Module-2

Embedded System Design Concepts

Topics:

1. Characteristics and Quality Attributes of Embedded Systems
2. Operational and non-operational quality attributes
3. Embedded Systems-Application and Domain Specific
4. Hardware Software Co-Design and Program Modeling
5. Embedded firmware design and development

2.1 CHARACTERISTICS OF AN EMBEDDED SYSTEM

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 as follows:
(1) Application and domain specifi c
(2) Reactive and Real Time
(3) Operates in harsh environments
(4) Distributed
(5) Small size and weight
(6) Power concerns

Application and Domain Specific

- Embedded system is designed to perform a set of defined functions and they are developed to do the intended functions only.
- They cannot be used for any other purpose. It is the major criterion which distinguishes an embedded system from a general purpose computing system.
- For example, you cannot replace the embedded control unit of your microwave oven with your air conditioner's 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 systems 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 behaviour of the system should be deterministic; meaning the system should respond to requests or tasks in a known 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 such 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 ATM card, transaction unit for performing transactions, a currency counter for dispatching/vending currency to the authorised person and a printer unit for printing the transaction details.
- We can visualise 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 sized 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 minimise 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 the 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 application. The more the power consumption the less is the battery life.

2.2 QUALITY ATTRIBUTES OF EMBEDDED SYSTEMS

Quality attributes are the 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'.***

2.2.1 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
(2) Throughput
(3) Reliability
(4) Maintainability
(5) Security
(6) Safety

Response: Response is a measure of quickness of the system. It gives you 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 impact 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 within this particular timeline.

Throughput: Throughput deals with the efficiency of a system. In general it can be 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 the 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.

Reliability: 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 the order of minutes.

Maintainability: 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 also 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 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. Hence it is obvious that maintainability is simply an indication of the availability of the product for use. In any embedded system design, the ideal value for availability is expressed as

where Ai = Availability in the ideal condition, MTBF = Mean Time Between Failures, and MTTR = Mean Time To Repair

Security: 'Confidentiality', ' Integrity', and ' Availability' (The term 'Availability' mentioned here is not related to the term 'Availability' mentioned under the 'Maintainability' section) are the three major measures of information security.

Confidentiality deals with the protection of data and application from unauthorised disclosure. Integrity deals with the protection of data and application from unauthorized modification. Availability deals with protection of data and application from unauthorised 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 a user name and password to access into a particular person's profile–This is 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 levels 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.

Safety: 'Safety' and 'Security' are two 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 or due to 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.

As stated before, some of the safety threats are sudden (like product breakdown) and some of them are gradual (like hazardous emissions from the product).

2.2.2 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
(2) Evolvability
(3) Portability
(4) Time to prototype and market
(5) Per unit and total cost.

Testability & Debug-ability: Testability deals with how easily one can test the 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 behaviour 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.

Evolvability: 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.

Portability: 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 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 re-work 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 (e.g. Intel x86) to a different target processor (say an ARM Cortex M3 processor from Freescale). 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 for 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 very 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 may not function on other operating systems; whereas applications developed using 'Java' from Sun Microsystems works on any operating system that supports Java standards.

Time-to-Prototype and Market: Time-to-market is the time elapsed between the conceptualisation 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.

Per Unit Cost and Revenue: 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 at 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

