UNIT I

Introduction to Embedded Processors, Devices and Communication Buses: Introduction to Embedded Systems - Design Metrics – Optimization Challenges in Embedded system Design - Embedded Processors – General Purpose Processor – Single Purpose Processor and Application Specific Instruction Set Processor - IC Terminology – Full-Custom/VLSI – Semi-Custom ASIC - PLD Introduction to RISC architecture, VLIW and DSP processors. Introduction to I/O Devices – Types - Synchronous, Iso-synchronous and Asynchronous Communications - Serial Communication – I2C, USB, CAN – Wireless Communication – IrDA – Bluetooth.

1.1 Introduction to Embedded systems:

An Embedded system is a system that has software embedded into computer hardware, which make system dedicated for an application or specific part of an application or product.

An embedded system is one that has dedicated purpose software embedded in computer hardware.

It is a dedicated computer based system for an application or product. It may be independent system or a part of large system. Its software usually embeds into a ROM or flash.

Examples:

Personal digital assistance (PDA), Printer, Cell phones, Automobiles, House hold appliances etc.

Application Examples:

Simple control: Front panel of microwave oven (function is very less)

Automobiles Examples: Automobile has 100's of microcontroller. 4 bit microcontroller for seat belt. 16/32 bit microcontroller is to control engine (complex function).

1.1.1 Characteristics of Embedded systems:

- Sophisticated functionality
- Real time operation
- Low manufacturing cost
- Application dependent processor
- Restricted memory
- Low power

1.1.2 Manufacturing cost:

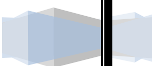
- Non-recurring engineering cost for design and development
- Cost for production and marketing each unit

1.1.3 Real time operation:

It must finish operation by dead lines

Hard real time: Missing dead line causes failures

Soft real time: Missing dead line result in degraded performance



Many systems are multi-rate. It must handle operations at widely varying rates from external world

1.1.4 Application dependent requirement:

Fault tolerance: It should continue operation despite of hardware or software faults.

Safe: Systems to avoid physical or economic damage to property.

1.1.5 Components of an embedded system:

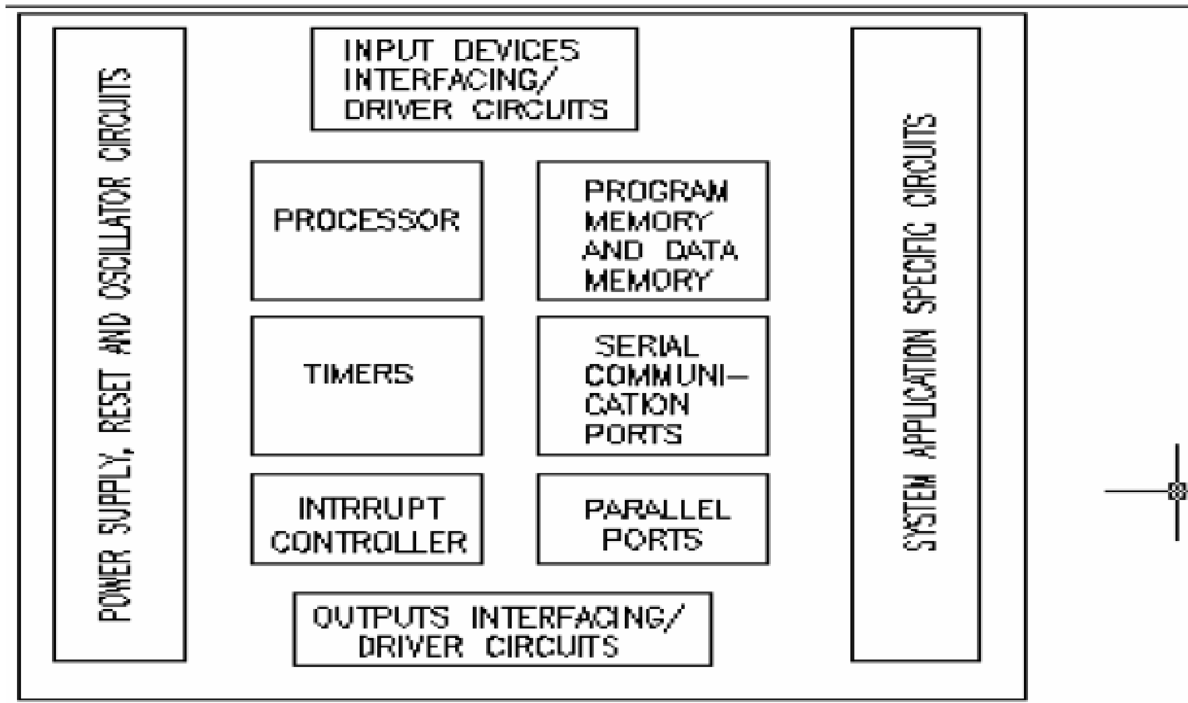


Fig: Components in embedded system

(i). Microprocessor:

Microprocessor is heart of any Real time embedded system. They are designed to meet specific requirements.

(ii). Memory:

The microprocessor and memory must coexist on same PCB. Compactness, Speed and low power consumption are characteristics required for memory used in real time embedded system (RTES) .

So, Low power semiconductor memories are used in all the devices. For housing operation system ROM is used. The program or data loaded may exist for considerable duration. It is changing setup of computer.

E.g.:- Changing ringtone of mobile, Screensaver etc.;

In this case memory should retain even after power should be removed. So non-volatile memory is used.

(iii). Input, Output devices and Interfaces:

Input/output interfaces are necessary to make RTES interact with external world. They could be screen in mobiles, Touchpad keyboards, Microphone etc. These RTES should also have open interface to other devices such as computer, LAN and other RTES.

e.g.:- We have to download address book from PDA

The input and output device provide necessary interface to support the standard.

(iv). Software:

RTES is just a physical body as long as it is not programmed. It is like a human body without life. When we switch ON our mobile phone we will notice activities on our screen. Battery low warning, Signal sign are taken care by RTOS (software) siting on non-volatile memory of RTES.

Besides the above an RTES have various other components and application specific integrated circuit (ASIC) for specific function such as motor control, Modulation, Demodulation, and CODEC etc.

1.2 Design metrics:

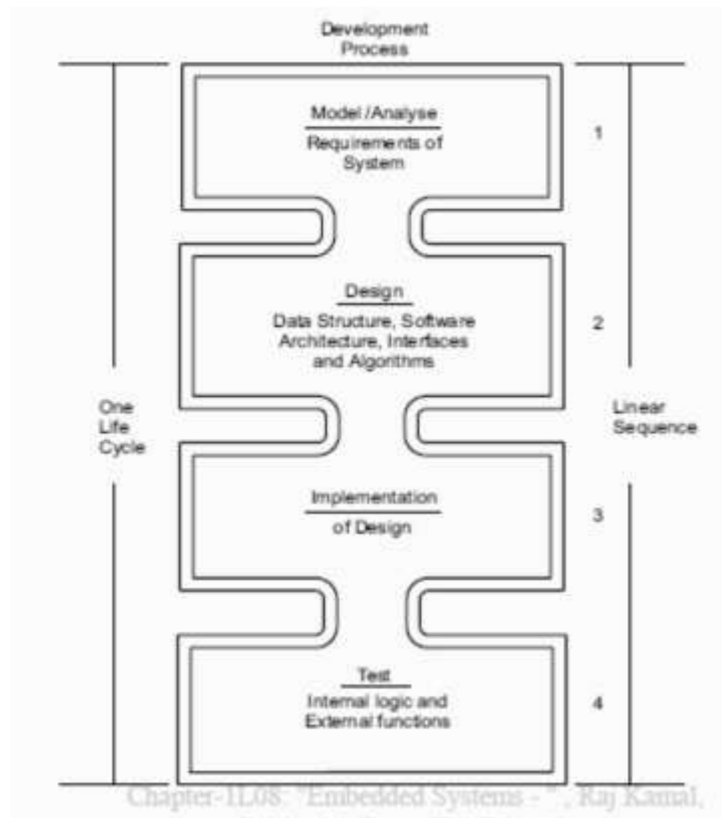
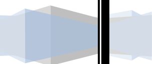


Fig: Design Metrics



1.2.1 Requirement:

It is only determined by a complete clarity of the required purpose, inputs, outputs, functioning, design metrics and validation requirements

Design Metrics:

(i). Power dissipation: For many systems, particularly battery operated systems, such as mobile phone or digital camera the power consumed by the system is an important feature. The battery needs to be charged less frequently if power dissipation is small

(ii). Performance: Smaller execution time means higher performance. For example, a mobile phone, voice signal processed between antenna and speaker is 0.1s shows phone performance.

(iii). Process Deadlines: There are number of processes in the system, Each process have deadlines. They have to complete the process within the required time and give results.

(iv). User interface: These includes GUIs and VUIs.

(v). Size: Size of the system is measured in terms of physical space required and RAM in kB and internal flash memory requirements in MB or GB for running the software and for data storage.

(vi). Engineering cost: Initial cost of developing, debugging and testing the hardware and software is called engineering cost and it is one time non recurring cost.

(vii). Manufacturing cost: It is the cost for manufacturing each unit.

(viii). Flexibility: Flexibility in a design enables, without any significance engineering cost, development of different version of a product and advanced version later on. For example, software enhancement by adding extra functions necessary by changing software re-engineering.

(ix). Prototype: Time taken in days or month for developing the prototype and in-house testing for system.

(x). Development time: It includes engineering time and making prototype time.

(xi). Time to market: Time taken in days or month after prototype development to put a product for user and customer.

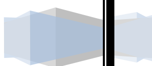
(xii). System and user safety: System safety in terms of accidental fall from hand or table, theft. For example phone tracking ability

(xiii). Maintenance: Maintenance means changeability and addition to the system. For example, adding or updating software, data and hardware. Example of data maintenance is additional ring tones, wallpaper etc.

1.2.2 Specification:

Clear specification of the required system are must. Specification need to be mentioned. The designer needs specification for

- Hardware, For example, peripherals, devices processor and memory specification.
- Data type and processing specification
- Expected system behavior specification
- constrains of design
- Expected life cycle specification



1.2.3 Architecture:

Data modeling design of attributes of data structure, data flow graph, Program model, software architecture layer and hardware architecture are defined. Software architecture layers are as follows:

- First layer is an architectural design. Here a design for system architecture is developed. Different element like data structures, databases, algorithms, control functions program flow are organized
- The second layer is an data design. Here a data organized type is mentioned. For example tree like structure.
- The third layer is interface design. Here interfacing the components is mentioned in detail

1.2.4. Components:

The fourth layer is a component level design. There is an additional requirement in design of embedded system, be optimized for memory usage and power dissipation. The following lists are the common hardware components.

- Processor, ASIP and single purpose processor in system
- Memory RAM,, ROM or internal and external flash memory in system
- Peripherals and devices internal and external to the systems
- Ports and busses in the system
- Power source or battery in the systems

1.2.5. System integration:

Build components are integrated in the system. Components may work fine independently, but when integrated may not fulfill the design metrics. The system is made to function and validated.

1.3 Optimizing challenges in design metrics:

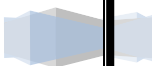
1.3.1 Clock rate reduction:

Power dissipation typically reduces 2.5µW per 100kHz of reduced clock rate. So reduction from 8000 kHz to 100 kHz reduces power dissipation by about 200µW, which is nearly similar to when the clock is non-functional.

The power 25µW is typically the residual dissipation needed to operate the timer and few other units. By operating the clock at lower frequency the advantage is power loss due to heat generation reduces.

1.3.2 Voltage reduction:

In portable or hand-held devices such as cellular phone, compared to 5v operation, CMOS circuit power dissipation reduces by one sixth, ~ $(2v/5v)^2$, in 2.0v operation.



Thus the time interval needed for recharging the battery increases by factor of six.

1.3.3 Wait, Stop and Cache Disable instructions:

An embedded system need to run continuously, without being switched off; the system design, therefore, is constrained by the need to limit power dissipation while it is ON but is in idle state. Total power consumption by the system while in running, waiting and idle states should be limited.

A microcontroller must provide Wait and Stop instruction for power down mode. One way to reduce power dissipation is by wait and stop instruction. Another is to operate the system at lowest voltage level in idle state and selecting power down mode in that state.

1.3.4 Process deadlines:

Meeting the deadline of all processes in the system while keeping the memory, power dissipation, processor clock rate and cost at minimum is a challenge.

1.3.5 Flexibility and Upgrade ability:

Flexibility and upgrade ability in design while keeping the cost minimum and without any significant engineering cost is a challenge. Flexibility and upgrade ability allow different and advanced version of a product to be introduced in market later on.

1.3.6 Reliability:

Designing a reliable product by appropriate deign, testing and verification is a challenge. The goal of testing is to find errors and to validate the software as per specification and requirement. Verification is to ensure that the specific function are correctly implemented. Validation is an to ensure that the system works correctly as per its requirement.

1.4 Embedded Processor:

Processor technology involves the architecture of the computation engine used to implement a system's desired functionality. There are three types of processor in designing embedded systems.

(i) General purpose processor
(ii) Single purpose processor
(iii) Application specific processor

1.4.1 General purpose processor:

The general-purpose processor is to build a device suitable for a variety of applications, to maximize the number of devices sold. One feature of such a processor is a program memory. The designer does not know what program will run on the processor, so cannot build the program into the digital circuit. Another feature is a general data path. The data path must be general enough to handle a variety of computations, so large register file and one or more general-purpose arithmetic-logic units (ALUs) are used. An embedded system designer simply

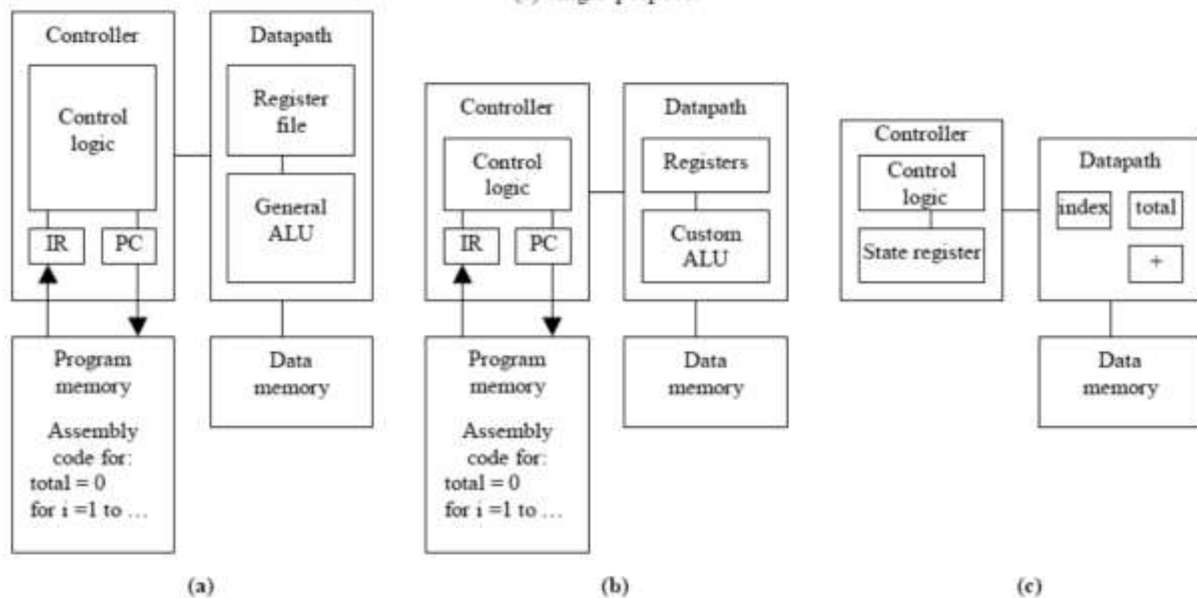
uses a general-purpose processor, by programming the processor's memory to carry out the required functionality.

Using a general-purpose processor in an embedded system may result in several design-metric benefits. **Design time** and **NRE** cost are **low**, because the designer must only write a program, but need not do any digital design. **Flexibility** is **high**, because changing functionality requires only changing the program. **Unit cost** may be relatively **low** in **small quantities**, since the processor manufacturer sells large quantities to other customers and hence distributes the NRE cost over many units. Performance may be fast for computation-intensive applications, if using a fast processor, due to advanced architecture features and leading edge IC technology.

However, there are also some design-metric drawbacks**. Unit cost** may be too **high** for **large quantities. Performance** may be **slow** for certain applications. **Size** and **power** may be **large** due to unnecessary processor hardware.

Figure 1.6(a) shows a simple architecture of a general-purpose processor implementing the array summing functionality. The functionality is stored in a program memory. The controller fetches the current instruction, as indicated by the program counter (PC), into the instruction register (IR). It then configures the data path for this instruction and executes the instruction. Finally, it determines the appropriate next instruction address, sets the PC to this address, and fetches again.

Figure 1.6: Implementing desired functionality on different processor types: (a) general-purpose, (b) application-specific, (c) single-purpose.



1.4.2 Single purpose processor:

A single-purpose processor is a digital circuit designed to execute exactly one program. Using a single-purpose processor in an embedded system results in several design metric benefits and drawbacks, which are essentially the inverse of those for general purpose processors.

Performance may be **fast**, **size** and **power** may be **small**, and **unit-cost** may be **low** for **large quantities**, while **design time** and **NRE costs** may be **high**, **flexibility** is **low**, **unit cost** may be **high** for **small quantities**, and performance may not match general-purpose processors for some applications.

Figure 1.6(c) illustrates the architecture of such a single-purpose processor for the example. Since the example counts from one to N, we add an index register. The index register will be loaded with N, and will then count down to zero, at which time it will assert a status line read by the controller. Since the example has only one other value, we add only one register labeled total to the datapath. Since the example's only arithmetic operation is addition, we add a single adder to the datapath. Since the processor only executes this one program, we hardwire the program directly into the control logic.

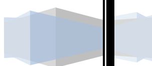
1.4.3 Application Specific processor:

An ASIP is designed for a particular class of applications with common characteristics, such as digital-signal processing, telecommunications, embedded control, etc. The designer of such a processor can optimize the data path for the application class, perhaps adding special functional units for common operations, and eliminating other infrequently used units. Using an ASIP in an embedded system can provide the **benefit** of **flexibility** while still achieving **good performance**, **power** and **size**. However, such processors can require **large NRE cost** to build the processor itself, and to build a compiler, if these items don't already exist. A DSP is a processor designed to perform common operations on digital signals, which are the digital encodings of analog signals like video and audio. These operations carry out common signal processing tasks like signal filtering, transformation, or combination. Such operations are usually math-intensive, including operations like multiply and add or shift and add. To support such operations, a DSP may have special purpose data path components such a multiply-accumulate unit.

Figure 1.6(b) shows the general architecture of an ASIP for the example. The data path may be customized for the example. It may have an auto-incrementing register, a path that allows the addition of a register plus a memory location in one instruction, fewer registers, and a simpler controller.

1.5 IC Terminology:

Every processor must eventually be implemented on an IC. IC technology involves the manner in which we map a digital (gate-level) implementation onto an IC. An IC (Integrated Circuit), often called a "chip," is a semiconductor device consisting of a set of connected transistors and other devices. To understand the differences among IC technologies, we must first recognize that semiconductors consist of numerous layers. The bottom layers form the transistors. The middle layers form logic gates. The top layers connect these gates with wires. One way to create these layers is by depositing photo-sensitive chemicals on the chip surface and then shining light through masks to change regions of the chemicals. Thus, the task of building



the layers is actually one of designing appropriate masks. A set of masks is often called a layout. The narrowest line that we can create on a chip is called the feature size, which today is well below one micrometer (sub-micron). For each IC technology, all layers must eventually be built to get a working IC.

1.5.1 Full Custom/VLSI:

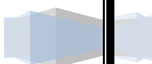
In a full-custom IC technology, we optimize all layers for our particular embedded system's digital implementation. Such optimization includes placing the transistors to minimize interconnection lengths, sizing the transistors to optimize signal transmissions and routing wires among the transistors. Once we complete all the masks, we send the mask specifications to a fabrication plant that builds the actual ICs. Full-custom IC design, often referred to as VLSI (Very Large Scale Integration) design, has **very high NRE cost** and **long turnaround times** (typically months) before the IC becomes available, but can yield **excellent performance** with **small size** and **power**. It is usually used only in high-volume or **extremely performance** critical applications.

1.5.2 Semi-custom ASIC:

In an ASIC (Application-Specific IC) technology, the lower layers are fully or partially built, leaving us to finish the upper layers. In a gate array technology, the masks for the transistor and gate levels are already built (i.e., the IC already consists of arrays of gates). The remaining task is to connect these gates to achieve our particular implementation. In a standard cell technology, logic-level cells (such as an AND gate or an AND-OR-INVERT combination) have their mask portions pre-designed, usually by hand. Thus, the remaining task is to arrange these portions into complete masks for the gate level, and then to connect the cells. ASICs are by far the most popular IC technology, as they provide **for good performance** and **size**, with **much less NRE cost** than full-custom IC's.

1.5.3 PLD:

In a PLD (Programmable Logic Device) technology, all layers already exist, so we can purchase the actual IC. The layers implement a programmable circuit, where programming has a lower-level meaning than a software program. The programming that takes place may consist of creating or destroying connections between wires that connect gates, either by blowing a fuse, or setting a bit in a programmable switch. Small devices, called programmers, connected to a desktop computer can typically perform such programming. We can divide PLD's into two types, simple and complex. One type of simple PLD is a PLA (Programmable Logic Array), which consists of a programmable array of AND gates and a programmable array of OR gates. Another type is a PAL (Programmable Array Logic), which uses just one programmable array to reduce the number of expensive programmable components. One type of complex PLD, growing very rapidly in popularity over the past decade, is the FPGA (Field Programmable Gate Array), which



offers more general connectivity among blocks of logic, rather than just arrays of logic as with PLAs and PALs, and is thus able to implement far more complex designs.

PLDs offer **very low NRE cost** and almost instant IC availability. However, they are typically bigger than ASICs, may have **higher unit cost**, may consume **more power**, and may be **slower** (especially FPGAs). They still provide **reasonable performance**, though, so are especially well suited to **rapid prototyping**.

1.6 Introduction to RISC architecture:

A RISC microprocessor provides the speedy processing of instruction each in a single clock cycle. This facilitates pipelining and superscalar processing. Besides greatly enhanced capabilities in speed of instruction is processed. RISC is used when the system needs to perform intensive computation in a speech processing system.

1.6.1 VLIW and DSP processor

A digital signal processor (DSP) is a processor core or chip for the applications that process digital signals. A microprocessor is essential for a computing system; similarly DSP is essential for an embedded system in a large number of applications needing processing of signals. Examples are in image processing, multimedia, audio, video, HDTV, DSP modem and telecommunication processing systems.

DSP executes discrete time, signal processing instructions. It has Very Large Instruction Word (VLIW) processing capabilities; it process Single Instruction Multiple Data (SIMD).

1.7 Introduction to IO devices:

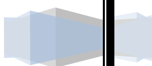
A serial port is a port for serial communication. Serial communication is for a given line or channel **one bit can communicate** and the bits transmit at periodic intervals generated by clock. A **serial port** communication is over **short or long distances**.

A parallel port is a port for parallel communication. Parallel communication means that **multiple bits can communicate** over a set of parallel line at any given time. A **parallel port** communication is over **very short distances** of less than a meter.

Ports can be wireless. **Wireless communication** can be used **without wires** over **short range** personal area network.

Serial and parallel ports of IO devices can be classified into following types:

i) Synchronous serial input
ii) Synchronous serial output
iii) Asynchronous serial input
iv) Asynchronous serial output
v) Parallel port input
vi) Parallel port output



1.7.1 Synchronous serial input:

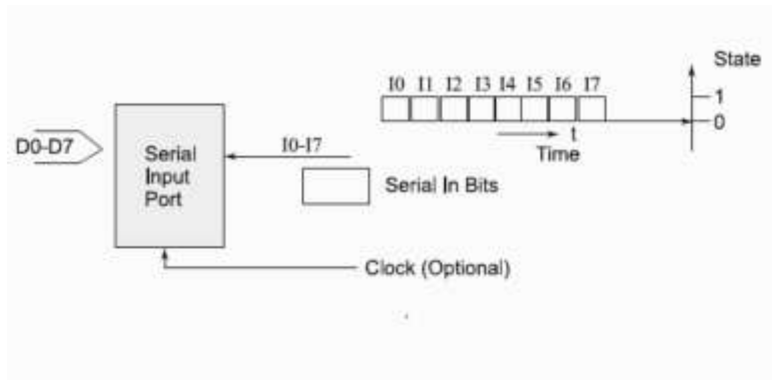


Fig shows synchronous serial input. Each bit in a byte are in synchronous. If a clock period equals T, then each byte at the port is received at input in period 8T.

The serial data input and clock pulse input are on same input line when the clock pulse encoded with serial data input. The receiver detects clock pulses and receives data bits after decoding the input data.

Synchronous serial input is also called as master output slave input(MOSI) when CLK is sent from sender to receiver and receiver is forced to synchronize the sent input from master(sender) clock.

It is also called as Master input slave output(MISO) when CLK is sent to sender(slave) from the receiver(master) and sender(slave) is forced to synchronize sending input as per master clock.

1.7.2 Synchronous serial output:

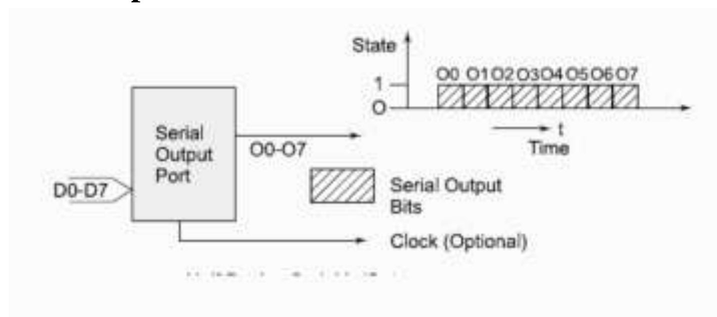
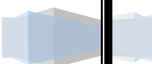


Fig shows synchronous serial output. Each bit in each byte is in synchronization with clock. If the clock period equals T, then the data transfer rate is 1/T bps.

The sender sends either the clock pulse or serial data output by encoding both the signals.



1.7.3 Asynchronous serial input:

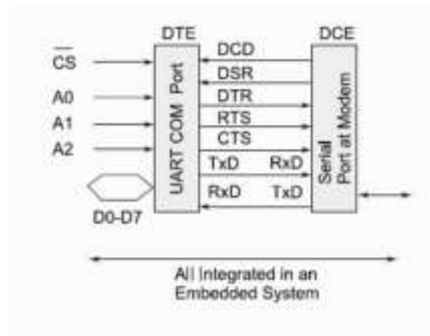


Fig shows Asynchronous serial input serial port line, denoted by RxD (received data). Each RxD bit is received at fixed interval of time but each received byte is not in synchronization. The bytes can be separated by variable intervals and phase difference.

