Module 5

RTOS and IDE for Embedded System Design

5.1 What is an Operating System?

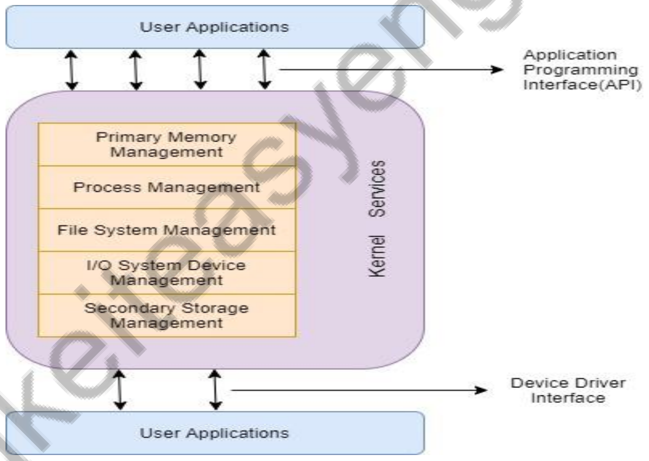
The operating system acts as a bridge between the user applications/tasks and the underlying system resources through a set of system functionalities and services.

The OS manages the system resources and makes them available to the user applications/tasks on a need basis. A normal computing system is a collection of different I/O subsystems, working, and storage memory. The primary functions of an operating system is

• Make the system convenient to use

• Organize and manage the system resources efficiently and correctly

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


User Applications
Application Programming Interface(API)
Primary Memory Management
Process Management
File System Management
I/O System Device Management
Secondary Storage Management
Kernel Services
Device Driver Interface
User Applications

❖ The Kernel

The kernel is the core of the operating system and 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 for handling the following.

- **Primary Memory Management:** The term primary memory refers to the volatile memory (RAM) where processes are loaded and variables and shared data associated with each process are stored. The Memory Management Unit (MMU) of the kernel is responsible for

• Keeping track of which part of the memory area is currently used by which process

• Allocating and De-allocating memory space on a need basis (Dynamic memory allocation).

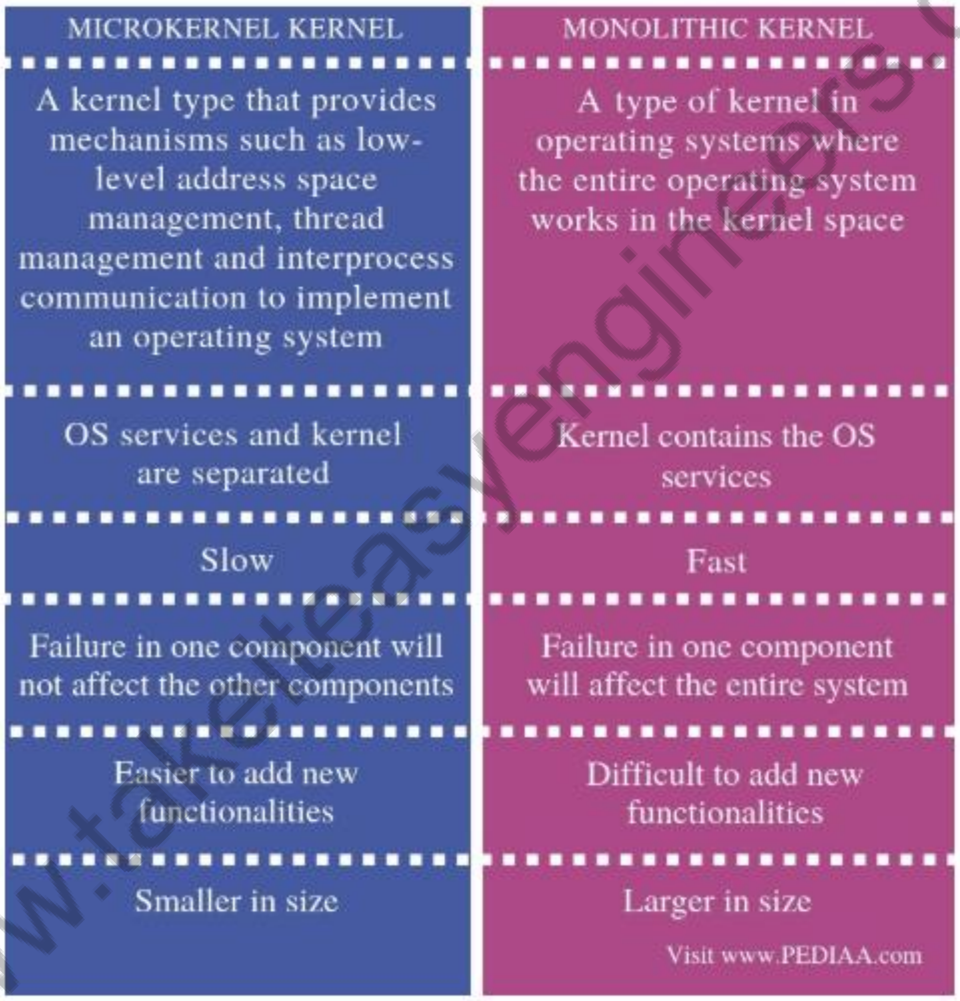
- **Process Management:** Process management deals with managing the processes/tasks. Process management includes setting up the memory space for the process, loading the process's code into the memory space, allocating system resources, scheduling and managing the execution of the process, setting up and managing the Process Control Block (PCB), Inter Process Communication and synchronization, process termination/deletion, etc.
- **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. Each of these files differ in the kind of information they hold and the way in which the information is stored. The file operation is a useful service provided by the OS. The 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 (e.g. Hard disk storage)
  - Providing automatic allocation of file space based on the amount of free space available
  - Providing a flexible naming convention 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, the direct accessing of I/O devices are not allowed and the access to them are provided through a set of Application Programming Interfaces (APIs) exposed by the kernel. The kernel maintains a list of all the I/O devices of the system.
- **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 since the main