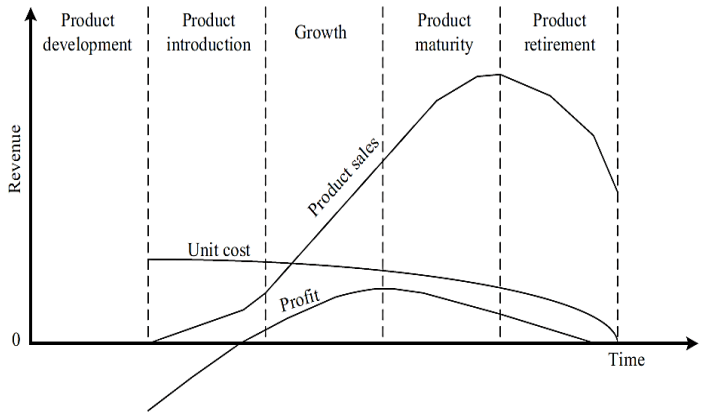

Fig. Product life cycle (PLC) curve

From the graph, it is clear that the total revenue increases from the product introduction stage to the product maturity stage. The revenue peaks at the maturity stage and starts falling in the decline/retirement stage. The unit cost is very high during the introductory stage (a typical example is cell phone; if you buy a new model of cell phone during its launch time, the price will be high and you will get the same model with a very reduced price after three or four months of its launching). The profi t increases with increase in sales and attains a steady value and then falls

with a dip in sales. You can see a negative value for profit during the initial period. It is because during the product development phase there is only investment and no returns. Profit occurs only when the total returns exceed the investment and operating cost.

2.3 WASHING MACHINE—APPLICATION-SPECIFIC EMBEDDED SYSTEM

People experience the power of embedded systems and enjoy the features and comfort provided by them, but they are totally unaware or ignorant of the intelligent embedded players working behind the products providing enhanced features and comfort.

Washing machine is a typical example of an embedded 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. You can see all these components in a washing machine if you have a closer look at it. Some of them are visible and some of them may be invisible to you.

The actuator part of the washing machine consists of a motorised 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 microprocessor/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 a washing machine is shown in Fig. below

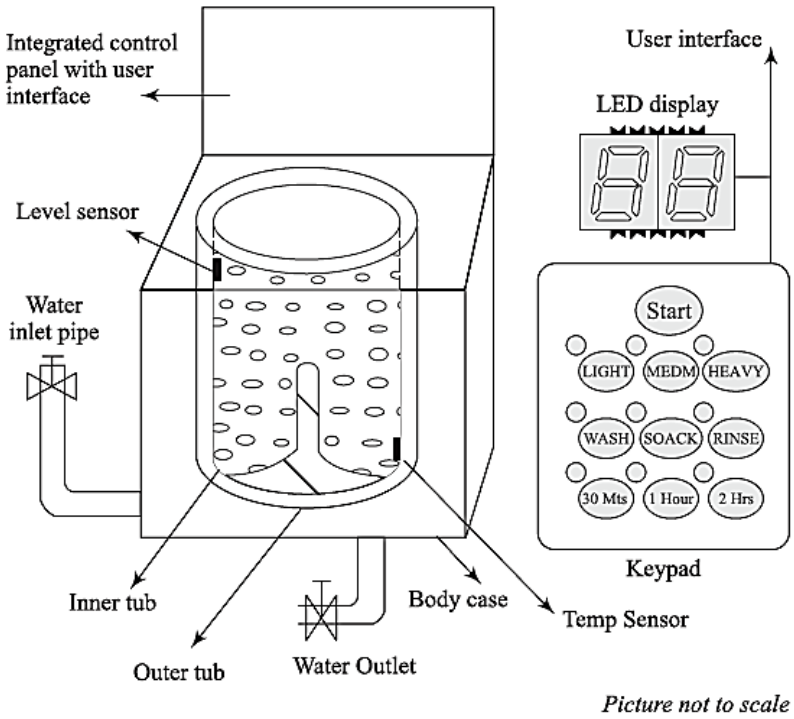

Fig. Washing machine – Functional block diagram

Washing machine comes in different designs, like 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 namely* 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.

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 namely* Wash, Spin and Rinse, cloth type selector namely* 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. It is to be noted that this interface may vary from manufacturer to manufacturer and model to model.

The other types of I/O interfaces which are invisible to the end user are different kinds of sensor interfaces, namely, water temperature sensor, water level sensor, etc. and actuator interface including motor control for agitator and tub movement control, inlet water flow control, etc.

