MODULE – 3

EMBEDDED SYSTEM COMPONENTS

INTRODUCTION TO EMBEDDED SYSTEMS

An ***embedded system*** is an electronic/ electro-mechanical system designed to perform a specific function and is a combination of both hardware and firmware (software).

Every embedded system is unique, and the hardware as well as the firmware is highly specialized to the application domain. Embedded systems are becoming an inevitable part of any product or equipment in all fields including household appliances, telecommunications, medical equipment, industrial control, consumer products, etc.

Characteristics of Embedded Systems:

- ✓ Embedded Systems must be highly reliable and stable.
- ✓ Embedded Systems have minimal or no user interface.
- ✓ Embedded Systems are usually feedback oriented or reactive.
- ✓ Embedded Systems are typically designed to meet real time constraints.
- ✓ Embedded Systems have limited memory and limited number of peripherals.
- ✓ Embedded Systems are typically designed for specific application or purpose.
- ✓ Embedded Systems are designed for low power consumption, as they use battery power.

EMBEDDED SYSTEMS versus GENERAL COMPUTING SYSTEMS:

General Computing System	Embedded System
1. A combination of generic hardware and a General Purpose Operating System (GPOS) for executing a variety of applications.	1. A combination of special purpose hardware embedded OS for executing a specific set of applications.
2. Applications are alterable (programmable) by the user.	2. The firmware is pre-programmed and it is non-alterable by the end-user (there may be exceptions).
3. Performance is the key deciding factor in the selection of the system. Always, 'Faster is Better'.	3. Application-specific requirements (like performance, power requirements, memory usage, etc.).
4. Less/ not at all tailored towards reduced operating power requirements.	4. Highly tailored to take advantage of the power saving modes supported by the hardware and the operating system.
5. Need not be deterministic in execution behavior; response requirements are not time critical.	5. Execution behavior is deterministic for certain types of embedded systems like 'Hard Real Time' systems.

HISTORY OF EMBEDDED SYSTEMS:

- Embedded systems were existing even before the Information Technology revolution. Initially, embedded systems were built around vacuum tube and transistor technologies; and embedded algorithm was developed by using low level programming languages.
    - Advances in semiconductor and nano-technology and IT revolution gave way to development of miniature embedded systems.

- ***Apollo Guidance Computer*** (AGC) developed (during 1960) by MIT Instrumentation Laboratory for the lunar expedition is the first recognized modern embedded system.
    - AGC included both Command Module (CM-to encircle the moon) and Lunar Excursion Module (LEM-to go down to the moon surface and land there safely).
    - There were 16 reaction control thrusters, a descent engine (designed to provide thrust to the lunar model out of the lunar orbit and land it safely on moon) and an ascent engine.
    - Original design was based on 4K words of fixed memory (ROM) and 256 words of erasable memory (RAM); which has been enhanced (during 1963) to 10K fixed and 1K erasable memory. The clock frequency was 1.024 MHz.
    - The computing unit of AGC consisted of approximately 11 instructions on 16-bit word logic.
    - A calculator type user interface was given and is known as DSKY (display/ keyboard).

- The first mass-produced embedded system was the guidance computer, ***Autonetics D-17***, for the Minuteman-I missile in 1961; built using discrete transistor logic and a hard-disk for main memory.

- The first microprocessor, the Intel 4004, was designed for calculators and other small systems; but still required many external memory and support chips.
- First microcontroller, TMS 1000, developed in 1974 by Texas Instruments. It had ROM, RAM, and clock circuitry on the chip along with the processing chip.
- In 1980, Intel introduced 8051 MCU and called it MCS-51 architecture.

- Laser and Inkjet printers emerged during 1980s; and early 1990, cell phones having five or six DSPs and CPUs emerged.

CLASSIFICATION OF EMBEDDED SYSTEMS:

Classification Based on Generation:

Generation with Example	Description
First Generation (1G) - Digital telephone keypads - Stepper motor	✓ 8-bit microprocessor and 4-bit microcontroller like 8085 and Z80 was used in 1G. ✓ Hardware circuit was simple. ✓ Assembly code is used for developing firmware.
Second Generation (2G) - Data acquisition systems like ADC, SCADA system	✓ Uses 16-bit microprocessor and 8-bit microcontroller. ✓ They are more complex and powerful than 1G microprocessor and microcontroller.
Third Generation (3G) - Robotics	✓ Uses 32-bit microprocessor and 16-bit microcontroller. ✓ Domain specific processor and controllers are used.
Fourth Generation (4G) - Smart phones	✓ Uses 64-bit microprocessor and 32-bit microcontroller. ✓ The concept of system on chips, multi-core processors evolved. ✓ Highly complex and very powerful.

Classification Based on Complexity and Performance:

1. ***Small-Scale Embedded Systems:***
    - Embedded systems which are simple in application needs and the performance parameters are not time critical (E.g.: Electronic toy).
    - Small-scale embedded systems are usually built around low performance and low cost 8 or 16 bit microprocessors/ microcontrollers.
    - It may or may not contain an operating system for its functioning.
2. ***Medium-Scale Embedded Systems:***
    - Embedded systems which are slightly complex in hardware and firmware (software) requirements.
    - Medium-scale embedded systems are usually built around medium performance, low cost 16 or 32 bit microprocessors/ microcontrollers or digital signal processors.
    - They usually contain an embedded operating system (general purpose/ real-time).
3. ***Large-Scale Embedded Systems/ Complex Systems:***
    - Embedded systems which involve highly complex hardware and firmware. They are employed in mission critical applications demanding high performance.
    - Large-scale embedded systems are commonly built around high performance 32 or 64 bit RISC processors/ controllers or Reconfigurable System on Chip (RSoC) or multi-core processors and programmable logic devices.

- They usually contain a high performance Real Time Operating System (RTOS) for task scheduling, prioritization, and management.

MAJOR APPLICATION AREAS OF EMBEDDED SYSTEMS:

Embedded systems play a vital role in our day-to-day life, starting from home to computer industry. Embedded technology has acquired a new dimension from its first generation model, the Apollo Guidance Computer, to the latest radio navigation system combined with in-car entertainment technology and wearable computing devices (Apple watch, Microsoft band, Fitbit fitness trackers, etc.).

The application areas and the products in the embedded domain are countless. A few of the important domains and products are listed below:

1. *Consumer Electronics:* Camcorders, cameras, etc.
2. *Household Appliances:* Television, DVD players, washing machine, fridge, microwave oven, etc.
3. *Home Automation and Security Systems:* Air conditioners, sprinklers, intruder detection alarms, closed circuit television cameras, fire alarms, etc.
4. *Automotive Industry:* Anti-lock breaking systems (ABS), engine control, ignition systems, automatic navigation systems, etc.
5. *Telecom:* Cellular telephones, telephone switches, handset multimedia applications, etc.
6. *Computer Peripherals:* Printers, scanners, fax machines, etc.
7. *Computer Networking Systems:* Network routers, switches, hubs, firewalls, etc.
8. *Healthcare:* Different kinds of scanners, EEG, ECG machines, etc.
9. *Measurement & Instrumentation:* Digital multi meters, digital CROs, logic analyzers PLC systems, etc.
10. *Banking & Retail:* Automatic teller machines (ATM) and currency counters, point of sales (POS).
11. *Card Readers:* Barcode, smart card readers, hand held devices, etc.
12. *Wearable Devices:* Health and fitness trackers, Smartphone screen extension for notifications, etc.
13. Cloud Computing and Internet of Things (IoT).

PURPOSE OF EMBEDDED SYSTEMS:

As mentioned in the previous section, embedded systems are used in various domains like consumer electronics, home automation, telecommunications, automotive industry, healthcare, control & instrumentation, retail and banking applications, etc. Each embedded system is designed to serve the purpose of any one or a combination o the following tasks:

1. ***Data Collection, Storage, Representation***
    - Data is collection of facts, such as values or measurements. It can be numbers, words, measurements, observations, or even just description of things.

- Purpose of embedded system design is data collection. It performs acquisition of data from the external world.
- Data collection is usually done for storage, analysis, manipulation, and transmission.
- Data can be analog or digital.
- Embedded systems with analog data capturing techniques collect data directly in the form of analog signal; whereas embedded systems with digital data collection mechanism convert the analog signal to corresponding digital signal using analog to digital (A/D) converters.
- If the data is digital, it can be directly captured by digital embedded system.
    - A digital camera is a typical example of an embedded system with data collection, storage, and representation of data. Images are captured and captured image may be stored within the memory of the camera. The captured image can also be presented to the user through a graphic LCD (Liquid Crystal Display) unit.

2. Data Communication

- Embedded data communication systems are deployed in applications ranging from simple home networking systems to complex satellite communication systems.
    - Network hubs, routers, switches are examples of dedicated data transmission embedded systems.
- Data transmission is in the form of wire medium or wireless medium. Initially wired medium is used by embedded systems; and as technology changes, wireless medium becomes de-facto standard in embedded systems.
    - USB, TCP/ IP are examples of wired communication; and BlueTooth, ZigBee and Wi-Fi are examples for wireless communication.
- Data can be transmitted by analog means or by digital means.

3. Data (Signal) Processing

- Embedded systems with signal processing functionalities are employed in applications demanding signal processing like speech coding, audio-video codec, transmission applications, etc.
    - A digital hearing aid is a typical example of an embedded system employing data processing.

4. Monitoring

- Almost all embedded products coming under the medical domain are with monitoring functions.
    - Patient heart beat is monitored by Electro cardiogram (ECG) machine.

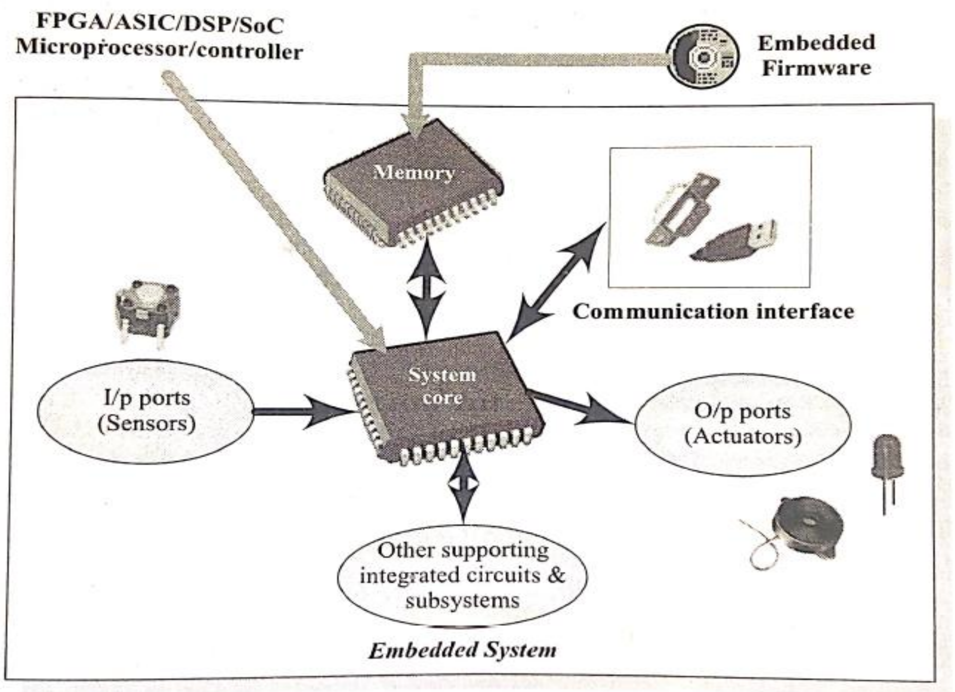
- Digital CRO, digital multi-meters, and logic analyzers are examples of monitoring embedded systems.

5. ***Control***
- Sensors and actuators are used for controlling the system.
    - Sensors are connected to the input port for capturing the changes in environmental variable or measuring variable.
    - Actuators connected to output port are controlled according to the changes in input variable.
- Air conditioner system used in our home to control the room temperature to a specified limit is a typical example for embedded system for control purpose. The air conditioner's compressor unit (actuator) is controlled according to the current room temperature (sensor) and the desired room temperature set by the user.

6. ***Application Specific User Interface***
- These are embedded systems with application-specific user interfaces like buttons, switches, keypad, lights, bells, display units, etc.
- Mobile phone is an example for this. In mobile phone, the user interface is provided through the keypad, graphic LCD module, system speaker, vibration alert, etc.

THE TYPICAL EMBEDDED SYSTEM

A typical embedded system (shown in the following Figure) contains a single chip controller, which acts as the master brain of the system.