memory is volatile. In most of the systems, the secondary storage is kept in disks (Hard Disk). The secondary storage management service of kernel deals with

1. Disk storage allocation
2. Disk scheduling (Time interval at which the disk is activated to backup data)
3. Free Disk space management

Monolithic and Micro-Kernel:



MICROKERNEL KERNEL	MONOLITHIC KERNEL
A kernel type that provides mechanisms such as low-level address space management, thread management and interprocess communication to implement an operating system	A type of kernel in operating systems where the entire operating system works in the kernel space
OS services and kernel are separated	Kernel contains the OS services
Slow	Fast
Failure in one component will not affect the other components	Failure in one component will affect the entire system
Easier to add new functionalities	Difficult to add new functionalities
Smaller in size	Larger in size

Visit www.PEDIAA.com

5.2 TYPES OF OPERATING SYSTEMS:

1. **General Purpose Operating System (GPOS)**

   The operating systems, which are deployed in general computing systems, are referred as General Purpose Operating Systems (GPOS). The kernel of such an OS is more generalized and it contains all kinds of services required for executing generic applications. General-purpose operating systems are often quite non-deterministic in behavior. Their services can inject random delays into application software and may cause slow responsiveness of an application at unexpected times.

   Personal Computer/ Desktop system is a typical example for a system where GPOSs are deployed. Windows XP/MS-DOS etc. are examples for General Purpose Operating Systems.

2. **Real-Time Operating System (RTOS)**

   'Real-Time' implies deterministic timing behavior. Deterministic timing behavior in RTQS context means the OS services consumes only known and expected amounts of time regardless the number of services. A Real-Time Operating System or RTOS implements policies and rules concerning time-critical allocation of a system's resources. The RTOS decides which applications should run in which order and how much time needs to be allocated for each application.

   Windows CE, QNX, VxWorks, MicroC/OS-II, etc. are examples of Real-Time Operating Systems (RTOS).

- **The Real Time Kernel**

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

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

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

   Task ID: Task Identification Number

   Task State: The current state of the task (e.g. State = 'Ready' for a task which is ready to execute)

   Task Type: Task type Indicates what is the type for this task. The task can be a hard real time or soft real time or background task.

   Task Priority: Task priority (e.g. Task priority = 1 for task with priority - 1)

   Task Context Pointer: Context pointer. Pointer for context saving

   Task Memory Pointers: Pointers to the code memory, data memory and stack memory for the task.

   Task System Resource Pointers: Pointers to system resources (semaphores, mutex, etc.) used by the task

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

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

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

5) **Memory Management:** Compared to the General Purpose Operating Systems, the memory management function of an RTOS kernel is slightly different. In general, the memory allocation time increases depending on the size of the block of memory needs to be allocated and the state of the allocated memory block (initialized memory block consumes more allocation time than un-initialized memory block). Since predictable timing and deterministic behavior are the primary focus of an RTOS, RTOS achieves this by compromising the effectiveness of memory allocation. RTOS makes use of '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.

6) **Interrupt Handling:** Deals with the handling of various types of interrupts. Interrupts provide Real- Time behavior to systems. Interrupts inform the processor that an external device or an associated task requires immediate attention of the CPU.

Interrupts can be either Synchronous or Asynchronous. Interrupts which occurs in sync with the currently executing task is known as Synchronous interrupts. Usually the software interrupts fall under the Synchronous Interrupt category. Divide by zero, memory segmentation error, etc. are examples of synchronous interrupts. For synchronous interrupts, the interrupt handler runs in the same context of the interrupting task.

Asynchronous interrupts are interrupts, which occurs at any point of execution of any task, and are not in sync with the currently executing task. The interrupts generated by external devices (by asserting the interrupt line of the processor/controller to which the interrupt line of the device is connected) connected to the processor/controller, timer overflow interrupts, and serial data reception / transmission interrupts, etc. are examples for asynchronous interrupts.

7) **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.

Hard Real-Time: Real-Time Operating Systems that strictly adhere 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.

