MODULE -5

EMBEDDED FIRMWARE DESIGN AND DEVELOPMENT

❖ Embedded Firmware Design & Development:

- The embedded firmware is responsible for controlling the various peripherals of the embedded hardware and generating response in accordance with the functional requirements of the product.
- The embedded firmware is the master brain of the embedded system
- The embedded firmware imparts intelligence to an Embedded system.
- The embedded firmware imparts intelligence to an Embedded system.
- It is a onetime process and it can happen at any stage.
- The product starts functioning properly once the intelligence imparted to the product by embedding the firmware in the hardware.
- The product will continue serving the assigned task till hardware breakdown occurs or a corruption in embedded firmware.
- In case of hardware breakdown, the damaged component may need to be replaced and for firmware corruptions the firmware should be re-loaded, to bring back the embedded product to the normal functioning.
- The embedded firmware is usually stored in a permanent memory (ROM) and it is non alterable by end users.
- The embedded firmware development process starts with the conversion of the firmware requirements into a program model using various modeling tools.
- The firmware design approaches for embedded product is purely dependent on the complexity of the functions to be performed and speed of operation required.
- There exist two basic approaches for the design and implementation of embedded firmware, namely;

  The Super loop based approach

The Embedded Operating System based approach

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

❖ Embedded firmware Design Approaches – The Super loop:

- The Super loop based firmware development approach is Suitable for applications that are not time critical and where the response time is not so important (Embedded systems where missing deadlines are acceptable).
- It is very similar to a conventional procedural programming where the code is executed task by task
- The tasks are executed in a never ending loop.
- The task listed on top on the program code is executed first and the tasks just below the top are executed after completing the first task
- A typical super loop implementation will look like:

1. Configure the common parameters and perform initialization for various hardware components memory, registers etc.
2. Start the first task and execute it
3. Execute the second task
4. Execute the next task
5. :
6. :
7. Execute the last defined task
8. Jump back to the first task and follow the same flow.

The 'C' program code for the super loop is given below void main ()

```
{
Configurations (); Initializations ();
while (1)
{
Task 1 ();
Task 2 ();
:
```

:

Task n ();

}

}

❖ Embedded firmware Design Approaches – Embedded OS based Approach:

- The embedded device contains an Embedded Operating System which can be one of:
  - A Real Time Operating System (RTOS)
  - A Customized General-Purpose Operating System (GPOS)

- OS based approach contains operating system which can be a "General purpose operating system (GPOS) & Real time operating system (RTOS).
- GPOS based design is very similar to a conventional PC based application development.

- Where device contain an OS windows/Unix/Linux etc for Desktop PC& you will be creating and running user applications

- Microsoft® Windows Embedded 8.1 is an example of GPOS for embedded devices

- Point of Sale (PoS) terminals, Gaming Stations, Tablet PCs etc are examples of embedded devices running on embedded GPOSs

- 'Windows CE', 'Windows Mobile','QNX', 'VxWorks', 'ThreadX', 'MicroC/OS-II', 'Embedded Linux', 'Symbian' etc are examples of RTOSs employed in Embedded Product development

- Mobile Phones, PDAs, Flight Control Systems etc are examples of embedded devices that runs on RTOSs

❖ Embedded firmware Development Languages

- **Assembly Language**
- **High Level Language**
  - o Subset of C (Embedded C)
  - o Subset of C++ (Embedded C++)

o Any other high level language with supported Cross-compiler

- **Mix of Assembly & High level Language**
  o Mixing High Level Language (Like C) with Assembly Code
  o Mixing Assembly code with High Level Language (Like C)

❖ Embedded firmware Development Languages – Assembly Language based Development

- 'Assembly Language' is the human readable notation of 'machine
- language'
- 'Machine language' is a processor understandable language
- Machine language is a binary representation and it consists of 1s and 0s
- Assembly language and machine languages are processor/controller
- dependent
- An Assembly language program written for one processor/controller family will not work with others.

- Assembly language programming is the process of writing processor specific machine code in mnemonic form, converting the mnemonics into actual processor instructions (machine language) and associated data using an assembler.
- The general format of an assembly language instruction is an Opcode followed by Operands The Opcode tells the processor/controller what to do and the Operands provide the data and information required to perform the action specified by the opcode

The 8051 Assembly Instruction

MOV A, #30

Moves decimal value 30 to the 8051 Accumulator register. Here MOV A is the Opcode and 30 is the operand (single operand). The same instruction when written in machine language will look like

011A10100 00011110

The first 8 bit binary value 01110100 represents the opcode MOV A and the second 8 bit binary value 00011110 represents the operand 30.

- A machine code program consists of a sequence of assembly language instructions, where each statement contains a mnemonic (Opcode + Operand)

Each line of an assembly language program is split into four fields as:

LABEL OPCODE OPERAND : COMMENTS

- LABEL is an optional field. A 'LABEL' is an identifier used extensively in

programs to reduce the reliance on programmers for remembering where data

or code is located. LABEL is commonly used for representing

- A memory location, address of a program, sub-routine, code portion etc.

- The maximum length of a label differs between assemblers. Assemblers insist strict formats for labeling. Labels are always suffixed by a colon and begin with a valid character. Labels can contain number from 0 to 9 and special character _ (underscore).

```
;#####################################################################
: SUBROUTINE FOR GENERATING DELAY
: DELAY PARAMETR PASSED THROUGH REGISTER R1
: RETURN VALUE NONE, REGISTERS USED: R0, R1
;#####################################################################
##### DELAY  :  MOV R0, #255  ;  Load Register R0 with 255
DJNZ R1, DELAY; Decrement R1 and loop till
RET; Return to calling program
```

- The symbol; represents the start of a comment. Assembler ignores the text in a line after the ; symbol while assembling the program
- DELAY is a label for representing the start address of the memory location where the piece of code is located in code memory
- The above piece of code can be executed by giving the label DELAY as part of the instruction. E.g. LCALL DELAY; LMP DELAY

❖ Assembly Language – Source File to Object File Translation:

- The Assembly language program written in assembly code is saved as .asm (Assembly file) file or a .src (source) file or a format supported by the assembler

- Similar to 'C' and other high level language programming, it is possible to have multiple source files called modules in assembly language programming. Each

module is represented by a '.asm' or '.src' file or the assembler supported file format similar to the '.c' files in C programming

- The software utility called 'Assembler' performs the translation of assembly code to machine code

- The assemblers for different family of target machines are different. A51 Macro Assembler from Keil software is a popular assembler for the 8051 family micro controller

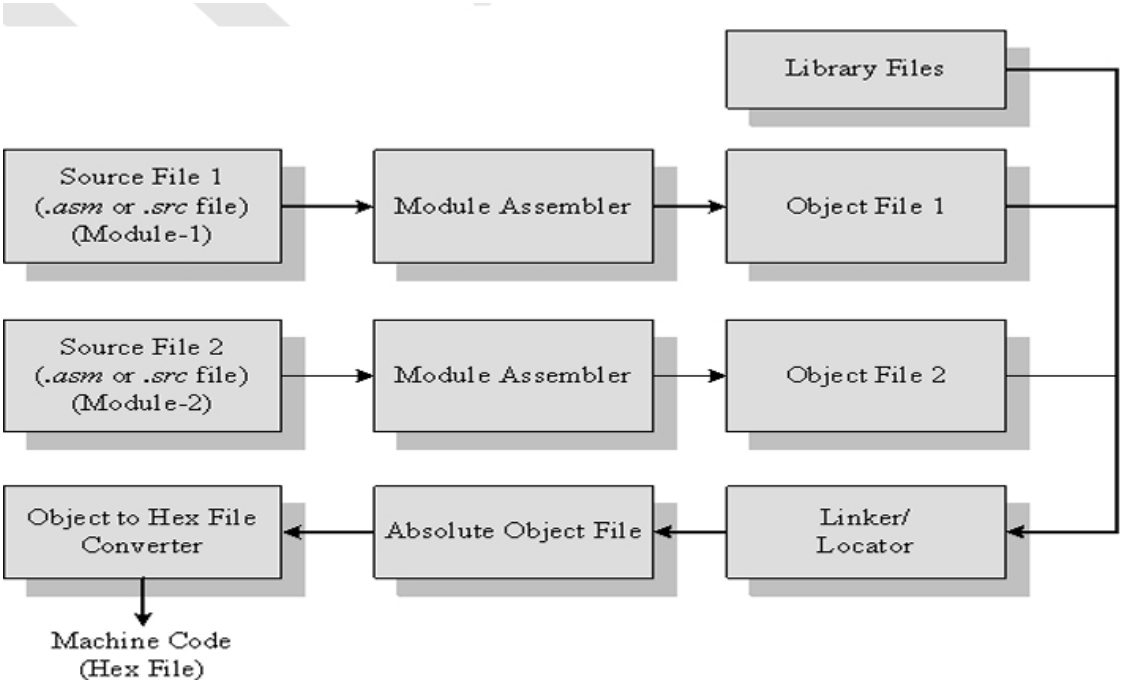


Figure 1: Assembly Language to machine language conversion process

- Each source file can be assembled separately to examine the syntax errors and incorrect assembly instructions.

