Module -4
Embedded System Design Concepts

Characteristics & Quality Attributes of Embedded Systems

Characteristics of Embedded System

Each Embedded System possess a set of characteristics which are unique to it. Some important characteristics of embedded systems are:

- ❑ Application & Domain Specific
- ❑ Reactive & Real Time
- ❑ Operates in ‘harsh’ environment
- ❑ Distributed
- ❑ Small size and Weight
- ❑ Power Concerns

Quality Attributes of Embedded Systems:

Represent the non-functional requirements that needs to be addressed in the design of an embedded system. The various quality attributes that needs to be addressed in any embedded system development are broadly classified into

- ❑ Operational Quality Attributes

Refers to the relevant quality attributes related to tan embedded system when it is in the operational mode or '***online***' mode

- ❑ Non-Operational Quality Attributes

The Quality attributes that needs to be addressed for the product 'not' on the basis of operational aspects are grouped under this category

Operational Quality Attributes

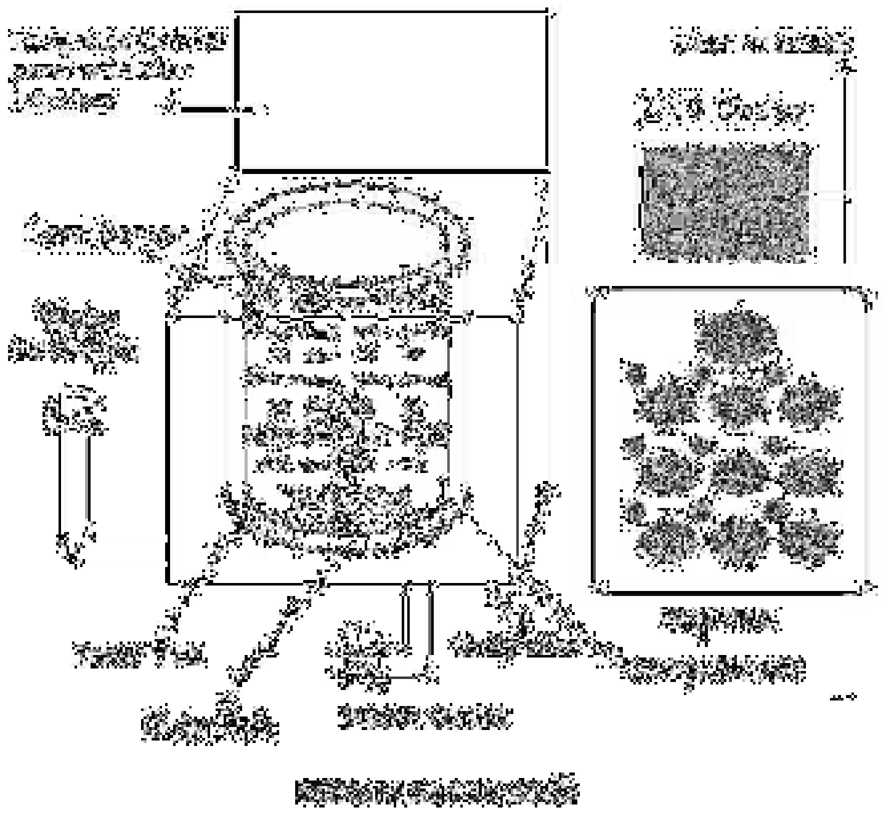
- ❑ Response
- ❑ Throughput
- ❑ Reliability
- ❑ Maintainability
- ❑ Security
- ❑ Safety

Non-Operational Quality Attributes

- ❑ Testability & Debug-ability
- ❑ Evolvability
- ❑ Portability
- ❑ Time to Prototype and Market
- ❑ Per Unit and Total Cost

Washing Machine – Application Specific Embedded System

- ✓ Extensively used in Home Automation for washing and drying clothes
- ✓ Contains User Interface units (I/O) like Keypads, Display unit, LEDs for accepting user inputs and providing visual indications
- ✓ Contains sensors like, water level sensor, temperature sensor etc.
- ✓ Contains actuators like spin and agitation control motor units
- ✓ Contains an integrates embedded controller for controlling the washing operations
- ✓ Sensors, actuators and I/O devices are interfaced to the I/O subsystem of the embedded control unit
- ✓ Specifically designed for serving the application 'Wash & Rinse' of clothes and cannot be used for any other applications