MODULE – 4

EMBEDDED SYSTEM DESIGN COMPONENTS

CHARACTERISTICS AND QUALITY ATTRIBUTES OF EMBEDDED SYSTEMS

No matter whether it is an embedded or a non-embedded system, there will be a set of *characteristics* describing the system. The non-functional aspects that need to be addressed in embedded system design are commonly referred as *quality attributes*. Whenever you design an embedded system, the design should take into consideration of both the functional and non-functional aspects.

CHARACTERISTICS OF AN EMBEDDED SYSTEM:

Unlike general purpose computing systems, embedded systems possess certain specific characteristics and these characteristics are unique to each embedded system.

Some of the important characteristics of an embedded system are:

1. Application and domain specific
2. Reactive and Real Time
3. Operates in harsh environments
4. Distributed
5. Small size and weight
6. Power concerns

Application and Domain Specific:

- If you closely observe any embedded system, you will find that each embedded system is having certain functions to perform.
- Embedded systems are developed in such a manner to do only intended functions. They cannot be used for any other purpose. It is the major criterion which distinguishes an embedded system from a general purpose system.
  - For example, you cannot replace the embedded control unit of your microwave oven with your air conditioners embedded control unit, because the embedded control units of microwave oven and air conditioner are specifically designed to perform certain specific tasks.
- Also you cannot replace an embedded control unit developed for a particular domain say telecom with another control unit designed to serve another domain like consumer electronics.

Reactive and Real Time:

- Embedded systems are in constant interaction with the Real world through sensors and user defined input devices which are connected to the input port of the system.
    - Any changes happening in the Real world (which is called an *Event*) are captured by the sensors or input devices in Real Time and the control algorithm running inside the unit reacts in a designed manner to bring the controlled output variables to the desired level.
- The event may be a periodic one or an unpredicted one. If the event is an unpredicted one, then such system should be designed in such a way that it should be scheduled to capture the events without missing them.
- Embedded systems produce changes in output in response to the changes in the input. So they are generally referred as *Reactive Systems*.
- *Real Time System* operation means the timing behavior of the system should be deterministic; meaning the system should respond to requests or tasks in a know amount of time.
- A Real Time system should not miss any deadlines for tasks or operations.
- It is not necessary that all embedded systems should be Real Time in operations.
    - Embedded applications or systems which are mission critical, like flight control systems, Antilock Brake Systems (ABS), etc. are examples of Real Time systems.
- The design of an embedded Real Time system should take the worst case scenario into consideration.

Operates in Harsh Environment:

- It is not necessary that all embedded systems should be deployed in controlled environments.
- The environment in which the embedded system deployed may be a dusty one or a high temperature zone or an area subject to vibrations and shock. Systems placed in such areas should be capable to withstand all these adverse operating conditions. The design should take care of the operating conditions of the area where the system is going to implement.
    - For example, if the system needs to be deployed in a high temperature zone, then all the components used in the system should be of high temperature grade.
- Here we cannot go for a compromise in cost. Also proper shock absorption techniques should be provided to systems which are going to be commissioned in places subject to high shock.
- Power supply fluctuations, corrosion and component aging, etc. are the other factors that need to be taken into consideration for embedded systems to work in harsh environments.

Distributed:

- The term *distributed* means that, embedded systems may be a part of larger systems.
- Many numbers of distributed embedded systems form a single large embedded control unit.
    - An automatic vending machine is a typical example for this. The vending machine contains a card reader (for pre-paid vending systems), a vending unit, etc. Each of them are independent embedded units but they work together to perform the overall vending function.
    - Another example is the Automatic Teller Machine (ATM). An ATM contains a card reader embedded unit, responsible for reading and validating the user's AIM card, transaction unit for performing transactions, a currency counter for dispatching/ vending currency to the authorized person and a printer unit for printing the transaction details. We can visualize these as independent embedded systems. But they work together to achieve a common goal.
    - Another typical example of a distributed embedded system is the Supervisory Control And Data Acquisition (SCADA) system used in Control & Instrumentation applications, which contains physically distributed individual embedded control units connected to a supervisory module.

Small Size and Weight:

- Product aesthetics is an important factor in choosing a product.
    - For example, when you plan to buy a new mobile phone, you may make a comparative study on the pros and cons of the products available in the market. Definitely the product aesthetics (size, weight, shape, style, etc. will be one of the deciding factors to choose a product.
- People believe in the phrase "Small is beautiful". Moreover it is convenient to handle a compact device than a bulky product.
- In embedded domain also compactness is a significant deciding factor. Most of the application demands small size and low weight products.

Power Concerns:

- Power management is another important factor that needs to be considered in designing embedded systems.
- Embedded systems should be designed in such a way as to minimize the heat dissipation by the system.

- The production of high amount of heat demands cooling requirements like cooling fans which in turn occupies additional space and make a system bulky.
- Nowadays ultra low power components are available in the market. Select the design according to the low power components like low dropout regulators, and controllers/ processors with power saving modes.
- Also power management is a critical constraint in battery operated applications. The more the power consumption the less is the battery life.

QUALITY ATTRIBUTES OF AN EMBEDDED SYSTEM:

Quality attribute are non-functional requirements that need to be documented properly in any system design. If the quality attributes are more concrete and measurable, it will give a positive impact on the system development process and the end product.

The various quality attributes that needs to be addressed in any embedded system development are broadly classified into two, namely '*Operational Quality Attributes*' and '*Non-Operational Quality Attributes*'.

Operational Quality Attributes:

The *operational quality attributes* represent the relevant quality attributes related to the embedded system when it is in the operational mode or 'online' mode.

The important quality attributes coming under this category are listed below:

1. ***Response:*** is a measure of quickness of the system. It gives an idea about how fast your system is tracking the changes in input variables.
   - Most of the embedded systems demand fast response which should be almost Real Time.
     - For example, an embedded system deployed in flight control application should respond in a Real Time manner. Any response delay in the system will create potential damages to the safety of the flight as well as the passengers.
   - It is not necessary that all embedded systems should be Real Time in response.
     - For example, the response time requirement for an electronic toy is not at all time critical. There is no specific deadline that this system should respond wit in this particular timeline.

2. ***Throughput:*** deals with the efficiency of a system. *Throughput* is defined as the rate of production or operation of a defined process over a stated period of time.
   - The rates can be expressed in terms of units of products, batches produced, or any other meaningful measurements.

  - In case of a Card Reader, throughput means how many transactions the Reader can perform in a minute or in an hour or in a day.
- Throughput is generally measured in terms of 'Benchmark'.
- A '*Benchmark*' is a reference point by which something can be measured.
- Benchmark can be a set of performance criteria that a product is expected to meet or a standard product that can be used for comparing other products of the same product line.

3. ***Reliability:*** is a measure of how much % you can rely upon the proper functioning of the system or what is the% susceptibility of the system to failures.
   - *Mean Time Between Failures* (MTBF) and *Mean Time To Repair* (MTTR) are the terms used in defining system reliability.
     - *MTBF* gives the frequency of failures in hours/ weeks/ months.
     - *MTTR* specifies how long the system is allowed to be out of order following a failure.
   - For an embedded system with critical application need, it should be of order of minutes.

4. ***Maintainability:*** deals with support and maintenance to the end user or client in case of technical issues and product failures or on the basis of a routine system checkup.
   - *Reliability* and *Maintainability* are considered as two complementary disciplines.
   - A more *reliable system* means a system with less corrective maintainability requirements and vice versa. As the reliability of the system increases, the chances of failure and non-functioning reduces, thereby the need for maintainability is also reduced.
   - Maintainability is closely related to the system availability. Maintainability can be broadly classified into two categories, namely, '*Scheduled* or *Periodic Maintenance* (*preventive maintenance*)' and '*Maintenance to unexpected failures* (*corrective maintenance*)'.
   - Some embedded products may use consumable components or may contain components which are subject to wear and tear and they should be replaced on a periodic basis. The period may be based on the total hours of the system usage or the total output the system delivered.
     - A printer is a typical example for illustrating the two types of maintainability. An inkjet printer uses ink cartridges, which are consumable components and as per the printer manufacturer the end user should replace the cartridge after each 'n' number of printouts, to get quality prints. This is an example for '*Scheduled or Periodic maintenance*'.
     - If the paper feeding part of the printer fails the printer fails to print and it requires immediate repairs to rectify this problem. This is an example of '*Maintenance to unexpected failure*'.

- In both of the maintenances (scheduled and repair), the-printer needs to be brought offline and during this time it will not be available for the user.
- In any embedded system design, the ideal value for availability is expressed as

Ai = MTBF / (MTBF + MTTR)

Where, Ai – Availability in the ideal conditions.

5. ***Security:*** aspect covers '*Confidentiality*', '*Integrity*', and '*Availability*' (The term 'Availability' mentioned here is not related to the term 'Availability' mentioned under the 'Maintainability' section).
   - *Confidentiality* deals with the protection of data and application from unauthorized disclosure.
   - *Integrity* deals with the protection of data and application from unauthorized modifications.
   - *Availability* deals with protection of data and application from unauthorized users.
     - A very good example of the '*Security*' aspect in an embedded product is a Personal Digital Assistant (PDA). The PDA can be either a shared resource (e.g. PDAs used in LAB setups) or an individual one.
   - If it is a shared one, there should be some mechanism in the form of user name and password to access into a particular person's profile – An example of' Availability.
   - Also all data and applications present in the PDA need not be accessible to all users. Some of them are specifically accessible to administrators only. For achieving this, Administrator and user level s of security should be implemented – An example of Confidentiality.
   - Some data present in the PDA may be visible to all users but there may not be necessary permissions to alter the data by the users. That is Read Only access is allocated to all users – An example of Integrity.

6. ***Safety:*** '*Safety*' and '*Security*' are confusing terms. Sometimes you may feel both of them as a single attribute. But they represent two unique aspects in quality attributes.
   - *Safety* deals with the possible damages that can happen to
     - the operators,
     - public and the environment;
     - due to
       - the breakdown of an embedded system,
       - the emission of radioactive or hazardous materials from the embedded products.
   - The breakdown of an embedded system may occur due to a hardware failure or a firmware failure.

- Safety analysis is a must in product engineering to evaluate the anticipated damages and determine the best course of action to bring down the consequences of the damages to an acceptable level.
- Some of the safety threats are sudden (like product breakdown) and some of them are gradual (like hazardous emissions from the product).

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.

The important quality attributes coming under this category are listed below:

1. ***Testability & Debug-ability:*** deals with how easily one can test his/ her design, application; and by which means he/ she can test it.
   - For an embedded product, *testability* is applicable to both the embedded hardware and firmware.
     - *Embedded hardware testing* ensures that the peripherals and the total hardware functions in the desired manner, whereas *firmware testing* ensures that the firmware is functioning in the expected way.
   - *Debug-ability* is a means of debugging the product as such for figuring out the probable sources that create unexpected behavior in the total system.
   - Debug-ability has two aspects in the embedded system development context, namely, hardware level debugging and firmware level debugging.
     - *Hardware debugging* is used for figuring out the issues created by hardware problems whereas *firmware debugging* is employed to figure out the probable errors that appear as a result of flaws in the firmware.

2. ***Evolvability:*** is a term which is closely related to Biology.
   - *Evolvability* is referred as the non-heritable variation. For an embedded system, the quality attribute 'Evolvability' refers to the ease with which the embedded product (including firmware and hardware) can be modified to take advantage of new firmware or hardware technologies.

3. ***Portability:*** is a measure of 'system independence'.
   - An embedded product is said to be *portable* if the product is capable of functioning 'as such' in various en environments, target processors/ controllers and embedded operating systems.
   - The ease with which an embedded product can be ported on to a new platform is a direct measure of the rework required.

- A standard embedded product should always be flexible and portable.
- In embedded products, the term '*porting*' represents the migration of the embedded firmware written for one target processor (i.e., Intel x86) to a different target processor (say ARM Cortex M3 processor).
- If the firmware is written in a high level language like 'C' with little target processor-specific functions (operating system extensions or compiler specific utilities), it is very easy to port the firmware for the new processor by replacing those 'target processor-specific functions' with the ones for the new target processor and re-compiling the program for the new target processor- specific settings. Re-compiling the program or the new target processor generates the new target processor-specific machine codes.
- If the firmware is written in Assembly Language for a particular family of processor (say x86 family), it will be difficult to translate the assembly language instructions to the new target processor specific language and so the portability is poor.
- If you look into various programming languages for application development for desktop applications, you will see that certain applications developed on certain languages run only on specific operating systems and some of them run independent of the desktop operating systems.
    - For example, applications developed using Microsoft technologies (e.g. Microsoft Visual C++ using Visual studio) is capable of running only on Microsoft platforms and will not function on other operating systems; whereas applications developed using 'Java' from Sun Microsystems works on any operating system that supports Java standards.