2.4 AUTOMOTIVE–DOMAIN SPECIFIC EXAMPLES OF EMBEDDED SYSTEM

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

Inner Workings of Automotive Embedded Systems

Automotive embedded systems are the one where electronics take control over the mechanical and electrical systems. The presence of automotive embedded system in a vehicle varies from simple mirror and wiper controls to complex air bag controller and Anti-lock 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 vehicle like Mercedes S and BMW 7 may contain over 100 embedded controllers. Government regulations on fuel economy, environmental factors and emission standards and increasing customer demands on safety, comfort and infotainment forces the automotive manufactures to opt for sophisticated embedded control units within the vehicle. The first embedded system used in automotive application was the microprocessor based fuel injection system introduced by Volkswagen 1600 in 1968.

Fig. Embedded system in the automotive domain

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

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) High-speed electronic control units (HECUs) are deployed in critical control units requiring fast response. They include fuel injection systems, antilock brake systems, engine control, electronic throttle, steering controls, transmission control unit and central control unit.

Low-speed Electronic Control Units (LECUs) Low-Speed Electronic Control Units (LECUs) are deployed in applications where response time is not so critical. They generally are built around low cost microprocessors/microcontrollers and digital signal processors. Audio controllers, passenger and driver door locks, door glass controls (power windows), wiper control, mirror control, seat control systems, head lamp and tail lamp controls, sun roof control unit, etc. are examples of LECUs.

2.5 Automotive Communication Buses

Automotive applications make use of serial buses for communication, which greatly reduces the amount of wiring required inside a vehicle. The following section will give you an overview of the different types of serial interface buses deployed in automotive embedded applications.

Controller Area Network (CAN) The CAN bus was originally proposed by Robert Bosch, pioneer in the Automotive embedded solution providers. It supports medium speed (ISO11519-class B with data rates up to 125 Kbps) and high speed (ISO11898 class C with data rates up to 1Mbps) data transfer. 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. The protocol format and interface application development for CAN bus will be explained in detail in another volume of this book series.

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.

Media Oriented System Transport (MOST) Bus The Media Oriented System Transport (MOST) is targeted for high-bandwidth automotive multimedia networking (e.g. audio/video, infotainment system 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 defi ne 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.

2.6 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 a variety of models from the requirements specification to the implementation aspect of the system design. The reason being, the objective varies with each phase; for example at the specification stage, only the functionality of the system is in focus and not the implementation information. When the design moves to the implementation aspect, the information about the system components is revealed and the designer has to switch to a model capable of capturing the system's structure. We will discuss about the different models in a later section of this chapter.

Selecting the Architecture A model only captures the system characteristics and does not provide information on 'how the system can be manufactured?'. The architecture specifi es 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 Specifi c 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 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 predefi ned 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. Most of the time the arithmetic units are connected in parallel with pipelining support for bringing high performance

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. Normally the datapath is implemented in a chip and the I/O pins of the chip acts as the data input output ports for the chip resident data path.

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 (e.g. Reading a register value and comparing it with a given value and then transfer the program execution to a new address location (The CJNE instruction for 8051 ISA)) 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. On the other hand, Reduced Instruction Set Computing (RISC) architecture uses 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 fi le 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 Elements. The scheduling of the instruction execution and controlling of each PE is performed through a single controller. The SIMD architecture forms the basis of re-configurable processor (We will discuss about re-configurable processors in a later chapter). 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 this language should be used for capturing this model. A model can be captured using multiple programming languages like C, C++, C#, Java, etc. for software implementations and languages like VHDL, System C, Verilog, etc. for hardware implementations. On the other hand, a single language can be used for capturing a variety of

models. Certain languages are good in capturing certain computational model. For example, C++ is a good candidate for capturing an object oriented model. The only pre-requisite in selecting a programming language for capturing a model is that the language should capture the model easily.

2.7 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. The following sections give an overview of these models.

