LECTURE NOTES

ON

EMBEDDED SYSTEMS DESIGN

M. Tech I semester
(Autonomous-R18)

Prepared By

Ms. S.SUSHMA
Assistant Professor

ELECTRONICS AND COMMUNICATION ENGINEERING

INSTITUTE OF AERONAUTICAL ENGINEERING
(Autonomous)
DUNDIGAL, HYDERABAD - 500 043

Embedded Systems Design
LECTURE NOTES

SYLLABUS:

Unit-I
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.

UNIT-II
Typical Embedded System:
Core of the Embedded System: General Purpose and Domain Specific Processors, ASICs, PLDs, Commercial Off-The-Shelf Components (COTS), Memory: ROM, RAM, Memory according to the type of Interface, Memory Shadowing, Memory selection for Embedded Systems, Sensors and Actuators, Communication Interface: Onboard and External Communication Interfaces.

UNIT-III
Embedded Firmware:
Reset Circuit, Brown-out Protection Circuit, Oscillator Unit, Real Time Clock, Watchdog Timer, Embedded Firmware Design Approaches and Development Languages.

UNIT-IV
RTOS Based Embedded System Design:
Operating System Basics, Types of Operating Systems, Tasks, Process and Threads, Multiprocessing and Multitasking, Task Scheduling.

UNIT- V
Task Communication:
Shared Memory, Message Passing, Remote Procedure Call and Sockets, Task Synchronization: Task Communication Synchronization Issues, Task Synchronization Techniques, Device Drivers, How to Choose an RTOS.

TEXT BOOKS:
1. Introduction to Embedded Systems - Shibu K.V, Mc Graw Hill.

REFERENCE BOOKS:
Embedded Systems - Raj Kamal, TMH.
Embedded System Design - Frank Vahid, Tony Givargis, John Wiley.
Embedded Systems – Lyla, Pearson, 2013
An Embedded Software Primer - David E. Simon, Pearson Education.

UNIT -I

Introduction to Embedded systems

INTRODUCTION:

- An embedded system is an electronic system, which includes a single chip microcomputers (Microcontrollers) like the ARM or Cortex or Stellaris LM3S1968.
- It is configured to perform a specific dedicated application.
- An embedded system is some combination of computer hardwareand software, either fixed in capability or programmable, that is designed for a specific function or for specific functions within a larger system.
- Here the microcomputer is embedded or hidden inside the system. Every **embeddedmicrocomputer** system accepts inputs, performs computations, and generates outputs and runs in
  ―real time.‖

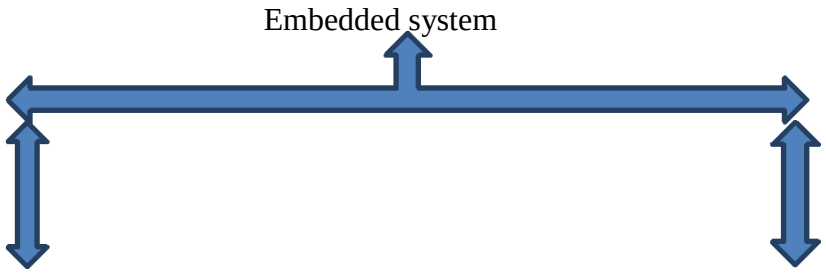
Ex: Cell phone, Digital camera, Microwave Oven, MP3 player, Portable digital assistant & automobileantilock brake system Industrial machines, agricultural and process industry devices, automobiles, medical equipment, household appliances, airplanes, vending machines and toys as well as mobile devices are all possible locations for an embedded system.etc.

Characteristics of an Embedded System: The important characteristics of an embedded system are

- Speed (bytes/sec) : Should be high speed
- Power (watts) : Low power dissipation
- Size and weight : As far as possible small in size and low weight
- Accuracy (% error) : Must be very accurate
- Adaptability: High adaptability and accessibility.
- Reliability: Must be reliable over a long period of time.

So, an embedded system must perform the operations at a high speed so that it can be readily used for real time applications and its power consumption must be very low and the size of the system should be as for as possible small and the readings must be accurate with minimum error. The system must be easily adaptable for different situations.

CATEGORIES OF EMBEDDED SYSTEMS: Embedded systems can be classified into the following 4categories based on their functional and performance requirements.



- Stand alone embedded systems — small scale embedded system
- Real time embedded system — medium scale embedded s/m
    a) Hard real time E.S
    b) Soft Real time E.S
- Networked embedded system — large scale embedded system
- Mobile embedded system .

Stand alone Embedded systems:

A stand-alone embedded system works by itself. It is a self-contained device which does not require any host system like a computer. It takes either digital or analog inputs from its input ports, calibrates, converts, and processes the data, and outputs the resulting data to its attached output device, which either displays data, or controls and drives the attached devices.

EX: Temperature measurement systems, Video game consoles, MP3 players, digital cameras, andmicrowave ovens are the examples for this category.

Real-time embedded systems:

An embedded system which gives the required output in a specified time or which strictly follows the time deadlines for completion of a task is known as a Real time system. i.e. a Real Time system , in addition to functional correctness, also satisfies the time constraints .

There are two types of Real time systems. (i) Soft real time system and (ii) Hard real time system.

- **Soft Real-Time system:** A Real time system in which, the violation of time constraints will cause only the degraded quality, but the system can continue to operate is known as a Soft real time system. In soft real-time systems, the design focus is to offer a guaranteed bandwidth to each real-time task and to distribute the resources to the tasks.

  Ex: A Microwave Oven, washing machine, TV remote etc.

- **Hard Real-Time system:** A Real time system in which, the violation of time constraints will causecritical failure and loss of life or property damage or catastrophe is known as a Hard Real t ime system.

These systems usually interact directly with physical hardware instead of through a human being .The hardware and software of hard real-time systems must allow a worst case execution (WCET) analysis that guarantees the execution be completed within a strict deadline. The chip selection and RTOS selection become important factors for hard real-time system design.

Ex: Deadline in a missile control embedded system , Delayed alarm during a Gas leakage , car airbag control system , A delayed response in pacemakers ,Failure in RADAR functioning etc.

Networked embedded systems:

The networked embedded systems are related to a network with network interfaces to access the resources. The connected network can be a Local Area Network (LAN) or a Wide Area Network (WAN), or the Internet. The connection can be either wired or wireless.

The networked embedded system is the fastest growing area in embedded systems applications. The embedded web server is such a system where all embedded devices are connected to a web server and can be accessed and controlled by any web browser.

Ex: A home security system is an example of a LAN networked embedded system where all sensors (e.g. motion detectors, light sensors, or smoke sensors) are wired and running on the TCP/IP protocol.

Mobile Embedded systems:

The portable embedded devices like mobile and cellular phones, digital cameras, MP3 players, PDA (Personal Digital Assistants) are the example for mobile embedded systems. The basic limitation of these devices is the limitation of memory and other resources.

Based on the performance of the Microcontroller they are also classified into (i) Small scaled embedded system (ii) Medium scaled embedded system and (iii) Large scaled embedded system.

Small scaled embedded system:

An embedded system supported by a single 8–16 bit Microcontroller with on-chip RAM and ROM designed to perform simple tasks is a Small scale embedded system.

Medium scaled embedded system:

An embedded system supported by 16–32 bit Microcontroller /Microprocessor with external RAM

and ROM that can perform more complex operations is a Medium scale embedded system.

Large scaled embedded system:

An embedded system supported by 32-64 bit multiple chips which can perform distributed jobs is considered as a Large scale embedded system.

Application Areas of Embedded Systems:

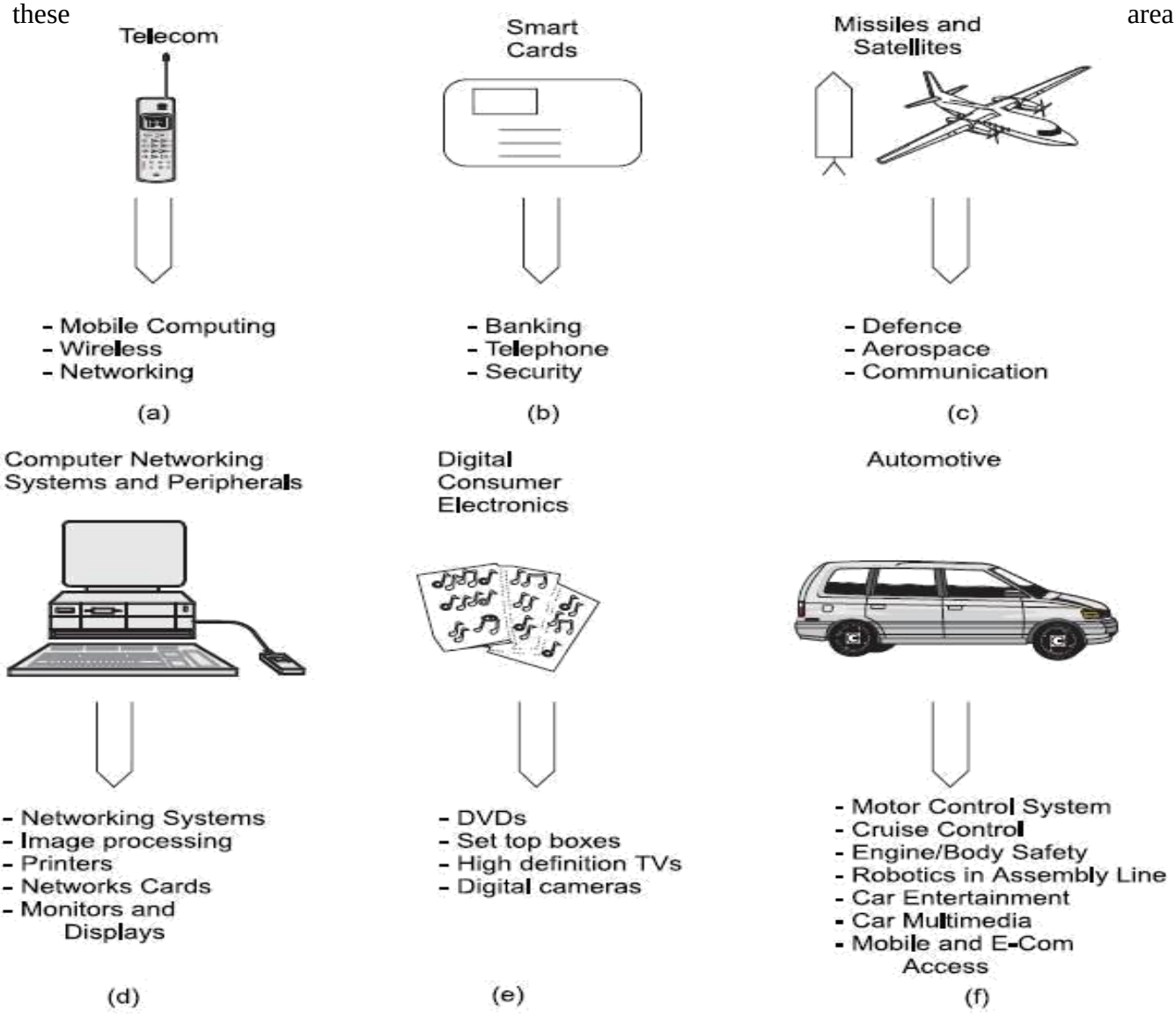
The embedded systems have a huge variety of application domains which varies from very low cost to very high cost and from daily life consumer electronics to industry automation equipments, from entertainment devices to academic equipments, and from medical instruments to aerospace and weapon control systems. So, the embedded systems span all aspects of our modern life. The following table gives the various applications of embedded systems.

Embedded System	**Application**
Home Appliances	Dishwasher, washing machine, microwave, Top-set box, security system , HVAC system, DVD, answering machine, garden sprinkler systems etc..
Office Automation	Fax, copy machine, smart phone system, modern, scanner, printers.
Security	Face recognition, finger recognition, eye recognition, building security system , airport security system, alarm system.
Academia	Smart board, smart room, OCR, calculator, smart cord.
Instrumentation	Signal generator, signal processor, power supplier,Process instrumentation,
Telecommunication	uter, hub, cellular phone, IP phone, web camera
Automobile	Fuel injection controller, anti-locking brake system, air-bag system, GPS, cruise control.
Entertainment	P3, video game, Mind Storm, smart toy.
Aerospace	Navigation system, automatic landing system, flight attitude controller, space explorer, space robotics.
Industrial automation	Assembly line, data collection system, monitoring systems on pressure, voltage, current, temperature, hazard detecting system, industrial robot.
Personal	PDA, iPhone, palmtop, data organizer.
Medical	CT scanner, ECG , EEG , EMG ,MRI, Glucose monitor, blood pressure monitor, medical diagnostic device.
Banking & Finance	ATM, smart vendor machine, cash register ,Share market

Miscellaneous:	Elevators, tread mill, smart card, security door etc.

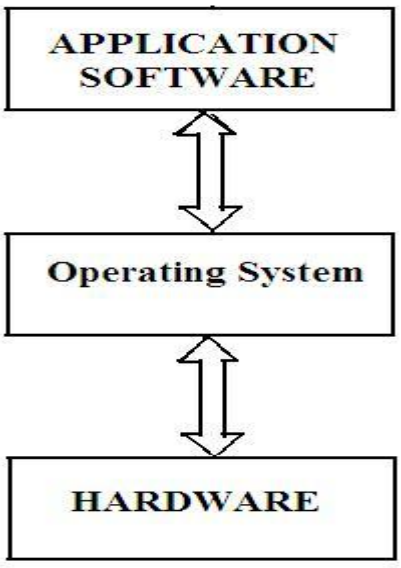
Exemplary applications of each type of embedded system

Embedded systems have very diversified applications. A few select application areas of embedded systems are Telecom, Smart Cards, Missiles and Satellites, Computer Networking, Digital Consumer Electronics, and Automotive. Figure 1.9 shows the applications of embedded systems in these area



Overview of embedded systems architecture:

- Every embedded system consists of customer-built hardware components supported by a Central Processing Unit (CPU), which is the heart of a microprocessor (µP) or microcontroller (µC).
- A microcontroller is an integrated chip which comes with built-in memory, I/O ports, timers, and other components.
- Most embedded systems are built on microcontrollers, which run faster than a custom-built system with a microprocessor, because all components are integrated within a single chip.
- Operating system plays an important role in most of the embedded systems. But all the embedded systems do not use the operating system.
- The systems with high end applications only use operating system. To use the operating system the embedded system should have large memory capability.
- So, this is not possible in low end applications like remote systems, digital cameras, MP3 players, robot toys etc.
- The architecture of an embedded system with OS can be denoted by layered structure as shown below.
- The OS will provide an interface between the hardware and application software.



In the case of embedded systems with OS, once the application software is loaded into memory it will run the application without any host system.

Coming to the hardware details of the embedded system, it consists of the following important blocks.

- CPU(Central Processing Unit)

- RAM and ROM
- I/O Devices
- Communication Interfaces
- Sensors etc. (Application specific circuitry)

This hardware architecture can be shown by the following block diagram.

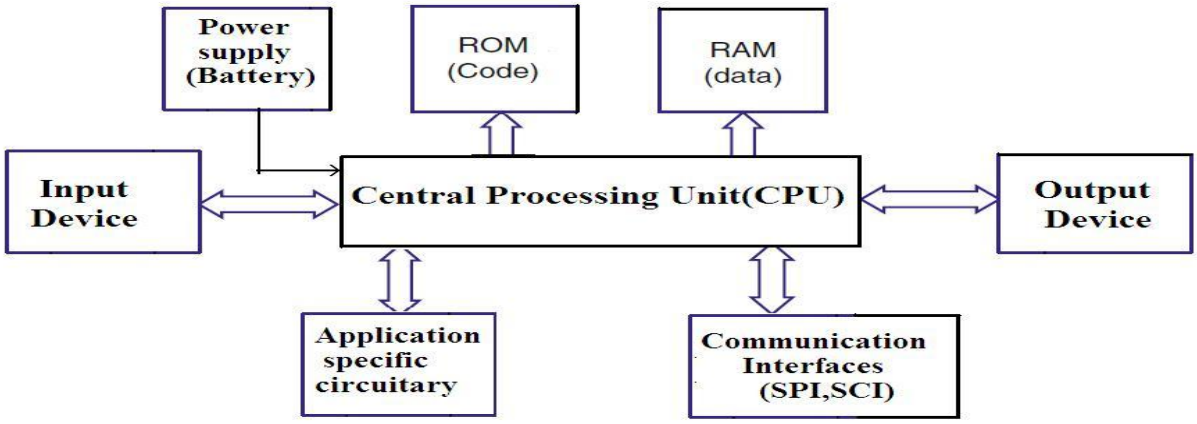
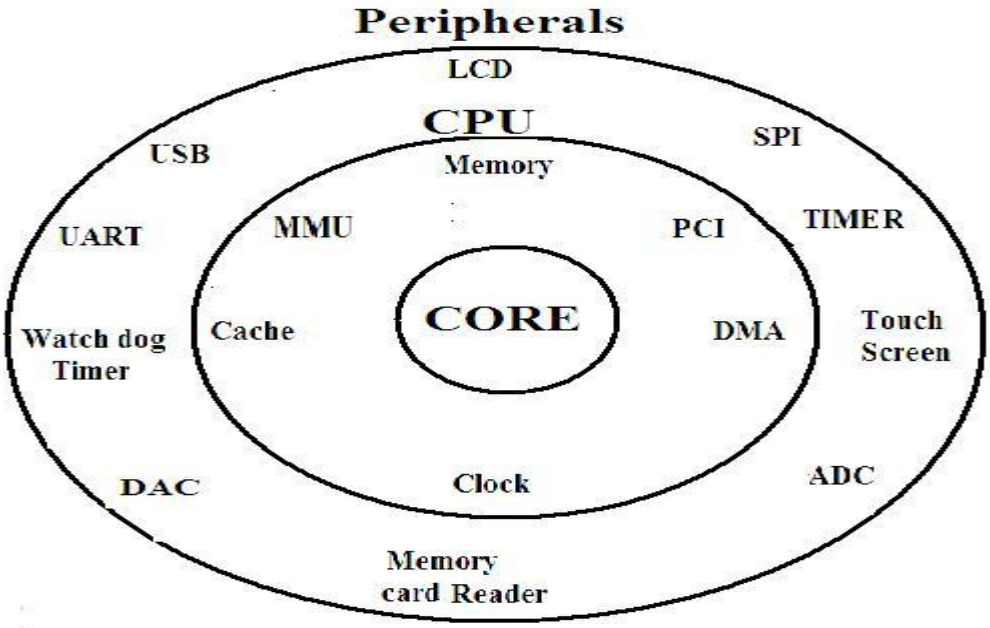


Fig: hardware architecture of embedded system

Central Processing Unit:

- A CPU is composed of an Arithmetic Logic Unit (ALU), a Control Unit (CU), and many internal registers that are connected by buses.
- The ALU performs all the mathematical operations (Add, Sub, Mul, Div), logical operations (AND, OR), and shifting operations within CPU.
- The timing and sequencing of all CPU operations are controlled by the CU, which is actually built of many selection circuits including latches and decoders. The CU is responsible for directing the flow of instruction and data within the CPU and continuously running program instructions step by step.
- The CPU works in a cycle of fetching an instruction, decoding it, and executing it, known as the fetch-decode-execute cycle.
- For embedded system design, many factors impact the CPU selection, e.g., the maximum size (number of bits) in a single operand for ALU (8, 16, 32, 64 bits), and CPU clock frequency for timing tick control, i.e. the number of ticks (clock cycles) per second in measures of MHz
- CPU contains the core and the other components which support the core to execute programs. Peripherals are the components which communicate with other systems or physical world (Like ports, ADC,DAC, Watch dog Timers etc.). The core is separated from other components by the system bus.

- The CPU in the embedded system may be a general purpose processor like a microcontroller or a special purpose processor like a DSP (Digital signal processor). But any CPU consists of of an Arithmetic Logic Unit (ALU), a Control Unit (CU), and many internal registers that are connected by buses. The ALU performs all the mathematical operations (Add, Sub, Mul, Div), logical operations (AND, OR), and shifting operations within CPU.

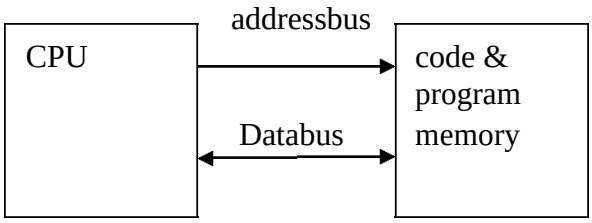


- There are many internal registers in the CPU.
- The accumulator (A) is a special data register that stores the result of ALU operations. It can also be used as an operand. The Program Counter (PC) stores the memory location of the next instruction to be executed. The Instruction Register (IR) stores the current machine instruction to be decoded and executed.
- The Data Buffer Registers store the data received from the memory or the data to be sent to memory. The Data Buffer Registers are connected to the data bus.
- The Address Register stores the memory location of the data to be accessed (get or set). The Address Register is connected to the address bus.
- In an embedded system, the CPU may never stop and run forever .The CPU works in a cycle of fetching an instruction, decoding it, and executing it, known as the fetch-decode-execute cycle. The cycle begins when an instruction is fetched from a memory location pointed to by the PC to the IR via the data bus.

When data and code lie in different memory blocks, then the architecture is referred as **Harvardarchitecture**. 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.



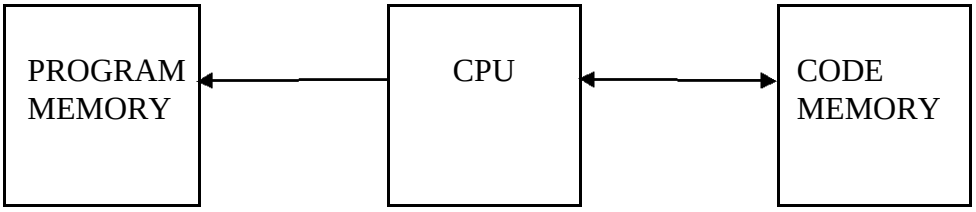
Von-Neumann architecture supports simple hardware. It allows the use of a single, sequential memory. Today's processing speeds vastly outpace memory access times, and we employ a very fast but small amount of memory (cache) local to the processor.

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.

Von-Neumann Architecture vs Harvard Architecture:



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

Von-Neumann Architecture	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.

CISC and RISC:

- CISC is a Complex Instruction Set Computer. It is a computer that can address a large number of instructions.
- In the early 1980s, computer designers recommended that computers should use fewer instructions with simple constructs so that they can be executed much faster within the CPU without having to use memory. Such computers are classified as Reduced Instruction Set Computer or RISC.

CISC vs RISC:

The following points differentiate a CISC from a RISC −

CISC	RISC
Larger set of instructions. Easy to program	Smaller set of Instructions. Difficult to program.
Simpler design of compiler, considering larger set of instructions.	Complex design of compiler.
Many addressing modes causing complex instruction formats.	Few addressing modes, fix instruction format.
Instruction length is variable.	Instruction length varies.
Higher clock cycles per second.	Low clock cycle per second.
Emphasis is on hardware.	Emphasis is on software.
Control unit implements large instruction set using micro-program unit.	Each instruction is to be executed by hardware.

Slower execution, as instructions are to be read from memory and decoded by the decoder unit.	Faster execution, as each instruction is to be executed by hardware.

Pipelining is not possible.	Pipelining of instructions is possible, considering single clock cycle.
Mainly used in normal pc‘s, workstations & servers.	Mainly used for real time applications.

Memory:

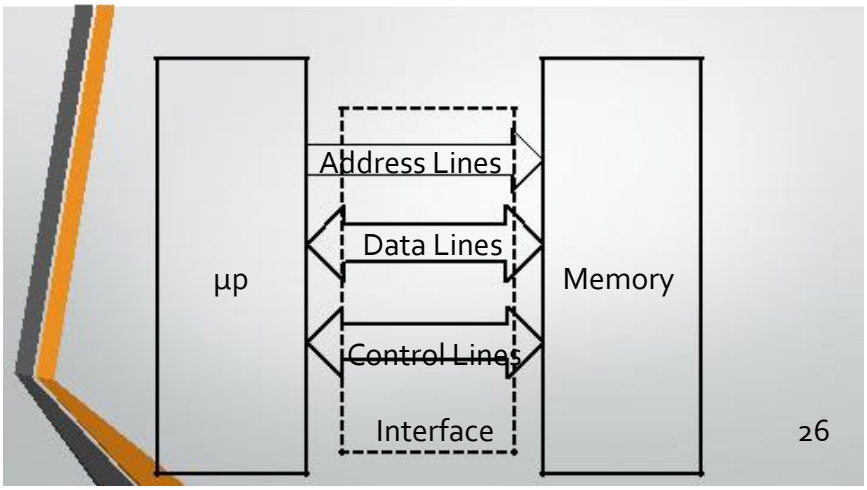
- Embedded system memory can be either on-chip or off-chip.
- On chip memory access is much fast than off-chip memory, but the size of on-chip memory is much smaller than the size of off-chip memory.
- Usually, it takes at least two I/O ports as external address lines plus a few control lines such as R/W and ALE control lines to enable the extended memory. Generally the data is stored in RAM and the program is stored in ROM.
- The ROM, EPROM, and Flash memory are all read-only type memories often used to store code in an embedded system.
- The embedded system code does not change after the code is loaded into memory.
- The ROM is programmed at the factory and cannot be changed over time.
- The newer microcontrollers come with EPROM or Flash instead of ROM.
- Most microcontroller development kits come with EPROM as well.
- EPROM and Flash memory are easier to rewrite than ROM. EPROM is an Erasable Programmable ROM in which the contents can be field programmed by a special burner and can be erased by a UV light bulb.
- The size of EPROM ranges up to 32kb in most embedded systems.
- Flash memory is an Electrically EPROM which can be programmed from software so that the developers don‘t need to physically remove the EPROM from the circuit to re-program it.
- It is much quicker and easier to re-write Flash than other types of EPROM.
- When the power is on, the first instruction in ROM is loaded into the PC and then the CPU fetches the instruction from the location in the ROM pointed to by the PC and stores it in the IR to start the continuous CPU fetch and execution cycle. The PC is advanced to the address of the next instruction depending on the length of the current instruction or the destination of the Jump instruction.

- The memory is divided into Data Memory and Code Memory.

- Most of data is stored in Random Access Memory (RAM) and code is stored in Read Only Memory (ROM).
- This is due to the RAM constraint of the embedded system and the memory organization.
- The RAM is readable and writable, faster access and more expensive volatile storage, which can be used to store either data or code.
- Once the power is turned off, all information stored in the RAM will be lost.
- The RAM chip can be SRAM (static) or DRAM (dynamic) depending on the manufacturer. SRAM is faster than DRAM, but is more expensive.

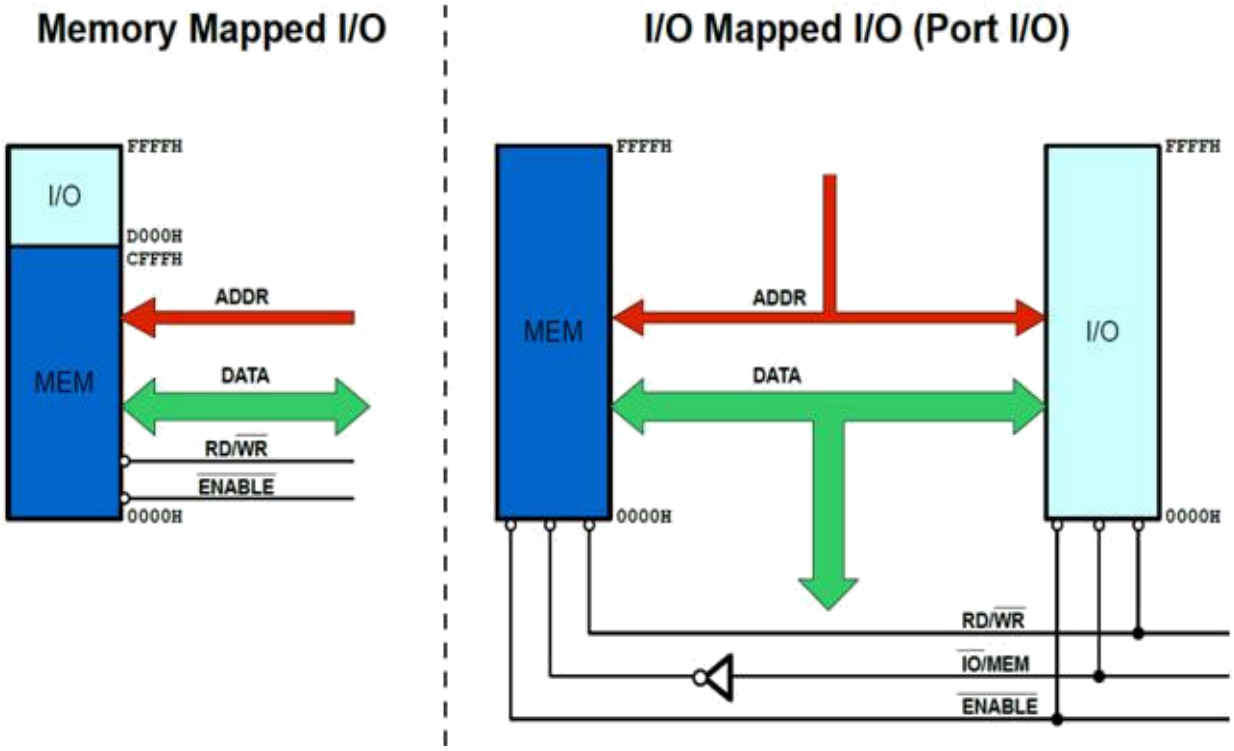
I/O Ports:

- The I/O ports are used to connect input and output devices. The common input devices for an embedded system include keypads, switches, buttons, knobs, and all kinds of sensors (light, temperature, pressure, etc).
- The output devices include Light Emitting Diodes (LED), Liquid Crystal Displays (LCD), printers, alarms, actuators, etc. Some devices support both input and output, such as communication interfaces including Network Interface Cards (NIC), modems, and mobile phones.



In parallel I/O have two types of interfacing

1. Memory mapped I/O
2. I/O mapped I/O



Memory Mapped IO

- IO is treated as memory.
- 16-bit addressing.
- More Decoder Hardware.
- Can address 2^{16}=64k locations.
- Less memory is available.

IO Mapped IO

- IO is treated IO.
- 8- bit addressing.
- Less Decoder Hardware.
- Can address 2^{8}=256 locations.
- Whole memory address space is available.

Memory Mapped IO	IO Mapped IO
• Memory Instructions are used. • Memory control signals are used. • Arithmetic and logic operations can be performed on data. • Data transfer b/w register and IO.	• Special Instructions are used like IN, OUT. • Special control signals are used. • Arithmetic and logic operations can not be performed on data. • Data transfer b/w accumulator and IO.

Communication Interfaces:

- To transfer the data or to interact with other devices, the embedded devices are provided the various communication interfaces like RS232, RS422, RS485 ,USB, SPI(Serial Peripheral Interface) ,SCI (Serial Communication Interface) ,Ethernet etc.

Application Specific Circuitry:

- The embedded system sometimes receives the input from a sensor or actuator. In such situations certain signal conditioning circuitry is needed. This hardware circuitry may contain ADC, Op-amps, DAC etc. Such circuitry will interact with the embedded system to give correct output.

ADC & DAC:

- Many embedded system application need to deal with non-digital external signals such as electronic voltage, music or voice, temperature, pressures, and many other signals in the analog form.
- The digital computer does not understand these data unless they are converted to digital formats. The ADC is responsible for converting analog values to binary digits.
- The DAC is responsible for outputting analog signals for automation controls such as DC motor or HVDC furnace control.

In addition to these peripherals, an embedded system may also have sensors, Display modules like LCD or Touch screen panels, Debug ports certain communication peripherals like I^2C, SPI, Ethernet, CAN, USB for high speed data transmission. Now a days various sensors are also becoming an important part in the design of real time embedded systems. Sensors like temperature sensors, light sensors, PIR sensors, gas sensors are widely used in application specific circuitry.

- According to computer architecture, a bus is defined as a system that transfers data between hardware components of a computer or between two separate computers.

- Initially, buses were made up using electrical wires, but now the term bus is used more broadly to identify any physical subsystem that provides equal functionality as the earlier electrical buses.

- Computer buses can be parallel or serial and can be connected as multidrop, daisy chain or by switched hubs.

- System bus is a single bus that helps all major components of a computer to communicate with each other.

- It is made up of an address bus, data bus and a control bus. The data bus carries the data to be stored, while address bus carries the location to where it should be stored.

Address Bus

- Address bus is a part of the computer system bus that is dedicated for specifying a physical address.

- When the computer processor needs to read or write from or to the memory, it uses the address bus to specify the physical address of the individual memory block it needs to access (the actual data is sent along the data bus).

- More correctly, when the processor wants to write some data to the memory, it will assert the write signal, set the write address on the address bus and put the data on to the data bus.

- Similarly, when the processor wants to read some data residing in the memory, it will assert the read signal and set the read address on the address bus.

- After receiving this signal, the memory controller will get the data from the specific memory block (after checking the address bus to get the read address) and then it will place the data of the memory block on to the data bus.

The size of the memory that can be addressed by the system determines the width of the data bus and vice versa. For example, if the width of the address bus is 32 bits, the system can address 232 memory blocks (that is equal to 4GB memory space, given that one block holds 1 byte of data).

Data Bus

- A data bus simply carries data. Internal buses carry information within the processor, while external buses carry data between the processor and the memory.