Soft Real-Time: Real-Time Operating System 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 for a Soft Real-time system if the frequency of deadline missing is within the compliance limit of the Quality of Service (QoS).

5.3 TASKS, PROCESS AND THREADS

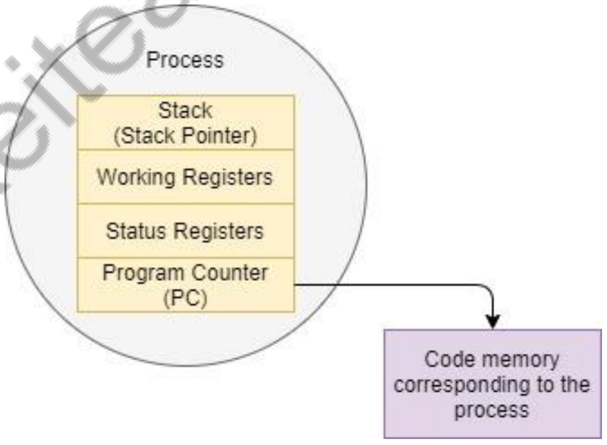
In the operating system context, a task is defined as the program in execution and the related information maintained by the operating system for the program. Task is also known as 'Job' in the operating system context. A program or part of it in execution is also called a 'Process'. The terms 'Task', 'Job' and 'Process' refer to the same entity in the operating system context and most often they are used interchangeably.

- **Process**

  A 'Process' is a program, or part of it, in execution. Process is also known as an instance of a program in execution. Multiple instances of the same program can execute simultaneously. 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. A process is sequential in execution.

  The Structure of a process:

  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 Fig.



  From a memory perspective, the memory occupied by the process is segregated into three regions, namely, Stack memory, Data memory and Code memory.

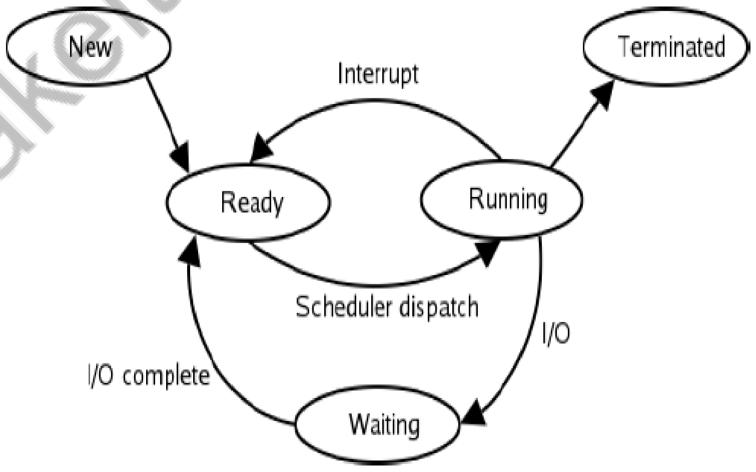
The 'Stack' memory holds all temporary data such as variables local to the process. Data memory holds all global data for the process. The code memory contains the program code (instructions) corresponding to the process. On loading a process into the main memory, a specific area of memory is allocated for the process. The stack memory usually starts (OS Kernel implementation dependent) at highest memory address from the memory area allocated for the process. Say for example, the memory map of the memory area allocated for the process is 2048 to 2100, the stack memory starts at address 2100 and grows downwards to accommodate the variables local to the process.

Process States and State Transition:

As a process executes, it changes state. The state of a process is defined by the correct activity of that process. Each process may be in one of the following states.

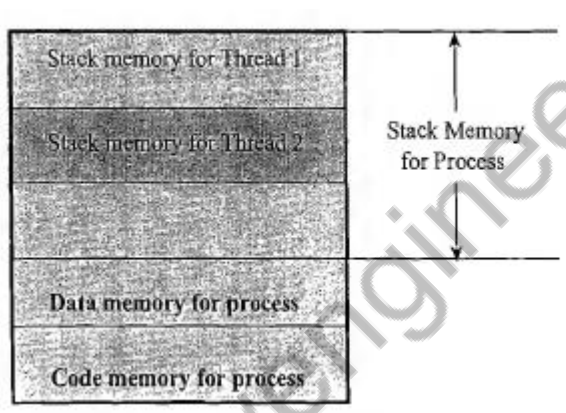
- **New:** The process is being created.
- **Ready:** The process is waiting to be assigned to a processor.
- **Running:** Instructions are being executed.
- **Waiting:** The process is waiting for some event to occur.
- **Terminated:** The process has finished execution.

Many processes may be in ready and waiting state at the same time. But only one process can be running on any processor at any instant.



