communication in 8051

Mode 3

- In this mode 11 bits are transmitted through TXD or received through RXD.
- The various bits are: a start bit (usually '0'), 8 data bits (LSB first), a programmable 9 th bit and a stop bit (usually '1').
- Mode-3 is same as mode-2, except the fact that the baud rate in mode-3 is variable (i.e., just as in mode-1).
- $f_{baud} = (2^{SMOD}/32) * (f_{osc} / 12 (256\text{-}TH1))$.
- This baud rate holds when Timer-1 is programmed in Mode-2.

Serial Data Communication in 8051

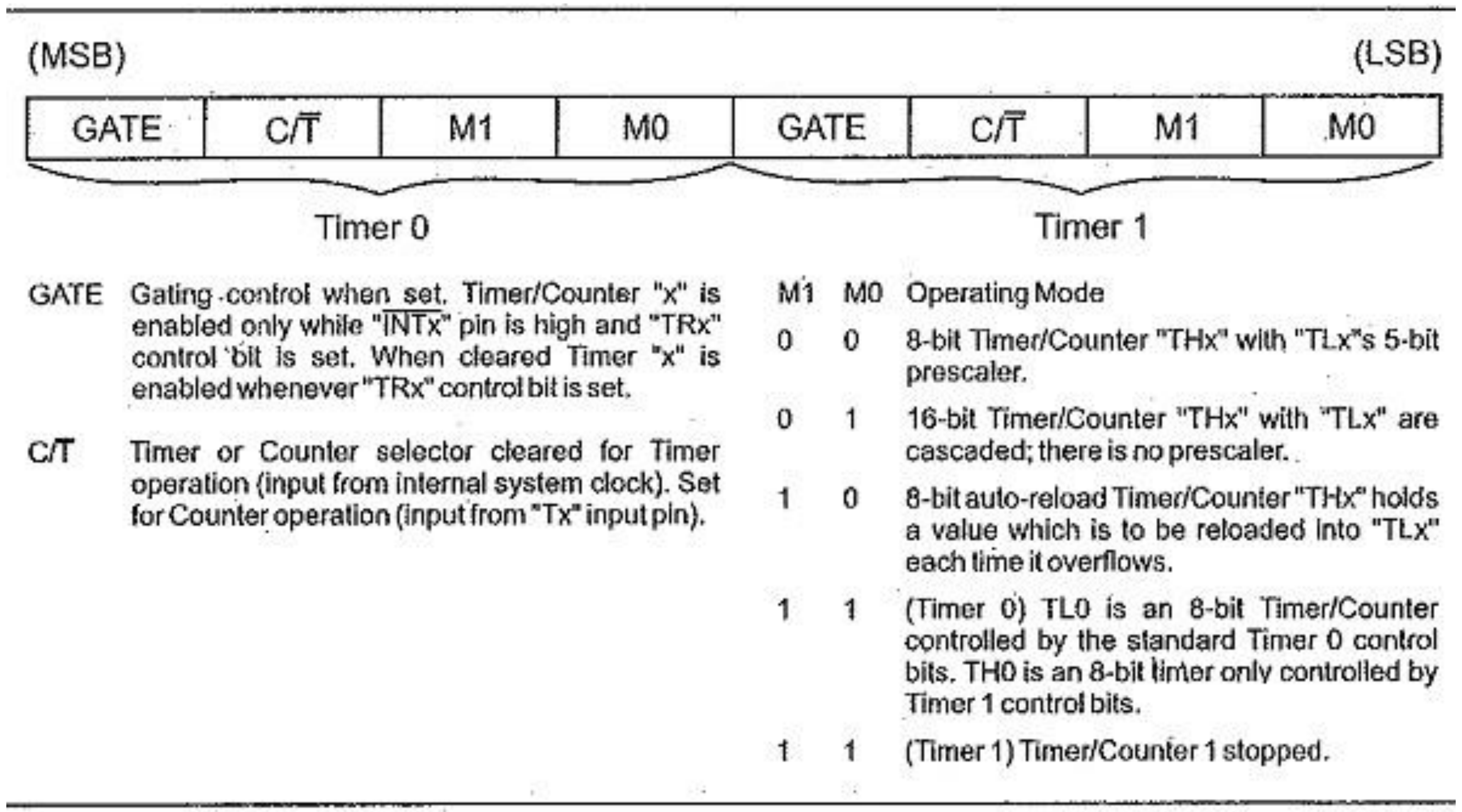
TI and RI Flags

TI (transmit interrupt)

- When 8051 finishes the transfer of the 8-bit character, it raises the TI flag to indicate that it is ready to transfer another byte
- TI bit is raised at the beginning of the stop bit

RI (receive interrupt)

- When the 8051 receives data serially via RxD, it places the byte in the SBUF register then raises the RI flag bit to indicate that a byte has been received and should be picked up before it is lost
- RI is raised halfway through the stop bit



(MSB) (LSB)

GATE	C/$\overline{T}$	M1	M0	GATE	C/$\overline{T}$	M1	M0

$\underbrace{\text{GATE C/}\overline{T}\text{ M1 M0}}_{\text{Timer 0}}$ $\underbrace{\text{GATE C/}\overline{T}\text{ M1 M0}}_{\text{Timer 1}}$

GATE	Gating control when set. Timer/Counter "x" is enabled only while "$\overline{\text{INTx}}$" pin is high and "TRx" control bit is set. When cleared Timer "x" is enabled whenever "TRx" control bit is set.
C/$\overline{T}$	Timer or Counter selector cleared for Timer operation (input from internal system clock). Set for Counter operation (input from "Tx" input pin).

M1	M0	Operating Mode
0	0	8-bit Timer/Counter "THx" with "TLx"s 5-bit prescaler.
0	1	16-bit Timer/Counter "THx" with "TLx" are cascaded; there is no prescaler.
1	0	8-bit auto-reload Timer/Counter "THx" holds a value which is to be reloaded into "TLx" each time it overflows.
1	1	(Timer 0) TL0 is an 8-bit Timer/Counter controlled by the standard Timer 0 control bits. TH0 is an 8-bit timer only controlled by Timer 1 control bits.
1	1	(Timer 1) Timer/Counter 1 stopped.

Fig. 12.15 TMOD : Timer/counter mode control register

Serial Data Communication in 8051

Programming the 8051 to transfer character bytes serially

1. TMOD register is loaded with the value 20H, indicating the use of timer 1 in mode 2 (8-bit auto-reload) to set baud rate.
2. The TH1 is loaded with one of the value to set baud rate for serial data transfer.
3. The SCON register is loaded with the value 50H, indicating serial mode 1, where an 8- bit data is framed with start and stop bits.
4. TR1 is set to 1 to start timer 1.
5. TI is cleared by CLR TI instruction.
6. The character byte to be transferred serially is written into SBUF register.
7. The TI flag bit is monitored with the use of instruction JNB TI,xx to see if the character has been transferred completely.
8. To transfer the next byte, go to step 5.

Serial Data Communication in 8051

Steps that 8051 goes through in transmitting a character via TxD

1. The byte character to be transmitted is written into the SBUF register
2. The start bit is transferred
3. The 8-bit character is transferred on bit at a time
4. The stop bit is transferred
   - ✓ It is during the transfer of the stop bit that 8051 raises the TI flag, indicating that the last character was transmitted
5. By monitoring the TI flag, we make sure that we are not overloading the SBUF
   - ✓ If we write another byte into the SBUF before TI is raised, the untransmitted portion of the previous byte will be lost
6. After SBUF is loaded with a new byte, the TI flag bit must be forced to 0 by CLR TI in order for this new byte to be transferred

Serial Data Communication in 8051

Importance of TI Flag

- By checking the TI flag bit, we know whether or not the 8051 is ready to transfer another byte
- If we write a byte into SBUF before the TI flag bit is raised, we risk the loss of a portion of the byte being transferred

Serial Data Communication in 8051

Programming the 8051 to receive character bytes serially

1. TMOD register is loaded with the value 20H, indicating the use of timer 1 in mode 2 (8-bit auto-reload) to set baud rate
2. The TH1 is loaded with one of the value to set baud rate for serial data transfer
3. The SCON register is loaded with the value 50H, indicating serial mode 1, where an 8- bit data is framed with start and stop bits
4. TR1 is set to 1 to start timer 1
5. RI is cleared by CLR RI instruction
6. The RI flag bit is monitored with the use of instruction JNB RI,xx to see if the entire character has been received yet
7. When RI is raised, SBUF has the byte, its contents are moved into a safe place to transfer the next byte, go to step 5

Serial Data Communication in 8051

Steps that 8051 goes through in receiving a character via RxD

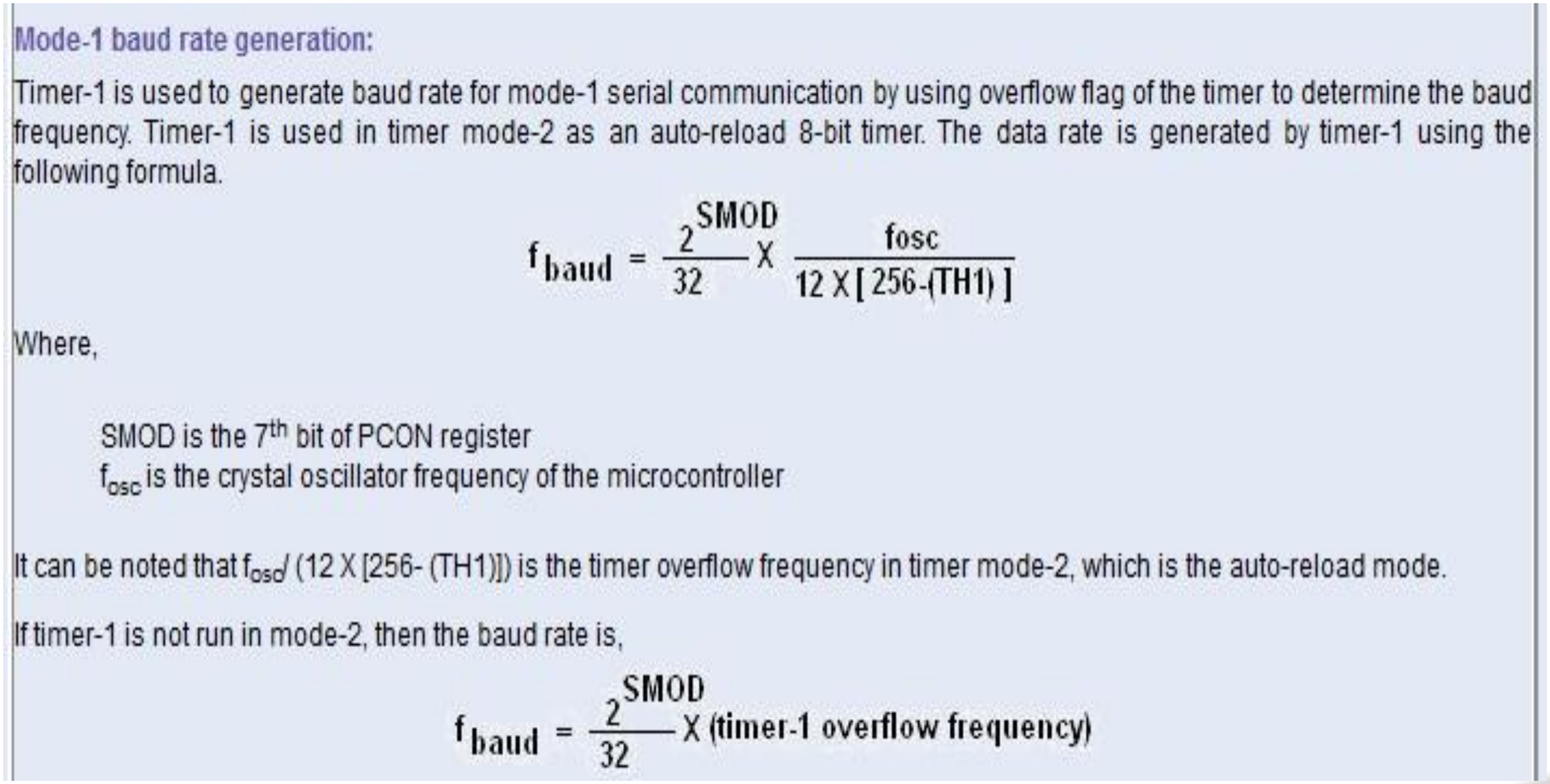
1. It receives the start bit
   - ✓ Indicating that the next bit is the first bit of the character byte it is about
     to receive
2. The 8-bit character is received one bit at time
3. The stop bit is received
   - ✓ When receiving the stop bit 8051 makes RI = 1, indicating that an entire character byte has been received and must be picked up before it gets overwritten by an incoming character
4. By checking the RI flag bit when it is raised, we know that a character has been received and is sitting in the SBUF register
   - ✓ We copy the SBUF contents to a safe place in some other register or memory before it is lost
5. After the SBUF contents are copied into a safe place, the RI flag bit must be forced to 0 by CLR RI in order to allow the next received character byte to be placed in SBUF
   - ✓ Failure to do this causes loss of the received character

Serial Data Communication in 8051

Importance of RI Flag

- By checking the RI flag bit, we know whether or not the 8051 received a character byte
- If we copy SBUF into a safe place before the RI flag bit is raised, we risk copying garbage

Serial Data Communication in 8051



Mode-1 baud rate generation:

Timer-1 is used to generate baud rate for mode-1 serial communication by using overflow flag of the timer to determine the baud frequency. Timer-1 is used in timer mode-2 as an auto-reload 8-bit timer. The data rate is generated by timer-1 using the following formula.

$$f_{baud} = \frac{2^{SMOD}}{32} X \frac{fosc}{12 X [256-(TH1)]}$$

Where,

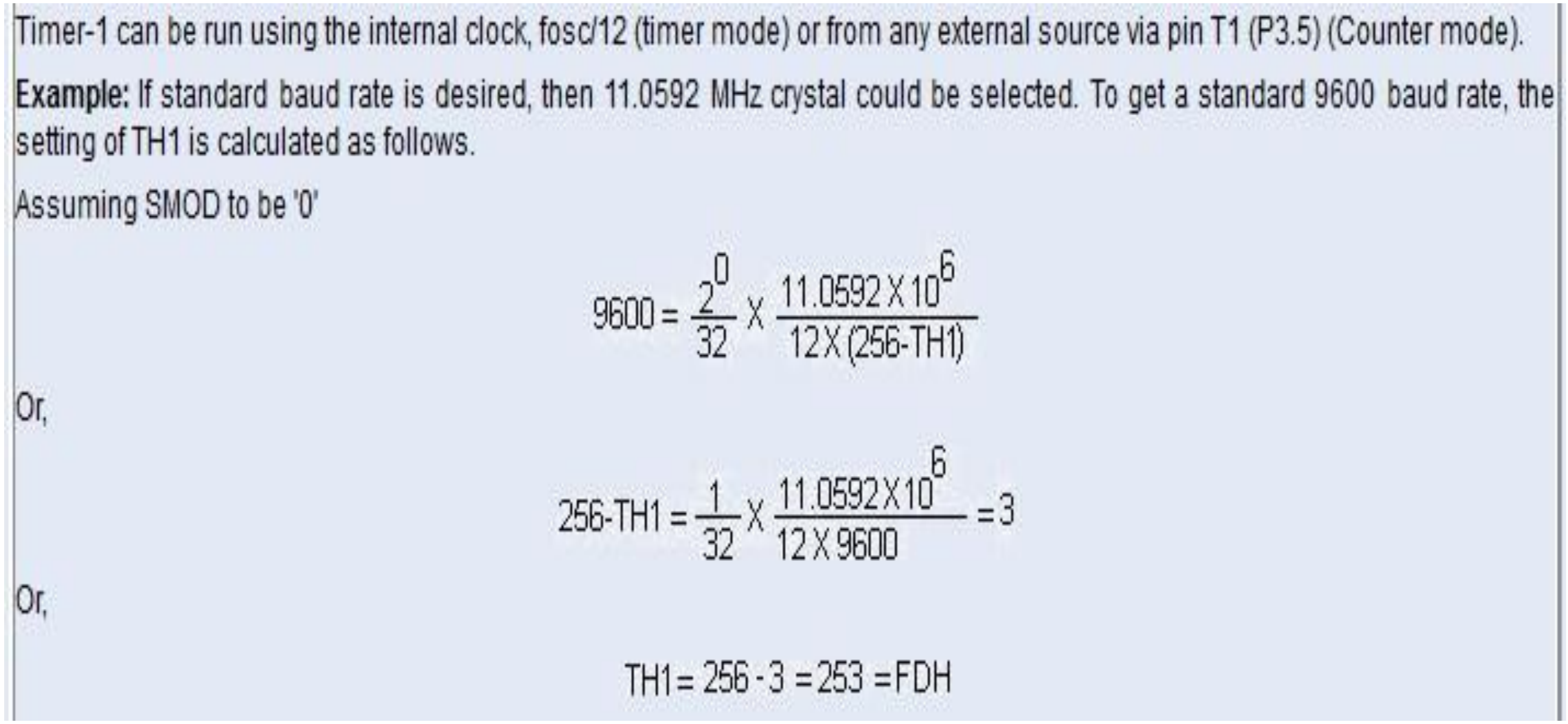
SMOD is the 7th bit of PCON register
f_{osc} is the crystal oscillator frequency of the microcontroller

It can be noted that f_{osc}/ (12 X [256- (TH1)]) is the timer overflow frequency in timer mode-2, which is the auto-reload mode.

If timer-1 is not run in mode-2, then the baud rate is,

$$f_{baud} = \frac{2^{SMOD}}{32} X \text{ (timer-1 overflow frequency)}$$

Serial Data Communication in 8051



Timer-1 can be run using the internal clock, fosc/12 (timer mode) or from any external source via pin T1 (P3.5) (Counter mode).

Example: If standard baud rate is desired, then 11.0592 MHz crystal could be selected. To get a standard 9600 baud rate, the setting of TH1 is calculated as follows.

Assuming SMOD to be '0'

$$9600 = \frac{2^0}{32} X \frac{11.0592 X 10^6}{12 X (256\text{-}TH1)}$$

Or,

$$256\text{-}TH1 = \frac{1}{32} X \frac{11.0592 X 10^6}{12 X 9600} = 3$$

Or,

$$TH1 = 256 - 3 = 253 = FDH$$

Serial Data Communication in 8051

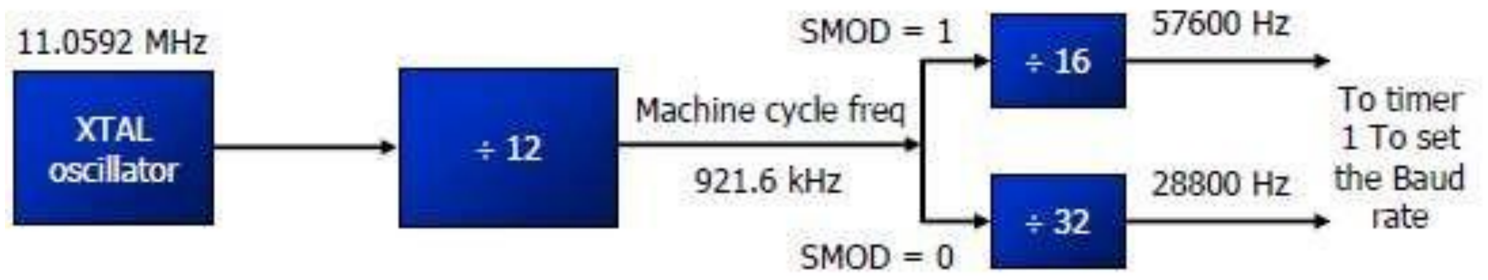
Doubling the Baud Rate in 8051

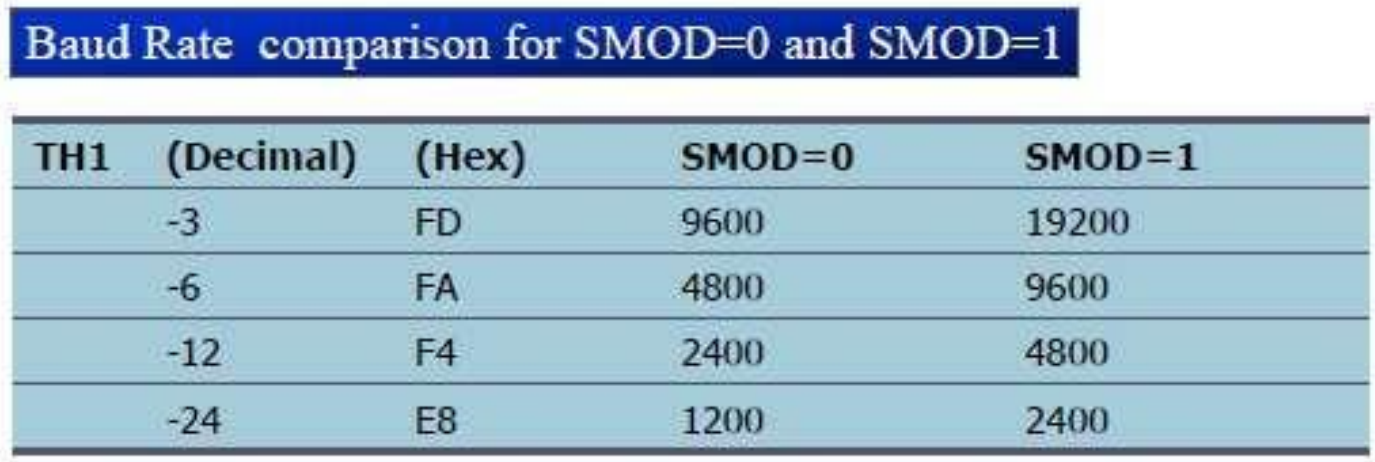
- There are two ways to increase the baud rate of data transfer
  - ✓ To use a higher frequency crystal
  - ✓ To set the SMOD bit in the PCON register
- PCON register is an 8-bit register, whose MSB is SMOD
- When 8051 is powered up, SMOD is zero
- We can set it to high by software and thereby double the baud rate
- PCON is not bit-addressable register. Hence, we cannot set SMOD bit directly. This may be done as:

```
MOV A, PCON          ;place a copy of PCON in ACC
SETB ACC.7           ;make D7=1
MOV PCON,A           ;changing any other bits
```

Serial Data Communication in 8051

Doubling the Baud Rate in 8051



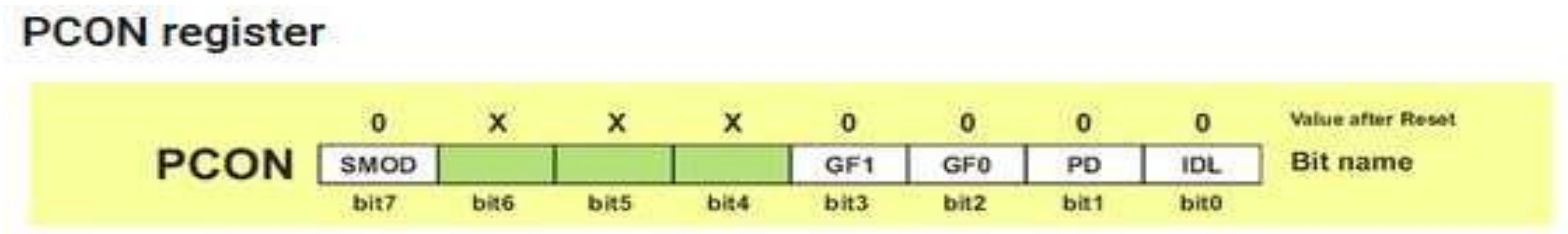


Baud Rate comparison for SMOD=0 and SMOD=1

TH1	(Decimal)	(Hex)	SMOD=0	SMOD=1
	-3	FD	9600	19200
	-6	FA	4800	9600
	-12	F4	2400	4800
	-24	E8	1200	2400

Serial Data Communication in 8051

PCON register and Power Saving Modes in 8051



PCON register

	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	
	0	X	X	X	0	0	0	0	Value after Reset
PCON	SMOD	🟩	🟩	🟩	GF1	GF0	PD	IDL	Bit name

The purpose of the Register PCON bits is:

- SMOD Baud rate is twice as much higher by setting this bit.
- GF1 General-purpose bit (available for use).
- GF1 General-purpose bit (available for use).
- GF0 General-purpose bit (available for use).
- PD By setting this bit the microcontroller enters the Power Down mode.
- IDL By setting this bit the microcontroller enters the Idle mode.

Serial Data Communication in 8051

PCON register and Power Saving Modes in 8051

- Generally speaking, the microcontroller is inactive for the most part and just waits for some external signal in order to takes its role in a show.
- This can cause some problems in case batteries are used for power supply.
- In extreme cases, the only solution is to set the whole electronics in sleep mode in order to minimize consumption.

Serial Data Communication in 8051

PCON register and Power Saving Modes in 8051

Idle Mode

- Upon the IDL bit of the PCON register is set, the microcontroller turns off the CPU unit while peripheral units such as serial port, timers and interrupt system continue operating normally
- In Idle mode, the state of all registers and I/O ports remains unchanged.
- In order to exit the Idle mode and make the microcontroller operate normally, it is necessary to reset.
- It will cause the IDL bit to be automatically cleared and the program resumes operation from instruction having set the IDL bit.

Serial Data Communication in 8051

PCON register and Power Saving Modes in 8051

Power Down Mode

- By setting the PD bit of the PCON register from within the program, the microcontroller is set to Power down mode,
- In power down mode, internal oscillator is off and reduces power consumption enormously.
- The only way to get the microcontroller back to normal mode is by reset.
- While the microcontroller is in Power Down mode, the state of all SFR registers and I/O ports remains unchanged.
- By setting it back into the normal mode, the contents of the SFR register is lost, but the content of internal RAM is saved.

Serial Data Communication in 8051

Write a C program for the 8051 to transfer the letter "A" serially at 4800 baud continuously. Use 8-bit data and I stop bit.

Solution:

```
#include <reg51.h>
void main(void)
{
    TMOD=0x20;              //use Timer 1,8-BIT auto-reload
    TH1=0xFA;               //4800 baud rate
    SCON=0x50;
    TR1=1;
    while(1)
      {
      SBUF=‘A’;             //place value in buffer
      while(TI==0);
      TI=0;
      }
}
```

Unit-III
RTOS FUNDAMENTALS AND PROGRAMMING

SYLLABUS:

- Operating system basics, types of operating systems, tasks and task states, process and threads, multiprocessing and multitasking, how to choose an RTOS ,task scheduling, semaphores and queues, hard real-time scheduling considerations, saving memory and power.

- **Task communication:** Shared memory, message passing, remote procedure call and sockets; Task synchronization: Task communication synchronization issues, task synchronization techniques, device drivers.

Operating System Basics

Operating System Basics:

- The **operating system** acts as **a bridge between** the user applications / tasks and the underlying system resources through a set of system functionalities and services.
- The OS manages the system resources and makes them available to the user applications/tasks on a need basis.
- A normal computing system is a collection of different I/O subsystems, working, and storage memory.
- The primary functions of an operating system are
  - Make the system convenient to use.
  - Organize and manage the system resources efficiently and correctly.

Figure 1 gives an insight into the basic components of an operating system and their interfaces with rest of the world.

Operating System Basics

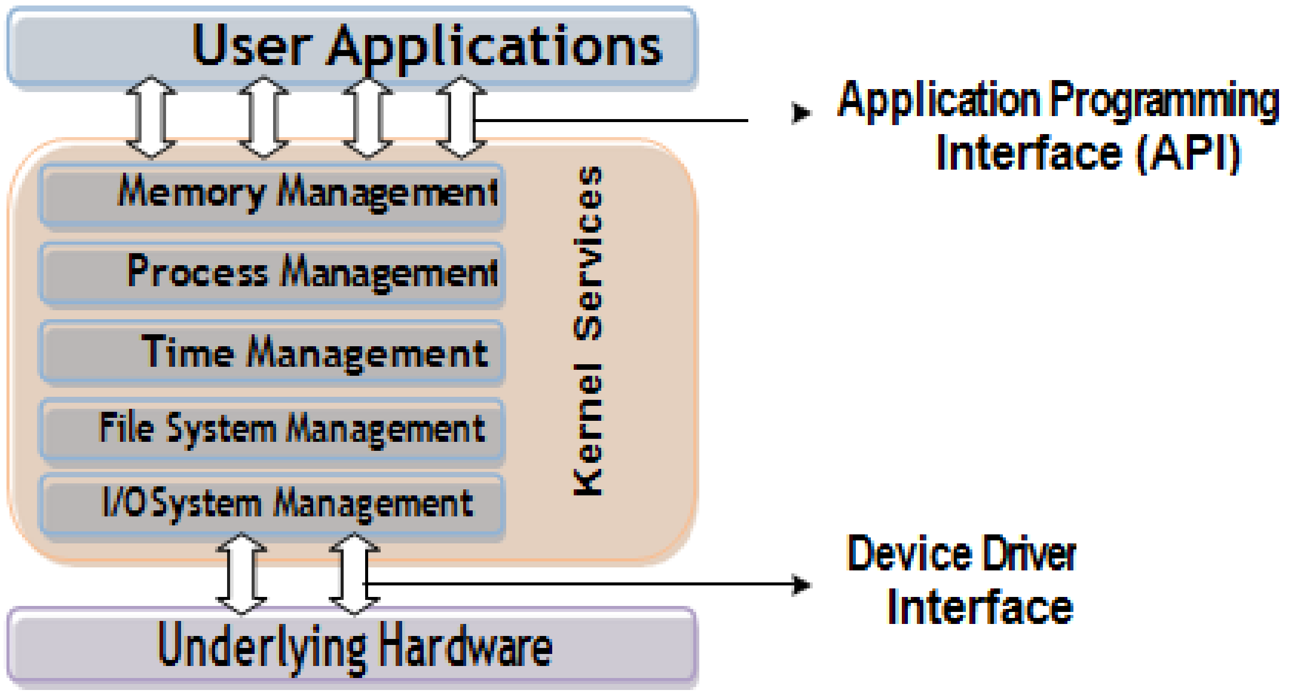


Figure 1: The Architecture of Operating System

Operating System Basics

(i)The Kernel:

- 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**.

Operating System Basics

For a general purpose OS, the kernel contains different services for handling the following.

Process Management:

Process management deals with managing the processes/tasks. Process management 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 synchronization,
- Process termination/deletion, etc.

Operating System Basics

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).

Operating System Basics

File System Management:

File is a collection of related information.

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.

• Providing a flexible naming convention for the files.

The various file system management operations are OS dependent. For example, the kernel of Micro-soft® DOS OS supports a specific set of file system management operations and they are not the same as the file system operations supported by UNIX Kernel.

Operating System Basics

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.
- ➢In a well-structured OS, the direct accessing of I/O devices are not allowed and the access to them are provided through a set of Application Programming Interfaces (APIs) exposed by the kernel.
- ➢The kernel maintains a list of all the I/O devices of the system. This list may be available in advance, at the time of building the kernel.
- ➢ Some kernels, dynamically updates the list of available devices as and when a new device is installed (e.g. Windows XP kernel keeps the list updated when a new plug 'n' play USB device is attached to the system).

Operating System Basics

The service 'Device Manager' (Name may vary across different OS kernels) of the kernel is responsible for handling all I/O device 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 drivers are specific to a device or a class of devices.

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.

Operating System Basics

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.

Operating System Basics

Protection Systems:

- Most of the modern operating systems are designed in such a way to support multiple users with different levels of access permissions (e.g. Windows XP with user permissions like `Administrator', 'Standard', 'Restricted', etc.).
- Protection deals with implementing the security policies to restrict the access to both user and system resources by different applications or processes or users.
- In multiuser supported operating systems, one user may not be allowed to view or modify the whole/ portions of another user's data or profile details.
- In addition, some application may not be granted with permission to make use of some of the system resources. This kind of protection is provided by the protection services running within the kernel.

Operating System Basics

Interrupt Handler:

Kernel provides handler mechanism for all external/internal interrupts generated by the system. These are some of the important services offered by the kernel of an operating system.

It does not mean that a kernel contains no more than components / services explained above. Depending on the type of the operating system, a kernel may contain lesser number of components/services or more number of components/servicer.

Many operating systems offer a number of add-on system components / services to the kernel. Network communication, network management, user-interface graphics, timer services (delays, timeouts, etc.), error handler, database management, etc. are examples for such components/services.

Operating System Basics

Kernel Space and User Space:

- The applications/services are classified into two categories, namely: **User applications** and **kernel applications**.
- The program code corresponding to the kernel applications/services are kept in a contiguous area (OS de-pendent) of primary (working) memory and is protected from the unauthorized access by user programs/ applications.
- 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.

Operating System Basics

- 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.
- Each process run in its own virtual memory space and are not allowed accessing the memory space corresponding to another processes, unless explicitly requested by the process.

Operating System Basics

Monolithic Kernel and Microkernel

As we know, 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**'.

(i) '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 utilization of the low-level features of the underlying system.
- The major drawback of monolithic kernel is that any error or failure in any one of the kernel modules leads to the crashing of the entire kernel application.
- **Examples of monolithic kernel:** LINUX, SOLARIS, MS-DOS. The architecture representation of a monolithic kernel is given in figure.

Operating System Basics

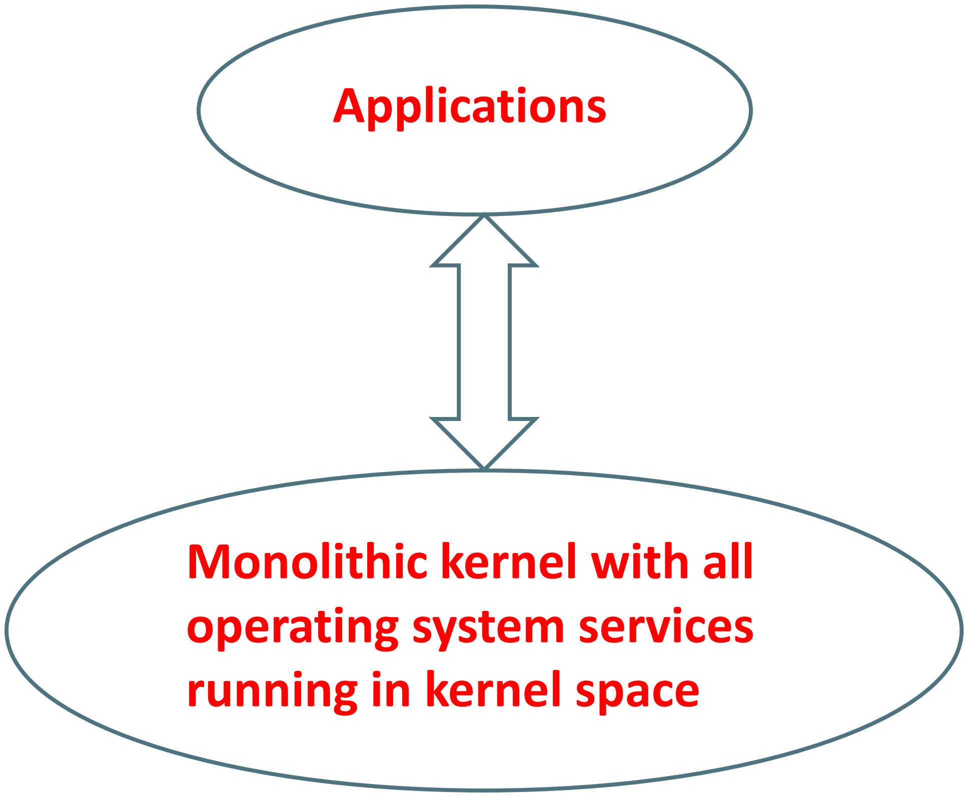

Applications
Monolithic kernel with all operating system services running in kernel space

Figure: Monolithic kernel model

Operating System Basics

(ii) Microkernel: 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.
- The kernel design is highly modular provides OS-neutral abstraction.
- Memory management, process management, timer systems and interrupt handlers are examples of essential services, which forms the part of the microkernel.

