# UNIT-I

# INTRODUCTION TO EMBEDDED SYSTEMS

# Definition of embedded system

## INTRODUCTION

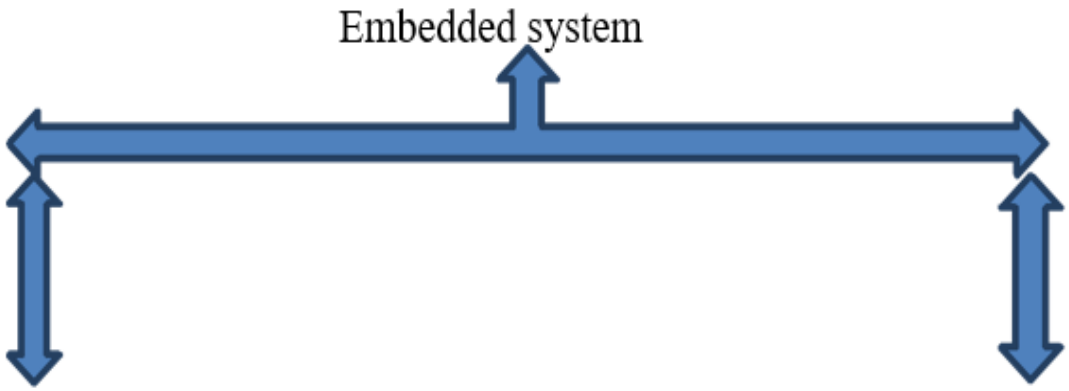
- An embedded system is a combination of computer hardware and software, either fixed in capability or programmable, designed for a specific function or functions within a larger system.
- An embedded system is an electronic system, which includes a single chip microcomputers (Microcontrollers) like the ARM or Cortex or Stellaris LM3S1968.
- **Examples** of **embedded** systems include washing machines, printers, automobiles, cameras, industrial machines and more.

# CHARACTERISTICS OF AN EMBEDDED SYSTEM

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

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

# Categories of embedded systems




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

# Embedded system VS Generalpurpose system

**Embedded System VS General Purpose System**

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

# Embedded system VS Generalpurpose system

## General Purpose Systems

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

# Embedded system VS Generalpurpose system

## Embedded Systems:

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

# HISTORY OF AN EMBEDDED SYSTEMS

- Embedded systems date back to the 1960s.
- Charles Stark Draper developed an integrated circuit (IC) in 1961 to reduce the size and weight of the Apollo Guidance Computer, the digital system installed on the Apollo Command Module and Lunar Module.
- The first computer to use ICs, it helped astronauts collect real-time flight data.
- In 1968, the first embedded system for a vehicle was released; the Volkswagen 1600 used a microprocessor to control.
- 1987, the first embedded operating system, the real-time VxWorks, was released by Wind River, followed by Microsoft's Windows Embedded CE in 1996.
- By the late 1990s, the first embedded Linux products began to appear. Today, Linux is used in almost all embedded devices.

# Classification of embedded system

**Embedded systems can be classified into three different types:**

small scale embedded systems, medium scale embedded systems, and sophisticated embedded systems.

1) **Small scale embedded systems**: These usually use 8 bit or 16 bit microcontrollers, and have minimum hardware and software. They are so small and require little power they may be powered by a battery.

2**) Medium scale embedded systems**: They use either one or a few 16 bit or 32 bit microcontrollers, and hardware and software is more complex as compared to small scale embedded systems.

3) **Sophisticated embedded systems**: The most complex of the three classifications mentioned.

# Major applications of Embedded systems

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

# Characteristics and Quality Attributes of Embedded Syste

**Characteristics of embedded system:**

- Application and Domain specific
- Reactive and Real time
- Operation in harsh environment
- Distributed
- Small size and weight
- Power concerns

# Characteristics and Quality Attributes of Embedded Syste

**QUALITY ATTRIBUTES OF EMBEDDED SYSTEM**

There are two types of quality attributes are:-

1. Operational Quality Attributes.
   - Response
   - Throughput
   - Reliability
   - Maintainability
   - Scheduled or Periodic Maintenance
   - Maintenance to unexpected failure
   - Security and Safety

2. Non-Operational Quality Attributes.
   - Testability and Debug-ability
   - Evolvability
   - Portability
   - Time to prototype and market
   - Per unit and total cost

# UNIT-II

# Typical Embedded System

# General Purpose and Domain Specific Processors

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

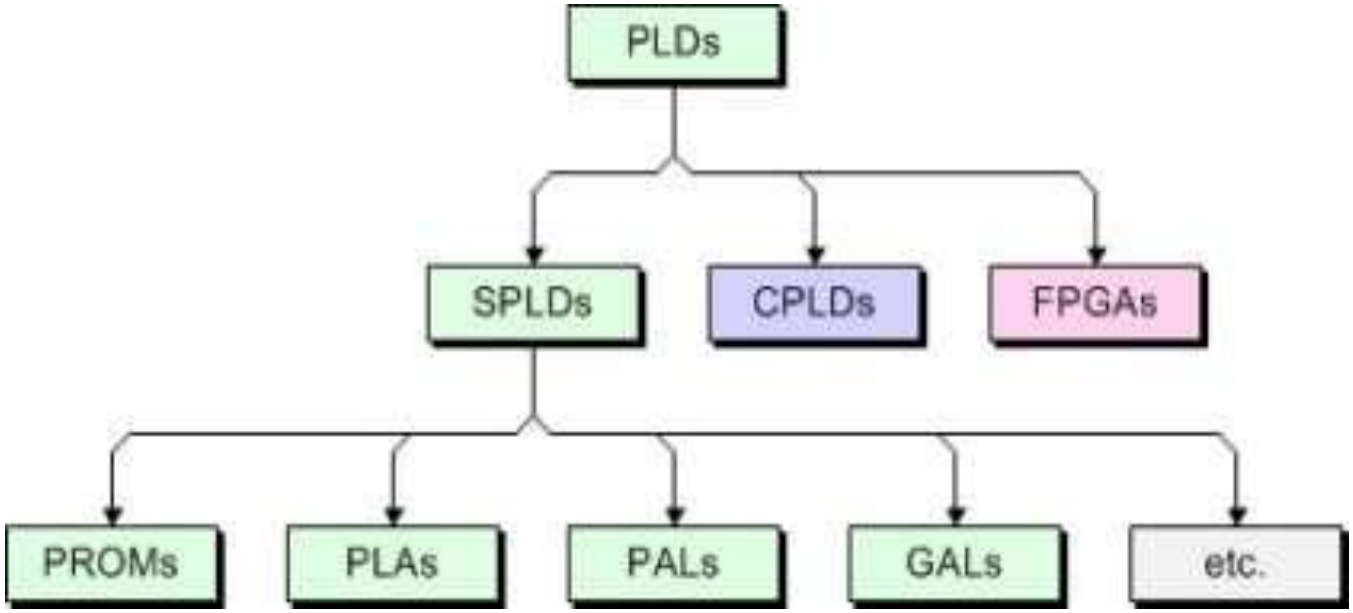
- **Generally a DSP processor** has separate program and data memories. This allows the processor to fetch an instruction, while simultaneously fetching operands or storing results for a previous instruction. Often it is also possible to fetch multiple data from memory in one clock cycle by using multiple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

# ASICs

An application-specific integrated circuit is an integrated circuit (IC) customized for a particular use, rather than intended for general-purpose use. For example, a chip designed to run in a digital voice recorder or a high-efficiency bitcoin miner is an ASIC. Application-specific standard products (ASSPs) are intermediate between ASICs and industry standard integrated circuits like the 7400 series or the 4000 series

# PLDs

A programmable logic device is an electronic component used to build reconfigurable digital circuits. Unlike integrated circuits which consist of logic gates and have a fixed function, a PLD has an undefined function at the time of manufacture



# Commercial Off-The-Shelf Components (COTS)

Short for **commercial off**-the-**shelf**, an adjective that describes software or hardware **products** that are ready-made and available for sale to the general public. For example, Microsoft Office is a **COTS** product that is a packaged software solution for businesses

# Memory: ROM

Read-only memory (ROM) is a type of non-volatile memory used in computers and other electronic devices. Data stored in ROM cannot be electronically modified after the manufacture of the memory device. Read-only memory is useful for storing software that is rarely changed during the life of the system, sometimes known as firmware. Software applications for programmable devices can be distributed as plug-in cartridges containing read-only memory.

# Memory:

There are two main kinds of semiconductor memory

- Volatile
- Non volatile

- Examples of non-volatile memory are flash memory (used as secondary memory) and ROM, PROM, EPROM and EEPROM memory (used for storing firmware such as BIOS).

- Examples of volatile memory are primary storage, which is typically dynamic random-access memory (DRAM), and fast CPU cache memory, which is typically static random-access memory (SRAM) that is fast but energyconsuming, offering lower memory areal density than

# RAM

- Random-access memory is a form of computer memory that can be read and changed in any order, typically used to store working data and machine code. A random-access memory device allows data items to be read or written in almost the same amount of time irrespective of the physical location of data inside the memory.