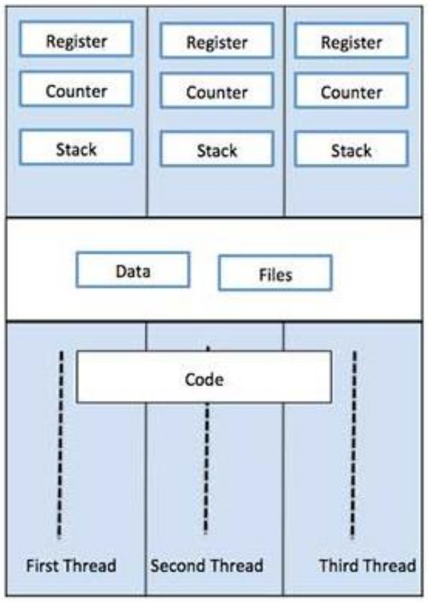
- **Threads**

  A thread is the primitive that can execute code. A thread is a single sequential flow of control within a process. 'Thread' is also known as lightweight process. A process can have many threads of execution. Different threads, which are part of a process, share the same address space; meaning they share the data memory, code memory and heap memory area. Threads maintain their own thread status (CPU register values), Program Counter (PC) and stack. The memory model for a process and its associated threads are given in Fig.



- **Multithreading**

  A process/task in embedded application may be a complex or lengthy one and it may contain various sub operations like getting input from I/O devices connected to the processor, performing some internal calculations/operations, updating some I/O devices etc. If all the sub functions of a task are executed in sequence, the CPU utilization may not be efficient. For example, if the process is waiting for a user input, the CPU enters the wait state for the event, and the process execution also enters a wait state. Instead of this single sequential execution of the whole process, if the task/process is split into different threads carrying out the different sub functionalities of the process, the CPU can be effectively utilized and when the thread corresponding to the I/O operation enters the wait state, another threads which do not require the I/O event for their operation can be switched into execution. This leads to more speedy execution of the process and the efficient utilization of the processor time and resources. The multithreaded architecture of a process can be better visualized with the thread-process diagram shown in Fig.



If 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. 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.

  A. **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.

int pthread_create(pthread_t *new_thread_ID, const pthread_attr_t *attribute; void * (*start_function)(void *), void *arguments);

Creates a new thread for running the function start_ function. Here pthread_t is the handle to the newly created thread and pthread_attr_t is the data type for holding the thread attributes, 'start_function’ is the function the thread is going to execute and arguments is the arguments for 'start_function’ (It is a void * in the above example). On successful creation of a Pthread, pthread_create() associates the Thread Control Block (TCB) corresponding to the newly created thread to the variable of type pthread_t.

int pthread_join(pthread_t new_thread,void * *thread_status);

blocks the current thread and waits until the completion of the thread pointed by it.

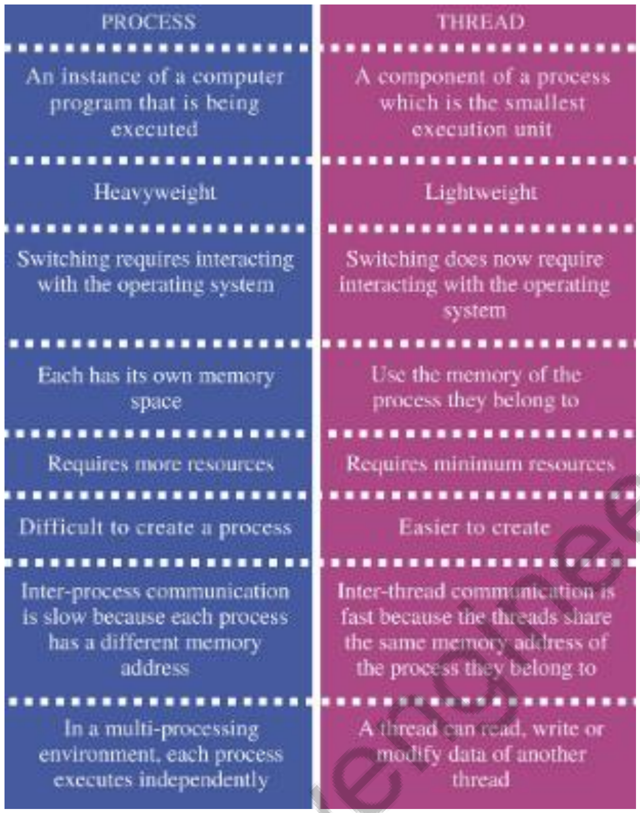
Example1:

Write a multithreaded application to print “Hello I’m in main thread” from the main thread and “Hello I’m in new thread” 5 times each, using the pthread_create() and pthread_Join() POSIX primitives.

```
//Assumes the application is running op an OS where POSIX library is available
#include <pthread.h>
#include <stblib.h>
#include <stdio.h>
//New thread function for printing "Hello I'm in new thread"
void *new_thread( void *thread_args )
{
int i, j;
for ( j= 0; j < 5; j++ )
{
printf("Hello I'm in new thread\n" );
//Wait for some time. Do nothing
//The fallowing line of code can be replaced with
//OS supported delay function like sleep(), delay() etc.
for ( i= 0; i < 10000; i++) ;
}
return NULL;
}
//Start of main thread
int main(void )
{
int i, j;
pthread_t tcb;
//Create the new thread for executing new_thread function
if (pthread_create(&tcb, NULL, new_thread, NULL ))
{
//New thread creation failed
printf ("Error in creating new thread\n" ) ;
return -1;
}
for ( j= 0; j < 5; j++)
{
printf("Hello I'm in main thread\n" ) ;
//Wait for some time. Do nothing
//The following line of code can be replaced with
//OS supported delay function like sleep() , delay() etc..
for(i=0;i<10000;i++);
}
if(pthread_join(tcb, NULL))
{
//Thread join failed
printf ("Error in Thread join") ;
return -1;
}
Return 1;
}
```

B. **Win32 Threads:** Win32 threads are the threads supported by various flavours 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.

