MODULE – 5

RTOS AND IDE FOR EMBEDDED SYSTEM DESIGN

IRTOS-BASED EMBEDDED SYSTEM DESIGN

The *super loop* based task execution model for firmware executes the tasks sequentially in order in which the tasks are listed within the loop. Here every task is repeated at regular intervals and the task execution is non-real time. Also, any response delay is acceptable and it will not create any operational issues or potential hazards.

But, certain applications demand time critical response to tasks/ events and delay in the response may be catastrophic. Examples: Flight control systems, Air bag control, Anti-lock Brake Systems (ABS) for vehicles, Nuclear monitoring devices, etc.

In embedded systems, the time critical response for tasks/ events may be addressed by –

- Assigning priority to tasks and execute the high priority task.
- Dynamically change the priorities of tasks, if required on a need basis.

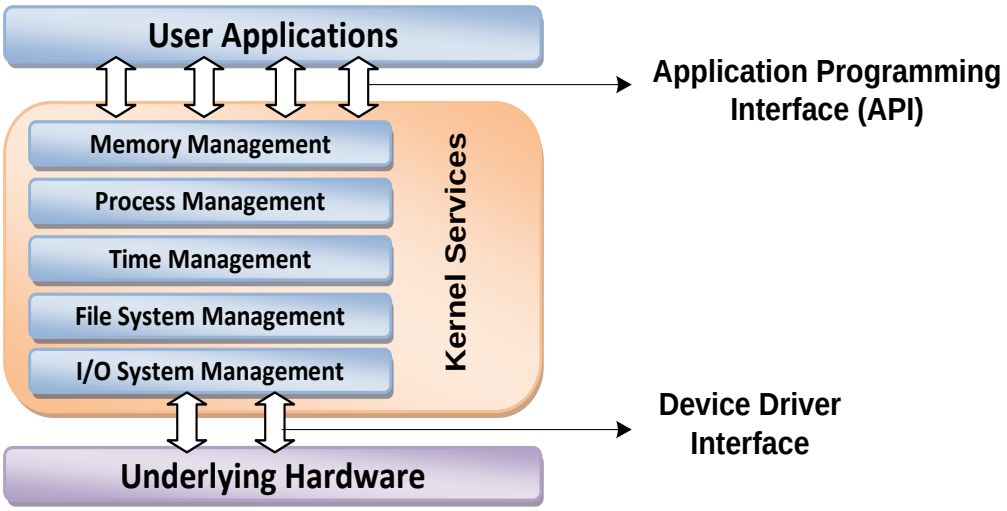
The introduction of operating system based firmware execution in embedded devices can address these needs to a greater extent.

OPERATING SYSTEM (OS) BASICS:

The *Operating System (OS)* acts as a bridge between the user applications/ tasks and the underlying system resources through a set of system functionalities and services. The primary functions of operating systems are

- Make the system convenient to use
- Organize and manage the system resources efficiently and correctly.

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



The Kernel:

The *kernel* is the core of the operating system. It is responsible for managing the system resources and the communication among the hardware and other system services. Kernel acts as the abstraction layer between system resources and user applications.

- Kernel contains a set of system libraries and services. For a general purpose OS, the kernel contains different services like memory management, process management, time management, file system management, I/O system management.

Process Management: The process management deals with managing the process/ tasks. Process management includes –

- setting up a memory for the process
- loading process code into memory
- allocating system resources
- scheduling and managing the execution of the process
- setting up and managing Process Control Block (PCB)
- inter process communication and synchronization
- process termination/ deletion, etc.

Primary Memory Management: Primary memory refers to a volatile memory (RAM), where processes are loaded and variables and shared data are stored.

The Memory Management Unit (MMU) of the kernel is responsible for –

- Keeping a track of which part of the memory area is currently used by which process
- Allocating and De-allocating memory space on a need basis.

File System Management: File is a collection of related information. A file could be a program (source code or executable), text files, image files, word documents, audio/ video files, etc. A file system management service of kernel is responsible for –

- The creation, deletion and alteration of files
- Creation, deletion, and alteration of directories
- Saving of files in the secondary storage memory
- Providing automatic allocation of file space based on the amount of free running space available
- Providing flexible naming conversion for the files.

I/O System (Device) Management: Kernel is responsible for routing the I/O requests coming from different user applications to the appropriate I/O devices of the system. In a well structured OS, direct

access to I/O devices is not allowed; access to them is establish through Application Programming Interface (API). The kernel maintains list of all the I/O devices of the system. The service '*Device Manager*' of the kernel is responsible for handling all I/O related operations. The Device Manager is responsible for –

- Loading and unloading of device drivers
- Exchanging information and the system specific control signals to and from the device.

Secondary Storage Management: The secondary storage management deals with managing the secondary storage memory devices (if any) connected to the system. Secondary memory is used as backup medium for programs and data, as main memory is volatile. In most of the systems secondary storage is kept in disks (hard disks). The secondary storage management service of kernel deals with –

- Disk storage allocation
- Disk scheduling
- Free disk space management

Protection Systems: Modern operating systems are designed in such way to support multiple users with different levels of access permissions. The *protection* deals with implementing the security policies to restrict the access of system resources and particular user by different application or processes and different user.

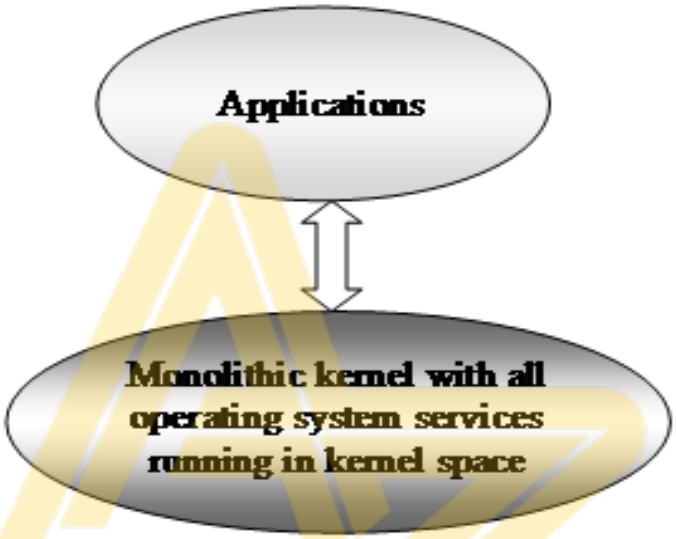
Interrupt Handler: Kernel provides interrupt handler mechanism for all external/ internal interrupt generated by the system.

The important services offered by the kernel of an OS:

- ***Kernel Space and User Space:*** The program code corresponding to the kernel applications/ services are kept in a contiguous area of primary (working) memory and is protected from the un-authorized access by user programs/ applications.
    - The memory space at which the kernel code is located is known as '*Kernel Space*'. All user applications are loaded to a specific area of primary memory and this memory area is referred as '*User Space*'.
    - The partitioning of memory into kernel and user space is purely Operating System dependent.
    - Most of the operating systems keep the kernel application code in main memory and it is not swapped out into the secondary memory.

Monolithic Kernel and Microkernel: Kernel forms the heart of OS. Different approaches are adopted for building an operating system kernel. Based on the kernel design, kernels can be classified into '*Monolithic*' and '*Micro*'.

- ***Monolithic Kernel:*** In monolithic kernel architecture, all kernel services run in the kernel space. All kernel modules run within the same memory space under a single kernel thread.
- The major drawback of monolithic kernel is that any error or failure in any one of the kernel modules leads to the crashing of the entire kernel application.
    - LINUX, SOLARIS, MS-DOS kernels are examples of monolithic kernel.



- ***Microkernel:*** The microkernel design incorporates only essential set of operating system services into the kernel. The rest of the operating systems services are implemented in program known as '*Servers*' which runs in user space. The memory management, timer systems and interrupt handlers are the essential services, which forms the part of the microkernel. The benefits of micro kernel based designs are –
    - *Robustness:* If a problem is encountered in any of the services, which runs as a server can be reconfigured and restarted without the restarting the entire OS. Here chances of corruption of kernel services are ideally zero.
    - *Configurability:* Any services, which runs as a server application can be changed without the need to restart the whole system. This makes the system dynamically configurable.

TYPES OF OPERATING SYSTEMS:

Depending on the type of kernel and kernel services, purpose and type of computing system, Operating Systems are classified into different types.

General Purpose Operating System (GPOS):

The operating systems, which are deployed in general computing systems, are referred as *GPOS*. The GPOSs are often quite non-deterministic in behavior.

- Windows 10/8.x/XP/MS-DOS, etc., are examples of GPIOs.

Real Time Operating System (RTOS):

Real Time implies deterministic in timing behavior.

- RTOS services consumes only known and expected amounts of time regardless the number of services.
- RTOS implements policies and rules concerning time-critical allocation of a system's resources.
- RTOS decides which applications should run in which order and how much time needs to be allocated for each application.
    - Windows Embedded Compact, QNX, VxWorks MicroC/OS-II, etc., are examples of RTOSs.

The Real-Time kernel: The kernel of a Real-Time OS is referred as Real-Time kernel. The Real-Time kernel is highly specialized and it contains only the minimal set of services required for running user applications/ tasks. The basic functions of a Real-Time kernel are listed below:

- Task/ Process management
- Task/ Process scheduling
- Task/ Process synchronization
- Error/ Exception handling
- Memory management
- Interrupt handling
- Time management.

- ***Task/ Process Management:*** Deals with setting up the memory space for the tasks, loading the task's code into the memory space, allocating system resources and setting up a *Task Control Block (TCB)* for the task and task/process termination/deletion.
    - A Task Control Block (TCB) is used for holding the information corresponding to a task. TCB usually contains the following set of information:
        - Task ID: Task Identification Number
        - Task State: The current state of the task. (E.g. State = 'Ready' for a task which is ready to execute)

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

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

- ***Task/ Process Scheduling:*** Deals with sharing the CPU among various tasks/ processes. A kernel application called '*Scheduler*' handles the task scheduling. Scheduler is an algorithm implementation, which performs the efficient and optimal scheduling of tasks to provide a deterministic behavior.

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

- ***Error/ Exception Handling:*** Deals with registering and handling the errors occurred/ exceptions rose during the execution of tasks.
  - Insufficient memory, timeouts, deadlocks, deadline missing, bus error, divide by zero, unknown instruction execution etc, are examples of errors/exceptions.
  - Errors/ Exceptions can happen at the kernel level services or at task level.
    - *Deadlock* is an example for kernel level exception, whereas *timeout* is an example for a task level exception.
      - ✓ Deadlock is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.
      - ✓ Timeouts and retry are two techniques used together. The tasks retries an event/ message certain number of times; if no response is received after exhausting the limit, the feature might be aborted.
  - The OS kernel gives the information about the error in the form of a system call (API).

- ***Memory Management:*** The memory management function of an RTOS kernel is slightly different compared to the General Purpose Operating Systems.
    - In general, the memory allocation time increases depending on the size of the block of memory need to be allocated and the state of the allocated memory block. RTOS achieves predictable timing and deterministic behavior, by compromising the effectiveness of memory allocation.
    - RTOS generally uses '*block*' based memory allocation technique, instead of the usual dynamic memory allocation techniques used by the GPOS. RTOS kernel uses blocks of fixed size of dynamic memory and the block is allocated for a task on a need basis. The blocks are stored in a '*Free buffer Queue*'.
    - Most of the RTOS kernels allow tasks to access any of the memory blocks without any memory protection to achieve predictable timing and avoid the timing overheads. Some commercial RTOS kernels allow memory protection as optional and the kernel enters a *fail-safe mode* when an illegal memory access occurs.
    - The memory management function a block of fixed memory is always allocated for tasks on need basis and it is taken as a unit. Hence, there will not be any memory fragmentation issues.

- ***Interrupt Handling:*** Deals with the handling of various interrupts. Interrupts inform the processor that an external device or an associated task requires immediate attention of the CPU.
    - Interrupts can be either Synchronous or Asynchronous.
        - Interrupts which occurs in sync with the currently executing task is known as Synchronous interrupts. Usually the software interrupts fall under the *Synchronous Interrupt* category.
            - ✓ Divide by zero, memory segmentation error etc are examples of Synchronous interrupts.
        - For synchronous interrupts, the interrupt handler runs in the same context of the interrupting task.
        - Interrupts which occurs at any point of execution of any task, and are not in sync with the currently executing task are *Asynchronous interrupts*.
            - ✓ Timer overflow interrupts, serial data reception/ transmission interrupts etc., are examples for asynchronous interrupts.
        - For asynchronous interrupts, the interrupt handler is usually written as separate task (depends on OS Kernel implementation) and it runs in a different context. Hence, a context switch happens while handling the asynchronous interrupts.