2.7.1 Data Flow Graph/Diagram (DFG) Model

The Data Flow Graph (DFG) model translates the data processing requirements into a data flow graph. The Data Flow Graph (DFG) model is a data driven model in which the program execution is determined by data.

This model emphasises on the data and operations on the data which transforms the input data to output data. Indeed Data Flow Graph (DFG) is a visual model in which the operation on the data (process) is represented using a block (circle) and data flow is represented using arrows. An inward arrow to the process (circle) represents input data and an outward arrow from the process (circle) represents output data in DFG notation.

Embedded applications which are computational intensive and data driven are modeled using the DFG model. DSP applications are typical examples for it.

Now let's have a look at the implementation of a DFG. Suppose one of the functions in our application contains the computational requirement x = a + b; and y = x – c.

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.

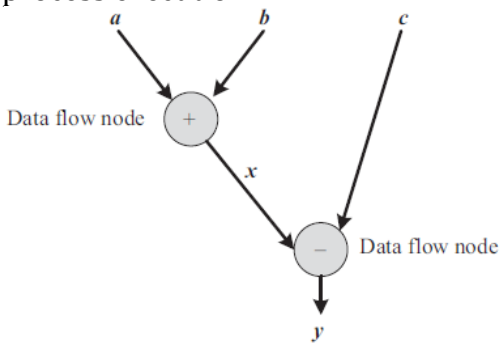

Fig. Data flow graph (DFG) model

2.7.2 Control Data Flow Graph/ Diagram (CDFG)

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

nodes, whereas DFG contains only data flow nodes. Let us have a look at the implementation of the CDFG for the following requirement.
If fl ag = 1, x = a + b; else y = a – b; This requirement contains a decision making process. The CDFG model for the same is given in Fig. below

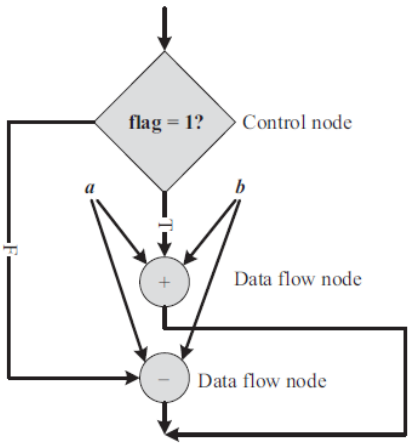

Fig. Control Data Flow Graph (CDFG) Model

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 modelling 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.

2.7.3 State Machine Model

The State Machine model is used for modelling reactive or event-driven embedded systems whose processing behaviour are dependent on state transitions. Embedded systems used in the control and industrial applications are typical examples for event driven systems. The State Machine model describes the system behaviour with 'States', 'Events', 'Actions' and 'Transitions'. State is a representation of a current situation. An event is an input to the state. The event acts as stimuli for state transition. Transition is the movement from one state to another. Action is an activity to be performed by the state machine.
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.

1. 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.
2. The Alarm is turned off when the alarm time (5 seconds) expires or if the driver/passenger fastens the belt or if the ignition switch is turned off, whichever happens first.

Here the states are 'Alarm Off', 'Waiting' 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 Fig. below.

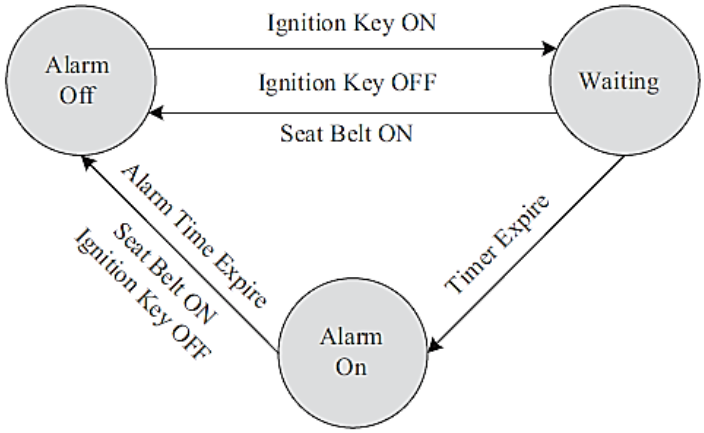

Fig. FSM Model for Automatic seat belt warning system

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 fi rst. 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 Fig. below.