Examples for microkernel: **QNX, Minix 3 kernels.**

Benefits of Microkernel:

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

Operating System Basics

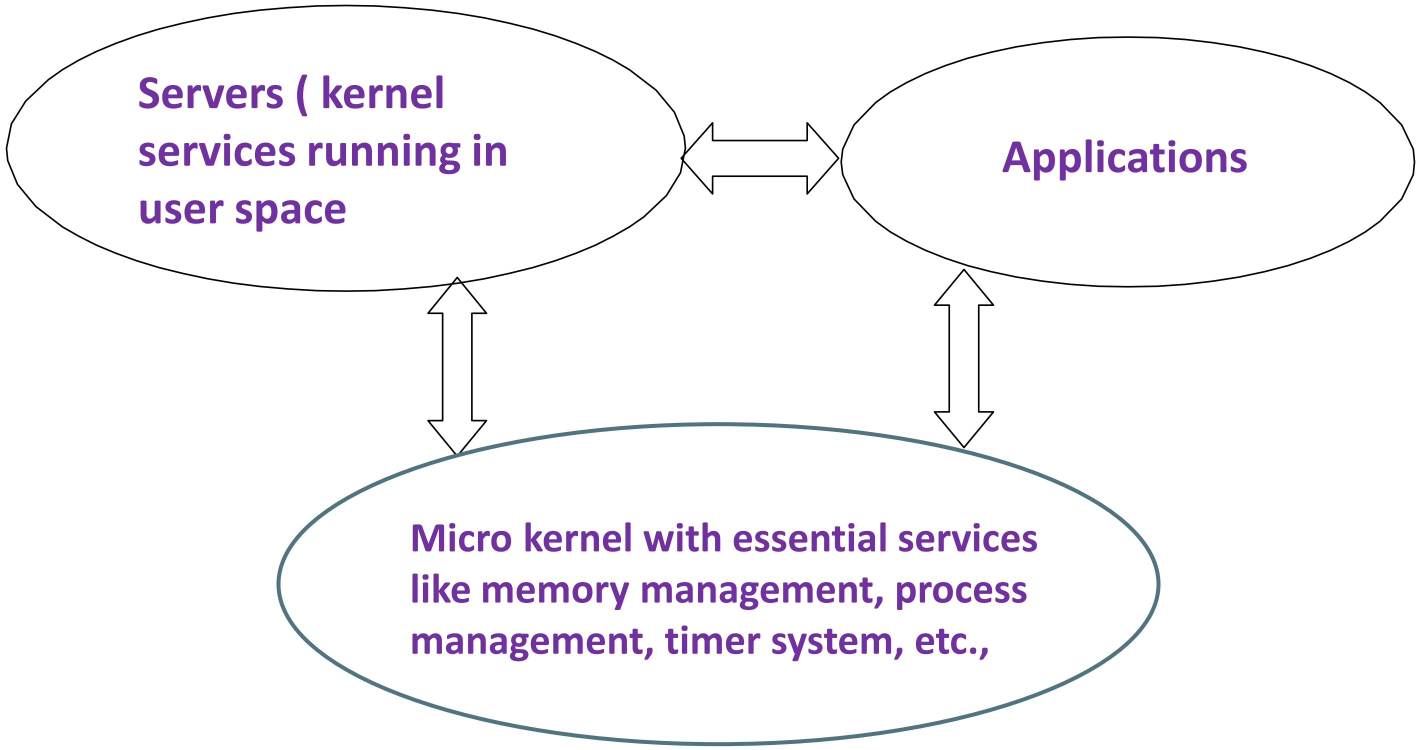
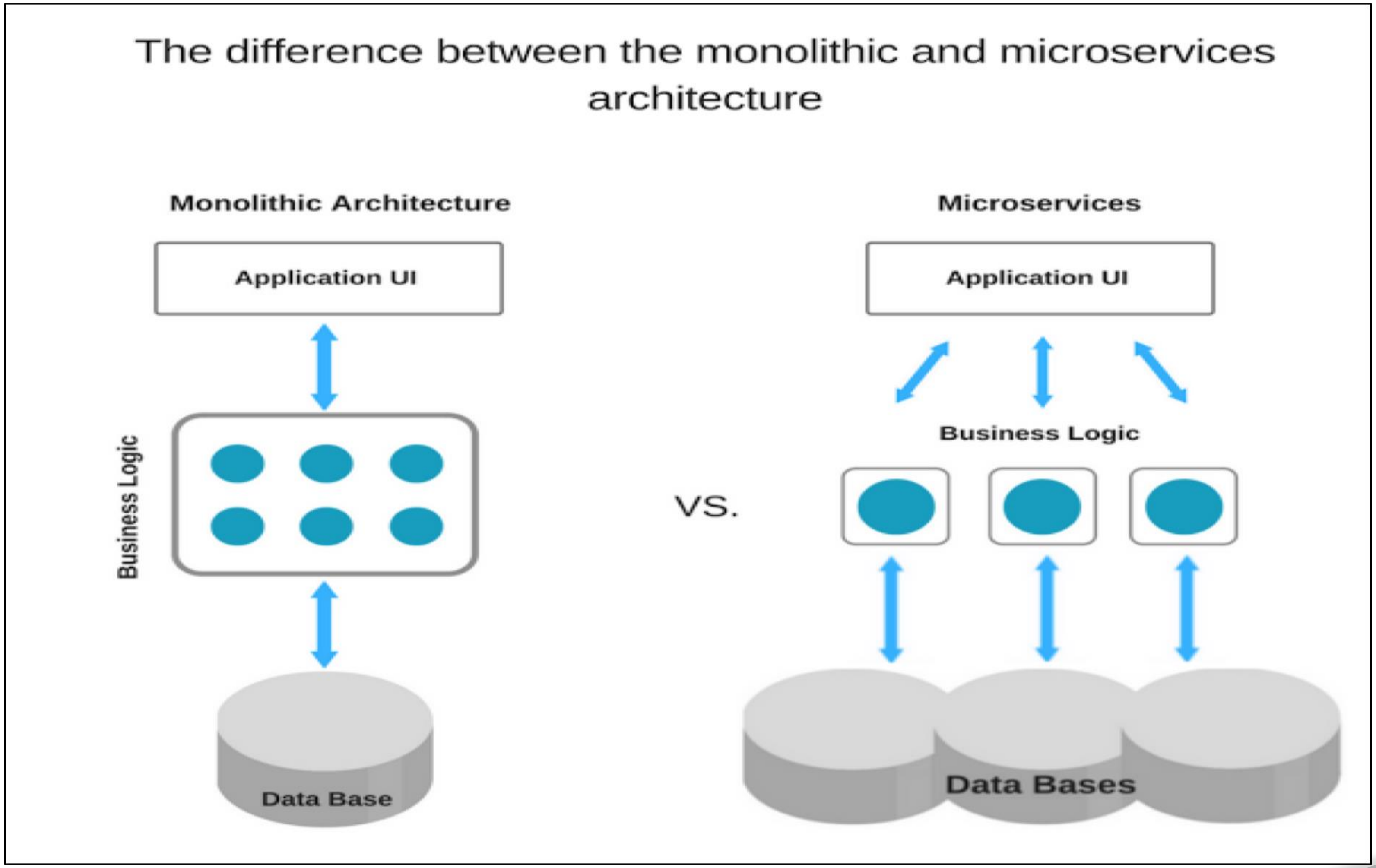

Servers (kernel services running in user space
Applications
Micro kernel with essential services like memory management, process management, timer system, etc.,

Figure: The Microkernel model


The difference between the monolithic and microservices architecture
Monolithic Architecture
Application UI
Business Logic
Data Base
VS.
Microservices
Application UI
Business Logic
Data Bases

Types of Operating Systems

Types of Operating Systems:

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

1) General Purpose Operating System (GPOS).

2) Real Time Purpose Operating System (RTOS).

Types of Operating Systems

1) General Purpose Operating System (GPOS):

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

Types of Operating Systems

2) Real Time Purpose Operating System (RTOS):

➢Operating Systems, which are deployed in embedded systems demanding real-time response.

➢Deterministic in execution behavior. Consumes only known amount of time for kernel applications.

➢Implements scheduling policies for executing the highest priority task/application always.

➢Implements policies and rules concerning time-critical allocation of a system’s resources

➢**Windows CE, QNX, VxWorks , MicroC/OS-II** etc are examples of Real Time Operating Systems (RTOS)

Types of Operating Systems

The Real Time Kernel:

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

I. Task/Process management
II. Task/Process scheduling
III. Task/Process synchronization
IV. Error/Exception handling
V. Memory Management
VI. Interrupt handling
VII. Time management

Types of Operating Systems

I. Task/Process Management: Deals with setting up the memory space for the tasks, loading the task's code into the memory space, allocating system resources, setting up a Task Control Block (TCB) for the task and task/process termination/deletion. A Task Control Block (TCB) is used for holding the information corresponding to a task. TCB usually contains the following set of information.

▪***Task ID:*** Task Identification Number

▪***Task State:*** The current state of the task. (E.g. State= 'Ready' for a task which is ready to execute)

▪***Task Type*:** Task type indicates what is the type for this task. The task can be a hard real time or soft real time or background task.

Types of Operating Systems

- ***Task Priority***: Task priority (E.g. Task priority =1 for task with priority).

- ***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).

Types of Operating Systems

II. Task/Process Scheduling:

➢Deals with sharing the CPU among various tasks/processes. A kernel application called ‘*Scheduler*’ handles the task scheduling.

➢Scheduler is nothing but an algorithm implementation, which performs the efficient and optimal scheduling of tasks to provide a deterministic behavior.

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

Types of Operating Systems

IV. Error/Exception handling:

- Deals with registering and handling the errors occurred/exceptions raised during the execution of tasks.
- Insufficient memory, timeouts, deadlocks, deadline missing, bus error, divide by zero, unknown instruction execution etc, are examples of errors/exceptions.
- Errors/Exceptions can happen at the kernel level services or at task level. Deadlock is an example for kernel level exception,
- **Deadlock** is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process. ...
- whereas timeout is an example for a task level exception. The OS kernel gives the information about the error in the form of a system call (API).

Types of Operating Systems

V. Memory Management:

- The memory management function of an RTOS kernel is slightly different compared to the General Purpose Operating Systems.

- The memory allocation time increases depending on the size of the block of memory needs to be allocated and the state of the allocated memory block (initialized memory block consumes more allocation time than un- initialized memory block)

- Since predictable timing and deterministic behavior are the primary focus for an RTOS, RTOS achieves this by compromising the effectiveness of memory allocation.

- RTOS generally uses ‘block’ based memory allocation technique, instead of the usual dynamic memory allocation techniques used by the GPOS.

Types of Operating Systems

- RTOS kernel uses blocks of fixed size of dynamic memory and the block is allocated for a task on a need basis. The blocks are stored in a ‘Free buffer Queue’.

- Most of the RTOS kernels allow tasks to access any of the memory blocks without any memory protection to achieve predictable timing and avoid the timing overheads.

- The memory management function of an RTOS kernel is slightly different compared to the General Purpose Operating Systems.

- A few RTOS kernels implement Virtual Memory concept for memory allocation if the system supports secondary memory storage (like HDD and FLASH memory).

Types of Operating Systems

VI. Interrupt Handling:

- Interrupts inform the processor that an external device or an associated task requires immediate attention of the CPU.
- Interrupts can be either **Synchronous** or **Asynchronous**.
- Interrupts which occurs in sync with the currently executing task is known as "**Synchronous interrupts**". Usually the software interrupts fall under the Synchronous Interrupt category. **Divide by zero, memory segmentation error** etc are examples of Synchronous interrupts.
- For synchronous interrupts, the interrupt handler runs in the same context of the interrupting task.
- **Asynchronous interrupts** are interrupts, which occurs at any point of execution of any task, and are not in sync with the currently executing task.

Types of Operating Systems

➢**The interrupts generated by external connected to the processor/ controller, timer overflow interrupts, serial data reception/ transmission interrupts** etc are examples for asynchronous interrupts.

➢For asynchronous interrupts, the interrupt handler is usually written as separate task (Depends on OS Kernel implementation) and it runs in a different context. Hence, a context switch happens while handling the asynchronous interrupts.

➢Priority levels can be assigned to the interrupts and each interrupts can be enabled or disabled individually.

➢Most of the RTOS kernel implements 'Nested Interrupts' architecture. Interrupt nesting allows the pre-emption (interruption) of an Interrupt Service Routine (ISR), servicing an interrupt, by a higher priority interrupt.

Types of Operating Systems

VII. Time Management:

➢Interrupts inform the processor that an external device or an associated task requires immediate attention of the CPU. Accurate time management is essential for providing precise time reference for all applications.

➢The time reference to kernel is provided by a high-resolution Real Time Clock (RTC) hardware chip (hardware timer).

➢The hardware timer is programmed to interrupt the processor/controller at a fixed rate. This timer interrupt is referred as ‘Timer tick’.

➢The ‘Timer tick’ is taken as the timing reference by the kernel. The ‘Timer tick’ interval may vary depending on the hardware timer. Usually the ‘Timer tick’ varies in the microseconds range.

➢The time parameters for tasks are expressed as the multiples of the ‘Timer tick’. The System time is updated based on the ‘Timer tick’.

Types of Operating Systems

Hard Real-time System:

➢A Real Time Operating Systems which strictly adheres to the timing constraints for a task.

➢A Hard Real Time system must meet the deadlines for a task without any slippage.

➢Missing any deadline may produce catastrophic results for Hard Real Time Systems, including permanent data lose and irrecoverable damages to the system/users.

➢Emphasize on the principle 'A late answer is a wrong answer'.

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

Types of Operating Systems

Soft Real-time System:

- Real Time Operating Systems that does not guarantee meeting deadlines, but, offer the best effort to meet the deadline.
- Missing deadlines for tasks are acceptable if the frequency of deadline missing is within the compliance limit of the Quality of Service(QoS).
- A Soft Real Time system emphasizes on the principle '*A late answer is an acceptable answer, but it could have done bit faster'.*
- **Automatic Teller Machine (ATM)** is a typical example of Soft Real Time System. If the ATM takes a few seconds more than the ideal operation time, nothing fatal happens.
- An **audio video play back system** is another example of Soft Real Time system. No potential damage arises if a sample comes late by fraction of a second, for play back.

Tasks and Task States

Tasks and Task States:

- The term **'task'** refers to something that needs to be done.

- In our day-to-day life, we are bound to the execution of a number of tasks. The task can be the one assigned by our managers or the one assigned by our professors/teachers or the one related to our personal or family needs.

- In addition, we will have an order of priority and schedule/timeline for executing these tasks.

- In the operating system context, a **task** is defined as the program in execution and the related information maintained by the operating system for the program.

Tasks and Task States

- A task is like a process or thread in an OS
- Task term used for the process in the RTOSs for the embedded systems
- A task consists of executable program (codes), state of which is controlled by OS
- A Task Control Block (TCB)- a data structure having the information using which the OS controls the task state
- A task can be any one of these state running, blocked or finished.

Tasks and Task States

➢Task is also known as 'Job' in the operating system context. A program or part of it in execution is also called a 'Process' .

➢The terms '**Task**', `**Job**' and '**Process**' refer to the same entity in the operating system context and most often they are used interchangeably.

Process and Threads

Process:

- A '**Process**' is a program and it is a single execution of a program.
- Each process has its own state that includes registers and memory.
- 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.
- **Threads**-are Processes that share the same address space.
- **Task**- composed of several processes or threads.

Process and Threads

(i). The Structure of a Process:

The concept of 'Process' leads to concurrent execution (pseudo parallelism) of tasks and thereby the efficient utilization of the CPU and other system resources.

Concurrent execution is achieved through the sharing of CPU among the processes. A process mimics a processor in properties and holds a set of registers, process status, a Program Counter (PC) to point to the next executable instruction of the process, a stack for holding the local variables associated with the process and the code corresponding to the process. This can be visualized as shown in Figure.

Process and Threads

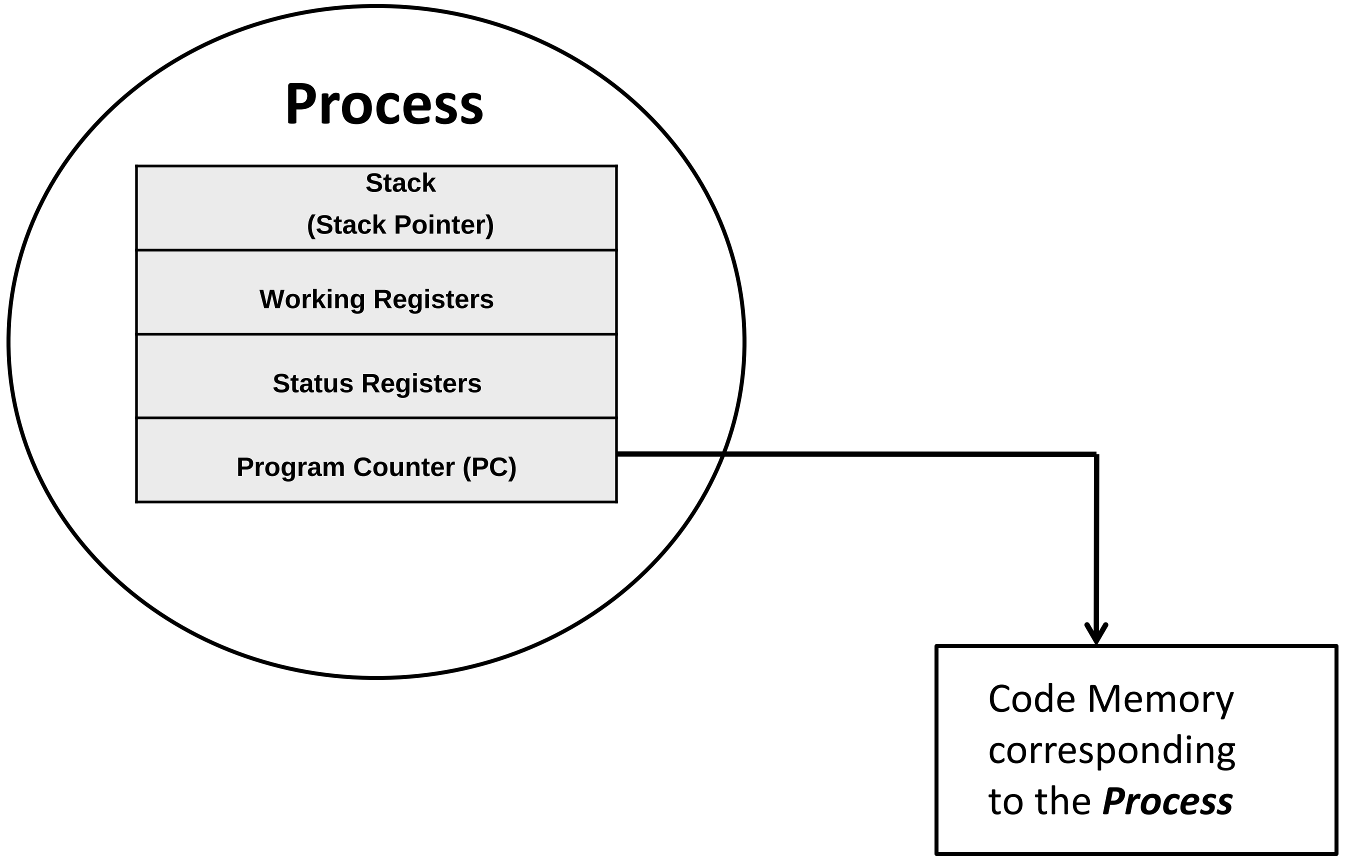


Figure: Structure of a Process

Process and Threads

Memory organization of Processes:

The memory occupied by the *process* is segregated into three regions namely; Stack memory, Data memory and Code memory.

•The **Stack memory** holds all temporary data such as variables local to the process

•**Data memory** holds all global data for the process

•The **Code memory** contains the program code (instructions) corresponding to the process

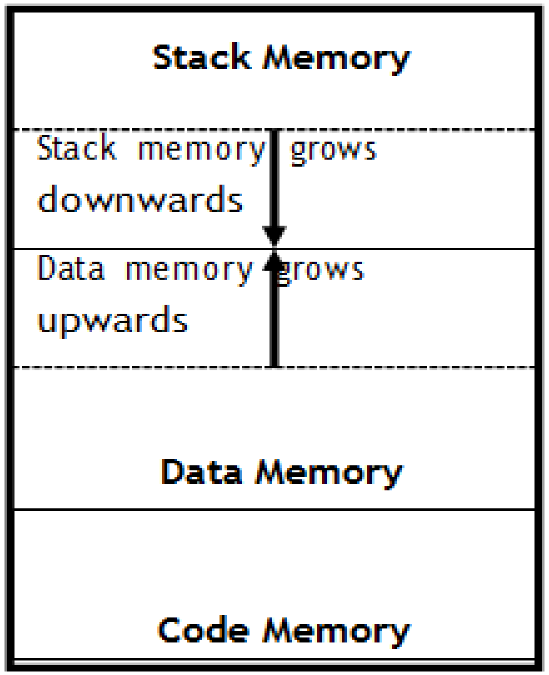


Fig: Memory organization of a Process

Process and Threads

(ii). Process States & State Transition:

➢The creation of a process to its termination is not a single step operation.

➢The process traverses through a series of states during its transition from the newly created state to the terminated state.

➢The cycle through which a process changes its state from '***newly created***' to '***execution completed***' is known as '***<u>Process Life Cycle</u>***'.

➢The various states through which a process traverses through during a Process Life Cycle indicates the current status of the process with respect to time and also provides information on what it is allowed to do next.

Process and Threads

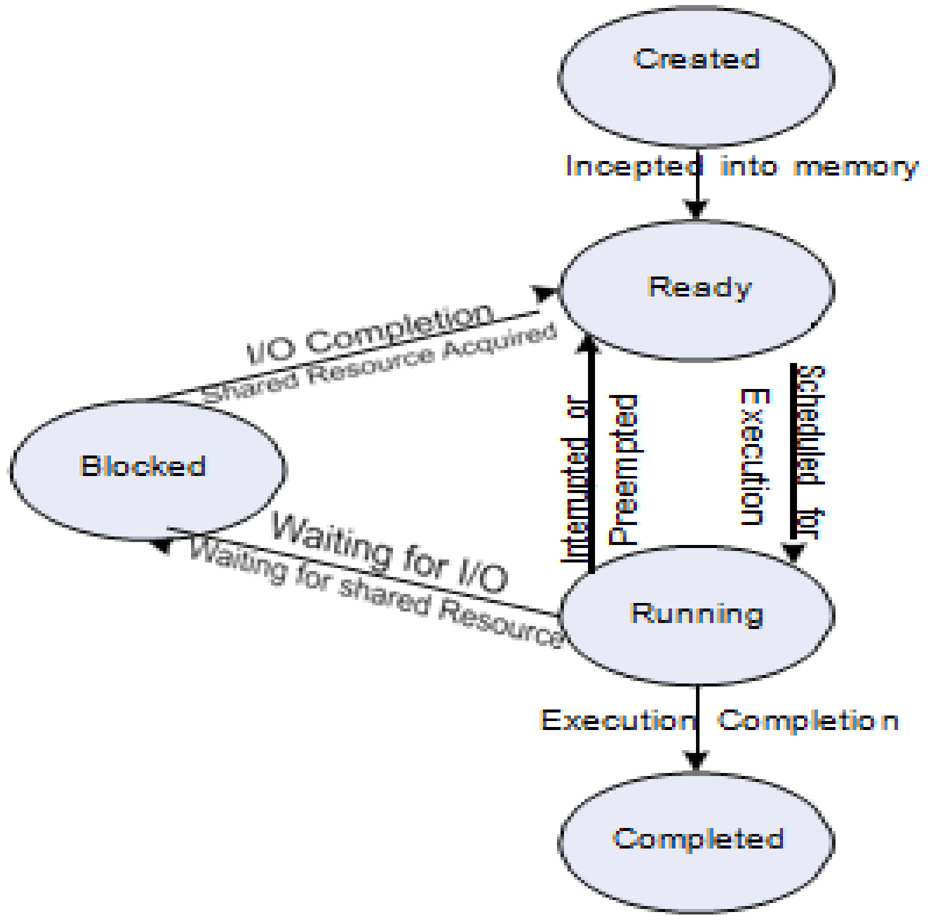


Fig: Process states and State transition

Process and Threads

Created State: The state at which a process is being created is referred as 'Created State'. The Operating System recognizes a process in the '*Created State*' but no resources are allocated to the process.

Ready State: The state, where a process is incepted into the memory and awaiting the processor time for execution, is known as '*Ready State*'. At this stage, the process is placed in the '*Ready list*' queue maintained by the OS.

Running State: The state where in the source code instructions corresponding to the process is being executed is called '*Running State*'. Running state is the state at which the process execution happens.

Process and Threads

Blocked State/Wait State: Refers to a state where a running process is temporarily suspended from execution and does not have immediate access to resources.

The blocked state might have invoked by various conditions like- the process enters a wait state for an event to occur

(E.g. Waiting for user inputs such as keyboard input) or waiting for getting access to a shared resource like semaphore, mutex etc.

Completed State: A state where the process completes its execution.

The transition of a process from one state to another is known as '***State transition***'

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

Process and Threads

(iii). Process Management :

Process management deals with the

- ⦿Creation of a process,
- ⦿Setting up the memory space for the process,
- ⦿Loading the process's code into the memory space,
- ⦿Allocating system resources,
- ⦿Setting up a Process Control Block (PCB) for the process and
  - •Process Control Block - a data structure having the information using which the OS controls the process state.
  - •Stores in the protected memory area of the kernel.
  - •Consists of the information about the process state.
- ⦿Process termination/deletion.

Process and Threads

Threads:

➢A thread is the primitive that can execute code.

➢A thread is a single sequential flow of control within a process.

➢'Thread' is also known as **lightweight process**.

➢A process can have many threads of execution.

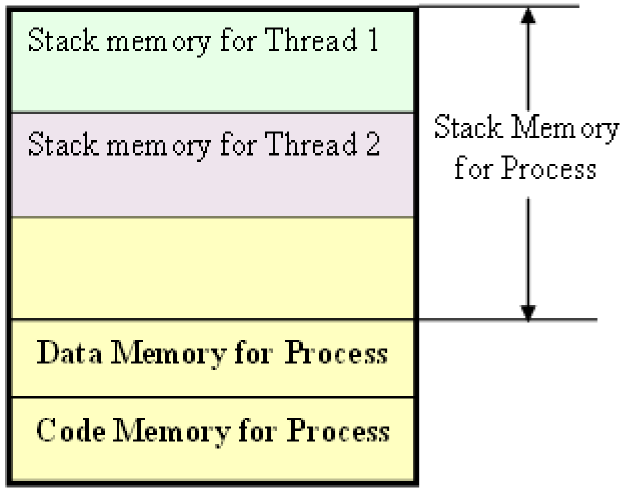


Figure : Memory organization of process and its associated Threads

➢Different threads, which are part of a process, share the same address space; meaning they share the data memory, code memory and heap memory area.

➢Threads maintain their own thread status (CPU register values), Program Counter (PC) and stack.

Process and Threads

Advantages of Threads:

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

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

Multiprocessing and Multitasking

Multiprocessing and Multitasking:

- The terms multiprocessing and multitasking are a little confusing and sounds alike.
- In the operating system context, *"Multiprocessing"* defines as 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"* .**

Multiprocessing and Multitasking

- In a uniprocessor system, it is not possible to execute multiple processes simultaneously. However, it is possible for a uniprocessor system to achieve some degree of pseudo parallelism in the execution of multiple processes by switching the execution among different processes.

- ***"Multitasking"*** refers to the ability of an operating system to hold multiple processes in memory and switch the processor (CPU) from executing one process to another process.

- Multitasking creates the illusion of multiple tasks executing in parallel. Multitasking involves the switching of CPU from executing one task to another.

Multiprocessing and Multitasking

- The switching of the virtual processor to physical processor is controlled by the scheduler of the OS kernel.

- 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.

- The act of switching CPU among the processes or changing the current execution context is known as ***'Context switching'.***

Multiprocessing and Multitasking

- 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', 'Context saving' and 'Context retrieval'.**

Multiprocessing and Multitasking

Example:

- **Toss juggling**—The skilful object manipulation game is a classic real world example for the multitasking illusion. The juggler uses a number of objects (balls, rings, etc.) and throws them up and catches them.

- At any point of time, he throws only one ball and catches only one per hand. However, the speed at which he is switching the balls for throwing and catching creates the illusion, he is throwing and catching multiple balls or using more than two hands simultaneously, to the spectators.

Multiprocessing and Multitasking

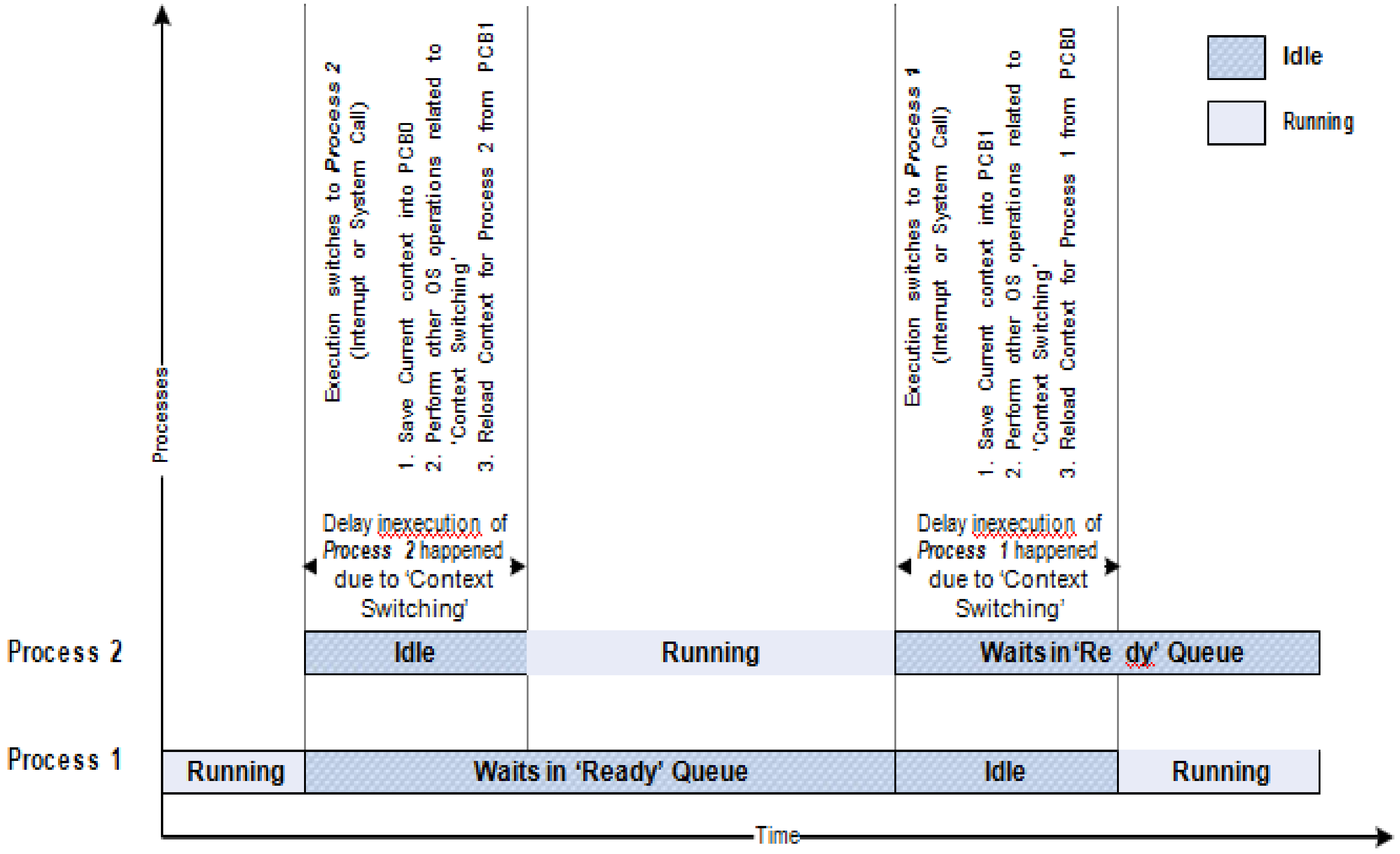

Processes
Execution switches to *Process 2* (Interrupt or System Call)
1. Save Current context into PCB0
2. Perform other OS operations related to 'Context Switching'
3. Reload Context for Process 2 from PCB1
Execution switches to *Process 1* (Interrupt or System Call)
1. Save Current context into PCB1
2. Perform other OS operations related to 'Context Switching'
3. Reload Context for Process 1 from PCB0
Idle
Running
Delay inexecution of *Process 2* happened due to 'Context Switching'
Delay inexecution of *Process 1* happened due to 'Context Switching'
Process 2
Idle
Running
Waits in 'Re dy' Queue
Process 1
Running
Waits in 'Ready' Queue
Idle
Running
Time

Figure: Context Switching

Multiprocessing and Multitasking

Types of Multitasking :

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

(i).Co-operative Multitasking:

In co-operative multitasking, the programs or applications that are running on an operating system uses the processor and the resources associated with that operating system for the execution of the program or application.

When the application is complete the processor becomes free and is then allocated to the next application in queue. But the behavior of time assignment of multitasking operating system causes a problem, this means if a task goes into a loop or it gets blocked, then all the other applications are stopped. This can also stop the whole operating system .

Multiprocessing and Multitasking

Co-operative multitasking is the most primitive form of multitasking in which a task/process gets a chance to execute only when the currently executing task/process voluntarily relinquishes the CPU.

•In this method, any task/process can avail the CPU as much time as it wants. Since this type of implementation involves the mercy of the tasks each other for getting the CPU time for execution, it is known as ***co-operative multitasking***.

•If the currently executing task is non-cooperative, the other tasks may have to wait for a long time to get the CPU

Multiprocessing and Multitasking

(ii). 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

Multiprocessing and Multitasking

(iii). Non-preemptive Multitasking:

The process/task, which is currently given the CPU time, is allowed to execute until it terminates (enters the ‘Completed’ state) or enters the ‘Blocked/Wait’ state, waiting for an I/O.

The co-operative and non-preemptive multitasking differs in their behavior when they are in the ‘Blocked/Wait’ state. In co-operative multitasking, the currently executing process/task need not relinquish the CPU when it enters the ‘Blocked/Wait’ state, waiting for an I/O, or a shared resource access or an event to occur whereas in non-preemptive multitasking the currently executing task relinquishes the CPU when it waits for an I/O.

How to choose RTOS

How to choose RTOS:

The decision of an RTOS for an embedded design is very critical.

A lot of factors need to be analyzed carefully before making a decision on the selection of an RTOS.

These factors can be either

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

How to choose RTOS: Functional Requirements

1. Functional Requirements:

i. Processor support:

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

•It is essential to ensure the processor support by the RTOS.

ii. Memory Requirements:

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

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

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

How to choose RTOS: Functional Requirements

iii. Real-Time Capabilities:

•It is not mandatory that the OS for all embedded systems need to be Real- Time and all embedded OS’s are ‘Real-Time’ in behavior.

•The Task/process scheduling policies plays an important role in the Real-Time behavior of an OS.

iv. 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.

v. Inter process Communication (IPC) and Task Synchronization:

•The implementation of IPC and Synchronization is OS kernel dependent.

How to choose RTOS: Functional Requirements

vi. Modularization Support:

•Most of the OS's provide a bunch of features.

•It is very useful if the OS supports modularization where in which the developer can choose the essential modules and re-compile the OS image for functioning.

vii. Support for Networking and Communication:

•The OS kernel may provide stack implementation and driver support for a bunch of communication interfaces and networking.

•Ensure that the OS under consideration provides support for all the interfaces required by the embedded product.

viii. Development Language Support:

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

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

How to choose RTOS: Non-Functional Requirements

2. Non-Functional Requirements:

i. Custom Developed or Off the Shelf:

•It is possible to go for the complete development of an OS suiting the embedded system needs or use an off the shelf, readily available OS.

•It may be possible to build the required features by customizing an open source OS.

•The decision on which to select is purely dependent on the development cost, licensing fees for the OS, development time and availability of skilled resources.

ii. 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.

How to choose RTOS: Non-Functional Requirements

iii. Development and Debugging tools Availability:

•The availability of development and debugging tools is a critical decision making factor in the selection of an OS for embedded design.

•Certain OS's may be superior in performance, but the availability of tools for supporting the development may be limited.

iv. 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.

v. After Sales:

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

Task Scheduling

Task Scheduling:

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

➢Determining which task/process is to be executed at a given point of time is known as "**task/process scheduling".**

➢Task scheduling forms the basis of multitasking.

➢Scheduling policies forms the guidelines for determining which task is to be executed when.

➢The scheduling policies are implemented in an algorithm and it is run by the kernel as a service.