- Typically, the same data bus is used for both read/write operations. When it is a write operation, the processor will put the data (to be written) on to the data bus.

- When it is the read operation, the memory controller will get the data from the specific memory block and put it in to the data bus.

What is the difference between Address Bus and Data Bus?

- Data bus is bidirectional, while address bus is unidirectional. That means data travels in both directions but the addresses will travel in only one direction.
- The reason for this is that unlike the data, the address is always specified by the processor. The width of the data bus is determined by the size of the individual memory block, while the width of the address bus is determined by the size of the memory that should be addressed by the system.

Power supply:

- Most of the embedded systems now days work on battery operated supplies.
- Because low power dissipation is always required. Hence the systems are designed to work with batteries.

Clock:

- The clock is used to control the clocking requirement of the CPU for executing instructions and the configuration of timers. For ex: the 8051 clock cycle is $(1/12)10^{-6}$ second (1/12μs) because the clock frequency is 12MHz. A simple 8051 instruction takes 12 cycles (1ms) to complete. Of course, some multi-cycle instructions take more clock cycles.
- A timer is a real-time clock for real-time programming. Every timer comes with a counter which can be configured by programs to count the incoming pulses. When the counter overflows (resets to zero) it will fire a timeout interrupt that triggers predefined actions. Many time delays can be generated by timers. For example ,a timer counter configured to 24,000 will trigger the timeout signal in 24000x 1/12μs = 2ms.
- In addition to time delay generation, the timer is also widely used in the real-time embedded system to schedule multiple tasks in multitasking programming. The watchdog timer is a special timing device that resets the system after a preset time delay in case of system anomaly. The watchdog starts up automatically after the system power up.
- One need to reboot the PC now and then due to various faults caused by hardware or software. An embedded system cannot be rebooted manually, because it has been embedded into its system. That is why many microcontrollers come with an on-chip watchdog timer which can be configured just like the counter in the regular timer. After a system gets stuck (power supply voltage out of range or regular timer does not issue timeout after reaching zero count) the watchdog eventually will restart the system to bring the system back to a normal operational condition.

Application Specific software:

It sits above the O.S. The application software is developed according to the features of the development tools available in the OS.

These development tools provide the function calls to access the services of the OS. These function calls include, creating a task ,to read the data from the port and write the data to the memory etc.

The various function calls provided by an operating system are

i. To create ,suspend and delete tasks.

ii. To do task scheduling to providing real time environment.

iii. To create inter task communication and achieve the synchronization between tasks.

iv. To access the I/O devices.

v. To access the communication protocol stack .

The designer develops the application software based on these function calls.

Recent trends in Embedded systems : With the fast developments in semiconductor industry andVLSI technology ,one can find tremendous changes in the embedded system design in terms of processor speed , power , communication interfaces including network capabilities and software developments like operating systems and programming languages etc.

- **Processor speed and Power :**With the advancements in processor technology ,the embeddedsystems are now days designed with 16,32 bit processors which can work in real time environment. These processors are able to perform high speed signal processing activities which resulted in the development of high definition communication devices like 3G mobiles etc.Also the recent

  developments in VLSI technology has paved the way for low power battery operated devices which are very handy and have high longevity. Also , the present day embedded systems are provided with higher memory capabilities ,so that most of them are based on tiny operating systems like android etc.

- **Communication interfaces :**Most of the present day embedded systems are aimed at internet basedapplications. So,the communication interfaces like Ethernet, USB, wireless LAN etc.have become very common resources in almost all the embedded systems. The developments in memory technologies also helped in porting the TCP/IP protocol stack and the HTTP server software on to the embedded systems. Such embedded systems can provide a link between any two devices anywhere in the globe.

- **Operating systems :**With recent software developments ,there is a considerable growth in theavailability of operating systems for embedded systems. Mainly new operating systems are developed which can be used in real time applications. There are both commercial RTOSes like Vx

Works , QNX,WIN-CE and open source RTOSes like RTLINUX etc. The Android OS in mobiles has revolutionized the embedded industry.

- **Programming Languages :**There is also a remarkable development in the programming languages.Languages like C++, Java etc. are now widely used in embedded application programming. For
  example by having the Java virtual machine in a mobile phones ,one can download Java applets from a server and can be executed on your mobile.

In addition to these developments, now a days we also find new devices like ASICs and FPGAs in the embedded system market. These new hardware devices are popular as programmable devices and reconfigurable devices.

Msp430 introduction:

- The MSP430 was introduced in the late 1990s. It is a particularly straightforward 16-bit processor with a von Neumann architecture, designed for low-power applications.
- Both the address and data buses are 16 bits wide. The registers in the CPU are also all 16 bits wide and can be used interchangeably for either data or addresses.
- This makes the MSP430 simpler than an 8-bit processor with 16-bit addresses. Such a processor must use its general-purpose registers in pairs for addresses or provide separate, wider registers.
- In many ways, the MSP430 fits between traditional 8- and 16-bit processors.
- The 16-bit data bus and registers clearly define it as a 16-bit processor. On the other hand, it can address only 216 =64KB of memory.
- The MSP430 has 16 registers in its CPU, which enhances efficiency because they can be used for local variables, parameters passed to subroutines, and either addresses or data.
- This is a typical feature of a RISC, but unlike a ―pure‖ RISC, it can perform arithmetic directly on values in main memory.
- Microcontrollers typically spend much of their time on such operations. The MSP430 is the simplest microcontroller in Texas Instrumentations.

Embedded Systems Vs General Computing Systems:

The difference between an embedded system and a general purpose computer system is one of purpose, and to a much lesser extent, design. While a general purpose system can be used for many things, an embedded system is only meant for one purpose.

General Purpose Systems

A general purpose computer system is what you think of when someone says the word "computer." The defining feature of a general purpose computer is that it can be reconfigured for a new purpose. In the early days of digital computers, this involved actually rewiring the entire system. Today, most end users aren't even aware that this is happening, as the process has become completely transparent.

Embedded Systems

An embedded system is a little harder to pin down. It is dedicated to a single purpose, or a small set of purposes. Embedded systems can be found in nearly every single piece of modern electronics--in fact, they are the electronics. A modern television, a portable music player, a computer-controlled air conditioning system or virtually anything made in the last 10 years that isn't a general purpose system and requires electricity: that is what an embedded system is.

Characteristics and Quality Attributes of Embedded Systems:

The characteristics of embedded system are different from those of a general purpose computer and so are its Quality metrics. This chapter gives a brief introduction on the characteristics of an embedded system and the attributes that are associated with its quality.

Characteristics Of Embedded System:

Following are some of the characteristics of an embedded system that make it different from a general purpose computer:

Application and Domain specific:

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

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

Certain embedded systems are specific to a domain: ex. A hearing aid is an application that belongs to the domain of signal processing.

Reactive and Real time:

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

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

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

Operation in harsh environment:

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

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

Distributed:

Certain embedded systems are part of a larger system and thus form components of a distributed system. These components are independent of each other but have to work together for the larger system to function properly.

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

Small size and weight:

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

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

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

Power concerns:

It is desirable that the power utilization and heat dissipation of any embedded system be low.If more heat is dissipated then additional units like heat sinks or cooling fans need to be added to the circuit. If more power is required then a battery of higher power or more batteries need to be accommodated in the embedded system.

3. QUALITY ATTRIBUTES OF EMBEDDED SYSTEM:

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

1. Operational Quality Attributes.
    - Response
    - Throughput
    - Reliability
    - Maintainability
    - Scheduled or Periodic Maintenance
    - Maintenance to unexpected failure
    - Security and Safety
2. Non-Operational Quality Attributes.
    - Testability and Debug-ability
    - Evolvability
    - Portability
    - Time to prototype and market
    - Per unit and total cost

Operational Quality Attributes

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

Non-Operational Quality Attributes

These are attributes **not** related to operation or functioning of an embedded system. The way an embedded system operates affects its overall quality.These are the attributes that are associated with the embedded system before it can be put in operation.

1. **Operational Quality Attributes**

Response:

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

Throughput:

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

Reliability:

Reliability is a measure of how much percentage you rely upon the proper functioning of the system .Mean Time between failures and Mean Time To Repair are terms used in defining system reliability.Mean Time between failures can be defined as the average time the system is functioning before a failure occurs.Mean time to repair can be defined as the average time the system has spent in repairs.

Maintainability:

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

It can be classified into two types :-

Scheduled or Periodic Maintenance:

This is the maintenance that is required regularly after a periodic time interval.

Example :Periodic Cleaning of Air Conditioners,Refilling of printer cartridges.

Maintenance to unexpected failure

This involves the maintenance due to a sudden breakdown in the functioning of the system.

Example:Air conditioner not powering on,Printer not taking paper in spite of a full paper stack

Security:

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

Confidentiality deals with protection data from unauthorized disclosure.Integrity gives protection from unauthorized modification.Availability gives protection from unauthorized user.Certain Embedded systems have to make sure they conform to the security measures.

Ex. An Electronic Safety Deposit Locker can be used only with a pin number like a password.

Safety:

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

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

2. **Non Operational Attributes**

Testability and Debug-ability:

- It deals with how easily one can test his/her design, application and by which mean he/she can test it.

- In hardware testing the peripherals and total hardware function in designed manner
- Firmware testing is functioning in expected way
- Debug-ability is means of debugging the product as such for figuring out the probable sources that create unexpected behavior in the total system

Evolvability:

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

Portability:

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

Time to prototype and market

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

Per unit and total cost

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

UNIT II

TYPICAL EMBEDDED SYSTEM

Embedded systems overview

An embedded system is nearly any computing system other than a desktop computer. An embedded system is a dedicated system which performs the desired function upon power up, repeatedly.

Embedded systems are found in a variety of common electronic devices such as consumer electronics ex. Cell phones, pagers, digital cameras, VCD players, portable Video games, calculators, etc.,

Embedded systems are found in a variety of common electronic devices, such as: (a)consumer electronics -- cell phones, pagers, digital cameras, camcorders, videocassette recorders, portable video games, calculators, and personal digital assistants; (b) home appliances
-- microwave ovens, answering machines, thermostat, home security, washing machines, and lighting systems; (c) office automation -- fax machines, copiers, printers, and scanners; (d) business equipment cash registers, curbside check-in, alarm systems, card readers, product scanners, and automated teller machines; (e) automobiles --transmission control, cruise control, fuel injection, anti-lock brakes, and active suspension

Classifications of Embedded systems

1. Small Scale Embedded Systems: These systems are designed with a single 8- or 16-bit microcontroller; they have little hardware and software complexities and involve board- level design. They may even be battery operated. When developing embedded software for these, an editor, assembler and cross assembler, specific to the microcontroller or processor used, are the main programming tools. Usually, _C_ is used for developing these systems. _C_ program compilation is done into the assembly, and executable codes are then appropriately located in the system memory. The software has to fit within the memory available and keep in view the need to limit power dissipation when system is running continuously.

2. Medium Scale Embedded Systems: These systems are usually designed with a single or few 16- or 32-bit microcontrollers or DSPs or Reduced Instruction Set Computers (RISCs). These have both hardware and software complexities. For complex software design, there are the following programming tools: RTOS, Source code engineering tool, Simulator, Debugger and Integrated Development Environment (IDE). Software tools also provide the solutions to the hardware complexities. An assembler is of little use as a programming tool. These systems may also employ the readily available ASSPs and IPs (explained later) for the various functions—for example, for the bus interfacing, encrypting, deciphering, discrete cosine transformation and inverse transformation, TCP/IP protocol stacking and network connecting functions.
3. Sophisticated Embedded Systems: Sophisticated embedded systems have enormous hardware and software complexities and may need scalable processors or configurable processors and programmable logic arrays. They are used for cutting edge applications that need hardware and software co-design and integration in the final system; however, they are constrained by the processing speeds available in their hardware units. Certain software functions such as encryption and deciphering algorithms, discrete cosine transformation and inverse transformation algorithms, TCP/IP protocol stacking and network driver functions are implemented in the hardware to obtain additional speeds by saving time. Some of the functions of the hardware resources in the system are also implemented by the software. Development tools for these systems may not be readily available at a reasonable cost or may not be available at all. In some cases, a compiler or retarget able compiler might have to be developed for these.

The processing units of the embedded system

1. Processor in an Embedded System A processor is an important unit in the embedded system hardware. A microcontroller is an integrated chip that has the processor, memory and several other hardware units in it; these form the microcomputer part of the embedded system. An embedded processor is a processor with special features that allow it to be embedded into a system. A digital signal processor (DSP) is a processor meant for applications that process digital signals.

2. Commonly used microprocessors, microcontrollers and DSPs in the small-, medium-and large scale embedded systems
3. A recently introduced technology that additionally incorporates the application-specific system processors (ASSPs) in the embedded systems.
4. Multiple processors in a system.

Embedded systems are a combination of hardware and software as well as other components that we bring together inti products such as cell phones,music player,a network router,or an aircraft guidance system.they are a system within another system as we see in Figure 1.1

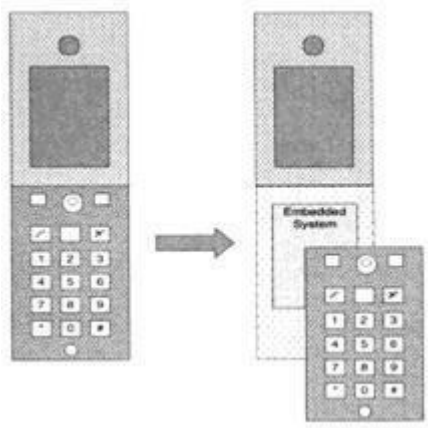


Figure 1.1: A simple embedded system

Building an embedded system

we embed 3 basic kinds of computing engines into our systems: microprocessor, microcomputer and microcontrollers. The microcomputer and other hardware are connected via A system bus is a single computer bus that connects the major components of a computer system. The technique was developed to reduce costs and improve modularity. It combines the functions of a data bus to carry information, an address bus to determine where it should be sent, and a control bus to determine its operation.

The system bus is further classified int address ,data and control bus.the microprocessor controls the whole system by executing a set of instructions call firmware that is stored in ROM.

An instruction set, or instruction set architecture (ISA), is the part of the computer architecture related to programming, including the native data types, instructions, registers, addressing modes, memory architecture, interrupt and exception handling, and external I/O. An ISA includes a specification of the set of opcodes (machine language), and the native commands

implemented by a particular processor. To run the application, when power is first turned ON, the microprocessor addresses a predefined location and fetches, decodes, and executes the instruction one after the other. The implementation of a microprocessor based embedded system combines the individual pieces into an integrated whole as shown in Figure 1.2, which represents the architecture for a typical embedded system and identifies the minimal set of necessary components.

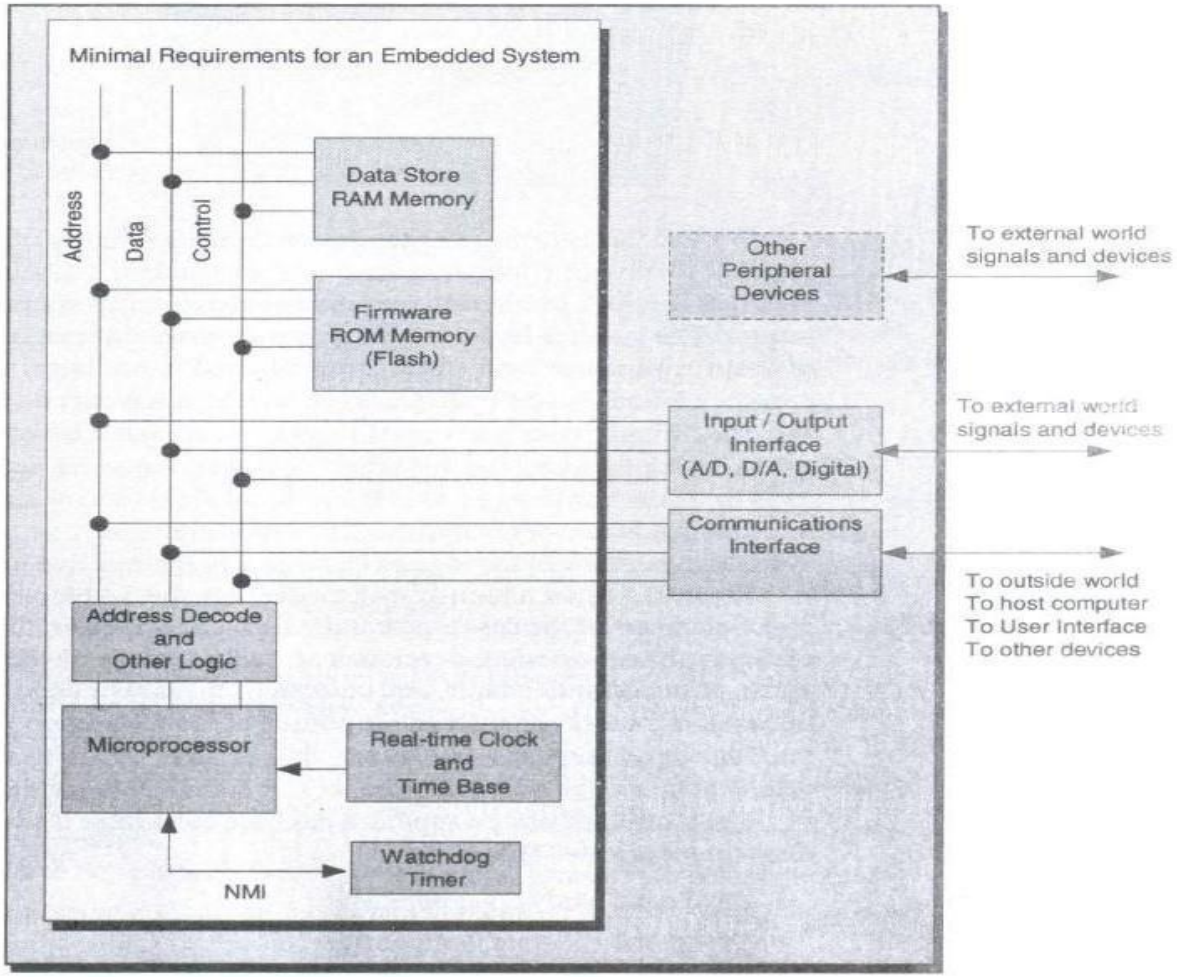


Figure 1.2 :A Microprocessor based Embedded system

Embedded design and development process

Figure1.3 shows a high level flow through the development process and identifies the major elements of the development life cycle.

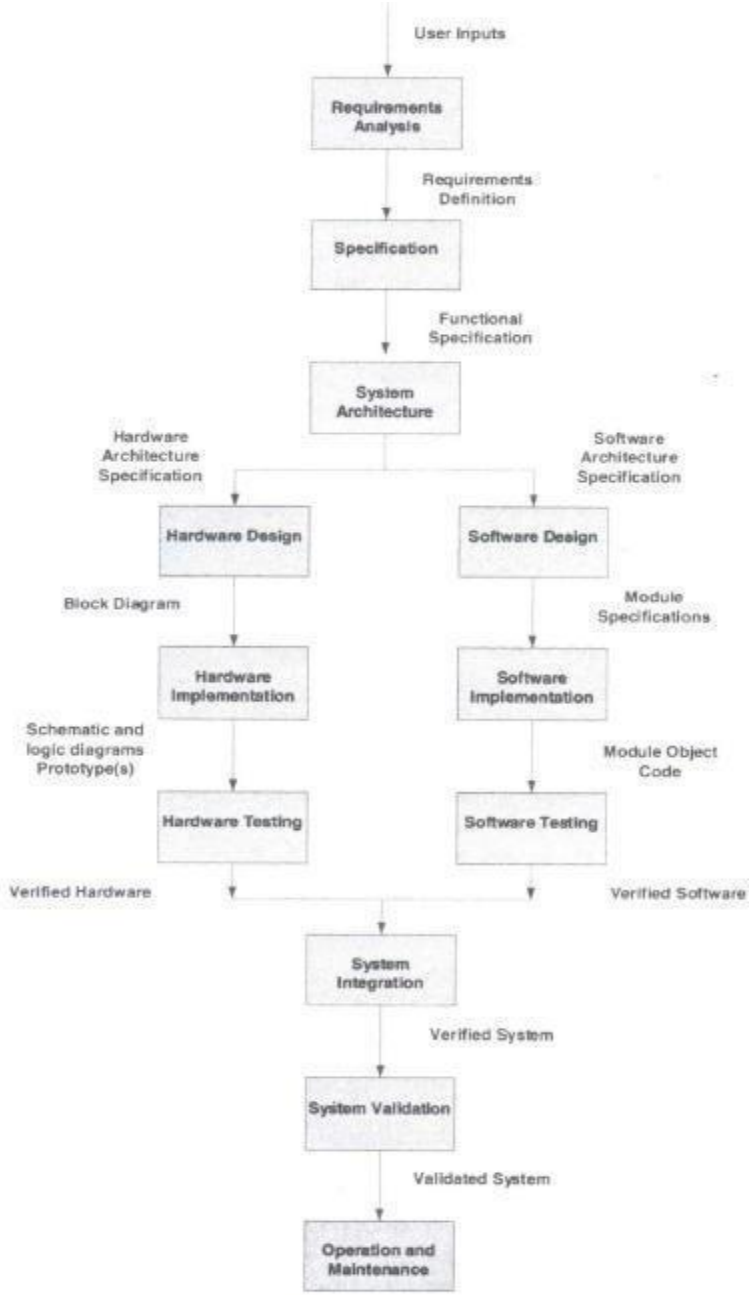


Figure 1.3 Embedded system life cycle

The traditional design approach has been traverse the two sides of the accompanying diagram separately, that is,

- Design the hardware components • Design the software components. • Bring the two together.
- Spend time testing and debugging the system.

The major areas of the design process are

- Ensuring a sound software and hardware specification.
- Formulating the architecture for the system to be designed. • Partitioning the h/w and s/w.
- Providing an iterative approach to the design of h/w and s/w.

The important steps in developing an embedded system are • Requirement definition.

- System specification. • Functional design
- Architectural design • Prototyping.

The major aspects in the development of embedded applications are

- Digital hardware and software architecture
- Formal design , development, and optimization process. • Safety and reliability.
- Digital hardware and software/firmware design.
- The interface to physical world analog and digital signals.
- Debug, troubleshooting and test of our design.

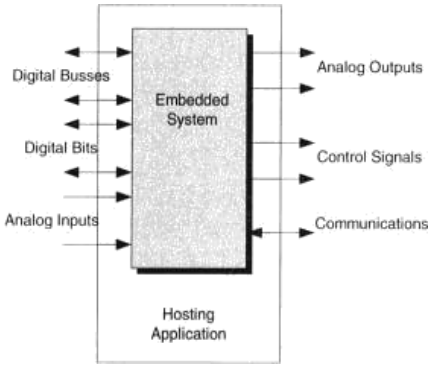


Figure 1.4: Interfacing to the outside world

THE HARDWARE SIDE

In today_s hi-tech and changing world, we can put together a working hierarchy of hard ware components. At the top, we find VLSI circuits comprising of significant pieces of functionality: microprocessor, microcontrollers, FPGA_s, CPLD, and ASIC.

Our study of hardware side of embedded systems begins with a high level view of the computing core of the system. we will expand and refine that view of hardware both inside and outside of the core. Figure 2.1 illustrates the sequence.

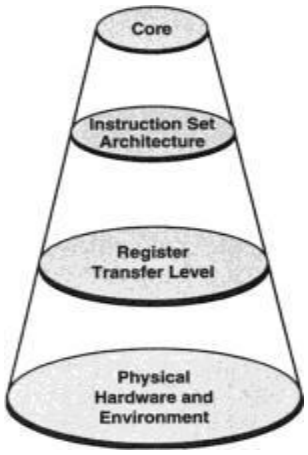


Figure 2.1 Exploring embedded systems

The core level

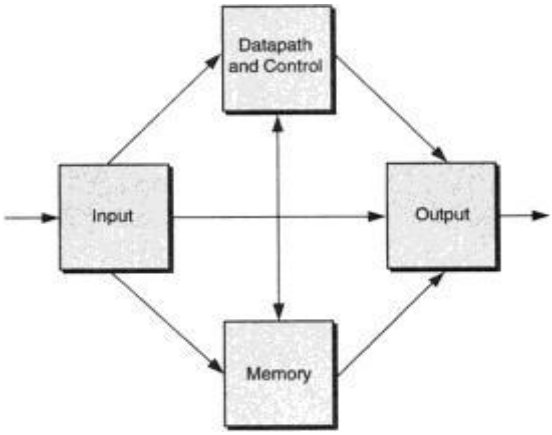


Figure 2.2 Four major blocks of an embedded hardware core

At the top, we begin with a model comprising four major functional blocks i.e., input, output, memory and data path and control depicting the embedded hardware core and high level signal flow as illustrated in figure 2.2.

The source of the transfer is the array of eight bit values; the destination is perhaps a display. in figure 2.3, we refine the high level functional diagram to illustrate a typical bus configuration comprising the address, data and control lines.

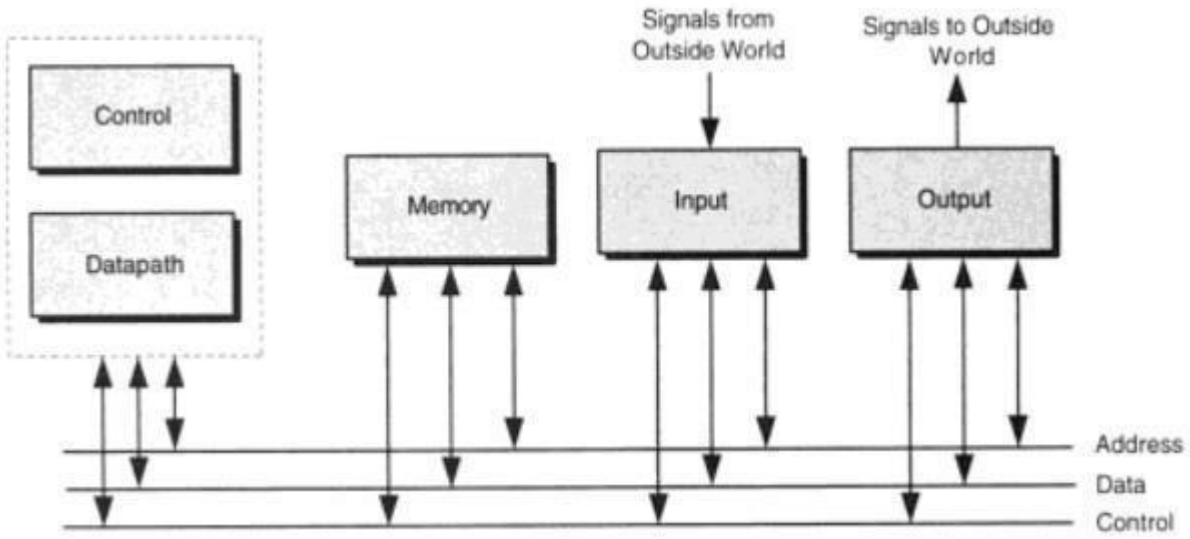


Figure 2.3 A typical Bus structure comprising address, data and control signals.

The Microprocessor

A microprocessor (sometimes abbreviated µP) is a programmable digital electronic component that incorporates the functions of a central processing unit (CPU) on a single semiconducting integrated circuit (IC). It is a multipurpose, programmable device that accepts digital data as input, processes it according to instructions stored in its memory, and provides results as output. It is an example of sequential digital logic, as it has internal memory. Microprocessors operate on numbers and symbols represented in the binary numeral system.

A microprocessor control program can be easily tailored to different needs of a product line, allowing upgrades in performance with minimal redesign of the product. Different features

can be implemented in different models of a product line at negligible production cost. Figure 2.4 shows a block diagram for a microprocessor based system.

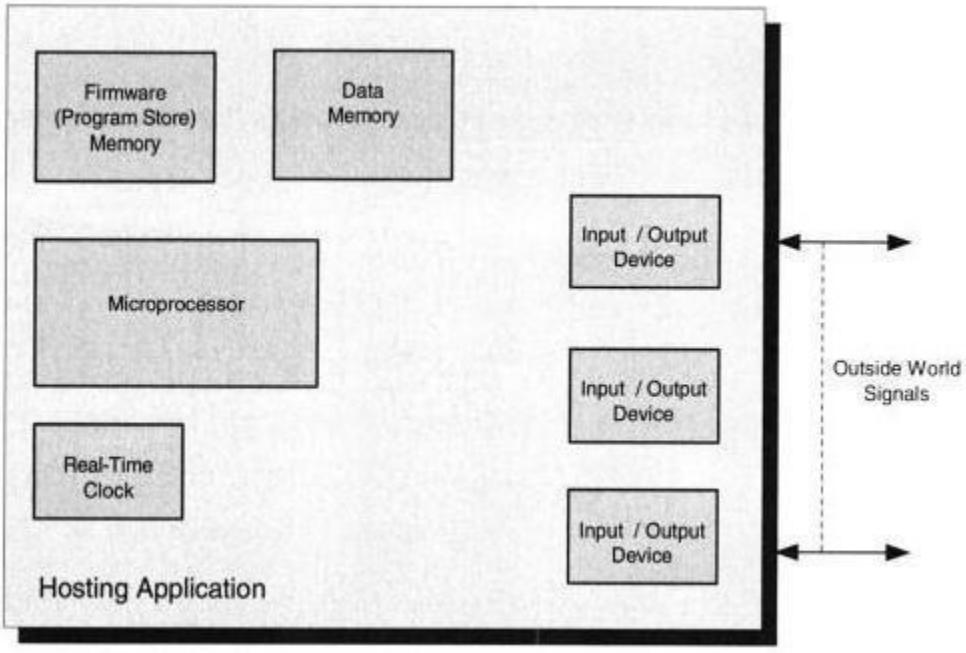


Figure 2.4 : A block diagram for a microprocessor based system

The microcomputer

The microcomputer is a complete computer system that uses a microprocessor as its computational core. Typically, a microcomputer will also utilizes numerous other large scale integrated circuits to provide necessary peripheral functionality. The complexity of microcomputers varies from simple units that are implemented on a single chip along with a small amount of on chip memory and elementary I/O system to the complex that will augment the microprocessor with a wide array of powerful peripheral support circuitry.

The microcontroller

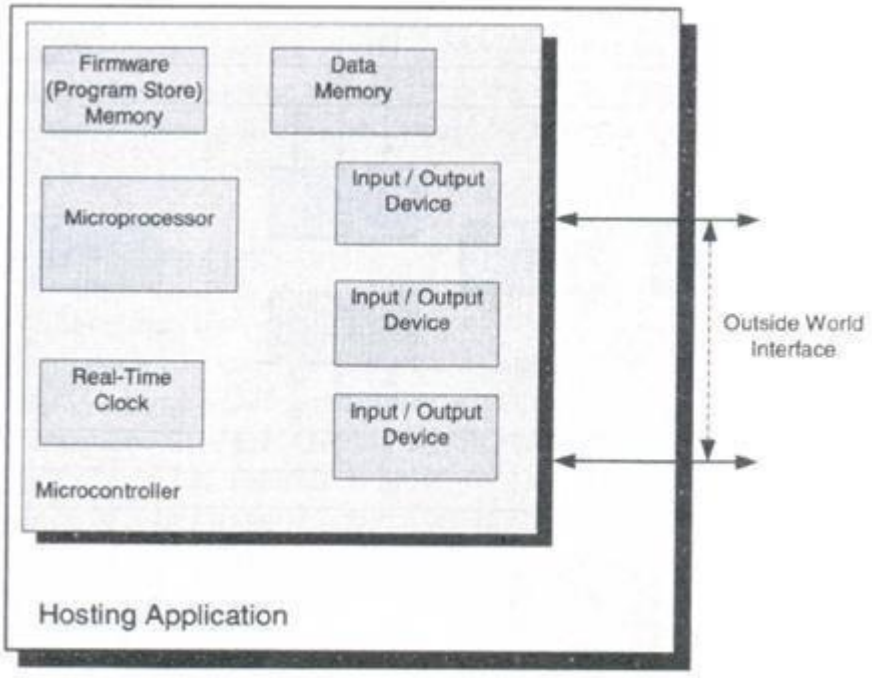
A microcontroller (sometimes abbreviated µC, uC or MCU) is a small computer on a single integrated circuit containing a processor core, memory, andprogrammable input/output peripherals. Program memory in the form of NOR flash or OTP

ROM is also often included on chip, as well as a typically small amount of RAM. Microcontrollers are designed for embedded applications, in contrast to the microprocessors used in personal computers or other general purpose applications.

Figure 2.5 shows together the microprocessor core and a rich collection of peripherals and I/O capability into a single integrated circuit.

Microcontrollers are used in automatically controlled products and devices, such as automobile engine control systems, implantable medical devices, remote controls, office machines, appliances, power tools, toys and other embedded systems. By reducing the size and cost compared to a design that uses a separate microprocessor, memory, and input/output devices, microcontrollers make it economical to digitally control even more devices and processes. Mixed signal microcontrollers are common, integrating analog components needed to control non-digital electronic systems.