4. ***Time to Prototype and Market:*** is the time elapsed between the conceptualization of a product and the time at which the product is ready for selling (for commercial product) or use (for non-commercial products).

- The commercial embedded product market is highly competitive and time to market the product is a critical factor in the success of a commercial embedded product. There may be multiple players in the embedded industry who develop products of the same category (like mobile phone, portable media players, etc.). If you come up with a new design and if it takes long time to develop and market it, the competitor product may take advantage of it with their product.
- Also, embedded technology is one where rapid technology change is happening. If you start your design by making use of a new technology and if it takes long time to develop and market the product, by the time you market the product, the technology might have superseded with a new technology.

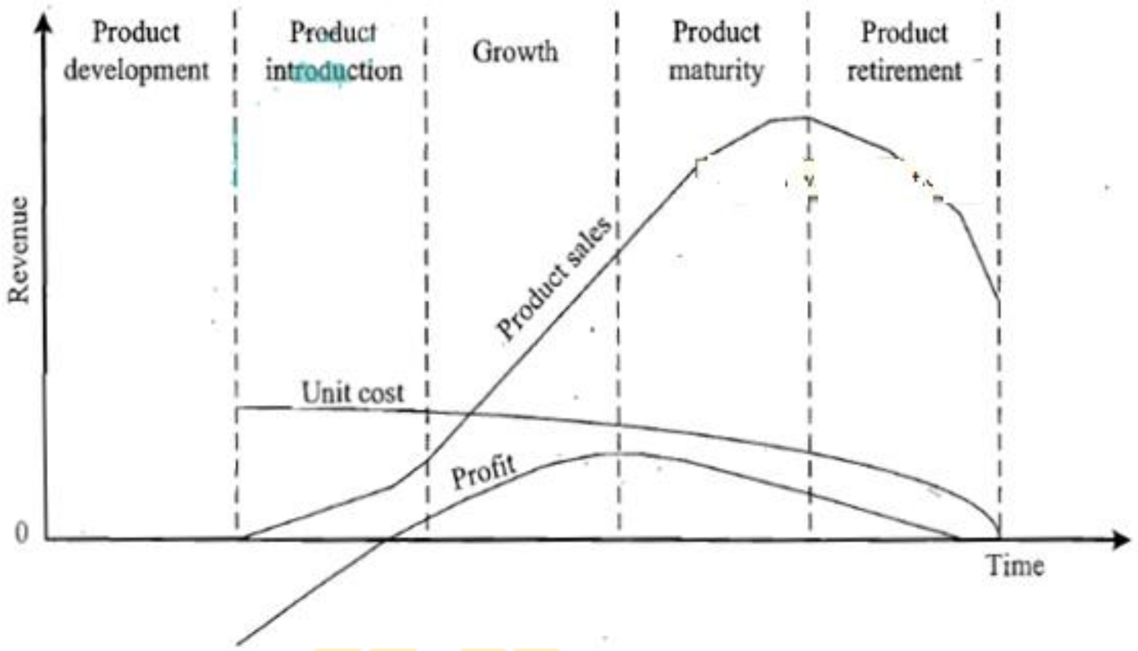
- *Product prototyping* helps a lot in reducing time-to-market. Whenever you have a product idea, you may not be certain about the feasibility of the idea.
- *Prototyping* is an informal kind of rapid product development in which the important features of the product under consideration are developed.
- The *time to prototype* is also another critical factor. If the prototype is developed faster, the actual estimated development time can be brought down significantly. In order to shorten the time to prototype, make use of all possible options like the use of off-the-shelf components, re-usable assets, etc.

5. ***Per Unit and Total Cost:*** is a factor which is closely monitored by both end user (those who buy the product) and product manufacturer (those who build the product).
   - Cost is a highly sensitive factor for commercial products. Any failure to position the cost of a commercial product at a nominal rate, may lead to the failure of the product in the market. Proper market study and cost benefit analysis should be carried out before taking a decision on the per-unit cost of the embedded product.
   - From a designer/ product development company perspective the ultimate aim of a product is to generate marginal profit. So the budget and total system cost should be properly balanced to provide a marginal profit.

The Product Life Cycle (PLC): Every embedded product has a product life cycle which starts with the design and development phase.

- The product idea generation; prototyping, Roadmap definition, actual product design and development are the activities carried out during this phase.
- During the *design and development phase* there is only investment and no returns.
- Once the product is ready to sell, it is introduced to the market. This stage is known as the *Product Introduction stage*.
- During the initial period the sales' and revenue will be low. There won't be much competition and the product sales and revenue increases with time. In the *growth phase*, the product grabs high market share.
- During the *maturity phase*, the growth and sales will be steady and the revenue reaches its peak.
- The *Product retirement/ Decline phase* starts with the drop in sales volume; market share and revenue. The decline happens due to various reasons like competition from similar product with enhanced features or technology changes, etc. At some point of the decline stage, the manufacturer announces discontinuing of the product.

- The different stages of the embedded products life cycle-revenue, unit cost and profit in each stage-are represented in the following Product Life-cycle graph.



EMBEDDED SYSTEMS – APPLICATION- AND DOMAIN- SPECIFIC

Embedded systems are *application and domain specific*, meaning; they are specifically built for certain applications in certain domains like consumer electronics, telecom, automotive, industrial control, etc.

It is possible to replace a general purpose computing system with another system which is closely matching with the existing system, whereas it is not the case with embedded systems.

Embedded systems are highly specialized in functioning and are dedicated for a specific application. Hence it is not possible to replace an embedded system developed for a specific application in a specific domain with another embedded system designed for some other application in some other domain.

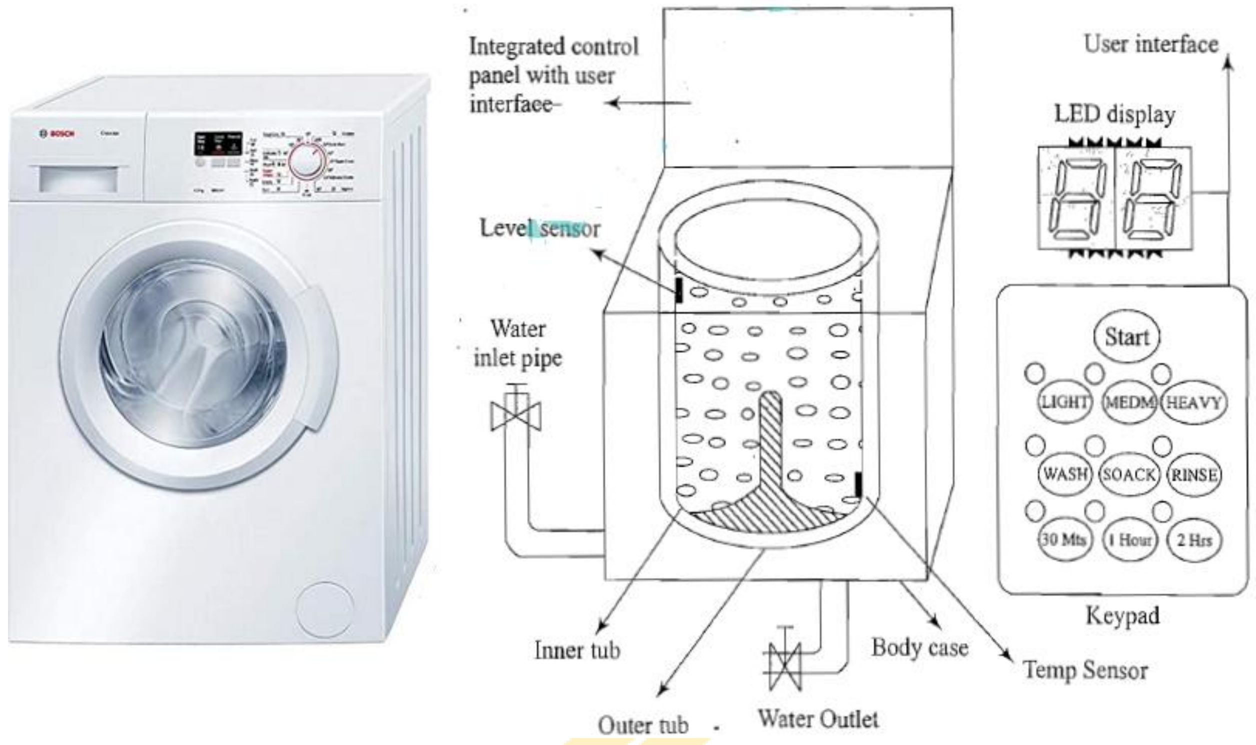
WASHING MACHINE – APPLICATION-SPECIFIC EMBEDDED STSREM:

Washing machine is a typical example of an embedded system providing extensive support in home automation applications.

- An embedded system contains sensors, actuators, control unit and application-specific user interfaces like keyboards, display units, etc. One can see all these components in a washing machine. Some of them are visible and some of them may be invisible.
- The actuator part of the washing machine consists of a motorized agitator, tumble tub, water drawing pump and inlet valve to control the flow of water into the unit.
- The sensor part consists of the water temperature sensor, level sensor, etc.
- The control part contains a micro- processor/ controller based board with interfaces to the sensors and actuators.

- The sensor data is fed back to the control unit and the control unit generates the necessary actuator outputs. The control unit also provides connectivity to user interfaces like keypad for setting the washing time, selecting the type of material to be washed like light, medium, heavy duty, etc. User feedback is reflected through the display unit and LEDs connected to the control board.

The functional block diagram of washing machine is shown in the following Figure.



Washing machine comes in two models, namely, top loading and front loading machines.

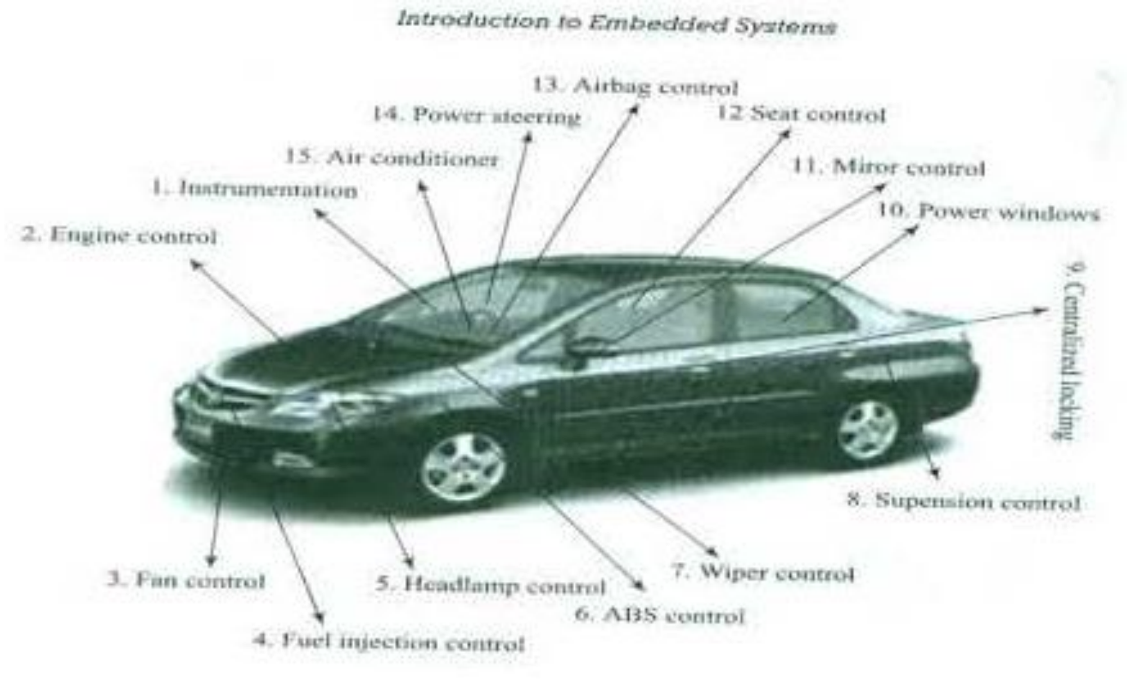
- In top loading models the agitator of the machine twists back and forth and pulls the cloth down to the bottom of the tub. On reaching the bottom of the tub the clothes work their way back up to the top of the tub where the agitator grabs them again and repeats the mechanism.
- In the front loading machines, the clothes are tumbled and plunged into the water over and over again. This is the first phase of washing.
- In the second phase of washing, water is pumped out from the tub and the inner tub uses centrifugal force to wring out more water from the clothes by spinning at several hundred Rotations Per Minute (RPM). This is called a '*Spin Phase*'.
- If you look into the keyboard panel of your washing machine you can see three buttons: Wash, Spin and Rinse. You can use these buttons to configure the washing stages.
- As you can see from the picture, the inner tub of the machine contains a number of holes and during the spin cycle the inner tub spins, and forces the water out through these holes to the stationary outer tub from which it is drained off through the outlet pipe.

