(R22CSE3147) EMBEDDED SYSTEMS

Unit Number	Unit Name	Topic Name
1	**Introduction to Embedded Systems**	Definition of Embedded System
		Embedded Systems Vs General Computing Systems
		History of Embedded Systems
		Classification
		Major Application Areas
		Purpose of Embedded Systems
		Characteristics and Quality Attributes of Embedded Systems
2	**Introduction to Microcontrollers**	Overview of 8051 Microcontroller
		Architecture
		I/O Ports
		Memory Organization
		Addressing Modes and Instruction set of 8051
		Simple Programs
3	**ARM Architecture**	ARM Processor fundamentals
		ARM Architecture
		Register
		CPSR
		Pipeline
		Exceptions and Interrupts
		Interrupt vector table
		ARM instruction set
		Data Processing, Branch instructions
		load store instructions
		Software interrupt instructions
		Program status register instructions
		loading constants, Conditional execution
		Introduction to Thumb instructions
4	**Embedded Firmware**	Reset Circuit, Brown-Protection Circuit
		Oscillator Unit, Real Time Clock
		Watchdog Timer
		Embedded Firmware Design
		Approaches and Development Languages
5	**RTOS Based Embedded System Design**	Operating Systems Basics
		Types of Operating Systems
		Tasks, Process and Threads
		Multiprocessing and Mutitasking
		Task Scheduling

UNIT-I

INTRODUCTION TO EMBEDDED SYSTEMS

Introduction:

An embedded system can be thought of as a computer hardware system having software embedded in it. It can be an independent system or it can be a part of a large system. An embedded system is a microcontroller or microprocessor based system which is designed to perform a specific task. For example, a fire alarm is an embedded system; it will sense only smoke.

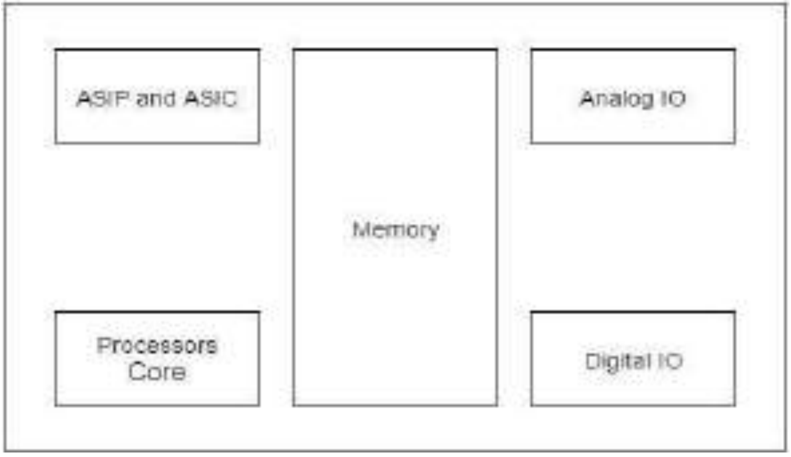
An embedded system has three components –

- It has hardware.
- It has application software.
- It has Real Time Operating System (RTOS) though a small scale embedded system may not have RTOS.

So we can define an embedded system as a Microcontroller based, software driven, reliable, real-time control system.

Characteristics of an Embedded System:

Single-functioned − An embedded system usually repeats a specialized operation.
Ex: A pager always functions as a pager.
Tightly constrained − All computing systems have constraints on design metrics like cost, size, power, and performance but those on an embedded system can be especially tight. Design metrics is a measure of an implementation's features. It must be of a size to fit on a single chip, must perform fast enough to process data in real time and consume minimum power to extend battery life.
Reactive and Real time − Many embedded systems must continually react to changes in the system's environment and must compute certain results in real time without any delay. For example, a car cruise controller that continually monitors and reacts to speed and brake sensors. It must compute acceleration or decelerations repeatedly within a limited time; a delayed computation can result in failure to control of the car.
Microprocessors based − It must be microprocessor or microcontroller based.
Memory − It must have a memory, as its software usually embeds in ROM.
Connected − It must have connected peripherals to connect input and output devices.
HW-SW systems − Software is used for more features and flexibility. Hardware is used for performance and security



Advantages:

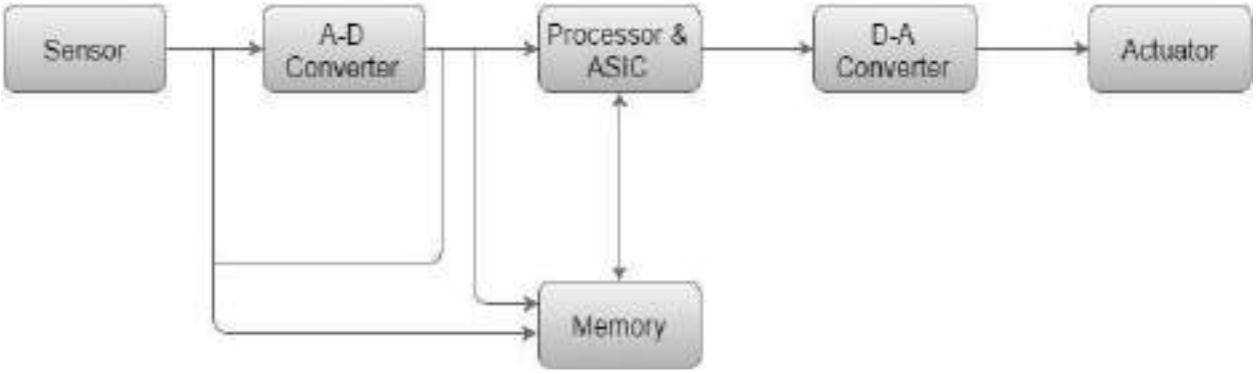
- Small size
- Low power consumption
- Low cost
- Enhanced performance
- Easily customizable

Disadvantages:

- High development effort
- Larger time to market

Basic Structure of an Embedded System:

The following illustration shows the basic structure of an embedded system –



Sensor – It measures the physical quantity and converts it to an electrical signal which can be read by an observer or by any electronic instrument like an A-D converter. A sensor stores the measured quantity to the memory.

A-D Converter – An analog-to-digital converter converts the analog signal sent by the sensor into a digital signal.

Processor & ASICs – Processors process the data to measure the output and store it to the memory.

D-A Converter – A digital-to-analog converter converts the digital data fed by the processor to analog data

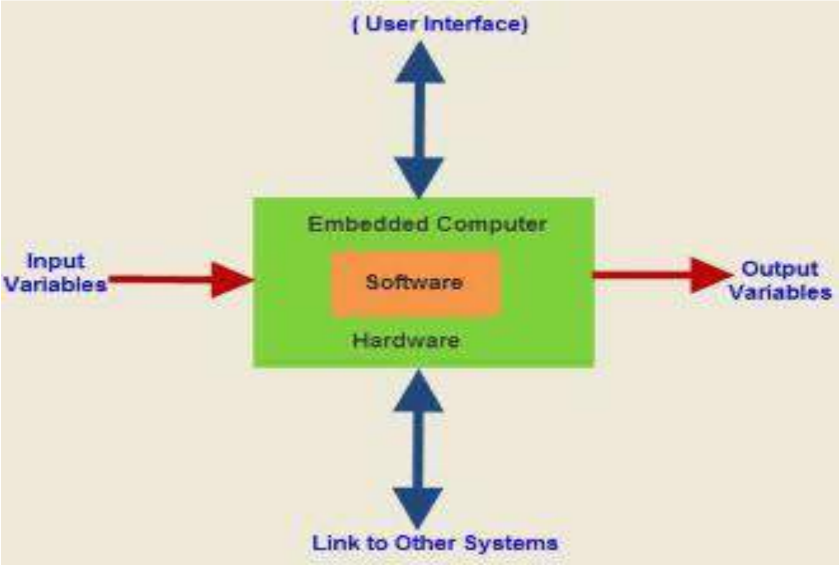
Actuator – An actuator compares the output given by the D-A Converter to the actual (expected) output stored in it and stores the approved output.

Difference between General purpose computer and Embedded systems:

General Purpose Computer	Embedded Systems
It is designed using a microprocessor as the main processing unit.	It is mostly designed using a microcontroller as the main processing unit.
It contains a large memory, semiconductor memories like cache and RAM. It also contains secondary storage like hard disks etc.	It uses semiconductor memories but does not require secondary memories like hard disk CD. It sometime has special memory called flash memory.

It is designed such that it can do multiple tasks as per requirement.	It is designed such that it can do a particular predefined task.
It is mostly costlier compared to the embedded systems	It is cheaper compared to a computer.
It requires huge number of peripheral devices and their controllers	It is cheaper as it requires less no of peripheral devices and their controllers are microcontroller chip itself.
The Operating system and other software for the general purpose computers, are normally complicated and occupy more memory space.	The operating system (mostly RTOS i.e Real Time Operating System) and other software occupy less memory space.

The Embedded system hardware includes elements like user interface, input/output interfaces, display and memory, etc. Generally, an embedded system comprises power supply, processor, memory, timers, serial communication ports and system application specific circuits.



Architechture of Embedded Systems:

The 8051 microcontrollers work with 8-bit data bus. So they can support external data memory up to 64K and external program memory of 64k. They can address 128k of external memory.

When data and code lie in different memory blocks, then the architecture is referred as **Harvard architecture**. In case data and code lie in the same memory block, then the architecture is referred as **Von Neumann architecture**.

Von Neumann Architecture: The Von Neumann architecture was first proposed by a computer scientist John von Neumann. In this architecture, one data path or bus exists for both instruction and data. As a result, the CPU does one operation at a time. It either fetches an instruction from memory, or performs read/write operation on data. So an instruction fetch and a data operation cannot occur simultaneously, sharing a common bus. Von-Neumann architecture supports simple hardware. It allows the use of a single, sequential memory.

Harvard Architecture:

The Harvard architecture offers separate storage and signal buses for instructions and data. This architecture has data storage entirely contained within the CPU, and there is no access to the instruction storage as data. Computers have separate memory areas for program instructions and data using internal data buses, allowing simultaneous access to both instructions and data. Programs needed to be loaded by an operator; the processor could not boot itself. In a Harvard architecture, there is no need to make the two memories share properties.

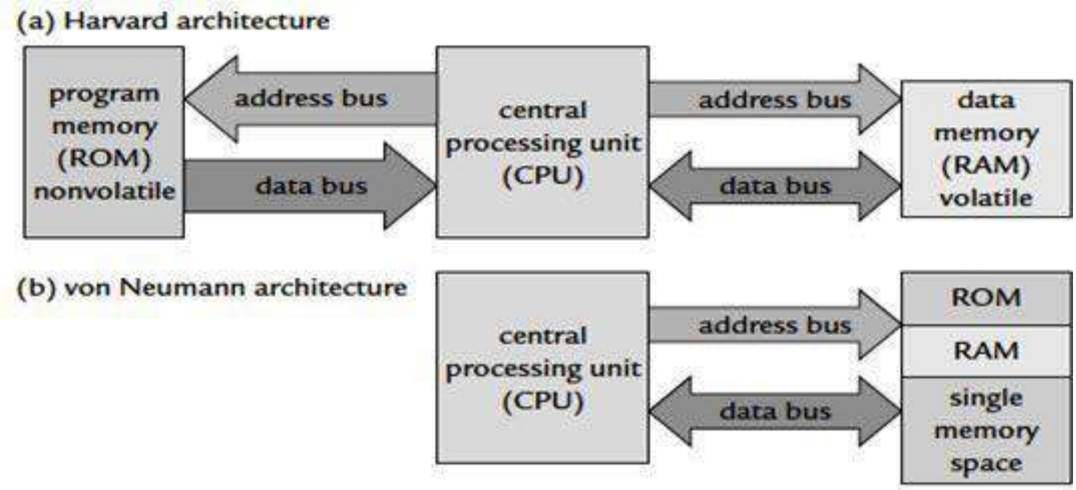

(a) Harvard architecture
program memory (ROM) nonvolatile
address bus
data bus
central processing unit (CPU)
address bus
data bus
data memory (RAM) volatile
(b) von Neumann architecture
central processing unit (CPU)
address bus
data bus
ROM
RAM
single memory space

Figure: Von-Neumann and Harvard architectures

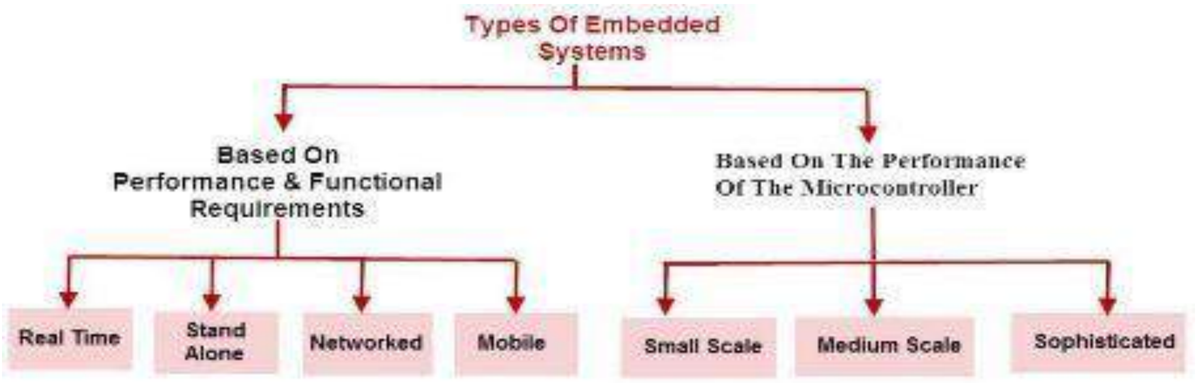
Von-Neumann Architecture vs Harvard Architecture:

The following points distinguish the Von Neumann Architecture from the Harvard Architecture.

Von-Neumann Architechture	**Harvard Architecture**
Single memory to be shared by both code and data.	Separate memories for code and data.
Processor needs to fetch code in a separate clock cycle and data in another clock cycle. So it requires two clock cycles.	Single clock cycle is sufficient, as separate buses are used to access code and data.
Higher speed, thus less time consuming.	Slower in speed, thus more time-consuming.
Simple in design.	Complex in design.

Types of Embedded systems:

Embedded systems can be classified into different types based on performance, functional requirements and performance of the microcontroller.


Types Of Embedded Systems
Based On Performance & Functional Requirements
Based On The Performance Of The Microcontroller
Real Time
Stand Alone
Networked
Mobile
Small Scale
Medium Scale
Sophisticated

Embedded systems are classified into four categories based on their performance and functional requirements:

Standalone embedded systems, Real time embedded systems, Networked embedded systems and Mobile embedded systems.

Embedded Systems are classified into three types based on the performance of the microcontroller such as:

Small scale embedded systems, Medium scale embedded systems, Sophisticated embedded systems

Stand Alone Embedded Systems

Standalone embedded systems do not require a host system like a computer, it works by itself. It takes the input from the input ports either analog or digital and processes, calculates and converts the data and gives the resulting data through the connected device-Which either controls, drives and displays the connected devices. Examples for the stand alone embedded systems are mp3 players, digital cameras, video game consoles, microwave ovens and temperature measurement systems.

Real Time Embedded Systems

A real time embedded system is defined as, a system which gives a required o/p in a particular time. These types of embedded systems follow the time deadlines for completion of a task. Real time embedded systems are classified into two types such as soft and hard real time systems.

Networked Embedded Systems

These types of embedded systems are related to a network to access the resources. The connected network can be LAN, WAN or the internet. The connection can be any wired or wireless. This type of embedded system is the fastest growing area in embedded system applications. The embedded web server is a type of system wherein all embedded devices are connected to a web server and accessed and controlled by a web browser. Example for the LAN networked embedded system is a home security system wherein all sensors are connected and run on the protocol TCP/IP

Mobile Embedded Systems

Mobile embedded systems are used in portable embedded devices like cell phones, mobiles, digital cameras, mp3 players and personal digital assistants, etc. The basic limitation of these devices is the other resources and limitation of memory.

Small Scale Embedded Systems

These types of embedded systems are designed with a single 8 or 16-bit microcontroller, that may even be activated by a battery. For developing embedded software for small scale embedded systems, the main programming tools are an editor, assembler, cross assembler and integrated development environment (IDE).

Medium Scale Embedded Systems

These types of embedded systems design with a single or 16 or 32-bit microcontroller, RISCs or DSPs. These types of embedded systems have both hardware and software complexities. For developing embedded software for medium scale embedded systems, the main programming tools are C, C++, JAVA, Visual C++, RTOS, debugger, source code engineering tool, simulator and IDE.

Sophisticated Embedded Systems

These types of embedded systems have enormous hardware and software complexities, that may need ASIPs, IPs, PLAs, scalable or configurable processors. They are used for cutting-edge applications that need hardware and software Co-design and components which have to assemble in the final system.

Applications of Embedded Systems:

Embedded systems are used in different applications like automobiles, telecommunications, smart cards, missiles, satellites, computer networking and digital consumer electronics.

1.1. What is an Embedded System?

Definition:

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

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

Embedded Systems are:

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

1.2. Embedded Systems Vs General Computing Systems:

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

1.3.History of Embedded Systems:

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

It has two modules

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

1.4.Classification of Embedded Systems:

- Based on Generation
- Based on Complexity & Performance
- Based on deterministic behaviour
- Based on Triggering

1.4.1.Embedded Systems - Classification based on Generation:

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

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

Ex: **Smart phone devices, MIDs etc.**

1.4.2.Embedded Systems - Classification based on Complexity & Performance:

Small Scale:

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

- washing machine.
- Oven.
- Automatic Door Lock.
- Motion Controlled Home Security System.
- Keyboard controller.
- CD Drive.
- fax machine.

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. Various examples of medium scale embedded systems are routers for networking, ATM (is. Automated Teller Machine for bank transactional machines etc.

Large Scale/Complex:

- Embedded Systems built around high performance 32- or 64-bit RISC processors/controllers, RSoC or multi-core processors and PLD.
- It requires complex hardware and software.
- This 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.

Note: RSOC stands for Reconfigurable System-on-Chip, which is a technology that uses microprocessors and programmable logic fabric on the same die.

1.4.3.Classification Based on deterministic behavior:

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

These are classified in to two types

1. ***Soft Real time Systems:*** Missing a deadline may not be critical and can be tolerated to a certain degree
2. ***Hard Real time systems:*** Missing a program/task execution time deadline can have catastrophic consequences (financial, human loss of life, etc.)

1.4.4.Embedded Systems - Classification Based on Triggering:

These are classified into two types

1. ***Event Triggered:*** Activities within the system (e.g., task run-times) are dynamic and depend upon occurrence of different events**.**
2. ***Time triggered:*** Activities within the system follow a statically computed schedule (i.e., they are allocated time slots during which they can take place) and thus by nature are predictable.

1.5.Major Application Areas of Embedded Systems:

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

1.6.Purpose of Embedded Systems:

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

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

1.6.1.Data Collection/Storage/Representation:-

Performs acquisition of data from the external world. The data may be text, audio, video or any physical quantities. The collected data can be either analog or digital. Data collection is usually done for storage, analysis, manipulation and transmission. The collected data may be stored directly in the system or may be transmitted to some other systems or it may be processed by the system or it may be deleted instantly.

Ex: Digital Camera

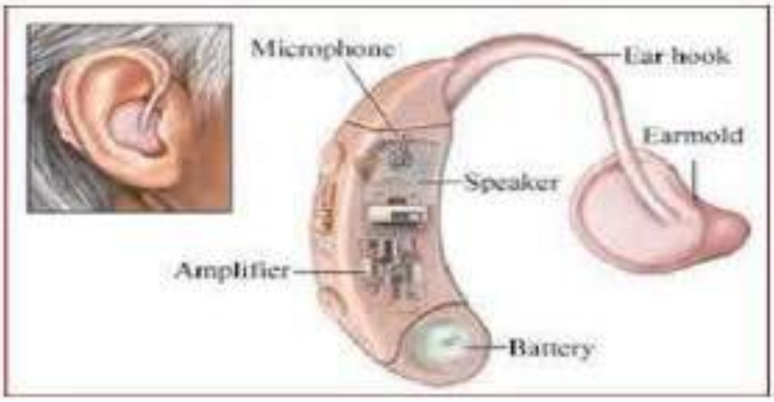
1.6.2.Data Communication:-

Embedded Data communication systems are deployed in applications ranging from complex satellite communication systems to simple home networking systems. Embedded Data communication systems are dedicated for data communication. The data communication can happen through a wired interface (like Ethernet, RS-232C/USB/IEEE1394 etc) or wireless interface (like Wi-Fi, GSM,/GPRS, Bluetooth, ZigBee etc)

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

1.6.3.Data (Signal) Processing

- Embedded systems with Signal processing functionalities are employed in applications demanding signal processing like Speech coding, synthesis, audio video codec, transmission applications etc
- Computationally intensive systems Employs Digital Signal Processors (DSPs)



1.6.4.Monitoring: -

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

1.6.5.Control:

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

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

Air conditioner contains a room temperature sensing element (sensor) which may be a thermistor and a handheld unit for setting up (feeding) the desired temperature. The air compressor unit acts as the actuator. The compressor is controlled according to the current room temperature and the desired temperature set by the end user.