- RAM contains multiplexing and demultiplexing circuitry, to connect the data lines to the addressed storage for reading or writing the entry. Usually more than one bit of storage is accessed by the same address, and RAM devices often have multiple data lines and are said to be "8-bit" or "16-bit", etc. devices.

# Memory according to the type of Interface,

DSP processor data paths are optimized to provide extremely high performance on certain kinds of arithmetic-intensive algorithms. However, a powerful data path is, at best, only part of a high-performance processor. To keep the data path fed with data and to store the results of data path operations, DSP processors require the ability to move large amounts of data to and from memory quickly.

DSP processor data paths are designed to perform a multiply-accumulate operation in one instruction cycle. This means that the arithmetic operations required for one tap can be computed in one instruction cycle. However, to achieve this performance, the processor must be able to make several accesses to memory within one instruction cycle. Specifically, the processor must:

- fetch the multiply-accumulate instruction,
- read the appropriate data value from the delay line,
- read the appropriate coefficient value, and
- write the data value to the next location in the delay line to shift data through the delay line.

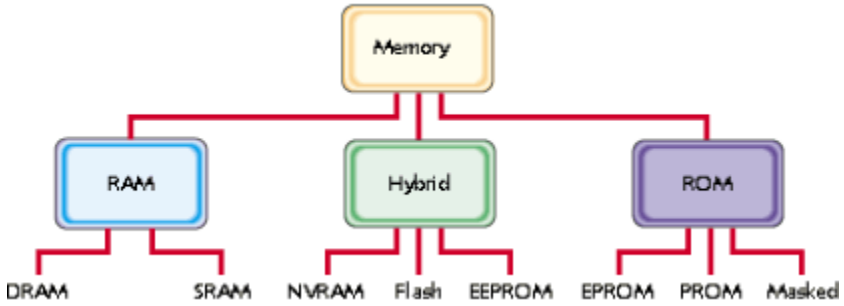
Thus, the processor must make four accesses to memory in one instruction cycle if the multiply-accumulate operation is to execute in a single instruction cycle. In practice, some processors use other techniques (as discussed later) to reduce the actual number of memory accesses needed to three or even two. Nevertheless, all processors require multiple memory accesses within one instruction cycle. This level of memory bandwidth is needed for many important DSP algorithms.

# Memory Shadowing

Shadow memory is a technique used to track and store information on computer memory used by a program during its execution. Shadow memory consists of shadow bytes that map to individual bits or one or more bytes in main memory. These shadow bytes are typically invisible to the original program and are used to record information about the original piece of data.

# Memory selection for Embedded Systems

- Many types of memory devices are available for use in modern computer systems.
- As an embedded software engineer, you must be aware of the differences between them and understand how to use each type effectively.
- The names of the memory types frequently reflect the historical nature of the development process and are often more confusing than insightful.

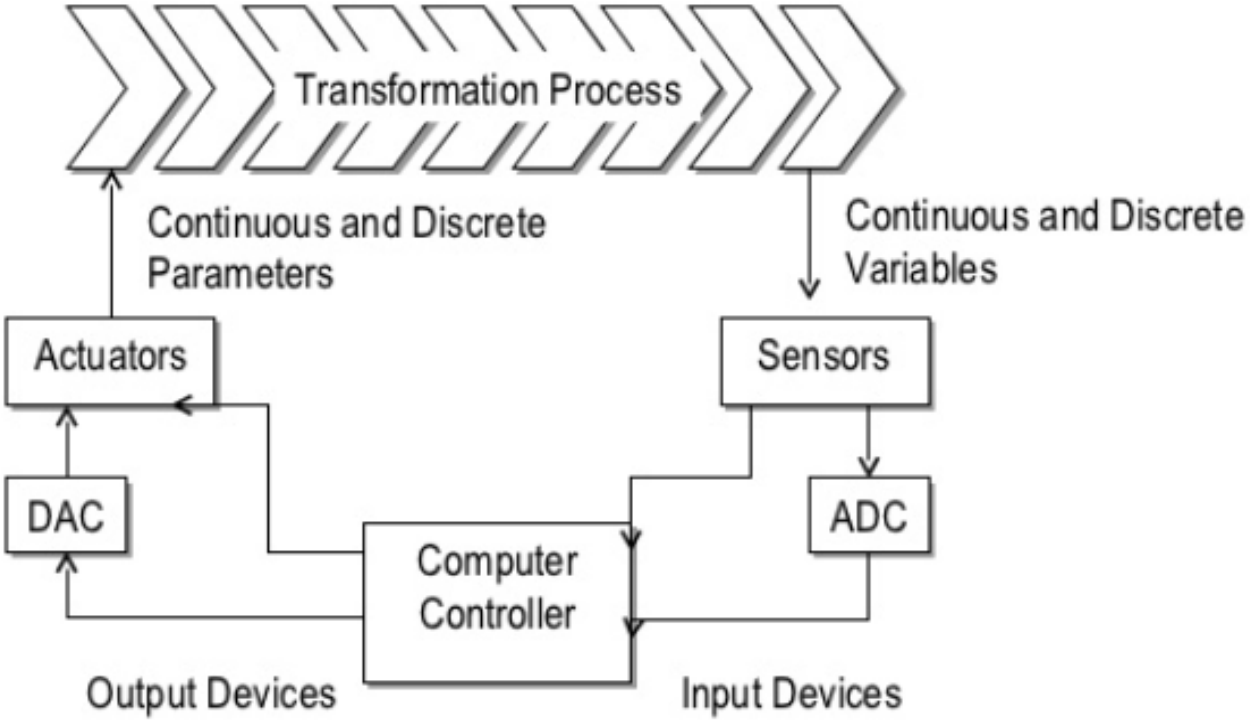


# Sensors and Actuators

- A **sensor** is a device that changes a physical parameter to an electrical output. As against, an **actuator** is a device that converts an electrical signal to a physical output. ... **Sensor** generates electrical signals while an **actuator** results in the production of energy in the form of heat or motion

- Sensors and Actuators are essential elements of the embedded systems. These are used in several real-life applications such as flight control system in an aircraft, process control systems in nuclear reactors, power plants that require to be operated on an automated control. Sensors and Actuators mainly differ by the purpose both provide, the sensor is used to monitor the changes in the environment by using measurands while the actuator is used when along with monitoring the control is also applied such as to control the physical change

# Sensors and Actuators



# Sensors

- Sensor is transducer that converts physical stimulus from one form into a more useful form to measure stimulus.

Temperature
RFID
Barcode
Proximity
Vision
Gyroscope
Compass
Tilt/Acceleration
Etc.

# ACTUATORS

Actuators that convert the controller command signal into a change in a physical parameter.. the change is usually mechanical..(e.g..position or velocity..)

Electrical actuators

- Electric motors
  - DC servomotors
  - AC motors
  - Stepper motors
- Solenoids

Hydraulic actuators

- Use hydraulic fluid to amplify the controlle command signal

Pneumatic actuators

- Use compressed air as the driving force

# Communication Interface: Onboard and External Communication Interfaces.

- **Onboard communication Interface.** The communication channel which interconnects the various components within an embedded product is referred as device/broad level communication interface. The product level communication interface is responsible for data transfer between the E.S and other devices or modules

- SoCs (System on a Chip) include **external interfaces**, typically for communication protocols. These are often based upon industry standards such as USB, FireWire, Ethernet, USART, SPI, HDMI, $I^2C$, etc. These interfaces will differ according to the intended application. Wireless networking protocols such as Wi-Fi, Bluetooth, 6LoWPAN and near-field communication may also be supported.

- When needed, SoCs include analog interfaces including analog-to-digital and digital-to-analog converters, often for signal processing. These may be able to interface with different types of sensors or actuators, including smart transducers. They may interface with application-specific modules or shields Or they may be internal to the SoC, such as if an analog sensor is built in to the SoC and its readings must be converted to digital signals for mathematical processing.

# UNIT-III

# EMBEDDED FIRMFARE

# Embedded firmware:

- Embedded firmware is the flash memory chip that stores specialized software running in a chip in an embedded device to control its functions.

- Firmware in embedded systems fills the same purpose as a ROM but can be updated more easily for better adaptability to conditions or interconnecting with additional equipment.

- For example, embedded software may run on ROM chips. Also, embedded software is often the only computer code running on a piece of hardware while firmware can also refer to the chip that houses a computer's EFI or BIOS, which hands over control to an OS that in turn launches and controls programs.

# Reset circuit:

➢The reset circuit is essential to ensure that the device is not operating at a voltage level where the device not guaranteed to operate. During system power ON. The reset signal brings the internal registers and the different hardware system of the processor/controller to a known state and starts the firmware execution from the reset vector.

➢The reset signal can be either active high (the processor undergoes reset when the reset pin of the processor is at logic high) or active low(the processor undergoes reset when the pin of the processor is at logic low).since the processor operation is synchronized to a clock signal, the reset pulse should be wide enough to give time for the clock oscillator to stabilize before the internal reset state starts.