- Assembling of each source file generates a corresponding object file. The object file does not contain the absolute address of where the generated code needs to be placed (a re-locatable code) on the program memory

- The software program called linker/locater is responsible for assigning absolute address to object files during the linking process

- The Absolute object file created from the object files corresponding to different source code modules contain information about the address where each instruction needs to be placed in code memory

- A software utility called 'Object to Hex file converter' translates the absolute object file to corresponding hex file (binary file)

Advantages of Assembly Language Based Development:

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

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

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

2.High Performance:

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

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

3.Low level Hardware Access:

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

4.Code Reverse Engineering:

- It is the process of understanding the technology behind a product by extracting the information from the finished product.
- It can easily be converted into assembly code using a dis-assembler program for the target machine.

Drawbacks of Assembly Language Based Development:

1.High Development time:

- The developer takes lot of time to study about architecture ,memory organization, addressing modes and instruction set of target processor/controller.
- More lines of assembly code is required for performing a simple action.

2.Developer dependency:

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

3.Non portable:

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

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

❖ High level language to machine language conversion process

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

- The Absolute object file created from the object files corresponding to different source code modules contain information about the address where each instruction needs to be placed in code memory
- A software utility called 'Object to Hex file converter' translates the absolute object file to corresponding hex file (binary file)

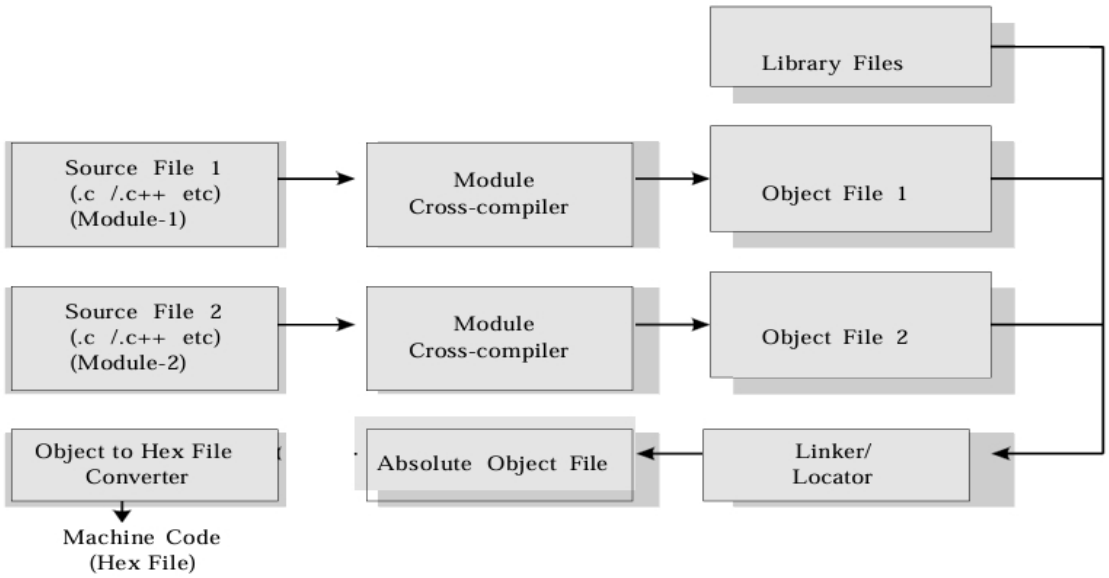


Figure: High level language to machine language conversion process

Advantages:

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

- Developer independency: The syntax used by most of the high level languages are universal and a program written high level can easily understand by a second person knowing the syntax of the language
- Portability: An Application written in high level language for particular target processor /controller can be easily be converted to another target processor/controller specific application with little or less effort

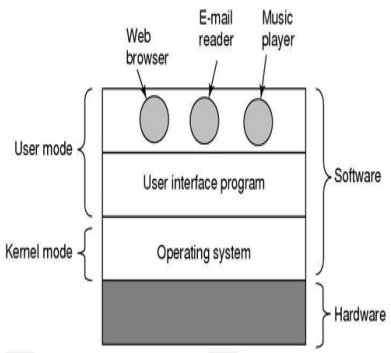
Drawbacks:

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

RTOS Based Embedded System Design

Operating System Basics:

- 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
  - OS manages resources and available to the system makes them the user applications/tasks on a need basis.



The primary functions of an Operating system is

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

- The primary functions of an Operating system is Make the system convenient to use Organize and manage the system resources efficiently.

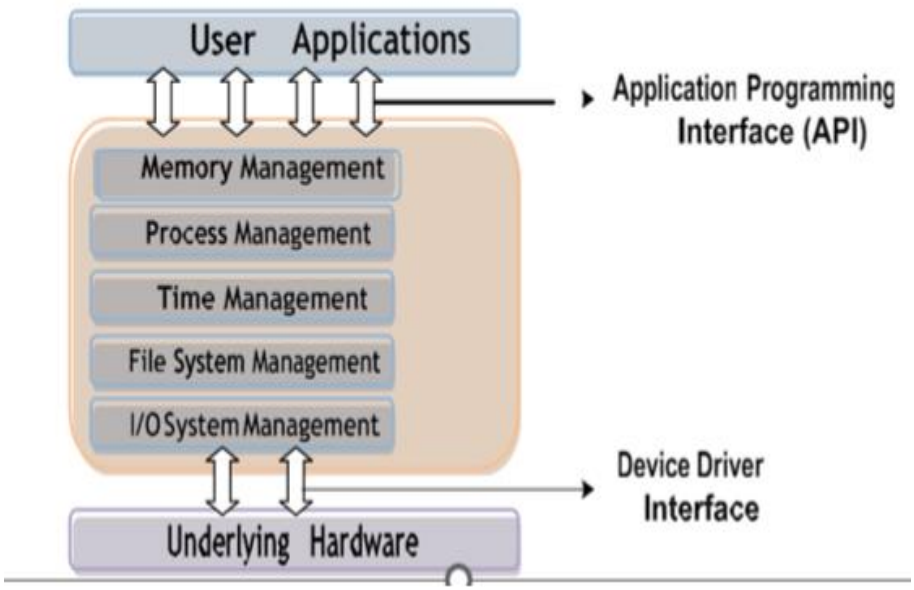


Figure 1: The Architecture of Operating System

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

- Process Management
- Primary Memory Management
- File System management
- I/O System (Device) Management
- Secondary Storage Management
- Protection
- Time management
- Interrupt Handling

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 synchronisation
- Process termination/deletion, etc.

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)

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.
- 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
- The various file system management operations are OS dependent.
- For example, the kernel of Microsoft DOS OS supports a specific set of file system management operations and they are not the same as the file system operations supported by UNIX Kernel.

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.
- May be available in advance or updated dynamically as and when a new device is installed.
- The service Device Manager of the kernel is responsible for handling all I/O device related operations.
- The kernel talks to the I/O device through a set of low-level systems calls, which are implemented in a service called device drivers.
- 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 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 Disk storage allocation
- Disk scheduling (Time interval at which the disk is activated to backup data) Free Disk space management

Protection Systems:

- Most of the modern operating systems are designed in such a way to support multiple users with different levels of access permissions.
- E.g. 'Administrator', 'Standard', 'Restricted' permissions in Windows XP.
- Protection deals with implementing the security policies to restrict the access both user and system resources by different applications or processes or users.
- ln multiuser supported operating systems, one user may not be allowed to view or modify the whole or portions of another user's data or profile details.
- In addition, some application may not granted with permission to make use of some of the system resources.
- This kind of protection is provided by the protection services running within the kernel.

Interrupt Handler:

- Kernel provides handler mechanism for all external/internal interrupts generated by the system.