1.6.5. Application Specific User Interface:-

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

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

Aimed at a specific target group of users

Mobile handsets, Control units in industrial applications etc are examples

1.7.Characteristics of Embedded systems:

- Application and domain specific
- Reactive and Real Time
- Operates in harsh environments
- Distributed
- Small Size and weight
- Power concerns

1.7.1.Application and Domain Specific: -

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

1.7.2.Reactive and Real Time: -

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

1.7.3.Operates in Harsh Environment: –

The design of E.S should take care of the operating conditions of the area where the system is going to implement.

Ex – If the system needs to be deployed in a high temperature zone, then all the components used in the system should be of high temperature grade. Also proper shock absorption techniques should be provided to systems which are going to be commissioned in places subject to high shock.

1.7.4.Distributed: –

It means that embedded systems may be a part of a larger system. Many numbers of such distributed embedded systems form a single large embedded control unit.

Ex – Automatic vending machine. It contains a card reader, a vending unit etc. Each of them are independent embedded units but they work together to perform the overall vending function.

1.7.5.Small Size and Weight: -

Product aesthetics (size, weight, shape, style, etc) is an important factor in choosing a product.

It is convenient to handle a compact device than a bulky product.

1.7.6.Power Concerns: -

Power management is another important factor that needs to be considered in designing embedded systems.

E.S should be designed in such a way as to minimize the heat dissipation by the system.

1.8.Quality Attributes of Embedded System:

Quality attributes are the non-functional requirements that need to be documented properly in any system design

Quality attributes can be classified as

1.Operational quality attributes

2.Non-operational quality attributes.

1.8.1.Operational quality attributes

The operational quality attributes represent the quality attributes related to the embedded system when it is in the operational mode or online mode

The Operational Quality Attributes are

- ✓ Response
- ✓ Throughput
- ✓ Reliability
- ✓ Maintainability
- ✓ Security
- ✓ Saftey

1.Response:

It is the measure of quickness of the system. It tells how fast the system is tracking the changes in input variables. Most of the E.S demands fast response which should be almost real time.

Ex – Flight control application

2.Throughput

It deals with the efficiency of a system. It can be defined as the rate of production or operation of a defined process over a stated period of time. The rates can be expressed in terms of products, batches produced or any other meaningful measurements.

Ex – In case of card reader throughput means how many transactions the reader can perform in a minute or in an hour or in a day.

3.Reliability:

It is a measure of how much we can rely upon the proper functioning of the system.

Mean Time between Failure (MTBF) and Mean Time To Repair (MTTR) are the terms used in determining system reliability. MTBF gives the frequency of failures in hours/weeks/months.

MTTR specifies how long the system is allowed to be out of order following a failure. For embedded system with critical application need, it should be of the order of minutes.

4.Maintainability:

It deals with support and maintenance to the end user or client in case of technical issues and product failure or on the basis of a routine system checkup. Reliability and maintainability are complementary to each other. A more reliable system means a system with less corrective maintainability requirements and vice versa. Maintainability can be broadly classified into two categories. Scheduled or Periodic maintenance (Preventive maintenance), Corrective maintenance to unexpected failures

5.Security:

Confidentiality, Integrity and availability are the three major measures of information security.

Confidentiality deals with protection of data and application from unauthorized disclosure.

Integrity deals with the protection of data and application from unauthorized modification.

Availability deals with protection of data and application from unauthorized users.

6.Saftey:

Safety deals with the possible damages that can happen to the operator, public and the environment due to the breakdown of an Embedded System. The breakdown of an embedded system may occur due to a hardware failure or a firmware failure. Safety analysis is a must in product engineering to evaluate the anticipated damages and determine the best course of action to bring down the consequences of damage to an acceptable level.

1.8.2.Non Operational Quality Attributes:

The quality attributes that needs to be addressed for the product not on the basis of operational aspects are grouped under this category.

The Non Operational Quality Attributes are

- ✓ Testability and Debuggability
- ✓ Evolvability
- ✓ Portability
- ✓ Time to Prototype and Market
- ✓ Per Unit Cost and Revenue

1.Testability and Debuggability:

Testability deals with how easily one can test the design, application and by which means it can be done. For an E.S testability is applicable to both the embedded hardware and firmware. Embedded hardware testing ensures that the peripherals and total hardware functions in the desired manner, whereas firmware testing ensures that the firmware is functioning in the expected way

Debug-ability is a means of debugging the product from unexpected behaviour in the system

Debug-ability is two level process 1. Hardware level 2. software level

1.Hardware level: It is used for finding the issues created by hardware problems.

2.Software level: It is employed for finding the errors created by the flaws in the software.

2.Evolvability: -

- It is a term which is closely related to Biology.
- It is referred as the non-heritable variation.
- For an embedded system evolvability refers to the ease with which the embedded product can be modified to take advantage of new firmware or hardware technologies.

3.Portability: -

It is the measure of system independence.
An embedded product is said to be portable if the product is capable of functioning in various environments, target processors and embedded operating systems.

4.Time-to-Prototype and Market: -

- It is the time elapsed between the conceptualization of a product and the time at which the product is ready for selling.
- The commercial embedded product market is highly competitive and time to market the product is critical factor in the success of commercial embedded product.
- There may be multiple players in embedded industry who develop products of the same category (like mobile phone).

5.Per Unit Cost and Revenue: -

- Cost is a factor which is closely monitored by both end user and product manufacturer.
- Cost is highly sensitive factor for commercial products
- Any failure to position the cost of a commercial product at a nominal rate may lead to the failure of the product in the market.
- Proper market study and cost benefit analysis should be carried out before taking a decision on the per-unit cost of the embedded product.
- The ultimate aim of the product is to generate marginal profit so the budget and total cost should be properly balanced to provide a marginal profit.

Text Book: -

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

(R22CSE3147) EMBEDDED SYSTEMS

Unit Number	Unit Name	Topic Name
2	Introduction to Microcontrollers	Overview of 8051 Microcontroller
		Architecture
		I/O Ports
		Memory Organization
		Addressing Modes and Instruction set of 8051
		Simple Programs

INTEL 8051 MICRCONTROLLER

Introduction:

A decade back the process and control operations were totally implemented by the Microprocessors only. But now-a-days the situation is totally changed and it is occupied by the new devices called Microcontroller. The development is so drastic that we can't find any electronic gadget without the use of a microcontroller. This microcontroller changed the embedded system design so simple and advanced that the embedded market has become one of the most sought after for not only entrepreneurs but for design engineers also.

What is a Microcontroller?

A single chip computer or A CPU with all the peripherals like RAM, ROM, I/O Ports, Timers, ADCs etc... on the same chip. For ex: Motorola's 6811, Intel/s 8051, Zilog's Z8 and PIC 16X etc…

Microprocessors & Microcontrollers:

Microprocessor: A CPU built into a single VLSI chip is called a microprocessor. It is a general-purpose device and additional external circuitry are added to make it a microcomputer. The microprocessor contains arithmetic and logic unit (ALU), Instruction decoder and control unit, Instruction register, Program counter (PC), clock circuit (internal or external), reset circuit (internal or external) and registers. But the microprocessor has no on chip I/O Ports, Timers, Memory etc. For example, Intel 8085 is an 8-bit microprocessor and Intel 8086/8088 a 16-bit microprocessor. The block diagram of the Microprocessor is shown in Fig.1

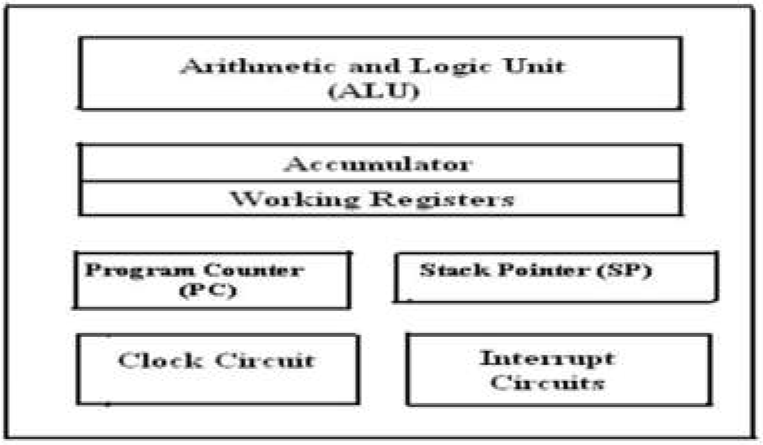


Fig.1 Block diagram of a Microprocessor.

Microcontroller:

A microcontroller is a highly integrated single chip, which consists of on chip CPU (Central Processing Unit), RAM (Random Access Memory), EPROM/PROM/ROM (Erasable Programmable Read Only Memory), I/O (input/output) – serial and parallel, timers, interrupt controller. For example, Intel 8051 is 8-bit microcontroller and Intel 8096 is 16-bit microcontroller. The block diagram of Microcontroller is shown in Fig.2.

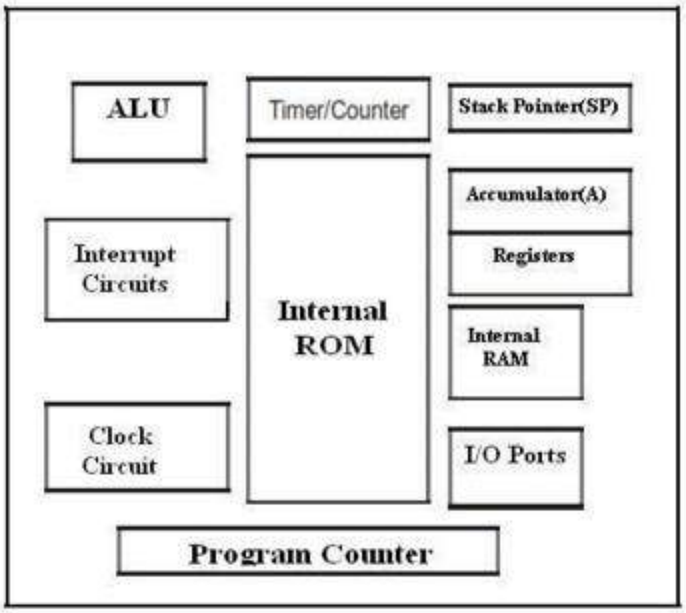


Fig.2.Block Diagram of a Microcontroller

Distinguish between Microprocessor and Microcontroller

S.No	Microprocessor	Microcontroller
i.	A microprocessor is a general purpose device which is called a CPU	A microcontroller is a dedicated chip which is also called single chip computer.
ii.	A microprocessor does not contain on chip I/O Ports, Timers, Memories etc.	A microcontroller includes RAM, ROM, serial and parallel interface, timers, interrupt circuitry (in addition to CPU) in a single chip.
iii.	Microprocessors are most commonly used as the CPU in microcomputer systems	Microcontrollers are used in small, minimum component designs performing Control-oriented applications.
iv.	Microprocessor instructions are mainly nibble or byte addressable	Microcontroller instructions are both bit Addressable as well as byte addressable.
v.	Microprocessor instruction sets are mainly intended for catering to Large volumes of data.	Microcontrollers have instruction sets catering to the control of inputs and Outputs.
vi.	Microprocessor based system design is complex and expensive	Microcontroller based system design is rather simple and cost effective
vii.	The Instruction set of microprocessors complex with large number of instructions.	The instruction set of a Microcontroller is very simple with less number of instructions. For, ex: PIC microcontrollers have only 35 instructions.

viii.	A microprocessor has zero status flag	A microcontroller has no zero flag.

Intel 8051 Microcontroller:

The 8051 microcontroller is a very popular 8-bit microcontroller introduced by Intel in the year 1981 and it has become almost the academic standard now-a-days. The 8051 is based on an 8-bit CISC core with Harvard architecture. Its 8-bit architecture is optimized for control applications with extensive Boolean processing. It is available as a 40-pin DIP chip and works at +5 Volts DC. The salient features of 8051 controllers are given below.

Salient Features: The salient features of 8051 Microcontroller are

1. 4 KB on chip program memory (ROM or EPROM)).
2. 128 bytes on chip data memory(RAM).
3. 8-bit data bus
4. 16-bit address bus
5. 32 general purpose registers each of 8bits
6. Two -16 bit timers T_0 and T_1
7. Five Interrupts (3 internal and 2external).
8. Four Parallel ports each of 8-bits (PORT0, PORT1, PORT2, PORT3) with a total of 32 I/O lines.
9. One 16-bit program counter and One 16-bit DPTR (data pointer)
10. One 8-bit stack pointer

Architecture & Block Diagram of 8051 Microcontroller:

The architecture of the 8051 microcontroller can be understood from the block diagram. It has Harvard architecture with RISC (Reduced Instruction Set Computer) concept. The block diagram of 8051 microcontroller is shown in Fig.3. below

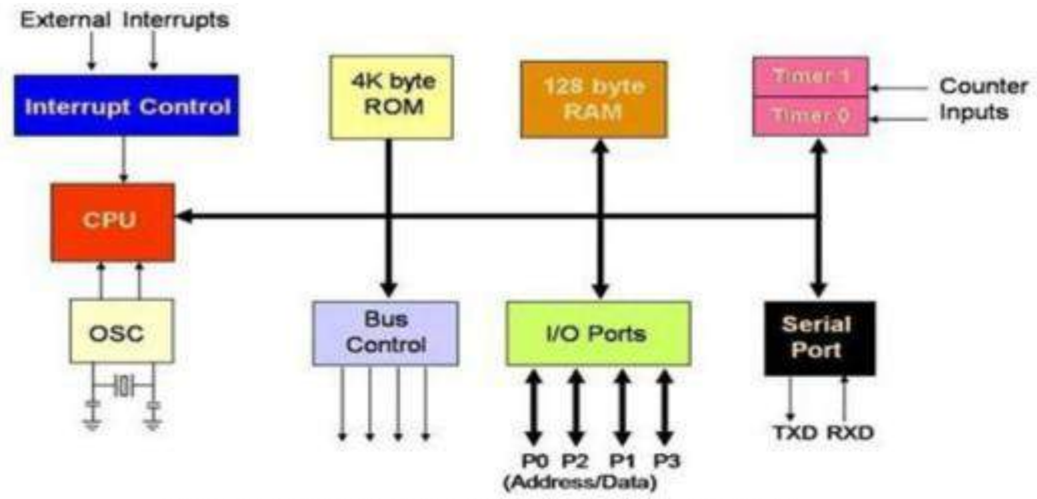
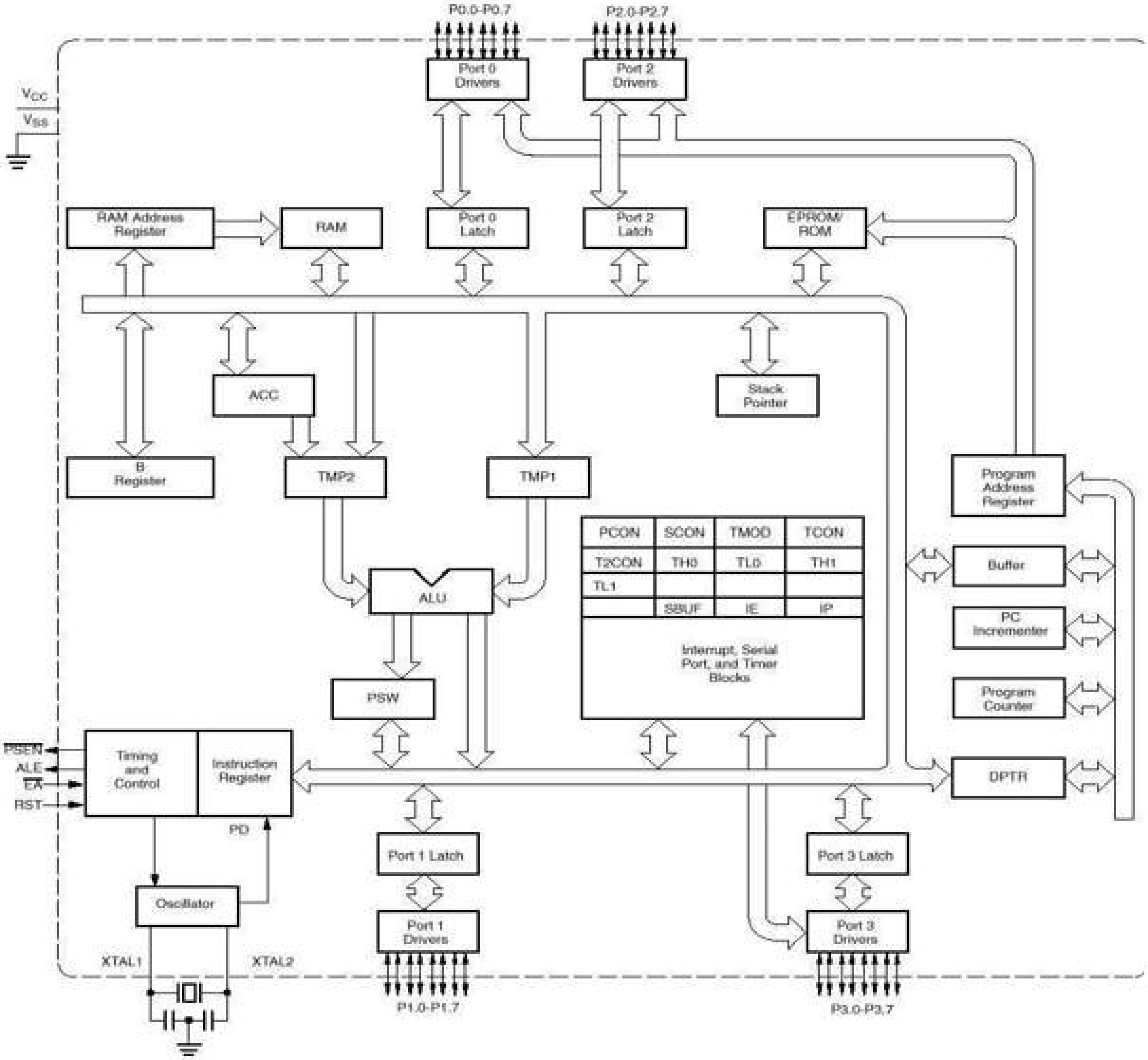


Fig.3. Block Diagram of 8051 Microcontroller.

It consists of an 8-bit ALU, one 8-bit PSW (Program Status Register), A and B registers, one 16-bit Program counter, one 16-bit Data pointer register(DPTR),128 bytes of RAM and 4kB of ROM and four parallel I/O ports each of 8-bit width.

8051 has 8-bit ALU which can perform all the 8-bit arithmetic and logical operations in one machine cycle. The ALU is associated with two registers A & B



A and B Registers: The A and B registers are special function registers which hold the results of many arithmetic and logical operations of 8051.The A register is also called the **Accumulator** and as its name suggests, is used as a general register to accumulate the results of a large number of instructions. By default, it is used for all mathematical operations and also data transfer operations between CPU and any external memory. The B register is mainly used for multiplication and division operations along with A register.MULAB : DIVAB.
It has no other function other than as a location where data may be stored.

The R registers: The "R" registers are a set of eight registers that are named R0, R1, etc. up to and including R7. These registers are used as auxiliary registers in many operations. The "R" registers are also used to temporarily store values.

Program Counter (PC): 8051 has a 16-bit program counter. The program counter always points to the address of the next instruction to be executed. After execution of one instruction the program counter is incremented to point to the address of the next instruction to be executed. It is the contents of the PC that are placed on the address bus to find and fetch the desired instruction. Since the PC is 16-bit width ,8051 can access program addresses from 0000H to FFFFH, a total of 6kB of code.