Task Scheduling

- The kernel service/application, which implements the scheduling algorithm, is known as ‘Scheduler’
- The task scheduling policy can be pre-emptive, non-preemptive or co-operative
- Depending on the scheduling policy the process scheduling decision may take place when a process switches its state to
    - ‘Ready’ state from ‘Running’ state
    - ‘Blocked/Wait’ state from ‘Running’ state
    - ‘Ready’ state from ‘Blocked/Wait’ state
    - ‘Completed’ state

Task Scheduling

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

CPU Utilization: The scheduling algorithm should always make the CPU utilization high. CPU utilization is a direct measure of how much percentage of the CPU is being utilized.

Throughput: This gives an indication of the number of processes executed per unit of time. The throughput for a good scheduler should always be higher.

Turnaround Time: It is the amount of time taken by a process for completing its execution. It includes the time spent by the process for waiting for the main memory, time spent in the ready queue, time spent on completing the I/O operations, and the time spent in execution. The turnaround time should be a minimum for a good scheduling algorithm.

Task Scheduling

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.

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

Task Scheduling

Task Scheduling – Queues:

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

Job Queue: Job queue contains all the processes in the system.

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

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

Classification of Scheduling

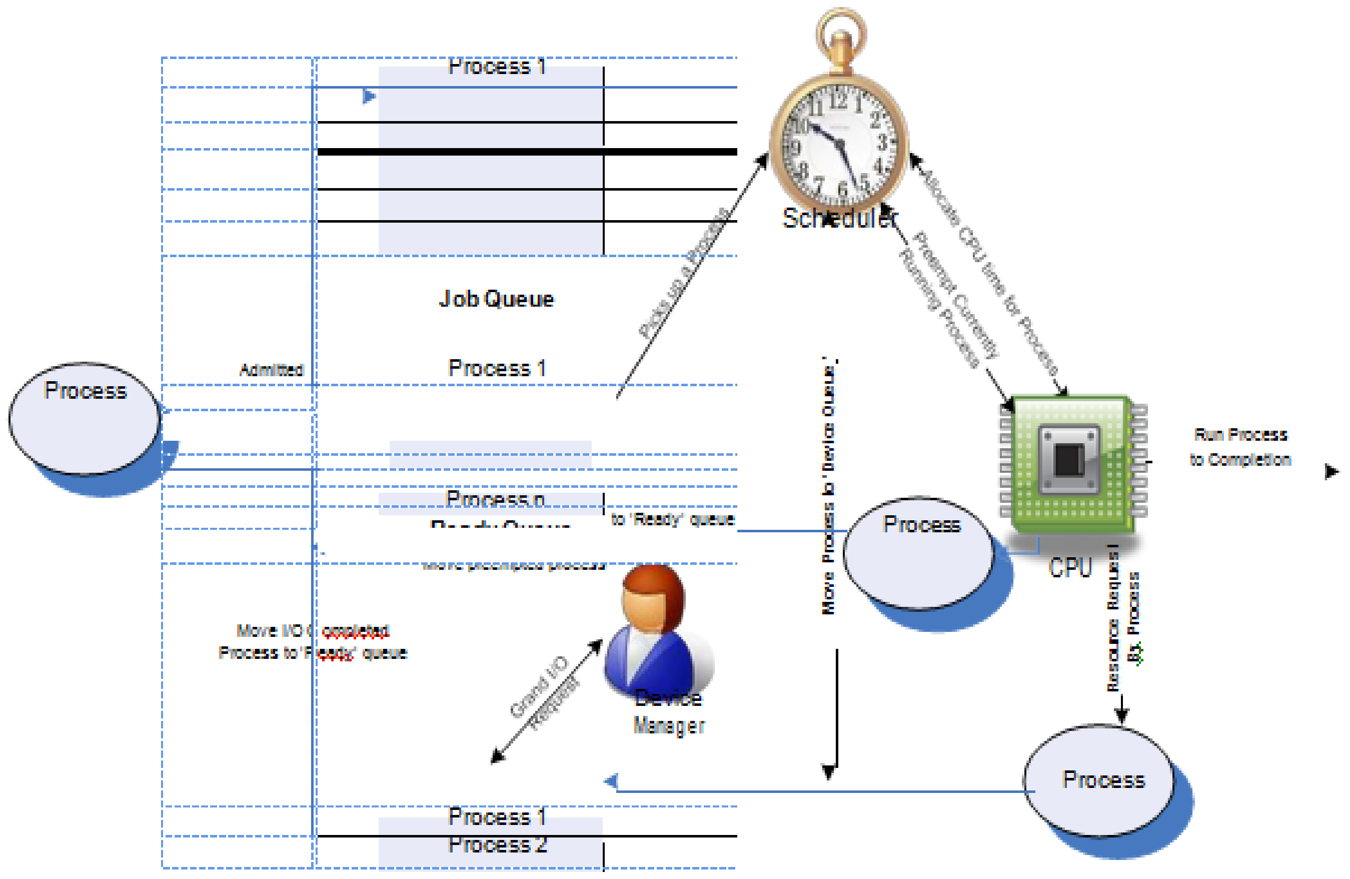


Figure: Illustration of Process Transition through various queues

Classification of Scheduling

Based on the scheduling algorithm used, the scheduling can be classified into the following categories.

1.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.

Classification of Scheduling

The various types of non-preemptive scheduling adopted in task/process scheduling are listed below.

(i). 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. It is same as any real world application where queue systems are used; e.g. Ticketing reservation system where people need to stand in a queue and the first person standing in the queue is serviced first.
- FCFS scheduling is also known as First In First Out (FIFO) where the process which is put first into the 'Ready' queue is serviced first.

Classification of Scheduling

(ii). 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.

Classification of Scheduling

(iii). 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.

Classification of Scheduling

(iv). Priority Based Scheduling:

- The Turn Around Time (TAT) and waiting time for processes in non-preemptive scheduling varies with the type of scheduling algorithm.

- 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 prioritized in the order the time required to complete the task.

- The lower the time required for completing a process the higher is the 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.

Classification of Scheduling

- 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.

- The priority variation from high to low is represented by numbers from 0 to the maximum priority or by numbers from maximum priority to 0. For Windows CE operating system a priority number' 0 indicates the highest priority and 255 indicates the lowest priority.

Classification of Scheduling

2.Preemptive Scheduling:

- In preemptive scheduling, every task in the ' Ready' queue gets a chance to execute. When and how often each process gets a chance to execute (gets the CPU time) is dependent on the type of preemptive scheduling algorithm used for scheduling the processes.
- In this kind of scheduling, the scheduler can preempt (stop temporarily) the currently executing task/process and select another task from the 'Ready' queue for execution.
- ***When to pre-empt a task and which task is to be picked up from the 'Ready' queue for execution after preempting the current task is purely dependent on the scheduling algorithm***. A task which is preempted by the scheduler is moved to the `Ready' queue. The act of moving a `Running' process/task into the `Ready' queue by the scheduler, without the processes requesting for it is known as *'Preemption'.*

Classification of Scheduling

- Preemptive scheduling can be implemented in different approaches.
- The two important approaches adopted in preemptive scheduling are time-based preemption and priority-based preemption.
- The various types of preemptive scheduling adopted in task/process scheduling are explained below.

Classification of Scheduling

(i).Preemptive SJF Scheduling/Shortest Remaining Time (SRT):

- The non-preemptive SJF scheduling algorithm sorts the 'Ready' queue only after completing the execution of the current process or when the process enters 'Wait' state, whereas the preemptive SF 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.

Classification of Scheduling

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

- Thus preemptive SJF scheduling always compares the execution completion time (It is same as the remaining time for the new process) of a new process entered the 'Ready' queue with the remaining time for completion of the currently executing process and schedules the process with shortest remaining time for execution.

- Preemptive SJF scheduling is also known as Shortest Remaining Time (SRT) scheduling.

Classification of Scheduling

EXAMPLE:

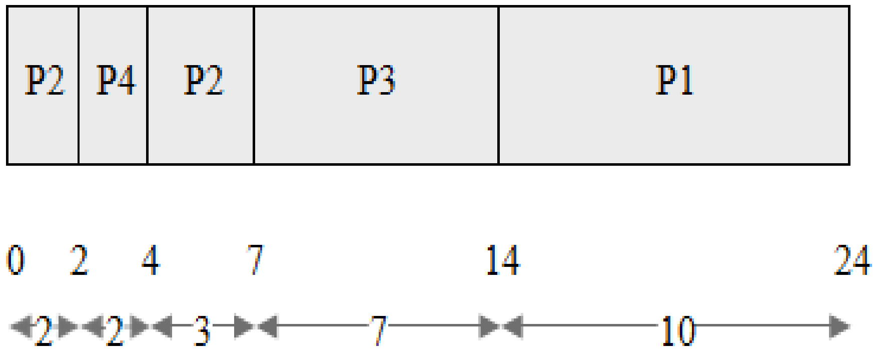
Three processes with process IDs P1, P2, P3 with estimated completion time 10, 5, 7 milliseconds respectively enters the ready queue together. A new process P4 with estimated completion time 2ms enters the 'Ready' queue after 2ms. Assume all the processes contain only CPU operation and no I/O operations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

Classification of Scheduling

Solution:

At the beginning, there are only three processes (P1, P2 and P3) available in the ‘Ready’ queue and the SRT scheduler picks up the process with the Shortest remaining time for execution completion (In this example P2 with remaining time 5ms) for scheduling.

Now process P4 with estimated execution completion time 2ms enters the ‘Ready’ queue after 2ms of start of execution of P2. The processes are re-scheduled for execution in the following order



Classification of Scheduling

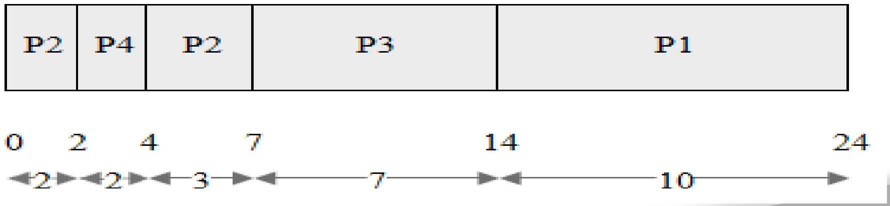
The waiting time for all the processes are given as

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

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

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

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



Classification of Scheduling

$$\boxed{\text{Average waiting time} = \frac{(\text{Waiting time for all the processes})}{\text{No. of Processes}}}$$

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

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

= 5.75 milliseconds

Classification of Scheduling

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

Turn Around Time (TAT) for P4 = 2 ms

= (Execution Start Time – Arrival Time) + Estimated Execution Time = (2-2) + 2)

Turn around Time (TAT) for P3 = 14 ms

Turn around Time (TAT) for P1 = 24

$$\boxed{\text{Average Turn around Time} = \frac{(\text{Turn around Time for all the processes})}{(\text{No. of Processes})}}$$

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

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

= **11.75 milliseconds.**

Classification of Scheduling

(ii). Round Robin (RI) Scheduling:

➢The term Round Robin is very popular among the sports and games activities. You might have heard about 'Round Robin' league or 'Knock out' league associated with any football or cricket tournament.

➢In the **'Round Robin'** league each team in a group gets an equal chance to play against the rest of the teams in the same group whereas in the **'Knock out'** league the losing team in a match moves out of the tournament .

➢In Round Robin scheduling, each process in the 'Ready' queue is executed for a pre-defined time slot.

Classification of Scheduling

- The execution starts with picking up the first process in the 'Ready' queue. 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 a 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.

Classification of Scheduling

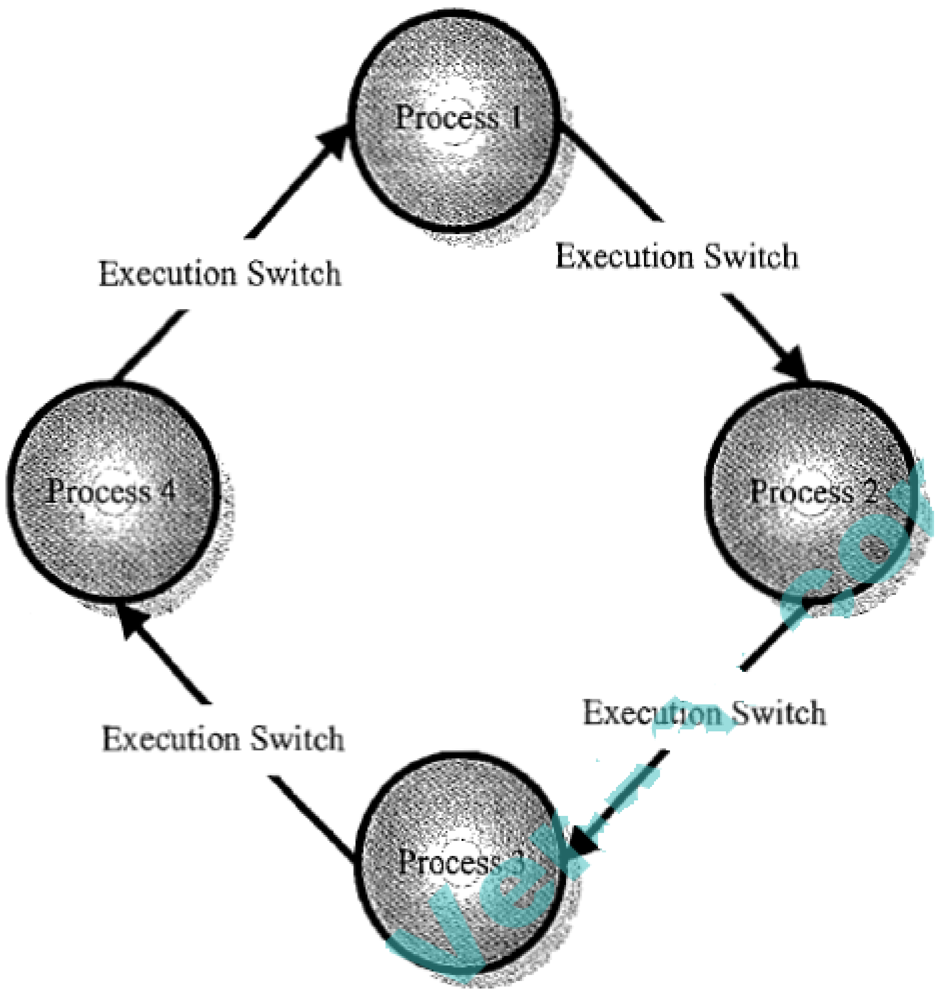


Figure: Round Robin Scheduling

Classification of Scheduling

Example:

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

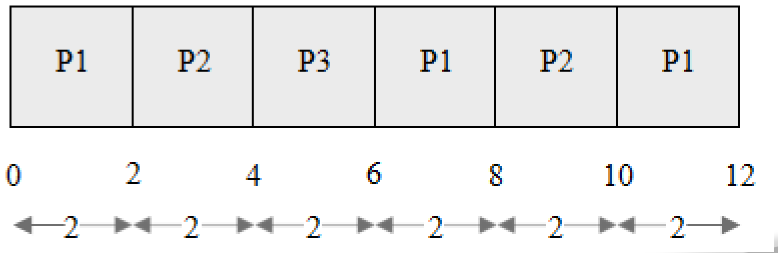
Classification of Scheduling

Solution:

The scheduler sorts the '*Ready*' queue based on the FCFS policy and picks up the first process P1 from the 'Ready' queue and executes it for the time slice 2ms.

When the time slice is expired, P1 is preempted and P2 is scheduled for execution. The Time slice expires after 2ms of execution of P2. Now P2 is preempted and P3 is picked up for execution. P3 completes its execution within the time slice and the scheduler picks P1 again for execution for the next time slice.

This procedure is repeated till all the processes are serviced. The order in which the processes are scheduled for execution is represented as



Classification of Scheduling

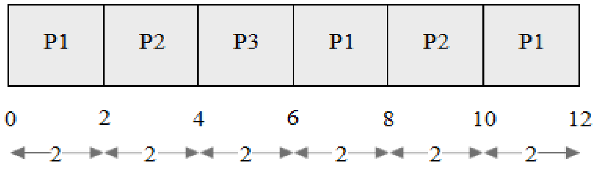
The waiting time for all the processes are given as

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

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

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

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



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

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

= 5.33 milliseconds

Classification of Scheduling

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

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

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

$$\boxed{\text{Average Turn around Time} = \frac{(\text{Turn around Time for all the processes})}{(\text{No. of Processes})}}$$

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

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

= 9.33 milliseconds.

Classification of Scheduling

(iii). 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 task/process in preemptive scheduling is indicated in the same way as that of the mechanism, adopted for non-preemptive multitasking.

Classification of Scheduling

EXAMPLE:

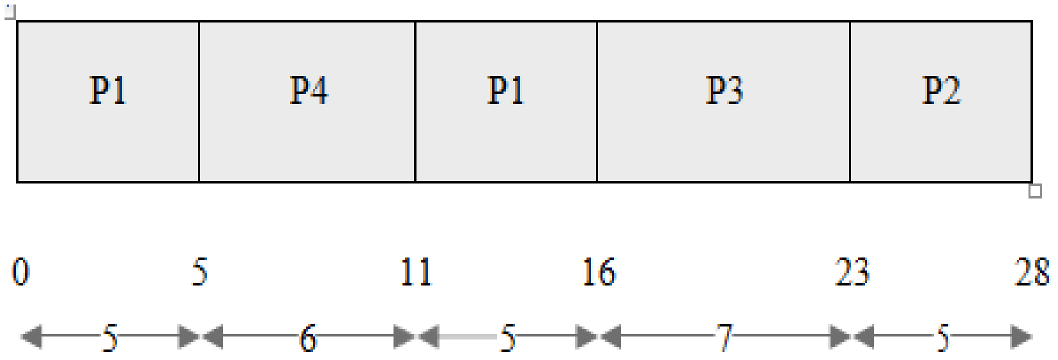
Three processes with process IDs P1, P2, P3 with estimated completion time 10, 5, 7 milliseconds and priorities 1, 3, 2 (0- highest priority, 3 lowest priority) respectively enters the ready queue together. A new process P4 with estimated completion time 6ms and priority 0 enters the ‘Ready’ queue after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

Classification of Scheduling

Solution:

➢At the beginning, there are only three processes (P1, P2 and P3) available in the 'Ready' queue and the scheduler picks up the process with the highest priority (In this example P1 with priority 1) for scheduling.

➢Now process P4 with estimated execution completion time 6ms and priority 0 enters the 'Ready' queue after 5ms of start of execution of P1. The processes are re-scheduled for execution in the following order



Classification of Scheduling

The waiting time for all the processes are given as

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

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

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

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

Classification of Scheduling

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

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

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

= 11.25 milliseconds

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

Turn Around Time (TAT) for P4 = 6ms (Time spent in Ready Queue + Execution Time).

= (5-5) + 6 = 6.

Classification of Scheduling

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

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

$$\boxed{\text{Average Turn around Time} = \frac{(\text{Turn around Time for all the processes})}{(\text{No. of Processes})}}$$

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

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

= 18.25 milliseconds.

Semaphores

Semaphores:

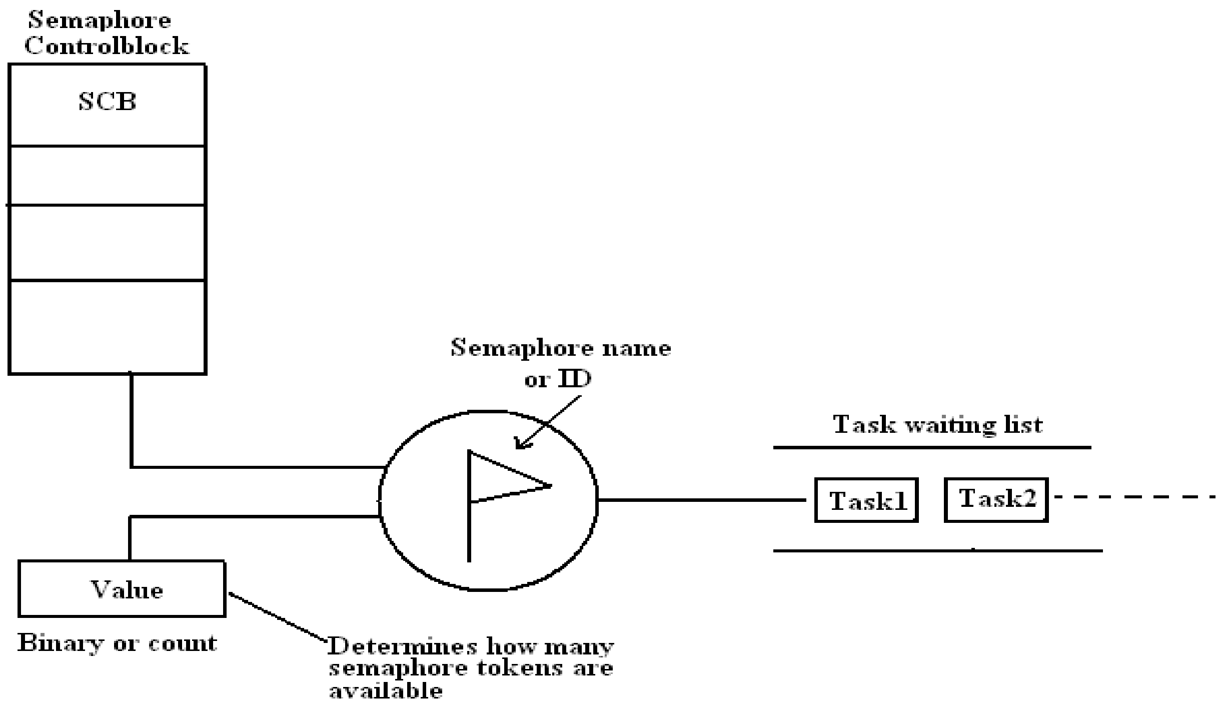
➢A semaphore is nothing but a value or variable or data which can control the allocation of a resource among different tasks in a parallel programming environment. The concept of semaphore was first proposed by the Dutch computer scientist Edsger Dijkstra in the year 1965.

➢So, Semaphores are a useful tool in the prevention of race conditions and deadlocks; however, their use is by no means a guarantee that a program is free from these problems.

➢Semaphores which allow an arbitrary resource count are called counting semaphores, while semaphores which are restricted to the values 0 and 1 (or locked/unlocked, unavailable/available) are called binary semaphores.

Semaphores

The operation of a semaphore can be understood from the following diagram.



Semaphores

Types of Semaphores: There are three types of semaphores

(i).Binary Semaphores,

(ii).Counting Semaphores and

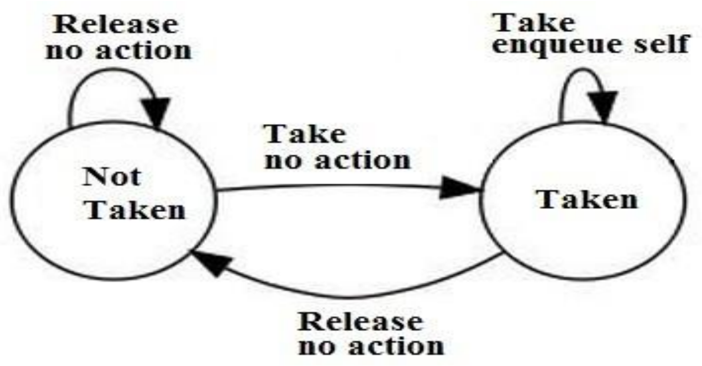
(iii).Mutexes.

A binary semaphore is a synchronization object that can have only two states 0 or 1 i.e not taken and taken.

Take: Taking a binary semaphore brings it in the “taken” state, trying to take a semaphore that is already taken enters the invoking thread into a waiting queue.

Release: Releasing a binary semaphore brings it in the “not taken” state if there are not queued threads. If there are queued threads then a thread is removed from the queue and resumed, the binary semaphore remains in the “taken” state. Releasing a semaphore that is already in its “not taken” state has no effect.

Semaphores



- Binary semaphores have no ownership attribute and can be released by any thread or interrupt handler regardless of who performed the last take operation.
- Because of this binary semaphores are often used to synchronize threads with external events implemented as ISRs, for example waiting for a packet from a network or waiting that a button is pressed. Because there is no ownership concept a binary semaphore object can be created to be either in the “taken” or “not taken” state initially.

Semaphores

Counting Semaphores:

⦿A counting semaphore is a synchronization object that can have an arbitrarily large number of states. The internal state is defined by a signed integer variable, the counter.

The counter value (N) has a precise meaning:

- The Negative value indicates that, there are exactly -N threads queued on the semaphore.
- The Zero value indicates that no waiting threads, a wait operation would put in queue the invoking thread.
- The Positive value indicates that no waiting threads, a wait operation would not put in queue the invoking thread.

⦿Two operations are defined for counting the semaphores.

Semaphores

Two operations are defined for counting the semaphores.

Wait: This operation decreases the semaphore counter .If the result is negative then the invoking thread is queued.

Signal: This operation increases the semaphore counter .If the result is nonnegative then a waiting thread is removed from the queue and resumed.

Counting semaphores have no ownership attribute and can be signaled by any thread or interrupt handler regardless of who performed the last wait operation .Because there is no ownership concept a counting semaphore object can be created with any initial counter value as long it is non-negative.

The counting semaphores are usually used as guards of resources available in a discrete quantity. For example the counter may represent the number of used slots into a circular queue, producer threads would “signal” the semaphores when inserting items in the queue, consumer threads would “wait” for an item to appear in queue, this would ensure that no consumer would be able to fetch an item from the queue if there are no items available.

Semaphores

The OS function calls provided for Semaphore management are

- Create a semaphore
- Delete a semaphore
- Acquire a semaphore
- Release a semaphore
- Query a semaphore

Mutexes

Mutexes :

- Mutex means mutual exclusion. A mutex is a synchronization object that can have only two states. They are not-owned and owned. Two operations are defined for mutexes.
- **Lock:** This operation attempts to take ownership of a mutex, if the mutex is already owned by another thread then the invoking thread is queued.
- **Unlock:** This operation relinquishes ownership of a mutex. If there are queued threads then a thread is removed from the queue and resumed, ownership is implicitly assigned to the thread.
- Mutex is basically a locking mechanism where a process locks a resource using mutex. As long as the process has mutex, no other process can use the same resource. Once process is done with resource, it releases the mutex. Here comes the concept of ownership. Mutex is locked and released by the same process/thread. It cannot happen that mutex is acquired by one process and released by other.

Mutexes

➢So, unlike semaphores, mutexes have owners. A mutex can be unlocked only by the thread that owns it. Most RTOSs implement this protocol in order to address the Priority Inversion problem.

➢Semaphores can also handle mutual exclusion problems but are best used as a communication mechanism between threads or between ISRs and threads.

The OS functions calls provided for mutex management are

- Create a mutex
- Delete a mutex
- Acquire a mutex
- Release a mutex
- Query a mutex
- Wait on a mutex

Hard Real-Time Scheduling Considerations

Hard Real-Time Scheduling Considerations

■ Meet the hard deadline.

■ To some extent, the ability to meet hard deadlines comes from writing fast code.

■ For hard real-time systems, it is important to write subroutines that always execute in the same amount of time or that have a clearly identifiable worst case.

■ Fixed sized buffer is preferred to a general purpose malloc().

■ Tasks that avoid semaphores are preferable, since the worst case performance does not depend upon other tasks who use the same semaphore .

Saving Memory

Saving memory:

- Embedded systems often have limited memory.
- RTOS: each task needs memory space for its stack.
- The first method for determining how much stack space a task needs is to examine your code.
- The second method is experimental. Fill each stack with some recognizable data pattern at startup, run the system for a period of time.

Saving Memory

Program Memory:

⦿Limit the number of functions used

⦿Check the automatic inclusions by your linker: may consider writing own functions

⦿Include only needed functions in RTOS

⦿Consider using assembly language for large routines

Data Memory:

⦿Consider using more static variables instead of stack variables

⦿On 8-bit processors, use char instead of int when possible

Saving Memory

Few ways to save code space:

- Make sure that you are not using two functions to do the same thing.
- Check that your development tools are not sabotaging you.
- Configure your RTOS to contain only those functions that you need.
- Look at the assembly language listings created by your cross-compiler to see if certain of your C statements translate into huge numbers of instructions.

Saving Power

Saving Power:

- The primary method for preserving battery power is to turn off parts or all of the system whenever possible.
- Most embedded-system microprocessors have at least one power-saving mode; many have several.
- The modes have names such as sleep mode, low-power mode, idle mode, standby mode, and so on.
- A very common power-saving mode is one in which the microprocessor stops executing instructions, stops any built-in peripherals, and stops its clock circuit. This saves a lot of power, but the drawback typically is that the only way to start the microprocessor up again is to reset it.

Saving Power

- Static RAM uses very little power when the microprocessor isn't executing instructions.

- Another typical power-saving mode is one in which the microprocessor stops executing instructions but the on-board peripherals continue to operate.

- Another common method for saving power is to turn off the entire system and have the user turn it back on when it is needed.

TASK COMMUNICATION

Task Communication

TASK COMMUNICATION:

In a multitasking system, multiple tasks/processes run concurrently (in pseudo parallelism) and each process may or may not interact between.

Based on the degree of interaction, the processes running on an OS are classified as,

1.Co-operating Processes: In the co-operating interaction model one process requires the inputs from other processes to complete its execution.

2.Competing Processes: The competing processes do not share anything among themselves but they share the system resources. The competing processes compete for the system resources such as file, display device, etc.

Task Communication

Co-operating processes exchanges information and communicate through the following methods.

Co-operation through Sharing: The co-operating process exchange data through some shared resources.

Co-operation through Communication: No data is shared between the processes. But they communicate for synchronization.

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.

Task Communication

1.Shared Memory:

➢Processes share some area of the memory to communicate among them. 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.

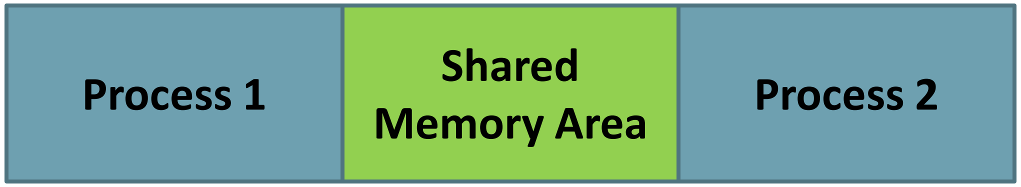


Figure: Concept of shared memory

➢It is same as the real world example where 'Notice Board' is used by corporate to publish the public information among the employees.

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

Task Communication

(i)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 conduit for information flow and has two conceptual ends. It can be unidirectional, allowing information flow in one direction or bidirectional allowing bi-directional information flow.

➢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.

Task Communication

The unidirectional pipe can be visualized as

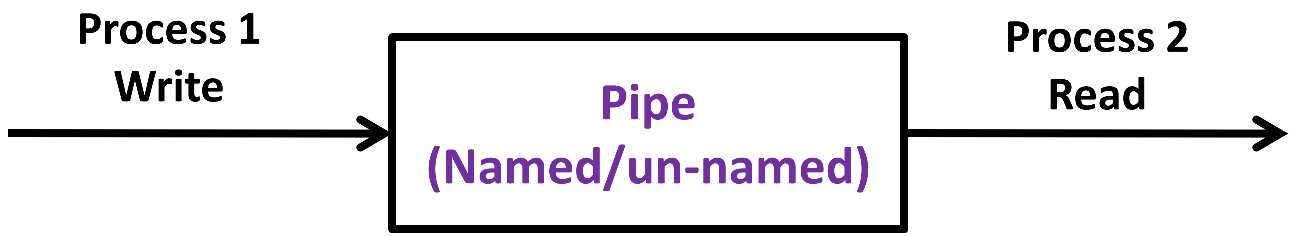


Figure: **Concept of pipe for IPC**

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.

Task Communication

(ii)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 process simultaneously (of course certain synchronization techniques should be applied to prevent inconsistent results).

➢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.

Task Communication

2.Message Passing:

- ➢Message passing-is an (a)synchronous 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 synchronization overheads compared to shared memory.

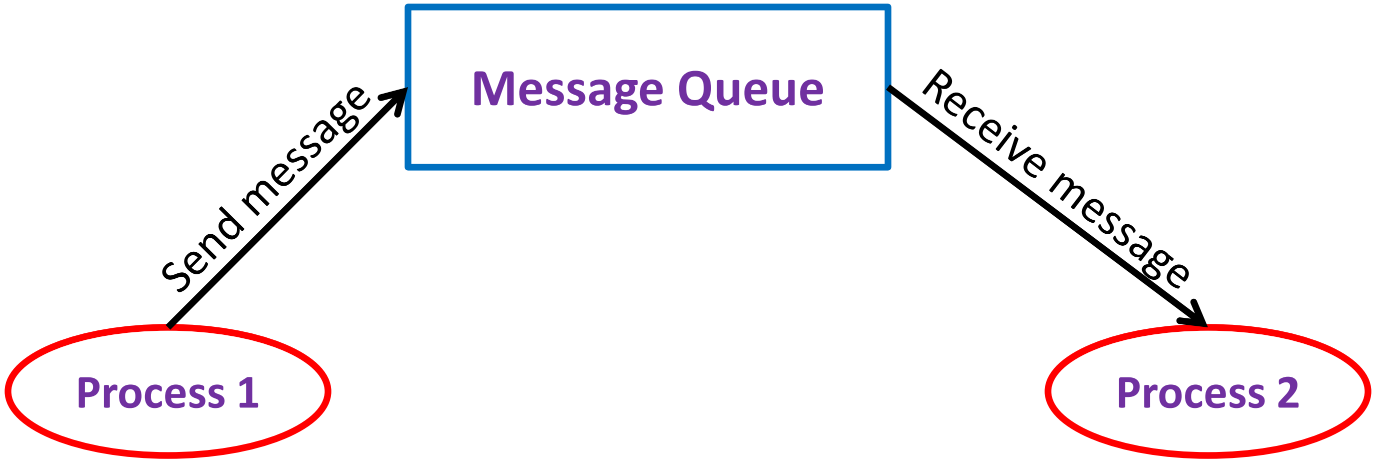
- ➢Based on the message passing operation between the processes, message passing is classified into:

  i. **Message Queue.**
  ii. **Mailbox.**
  iii. **Signaling.**

Task Communication

(i).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.

Task Communication

➢A thread which wants to communicate with another 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 thread and then posts the message to the message queue of the corresponding thread.

➢For posting a message to a thread's message queue, the kernel fills a message structure MSG and copies it to the message queue of the thread.

➢ The message structure MSG contains the handle of the process/thread for which the message is intended, the message parameters, the time at which the message is posted, etc.

Task Communication

A thread can simply post a message to another thread and can continue its operation or it may wait for a response from the thread to which the message is posted.

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.