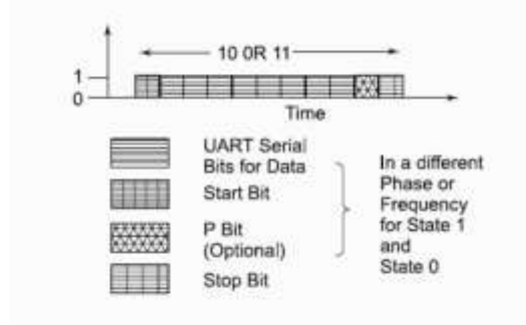


Fig shows starting point of receiving the bits for each byte. When the sender shifts after every clock period T, then a byte at the port is received at 10T or 11T.

The bit transfer rate is (1/T) baud per second but different bytes may be received at varying intervals. The sender cannot send clock pulse along with data bits.

1.7.4 Asynchronous serial output:

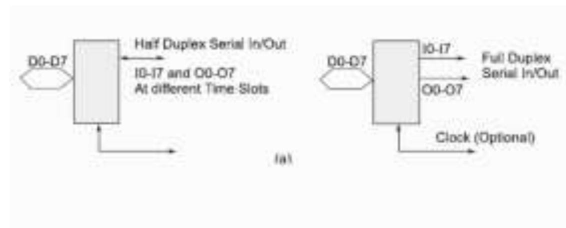


Fig shows Asynchronous serial output serial port line, denoted by TxD (transmit data). Each TxD bit is sent at fixed interval of time but each output byte is not in synchronization.

1.7.5 Synchronous, Iso-synchronous and Asynchronous communication from serial port:

Synchronous communication: When a byte (character) or frame (a collection of bytes) of data is received or transmitted at constant time intervals with uniform phase difference, the communication is called synchronous. Bits of data frame are sent in fixed maximum time interval.

Iso-synchronous is a special case when maximum time interval can be varied.

An example of synchronous serial communication is frames sent over LAN.

There are two characteristics of synchronous communication. They are as follows:

i) Bytes maintain constant phase difference. That is in synchronous there is no permission to send either bytes or frames at random time intervals. This mode does not provide handshaking during communication intervals.

ii) The clock is not always implicit to the synchronous data receiver. The transmitter generally transmits the clock rate information in the synchronous communication of data.

Synchronization ways:

1) Separate clock pulses along with the data bits
   - PISO(parallel in serial out for transmitter)
   - SIPO(serial in parallel out for receiver)
2) Data bits modulated or encoded with clock information
   - FM
   - MFM
   - QAM
   - Bi-phase
   - Manchester
3) Embedded clock information with a data frame before transmitting
   - Synchronization code bits preceding a data bit-frame
   - In-between frames signaling bits
   - Bi-sync coding

Asynchronous communication is when a byte of frames of data is transmitted or received at variable time intervals, communication is called as Asynchronous. Voice data on the line is asynchronous mode in telephone network.

There are two characteristics of Asynchronous communication. They are as follows:

i) Bytes or frames need not to maintain a constant phase difference and are asynchronous. They can be sent at variable time interval. This mode requires handshaking between transmitter and receiver.

ii) The transmitter or receiver does not send any clock signals.

1.8 Serial communication:

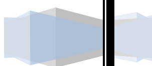
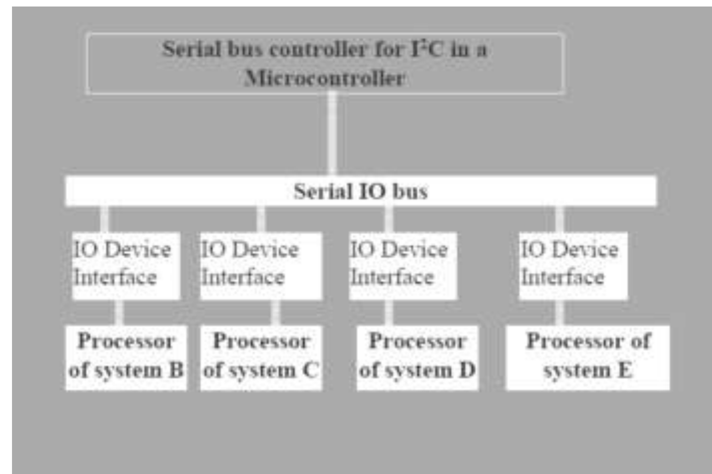


Fig shows a processor of embedded system connected to system memory bus and networked to other system through serial bus.



1.8.1 I^2C Bus:

It is a standard bus for **integrated circuit**. The I^2C bus has become the standard bus for circuit involving ICs which need to mutually network through a common bus. Examples are temperature and pressure measurement.

There are three standards. They are

i) Industrial 100kbps I^2C

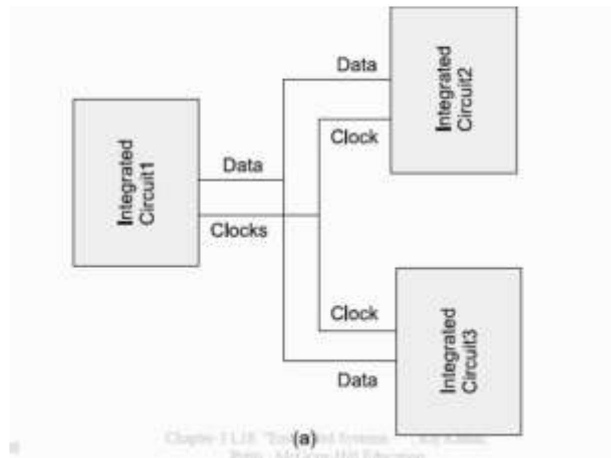
ii) 100kbps SM I^2C

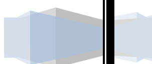
iii) 400 kbps I^2C

The I^2C bus has two lines that carry the signals. They are

i) One line is for clock signal

ii) Another line is for bi-directional data

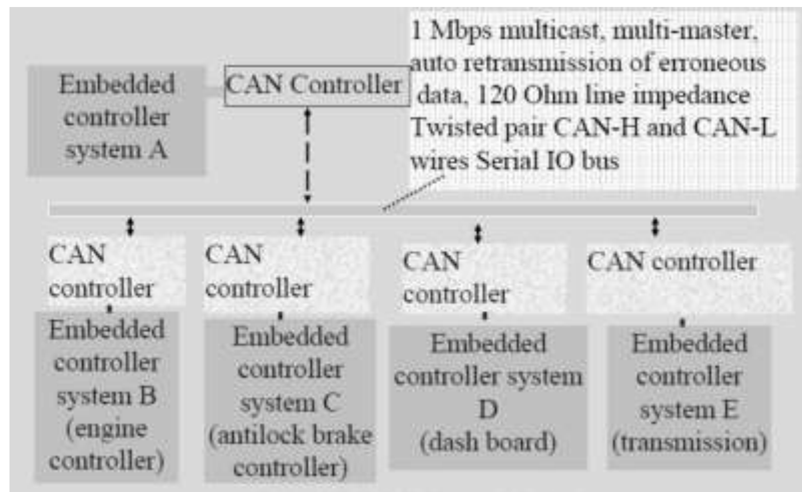




Field and its length	Explanation
First Field of 1 bit	It is start bit like an UART
Second field of 7 bits	It is called address field. It defines the slave address, which is being sent the data frame by the master
Third field of 1 control bit	It defines whether to read or write a data
Fourth field of 1 control bit	Bit defines whether the present data is acknowledgement
Fifth field of 8 bits	It is for IC device data type
Sixth field of 1 bit	It is bit for NACK(Negative Acknowledgement)
Seventh field of 1 bit	It is a stop bit

1.8.2 CAN Bus:

The CAN bus is a standard bus in **distributed network**. It is mainly used in automotive electronics. It has a serial line, which is bidirectional. It receives or sends a bit at an instance by operating at maximum rate of 1Mbps. It employs twisted pair cable connection to each node. The maximum length of cable is 40 meters.



Characteristics:

i) CAN serial line is pulled to logic level “1” by a resistor between the line and +4.5V to +12V line logic “1” in its idle state also called as “recessive state”

ii) Each node has a buffer gate between an input pin and a CAN serial line. A node gets the input at any instance from the line after sensing that instant when the line is pulled down to “0”. This state is called “Dominant state”

iii) Each node has a current driver circuit between an output pin and the serial line. A node sends the output to the line at an instance by pulling the line “0” by its driver.

iv) A node sends data bits as a data frame. Data frames always starts with “1” and always ends with seven “0”s.

v) There is an arbitration method called CSMA/AMP (Carrier Sense Multiple access with Arbitration on Message priority)

Field and Its Length	**Explanation**
First field of 12 Bits	It is called “arbitration field”. It contains the packet 11 bit destination address and RTR (remote Transmission request) bit
Second Field of 6 bits	It is called control field. The first bit is the identifier extension. The second bit is always “1”. The last 4 bits are coded for data length
Third field of 0 to 64 bits	Its length depends on the data length code in the control field
Fourth field(Third if data field has no bit present) is of 16 bits	It is the CRC (Cyclic Redundancy Check) word. The receiver node uses it to detect errors, if any, during the transmission
Fifth field of two bits	First bit is the “ACK slot”. The sender sends it as “1” and receiver sends back “0” in this slot when receiver detects an error in the reception. Sender after sensing “0” in the ACK slot generally retransmit the data. The second bit is the “ACK delimiter” bit
Sixth field of 7 bits	It is for end of the frame specification and has seven “0”s

1.8.3 USB Bus:

The Universal Serial Bus (USB) is a bus between host system and a number of interconnected peripherals.

There are two standards in USB

i) USB 1.1 (a low speed 1.5 Mbps 3 meter channel along with a high speed 12 Mbps 25 meter channel)

ii) USB 2.0 (High speed 480 Mbps 25 meter channel)

iii) wireless USB

Key features:

USB protocol has a features that a device can be attached, configured and used, reconfigured and used, share the bandwidth with other devices detached and reattached.

A device can be either bus powered or self-powered. In addition, there is power management by the software at the host for the USB ports.

The host connects to the devices or nodes using USB ports driving software and host controller.

Design Issues:

USB cable has four wires, one for +5V, two for twisted pairs and one for ground. There is termination impedance at each end as per the device speed. The electromagnetic interference (EMI) shielded cable is used for 15Mbps devices.

The data transfer is of four types

i) Controlled data transfer
ii) Bulk data transfer
iii) Interrupt driven data transfer
iv) Iso-synchronous transfer

USB is a polled bus. The host controller regularly polls the presence of a device as scheduled by software. It sends a token packet. The token consists of field for type, direction and USB device address. The device uses handshaking during transmission. A CRC field in a data permits error detection

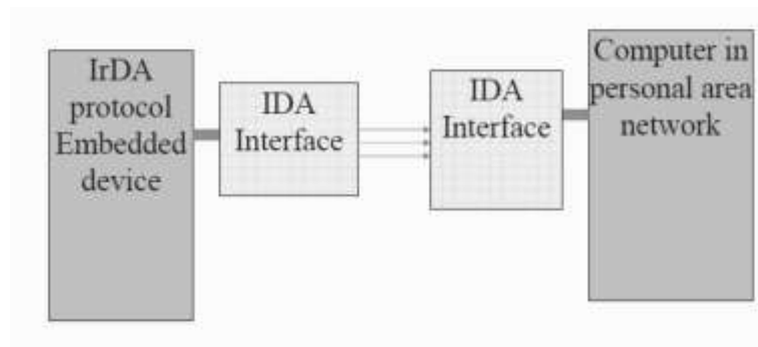
USB support three types of pipes

a) 'Steam' with no USB defined protocol. It is used when the connection is already established and the data flow starts.
b) 'Default control' for providing access.
c) 'Message' to control function of the devices.

1.9 Wireless Communication:

1.9.1 Infrared Data Association (IrDA):

Infrared (IR) is electromagnetic radiation of wavelength greater than visible red light. An infrared source consists of gallium-arsenic-phosphorous junction based diode. An infrared receiver consists of gallium-arsenic-phosphorous junction based phototransistor, which conduct electric current when the IR beam falls on it and does not conduct when IR beam does not fall on it.



IrDA supports data transfer rate of up to 4Mbps. It supports bi-directional serial communication over viewing angle between $\pm 15^0$ and distance of nearly 1m. At 5m, the IR transfer data can be up to data transfer rate of 75kbps. There should be no wall or obstruction in between the source and receiver.

IrDA supports 5 levels of communication. Level 1 is minimum level required communication. Level 2 is access based communication. Level 3 is index based communication. Level 4 is synchronized communication. Level 5 is SyncML (Synchronization Markup Language) based communication. A SyncML is used for device management and synchronization with server and client devices, which are connected by IrDA.

IrDA is used in mobile phones, digital cameras, keyboard, mouse, printers to communicate to laptop computer and for data and picture download and synchronization. IrDA is also used for control TV, air-conditioning, LCD projector, VCD devices from a distance.

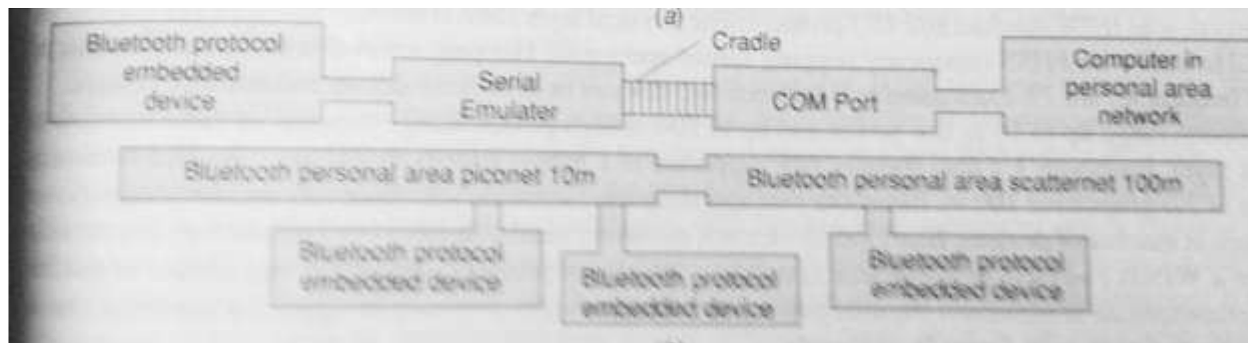
IrDA supports several protocols at three layers. Lower layer is physical layer 1.0 or 1.1. It supports data transfer rates of 9.6 kbps to 115.2 kbps and 115.2 kbps to 4Mbps I IrDA physical layer 1.0 and 1.1 respectively.

Intermediate layer is data link layer. At data link layer, it specifies IrLMP (IR management protocol) upper sub layer is HDLC communication

An IrDA upper layer protocol is tiny TP (transfer protocol). Another upper layer protocol is IrLMIAS (IR Link Management Information Access Service Protocol). A transport layer protocol during transmission specifies ways of flow control, segmentation of data and packetization. During reception, it assembles the segments and packets.

An infrared monitor in windows monitors the IR port of the IR device. It detects a nearby IR source. It controls, detects and selects the IR communication activity. An IR device on command sets up connection using IrDA, and starts the IR communication. When IR communication is inactive, the monitor enables plug and play

1.9.2 Bluetooth:



Bluetooth hardware is connected to embedded system buses and Bluetooth embeds in the system to support WPAN using Bluetooth wireless protocol. Fig shows a handheld device connected to other computers through wireless protocol using Bluetooth. A large number of CD players and mobile devices are Bluetooth enabled. Bluetooth is also used for hands free listening of Bluetooth enabled iPod or CD music player or mobile phone.

Bluetooth is an IEEE 802.15.1 protocol. The physical layer radio communicates at carrier frequency of 2.4 GHz band with FHSS (Frequency Hop Spread Spectrum). Hoping interval is 625µs and number of hopped frequencies are 79. Data transfer is between two devices or multiple devices.

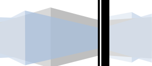
It support range up to 10m low power and up to 100m high power. Range depends on radio interface at physical layer. Bluetooth 1.x data transfer rate supported is 1Mbps. Bluetooth 2.0 has enhanced maximum data rate of 3.0 Mbps over 100m. Bluetooth protocol supports

automatic self-discovery and self-organization of network in number of devices. Bluetooth devices self discovers nearby devices (<10m) and they synchronize and form a WPAN network. Bluetooth protocol supports power control so that the devices communicate at minimum power level. This prevents drowning of signals by superimposition of high power signals with low level signals.

The physical layer has three sub layers. Radio, baseband and link manager or host controller interface. There are two types of link: best effort traffic links and real-time voice traffic links.

The host controller interface (HCI) interface is a hardware abstraction sub layer. It is used in place of the link manager sub layer.

Its communication latency is 3s. It has large protocol stack overhead of 250KB. Provision of encrypted secure communication. Self-discovery and self-organization and radio based communication between any antennas are three main features of Bluetooth.



UNIT II

Embedded Program Modeling Concepts in C: Programming in assembly language (ALP) vs High Level Language - C Program Elements:- Macros and functions, Use of Date Types, Structure, Pointers, Function Calls – Program Modeling Concepts – Program Models- DFG Models – FSM Models – Modeling of Multiprocessor Systems.

2.1 Programming in Assembly Language (ALP) Vs. High Level Language:

2.1.1 Assembly Language Programming:

Assembly language coding has the following advantages

i) The assembly language codes are sensitive to the processor, memory, ports and devices hardware. It gives **control of the processor internal devices** and **complete use of the processor** specific features in its instruction sets and addressing modes.

ii) The machine codes are compact, processor sensitive and memory sensitive. This is because the codes for **declaring the conditions, rules and data types does not exists**. The system needs **smaller memory**.

iii) Device driver codes may need only a **few assembly instructions**. For example, consider a timer device in microwave oven or an automatic washing machine or an automatic chocolate vending machine. Assembly codes for these can be **compact and are conveniently written.**

iv) We use **bottom up approach**. It is an approach in which programming is **first done for sub modules**. An example is program for software timer. Program for delay, counting, finding time intervals and many operation are written first. Then the final program is designed. The approach is **to first code the basic functional modules** and then use these to **build a bigger module**.

2.1.2 High Level language programming:

High level language coding if source files in C or C++ has great advantages. So, most of the program are in high level. They are as follows:

i) ***High level program development cycle is short even for complex systems*** because of the use of routines, standard library functions, use of modular programming approach, top down design. Application programs are structured to ensure that the software is based on hardware and network drivers.

- A **function** defines a method of operation, and sets of statements and commands are run, when the function is called.
- **Library functions** are **standard functions**, which are readily available to a programmer and the codes for them is not defined by a programmer. For example, square root can be defined by square root(), saves the time to coding.
- **Identical devices such as serial line devices** (Buses) are used in most number of embedded systems. Directly programming these functions in each system bus will

mean **repetitive and redundant coding** for each device buses. It is better to use the high level programming, which use functions. Some of the function uses only **arguments is passed to system bus** when needed and use it at the requirement.

- **Modular programming approach** is an approach in which building blocks are reusable software components. A module is built by software components. The components are built by a set of functions. Module building block may call **several functions and library functions**. A module is then tested for a requirement. It should have only one calling method. There should be return point from it. It should not affect any data other than it operates that there should be **data encapsulation**.
- **Top down design** is another programming approach in which **main program is first designed** and then its **sub modules** are designed.

ii) **High level program facilitates data type declarations:** Data type declarations provide programming very simple. For example there are four types of integers, *int*, *unsignedint*, *short* and *long*. When the required values are only positive values we can use *unsigned int.* We need signed integer, *int* (32 bits) in arithmetical calculations. An integer can also be declared as data types as *short* (16 bits) or *long* (64 bits). To operate the text and strings *char* data type is used.

iii) **High level program facilitates '*type checking*'** makes the program less error. For example *type checking* does not permit subtraction, multiplication or division operation on *char* data types. It permits '*plus*' operator to be used for concatenation when using *char* data types.

iv) **High level program facilitates use of control structures** (e.g., *while, do while break* and *for*) **and conditional statements**(e.g., *if, if else* and *switch case*) to specify the program flow by simple statements.

v) **High level program has portability** of non processor specific codes. When the hardware changes, only the modules for ISRs of the devices driver and device management, initialization and program locator modules and initial boot up record data need modifications.

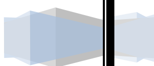
2.2 "C" program Elements

A "C" program has following structural elements

- Preprocessor declarations, definitions and statements.
- Main function.
- Functions, exceptions and ISRs

A C program has the following preprocessor structural elements

- *Include* directive for inclusion of file.
- *Definitions for preprocessor global values*
- *Definitions* of constants



- *Declarations* for global data type, type declaration and data structures, macros and functions

2.2.1 Include Directives for the inclusion of files

Any C program first include the header and source file that are readily available. Some of the examples are

- #include “vXWorks.h” ----> including VxWorks function
- #include “semLib.h” ----> including Semaphore functions library
- #include “taskLib.h” ----> including multitasking functions library
- #include “msgQlib.h” ----> including Message queue function library

Include is a preprocessor directive to include the contents (codes) of a file. Inclusion of all header file has to be done as per the requirements.

i) **Including Code files:** These are the files for the codes are readily available. For example #include ‘practHandlers.c’

ii) **Including constant data files:** These are the files for the codes and may have the extension ‘,const’

iii) **Including string data files:** These are the files for string and may have the extension ‘.string’ or ‘.str’ or ‘.txt’. For example #include ‘netDrvConfig.txt’

iv) **Including initial data files:** These are the files for initial or default data in the ROM of an embedded system. The boot up program is later copied into RAM and may have extension ‘.init’ or ‘.data’

v) **Including basic variable files:** These are the files for the local or global static variables that are stored in the RAM because they do not have initial values. It may have extension ‘.bss’.

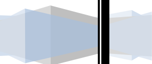
vi) **Including header files:** It is a preprocessor directive, which includes the content (codes) of a set of source files. These are the files of a specific module. A header file has an extension ‘.h’. For example #include ‘string.h’.

2.2.2 Source Files:

Source files are the program files for function of application software. The source files need to be compiled. A source file will also possess the preprocessor directivities of the application and **have first function from where the processing will start**. Its code start with void main ( ). The main calls other functions.

2.2.3 Configuration Files:

Configuration files are the files for **configuration for the system**. Device configuration codes can be put in a file of basic variable and included when needed. If these codes are in the file “SerialLine_cfg.h” then #include ‘SerialLine_cfg.h’ will be the preprocessor directives.



2.2.4 Preprocessor Directives:

Preprocessor constant, variables and inclusion of configuration files, text files and library functions are used in embedded C program

- **Preprocessor global variables:**For example, in a program the IntrDisable, IntrPortAEnable, IntrPortADisable, STAF and STAI are global variables for disabling interrupts, enabling port A, Disabling port A, status flag and status flag for interrupt respectively.
- **Preprocessor constants:**'#define false 0' is a preprocessor directive which means it assume 'false' as 0 for entire program.

2.3 Program Elements

2.3.1 Macros and Functions:

A macro is a collection of codes that is defined in a C program by a name. If a macro is defined by certain name, the compiler puts the corresponding codes for it at every place where that macro name appears. For example 'enable_maskable_intr ( )'. Here the pair of brackets are optional. When the name enable_maskable_intr appears, the compiler places the codes designed for it.

Macros differs from function:

i) The codes for a function is compiled **one time only**. On calling that function, the system has to save the content and on return have to restore the content.

ii) The codes for macro are compiles in every operation wherever that macro name is used. the**compiler puts the corresponding code** assigned to it. On using macros, the processor no need to save the content and no need to restore the value since there is**no return.**

iii) Macros are used for **short codes** only. This is because, if a function call is used instead of macro, he overheads will take a time. Overhead means content saving and new content retrieving and return. $T_{overhead}$that is the same order of magnitude as the time, $T_{exec}$is a time taken to execution of short codes within a function. We use function when $T_{overhead} << T_{exec}$ and macros when $T_{overhead} \geq T_{exec}$.

2.3.2 Use of Data Types:

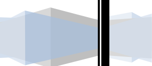
When a data is named, it will have addresses at allocated memory. The number of addressed depends on the data type.

C program allows primitive data types. The ***char*** (8 bits) for characters, ***byte*** (8 bits), ***unsigned short*** (16 bits), ***short*** (16 bits), ***unsigned int*** (32 bits), ***int*** (32 bits), ***long double*** (64 bits), ***float*** (32 bits) and ***double*** (64 bits).

The ***typedef*** is used to create a Boolean type variable in the C program.

2.3.3 Use of Pointers:

Pointers are the tools used in C programs. Pointer is a reference to a starting memory address. A pointer can refer to a variable, data structure or function. Before a pointer in C



program * symbol is used. For example unsigned char *1000 refers to a character of 8 bit at address 1000.

2.3.4 Use of Data Structures:

A data structure is organizing many data elements of same type or different types together at consecutive memory addresses. Marks of a students in different subjects are in table. The table in mark sheet shows them in an organized way.

The data structure is one of the important element in any program. Some of the important data structure are*Stack, one dimensional array, queue, circular array, a table, look up table, hash table and list.*

Stack:A stack is a special program element in a data structure.A stack means an allotted memory block data element is read from LIFO (last in first out) way. An element is popped or pushed from an addresses pointed by SP (stack pointer).

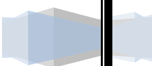
Various stack structure can be created during processing. For handling each stack one pointer is needed to point the top of the stack.

i) A call can be made for another routine during running of a routine. When the routine called is completed, the processor returns to calling function (main function), the instruction address for return must be pushed on the top of the stack.

ii) There may be at the beginning of an input data, for example, received call numbers in a phone are saved into a stack RAM in order to be retrieved in LIFO mode.

iii) An application may also create the run time stack structures. There can be *multiple data* stacks at the different memory blocks, each having a separate pointer addresses. There can be multiple stack shown as Stack 1, Stack 2.....Stack N.

iv) Each task in a multitasking should have its own stack where its content is saved. The content is saved on the processor on switching to another task. The content includes return address for PC retrieval on coming back to the task. There can be multiple stack for different memory blocks.



Array:A data structure, array is an important programming element. An array has multiple data elements, which are identified by an index. An N-dimensional array is a special data structure at the memory. It has a pointer that always points to first element of the array.

Assume one dimensional array. From the first element pointer and an index of that element starts from 0 to (array length-1) in a given dimension(length).

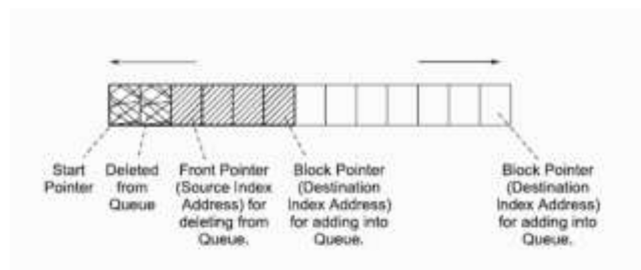
Data can be retrieved from any element addresses in the block that is allocated to the array. It can be also used for storing string data type element.

Queue: A data structure , called queue is another programming element. In array the reading of data with the help of index value and first element address. So any element can be read or write at any instance. In queue each element is read from an addresses next to the address from where the queue element was last read. This is called *deletion*.

In queue it is written to an address from where the queue element was last written. This writing is called *insertion*. A Queue means an allotted memory block from which a data is retrieved in FIFO mode.

For queue two pointers are needed. The starting memory address is represented by $*Q_{start}$ and ending memory address is represented by $*Q_{limit}$. $*Q_{tail}$ is for pointing to an address in a memory block where an element is to be inserted (added).