It is to be noted that the design of washing machines may vary from manufacturer to manufacturer, but the general principle underlying in the working of the washing machine remains the same.

- The basic controls consist of a timer, cycle selector mechanism, water temperature selector, load size selector and start button.
- The mechanism includes the motor, transmission, clutch, pump, agitator, inner tub, outer tub and water inlet valve. Water inlet valve connects to the water supply line using at home and regulates the flow of water into the tub.
- The integrated control panel consists of a microprocessor/ controller based board with I/O interfaces and a control algorithm running in it.
- Input interface includes the keyboard which consists of wash type selector: Wash, Spin and Rinse; clothe selector: Light, Medium, Heavy duty and washing time setting, etc.
- The output interface consists of LED/ LCD displays, status indication LEDs, etc. connected to the I/O bus of the controller.
- The other types of I/O interfaces which are invisible to the end user are different kinds of sensor interfaces: water temperature sensor, water level sensor, etc., and actuator interface including motor control for agitator and tub movement control, inlet water flow control, etc.

AUTOMATIVE – DOMAIN-SPECIFIC EXAMPLES EMBEDDED STSREM:

The major application domains of embedded systems are consumer, industrial, automotive, telecom, etc., of which telecom and automotive industry holds a big market share. The following Figure gives an overview of the various types of electronic control units employed in automotive applications.



Inner Working of Automotive Embedded Systems:

- Automotive embedded systems are the one where electronics take control over the mechanical systems. The presence of automotive embedded system in a vehicle varies from simple mirror and wiper controls to complex air bag controller and antilock brake systems (ABS).
- Automotive embedded systems are normally built around microcontrollers or DSPs or a hybrid of the two and are generally known as Electronic Control Units (ECUs). The number of embedded controllers in an ordinary vehicle varies from 20 to 40 whereas a luxury veh1cle like Mercedes S and BMW 7 may contain over 100 embedded controllers.
- The first embedded system used in automotive application was the microprocessor based fuel injection system introduced by Volkswagen 1600 in 1968.

The various types of electronic control units (ECUs) used in the automotive embedded industry can be broadly classified into two-*High-speed embedded control units* and *Low-speed embedded control units.*

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): 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:

Automotive applications make use of serial buses for communication, which greatly reduces the amount of wiring required inside a vehicle. The different types of serial interface buses deployed in automotive embedded applications are –

1. ***Controller Area Network (CAN):*** The CAN bus was originally proposed by Robert Bosch, pioneer in the Automotive embedded solution providers.
   - CAN supports medium speed (ISO 11519-class B with data rates up to 125 Kbps) and high speed (ISO 11898 class C with data rates up to 1Mbps) data transfer.
   - CAN is an event-driven protocol interface with support for error handling in data transmission.
   - It is generally employed in-safety system like airbag control; power train systems like engine control and Antilock Brake System (ABS); and navigation systems like GPS.

2. ***Local Interconnect Network (LIN):*** LIN bus is 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 20 Kbps and is used for sensor/ actuator interfacing.
    - LIN bus 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.

3. Media Oriented System Transport (MOST) Bus: MOST is targeted for automotive audio/ video equipment interfacing, used primarily in European cars.
    - A MOST bus is a multimedia fibre-optic point-to-point network implemented in a star, ring or daisy- chained topology over optical fibre 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 fibre cable connected between the Electrical Optical Converter (EOC) and Optical Electrical Converter (OEC), which would translate into the optical cable MOST bus.

Key Players of the Automotive Embedded Market:

The key players of the automotive embedded market can be visualized in three verticals namely, silicon providers, tools and platform providers and solution providers.

1. ***Silicon Providers:*** are responsible for providing the necessary chips which are used in the control application development.
    - The chip may be a standard product like microcontroller or DSP or ADC/ DAC chips.
    - Some applications may require specific chips and they are manufactured as Application Specific Integrated Chip (ASIC).
    - The leading silicon providers in the automotive industry are:
        a) **Analog Devices (www.analog.com):** Provider of world class digital signal processing chips, precision analog microcontrollers, programmable inclinometer/accelerometer, LED drivers, etc. for automotive signal processing applications, driver assistance systems, audio system, GPS/Navigation system, etc.
        b) **Xilinx (www.xilinx.com):** Supplier of high performance FPGAs, CPLDs and automotive specific IP cores for GPS navigation systems, driver information systems, distance

control, collision avoidance, rear1seat entertainment, adaptive cruise control, voice recognition, etc.

c) **Atmel (www.atmel.com):** Supplier of cost-effective high-density Flash controllers and memories. Atmel provides a series of high performance microcontrollers, namely, ARM®1 and 80C51. A wide range of Application Specific Standard Products (ASSPs) for chassis, body electronics, security, safety and car infotainment and automotive networking products for CAN, LIN and FlexRay are also supplied by Atmel.

d) **Maxim/Dallas (www.maxim-ic.com):** Supplier of world class analog, digital and mixed signal products (Microcontrollers, ADC/ DAC, amplifiers, comparators, regulators, etc), RF components, etc. for all kinds of automotive solutions.

e) **NXP semiconductor (www.nxp.com):** Supplier of 8/ 16/ 32 Flash microcontrollers.

f) **Renesas (www.renesas .com):** Provider of high speed microcontrollers, battery control systems, power train solutions, chassis and safety solutions, body control solutions, instrument cluster solutions, etc.

g) **Texas Instruments (www.ti.com):** Supplier of microcontrollers, digital signal processors and automotive communication control chips for Local Inter Connect (LIN bus products.

h) **Fujitsu (www.fmal.fujitsu.com):** Fujitsu is global leader in graphics display controllers (GDCs), including instrument clusters, in-dash navigation, heads-up displays and rear-seat entertainment. It also offers automotive controller for HD video in vehicle networks.

i) **Infineon (www.infineon.com):** Supplier of high performance microcontrollers and customized application specific chips.

j) **Freescale Semiconductor (www.freescale.com):** Solution provider for Advanced Driver Assistance Systems (ADAS), Body Electronics, Chassis and Safety, Instrument cluster, Power train and Hybrid systems.

k) **Microchip (www.microchip.com):** Supplier of robust automotive grade microcontroller, analog and memory products, CAN/ LIN transceivers, etc.

2. ***Tool and Platform Providers:*** are manufacturers and suppliers of various kinds of development tools and Real Time Embedded Operating Systems for developing and debugging different control unit related applications.
   - Tools fall into two categories, namely embedded software application development tools and embedded hardware development tools.
   - Some of the leading suppliers of tools and platforms in automotive embedded applications are listed below:

a) **ENEA (www.enea.com):** Enea Embedded Technology is the developer of the OSE Real-Time operating system. The OSE RTOS supports both CPU and DSP and has also been specially developed to support multi-core and fault-tolerant system development. .

b) **The Math Works (www.mathworks.com):** It is the world's leading developer and supplier of technical software. It offers a wide range of tools, consultancy and training for numeric computation, visualization, modeling and simulation across many different industries. MathWork's breakthrough product is MATLAB – a high-level programming language and environment for technical computation and numerical analysis. Together MATLAB, SIMULINK, Stateflow and Real-Time Workshop provide top quality tools for data analysis, test & measurement, application development and deployment, image processing and development of dynamic and reactive systems for DSP and control applications.

c) **Keil Software (www.keil.com):** The Integrated Development Environment Keil Microvision from Keil software is a powerful embedded software design tool for 8051 & C166 family of microcontrollers.

d) **Lauterbach (http://www.lauterbach.com/):** It is the world's number one supplier of debug tools, providing support for processors from multiple silicon vendors in the automotive market.

e) Atego Modeling Tools (http://www.atego.com): It is the leading supplier of collaborative modeling tools for requirement analysis, specification, design and development of complex applications.

f) **Microsoft (www.microsoft.com):** It is a platform provider for automotive embedded applications. Microsoft's Windows Embedded Automotive is an extensible technology platform for automakers and suppliers to deliver in-car experiences that keep drivers connected and informed.

3. ***Solution Providers:*** Solution providers supply Original Equipment Manufacturer (OEM) and complete solution for automotive applications making use of the chips, platforms and different development tools.

   - The major players of this domain are listed below:

     a) **Bosch Automotive (www.boschindia.com):** Bosch is providing complete automotive solution ranging from body electronics, diesel engine control, gasoline engine control, power train systems, safety systems, in-car navigation systems and infotainment systems.

     b) **DENSO Automotive (www.globaldensoproducts.com):** Denso is an OEM and solution provider for engine management, climate control, body electronics, driving control & safety, hybrid vehicles, embedded infotainment and communications.

c) **Infosys Technologies (www.infosys.com):** Infosys is a solution provider for automotive embedded hardware and software. Infosys provides the competitive edge in integrating technology change through cost effective solutions.

d) **Delphi (www.delphi.com):** Delphi is the complete solution provider for engine control, safety, infotainment, etc., and OEM for spark plugs, bearings, etc.

HARDWARE SOFTWARE CO-DESIGN AND PROGRAM MODELING

In the traditional embedded system development approach, the hardware software partitioning is done at an early stage and 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.

FUNDAMENTAL ISSUES IN HARDWARE SOFTWARE CO-DESIGN:

The hardware software co-design is a problem statement and when we try to solve this problem statement in real life we may come across multiple issues in the design. The following section illustrates some of the fundamental issues in hardware software co-design.

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 varieties 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 component is 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.
- 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.
- 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.).
    - The ***Controller Architecture*** implements the finite state machine model (FSM) using a state register and two combinational circuits. The state register holds the present state and the combinational circuits implement the logic for next state and output.
    - The ***Datapath Architecture*** is best suited for implementing the data flow graph model where the output is generated as a result of a set of predefined computations on the input data. A datapath represents a channel between the input and output; and in datapath architecture the datapath may contain registers, counters, register files, memories and ports along with high speed arithmetic units. Ports connect the datapath to multiple buses.
    - The ***Finite State Machine Datapath (FSMD) architecture*** combines the controller architecture with datapath architecture. It implements a controller with datapath. The controller generates the control input, whereas the datapath processes the data. The datapath contains two types of I/O ports, out of which one acts as the control port for receiving/ sending the control signals from/ to the controller unit and the second I/O port interfaces the datapath with external world for data input and data output.
    - The ***Complex Instruction Set Computing (CISC) architecture*** uses an instruction set representing complex operations. It is possible for a CISC instruction set to perform a large complex operation with a single instruction. The use of a single complex instruction in place of multiple simple instructions greatly reduces the program memory access and program memory size requirement. However it requires additional silicon for implementing microcode decoder for decoding the CISC instruction. The datapath for the CISC processor is complex.
    - The ***Reduced Instruction Set Computing (RISC) architecture*** reuses instruction set representing simple operations and it requires the execution of multiple RISC instructions to perform a complex operation. The data path of RISC architecture contains a large register file for storing the operands and output. RISC instruction set is designed to operate on registers. RISC architecture supports extensive pipelining.

- The ***Very Long Instruction Word (VLIW) architecture*** implements multiple functional units (ALUs, multipliers, etc.) in the datapath. The VLIW instruction packages one standard instruction per functional unit of the datapath.
- ***Parallel Processing architecture*** implements multiple concurrent Processing Elements (PEs) and each processing element may associate a datapath containing register and local memory.
- ***Single Instruction Multiple Data (SIMD)*** and ***Multiple Instruction Multiple Data (MIMD) architectures*** are examples for parallel processing architecture.
  - In SIMD architecture, a single instruction is executed in parallel with the help of the Processing Element. The scheduling-of the instruction execution and controlling of each PE is performed through a single controller. The SIMD architecture forms the basis of reconfigurable processor.
  - On the other hand, the processing elements of the MIMD architecture execute different instructions at a given point of time. The MIMD architecture forms the basis of multiprocessor systems. The PEs in a multiprocessor system communicates through mechanisms like shared memory and message passing.

Selecting the Language: A programming language captures a 'Computational Model' and maps it into architecture. There is no hard and fast rule to specify which 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.