Task Communication

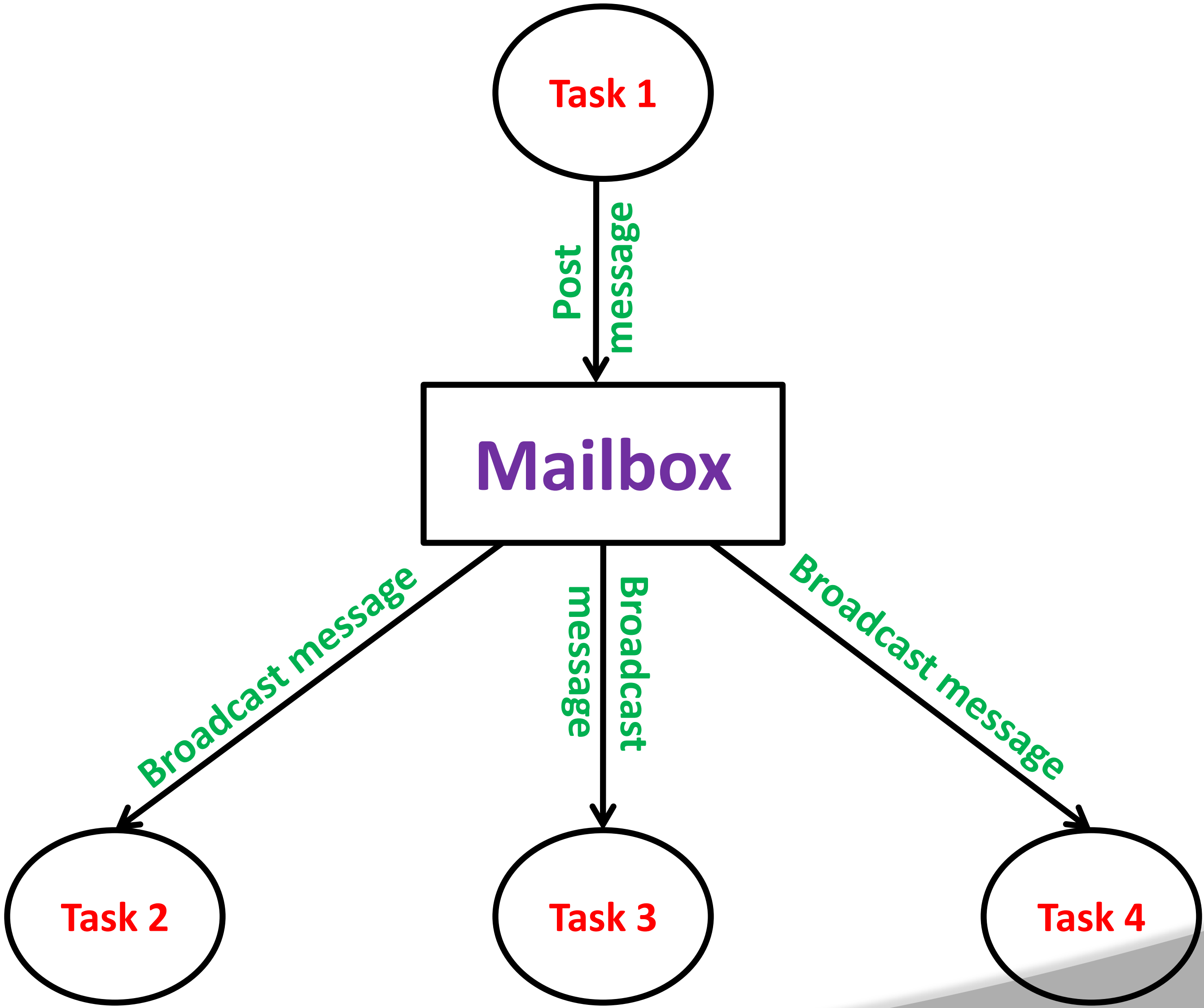
(ii).Mailbox:

➢Mailbox is an alternate form of **'Message queues'** 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.

Task Communication



Task Communication

- The mailbox creation, subscription, message reading and writing are achieved through OS kernel provided API calls. Mailbox and message queues are same in functionality. The only difference is in the number of messages supported by them. Both of them are used for passing data in the form of message(s) from a task to another task(s).

- Mailbox is used for exchanging a single, message between two tasks or between an Interrupt Service Routine (ISR) and a task.

- Mailbox associates a pointer pointing to the mailbox and a wait list to hold the tasks waiting for a message to appear in the mailbox. The implementation of mailbox is OS kernel dependent. The MicroC/OS-II implements mailbox as a mechanism for inter-task communication.

Task Communication

(iii).Signaling:

➢Signaling is a primitive way of communication between process-es/threads. Signals are used for asynchronous notifications where one process/thread fires a signal, indicating the occurrence of a scenario which the other process(es)/thread(s) is waiting. Signals are not queued and they do not carry any data.

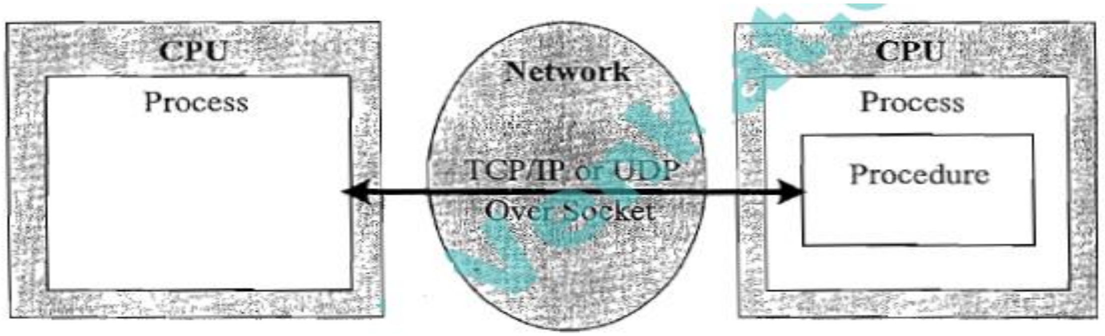
➢The communication mechanisms used in RTX51 Tiny OS is an example for Signaling. The amend signal kernel call under RTX 51 sends a signal from one task to a specified task. Similarly the os_wait kernel call waits for a specified signal. The VxWorks RTOS kernel also implements 'signals' for inter process communication. Whenever a signal occurs it is handled in a signal handler associated with the signal.

Remote Procedure Call (RPC) and Sockets

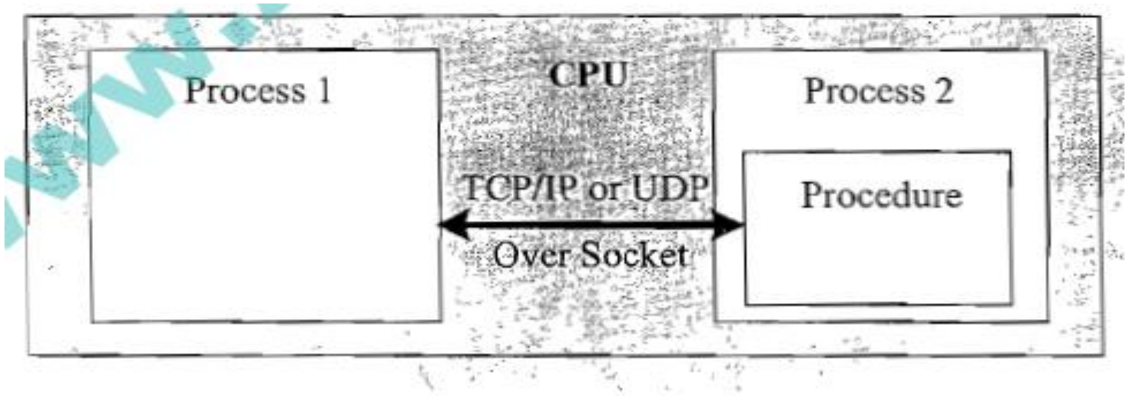
Remote Procedure Call (RPC) and Sockets:

- Remote Procedure Call or RPC is the Inter Process Communication (IPC) 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 (RMI).
- RPC 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.

Remote Procedure Call (RPC) and Sockets



Processes running on different CPUs which are networked



Processes running on same CPU

Remote Procedure Call (RPC) and Sockets

➢It is possible to implement RPC communication with different invocation interfaces. In order to make the RPC communication compatible across all platforms, it should stick on to certain standard formats. Interface Definition Language (IDL) defines the interfaces for RPC.

➢Microsoft Interface Definition Language (MIDL) is the IDL implementation from Microsoft for all Microsoft platforms. 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. The result from the remote procedure is returned back to the caller through mechanisms like callback functions.

Remote Procedure Call (RPC) and Sockets

- On security front, RPC employs authentication mechanisms to protect the systems against vulnerabilities. The client applications (processes)- should authenticate themselves with the server for getting access.

- Authentication mechanisms like IDs, public-key cryptography, etc. are used by the client for authentication. Without authentication, any client can access the remote procedure. This may lead to potential security risks.

- Sockets are used for RPC communication. The 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.

Remote Procedure Call (RPC) and Sockets

Sockets are of different types, namely, Internet sockets (INET), UNIX sockets, etc. The INET socket works on internet communication protocol TCP/IP, UDP (User Datagram Protocol) , etc. are the communication protocols used by INET sockets.

INET sockets are classified into:

1. Stream sockets

2. Datagram sockets

Remote Procedure Call (RPC) and Sockets

Stream sockets are connection-oriented and they use TCP to establish liable connection. On the other hand, **Datagram sockets** rely on UDP for establishing a connection.

The UDP connection is unreliable when compared to TCP. The client-server communication model uses a socket at the client-side and a socket at the server-side.

A port number is assigned to both of these sockets. The client and server should be aware of the port number associated with the socket. In order to start the communication, the client needs to send a connection request to the server at the specified port number.

The client should be aware of the name of the server along with its port number. The server always listens to the specified port number on the network.

Remote Procedure Call (RPC) and Sockets

Upon receiving a connection request from the client, based on the success of authentication, the server grants the connection request and a communication channel is established between the client and server.

The client uses the hostname and port number of the server for sending requests and the server uses the client's name and port number for sending responses.

TASK SYNCHRONIZATION

Task Synchronization

TASK SYNCHRONISATION:

- In a multitasking environment, multiple processes run concurrently (in pseudo parallelism) and share the system resources. Apart from this, each process has its own boundary wall and they communicate with each other with different IPC mechanisms including shared memory and variables.

- Imagine a situation where two processes try to access display hardware connected to the system or two processes try to access a shared memory area where one process tries to write to a memory location when the other process is trying to read from this.

- What could be the result in these scenarios? Obviously unexpected results. How these issues can be addressed? The solution is, make each process aware of the access of a shared resource either directly or indirectly.

Task Synchronization

The act of making processes aware of the access of shared resources by each process to avoid conflicts is known as **`Task/Process Synchronization'.**

Various synchronization issues may arise in a multitasking environment if processes are not synchronized properly.

The following sections describe the major task communication/ synchronization issues observed in multitasking and the commonly adopted synchronization techniques to overcome these issues.

Task Synchronization Issues

Task Communication/Synchronization Issues observed in multitasking :

i. **Racing.**

ii. **Deadlock.**

iii. **The Dining Philosophers Problem.**

iv. **Producer-Consumer/Bounded Buffer Problem.**

v. **Readers-Writers Problem.**

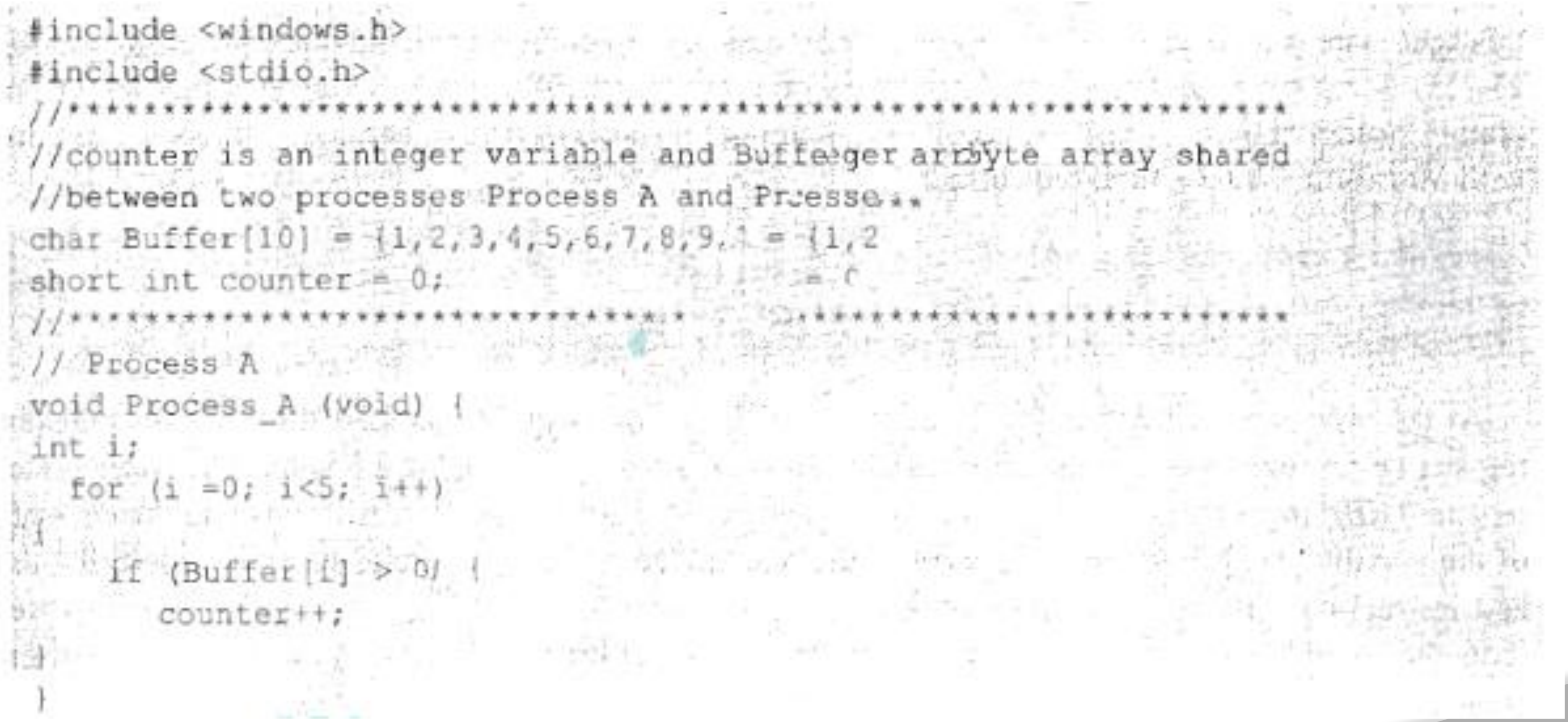
vi. **Priority Inversion.**

Task Synchronization Issues-Racing

Task Communication/Synchronization Issues:

(i). Racing:

Let us have a look at the following piece of code.



```
#include <windows.h>
#include <stdio.h>
//*********************************************************************
//counter is an integer variable and Buffeeger arrbyte array shared
//between two processes Process A and Prcesse..
char Buffer[10] = {1,2,3,4,5,6,7,8,9.1 = {1,2
short int counter = 0;
//********************************  ..*************************
// Process A
void Process_A (void) {
int i;
  for (i =0; i<5; i++)
{
    if (Buffer[i] > 0) {
       counter++;
}
 }
```

Task Synchronization Issues-Racing

```
//****************************************************************
// Process B
void Process_B(void) {
int j;
   for (j =5; j<10; j++)
{
      if (Buffer[j] > 0)
         counter++;
}
}
//****************************************************************
//Main Thread.
int main() {
         DWORD id;

      CreateThread(NULL,0,
                     (LPTHREAD_START_ROUTINE)Process_A,
                     (LPVOID)0,0,&id);
      CreateThread(NULL,0,
                     (LPTHREAD_START_ROUTINE)Process_B,
                     (LPVOID)0,0,&id);
      Sleep(100000);
      return 0;
};
```

Task Synchronization Issues-Racing

- From a programmer perspective, the value of the counter will be 10 at the end of the execution of processes A & B. But 'it need not be always' in a real-world execution of this piece of code under a multitasking kernel.

- The results depend on the process scheduling policies adopted by the OS kernel. The program statement counter++; looks like a single statement from a high-level programming language (`C' language) perspective.

- The low-level implementation of this statement is dependent on the underlying processor instruction set and the (cross) compiler in use. The low-level implementation of the high-level program statement counter++; under Windows XP operating system running on an Intel Centrino Duo processor is given below.

Task Synchronization Issues-Racing

```
mov eax, dword ptr [ebp-4]      ; Load counter in Accumulator
add eax,1                       ; Increment Accumulator by 1
mov dword ptr [ebp-4], eax      ; Store counter with Accumulator
```

➢At the processor instruction level, the value of the variable counter is loaded to the Accumulator register (EAX register). The memory variable counter is represented using a pointer.

➢The base pointer register (EBP register) is used for pointing to the memory variable counter. After loading the contents of the variable-counter to the Accumulator, the Accumulator content is incremented by one using the add instruction.

➢Finally the content of Accumulator is loaded to the memory location which represents the variable counter. Both the processes Process A and Process B contain the program statement counter++; Translating this into the machine instruction.

Task Synchronization Issues-Racing

Process A	Process A
mov eax, dword ptr [ebp-4]	mov eax, dword ptr [ebp-4]
add eax,1	add eax,1
mov dword ptr [ebp-4], eax	mov dword ptr [ebp-4], eax

- 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'.***

Task Synchronization Issues-Racing

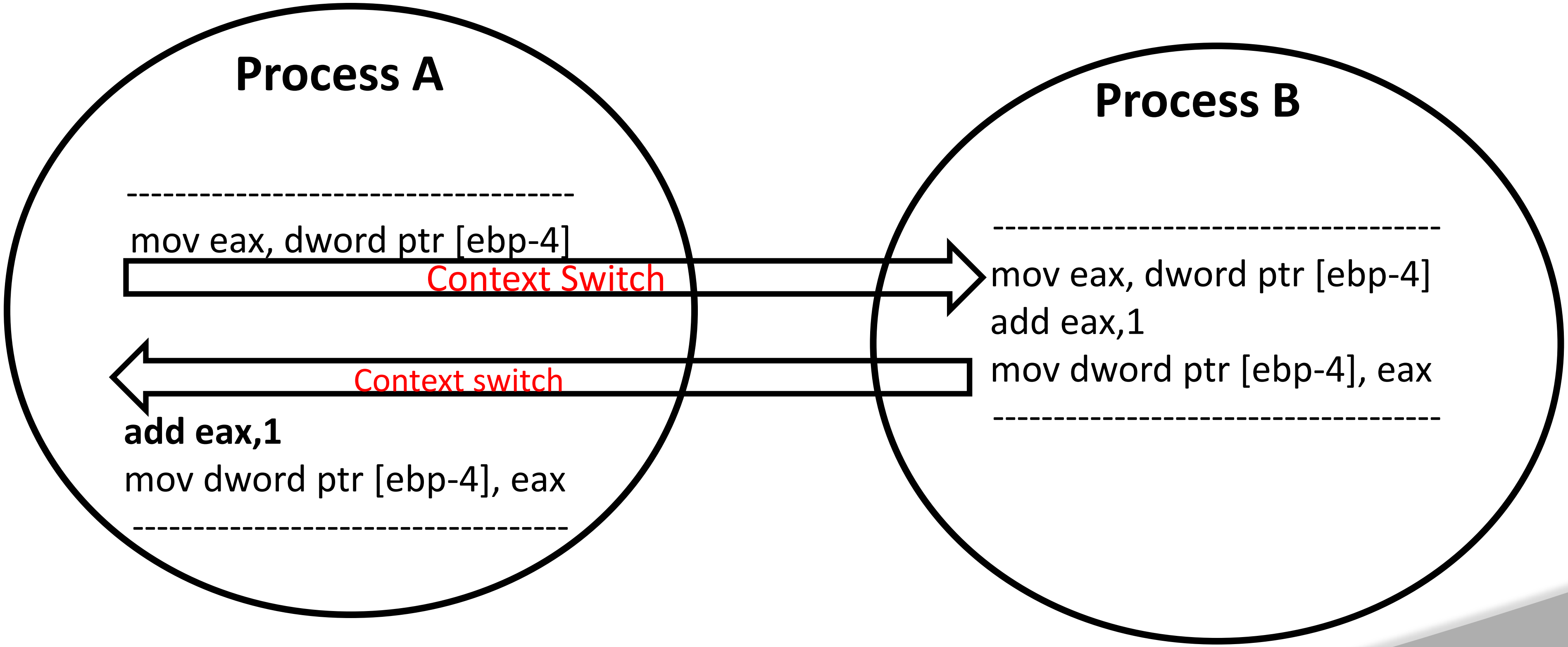


Figure: Race condition

Task Synchronization Issues-Racing

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 non-atomic Operation on variables. This issue wouldn't have been occurred if the underlying actions corresponding to the program statement counter++; is finished in a single CPU execution cycle.

The best way to avoid this situation is make the access and modification of shared variables mutually exclusive; meaning when one process accesses a shared variable, prevent the other processes from accessing it.

To summarize, **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.

Task Synchronization Issues-Deadlock

(ii).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 get of deadlocked processes.

A situation very similar to our traffic jam issues in a junction.

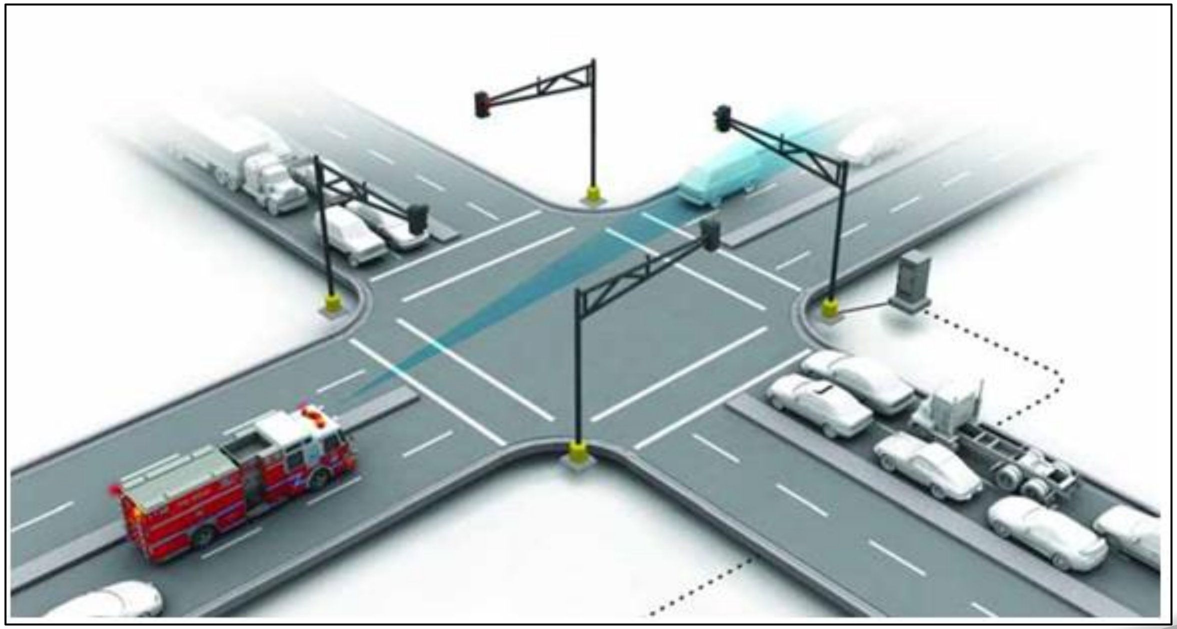


Figure: Deadlock Visualization

Task Synchronization Issues-Deadlock

- 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.

- **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 processes will be able to access the resources held by other processes since they are locked by the respective processes.

Task Synchronization Issues-Deadlock

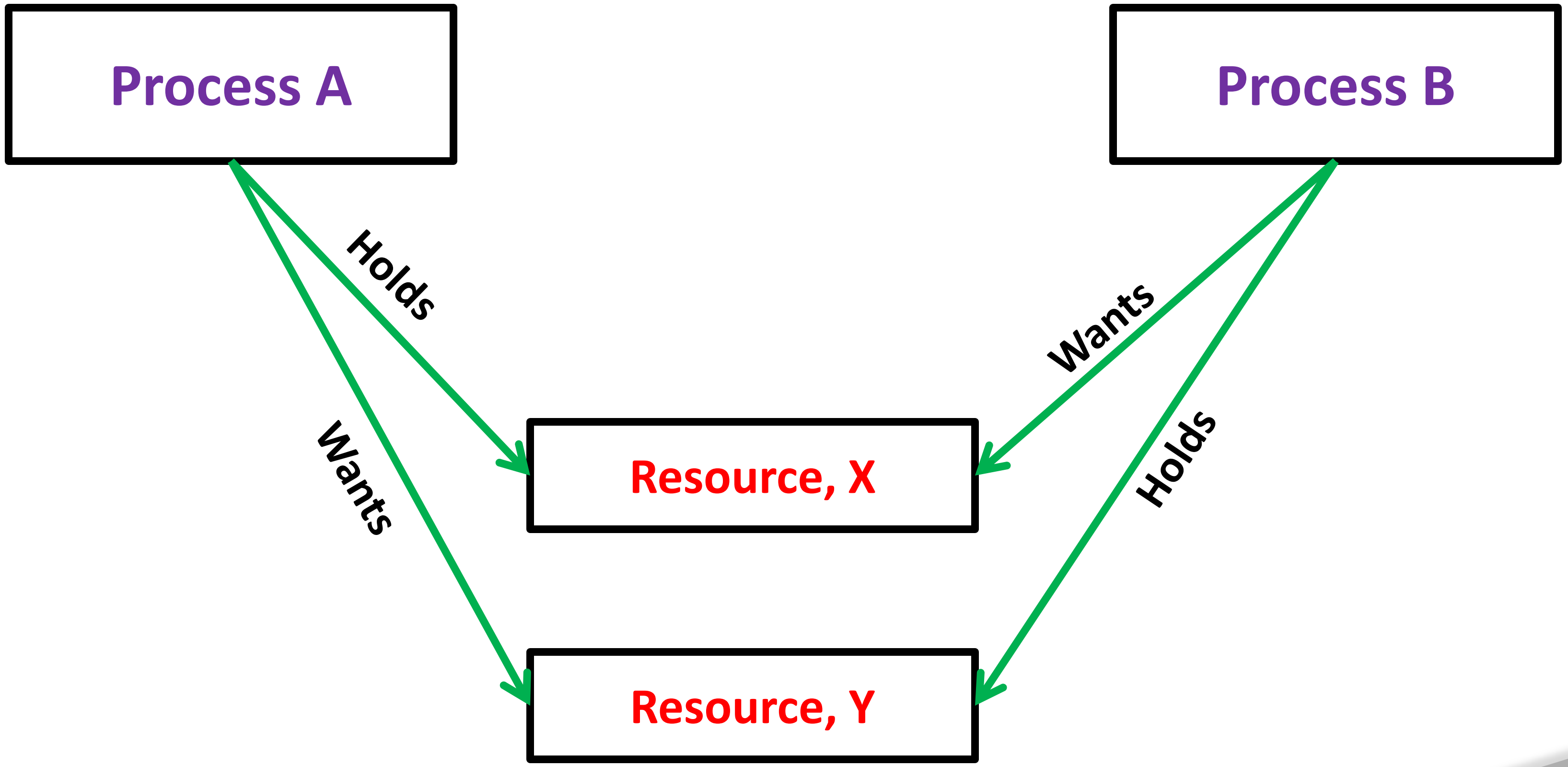


Figure: Scenarios leading to deadlock

Task Synchronization Issues-Deadlock

The different conditions favoring a deadlock situation are listed below.

Mutual Exclusion: The criteria that only one process can hold 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 Walt: 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 Preemption: 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.

Task Synchronization Issues-Deadlock

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 P0, P1, Pn with P0 is waiting for a resource held by P1 and P1 is waiting for a resource held P0, Pn is waiting for a resource held by P0 and P0 is waiting for a resource held by Pn and so on... This forms a circular wait queue.

Deadlock Handling: A smart OS may foresee the deadlock condition and will act proactively to avoid such a situation. Now if a deadlock occurred, how the OS responds to it? The reaction to deadlock condition by OS is nonuniform. The OS may adopt any of the following techniques to detect and prevent deadlock conditions.

Task Synchronization Issues-Deadlock

(i).Ignore Deadlocks: Always assume that the system design is deadlock free. This is acceptable for the reason the cost of removing a deadlock is large compared to the chance of happening a deadlock. UNIX is an example for an OS following this principle. A life critical system cannot pretend that it is deadlock free for any reason.

(ii). Detect and Recover: This approach suggests the detection of a deadlock situation and recovery from it. This is similar to the deadlock condition that may arise at a traffic junction.

When the vehicles from different directions compete to cross the junction, deadlock (traffic jam) condition is resulted. Once a deadlock (traffic jam) is happened at the junction, the only solution is to back up the vehicles from one direction and allow the vehicles from opposite direction to cross the junction. If the traffic is too high, lots of vehicles may have to be backed up to resolve the traffic jam. This technique is also known as `back up cars' technique.

Task Synchronization Issues-Deadlock

Operating systems keep a resource graph in their memory. The resource graph is updated on each resource request and release.

Avoid Deadlocks: Deadlock is avoided by the careful resource allocation techniques by the Operating System. It is similar to the traffic light mechanism at junctions to avoid the traffic jams.

Prevent Deadlocks: Prevent the deadlock condition by negating one of the four conditions favoring the deadlock situation.

• Ensure that a process does not hold any other resources when it requests a resource. This can be achieved by implementing the following set of rules/guidelines in allocating resources to processes.

1. A process must request all its required resource and the resources should be allocated before the process begins its execution.

2. Grant resource allocation requests from processes only if the process does not hold a resource currently.

Task Synchronization Issues-Deadlock

Ensure that resource preemption (resource releasing) is possible at operating system level. This can be achieved by implementing the following set of rules/guidelines in resources allocation and releasing.

1. Release all the resources currently held by a process if a request made by the process for a new resource is not able to fulfill immediately.
2. Add the resources which are preempted (released) to a resource list describing the resources which the process requires to complete its execution.
3. Reschedule the process for execution only when the process gets its old resources and the new resource which is requested by the process.

Imposing these criterions may introduce negative impacts like low resource utilization and starvation of processes.

Task Synchronization Issues-Deadlock

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.

The livelock condition is better explained with the real world example, **two people attempting to cross each other in a narrow corridor**.

Both the persons move towards each side of the corridor to allow the opposite person to cross. Since the corridor is narrow, none of them are able to cross each other. Here both of the persons perform some action but still they are unable to achieve their target, cross each other.

Task Synchronization Issues-Deadlock

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 byproduct of preventive measures of deadlock, scheduling policies favoring high priority tasks and tasks with shortest execution time, etc.

Task Synchronization Issues- The Dining Philosophers' Problem

(iii).The Dining Philosophers' Problem:

The *'Dining philosophers 'problem'* is an interesting example for synchronization issues in resource utilization. The terms 'dining', 'philosophers', etc. may sound awkward in the operating system context, but it is the best way to explain technical things abstractly using non-technical terms.

Now coming to the problem definition:

***Five philosophers* (It can be 'n'. The number 5 is taken for illustration) are sitting around a round table, involved in eating and brainstorming.**

At any point of time each philosopher will be in any one of the three states: ***eating, hungry or brainstorming***. (While eating the philosopher is not involved in brainstorming and while brainstorming the philosopher is not involved in eating).

Task Synchronization Issues- The Dining Philosophers' Problem

- For eating, each **philosopher requires 2 forks**.
- There are only 5 forks available on the dining table ('n' for 'n' number of philosophers) and they are arranged in a fashion one fork in between two philosophers.
- The philosopher can only use the forks on his/her immediate left and right that too in the order pickup the left fork first and then the right fork. Analyze the situation and explain the possible outcomes of this scenario.
- **Let's analyze the various scenarios that may occur in this situation.**

Scenario 1: All the philosophers involve in brainstorming together and try to eat together. Each philosopher picks up the left fork and is unable to proceed since two forks are required for eating the spaghetti present in the plate.

- Philosopher 1 thinks that Philosopher 2 sitting to the right of him/her will put the fork down and waits for it. Philosopher 2 thinks that Philosopher 3' sitting to the right of him/her will put the fork down and waits for it, and so on.

Task Synchronization Issues- The Dining Philosophers' Problem

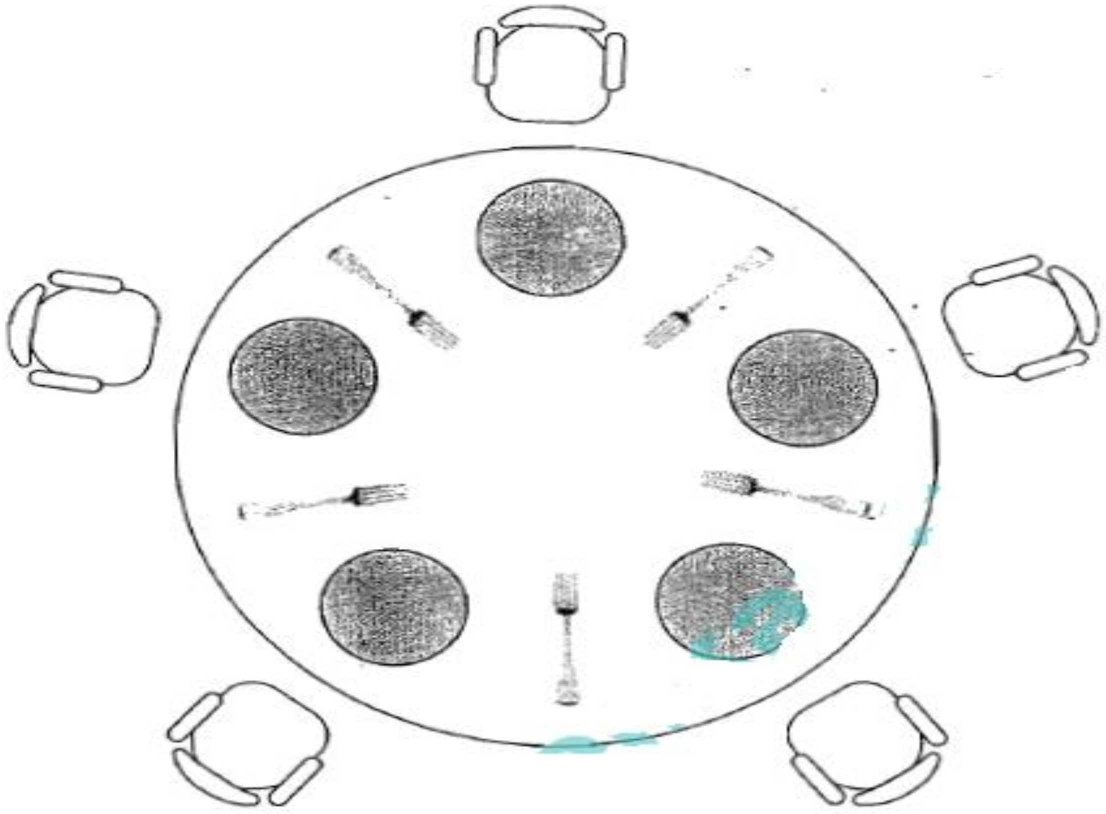


Figure: Visualization of the 'Dining Philosophers' Problem'

Task Synchronization Issues- The Dining Philosophers' Problem

Scenario 2: All the philosophers start brainstorming together. One of the philosophers is hungry and he/ she picks up the left fork.

➢When the philosopher is about to pick up the right fork, the philosopher sitting to his right also become hungry and tries to grab the left fork which is the right fork of his neighboring philosopher who is trying to lift it, resulting in a '***Race condition***'.

Scenario 3: All the philosophers involve in brainstorming together and try to eat together. Each philosopher picks up the left fork and is unable to proceed, since two forks are required for eating the spaghetti present in the plate.

➢Each of them anticipates that the adjacently sitting philosopher will put his/her fork down and waits for a fixed duration grid after this puts the fork down.