Automotive – Domain Specific Example of Embedded Systems

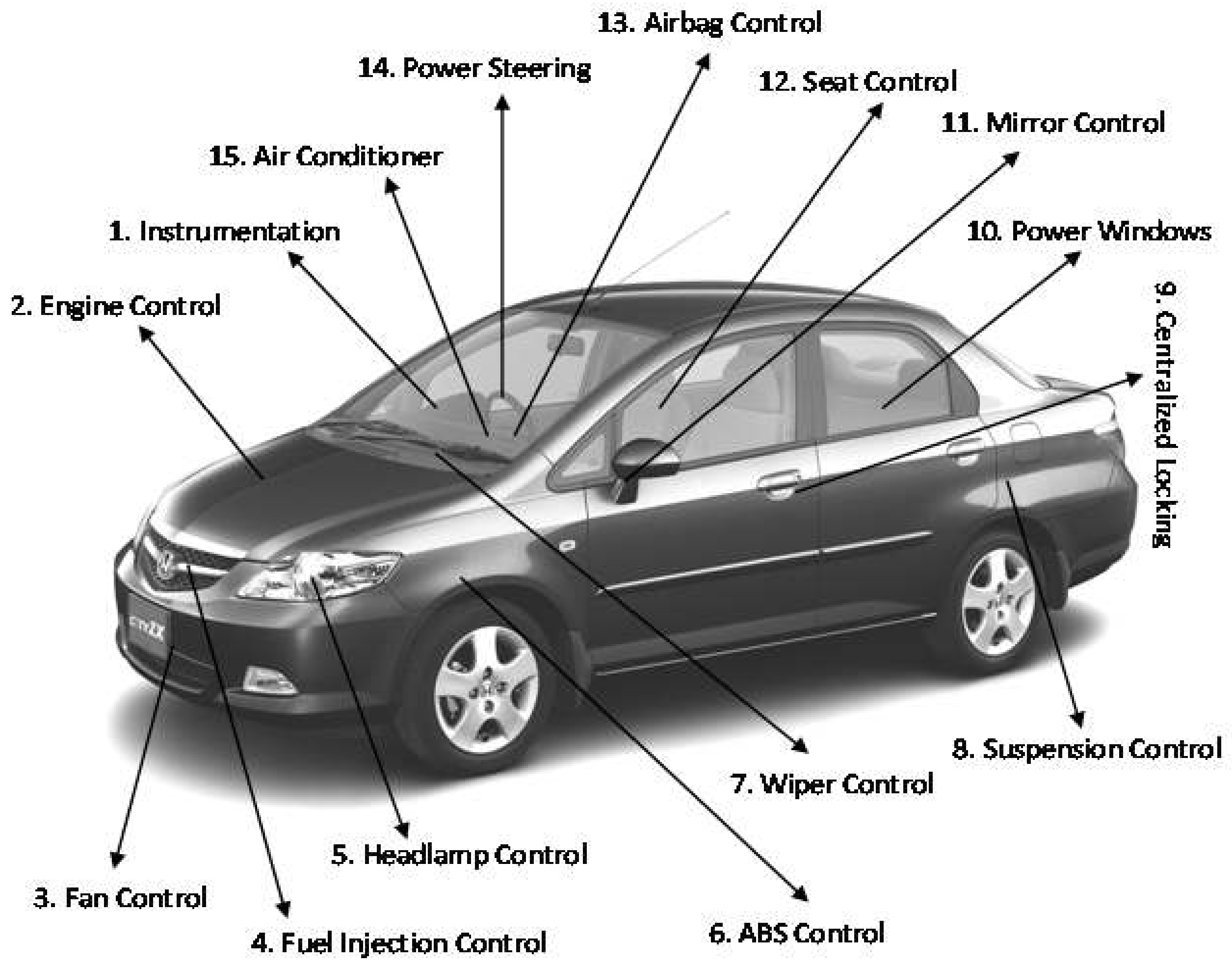


Photo Courtesy of Honda Siel Car India (www.hondacarindia.com)

Automotive Embedded Systems

- ✓ Generally built around Digital Signal Processors, Application Specific Instruction Set Processors (like Atmel Automotive AVR) and General purpose processors/Controllers, System on Chips, Programmable Logic Devices or Application Specific Integrated/Standard Products (ASIC/ASSP) or a combination of these.
- ✓ Automotive Embedded Control units are generally known as Electronic Control Units (ECUs)
- ✓ The presence of ECUs vary from simple mirror control units to complex Airbag deployment and Antilock braking Systems (ABS)
- ✓ The number of embedded controllers in an ordinary vehicle varies from 20 to 40 whereas a luxury vehicle may contain 100s embedded control units
- ✓ The first embedded system used in automotive application was the microprocessor based fuel injection system introduced by Volkswagen 1600 in 1968

Automotive Embedded Systems

High speed Electronic Control Units (HECUs)

High speed Electronic Control Units (HECUs) are deployed in critical control units requiring fast response. They include Fuel Injection Systems, Antilock Brake Systems, Engine Control, Electronic throttle, Steering Controls, Transmission Control unit and Central Control unit

Low speed Electronic Control Units (LECUs)

Low Speed Electronic Control Units (LECUs) are deployed in applications where response time is not so critical. They generally are built around low cost microprocessors/microcontrollers and Digital Signal Processors. Audio controllers, Passenger and driver door locks, Door glass controls (Power Windows), Wiper control, Mirror Control, Seat control systems, Head lamp and tail lamp controls, Sun roof control unit etc. are examples of LECUs

Automotive Communication Buses

Controller Area Network (CAN)

- ✓ Originally proposed by Robert Bosch, pioneers in the Automotive Embedded Solution providers
- ✓ Supports medium speed (ISO11519-Class B with data rates up to 125 Kbps) and high speed (ISO11898 Class C with data rates up to 1Mbps) data transfer.
- ✓ An event driven protocol interface with support for error handling in data transmission
- ✓ Generally employed in safety system like Airbag control; powertrain systems like Engine control and Antilock Brake Systems (ABS); and navigation systems like GPS

Local Interconnect Network (LIN)