➢Some microprocessors/controllers contain built in internal reset circuitry and they don't require external reset circuitry. Figure illustrates a resistor capacitor based passive reset circuit for active high and low configurations. The reset pulse width can be adjusted by changing the resistance value R and capacitance value C.

# RC Based reset circuit

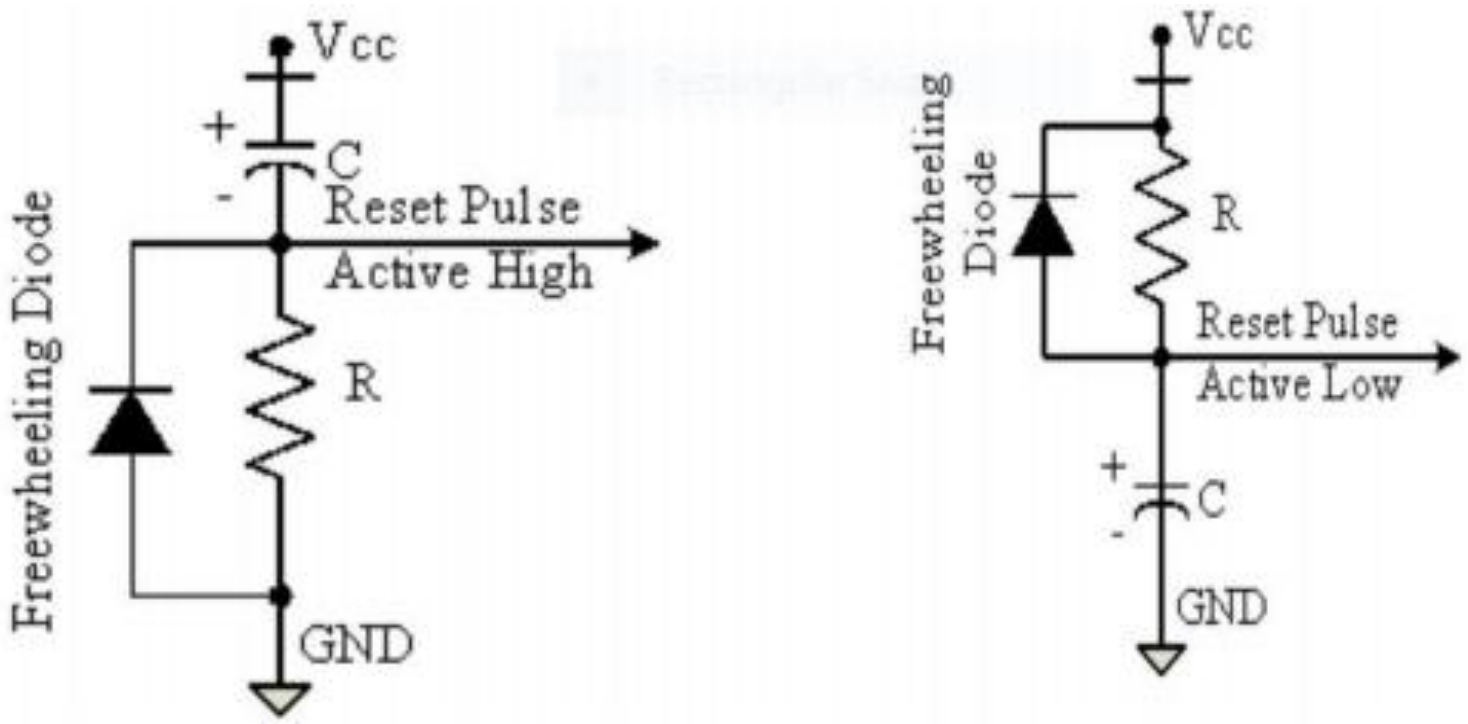



**Figure : RC based Reset circuit**

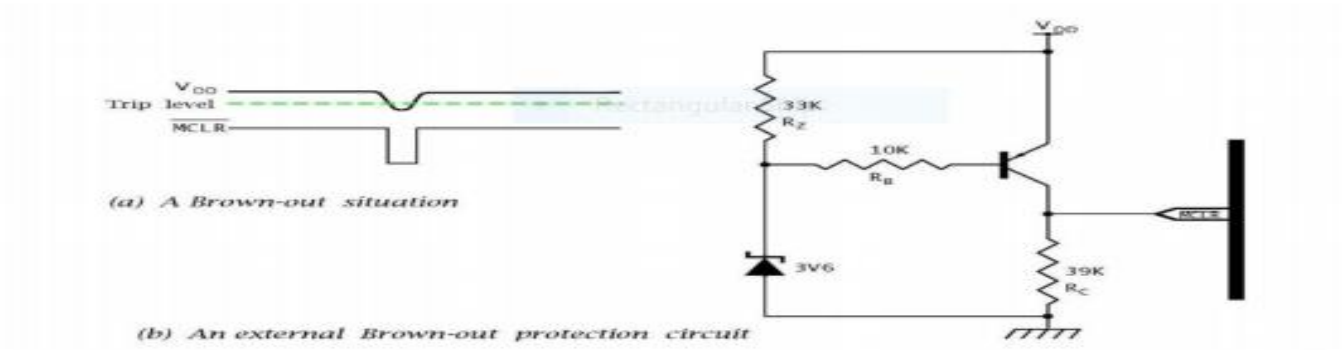
# Brown-out Protection Circuit:

- Brown-out protection circuit prevents the processor/controller from unexpected program execution behavior when the supply voltage to the processor/controller falls below a specified voltage.

- The processor behavior may not be predictable if the supply voltage falls below the recommended operating voltage. It may lead to situations like data corruption.

- A brown-out protection circuit holds the processor/controller in reset state, when the operating voltage falls below the threshold, until it rises above the threshold voltage.

# External Brown out protection circuit




- Certain processors/controllers support built in brown-out protection circuit which monitors the supply voltage internally
- If the processor/controller doesn't integrate a built-in brown-out protection circuit, the same can be implemented using external passive circuits or supervisor ICs
- The zener diode D and transistor Q forms the heart of the circuit. The transistor conducts always when the supply voltage VCC is greater than that of the sum of VBE and VZ.
- The transistor stops conducting when the supply voltage falls below the sum of VBE and VZ. The values of the resistors can be selected based on the electrical characteristics.

# Oscillator Unit:

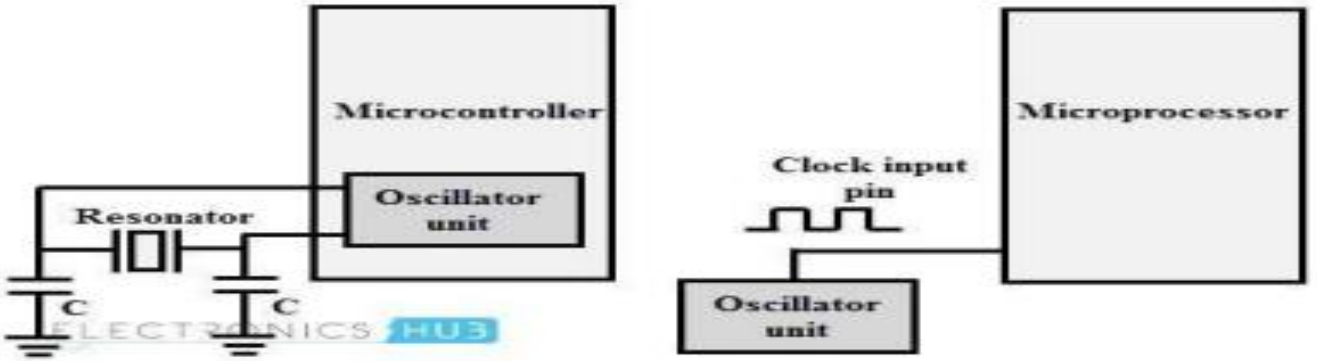
- A microprocessor/microcontroller is a digital device made up of digital combinational and sequential circuits .

- The instruction execution of a microprocessor/controller occurs in sync with a clock signal.