Task Synchronization Issues- The Dining Philosophers' Problem

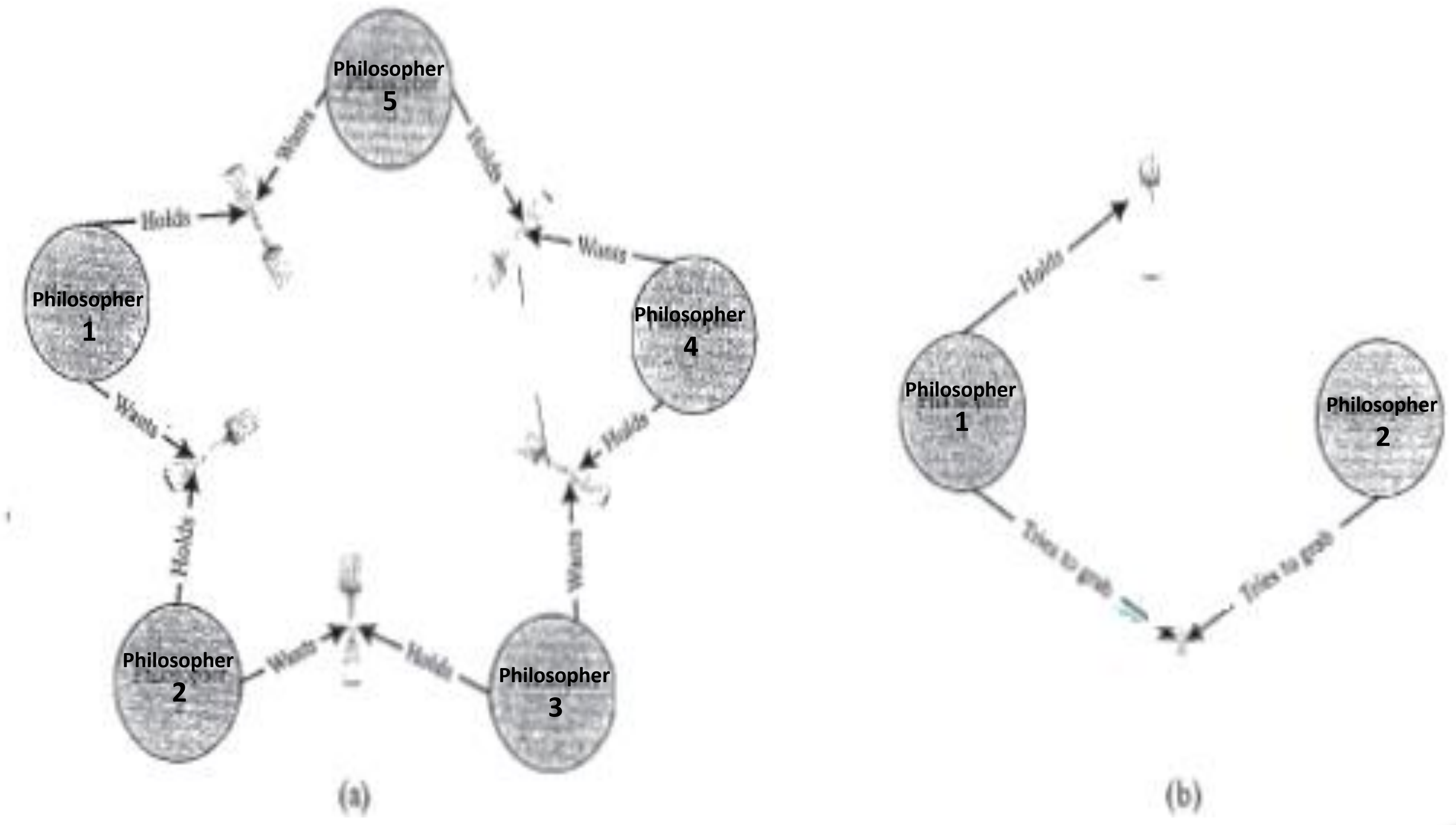


Figure: The 'Real Problems' in the 'Dining Philosophers problem' (a) Starvation and Deadlock (b) Racing

Task Synchronization Issues- The Dining Philosophers' Problem

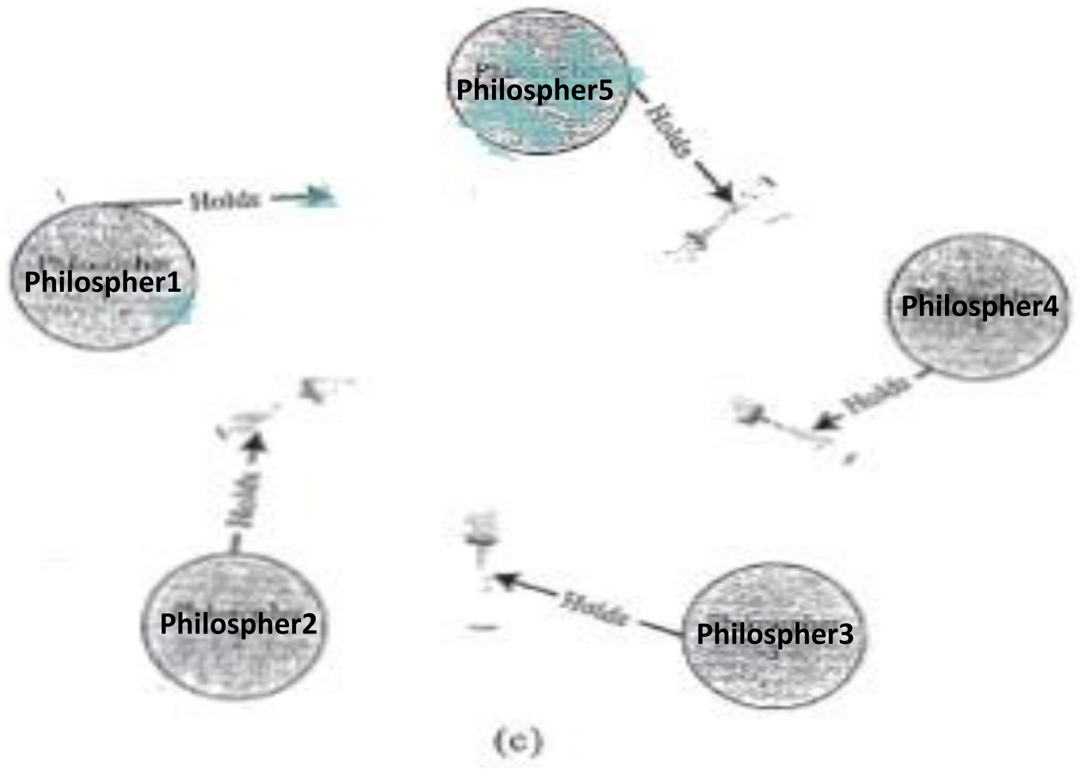


Figure: The 'Real Problems' in the 'Dining Philosophers problem‘
(c) Livelock and Starvation

Task Synchronization Issues- The Dining Philosophers' Problem

- Each of them again tries to lift the fork after a fixed duration of time. Since all philosophers are trying to lift the fork at the same time, none of them will be able to grab two forks.

- This condition leads to ***livelock*** and ***starvation*** of philosophers, where each philosopher tries to do something, but they are unable to make any progress in achieving the target.

Task Synchronization Issues- The Dining Philosophers' Problem

Solution:

We need to find out alternative solutions to avoid the deadlock, livelock, racing and starvation condition that may arise due to the concurrent access of forks by philosophers.

This situation can be handled in many ways by allocating the forks in different allocation techniques including round Robin allocation, FIFO allocation, etc.

But the requirement is that the solution should be optimal, avoiding deadlock and starvation of the philosophers and allowing maximum number of philosophers to eat at a time.

Task Synchronization Issues- The Dining Philosophers' Problem

One solution that we could think of is:

- Imposing rules in accessing the forks by philosophers, like:

- The philosophers should put down the fork he/she already have in hand (left fork) after waiting for a fixed duration for the second fork (right fork) and should wait for a fixed time before making the next attempt.

- This solution works fine to some extent, but, if all the philosophers try to lift the forks at the same time, a livelock situation is resulted.

Task Synchronization Issues- The Dining Philosophers' Problem

- **Another solution which gives maximum concurrency that can be thought of** is each philosopher acquires a semaphore (mutex) before picking up any fork.

- When a philosopher feels hungry he/she checks whether the philosopher sitting to the left and right of him is already using the fork, by checking the state of the associated semaphore.

- If the forks are in use by the neighboring philosophers, the philosopher waits till the forks are available. A philosopher when finished eating puts the forks down and informs the philosophers sitting to his/her left and right, who are hungry (waiting for the forks), by signaling the semaphores associated with the forks.

Producer-Consumer/Bounded Buffer Problem

(iv).Producer-Consumer/Bounded Buffer Problem:

➢Producer-Consumer problem is a common data sharing problem where two processes concurrently access a shared buffer with fixed size.

➢A thread/process which produces data is called 'Producer thread/process' and a thread/process which consumes the data produced by a producer thread/process is known as 'Consumer thread/process'.

➢Imagine a situation where the producer thread keeps on producing data and puts it into the buffer and the consumer thread keeps on consuming the data from the buffer and there is no synchronization between the two.

Producer-Consumer/Bounded Buffer Problem

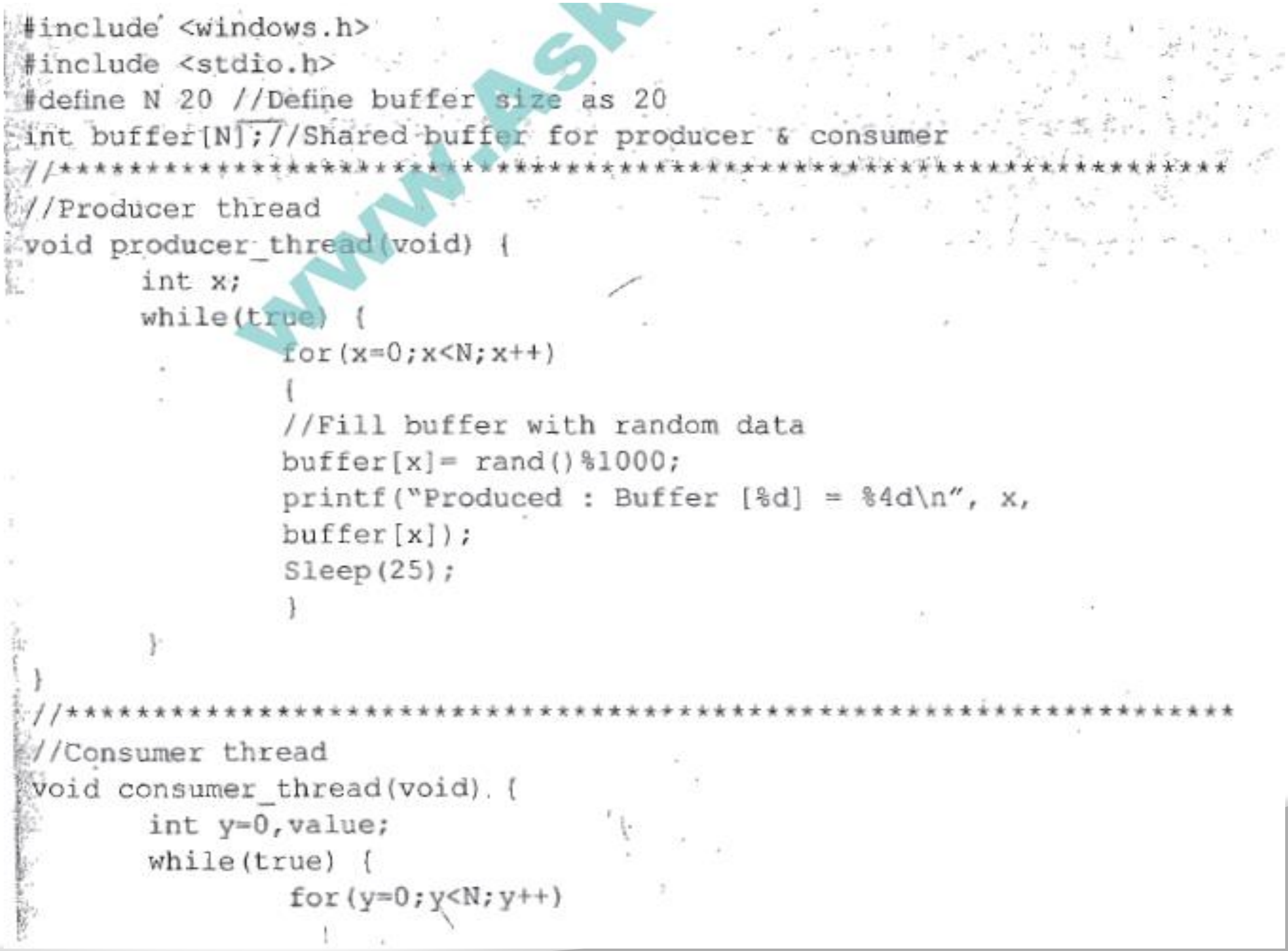
- There may be chances where in which the producer produces data at a faster rate than the rate at which it is consumed by the consumer. This will lead to '**buffer overrun**' where the producer tries to put data to a full buffer.

- If the consumer consumes data at a faster rate than the rate at which it is produced by the producer, it will lead to the situation `**buffer under-run**' in which the consumer tries to read from an empty buffer.

- Both of these conditions will lead to **inaccurate data and data loss**.

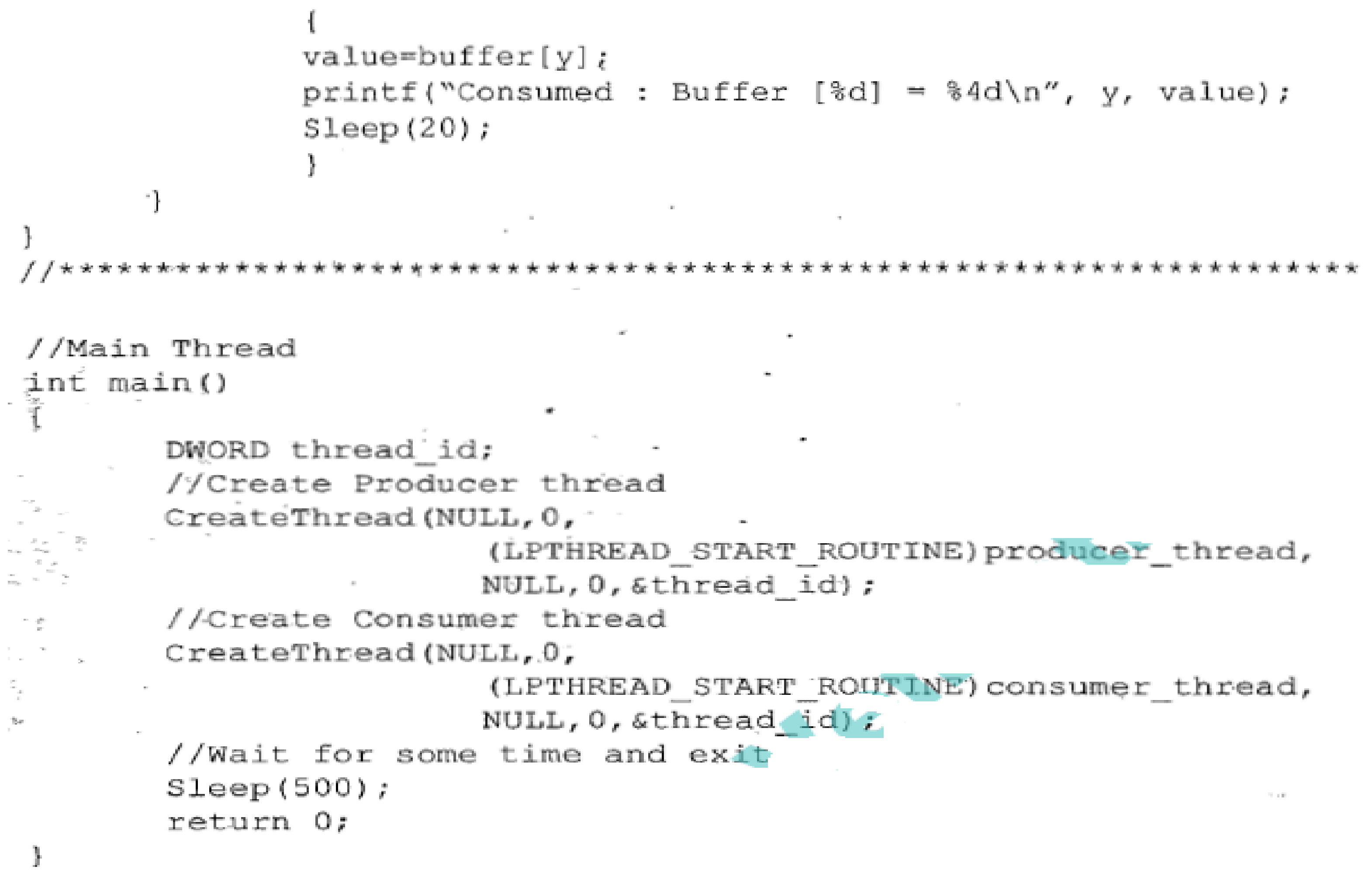
- The following code snippet illustrates the producer-consumer problem

Producer-Consumer/Bounded Buffer Problem



```
#include <windows.h>
#include <stdio.h>
#define N 20 //Define buffer size as 20
int buffer[N];//Shared buffer for producer & consumer
//****************************************************************************
//Producer thread
void producer_thread(void) {
        int x;
        while(true) {
                for(x=0;x<N;x++)
                {
                //Fill buffer with random data
                buffer[x]= rand()%1000;
                printf("Produced : Buffer [%d] = %4d\n", x,
                buffer[x]);
                Sleep(25);
                }
        }
}
//****************************************************************************
//Consumer thread
void consumer_thread(void) {
        int y=0,value;
        while(true) {
                for(y=0;y<N;y++)
```

Producer-Consumer/Bounded Buffer Problem



```
                    {
                    value=buffer[y];
                    printf("Consumed : Buffer [%d] = %4d\n", y, value);
                    Sleep(20);
                    }
            }
}
//*********************************************************************

//Main Thread
int main()
{
        DWORD thread_id;
        //Create Producer thread
        CreateThread(NULL,0,
                        (LPTHREAD_START_ROUTINE)producer_thread,
                        NULL,0,&thread_id);
        //Create Consumer thread
        CreateThread(NULL,0,
                        (LPTHREAD_START_ROUTINE)consumer_thread,
                        NULL,0,&thread_id);
        //Wait for some time and exit
        Sleep(500);
        return 0;
}
```

Producer-Consumer/Bounded Buffer Problem

- Here the 'producer thread' produces random numbers and puts it in a buffer of size 20. If the 'producer thread' fills the buffer fully it re-starts the filling of the buffer from the bottom.

- The 'consumer thread' consumes the data produced by, the 'producer thread'. For consuming the data, the 'consumer thread' reads the buffer which is shared with the 'producer thread'.

- Once the 'consumer thread' consumes all the data, it starts consuming the data from the bottom of the buffer. These two threads run independently and are scheduled for execution based on the scheduling policies adopted by the OS.

Producer-Consumer/Bounded Buffer Problem

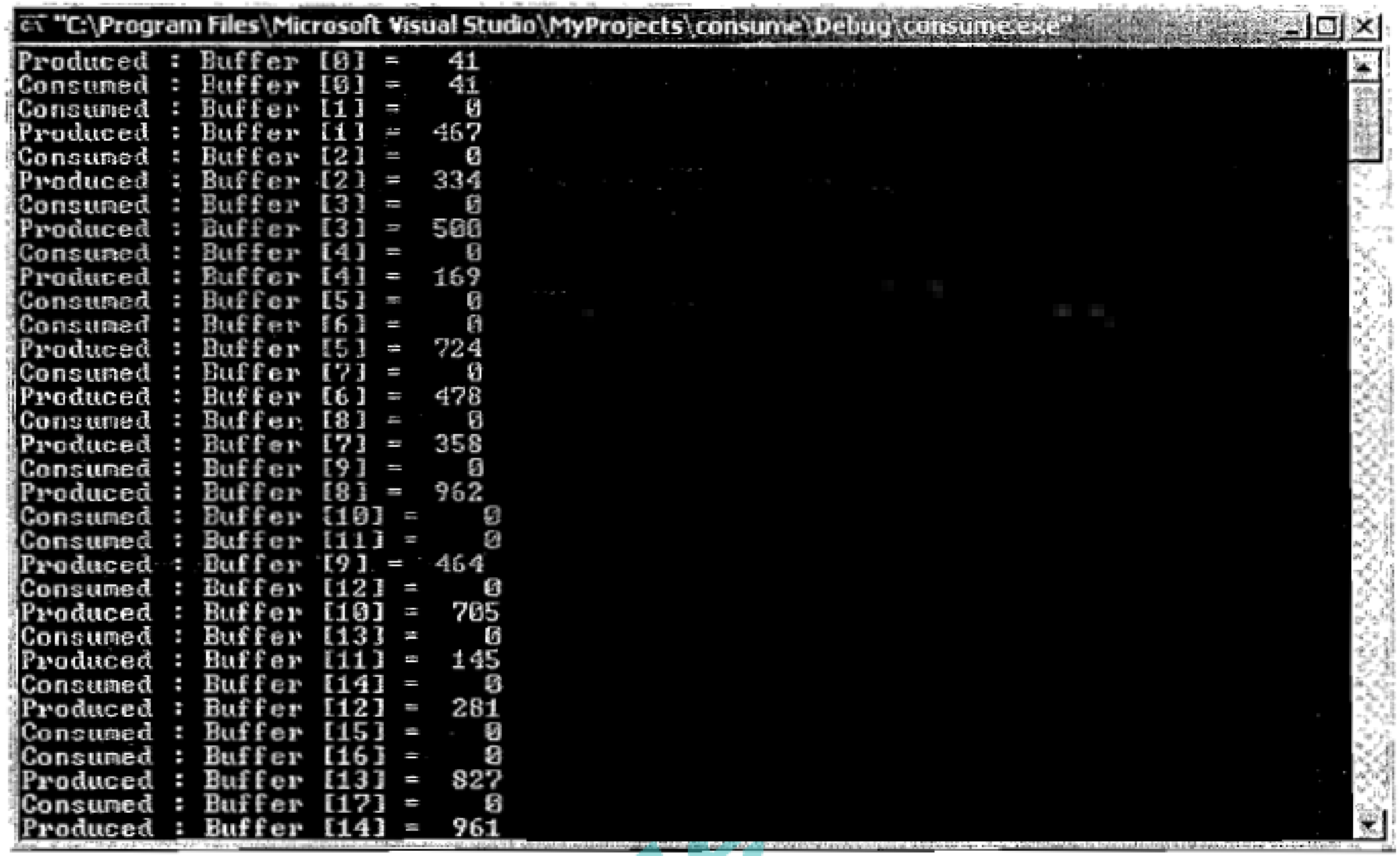
The different situations that may arise based on the scheduling of the 'producer thread' and 'consumer thread' is listed below.

1. '**Producer thread**' is scheduled more frequently than the 'consumer thread': There are chances for overwriting the data in the buffer by the 'producer thread'. This leads to inaccurate data.

2. '**Consumer thread**' is scheduled more frequently than the 'producer thread': There are chances for reading the old data in the buffer again by the 'consumer thread'. This will also lead to inaccurate data.

The output of the above program when executed on a Windows XP machine is shown in Figure. The output shows that the consumer thread runs faster than the producer thread and most often leads to buffer **under-run** and thereby inaccurate data.

Producer-Consumer/Bounded Buffer Problem


"C:\Program Files\Microsoft Visual Studio\MyProjects\consume\Debug\consume.exe"

```
Produced : Buffer [0] =   41
Consumed : Buffer [0] =   41
Consumed : Buffer [1] =    0
Produced : Buffer [1] =  467
Consumed : Buffer [2] =    0
Produced : Buffer [2] =  334
Consumed : Buffer [3] =    0
Produced : Buffer [3] =  500
Consumed : Buffer [4] =    0
Produced : Buffer [4] =  169
Consumed : Buffer [5] =    0
Consumed : Buffer [6] =    0
Produced : Buffer [5] =  724
Consumed : Buffer [7] =    0
Produced : Buffer [6] =  478
Consumed : Buffer [8] =    0
Produced : Buffer [7] =  358
Consumed : Buffer [9] =    0
Produced : Buffer [8] =  962
Consumed : Buffer [10] =     0
Consumed : Buffer [11] =     0
Produced : Buffer [9] =  464
Consumed : Buffer [12] =     0
Produced : Buffer [10] =   705
Consumed : Buffer [13] =     0
Produced : Buffer [11] =   145
Consumed : Buffer [14] =     0
Produced : Buffer [12] =   281
Consumed : Buffer [15] =     0
Consumed : Buffer [16] =     0
Produced : Buffer [13] =   827
Consumed : Buffer [17] =     0
Produced : Buffer [14] =   961
```

Figure: Output of win32 program illustrating producer-consumer problem

Producer-Consumer/Readers-Writers Problem

(v).Readers-Writers Problem:

- Tire Readers-Writers problem is a common issue observed in processes *competing for limited shared resources.*

- The Readers-Writers problem is characterized by multiple processes trying to read and write shared data concurrently.

- A typical real-world example for the Readers-Writers problem is the ***banking system*** where one process tries to read the account information like available balance and the other process tries to update the available balance for that account.

- This may result in inconsistent results. If multiple processes try to read a shared data concurrently it may not create any impacts, whereas when multiple processes try to write and read concurrently it will definitely create inconsistent results. Proper synchronization techniques should be applied to avoid the readers-writers problem.

Task Synchronization Issues- Priority Inversion

(vi).Priority Inversion:

Priority inversion is the by-product of the combination of blocking based (lock based) process synchronization and pre-emptive priority scheduling.

'**Priority inversion**' is the condition in which a high priority task needs to wait for a low priority task to release a resource which is shared between the high priority task and the low priority task, and a medium priority task which doesn't require the shared resource continue its execution by preempting the low priority task.

Priority based preemptive scheduling technique ensures that a high priority task is always executed first, whereas the lock based process synchronization mechanism (like mutex, semaphore, etc.) ensures that a process will not access a shared resource, which is currently in use by another process.

Task Synchronization Issues- Priority Inversion

The synchronization technique is only interested in avoiding conflicts that may arise due to the concurrent access of the shared resources and not at all bothered about the priority of the process which tries to access the shared resource.

In fact, the priority based preemption and lock based synchronization are the two contradicting OS primitives.

Priority inversion is better explained with the following scenario:

Let Process A, Process B and Process C be three processes with priorities High, Medium and Low respectively. Process A and Process C share a variable 'X' and the access to this variable is synchronized through a mutual exclusion mechanism like Binary Semaphore S.

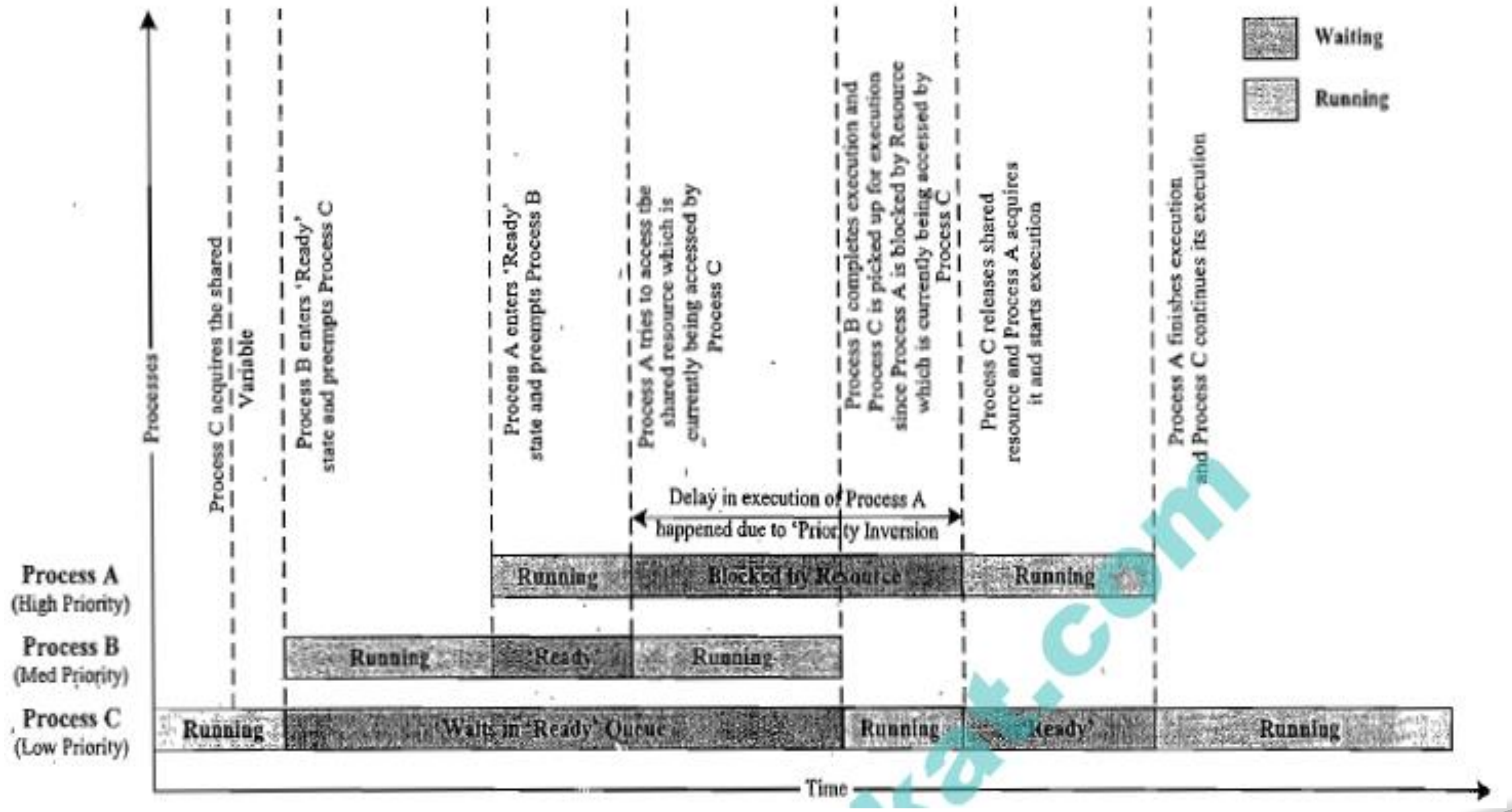
Task Synchronization Issues- Priority Inversion

Imagine a situation where Process C is ready and is picked up for execution by the scheduler and 'Process C' tries to access the shared variable 'X'. 'Process C' acquires the 'Semaphore S' to indicate the other processes that it is accessing the shared variable 'X'.

Immediately after 'Process C' acquires the 'Semaphore S', 'Process B' enters the 'Ready' state. Since 'Process B' is of higher priority compared to 'Process C', 'Process C' is preempted, and 'Process B' starts executing.

Now imagine 'Process A' enters the 'Ready' state at this stage. Since 'Process A' is of higher priority than 'Process B', 'Process B' is preempted, and 'Process A' is scheduled for execution. 'Process A' involves accessing of shared variable 'X' which is currently being accessed by 'Process C'.

Task Synchronization Issues- Priority Inversion


Waiting
Running
Processes
Process C acquires the shared Variable
Process B enters 'Ready' state and preempts Process C
Process A enters 'Ready' state and preempts Process B
Process A tries to access the shared resource which is currently being accessed by Process C
Process B completes execution and Process C is picked up for execution since Process A is blocked by Resource which is currently being accessed by Process C
Process C releases shared resource and Process A acquires it and starts execution
Process A finishes execution and Process C continues its execution
Delay in execution of Process A happened due to 'Priority Inversion
Process A (High Priority)
Running
Blocked by Resource
Running
Process B (Med Priority)
Running
'Ready'
Running
Process C (Low Priority)
Running
Waits in 'Ready' Queue
Running
'Ready'
Running
Time

Task Synchronization Issues- Priority Inversion

Since 'Process C' acquired the semaphore for signaling the access of the shared variable 'X', 'Process A' will not be able to access it.

Thus 'Process A' is put into blocked state (This condition is called Pending on resource). Now 'Process B' gets the CPU and it continues its execution until it relinquishes the CPU voluntarily or enters a wait state or preempted by another high priority task.

The highest priority process 'Process A' has to wait till 'Process C' gets a chance to execute and release the semaphore. This produces unwanted delay in the execution of the high priority task which is supposed to be executed immediately when it was 'Ready'.

Priority inversion *may be sporadic in nature but can lead to potential damages as a result of missing critical deadlines. Literally speaking, priority inversion 'inverts' the priority of a high priority task with that of a low priority task.*

Task Synchronization Issues- Priority Inversion

Priority Inheritance: A low-priority task that is currently accessing (by holding the lock) a shared resource requested by a high-priority task temporarily 'inherits' the priority of that high-priority task, from the moment the high-priority task raises the request.

Boosting the priority of the low priority task to that of the priority of the task which requested the shared resource holding by the low priority task eliminates the preemption of the low priority task by other tasks whose priority are below that of the task requested the shared resource 'and thereby reduces the delay in waiting to get the resource requested by the high priority task.

The priority of the low priority task which is temporarily boosted to high is brought to the original value when it releases the shared resource. Implementation of Priority inheritance workaround in the priority inversion problem discussed for Process A, Process B and Process C example will change the execution sequence as shown in Figure.

Task Synchronization Issues- Priority Inversion

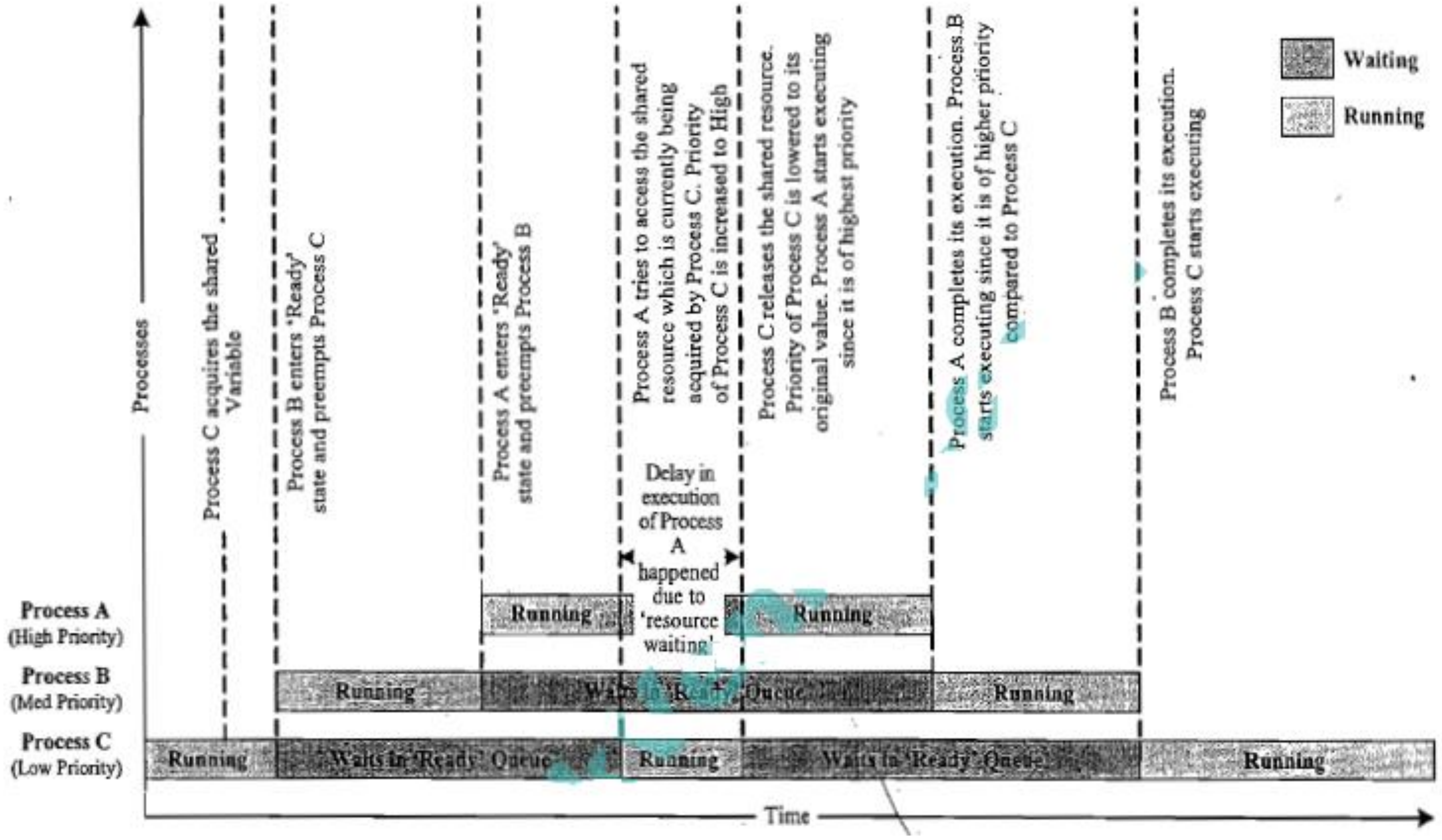

Processes
Process C acquires the shared Variable
Process B enters 'Ready' state and preempts Process C
Process A enters 'Ready' state and preempts Process B
Process A tries to access the shared resource which is currently being acquired by Process C. Priority of Process C is increased to High
Process C releases the shared resource. Priority of Process C is lowered to its original value. Process A starts executing since it is of highest priority
Process A completes its execution. Process B starts executing since it is of higher priority compared to Process C
Process B completes its execution. Process C starts executing
Waiting
Running
Delay in execution of Process A happened due to 'resource waiting'
Process A (High Priority)
Process B (Med Priority)
Process C (Low Priority)
Running
Waits in 'Ready' Queue
Time

Figure: Handling Priority Inversion problem with priority Inheritance

Task Synchronization Issues- Priority Inversion

Priority Inversion: It refers to a situation where the use of a resource by a low-priority thread delays the execution of a high-priority thread when both are contending for the same resources. To address this you have following solutions:

Priority Inheritance Protocol: It temporarily raises the priority of the low-priority thread to match that of the blocked thread until low priority thread releases the resource.