- Priority levels can be assigned to the interrupts and each interrupts can be enabled or disabled individually. Most of the RTOS kernel implements '*Nested Interrupts*' architecture.

- ***Time Management:*** Accurate time management is essential for providing precise time reference for all applications. The time reference to kernel is provided by a high-resolution Real Time Clock (RTC) hardware chip (hardware timer).
  - The hardware timer is programmed to interrupt the processor/ controller at a fixed rate. This timer interrupt is referred as '*Timer tick*'. The 'Timer tick' is taken as the timing reference by the kernel. The 'Timer tick' interval may vary depending on the hardware timer. Usually, the 'Timer tick' varies in the microseconds range. The time parameters for tasks are expressed as the multiples of the 'Timer tick'.
  - The System time is updated based on the 'Timer tick'. If the System time register is 32 bits wide and the 'Timer tick' interval is 1 microsecond, the System time register will reset in;

    $2^{32} * 10^{-6} / (24 * 60 * 60) = \sim 0.0497$ Days = 1.19 Hours
  - If the 'Timer tick' interval is 1 millisecond, the System time register will reset in

    $2^{32} * 10^{-3} / (24 * 60 * 60) = 49.7$ Days = ~ 50 Days
  - The 'Timer tick' interrupt is handled by the '*Timer Interrupt*' handler of kernel. The 'Timer tick' interrupt can be utilized for implementing the following actions:
    - Save the current context (Context of the currently executing task)
    - Increment the System time register by one. Generate timing error and reset the System time register if the timer tick count is greater than the maximum range available for System time register.
    - Update the timers implemented in kernel (Increment or decrement the timer registers for each timer depending on the count direction setting for each register. Increment registers with count direction setting = '*count up*' and decrement registers with count direction setting = '*count down*')
    - Activate the periodic tasks, which are in the idle state
    - Invoke the scheduler and schedule the tasks again based on the scheduling algorithm
    - Delete all the terminated tasks and their associated data structures (TCBs)
    - Load the context for the first task in the ready queue. Due to the re-scheduling, the ready task might be changed to a new one from the task, which was pre-empted by the '*Timer Interrupt*' task.

- ***Hard Real-Time:*** A Real Time Operating Systems which strictly adheres to the timing constraints for a task is referred as *hard real-time* systems. A Hard Real Time system must meet the deadlines for a task without any slippage. Missing any deadline may produce catastrophic results for Hard Real Time Systems, including permanent data lose and irrecoverable damages to the system/users.
    - Hard real-time systems emphasize on the principle 'A late answer is a wrong answer'.
        - For example, Air bag control systems and Anti-lock Brake Systems (ABS) of vehicles are typical examples of Hard Real Time Systems.
    - Most of the Hard Real Time Systems are automatic.

- ***Soft Real-Time:*** Real Time Operating Systems that does not guarantee meeting deadlines, but, offer the best effort to meet the deadline are referred as *soft real-time* systems. Missing deadlines for tasks are acceptable if the frequency of deadline missing is within the compliance limit of the Quality of Service (QoS).
    - Soft real-time system emphasizes on the principle 'A late answer is an acceptable answer, but it could have done bit faster'.
    - Automatic Teller Machine (ATM) is a typical example of Soft Real Time System. If the ATM takes a few seconds more than the ideal operation time, nothing fatal happens.

TASKS, PROCESSES AND THREADS:

The term '*task*' refers to something that needs to be done. In the Operating System context, a *task* is defined as the program in execution and the related information maintained by the Operating system for the program. Task is also known as '*Job*' in the operating system context. A program or part of it in execution is also called a '*Process*'.

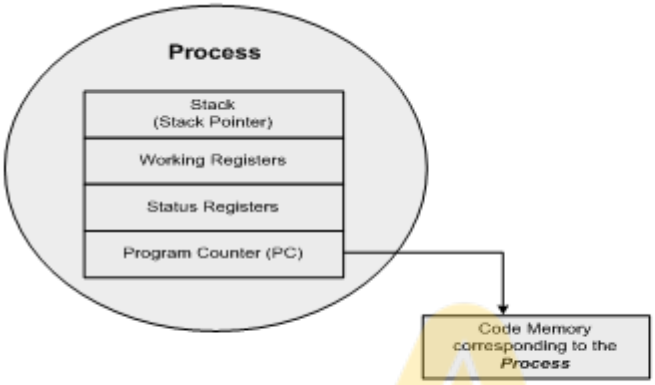
- The terms '*Task*', '*Job*' and '*Process*' refer to the same entity in the Operating System context and most often they are used interchangeably.

Process:

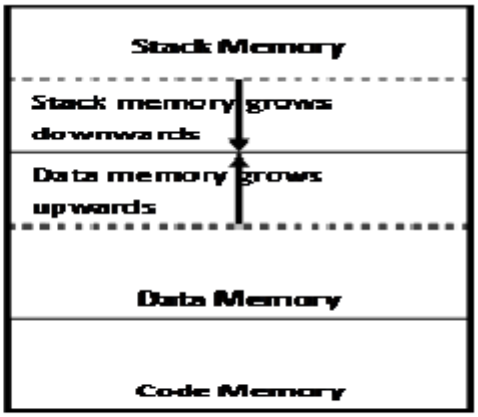
A 'Process' is a program, or part of it, in execution. Process is also known as an instance of a program in execution. A process requires various system resources like CPU for executing the process, memory for storing the code corresponding to the process and associated variables, I/O devices for information exchange etc.

- **Structure of a Processes:** The concept of 'Process' leads to concurrent execution of tasks and thereby, efficient utilization of the CPU and other system resources. Concurrent execution is achieved through the sharing of CPU among the processes.

- A process mimics a processor in properties and holds a set of registers, process status, a Program Counter (PC) to point to the next executable instruction of the process, a stack for holding the local variables associated with the process and the code corresponding to the process. This can be visualized as shown in the following Figure.



Structure of a Process

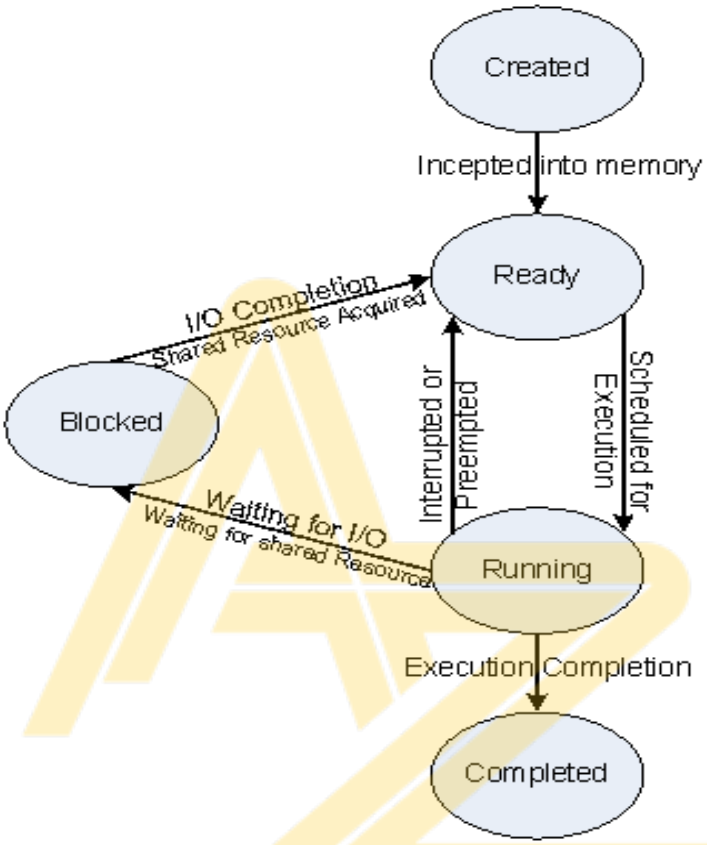


Memory Oraganization of a Process

- A process, which inherits all the properties of the CPU, can be considered as a virtual processor, awaiting its turn to have its properties switched into the physical processor. When the process gets its turn, its registers and Program Counter register becomes mapped to the physical registers of the CPU.
- The memory occupied by the process is segregated into three regions namely; Stack memory, Data memory and Code memory (Figure, shown above).
  - The '*Stack*' memory holds all temporary data such as variables local to the process.
  - The '*Data*' memory holds all global data for the process.
  - The '*Code*' memory contains the program code (instructions) corresponding to the process.
- On loading a process into the main memory, a specific area of memory is allocated for the process. The stack memory usually starts at the highest memory address from the memory area allocated for the process.

- ***Process States & State Transition:*** The creation of a process to its termination is not a single step operation. The process traverses through a series of states during its transition from the newly created state to the terminated state.
  - The cycle through which a process changes its state from '*newly created*' to '*execution completed*' is known as '*Process Life Cycle*'.

- The various states through which a process traverses through during a Process Life Cycle indicates the current status of the process with respect to time and also provides information on what it is allowed to do next.
- The transition of a process from one state to another is known as '*State transition*'. The Process states and state transition representation are shown in the following Figure.



- *Created State:* The state at which a process is being created is referred as 'Created State'. The Operating System recognizes a process in the 'Created State' but no resources are allocated to the process.
- *Ready State:* The state, where a process is incepted into the memory and awaiting the processor time for execution, is known as 'Ready State'. At this stage, the process is placed in the 'Ready list' queue maintained by the OS.
- *Running State:* The state where in the source code instructions corresponding to the process is being executed is called 'Running State'. Running state is the state at which the process execution happens.
- *Blocked State/ Wait State:* Refers to a state where a running process is temporarily suspended from execution and does not have immediate access to resources. The blocked state might have invoked by various conditions like- the process enters a wait state for an event to occur (E.g. Waiting for user inputs such

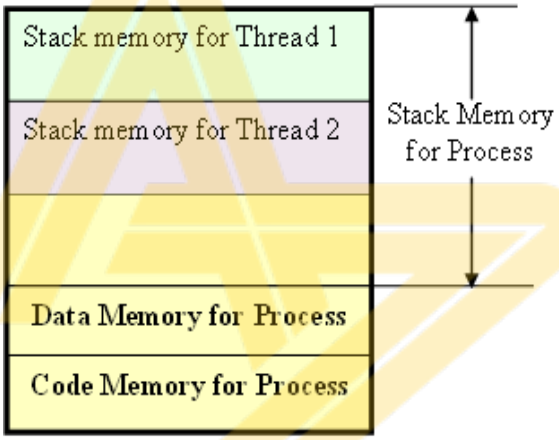
as keyboard input) or waiting for getting access to a shared resource like semaphore, mutex etc.

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

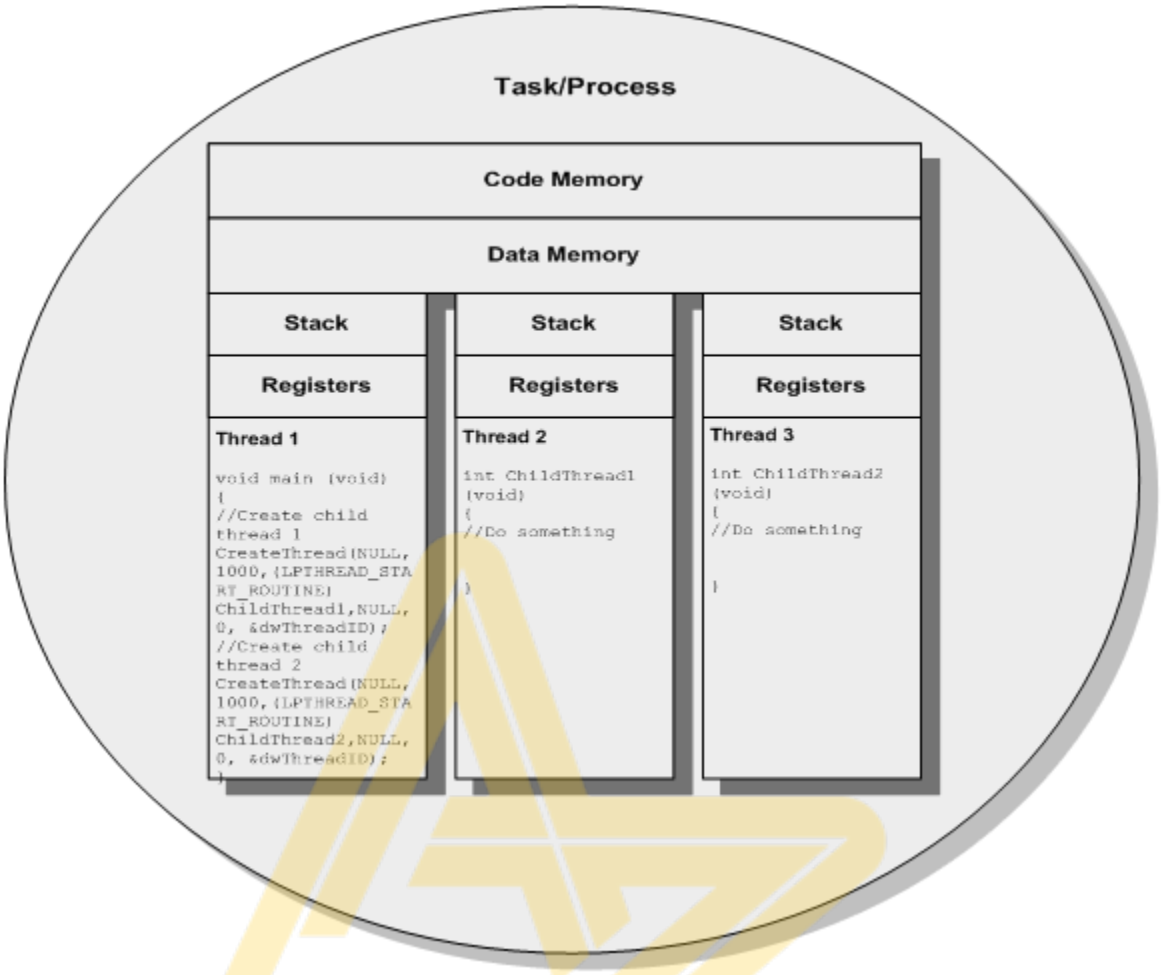
Threads:

A *thread* is the primitive that can execute code. A *thread* is a single sequential flow of control within a process. A *thread* is also known as lightweight process.