For a queue of integer, int $*Q_{tail}$, $*Q_{head}$ are declared. $*Q_{tail}$ initially equals to $*Q_{start}$ and it should increment on each insertion on queue tail pointer. The other $*Q_{head}$ initially equals $*Q_{start}$ and is for pointing to an address in a memory block from where the element is to be deleted. This pointer should increment on each deletion.



.

Circular queue: A queue is called circular queue when a pointer reaching a limit $*Q_{limit}$, returns to its starting value $*Q_{start}$. From circular queue the data element is retrieved in FIFO mode but there is no condition for exceeding the memory block allotted.

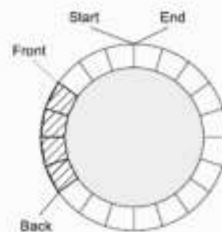
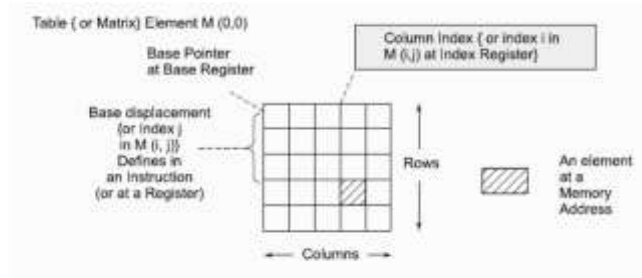


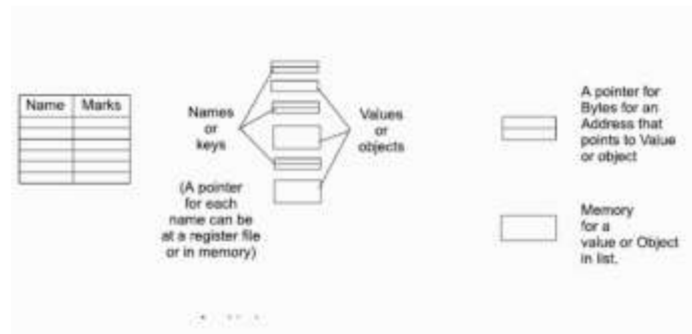
Table: A table is two-dimensional structure (matrix) and is an important data set that is allocated a memory block. There is always base pointer for a table. It points to its first element of the first

column and first row. There are two indices, one for column and the other is for row. Like array any element can be retrieved from addresses for the table base, column index and row index.

A look-up table is a two dimensional structure. It has only rows and each rows has key and on reading the key value , the data is traced.



hash table:A hash table is a data set that is a collection of pairs of key and corresponding value. A hash table has a key name in one column. The corresponding value in second column. The keys may be at non-consecutive memory addresses.

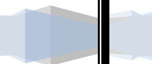


List: A list is a data structure with a number of memory block for each element. A list has a top (head) pointer for the memory address from where it starts.Each list element also stores the pointer to the next element. The last elements points to NULL.

2.3.5 Use of Loops, Infinite loops and Conditions:

Sometimes a set of instruction is repeated in a loop. Generally loops are used when executing a set of statements repeatedly. A loop starts from an initial value or condition and executes till the limiting condition is fulfilled. There are certain parameters which change each time from its initial condition to limiting condition.

```
for (i = 0; i <= 100; i++)
        {
        ----
        ----
        /* A set of statement which is repeatedly execute */
```

```
-----
----
}
```

Here the initial condition is assigned as i = 0 and the last condition for the loop to execute till i is less or equal to 100. The set of statements in the bracket executes from start to end and before return to start i is incremented by 1 (i++). The “for” statement allows set of statement to execute repeatedly 101 times with values of 0 to 100.

```
i = 0;
while (i <= 100)
{
----
----
/* A set of statement which is repeatedly execute */
----
----
i++;
}
```

The initial condition is assigned as i = 0 and is set before the *while* loop. The while loop executes till i remains less or equal to 100.i++ increments before the return to test *while* condition.The *while* statement allows set of statement to execute repeatedly 101 times with values of 0 to 100.

If the condition remains true, then *while* loop will execute infinitely. For example

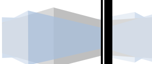
```
while(1)
{
----
----
/* A set of statement which is repeatedly execute */
-----
----
}
```

The loop will execute infinitely because “1” is always true and doesn’t come out of the loop.

Conditional statementare used when a defined condition is fulfilled, then the statements within the curly braces are executed, otherwise the program proceeds to the next set of instructions.

2.3.6 Use of function calls:

The following steps to be followed when using a function in a program



Declaring a function:just as each variable have declaration, each function must be declared by any name.

Defining the statements in function: Just as each variable has to be given the contents or value, each function must have statement. Consider the statement 'run'. These are within a pair of curly braces as follows:

```
intRTCSWT run(intindexRTCSWT, unsigned intmaxlength, SWT_Action..).
{
----
----
/* A set of statement which is repeatedly execute */
-----
----
return ( );
}
```

The last statement in a function is for the *return*and may also be for returning an element or object.

Call to a function: Consider an example

```
if ( delay_F = = true &&SWTDelayEnable = = true)
ISR_Delay (100);
```

There is a call on fulfilling a condition. The call occurs several times and can be repeatedly made.

The steps for the transfer of values from argument of calling function to called function arguments:

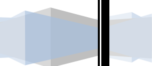
Passing the elements (elements): The values are copied from argument in calling to called function argument. When the function is executed in this way, it does not change its variable value at the calling function on return from the called function. A function use only the copied value as its own variables through the arguments.

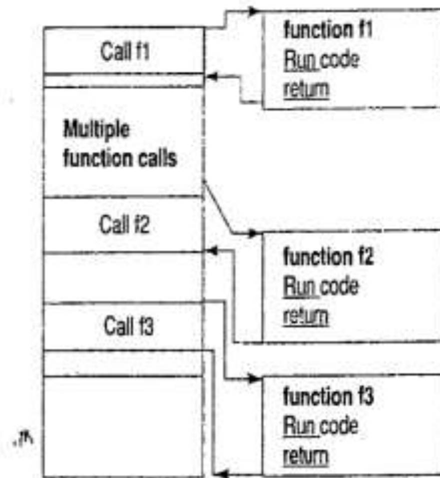
Reentrant functions call: Re-entrant function is used by several tasks and routines at the same time (synchronously). This is because all its arguments values are retrievable from a stack of the local variables or data structure.

Passing the reference:when an argument value to a function passes through a pointer, the called function can change this value. On returning from this function, the new value is available in the calling program or another program called by this function. This is because there is no saving on stack of a value that either passes through a pointer in the function arguments.

2.3.7 Multiple functions calls in cyclic order:

One of the most common methods is multiple function call made in cyclic order in an infinite loop.





2.3.8 Queuing of functions on interrupts:

When there are multiple ISRs, a high priority ISR is executed first and then lowest priority is executed. It is possible that function call and statements in any of the higher priority may block the execution of lower priority ISR and there may be deadline miss for low priority ISR. Using the function pointers in the routines and forming the queue is a solution for this problem. The queued functions are executed at later stages.

2.4 Programming Modeling Concepts:

i) *Polling for event model:* The polling in the cyclic events, state variables and signal are controlled by using switch case statements.

ii) *Sequential program model:* This model occurs when there is multiple function call within a function.

iii) *Data flow model:* Data flow graph (DFG) and Control data flow graph (CDFG) are used for modeling the data paths and program of the software. A program is modeled as handling input data stream and creating output data stream.

iv) *State machine model:* A programming model will change its state from one to another. Example for state machine model is that the key marked 5 can produce on pressing different states (0,5) ----> (1,5) ----> (1,j) ----> (1,k) ----> (1,5) and its repeats in cyclic manner.

v) *Concurrent process and inter process communication:* When there are several concurrent tasks (at the same time) and each task has sequential codes in infinite loop. A task send signal for another task. A task which gets signal runs the remaining task in the blocked state.

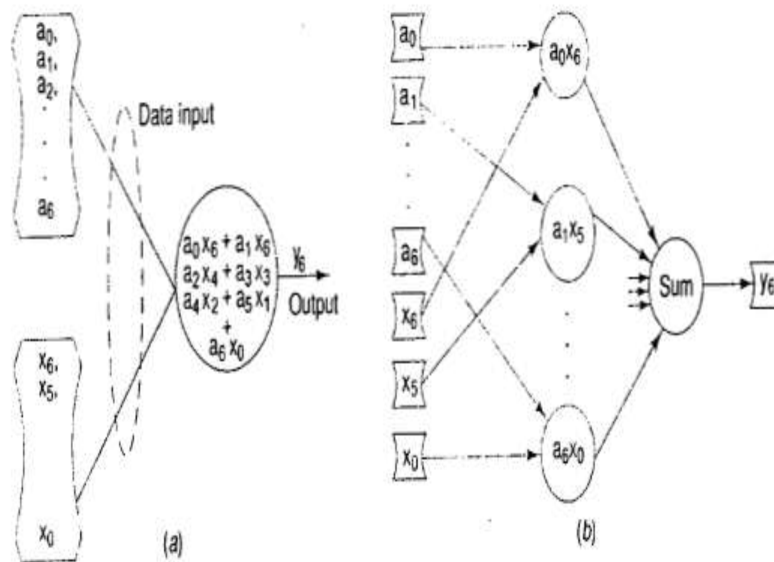
2.4.1 Data flow graph (DFG):

A data flow means that the program execution steps are determined only by the data. The data inputs are predetermined and program is done for output data. For example to find an

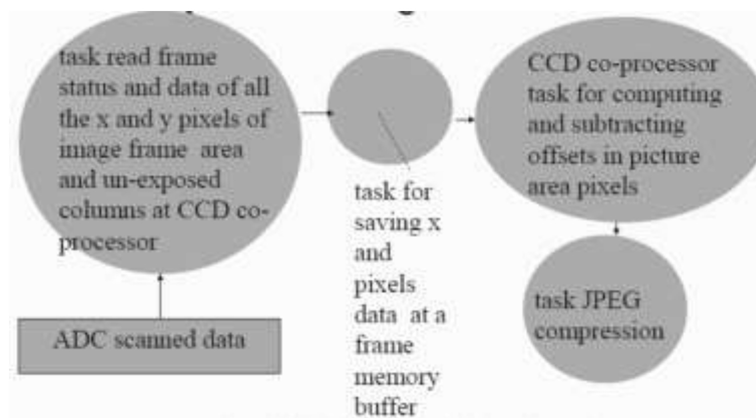
average all given marks, we will have data input as all subject marks and output data is average. Data input becomes data output after several operation in it. A **DFG does not have any conditions within it**. **The program has data entry point and data output point**.

A circle represents each process in DFG. An arrow directed **towards the circle** represents the **data input** and an arrow **away from the circle** represents **data output**. Data input along with an input edge is called token. The circle is represented as **node**.

when there is only one set of value for each input and output, then the DFG is also called as ADFG (acrylic data flow graph). All the input are instantaneously available in ADFG.



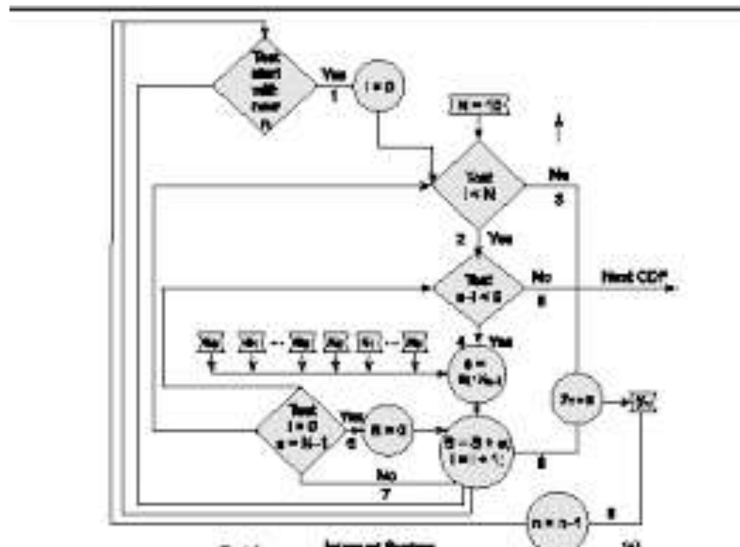
From fig. it is observed that there is no complexity in the process for output y_0. DFG model is a simple code design. A DFG models a fundamental program element that has independent path. There is no control for the program flow. The following fig. shows a DFG model for saving a picture in a digital camera.



2.4.1.1 Control DFG model (CDFG):

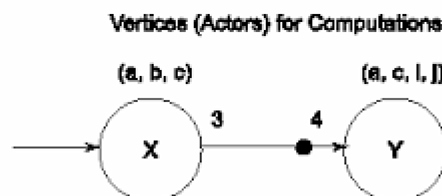
A control flow means that the program controls the execution steps and flow of the program. The control is taken **by taking decision during data operations and data flow**. It may also have loop statements for controlling. Data input will generate data output only after a control data flow by conditions. The output depends on the control statements in the program.

A circle represents each process in DFG. An arrow directed **towards the circle** represents the **data input** and an arrow **away from the circle** represents **data output**. Data input along with an input edge is called token. The circle is represented as **node**. A box (either square or rectangle with diagonal axes vertical) represents the condition. The **directed arrow from the box** determines the condition is **true**.



2.4.1.2 Synchronous data flow graph (SDFG):

When there are number of token (inputs along with the edge) required for processing to generate more token (outputs along with the edge) in a single instance. the data flow is called SDFG. Let an arc represent a buffer in memory. The arc contains one or more input data **with delays**. Vertices will perform the computations. An edge between vertices(arc with an arrow for direction) represents queue of output values form vertex and queue of input values to another vertex.



Let X and Y be the set of instructions that started once and X generates the output values a, b and c. Let Y get the input values a, c, i and j; and let i has a delay. The number of inputs to Y

need not equal to the number of output from X. Y gets additional inputs and it does not need all the output from X. The i with **delay** is represented by a **dot.** Here i and j are initial token for vertex Y. All the computations are scheduled in ADFG at each vertices. Then SDFG translate model program into a sequential model program.

2.4.2 State machine programming model for event controlled program flow:

A state machine model is for if there are state and state transition function, which produce a new state. A state transition function changes a state to its next state.

In a state machine model the input of the program that changes into new state and generates the output, which may be input for next state.

2.4.2.1 Finite state machine (FSM) model:

In FSM model there are finite number of possible states in a system. Fig 1 shows the states are modeled as FSM of a timer as there are finite number of states. Fig 2 shows how the state of the task can be modeled as a FSM. Fig 3 shows state in a program model for ACVM (Automatic Chocolate Vending Machine). There can be transition of the present state to next state, which depends on the transition functions.

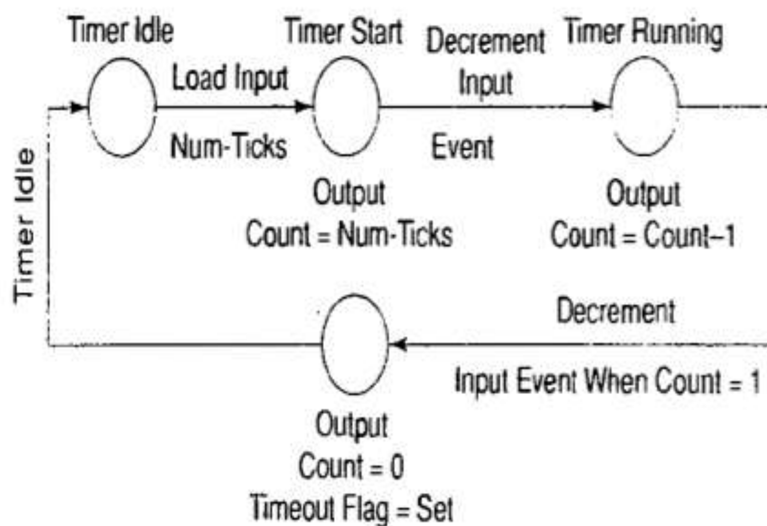


Fig. 1

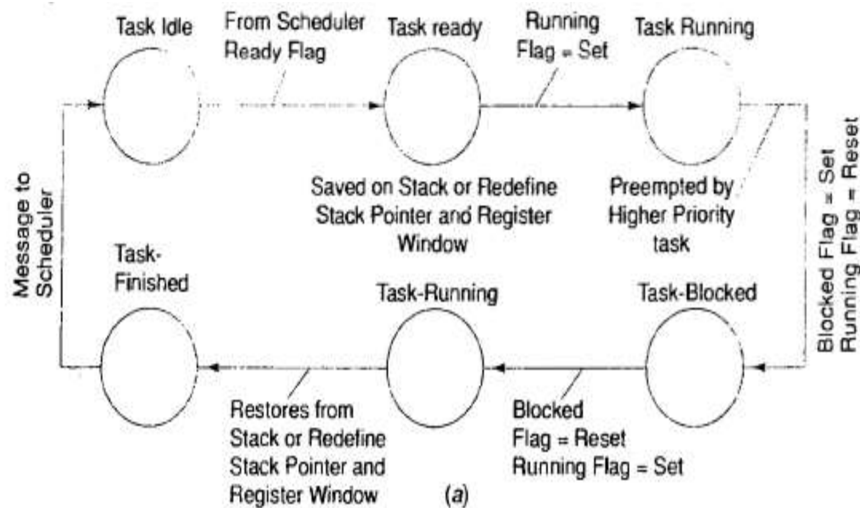


Fig.2

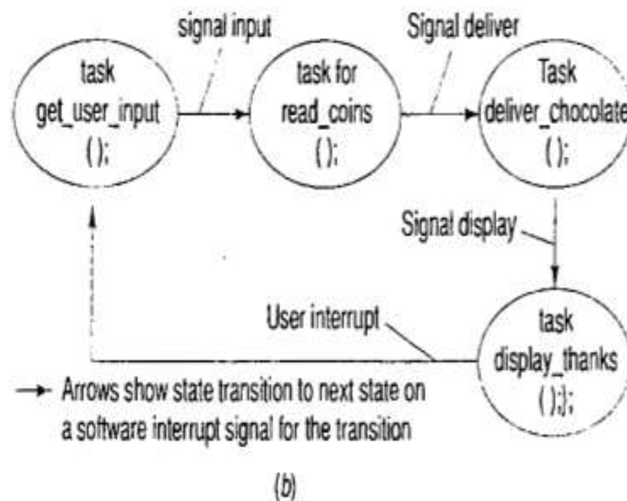


Fig. 3

While modeling a process in FSM, the designer specifies the following process for each state:

i) The state with **finite numbers**.
ii) **Finite set of inputs** and their values for the state.
iii) **Finite actions** (computation) during the state and **finite set of output** with their values and **output functions** are determined.
iv) **State transition function** for present state to next state.

The steps to model the states and interstate transition in FSM modeling are as follows:

i) A transition to a new state should be from previous state events. The event may be setting a value of the parameters.
ii) A state can receive multiple input from other state. An event input may be asynchronous (event input may occur at any instance of time).

iii) A state can generate multiple output. An output variables identifies the next stateon mapping the inputs and previous state.

2.4.2.2 FSM State Table:

To design a software using FSM model, a state table can be designed to representing every state in its rows. The following columns are made for such table:

i) **Present state name** or identification.
ii) Action of the state until some events.
iii) The event that cause the execution of the **state transition function**.
iv) **Output** from the output state functions.
v) Next state
vi) **Expected time interval** for finishing the transition to a new state after each event.

2.4.3 Modeling of Multiprocessing Systems:

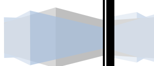
2.4.3.1 Multiprocessor systems:

A large complex program can be partitioned into the task of instruction and ISRs. The task and ISRs runs concurrently on different processor and they can communicate with each other finally.

Partitioning the program between the various processor is very important. The problems in modeling the processing of instruction in multi processor system are as follows:

i) Partitioning of processes, instruction sets and instructions.
ii) Concurrent processing of processes on each processor.
iii) Static scheduling by the compiler in a superscalar processor.(Each superscalar processor has multiple processing unit in parallel).
iv) When superscalar unit are present in parallel, then two or more pipelines of instructions are executed in parallel.
v) hardware scheduling issue. For example whether the chosen hardware will support the scheduling algorithm.
vi) Static scheduling issue.(The performance should not be affected when the processing actions are predictable)
vii) Synchronizing issues, it means inter-processor communication should be in definite order.
viii) Dynamic scheduling issues, For example the performance is affected when there are interrupts and the tasks are asynchronous.
ix) Scheduling of instructions, SIMDs (Single instruction multiple data), MIMDs (Multiple instruction multiple data) and VLIWs (Very large instruction words) and scheduling them for each processor.

Consider two processor PA and PB, interfaces with a memory in a system.



Case 1: Processor share the **same address space** through a common bus. It is called **tightly coupled** between processor.
Case 2: Processor have **different addresses space** and shared data set. It is called **loosely coupled** between processor.
Fig. 6.12 (a) and Fig. 6.12(b) shows both the cases.

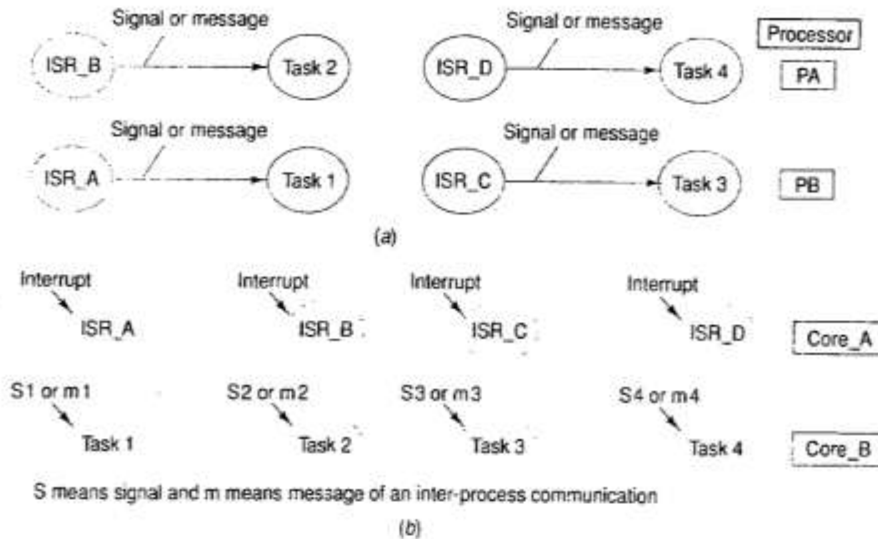


Fig. 6.11 (*a*) Static scheduling of tasks and interrupt service routines on two processors (*b*) Static scheduling on two processor cores

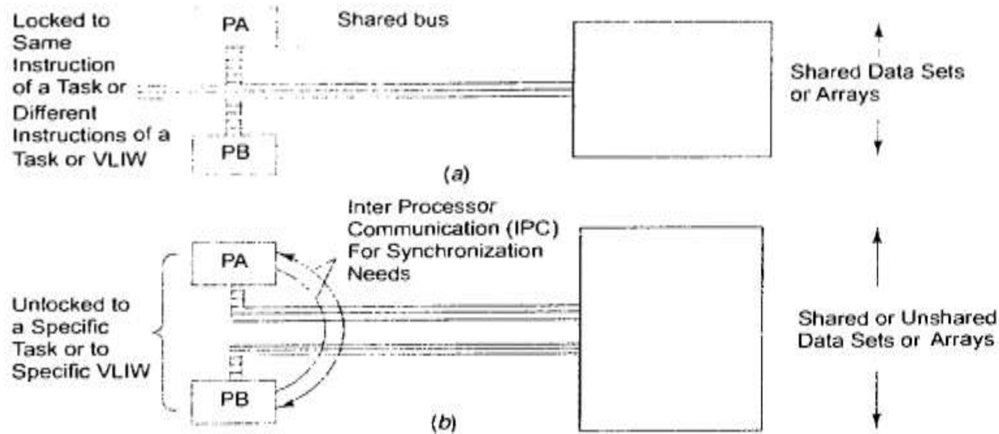
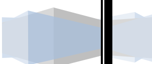


Fig. 6.12 (*a*) Tightly coupled processors sharing the same address space while processing multiple tasks (*b*) Loosely coupled processors having separate autonomous address spaces as in a network as well as shared address space for data sets and arrays



2.4.3.2 Model of Unfolding SDFGs into Homogeneous SDFGs:

An SDFG model delays the number of input and outputs. When there is only one data at the input and one at the output then an SDFG is called homogeneous SDFG (HSDFG). Fig. *** shows modeling of SDFG. Fig 6.13 shows an HSDFG representation after unfolding the SDFG. The dot shows that there is a delay.

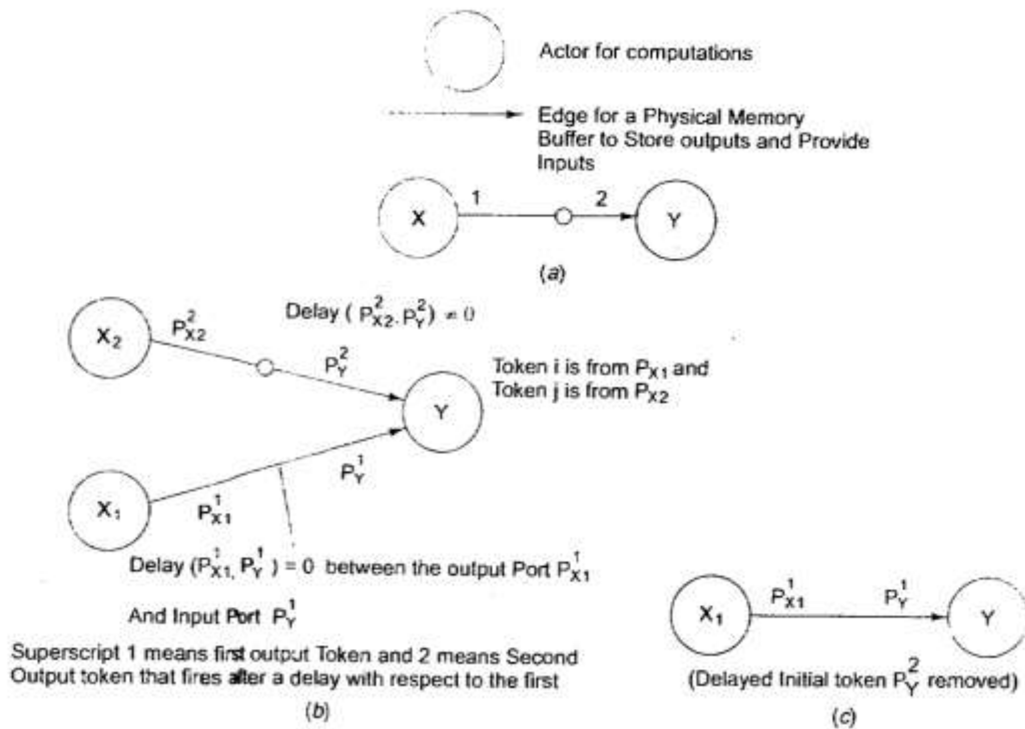


Fig. 5.13 (*a*) A modeling of computations by an SDFG. The dot and label over the edge show delayed two number input tokens at vertex Y (*b*) A homogeneous SDFG representation after unfolding the SDFG (*c*) An APEG representation from an HSDFG after removing the delayed edge

For example the output of vertex X is ‘a’ and input to Y is also ‘a’. Then an SDFG can be unfold into HSDFG. An SDFG can be unfolded into one or more HSDFGs. Two vertices can be connected by one or more vertices in HSDFG. An HSDFG usually has more vertices than SDFG.