Priority Ceiling Protocol: Here the priority of the low-priority thread is raised immediately when it lock the resource. rather then waiting for a subsequent lock attempt by a high priority thread.

Task Synchronization Techniques

Task Synchronization Techniques:

So far we discussed about the various task/process synchronization issues encountered in multitasking systems due to concurrent resource access. Now let's have a discussion on the various techniques used for synchronization in concurrent access in multitasking. Process/Task synchronization 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. The producer consumer problem is a typical example for processes requiring proper sequence of operation.

Task Synchronization Techniques

In producer consumer problem, accessing the shared buffer by different processes is not the issue; the issue is the writing process should write to the shared buffer only if the buffer is not full and the consumer thread should not read from the buffer if it is empty. Hence proper synchronization should be provided to implement this sequence of operations.

3. Communicating between processes.

Task Synchronization Techniques

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 synchronize 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.

Let us have a look at how mutual exclusion is important in concurrent access.

Consider two processes Process A and Process B running on a multitasking system. Process A is currently running and it enters its critical section. Before Process A completes its operation in the critical section, the scheduler preempts Process A and schedules Process B for execution (Process B is of higher priority compared to Process A).

Task Synchronization Techniques

Process B also contains the access to the critical section which is already in use by Process A. If Process B continues its execution and enters the critical section which is already in use by Process A, a racing condition will be resulted.

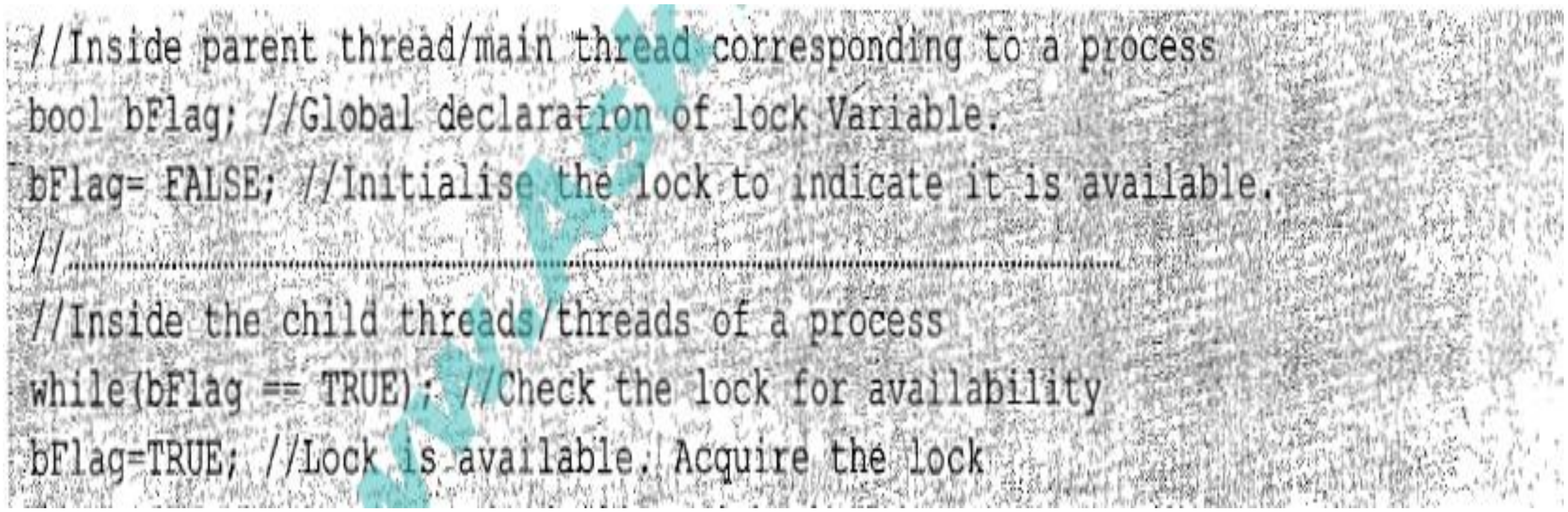
A mutual exclusion policy enforces mutually exclusive access of critical sections. Mutual exclusions can be enforced in different ways. Mutual exclusion blocks a process. Based on the behaviour of the blocked process, mutual exclusion methods can be classified into two categories.

1. Mutual Exclusion through Busy Waiting/Spin Lock.
2. Mutual Exclusion through Sleep & Wakeup.

Mutual Exclusion through Busy Waiting/Spin Lock

1. Mutual Exclusion through Busy Waiting/Spin Lock:

‘Busy waiting' is the simplest method for enforcing mutual exclusion. The following code snippet illustrates how 'Busy waiting' enforces mutual exclusion.



```
//Inside parent thread/main thread corresponding to a process
bool bFlag; //Global declaration of lock Variable.
bFlag= FALSE; //Initialise the lock to indicate it is available.
//...........................................................
//Inside the child threads/threads of a process
while(bFlag == TRUE); //Check the lock for availability
bFlag=TRUE; //Lock is available. Acquire the lock
```

Mutual Exclusion through Busy Waiting/Spin Lock

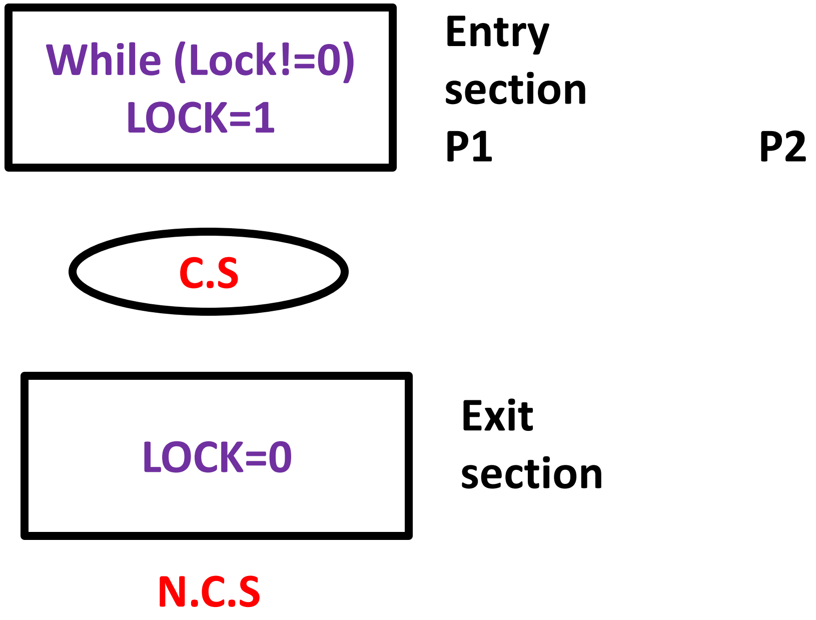
The 'Busy waiting' technique uses a lock variable for implementing mutual exclusion. Each process/ thread checks this lock variable before entering the critical section.

Lock Variable:

•Software mechanism (User mode)

•Busy waiting solution

•More than 2 processes.



Mutual Exclusion through Busy Waiting/Spin Lock

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 synchronization is the non-availability of a single atomic instruction which combines the reading, comparing and setting of the lock variable.

Most often the three different operations related to the locks, viz. the operation of Reading the lock variable, checking its present value, and setting it are achieved with multiple low-level instructions.

The low-level implementation of these operations are dependent on the underlying processor instruction set and the (cross) compiler in use.

Mutual Exclusion through Busy Waiting/Spin Lock

The assembly language instructions reveals that the two high level instructions (while(bFlag==false); and bFlag=true;), corresponding to the operation of reading the lock variable, checking its present value and setting it is implemented in the processor level using six low level instructions.

Imagine a situation where ‘Process 1' read the lock variable and tested it and found that the lock is available and it is about to set the lock for acquiring the critical section. But just before 'Process 1' sets the lock variable, 'Process 2' preempts 'Process 1' and starts executing.

'Process 2' contains a critical section code and it tests the lock variable for its availability. Since 'Process 1' was unable to set the lock variable, its state is still '0' and 'Process 2' sets it and acquires the critical section.

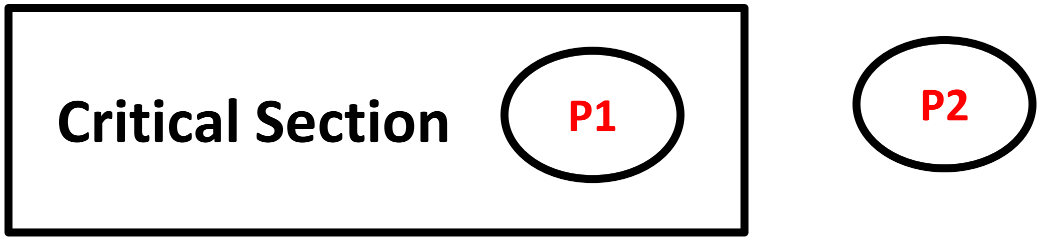
Mutual Exclusion through Busy Waiting/Spin Lock

Now the scheduler preempts 'Process 2' and schedules 'Process 1' before 'Process 2' leaves the critical section. Remember, `Process 1' was preempted at a point just before setting the lock variable (‘Process 1' has already tested the lock variable just before it is preempted and found that the lock is available). Now 'Process 1' sets the lock variable and enters the critical section. It violates the mutual exclusion policy and may produce unpredicted results.

The above issue can be effectively tackled by combining the actions of reading the lock variable, testing its state and setting the lock into a single step. This can be achieved with the combined hardware and software support.

Most of the processors support a single instruction **'Test and Set Lock (TSL)'** for testing and setting the lock variable. The **'Test and Set Lock (TSL)'** instruction call copies the value of the lock variable and sets it to a nonzero value. It should be noted that the implementation and usage of

Mutual Exclusion through Busy Waiting/Spin Lock



-Wasting CPU time

Mutual Exclusion through Busy Waiting/Spin Lock

`Test and Set Lock (TSL)' instruction is processor architecture dependent. The Intel 486 and the above family of processors support the 'Test and Set Lock (TSL)' instruction with a special instruction ***CMPX-CHG***—Compare and Exchange. The usage of CMPXCHG instruction is given below.

CMPXCHG dest, src

This instruction compares the Accumulator (EAX register) with 'dest'. If the Accumulator and 'dest' contents are equal, 'dest' is loaded with src'. If not, the Accumulator is loaded with 'deg'.

Executing this instruction changes the six status bits of the Program Control and Status register EFLAGS. The destination (`dest') can be a register or a memory location. The source (`src') is always a register.

Mutual Exclusion through Sleep &Wakeup

2. Mutual Exclusion through Sleep &Wakeup:

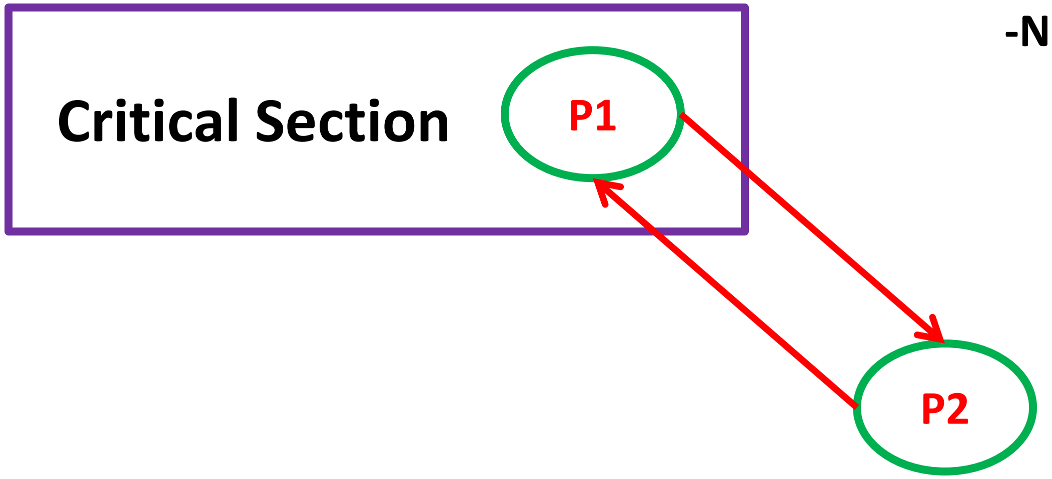
The `Busy waiting' mutual exclusion enforcement mechanism used by processes makes the CPU always busy by checking the lock to see whether they can proceed.

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.

Mutual Exclusion through Sleep &Wakeup

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.



-No Wastage of CPU time

Mutual Exclusion through Sleep &Wakeup

Implementation of this policy is OS kernel dependent.

The following section describes the important techniques for 'Sleep & Wakeup' policy implementation for mutual exclusion by Windows XP/CE OS kernels.

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.

The resources which are shared among a process can be either for exclusive use by a process or for using by a number of processes at a time. The display device of an embedded system is a typical example for the shared resource which needs exclusive access by a process.

Mutual Exclusion through Sleep &Wakeup

The Hard disk (secondary storage) of a system is a typical example for sharing the resource among a limited number of multiple processes. Various processes can access the different sectors of the hard-disk concurrently.

Based on the implementation of the sharing limitation of the shared resource, semaphores are classified into two; namely

'Binary Semaphore' and 'Counting Semaphore'.

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.

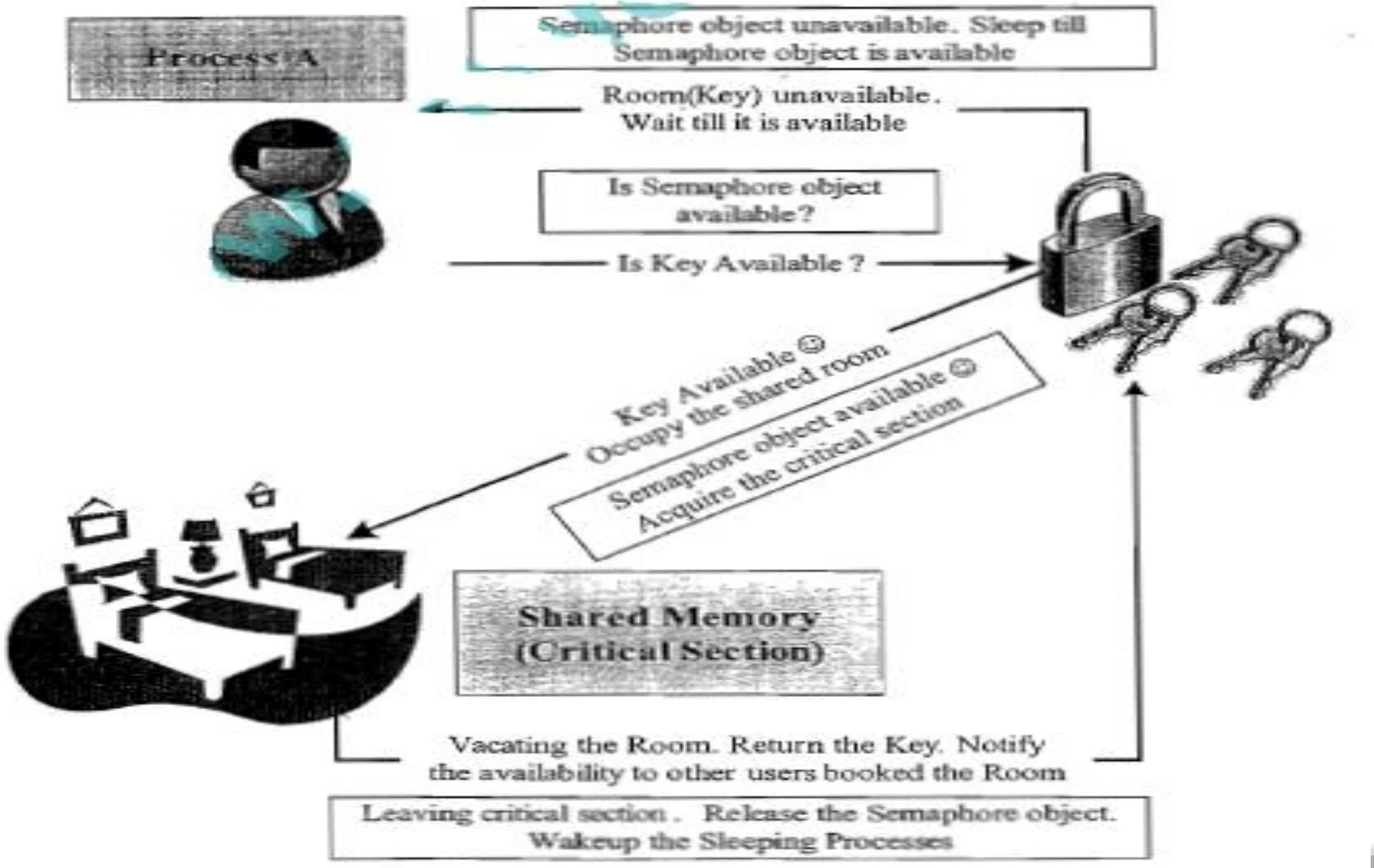
Mutual Exclusion through Sleep &Wakeup

Unlike a binary semaphore, the ***'Counting Semaphore'*** limits the access of resources by a fixed number of processes/threads. It limits the usage of the resource to the maximum value of the count supported by it. The state of the counting semaphore object is set to 'signalled' when the count of the object is greater than zero.

The count associated with a 'Semaphore object' is decremented by one when a process/thread acquires it and the count is incremented by one when a process/thread releases the 'Semaphore object'.

The state of the 'Semaphore object' is set to non-signalled when the semaphore is acquired by the maximum number of processes/threads that the semaphore can support (i.e. when the count associated with the 'Semaphore object' becomes zero).

Mutual Exclusion through Sleep &Wakeup


Process A
Semaphore object unavailable. Sleep till Semaphore object is available
Room(Key) unavailable. Wait till it is available
Is Semaphore object available?
Is Key Available ?
Key Available ☺ Occupy the shared room
Semaphore object available ☺ Acquire the critical section
Shared Memory (Critical Section)
Vacating the Room. Return the Key. Notify the availability to other users booked the Room
Leaving critical section. Release the Semaphore object. Wakeup the Sleeping Processes

The Concept of Counting Semaphore

Mutual Exclusion through Sleep &Wakeup

A real world example for the counting semaphore concept is the dormitory system for accommodation (Fig). 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. Those who are availing the dormitory shares the dorm facilities like TV, telephone, toilet, etc.

If beds are not available currently, the user can register his/her name to get notifications when a slot is available. 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.

Device Driver

Device Driver:

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.

The architecture of the OS kernel will not allow direct device access from the user application. All the device related access should flow through the OS kernel and the OS kernel mutes it to the concerned hardware peripheral.

OS provides interfaces in the form of Application Programming Interfaces (APIs) for accessing the hardware. The device driver abstracts the hardware from user applications. The topology of user applications and hardware interaction in an RTOS based system is depicted in Fig.

Device Driver

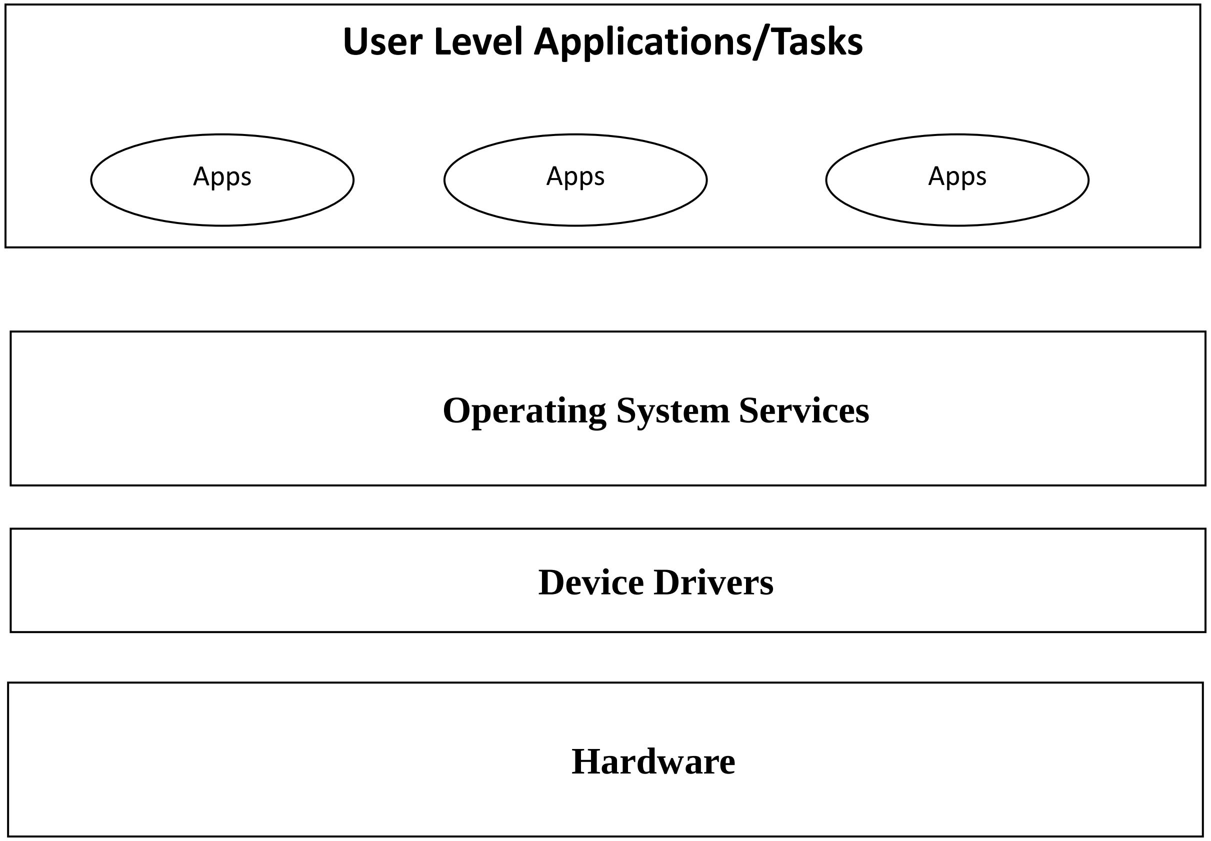


Figure: Role of device driver in Embedded OS based products

Device Driver

Device drivers are responsible for initiating and managing the communication with the hardware peripherals.

They are responsible for

- Establishing the connectivity,
- Initializing the hardware (setting up various registers of the hardware device), and
- Transferring data.

An embedded product may contain different types of hardware components like Wi-Fi module, File systems, Storage device interface, etc. The initialization of these devices and the protocols required for communicating with these devices may be different. All these requirements are implemented in drivers and a single driver will not be able to satisfy all these. Hence each hardware (more specifically each class of hardware) requires a unique driver component.

Device Driver

Certain drivers come as part of the OS kernel and certain drivers need to be installed on the fly.

For example, the program storage memory for an embedded product, say NAND Flash memory requires a NAND Flash driver to read and write data from/to it. This driver should come as part of the OS kernel image.

It contains only the necessary drivers to communicate with the onboard devices (Hardware devices which are part of the platform) and for certain set of devices supporting standard protocols and device class (Say USB Mass storage device or HID devices like Mouse/keyboard).

If an external device, whose driver software is not available with OS kernel image, is connected to the embedded device, the OS prompts the user to install its driver manually.

Device Driver

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. Whenever the device is connected, the OS loads the corresponding driver to memory. When the device is removed, the driver is unloaded from memory. The Operating system maintains a record of the drivers corresponding to each hardware.

Device Driver

However regardless of the OS types, a device driver implements the following:

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

The Device (Hardware) initialisation part of the driver deals with configuring the different registers of the device (target hardware). For example configuring the I/O port line of the processor as Input or output line and setting its associated registers for building a General Purpose IO (GPIO) driver.

Device Driver

The interrupt configuration part deals with configuring the interrupts that needs to be associated with the hardware. In the case of the GPIO driver, if the intention is to generate an interrupt when the Input line is asserted, we need to configure the interrupt associated with the I/O port by modifying its associated registers.

The client interfacing implementation makes use of the Inter Process communication mechanisms supported by the embedded OS for communicating and synchronising with user applications and drivers.

UNIT-IV

EMBEDDED SOFTWARE DEVELOPMENT TOOLS

SYLLABUS:

Host and target machines, linker/locators for embedded software, getting embedded software into the target system; Debugging techniques: Testing on host machine, using laboratory tools, an example system.

HOST AND TARGET MACHINES

Host:

Where the embedded software is developed, compiled, tested, debugged, optimized, and prior to its translation into target device. (Because the host has keyboards, editors, monitors, printers, more memory, etc. for development, while the target may have not of these capabilities for developing the software.)

Target:

After development, the code is cross-compiled, translated – cross-assembled, linked (into target processor instruction set) and ***located*** into the target

HOST AND TARGET MACHINES

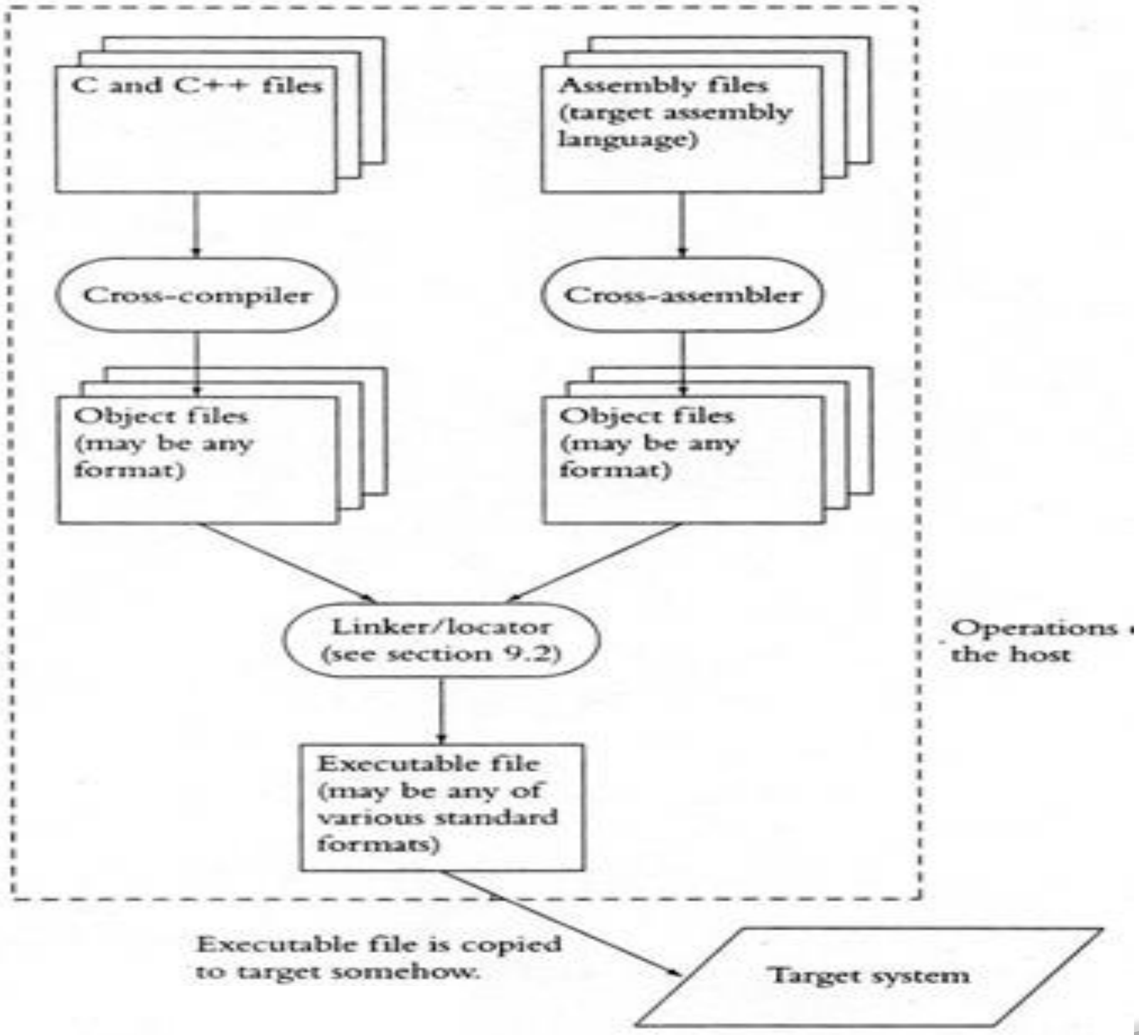
- **Cross-Compilers :**

  - Native tools are good for host, but to port/locate embedded code to target, the host must have a tool-chain that includes a cross-compiler, one which runs on the host but produces code for the target processor
  - Cross-compiling doesn’t guarantee correct target code due to (e.g., differences in word sizes, instruction sizes, variable declarations, library functions)

HOST AND TARGET MACHINES

- **Cross-Assemblers and Tool Chain:**
  - Host uses cross-assembler to assemble code in target’s instruction syntax for the target
  - Tool chain is a collection of compatible, translation tools, which are ‘pipelined’ to produce a complete binary/machine code that can be linked and located into the target processor

HOST AND TARGET MACHINES


C and C++ files
Assembly files (target assembly language)
Cross-compiler
Cross-assembler
Object files (may be any format)
Object files (may be any format)
Linker/locator (see section 9.2)
Executable file (may be any of various standard formats)
Operations on the host
Executable file is copied to target somehow.
Target system

LINKERS AND LOCATORS

Linker/Locators for Embedded Software:

- Native linkers are different from cross-linkers (or locators) that perform additional tasks to ***locate*** embedded binary code into target processors
- Address Resolution –
  - Native Linker: produces host machine code on the hard-drive (in a named file), which the loader loads into RAM, and then schedules (under the OS control) the program to go to the CPU.

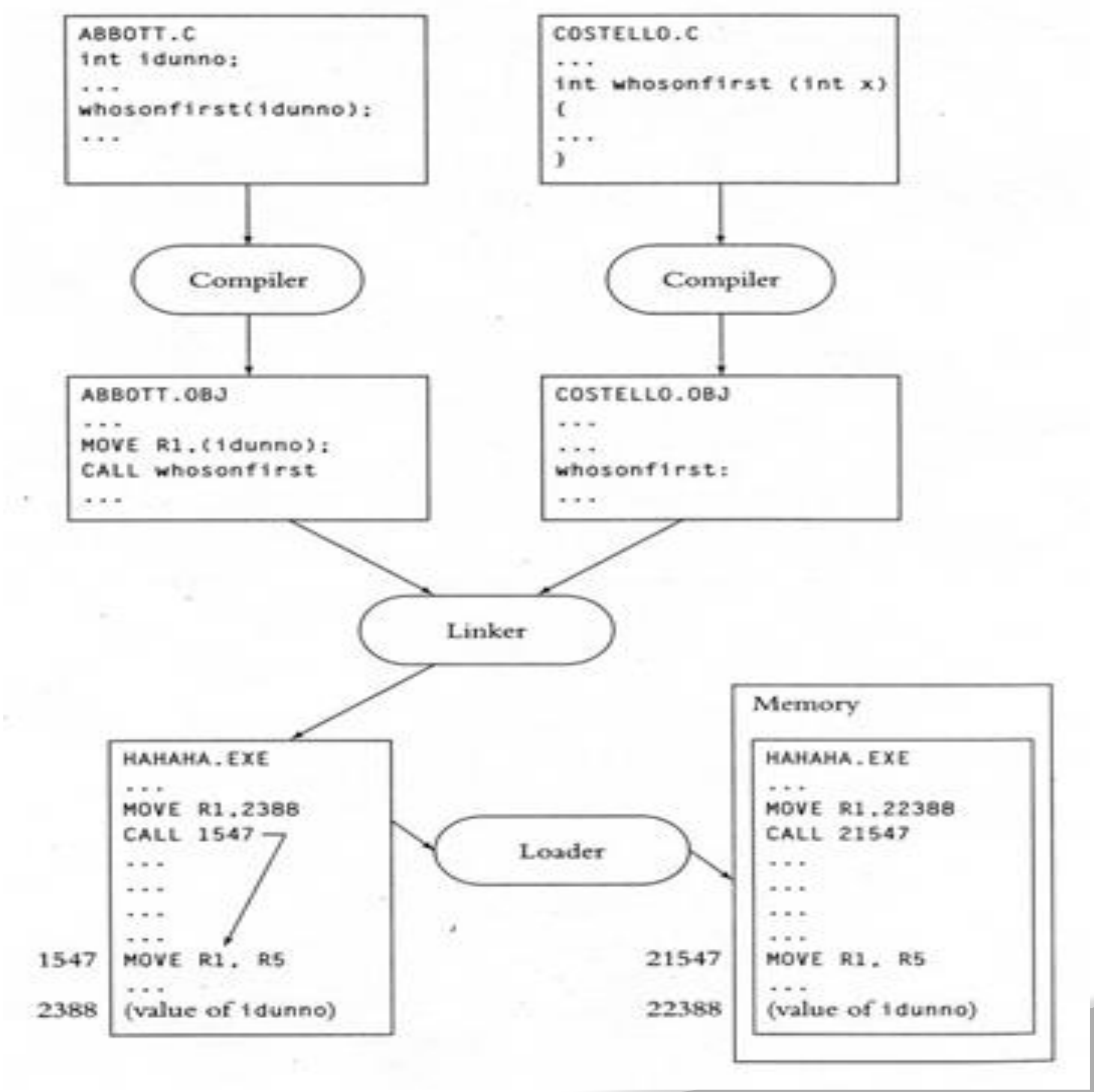
LINKERS AND LOCATORS

Linker/Locators for Embedded Software:

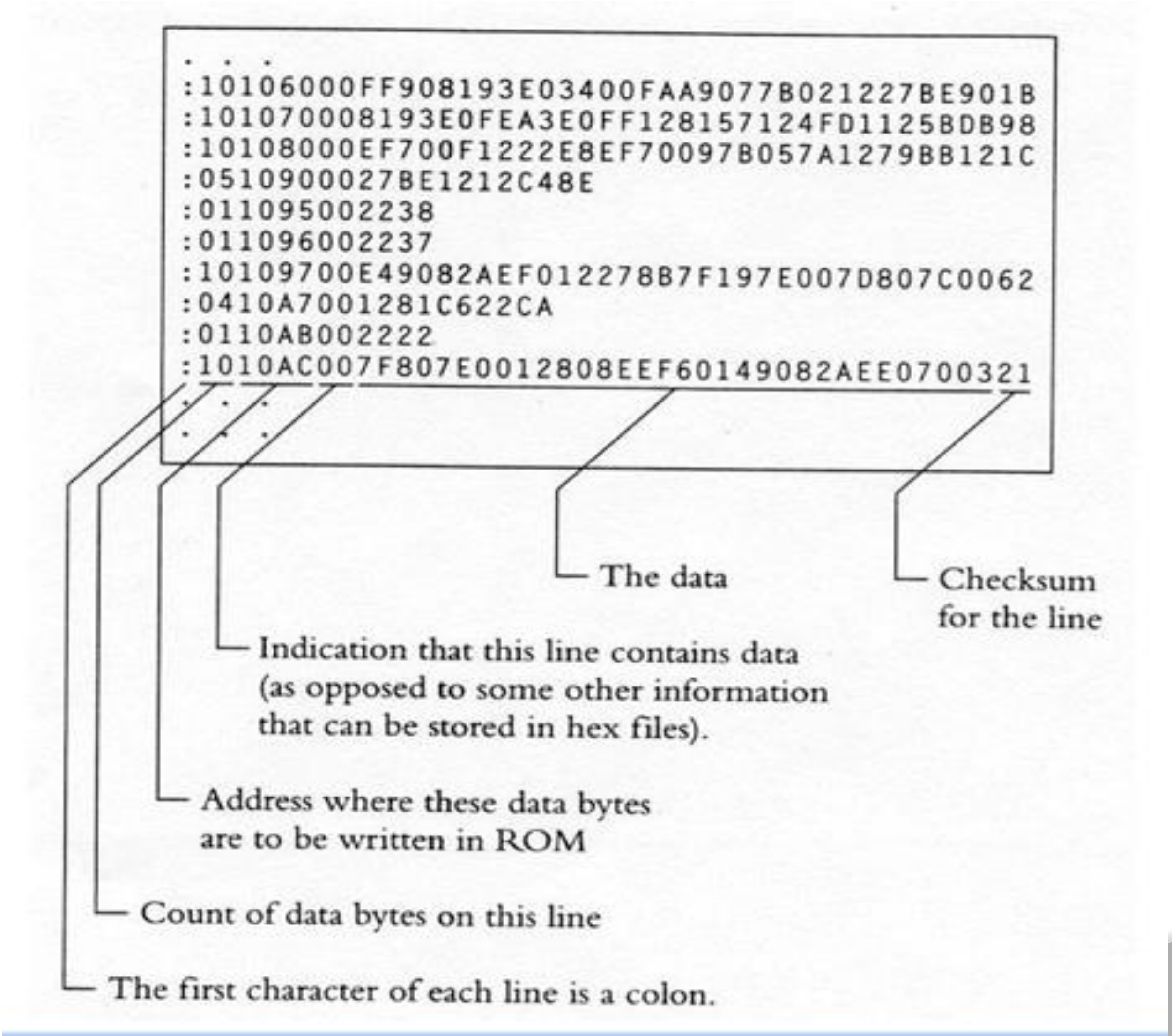
- Function calls, are ordered or organized by the linker. The loader then maps the logical addresses into physical addresses a process called **address resolution**. The loader then loads the code accordingly into RAM . In the process the loader also resolves the addresses for calls to the native OS routines

- Locator: produces target machine code (which the locator glues into the RTOS) and the combined code (called **map**) gets copied into the target ROM. The locator doesn't stay in the target environment, hence all addresses are resolved, guided by locating-tools and directives, prior to running the code.