- A process can have many threads of execution. Different threads, which are part of a process, share the same address space; meaning they share the data memory, code memory and heap memory area.
- Threads maintain their own thread status (CPU register values), Program Counter (PC) and stack. The memory model for a process and its associated threads are given in the following figure.



- ***The Concept of Multithreading:*** The process is split into multiple threads, which executes a portion of the process; there will be a main thread and rest of the threads will be created within the main thread.
  - The multithreaded architecture of a process can be visualized with the thread-process diagram, shown below.
  - Use of multiple threads to execute a process brings the following advantage:
    - *Better memory utilization:* Multiple threads of the same process share the address space for data memory. This also reduces the complexity of inter thread communication since variables can be shared across the threads.
    - Since the process is split into different threads, when one thread enters a wait state, the CPU can be utilized by other threads of the process that do not require the event, which the other thread is waiting, for processing. This speeds up the execution of the process.
    - Efficient CPU utilization. The CPU is engaged all time.



- ***Thread Standards:*** Thread standards deal with the different standards available for thread creation and management. These standards are utilized by the Operating Systems for thread creation and thread management. It is a set of thread class libraries. The commonly available thread class libraries are –
    - *POSIX Threads:* POSIX stands for Portable Operating System Interface. The POSIX.4 standard deals with the Real Time extensions and POSIX.4a standard deals with thread extensions. The POSIX standard library for thread creation and management is '*Pthreads*'. '*Pthreads*' library defines the set of POSIX thread creation and management functions in '*C*' language. (*Example 1 – Self study*).
    - *Win32 Threads:* Win32 threads are the threads supported by various flavors of Windows Operating Systems. The Win32 Application Programming Interface (Win32 API) libraries provide the standard set of Win32 thread creation and management functions. Win32 threads are created with the API.
    - *Java Threads:* Java threads are the threads supported by Java programming Language. The java thread class 'Thread' is defined in the package '*java.lang*'. This package needs to be imported for using the thread creation functions supported by the Java thread class.

There are two ways of creating threads in Java: Either by extending the base 'Thread' class or by implementing an interface. Extending the thread class allows inheriting the methods and variables of the parent class (Thread class) only whereas interface allows a way to achieve the requirements for a set of classes.

- ***Thread Pre-emption:*** *Thread pre-emption* is the act of pre-empting the currently cunning thread (stopping temporarily). It is dependent on the Operating System. It is performed for sharing the CPU time among all the threads. The execution switching among threads are known as '*Thread context switching*'. Threads falls into one of the following types:
  - *User Level Thread:* User level threads do not have kernel/ Operating System support and they exist only in the running process. A process may have multiple user level threads; but the OS threats it as single thread and will not switch the execution among the different threads of it. It is the responsibility of the process to schedule each thread as and when required. Hence, user level threads are non-preemptive at thread level from OS perspective.
  - *Kernel Level/ System Level Thread:* Kernel level threads are individual units of execution, which the OS treats as separate threads. The OS interrupts the execution of the currently running kernel thread and switches the execution to another kernel thread based on the scheduling policies implemented by the OS.
    - The execution switching (thread context switching) of user level threads happen only when the currently executing user level thread is voluntarily blocked. Hence, no OS intervention and system calls are involved in the context switching of user level threads. This makes context switching of user level threads very fast.
    - Kernel level threads involve lots of kernel overhead and involve system calls for context switching. However, kernel threads maintain a clear layer of abstraction and allow threads to use system calls independently.
    - There are many ways for binding user level threads with kernel/ system level threads; which are explained below:
      - Many-to-One Model: Many user level threads are mapped to a single kernel thread. The kernel treats all user level threads as single thread and the execution switching among the user level threads happens when a currently executing user level thread voluntarily blocks itself or relinquishes the CPU. Solaris Green threads and GNU Portable Threads are examples for this.

- One-to-One Model: Each user level thread is bonded to a kernel/ system level thread. Windows XP/NT/2000 and Linux threads are examples of One-to-One thread models.
- Many-to-Many Model: In this model many user level threads are allowed to be mapped to many kernel threads. Windows NT/2000 with *ThreadFiber* package is an example for this.

- ***Thread versus Process:***

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

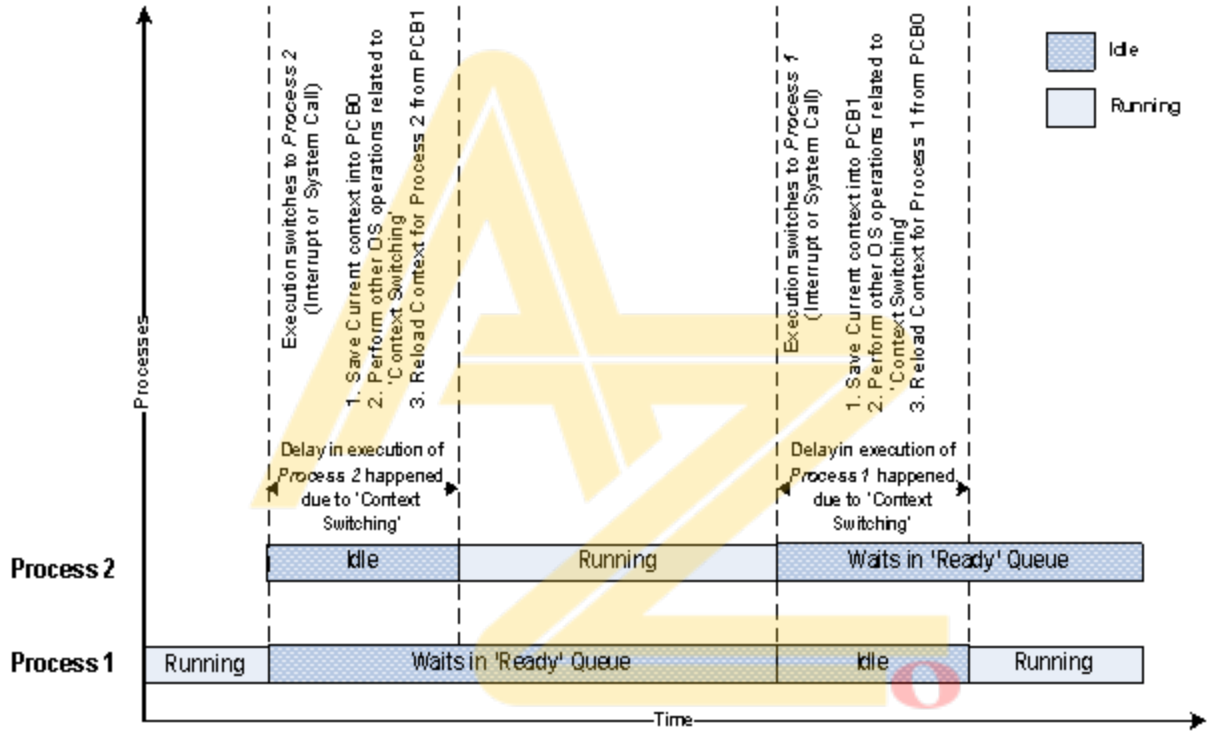
MULTIPROCESSING AND MULTITASKING:

The ability to execute multiple processes simultaneously is referred as *multiprocessing*. Systems which are capable of performing multiprocessing are known as *multiprocessor systems*.

- Multiprocessor systems possess multiple CPUs and can execute multiple processes simultaneously.
- The ability of the Operating System to have multiple programs in memory, which are ready for execution, is referred as *multiprogramming*. In a uni-processor system, it is not possible to execute multiple processes simultaneously.

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

- Multitasking involves '*Context switching*' (see the following Figure), '*Context saving*' and '*Context retrieval*'.
    - The act of switching CPU among the processes or changing the current execution context is known as '*Context switching*'.
    - The act of saving the current context (details like Register details, Memory details, System Resource Usage details, Execution details, etc.) for the currently running processes at the time of CPU switching is known as '*Context saving*'.
    - The process of retrieving the saved context details for a process, which is going to be executed due to CPU switching, is known as '*Context retrieval*'.



Types of Multitasking:

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

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

- ***Preemptive Multitasking:*** Preemptive multitasking ensures that every task/ process gets a chance to execute. When and how much time a process gets is dependent on the implementation of the preemptive scheduling. As the name indicates, in preemptive multitasking, the currently running task/process is preempted to give a chance to other tasks/process to execute. The preemption of task may be based on time slots or task/ process priority.
- ***Non-preemptive Multitasking:*** The process/ task, which is currently given the CPU time, is allowed to execute until it terminates (enters the '*Completed*' state) or enters the '*Blocked/ Wait*' state, waiting for an I/O. The co-operative and non-preemptive multitasking differs in their behavior when they are in the 'Blocked/Wait' state. In co-operative multitasking, the currently executing process/task need not relinquish the CPU when it enters the '*Blocked/ Wait*' sate, waiting for an I/O, or a shared resource access or an event to occur whereas in non-preemptive multitasking the currently executing task relinquishes the CPU when it waits for an I/O.

TASK COMMUNICATION:

In a multitasking system, multiple tasks/ processes run concurrently (in pseudo parallelism) and each process may or may not interact between. Based on the degree of interaction, the processes/ tasks running on an OS are classified as –

- ***Co-operating Processes:*** In the co-operating interaction model, one process requires the inputs from other processes to complete its execution.
- ***Competing Processes:*** The competing processes do not share anything among themselves but they share the system resources. The competing processes compete for the system resources such as file, display device, etc.
    - The co-operating processes exchanges information and communicate through the following methods:
        - *Co-operation through sharing:* Exchange data through some shared resources.
        - *Co-operation through Communication:* No data is shared between the processes. But they communicate for execution synchronization.

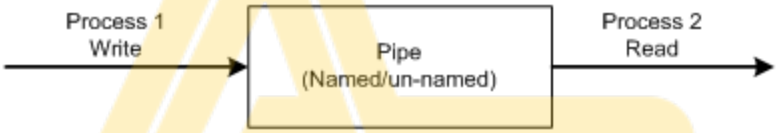
The mechanism through which tasks/ processes communicate each other is known as *Inter Process/ Task Communication (IPC)*. IPC is essential for process co-ordination. The various types of IPC mechanisms adopted by process are kernel (Operating System) dependent. They are explained below.

IPC Mechanism - Shared Memory:

Processes share some area of the memory to communicate among them (see the following Figure). Information to be communicated by the process is written to the shared memory area. Processes which require this information can read the same from the shared memory area.