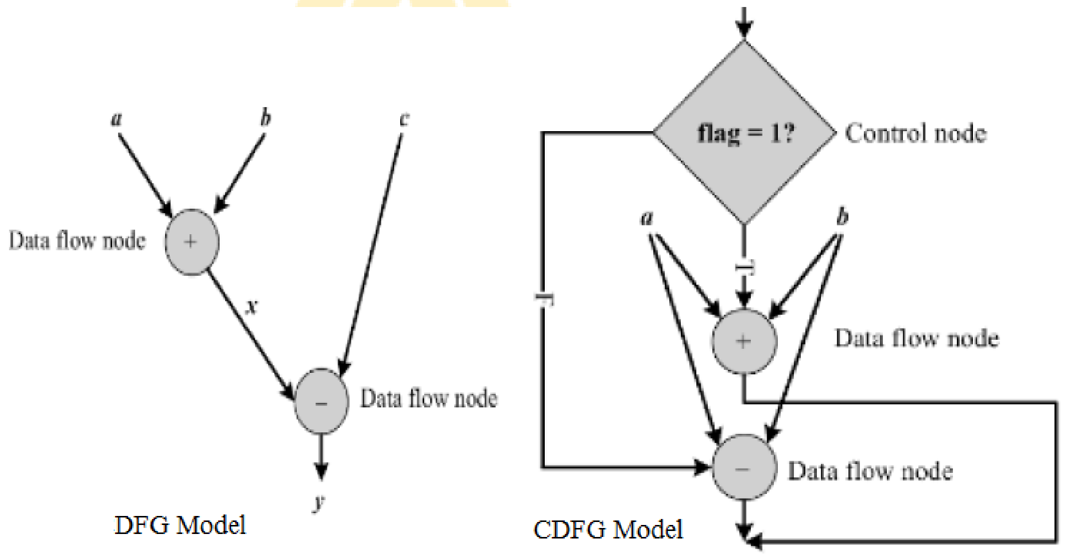
Partitioning System Requirements into Hardware and Software: It may be possible to implement the system requirements in either hardware or software (firmware). It is a tough decision making task to figure out which one to opt. Various hardware software trade-offs are used for making a decision on the hardware-software partitioning.

COMPUTATIONAL MODELS IN EMBEDDED DESIGN:

Data Flow Graph (DFG) model, State Machine model, Concurrent Process model, Sequential Program model, Object Oriented model, etc. are the commonly used *computational models* in embedded system design.

Data Flow Graph/ Diagram (DFG) Model: The *DFG* model translates the data processing requirements into a data flow graph.

- The *Data Flow Graph model* is a data driven model in which the program execution is determined by data. This model emphasizes on the data and operations on the data transform the input data to output data.
- Indeed Data Flow Graph 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.
- Embedded applications which are computational intensive and data driven are modeled using the DFG model.
- Suppose one of the functions in an application contains the computational requirement $x = a + b$; and $y = x - c$. The following Figure illustrates the implementation of a DFG model for implementing these requirements.



- 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: In a DFG model, the execution is controlled by data and it doesn't involve any control operations (conditionals).

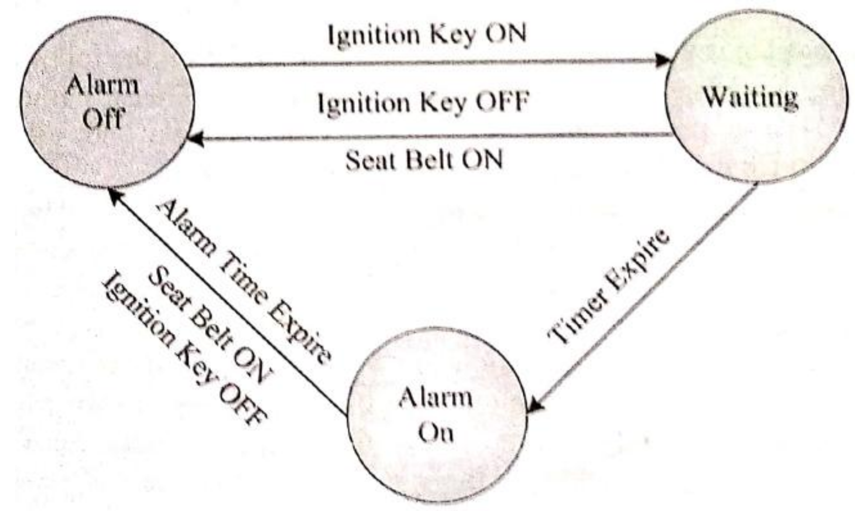
- The *Control DFG (CDFG) model* is used for modeling applications involving conditional program execution. CDFG models contains both data operations and control operations.

- 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. Let us have a look at 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 above 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 modeled to a concurrent process model. The decision on which process is to be executed is determined by the control node.
    - A real world example for modeling the embedded application using CDFG is the capturing and saving of the image to a format set by the user in a digital still camera where everything is data driven starting from the Analog Front End which converts the CCD sensor generated analog signal to Digital Signal and the task which stores the data from ADC to a frame buffer for the use of a media processor which performs various operations like, auto correction, white balance adjusting, etc. 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.

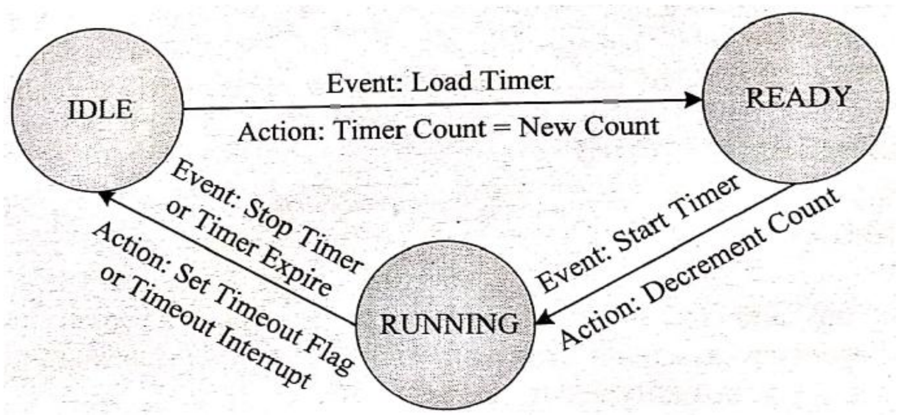
State Machine Model: The *State Machine model* is used for modeling reactive or event-driven embedded systems whose processing behavior is dependent on state transitions. Embedded systems used in the control and industrial applications are typical examples for event driven systems.

- The State Machine model describes the system 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.
    - 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 fasten the belt or if the ignition switch is turned off, whichever happens first.
- Here the states are '*Alarm Off*', '*Waiting*' and '*Alarm On*' and the events are '*Ignition Key ON*', '*Ignition Key OFF*', '*Timer Expire*', '*Alarm Time Expire*' and '*Seat Belt ON*'.
- Using the FSM, the system requirements can be modeled as given in following 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 modeled as shown in the following Figure.

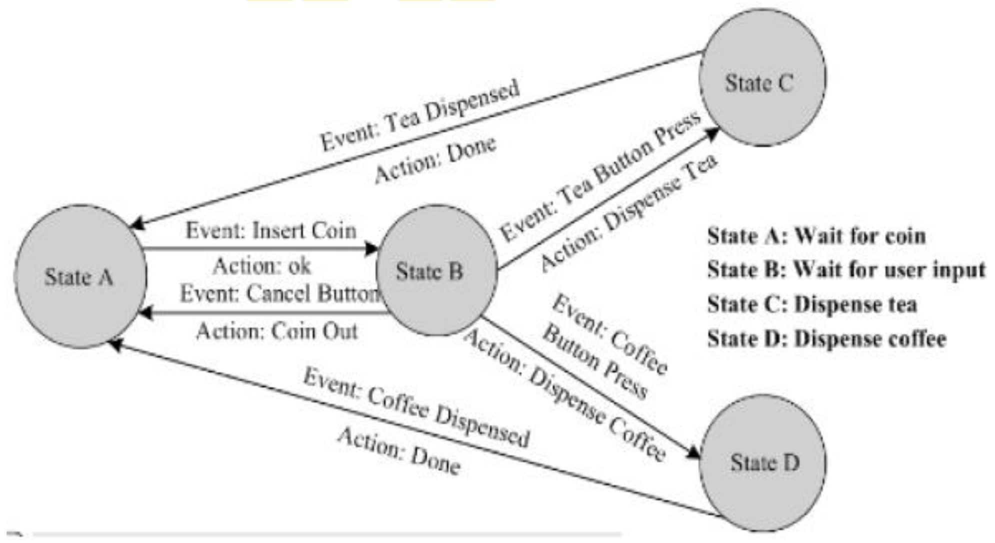

IDLE
Event: Load Timer
Action: Timer Count = New Count
READY
Event: Stop Timer or Timer Expire
Action: Set Timeout Flag or Timeout Interrupt
RUNNING
Event: Start Timer
Action: Decrement Count

- As seen from the FSM, the timer state can be either '*IDLE*' or '*READY*' or '*RUNNING*'.
- During the normal condition when the timer is not running, it is said to be in the '*IDLE*' state.
- 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.

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.

The FSM representation for the above requirement is given in the following Figure.

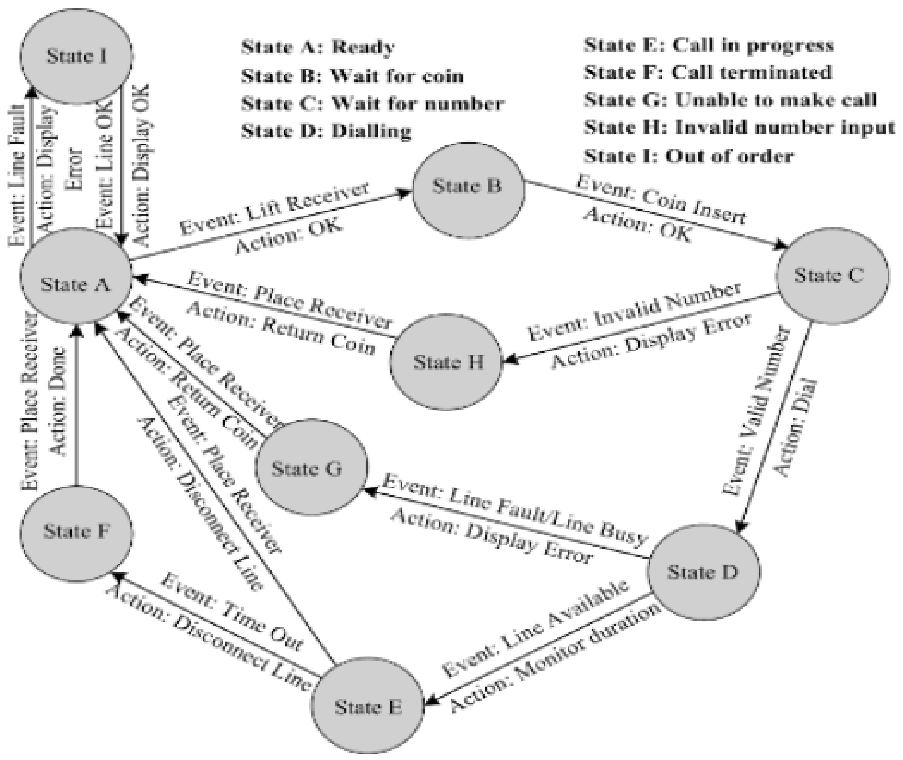


- The FSM representation contains four states namely; '*Wait for coin*' '*Wait for User Input*', '*Dispense Tea*' and '*Dispense Coffee*'.
- The event '*Insert Coin*' (5 rupee coin insertion), transitions the state to '*Wait for User Input*'. The system stays in this state until a user input is received from the buttons '*Cancel*', '*Tea*' or '*Coffee*'.
- 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*' or '*Dispense Coffee*' respectively.
- Once the coffee/ tea vending is over, the respective states transitions back to the '*Wait for Coin*' state.

Example 2: Design a coin operated public telephone unit based on FSM model for the following requirements.

1. The calling process is initiated by lifting the receiver (off-hook) of the telephone unit
2. After lifting the phone the user needs to insert a 1 rupee coin to make the call
3. If the line is busy, the coin is returned on placing the receiver back on the hook (on-hook)
4. 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
5. If the user doesn't insert another 1 rupee coin, the call is terminated on completing the 60 seconds time slot
6. The system is ready to accept new call request when the receiver is placed back on the hook (on-hook)
7. The system goes to the '*Out of Order*' state when there is a line fault.

The FSM model shown in the following Figure is a simple representation.