Stack Pointer Register (SP): It is an 8-bit register which stores the address of the stack top. i.e. the Stack Pointer is used to indicate where the next value to be removed from the stack should be taken from. When a value is pushed onto the stack, the 8051 first increments the value of SP and then stores the value at the resulting memory location. Similarly, when a value is popped off the stack, the 8051 returns the value from the memory location indicated by SP, and then decrements the value of SP. Since the SP is only 8-bit wide it is incremented or decremented by two. SP is modified directly by the 8051 by six instructions: PUSH, POP, ACALL, LCALL, RET, and RETI. It is also used intrinsically whenever an interrupt is triggered.

Stack in 8051 Microcontroller: The stack is a part of RAM used by the CPU to store information temporarily. This information may be either data or an address. The CPU needs this storage area as there are only limited number of registers. The register used to access the stack is called the Stack pointer which is an 8-bit register. So, it can take values of 00 to FF H. When the 8051 is powered up, the SP register contains the value 07.i.e the RAM location value 08 is the first location being used for the stack by the 8051 controller.

There are two important instructions to handle this stack. One is the PUSH and the other is the POP. The loading of data from CPU registers to the stack is done by PUSH and the loading of the contents of the stack back into a CPU register is done by POP.

```
EX:     MOV R6, #35 H
        MOV R1, #21 H
        PUSH6
        PUSH1
```

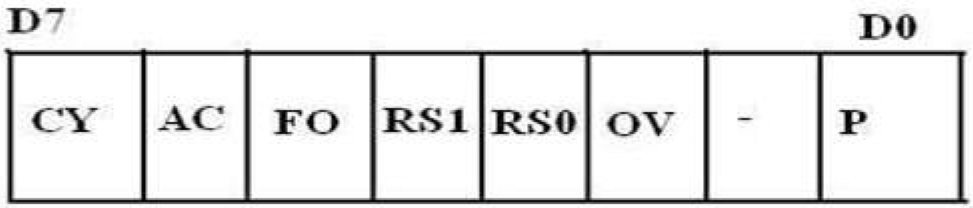
In the above instructions the contents of the Registers R6 and R1 are moved to stack and they occupy the 08 and 09 locations of the stack. Now the contents of the SP are incremented by two and it is 0A

Similarly, POP 3 instruction pops the contents of stack into R3 register. Now the contents of the SP is decremented by 1

In 8051 the RAM locations 08 to 1F (24 bytes) can be used for the Stack. In any program if we need more than 24 bytes of stack, we can change the SP point to RAM locations 30- 7F H. this can be done with the instruction MOV SP, # XX.

Data Pointer Register (DPTR): It is a 16-bit register which is the only user-accessible. DPTR, as the name suggests, is used to point to data. It is used by a number of commands which allow the 8051 to access external memory. When the 8051 accesses external memory it will access external memory at the address indicated by DPTR. This DPTR can also be used as two 8-registers DPH and DPL.

Program Status Register (PSW): The 8051 has an 8-bit PSW register which is also known as Flag register. In the 8-bit register only 6-bits are used by 8051.The two unused bits are user definable bits. In the 6-bits four of them are conditional flags. They are Carry –CY, Auxiliary Carry-AC, Parity-P, and Overflow-OV. These flag bits indicate some conditions that resulted after an instruction was executed.



D7							D0
CY	AC	FO	RS1	RS0	OV	-	P

The bits PSW3 and PSW4 are denoted as RS0 and RS1 and these bits are used the select the bank registers of the RAM location. The meaning of various bits of PSW register is shown below

CY	PSW.7	Carry Flag
AC	PSW.6	Auxiliary Carry Flag
FO	PSW.5	Flag 0 available for general purpose.
RS1	PSW.4	Register Bank select bit 1
RS0	PSW.3	Register bank select bit 0
OV	PSW.2	Overflow flag
---	PSW.1	User definable flag
P	PSW.0	Parity flag .set/cleared by hardware.

The selection of the register Banks and their addresses are given below.

RS1	RS0	Register Bank	Address
0	0	0	00H-07H
0	1	1	08H-0FH
1	0	2	10H-17H
1	1	3	18H-1FH

Memory Organization: The 8051 microcontroller has 128 bytes of Internal RAM and 4kB of on chip ROM. The RAM is also known as Data memory and the ROM is known as program memory. The program memory is also known as Code memory. This Code memory holds the actual 8051 program that is to be executed. In 8051 this memory is limited to 64K. Code memory may be found on-chip, as ROM or EPROM. It may also be stored completely off-chip in an external ROM or, more commonly, an external EPROM. The 8051 has only 128 bytes of Internal RAM but it supports 64kB of external RAM. As the name suggests, external RAM is any random access memory which is off-chip. Since the memory is off-chip it is not as flexible in terms of accessing, and is also slower. For example, to increment an Internal RAM location by 1, it requires only 1 instruction and 1 instruction cycle but to increment a 1-byte value stored in

External RAM requires 4 instructions and 7 instruction cycles. So, here the external memory is 7 times slower

Internal RAM of 8051: This Internal RAM is found on-chip on the 8051. So it is the fastest RAM available, and it is also the most flexible in terms of reading, writing, and modifying its contents. Internal RAM is volatile, so when the 8051 is reset this memory is cleared. The 128 bytes of internal RAM is organized as below.

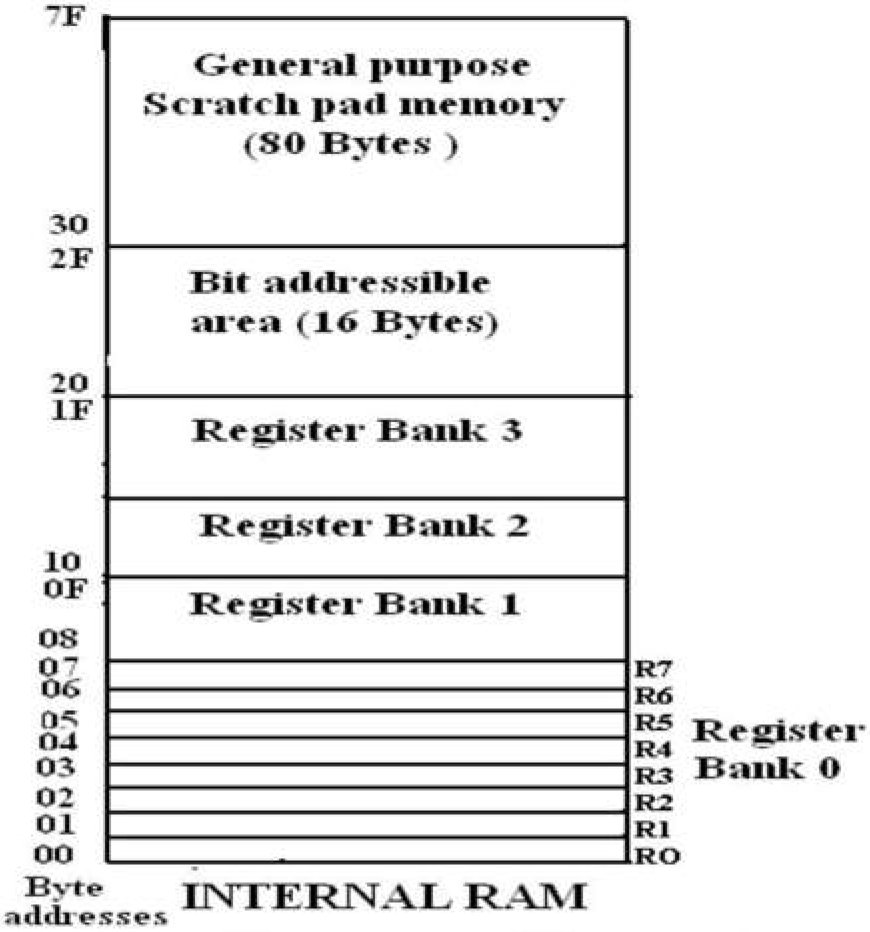
i. Four register banks (Bank0, Bank1, Bank2 and Bank3) each of 8-bits (total 32 bytes). The default bank register is Bank0. The remaining Banks are selected with the help of RS0 and RS1 bits of PSW Register.

ii. 16 bytes of bit addressable area and

iii. 80 bytes of general purpose area (Scratch pad memory) as shown in the diagram below. This area is also utilized by the microcontroller as a storage area for the operating stack.

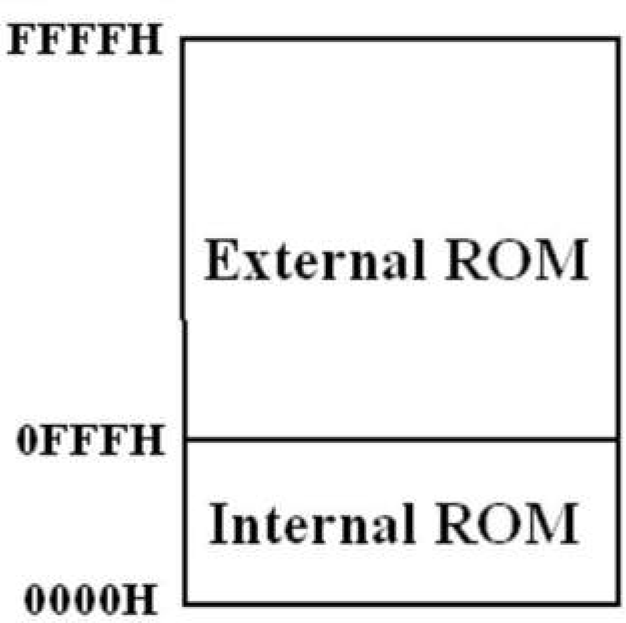
The 32 bytes of RAM from address 00 H to 1FH are used as working registers organized as four banks of eight registers each. The registers are named as R0-R7. Each register can be addressed by its name or by its RAM address.

For EX: MOV A, R7 or MOVR7, #05H



Internal ROM (On –chip ROM): The 8051 microcontroller has 4kB of on chip ROM but it can be extended up to 64kB.This ROM is also called program memory or code memory. The

CODE segment is accessed using the program counter (PC) for opcode fetches and by DPTR for data. The external ROM is accessed when the EA (active low) pin is connected to ground or the contents of program counter exceeds 0FFFH.When the Internal ROM address is exceeded the 8051 automatically fetches the code bytes from the external program memory.



Parallel I /O Ports:

The 8051 microcontroller has four parallel I/O ports, each of 8-bits. So, it provides the user 32 I/O lines for connecting the microcontroller to the peripherals. The four ports are P0 (Port 0), P1(Port1), P2(Port 2) and P3 (Port3). Upon reset all the ports are output ports. In order to make them input, all the ports must be set i.e. a high bit must be sent to all the port pins. This is normally done by the instruction “SETB”.

```
Ex: MOVA, #0FFH            ; A =FF
MOV P0, A                  ; make P0 an input port
```

PORT 0: Port 0 is an 8-bit I/O port with dual purpose. If external memory is used, these port pins are used for the lower address byte address/data (AD0-AD7), otherwise all bits of the port are either input or output.

Dual role of port 0: Port 0 can also be used as address/data bus(AD0-AD7), allowing it to be used for both address and data. When connecting the 8051 to an external memory, port 0 provides both address and data. The 8051 multiplexes address and data through port 0 to save the pins. ALE indicates whether P0 has address or data. When ALE = 0, it provides data D0-D7, and when ALE =1 it provides address

Port 1: Port 1 occupies a total of 8 pins (pins 1 through 8). It has no dual application and acts only as input or output port. Upon reset, Port 1 is configured as an output port. To configure it as an input port, port bits must be set i.e. a high bit must be sent to all the port pins. This is normally done by the instruction “SETB”.

For Ex:

```
MOV A, #0FFH; A=FF HEX
MOV P1, A; make P1 an input port by writing 1‟s to all of its pins
```

Port 2: Port 2 is also an eight-bit parallel port. (pins 21- 28). It can be used as input or output port. Upon reset, Port 2 is configured as an output port. If the port is to be used as input port, all the port bits must be made high by sending FF to the port.

For Ex:

```
    MOV A, #0FFH      ; A=FF hex
    MOV P2, A         ; make P2 an input port by writing all 1‟s to it
```

Dual role of port 2: Port2 lines are also associated with the higher order address lines A8-A15. Port 2 is used along with P0 to provide the 16-bit address for the external memory. Since an 8051 is capable of accessing 64K bytes of external memory, it needs a path for the 16 bits of the address. While P0 provides the lower 8 bits via A0-A7, it is the job of P2 to provide bits A8-A15 of the address. In other words, when 8051 is connected to external memory, Port 2 is used for the upper 8 bits of the 16-bit address, and it cannot be used for I/O operations.

Port3 is also an 8-bit parallel port with dual function. (pins 10 to 17). The port pins can be used for I/O operations as well as for control operations. The details of these additional operations are given below in the table. Upon reset port 3 is configured as an output port. If the port is to be used as input port, all the port bits must be made high by sending FF to the port.

For Ex:

```
   MOV A, #0FFH      ; A= FF hex
   MOV P3, A         ;make P3 an input port by writing all 1s to
                     it
```

Alternate Functions of Port 3: P3.0 and P3.1 are used for the RxD (Receive Data) and TxD (Transmit Data) serial communications signals. Bits P3.2 and P3.3 are meant for external interrupts. Bits P3.4 and P3.5 are used for Timers 0 and 1 and P3.6 and P3.7 are used to provide the write and read signals of external memories connected in 8031 based systems

Table: PORT 3 alternate functions

S.No	Port 3 bit	Pin No	Function
1	P3.0	10	RxD
2	P3.1	11	TxD
3	P3.2	12	
4	P3.3	13	$\overline{\text{INT1}}$
5	P3.4	14	T0
6	P3.5	15	T1
7	P3.6	16	$\overline{\text{WR}}$

$\overline{\text{RD}}$

8	P3.7	17	

8051 Instructions: The process of writing program for the microcontroller mainly consists of giving instructions (commands) in the specific order in which they should be executed in order to carry out a specific task. All commands are known as INSTRUCTION SET. All microcontrollers compatible with the 8051 have in total of 255 instructions These can be grouped into the following categories

1. **Arithmetic Instructions**
2. **Logical Instructions**
3. **Data Transfer instructions**
4. **Boolean Variable Instructions**
5. **Program Branching Instructions**

The following nomenclatures for register, data, address and variables are used while write instructions.

A: Accumulator

B: "B" register C: Carry bit

Rn: Register R0 - R7 of the currently selected register bank

Direct: 8-bit internal direct address for data. The data could be in lower 128bytes of RAM (00 - 7FH) or it could be in the special function register (80 - FFH).

@Ri: 8-bit external or internal RAM address available in register R0 or R1. This is used for indirect addressing mode.

#data8: Immediate 8-bit data available in the instruction.

#data16: Immediate 16-bit data available in the instruction.

Addr11: 11-bit destination address for short absolute jump. Used by instructions AJMP & ACALL. Jump range is 2 Kbyte (one page).

Addr16: 16-bit destination address for long call or long jump.

bit: Directly addressed bit in internal RAM or SFR

- The first part of each instruction, called MNEMONIC refers to the operation an instruction performs (copy, addition, logic operation etc.). Mnemonics are abbreviations of the name of operation being executed.
- The other part of instruction, called OPERAND is separated from mnemonic by at least one whitespace and defines data being processed by instructions. Some of the instructions have no operand, while some of them have one, two or three. If there is more than one operand in an instruction, they are separated by a comma.

1.Arithmetic Instructions: Arithmetic instructions perform several basic arithmetic operations such as addition, subtraction, division, multiplication etc. After execution, the result is stored in the first operand.

For example:

ADD A,R1 - The result of addition (A+R1) will be stored in the accumulator.

Mnemonics	Description
ADD A, Rn	A ← A + Rn
ADD A, direct	A ← A +(direct)
ADD A, @Ri	A ← A +@Ri
ADD A, #data	A A + data
ADDC A, Rn	A A + Rn +C
ADDC A, direct	A A + (direct) +C
ADDC A, @Ri	A A + @Ri +C
ADDC A, #data	A A + data +C
DA A	Decimal adjust accumulator
DIV AB	Divide A by B A quotient B remainder
DEC A	A A-1
DEC Rn	Rn Rn -1
DEC direct	(direct) (direct) - 1
DEC @Ri	@Ri @Ri -1
INC A	A A+1
INC Rn	Rn Rn +1
INC direct	(direct) (direct) + 1
INC @Ri	@Ri @Ri +1
INC DPTR	DPTR DPTR+1
MUL AB	Multiply A by B A low byte (A*B) B high byte (A*B)
SUBB A, Rn	A A - Rn -C
SUBB A, direct	A A - (direct) -C
SUBB A, @Ri	A A - @Ri -C
SUBB A, #data	A A - data -C

2.Logical Instructions: The Logical Instructions are used to perform logical operations like AND, OR, XOR, NOT, Rotate, Clear and Swap. Logical Instruction are performed on Bytes of data on a bit-by-bit basis. Logic instructions perform logic operations upon corresponding bits of two registers. After execution, the result is stored in the first operand

Mnemonics	Description
ANL A, Rn	A ← A AND Rn
ANL A, direct	A ← A AND(direct)
ANL A, @Ri	A ← A AND @Ri
ANL A, #data	A A AND data
ANL direct, A	(direct) (direct) AND A
ANL direct, #data	(direct) (direct) AND data
CLR A	A 00H
CPL A	A A
ORL A, Rn	A A OR Rn
ORL A, direct	A A OR(direct)
ORL A, @Ri	A A OR @Ri
ORL A, #data	A A OR data
ORL direct, A	(direct) (direct) OR A
ORL direct, #data	(direct) (direct) OR data
RL A	Rotate accumulator left
RLC A	Rotate accumulator left through carry
RR A	Rotate accumulator right
RRC A	Rotate accumulator right through carry
SWAP A	Swap nibbles within Accumulator
XRL A, Rn	A A EXOR Rn
XRL A, direct	A A EXOR(direct)
XRL A, @Ri	A A EXOR @Ri
XRL A, #data	A A EXOR data
XRL direct, A	(direct) (direct) EXOR A
XRL direct, #data	(direct) (direct) EXOR data

3.Data Transfer Instructions: Data transfer instructions move the content of one register to another. The register the content of which is moved remains unchanged. If they have the suffix "X" (MOVX), the data is exchanged with external memory.