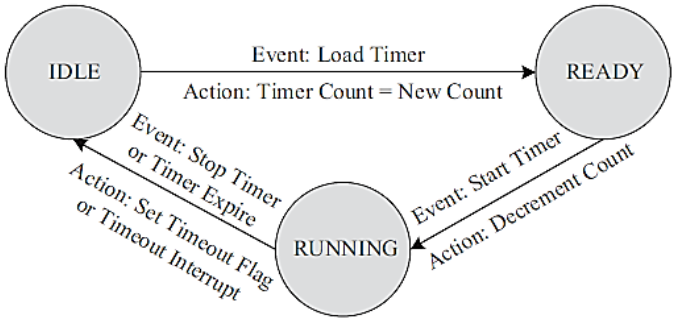

Fig. FSM Model for timer

As seen from the FSM, the timer state can be either 'IDLE' or 'READY' or 'RUNNING'. During thenormal 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 Fig.

In its simplest representation, it 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' (Tea and Coffee are the drink select button).
- If the event triggered in 'Wait State' is 'Cancel' button press, the coin is pushed out and the state transitions to 'Wait for Coin'.
- If the event received in the 'Wait State' is either 'Tea' button press, or 'Coffee' button press, the state changes to 'Dispense Tea' and 'Dispense Coffee' respectively.
- Once the coffee/tea vending is over, the respective states transitions back to the 'Wait for Coin' state.

A few modifications like adding a timeout for the 'Wait State' (Currently the 'Wait State' is infinite; it can be re-designed to a timeout based 'Wait State'. If no user input is received within the timeout period, the coin is returned back and the state automatically transitions to 'Wait for Coin' on the timeout event) and capturing another events like, 'Water not available', 'Tea/Coffee Mix not available' and changing the state to an 'Error State' can be added to enhance this design. It is left to the readers as exercise.

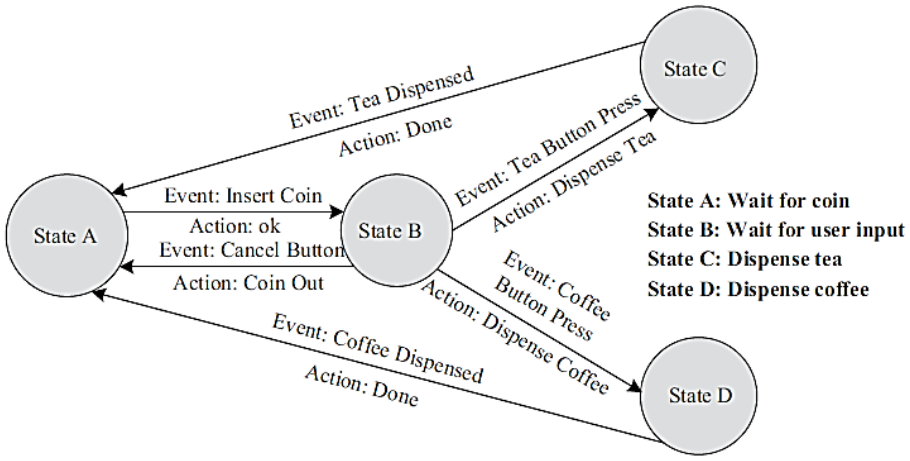

Fig FSM Model for Automatic Tea\Coffee Vending Machine

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 Fig , is a simple representation and it doesn't take care of scenarios like, user doesn't insert a coin within the specified time after lifting the receiver, user inserts coins other than a one rupee etc. Handling these scenarios is left to the readers as exercise.
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.