C. **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.



PROCESS	THREAD
An instance of a computer program that is being executed	A component of a process which is the smallest execution unit
Heavyweight	Lightweight
Switching requires interacting with the operating system	Switching does now require interacting with the operating system
Each has its own memory space	Use the memory of the process they belong to
Requires more resources	Requires minimum resources
Difficult to create a process	Easier to create
Inter-process communication is slow because each process has a different memory address	Inter-thread communication is fast because the threads share the same memory address of the process they belong to
In a multi-processing environment, each process executes independently	A thread can read, write or modify data of another thread

5.4 Multiprocessing and Multithreading

- In the operating system context multiprocessing describes the ability to execute multiple processes simultaneously. 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 uniprocessor system, it is not possible to execute multiple processes simultaneously. However, it is possible for a uniprocessor system to achieve some degree of pseudo parallelism in the execution of multiple processes by switching the execution among different processes.
- The ability of an operating system to hold multiple processes in memory and switch the processor (CPU) from executing one process to another process is known as multitasking. Multitasking creates the illusion of multiple tasks executing in parallel.

Types of Multitasking:

1. **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 hold 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.
2. **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.
3. **Non-preemptive Multitasking:** In 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 or system resource. The co-operative and non-preemptive multi¬ tasking differs in their behavior when they are in the 'Blocked/Wait' state.

5.5 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 running on an OS are classified as;

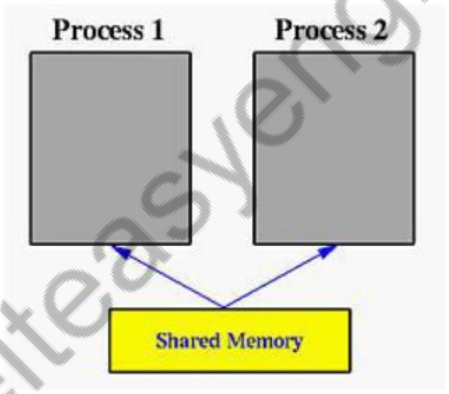
- **Co-operating Processes:** In the co-operating interaction model one process requires the inputs from other processes to complete its execution.
- **Competing Processes:** The competing processes do not share anything among themselves but they share the system resources. The competing processes compete for the system resources such as file, display device, etc.

Co-operating processes exchanges information and communicate through the following methods.

- **Co-operation through Sharing:** The co-operating process exchange data through some shared resources.
- **Co-operation through Communication:** No data is shared between the processes. But they communicate for synchronization.

Shared Memory

A ***shared memory*** is an extra piece of memory that is *attached* to some address spaces for their owners to use. As a result, all of these processes share the same memory segment and have access to it. Consequently, race conditions may occur if memory accesses are not handled properly. The following figure shows two processes and their address spaces. The yellow rectangle is a shared memory attached to both address spaces and both process 1 and process 2 can have access to this shared memory as if the shared memory is part of its own address space. In some sense, the original address space is "extended" by attaching this shared memory.



- **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 a pipe server and a process which connects to a pipe is known as pipe client. A pipe can be considered as a conduit 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 implementation of 'Pipes' is also OS dependent. Microsoft Windows Desktop Operating Systems support two types of 'Pipes' for Inter Process Communication. They are:

- **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.

- **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 (of course certain synchronization techniques should be applied to prevent inconsistent results). 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.

Message Passing:

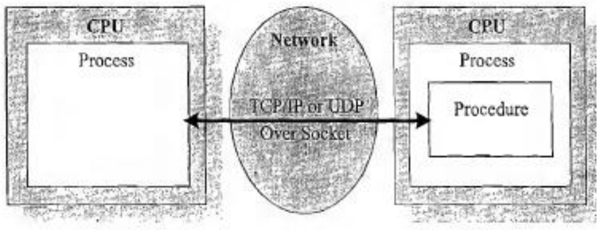
Message passing is an asynchronous information exchange mechanism used for Inter Process/Thread Communication. The major difference between shared memory and message passing technique is that, through shared memory lots of data can be shared whereas only limited amount of info/data is passed through message passing. Also 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 Queue:** Usually the 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.