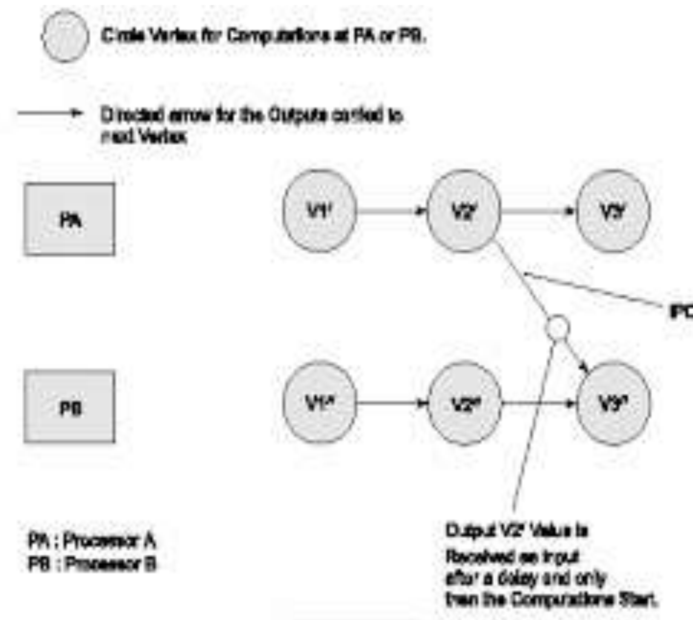
All computations are static scheduled in HSDFG vertices. Let there be a sequence of computations that are fired at the vertices. Let precedence in a directed graph defines a computation order which vertices are placed first. then next to next. Input from another processor can be delayed. A SDF model program then translates into number of concurrent process using HSDFGs.

2.4.3.3 Model of Unfolding HSDFG into APEGs:

It is a precedence of vertices in a directed graph such that there is no delays in the vertices. If initial delays are removed from HSDFG then Acrylic Precedence Expansion Graph (APEG) is obtained.

APEGs are important for scheduling in multiprocessor systems. An APEG has starting input identical to the output from a previous vertex and also delaying vertices. An APEG algorithm is to schedule that the precedence problem in algorithm remains same in original state. APEG graph has no delays. It drives from SDFG or HSDFG.

A task level concurrent processes in IPC (inter process communication) can be modeled using APEG. A thread(delay) running on one processor modeled as APEG can pass a control to another processor by blocking itself.



2.4.3.4 Applications of Graphs to Multiprocessor systems: Partitioning and scheduling

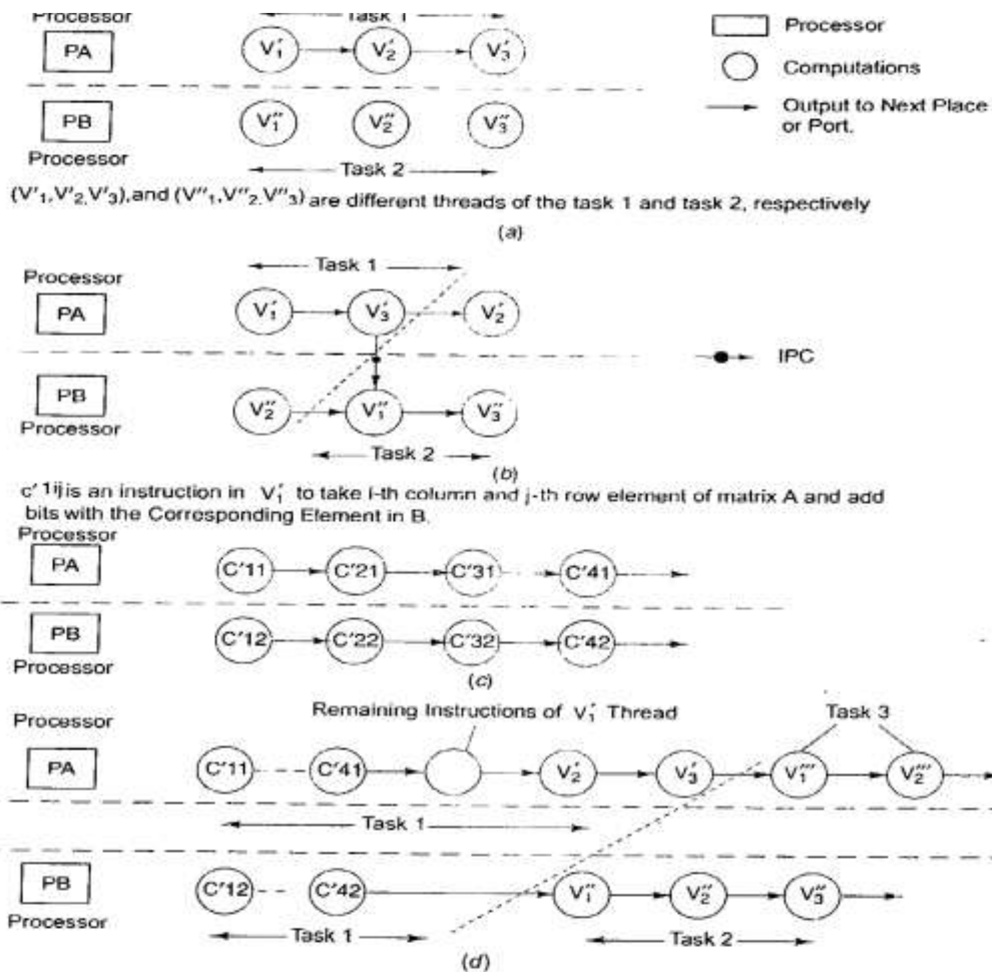
When there are multiple processor in parallel, then the partitioning is done as follows:

i) There are minimum number of IPCs so that total number of IPC delays can be minimized.

ii) There is a load balancing. Each processor has the least waiting time by sharing the processor load.

iii) The performance cost minimizes.

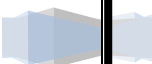
Consider Fig.6.15 . At each vertices computations occur such that the delay problem maintained. The graph of a program thus partitioned into the function or task. The following strategies can schedule a program for running.



g. 6.15 (*a*) Each task or function is executed on an assigned processor (*b*) Each task or function is executed on different processors at different periods (*c*) Instructions of four different tasks are partitioned on two processors (*d*) Instructions of four different tasks are partitioned and scheduled on two processors differently during different periods

i) Each task is executed on assigned processor only. Instructions of four different task are partitioned on two or more processor. These are scheduled in different periods.

ii) Each set of data is partitioned in VLIW instruction and is executed on the different processors, which execute a same program. Partitioning is preferred using VLIW for matrix addition process.



UNIT III

Real Time Operating Systems: Real Time Systems – Issues in Real Time Computing – Structure of a real time system – Process – Task – Threads – Classification of Tasks – Task Periodicity – Periodic Tasks- Sporadic Tasks – Aperiodic Tasks – Task Scheduling – Classification of Scheduling Algorithms – Event Driven Scheduling – Rate monotonic scheduling – Earliest deadline first scheduling. Inter Process Communication:- Shared data problem, Use of Semaphore(s), Priority Inversion Problem and Deadlock Situations - Evaluating operating system performance – Power optimization strategies for processes.

3.7 Classification of task:

Based on their time task can be classified into three categories. These are

- Hard real time task
- Firm real time task
- Soft real time task

3.7.1 Hard real time task:

Hard real time task are those which produce their result within a time limit or deadline. Missing such a deadline causes the **task has failed**.

An example is a circuit contains several motors along with sensors and actuators. The sensor senses various condition and issues command to the actuators. It is necessary that the controlled is within a time limit. The inputs may be from fire-sensor, power sensor etc. A delay in detection and reaction process may result in failure of result.

A very important feature of hard real time system is criticality. Many of the hard real time systems are safety-critical. The medical instruments monitoring and controlling the health of the patient is another example. In hard real time systems we have to determine the dead line and the task should complete within a deadline.

3.7.2 Firm real time task:

A firm real time task also has its own deadline time. If the task failed to complete within its deadline then the system **does not fail altogether**. Only **some of the result will get affected**.

.

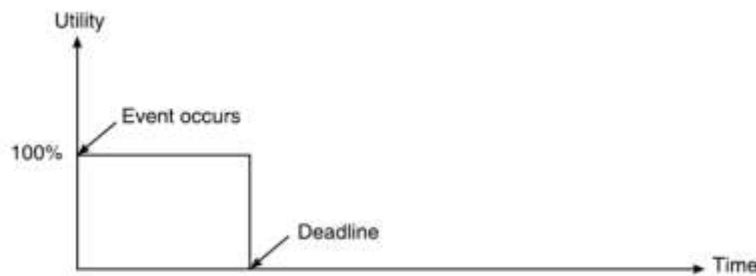


Fig.3.1 Utility result for firm real time task

For example in **video conferencing**, video frames are sent over a network. Depending upon the quality of the network some frames may arrive late or it gets lost. The overall effect is degradation quality in the particular video.

The main feature is that any result computed after the deadline is no value and it is discarded. As shown in Fig. 3.1 after the event has occurred, the utility of response is 100% if it occurs within the deadline. Beyond the deadline, the utility becomes zero and the result is simply discarded.

3.7.3 Soft real time task:

In soft real time task similarly there is a deadline. The task is expected to complete within a deadline. If the task does not complete within the deadline, still the **system runs without any failure**. Late arrival of results does not force a total discarding of task. But as time passes the utility of the result drops as shown in fig. 3.2

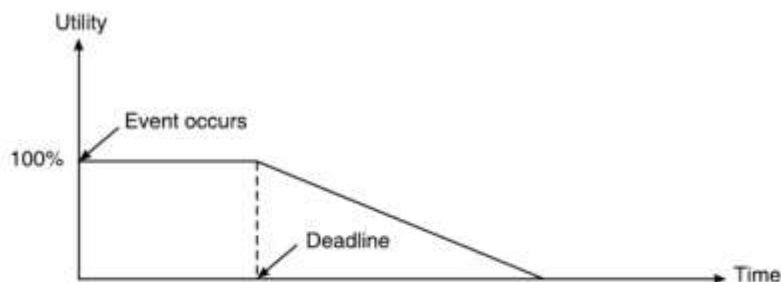


Fig. 3.2 Utility result of soft real time task.

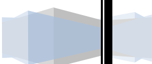
A typical example for soft real time task is the **railway reservation system**. It is expected that the average time taken to process the request is small, and particular request takes high time then still the system is working.

Another example is **web-browsing**. After typing URL, we expect the page to arrive soon. But it will take longer time sometimes.

3.8 Task periodicity:

An embedded system is generally designed to perform a specific application. It normally consists of small set of tasks. For example, a monitoring system continuously monitors its environment and takes appropriate actions based on inputs. Such a system will generally consist of set of task to be performed at regular interval of time. Based on this periodicity property, tasks are classified into three categories namely

- Periodic task
- Sporadic task
- Aperiodic task



3.8.1 Periodic Tasks:

A periodic task **repeats itself regularly** after a certain fixed interval of time. Such a task T_i can be characterized by four factors;φ_i represents the phase, d_i represents the deadline, $e_i$represents the execution time and p_i represents the periodicity.

The phase (φ_i) identifies the first time instant at which the task T_i occurs. For a particular occurrence of the task, the deadline d_i is the time by which the task must be completed. The actual time required to execute the task in worst case is e_i, which is smaller than d_i.Finally p_i identifies the periodicity of the task.

Thus the task T_i can be represented by four factors <φ_i, d_i, e_i, p_i>. The periodic task is represented as shown in Fig. 3.3

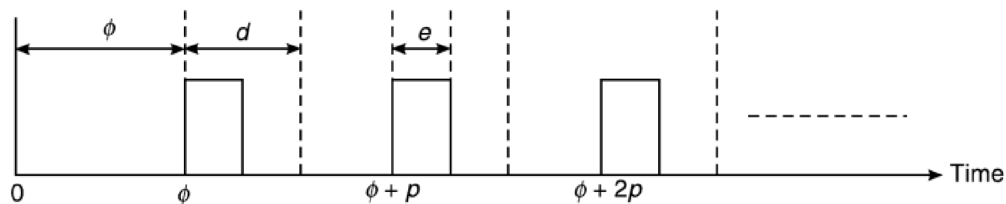


Fig. 3.3 Periodic task

Most of the tasks in real time embedded systems are periodic in nature.Sometimes the deadline d_i is taken to be same as periodicity p_i.

3.8.2 Sporadic Tasks:

A sporadic task can occur at any time instant. However, **after the occurrence of first instance**, there is minimum separation after another occurrence of task. Thus a sporadic task can be represented by three factors < d_i, e_i, g_i>. Where e_i is the worst case time, g_i denotes minimum separation between two consecutive task and d_i represents deadline.

Examples of sporadic tasks are the interrupts which may be generated from different conditions. Fig. 3.4 shows a sporadic task behavior.

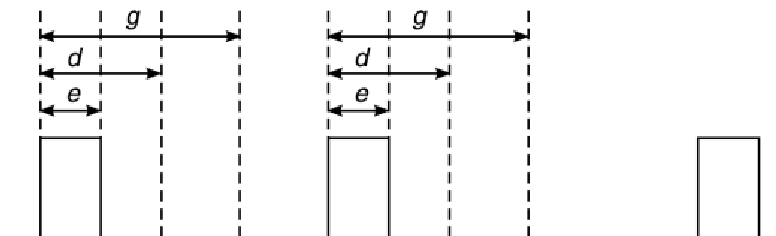


Fig. 3.4 Sporadic task behavior.

3.8.3 Aperiodic Tasks:

Aperiodic tasks are similar to the sporadic task only in both type of task can arrive at random time interval. In aperiodic task there is no guarantee that another instance of **task will**

not arrive before minimum amount of time. The **deadline of this task is random** and expressed as average value or expected value.

An example of aperiodic task is railway reservation systems.

3.9 Task Scheduling:

Task scheduling is to identify the order in which the task is to be executing in a system. Since most of the task in a real time embedded systems re periodic in nature, the real time task scheduling algorithm mostly concentrate on periodic tasks. Sporadic and aperiodic tasks are handled when they occur, without disturbing deadlines of the already scheduled tasks.

The quality of a schedule is identified by a term called processor utilization. The processor utilization of a task is defined to be the fraction of time for which the processor is used by it. Thus if the execution of task is e_i and periodicity be p_i then the utilization u_i= ─.

The overall utilization is given by $U = \sum$ ─ .

3.10 Classification of Scheduling Algorithms:

Based upon scheduling points, the algorithm is classified into following categories:

- Clock driven scheduling.
- Event driven scheduling.
- Hybrid scheduling.

In a clock driven scheduling, the scheduling points are the interrupts received from a periodic clock. There are two types of clock driven schedulers:

- Table driven.
- Cyclic.

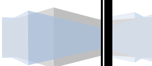
In event driven schedulers, it responds to different event in a system. They perform scheduling at a instance when a task is finished executing. There are three types of event driven schedulers:

- Simple priority scheduling (or) Foreground background scheduling.
- Rate monotonic scheduling (RMS)
- Earliest deadline first (EDF) scheduling

3.11 Clock Driven Scheduling:

A clock driven scheduler works in synchronism with clock signal. The timer periodically generates interrupts. On receiving an interrupt, the scheduler is processed and then takes decision about which process to be scheduled next. Since the set of task and their periodicity values, execution times and deadline are known already. Then it is possible to schedule the process. But, the major drawback of this type is it is inability to handle aperiodic and sporadic task.

3.11.1 Table Driven Scheduling:



A table driven scheduler uses pre-computed table that stores the task to be run at different clock intervals. For example, consider a set of tasks T = {T_1, T_2, T_3} with the associated parameter as shown in table 3.1.

Table 3.1 Example task set with parameters

Task id (T_i)	Execution time (e_i)	Periodicity (p_i)
T_1	2	6
T_2	1	3
T_3	4	12

A possible schedule for this set of tasks can be shown in table 3.2. It is assume that at time instant 0, all three tasks have arrived. Next instance of time T_2 will arrive at time instant 3. It is easy to see that the arrival pattern of task instance will repeat itself from 12th instant of time, the LCM of the periodicity of individual tasks.From Table 3.2.It is sufficient to store the number of entries equal to LCM of periods of the task. This LCM determining the size of the schedule table is called **major cycle.**

Time instant	Task arrived	Task scheduled
0	T_1, T_2, T_3	T_1
1		T_1
2		T_2
3	T_2	T_3
4		T_3
5		T_3
6	T_1, T_2	T_3
7		T_2
8		T_1
9	T_2	T_1
10		T_2
11		T_1
12	T_1, T_2, T_3	T_1
. . . .	. . . .	. . . .

3.11.2 Cyclic Scheduling:

A major problem in table driven scheduling is the size of the schedule table. If the LCM of periods or execution time is a large number, we need to have so many slots in the table. In

cyclic scheduling, the major cycle (Equal to LCM of periodicity of tasks) is divided into a number of equal sized minor cycles or frames.

One or more frames are allocated to individual tasks.The condition is shown in Fig. 3.5 in which major cycle has been divided into four equal sized frames, F_1, F_2, F_3, F_4. Now three tasks T_1, T_2 and T_3 have been allotted to various frames within a major cycle.

The task T_1 has been assigned by two frames F_1 and F_3. The tasks T_2 and T_3 have been assigned to frames F_2 and F_4 respectively. The schedule table stores the task to be run for different frames, thus the size of the table is reduced to be equal to the number of frames.

The frames size is chosen such that if large frame size is chosen then lesser number of frames in a major cycle. This may leads to wastage of CPU time. The limitations behind selection of a frame size are the following:

i. **Minimize context switching:**The frame size should be such that even the largest task can complete within a frame.Otherwise multiple frames need to be allocated to a task. Since the scheduler needs to context switching at each and every boundary of a frames. These times are wasted.

ii. **Schedule table minimization:**Since the schedule table holds information for each of the frames, a large frame size is advisable. It means lesser number of frames. Frame size should be such that the major cycle is multiple of frames. Otherwise storing information for one major cycle is not enough for scheduling.

iii. **Satisfaction of task deadlines:** This is one of the important issues in determining the frame size. Suppose if a task arrives after the start of the frame, the task deadline may be very close, resulting in missing the deadline for the task. The situation is shown in Fig 3.6 in which an instance of a task has arrived Δt times later than the beginning of frame k. It can be scheduled earliest in frame (k+1). If the execution time of the task is more than the time difference between the start time of (K+1)th frame and task deadline d, the task will miss its deadline.

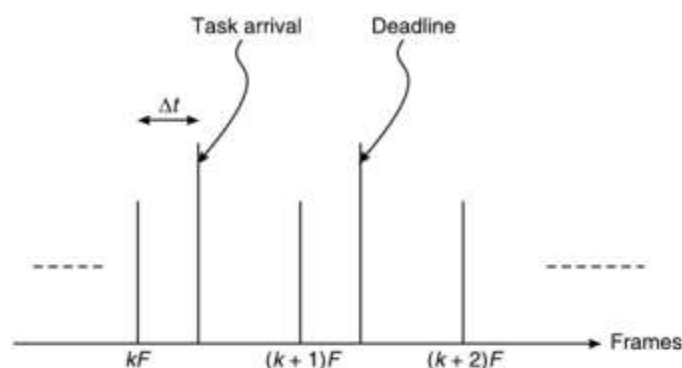


Fig. 3.6.Possibility of missing deadline.

To solve this problem, it is required that there are atleast one complete frame between the arrival of a task and its deadline. The condition is shown in Fig. 3.7. That is,

$$2F - \Delta t \leq d_i.$$

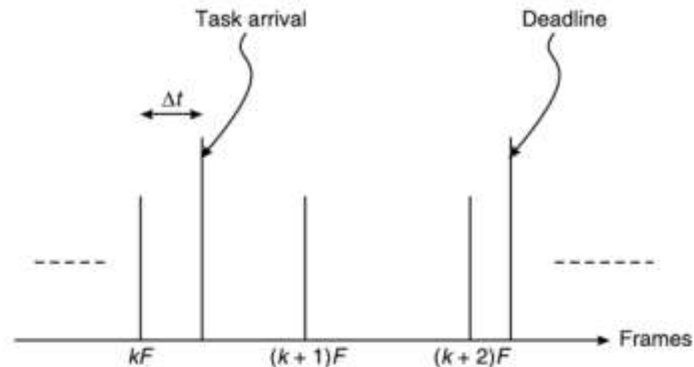


Fig. 3.7 Existence of full frame.

To define the dondition further, it can be shown that the minimum value of Δt is equal to GCD(F,p_i), P_i being the periodicity of the task.

$$\begin{aligned} 2F - GCD(F, p_i) &\leq d_i \\ \Rightarrow \quad F &\leq 0.5(d_i + GCD(F, p_i)) \end{aligned}$$

3.12 Event Driven Scheduling:

A basic problem with clock driven scheduling is their inability to handle a large number of tasks. Determining frame size is difficult. Other tasks such as sporadic task and aperiodic task cannot be handled efficiently. So event driven scheduling is chosen. The priority may be static or dynamic. There are three important schedulers.

- Foreground background scheduling
- Rate monotonic scheduling
- Earliest deadline first scheduling.

3.12.1 Foreground background scheduling:

This is the simplest possible priority driven scheduling strategy. In this the periodic real time task have higher priority than sporadic task and aperiodic tasks. The periodic task runs in foreground. A background task can run only when no foreground tasks are running.

3.12.2 Rate monotonic scheduling:

Rate monotonic scheduling (RMS) is a static-priority-based event-driven scheduling algorithm for periodic tasks. The priorities are assigned to the tasks based on their periodicity values. A task with lower periodicity value (that is, more number of occurrences within a fixed time interval) is assigned higher priority. Just like other preemptive priority-based schemes, arrival of a higher priority task will force preemption of any running lower priority task.

The theory underlying RMS is the *rate monotonic analysis* (RMA), that uses simple model of the system stated as follows:

1. All processes run on a single CPU, thus there is no task parallelism.
2. Context switching time ignored.
3. Execution time for different invocations of a task are same and constant.
4. Tasks are totally independent of each other.
5. The deadline of an instance of a task occurs at the end of its period.

The theory has been stated in the form of a theorem as follows:

Given a set of periodic tasks to be scheduled using a preemptive priority scheduling, assigning priorities, such that the tasks with shorter periods have higher priorities, yields an optimal scheduling algorithm.

It may be noted that the term *rate monotonic* refers to the fact that monotonically higher priorities are assigned to tasks having higher rates of occurrences. The optimality criteria states that if a schedule meeting all the deadlines exists with fixed priorities, then the rate-monotonic approach will definitely identify a feasible schedule.

Let us consider an example to understand the RMS principle. Table 6.3 shows a set of three tasks along with their execution times (e_i) and periodicities (p_i). The task $P1$ has the smallest period, thus it has the highest priority followed by the tasks $P2$ and $P3$.

Table 6.3 Example of RMS

Task	*Execution time* (e_i)	*Period* (p_i)
$P1$	1	3
$P2$	2	5
$P3$	3	15

shown in Fig. 6.8. At time instant 0, since $P1$ is the highest priority task, it gets scheduled. Once it is over at time instant 1, $P2$ gets scheduled. It executes for full 2 time units to finish off. At time instant 3, another instance of $P1$ has arrived. Though $P3$ is waiting, it does not get a chance to execute as $P1$ is of higher priority. At time instant 4, none of $P1$ and $P2$ is pending. Thus, $P3$ is scheduled for a single time unit. At time instant 5, another instance of $P2$ arrives and preempts the running lower priority task $P3$. The process continues and at time instant 14, no task is ready for execution. Thus, the CPU remains idle. Hence, this set of tasks can be scheduled in RMS principle.

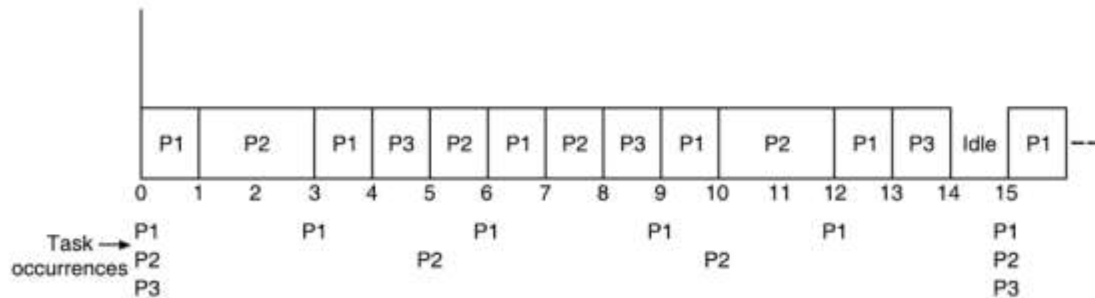


Fig. 6.8 RMS scheduling.

A complete schedule for this set of tasks has been shown in Fig. 6.8. At time instant 0, it is assumed that all the three tasks $P1, P2$ and $P3$ have arrived. Further instances of $P1$ will come at time instants 3, 6, 9, 12, etc. Similarly, occurrences of instances of other tasks are also

Next, we concentrate on a set of results that answers the question that whether a given set of periodic tasks can be scheduled using the RMS technique and ensure that none of them will miss their deadlines.

1. *Necessary condition:* For a set of periodic tasks to be scheduled using RMS principle, the sum of CPU utilizations of individual tasks be less than or equal to 1. That is,

$$\sum_{i=1}^{n}(e_i/p_i) = \sum_{i=1}^{n} u_i \leq 1$$

where, e_i is the worst case execution time of the task and p_i is its periodicity. For our example, considered previously,

$$\frac{1}{3} + \frac{2}{5} + \frac{3}{15} = 0.93 \leq 1$$

Thus, the tasks satisfy the necessary condition.

2. *Sufficient condition: Liu* and *Leland* derived a sufficiency condition to ensure that a set of tasks is schedulable using RMS. It says that a set of n real-time periodic tasks is schedulable if,

$$\sum_{i=1}^{n} u_i \leq n(2^{1/n} - 1)$$

where, u_i is the CPU utilization of task i as calculated earlier. Setting the value of $n = 3$, we get that for 3 tasks to be RMS schedulable, total utilization be $\leq 3(2^{1/3} - 1)$, that is, 0.78. However, for the set of tasks in our example, the total CPU utilization is $0.93 > 0.78$. Thus, our example fails the sufficiency test, though it is still RMS schedulable.

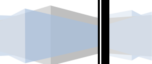
Another test, known as *Lehoczky test* has been introduced to check whether a set of tasks is RMS schedulable or not. It can be stated as follows:

A set of periodic real-time tasks are schedulable using RMS technique under any task phasing if all the tasks meet their respective first deadlines under zero phasing.