- ✓ A single master multiple slave (up to 16 independent slave nodes) communication interface
- ✓ LIN is a low speed, single wire communication interface with support for data rates up to 20Kbps
- ✓ Mainly used for sensor/actuator interfacing
- ✓ Follows the Master communication triggering technique to eliminate the possible bus arbitration problem that can occur by the simultaneous talking of different slave nodes connected to a single interface bus
- ✓ LIN bus is employed in applications like mirror controls, fan controls, seat positioning controls, window controls, and position controls where response time is not a critical issue

Media Oriented System Transport (MOST) bus

- ✓ Targeted for automotive Audio Video equipment interfacing
- ✓ A multimedia fiber-optic point-to-point network implemented in a star, ring or daisy-chained topology over optical fibers cables
- ✓ The MOST bus specifications define the Physical (Electrical and Optical parameters) layer as well as the Application Layer, Network Layer, and Media Access Control
- ✓ MOST bus is an optical fiber cable connected between the Electrical Optical Converter (EOC) and Optical Electrical Converter (OEC), which would translate into the optical cable MOST bus

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, and 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 and once the hardware and software are ready, the integration is performed

Hardware Software Co-design Approach for Embedded System Development

- ✓ The product requirements captured from the customer are converted into system level needs or processing requirements rather than partitioning them to either h/w or s/w
- ✓ 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 this 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

Fundamental issues in H/w S/w Co-design

- **Model Selection**
  - ✓ A Model captures and describes 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 objectives vary with each phase
- **Architecture Selection**
  - ✓ 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
  - ✓ 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 are the commonly used architectures in system design

Fundamental issues in H/w S/w Co-design

- ❑ **Language Selection**
  - ✓ A programming Language captures a 'Computational Model' and maps it into architecture
  - ✓ 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
  - ✓ 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

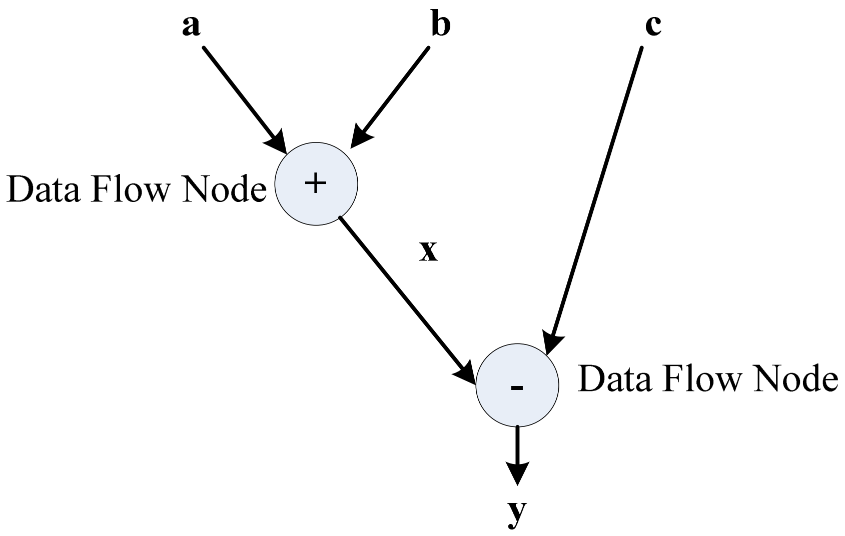
- ❑ **Partitioning of System Requirements into H/w and S/w**
  - ✓ Implementation aspect of a System level Requirement
  - ✓ It may be possible to implement the system requirements in either hardware or software (firmware)
  - ✓ Various hardware software trade-offs like performance, re-usability, effort etc are used for making a decision on the hardware-software partitioning

Computational Models in Embedded Design

- **Data Flow Graph/Diagram (DFG) Model**
  - ✓ Translates the data processing requirements into a data flow graph
  - ✓ A data driven model in which the program execution is determined by data.
  - ✓ Emphasizes on the data and operations on the data which transforms the input data to output data.
  - ✓ 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
  - ✓ Best suited for modeling Embedded systems which are computational intensive (like DSP applications)

Computational Models in Embedded Design – Data Flow Graph/Diagram (DFG) Model

E.g. Model the requirement x = a + b; and y = x - c;