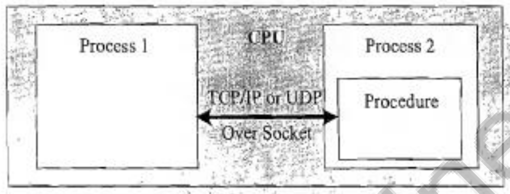
   Messages are sent and received through send and receive methods. The messages are exchanged through a message queue. The implementation of the message queue, send and receive methods are OS kernel dependent.
2. **Mailbox:** Mailbox is an alternate form of 'Message queues' and it is used in certain Real- Time Operating Systems for IPC. Mailbox technique for IPC in RTOS is usually used for one way messaging. The task/thread which wants to send a message to other tasks/threads creates a mailbox for post¬ ing the messages. The threads which are interested in receiving the messages posted to the mailbox by the mailbox creator thread can subscribe to the mailbox. The thread which creates the mailbox is known as 'mailbox server' and the threads which subscribe to the mailbox are known as 'mailbox clients'. The mailbox server posts messages to the mailbox and notifies it to the clients which are subscribed to the mailbox. The clients read the message from the mailbox on receiving the notification.
3. **Signaling:** Signaling is a primitive way of communication between processes/threads. Signals are used for asynchronous notifications where one process/thread fires a signal, indicating the occurrence of a scenario which the other processes/thread(s) is waiting. Signals are not queued and they do not carry any data.

Remote Procedure Call (RPC) and Sockets

Remote Procedure Call or RPC 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). RPC is mainly used for distributed applications like client-server applications. With RPC it is possible to communicate over a heterogeneous network (i.e. Network where Client and server applications are running on different Operating systems). 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.



Processes running on different CPUs which are networked



Processes running on same CPU

5.6 TASK SYNCHRONIZATION

In a multitasking environment, multiple processes run concurrently (in pseudo parallelism) and share the system resources. Apart from this, each process has its own boundary wall and they communicate with each other with different 1PC mechanisms including shared memory and variables.

Task Communication/Synchronization Issues

Racing: Racing or Race condition is the situation in which multiple processes compete (race) each other to access and manipulate shared data concurrently. In a Race condition the final value of the shared data depends on the process which acted on the data finally.

Deadlock: A race condition produces incorrect results whereas a deadlock condition creates a situation where none of the processes are able to make any progress in their execution, resulting in a set of deadlocked processes. A situation very similar to our traffic jam issues in a junction.

In its simplest form '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. To elaborate: 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. The result of the competition is 'deadlock'. None of the competing process will be able-to access the resources held by other processes since they are locked by the respective processes.

The different conditions favouring a deadlock situation are listed below.

- **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 hardware in an embedded device.
- **Hold and 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, PI ... Pn with P0 is waiting for a resource held by PI and PI is waiting for a resource held by PO, 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.
- **Deadlock Handling:** A smart OS may foresee the deadlock condition and will act proactively to avoid such a situation. Now if a deadlock occurred, how the OS responds to it? The reaction to deadlock condition by OS is non-uniform. 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 ofhappening 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.

- **Prevent Deadlocks:** 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 ifa 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.

- **Livelock:** The Livelock condition is similar to the deadlock condition except that a process in livelock condition changes its state with time. While in deadlock a process enters in wait state for a resource and continues in that state forever without making any progress in the execution, in a livelock condition a process always does something but is unable to make any progress in the execution completion.
- Starvation: In the multitasking context, starvation is the condition in which a process does not get the resources required to continue its execution for a long time. As time progresses the process starves on resource. Starvation may arise due to various conditions like byproduct of preventive measures of deadlock, scheduling policies favoring high priority tasks and tasks with shortest execution time, etc.

5.7 Task Synchronization Techniques:

Process/Task synchronization is essential for

1. Avoiding conflicts in resource access (racing, deadlock, starvation, livelock, etc.) in a multitasking environment.
2. Ensuring proper sequence of operation across processes. The producer consumer problem is a typical example for processes requiring proper sequence of operation.
3. Communicating between processes.

Mutual Exclusion through Sleep &Wakeup: The 'Busy waiting' mutual exclusion enforcement mechanism used by processes makes, the CPU always busy by checking the lock to see whether they can proceed. This results in the wastage of CPU time and leads to high power consumption. This is not affordable in embedded systems powered on battery, since it affects the battery backup time of the device. An alternative to 'busy waiting' is the 'Sleep & Wakeup' mechanism. When a process is not allowed to access the critical section, which is currently being locked by another process, the process undergoes 'Sleep' and enters the 'blocked" state. The process which is blocked on waiting for access to the critical section is awakened by the process which currently owns the critical section. The process which owns the critical section sends a wakeup message to the process, which is sleeping as a result of waiting for the access to the critical section, when the process leaves the critical section.

Semaphore: Semaphore is a sleep and wakeup based mutual exclusion implementation for shared resource access. Semaphore is a system resource and the 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 acquired 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 for the shared resource which needs exclusive access by a process.

Based on the implementation of the sharing limitation of the shared resource, semaphores are classified into two; namely 'Binary Semaphore' and 'Counting Semaphore'.
The binary semaphore provides 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

owned by a process. The implementation of binary semaphore is OS kernel dependent. Under certain OS kernel it is referred as mutex.