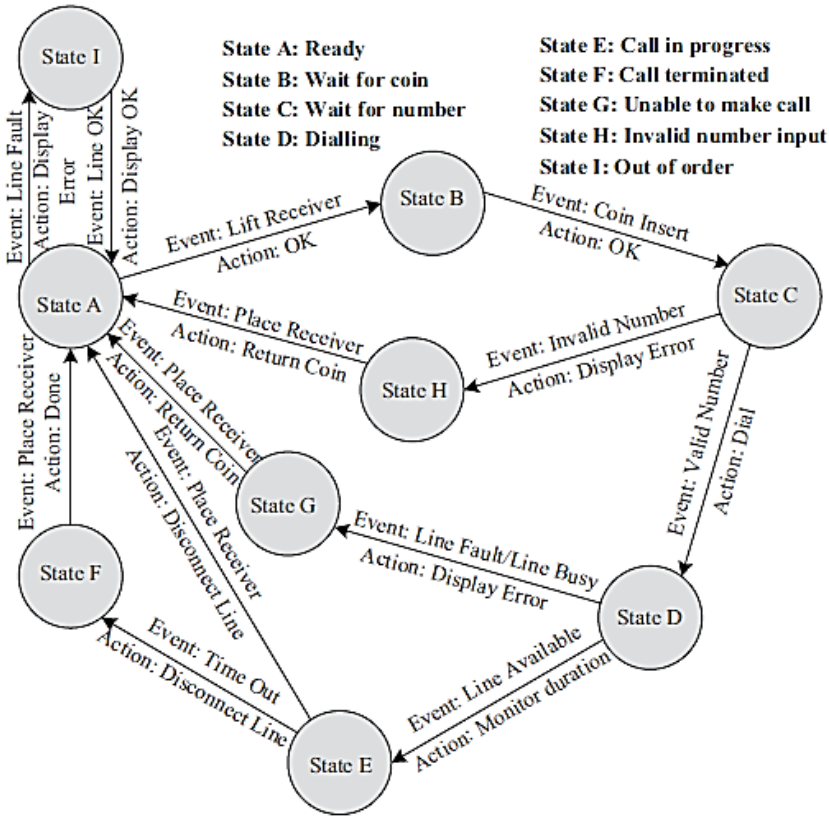

Fig. FSM Model for Coin Operated Telephone System

The Harel Statechart, UML State diagram, etc. are examples for popular state charts used for the HCFSM modelling of embedded systems. In state charts, the state is usually represented using geometric shapes like rounded rectangle, rectangle, ellipse, circle, etc. The Harel Statechart uses a rounded rectangle for representing state. Arrows are used for representing the state transition and they are marked with the event associated with the state transition. Sometimes an optional parenthesised condition is also labelled with the arrow. The condition specifies on what basis the state transition happens at the occurrence of the specified event. Lots of design tools are available for state machine and statechart based system modelling.

2.7.4 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 modelling. Another important tool used for modelling 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.