A formal proof of the above statement is beyond the scope of this book. Interested readers may refer to books on *Operating Systems*. However, an intuitive justification can be achieved by considering a definition of *critical instant*. For a task P_i, its critical instant is the point at which an instance of the task arrives but has to wait for the longest period of time, as all other higher priority tasks are also ready at that point. This low priority task can be scheduled only at a time when no other higher priority tasks are pending. It is obvious that more number of instances of higher priority tasks will occur during a period of lower priority task, when these tasks are in-phase with each other, rather than being out of phase. Hence, if with zero phasing a task meets its deadline, it will meet all its deadlines with non-zero phasings also.

Let a set of tasks $P_1, P_2, \ldots, P_n$ with corresponding execution times $e_1, e_2, \ldots, e_n$, and periodicity values $p_1, p_2, \ldots, p_n$ be given. We have to check whether they pass the *Lehoczky test* or not. Without any loss of generality, we assume that the tasks have been ordered in their decreasing order of priority. That is, P_1 is of highest priority, while P_n has the lowest priority. Now, the task P_i meets its first deadline only if,

$$e_i + \sum_{k=1}^{i-1}(p_i/p_k) \times e_k \leq p_i$$



This is because within the period p_i of task P_i, a higher priority task P_k can appear (p_i/p_k) times. All these instances are to be scheduled before the first instance of task P_i gets scheduled. Thus, checking for all the tasks in the system in this manner, if all of them can be scheduled before their first deadlines, the set is schedulable using RMS principle.

In our previous example, for task P_1, $e_1 = 1 \leq p_1 = 3$.

For task P_2,

$$e_2 + (p_2/p_1) \times e_1 = 2 + (5/3) \times 1 = 2.6 \leq p_2 = 5.$$

For task P_3,

$$e_3 + (p_3/p_1) \times e_1 + (p_3/p_2) \times e_2 = 3 + (15/3) \times 1 + (15/5) \times 2 = 14 \leq p_3 = 15.$$

Thus, all tasks pass the test. Hence, the set of tasks is schedulable using RMS technique.

It should also be noted that a set of tasks, if not satisfying the *Lehoczky's test*, may still be schedulable using RMS policy. This can happen because the test takes all tasks to have zero phasing. In reality, if phases of tasks are different, there may be some free time for the CPU in which it can be assigned some task so that the deadlines are met.

The disadvantages of using RMS are the following:

1. It is difficult to handle aperiodic and sporadic tasks. This happens because their periodicity values are not known, thus no priority assignment can be done for them.
2. RMS does not produce optimal results if the deadlines differ from periodicity of tasks. In this direction, there exists another algorithm, called *Deadline Monotonic Algorithm* (DMA). In DMA, priorities are assigned to the processes based on their deadlines. Tasks with shorter deadlines will have higher priorities. Naturally, if deadlines are equal to periodicity values or are proportional to those, RMS and DMA yield same results. However, if deadlines are different from periodicity, there may be cases in which RMS fails, but DMA can produce valid schedules. On the contrary, whereever DMA fails, RMA also fails.

Context switching overhead

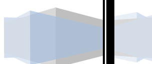
In our discussion about the RMS policy, so far we have ignored the context switching time. However, this needs to be considered, as a large number of context switchings can result in considerable wastage of time. In general, all these context switchings may be added to determine whether a set of tasks still remains RMS schedulable. For example, for the set of three tasks that we are considering, if we include a context switching time of 0.1 unit, there are 13 context switchings leading to an overhead of 1.3 time units. In this case, the schedule is no more a feasible one, as the total time required is 15.3 time units.

Aperiodic tasks

Aperiodic tasks are part and parcel of any real-time system. These often correspond to the critical emergency situations which should be responded to and completed within a limited time frame. On the other hand, since these are aperiodic, we cannot assign them any static priority to be used for RMS. Thus, special care needs to be taken to handle such tasks.

Typical solution to this problem involves dedicating one (or more) periodic task(s) to *pick up* aperiodic tasks that may need to be executed. Such a periodic task is called *aperiodic server*. There are two types of aperiodic servers:

- Deferrable server
- Sporadic server



Both the servers work on a fixed *budget* for resource utilization. If a new aperiodic task arrives and there is enough budget, the task will get scheduled immediately. However, if the budget is not enough, the task has to wait till enough budget is accumulated. In the case of *deferrable server*, the budget is replenished to its full value at regular intervals of time (dictated by the periodicity of the server). This makes the implementation of the server easier, however, at the same time makes the schedulability analysis very hard due to a phenomenon called *deferred execution effect* or *jitter effect*. In particular, assume that an aperiodic task arrives near the end of the server's period. If the server has enough budget, it will be scheduled immediately. The task may utilize its budget just in time for a new server's period. Since the budget gets replenished, the task will grab the resource for another execution. Moreover, if for a large time no aperiodic task arrives, the budget will go on accumulating. It may result in admitting a large number of aperiodic tasks in a burst.

On the other hand, in a *sporadic server*, the time of budget replenishment depends on the last time at which previous aperiodic task finished. As soon as the budget is utilized, the system initiates a timer. The budget is replenished at the expiry of the timer. Thus, there is a guaranteed minimum separation between two instances of a task. This aids in considering the aperiodic tasks also as periodic ones for RMS analysis purpose.

Limited priority levels

In most of the processors and operating systems, the number of different priority levels is limited. This can create problem in handling a large number of tasks. This is because even if based upon their periodicity values, the tasks are assigned different priorities, in actual implementation, so many priority levels may not be available. This will necessitate grouping of tasks with different priorities into same level. There are several schemes for grouping tasks into priority levels. These are:

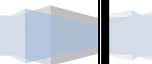
- Uniform
- Arithmetic
- Geometric
- Logarithmic

In *uniform* assignment, equal number of tasks are assigned to all the levels. In case it is not possible to distribute evenly, more number of tasks are allocated to lower priority levels. For example, if there are 11 tasks of different priorities to be assigned to 4 priority levels, the following is the distribution. Two tasks are assigned to the highest priority level. Now, 3 levels are left with 9 tasks. Thus, three tasks are assigned to each of these levels.

In *arithmetic* assignment scheme, the number of tasks assigned to successive levels form an arithmetic progression. For example, if there are 15 tasks and 5 priority levels, the tasks assigned to successive levels from the highest to the lowest level may be 1, 2, 3, 4, 5.

In *geometric* scheme, tasks assigned to successive levels form a geometric progression. For example, if there are 15 tasks to be distributed between 4 priority levels, number of tasks alloted to different levels from the highest to the lowest level may be 1, 2, 4, 8.

In *logarithmic* priority assignment process, the range of task periods are divided into logarithmic intervals. If p_{min} be the smallest period and p_{max} be the largest period, then we calculate a factor $r = (p_{max}/p_{min})^{1/n}$, n being the total number of priority levels. Now, all tasks with periods less than $p_{min} \times r$ are assigned to the highest priority level, tasks with periods between $p_{min} \times r$ and $p_{min} \times r^2$ are assigned to the next priority level, and so on. For example, consider a set of 11 tasks with periods 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 and the number of available priority levels to be 4. Thus, $p_{min} = 5, p_{max} = 15, n = 4$. Thus,



$$
\begin{aligned}
p_{min} \times r &= 6.58 \\
p_{min} \times r^2 &= 8.66 \\
p_{min} \times r^3 &= 11.79 \\
p_{min} \times r^4 &= 15.0
\end{aligned}
$$

3.12.3 Earliest Deadline first scheduling:

Earliest deadline first (EDF) is a dynamic priority algorithm in which priority of an instance of a task depends on its deadline. Unlike RMS, the priority of a task is not fixed for all instances. The priority changes dynamically during the life-time of the system. At any scheduling point, among all the task instances ready for execution, the one whose deadline is the earliest is picked up for scheduling. This justifies the name *earliest deadline first*. Since the scheduling decisions are taken dynamically, it may so happen that a task set misses deadline when scheduled using fixed priority approach, however, it becomes schedulable in the EDF policy. For example, consider a set of three tasks P_1, P_2, and P_3 with worst case execution times of 1, 2, and 3 units, respectively. Assume the respective periodicity values to be 4, 6, and 8. Now, in the fixed priority assignment scheme with RMS principle, P_1 will have the highest priority, followed by P_2 and P_3. The overall utilization is given by, $u = 1/4 + 2/6 + 3/8 = 0.958$, much higher than 0.692. Figure 6.9 shows a snap-shot of the schedule as produced using the fixed priority values. Here, at time instant 6, though the deadline of P_3 is the closest one, since P_2 has got higher priority, it gets scheduled. Consequently, it leads to a deadline miss for the task P_3. On the otherhand, using EDF policy, the same set of tasks can be scheduled, so that no deadline miss occurs. This is shown in Fig. 6.10.

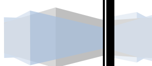
Given a set of n periodic and sporadic tasks with relative deadlines equal to their periods, the task set is schedulable by EDF if and only if the total utilization is less than or equal to unity, that is,

$$U = \sum_{i=1}^{n} e_i/p_i \leq 1$$

Thus, EDF is an optimal algorithm in the sense that if a task set is schedulable, it is schedulable by EDF as well. The phases of tasks also do not matter.

EDF enjoys the following advantages over the fixed priority-based approaches.

1. There is no need to define priorities. Based on deadlines, the priority values can change. On the other hand, for fixed-priority cases, it cannot be altered.
2. In general, EDF has lesser context switches.
3. The processor utilization is generally very high, reaching almost 100%. Thus, idle time is less.



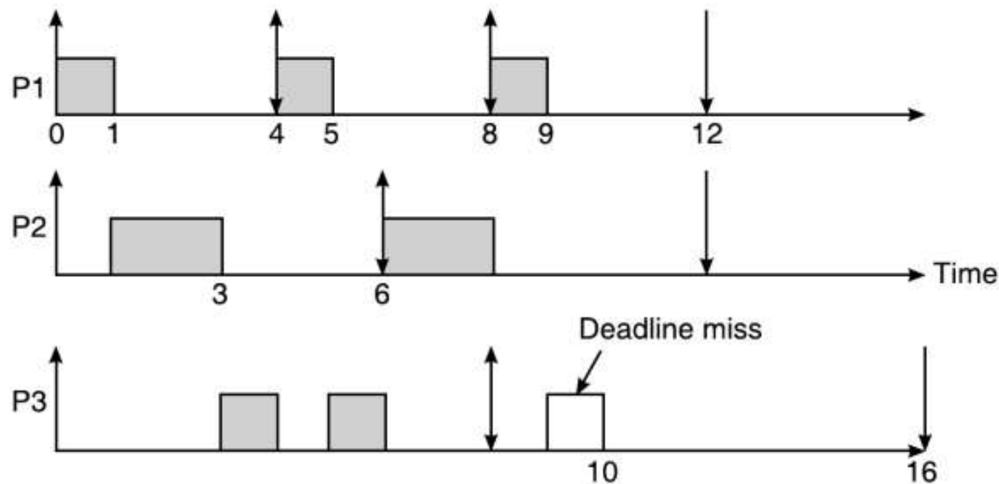


Fig. 6.9 Deadline miss with fixed priority schedule.

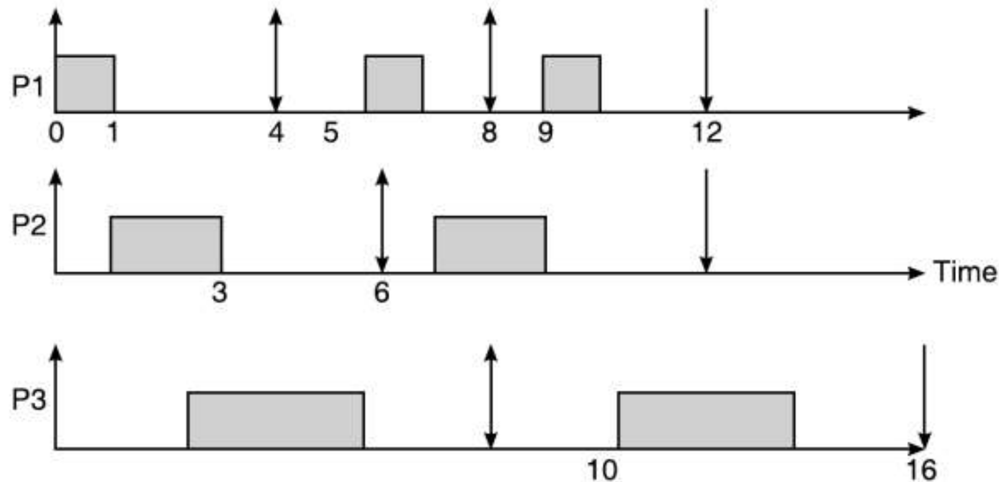
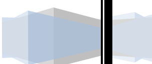


Fig. 6.10 No deadline miss with EDF schedule.

Disadvantages of EDF

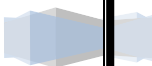
EDF suffers from a number of shortcomings as well. In the following list we highlight some of these shortcomings:

1. EDF is less predictable. The response time of a particular instance of a task depends upon the deadlines of instances of other tasks. Thus, the response time is variable.
2. EDF is less controllable in the sense that we cannot control the response time of a task. In fixed-priority algorithms, we can increase the priority of the task to reduce the response time. This is not possible with EDF.
3. Implementation overhead is higher than fixed priority approaches. This has already been discussed.



4. **Transient overload problem:** As discussed earlier, transient overload refers to the situation in which some task instance takes more time to complete than the planned one. This is a very transient behaviour and can occur due to various exceptional execution sequences occurring in the task. This overshooting may cause some other tasks to miss deadlines. As there is no fixed sequence in which the tasks are executed, it is very much possible that the most critical task of an application is the victim of the least critical one overshooting its execution time estimate. It may be noted that such a situation does not occur with fixed priority approaches like RMS.
5. **Domino effect:** EDF can also show domino effect. This may occur when the utilization of a set of tasks becomes greater than 1. Here, all the tasks in the application miss their deadlines, one after the other like a set of dominos, such that, falling of one domino initiates the falling of other dominos in sequence. This has been shown in Fig. 6.11, in which there are four tasks P_1, P_2, P_3, and P_4. Their deadlines are close to each other. Total utilization for these four tasks is equal to $2/5 + 2/6 + 2/7 + 2/8 = 1.27$. As shown in the figure, all the tasks miss their deadlines. The periodicity of the tasks are assumed to be equal to their deadlines.

   It may be noted that the fixed priority algorithms are more predictable. In a similar situation as in Fig. 6.12, only the lower priority tasks may miss deadlines. The tasks P_1 and P_2 never miss their deadlines. Task P_3 misses its deadline quite often, whereas the task P_4 never gets a chance to execute. That way, EDF is more *fair* as all tasks are treated in the same way!



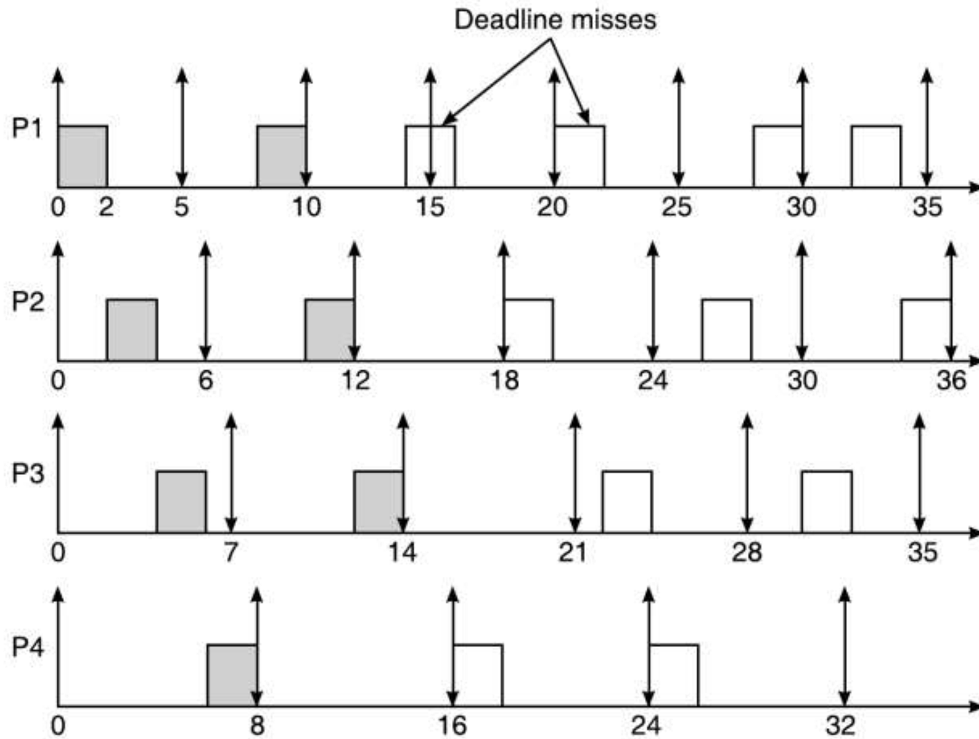


Fig. 6.11 Domino effect with EDF schedule.

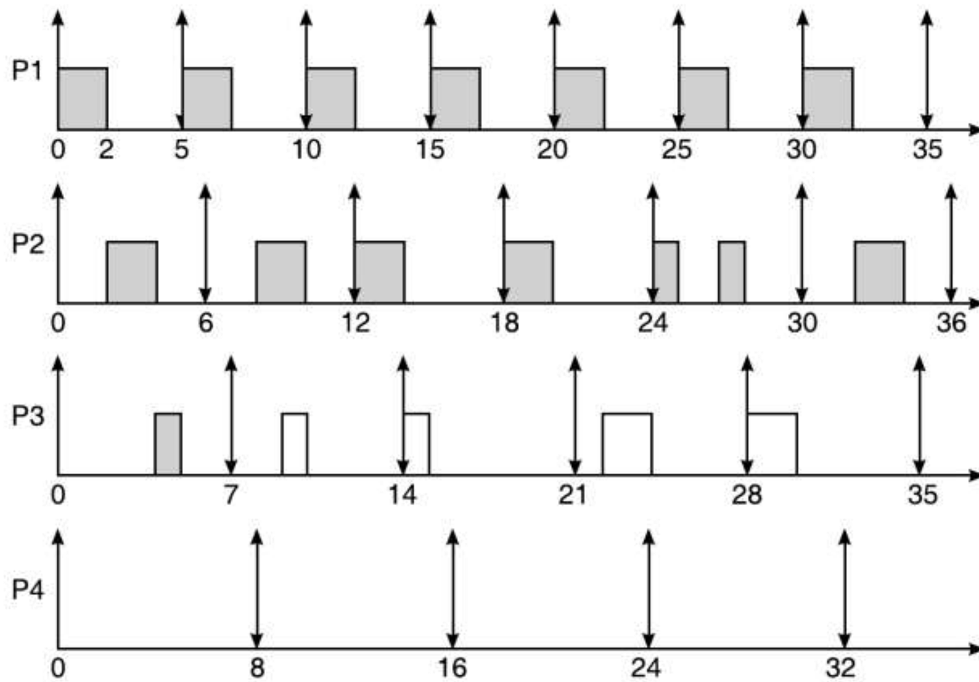
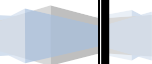


Fig. 6.12 No deadline miss for high priority tasks with fixed priority.



3.13 Inter process communication:

Inter process communication (IPC) means that a process (scheduler or task or ISR) generates some information by setting or resetting a flag value or generates an output. So that it let another process take use of it under the control of an OS.

IPCs in a multiprocessor systems are used to generate information about certain sets of computations finishing on one processor and allowing another processor to use it.

i) Signal
ii) Semaphore(as flag, mutex) for the inter task communication between tasks.
iii) Queue, pipe and mailbox
iv) Socket
v) Remote Procedure Call (RPC) for distributed processes

3.13.1 Shared data problem

Shared data problem can arises in a system when another higher priority task finishes an operation and modifies the data or a variable disabling interrupt mechanism using semaphores and using re-entrant function. Here are some solutions.

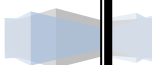
- The shared data problem can be explained as follows. Assume that several functions share a variable.
- Inconsistency may results as a result of access of same variable by many functions.
- The following steps are the steps that, if used together, almost eliminate a likely error in the program due to shared data problem

Use of modifier volatile with a declaration for a variable that returns from the interrupt.
Use re-entrant functions with atomic instructions in that part of a function that needs its complete execution before it can be interrupted. This part is called the critical section.
Put a shared variable in a circular queue. A function that requires the value of this always deletes (takes) it from queue front, and another function, which inserts (writes) the value of this variable, always written in queue back. Now a problem can occur in case there are a large number of functions that send the value into and get the value from the queue, and the queue is insufficient.
Disable the interrupts before a critical section starts executing and enable the interrupts on its completion. An interrupt, even if of higher priority than the present critical function, gets disabled.

3.13.2 Uses of Semaphore:

An RTOS provides the IPC function for creating and using semaphores as event flags Mutex resource keys (for resource locking and unlocking into process) and as counting semaphore.

- The semaphore as event flag facilitates inter-task communication for messaging (through a scheduler) a waiting task M to the running task N or at an ISR.
- A semaphore token generated at one place is usable at another place. Before entering critical section at the running place, a flag, Sem_NTakenFlag becomes “true”.
- Mutex gives a resource key and facilitates the communication between the task and scheduler. The key is to get an access to a resource.



- It solves the shared data problem (shared resource problem). It works as a resource locking mechanism it the access to certain resources is blocked.
- The blocking period of a task during the period when other tasks have taken the semaphore can be limited by defining time out value.
- Using the key, in a similar manner, a time consuming ISR can also be blocked after a present time interval time-out. The ISR takes a semaphore and releases it after the time-out to let the other ISR run.
- A spin lock does not let a running task may be blocked instantly, but first successively tries decreasing the trial periods before finally blocking a task.

The counting semaphore facilitates multiple inter task communications.

3.13.3 Priority Inversion Problem and Deadlock Situations:

Let the priority of the task be in an order such that task I is of highest priority, task J is of lower and task k is the lowest. Assume that only task I and K shares the data and J does not share the data with K. Also let tasks I and K alone share a semaphore and not J.

At any instance t_0, suppose task K shares semaphore, the OS does not blocks task J and I. This is because only task I and K shares the data and J does not shares the data

The problem that arises on selective sharing between K and I.

At next instance of time t_1, let task K began to ready to run.

At next instance of time t_2, task I began to ready on an interrupt. At this instance, task K is in critical section. So, task I cannot start this instance due to K being in critical section. Now, at next instance t_3, some event causes the unblocked higher than the K priority task J to run.

After time t_3, running task J does not allow the higher priority task J to run. This is because even though K is not running and thus unable to unlock the task I.

Further the task J is in the same condition and does not allow task I to run.

Thus J action is now higher priority than I. This brings K to enter the critical section.

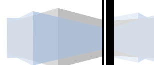
The priority information of another higher priority task I should have also been blocked by K temporarily. if K waits for I , but J does not wait. So J runs.

This situation is called priority inversion problem

Consider another problem. Assume the following situation.

1. Let the priorities of tasks be such that task *H* is of highest priority. Then task *I* has a lower priority and task *J* has the lowest.
2. There are two semaphores, *SemTok1* and *SemTok2*. This is because the tasks *I* and *H* have a shared resource through *SemTok1* only. Tasks *I* and *J* have two shared resources through two semaphores, *SemTok1* and *SemTok2*.
3. Let *J* interrupt at an instant t_0 and first take both the semaphores *SemTok1* and *SemTok2* and run.

Assume that at a next instant t_1, being now of a higher priority, the task *H* interrupts the tasks *I* and *J* after it takes the semaphore *SemTok1*, and thus blocks both *I* and *J*. In-between the time interval t_0 and t_1, the *SemTok1* was released but *SemTok2* was not released during the run of task *J*. But the latter did not matter as the tasks *I* and *J* do not share *SemTok2*. At an instant t_2, if *H* now releases the *SemTok1*, allows the task *I* to take it. Even then it cannot run because it is also waiting for task *J* to release the *SemTok2*. The task *J* is waiting at a next instant t_3, for either *H* or *I* to release the *SemTok1* *because it needs this to again enter a critical section*. Neither task *I* can run after instant t_2 nor task *J*. There is a circular dependency established between *I* and *J*.



Thus the use of MUTEX solves the deadlock problem in certain OSes.

3.14 Evaluating operating system performance:

The scheduling policy does not tell us about the performance of a real system running processes. Our analysis of scheduling policies makes some simplifying assumptions:

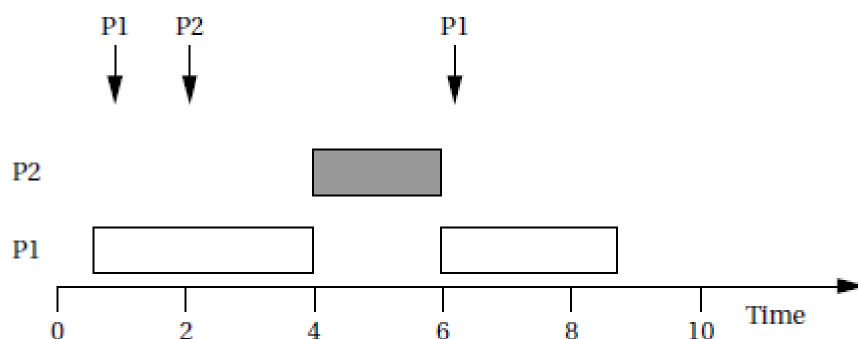
- We have assumed that context switches require zero time. Although it is often reasonable to neglect context switch time when it is much smaller than the process execution time, context switching can add significant delay in some cases.
- We have assumed that we know the execution time of the processes. In fact, that program time is not a single number, but can be bounded by worst-case and best-case execution times.
- We probably determined worst-case or best-case times for the processes in isolation. But in fact they interact with each other in the cache. Cache conflicts among processes can drastically degrade process execution time.

The zero-time context switch assumption used in the analysis of RMS is not accurate, we must execute instructions to save and restore context, and we must execute additional instructions to implement the scheduling policy. On the other hand, context switching can be implemented efficiently context switching need not kill performance. The effects of nonzero context switching time must be carefully analyzed in the context of a particular implementation to be sure that the predictions of an ideal scheduling policy are sufficiently accurate.

The following example shows that context switching can, in fact, cause a system to miss a deadline.

Process	Execution time	Deadline
P1	3	5
P2	3	10

First, let us try to find a schedule assuming that context switching time is zero. Following is a feasible schedule for a sequence of data arrivals that meets all the deadlines:



Now let us assume that the total time to initiate a process, including context switching and scheduling policy evaluation, is one time unit. It is easy to see that there is no feasible schedule for the above release time sequence, since we require a total of

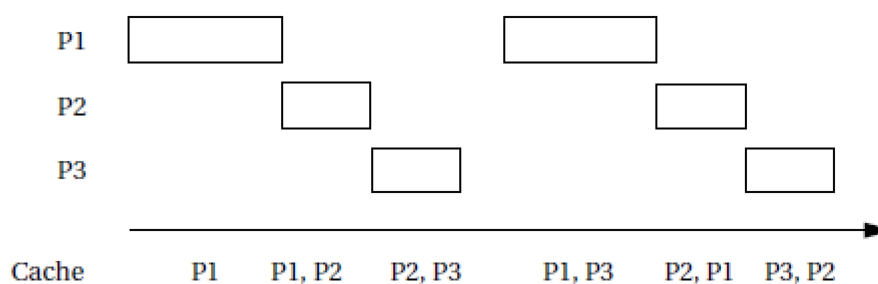
$2TP1 + TP2 = 2 \times (1 + 3) + (1 + 3) = 11$ time units to execute one period of P2 and two periods of P1.