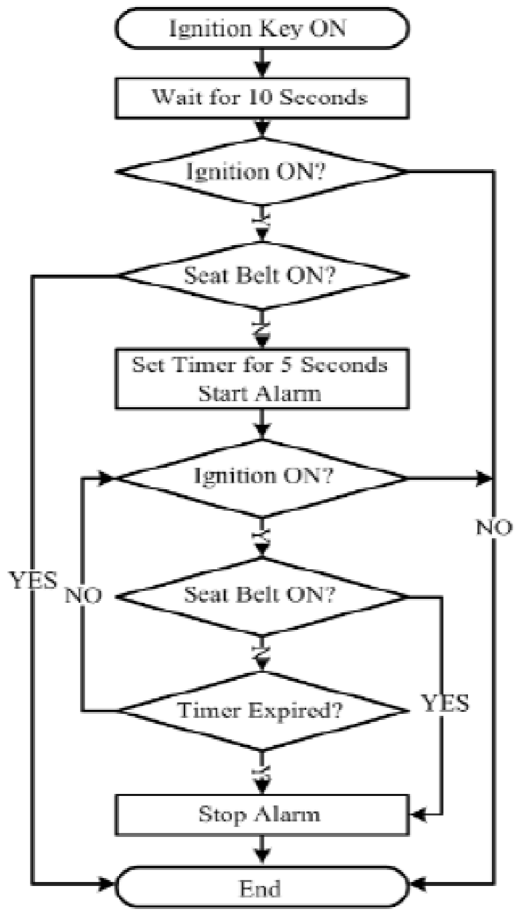
- Most of the time state machine model translates the requirements into sequence driven program and it is difficult to implement concurrent processing with FSM. This limitation is addressed by the *Hierarchical/ Concurrent Finite State Machine model (HCFSM)*.
- The HCFSM is an extension of the FSM for 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.

Sequential Program Model: In the sequential programming Model, the functions or processing requirements are executed in sequence. It is same as the conventional procedural programming.

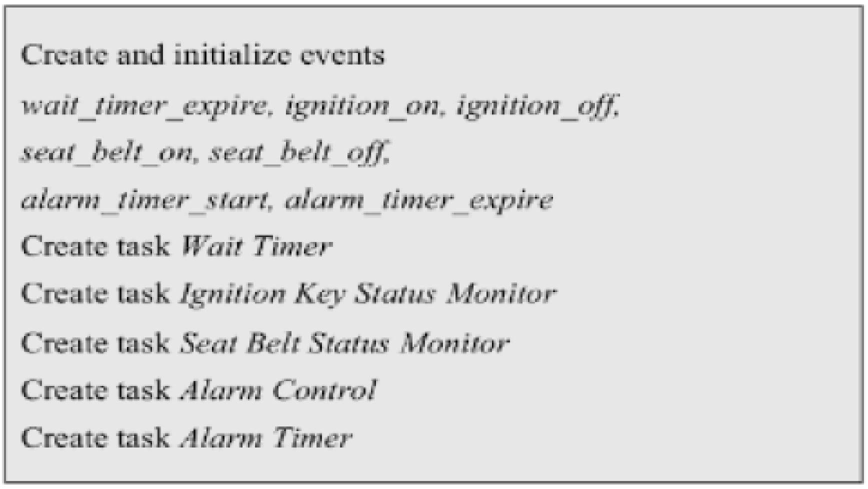
- Here the program instructions are iterated and executed conditionally and the data gets transformed through a series of operations.
- FSMs are good choice for sequential program modeling.
- Another important tool used for modeling sequential program is Flow Charts.
- The FSM approach represents the 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 ()
                               == OFF) &&
                              (timer_expire () == ON));
                              stop_alarm ():
                  }
      }
}
```



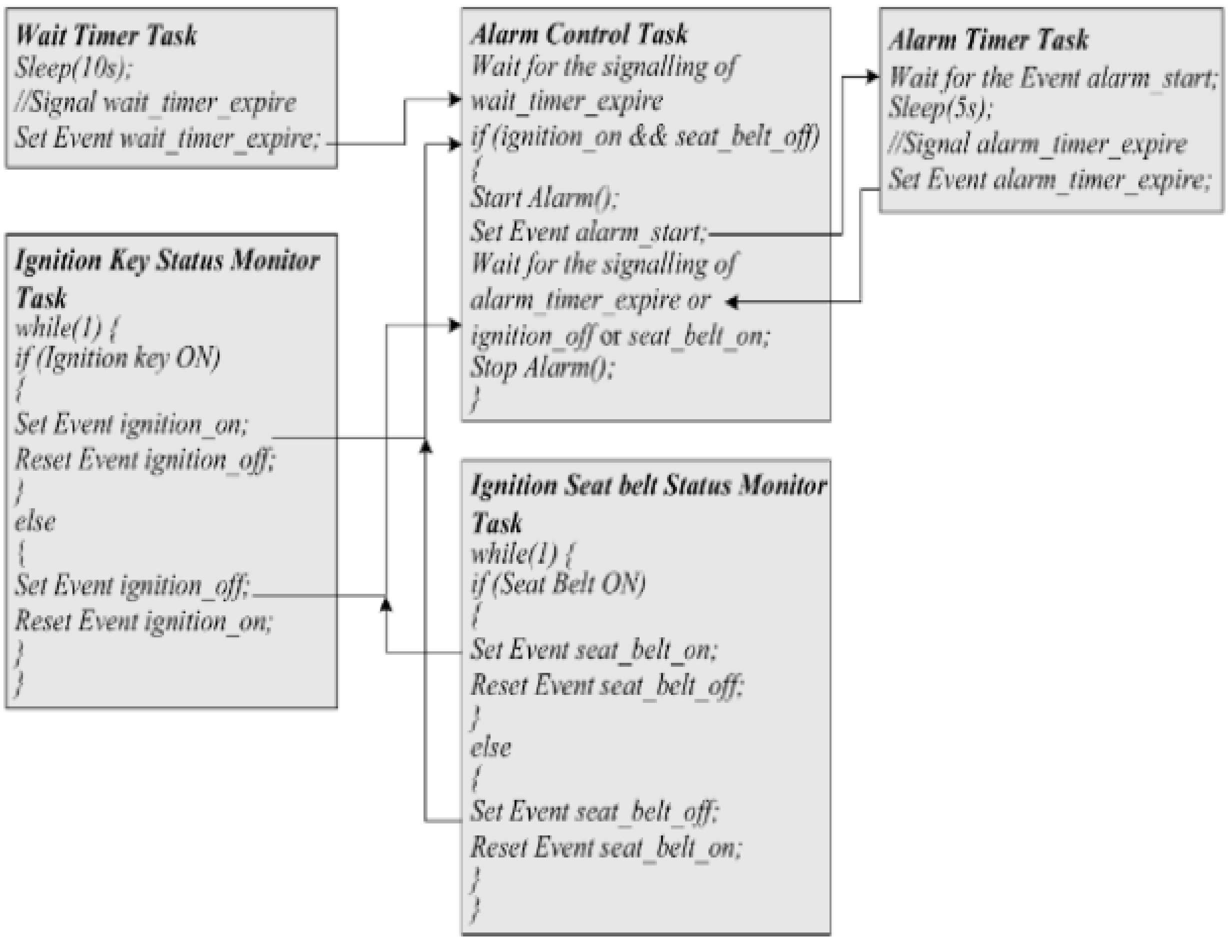
Concurrent/ Communicating Process Model: The concurrent or communicating process model models concurrently executing tasks/ processes.

- 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 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.
- However, concurrent processing model requires additional overheads in task scheduling, task synchronization and communication.
- As an example for the concurrent processing model let us examine how we can implement the '*Seat Belt Warning*' system in concurrent processing model. We can split the tasks into:
    1. Timer task for waiting 10 seconds (wait timer task)
    2. Task for checking the ignition key status (ignition key status monitoring task)
    3. Task for checking the seat belt status (seat belt status monitoring task)
    4. Task for starting and stopping the alarm (alarm control task)
    5. Alarm timer task for waiting 5 seconds (alarm timer task)
- We have five tasks here and we cannot execute them randomly or sequentially. We need to synchronize their execution through some mechanism.
- We need to start the alarm only after the expiration of the 10 seconds wait timer and that too only if the seat belt is OFF and the ignition key is ON. Hence the alarm: control task is executed only when the wait timer is expired and if the ignition key is in the ON state and seat belt is in the OFF state.
- One way of implementing a concurrent model for the '*Seat Belt Warning*' system is illustrated in the following Figure.



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*

Tasks for "Seat Belt Warning" System



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

```
Alarm Control Task
Wait for the signalling of
wait_timer_expire
if (ignition_on && seat_belt_off)
{
Start Alarm();
Set Event alarm_start;
Wait for the signalling 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 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;
}
}
```

```
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;
}
}
```

Concurrent processing Program model for "Seat Belt Warning" System

Object-Oriented Model: The object-oriented model is an object based model for modeling 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 modeling, object is an entity used for representing or modeling a particular piece of the system. Each object is characterized by a set of unique behavior 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 behavior 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.
- The concept of object and class brings abstraction, hiding and protection.

EMBEDDED FIRMWARE DESIGN AND DEVELOPMENT

The embedded firmware is responsible for controlling the various peripherals of the embedded hardware and generating response in accordance with the functional requirements mentioned in the requirements for the particular embedded product. Firmware is considered as the master brain of the embedded system. Embedded firmware is stored at a permanent memory (ROM).

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 (low level assembly language or a high level languages like C/C++/JAVA).

Embedded firmware development process starts with the conversion of the firmware requirements into a program model using modeling skills like UML or flow chart based representation. The UML diagrams or flow chart gives a diagrammatic representation of the decision items to be taken and the tasks to be performed.

EMBEDDED FIRMWARE DESIGN APPROACHES:

The firmware design approaches for embedded product is purely dependent on the complexity of the functions to be performed, the speed of operation required, etc.

- Two basic approaches are used for embedded firmware design. They are '*Conventional Procedural Based Firmware Design*' (also known as '*Super Loop Model*') and '*Embedded Operating System (OS) Based Design*'.

The Super Loop Based Approach:

The *Super Loop based firmware development approach* is adopted for applications that are not time critical and where the response time is not so important.

- Super loop approach is very similar to a conventional procedural programming where the code is executed task by task. The task, listed at the top of the program code, is executed first and the tasks, just below the top are executed after completing the first task. This is a true procedural one.
- In a multiple task based system, each task is executed in serial in this approach. The firmware execution flow for this will be –
    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.

- From the firmware execution sequence, it is obvious that the order in which the tasks to be executed are fixed and they are hard coded in the code itself. Also the operation is an infinite loop based approach.
- We can visualize the operational sequence listed above in terms of a '*C*' program code as –