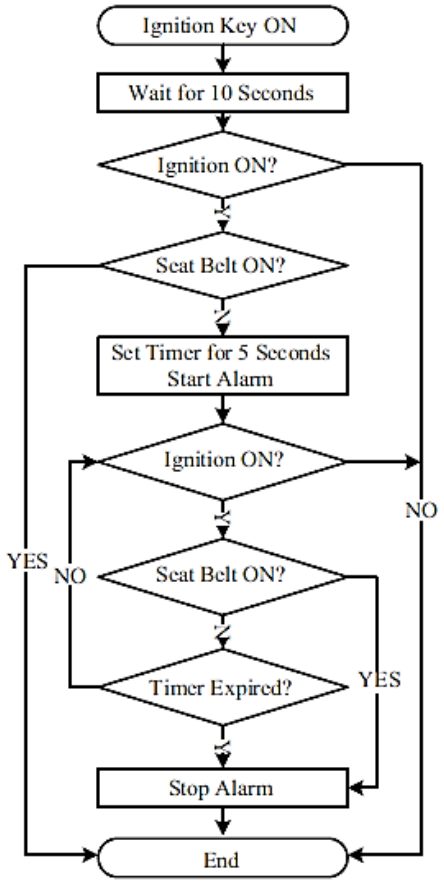

Fig. Sequential Program Model for seat belt warning system

```
#define ON 1
#define OFF 0
#define YES 1
#define NO 0
void seat_belt_warn()
{
wait_10sec();
if (check_ignition_key()==ON)
{
if (check_seat_belt()==OFF)
{
set_timer(5);
```

```
start_alarm();
while ((check_seat_belt()==OFF )&&(check_ignition_key()==OFF )&&
(timer_expire()==NO));
stop_alarm();
}
}
}
```

Figure illustrates the flow chart approach for modelling the 'Seat Belt Warning' system explained in the FSM modelling section.

2.7.5 Concurrent/Communicating Process Model

The concurrent or communicating process model models concurrently executing tasks/processes. It is easier to implement certain requirements in concurrent processing model than the conventional sequential execution. Sequential execution leads to a single sequential execution of task and thereby leads to poor processor utilisation, when the task involves I/O waiting, sleeping for specifi ed 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 synchronisation 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 synchronise their execution through some mechanism. We need to start the alarm only after the expiration of the 10seconds 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 seatbelt is in the OFF state. Here we will use events to indicate these scenarios. The wait_timer_expire event is associated with the timer task event and it will be in the reset state initially and it is set when the timer expires.

Similarly, events ignition_on and ignition_off are associated with the task ignition key status monitoring and the events seat_belt_on and seat_belt_off are associated with the task seat belt status morning. The events ignition_off and ignition_on are set and reset respectively when the ignition key status is OFF and reset and set respectively when the ignition key status is ON, by the ignition key status monitoring task. Similarly the events seat_belt_off and seat_belt_on are set and reset respectively when the seat belt status is OFF and reset and set respectively when the seat belt status is ON, by the seat belt status monitoring task. The events alarm_timer_start and alarm_timer_expire are associated with the alarm timer task. The alarm_timer_start event will be in the reset state initially and it is set by the alarm control task when the alarm is started. Thealarm_timer_expire event will be in the reset state initially and it is set when the alarm timer expires. Thealarm control task waits for the signaling of the event wait_timer_expire and starts the alarm timer and alarm if both the events ignition_on and seat_belt_off are in the set state when the event wait_timer_expiresignals. If not the alarm control task simply completes its execution and returns. In case the alarm is started, the alarm control task waits for the signalling of any one of the

events alarm_timer_expire or ignition_off or seat_belt_on. Upon signalling any one of these events, the alarm is stopped and the alarm control task simply completes its execution and returns. Figure illustrates the same.