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

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

# OSCILLATOR:




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

# Real Time Clock (RTC):

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

# Watch Dog Timer (WDT):

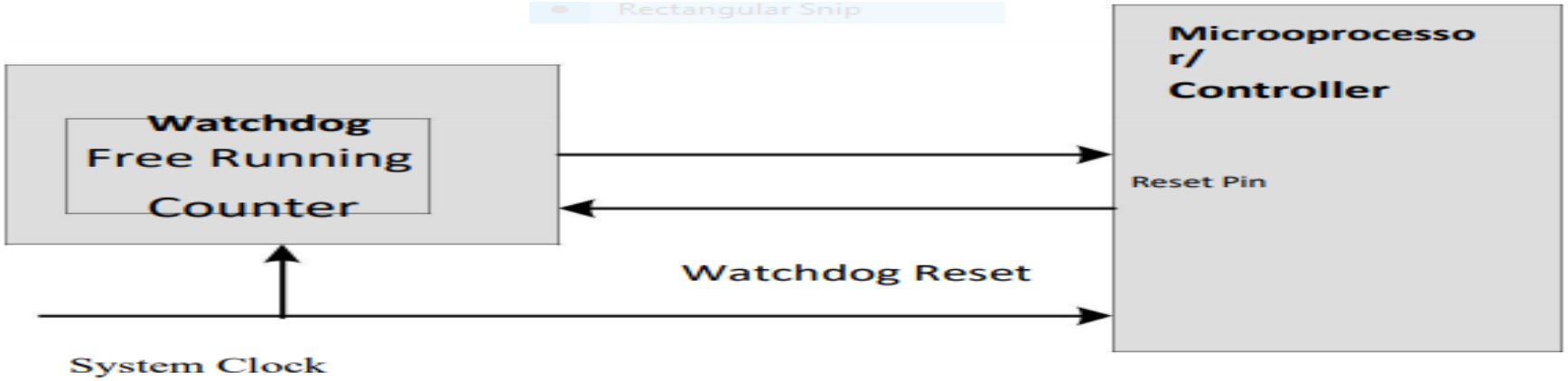
- A timer unit for monitoring the firmware execution.

- Depending on the internal implementation, the watchdog timer increments or decrements a free running counter with each clock pulse and generates a reset signal to reset the processor if the count reaches zero for a down counting watchdog, or the highest count value for an up counting watchdog.

- If the watchdog counter is in the enabled state, the firmware can write a zero (for up counting watchdog implementation) to it before starting the execution of a piece of code (subroutine or portion of code which is susceptible to execution hang up) and the watchdog will start counting. If the firmware execution doesn't complete due to malfunctioning, within the time required by the watchdog to reach the maximum count, the counter will generate a reset pulse and this will reset the processor.

- If the firmware execution completes before the expiration of the watchdog

# WATCH DOG TIMER




- Most of the processors implement watchdog as a built-in component and provides status register to control the watchdog timer (like enabling and disabling watchdog functioning) and watchdog timer register for writing the count value. If the processor/controller doesn't contain a built in watchdog timer, the same can be implemented using an external watchdog timer IC circuit.

# Embedded Firmware Design & Development language

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

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

➢The embedded firmware imparts intelligence to an Embedded system.

➢It is a one time process and it can happen at any stage.

➢There exist two basic approaches for the design and implementation of embedded firmware, namely.

- **The Super loop based approach**
- **The Embedded Operating System based approach**

# Embedded firmware Design Approaches

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

**1.Embedded firmware Design Approaches – The Super loop:**

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

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

➢The tasks are executed in a never ending loop.

# Pros and Cons:

**Pros:**

- Doesn't require an Operating System for task scheduling and monitoring and free from OS related overheads
- Simple and straight forward design
- Reduced memory footprint

**Cons:**

- Non Real time in execution behavior (As the number of tasks increases the frequency at which a task gets CPU time for execution also increases)
- Any issues in any task execution may affect the functioning of the product (This can be effectively tackled by using Watch Dog Timers for task execution monitoring)

**Enhancements:**

- Combine Super loop based technique with interrupts

# Embedded firmware Design Approaches – Embedded OS based Approach:

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

- The embedded device contains an Embedded Operating System which can be one of:
- A Real Time Operating System (RTOS)
- A Customized General Purpose Operating System (GPOS)
- The Embedded OS is responsible for scheduling the execution of user tasks and the allocation of system resources among multiple tasks
- It Involves lot of OS related overheads apart from managing and executing user defined tasks
- Microsoft® Windows XP Embedded is an example of GPOS for embedded

# Embedded firmware Development Languages/Options

**Embedded firmware Development Languages/Options**

**1.Assembly Language**

**2.High Level Language**

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

**3.Mix of Assembly & High level Language**

- Mixing High Level Language (Like C) with Assembly Code
- Mixing Assembly code with High Level Language (Like C)
- Inline Assembly

# Embedded firmware Development Languages/Options

**Embedded firmware Development Languages/Options – Assembly Language**

- 'Assembly Language' is the human readable notation of 'machine language'
- 'Machine language' is a processor understandable language
- Machine language is a binary representation and it consists of 1s and 0s
- Assembly language and machine languages are processor/controller dependent
- An Assembly language program written for one processor/controller family will not work with others
- Assembly language programming is the process of writing processor specific machine code in mnemonic form, converting the mnemonics into actual processor instructions (machine language) and associated data using an assembler
- The general format of an assembly language instruction is an Op code followed by Operands
- The Op code tells the processor/controller what to do and the Operands provide the data and information required to perform the action specified by the op code
- It is not necessary that all op code should have Operands following them. Some of the Op code implicitly contains the operand and in such situation no operand is required. The operand may be a single operand, dual operand or more

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

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

- The Assembly language program written in assembly code is saved as .asm (Assembly file) file or a .src (source) file or a format supported by the assembler
- The software utility called 'Assembler' performs the translation of assembly code to machine code
- The assemblers for different family of target machines are different. A51 Macro Assembler from Keil software is a popular assembler for the 8051 family micro

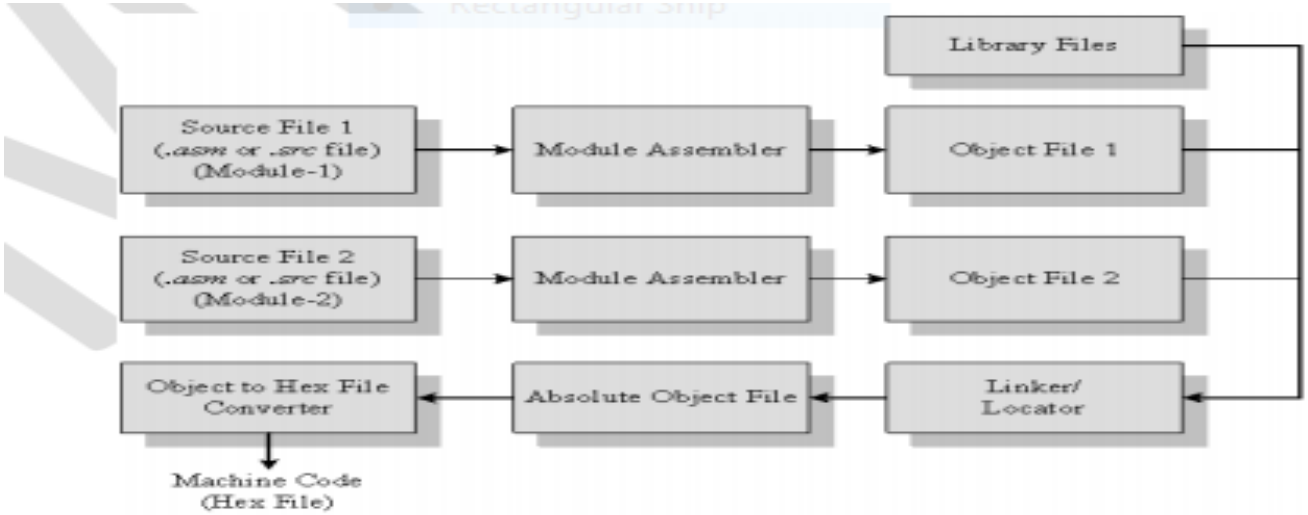



Figure : Assembly Language to machine language conversion

# Advantages:

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

➢The developer is well aware of the target processor architecture and memory organization, so optimized code can be written for performing operations.

➢This leads to less utilization of code memory and efficient utilization of data memory.

**2. High Performance:**

➢Optimized code not only improves the code memory usage but also improves the total system performance.

➢Through effective assembly coding, optimum performance can be achieved for target processor.

**3.Low level Hardware Access:**

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

**4.Code Reverse Engineering**:

➢It is the process of understanding the technology behind a product by extracting the information from the finished product.

➢It can easily be converted into assembly code using a dis-assembler program for

# Drawbacks:

**1. High Development time:**

➢The developer takes lot of time to study about architecture , memory organization, addressing modes and instruction set of target processor/controller.

➢More lines of assembly code is required for performing a simple action.

**2. Developer dependency:**

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

**3. Non portable:**

➢Target applications written in assembly instructions are valid only for that particular family of processors and cannot be re-used for another target processors/controllers.

➢If the target processor/controller changes, a complete re-writing of the application using assembly language for new target processor/controller is required.

# Embedded firmware Development Languages/Options

## Embedded firmware Development Languages/Options – High Level Language

- The embedded firmware is written in any high level language like C, C++
- A software utility called 'cross-compiler' converts the high level language to target processor specific machine code
- The cross-compilation of each module generates a corresponding object file. The object file does not contain the absolute address of where the generated code needs to be placed (a re-locatable code) on the program memory
- The software program called linker/locater is responsible for assigning absolute address to object files during the linking process
- The Absolute object file created from the object files corresponding to different source code modules contain information about the address where each instruction needs to be placed in code memory
- A software utility called 'Object to Hex file converter' translates the absolute object file to corresponding hex file (binary file)

**Advantages:**

**Reduced Development time**:

- Developer requires less or little knowledge on internal hardware details and architecture of the target processor/Controller.