Figure 2.5 :A block diagram for a microcontroller based system



The digital signal processor

A digital signal processor (DSP) is a specialized microprocessor with an architecture optimized for the operational needs of digital signal processing. A DSP provides fast, discrete-

time, signal-processing instructions. It has Very Large Instruction Word (VLIW) processing capabilities; it processes Single Instruction Multiple Data (SIMD) instructions fast; it processes Discrete Cosine Transformations (DCT) and inverse DCT (IDCT) functions fast. The latter are a must for fast execution of the algorithms for signal analyzing, coding, filtering, noise cancellation, echo-elimination, compressing and decompressing, etc. Figure 2.6 shows the block diagram for a digital signal processor

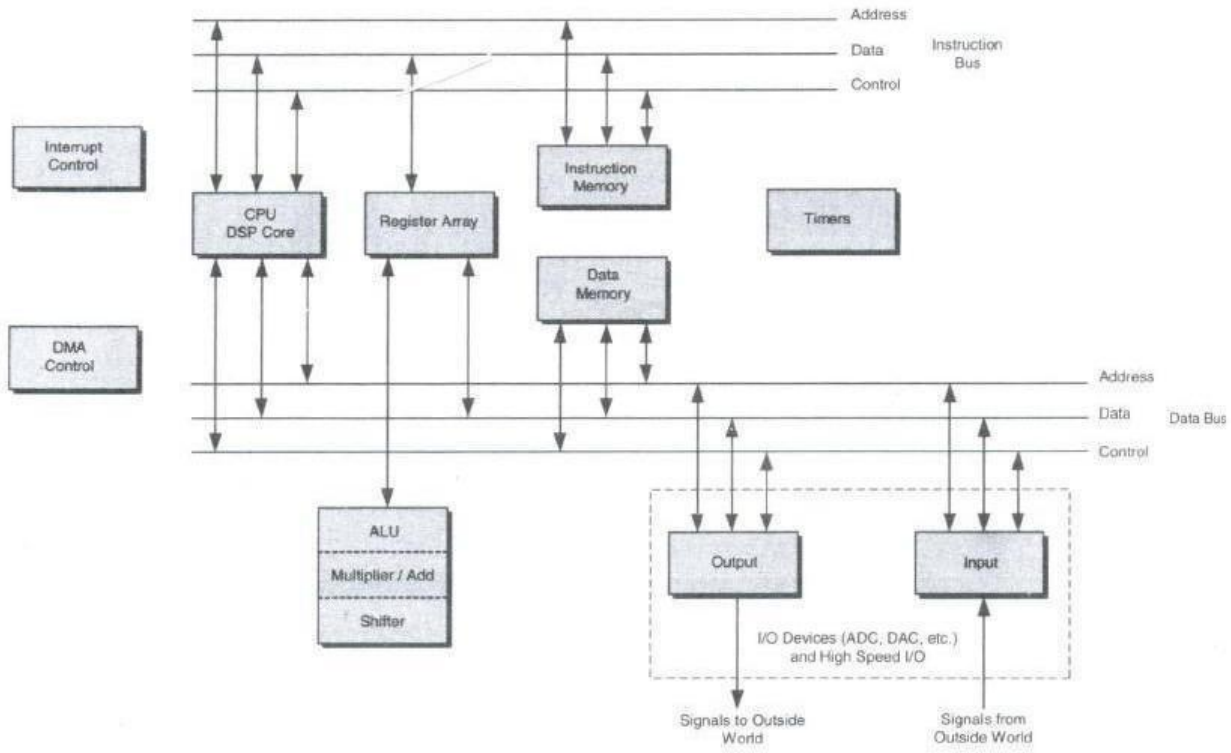


Figure 2.6 A block diagram for a digital signal processor

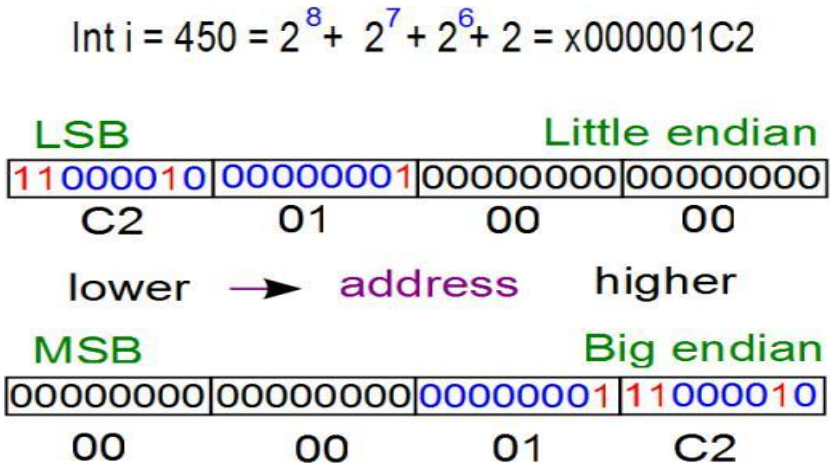
By the standards of general-purpose processors, DSP instruction sets are often highly irregular. One implication for software architecture is that hand-optimized assembly-code routines are commonly packaged into libraries for re-use, instead of relying on advanced compiler technologies to handle essential algorithms.

Hardware features visible through DSP instruction sets commonly include:

- Hardware modulo addressing, allowing circular buffers to be implemented without having to constantly test for wrapping.

- Memory architecture designed for streaming data, using DMA extensively and expecting code to be written to know about cache hierarchies and the associated delays.
- Driving multiple arithmetic units may require memory architectures to support several accesses per instruction cycle
- Separate program and data memories (Harvard architecture), and sometimes concurrent access on multiple data busses
- Special SIMD (single instruction, multiple data) operations
- Some processors use VLIW techniques so each instruction drives multiple arithmetic units in parallel
- Special arithmetic operations, such as fast multiply–accumulates (MACs). Many fundamental DSP algorithms, such as FIR filters or the Fast Fourier transform (FFT) depend heavily on multiply–accumulate performance.
- Bit-reversed addressing, a special addressing mode useful for calculating FFTs
- Special loop controls, such as architectural support for executing a few instruction words in a very tight loop without overhead for instruction fetches or exit testing
- Deliberate exclusion of a memory management unit. DSPs frequently use multi-tasking operating systems, but have no support for virtual memory or memory protection. Operating systems that use **virtual memory** require more time for context switching among processes, which increases latency.

Representing Information



Big endian systems are simply those systems whose memories are organized with the most significant digits or bytes of a number or series of numbers in the upper left corner of a memory page and the least significant in the lower right, just as in a normal spreadsheet.

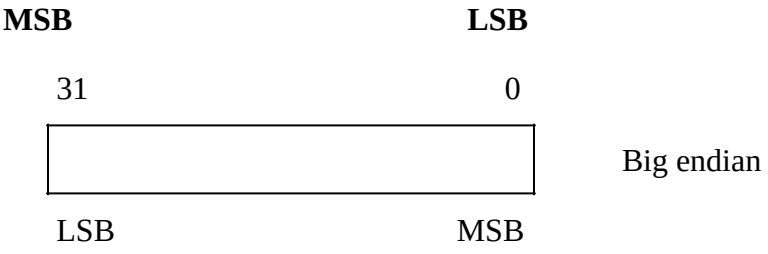
Little endian systems are simply those system whose memories are organized with the least significant digits or bytes of a number or series of numbers in the upper left corner of a memory page and the most significant in the lower right. There are many examples of both types of systems, with the principle reasons for the choice of either format being the underlying operation of the given system.

Understanding numbers

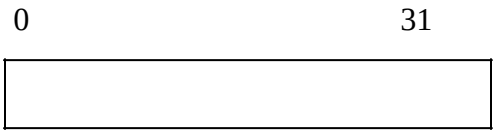
We have seen that within a microprocessor, we don_t have an unbounded numbers of bits with which to express the various kinds of numeric information that we will be working with in an embedded application. The limitation of finite word size can have unintended consequences of results of any mathematical operations that we might need to perform. Let_s examine the effects of finite word size on resolution, accuracy, errors and the propagation of errors in these operation. In an embedded system, the integers and floating point numbers are normally represented as binary values and are stored either in memory or in registers. The expensive power of any number is dependent on the number of bits in the number.

Addresses

In the earlier functional diagram as well as in the block diagram for a microprocessor, we learned that information is stored in memory. Each location in memory has an associated address much like an index in the array. If an array has 16 locations to hold information, it will have 16 indices. if a memory has 16 locations to store information ,it will have 16 addresses. Information is accessed in memory by giving its address.



Big endian



Little endian

Figure Expressing Addresses

Instructions

An **instruction set**, or **instruction set architecture** (**ISA**), is the part of the computer architecture related to programming, including the native data types, instructions, registers, addressing modes, memory architecture, interrupt and exception handling, and external I/O. An ISA includes a specification of the set of opcodes (machine language), and the native commands implemented by a particular processor.

The entities that instructions operate on are denoted Operand. The number of operands that an instruction operates on at any time is called the arity of the operation.

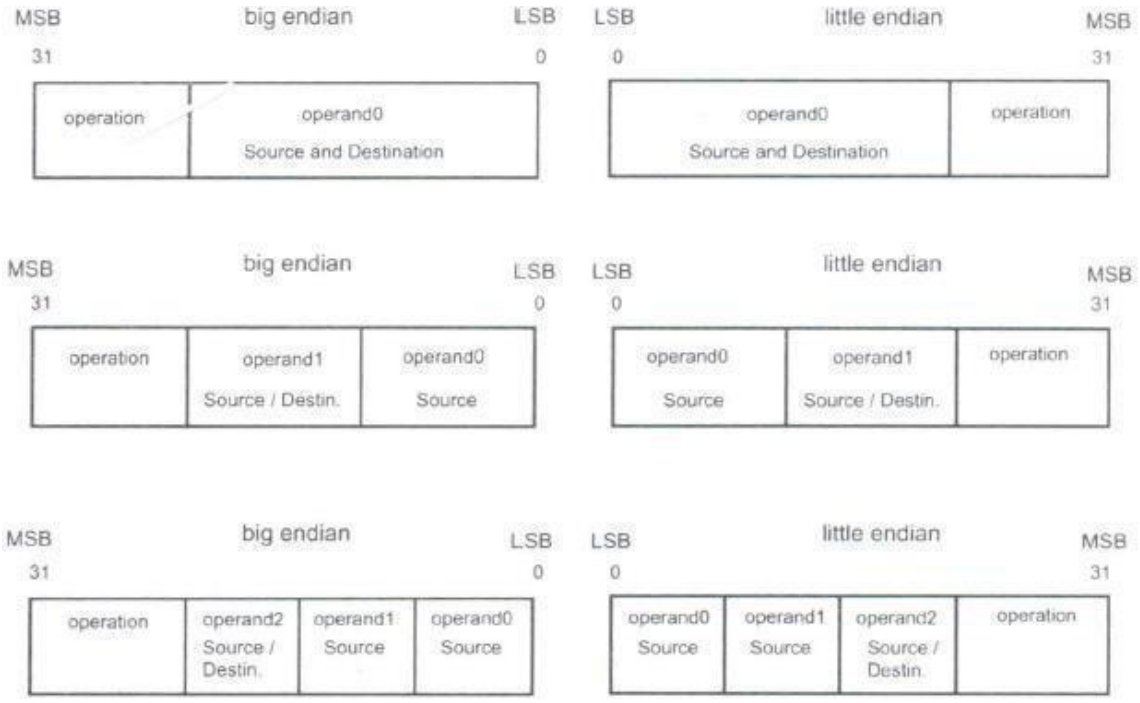


Figure 2.7 Expressing Instructions

General Purpose and Domain Specific Processors:

- **General purpose processors (GPP)** are designed for general purpose computers such as PCs or workstations. The computation speed of a GPP is the main concern and the cost of the GPP is usually much higher than that of DSPs and microcontrollers. All techniques that can increase CPU speed have been applied to GPPs.

- **Domain Specific Processor(DSP)**has separate program and data memories. This allows the processor to fetch an instruction, while simultaneously fetching operands or storing results for a previous instruction. Often it is also possible to fetch multiple data from memory in one clock cycle by using multiple busses and multi port memories or multiple independent data memories.

ASICS:

An Integrated circuit (IC) customized for a particular use and not for general use is called an Application-Specific Integrated Circuit (ASIC). An example of this is a chip which is designed only to run a cell phone. Whereas there are 7400 series and 4000 series integrated circuits as logic building blocks which can be wired together and used in various applications. There are also Application Specific Standard Products (ASSPs) which are intermediate between application specific integrated circuits (ASIC) and standard products.

Application specific integrated circuits (ASICs) can consolidate the work of many chips into a single, smaller, faster package, reducing manufacturing and support costs while boosting the speed of the device built with them. Application specific integrated circuits (ASICs) technology is now so advanced that many functions traditionally implemented in software can be migrated to Application Specific Integrated Circuits (ASICs).

Designers prefer using application specific integrated circuits as it lets them use the power of constantly improving silicon technology to build devices targeted at specific functions, such as routing. Performance improvements of up to threefold can be achieved through the use of application specific integrated circuits (ASIC) when compared with the same functions being executed in software.

ASICs are built to suit need specific application needs; hence we have three broad varieties of ASICs with us:

1. Cell Based Application Specific Integrated Circuits (ASICS)

These feature rich, high-density cell-based ASICs are needed in your high-volume projects. We provide the options needed for high performance system-level integration with leading ASIC process technology and embedded DRAM.

Benefits:

- Highest performance, power and integration
- Cell-based ASICs are offered with a selection of IP cores.
- Perfect for high-volume designs that require the highest density and performance.
- available in a range of process technologies
- Max. clock speed: Up to 1 GHz
- Memory: High-density SRAM and embedded DRAM
- Signal I/Os: Up to 1400
- Usable ASIC gates: Up to 100M
- Featured IP: ARM CPUs, Tensilica CPUs and DSPs, USB 2.0, eDRAM, SerDes, PCI Express, SATA/SAS, SPI4.2
- **IP Core**

  There are a large number of IP cores in cell-based ICs, including CPU cores and analog logic cores. Customers are supported in their own system chip design development by these options. Some of the cores available with us are listed below.

  Core List
- Digital Core
- Analog Core
- SRAM
- eDRAM
- I/O

ServicesProvided During Cell-Based IC Development

Linked processes from logic design to manufacturing features in production of cell-based ICs helps in providing an environment for short turnaround time in mass-producing large-scale integration, advanced performance, and low power consumption.

Benefits:

- Logic design taking layout design into account at an early stage
- Layout design taking manufacturing (DFY or DFM) into account
- Test design improving manufacturing quality
- Feedback from manufacturing to logic design and test design

2. **Gate Array & Embedded Array ASICs**

Gate arrays help reduce the cost, turnaround time and risk while improving power consumption, EMI and performance. These arrays, based on a sea-of-gates architecture, which is customized using only the metallization layers for interconnect. Low NRE costs and fast turnaround time are its results.The manufacturers of GATE ARRAY &EMBEDDED AREEAY ASICs listed on our website has been the worldwide leader in gate arrays since 1993, with over 200 tape outs each year.

Highlights:

From the #1 supplier worldwide, with over 750 tape-outs over the past 3 years

Low price, low NRE cost (as low as $10,000), easy-to-design option, easy Logic consolidation due to small gate counts and small package design.

FPGA-compatible packages also available. In just nine short days from tape out, we can have engineering samples ready to go.

- Max. clock speed: 133 MHz
- Memory: Diffused and metallized SRAM
- Signal I/Os: From 20 to 876
- Usable ASIC Gates: From 1.5K to 1.6M
- Featured IP: DMA controller, interrupt controller, PCI controller, DPLL, UART, timer, parallel interface unit, UART + FIFO, Tensilica CPUs and DSPs
- Master slices: 158

3. **EDRAM Application Specific Integrated Circuits (ASICS)**

We are associated with the leaders of the EDRAM ASICs producers and manufacturers. This gives you only the best and most preferred components in the electronic industry. Following are some of the features that make EDRAM, the ideal solution for a wide variety of applications.Random access time an order of magnitude shorter than competitor offerings. Fully CMOS-compatible process using a single fab and few extra masks to minimize cost.

Applications of ASICS:

These characteristics have helped fuel the growing popularity of EDRAM for a variety of applications – from communications systems to home electronics, from enterprise servers to entertainment systems. The success complex system LSI chip for the Nintendo Wii™ and Microsoft® XBox 360™ has propelled fabrication volume to many millions of devices.

PLDS:

A **programmable logic device** (**PLD**) is an electronic component used to build reconfigurable digital circuits. Unlike integrated circuits (IC) which consist of logic gates and have a fixed function, a PLD has an undefined function at the time of manufacture. Before the PLD can be used in a circuit it must be programmed (reconfigured) by using a specialized program.

There are three fundamental **types** of standard **PLDs**: PROM, PAL, and PLA. A fourth **type of PLD**, which is discussed later, is the Complex Programmable Logic Device (**CPLD**), e.g., Field Programmable Gate Array (FPGA). A typical **PLD may** have hundreds to millions of gates

PROMS

The first of the simple PLDs were *Programmable Read-Only Memories (PROMs),* which appeared on the scene in 1970. One way to visualize the manner in which these devices perform their magic is to consider them as consisting of a **fixed array** of AND functions driving a **programmable array** of OR functions.

When PROMs are employed to implement combinational logic, they are useful for equations requiring a large number of product terms, but they can only support relatively few inputs because every input combination is always decoded and used. This led engineers to start considering alternative architectures...

PLAs

In order to address the limitations imposed by the PROM architecture, the next step up the PLD evolutionary ladder was that of *Programmable Logic Arrays (PLAs),* which first became available circa 1975. These were the most user-configurable of the simple PLDs, because **both** the AND and OR arrays were **programmable**.

On the downside, signals take a relatively long time to pass through programmable links as opposed to their predefined counterparts (this effect was much more pronounced in the early devices with their

humongous [by today's standards] structures). Thus, the fact that both their AND and OR arrays were programmable meant that PLAs were significantly slower than PROMs.

PALs and GALs

Thus, in order to address the speed problems posed by PLAs, a new class of device called *Programmable Array Logic (PAL)* was introduced in the late 1970s. Conceptually, a PAL is almost the exact opposite to a PROM, because it has a **programmable** AND array and a **predefined** OR array.

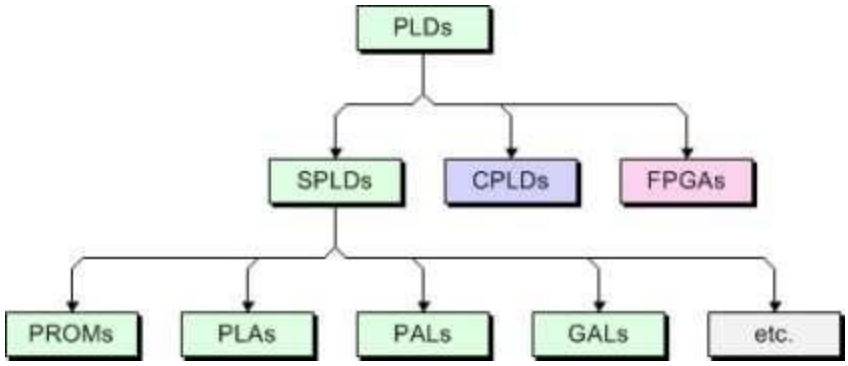
In 1983, **Lattice Semiconductor Corporation** introduced a suite of *Generic Array Logic (GAL)* devices, which provided sophisticated CMOS-based electrically erasable (E^2) variations on the PAL concept.

The advantage of PALs and GALs (as compared to PLAs) is that they are faster because only one of their arrays is programmable. On the downside, PALs and GALs are more limited because they only allow a restricted number of product terms to be OR-ed together

CPLDs

The tail end of the 1970s and the early 1980s began to see the emergence of more sophisticated PLD devices. In order to distinguish these little scamps from their less-sophisticated ancestors (which still find use to this day), these new devices were referred to as *Complex PLDs (CPLDs)*. Perhaps not surprisingly, it subsequently became common practice to refer to the original, less-pretentious versions as *Simple PLDs (SPLDs)*.

Just to make life more confusing, some people understand the terms PLD and SPLD to be synonymous, while others regard PLD as being a superset that encompasses both SPLDs and CPLDs. More recently, the term PLD (Programmable Logic Device) has come to be understood to refer to SPLDs, CPLDs, and FPGAs as illustrated below:



Leading the fray were the inventors of the original PAL devices —the guys and gals at Monolithic Memories Inc. (MMI) —who introduced a component they called a MegaPAL. This was an 84-pin device that essentially comprised four standard PALs with some interconnect linking them together. Unfortunately, the MegaPAL consumed a disproportionate amount of power and it was generally perceived to offer little advantage compared to using four individual devices.
The big leap forward occurred in 1984, when newly formed **Altera Corporation** (which was founded in 1983) introduced a CPLD based on a combination of CMOS and EPROM technologies. Using CMOS allowed Altera to achieve tremendous functional density and complexity while consuming relatively little power. And basing the programmability of these devices on EPROM cells made them ideal for use in development and prototyping environments.

MEMORY

In a system, there are various types of memories. Figure 1.4 shows a chart for the various forms of memories that are present in systems. These are as follows:

(i) Internal RAM of 256 or 512 bytes in a microcontroller for registers, temporary data and stack.

(ii) (ii) Internal ROM/PROM/EPROM for about 4 kB to 16 kB of program (in the case of microcontrollers).

(iii) (iii) External RAM for the temporary data and stack (in most systems).

(iv) (iv) Internal caches (in the case of certain microprocessors).

(v) (v) EEPROM or flash (in many systems saving the results of processing in nonvolatile memory: for example, system status periodically and digital-camera images, songs, or speeches after a suitable format compression).

(vi) (vi) External ROM or PROM for embedding software (in almost all nonmicrocontroller- based systems).

(vii) (vii) RAM Memory buffers at the ports. (viii) Caches (in superscalar microprocessors).

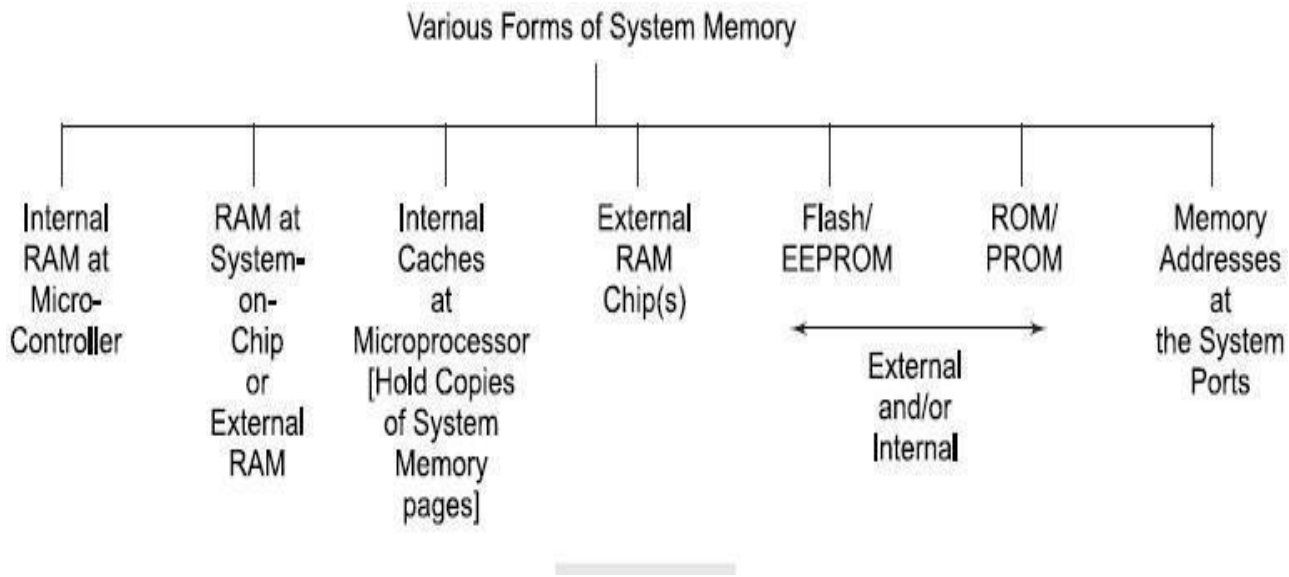


Table 1.1 gives the functions assigned in the embedded systems to the memories. ROM or PROM or EPROM embeds the embedded software specific to the system.

Table1.1

Functions Assigned to the Memories in a System

Memory Needed	Functions
ROM or EPROM	Storing Application programs from where the processor fetches the instruction codes. Storing codes for system booting, initializing, Initial input data and Strings. Codes for RTOS. Pointers (addresses) of various service routines.
RAM (Internal and External) and RAM for buffer	Storing the variables during program run and storing the stack. Storing input or output buffers, for example, for speech or image.
EEPROM or Flash	Storing non-volatile results of processing.
Caches	Storing copies of instructions and data in advance from external memories and storing temporarily the results during fast processing.

The memory unit in an embedded system should have low access time and high density (a memory chip has greater density if it can store more bits in the same amount of space). Memory in an embedded system consists of ROM (only read operations permitted) and RAM (read and write operations are permitted). The contents of ROM are non-volatile (power failure does not erase the contents) while RAM is volatile. The classification of the ROM is given in Figure 3.1. ROM stores the program code while RAM is used to store transient input or output data. Embedded systems generally do not possess secondary storage devices such as magnetic disks. As programs of embedded systems are small there is no need for virtual storage.

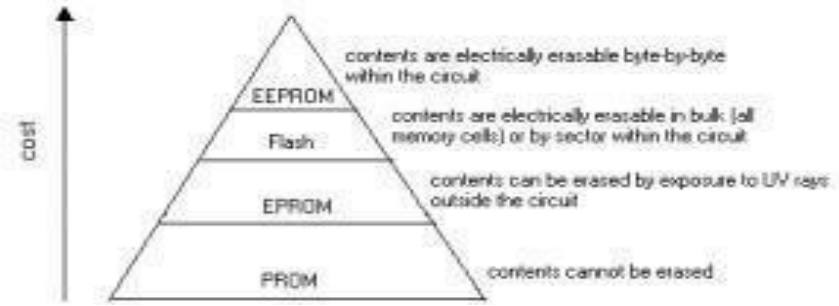


Figure 3.1: Classifications of ROM

Volatile memory

A primary distinction in memory types is volatility. Volatile memories only hold their contents while power is applied to the memory device. As soon as power is removed, the memories lose their contents; consequently, volatile memories are unacceptable if data must be retained when the memory is switched off. Examples of volatile memories include static RAM (SRAM),synchronous static RAM (SSRAM), synchronous dynamic RAM (SDRAM), and FPGA on-chip memory.

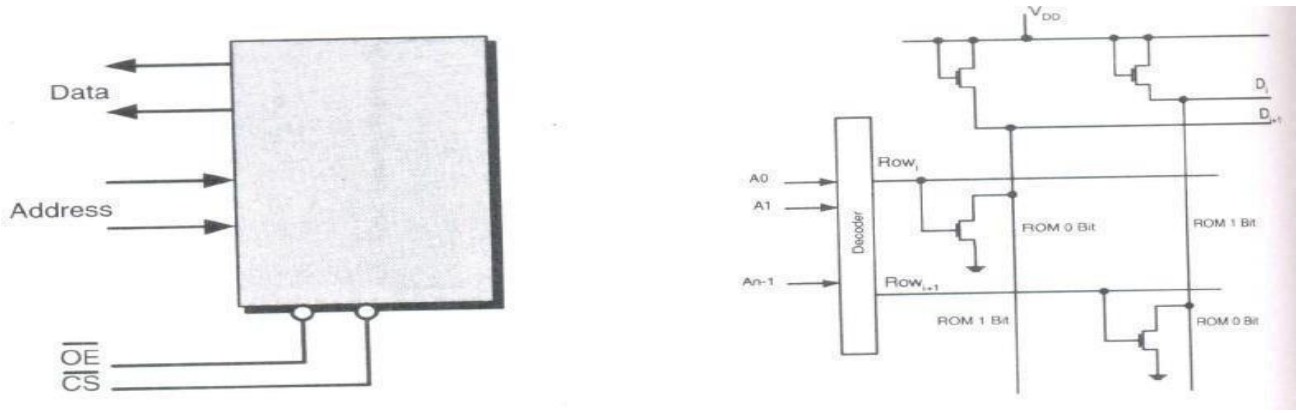
Nonvolatile memory

Non-volatile memories retain their contents when power is switched off, making them good choices for storing information that must be retrieved after a system power-cycle. Processor boot-code, persistent application settings, and FPGA configurationdataaretypicallystoredinnon-volatilememory.Althoughnon-volatile memory has the advantage of retaining its data when power is removed, it is typically much slower to write to than volatile memory, and often has more complex writing and erasing procedures. Non-volatile memory is also usually only guaranteed to be erasable a given number of times, after which it may fail. Examples of non- volatile memories include all types

of flash, EPROM, and EEPROM. Most modern embedded systems use some type of flash memory for non-volatile storage. Many embedded applications require both volatile and non-volatile memories because the two memory types serve unique and exclusive purposes. The following sections discuss the use of specific types of memory in embedded systems.

ROM Overview

Although there are exceptions, the ROM is generally viewed as read only device. A high level interface to the ROM is as shown in figure 3.2. when the ROM is implemented,positions in the array that are to store a logical 0 have a transistor connected as shown in figure. Those positions intended to store a logical 1 have none.



Read Operation

A value is read from a ROM by asserting one of the row lines. Those rows in which there is a transistor will be pulled to ground thereby expressing a logical 0. Those without the transistor will express a logical 1. Timing for a ROM read operation is given in figure 3.3.

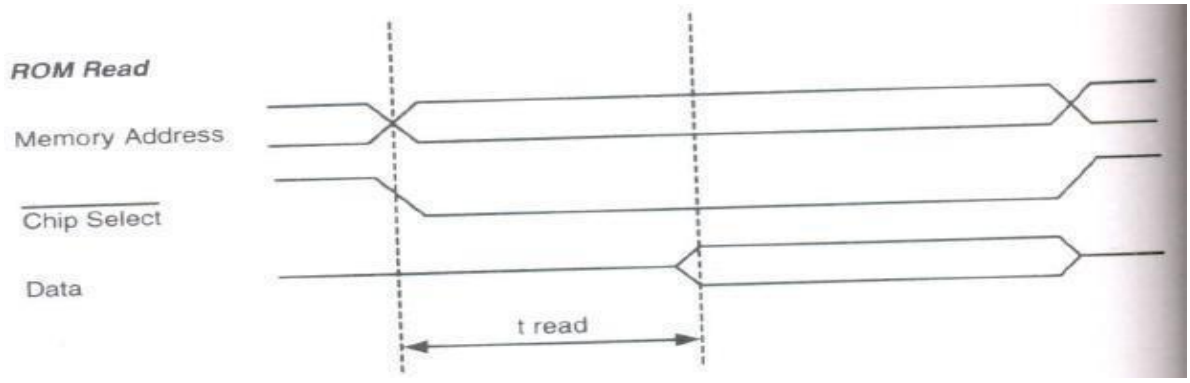


Figure 3.3 The ROM –read operation timing

Static RAM overview

A high level interface to the SRAM is very similar to that for the ROM.The major differences arise from support for write capability. Figure 3.4 represents the major I/O signals and a typical cell in an SRAM array.

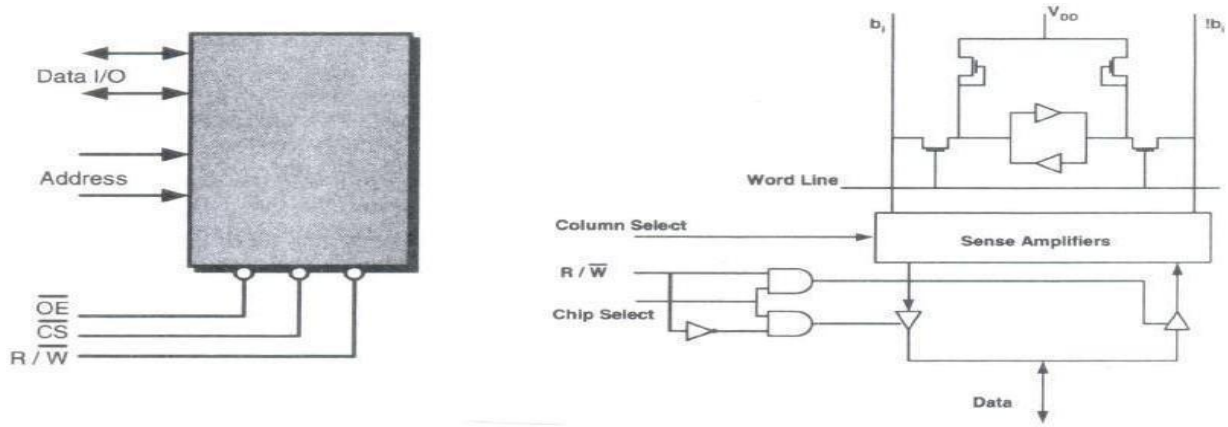


Figure 3.4 The SRAM – inside and outside

Write operation

A value is written into the cell by applying a signal to bi and bibar through the write/sense amplifiers. Asserting the word line causes the new value to be written into the latch.

Read Operation

A value is read from the cell by first precharging bi and bibar to a voltage that is halfway between a 0 and 1.the values are sensed and amplified by write/sense amplifier.

Typical timimg for a read and write operation is shown in Figure 3.5 .