LINKERS AND LOCATORS



HOST AND TARGET MACHINES



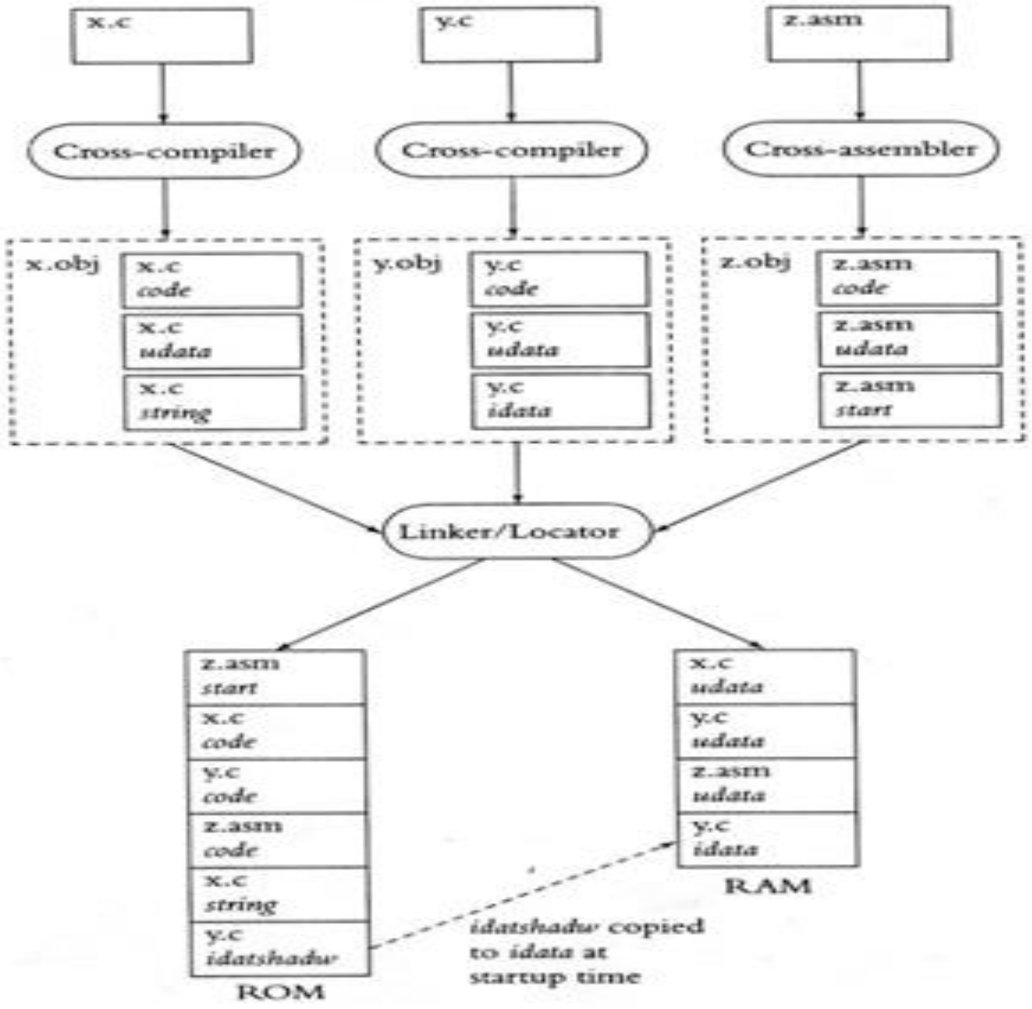
```
.  .  .
:10106000FF908193E03400FAA9077B021227BE901B
:1010700008193E0FEA3E0FF128157124FD1125BDB98
:10108000EF700F1222E8EF70097B057A1279BB121C
:0510900027BE1212C48E
:011095002238
:011096002237
:10109700E49082AEF012278B7F197E007D807C0062
:0410A7001281C622CA
:0110AB002222
:1010AC007F807E0012808EEF60149082AEE0700321
.  .  .
.  .  .
```

The data
Checksum for the line
Indication that this line contains data (as opposed to some other information that can be stored in hex files).
Address where these data bytes are to be written in ROM
Count of data bytes on this line
The first character of each line is a colon.

LINKERS AND LOCATORS

- Locating Program Components – Segments
- Unchanging embedded program (binary code) and constants must be kept in ROM to be remembered even on power-off
- Changing program segments (e.g., variables) must be kept in RAM
- Chain tools separate program parts using **segments** concept
- Chain tools (for embedded systems) also require a ‘start-up’ code to be in a separate segment and ‘located’ at a microprocessor-defined location where the program starts execution
- Some cross-compilers have default or allow programmer to specify segments for program parts, but cross-assemblers have no default behavior and programmer must specify segments for program parts

HOST AND TARGET MACHINES

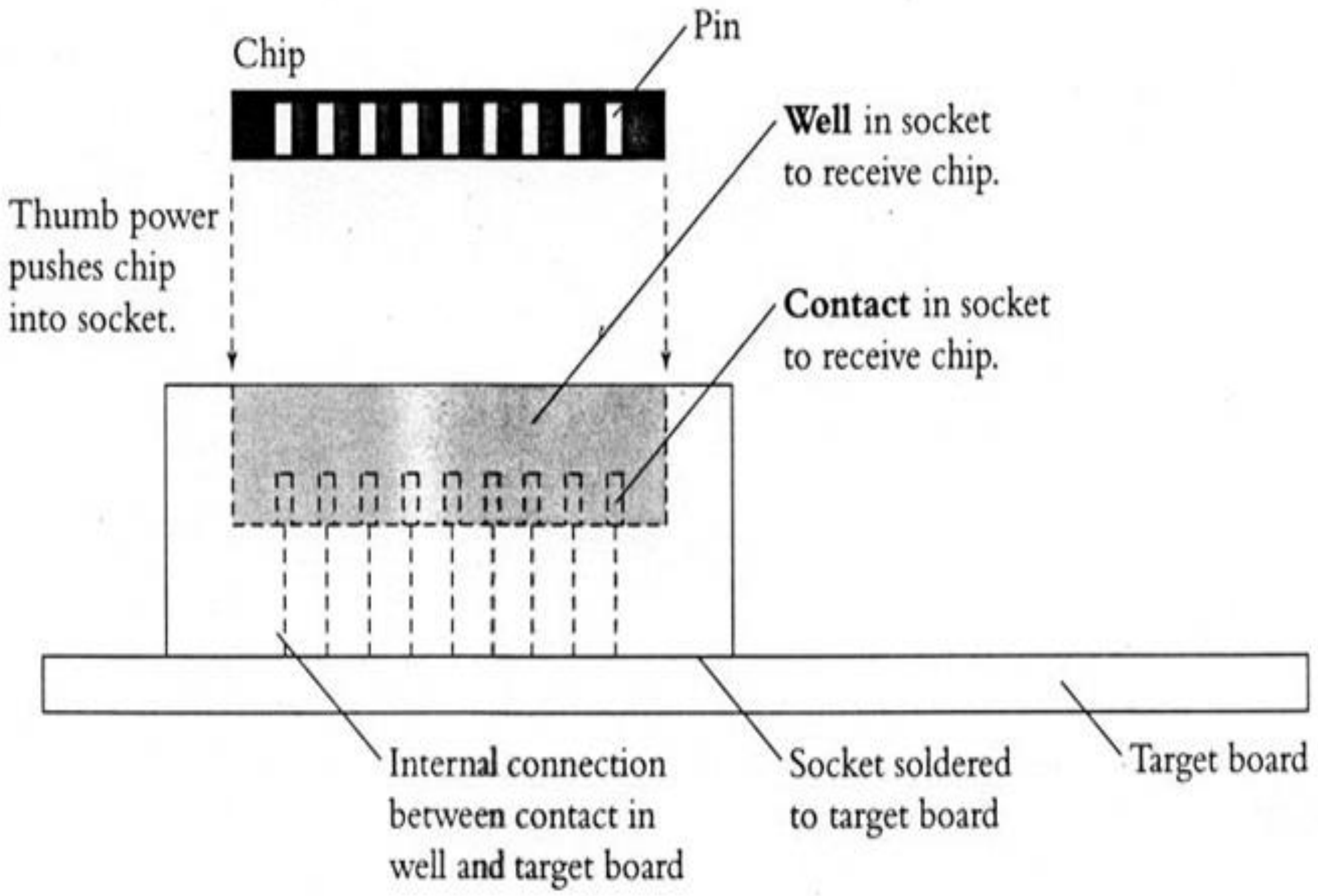


GETTING EMBEDDED SOFTWARE INTO TARGET SYSTEM

Getting Embedded Software into Target System

- Moving maps into ROM or PROM, is to create a ROM using hardware tools or a PROM programmer (for small and changeable software, during debugging)
- If PROM programmer is used (for changing or debugging software), place PROM in a **socket** (which makes it erasable – for EPROM, or removable/replaceable) rather than ‘burnt’ into circuitry
- PROM’s can be pushed into sockets by hand, and pulled using a chip puller
- The PROM programmer must be compatible with the format (syntax/semantics) of the Map

GETTING EMBEDDED SOFTWARE INTO TARGET SYSTEM


Chip
Pin
Well in socket to receive chip.
Thumb power pushes chip into socket.
Contact in socket to receive chip.
Internal connection between contact in well and target board
Socket soldered to target board
Target board

GETTING EMBEDDED SOFTWARE INTO TARGET SYSTE

Getting Embedded Software into Target System – 1

- ROM Emulators – Another approach is using a ROM emulator (hardware) which emulates the target system, has all the ROM circuitry, and a serial or network interface to the host system. The locator loads the Map into the emulator, especially, for debugging purposes.

- Software on the host that loads the Map file into the emulator must understand (be compatible with) the Map's syntax/semantics

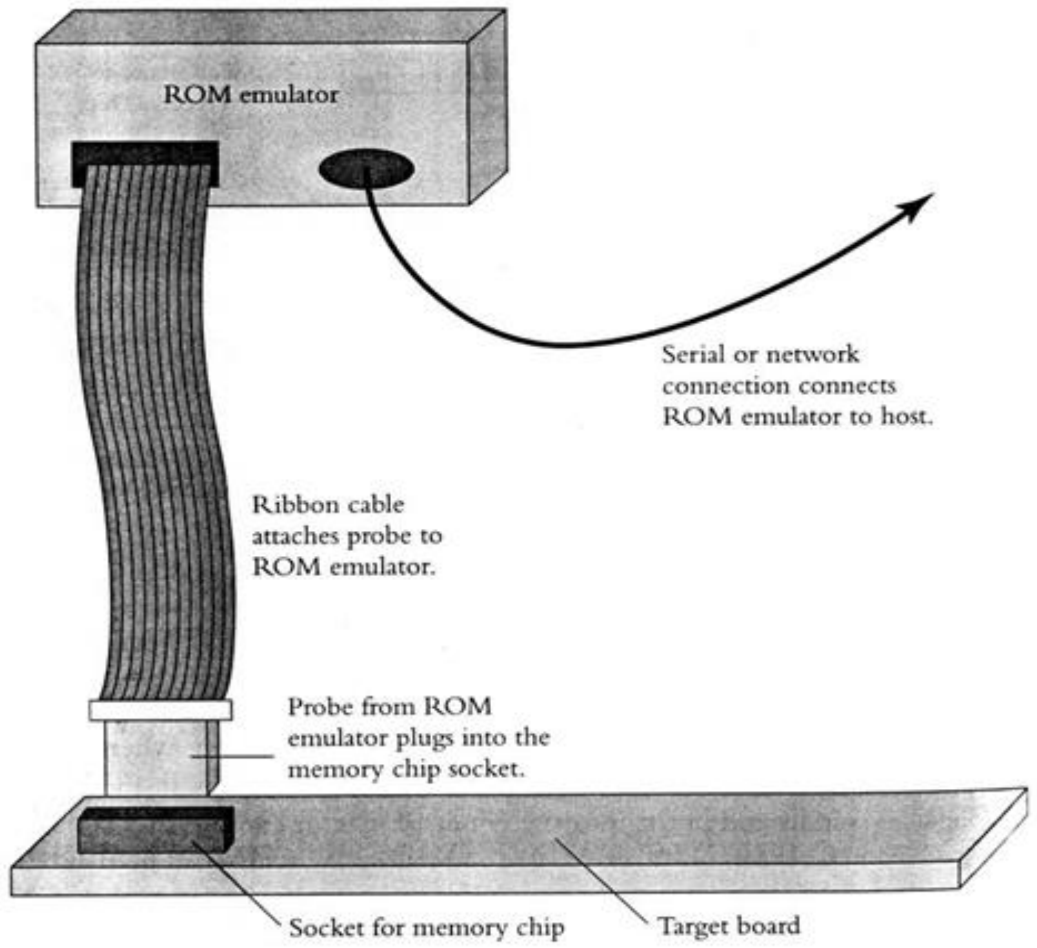
DEBUGGING TECHNIQUES

- **Getting Embedded Software into Target System – 1**

➢ Using Flash Memory

➢ For debugging, a flash memory can be loaded with target Map code using a software on the host over a serial port or network connection (just like using an EPROM)

DEBUGGING TECHNIQUES


ROM emulator
Serial or network connection connects ROM emulator to host.
Ribbon cable attaches probe to ROM emulator.
Probe from ROM emulator plugs into the memory chip socket.
Socket for memory chip
Target board

DEBUGGING TECHNIQUES

Advantages:

- No need to pull the flash (unlike PROM) for debugging different embedded code
- Transferring code into flash (over a network) is faster and hassle-free
- Modifying and/or debugging the flash programming software requires moving it into RAM, modify/debug, and reloading it into target flash memory using above methods

DEBUGGING TOOLS

Advantages:

New versions of embedded software (supplied by vendor) can be loaded into flash memory by customers over a network - Requires a) protecting the flash programmer, saving it in RAM and executing from there, and reloading into flash after new version is written and b) the ability to complete loading new version even if there are crashes and protecting the startup code as in (a)

DEBUGGING TECHNIQUES

Advantages:

- No need to pull the flash (unlike PROM) for debugging different embedded code
- Transferring code into flash (over a network) is faster and hassle-free
- Modifying and/or debugging the flash programming software requires moving it into RAM, modify/debug, and reloading it into target flash memory using above methods

SIMPLE VOLT/OHM METER

- Simple volt-ohm meter can be used to test the target hardware.
- It has two leds red and black
- One end is connected to meter and other is connected to point between which the voltage or resistance is to be measured
- The meter is set for volt for checking the power supply voltage at sorce and voltage level at chips and port pins.
- The meter is set for ohm for checking thebroken connections,improper ground connections,burn out resistance and diods.

SIMPLE LED TESTS AND LOGIC PROBE

- A logic probe is a hand-held test probe used for analyzing and troubleshooting the logical states (boolean 0 or 1) of a digital circuit.

- Most modern logic probes typically have one or more LEDs on the body of the probe:
    - an LED to indicate a high (1) logic state.
    - an LED to indicate a low (0) logic state.
    - an LED to indicate changing back and forth between low and high states.

OSCILLOSCOPE

- An 'oscilloscope', previously called an 'oscillograph', and informally known as a scope or o-scope, CRO (for cathode-ray oscilloscope), or DSO (for the more modern digital storage oscilloscope), is a type of electronic test instrument that graphically displays varying signal voltage, usually as a two-dimensional plot of one or more signals as a function of time. Other signals (such as sound or vibration) can be converted to voltages and displayed.
- Oscilloscopes display the change of an electrical signal over time, with voltage and time as the Y- and X-axes, respectively, on a calibrated scale.

OSCILLOSCOPE

- The waveform can then be analyzed for properties such as amplitute, f requency, rise time , time interval, distortion, and others.
- The oscilloscope can be adjusted so that repetitive signals can be observed as a continuous shape on the screen.
- A storage oscilloscope can capture a single event and display it continuously, so the user can observe events that would otherwise appear too briefly to see directly.
- Oscilloscopes are used in the sciences, medicine, engineer

BIT RATE METER

- In telecommunications and computng , bit rate (bit rate or as a variable *R*) is the number of bits that are conveyed or processed per unit of time.
- The bit rate is quantified using the bits per second unit (symbol: "bit/s"), often in conjunction with an SI prefix such as "kilo" (1 kbit/s = 1,000 bit/s), "mega" (1 Mbit/s = 1,000 kbit/s), "giga" (1 Gbit/s = 1,000 Mbit/s) or "tera" (1 Tbit/s = 1000 Gbit/s). The non-standard abbreviation "bps" is often used to replace the standard symbol "bit/s", so that, for example, "1 Mbps" is used to mean one million bits per second.

BIT RATE METER

The bit rate is calculated using the formula:

1.Frequency × bit depth × channels = bit rate.

2.44,100 samples per second × 16 bits per sample × 2 channels = 1,411,200 bits per second (or 1,411.2 kbps)

3.14,411,200 × 240 = 338,688,000 bits (or 40.37 megabytes)

LOGIC ANALYZER

- A **logic analyzer** is an electronic instrument that captures and displays multiple signals from a digital system or digital circuit.

- A logic analyzer may convert the captured data into timing diagrams, protocol decodes, state machine traces, assembly language, or may correlate assembly with source-level software.

- Logic analyzers have advanced triggering capabilities, and are useful when a user needs to see the timing relationships between many signals in a digital system

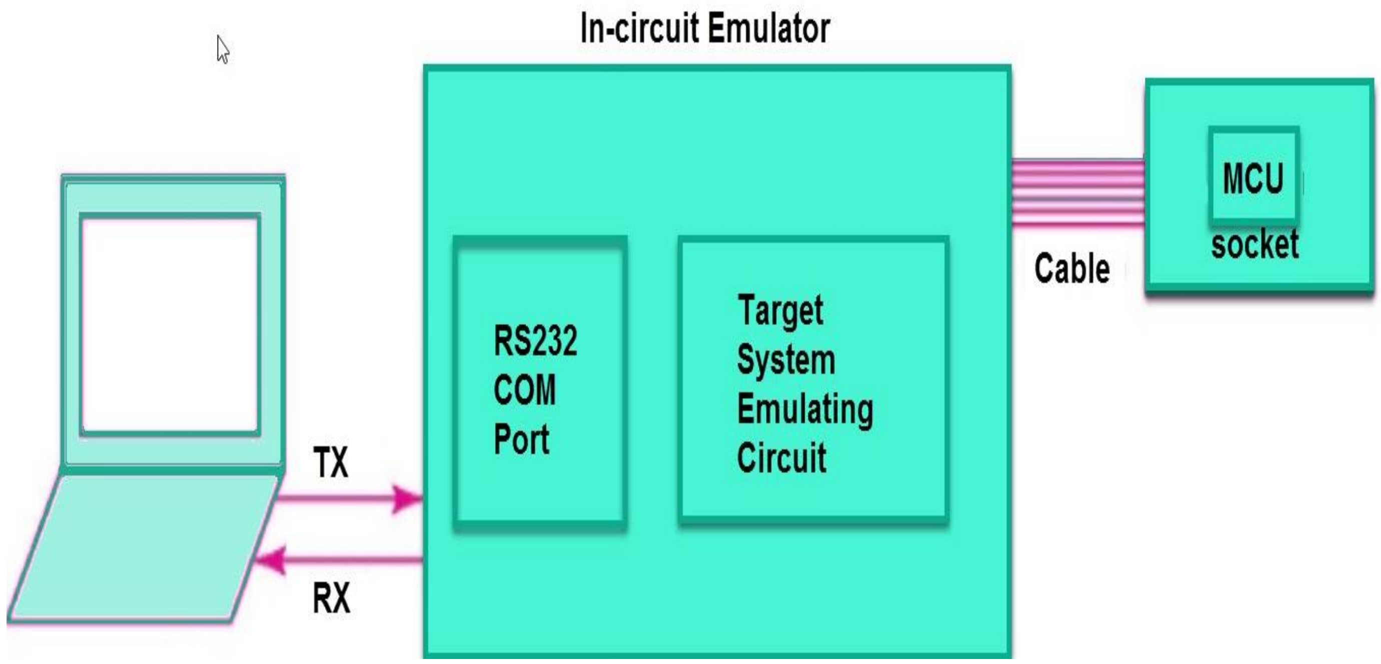
IN-CIRCUIT EMULATOR

- An In-circuit emulator (ICE) is a debugging tool that allows you to access a target MCU for in-depth debugging.

- In-circuit emulation (ICE) is the use of a hardware device or in-circuit emulator used to debug the software of an embedde system.

- It operates by using a processor with the additional ability to support debugging operations, as well as to carry out the main function of the system.

IN-CIRCUIT EMULATOR



IN-CIRCUIT EMULATOR

- ICE consists of a hardware board with accompanying software for the host computer. The ICE is physically connected between the host computer and the target MCU.
- The debugger on the host establishes a connection to the MCU via the ICE. ICE allows a developer to see data and signals that are internal to the MCU, and to step through the source code (e.g., C/C++ on the host) or set breakpoints; the immediate ramifications of executed software are observed during run time.
- Since the debugging is done via hardware, not software, the MCU’s performance is left intact for the most part, and ICE does not compromise MCU resources.

MONITOR

- Monitor is a debugging tool for actual target microprocessor or microcontroller in ICE ROM emulator or in target development board.

- It also lets host system debugging interface just like as an ICE.

- Monitor means a ROM resident program at the target board or ROM emulator connected to ICE.It monitors the device applications ,the runs for different hardware architecture and is used for debugging.

UNIT-V

INTRODUCTION TO ADVANCED PROCESSORS

SYLLABUS:

Introduction to advanced architectures: ARM and SHARC, processor and memory organization and instruction level parallelism; Networked embedded systems: Bus protocols, I2C bus and CAN bus; Internet-Enabled systems, design example-Elevator controller.

ARM instruction set

- ARM versions.
- ARM assembly language.
- ARM programming model.
- ARM memory organization.
- ARM data operations.
- ARM flow of control

ARM versions

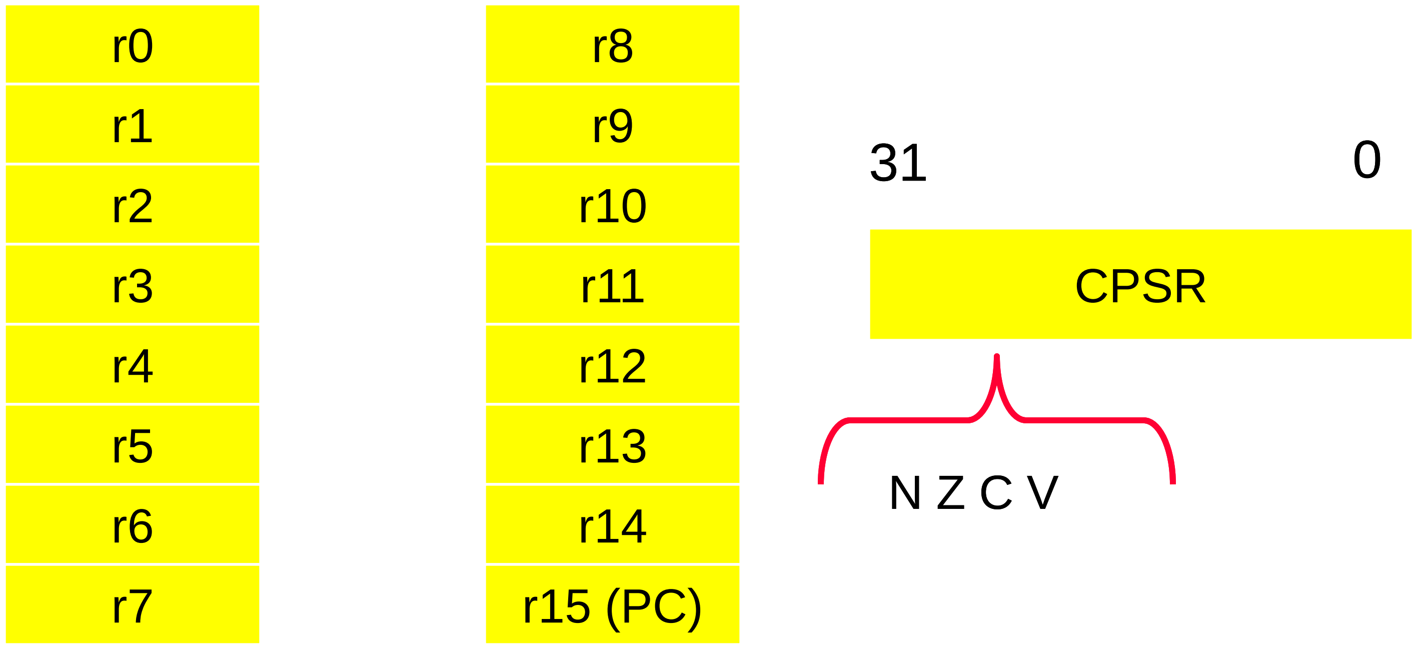
- ARM architecture has been extended over several versions.
- We will concentrate on ARM7.

ARM assembly language

- Fairly standard assembly language:

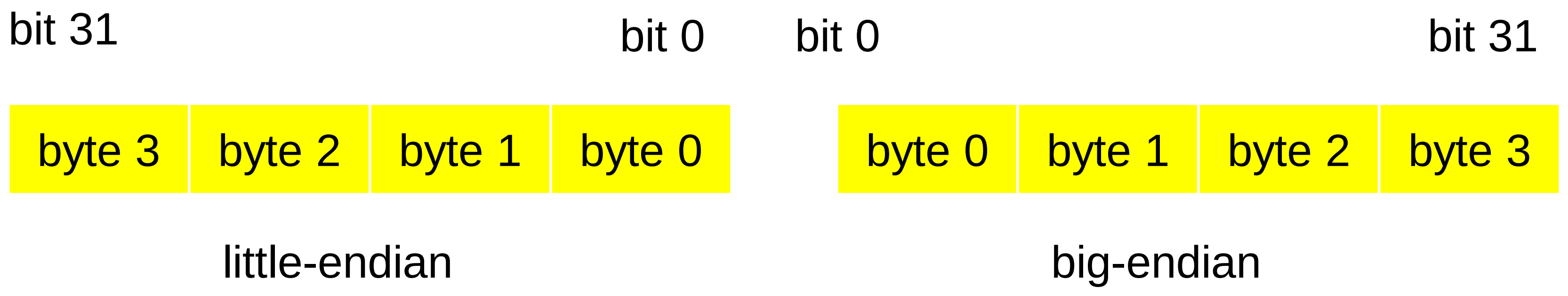
```
LDR r0,[r8] ; a comment
label    ADD r4,r0,r1
```

ARM programming model



Endianness

- **Relationship between bit and byte/word ordering defines endianness:**



ARM data types

- Word is 32 bits long.
- Word can be divided into four 8-bit bytes.
- ARM addresses can be 32 bits long.
- Address refers to byte.
- Address 4 starts at byte 4.
- Can be configured at power-up as either little- or bit-endian mode.

ARM status bits

- Every arithmetic, logical, or shifting operation sets CPSR bits:
  N (negative), Z (zero), C (carry), V (overflow).
- Examples:
  -1 + 1 = 0: NZCV = 0110.
  2^{31}-1+1 = -2^{31}: NZCV = 1001.

ARM data instructions

- Basic format:

  ADD r0,r1,r2

  -Computes r1+r2, stores in r0.

- Immediate operand:

  ADD r0,r1,#2

  -Computes r1+2, stores in r0.

ARM data instructions

- ADD, ADC : add (w. carry)
- SUB, SBC : subtract (w. carry)
- RSB, RSC : reverse subtract (w. carry)
- MUL, MLA : multiply (and accumulate)
- AND, ORR, EOR
- BIC : bit clear
- LSL, LSR : logical shift left/right
- ASL, ASR : arithmetic shift left/right
- ROR : rotate right
- RRX : rotate right extended with C

Data operation varieties

- Logical shift:

  -fills with zeroes.

- Arithmetic shift:

  -fills with ones.

- RRX performs 33-bit rotate, including C bit from CPSR above sign bit.

ARM comparison instructions

- **CMP : compare**
- **CMN : negated compare**
- **TST : bit-wise AND**
- **TEQ : bit-wise XOR**
- **These instructions set only the NZCV bits of CPSR.**

ARM move instructions

- MOV, MVN : move (negated)

  MOV r0, r1 ; sets r0 to r1

NUMBER BASE CONVERSION

- LDR, LDRH, LDRB : load (half-word, byte)
- STR, STRH, STRB : store (half-word, byte)
- Addressing modes:

  -register indirect : LDR r0,[r1]

  -with second register : LDR r0,[r1,-r2]

  -with constant : LDR r0,[r1,#4]

ARM ADR pseudo-op

- Cannot refer to an address directly in an instruction.
- Generate value by performing arithmetic on PC.
- ADR pseudo-op generates instruction required to calculate address:

  ADR r1,FOO

Example: C assignments

- C:

```
x = (a + b) - c;
```

- Assembler:

```
ADR r4,a            ; get address for a
LDR r0,[r4]         ; get value of a
ADR r4,b            ; get address for b, reusing r4
LDR r1,[r4]         ; get value of b
ADD r3,r0,r1        ; compute a+b
ADR r4,c            ; get address for c
LDR r2,[r4]         ; get value of c
```

C assignment, cont’d.

```
SUB r3,r3,r2      ; complete computation of x
ADR r4,x          ; get address for x
 STR r3,[r4]      ; store value of x
```

Example: C assignment

- C:

```
y = a*(b+c);
```

- Assembler:

```
ADR r4,b ; get address for b
LDR r0,[r4] ; get value of b
ADR r4,c ; get address for c
LDR r1,[r4] ; get value of c
ADD r2,r0,r1 ; compute partial result
ADR r4,a ; get address for a
LDR r0,[r4] ; get value of a
```

C assignment, cont'd.

```
MUL r2,r2,r0 ; compute final value for y
ADR r4,y       ; get address for y
STR r2,[r4]     ; store y
```

Example: C assignment

- C:

```
z = (a << 2) |  (b & 15);
```

- Assembler:

```
ADR r4,a ; get address for a
LDR r0,[r4] ; get value of a
MOV r0,r0,LSL 2 ; perform shift
ADR r4,b ; get address for b
LDR r1,[r4] ; get value of b
AND r1,r1,#15 ; perform AND
ORR r1,r0,r1 ; perform OR
```

C assignment, cont’d.

```
ADR r4,z     ; get address for z
STR r1,[r4]  ; store value for z
```

Additional addressing modes

- Base-plus-offset addressing:
  LDR r0,[r1,#16]
  Loads from location r1+16
- Auto-indexing increments base register:
  LDR r0,[r1,#16]!
- Post-indexing fetches, then does offset:
  LDR r0,[r1],#16
  Loads r0 from r1, then adds 16 to r1.

ARM flow of control

- All operations can be performed conditionally, testing CPSR:
  EQ, NE, CS, CC, MI, PL, VS, VC, HI, LS, GE, LT, GT, LE
- Branch operation:
  B #100
  Can be performed conditionally

Example: if statement

- C:

```
if (a > b) { x = 5; y = c + d; } else x = c - d;
```

- Assembler:

```
; compute and test condition
 ADR r4,a ; get address for a
 LDR r0,[r4] ; get value of a
 ADR r4,b ; get address for b
 LDR r1,[r4] ; get value for b
 CMP r0,r1 ; compare a < b
 BLE fblock ; if a ><= b, branch to false block
```

If statement, cont'd.

```
; true block
  MOV r0,#5 ; generate value for x
  ADR r4,x ; get address for x
  STR r0,[r4] ; store x
  ADR r4,c ; get address for c
  LDR r0,[r4] ; get value of c
  ADR r4,d ; get address for d
  LDR r1,[r4] ; get value of d
  ADD r0,r0,r1 ; compute y
  ADR r4,y ; get address for y
  STR r0,[r4] ; store y
  B after ; branch around false block
```

If statement, cont'd.

```
; false block
 fblock ADR r4,c ; get address for c
   LDR r0,[r4] ; get value of c
   ADR r4,d ; get address for d
   LDR r1,[r4] ; get value for d
   SUB r0,r0,r1 ; compute a-b
   ADR r4,x ; get address for x
   STR r0,[r4] ; store value of x
 after ...
```

Example: switch statement

- **C:**

  switch (test) { case 0: … break; case 1: … }

- **Assembler:**

```
ADR r2,test ; get address for test
LDR r0,[r2] ; load value for test
ADR r1,switchtab ; load address for switch table
LDR r1,[r1,r0,LSL #2] ; index switch table
        switchtab DCD case0
DCD case1
        ...
```

Example: FIR filter

- **C**:

```
for (i=0, f=0; i<N; i++)
 f = f + c[i]*x[i];
```

- **Assembler**

```
; loop initiation code
 MOV r0,#0 ; use r0 for I
 MOV r8,#0 ; use separate index for arrays
 ADR r2,N ; get address for N
 LDR r1,[r2] ; get value of N
 MOV r2,#0 ; use r2 for f
```

FIR filter, cont’.d

ADR r3,c ; load r3 with

base of c

ADR r5,x ; load r5 with

base of x

; loop body

loop LDR r4,[r3,r8] ; get c[i]

LDR r6,[r5,r8] ; get x[i]

MUL r4,r4,r6 ; compute

c[i]*x[i]

ADD r2,r2,r4 ; add into

running sum

ADD r8,r8,#4 ; add one

ARM subroutine linkage

- Branch and link instruction:

  BL foo

Copies current PC to r14.

To return from subroutine:

MOV r15,r14

Nested subroutine calls

- **Nesting/recursion requires coding convention:**

```
f1 LDR r0,[r13] ; load arg into r0 from stack

; call f2()

STR r14,[r13]! ; store f1’s return adrs

STR r0,[r13]! ; store arg to f2 on stack

BL f2 ; branch and link to f2

; return from f1()

SUB r13,#4 ; pop f2’s arg off stack

LDR r13!,r15 ; restore register and return
```

SHARC instruction set

- SHARC programming model.
- SHARC assembly language.
- SHARC memory organization.
- SHARC data operations.
- SHARC flow of control

SHARC programming model

- **Register files:**

  R0-R15 (aliased as F0-F15 for floating point)
- **Status registers.**
- **Loop registers.**
- **Data address generator registers.**
- **Interrupt registers.**

SHARC assembly language

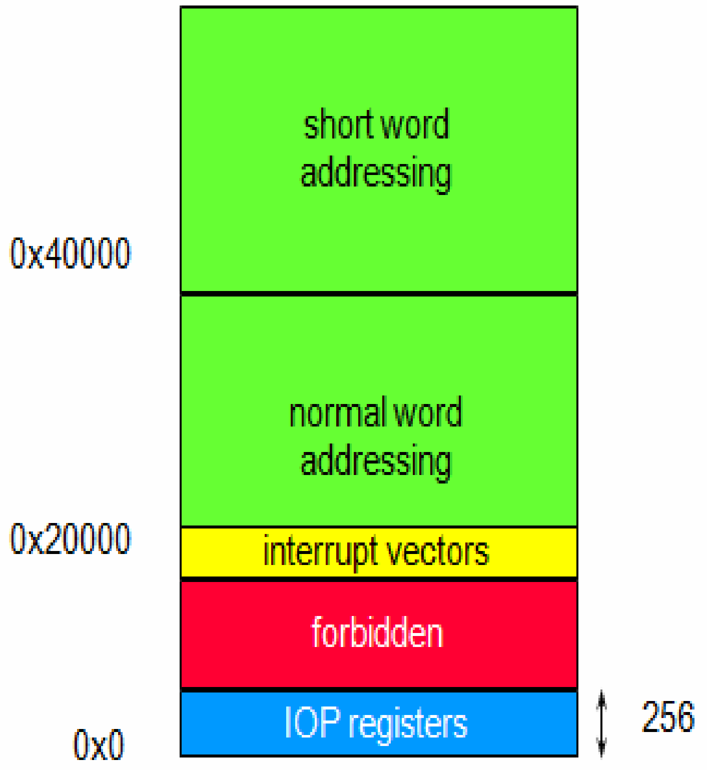
Algebraic notation terminated by semicolon:

```
R1=DM(M0,I0), R2=PM(M8,I8); ! comment
label: R3=R1+R2;
```

data memory access

program memory access

SHARC MEMORY SPACE



SHARC DATA TYPES

- 32-bit IEEE single-precision floating-point.
- 40-bit IEEE extended-precision floating-point.
- 32-bit integers.
- Memory organized internally as 32-bit words.