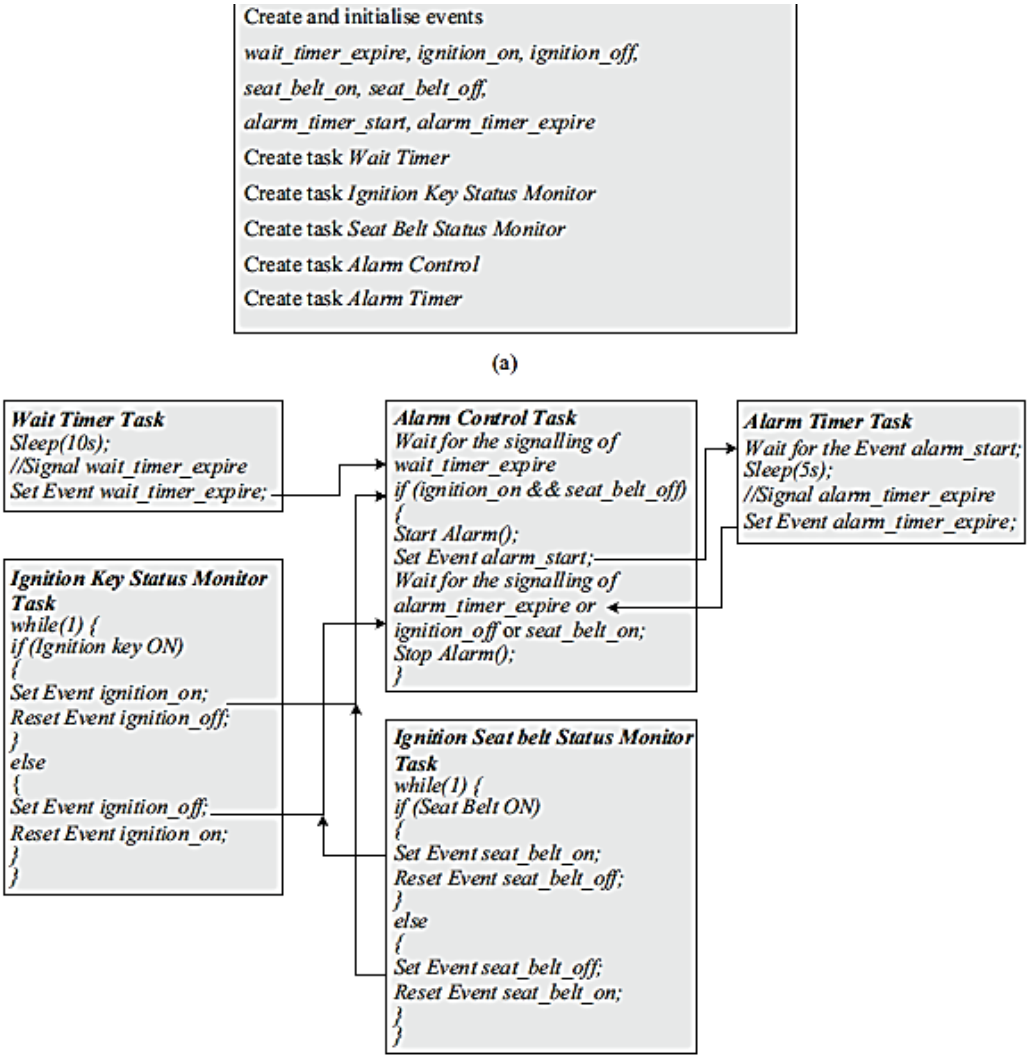

Fig. (a) Tasks for 'Seat Belt Warning System' (b) Concurrent processing Program model for 'Seat Belt Warning System'

It should be noted that the method explained here is just one way of implementing a concurrent model for the 'Seat Belt Warning' system. The intention is just to make the readers familiar with the concept of multi tasking and task communication/synchronisation. There may be other ways to model the same requirements.

It is left to the readers as exercise. The concurrent processing model is commonly used for the modelling of 'Real Time' systems. Various techniques like 'Shared memory', 'Message Passing', 'Events', etc. are used for communication and synchronising between concurrently executing processes.

2.7.6 Object-Oriented Model

The object-oriented model is an object based model for modelling system requirements. It disseminates a complex software requirement into simple well defined pieces called objects. Object-oriented model brings re-usability, maintainability and productivity in system design.

In the object-oriented modelling, object is an entity used for representing or modelling a particular piece of the system. Each object is characterised by a set of unique behaviour and state. A class is an abstract description of a set of objects and it can be considered as a 'blueprint' of an object.

A class represents the state of an object through member variables and object behaviour through member functions. The member variables and member functions of a class can be private, public or protected. Private member variables and functions are accessible only within the class, whereas public variables and functions are accessible within the class as well as outside the class. The protected variables and functions are protected from external access.

However classes derived from a parent class can also access the protected member functions and variables. The concept of object and class brings abstraction, hiding and protection.