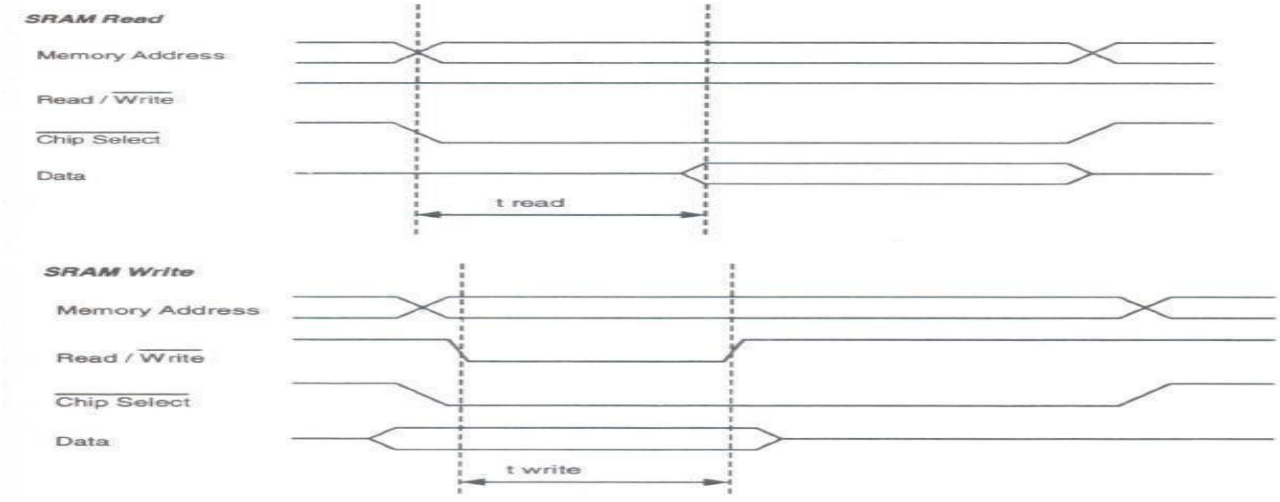


Figure 3.5 timing for the SRAM-read and write operation

SDRAM

SDRAM is another type of volatile memory. It is similar to SRAM, except that it is dynamic and must be refreshed periodically to maintain its content. The dynamic memory cells in SDRAM are much smaller than the static memory cells used in SRAM. This difference in size translates into very high-capacity and low-cost memory devices. In addition to the refresh requirement, SDRAM has other very specific interface requirements which typically necessitate the use of special controller hardware. Unlike SRAM, which has a static set of address lines, SDRAM divides up its memory space into banks, rows, and columns. Switching between banks and rows incurs some overhead, so that efficient use of SDRAM involves the careful ordering of accesses. SDRAM also multiplexes the row and column addresses over the same address lines, which reduces the pin count necessary to implement a given size of SDRAM. Higher speed varieties of SDRAM such as DDR, DDR2, and DDR3 also have strict signal integrity requirements which need to be carefully considered during the design of the PCB. SDRAM devices are among the least expensive and largest-capacity types of RAM devices available, making them one of the most popular. Most modern embedded systems use SDRAM. A major part of an SDRAM interface is the SDRAM controller. The SDRAM controller manages all the address-multiplexing, refresh and row and bank switching tasks, allowing the rest of the system to access SDRAM without knowledge of its internal architecture.

Dynamic RAM Overview

Larger microcomputer systems use Dynamic RAM (DRAM) rather than Static RAM (SRAM) because of its lower cost per bit. DRAMs require more complex interface circuitry because of their multiplexed address bus and because of the need to refresh each memory cell periodically.

A typical DRAM memory is laid out as a square array of memory cells with an equal number of rows and columns. Each memory cell stores one bit. The bits are addressed by using half of the bits (the most significant half) to select a row and the other half to select a column.

Each DRAM memory cell is very simple – it consists of a capacitor and a MOSFET switch. A DRAM memory cell is therefore much smaller than an SRAM cell which needs at least two gates to implement a flip-flop. A typical DRAM array appears as illustrated in figure 3.6 .

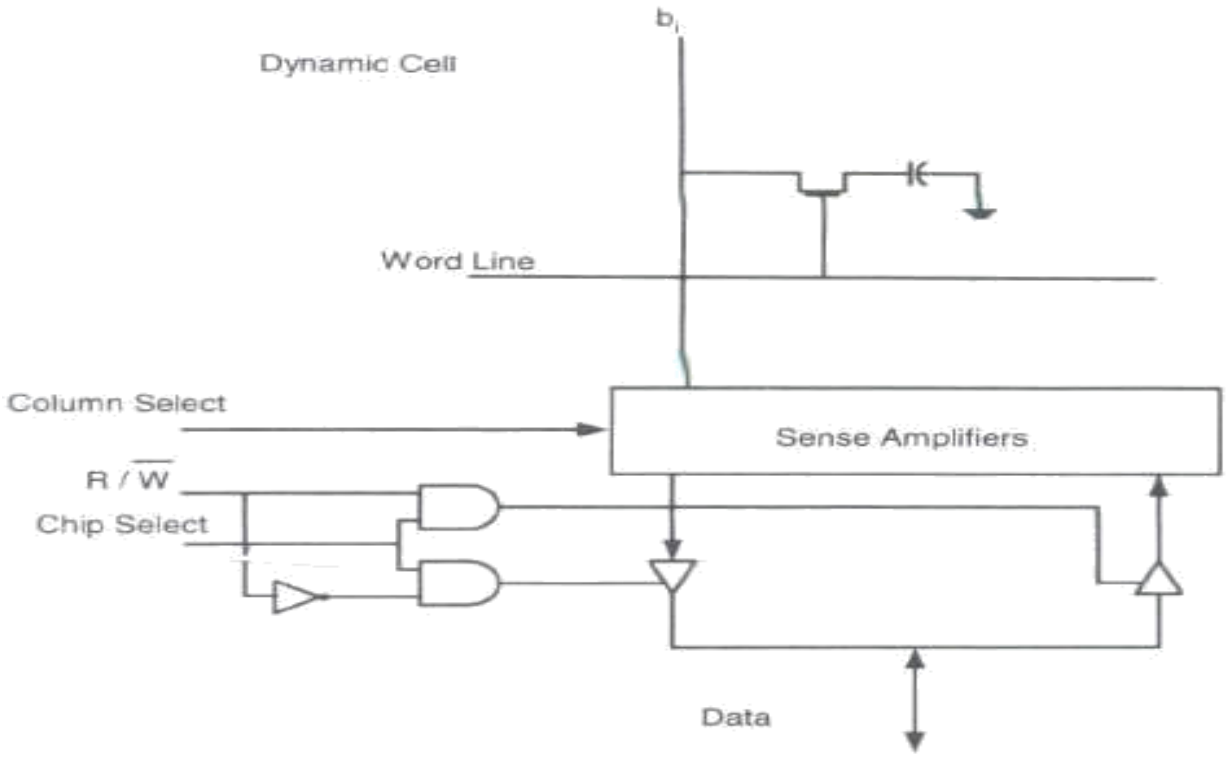


Figure 3.6 The DRAM inside

Read Operation

A value is read from the cell by first precharging bi and bibar to a voltage that is halfway between a 0 and 1. Asserting the word line enables the stored signal onto bi. If the stored value is a logical 1,through charge sharing,the value on line bi will increase. Conversely,if the stored value is a logical 0, charge sharing will cause the value on bi to decrease. The change in the values are sensed and amplified by write/sense amplifier

Write operation

A value is written into the cell by applying a signal to bi and bibar through the write/sense amplifiers. Asserting the word line charges the capacitor if a logical 1 is to be stored and discharges if it a logical 0 to be stored.

Typical DRAM read and write timing is given in figure 3.7 .

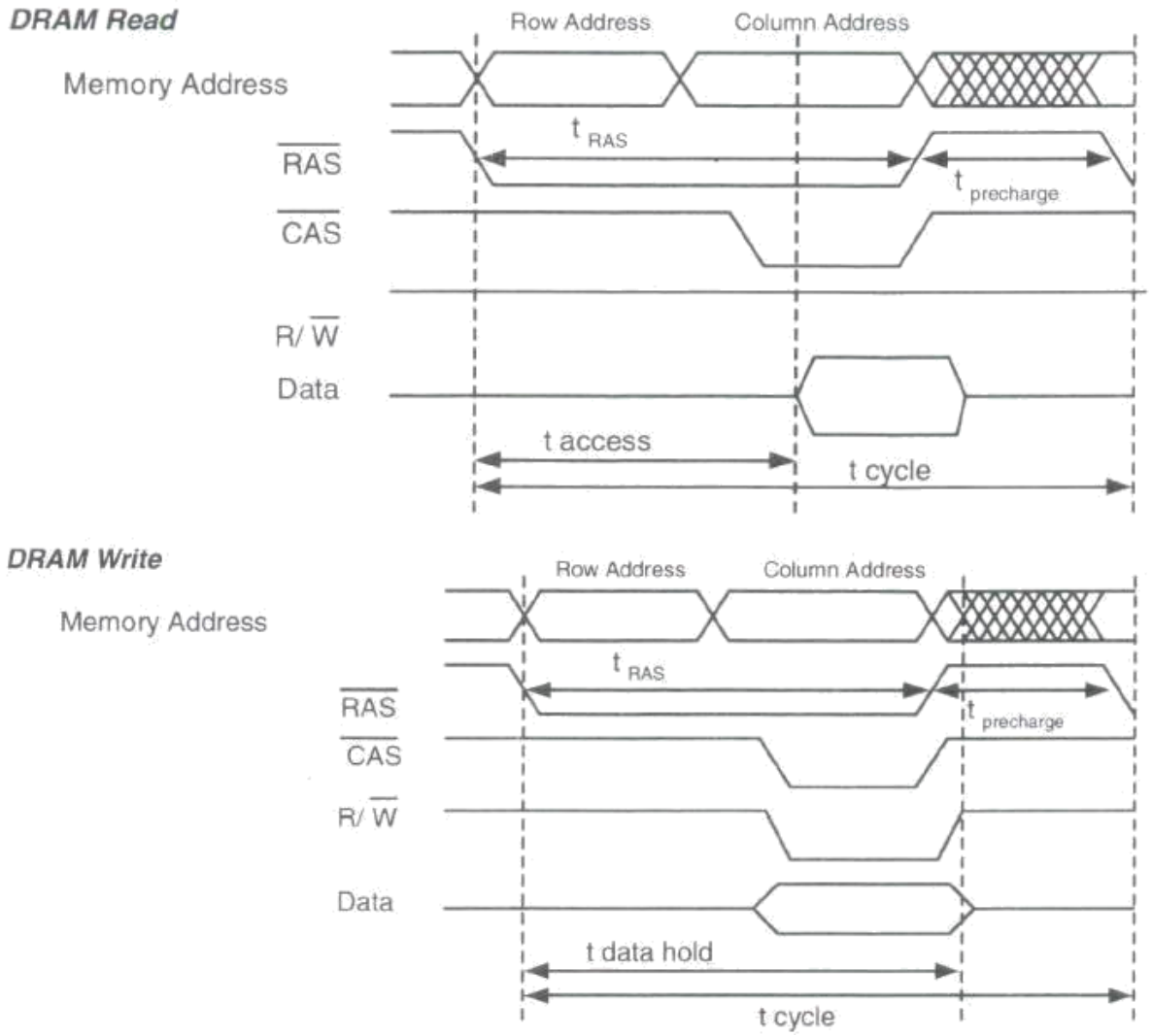


Figure 3.7 : Timing for the DRAM read and write cycles

The memory map

As a first step towards understanding the memory subsystem in an embedded application, we begin with a memory map. The map specifies the allocation and use of each location in the physical memory address space. A typical memory map for a small 16 bit machine is presented in figure 3.11

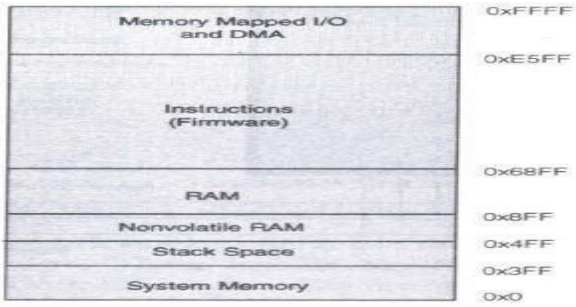


Figure 3.11 : Basic Memory map

This is the primary physical memory. From a high level perspective, the memory subsystem is comprised of two basic types:ROM and RAM. It is possible for the required code and data space to exceed total available primary memory.under such circumstances, one must use techniques called virtual memory and overlays to accommodate the expanded needs.

Memory subsystem Architecture

The block labeled memory in the diagrrm for a vonneumann machine is sctually comprised of a memory components of different size, kinds, and speeds arranged in a heirarchial manner and designed to coopereate with each other.such a hierarchy is given in figure 3.12 .

Figure 3.12 : Typical memory hierarchy utilizing a variety of memory types.

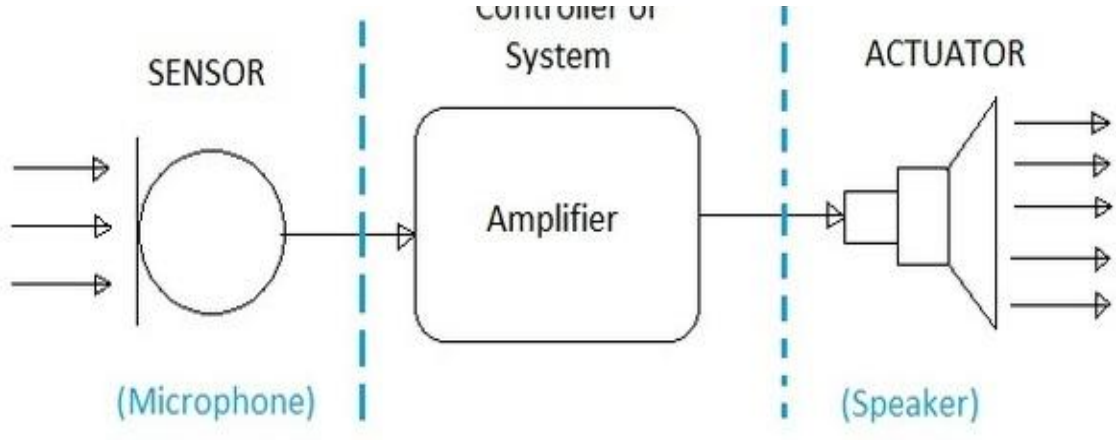
At the top are the slowest,largest and least expensive memories. These are known as secondary memory. At the bottom are the smallest,fastest,called cache memory. These are typically higher speed SRAM. These devices are more expensive.

SENSORS AND ACTUATORS:

- A sensor is a device that changes a physical parameter to an electrical output. As against, an actuator is a device that converts an electrical signal to a physical output. ... Sensor generates electrical signals while an actuator results in the production of energy in the form of heat or motion
- Sensors and Actuators are essential elements of the embedded systems. These are used in several real-life applications such as flight control system in an aircraft, process control systems in nuclear reactors, power plants that require to be operated on an automated control. Sensors and Actuators mainly differ by the purpose both provide, the sensor is used to monitor the changes in the environment by using measurands while the actuator is

used when along with monitoring the control is also applied such as to control the physical change

Sensors sense - that is to say they act as the eyes and ears of your device, and detect changes in the environment around them, providing information for a processor, MCU or other system to react toActuators, on the other hand, provide a mechanical response according to the input provided by the sensors and (usually) processed by other electronics. In some cases, a sensor can be used as an actuator and vice-versa; as is the case with a few microphones and motors.



Difference between sensors and actuators

Basis for comparison	**sensors**	**actuators**
Basic	Used to measure the continuous and discrete process variables.	Impel continuous and discrete processes parameters.
Placed at	Input port	Output port
Outcome	Electrical signal	Heat or motion
Example	Magnetometer, Cameras, Accelerometer, microphones.	LED, Laser, Loudspeaker, Solenoid, motor controllers.

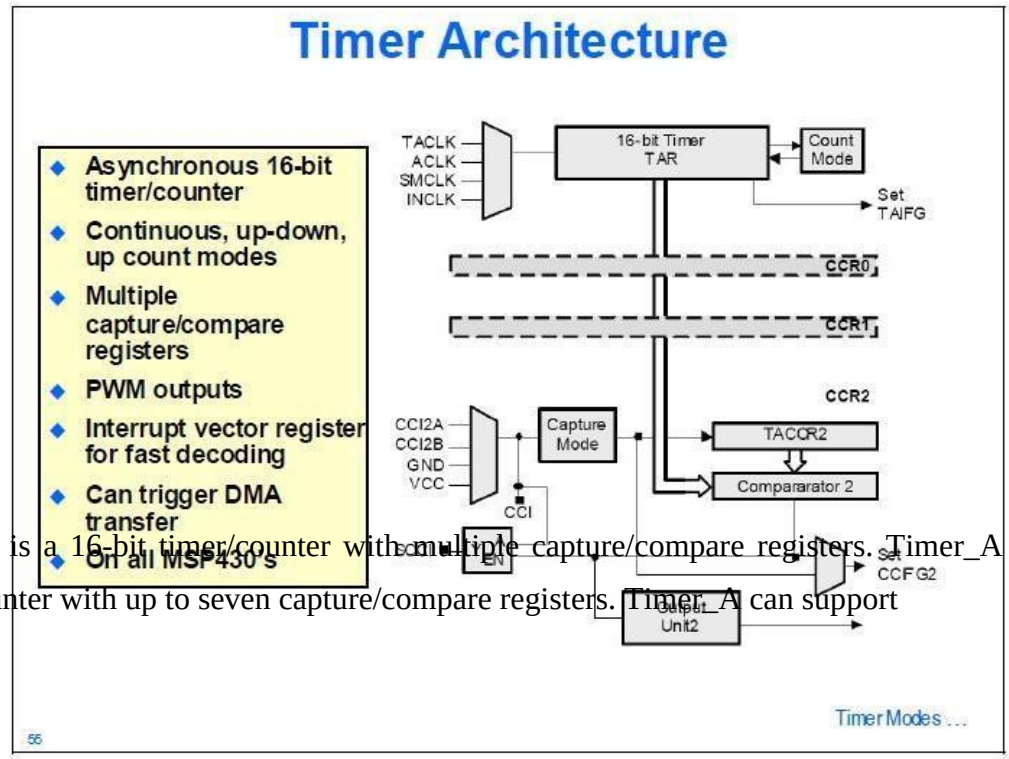
UNIT III
EMBEDDED FIRMWARE

Embedded firmware is the flash memory chip that stores specialized software running in a chip in an embedded device to control its functions .Firmware in embedded systems fills the same purpose as a ROM but can be updated more easily for better adaptability to conditions or interconnecting with additional equipment. For example, embedded software may run on ROM chips. Also, embedded software is often the only computer code running on a piece of hardware while firmware can also refer to the chip that houses a computer's EFI or BIOS, which hands over control to an OS that in turn launches and controls programs..

Timer & Real Time Clock (RTC), PWM control, timing generation and measurements. Analog interfacing and data acquisition: ADC and Comparator in MSP430, data transfer using DMA.

Case Study: MSP430 based embedded system application using ADC & PWM demonstrating peripheral intelligence. ―Remote Controller of Air Conditioner Using MSP430‖.

Timer Architecture



Timer_A is a 16-bit timer/counter with multiple capture/compare registers. Timer_A is a 16-bit timer/counter with up to seven capture/compare registers. Timer_A can support

multiple capture/compares, PWM outputs, and interval timing. Timer_A also has extensive interrupt capabilities. Interrupts may be generated from the counter on overflow conditions and from each of the capture/compare registers.

Timer_A features include:

Asynchronous 16-bit timer/counter with four operating modes

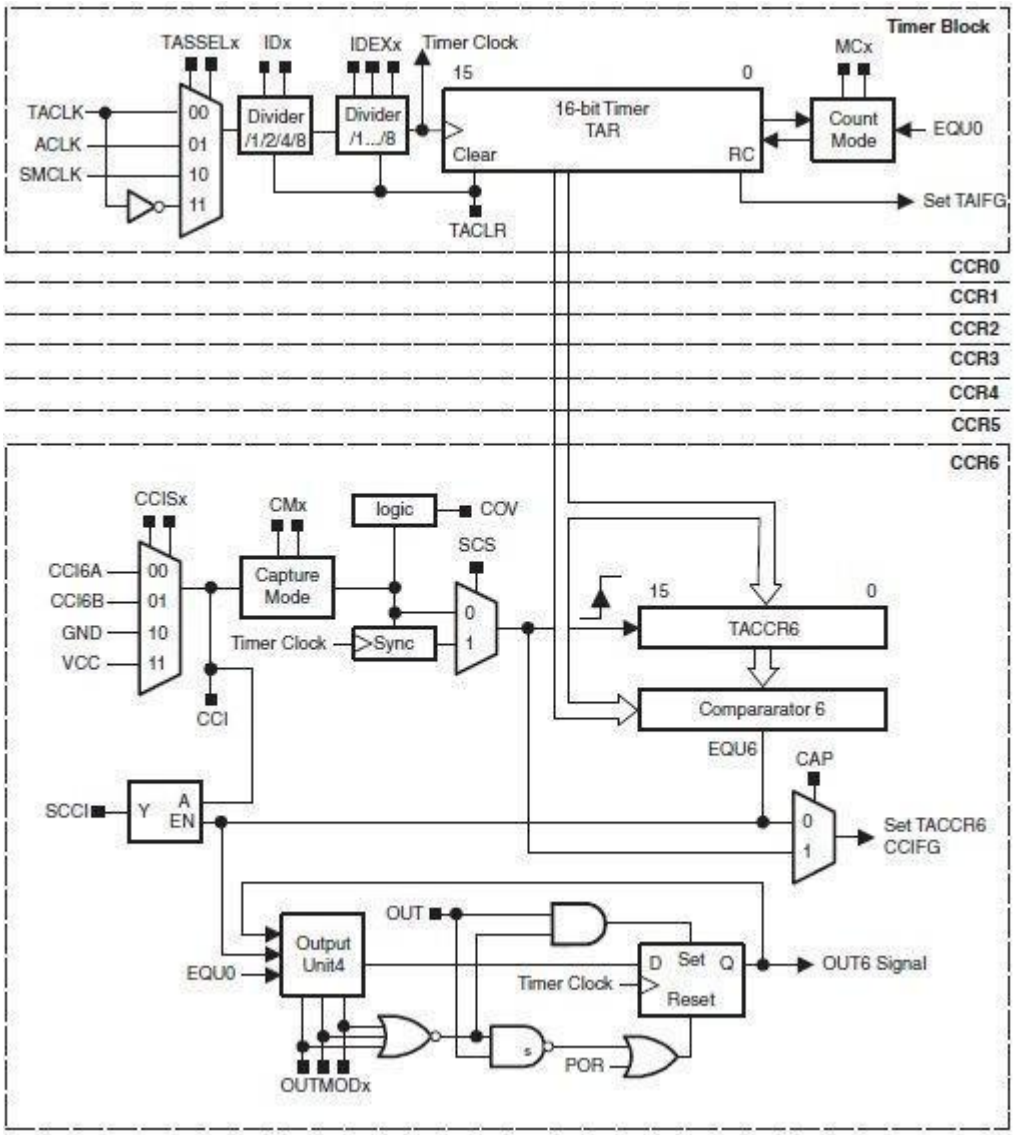
Selectable and configurable clock source

Up to seven configurable capture/compare registers

Configurable outputs with PWM capability

Asynchronous input and output latching

Interrupt vector register for fast decoding of all Timer_A interrupts



The 16-bit timer/counter register, TAR, increments or decrements (depending on mode of operation) with each rising edge of the clock signal. TAR can be read or written with software. Additionally, the timer can generate an interrupt when it overflows.

TAR may be cleared by setting the TACLR bit. Setting TACLR also clears the clock divider and count direction for up/down mode.

Clock Source Select and Divider

The timer clock TACLK can be sourced from ACLK, SMCLK, or externally via TACLK. The clock source is selected with the TASSELx bits. The selected clock source may be passed directly to the timer or divided by 2, 4, or 8, using the IDx bits The selected clock source can be further divided by 2, 3, 4, 5, 6, 7, or 8 using the IDEXx bits.The TACLK dividers are reset when TACLR is set.

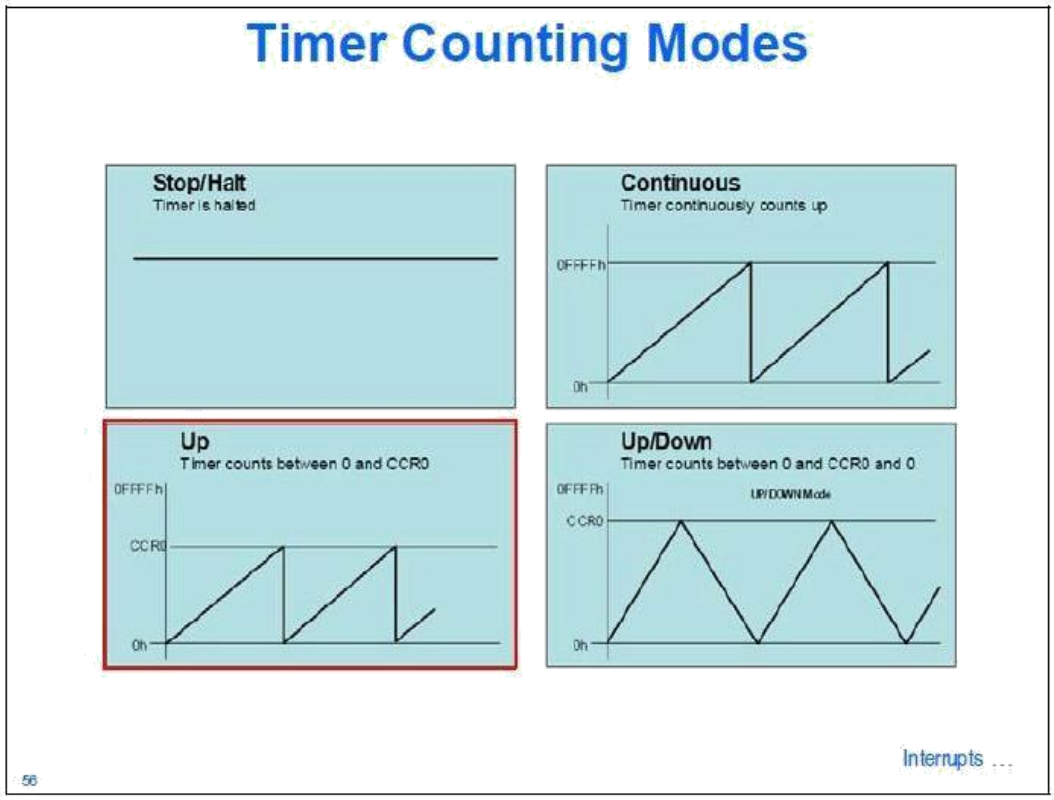
The timer may be started, or restarted in the following ways:

The timer counts when MCx > 0 and the clock source is active.

When the timer mode is either up or up/down, the timer may be stopped by writing 0 to TACCR0. The timer may then be restarted by writing a nonzero value to TACCR0. In this scenario, the timer starts incrementing in the up direction from zero.

MCx	Mode	Description
00	Stop	The timer is halted.
01	Up	The timer repeatedly counts from zero to the value of TACCR0
10	Continuous	The timer repeatedly counts from zero to 0FFFFh.
11	Up/down	The timer repeatedly counts from zero up to the value of TACCR0 and backdown to zero.

Counting Modes



Up Mode

The up mode is used if the timer period must be different from 0FFFFh counts. The timer repeatedly counts up to the value of compare register TACCR0, which defines the period, as shown in Figure. The number of timer counts in the period is TACCR0+1. When the timer value equals TACCR0 the timer restarts counting from zero. If up mode is selected when the timer value is greater than TACCR0, the timer immediately restarts counting from zero

The TACCR0 CCIFG interrupt flag is set when the timer counts to the TACCR0 value. The TAIFG interrupt flag is set when the timer counts from TACCR0 to zero.

Continuous Mode:

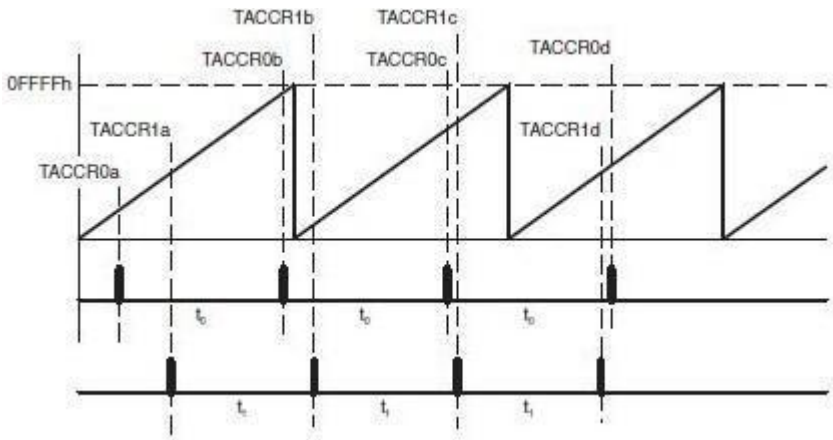
In the continuous mode, the Continuous Mode timer repeatedly counts up to 0FFFFh and restarts from zero as shown in Figure. The capture/compare register TACCR0 works the same way as the other capture/compare registers.

The TAIFG interrupt flag is set when the timer counts from 0FFFFh to zero

Use of the Continuous Mode

The continuous mode can be used to generate independent time intervals and output frequencies. Each time an interval is completed, an interrupt is generated. The next time interval is added to the TACCRx register in the interrupt service routine. Figure shows two separate time intervals t0 and t1 being added to the capture/compare registers. In this usage, the time interval is controlled by hardware, not software, without impact from interrupt latency. Up to n (Timer_An), where n = 0 to 7, independent time intervals or output frequencies can be generated using capture/compare registers.

Time intervals can be produced with other modes as well, where TACCR0 is used as the period register. Their handling is more complex since the sum of the old TACCRx data and the new period can be higher than the TACCR0 value. When the previous TACCRx value plus tx is greater than the TACCR0 data, the TACCR0 value must be subtracted to obtain the correct time interval.



Up/Down Mode

The up/down mode is used if the timer period must be different from 0FFFFh counts, and if symmetrical pulse generation is needed. The timer repeatedly counts up to the value of compare register TACCR0 and back down to zero,

The count direction is latched. This allows the timer to be stopped and then restarted in the same direction it was counting before it was stopped. If this is not desired, the TACLR bit must be set to clear the direction. The TACLR bit also clears the TAR value and the TACLK divider.

In up/down mode, the TACCR0 CCIFG interrupt flag and the TAIFG interrupt flag are set

only once during a period, separated by 1/2 the timer period. The TACCR0 CCIFG interrupt flag is set when the timer counts from TACCR0-1 to TACCR0, and TAIFG is set when the timer completes counting down from 0001h to 0000h.

Use of the Up/Down Mode

The up/down mode supports applications that require dead times between output signals (see section *Timer_A Output Unit*). For example, to avoid overload conditions, two outputs driving an H-bridge must never be in a high state simultaneously. In the example shown in Figure 12-9 the t_{dead} is:

$t_{dead} = t_{timer} \times (TACCR1 - TACCR2)$

With:

t_{dead} = Time during which both outputs need to be inactive

t_{timer} = Cycle time of the timer clock

TACCRx = Content of capture/compare register x

The TACCRx registers are not buffered. They update immediately when written to. Therefore, any required dead time will not be maintained automatically.

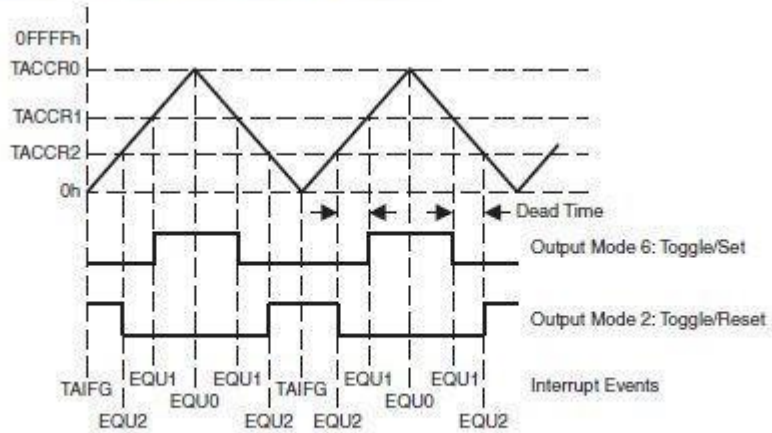


Figure 12-9. Output Unit in Up/Down Mode

Capture/Compare Blocks:

Three or five identical capture/compare blocks, TACCRx, are present in Timer_A. Any of the blocks may be used to capture the timer data, or to generate time intervals. Capture Mode The capture mode is selected when CAP = 1. Capture mode is used to record time events. It can be used for speed computations or time measurements. The capture inputs CCIxA and CCIxB are connected to external pins or internal signals and are selected with the CCISx bits. The CMx bits select the capture edge of the input signal as rising, falling, or both. A capture occurs on the selected edge of the input signal. If a capture occurs:

The timer value is copied into the TACCRx register

The interrupt flag CCIFG is set

The input signal level can be read at any time via the CCI bit. MSP430x5xx family devices may have different signals connected to CCIxA and CCIxB. Refer to the device-specific data sheet for the connections of these signals. The capture signal can be asynchronous to the timer clock and cause a race condition. Setting the SCS bit will synchronize the capture with the next timer clock. Setting the SCS bit to synchronize the capture signal with the timer clock is recommended.