Kernel Space and User Space:

- The program code corresponding to the kernel applications/services are kept in a contiguous area (OS dependent) 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'
- 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
- An operating system with virtual memory support, loads the user applications into its corresponding virtual memory space with demand paging technique application

- 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:

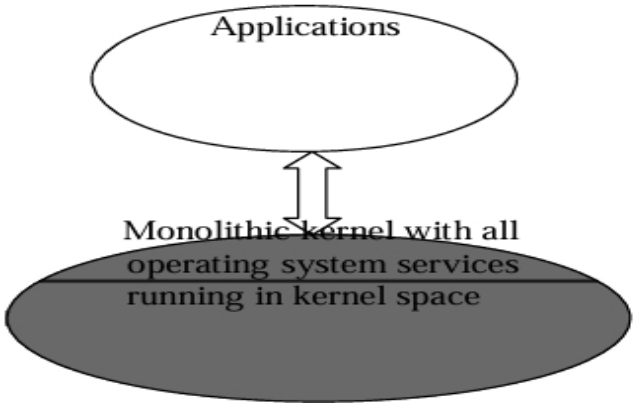


Figure 2: The Monolithic Kernel Model

- All kernel services run in the kernel space
- All kernel modules run within the same memory space under a single kernel thread
- The tight internal integration of kernel modules in monolithic kernel architecture allows the effective utilization of the low-level features of the underlying system
- The major drawback of monolithic kernel is that any error or failure in any one of the kernel module leads to the crashing Applications of the entire kernel
- LINUX, SOLARIS, MS-DOS kernels
    - are examples of monolithic kernel

Microkernel:

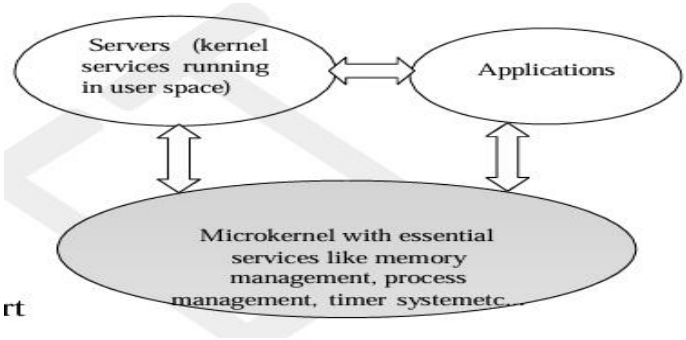


Figure 3: The Microkernel Model

- The microkernel design incorporates only the essential set of Operating System services into the kernel
- Rest of the Operating System services are implemented in programs known as 'Servers' which runs in user space
- The kernel design is highly modular provides OS-neutral abstraction.
- Memory management, process management, timer systems an interrupt handlers are examples of essential services, which forms the part
- QNX, Minix 3 kernels are examples for microkernel.

Benefits of Microkernel:

1. Robustness: If a problem is encountered in any services in server can reconfigured and re-started without the need for re-starting the entire OS.

2. Configurability: Any services , which run as 'server' application can be changed without need to restart the whole system.

Types of Operating Systems:

Depending on the type of kernel and kernel services, purpose and type of computing systems where the OS is deployed and the responsiveness to applications, Operating Systems are classified into

1. General Purpose Operating System (GPOS):
2. Real Time Purpose Operating System (RTOS):

1. General Purpose Operating System (GPOS):

- Operating Systems, which are deployed in general computing systems

- The kernel is more generalized and contains all the required services to execute generic applications
- Need not be deterministic in execution behaviour
- May inject random delays into application software and thus cause slow responsiveness of an application at unexpected times
- Usually deployed in computing systems where deterministic behaviour is not an important criterion
- Personal Computer/Desktop system is a typical example for a systemwhere GPOSs are deployed.
- Windows XP/MS-DOS etc are examples of General Purpose Operating System

Real Time Purpose Operating System (RTOS):

- Operating Systems, which are deployed in embedded systems demanding real-time response
- Deterministic in execution behaviour. Consumes only known amount of time for kernel applications
- Implements scheduling policies for executing the highest priority task/application always
- Implements policies and rules concerning time-critical allocation of a system's resources
- 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 Timekernel are

a) Task/Process management

b) Task/Process scheduling

c) Task/Process synchronization

d) Error/Exception handling

e) Memory Management

f) Interrupt handling

g) Time management

Real Time Kernel 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*
- *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 ϖ*
- *Task Pointers: Pointers to other TCBs (TCBs for preceding, next and waiting tasks) ϖ 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 nothing but an algorithm implementation, which performs the efficient and optimal scheduling of tasks to provide a deterministic behaviour.

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 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).

Memory Management:

- The memory management function of an RTOS kernel is slightly different compared to the General Purpose Operating Systems
- 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 for an RTOS, RTOS achieves this 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
- RTOS kernels assume that the whole design is proven correct and protection is unnecessary. 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 of an RTOS kernel is slightly different compared to the General Purpose Operating Systems
- A few RTOS kernels implement Virtual Memory concept for memory

allocation if the system supports secondary memory storage (like HDD and FLASH memory).

- In the 'block' based memory allocation, 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.
- The memory allocation can be implemented as constant functions and thereby it consumes fixed amount of time for memory allocation. This leaves the deterministic behavior of the RTOS kernel untouched.

Interrupt Handling:

- 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, 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. Interrupt nesting allows the pre-emption (interruption) of an Interrupt Service Routine (ISR), servicing an interrupt, by a higher priority interrupt.

Time Management:

- Interrupts inform the processor that an external device or an associated task requires immediate attention of the CPU.
- 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 1microsecond, the System time register will reset in
- 232 * 10-6/ (24 * 60 * 60) = 49700 Days =~ 0.0497 Days = 1.19 Hours
- If the 'Timer tick' interval is 1 millisecond, the System time register will reset in 232 * 10-3 / (24 * 60 * 60) = 497 Days = 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 System:

- A Real Time Operating Systems which strictly adheres to the timing constraints for a task
- 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
- Emphasize on the principle 'A late answer is a wrong answer'
- Air bag control systems and Anti-lock Brake Systems (ABS) of vehicles are typical examples of Hard Real Time Systems
- As a rule of thumb, Hard Real Time Systems does not implement the virtual memory model for handling the memory. This eliminate the delay in swapping in and out the code corresponding to the task to and from the primary memory
- The presence of Human in the loop (HITL) for tasks introduces unexpected delays in the task execution. Most of the Hard Real Time Systems are automatic and does not contain a 'human in the loop'

Soft Real-time System:

- Real Time Operating Systems that does not guarantee meeting deadlines, but, offer the best effort to meet the deadline
- Missing deadlines for tasks are acceptable if the frequency of deadline missing is within the compliance limit of the Quality of Service(QoS)
- A Soft Real Time system emphasizes on the principle 'A late answer ian acceptable answer, but it could have done bit faster'
- Soft Real Time systems most often have a 'human in the loop (HITL)' 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.

- An audio video play back system is another example of Soft Real Time system. No potential damage arises if a sample comes late by fraction of a second, for play back.

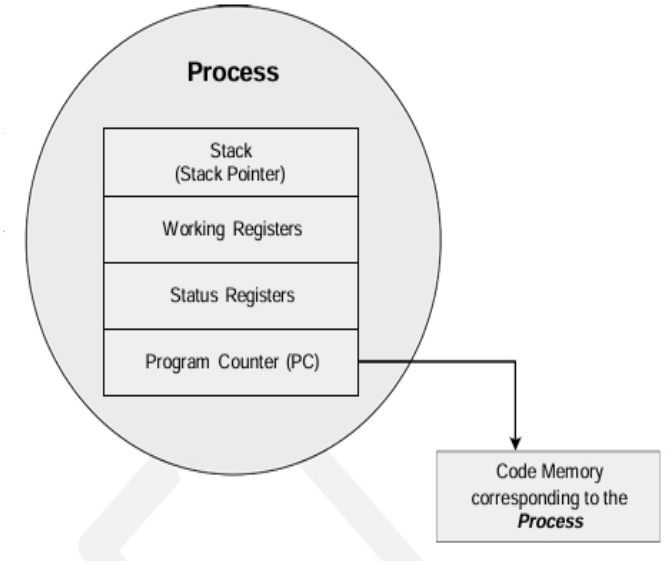
Tasks, Processes & Threads :

- 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 systemcontext
- A program or part of it in execution is also called a 'Process' The structure of a Processes
- The terms 'Task', 'job' and 'Process' refer to the same entity in the Operating System context and most often they are usedinterchangeably
- 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

The structure of a Processes

- The concept of 'Process' leads to concurrent execution (pseudo parallelism) of tasks and thereby the 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
- 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

Figure: 4 Structure of a Process



- When the process gets its turn, its registers and Program counter register becomes mapped to the physical registers of the CPU

Memory organization of Processes:

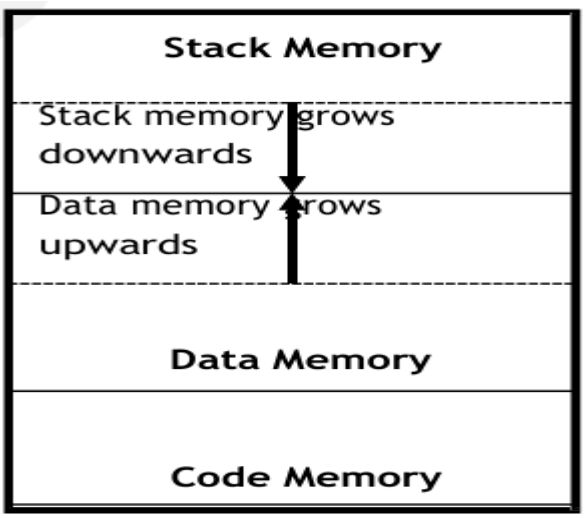


Fig: 5 Memory organization of a Process

- The memory occupied by the process is segregated into three region 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

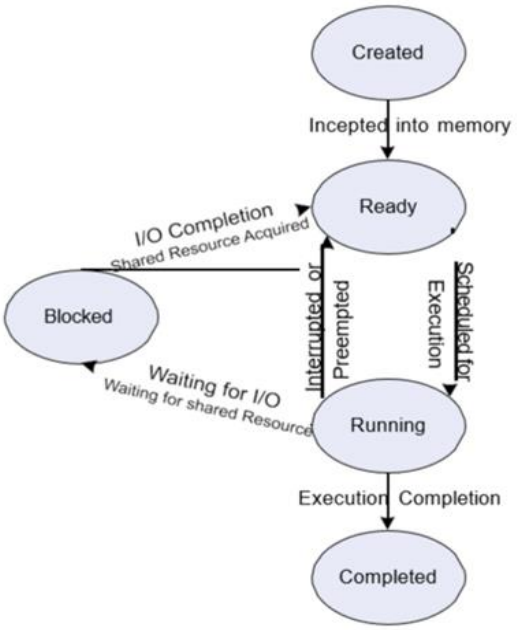
- On loading a process into the main memory, a specific area of memory is allocated for the process
- The stack memory usually starts at the highest memory address from the memory area allocated for the process (Depending on the OS kernel implementation)

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 Process States & State Transition: '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

Process States & State Transition:

- **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
  - The transition of a process from one state to another is known as '*State transition*'
  - When a process changes its state from Ready to running or from running to blocked or terminated or from blocked to running, the CPU allocation for the process may also change

Process Management

Process management deals with
- creation of a process
- setting up the memory space for the process
- loading the process's code into the memory space
- allocating system resources
- setting up a Process Control Block (PCB) for the process
- process termination/deletion

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

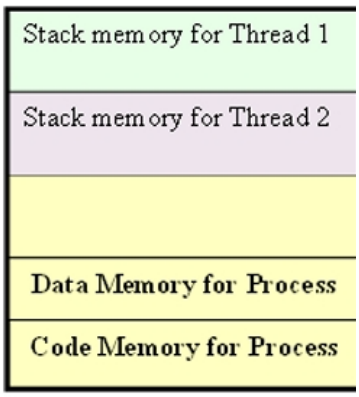
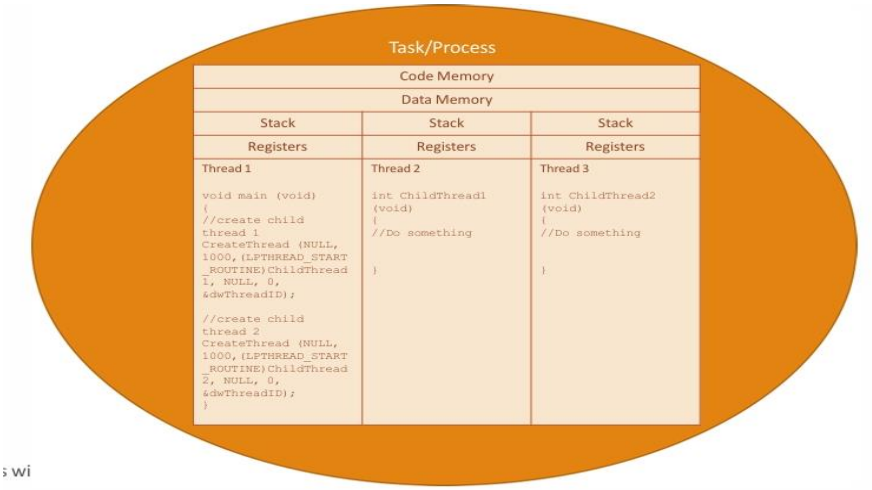


Figure 7 Memory organization of process and its associated Threads

The Concept of multithreading:

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 utilised 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
  - Win32 Threads
  - Java Threads

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);
```

- This primitive 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'.
- 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 (new_thread ID in our example).

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

- This primitive blocks the current thread and waits until the completion of the thread pointed by it (new_thread in this example).
- All the POSIX 'thread calls' returns an integer.
- A return value of zero indicates the success of the call.
- The termination of a thread can happen in different ways:
  - Natural termination:
    - The thread completes its execution and returns to the main thread through a simple return or by executing the pthread_exit() call.
  - Forced termination:
    - This can be achieved by the call pthread_cancel() or through the termination of the main thread with exit or exec functions.

pthread_cancel() call is used by a thread to terminate another thread.

POSIX Threads – Example

- 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 on an OS where POSIX library is available
#include<pthread.h>
#include<stdlib.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");
  for(i=0; i<10000; i++);            //Wait for some time. Do nothing.
 }
 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");
  for(i=0; i<10000; i++);            //Wait for some time. Do nothing.
 }
 if (pthread_join(tcb, NULL))
 {
  //Thread join failed
  printf("Error in Thread join\n");
  return -1;
 }
return 1;
}
```

Types of Threads:

• Kernel Level Threads:

- 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.
- In summary, kernel level threads are pre-emptive.

- 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.

Thread Binding Models:

There are many ways for binding user level threads with system kernel level threads.

- Many-to-One Model
  - Here, many user level threads are mapped to a single kernel thread.
  - In this model, 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.
  - The 'PThread' example is an illustrative example for application with Many- to-One thread model.
- One-to-One Model
  - Here, each user level thread is bonded to a kernel/system level thread.
  - Windows XP/NT/2000 and Linux threads are examples for One-to-One thread models.
  - The modified 'PThread' example is an illustrative example for application with One-to-One thread model.
- Many-to-Many Model
  - In this model, many user level threads are allowed to be mapped to many kernel threads.
  - Windows NT/2000 with ThreadFibre package is an example for this.

Advantages of Threads:

1. 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.

2. Efficient CPU utilization: The CPU is engaged all time.

3. Speeds up the execution of the process: 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.

Thread V/s 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. It shares the data memory and heap memory with other threads of the same process.	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 thread (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 (shares the total stack memory of the process).
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 lot of OS overhead and is comparatively slower.
If a thread expires, its stack is reclaimed by the process.	If a process dies, the resources allocated to it are reclaimed by the OS and all the associated threads of the process also dies.

Hardware Software Co design and Program Modelling

Hardware Software Co-Design:

- In the traditional embedded system development approach, the hardware software partitioning is done at an early stage.
- Engineers from the software group take care of the software architecture development and implementation, whereas engineers from the hardware group are responsible for building the hardware required for the product.
- There is less interaction between the two teams and the development happens either serially or in parallel.
- Once the hardware and software are ready, the integration is performed.
- The increasing competition in the commercial market and need for reduced 'time-to- market' the product calls for a novel approach for embedded system design in which the hardware and software are co-developed instead of independently developing both.

- During the co-design process, the product requirements captured from the customer are converted into system level needs or processing requirements. At this point of time it is not segregated as either hardware requirement or software requirement, instead it is specified as functional requirement.
- The system level processing requirements are then transferred into functions which can be simulated and verified against performance and functionality.
- The Architecture design follows the system design.
  - The partition of system level processing requirements into hardware and software takes place during the architecture design phase.
  - Each system level processing requirement is mapped as either hardware and/or software requirement.
  - The partitioning is performed based on the hardware-software trade-offs.
- The architectural design results in the detailed behavioural description of the hardware requirement and the definition of the software required for the hardware.
- The processing requirement behaviour is usually captured using computational models.
- The models representing the software processing requirements are translated into firmware implementation using programming languages.

Fundamental Issues in Hardware Software Co-Design:

➢ The fundamental issues in hardware software co-design are:

- Selecting the Model
- Selecting the Architecture
- Selecting the Language
- Partitioning System Requirements into Hardware and Software

Selecting the Model:

➢ In hardware software co-design, models are used for capturing and describing the system characteristics.

➢ A model is a formal system consisting of objects and composition rules.

➢ It is hard to make a decision on which model should be followed in a particular system design.