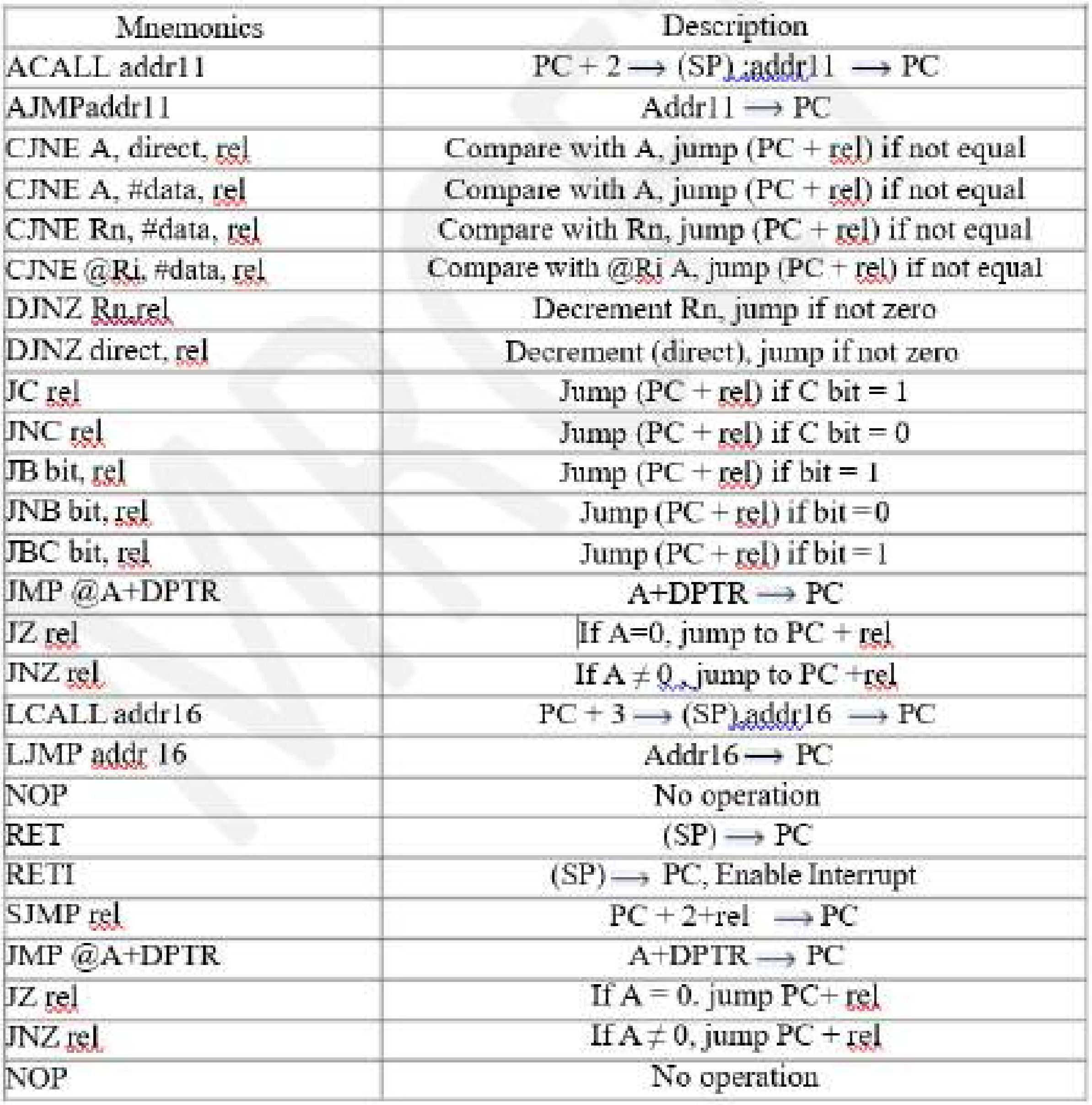
Mnemonics	Description
MOV A, Rn	A ← Rn
MOV A, direct	A ← (direct)
MOV A, @Ri	A ← @Ri
MOV A, #data	A ← data
MOV Rn, A	Rn ← A
MOV Rn, direct	Rn ← (direct)
MOV Rn, #data	Rn ← data
MOV direct, A	(direct) ← A
MOV direct, Rn	(direct) ← Rn
MOV direct1, direct2	(direct1) ← (direct2)
MOV direct, @Ri	(direct) ← @Ri
MOV direct, #data	(direct) #data
MOV @Ri, A	@R iA
MOV @Ri, direct	@Ri (direct)
MOV @Ri, #data	@Ri data
MOV DPTR, #data16	DPTR data16
MOVC A, @A+DPTR	A Code byte pointed by A+DPTR
MOVC A, @A+PC	A Code byte pointed by A +PC
MOVC A, @Ri	A CodebytepointedbyRi8-bitaddress)
MOVX A, @DPTR	A External data pointed by DPTR
MOVX @Ri, A	@Ri A (External data - 8bitaddress)
MOVX @DPTR, A	@DPTR A(External data - 16bitaddress)
PUSH direct	(SP) (direct)
POP direct	(direct) (SP)
XCH Rn	Exchange A with Rn
XCH direct	Exchange A with direct byte
XCH @Ri	Exchange A with indirect RAM
XCHD A, @Ri	Exchange least significant nibble of A with that of indirect RAM

4.Boolean Variable Instructions: Boolean or Bit Manipulation Instructions will deal with bit variables. Similar to logic instructions, bit-oriented instructions perform logic operations. The difference is that these are performed upon single bits.

Mnemonics	Description
CLR C	C -bit0
CLR bit	bit 0
SET C	C 1
SET bit	bit 1
CPL C	C ← $\overline{\text{C-bit}}$
CPL bit	bit ← $\overline{\text{bit}}$
ANL C, /bit	C ← C.$\overline{\text{bit}}$
ANL C, bit	C ← C.bit
ORL C, /bit	C ← C + $\overline{\text{bit}}$
ORL C, bit	C ← C +bit
MOV C, bit	C ← bit
MOV bit, C	Bit ← C

5.Program Branching Instructions:

There are two kinds of branch instructions: **Unconditional jump instructions:** upon their execution a jump to a new location from where the program continues execution is executed. **Conditional jump instructions:** Jump to a new program location is executed only if a specified condition is met. Otherwise, the program normally proceeds with the next instruction.



Mnemonics	Description
ACALL addr11	PC + 2 → (SP) ;addr11 → PC
AJMPaddr11	Addr11 → PC
CJNE A, direct, rel	Compare with A, jump (PC + rel) if not equal
CJNE A, #data, rel	Compare with A, jump (PC + rel) if not equal
CJNE Rn, #data, rel	Compare with Rn, jump (PC + rel) if not equal
CJNE @Ri, #data, rel	Compare with @Ri A, jump (PC + rel) if not equal
DJNZ Rn,rel	Decrement Rn, jump if not zero
DJNZ direct, rel	Decrement (direct), jump if not zero
JC rel	Jump (PC + rel) if C bit = 1
JNC rel	Jump (PC + rel) if C bit = 0
JB bit, rel	Jump (PC + rel) if bit = 1
JNB bit, rel	Jump (PC + rel) if bit = 0
JBC bit, rel	Jump (PC + rel) if bit = 1
JMP @A+DPTR	A+DPTR → PC
JZ rel	If A=0, jump to PC + rel
JNZ rel	If A ≠ 0 , jump to PC +rel
LCALL addr16	PC + 3 → (SP),addr16 → PC
LJMP addr 16	Addr16 → PC
NOP	No operation
RET	(SP) → PC
RETI	(SP) → PC, Enable Interrupt
SJMP rel	PC + 2+rel → PC
JMP @A+DPTR	A+DPTR → PC
JZ rel	If A = 0, jump PC+ rel
JNZ rel	If A ≠ 0, jump PC + rel
NOP	No operation

Memory interfacing to 8051: The system designer is not limited by the amount of internal RAM and ROM available on chip. Two separate external memory spaces are made available by the 16-bit PC and DPTR and by different control pins for enabling external ROM and RAM chips. External RAM, which is accessed by the DPTR, may also be needed when 128 bytes of internal data storage is not sufficient. External RAM, up to 64K bytes, may also be added to any chip in the 8051 family.

Connecting External Memory: Figures shows the connections between an 8051 and an external memory configuration consisting of EPROM and static RAM. The 8051 accesses external RAM whenever certain program instructions are executed. External ROM is accessed whenever the EA (external access) pin is connected to ground or when the PC contains an address higher than the last address in the internal 4K bytes ROM (OFFFh). 8051 designs can thus use internal and external ROM automatically; the 8051, having no internal ROM, must have (EA)' grounded.

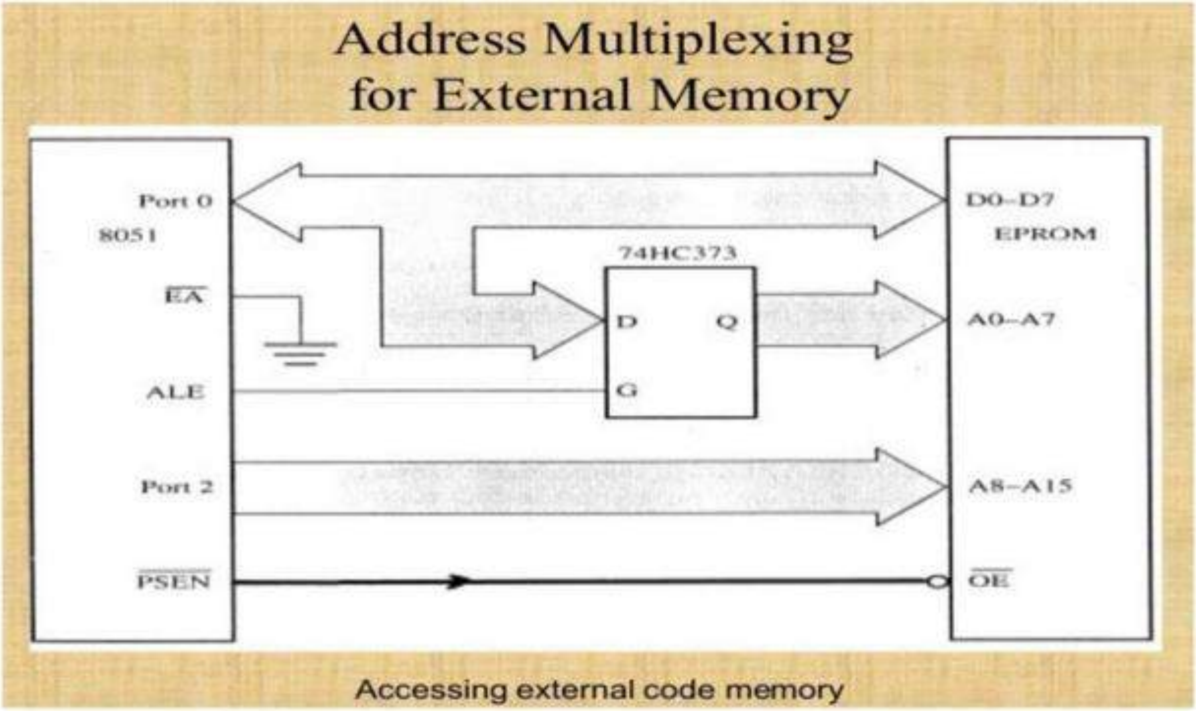
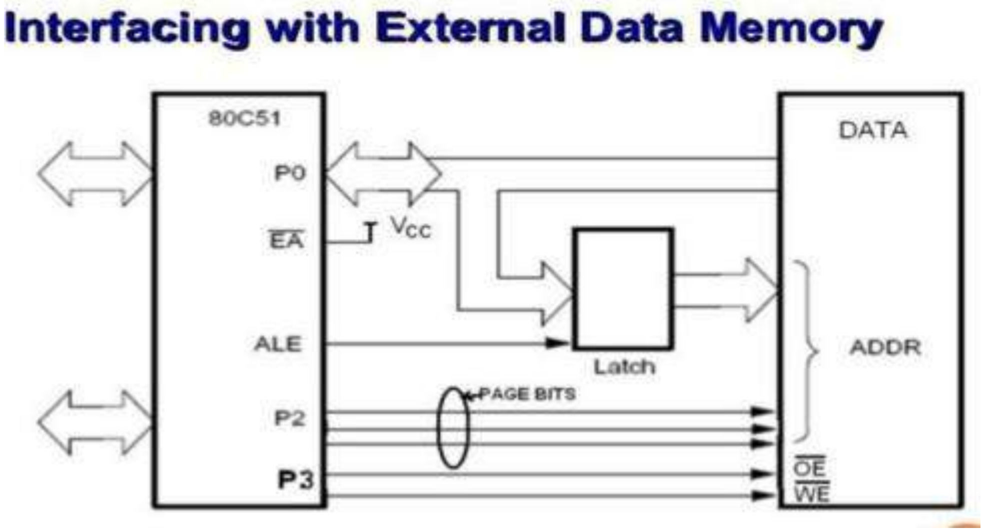


Figure: Interfacing External code memory to 8051



If the memory access is for a byte of program code in the ROM, the (PSEN)'(program store enable) pin will go low to enable the ROM to place a byte of program code on the data bus. If the access is for a RAM byte, the (WR)'(write) or (RD)'(read) pins will go low, enabling data to flow between the RAM and the data bus.
Note that the (WR)' and (RD)' signals are alternate uses for port 3 pins 16 and
17. Also, port 0 is used for the lower address byte and data; port 2 is used for upper address bits. The use of external memory consumes many of the port pins, leaving only port 1 and parts of port 3 for general I/O.

Text Books

1. D. V. Hall, Microprocessors and Interfacing, TMGH, 2nd Edition2006.
2. Advanced Microprocessors and Peripherals – A. K. Ray and K.M. Bhurchandi, TMH, 2nd Edition2006
3. Kenneth. J. Ayala, The 8051 Microcontroller, 3rd Ed., Cengage Learning

UNIT-V

ARM Architecture

ARM → Advanced RISC machine. (processor)

* **ARM fundamentals:**

RISC → Reduced instruction set of computers.

→ It consists large no. of register files.

→ ARM is a first RISC processor

→ The ARM was developed in ACORN computer limited of Cambridge b/wn 1983-1985.

→ It supports low power embedded applications.

→ The ARM is supported for hardware modeling and software testing and assembler, C and C++ compilers, a linker and symbolic debuggers.

Applications:

Video games, Modems, mobile phones. Handy cam.

* **ARM 7TDMI - S processor:**

7TDMI - S

T → 16 bit thumb

D → debug

M → Multiplier

I → Embedded ICE [In circuit emulator]

S → Synthesizable .

→ It is a general purpose 32 bit microprocessor. It supports high performance, low power consumption.

→ It has Von. neumann's architecture. It supports load, store and swap instructions to access data from the memory.

→ It uses 3 stage pipeline architecture to increase the speed of flow of instructions to the processor

→ It performs several operations simultaneously.

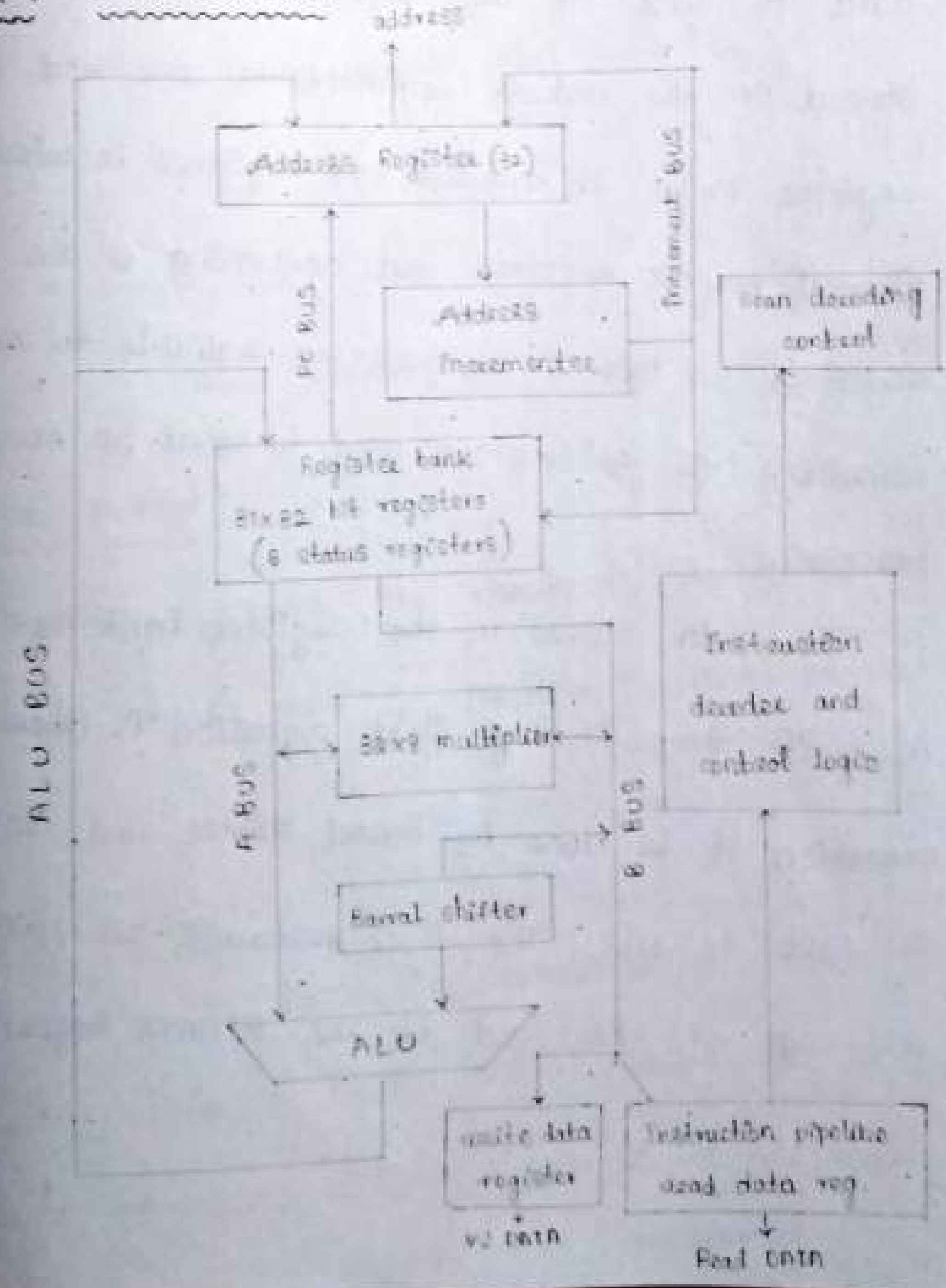
* Three - Stage Pipeline :

Instruction Fetch	→ Thumb	ARM decode Reg. select	Reg. Read	Shift	ALU	Reg. write
Fetch		Decode		Execute		

Features of ARM:

- Large registered file
- Load Store architecture
- Simple addressing modes [fixed length instructions]
- It controls ALU and shift
- Auto increment and auto decrement addressing modes.

ARM Architecture:



→ It consists 32 bit Address register

→ ALU also 32 bit

→ One Barrel shifter

→ One multiplier

** Operation:**

The data stored in the instruction pipeline read data is send to the register banks. The data stored in the address registers is also send to the register banks. It contains 37 registers in which 31 are General purpose registers and remaining 6 are status registers. In this 16 registers are available to user and remaining 15 registers are used to speed up execution processing.

The data stored in the register bank is sent to ALU. If there is any shift operation is there in execution it is done by Barrel shifter and the result is sent to ALU. The ALU performs all operations and o/p will be send to the W DATA register

Register Organisation in ARM :

ARM consists 37 registers in which 31 are general purpose registers of 32 bits and 6 are status registers. In 31 registers only 16 registers are available to user and remaining 15 registers are used to speed up the execution of process. It consists 2 status registers.

(i) CPSR – current program status register

(ii) SPSR – saved program status register

													SP	LR	PC
r0	r1	r2	r3	r4	r5	r6	r7	r8	r9	r10	r11	r12	r13	r14	r15

General purpose register

special purpose register

r13 (stack pointer) :

It is used to store the address of the top of the stack and it is used to perform the PUSH or POP operation.

r14 (Link register) :

This register is holds the address of the next instruction after a branch which is instruction is used to make a sub routine call. It is also called as Return Address Information.

r15 (program counter):

It is used to fetch the address of the next instruction to be executed.