Data path: The data flow path from input to output

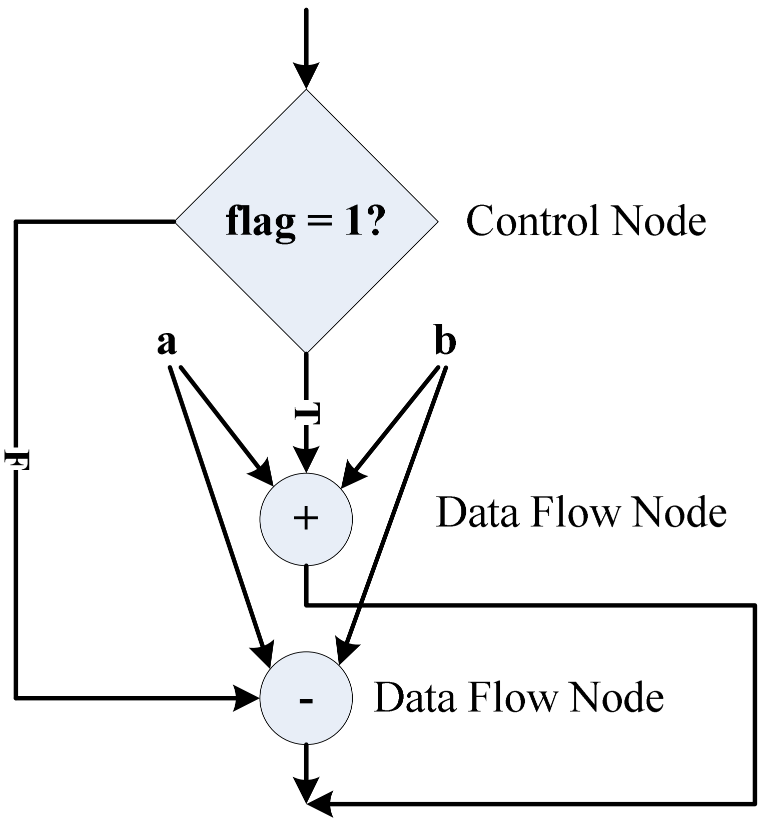
A DFG model is said to be acyclic DFG (ADFG) if it doesn’t contain multiple values for the input variable and multiple output values for a given set of input(s). Feedback inputs (Output is feed back to Input), events etc are examples for non-acyclic inputs. A DFG model translates the program as a single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

Computational Models in Embedded Design

- **Control Data Flow Graph/Diagram (CDFG) Model**
  - ✓ Translates the data processing requirements into a data flow graph
  - ✓ Model applications involving conditional program execution
  - ✓ Contains both data operations and control operations
  - ✓ 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
  - ✓ 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.
  - ✓ The control node is represented by a 'Diamond' block which is the decision making element in a normal flow chart based design
  - ✓ Translates the requirement, which is modeled to a concurrent process model
  - ✓ The decision on which process is to be executed is determined by the control node
  - ✓ Capturing of image and storing it in the format selected (bmp, jpg, tiff, etc.) in a digital camera is a typical example of an application that can be modeled with CDFG

Computational Models in Embedded Design – Control Data Flow Graph/Diagram (CDFG) Model

E.g. Model the requirement   If flag = 1, x = a + b; else y = a-b;



Computational Models in Embedded Design

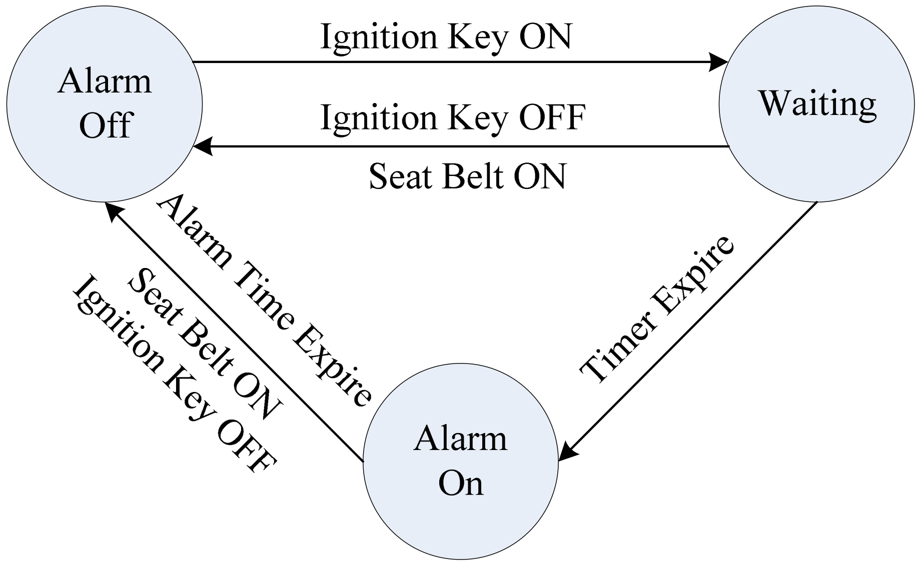
- **State Machine Model**
  - ✓ Based on ‘States’ and ‘State Transition’
  - ✓ Describes the system behavior with ‘States’, ‘Events’, ‘Actions’ and ‘Transitions’
  - ✓ *State* is a representation of a current situation.
  - ✓ An *event* is an input to the *state*. The *event* acts as stimuli for state transition.
  - ✓ *Transition* is the movement from one state to another.
  - ✓ *Action* is an activity to be performed by the state machine.
  - ✓ A Finite State Machine (FSM) Model is one in which the number of states are finite. In other words the system is described using a finite number of possible states
  - ✓ Most of the time State Machine model translates the requirements into sequence driven program
  - ✓ The Hierarchical/Concurrent Finite State Machine Model (HCFSM) is an extension of the FSM for supporting concurrency and hierarchy
  - ✓ HCFSM extends the conventional state diagrams by the AND, OR decomposition of States together with inter level transitions and a broadcast mechanism for communicating between concurrent processes
  - ✓ HCFSM uses statecharts for capturing the states, transitions, events and actions. The Harel Statechart, UML State diagram etc are examples for popular statecharts used for the HCFSM modeling of embedded systems

Computational Models in Embedded Design – Finite State Machine (FSM) Model

E.g. Automatic ‘Seat Belt Warning’ in an automotive

<u>Requirement:</u>

- 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.