- The implementation of shared memory is kernel dependent. Different mechanisms are adopted by different kernels for implementing this, a few among are s follows:

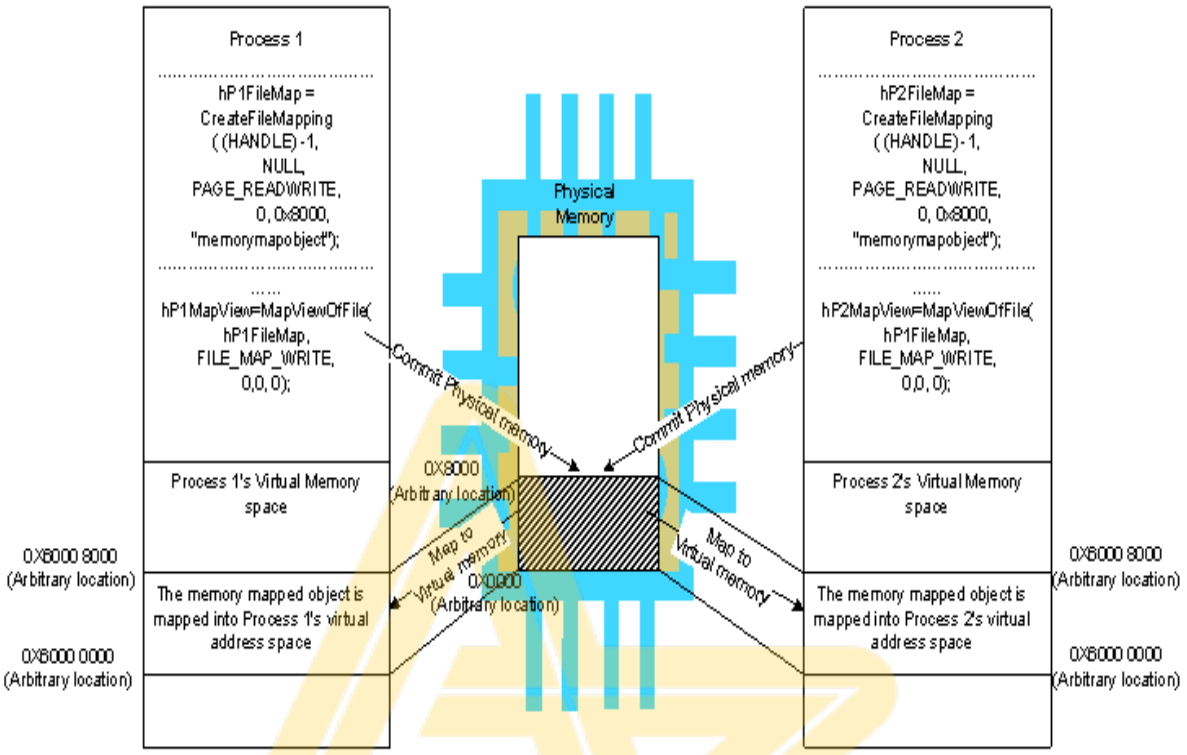
1. ***Pipes:*** '*Pipe*' is a section of the shared memory used by processes for communicating. Pipes follow the client-server architecture. A process which creates a pipe is known as *pipe server* and a process which connects to a pipe is known as *pipe client*. A pipe can be considered as a medium for information flow and has two conceptual ends. It can be *unidirectional*, allowing information flow in one direction or *bidirectional* allowing bi-directional information flow. A unidirectional pipe allows the process connecting at one end of the pipe to write to the pipe and the process connected at the other end of the pipe to read the data, whereas a bi-directional pipe allows both reading and writing at one end. The unidirectional pipe can be visualized as



   - The implementation of 'Pipes' is OS dependent. Microsoft® Windows Desktop Operating Systems support two types of 'Pipes' for Inter Process Communication. Namely;
     - *Anonymous Pipes:* The anonymous pipes are unnamed, unidirectional pipes used for data transfer between two processes.
     - *Named Pipes:* Named pipe is a named, unidirectional or bi-directional pipe for data exchange between processes. Like anonymous pipes, the process which creates the named pipe is known as pipe server. A process which connects to the named pipe is known as pipe client. With named pipes, any process can act as both client and server allowing point-to-point communication. Named pipes can be used for communicating between processes running on the same machine or between processes running on different machines connected to a network.

2. ***Memory Mapped Objects:*** Memory mapped object is a shared memory technique adopted by certain Real Time Operating Systems for allocating a shared block of memory which can be accessed by multiple process simultaneously. In this approach, a mapping object is created and physical storage for it is reserved and committed. A process can map the entire committed physical area or a block of it to its virtual address space. All read and write operation to this virtual address space by a process is directed to its committed physical area.

Any process which wants to share data with other processes can map the physical memory area of the mapped object to its virtual memory space and use it for sharing the data. The concept of memory mapped object is shown bellow.



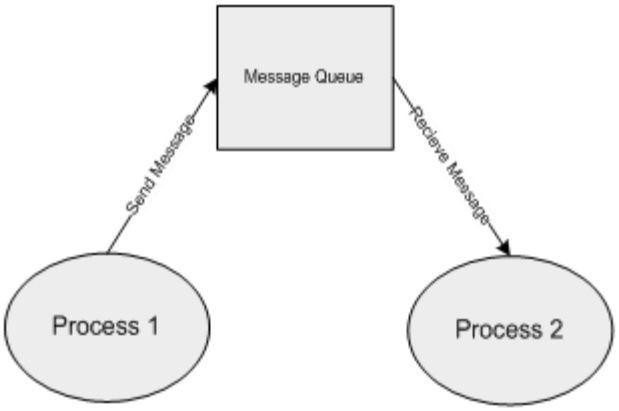
IPC Mechanism - Message Passing:

Message passing is a/ an synchronous/ asynchronous information exchange mechanism for Inter Process/ Thread Communication. The major difference between shared memory and message passing technique is

- Through shared memory lots of data can be shared whereas only limited amount of info/ data is passed through message passing.
- Message passing is relatively fast and free from the synchronization overheads compared to shared memory.

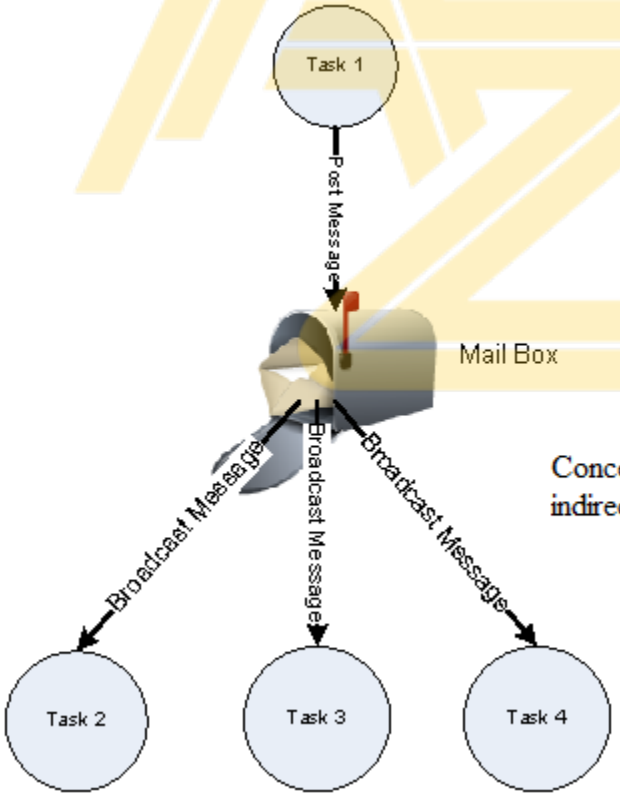
Based on the message passing operation between the processes, message passing is classified into –

1. ***Message Queues:*** Process which wants to talk to another process posts the message to a First-In-First-Out (FIFO) queue called '*Message queue*', which stores the messages temporarily in a system defined memory object, to pass it to the desired process. Messages are sent and received through *send (Name of the process to which the message is to be sent, message*) and *receive* (*Name of the process from which the message is to be received, message*) methods. The messages are exchanged through a message queue. The implementation of the message queue, send and receive methods are OS kernel dependent.



Concept of message queue based indirect messaging for IPC

2. ***Mailbox:*** Mailbox is a special implementation of message queue. Usually used for one way communication, only a single message is exchanged through mailbox whereas 'message queue' can be used for exchanging multiple messages. One task/process creates the mailbox and other tasks/process can subscribe to this mailbox for getting message notification. The implementation of the mailbox is OS kernel dependent. The MicroC/ OS-II RTOS implements mailbox as a mechanism for inter task communication

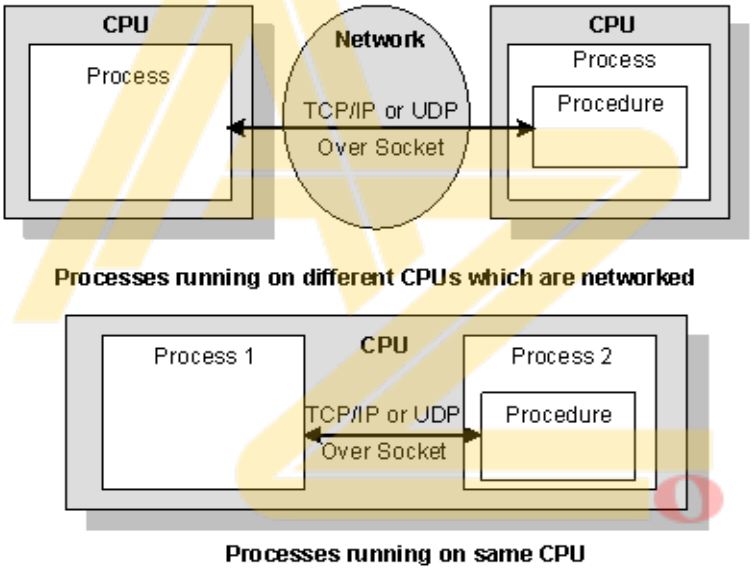


Concept of Mailbox based indirect messaging for IPC

3. ***Signalling:*** Signals are used for an asynchronous notification mechanism. The signal mainly used for the execution synchronization of tasks process/ tasks. Signals do not carry any data and are not queued. The implementation of signals is OS kernel dependent and VxWorks RTOS kernel implements 'signals' for inter process communication.

IPC Mechanism - Remote Procedure Call (IPC) and Sockets: Remote Procedure Call is the Inter Process Communication (IPC) mechanism used by a process, to call a procedure of another process running on the same CPU or on a different CPU which is interconnected in a network. In the object oriented language terminology, RPC is also known as *Remote Invocation* or *Remote Method Invocation* (*RMI*). The CPU/ process containing the procedure which needs to be invoked remotely is known as server. The CPU/ process which initiates an RPC request is known as client.

- In order to make the RPC communication compatible across all platforms, it should stick on to certain standard formats.
- Interface Definition Language (IDL) defines the interfaces for RPC. Microsoft Interface Definition Language (MIDL) is the IDL implementation from Microsoft for all Microsoft platforms.
- The RPC communication can be either Synchronous (Blocking) or Asynchronous (Non-blocking).



Concept of Remote Procedure Call (RPC) for IPC

Sockets are used for RPC communication. ***Socket*** is a logical endpoint in a two-way communication link between two applications running on a network. A port number is associated with a socket so that the network layer of the communication channel can deliver the data to the designated application. Sockets are of different types namely; Internet sockets (INET), UNIX sockets, etc.

- The *INET Socket* works on Internet Communication protocol. TCP/ IP, UDP, etc., are the communication protocols used by INET sockets.
- INET sockets are classified into:
    - *Stream Sockets:* are connection oriented and they use TCP to establish a reliable connection.
    - *Datagram Sockets:* rely on UDP for establishing a connection.

TASK SYNCHRONIZATION:

In a multitasking environment, multiple processes run concurrently and share the system resources. Also, each process may communicate with each other with different IPC mechanisms. Hence, there may be situations that; two processes try to access a shared memory area, where one process tries to write to the memory location when the other process is trying to read from the same memory location. This will lead to unexpected results.

The solution is, make each process aware of access of a shared resource. The act of making the processes aware of the access of shared resources by each process to avoid conflicts is known as "*Task/ Process Synchronization*".

Task/ Process Synchronization is essential for –

1. Avoiding conflicts in resource access (racing, deadlock, etc.) in multitasking environment.
2. Ensuring proper sequence of operation across processes.
3. Establish proper communication between processes.

The code memory area which holds the program instructions (piece of code) for accessing a shared resource is known as 'Critical Section'. In order to synchronize the access to shared resources, the access to the critical section should be exclusive.