* CPSR [Current Program Status register]

31	30	29	28	27 8	7	6	5	4 0
N	Z	C	V	Reserved	I	F	T	modes

N → negative flag

Z → zero flag

C → carry flag

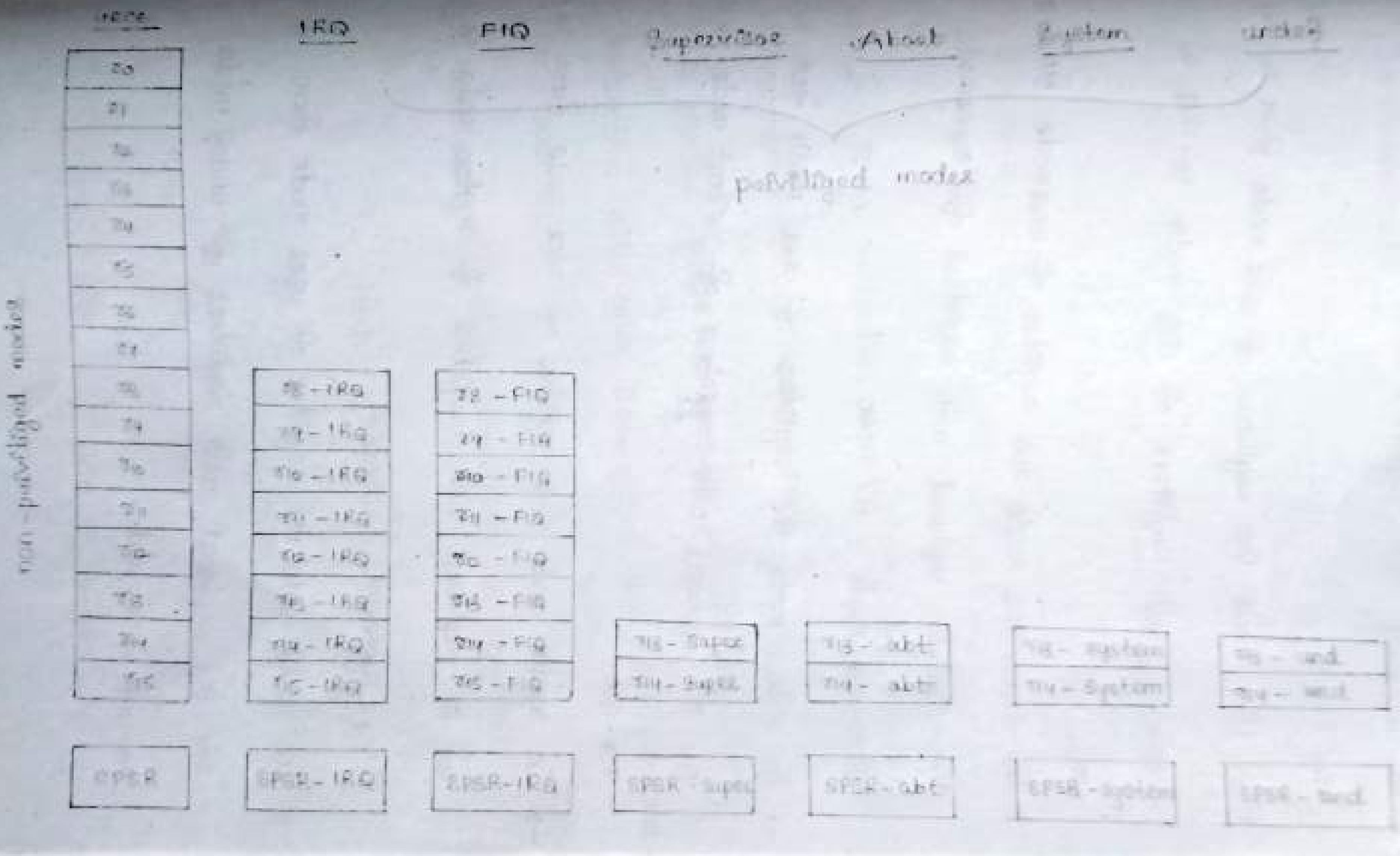
V → overflow flag

I → Interrupt request

F → Fast interrupt request

T → T=0 [Arm instructions]

T=1 [Thumb instructions 16 bit]


user
IRQ
FIQ
Supervisor
Abort
System
undef
privileged modes
non-privileged modes
r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13
r14
r15
CPSR
r8 - IRQ
r9 - IRQ
r10 - IRQ
r11 - IRQ
r12 - IRQ
r13 - IRQ
r14 - IRQ
r15 - IRQ
SPSR - IRQ
r8 - FIQ
r9 - FIQ
r10 - FIQ
r11 - FIQ
r12 - FIQ
r13 - FIQ
r14 - FIQ
r15 - FIQ
SPSR - IRQ
r13 - Super
r14 - Super
SPSR - Super
r13 - abt
r14 - abt
SPSR - abt
r13 - System
r14 - System
SPSR - System
r13 - und
r14 - und
SPSR - und

→ In FIQ mode the registers of user mode from $r_8 - r_{14}$ is replaced with registers of FIQ mode r_8 FIQ to r_{14} FIQ

→ In IRQ mode the registers of user mode from $r_8 - r_{14}$ is replaced with registers of IRQ mode r_8-IRQ to r_{14}-IRQ.

→ In supervisor mode the register of usermode from r_8- $r_{13} - r_{14}$ is replaced with registers of supermode from r_{13} super - r_{14} super

→ In Abort mode the register of user mode from r_{13} and r_{14} is replaced with registers of Abort mode r_{13} abt-r_{14} abt.

→ In system mode the register of user mode from r_{13} and r_{14} is replaced with registers of system mode r_{13} system - r_{14} system.

→ In undef mode the register of user mode from r_{13} and r_{14} is replaced with registers of undef mode r_{13}-und - r_{14} und

3 stage pipeline:

Instruction cycle

N	Fetch	Decode	execute		
N+1		Fetch	Decode	execute	
N+2			Fetch	Decode	Execute

→ Pipeline is a mechanism used by RISC processor to execute instructions at an increase speed

→ This pipeline speeds up execution by fetching next instruction with other instructions are being decoded and executed

Ex: Construct 3 instructions ADD, SUB, COMPARE .

	Fetch	Decode	Execute
N	ADD		
N+1	SUB	ADD	
N+2	CMP	SUB	ADD

* Exceptions, Interrupts and Vectortable:

→ Exceptions are generated by internal and external sources to cause ARM process to handle an event such as an external interrupt or software interrupt

→ The process state just before handling exceptions is normally pe preserved so that original program can be resumed after the completion of exception routine.

→ ARM exceptions may be considered into 3 groups.

1. Exceptions generated as a direct effect of an executing an instructions [software, undefined instructions, pre prefetch aborts]

2. Exceptions generated as an instruction of data aborts.

3. Exceptions generated ~~are~~ externally, reset, IRQ, FIQ

→ ARM architecture supports 7 types of exceptions.

1. RESET

2. Undefined Instruction

3. Software Instruction

4. Prefetch abort (it is fails to access memory)

5. Data abort (it is fails to access data).

6. IRQ (normal interrupt)

7. FIQ (fast interrupt request)

→ When an exception occurs the processor performs following sequence of actions

1. It changes to operating mode corresponding to particular exception.

2. It saves the address of instruction. The entry of exception in r14 of the new mode

3. It saves the old value of CPSR address to SPSR of the new mode

4. It disabled the IRQ by setting bit seven of the CPSR

5. It is disables the FIQ by setting bit six of the CPSR.

* Vector Table :

Exception/interrupts	Name	Address	High address
Reset	RESET	0x00000000	0xffff0000
undefined instruction	UNDEF	0x00000004	0xffff0004
software interrupt	SWI	0x00000008	0xffff0008
Prefetch abort	PABT	0x0000000C	0xffff000C
Data abort	DABT	0x00000010	0xffff0010
IRQ	IRQ	0x00000018	0xffff0018
FIQ	FIQ	0x0000001C	0xffff001C

→ RESET vector is the location of 1st instruction executed by the processor when power is applied

→ Undefined instruction vector is when the processor cannot decode an instruction

→ Software interrupt vector is called when the processor executes a software interrupt instructions.

→ Prefetch abort occurs when the processor attempts to fail the access of memory.

→ Data abort vector occurs when the processor attempts to fail the access of data

* ARM processor families:

ARM Family	Architecture	Pipeline	Operational Frequency	Multiplier
ARM7	Von-Neumann	3 stage	80 mHz	8x32
ARM9	Harvard	5 stage	150 mHz	8x32
ARM10	Harvard	6 stage	260 mHz	16x32
ARM11	Harvard	8 stage	335 mHz	16x32

* ARM Instruction Set:

→ It supports 32-bit of instruction set.

→ It support load/store architecture

→ It consists three operands.

* Data Processing Instructions:

1. move

mov r_1, r_2

$\{r_2 \rightarrow r_1\}$

2. <u>mvn</u> (move negate)

mvn r_1, r_2

$\{\sim r_2 \rightarrow r_1\}$

* Arthemetic Instruction :

1. **Addition**

ADD $\underset{\downarrow \\ \text{destination}}{r_0}, \underbrace{r_1, r_2}_{\text{2 sources}}$

$\{r_1 + r_2 \longrightarrow r_0\}$

2. **Addition with carry :**

ADC r_0, r_1, r_2

$\{r_1 + r_2 + c \longrightarrow r_0\}$

3. **Subtraction :**

SUB r_0, r_1, r_2

$\{r_0 = r_1 - r_2\}$

4. **SBC** (Subtraction with carry)

SBC r_0, r_1, r_2

$\{r_0 = r_1 - r_2 - c\}$

5. **RSB** [reverse subtraction]

RSB r_0, r_1, r_2

$\{r_0 = r_2 - r_1\}$

6. **RSC** [Reverse subtraction with carry]

RSC r_0, r_1, r_2

$\{r_0 = r_2 - r_1 - C\}$

eg: ADD r_0, r_1, #4

$\{r_0 = r_1 + 4\}$

* **Logical Instructions:**

1. **AND**

AND r_0, r_1, r_2

$\{r_0 = r_1 \,\&\, r_2\}$

2. **OR**

ORR r_0, r_1, r_2

$\{r_0 = r_1 / r_2\}$

3. **EOR** [XOR]

EOR r_0, r_1, r_2

$\{r_0 = r_1 \wedge r_2\}$

4. **BIC** [logic bit clear]

BIC r_0, r_1, r_2

$\{r_0 = r_1 \& r_2 \sim r_2\}$

* Comparison Instructions:

The comparison instructions are used to compare or test a register with a 32 bit value. the result is not stored in any register

1. CMP r_1 r_2

   $\{r_1 - r_2\}$

2. **CMN** [compare negate]

   CMN r_1, r_2

   $\{r_1 + r_2\}$

3. **TEQ** [Test for equality]

   TEQ r_1, r_2

   $\{r_1 \wedge r_2\}$

4. **TST** [Test]

   TST r_1, r_2

   $\{r_1 \& r_2\}$

* Shift and Rotate instructions:

Logical

- LSL
- LSR

Arithmetic

- ASR
- ASL

1. LSL (logical left shift)

LSL r_0, r_1, #2

It performs logical left shift of r_1 by 2 times.

eg: LDR R_2 = 0x0000 0030

↓

load register

mov R_0, R_2, LSL #2

R_2 = 0x 0000 0000 0000 0000 0000 0000 0011 0000

LSL #2

① 0x0000 0000 0000 0000 0000 0000 0110 0000

② 0x0000 0000 0000 0000 0000 0000 1100 0000

R_0 = 0x000000C0

Eg: LDR R1=0x11223344

mov R2, R1, LSL #4

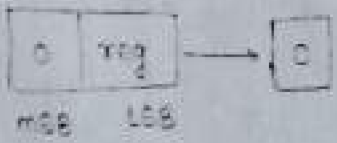
① 0x 12 23 34 40

② 0x 22 33 44 00

③ 0x 23 34 40 00

④ 0x 33 44 00 00

2. LSR [logical right shift]



LSR r0, r1, #immediate.

LSR r0, r1, #2

Eg: LDR r1= 0x 11 22 33 44

mov r0, r1, LSR. #2

① 0x 01 12 23 34

② 0x 00 11 22 33

R0=0x00 11 22 33

LSR R_1, R_2

{ $R_1 = R_1 >> R_2$ }

{right shift}

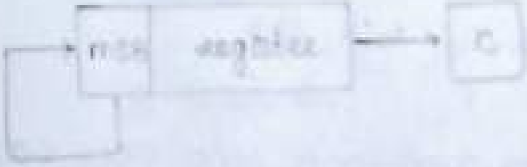
right shift R_1 by R_2 times and the result is stored in R_1

LSL R_1, R_2

{ $R_1 = R_1 << R_2$ }

3 ASR [Arithmetic right shift]

ASR r_1, r_2, #4



$\boxed{r_1 = r_2 >> \text{immediate}}$

eg: LDR R_2 = 0x A0000030

mov R_0, R_2, ASR #2

R_2 = 0x 1010 0000 0000 0000 0000 0000 0011 0000

① = 0x 1101 0000 0000 0000 0000 0000 0001 1000

② = 0x 1110 1000 0000 0000 0000 0000 0000 1100

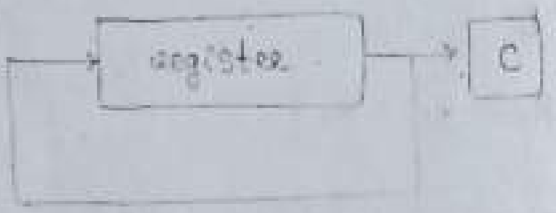
$\boxed{R_0 = \text{0x E800000C}}$

4. ASL [Arthimetic left shift]

ASL R_0, R_1, #3

same operation as LSL

5. ROR [Rotate right]



ROR R_1, R_2, #3.

eg:

LDR R_2 = 0x 00000330

mov R_1, R_2, ROR, #3

R_2 =

⓪ 0 x 0000 0000 0000 0000 0000 0011 0011 0000

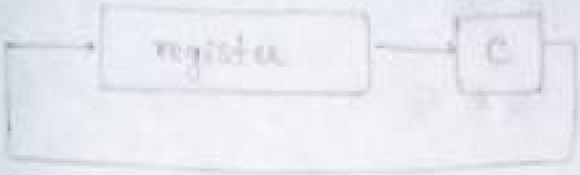
① 0 x 0000 0 0 0 0 0001 1001 1000 0

② 0 x 0000 0 0 0 0 0 1100 1100 0

③ 0 x 0000 0 0 0 0 0 0110 0110 0

0x 000 00066.

6. RRX [rotate right extended by 1 bit only]



Branch Instructions:

Branch instructions changes the normal flow of the executions of main program or it is used to call a subroutine program.

ex: B label
↓
Branch

```
B forward
ADD, r1, r2. #4
ADD, r0, r6, #2
ADD, r0, r7 , #4
forward: SUB r1, r2 #4
```

Branch with link:

```
BC subroutine
CMP r1 . #5
mov eq r1, #0

subroutine
mov PC, lr.
```

Multiplication:

MUL r_0, r_1, r_2

$r_4 = r_3 * r_2$

MLA r_4, r_3, r_2, r_1

$r_4 = r_3 * (r_2 + r_1)$

* Load and Store Instructions:

1. Single register transfer
2. Multi register transfer

(i) LDR

(ii) LDR r_0, [r_1]

load → memory → data

store → data → memory

memory ↓

0x00000005	
0x00000004	EA
0x00000003	78
0x00000002	56
r_1 = 0x00000001	34

r_0 = 0XEA785634

Pre-Index

LDR r1 [r0, #4]

r1 ⟷ r0+4

r0 ⟵ unchanged

r0 = 0x20000001

20000009	99
20000008	88
20000007	79
20000006	78
20000005	56
20000004	44
20000003	33
20000002	32
r0 = 0x20000001	23

r1 = 0x88797656.

Post Index

LDR r1, r0, #4

r1 ⟵ r0

r0 ⟵ r0+4

40003005	88
40003004	77
40003003	66
20003002	55
20003001	44
r0 = 0x40003000	33

r0 = 0x20003000

After execution

r0 → 0x 20003004

r1 → 0x 66554433

pre index with up data

LDR r1 [r0, #4]!

r1 ← r0+4

r0 ← r0+4

EA
10
99
88
77
66
55
44
33

After execution

r1 = 0x 10998877

r0 = 0x200030004

r0 = 0x20003000

* Multiple data transfer load instruction:

LDMIA → Load multiple instruction increment after

LDMIB → Load multiple instruction increment before

LDMDA → Load multiple instruction decrement after

LDMDB → Load multiple instruction decrement before

* LDMIA: R0. {R1, R2, R3}

memory	data
0x00000013	0x000000007
0x00000014	0x000000006
0x00000010	0x000000005
0x0000000C	0x000000004
0x00000008	0x000000003
0x00000004	0x000000002
0x00000000	0x000000001

Initially

r0 = 0x00000000

R1 = 0x00000000

R2 = 0x00000000

R3 = 0x00000000

After execution

R0 = 0x000000

R1 = 0x00000001

R2 = 0x00000002

R3 = 0x00000003

R0 = 0x0000000C

* LDMIB:

LDMIB R0, {R1, R2, R3}

(or)

LDMIB R0! {R1, R2, R3}

(or)

LDMIB R0! {R1-R3}

Initially: R0 = 0x0000001C

R1 = 0x00000000

R2 = 0x00000000

.

R8 = 0x00000000

0x00000030	0x00000088
0x00000028	0x00000077
0x00000024	0x00000066
0x00000020	0x00000055
0x0000001C	0x00000044
0x00000018	0x00000033
0x00000014	0x00000022
0x00000010	0x00000011
0x00000008	0x00000055
0x00000004	0x00000044
0x00000000	0x00000033

after execution

R0 = 0x00000020

R1 = 0x00000055

R2 = 0x00000066

R3 = 0x00000077

* LDMIA : Ro, {R1, R2, R3}

memory	data
0x0000001C	0X000000F6
0x00000018	0X000000EA
0x00000017	0x00000016
0x00000010	0X00000013
0X0000000C	0X00000011
0X00000004	0X00000010
0X00000000	0X00000004

Initially

Ro = 0X0000000C

R1 = 0X00000000

R2 = 0X00000000

R3 = 0X00000000

after execution

Ro = 0X 00000008

R1 = 0x 00000001

R2 = 0X 00000010

R3 = 0x 00000005

* LDMDB : Ro , {R1, R2, R3}

Initially

Ro = 0X 00000014

R1 = 0X 00000000

R2 = 0X 0000000

R3 = 0X 0000000

After execution

Ro = 0X 00000010

R1 = 0X00000013

R2 = 0X00000011

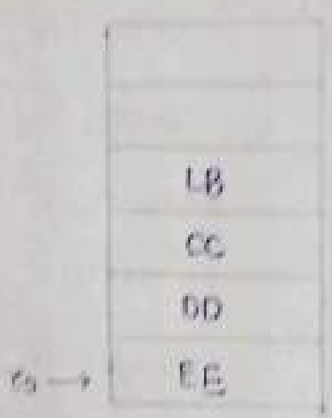
R3 = 0X00000010

- Store Instructions :

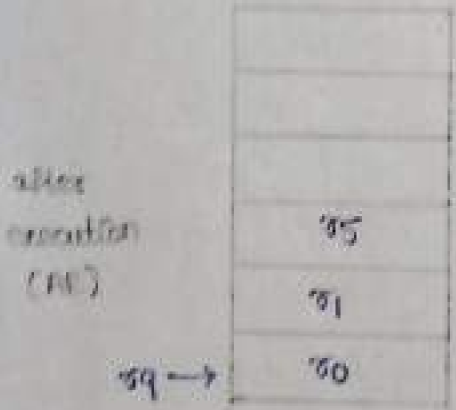
STR r1,[r6]

The data transfer from register to location address

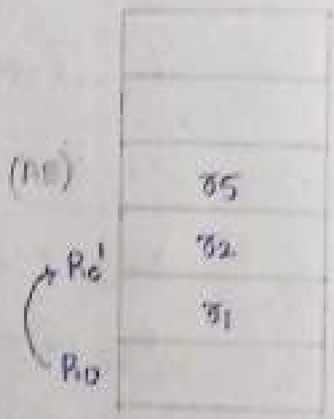
r1= 0x1BCCDDEE



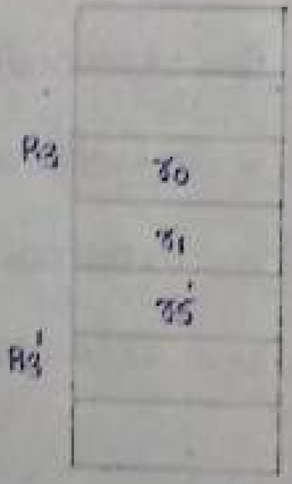
STMIA R9! [r0,r5,r1]



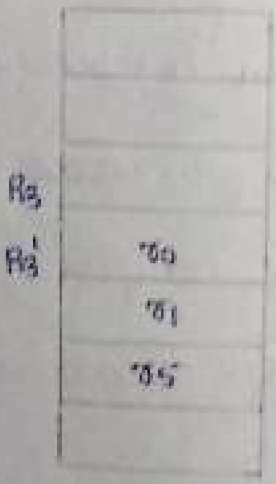
STMIB R0! [r1,r2,r5]



STMDA R3! [r0,r1,r5]



STMDB R3! [r0,r1,r5]



* Stack Operations:

STMFA : Store multiple register in full stack in ascending order

STMFD: Store multiple register in full stack in descending order

STMEA: Store multiple register on empty stack in ascending order

STMED : Store multiple register on empty stack in descen -ding order

LDMFA : Load multiple register from full stack in ascendi -ng order

LDMFD : Load multiple register from full stack in descending order

STMFA SP! . {R0-R10} (or) STMFA R_{13}! {R_0 - R_{10}}

	Address	data
	7FFEH	
	7FFFH	
r13 SP	8000H	XX
↳SP	8001H	R_0
		R_1
		⋮ R_{10}

STMFD r13! {R0-R10}

	Address	Data
	7FFEH	
SP-1	7FFFH	R0
SP	8000H	XX

STMEA r13! {r0-r10}

SP	8000H	XX
SP+1	8001H	R0

first store the value after increment the stack pointer.

STMED R13, {R0-R10}

SP'	1FFFH	R0
SP	8000H	XX

first store the values and then decrease the SP.

Eg: LDMFA R13, {R0 - R10}

SP'	7FFFH	R10
SP	8000H	XX
	8001H	

LDMFD R13! {R0 - R10}

SP	8000H	XX
SP+1		R10

* Swap Instructions:

SWP - swap a word b/w memory & register

SWPB - swap a byte b/w memory & register

SWP r0, r1 [r2]

BE : mem 32 [0x9000] = 0x 12345678

r0 = 0x 00000000

r1 = 0x11112222

r2 = 0x00009000

AE: r0 = 0x12345678

r1 = 0x11112222

r2 = 0x00009000

mem 32 [0x9000] = 0x11112222

* Software Interrupt Instructions :

It is used in supervisor mode

Assume:

SWI is user mode

Pre execution (user)

CPSR = nzcvq iFt - user

PC = 0x00008000

lr = r14 = 0x003ffff

SWI 0x123456

Post execution:

It is changed to supervisor mode.