Computational Models in Embedded Design

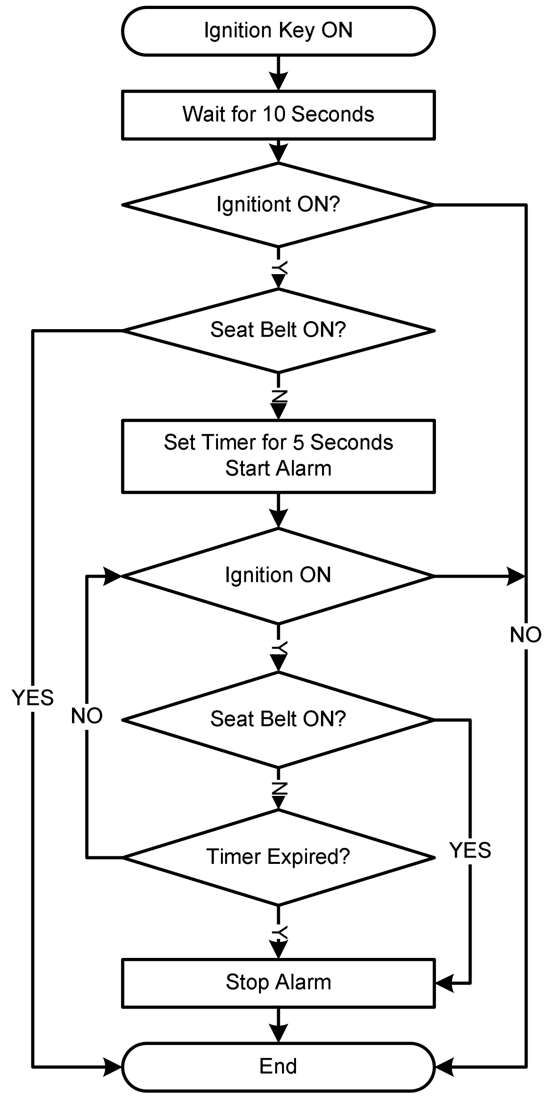
- **Sequential Program Model**
  - ✓ The functions or processing requirements are executed in sequence
  - ✓ The program instructions are iterated and executed conditionally and the data gets transformed through a series of operations
  - ✓ FSMs are good choice for sequential Program modeling.
  - ✓ Flow Charts is another important tool used for modeling sequential program
  - ✓ The FSM approach represents the states, events, transitions and actions, whereas the Flow Chart models the execution flow

Computational Models in Embedded Design –Sequential Program Model

E.g. Automatic 'Seat Belt Warning' in an automotive

Requirement:

- 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.



Computational Models in Embedded Design –Sequential Program Model

E.g. Automatic ‘Seat Belt Warning’ in an automotive

```
#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()==OFF )&& (timer_expire()==NO));
stop_alarm();
}
}
}
```

Computational Models in Embedded Design

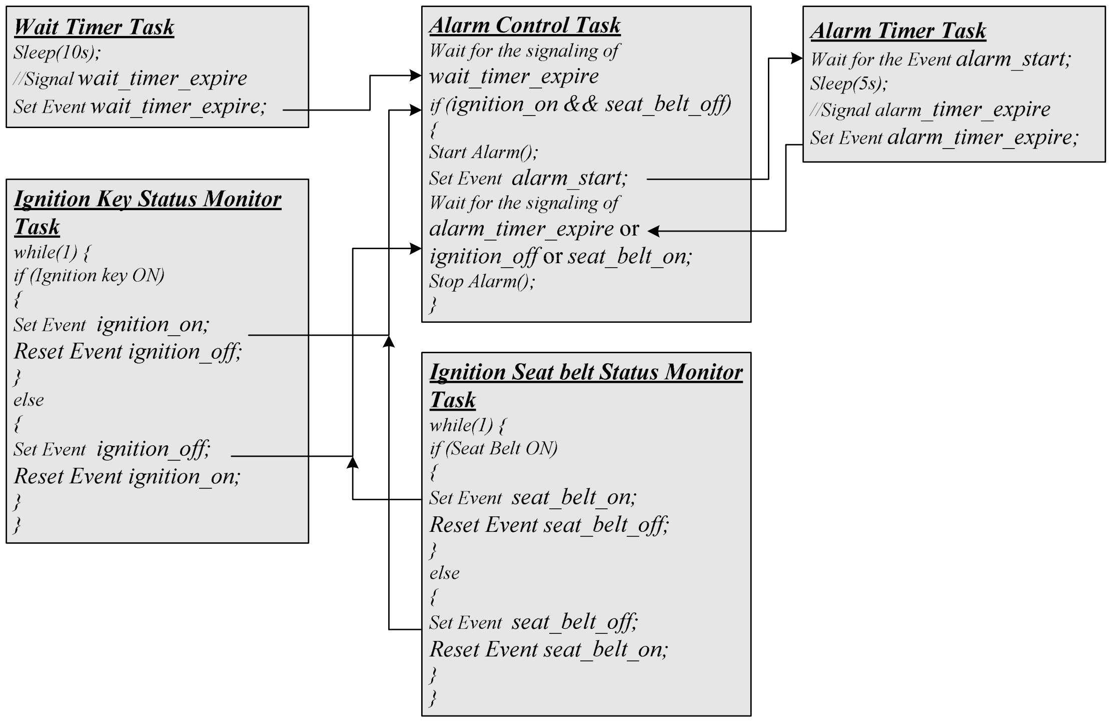
- **Concurrent/Communicating Process Model**
  - ✓ Models concurrently executing tasks/processes The program instructions are iterated and executed conditionally and the data gets transformed through a series of operations
  - ✓ Certain processing requirements are easier to model in concurrent processing model than the conventional sequential execution.
  - ✓ Sequential execution leads to a single sequential execution of task and thereby leads to poor processor utilization, 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, when the subtask under execution goes to a wait or sleep mode, by switching the task execution.
  - ✓ Concurrent processing model requires additional overheads in task scheduling, task synchronization and communication