Overflow logic is provided in each capture/compare register to indicate if a second capture was performed before the value from the first capture was read. Bit COV is set

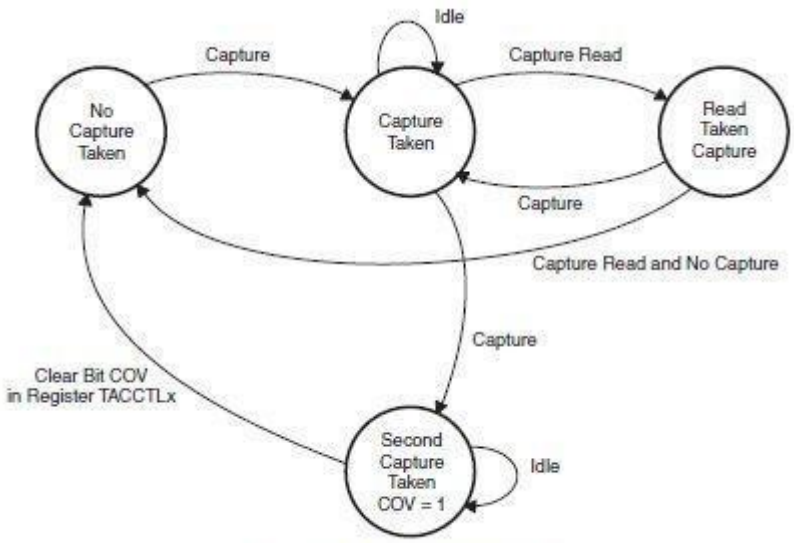


Figure 12-11. Capture Cycle

Capture Mode:

Captures can be initiated by software. The CMx bits can be set for capture on both edges. Software then sets CCIS1 = 1 and toggles bit CCIS0 to switch the capture signal between VCC and GND, initiating a capture each time CCIS0 changes state:

MOV #CAP+SCS+CCIS1+CM_3,&TACCTLx ; Setup TACCTLx XOR #CCIS0,&TACCTLx ; TACCTLx = TAR

Compare Mode

The compare mode is selected when CAP = 0. The compare mode is used to generate PWM output signals or interrupts at specific time intervals. When TAR counts to the value in a TACCRx:

Interrupt flag CCIFG is set

Internal signal EQUx = 1

EQUx affects the output according to the output mode

The input signal CCI is latched into SCCI

Output Unit:

Each capture/compare block contains an output unit. The output unit is used to generate output signals such as PWM signals. Each output unit has eight operating modes that generate signals based on the EQU0 and EQUx signals.

Output Modes

The output modes are defined by the OUTMODx bits and are described in Table. The OUTx signal is changed with the rising edge of the timer clock for all modes except mode 0. Output modes 2, 3, 6, and 7 are not useful for output unit 0 because EQUx = EQU0.

Output Example—Timer in Up Mode

The OUTx signal is changed when the timer *counts* up to the TACCRx value, and rolls from TACCR0 to zero, depending on the output mode. An example is shown in Figure 12-12 using TACCR0 and TACCR1.

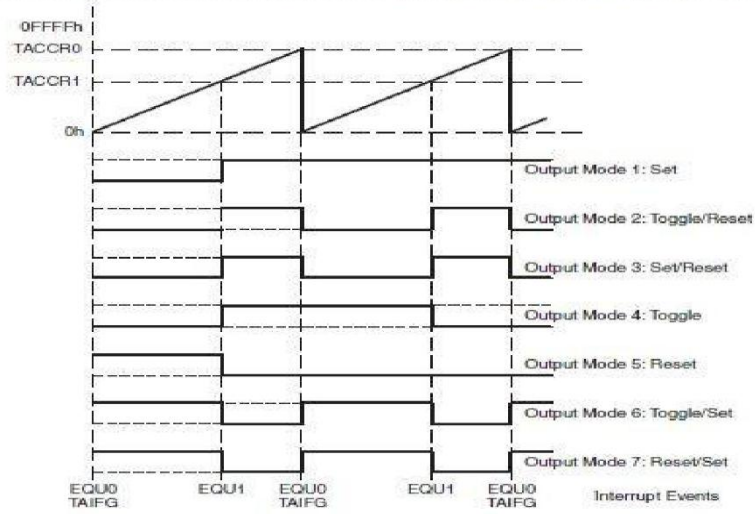


Figure 12-12. Output Example—Timer in Up Mode

Table 12-2. Output Modes

OUTMODx	Mode	Description
000	Output	The output signal OUTx is defined by the OUTx bit. The OUTx signal updates immediately when OUTx is updated.
001	Set	The output is set when the timer *counts* to the TACCRx value. It remains set until a reset of the timer, or until another output mode is selected and affects the output.
010	Toggle/Reset	The output is toggled when the timer *counts* to the TACCRx value. It is reset when the timer *counts* to the TACCR0 value.
011	Set/Reset	The output is set when the timer *counts* to the TACCRx value. It is reset when the timer *counts* to the TACCR0 value.
100	Toggle	The output is toggled when the timer *counts* to the TACCRx value. The output period is double the timer period.
101	Reset	The output is reset when the timer *counts* to the TACCRx value. It remains reset until another output mode is selected and affects the output.
110	Toggle/Set	The output is toggled when the timer *counts* to the TACCRx value. It is set when the timer *counts* to the TACCR0 value.
111	Reset/Set	The output is reset when the timer *counts* to the TACCRx value. It is set when the timer *counts* to the TACCR0 value.

Output Example—Timer in Continuous Mode

The OUTx signal is changed when the timer reaches the TACCRx and TACCR0 values, depending on the output mode. An example is shown in Figure 12-13 using TACCR0 and TACCR1.

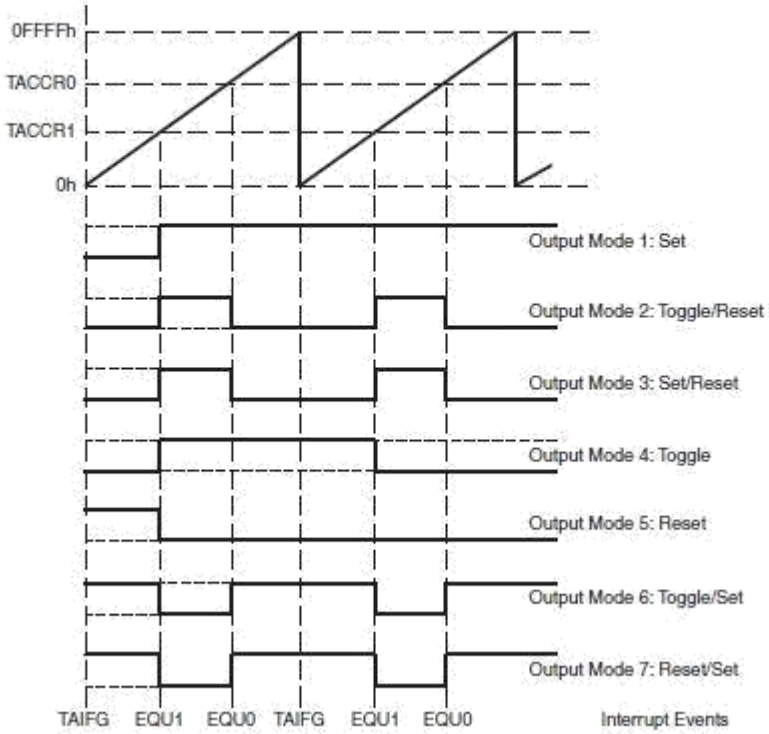


Figure 12-13. Output Example—Timer in Continuous Mode

Timer_A Interrupts

Two interrupt vectors are associated with the 16-bit Timer_A module:

TACCR0 interrupt vector for TACCR0 CCIFG

TAIV interrupt vector for all other CCIFG flags and TAIFG

In capture mode any CCIFG flag is set when a timer value is captured in the associated TACCRx register. In compare mode, any CCIFG flag is set if TAR counts to the associated TACCRx value. Software may also set or clear any CCIFG flag. All CCIFG flags request an interrupt when their corresponding CCIE bit and the GIE bit are set.

TACTL, Timer_A Control Register

15	14	13	12	11	10	9	8
Unused						TASSELx	
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)

7	6	5	4	3	2	1	0
IDx		MCx		Unused	TACLR	TAIE	TAIFG
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	w-(0)	rw-(0)	rw-(0)

Unused	Bits 15-10	Unused
TASSELx	Bits 9-8	Timer_A clock source select 00 TACLK 01 ACLK 10 SMCLK 11 Inverted TACLK
IDx	Bits 7-6	Input divider. These bits along with the IDEXx bits select the divider for the input clock. 00 /1 01 /2 10 /4 11 /8
MCx	Bits 5-4	Mode control. Setting MCx = 00h when Timer_A is not in use conserves power. 00 Stop mode: the timer is halted 01 Up mode: the timer counts up to TACCR0 10 Continuous mode: the timer counts up to 0FFFFh 11 Up/down mode: the timer counts up to TACCR0 then down to 0000h
Unused	Bit 3	Unused
TACLR	Bit 2	Timer_A clear. Setting this bit resets TAR, the TACLK divider, and the count direction. The TACLR bit is automatically reset and is always read as zero.
TAIE	Bit 1	Timer_A interrupt enable. This bit enables the TAIFG interrupt request. 0 Interrupt disabled 1 Interrupt enabled
TAIFG	Bit 0	Timer_A interrupt flag 0 No interrupt pending 1 Interrupt pending

TACCTLx, Capture/Compare Control Register

15	14	13	12	11	10	9	8
CMx		CCISx		SCS	SCCI	Unused	CAP
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	r-(0)	r-(0)	rw-(0)

7	6	5	4	3	2	1	0
OUTMODx			CCIE	CCI	OUT	COV	CCIFG
rw-(0)	rw-(0)	rw-(0)	rw-(0)	r	rw-(0)	rw-(0)	rw-(0)

CMx	Bit 15-14	Capture mode 00 No capture 01 Capture on rising edge 10 Capture on falling edge 11 Capture on both rising and falling edges
CCISx	Bit 13-12	Capture/compare input select. These bits select the TACCRx input signal. See the device-specific data sheet for specific signal connections. 00 CCIxA 01 CCIxB 10 GND 11 Vcc
SCS	Bit 11	Synchronize capture source. This bit is used to synchronize the capture input signal with the timer clock. 0 Asynchronous capture 1 Synchronous capture
SCCI	Bit 10	Synchronized capture/compare input. The selected CCI input signal is latched with the EQUx signal and can be read via this bit.
Unused	Bit 9	Unused. Read only. Always read as 0.
CAP	Bit 8	Capture mode 0 Compare mode 1 Capture mode
OUTMODx	Bits 7-5	Output mode. Modes 2, 3, 6, and 7 are not useful for TACCR0 because EQUx = EQU0. 000 OUT bit value 001 Set 010 Toggle/reset 011 Set/reset 100 Toggle 101 Reset 110 Toggle/set 111 Reset/set
CCIE	Bit 4	Capture/compare interrupt enable. This bit enables the interrupt request of the corresponding CCIFG flag. 0 Interrupt disabled 1 Interrupt enabled
CCI	Bit 3	Capture/compare input. The selected input signal can be read by this bit.
OUT	Bit 2	Output. For output mode 0, this bit directly controls the state of the output. 0 Output low 1 Output high
COV	Bit 1	Capture overflow. This bit indicates a capture overflow occurred. COV must be reset with software. 0 No capture overflow occurred 1 Capture overflow occurred

Timer B

Timer_B is a 16-bit timer/counter with three or seven capture/compare registers. Timer_B can support multiple capture/compares, PWM outputs, and interval timing. Timer_B also has extensive interrupt capabilities. Interrupts may be generated from the counter on overflow conditions and from each of the capture/compare registers.

Timer_B features include :

- Asynchronous 16-bit timer/counter with four operating modes and four selectable lengths Selectable and configurable clock source
- Up to seven configurable capture/compare registers
- Configurable outputs with PWM capability
- Double-buffered compare latches with synchronized loading
- Interrupt vector register for fast decoding of all Timer_B interrupts
- imer_B is identical to Timer_A with the following exceptions:
- The length of Timer_B is programmable to be 8, 10, 12, or 16 bits.
- Timer_B TBCCRx registers are double-buffered and can be grouped.
- All Timer_B outputs can be put into a high-impedance state.
- The SCCI bit function is not implemented in Timer_B.

Real Time Clock:

The Real-Time Clock module provides a clock with calendar that can also be configured as a generalpurpose counter.

Real-Time Clock features include:

- Configurable for Real-Time Clock mode or general purpose counterProvides seconds, minutes, hours, day of week, day of month, month and year in calender mode.
- Interrupt capability.
- Selectable BCD or binary format in Real-Time Clock

  mode Programmable alarms in Real-Time Clock mode
- Calibration logic for time offset correction in Real-Time clock mode

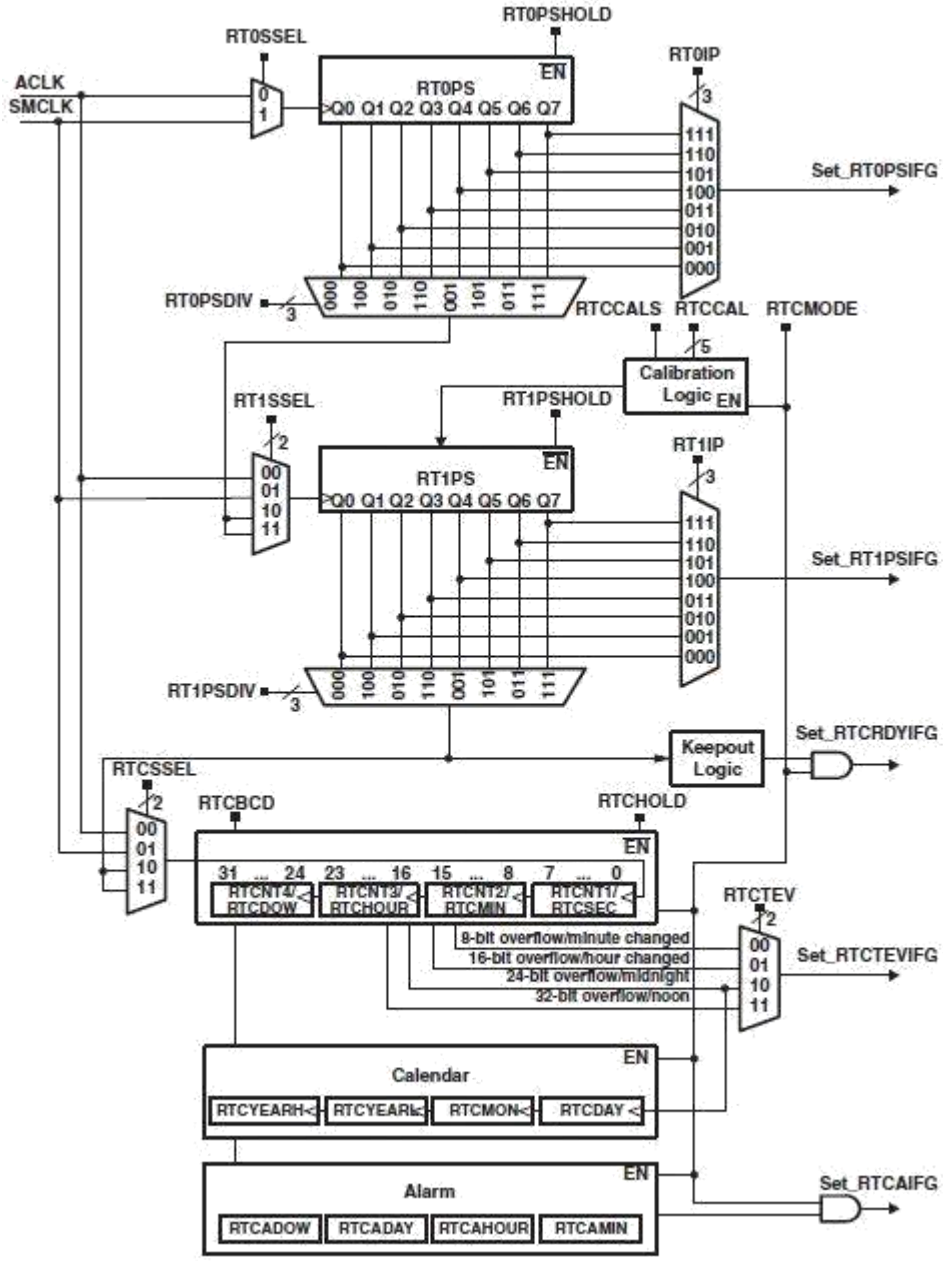


Figure 14-1. Real-Time Clock

The Real-Time Clock module can be configured as a real-time clock with calendar function or as a 32-bit general purpose counter with the RTCMODE bit.

Counter mode

Counter mode is selected when RTCMODE is reset. In this mode, a 32-bit counter is provided that isdirectly accessible by software. Switching from calendar mode to counter mode resets the count value (RTCNT1, RTCNT2, RTCNT3, RTCNT4), as well as, the prescale counters (RT0PS, RT1PS). The clock to increment the counter can be sourced from ACLK, SMCLK, or prescaled versions of ACLK or SMCLK. Prescaled versions of ACLK or SMCLK are sourced from the prescale dividers , RT0PS and RT1PS. RT0PS and RT1PS output /2, /4, /8, 16, /32,

/64, /128, /256 versions of ACLK and SMCLK, respectively. The output of RT0PS can be cascaded with RT1PS. The cascaded output can be used as a clock source input to the 32-bit counter.

Four individual 8-bit counters are cascaded to provide the 32-bit counter. This provides 8-bit, 16-bit, 24-bit, or 32-bit overflow intervals of the counter clock. The RTCTEV bits select the respective trigger event. An RTCTEV event can trigger an interrupt by setting the RTCTEVIE bit. Each counter RTCNT1 through RTCNT4 is individually accessible and may be written to. RT0PS and RT1PS can be configured as two 8-bit counters or cascaded into a single 16-bit counter. RT0PS and RT1PS can be halted on an individual basis by setting their respective RT0PSHOLD and RT1PSHOLD bits. When RT0PS is cascaded with RT1PS, setting RT0PSHOLD will cause both RT0PS and RT1PS to be halted. The 32-bit counter can be halted several ways depending on the configuration. If the 32-bit counter is sourced directly from ACLK or SMCLK, it can be halted by setting RTCHOLD. If it is sourced from the output of RT1PS, it can be halted by setting RT1PSHOLD or RTCHOLD. Finally, if it is sourced from the cascaded outputs of RT0PS and RT1PS, it can be halted by setting RT0PSHOLD, RT1PSHOLD, or RTCHOLD.

Calendar mode

Calendar mode is selected when RTCMODE is set. In calendar mode, the Real-Time Clock module provides seconds, minutes, hours, day of week, day of month, month, and year in selectable BCD or hexadecimal format. The calendar includes a leap year algorithm that considers all years evenly divisible by 4 as leap years. This algorithm is accurate from the year 1901 through 2099.

The prescale dividers, RT0PS and RT1PS are automatically configured to provide a one second clock interval for the Real-Time Clock. RT0PS is sourced from ACLK. ACLK must be set to 32768 Hz, nominal for proper Real-Time Clock calendar operation. RT1PS is cascaded with the output ACLK/256 of RT0PS. The Real-Time Clock is sourced with the /128 output of RT1PS, thereby providing the required one second interval. Switching from counter to calendar mode clears the seconds, minutes, hours, day-of-week, and year counts and sets day-of-month and month counts to 1. In addition, the RT0PS and RT1PS are cleared.

When RTCBCD = 1, BCD format is selected for the calendar registers. The format must be selected before the time is set. Changing the state of RTCBCD clears the seconds, minutes, hours, day-of-week, and year counts and sets day-of-month and month counts to 1. In addition, RT0PS and RT1PS are cleared.

In calendar mode, the RT0SSEL, RT1SSEL, RT0PSDIV, RT1PSDIV, RT0PSHOLD,

RT1PSHOLD, and RTCSSEL bits are do not care. Setting RTCHOLD halts the real-time counters and prescale counters, RT0PS and RT1PS.

Real-Time Clock and Prescale Dividers

The prescale dividers, RT0PS and RT1PS are automatically configured to provide a one second clock interval for the Real-Time Clock. RT0PS is sourced from ACLK. ACLK must be set to 32768 Hz, nominal for proper Real-Time Clock calendar operation. RT1PS is cascaded with the output ACLK/256 of RT0PS. The Real-Time Clock is sourced with the

/128 output of RT1PS, thereby providing the required one second interval. Switching from counter to calendar mode clears the seconds, minutes, hours, day-of-week, and year counts and sets day-of-month and month counts to 1. In addition, the RT0PS and RT1PS are cleared.

When RTCBCD = 1, BCD format is selected for the calendar registers. The format must be selected before the time is set. Changing the state of RTCBCD clears the seconds, minutes,

hours, day-of-week, and year counts and sets day-of-month and month counts to 1. In addition, RT0PS and RT1PS are cleared.

In calendar mode, the RT0SSEL, RT1SSEL, RT0PSDIV, RT1PSDIV, RT0PSHOLD,

RT1PSHOLD, and RTCSSEL bits are do not care. Setting RTCHOLD halts the real-time counters and prescale counters, RT0PS and RT1PS.

Real-Time Clock Alarm Function

The Real-Time Clock module provides for a flexible alarm system. There is a single, user programmable alarm that can be programmed based on the settings contained in the alarm registers for minutes, hours, day of week, and day of month. The user programmable alarm function is only available in calendar mode of operation.

Each alarm register contains an alarm enable bit, AE that can be used to enable the respective alarm register. By setting AE bits of the various alarm registers, a variety of alarm events can be generated. For example, a user wishes to set an alarm every hour at 15 minutes past the

hour i.e. 00:15:00, 01:15:00, 02:15:00, etc. This is possible by setting RTCAMIN to 15. By setting the AE bit of the RTCAMIN, and clearing all other AE bits of the alarm registers, the alarm will be enabled. When enabled, the AF will be set when the count transitions from 00:14:59 to 00:15:00, 01:14:59 to 01:15:00, 02:14:59 to 02:15:00,etc.

For example, a user wishes to set an alarm every day at 04:00:00. This is possible by setting RTCAHOUR to 4. By setting the AE bit of the RTCHOUR, and clearing all other AE bits of the alarm registers, the alarm will be enabled. When enabled, the AF will be set when the count transitions from 03:59:59 to 04:00:00.

For example, a user wishes to set an alarm for 06:30:00. RTCAHOUR would be set to 6 and RTCAMIN would be set to 30. By setting the AE bits of RTCAHOUR and RTCAMIN, the alarm will be enabled. Once enabled, the AF will be set when the the time count transitions from 06:29:59 to 06:30:00. In this case, the alarm event will occur every day at 06:30:00.

For example, a user wishes to set an alarm every Tuesday at 06:30:00. RTCADOW would be set to 2, RTCAHOUR would be set to 6 and RTCAMIN would be set to 30. By setting the AE bits of RTCADOW, RTCAHOUR and RTCAMIN, the alarm will be enabled. Once enabled, the AF will be set when the the time count transitions from 06:29:59 to 06:30:00 and the RTCDOW transitions from 1 to 2.

For example, a user wishes to set an alarm the fifth day of each month at 06:30:00. RTCADAY would be set to 5, RTCAHOUR would be set to 6 and RTCAMIN would be set to

30. By setting the AE bits of RTCADAY, RTCAHOUR and RTCAMIN, the alarm will be enabled. Once enabled, the AF will be set when the the time count transitions from 06:29:59 to 06:30:00 and the RTCDAY equals 5.

Reading or Writing Real-Time Clock Registers in Calendar Mode

must be used when accessing the Real-Time Clock registers.

In calendar mode, the real-time clock registers are updated once per second. In order to prevent reading any real-time clock register at the time of an update that could result in an invalid time being read, a keepout window is provided. The keepout window is centered approximately - 128/32768 seconds around the update transition. The read only RTCRDY bit is reset during the keepout window period and set outside the keepout the window period.
Any read of the clock registers while RTCRDY is reset, is considered to be potentially invalid, and the time read should be ignored.An easy way to safely read the real-time clock registers is

to utilize the RTCRDYIFG interrupt flag. Setting RTCRDYIE enables the RTCRDYIFG interrupt. Once enabled, an interrupt will be generated based on the rising edge of the RTCRDY bit, causing the RTCRDYIFG to be set. At this point, the application has nearly a complete second to safely read any or all of the real-time clock registers. This synchronization process prevents reading the time value during transition. The RTCRDYIFG flag is reset automatically

when the interrupt is serviced, or can be reset with software. In counter mode, the RTCRDY bit remains reset. The RTCRDYIE is a do not care and the RTCRDYIFG remains reset.

Real-Time Clock Interrupts

Real-Time Clock Interrupts in Calendar Mode

In calendar mode, five sources for interrupts are available, namely RT0PSIFG, RT1PSIFG, RTCRDYIFG, RTCTEVIFG, and RTCAIFG. These flags are prioritized and combined to source a single interrupt vector. The interrupt vector register RTCIV is used to determine which flag requested an interrupt. The highest priority enabled interrupt generates a number in the RTCIV register (see register description).

This number can be evaluated or added to the program counter to automatically enter the appropriate software routine. Disabled RTC interrupts do not affect the RTCIV value. Any access, read or write, of the RTCIV register automatically resets the highest pending interrupt flag. If another interrupt flag is set, another interrupt is immediately generated after servicing the initial interrupt. In addition, all flags can be cleared via software.

The user programmable alarm event sources the real-time clock interrupt, RTCAIFG. Setting the RTCAIE enables the interrupt. In addition to the user programmable alarm, The Real-Time Clock Module provides for an interval alarm that sources real-time clock interrupt, RTCTEVIFG. The interval alarm can be selected to cause an alarm event when RTCMIN changed, RTCHOUR changed, every day at midnight (00:00:00), or every day at noon (12:00:00). The event is selectable with the RTCTEV bits Setting the RTCTEVIE bit enables the interrupt.

The RTCRDY bit sources the real-time clock interrupt, RTCRDYIFG and is useful in synchronizing the read of time registers with the system clock. Setting the RTCRDYIE bit enables the interrupt. The RT0PSIFG can be used to generate interrupt intervals selectable by

the RT0IP bits. In calendar mode, RT0PS is sourced with ACLK at 32768 Hz, so intervals of 16384 Hz, 8192 Hz, 4096 Hz, 2048 Hz, 1024 Hz, 512 Hz, 256 Hz, or 128 Hz are possible. Setting the RT0PSIE bit enables the interrupt. The RT1PSIFG can be used to generate interrupt intervals selectable by the RT1IP bits. In calendar mode, RT1PS is sourced with the output of RT0PS, which is 128Hz (32768/256 Hz). Therefore, intervals of 64 Hz, 32 Hz, 16 Hz, 8 Hz, 4 Hz, 2 Hz, 1 Hz, or 0.5 Hz are possible. Setting the RT1PSIE bit enables the interrupt.

Real-Time Clock Interrupts in Counter Mode

In counter mode, a three interrupt sources are available, namely RT0PSIFG, RT1PSIFG, and RTCTEVIFG. The RTCAIFG and RTCRDYIFG are cleared. RTCRDYIE and RTCAIE are do not care.The RT0PSIFG can be used to generate interrupt intervals selectable by the RT0IP bits. In counter mode, RT0PS is sourced with ACLK or SMCLK so divide ratios of /2, /4, /8, /16, /32, /64, /128, /256 of the respective clock source are possible. Setting the RT0PSIE bit enables the interrupt.

The RT1PSIFG can be used to generate interrupt intervals selectable by the RT1IP bits. In counter mode, RT1PS is sourced with ACLK, SMCLK, or the output of RT0PS so divide ratios of /2, /4, /8, /16, /32, /64, /128, /256 of the respective clock source are possible. Setting the RT1PSIE bit enables the interrupt. The Real-Time Clock Module provides for an interval timer that sources real-time clock interrupt, RTCTEVIFG. The interval timer can be selected to cause an interrupt event when an 8-bit, 16-bit, 24-bit, or 32-bit overflow occurs within the 32-bit counter. The event is selectable with the RTCTEV bits Setting the RTCTEVIE bit enables the interrupt.

RTCCTL0, Real-Time Clock Control Register 0

7	6	5	4	3	2	1	0
Reserved	RTCTEVIE	RTCAIE	RTCRDYIE	Reserved	RTCTEVIFG	RTCAIFG	RTCRDYIFG
r0	rw-0	rw-0	rw-0	r0	rw-(0)	rw-(0)	rw-(0)

Reserved	Bit 7	Reserved. Always read as 0.
RTCTEVIE	Bit 6	Real-time clock time event interrupt enable
		0 Interrupt not enabled
		1 Interrupt enabled
RTCAIE	Bit 5	Real-time clock alarm interrupt enable. This bit remains cleared when in counter mode (RTCMODE = 0).
		0 Interrupt not enabled
		1 Interrupt enabled
RTCRDYIE	Bit 4	Real-time clock alarm interrupt enable
		0 Interrupt not enabled
		1 Interrupt enabled
Reserved	Bit 3	Reserved. Always read as 0.
RTCTEVIFG	Bit 2	Real-time clock time event flag
		0 No time event occurred.
		1 Time event occurred.
RTCAIFG	Bit 1	Real-time clock alarm flag. This bit remains cleared when in counter mode (RTCMODE = 0).
		0 No time event occurred.
		1 Time event occurred.
RTCRDYIFG	Bit 0	Real-time clock alarm flag
		0 RTC can not be read safely
		1 RTC can be read safely

Table 14-1. Real-Time Clock Registers

Register	Short Form	Register Type	Address Offset	Initial State
Real-Time Clock control register 0	RTCCTL0	Read/write	00h	00h
Real-Time Clock control register 1	RTCCTL1	Read/write	01h	40h
Real-Time Clock control register 2	RTCCTL2	Read/write	02h	00h
Real-Time Clock control register 3	RTCCTL3	Read/write	03h	00h
Real-Time Prescale Timer 0 control register	RTCPS0CTL	Read/write	08h	10h
Real-Time Prescale Timer 1 control register	RTCPS1CTL	Read/write	0Ah	10h
Real-Time Prescale Timer 0	RTCPS0	Read/write	0Ch	Unchanged
Real-Time Prescale Timer 1	RTCPS1	Read/write	0Dh	Unchanged
Real Time Clock Interrupt vector	RTCIV	Read	0Eh	00h
Real-Time Clock Second Real-Time Counter register 1	RTCSEC/RTCNT1	Read/write	10h	Unchanged
Real-Time Clock Minute Real-Time Counter register 2	RTCMIN/RTCNT2	Read/write	11h	Unchanged
Real-Time Clock Hour Real-Time Counter register 3	RTCHOUR/RTCNT3	Read/write	12h	Unchanged
Real-Time Clock Day of Week Real-Time Counter register 4	RTCDOW/RTCNT4	Read/write	13h	Unchanged
Real-Time Clock Day of Month	RTCDAY	Read/write	14h	Unchanged
Real-Time Clock Month	RTCMON	Read/write	15h	Unchanged
Real-Time Clock Year (Low Byte)	RTCYEARL	Read/write	16h	Unchanged
Real-Time Clock Year (High Byte)	RTCYEARH	Read/write	17h	Unchanged
Real-Time Clock Minute Alarm	RTCAMIN	Read/write	18h	Unchanged
Real-Time Clock Hour Alarm	RTCAHOUR	Read/write	19h	Unchanged
Real-Time Clock Day of Week Alarm	RTCADOW	Read/write	1Ah	Unchanged
Real-Time Clock Day of Month Alarm	RTCADAY	Read/write	1Bh	Unchanged

RTCCTL1, Real-Time Clock Control Register 1

7	6	5	4	3	2	1	0
RTCBCD	RTCHOLD	RTCMODE	RTCRDY	RTCSSEL		RTCTEV	
rw-(0)	rw-(1)	rw-(0)	r-(0)	rw-0	rw-0	rw-(0)	rw-(0)

RTCBCD Bit 7 Real-time clock BCD select. Selects BCD counting for real-time clock. Applies to calendar mode (RTCMODE = 1) only - setting will be ignored in counter mode. Changing this bit will clear seconds, minutes, hours, day of week, and year are to 0 and sets day of month and month to 1. The real-time clock registers need to be set by software afterwards.

0 Binary/hexadecimal code selected

1 BCD (Binary Coded Decimal) code selected

RTCHOLD Bit 6 Real-time clock hold

0 Real-Time Clock (32-bit counter or calendar mode) is operational

1 In counter mode (RTCMODE = 0) only the 32-bit counter is stopped. In calendar mode (RTCMODE = 1) the calendar is stopped as well as the Prescale counters, RT0PS and RT1PS. RT0PSHOLD and RT1PSHOLD are do not care.

RTCMODE Bit 5 Real-time clock mode

0 32-bit counter mode

1 Calendar modeSwitching between counter and calendar mode will reset the real-time clock/counter registers. Switching to calendar mode clears seconds, minutes, hours, day of week, and year are to 0 and sets day of month and month to 1. The real-time clock registers need to be set by software afterwards. The Basic Timer counters, BT0CNT and BT1CNT, are also cleared.

RTCRDY Bit 4 Real-time clock ready

0 RTC time values in transition (calendar mode only).

1 RTC time values safe for reading (calendar mode only)This bit indicates when the RTC time values are safe for reading (calendar mode only). In counter mode, RTCRDY signal remains cleared.

RTCSSEL Bits 3-2 Real-time clock source select. Selects clock input source to the RTC/32-bit counter. In Real-Time Clock calendar mode, these bits are do not care. The clock input is automatically set to the output of RT1PS.