Task Communication/ Synchronization Issues:

Various synchronization issues may arise in a multitasking environment, if processes are not synchronized properly in shared resource access, such as:

1. ***Racing:*** Look into the following piece of code:

```
#include <stdio.h>
//*****************************************************************
//counter is an integer variable and Buffer is a byte array shared
//between two processes Process A and Process B.
char Buffer [10] = {1,2,3,4,5,6,7,8,9,10};
short int counter = 0;
//*****************************************************************
// Process A
Void Process_A (void)
{
        int i;
        for (i =0; i<5; i++)
        {
                if (Buffer [i] > 0)
```

```
                counter++;
        }
}
//*********************************************************************
// Process B
Void Process_B (void)
{
        int j;
        for (j =5; j<10; j++)
        {
                if (Buffer[j] > 0)
                counter++;
        }
}
//*********************************************************************
//Main Thread.
int main()
{
        DWORD id;
        CreateThread (NULL, 0, (LPTHREAD_START_ROUTINE) Process_A,
(LPVOID) 0, 0, &id);
        CreateThread (NULL, 0, (LPTHREAD_START_ROUTINE) Process_B,
(LPVOID) 0, 0, &id);
        Sleep (100000);
        return 0;
}
```

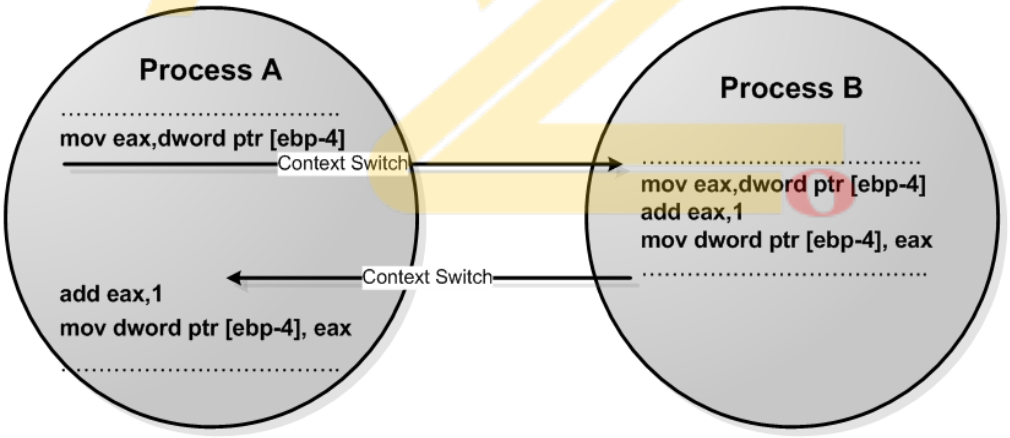
- From a programmer perspective, the value of counter will be 10 at the end of execution of processes A & B. But it need not be always.
  - The program statement *counter++;* looks like a single statement from a high level programming language (C Language) perspective. The low level implementation of this statement is dependent on the underlying processor instruction set and the (cross) compiler in use. The low level implementation of the high level program statement *counter++;* under Windows XP operating system running on an Intel Centrino Duo processor is given below.

```
mov eax, dword ptr [ebp-4]      ;Load counter in Accumulator
add eax, 1                      ; Increment Accumulator by 1
mov dword ptr [ebp-4], eax      ;Store counter with Accumulator
```

- At the processor instruction level, the value of the variable counter is loaded to the Accumulator register (EAX Register). The memory variable counter is represented using a pointer. The base pointer register (EBP Register) is used for pointing to the memory variable counter. After loading the contents of the variable counter to the Accumulator, the Accumulator content is incremented by one using the add instruction. Finally the content of Accumulator is loaded to the memory location which represents the variable counter. Both the processes; Process A and Process B contain the program statement *counter++;* Translating this into the machine instruction.

Process A	Process B
mov eax,dword ptr [ebp-4]	mov eax, dword ptr [ebp-4]
add eax, 1	add eax, 1
mov dword ptr [e bp-4], eax	mov dword ptr [ebp-4], eax

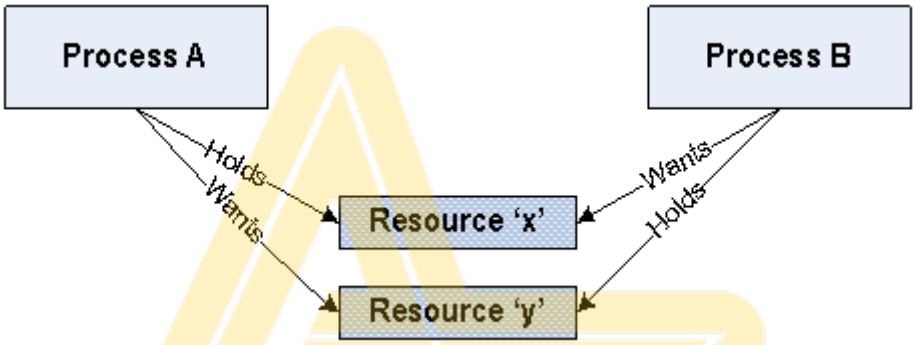
- Imagine a situation where a process switching (context switching) happens from Process A to Process B when Process A is executing the *counter++;* statement. Process A accomplishes the *counter++;* statement through three different low level instructions. Now imagine that the process switching happened at the point, where Process A executed the low level instruction *mov eax, dword ptr [ebp-4]* and is about to execute the next instruction *add eax, 1*. The scenario is illustrated in the following Figure.



- Process B increments the shared variable 'counter' in the middle of the operation where Process A tries to increment it. When Process A gets the CPU time for execution, it starts from the point where it got interrupted (If Process B is also using the same registers *eax* and *ebp* for executing *counter++;* instruction, the original content of these registers will be saved as part of context saving and it will be retrieved back as part of the context retrieval, when Process A gets the CPU for execution. Hence the content of *eax* and *ebp* remains intact irrespective of context switching). Though the variable counter is incremented by Process B,

Process A is unaware of it and it increments the variable with the old value. This leads to the loss of one increment for the variable counter.

2. ***Deadlock:*** *Deadlock* is the condition in which a process is waiting for a resource held by another process which is waiting for a resource held by the first process; hence, none of the processes are able to make any progress in their execution.
   - Process A holds a resource '*x*' and it wants a resource '*y*' held by Process B. Process B is currently holding resource '*y*' and it wants the resource '*x*' which is currently held by Process A. Both hold the respective resources and they compete each other to get the resource held by the respective processes.



   - ***Conditions Favoring Deadlock:***
     - *Mutual Exclusion:* The criteria that only one process can hold a resource at a time. Meaning processes should access shared resources with mutual exclusion. Typical example is the accessing of display device in an embedded device.
     - *Hold & Wait:* The condition in which a process holds a shared resource by acquiring the lock controlling the shared access and waiting for additional resources held by other processes.
     - *No Resource Preemption:* The criteria that Operating System cannot take back a resource from a process which is currently holding it and the resource can only be released voluntarily by the process holding it.
     - *Circular Wait:* A process is waiting for a resource which is currently held by another process which in turn is waiting for a resource held by the first process. In general there exists a set of waiting process P0, P1 …. Pn with P0 is waiting for a resource held by P1 and P1 is waiting for a resource held by P0, ……,Pn is waiting for a resource held by P0 and P0 is waiting for a resource held by Pn and so on… This forms a circular wait queue.
   - ***Handling Deadlock:*** The OS may adopt any of the following techniques to detect and prevent deadlock conditions.

- *Ignore Deadlocks:* Always assume that the system design is deadlock free. This is acceptable for the reason the cost of removing a deadlock is large compared to the chance of happening a deadlock. UNIX is an example for an OS following this principle. A life critical system cannot pretend that it is deadlock free for any reason.
- *Detect and Recover:* This approach suggests the detection of a deadlock situation and recovery from it.
    - This is similar to the deadlock condition that may arise at a traffic junction. When the vehicles from different directions compete to cross the junction, deadlock (traffic jam) condition is resulted. Once a deadlock (traffic jam) is happened at the junction, the only solution is to back up the vehicles from one direction and allow the vehicles from opposite direction to cross the junction. If the traffic is too high, lots of vehicles may have to be backed up to resolve the traffic jam. This technique is also known as '*back up cars*' technique.
    - Operating Systems keep a resource graph in their memory. The resource graph is updated on each resource request and release. A deadlock condition can be detected by analyzing the resource graph by graph analyzer algorithms. Once a deadlock condition is detected, the system can terminate a process or preempt the resource to break the deadlocking cycle.

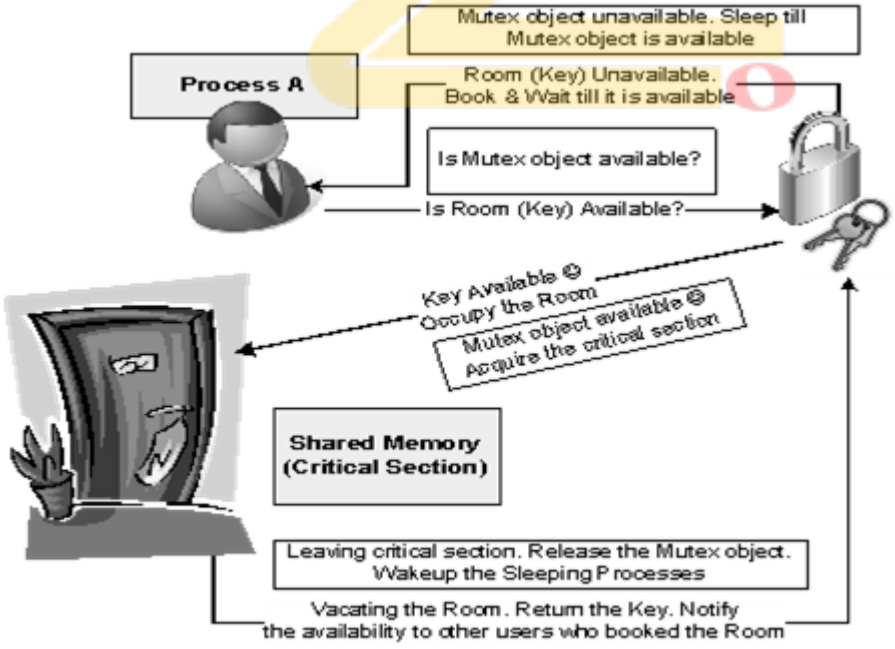
- ***Avoid Deadlocks:*** Deadlock is avoided by the careful resource allocation techniques by the Operating System. It is similar to the traffic light mechanism at junctions to avoid the traffic jams.
- ***Prevent Deadlocks:*** Prevent the deadlock condition by negating one of the four conditions favoring the deadlock situation.
- Ensure that a process does not hold any other resources when it requests a resource. This can be achieved by implementing the following set of rules/ guidelines in allocating resources to processes.
    1. A process must request all its required resource and the resources should be allocated before the process begins its execution.
    2. Grant resource allocation requests from processes only if the process does not hold a resource currently.
- Ensure that resource preemption (resource releasing) is possible at operating system level. This can be achieved by implementing the following set of rules/ guidelines in resources allocation and releasing:
    1. Release all the resources currently held by a process if a request made by the process for a new resource is not able to fulfill immediately.

2. Add the resources which are preempted (released) to a resource list describing the resources which the process requires to complete its execution.
3. Reschedule the process for execution only when the process gets its old resources and the new resource which is requested by the process.

Task Synchronization Techniques:

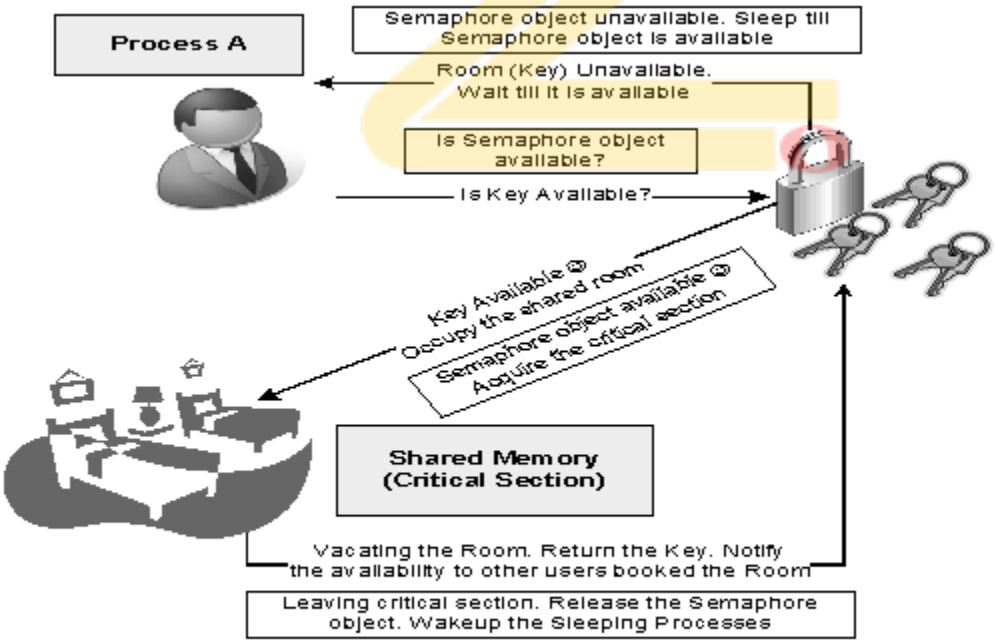
The technique used for task synchronization in a multitasking environment is *mutual exclusion*. Mutual exclusion blocks a process. Based on the behavior of blocked process, mutual exclusion methods can be classified into two categories: Mutual exclusion through busy waiting/ spin lock & Mutual exclusion through sleep & wakeup.