Computational Models in Embedded Design – Concurrent Processing Model

E.g. Automatic 'Seat Belt Warning' in an automotive

```
Create and initialize events
wait_timer_expire, ignition_on, ignition_off,
seat_belt_on, seat_belt_off,
alarm_timer_start, alarm_timer_expire
Create task Wait Timer
Create task Ignition Key Status Monitor
Create task Seat Belt Status Monitor
Create task Alarm Control
Create task Alarm Timer
```

(a)



Wait Timer Task
```
Sleep(10s);
//Signal wait_timer_expire
Set Event wait_timer_expire;
```

Ignition Key Status Monitor Task
```
while(1) {
if (Ignition key ON)
{
Set Event ignition_on;
Reset Event ignition_off;
}
else
{
Set Event ignition_off;
Reset Event ignition_on;
}
}
```

Alarm Control Task
```
Wait for the signaling of
wait_timer_expire
if (ignition_on && seat_belt_off)
{
Start Alarm();
Set Event alarm_start;
Wait for the signaling of
alarm_timer_expire or
ignition_off or seat_belt_on;
Stop Alarm();
}
```

Alarm Timer Task
```
Wait for the Event alarm_start;
Sleep(5s);
//Signal alarm_timer_expire
Set Event alarm_timer_expire;
```

Ignition Seat belt Status Monitor Task
```
while(1) {
if (Seat Belt ON)
{
Set Event seat_belt_on;
Reset Event seat_belt_off;
}
else
{
Set Event seat_belt_off;
Reset Event seat_belt_on;
}
}
```

(b)

- ✓ The processing requirements are split in to multiple tasks
- ✓ Tasks are executed concurrently
- ✓ 'Events' are used for synchronizing the execution of tasks

- ✓ 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 usually stored in a permanent memory (ROM) and it is non alterable by end users
- ✓ Designing Embedded firmware requires understanding of the particular embedded product hardware, like various component interfacing, memory map details, I/O port details, configuration and register details of various hardware chips used and some programming language (either low level Assembly Language or High level language like C/C++ or a combination of the two)
- ✓ The embedded firmware development process starts with the conversion of the firmware requirements into a program model using various modeling tools
- ✓ 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

- ✓ Suitable for applications that are not time critical and where the response time is not so important (Embedded systems where missing deadlines are acceptable)
- ✓ 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

Embedded firmware Design Approaches – The Super loop

Pros:

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

Cons:

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

Enhancements:

- ✓ Combine Super loop based technique with interrupts
- ✓ Execute the tasks (like keyboard handling) which require Real time attention as Interrupt Service routines

```
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)
- ✓ The Embedded OS is responsible for scheduling the execution of user tasks and the allocation of system resources among multiple tasks
- ✓ Involves lot of OS related overheads apart from managing and executing user defined tasks
- ✓ Microsoft® Windows XP Embedded is an example of GPOS for embedded 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/Options

- ✓Assembly Language
- ✓High Level Language
    - ✓Subset of C (Embedded C)
    - ✓Subset of C++ (Embedded C++)
    - ✓Any other high level language with supported Cross-compiler
- ✓Mix of Assembly & High level Language
    - ✓Mixing High Level Language (Like C) with Assembly Code
    - ✓Mixing Assembly code with High Level Language (Like C)
    - ✓Inline Assembly

Embedded firmware Development Languages/Options – Assembly Language

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

Embedded firmware Development Languages/Options – Assembly Language

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

```
01110100        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.

- ✓ Assembly language instructions are written one per line
- ✓ 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).