Li and Wolf [Li99] developed a model for estimating the performance of multiple processes sharing a cache. In the model, some processes can be given reservations in the cache, such that only a particular process can inhabit a reserved section of the cache; other processes are left to share the cache. We generally want to use cache partitions only for performance-critical processes since cache reservations are wasteful of limited cache space. Performance is estimated by constructing a schedule, taking into account not just execution time of the processes but also the state of the cache. Each process in the shared section of the cache is modelled by a binary variable: 1 if present in the cache and 0 if not. Each process is also characterized by three total execution times: assuming no caching, with typical caching, and with all code always resident in the cache.The always-resident time is unrealistically optimistic, but it can be used to find a lower bound on the required schedule time. During construction of the schedule, we can look at the current cache state to see whether the no-cache or typical-caching execution time should be used at this point in the schedule. We can also update the cache state if the cache is needed for another process. Although this model is simple, it provides much more realistic performance estimates than assuming the cache either is nonexistent or is perfect.

Another example shows how cache management can improve CPU utilization.

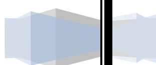
Process	Worst-case CPU time	Average-case CPU time
P1	8	6
P2	4	3
P3	4	3

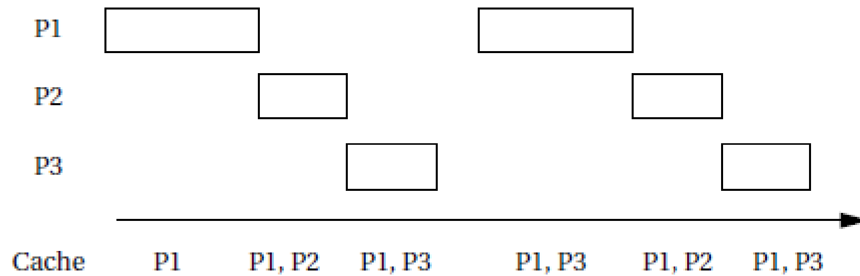
Each process uses half the cache, so only two processes can be in the cache at the same time. Appearing below is a first schedule that uses a least-recently-used cache replacement policy on a process-by-process basis.



In the first iteration, we must fill up the cache, but even in subsequent iterations, competition among all three processes ensures that a process is never in the cache when it starts to execute. As a result, we must always use the worst-case execution time.

Another schedule in which we have reserved half the cache for P1 is shown below. This leaves P2 and P3 to fight over the other half of the cache.





3.15 Power optimization strategies for processes

A power management policy is a strategy for determining when to perform certain power management operations. A power management policy in general examines the state of the system to determine when to take actions. However, the overall strategy embodied in the policy should be designed based on the characteristics of the static and dynamic power management mechanisms.

Going into a low-power mode takes time; generally, the more that is shut off, the longer the delay incurred during restart. Because power-down and power-up are not free, modes should be changed carefully. Determining when to switch into and out of a power-up mode requires an analysis of the overall system activity.

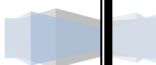
- Avoiding a power-down mode can cost unnecessary power.
- Powering down too soon can cause severe performance penalties

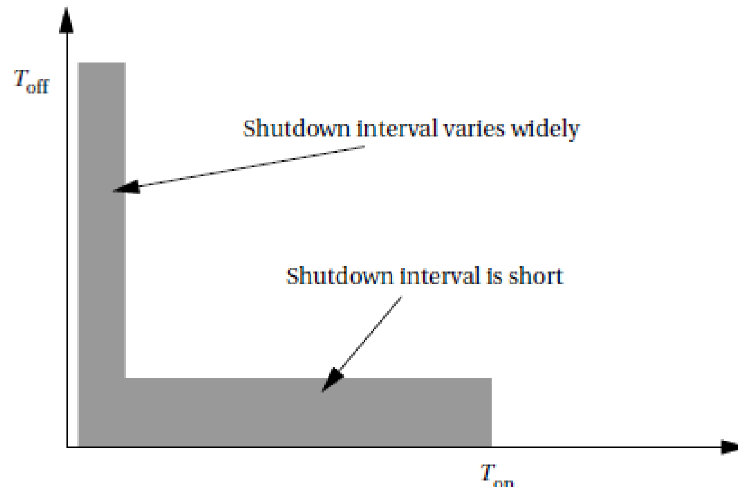
A straightforward method is to power up the system when a request is received. This works as long as the delay in handling the request is acceptable. A more sophisticated technique is **predictive shutdown**. The goal is to predict when the next request will be made and to start the system just before that time, saving the requestor the start-up time. In general, predictive shutdown techniques are probabilistic. They make guesses about activity patterns based on a probabilistic model of expected behavior. Because they rely on statistics, they may not always correctly guess the time of the next activity. This can cause two types of problems:

- The requestor may have to wait for an activity period. In the worst case, the requestor may not make a deadline due to the delay incurred by system start-up.
- The system may restart itself when no activity is imminent. As a result, the system will waste power.

A very simple technique is to use fixed times. For instance, if the system does not receive inputs during an interval of length *T*on, it shuts down; a powered-down system waits for a period *T*off before returning to the power-on mode. The choice of *T*off and *T*on must be determined by experimentation.

The observed idle time (*T*off*)* of a graphics terminal versus the immediately preceding active time (*T*on*)*.The result was an L-shaped distribution as illustrated in Fig. In this distribution, the idle period after a long active period is usually very short, and the length of the idle period after a short active period is uniformly distributed. Based on this distribution, they proposed a shut down threshold that depended on the length of the last active period. They shutdown when the active period length was below a threshold, putting the system in the vertical portion of the *L* distribution.

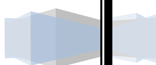


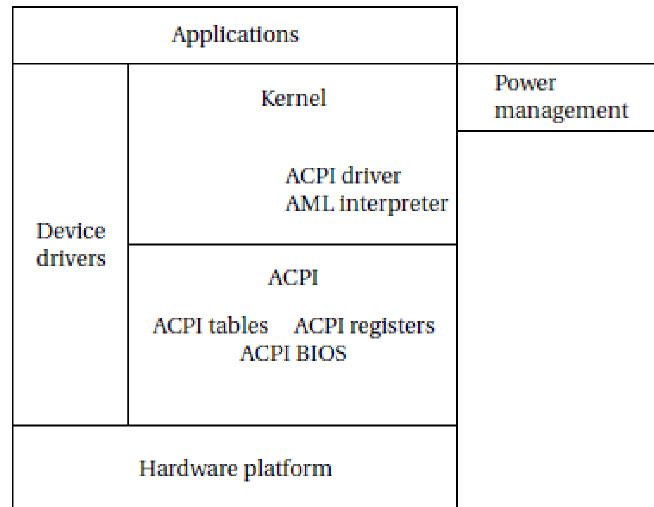


The ***Advanced Configuration and Power Interface (ACPI)*** is an open industry standard for power management services. It is designed to be compatible with a wide variety of OSs. It was targeted initially to PCs. The role of ACPI in the system is illustrated in Figure 6.18.ACPI provides some basic power management facilities and abstracts the hardware layer, the OS has its own power management module that determines the policy, and the OS then uses ACPI to send the required controls to the hardware and to observe the hardware's state as input to the power manager.

ACPI supports the following five basic global power states:

- G3, the mechanical off state, in which the system consumes no power.
- G2, the soft off state, which requires a full OS reboot to restore the machine to working condition. This state has four substrates:
  i) S1, a low wake-up latency state with no loss of system context;
  ii) S2, a low wake-up latency state with a loss of CPU and system cache state;
  iii) S3, a low wake-up latency state in which all the system except for main memory is lost;
  iv) S4, the lowest-power sleeping state, in which all devices are turned off.
- G1, the sleeping state, in which the system appears to be off and the time required to return to working condition is inversely proportional to power consumption.
- G0, the working state, in which the system is fully usable.
- The legacy state, in which the system does not comply with ACPI.





STRUCTURE OF A REAL TIME SYSTEM

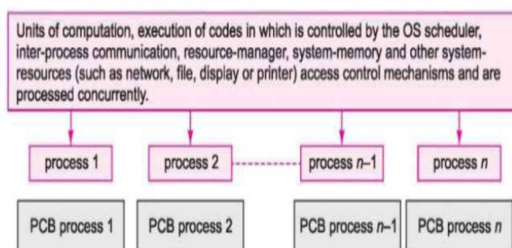
The state of the controlled process and of the operating environment (e.g pressure, temperature, speed and altitude) is acquired by sensors, which provides inputs to the controller,the real time computer.The Data from each sensor depends on how quickly the measured parameters can change,it is usually less than 1kb/second.

.

Process Concepts

A process consists of executable program (codes), *state* of which is controlled by OS the *state* during running of a process represented by process-status (running, blocked or finished), process structure—its data, objects and resources, and process control block (PCB).

Process



Runs when it is scheduled to run by the OS (kernel)

OS gives the control of the CPU on a process‘s request (system call).

Runs by executing the instructions and the continuous changes of its state takes Place as the program counter (PC) changes.

• Process is that executing unit of computation, which is controlled by some process (of the OS) for a scheduling mechanism that lets it execute on the CPU and by some process at OS for a resource management mechanism that lets it use the system memory and other system resources such as network, file, display or printer.

Application program can be said to consist of number of processes

Example - Mobile Phone Device embedded software

- Software highly complex.
- Number of functions, ISRs, processes threads, multiple physical and virtual device drivers, and several program objects that must be concurrently processed on a single processor.
- Voice encoding and convoluting process— the device captures the spoken words through a speaker and generates the digital signals after analog to digital conversion, the digits are encoded and convoluted using a CODEC,
- Modulating process,
- Display process,
- GUIs (graphic user interfaces), and
- Key input process — for provisioning of the user interrupts

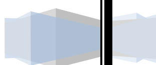
Process Control Block

- A data structure having the information using which the OS controls the Process state.
- Stores in protected memory area of the kernel.
- Consists of the information about the process state

Information about the process state at Process Control Block…

Process ID,

- process priority,
- Parent process (if any),
- child process (if any), and
- address to the next process PCB which will run,
- allocated program memory address blocks in physical memory and in secondary virtual) memory for the process-codes,
- allocated process-specific data address blocks
- allocated process-heap (data generated during the program run) addresses,
- allocated process-stack addresses for the functions called during running of the process,
- allocated addresses of CPU register-save area as a process context represents by CPU registers, which include the program counter and stack pointer



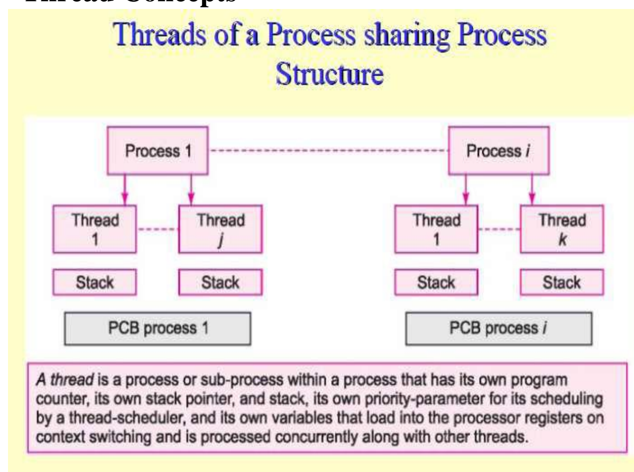
- allocated addresses of CPU register-save area as a process context [Register-contents (define process context) include the program counter and stack pointer contents]
- process-state signal mask [when mask is set to 0 (active) the process is inhibited from running and when reset to 1, the process is allowed to run],
- Signals (messages) dispatch table [process IPC functions],
- OS allocated resources' descriptors (for example, file descriptors for open files, device descriptors for open (accessible) devices, device-buffer addresses and status, socket descriptor for open socket), and Security restrictions and permissions.

Context

- Context loads into the CPU registers from memory when process starts running, and the registers save at the addresses of register-save area on the context switch to another process
- The present CPU registers, which include program counter and stack pointer are called context
- When context saves on the PCB pointed process-stack and register-save area addresses, then the running process stops.
- Other process context now loads and that process runs— This means that the context has switched.

Threads and Tasks

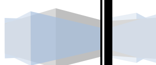
Thread Concepts



- A thread consists of executable program (codes), *state* of which is controlled by OS,
- The state information— *thread-status* (running, blocked, or finished), *thread structure*—its data, objects and a subset of the process resources, and *thread-stack.* Considered a lightweight process and a process level controlled entity.[Light weight means its running does not depend on system resources] .

Process… heavyweight

• Process considered as a heavyweight process and a kernel-level controlled entity.



• Process thus can have codes in secondary memory from which the pages can be swapped into the physical primary memory during running of the process. [Heavy weight means its running may depend on system resources]

• May have process structure with the virtual memory map; file descriptors, user–ID, etc.

• Can have multiple threads, which share the process structure thread

• *A* process or sub-process within a process that has its own program counter, its own stack pointer and stack, its own priority parameter for its scheduling by a thread scheduler

• Its variables that load into the processor registers on context switching.

• Has own signal mask at the kernel. Thread‘s signal mask

• When unmasked lets the thread activate and run.

• When masked, the thread is put into a queue of pending threads.

Thread‘s Stack

• A thread stack is at a memory address block allocated by the OS.

Application program can be said to consist of number of threads or Processes:

Multiprocessing OS

• A multiprocessing OS runs more than one processes.

• When a process consists of multiple threads, it is called multithreaded process.

• A thread can be considered as daughter process.

• A thread defines a minimum unit of a multithreaded process that an OS schedules On to the CPU and allocates other system resources.

Thread parameters

• Each thread has independent parameters ID, priority, program counter, stack pointer, CPU registers and its present status.

• Thread states─ starting, running, blocked (sleep) and finished

Thread’s stack

• When a function in a thread in OS is called, the calling function state is placed on the stack top.

• When there is return the calling function takes the state information from the stack top

• A data structure having the information using which the OS controls the thread state.

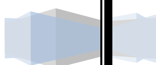
• Stores in protected memory area of the kernel.

• Consists of the information about the thread state

Thread and Task

• Thread is a concept used in Java or Unix.

• A thread can either be a sub-process within a process or a process within an application program.



• To schedule the multiple processes, there is the concept of forming thread groups and thread libraries.

• A task is a process and the OS does the multitasking.

• Task is a kernel-controlled entity while thread is a process-controlled entity.

• A thread does not call another thread to run. A task also does not directly call another task to run.

• Multithreading needs a thread-scheduler. Multitasking also needs a task-scheduler.

• *There may or may not be task groups* and task libraries in a given OS

Task and Task States

Task Concepts

• An application program can also be said to be a program consisting of the tasks and task behaviors in various states that are controlled by OS.

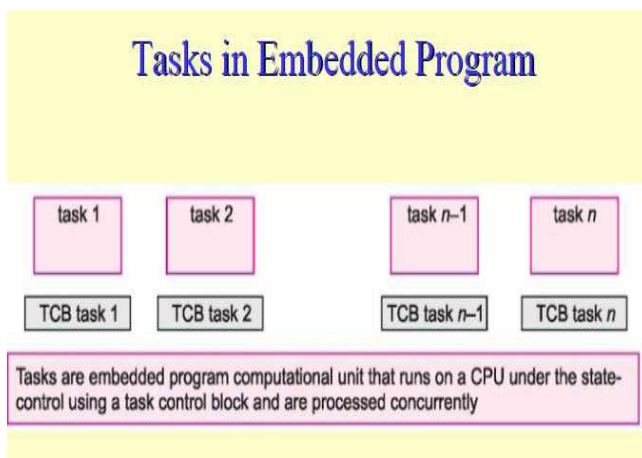
• A task is like a process or thread in an OS.

• Task─ term used for the process in the RTOSes for the embedded systems. For example, VxWorks and μCOS-II are the RTOSes, which use the term task.

• A task consists of executable program (codes), *state* of which is controlled by OS, the *state* during running of a task represented by information of process status (running, blocked, or finished),process-structure—its data, objects and resources, and task control block (PCB).

• Runs when it is scheduled to run by the OS (kernel), which gives the control of the CPU on a task request (system call) or a message.

• Runs by executing the instructions and the continuous changes of its state takes place as the program counter (PC) changes.

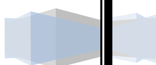


• Task is that executing unit of computation, which is controlled by some process at the OS scheduling mechanism, which lets it execute on the CPU and by some process at OS for a resource-management mechanism that lets it use the system memory and other system-resources such as network, file, display or printer.

• A task─ an independent process.

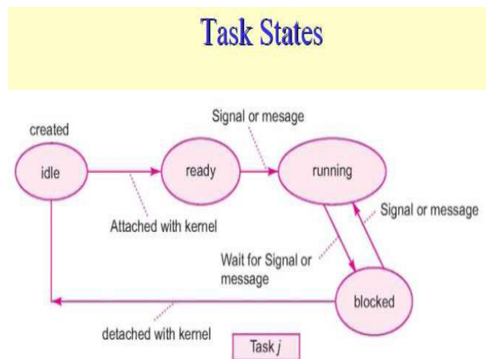
• No task can call another task. [It is unlike a C (or C++) function, which can call another function.]

• The task─ can send signal (s) or message(s) that can let another task run.



• The OS can only block a running task and let another task gain access of CPU to run the servicing codes

Task States



(i) Idle state [Not attached or not registered]

(ii) Ready State [Attached or registered]

(iii) Running state

(iv) Blocked (waiting) state

(v) Delayed for a preset period

Idle (created) state

• The task has been created and memory allotted to its structure however, it is not ready and is not schedulable by kernel.

Ready (Active) State

• The created task is ready and is schedulable by the kernel but not running at present as another higher priority task is scheduled to run and gets the system resources at this instance.

Running state

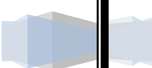
• Executing the codes and getting the system resources at this instance. It will run till it needs some IPC (input) or wait for an event or till it gets pre-empted by another higher priority task than this one.

Blocked (waiting) state

• Execution of task codes suspends after saving the needed parameters into its Context. It needs some IPC (input) or it needs to wait for an event or wait for higher priority task to block to enable running after blocking.

Deleted (finished) state

• Deleted Task— The created task has memory de allotted to its structure. It frees the memory. Task has to be re-created.



UNIT IV

Reliability and Clock Synchronization: Introduction to Reliability Evaluation Techniques – Reliability Models for Hardware Redundancy – Permanent faults only - Transient faults. Introduction to clock synchronization – A Non-Fault-Tolerant Synchronization Algorithm - Fault-Tolerant Synchronization in Hardware – Completely connected zero propagation time system – Sparse interconnection zero propagation time system –Fault tolerant analysis with Signal Propagation delays.

4.1 Introduction to reliability evaluation technique:

Suppose we have a system that is supposed to fail on the average for once in every 10^{10} hours. To validate this experiment, we have to run this system for a millions of years. To overcome this difficulty, we use mathematical model of reliability. We construct a mathematical model of real time system and solve it. There are two reliable models

- Hardware reliability model
- Software reliability model

(i) Obtaining parameter value:

The first step in developing model is to decide the input parameters. A model should always placed on parameters that can be either accurately measured or estimated.

- **Obtaining device failure rate:**

  There are two ways to obtain device failure rate

  (a) Collecting field data

  (b) Life cycle testing in laboratory

  Collecting field data is more realistic, Since it represent the failure rate when the device is used in normal operating condition.

  Life cycle testing is used only when the field data of device is not known.

In laboratory, the devices are subjected to “accelerated testing”. That is to gather the data, we stress the device so that their failure rare can increase by some factor. the most common accelerant is temperature.

The acceleration factor is given by

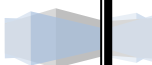
$$R(t) = Ae^{\frac{-Ea}{KT}}.$$

Where A is constant

T is temperature

E_a is activation energy and depends on logic family used

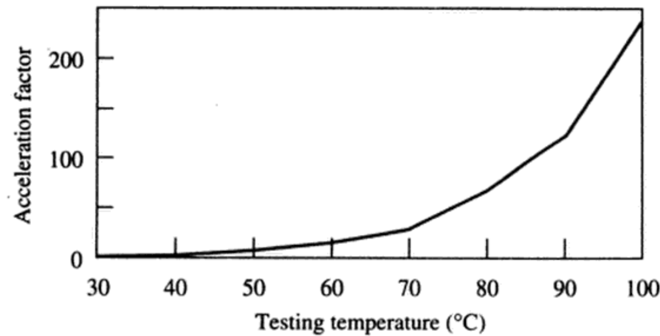
K is Boltzmann constant (0.8625 X 10^{-4} eV/ K).



At temperature T_1, the device failure rate can be

$$\frac{R(\mathrm{T1})}{R(\mathrm{T2})} = e^{\frac{-Ea}{K}\left(\frac{1}{T1} - \frac{1}{T2}\right)}.$$

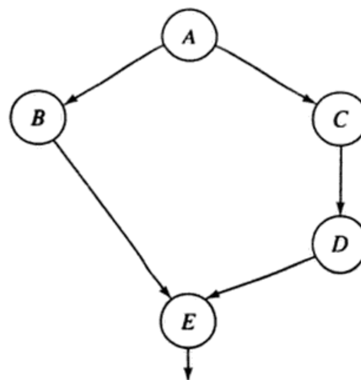
We can see that for testing for an hour at 100° C is equivalent to testing for almost 250 hours at 25° C.



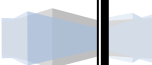
(ii) Measuring Error propagation time:

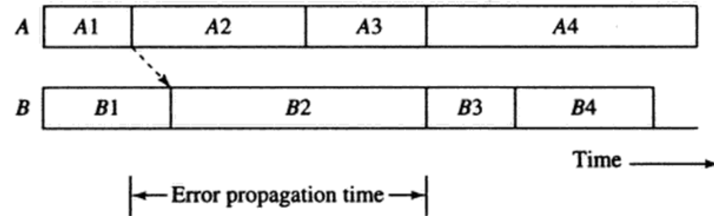
To measure how quickly an error can propagate through, we use fault injection. Special purpose hardware is used to simulate a fault on a selected line. The status of related line is monitored by using logic analyzers to determine the flow of error propagation.

The flow of error can be expressed by directed graph. For example if an error originates (starts) at A and propagates to B and C. This error creates an error in output of these processor and output of C causes error in output of D. Finally E receives error input from B and D produce error result.



Such a graph along with time taken by each module to produce the error in response to error input, can be used to compute error propagation time from output of A to output of E.





The task schedule for processor A and B as Shown in Fig.***. Task A_i and B_i are run on A and B respectively. The A_1 output is error and used by the task B_2. The time for propagation of the error from the A_1 output to the B_2 output depends on when B_2 completes execution.

4.2 Reliability model for Hardware Redundancy:

The most difficult problem in reliability model is to keep the complexity of model is small. when the parameters of model are exponentially distributed, there is no such complexity problem. But to accurately model with parameters that observe other distribution usually result in **complexity for smaller systems.**

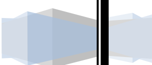
- Current technique to reduce the complexity is **state aggregation. State aggregation** is multiple states are grouped together and treated as single state.
- In decomposition, the overall model is broken into sub model and each sub model is interconnected between them.

In order to model the reliability of a system, we have to know reliability of each of its components and measure the failure of each component in overall systems.

The reliability of components is usually specified through a probability distribution function of the lifetime of that component.[2] For example, if failures occur as a Poisson process with rate λ, the lifetime distribution is given by $F_\ell(t) = 1 - \exp(-\lambda t)$. If failures occur as a Weibull process with a shape parameter α and scale parameter λ, the lifetime distribution is $F_\ell(t) = 1 - \exp(-[\lambda t]^\alpha)$. We will denote by $f_\ell(t)$ the associated density function (we will assume here that $F_\ell(t)$ is differentiable).

The *hazard rate* $h(t)$ of a component with age t is defined as the rate of failure at time t, given that it has not failed up to time t. We can use Bayes's law to express the hazard rate as a function of the lifetime distribution function.

$$\begin{aligned} h(t)dt &= \text{Prob}\{\text{System fails in } [t, t+dt] \mid \text{System has not failed up to } t\} \\ &= \frac{\text{Prob}\{\text{System fails in } [t, t+dt] \cap \text{System has not failed up to } t\}}{\text{Prob}\{\text{System has not failed up to } t\}} \\ &= \frac{f_\ell(t)dt}{1 - F_\ell(t)} \\ \Rightarrow h(t) &= \frac{f_\ell(t)}{1 - F_\ell(t)} \qquad (8.6) \end{aligned}$$



If the failure process is Poisson with rate λ (i.e., if the lifetime distribution is exponentially distributed with mean $1/\lambda$), then the hazard rate is

$$h(t) = \frac{\lambda e^{-\lambda t}}{e^{-\lambda t}} = \lambda \qquad (8.7)$$

The hazard rate is thus independent of the age of the component if the failure process is Poisson. Hence, to analyze the reliability of systems built out of such components, we do not need to know their age. This simplifies modeling tremendously.

If the failure process is Weibull with shape and scale parameters α and λ, respectively, the hazard rate is given by

$$h(t) = \alpha\lambda(\lambda t)^{\alpha-1} \qquad (8.8)$$

If $0 < \alpha < 1$, then $h(t)$ decreases with time. This means that the failure rate of a component drops as it ages. Components with decreasing hazard rates are said to have the *used-better-than-new* property. If $\alpha = 1$, the failure process is Poisson. If $\alpha > 1$, $h(t)$ increases with time; that is, the failure rate increases with age, and such components have the *new-better-than-used* property.

Many real-life components have a hazard rate shaped according to the *bath-tub curve*, shown in Figure 8.4. In the beginning the hazard rate is quite high, and then it begins to drop. This is known as the infant-mortality phase, where

FIGURE 8.4
The bathtub curve.

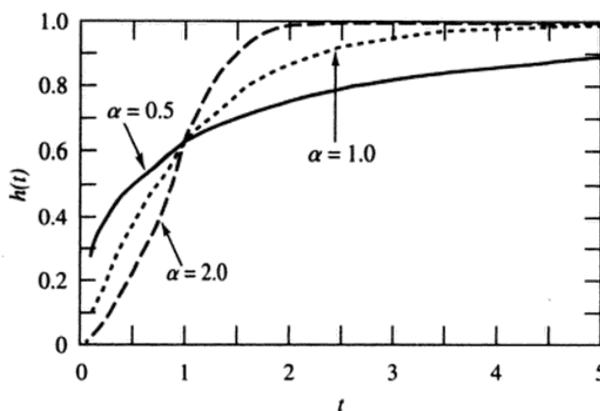


FIGURE 8.5
Examples of lifetime distributions, for $\lambda = 1$.

components with manufacturing defects are weeded out. The rate then becomes approximately constant, before aging effects set in and cause the hazard rate to rise with age. Note that Figure 8.4 is not drawn to scale. Typically, the interval of constant hazard rate is much longer than either the infant-mortality or aging regions.

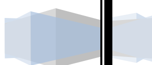
Define $\ell(t) = \int_0^t h(x)dx$. Then, the probability that the component will fail some time in the interval $[0, t]$, which is the PDF of the component lifetime, is given by $1 - e^{-\ell(t)}$. Figure 8.5 provides some numerical illustrations of the lifetime distribution associated with the Weibull distribution, $h(t) = \alpha\lambda(\lambda t)^{\alpha-1}$.