- ***Semaphore:*** *Semaphore* is a sleep and wakeup based mutual exclusion implementation for shared resource access. Semaphore is a system resource; and a process which wants to access the shared resource can first acquire this system object to indicate the other processes which wants the shared resource that the shared resource is currently in use by it.
- The resources which are shared among a process can be either for exclusive use by a process or for using by a number of processes at a time.
- The display device of an embedded system is a typical example of a shared resource which needs exclusive access by a process. The Hard disk (secondary storage) of a system is a typical example for sharing the resource among a limited number of multiple processes.
- Based on the implementation, Semaphores can be classified into *Binary Semaphore* and *Counting Semaphore*.



The concept of Binary Semaphore

- ***Binary Semaphore:*** Implements exclusive access to shared resource by allocating the resource to a single process at a time and not allowing the other processes to access it when it is being used by a process.
  - 'Only one process/ thread' can own the binary semaphore at a time.
  - The state of a 'binary semaphore' object is set to signaled when it is not owned by any process/ thread, and set to non-signaled when it is owned by any process/ thread.
  - The implementation of binary semaphore is OS kernel dependent. Under certain OS kernel it is referred as mutex.
- ***Counting Semaphore:*** Maintains a count between zero and a maximum value. It limits the usage of resource by a fixed number of processes/ threads.
- The count associated with a '*Semaphore object*' is decremented by one when a process/ thread acquires it and the count is incremented by one when a process/ thread releases the 'Semaphore object'.
- The state of the counting semaphore object is set to 'signaled' when the count of the object is greater than zero.
- The state of the 'Semaphore object' is set to non-signaled when the semaphore is acquired by the maximum number of processes/ threads that the semaphore can support (i.e. when the count associated with the 'Semaphore object' becomes zero).
- The creation and usage of 'counting semaphore object' is OS kernel dependent.



The concept of Counting Semaphore

HOW TO CHOOSE AN RTOS:

The decision of choosing an RTOS for an embedded design is very crucial. A lot of factors need to be analyzed carefully before making a decision on the selection of an RTOS. These factors can be either *functional* or *non-functional*.

Functional Requirements:

- *Processor Support:* It is not necessary that all RTOS's support all kinds of processor architecture. It is essential to ensure the processor support by the RTOS.
- *Memory Requirements:* The OS requires ROM memory for holding the OS files and it is normally stored in a non-volatile memory like FLASH. OS also requires working memory RAM for loading the OS services. Since embedded systems are memory constrained, it is essential to evaluate the minimal ROM and RAM requirements for the OS under consideration.
- *Real-time Capabilities:* It is not mandatory that the operating system for all embedded systems need to be Real-time and all embedded Operating systems-are 'Real-time' in behavior. The task/ process scheduling policies play an important role in the 'Real-time' behavior of an OS. Analyze the real-time capabilities of the OS under consideration and the standards met by the operating system for real-time capabilities.
- *Kernel and Interrupt Latency:* The kernel of the OS may disable interrupts while executing certain services and it may lead to interrupt latency. For an embedded system whose response requirements are high, this latency should be minimal.
- *Inter Process Communication and Task Synchronization:* The implementation of Inter Process Communication and Synchronization is OS kernel dependent. Certain kernels may provide a bunch of options whereas others provide very limited options. Certain kernels implement policies for avoiding priority inversion issues in resource sharing.
- *Modularization Support:* Most of the operating systems provide a bunch of features. At times it may not be necessary for an embedded product for its functioning. It is very useful if the OS supports moclularisation where in which the developer can choose the essential modules and re-compile the OS image for functioning. Windows CE is an example for a highly modular operating system.
- *Support for Networking and Communication:* The OS kernel may provide stack implementation and driver support for a bunch of communication interfaces and networking. Ensure that the OS under consideration provides support for all the interfaces required by the embedded product.
- *Development Language Support:* Certain operating systems include the run time libraries required for running applications written in languages like Java and C#. A Java Virtual Machine (JVM) customized for the Operating System is essential for running java applications. Similarly

the .NET Compact Framework (.NETCF) is required for running Microsoft .NET applications on top of the Operating System. The OS may include these components as built-in component, if not; check the availability of the same from a third party vendor or the OS under consideration.

Non-functional Requirements:

- *Custom Developed or Off the Shelf:* Depending on the OS requirement, it is possible to go for the complete development of an operating system suiting the embedded system needs or use an off the shelf, readily available operating system, which is either a commercial product or an Open Source product, which is in close match with the system requirements. Sometimes it may be possible to build the required features by customizing an Open source OS. The decision on which to select is purely de- pendent on the development cost, licensing fees for the OS, development time and availability of skilled resources.
- *Cost*: The total cost for developing or buying the OS and maintaining it in terms of commercial product and custom build needs to be evaluated before taking a decision on the selection of OS.
- *Development and Debugging Tools Availability:* The availability of development and debugging tools is a critical decision making factor in the selection of an OS for embedded design. Certain Operating Systems may be superior in performance, but the availability of tools for supporting the development may be limited. Explore the different tools available for the OS under consideration.
- *Ease of Use:* How easy it is to use a commercial RTOS is another important feature that needs to be considered in the RTOS selection.
- *After Sales:* For a commercial embedded RTOS, after sales in the fom1 of e-mail, on-call services etc., for bug fixes, critical patch updates and support for production issues, etc., should be analyzed thoroughly.

INTREGRATION AND TESTING OF EMBEDDED HARDWARE AND FIRMWARE

Integration testing of the embedded hardware and firmware is the immediate step following the embedded hardware and firmware development.

- The final embedded hardware constitute of a PCB with all necessary components affixed to it as per the original schematic diagram.
- Embedded firmware represents the control algorithm and configuration data necessary to implement the product requirements on the product. Embedded firmware will be in a target processor/ controller understandable format called machine language (sequence of ls and 0s-Binary).

- The target embedded hardware without embedding the firmware is a dumb device and cannot function properly. If you power up the hardware without embedding the firmware, the device may behave in an unpredicted manner.
- Both embedded hardware and firmware should be independently tested (Unit Tested) to ensure their proper functioning.
- Functioning of individual hardware sections can be done by writing small utilities which checks the operation of the specified part.
- The functionalities of embedded firmware can easily be checked by the simulator environment provided by the embedded firmware development tool's IDE. By simulating the firmware, the memory contents, register details, status of various flags and registers can easily be monitored and it gives an approximate picture of "What happens inside the processor/ controller and what are the states of various peripherals" when the firmware is running on the target hardware. The IDE gives necessary support for simulating the various inputs required from the external world, like inputting data on ports, generating an interrupt condition, etc.

INTEGRATION OF HARDWARE AND FIRMWARE:

Integration of hardware and firmware deals with the embedding of firmware into the target hardware board. It is the process of '*Embedding Intelligence*' to the product.

- The embedded processors/ controllers used in the target board may or may not have built in code memory. For non-operating system based embedded products, if the processor/ controller contain internal memory and the total size of the firmware is fitting into the code memory area, the code memory is downloaded into the target controller/ processor.
- If the processor/ controller does not support built in code memory or the size of the firmware is exceeding the memory size supported by the target processor/ controller, an external dedicated EPROM/ FLASH memory chip is used for holding the firmware. This chip is interfaced to the processor/ controller.

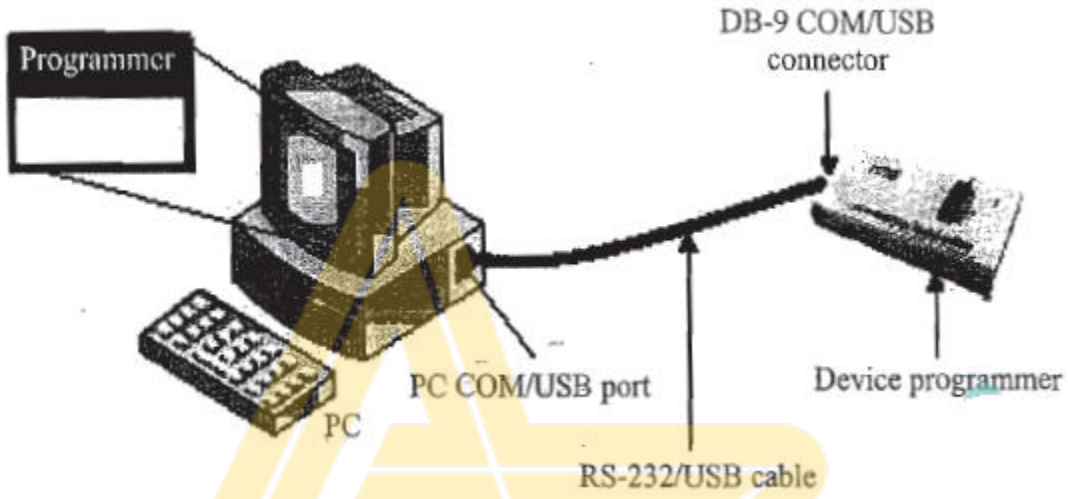
A variety of techniques are used for embedding the firmware into the target board. The commonly used firmware embedding techniques for a non-OS based embedded system are explained below. The non-OS based embedded systems store the firmware either in the on-chip processor/ controller memory or off-chip memory (FLASHI/ NVRAM, etc.).

Out-of-Circuit Programming:

Out-of-circuit programming is performed outside the target board. The processor or memory chip into which the firmware needs to be embedded is taken out of the target board and it is programmed with the help of a programming device.

The programming device is a dedicated unit which contains the necessary hardware circuit to generate the programming signals. Most of the programming devices available in the market are capable of programming different family of devices.

The programming device will be under the control of a utility program running on a PC. Usually the programming device is interfaced to the PC through RS-232C/USB/Parallel Port Interface. The commands to control the programmer are sent from the utility program to the programmer through the interface (see the following Figure).



The sequence of operations for embedding the firmware with a programmer is listed below:

1. Connect the programming device to the specified port of PC (USB/COM port/Parallel port)
2. Power up the device (Most of the programmers incorporate LED to indicate Device power up. Ensure that the power indication LED is ON)
3. Execute the programming utility on the PC and ensure proper connectivity is established between PC and programmer. In case of error turn off device power and try connecting it again
4. Unlock the ZIF socket by turning the lock pin
5. Insert the device to be programmed into the open socket as per the insert diagram shown on the programmer
6. Lock the ZIF socket
7. Select the device name from the list of supported devices
8. Load the hex file which is to be embedded into the device
9. Program the device by 'Program' option of utility program
10. Wait till the completion of programming operation (Till busy LED of programmer is off)
11. Ensure that programming is success by checking the status LED on the programmer (Usually 'Green' for success and 'Red' for error condition) or by noticing the feedback from the utility program
12. Unlock the ZIF socket and d take the device out of programmer.

Now the firmware is successfully embedded into the device. Insert the device into the board, power up the board and test it for the required functionalities. It is to be noted that the most of programmers support only Dual Inline Package (DIP) chips, since its ZIF socket is designed to accommodate only DIP chips.

Option for setting firmware protection will be available on the programming utility. If you really want the firmware to be protected against unwanted external access, and if the device is supporting memory protection, enable the memory protection on the utility before programming the device.

The programmer usually erases the existing content of the chip before programming the chip. Only EEPROM and FLASH memory chips are erasable by the programmer.

The major drawback of out-of-circuit programming is the high development time. Whenever the firmware is changed, the chip should be taken out of the development board for re-programming. This is tedious and prone to chip damages due to frequent insertion and removal.

The out-of-system programming technique is used for firmware integration for low end embedded products which runs without an operating system. Out-of-circuit programming is commonly used for development of low volume products and Proof of Concept (PoC) product Development.

In System Programming (ISP):

With ISP, programming is done 'within the system', meaning the firmware is embedded into the target device without removing it from the target board. It is the most flexible and easy way of firmware embedding. The only pre-requisite is that the target device must have an ISP support. Apart from the target board, PC, ISP cable and ISP utility, no other additional hardware is required for ISP.

The target board can be interfaced to the utility program running on PC through Serial Port/ Parallel Port/ USB. The communication between the target device and ISP will be in a serial format. The serial protocols used for ISP may be '*Joint Test Act Group (JTAG)*' or '*Serial Peripheral Interface (SPI)*' or any other proprietary protocol.