Embedded firmware Development Languages/Options – Assembly Language

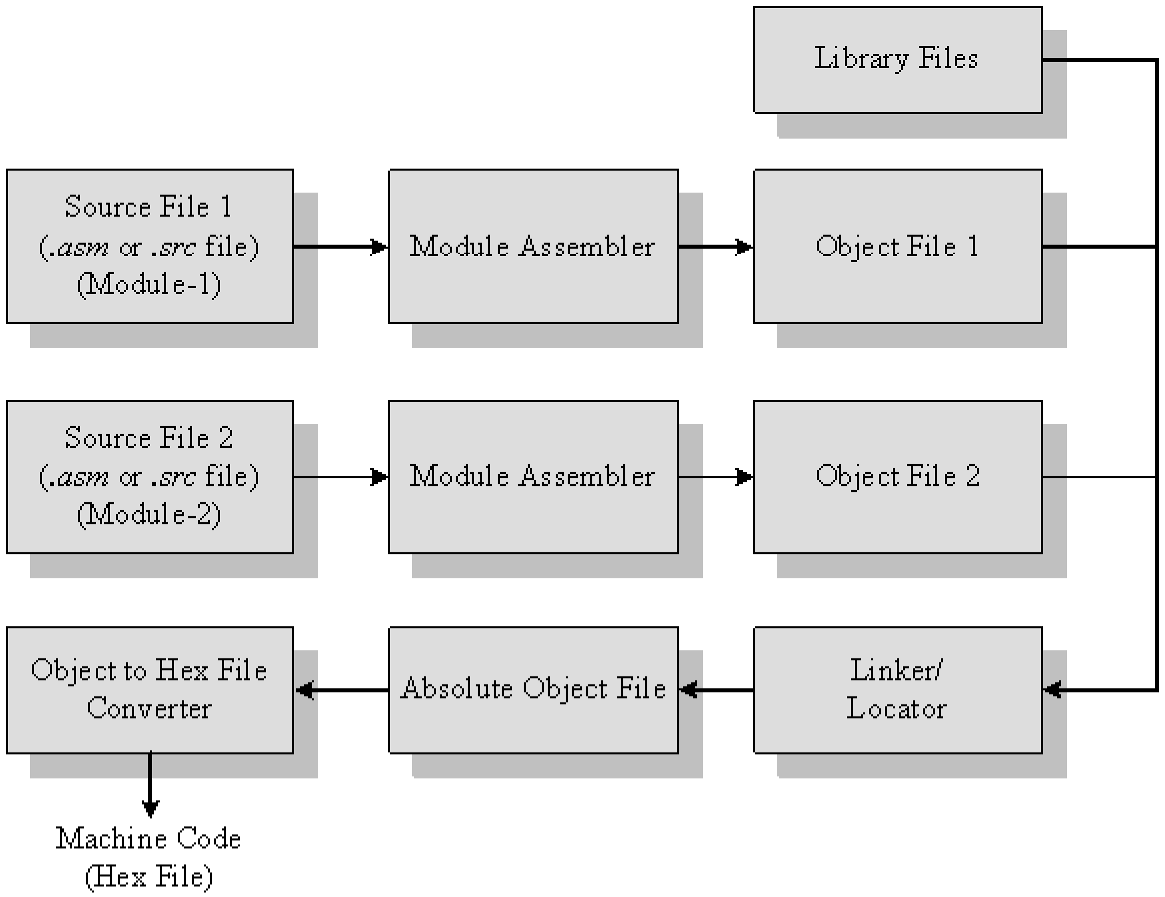
```
;####################################################################
;               SUBROUTINE FOR GENERATING DELAY
;               DELAY PARAMETR PASSED THROUGH REGISTER R1
;               RETURN VALUE NONE
;               REGISTERS USED: R0,  R1
;####################################################################
                DELAY:    MOV     R0, #255  ; Load Register R0 with 255
                          DJNZ    R1, DELAY            ; Decrement R1 and loop till   R1= 0
                          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

Embedded firmware Development Languages/Options – Assembly Language – Source File to Hex File Translation

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

Embedded firmware Development Languages/Options – Assembly Language – Source File to Hex File Translation



Assembly language to machine language Conversion process

Embedded firmware Development Languages/Options – Assembly Language – Source File to Hex File Translation

Advantages:

✓Efficient Code Memory & Data Memory Usage (Memory Optimization)
✓High Performance
✓Low level Hardware Access
✓Code Reverse Engineering

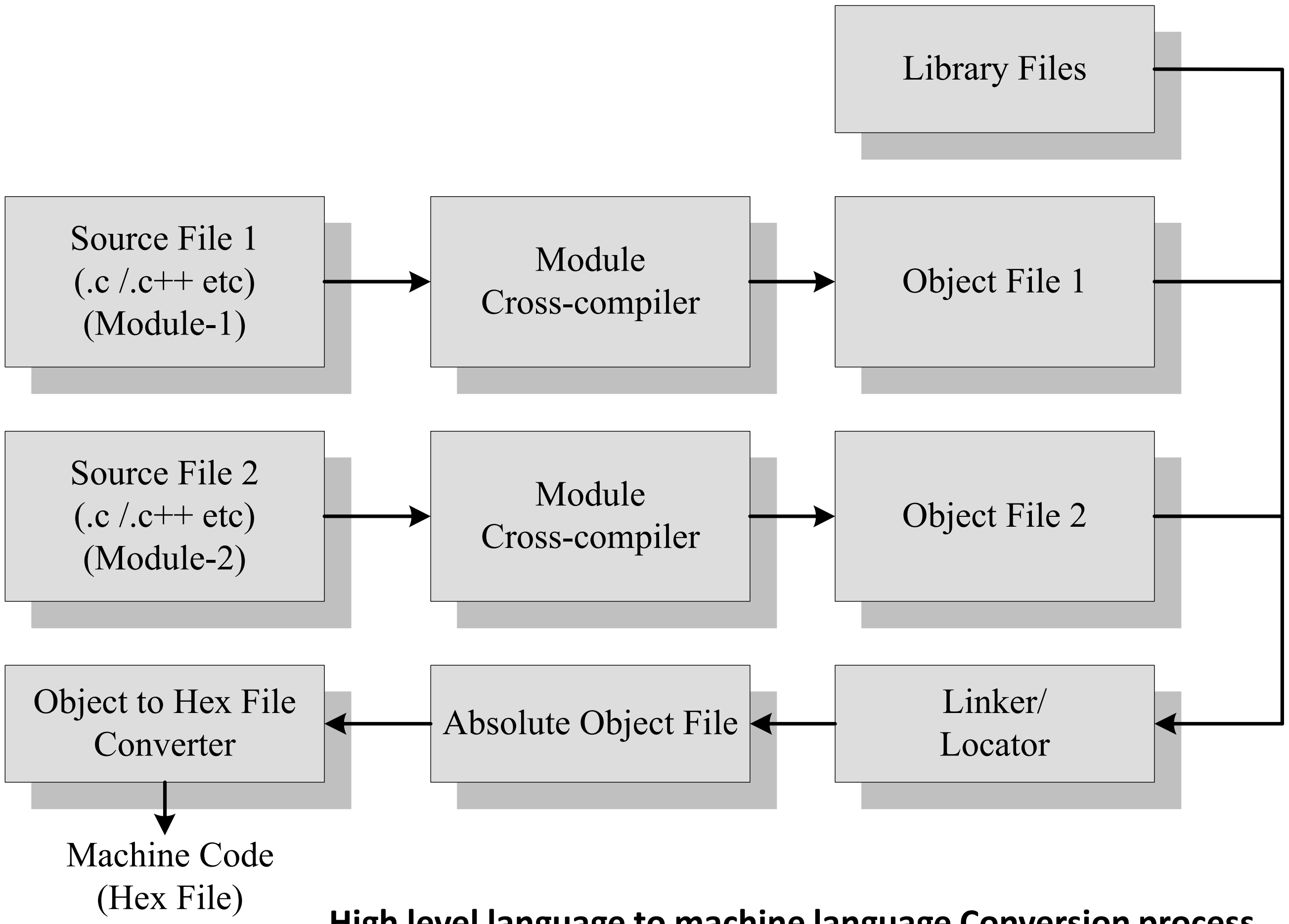
Drawbacks:

✓High Development time
✓Developer dependency
✓Non portable

Embedded firmware Development Languages/Options – High Level Language

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

Embedded firmware Development Languages/Options – High Level Language – Source File to Hex File Translation



High level language to machine language Conversion process

Embedded firmware Development Languages/Options – High Levely Language – Source File to Hex File Translation

Advantages:

✓Reduced Development time
✓Developer independency
✓Portability

Embedded firmware Development Languages/Options – Mixing of Assembly Language with High Level Language

- ✓ Certain situations in Embedded firmware development may require the mixing of Assembly Language with high level language or vice versa. Interrupt handling, Source code is already available in high level language\Assembly Language etc are examples
- ✓ High Level language and low level language can be mixed in three different ways
  - ✓ Mixing Assembly Language with High level language like ‘C’
  - ✓ Mixing High level language like ‘C’ with Assembly Language
  - ✓ In line Assembly
- ✓ The passing of parameters and return values between the high level and low level language is cross-compiler specific

High Level Language – ‘C’ V/s Embedded C

- ✓ ‘*C*’ is a well structured, well defined and standardized general purpose programming language with extensive bit manipulation support