**Developer independency**:

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

**Portability:**

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

**Drawbacks:**

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

# Mixing High level language like ‘C’ with Assembly Langu

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

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

**3. In line Assembly:**

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

# UNIT-IV

## RTOS BASED EMBEDDED SYSTEM DESIGN

# Operating System Basics

An Operating system (OS) is a piece of software that controls the overall operation of the Computer. It acts as an interface between hardware and application programs .It facilitates the user to format disks, create, print, copy, delete and display files, read data from files ,write data to files , control the I/O operations, allocate memory locations and process the interrupts etc.

# Operating System Basics..

It provides the users an interface to the hardware resources. In a multiuser system it allows several users to share the CPU time, share the other system resources and provide inter task communication, Timers, clocks, memory management and also avoids the interference of different users in sharing the resources etc. Hence the OS is also known as a resource manager.

# Types of operating systems

An Operating system (OS) is nothing but a piece of software that controls the overall operation of the Computer. It acts as an interface between hardware and application programs .It facilitates the user to format disks, create ,print ,copy , delete and display files , read data from files ,write data to files ,control the I/O operations , allocate memory locations and process the interrupts etc. It provides the users an interface to the hardware resources.

# Types of operating systems..

In a multiuser system it allows several users to share the CPU time, share the other system resources and provide inter task communication, Timers, clocks, memory management and also avoids the interference of different users in sharing the resources etc. Hence the OS is also known as a resource manager.

There are three important types of operating systems .They are (i).Embedded Operating System (ii). Real time operating system and (iii).Handheld operating system.

# REAL TIME SYSTEMS:

Real-time systems are those systems in which the correctness of the system depends not only on the Output, but also on the time at which the results are produced (Time constraints must be strictly followed).

Real time systems are two types. (i) Soft real time systems and (ii) Hard real time systems. A Soft real time system is one in which the performance of the system is only degraded but, not destroyed if the timing deadlines are not met.

# REAL TIME OPERATING SYSTEM (RTOS)

It is an operating system that supports real-time applications by providing logically correct result within the deadline set by the user. A real time operating system makes the embedded system into a real time embedded system. The basic structure of RTOS is similar to regular OS but, in addition, it provides mechanisms to allow real time scheduling of tasks.

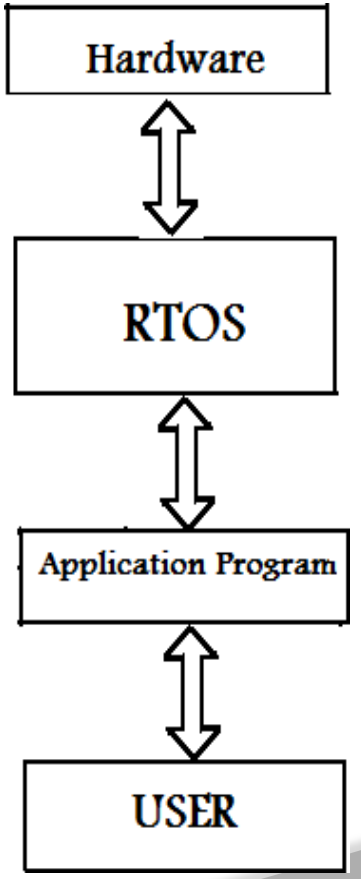
Though the real-time operating systems may or may not increase the speed of execution, but they provide more precise and predictable timing characteristics than general-purpose OS.

The figure below shows the embedded system with RTOS.

# REAL TIME OPERATING SYSTEM (RTOS)

All the embedded systems are not designed with RTOS. Low end application systems do not require the RTOS but only High end application oriented embedded systems which require scheduling alone need the RTOS.

For example an embedded system which measures Temperature or Humidity etc. do not require any operating system. Whereas a Mobile phone , RADAR or Satellite system used for high end applications require an operating system.



# Task

A task is a basic unit or atomic unit of execution that can be scheduled by an RTOS to use the system resources like CPU, Memory, I/O devices etc. It starts with reading of the input data and of the internal state of the task, and terminates with the production of the results and updating the internal state. The control signal that initiates the execution of a task is provided by the operating system.

There are two types of tasks.

(i)Simple Task(S-Task) and

(ii) Complex Task(C-Task).

# Task States


At any instant of time a task can be in one of the following states:

**(i)Dormant (ii). Ready (iii). Running and (iv).Blocked.**

When a task is first created, it is in the dormant task. When it is added to RTOS for scheduling, it is a ready task. If the input or a resource is not available, the task gets blocked.

# Task States

An Idle Task does nothing .The idle task has the lowest priority.

```
void Idle task(void)
{
While(1);
}
```

# Creation of a Task

A task is characterized by the parameters like task name , its priority , stack size and operating system options .To create a task these parameters must be specified .A simple program to create a task is given below.

```
result = task-create(“Tx Task”, 100,0x4000,OS_Pre-emptiable);     /*task create*/ if (result = = os_success)
{           /*task successfully created*/
}
```

# Process or Task:

Embedded program (a static entity) = a collection of firmware modules. When a firmware module is executing, it is called a process or task . A task is usually implemented in C by writing a function. A task or process simply identifies a job that is to be done within an embedded application.

When a process is created, it is allocated a number of resources by the OS, which may include: – Process stack – Memory address space – Registers (through the CPU) – A program counter (PC) – I/O ports, network connections, file descriptors, etc.

# Threads

A process or task is characterized by a collection of resources that are utilized to execute a program. The smallest subset of these resources (a copy of the CPU registers including the PC and a stack) that is necessary for the execution of the program is called a thread. A thread is a unit of computation with code and context, but no private data.

# Process and Threads

**PROCESS AND THREADS:**

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

# PROCESS vs THREADS

| Process | Thread |
|---|---|
| 1) System calls involved in process. | 1) No system calls involved. |
| 2) Context switching required. | 2) No context switching required. |
| 3) Different process have different copies of code and data. | 3) Sharing same copy of code and data can be possible among different threads.. |
| 4) Operating system treats different process differently. | 4) All user level threads treated as single task for operating system. |
| 5) If a process got blocked, remaining process continue their work. | 5) If a user level thread got blocked, all other threads get blocked since they are treated as single task to OS. (Noted: This is can be avoided in kernel level threads). |
| 6) Processes are independent. | 6) Threads exist as subsets of a process. They are dependent. |
| 7) Process run in separate memory space. | 7) Threads run in shared memory space. And use memory of process which it belong to. |
| 8) Processes have their own program counter (PC), register set, and stack space. | 8) Threads share Code section, data section, Address space with other threads. |
| 9) Communication between processes requires some time. | 9) Communication between processes requires less time than processes. |
| 10) Processes don't share the memory with any other process. | 10) Threads share the memory with other threads of the same process |
| 11) Process have overhead. | 11) Threads have no overhead. |

## MULTI PROCESSING AND MULTI TASKING:

**Multiprogramming** – A computer running more than one program at a time (like running Excel and Firefox simultaneously).