4.2.1 Permanent Faults only:

(a) Series systems:

The components in a set is said to be in series from a reliability if they must all work then only the entire system will meet a success. If any one component meet failure then the entire system will get fail.

Consider a system consisting of two independent components A and B connected in series, from a reliabiliy point of view, as shown in Figure 4.1. This arrangement implies that both components must work to ensure system success. Let R_A, R_B = probability of successful operation of components A and B respectively, and Q_A, Q_B = probability of failure of



components A and B respectively. Since success and failure are mutually exclusive and complementary,

$$R_A + Q_A = 1 \quad \text{and} \quad R_B + Q_B = 1$$

The requirement for system success is that 'both A and B' must be working. Equation 2.9 can be used to give the probability of system success or reliability as

$$R_S = R_A \cdot R_B \tag{4.1}$$

If there are now n components in series, Equation 4.1 can be generalized to give

$$R_S = \prod_{i=1}^{n} R_i \tag{4.2}$$

This equation frequently is referred to as the product rule of reliability since it establishes that the reliability of a series system is the product of the individual component reliabilities.

In some applications it may be considered advantageous to evaluate the unreliability or probability of system failure rather than evaluating the reliability or probability of system success. System success and system failure are complementary events and therefore for the two component system the unreliability is

$$Q_S = 1 - R_A R_B \tag{4.3}$$

$$= 1 - (1 - Q_A)(1 - Q_B)$$

$$= Q_A + Q_B - Q_A \cdot Q_B \tag{4.4}$$

or for an n component system,

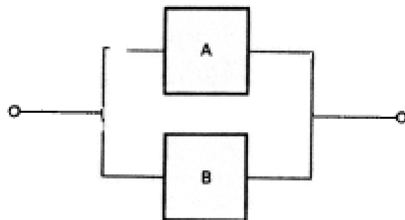
$$Q_S = 1 - \prod_{i=1}^{n} R_i \tag{4.5}$$

Equations 4.3 and 4.5 could have been derived directly from Equation 2.12 since the requirement for system failure is that 'A or B or both' must fail.

Now consider the application of these techniques to some specific problems.

(b) Parallel Systems:

The components in a set is said to be in parallel from a reliability if any one of them may work then the entire system will meet a success. If all component meet failure then the entire system will get fail.



In this case the system requirement is that only one component need be working for system success. The system reliability can be obtained as the complement of the system unreliability or by using Equation 2.12 since 'either A or B or both' constitutes success to give

$$R_P = 1 - Q_A \cdot Q_B \tag{4.6}$$

$$= R_A + R_B - R_A \cdot R_B \tag{4.7}$$

or for an n component system:

$$R_P = 1 - \prod_{i=1}^{n} Q_i \tag{4.8}$$

Also

$$Q_P = Q_A \cdot Q_B \tag{4.9}$$

and for an n component system:

$$Q_P = \prod_{i=1}^{n} Q_i \tag{4.10}$$

(c) Series parallel Systems:

The series and parallel systems discussed in the two previous sections form the basis for analysing more complicated configurations. The general principle used is to reduce sequentially the complicated configuration by combining appropriate series and parallel branches of the reliability model until a single equivalent element remains. This equivalent element then represents the reliability (or unreliability) of the original configuration. The following examples illustrate this technique which is generally known as a (network) reduction technique.

Example 4.7

Derive a general expression for the reliability of the model shown in Figure 4.5 and hence evaluate the system reliability if all components have a reliability of 0.9.

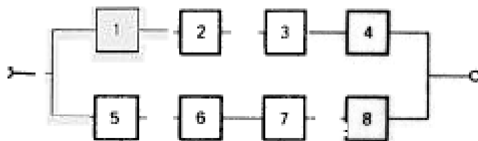


Fig. 4.5 Reliability diagram of Example 4.7

Solution:

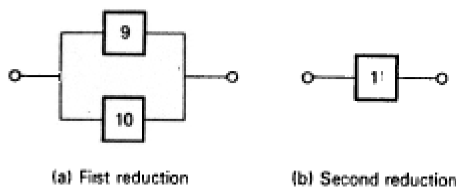


Fig. 4.6 Reduction of Example 4.7.

If $R_1, R_2, \ldots, R_8$ are the reliabilities of components $1, 2, \ldots, 8$ respectively then

$$R_9 = R_1R_2R_3R_4$$
$$R_{10} = R_5R_6R_7R_8$$
$$R_{11} = 1-(1-R_9)(1-R_{10})$$
$$= R_9 + R_{10} - R_9R_{10}$$
$$= R_1R_2R_3R_4 + R_5R_6R_7R_8 - R_1R_2R_3R_4R_5R_6R_7R_8$$

In deriving expressions of this type, it is possible to produce a number of apparently different equations when the final expression is written in terms of both Rs and Qs. These apparently different versions could all be correct and should reduce to the same one if manipulated and expressed in terms of either R or Q.

Using the data of Example 4.7, then

$$R_{11} = 0.9^4 + 0.9^4 - 0.9^8 = 0.8817$$

Example 4.8

Derive a general expression for the unreliability of the model shown in Figure 4.7 and hence evaluate the unreliability of the system if all components have a reliability of 0.8.

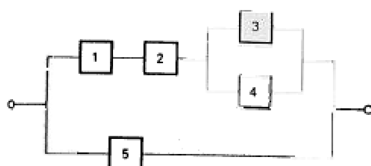


Fig. 4.7 Reliability diagram for Example 4.8

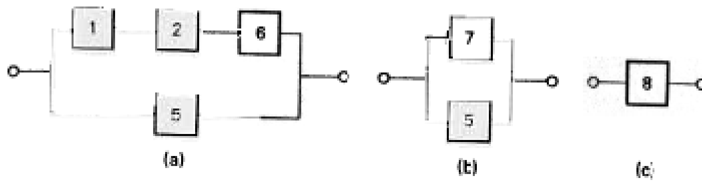


Fig. 4.8 Reduction of Example 4.8. (a) First reduction. (b) Second reduction. (c) Third reduction

If $R_1, \dots, R_5$ and $Q_1, \dots, Q_5$ are the reliabilities and unreliabilities of components $1, \dots, 5$ respectively, then

$$Q_6 = Q_3 Q_4$$

$$Q_7 = 1-(1-Q_1)(1-Q_2)(1-Q_6)$$

$$= Q_1 + Q_2 + Q_6 - Q_1Q_2 - Q_2Q_6 - Q_6Q_1 + Q_1Q_2Q_6$$

$$Q_8 = Q_5 Q_7$$

$$= Q_5(Q_1 + Q_2 + Q_3Q_4 - Q_1Q_2 - Q_2Q_3Q_4 - Q_3Q_4Q_1 + Q_1Q_2Q_3Q_4)$$

For the data given, $R_i = 0.8$ thus $Q_i = 0.2$ and $Q_8 = 0.07712$.

An equivalent expression to the above could have been deduced in terms of R_i.

$$R_6 = R_3 + R_4 - R_3R_4$$

$$R_7 = R_1R_2R_6$$

$$R_8 = R_5 + R_7 - R_5R_7$$

$$= R_5 + R_1R_2(R_3 + R_4 - R_3R_4) - R_5R_1R_2(R_3 + R_4 - R_3R_4)$$

which, for $R_i = 0.8$, gives:

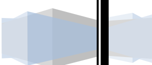
$$R_8 = 0.92288 \quad \text{or} \quad Q_8 = 1 - 0.92288 = 0.07712$$

4.3 Introduction to clock synchronization:

Clock: Clock c_i is a mapping real time to system clock time. That is at real time t, $c_i(t)$ is the time hold by clock c_i.

The clock of a computer is accurate to the clock of the real world. The clock should not gain or loss time at too high rate

The **drift rate** is the rate at which the clock can gain or loss time. The drift rate is given by



$$\rho = \max_{t,\Delta} \left| \frac{C_i(t+\Delta) - C_i(\Delta)}{\Delta} - 1 \right| \tag{9.1}$$

be as small as possible. The *drift rate* is the rate at which the clock can gain or lose time.

Conversely, if we specify ρ as the maximum drift rate of a nonfaulty clock, we must have

$$(1-\rho)(t_2 - t_1) \le C(t_2) - C(t_1) \le (1+\rho)(t_2 - t_1) \tag{9.2}$$

The clock is constrained to remain within the cone of acceptability, shown as heavy lines in Figure 9.1. Since $\rho \ll 1$ for all good clocks,

$$1 - \rho \approx (1+\rho)^{-1}$$

$$1 + \rho \approx (1-\rho)^{-1}$$

To see this, expand $(1+\rho)^{-1}$ in a Taylor series:

$$(1+\rho)^{-1} = 1 - \rho + \frac{\rho^2}{2!} + \cdots + (-1)^n \frac{\rho^n}{n!} + \cdots$$

For $\rho \approx 10^{-6}$, $(1-\rho) - (1+\rho)^{-1} \approx 10^{-12}$. Similarly for $(1-\rho)^{-1}$. Hence, Equation (9.2) is approximately the same as:

$$\frac{t_2 - t_1}{1+\rho}(t_2 - t_1) \le C(t_2) - C(t_1) \le \frac{t_2 - t_1}{1-\rho} \tag{9.3}$$

We will henceforth assume that ρ is such that Equation (9.3) holds for nonfaulty clocks. When we specify the maximum drift of a clock as ρ, we are in effect saying that both Equations (9.2) and (9.3) will hold as long as the clock is nonfaulty.

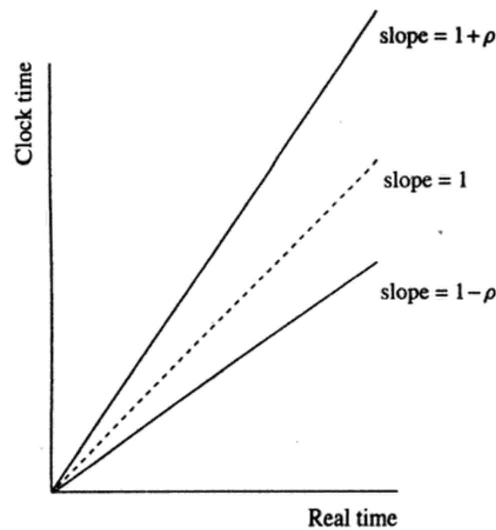
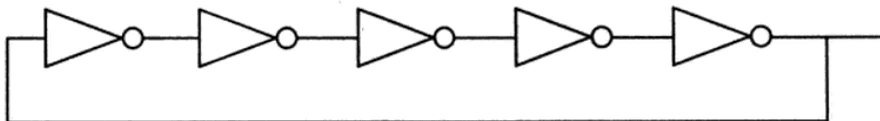


FIGURE 9.1
Range of permitted clock times.

The clock in the computer are square wave generator and time is expressed in multiple of square wave periods. Simplest digital clock is odd number of invertors as shown below. The propagation time for gate vary from gate to gate.



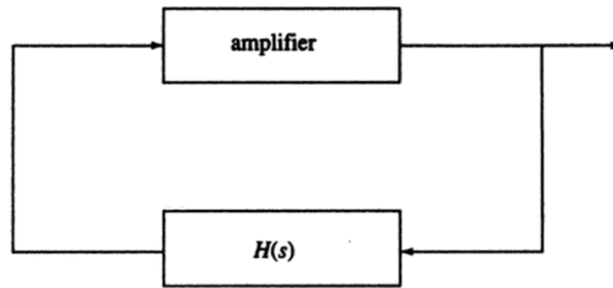


FIGURE 9.3
Using a filter to make an oscillator.

An another approach is to use feedback loop as an amplifier. If the gain of loop is at least one and amplifier is not saturated , then the output is sine wave output. If gain of loop is atleast one and amplifier is saturated , then the output is square wave output. It is better to use quartz crystal as filter.

Synchronization:

Two clocks are said to be synchronized if the times they tell are sufficiently close. More precisely, clocks c_i and c_j are synchronized at c-time T if for some given $\delta > 0$,

$$|c_i(T) - c_j(T)| < \delta \tag{9.4}$$

$|c_i(T) - c_j(T)|$ is the *skew* between clocks c_i and c_j at c-time T.

An alternative definition of synchronization is to define the clock skew at real time t as $|C_i(t) - C_j(t)|$, and to say that clocks c_i and c_j are synchronized

4.4 Non fault tolerant synchronisation algorithm:

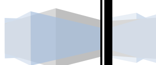
- At regular interval of T, each clock sends out its timing signals to the other clocks.
- A clock compares its own timing signals with those it receives from the others and adjusts itself appropriately.
- Clock c1 will slow itself down, clock c3 will speed itself up so that their next clock ticks will align as closely as possible with the next clock tick of clock c2.
- Adjust the clocks so that the next comparison point, they try to be aligned.
- The signal propagation times are zero
- The skew becomes worse when the propagation times are not exactly known.

C1 can deliver its next clock tick in the r-interval

$I_1 = [(1-\rho)T + t_2 + \mu_{2,1} - x, (1+\rho)T + t_2 + \mu_{2,1} - x]$

C3 delivers its next clock tick in the r-interval

$$I_3 = [(1-\rho)T + t_2 + \mu_{2,3} - x, (1+\rho)T + t_2 + \mu_{2,3} - x]$$



Clock c2 delivers its next clock tick in the r-interval

$I_2 = [(1-\rho)T + t_2, (1+\rho)\mathrm{T} + t_2]$

The clock skew is given by

$$[(1+\rho)T + t_2 + \mu_{max} - x] - (1-\rho)T + t_2 + \mu_{min} - x = 2\rho T + \mu_{max} - \mu_{min}$$

- An alternative to mutual synchronization is to use a master-slave structure.
- The slave clocks try to align themselves to the master clock.
- Sending a read-clock request to the master, which responds with a message containing its clock time when it received this request.

$$T + \frac{r(1+\rho)}{2} - \mu_{min}$$

The error in making this estimate

$$\frac{r(1+\rho)}{2} - \mu_{min}$$

The duration of r, when measured, may be as great as $(1+\rho)r$

$$I' = [T + \mu_{min}(1-\rho), T + (r(1+\rho) - \mu_{min})(1+\rho)]$$

The estimate of the time may be

$$T + \frac{r(1+\rho)^2}{2} - \mu_{min}$$

The estimate error is thus upper-bounded by

$$\frac{r(1+\rho)^2}{2} - \mu_{min} \approx \frac{r(1+2\rho)}{2} - \mu_{min}$$

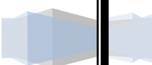
If the messages between master and slave clocks pass over a network that is shared by other traffic,

The slave clock can try to limit the estimation error by simply discarding all the messages

4.5 Fault tolerant synchronisation algorithm in hardware:

- To synchronize in hardware, we can use phase-locked loops.
- The output of the oscillator with an oscillatory signal input.
- The comparator puts out a signal that is proportional to the difference between the phase of the input and that of the oscillator.
- Filter is used to modify the frequency of a voltage controlled oscillator(VCO)

$$v_c(t) = k_c\{\emptyset_i(t) - \emptyset_r(t)\}$$



$\boldsymbol{\emptyset_i(t)}$ the output voltage of the comparator at any time t proportional to the difference between the phase of the signal input

$\boldsymbol{\emptyset_r(t)}$ VCO

k_c is the comparator gain

The Laplace transform of the output is given by

$$v_c(s) = k_c\{\emptyset_i(s) - \emptyset_r(s)\}$$

The output of the filter has the laplace transform

$$V_{VCO}(s) = V_c(s)L(s)$$

The output of the filter is applied to the VCO. The frequency of the ideal VCO signal is given by

$$\omega(t) = \omega_c + k_{VCO}v_{VCO}(t)$$

Limited to the range $[\omega_{min}, \omega_{max}]$. K_{VCO} is called the VCO gain factor.

The laplace transform of the VCO frequency is given by

$$\Omega_r(s) = \frac{\omega_c}{s} + K_{VCO}V_{VCO}(s)$$

The phase of a signal is the integral of its frequency,

$$\emptyset_r(s) = \frac{\Omega(s)}{s} = \frac{\omega_c}{s^2} + \frac{K_{VCO}V_{VCO}(s)}{s}$$

Solving these equations,

$$\emptyset_r(s) = \frac{\omega_c}{s(s + K_c K_{VCO}L(s)} + \frac{K_c K_{vco}L(s)}{s + K_c K_{vco}L(s)}\emptyset_i(s)$$

Laplace transform of the tracking error between the input phase and the VCO output.

$$\emptyset_i(s) - \emptyset_r(s) = \frac{s\Theta_i(s)}{s + K_c K_{VCO}L(s)}$$

4.5.1 Completely connected, zero propagation-time system

PLL has ability to track the input signal and thus synchronise the output with respect to input signal. Thus if we can define a reference input as a function of clock, we can synchronise the clock.

Fig shows the structure of each of the clock. Every clock is connected by dedicated line to every other clock in the system. We assume to begin with signal propagation time are zero.

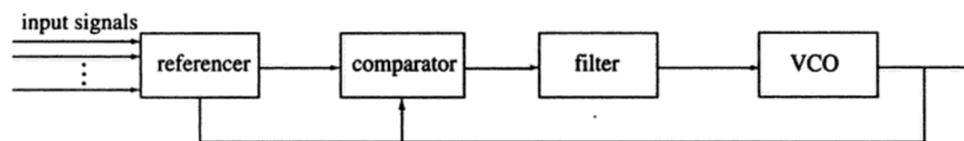
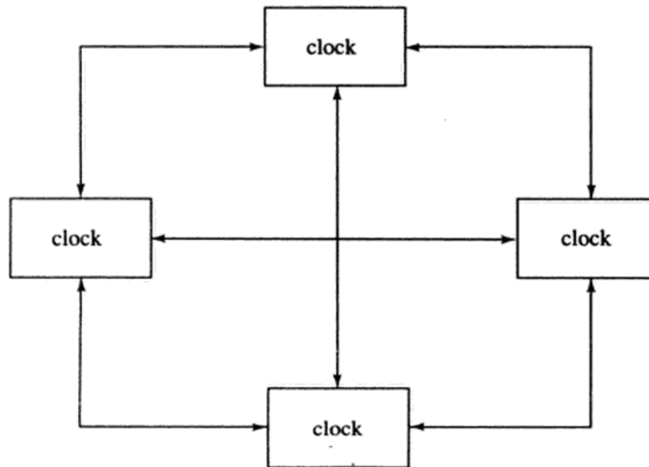


FIGURE 9.14
Structure of a phase-locked loop used in synchronization.

- Each clock has a reference circuit (accept as input the clock ticks)
- Generates a reference signal to which its VCO tries to align itself.

- To make the reference signal equal to the median of the incoming signal



We can define ordered partition G1, G2 of the good clocks with respect to K^{th} clock tick. We group good clock into sets of G_1 and G_2.

C1:- If all the clocks in G_1 use the reference signal that is faster than any clock in G_2. Then there must be at least one clock in G_2 used reference either slower clock in G_1.

C2:- If good clock "X" uses as the reference signal of faulty clock "Y" , then there must exists non faulty clock Z_1 and Z2. Z1may be faster than or slower than Z_2.

Condition of correctness C1, C2

Case (i) $max_{x\in G1} f_{p(x)}(N,m) \leq \left||G1|\right| + m$:

If clock in G1 faster than G2 clock, no reference to any clock outside G1

Case (ii) $min_{y\in G2} f_{p(y)}(N,m) \geq \left||G1|\right| + 1$:

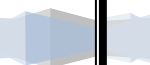
Similar to case(i) this requires that there be at least one clock in G1 whose reference is drawn from G2.

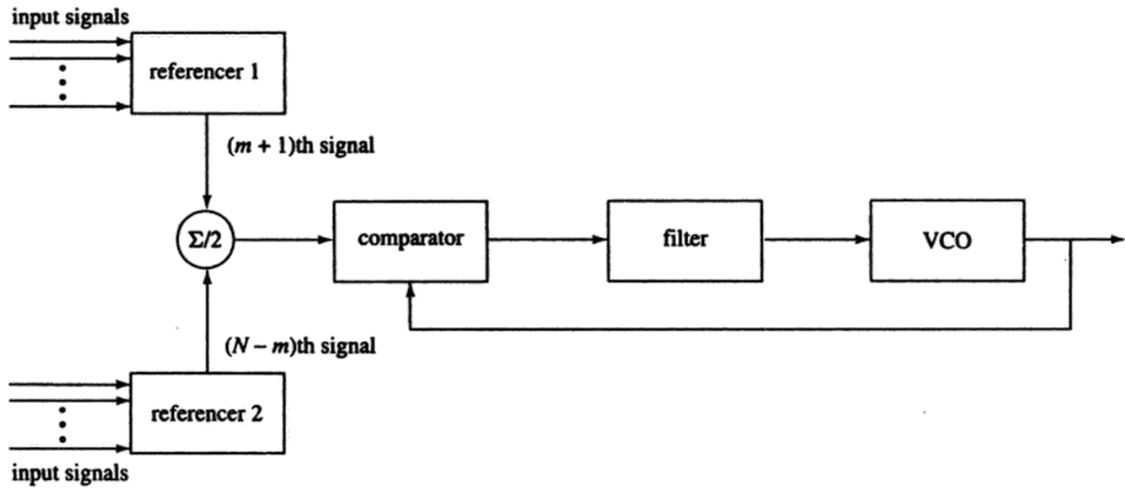
Case(iii) $max_{x\in G1} f_{p(x)}(N,m)$ >||G1|| + m:

No potential exists for the formation of nonoverlapping cliques.

Conditions of correctness thus boil down to the following requirement

- If $max_{x\in G1} f_{p(x)}(N,m) \leq \left||G1|\right| + m$, then $min_{y\in G2} f_{p(y)}(N,m) \leq \left||G1|\right|$.
- Otherwise, if $min_{y\in G2} f_{p(y)}(N,m) \geq \left||G1|\right| + 1$, then $max_{x\in G1} f_{p(x)}(N,m) \geq \left||G1|\right| + m + 1$



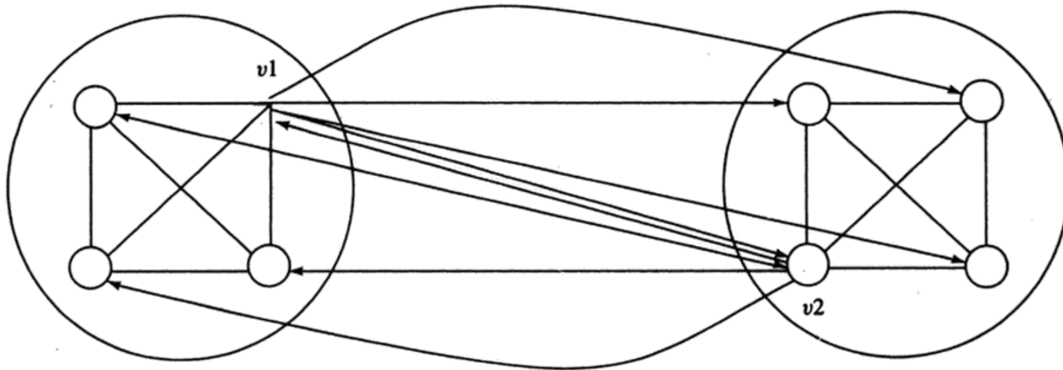


Assumptions:

- That the clocks form a completely connected structure, with each clock having a dedicated line to every other clock
- That the signal propagation times are zero.

4.5.2 Sparse-Interconnection, Zero-propagation times system

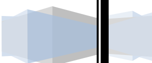
- Instead of a completely connected structure, clocks organized into multiple clusters.
- Each clock in a cluster is connected by a dedicated link to every other clock in that cluster.
- $CL_i \;\; and \; CL_j$ pair of clusters
- Assume signal propagation times are zero
- Each clock can run the phase-locked algorithm
- Failures can increase the skew between clusters.



3m – M + 2$\leq p_i \leq 2M - 2$, for i = 1,2,......,M,

M$\geq m + 1$

For every pair of clusters, there will either be a direct link between two clusters or a link through a third cluster. The skew is no greater than 3δ.



Analysis

- $IN_l = \{CL_s : (s,l) is\ nonfaulty\}$
- $IF_l = \{CL_s : (s,l) is\ faulty\}$
- $ON_l = \{CL_s : (l,s) is\ nonfaulty$
- $OF_l = \{CL_s$: (l, s) is faulty}

4.5.3 Signal propagation delays

- Assume signal propagation times are negligible
- It is true if the geographical extent of the system is not large.
- $\emptyset$ is the nominal clock frequency and θ is the minimum phase difference

Signal propagation delays are negligible if they are less than $\theta/2\pi\emptyset$

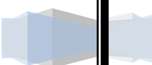
If propagation times are much greater than $\theta/2\pi\emptyset$, design the system to compensate for them.

Large variation in the propagation time between the various clock pairs, failure to correct(result in the formation of multiple non overlapping cliques).

If the connections are point to point and dedicated to transmitting clock pulses, possible to estimate propagation delays.

$$c^k{}_i - c^k{}_j + \frac{d(i,j) - d(j,i)}{2}$$

The output of the average approximations the correct skew.



UNIT V

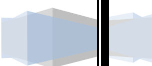
Low Power Embedded System Design: Sources of Power Dissipation–Power Reduction Techniques–Algorithmic Power Minimization–Architectural Power Minimization– Logic and Circuit Level Power Minimization – Control Logic Power Minimization – System Level Power Management.

Power consumption is a very important issue in embedded system design. Many of the embedded applications are built around batteries, and thus, battery life is a critical concern in judging their acceptance. Some of the important issues to consider include power consumption limits, size restrictions, I/O requirements, operational duty cycle, etc. There are several practical considerations that are involved in the process.

1. *Recharging facility:* It is not always possible to recharge/replace the batteries of embedded systems. This is particularly true for systems employed at remote and difficult-to-reach places, for example, a data collection centre for wildlife.
2. *Device size:* Unfortunately, battery technology has not scaled down at the same rate as IC technology. Thus, the weight and size of the device is often guided by the required battery capacity and its associated size. This is a very practical problem for mobile/portable devices.
3. *Duration of operation:* The duration for which a device needs to be active, is another major issue determining the total energy consumption. If the device is idle for majority of the time, a power-down mode may be incorporated into the design to save power.
4. *Power required by the device:* This is the most important issue and needs to be estimated even at the stage of initial design. A better estimation helps in the design process to try out alternatives.
5. *I/O device types:* The I/O interfaces, such as optically isolated I/O and electromechanical relays consume high power. Thus, the system designer needs to avoid them altogether, or minimize their active periods.
6. *Speed of operation:* As we will see later, power consumption is directly proportional to the frequency of operation. The system designer needs to identify the optimum speed for each component, so that the performance target is met, at the same time, individual subsystems are not operated at a speed higher than the required one.

11.1 Sources of Power Dissipation

Since majority of the systems designed are built around CMOS, in this section we will look into the different sources of power dissipation in CMOS. The total power consumed can broadly be divided into *dynamic power* and *static power*.



11.1.1 Dynamic Power Dissipation

This refers to the power consumed by a circuit/system due to some activities within it. For a *static CMOS* realization, at steady-state, either the pull-up (p) or the pull-down (n) network is OFF. Thus, when there are no activities in the system, power consumption is expected to be very low. During circuit activities, the power consumed depend upon the following two mechanisms.