In System Programming with SPI Protocol: Devices with SPI (Serial Peripheral Interface) ISP (In System Programming) support contains a built-in SPI interface and the on-chip EEPROM or FLASH memory. The primary I/O lines involved in SPI-In System Programming are listed below:

MOSI – Master Out Slave In

MISO – Master In Slave Out

SCK – System Clock

RST – Reset of Target Device

GND – Ground of Target Device

PC acts as the master and target device acts as the slave in ISP. The program data is sent to the MOSI pin of target device and the device acknowledgement is originated from the MISO pin of the device. SCK pin

acts as the clock for data transfer. A utility program can be developed on the PC side to generate the above signal lines.

Standard SPI-ISP utilities are feely available on the internet and, there is no need for going for writing own program. For ISP operations, the target device needs to be powered up in a pre-defined sequence. The power up sequence for In System Programming for Atmel's AT89S series microcontroller family is listed below:

1. Apply supply voltage between VCC and GND pins of target chip
2. Set RST pin to "HIGH" state
3. If a crystal is not connected across pins XTAL 1 and XTAL2, apply a 3 MHz to 24 MHz clock to XTALl pin and wait for at least 10 milliseconds
4. Enable serial programming by sending the Programming Enable serial instruction to pin MOSI/ Pl.5. The frequency of the shift clock supplied at pin SCK/ P1.7 needs to be less than the CPU clock at XTALl divided by 40
5. The Code or Data array is programmed one byte at a time by supplying the address and data together with the appropriate Write instruction. The selected memory location is first erased before the new data is written. The write cycle is self-timed and typically takes less than 2.5 ms at 5V
6. Any memory location can be verified by using the Read instruction, which returns the content at the selected address at serial output MISO/ Pl.6
7. After successfully programming the device, set RST pin low or turn off the chip power supply and turn it ON to commence the normal operation.

The key player behind ISP is a factory programmed memory (ROM) called '*Boot ROM*'. The Boot ROM normally resides at the top end of code memory space and it varies in the order of a few Kilo Bytes (For a controller with 64K code memory space and lK Boot ROM, the Boot ROM resides at memory location FC00H to FFFFH). It contains a set of Low-level Instruction APIs and these APIs allow the processor/ controller to perform the FLASH memory programming, erasing and Reading operations. The contents of the Boot ROM are provided by the chip manufacturer and the same is masked into every device.

In Application Programming (IAP):

In Application Programming is a technique used by the firmware running on the target device for modifying a selected portion of the code memory. It is not a technique for first time embedding of user written firmware. It modifies the program code memory under the control of the embedded application.

Updating calibration data, look-up tables, etc., which are stored in code memory, are typical examples of IAP.

Use of Factory Programmed Chip:

It is possible to embed the firmware into the target processor/ controller memory at the time of chip fabrication itself. Such chips are known as *'Factory Programmed Chips*'. Once the firmware design is over and the firmware achieved operational stability, the firmware files can be sent to the chip fabricator to embed it into the code memory.

Factory programmed chips are convenient for mass production applications and it greatly reduces the product development time. It is not recommended to use factory programmed chips for development purpose where the firmware undergoes frequent changes. Factory programmed ICs are bit expensive.

Firmware Loading for Operating System Based Devices:

The OS based embedded systems are programmed using the In System Programming (ISP) technique. OS based embedded systems contain a special piece of code called '*Boot loader*' program which takes control of the OS and application firmware embedding and copying of the OS image to the RAM of the system for execution.

The 'Boot 1oader' for such embedded systems comes as pre-loaded or it can be loaded to the memory using the various interface supported like JTAG. The boot loader contains necessary driver initialization implementation for initializing the supported interfaces like UART/ I2C, TCP/ IP, etc. Boot loader implements menu options for selecting the source for OS image to load (Typical menu item examples are Load from FLASH ROM, Load from Network, Load through UART, etc).

Once a communication link is established between the host and target machine, the OS image can be directly downloaded to the FLASH memory of the target device.

BOARD BRING UP:

Once the firmware is embedded into the target board using one of the programming techniques, then power up the board. You may be expecting the device functioning exactly in a way as you designed. But in real scenario it need not be and if the board functions well in the first attempt itself you are very lucky.

Sometimes the first power up may end up in a messy explosion leaving the smell of burned components behind. It may happen due to various reasons, like Proper care was not taken in applying the power and power applied in reverse polarity (+ve of supply connected to –ve of the target board and vice versa), components were not placed in the correct polarity order (E.g. a capacitor on the target board is connected to the board with +ve terminal to –ve of the board and vice versa), etc ... etc ...

The prototype/ evaluation/ production version must pass through a varied set of tests to verify that embedded hardware and firmware functions as expected. *Bring up* process includes –

- basic hardware spot checks/ validations to make sure that the individual components and busses/ interconnects are operational – which involves checking power, clocks, and basic functional connectivity;
- basic firmware verification to make sure that the processor is fetching the code and the firmware execution is happening in the expected manner;
- running advanced validations such as memory validations, signal integrity validation, etc.

THE EMBEDDED SYSTEM DEVELOPMENT ENVIRONMENT

The embedded system development environment consists of –

- Development Computer (PC) or Host – acts as the heart of the development environment
- Integrated Development Environment (IDE) Tool – for embedded firmware development and debugging
- Electronic Design Automation (IDA) Tool – for embedded hardware design
- An emulator hardware – for debugging the target board
- Signal sources (like CRO, Multimeter, Logic Analyzer, etc.)
- Target hardware.

THE INTEGRATED DEVELOPMENT ENVIRONMENT (IDE):

In embedded system development context, *Integrated Development Environment (IDE)* stands for an integrated environment for developing and debugging the target processor specific embedded firmware. IDE is a software package which bundles –

- a "Text Editor (Source Code Editor)",
- "Cross-complier (for cross platform development and complier for same platform development)",
- "Linker", and
- a "Debugger".

Some IDEs may provide –

- interface to target board emulators,
- target processor's/ controller's Flash memory programmer, etc.

IDE may be command line based or GUI based.

NOTE: The Keil µVision IDE & An Overview of IDEs – lest as an exercise/ self study topic.

DISASSEMBLER/ DECOMPLIER:

Disassembler is a utility program which converts machine codes into target processor specific Assembly codes/ instructions. The process of converting machine codes into Assembly code is known as *'Disassembling*'. In operation, disasseri1bling is complementary to assembling/ cross-assembling.

Decompiler is the utility program for translating machine codes into corresponding high level language instructions. Decompiler performs the reverse operation of compiler/ cross-compiler.

The *disassemblers/ decompilers* for different family of processors/ controllers are different. Disassemblers/ Decompilers are deployed in reverse engineering. *Reverse engineering* is the process of revealing the technology behind the working of a product. Reverse engineering in Embedded Product development is employed to find out the secret behind the working of popular proprietary products. Disassemblers /decompilers help the reverse engineering process by translating the embedded firmware into Assembly/ high level language instructions.

Disassemblers/ Decompilers are powerful tools for analyzing the presence of malicious codes (virus information) in an executable image. Disassemblers/ Decompilers are available as either freeware tools readily available for free download from internet or as commercial tools.

It is not possible for a disassembler/ decompiler to generate an exact replica of the original assembly code/ high level source code in terms of the symbolic constants and comments used. However disassemblers/ decompilers generate a source code which is somewhat matching to the original source code from which the binary code is generated.

SIMULATORS, EMULATORS AND DEBUGGING:

Simulators and *emulators* are two important tools used in embedded system development.

- *Simulator* is a software tool use for simulating the various conditions for checking the functionality of the application firmware. The Integrated Development Environment (IDE) itself will be providing simulator support and they help in debugging the firmware for checking its required functionality. In certain scenarios, simulator refers to a soft model (GUI model) of the embedded product.
    - For example, if the product under development is a handheld device, to test the functionalities of the various menu and user interfaces, a soft form model of the product with all UI as given in the end product can be developed in software. Soft phone is an example for such a simulator.
- *Emulator* is hardware device which emulates the functionalities of the target device and allows real time debugging of the embedded firmware in a hardware environment.

Simulators:

Simulators simulate the target hardware and the firmware execution can be inspected using simulators.

The features of simulator based debugging are listed below.

1. Purely software based
2. Doesn't require a real target system
3. Very primitive (Lack of featured I/O support. Everything is a simulated one)

4. Lack of Real-time behavior.

Advantages of Simulator Based Debugging: Simulator based debugging techniques are simple and straightforward .The major advantages of simulator based firmware debugging techniques are explained below.

- *No Need for Original Target Board:* Simulator based debugging technique is purely software oriented. IDE's software support simulates the CPU of the target board. User only needs to know about the memory map of various devices within the target board and the firmware should be written on the basis of it. Since the real hardware is not required, firmware development can start well in advance immediately after the device interface and memory maps are finalized. This saves development time.
- *Simulate I/O Peripherals:* Simulator provides the option to simulate various I/O peripherals. Using simulator's I/O support you can edit the values for I/O registers and can be used as the input/ output value in the firmware execution. Hence it eliminates the need for connecting I/O devices for debugging the firmware.
- *Simulates Abnormal Conditions:* With simulator's simulation support you can input any desired value for any parameter during debugging the firmware and can observe the control flow of firmware. It really helps the developer in simulating abnormal operational environment for firmware and helps the firmware developer to study the behavior of the firmware under abnormal input conditions.

Limitations of Simulator Based Debugging: Though simulation based firmware debugging technique is very helpful in embedded applications, they possess certain limitations and we cannot fully rely on the simulator-based firmware debugging. Some of the limitations of simulator-based debugging are explained below:

- *Deviation from Real Behavior:* Simulation-based firmware debugging is always carried out in a development environment where the developer may not be able to debug the firmware under all possible combinations of input. Under certain operating conditions, we may get some particular result and it need not be the same when the firmware runs in a production environment.
- *Lack of Real Timeliness:* The major limitation of simulator based debugging is that it is not real-time in behavior. The debugging is developer driven and it is no way capable of creating a real time behavior. Moreover in a real application the I/O condition may be varying or unpredictable. Simulation goes for simulating those conditions for known values.

Emulators and Debuggers:

Debugging in embedded application is the process of diagnosing the firmware execution, monitoring the target processor's registers and memory, while the firmware is running and checking the signals from various buses of the embedded hardware. Debugging process in embedded application is broadly classified into two, namely; hardware debugging and firmware debugging.

- *Hardware debugging* deals with the monitoring of various bus signals and checking the status lines of the target hardware.
- *Firmware debugging* deals with examining the firmware execution, execution flow, changes to various CPU registers and status registers on execution of the firmware to ensure that the firmware is running as per the design.

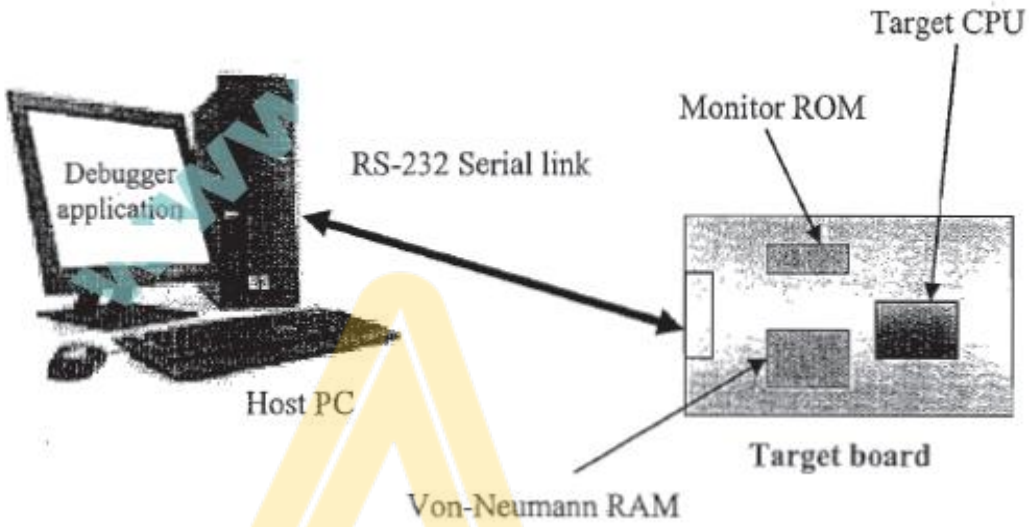
Firmware debugging is performed to figure out the bug or the error in the firmware which creates the unexpected behavior. The following section describes the improvements over firmware debugging starting from the most primitive type of debugging to the most sophisticated On Chip Debugging (OCD):

- ***Incremental EEPROM Burning Technique:*** This is the most primitive type of firmware debugging technique where the code is separated into different functional code units. Instead of burning the entire code into the EEPROM chip at once, the code is burned in incremental order, where the code corresponding to all functionalities are separately coded, cross-compiled and burned into the chip one by one.