**Multiprocessing** – A computer using more than one CPU at a time.

**Multitasking** – Tasks sharing a common resource (like 1 CPU). Multithreading is an extension of **multitasking**.

## Multiprocessing –

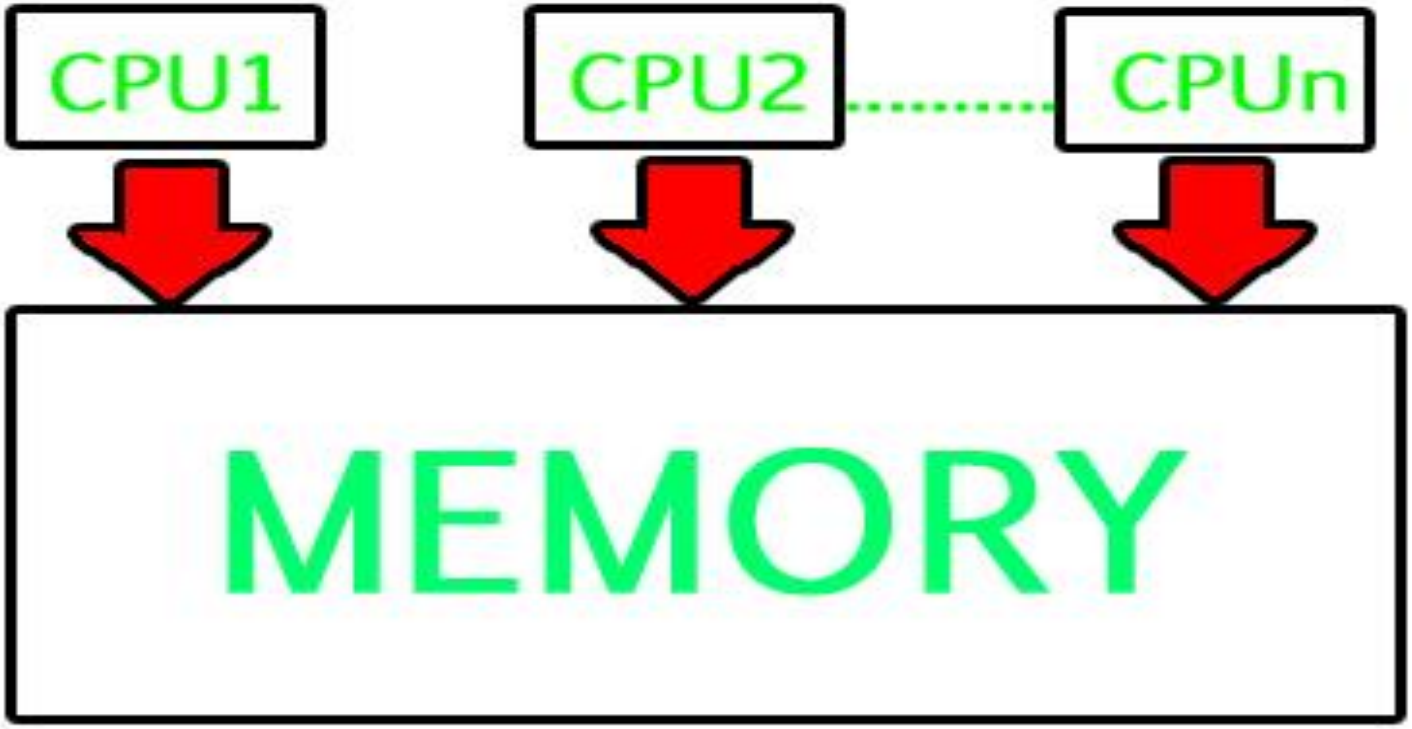
In a uni-processor system, only one process executes at a time. Multiprocessing is the use of two or more CPUs (processors) within a single Computer system. The term also refers to the ability of a system to support more than one processor within a single computer system. Now since there are multiple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

# MULTI PROCESSIN WORKING

**Multi processing system's working –**

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

# MULTI PROCESSING



# Multitasking

A multitasking environment allows applications to be constructed as a set of independent tasks, each with a separate thread of execution and its own set of system resources. The inter-task communication facilities allow these tasks to synchronize and coordinate their activity. Multitasking provides the fundamental mechanism for an application to control and react to multiple, discrete real-world events and is therefore essential for many real-time applications.

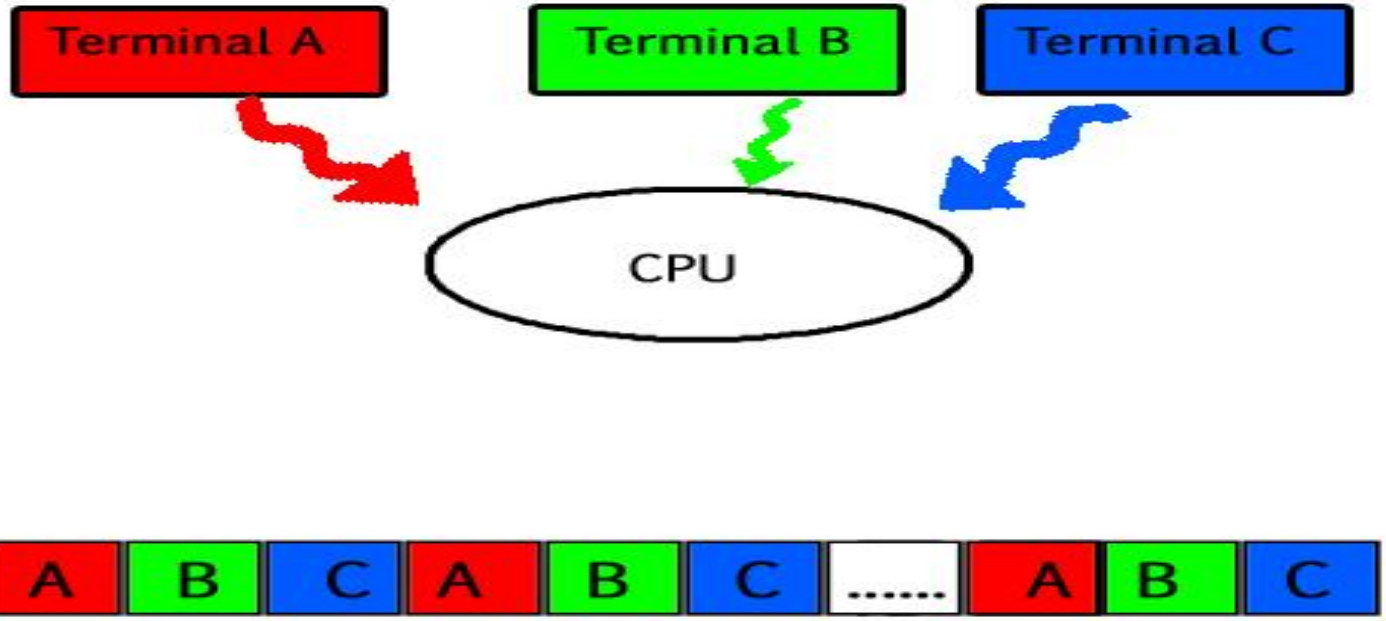
# Multitasking….

Multitasking creates the appearance of many threads of execution running concurrently when, in fact, the kernel interleaves their execution on the basis of a scheduling algorithm. This also leads to efficient utilization of the CPU time and is essential for many embedded applications where processors are limited in computing speed due to cost, power, silicon area and other constraints. In a multi-tasking operating system it is assumed that the various tasks are to cooperate to serve the requirements of the overall system.

# Multitasking….

Co-operation will require that the tasks communicate with each other and share common data in an orderly an disciplined manner, without creating undue contention and deadlocks. The way in which tasks communicate and share data is to be regulated such that communication or shared data access error is prevented and data, which is private to a task, is protected. Further, tasks may be dynamically created and terminated by other tasks, as and when needed.

# Multitasking….



# Task Scheduler

Task scheduler is one of the important component of the Kernel .Basically it is a set of algorithms that manage the multiple tasks in an embedded system. The various tasks are handled by the scheduler in an orderly manner. This produces the effect of simple multitasking with a single processor. The advantage of using a scheduler is the ease of implementing the sleep mode in microcontrollers which will reduce the power consumption considerably (from mA to μA). This is important in battery operated embedded systems.

# Task Scheduler….

The task scheduler establishes task time slots. Time slot width and activation depends on the available resources and priorities.

A scheduler decides which task will run next in a multitasking system. Every RTOS provides three specific functions.

(i).Scheduling (ii) Dispatching and (iii). Inter-process communication and synchronization.

The scheduling determines ,which task ,will run next in a multitasking system and the dispatches perform the necessary book keeping to start the task and Inter-process communication and synchronization assumes that each task cooperate with others.

# Process or Task:

Embedded program (a static entity) = a collection of firmware modules. When a firmware module is executing, it is called a process or task . A task is usually implemented in C by writing a function. A task or process simply identifies a job that is to be done within an embedded application.

When a process is created, it is allocated a number of resources by the OS, which may include: – Process stack – Memory address space – Registers (through the CPU) – A program counter (PC) – I/O ports, network connections, file descriptors, etc.

# Threads

A process or task is characterized by a collection of resources that are utilized to execute a program. The smallest subset of these resources (a copy of the CPU registers including the PC and a stack) that is necessary for the execution of the program is called a thread. A thread is a unit of computation with code and context, but no private data.

# Multitasking

A multitasking environment allows applications to be constructed as a set of independent tasks, each with a separate thread of execution and its own set of system resources. The inter-task communication facilities allow these tasks to synchronize and coordinate their activity. Multitasking provides the fundamental mechanism for an application to control and react to multiple, discrete real-world events and is therefore essential for many real-time applications.

# Multitasking….

Multitasking creates the appearance of many threads of execution running concurrently when, in fact, the kernel interleaves their execution on the basis of a scheduling algorithm. This also leads to efficient utilization of the CPU time and is essential for many embedded applications where processors are limited in computing speed due to cost, power, silicon area and other constraints. In a multi-tasking operating system it is assumed that the various tasks are to cooperate to serve the requirements of the overall system.

# Multitasking….

Co-operation will require that the tasks communicate with each other and share common data in an orderly an disciplined manner, without creating undue contention and deadlocks. The way in which tasks communicate and share data is to be regulated such that communication or shared data access error is prevented and data, which is private to a task, is protected. Further, tasks may be dynamically created and terminated by other tasks, as and when needed.