CPSR = nzcqift - svc

SPSR = nzCaqift - user

PC → SWI

PC = 0x00000008

lr = 0x00008004.

* Introduction to Thumb instruction set:

Thumb encodes a subset of 32-bit ARM instructions into a 16-bit instruction set space. Since Thumb has higher performance than ARM on a processor with a 16-bit data bus, but lower performance than ARM on a 32-bit data bus, use Thumb for memory-constrained systems. Thumb has higher code density - the space taken up in a memory by an executable program - than ARM. For memory-constrained embedded systems. for example, mobile phones and PDA's. code density is very important. Cost pressures also limit memory size, width and speed.

Thumb execution is flagged by the T bit (cpsr[5]) in the CPSR. A thumb implementation of the same code takes up around 30% less memory than the equivalent ARM implementation. Even though the thumb implementation uses more instructions: the overall memory foot print is reduced. Code density was the main driving force for the Thumb instruction set. Because it was also designed as a compiler target, rather than for hand-written assembly code. Below example explains the difference between ARM and Thumb code.

ARM code

```
ARM divide
; IN: r0 (value), r1 (divisor)
; OUT: r2 (modulus), r3 (divide)
        mov  r3, #0
 loop
        SUBS   r0, r0, r1
        ADDGE  r3, r3, #1
        BGE    loop
        ADD    r2, r0, r1
      5x4 = 20 bytes.
```

Thumb code

```
Thumb divide
; IN: r0 (value), r1 (divisor)
; OUT: r2 (modulus), r3 (divide)
        mov r3, #0
   loop
        ADD  r3, #1
        SUB  r0, r1
        BGE  loop
        SUB  r3, #1
        ADD  r2, r0, r1
      6x2 = 12 bytes.
```

From the above example it is clear that Thumb code is more denser than ARM code. Exceptions generated during Thumb execution switch to ARM execution before the executing the exception handler. The state of T bit is preserved in SPSR, and the LR of exception mode is set so that normal return instructions performs correctly, regardless of whether exception occured during ARM or Thumb execution.

In Thumb state, all the registers can be accessed. Only the low registers r_0 to r_7 can be accessed. The higher registers r_8 to r_{12} are only accessible with MOV, ADD or CMP instructions. CMP and all the data processing instructions that operate on low registers update condition flags in the CPSR.

The list of registers and their accessibility in Thumb mode are shown in following table

S.no	Registers	Access.
1.	r0 - r7	Fully accessible
2.	r8 - r12	only accessible by MOV, ADD and CMP
3.	r13 SP	limited accessibility
4.	r14 lr	limited accessibility
5.	r15 PC	limited accessibility
6.	CPSR	only indirect address.
7	SPSR	no access.

ARM thumb interworking is the method of linking ARM and thumb code together for both assembly and C/C++. It handles the transition b/w 2 states. To call a Thumb routine from an ARM architecture routine, the core has to change state. This is done with the T bit of CPSR. The BX and BLX branch instructions cause a switch b/wn ARM and Thumb state while branching to a routine. The BX lr instructions return from a routine, also with a state switch if necessary.

The data processing instructions manipulate data within registers. They include move instructions, arithmetic instructions, shifts, logical instructions, comparison instruct-ions and multiply instructions. The thumb data processing instructions are a subset of ARM data processing instructions.

ADC : add two 32-bit values and carry Rd= Rd+Rm+Cflag

ADD : add two 32-bit values Rd= Rn + Immediate

Rd= Rd + Immediate

Rd= Rd+ Rm.

AND: logical bitwise AND of two 32-bit values Rd=Rd&Rm

ASR: arithmetic shift right Rd= Rm- Immediate

Cflag = Rm [Immediate -1]

Rd = Rd- Rs, Cflag = Rd [RS-1]

BIC: logical bit clear [AND not] of two 32 bit

Rd= Rd AND NOT (Rm) values.

CMN: compare -ve two 32-bits values Rn+Rm sets flags.

CMP: compare two 32-bit integers Rn- Immediate sets flags Rn- Rm sets flags.

EOR: logical exclusive OR of two 32-bit values Rd = Rd EOR Rm

LSL: logical shift left Rd = Rm- immediate

C = flag Rm [32 - immediate]

Rd = Rd - Rs, C flag = Rd [Rs-1]

MOV: move a 32-bit value into a register Rd = immediate

Rd = Rn

Rd = Rm

MUL: multiply two 32-bit values Rd = (Rm * Rd) [31:0]

MVN: move the logical not of a 32-bit value into a register Rd = NOT (Rm).

NEG: negate a 32-bit value Rd = 0-Rm

ORR: logical bitwise OR of two 32-bit values Rd = Rd OR Rm

ROR: rotate right a 32-bit value

Rd = Rd. RIGHT- ROTATE Rs,

c flag = Rd [Rs-1]

SBC: subtract with a carry a 32 bit value

Rd = Rd - Rm - NOT (c flag)

SUB: subtract two 32-bit registers values

Rd = Rn - immediate

Rd = Rd - immediate

Rd = Rd - Rm

SP = SP - (immediate - 2)

TST: test bits of a 32 bit value Rn AND Rm sets flags.

Embedded Systems

UNIT-4

EMBEDDED FIRMWARE

B. Deepika Rathod,

Department of ECE

R22CSE3147) EMBEDDED SYSTEMS

4	**Embedded Firmware**	Reset Circuit, Brown-Protection Circuit
		Oscillator Unit, Real Time Clock
		Watchdog Timer
		Embedded Firmware Design
		Approaches and Development Languages

Embedded Firmware

Introduction to Embedded Firmware

Firmware is a type of software that's embedded into hardware devices to control their functions. It's a low-level program that acts as an intermediary between the hardware and higher-level software.

How does firmware work?

- Firmware boots the device and provides instructions for it to function.
- It controls the processor and peripherals within the device.
- It ensures that the hardware operates as intended.

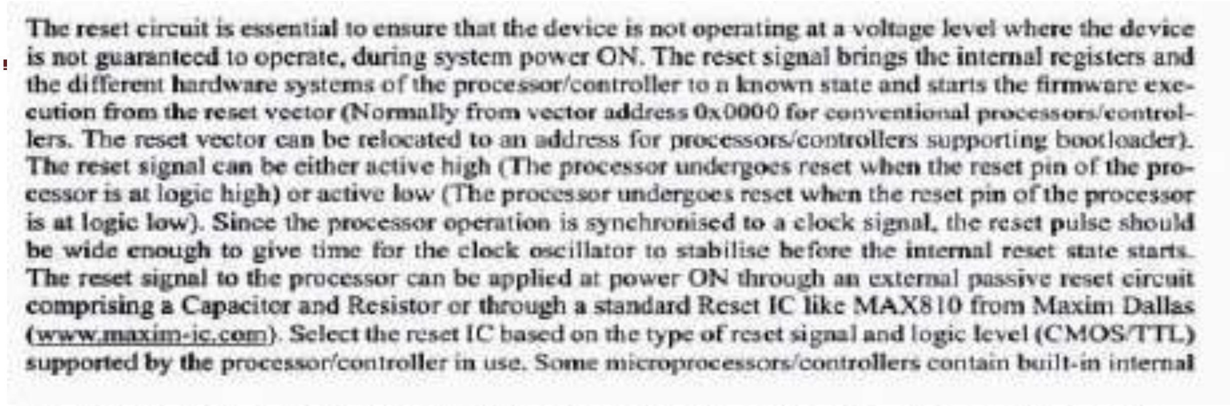
Where is firmware stored?

- Firmware is usually stored in non-volatile memory, such as read-only memory (ROM), electrically erasable programmable read-only memory (EEPROM), or flash memory.
- Firmware stored in ROM or EEPROM is usually low-level and can't be updated.
- Firmware stored in flash memory is usually high-level and can be updated.

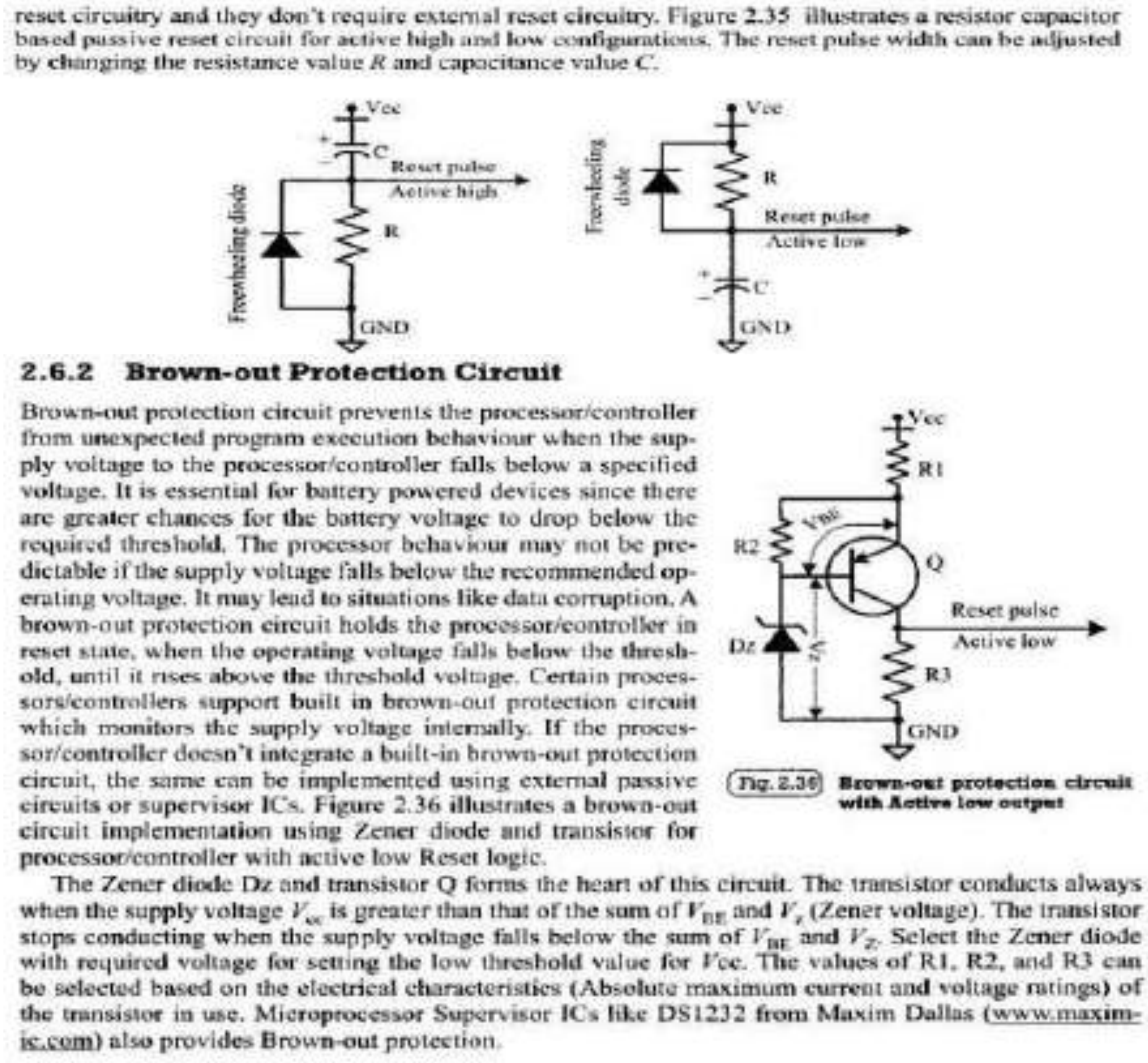
Examples of devices that use firmware

Cameras, Mobile phones, Network cards, Optical drives, Printers, Routers, Scanners, Television remotes, and Washing machines.

Reset Circuit & Brown-Protection Circuit



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



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

Vcc
C
Reset pulse
Active high
Freewheeling diode
R
GND
Vcc
Freewheeling diode
R
Reset pulse
Active low
C
GND

2.6.2 Brown-out Protection Circuit

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

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

The Zener diode Dz and transistor Q forms the heart of this circuit. The transistor conducts always when the supply voltage V_{cc} is greater than that of the sum of V_{BE} and V_z (Zener voltage). The transistor stops conducting when the supply voltage falls below the sum of V_{BE} and V_Z. Select the Zener diode with required voltage for setting the low threshold value for *Vcc*. The values of R1, R2, and R3 can be selected based on the electrical characteristics (Absolute maximum current and voltage ratings) of the transistor in use. Microprocessor Supervisor ICs like DS1232 from Maxim Dallas (www.maxim-ic.com) also provides Brown-out protection.

Oscillator Unit & Real Time Clock

2.6.3 Oscillator Unit

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

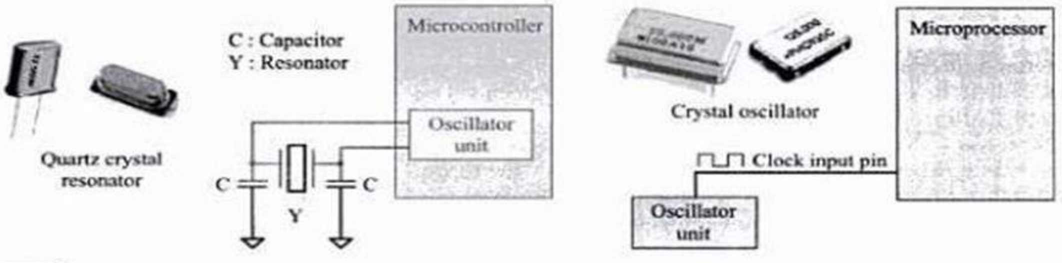


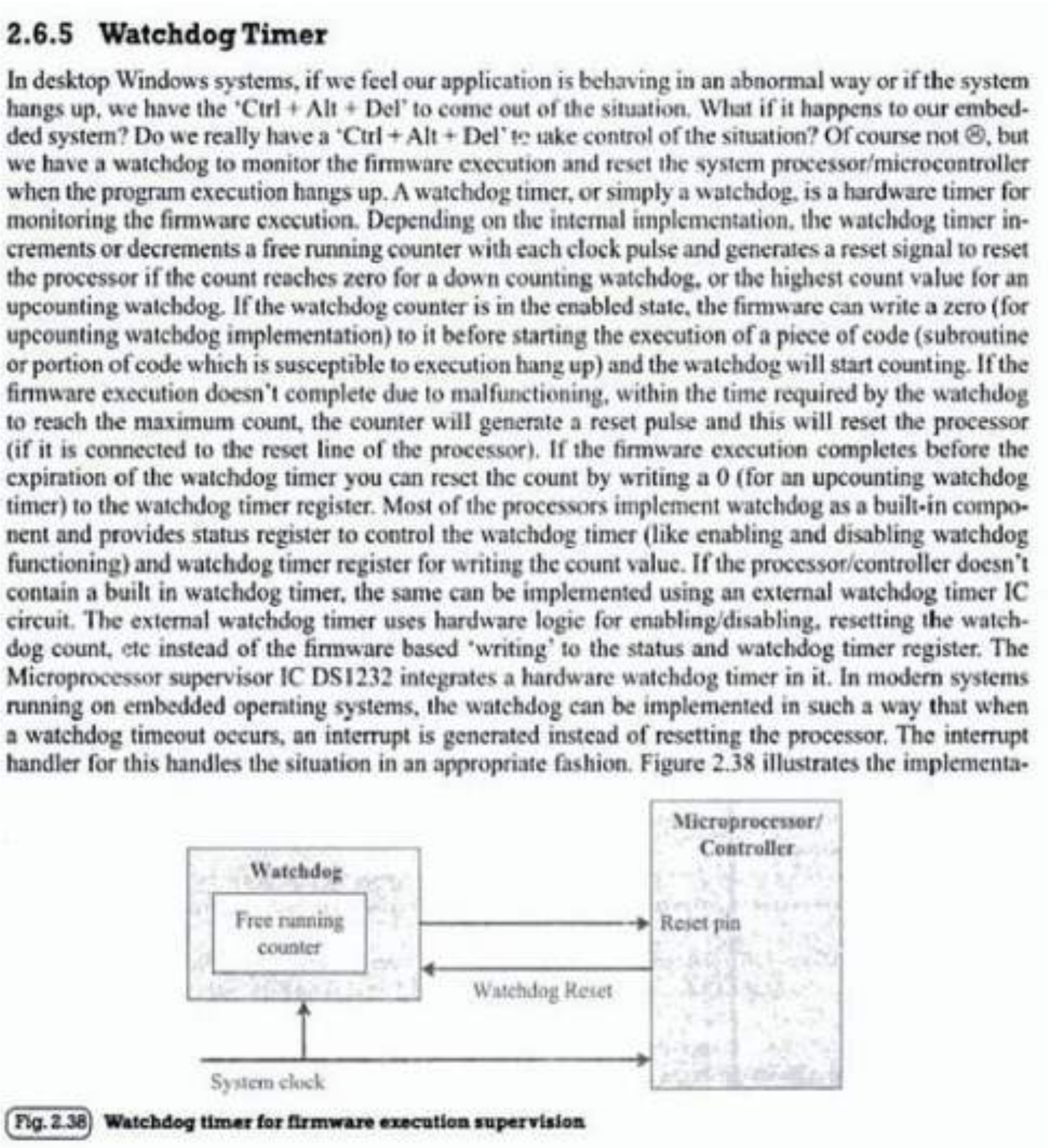
Fig. 2.37 **Oscillator circuitry using quartz crystal and quartz crystal oscillator**

2.6.4 Real-Time Clock (RTC)

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

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

Watchdog Timer



2.6.5 Watchdog Timer

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

Fig. 2.38 **Watchdog timer for firmware execution supervision**

Embedded Firmware

Introduction:

- The control algorithm (Program instructions) and or the configuration settings that an embedded system developer dumps into the code (Program) memory of the embedded system.
- It is an un-avoidable part of an embedded system.
- The embedded firmware can be developed in various methods like
  - Write the program in high level languages like Embedded C/C++ using an Integrated Development Environment (The IDE will contain an editor, compiler, linker, debugger, simulator etc. IDEs are different for different family of processors/controllers.
  - Write the program in Assembly Language using the Instructions Supported by your application's target processor/controller

Embedded Firmware Design & Development:

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

- The embedded firmware is the master brain of the embedded system.

- The embedded firmware imparts intelligence to an Embedded system.

- It is a onetime process and it can happen at any stage.

- The product starts functioning properly once the intelligence imparted to the product by embedding the firmware in the hardware.

- The product will continue serving the assigned task till hardware breakdown occurs or a corruption in embedded firmware.

- In case of hardware breakdown, the damaged component may need to be replaced and for firmware corruptions the firmware should be re-loaded, to bring back the embedded product to the normal functioning.

- The embedded firmware is usually stored in a permanent memory (ROM) and it is non alterable by end users.

- Designing Embedded firmware requires understanding of the particular embedded product hardware, like various component interfacing, memory map details, I/O port details, configuration and register details of various

hardware chips used and some programming language (either low level Assembly Language or High level language like C/C++ or a combination of the two)

- The embedded firmware development process starts with the conversion of the firmware requirements into a program model using various modeling tools.

- The firmware design approaches for embedded product is purely dependent on the complexity of the functions to be performed and speed of operation required.

- There exist two basic approaches for the design and implementation of embedded firmware, namely;

    - **The Super loop based approach**

    - **The Embedded Operating System based approach**

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

1. Embedded firmware Design Approaches – The Super loop:

- The Super loop based firmware development approach is Suitable for applications that are not time critical and where the response time is not so important (Embedded systems where missing deadlines are acceptable).

- It is very similar to a conventional procedural programming where the code is executed task by task

- The tasks are executed in a never ending loop.