- ✓ ‘*C*’ offers a combination of the features of high level language and assembly and helps in hardware access programming (system level programming) as well as business package developments (Application developments like pay roll systems, banking applications etc)
- ✓ The conventional ‘*C*’ language follows ANSI standard and it incorporates various library files for different operating systems
- ✓ A platform (Operating System) specific application, known as, compiler is used for the conversion of programs written in ‘*C*’ to the target processor (on which the OS is running) specific binary files
- ✓ Embedded C can be considered as a subset of conventional ‘C’ language
- ✓ Embedded C supports all ‘C’ instructions and incorporates a few target processor specific functions/instructions
- ✓ The standard ANSI ‘C’ library implementation is always tailored to the target processor/controller library files in Embedded C
- ✓ The implementation of target processor/controller specific functions/instructions depends upon the processor/controller as well as the supported cross-compiler for the particular Embedded C language
- ✓ A software program called ‘Cross-compiler’ is used for the conversion of programs written in Embedded C to target processor/controller specific instructions

High Level Language Based Development – ‘Compiler’ V/s ‘Cross-Compiler’

- ✓ Compiler is a software tool that converts a source code written in a high level language on top of a particular operating system running on a specific target processor architecture (E.g. Intel x86/Pentium).
- ✓ The operating system, the compiler program and the application making use of the source code run on the same target processor.
- ✓ The source code is converted to the target processor specific machine instructions
- ✓ A native compiler generates machine code for the same machine (processor) on which it is running.
- ✓ Cross compiler is the software tools used in cross-platform development applications
- ✓ In cross-platform development, the compiler running on a particular target processor/OS converts the source code to machine code for a target processor whose architecture and instruction set is different from the processor on which the compiler is running or for an operating system which is different from the current development environment OS
- ✓ Embedded system development is a typical example for cross-platform development where embedded firmware is developed on a machine with Intel/AMD or any other target processors and the same is converted into machine code for any other target processor architecture (E.g. 8051, PIC, ARM9 etc).
- ✓ Keil C51compiler from Keil software is an example for cross-compiler for 8051 family architecture