00 ACLK

01 SMCLK

10 Output from RT1PS

11 Output from RT1PS

RTCTEV Bits 1-0 Real-time clock time event

RTC Mode	RTCTEVx	Interrupt Interval
Counter Mode (RTCMODE = 0)	00	8-bit overflow
	01	16-bit overflow
	10	24-bit overflow
	11	32-bit overflow
Calendar Mode (RTCMODE = 1)	00	Minute changed
	01	Hour changed
	10	Every day at midnight (00:00)
	11	Every day at noon (12:00)

DMA Controller

The direct memory access (DMA) controller module transfers data from one address to another, without CPU intervention. This chapter describes the operation of the DMA controller.

Direct Memory Access (DMA) Introduction

The DMA controller transfers data from one address to another, without CPU intervention, across the entire address range. For example, the DMA controller can move data from the ADC conversion memory to RAM. Devices that contain a DMA controller may have up to eight DMA channels available. Therefore, depending on the number of DMA channels available, some features described in this chapter are not applicable to all devices. See the device-specific data sheet for number of channels supported. Using the DMA controller can increase the throughput of peripheral modules. It can also reduce system power consumption by allowing the CPU to remain in a low-power mode, without having to awaken to move data to or from a peripheral.

DMA controller features include:

- Up to eight independent transfer channels
- Configurable DMA channel priorities
- Requires only two MCLK clock cycles per transfer
- Byte or word and mixed byte/word transfer capability
- Block sizes up to 65535 bytes or words
- Configurable transfer trigger selections
- Selectable-edge or level-triggered transfer
- Four addressing modes
- Single, block, or burst-block transfer modes

OSCILLATOR CIRCUIT

The digital computer systems must contain an oscillator circuit for its functioning. The oscillator circuit is often regarded as the 'heartbeat' of an embedded system. Any malfunctioning of the oscillator will directly have an impact in the normal functioning of the system. The system depends upon the oscillator for all time-related calculations.

Generally an **oscillator** in an **embedded system** refers to the hardware (either internal to the microcontroller, or external to it) that creates the **system** clock, usually from a crystal that

determines the base frequency of the **oscillator**. ... One example is an **oscillator** that drives an RTC (Real Time Clock).

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.The oscillator unit of the embedded system is responsible for generating the precise clock for the processor.

Certain processor/controller chips may not contain a built-in oscillator unit and require the clock pulses to be generated and supplied externally.

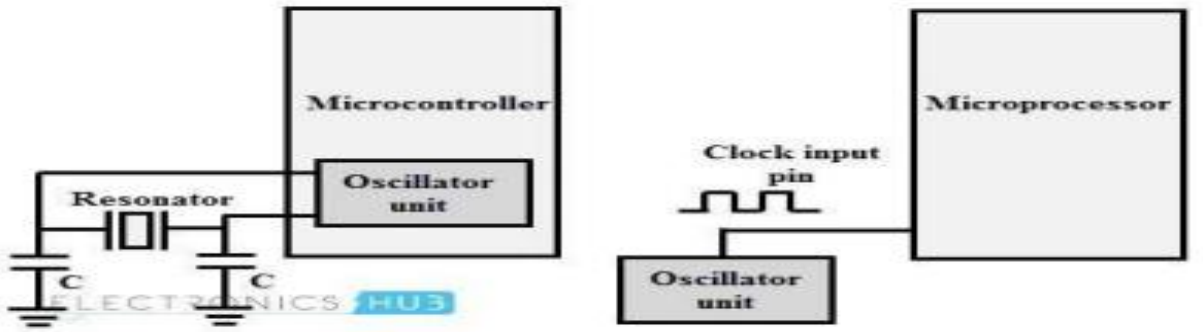


Fig: Oscillator circuit

Quartz crystal Oscillators are example for clock pulse generating devices. The total system power consumption is directly proportional to the clock frequency. The power consumption increase with the increase in the clock frequency .The accuracy of the program execution depends on the accuracy of the clock signal.

An **oscillator** is a device that **can** give you a clock at a certain frequency given a constant voltage. One part of the **oscillator** is a crystal that is basically a very good filter. Most **microcontrollers** only require a crystal because they contain all other parts of an **oscillator** already.

Why a low oscillator frequency microcontroller is preferred?

- Many people might think that selecting a microcontroller of higher frequency delivers maximum performance. This can be a mistake, because of the following reasons -:
- Using applications which don't require the level of performance that modern 8051 device offers.
- In modern 8051s, the oscillator frequency and power supply current vary linearly with each other. As a result, by choosing a device with the low frequency the power requirements can be brought down.

- The electromagnetic interference generated by a circuit can be reduced by using a device with low clock frequency.
- While accessing low-speed peripherals, the cost of peripheral components can be reduced if the chip is operating at low frequency. Also, the design and programming become simplified.

WATCH DOG TIMER

A **watchdog timer** (sometimes called a *computer operating properly* or *COP* timer, or simply a *watchdog*) is an electronic timer that is used to detect and recover from computer malfunctions. During normal operation, the computer regularly resets the watchdog timer to prevent it from elapsing, or "timing out". If, due to a hardware fault or program error, the computer fails to reset the watchdog, the timer will elapse and generate a timeout signal. The timeout signal is used to initiate corrective action or actions. The corrective actions typically include placing the computer system in a safe state and restoring normal system operation.

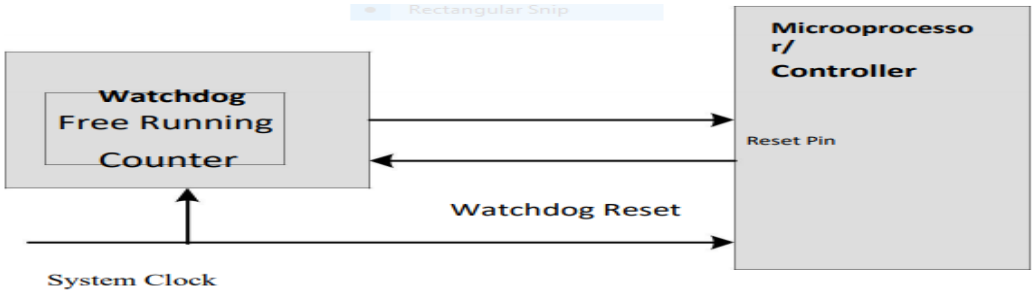
Watchdog timers are commonly found in embedded systems and other computer-controlled equipment where humans cannot easily access the equipment or would be unable to react to faults in a timely manner. In such systems, the computer cannot depend on a human to invoke a reboot if it hangs; it must be self-reliant. For example, remote embedded systems such as space probes are not physically accessible to human operators; these could become permanently disabled if they were unable to autonomously recover from faults. A watchdog timer is usually employed in cases like these. Watchdog timers may also be used when running untrusted code in a sandbox, to limit the CPU time available to the code and thus prevent some types of denial-of-service attacks.

Function of WATCH DOG TIMER:

- A timer unit 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 up counting watchdog.
- If the watchdog counter is in the enabled state, the firmware can write a zero (for up counting 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 the firmware execution completes before the expiration of the watchdog timer the WDT can be stopped from action.
- 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.



Embedded firmware Development Languages/Options :

1.Assembly LanguagE

2.High Level Language

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

3.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

Embedded firmware Development Languages/Options – Assembly Language

- 'Assembly Language' is the human readable notation of 'machine language'
- 'Machine language' is a processor understandable language
- Machine language is a binary representation and it consists of 1s and 0s

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

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.

UNIT IV

RTOS BASED EMBEDDED SYSTEM DESIGN

Introduction

Most embedded systems are bound to real-time constraints. In production control the various machines have to receive their orders at the right time to ensure smooth operation of a plant and to fulfill customer orders in time. Railway switching systems obviously have to act in a timely manner.An embedded system with a single CPU can run only one process at an instance the process at any instance either be an ISR

Principles: semaphore and queues:

The principles will discuss the design considerations that have application to a broad range of embedded system.

- Write Short interrupt routines
- Limit the number of tasks
- Avoid creating and destroying tasks
- Avoid time-slicing
- Encapsulate semaphores in separate functions
- Encapsulate queues in separate functions

Encapsulating Queues:
consider encapsulating queues that tasks use to receive messages from other tasks. we wrote code to handle a shared flash memory. That code deals correctly with synchronizing the requests for reading from and writing to the flash memory. Since any task can write onto the flash memory

task input queue, any programmer can blow it and send a message that does not contain a FLASH_MSG structure.

Hard Real-Time Scheduling Considerations:

Hard Real Time Systems: Systems were time constraints are absolutely critical

Soft Real Time Systems: Systems with time constraints where minor errors are tolerated

The obvious issue that arises in hard real-time systems is that you must somehow guarantee that the system will meet the hard deadlines. The ability to meet hard deadlines comes from writing fast code To write some frequently called subroutine in assembly language.

If you can characterize your tasks, then the studies can help you determine if your system will meet its deadlines,

- Saving Memory and Power
- Saving memory

Embedded systems often have limited memory

RTOS: each task needs memory space for its stack.

- The first method for determining how much stack space a task needs is to examine your code
- The second method is experimental. Fill each stack withsome recognizable data pattern at startup, run the system for a period of time

Program Memory:

Limit the number of functions used. Check the automatic inclusions by your linker: may consider writing own functions. Include only needed functions in RTOS Consider using assembly language for large routines.

Data Memory:

Consider using more static variables instead of stack variables

Few ways to save code space:

- Make sure that you are not using two functions to do the same thing.
- Check that your development tools are not sabotaging you.

- Configure your RTOS to contain only those functions that you need.
- Look at the assembly language listings created by your cross-compiler to see if certain of your C statements translate into huge numbers of instructions.

Saving power:

The primary method for preserving battery power is to turn off parts or all of the system whenever possible.Most embedded-system microprocessors have at least one power-saving mode; many have several.The modes have names such as sleep mode, low-power mode, idle mode, standby mode, and so on.A very common power-saving mode is one in which the microprocessor stops executing instructions, stops any built-in peripherals, and stops its clock circuit. This saves a lot of power, but the drawback typically is that the only way to start the microprocessor up again is to reset it.Static RAM uses very little power when the microprocessor isn't executing instructions.Another typical power-saving mode is one in which the microprocessor stops executing instructions but the on-board peripherals continue to operate.Another common method for saving power is to turn off the entire system and have the user turn it back on when it is needed.

OPERATING SYSTEMS:

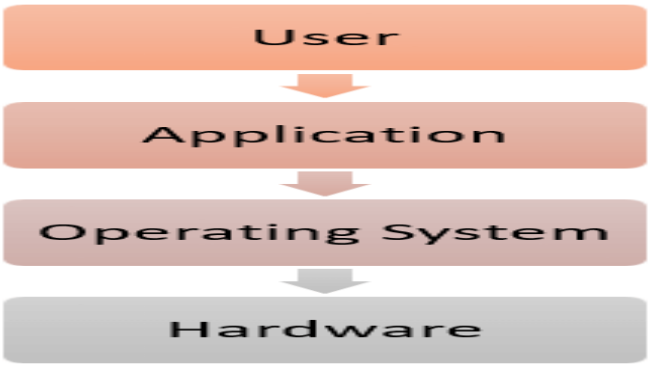
An operating system is a software which acts as an interface between the end user and computer hardware. Every computer must have at least one OS to run other programs. An application like Chrome, MS Word, Games, etc needs some environment in which it will run and perform its task. The OS helps you to communicate with the computer without knowing how to speak the computer's language. It is **not** possible for the user to use any computer or mobile device without having an operating system.

Features of Operating System

Here is a list commonly found important features of an Operating System:

- Protected and supervisor mode
- Allows disk access and file systems Device drivers Networking Security
- Program Execution
- Memory management Virtual Memory Multitasking
- Handling I/O operations
- Manipulation of the file system

- Error Detection and handling
- Resource allocation
- Information and Resource Protection



KERNEL

The kernel is the central component of a computer operating systems. The only job performed by the kernel is to the manage the communication between the software and the hardware. A Kernel is at the nucleus of a computer. It makes the communication between the hardware and software possible. While the Kernel is the innermost part of an operating system, a shell is the outermost one.

Types of Operating system

- Batch Operating System
- Multitasking/Time Sharing OS
- Multiprocessing OS
- Real Time OS
- Distributed OS
- Network OS
- Mobile OS

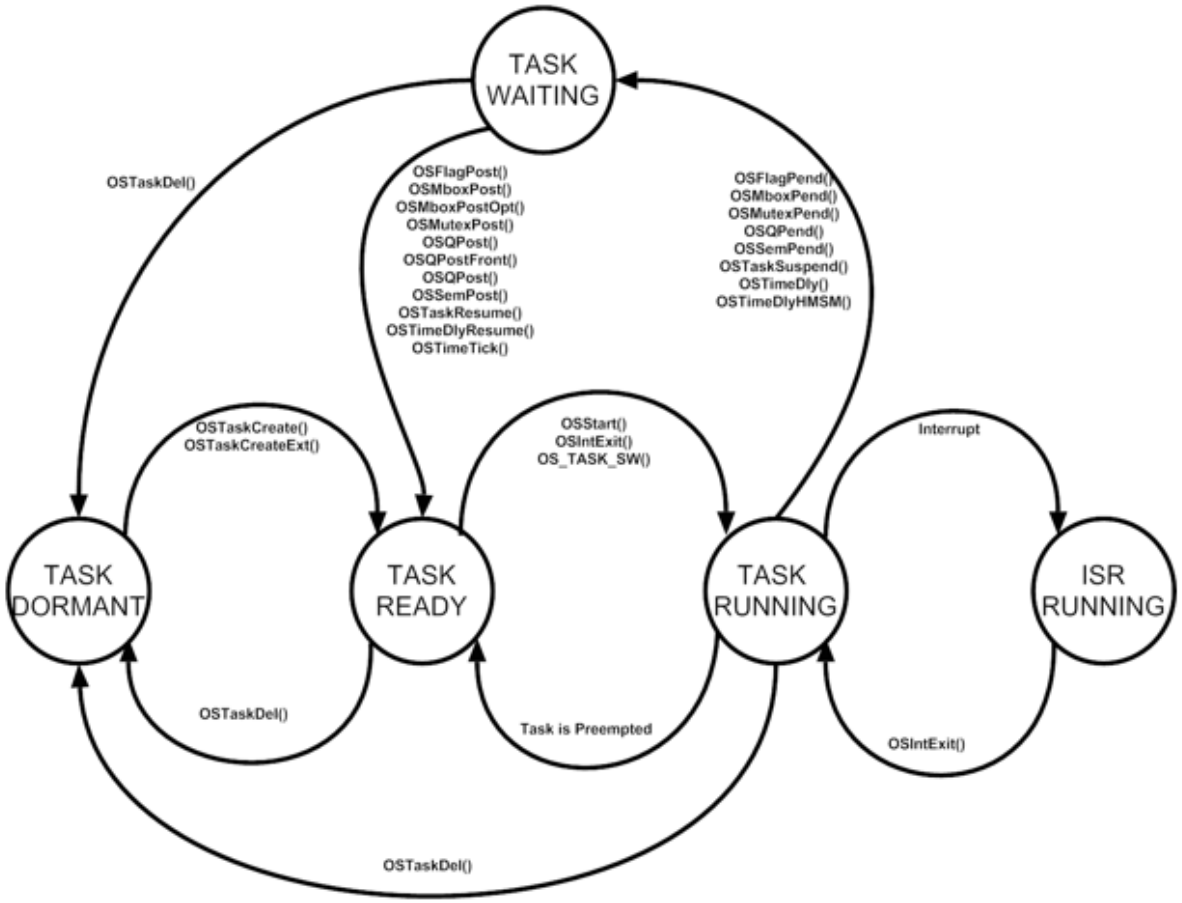
TASKS

A **task** is also called a thread, is a simple program that thinks it has a CPU all to itself. A **task** consists of a sequentially executable program under a state-control by OS. ... Thus a **task** can be defined as an **embedded** program computational unit that runs on a CPU under the state-control of kernel of an OS

1. A task is also called a thread, is a simple program that thinks it has a CPU all to itself.
2. A task consists of a sequentially executable program under a state-control by OS.

3. The state information of a task is represented by task state, task structure – its data, objects and resource and task control block.
4. The design for real time application involves splitting the work into tasks responsible for a portion of problem.
5. Each task has its own priority, its own set of CPU register and its own stack area.
6. Thus a task can be defined as an embedded program computational unit that runs on a CPU under the state-control of kernel of an OS. It has a state which at instance defines its status, structure, data, objects and resources and control block.

Task States:



Each task typically is an infinite loop that can be in any one of five states: dormant, ready, running, waiting (for an event), or ISR (interrupted)

1. **Dormant**: The task has been created and a memory is allotted to its structure but the kernel has not scheduled this task.

2. **Ready**: The created task is ready and has been scheduled by the kernel. This task however is not running because a higher priority task is under execution by the CPU.
3. **Running**: The task has access to all the shared resources required for its execution. The CPU executes the task in this state until an higher priority task requests the CPU for its execution
4. **Waiting:** The task is said to be blocked. In this case the task is waiting for a resource which could be some data being processed by some other task or an external real- world input.
5. **Interrupted (ISR):** Finally, a task is in the ISR state when an interrupt has occurred and the CPU is in the process of servicing the interrupt. Figure also shows the functions provided by μC/OS-II to make a task move from one state to another.

PROCESS AND THREADS:

The **process** is an execution of a program whereas **thread** is an execution of a program driven by the environment of a **process**. Another major point which differentiates **process and thread** is that **processes** are isolated with each other whereas **threads** share memory or resources with each other.

Process vs Thread – Difference between Process and Thread

Process	Thread
1) System calls involved in process.	1) No system calls involved.
2) Context switching required.	2) No context switching required.
3) Different process have different copies of code and data.	3) Sharing same copy of code and data can be possible among different threads..
4) Operating system treats different process differently.	4) All user level threads treated as single task for operating system.
5) If a process got blocked, remaining process continue their work.	5) If a user level thread got blocked, all other threads get blocked since they are treated as single task to OS. (Noted: This is

	can be avoided in kernel level threads).
6) Processes are independent.	6) Threads exist as subsets of a process. They are dependent.
7) Process run in separate memory space.	7) Threads run in shared memory space. And use memory of process which it belong to.
8) Processes have their own program counter (PC), register set, and stack space.	8) Threads share Code section, data section, Address space with other threads.
9) Communication between processes requires some time.	9) Communication between processes requires less time than processes.
10) Processes don't share the memory with any other process.	10) Threads share the memory with other threads of the same process
11) Process have overhead.	11) Threads have no overhead.

MULTI PROCESSING AND MULTI TASKING:

Multiprogramming – A computer running more than one program at a time (like running Excel and Firefox simultaneously). **Multiprocessing** – A computer using more than one CPU at a time. **Multitasking** – Tasks sharing a common resource (like 1 CPU). Multithreading is an extension of **multitasking**.

Multiprocessing –

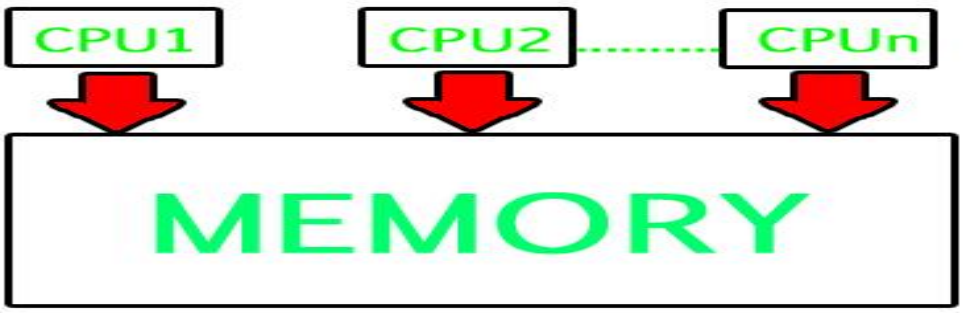
In a uni-processor system, only one process executes at a time.Multiprocessing is the use of two or more CPUs (processors) within a single Computer system. The term also refers to the ability of a system to support more than one processor within a single computer system. Now since there are multiple processors available, multiple processes can be executed at a time. These multi processors share the computer bus, sometimes the clock, memory and peripheral devices also.

Multi processing system's working –

- With the help of multiprocessing, many processes can be executed simultaneously. Say processes P1, P2, P3 and P4 are waiting for execution. Now in a single processor system, firstly one process will execute, then the other, then the other and so on.
- But with multiprocessing, each process can be assigned to a different processor for its execution. If its a dual-core processor (2 processors), two processes can be executed simultaneously and thus will be two times faster, similarly a quad core processor will be four times as fast as a single processor.

Why use multi processing –

- The main advantage of multiprocessor system is to get more work done in a shorter period of time. These types of systems are used when very high speed is required to process a large volume of data. Multi processing systems can save money in comparison to single processor systems because the processors can share peripherals and power supplies.
- It also provides increased reliability in the sense that if one processor fails, the work does not halt, it only slows down. e.g. if we have 10 processors and 1 fails, then the work does not halt, rather the remaining 9 processors can share the work of the 10th processor. Thus the whole system runs only 10 percent slower, rather than failing altogether.



Multiprocessing refers to the hardware (i.e., the CPU units) rather than the software (i.e., running processes). If the underlying hardware provides more than one processor then that is multiprocessing. It is the ability of the system to leverage multiple processors' computing power.

Multitasking –

As the name itself suggests, multi tasking refers to execution of multiple tasks (say processes, programs, threads etc.) at a time. In the modern operating systems, we are able to play MP3 music,

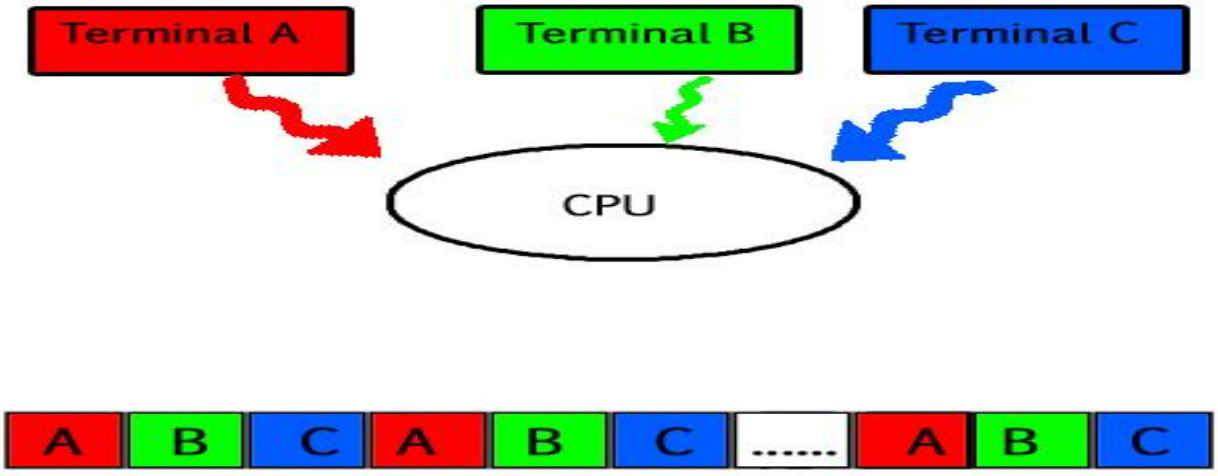
edit documents in Microsoft Word, surf the Google Chrome all simultaneously, this is accomplished by means of multi tasking.

Multitasking is a logical extension of multi programming. The major way in which multitasking differs from multi programming is that multi programming works solely on the concept of context switching whereas multitasking is based on time sharing alongside the concept of context switching.

Multi tasking system's working –

- In a time sharing system, each process is assigned some specific quantum of time for which a process is meant to execute. Say there are 4 processes P1, P2, P3, P4 ready to execute. So each of them are assigned some time quantum for which they will execute e.g time quantum of 5 nanoseconds (5 ns). As one process begins execution (say P2), it executes for that quantum of time (5 ns). After 5 ns the CPU starts the execution of the other process (say P3) for the specified quantum of time.
- Thus the CPU makes the processes to share time slices between them and execute accordingly. As soon as time quantum of one process expires, another process begins its execution.
- Here also basically a context switch is occurring but it is occurring so fast that the user is able to interact with each program separately while it is running. This way, the user is given the illusion that multiple processes/ tasks are executing simultaneously. But actually only one process/ task is executing at a particular instant of time. In multitasking, time sharing is best manifested because each running process takes only a fair quantum of the CPU time.

In a more general sense, multitasking refers to having multiple programs, processes, tasks, threads running at the same time. This term is used in modern operating systems when multiple tasks share a common processing resource (e.g., CPU and Memory).



- As depicted in the above image, At any time the CPU is executing only one task while other tasks are waiting for their turn. The illusion of parallelism is achieved when the CPU is reassigned to another task. i.e all the three tasks A, B and C are appearing to occur simultaneously because of time sharing.
- So for multitasking to take place, firstly there should be multiprogramming i.e. presence of multiple programs ready for execution. And secondly the concept of time sharing.

TASK SCHEDULING

The **Task Scheduler** is a tool included with Windows that allows predefined actions to be automatically executed whenever a certain set of conditions is met. For example, you can schedule a **task** to run a backup script every night, or send you an e-mail whenever a certain system event occurs

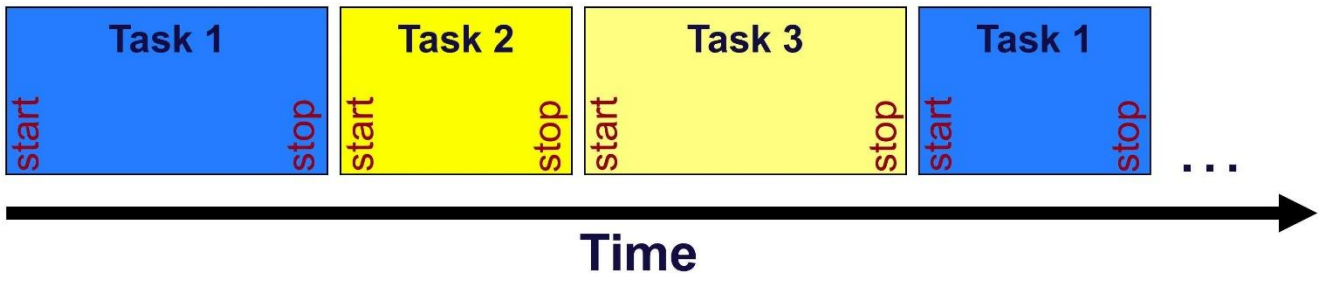
Schedulers

As we know, the illusion that all the tasks are running concurrently is achieved by allowing each to have a share of the processor time. This is the core functionality of a kernel. The way that time is allocated between tasks is termed “scheduling”. The scheduler is the software that determines which task should be run next. The logic of the scheduler and the mechanism that determines when it should be run is the scheduling algorithm. We will look at a number of scheduling algorithms in this section. Task scheduling is actually a vast subject, with many whole books devoted to it. The intention here is

to just give sufficient introduction that you can understand what a given RTOS has to offer in this respect.

Run to Completion (RTC) Scheduler

RTC scheduling is very simplistic and uses minimal resources. It is, therefore, an ideal choice, if the application's needs are fulfilled. Here is the timeline for a system using RTC scheduling:

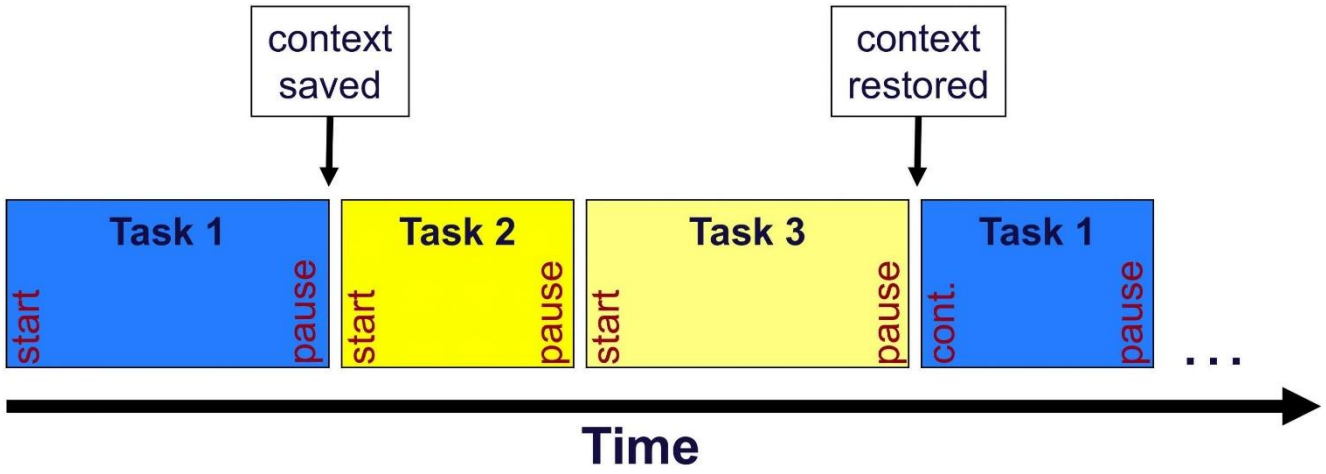


The scheduler simply calls the top level function of each task in turn. That task has control of the CPU (interrupts aside) until the top level function executes a **return** statement. If the RTOS supports task suspension, then any tasks that are currently suspended are not run. This is a topic discussed below; see *Task Suspend* .

The big advantages of an RTC scheduler, aside from its simplicity, are the need for just a single stack and the portability of the code (as no assembly language is generally required). The downside is that a task can "hog" the CPU, so careful program design is required. Although each task is started "from the top" each time it is scheduled – unlike other kinds of schedulers which allow the code to continue from where it left off – greater flexibility may be programmed by use of static "state" variables, which determine the logic of each sequential call.

Round Robin (RR) Scheduler

An RR scheduler is similar to RTC, but more flexible and, hence, more complex. In the same way, each task is run in turn (allowing for task suspension), thus:

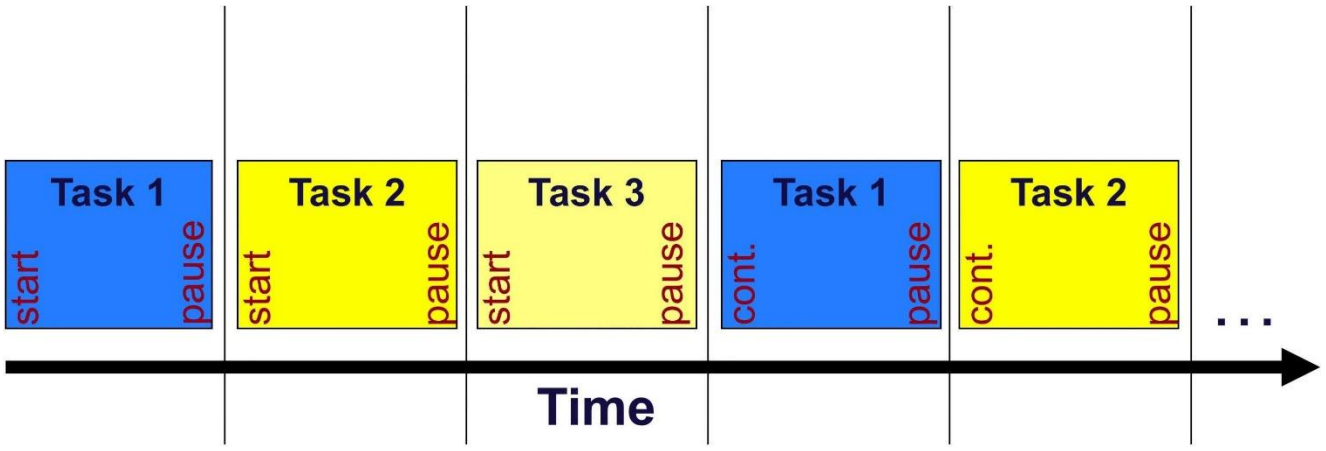


However, with the RR scheduler, the task does not need to execute a **return** in the top level function. It can relinquish the CPU at any time by making a call to the RTOS. This call results in the kernel saving the context (all the registers – including stack pointer and program counter) and loading the context of the next task to be run. With some RTOSes, the processor may be relinquished – and the task suspended – pending the availability of a kernel resource. This is more sophisticated, but the principle is the same.

The greater flexibility of the RR scheduler comes from the ability for the tasks to continue from where they left off without any accommodation in the application code. The price for this flexibility is more complex, less portable code and the need for a separate stack for each task.