Unlike a binary semaphore, the 'Counting Semaphore' limits the access of resources by a fixed number of processes/threads. 'Counting Semaphore' maintains a count between zero and a value. It limits the usage of the resource to the maximum value of the count supported by it.

5.7 HOWTO CHOOSE AN RTOS

- 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: The task/process scheduling policies plays 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.

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 modularization 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.

- **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.

  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 form of e-mail, on-call services, etc. for bug fixes, critical patch updates and support for production issues, etc. should be analyzed thoroughly.

5.8 Integration 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.

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 contains 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.
- 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.

A. 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.

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 successful 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 take the device out of programmer

B. 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. Chips supporting ISP generates the necessary programming signals internally, using the chip’s supply voltage. 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 utility will be in a serial format. The serial protocols used for ISP may be 'Joint Test Action Group (JTAG)' or 'Serial Peripheral Interface (SPI)' or any other proprietary protocol.

In System Programming with SPI Protocol:

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 XTAL1 and XTAL2, apply a 3 MHz to 24 MHz clock to XTAL1 pin and wait for at least 10 milliseconds.
4. Enable serial programming by sending the Programming Enable serial instruction to pin MOSI/ PI.5. The frequency of the shift clock supplied at pin SCK/P1.7 needs to be less than the CPU clock at XTAL1 divided by 40.
5. The Code or Data array is programmed one byte at a time by supplying the address and data to¬ gether 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/P1.6.
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.

C. In Application Programming (IAP)

- In Application Programming (IAP) 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.
- The Boot ROM resident API instructions which perform various functions such as programming, erasing, and reading the Flash memory during ISP-mode, are made available to the end-user written firmware for IAP. Thus it is possible for an end-user application to perform operations on the Flash memory.

D. Use of Factory Programmed Chip

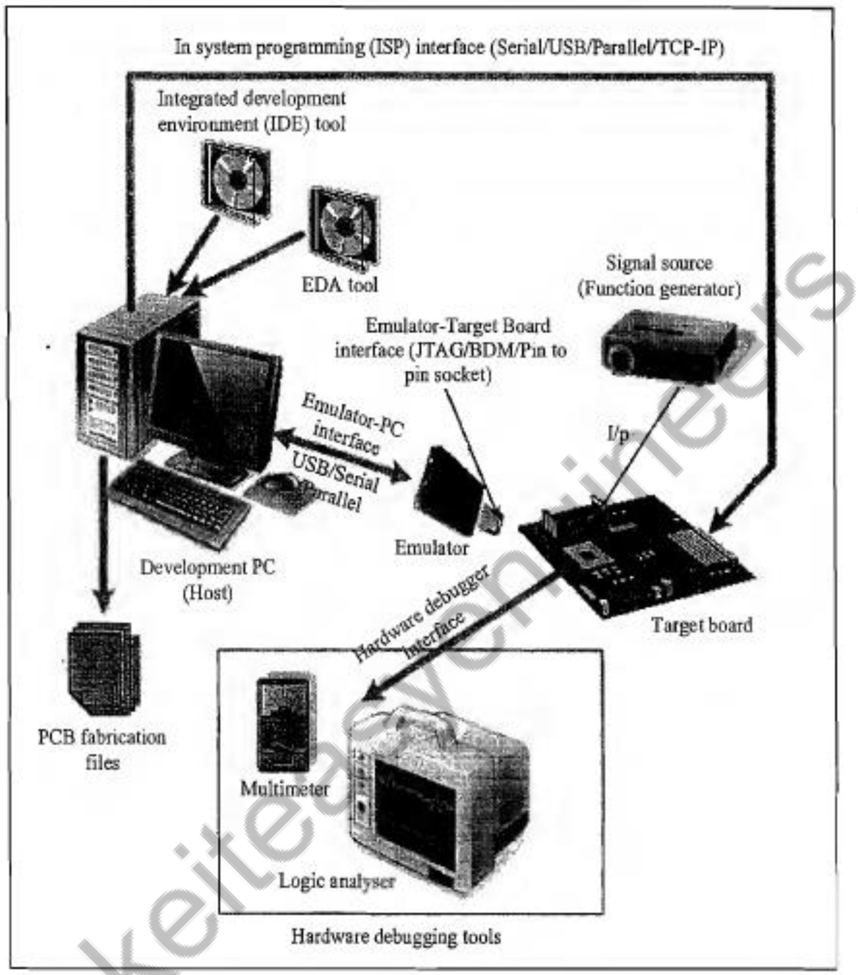
- It is possible to embed the firmware into the target processor/controller memory at the time of chip fabrication itself. Such chips are known as 'Factory programmed chips'.
- Once the firmware design is over and the firmware achieved operational stability, the firmware files can be sent to the chip fabric tor 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.

E. 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 loader' for such embedded systems comes as pre-loaded or it can be loaded to the memory using the various interface supported like JTAG.
- The bootloader contains necessary driver initialization implementation for initializing the supported interfaces like UART, TCP/IP etc.