- The task listed on top on the program code is executed first and the tasks just below the top are executed after completing the first task

A typical super loop implementation will look like:

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

5. :

6. :

7. Execute the last defined task

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

The ‘C’ program code for the super loop is given below void main

```
()

{

Configurations (); Initializations
();

while (1)

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

:

Task n ();

}

}
```

Pros:

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

Simple and straight forward design Reduced

memory footprint

Cons:

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

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

Enhancements:

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

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

- The embedded device contains an Embedded Operating System which can be one of:
  - **A Real Time Operating System (RTOS)**
  - **A Customized General Purpose Operating System (GPOS)**
- The Embedded OS is responsible for scheduling the execution of user tasks and the allocation of system resources among multiple tasks
- It Involves lot of OS related overheads apart from managing and executing user defined tasks
- Microsoft® Windows XP Embedded is an example of GPOS for embedded devices
- Point of Sale (PoS) terminals, Gaming Stations, Tablet PCs etc are examples of embedded devices running on embedded GPOSs
- '*Windows CE*', '*Windows Mobile*','*QNX*', '*VxWorks*', '*ThreadX*', '*MicroC/OS-II*', '*Embedded Linux*', '*Symbian*' etc are examples of RTOSs employed in Embedded Product development
- Mobile Phones, PDAs, Flight Control Systems etc are examples of embedded devices that runs on RTOSs

Embedded firmware Development Languages/Options

- **Assembly Language**
- **High Level Language**
  - Subset of C (Embedded C)
  - Subset of C++ (Embedded C++)
  - Any other high level language with supported Cross-compiler

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

1. Embedded firmware Development Languages/Options – Assembly Language

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

- ‘M***achine language***’ is a processor understandable language
- Machine language is a binary representation and it consists of 1s and 0s
- Assembly language and machine languages are processor/controller dependent
- An Assembly language program written for one processor/controller family will not work with others
- Assembly language programming is the process of writing processor specific machine code in mnemonic form, converting the mnemonics into actual processor instructions (machine language) and associated data using an assembler

- The general format of an assembly language instruction is an Opcode followed by Operands

- The Opcode tells the processor/controller what to do and the Operands provide the data and information required to perform the action specified by the opcode

- It is not necessary that all opcode should have Operands following them. Some of the Opcode implicitly contains the operand and in such situation no operand is required. The operand may be a single operand, dual operand or more

The 8051 Assembly Instruction

```
MOV A, #30
```

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

```
01110100 00011110
```

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

- Assembly language instructions are written one per line
- A machine code program consists of a sequence of assembly language instructions, where each statement contains a mnemonic (Opcode + Operand)
- Each line of an assembly language program is split into four fields as:

LABEL OPCODE OPERAND COMMENTS

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

  ❖ A memory location, address of a program, sub-routine, code portion etc.

  ❖ The maximum length of a label differs between assemblers. Assemblers insist strict formats for labeling. Labels are always suffixed by a colon and begin with a valid character. Labels can contain number from 0 to 9 and special character _ (underscore).

```
;################################################################
;     SUBROUTINE FOR GENERATING DELAY
;     DELAY PARAMETR PASSED THROUGH REGISTER R1
;     RETURN VALUE NONE, REGISTERS USED: R0,  R1
;################################################################
#####       DELAY:       MOV R0, #255        ; Load Register R0 with 255

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

            RET                          ; Return to calling program
```

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

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

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

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

- The software utility called ‘Assembler’ performs the translation of assembly code to machine code

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

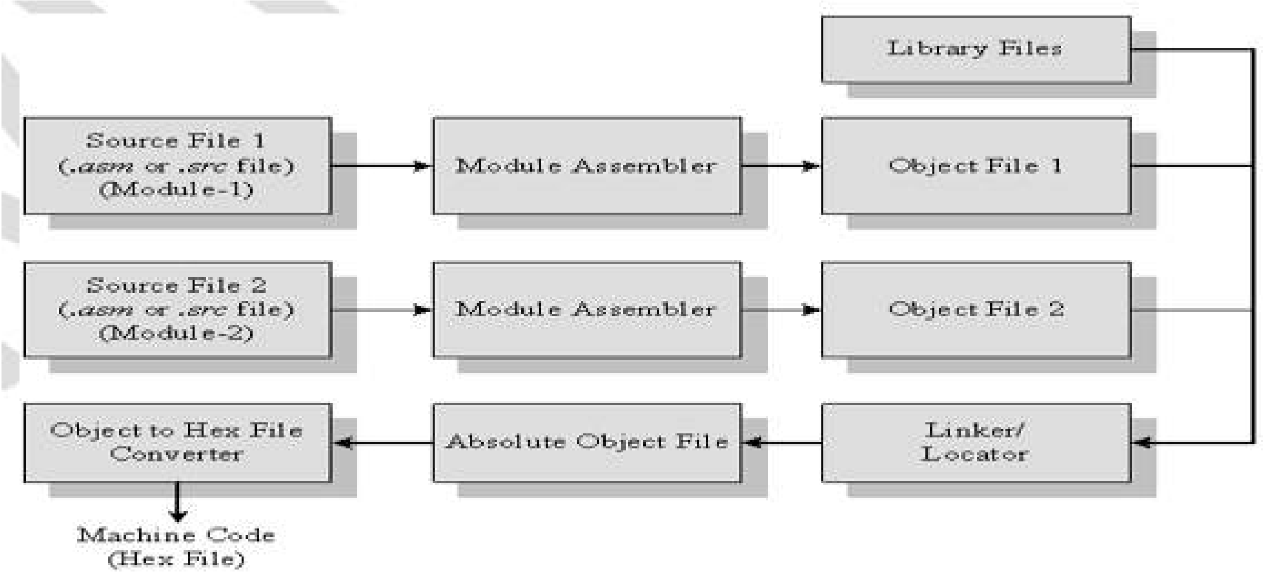


Figure 5: Assembly Language to machine language conversion process

- Each source file can be assembled separately to examine the syntax errors and incorrect assembly instructions

- Assembling of each source file generates a corresponding object file. The object file does not contain the absolute address of where the generated code needs to be placed (a re-locatable code) on the program memory

- The software program called linker/locater is responsible for assigning absolute address to object files during the linking process

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

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

Advantages:

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

- ➢ The developer is well aware of the target processor architecture and memory organization, so optimized code can be written for performing operations.
- ➢ This leads to less utilization of code memory and efficient utilization of data memory.

✶ 2.High Performance:

- ➢ Optimized code not only improves the code memory usage but also improves the total system performance.
- ➢ Through effective assembly coding, optimum performance can be achieved for target processor.

✶ 3.Low level Hardware Access:

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

✶ 4.Code Reverse Engineering:

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

Drawbacks:

✶ 1.High Development time:

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

✶ 2.Developer dependency:

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

✶ 3.Non portable:

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

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

- The embedded firmware is written in any high level language like C, C++
- A software utility called 'cross-compiler' converts the high level language to target processor specific machine code

- ▪ The cross-compilation of each module generates a corresponding object file. The object file does not contain the absolute address of where the generated code needs to be placed (a re-locatable code) on the program memory
- ▪ The software program called linker/locater is responsible for assigning absolute address to object files during the linking process
- ▪ The Absolute object file created from the object files corresponding to different source code modules contain information about the address where each instruction needs to be placed in code memory
- ▪ A software utility called 'Object to Hex file converter' translates the absolute object file to corresponding hex file (binary file)

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

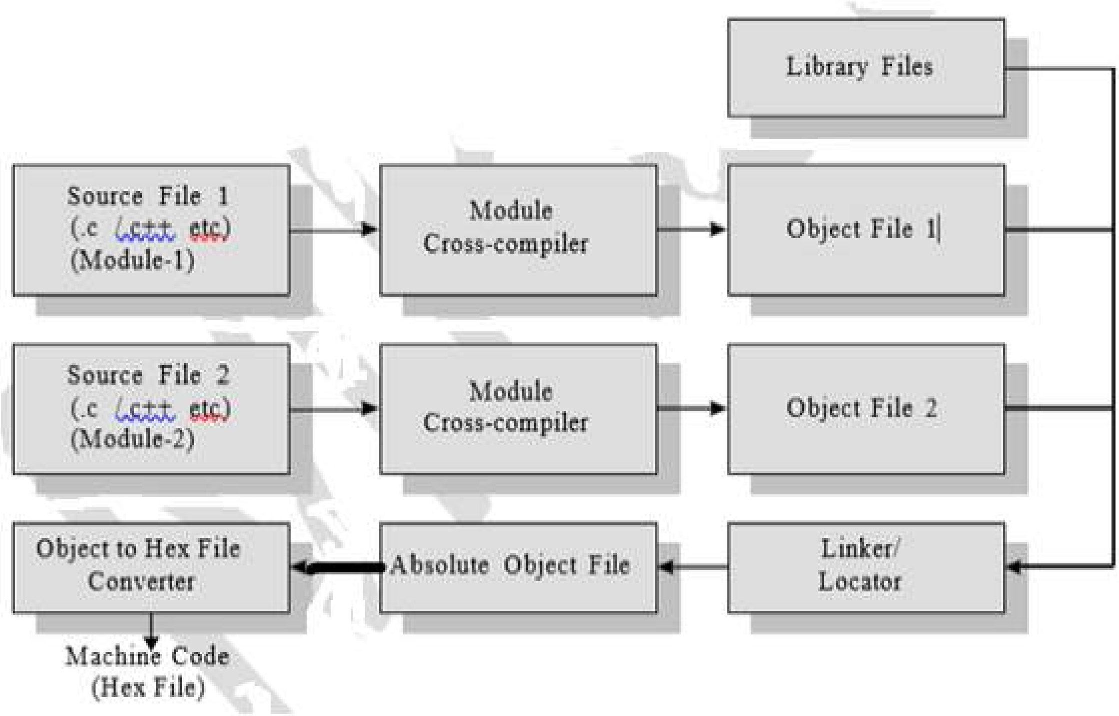


Figure 6: High level language to machine language conversion process

Advantages:

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

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

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

Drawbacks:

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

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

- Embedded firmware development may require the mixing of Assembly Language with high level language or vice versa.
- Interrupt handling, Source code is already available in high level language\Assembly Language etc are examples
- High Level language and low level language can be mixed in three different ways
  - ✓ Mixing Assembly Language with High level language like 'C'
  - ✓ Mixing High level language like 'C' with Assembly Language
  - ✓ In line Assembly
- The passing of parameters and return values between the high level and low level language is cross-compiler specific

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

- Assembly routines are mixed with 'C' in situations where the entire program is written in 'C' and the cross compiler in use do not have built in support for implementing certain features like ISR.
- If the programmer wants to take advantage of the speed and optimized code offered by the machine code generated by hand written assembly rather than cross compiler

generated machine code.

- For accessing certain low level hardware, the timing specifications may be very critical and cross compiler generated machine code may not be able to offer the required time specifications accurately.
- Writing the hardware/peripheral access routine in processor/controller specific assembly language and invoking it from ‘C’ is the most advised method.
- Mixing ‘C’ and assembly is little complicated.
- The programmer must be aware of how to pass parameters from the ‘C’ routine to assembly and values returned from assembly routine to ‘C’ and how Assembly routine is invoked from the ‘C’ code.

- Passing parameter to the assembly routine and returning values from the assembly routine to the caller ‘C’ function and the method of invoking the assembly routine from ‘C’ code is cross compiler dependent.
- There is no universal written rule for purpose.
- We can get this information from documentation of the cross compiler.

Different cross compilers implement these features in different ways depending on GPRs and memory supported by target processor/controller

2. Mixing High level language like ‘C’ with Assembly Language (‘C’ with Assembly Language)

- The source code is already available in assembly language and routine written in a high level language needs to be included to the existing code.
- The entire source code is planned in Assembly code for various reasons like optimized code, optimal performance, efficient code memory utilization and proven expertise in handling the assembly.
- The functions written in ‘C’ use parameter passing to the function and returns values to the calling functions.
- The programmer must be aware of how parameters are passed to the function and how values returned from the function and how function is invoked from the assembly language environment.

- Passing parameter to the function and returning values from the function using CPU registers, stack memory and fixed memory.
- Its implementation is cross compiler dependent and varies across compilers.

3. In line Assembly:

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

  Mov A,#13H

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

Text Books:

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

EMBEDDED SYSTEM DESIGN

UNIT-V

RTOS Based Embedded System Design

B. DEEPIKA RATHOD
Department of ECE

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 Kernel:

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

Kernel Space and User Space:

- The program code corresponding to the kernel applications/services are kept in a contiguous area (OS dependent) of primary (working) memory and is protected from the un-authorized access by user programs/applications
- The memory space at which the kernel code is located is known as '*Kernel Space'*

- All user applications are loaded to a specific area of primary memory and this memory area is referred as '*User Space*'
- The partitioning of memory into kernel and user space is purely Operating System dependent
- An operating system with virtual memory support, loads the user applications into its corresponding virtual memory space with demand paging technique
- Most of the operating systems keep the kernel application code in main memory and it is not swapped out into the secondary memory

Monolithic Kernel:

- All kernel services run in the kernel space
- All kernel modules run within the same memory space under a single kernel thread
- The tight internal integration of kernel modules in monolithic kernel architecture allows the effective utilization of the low-level features of the underlying system
- The major drawback of monolithic kernel is that any error or failure in any one of the kernel modules leads tothe crashing of the entire kernel application
- LINUX, SOLARIS, MS-DOS kernels are examples of monolithic kernel

Microkernel

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

...

- QNX, Minix 3 kernels are examples for microkernel.

Benefits of Microkernel:

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

Types of Operating Systems:

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

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

1. General Purpose Operating System (GPOS):

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

2. Real Time Purpose Operating System (RTOS):

- Operating Systems, which are deployed in embedded systems demanding real-time response
- Deterministic in execution behavior. Consumes only known amount of time for kernel applications
- Implements scheduling policies for executing the highest priority task/application always
- Implements policies and rules concerning time-critical allocation of a system's resources
- Windows CE, QNX, VxWorks , MicroC/OS-II etc are examples of Real Time Operating Systems (RTOS)

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

a) Task/Process management

b) Task/Process scheduling

c) Task/Process synchronization

d) Error/Exception handling

e) Memory Management

f) Interrupt handling

g) Time management

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

  - ❖ *Task ID:* Task Identification Number
  - ❖ *Task State:* The current state of the task. (E.g. State= 'Ready' for a task which is ready to execute)
  - ❖ *Task Type*: Task type. Indicates what is the type for this task. The task can be a hard real time or soft real time or background task.
  - ❖ *Task Priority*: Task priority (E.g. Task priority =1 for task with priority = 1)
  - ❖ *Task Context Pointer*: Context pointer. Pointer for context saving
  - ❖ *Task Memory Pointers*: Pointers to the code memory, data memory and stack memory for the task
  - ❖ *Task System Resource Pointers*: Pointers to system resources (semaphores, mutex etc) used by the task
  - ❖ *Task Pointers:* Pointers to other TCBs (TCBs for preceding, next and waiting tasks)
  - ❖ *Other Parameters* Other relevant task parameters

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

- **Task/Process Scheduling:** Deals with sharing the CPU among various tasks/processes. A kernel application called '*Scheduler*' handles the task scheduling. Scheduler is nothing but an algorithm implementation, which performs the efficient and optimal scheduling of tasks to provide a deterministic behavior.

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

□ **Error/Exception handling:** Deals with registering and handling the errors occurred/exceptions raised during the execution of tasks. Insufficient memory, timeouts, deadlocks, deadline missing, bus error, divide by zero, unknown instruction execution etc, are examples of errors/exceptions. Errors/Exceptions can happen at the kernel level services or at task level. Deadlock is an example for kernel level exception, whereas timeout is an example for a task level exception. The OS kernel gives the information about the error in the form of a system call (API).

□ **Memory Management:**

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

❖ The memory allocation time increases depending on the size of the block of memory needs to be allocated and the state of the allocated memory block (initialized memory block consumes more allocation time than 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 theGPOS.

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

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

❖ RTOS kernels assume that the whole design is proven correct and protection is unnecessary. Some commercial RTOS kernels allow memory protection as optional and the kernel enters a *fail-safe* mode when an illegal memory access occurs

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

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

- In the '*block*' based memory allocation, a block of fixed memory is always allocated for tasks on need basis and it is taken as a unit. Hence, there will not be any memory fragmentation issues.

- The memory allocation can be implemented as constant functions and thereby it consumes fixed amount of time for memory allocation. This leaves the deterministic behavior of the RTOS kernel untouched.

Interrupt Handling:

- Interrupts inform the processor that an external device or an associated task requires immediate attention of the CPU.

- Interrupts can be either *Synchronous* or *Asynchronous*.

- Interrupts which occurs in sync with the currently executing task is known as *Synchronous* interrupts. Usually the software interrupts fall under the Synchronous Interrupt category. Divide by zero, memory segmentation error etc are examples of Synchronous interrupts.

- For synchronous interrupts, the interrupt handler runs in the same context of the interrupting task.

- Asynchronous interrupts are interrupts, which occurs at any point of execution of any task, and are not in sync with the currently executing task.

- The interrupts generated by external devices (by asserting the Interrupt line of the processor/controller to which the interrupt line of the device is connected) connected to the processor/controller, timer overflow interrupts, serial data reception/ transmission interrupts etc are examples for asynchronous interrupts.
- For asynchronous interrupts, the interrupt handler is usually written as separate task (Depends on OS Kernel implementation) and it runs in a different context. Hence, a context switch happens while handling the asynchronous interrupts.

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

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

Time Management:

- Interrupts inform the processor that an external device or an associated task requires immediate attention of the CPU.

- Accurate time management is essential for providing precise time reference for all applications

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

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

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

- The time parameters for tasks are expressed as the multiples of the ‘Timer tick’

- The System time is updated based on the ‘Timer tick’

- If the System time register is 32 bits wide and the ‘Timer tick’ interval is 1microsecond, the System time register will reset in

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

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

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

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

- Save the current context (Context of the currently executing task)

- Increment the System time register by one. Generate timing error and reset the System time register if the timer tick count is greater than the maximum range available for System time register

- Update the timers implemented in kernel (Increment or decrement the timer registers for each timer depending on the count direction setting for each register. Increment registers with count direction setting = ‘*count up*’ and decrement registers with count direction setting = ‘*count down*’)
- Activate the periodic tasks, which are in the idle state
- Invoke the scheduler and schedule the tasks again based on the scheduling algorithm
- Delete all the terminated tasks and their associated data structures (TCBs)
- Load the context for the first task in the ready queue. Due to the re- scheduling, the ready task might be changed to a new one from the task, which was pre- empted by the ‘Timer Interrupt’ task

- **Hard Real-time System:**
  - ❖ A Real Time Operating Systems which strictly adheres to the timing constraints for a task
  - ❖ A Hard Real Time system must meet the deadlines for a task without any slippage
  - ❖ Missing any deadline may produce catastrophic results for Hard Real Time Systems, including permanent data 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
  - ❖ As a rule of thumb, Hard Real Time Systems does not implement the virtual memory model for handling the memory. This eliminates the delay in swapping in and out the code corresponding to the task to and from the primary memory
  - ❖ The presence of *Human in the loop* (*HITL*) for tasks introduces un- expected delays in the task execution. Most of the Hard Real Time Systems are automatic and does not contain a ‘human in the loop’

- **Soft Real-time System:**
  - ❖ Real Time Operating Systems that does not guarantee meeting deadlines, but, offer

the best effort to meet the deadline

- ❖ Missing deadlines for tasks are acceptable if the frequency of deadline missing is within the compliance limit of the Quality of Service(QoS)
- ❖ A Soft Real Time system emphasizes on the principle '*A late answer is an acceptable answer, but it could have done bit faster'*
- ❖ Soft Real Time systems most often have a '*human in the loop (HITL)*'
- ❖ Automatic Teller Machine (ATM) is a typical example of Soft Real Time System. If the ATM takes a few seconds more than the ideal operation time, nothing fatal happens.
- ❖ An audio video play back system is another example of Soft Real Time system. No potential damage arises if a sample comes late by fraction of a second, for play back.