```
void main ()
    {
            configurations ();
            initializations ();
            while (1)
            {
                    Task 1 ();
                    Task 2 ():
                    :
                    :
                    Task n ();
            }
    }
```

- From the above '*C*' code you can see that the tasks *1* to *n* are performed one after another and when the last task (*n*th task) is executed, the firmware execution is again re-directed to Task *1* and it is repeated forever in the loop. This repetition is achieved by using an infinite loop. This approach is also referred as '*Super loop based Approach*'.
- Since the tasks are running inside an infinite loop, the only way to come out of the loop is either a hardware reset or an interrupt assertion.
    - A hardware reset brings the program execution back to the main loop.
    - An interrupt request suspends the task execution temporarily and performs the corresponding interrupt routine and on completion of the interrupt routine it restarts the task execution from the point where it got interrupted.

- The 'Super loop based design' doesn't require an operating system, since there is no need for scheduling which task is to be executed and assigning priority to each task. In a super loop based design, the priorities are fixed and the order in which the tasks to be executed are also fixed. Hence the code for performing these tasks will be residing in the code memory without an operating system image.
- Super loop based design is deployed in low cost embedded products and products where response time is not time critical. Some embedded products demand this type of approach.
  - For example, reading/ writing data to and from a card using a card reader requires a sequence of operations like checking the presence of card, authenticating the operation, reading/writing, etc.
  - A typical example of a 'Super loop based' product is an electronic video game toy containing keypad and display unit. The program running inside the product may be designed in such a way that it reads the keys to detect whether the user has given any input and if any key press is detected the graphic display is updated. Even if the application misses a key press, it won't create any critical issues; rather it will be treated as a bug in the firmware.
- The 'Super loop based design' is simple and straight forward without any OS related overheads.
- The major drawback of this approach is that any failure in any part of a task will affect the total system. If the program hangs up at some point while executing a task, it may remain there forever and ultimately the product stops functioning.
- Another major drawback of 'Super loop based design' approach is the lack of real timeliness. If the number of tasks to be executed within an application increases, the time at which each task is repeated also increases. This brings the probability of missing out some events.
  - For example in a system with Keypads, according to the 'Super loop design', there will be a task for monitoring the keypad connected I/O lines and this need not be the task running while you press the keys. In order to identify the key press, you may have to press the keys for a sufficiently long time, till the keypad status monitoring task is executed internally by the firmware. This will really lead to the lack of real timeliness.
- There are corrective measures for this also. The best advised option in use interrupts for external events requiring real time attention.

The Embedded Operating System (OS) Based Approach:

The *Operating System (OS) based approach* contains operating systems, which can be either a General Purpose Operating System (GPOS) or a Real Time Operating System (RTOS) to host the user written application firmware.

- The *General Purpose OS (GPOS) based design* is very similar to a conventional PC based application development, where the device contains an operating system (Windows/ Unix/ Linux, etc. for Desktop PCs) and you will be creating and running user applications on top of it.
    - Example of a GPOS used in embedded product development is Microsoft® Windows XP Embedded 8.1, which offers customization to use industry devices like Personal Digital Assistants (PDAs), Hand held devices/ Portable devices, Point of Sale (PoS) Terminals, Patient Monitoring Systems, etc.
- OS based applications also require 'Driver software' for different hardware present on the board to communicate with them.
- The *Real Time Operating System (RTOS) based design* is employed in embedded products demanding Real-time response.
- RTOS respond in a timely and predictable manner to events. Real Time operating system contains a Real Time kernel responsible for performing pre-emptive multitasking, scheduler for scheduling tasks, multiple threads, etc. A Real Time Operating System (RTOS) allows flexible scheduling of system resources like the CPU and memory and offers some way to communicate between tasks.
    - '*Windows Embedded Compact*', '*pSOS*', '*VxWorks*', '*ThreadX*', '*MicroC/OS-III*', '*Embedded Linux*', '*Symbian*', etc., are examples of RTOS employed in embedded product development.

EMBEDDED FIRMWARE DEVELOPMENT LANGUAGES:

Embedded firmware can be –

✓ a target processor/ controller specific language (Assembly language or Low level language) or

✓ a target processor/ controller independent language (C, C++, JAVA, etc.) commonly known as High Level Language or

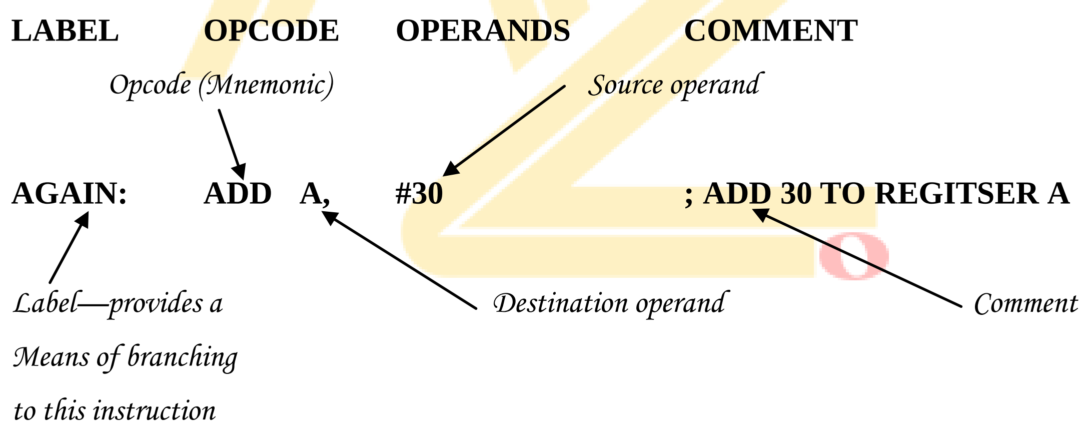
✓ a combination of Assembly and High level Language.

Assembly Language Based Development:

'*Assembly language*' is the human readable notation of '*machine language*', whereas '*machine language*' is a processor understandable language.

- Processors deal only with binaries (*1*s and *0*s). Machine language is a binary representation. Machine language is made readable by using specific symbols called '*mnemonics*'. Hence machine language can be considered as an interface between processor and programmer.
- Assembly language and machine languages are processor/ controller dependent and an assembly program written for one processor/ controller family will not work with others.

- *Assembly language programming* is the task 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.
    - Example: *MOV A, #30* – This instruction mnemonic moves decimal value 30 to the 8051 Accumulator register. Here *MOV A* is the Opcode and 30 is the operand. The same instruction when written in machine language: *01110100 00011110* – where the first 8-bit binary value *01110100*, represents the opcode *MOV A* and the second 8-bit binary value *00011110* represents the operand 30.
    - The mnemonic *INC A* is an example for instruction holding operand implicitly in the Opcode. The machine language representation of the same is *00000100*.
- Assembly language instructions are written one per line.
- A machine code program thus consists of a sequence of assembly language instructions, where each statement contains a mnemonic (Opcode + Operands). Each line of an assembly language program is split into four fields as given below:



- *LABEL* is an optional field. A '*LABEL*' is an identifier used extensively in programs to reduce the reliance on programmers or 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. Labels are used for representing subroutine names and jump locations in Assembly language programming.
- 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).
- 'LABEL' is not a mandatory field; it is optional.

- The sample code given below using 8051 Assembly language illustrates the structured assembly language programming. Each Assembly instruction should be written in a separate line.

```
DELAY:      MOV     R0, #255            ; Load Register R0 with 255.
            DJNE    R1, DELAY           ; Decrement R1 and loop till R1 = 0.
            RET                         ; Return to calling program.
```

- The Assembly program contains a main routine and it may or may not contain subroutines. The example given above is a subroutine, which can be invoked by a main program by the Assembly instruction: *LCALL DELAY*
    - Executing this instruction transfers the program flow to the memory address referenced by the 'LCALL DELAY'.
- It is a good practice to provide comments to your subroutines by indicating the purpose of that subroutine/ instruction.
- While assembling the code a ';' informs the assembler that the rest of the part coming in the line after the ';' symbol is comments and simply ignore it.
- In the above example the label DELAY represents the reference to the start of the subroutine DELAY. The required address is calculated by the assembler at the time of assembling the program and it replaces the label.
- Label can also be directly replaced by putting the desired address first and then writing the Assembly code, as given below:

```
ORG 0100H
            MOV     R0, #255            ; Load Register R0 with 255.
            DJNE    R1, DELAY           ; Decrement R1 and loop till R1 = 0.
            RET                         ; Return to calling program.
```

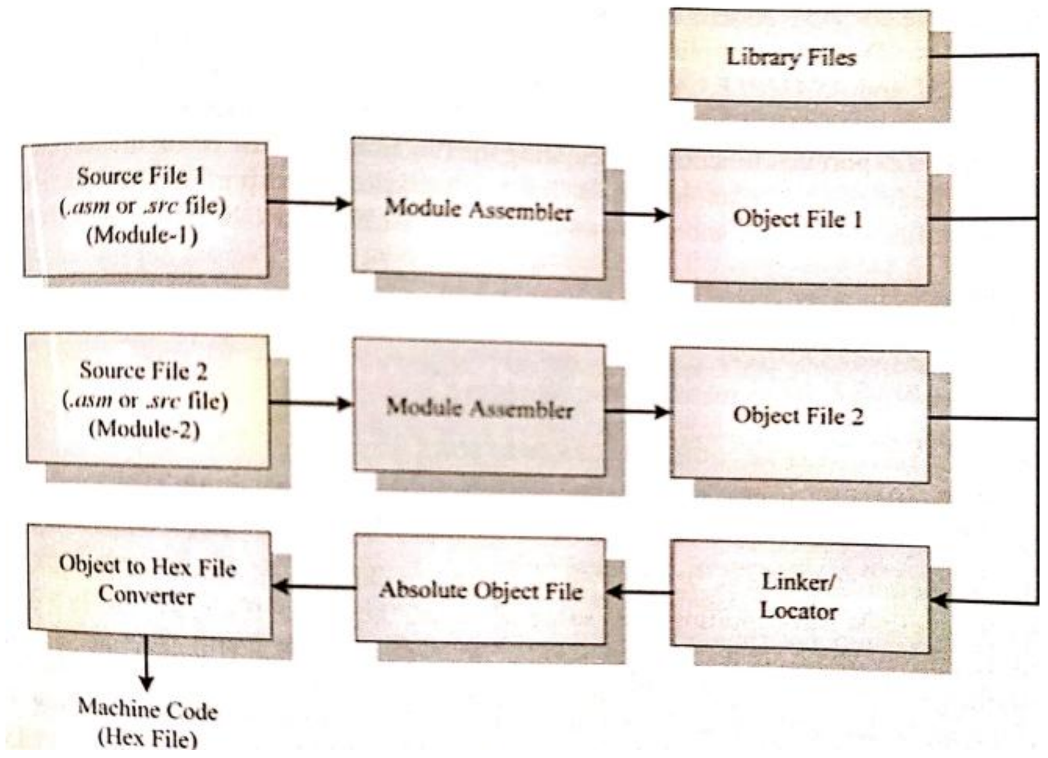
- The statement *ORG 0100H* in the above example is not an assembly language instruction; it is an *assembler directive* instruction. It tells the assembler that the Instructions from here onward should be placed at location starting from *0100H*. The Assembler directive instructions are known as '*pseudo- ops*'. They are used for –

  1. Determining the start address of the program (e.g. *ORG 0000H*)
  2. Determining the entry address of the program (e.g. *ORG 0100H*)
  3. Reserving memory for data variables, arrays and structures (e.g. *NUM1 EQU 70H*)
  4. Initializing variable values (e.g. *VALUE DATA 12H*)
     - The *EQU directive* is used for allocating memory to a variable and *DATA directive* is used for initializing a variable with data. No machine codes are generated for the 'pseudo-ops'.

- The Assembly language program written in assembly code is saved as *.asm* (Assembly file) file or *.src* (source) file. Any text editor like 'notepad' or 'WordPad' from Microsoft® or the text editor provide by an Integrated Development (IDE) tool can be used for writing the assembly instructions.
- When a program is too complex or too big; the entire code can be divided into sub-modules and each module can be re-usable (called as *Modular Programming*). Modular programs are usually easy to code, debug and alter.

Source File to Object File Translation: Translation of assembly code to machine code is performed by assembler. The assemblers for different target machines are different. Assemblers from multiple vendors are available in market. A51 Macro assembler from Keil software is a popular assembler for the 8051 family microcontroller.

- The various steps involved in the conversion of a program written in assembly language to corresponding binary file/ machine language is illustrated in the following Figure.



- Each *source module* is written in Assembly and is stored as .src file or .asm file. Each file can be assembled separately to examine the syntax errors and incorrect assembly instructions. On successful assembling of each .src/ .asm file a corresponding object file is created with extension '.obj'.
- The *object file* does not contain the absolute address of where the generated code needs to be placed on the program memory and hence it is called *re-locatable segment*. It can be placed at

any code memory location and it is the responsibility of the linker/ locater to assign absolute address for this module.

- *Absolute address allocation* is done at the absolute object file creation stage. Each module can share variables and subroutines (functions) among them.
- Exporting a variable/ function from a module (making a variable/ function from a module available to all other modules) is done by declaring that variable function as *PUBLIC* in the source module.
- Importing a variable or function from a module (taking a variable or function from any one of other modules) is done by declaring that variable or function as *EXTREN*) in the module where it is going to be accessed.

Library File Creation and Usage: Libraries are specially formatted, ordered program collections of object modules that may be used by the linker at a later time. When the linker processes a library, only those object modules in the library that are necessary to create the program are used. Library files are generated with extension '.*lib*'.

- Library is some kind of source code hiding technique. If you don't want to reveal the source code behind the various functions you have written in your program and at the same time you want them to be distributed to application developers for making use of them in their applications, you can supply them as library files and give them the details of the public functions available from the library (function name, function input/output, etc). For using a library file in a project, add the library to the project.
    - '*LIB51*' from K eil Software is an example for a library creator and it is used for creating library files for A51 Assembler/ C51 Compiler for 8051 specific controller.

Linker and Locater: *Linker and Locater* is another software utility responsible for "linking the various object modules in a multi-module project and assigning absolute address to each module".

- *Linker* is a program which combines the target program with the code of other programs (modules) and library routines.
- During the process of linking, the absolute object module is created. The *object module* contains the target code and information about other programs and library routines that are required to call during the program execution.
- An absolute object file or module does not contain any re-locatable code or data. All code and data reside at fixed memory locations. The absolute object file is used for creating hex files for dumping into the code memory of the processor/ controller.

- '*BL51*' from Keil Software is an example for a Linker & Locater for A51 Assembler/ C51 Compiler for 8051 specific controller.

Object to Hex File Converter: This is the final stage in the conversion of Assembly language (mnemonics) to machine understandable language (machine code).

- *Hex File* is the representation of the machine code and the hex file is dumped into the code memory of the processor/ controller.
- The hex file representation varies depending on the target processor/ controller make.
    - For Intel processors/ controllers the target hex file format will be '*Intel HEX*' and for Motorola, the hex file should be in '*Motorola HEX*' format.
- *HEX files* are ASCII files that contain a hexadecimal representation of target application. Hex file is created from the final 'Absolute Object File' using the Object to Hex File Converter utility.
- '*QH51*' from Keil software is an example for Object to Hex File Converter utility for A51 Assembler/ C51 Compiler for 8051 specific controller.

Advantages of Assembly Language Based Development: Assembly Language based development was (is) the most common technique adopted from the beginning of embedded technology development. Thorough understanding of the processor architecture memory organization, register sets and mnemonics is very essential for Assembly Language based development. The major advantages of Assembly Language based development listed below.

- ***Efficient Code Memory and Data Memory Usage (Memory Optimization):*** Since the developer is well versed with the target processor architecture and memory organization, optimized code can be written for performing operations. This lead less utilization of code memory and efficient utilization of data memory.
- ***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 a target application.
- ***Low Level Hardware Access:*** Most of the code for low level programming like accessing external device specific registers from the operating system kernel, device drivers, and low level interrupt routines, etc., are making use of direct assembly coding; since low level device specific operation support is not commonly available with most of the high-level language cross compilers.
- ***Code Reverse Engineering:*** *Reverse engineering* is the process of understanding the technology behind a product by extracting the information from a finished product. Reverse engineering is performed by 'hackers' to reveal the technology behind 'Proprietary Products'.

Drawbacks of Assembly Language Based Development: Every technology has its own pros and cons. From certain technology aspects assembly language development is the most efficient technique. But it is having the following technical limitations also.

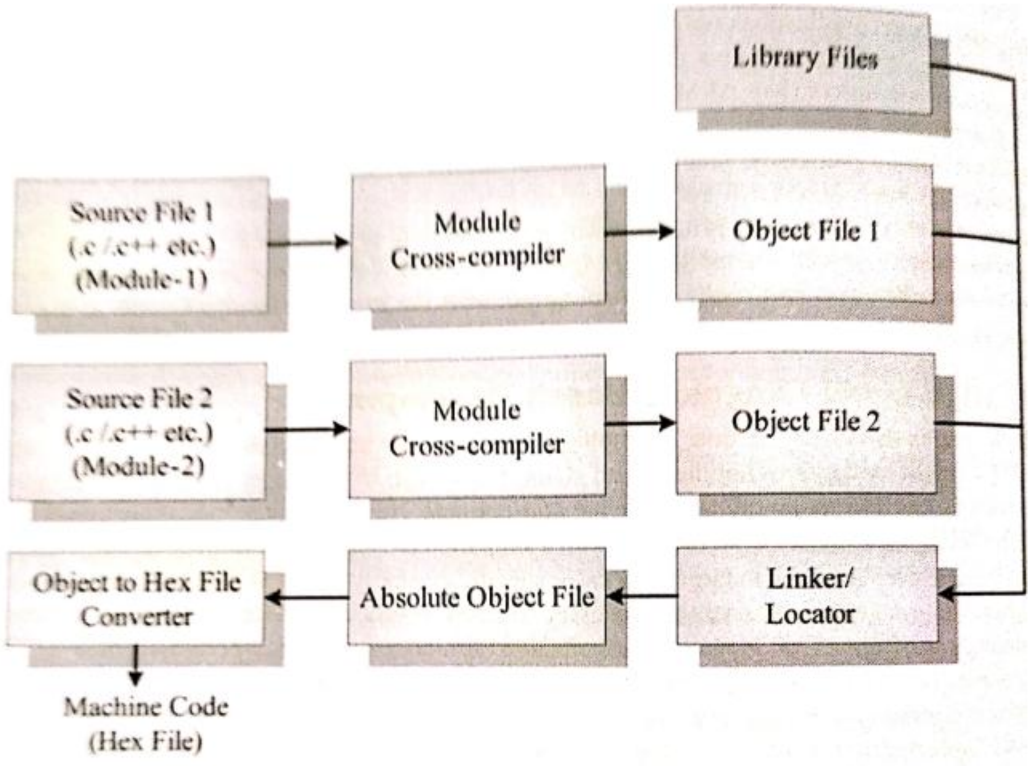
- ***High Development Time:*** Assembly language is much harder to program than high level languages. The developer must pay attention to more details and must have thorough knowledge of the architecture, memory organization and register details of the target processor in use. Learning the inner details of the processor and its assembly instructions is highly time consuming and it creates a delay impact in product development.
- ***Developer Dependency:*** There is no common written rule for developing assembly language based applications, whereas all high level languages instruct certain set of rules for application development. In assembly language programming, the developers will have the freedom to choose the different memory location and registers. Also the programming approach varies from developer to developer depending on his/ her taste.
  - For example moving data from a memory location to accumulator can be achieved through different approaches.
  - If the approach done by a developer is not documented properly at the development stage, he/ she may not be able to recollect why this approach is followed at a later stage or when a new developer is instructed to analyze this code, he/ she also may not be able to understand. Hence upgrading an assembly program on a later stage is very difficult.
- ***Non-Portable:*** Target applications written in assembly instructions are valid only for that particular family of processors (e.g. Application written for Inte x86 family of processors) and cannot be re-used for another target processors/ controllers (Say ARM Cortex M family of processors). If the target processor/ controller changes, a complete re-writing of the application using the assembly instructions for the new target processor/ controller is required.

High Level Language Based Development:

Assembly language based programming is highly time consuming, tedious and requires skilled programmers with sound knowledge of the target processor architecture. Also applications developed in Assembly language are non-portable.

- Any high level language (like C, C++ or Java) with a supported cross compiler (for converting the application developed in high level language to target processor specific assembly code) for the target processor can be used for embedded firmware development.
- The most commonly used high level language for embedded firmware application development is 'C'.

- '*C*' is the well defined, easy to use high level language with extensive cross platform development tool support. Nowadays Cross-compilers for C++ is also emerging out and embedded developers are making use of C++ for embedded application development.
- The various steps involved in high level language based embedded firmware development is same as that of assembly language based development, except that the conversion of source file written in high level language to object file is done by a cross-compiler, whereas in Assembly language based development, it is carried out by an assembler.
- The various steps involved in the conversion of a program written in high level language to corresponding binary file/ machine language is illustrated in the following Figure.



- The program written in any of the high level language is saved with the corresponding language extension (.c for C, .cpp for C++ etc). Any text editor like 'notepad' or 'WordPad ' from Microsoft® or the text editor provided by an Integrated Development (IDE) tool supporting the high level language can be used for writing the program.
- Most of the high level languages support modular programming approach and hence you can have multiple source files called modules written in corresponding high level language.
- Translation of high level source code to executable object code is done by a cross-compiler. The cross-compilers for different high level languages for the same target processor are different.
- C51 is a popular. Cross-compiler available for '*C*' language for the 8051 family of micro controller. Conversion of each module's source code to corresponding object file is performed by the cross compiler.

- Rest of the steps involved in the conversion of high level language to target processor's machine code are same as that of the steps involved in assembly language based development.

Advantages of High Level Language Based Development:

- ***Reduced Development Time:*** Developer requires less or little knowledge on the internal hardware details and architecture of the target processor/ controller. Bare minimal knowledge of the memory organization and register details of the target processor in use and syntax of the high level language are the only pre-requisites for high level language based firmware development.
    - All other things will be taken care of by the cross-compiler used for the high level language. Thus the ramp up time required by the developer in understanding the target hardware and target machine's assembly instructions is waived off by the cross compiler and it reduces the development time by significant reduction in developer effort.
    - High level language based development also refines the scope of embedded firmware development from a team of specialized architects to anyone knowing the syntax of the language and willing to put little effort on understanding the minimal hardware details.
    - With high level language, each task can be accomplished by lesser number of lines of code compared to the target processor/ controller specific Assembly language based development.
- ***Developer Independency:*** The syntax used by most of the high level languages are universal and a program written in the high level language can easily be understood by a second person knowing the syntax of the language.
    - High level languages always instruct certain set of rules for writing the code and commenting the piece of cadet lf the developer strictly adheres to the rules, the firmware will be 100% developer independent.
- ***Portability:*** Target applications written in high level languages are converted to target processor / controller understandable format (machine codes) by cross-compiler.
- An application written in high level language for a particular target processor can easily be converted to another target processor/ controller specific application, with little or less effort by simply re-compiling/ little code modification followed by re-compiling the application for the required processor/ controller.

Limitations of High Level Language Based Development:

- Some cross-compilers available for high level languages may not be so efficient in generating 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 (Integrated Development Environment incorporating cross-compiler) is high compared to Assembly Language based firmware development tools.

Mixing Assembly and High Level language:

Certain embedded firmware development situations may demand the mixing of high level language with Assembly and vice versa. High level language and assembly languages are usually mixed in three ways; namely, *mixing Assembly Language with High Level Language*, *mixing High Level Language with Assembly* and *In-line Assembly programming*.

Mixing Assembly with High Level Language (e.g. Assembly Language with 'C'): Assembly routines are mixed with '*C*' in situations where the entire program is written in '*C*' and the cross-compiler do not have a built in support for implementing certain features like Interrupt Service Routine functions (ISR) or if the programmer wants to take advantage of the speed and optimized code offered by machine code generated by hand written assembly rather than cross compiler generated machine code.

When accessing certain low level hardware, the timing specifications may be very critical and a cross-compiler generated binary may not be able to offer the required time specifications accurately. Writing the hardware/ peripheral access routine in processor/ controller specific Assembly language and invoking it from '*C*' is the most advised method to handle such situations.

Mixing '*C*' and Assembly is little complicated; in the sense-the programmer must be aware of how parameters are passed from the '*C*' routine to Assembly and values a returned from assembly routine to '*C*' and how 'Assembly routine' is invoked from the '*C*' code.

The following steps give an idea how *C51* cross-compiler performs the mixing of Assembly code with '*C*':

1. Write a simple function in *C* that passes parameters and returns values the way you want your assembly routine to.
2. Use the SRC directive (*#PRAGMA SRC* at the top of the file) so that the *C* compiler generates an *.SRC* file instead of an *.OBJ* file.
3. Compile the *C* file. Since the *SRC* directive is specified, the *.SRC* file is generated. The *.SRC* file contains the assembly code generated for the *C* code you wrote.
4. Rename the *.SRC* file to *.A51* file.
5. Edit the *.A51* file and insert the assembly code you want to execute in the body of the assembly function shell included in the *.A51* file.

Mixing High level language with Assembly (e.g. 'C' with Assembly Language): Mixing the code written in a high level language like '*C*' and Assembly language is useful in the following scenarios:

1. The source code is already available in Assembly language and a routine written in a high level language like '*C*' needs to be included to the existing code.
2. The entire source code is planned in Assembly code for various reasons like optimized code, optimal performance, efficient code memory utilization and proven expertise in handling the Assembly, etc. But some portions of the code may be very difficult and tedious to code in Assembly. For example 16-bit multiplication and division in 8051 Assembly Language.
3. To include built in library functions written in '*C*' language provided by the cross compiler. For example: Built in Graphics library functions and String operations supported by '*C*'.

Inline Assembly: *Inline assembly* is another technique for inserting target processor/ controller specific Assembly instructions at any location of a source code written in high level language '*C*'. This avoids the delay in calling an assembly routine from a '*C*' code. Special keywords are used to indicate the start and end of Assembly instructions. *C51* uses the keywords *#pragma asm* and *#pragma endasm* to indicate a block of code written in assembly.

'C' versus '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. Hence it is a platform specific development.

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. It should be noted that 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 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 (machine language).

C	Embedded C
C is a general purpose programming language, which can be used to design any type of desktop-based applications.	Embedded C is an extension of C language and used to develop microcontroller-based applications.
C language program is hardware independent.	Embedded C program is hardware dependent.
C language uses standard compilers to compile and execute the program.	Embedded C requires specific compilers that are able to generate particular hardware/ microcontroller-based output.
Readability, modifications, bug fixing, etc., are very easy in a C language program.	Readability, modifications, bug fixing, etc., are not easy in a Embedded C language program.

Compiler versus 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 specific target processor architecture (e.g. Intel x86/ Pentium). Here the operating system, the complier 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. The development is platform specific (OS as well as target processor on which the OS is running). Compilers are generally termed as '*Native Compilers*'. A native compiler generates machine code for the same machine (processor) on which it is running.

Cross-compilers are 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*, *ARM*, etc.). Keil *C51* is an example for cross-compiler.

NOTE: The term 'Compiler' is used interchangeably with 'Cross-compiler' in embedded firmware applications. Whenever you see the term 'Compiler' related to any embedded firmware application, please understand that it is referring the cross-compiler.

By: **DR. MAHESH PRASANNA K.,**

DEPT. OF CSE, VCET.

****

****