➢ Most often designers switch between a variety of models from the requirements specification to the implementation aspect of the system design.

- The reason being, the objective varies with each phase.
- For example, at the specification stage, only the functionality of the system

is in focus and not the implementation information.

- When the design moves to the implementation aspect, the information about the system components is revealed and the designer has to switch to a model capable of capturing the system's structure.

Selecting the Architecture:

- A model only captures the system characteristics and does not provide information on 'how the system can be manufactured?'.
- The architecture specifies how a system is going to implement in terms of the number and types of different components and the interconnection among them.
- The commonly used architectures in system design are Controller Architecture, Datapath Architecture, Complex Instruction Set Computing (CISC), Reduced Instruction Set Computing (RISC), Very Long Instruction Word Computing (VLIW), Single Instruction Multiple Data (SIMD), Multiple Instruction Multiple Data (MIMD), etc.
    - Some of them fall into Application Specific Architecture Class (like controller architecture), while others fall into either general purpose architecture class (CISC, RISC, etc.) or Parallel processing class (like VLIW, SIMD, MIMD, etc.).

Selecting the Language:

- A programming language captures a 'Computational Model' and maps it into architecture.
- There is no hard and fast rule to specify this language should be used for capturing this model.
- A model can be captured using multiple programming languages like C, C++, C#, Java, etc. for software implementations and languages like VHDL, System C, Verilog, etc. for hardware implementations.
- On the other hand, a single language can be used for capturing a variety of models.
- Certain languages are good in capturing certain computational model.
- For example, C++ is a good candidate for capturing an object oriented model.
- The only pre-requisite in selecting a programming language for capturing a model is that the language should capture the model easily.

Partitioning System Requirements into Hardware and Software:

- From an implementation perspective, it may be possible to implement the system requirements in either hardware or software (firmware).
- It is a tough decision making task to figure out which one to opt.
- Various hardware software trade-offs are used for making a decision on the hardware- software partitioning.

Computational Models in Embedded Design:

The commonly used computational models in embedded system design are:

- Data Flow Graph Model
- Control Data Flow Graph Model
- State Machine Model
- Sequential Program Model
- Concurrent/Communicating Process Model
- Object-Oriented Mode

Data Flow Graph/Diagram (DFG) Model:

- The Data Flow Graph (DFG) model translates the data processing requirements into a data flow graph.
- It is a data driven model in which the program execution is determined by data.
- This model emphasises on the data and operations on the data which transforms the input data to output data.
- Embedded applications which are computational intensive and data driven are modelled using the DFG model.
  - DSP applications are typical examples for it.
- Data Flow Graph (DFG) is a visual model in which the operation on the data (process) is represented using a block (circle) and data flow is represented using arrows.
- An inward arrow to the process (circle) represents input data and an outward arrow from the process (circle) represents output data in DFG notation.
- Suppose one of the functions in our application contains the computational requirement □ □ = $a + b$ and $y = x - c$.
- Figure illustrates the implementation of a DFG model for implementing these requirements.

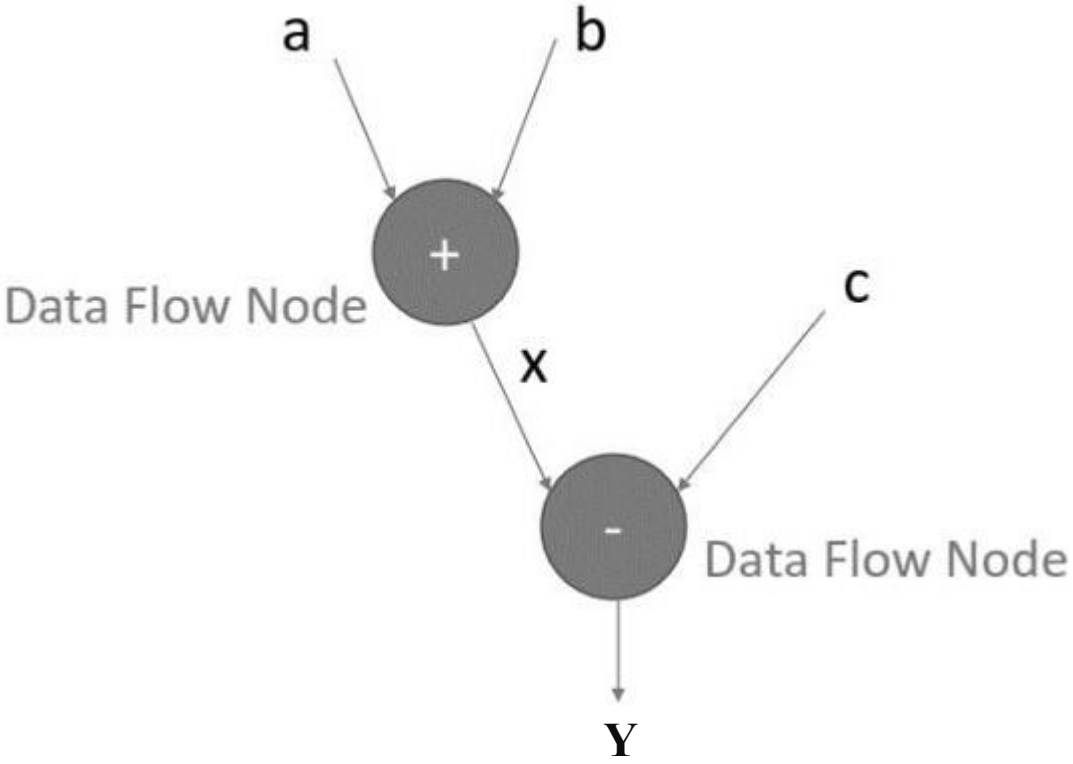


Fig : Diagram (DFG) Model

- In a DFG model, a data path is the data flow path from input to output.
- A DFG model is said to be acyclic DFG (ADFG) if it doesn't contain multiple values for the input variable and multiple output values for a given set of input(s).
    - Feedback inputs (Output is fed back to Input), events, etc. are examples for non- acyclic inputs.
- A DFG model translates the program as a single sequential process execution.

Control Data Flow Graph/Diagram (CDFG) Model:

- The DFG model is a data driven model in which the execution is controlled by data and it doesn't involve any control operations (conditionals).
- The Control DFG (CDFG) model is used for modelling applications involving conditional program execution.
- CDFG models contains both data operations and control operations. The CDFG uses Data Flow Graph (DFG) as element and conditional (constructs) as decision makers.
- CDFG contains both data flow nodes and decision nodes, whereas DFG contains

only data flow nodes.

- Consider the implementation of the CDFG for the following requirement.
- $If\ flag = 1, x = a + b; else\ y = a - b;$
- This requirement contains a decision-making process.
- The CDFG model for the same is given in the figure.
- The control node is represented by a 'Diamond' block which is the decision-making element in a normal flow chart-based design.
- CDFG translates the requirement, which is modelled to a concurrent process model.
- The decision on which process is to be executed is determined by the control node.

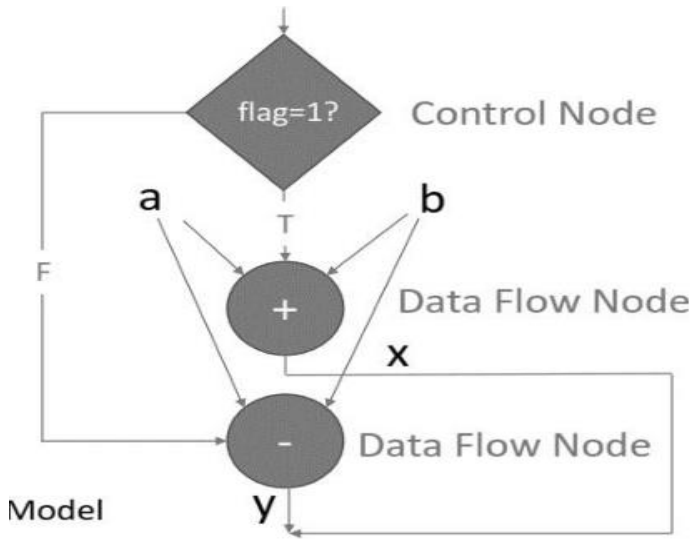


Fig : Control Data Flow Graph Model

- A real world example for modelling the embedded application using CDFG is capturing and saving of the image to a format set by the user in a digital still camera.
- The decision on, in which format the image is stored (formats like JPEG, TIFF, BMP, etc.) is controlled by the camera settings, configured by the user.

State Machine Model:

- The State Machine Model is used for modelling reactive or event-driven embedded systems whose processing behaviour are dependent on state

transitions.