□ Tasks, Processes & Threads:

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

□ The structure of a Processes

- ▪ The concept of '*Process*' leads to concurrent execution (pseudo parallelism) of tasks and thereby the efficient utilization of the CPU and other system resources
- ▪ Concurrent execution is achieved through the sharing of CPU among the processes.
- ▪ A process mimics a processor in properties and holds a set of registers, process status, a Program Counter (PC) to point to the next executable instruction of the process, a stack for holding the local variables associated with the process and the code corresponding to the process

- A process, which inherits all the properties of the CPU, can be considered as a virtual processor, awaiting its turn to have its properties switched into the physical processor

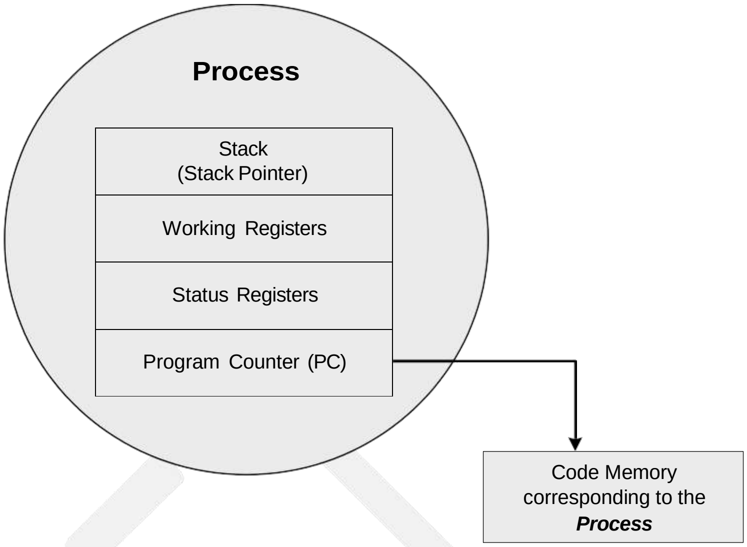


Figure: 4 Structure of a Process

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

Memory organization of Processes:

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

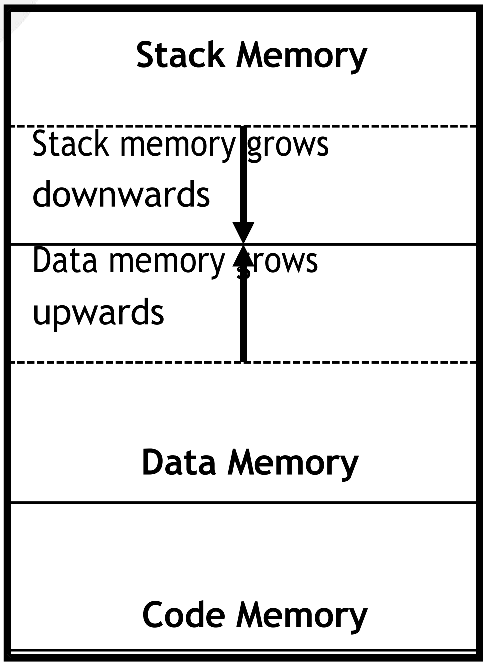


Fig: 5 Memory organization of a Process

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

□ **Process States & State Transition**

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

□ **Process States & State Transition**:

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

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

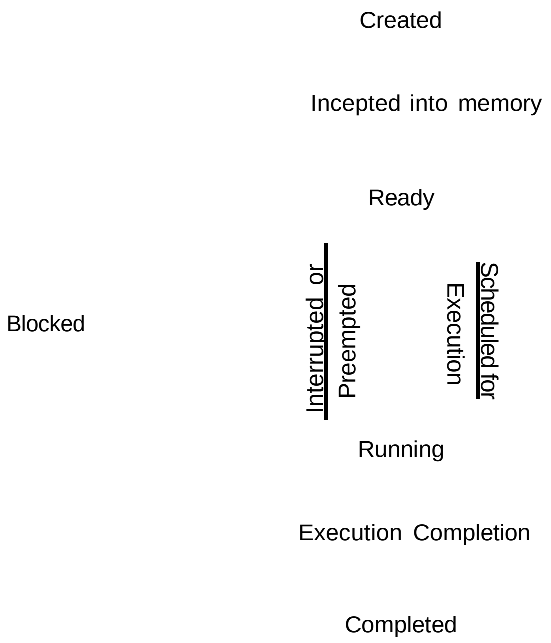


Figure 6.Process states and State transition

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

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

Threads

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

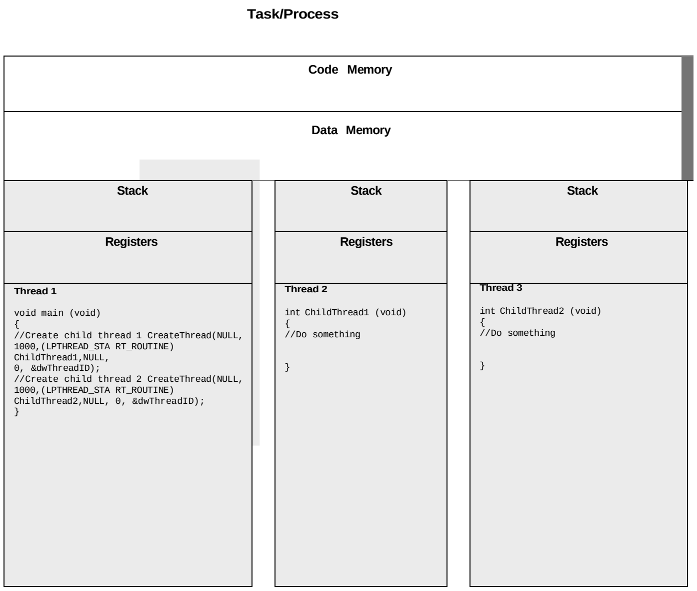
- Different threads, which are part of a process, share the same address space; meaning they share the data memory, code memory and heap memory area
- Threads maintain their own thread status (CPU register values), Program Counter (PC) and stack

Figure 7 Memory organization of process and its associated Threads

The Concept of multithreading

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

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



```
void main (void)
{
//Create child thread 1 CreateThread(NULL,
1000,(LPTHREAD_STA RT_ROUTINE)
ChildThread1,NULL,
0, &dwThreadID);
//Create child thread 2 CreateThread(NULL,
1000,(LPTHREAD_STA RT_ROUTINE)
ChildThread2,NULL, 0, &dwThreadID);
}
```

```
int ChildThread1 (void)
{
//Do something


}
```

```
int ChildThread2 (void)
{
//Do something


}
```

Figure 8 Process with multi-threads

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

□ Thread V/s Process

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

Advantages of Threads:

1. **Better memory utilization:** Multiple threads of the same process share the address space for data memory. This also reduces the complexity of inter thread communication since variables can be shared across the threads.

2. **Efficient CPU utilization:** The CPU is engaged all time.

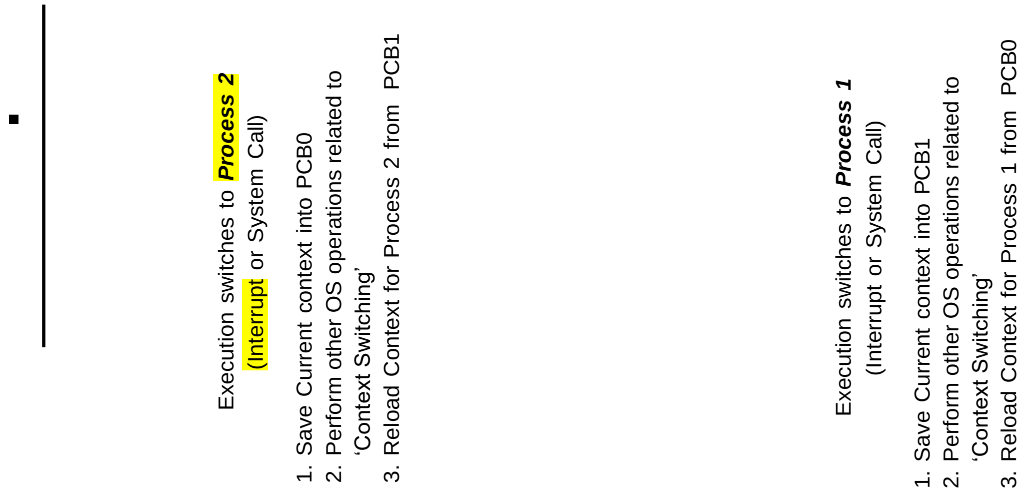
3. **Speeds up the execution of the process:** The process is split into different threads, when one thread enters a wait state, the CPU can be utilized by other threads of the process that do not require the event, which the other thread is waiting, for processing.

Multiprocessing & Multitasking

- The ability to execute multiple processes simultaneously is referred as *multiprocessing*

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

Multitasking – Context Switching:


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

Types of Multitasking:

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

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

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

- **Non-preemptive Multitasking:** The process/task, which is currently given the CPU time, is allowed to execute until it terminates (enters the 'Completed' state) or enters the

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

Task Scheduling:

- In a multitasking system, there should be some mechanism in place to share the CPU among the different tasks and to decide which process/task is to be executed at a given point of time
- Determining which task/process is to be executed at a given point of time is known as task/process scheduling
- Task scheduling forms the basis of multitasking
- Scheduling policies forms the guidelines for determining which task is to be executed when
- The scheduling policies are implemented in an algorithm and it is run by the kernel as a service
- The kernel service/application, which implements the scheduling algorithm, is known as ‘*Scheduler*’
- The task scheduling policy can be *pre-emptive*, *non-preemptive* or *co- operative*
- Depending on the scheduling policy the process scheduling decision may take place when a process switches its state to
    - ‘*Ready*’ state from ‘*Running*’ state
    - ‘*Blocked/Wait*’ state from ‘*Running*’ state
    - ‘*Ready*’ state from ‘*Blocked/Wait*’ state
    - ‘*Completed*’ state

Task Scheduling - Scheduler Selection:

The selection of a scheduling criteria/algorithm should consider

- **CPU Utilization:** The scheduling algorithm should always make the CPU utilization high. CPU utilization is a direct measure of how much percentage of the CPU is being utilized.
- **Throughput:** This gives an indication of the number of processes executed per unit of

time. The throughput for a good scheduler should always be higher.

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

- **Waiting Time:** It is the amount of time spent by a process in the '*Ready*' queue waiting to get the CPU time for execution. The waiting time should be minimal for a good scheduling algorithm.
- **Response Time:** It is the time elapsed between the submission of a process and the first response. For a good scheduling algorithm, the response time should be as least as possible.

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

Task Scheduling - Queues

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

- Job Queue: Job queue contains all the processes in the system
- Ready Queue: Contains all the processes, which are ready for execution and waiting for CPU to get their turn for execution. The Ready queue is empty when there is no process ready for running.
- Device Queue: Contains the set of processes, which are waiting for an I/O device

Task Scheduling – Task transition through various Queues

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

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

Drawbacks:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

= 15.66 milliseconds

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

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

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

Drawbacks:

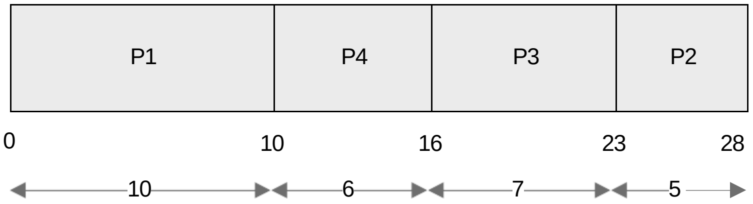
- Favors monopoly of process. A process, which does not contain any I/O operation, continues its execution until it finishes its task

- In general, LCFS favors CPU bound processes and I/O bound processes may have to wait until the completion of CPU bound process, if the currently executing process is a CPU bound process. This leads to poor device utilization.

- The average waiting time is not minimal for LCFS scheduling algorithm

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

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



The waiting time for all the processes are given as Waiting Time

for P1 = 0 ms (P1 starts executing first)

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

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

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

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

= 11 milliseconds

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

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

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

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

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

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

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

= 18 milliseconds

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

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

Drawbacks:

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

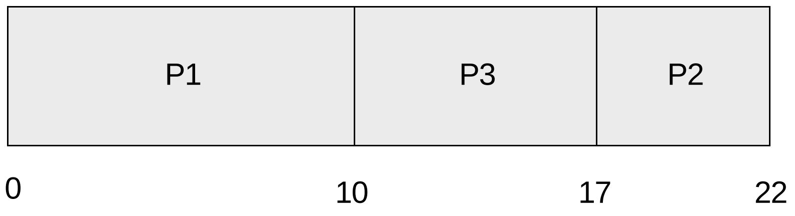
scheduling since new processes with different estimated execution time keep entering the ‘*Ready*’ queue at any point of time.

Non-preemptive scheduling – Priority based Scheduling

- A priority, which is unique or same is associated with each task
- The priority of a task is expressed in different ways, like a priority number, the time required to complete the execution etc.
- In number based priority assignment the priority is a number ranging from 0 to the maximum priority supported by the OS. The maximum level of priority is OS dependent.
- Windows CE supports 256 levels of priority (0 to 255 priority numbers, with 0 being the highest priority)
- The priority is assigned to the task on creating it. It can also be changed dynamically (If the Operating System supports this feature)
- The non-preemptive priority based scheduler sorts the ‘Ready’ queue based on the priority and picks the process with the highest level of priority for execution

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

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



10 7 5

The waiting time for all the processes are given as

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

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

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

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

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

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

= 9 milliseconds

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

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

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

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

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

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

= 16.33 milliseconds

Drawbacks:

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

Preemptive scheduling:

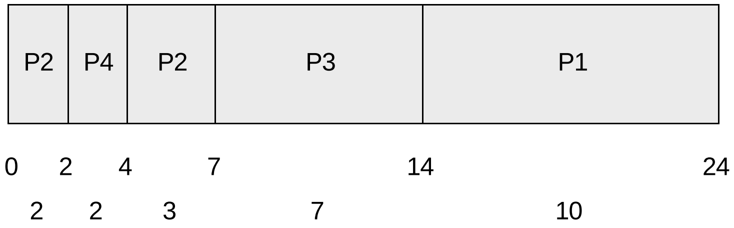
- Employed in systems, which implements preemptive multitasking model
- Every task in the '*Ready*' queue gets a chance to execute. When and how often each process gets a chance to execute (gets the CPU time) is dependent on the type of preemptive scheduling algorithm used for scheduling the processes
- The scheduler can preempt (stop temporarily) the currently executing task/process and select another task from the '*Ready*' queue for execution
- When to pre-empt a task and which task is to be picked up from the '*Ready*' queue for execution after preempting the current task is purely dependent on the scheduling algorithm
- A task which is preempted by the scheduler is moved to the '*Ready*' queue. The act of moving a '*Running*' process/task into the '*Ready*' queue by the scheduler, without the processes requesting for it is known as '*Preemption*'
- Time-based preemption and priority-based preemption are the two important approaches adopted in preemptive scheduling

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

- The *non preemptive SJF* scheduling algorithm sorts the 'Ready' queue only after the current process completes execution or enters wait state, whereas the *preemptive SJF* scheduling algorithm sorts the 'Ready' queue when a new process enters the 'Ready' queue and checks whether the execution time of the new process is shorter than the remaining of the total estimated execution time of the currently executing process
- If the execution time of the new process is less, the currently executing process is preempted and the new process is scheduled for execution
- Always compares the execution completion time (ie the remaining execution time for the new process) of a new process entered the 'Ready' queue with the remaining time for completion of the currently executing process and schedules the process with shortest remaining time for execution.

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

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



The waiting time for all the processes are given as

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

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

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

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

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

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

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

= 5.75 milliseconds

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

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

Turn Around Time (TAT) for P3 = 14 ms (Time spent in Ready Queue + Execution Time)
Turn Around Time (TAT) for P1 = 24 ms (Time spent in Ready Queue + Execution Time)
Average Turn Around Time = (Turn Around Time for all the processes) / No. of Processes

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

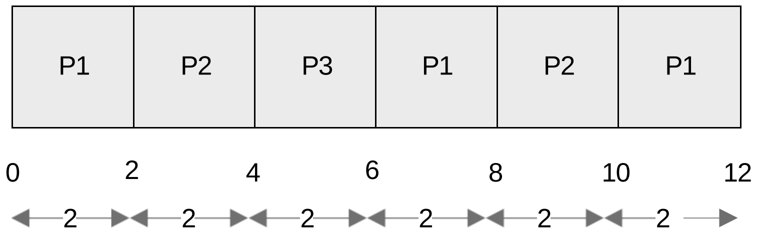
= (7+2+14+24)/4 = 47/4

= 11.75 milliseconds

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

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

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



The waiting time for all the processes are given as

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

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

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

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

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

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

= 5.33 milliseconds

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

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

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

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

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

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

= 9.33 milliseconds.

Preemptive scheduling – Priority based Scheduling

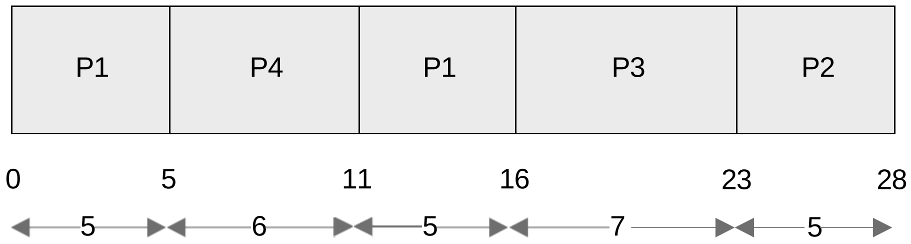
- Same as that of the ***non-preemptive priority*** based scheduling except for the switching of execution between tasks

- In ***preemptive priority*** based scheduling, any high priority process entering the 'Ready' queue is immediately scheduled for execution whereas in the ***non- preemptive*** scheduling any high priority process entering the 'Ready' queue is scheduled only after the currently executing process completes its execution or only when it voluntarily releases the CPU

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

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

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



The waiting time for all the processes are given as

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

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

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

P2 = 23 ms (P2 starts executing after completing P1, P4 and P3) Average waiting time =

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

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

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

= 11.25 milliseconds

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

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

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

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

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

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

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

= 18.25 milliseconds

How to choose RTOS:

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

These factors can be either

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

1. Functional Requirements:

1. Processor support:

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

It is essential to ensure the processor support by the RTOS

2. Memory Requirements:

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

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

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

3. Real-Time Capabilities:

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

3. Kernel and Interrupt Latency:

- The kernel of the OS may disable interrupts while executing certain services and it may lead to interrupt latency.

- For an embedded system whose response requirements are high, this latency should be minimal.

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

6. Modularization Support:

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

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

7. Support for Networking and Communication:

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

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

8. Development Language Support:

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

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

2. Non-Functional Requirements:

1. Custom Developed or Off the Shelf:

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

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

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

2. Cost:

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

3. Development and Debugging tools Availability:

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

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

4. Ease of Use:

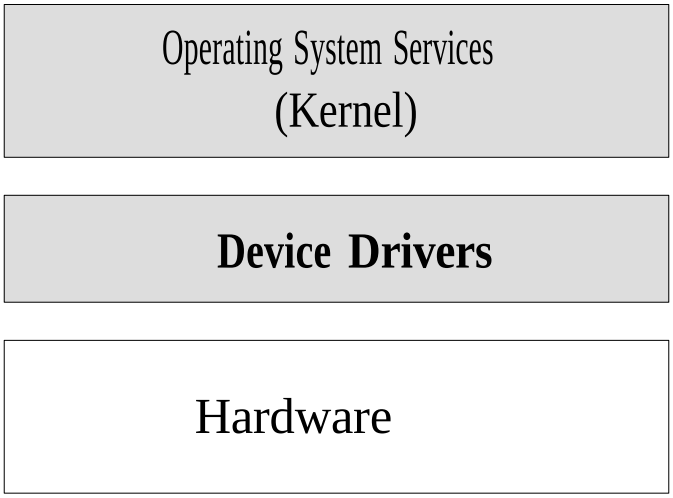
How easy it is to use a commercial RTOS is another important feature that needs to be considered in the RTOS selection.

5. After Sales:

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

Device Drivers:

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



- The architecture of the OS kernel will not allow direct device access from the user application
- All the device related access should flow through the OS kernel and the OS kernel routes it to the concerned hardware peripheral
- OS Provides interfaces in the form of Application Programming Interfaces (APIs) for accessing the hardware
- The device driver abstracts the hardware from user applications
- Device drivers are responsible for initiating and managing the communication with the hardware peripherals
- Drivers which comes as part of the Operating system image is known as ‘built-in drivers’ or ‘onboard’ drivers. Eg. NAND FLASH driver
- Drivers which needs to be installed on the fly for communicating with add- on devices are known as ‘Installable drivers’
- For installable drivers, the driver is loaded on a need basis when the device is present and it is unloaded when the device is removed/detached
- The ‘Device Manager service of the OS kernel is responsible for loading and unloading the driver, managing the driver etc.
- The underlying implementation of device driver is OS kernel dependent
- The driver communicates with the kernel is dependent on the OS structure and implementation.
- Device drivers can run on either user space or kernel space
- Device drivers which run in user space are known as *user mode drivers* and the drivers which run in kernel space are known as *kernel mode drivers*
- User mode drivers are safer than kernel mode drivers
- If an error or exception occurs in a user mode driver, it won’t affect the services of the kernel
- If an exception occurs in the kernel mode driver, it may lead to the kernel crash
- The way how a device driver is written and how the interrupts are handled in it are

Operating system and target hardware specific.

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

Reference Books:

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