SHARC MICRO ARCHITECTURE

- Modified Harvard architecture.
- Program memory can be used to store some data.
- Register file connects to:
- multiplier
- shifter;
- ALU.

SHARC MODE REGISTERS

- Most important:
- ASTAT: arithmetic status.
- STKY: sticky.
- MODE 1: mode 1.

ROUNDING AND SATURATION

- Floating-point can be:
  - rounded toward zero;
  - rounded toward nearest.
- ALU supports saturation arithmetic (ALUSAT bit in MODE1).
  - Overflow results in max value, not rollover.

MULTIPLIER

- Fixed-point operations can accumulate into local MR registers or be written to register file. Fixed-point result is 80 bits.
- Floating-point results always go to register file.
- Status bits: negative, under/overflow, invalid, fixed-point underflow, floating-point underflow, floating-point invalid.

ALU/SHIFTER STATUS FLAGS

ALU:

- zero, overflow, negative, fixed-point carry, input sign, floating-point invalid, last op was floating-point, compare accumulation registers, floating-point under/oveflow, fixed-point overflow, floating-point invalid

Shifter:

- zero, overflow, sign

FLAG OPERATIONS

- All ALU operations set AZ (zero), AN (negative), AV (overflow), AC (fixed-point carry), AI (floating-point invalid) bits in ASTAT.
- STKY is sticky version of some ASTAT bits.

SHARC load/store

- Load/store architecture: no memory-direct operations.
- Two data address generators (DAGs):
  - program memory;
  - data memory.
- Must set up DAG registers to control loads/stores.

SHARC program sequencer

Features:

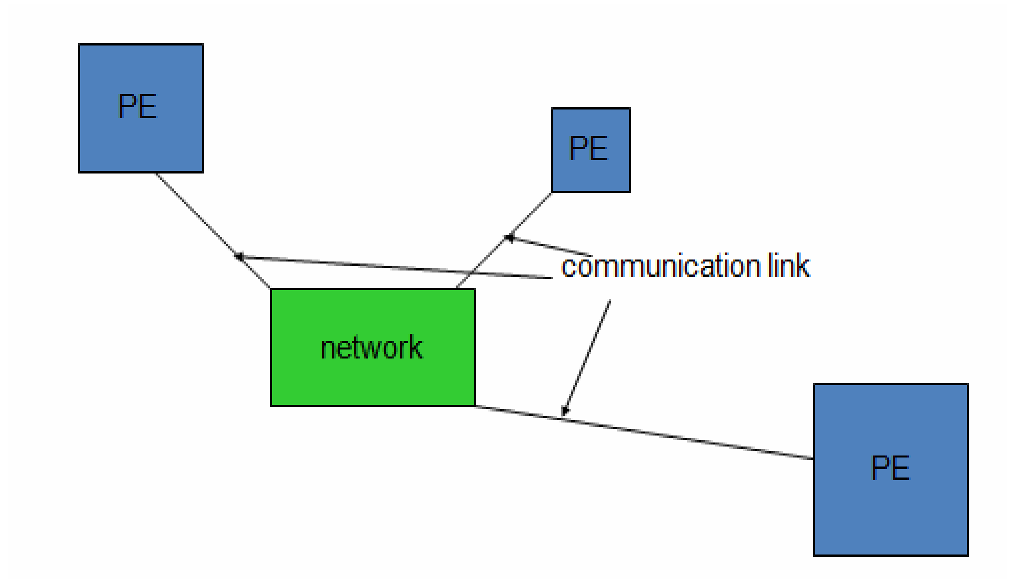
- instruction cache;
- PC stack;
- status registers;
- loop logic;
- data address generator;

Networking for Embedded Systems

- Why we use networks.
- Network abstractions.
- Example networks.

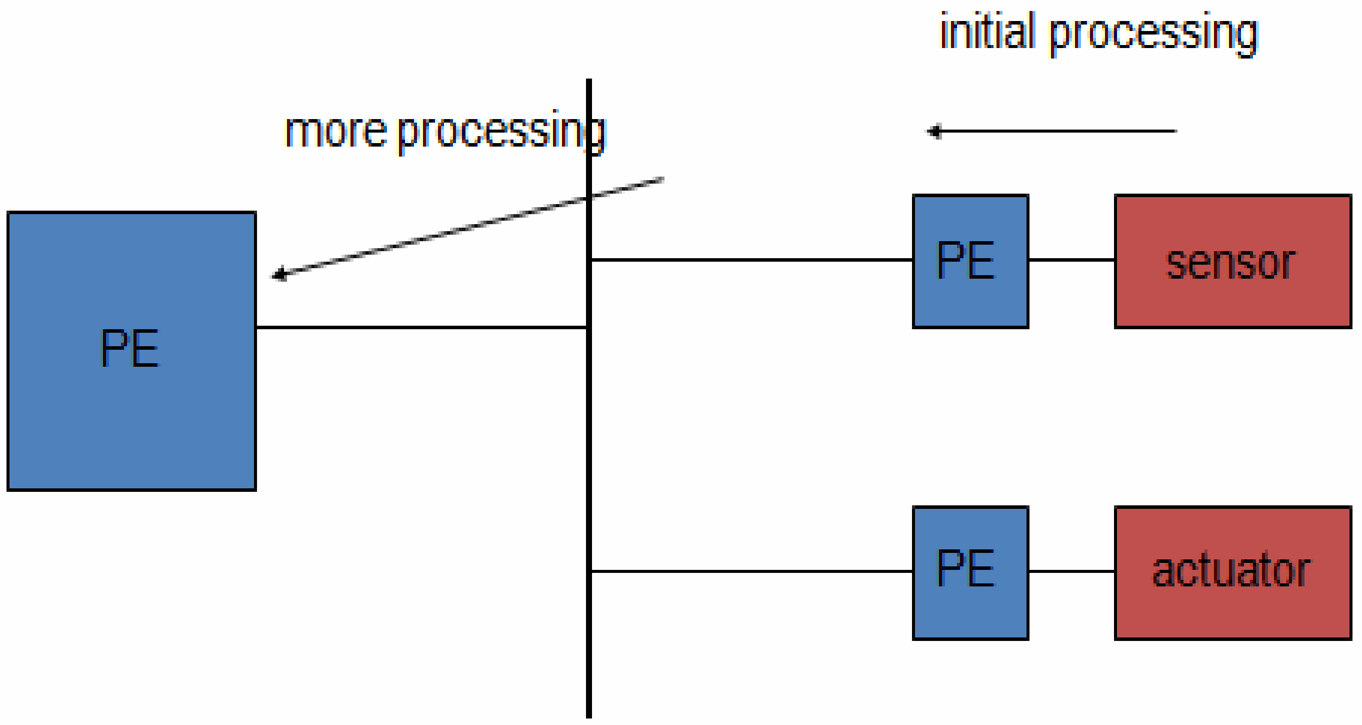
Network elements

Distributed computing platform:



PEs may be CPUs or ASICs.

Networks in embedded systems



Why distributed?

- Higher performance at lower cost.
- Physically distributed activities---time constants may not allow transmission to central site.
- Improved debugging---use one CPU in network to debug others.
- May buy subsystems that have embedded processors.

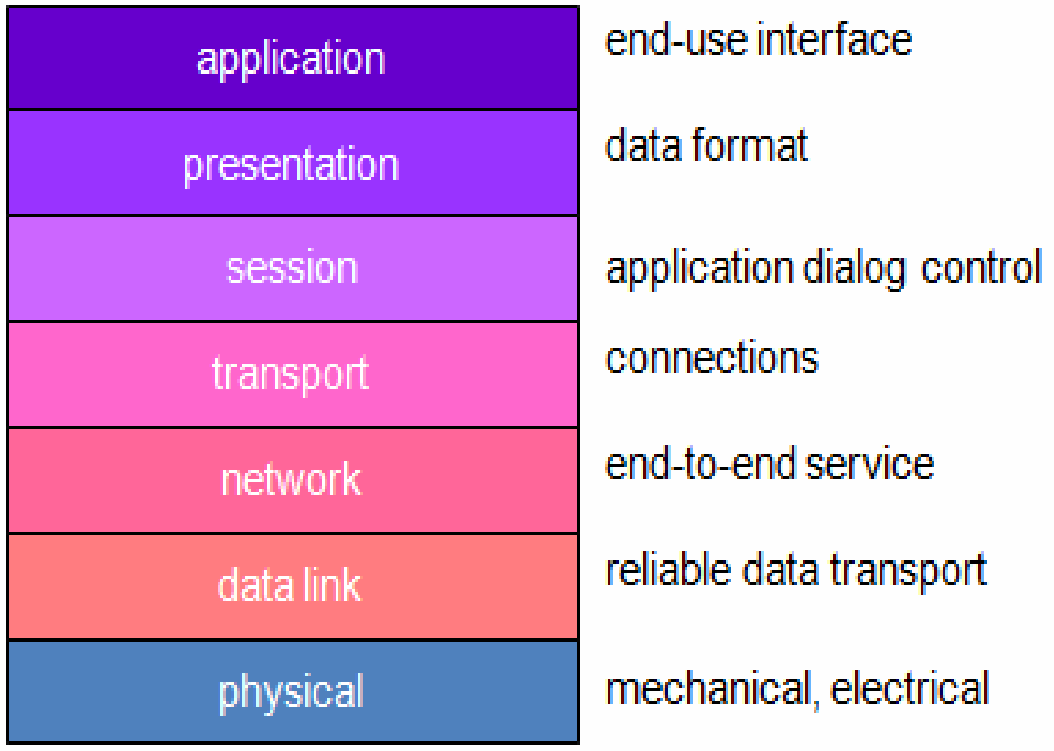
Network abstractions

- International Standards Organization (ISO) developed the Open Systems Interconnection (OSI) model to describe networks:

  7-layer model.

- Provides a standard way to classify network components and operations.

OSI model

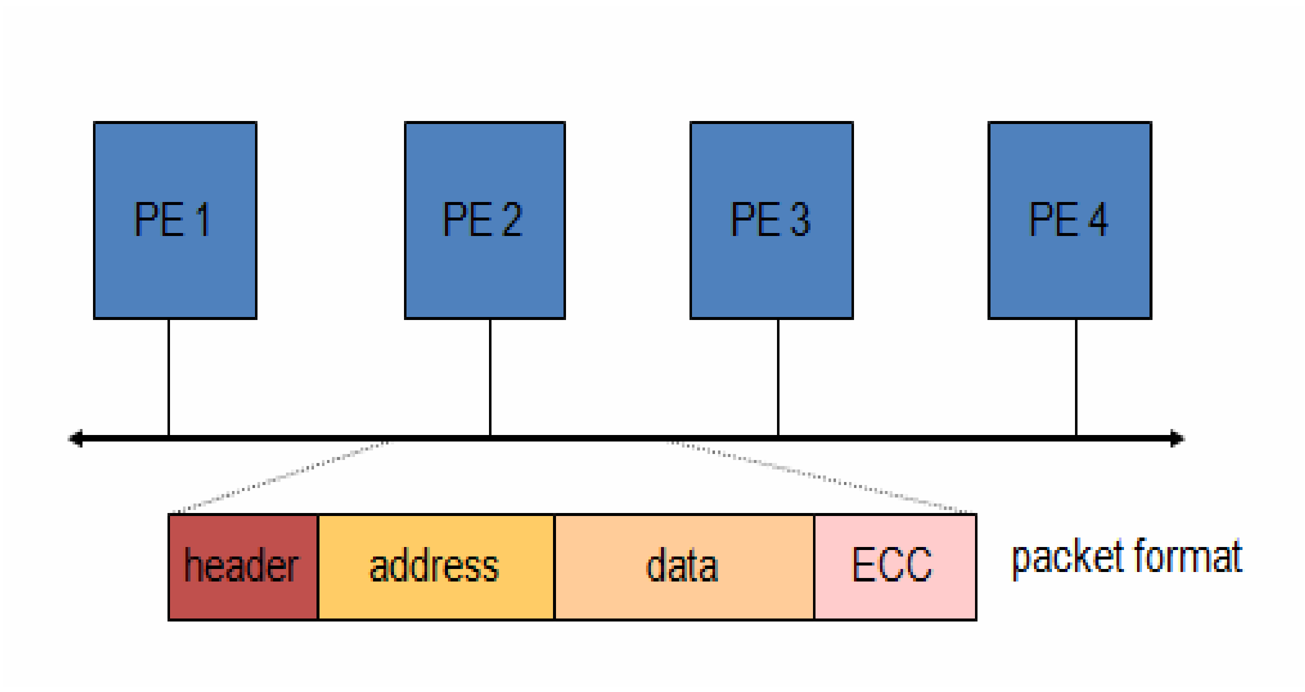

application
presentation
session
transport
network
data link
physical
end-use interface
data format
application dialog control
connections
end-to-end service
reliable data transport
mechanical, electrical

OSI layers

- Physical: connectors, bit formats, etc.
- Data link: error detection and control across a single link (single hop).
- Network: end-to-end multi-hop data communication.
- Transport: provides connections; may optimize network resources.
- Session: services for end-user applications: data grouping, check pointing, etc.
- Presentation: data formats, transformation services.
- Application: interface between network and end-user programs.

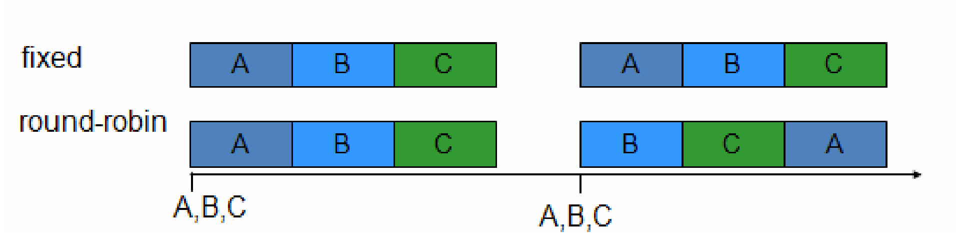
Bus networks

- Common physical connection:

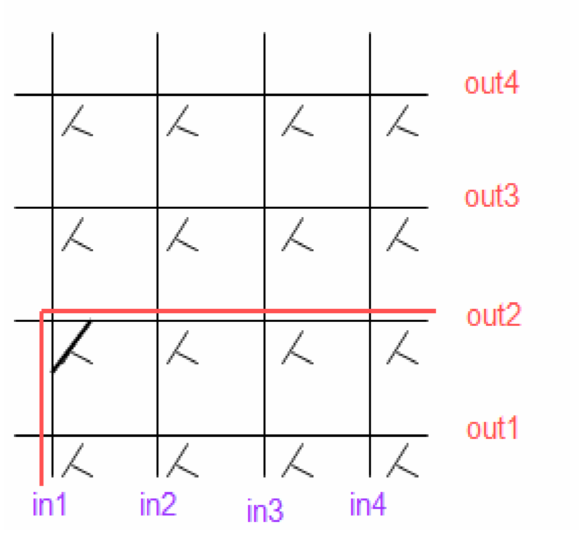


Bus arbitration

- Fixed: Same order of resolution every time.
- Fair: every PE has same access over long periods.
- Round-robin: rotate top priority among Pes.



Crossbar



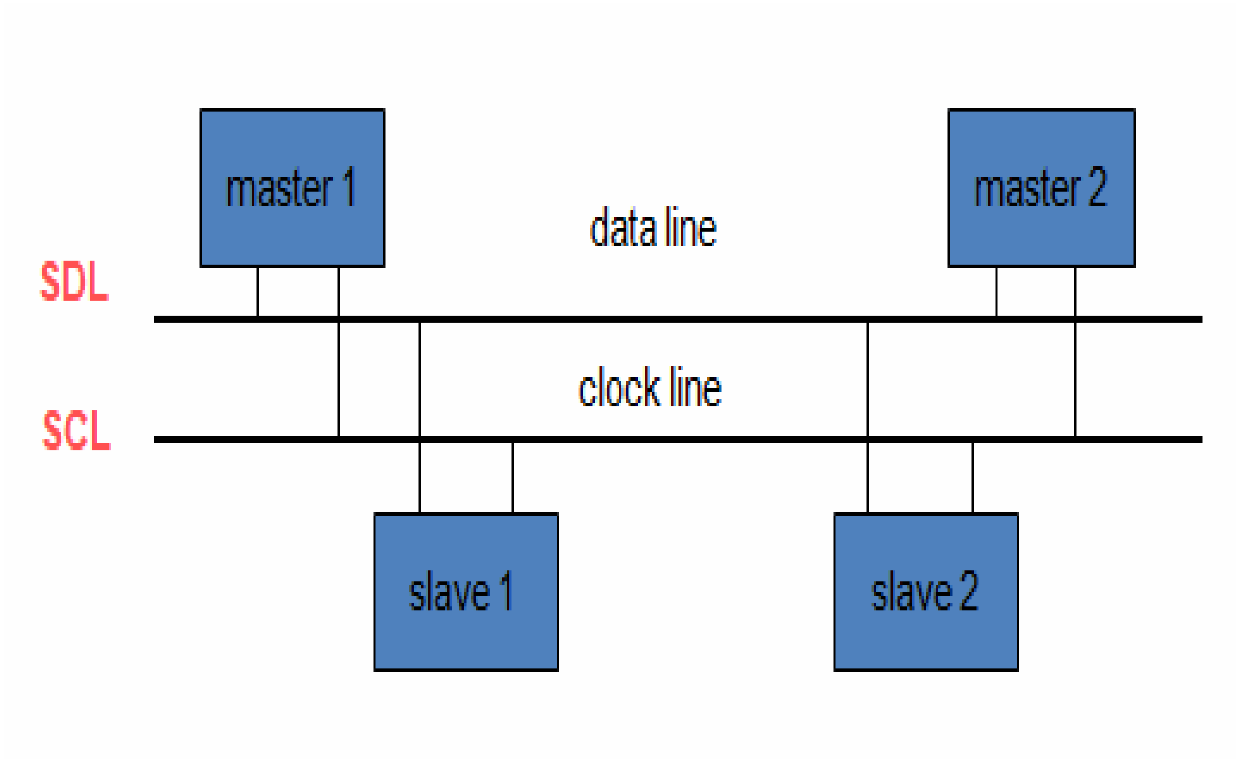
Crossbar characteristics:

- Non-blocking.
- Can handle arbitrary multi-cast combinations.
- Size proportional to n^2.

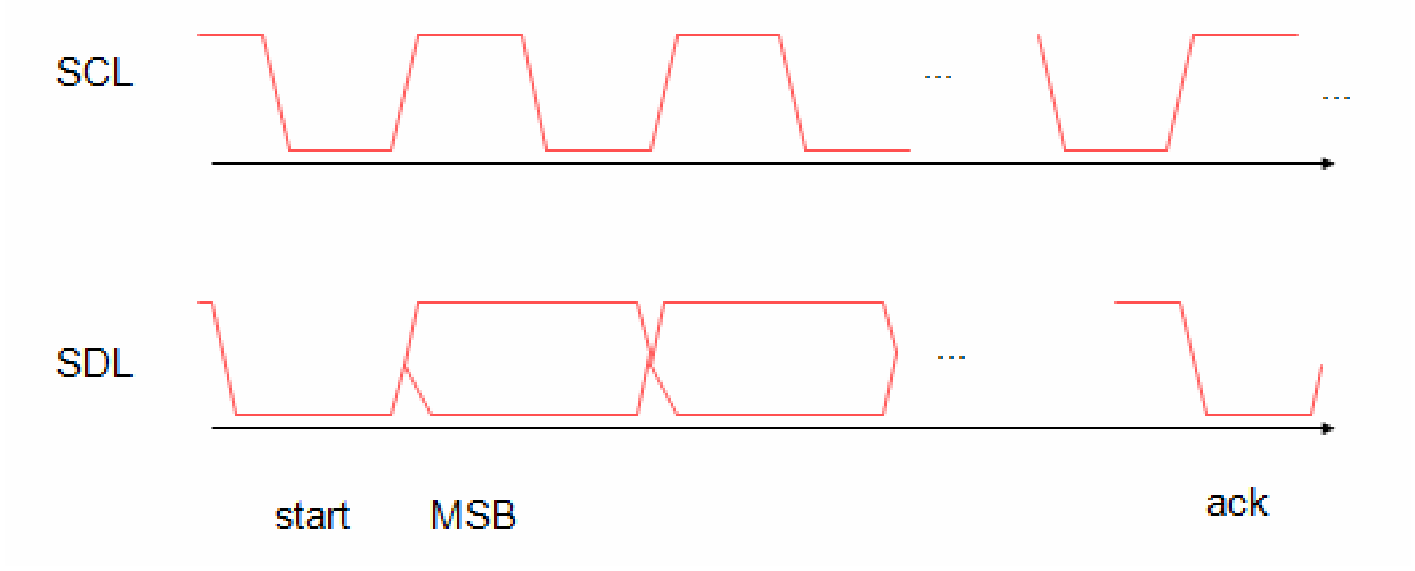
I2C bus

- Designed for low-cost, medium data rate applications.
- Characteristics:
  - serial;
  - multiple-master;
  - fixed-priority arbitration.
- Several microcontrollers come with built-in I^{2}C controllers.

I2C physical layer



I2C data format



I2C signaling

- Sender pulls down bus for 0.
- Sender listens to bus---if it tried to send a 1 and heard a 0, someone else is simultaneously transmitting.
- Transmissions occur in 8-bit bytes.

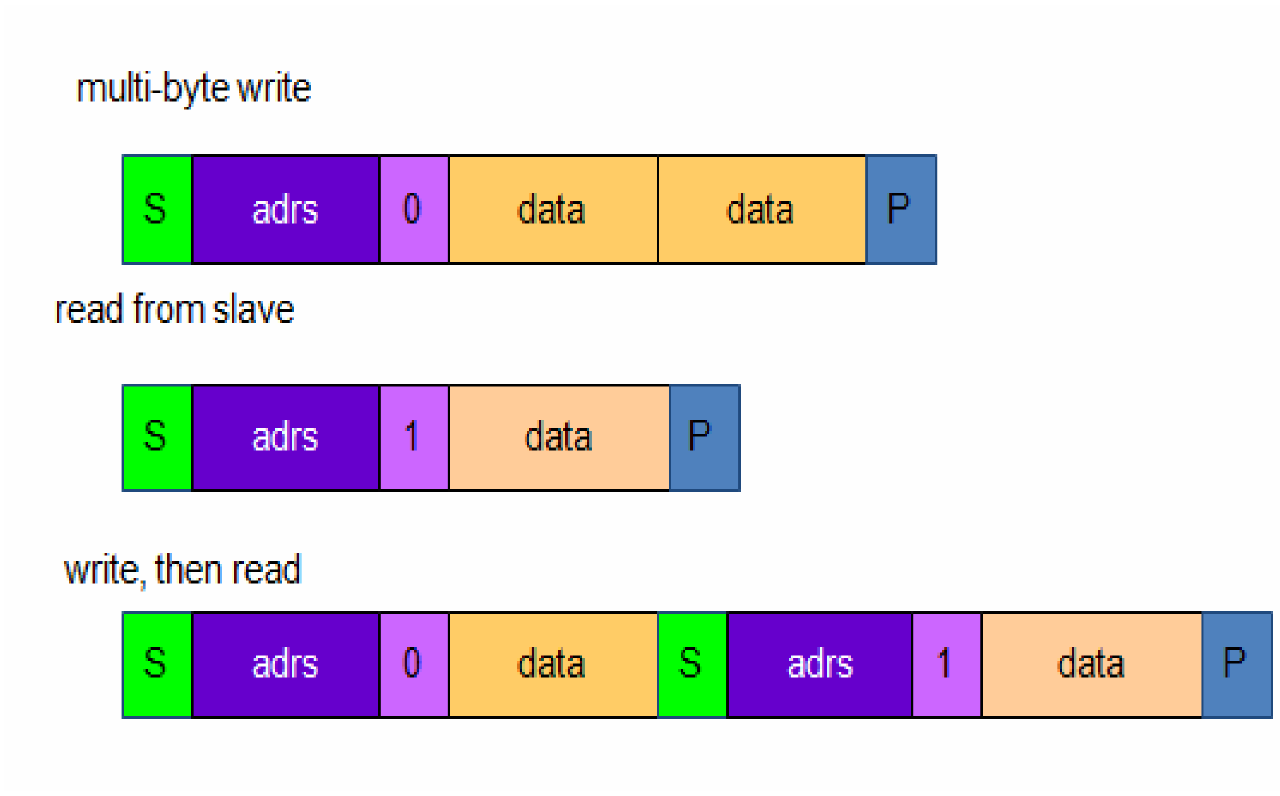
I^{2}C data link layer

- Every device has an address (7 bits in standard, 10 bits in extension).Bit 8 of address signals read or write.
- General call address allows broadcast.

I2C bus arbitration

- Sender listens while sending address.
- When sender hears a conflict, if its address is higher, it stops signaling.
- Low-priority senders relinquish control early enough in clock cycle to allow bit to be transmitted reliably.

I2C transmissions

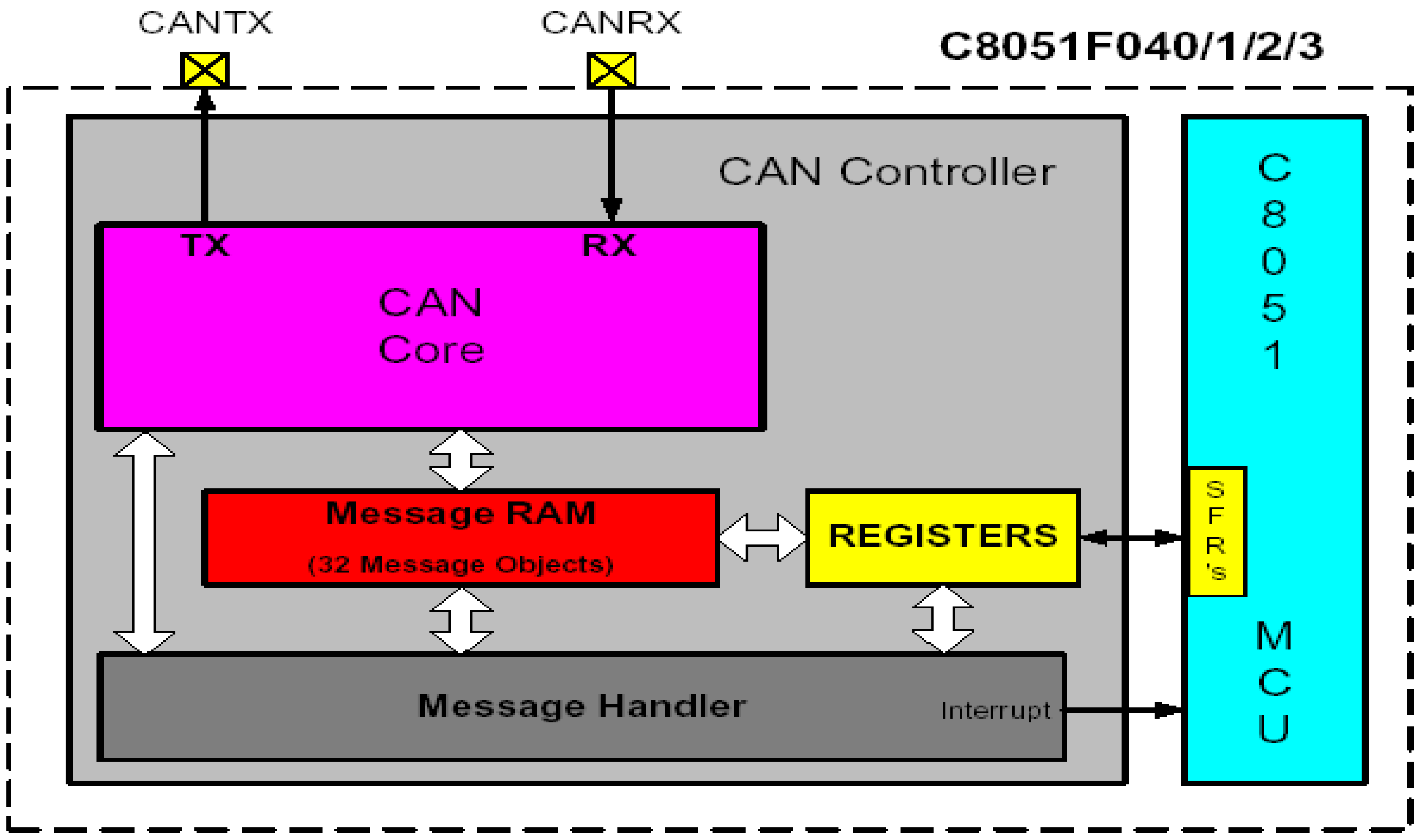


CAN BUS

- CAN (Controller Area Network) is a serial bus system used to communicate between several embedded 8-bit and 16-bit microcontrollers.

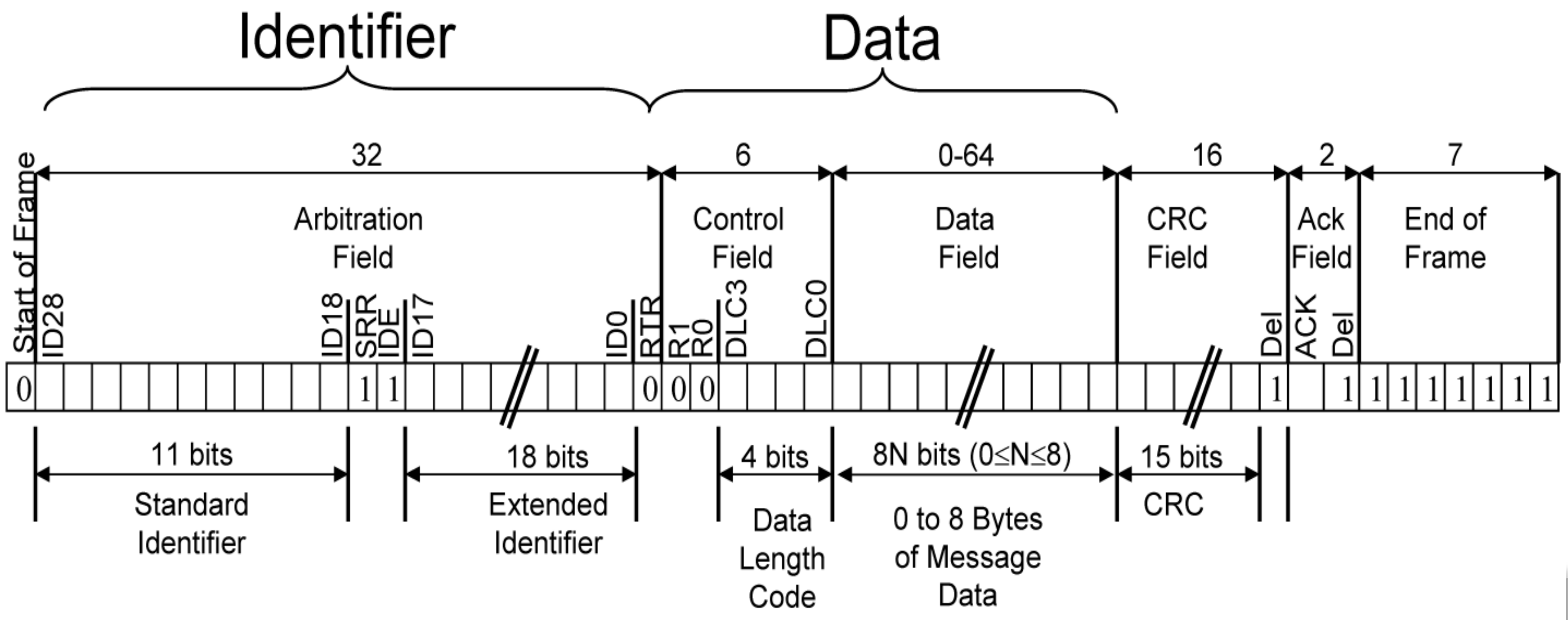
- It was originally designed for use in the automotive industry but is used today in many other systems (e.g. home appliances and industrial machines).

CAN Controller Diagram



Data Format

- Each message has an ID, Data and overhead.
- Data –8 bytes max
- Overhead – start, end, CRC, ACK



Internet –EnAnalyzed systems

- Embedded systems are internet enabled by using TCP/IP protocols for networking to internet and assigning IP addresses to each systems.
- Internet provides a standard way for embedded systems to act in concert with other devices and with users.eg.

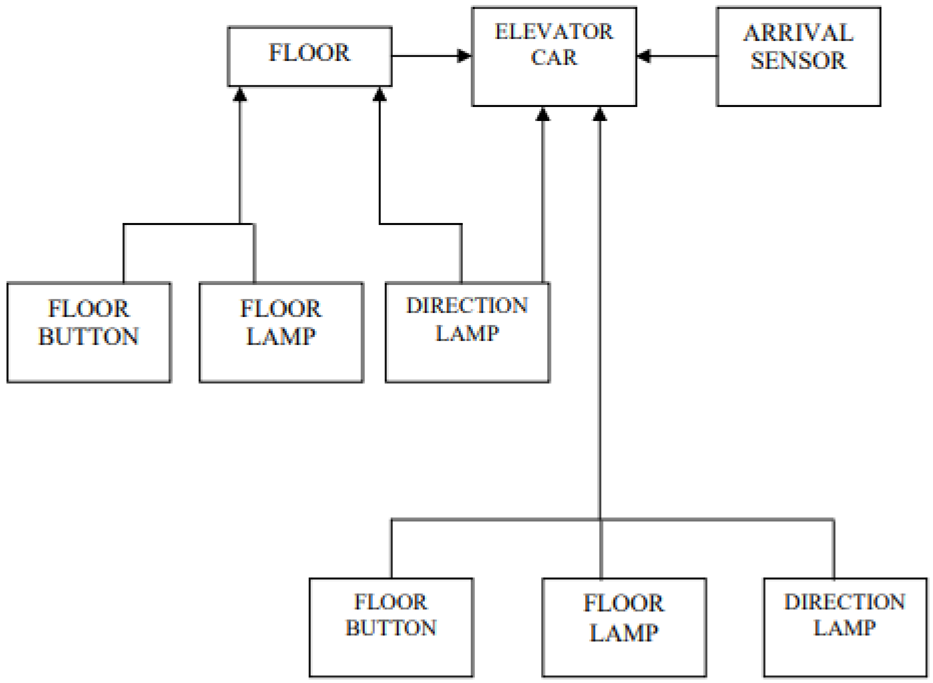
1.High end laser printers use internet protocols to receive print jobs from host machines.

2.PDA can display web pages ,read email and synchronous calendar information with remote computer.

ELEVATOR CONTROLLER

- An elevator system is a vertical transport vehicle that efficiently moves people or goods between floors of a building. They are generally powered by electric motors.
- The most popular elevator is the rope elevator. In the rope elevator, the car is raised and lowered by transaction with steel rope.
- Elevators also have electromagnetic brakes that engage, when the car comes to a stop. The electromagnetic actually keeps the brakes in the open position. Instead of closing them with the design, the brakes will automatically clamp shut if the elevator loses power.
- Elevators also have automatic braking systems near the top and the bottom of the elevator shaft.

ELEVATOR CONTROLLER



ELEVATOR SYSTEM OVERVIEW