- Embedded systems used in the control and industrial applications are typical examples for event driven systems.

➢ The State Machine model describes the system behaviour with 'States', 'Events', 'Actions' and 'Transitions'.

- State is a representation of a current situation.
- An event is an input to the state.
  - ✓ The event acts as stimuli for state transition.
- Transition is the movement from one state to another.
- Action is an activity to be performed by the state machine.

Finite State Machine (FSM) Model:

➢ A Finite State Machine (FSM) model is one in which the number of states are finite.

- The system is described using a finite number of possible states.

➢ As an example, let us consider the design of an embedded system for driver/passenger 'Seat Belt Warning' in an automotive using the FSM model.

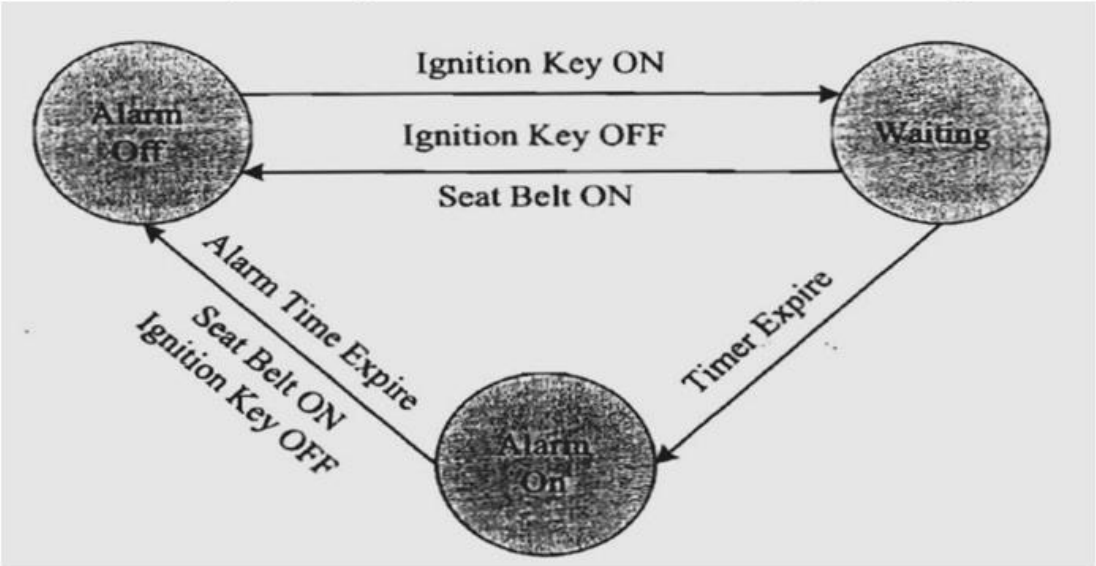
➢ The system requirements are captured as.

- When the vehicle ignition is turned on and the seat belt is not fastened within 10 seconds of ignition ON, the system generates an alarm signal for 5 seconds.
- The Alarm is turned off when the alarm time (5 seconds) expires or if the driver/passenger fastens the belt or if the ignition switch is turned off, whichever happens first.

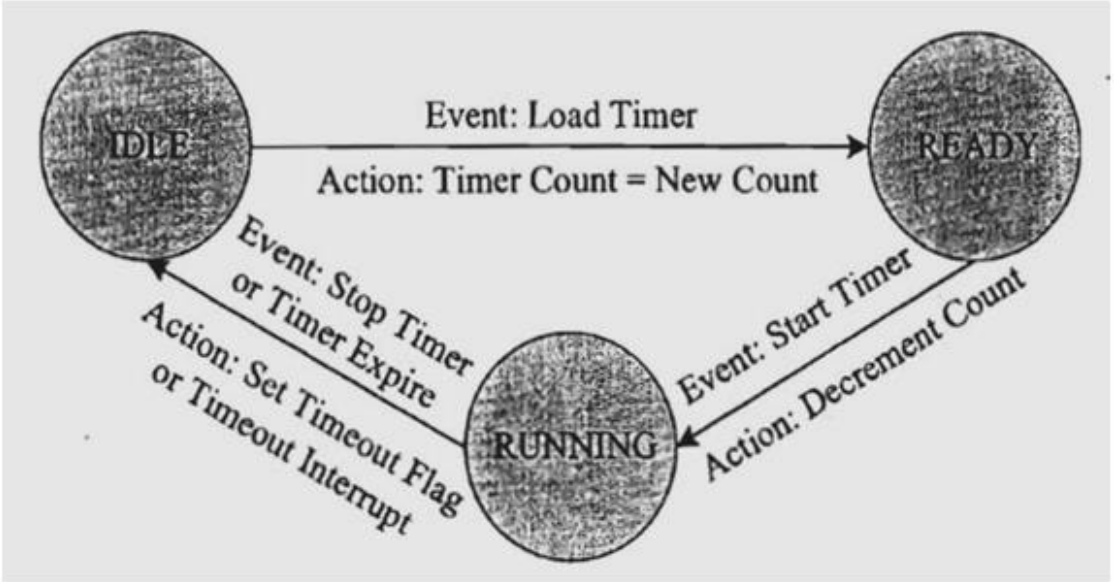
➢ Here the states are

- 'Alarm Off' •
- 'Waiting' • '
- Alarm On'

- The events are
  - 'Ignition Key ON'
  - 'Ignition Key OFF'
  - 'Timer Expire'
  - Alarm Time Expire'
  - 'Seat Belt ON'
- Using the FSM, the system requirements can be modeled as given in figure.



- The 'Ignition Key ON' event triggers the 10 second timer and transitions the state to 'Waiting'.
- If a Seat Belt ON' or 'Ignition Key OFF' event occurs during the wait state, the state transitions into 'Alarm Off'.
- When the wait timer expires in the waiting state, the event 'Timer Expire' is generated and it transitions the state to 'Alarm On' from the 'Waiting' state.
- The 'Alarm On' state continues until a 'Seat Belt ON' or 'Ignition Key OFF' event or 'Alarm Time Expire' event, whichever occurs first.
- The occurrence of any of these events transitions the state to 'Alarm Off'.
- The wait state is implemented using a timer.
  - The timer also has certain set of states and events for state transitions.
  - Using the FSM model, the timer can be modelled as shown in the figure.



- As seen from the FSM, the timer state can be either 'IDLE' or 'READY' or 'RUNNING'.
- As seen from the FSM, the timer state can be either 'IDLE' or 'READY' or 'RUNNING'.
- The timer is said to be in the 'READY' state when the timer is loaded with the count corresponding to the required time delay
- The timer remains in the 'READY' state until a 'Start Timer' event occurs.
- The timer changes its state to 'RUNNING' from the 'READY' state on receiving a 'Start Timer' event and remains in the 'RUNNING' state until the timer count expires or a 'Stop Timer' even occurs.
- The timer state changes to 'IDLE' from 'RUNNING' on receiving a 'Stop Timer' or 'Timer Expire' event.

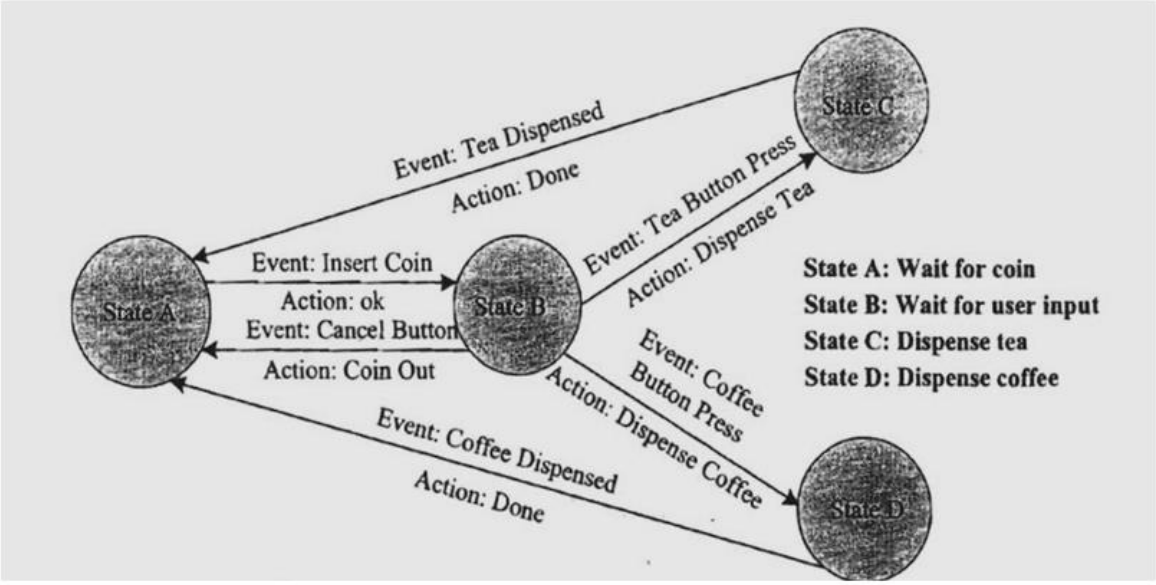
FSM Model – Example 1

- Design an automatic tea/coffee vending machine based on FSM model for the following requirement.
  - The tea/coffee vending is initiated by user inserting a 5 rupee coin.
  - After inserting the coin, the user can either select 'Coffee' or 'Tea' or press 'Cancel' to cancel the order and take back the coin.