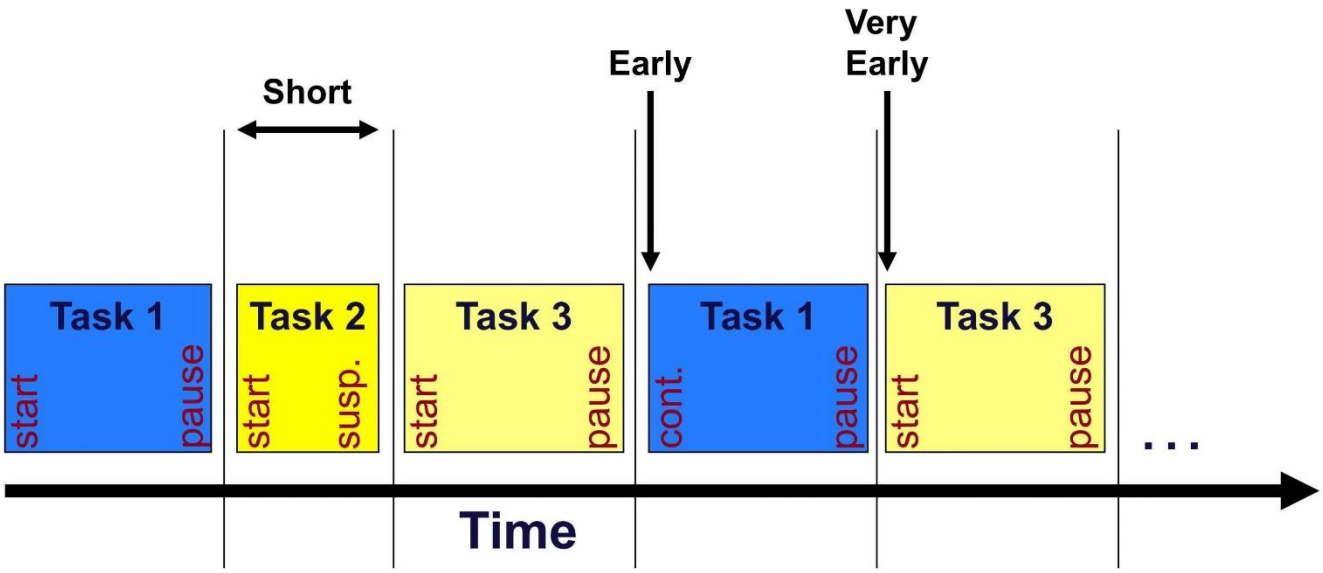
Time Slice (TS) Scheduler

A TS scheduler is the next step in complexity from RR. Time is divided into "slots", with each task being allowed to execute for the duration of its slot, thus:

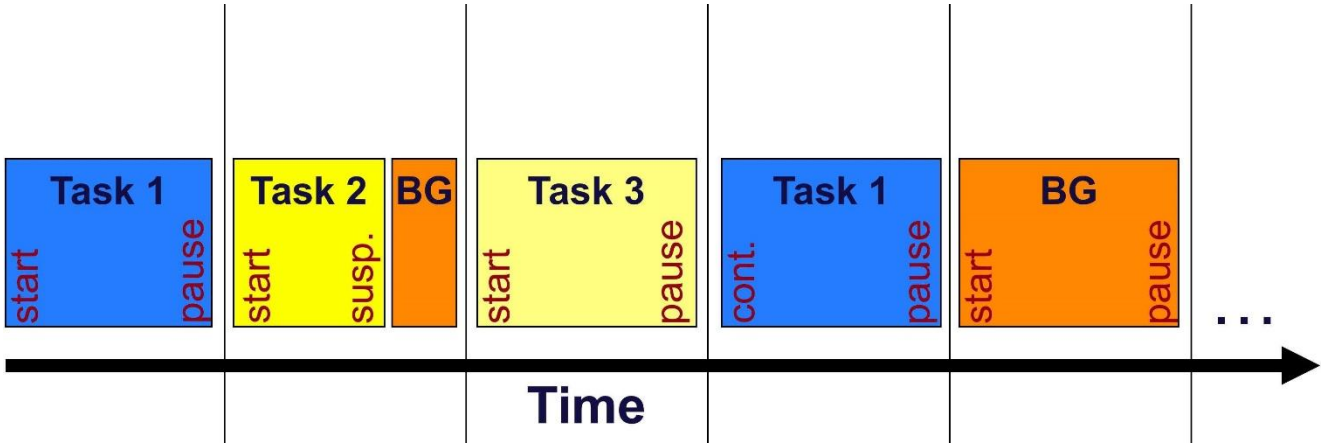


In addition to being able to relinquish the CPU voluntarily, a task is preempted by a scheduler call made from a clock tick interrupt service routine. The idea of simply allocating each task a fixed time slice is very appealing – for applications where it fits the requirements – as it is easy to understand and very predictable.

The only downside of simple TS scheduling is the proportion of CPU time allocated to each task varies, depending upon whether other tasks are suspended or relinquish part of their slots, thus:



A more predictable TS scheduler can be constructed if the concept of a "background" task is introduced. The idea, shown here, is for the background task to be run instead of any suspended tasks and to be allocated the remaining slot time when a task relinquishes (or suspends itself).



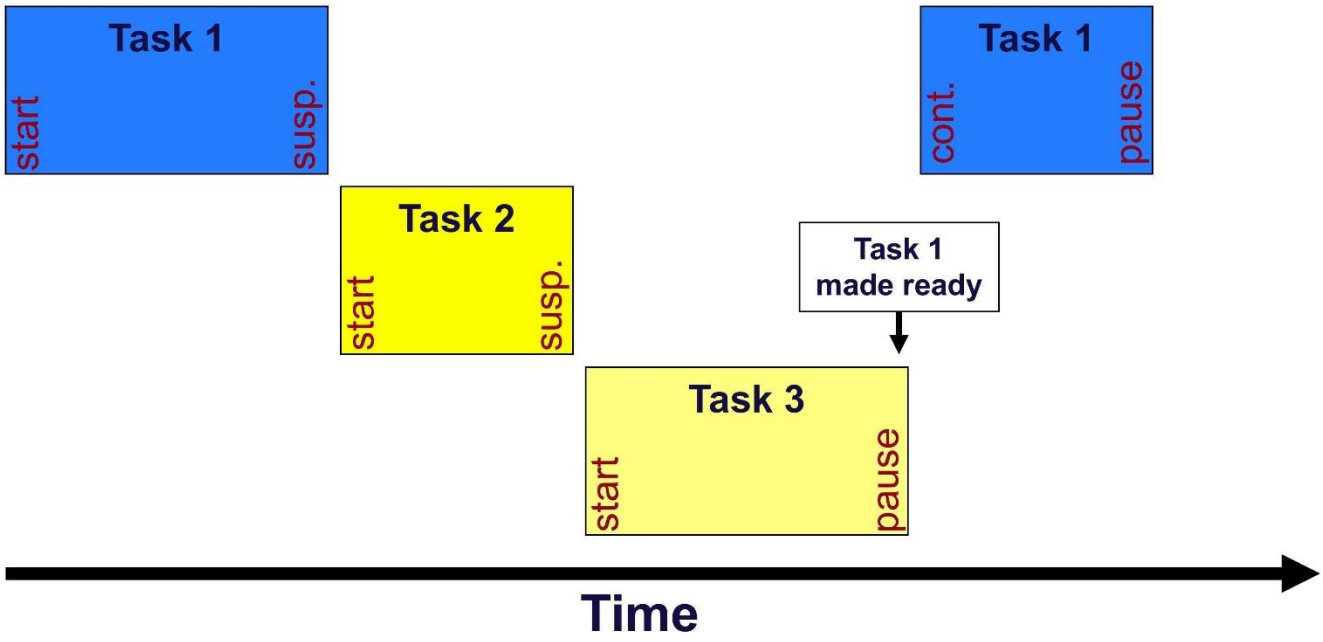
Obviously the background task should not do any time-critical work, as the amount of CPU time it is allocated is totally unpredictable – it may never be scheduled at all.

This design means that each task can predict when it will be scheduled again. For example, if you have 10ms slots and 10 tasks, a task knows that, if it relinquishes, it will continue executing after 100ms. This can lead to elegant timing loops in application tasks.

An RTOS may offer the possibility for different time slots for each task. This offers greater flexibility, but is just as predictable as with fixed slot size. Another possibility is to allocate more than one slot to the same task, if you want to increase its proportion of allocated processor time.

Priority Scheduler

Most RTOSes support Priority scheduling. The idea is simple: each task is allocated a priority and, at any particular time, whichever task has the highest priority and is "ready" is allocated the CPU, thus:



The scheduler is run when any "event" occurs (e.g. interrupt or certain kernel service calls) that may cause a higher priority task being made "ready". There are broadly three circumstances that might result in the scheduler being run:

- The task suspends itself; clearly the scheduler is required to determine which task to run next.
- The task readies another task (by means of an API call) of higher priority.

- An interrupt service routine (ISR) readies another task of higher priority. This could be an input/output device ISR or it may be the result of the expiration of a timer (which are supported my many RTOSes – we will look at them in detail in a future article).

The number of levels of priority varies (from 8 to many hundreds) and the significance of higher and lower values differs; some RTOSes use priority 0 as highest, others as lowest.

Some RTOSes only allow a single task at each priority level; others permit multiple tasks at each level, which complicates the associated data structures considerably. Many OSes allow task priorities to be changed at runtime, which adds further complexity.

Composite Scheduler

We have looked at RTC, RR, TS and Priority schedulers, but many commercial RTOS products offer more sophisticated schedulers, which have characteristics of more than one of these algorithms. For example, an RTOS may support multiple tasks at each priority level and then use time slicing to divide time between multiple ready tasks at the highest level.

Task States

At any one moment in time, just one task is actually running. Aside from CPU time spent running interrupt service routines (more on that in the next article) or the scheduler, the "current" task is the one whose code is currently being executed and whose data is characterized by the current register values. There may be other tasks that are "ready" (to run) and these will be considered when the scheduler is executed. In a simple RTOS, using a Run to Completion, Round Robin or Time Slice scheduler, this may be the whole story. But, more commonly, and always with a Priority scheduler, tasks may also be in a "suspended" state, which means that they are not considered by the scheduler until they are resumed and made "ready".

Task Suspend

Task suspension may be quite simple – a task suspends itself (by making an API call) or another task suspends it. Another API call needs to be made by another task or ISR to resume the suspended task. This is an "unconditional" or "pure" suspend. Some OSes refer to a task as being "asleep".

An RTOS may offer the facility for a task to suspend itself (go to sleep) for a specific period of time, at the end of which it is resumed (by the system clock ISR, see below). This may be termed "sleep suspend".

Another more complex suspend may be offered, if an RTOS supports “blocking” API calls. Such a call permits the task to request a service or resource, which it will receive immediately if it is available, otherwise it is suspended until it is available. There may also be a timeout option whereby a task is resumed if the resource is not available in a specific timeframe.

Other Task States

Many RTOSes support other task states, but the definition of these and the terminology used varies. Possibilities include a “finished” state, which simply means that the task’s outermost function has exited (either by executing a **return** or just ending the outer function block). For a finished task to run again, it would probably need to be reset in some way.Another possibility is a “terminated” state. This is like a pure suspend, except that the task must be reset to its initial state in order to run again.If an RTOS supports dynamic creation and deletion of tasks (see the next article), this implies another possible task state: “deleted”.

Unit-V

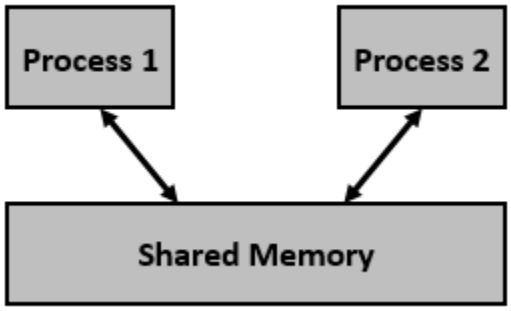
TASK COMMUNICATION

Shared memory is a memory shared between two or more processes. However, why do we need to share memory or some other means of communication?

To reiterate, each process has its own address space, if any process wants to communicate with some information from its own address space to other processes, then it is only possible with IPC (inter process communication) techniques. As we are already aware, communication can be between related or unrelated processes.

Usually, inter-related process communication is performed using Pipes or Named Pipes. Unrelated processes (say one process running in one terminal and another process in another terminal) communication can be performed using Named Pipes or through popular IPC techniques of Shared Memory and Message Queues.

We have seen the IPC techniques of Pipes and Named pipes and now it is time to know the remaining IPC techniques viz., Shared Memory, Message Queues, Semaphores, Signals, and Memory Mapping.



PIPES:

1. A message pipe is an inter-process task communication tool used for inserting, deleting messages between two given interconnected task or two sets of tasks.
2. In a pipe there are no fixed numbers of bytes per message but there is an end-point. A pipe can therefore be inserted limited number of bytes and have a variable number of bytes per message between initial and final pointers.
3. Pipes are unidirectional i.e. one thread or task inserts into it and the other one reads and deletes from it.
4. The read method of a pipe has no knowledge of where the write started, the interacting tasks must therefore share start and end locations as the messages are inserted into the pipe.

Memory mapped objects

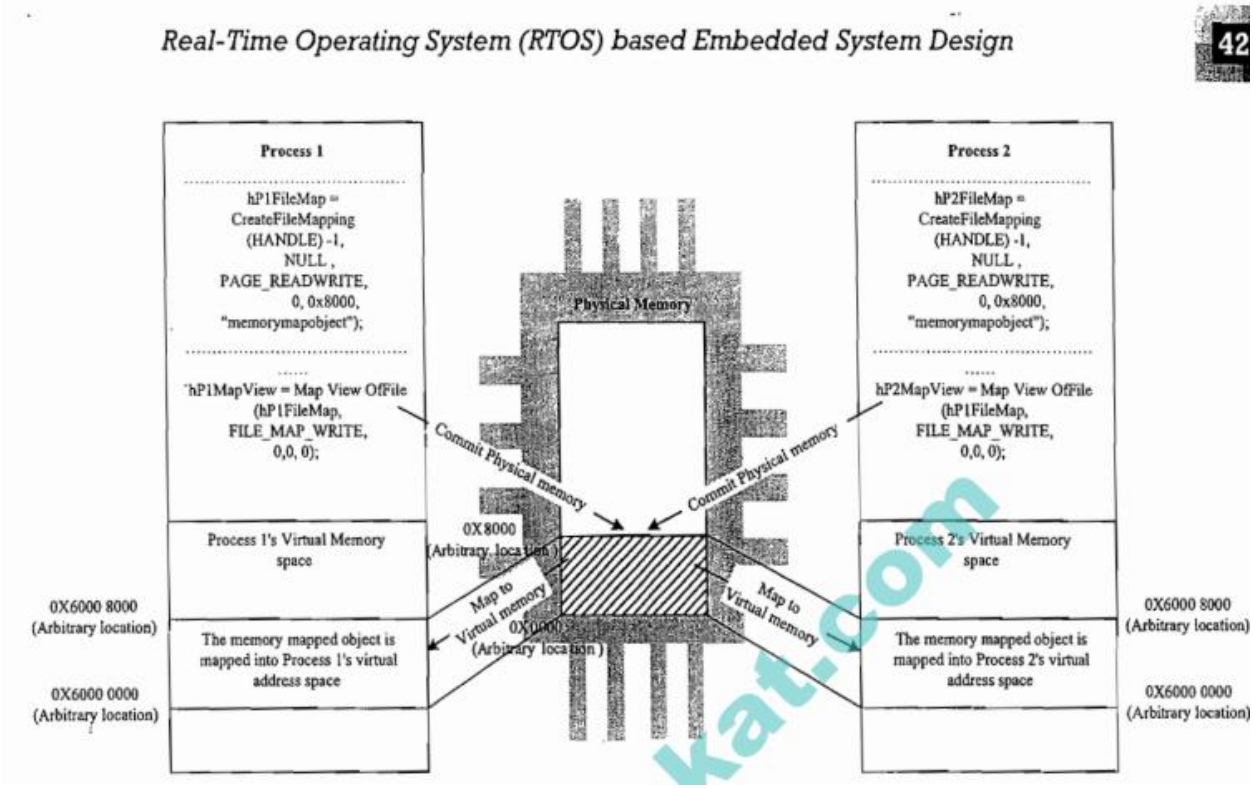
A memory-mapped file contains the contents of a file in virtual memory. This mapping between a file and memory space enables an application, including multiple processes, to modify the file by reading and writing directly to the memory. Starting with the .NET Framework 4, you can use managed code to access memory-mapped files in the same way that native Windows functions access memory-mapped files, as described in Managing Memory-Mapped File.

There are two types of memory-mapped files:

- Persisted memory-mapped files:Persisted files are memory-mapped files that are associated with a source file on a disk. When the last process has finished working with the file, the data is saved to the source file on the disk. These memory-mapped files are suitable for working with extremely large source files.

- Non-persisted memory-mapped files:Non-persisted files are memory-mapped files that are not associated with a file on a disk. When the last process has finished working with the file, the data is lost and the file is reclaimed by garbage collection.

These files are suitable for creating shared memory for inter-process communications (IPC).



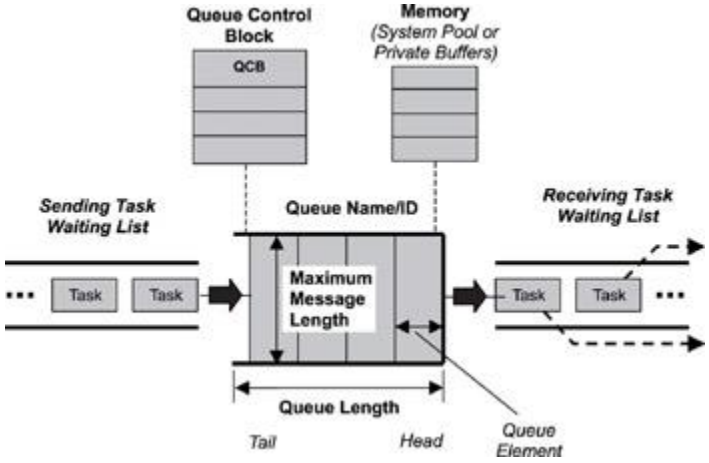
CONCEPT OF MEMORY BASED OBJECT

Message queues

A message queue provides an asynchronous communications protocol, which is a system that puts a message onto a message queue and does not require an immediate response to continuing processing. Email is probably the best example of asynchronous communication. When an email is sent, the sender continues to process other things without needing an immediate response from the receiver. This way of handling messages decouples the producer from the consumer so that they do not need to interact with the message queue at the same time.

A message queue is a buffer-like object through which tasks and ISRs send and receive messages to communicate and synchronize with data. A message queue is like a pipeline. It temporarily holds messages from a sender until the intended receiver is ready to read them. This temporary buffering decouples a sending and receiving task; that is, it frees the tasks from having to send and receive messages simultaneously.

A message queue has several associated components that the kernel uses to manage the queue. When a message queue is first created, it is assigned an associated queue control block (QCB), a message queue name, a unique ID, memory buffers, a queue length, a maximum message length, and one or more task-waiting lists



It is the kernel's job to assign a unique ID to a message queue and to create its QCB and task-waiting list. The kernel also takes developer-supplied parameters—such as the length of the queue and the maximum message length—to determine how much memory is required for the message queue. After the kernel has this information, it allocates memory for the message queue from either a pool of system memory or some private memory space.

The message queue itself consists of a number of elements, each of which can hold a single message. The elements holding the first and last messages are called the *head* and *tail* respectively. Some elements of the

queue may be empty (not containing a message). The total number of elements (empty or not) in the queue is the *total length of the queue* . The developer specified the queue length when the queue was created.

Mail box

- In general, mailboxes are similar to message queues. Mail box technique for inter task communication in RTOS based system used for one way messaging.
- The task/thread creates mail box to send the message. The receiver task can subscribe the mail box. The thread which creates the mail box is known as mailbox server.
- The others are known as client. RTOS has function to create, write and read from mail box. No of messages (limited or unlimited) in mail box have been decided by RTOS.

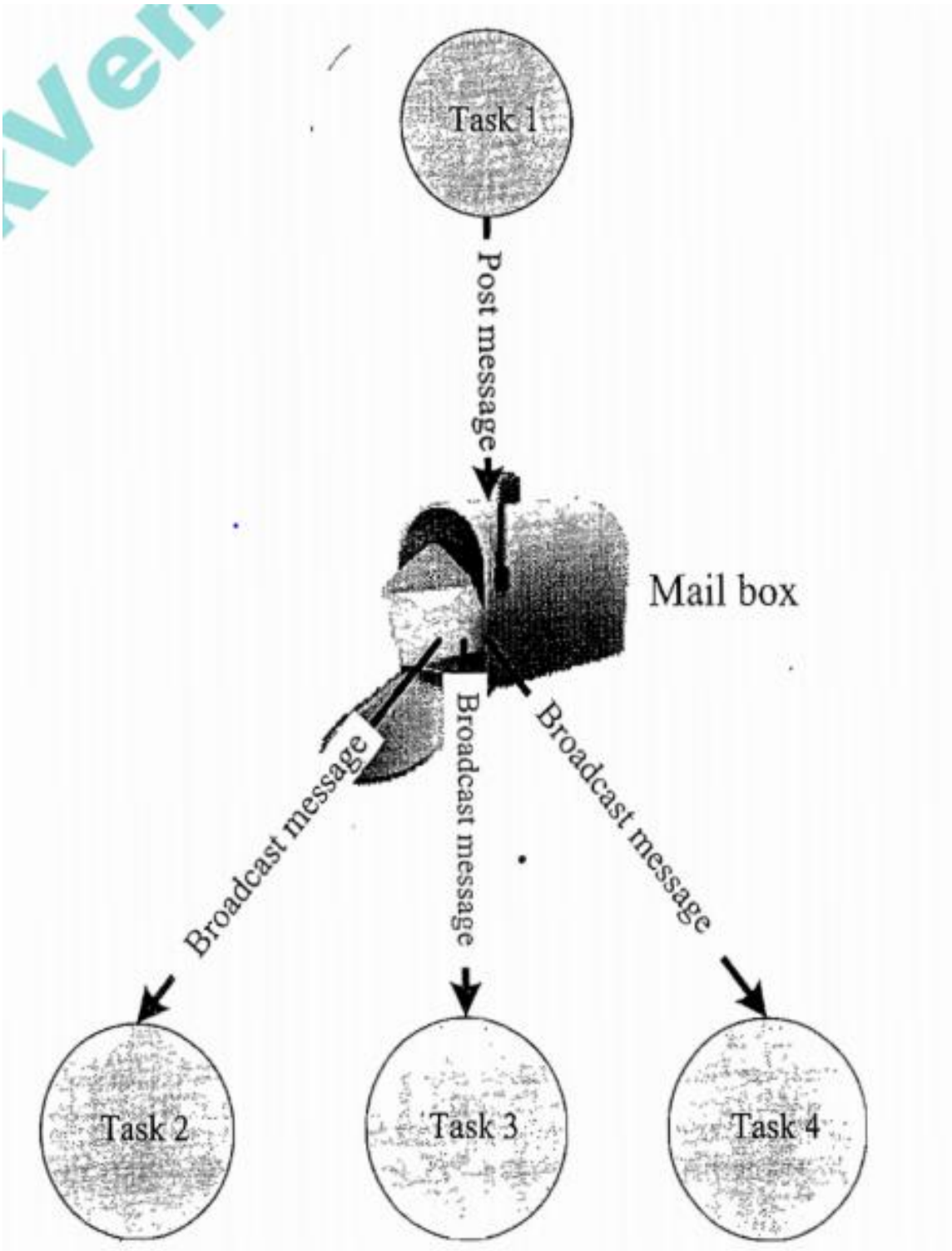


Fig. 10.21 **Concept of Mailbox based indirect messaging for IPC**

Remote procedure call:

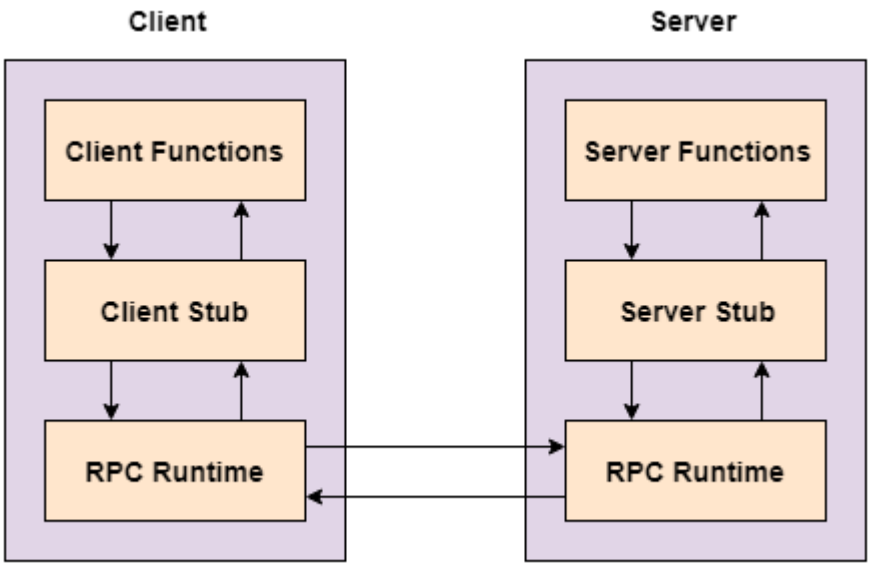
A remote procedure call is an inter process communication technique that is used for client-server based applications. It is also known as a subroutine call or a function call.

A client has a request message that the RPC translates and sends to the server. This request may be a procedure or a function call to a remote server. When the server receives the request, it sends the required response back to the client. The client is blocked while the server is processing the call and only resumed execution after the server is finished.

The sequence of events in a remote procedure call are given as follows:

- The client stub is called by the client.
- The client stub makes a system call to send the message to the server and puts the parameters in the message.
- The message is sent from the client to the server by the client's operating system.
- The message is passed to the server stub by the server operating system.
- The parameters are removed from the message by the server stub.
- Then, the server procedure is called by the server stub.

A diagram that demonstrates this is as follows:



Advantages of Remote Procedure Call

Some of the advantages of RPC are as follows:

- Remote procedure calls support process oriented and thread oriented models.
- The internal message passing mechanism of RPC is hidden from the user.
- The effort to re-write and re-develop the code is minimum in remote procedure calls.
- Remote procedure calls can be used in distributed environment as well as the local environment.

- Many of the protocol layers are omitted by RPC to improve performance.

Disadvantages of Remote Procedure Call

Some of the disadvantages of RPC are as follows:

- The remote procedure call is a concept that can be implemented in different ways. It is not a standard.
- There is no flexibility in RPC for hardware architecture. It is only interaction based.
- There is an increase in costs because of remote procedure call.

Socket:

A socket is a software object that acts as an end point establishing a bidirectional network communication link between a server-side and a client-side program.

In UNIX, a socket can also be referred to as an endpoint for inter process communication (IPC) within the operating system(OS).

In Java, socket classes represent the communication between client and server programs. Socket classes handle client-side communication, and server socket classes handle server-side communication.

Mostly URLs and their connections are used to access the Internet, but sometimes programs require a simple communication link between the program's client and server side. This role would be associated to a socket which would tie the program's client and server sides.

When a client establishes communication with the server, for example by querying the database, a reliable server and client connection is established via a TCP communication channel. In this type of communication, the client and server can read or write on sockets tied to specific communication channel.

Sockets are mainly classified into two types: active and passive. Active sockets are connected with the remote active sockets through an open data connection. If this connection is closed, the active sockets at each end point is destroyed. Passive sockets are not connected; instead, they wait for an incoming connection that will spawn a new active socket.

Even though there exists a close relationship between a socket and a port, the socket is not actually a port. Every port may have a single passive socket waiting for incoming connections and several active sockets each respective to an open connection in the port.

Sockets allow communication between two different processes on the same or different machines. To be more precise, it's a way to talk to other computers using standard Unix file descriptors. In Unix, every I/O action is done by writing or reading a file descriptor. A file descriptor is just an integer associated with an open file and it can be a network connection, a text file, a terminal, or something else.

To a programmer, a socket looks and behaves much like a low-level file descriptor. This is because commands such as read() and write() work with sockets in the same way they do with files and pipes.

Sockets were first introduced in 2.1BSD and subsequently refined into their current form with 4.2BSD. The sockets feature is now available with most current UNIX system releases.

Where is Socket Used?

A Unix Socket is used in a client-server application framework. A server is a process that performs some functions on request from a client. Most of the application-level protocols like FTP, SMTP, and POP3 make use of sockets to establish connection between client and server and then for exchanging data.

Socket Types

There are four types of sockets available to the users. The first two are most commonly used and the last two are rarely used.

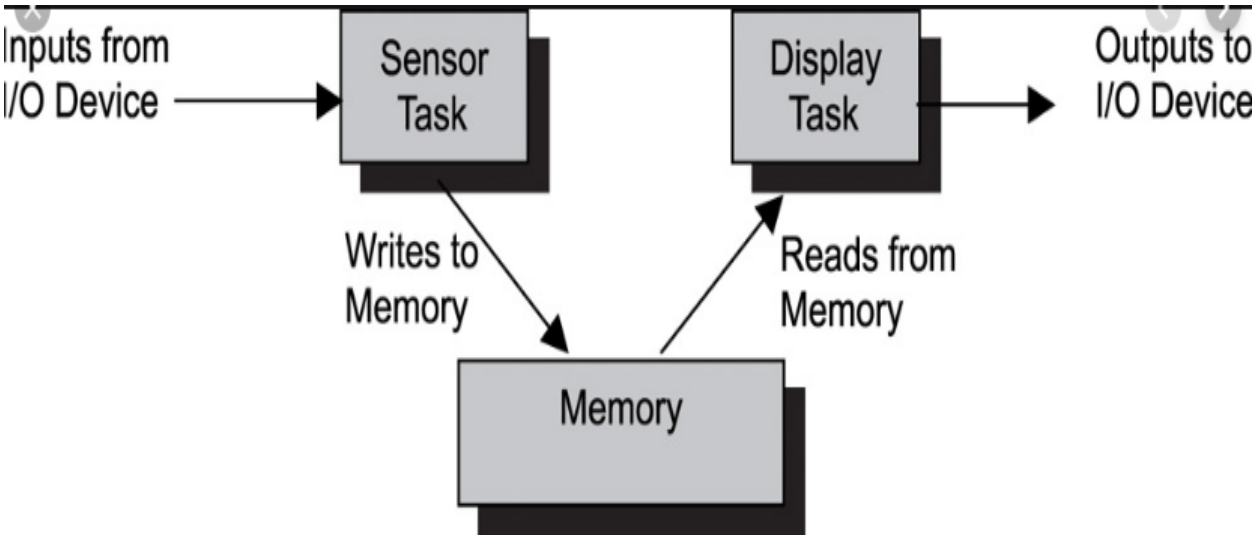
Processes are presumed to communicate only between sockets of the same type but there is no restriction that prevents communication between sockets of different types.

- **Stream Sockets** − Delivery in a networked environment is guaranteed. If you send through the stream socket three items "A, B, C", they will arrive in the same order − "A, B, C". These sockets use TCP (Transmission Control Protocol) for data transmission. If delivery is impossible, the sender receives an error indicator. Data records do not have any boundaries.
- **Datagram Sockets** − Delivery in a networked environment is not guaranteed. They're connectionless because you don't need to have an open connection as in Stream Sockets − you build a packet with the destination information and send it out. They use UDP (User Datagram Protocol).
- **Raw Sockets** − These provide users access to the underlying communication protocols, which support socket abstractions. These sockets are normally datagram oriented, though their exact characteristics are dependent on the interface provided by the protocol. Raw sockets are not intended for the general user; they have been provided mainly for those interested in developing new communication protocols, or for gaining access to some of the more cryptic facilities of an existing protocol.
- **Sequenced Packet Sockets** − They are similar to a stream socket, with the exception that record boundaries are preserved. This interface is provided only as a part of the Network Systems (NS) socket abstraction, and is very important in most serious NS applications. Sequenced-packet sockets allow the user to manipulate the Sequence Packet Protocol (SPP) or Internet Datagram Protocol (IDP) headers on a packet or a group of packets, either by writing a prototype header along with whatever data is to be sent, or by specifying a default header to be used with all outgoing data, and allows the user to receive the headers on incoming packets.

Task synchronization:

In general, a **task** must **synchronize** its activity with other **tasks** to execute a multithreaded program properly. Activity **synchronization** is also called condition **synchronization** or sequence control . ... For example,

in **embedded** control **systems**, a complex computation can be divided and distributed among multiple **tasks**.



Task communication/synchronization issues:

Deadlock: A deadlock is a situation in which two computer programs sharing the same resource are effectively preventing each other from accessing the resource, resulting in both programs ceasing to function

The earliest computer operating systems ran only one program at a time. All of the resources of the system were available to this one program. Later, operating systems ran multiple programs at once, interleaving them. Programs were required to specify in advance what resources they needed so that they could avoid conflicts with other programs running at the same time. Eventually some operating systems offered dynamic allocation of resources. Programs could request further allocations of resources after they had begun running. This led to the problem of the deadlock. Here is the simplest example:

Program 1 requests resource A and receives it.

Program 2 requests resource B and receives it.

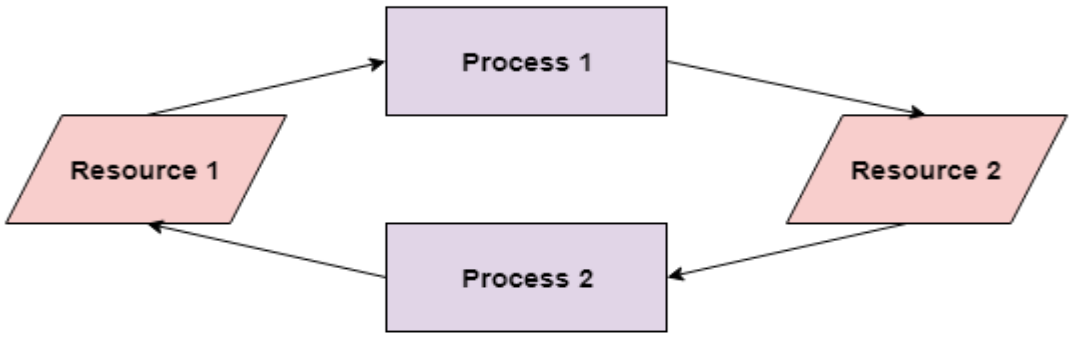
Program 1 requests resource B and is queued up, pending the release of B.