- ***Inline Breakpoint Based Firmware Debugging:*** Inline breakpoint based debugging is another primitive method of firmware debugging. Within the firmware where you want to ensure that firmware execution is reaching up to a specified point, insert an inline debug code immediately after the point. The debug code is a *printf()* function which prints a string given as per the firmware. You can insert debug codes (*printf()*) commands at each point where you want to ensure the firmware execution is covering that point. Cross-compile the source code with the debug codes embedded within it. Burn the corresponding hex file into the EEPROM.

- ***Monitor Program Based Firmware Debugging:*** Monitor program based firmware debugging is the first adopted invasive method for firmware debugging (see the following Figure). In this approach a monitor program which acts as a supervisor is developed. The monitor program controls the downloading of user code into the code memory, inspects and modifies register/ memory locations; allows single stepping of source code, etc. The monitor program implements the debug functions as per a pre-defined command set from the debug application interface. The

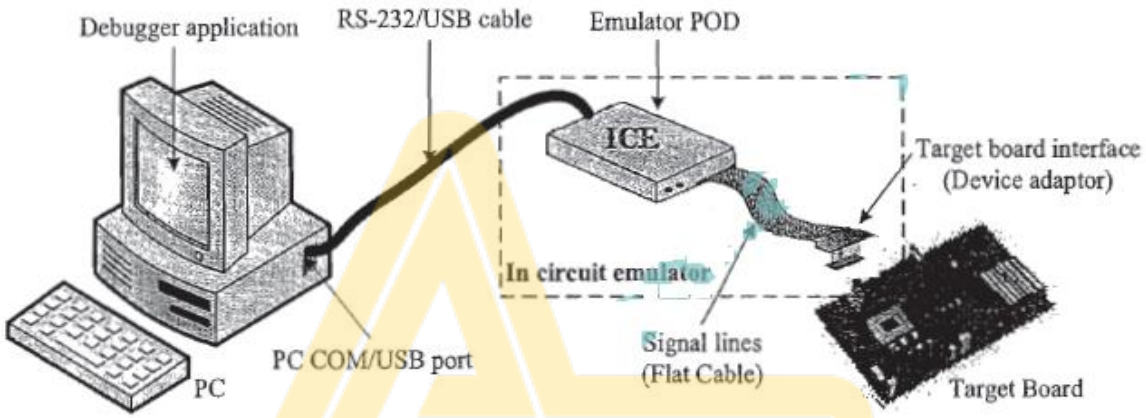
monitor program always listens to the serial port of the target device and according to the command received from the serial interface it performs command specific actions like firmware downloading, memory inspection/ modification, firmware single stepping and sends the debug information (various register and memory contents) back to the main debug program running on the development PC, etc.



- The first step in any monitor program development is determining a set of commands for performing various operations like firmware downloading, memory/ register inspection/ modification, single stepping, etc. The entire code stuff handling the command reception and corresponding action implementation is known as the "*monitor program*". The most common type of interface used between target board and debug application is RS-232C Serial interface.
- The monitor program contains the following set of minimal features:
    1. Command set interface to establish communication with the debugging application
    2. Firmware download option to code memory
    3. Examine and modify processor registers and working memory (RAM)
    4. Single step program execution
    5. Set breakpoints in firmware execution
    6. Send debug information to debug application running on host machine.

- ***In Circuit Emulator (ICE) Based Firmware Debugging:*** The terms 'Simulator' and 'Emulator' are little bit confusing and sounds similar. Though their basic functionality is the same-"Debug the target firmware", the way in which they achieve this functionality is totally different. The simulator 'simulates' the target board CPU and the emulator 'emulates' the target board CPU.

- '*Simulator*' is a software application that precisely duplicates (mimics) the target CPU and simulates the various features and instructions supported by the target CPU.
- '*Emulator*' is a self-contained hardware device which emulates the target CPU. The emulator hardware contains necessary emulation logic and it is hooked to the debugging application running on the development PC on one end and connects to the target board through some interface on the other end.
- The Emulator POD (see the following Figure) forms the heart of any emulator system and it contains the following functional units.



- *Emulation Device:* is a replica of the target CPU which receives various signals from the target board through a device adaptor connected to the target board and performs the execution of firmware under the control of debug commands from the debug application.
- *Emulation Memory:* is the Random Access Memory (RAM) incorporated in the Emulator device. It acts as a replacement to the target board's EEPROM where the code is supposed to be downloaded after each firmware modification. Hence the original EEPROM memory is emulated by the RAM of emulator. This is known as 'ROM Emulation'. ROM emulation eliminates the hassles of ROM burning and it offers the benefit of infinite number of reprogramming.
- *Emulator Control Logic:* is the logic circuits used for implementing complex hardware breakpoints, trace buffer trigger detection, trace buffer control, etc. Emulator control logic circuits are also used for implementing logic analyzer functions in advanced emulator devices. The 'Emulator POD' is connected to the target board through a 'Device adaptor' and signal cable.
- *Device Adaptors:* act as an interface between the target board and emulator POD. Device adaptors are normally pin-to-pin compatible sockets which can be inserted/ plugged into the target board for routing the various signals from pins assigned for the target

processor. The device adaptor is usually connected to the emulator POD using ribbon cables.

- ***On Chip Firmware Debugging (OCD):*** Advances in semiconductor technology has brought out new dimensions to target firmware debugging. Today almost all processors/controllers in-corporate built in debug modules called On Chip Debug (OCD) support. Though OCD adds silicon complexity and cost factor, from a developer perspective it is a very good feature supporting fast and efficient firmware debugging. The On Chip Debug facilities integrated to the processor/ controller are chip vendor dependent and most of them are proprietary technologies like Background Debug Mode (BDM), OnCE, etc.

TARGET HARDWARE DEBUGGING:

Even though the firmware is bug free and everything is intact in the board, your embedded product need not function as per the expected behavior in the first attempt for various hardware related reasons like dry soldering of components, missing connections in the PCB due to any un-noticed errors in the PCB layout design, misplaced components, signal corruption due to noise, etc. The only way to sort out these issues and figure out the real problem creator is debugging the target board.

Hardware debugging is not similar to firmware debugging. Hardware debugging involves the monitoring of various signals of the target board (address/ data lines, port pins, etc.), checking the inter connection among various components, circuit continuity checking, etc.

The various hardware debugging tools used in Embedded Product Development are explained below.

Magnifying Glass (Lens):

You might have noticed watch repairer wearing a small magnifying glass while engaged -in repairing a watch. They use the magnifying glass to view the minute components inside the watch in an enlarged manner so that they can easily work with them.

Similar to a watch repairer, *magnifying glass* is the primary hardware debugging tool for an embedded hardware debugging professional.

A *magnifying glass* is a powerful visual inspection tool. With a magnifying glass (lens), the surface of the target board can be examined thoroughly for dry soldering of components, missing components, improper placement of components, improper soldering, track (PCB connection) damage, short of tracks, etc. Nowadays high quality magnifying stations are available for visual inspection.

Multimeter:

A *multimeter* is used for measuring various electrical quantities like voltage (Both AC and DC), current (DC as well as AC), resistance, capacitance, continuity checking, transistor checking, cathode and anode identification of diode, etc.

Any multimeter will work over a specific range for each measurement. A multimeter is the most valuable tool in the tool kit of an embedded hardware developer. It is the primary debugging tool for physical contact based hardware debugging and almost all developers start debugging the hardware with it.

Digital CRO:

Cathode Ray Oscilloscope (CRO) is a little more sophisticated tool compared to a multimeter. CRO is used for waveform capturing and analysis, measurement of signal strength, etc. By connecting the point under observation on the target board to the Channels of the Oscilloscope, the waveforms can be captured and analyzed for expected behavior.

CRO is a very good tool in analyzing interference noise in the power supply line and other signal lines. Monitoring the crystal oscillator signal from the target board is a typical example of the usage of CRO for waveform capturing and analysis in target board debugging.

CROs are available in both analog and digital versions. Though Digital CROs are costly, feature-wise they are best suited for target board debugging applications. Digital CROs are available for high frequency support and they also incorporate modem techniques for recording waveform over a period of time, capturing waves on the basis of a configurable event (trigger) from the target board.

Various measurements like phase, amplitude, etc. are also possible with CROs. Tektronix, Agilent, Philips, etc. are the manufacturers of high precision good quality digital CROs.

Logic Analyzer:

A logic analyzer is the big brother of digital CRO. *Logic analyzer* is used for capturing digital data (logic 1 and 0) from a digital circuitry whereas CRO is employed in capturing all kinds of waves including logic signals. Another major limitation of CRO is that the total number of logic signals/ waveforms that can be captured with a CRO is limited to the number of channels.

A logic analyzer contains special connectors and clips which can be attached to the target board for capturing digital data. In target board debugging applications, a logic analyzer captures the states of various port pins, address bus and data bus of the target processor/ controller, etc.

Logic analyzers give an exact reflect on of what happens when a particular line of firmware is running. This is achieved by capturing the address line logic and data line logic of target hardware. Most modem logic analyzers contain provisions for storing captured data, selecting a desired region of the captured waveform, zooming selected region of the captured waveform, etc. Tektronix, Agilent, etc. are the giants in the logic analyzer market.

Function Generator:

Function generator is not a debugging tool. It is a input signal simulator tool. A *function generator* is capable of producing various periodic waveforms like sine wave, square wave, saw-tooth wave, etc. with different frequencies and amplitude.

Sometimes the target board may require some kind of periodic waveform with a particular frequency as input to some part of the board. Thus, in a debugging environment, the function generator serves the purpose of generating and supplying required signals.

BOUNDARY SCAN:

As the complexity of the hardware increase, the number of chips present in the board and the interconnection among them may also increase. The device packages used in the PCB become miniature to reduce the total board space occupied by them and multiple layers may be required to route the interconnections among the chips. With miniature device packages and multiple layers for the PCB it will be very difficult to debug the hardware using magnifying glass, multimeter, etc. to check the interconnection among the various chips.

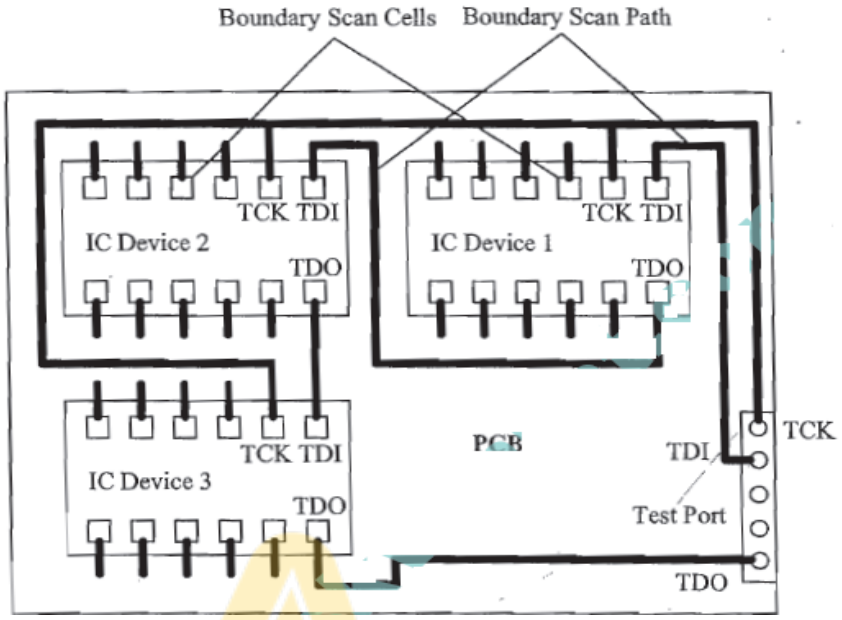
Boundary scan is a technique used for testing the interconnection among the various chips, which support JTAG interface, present in the board. Chips which support boundary scan associate a boundary scan cell with each pin of the device.

A JTAG port contains the five signal lines, namely, TDI, TDO, TCK, TRST and TMS form the Test Access Port (TAP) for a JTAG supported chip. Each device will have its own TAP. The PCB also contains a TAP for connecting the JTAG signal lines to the external world.

A boundary scan path is formed inside the board by interconnecting the devices through JTAG signal lines. The TDI pin of the TAP of the PCB is connected to the TDI pin of the first device.

The TDO pin of the first device is connected to the TDI pin of the second device. In this way all devices are interconnected and the TDO pin of the last JTAG device is connected to the TDO pin of the TAP of the PCB. The clock line TCK and the Test Mode Select (TMS) line of the devices are connected to the clock line and Test mode select line of the Test Access Port of the PCB respectively. This forms a boundary scan path.

The following Figure illustrates the same.



By: DR. MAHESH PRASANNA K.

MR. SANDESHA KARANTH P. K.

DEPT. OF CSE, VCET.

****

****