Solution:

The FSM Model contains four states namely:

- Wait for coin'
- 'Wait for User Input'
- Dispense Tea'
- 'Dispense Coffee'

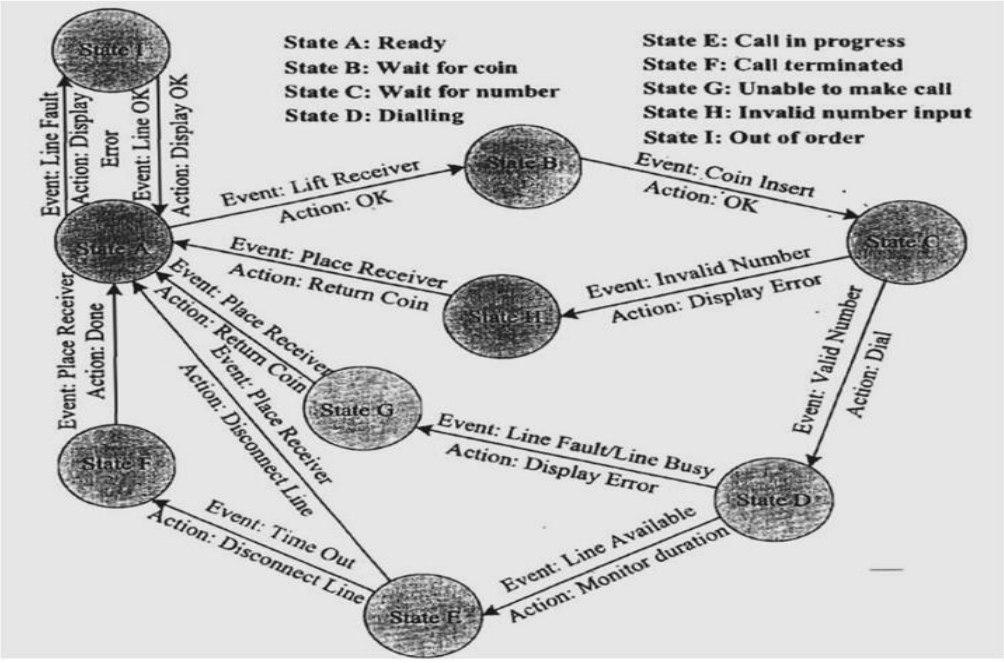


- The event 'Insert Coin' (5 rupee coin insertion), transitions the state to 'Wait for User Input'.
- The system stays in this state until a user input is received from the buttons 'Cancel', 'Tea' or 'Coffee' (Tea and Coffee are the drink select button).
- If the event triggered in 'Wait State' is 'Cancel' button press, the coin is pushed out and the state transitions to 'Wait for Coin'.
- If the event received in the 'Wait State' is either 'Tea' button press, or 'Coffee' button press, the state changes to 'Dispense Tea' and 'Dispense Coffee' respectively.
- Once the coffee/tea vending is over, the respective states transition back to the 'Wait for Coin' state.

FSM Model – Example 2

- Design a coin operated public telephone unit based on FSM model for the following requirements.
  - The calling process is initiated by lifting the receiver (off-hook) of the telephone unit.
  - After lifting the phone the user needs to insert a 1 rupee coin to make the call.
  - If the line is busy, the coin is returned on placing the receiver back on the hook (on-hook).
  - If the line is through, the user is allowed to talk till 60 seconds and at the end of 45th second, prompt for inserting another 1 rupee coin for continuing the call is initiated.

- If the user doesn't insert another 1 rupee coin, the call is terminated on completing the 60 seconds time slot.
- The system is ready to accept new call request when the receiver is placed back on the hook (on-hook).
- The system goes to the 'Out of Order' state when there is a line fault.

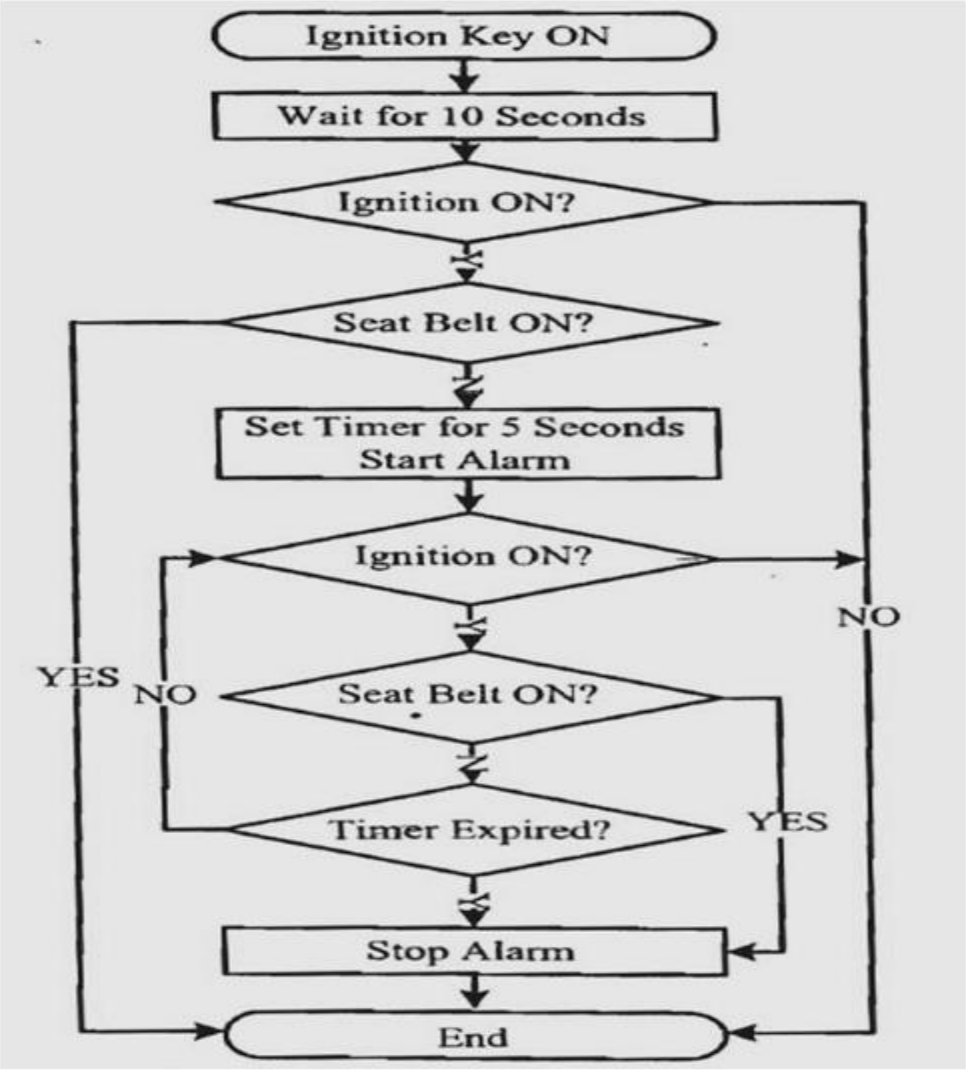


Sequential Program Model:

- In the Sequential Program Model, the functions or processing requirements are executed in sequence.
  - It is same as the conventional procedural programming.
- Here the program instructions are iterated and executed conditionally and the data gets transformed through a series of operations.
- Finite State Machines (FSMs) and Flow Charts are used for modelling sequential program
  - he FSM approach represents the states, events, transitions and actions, whereas the Flow Chart models the execution flow.
- The execution of functions in a sequential program model for the 'Seat Belt Warning' system is illustrated below:

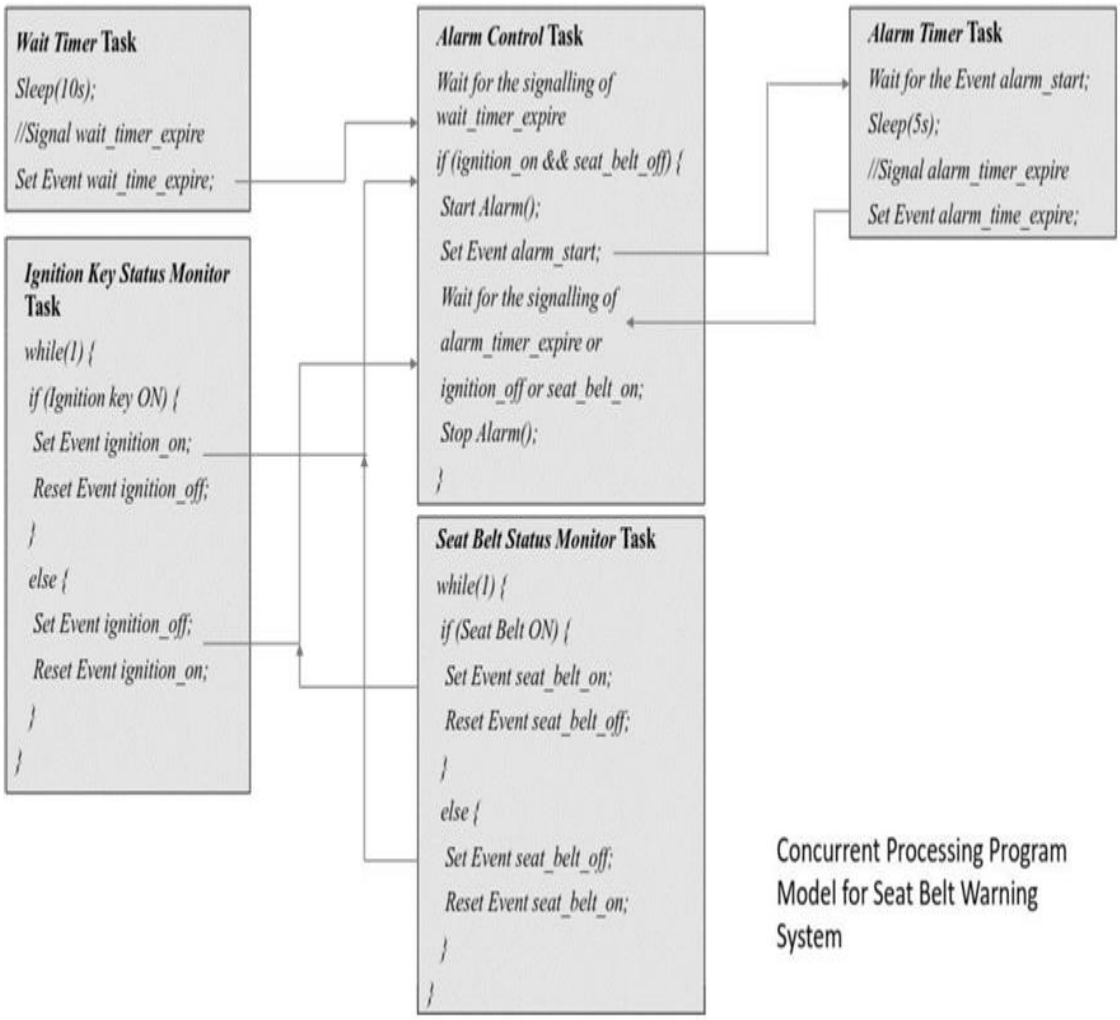
```
#define ON 1
#define OFF 0
#define YES 1
#define NO 0
void seat_belt_warn()
{
 wait_10sec();
 if (check_ignition_key()==ON)
 {
  if (check_seat_belt()==OFF)
  {
   set_timer(5);
   start_alarm();
   while ((check_seat_belt()==OFF)&&(check_ignition_key()==ON)&&(timer_expire()==NO));
   stop_alarm();
   }
  }
 }
}
```

- Sequential Program Model for Seat Belt Warning System



Concurrent/Communicating Process Model:

- The concurrent or communicating process model models concurrently executing tasks/processes.
- It is easier to implement certain requirements in concurrent processing model than the conventional sequential execution.
  - Sequential execution leads to a single sequential execution of task and thereby leads to poor processor utilisation, when the task involves I/O waiting, sleeping for specified duration etc.
  - If the task is split into multiple subtasks, it is possible to tackle the CPU usage effectively by switching the task execution, when the subtask under execution goes to a wait or sleep mode.
  - However, concurrent processing model requires additional overheads in task scheduling, task synchronization and communication.
  - As an example, consider the implementation of the 'Seat Belt Warning' system using concurrent processing model.
  - We can split the tasks into:
    Timer task for waiting 10 seconds (wait timer task)
    Task for checking the ignition key status (ignition key status monitoring task)
    Task for checking the seat belt status (seat belt status monitoring task)
    Task for starting and stopping the alarm (alarm control task)
    Alarm timer task for waiting 5 seconds (alarm timer task)
  - The tasks cannot be executed them randomly or sequentially. ¬ We need to synchronize their execution through some mechanism.

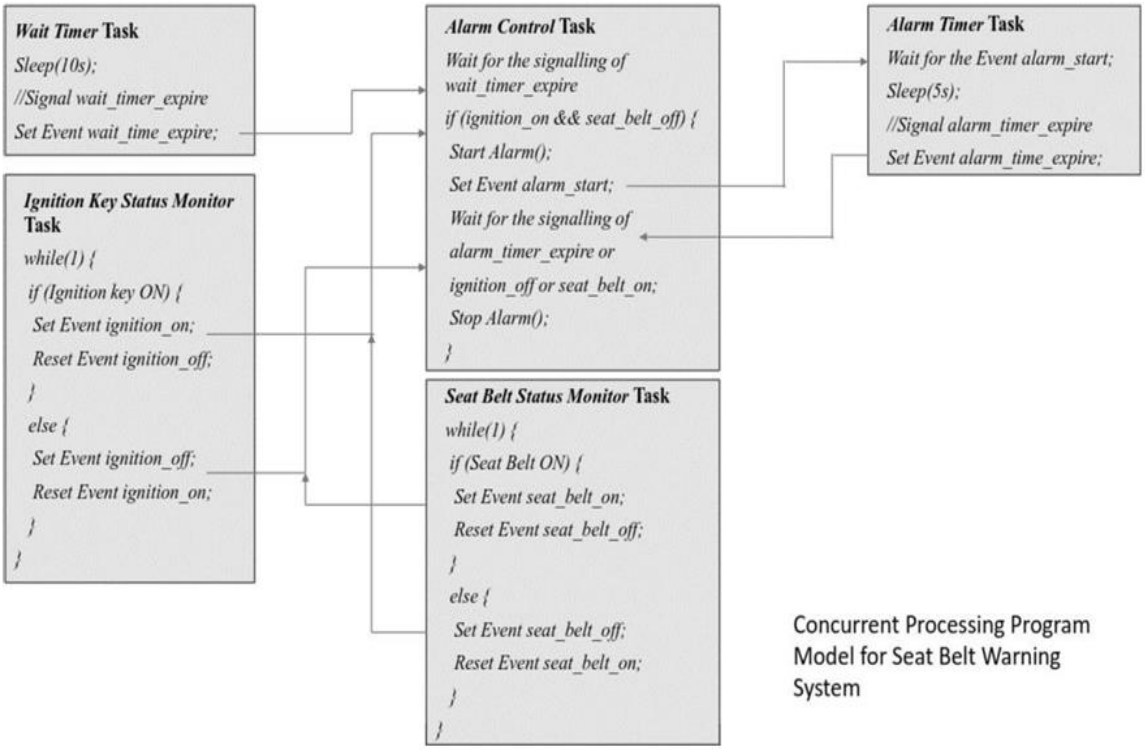


Concurrent Processing Program Model for Seat Belt Warning System

Object-Oriented Model:

- The object-oriented model is an object based model for modelling system requirements.
- It disseminates a complex software requirement into simple well defined pieces called objects.
- Object-oriented model brings re-usability, maintainability and productivity in system design.
- In the object-oriented modelling, object is an entity used for representing or modelling a particular piece of the system.
  - Each object is characterized by a set of unique behaviour and state.

- A class is an abstract description of a set of objects and it can be considered as a 'blueprint' of an object.
- A class represents the state of an object through member variables and object behaviour through member functions.
- The member variables and member functions of a class can be private, public or protected.
  - Private member variables and functions are accessible only within the class, whereas public variables and functions are accessible within the class as well as outside the class.
  - The protected variables and functions are protected from external access.
  - However, classes derived from a parent class can also access the protected member functions and variables.



Concurrent Processing Program Model for Seat Belt Warning System