Program 2 requests resource A and is queued up, pending the release of A.

Now neither program can proceed until the other program releases a resource. The operating system cannot know what action to take. At this point the only alternative is to abort (stop) one of the programs.

Learning to deal with deadlocks had a major impact on the development of operating systems and the structure of databases. Data was structured and the order of requests was constrained in order to avoid creating deadlocks.

A deadlock happens in operating system when two or more processes need some resource to complete their execution that is held by the other process.



Deadlock in Operating System

In the above diagram, the process 1 has resource 1 and needs to acquire resource 2. Similarly process 2 has resource 2 and needs to acquire resource 1. Process 1 and process 2 are in deadlock as each of them needs the other's resource to complete their execution but neither of them is willing to relinquish their resources.

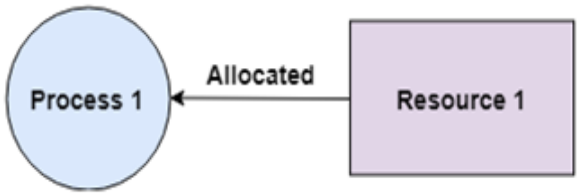
Coffman Conditions

A deadlock occurs if the four Coffman conditions hold true. But these conditions are not mutually exclusive.

The Coffman conditions are given as follows:

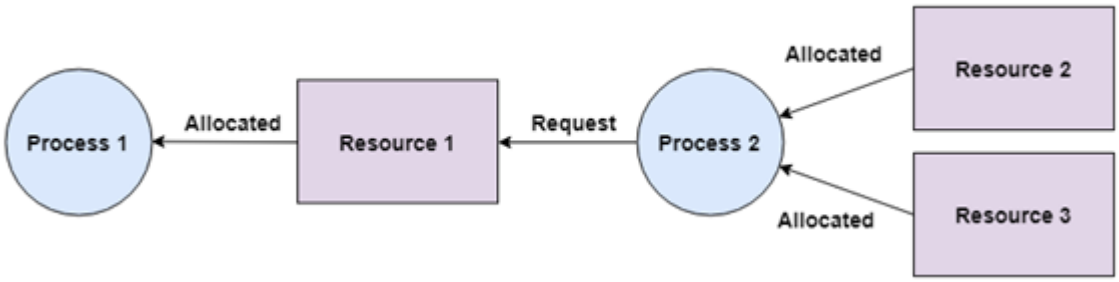
1.Mutual Exclusion

There should be a resource that can only be held by one process at a time. In the diagram below, there is a single instance of Resource 1 and it is held by Process 1 only.



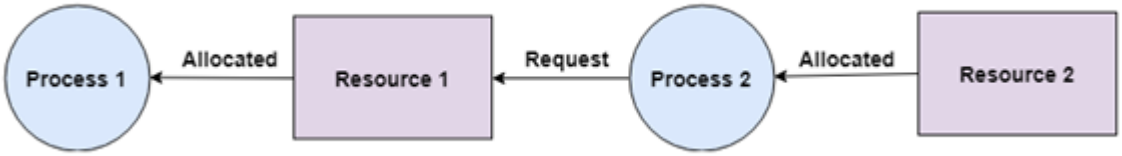
2.Hold and Wait

A process can hold multiple resources and still request more resources from other processes which are holding them. In the diagram given below, Process 2 holds Resource 2 and Resource 3 and is requesting the Resource 1 which is held by Process 1.



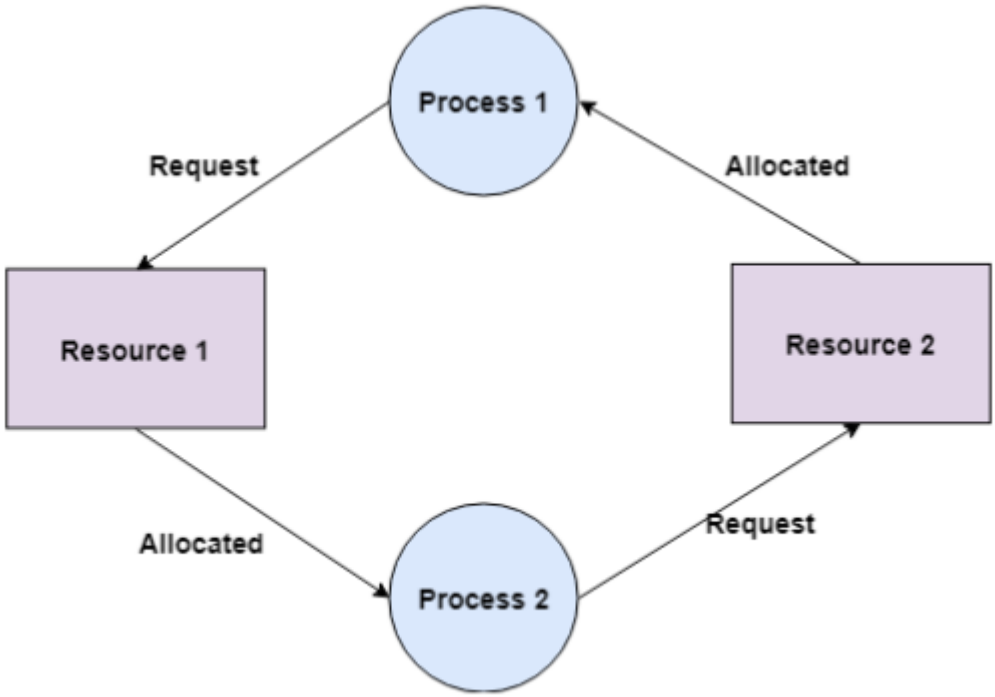
3.No Preemption

A resource cannot be preempted from a process by force. A process can only release a resource voluntarily. In the diagram below, Process 2 cannot preempt Resource 1 from Process 1. It will only be released when Process 1 relinquishes it voluntarily after its execution is complete.



4.Circular Wait

A process is waiting for the resource held by the second process, which is waiting for the resource held by the third process and so on, till the last process is waiting for a resource held by the first process. This forms a circular chain. For example: Process 1 is allocated Resource2 and it is requesting Resource 1. Similarly, Process 2 is allocated Resource 1 and it is requesting Resource 2. This forms a circular wait loop.



Deadlock Detection

A deadlock can be detected by a resource scheduler as it keeps track of all the resources that are allocated to different processes. After a deadlock is detected, it can be resolved using the following methods:

1. All the processes that are involved in the deadlock are terminated. This is not a good approach as all the progress made by the processes is destroyed.
2. Resources can be preempted from some processes and given to others till the deadlock is resolved.

Deadlock Prevention

It is very important to prevent a deadlock before it can occur. So, the system checks each transaction before it is executed to make sure it does not lead to deadlock. If there is even a slight chance that a transaction may lead to deadlock in the future, it is never allowed to execute.

Deadlock Avoidance

It is better to avoid a deadlock rather than take measures after the deadlock has occurred. The wait for graph can be used for deadlock avoidance. This is however only useful for smaller databases as it can get quite complex in larger databases.

Live lock: Live lock occurs when two or more processes continually repeat the same interaction in response to changes in the other processes without doing any useful work. These processes are not in the waiting state, and they are running concurrently. This is different from a deadlock because in a deadlock all processes are in the waiting state.

Or

A **live lock** is similar to a **deadlock**, except that the states of the processes involved in the live lock constantly change with regard to one another, none progressing. Live lock is a special case of resource **starvation**; the general definition only states that a specific process is not progressing.

Starvation:

Starvation occurs if a process is indefinitely postponed. This may happen if the process requires a resource for execution that it is never alloted or if the process is never provided the processor for some reason.

Some of the common causes of starvation are as follows:

1. If a process is never provided the resources it requires for execution because of faulty resource allocation decisions, then starvation can occur.
2. A lower priority process may wait forever if higher priority processes constantly monopolize the processor.
3. Starvation may occur if there are not enough resources to provide to every process as required.
4. If random selection of processes is used then a process may wait for a long time because of non-selection.

Some solutions that can be implemented in a system to handle starvation are as follows:

1. An independent manager can be used for allocation of resources. This resource manager distributes resources fairly and tries to avoid starvation.
2. Random selection of processes for resource allocation or processor allocation should be avoided as they encourage starvation.
3. The priority scheme of resource allocation should include concepts such as aging, where the priority of a process is increased the longer it waits. This avoids starvation.

2.DINING PHILOSOPHER'S PROBLEM:

The dining philosophers problem states that there are 5 philosophers sharing a circular table and they eat and think alternatively. There is a bowl of rice for each of the philosophers and 5 chopsticks. A philosopher needs both their right and left chopstick to eat. A hungry philosopher may only eat if there are both chopsticks available. Otherwise a philosopher puts down their chopstick and begin thinking again.

The dining philosopher is a classic synchronization problem as it demonstrates a large class of concurrency control problems.

Difficulty with the solution

The above solution makes sure that no two neighboring philosophers can eat at the same time. But this solution can lead to a deadlock. This may happen if all the philosophers pick their left chopstick simultaneously. Then none of them can eat and deadlock occurs.

Some of the ways to avoid deadlock are as follows:

- There should be at most four philosophers on the table.

- An even philosopher should pick the right chopstick and then the left chopstick while an odd philosopher should pick the left chopstick and then the right chopstick.
- A philosopher should only be allowed to pick their chopstick if both are available at the same time.

3. producer consumer problem/bounded buffer problem:

The producer consumer problem is a synchronization problem. There is a fixed size buffer and the producer produces items and enters them into the buffer. The consumer removes the items from the buffer and consumes them.

A producer should not produce items into the buffer when the consumer is consuming an item from the buffer and vice versa. So the buffer should only be accessed by the producer or consumer at a time.

The producer consumer problem can be resolved using semaphores. The codes for the producer and consumer process are given as follows:

Producer Process

The code that defines the producer process is given below:

```
do {
      .
      . PRODUCE ITEM
      .
      wait(empty);
      wait(mutex);
      .
      . PUT ITEM IN BUFFER
      .
      signal(mutex);
      signal(full);

} while(1);
```

In the above code, mutex, empty and full are semaphores. Here mutex is initialized to 1, empty is initialized to n (maximum size of the buffer) and full is initialized to 0.

The mutex semaphore ensures mutual exclusion. The empty and full semaphores count the number of empty and full spaces in the buffer.

After the item is produced, wait operation is carried out on empty. This indicates that the empty space in the buffer has decreased by 1. Then wait operation is carried out on mutex so that consumer process cannot interfere.

After the item is put in the buffer, signal operation is carried out on mutex and full. The former indicates that consumer process can now act and the latter shows that the buffer is full by 1.

Consumer Process

The code that defines the consumer process is given below:

```
do {

      wait(full);
      wait(mutex);
      .        .
      .  REMOVE ITEM FROM BUFFER
      .
      signal(mutex);
      signal(empty);
      .
      . CONSUME ITEM
      .
} while(1);
```

The wait operation is carried out on full. This indicates that items in the buffer have decreased by 1. Then wait operation is carried out on mutex so that producer process cannot interfere.

Then the item is removed from buffer. After that, signal operation is carried out on mutex and empty. The former indicates that consumer process can now act and the latter shows that the empty space in the buffer has increased by 1.

4.READER WRITER PROBLEM:

The readers-writers problem relates to an object such as a file that is shared between multiple processes. Some of these processes are readers i.e. they only want to read the data from the object and some of the processes are writers i.e. they want to write into the object.

The readers-writers problem is used to manage synchronization so that there are no problems with the object data. For example - If two readers access the object at the same time there is no problem. However if two writers or a reader and writer access the object at the same time, there may be problems.

To solve this situation, a writer should get exclusive access to an object i.e. when a writer is accessing the object, no reader or writer may access it. However, multiple readers can access the object at the same time.

This can be implemented using semaphores. The codes for the reader and writer process in the reader-writer problem are given as follows:

Reader Process

The code that defines the reader process is given below:

```
wait (mutex);
rc ++;
if (rc == 1)
wait (wrt);
signal(mutex);
.
.  READ THE OBJECT
.
wait(mutex);
rc --;
if (rc == 0)
signal (wrt);
signal(mutex);
```

In the above code, mutex and wrt are semaphores that are initialized to 1. Also, rc is a variable that is initialized to 0. The mutex semaphore ensures mutual exclusion and wrt handles the writing mechanism and is common to the reader and writer process code.

The variable rc denotes the number of readers accessing the object. As soon as rc becomes 1, wait operation is used on wrt. This means that a writer cannot access the object anymore. After the read operation is done, rc is decremented. When re becomes 0, signal operation is used on wrt. So a writer can access the object now.

Writer Process

The code that defines the writer process is given below:

```
wait(wrt);
.
.  WRITE INTO THE OBJECT
.
signal(wrt);
```

If a writer wants to access the object, wait operation is performed on wrt. After that no other writer can access the object. When a writer is done writing into the object, signal operation is performed on wrt.

Priority inversion:

Priority inversion is a operating system scenario in which a higher priority process is preempted by a lower priority process. This implies the inversion of the priorities of the two processes.

Problems due to Priority Inversion

Some of the problems that occur due to priority inversion are given as follows:

1. A system malfunction may occur if a high priority process is not provided the required resources.

2. Priority inversion may also lead to implementation of corrective measures. These may include the resetting of the entire system.
3. The performance of the system can be reduces due to priority inversion. This may happen because it is imperative for higher priority tasks to execute promptly.
4. System responsiveness decreases as high priority tasks may have strict time constraints or real time response guarantees.
5. Sometimes there is no harm caused by priority inversion as the late execution of the high priority process is not noticed by the system.

Solutions of Priority Inversion

Some of the solutions to handle priority inversion are given as follows:

1. **Priority Ceiling**

   All of the resources are assigned a priority that is equal to the highest priority of any task that may attempt to claim them. This helps in avoiding priority inversion.

2. **Disabling Interrupts**

   There are only two priorities in this case i.e. interrupts disabled and pre emption .So priority inversion is impossible as there is no third option.

3. **Priority Inheritance**

   This solution temporarily elevates the priority of the low priority task that is executing to the highest priority task that needs the resource. This means that medium priority tasks cannot intervene and lead to priority inversion.

4. **No blocking:** Priority inversion can be avoided by avoiding blocking as the low priority task blocks the high priority task.
5. **Random boosting:**

   The priority of the ready tasks can be randomly boosted until they exit the critical section**.**

TASK SYNCHRONISATION TECHNIQUES:

so far we discussed about the various task/process synchronization issues encountered multitasking systems due to concurrent resource access. Now lets have a discussion on the various techniques used for the synchronization in concurrent access in multitasking. process/task synchronization is essential for

- Avoiding conflicts in resource access(racing, deadlock ,starvation, live lock etc)in multitasking environment
- Ensuring proper sequence of operation across processes. The producer consumer problem is a typical example for process requiring proper sequence of operation. In producer consumer

problem, accessing the shared buffer by different process is not the issue, the issue is the writing process should write to the shared buffer only if the buffer is not full and the consumer thread should not read from the buffer if it is empty. hence proper synchronization should be provided to implement the sequence of operation.

- Communicating between processes

  The code memory area which holds the program instructions(piece of code) for accessing a shared resource (like shared memory, shared variables etc) is known as critical section.

Mutual exclusion through busy waiting/spin lock:

In software engineering, a **spinlock** is a lock which causes a thread trying to acquire it to simply wait in a loop ("spin") while repeatedly checking if the lock is available. Since the thread remains active but is not performing a useful task, the use of such a lock is a kind of busy waiting. Once acquired, spinlocks will usually be held until they are explicitly released, although in some implementations they may be automatically released if the thread being waited on (the one which holds the lock) blocks, or "goes to sleep".

Because they avoid overhead from operating system process rescheduling or context switching, spinlocks are efficient if threads are likely to be blocked for only short periods. For this reason, operating-system kernels often use spinlocks. However, spinlocks become wasteful if held for longer durations, as they may prevent other threads from running and require rescheduling. The longer a thread holds a lock, the greater the risk that the thread will be interrupted by the OS scheduler while holding the lock. If this happens, other threads will be left "spinning" (repeatedly trying to acquire the lock), while the thread holding the lock is not making progress towards releasing it. The result is an indefinite postponement until the thread holding the lock can finish and release it. This is especially true on a single-processor system, where each waiting thread of the same priority is likely to waste its quantum (allocated time where a thread can run) spinning until the thread that holds the lock is finally finished.

Implementing spin locks correctly offers challenges because programmers must take into account the possibility of simultaneous access to the lock, which could cause race conditions. Generally, such an implementation is possible only with special assembly-language instructions, such as atomic test-and-set operations and cannot be easily implemented in programming languages not supporting truly atomic operations.[1] On architectures without such operations, or if high-level language implementation is required, a non-atomic locking algorithm may be used, e.g. Peterson's algorithm. However, such an implementation may require more memory than a spinlock, be slower to allow progress after unlocking, and may not be implementable in a high-level language if out-of-order execution is allowed.

Mutual exclusion through sleep and wakeup:

The “busy waiting” mutual exclusion enforcement mechanism used by processes make the CPU always busy by checking the lock to see weather they can proceed. This results in the wastage of CPU time and leads to high power consumption. This is not affordable in embedded systems powered on battery, since it effects the battery backup time of the device. An alternative to “busy waiting” is the “sleep and wakeup” Mechanism. When a process is not allowed to access the critical section, which is currently being locked by another process, the process undergoes “sleep” and enters the “blocked” state. The process which is blocked on waiting for access to the critical section is awakened by the

process which currently owns the critical section. The process which owns the critical section sends a wakeup message to the process, which is sleeping as a result of waiting for the access to the critical section, when process leaves the critical section. "The sleep and wakeup" policy for mutual exclusion can be implemented in different ways. Implementation of this policy is OS kernel dependent.

The following section describes the important technique for "sleep and wakeup" policy implementation for mutual exclusion by windows XP per CE OS kernel.

Semaphore: A semaphore is a signaling mechanism and a thread that is waiting on a semaphore can be signaled by another thread. This is different than a mutex as the mutex can be signaled only by the thread that called the wait function.

A semaphore uses two atomic operations, wait and signal for process synchronization.

The wait operation decrements the value of its argument S, if it is positive. If S is negative or zero, then no operation is performed.

Types of Semaphores

There are two main types of semaphores i.e. counting semaphores and binary semaphores. Details about these are given as follows:

1. **Counting Semaphores**

   These are integer value semaphores and have an unrestricted value domain. These semaphores are used to coordinate the resource access, where the semaphore count is the number of available resources. If the resources are added, semaphore count automatically incremented and if the resources are removed, the count is decremented.

2. **Binary Semaphores**

   The binary semaphores are like counting semaphores but their value is restricted to 0 and 1. The wait operation only works when the semaphore is 1 and the signal operation succeeds when semaphore is 0. It is sometimes easier to implement binary semaphores than counting semaphores.

Advantages of Semaphores

Some of the advantages of semaphores are as follows:

1. Semaphores allow only one process into the critical section. They follow the mutual exclusion principle strictly and are much more efficient than some other methods of synchronization.
2. There is no resource wastage because of busy waiting in semaphores as processor time is not wasted unnecessarily to check if a condition is fulfilled to allow a process to access the critical section.
3. Semaphores are implemented in the machine independent code of the microkernel. So they are machine independent.

Disadvantages of Semaphores

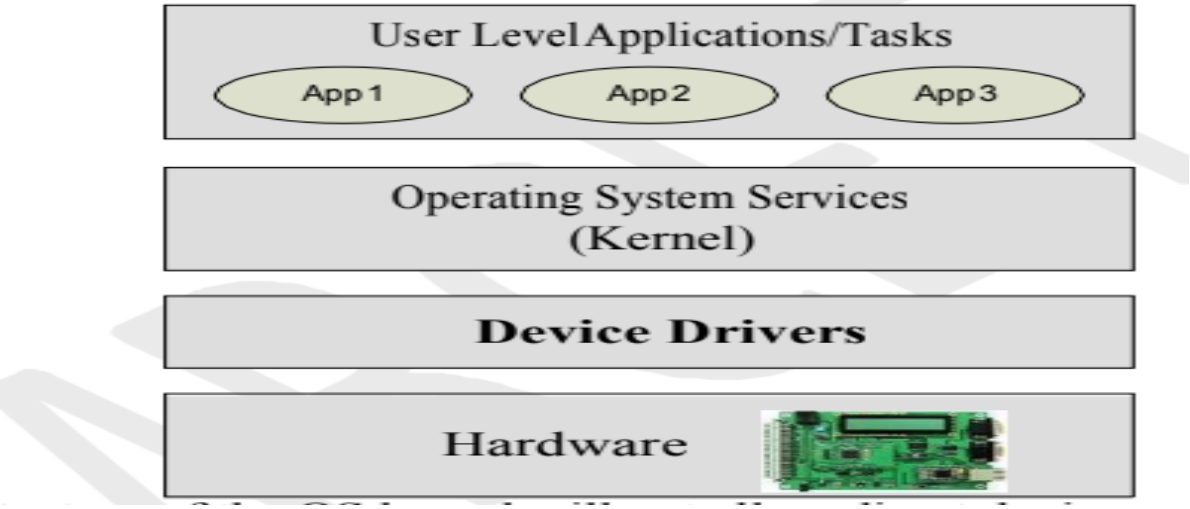
Some of the disadvantages of semaphores are as follows:

1. Semaphores are complicated so the wait and signal operations must be implemented in the correct order to prevent deadlocks.
2. Semaphores are impractical for last scale use as their use leads to loss of modularity. This happens because the wait and signal operations prevent the creation of a structured layout for the system.
3. Semaphores may lead to a priority inversion where low priority processes may access the critical section first and high priority processes later.

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.

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.

4.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 opensource 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 decisionmaking 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.

```
int msqid; /* return value from msgget() */

...

key = ...

msgflg = ...

if ((msqid = msgget(key, msgflg)) == &ndash;1)

 {

  perror("msgget: msgget failed");

  exit(1);

  } else

  (void) fprintf(stderr, &ldquo;msgget succeeded");
```

Mailbox:

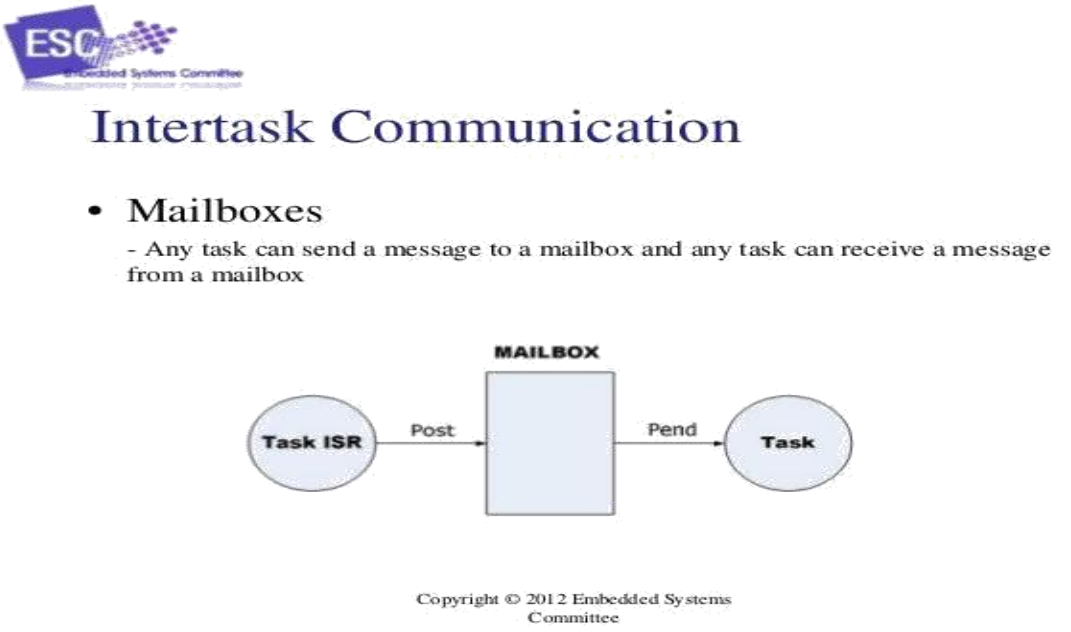
- Mailbox (for message) is an IPC through a message-block at an OS that can be used only by a single destined task.
-
- A task on an OS function call puts (means post and also send) into the mailbox nly a pointer to a mailbox message

- Mailbox message may also include a header to identify the message-type specification.

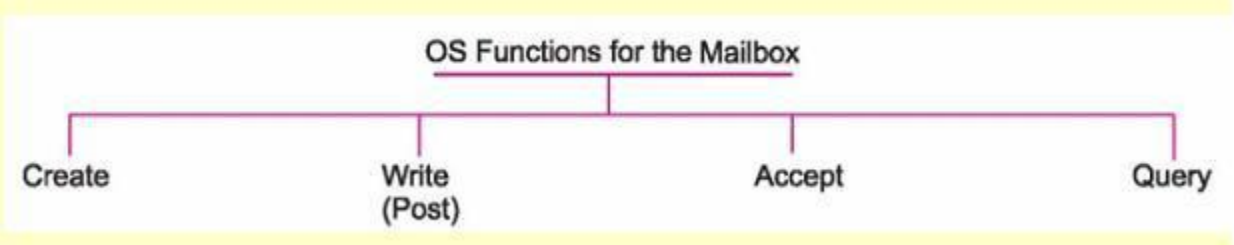
Mailbox IPC features:

•OS provides for inserting and deleting message into the mailbox message- pointer. Deleting eans message-pointer pointing to Null.

•Each mailbox for a message need initialization (creation) before using the functions in the scheduler for the message queue and message pointer pointing to null.



Mailbox Related Functions at the OS:



Pipe Function:

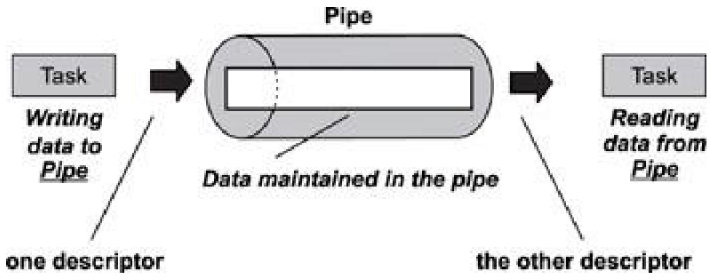
Pipe

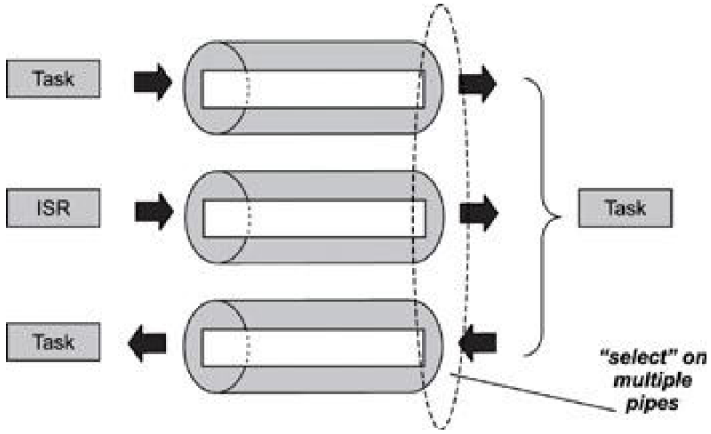
- Pipe is a device used for the interprocess communication
- Pipe has the functions create, connect and delete and functions similar to a device driver

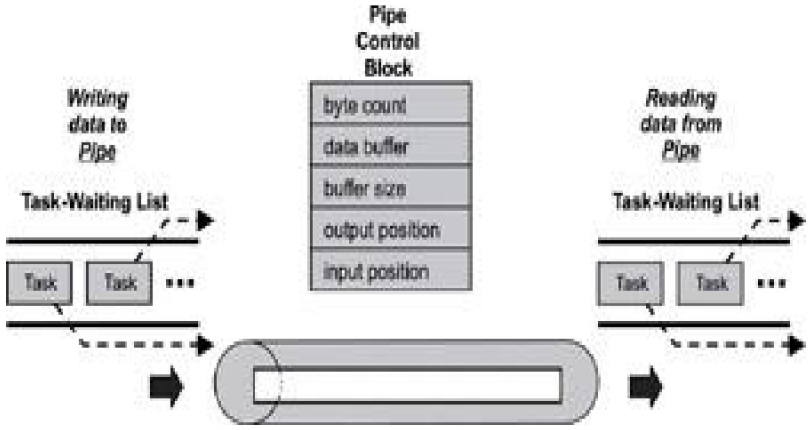
Writing and reading a Pipe:

• A message-pipe─ a device for inserting (writing) and deleting (reading) from that between two given inter-connected tasks or two sets of tasks.

• Writing and reading from a pipe is like using a C command*fwrite with a file name* to write into a named file, and C command *fread with a file name*to read into a named


Pipe
Task
Writing data to Pipe
Data maintained in the pipe
Task
Reading data from Pipe
one descriptor
the other descriptor


Task
ISR
Task
Task
"select" on multiple pipes


Pipe Control Block
byte count
data buffer
buffer size
output position
input position
Writing data to Pipe
Task-Waiting List
Task
Task
Reading data from Pipe
Task-Waiting List
Task
Task

Pipe function calls:

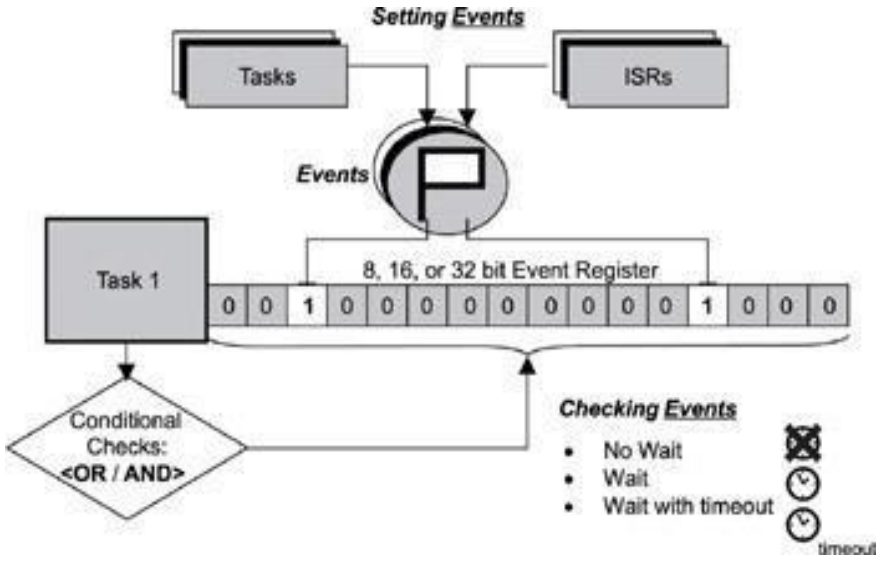
- Create a pipe
- Open pipe
- Close pipe
- Read from the pipe
- Write to the pipe

Event Functions:

- Wait for only one event (semaphore or mailboxmessage posting event)
- Event related OS functions can wait for number of events before initiating an action or wait for any of the predefined set of events
- Events for wait can be from different tasks or the ISRs

Event functions at OS:

Some OSes support and some don‘t support event functions for a group of event



Event registers function calls:

- Create an event register
- Delete an event register
- Query an event register
- Set an event register
- Clear an event register
- Each bit I an event register can be used to obtain the states of an event .
- A task can have an event register and other tasks can set/clear the bits in the event register

Signal:

- one way for messaging is to use an OS function signal ( ).
- Provided in Unix, Linux and several RTOSes.
- Unix and Linux OSes use signals profusely and have thirty-one different types of signals for the various events.
- A signal is the software equivalent of the flag at a register that sets on a hardware interrupt. Unless masked by a signal mask, the signal allows the execution of the Signal handling function and allows the handler to run just as a hardware interrupt allows the execution of an

ISR

- Signal provides the shortest communication.

Signal management fuction calls:

- Install a signal handler
- Remove an installed signal handler
- Send a signal to another task
- Block a signal from being delivered
- Unblock a blocked signal
- Ignore a signal

Timers:

- Real time clock ─ system clock, on each tick SysClkIntr interrupts
- Based on each SysClkIntr interrupts─ there are number of OS timer functions
- Timer are used to message the elasped time of events for instance , the kernel has to keep track of different times
- Get time
- Set time

- Time delay(in system clock)
- Time delay(in sec.)
- Reset timer

Memory management:

Memory allocation:

- Memory allocation When a process is created, the memory manager allocates the memory addresses (blocks) to it by mapping the process address space.
- Threads of a process share the memory space of the process

Memory Managing Strategy for a system

- Fixed
- blocks allocation
- Dynamic
- blocks Allocation
- Dynamic Page
- Allocation
- Dynamic Data memory Allocation

Interrupt service routine (ISR):

- Interrupt is a hardware signal that informs the cpu that an important event has occurred when interrupt occured, cpu saves its content and jumps to the ISR

- In RTOS

    - Interrupt latency
    - Interrupt responseoInterrupt recovery

Mutex:

Mutex standards for mutual exclusion ,mutex is the general mechanism used for both rsource synchronization as well as task synchronization

It has following mechanisms

- Disabling the scheduler

- Disabling the interrupts

- By test and set operations

- Using semaphore