# UNIT-V

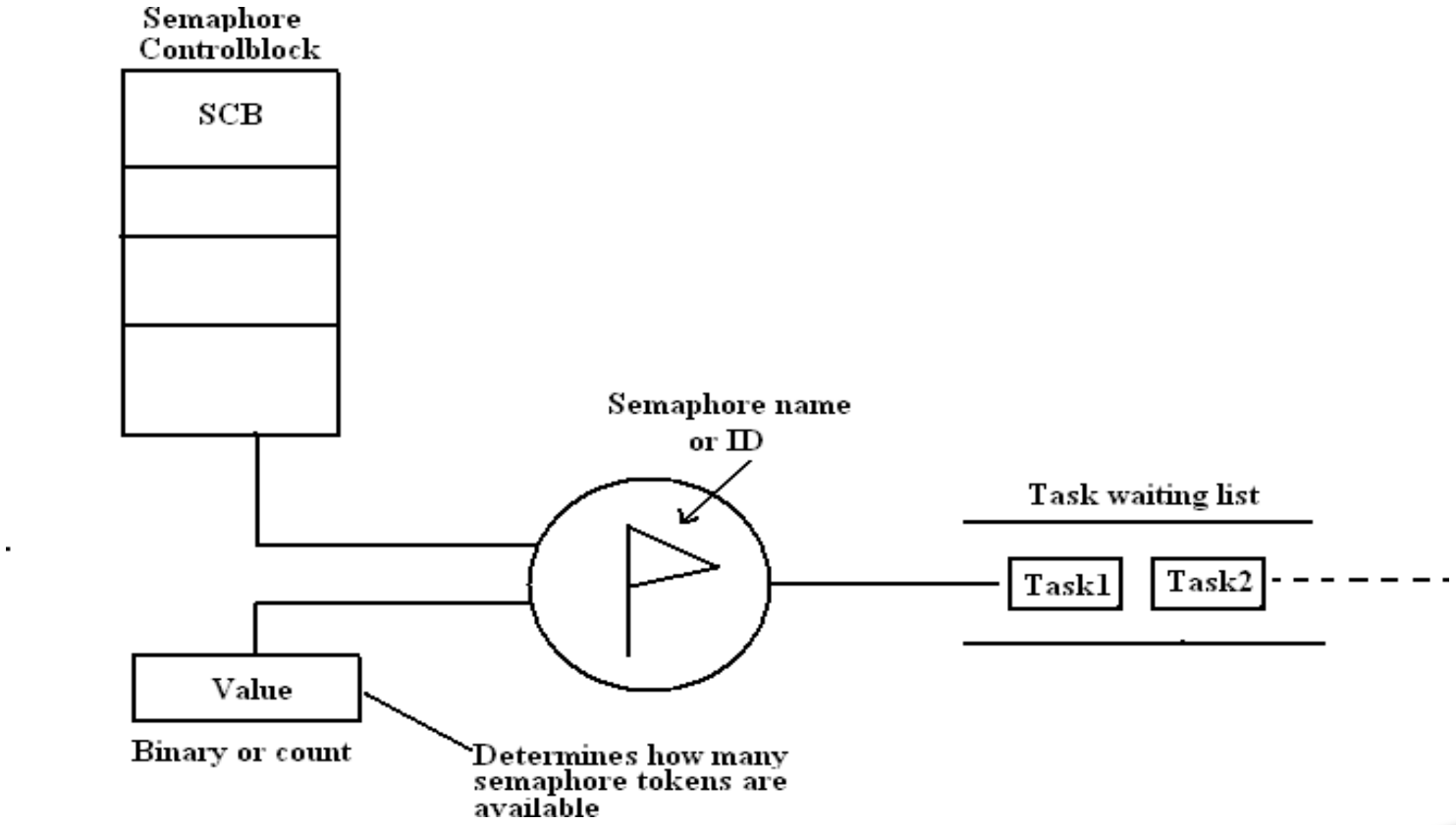
## TASK COMMUNICATION

# Semaphores

A semaphore is nothing but a value or variable or data which can control the allocation of a resource among different tasks in a parallel programming environment. So, Semaphores are a useful tool in the prevention of race conditions and deadlocks; however, their use is by no means a guarantee that a program is free from these problems.

Semaphores which allow an arbitrary resource count are called counting semaphores, whilst semaphores which are restricted to the values 0 and 1 (or locked/unlocked, unavailable/available) are called binary semaphores.

# Semaphores…


Semaphore
Controlblock
SCB
Semaphore name
or ID
Task waiting list
Task1
Task2
Value
Binary or count
Determines how many
semaphore tokens are
available

# Semaphores…

**Types of Semaphores:** There are three types of semaphores

- Binary Semaphores,
- Counting Semaphores and
- Mutexes.

# Message Queues

The Message Queues, are used to send one or more messages to a task i.e. the message queues are used to establish the Inter task communication. Basically Queue is an array of mailboxes. Tasks and ISRs can send and receive messages to the Queue through services provided by the kernel. Extraction of messages from a queue follow FIFO or LIFO structure.

# Message Queues…

Applications of message queue are

- Taking the input from a keyboard
- To display output
- Reading voltages from sensors or transducers
- Data packet transmission in a network

In each of these applications, a task or an ISR deposits the message in the message queue. Other tasks can take the messages. Based on our application, the highest priority task or the first task waiting in the queue can take the message. At the time of creating a queue, the queue is given a name or ID, queue length, sending task waiting list and receiving task waiting list.

# Saving Memory and Power

**Saving memory:**

Embedded systems often have limited memory.

RTOS: each task needs memory space for its stack.

The first method for determining how much stack space a task needs is to examine your code

The second method is experimental. Fill each stack with some recognizable data pattern at startup, run the system for a period of time

# Saving Memory and Power…

**Program Memory:**

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

# Saving Memory and Power…

**Data Memory**

Consider using more static variables instead of stack variables

On 8-bit processors, use char instead of int when possible.

# Saving Memory and Power…

**Saving power:**

The primary method for preserving battery power is to turn off parts or all of the system whenever possible.

Most embedded-system microprocessors have at least one power-saving mode.

The modes have names such as sleep mode, low-power mode, idle mode, standby mode, and so on.

A very common power-saving mode is one in which the microprocessor stops executing instructions, stops any built-in peripherals, and stops its clock circuit.

# Saving Memory and Power…

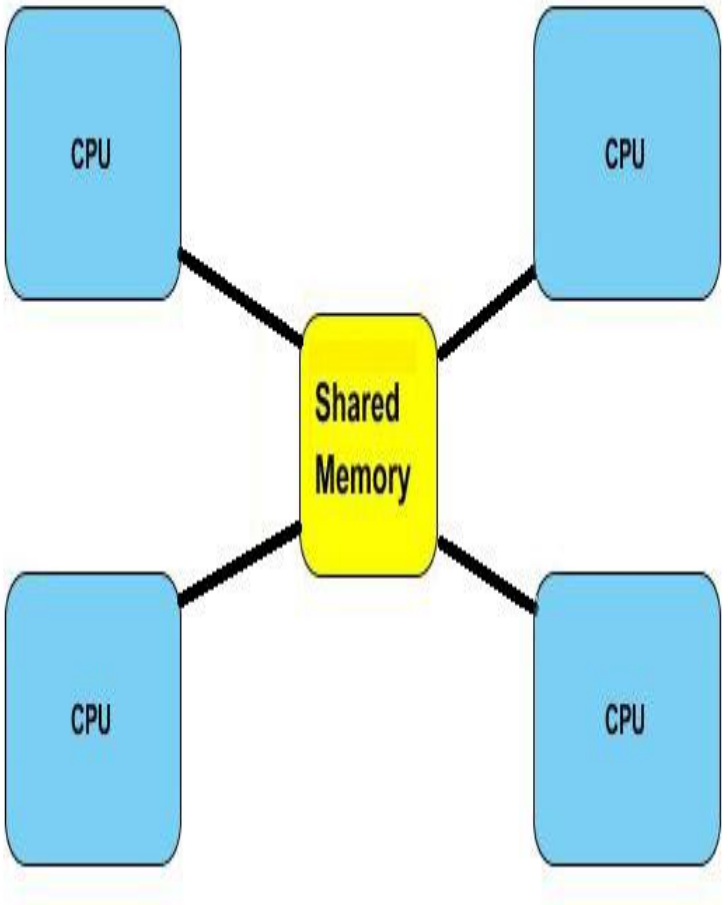
**Shared memory:**

In this model stored information in a shared region of memory is processed, possibly under the control of a supervisor process.

An example might be a single node with multiple cores.

share a global memory space

cores can efficiently exchange/share data.



# Message Passing

In this model, data is shared by sending and receiving messages between co-operating processes, using system calls. Message Passing is particularly useful in a distributed environment where the communicating processes may reside on different, network connected, systems. Message passing architectures are usually easier to implement but are also usually slower than shared memory architectures.

# Remote Procedure Call (RPC)

RPC allows programs to call procedures located on other machines. When a process on machine A calls' a procedure on machine B, the calling process on A is suspended, and execution of the called procedure takes place on B. Information can be transported from the caller to the callee in the parameters and can come back in the procedure result. No message passing at all is visible to the programmer. This method is known as Remote Procedure Call, or often just RPC.