1. *Short-circuit power:* If there is a finite rise and fall time at the input of a CMOS gate, both p- and n-networks are ON simultaneously, shorting the power supply line to the ground. If V_{DD} is the supply voltage and I_{mean} is the average current drawn during the input transition, then the short-circuit power is given by,

$$P_{shortcircuit} = I_{mean} \times V_{DD}$$

   For properly sized and ratioed gates, the contribution to the overall dynamic power due to $P_{shortcircuit}$ is of the order of 10–20%.

2. *Switching power dissipation:* This is the power consumed due to charging and discharging of capacitive loads when the circuit has some activities due to change in inputs. The capacitive load at different circuit gates depends upon the fanout of the gate, output capacitance, and wiring capacitances. It may be noted that a node with load capacitance might not switch when the clock is switching. To take care of this, a quantity called *switching activity* (α) is often used. It determines how often switching occurs on a node with load capacitance. If V_{DD} is the supply voltage, V_{swing} is the change in voltage level of the switched capacitance, C is the capacitance being switched and f is the frequency of operation, the switching power is given by,

$$P_{switching} = C \times V_{DD} \times V_{swing} \times \alpha \times f$$

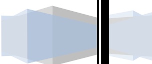
   Since in most of the cases, $V_{swing} = V_{DD}$,

$$P_{switching} = C \times V_{DD}^2 \times \alpha \times f$$

11.1.2 Static Power Dissipation

This is the power consumed when the circuit is not in active mode of operation. In such a situation, there is still some power dissipation due to various leakage mechanisms. The situation is aggravated with the scaling of supply voltages. As the supply voltage is reduced, to keep the delay of a gate unchanged, the transistors need to be turned ON early by reducing their threshold voltages. This, in turn, increases the leakage current in the subthreshold range of operation of the circuit. Due to the exponential nature of leakage current in the subthreshold regime of the transistor, it can no longer be ignored. The *International Technology Roadmap for Semiconductors* (ITRS) has projected an exponential increase in leakage power with minimization of devices. Also, leakage increases with temperature. Thus, the increased heat dissipation resulting from increase in leakage power consumption has a positive feedback on leakage. There are three major leakage mechanisms—*subthreshold leakage*, *gate direct tunneling* and *junction band-to-band tunneling*. This has been shown in Fig. 11.1.

- *Subthreshold leakage:* When gate voltage is below threshold voltage but very close to it, subthreshold conduction current flows between source and drain. It is caused by



the diffusion of minority carriers. It depends exponentially on the threshold voltage of the transistor. In nano-scaled devices, *short channel effect* (SCE) reduces the threshold voltage, thereby increasing the subthreshold current.

- *Gate direct tunneling leakage:* Due to ultra-thin gate oxide, a high electric field can cause electrons to tunnel through the gate-oxide. This results in large gate leakage in nano-scale transistors. An increase in the supply voltage and/or reduction in oxide thickness results in an exponential increase in gate tunneling current.
- *Junction band-to-band tunneling leakage:* Application of reverse bias across the highly doped p-n junction results in tunneling of electrons from valence band of p-side to the conduction band of n-side. This is called *band-to-band tunneling.* In nano-scale devices, due to the use of high junction doping, large junction BTBT occurs at OFF state with the drain at V_{DD} and substrate at ground. The junction BTBT increases exponentially with an increase in junction doping and supply voltage.

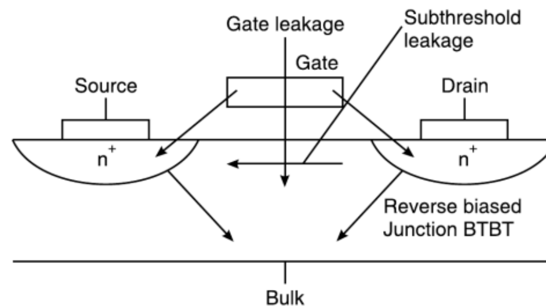


Fig. 11.1 Leakage components in a transistor.

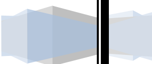
11.2 Power Reduction Techniques

Power reduction can be attempted at all levels of design hierarchy—*algorithm, architecture, logic, and device* levels. In the following sub-sections we give an overview of each of these (except the device level on which the embedded system designer often does not have any control). Higher the level at which power minimization is addressed, higher is the expected power saving.

11.2.1 Algorithmic Power Minimization

It mainly focuses on reducing the number of operations requiring larger power in a target implementation. For example, in many processors, the cost of an addition/subtraction operation may be different from a logical operation. Thus, to check "*whether x is equal to y*", one may first perform a subtraction operation followed by checking the status register for zero-bit. On the other hand, if the logical operation takes lesser power, x may be directly compared with y using a comparison instruction. The following are some of the important issues to be judged for selecting a particular algorithm from alternatives:

1. *Memory reference:* This is very important as memory is normally off-chip from the processor. A large number of accesses to the memory mean good amount of activity in



the address/data bus lines. The memory access pattern is also important. If the access pattern is sequential, only the least significant bits of address bus change, whereas for random access through the memory, most of the address bits will switch, thus creating higher power dissipation.

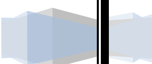
2. *Presence of cache memory:* The presence and structure of cache memory plays an important role. Cache can be fruitfully utilized to reduce both execution time and power of an implementation if the underlying algorithm has got *locality* in its behaviour. The locality may be both *temporal* and *spatial* in nature. While a temporal locality refers to the fact that a memory location accessed at some time is also likely to be accessed in near future, spatial locality means if a memory location is accessed at some time, its neighbouring locations are also likely to be accessed in near future. Thus, caching them inside the CPU cache saves not only the memory access time, but also the bus energy consumption is reduced.
3. *Recomputation vs. memory load/store:* Normal power minimization techniques at algorithm level attempt to reduce the number of arithmetic operations. However, it may so happen that to reduce the number of operations, some repeatedly performed computation is done only once and stored at a memory location. Later, as and when necessary, it is reloaded from the memory. This may lead to increased power consumption due to extra memory accesses. If the operands are already available in CPU registers or on-chip cache, it may be better to recompute the value, instead of loading it from memory, from power consumption point of view.

4. *Compiler optimization technique:* The typical techniques used by an optimizing compiler can be used to reduce power consumption of a piece of code. The strategies involve *strength reduction, common subexpression elimination, minimizing memory traffic* etc. *Loop unrolling* is also often beneficial as it reduces loop overhead.
5. *Number representation:* This is another area for algorithmic power trade-off. The following points may be noted:
   - *Fixed vs. floating point representation:* Fixed point operations are much simpler than floating point ones. Thus, it normally leads to power saving, though accuracy may suffer.
   - *Sign-magnitude vs. 2's complement:* Selection of sign-magnitude representation may have significant power saving over 2's complement, if input samples are uncorrelated and range is minimized.
   - *Precision of operations:* This is important, since having lower precision allows one to reduce the size of space needed to store the values. A typical example of this is to reduce the number of bits in mantissa portion in several signal processing applications including speech and image to improve circuit delay and power.

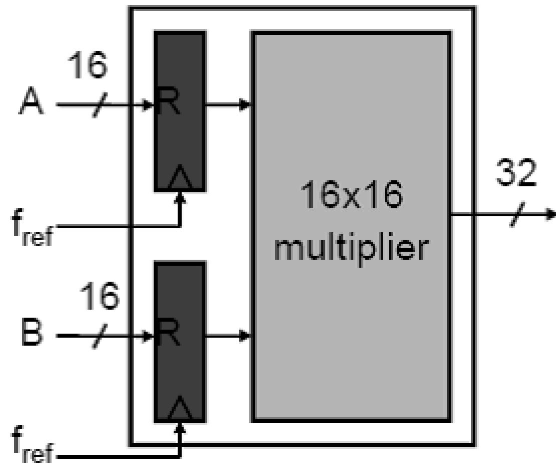
11.2.2 Architectural power minimization:

The architectural power minimization generally used power saving purposes. It reduces unwanted circuit that present in the system. The two generic technique to save power are as follows:

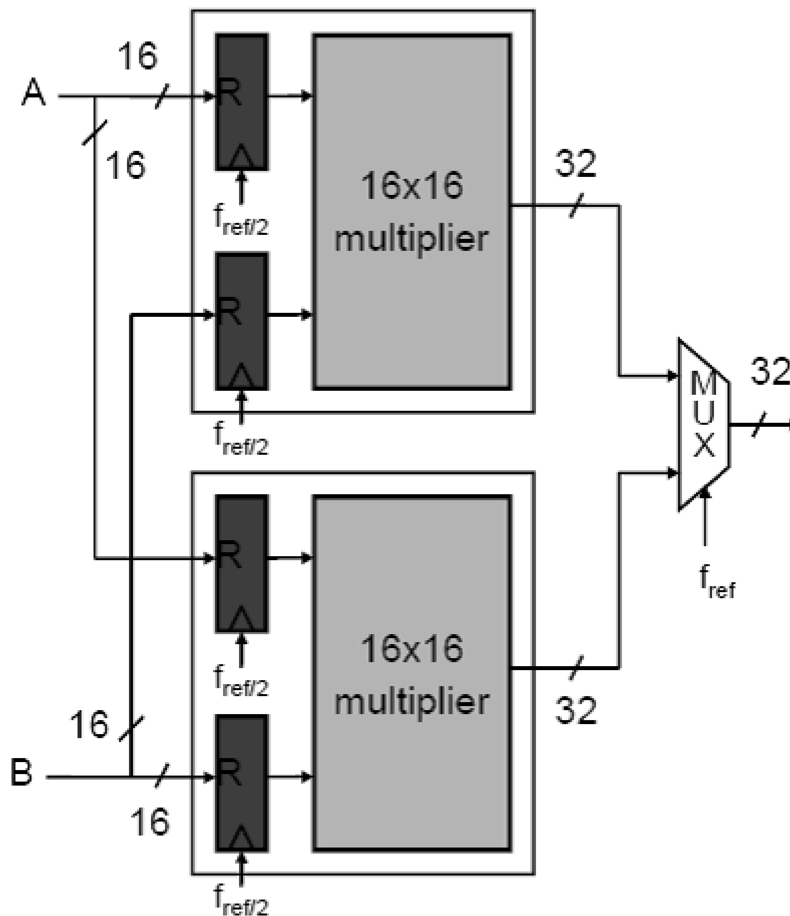
- Parallelism
- Pipelining

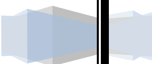


Parallelism:
Assume that the 16 x 16 multiplier, the power supply can be reduced from v_{ref} to $v_{ref}/1.83$.



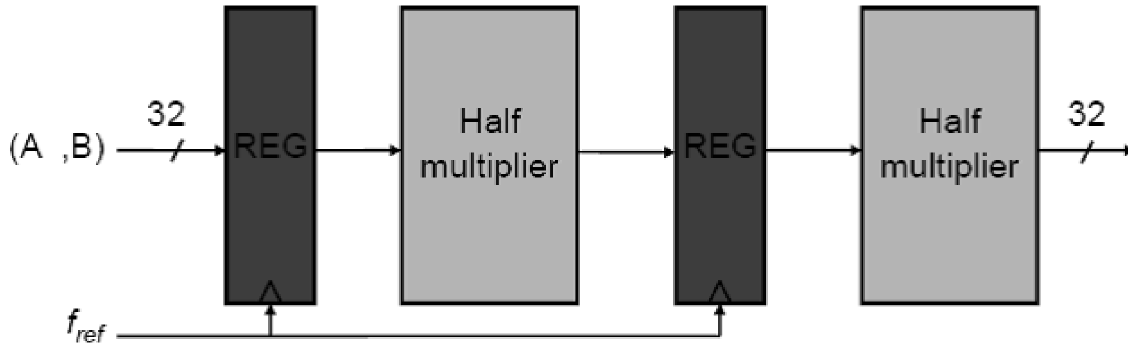
$$P_{parallel} = 2.2C_{ref}\left(\frac{V_{ref}}{1.83}\right)^2 \frac{f_{ref}}{2} = 0.33P_{ref}$$





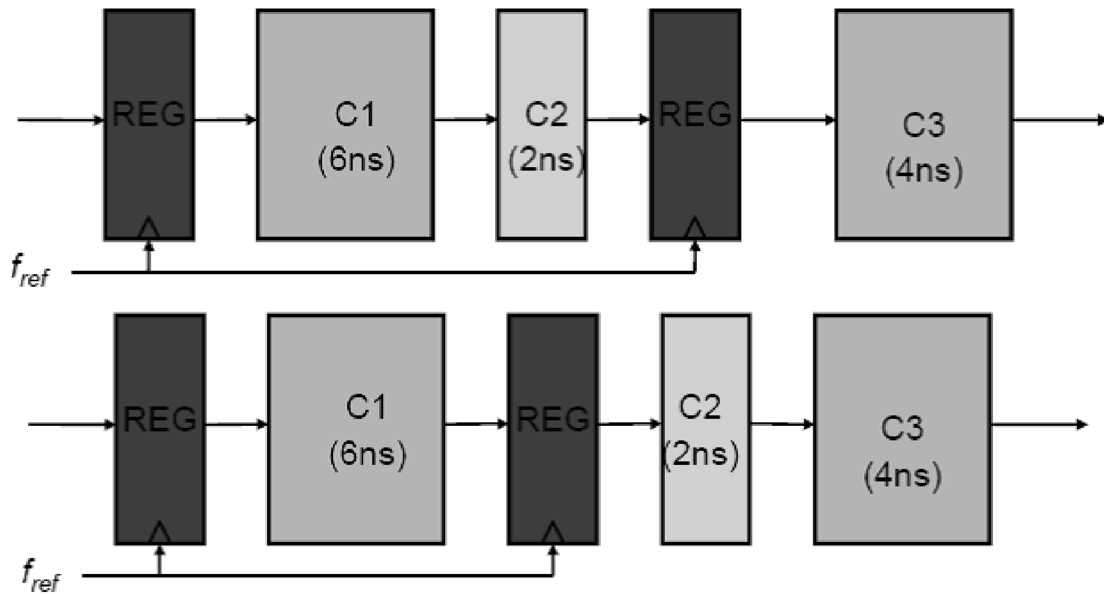
Pipelining:

The hardware between the pipeline stages is reduced then the reference voltage Vref can be reduced to Vnew to maintain the same worst case delay. For example, let a 50MHz multiplier is broken into two equal parts as shown below. The delay between the pipeline stages can be remained at 50MHz when the voltage Vnew is equal to Vref/1.83.



$$P_{pipeline} = 1.2C_{ref}(\frac{V_{ref}}{1.83})^2 f_{ref} = 0.36P_{ref}$$

Retiming for pipeline design



11.2.3 Logic and Circuit Level Power Reduction Techniques

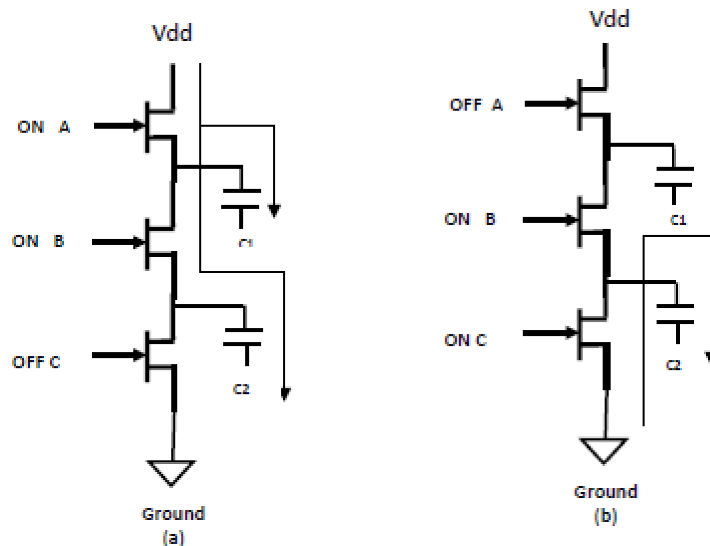
Transistor Sizing

Transistor sizing reduces the width of transistors to reduce their dynamic power consumption, but reducing the width also increases the transistor's delay; hence the transistors that lie away from the critical paths of a circuit are usually the best suited for this technique.

Algorithms for applying this technique usually associate with each transistor a tolerable delay, which tries to scale each transistor to be as small as possible without violating its tolerable delay.

Transistor Reordering

The arrangement of transistors in a circuit affects energy consumption. Figure shows two possible implementations of the same circuit that differ only in their placement of the transistors marked A and B. Suppose that the input to transistor A is 1, for B is 1 and for C is 0. Then transistors A and B will be on, allowing current from Vdd to flow through them and charge the capacitors C1 and C2.

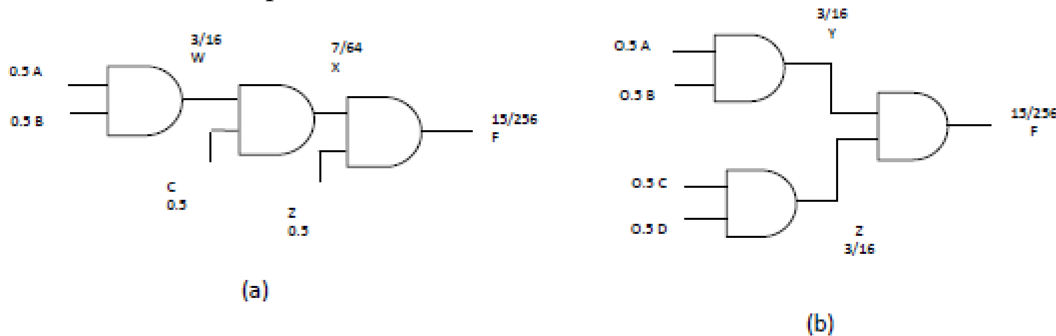


Now, suppose the inputs change and that A’s input becomes 0, and for C it is 1. Then A will be off while B and C will be on. Now the implementations in (a) and (b) will differ in the amounts of switching activity. In (a), current from ground will flow through B and C, discharging both the capacitors C1 and C2. However, in (b), the current from ground will only flow through C2 it will not pass through A since A is turned off. Thus it will only discharge the capacitor C2, rather than both C1 and C2 as in part (a). Thus the implementation in (b) will consume less power than that in (a). Transistor reordering rearranges transistors to minimize their switching activity.

Logic Gates Restructuring

There are many ways to connect a circuit using logic gates but the way the gates and their signals are connected affects power consumption. Consider two implementations of a four-input AND gate shown in Figure 3.2 with signal probabilities (1 or 0) at each of the primary inputs (A,B,C,D) with the transition probabilities (0→1) for each output (W, X, F, Y, Z). If each input has an equal probability of being a 1 or a 0, then the calculation shows that implementation (a) is likely to switch less than the implementation (b). This is because each gate in (a) has a lower probability of having a 0→1transition. In (b) some gates may share a parent (in the tree topology) instead of being directly connected together. These gates could have the same transition probabilities. The circuit (a) do not necessarily save more energy than Circuit (b).

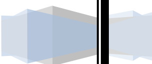
There may be many other issues such as glitches or spurious transitions which occur when a gate does not receive all of its inputs at the same time.



11.2.4 Control Logic Power Minimization

Power optimization of controller circuitry is very important. This is because while the datapath of a design can be selectively turned OFF when not in use, controller is always active. A controller is often specified and realized as a *Finite State Machine* (FSM). The synthesis of FSM targeting low power can be addressed from various angles as discussed next.

1. *Storage element design:* Several flip-flop design have been reported in the literature with various area-power trade-offs. Detailed discussion on this is beyond the scope of this book.
2. *State assignment:* This is the assignment of binary codes to the FSM states to realize it. For low power state encoding, first the steady-state probability for each of the states is determined. Next, the transition probabilities between the states are calculated. Codes with lesser Hamming distance are allocated to the states having higher transition probabilities between them. This minimizes the number of transitions in the next-state and output combinational logic.
3. *FSM partitioning:* Often it is the case that the FSM states form clusters. Once the FSM is in some state belonging to a cluster, the probability of its remaining within the states of this cluster for quite a few transitions is high. Probabilities of inter-cluster transitions are low. Thus, the FSM can be partitioned into two or more sub-FSMs. At any point of time, only one sub-FSM is active. We can do two things with the remaining submachines. The first alternative is to stop clocks to them and make sure that their primary input lines are masked-off. This is commonly known as *clock gating*. The scheme has been shown in Fig. 11.5. Thus, there is no dynamic power consumption in these sub-FSMs. The other alternative is to use *power gating* by introducing *sleep transistors* to turn OFF power supply to them. Figure 11.6 shows introduction of sleep transistors. Power gating reduces leakage power as well, however, wake-up takes time. Thus, either the circuit performance suffers, or we have to turn ON probable active sub-FSMs early—this leads to wastage of power as more than one sub-FSMs are active.



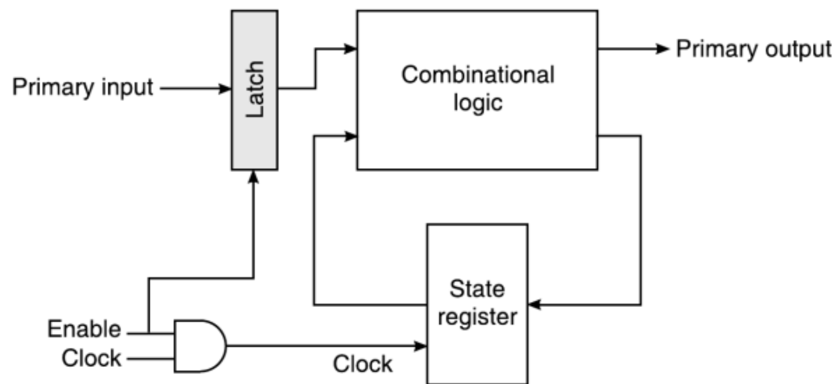


Fig. 11.5 Clock gating of FSM.

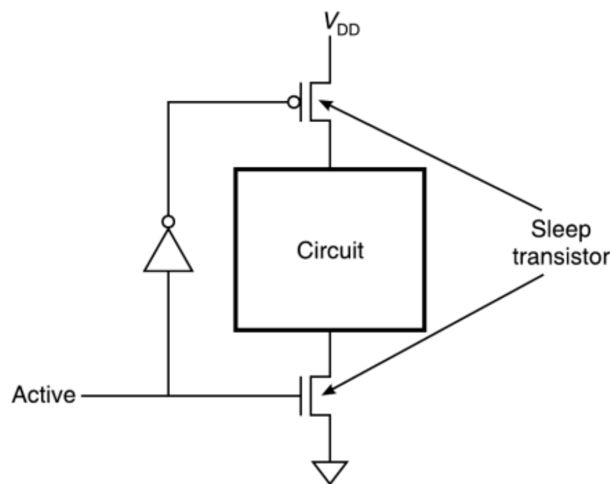


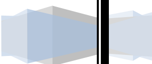
Fig. 11.6 Power gating of FSM.

11.3 System Level Power Management

The techniques discussed so far are good to the extent that they deal with local subsystems. In an embedded system, consisting of various subsystems, it may be possible that all of them are not needed simultaneously to remain active. Thus, it needs a system level management policy to turn OFF and turn ON the subsystems. A suitable power management policy is needed for the following purposes:

- going to a low power state takes time. The longer the duration for which we want to shutdown a system, higher is the time taken during reactivation.
- avoiding a power-down mode will cost unnecessary power.
- frequent power-down mode will affect system performance.

A naive approach may be to power-down a system whenever there is no request. This will definitely affect performance severely. A more sophisticated method is to use *predictive shutdown*. In this approach, the goal is to predict the next arrival of service request and wake up the system just before that. Prediction can be made in several different ways as follows.



1. *Fixed times:* If the system does not receive any service request during an interval of length T_{ON}, it shuts down for a fixed period of time T_{OFF}. Choice of T_{ON} and T_{OFF} may be made experimentally by studying system behaviour.
2. *Analysing system state:* In this approach, there is a constant monitoring of the service requests. The monitoring is done via a *power manager* that observes the system components—the *service provider*, the *service requestor* and the *queue* between them. Based on the observations, the power manager sends power management commands to the service provider. The situation has been shown in Fig. 11.7.

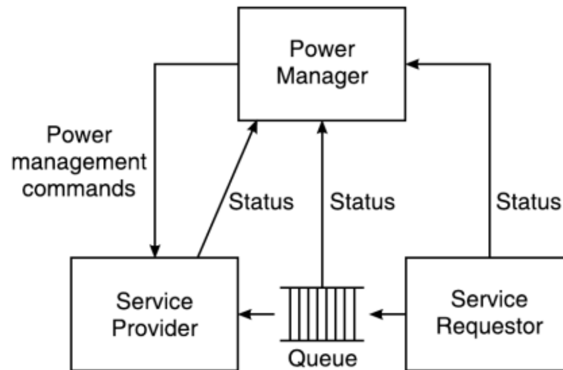


Fig. 11.7 Power analysis strategy.

11.3.1 Advanced Configuration and Power Interface (ACPI)

ACPI is an open industry standard for power management services. It is designed to be compatible with a wide variety of operating systems. Initially targeted at *Personal Computers*, ACPI provides some basic power management facilities and abstracts the hardware layer. Host operating system will have its own power management module that determines the policy. The operating system utilizes ACPI to send the required controls to the ACPI compliant hardware and observe the hardware's state as input to the power manager. The concept has been shown in Fig. 11.8.

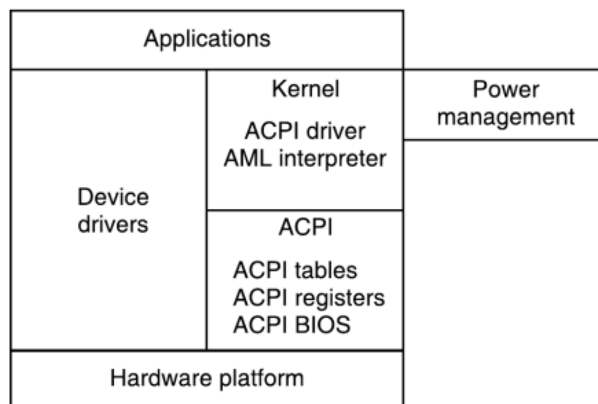
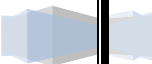


Fig. 11.8 ACPI interface.



ACPI supports five basic global power states, as follows:

- *G3:* The mechanical OFF state, in which system consumes no power.
- *G2:* The soft OFF state, which requires a full operating system reboot to restore the machine to working condition. It has four substates:
  - *S1:* a low wake-up latency state with no loss of system context.
  - *S2:* a low wake-up latency state with a loss of CPU and system cache state.
  - *S3:* a low wake-up latency state in which all system state except the main memory is lost.
  - *S4:* the lowest power sleeping state, in which all devices are turned off.
- *G1:* The sleeping state in which the system appears to be off, and the time required to return to working condition is inversely proportional to power consumption.
- *G0:* The working state, in which the system is fully usable.
- *The legacy state:* The system does not comply with ACPI.

The power manager typically includes an observer that receives messages through the ACPI interface that describe the system behaviour. It also includes a decision module that determines the power management actions based on those observations.