5.9 The Embedded System Development Environment

A typical embedded system development environment is illustrated in Fig.


In system programming (ISP) interface (Serial/USB/Parallel/TCP-IP)
Integrated development environment (IDE) tool
EDA tool
Signal source (Function generator)
Emulator-Target Board interface (JTAG/BDM/Pin to pin socket)
Emulator-PC interface USB/Serial Parallel
I/p
Emulator
Development PC (Host)
Target board
Hardware debugger interface
PCB fabrication files
Multimeter
Logic analyser
Hardware debugging tools

As illustrated in the figure, the development environment consists of a Development Computer (PC) or Host, which acts as the heart of the development environment, Integrated Development Environment (IDE) Tool for embedded firmware development and debugging, Electronic Design Automation (EDA) Tool for Embedded Hardware design, An emulator hardware for debugging the target board, Signal sources (like Function generator) for simulating the inputs to the target board, Target hardware debugging tools (Digital CRO, Multimeter, Logic Analyser, etc.) and the target hardware. The Integrated Development Environment (IDE) and Electronic Design Automation (EDA) tools are selected based on the target hardware development requirement and they are supplied as Installable files in CDs by vendors. These tools need to be installed on the host PC used for development activities. These tools can be either freeware or licensed copy or evaluation versions.

5.10

a) DISASSEMBLER/DECOMPILER

- Disassembler is a utility program which converts machine codes into target processor specific Assembly codes/instractions.
- The process of converting machine codes into Assembly code is known as 'Disassembling'. In operation, disassembling 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.

b) SIMULATORS, EMULATORS AMD DEBUGGING

- Simulators and emulators are two important tools used in embedded system development.
- Simulator is a software tool used 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.

- 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.

I. 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:

- **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.
- **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.
- **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.

Disadvantages of Simulator Based Debugging:

- **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.

II. 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 behaviour.
- '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.

c) TARGET HARDWARE DEBUGGING

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):** 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. The magnifying station incorporates magnifying glasses attached to a stand with CFL tubes for providing proper illumination for 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.
  In embedded hardware debugging it is mainly used for checking the circuit continuity between different points on the board, measuring the supply voltage, checking the signal value, polarity, etc. Both analog and digital versions of a multimeter are available.

- **Digital CRO**: Cathode Ray Oscilloscope (CRO) is a little more sophisticated tool compared to a multimeter. CRO is a very good tool in analysing 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.

- **Logic Analyser:** Logic analyser 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 analyser contains special connectors and clips which can be attached to the target board for capturing digital data.

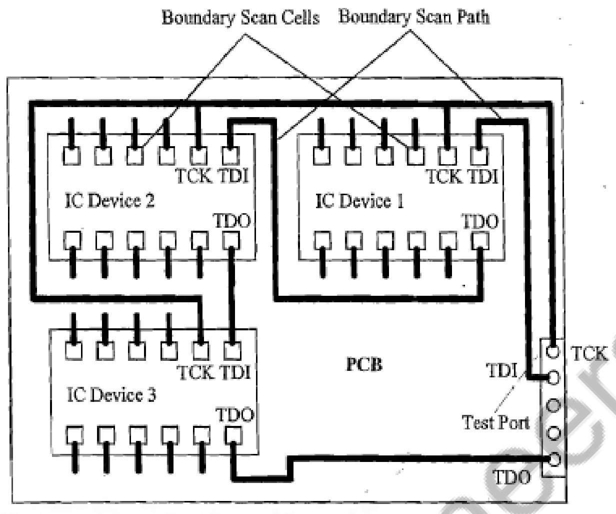
- **Function Generator:** Function generator is not a debugging tool. It is an input signal simulator tool. A function generator is capable of producing various periodic waveforms like sine wave, square wave, saw-tooth wave, etc. with different frequencies and amplitude.

d) 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.

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 which contains the five signal lines namely TDI, TDO, TCK, TRST and TMS form the Test Access Port (TAP) for a JTAG sup¬ ported 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. Figure illustrates the same.



The boundary scan cell associated with the input pins of an IC is known as 'input cells' and the boundary scan cells associated with the output pins of an IC are known as 'output cells'. The boundary scan cells can be used for capturing the input pin signal state and passing it to the internal circuitry, capturing the signals from the internal circuitry and passing it to the output pin, and shifting the data received from the Test Data In pin of the TAP.

The boundary scan cells can be operated in Normal, Capture, Update and Shift modes.

- In the Normal mode, the input of the boundary scan cell appears directly at its output.
- In the Capture mode, the boundary scan cell associated with each input pin of the chip captures the signal from the respective pins to the cell and the boundary scan cell associated with each output pin of the chip captures the signal from the internal circuitry.
- In the Update mode, the boundary scan cell associated with each input pin of the chip passes the already captured data to the internal circuitry and the boundary scan cell associated with each output pin of the chip passes the already captured data to the respective output pin.
- In the shift mode, data is shifted from TDI pin to TDO pin of the device through the boundary scan cells.