# Remote Procedure Call (RPC)…

It can be said as the special case of message-passing model. It has become widely accepted because of the following features: Simple call syntax and similarity to local procedure calls. Its ease of use, efficiency and generality. It can be used as an IPC mechanism between processes on different machines and also between different processes on the same machine.

# Sockets

Sockets (Berkley sockets) are one of the most widely used communication APIs. A socket is an object from which messages and are sent and received. A socket is a network communication endpoint.

In connection-based communication such as TCP, a server application binds a socket to a specific port number. This has the effect of registering the server with the system to receive all data destined for that port. A client can then rendezvous with the server at the server's port, as illustrated here: Data transfer operations on sockets work just like read and write operations on files. A socket is closed, just like a file, when communications is finished.

# Sockets…

Network communications are conducted through a pair of cooperating sockets, each known as the peer of the other.

Processes connected by sockets can be on different computers (known as a heterogeneous environment) that may use different data representations.

Data is serialized into a sequence of bytes by the local sender and deserialized into a local data format at the receiving end.

# TASK SYNCHRONISATON

- In general ,a task must synchronize its activity with other task to execute a multithreaded program properly
- Consider a situation where two processor try to access a shared memory area where one process tries to write to memory allocation when other process try to read
- This scenario leads to conflicts  i.e, task synchronization issue

# TASK SYNCHRONIZATION ISSUES

- **Deadlock** : It creates a situation where none of the processes are able to make any progress in their execution
- The different condition favouring a deadlock situation are:-

i) **Mutual exclusion** : The condition in which a process can hold one resource at a time

eg: Hardware in a embedded device

# TASK SYNCHRONIZATION ISSUES

ii) **Hold and wait** : The condition in which a process hold a shared resource by acquiring the lock controlling the shared access and waiting for additional resource held by other processor

iii**) No resource preemption** : The criteria that operating system cannot take back a resource from a process which is currently holding it and resource can only be released voluntarily by the processor holding it

# TASK SYNCHRONIZATION ISSUES

iv) **Circular wait** : A process is waiting for a resource which is currently held by another process which in turn waiting for a resource held by the first resource .In general ,there exists a set of waiting process p0,p1,….pn with p0 is waiting for a resource held by p1 and p1 is waiting for a resource held by p0,..pn is waiting for a resource held by p0 and p0 is waiting for resource held by pn and so on …This forms a circular wait queue

# TASK SYNCHRONIZATION ISSUES

- **Deadlock Handling**:

The OS may adopt any of the following techniques to detect and prevent deadlock condition

i)**Ignore Deadlocks**: Always assume that the system design is deadlock free .This is acceptable for the reason the cost of removing a deadlock is large compared to the chance of happening a deadlock

# TASK SYNCHRONIZATION ISSUES

- **Detect and Recover:** This approach suggests the detection of a deadlock situation and recovery from it. This is similar to the deadlock condition that may arises at a traffic condition .When the vehicles from different direction compete to cross the junction , deadlock condition is resulted
- **Avoid Deadlocks:** Deadlock is avoided by the careful resource allocation techniques by operating system .It is similar to the traffic light mechanism at junctions at avoid the traffic jam

# TASK SYNCHRONIZATION ISSUES

- **Live lock :** It is similar to a deadlock ,except that the status of the process involved in the live lock constanly change with regard to one another. In this process are not in the waiting state and they are running concurrently
- **Starvation :**It occurs when a low priority program is requesting for a system resource, but are not able to execute because a higher priority program utilizing that resource for an extended period.

# TASK SYNCHRONIZATION ISSUES

- **Prevent Deadlocks** : Prevent the deadlock condition by negating the below condition of the deadlock situation

i) Ensure that a process does not hold any other resources, when it request a resource.

ii) A process must request all its required resource and the resources should be allocated before the process begins its execution

# TASK SYNCHRONIZATION ISSUES

Producer–consumer/Bounded Buffer problem

- A thread/process which produces a data is called ‘Producer thread/process’ and a thread /process which consumes the data produced by a producer thread/process is known as ‘consumer thread/process’.
- The producer thread keeps on producing data and puts it in the buffer and the consumer thread keeps on consuming the data from the buffer . which may leads to cause two cases

# TASK SYNCHRONIZATION ISSUES

i) If the producer produces the data at a faster rate than the rate at which it is consumed by consumer this lead to ‘buffer overrun’.

ii) If the consumer consumes data at a faster rate than the rate at which it is produced by the producer, this leads to ‘buffer under run’.

- The producer-consumer problem can be rectified in various method.one simple solution is ‘sleep and wake-up’.

# TASK SYNCHRONIZATION ISSUES

**Readers and Writers Problem:**

The Readers-writers problem is common issue observed in processes competing for limited shared resources . The Reader-writers problem is characterized by multi processes trying to read and write the shared data concurrently

# TASK SYNCHRONIZATION ISSUES

**Priority Inversion** : It is the combination of blocking based process synchronisation and pre-emptive priority scheduling.

- Priority Inversion is a condition in which a high priority task needs to wait for a low priority task to release a resource which is shared between the high priority task and low priority task, and a medium priority task which doesn't require the shared resource continues its execution by preempting the low priority task.

# HOW TO CHOOSE RTOS

Selecting the right RTOS is a critical step in any embedded software development project. Selecting the wrong RTOS could affect project costs, time to market and have real-time implications on the behavior of the system. When selecting an RTOS, teams usually focus just on cost but there are seven characteristics that should be considered. Let’s examine each on

# HOW TO CHOOSE RTOS

- **Characteristic #1 – Performance**
- RTOS performance is a critical factor to consider when selecting an RTOS. All RTOSes are NOT created equal and an attempt to save a few dollars can cost orders of magnitude more. When it comes to performance, developers have a variety of factors that need to be considered. First, memory requirements such as ROM, flash and RAM footprints need to be considered. RTOSes are powerful and with that power comes additional code and data needs. Second, processing speed such as interrupt latency and context switch times should be reviewed. A high quality RTOS will document these parameters for a variety of architectures and clock speeds.

# HOW TO CHOOSE RTOS

- **Characteristic #3 – Cost**
- Undoubtedly one of the largest, if not only thought about RTOS characteristic is cost. Despite the huge efforts required in labor to develop robust software, no one wants to pay for it! Developers need to get over it and probably evaluate what an RTOS may really cost. A few considerations for commercial RTOSes are the upfront costs for licensing and any recurring licenses such as royalties. In additional to these obvious costs, developers need to also consider the total cost of ownership for the RTOS. That is, the cost to learn, setup, integrate and debug the selected operating system. Total cost of ownership for an open source RTOS can potentially exceed that of a commercially purchased RTOS due to lack of support, poor code quality and so forth.

# HOW TO CHOOSE RTOS

**Characteristic #4 – Ecosystem**

Having the best performance, features and cost doesn't mean a thing if there isn't a large and vibrant community to support the RTOS. A software products ecosystem is a critical piece of the selection process in order to ensure ease of integration, support and product lifetime. When developers investigate an RTOS ecosystem, they should determine whether the RTOS is supported and adopted by their industry and the embedded software industry as a whole. They should determine whether there is support for a variety of architectures and processors or whether the RTOS is just a one trick pony. The availability of numerous examples and ports is also an important indicator that the RTOS is supported and has a strong community of users around it.

# HOW TO CHOOSE RTOS

**Characteristic #5 – Middleware**

Many RTOSes come with middleware components or have third parties who have developed components that integrate into the RTOS. Developers should evaluate their RTOSes middleware and determine what the integration effort might be. Sometimes the integration is seamless while other times it is an obvious nightmare. Some RTOSes lack support for middleware and open source components need to be integrated which can lead to a variety of time consuming integration issues. Verify that the middleware has common components such as USB, TCP/IP, file systems and graphics generators before jumping in with both feet.

# HOW TO CHOOSE RTOS

- **Characteristic #6 – Vendor**
- Take a good hard look at the vendor who developed, maintains and distributes the RTOS. Do they have a good track record dating back at least five, ten or fifteen years? Examine their source code and documentation. A good supplier with have meticulous documentation that answers many of the questions that would arise while integrating the RTOS into the system. No matter how good documentation gets, it will never be perfect. Testing how fast the vendor is to respond to question and support issues could be critical and save precious time and money getting the product out the door. Don't just blindly trust. Be an engineer and put the vendor to the test and see if they squirm or roll up their sleeves

# HOW TO CHOOSE RTOS

- **Characteristic #7 – Engineering Team**
- The characteristic of RTOS selection that is probably the most common to overlook is the engineering team. The RTOS that is selected should minimize the labor intensity for the team and allow them to focus on product differentiators rather than increase it as they learn how to integrate and setup an RTOS. As much as we like to grow professionally as engineers, we should attempt to select an RTOS we are familiar with and can work most efficiently with. Sometimes development doesn't work out that way but we should at least try